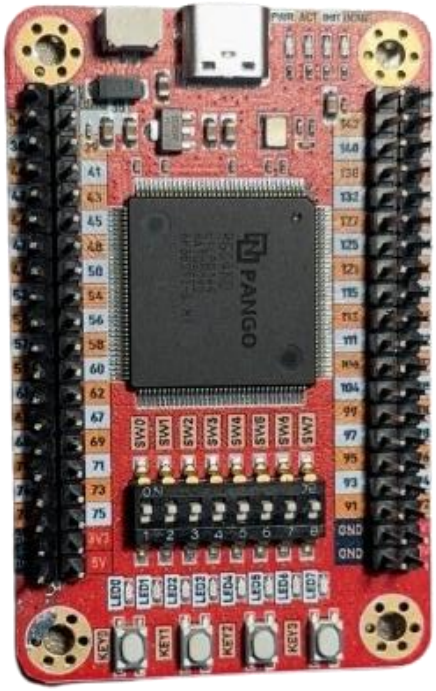
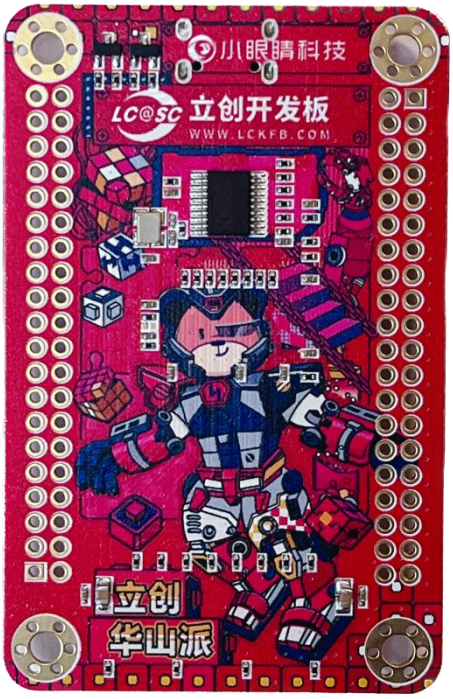




逻辑派 Z1 开发板文档教程

目 录

图 目 录

表 目 录

第一部分、特性介绍

1、开发板硬件参数

逻辑派 Z1 开发板是一套基于紫光同创 Compact 系列 PGC4KD-6ILPG144 芯片为核心的开发套件，板载 JTAG 调试接口，预留两组 40PIN 扩展 IO，配备按键、led 灯、拨码开关等硬件资源，为用户提供基本的开发环境。其芯片内部有一块嵌入式 Flash 芯片，可以实现位流从芯片内部自主加载，实现掉电保存程序，所以其启动方式为 Flash 启动。

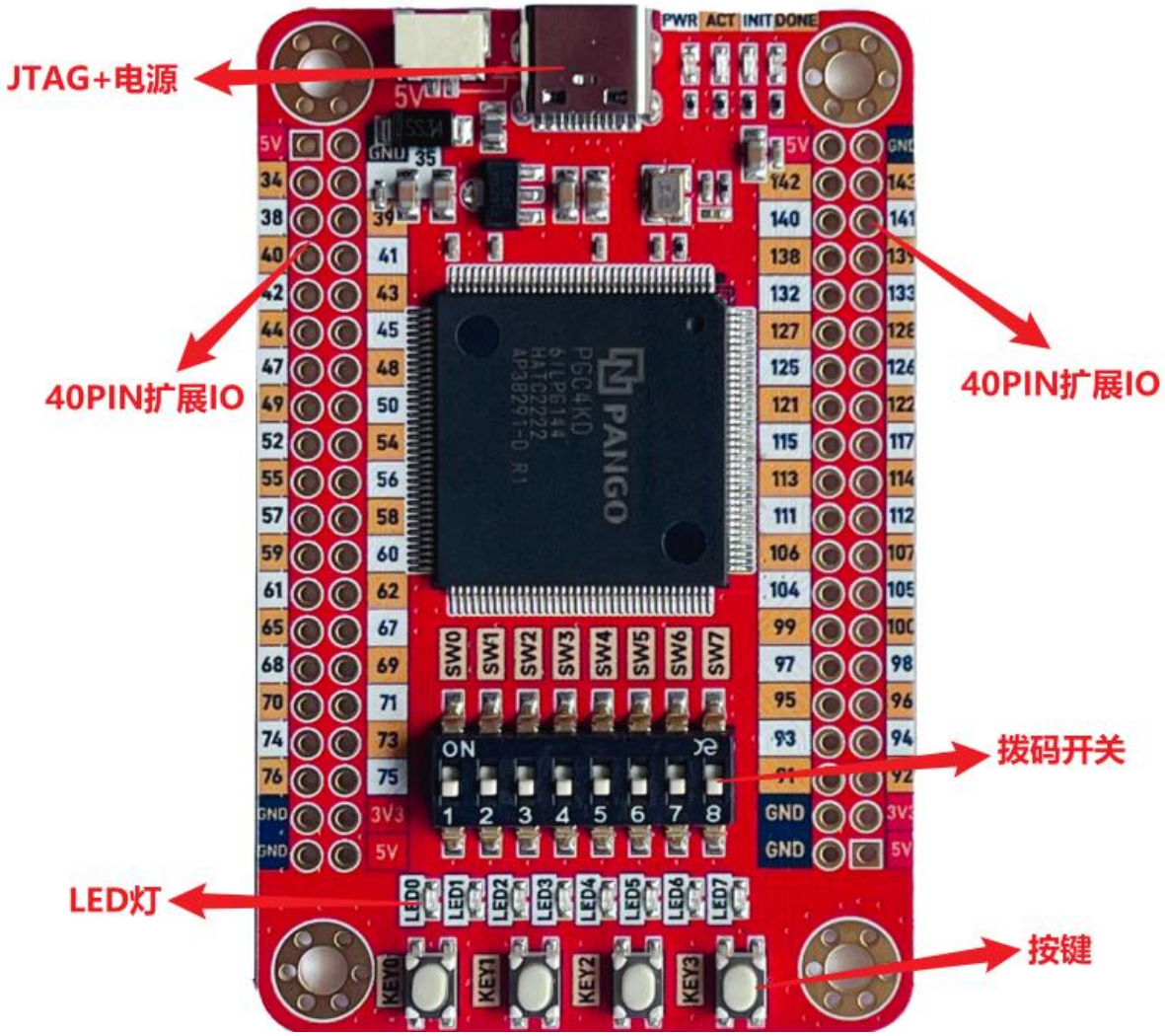


图 1-1-1.1-1 逻辑派 Z1 开发板正面展示图

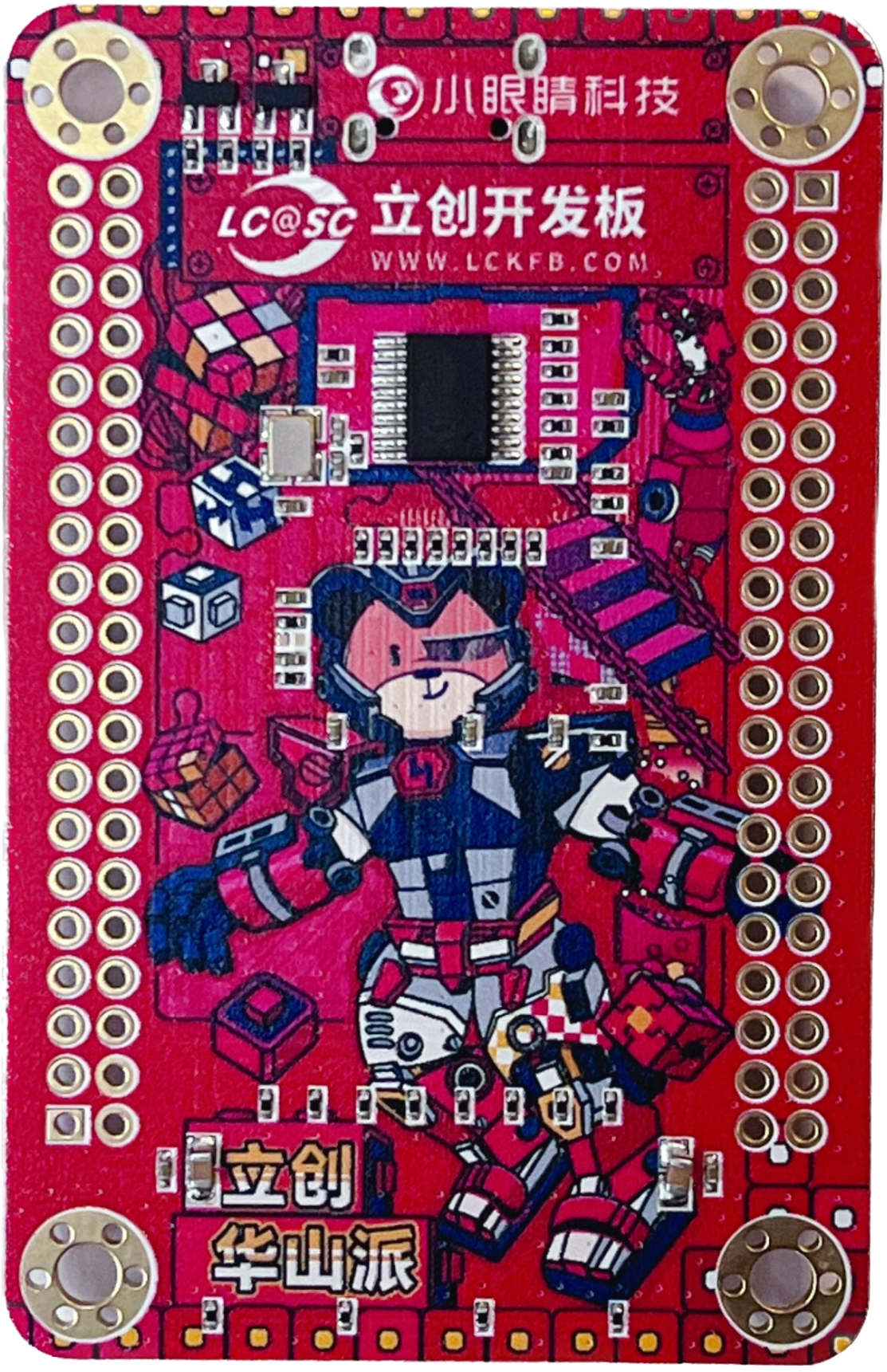


图 1-1-1.1-2 逻辑派 Z1 开发板背面展示图

该板卡资源如下：

表 1-1-1 板卡资源表

逻辑派 Z1 资源介绍		
逻辑资源	等效 LUT4	4761
	Flip-Flop(个)	3036
RAM 资源	分布式 RAM(Kbit)	39
	内嵌 9K 块 RAM(个)	11
	块存储器(Kbit)	99
时钟资源	锁相环 PLL 和 Global Clock	2/16
时钟资源	IO Banks	6
	最大用户 IO	280
	最大差分对	18
硬核资源	I2C 硬核	2
	SPI 硬核	1
	定时器/计数器/硬核	1
	片上振荡器	1
	上电启动时间(ms)	4.67

2、开发板外设资源

表 1-2-1 外设资源表

板载资源	
8 位按键	流水灯实验，键控 LED 实验
8 位按键	键控 LED 实验
Type-C/Jtag 接口	键控 LED 实验
Type-C/Jtag 接口	即作为电源接口也作为下载接口和串口通讯接口
50MHZ	为 CPLD 提供工作时钟
40PIN 扩展 IOX2	可接入外部模块
外接模块资源	
VGA 模块、分频器模块、蜂鸣器模块、音频输出输出模块、PMOD 模块、矩阵按键模块、ADC 模块、OLED 模块	

第二部分、开发环境搭建

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。注意，正常的版本需要 License 才可以使用，我们也会提供不需要 License 的版本，即 Lite 版，该版本不需要 License，安装后即可使用。

2、软件的安装

先通过百度网盘下载我们提供的资料包， 之后双击安装包中的安装程序 Setup.exe，启动安装程序。

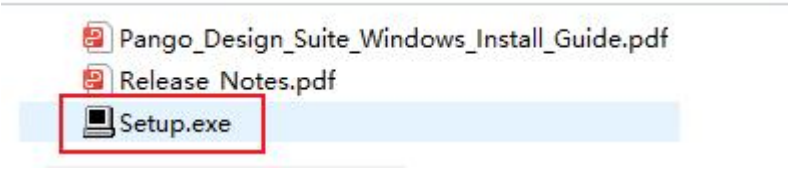


图 2-1.1-1 软件安装流程图 1

（1） 启动安装程序后的界面如下：

图 2-1.1-2 软件安装流程图 2

（2） 点击“Next”，跳转至许可协议对话框：

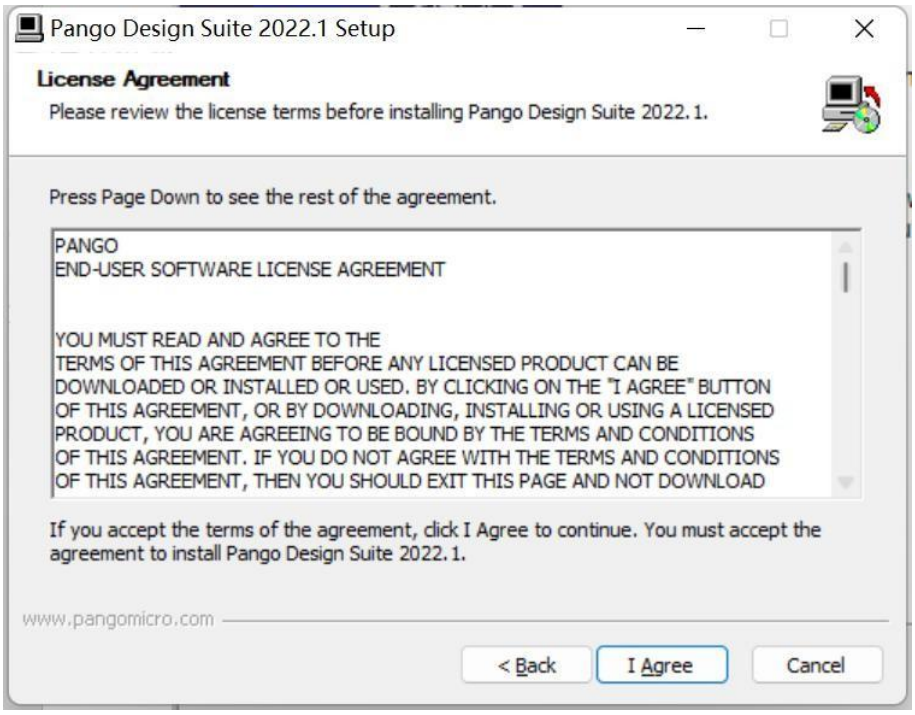


图 2-1.1-3 软件安装流程图 3

（3） 选择接受许可协议，点击“I Agree”按钮，进入选择安装路径选择框，如下图所示，默认安装路径为 C:\pango\PDS_2022.1，建议更换到别的路径。

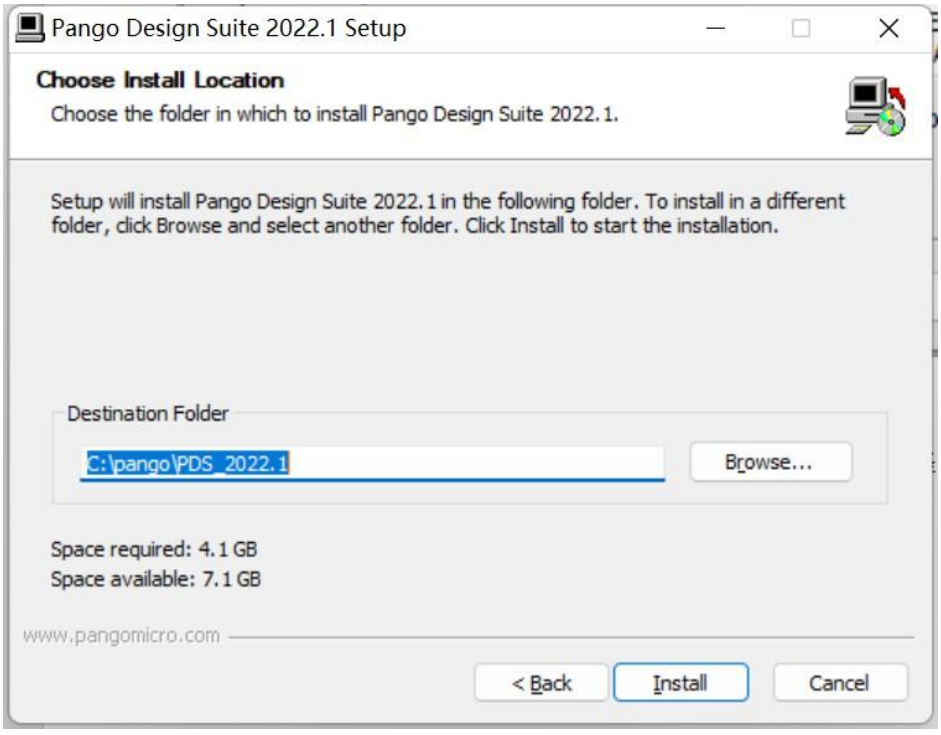


图 2-1.1-4 软件安装流程图 4

（4） 直接点击“Install”，则跳转到安装界面。

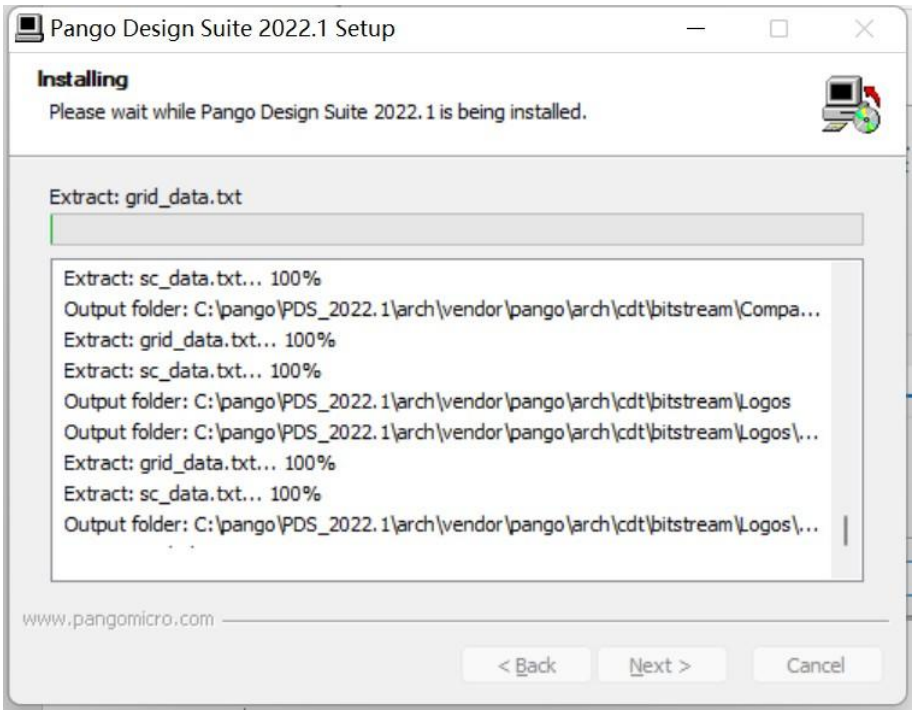


图 2-1.1-5 软件安装流程图 5

⑸ 耐心等待至完成全部安装过程，点击“Finish”，安装完毕。

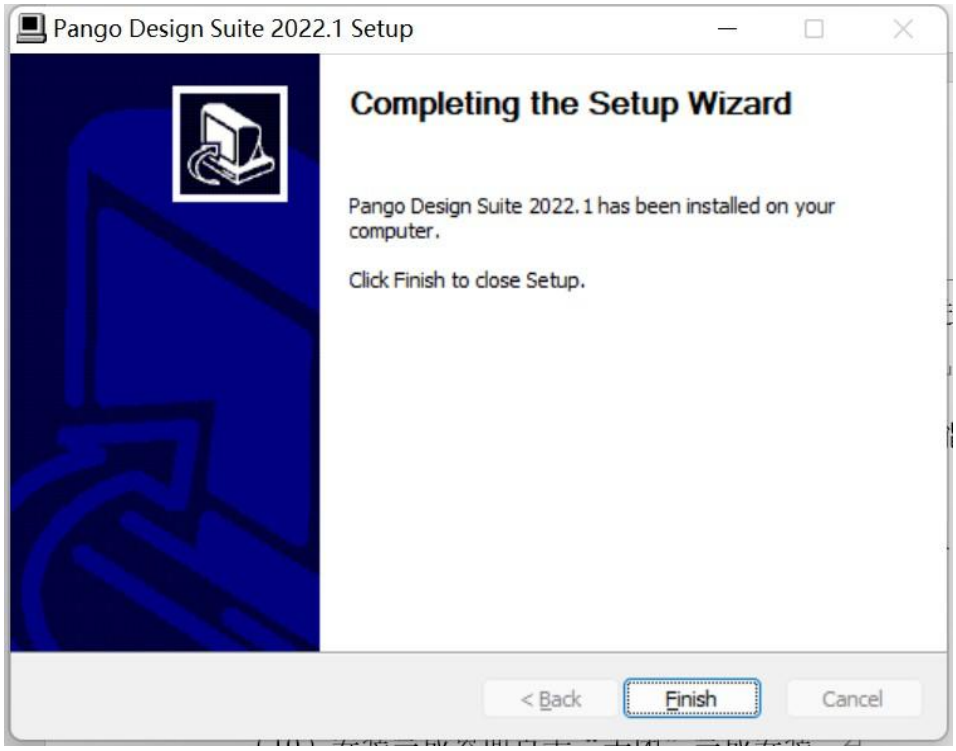


图 2-1.1-6 软件安装流程图 6

（6） 安装完成后，会提示是否需要安装运行库 vcredist_VS2017.exe。若电脑之前未安装过则需要安装此运行库后才能运行 PDS，点击“是”按钮进行安装；若电脑之前已安

装过此运行库则无需再次安装，点击“否”按钮不进行安装即可。（如果没有这个提示，可以不管）

注：如果不确定，建议点击“是”进行安装，否则可能导致 PDS 无法运行。

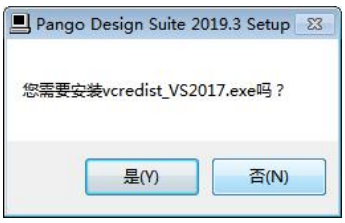


图 2-1.1-7 软件安装流程图 7

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

图 2-1.1-8 软件安装流程图 8

（8） 安装完成界面点击“关闭”完成安装。

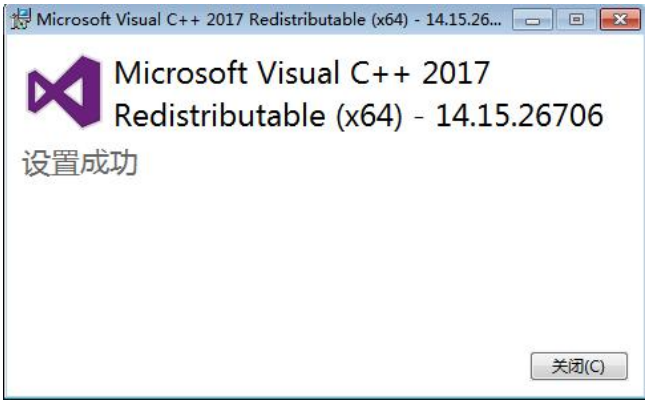


图 2-1.1-9 软件安装流程图 9

（9） 完成安装后，会提示是否需要安装 USB Cable Driver。安装点击“是”，否则可能

导致下载器无法正常使用。

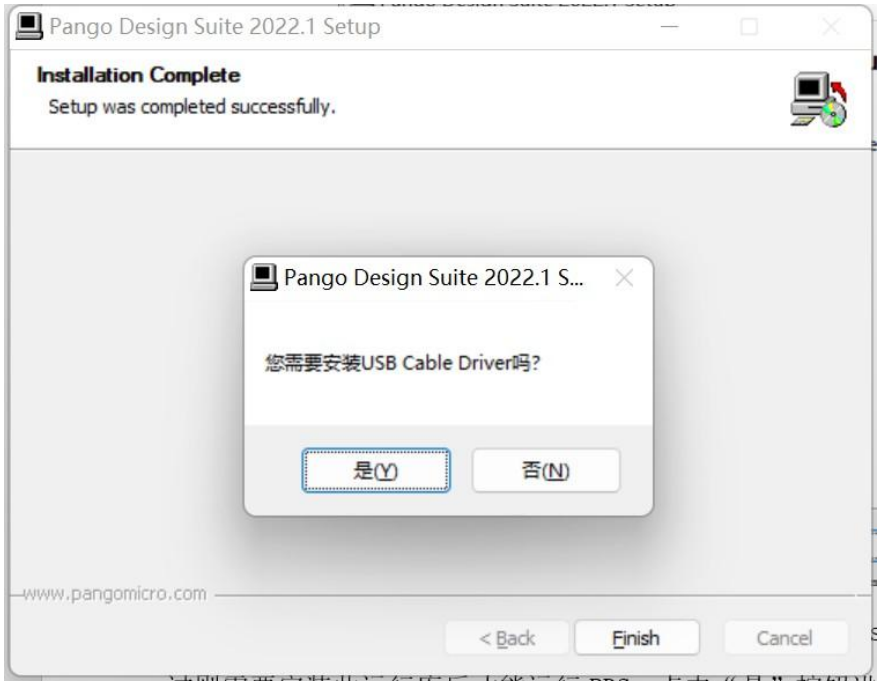


图 2-1.1-10 软件安装流程图 10

（10） 点击“是”进入驱动程序安装界面：

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

图 2-1.1-11 软件安装流程图 11

图 2-1.1-12 软件安装流程图 12

（12） 点击“完成”，完成驱动程序的安装。

图 2-1.1-13 软件安装流程图 13

（13） 完成安装后，会提示是否需要安装 ParellelPortDriver。安装点击“是”。

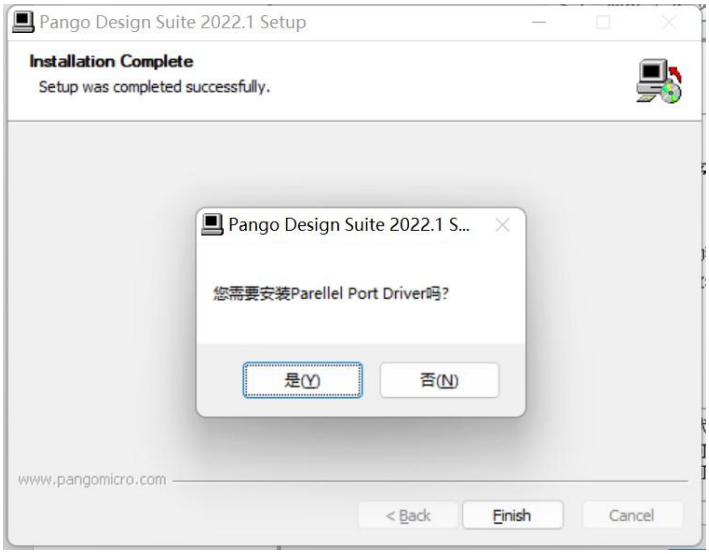


图 2-1.1-14 软件安装流程图 14

（14） 点击“是”进行并口驱动程序安装。

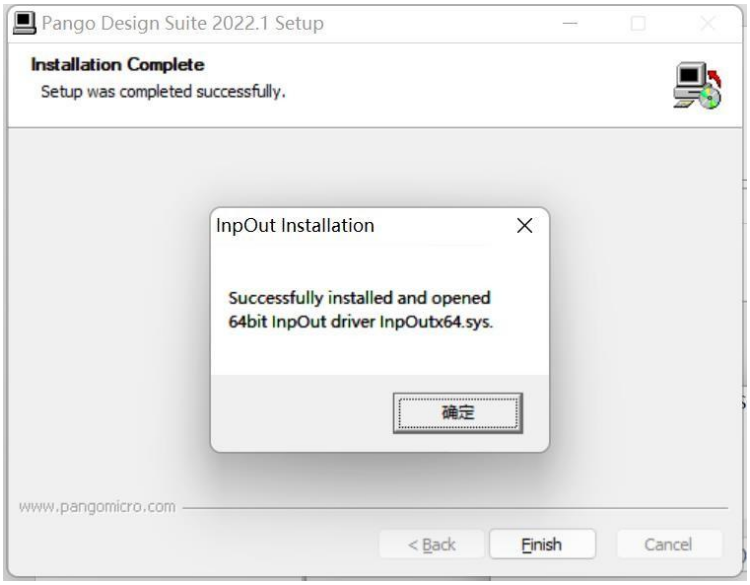


图 2-1.1-15 软件安装流程图 15

（15） 完成后点击“确定”，结束安装。在桌面上看到如下图标：

图 2-1.1-16 软件安装流程图 16

3、License 的关联

同样的，在我们提供的资料包里也准备了对应的 Licensen。使用我们提供的 License 的话，也需要安装一个软件来生成虚拟网卡。Tap-windows。

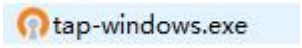


图 2-1.1-1 虚拟网卡生成软件

双击该 exe 程序，该软件的安装，直接默认安装即可，具体步骤如下所示：

图 2-1.1-2 虚拟网卡生成软件安装流程图 1

点击 Next。

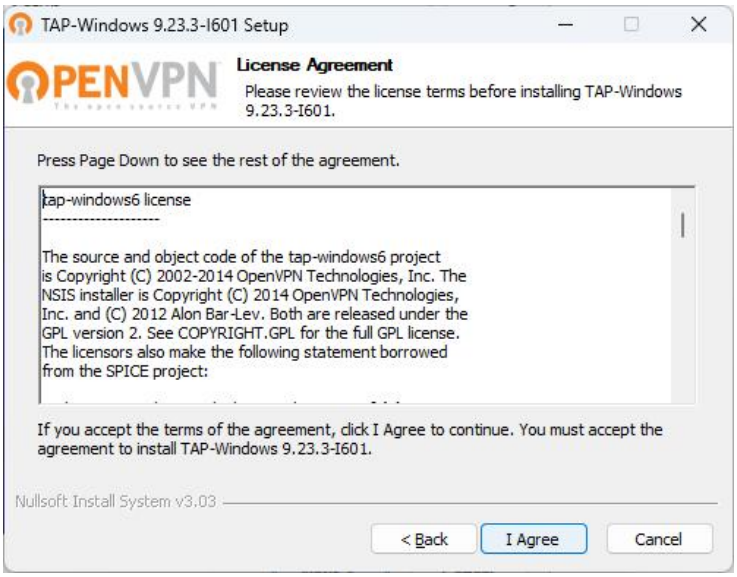


图 2-3-3 虚拟网卡生成软件安装流程图 2

点击 I Agree

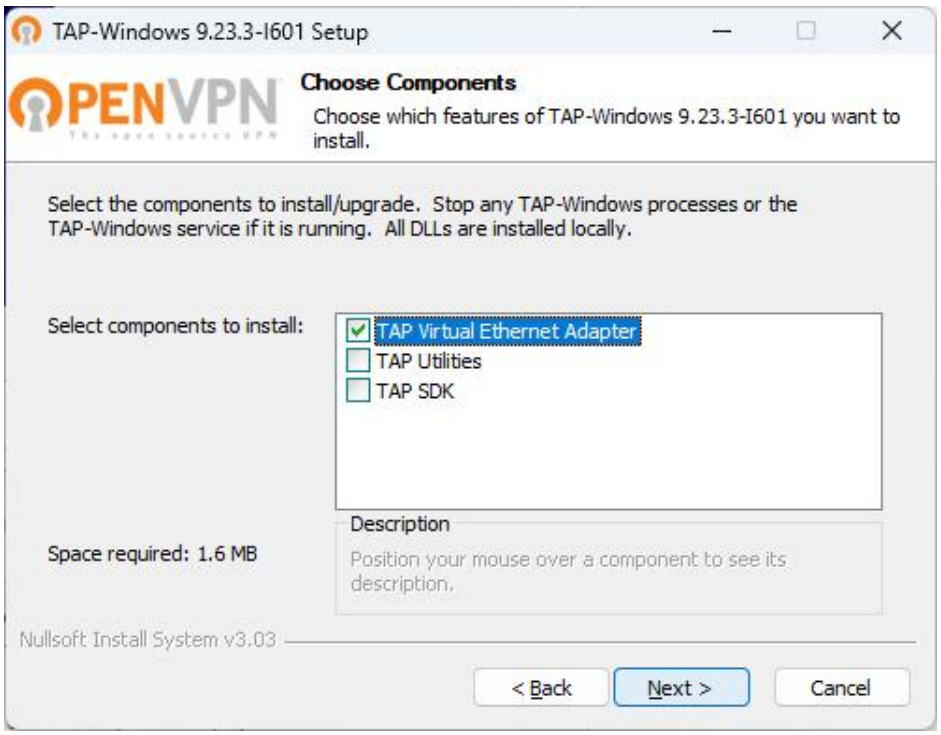


图 2-3-4 虚拟网卡生成软件安装流程图 3

保持默认，点击 Next。

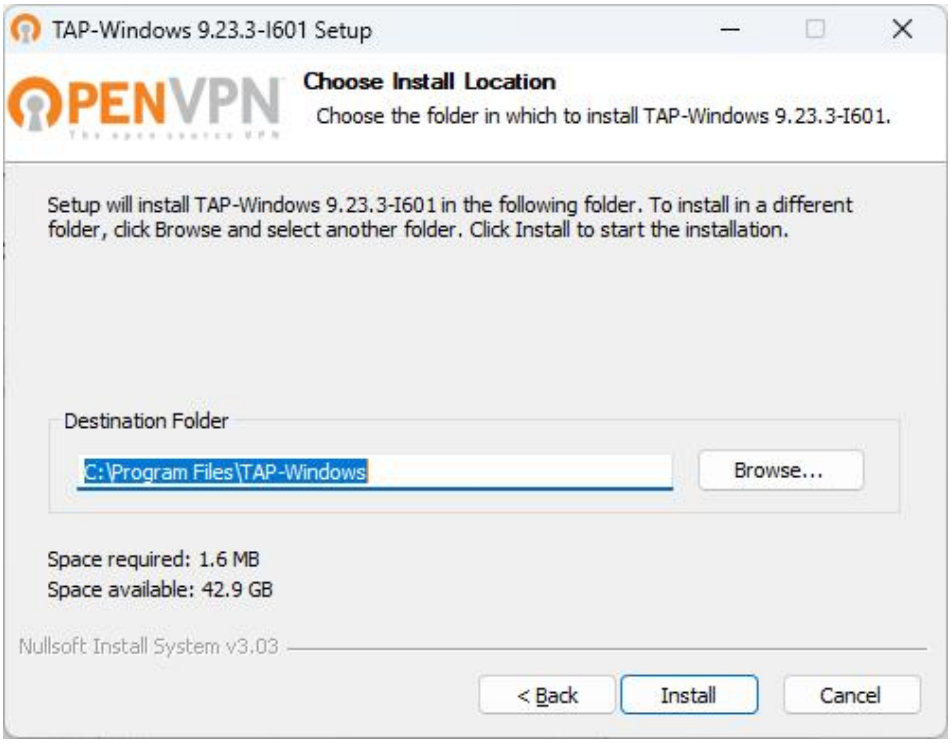


图 2-3-5 虚拟网卡生成软件安装流程图 4

安装路径保持默认，点击 Install，等待安装完成。

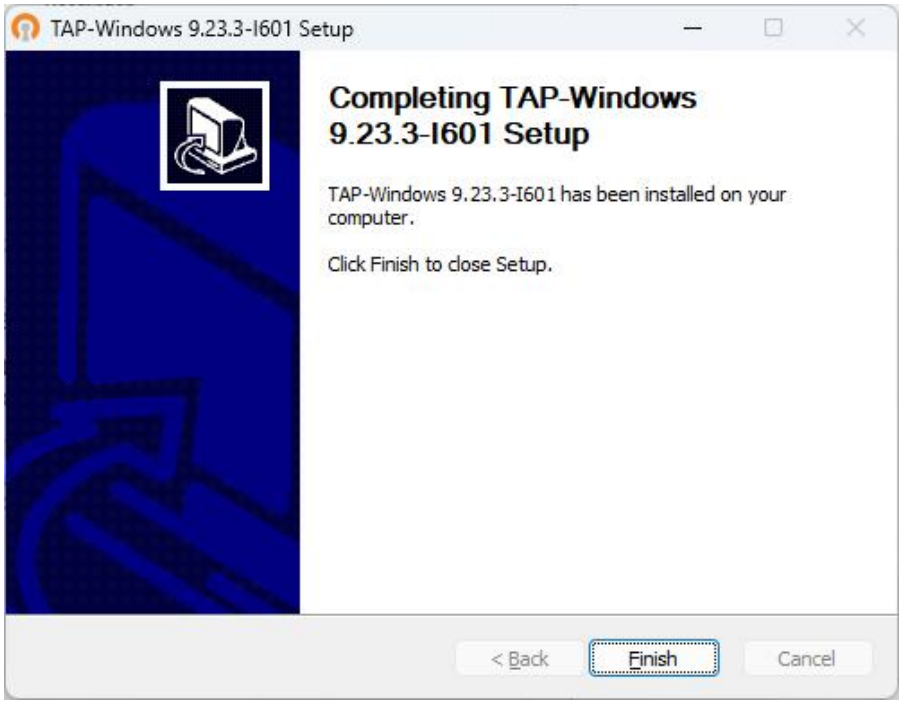


图 2-3-6 虚拟网卡生成软件安装流程图 5

点击 Finish，至此，安装完成。

接下来是 Licnse 的关联：

软件完成安装后，Pango Design Suite 需要License 文件才能正常使用。

1. License 申请请联系相关销售，提供相关资料，他会帮您申请；目前我们也直接提供一个可以使用的 License。

2. 若只使用 Verilog 则只需申请 PDS License 即可，若使用 VHDL 或System verilog 则还需要申请 Synplify license；

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

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

图 2-3-7 License 关联流程图 1

或者直接打开系统环境变量

图 2-3-8 License 关联流程图 2

PDS license 环境变量设置：

若 PDS License 文件路径为 D:\pango\license\pds_node-locked.lic。

直接设置环境变量：

变量名：PANGO_LICENSE_FILE

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

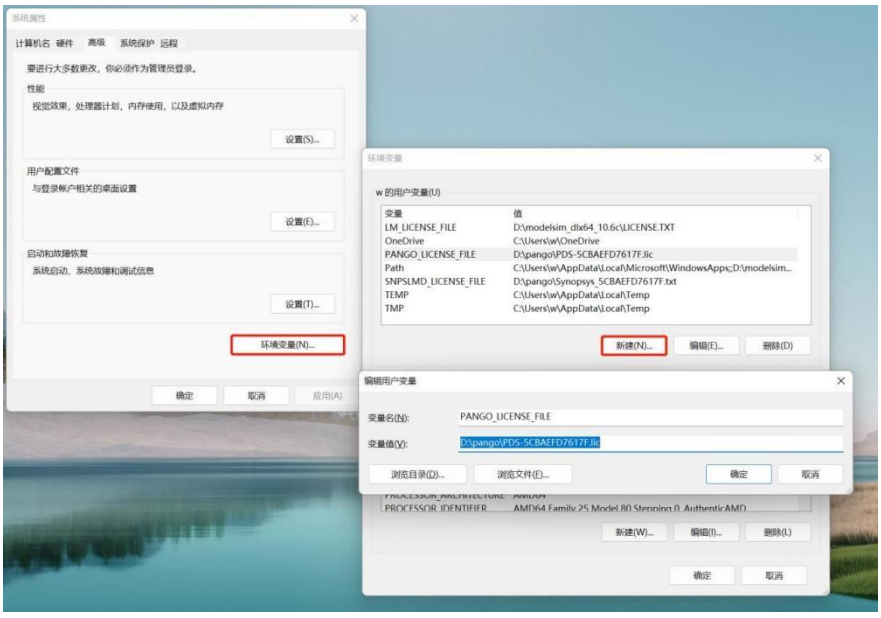


图 2-3-9 License 关联流程图 3

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

图 2-3-10 License 关联流程图 4

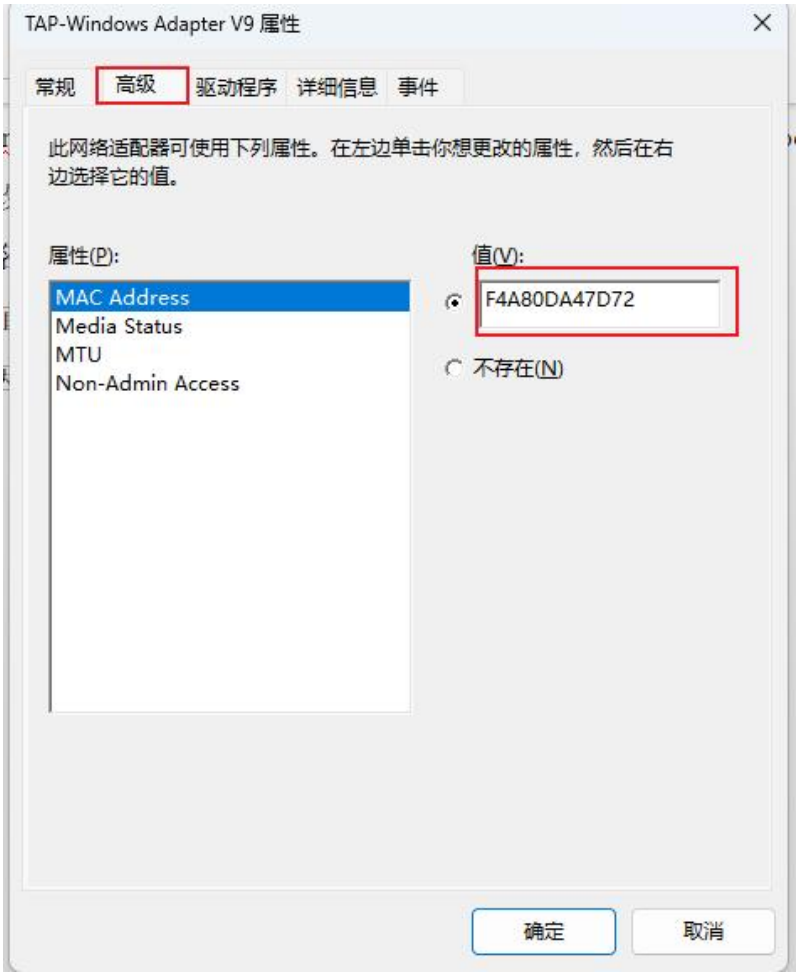


图 2-3-11 License 关联流程图 5

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。

4、Modelsim 与 PDS 联合仿真

安装完 PDS 软件后，安装 Modelsim 仿真工具。PDS 并没有自带的仿真工具，故需要联合第三方一同使用。由于 Modelsim 涉及到破解，这里不过多描述。大家可以去百度上查找安装教程，在网上是非常多的教程的，例如 CSDN 等。

安装完 Modelsim 后，接下来就是编译我们的 PDS 仿真库文件，具体如下所示：

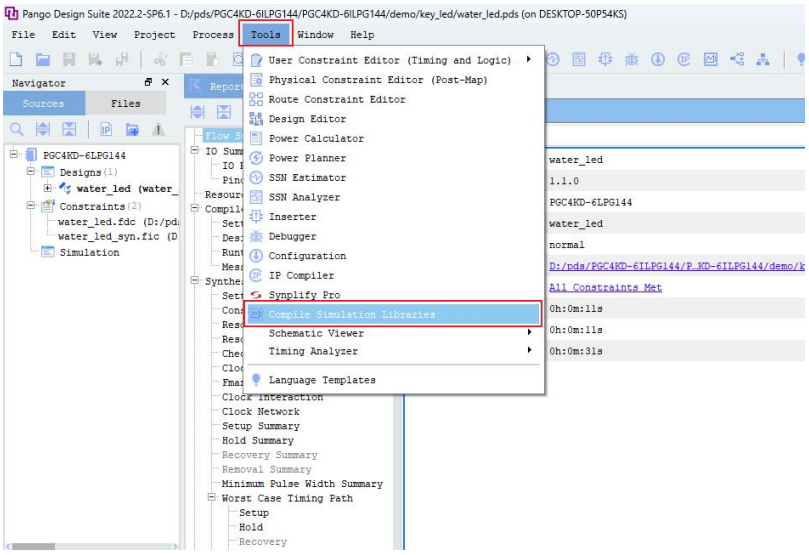


图 2-4-1 Modelsim 与 PDS 联合仿真示意图 1

打开 PDS 工程后，选择工具栏里的 Tools->Comolie Simulation Libraries。

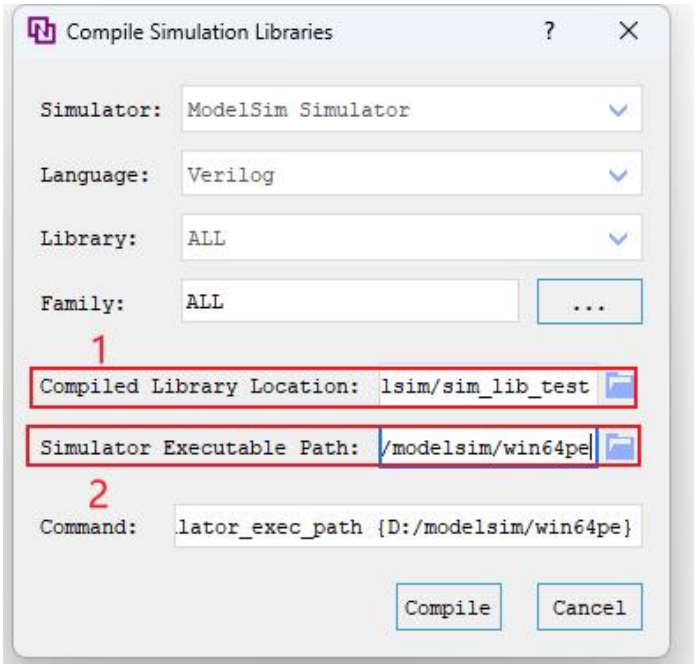


图 2-4-2 Modelsim 与 PDS 联合仿真示意图 2

红框 1 选择存放生成的仿真库的路径，红框 2 选择 Modelsim 的启动程序路径，一般在 win64pe 或者 win64 或者 win32 文件夹，具体和 Modelsim 的版本有关，笔者所用的 Modelsim 为 10.6c。

选择好路径后，点击 Compile，然后等待编译结束即可。

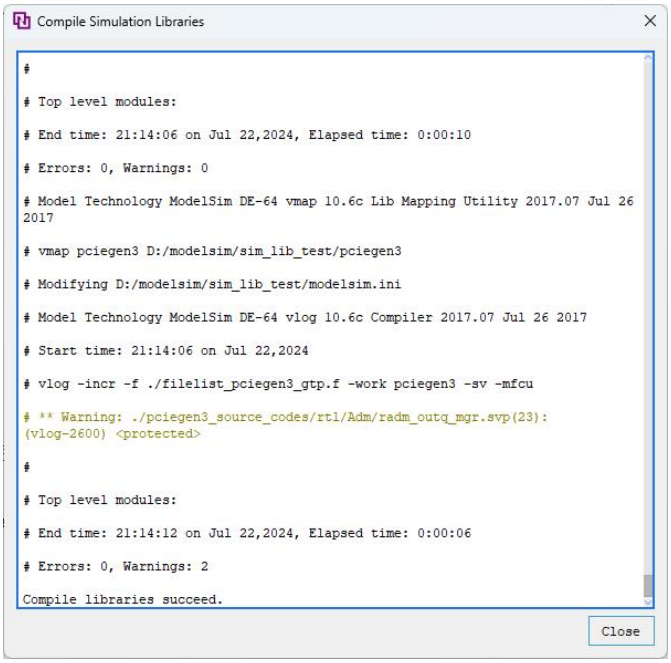


图 2-4-3 Modelsim 与 PDS 联合仿真示意图 3

当出现以上结果时，表示生成成功，点击 Close。

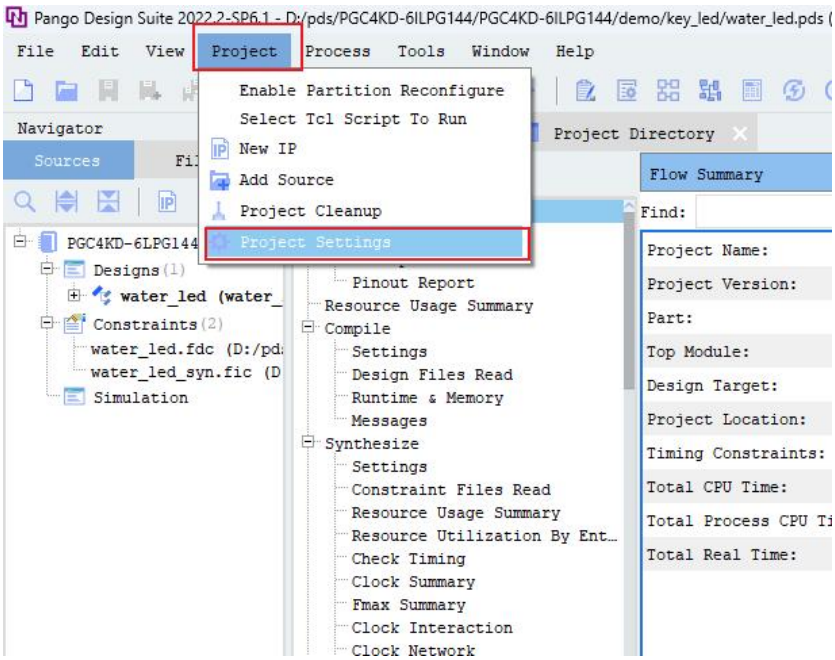


图 2-4-4 Modelsim 与 PDS 联合仿真示意图 4

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

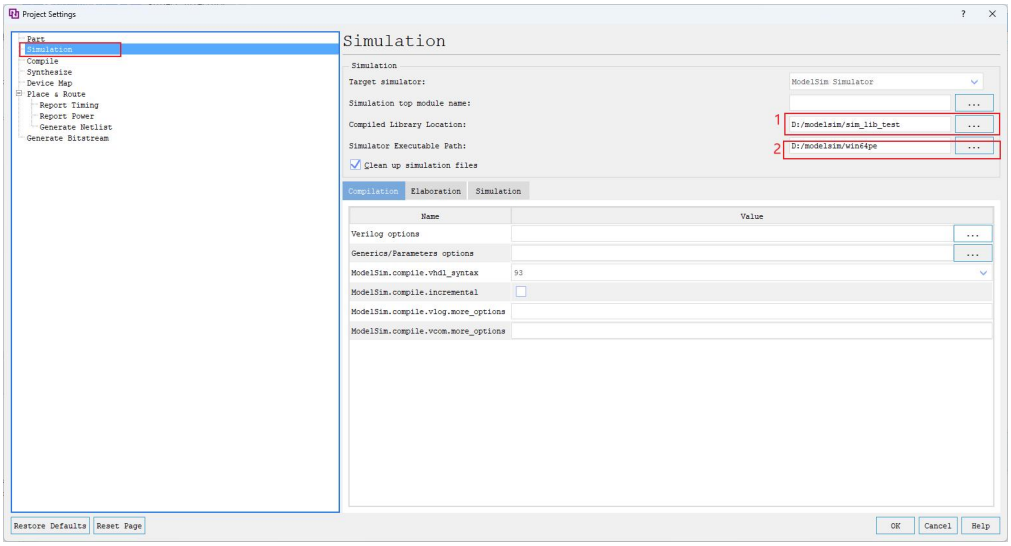


图 2-4-5 Modelsim 与 PDS 联合仿真示意图 5

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。

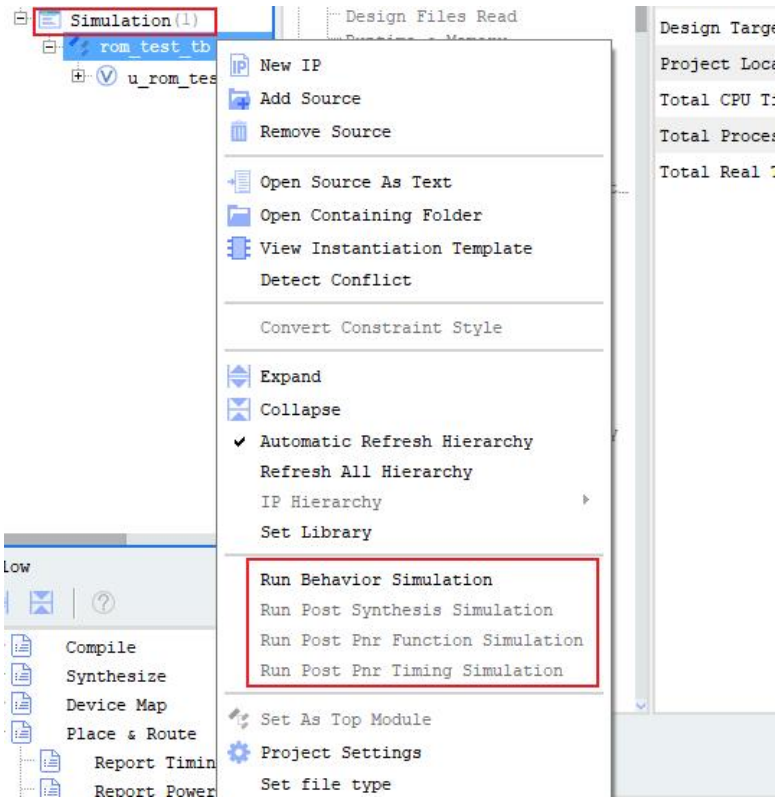


图 2-4-6 Modelsim 与 PDS 联合仿真示意图 6

（1）“Run Behavior Simulation”是综合前的行为级仿真，是开发流程中必要的环节，它能确保你所设计的功能是否正确。

（2）“Run Post Synthesis Simulation”是综合后 gtp 仿真，在复杂的工程项目中，可以通过它来排除大部分的时序问题。

（3）“Run Post Pnr Function Simulation”是 pnr 后的功能仿真。

（4）“Run Post Pnr Timing Simulation“是 pnr 后带 sdf 的时序仿真。

之后,右键要仿真的文件,选择 Run Behavior Simulation 即可开始行为仿真,这是比较常用的仿真,可以查看我们程序涉及的逻辑的波形是否正确。

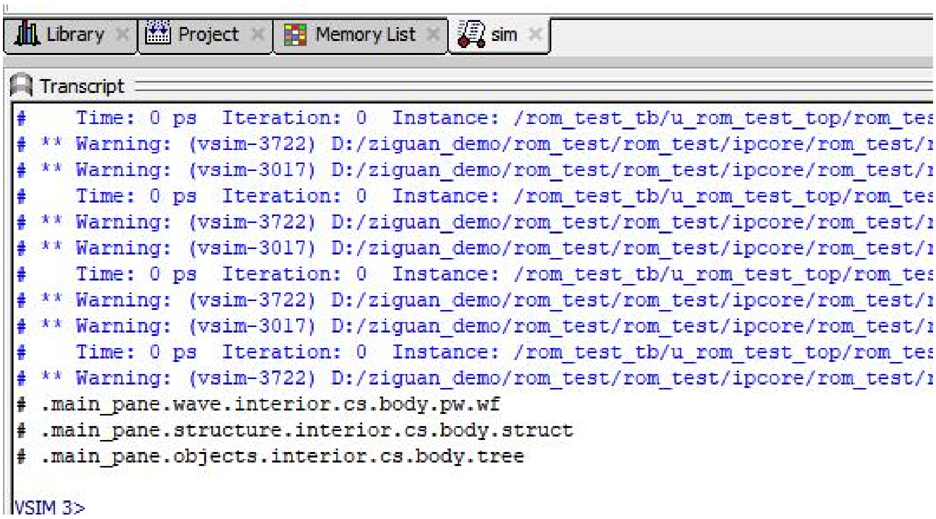


图 2-4-7 Modelsim 与 PDS 联合仿真示意图 7

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。

第三部分、实验指导

1、点亮 LED 灯实验

1.1、实验目的

实现对多个 LED 灯的控制;

1.2、实验要求

控制 8 个 LED 以 1s 的周期闪烁（0.5s 亮，0.5s 灭）

1.3、实验原理

生活中使用时分秒进行进位计时；1 小时=60 分钟=3600 秒，当时针转动 1 小时，秒针跳动 3600 次；

那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）。

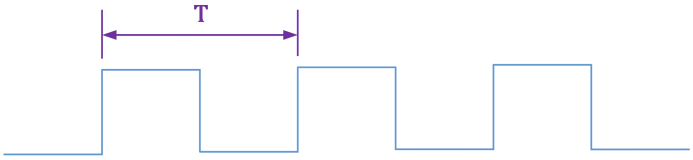


图 3-1.3-1 时钟信号波形图

在数字系统中通常关注到时钟的频率,那频率与周期的关系如下：

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

逻辑派 Z1 开发板上有一个 50MHz 的晶振提供时钟给到 PGC4KD;

实验分析:

控制 LED 亮灭需要控制 IO 输出的高低电平即可(高电平点亮,低电平熄灭),原理图如图 3-2 所示：

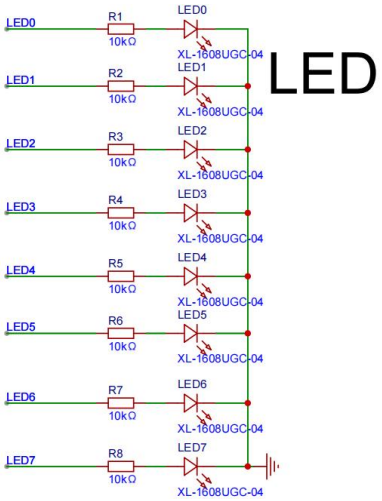


图 3-1.3-2 LED 电路原理图

led 灯电路图

控制 LED 周期性的维持 0.5s 亮,0.5s 灭,需要控制 IO 输出 0.5s 高电平,0.5s 低电平周期变化,如下图波形:

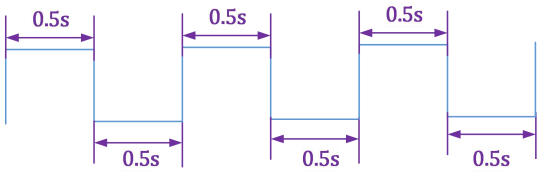


图 3-1.3-3 LED 实验信号波形图

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

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

IO 输出状态只有两种：1 或 0；我们可以使用一个计数器，计数满 25000000 个时钟周期时将 IO 状态进行翻转，即可完成每 0.5S 输出状态跳转，即 LED 灯会以 0.5S 的间隔亮灭变化；

1.4、实验源码设计

```
1.  timescale 1ns / 1ps
2.  define UD #1
3.
4.
5.
```

```
6.
7.    module led_light(
8.        input         clk,
9.        input         rstn,
10.
11.       output [7:0]   led
12.   );
13.
14.   //=============================================================================
15.   //reg and wire
16.
17.       reg [25:0] led_light_cnt    = 26'd0        ;
18.       reg [ 7:0] led_status       = 8'b0000_0000  ;
19.
20.   //time counter
21.       always @(posedge clk)
22.       begin
23.           if(!rstn)
24.               led_light_cnt <= UD 26'd0;
25.           else if(led_light_cnt == 26'd24_999_999)
26.               led_light_cnt <= UD 26'd0;
27.           else
28.               led_light_cnt <= UD led_light_cnt + 26'd1;
29.       end
30.
31.   //led status change
32.       always @(posedge clk)
33.       begin
34.           if(!rstn)
35.               led_status <= UD 8'b0000_0000;
36.           else if(led_light_cnt == 25'd24_999_999)
37.               led_status <= UD ~led_status;
38.   end
39.
40.       assign led = led_status;
41.
42.   endmodule
```

1.5、实验步骤

逻辑派 Z1 板卡在使用 PDS 开发前需要先对 PDS 进行驱动的下载，请参考驱动安装说明进行驱动的下载。

1.5.1、 打开 PDS 软件，创建工程

Step1：打开 PDS 软件，单击 New Project。

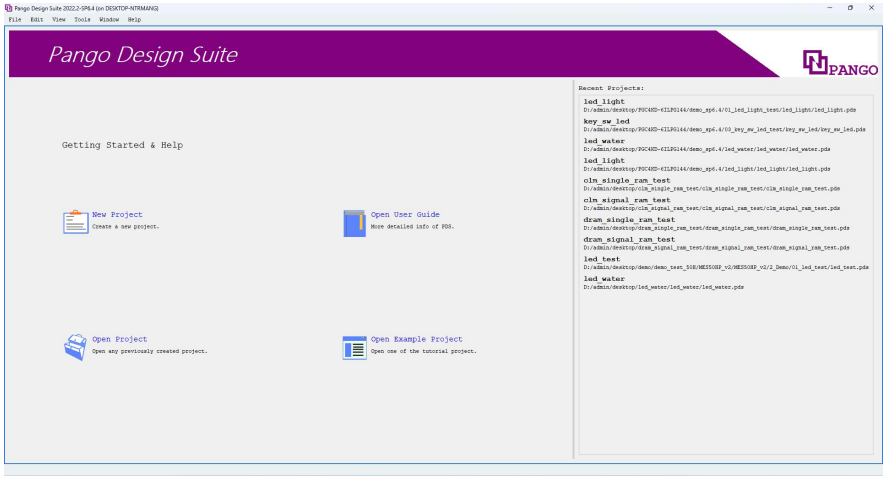


图 3-1.5-1 PDS 工程创建流程图 1

Step2：单击 NEXT。

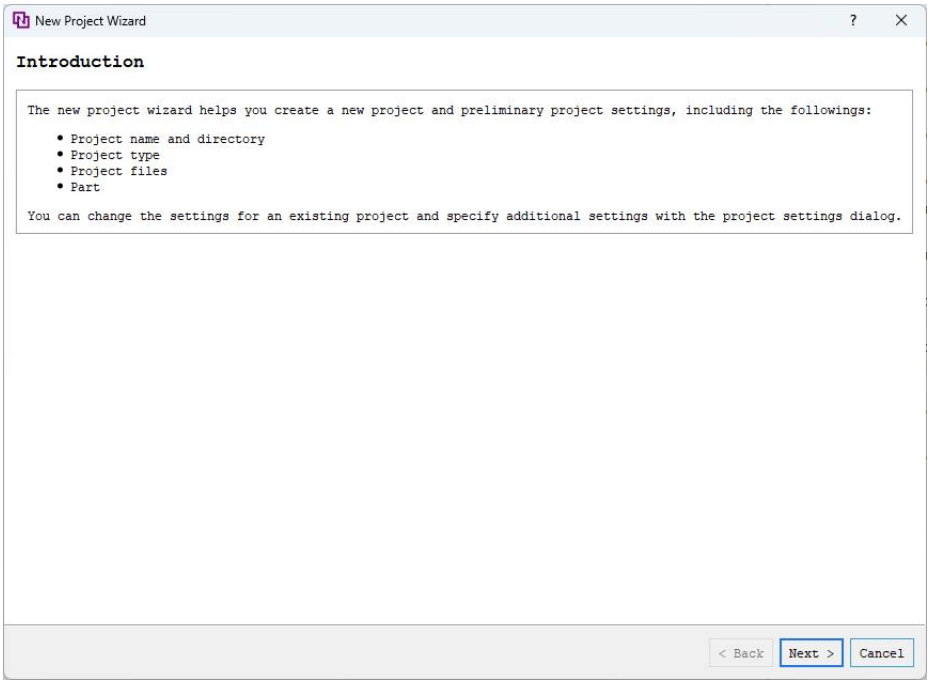


图 3-1.5-2 PDS 工程创建流程图 2

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

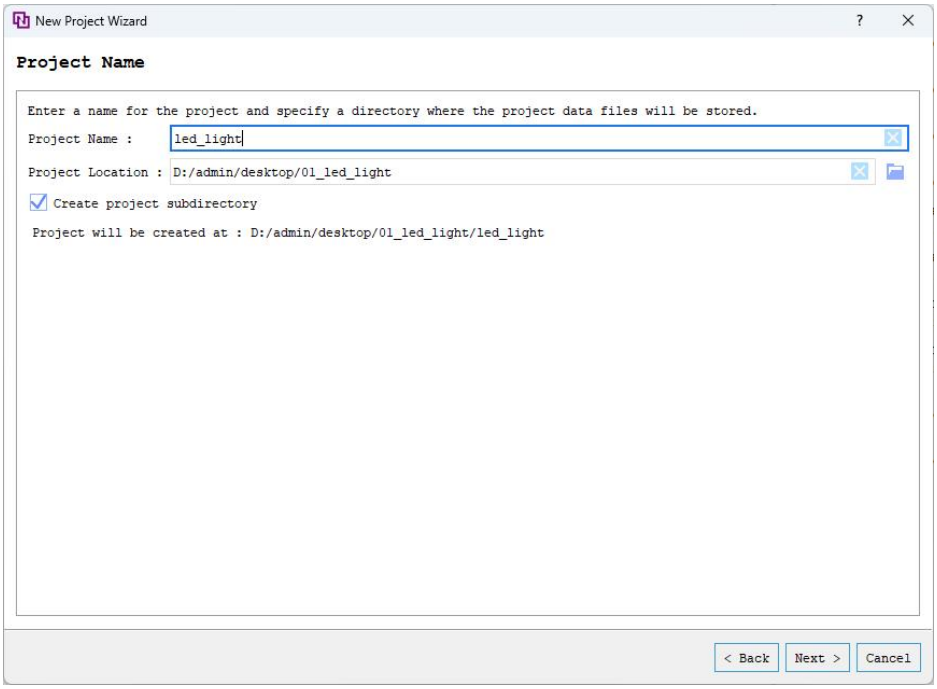


图 3-1.5-3 PDS 工程创建流程图 3

Step4：选择 RTL project，单击 Next。

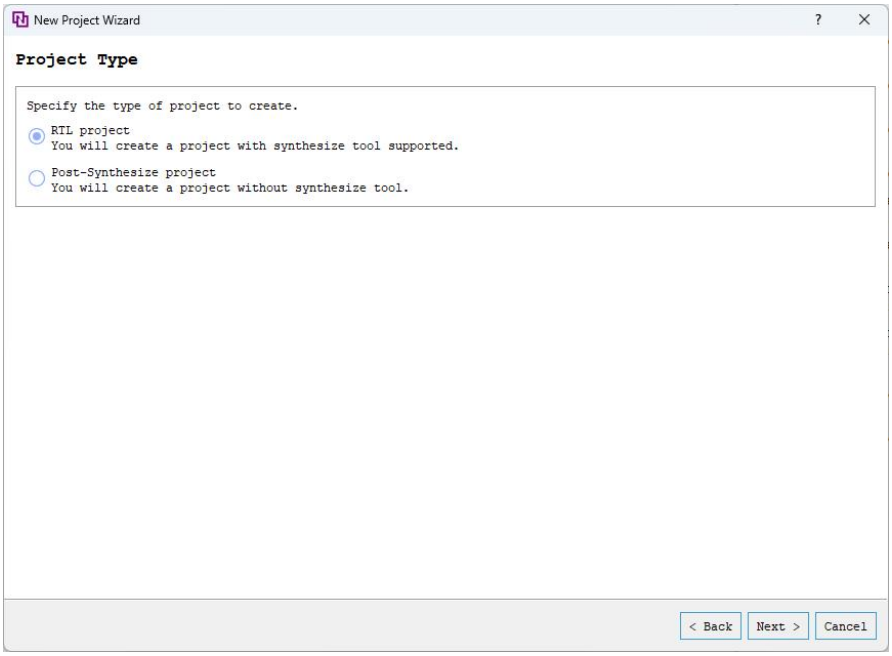


图 3-1.5-4 PDS 工程创建流程图 4

Step5：单击 Next(也可添加.v 文件)。

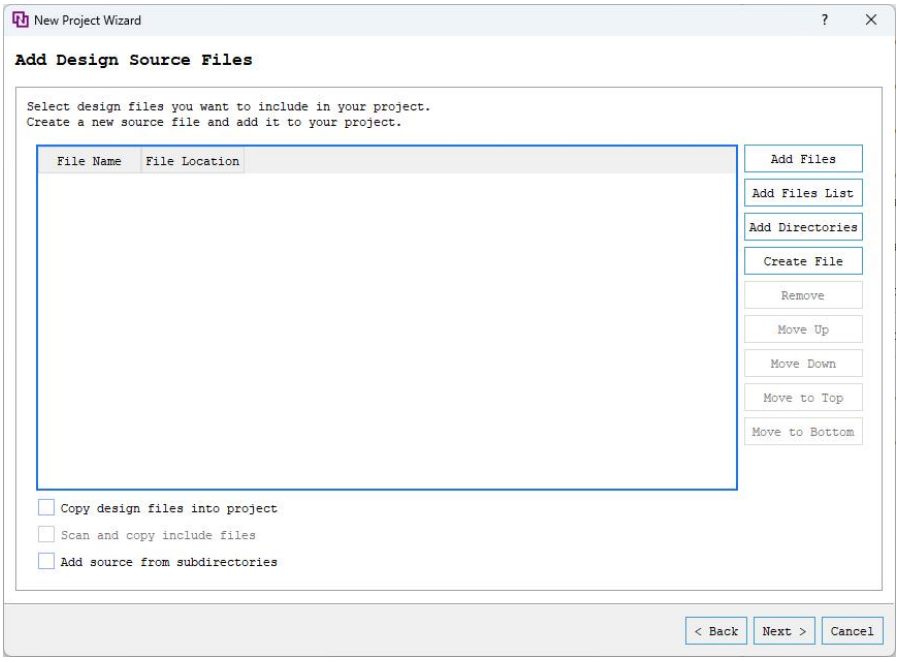


图 3-1.5-5 PDS 工程创建流程图 5

Step6：单击 Next。

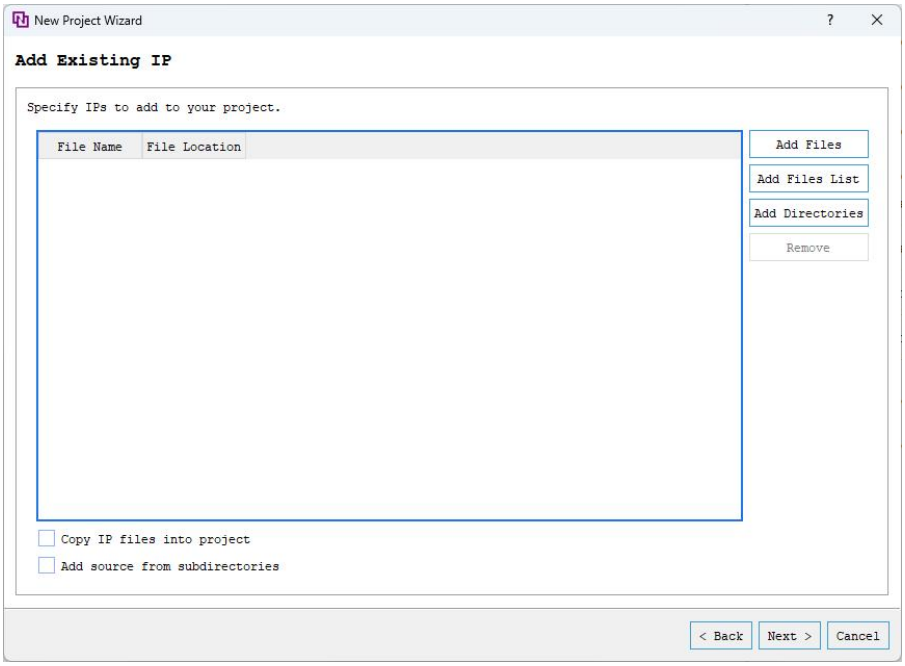


图 3-1.5-6 PDS 工程创建流程图 6

Step7：单击 Next。

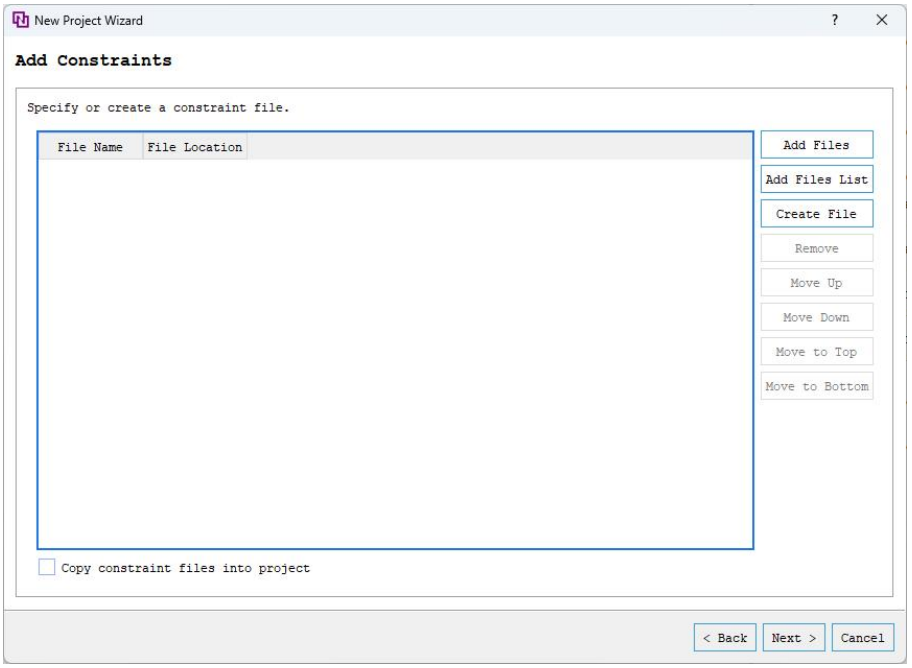


图 3-1.5-7 PDS 工程创建流程图 7

Step8：选择器件系列、型号、封装、速率、综合工具，之后单击 Next。

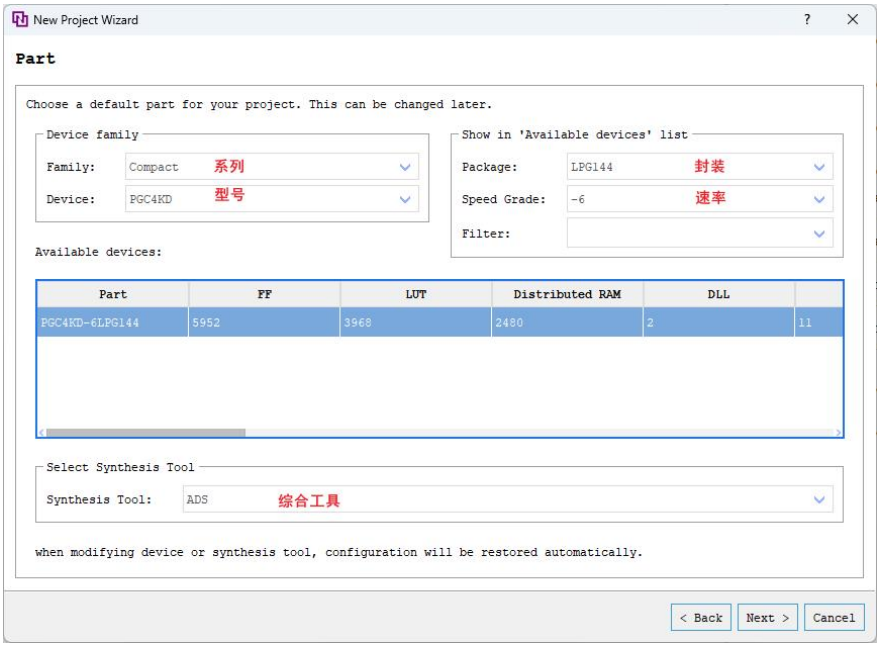


图 3-1.5-8 PDS 工程创建流程图 8

Step9：单击 Finish,完成工程创建。

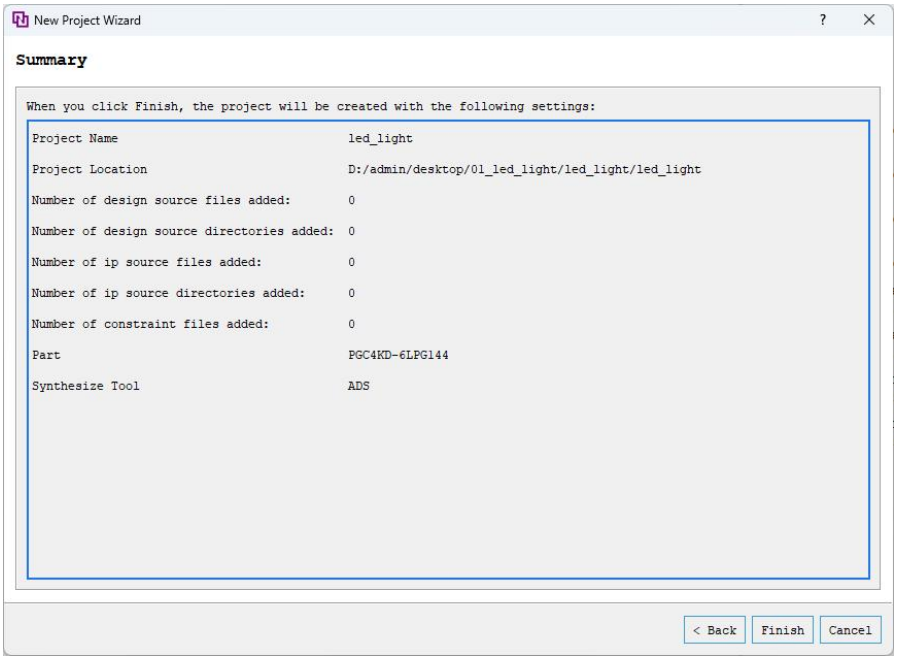


图 3-1.5-9 PDS 工程创建流程图 9

1.5.2、添加设计文件或创建设计文件

- 创建设计文件：

step1：双击 Designs。

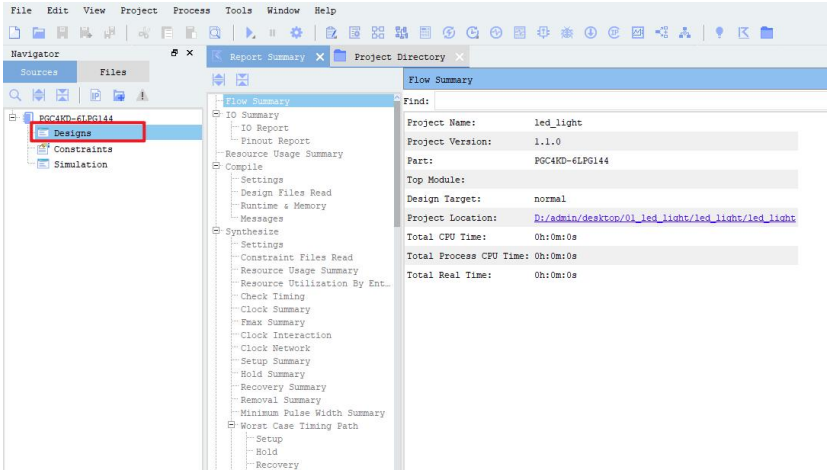


图 3-1.5-10 创建设计文件流程图 1

step2：点击 Create File。

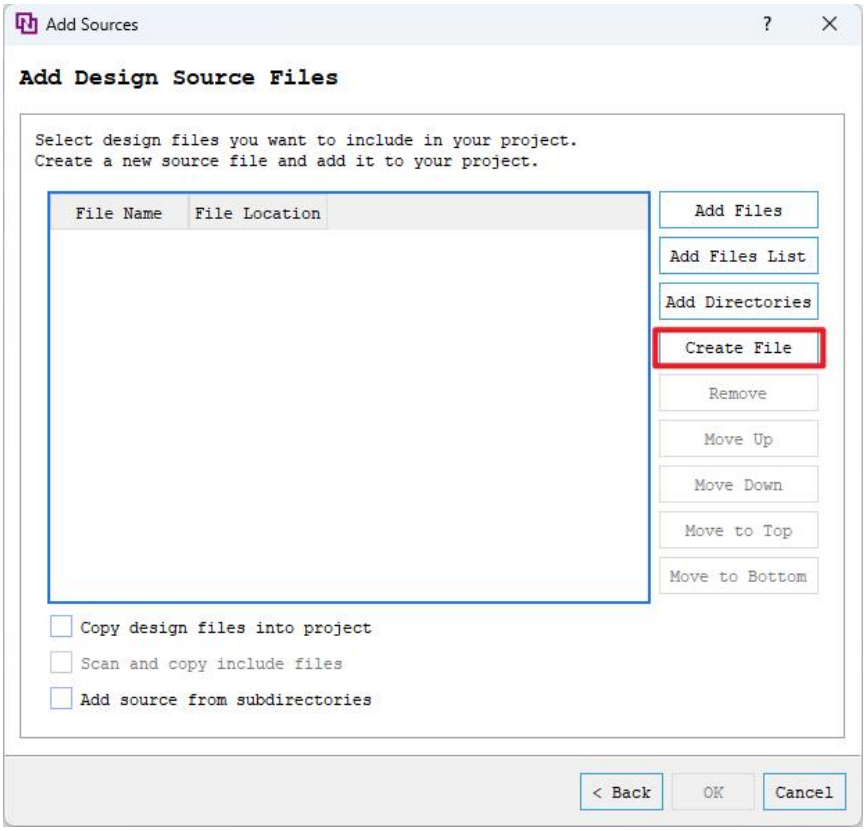


图 3-1.5-11 创建设计文件流程图 2

step3：选择 Verliog Design File，File Name 与 module 名需保持一致，默认

路径，点击 OK。

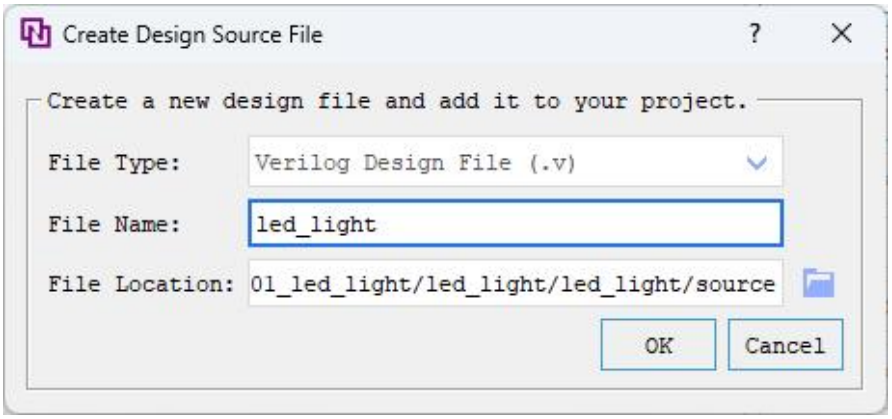


图 3-1.5-12 创建设计文件流程图 3

step4：点击 OK。

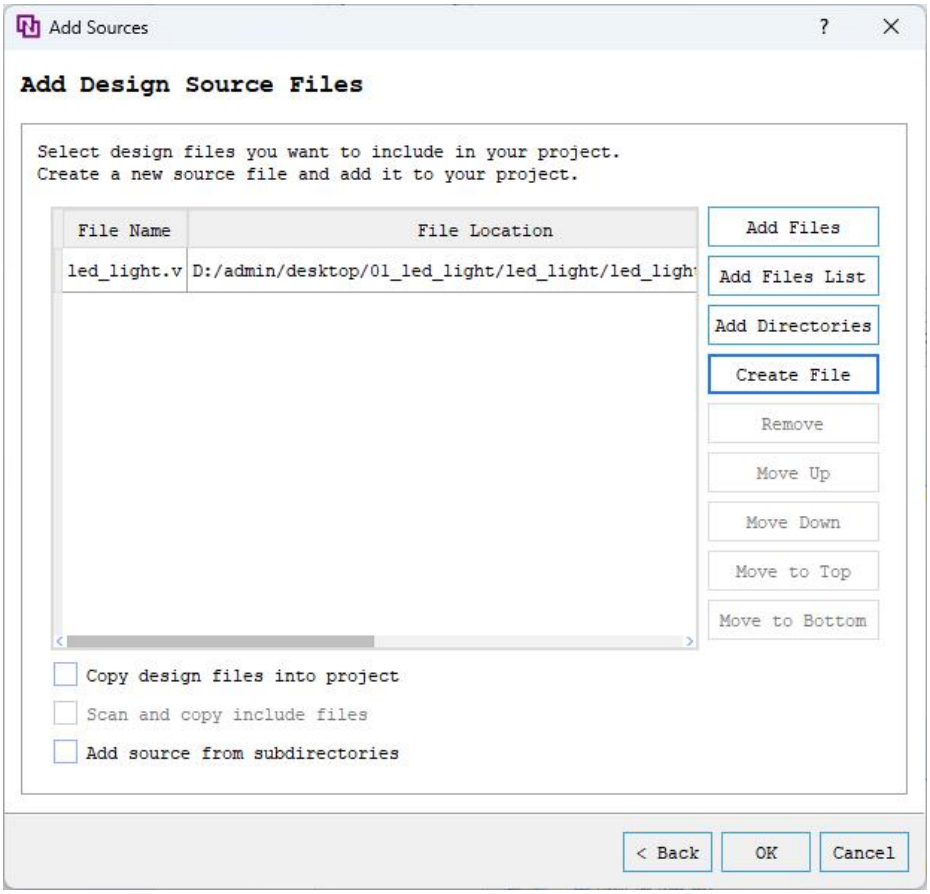


图 3-1.5-13 创建设计文件流程图 4

step5：点击 cancel。

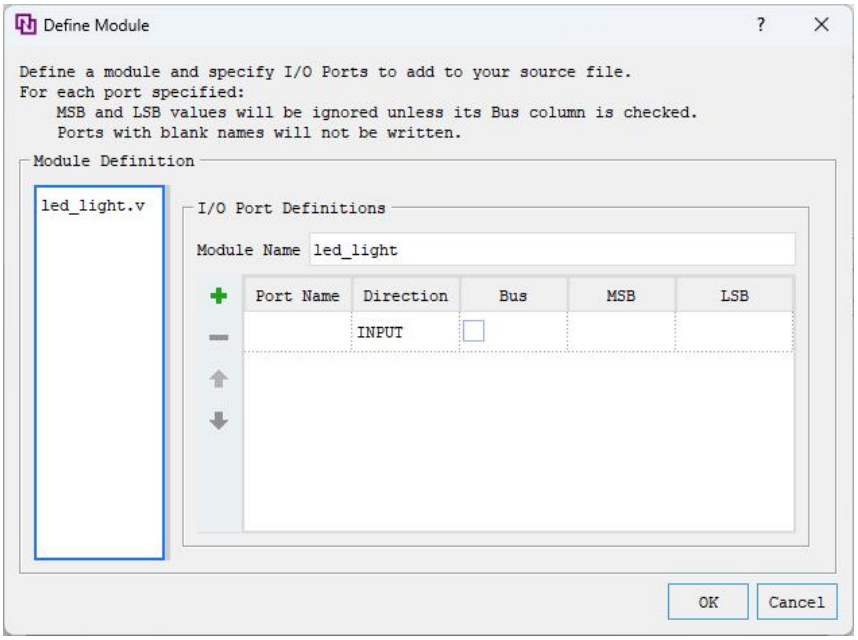


图 3-1.5-14 创建设计文件流程图 5

step6：打开新建好的文件，将设计代码复制进去后，点击保存。

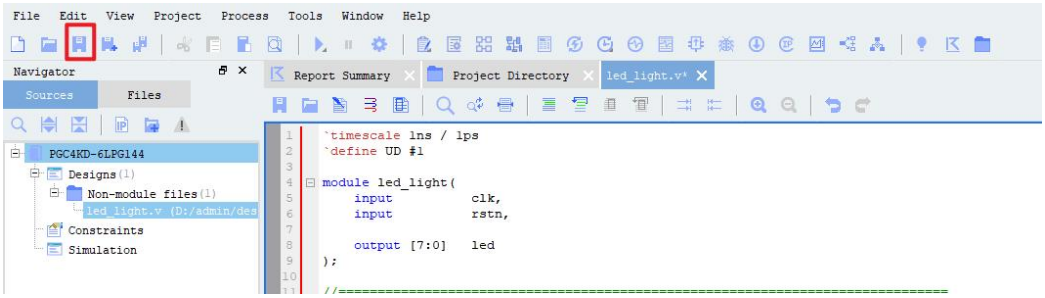


图 3-1.5-15 创建设计文件流程图 6

- 添加设计文件

step1：双击 Designs。

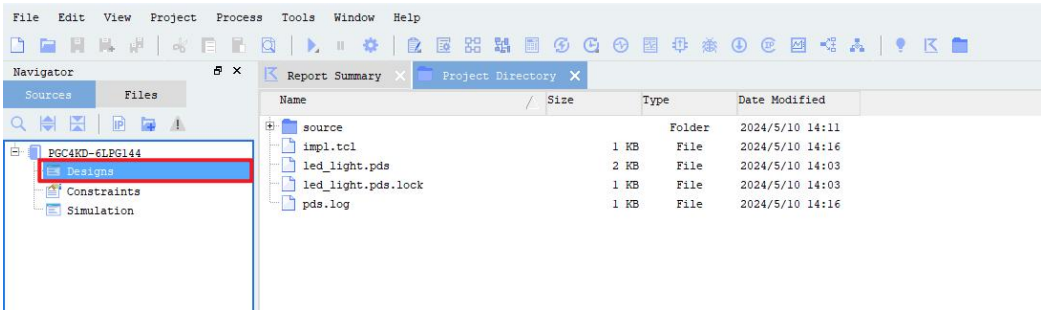


图 3-1.5-16 添加设计文件流程图 1

step2：Add Files。

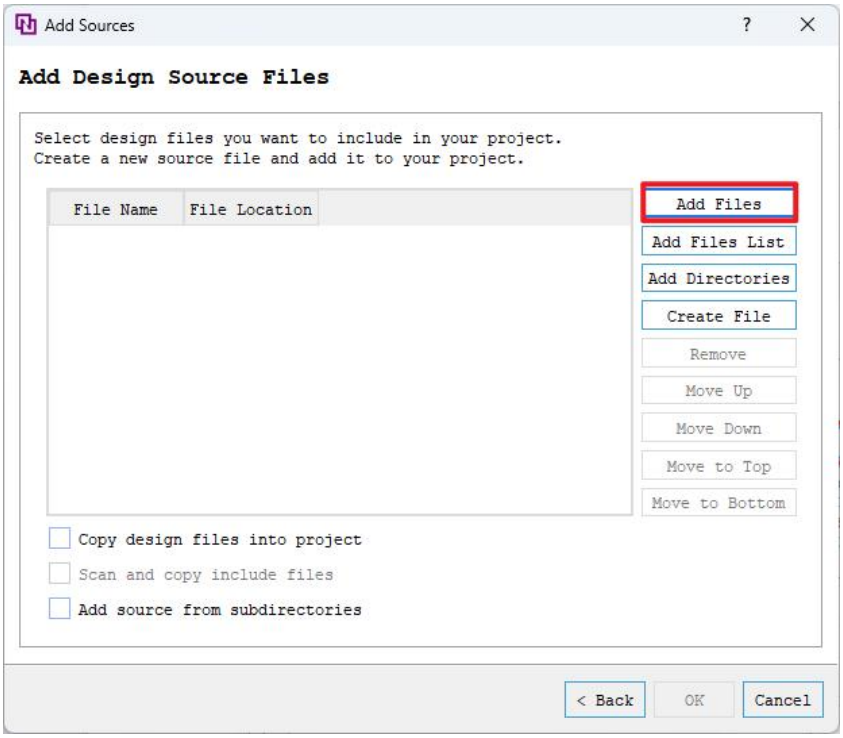


图 3-1.5-17 添加设计文件流程图 2

1.5.3、编译

双击 Flow 下的 Compile 或右键点击 Compile 选择 run 选项，完成编译。

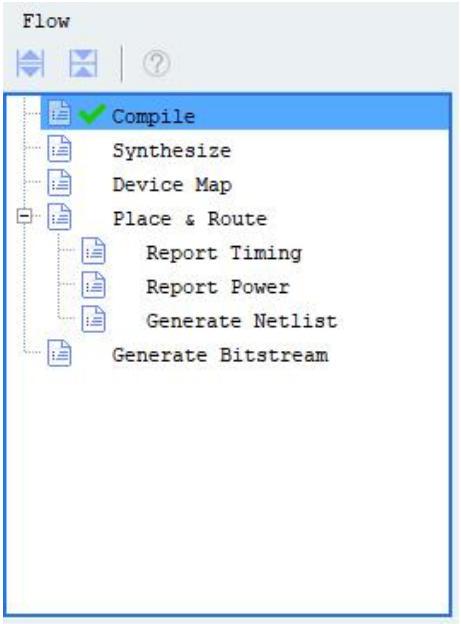


图 3-1.5-18 编译示意图

1.5.4、工程约束

- 时钟约束：

打开 UCE 后，选择 Timing Constraints 后选择 Create Clock 添加基准时钟。

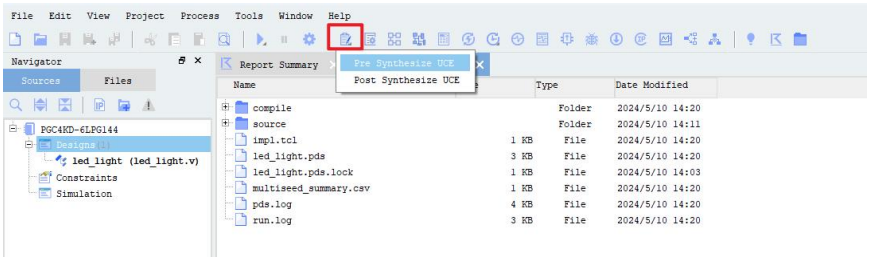


图 3-1.5-19 添加时钟约束示意图 1

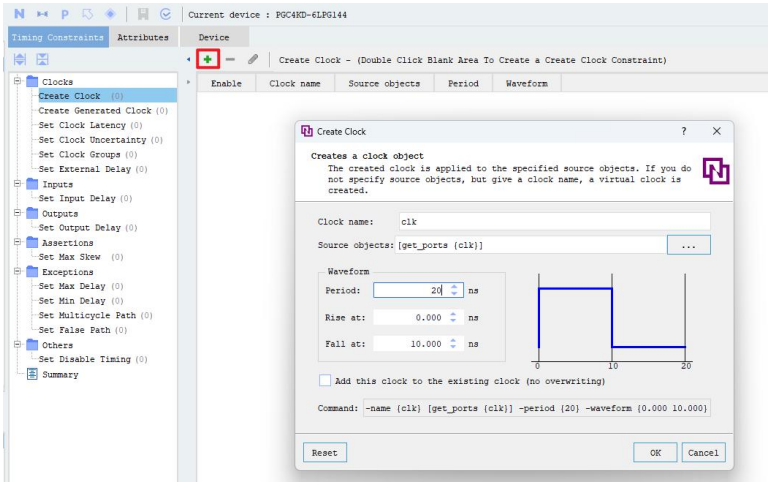


图 3-1.5-20 添加时钟约束示意图 2

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：

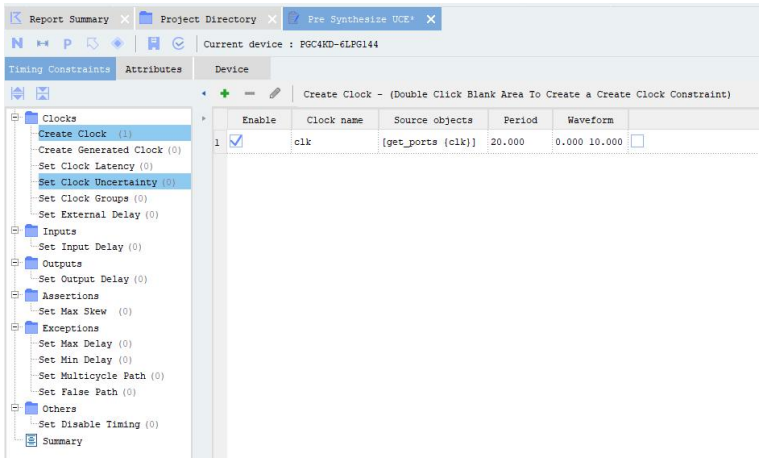


图 3-1.5-21 添加时钟约束示意图 3

- 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。

逻辑派 Z1 管脚 IO 为 3.3V，因此在分配管脚时 VCCIO 应选择 3.3。

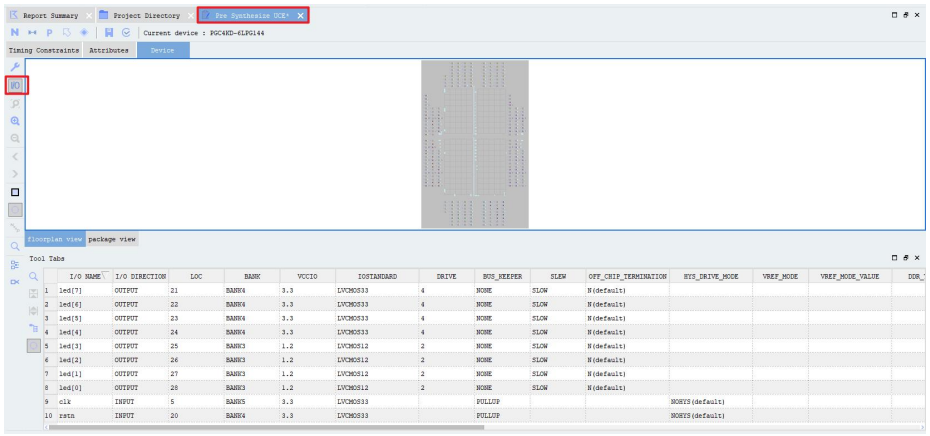


图 3-1.5-22 添加物理约束示意图 1

编辑好 IO 分配后，点击保存，完成约束。

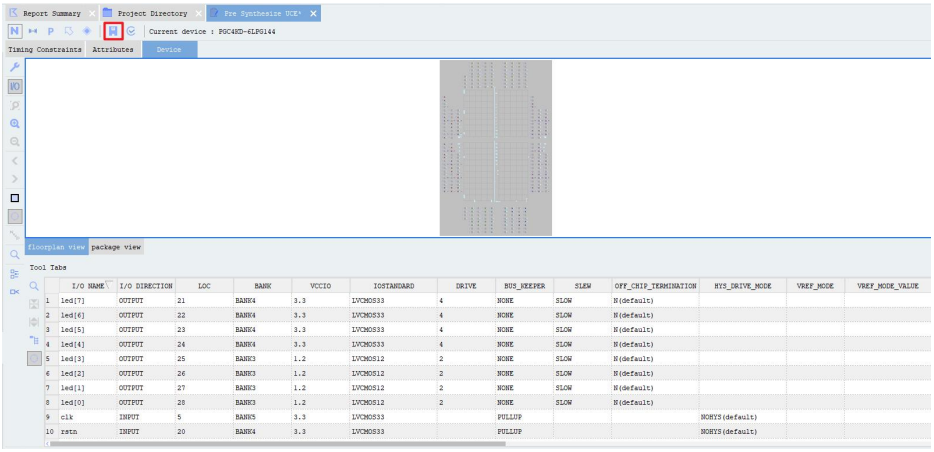


图 3-1.5-23 添加物理约束示意图 2

1.5.5、产生 bit 文件

双击 Generate Bitstream 或右键点击 Generate Bitstream 的 run 选项，PDS 软件将自动完成 Synthesize、Device Map、Place & Route、Report Timing、Gnenerate Bitstream 操作。

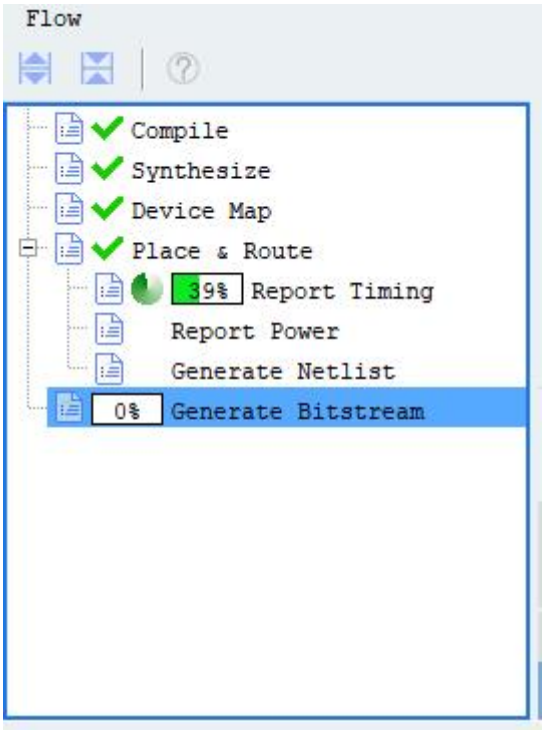


图 3-1.5-24 产生 bit 文件示意图 1

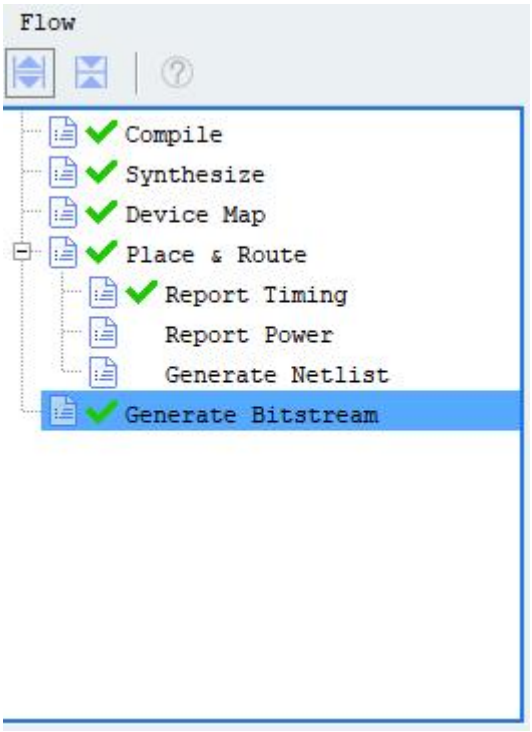


图 3-1.5-25 产生 bit 文件示意图 2

1.6、下载位流（bit）文件

完成板卡连线，将 TYPE-C 连接板卡，另一端接入电脑，如下所示：

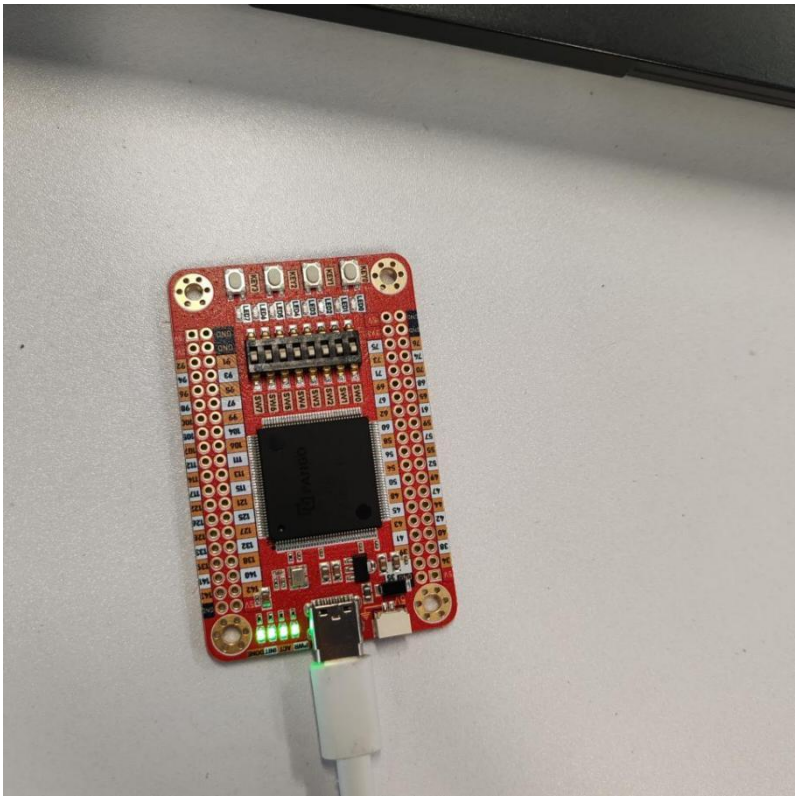


图 3-1.6-1 下载 bit 流文件效果图

在下载程序前，我们需要安装一个驱动，电脑才能识别到我们的板载 JTAG。

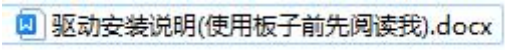


图 3-1.6-2 驱动安装说明文档图

在提供的资料包里也有一个文档，里面说明了驱动如何安装。之后打开 PDS 软件，按如下步骤操作即可。

step1：点击页面 Configuration。

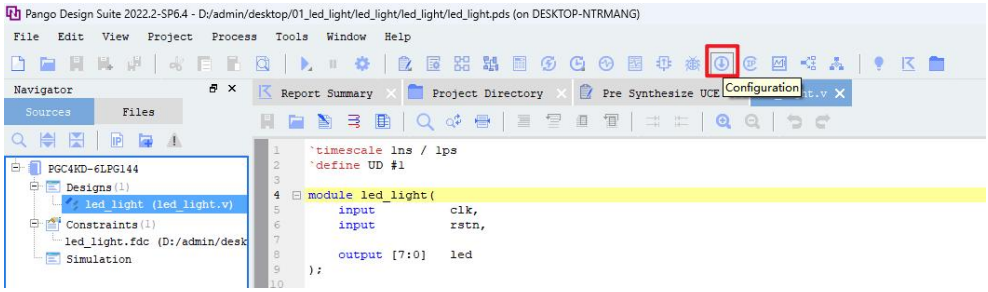


图 3-1.6-3 下载 bit 流流程图 1

step2：点击 Scan Device。

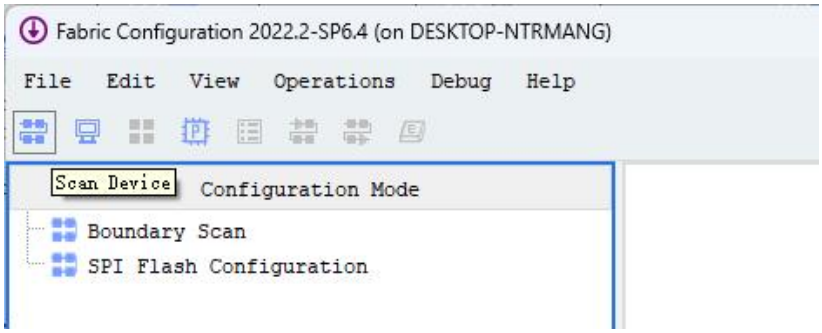


图 3-1.6-4 下载 bit 流流程图 2

step3：扫描成功后，选择.sbit 文件，点击 OPEN。

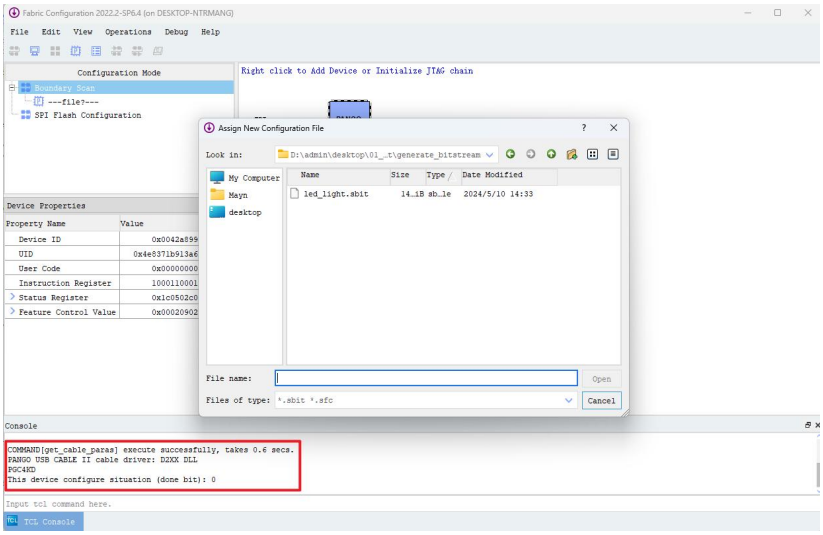


图 3-1.6-5 下载 bit 流流程图 3

step4：右键点击蓝色区域，点击 Program，进行位流的下载。

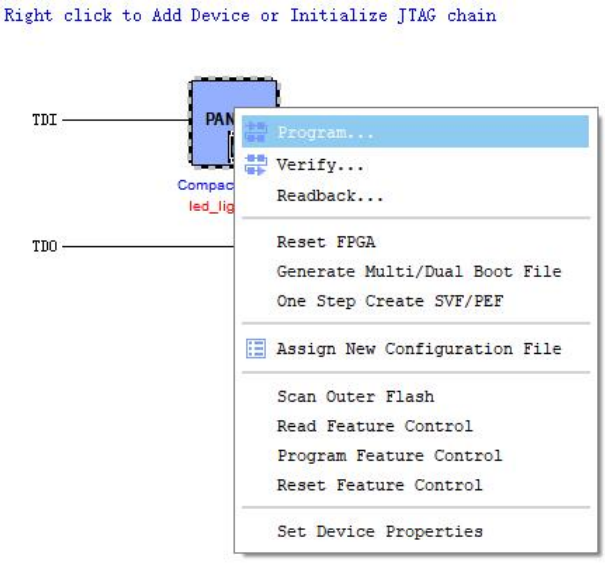


图 3-1.6-6 下载 bit 流流程图 4

补充说明：PGC4KD 内置 flash，因此右键点击带有 flash 字样的黑色区域，点击 Program，可以进行 flash 的固化操作。

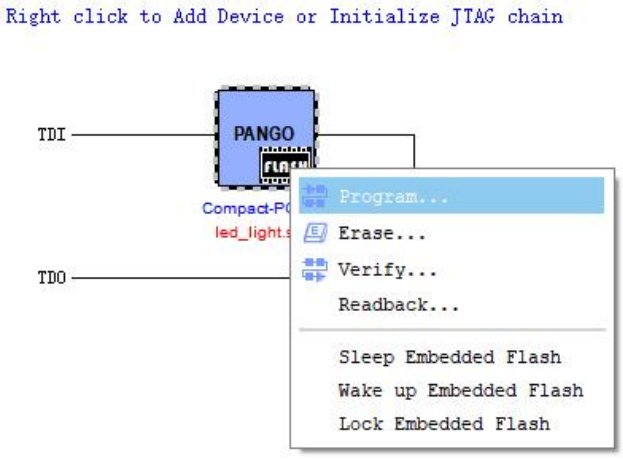
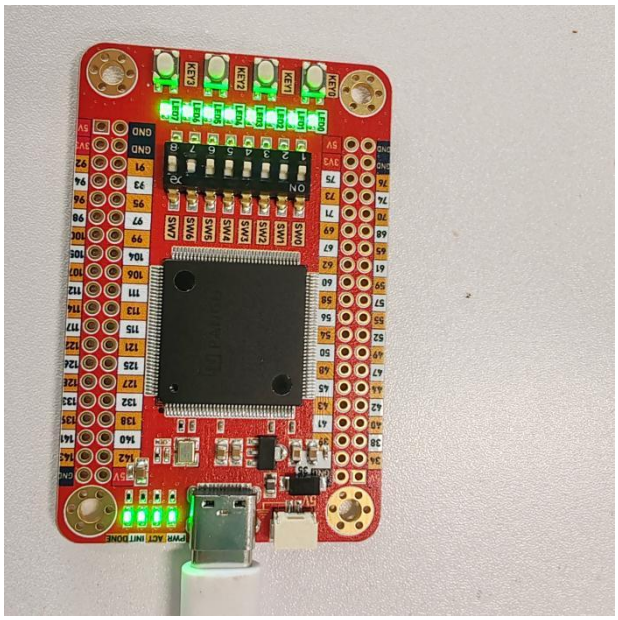
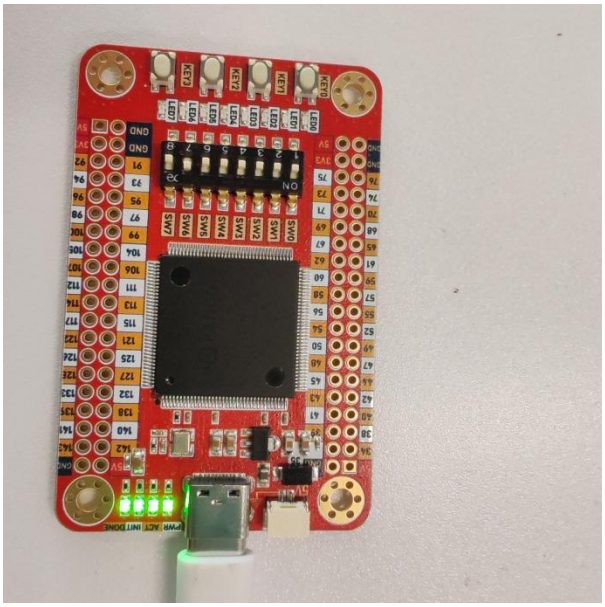


图 3-1.6-7 下载 bit 流流程图 5

1.7、观察实验现象



(a)0.5s 灯亮



(b)0.5s 灯灭

图 3-1.7-1 实验现象效果图。

2、流水灯实验

2.1、实验目的

掌握流水灯原理并实现流水灯。

2.2、实验要求

流水灯：8 个 LED 灯以 0.5s 间隔交替闪烁。

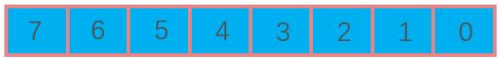
2.3、实验原理

使用计时器计时 0.5s，每隔 0.5s 改变依次 LED 灯的状态。将 8 个 LED 灯流水式的点亮；

在 C 语言中做流水灯的实验需要用到一个中间变量(代码如图 3-37，数据位的搬移如图 3-38)：

```
1   int data;
2   int temp1,temp2;
3
4   data = 0x01;
5   temp1 = data >> 7;
6   temp2 = data <<1;
7   data = temp1 | temp2;
```

图 3-2.3-1 C 语言代码示意图



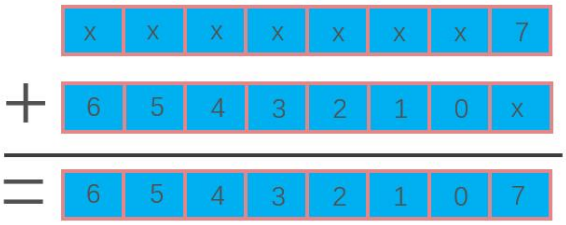


图 3-2.3-2 数据位搬移示意图

在 FPGA 的开发中使用的时基于硬件的硬件描述语言，实验中使用的硬件描

述语言为 verilog 语言，verilog 的处理单位为 1bit；8bit 的位宽数据可看作 8 个独立的信号线，这 8 个信号线之间的排序及相互之间的赋值可以随意组合；代码如下：

```
reg [7:0] data;
always @(posedge clk)
data <= {data[6:0],data[7]};
//或：
wire [7:0] data1;
wire [7:0] out;
assign out = {data1[6:0],data1[7]};
```

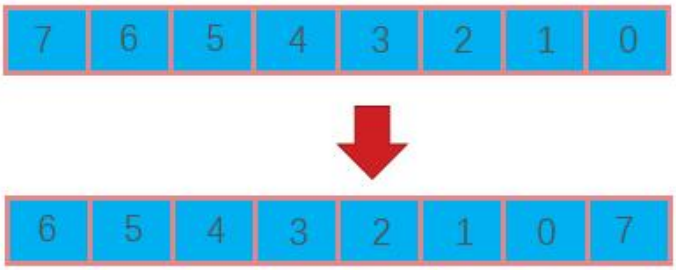


图 3-2.3-3 数据位移位示意图

2.4、实验源码设计

计时器计时 0.5s 时归 0 重新计时，并且此时通过移位变更 LED 灯状态。

timescale 1ns / 1ps

```
define UD #1

module led_water(
    input       clk    ,
    input       rstn   ,
    output  [7:0]led
    );
//=============================================================
reg [25:0] led_light_cnt    = 26'd0         ;
```

```
11.     reg [ 7:0] led_status      = 8'b0000_0001  ;
12.  //time counter
13.     always @(posedge clk)
14.     begin
15.         if(!rstn)
16.             led_light_cnt <= UD 26'd0;
17.         else if(led_light_cnt == 26'd24_999_999)
18.             led_light_cnt <= UD 26'd0;
19.         else
20.             led_light_cnt <= UD led_light_cnt + 26'd1;
21.     end
22.  //led status change
23.     always @(posedge clk)
24.     begin
25.         if(!rstn)
26.             led_status <= UD 8'b0000_0001;
27.         else if(led_light_cnt == 25'd24_999_999)
28.             led_status <= UD {led_status[6:0],led_status[7]};
29.  end
30.
31.     assign led = led_status;
32.  endmodule
33.
```

2.5、实验现象

打开烧录程序界面，设备连接好后连接 JTAG，选择要下载的程序的 sbit，如下所示：

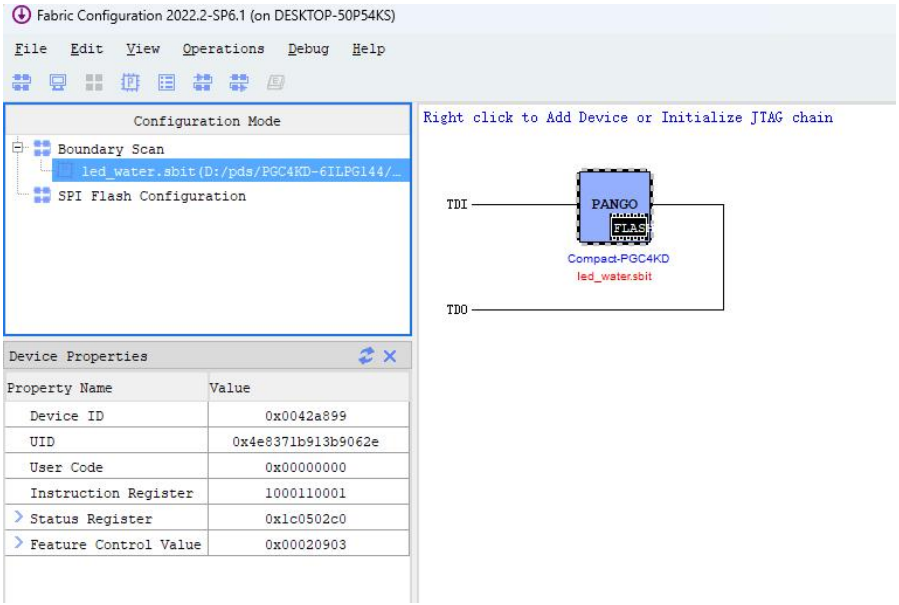
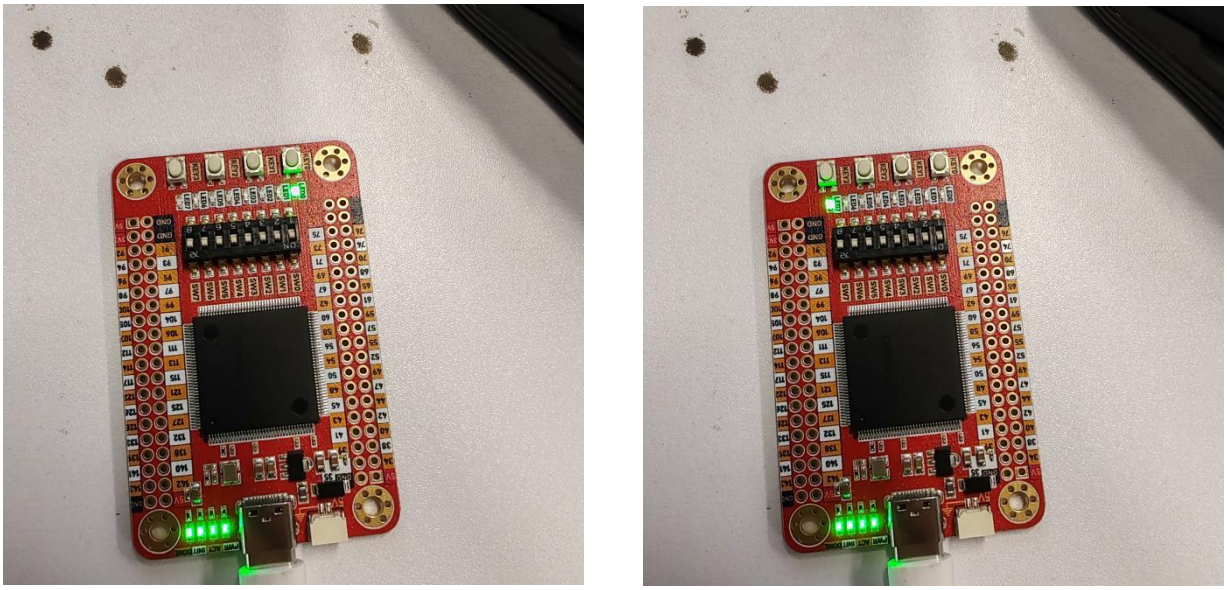


图 3-2.5-1 烧录程序界面示意图

下载完成后可以观察到流水灯：8 个 LED 灯以 0.5s 间隔交替闪烁。



(a)LED1 亮灯　　(b)LED8 亮灯

图 3-2.5-2 流水灯交替闪烁效果图

3、按键拨码开关控制 LED 灯实验

3.1、实验目的

实验按键与拨码开关对 LED 灯的控制，掌握按键消抖。

3.2、实验要求

拨码开关控制 LED 灯的亮灭，按键控制流水灯的流水模式。

3.3、实验原理

逻辑派 Z1 开发板配置了 8 个拨码开关，电路设计上，当拨码开关拨上时，IO 为低电平，未拨上时 IO 默认高电平。

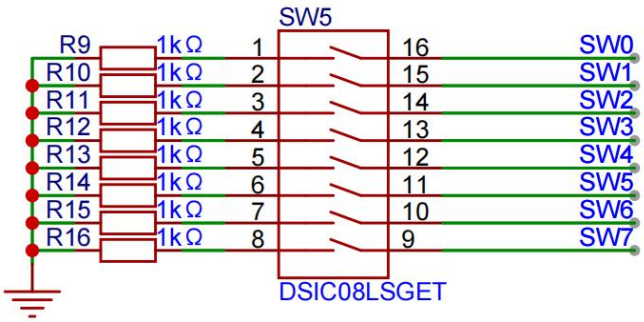


图 3-3.3-1 拨码开关电路原理图

逻辑派 Z1 开发板配置了 4 个用户按键，IO 默认高电平，按键按下后 IO 为低电平。

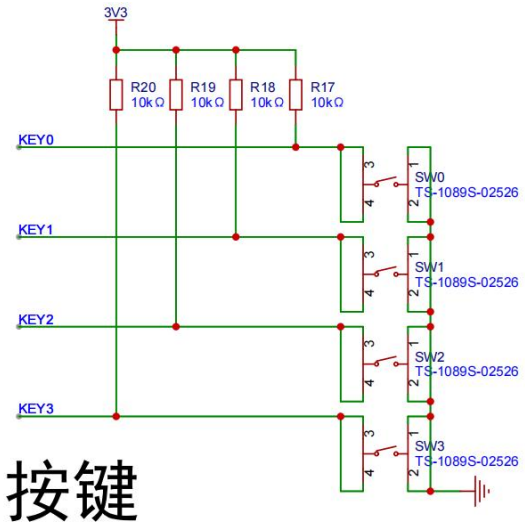


图 3-3.3-2 用户按键电路原理图

由于机械式弹片按键在按下或松开时会有机械抖动，导致在按下或松开时按

键的状态不稳定，在按键信号状态快速的变化时，使用按键作为输入信号，如果采集了按键抖动时的状态，会导致工程运行出现不可控的变化，故而我们需要将这段时间的抖动信号给滤除掉，此过程叫做按键消抖。

当按键按下时前后抖动时间约为 5～10ms，前后抖动共在 20ms。

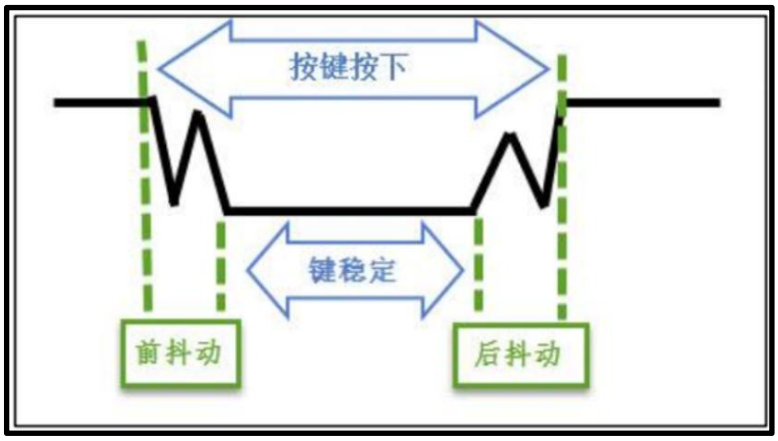


图 3-3.3-3 按键抖动示意图

当检测到按键状态变化时，对按键信号进行 20ms 的锁存，就可以避开前抖动与后抖动这段区间。

设计一个信号当按键输入信号状态变化时进行拉高操作，此时计数器开始计时，当计数满 20ms 时，在将此信号拉低，在此信号拉高时，对按键信号进行锁存，从而避免按键抖动。

3.4、实验源码设计

顶层代码：

使用计数器计时 0.5s，以 0.5s 为间隔使 LED 灯交替闪烁，并且计数按键按下的次数切换 LED 灯的流水模式。

```
1.  timescale 1ns / 1ps
2.  define UD #1
3.
4.  module key_sw_led(
5.      input         clk,
6.      input         rstn,
7.      input  [7:0]   sw ,
8.      input  [2:0]   key ,
```

```
9.      output [7:0]   led
10.   );
11.  //-------------------------------------------------------------------------
12.  wire [7:0]sw_bed /* synthesis PAP_MARK_DEBUG="true" */;
13.  wire [2:0]key_bed /* synthesis PAP_MARK_DEBUG="true" */;
14.  reg  [2:0]key_reg /* synthesis PAP_MARK_DEBUG="true" */;
15.  reg  [2:0]key_fall /* synthesis PAP_MARK_DEBUG="true" */;
16.  reg  [1:0]cnt /* synthesis PAP_MARK_DEBUG="true" */;
17.
18.  btn_deb#(
19.     .BTN_WIDTH (4'd8),
20.     .BTN_DELAY (20'd800_000)
21.  )btn_deb
22.  (
23.     .clk          (clk     ) ,
24.     .btn_in       (sw      ) ,
25.     .btn_deb      (sw_bed  )
26.  );
27.
28.  btn_deb#(
29.     .BTN_WIDTH (4'd3),
30.     .BTN_DELAY (20'd800_000)
31.  )btn_deb_1
32.  (
33.     .clk          (clk     ) ,
34.     .btn_in       (key     ) ,
35.     .btn_deb      (key_bed  )
36.  );
37.
38.  always @(posedge clk ) begin
39.     key_reg <= key_bed ;
40.     key_fall <= ((~key_bed)&(key_reg));
41.  end
42.
43.  always @(posedge clk ) begin
44.     cnt <= 2'd0 ;
45.  //    if (key_fall[0]||key_fall[1]||key_fall[2])
```

```
46.     if (|key_fall)
47.         cnt <= cnt + 2'b1 ;
48.     else
49.         cnt <= cnt ;
50. end
51. //=============================================================
==================
52. //reg and wire
53.
54.     reg [25:0] led_light_cnt/* synthesis PAP_MARK_DEBUG="true" */;
55.     reg [ 7:0] led_status/* synthesis PAP_MARK_DEBUG="true" */;
56.     reg tran_flag /* synthesis PAP_MARK_DEBUG="true" */;
57.     //  time counter
58.     always @(posedge clk)
59.     begin
60.         if(!rstn)
61.             led_light_cnt <= UD 26'd0;
62.         else if(led_light_cnt == 26'd24_999_999)
63.             led_light_cnt <= UD 26'd0;
64.         else
65.             led_light_cnt <= UD led_light_cnt + 26'd1;
66.     end
67.
68.     always @(posedge clk ) begin
69.             tran_flag <= 1'b0 ;
70.         if (led_light_cnt == 25'd24_999_999)
71.             tran_flag <= 1'b1 ;
72.         else
73.             tran_flag <= 1'b0 ;
74. end
75.
76.     // led status change
77.     always @(posedge clk)
78.     begin
79.         if(!rstn)
80.             led_status <= UD 8'b0000_0001;
81.         else if (tran_flag)begin
```

```
82.             case (cnt)
83.               0 : led_status <= UD {led_status[6:0],led_status[7]};
84.               1 : led_status <= UD {led_status[0],led_status[7:1]};
85.               2 : led_status <= UD {led_status[5:0],led_status[7:6]};
86.               3 : led_status <= UD {led_status[1:0],led_status[7:2]};
87.                default: led_status <= UD 8'b0000_0001;
88.             endcase
89.         end
90. end
91.
92.     assign led = (led_status & (~sw_bed));
93. endmodule
```

按键消抖模块代码：

当检测到按键输入信号有状态变化时，对按键信号进行锁存 20ms，以此避免按键按下的抖动状态。

```
1.  timescale 1ns / 1ps
2.  define UD #1
3.  module btn_deb#(
4.      parameter               BTN_WIDTH = 4'd8,
5.      parameter               BTN_DELAY = 20'hF423F
6.  )
7.  (
8.      input                   clk,  //
9.      input     [BTN_WIDTH-1:0] btn_in,
10.
11.     output reg [BTN_WIDTH-1:0] btn_deb
12. );
13.
14.     reg [19:0]          cnt[BTN_WIDTH-1:0];
15.     reg [BTN_WIDTH-1:0] flag;
16.
17.     reg [BTN_WIDTH-1:0] btn_in_reg;
18.     always @(posedge clk)
19.     begin
```

```
20.       btn_in_reg <= UD btn_in;
21.   end
22.
23.     genvar i;
24.     generate
25.     begin
26.       for(i=0;i<BTN_WIDTH;i=i+1)
27.       begin
28.         always @(posedge clk)
29.           begin
30.             if (btn_in_reg[i] ^ btn_in[i]) //The interval at which the edge of a key begins to j
itter is marked
31.               flag[i] <= UD 1'b1;
32.             else if (cnt[i]== BTN_DELAY)//Last for 20ms and then return to zero
33.               flag[i] <= UD 1'b0;
34.             else
35.                 flag[i] <= UD flag[i];
36.           end
37.
38.         always @(posedge clk)
39.           begin
40.             if(cnt[i]== BTN_DELAY)      //Last for 20ms and then return to zero
41.               cnt[i] <= UD 20'd0;
42.             else if(flag[i])          //Count when the jitter interval is valid
43.               cnt[i] <= UD cnt[i] + 1'b1;
44.            else           //The non-jitter interval remains 0
45.               cnt[i] <= UD 20'd0;
46.           end
47.
48.         always @(posedge clk)
49.           begin
50.             if(flag[i]) //Jitter interval, anti-jitter output maintained
51.
52.               btn_deb[i] <= UD btn_deb[i];
53.
54.             else    //In the non-jitter interval, the key state is transferred to the anti-jitter out
put
```

```
55.                 btn_deb[i] <= UD btn_in[i];
56.
57.             end
58.         end
59.     end
60. endgenerate
61.
62. endmodule
```

3.5、实验现象

打开烧录程序界面，设备连接好后连接 JTAG，选择要下载的程序的 sbit，如下所示：

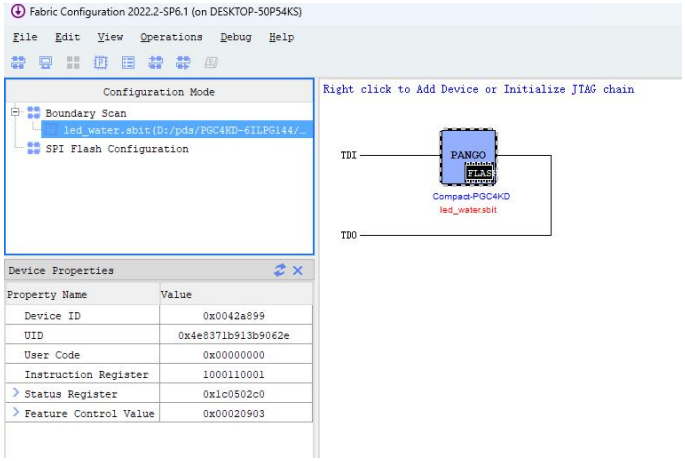


图 3-3.5-1 烧录程序界面示意图

拨码开关拨上时对应 LED 灯亮，按键 KEY0 为复位按键，按下后返回初始状 s 态 LED0 亮，KEY1、KEY2、KEY3 按键控制 LED 灯流水模式，LED 灯流水状态共 4 中：逐个向右流水、逐个向左流水，每次跳一个 LED 灯向右流水，每次跳一个 LED 灯向左流水。

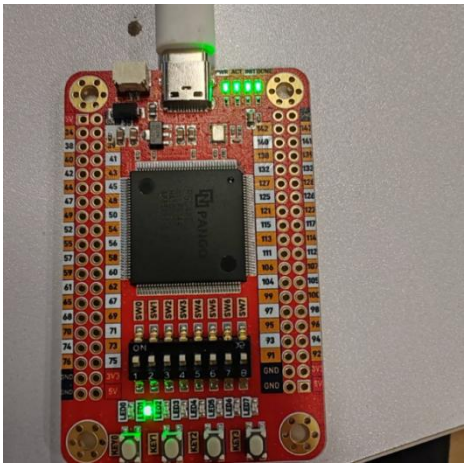


图 3-3.5-2 拨码开关控制流水灯效果图

4、二选一电路实验例程

4.1、实验目的

实现二选一选择器代码的编写与仿真验证

4.2、实验原理

使用一个 sel 信号根据其值对输入的两路信号进行选择。

4.3、代码设计

RTL 代码如下：

```
1.  module mux2to1(
2.      input wire a,       // 输入 a
3.      input wire b,       // 输入 b
4.      input wire sel,     // 选择信号
5.      output wire y       // 输出 y
6.  );
7.
8.      assign y = (sel) ? b : a; // 根据 sel 决定输出 a 还是 b
9.
10. endmodule
```

代码实现了一个 2 选 1 多路选择器。根据输入的选择信号 sel，在两个输入信号 a 和 b 之间选择一个作为输出 y。当选择信号 sel 为低电平（0）时，输出 y 等于输入信号 a；当 sel 为高电平（1）时，输出 y 等于输入信号 b。

4.4、实验现象

```
VSIM 5> run -all
#  a = 0, b = 0, sel = 0, y = 0
#  a = 0, b = 1, sel = 0, y = 0
#  a = 1, b = 0, sel = 1, y = 0
#  a = 1, b = 1, sel = 1, y = 1
# ** Note: $stop    : D:/My_work/Workplace/mux2to1/prj/source/prj/tb_mux2to1.v(44)
#    Time: 40 ns  Iteration: 0  Instance: /tb_mux2to1
# Break in Module tb_mux2to1 at D:/My_work/Workplace/mux2to1/prj/source/prj/tb_mux2to1.v line 44
```

图 3-4.4-1 二选一电路实验现象结果展示图

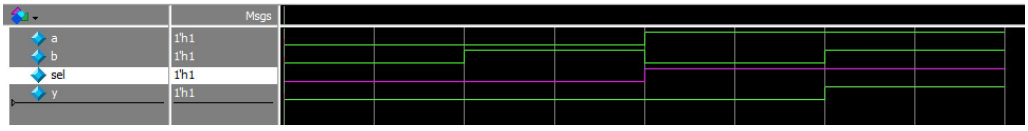


图 3-4.4-2 二选一电路实验结果波形图

结合 modelsim 命令行打印的信息与 wave 窗口的波形来看，实验符合预期，代码正确。

5、四选一电路实验例程

5.1、实验目的

实现四选一多路选择器代码的编写与仿真验证

5.2、实验原理

使用两位宽的 sel 信号根据其值对输入的四路信号进行选择。

5.3、代码设计

RTL 代码如下：

```
1.  module mux4to1(
2.      input wire a,       // 输入 a
3.      input wire b,       // 输入 b
4.      input wire c,       // 输入 c
5.      input wire d,       // 输入 d
6.      input wire [1:0] sel, // 两位选择信号
7.      output wire y       // 输出 y
8.  );
9.
10.     assign y = (sel == 2'b00) ? a :
11.                (sel == 2'b01) ? b :
12.                (sel == 2'b10) ? c : d;
13. endmodule
```

代码实现了一个 4 选 1 多路选择器。根据输入的选择信号 sel，在两个输入信号 a b c d 之间选择一个作为输出 y。当选择信号 sel 为 00 时，输出 y 等于输入信号 a；当 sel 为 01 时，输出 y 等于输入信号 b;当 sel 为 10 时，输出 y 等于输入信 c;当 sel 为 11 时，输出 y 等于输入信号 d。

测试代码如下：

```
1.  timescale 1ns / 1ps
2.
3.  module tb_mux4to1;
```

```
4.      // 定义仿真中的输入和输出信号
5.      reg a;
6.      reg b;
7.      reg c;
8.      reg d;
9.      reg [1:0] sel;
10.     wire y;
11.
12.     // 实例化 4 选 1 多路选择器
13.     mux4to1 uut (
14.         .a(a),
15.         .b(b),
16.         .c(c),
17.         .d(d),
18.         .sel(sel),
19.         .y(y)
20.     );
21.     //紫光全局复位原语,仿真需要添加
22.     GTP_GRS GRS_INST(
23.         .GRS_N(1'b1)
24.     );
25.     // 初始化和仿真信号激励
26.     initial begin
27.         // 打印波形信号头部
28.         $monitor("a = %b, b = %b, c = %b, d = %b, sel = %b, y = %b",  a, b, c, d, sel, y);

29.         // 初始化输入信号
30.         a = 0; b = 0; c = 0; d = 0; sel = 2'b00;
31.         #10;  // 等待 10ns
32.         a = 1; sel = 2'b00; // 选择 a
33.         #10;
34.         b = 1; sel = 2'b01; // 选择 b
35.         #10;
36.         c = 1; sel = 2'b10; // 选择 c
37.         #10;
```

```
38.         d = 1; sel = 2'b11; // 选择 d
39.         #10;
40.         // 结束仿真
41.         $stop;
42.     end
43.
44. endmodule
```

代码中通过 timescale 1ns / 1ns 设置时间单位和精度为 1 纳秒。定义了输入信号 a、b、c、d、sel 和输出信号 y。然后实例化待测试的多路选择器模块 mux4to1，并命名为 uut，将其输入和输出连接到相应信号上。在紫光系列 FPGA 中，为了仿真环境正常复位，添加了紫光 特定的全局复位原语。initial begin 块用于初始化信号和提供仿真激励，通过$monitor 输出信号变化，每隔 10 纳秒改变输入信号的值，测试不同条件下多路选择器的输出。仿真过程中，通过观察 a、b、c、d、sel 的变化，验证输出 y 的正确性，最后用$stop 结束仿真。

5.4、实验现象

```
# a = 0, b = 0, c = 0, d = 0, sel = 00, y = 0
# a = 1, b = 0, c = 0, d = 0, sel = 00, y = 1
# a = 1, b = 1, c = 0, d = 0, sel = 01, y = 1
# a = 1, b = 1, c = 1, d = 0, sel = 10, y = 1
# a = 1, b = 1, c = 1, d = 1, sel = 11, y = 1
# ** Note: $stop    : D:/My_work/Workplace/mux4to1/prj/source/prj/tb_mux4to1.v(50)
#    Time: 50 ns  Iteration: 0  Instance: /tb_mux4to1
# Break in Module tb_mux4to1 at D:/My_work/Workplace/mux4to1/prj/source/prj/tb_mux4to1.v line 50
```

图 3-5.4-1 四选一电路实验现象结果展示图

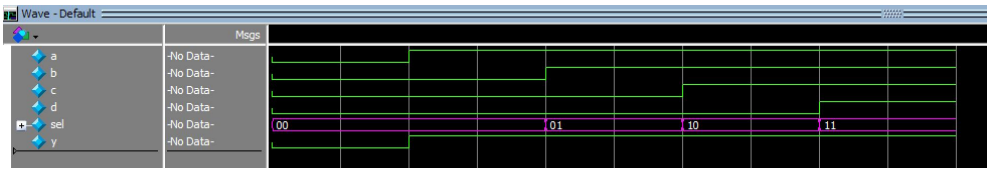


图 3-5.4-2 四选一电路实验结果波形图

结合 modelsim 命令行打印的信息与 wave 窗口的波形来看，实验符合预期，代码正确。

6、呼吸灯实验例程

6.1、实验目的

在逻辑派 Z1 开发板上利用 8 个 led 灯进行呼吸灯闪烁。

6.2、实验原理

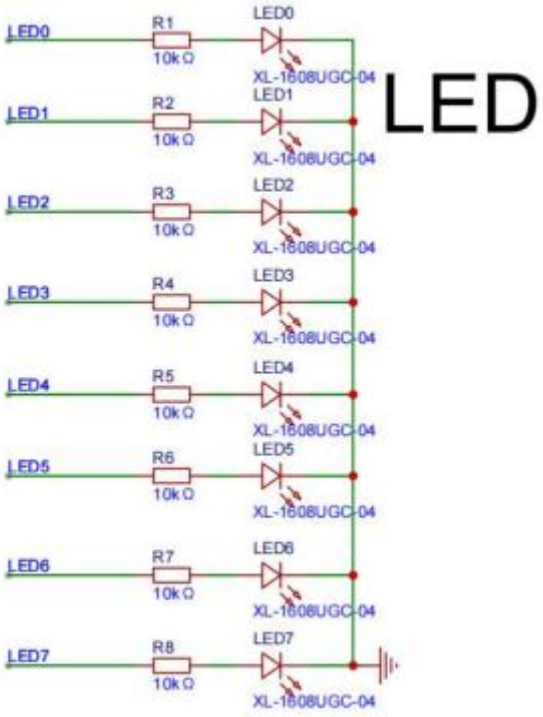


图 3-6.2-1 LED 电路原理图

由原理图可知，逻辑派 Z1 开发板板载 led 为高电平点亮。我们可以利用板卡输出 PWM 波来控制小灯亮灭，在相同时间段内的不同占空比，即在同样小时间段内，小灯亮的时间依次增加到最大后再依次减小，从而实现渐亮到渐灭的“呼吸”效果。

6.3、代码设计

我们可以使用简单的计数器原理实现呼吸灯功能，其具体代码如下：

```
1.  timescale 1ns/1ns
2.
3.  module breath_led
4.  #(
```

```
5.      parameter CNT_1US_MAX = 6'd49   ,
6.      parameter CNT_1MS_MAX = 10'd999 ,
7.      parameter CNT_1S_MAX  = 10'd999
8.  )
9.  (
10.     input   wire    sys_clk     ,  //系统时钟 50Mhz
11.     input   wire    sys_rst_n   ,  //全局复位
12.
13.     output  [7:0]   led_out        //输出信号，控制 led 灯
14. );
15.
16.
17.
18. //reg define
19. reg [5:0]   cnt_1us    ;
20. reg [9:0]   cnt_1ms    ;
21. reg [9:0]   cnt_1s     ;
22. reg         cnt_1s_en  ;
23. reg         led        ;
24.
25. assign led_out = {8{led}};
26.
27. //cnt_1us:1us 计数器
28. always@(posedge sys_clk or negedge sys_rst_n)
29.     if(sys_rst_n == 1'b0)
30.         cnt_1us <= 6'b0;
31.     else    if(cnt_1us == CNT_1US_MAX)
32.         cnt_1us <= 6'b0;
33.     else
34.         cnt_1us <= cnt_1us + 1'b1;
35.
36. //cnt_1ms:1ms 计数器
37. always@(posedge sys_clk or negedge sys_rst_n)
38.     if(sys_rst_n == 1'b0)
39.         cnt_1ms <= 10'b0;
40.     else    if(cnt_1ms == CNT_1MS_MAX && cnt_1us == CNT_1US_MAX)
```

```
41.         cnt_1ms <= 10'b0;
42.     else    if(cnt_1us == CNT_1US_MAX)
43.         cnt_1ms <= cnt_1ms + 1'b1;
44.
45.  //cnt_1s:1s 计数器
46.  always@(posedge sys_clk or negedge sys_rst_n)
47.     if(sys_rst_n == 1'b0)
48.         cnt_1s <= 10'b0;
49.     else    if(cnt_1s == CNT_1S_MAX && cnt_1ms == CNT_1MS_MAX
50.                                 && cnt_1us == CNT_1US_MAX)
51.         cnt_1s <= 10'b0;
52.     else    if(cnt_1ms == CNT_1MS_MAX && cnt_1us == CNT_1US_MAX)
53.         cnt_1s <= cnt_1s + 1'b1;
54.
55.  //cnt_1s_en:1s 计数器使能信号
56.  always@(posedge sys_clk or negedge sys_rst_n)
57.     if(sys_rst_n == 1'b0)
58.         cnt_1s_en <= 1'b0;
59.     else    if(cnt_1s == CNT_1S_MAX && cnt_1ms == CNT_1MS_MAX
60.                                     && cnt_1us == CNT_1US_MAX)
61.         cnt_1s_en <= ~cnt_1s_en;
62.
63.  //led:输出信号连接到外部的 led 灯
64.  always@(posedge sys_clk or negedge sys_rst_n)
65.     if(sys_rst_n == 1'b0)
66.         led <= 1'b0;
67.     else    if((cnt_1s_en == 1'b1 && cnt_1ms < cnt_1s) ||
68.                         (cnt_1s_en == 1'b0 && cnt_1ms > cnt_1s))
69.         led <= 1'b0;
70.     else
71.         led <= 1'b1;
72.
73.  endmodule
```

代码中定义了一个微秒级、毫秒级和秒级的计数器。通过这三个计数器，系统时钟被精确分割为 1 微秒、1 毫秒和 1 秒的时间单位。每当微秒计数器达到

50 个时钟周期（即 1 微秒）时，它会复位并重新开始计数。类似地，当毫秒计数器达到 1000 微秒时，也会复位，1 秒计数器则是每过 1000 毫秒递增一次。

在 1 秒计数器中，使能信号 cnt_1s_en 会每过 1 秒时翻转一次，控制 LED 亮度的渐变方向。具体来说，当使能信号为高时，LED 逐渐变亮；当使能信号为低时，LED 逐渐变暗。输出信号 led_out 通过八位宽度的信号控制 LED 灯，使所有连接的 LED 灯同时亮暗变化，呈现出呼吸灯的效果。

通过这种逐渐增加和减少亮度的方式，LED 实现了类似呼吸的效果。

约束文件如下：

```
1.  create_clock -name {clk} [get_ports {sys_clk}] -period {20.000} -waveform {0.000 10.000}
2.  define_attribute {p:led_out[7]} {PAP_IO_DIRECTION} {OUTPUT}
3.  define_attribute {p:led_out[7]} {PAP_IO_LOC} {28}
4.  define_attribute {p:led_out[7]} {PAP_IO_VCCIO} {1.2}
5.  define_attribute {p:led_out[7]} {PAP_IO_STANDARD} {LVCMOS12}
6.  define_attribute {p:led_out[7]} {PAP_IO_DRIVE} {2}
7.  define_attribute {p:led_out[7]} {PAP_IO_NONE} {TRUE}
8.  define_attribute {p:led_out[7]} {PAP_IO_SLEW} {SLOW}
9.  define_attribute {p:led_out[6]} {PAP_IO_DIRECTION} {OUTPUT}
10. define_attribute {p:led_out[6]} {PAP_IO_LOC} {27}
11. define_attribute {p:led_out[6]} {PAP_IO_VCCIO} {1.2}
12. define_attribute {p:led_out[6]} {PAP_IO_STANDARD} {LVCMOS12}
13. define_attribute {p:led_out[6]} {PAP_IO_DRIVE} {2}
14. define_attribute {p:led_out[6]} {PAP_IO_NONE} {TRUE}
15. define_attribute {p:led_out[6]} {PAP_IO_SLEW} {SLOW}
16. define_attribute {p:led_out[5]} {PAP_IO_DIRECTION} {OUTPUT}
17. define_attribute {p:led_out[5]} {PAP_IO_LOC} {26}
18. define_attribute {p:led_out[5]} {PAP_IO_VCCIO} {1.2}
19. define_attribute {p:led_out[5]} {PAP_IO_STANDARD} {LVCMOS12}
20. define_attribute {p:led_out[5]} {PAP_IO_DRIVE} {2}
21. define_attribute {p:led_out[5]} {PAP_IO_NONE} {TRUE}
22. define_attribute {p:led_out[5]} {PAP_IO_SLEW} {SLOW}
23. define_attribute {p:led_out[4]} {PAP_IO_DIRECTION} {OUTPUT}
24. define_attribute {p:led_out[4]} {PAP_IO_LOC} {25}
25. define_attribute {p:led_out[4]} {PAP_IO_VCCIO} {1.2}
```

```
26. define_attribute {p:led_out[4]} {PAP_IO_STANDARD} {LVCMOS12}
27. define_attribute {p:led_out[4]} {PAP_IO_DRIVE} {2}
28. define_attribute {p:led_out[4]} {PAP_IO_NONE} {TRUE}
29. define_attribute {p:led_out[4]} {PAP_IO_SLEW} {SLOW}
30. define_attribute {p:led_out[3]} {PAP_IO_DIRECTION} {OUTPUT}
31. define_attribute {p:led_out[3]} {PAP_IO_LOC} {24}
32. define_attribute {p:led_out[3]} {PAP_IO_VCCIO} {1.2}
33. define_attribute {p:led_out[3]} {PAP_IO_STANDARD} {LVCMOS12}
34. define_attribute {p:led_out[3]} {PAP_IO_DRIVE} {2}
35. define_attribute {p:led_out[3]} {PAP_IO_NONE} {TRUE}
36. define_attribute {p:led_out[3]} {PAP_IO_SLEW} {SLOW}
37. define_attribute {p:led_out[2]} {PAP_IO_DIRECTION} {OUTPUT}
38. define_attribute {p:led_out[2]} {PAP_IO_LOC} {23}
39. define_attribute {p:led_out[2]} {PAP_IO_VCCIO} {1.2}
40. define_attribute {p:led_out[2]} {PAP_IO_STANDARD} {LVCMOS12}
41. define_attribute {p:led_out[2]} {PAP_IO_DRIVE} {2}
42. define_attribute {p:led_out[2]} {PAP_IO_NONE} {TRUE}
43. define_attribute {p:led_out[2]} {PAP_IO_SLEW} {SLOW}
44. define_attribute {p:led_out[1]} {PAP_IO_DIRECTION} {OUTPUT}
45. define_attribute {p:led_out[1]} {PAP_IO_LOC} {22}
46. define_attribute {p:led_out[1]} {PAP_IO_VCCIO} {1.2}
47. define_attribute {p:led_out[1]} {PAP_IO_STANDARD} {LVCMOS12}
48. define_attribute {p:led_out[1]} {PAP_IO_DRIVE} {2}
49. define_attribute {p:led_out[1]} {PAP_IO_NONE} {TRUE}
50. define_attribute {p:led_out[1]} {PAP_IO_SLEW} {SLOW}
51. define_attribute {p:led_out[0]} {PAP_IO_DIRECTION} {OUTPUT}
52. define_attribute {p:led_out[0]} {PAP_IO_LOC} {21}
53. define_attribute {p:led_out[0]} {PAP_IO_VCCIO} {1.2}
54. define_attribute {p:led_out[0]} {PAP_IO_STANDARD} {LVCMOS12}
55. define_attribute {p:led_out[0]} {PAP_IO_DRIVE} {2}
56. define_attribute {p:led_out[0]} {PAP_IO_NONE} {TRUE}
57. define_attribute {p:led_out[0]} {PAP_IO_SLEW} {SLOW}
58. define_attribute {p:sys_clk} {PAP_IO_DIRECTION} {INPUT}
59. define_attribute {p:sys_clk} {PAP_IO_LOC} {5}
60. define_attribute {p:sys_clk} {PAP_IO_VCCIO} {1.2}
61. define_attribute {p:sys_clk} {PAP_IO_STANDARD} {LVCMOS12}
62. define_attribute {p:sys_clk} {PAP_IO_PULLUP} {TRUE}
```

```
63.  define_attribute {p:sys_rst_n} {PAP_IO_DIRECTION} {INPUT}
64.  define_attribute {p:sys_rst_n} {PAP_IO_LOC} {20}
65.  define_attribute {p:sys_rst_n} {PAP_IO_VCCIO} {1.2}
66.  define_attribute {p:sys_rst_n} {PAP_IO_STANDARD} {LVCMOS12}
67.  define_attribute {p:sys_rst_n} {PAP_IO_PULLUP} {TRUE}
```

6.4、实验现象

逻辑派 Z1 开发板上 8 个 led 灯以两秒为一个周期进行闪烁，形成呼吸灯的效果。

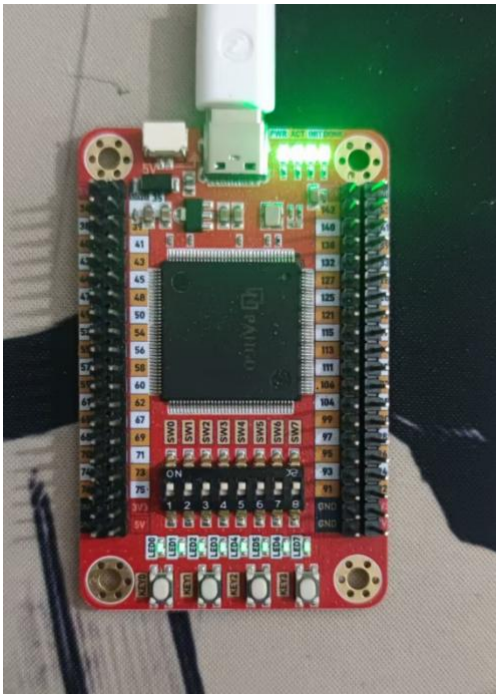


图 3-6.4-1 呼吸灯实验现象效果图 1

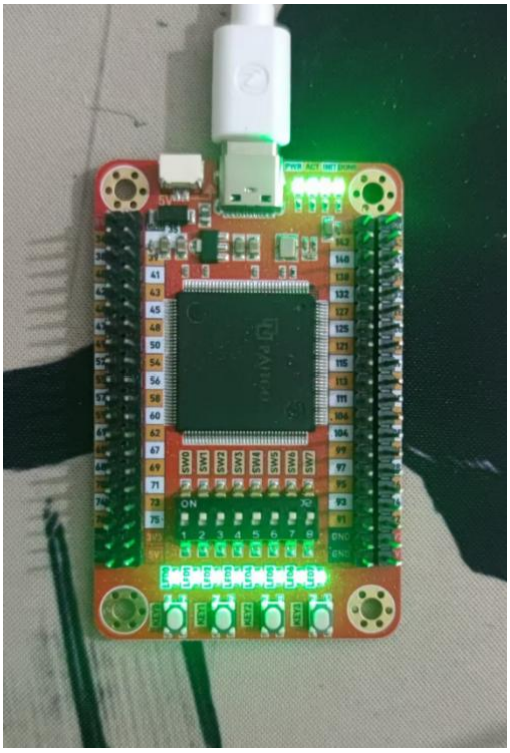


图 3-6.4-2 呼吸灯实验现象效果图 2

7、串口回环实验例程

7.1、实验目的

在逻辑派 Z1 开发板实现串口回环功能，上位机发送数据给逻辑派 Z1 开发板，逻辑派 Z1 开发板能够返回相应的数据。

7.2、实验原理

串口通信是一种常见的点对点通信方式，通常用于设备之间的数据传输。串口通信通过串行的方式逐位传输数据，即一位一位地将数据按顺序从发送端传输到接收端，具有结构简单、成本低的优点。

7.2.1、串口通讯原理

1. 串行传输：数据按位依次从发送端传输到接收端，而不是同时传输多个数据位。

2. 数据帧格式：数据一般是按帧的方式传输，一个典型的数据帧包括：

起始位：用于通知接收端即将有数据到来。

数据位：实际传输的数据，通常是 8 位。

校验位（可选）：用于检验数据是否在传输过程中出现错误。

停止位：用于通知接收端一帧数据已经结束。

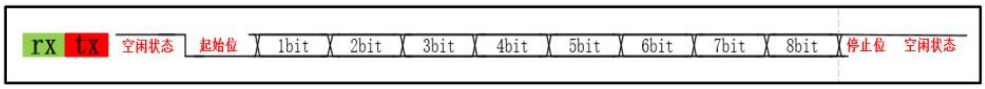


图 3-7.2-1 数据帧结构图

7.2.2、通信流程

发送方准备好数据并通过 TXD 发送；

接收方通过 RXD 接收数据；

数据在传输时，按预设的帧格式进行传输，包括起始位、数据位、校验位和停止位。

7.3、代码设计

uart_loop 顶层代码如下:

```
1.  module uart_loop(
2.      input                           clk                     ,//外部50MHz时
    钟
3.      input                           rst_n                   ,//系外部复位信
    号，低有效
4.
5.      //UART端口
6.      input                           uart_rxd                ,//UART接收
7.      output                          uart_txd                 //UART发送
8.      );
9.
10. //parameter define
11.     parameter                       SYS_CLK_FREQ            = 50000000;
     //输入时钟50Mhz
12.     parameter                       UART_BPS                = 115200;     /
    /定义串口波特率
13.
14. //wire define
15.     wire                            uart_rx_finish          ;//UART接收完
    成信号
16.     wire            [   7: 0]       uart_rx_data            ;//UART接收数
    据
17.     wire                            uart_tx_busy            ;
18.
19. //接收模块
20. uart_rx #(
21.     .SYS_CLK_FREQ                   (SYS_CLK_FREQ           ),
22.     .UART_BPS                       (UART_BPS               )
23.     )
24.     uart_rx_inst(
25.     .clk                            (clk                    ),
26.     .rst_n                          (rst_n                  ),
27.     .uart_rxd                       (uart_rxd               ),
28.     .uart_rx_finish                 (uart_rx_finish         ),
29.     .uart_rx_data                   (uart_rx_data           )
30.     );
31.
32. //发送模块
33. uart_tx #(
34.     .SYS_CLK_FREQ                   (SYS_CLK_FREQ           ),
35.     .UART_BPS                       (UART_BPS               )
36.     )
37.     uart_tx_inst(
38.     .clk                            (clk                    ),
39.     .rst_n                          (rst_n                  ),
40.     .uart_tx_req                    (uart_rx_finish         ),
41.     .uart_tx_data                   (uart_rx_data           ),
42.     .uart_txd                       (uart_txd               ),
43.     .uart_tx_busy                   (uart_tx_busy           )
44.     );
45.
46. endmodule
```

在顶层模块中我们将串口发送模块（uart_tx）与串口接收模块（uart_rx）相连接，串口接收模块负责按照 uart 数据帧结构将串口数据解析为单个字节的 uart_rx_data，同时输出一个单字节接收完成的标志信号 uart_rx_finish。串口发送模块根据这些信号将 PC 端发送的数据再重新发送给 PC，实现串口数据回环

的效果，串口发送模块（uart_tx）与串口接收模块（uart_rx）我们会在下文详细介绍。

uart_rx 串口接收模块代码如下：

```
1.  module uart_rx(
2.      input                           clk                     ,
3.      input                           rst_n                   ,
4.
5.      input                           uart_rxd                ,//UART 接收端口
6.      output reg                      uart_rx_finish          ,//一帧数据接收完成
7.      output reg          [   7: 0]   uart_rx_data             //解析后的数据
8.      );
9.
10.     parameter                       SYS_CLK_FREQ            = 50000000;
11.     parameter                       UART_BPS                = 115200;
12.     localparam                      BAUD_CNT_MAX            = SYS_CLK_FREQ/UART_BPS;  //根据时钟和波特率计算波特计数器的值
13.
14. //reg define
15.     reg                             uart_rxd_d0              ;
16.     reg                             uart_rxd_d1              ;
17.     reg                             uart_rxd_d2              ;
18.     reg                             work_en                  ;//接收过程标志信号
19.     reg                 [   3: 0]   rx_data_cnt              ;//接收数据计数器
20.     reg                 [  15: 0]   baud_cnt                 ;//波特率计数器
21.     reg                 [   7: 0]   rx_data_reg              ;//接收数据寄存器
22.     wire                            start_flag               ;
23.
24. //检测下降沿(起始位)
25.     assign                          start_flag              = uart_rxd_d2 & (~uart_rxd_d1) & (~work_en);
26.
27. //多级寄存器缓存消除亚稳态
28. always @(posedge clk or negedge rst_n) begin
29.     if(!rst_n) begin
30.         uart_rxd_d0 <= 1'b0;
31.         uart_rxd_d1 <= 1'b0;
32.         uart_rxd_d2 <= 1'b0;
33.     end
34.     else begin
35.         uart_rxd_d0 <= uart_rxd;
36.         uart_rxd_d1 <= uart_rxd_d0;
37.         uart_rxd_d2 <= uart_rxd_d1;
38.     end
39. end
40.
41. //tx_flag
42. always @(posedge clk or negedge rst_n) begin
```

```
43.     if(!rst_n)
44.         work_en <= 1'b0;
45.     else if(start_flag)                                    //检测到起始
    位
46.         work_en <= 1'b1;
47.     else if((rx_data_cnt == 4'd9) && (baud_cnt == BAUD_CNT_MAX/2 - 1'b1))
48.         work_en <= 1'b0;
49.     else
50.         work_en <= work_en;
51. end
52.
53. //波特率的计数器赋值
54. always @(posedge clk or negedge rst_n) begin
55.     if(!rst_n)
56.         baud_cnt <= 16'd0;
57.     else if(work_en) begin
58.         if(baud_cnt < BAUD_CNT_MAX - 1'b1)
59.             baud_cnt <= baud_cnt + 16'b1;
60.         else
61.             baud_cnt <= 16'd0;
62.     end
63.     else
64.         baud_cnt <= 16'd0;
65. end
66.
67. //接收 bit 计数器
68. always @(posedge clk or negedge rst_n) begin
69.     if(!rst_n)
70.         rx_data_cnt <= 4'd0;
71.     else if(work_en) begin
72.         if(baud_cnt == BAUD_CNT_MAX - 1'b1)
73.             rx_data_cnt <= rx_data_cnt + 1'b1;
74.         else
75.             rx_data_cnt <= rx_data_cnt;
76.     end
77.     else
78.         rx_data_cnt <= 4'd0;                               //接收完成计数
    器清零
79. end
80.
81. //接收端口赋值
82. always @(posedge clk or negedge rst_n) begin
83.     if(!rst_n)
84.         rx_data_reg <= 8'b0;
85.     else if(work_en) begin
86.         if(baud_cnt == BAUD_CNT_MAX/2 - 1'b1) begin        //在波特计数器
    中间寄存数据，以得到稳定数据
87.             if (rx_data_cnt >= 4'd1 && rx_data_cnt <= 4'd8) begin
88.                 rx_data_reg[rx_data_cnt - 1] <= uart_rxd_d2;        // 动态索引更
    新对应位
89.             end
90.         end
91.         else
92.             rx_data_reg <= rx_data_reg;
93.     end
94.     else
95.         rx_data_reg <= 8'b0;
96. end
97.
98.
99. //给出数据和有效信号
100. always @(posedge clk or negedge rst_n) begin
```

```
101.     if(!rst_n) begin
102.         uart_rx_finish <= 1'b0;
103.         uart_rx_data <= 8'b0;
104.     end
105.     else if(rx_data_cnt == 4'd9 && baud_cnt == BAUD_CNT_MAX/2 - 1'b1) begin
106.         uart_rx_finish <= 1'b1     ;                                    //拉高接收完
    成信号
107.         uart_rx_data <= rx_data_reg;                                    //取出解析完
    成数据
108.     end
109.     else begin
110.         uart_rx_finish <= 1'b0;
111.         uart_rx_data <= uart_rx_data;
112.     end
113. end
114.
115. endmodule
```

模块的端口描述如下表：

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
uart_rxd	input	1	UART 接收端口输入信号
uart_rx_data	output	8	解析后的串口数据输出信号

模块的第 25 行是一个经典的边沿检测电路，通过检测串口接收端 uart_rxd 的下降沿来捕获起始位。一旦检测到起始位，输出一个时钟周期的脉冲 start_flag，并进入串口接收过程。串口接收状态用 work_en 来标志，work_en 为高标志着串口接收过程正在进行，此时启动系统时钟计数器 baud_cnt 与接收数据计数器 rx_data_cnt。这里需要明白，我们不知道 uart_rxd 信号何时会跳变，也就是说这是一个异步信号，异步信号会带来亚稳态问题。因此，对于异步信号，我们需要进行异步处理，常用的异步处理方式是打拍处理。uart_rxd_d0 是打第一拍，uart_rxd_d1 打第二拍，uart_rxd_d2 打第三拍。通常打两拍就基本上能避免亚稳态问题，但这里使用了三级寄存器来确保更高的稳定性。

由第 11 行的公式 BAUD_CNT_MAX = SYS_CLK_FREQ / UART_BPS 可知，BAUD_CNT_MAX 为当前波特率下，串口传输一位所需要的系统时钟周期数。因此，baud_cnt 从零计数到 BAUD_CNT_MAX - 1 时，串口刚好完成一位

数据的传输。由于接收数据计数器 rx_data_cnt 在每次 baud_cnt 计数到 BAUD_CNT_MAX - 1 时加 1，因此可以通过 rx_data_cnt 的值来判断串口当前传输的是第几位数据。

从第 53 行到第 84 行，根据 baud_cnt 的值将 uart_rxd 接收端口的数据寄存到接收数据寄存器 rx_data_reg 对应的位，从而实现接收数据的串并转换。具体来说，当 baud_cnt 计数到 BAUD_CNT_MAX / 2 - 1 时寄存接收端口数据，这是因为计数到数据中间时的采样结果最稳定。当 rx_data_cnt 跳变到 9 时，一帧串口数据解析完成，此时拉高一个时钟周期的 uart_rx_finish，并将数据赋值给 uart_rx_data，同时 work_en 拉低，一帧串口数据解析完成

uart_tx 串口发送模块代码如下：

```
1.  module uart_tx(
2.      input                           clk                     ,
3.      input                           rst_n                   ,
4.      input                           uart_tx_req             ,//发送使能请
    求
5.      input               [   7: 0]   uart_tx_data            ,//UART 要发送
    的数据
6.
7.      output reg                      uart_txd                ,//UART 发送端
    口
8.      output reg                      uart_tx_busy             //发送忙
9.      );
10.
11. //parameter define
12.     parameter                       SYS_CLK_FREQ            = 50000000;
13.     parameter                       UART_BPS                = 115200;
14.     localparam                      BAUD_CNT_MAX            = SYS_CLK_FREQ
    /UART_BPS; //根据时钟和波特率计算波特计数器的值
15.
16. //reg define
17.     reg                 [   7: 0]   tx_data_reg              ;//发送数据寄存
    器
18.     reg                 [   3: 0]   tx_data_cnt              ;//发送数据计数
    器
19.     reg                 [  15: 0]   baud_cnt                 ;//波特率计数
    器
20.
21.
22. //当 uart_tx_en 为高时，寄存输入的并行数据，并拉高发送忙信号
23. always @(posedge clk or negedge rst_n) begin
24.     if(!rst_n) begin
25.         tx_data_reg <= 8'b0;
26.         uart_tx_busy <= 1'b0;
27.     end
```

```
28.     else if(uart_tx_req == 'd1) begin
29.         tx_data_reg <= uart_tx_data;
30.         uart_tx_busy <= 1'b1;
31.     end
32.     else if(tx_data_cnt == 4'd9 && baud_cnt == BAUD_CNT_MAX - 1) begin
33.         tx_data_reg <= 8'b0;
34.         uart_tx_busy <= 1'b0;
35.     end
36.     else begin
37.         tx_data_reg <= tx_data_reg;
38.         uart_tx_busy <= uart_tx_busy;
39.     end
40. end
41. 
42. //波特率计数器
43. always @(posedge clk or negedge rst_n) begin
44.     if(!rst_n)
45.         baud_cnt <= 16'd0;
46.     else if(uart_tx_req)
47.         baud_cnt <= 16'd0;
48.     else if(uart_tx_busy) begin
49.         if(baud_cnt < BAUD_CNT_MAX - 1'b1)
50.             baud_cnt <= baud_cnt + 16'b1;
51.         else
52.             baud_cnt <= 16'd0;
53.     end
54.     else
55.         baud_cnt <= 16'd0;
56. end
57. 
58. //发送位计数器
59. always @(posedge clk or negedge rst_n) begin
60.     if(!rst_n)
61.         tx_data_cnt <= 4'd0;
62.     else if(uart_tx_req)
63.         tx_data_cnt <= 16'd0;
64.     else if(uart_tx_busy) begin
65.         if(baud_cnt == BAUD_CNT_MAX - 1'b1)
66.             tx_data_cnt <= tx_data_cnt + 1'b1;
67.         else
68.             tx_data_cnt <= tx_data_cnt;
69.     end
70.     else
71.         tx_data_cnt <= 4'd0;                                    //发送完成清
    零
72. end
73. 
74. //发送端口赋值
75. always @(posedge clk or negedge rst_n) begin
76.     if (!rst_n) begin
77.         uart_txd <= 1'b1;                                       // 空闲
78.     end else if (uart_tx_busy) begin
79.         if (tx_data_cnt == 4'd0)
80.             uart_txd <= 1'b0;                                   // 发送起始
    位
81.         else if (tx_data_cnt >= 4'd1 && tx_data_cnt <= 4'd8)
82.             uart_txd <= tx_data_reg[tx_data_cnt - 1];           // 发送数据
    位
83.         else if (tx_data_cnt == 4'd9)
84.             uart_txd <= 1'b1;                                   // 停止位
85.         else
86.             uart_txd <= 1'b1;                                   // 空闲
```

```
87.     end else begin
88.         uart_txd <= 1'b1;
89.     end
90. end
91.
92. endmodule
```

模块的端口描述如下表：

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
uart_tx_req	input	1	UART 发送模块的发送请求信号
uart_tx_data	intput	8	待发送的数据
uart_tx_busy	output	1	模块发送忙
uart_txd	output	1	UART 发送端口输出信号

串口发送模块与串口接收模块实现思想大致相同。相信读者在理解了 uart 协议的接收过程后，也能够看懂发送过程的逻辑。我们这里对代码中的关键信号进行说明。

在代码中，uart_tx_req 信号用于触发数据发送过程。当检测到 uart_tx_req 的上升沿后，模块将 uart_tx_data 端口上的待发送数据寄存到 tx_data_reg 中，并同时将 uart_tx_busy 信号拉高，表示发送器正处于繁忙状态。这意味着外部模块需要等待当前发送过程结束后，即 uart_tx_busy 信号再次变为低电平，才能启动新的发送请求。

baud_cnt 用于计算一个波特率周期内的时钟脉冲。当发送使能信号 uart_tx_req 为高时，波特率计数器会被清零，以开始一个新的数据帧的发送。在 uart_tx_busy 信号为高电平期间，波特率计数器会在每个时钟周期递增，直到达到一个完整的波特率周期（BAUD_CNT_MAX - 1），然后清零，准备发送下一个数据位。

tx_data_cnt 用于跟踪当前正在发送的数据位。当发送使能信号 uart_tx_req 为高时，发送位计数器会被清零。在 uart_tx_busy 信号为高电平期间，当波特率计

数器达到 BAUD_CNT_MAX - 1 时，发送位计数器递增，表示下一个数据位将要发送。

uart_txd 是串口发送的输出信号。在复位时，uart_txd 被设置为高电平，表示空闲状态。当 uart_tx_busy 信号为高电平时，根据 tx_data_cnt 的值，uart_txd 会依次输出起始位（低电平）、数据位和停止位（高电平）。具体而言：当 tx_data_cnt = 0：输出起始位，高电平拉低（低电平）。当 tx_data_cnt = 1 至 8：依次输出数据位，这些位从 tx_data_reg 中读取。当 tx_data_cnt = 9：输出停止位，低电平拉高（高电平）。

为了确保发送过程的可靠性，当 tx_data_cnt 计数到 9 并且 baud_cnt 也达到 BAUD_CNT_MAX - 1 时，表示一个完整的数据帧发送完成，uart_tx_busy 信号被拉低，模块进入空闲状态。

在整个发送过程中，uart_tx_busy 信号保持高电平，表示发送器正在忙碌。当发送完成时，uart_tx_busy 信号被拉低，模块进入空闲状态，准备接收下一个发送请求。

接下来我们直接将其下载进入板卡观察其实验现象。

7.4、实验现象

逻辑派 Z1 开发板搭载了 CH747T 芯片，CH374T 是一款功能强大的接口芯片，集成了 **JTAG 和 UART** 两种重要通信方式，分别用于调试、以及数据传输。根据原理图所示，其 TXD1 与 RXD1 引脚分别与 PGC4KD-6ILPG144 的 32、33 号引脚相连。我们根据原理图，设置以下 io 约束：

```
create_clock -name {clk} [get_ports {clk}] -period {20.000} -waveform {0.000 10.000}
define_attribute {p:uart_txd} {PAP_IO_DIRECTION} {OUTPUT}
define_attribute {p:uart_txd} {PAP_IO_LOC} {33}
define_attribute {p:uart_txd} {PAP_IO_VCCIO} {1.2}
define_attribute {p:uart_txd} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:uart_txd} {PAP_IO_DRIVE} {2}
define_attribute {p:uart_txd} {PAP_IO_NONE} {TRUE}
define_attribute {p:uart_txd} {PAP_IO_SLEW} {SLOW}
define_attribute {p:clk} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:clk} {PAP_IO_LOC} {5}
define_attribute {p:clk} {PAP_IO_VCCIO} {1.2}
define_attribute {p:clk} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:clk} {PAP_IO_PULLUP} {TRUE}

define_attribute {p:uart_rxd} {PAP_IO_DIRECTION} {INPUT}
```

```
define_attribute {p:uart_rxd} {PAP_IO_LOC} {32}
define_attribute {p:uart_rxd} {PAP_IO_VCCIO} {1.2}
define_attribute {p:uart_rxd} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:uart_rxd} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:rst_n} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:rst_n} {PAP_IO_LOC} {20}
define_attribute {p:rst_n} {PAP_IO_VCCIO} {1.2}
define_attribute {p:rst_n} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:rst_n} {PAP_IO_PULLUP} {TRUE}
```

将程序编译生成比特流后，只需要将板卡与电脑使用一根 usb 转 type-c 线材进行连接，即可实现供电、调试、串口通信功能。接下来将程序下载进入开发板，打开上位机串口调试助手，设置波特率为 115200，打开串口，发送 1234567890。观察到板卡返回相同数据，串口回环验证成功。

图 3-7.4-1 上位机验证实验结果

8、串口点亮 Led 灯实验例程

8.1、实验目的

上位机通过串口发送 8 位密码给逻辑派 Z1 开发板、开发板接收信息并按位点亮 8 个 led 灯。

8.2、实验原理

上位机通过串口发送 8 位数据给 FPGA。FPGA 接收串口数据将 8 位电灯数据寄存，并点亮相应位的 led 灯。

8.3、代码设计

串口点亮 led 灯 top_uart_led 顶层代码如下：

```
1.  module top_uart_led(
2.      input                           clk                     ,//系统时钟，50MHZ
3.      input                           rst_n                   ,//系统复位，低电平有效
4.      output              [   7: 0]   led                     ,//点亮呼吸灯
5.
6.      input                           uart_rx                  //串口接收的数据
7.  );
8.
9.      wire                            uart_rx_finish          ;//UART 接收完成信号
10.     wire                [   7: 0]   uart_rx_data            ;//UART 接收数据
11.
12.     uart_rx u_uart_rx(
13.     .clk                            (clk                    ),
14.     .rst_n                          (rst_n                  ),
15.     .uart_rxd                       (uart_rx                ),// UART 接收端口
16.     .uart_rx_finish                 (uart_rx_finish         ),// 一帧数据接收完成
17.     .uart_rx_data                   (uart_rx_data           ) // 解析后的数据
18.     );
19.
20.
21.     assign                          led                   = uart_rx_data;

22.
23. endmodule
```

顶层代码例化了串口接收模块 uart_rx、用于接收上位机发送的串口数据。然后将接收到的数据通过 assign 语句赋值给 8 个 led 灯。实现方法十分简单，其中串口接收模块 uart_rx 在之前的章节已经进行了详细的说明，如果对其工作逻辑不太清楚可以回到串口回环章节进行了解。

8.4、实验现象

我们将程序下载进入板卡，将板卡通过扩展 io 口连接上位机，打开串口调试助手，设置串口波特率为 115200，设置为 16 进制发送。发送数据 FF（二进制的 11111111），此时板卡 led 灯全部被点亮。发送数据 55（也就是二进制的 01010101），可以发现板卡 led 被交替点亮。

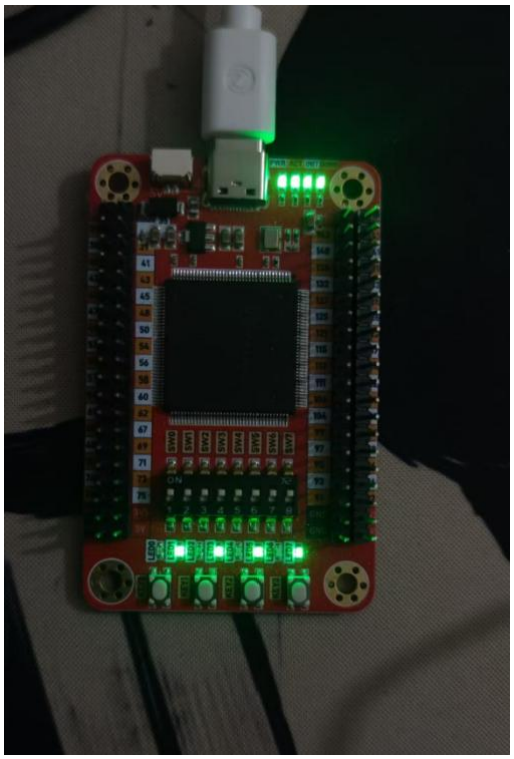


图 3-8.4-1 串口点亮 LED 灯实验效果图 1

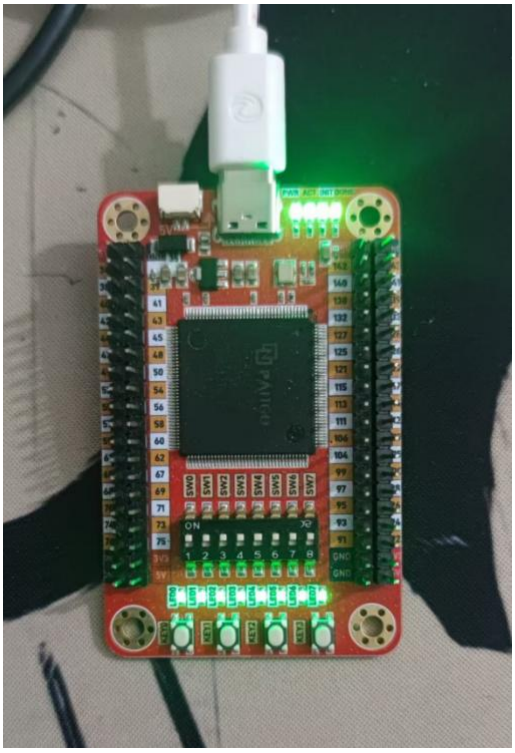


图 3-8.4-2 串口点亮 LED 灯实验效果图 2

9、串口密码锁实验例程

9.1、实验目的

上位机通过串口发送 8 位密码给逻辑派 Z1 开发板、开发板使用 8 个拨码开关输入密码，并使用一个按键进行验证。若输入密码与上位机发送的密码相同，则点亮 8 个 led 灯。若不同则不点亮 led 灯，同时支持上位机再次更改密码。

9.2、实验原理

上位机通过串口发送密码给 FPGA。FPGA 将密码解析并校验，如果接收成功则将 8 位密码寄存，在按键信号有效时将其与拨码开关 8 位数据进行比对。若比对成功则点亮 led 灯，若比对不成功则不点亮 led 灯。

9.3、代码设计

串口密码机 top_uart_locker 顶层代码如下：

```
1.  module top_uart_locker(
2.      input                           sys_clk                 ,//系统时钟，
    50MHZ
3.      input                           sys_rst_n               ,//系统复位，低
    电平有效
4.      input                           key_in                  ,//按键用来触发
    密码验证
5.      input               [   7: 0]   switch                  ,//八个开关用于
    输入密码
6.      output              [   7: 0]   led                     ,//密码验证通过
    点亮呼吸灯
7.
8.      input                           uart_rxd                ,//串口接收的数
    据
9.      output                          uart_txd                 //串口发送的数
    据
10. );
11.
12. //parameter define
13.     parameter                       PACKET_HEAD             = 8'h55 ;    /
    /定义数据包头
14.     parameter                       ERR_HEAD                = 8'hE0 ;    /
    /包头检测错误
15.     parameter                       ERR_CHECKSUM            = 8'hE1 ;    /
    /校验错误
16.     parameter                       PARSE_RIGHT               = 8'h00 ;
      //数据包解析正确
17.
18. //wire define
19.
20.     wire                [   7: 0]   uart_rx_data             ;//UART 接收端
    接收的数据
```

```
21.     wire                            rx_finish               ;//UART 接收一字节数据完成标志
22.     wire                            parse_done              ;//解包完成标志
23.     wire            [   7: 0]       parse_result            ;//解包后的结果,8'h00:解析正确 8'hEx:解析错误
24.
25.     wire                            uart_tx_en              ;//UART 发送端使能信号
26.     wire            [   7: 0]       uart_tx_data            ;//UART 发送端的数据
27.     wire                            uart_tx_busy            ;//UART 发送端忙率标志信号
28.     wire                            packet_tx_done          ;//一包数据全部发送完成标志
29.
30.     wire                            key_flag                ;//消抖后的按键信号
31.     wire            [   7: 0]       packet_data             ;//密码数据
32.
33.         uart_rx u_uart_rx(
34.     .clk                            (sys_clk                ),
35.     .rst_n                          (sys_rst_n              ),
36.     .uart_rxd                       (uart_rxd               ),// UART 接收端口
37.     .uart_rx_finish                 (rx_finish              ),// 一帧数据接收完成
38.     .uart_rx_data                   (uart_rx_data           ) // 解析后的数据
39.     );
40.
41. //例化数据包解析模块
42. packet_decode #(
43.     .PACKET_HEAD                    (PACKET_HEAD            ),
44.     .ERR_HEAD                       (ERR_HEAD               ),
45.     .ERR_CHECKSUM                   (ERR_CHECKSUM           ),
46.     .PARSE_RIGHT                       (PARSE_RIGHT      )           )
47.     u_packet_decode(
48.     .clk                            (sys_clk                ),
49.     .rst_n                          (sys_rst_n              ),
50.     .uart_rx_data                   (uart_rx_data           ),
51.     .uart_rx_done                   (rx_finish              ),
52.     .packet_tx_done                 (packet_tx_done         ),
53.
54.     .parse_done                     (parse_done             ),
55.     .parse_result                   (parse_result           ),
56.     .packet_data                    (packet_data            ) //密码数据
57. );
58.
59. locker u_locker(
60.     .clk                            (sys_clk                ),
61.     .rst_n                          (sys_rst_n              ),
62.     .key                            (key_flag               ),
63.     .switch                         (switch                 ),
64.     .packet_data                    (packet_data            ),
65.     .led                            (led                    )
66. );
67.
68.
69.
```

```
70. //例化数据包打包模块
71. packet_code #(
72.     .PACKET_HEAD                    (PACKET_HEAD            ),
73.
74.     .PARSE_RIGHT                      (PARSE_RIGHT      )           )
75.     u_packet_code(
76.     .clk                            (sys_clk                ),
77.     .rst_n                          (sys_rst_n              ),
78.     .uart_tx_busy                   (uart_tx_busy           ),
79.     .parse_done                     (parse_done             ),
80.     .parse_result                   (parse_result           ),
81.
82.     .packet_tx_done                 (packet_tx_done         ),
83.     .uart_tx_req                    (uart_tx_en             ),
84.     .uart_tx_data                   (uart_tx_data           )
85. );
86.
87.
88. uart_tx u_uart_tx(
89.     .clk                            (sys_clk                ),
90.     .rst_n                          (sys_rst_n              ),
91.     .uart_tx_req                    (uart_tx_en             ),// 发送使能
    请求
92.     .uart_tx_data                   (uart_tx_data           ),// UART 要发
    送的数据
93.     .uart_txd                       (uart_txd               ),// UART 发送
    端口
94.     .uart_tx_busy                   (uart_tx_busy           ) // 发送忙
95. );
96.
97.
98. key_filter u_key_filter(
99.     .sys_clk                        (sys_clk                ),
100.     .sys_rst_n                      (sys_rst_n              ),
101.     .key_in                         (key_in                 ),
102.     .key_flag                       (key_flag               )
103. );
104.
105.
106. endmodule
```

顶层代码例化了串口收发模块（uart_rx/uart_tx）、按键消抖模块（key_filter）、密码机模块（locker）、串口数据包解析模块（packet_decode）、串口数据包打包模块（packet_code）。具体模块的功能在下文进行介绍。

串口数据包解析模块（packet_decode）：

```
1.  module packet_decode(
2.      input                           clk                     ,//时钟
3.      input                           rst_n                   ,//复位，低电平
    有效
4.
5.      input           [   7: 0]       uart_rx_data            ,//UART 接收数
    据完成信号
```

```
    input                           uart_rx_done            ,//UART 接收的数据
    input                           packet_tx_done          ,//一包数据全部发送完成标志

    output reg                      parse_done              ,//解包完成标志
    output reg          [   7: 0]   parse_result            ,//解包后的结果,8'h00:解析正确 8'hEx:解析错误
    output reg          [   7: 0]   packet_data
);

//parameter define
    parameter                       PACKET_HEAD             = 8'h55 ;//定义数据包头

    parameter                       ERR_HEAD                = 8'hE0 ;//包头检测错误
    parameter                       ERR_CHECKSUM            = 8'hE1 ;//校验错误
    parameter                       PARSE_RIGHT             = 8'h00 ;//数据包解析正确

    localparam                      rec_head                = 6'b00_0001;//解析包头
    localparam                      rec_data                = 6'b00_1000;//解析数据
    localparam                      rec_check_sum           = 6'b01_0000;//校验
    localparam                      rec_end                 = 6'b10_0000;//结束状态

    reg                 [   5: 0]   cur_state               ;//状态机的现态
    reg                 [   5: 0]   next_state              ;//状态机的次态

    reg                 [   7: 0]   rec_data_c              ;//接收到的数据
    reg                 [   7: 0]   rec_data_t              ;//接收的数据寄存一拍
    reg                 [   7: 0]   checksum                ;//包的累加校验和
    reg                             skip_en                 ;//控制状态跳转使能信号


//同步时序模块
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)
        cur_state <= rec_head;
    else
        cur_state <= next_state;
end

//转态转换条件
always@(*)begin
    next_state = rec_head;
```

```
48.     case(cur_state)
49.         rec_head:begin
50.             if(skip_en)
51.                 next_state = rec_data;
52.             else if(parse_result == ERR_HEAD)
53.                 next_state = rec_end;
54.             else
55.                 next_state = rec_head;
56.         end
57.         rec_data:begin
58.             if(skip_en)
59.                 next_state = rec_check_sum;
60.             else
61.                 next_state = rec_data;
62.         end
63.         rec_check_sum:begin
64.             if(skip_en)
65.                 next_state = rec_end;
66.             else if( parse_result == ERR_CHECKSUM)
67.                 next_state = rec_end;
68.             else
69.                 next_state = rec_check_sum;
70.         end
71.         rec_end :begin
72.             if(packet_tx_done)
73.                 next_state = rec_head;
74.             else
75.                 next_state = rec_end;
76.         end
77.         default:
78.             next_state = rec_head;
79.     endcase
80. end
81.
82. //时序电路描述状态输出
83. always @(posedge clk or negedge rst_n) begin
84.     if(!rst_n)begin
85.         parse_result <= 8'b0;
86.         parse_done   <= 1'b0;
87.         skip_en      <= 1'b0;
88.         rec_data_c     <= 8'b0;
89.         rec_data_t   <= 8'b0;
90.         checksum     <= 8'b0;
91.     end
92.     else begin
93.         skip_en <= 1'b0;
94.         parse_done <= 1'b0;
95.         case(cur_state)
96.            rec_head: begin                                    //检测包头
97.                 if(uart_rx_done)
98.                     if(uart_rx_data == PACKET_HEAD)begin
99.                         skip_en <= 1'b1;
100.                         checksum <= PACKET_HEAD;
101.                    end
102.                    else begin
103.                        parse_result <= ERR_HEAD;              //包头检测错
    误
104.                        parse_done    <= 1'b1;
105.                    end
106.            end
107.            rec_data: begin                                    //接收有效数
    据与累加
```

```
                if(uart_rx_done)begin
                    rec_data_c <= uart_rx_data;
                    rec_data_t <= rec_data;
                    checksum <= uart_rx_data + checksum ;
                    skip_en  <= 1'b1;
                    end
                    else
                        checksum <= checksum;
                end
            rec_check_sum:begin                                     //校验
                if(uart_rx_done)begin
                    if(checksum == uart_rx_data)begin
                        skip_en      <= 1'b1;
                        parse_done   <= 1'b1;                       //接收完成
                        parse_result <= PARSE_RIGHT;                //接收成功
                    end
                    else begin
                        parse_result <= ERR_CHECKSUM;               //校验错误
                        parse_done    <= 1'b1;
                    end
                end
            end
            rec_end:begin
                if(packet_tx_done)begin
                    skip_en       <= 1'b0 ;
                    parse_done    <= 1'b0 ;
                    parse_result  <= 8'b0 ;
                    checksum      <= 8'd0 ;
                end
            end
            default: ;
        endcase
    end
end

//接收完成后处理数据
always @(posedge clk or negedge rst_n) begin
    if(!rst_n)begin
        packet_data <= 8'b0;
    end
    else if((parse_result == PARSE_RIGHT) && (parse_done == 1'b1)) begin
            packet_data <= rec_data_c;
        end
end

endmodule
```

为了保证串口数据发送可靠，我们使用了一个简单的数据包。其格式包括包头、密码、校验三部分。其中包头固定为 16 进制的 55、密码也为 16 进制的 XX。校验为包头与密码的和取最后八位，例如如果密码是 16 进制的 FF，那么校验位则为 54。因为 16 进制的 55+FF=154，取最后八位即为 54。我们还引入了校验错误的机制，如果上位机发送的数据包头不为 55，则返回错误值 E0。如果检测到校验位错误，则返回错误值 E1。若接收成功则返回校验值 00。这些信息会在串口数据打包模块打包发送回上位机，反馈解包结果。

串口数据包打包模块(packet_code):

```
1.  module packet_code(
2.      input                           clk                     ,
3.      input                           rst_n                   ,
4.      input                           uart_tx_busy            ,//发串口发送模
    块发送忙信号
5.      input                           parse_done              ,//包解析完成标
    志
6.      input               [   7: 0]   parse_result            ,//解包后的结
    果,8'h00:解析正确 8'hEx:解析错误
7.
8.      output reg                      packet_tx_done          ,//数据全部发送
    完成标志
9.      output reg                      uart_tx_req             ,//串口发送使
    能
10.     output reg          [   7: 0]   uart_tx_data             //串口发送数
    据
11.     );
12.
13. //parameter define
14.     parameter                       PACKET_HEAD             = 8'h55 ;    /
    /定义数据包头
15.     parameter                       PARSE_RIGHT             = 8'h00 ;    /
    /数据包解析正确
16.     parameter                       PACKET_LEN              = 8'd3  ;    /
    /定义数据包长度
17.
18. //获取下降沿
19.     reg                             tx_busy_d0              ;
20.     reg                             tx_busy_d1              ;
21.
22.     reg                             tx_req                  ;//发送使能信
    号
23.     reg                 [   7: 0]   data_send_cnt           ;//发送数据计数
    器
24.     reg                 [  39: 0]   uart_packet_data        ;//定义发送包数
    据长度
25.
26.
27.     wire                            tx_busy_fall            ;//uart 发送忙
    信号的下降沿
28.
29. //获取下降沿//也作为一个字节数据发送完成标志
30.     assign                          tx_busy_fall            = tx_busy_d1 &
     (~tx_busy_d0);
31.
32. //采下降沿,寄存两拍数据
33. always@(posedge clk or negedge rst_n)begin
34.     if(!rst_n)begin
35.         tx_busy_d0 <= 1'b0;
36.         tx_busy_d1 <= 1'b0;
37.     end
38.     else begin
39.         tx_busy_d0 <= uart_tx_busy;
40.         tx_busy_d1 <= tx_busy_d0;
41.     end
42. end
43.
```

```
44. //发送数据计数器
45. always@(posedge clk or negedge rst_n)begin
46.     if(!rst_n)
47.         data_send_cnt <= 8'b0;
48.     else if(tx_busy_fall) begin
49.         if (data_send_cnt == (PACKET_LEN - 8'd1))
50.             data_send_cnt <= 8'b0;
51.         else
52.             data_send_cnt <= data_send_cnt + 8'b1;
53.     end
54. end
55. 
56. // 当数据全部发送完成，packet_tx_done 拉高
57. always@(posedge clk or negedge rst_n)begin
58.     if(!rst_n)
59.         packet_tx_done <= 1'b0;
60.     else if(data_send_cnt == (PACKET_LEN - 8'd1) && tx_busy_fall )
61.         packet_tx_done <= 1'b1;
62.     else
63.         packet_tx_done <= 1'b0;
64. end
65. 
66. //字节数据发送完成拉高发送使能信号
67. always@(posedge clk or negedge rst_n )begin
68.     if(!rst_n)
69.         tx_req <= 1'b0 ;
70.     else if(tx_busy_fall && (data_send_cnt < (PACKET_LEN - 8'd1)))
71.         tx_req <= 1'b1;
72.     else
73.         tx_req <= 1'b0;
74. end
75. 
76. //判断向上位机发送数据信号，并在串口发送模块空闲时给出串口发送使能信号
77. always@(posedge clk or negedge rst_n )begin
78.     if(!rst_n)
79.         uart_tx_req <= 1'b0 ;
80.     else if(parse_done || tx_req)
81.         uart_tx_req <= 1'b1;
82.     else
83.         uart_tx_req <= 1'b0;
84. end
85. 
86. //打包发送给上位机的数据包
87. always @(posedge clk or negedge rst_n ) begin
88.     if(!rst_n)
89.         uart_packet_data <= {32'd0,PACKET_HEAD};
90.     else if(parse_done)begin
91.             uart_packet_data[15:8] <= parse_result;
92.             uart_packet_data[23:16] <= PACKET_HEAD + parse_result;
93.         end
94. end
95. 
96. //发送数据
97. always @(posedge clk or negedge rst_n ) begin
98.     if(!rst_n)
99.         uart_tx_data <= 8'b0;
100.     else begin
101.         case(data_send_cnt)
102.             8'd0 : uart_tx_data <= uart_packet_data[7:0];
103.             8'd1 : uart_tx_data <= uart_packet_data[15:8];
104.             8'd2 : uart_tx_data <= uart_packet_data[23:16];
105.             default : uart_tx_data <= 8'b0;
```

```
        endcase
    end
end

endmodule
```

串口数据包打包模块将数据解包信息反馈给上位机，通过 uart_tx 模块发送。如果密码解析正确，则发送 16 进制的 550055。其格式与串口数据解包模块格式相同。55 代表包头，其值固定为 55。00 代表密码解析正确，而最后一个 55 是校验，用来检验本次发送是否正确。举个例子，如果包头检测错误，则返回的信息为 55E035。如果密码校验信息错误，则返回的信息为 55E136。

密码锁（locker）模块如下：

```
module locker(
    //input
    input               clk         ,   //系统时钟，50MHZ
    input               rst_n       ,   //系统复位，低电平有效
    input               key         ,   //确认按钮
    input   [7:0]       switch      ,   //八个拨码开关
    input   [7:0]       packet_data,    //密码

    //output
    output   reg [7:0]    led            //LED 灯

);

reg [7:0] switch_r;
reg [7:0] packet_data_r;
wire [7:0] led_out;

reg led_en;
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        led_en        <= 1'b0;
    end
    else if(key)begin
        if(switch_r == packet_data_r)
        led_en        <= 1'b1     ;
        else
        led_en        <= 1'b0;
    end
end

//数据寄存
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        switch_r        <= 8'd0;
        packet_data_r   <= 8'd0;
    end
    else begin
        switch_r        <= switch     ;
        packet_data_r   <= packet_data;
```

```
    end
end

//按键确认密码，如果密码正确则点亮 led 灯
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)
        led <= 8'b0;
    else if(led_en)
        led <= led_out;
    else
        led <= 8'b0;
end

breath_led#(
   .CNT_1US_MAX    (6'd49          ),
   .CNT_1MS_MAX    (10'd999        ),
   .CNT_1S_MAX     (10'd999        )
)
 u_breath_led(
    .sys_clk                        (clk                    ), // 系统时钟 50MHz
    .sys_rst_n                      (rst_n                  ), // 全局复位
    .led_out                        (led_out                ) // 输出信号，控制 LED 灯
);

endmodule
```

其功能较为简单，利用消抖后的按键信号为触发条件，将 8 位拨码开关信号与串口数据包解包模块传出的密码数据 packet_data 进行比对。若比对成功则将呼吸灯信号接入 8 个 led 灯，将 led 灯点亮。

测试代码如下：

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

//parameter define
parameter CLK_PERIOD = 20                   ;//时钟周期为 20ns
// reg define
reg        sys_clk     ;
reg        sys_rst_n   ;
reg  [7:0] uart_rx_data;
reg        uart_rx_done;

//wire define

//初始化设置
initial begin
    sys_clk      <= 1'b0 ;
    sys_rst_n    <= 1'b0 ;
    uart_rx_data <= 8'h00;
    uart_rx_done <= 1'b0 ;

    #20
    sys_rst_n <= 1'b1 ;


```

```
25.
26.     //测试
27.     #150000
28.     uart_rx_data <= 8'h55;
29.     uart_rx_done <= 1'b1 ;
30.     #20
31.     uart_rx_done  <= 1'b0;
32.
33.     #150000
34.     uart_rx_data <= 8'h56;
35.     uart_rx_done <= 1'b1 ;
36.     #20
37.     uart_rx_done <= 1'b0 ;
38.
39.     #150000
40.     uart_rx_data <= 8'hAB;
41.     uart_rx_done <= 1'b1 ;
42.     #20
43.     uart_rx_done <= 1'b0 ;
44.
45.     #150000
46. //测试
47.     #150000
48.     uart_rx_data <= 8'h55;
49.     uart_rx_done <= 1'b1 ;
50.     #20
51.     uart_rx_done  <= 1'b0;
52.
53.     #150000
54.     uart_rx_data <= 8'h57;
55.     uart_rx_done <= 1'b1 ;
56.     #20
57.     uart_rx_done <= 1'b0 ;
58.
59.     #150000
60.     uart_rx_data <= 8'hAC;
61.     uart_rx_done <= 1'b1 ;
62.     #20
63.     uart_rx_done <= 1'b0 ;
64.
65.     #150000
66. //测试
67.     #150000
68.     uart_rx_data <= 8'h50;
69.     uart_rx_done <= 1'b1 ;
70.     #20
71.     uart_rx_done  <= 1'b0;
72.
73.     #150000
74.     uart_rx_data <= 8'h57;
75.     uart_rx_done <= 1'b1 ;
76.     #20
77.     uart_rx_done <= 1'b0 ;
78.
79.     #150000
80.     uart_rx_data <= 8'hAC;
81.     uart_rx_done <= 1'b1 ;
82.     #20
83.     uart_rx_done <= 1'b0 ;
84.
85.     #150000
86.     //测试
87.     #150000
```

```
    uart_rx_data <= 8'h55;
    uart_rx_done <= 1'b1 ;
    #20
    uart_rx_done  <= 1'b0;

    #150000
    uart_rx_data <= 8'h57;
    uart_rx_done <= 1'b1 ;
    #20
    uart_rx_done <= 1'b0 ;

    #150000
     uart_rx_data <= 8'hAB;
     uart_rx_done <= 1'b1 ;
     #20
     uart_rx_done <= 1'b0 ;

end

//50Mhz 的时钟，周期则为 1/50Mhz=20ns,所以每 10ns，电平取反一次
always #(CLK_PERIOD/2) sys_clk = ~sys_clk;


    //紫光全局复位原语,仿真需要添加
    GTP_GRS GRS_INST(
        .GRS_N(1'b1)
    );

top_uart_locker u_top_uart_locker(
    //input
    .sys_clk    (sys_clk   ),       //外部 50Mhn 时钟
    .sys_rst_n  (sys_rst_n ),       //外部复位信号，低有效
    .uart_rxd   (uart_txd  ),       //UART 接收端口
    //output

    .uart_txd   ()                  //UART 发送端口
        );


uart_tx u_uart_tx(
    .clk                                (sys_clk                    ),
    .rst_n                              (sys_rst_n                  ),
    .uart_tx_req                        (uart_rx_done                 ),// 发送使能请求
    .uart_tx_data                       (uart_rx_data               ),// UART 要发送的数据
    .uart_txd                           (uart_txd                   ),// UART 发送端口
    .uart_tx_busy                       (               ) // 发送忙
);

endmodule
```

代码中通过 timescale 1ns / 1ns 设置时间单位和精度为 1 纳秒。定义了串口收发信号、时钟、复位信号作为激励信号。使用 initial 语句模拟产生密码数据。例化串口发送模块 uart_tx 模块将数据包进行发送。同时例化顶层模块将数据包进

行接收，我们主要观察串口解包模块与组包模块的功能是否正确。至于简单的模块我们直接上板验证即可。

9.4、实验现象

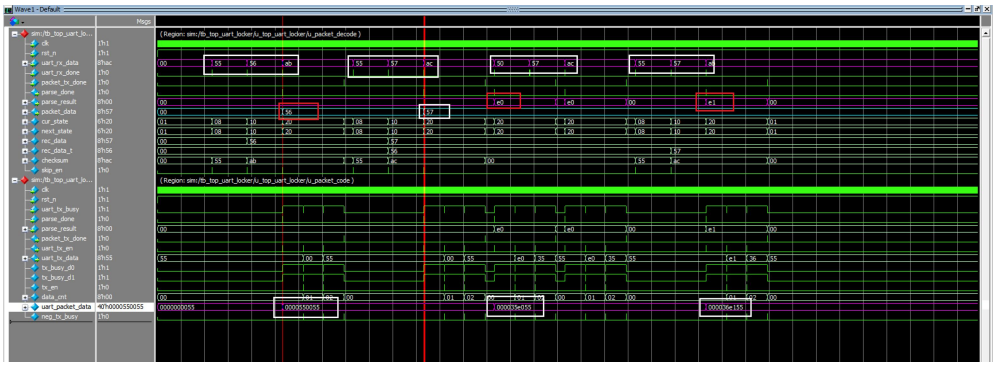


图 3-9.4-1 串口密码锁实验结果波形图

结合仿真波形分析，我们模拟发送了四组数据，分别是 5556ab（正确的数据包）、5557ac（正确的数据包）、5057ac（包头错误）、5557ab（校验错误）。可以发现发送正确数据时 packet_data 解析出正确的密码数据 56、57。但是发送错误的数据时 parse_result 输出错误信号 E0、E1。同时在组包模块 uart_packet_data 发送了四次数据包，分别是 550055、550055、35e055、36e155。发送 550055 代表密码解析正确，发送错误码 e0 代表包头检测错误，发送错误码 e1 代表校验错误。仿真图中因为串口模块从低位发送，所以这里反序输出了数据包，对功能没有影响。

我们将程序下载进入板卡，将板卡通过扩展 io 口连接上位机，打开串口调试助手，设置串口波特率为 115200，16 进制发送，16 进制接收。发送数据 55FF54。

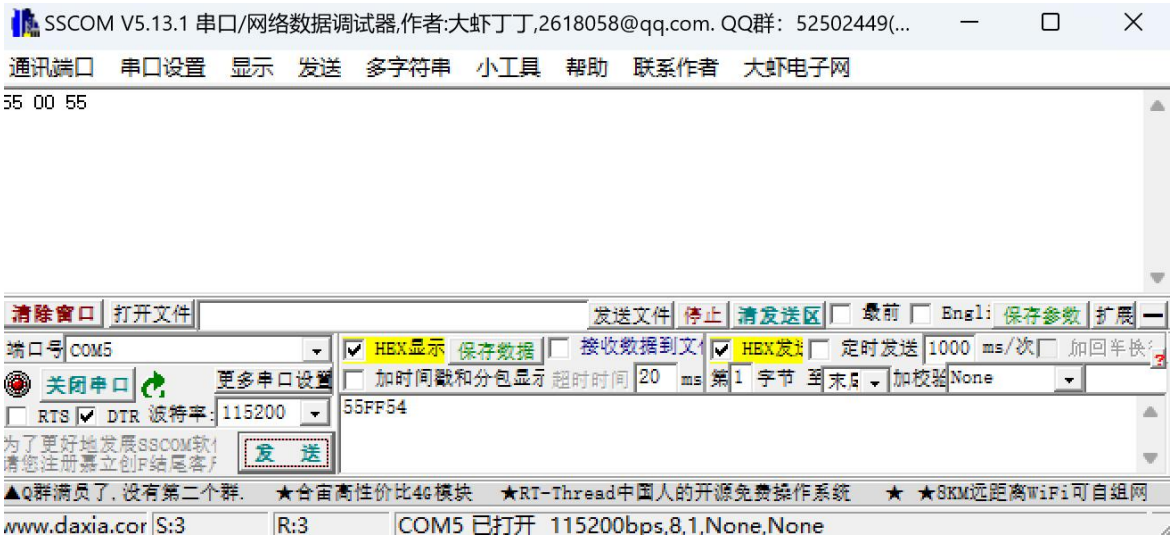


图 3-9.4-2 上位机验证实验结果图 1

可以看到板卡返回数据 550055，代表密码解析成功。我们将板卡拨码开关

全部打开（拨至 led 灯一侧）。然后按下按键 key1 进行校验，可以发现 led 被点亮。

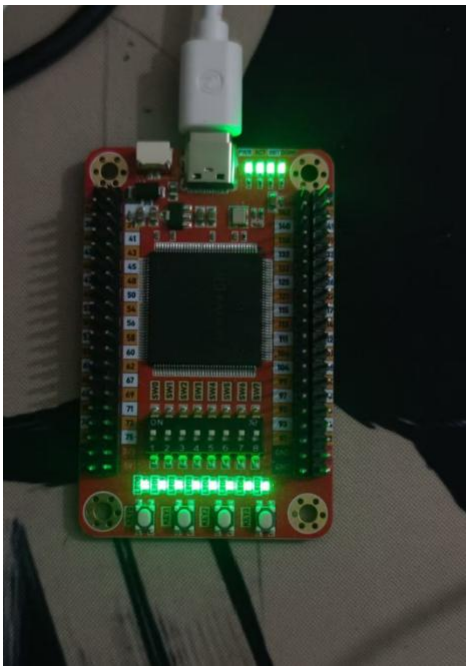


图 3-9.4-3 串口密码锁实验效果图

我们继续通过串口发送信息 50FF54（包头错误）、55FF55（校验错误）。发现板卡返回正确错误码，至此串口密码锁功能验证成功。

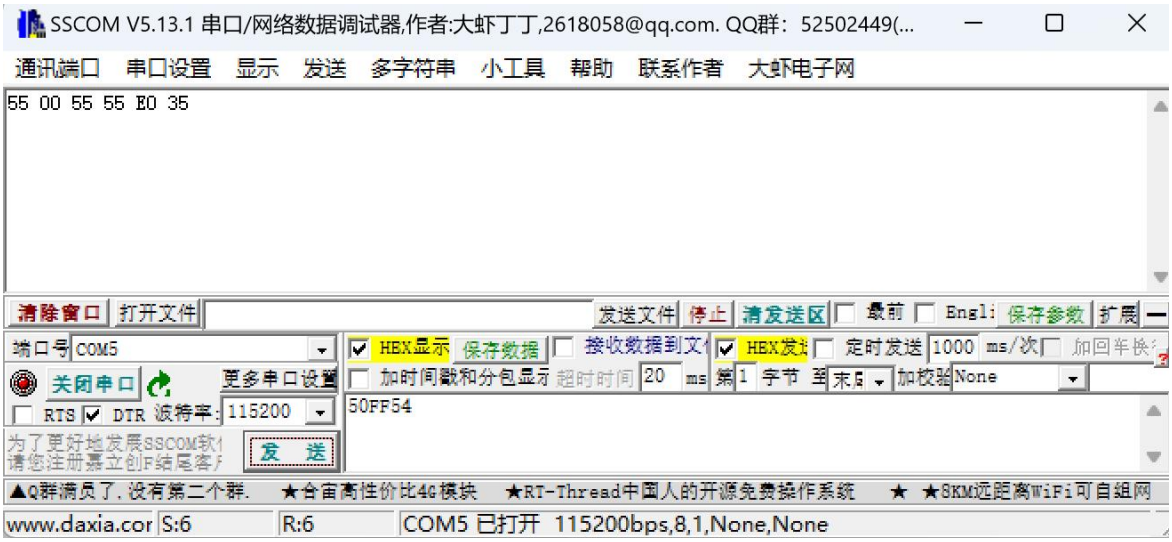


图 3-9.4-4 上位机验证实验结果图 2

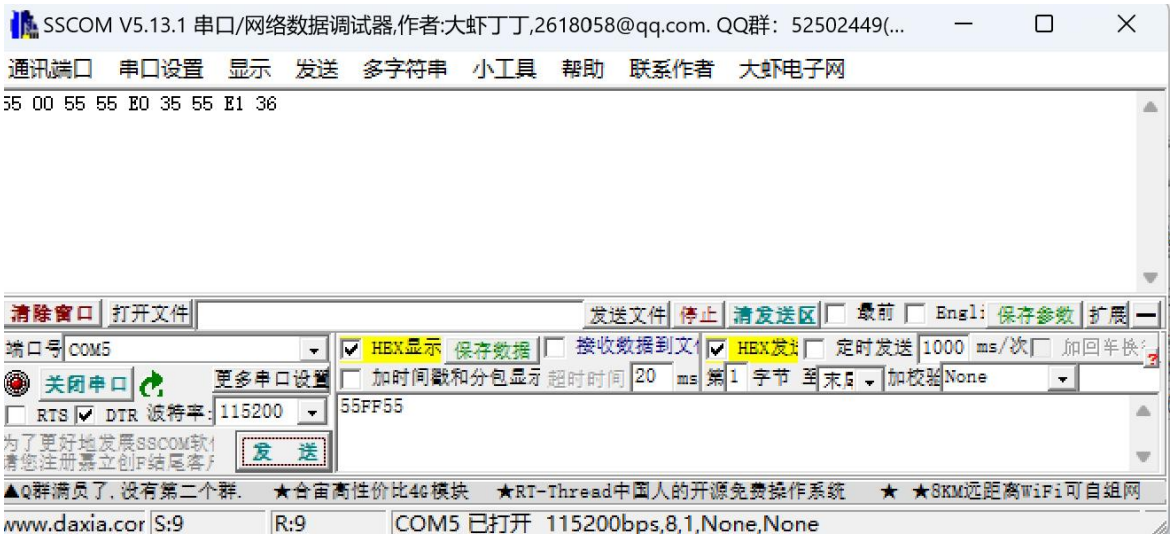


图 3-9.4-5 上位机验证实验结果图 3

10、SPI 屏幕驱动显示实验例程

10.1、实验目的

使用逻辑派 Z1 开发板（基于紫光同创 compa 系列 PGC4KD-6ILPG144 芯片）编写 spi 屏幕的驱动代码，最终在屏幕上实现红绿蓝交错的九宫格图案。

10.2、实验原理

10.2.1、SPI 协议介绍

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

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

SPI 有四种通讯模式，由时钟极性和时钟相位共同决定。时钟极性全称 ClockPolarity，通常简写成 CPOL。它是指时钟信号在空闲状态下是高电平还是低电平，当时钟空闲时为低电平即 CPOL==0，反之则 CPOL==1；时钟相位全称 ClockPhase，通常简写成 CPHA。它是指时钟信号开始有效的第一个边沿和数据的关系，当 CPHA==0 时，数据采样发生在时钟信号有效的第一个边沿（也叫奇边沿），当 CPHA==1 时，数据采样发生在时钟信号有效的第二个边沿（也叫偶边沿）；我们以 CPOL=1，CPHA=1 的模式进行分析：

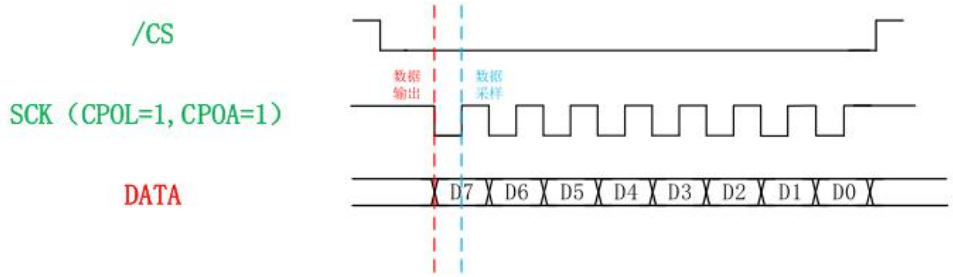


图 3-10.2-1 SPI 通讯时序图

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

10.2.2、SPI 屏幕介绍

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

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

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

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

显示要求不高的小型设备上。

本次实验选用深圳金逸晨电子有限公司生成的型号为 GMT024-8pinSPI_LCM 的 SPI 屏幕，GMT024-8pinSPI_LCM 使用 Sitronix 公司的 ST7789VM 芯片来控制 tft 液晶显示屏幕的显示，其分辨率为 240*320，其引脚说明如下：

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

图 3-10.2-2 tft 液晶显示屏幕引脚说明图

10.3、代码设计

10.3.1、总体介绍

我们将 SPI 屏幕的驱动功能分为以下模块：

test：产生测试数据，例化整个 spi 屏幕驱动模块，用于板级测试产生图像数据。

spi_screen_top：对屏幕驱动模块进行封装，使之便于用户操作。

spi_tft_screen_driver：spi 屏幕驱动模块，例化屏幕初始化模块，屏幕刷新显示模块，spi 发送模块。

spi_tft_screen_flush：对用户输入的图像数据进行刷新显示。

spi_tft_screen_init：spi 屏幕初始化模块，对 ST7789VM 进行相关配置。

spi_master_driver：接收各个模块传入的命令与数据并通过 spi 协议发送给 ST7789VM。

模块层次框图如下：

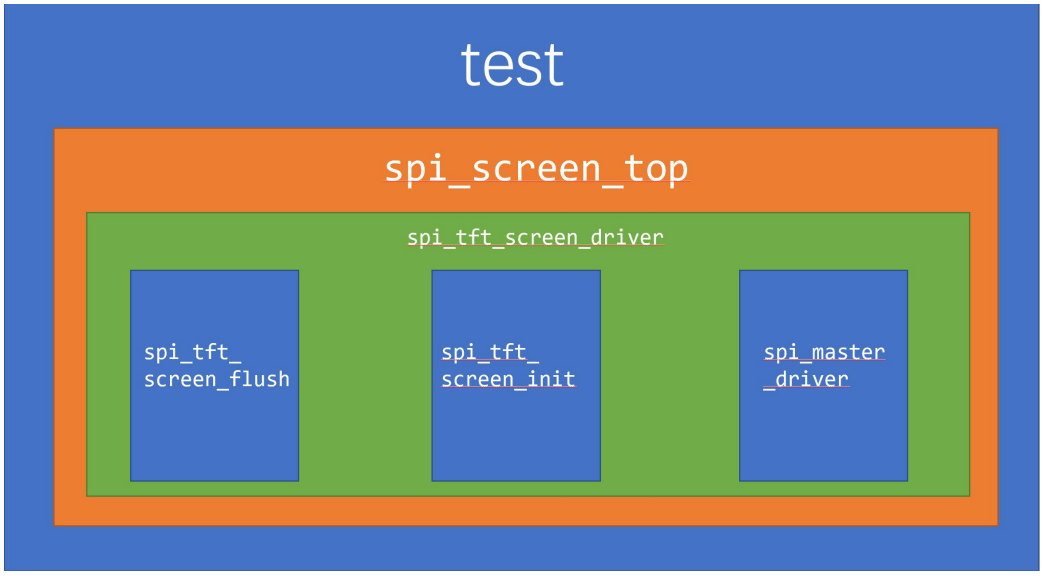


图 3-10.3-1 SPI 屏幕驱动功能模块层次框图

10.3.2、模块介绍

test 模块：

```
1.  `timescale 1ns / 1ps
2.  module test(
3.      input                           sys_clk                 ,// 系统时钟
4.      input                           sys_rst_n               ,// 复位
5.
6.      //spi tft screen   屏幕接口
7.      output                          lcd_spi_sclk            ,// 屏幕 spi 时
    钟接口
8.      output                          lcd_spi_mosi            ,// 屏幕 spi 数
    据接口
9.      output                          lcd_spi_cs              ,// 屏幕 spi 使
    能接口
10.     output                          lcd_dc                  ,// 屏幕 数据/
    命令 接口
11.     output                          lcd_reset               ,// 屏幕复位接
    口
12.     output                          lcd_blk                  // 屏幕背光接
    口
13.
14. );
15.
16.     wire                            flush_data_update       ;//更新当前坐标
    点显示数据使能
17.     reg             [  15: 0]       flush_data              ;//当前坐标点显
    示的数据
18.     wire            [  15: 0]       flush_addr_width        ;//当前刷新的 x
    坐标
19.     wire            [  15: 0]       flush_addr_height       ;//当前刷新的 y
    坐标
20.
21.
```

```
22. always @(posedge sys_clk or negedge sys_rst_n) begin                    // 显示九宫格
    测试图片 240*320
23.     if (sys_rst_n == 1'b0)
24.         flush_data <= 16'd0;
25.     else if ((flush_addr_width >= 'd0   && flush_addr_width < 'd106) && (flush_a
    ddr_height >= 'd0 && flush_addr_height < 'd80))
26.         flush_data <= 16'hF800;//红  1111 1000 0000 0000
27.     else if ((flush_addr_width >= 'd106 && flush_addr_width < 'd212) && (flush_a
    ddr_height >= 'd0 && flush_addr_height < 'd80))
28.         flush_data <= 16'h07E0;//绿  0000 0111 1110 0000
29.     else if ((flush_addr_width >= 'd212 && flush_addr_width < 'd320) && (flush_a
    ddr_height >= 'd0 && flush_addr_height < 'd80))
30.         flush_data <= 16'h001F;//蓝  0000 0000 0001 1111
31.
32.     else if ((flush_addr_width >= 'd0   && flush_addr_width < 'd106) && (flush_a
    ddr_height >= 'd80 && flush_addr_height < 'd160))
33.         flush_data <= 16'h07E0;//绿
34.     else if ((flush_addr_width >= 'd106 && flush_addr_width < 'd212) && (flush_a
    ddr_height >= 'd80 && flush_addr_height < 'd160))
35.         flush_data <= 16'h001F;//蓝
36.     else if ((flush_addr_width >= 'd212 && flush_addr_width < 'd320) && (flush_a
    ddr_height >= 'd80 && flush_addr_height < 'd160))
37.         flush_data <= 16'hF800;//红
38.
39.     else if ((flush_addr_width >= 'd0   && flush_addr_width < 'd106) && (flush_a
    ddr_height >= 'd160 && flush_addr_height < 'd240))
40.         flush_data <= 16'h001F;//蓝
41.     else if ((flush_addr_width >= 'd106 && flush_addr_width < 'd212) && (flush_a
    ddr_height >= 'd160 && flush_addr_height < 'd240))
42.         flush_data <= 16'hF800;//红
43.     else if ((flush_addr_width >= 'd212 && flush_addr_width < 'd320) && (flush_a
    ddr_height >= 'd160 && flush_addr_height < 'd240))
44.         flush_data <= 16'h07E0;//绿
45.     else
46.         flush_data <= 16'h0000;
47. end
48. //整个 spi 屏幕控制顶层，用户只需要提供显示数据就可以使用这个顶层进行 spi 屏幕显示
49. spi_screen_top spi_screen_top_inst(
50.     .sys_clk                                (sys_clk                    ),
51.     .sys_rst_n                              (sys_rst_n                  ),
52.
53.
54.     //用户信号
55.     .flush_data_update_o                    (flush_data_update          ),//更新当前坐
    标点显示数据使能
56.     .flush_data_i                           (flush_data                 ),//当前坐标点
    显示的数据
57.     .flush_addr_width_o                     (flush_addr_width           ),//当前刷新的
    x 坐标
58.     .flush_addr_height_o                    (flush_addr_height          ),//当前刷新的
    y 坐标
59.
60.      //spi tft screen   屏幕接口
61.     .lcd_spi_sclk                           (lcd_spi_sclk               ),// 屏幕spi时
    钟接口
62.     .lcd_spi_mosi                           (lcd_spi_mosi               ),// 屏幕spi数
    据接口
63.     .lcd_spi_cs                             (lcd_spi_cs                 ),// 屏幕spi使
    能接口
```

```
64.     .lcd_dc                         (lcd_dc                 ),// 屏幕 数据
    /命令 接口
65.     .lcd_reset                      (lcd_reset              ),// 屏幕复位
    接口
66.     .lcd_blk                        (lcd_blk                ) // 屏幕背光
    接口
67. );
68.
69. endmodule
```

这是整个实验工程的顶层模块，用于产生测试数据进行板级验证，模块中例化了 spi_screen_top 模块，根据 spi_screen_top 输出的 flush_addr_width 与 flush_addr_height 信号分配不同的显示数据，产生红绿蓝交错的九宫格测试图案。以下是对代码具体的解释说明。

代码第 8-14 行定义了与 TFT 屏幕通信的 SPI 接口信号，包括时钟、数据、片选、数据/命令控制、复位和背光控制这些信号都由 spi_screen_top 模块传出。

代码第 17 行定义了信号 flush_data_update，用于标志当前坐标点显示数据更新状态，若为高电平则代表一个 16bit 的 rgb565 图像数据通过 spi 协议成功发送给屏幕。

代码第 18 行定义了寄存器 flush_data，用于存储当前坐标点的显示数据，同时将数据传入 spi_screen_top 模块。

代码第 19-20 行定义了 flush_addr_width 和 flush_addr_height，表示当前刷新的 X 和 Y 坐标。

代码第 22-38 行根据 flush_addr_width 和 flush_addr_height 的坐标值，将 flush_data 赋值为不同的 16 位颜色数据，形成九宫格显示效果。

代码第 40-52 行实例化了 spi_screen_top 模块，负责 SPI 屏幕通信，并根据 spi_screen_top 模块输出的坐标将图像数据 flush_data 传入用于驱动 TFT 屏幕显示。

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

spi_screen_top 模块：

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

```
47.
48.     //当发送完 16bit 的图像数据时，坐标点显示数据使能拉高
49.     assign  flush_data_update_o = (spi_screen_flush_updte == 1'b1 && data_update
    _cnt == 1'b0 && flush_updte_en == 1'b1) ? 1'b1 : 1'b0;
50.
51.     //将长宽计数器连接到输出
52.     assign  flush_addr_width_o        = width_cnt;
53.     assign  flush_addr_height_o       = height_cnt;
54. always@(posedge sys_clk or negedge sys_rst_n) begin
55.     if( sys_rst_n == 1'b0)
56.         flush_updte_en <= 'd0;
57.     else if( spi_screen_flush_fsync == 1'b1 )    //刷新模块发送完一帧数据,清
    零
58.         flush_updte_en <= 'd0;
59.     else if( spi_screen_flush_updte == 1'b1)     //发送完 8 位数据后,flush_updte_en
    拉高
60.         flush_updte_en <= 'd1;
61.     else
62.         flush_updte_en <= flush_updte_en;
63. end
64. always@(posedge sys_clk or negedge sys_rst_n) begin
65.     if( sys_rst_n == 1'b0)
66.         data_update_cnt <= 'd0;
67.     else if( spi_screen_flush_fsync == 1'b1 )                          //刷新模块发送
    完一帧数据
68.         data_update_cnt <= 'd0;
69.     else if( spi_screen_flush_updte == 1'b1)                           //刷新模块通过
    spi 发送完一个 8bit 数据，data_update_cnt 加一
70.         data_update_cnt <= data_update_cnt + 1'b1;                     //发送高八位时
    data_update_cnt=1，发送低八位时
    data_update_cnt=0，
71.     else                                                               //所以发送完
    16bit 数据时 data_update_cnt=0
72.         data_update_cnt <= data_update_cnt;
73. end
74.
75.
76. always@(posedge sys_clk or negedge sys_rst_n) begin
77.     if( sys_rst_n == 1'b0 )
78.         width_cnt <= 'd0;
79.     else if( spi_screen_flush_fsync == 1'b1 )                          //一帧图像数据
    发送完，计数器清零
80.         width_cnt <= 'd0;
81.     else if( flush_data_update_o)//发送完当前像素点 16bit 图像数据时
82.         if( width_cnt == (SCREEN_WIDTH-1))                        //计数到计数器最大
    值时清零
83.             width_cnt <= 'd0;
84.         else
85.             width_cnt <= width_cnt + 1'b1;                         //width_cnt
    计数器加 1
86.     else
87.         width_cnt <= width_cnt;
88. end
89.
90.
91. always@(posedge sys_clk or negedge sys_rst_n) begin
92.     if( sys_rst_n == 1'b0 )
93.         height_cnt <= 'd0;
94.     else if( spi_screen_flush_fsync == 1'b1)                           //一帧图像数据
    发送完，计数器清零
```

```
95.         height_cnt <= 'd0;
96.     else if( width_cnt == (SCREEN_WIDTH-1) && flush_data_update_o)//当图像绘制完一
    行时
97.         if( height_cnt == (SCREEN_HEIGHT-1))                    //计数到计数器最
    大值时清零
98.             height_cnt <= 'd0;
99.         else
100.             height_cnt <= height_cnt + 1'b1;                    //height_cn
    t 计数器加一
101.     else
102.         height_cnt <= height_cnt;
103. end
104.
105.
106.
107. always@(posedge sys_clk or negedge sys_rst_n) begin
108.     if( sys_rst_n == 1'b0)
109.         flush_data_reg <= 'd0;
110.     else if( spi_screen_flush_updte == 1'b1)                //刷新模块发送完一个图
    像数据
111.         if( data_update_cnt == 1'b0 )                       //data_update_cnt=0
    时,发送完了低八位,寄存新数据
112.             flush_data_reg <= flush_data_i;
113.         else
114.             flush_data_reg <= flush_data_reg << 8;         //data_update_cnt=1 时,
    发送完了高八位,将数据向左移动 8 位，发送低八位数据
115.     else
116.         flush_data_reg <= flush_data_reg;
117. end
118.
119.
120.
121.
122. spi_tft_screen_driver spi_tft_screen_driver_inst(
123.     .sys_clk                            (sys_clk                    ),
124.     .sys_rst_n                          (sys_rst_n                  ),
125.
126.
127.     //用户接口
128.     .spi_screen_flush_data_i            (spi_screen_flush_data      ),//屏幕显示
    数据
129.     .spi_screen_flush_updte_o           (spi_screen_flush_updte     ),//像素点数
    据刷新//在进行数据到 tft 屏幕的显示
130.     .spi_screen_flush_fsync_o           (spi_screen_flush_fsync     ),//屏幕帧同
    步
131.
132.      //spi tft screen   屏幕接口
133.     .lcd_spi_sclk                       (lcd_spi_sclk               ),// 屏幕 spi
    时钟接口
134.     .lcd_spi_mosi                       (lcd_spi_mosi               ),// 屏幕 spi
    数据接口
135.     .lcd_spi_cs                         (lcd_spi_cs                 ),// 屏幕 spi
    使能接口
136.     .lcd_dc                             (lcd_dc                     ),// 屏幕 数
    据/命令 接口
137.     .lcd_reset                          (lcd_reset                  ),// 屏幕复位
    接口
138.     .lcd_blk                            (lcd_blk                    ) // 屏幕背光
    接口
139. );
```

```
140. endmodule
```

spi_screen_top 模块是对 spi_tft_screen_driver 模块的封装，简化了 spi 驱动模块（spi_tft_screen_driver 模块）的操作方式，使得用户端只需要使用类似的 vga 接口的方式给在行列坐标刷新的过程中给出数据即可。以下是对 spi_screen_top 模块的具体介绍：

代码第 3 行定义了模块 spi_screen_top，这是 SPI 屏幕控制的顶层模块，用户只需提供显示数据即可使用。

代码第 10-13 行定义了用户接口信号，包括数据刷新使能、当前显示数据、刷新 X 坐标和 Y 坐标输出。

代码第 16-21 行定义了 SPI 屏幕的接口信号，用于与 TFT 屏幕通信。

代码第 23-24 行定义了屏幕宽度和高度的参数，分别为 320 和 240。

代码第 26-30 行定义了内部信号，包括用于刷新屏幕的像素数据 spi_screen_flush_data 、数据更新 spi_screen_flush_updte 和帧同步信号 spi_screen_flush_fsync 。spi_screen_flush_updte 信号由屏幕显示刷新模块（spi_tft_screen_flush）传出，其每拉高一个高电平代表图像数据发送完成 8bit。spi_screen_flush_fsync 信号也由屏幕显示刷新模块（spi_tft_screen_flush）传出，其每拉高一个高电平代表发送完成一帧数据。

代码第 39 行定义了 data_update_cnt 信号，由于用户给出的数据是 16bit 的 rgb565 数据，而 spi 发送数据只能是 8bit，所以使用这个信号对用户的 16bit 数据进行高位与低位的区分，当 data_update_cnt 为 0 时发送用户高 8 位数据，当 data_update_cnt 为 1 时发送用户低 8 位数据。

代码第 47-57 行驱动 width_cnt，当帧同步信号（spi_screen_flush_fsync）有效时 width_cnt 置零，当数据更新信号 spi_screen_flush_updte 为高电平并且 data_update_cnt 也为高电平时（已经发送完成用户 16bit 图像数据的高位和低位）时，width_cnt 根据屏幕显示坐标自增和复位（自增到 320 然后清零）。

代码第 59-69 行驱动 height_cnt，更新逻辑与 width_cnt 信号相同，不同的是 height_cnt 自增到 240 然后清零。

代码第 71-79 行通过时钟驱动 flush_data_reg，根据数据更新信号

data_update_cnt 将输入的显示数据逐字节移位（0 时发送高 8 位，1 时发送低 8 位），并输出给屏幕。

代码第 81-96 行实例化了子模块 spi_tft_screen_driver，用于将显示数据通过 SPI 接口传输到 TFT 屏幕，完成显示刷新。

通过这样的机制用户只需要在行列坐标刷新时给模块提供图像数据即可完成图片的显示，操作十分方便。

spi_tft_screen_driver 模块：

```
1.  `timescale 1ns / 1ps
2.  module spi_tft_screen_driver(
3.      input                               sys_clk                 ,
4.      input                               sys_rst_n               ,
5.
6.
7.      //用户接口
8.      input               [   7: 0]       spi_screen_flush_data_i    ,//屏幕显示数
    据
9.      output                              spi_screen_flush_updte_o   ,//像素点数据刷
    新
10.     output                              spi_screen_flush_fsync_o   ,//屏幕帧同
    步
11.     //------
12.
13.      //spi tft screen   屏幕接口
14.     output                              lcd_spi_sclk               ,// 屏幕 spi 时
    钟接口
15.     output                              lcd_spi_mosi               ,// 屏幕 spi 数
    据接口
16.     output                              lcd_spi_cs                 ,// 屏幕 spi 使
    能接口
17.     output                              lcd_dc                     ,// 屏幕  数据/
    命令 接口
18.     output                              lcd_reset                  ,// 屏幕复位接
    口
19.     output                              lcd_blk                     // 屏幕背光接
    口
20. );
21.
22. //屏幕尺寸
23.     parameter                           SCREEN_WIDTH            = 32'd320;
24.     parameter                           SCREEN_HEIGHT           = 32'd240;
25.
26. //总模块信号
27.     reg                                 lcd_init_done           ;//初始化完成标
    志信号
28.     wire                                spi_start               ;//spi 开始信
    号
29.     wire                                spi_end                 ;//spi 结束信
    号
30.     wire                [   7: 0]       spi_send_data           ;//spi 发送数
    据
```

```
31.     wire                                spi_send_ack                ;//spi 数据发送
    完成响应
32.     wire                                lcd_dc_i                    ;//lcd 数据/命
    令信号
33.
34. // 初始化模块信号
35.     wire                                tft_screen_init_req         ;//初始化模块请
    求
36.     wire                                tft_screen_init_ack         ;//初始化模块完
    成
37.     wire                [   7: 0]       tft_screen_init_data        ;//初始化模块数
    据
38.     wire                                tft_screen_init_dc          ;//初始化模块
    dc
39.     wire                                spi_send_init_req           ;//spi 发送数据
    请求
40.     wire                                spi_send_init_end           ;//结束 spi 发
    送
41.     wire                                spi_send_init_ack           ;//spi 数据发送
    完成响应
42.
43. //刷新模块
44.
45.     wire                                tft_screen_flush_req        ;//刷新模块请
    求
46.     wire                [   7: 0]       tft_screen_flush_data       ;//刷新模块数
    据
47.     wire                                tft_screen_flush_dc         ;//刷新模块数据
    /命令信号
48.
49.     wire                                spi_send_flush_req          ;//刷新模块发送
    请求
50.     wire                                spi_send_flush_end          ;//刷新模块发送
    结束
51.     wire                                spi_send_flush_ack          ;//刷新模块数据
    发送完成响应
52.
53.
54. //屏幕驱动信号，默认
55.     assign                              lcd_reset                   = 1'b1;
56.     assign                              lcd_blk                     = 1'b1;
57.
58.
59.     assign                              tft_screen_flush_req        = ( lcd_init_d
    one == 1'b1) ? 1'b1 : 1'b0;
60.     assign                              spi_send_flush_ack          = ( lcd_init_d
    one == 1'b1) ? spi_send_ack : 1'b0;
61.
62.
63.     assign                              tft_screen_init_req         =  ~lcd_init_d
    one;
64.     assign                              spi_send_init_ack           = ( lcd_init_d
    one == 1'b0) ? spi_send_ack : 1'b0;
65.
66.     //像素初始化完成之后进入显示数据刷新模式
67.     assign                              spi_start                   = ( lcd_init_d
    one == 1'b0) ? spi_send_init_req : spi_send_flush_req;
68.     assign                              spi_end                     = ( lcd_init_d
    one == 1'b0) ? spi_send_init_end : spi_send_flush_end;
```

```
69.     assign                                  spi_send_data             = ( lcd_init_d
    one == 1'b0) ? tft_screen_init_data : tft_screen_flush_data;
70.     assign                                  lcd_dc_i                  = ( lcd_init_d
    one == 1'b0) ? tft_screen_init_dc : tft_screen_flush_dc;
71.
72.
73. //初始化是否完成
74. always@(posedge sys_clk or negedge sys_rst_n) begin
75.     if( sys_rst_n == 1'b0)
76.         lcd_init_done <= 1'b0;
77.     else if( tft_screen_init_ack == 1'b1)   //初始化模块完成 ,lcd_init_done 拉高
78.         lcd_init_done <= 1'b1;
79.     else
80.         lcd_init_done <= lcd_init_done;
81. end
82.
83.
84.
85. //刷新模块
86. spi_tft_screen_flush #(
87.     .SCREEN_WIDTH                       (SCREEN_WIDTH               ),
88.     .SCREEN_HEIGHT                      (SCREEN_HEIGHT              ),
89.     .Number_Of_Pixels                   (SCREEN_WIDTH*SCREEN_HEIGHT*'d2)
90. )spi_tft_screen_flush_inst(
91.     .sys_clk                            (sys_clk                    ),
92.     .sys_rst_n                          (sys_rst_n                  ),
93.
94.      //用户接口
95.     .spi_screen_flush_data_i            (spi_screen_flush_data_i    ),//屏幕显示数
    据
96.     .spi_screen_flush_updte_o           (spi_screen_flush_updte_o   ),//像素点数据
    刷新
97.     .spi_screen_flush_fsync_o           (spi_screen_flush_fsync_o   ),//屏幕帧同
    步
98.
99.
100.     .tft_screen_flush_req_i             (tft_screen_flush_req       ),//刷新请
    求
101.     .tft_screen_flush_data_o            (tft_screen_flush_data      ),//刷新数
    据
102.     .tft_screen_flush_dc_o              (tft_screen_flush_dc        ),//刷新 dc
103.
104.
105.     .spi_send_flush_req_o               (spi_send_flush_req         ),//spi 发送数
    据请求
106.     .spi_send_flush_end_o               (spi_send_flush_end         ),//结束 spi
    发送
107.     .spi_send_flush_ack_i               (spi_send_flush_ack         ) //spi 一个数
    据发送完成
108. );
109.
110.
111.
112. //初始化模块
113. spi_tft_screen_init  #(
114.     .SCREEN_WIDTH                       (SCREEN_WIDTH               ),
115.     .SCREEN_HEIGHT                      (SCREEN_HEIGHT              )
116. ) spi_tft_screen_init_inst(
117.     .sys_clk                            (sys_clk                    ),
118.     .sys_rst_n                          (sys_rst_n                  ),
119.
```

```
120.
121.     .tft_screen_init_req_i              (tft_screen_init_req       ),//初始化请
    求
122.     .tft_screen_init_ack_o              (tft_screen_init_ack       ),//初始化完
    成
123.     .tft_screen_init_data_o             (tft_screen_init_data      ),//初始化数
    据
124.     .tft_screen_init_dc_o               (tft_screen_init_dc        ),//初始化
    dc
125.
126.
127.     .spi_send_init_req_o                (spi_send_init_req         ),//spi 发送数
    据请求
128.     .spi_send_init_end_o                (spi_send_init_end         ),//结束 spi
    发送
129.     .spi_send_init_ack_i                (spi_send_init_ack         ) //spi 一个数
    据发送完成
130. );
131.
132.
133.
134. //spi 主机驱动模块
135. spi_master_driver spi_master_driver_inst(
136.     //系统接口
137.     .sys_clk                            (sys_clk                   ),
138.     .sys_rst_n                          (sys_rst_n                 ),
139.
140.     //用户接口
141.     .spi_start_i                        (spi_start                 ),// spi 开始
    信号
142.     .spi_end_i                          (spi_end                   ),// spi 结束
    信号
143.     .spi_send_data_i                    (spi_send_data             ),// spi 发送
    数据
144.     .spi_send_ack_o                     (spi_send_ack              ),// spi 发送
    8bit 数据完成信号
145.
146.     .lcd_dc_i                           (lcd_dc_i                  ),//数据还是
    命令信号输入
147.     .lcd_dc                             (lcd_dc                    ),//数据还是
    命令信号输出
148.
149.     //spi   端口
150.     .spi_sclk                           (lcd_spi_sclk              ),
151.     .spi_mosi                           (lcd_spi_mosi              ),
152.     .spi_cs                             (lcd_spi_cs                )
153. );
154.
155. endmodule
```

此模块是屏幕数据刷新模块（spi_tft_screen_flush）、屏幕初始化模块（spi_tft_screen_init）、spi 发送模块（spi_master_driver）的顶层代码。其主要功能是连接三个模块，配置并刷新屏幕，同时提供给用户接口信号，发出 spi 屏幕控制信号。代码简单，我们只对其做简单的介绍。

代码第 10-13 行定义了用户接口信号，包括显示数据、像素点数据刷新信号和屏幕帧同步信号。

代码第 16-21 行定义了 SPI 屏幕的接口信号，用于与 TFT 屏幕通信。

代码第 23-25 行定义了屏幕尺寸的参数，分别为宽度 320 和高度 240。

代码第 28-29 行默认将 lcd_reset 和 lcd_blk 信号设置为高电平，控制屏幕复位和背光。

代码第 32-36 行定义了驱动屏幕显示的相关信号，包括 SPI 开始/结束信号、发送的数据和 DC（数据/命令）信号（高电平数据，低电平命令）。

代码第 61-66 行通过 lcd_init_done 信号控制初始化完成状态，决定系统进入刷新模式。

代码第 68-80 行实例化了 spi_tft_screen_flush 模块，负责屏幕显示数据的刷新控制。

代码第 83-93 行实例化了 spi_tft_screen_init 模块，负责给出初始化屏幕命令与数据。

代码第 95-104 行实例化了 spi_master_driver 模块，负责通过 SPI 接口将数据发送到 TFT 屏幕，并输出 SPI 时钟、数据和片选信号。

spi_tft_screen_flush 模块

```
1.  `timescale 1ns / 1ps
2.
3.  //对 spi tft 屏幕进行刷新
4.  module spi_tft_screen_flush(
5.      input                               sys_clk                 ,
6.      input                               sys_rst_n               ,
7.
8.
9.      //用户接口
10.     input               [   7: 0]       spi_screen_flush_data_i    ,//屏幕显示数
    据
11.     output                              spi_screen_flush_updte_o   ,//像素点数据刷
    新
12.     output                              spi_screen_flush_fsync_o   ,//屏幕帧同
    步
13.
14.
15.     //驱动模块
16.     input                               tft_screen_flush_req_i     ,//刷新请求//
    初始化完成之后此信号拉高模块进入刷新模式
17.     output reg          [   7: 0]       tft_screen_flush_data_o    ,//刷新数据//
    刷新的图像数据和刷新命令通过 spi 模块发送
```

```
18.     output reg                          tft_screen_flush_dc_o       ,//刷新 dc//区
    分命令与数据
19.
20.     //SPI 主模块
21.     output                              spi_send_flush_req_o        ,//spi 发送数据
    请求
22.     output                              spi_send_flush_end_o        ,//结束 spi 发
    送
23.     input                               spi_send_flush_ack_i         //spi 一个数据
    发送完成
24. );
25.     parameter                           SCREEN_WIDTH                = 16'd320;
26.     parameter                           SCREEN_HEIGHT               = 16'd240;
27.     parameter                           Number_Of_Pixels            = 32'd240*32'd
    320*32'd2; // 像素点个数
28.
29.
30.     localparam                          S_IDLE                      = 4'b0001;
31.     localparam                          S_DATA                      = 4'b0010;  //
    发送数据
32.     localparam                          S_DELAY                     = 4'b0100;  //
    延时
33.     localparam                          S_FRAME_SYNC                = 4'b1000;  //
     帧同步
34.
35.
36.     localparam                          DELAY_5clk                  = 'd5   ; //命
    令与数据之间切换等待 5 个时钟周期
37.
38.     reg                 [  31: 0]       flush_cnt                    ;        //刷
    新模块写命令/数据计数器
39.     reg                 [  12: 0]       delay_cnt                    ;        //延
    迟计数
40.     reg                 [   3: 0]       state           ,next_state;         //状
    态寄存器
41.
42.     //为写数据状态时，请求拉高
43.     assign                              spi_send_flush_req_o        = (state == S_
    DATA) ? 1'b1 : 1'b0;
44.     //为延迟状态或帧同步状态时结束信号拉高
45.     assign                              spi_send_flush_end_o        = (state == S_
    DELAY || state == S_FRAME_SYNC) ? 1'b1 : 1'b0;
46.
47.     //当发送完数据时，且写命令/数据计数器 计数到 10 时，像素点数据刷新拉高
48.     assign                              spi_screen_flush_updte_o    = ( spi_send_f
    lush_ack_i == 1'b1 && flush_cnt >= 'd10 ) ? 1'b1 : 1'b0;
49.     //为帧同步状态时，屏幕帧同步拉高，产生帧同步信号
50.     assign                              spi_screen_flush_fsync_o    = ( state == S
    _FRAME_SYNC ) ? 1'b1 : 1'b0;
51.
52. always@(posedge sys_clk or negedge sys_rst_n) begin
53.     if( sys_rst_n == 1'b0 )
54.         state <= S_IDLE;
55.     else
56.         state <= next_state;            //将状态赋值为下一状态
57. end
58.
59.
60. always@(*) begin
61.     case(state)
```

```
62.     S_IDLE:
63.         if( tft_screen_flush_req_i == 1'b1 )                    //初始化完成之后此信号拉高模块进入刷新模式
64.             next_state = S_DATA;
65.         else
66.             next_state = S_IDLE;
67.     S_DATA:
68.         if( spi_send_flush_ack_i == 1'b1 && flush_cnt <= 'd10 )     //地址设置需要发送命令和写入四个参数，设置 XY 地址共需要发送 10 个数据
69.             next_state = S_DELAY;                                   //加上发送内存写入命令共需要 11 个数据，所以 flush_cnt 计数到 10
70.         else if( spi_send_flush_ack_i == 1'b1 && flush_cnt == (Number_Of_Pixels + 'd10))//一帧图像数据发送完成，跳转到帧同步状态
71.             next_state = S_FRAME_SYNC;
72.         else
73.             next_state = S_DATA;
74.     S_DELAY:
75.         if( delay_cnt == DELAY_5clk)  //延迟 5 给时钟周期后，跳转到 S_DATA 状态
76.             next_state = S_DATA;
77.         else
78.             next_state = S_DELAY;
79.     S_FRAME_SYNC:
80.         next_state = S_IDLE;
81.     default: next_state = S_IDLE;
82.     endcase
83. end
84.
85.
86. //发送数据计数
87. always@(posedge sys_clk or negedge sys_rst_n) begin
88.     if( sys_rst_n == 1'b0 )
89.         flush_cnt <= 'd0;
90.     else if( spi_send_flush_ack_i == 1'b1 &&  flush_cnt == (Number_Of_Pixels + 'd10))//一帧图像数据发送完成，flush_cnt 清零
91.         flush_cnt <= 'd0;
92.     else if( spi_send_flush_ack_i == 1'b1 )//发送完一个数据，flush_cnt 加 1
93.         flush_cnt <= flush_cnt + 1'b1;
94.     else
95.         flush_cnt <= flush_cnt;
96. end
97.
98.
99. //延时计数
100. always@(posedge sys_clk or negedge sys_rst_n) begin
101.     if( sys_rst_n == 1'b0)
102.         delay_cnt <= 'd0;
103.     else if( state == S_DELAY)//为延迟状态时，delay_cnt 加 1
104.         delay_cnt <= delay_cnt + 1'b1;
105.     else
106.         delay_cnt <= 'd0;
107. end
108.
109. //tft_screen_flush_dc_o=0 时写命令
110. //tft_screen_flush_dc_o=1 时写数据
111. always @(*) begin
112.     case(flush_cnt)
113.     'd0: begin
114.         tft_screen_flush_data_o = 8'h2A;                        //设置列地址
115.         tft_screen_flush_dc_o   = 1'b0;
116.     end
117.     //写 X
```

```
118.     'd1: begin
119.         tft_screen_flush_data_o = 8'h00;                        //列地址开始的高
    8 位
120.         tft_screen_flush_dc_o   = 1'b1;
121.     end
122.     'd2: begin
123.         tft_screen_flush_data_o = 8'h00;                        //列地址开始的低
    8 位
124.         tft_screen_flush_dc_o   = 1'b1;
125.     end
126.     'd3: begin
127.         tft_screen_flush_data_o = SCREEN_WIDTH[15:8];           //列地址结束的高
    8 位
128.         tft_screen_flush_dc_o   = 1'b1;
129.     end
130.     'd4: begin
131.         tft_screen_flush_data_o = SCREEN_WIDTH[7:0] - 1'b1;     //列地址结束的低
    8 位
132.         tft_screen_flush_dc_o   = 1'b1;
133.     end
134.     //写 Y
135.     'd5: begin
136.         tft_screen_flush_data_o = 8'h2B;                        //设置行地址
137.         tft_screen_flush_dc_o   = 1'b0;
138.     end
139.     'd6: begin
140.         tft_screen_flush_data_o = 8'h00;                        //行地址开始的高
    8 位
141.         tft_screen_flush_dc_o   = 1'b1;
142.     end
143.     'd7: begin
144.         tft_screen_flush_data_o = 8'h00;                        //行地址开始的低
    8 位
145.         tft_screen_flush_dc_o   = 1'b1;
146.     end
147.     'd8: begin
148.         tft_screen_flush_data_o = SCREEN_HEIGHT[15:8];          //行地址结束的高
    8 位
149.         tft_screen_flush_dc_o   = 1'b1;
150.     end
151.     'd9: begin
152.         tft_screen_flush_data_o = SCREEN_HEIGHT[7:0] - 1'b1;    //行地址结束的低
    8 位
153.         tft_screen_flush_dc_o   = 1'b1;
154.     end
155.     //写数据
156.     'd10: begin
157.         tft_screen_flush_data_o = 8'h2C;                        //发送图像数据
158.         tft_screen_flush_dc_o   = 1'b0;
159.     end
160.     default: begin
161.         tft_screen_flush_data_o = spi_screen_flush_data_i;      //图像显示数据
162.         tft_screen_flush_dc_o   = 1'b1;
163.     end
164.     endcase
165. end
166. endmodule
```

此模块接收经过 spi_screen_top 模块转换的 8bit 数据（用户给的是 16bit 的 rgb565 格式的），将其不断发送至屏幕进行显示，以下是对代码的分析：

第 3-23 行：模块的输入输出接口定义。

定义了系统时钟 sys_clk 和复位信号 sys_rst_n。

用户接口包括 spi_screen_flush_data_i（输入的屏幕显示数据，8bit）、spi_screen_flush_updte_o（像素刷新信号）和 spi_screen_flush_fsync_o（屏幕帧同步信号）。

驱动模块接口定义了 tft_screen_flush_req_i（刷新请求输入），tft_screen_flush_data_o（输出的刷新数据），tft_screen_flush_dc_o（DC 控制位，高电平代表本次发送的是数据，低电平代表本次发送的是命令），spi_send_flush_req_o（SPI 发送请求），spi_send_flush_end_o（SPI 发送结束信号），以及 spi_send_flush_ack_i（SPI 发送成功反馈，每拉高一次代表成功发送了一个 8bit 数据）。

第 23-27 行：定义屏幕的参数。

SCREEN_WIDTH 和 SCREEN_HEIGHT 分别表示屏幕宽度 320 和高度 240。

Number_Of_Pixels 表示屏幕像素总数 320*240*2=153600（用户顶层用户接口提供的数据格式是 16bit 的，在这里每 8bit 就发送一次，发送两次才相当于刷新了一个像素数据。像素总数是 320*240 但是总共要发送 320*240*2 次 8bit 数据才能完成一帧图像的刷新，所以这里需要*2）。

第 27-33 行：状态机的状态定义。

S_IDLE 表示空闲状态。

S_DATA 表示发送数据状态。

S_DELAY 表示延时状态（根据芯片数据手册，发送的命令之间需要适当延时）。

S_FRAME_SYNC 表示帧同步状态（发送完成一帧数据）。

第 36 行：DELAY_5clk='d5。

定义了在命令和数据之间切换时需要延迟的时钟周期数，5 个时钟周期。

第 38-40 行：寄存器定义。

flush_cnt 用于记录当前发送数据的进度（其中每一帧的前 10 个数据为芯片配置数据，设置当前帧图像的大小）。

delay_cnt 用于在 S_DELAY 状态下实现延时。

state 和 next_state 用于描述三段式状态机的当前状态和下一个状态。

第 43-44 行：控制 SPI 发送请求和结束信号。

当状态为 S_DATA 时，spi_send_flush_req_o 置为高，发出 SPI 发送请求。

当状态为 S_DELAY 或 S_FRAME_SYNC 时，spi_send_flush_end_o 置为高，表示此时不进行 SPI 数据的传输。

第 47-48 行：像素点刷新和帧同步信号的输出。

当 flush_cnt>=11 为 1 并且 spi_send_flush_ack_i 同时也为 1，spi_screen_flush_updte_o 置为高，已经通过 SPI 发送完一个 8bit 数据。

当状态为 S_FRAME_SYNC 时，spi_screen_flush_fsync_o 置为高，产生帧同步信号。

第 50-55 行：三段式状态机的时序逻辑状态更新。

第 58-81 行：状态机的状态转移逻辑。

在 S_IDLE 状态，如果收到 tft_screen_flush_req_i（刷新请求）信号，进入 S_DATA 状态。

在 S_DATA 状态，首次进入 S_DATA 状态代码 43 行会将 spi_send_flush_req_o 发送请求信号拉高，发送第一个命令。接下来如果 spi_send_flush_ack_i 为 1 且 flush_cnt<=10，进入 S_DELAY 状态；如果 flush_cnt 达到最大像素数，进入 S_FRAME_SYNC。

在 S_DELAY 状态，如果延迟计数器 delay_cnt 达到 DELAY_5clk，回到 S_DATA。

在 S_FRAME_SYNC 状态，状态返回 S_IDLE。

第 84-94 行：数据发送计数器 flush_cnt 的更新逻辑。

如果复位信号为 0，flush_cnt 复位为 0。

如果 spi_send_flush_ack_i 为 1 且 flush_cnt 达到最大像素数+10（这 10 个数据是配置的命令和数据），flush_cnt 复位为 0。

否则 flush_cnt 每次 spi_send_flush_ack_i（spi 每发送成功 8bit 拉高一次） 为 1 时自增 1。

第 97-105 行：延迟计数器 delay_cnt 的更新逻辑。

如果复位信号为 0，delay_cnt 复位为 0。

如果状态为 S_DELAY，delay_cnt 自增。

否则 delay_cnt 复位为 0。

第 108-163 行：根据 flush_cnt 输出相应的 SPI 数据和 DC 控制信号。

flush_cnt==0 时发送命令 8'h2A，表示设置列地址。

flush_cnt==1-4 时依次发送 X 轴的起始和结束地址（每次发送 8bit，总共发送了四次）。

flush_cnt==5 时发送命令 8'h2B，表示设置行地址。

flush_cnt==6-9 时依次发送 Y 轴的起始和结束地址（每次发送 8bit，总共发送了四次）。

flush_cnt==10 时发送命令 8'h2C，表示开始发送图像数据。

其余情况下，发送用户输入的图像数据 spi_screen_flush_data_i。

屏幕数据刷新模块在外部信号 tft_screen_flush_req_i（刷新请求）的触发下开始工作，用一个状态机产生需要写入芯片的数据，并向 spi 发送模块发送请求信号，将命令与数据不断的产生并发送给芯片。在芯片初始化完成之后，这个模块是主要的驱动逻辑。

spi_tft_screen_init 模块

```
1.  `timescale 1ns / 1ps
2.
3.  //对 spi tft 屏幕进行初始化
4.  module spi_tft_screen_init(
5.      input                               sys_clk                 ,
6.      input                               sys_rst_n               ,
7.
8.
9.      input                               tft_screen_init_req_i   ,//初始化请
    求
10.     output                              tft_screen_init_ack_o   ,//初始化完
    成
11.     output reg          [   7: 0]       tft_screen_init_data_o  ,//初始化数
    据
12.     output reg                          tft_screen_init_dc_o    ,//初始化 dc
13.
```

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

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

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

```
171.             tft_screen_init_data_o  = SCREEN_WIDTH[7:0] - 1'b1;     //列地址结束
    的低 8 位
172.             tft_screen_init_dc_o    = 1'b1;
173.         end
174.
175.         'd11: begin
176.             tft_screen_init_data_o  = 8'h2B;                       //RASET//行
    地址设置
177.             tft_screen_init_dc_o    = 1'b0;
178.         end
179.
180.         'd12: begin
181.             tft_screen_init_data_o  = 8'h00;                       //与列地址设
    置相似
182.             tft_screen_init_dc_o    = 1'b1;
183.         end
184.         'd13: begin
185.             tft_screen_init_data_o  = 8'h00;                       //数据
186.             tft_screen_init_dc_o    = 1'b1;
187.         end
188.         'd14: begin
189.             tft_screen_init_data_o  = SCREEN_HEIGHT[15:8];         //数据
190.             tft_screen_init_dc_o    = 1'b1;
191.         end
192.         'd15: begin
193.             tft_screen_init_data_o  = SCREEN_HEIGHT[7:0] - 1'b1;   //数据
194.             tft_screen_init_dc_o    = 1'b1;
195.         end
196.         'd16: begin
197.             tft_screen_init_data_o  = 8'h13;                       //NORON//转
    换为正常显示模式
198.             tft_screen_init_dc_o    = 1'b0;
199.         end
200.         'd17: begin
201.             tft_screen_init_data_o  = 8'h29;                       //DISPON//
    开启显示
202.             tft_screen_init_dc_o    = 1'b0;
203.         end
204.         default: begin
205.             tft_screen_init_data_o  = 8'h01;                       //SWRESET//
    软复位
206.             tft_screen_init_dc_o    = 1'b0;
207.         end
208.     endcase
209. end
210.
211. endmodule
```

屏幕初始化模块是本实验最先执行的逻辑，系统上电阶段 spi_tft_screen_driver 模块会将本模块输入信号 tft_screen_init_req_i（初始化请求信号）拉高，开始对 ST7789VM 芯片进行初始化配置，当配置完成后拉高 tft_screen_init_ack（初始化完成）信号，同时 spi_tft_screen_driver 模块拉低 tft_screen_init_req_i（初始化请求信号）。代表 ST7789VM 芯片初始化配置完成。

以下是本模块代码分析：

第 3-16 行：模块接口定义

这个模块名为 spi_tft_screen_init，主要负责通过 SPI 协议对 TFT 屏幕进行初始化。

sys_clk 是系统时钟输入，sys_rst_n 是系统复位输入。

tft_screen_init_req_i 是初始化请求输入信号，tft_screen_init_ack_o 是初始化完成输出信号。

tft_screen_init_data_o 是初始化数据输出信号，tft_screen_init_dc_o 是命令/数据选择信号（高电平输出数据，低电平输出命令）。

spi_send_init_req_o 是 SPI 发送数据请求信号，spi_send_init_end_o 表示 SPI 发送完成，spi_send_init_ack_i 是 SPI 发送反馈，代表成功发送一个 8bit 配置数据。

第 18-20 行：参数定义

SCREEN_WIDTH 定义屏幕宽度为 320 像素。SCREEN_HEIGHT 定义屏幕高度为 240 像素。

第 22-25 行：延时常量定义

DELAY_255ms 定义了 255 毫秒的延时时间。DELAY_200us 定义了 200 微秒的延时时间。

第 27-31 行：状态机状态定义

S_IDLE 表示空闲状态。

S_SEND_DATA 表示正在发送数据状态。

S_DELAY 表示延时状态。

S_ACK 表示初始化完成的状态。

第 33-36 行：寄存器定义

init_cnt 是用于计数初始化过程中发送命令和数据的计数器。

delay_cnt 是延时计数器。

state 和 next_state 是状态机的当前状态和下一个状态。

第 38-41 行：输出信号的定义

tft_screen_init_ack_o 表示当状态机进入 S_ACK 状态时，初始化完成。

spi_send_init_req_o 表示在 S_SEND_DATA 状态时发出 SPI 数据发送请求（当该信号拉高时请求 spi 数据发送模块发送当前配置数据）。

spi_send_init_end_o 表示在 S_DELAY 状态暂时停止发送配置数据（根据芯片数据手册要求命令与命令之间需要延时 255ms）。

第 43-48 行：状态机状态更新逻辑

如果复位信号 sys_rst_n 为 0，状态机回到 S_IDLE 状态。

否则，状态机的当前状态被更新为 next_state。

第 50-71 行：状态机的状态转移逻辑

在 S_IDLE 状态，当 tft_screen_init_req_i（初始化请求信号）为高时，进入 S_SEND_DATA 状态。

在 S_SEND_DATA 状态，当 SPI 发送完成(spi_send_init_ack_i 为高，spi 发送模块发送完成 8bit 配置数据)时，进入 S_DELAY 状态。

在 S_DELAY 状态，根据不同的初始化命令和数据的计数(init_cnt)来决定是否进行延时或进入下一个状态：

如果 init_cnt=='d18 且延时结束，进入 S_ACK 状态。

否则，针对特定的 init_cnt 值，如 d1,d2,d4,d17，延时结束后进入 S_SEND_DATA 状态。

对于其他情况，当延时结束（delay_cnt==DELAY_200us），进入 S_SEND_DATA 状态。

在 S_ACK 状态，表示初始化完成，状态回到 S_IDLE。

第 73-78 行：初始化计数器 init_cnt 更新逻辑

如果复位信号 sys_rst_n 为 0，init_cnt 清零。

如果 spi_send_init_ack_i（代表 spi 发送模块发送完成 8bit 配置数据）为高，init_cnt 自增。

第 80-85 行：延时计数器 delay_cnt 更新逻辑

如果复位信号为 0，delay_cnt 清零。

如果当前状态是 S_DELAY，则 delay_cnt 自增，否则清零。

第 87-123 行：根据 init_cnt 输出相应的初始化命令和数据

init_cnt==0 时，发送复位命令 8'h01（软件复位）。

init_cnt==1 时，发送命令 8'h11（唤醒命令）。

init_cnt==2 时，发送命令 8'h3A（设置像素格式），紧接着是 8'h55，表示使用 RGB565 格式。

init_cnt==4 时，发送命令 8'h36（芯片内存数据的读写扫描方向），紧接着是 8'h78，设置扫描方式。

其他 init_cnt 对应列地址设置、行地址设置、显示模式为正常、以及开启显示命令，依次为 8'h2A,8'h2B,8'h138'h29

spi_master_driver 模块：

```
1.  `timescale 1ns / 1ps
2.
3.  // //模式 0：CPOL= 0，CPHA=0。SCK 串行时钟线空闲是为低电平，数据在 SCK 时钟的上升沿被采样，数据在 SCK 时钟的下降沿切换
4.  // //模式 1：CPOL= 0，CPHA=1。SCK 串行时钟线空闲是为低电平，数据在 SCK 时钟的下降沿被采样，数据在 SCK 时钟的上升沿切换
5.  // //模式 2：CPOL= 1，CPHA=0。SCK 串行时钟线空闲是为高电平，数据在 SCK 时钟的下降沿被采样，数据在 SCK 时钟的上升沿切换
6.  // //模式 3：CPOL= 1，CPHA=1。SCK 串行时钟线空闲是为高电平，数据在 SCK 时钟的上升沿被采样，数据在 SCK 时钟的下降沿切换
7.
8.
9.  //模式 3：CPOL= 1，CPHA=1。SCK 串行时钟线空闲是为高电平，数据在 SCK 时钟的上升沿被采样，数据在 SCK 时钟的下降沿切换
10. module spi_master_driver(
11.     //系统接口
12.     input                               sys_clk                 ,
13.     input                               sys_rst_n               ,
14.
15.     //用户接口
16.     input                               spi_start_i             ,// spi 开始信号
17.     input                               spi_end_i               ,// spi 结束信号
18.     input               [   7: 0]       spi_send_data_i         ,// spi 发送数据
19.     output      reg                     spi_send_ack_o          ,// spi 发送 8bit 数据完成信号
20.
21.     input                               lcd_dc_i                ,//数据还是命令信号输入
22.     output reg                          lcd_dc                  ,//数据还是命令信号输出
23.     //spi  端口
24.     output reg                          spi_sclk                ,//spi 时钟
25.     output reg                          spi_mosi                ,//spi 数据
```

```
    output                                  spi_cs                          //spi 使能
);
localparam  IDLE = 3'b001,  //空闲状态
            DATA = 3'b010,  //发送数据状态
            STOP = 3'b100;  //停止状态
    reg                 [2:0]       cure_state;//状态寄存器
    reg                 [2:0]       next_state;//下一状态寄存器

    reg                 [7:0]       spi_send_data_reg           ;//数据寄存器
    reg                 [3:0]       spi_send_data_bit_cnt       ;//数据发送 bit 计数器

assign spi_cs = (cure_state == IDLE) ?1:0;  //片选信号，低电平有效，为空闲状态时拉高
always @(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0 )
        cure_state <= IDLE;
    else
        cure_state <= next_state;
always @(*)
    case(cure_state)
        IDLE:begin  //空闲
            if(spi_start_i) //spi_start 开始信号来临时，开始进行数据发送
                next_state = DATA;
            else
                next_state = IDLE;
        end
        DATA:begin  //发送数据
            if(spi_send_data_bit_cnt == 7 &&  spi_sclk == 1'b0) //字节发送完毕
                next_state = STOP;
            else
                next_state = DATA;
        end
        STOP:next_state = IDLE; //停止
        default:next_state = IDLE;
    endcase

//发送数据缓存
always@(posedge sys_clk or negedge sys_rst_n) begin
    if( sys_rst_n == 1'b0 )
        spi_send_data_reg <= 'd0;
    else if(spi_send_data_bit_cnt == 'd0)         //8bit 数据发送完成后，缓存新的数据
        spi_send_data_reg <= spi_send_data_i;
    else
        spi_send_data_reg <= spi_send_data_reg;
end
always@(posedge sys_clk or negedge sys_rst_n) begin
    if( sys_rst_n == 1'b0 )
        spi_send_ack_o <= 'd0;
    else if(spi_send_data_bit_cnt == 'd7 && spi_sclk == 1'b0 && spi_cs == 1'b0)
    //发送完 8 位数据后，spi 发送 8bit 数据完成信号拉高
        spi_send_ack_o <= 'd1;
    else
        spi_send_ack_o <= 'd0;
end
//数据是命令还是数据
always@(posedge sys_clk or negedge sys_rst_n) begin
    if( sys_rst_n == 1'b0)
        lcd_dc <= 1'b1;
    else if( spi_start_i == 1'b1)  //接收到开始信号时，lcd_dc 赋值为 lcd_dc_i
```

```
83.         lcd_dc <= lcd_dc_i;
84.     else
85.         lcd_dc <= lcd_dc;
86. end
87. //产生 spi 时钟
88. always@(posedge sys_clk or negedge sys_rst_n) begin
89.     if( sys_rst_n == 1'b0)
90.         spi_sclk <= 1'b1;
91.     else if(cure_state != DATA )
92.         spi_sclk <= 1'b1;
93.     else if( spi_cs == 1'b0 )
94.         spi_sclk <= ~spi_sclk;  //当 spi_cs 低电平时，翻转 spi_sclk，生成 40ns 周期的
    sclk 时钟
95.     else
96.         spi_sclk <= 1'b1;
97. end
98. //数据发送 bit 数寄存器
99. always@(posedge sys_clk or negedge sys_rst_n) begin
100.     if( sys_rst_n == 1'b0)
101.         spi_send_data_bit_cnt <= 'd0;
102.     else if( spi_cs == 1'b0 && spi_sclk == 1'b0) //使用 SPI 模式三，所以数据在下降沿
    切换，所以当时钟为低电平时，数据计数器加 1
103.         if( spi_send_data_bit_cnt == 'd7)//发送完 8 位数据，清零
104.             spi_send_data_bit_cnt <= 'd0;
105.         else
106.             spi_send_data_bit_cnt <= spi_send_data_bit_cnt + 1'b1;
107.     else if( spi_cs == 1'b0)
108.         spi_send_data_bit_cnt <= spi_send_data_bit_cnt;//在时钟上升沿时，保持不
    变
109.     else
110.         spi_send_data_bit_cnt <= 'd0;
111. end
112.
113. //spi 数据发送
114. always@(posedge sys_clk or negedge sys_rst_n) begin
115.     if( sys_rst_n == 1'b0)
116.         spi_mosi <= 1'b1;
117.     else if( spi_cs == 1'b0)
118.         spi_mosi <= spi_send_data_reg['d7 - spi_send_data_bit_cnt];//将数据从高
    位到低位逐位发送
119.     else
120.         spi_mosi <= spi_mosi;
121. end
122.
123. endmodule
```

本模块为 SPI 发送模块，使用时钟相位时钟极性均设置为 1。也就是时钟低电平有效，数据采样边沿为偶边沿。本模块为其他模块发送配置信息，也负责图像数据的发送。以下是具体分析：

模块使用一个状态机来控制来 SPI 通信的过程，其包括三个状态：

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

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

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

模块的核心功能是实现数据的逐位传输。

我们首先来看 SPI 时钟的生成逻辑：在代码第 88~97 行，SPI 时钟（spi_sclk）的生成由状态机控制。在数据发送状态下，当 spi_cs 低电平时，将时钟信号周期性地反转，生成一个周期为 40ns 的时钟，用于同步 MOSI 数据线的变化。模块使用的是 SPI 模式 3，因此时钟在空闲时为高电平，数据在时钟的上升沿被采样，数据在下降沿切换。

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

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

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

10.4、实验现象

新建工程，添加 rtl 代码，添加 fdc 文件；本实验管脚约束如下：（示例工程

中也有仿真代码，读者若不清楚时序也可自行仿真分析）

信号名称(括号中为屏幕原理图管脚名称)	管脚约束
sys_clk	5
sys_rst_n	20
lcd_spi_sclk (SCLK)	75
lcd_spi_mosi (SDA)	73
lcd_reset (RST)	71
lcd_dc (DC)	69
lcd_spi_cs (CS)	67
lcd_blk (BL)	62

图 3-10.4-1 SPI 屏幕驱动显示实验管脚约束

也可直接添加 fdc 约束文件：

```
1.  create_clock -name {clk} [get_ports {sys_clk}] -period {20.000} -waveform {0.000 10.000}
2.  define_attribute {p:lcd_blk} {PAP_IO_DIRECTION} {OUTPUT}
3.  define_attribute {p:lcd_blk} {PAP_IO_LOC} {62}
4.  define_attribute {p:lcd_blk} {PAP_IO_VCCIO} {1.2}
5.  define_attribute {p:lcd_blk} {PAP_IO_STANDARD} {LVCMOS12}
6.  define_attribute {p:lcd_blk} {PAP_IO_DRIVE} {2}
7.  define_attribute {p:lcd_blk} {PAP_IO_NONE} {TRUE}
8.  define_attribute {p:lcd_blk} {PAP_IO_SLEW} {SLOW}
9.  define_attribute {p:lcd_dc} {PAP_IO_DIRECTION} {OUTPUT}
10. define_attribute {p:lcd_dc} {PAP_IO_LOC} {69}
11. define_attribute {p:lcd_dc} {PAP_IO_VCCIO} {1.2}
12. define_attribute {p:lcd_dc} {PAP_IO_STANDARD} {LVCMOS12}
13. define_attribute {p:lcd_dc} {PAP_IO_DRIVE} {2}
14. define_attribute {p:lcd_dc} {PAP_IO_NONE} {TRUE}
15. define_attribute {p:lcd_dc} {PAP_IO_SLEW} {SLOW}
16. define_attribute {p:lcd_reset} {PAP_IO_DIRECTION} {OUTPUT}
17. define_attribute {p:lcd_reset} {PAP_IO_LOC} {71}
18. define_attribute {p:lcd_reset} {PAP_IO_VCCIO} {1.2}
19. define_attribute {p:lcd_reset} {PAP_IO_STANDARD} {LVCMOS12}
20. define_attribute {p:lcd_reset} {PAP_IO_DRIVE} {2}
21. define_attribute {p:lcd_reset} {PAP_IO_NONE} {TRUE}
22. define_attribute {p:lcd_reset} {PAP_IO_SLEW} {SLOW}
23. define_attribute {p:lcd_spi_cs} {PAP_IO_DIRECTION} {OUTPUT}
```

```
24. define_attribute {p:lcd_spi_cs} {PAP_IO_LOC} {67}
25. define_attribute {p:lcd_spi_cs} {PAP_IO_VCCIO} {1.2}
26. define_attribute {p:lcd_spi_cs} {PAP_IO_STANDARD} {LVCMOS12}
27. define_attribute {p:lcd_spi_cs} {PAP_IO_DRIVE} {2}
28. define_attribute {p:lcd_spi_cs} {PAP_IO_NONE} {TRUE}
29. define_attribute {p:lcd_spi_cs} {PAP_IO_SLEW} {SLOW}
30. define_attribute {p:lcd_spi_mosi} {PAP_IO_DIRECTION} {OUTPUT}
31. define_attribute {p:lcd_spi_mosi} {PAP_IO_LOC} {73}
32. define_attribute {p:lcd_spi_mosi} {PAP_IO_VCCIO} {1.2}
33. define_attribute {p:lcd_spi_mosi} {PAP_IO_STANDARD} {LVCMOS12}
34. define_attribute {p:lcd_spi_mosi} {PAP_IO_DRIVE} {2}
35. define_attribute {p:lcd_spi_mosi} {PAP_IO_NONE} {TRUE}
36. define_attribute {p:lcd_spi_mosi} {PAP_IO_SLEW} {SLOW}
37. define_attribute {p:lcd_spi_sclk} {PAP_IO_DIRECTION} {OUTPUT}
38. define_attribute {p:lcd_spi_sclk} {PAP_IO_LOC} {75}
39. define_attribute {p:lcd_spi_sclk} {PAP_IO_VCCIO} {1.2}
40. define_attribute {p:lcd_spi_sclk} {PAP_IO_STANDARD} {LVCMOS12}
41. define_attribute {p:lcd_spi_sclk} {PAP_IO_DRIVE} {2}
42. define_attribute {p:lcd_spi_sclk} {PAP_IO_NONE} {TRUE}
43. define_attribute {p:lcd_spi_sclk} {PAP_IO_SLEW} {SLOW}
44. define_attribute {p:sys_clk} {PAP_IO_DIRECTION} {INPUT}
45. define_attribute {p:sys_clk} {PAP_IO_LOC} {5}
46. define_attribute {p:sys_clk} {PAP_IO_VCCIO} {1.2}
47. define_attribute {p:sys_clk} {PAP_IO_STANDARD} {LVCMOS12}
48. define_attribute {p:sys_clk} {PAP_IO_PULLUP} {TRUE}
49. define_attribute {p:sys_rst_n} {PAP_IO_DIRECTION} {INPUT}
50. define_attribute {p:sys_rst_n} {PAP_IO_LOC} {20}
51. define_attribute {p:sys_rst_n} {PAP_IO_VCCIO} {1.2}
52. define_attribute {p:sys_rst_n} {PAP_IO_STANDARD} {LVCMOS12}
53. define_attribute {p:sys_rst_n} {PAP_IO_PULLUP} {TRUE}
```

综合生成比特流，将程序下载进入板卡，由此可见实验设计成功：

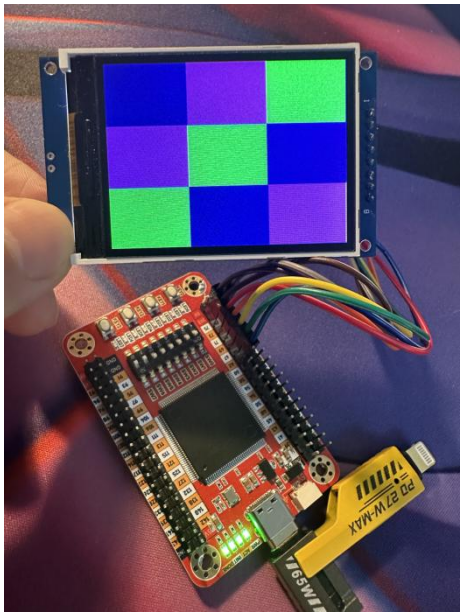


图 3-10.4-2 SPI 屏幕驱动显示实验效果图

11、PLL IP 使用实验例程

11.1、实验目的

了解 Compact 系列的 PLL 的使用及配置方法。

11.2、实验原理

11.2.1、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。

11.2.2、IP 配置

首先点击快捷工具栏的“IP”图标，进入 IP 例化设置

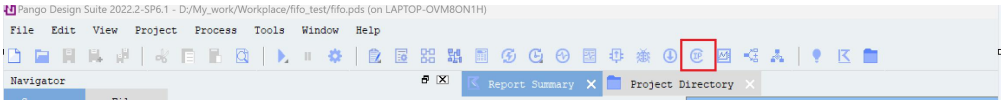


图 3-11.2-1 “IP”图标示意图

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：

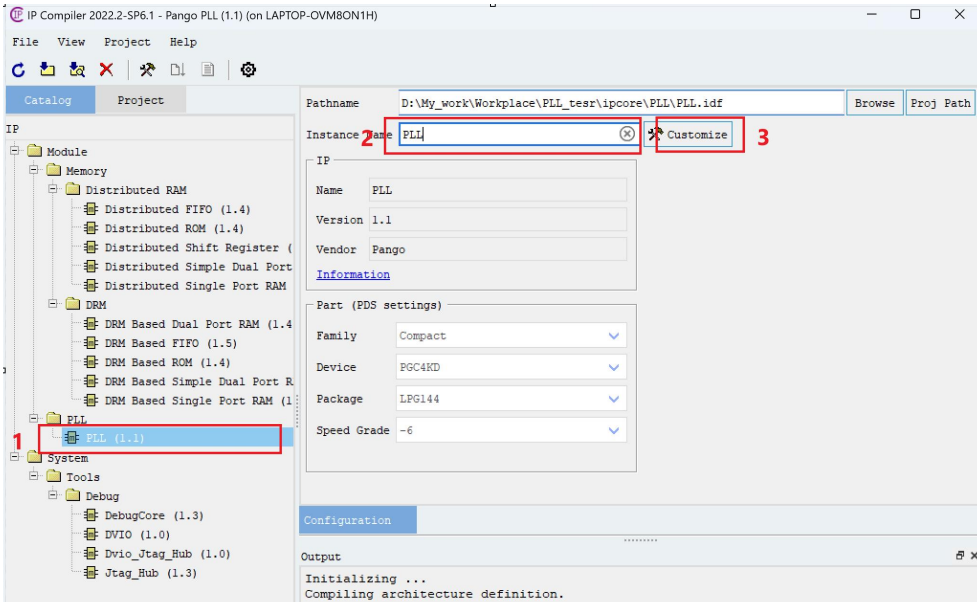


图 3-11.2-2 IP 配置流程图 1

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。

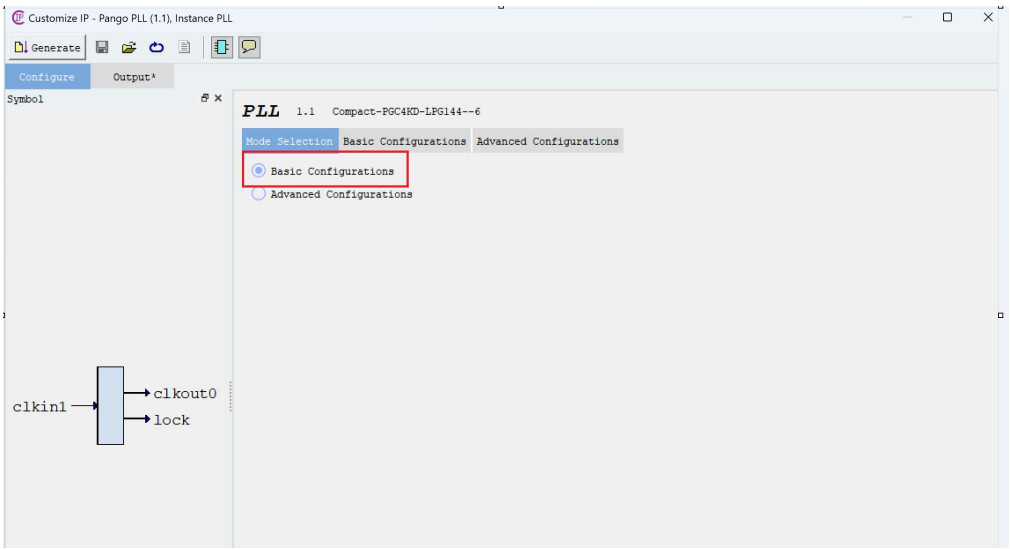


图 3-11.2-3 IP 配置流程图 2

接下来进行基础配置：

在 Public Configurations 一栏将输入时钟频率设置为 50MHZ。

在 Clockout0 Configurations 选项卡下，勾选使能 clkout0，将输出频率设置为 25MHZ。

在 Clockout1 Configurations 选项卡下，勾选使能 clkout1，将输出频率设置为 100MHZ。

在 Clockout2 Configurations 选项卡下，勾选使能 clkout2，将输出频率设置为 100MHZ，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：

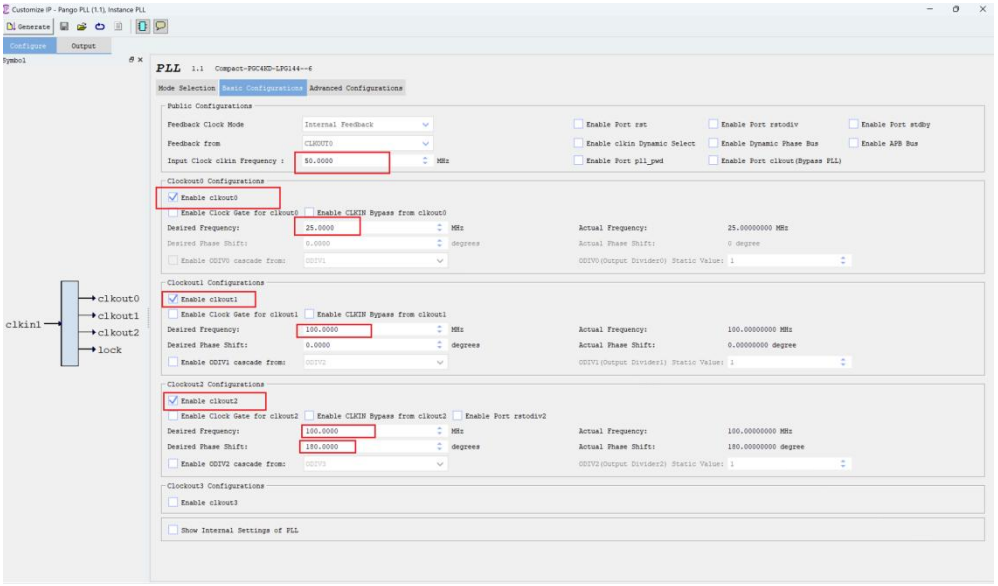


图 3-11.2-4 IP 配置流程图 3

点击左上角 generate 生成 IP。

11.3、代码设计

模块接口列表如下所示：

表 3-11-1 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	25MHZ 时钟
clkout1	output	1	100MHZ 时钟
clkout2	output	1	100MHZ 时钟，相位偏移 180 度
lock	output	1	时钟锁定信号，当为高电平时，代表 IP 核输出时钟稳定。

PLL_TEST 顶层代码：

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

该模块的功能是例化 PLL IP 核，功能简单，在此不做说明。

PLL_tb 测试代码：

```
1.  timescale 1ns / 1ps
2.
3.  module   PLL_tb();
4.     reg                          sys_clk              ;
5.     wire                         clkout0              ;
6.     wire                         clkout1              ;
7.     wire                         clkout2              ;
8.     wire                         lock                 ;
9.
10.
11.
12.    initial
13.       begin
14.          #2
15.                sys_clk <= 0    ;
16.       end
17.
18.    parameter   CLK_FREQ = 50;//Mhz
19.    always # ( 1000/CLK_FREQ/2 ) sys_clk = ~sys_clk ;
20.
21.
22. PLL_TEST u_PLL_TEST(
23.    .sys_clk                    (sys_clk             ),
24.    .clkout0                    (clkout0             ),
25.    .clkout1                    (clkout1             ),
26.    .clkout2                    (clkout2             ),
27.    .lock                      (lock                )
28. );
29.
30.
31. endmodule
```

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

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

代码定义了一个时钟频率参数 CLK_FREQ 为 50 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 10 纳秒翻转一次，从而生成一个 50 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。

11.4、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。

首先需要编译我们的 PDS 仿真库文件，具体如下所示：

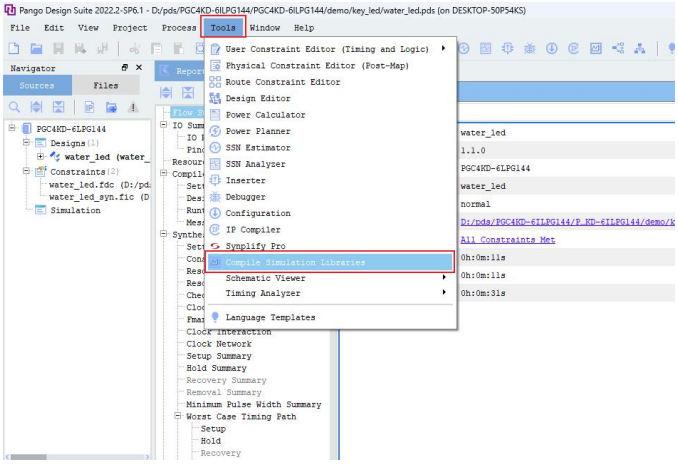


图 3-11.4-1 PDS 和 Modelsim 联合仿真流程图 1

打开 PDS 工程后，选择工具栏里的 Tools->Comolie Simulation Libraries。

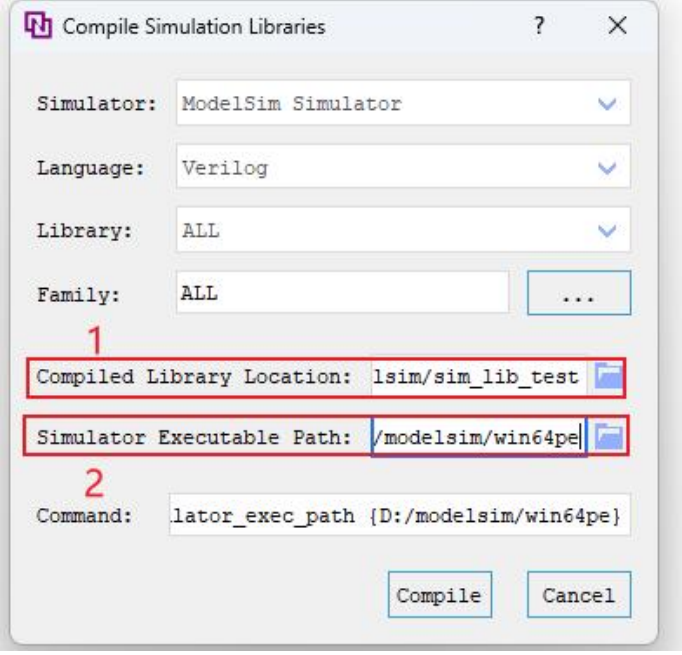


图 3-11.4-2 PDS 和 Modelsim 联合仿真流程图 2

红框 1 选择存放生成的仿真库的路径，红框 2 选择 Modelsim 的启动程序路径，一般在 win64pe 或者 win64 或者 win32 文件夹，具体和 Modelsim 的版本有关，笔者所用的 Modelsim 为 10.6c。

选择好路径后，点击 Compile，然后等待编译结束即可。

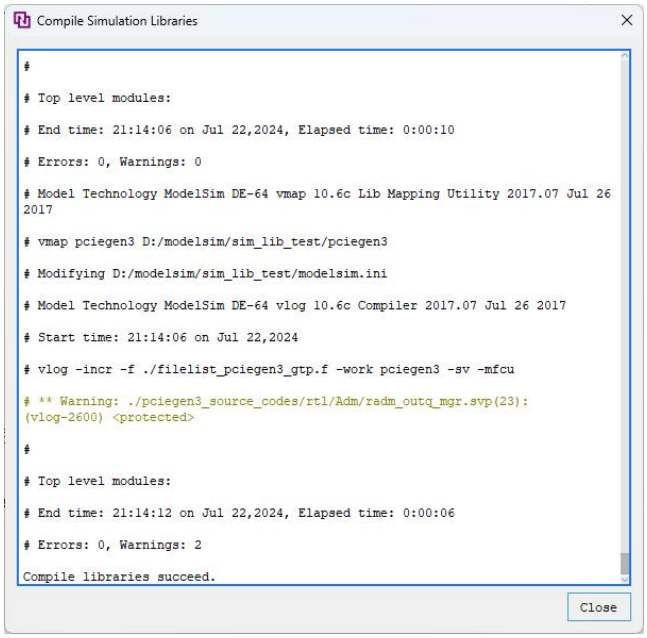


图 3-11.4-3 PDS 和 Modelsim 联合仿真流程图 3

当出现以上结果时，表示生成成功，点击 Close。

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

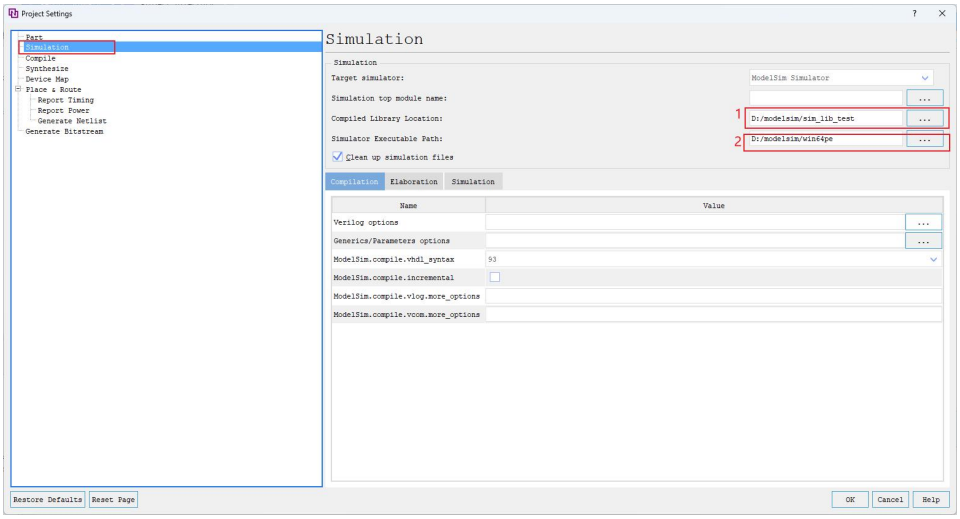


图 3-11.4-4 PDS 和 Modelsim 联合仿真流程图 4

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。

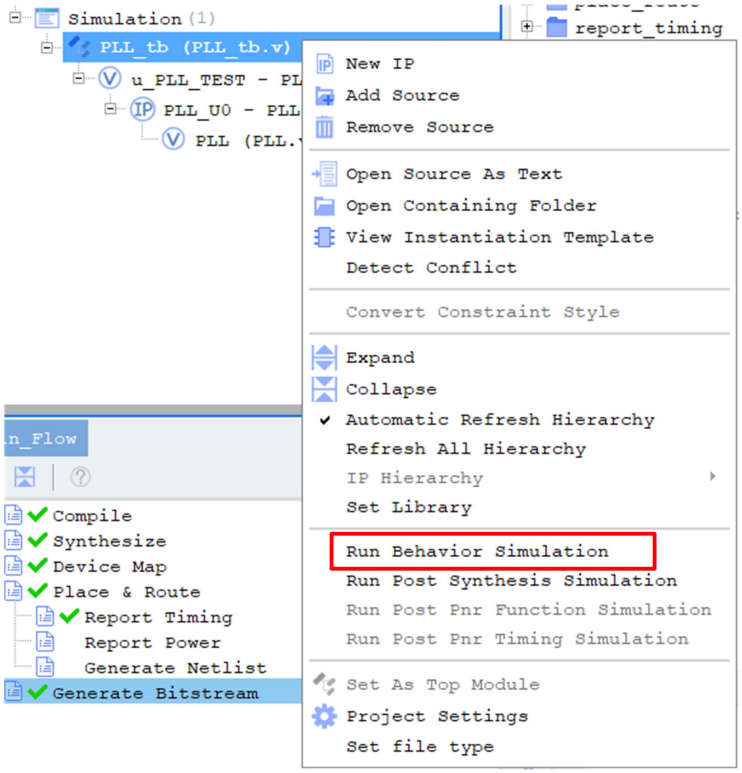


图 3-11.4-5 PDS 和 Modelsim 联合仿真流程图 5

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。

```
# -- Compiling module PLL_TEST
#
# Top level modules:
#       PLL_TEST
# End time: 19:01:57 on Nov 21,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim SE-64 vlog 2020.4 Compiler 2020.10 Oct 13 2020
# Start time: 19:01:57 on Nov 21,2024
# vlog -reportprogress 300 F:/sx/Topic1/PGC4K_IP/PLL_test/ipcore/PLL/PLL.v -work work
# -- Compiling module PLL
#
# Top level modules:
#       PLL
# End time: 19:01:57 on Nov 21,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim SE-64 vlog 2020.4 Compiler 2020.10 Oct 13 2020
# Start time: 19:01:57 on Nov 21,2024
# vlog -reportprogress 300 F:/sx/Topic1/PGC4K_IP/PLL_test/source/PLL_tb.v -work work
# -- Compiling module PLL_tb
#
# Top level modules:
#       PLL_tb
# End time: 19:01:57 on Nov 21,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# do {run_behav_simulate.tcl}
# vsim -voptargs=""+acc"" -L work -L usim -L adc -L iolhr_dft -L iserdes_e1 -L oserdes_e1 PLL_tb usim.GTP_GRS
# Start time: 19:01:58 on Nov 21,2024
# ** Note: (vsim-3812) Design is being optimized...
# Loading work.PLL_tb(fast)
# Loading work.PLL_TEST(fast)
# Loading work.PLL(fast)
# Loading usim.GTP_PLL_E2(fast)
# Loading usim.GTP_GRS(fast)
# .main_pane.wave.interior.cs.body.pw.wf
# .main_pane.structure.interior.cs.body.struct
# .main_pane.objects.interior.cs.body.tree

VSIM 3>
```

图 3-11.4-6 PDS 和 Modelsim 联合仿真流程图 6

11.5、实验现象

点击 Wave 观察 PLL 输出信号：

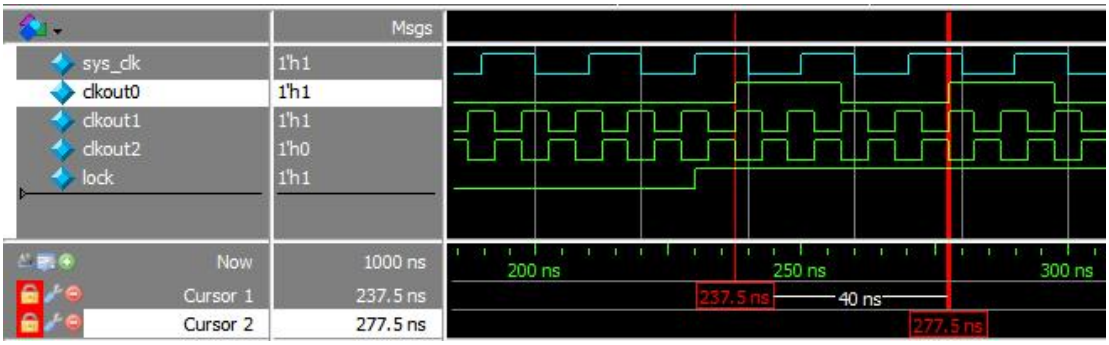


图 3-11.5-1 PLL IP 使用实验结果波形图 1

使用标尺测量 clkout0，发现其一个时钟周期是 40ns，也就是 25MHZ。符合 PLL IP 设置。

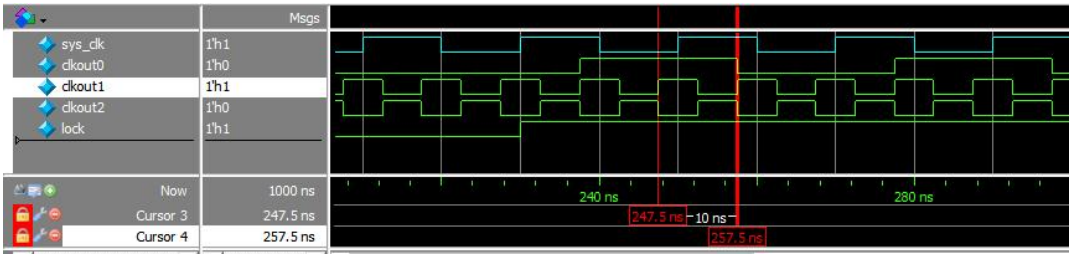


图 3-11.5-2 PLL IP 使用实验结果波形图 2

使用标尺测量 clkout1，发现其一个时钟周期是 10ns，也就是 100MHZ，符合 PLL IP 设置，同时发现 clkout2 时钟相位超前 clkout1 半个时钟周期，也就是 180 度，也符合 PLL IP 设置。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。

12、FIFO IP 使用实验例程

12.1、实验目的

了解 Compact 系列的 FIFO IP 的使用以及配置的方法。

12.2、实验原理

12.2.1、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。

12.2.2、IP 配置

首先点击快捷工具栏的“IP”图标，进入 IP 例化设置

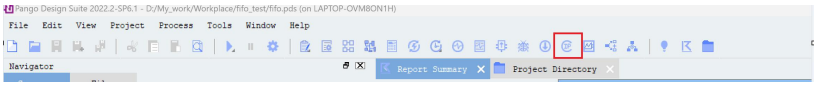


图 3-12.2-112.2-2-12.2-3 “IP”图标示意图

然后在 IP 目录处选择 DRM Based FIFO，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：

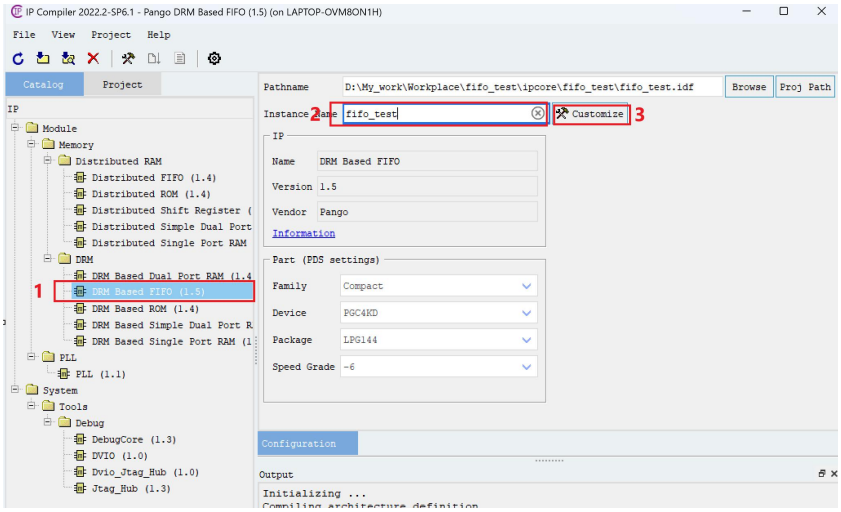


图 3-12-12.2-4 FIFO IP 配置流程图 1

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。

本次实验我们选择异步 FIFO（ASYNC），不启用字节读写功能，写地址深度位宽设置为 11，写数据位宽设置为 8，读地址深度位宽设置为 11，读数据位宽设置为 8 位。然后使能 almost_full_water_level 和 almost_empty_water_level，用来观察 FIFO 中数据写入和读出的情况。 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，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。

而下面的 Almost Full Numbers 的设置是表示当写入 124 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。

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。

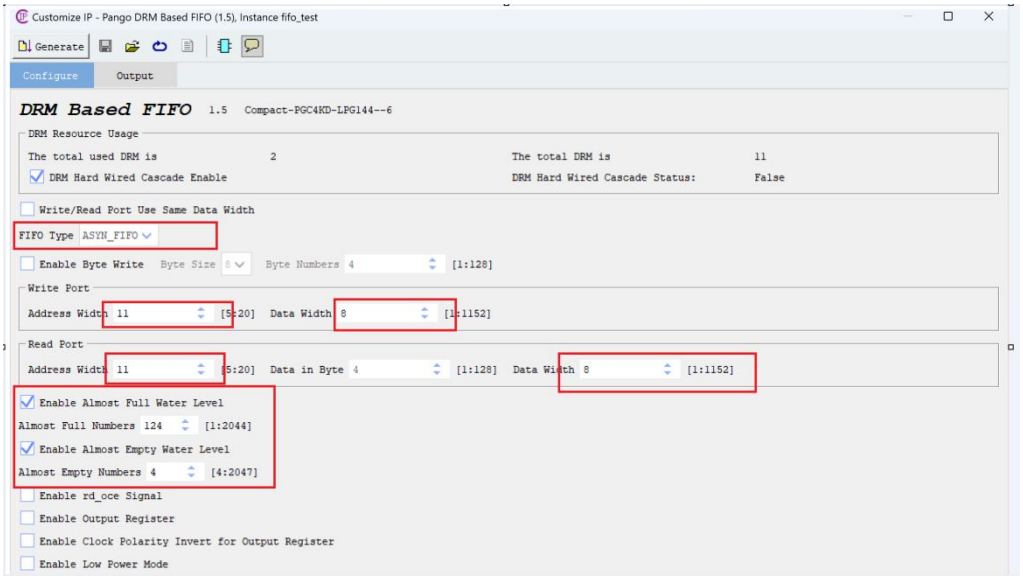


图 3-12.2-5 FIFO IP 配置流程图 1

配置完成的 FIFO 端口如下：

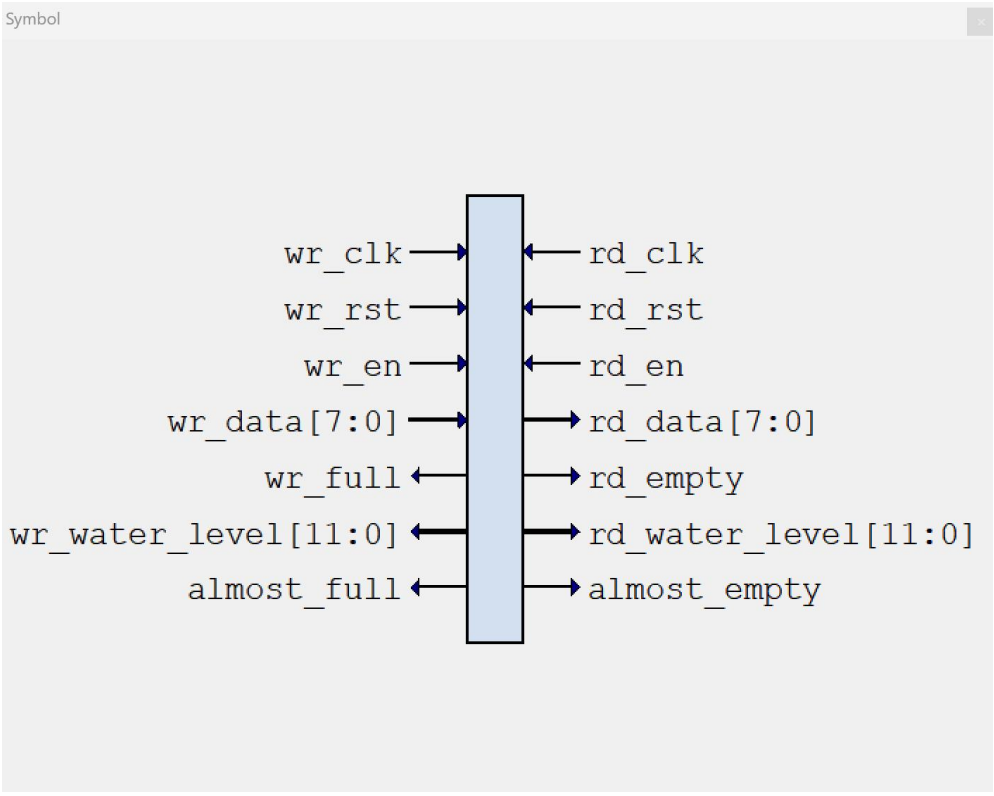


图 3-12.2-6 FIFO 端口示意图

其他各个端口的信号描述我们可以参考 IP 手册：

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 3-12.2-7 IP 手册端口信号描述图

接着我们点击左上角的 generate 按钮，生成 IP；

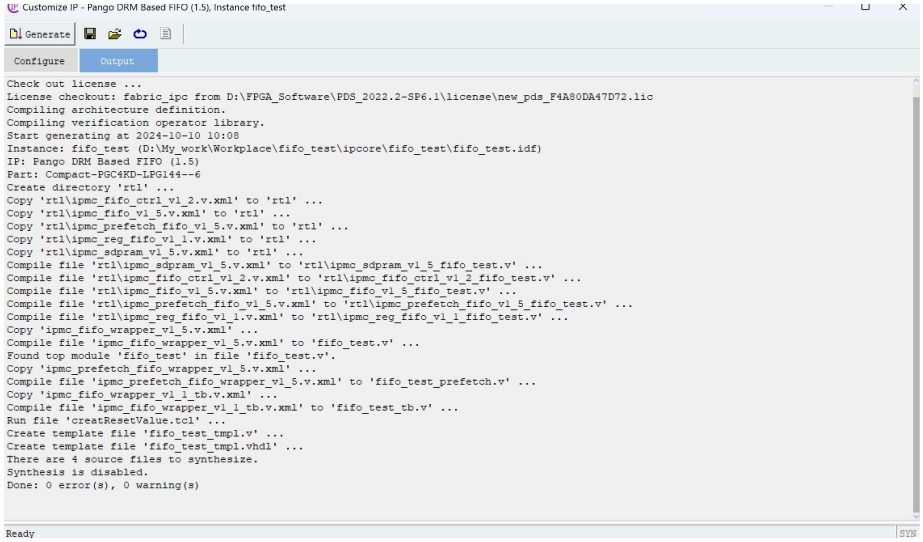


图 3-12.2-8 IP 生成界面结果图

提示 0 错误 0 警告即 IP 生成完毕，同时 PDS 将自动打开实例化模板文件 fifo_test_tmpl.v,我们打开这个模板文件即可方便的将 IP 实例化进我们的设计中。

12.2.3、FIFO 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)。

12.2.3.1、FIFO 未满时的写时序

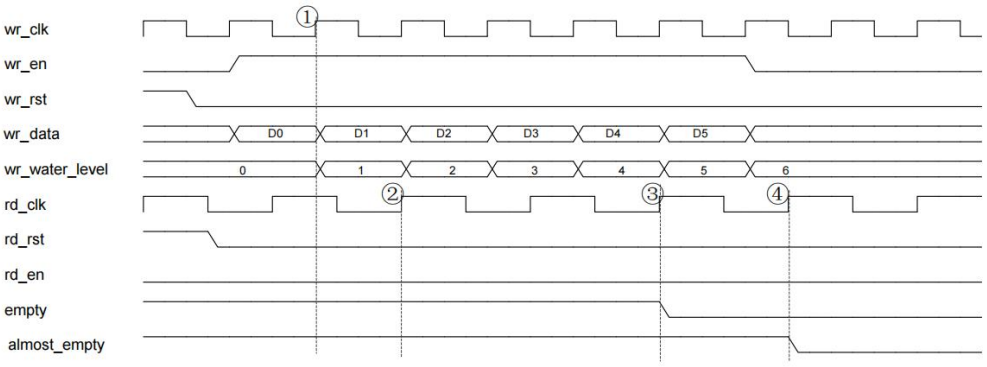


图 3-12.2-9 FIFO 未满写时序图

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。

12.2.3.2、FIFO 将满时的写时序

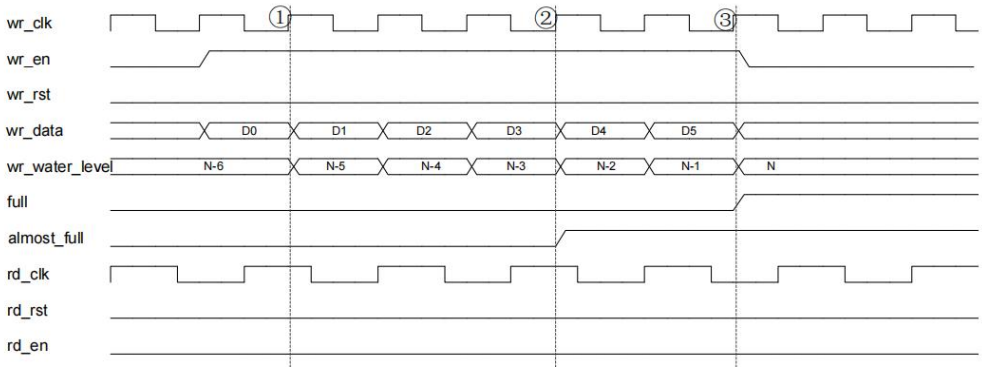


图 3- 12.2- 10 FIFO 将满写时序图

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。

12.2.3.3、FIFO 在满状态下的读时序

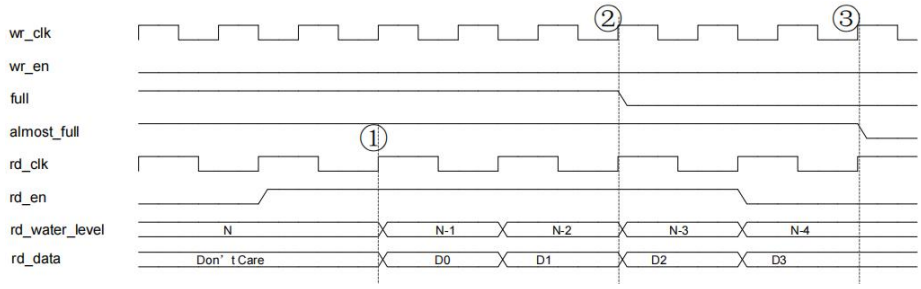


图 3- 12.2- 11 FIFO 满状态读时序图

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。

12.2.3.4、FIFO 将空时的读时序

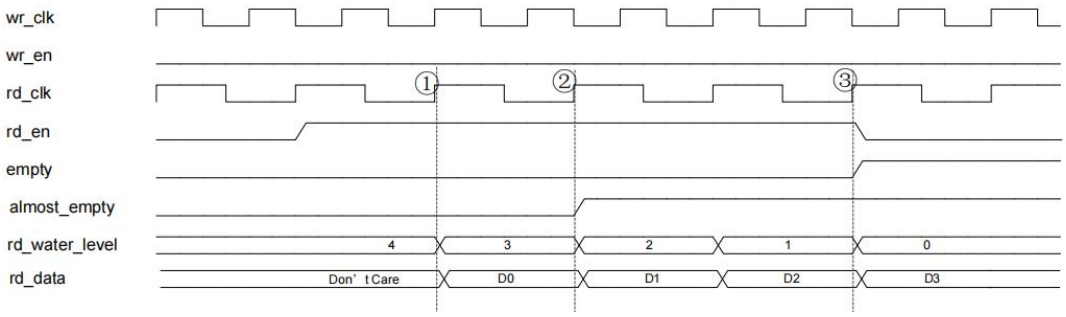


图 3- 12.2- 12 FIFO 将空读时序图

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。

12.3、代码设计

顶层代码端口列表如下：

表 3-12-1 FIFO IP 实验顶层代码端口表

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

FIFO 顶层模块：

```
1.  //FIFO 测试顶层
2.  module fifo_test_top
3.  (
4.      input  wire        sys_clk      ,
5.      input  wire        rst_n        ,
6.
7.      input  wire [7:0]  wr_data      ,
8.      input  wire        wr_en        ,
9.      input  wire        rd_en        ,
10.
11.     output wire [7:0]  wr_water_level ,
12.     output wire [7:0]  rd_water_level ,
13.
14.     output wire [7:0]  rd_data
15.
16. );
```

```
17.
18.  wire          wr_full;
19.  wire          almost_full;
20.  wire          rd_empty;
21.  wire          almost_empty;
22.
23.  fifo_test fifo_test_inst (
24.    .wr_clk(sys_clk),                 // input
25.    .wr_rst(~rst_n),                  // input
26.    .wr_en(wr_en),                    // input
27.    .wr_data(wr_data),                // input [7:0]
28.    .wr_full(wr_full),                // output
29.    .wr_water_level(wr_water_level),  // output [11:0]
30.    .almost_full(almost_full),        // output
31.
32.    .rd_clk(sys_clk),                 // input
33.    .rd_rst(~rst_n),                  // input
34.    .rd_en(rd_en),                    // input
35.    .rd_data(rd_data),                // output [7:0]
36.    .rd_empty(rd_empty),              // output
37.    .rd_water_level(rd_water_level),  // output [11:0]
38.    .almost_empty(almost_empty)       // output
39.  );
40.  endmodule
```

该模块就是简单的实例化了 FIFOIP，将读写控制信号引出，不需要过多介绍。

fifo_test_tb.v 测试文件代码如下：

```
1.   timescale 1ns/1ns
2.   module fifo_test_tb();
3.
4.   reg sys_clk;
5.   reg rst_n;
6.
7.   reg       [7:0]   wr_data;
```

```
8.  reg             wr_en;
9.  reg             rd_en;
10.
11. reg             rd_state;   //读状态
12. reg             wr_state;
13.
14. wire      [7:0]  rd_data;
15. reg       [7:0]  rd_cnt;
16.
17. wire      [7:0]  rd_water_level;
18. wire      [7:0]  wr_water_level;
19.
20. initial
21. begin
22.     rst_n  <=  1'd0;
23.     sys_clk <=  1'd0;
24.     #20
25.     rst_n  <=  1'd1;
26.
27.
28. end
29.
30. always#10 sys_clk = ~sys_clk;   //50MHZ
31.
32. always@(posedge sys_clk or negedge rst_n)  begin
33.     if(!rst_n)
34.     begin
35.         wr_state   <=   1'd0;
36.         wr_en  <=  1'd0;
37.         wr_data <=  8'd0;
38.     end
39.     else
40.     begin
41.         case(wr_state)
42.             1'd0:   if(wr_water_level == 127)   //128 个数据
43.                     begin
44.                         wr_en   <=   #2 1'd0;
```

```
45.                     wr_data <=   #2 8'd0;
46.                     wr_state <=   #2 1'd1;
47.                 end
48.                 else
49.                 begin
50.                     wr_en   <=   #2 1'd1;
51.                     wr_data <=   #2 wr_data+1'b1;
52.                     wr_state <=   #2 1'd0;
53.                 end
54.
55.             1'd1:   if(rd_cnt == 127)
56.                         wr_state   <=   #2 1'd0;
57.
58.
59.             default:   wr_state   <=1'd0;
60.         endcase
61.     end
62. end
63.
64. always@(posedge sys_clk or negedge rst_n)   begin
65.     if(!rst_n)
66.     begin
67.         rd_state<=   1'd0;
68.         rd_en   <=   1'd0;
69.         rd_cnt  <=   8'd0;
70.     end
71.     else
72.     begin
73.         case(rd_state)
74.             1'd0:   if(rd_water_level >= 8'd128)   //等待 128 个数据
75.                 begin
76.                     rd_state   <=   #2 1'd1;
77.                     rd_en      <=   #2 1'd1;
78.                 end
79.                 else
80.                 begin
81.                     rd_cnt     <=  #2 8'd0;
```

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

本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，因为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，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。

12.4、实验现象

右键仿真的文件,选择 Run Behavior Simulation 开始行为仿真。接下来我们启动 Modelsim 观察仿真波形

图 3-12.4-1 数据写入结果波形图

可以发现当 wr_en 为高电平时 wr_data 被写入 FIFO。同时在下一个时钟 wr_water_level 变为 1，表示 FIFO 中写入了一个数据。同时 rd_water_level 在三个时钟周期后加一，也符合之前对 fifo 的时序分析的时序分析。

图 3-12.4-2 数据写满结果波形图

当写入 128 个数据之后 wr_state 转变为 1，并且拉低 wr_en 停止数据的写入，符合 tb 代码中的设计，同时在下一个时钟 wr_water_level 变为 128，

rd_water_level 在三个时钟周期后也变为 128。虽然在这里我们拉低了写使能进入读数据的状态，但是不代表 FIFO 数据的读写不能同时进行，事实上同时对 FIFO 的读写端口进行操作是十分正常的。

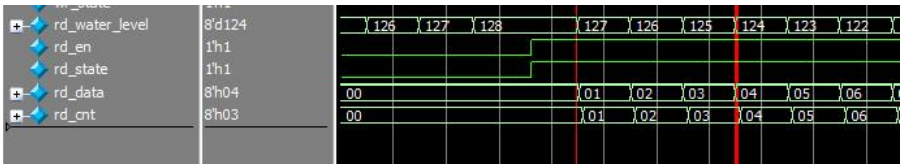


图 3-12.4-3 数据读出结果波形图

接下来进行读数据的状态，rd_state 变为 1，拉高读使能 rd_en。这时数据在下一个时钟周期被读出，因为 FIFO 的先进先出的特性，第一个被写入的数据是 1，所以读出的第一个数据也是 1，同时 rd_water_leve 减一。

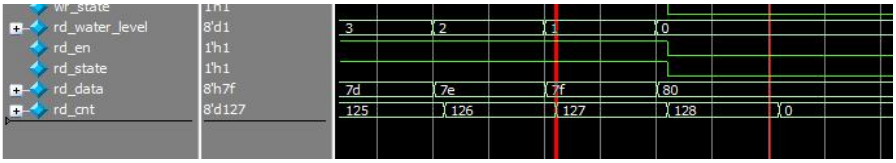


图 3-12.4-4 数据读空结果波形图

当读数据计数器计数计数到 127 时拉低读使能，下一个时钟周期后最后一个数据 128 被读出。

13、ROM IP 使用实验例程

13.1、实验目的

了解 Compact 系列的 ROM IP 的使用以及配置的方法。

13.2、实验原理

13.2.1、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 如图所示：

13.2.2、IP 配置

首先点击快捷工具栏的“IP”图标，进入 IP 例化设置

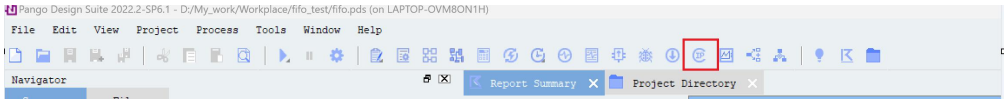


图 3-13.2-1 “IP”图标示意图

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：

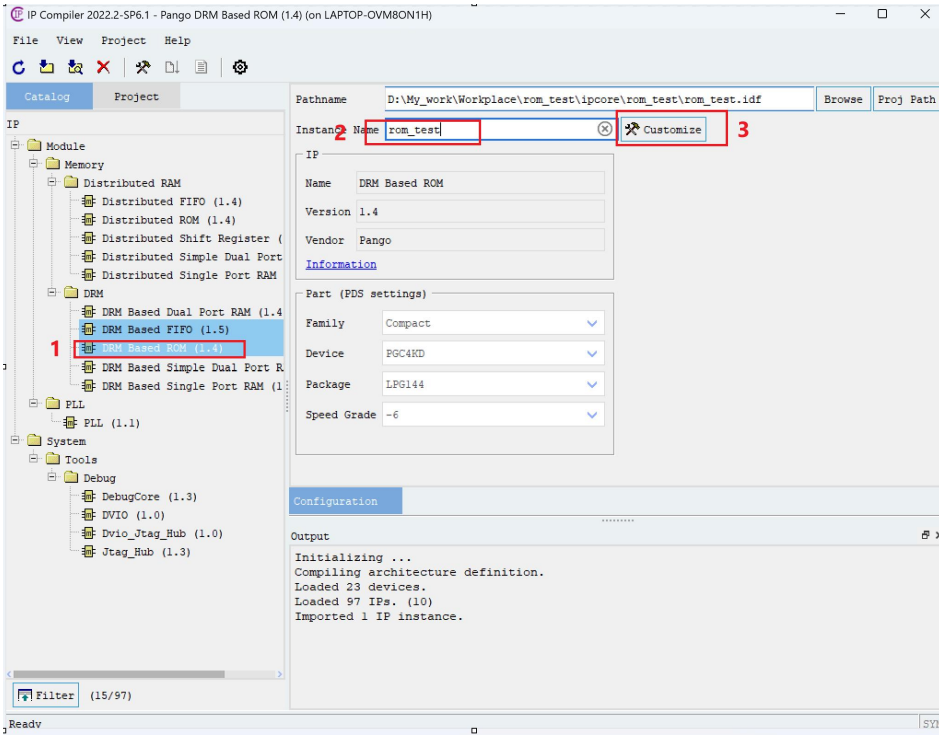


图 3-13.2-2 ROM IP 配置流程图 1

接着进行如下配置：其他选项卡可以保持默认。

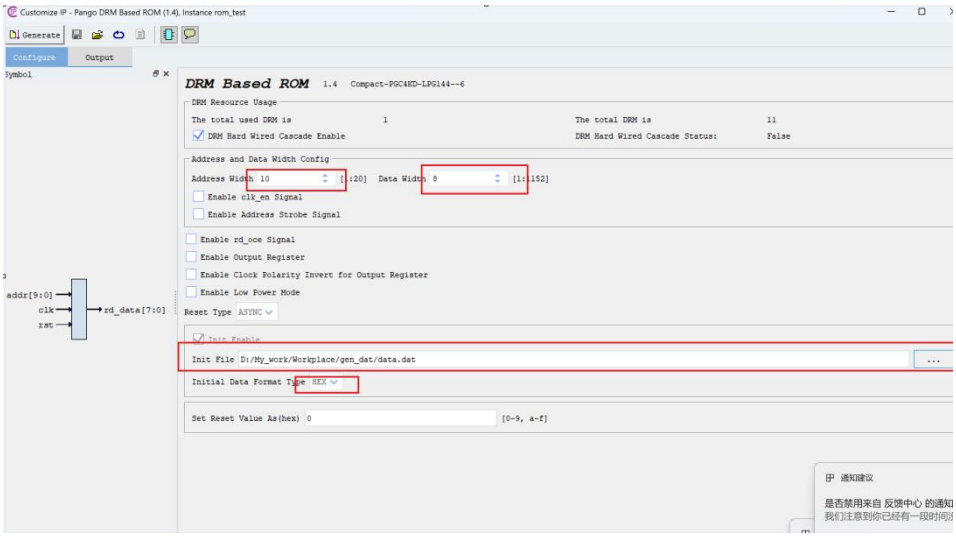


图 3-13.2-3 ROM IP 配置流程图 2

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。

同时，可以看到 Enable Init 选项是默认勾选的，并且不可取消。

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

图所示:

可以看到下图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。

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 3-13.2-4 端口信号示意图

我们可以简单的使用 matlab 生成.dat 文件，然后将生成的 dat 文件在 ip 配置页面指定其位置，注意是 16 进制，其代码如下：

```
% 生成从 0 到 127 的数字
numbers = 0:127;

% 打开一个名为 'data.dat' 的文件以写入
fileID = fopen('data.dat', 'w');

% 遍历数字数组，并将每个数字以十六进制格式写入文件
for i = 1:length(numbers)
    fprintf(fileID, '%02X\n', numbers(i));
end

% 关闭文件
fclose(fileID);

disp('数据已写入到 data.dat 文件中');
```

13.2.3、ROM 的读时序

以下时序图均来自官方 IP 手册，并且均未使能输出寄存。

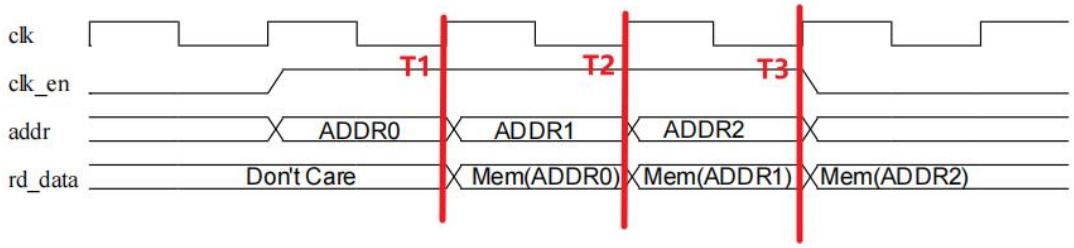


图 3-13.2-5 ROM 的读时序图

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。

13.3、代码设计

模块端口列表如下：

表 3-13-1 ROM IP 实验模块端口表

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	output	8	从 ROM 读出的数据

rom_test_top 顶层模块如下：

```
1.  module rom_test_top
2.  (
3.      input   wire        rd_clk      ,//读时钟
4.      input   wire        rst_n       ,//复位
5.
6.
7.
```

```
8.      input   wire   [9:0]     rd_addr      ,//读地址
9.      output  wire   [7:0]     rd_data      //读数据
10.
11.
12. );
13.
14.
15. rom_test rom_test_inst (
16.   .addr(rd_addr),         // input [9:0]
17.   .clk(rd_clk),           // input
18.   .rst(~rst_n),           // input
19.   .rd_data(rd_data)     // output [7:0]
20. );
21.
22. endmodule
```

该模块的功能是例化 rom ip，这里不多做介绍。

rom_test_tb 测试代码如下：

```
1.  timescale 1ns/1ns
2.  module rom_test_tb();
3.  reg   sys_clk;
4.  reg   rst_n;
5.  reg   [9:0]   rd_addr;
6.  wire  [7:0]   rd_data;
7.
8.  initial
9.  begin
10.    rst_n   <=   1'd0;
11.    sys_clk  <=   1'd0;
12.    #20
13.    rst_n   <=   1'd1;
14.
15. end
16.
17.
```

```
18.  //50MHZ
19.  always#10 sys_clk = ~sys_clk;
20.
21.
22.  //
23.  GTP_GRS GRS_INST(
24.      .GRS_N(1'b1)
25.      ) ;
26.
27.  always@(posedge sys_clk or negedge rst_n)    begin
28.      if(!rst_n)
29.          rd_addr    <=    10'd0;
30.      else    if(rd_addr == 10'd127)
31.          rd_addr    <=    10'd0;
32.      else
33.          rd_addr    <=    #2 rd_addr + 1'b1;
34.  end
35.
36.  rom_test_top u_rom_test_top(
37.      .rd_clk   ( sys_clk  ),
38.      .rst_n    ( rst_n    ),
39.      .rd_addr  ( rd_addr  ),
40.      .rd_data  ( rd_data  )
41.  );
42.
43.  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，到达 127 后重新计数，将 rom 中初始化的数据循环读出。

13.4、实验现象

右键仿真的文件,选择 Run Behavior Simulation 开始行为仿真。启动 Moselsim 进行仿真：

图 3-13.4-1 ROM IP 使用实验结果波形图

dat 文件中储存了 0~127 共 128 个数据，当复位信号为高时，ROM IP 根据当前输入的地址输出数据，数据在地址送入的后一个时钟周期输出。

14、RAM IP 使用实验例程

14.1、实验目的

了解 Compact 系列的 RAM IP 的使用以及配置的方法。

14.2、实验原理

14.2.1、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 3-14-1 RAM 分类表

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，两个端口均可以独领进行读写

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。

14.2.2、IP 配置

首先点击快捷工具栏的“IP”图标，进入 IP 例化设置

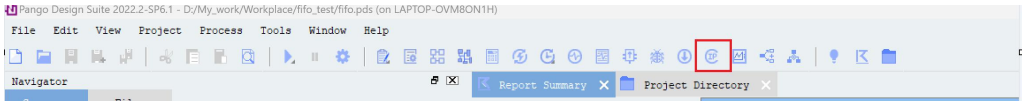


图 3-14.2-1 “IP”图标示意图

然后在 IP 目录处选择 DRM Based Simple Dual Port RAM，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：

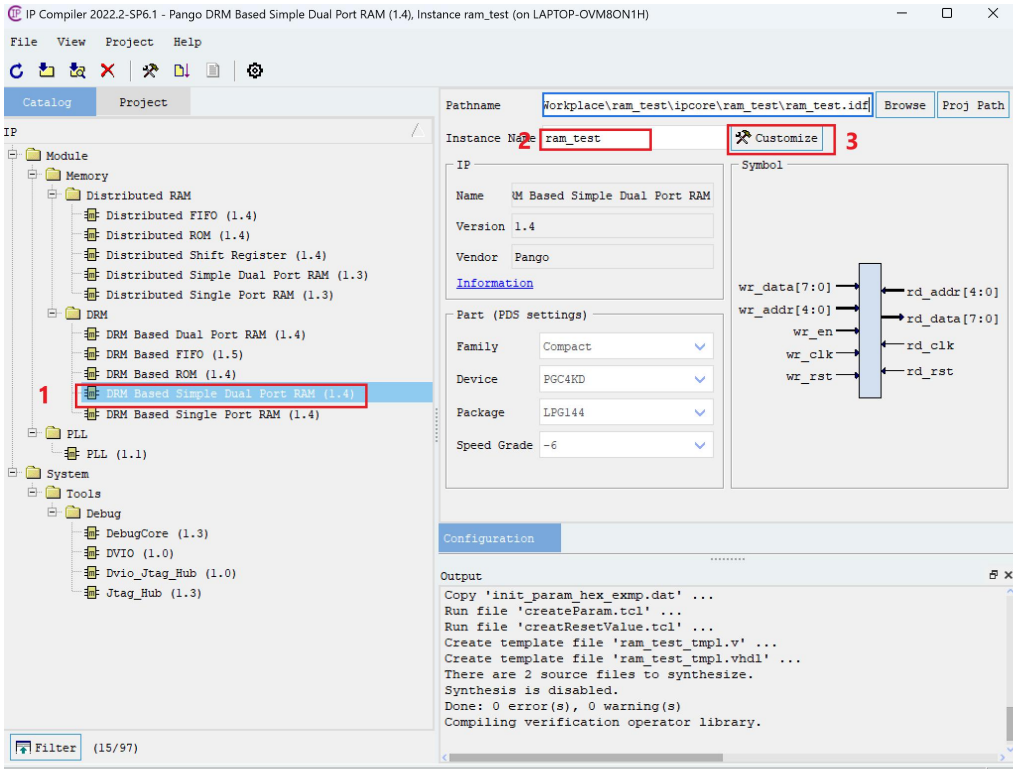


图 3-14.2-2 RAM IP 配置流程图 1

接下来是 ip 配置页面，本实验只需要在这个界面简单的设置读写数据深度和数据位宽，其他设置保持默认即可。

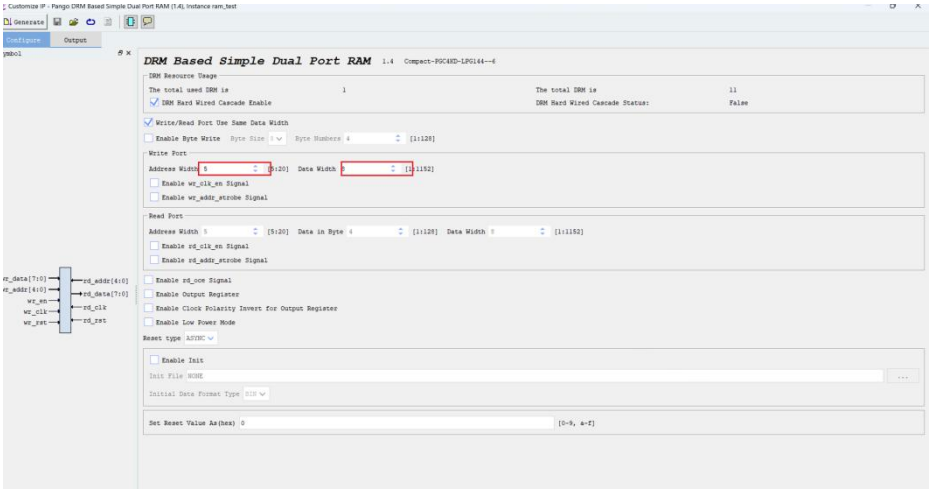


图 3-14.2-3 RAM IP 配置流程图 2

此页面选项卡功能在 ip 手册有详细说明，具体如下图：

配置名称	配置说明
DRM Hard Wired Cascade Enable	DRM 16Kx1 模式下的硬级联使能：当勾选后，会自动根据用户配置情况选择是否使用 DRM 16Kx1 模式下的硬级联
Write/Read Port use same Data Width	配置读写端口是否使用混合位宽模式
Enable Byte Write	配置是否使能 Byte Write 功能 注：当 Byte Write 使能时，不支持 Address Strobe 功能
Byte Size	Byte Write 使能时，配置 Byte 的位宽，可选择 8 或者 9
Byte Numbers	Byte Write 使能时，配置所使用的 Byte 个数
Write Port Address Width	写端口地址位宽，当前 RAM 若无混合位宽读写，此参数合法范围为 9~20；当前 RAM 有混合位宽读写时，A，B 端口所配置的 RAM 容量必须一致
Write Port Data Width	写端口数据位宽，合法范围 1~1152，但是需遵循 RAM 容量必须小于 1152K 的规则
Enable wr_clk_en Signal	配置是否使能写端口的 wr_clk_en 信号
Enable wr_addr_strobe Signal	配置是否使能写端口的 wr_addr_strobe 信号 注：当 Byte Write 使能时，不支持 Address Strobe 功能
Read Port Address Width	读端口地址位宽，当前 RAM 若无混合位宽读写，此参数合法范围为 9~20；当前 RAM 有混合位宽读写时，A，B 端口所配置的 RAM 容量必须一致
Read Port Data Width	读端口数据位宽，合法范围 1~1152，但是需遵循 RAM 容量必须小于 1152K 的规则
Enable rd_clk_en Signal	配置是否使能写端口的 rd_clk_en 信号
Enable rd_addr_strobe Signal	配置是否使能读端口的 rd_addr_strobe 信号 注：当 Byte Write 使能时，不支持 Address Strobe 功能

图 3-14.2-4 IP 手册端口信号说明图 1

配置名称	配置说明
Enable Output Register	配置是否使能读端口输出寄存
Enable rd_oce Signal	配置是否使能 rd_oce 信号；使能 rd_oce 信号时，默认且必须使能读端口输出寄存
Enable Clock Polarity Invert for Output Register	配置是否使能读端口输出时钟极性反向；使能读端口输出时钟极性反向时，默认且必须使能读端口输出寄存
Enable Low Power Mode	配置是否使能低功耗模式
Reset Type	配置复位方式："ASYNC"异步复位，"SYNC"同步复位
Enable Init	配置是否使能对当前 RAM 进行初始化 注：混合位宽时，不支持 RAM 初始化功能
Init File	配置使能对当前 RAM 进行初始化，指定初始化文件路径，若不指定，则生成初始值为全"指定的初始化文件.v 文件
File Type	配置初始化文件数据格式："BIN"二进制，"HEX"十六进制
Reset Value	配置复位时，数据端口的输出值，配置值需为十六进制

图 3-14.2-5 手册端口信号说明图 2

如设计需要用到其他功能可参考用户手册进行配置。

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 3-14.2-6 手册端口信号说明图 3

14.2.3、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。

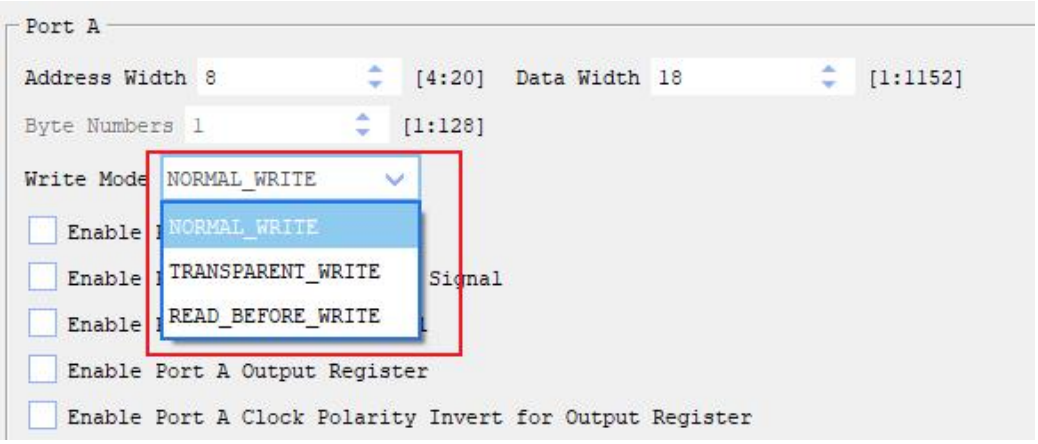


图 3-14.2-7 真双端口模式界面图

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。

(1)NORMAL_WRITE

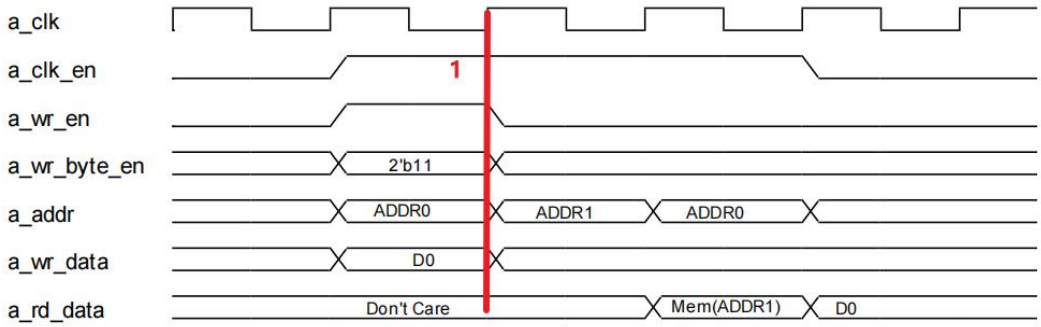


图 3-14.2-8 NORMAL_WRITE 模式时序图

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。

(2)READ_BEFORE_WRITE

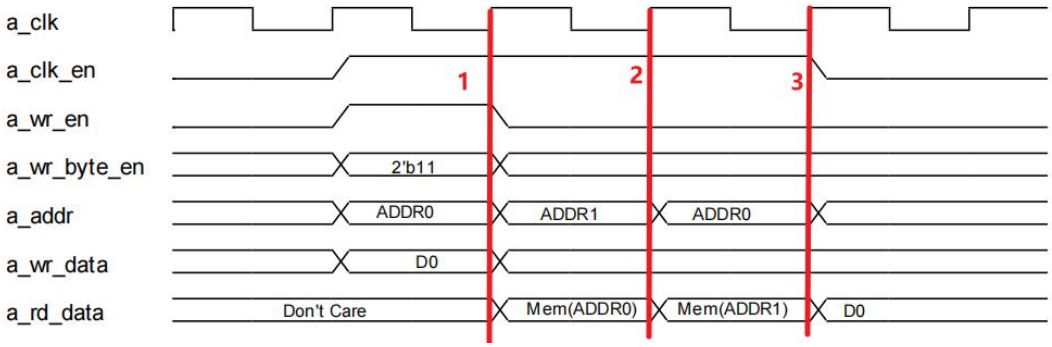


图 3-14.2-9 NORMAL_WRITE 模式时序图

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。

(3)Transparent_Write

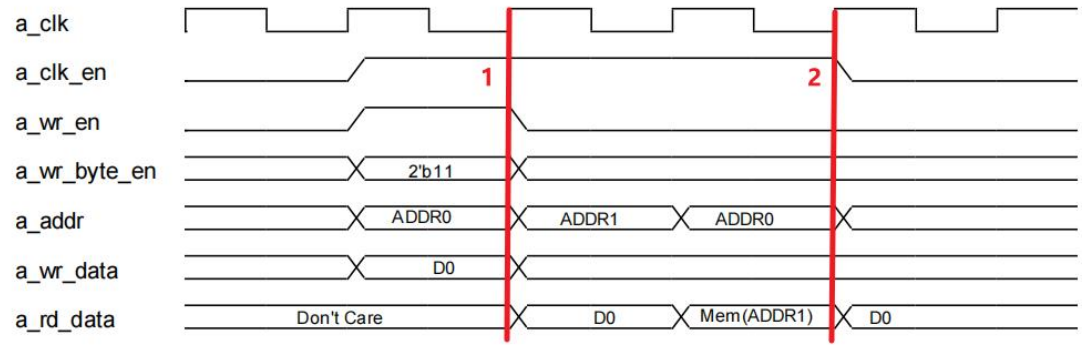


图 3-14.2-10 Transparent_Write 模式时序图

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。

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，我们看图时序来分析：

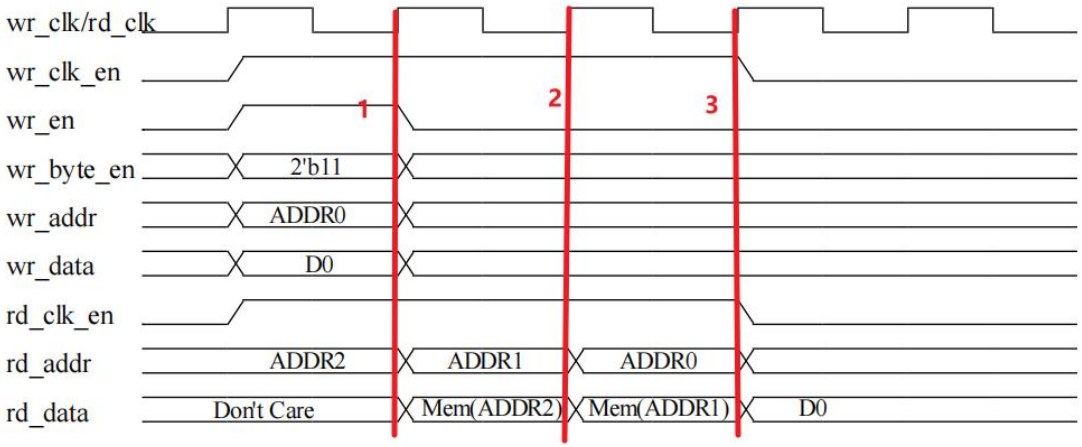


图 3-14.2-11 伪双端口的读写时序图

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。

14.3、代码设计

模块顶层端口列表如下：

表 3-14-2 RAM IP 实验模块顶层端口表

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

模块顶层代码如下：

```
1.  module ram_test_top
2.  (
3.      input   wire            wr_clk      ,//写时钟
4.      input   wire            rd_clk      ,//读时钟
5.      input   wire            rst_n       ,//复位
6.
7.      input   wire            rw_en       ,//读写使能信号
8.      input   wire    [4:0]   wr_addr     ,//写地址
9.      input   wire    [4:0]   rd_addr     ,//读地址
10.     input   wire    [7:0]   wr_data     ,//写数据
11.
12.     output  wire    [7:0]   rd_data     //读数据
13. );
14.
15.
16. ram_test ram_test_inst (
```

```
    .wr_data(wr_data),    // input [7:0]
    .wr_addr(wr_addr),    // input [4:0]
    .wr_en(rw_en),        // input
    .wr_clk(wr_clk),      // input
    .wr_rst(~rst_n),      // input

    .rd_addr(rd_addr),    // input [4:0]
    .rd_data(rd_data),    // output [7:0]
    .rd_clk(rd_clk),      // input
    .rd_rst(~rst_n)       // input
);


endmodule
```

该模块功能是例化 ram ip 并将相关信号在顶层引出。

测试代码如下：

```
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;
```

```
19.     sys_clk  <=   1'd0;
20.     rd_clk   <=   1'd0;
21.     #20
22.     rst_n    <=   1'd1;
23.
24. end
25.
26. //读写控制
27. always@(posedge sys_clk or negedge rst_n)    begin
28.     if(!rst_n)
29.     begin
30.         state    <=   2'd0;
31.         wr_data  <=   8'd0;
32.         rw_en    <=   1'd0;
33.         wr_addr  <=   8'd0;
34.         rd_addr  <=   8'd0;
35.     end
36.     else
37.     begin
38.     case(state)
39.         2'd0:begin
40.                 rw_en    <=   1'd1;
41.                 state    <=   2'd1;
42.             end
43.
44.         2'd1:begin
45.                 if(wr_addr == 5'd31)
46.                 begin
47.                     rw_en    <=   1'd0;
48.                     state    <=   2'd2;
49.                     wr_data  <=   8'd0;
50.                     wr_addr  <=   5'd0;
51.                     rd_addr  <=   5'd0;
52.                 end
53.                 else
54.                 begin
55.                     state    <=   2'd1;
```

```
56.                 wr_data  <=   wr_data+1'b1;
57.                 rd_addr  <=   rd_addr+1'b1;
58.                 wr_addr  <=   wr_addr+1'b1;
59.               end
60.             end
61.         2'd2:begin
62.               if(rd_addr == 5'd31)
63.               begin
64.                 state   <=   2'd3;
65.                 rd_addr  <=   5'd0;
66.               end
67.               else
68.               begin
69.                 state   <=   2'd2;
70.                 rd_addr  <=   rd_addr+1'b1;
71.               end
72.             end
73.         2'd3:begin
74.               state   <=     2'd0;
75.             end
76.
77.         default:   state   <=   2'd0;
78.         endcase
79.     end
80. end
81.
82. //50MHZ
83. always#10 sys_clk = ~sys_clk;
84.
85. //
86. GTP_GRS GRS_INST(
87.     .GRS_N(1'b1)
88.     ) ;
89.
90. ram_test_top u_ram_test_top(
91.     .wr_clk   ( sys_clk   ),
92.     .rd_clk   ( sys_clk   ),
```

```
93.     .rst_n   ( rst_n   ),
94.     .rw_en   ( rw_en   ),
95.     .wr_addr ( wr_addr ),
96.     .rd_addr ( rd_addr ),
97.     .wr_data ( wr_data ),
98.     .rd_data ( rd_data )
99.  );
100. endmodule
```

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，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。

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。

14.4、实验现象

右键仿真的文件,选择 Run Behavior Simulation 开始行为仿真。启动 Modelsim 进行仿真：

我们截取一部分波形来查看 ram 读写时序：

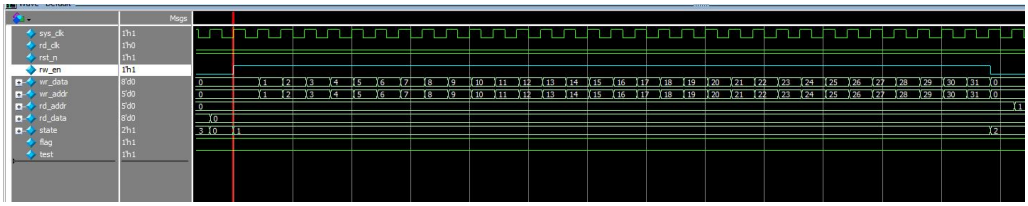


图 3-14.4-1 RAM IP 实验结果波形图 1

当 state 为 0 时，将读写使能 rw_en 拉高，进行数据的写入操作，结合波形我们可以看到 rw_en 拉高同时 state 变为 1 状态，对 ram 写入 0~31 共 32 个数据。

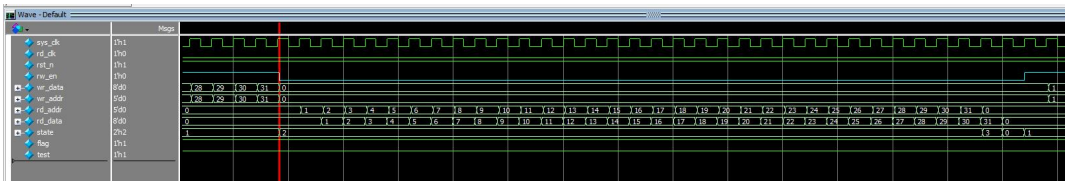


图 3-14.4-2 RAM IP 实验结果波形图 2

当写入第 32 个数据之后 rw_en 拉低，同时 state 跳转到 2 状态，对 ram 中的数据进行读取。当读取完 32 个数据之后 state 跳转到 3 状态再跳转到 0 状态（因为数据是在下一个时钟周期才会读出，所以这里用 state 跳转的形式等待一个时钟周期便于观察），如此往复。

15、舵机控制实验例程

15.1、实验目的

使用逻辑派 Z1 开发板（基于紫光同创 compa 系列 PGC4KD-6ILPG144 芯片）编写 SG90 180 度舵机驱动代码，最终使用开发板按键控制舵机左右旋转和归位。

15.2、实验原理

15.2.1、SG90 180 度舵机介绍

SG90 舵机是一种位置（角度）伺服的驱动器，适用于那些需要角度不断变化并可以保持的控制系统。在机器人机电控制系统中，舵机控制效果是性能的重要影响因素。舵机可以在微机电系统和航模中作为基本的输出执行机构，因为其简单的控制和输出机制使得其在嵌入式系统中应用十分广泛。

本实验使用的舵机模块如下：

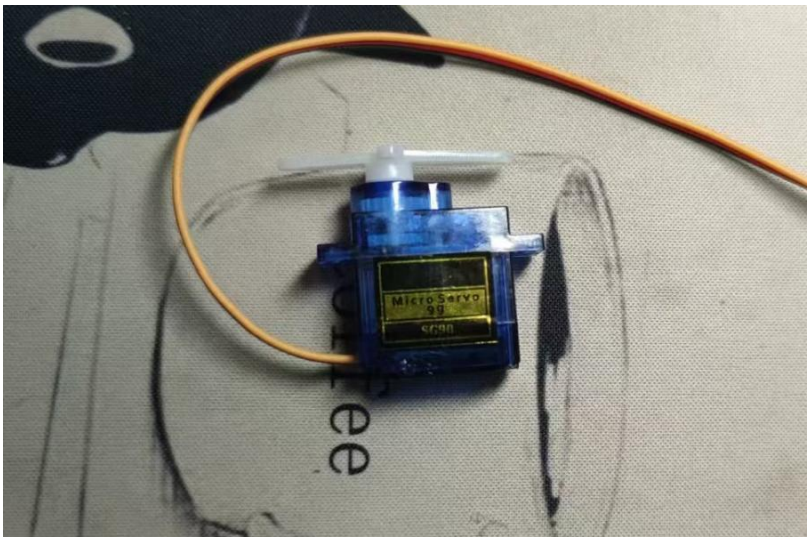


图 3-15.2-1 SG90 舵机图

15.2.2、SG90 舵机工作原理介绍

SG90 舵机是一种小型舵机，具有三条信号线：GND（接地）、VCC（电源正极、5V）和 SIG（控制信号）。它通过接收 SIG 信号线传入的脉宽调制信号（PWM），解析目标角度信息，并利用内部信号调制芯片将信号转化为直流偏置电压。舵机内部的电位器与输出轴机械连接，实时反馈当前角度并产生对应

电压。信号调制芯片通过比较直流偏置电压和电位器反馈电压计算电压差，并将其传递给电机驱动芯片控制电机的转动方向。当电机转动时，电位器同步旋转，逐步减小电压差，直到达到目标角度时停止电机。通过减速齿轮组，电机的高速低扭矩输出被转化为低速高扭矩的精确控制，从而实现舵机的稳定动作和自校正功能。这里只需要大概了解其工作原理即可，我们主要重点关注的是 SG90 舵机的控制原理。

15.2.3、SG90 舵机控制原理

我们在上文中提到，SG90 舵机通过脉宽调制信号（PWM）控制。其需要一个 20ms 的时基脉冲，高电平持续时间在 0.5ms~2.5ms。对于 180 度舵机而言。不同一个 PWM 信号的不同高电平持续时间与舵机旋转角度的对应关系如下：

高电平持续时间/ms	舵机角度/°
0.5	0
1.0	45
1.5	90
2.0	135
2.5	180

图 3-15.2-2 高电平持续时间与舵机旋转角度关系图

我们使用开发板输出高电平持续时间在 0.5ms~2.5ms 之间，周期为 20ms 的 PWM 信号就能够控制舵机任意旋转 0~180 度。

15.3、代码设计

模块端口如下表：

表 3-15-1 舵机控制实验端口表

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
key_in	input	3	key_in0 右转、key_1 左转、key_2 归位

pwm	output	1	舵机控制信号

舵机驱动代码如下：

```
1.  module steering_engine(
2.  input                   clk             ,
3.  input                   rst_n           ,
4.
5.  input           [ 2: 0]     key_in          ,
6.  output reg                  pwm
7.
8.
9.  //参数
10. parameter               TIME_20MS           = 100_0000;
11. parameter               TIME_3S             = 1000;
12. parameter               pwm_0d5ms           = 2_5000;
13. parameter               pwm_1ms             = 5_0000;
14. parameter               pwm_1d5ms           = 7_5000;
15. parameter               pwm_2ms             = 10_0000;
16. parameter               pwm_2d5ms           = 12_5000;
17.
18.
19. //内部信号
20. reg             [ 16: 0]    pwm  level          ;
21. reg             [ 19: 0]    cnt_20ms            ;
22. reg             [ 19: 0]    cnt_3s              ;//这里是三秒钟转 180 度
23.
24.
25. //20ms 计数器
26. always @(posedge clk or negedge rst  n)begin
27.   if(!rst_n)
28.     cnt  20ms <= 0;
29.   else if(cnt_20ms== TIME_20MS 1)
30.         cnt  20ms <= 0;
31.   else
32.         cnt  20ms <= cnt  20ms + 1;
33. end
34. always @(posedge clk or negedge rst  n)begin
35.   if(!rst_n)
36.     cnt  3s <= 0;
37.   else if(cnt_3s== TIME_3S 1)
38.         cnt  3s <= 0;
39.   else
40.         cnt  3s <= cnt  3s + 1;
41. end
42.
43. //按键情况列出
```

```
44.  always  @(posedge clk or negedge rst  n)begin
45.    if(rst_n==1'b0)
46.      pwm_level <= pwm_0d5ms;//默认归位
47.    else if((key_in[0] == 0)&&(cnt_3s== TIME_3S 1)&&(pwm_level<pwm_2d5ms))
48.      pwm_level <= pwm_level + 1'b1;
49.    else if((key_in[1] == 0)&&(cnt_3s== TIME_3S 1)&&(pwm_level>pwm_0d5ms))
50.      pwm_level <= pwm_level 1'b1;
51.    else if(key_in[2] == 0)
52.      pwm_level <= pwm_0d5ms;//归位
53.    else
54.      pwm_level <= pwm_level;
55.  end
56.
57.
58.
59.  //输出 pwm
60.  always  @(posedge clk or negedge rst  n)begin
61.    if(rst_n==1'b0)
62.      pwm <= 0;
63.    else if(cnt_20ms < pwm_level)
64.      pwm <= 1;
65.    else
66.      pwm <= 0;
67.  end
68.
69.
70.  module
```

模块的核心是如何产生受按键控制的可以调整的 pwm 信号，我们使用一个 20ms 计数器 cnt_20ms 用来产生 20ms 的时基脉冲。使用一个 pwm_level 信号决定输出的 pwm 信号高电平持续时间，只有当 cnt_20ms 的值小于 pwm_level，输出的 pwm 信号才为高电平，当 cnt_20ms 的值大于 pwm_level，输出的 pwm 信号为低电平。所以我们只需要通过按键逻辑更改 pwm_level 的值，即可控制输出 pwm 信号的高电平持续时间，pwm_level 默认为 pwm_0d5ms，也就是默认舵机旋转 0 度。

当按键 key0 被按下后，每当 cnt_3s 计数到 1000，并且 pwm_level 的值小于 pwm_2d5ms 时，pwm_level 才会加一，这是为了使得舵机转动得不至于过快和防止高电平持续时间大于 2.5ms 不满足 SG90 舵机的控制要求。

当按键 key1 被按下后，每当 cnt_3s 计数到 1000，并且 pwm_level 的值大于于 pwm_0d5ms 时，pwm_level 才会减一，这是为了使得舵机转动得不至于过快和防止高电平持续时间小于 0.5ms 不满足 SG90 舵机的控制要求。

当按键 key2 被按下后，pwm_level 的值被重新被设置为 pwm_0d5ms，使得舵机旋转到 0 度位置，舵机归位。

约束文件如下：

```
define_attribute {p:key_in[2]} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:key_in[2]} {PAP_IO_LOC} {17}
define_attribute {p:key_in[2]} {PAP_IO_VCCIO} {1.2}
define_attribute {p:key_in[2]} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:key_in[2]} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:key_in[1]} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:key_in[1]} {PAP_IO_LOC} {19}
define_attribute {p:key_in[1]} {PAP_IO_VCCIO} {1.2}
define_attribute {p:key_in[1]} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:key_in[1]} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:key_in[0]} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:key_in[0]} {PAP_IO_LOC} {20}
define_attribute {p:key_in[0]} {PAP_IO_VCCIO} {1.2}
define_attribute {p:key_in[0]} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:key_in[0]} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:pwm} {PAP_IO_DIRECTION} {OUTPUT}
define_attribute {p:pwm} {PAP_IO_LOC} {92}
define_attribute {p:pwm} {PAP_IO_VCCIO} {1.2}
define_attribute {p:pwm} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:pwm} {PAP_IO_DRIVE} {2}
define_attribute {p:pwm} {PAP_IO_NONE} {TRUE}
define_attribute {p:pwm} {PAP_IO_SLEW} {SLOW}
define_attribute {p:clk} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:clk} {PAP_IO_LOC} {5}
define_attribute {p:clk} {PAP_IO_VCCIO} {1.2}
define_attribute {p:clk} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:clk} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:rst_n} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:rst_n} {PAP_IO_LOC} {15}
define_attribute {p:rst_n} {PAP_IO_VCCIO} {1.2}
define_attribute {p:rst_n} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:rst_n} {PAP_IO_PULLUP} {TRUE}
```

15.4、代码仿真

我们对舵机驱动模块进行仿真

仿真代码如下：

```
module  tb();
    reg                 clk             ;
    reg                 rst_n           ;
    reg         [ 2: 0] key_in          ;
```

```
5.      wire                            pwm                 ;
6.
7.
8.      defparam u_steering_engine.TIME_20MS            = 400,
9.               u_steering_engine.TIME_3S              = 30  ,
10.              u_steering_engine.pwm_0d5ms            = 2_5 ,
11.              u_steering_engine.pwm_1ms              = 5_0 ,
12.              u_steering_engine.pwm_1d5ms            = 7_5 ,
13.              u_steering_engine.pwm_2ms              = 10_0,
14.              u_steering_engine.pwm_2d5ms            = 12_5;
15.     parameter   CLK_FREQ = 50;//Mhz
16.
17.
18.     initial
19.       begin
20.           #2
21.               rst_n = 0   ;
22.               clk = 0     ;
23.           #10
24.               rst_n = 1   ;
25.       end
26.
27.
28.     always # ( 1000/CLK_FREQ/2 ) clk = ~clk ;
29.
30.     GTP_GRS GRS_INST(
31.     .GRS_N                  (1'b1               )
32.     );
33.
34.
35.
36.   steering_engine u_steering_engine(
37.     .clk                    (clk                ),
38.     .rst_n                  (rst_n              ),
39.     .key_in                 (3'b110             ),
40.     .pwm                    (pwm                )
41.   );
42.
43.
44.   endmodule
```

舵机驱动模块输入输出逻辑较为简单，我们只需要模拟参数时钟与复位即可，为了便于观察，我们使用 defparam 语句对舵机驱动模块内的参数进行重定义，我们这里是模拟按下按键 k0 使得舵机向右旋转，期望输出的 pwm 信号应该是占空比逐渐增大的。Modelsim 与 PDS 进行联合仿真的操作在第二部分开发环境搭建章节已有介绍，我们这里启动 Modelsim 进行仿真，查看 pwm 信号是否如预期般占空比逐渐增

大。

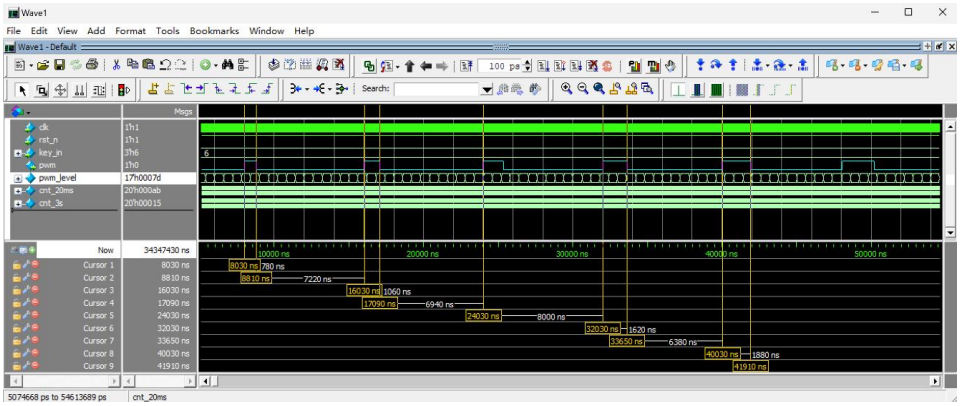


图 3- 15.4- 1 舵机控制实验仿真波形图

如上图所示，当按下按键 k0 之后（key_in 位宽为 3，按键按下后为低电平，未按下为高电平，按下 k0 后变为 3’b110,也就是十进制的 6），可以发现输出的 pwm 信号高电平部分在逐渐增加，由于我们在 tb 代码中更改了参数，所以这里的 pwm 信号周期、高电平持续时间并不满足舵机控制的要求，只是为了直观的看出占空比的变换，对 pwm 信号进行了调整。在实际舵机控制过程中，为了舵机运行得更加平稳，不出现抽搐等现象，pwm 波形的变换过程会更加平缓。

同理如果按下 k1，舵机将会向左旋转，对应的 pwm 占空比应该会逐渐减小，按下 k2，舵机将会归位，回到 0 度位置，读者可以在 tb 文件中对输入按键信号进行修改，观察 pwm 信号的变换情况。

15.5、实验现象

当按下开发板按键 k0 时，舵机向右旋转，当按下 k1 时，舵机向左旋转，当按下 k2 时舵机归位。

16、超声波测距实验例程

16.1、实验目的

使用逻辑派 Z1 开发板（基于紫光同创 compa 系列 PGC4KD-6ILPG144 芯片）编写 HC_SR04 超声波测距模块驱动代码，最终使用开发板按键触发超声波测距模块的测距操作。并将测距得到的距离使用串口发送到上位机。

16.2、实验原理

16.2.1、超声波测距原理

超声波测距是一种利用超声波在空气中的传播时间来测量距离的方法。只要知道超声波的传播速度以及发射超声波与接收到反射声波的时间就能计算出物体的距离。其具体操作过程如下（不考虑温度、压强对声波传输的影响）：

1.发射超声波

超声波测距模块首先通过一个超声波换能器（通常是压电陶瓷或压电薄膜）发射出一束超声波脉冲。这些脉冲在空气中以一定的速度传播。

2.接收反射波

当超声波脉冲被目标物体反射后，返回的反射波被同一个或另一个超声波换能器接收。测距模块记录从发射到接收反射波的时间。

3.计算距离

根据超声波的传播时间和传播速度，只需要将（传播时间×声速）/2 就可以得到目标物体与测距模块之间的距离。这里的传播时间是从发射到接收到反射波的总时间，除以 2 是因为超声波往返的总时间包括去程和返程。

超声波测距模块图如下：

图 3- 16.2- 1 超声波测距模块图

16.2.2、HC_SR04 超声波测距模块工作原理

HC-SR04 超声波测距模块是一种经济实用的非接触式距离测量设备，广泛用于各种需要精确测距的应用，如机器人避障、汽车倒车雷达等。该模块包含一个超声波发射器和接收器，通过四个引脚（VCC、Trig、Echo、GND）与微控制器连接。工作时，需要微控制器向 Trig 引脚发送一个 10 微秒的高电平脉冲，触发模块发射超声波脉冲。模块接收到反射波后，Echo 引脚输出一个与超声波往返时间成正比的高电平脉冲，微控制器需要通过测量 Echo 引脚高电平脉冲的持续时间，利用距离=（传播时间×声速）/2 计算出目标距离。

其具体控制时序如下：

图 3-16.2-2 控制时序图

我们使用逻辑派 Z1 开发板作为微控制器，首先我们向 HC-SR04 模块的 trig 引脚发送一个持续 10us 的触发信号，HC-SR04 收到我们给出的触发信号之后，会使用超声波探头发射 8 个 40khz 的脉冲。当测距结束后，HC-SR04 会通过 echo 引脚发送一个高电平信号，高电平持续时间就是声波往返物体经历的时间。我们利用这段时间就可以计算出物体到超声波模块的距离。

16.3、代码设计

超声波驱动模块端口如下：

表 3-16-1 超声波驱动模块端口表

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
cap_flag	input	1	触发采集信号（开始一次测距）

distance	output	8	计算得到的距离（单位厘米）
data_vld	output	1	distance 的有效标志
data_vld_d	output	1	个、十、百、千位的数据有效标志
s_g	output	4	distance 的个位
s_s	output	4	distance 的十位
s_b	output	4	distance 的百位
s_q	output	4	distance 的千位
echo	input	1	超声波模块输入
trig	output	1	输出给超声波模块的触发信号

超声波驱动代码如下：

```
1.  module HC_SR04_drive(
2.      input                   clk             ,
3.      input                   rst_n           ,
4.      input                   cap_flag        ,
5.
6.      output reg      [ 7: 0]     distance        ,
7.      output reg                  data_vld        ,
8.      output reg                  data_vld_d      ,
9.      output reg      [ 3: 0]     s_g             ,//个位
10.     output reg      [ 3: 0]     s_s             ,//十位
11.     output reg      [ 3: 0]     s_b             ,//百位
12.     output reg      [ 3: 0]     s_q             ,//千位
13.     input                   echo            ,
14.     output                  trig
15.
16.
17.     );
18.
19.
20.     parameter               MAX_CNT         = 50    ; // 计数到 100 时产生 1μs
脉冲
21.
22.     reg         [ 9: 0]     cnt_trig        ;//12us 计数器
23.     reg                     cnt_en          ;
24.
25.     reg                     echo_syn1       ;
26.     reg                     echo_syn2       ;
27.     reg                     echo_syn3       ;
28.
```

```
29.    reg             [  6: 0]      cnt_1us                ;
30.    reg             [ 12: 0]       cnt  us  num            ;
31.
32.    wire                         echo  fall              ;
33.    assign                        echo_fall              = ~echo_syn2 & echo_syn3;//下降沿
34.
35.
36.    assign                        trig                   = cnt_en;
37.
38.  //时钟同步以及打拍
39.    always @(posedge clk or negedge rst_n)begin
40.      if(!rst_n)begin
41.        echo_syn1 <= 0;
42.        echo_syn2 <= 0;
43.      end
44.      else begin
45.        echo_syn1 <= echo;
46.        echo_syn2 <= echo_syn1;
47.        echo_syn3 <= echo_syn2;
48.      end
49.    end
50.
51.    always @(posedge clk or negedge rst_n)
52.      if(!rst_n)
53.        cnt_en <= 1'b0;
54.      else if(cap  flag)
55.        cnt_en <= 1'b1;
56.      else if(cnt  trig == 10'd500 -1)
57.        cnt_en <= 1'b0;
58.
59.    always @(posedge clk or negedge rst_n)
60.      if(!rst  n)
61.        cnt_trig <= 10'b0;
62.      else if(cnt  trig == 10'd500 -1)
63.        cnt_trig <= 10'b0;
64.      else if(cnt  en)
65.        cnt_trig <= cnt_trig + 1'b1;
66.
67.
68.    always @(posedge clk or negedge rst  n)
69.      if(!rst_n)
70.        cnt  1us <= 7'b0;
71.      else if((cnt_1us==MAX_CNT-1)||(cap_flag))
72.          cnt  1us <= 7'b0;
73.      else if(echo_syn3)
74.          cnt_1us <= cnt_1us + 1'b1;
75.
76.    always @(posedge clk or negedge rst_n)
77.      if(!rst_n)
```

```
78.          cnt_us_num <= 13'b0;
79.        else if((cap_flag))
80.            cnt_us_num <= 13'b0;
81.        else if((echo_syn3)&&(cnt_1us==MAX_CNT-1))
82.            cnt_us_num <= cnt_us_num + 1'b1;
83.
84.     always @(posedge clk or negedge rst_n)
85.        if(!rst_n)begin
86.          distance<= 8'b0;
87.          data_vld<= 1'b0;
88.        end
89.        else if(echo_fall)begin
90.          distance <= (cnt_us_num * 17)/1000;
91.          data_vld <= 1'b1;
92.        end
93.        else
94.          data_vld<= 1'b0;
95.
96.   always @(posedge clk or negedge rst_n) begin
97.     if (rst_n == 1'b0) begin
98.        s_g <= 0;
99.        s_s <= 0;
100.       s_b <= 0;
101.       s_q <= 0;
102.       data_vld_d<= 1'b0;
103.    end else begin
104.       s_g <= distance % 10;
105.       s_s <= (distance / 10) % 10;
106.       s_b <= (distance / 100) % 10;
107.       s_q <= (distance / 1000) % 10;
108.       data_vld_d<= data_vld;
109.    end
110. end
111.
112.
113.    endmodule
```

这个模块的功能是通过控制 HC-SR04 超声波测距模块的触发信号和接收反射信号，测量目标物体的距离并将结果输出。模块根据触发标志信号 cap_flag 启动测距过程。在代码 62 行，当 cap_flag 有效时，将 cnt_en 拉高，开始 10us 的触发计数器 cnt_trig 的计数当计数到 499 时（模块时钟是 50Mhz，一个时钟周期是 20ns，10us 的时间就是 500 个时钟周期，由于计数器是从 0 开始计数，所以计数最大值是 499），清零并拉低 cnt_en。这个 cnt_en 信号持续了 10us，将其作为超声波测距模块的触发信号输出。

由于超声波模块输出的信号时钟与我们开发板的时钟不同步，我们声明 echo_syn1、echo_syn2、echo_syn3 使用多级寄存器打拍的方式对 echo 信号进行时钟同步。当 echo_syn3 有效时，启动微秒计数器 cnt_1us 对 echo 高电平持续时间进行计数。同时使用 cnt_us_num 对高电平持续时间有几微秒进行记录。当捕获到 echo 的下降沿之后，结合 cnt_us_num 的值对距离进行计算。通过以下这行可以实现：

distance <= (cnt_us_num * 17)/1000;

其实是 distance <= (cnt_us_num * 340)/2/1000;模块中对其进行了适当的简化，除以 1000 是将单位转换为厘米。

最终，模块将距离值分解为个位、十位、百位和千位，通过输出寄存器输出。

模块顶层代码端口如下：

表 3-16-2 超声波测距实验模块顶层端口表

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
key_in	input	1	触发采集信号（开始一次测距）
echo	input	1	超声波模块输入
trig	output	1	输出给超声波模块的触发信号
uart_txd	output	1	串口发送

模块顶层代码如下：

```
1.  module HC_SR04_top(
2.    input                 clk             ,
3.    input                 rst_n           ,
4.
5.
6.    input                 key_in          ,
7.    input                 echo            ,
8.    output                trig            ,
9.    output                uart_txd
10. );
11.
12.
13.   parameter               CNT_MAX           = 20'd999_999; //消抖计数器
```

```
    parameter                STR              = 192'hb3_ac_c9_f9_b2_a8_c4_a3_bf_e9_b2_e2_be_e0_d6_b5_ce_aa_a3_ba_c0_e5_c3_d7;//"超声波模块测距值为：厘米"的GB2312 编码


//信号定义
    wire                        data_vld_d          ;
    wire            [ 31: 0]     temperature_data        ;
    wire                        uart_tx_busy            ;
    wire                        uart_tx_done          ;

    reg             [  7: 0]     uart_tx_data            ;
    reg                         uart_tx_req           ;
    reg                         work_en               ;
    reg             [  6: 0]     tx_byte_cnt           ;

    reg             [ 19: 0]     cnt_20ms               ;//消抖计数器
    reg                         key_flag              ;
    reg             [  7: 0]     ascii_table       [0:9]  ;//储存 0~9 的 ASCII 值

    wire            [  3: 0]     s_g                   ;//个位
    wire            [  3: 0]     s_s                   ;//十位
    wire            [  3: 0]     s_b                   ;//百位
    wire            [  3: 0]     s_q                   ;//千位;

//cnt_20ms:如果时钟的上升沿检测到外部按键输入的值为低电平时，计数器开始计数
always@(posedge clk or negedge rst_n)
    if(rst_n == 1'b0)
        cnt_20ms <= 20'b0;
    else    if(key_in == 1'b1)
        cnt_20ms <= 20'b0;
    else    if(cnt_20ms == CNT_MAX && key_in == 1'b0)
        cnt_20ms <= cnt_20ms;
    else
        cnt_20ms <= cnt_20ms + 1'b1;


always@(posedge clk or negedge rst_n)
    if(rst_n == 1'b0)
        key_flag <= 1'b0;
    else    if(cnt_20ms == CNT_MAX 1'b1)
        key_flag <= 1'b1;
    else
        key_flag <= 1'b0;


always@(posedge clk or negedge rst_n)
    if(rst_n == 1'b0)
```

```
60.       work_en <= 1'b0;
61.     else    if(data_vld_d == 1'b1)
62.       work_en <= 1'b1;
63.     else    if(tx_byte_cnt== 7'd28&&uart_tx_done)
64.       work_en <= 1'b0;
65.     else
66.       work_en <= work_en;
67.
68.  //串口发送逻辑
69.
70.  always @(posedge clk or negedge rst_n) begin
71.    if (!rst_n) begin
72.      tx_byte_cnt <= 7'b0;
73.    end else if (work_en) begin
74.      if (tx_byte_cnt == 7'd28&&uart_tx_done)                  // 总共发送 29 字节
75.          tx_byte_cnt <= 7'b0;
76.      else if(uart_tx_done)
77.          tx_byte_cnt <= tx_byte_cnt + 1'b1;
78.    end
79.  end
80.
81.
82.  // 串口发送数据控制
83.  always @(posedge clk or negedge rst_n) begin
84.    if (!rst_n) begin
85.      uart_tx_req <= 1'b0;
86.      uart_tx_data <= 8'b0;
87.      ascii_table[0] = 8'd48;                          //'0'
88.      ascii_table[1] = 8'd49;                          //'1'
89.      ascii_table[2] = 8'd50;                          //'2'
90.      ascii_table[3] = 8'd51;                          //'3'
91.      ascii_table[4] = 8'd52;                          //'4'
92.      ascii_table[5] = 8'd53;                          //'5'
93.      ascii_table[6] = 8'd54;                          //'6'
94.      ascii_table[7] = 8'd55;                          //'7'
95.      ascii_table[8] = 8'd56;                          //'8'
96.      ascii_table[9] = 8'd57;                          //'9'
97.
98.    end else if (work_en && !uart_tx_busy) begin
99.      case (tx_byte_cnt)                          // 根据当前字节计数器发送数据
100.         7'd0:  uart_tx_data <= STR[191:184];          // "超"
101.         7'd1:  uart_tx_data <= STR[183:176];          // "超"
102.         7'd2:  uart_tx_data <= STR[175:168];          // "声"
103.         7'd3:  uart_tx_data <= STR[167:160];          // "声"
104.         7'd4:  uart_tx_data <= STR[159:152];          // "波"
105.         7'd5:  uart_tx_data <= STR[151:144];          // "波"
106.         7'd6:  uart_tx_data <= STR[143:136];          // "模"
107.         7'd7:  uart_tx_data <= STR[135:128];          // "模"
108.         7'd8:  uart_tx_data <= STR[127:120];          // "块"
```

```
109.            7'd9:  uart_tx_data <= STR[119:112];              // "块"
110.            7'd10: uart_tx_data <= STR[111:104];              // "测"
111.            7'd11: uart_tx_data <= STR[103:96];               // "测"
112.            7'd12: uart_tx_data <= STR[95:88];                // "距"
113.            7'd13: uart_tx_data <= STR[87:80];                // "距"
114.            7'd14: uart_tx_data <= STR[79:72];                // "值"
115.            7'd15: uart_tx_data <= STR[71:64];                // "值"
116.            7'd16: uart_tx_data <= STR[63:56];                // "为"
117.            7'd17: uart_tx_data <= STR[55:48];                // "为"
118.            7'd18: uart_tx_data <= STR[47:40];                // "："
119.            7'd19: uart_tx_data <= STR[39:32];                // "："
120.            7'd20: uart_tx_data <= ascii_table[s_q];          // 测量距离的个位
121.            7'd21: uart_tx_data <= ascii_table[s_b];          // 测量距离的十位
122.            7'd22: uart_tx_data <= ascii_table[s_s];          // 测量距离的百位
123.            7'd23: uart_tx_data <= ascii_table[s_g];          // 测量距离的千位
124.            7'd24: uart_tx_data <= STR[31:24];                // "厘"
125.            7'd25: uart_tx_data <= STR[23:16];                // "厘"
126.            7'd26: uart_tx_data <= STR[15:8];                 // "米"
127.            7'd27: uart_tx_data <= STR[7:0];                  // "米"
128.            7'd28: uart_tx_data <= 8'h0A;                     // 换行符
129.            default: uart_tx_data <= 8'b0;
130.        endcase
131.        uart_tx_req <= 1'b1;                                  // 拉高请求信号
132.    end else begin
133.        uart_tx_req <= 1'b0;                                  // 请求信号拉高仅一个时钟周期
134.    end
135. end
136.
137.
138.
139. HC_SR04_drive u_HC_SR04_drive(
140.    .clk                    (clk                ),
141.    .rst_n                  (rst_n              ),
142.    .cap_flag               (key_flag           ),
143.    .distance               (                   ),
144.    .data_vld               (                   ),
145.    .data_vld_d             (data_vld_d         ),
146.    .s_g                    (s_g                ),// 个位
147.    .s_s                    (s_s                ),// 十位
148.    .s_b                    (s_b                ),// 百位
149.    .s_q                    (s_q                ),// 千位
150.    .echo                   (echo               ),
151.    .trig                   (trig               )
152. );
153.
154.
155.
156. uart_tx u_uart_tx(
157.    .clk                    (clk                ),
```

```
158.    .rst_n                  (rst_n              ),
159.    .uart_tx_req            (uart_tx_req&work_en    ),// 发送使能请求
160.    .uart_tx_data           (uart_tx_data       ),// UART 要发送的数据
161.    .uart_txd               (uart_txd           ),// UART 发送端口
162.    .uart_tx_done           (uart_tx_done       ),//8bit 数据发送完成
163.    .uart_tx_busy           (uart_tx_busy       ) // 发送忙
164. );
165.
166. endmodule
```

我们在顶层模块将超声波测距得到的数据通过串口发送到上位机进行查看，汉字使用 GB2312 编码，模块中初始化了一个 ascii_table 用来储存阿拉伯数字 0~9 的 ASCII 值，将超声波测距得到的数据分为个位、十位、百位、千位在 ascii_table 索引到对应的 ASCII 值进行发送。其中串口发送部分的知识在之前的章节已有介绍，读者若对这部分不熟悉，可到串口回环章节进行查看。

约束代码如下：

```
1.  define_attribute {p:trig} {PAP_IO_DIRECTION} {OUTPUT}
2.  define_attribute {p:trig} {PAP_IO_LOC} {92}
3.  define_attribute {p:trig} {PAP_IO_VCCIO} {1.2}
4.  define_attribute {p:trig} {PAP_IO_STANDARD} {LVCMOS12}
5.  define_attribute {p:trig} {PAP_IO_DRIVE} {2}
6.  define_attribute {p:trig} {PAP_IO_NONE} {TRUE}
7.  define_attribute {p:trig} {PAP_IO_SLEW} {SLOW}
8.  define_attribute {p:uart_txd} {PAP_IO_DIRECTION} {OUTPUT}
9.  define_attribute {p:uart_txd} {PAP_IO_LOC} {33}
10. define_attribute {p:uart_txd} {PAP_IO_VCCIO} {1.2}
11. define_attribute {p:uart_txd} {PAP_IO_STANDARD} {LVCMOS12}
12. define_attribute {p:uart_txd} {PAP_IO_DRIVE} {2}
13. define_attribute {p:uart_txd} {PAP_IO_NONE} {TRUE}
14. define_attribute {p:uart_txd} {PAP_IO_SLEW} {SLOW}
15. define_attribute {p:clk} {PAP_IO_DIRECTION} {INPUT}
16. define_attribute {p:clk} {PAP_IO_LOC} {5}
17. define_attribute {p:clk} {PAP_IO_VCCIO} {1.2}
18. define_attribute {p:clk} {PAP_IO_STANDARD} {LVCMOS12}
19. define_attribute {p:clk} {PAP_IO_PULLUP} {TRUE}
20. define_attribute {p:echo} {PAP_IO_DIRECTION} {INPUT}
21. define_attribute {p:echo} {PAP_IO_LOC} {91}
22. define_attribute {p:echo} {PAP_IO_VCCIO} {1.2}
23. define_attribute {p:echo} {PAP_IO_STANDARD} {LVCMOS12}
24. define_attribute {p:echo} {PAP_IO_PULLUP} {TRUE}
25. define_attribute {p:key_in} {PAP_IO_DIRECTION} {INPUT}
26. define_attribute {p:key_in} {PAP_IO_LOC} {20}
27. define_attribute {p:key_in} {PAP_IO_VCCIO} {1.2}
28. define_attribute {p:key_in} {PAP_IO_STANDARD} {LVCMOS12}
```

```
29.  define_attribute {p:key_in} {PAP_IO_PULLUP} {TRUE}
30.  define_attribute {p:rst_n} {PAP_IO_DIRECTION} {INPUT}
31.  define_attribute {p:rst_n} {PAP_IO_LOC} {19}
32.  define_attribute {p:rst_n} {PAP_IO_VCCIO} {1.2}
33.  define_attribute {p:rst_n} {PAP_IO_STANDARD} {LVCMOS12}
34.  define_attribute {p:rst_n} {PAP_IO_PULLUP} {TRUE}
```

16.4、代码仿真

由于本次实验涉及到与 HC_SR04 超声波测距模块的交互，编写测试代码模拟 HC_SR04 超声波测距模块的答复比较麻烦，而且不能贴近真实的模块真实的测距情景。所以我们可以使用 PDS 在线 debug 的功能对 HC_SR04 驱动模块进行验证。

我们在 HC_SR04 驱动模块代码中对关键信号打上标记（/* synthesis PAP_MARK_DEBUG="true" */），使用在线 debug 工具对模块进行分析。

我们将采样触发信号 cap_flag、计算得到的距离 distance、输出的数据有效信号 data_vld、超声波测距模块输入的回响信号 echo、输入给超声波测距模块的触发信号 trig、回响信号的微秒计数器 cnt_us_num 打上标记，进行分析。

我们将触发模式设置为 cap_flag 信号高电平触发模式，按下按键，cap_flag 被拉高（下图中 cap_flag 的值为 1，只是被标志线遮挡住了），同时触发信号 trig 正确的拉高了。说明已经向超声波测距模块发送了测距触发信号。

图 3-16.4-1 超声波测距实验仿真波形图 1

接下来在超声波测距模块 30cm 处放置一物体（需要大一些），将触发模式设置为 data_vld 信号高电平触发，按下开发板按键 k0，发现计算出的距离值为 30，微秒计数器的值为 1769，符合我们的预期。如下图所示：

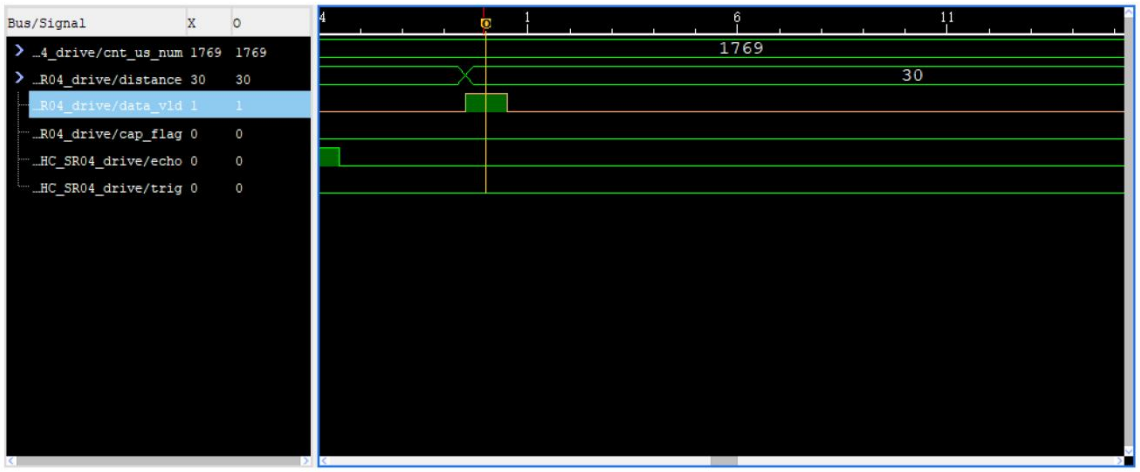


图 3-16.4-2 超声波测距实验仿真波形图 2

16.5、实验现象

将板卡与超声波模块根据约束文件正确连接后，将程序下载进入板卡，按下板卡按键 k0，将会触发一次测距，并将得到的结果通过串口打印到上位机

现象如下：

在 30cm 处放置一纸箱，按下板卡 key0 之后上位机接收如下：

图 3-16.5-1 超声波测距实验结果图 1

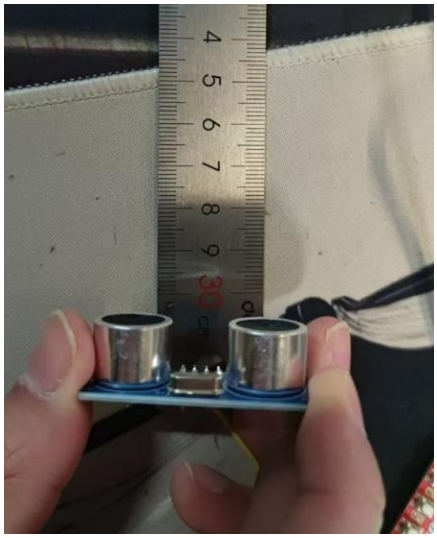


图 3-16.5-2 超声波测距实验结果图 2

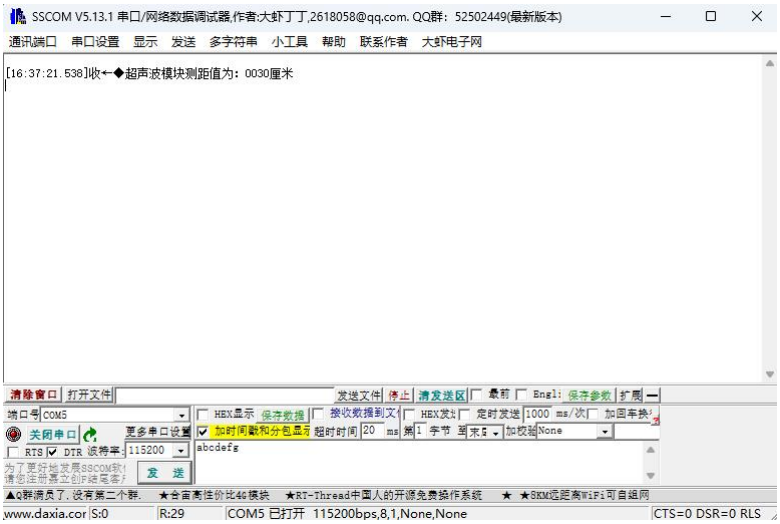


图 3-16.5-3 超声波测距实验结果图 3

17、反射式光传感器实验例程

17.1、实验目的

使用逻辑派 Z1 开发板（基于紫光同创 compa 系列 PGC4KD-6ILPG144 芯片）编写 TCRT5000 反射式光传感器模块验证代码，当传感器检测到物体接近时，板卡通过串口向上位机发送了“检测到物体接近”当物体离开时，板卡通过串口向上位机发送“检测到物体离开”。

17.2、实验原理

17.2.1、TCRT5000 反射式光传感器工作原理

TCRT5000 反射式光传感器是一种集成的光电传感器，内部包含一个红外发光二极管（IR LED）和一个光电晶体管（Phototransistor）。工作时，IR LED 发射波长为 940 纳米的红外光，当这些红外光照射到目标物体并被反射回来时，光电晶体管会接收到反射光并根据其强度改变输出电压。通过检测输出电压的变化，可以判断目标物体的存在与否及其距离。这种传感器常用于机器人避障、接近检测和物体检测等近距离应用，具有结构简单、成本低廉和响应快速的特点。模块图如下：

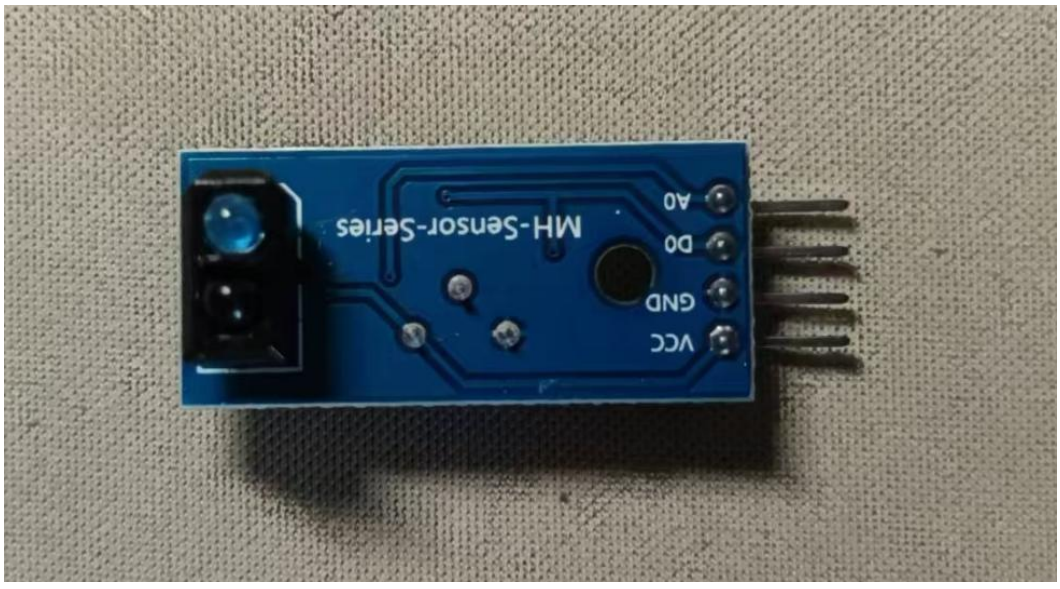


图 3-17.2-1 TCRT5000 反射式光传感器模块图 1

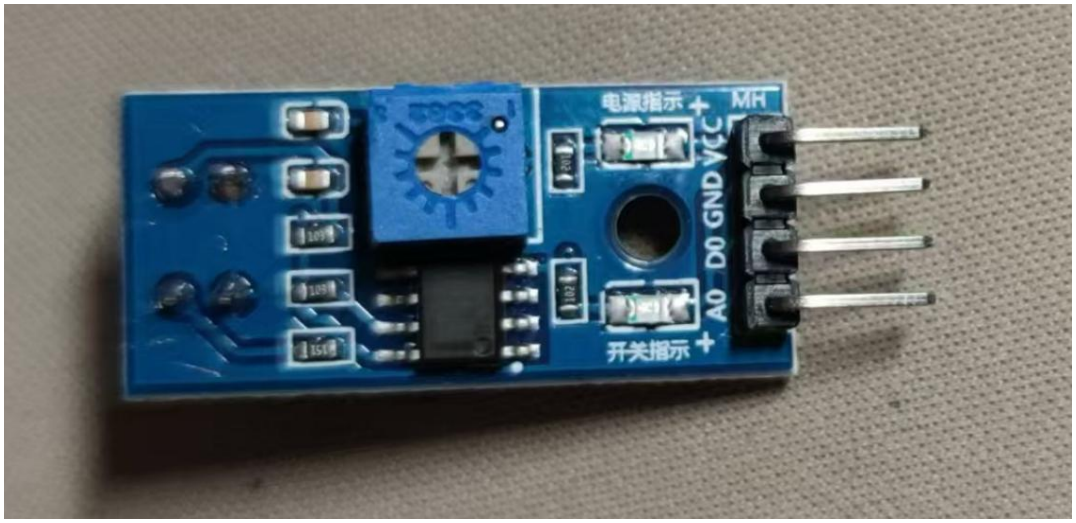


图 3-17.2-2 TCRT5000 反射式光传感器模块图 2

模块具有 4 个管脚，分别是 VCC、GND、A0、D0。其中 A0 是模拟信号输出，D0 是 TTL 电平输出。我们将 D0 接入开发板，当模块探测到物体时，D0 会输出 0，其他时候 D0 为高电平，我们根据这个原理设计代码。

17.3、代码设计

模块端口如下表：

表 3-17-1 反射式光传感器实验模块端口

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
D0	input	1	输出给超声波模块的触发信号
uart_txd	output	1	串口发送

模块代码如下：

```
1.  module TCRT5000(
2.    input                clk          ,
3.    input                rst_n        ,
4.
5.    input                D0           ,
6.    output               uart_txd
7.  );
8.
9.
```

```
10.     parameter                   CNT_MAX             = 20'd999_999; //消抖计数器
11.     parameter STR_approach = 112'hbc_ec_b2_e2_b5_bd_ce_ef_cc_e5_bd_d3_bd_fc; // "检测到物体接近" 的 GB2312 编码
12.     parameter STR_leave = 112'hbc_ec_b2_e2_b5_bd_ce_ef_cc_e5_c0_eb_bf_aa; // "检测到物体离开" 的 GB2312 编码
13.
14.
15.
16.   //信号定义
17.
18.     wire                    uart_tx_busy        ;
19.     wire                    uart_tx_done        ;
20.     wire start_approach;
21.     wire start_leave;
22.
23.     reg             [  7: 0]    uart_tx_data        ;
24.     reg                     uart_tx_req         ;
25.     reg                     work_en             ;
26.     reg             [  6: 0]    tx_byte_cnt         ;
27.     reg             [111: 0]    str                 ;
28.
29.
30.     reg D0_syn1;
31.     reg D0_syn2;
32.     reg D0_syn3;
33.
34.     assign                  start_approach          = ~D0_syn2 & D0_syn3;//下降沿//物体离开
35.     assign                  start_leave             = D0_syn2 & ~D0_syn3;//上升沿//物体接近
36.
37.   //时钟同步以及打拍
38.     always @(posedge clk or negedge rst_n)begin
39.       if(!rst_n)begin
40.         D0_syn1<= 1'b0;
41.         D0_syn2<= 1'b0;
42.         D0_syn3<= 1'b0;
43.       end
44.       else begin
45.         D0_syn1<= D0;
46.         D0_syn2<= D0_syn1;
47.         D0_syn3<= D0_syn2;
48.       end
49.     end
50.
51.
52.   always@(posedge clk or negedge rst_n)
53.     if(rst_n == 1'b0)
54.       work_en <= 1'b0;
```

```
55.     else    if(start_approach||start_leave)
56.       work  en <= 1'b1;
57.     else    if(tx_byte_cnt== 7'd15&&uart_tx_done)
58.       work  en <= 1'b0;
59.     else
60.       work  en <= work  en;
61.
62.   //串口发送逻辑
63.   always @(posedge clk or negedge rst_n) begin
64.     if (!rst_n) begin
65.       tx_byte_cnt <= 7'b0;
66.     end else if (work_en) begin
67.       if (tx_byte_cnt == 7'd15&&uart_tx_done)                  // 总共发送 16 字节
68.           tx_byte_cnt <= 7'b0;
69.       else if(uart_tx_done)
70.           tx_byte_cnt <= tx_byte_cnt + 1'b1;
71.     end
72.   end
73.
74.   always @(posedge clk or negedge rst_n)
75.     if (!rst_n)
76.       str <= 112'b0;
77.     else if(start_approach)
78.       str <= STR_approach;
79.     else if(start_leave)
80.       str <= STR  leave;
81.     else
82.       str <= str;
83.
84.   // 串口发送数据控制
85.   always @(posedge clk or negedge rst_n) begin
86.     if (!rst_n) begin
87.       uart_tx_req <= 1'b0;
88.       uart_tx_data <= 8'b0;
89.
90.     end else if (work_en && !uart_tx_busy) begin
91.       case (tx_byte_cnt)                                  // 根据当前字节计数器发送数据
92.           7'd1: uart_tx_data <= str[111:104];              // "检"
93.           7'd2: uart_tx_data <= str[103:96];               // "检"
94.           7'd3: uart_tx_data <= str[95:88];                // "测"
95.           7'd4: uart_tx_data <= str[87:80];                // "测"
96.           7'd5: uart_tx_data <= str[79:72];                // "到"
97.           7'd6: uart_tx_data <= str[71:64];                // "到"
98.           7'd7: uart_tx_data <= str[63:56];                // "物"
99.           7'd8: uart_tx_data <= str[55:48];                // "物"
100.          7'd9: uart_tx_data <= str[47:40];                // "体"
101.          7'd10: uart_tx_data <= str[39:32];               // "体"
102.          7'd11: uart_tx_data <= str[31:24];               // "接"/"离"
103.          7'd12: uart_tx_data <= str[23:16];               // "接"/"离"
```

```
104.            7'd13: uart_tx_data <= str[15:8];                  // "进"/"开"
105.            7'd14: uart_tx_data <= str[7:0];                   // "进"/"开"
106.            7'd15: uart_tx_data <= 8'h0A;                      // 换行符
107.            default: uart_tx_data <= 8'b0;
108.        endcase
109.        uart_tx_req <= 1'b1;                                   // 拉高请求信号
110.    end else begin
111.        uart_tx_req <= 1'b0;                                   // 请求信号拉高仅一个时钟周期
112.    end
113. end
114. uart_tx u_uart_tx(
115.    .clk                    (clk                    ),
116.    .rst_n                  (rst_n                  ),
117.    .uart_tx_req            (uart_tx_req&work_en    ),// 发送使能请求
118.    .uart_tx_data           (uart_tx_data           ),// UART 要发送的数据
119.    .uart_txd               (uart_txd               ),// UART 发送端口
120.    .uart_tx_done           (uart_tx_done           ),//8bit 数据发送完成
121.    .uart_tx_busy           (uart_tx_busy           ) // 发送忙
122. );
123.  endmodule
```

在这个模块中，由于模块输出的 D0 信号时钟与我们开发板的时钟不同步，我们使用 D0_syn1、D0_syn2、D0_syn3 寄存器对 D0 信号进行时钟同步，同时获取上升沿 start_leave 和下降沿 start_approach。在之前的介绍中我们提到 D0 信号默认是高电平，当模块检测到有物体时，D0 才会变成低电平。于是我们捕获 D0 信号的下降沿，作为触发信号，发送 16 字节的“检测到物体接近”，同样的捕获 D0 信号的上升沿作为触发信号发送“检测到物体离开”（汉字编码使用 GB2312 编码）。

对于串口发送模块的逻辑在“串口回环实验例程”章节已经介绍，若不熟悉请到对应章节进行了解。

约束文件如下：

```
1.  define_attribute {p:uart_txd} {PAP_IO_DIRECTION} {OUTPUT}
2.  define_attribute {p:uart_txd} {PAP_IO_LOC} {33}
3.  define_attribute {p:uart_txd} {PAP_IO_VCCIO} {1.2}
4.  define_attribute {p:uart_txd} {PAP_IO_STANDARD} {LVCMOS12}
5.  define_attribute {p:uart_txd} {PAP_IO_DRIVE} {2}
6.  define_attribute {p:uart_txd} {PAP_IO_NONE} {TRUE}
7.  define_attribute {p:uart_txd} {PAP_IO_SLEW} {SLOW}
8.  define_attribute {p:D0} {PAP_IO_DIRECTION} {INPUT}
9.  define_attribute {p:D0} {PAP_IO_LOC} {92}
10. define_attribute {p:D0} {PAP_IO_VCCIO} {1.2}
11. define_attribute {p:D0} {PAP_IO_STANDARD} {LVCMOS12}
```

```
define_attribute {p:D0} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:clk} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:clk} {PAP_IO_LOC} {5}
define_attribute {p:clk} {PAP_IO_VCCIO} {1.2}
define_attribute {p:clk} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:clk} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:rst_n} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:rst_n} {PAP_IO_LOC} {19}
define_attribute {p:rst_n} {PAP_IO_VCCIO} {1.2}
define_attribute {p:rst_n} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:rst_n} {PAP_IO_PULLUP} {TRUE}
```

17.4、代码仿真

由于 TCRT5000 模块的使用方法简单，主要是 FPGA 逻辑对 TCRT5000 模块传回的数据进行处理。至于串口发送的逻辑相信读者已经很清楚了，我们可以将 TCRT5000 模块输入 FPGA 的信号 D0_syn3（时钟同步后的信号）打上 debug 标记，分析 TCRT5000 模块的工作状态。

当没有物体接近时模块输入的 D0_syn3 为高电平，如下图：

图 3-17.4-1 反射式光传感器实验仿真波形图 1

当有物体接近时模块输入的 D0_syn3 为低电平，如下图：

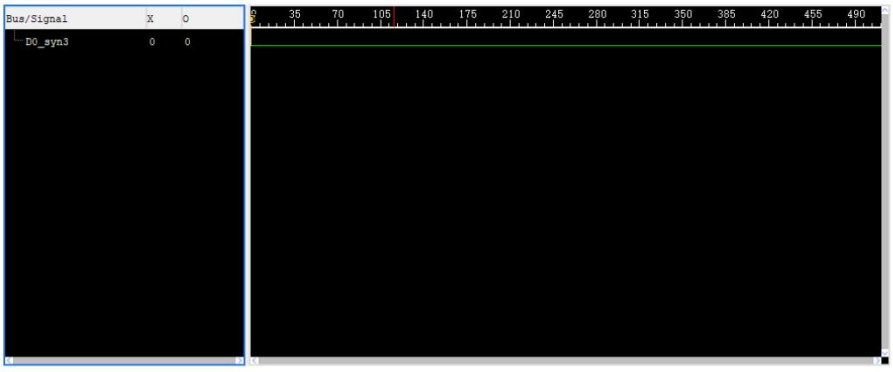


图 3-17.4-2 反射式光传感器实验仿真波形图 2

我们在代码中获取其下降沿 start_approach 与上升沿 start_leave 作为串口数据的发送标志，将物体离开或接近信息发送到上位机进行查看。

17.5、实验现象

将程序下载进入板卡，根据约束文件将板卡与 TCTR5000 模块连接。打开串口助手，设置波特率为 115200，使用物体靠近和原理模块的发射管和接收管。可以观察到串口助手有如下信息：

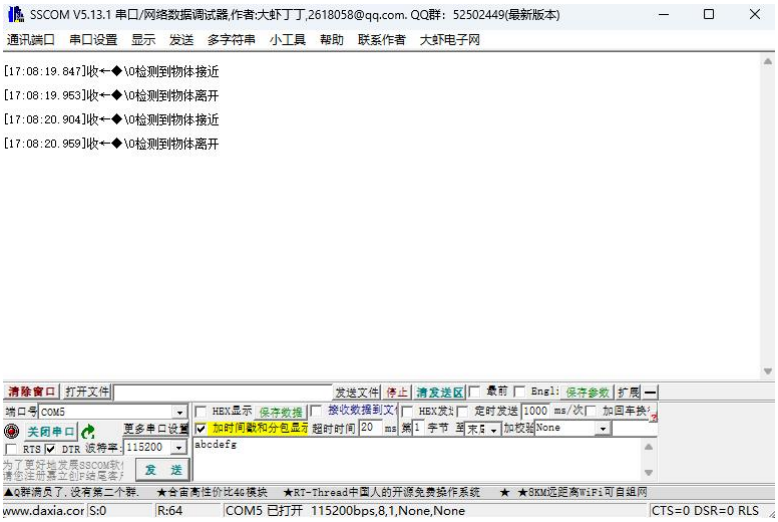


图 3-17.5-1 反射式光传感器实验现象图

18、温湿度测量实验例程

18.1、实验目的

使用逻辑派 Z1 开发板（基于紫光同创 compa 系列 PGC4KD-6ILPG144 芯片）编写 DHT11 温湿度传奇模块驱动代码，使用按键触发对环境温度、湿度的测量并使用串口向上位机发送当前测量到的温度、湿度。

18.2、实验原理

18.2.1、DHT11 模块介绍

DHT11 是一种数字温湿度传感器，集成了温度和湿度测量功能。它采用电容式感湿元件和负温度系数电阻测温元件，并结合高性能 8 位单片机进行信号的采集处理和输出。DHT11 通过单线数字接口与微处理器通信，输出经过校准的数字信号，温度测量范围为 0 至 50 摄氏度，精度为±2℃；湿度测量范围为 20%至 90%RH，精度为±5%RH。因其良好的稳定性和性价比，被广泛应用于各种环境监测设备、智能家居系统和自动化控制领域。其模块图如下：

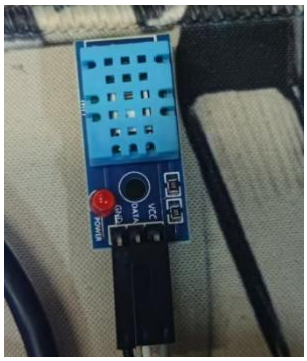


图 3-18.2-1 DHT11 模块图

模块接口描述如下表：

表 3-18-1 温湿度测量实验模块接口表

接口名称	接口描述
VCC	电源正极 3~5.5V
GND	电源负级
DATA	串行数据，单总线

18.2.2、DHT11 模块通讯协议以及数据格式

DHT11 是一种数字温湿度传感器，它通过单总线协议与微处理器通信。正常情况下，DHT11 处于低功耗模式。当单片机需要读取温湿度数据时，首先发送一次复位信号。DHT11 接收到复位信号后，从低功耗模式转换到高速模式，开始执行一次温湿度采集过程，完成后通过单总线发送响应信号，并将总线拉高，准备传输数据。

一次完整的数据传输包括 40 位（5 字节）的数据，具体格式为：8 位湿度整数数据 ＋8 位湿度小数数据 ＋8 位温度整数数据 ＋8 位温度小数数据 ＋8 位校验和。由于 DHT11 的分辨率限制，湿度和温度的小数部分数据始终为 0。校验和是前 4 个字节数据的累加和，用于验证数据传输的准确性，确保接收端接收到的数据没有错误。

DHT11 只有在接收到开始信号后才会触发温湿度采集，如果没有接收到复位信号，它将保持在低功耗模式，不会主动进行数据采集。当数据采集和传输完成后，DHT11 会自动切换回低功耗模式，等待下一次触发信号。

通讯过程示意图如下：

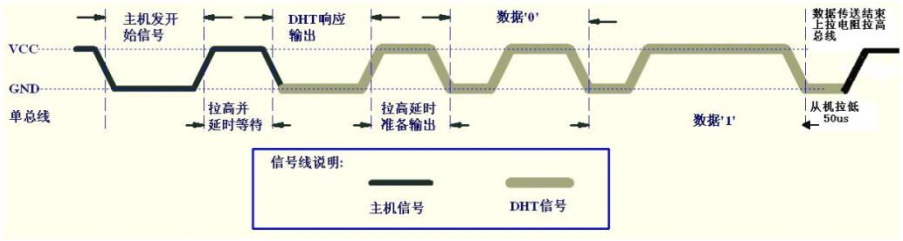


图 3-18.2-2 DHT11 通信过程示意图

总线空闲状态为高电平,主机把总线拉低等待 DHT11 响应,主机把总线拉低必须大于 18 毫秒，保证 DHT11 能检测到起始信号，然后拉高等待 20-40us, 读取 DHT11 的响应信号，此时主机释放总线。DHT11 接收到主机的开始信号后，发送（70~100）us 低电平响应信号。然后然后 DHT11 拉高总线 （70~100）us，开始传输数据。

主机发送触发信号与 DHT11 响应的过程示意图如下：

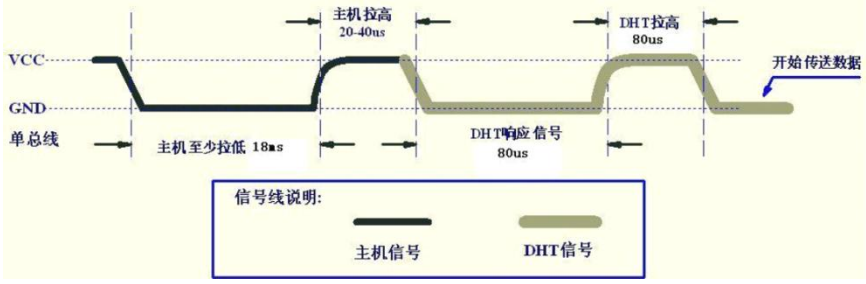


图 3-18.2-3 DHT11 响应过程示意图

由该图可知触发信号由主机发送，主机需要拉低总线至少 18ms，然后再拉高总线，延时 20~40us。DHT11 检测到触发信号后，开始一次采样，并拉低总线 80us 表示响应信号，告诉主机数据已经准备好了。然后 DHT11 拉高总线 80us，之后开始传输数据。

数字 1 与数字 0 的表示方法示意图如下：

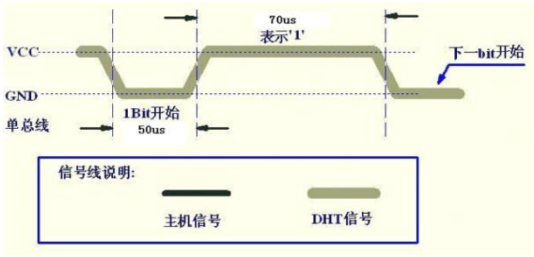


图 3-18.2-4 数字 1 表示方法示意图

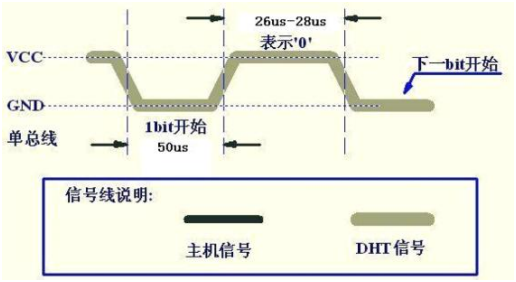


图 3-18.2-5 数字 0 表示方法示意图

由示意图可知“0”的高电平持续 26~28us，“1”的高电平持续 70us，每一位数据前都有 50us 的起始时隙。到此 DHT11 的通讯协议以及数据格式就介绍完了。

18.3、代码设计

DHT11 驱动模块端口描述如下表：

表 3-18-2 DHT11 驱动模块端口表

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
cap_start_flag	input	1	触发一次采样
dht11	inout	1	DHT11 通讯总线
data_vld	output	1	输出温度有效数据标志

temperature_data	output	32	模块测量得到的温湿度数据

DHT11 驱动模块代码如下：

```
1.  module DHT11_drive(
2.      input                   clk             ,
3.      input                   rst_n           ,
4.      input                   cap_start_flag  ,//触发采样
5.      inout                   dht11           ,//单总线（双向信号）
6.
7.      output reg              data_vld        ,
8.      output reg  [ 31: 0]    temperature_data    //输出的有效数据
9.  );
10.
11.
12. //参数定义
13.     localparam              WAIT_1S         = 7'b0000001,
14.                             WAIT_TRI        = 7'b0000010 ,
15.                             START           = 7'b0000100 ,
16.                             DELAY_30US      = 7'b0001000 ,
17.                             BUS_REPLY_LOW   = 7'b0010000 ,
18.                             BUS_REPLY_HIGH  = 7'b0100000 ,
19.                             REV_data        = 7'b1000000 ;
20.
21.     localparam              TIME_1S         = 999_999,      //上电 1s 延时计数，单位 us
22.             TIME_BE = 17_999    ,               //主机起始信号拉低时间，单位 us
23.             TIME_GO = 30        ;               //主机释放总线时间，单位 us
24.
25. //信号申明
26.     reg         [  6: 0]    cur_state       /*synthesisPAP_MARK_DEBUG="true"*/ ;//现态
27.     reg         [  6: 0]    next_state      ;//次态
28.     reg         [  4: 0]    cnt             ;//50 分频计数器，1Mhz(1us)
29.     reg                     dht11_out       ;//双向总线输出
30.     reg                     dht11_en        ;//双向总线输出使能，1 则输出，0 则高阻态
31.     reg                     dht11_d1        ;//总线信号打 1 拍
32.     reg                     dht11_d2        ;//总线信号打 2 拍
33.     reg                     clk_1us         ;//us 时钟
34.     reg         [ 21: 0]    us_cnt          ;//us 计数器,最大可表示 4.2s
35.     reg         [  5: 0]    bit_cnt         ;//接收数据计数器，最大可以表示 64 位
36.     reg         [ 39: 0]    data_temp       ;//包含校验的 40 位输出
37.
38.     wire                    dht11_in        ;//双向总线输入
39.     wire                    dht11_rise      ;//上升沿
```

```
40.     wire                    dht11_fall          ;//下降沿
41.
42.     // 脉冲扩展
43.     reg             [  6: 0]    counter             ;
44.     reg                     cap_start           ;
45.     always @(posedge clk or negedge rst_n) begin
46.       if (!rst_n) begin
47.           cap_start <= 1'b0;
48.           counter <= 0;
49.       end else begin
50.           if (cap_start_flag) begin
51.             cap_start <= 1'b1;
52.             counter <= 200;
53.           end else if (counter > 0) begin
54.             cap_start <= 1'b1;                  // 计数器递减，维持高电平
55.             counter <= counter 1;
56.           end else begin
57.             cap_start <= 1'b0;                  // 计数结束，输出恢复低电平
58.           end
59.       end
60.     end
61.
62.
63.   //双向端口使用方式
64.     assign                  dht11_in            = dht11;//高阻态的话，则把总线上的数据赋给 dht11_in
65.     assign                  dht11               = dht11_en ? dht11_out : 1'bz;//使能 1 则输出，0 则高阻态
66.
67.   //us 时钟生成，因为时序都是以 us 为单位，所以生成一个 1us 的时钟会比较方便
68.   //50 分频计数
69.   always @(posedge clk or negedge rst_n)begin
70.     if(!rst_n)
71.       cnt <= 5'd0;
72.     else if(cnt == 5'd24)                       //每 25 个时钟 500ns 清零
73.       cnt <= 5'd0;
74.     else
75.       cnt <= cnt + 1'd1;
76.   end
77.   //生成 1us 时钟
78.   always @(posedge clk or negedge rst_n)begin
79.     if(!rst_n)
80.       clk_1us <= 1'b0;
81.     else  if(cnt == 5'd24)                      //每 500ns
82.       clk_1us <= ~clk_1us;                      //时钟反转
83.     else
84.       clk_1us <= clk_1us;
85.   end
86.
```

```
87.  //检测总线上的上升沿和下降沿
88.    assign                    dht11_rise          = ~dht11_d2 && dht11_d1;//上升沿
89.    assign                    dht11_fall          = ~dht11_d1 && dht11_d2;//下降沿
90.  always @(posedge clk  1us or negedge rst  n)begin
91.    if(!rst_n)begin
92.      dht11  d1 <= 1'b0;
93.      dht11_d2 <= 1'b0;
94.    end
95.    else begin
96.      dht11  d1 <= dht11;
97.      dht11_d2 <= dht11_d1;
98.    end
99.  end
100.
101. //同步时序描述状态转移
102. always @(posedge clk_1us or negedge rst_n)begin
103.   if(!rst_n)
104.     cur_state <= WAIT_1S;
105.   else
106.     cur_state <= next_state;
107. end
108. //组合逻辑判断状态转移条件，描述状态转移规律以及输出
109. always @(*)begin
110.   next_state = WAIT_1S;
111.   case(cur_state)
112.     WAIT_1S       :begin
113.       if(us_cnt == TIME_1S)                    //满足上电延时的时间
114.         next_state = WAIT_TRI;
115.       else
116.         next_state = WAIT_1S;
117.     end
118.     WAIT_TRI    :begin
119.       if(cap_start)
120.         next_state = START;
121.       else
122.         next_state = WAIT_TRI;
123.     end
124.     START       :begin
125.       if(us_cnt == TIME_BE)                    //满足拉低总线的时间
126.         next_state = DELAY_30US;
127.       else
128.         next_state = START;
129.     end
130.     DELAY_30US  :begin
131.       if(us_cnt == TIME_GO)                    //满足主机释放总线时间
132.         next  state = BUS  REPLY  LOW;
133.       else
134.         next  state = DELAY  30US;
135.     end
```

```
136.      BUS_REPLY_LOW   :begin
137.        if(us_cnt <= 'd500)begin                    //不到 500us
138.          if(dht11_rise && us_cnt >= 'd70
139.          && us_cnt <= 'd100)                       //上升沿响应，且低电平时间介于
70~100us
140.            next_state = BUS_REPLY_HIGH;            //跳转到 DELAY_75us
141.          else
142.            next_state = BUS_REPLY_LOW;             //条件不满足状态不变
143.        end
144.        else
145.          next_state = START;                       //超过 500us 仍没有上升沿响应则
跳转到 START
146.      end
147.      BUS_REPLY_HIGH  :begin
148.        if(dht11_fall && us_cnt >= 'd70)            //上升沿响应，且低电平时间大
于 70us
149.          next_state = REV_data;                    //跳转到 REV_data
150.        else
151.          next_state = BUS_REPLY_HIGH;              //条件不满足状态不变
152.      end
153.      REV_data    :begin
154.        if(dht11_rise && bit_cnt == 'd40)           //接收完了所有 40 个数据后会
拉低一段时间作为结束
155.                                                    //捕捉到上升沿且接收数据个数为
40
156.          next_state = WAIT_TRI;                    //状态跳转到 START，重新开始
新一轮采集
157.        else
158.          next_state = REV_data;                    //条件不满足状态不变
159.      end
160.      default:next_state = START;                   //默认状态为 START
161.    endcase
162. end
163.
164. //时序逻辑描述输出
165. always @(posedge clk_1us or negedge rst_n)begin
166.    if(!rst_n)begin                                 //复位状态下输出如下
167.      dht11_en <= 1'b0;
168.      dht11_out <= 1'b0;
169.      us_cnt <= 22'd0;
170.      bit_cnt <=  6'd0;
171.      data_temp <= 40'd0;
172.    end
173.    else
174.      case(cur_state)
175.        WAIT_1S       :begin
176.          dht11_en <= 1'b0;                         //释放总线，由外部电阻拉高
177.          if(us_cnt == TIME_1S)
178.            us_cnt <= 22'd0;
```

```
179.          else
180.            us_cnt <= us_cnt + 1'd1;
181.        end
182.        WAIT_TRI    :begin
183.          dht11_en <= 1'b0;
184.        end
185.        START       :begin
186.          dht11_en <= 1'b1;                    //占用总线
187.          dht11_out <= 1'b0;                   //输出低电平
188.          if(us_cnt == TIME_BE)
189.            us_cnt <= 22'd0;
190.          else
191.            us_cnt <= us_cnt + 1'd1;
192.        end
193.        DELAY_30US    :begin
194.          dht11_en <= 1'b0;                    //释放总线，由外部电阻拉高//这里
是否要拉高
195.          if(us_cnt == TIME_GO)
196.            us_cnt <= 22'd0;
197.          else
198.            us_cnt <= us_cnt + 1'd1;
199.        end
200.        BUS_REPLY_LOW       :begin
201.          dht11_en <= 1'b0;                    //释放总线，由外部电阻拉高
202.          if(us_cnt <= 'd500)begin               //计时不到 500us
203.            if(dht11_rise && us_cnt >= 'd70
204.            && us_cnt <= 'd100)                  //上升沿响应，且低电平时间介于
70~100us
205.              us_cnt <= 22'd0;                   //计时清零
206.            else
207.              us_cnt <= us_cnt + 1'd1;
208.          end
209.          else
210.            us_cnt <= 22'd0;                   //超过 500us 仍没有上升沿响应，则
计数清零
211.        end
212.        BUS_REPLY_HIGH  :begin
213.          dht11_en <= 1'b0;                    //释放总线，由外部电阻拉高
214.          if(dht11_fall && us_cnt >= 'd70)          //上升沿响应，且低电平时间大
于 70us
215.            us_cnt <= 22'd0;                   //计时清零
216.          else
217.            us_cnt <= us_cnt + 1'd1;
218.        end
219.        REV_data    :begin
220.          dht11_en <= 1'b0;                    //释放总线，由外部电阻拉高，进入
读取状态
221.          if(dht11_rise && bit_cnt == 'd40)begin          //数据接收完毕
222.            bit_cnt <=  6'd0;
```

```
223.              us_cnt <= 22'd0;
224.            end
225.            else if(dht11_fall)begin                    //检测到低电平，则说明接收到一个
数据
226.              bit_cnt <= bit_cnt + 1'd1;                //数据接收计数器+1
227.              us_cnt <= 22'd0;
228.              if(us  cnt <= 'd100)
229.                data_temp[39-bit_cnt] <= 1'b0;          //总共所有的时间少于 100us,则
说明接收到“0”
230.              else
231.                data_temp[39-bit_cnt] <= 1'b1;          //总共所有的时间大于 100us,则
说明接收到“1”
232.            end
233.            else begin                                  //所有数据没有接收完，且正处于 1 个
数据的接收进程中
234.              bit  cnt <= bit  cnt;
235.              data_temp <= data_temp;
236.              us  cnt <= us  cnt + 1'd1;
237.            end
238.          end
239.          default:;
240.        endcase
241. end
242.
243. //校验读取的数据是否符合校验规则
244. always @(posedge clk_1us or negedge rst_n)begin
245.    if(!rst_n)
246.      temperature_data <= 32'd0;
247.    else if((data_temp[7:0] == data_temp[39:32] + data_temp[31:24] +
248.    data_temp[23:16] + data_temp[15:8]))
249.      temperature_data <= data_temp[39:8];             //符合规则，则把有效数据赋
值给输出
250.    else
251.      temperature_data <= temperature_data;            //不符合规则，则舍弃这次
读取的数据，输出仍保持上次的状态不变
252. end
253.
254.
255.
256. always @(posedge clk  1us or negedge rst  n)begin
257.    if(!rst_n)
258.      data  vld <= 1'b0;
259.    else if(dht11_rise && bit_cnt == 'd40)
260.      data  vld <= 1'b1;
261.    else
262.      data  vld <= 1'b0;
263. end
264.
265.
```

```
266. endmodule
```

模块使用一个三段式状态机处理板卡与 DHT11 的通讯逻辑。我们可以通过状态机的跳转过程来理解模块的工作逻辑。这个状态机总共有如下状态：WAIT_1S（上电延迟状态）、WAIT_TRI（等待触发状态）、START（发送触发信号状态）、DELAY_30US（主机释放总线 30us）、BUS_REPLY_LOW（接收 DHT11 低电平响应）、BUS_REPLY_HIGH（接收 DHT11 高电平响应）、REV_data（解析 DHT11 数据）。下面我们挨个解释这些状态:

1. WAIT_1S：上电后等待 1 秒的时间，确保 DHT11 传感器稳定工作。

2. WAIT_TRI：等待外部触发信号 cap_start_flag，用于启动温湿度数据采集。cap_start_flag 在顶层模块由按键产生。

3. START：发送触发信号，拉低总线 18 毫秒，使 DHT11 传感器进入响应模式。

4. DELAY_30US：主机释放总线 30 微秒，等待 DHT11 传感器的响应信号。

5. BUS_REPLY_LOW：检测 DHT11 传感器的低电平响应信号，低电平持续时间由 us_cnt 计数，需要满足低电平持续时间在 70~100us。

6. BUS_REPLY_HIGH：检测 DHT11 传感器的高电平响应信号，低电平持续时间由 us_cnt 计数，需要满足高电平持续时间大于在 70us。

7. REV_data：当响应信号接收无误之后开始，接收 DHT11 传感器发送的 40 位数据，并根据接收到的数据位更新内部数据存储，直到接收完所有数据。

这里需要注意一下，代码中：assign dht11 = dht11_en ? dht11_out : 1'bz;

通过 dht11_en 信号来控制双向端口 dht11。当 dht11_en 为 1 时双向端口 dht11 为输出，当 dht11_en 为 0 时双向端口 dht11 为输入。这是一种常用的控制双向端口的方式。

模块顶层代码端口描述如下表：

表 3-18-3 温湿度测量实验顶层模块端口表

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号

key_in	input	1	通过按键触发一次采样
dht11	inout	1	DHT11 通讯总线
uart_txd	output	1	串口发送端口

顶层模块代码如下：

```
1.  module DHT11_top(
2.      input                   clk             ,
3.      input                   rst_n           ,
4.      inout                   dht11           ,
5.
6.      input                   key_in          ,
7.      output                  uart_txd
8.  );
9.
10.
11.     parameter               CNT_MAX         = 20'd999_999; //消抖计数器
12.     parameter               WENDU_STR       = 48'hce_c2_b6_c8_a3_ba;//“温
度：”的 GB2312 编码
13.     parameter               SHIDU_STR       = 48'hca_aa_b6_c8_a3_ba;//“湿
度：”的 GB2312 编码
14.
15.  //信号定义
16.     wire                    data_vld        ;
17.     wire        [ 31: 0]    temperature_data    ;
18.     wire                    uart_tx_busy        ;
19.     wire                    uart_tx_done        /*synthesisPAP_MARK_DEBUG="true"
*/;
20.
21.     reg         [  7: 0]    uart_tx_data        ;
22.     reg                     uart_tx_req         /*synthesisPAP_MARK_DEBUG="true"*
/;
23.     reg                     work_en             /*synthesisPAP_MARK_DEBUG="true"*/
;
24.     reg         [  6: 0]    tx_byte_cnt         /*synthesisPAP_MARK_DEBUG="true
"*/;
25.     reg         [ 47: 0]    wendu_str           ;
26.     reg         [ 47: 0]    shidu_str           ;
27.     reg         [ 19: 0]    cnt_20ms            ;//消抖计数器
28.     reg                     key_flag            ;
29.     reg         [  7: 0]    ascii_table     [0:9] ;//储存 0~9 的 ASCII 值
30.
31.
32.
33.
```

```
34.  //cnt_20ms:如果时钟的上升沿检测到外部按键输入的值为低电平时，计数器开始计数
35.  always@(posedge clk or negedge rst_n)
36.    if(rst_n == 1'b0)
37.      cnt_20ms <= 20'b0;
38.    else   if(key_in == 1'b1)
39.      cnt_20ms <= 20'b0;
40.    else   if(cnt_20ms == CNT_MAX && key_in == 1'b0)
41.      cnt_20ms <= cnt_20ms;
42.    else
43.      cnt_20ms <= cnt_20ms + 1'b1;
44.
45.
46.  always@(posedge clk or negedge rst_n)
47.    if(rst_n == 1'b0)
48.      key_flag <= 1'b0;
49.    else   if(cnt_20ms == CNT_MAX 1'b1)
50.      key_flag <= 1'b1;
51.    else
52.      key_flag <= 1'b0;
53.
54.
55.  always@(posedge clk or negedge rst_n)
56.    if(rst_n == 1'b0)
57.      work_en <= 1'b0;
58.    else   if(data  vld == 1'b1)
59.      work_en <= 1'b1;
60.    else   if(tx  byte  cnt== 7'd23&&uart  tx  done)
61.      work_en <= 1'b0;
62.    else
63.      work_en <= work_en;
64.
65.
66.  //串口发送逻辑
67.
68.
69.  always @(posedge clk or negedge rst_n) begin
70.    if (!rst_n) begin
71.      tx_byte_cnt <= 7'b0;
72.    end else if (work_en) begin
73.      if (tx_byte_cnt == 7'd23&&uart_tx_done)                  // 总共发送 24 字节
74.          tx_byte_cnt <= 7'b0;
75.      else if(uart_tx_done)
76.          tx_byte_cnt <= tx_byte_cnt + 1'b1;
77.    end
78.  end
79.
80.
81.
```

```
82.
83.  // 串口发送数据控制
84.  always @(posedge clk or negedge rst_n) begin
85.    if (!rst_n) begin
86.      uart_tx_req <= 1'b0;
87.      uart_tx_data <= 8'b0;
88.      wendu_str <= WENDU_STR;
89.      shidu_str <= SHIDU_STR;
90.      ascii_table[0] = 8'd48; //'0'
91.      ascii_table[1] = 8'd49; //'1'
92.      ascii_table[2] = 8'd50; //'2'
93.      ascii_table[3] = 8'd51; //'3'
94.      ascii_table[4] = 8'd52; //'4'
95.      ascii_table[5] = 8'd53; //'5'
96.      ascii_table[6] = 8'd54; //'6'
97.      ascii_table[7] = 8'd55; //'7'
98.      ascii_table[8] = 8'd56; //'8'
99.      ascii_table[9] = 8'd57; //'9'
100.
101.
102.   end else if (work_en && !uart_tx_busy) begin
103.     case (tx_byte_cnt)                          // 根据当前字节计数器发送数据
104.         7'd0: uart_tx_data <= wendu_str[47:40];          // "温"
105.         7'd1: uart_tx_data <= wendu_str[39:32];          // "温"
106.         7'd2: uart_tx_data <= wendu_str[31:24];          // "度"
107.         7'd3: uart_tx_data <= wendu_str[23:16];          // "度"
108.         7'd4: uart_tx_data <= wendu_str[15:8];           // "："
109.         7'd5: uart_tx_data <= wendu_str[7:0];            // "："
110.         7'd6: uart_tx_data <= ascii_table[temperature_data[15:12]];      // 温度高字节
的 ASCII
111.         7'd7: uart_tx_data <= ascii_table[temperature_data[11:8]];       // 温度低字节
的 ASCII
112.         7'd8: uart_tx_data <= 8'h2e;                     // 小数点
113.         7'd9: uart_tx_data <= ascii_table[temperature_data[7:4]];        // 温度高字节
的 ASCII
114.         7'd10: uart_tx_data <= ascii_table[temperature_data[3:0]];       // 温度低字节
的 ASCII
115.         7'd11: uart_tx_data <= 8'h20;                    // 空格
116.         7'd12: uart_tx_data <= shidu_str[47:40];         // "湿"
117.         7'd13: uart_tx_data <= shidu_str[39:32];         // "湿"
118.         7'd14: uart_tx_data <= shidu_str[31:24];         // "度"
119.         7'd15: uart_tx_data <= shidu_str[23:16];         // "度"
120.         7'd16: uart_tx_data <= shidu_str[15:8];          // "："
121.         7'd17: uart_tx_data <= shidu_str[7:0];           // "："
122.         7'd18: uart_tx_data <= ascii_table[temperature_data[31:28]];     // 湿度高字
节的 ASCII
123.         7'd19: uart_tx_data <= ascii_table[temperature_data[27:24]];     // 湿度高字
节的 ASCII
124.         7'd20: uart_tx_data <= 8'h2e;                    // 小数点
```

```
125.            7'd21: uart_tx_data <= ascii_table[temperature_data[23:20]];        // 湿度低字节的 ASCII
126.            7'd22: uart_tx_data <= ascii_table[temperature_data[19:16]];        // 湿度低字节的 ASCII
127.            7'd23: uart_tx_data <= 8'h0A;                       // 换行符
128.            default: uart_tx_data <= 8'b0;
129.        endcase
130.        uart_tx_req <= 1'b1;                                    // 拉高请求信号
131.    end else begin
132.        uart_tx_req <= 1'b0;                                    // 请求信号拉高仅一个时钟周期
133.    end
134. end
135.
136.
137.
138. DHT11_drive u_DHT11_drive(
139.    .clk                (clk                ),
140.    .rst_n              (rst_n              ),
141.    .cap_start_flag     (key_flag           ),
142.    .dht11              (dht11              ),// 单总线（双向信号）
143.    .data_vld           (data_vld           ),
144.    .temperature_data   (temperature_data   ) // 输出的有效数据
145. );
146.
147.
148.
149.
150. uart_tx u_uart_tx(
151.    .clk                (clk                ),
152.    .rst_n              (rst_n              ),
153.    .uart_tx_req        (uart_tx_req&work_en    ),// 发送使能请求
154.    .uart_tx_data       (uart_tx_data       ),// UART 要发送的数据
155.    .uart_txd           (uart_txd           ),// UART 发送端口
156.    .uart_tx_done       (uart_tx_done       ),//8bit 数据发送完成
157.    .uart_tx_busy       (uart_tx_busy       ) // 发送忙
158. );
159.
160.
161. endmodule
```

我们在顶层模块接收用户输入的按键信号 key_in 使用一个 20ms 计数器对其进行消抖产生温湿度驱动模块的采样触发信号 cap_start_flag。将 DHT11 得到的数据通过串口发送到上位机进行查看，汉字使用 GB2312 编码，模块中初始化了一个 ascii_table 用来储存阿拉伯数字 0~9 的 ASCII 值，将 DHT11 采样得到的温湿度数据在 ascii_table

索引到对应的 ASCII 值进行发送。其中串口发送部分的知识在之前的章节已有介绍，读者若对这部分不熟悉，可到串口回环章节进行查看。

约束代码如下：

```
define_attribute {p:dht11} {PAP_IO_DIRECTION} {INOUT}
define_attribute {p:dht11} {PAP_IO_LOC} {92}
define_attribute {p:dht11} {PAP_IO_VCCIO} {1.2}
define_attribute {p:dht11} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:dht11} {PAP_IO_DRIVE} {2}
define_attribute {p:dht11} {PAP_IO_NONE} {TRUE}
define_attribute {p:dht11} {PAP_IO_SLEW} {SLOW}
define_attribute {p:uart_txd} {PAP_IO_DIRECTION} {OUTPUT}
define_attribute {p:uart_txd} {PAP_IO_LOC} {33}
define_attribute {p:uart_txd} {PAP_IO_VCCIO} {1.2}
define_attribute {p:uart_txd} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:uart_txd} {PAP_IO_DRIVE} {2}
define_attribute {p:uart_txd} {PAP_IO_NONE} {TRUE}
define_attribute {p:uart_txd} {PAP_IO_SLEW} {SLOW}
define_attribute {p:clk} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:clk} {PAP_IO_LOC} {5}
define_attribute {p:clk} {PAP_IO_VCCIO} {1.2}
define_attribute {p:clk} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:clk} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:key_in} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:key_in} {PAP_IO_LOC} {20}
define_attribute {p:key_in} {PAP_IO_VCCIO} {1.2}
define_attribute {p:key_in} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:key_in} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:rst_n} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:rst_n} {PAP_IO_LOC} {19}
define_attribute {p:rst_n} {PAP_IO_VCCIO} {1.2}
define_attribute {p:rst_n} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:rst_n} {PAP_IO_PULLUP} {TRUE}
```

18.4、代码仿真

由于本实验需要与 DHT11 通过总线交互信息，在测试文件中编写 DHT11 的应答逻辑比较麻烦，我们可以将 DHT11 驱动模块中的关键信号打上 debug 标记，分析 DHT11 驱动模块是否正确工作，同时检查 DHT11 是否正确应答我们通过总线发送的信息。

DHT11 的控制流程大致为主机触发、DHT11 应答、DHT11 发送温度数据主机接收温度数据。我们将 DHT11 驱动代码中的状态机 cur_stste、顶层模块发送的采样触

发信号 cap_start_flag、DHT11 总线信号 dht11、微秒计数器 us_cnt、接收数据比特计数器 bit_cnt 添加上 debug 标记。

将触发模式设置为 cap_start_flag 高电平触发，按下板卡按键 k0，捕捉到的波形如下图所示，总线信号 dht11 在空闲状态为高电平，在顶层发送触发信号后状态机由等待触发状态（WAIT_TIR）跳转至 START 状态，此时 FPGA 占用总线并拉低总线 18ms，向 DHT11 发送开始信号。

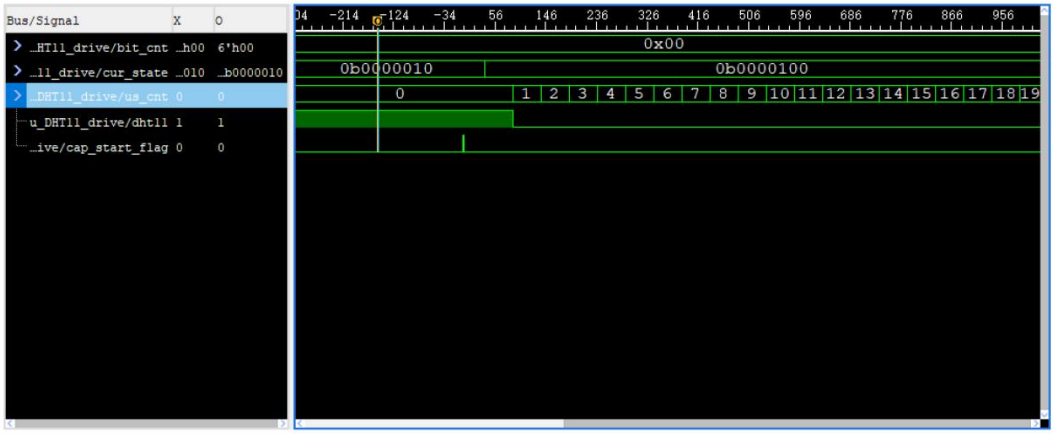


图 3-18.4-1 温湿度测量实验仿真波形图 1

将触发模式设置为（cur_state = 7'b0010000&&us_cnt = 70）（状态机在接收来自 DHT11 的低电平响应状态），按下按键，观察捕获到的波形图，如果状态机正确跳转，那么说明我们向 DHT11 发送的触发信号被回应。

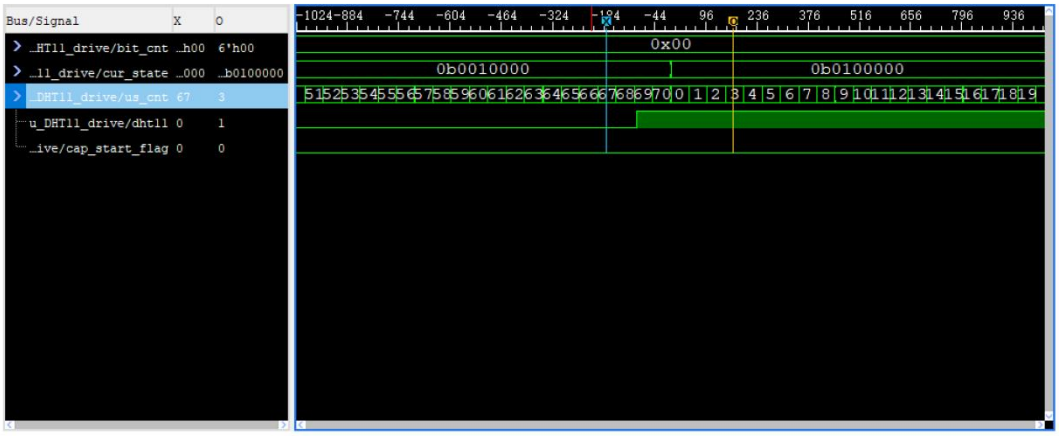


图 3-18.4-2 温湿度测量实验仿真波形图 2

由上图可知我们状态机在状态机在接收来自 DHT11 的低电平响应状态时，dht11 为低电平，并且 us_cnt 成功计数到 70 以上，满足响应时间在（70~100）us 的要求，说明来自 DHT11 的低电平响应是正确的。

同样的我们将触发条件设置为（cur_state = 7'b0100000&&us_cnt = 70）（状态机在接收来自 DHT11 的高电平响应状态）按下按键，观察捕获到的波形图如下：

图 3-18.4-3 温湿度测量实验仿真波形图 3

结合波形图可以发现状态机在接收高电平响应的状态时，us_cnt 也成功计数到 70us 以上，说明来自 DHT11 的高电平响应也是正确的。至此 DHT11 应答成功，开始发送温度数据。

我们知道，DHT11 发送的温度数据是 40bit 的 8 位湿度整数数据 +8 位湿度小数数据 +8 位温度整数数据 +8 位温度小数数据 +8 位校验和。我们将触发状态设置为（cur_state = 7'b1000000&&bit_cnt = 40）（状态机在接收来自 DHT11 的数据并且接收完成），再次按下按键，观察捕获到的波形图如下：

图 3-18.4-4 温湿度测量实验仿真波形图 4

由捕获到的波形可知，状态机在接收数据状态时，成功接收数据，并且接收 bit 计数器成功计数到 40，说明接收到了来自 DHT11 的完整 40bit 数据，至此，模块功

能验证成功。

18.5、实验现象

将程序下载进入板卡后，将 DHT11 按照约束文件正确连接，打开上位机串口助手，按下板卡按键 k0，可得到如下结果。

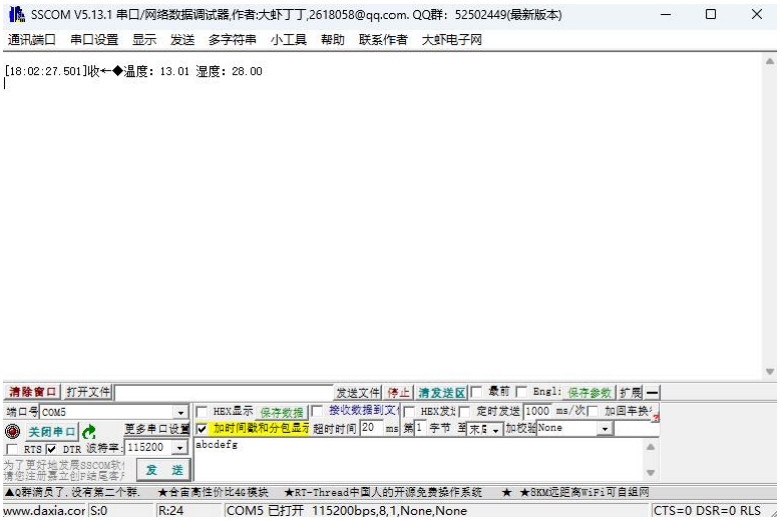


图 3-18.5-1 温湿度测量实验现象图

19、温度测量实验例程

19.1、实验目的

使用逻辑派 Z1 开发板（基于紫光同创 compa 系列 PGC4KD-6ILPG144 芯片）编写 DS18B20 温度传感器模块驱动代码，使用按键触发对环境温度的测量并使用串口向上位机发送当前测量到的温度。

19.2、实验原理

19.2.1、DS18B20 模块介绍

DS18B20 是一种数字温度传感器，支持单线通信协议，这意味着它只需要一根数据线与 FPGA 通讯仅需占用一个 I/O 端口，无须任何外部元件，直接将环境温度转化成数字信号，以数字码方式串行输出，从而大大简化了传感器与 FPGA 的接口设计。极大地简化了温度测量系统的连接复杂性。DS18B20 具有唯一的 64 位 ROM 地址，允许在单根数据线上连接多个传感器，实现多点温度测量。此外，它还支持寄生电源模式，可以在数据线供电的情况下工作，进一步简化了系统设计。由于其可靠的性能、低功耗、高精度以及易于集成的特点，DS18B20 被广泛应用于各种温度监控和数据采集系统中，如家庭自动化、工业控制和环境监测等。

本次实验使用的 DS18B20 模块图如下：

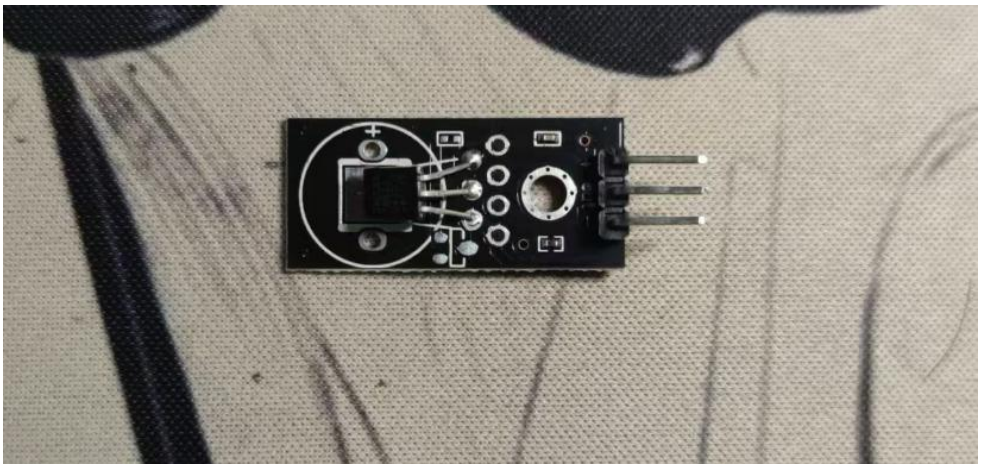


图 3- 19.2- 1 DS18B20 模块图

本次实验使用美国 DALLAS 半导体公司推出的数字化温度传感器 DS18B20 其示意图如下：

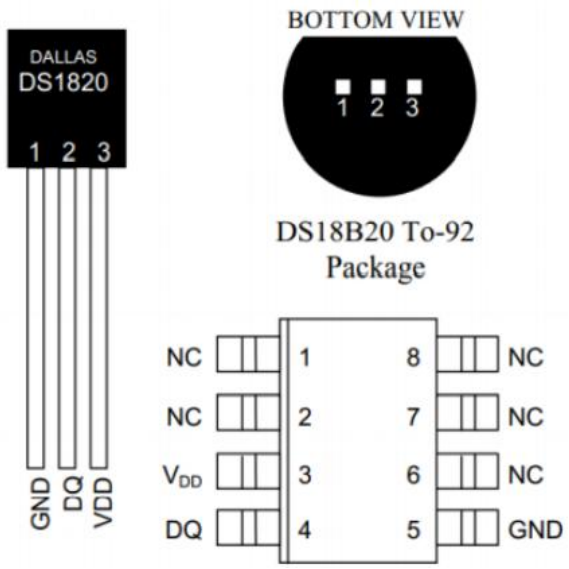


图 3-19.2-2 DS18B20 示意图

DS18B20 测温范围从-55°C 到+125°C，在-10°C 到+85°C 的范围内，精度可达±0.5°C。现场（实时）温度直接以“单总线” 的数字方式传输，大大提高了系统的抗干扰性。它能直接读出被测温度，并且可根据实际要求通过简单的编程实现 9~l2 位的数字值读数方式。

其芯片内部框图如下

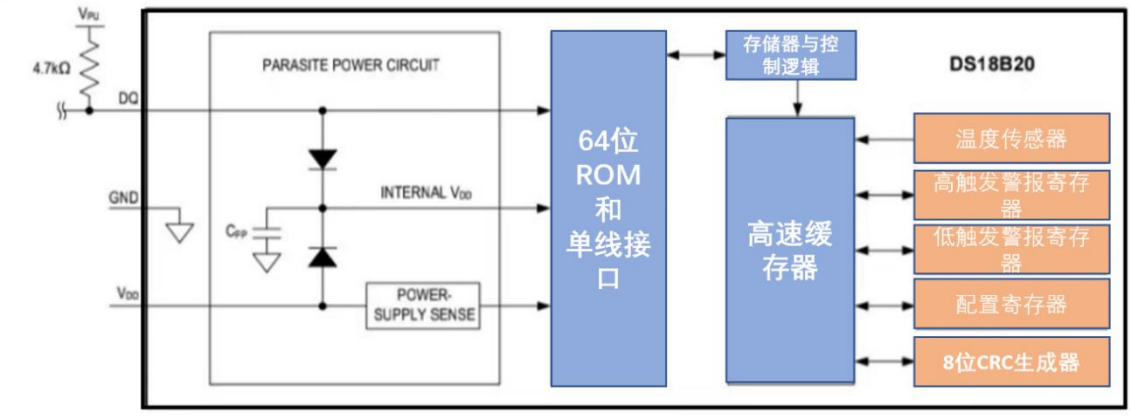


图 3-19.2-3 DS18B20 芯片内部框图

其中高速缓存器结构示意图如下：

图 3-19.2-4 高速缓存器结构示意图

DS18B20 的高速缓冲器是一个临时存储温度测量数据和配置参数的小型存储器，用于在温度转换和数据通信过程中缓存必要的信息。由上图可知缓冲器由 9 个字节组成，其中包括测量的温度数据（前两个字节）、配置寄存器（第四个字节）、以及内部保留位等。用户可通过配置寄存器设置分辨率，而其他字节中的特定位被保留用于内部用途，不可写入。在配置寄存器的 8 个比特位中，只有第 5 位（R0）与第 6 位（R1）可以由用户配置，其他位都保留给 DS18B20 内部使用。R0 与 R1 的组合决定了传感器的分辨率，以及温度转换所需的时间。它们的作用可以理解为配置选项的二进制编码开关，通过不同的值组合实现分辨率的选择。详情请看下面的图片：

R1	R0	分辨率（位）	温度精度（℃）	典型转换时间（ms）
0	0	9 位	±0.5	93.75
0	1	10 位	±0.25	187.5
1	0	11 位	±0.125	375
1	1	12 位（默认）	±0.0625	750

图 3-19.2-5 不同分辨率的选择

当用户不进行配置时，R1 和 R0 的默认值为 1，即采集到的温度信息使用 12 位分辨率。

当 DS18B20 配置为 12 位分辨率的温度数据时，（高速缓冲器的第 0、1 字节用于输出温度数据）其中输出的温度数据最高位为符号位，温度值实际占 11 位，最低 4 位表示小数部分。FPGA 读取温度数据时，将会一次读取 16 位（2 字节），从中提取低 11 位的二进制值，将其转换为十进制后乘以 0.0625 得到实际温度值。需要注意，

温度的正负由前 5 位符号位决定，这 5 位同时变化，只需判断其中任意一位即可：若符号位为 1，表示温度为负值，需对提取的二进制值取反加 1 后再乘以 0.0625；若符号位为 0，则直接乘以 0.0625。

19.2.2、DS18B20 通讯时序

1.DS18B20 总线复位时序

控制器与 DS18B20 所有的通信都是由初始化开始的，初始化由主设备发出的复位脉冲及 DS18B20 响应的存在脉冲组成。当 DS18B20 响应复位信号的后，向主设备表明其在该总线上，并且已经做好了执行命令的准备。在这个过程中，总线上的主设备需要拉低总线最少 480us 来表示发送复位脉冲。发送完之后，主设备要释放总线进入接收模式。当总线释放后，上拉电阻将总线拉至高电平。当 DS18B20 检测到该上升沿信号后，其等待 15us 至 60us 后将总线拉低 60us 至 240us 来实现发送回复。其具体时序图如下：

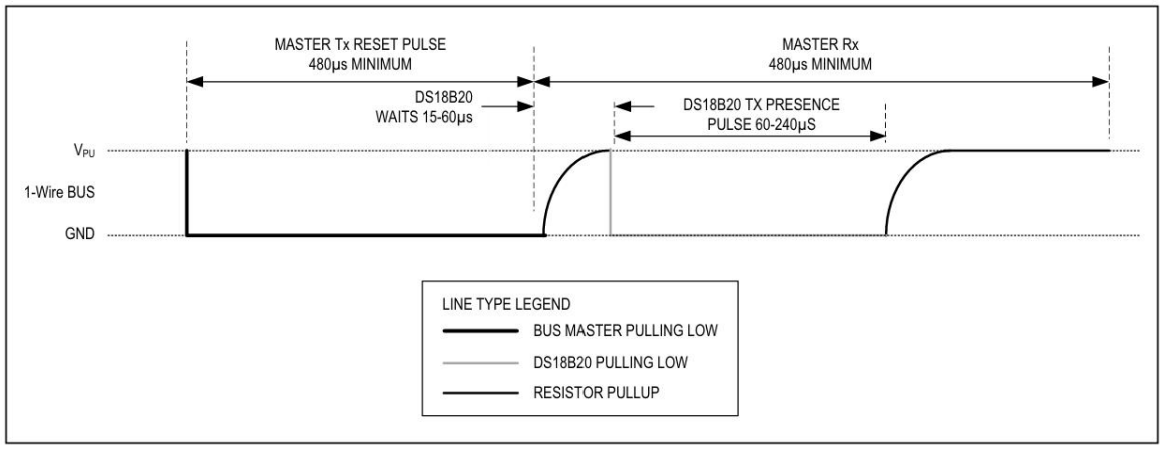


图 3- 19.2- 6 DS18B20 总线复位时序图

2.DS18B20 总线写时序

在写过程中写入数据有两种情况：写“1”和写“0”。写两种数据需要遵循不同的时序。当主设备将总线从高电平拉至低电平时，启动写操作，所有的写操作持续时间最少为 60us，每个写操作间的恢复时间最少为 1us。当总线（DQ）拉低后，DS18B20 在 15us 至 60us 之间对总线进行采样，如果采的 DQ 为高电平则发生写 1，如果为低电平则发生写 0，如下图所示（图中的总线控制器即为主设备）。如果主机要写 1，必须先将总线拉至逻辑低电平然后释放总线，允许总线在写操作开始后 15us 内上拉至高电平。若要写 0，必须将总线拉至逻辑低电平并保持不变最少 60us。

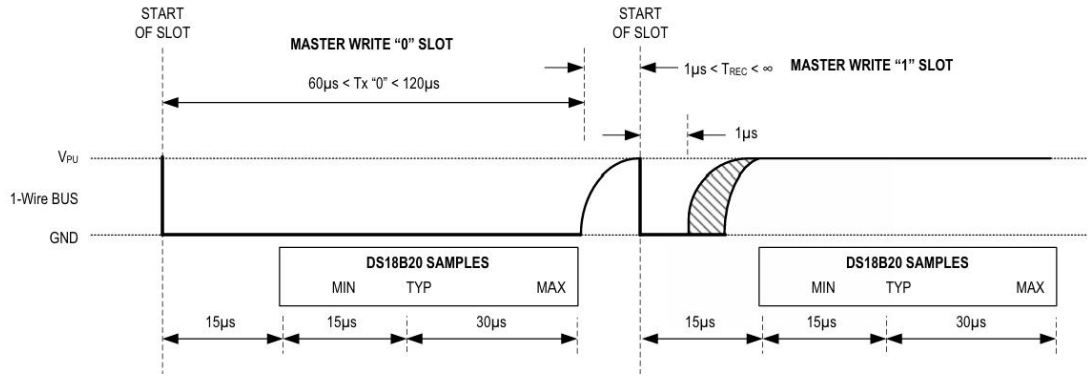


图 3-19.2-7 DS18B20 总线写时序图

3.DS18B20 总线读时序

同样的在读过程中也有有两种情况：读出“1”和读出“0”。每个读时序最小有 60us 的持续时间以及每个读时序之间有 1us 的恢复时间。当主设备将总线从高电平拉至低电平超过 1us，启动读操作。当启动读操作后，DS18B20 将会向主设备发送“0”或者“1”。DS18B20 通过将总线拉高来发送 1，将总线拉低来发送 0。当读时隙完成后，DQ 引脚将通过上拉电阻将总线拉高至高电平的闲置状态。从 DS18B20 中输出的数据在启动读时隙后的 15us 内有效，所以，主设备在读时隙开始后的 15us 内必须释放总线，并且对总线进行采样。其具体时序图如下：

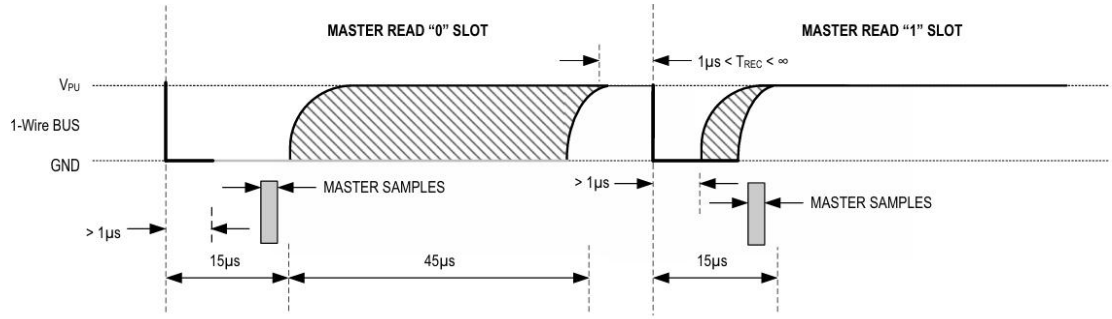


图 3-19.2-8 DS18B20 总线读时序图

19.2.3、DS18B20 通讯过程

操作 DS18B20 需要三个步骤，分别是初始化、写 ROM 命令、写功能命令。其具体解释如下：

1.初始化

总线上的所有事件都必须以初始化为开始。初始化信号由总线上的主设备发出的复位脉冲以及紧跟着从设备回应的存在脉冲构成。该存在脉冲是让总线主设备知道

DS18B20 在总线上并准备好运行。具体的时序在 19.2.2 中 DS18B20 总线复位时序小节有具体介绍。

2.写 ROM 命令

在初始化完成后,可以执行ROM命令,这些命令用于操作每个设备的64位 ROM编码，帮助主设备在总线上识别和管理多个从设备。ROM 命令共有 5 种，每种命令长度均为 8 位，具体如下：

(1)搜索 ROM [F0h]

在系统上电初始化后，主设备使用该命令识别总线上的所有从设备及其 ROM 编码，从而确定从设备的类型和数量。

(2)读 ROM [33h]

该命令允许主设备读取DS18B20的64位ROM编码,仅在总线上只有一个DS18B20时可用。若总线上存在多个从设备，使用此命令将导致所有设备同时响应，从而引起数据冲突。

(3)匹配 ROM [55h]

此命令后接 64 位 ROM 编码，用于在多点总线中定位特定的 DS18B20。只有编码与指定 ROM 完全匹配的设备会响应，其他设备将等待下一个复位脉冲。该命令可在单点或多点总线上使用。

(4)跳过 ROM [CCh]

该命令允许主设备跳过64位ROM编码直接执行下一步操作,适用于单点总线(只有一个 DS18B20）的情况（本次实验就是此种情况），可节省时间。但在多点总线中，若发送跳过 ROM 命令后执行读操作，则所有从设备将同时响应，导致数据冲突。

(5)警报搜索 [ECh]

此命令功能类似跳过 ROM，但只有温度超出报警阈值（高于 TH 或低于 TL）的从设备会响应。报警状态会持续保留，直到温度回到正常范围或掉电为止。

3.写功能命令

当主设备通过 ROM 命令确认某个 DS18B20 可以通信后，便可向目标从设备发送功能命令，以执行特定操作。以下是 DS18B20 的功能命令及其作用：

(1)温度转换 [44h]

该命令用于启动单次温度转换，完成后，转换结果会存储在高速缓存器的 byte0（温度低 8 位）和 byte1（温度高 8 位）中，随后 DS18B20 进入低功耗闲置状态。

- 若总线在命令后发出读时隙，DS18B20 会返回：
  - "0"：表示温度转换尚未完成。
  - "1"：表示温度转换已完成。
- **寄生电源模式注意事项**：在发送该命令后，必须立即强制拉高总线，且拉高时间需满足时序要求。

(2)写入暂存器 [4Eh]

此命令允许主设备向高速缓存器写入 3 个字节数据，按顺序写入以下寄存器：byte2（高温触发值）：报警触发高温值。byte3（低温触发值）：报警触发低温值。byte4（配置寄存器）：分辨率配置等参数。数据写入按低位到高位顺序进行，可通过复位随时中断写入操作。

(3)读取高速缓存器 [BEh]

该命令从高速缓存器读取数据，读取从 byte0（温度低 8 位）开始，到 byte8（CRC 校验）结束。数据从低位开始传送，读取过程可通过复位随时终止。

(4)复制高速缓存器 [48h]

将高速缓存器中的高温触发值(byte2)、低温触发值(byte3)和配置寄存器(byte4)的值复制到非易失性存储器(EEPROM)中，若命令后主机发出读请求，DS18B20 会返回：

- "0"：表示复制操作正在进行。
- "1"：表示复制完成。

需要注意的是如果使用寄生电源模式,发送该命令后，必须立即强制拉高总线至少 10ms。

(5)召回 EEPROM [B8h]

将 EEPROM 中存储的**高温触发值（byte2）**、**低温触发值（byte3）**和**配置寄存器（byte4）**的数据恢复到高速缓存器中。

- 上电后，召回操作会自动执行一次，确保缓存器中有有效数据。
- 若执行该命令后主机发出读请求，DS18B20 会返回：
  - ″0″：表示正在召回数据。
  - ″1″：表示召回完成。

(6)读取供电模式 [B4h]

该命令用于判断 DS18B20 的供电方式：返回 ″0″：表示使用**寄生电源模式**。返回 ″1″：表示使用**外部电源模式**。

19.3、代码设计

根据以上对 DS18B20 的介绍, 我们使用逻辑派 Z1 开发板对 DS18B20 进行初始化, 并写入将模块测量的环境温度解析出来,使用串口发送到上位机。

DS18B20 驱动模块端口描述如下表:

表 3-19-1 DS18B20 驱动模块端口表

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
cap_flag	input	1	触发采集信号（开始温度测量）
distance	output	8	计算得到的距离（单位厘米）
data_vld	output	1	distance 的有效标志
dq	inout	1	DS18b20 通讯总线
s_bf	output	4	温度数据的百分位

s_sf	output	4	温度数据的十分位
s_sw	output	4	温度数据的十位
s_gw	output	4	温度数据的个位
data_out	output	20	温度数据
data_symbol	output	1	温度正负指示

具体代码如下:

```
1.  module DS18B20_drive
2.  (
3.    input  wire                clk              ,
4.    input  wire                rst_n            ,
5.    input  wire                cap_flag           ,
6.    inout  wire                dq               ,
7.
8.    output reg      [  3: 0]     s_bf             ,//百分位
9.    output reg      [  3: 0]     s_sf             ,//十分位
10.   output reg      [  3: 0]     s_sw              ,//十位
11.   output reg      [  3: 0]     s_gw              ,//个位
12.   output reg      [  5: 0]     data_symbol          ,//指示温度符号
13.   output reg                   data_vld          ,//温度有效标志
14.
15.   output wire     [ 19: 0]     data_out
16. );
17.   parameter                WAIT_TRI          = 7'b0000_001,
18.                          INIT              = 7'b0000_010,
19.                          WR_CMD            = 7'b0000_100,
20.                          WAIT              = 7'b0001_000,
21.                          INIT_AGAIN        = 7'b0010_000,
22.                          RD_CMD            = 7'b0100_000,
23.                          RD_TEMP           = 7'b1000_000;
24.
25.   parameter                WAIT_MAX          = 20'd750_000;
26.
27.   parameter                WR_CC_44          = 16'h44_cc,
28.                          WR_CC_BE          = 16'hbe_cc;
29.
30.   reg                      clk_us            ;
31.   reg             [  5: 0]   cnt             ;
32.   reg             [  6: 0]   state           ;
33.   reg             [ 19: 0]   us_cnt            ;
34.   reg                      flag              ;
35.   reg             [  3: 0]   bit_cnt           ;
36.   reg             [ 15: 0]   data_temp           ;
37.   reg             [ 19: 0]   data              ;
```

```
38.    reg                         dq_en                ;
39.    reg                         dq_out               ;
40.
41.    reg             [ 19: 0]    temperature          ;
42.
43.
44.    assign                      dq                   = (dq_en == 1'b1) ? dq_out : 1'bz;
45.    assign                      data_out             = temperature;
46.
47.
48.
49.
50.
51.
52.    // 脉冲扩展
53.    reg             [ 6: 0]     counter              ;
54.    reg                         cap_start            ;
55.    always @(posedge clk or negedge rst_n) begin
56.      if (!rst_n) begin
57.          cap_start <= 1'b0;
58.          counter <= 0;
59.      end else begin
60.          if (cap_flag) begin
61.            cap_start <= 1'b1;
62.            counter <= 200;
63.          end else if (counter > 0) begin
64.            cap_start <= 1'b1;                    // 计数器递减，维持高电平
65.            counter <= counter 1;
66.          end else begin
67.            cap_start <= 1'b0;                    // 计数结束，输出恢复低电平
68.          end
69.      end
70.    end
71.
72.  //cnt，计数 50 次，实现 1us
73.  always@(posedge clk or negedge rst_n)
74.    if(rst_n == 1'b0)
75.      cnt <= 6'd0;
76.    else   if(cnt == 6'd49)
77.      cnt <= 6'd0;
78.    else
79.      cnt <= cnt + 1'b1;
80.  //clk_us,1us 的时钟
81.  always@(posedge clk or negedge rst_n)
82.    if(rst_n == 1'b0)
83.      clk_us <= 1'b0;
84.    else   if(cnt == 6'd49)
85.      clk_us <= ~clk_us;
86.    else
```

```
87.       clk_us <= clk_us;
88.   //us_cnt,1us 时钟计数器，用于状态跳转
89.   always@(posedge clk_us or negedge rst_n)
90.     if(rst_n == 1'b0)
91.       us_cnt <= 20'd0;
92.     else   if(((state == INIT || state == INIT_AGAIN) && (us_cnt == 20'd999))
93.               || ((state == WR_CMD || state == RD_CMD || state == RD_TEMP) && (us_cn
t == 20'd64))
94.               || ((state == WAIT && us_cnt == WAIT_MAX)))
95.       us_cnt <= 20'd0;
96.     else
97.       us_cnt <= us_cnt + 1'b1;
98.   //bit_cnt,bit 计数器，写 1bit 或读 1bit 加 1，一次写完之后清零
99.   always@(posedge clk_us or negedge rst_n)
100.    if(rst_n == 1'b0)
101.      bit_cnt <= 4'd0;
102.    else   if((state == WR_CMD || state == RD_CMD || state == RD_TEMP)
103.              && (bit_cnt == 4'd15) && (us_cnt == 20'd64))
104.      bit_cnt <= 4'd0;
105.    else   if((state == WR_CMD || state == RD_CMD || state == RD_TEMP)
106.              && (us_cnt == 20'd64))
107.      bit_cnt <= bit_cnt + 1'b1;
108. //flag
109. always@(posedge clk_us or negedge rst_n)
110.    if(rst_n == 1'b0)
111.      flag <= 1'b0;
112.    else   if((state == INIT || state == INIT  AGAIN) && (us  cnt == 20'd570) && (dq ==
1'b0))
113.      flag <= 1'b1;
114.    //else   if((state == INIT || state == INIT_AGAIN) && (us_cnt == 20'd999))
115.    else   if(us  cnt == 20'd999)
116.      flag <= 1'b0;
117.    else
118.      flag <= flag;
119. always@(posedge clk  us or negedge rst  n)
120.    if(rst_n == 1'b0)
121.      state <= WAIT  TRI;
122.    else
123.      case(state)
124.          WAIT_TRI   :
125.            if(cap  start)
126.              state <= INIT;
127.            else
128.              state <= WAIT_TRI;
129.          INIT      :
130.            if(us_cnt == 20'd999 && flag == 1'b1)
131.              state <= WR_CMD;
132.            else
133.              state <= INIT;
```

```
134.            WR_CMD     :
135.              if(bit_cnt == 4'd15 && us_cnt == 20'd64)
136.                state <= WAIT;
137.              else
138.                state <= WR_CMD;
139.            WAIT       :
140.              if(us_cnt == WAIT_MAX)
141.                state <= INIT_AGAIN;
142.              else
143.                state <= WAIT;
144.            INIT_AGAIN :
145.              if(us_cnt == 20'd999 && flag == 1'b1)
146.                state <= RD_CMD;
147.              else
148.                state <= INIT_AGAIN;
149.            RD_CMD     :
150.              if(bit_cnt == 4'd15 && us_cnt == 20'd64)
151.                state <= RD_TEMP;
152.              else
153.                state <= RD_CMD;
154.            RD_TEMP    :
155.              if(bit_cnt == 4'd15 && us_cnt == 20'd64)
156.                state <= WAIT_TRI;
157.              else
158.                state <= RD_TEMP;
159.            default:state <= WAIT_TRI;
160.        endcase
161. always@(posedge clk_us or negedge rst_n)
162.     if(rst_n == 1'b0)
163.       begin
164.       dq_en  <= 1'b0;
165.       dq_out <= 1'b0;
166.       end
167.     else
168.       case(state)
169.            WAIT_TRI      :
170.              begin
171.                dq_en  <= 1'b1;
172.                dq_out <= 1'b1;
173.              end
174.            INIT       :
175.              if(us_cnt < 20'd499)
176.                begin
177.                    dq_en  <= 1'b1;
178.                    dq_out <= 1'b0;
179.                end
180.              else
181.                begin
182.                    dq_en  <= 1'b0;
```

```
183.                dq  out <= 1'b0;
184.            end
185.        WR  CMD     :
186.            if(us_cnt > 20'd62)
187.            begin
188.                dq_en  <= 1'b0;
189.                dq  out <= 1'b0;
190.            end
191.            else    if(us  cnt <= 20'd1)
192.            begin
193.                dq  en  <= 1'b1;
194.                dq_out <= 1'b0;
195.            end
196.            else    if(WR_CC_44[bit_cnt] == 1'b0)
197.            begin
198.                dq_en  <= 1'b1;
199.                dq_out <= 1'b0;
200.            end
201.            else    if(WR_CC_44[bit_cnt] == 1'b1)
202.            begin
203.                dq_en  <= 1'b0;
204.                dq_out <= 1'b0;
205.            end
206.        WAIT      :
207.            begin
208.            dq_en  <= 1'b1;
209.            dq_out <= 1'b1;
210.            end
211.        INIT_AGAIN :
212.            if(us_cnt < 20'd499)
213.            begin
214.                dq_en  <= 1'b1;
215.                dq_out <= 1'b0;
216.            end
217.            else
218.            begin
219.                dq_en  <= 1'b0;
220.                dq_out <= 1'b0;
221.            end
222.        RD_CMD     :
223.            if(us_cnt > 20'd62)
224.            begin
225.                dq_en  <= 1'b0;
226.                dq_out <= 1'b0;
227.            end
228.            else    if(us_cnt <= 20'd1)
229.            begin
230.                dq_en  <= 1'b1;
231.                dq_out <= 1'b0;
```

```
232.                end
233.              else  if(WR_CC_BE[bit_cnt] == 1'b0)
234.                begin
235.                    dq_en <= 1'b1;
236.                    dq_out <= 1'b0;
237.                end
238.              else  if(WR_CC_BE[bit_cnt] == 1'b1)
239.                begin
240.                    dq_en <= 1'b0;
241.                    dq_out <= 1'b0;
242.                end
243.            RD_TEMP   :
244.              if(us_cnt <= 1)
245.                begin
246.                    dq_en <= 1'b1;
247.                    dq_out <= 1'b0;
248.                end
249.              else
250.                begin
251.                    dq_en <= 1'b0;
252.                    dq_out <= 1'b0;
253.                end
254.            default:
255.              begin
256.                dq_en <= 1'b0;
257.                dq_out <= 1'b0;
258.              end
259.        endcase
260.
261.
262. //data_temp
263. always@(posedge clk_us or negedge rst_n)
264.    if(rst_n == 1'b0)
265.        data_temp <= 16'b0;
266.    else  if((state == RD_TEMP) && (us_cnt == 20'd13))
267.        data_temp <= {dq,data_temp[15:1]};
268.    else
269.        data_temp <= data_temp;
270. //data
271. always@(posedge clk_us or negedge rst_n)
272.    if(rst_n == 1'b0)begin
273.            data <= 20'd0;
274.            data_symbol <= 6'd10;
275.    end
276.
277.    else  if((state == RD_TEMP) && (bit_cnt == 4'd15) && (us_cnt == 20'd60)
278.              && (data_temp[15] == 1'b0))begin              //如果是正数直接输出
279.                data <= data  temp[10:0];
280.                data_symbol <= 6'd10;                      //ASCII "+"
```

```
281.            end
282.    else    if((state == RD_TEMP) && (bit_cnt == 4'd15) && (us_cnt == 20'd60)
283.            && (data_temp[15] == 1'b1))begin                //如果是负数
284.              data <= ~data_temp[10:0] + 1;
285.              data_symbol <= 6'd11;                         //ASCII "-"
286.            end
287.
288.    reg                     rev_done            ;
289.    reg                     rev_done_dl         ;
290.    wire                    rev_done_fall       ;
291.    assign                  rev_done_fall       = rev_done_dl&~rev_done;
292. always@(posedge clk_us or negedge rst_n)
293.    if(rst_n == 1'b0)
294.      rev_done <= 1'b0;
295.    else if((state == RD_TEMP) && (bit_cnt == 4'd15) && (us_cnt == 20'd60))
296.      rev_done <= 1'b1;
297.    else
298.      rev_done <= 1'b0;
299.
300. always@(posedge clk or negedge rst_n)
301.    if(rst_n == 1'b0)
302.      rev_done_dl <= 1'b0;
303.    else
304.      rev_done_dl <= rev_done;
305.
306. always@(posedge clk_us or negedge rst_n)
307.    if(rst_n == 1'b0)
308.      temperature <= 20'b0;
309.    else if(rev_done == 1'b1)
310.      temperature <= (data * 625) / 100;
311.
312.
313. always @(posedge clk or negedge rst_n) begin
314.    if (rst_n == 1'b0) begin
315.      s_bf <= 0;
316.      s_sf <= 0;
317.      s_sw <= 0;
318.      data_vld<= 1'b0;
319.    end else begin
320.      s_bf <= temperature % 10;
321.      s_sf <= (temperature / 10) % 10;
322.      s_gw <= (temperature / 100) % 10;
323.      s_sw <= (temperature / 1000) % 10;
324.      data_vld<= rev_done_fall;
325.    end
326. end
327. endmodule
```

在这个模块我们执行的操作如下：等待顶层触发信号、向 DS18B20 发送初始化信号、向 DS18B20 发送跳过 ROM 命令[CCh]和温度转换命令[44h]、等待 DS18B20 温度转换完成、再次向 DS18B20 发送初始化信号、向 DS18B20 发送跳过 ROM 命令[CCh]和温度读取命令[BEh]、解析温度数据并重新回到初始状态等待顶层模块下一次触发温度采样过程。

因为我们使用的是默认配置，所以温度采样分辨率为 12 位，温度转换时间为 750ms。模块核心逻辑是一个三段式状态机，我们通过设计这个状态机来处理顶层触发信号与 DS18B20 的交互。状态机总共有七个状态：DS18B20 驱动模块的状态机共有 7 个状态，每个状态的功能如下：

1. WAIT_TRI: 在此状态下，等待顶层模块的触发信号 cap_flag 启动信号的到来。如果接收到启动信号，状态将转移到 INIT。

2. INIT: 初始化状态。在该状态中，按照总线时序发送发送初始化信号。如果初始化信号发送成功，且接收到 DS18B20 的应答，状态机将转移到 WR_CMD 状态。

3. WR_CMD: 写命令状态。在此状态中，发送发送跳过 ROM 命令[CCh]和温度转换命令[44h]。发送完成后，状态机将转移到 WAIT 状态，等待 DS18B20 采集温度数据完成。

4. WAIT: 等待状态。在此状态下，进行 750ms 的等待。DS18B20 温度转换完成，当等待完成后状态机将转移到 INIT_AGAIN 状态。

5. INIT_AGAIN: 重新初始化状态。发送初始化命令，如果初始化信号发送成功，且接收到 DS18B20 的应答，状态机将转移到 RD_CMD 状态。

6. RD_CMD: 读命令状态。在此状态下，向 DS18B20 发送跳过 ROM 命令[CCh]和温度读取命令[BEh]。读取命令发送完成后，状态机将转移到 RD_TEMP 状态。

7. RD_TEMP: 读取温度状态。在此状态中，处理 DS18B20 返回的温度数据。处理完成后，状态机将重新回到等待启动状态(WAIT_TRI)。

分析这个模块需要清楚的理解 DS18B20 的操作过程，若读者对 DS18B20 的操作时序不太清楚，请务必返回理论部分进行学习。

DS18B20 实验例程顶层模块端口描述如下表：

表 3-19-2 DS18B20 实验例程顶层模块端口表

端口	I/O	位宽	描述
clk	input	1	50Mhz 时钟
rst_n	input	1	复位信号
key_in	input	1	通过按键触发一次采样
dq	inout	1	DS18B20 通讯总线
uart_txd	output	1	串口发送端口

模块代码如下：

```
1.  module DS18B20_top(
2.      input                   clk             ,
3.      input                   rst_n           ,
4.
5.
6.      input                   key_in          ,
7.      inout                   dq              ,
8.      output                  uart_txd
9.  );
10.
11.
12.     parameter               CNT_MAX         = 20'd999_999; //消抖计数器
13.     parameter STR = 128'hb5_b1_c7_b0_bb_b7_be_b3_ce_c2_b6_c8_ce_aa_a3_ba; // "
当前环境温度为：" 的 GB2312 编码
14.
15.
16.
17.  //信号定义
18.     wire                    data_vld        ;
19.     wire        [ 31: 0]    temperature_data    ;
20.     wire                    uart_tx_busy        ;
21.     wire                    uart_tx_done        ;
22.
23.     reg         [  7: 0]    uart_tx_data        ;
24.     reg                     uart_tx_req         ;
25.     reg                     work_en             ;
26.     reg         [  6: 0]    tx_byte_cnt         ;
27.
28.     reg         [ 19: 0]    cnt_20ms            ;//消抖计数器
29.     reg                     key_flag            ;
30.     reg         [  7: 0]    ascii_table     [0:11] ;//储存阿拉伯数字 0~9 与符号+-的
ASCII 值
31.
```

```
32.     wire            [  3: 0]      s_bf                ;//百分位
33.     wire            [  3: 0]      s_sf                ;//十分位
34.     wire            [  3: 0]      s_sw                 ;//十位
35.     wire            [  3: 0]      s_gw                 ;//个位
36.     wire [5:0] data_symbol;
37.  //cnt_20ms:如果时钟的上升沿检测到外部按键输入的值为低电平时，计数器开始计数
38.  always@(posedge clk or negedge rst  n)
39.    if(rst_n == 1'b0)
40.      cnt  20ms <= 20'b0;
41.    else   if(key_in == 1'b1)
42.      cnt  20ms <= 20'b0;
43.    else   if(cnt_20ms == CNT_MAX && key_in == 1'b0)
44.      cnt  20ms <= cnt  20ms;
45.    else
46.      cnt  20ms <= cnt  20ms + 1'b1;
47.
48.
49.
50.
51.
52.
53.  always@(posedge clk or negedge rst_n)
54.    if(rst_n == 1'b0)
55.      key_flag <= 1'b0;
56.    else   if(cnt_20ms == CNT_MAX 1'b1)
57.      key_flag <= 1'b1;
58.    else
59.      key_flag <= 1'b0;
60.
61.
62.
63.
64.  always@(posedge clk or negedge rst_n)
65.    if(rst_n == 1'b0)
66.      work_en <= 1'b0;
67.    else   if(data_vld == 1'b1)
68.      work_en <= 1'b1;
69.    else   if(tx_byte_cnt== 7'd23&&uart_tx_done)
70.      work_en <= 1'b0;
71.    else
72.      work_en <= work_en;
73.
74.  //串口发送逻辑
75.  always @(posedge clk or negedge rst  n) begin
76.    if (!rst_n) begin
77.      tx  byte  cnt <= 7'b0;
78.    end else if (work_en) begin
79.      if (tx_byte_cnt == 7'd23&&uart_tx_done)              // 总共发送 22 字节
```

```
            tx_byte_cnt <= 7'b0;
        else if(uart_tx_done)
            tx_byte_cnt <= tx_byte_cnt + 1'b1;
    end
end


// 串口发送数据控制

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        uart_tx_req <= 1'b0;
        uart_tx_data <= 8'b0;
        ascii_table[0] = 8'd48;                    //'0'
        ascii_table[1] = 8'd49;                    //'1'
        ascii_table[2] = 8'd50;                    //'2'
        ascii_table[3] = 8'd51;                    //'3'
        ascii_table[4] = 8'd52;                    //'4'
        ascii_table[5] = 8'd53;                    //'5'
        ascii_table[6] = 8'd54;                    //'6'
        ascii_table[7] = 8'd55;                    //'7'
        ascii_table[8] = 8'd56;                    //'8'
        ascii_table[9] = 8'd57;                    //'9'
        ascii_table[10] = 8'd43;                    //'+'
        ascii_table[11] = 8'd45;                    //'-'
    end else if (work_en && !uart_tx_busy) begin
        case (tx_byte_cnt)                        // 根据当前字节计数器发送数据
            7'd1:  uart_tx_data <= STR[127:120];        // "当"
            7'd2:  uart_tx_data <= STR[119:112];        // "当"
            7'd3: uart_tx_data <= STR[111:104];        // "前"
            7'd4: uart_tx_data <= STR[103:96];        // "前"
            7'd5: uart_tx_data <= STR[95:88];        // "环"
            7'd6: uart_tx_data <= STR[87:80];        // "环"
            7'd7: uart_tx_data <= STR[79:72];        // "境"
            7'd8: uart_tx_data <= STR[71:64];        // "境"
            7'd9: uart_tx_data <= STR[63:56];        // "温"
            7'd10: uart_tx_data <= STR[55:48];        // "温"
            7'd11: uart_tx_data <= STR[47:40];        // "度"
            7'd12: uart_tx_data <= STR[39:32];        // "度"
            7'd13: uart_tx_data <= STR[31:24];        // "为"
            7'd14: uart_tx_data <= STR[23:16];        // "为"
            7'd15: uart_tx_data <= STR[15:8];        // "："
            7'd16: uart_tx_data <= STR[7:0];        // "："
            7'd17: uart_tx_data <= ascii_table[data_symbol];      // 温度符号
            7'd18: uart_tx_data <= ascii_table[s_sw];          // 温度十位
            7'd19: uart_tx_data <= ascii_table[s_gw];          // 温度个位
            7'd20: uart_tx_data <= 8'd46;//小数点
```

```
129.            7'd21: uart_tx_data <= ascii_table[s_sf];              // 温度十分位
130.            7'd22: uart_tx_data <= ascii_table[s_bf];              // 温度百分位
131.            7'd23: uart_tx_data <= 8'h0A;                          // 换行符
132.            default: uart  tx  data <= 8'b0;
133.       endcase
134.       uart_tx_req <= 1'b1;                                        // 拉高请求信号
135.    end else begin
136.       uart_tx_req <= 1'b0;                                        // 请求信号拉高仅一个时钟周期
137.    end
138. end
139. DS18B20_drive u_DS18B20_drive(
140.    .clk                    (clk                ),
141.    .rst_n                  (rst_n              ),
142.    .cap_flag               (key_flag           ),
143.    .dq                     (dq                 ),
144.    .s_bf                   (s_bf               ), //百分位
145.    .s_sf                   (s_sf               ), //十分位
146.    .s_sw                   (s_sw               ), //十位
147.    .s_gw                   (s_gw               ), //个位
148.    .data_symbol            (data_symbol        ), // 指示温度符号
149.    .data_vld               (data_vld           ), // 温度有效标志
150.    .data_out               (data_out           )
151. );
152.
153. uart_tx u_uart_tx(
154.    .clk                    (clk                ),
155.    .rst_n                  (rst_n              ),
156.    .uart_tx_req            (uart_tx_req&work_en    ),// 发送使能请求
157.    .uart_tx_data           (uart_tx_data       ),// UART 要发送的数据
158.    .uart_txd               (uart_txd           ),// UART 发送端口
159.    .uart_tx_done           (uart_tx_done       ),//8bit 数据发送完成
160.    .uart_tx_busy           (uart_tx_busy       ) // 发送忙
161. );
```

我们在顶层模块接收用户输入的按键信号 key_in 使用一个 20ms 计数器对其进行消抖产生 DS18B20 驱动模块的采样触发信号 cap_flag。当 data_vld 有效时，将 DS18B20 模块解析得到温度数据十位、个位、十分位、百分位通过串口发送到上位机进行查看，汉字使用 GB2312 编码，模块中初始化了一个 ascii_table 用来储存阿拉伯数字 0~9 和符号“+ -”的 ASCII 值，将 DS18B20 采样得到的温湿度数据在 ascii_table 索引到对应的 ASCII 值进行发送。其中串口发送部分的知识在之前的章节已有介绍，读者若对这部分不熟悉，可到串口回环章节进行查看。

本实验约束代码如下：

```
define_attribute {p:dq} {PAP_IO_DIRECTION} {INOUT}
define_attribute {p:dq} {PAP_IO_LOC} {92}
define_attribute {p:dq} {PAP_IO_VCCIO} {1.2}
define_attribute {p:dq} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:dq} {PAP_IO_DRIVE} {2}
define_attribute {p:dq} {PAP_IO_NONE} {TRUE}
define_attribute {p:dq} {PAP_IO_SLEW} {SLOW}
define_attribute {p:uart_txd} {PAP_IO_DIRECTION} {OUTPUT}
define_attribute {p:uart_txd} {PAP_IO_LOC} {33}
define_attribute {p:uart_txd} {PAP_IO_VCCIO} {1.2}
define_attribute {p:uart_txd} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:uart_txd} {PAP_IO_DRIVE} {2}
define_attribute {p:uart_txd} {PAP_IO_NONE} {TRUE}
define_attribute {p:uart_txd} {PAP_IO_SLEW} {SLOW}
define_attribute {p:clk} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:clk} {PAP_IO_LOC} {5}
define_attribute {p:clk} {PAP_IO_VCCIO} {1.2}
define_attribute {p:clk} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:clk} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:key_in} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:key_in} {PAP_IO_LOC} {20}
define_attribute {p:key_in} {PAP_IO_VCCIO} {1.2}
define_attribute {p:key_in} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:key_in} {PAP_IO_PULLUP} {TRUE}
define_attribute {p:rst_n} {PAP_IO_DIRECTION} {INPUT}
define_attribute {p:rst_n} {PAP_IO_LOC} {19}
define_attribute {p:rst_n} {PAP_IO_VCCIO} {1.2}
define_attribute {p:rst_n} {PAP_IO_STANDARD} {LVCMOS12}
define_attribute {p:rst_n} {PAP_IO_PULLUP} {TRUE}
```

19.4、代码仿真

由于本实验需要与 DS18B20 通过总线交互信息，在测试文件中编写 DS18B20 的应答逻辑比较麻烦，我们可以将 DS18B20 驱动模块中的关键信号打上 debug 标记，分析 DS18B20 驱动模块是否正确工作，同时检查 DS18B20 是否正确应答我们通过总线发送的信息。

我们将顶层模块发送拉低采样触发信号 cap_flag、总线信号 dq、温度符号指示 data_symbol、温度有效标志 data_vld、温度数据 data_out、状态机信号 state、微秒计数器 us_cnt、接收数据比特计数器 bit_cnt 添加上 debug 标记进行分析。

首先我们设置触发模式为 cap_flag = 1，按下按键捕获到波形，观察状态机是否正确响应触发信号。波形如下图所示，状态机正确跳转到初始化状态，对 DS18B20 进行初始化。

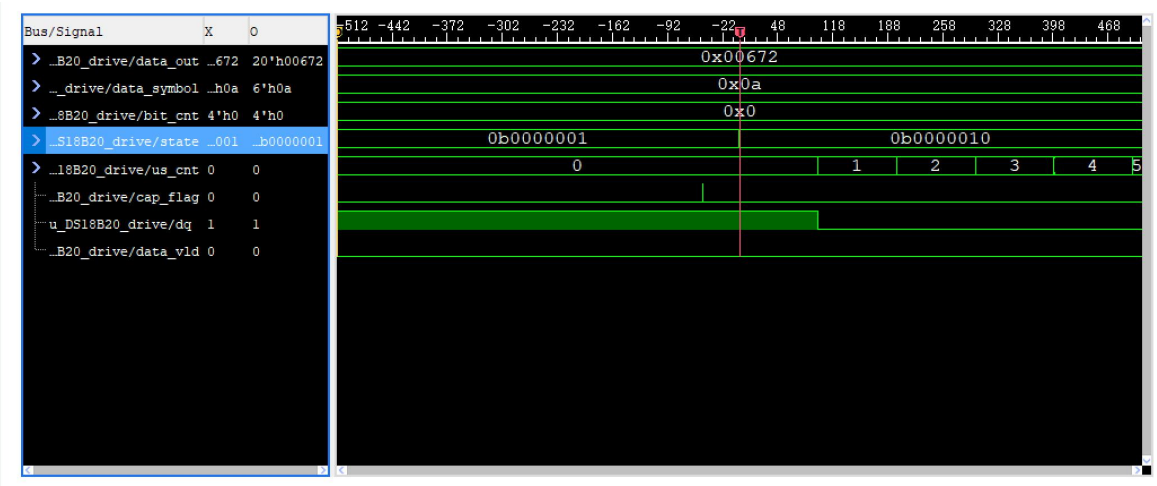


图 3-19.4-1 温度测量实验波形图 1

然后设置触发模式为（state = 7'b0000_010&&us_cnt = 959）（状态机处于 DS18B20 初始化状态，接收响应完成），DS18B20 总线复位时序中我们有说明，一整个复位操作包括主机发送复位与从机应答总共 960us，我们以此作为条件观察复位是否成功，按下按键观察捕获到的波形如下：

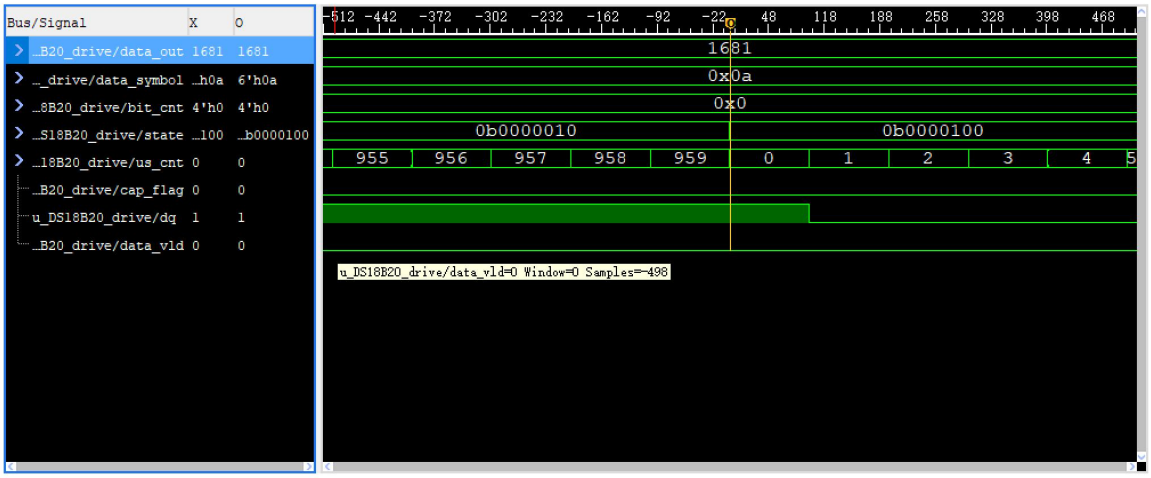


图 3-19.4-2 温度测量实验波形图 2

由上图可知当 us_cnt = 959 时，状态机跳转到下一状态，同时总线拉低，符合我们的复位时序。

接下来将触发模式设置为（state = 7'b0000_100&&bit_cnt = 15&&us_cnt =64）（状态机发送 16bit 命令状态，并且发送完成），捕获到的波形图如下图：

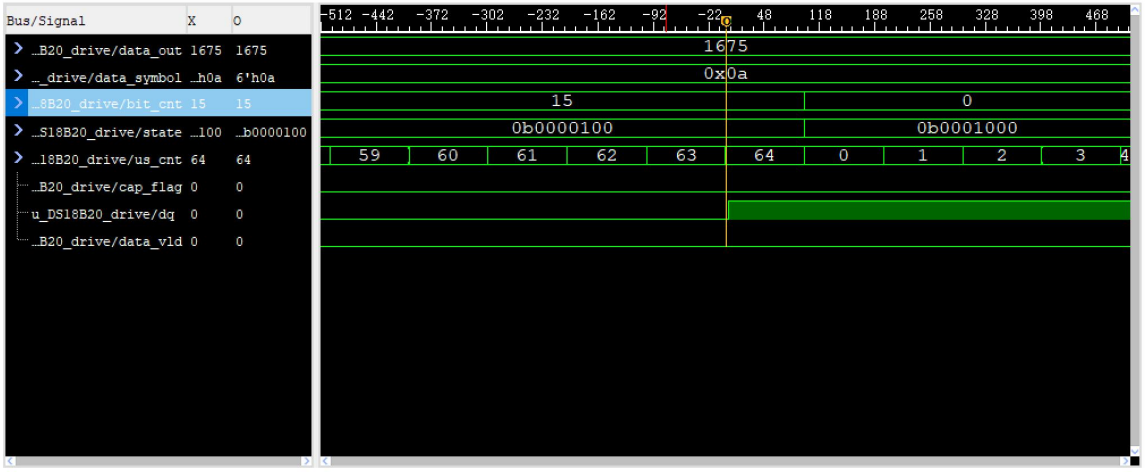


图 3-19.4-3 温度测量实验波形图 3

由上图可知在发送完成 16bit 命令[CCh][44h]之后状态机跳转到等待状态，等待 DS18B20 进行温度转换，共等待 750ms，此时总线拉高。由此可见状态跳转逻辑正确。接下来就是再次进行初始化，发送命令[CCh][BEh]，状态机跳转到接收温度数据状态，开始读数据。我们将触发模式设置为（state = 7'b1000_000&&bit_cnt = 15&&us_cnt =64）（状态机接收温度数据，并且接收完成），捕获到的波形图如下：

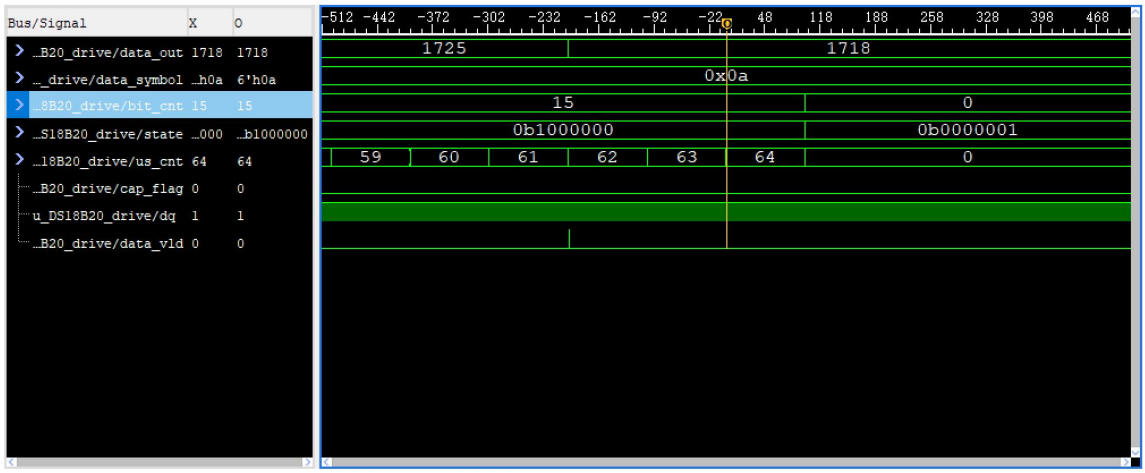


图 3-19.4-4 温度测量实验波形图 4

在这张图中，状态机接收数据完成，并计算出当前温度为 17.18 度，同时 data_vld 信号拉高，表示当前温度数据解析完成，数据有效。之后状态机跳转到等待触发状态，等待顶层触发信号再次到来。由此可见，模块功能正确。

19.5、实验现象

将 DS18B20 模块按照约束文件与板卡进行连接后，将程序下载到开发板，使用 USB

转 TTL 的转换器连接开发板与电脑，打开串口助手，按下开发板按键 k0，等待温度采样完成之后，开发板将会返回当前环境温度的测量值。具体现象如下：

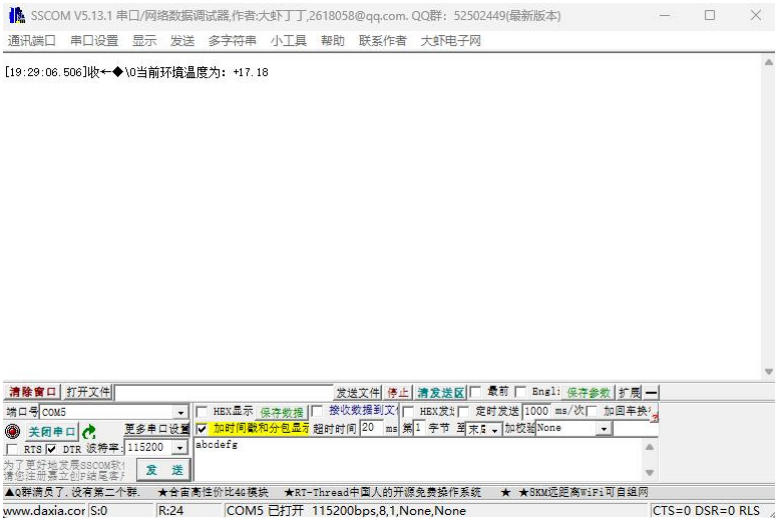


图 3-19.5-1 温度测量实验结果

第四部分、常见问题说明

1、综合阶段提示管脚冲突

1.1、问题描述

在 PDS 里综合阶段，提示逻辑派板卡上引出的 IO，44、45、70、71、125、126 处于 config mode，不能将其分配给我们的设计端口。报错信息如下：

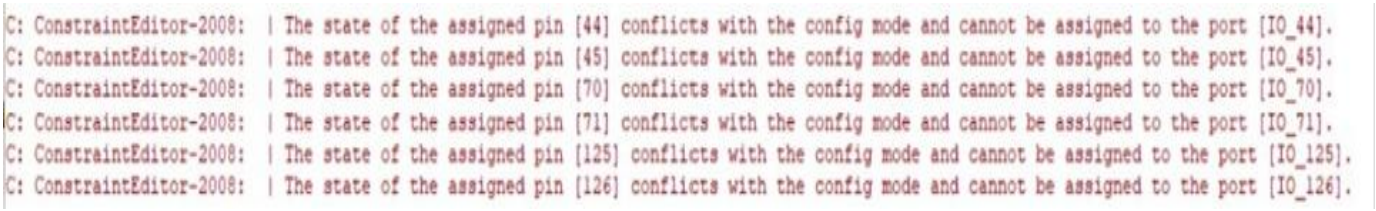


图 4 - 1.1- 1 报错信息示意图

1.2、解决方法

逻辑派 Z1 开发板基于紫光同创 Compact 系列 CPLD：PGC4RD-6LPG144。它是基于 SRAM 的可编程逻辑器件，芯片内部包含一块嵌入式 Flash，可实现位流从芯片内部自主加载。由于 CPLD 器件掉电后 SRAM 中的配置信息将会丢失，因此每次上电需要重新对 CPLD 进行配置。

配置（Configuration）是把用户设计的位流写入 CPLD 内部的过程，而整个配置过程是由配置控制系统（CCS）来完成的，其结构图如下图所示，其中用户逻辑通过 APB 总线连接 7 个从设备。位流可以由芯片主动从外部 SPI Flash 或嵌入式 Flash 中获取，也可通过外部处理器/控制器等主设备将位流下载到芯片中。

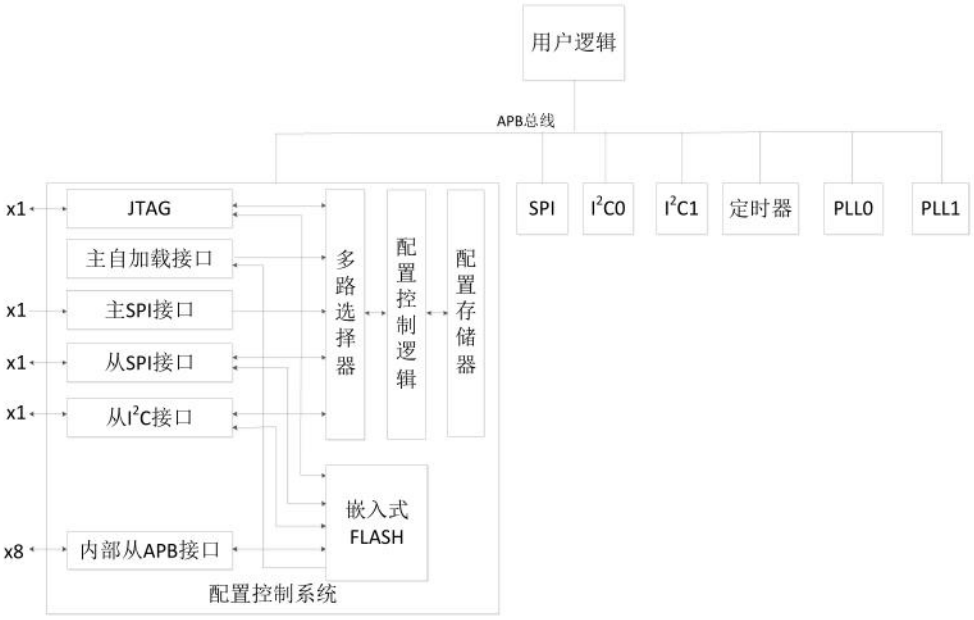


图 4 - 1.2- 1 配置系统结构图

CPLD 器件有 5 种配置模式，具体情况如下所示：

1、JTAG 模式，数据位宽 X1，符合 IEEE 1149.1 和 IEEE 1532 标准，支持配置和重配置普通/压缩位流，支持回读普通/压缩位流

2、主自加载模式，支持配置普通位流、压缩位流

3、主 SPI 模式，数据位宽 X1，支持配置普通位流、压缩位流

4、从 SPI 模式，数据位宽 X1，支持配置和重配置普通位流、压缩位流

5、从 I2C 模式，数据位宽 X1，支持配置和重配置普通位流、压缩位流

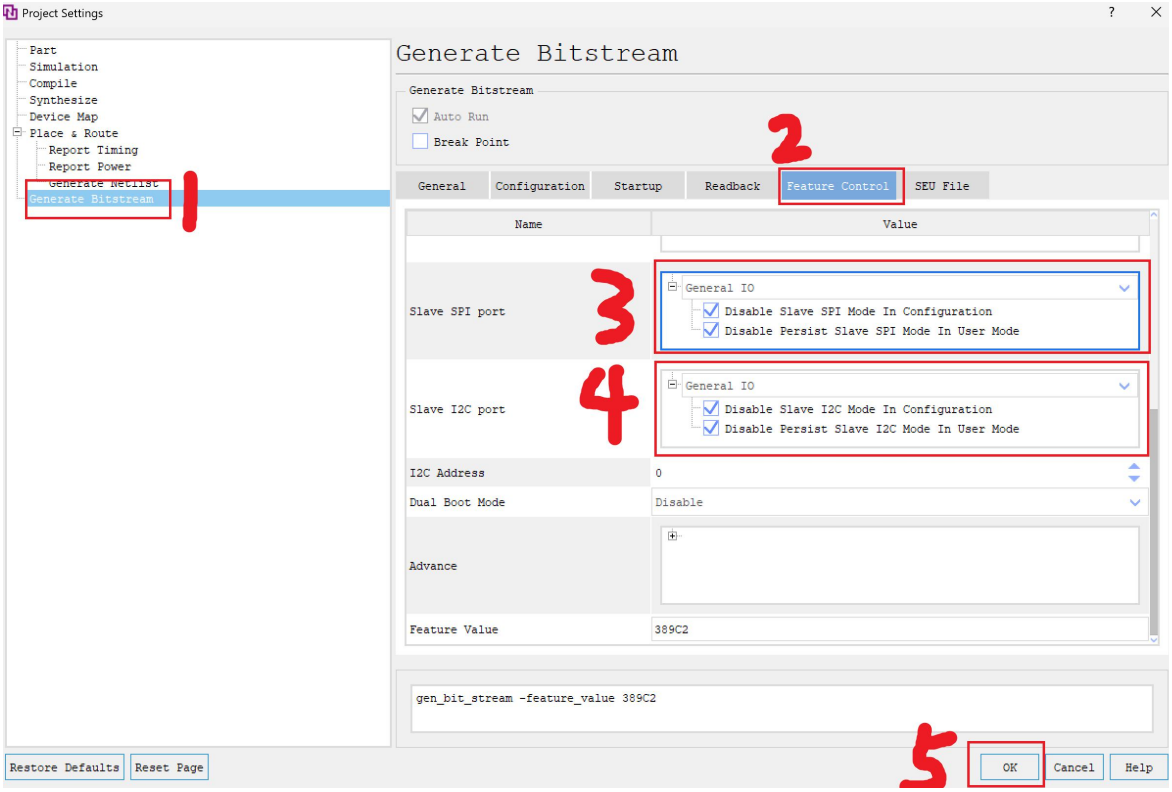
逻辑派 Z1 开发板用户 io 44、45 号管脚为从 SPI 配置接口；70、71 号管脚为主 SPI 配置接口；125、126 号接口为从 I2C 配置接口。它们作为配置接口的同时也可作为普通用户 io 使用。我们在使用 PDS 进行开发设计时，默认处于配置模式下。此时上述管脚处于配置模式下，若此时用户将其绑定到设计文件的输入输出端口上，PDS 在编译综合时会提示以下警告信息：（以 44 号管脚为例）

C: ConstraintEditor-2008: | The state of the assigned pin [44] conflicts with the config mode and cannot be assigned to the port [led_out[7]].

该警告提示我们 44 号引脚处于配置模式下，不能将其连接到指定的端口。如果我们没有配置需求，想要将其作为普通输入输出端口使用。我们需要进行以下

设置：

打开 PDS，选择一个自己开发的工程，在左上角点击 Project -> Project Setting，按照下图进行配置：



4-1.2-2 设置示意图。

经过如上配置后我们将 SPI 与 I2C 接口均作为普通 io 使用。此时点击 OK，保存配置，在自己的设计中重新编译综合即可将这些复用 io 接口作为普通 io 使用了。如需了解更多详细信息请参见 UG030004_Compact 系列 CPLD 配置（Configuration）用户指南。