PGX-Lite 7K 开发板

实验指导

版本

修改日期	修改版本	修改原因
2024/02/20	v1.0	创建文档

Myminieye 微信公众号二维码如下图，欢迎扫码关注，我们将会不定期的发送一些 FPGA 相关的技术推文或者半导体行业时讯的推文：

目录

1. 控制 LED 灯 6
1.1 实验目的 6
1.2 实验要求 6
1.3 实验原理 6
1.4 实验源码设计（完整源码查看 DEMO 源文件） 7
1.4.1 文件头设计 7
1.4.2 设计 module 8
1.4.3 完整的 Module（不含注释） 10
1.4.4 硬件管脚分配 10
1.5 实验步骤 11
1.5.1 打开 PDS 软件，创建工程 12
1.5.2 添加设计文件 16
1.5.3 编译 22
1.5.4 工程约束 22
1.6 生成位流文件并下载 25
1.7 实验现象 25

2. LED 流水灯 26
2.1 实验目的 26
2.2 实验要求 26
2.3 实验原理 26
2.4 实验源码设计（完整源码查看 DEMO 源文件） 27
2.5 实验步骤 28
2.6 实验现象 28

3. 键控彩灯 29
3.1 实验目的 29
3.2 实验要求 29
3.3 实验原理 29
3.3.1 按键控制模块功能 30
3.3.2 按键消抖 30
3.3.3 LED 控制模块功能 32
3.4 实验源码设计（完整源码查看 DEMO 源文件） 33
3.4.1 顶层文件源码 33
3.4.2 按键控制模块 34
3.4.3 LED 控制模块 36
3.5 实验现象 36

4. 数码管动态显示 37
4.1 实验目的 37
4.2 实验要求 37
4.3 实验原理 37
4.4 实验源码（完整源码查看 DEMO 源文件） 40

4.4.1 顶层模块.....40
4.4.2 按键消抖模块.....46
4.4.3 按键计数模块.....47
4.4.4 时钟分频模块.....48
4.4.5 数码管显示模块.....48
4.5 实验现象.....49
5. 序列检测器.....50
5.1 实验目的.....50
5.2 实验要求.....50
5.3 实验原理.....50
5.4 实验源码（完整源码查看 DEMO 源文件）.....50
5.4.1 方案设计.....50
5.4.2 顶层模块（含数码管显示模块）设计.....51
5.4.3 按键 LED 控制模块.....54
5.4.4 序列检测模块设计.....57
5.5 实验现象.....58
6. 密码锁.....59
6.1 实验目的.....59
6.2 实验要求.....59
6.3 实验原理.....59
6.4 实验源码（完整源码查看 DEMO 源文件）.....59
6.4.1 顶层模块设计.....60
6.4.2 按键控制设计.....62
6.4.3 按键消抖设计.....64
6.4.4 对比模块设计.....64
6.4.5 显示模块设计.....65
6.5 实验现象.....65
7. 数字钟.....66
7.1 实验目的.....66
7.2 实验要求.....66
7.3 实验原理.....66
7.4 实验源码（完整源码查看 DEMO 源文件）.....67
7.4.1 顶层设计.....67
7.4.2 数字时钟产生与控制模块设计.....71
7.4.3 时钟分频模块.....78
7.4.4 数码管显示模块设计.....79
7.5 实验现象.....80
8. 串口收发实验.....81
8.1 实验目的.....81
8.2 实验要求.....81
8.3 实验原理.....81
8.3.1 串口原理.....81

8.4 串口传输步骤.....83
8.4.1 串口发送流程.....83
8.4.2 串口接收流程.....83
8.4.3 串口发送字符.....83
8.5 实验源码设计.....84
8.5.1 串口发射模块设计.....85
8.5.2 串口接收模块设计.....90
8.5.3 串口发射控制模块设计.....95
8.5.4 串口实验顶层模块设计.....97
8.6 实验现象.....100

1.控制 LED 灯

1.1实验目的

实现对多 LED 灯的控制;

1.2实验要求

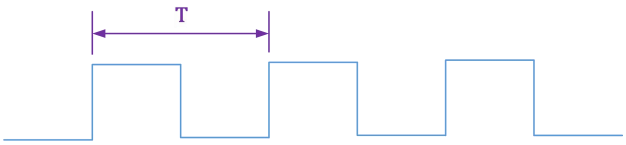
控制 8 个 LED 以 1s 的周期闪烁（0.5s 亮，0.5s 灭）

1.3实验原理

通常的时，分，秒的计时进位大家应该不陌生；

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

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）。



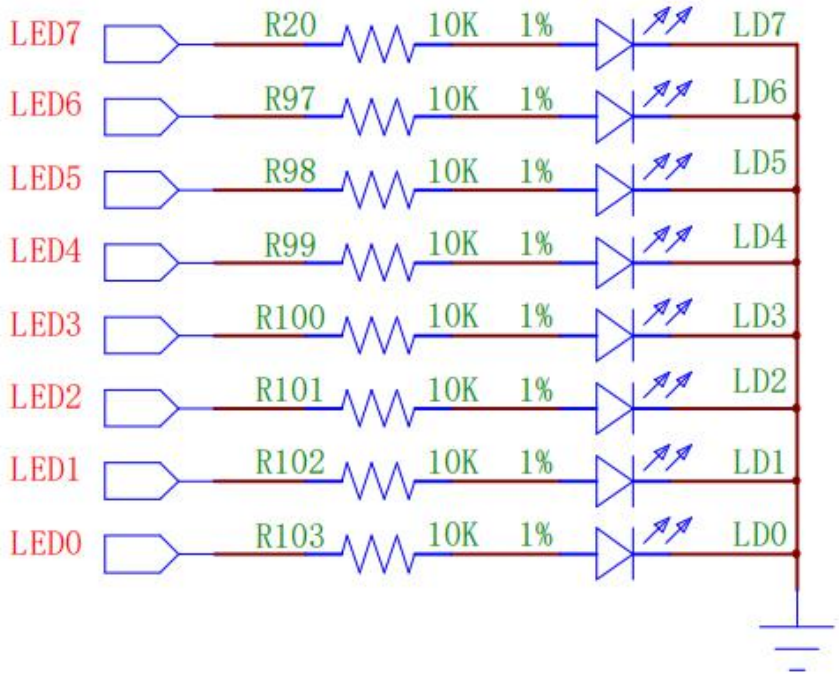
在数字系统中通常关注到时钟的频率,那频率与周期的关系如下：

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

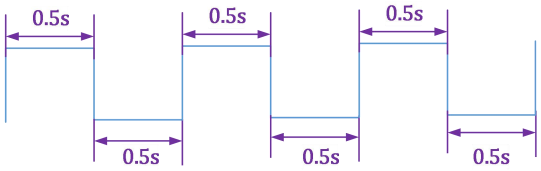
PGX-Lite 7K 板卡上有一个 50MHz 的晶振提供时钟给到 PGC7KD;

实验分析:

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



控制 LED 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:



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（完整源码查看 demo 源文件）

1.4.1文件头设计

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

改描述）等信息；以及仿真时间单位定义；

```
//////////////////////////////////////////////////////////////////////////////////
// Company:Meyesemi
// Engineer:
//
// Create Date:
// Design Name:
// Module Name:
// Project Name:
// Target Devices: Pango
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 1.0 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
/////////////
```

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

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

1.4.2设计 module

1.4.2.1 创建 module，确定输入输出信号

```
1  module led_light(
2    input                clk,        // input clock, the frequency is 50MHz
3    input          rstn,       // input reset signal, active at low
4    output [7:0] led          // output LED control signal , lignting at high
5  );
6  endmodule
```

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

创建 module 时需要定义输入输出信号;本实验输入时钟和复位即可，输出是控制 LED 的亮灭，PGX-Lite 7K 板卡上共有 8 个 LED，故而输出 8bit 位宽的信

号；

1.4.2.2设计一个计数器；

单个状态计数25000000，1个亮灭周期的计数即为50000000 = 26’h2FAF080;所以计数器的位宽为26位即可，此处请结合数字电路中的同步计数器的工作原理分析；

```
reg [25:0] led_light_cnt;
//   time counter
always @(posedge clk) // 触发条件；posedge 为上升沿，negedge 为下降沿
begin
      if(!rstn)
            led_light_cnt <= `UD 26'd0;
      else if(led_light_cnt == 26'd24_999_999)
               led_light_cnt <= `UD 26'd0;
          else
               led_light_cnt <= `UD led_light_cnt + 26'd1;
   end
```

当计数器计数到26’d24999999时，计数周期包含了从0～26’d24999999的时钟周期，故而总时长时26’d25000000 × T_{clk}；硬件输入时钟为50MHz，所以此计数器的技术周期是0.5s；

1.4.2.3led 显示状态控制

在指定的时间刻度上对LED的状态进行变更，以达到控制LED规律的亮灭的目的；

led_light_cnt的计时周期为0.5s，故在led_light_cnt上取一个点来变更LED的显示状态即可完成每隔0.5s LED显示发生变化；由于LED亮和灭只有两个状态，在赋值处理上将寄存器取反即可得到对应的从亮到灭变化（或从灭到亮的变化）；

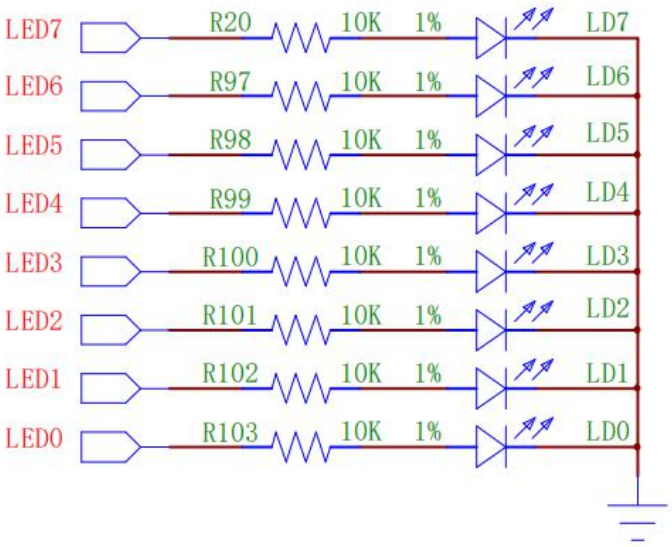
```
reg [7:0] led_status;

always @(posedge clk)
begin
      if(!rstn)
            led_status <= `UD 8'd0;
      else if(led_light_cnt == 26'd24_999_999)
            led_status <= `UD ~led_status;
end
assign led = led_status;
```

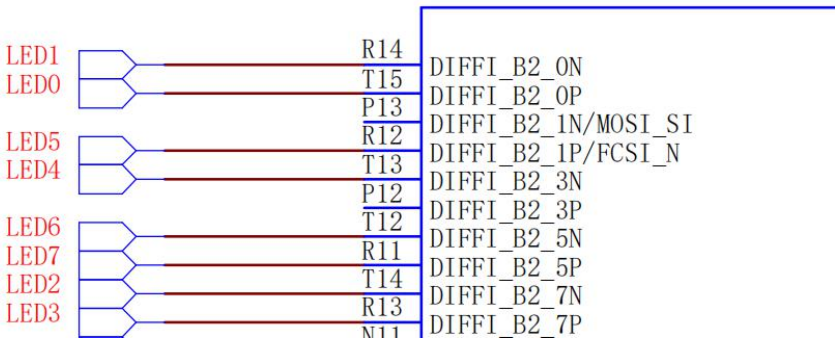
1.4.3完整的 Module（不含注释）

```
`timescale 1ns / 1ps
`define UD #1
module led_light(
    input                clk,
    input                rstn,
    output [7:0]       led
);
//=============================================================
    //reg and wire
    reg [25:0] led_light_cnt;
    reg [ 7:0] led_status;
    //   time counter
    always @(posedge clk)
    begin
        if(!rstn)
            led_light_cnt <= `UD 26'd0;
        else if(led_light_cnt == 26'd24_999_999)
            led_light_cnt <= `UD 26'd0;
        else
            led_light_cnt <= `UD led_light_cnt + 26'd1;
    end

    // led status change
    always @(posedge clk)
    begin
        if(!rstn)
            led_status <= `UD 8'd0;
        else if(led_light_cnt == 26'd24_999_999)
            led_status <= `UD ~led_status;
    end
    assign led = led_status;

endmodule
```

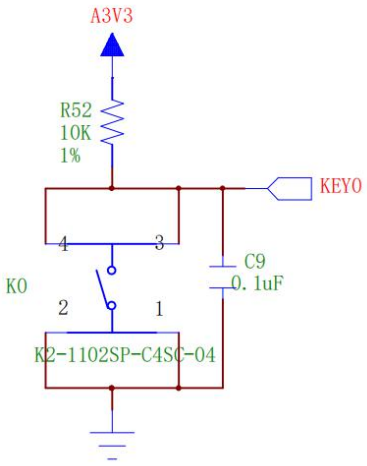
1.4.4硬件管脚分配

PGX-Lite 7K 的 LED 和 CLK 与 FPGA 的 IO 连接部分的原理图如下（在工程中做物理约束时需要结合原理图的 FPGA 管脚分配进行约束）：





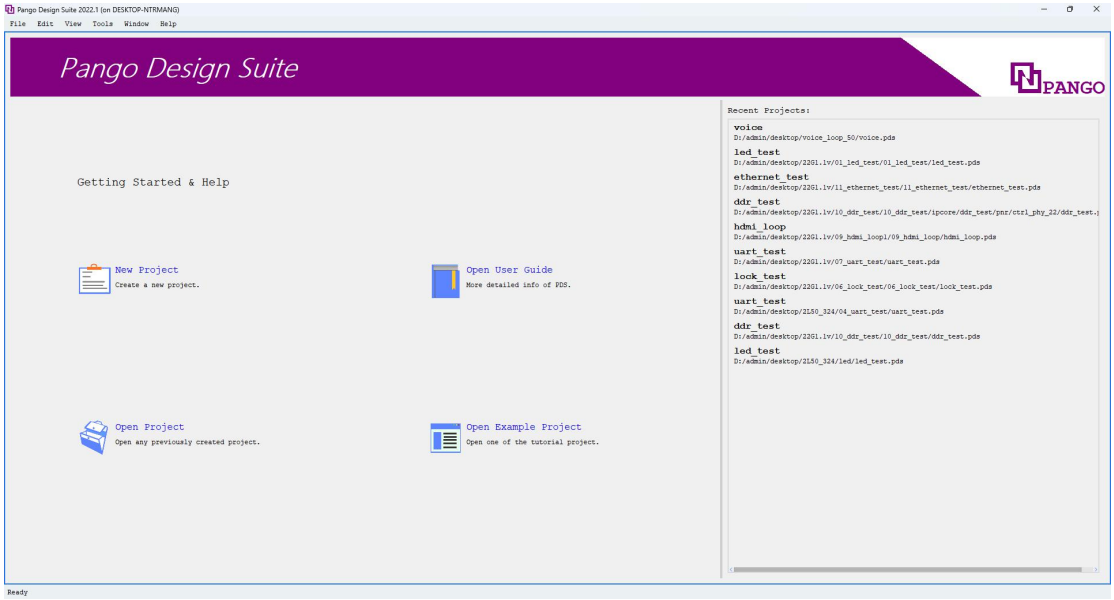
复位设计是低电平有效，而 PGX-Lite 7K 板卡上的按键按下时为低电平，松开为高电平，可用按键输入来做复位信号；



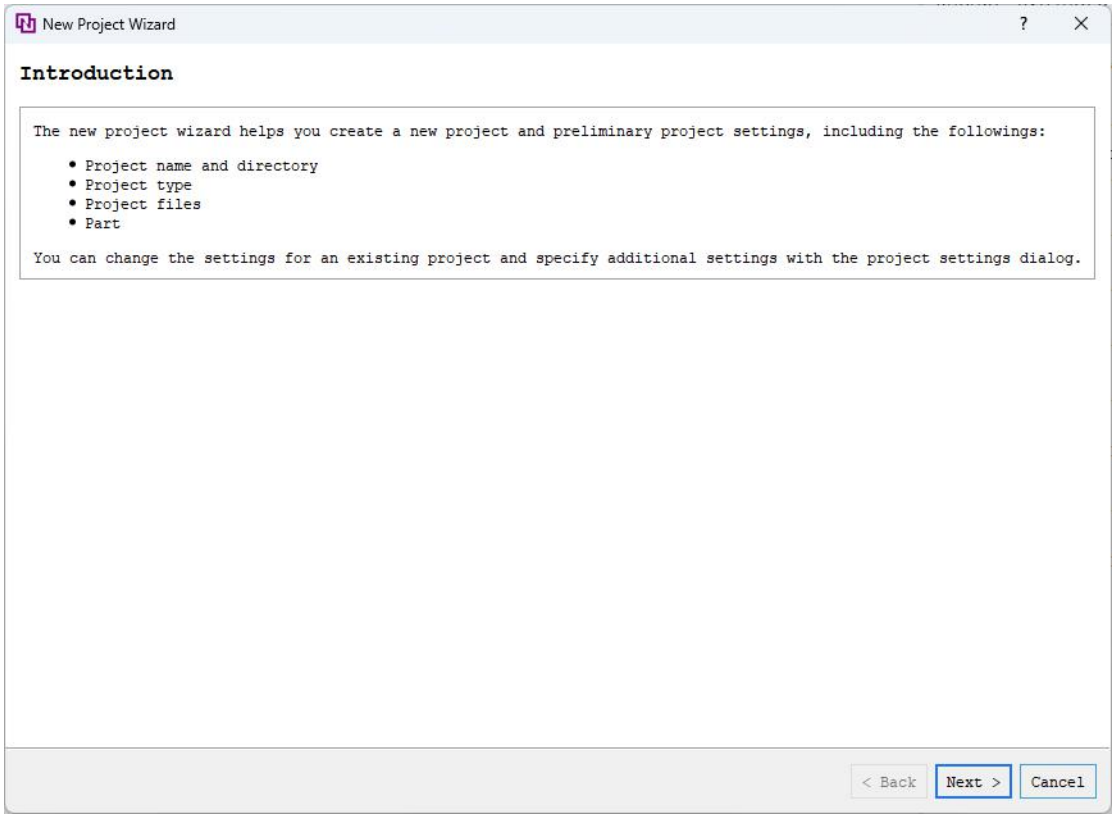
1.5实验步骤

1.5.1打开 PDS 软件，创建工程

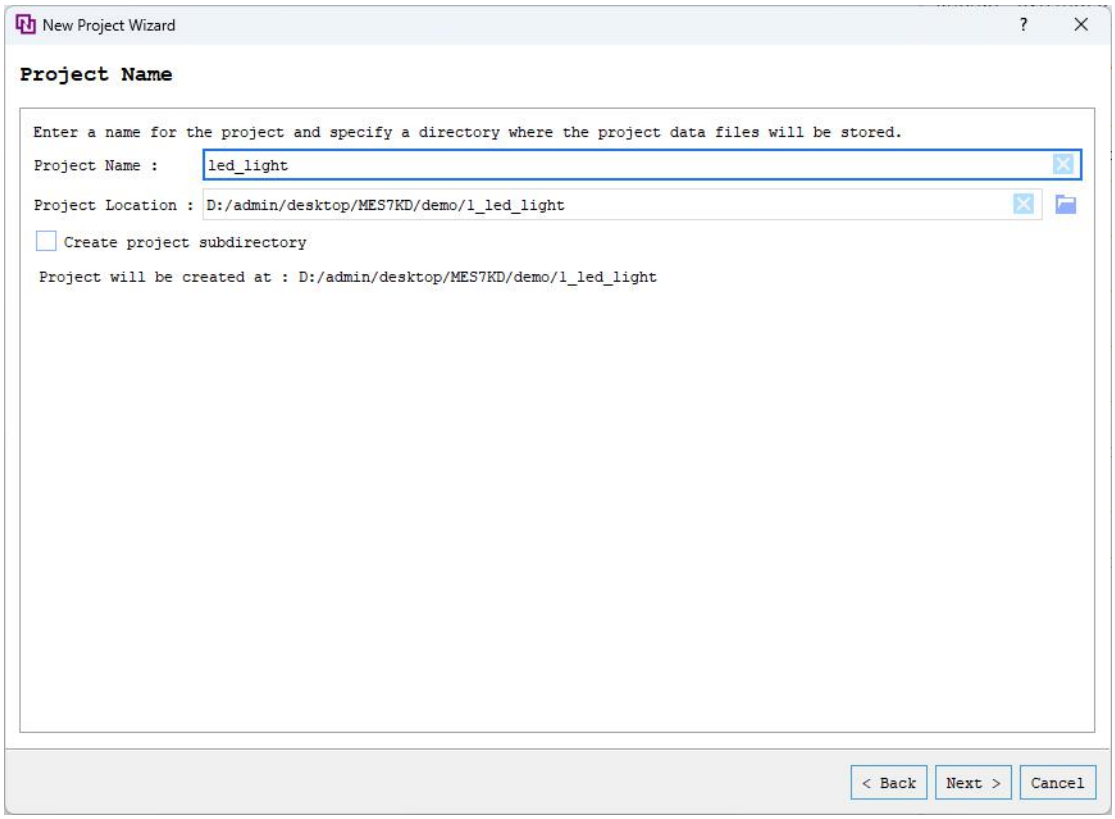
Step1：打开 PDS 软件，单击 New Project



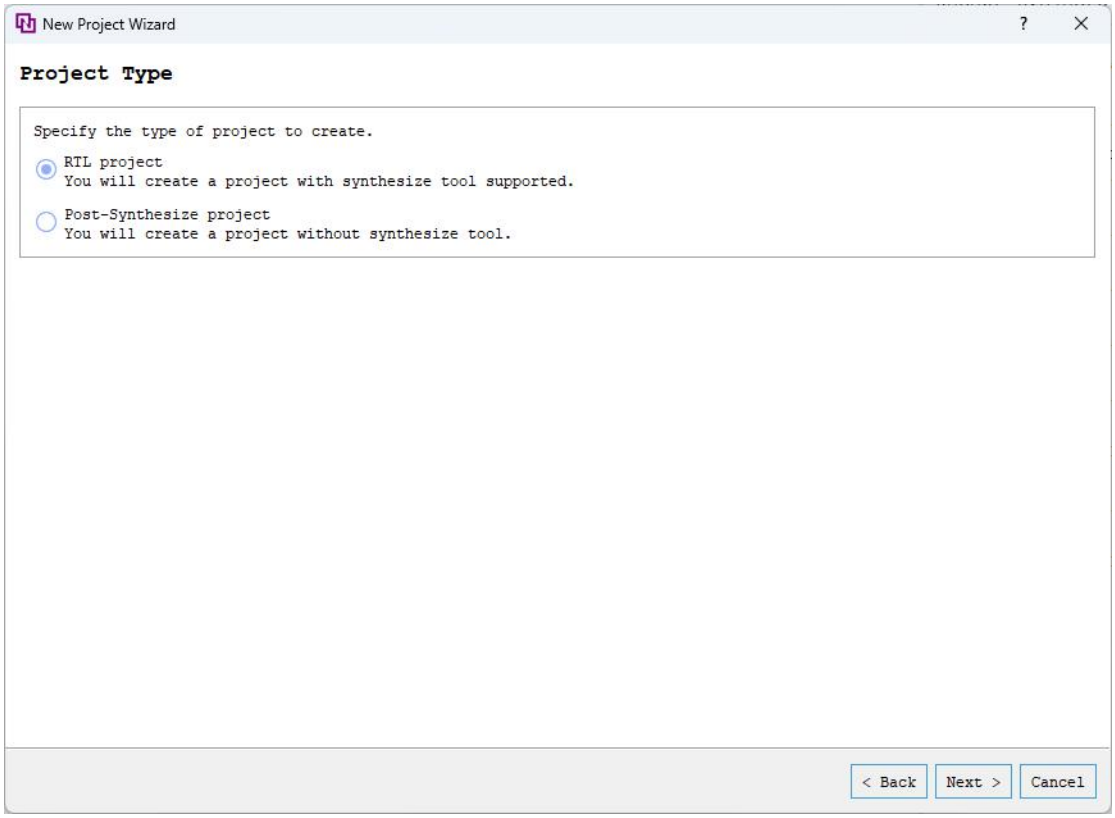
Step2：单击 NEXT



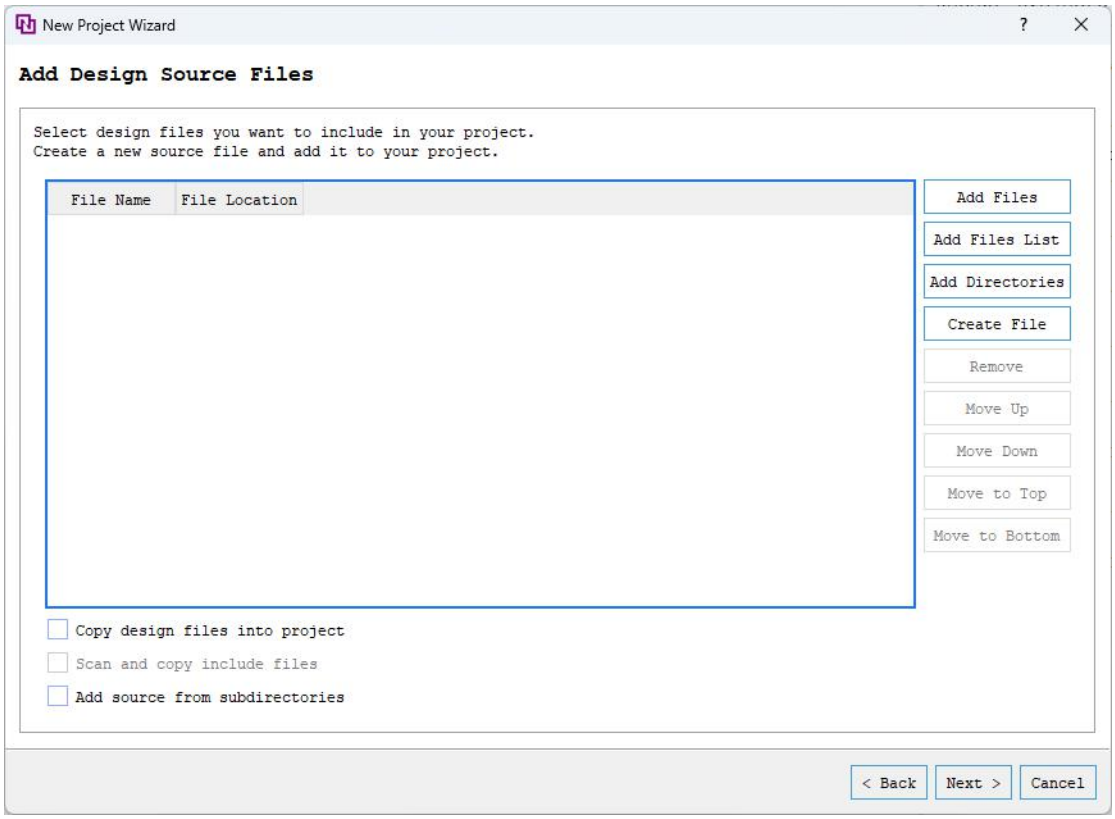
Step3：创建名为 led_light 的工程到对应的文件目录，之后单击 Next



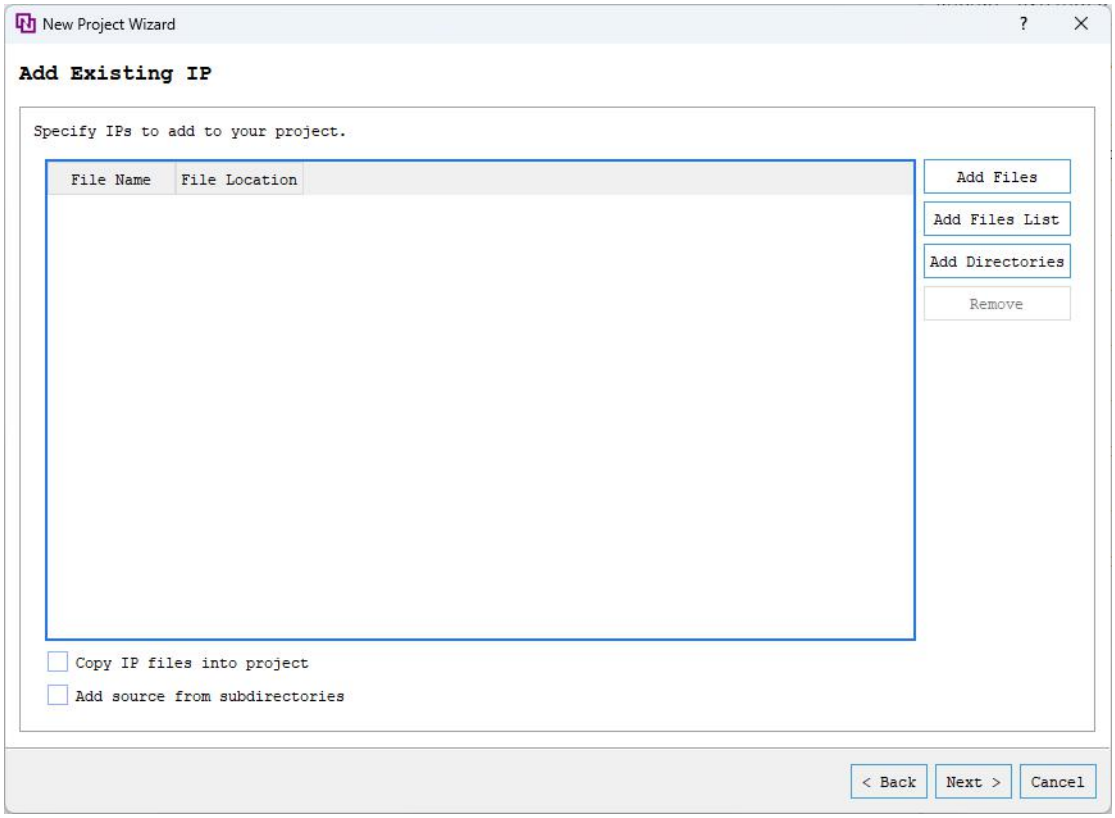
Step4：选择 RTL project，单击 Next



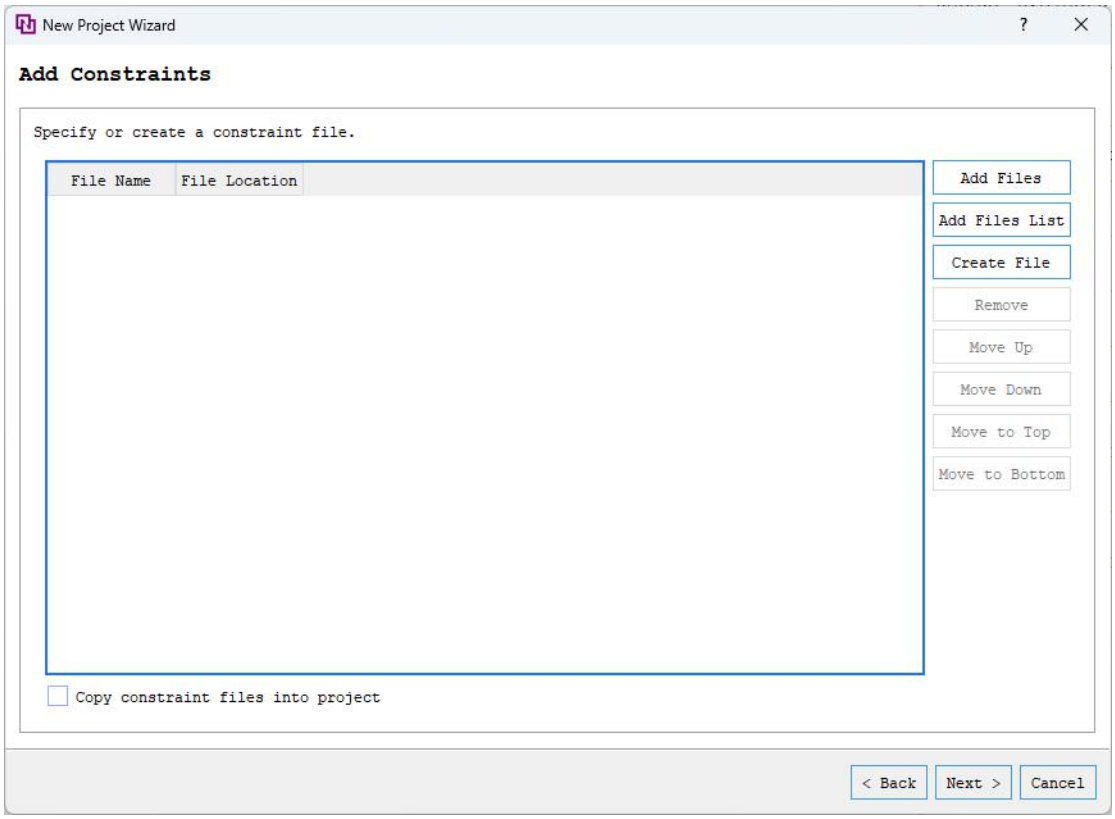
Step5：单击 Next(也可添加.v 文件)



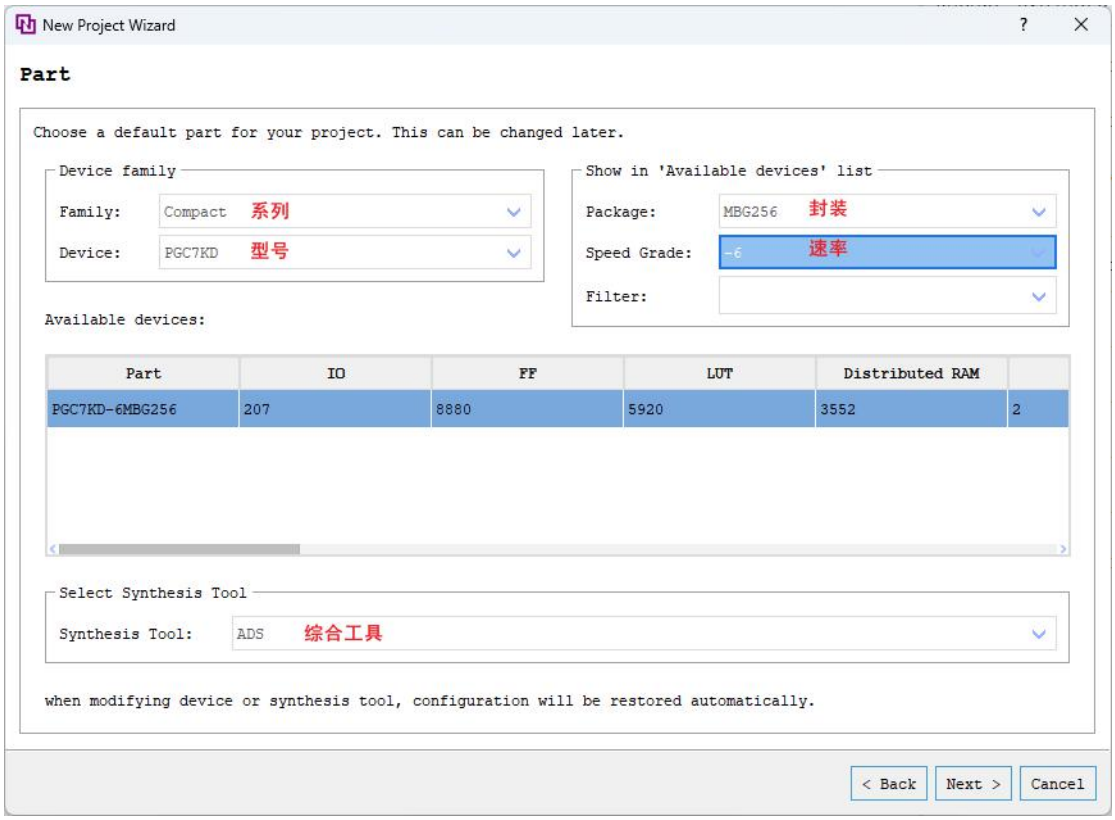
Step6：单击 Next



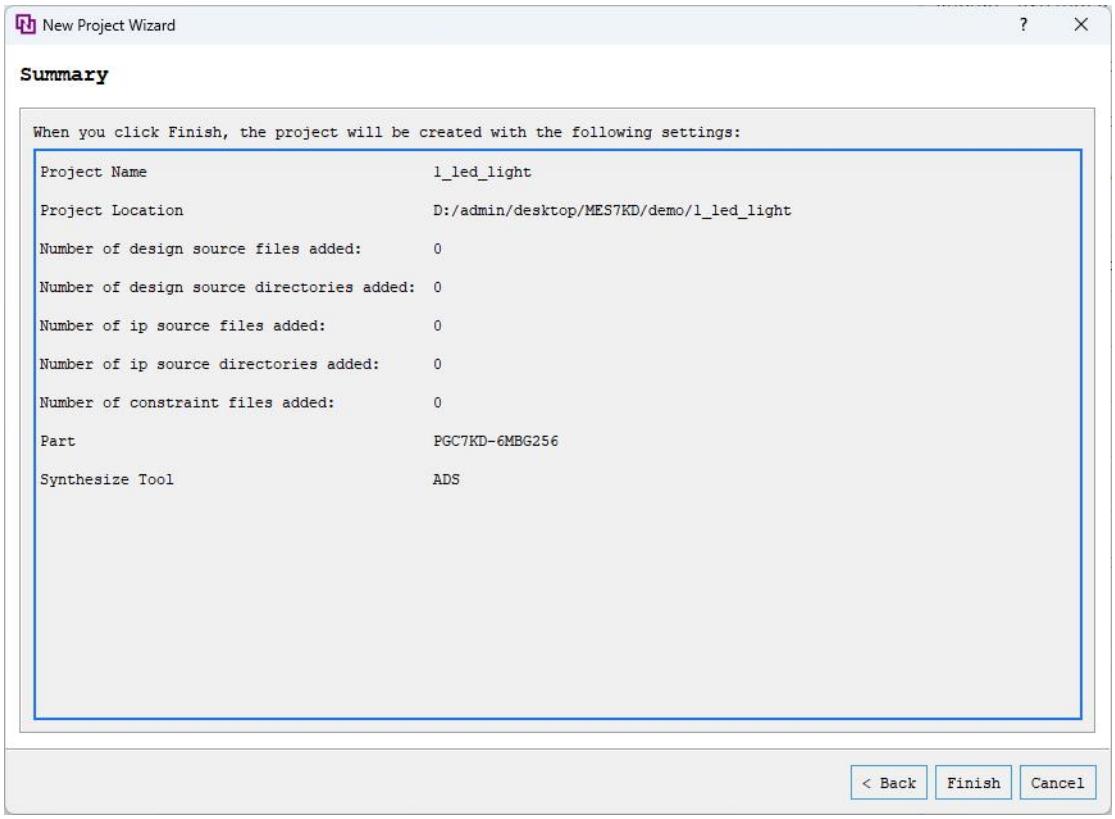
Step7：单击 Next



Step8：选择器件系列、型号、封装、速率、综合工具，之后单击 Next

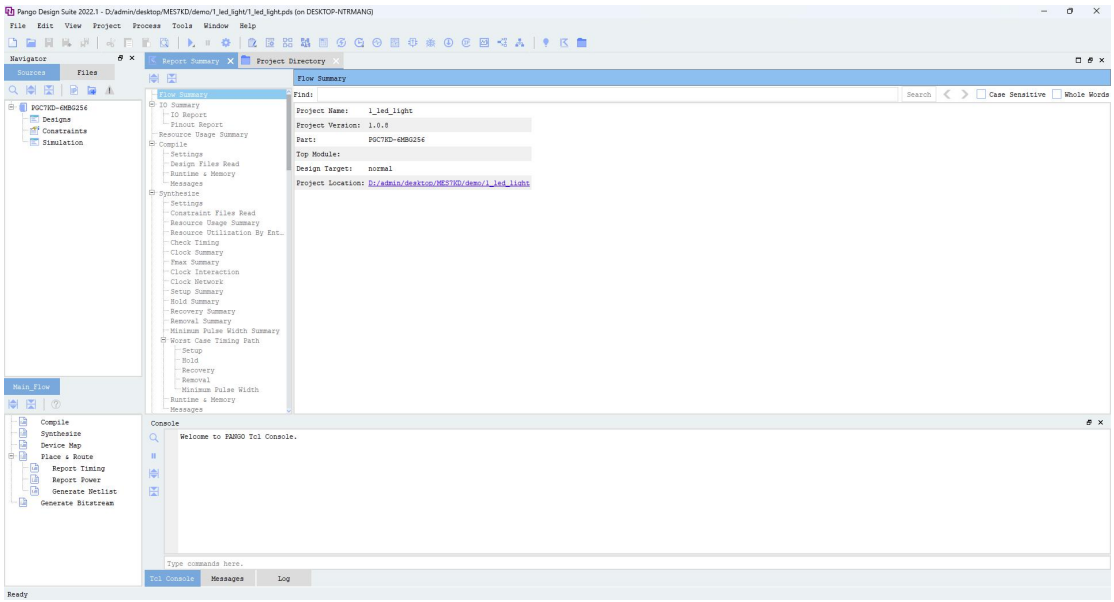


Step9：单击 Finish,完成工程创建

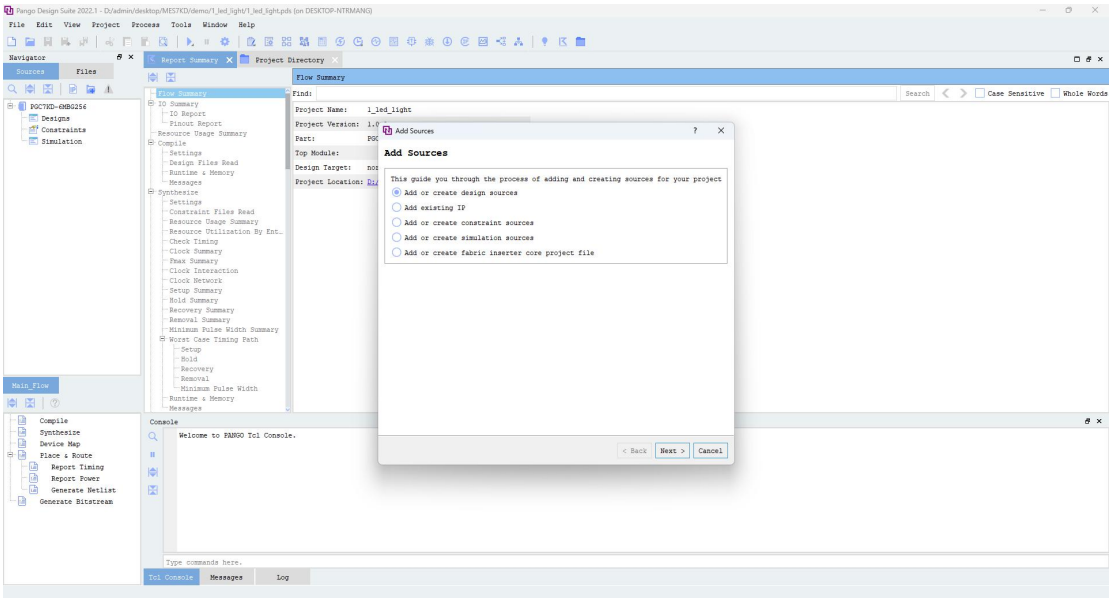


1.5.2添加设计文件

PDS 软件界面如下图：

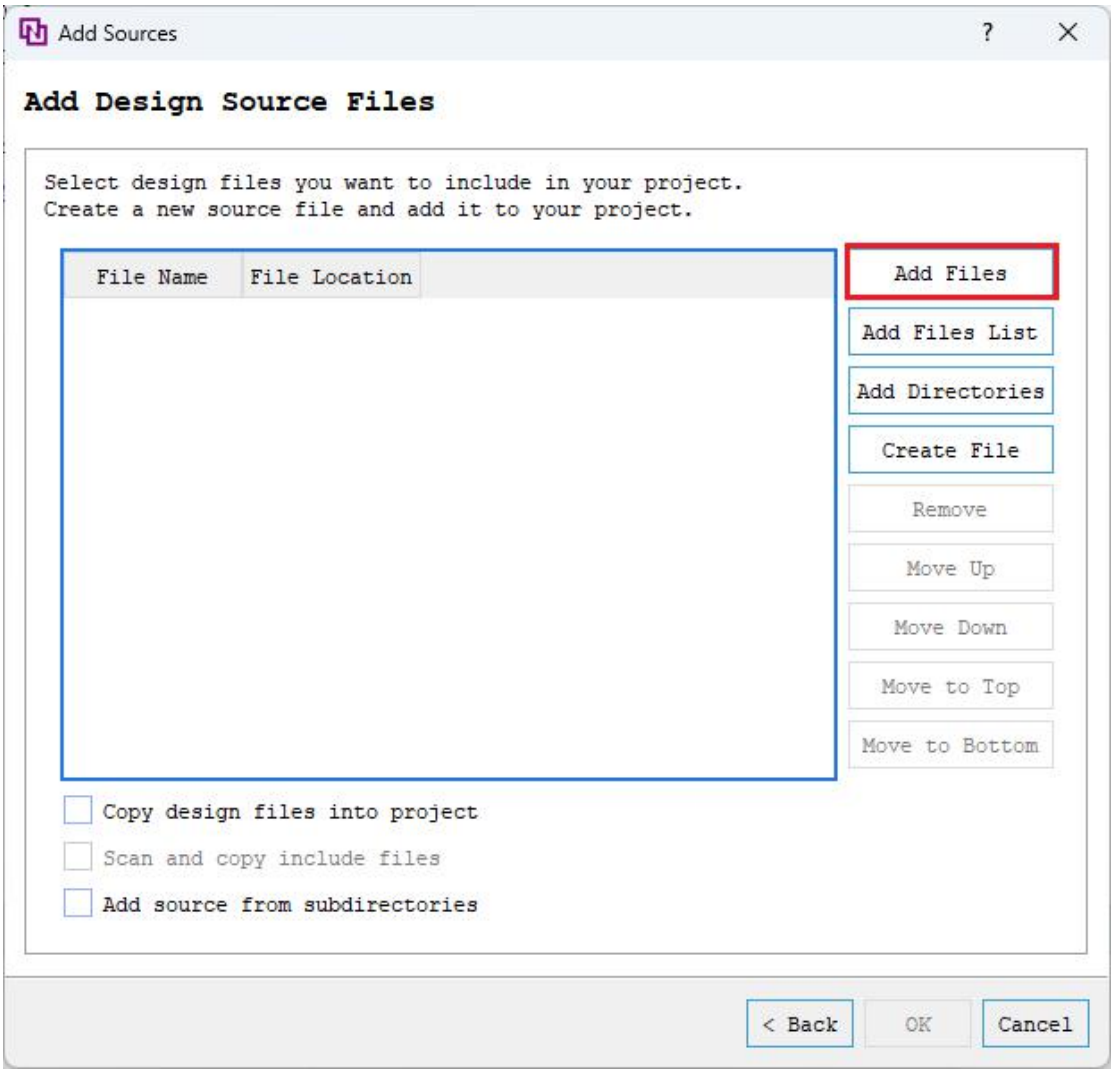


双击 Designs，将前面设计的 module 新建到文件中，或者将前面编辑好的 verilog 文件添加到工程中：



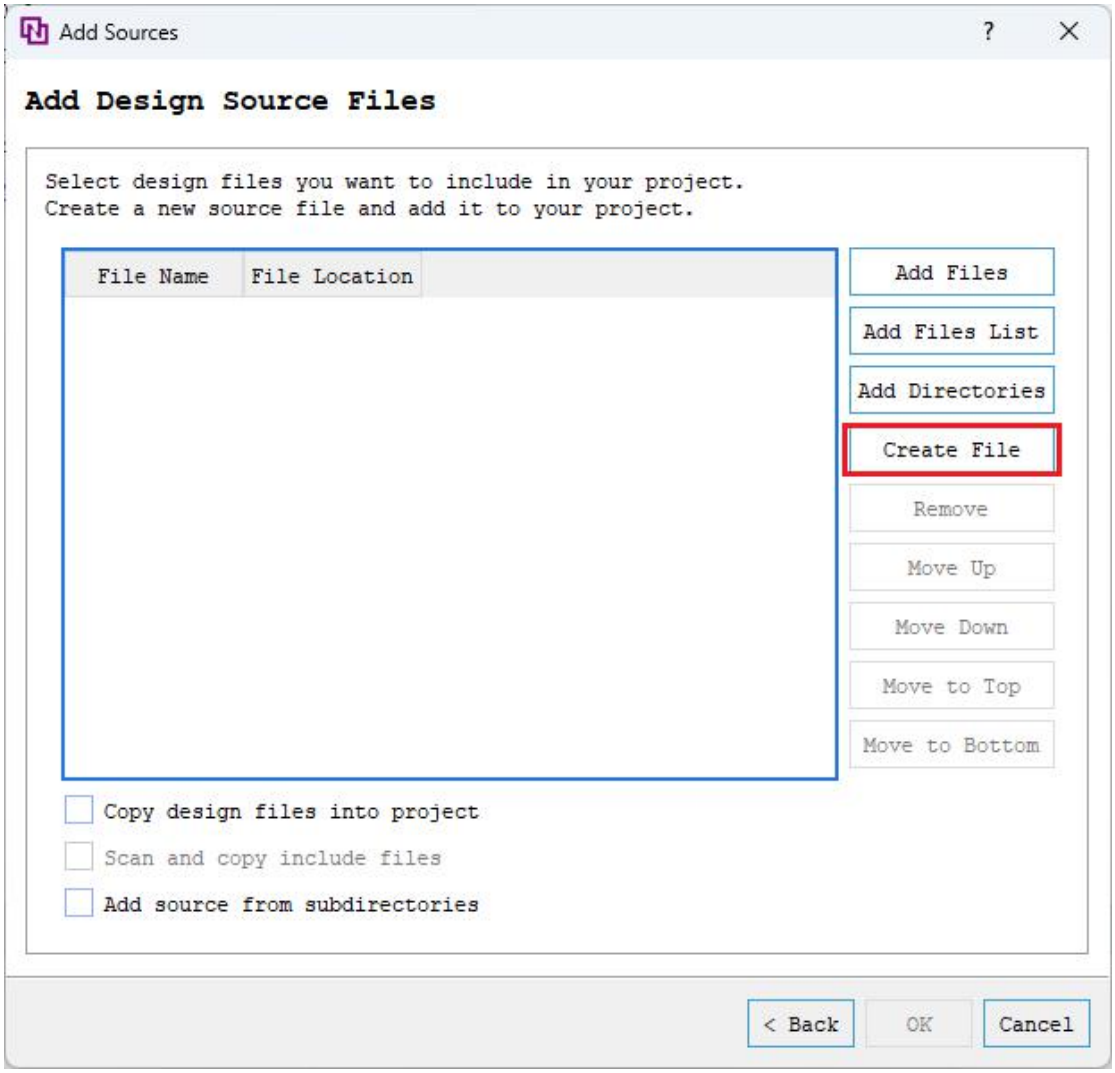
添加文件到工程：

在窗口中点击 Add Files，选择添加文件到工程；

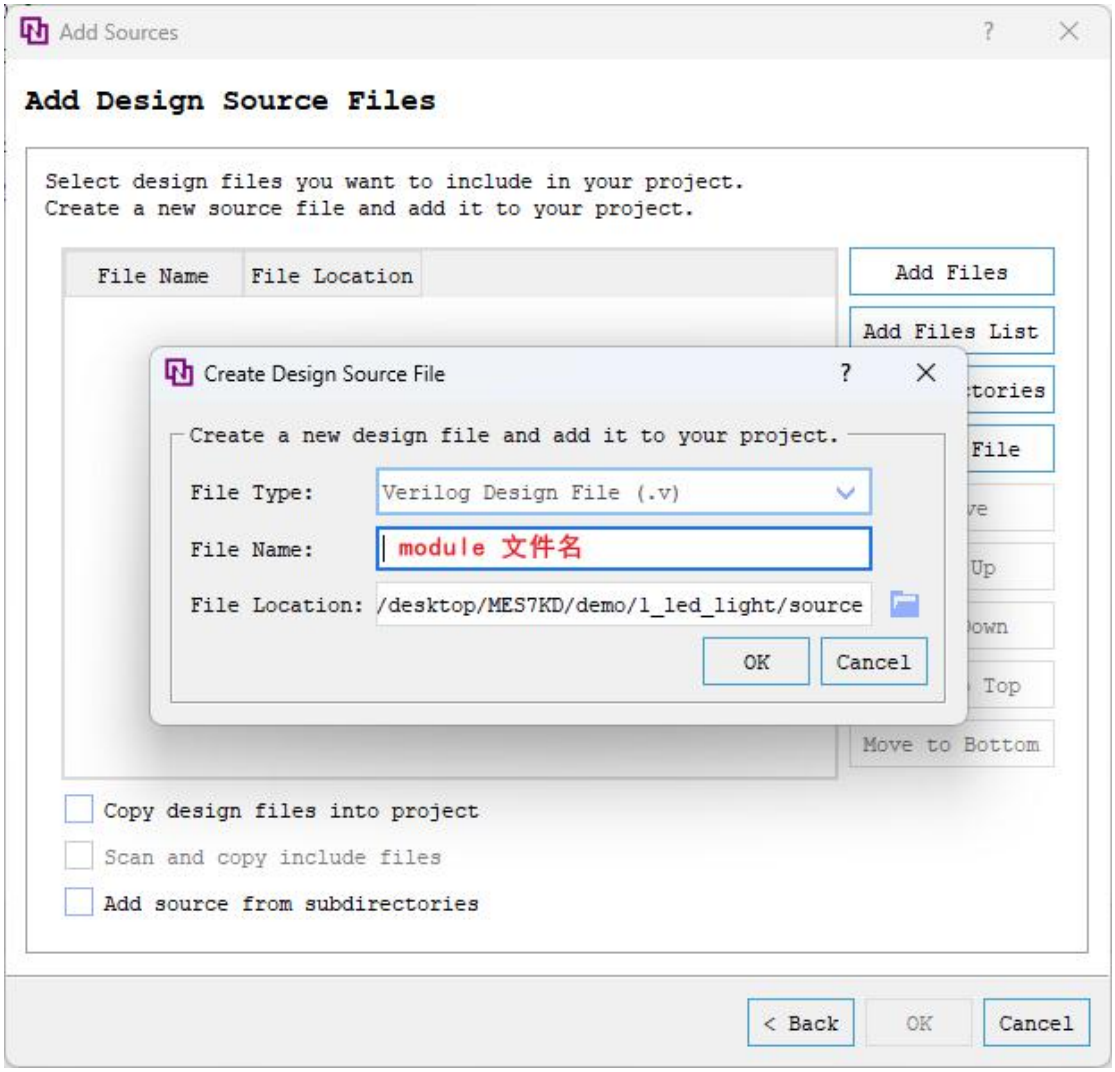


新建文件到工程：

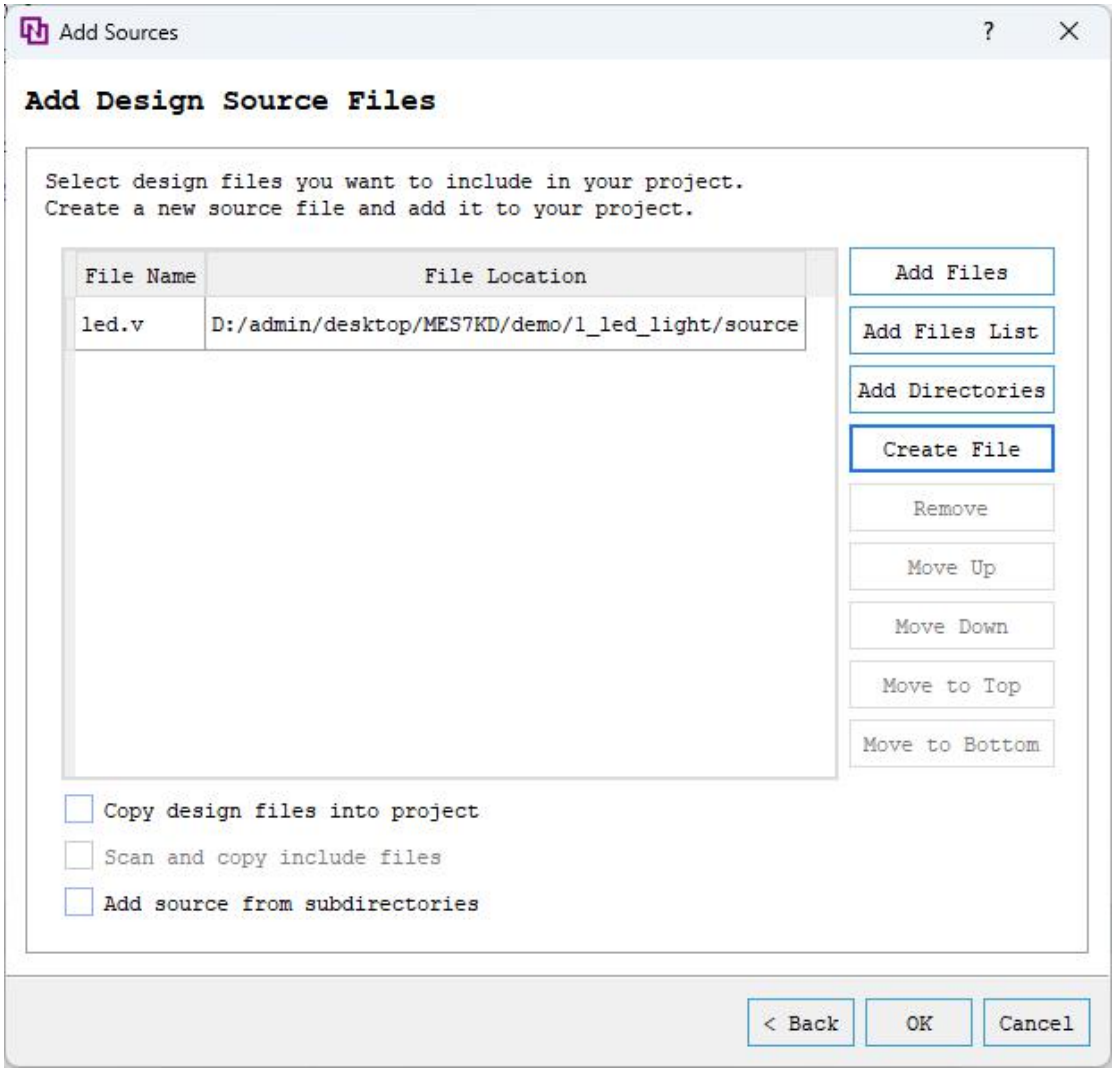
1）在窗口中点击 Create File；



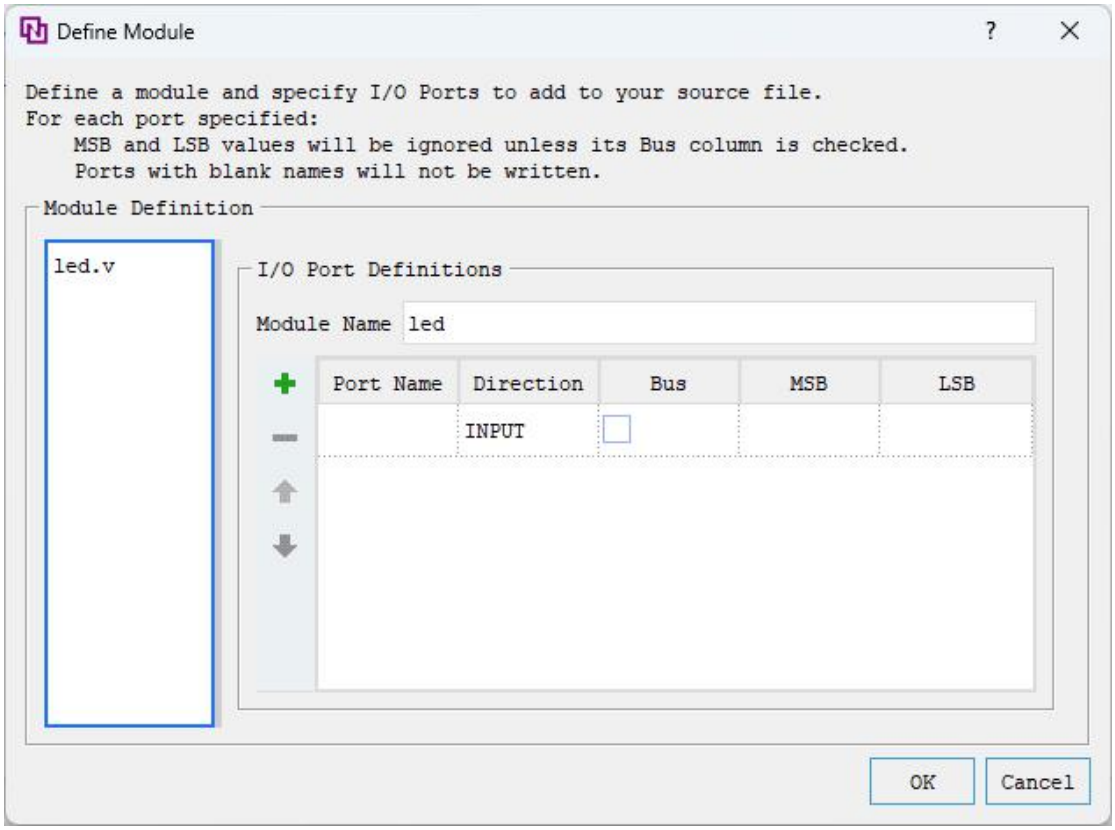
2）选择 Verilog Design File，文件名和 module 名一致，默认路径，点击 OK；


Add Sources
Add Design Source Files
Select design files you want to include in your project.
Create a new source file and add it to your project.
File Name
File Location
Add Files
Add Files List
Create Design Source File
Create a new design file and add it to your project.
File Type:
Verilog Design File (.v)
File Name:
module 文件名
File Location:
/desktop/MES7KD/demo/1_led_light/source
OK
Cancel
Move to Bottom
Copy design files into project
Scan and copy include files
Add source from subdirectories
< Back
OK
Cancel

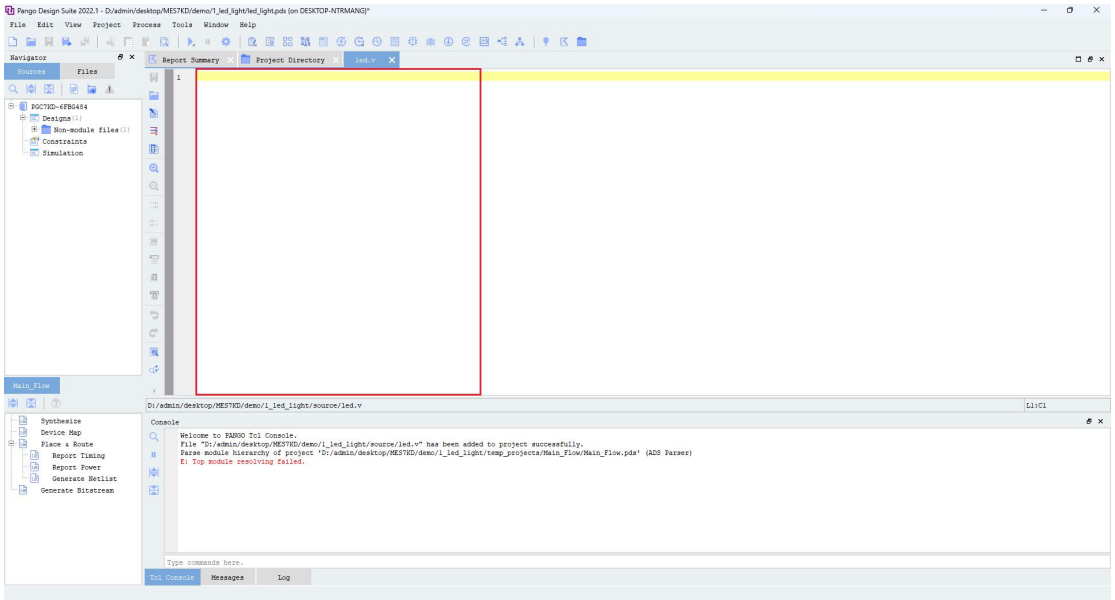
3）点击 OK；


Add Sources
Add Design Source Files
Select design files you want to include in your project.
Create a new source file and add it to your project.
File Name
File Location
led.v
D:/admin/desktop/MES7KD/demo/1_led_light/source
Add Files
Add Files List
Add Directories
Create File
Remove
Move Up
Move Down
Move to Top
Move to Bottom
Copy design files into project
Scan and copy include files
Add source from subdirectories
< Back
OK
Cancel

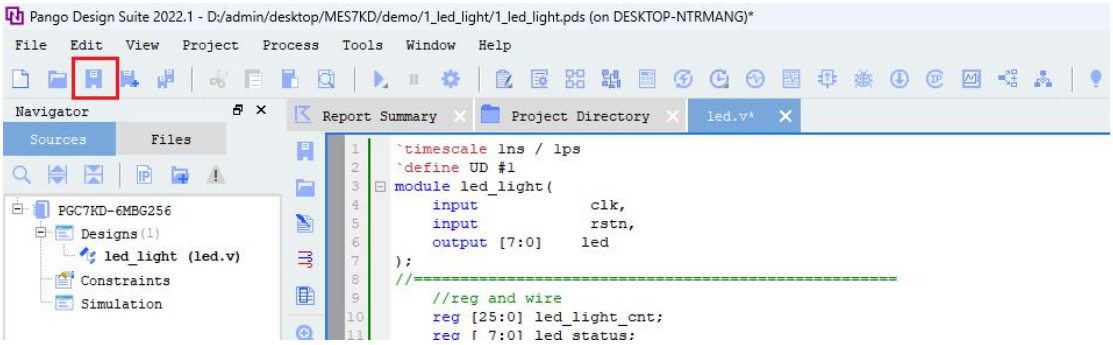
3）点击 Cancel；



4）默认打开新建文件，将前面设计的 module 复制进去，

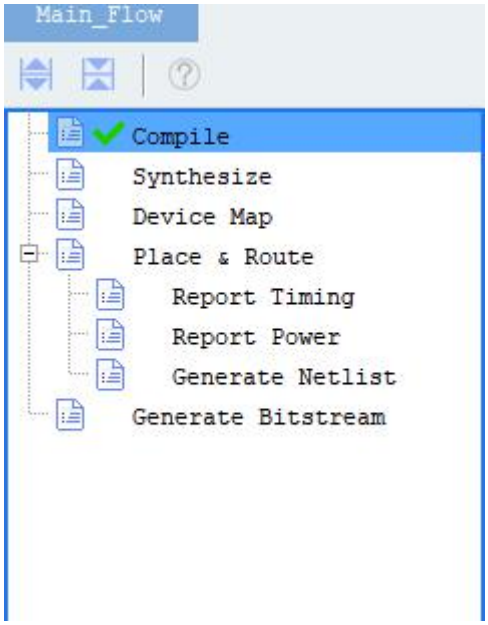


5）点击保存，新建文件完成



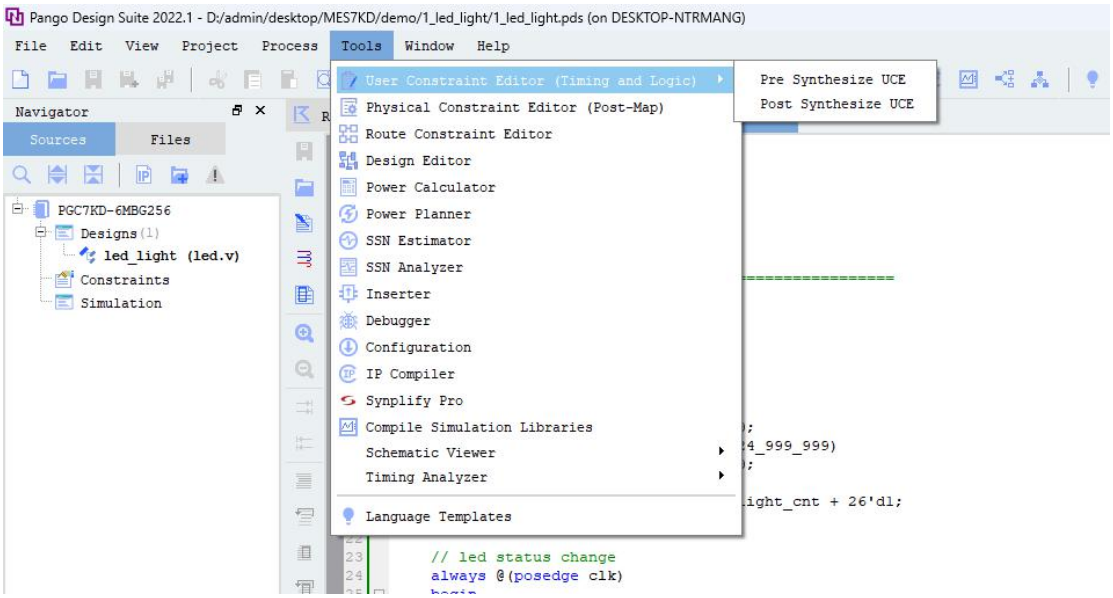
1.5.3编译

双击 Compile 或右击选择 Run，编译完成后如下；

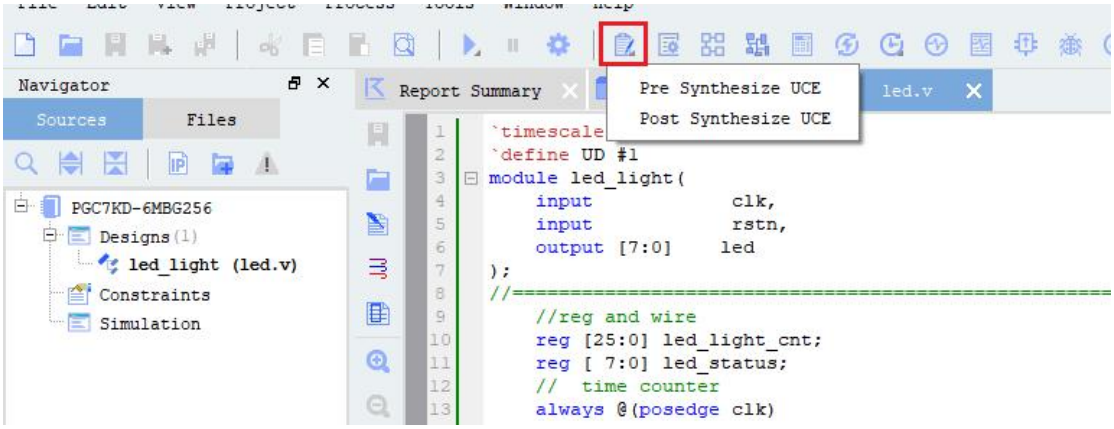


1.5.4工程约束

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



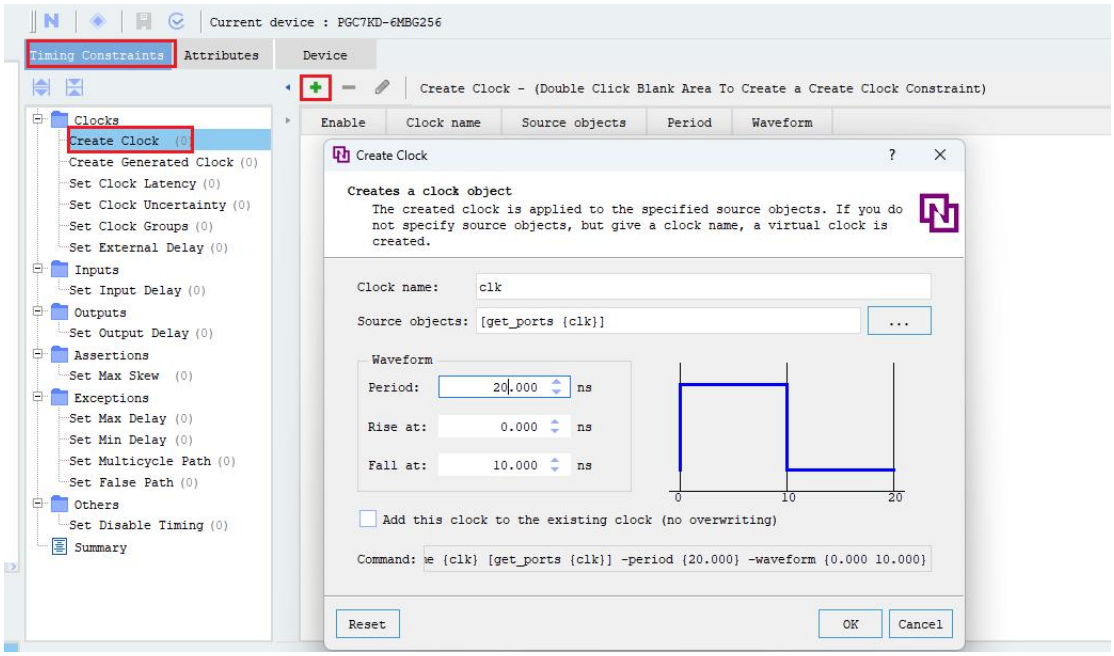
Tools 下的 User Constraint Editor(Timing and Logic)



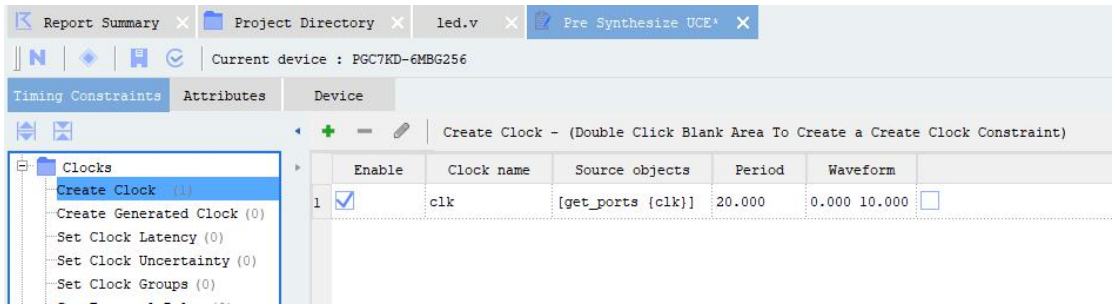
工具栏 User Constraint Editor(Timing and Logic)图标

1.5.4.1 时钟约束

打开 UCE 后，选择 Timing Constraints 后选择 Create Clock 添加基准时钟，基准时钟一般是通过输入 port 输入用户涉及的板上时钟。

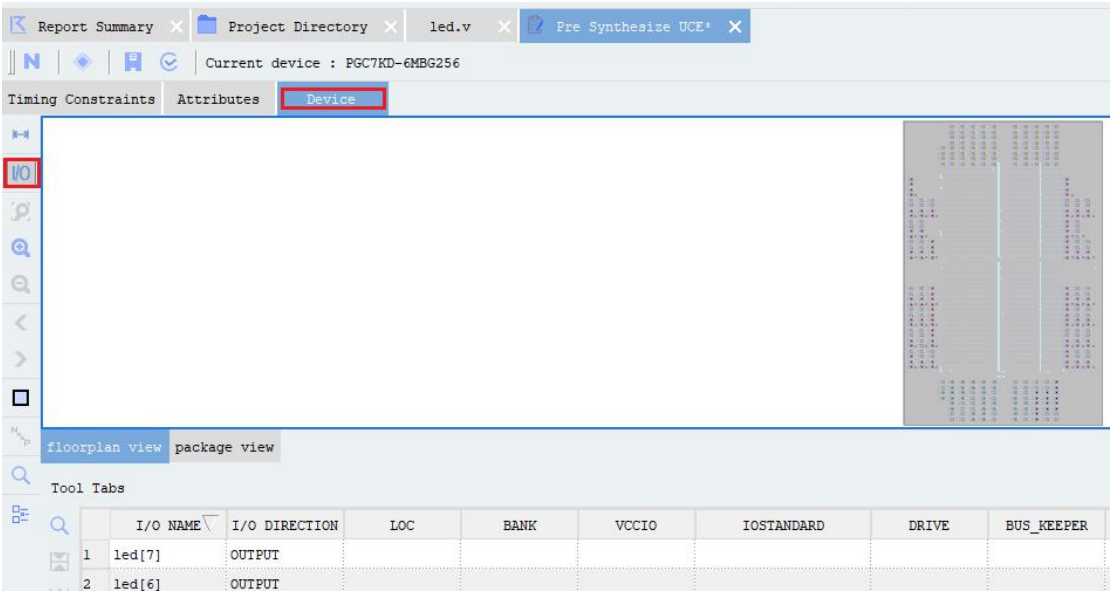


在弹窗中对时钟命名，关联时钟管脚，添加时钟参数，点击 OK 会创建一条时钟约束，Reset 重置该页面。创建完成如下图所示：



1.5.4.2 物理约束

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



	I/O NAME	I/O DIRECTION	LOC	BANK	VCCIO	IOSTANDARD
1	led[7]	OUTPUT	R11	BANK2	3.3	LVCMOS33
2	led[6]	OUTPUT	T12	BANK2	3.3	LVCMOS33
3	led[5]	OUTPUT	R12	BANK2	3.3	LVCMOS33
4	led[4]	OUTPUT	T13	BANK2	3.3	LVCMOS33
5	led[3]	OUTPUT	R13	BANK2	3.3	LVCMOS33
6	led[2]	OUTPUT	T14	BANK2	3.3	LVCMOS33
7	led[1]	OUTPUT	R14	BANK2	3.3	LVCMOS33
8	led[0]	OUTPUT	T15	BANK2	3.3	LVCMOS33
9	clk	INPUT	A9	BANK0	3.3	LVCMOS33
10	rstn	INPUT	T7	BANK2	3.3	LVCMOS33

编辑好 IO 分配后，点击保存，完成约束。

1.6生成位流文件并下载

双击 Generate Bitstream 或右击选择 Run，生成二进制位流文件。

下载位流文件请参考《PDS 快速使用手册》2.8 下载位流文件

1.7实验现象

8 个 LED 灯同时亮和灭，亮和灭之间间隔时间为 0.5s；

2.LED 流水灯

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

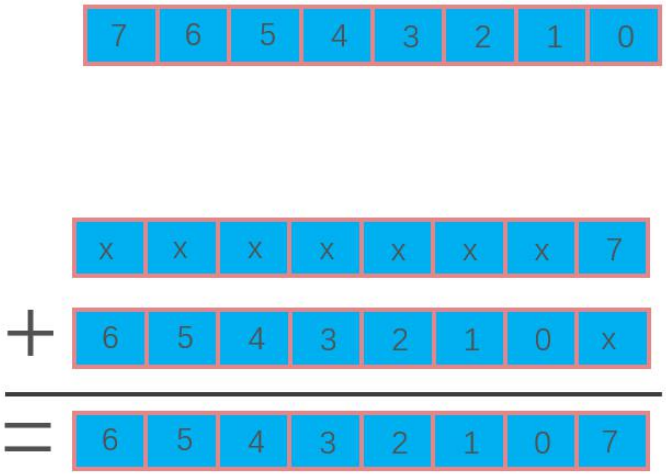
流水灯：8 个 LED 以 0.5s 间隔接替闪烁

2.3实验原理

相比上一个 LED 闪烁的实现，只需要改变 LED 的状态。将 8 个 LED 灯流水式的点亮；

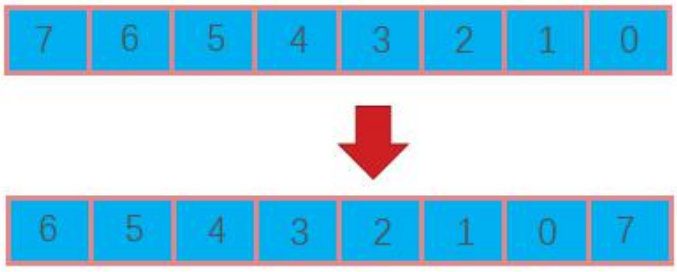
在 C 语言中做流水灯的实验需要用到一个中间变量(代码如下左侧，数据位的搬移如下右图)：

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



在 FPGA 的开发中是基于硬件，语言也是硬件描述语言，verilog 的处理单位就是 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：

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



2.4实验源码设计（完整源码查看 demo 源文件）

Module 的具体内容如下：

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

module water_led(
    input               clk,    //50Mhz
    input               rstn,

    output [7:0]     led
);
//=============================================================
=====================
//reg and wire

    reg [25:0] led_light_cnt;
    reg [ 7:0] led_status;

    //   time counter
    always @(posedge clk)
    begin
```

```
        if(!rstn)
            led_light_cnt <= `UD 26'd0;
        else if(led_light_cnt == 26'd24_999_999)
            led_light_cnt <= `UD 26'd0;
        else
            led_light_cnt <= `UD led_light_cnt + 26'd1;
    end

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

    assign led = led_status;

endmodule
```

2.5实验步骤

工程创建及编译流程与前面 Led 闪烁实验一致，在添加文件的步骤，添加本实验的 water_led 的 verilog 文件即可，管脚分配如下：

	I/O NAME	I/O DIRECTION	LOC	BANK	VCCIO	IOSTANDARD
1	led[7]	OUTPUT	R11	BANK2	3.3	LVCMOS33
2	led[6]	OUTPUT	T12	BANK2	3.3	LVCMOS33
3	led[5]	OUTPUT	R12	BANK2	3.3	LVCMOS33
4	led[4]	OUTPUT	T13	BANK2	3.3	LVCMOS33
5	led[3]	OUTPUT	R13	BANK2	3.3	LVCMOS33
6	led[2]	OUTPUT	T14	BANK2	3.3	LVCMOS33
7	led[1]	OUTPUT	R14	BANK2	3.3	LVCMOS33
8	led[0]	OUTPUT	T15	BANK2	3.3	LVCMOS33
9	clk	INPUT	A9	BANK0	3.3	LVCMOS33
10	rstn	INPUT	T7	BANK2	3.3	LVCMOS33

2.6实验现象

8 个 led 依次被点亮，后一个灯被点亮时前一个灯熄灭，依次往返，让亮起来的 led 灯像是在 8 个 led 灯上流动起来一样，故而此实验称之为流水灯。

3.键控彩灯

3.1实验目的

1　设计 8 种彩灯效果，可通过按键切换。

2　设置 1 个普通按键都作为控制输入，按下一次换一种显示效果，在 8 种效果中循环。

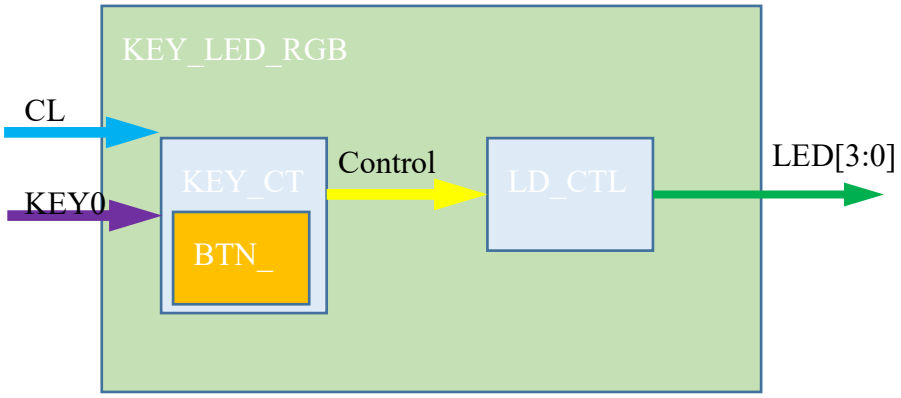
3.2实验要求

1、实验平台：PGX-Lite 7K 开发板；

2、按键输入由 KEY0 输入，LED 输出为 LED1～LED4。

3.3实验原理

实现框架如下：



1、顶层实现按键切换 LED 的彩灯状态；

2、需要设计一个输入控制模块及一个输出控制模块；

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

模块例化方式如下：

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

3.3.1按键控制模块功能

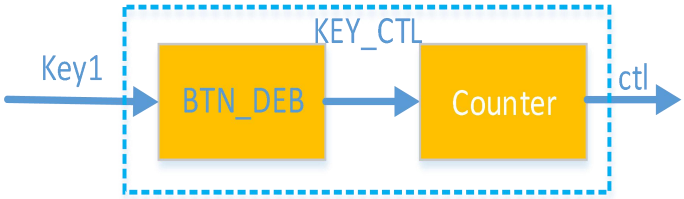
接收按键输入信号。统计按键按下次数，由于彩灯模式是 8 种，计数统计范围是 0～7 循环，将计数结果传递给 LD 控制模块；

根据需求输入信号有：时钟，按键；输出信号有：彩灯控制信号；

内部功能处理：

<1>内部需要对按键信号做消抖处理；

<2>按键触发计数器（计数值输出）改变继而调整彩灯的状态；

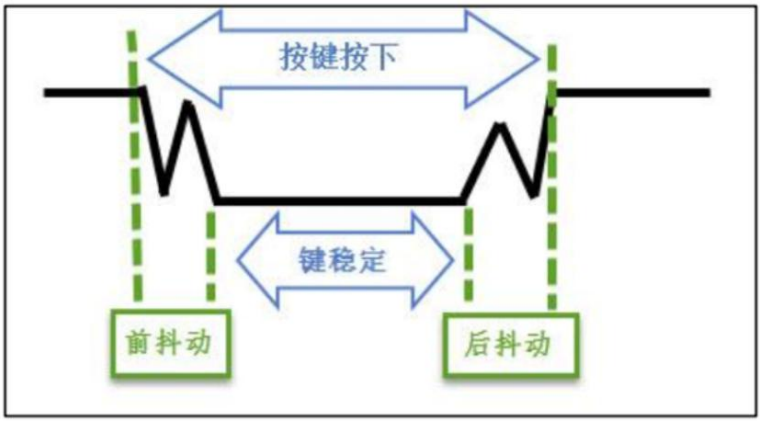


3.3.2按键消抖

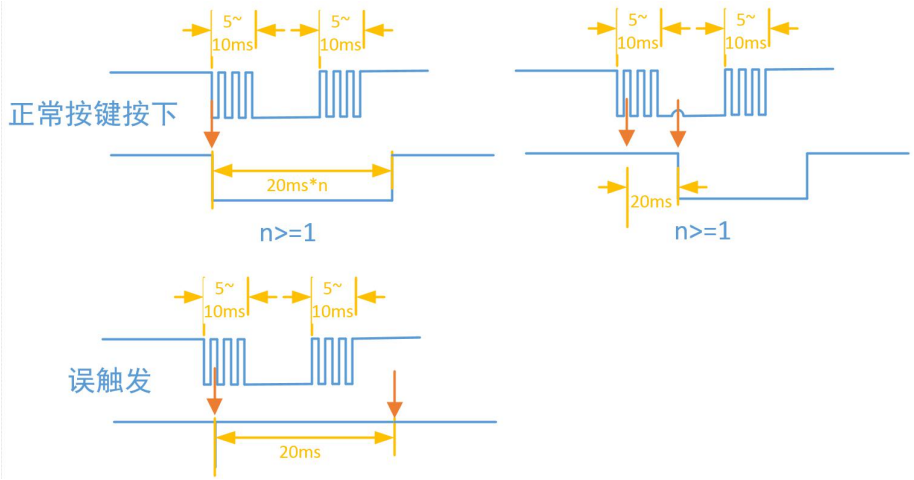
3.3.2.1 消抖目的

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，在快速的变化，在使用按键输入信号时如果采集了抖动时的状态，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，故此实验称之为按键消抖；

3.3.2.2 实验原理



前后抖动时间约为 5～10ms，前后抖动共在 20ms，以最大 20ms 做设计，使用计数到 N 归零的计数器来做时间刻度计时；以 20ms 的间歇对按键输入信号进行采集，从而避开按键的抖动引起的信号快速变化；



设计 1 个 20bit 的计数器，其计数最大值为：N = 20’hF4240 = 20’d1000000

最大计数值时，计时为：t= N*T = N/f = 1000000/50M = 20ms;

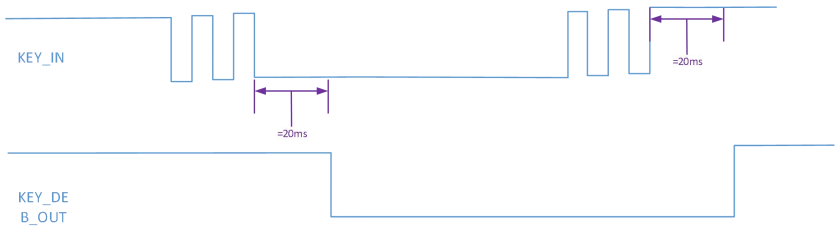
注：对于计数器完成计时功能在 LED 灯控制中已有详细讲解，需要关注输入时钟频率以及目标计时时长，从而得到计数器的计数范围；

3.3.2.3 实验源码设计

```
`timescale 1ns / 1ps
`define UD #1
module btn_deb#(
    parameter                   BTN_WIDTH = 4'd8,
    parameter                   MS_20 = 20'd1_000_000
)
(
```

```
    input                               clk,//50MHz
    input       [BTN_WIDTH-1:0] btn_in,

    output reg [BTN_WIDTH-1:0] btn_deb
);

//=============================================================
============================
    reg [19:0] time_cnt= 20'd0;
    always@(posedge clk)
    begin
        if(time_cnt == MS_20 - 1'b1)
            time_cnt <= `UD 20'd0;
        else
            time_cnt <= `UD time_cnt + 1'd1;
    end

    always @(posedge clk)
    begin
        if(time_cnt == MS_20 - 1'b1)
            btn_deb <= `UD btn_in;
    end

endmodule
```

此种方法有一定误触发概率出现，大家可在此基础上做补充完善；思路如下（扩展实现）：



这个 module 的设计中新增加一种语法：parameter ；在 verilog 中 parameter 是对常量进行定义，将 parameter 定义放在 module 的接口中是可进行模块传递，传递方式请看后面模块例化；

3.3.3LED 控制模块功能

8 种流水灯模式有按键传递过来的计数控制切换，每一个 LED 的显示状态

完整后进入下一模式初始化。根据需求可得到如下信息：

输入信号：时钟，彩灯模式控制信号； 出信号：12bit 位宽的 LED 控制信号；

功能处理注意事项：彩灯状态切换点，不同状态的切换时如何初始化；

3.4实验源码设计（完整源码查看 demo 源文件）

3.4.1顶层文件源码

```
`timescale 1ns / 1ps
`define UD #1
module key_led_rgb_top(
    input                       clk/* synthesis PAP_MARK_DEBUG="true" */,//50MHz
    input               key,

    output [11:0]       led_rgb/* synthesis PAP_MARK_DEBUG="true" */
);

    wire [2:0] ctrl;

    key_ctl key_ctl(
        .clk            (  clk  ),//input                 clk,
        .key            (  key  ),//input                 key,

        .ctrl           (  ctrl  )//output        [1:0] ctrl
    );

    led_rgb  u_led_rgb(
        .clk    (  clk    ),//input             clk,
        .ctrl   (  ctrl   ),//input   [1:0]   ctrl,

        .led_rgb    (  led_rgb    ) //output [11:0]   led
    );

endmodule
```

3.4.2按键控制模块

```
`timescale 1ns / 1ps
`define UD #1
module key_ctl(
    input               clk,
    input               key,

    output      [3:0] ctrl
);

    wire        btn_deb;
    // 按键消抖
    btn_deb#(
        .BTN_WIDTH      (     4'd1                  ),  //parameter BTN_WIDTH = 4'd8
        .MS_20          (  20'd1_000_000  )
    ) U_btn_deb
    (
        .clk                    (     clk                   ),//input clk,
        .btn_in         (  key              ),//input       [BTN_WIDTH-1:0] btn_in,

        .btn_deb        (  btn_deb        ) //output reg [BTN_WIDTH-1:0] btn_deb
    );

    reg         btn_deb_1d;
    always @(posedge clk)
    begin
        btn_deb_1d <= `UD btn_deb;
    end

    reg [2:0]key_push_cnt[0:7];

generate
    genvar i ;
        for (i=0;i<1;i=i+1) begin : key_ctrl
    always @(posedge clk)
    begin
        if(~btn_deb[i] & btn_deb_1d[i])
        begin
                key_push_cnt[i] <= 3'd0 ;
```

```
                if(key_push_cnt[i] == 3'd7)
                    key_push_cnt[i] <= `UD 2'd0;
                else
                    key_push_cnt[i] <= `UD key_push_cnt[i] + 3'd1;
            end
        end
    end
endgenerate


    assign ctrl = key_push_cnt[0] ;

endmodule
```

按键消抖

```
`timescale 1ns / 1ps
`define UD #1
module btn_deb#(
    parameter                       BTN_WIDTH = 4'd8,
    parameter                       MS_20 = 20'd1_000_000
)
(
    input                           clk,//50MHz
    input       [BTN_WIDTH-1:0] btn_in,

    output reg [BTN_WIDTH-1:0] btn_deb
);

//=============================================================
============================
    reg [19:0] time_cnt= 20'd0;
    always@(posedge clk)
    begin
        if(time_cnt == MS_20 - 1'b1)
            time_cnt <= `UD 20'd0;
        else
            time_cnt <= `UD time_cnt + 1'd1;
    end

    always @(posedge clk)
    begin
        if(time_cnt == MS_20 - 1'b1)
            btn_deb <= `UD btn_in;
    end
```

```
endmodule
```

3.4.3LED 控制模块

```
`timescale 1ns / 1ps
`define UD #1
module led_rgb(
    input             clk,//50MHz
    input   [2:0]   ctrl,

    output   reg [11:0]   led_rgb
);

    // led_rgb status change
    always @(posedge clk)
        begin
            case(ctrl)
                    3'd0 :   //R
                        led_rgb<=`UD 12'b0000_1111_1111;
                    3'd1 :   //G
                        led_rgb<=`UD 12'b1111_0000_1111;
                    3'd2 :   //B
                        led_rgb<=`UD 12'b1111_1111_0000;
                    3'd3 :   //RG
                        led_rgb<=`UD 12'b0000_0000_1111;
                    3'd4 :   //RB
                        led_rgb<=`UD 12'b0000_1111_0000;
                    3'd5 :   //GB
                        led_rgb<=`UD 12'b1111_0000_0000;
                    3'd6 :   //RGB
                        led_rgb<=`UD 12'b0000_0000_0000;
                    default:led_rgb<=`UD 12'b1111_1111_1111;
                endcase
            end

endmodule
```

3.5实验现象

上电后下载完固件，默认 LD1～LD4 流水，每按下一次 KEY0，彩灯状态切换一次，总共 8 种状态可供循环切换；

4.数码管动态显示

4.1实验目的

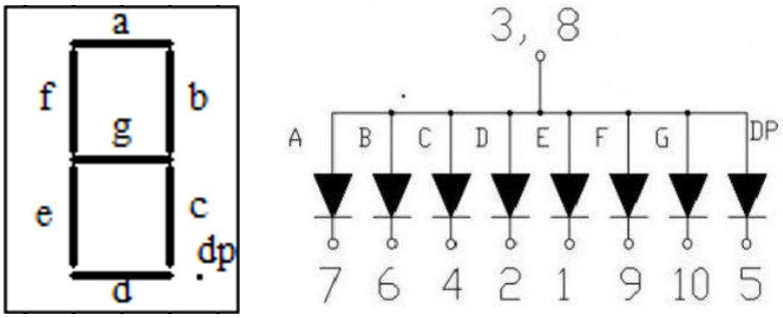
动态控制两个 4 位数码管显示不同的数值；

4.2实验要求

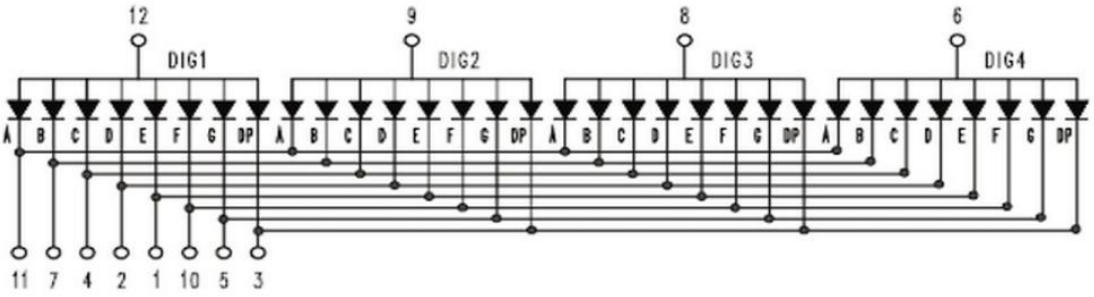
八个数码管显示不同的数字，按键 K0 控制右侧起第一个数码管，按一下数字加 1，从 0 到 9，按键 K1 控制右侧起第二个数码管，按一下数字加 1，从 0 到 9，按键 K2 控制右侧起第三个数码管、依次类推。

4.3实验原理

数码管是一种半导体发光器件，其基本单元是发光二极管。能显示 4 个数码管叫四位数码管。数码管按段数分为七段数码管和八段数码管，八段数码管比七段数码管多一个发光二极管单元（多一个小数点显示）；按发光二极管单元连接方式分为共阳极数码管和共阴极数码管。共阳数码管是指将所有发光二极管的阳极接到一起形成公共阳极(COM)的数码管。共阳数码管在应用时应将公共极 COM 接到+5V，当某一字段发光二极管的阴极为低电平时，相应字段就点亮。当某一字段的阴极为高电平时，相应字段就不亮。共阴数码管是指将所有发光二极管的阴极接到一起形成公共阴极(COM)的数码管。共阴数码管在应用时应将公共极 COM 接到地线 GND 上，当某一字段发光二极管的阳极为高电平时，相应字段就点亮。当某一字段的阳极为低电平时，相应字段就不亮。



4 位共阳数码管内部管脚连接图如下：



段选：段选由 8 根 led 灯组成，分别为 a，b，c，d，e，f，g，dp；

由段选信号控制某段数码管点亮；

位选：位选由 4 组 8 个段选 LED 组成，分别为 seg1，seg2，seg3，seg4；

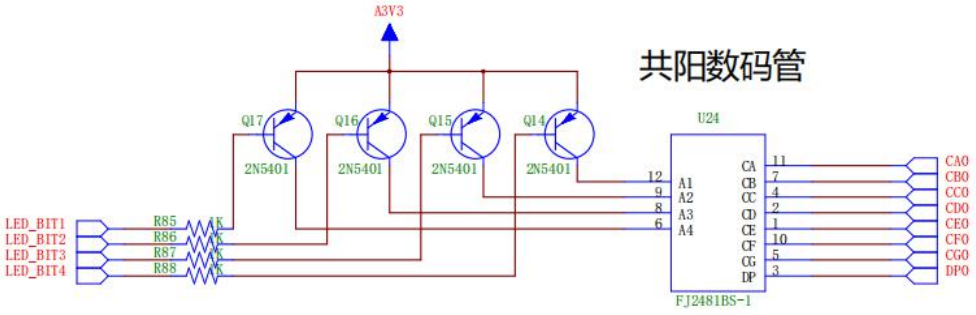
由选通信号控制第几块数码管点亮；

例：如果我们只点亮第一位的 A：需要将 11 脚配置低电平，其他段选（1-5，7，10，11）配置高电平；将 12 脚配置高电平，其他位选脚配置（6，8，9）低电平；

点亮数码管原理：

输入相应的电平点亮一根根小火柴 a-b-c-d-e-f-g-dp。如果数码管是共阴极，给高电平 1 即可相应点亮，反之如果是共阳极，给低电平 0 即可相应点亮。

PGX-Lite 7K 数码管底板的数码管使用共阳数码管；当 LED_BIT1 为低电平时，A4 为高电平，对应位数码管亮，当 LED_BIT2 为低电平时，A3 为高电平，对应位数码管亮，当 LED_BIT3 为低电平时，A2 为高电平，对应位数码管亮，当 LED_BIT4 为低电平时，A1 为高电平，对应位数码管亮。



数码管显示出 0～9，代码如下，通过传递要显示的数值给到 key 上，可显示对应数值，sel 选择对应的数码管，如需 4 个如果要显示同样的字符，仅需将 dig 的 4 位全部置 1（位选信号经过 PMOS 管后传输到数码管，因此低电平有效，

所以 dig 信号在顶层输出时，应进行一次取反操作），需要做好对应编码；

```
module seq_control
(
    input [1:0]sel,
    input [3:0]key,
    output reg [3:0]dig,
    output reg [7:0]smg
);
    /*================================================
                        位选择映射
    ================================================*/
    always @(*)
    begin
        case(sel)
            2'd0:dig = 4'b0001;
            2'd1:dig = 4'b0010;
            2'd2:dig = 4'b0100;
            2'd3:dig = 4'b1000;
            default:dig = 4'b0000;
        endcase
    end

    //=========================================
    // 0 1 2 3 4 5 6 7
    // G F E D C B A P
    //共阳极数码管，为0有效，即点亮
    //=========================================
    always @(*)
    begin
        case(key)
            4'd0:smg = 8'b1000_0001;//"0"
            4'd1:smg = 8'b1100_1111;//"1"
            4'd2:smg = 8'b1001_0010;//"2"
            4'd3:smg = 8'b1000_0110;//"3"
            4'd4:smg = 8'b1100_1100;//"4"
            4'd5:smg = 8'b1010_0100;//"5"
            4'd6:smg = 8'b1010_0000;//"6"
            4'd7:smg = 8'b1000_1111;//"7"
            4'd8:smg = 8'b1000_0000;//"8"
            4'd9:smg = 8'b1000_0100;//"9"
            default:smg = 8'b1111_1111;
        endcase
    end

endmodule

```

硬件连接上无法同一个时间点显示出不同的数值，我们可以通过刷新显示的方式造成视觉上同时显示了不同的数值，依据如下：

人眼对于时间频率的响应近似一个滤波器，在一般室内强光下，对 15~20Hz 信号最敏感，有很强闪烁感(flick)，大于 75Hz 响应为 0，闪烁感消失。刚到达闪

烁感消失的频率叫做临界融合频率(CFF)。在较暗的环境下，呈低通特性，且 CFF 会降低，这时对 5Hz 信号最敏感，大于 25Hz 闪烁基本消失。电影院环境很暗，放映机的刷新率为 24Hz 也不感到闪烁；这种特性也可以解析为视觉暂留特性，即当影像消失/变化时，大脑的影像不会立刻消失，而是保留一个短暂时间。

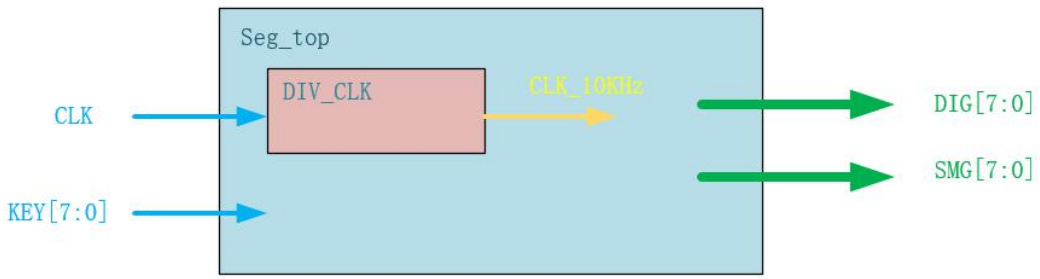
在设计数码管闪烁式显示时，对于人眼观测来说，频率越高越好，但是数码管中的 LED 灯珠点亮对于高电平（关注发光响应时间）是有要求的，故而不是越高越好，取一个适当的刷新频率即可，实验中我们取刷新率为 10KHz。

方案设计：

1、按键消抖：参考按键流水灯实验

2、按键计数：参考按键流水灯实验

3、数码管的分时显示；



4.4实验源码（完整源码查看 demo 源文件）

4.4.1顶层模块

```
`timescale 1ns / 1ps
`define UD #1
module top_seq
(
    input clk/* synthesis PAP_MARK_DEBUG="true" */,//50MHZ 20ns
    input [7:0]button,
    output [3:0]dig_1/* synthesis PAP_MARK_DEBUG="true" */,
    output [3:0]dig_2/* synthesis PAP_MARK_DEBUG="true" */,
    output [7:0]smg_1/* synthesis PAP_MARK_DEBUG="true" */,
    output [7:0]smg_2/* synthesis PAP_MARK_DEBUG="true" */
);
```

```
reg [3:0]dig /* synthesis PAP_MARK_DEBUG="true" */;
reg [7:0]smg /* synthesis PAP_MARK_DEBUG="true" */;

//assign dig_1 = 4'b0111 ;
//assign dig_2 = 4'b1110 ;

assign dig_1 = (sel_reg <= 3'd3)? dig : 4'b1111 ;
assign dig_2 = (sel_reg >   3'd3)? dig : 4'b1111 ;
assign smg_1 = (sel_reg <= 3'd3)? smg : 8'b1111_1111 ;
assign smg_2 = (sel_reg >   3'd3)? smg : 8'b1111_1111 ;

/*=====================================================
                        按键消抖
=====================================================*/
wire [7:0]key;
btn_deb
#(
    .BT_WIDTH(4'd8)
)
u_btn_deb
(
      .clk(clk),
      .btn_in(button),
      .btn_out(key)
);
/*=====================================================
                        4 个按键的计数
=====================================================*/
wire [3:0] key0_cnt;
key_cnt key1
(
      .clk(clk),
      .key(key[0]),
      .key_times(key0_cnt)
);

wire [3:0] key1_cnt;
key_cnt key2
(
      .clk(clk),
      .key(key[1]),
      .key_times(key1_cnt)
);
```

```
wire [3:0] key2_cnt;
key_cnt key3
(
        .clk(clk),
        .key(key[2]),
        .key_times(key2_cnt)
);

wire [3:0] key3_cnt;
key_cnt key4
(
        .clk(clk),
        .key(key[3]),
        .key_times(key3_cnt)
);

wire [3:0] key4_cnt;
key_cnt key5
(
        .clk(clk),
        .key(key[4]),
        .key_times(key4_cnt)
);

wire [3:0] key5_cnt;
key_cnt key6
(
        .clk(clk),
        .key(key[5]),
        .key_times(key5_cnt)
);

wire [3:0] key6_cnt;
key_cnt key7
(
        .clk(clk),
        .key(key[6]),
        .key_times(key6_cnt)
);

wire [3:0] key7_cnt;
key_cnt key8
(
```

```
        .clk(clk),
        .key(key[7]),
        .key_times(key7_cnt)
);
/*=====================================================
                        时钟分频
=====================================================*/
wire clk_10khz;
div_clk div_clk
(
        .clk            (clk),
        .clk_10khz (clk_10khz)    //更改为 0.01M
);
/*=====================================================
                        数码管显示
=====================================================*/
reg    [2:0]sel=0;
wire [3:0]dig0/* synthesis PAP_MARK_DEBUG="true" */;
wire [7:0]smg0/* synthesis PAP_MARK_DEBUG="true" */;

always @(posedge clk_10khz)
begin
        sel <= `UD sel+1'b1;
end

seq_control seq_control_0
(
        .sel(3'd0),
        .key(key4_cnt),
        .dig(dig0),
        .smg(smg0)
);

wire [3:0]dig1/* synthesis PAP_MARK_DEBUG="true" */;
wire [7:0]smg1/* synthesis PAP_MARK_DEBUG="true" */;

seq_control seq_control_1
(
        .sel(3'd1),
        .key(key5_cnt),
        .dig(dig1),
        .smg(smg1)
);
```

```
wire [3:0]dig2/* synthesis PAP_MARK_DEBUG="true" */;
wire [7:0]smg2/* synthesis PAP_MARK_DEBUG="true" */;

seq_control seq_control_2
(
    .sel(3'd2),
    .key(key6_cnt),
    .dig(dig2),
    .smg(smg2)
);

wire [3:0]dig3/* synthesis PAP_MARK_DEBUG="true" */;
wire [7:0]smg3/* synthesis PAP_MARK_DEBUG="true" */;

seq_control seq_control_3
(
    .sel(3'd3),
    .key(key7_cnt),
    .dig(dig3),
    .smg(smg3)
);

wire [3:0]dig4/* synthesis PAP_MARK_DEBUG="true" */;
wire [7:0]smg4/* synthesis PAP_MARK_DEBUG="true" */;

seq_control seq_control_4
(
    .sel(3'd4),
    .key(key0_cnt),
    .dig(dig4),
    .smg(smg4)
);

wire [3:0]dig5/* synthesis PAP_MARK_DEBUG="true" */;
wire [7:0]smg5/* synthesis PAP_MARK_DEBUG="true" */;

seq_control seq_control_5
(
    .sel(3'd5),
    .key(key1_cnt),
    .dig(dig5),
    .smg(smg5)
);
```

```
wire [3:0]dig6/* synthesis PAP_MARK_DEBUG="true" */;
wire [7:0]smg6/* synthesis PAP_MARK_DEBUG="true" */;

seq_control seq_control_6
(
    .sel(3'd6),
    .key(key2_cnt),
    .dig(dig6),
    .smg(smg6)
);

wire [3:0]dig7/* synthesis PAP_MARK_DEBUG="true" */;
wire [7:0]smg7/* synthesis PAP_MARK_DEBUG="true" */;

seq_control seq_control_7
(
    .sel(3'd7),
    .key(key3_cnt),
    .dig(dig7),
    .smg(smg7)
);

always @(posedge clk_10khz)
begin
    if(sel==3'b000)
        dig <= `UD ~dig0;
    else if(sel==3'b001)
        dig <= `UD ~dig1;
    else if(sel==3'b010)
        dig <= `UD ~dig2;
    else if(sel==3'b011)
        dig <= `UD ~dig3;
    else if(sel==3'b100)
        dig <= `UD ~dig4;
    else if(sel==3'b101)
        dig <= `UD ~dig5;
    else if(sel==3'b110)
        dig <= `UD ~dig6;
    else if(sel==3'b111)
        dig <= `UD ~dig7;
end

always @(posedge clk_10khz)
begin
```

```
        if(sel==3'b000)
            smg <= `UD smg0;
        else if(sel==3'b001)
            smg <= `UD smg1;
        else if(sel==3'b010)
            smg <= `UD smg2;
        else if(sel==3'b011)
            smg <= `UD smg3;
        else if(sel==3'b100)
            smg <= `UD smg4;
        else if(sel==3'b101)
            smg <= `UD smg5;
        else if(sel==3'b110)
            smg <= `UD smg6;
        else if(sel==3'b111)
            smg <= `UD smg7;
    end

    reg [2:0]sel_reg = 3'b0 ;

    always @(posedge clk_10khz)
    begin
        sel_reg <= sel ;
    end

    endmodule
```

4.4.2按键消抖模块

```
`timescale 1ns / 1ps
`define UD #1
module btn_deb #(
    parameter     BT_WIDTH = 4'd8
)
(
    input                          clk,//50M.20ns
    input          [BT_WIDTH-1:0]      btn_in,

    output reg [BT_WIDTH-1:0]      btn_out
);


    reg [19:0] time_cnt=20'd0;//20ms 计时计数寄存器;
```

```
        always @(posedge clk)
        begin
            time_cnt <= `UD time_cnt + 20'd1;   //计数器到
20'hf_ffff时周期约为21ms
        end

        always @(posedge clk)
        begin
            if(time_cnt == 0)   //每隔20ms取一次按键状态值;
                btn_out <= `UD btn_in;
        end

    endmodule
```

4.4.3按键计数模块

```
    `timescale 1ns / 1ps
    `define UD #1
    module key_cnt
    (
        input clk,//50M,20ns
        input rstn_key,
        input key,
        output reg [3:0]key_times
    );

    reg key_reg;
    always @(posedge clk)
    begin
        key_reg <= `UD key;
    end

    always @(posedge clk )
    begin
        if(key_reg&&~key&&key_times==4'd9)
            key_times <=`UD 4'd0;
        else if(key_reg&&~key)
            key_times <=`UD key_times + 1'b1;
    end
```

```
endmodule
```

4.4.4时钟分频模块

```
``timescale 1ns / 1ps
`define UD #1
module div_clk
(
    input clk,//50M
    output   clk_10khz//0.01M
);
reg [19:0]cnt;

always @(posedge clk)
begin
    if(cnt == 20'd499_9)
        cnt<= `UD 9'd0;
    else
        cnt <= `UD cnt + 1'b1;
end

reg flag=1'b0;
always @(posedge clk)
begin
    if(cnt == 20'd259_9)
        flag <= `UD 1'b1;
    else if(cnt == 20'd499_9)
        flag <= `UD 1'b0;
end
assign clk_10khz = flag;

endmodule
```

4.4.5数码管显示模块

```
`timescale 1ns / 1ps
`define UD #1
module seq_control
(
    input [1:0]sel,
    input [3:0]key,
    output reg [3:0]dig,
    output reg [7:0]smg
);
/*==================================================
```

```
                        位选择映射
==========================================================*/
always @(*)
begin
    case(sel)
        2'd0:dig = 4'b0001;
        2'd1:dig = 4'b0010;
        2'd2:dig = 4'b0100;
        2'd3:dig = 4'b1000;
        default:dig = 4'b0000;
    endcase
end
//共阳极数码管，为 0 有效，即点亮
always @(*)
begin
    case(key)
        4'd0:smg = 8'b1000_0001;//"0"
        4'd1:smg = 8'b1100_1111;//"1"
        4'd2:smg = 8'b1001_0010;//"2"
        4'd3:smg = 8'b1000_0110;//"3"
        4'd4:smg = 8'b1100_1100;//"4"
        4'd5:smg = 8'b1010_0100;//"5"
        4'd6:smg = 8'b1010_0000;//"6"
        4'd7:smg = 8'b1000_1111;//"7"
        4'd8:smg = 8'b1000_0000;//"8"
        4'd9:smg = 8'b1000_0100;//"9"
        default:smg = 8'b1111_1111;
    endcase
end
endmodule
```

4.5实验现象

KEY0～7 分别控制数码管从右到左的数码管显示，按键 K0 控制右边起第一个数码管，按一下数字加 1，从 0 到 9，按键 K2 控制右边起第二个数码管，按一下数字加 1，从 0 到 9，按键 K3 控制右边起第三个数码管，按键 K4 控制第四个数码管，依次类推。

5.序列检测器

5.1实验目的

在连续信号中，检测是否包含特定序列，例如检测“1101”中是否包含“01”

5.2实验要求

1、 拨码开关 SW7-SW4 作为序列信号输入；

2、 KEY7-KEY6 作为特定信号输入序列，KEY 按下后对应的 LED 灯会亮起，表示对应位为 1，再按一下会熄灭，表示对应位为 0；

3、 K4 为序列检测开始和序列检测结束按键，初次按下 KEY4，开始检测，此时 LED4 也会被点亮，显示当前状态，再按一下停止检测，LED4 熄灭；结束后序列串中出现特定序列的次数显示在数码管上。

5.3实验原理

SW7～SW4 的状态为检测序列；

LED7～LED6 为特定序列；

数码管显示的结果为 LED[7:6]在 SW[7:4]中出现的次数；

5.4实验源码（完整源码查看 demo 源文件）

5.4.1方案设计

从实验目的分析此实验的实现需要有三个功能模块：

1、按键 LED 模块；

按键调整特定序列，由 KEY[7:6]控制特定序列值；KEY4 控制是否检测；输出用 LED 来显示及保存特定序列，同时也将特定序列与检测使能信号传递给检测模块；

2、序列对比模块；

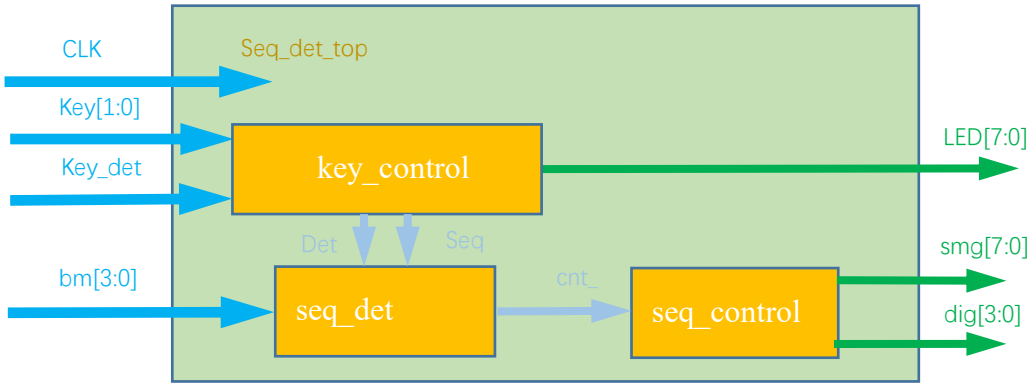
由拨码开关提供待检测序列，接收按键控制模块传递过来的特定序列与检测使能信号控制与待检测序列进行比较；比较结果输出给到数码管显示模块进行显示；

3、数码管控制模块

数码管显示模块的目标是将统计结果显示出来，用动态数码管显示的方式即

可；

对应模块之间的连线如下框图：



5.4.2顶层模块（含数码管显示模块）设计

```
`timescale 1ns / 1ps
`define UD #1
module top_seq_det
(
    input           clk,            // input clock    50M
    input           key_det,        // KEY 4 input control detected enable
    input [1:0] key_in,             // KEY[7:6] input control detected sequence
    input [3:0] bm,                 // DIP Switch[7:4] input used to detecting
    output          key_det_led,    // display the detecte status
    output [1:0]        key_in_led, // display the detected sequence
    output reg [3:0]        dig,    // output Digital tube Bit selection
    output reg [7:0]        smg     // output Digital tube segment selection
);

    /*==========================================================
                            按键消抖及状态标记
    ==========================================================*/
    wire [1:0]seq_data;
    key_control key_control
    (
        .clk                    (   clk                 ),
        .key_det            (   key_det             ),
        .key_in             (   key_in              ),
        .key_det_led        (   key_det_led         ),
        .key_in_led         (   key_in_led[1:0]     ),
        .seq_data           (   seq_data            )
    );
```

```
/*=============================================================
                        序列检测
=============================================================*/
wire [3:0]data;

seq_det seq_det (
    .clk            (  clk          ),
    .key_det_led    (  key_det_led ),//检测状态标记
    .key_in_led     (  seq_data     ),//待检测序列
    .bm             (  bm           ),//输入序列
    .data           (  data         )
);
/*=============================================================
                        时钟分频
=============================================================*/
wire clk_1khz;
div_clk div_clk(
    .clk            (  clk          ),
    .clk_1khz       (  clk_1khz     )
);
/*=============================================================
                        数码管显示
=============================================================*/
reg   [1:0]sel=0;
wire [3:0]dig0;
wire [7:0]smg0;

always @(posedge clk_1khz)
begin
    sel <= `UD sel+1'b1;
end
// Digital Tube 0 output
seq_control seq_control_0(
    .sel(sel),
    .key(data),
    .dig(dig0),
    .smg(smg0)
);
// Digital Tube 1 output
wire [3:0]dig1;
wire [7:0]smg1;
seq_control seq_control_1
(
```

```
        .sel(sel),
        .key(4'd0),
        .dig(dig1),
        .smg(smg1)
    );
    // Digital Tube 2 output
    wire [3:0]dig2;
    wire [7:0]smg2;
    seq_control seq_control_2
    (
        .sel(sel),
        .key(4'd0),
        .dig(dig2),
        .smg(smg2)
    );
    wire [3:0]dig3;
    wire [7:0]smg3; // Digital Tube 3 output
    seq_control seq_control_3
    (
        .sel(sel),
        .key(4'd0),
        .dig(dig3),
        .smg(smg3)
    );
    // display by Digital Tube
    always @(posedge clk_1khz)
    begin
        if(sel==2'b00)
            dig <= `UD ~dig0;
        else if(sel==2'b01)
            dig <= `UD ~dig1;
        else if(sel==2'b10)
            dig <= `UD ~dig2;
        else if(sel==2'b11)
            dig <= `UD ~dig3;
    end

    always @(posedge clk_1khz)
    begin
        if(sel==2'b00)
            smg <= `UD smg0;
        else if(sel==2'b01)
            smg <= `UD smg1;
        else if(sel==2'b10)
```

```
                smg <= `UD smg2;
            else if(sel==2'b11)
                smg <= `UD smg3;
        end

    endmodule
```

5.4.3按键 LED 控制模块

```
    `timescale 1ns / 1ps
    `define UD #1
    module key_control
    (
        input               clk,      // input clock
        input               key_det,        // KEY 4 input
        input [1:0] key_in,       // KEY[2:0] input
        output reg    key_det_led,    // LED3 control sigal
        output reg [1:0]      key_in_led,// LED[1:0] control signal
        output reg [1:0]      seq_data      // Sequence to be detected
    );

    //==按键消斗=========================================
        wire [1:0] key_out;
        wire          key_det_out;
        btn_deb#(
            .BTN_WIDTH      (    4'd2                          ) , //parameter
BTN_WIDTH = 4'd8
            .MS_20          (   20'd1_000_000           )
        ) btn_deb_key
        (
            .clk                (     clk                                 ),//input
clk,
            .btn_in         (   key_in                  ),//input [BTN_WIDTH-1:0]
btn_in,

            .btn_deb            (     key_out                       ) //output  reg
[BTN_WIDTH-1:0]btn_deb
        );

        btn_deb#(
            .BTN_WIDTH      (    4'd1                          ) , //parameter
BTN_WIDTH = 4'd8
```

```
        .MS_20          (   20'd1_000_000           )
    ) btn_deb_det
    (
        .clk            (   clk                     ),//input
clk,
        .btn_in         (   key_det                 ),//input
[BTN_WIDTH-1:0] btn_in,

        .btn_deb        (   key_det_out             ) //output reg
[BTN_WIDTH-1:0]btn_deb
    );

    reg [1:0]key_out_reg;
    reg key_det_out_reg;

    always @(posedge clk)
    begin
        key_out_reg <= `UD key_out;
        key_det_out_reg<= `UD key_det_out;
        key_det_led <= `UD key_8_flag;
    end

    reg key_8_flag = 1'b0;
    always @(posedge clk)
    begin
        if(!key_det_out && key_det_out_reg)
            key_8_flag <= `UD ~key_8_flag;
        else
            key_8_flag <= `UD key_8_flag;
    end

    reg [1:0]key_flag=2'b00;
    always @(posedge clk)
    begin
        if(~(~key_det_led || key_8_flag)) //检测结束，序列复位
            key_flag[0] <= `UD 1'b0;
        else if(!key_out[0] && key_out_reg[0])
            key_flag[0] <= `UD ~key_flag[0];
        else
            key_flag[0] <= `UD key_flag[0];
    end

    always @(posedge clk)
    begin
```

```
            if(~(~key_det_led || key_8_flag))//检测结束，序列复位
                key_flag[1] <= `UD 1'b0;
            else if(!key_out[1] && key_out_reg[1])
                key_flag[1] <= `UD ~key_flag[1];
            else
                key_flag[1] <= `UD key_flag[1];
        end

        always @(posedge clk)
        begin
            key_in_led <= `UD key_flag;
        end

        //seq_data
        always @(posedge clk)
        begin
            if(key_8_flag)
                seq_data <= `UD key_in_led;
        end

    endmodule
```

按键消抖

```
    `timescale 1ns / 1ps
    `define UD #1
    module btn_deb#(
        parameter                       BTN_WIDTH = 4'd8,
        parameter                       MS_20 = 20'd1_000_000
    )
    (
        input                           clk,
        input       [BTN_WIDTH-1:0] btn_in,

        output reg [BTN_WIDTH-1:0] btn_deb
    );

    //================================================================
        reg [19:0] time_cnt= 20'd0;
        always@(posedge clk)
        begin
            if(time_cnt == MS_20 - 1'b1)
                time_cnt <= 20'd0;
```

```
        else
            time_cnt <= time_cnt + 1'd1;
    end

    always @(posedge clk)
    begin
        if(time_cnt == MS_20 - 1'b1)
            btn_deb <= btn_in;
    end

endmodule
```

5.4.4序列检测模块设计

```
`timescale 1ns / 1ps
`define UD #1
module seq_det
(
    input               clk,
    input               key_det_led,//检测状态标记
    input [1:0]         key_in_led,//待检测序列
    input [3:0]         bm,//输入序列
    output reg [3:0]    data
);

    // 4bit data detecte 2 bit sequence，we need compare three numbers
    reg [2:0]flag=0;
    reg       det_en=0;

    reg key_det_led_1d=0;
    always @(posedge clk)
    begin
        key_det_led_1d <= `UD key_det_led;
    end

    always @(posedge clk)
    begin
        if(~key_det_led_1d && key_det_led) //检测按键触发后才进入
            det_en <= `UD 1'b1;
    end

    always @(posedge clk)
```

```
    begin
        if(key_det_led && det_en)
        begin
            flag[0] <= `UD (bm[3:2]==key_in_led);
            flag[1] <= `UD (bm[2:1]==key_in_led);
            flag[2] <= `UD (bm[1:0]==key_in_led);
        end
    end

    always @(posedge clk)
    begin
        if(~key_det_led && key_det_led_1d)//检测结束统计结果
            data <= `UD flag[2] + flag[1] + flag[0];
    end

endmodule
```

5.5实验现象

实验步骤：

1、按下轻触按键 KEY4，进入检测状态；

2、调整输入序列，更改拨码开关的输入值（SW[7：4]）；

3、调整固定序列，通过轻触按键改变 LED 状态（LED[7:6]）；

4、按下轻触按键 KEY4，退出检测，查看数码管显示的统计结果，重新执行前面三个步骤；

实验现象举例：

当 SW[7:4]=4’b1010;LED[7:6]=2’b01 时，按下 Key4 后数码管显示数字 1；

当 SW[7:4]=4’b1010;LED[7:6]=3’b10 时，按下 Key4 后数码管显示数字 2；

6.密码锁

6.1实验目的

利用盘古 PGX-Lite 7K 板卡上的按键，拨码开关以及数码管实现一种简单的密码锁；

6.2实验要求

利用拨码开关设置密码，使用按键输入开锁密码。当开锁密码与设定密码相同时开锁成功，数码管显示 8888，密码错误时显示 7777。

SW0~SW3 设置 2 位数密码，每两位设置一位密码，SW[1:0]设置第一位数据对应的二进制数值，SW[3:2]设置第二位数据对应的二进制数值。所以密码是由 0，1，2，3 组成的四位数。

KEY1-KEY0 作为密码输入，按键按一下数字加 1，数字由数码管显示，数字在 0，1，2，3 中循环。

KEY2 作为确认按键，按下 KEY2，输入的密码与设置的密码比对，如相同则显示 8888，若不同则显示 7777。按下 KEY3 清零，按下后数码管显示 0000，可以重新输密码。

6.3实验原理

原理上与前一个章节的序列检测是类似的，在前一个实验的基础上有了一些延伸；

序列对比的位宽发生改变，单个数据占 2bit，一个按键控制输入密码数据设置为 2bit 即可；对比与重新开始在此实验用两个按键实现，一个确认对比，一个清空结果；

6.4实验源码（完整源码查看 demo 源文件）

根据需求我们需要如下三个子模块：

①按键控制模块；

1、对 4 个按键输入信号均做消抖处理

2、KEY3 和 KEY2 取下降沿输出

3、KEY[1：0]以下降沿来变更各自的输入密码，每次数字加 1（0～3 循环，

2bit 即可）

②数码管显示模块；

显示状态有两种：

密码输入状态：

1、上电默认状态； 2、KEY3 下降沿触发进入重置状态；3、实时显示 2 位输入密码；

密码验证状态：

1、KEY2 下降沿触发进入；

2、显示密码验证结果，正确则显示 8888，错误则显示 7777；

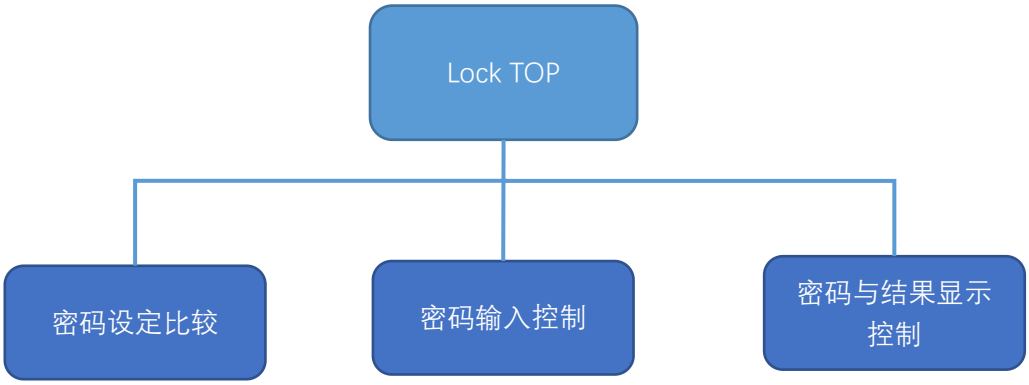
③密码验证模块；

KEY2 下降沿触发使能工作； KEY2 下降沿触发所存输入密码，并与拨码开关设置的密码进行比较；

输出密码比较结果，提供个数码管显示模块。

6.4.1顶层模块设计

顶层模块与上述三个模块之间的关系如下图：



输入输出信号如下表：

信号	位宽	方向	描述
clk	1	输入	外部输入时钟，输入时钟为 50MHz
key	2	输入	轻触按键输入信号， K0~K1 输入
enter	1	输入	密码确认比对信号， K2 输入
init	1	输入	密码确认重新输入信号， K3 输入
sw	4	输入	密码设置输入信号， SW0~3 输入
smg	8	输出	密码对比结果显示数码管段选信号输出
dig	4	输出	密码对比结果显示数码管位选信号输出

Module 设计如下：

```
`timescale 1ns / 1ps
`define UD #1
module lock_top(
    input               clk,
    input   [1:0]       key,
    input               enter,
    input               init,
    input   [3:0]       sw,

    output [7:0]        smg,
    output [3:0]        dig
);

    wire                enter_trig;
    wire                init_trig;
    wire [3:0]          ctrl;
    wire                com_result;

    key_ctl key_ctl(
        .clk            (   clk         ),//input             clk,
        .key            (   key         ),//input       [1:0] key,
        .enter          (   enter       ),//input         enter,
        .init           (   init        ),//input          init,

        .enter_trig     (   enter_trig  ),//output      enter_trig,
        .init_trig      (   init_trig   ),//output       init_trig,
        .ctrl           (   ctrl        ) //output      [3:0] ctrl
    );

    compare compare(
        .clk            (   clk         ),//input         clk,
        .sw             (   sw          ),//input [3:0]   sw,
        .ctrl           (   ctrl        ),//input [3:0]   ctrl,
        .enter_trig     (   enter_trig  ),//input         enter_trig,
        .com_result     (   com_result  ) //output        com_result
    );

    seq_display seq_display(
        .clk                            (   clk             ),//input
clk,
        .enter_trig     (   enter_trig  ),//input             enter_trig,
        .init_trig      (   init_trig   ),//input              init_trig,
```

```
                .com_result                 (       com_result      ),//input
com_result,
                .ctrl                       (   ctrl        ),//input       [3:0]
ctrl,

                .smg            (   smg         ),//output reg   [7:0]    smg,
                .dig            (   dig         ) //output reg   [3:0]    dig
            );

    endmodule
```

6.4.2按键控制设计

```
    `timescale 1ns / 1ps
    `define UD #1
    module key_ctl(
        input           clk,
        input   [1:0] key,
        input           enter,
        input           init,

        output          enter_trig,
        output          init_trig,
        output  [3:0] ctrl
    );

        wire [3:0] btn_deb;
        // 按键消抖
        btn_deb#(
            .BTN_WIDTH      (   4'd4                    ) , //parameter
BTN_WIDTH = 4'd8
            .MS_20          (   20'd1_000_000           )
        ) btn_deb_key
        (
            .clk                (   clk                     ),//input
clk,
            .btn_in         (   {enter,init,key}        ),//input [BTN_WIDTH-1:0]
btn_in,
            .btn_deb        (   btn_deb                 ) //output reg
[BTN_WIDTH-1:0]btn_deb
        );
```

```
    reg [1:0]   key1_push_cnt=2'd0;
    reg [1:0]   key2_push_cnt=2'd0;

    reg btn1_deb_1d,btn2_deb_1d;
    reg enter_deb_1d,init_deb_1d;

    assign enter_trig = ~btn_deb[3] & enter_deb_1d;
    assign init_trig = ~btn_deb[2] & init_deb_1d;

    always @(posedge clk)
    begin
        btn1_deb_1d   <= `UD btn_deb[0];
        btn2_deb_1d   <= `UD btn_deb[1];
        init_deb_1d   <= `UD btn_deb[2];
        enter_deb_1d <= `UD btn_deb[3];
    end

    always @(posedge clk)
    begin
        if(~btn_deb[2] & init_deb_1d)
            key1_push_cnt <= `UD 2'd0;
        else if(~btn_deb[0] & btn1_deb_1d)
        begin
            key1_push_cnt <= `UD key1_push_cnt + 2'd1;
        end
    end

    always @(posedge clk)
    begin
        if(~btn_deb[2] & init_deb_1d)
            key2_push_cnt <= `UD 2'd0;
        else if(~btn_deb[1] & btn2_deb_1d)
        begin
            key2_push_cnt <= `UD key2_push_cnt + 2'd1;
        end
    end

    assign ctrl = {key2_push_cnt,key1_push_cnt};

endmodule
```

6.4.3按键消抖设计

```
`timescale 1ns / 1ps
`define UD #1
module btn_deb#(
    parameter                       BTN_WIDTH = 4'd8,
    parameter                       MS_20 = 20'd1_000_000
)
(
    input                           clk,//50MHz
    input       [BTN_WIDTH-1:0] btn_in,

    output reg [BTN_WIDTH-1:0] btn_deb
);
//=============================================================
============================
    reg [19:0] time_cnt= 20'd0;
    always@(posedge clk)
    begin
        if(time_cnt == MS_20 - 1'b1)
            time_cnt <= 20'd0;
        else
            time_cnt <= time_cnt + 1'd1;
    end

    always @(posedge clk)
    begin
        if(time_cnt == MS_20 - 1'b1)
            btn_deb <= btn_in;
    end

endmodule
```

6.4.4对比模块设计

```
`timescale 1ns / 1ps
`define UD #1
module compare(
    input           clk,
    input [3:0]   sw,
    input [3:0]   ctrl,
    input           enter_trig,
    output          com_result
```

```
    );
    //=============================================================
=======================
        //锁存当前的输入密码；
        reg [3:0] ctrl_1d;
        always @(posedge clk)
        begin
            if(enter_trig)
                ctrl_1d <= `UD ctrl;
        end
        assign com_result = (ctrl_1d == sw);

    endmodule
```

6.4.5显示模块设计

此模块设计需要注意数码管显示的两种模式：密码输入模式与密码对比结果显示模式；两种模式的切换由 enter_trig 与 init_trig 触发进入；

对于数码管的显示控制模块这里就不重复描述了；

6.5实验现象

验证步骤：

1、 调整输入序列，更改拨码开关的输入值（SW[3：0]）；

2、 调整固定序列，通过轻触按键 KEY1~KEY0 调整输入密码，数码管实时显示输入密码；

3、 按下轻触按键 KEY2，触发进行密码比对，并且数码管显示比对结果；

4、 按下轻触按键 KEY3，进入重新输入密码状态，重新执行前面三个步骤；

实验现象

当 SW[3:0]=8’b1010;当输入密码状态时显示 0022 时，按下 Key2 后数码管显示数字 8888；当输入密码状态时显示不是 0022 时，按下 Key2 后数码管显示数字 7777；按下 Key3 后重新调整密码，进入输入密码状态；

7.数字钟

7.1实验目的

设计一个具有计时功能和校时功能的数字时钟。

7.2实验要求

数码管显示小时和分钟，秒钟用 LED 闪烁标识。

三个按键用于时钟校准。

K0 用于切换正常计时，校准小时和分钟

K1 用于时钟的“+”

K2 用于时钟的“-”

校准相应的刻度，该数码管闪烁。

7.3实验原理

从上述的实验要求分析可得到此数字钟我们实现过程中要注意两个功能点：

1、计时显示功能：LED 闪烁显示秒钟读秒，数码管右侧两位显示分钟计时，数码管左侧两位显示时钟计时；

此功能的实现由两个细节功能实现：①1S 计时控制，与前面的实验中需要计时功能模块实现方式一致，注意此处计时的周期为 1S 即可；②计时过程中进位控制；进位控制有四处需要进位：

秒 进 分	LED 灯亮灭一次计数为 1，亮灭的一个周期为 1S,当 LED 亮灭 60 次时分钟计数个位加 1；
分$_{个位}$ 进 分$_{十位}$	当个位计数到 9 时，秒到分再次进位时，十位需要加一，个位置位为 0；
分进时	1 小时=60 分钟；当分钟计时为 59，并且秒到分再次进位时，时钟计数个位加 1；分钟计数置位为 0；
时$_{个位}$ 进 时$_{十位}$	显示为 24 小时制，当时钟计数十位小于 2 时，时钟个位个位计数到 9 时，分到时再次进位时，十位加 1；
24 小时制计满时归零	当时钟计数十位等于 2 时，时钟个位个位计数到 3 时，分到时再次进位时，时，分，秒三部分计数均归零；

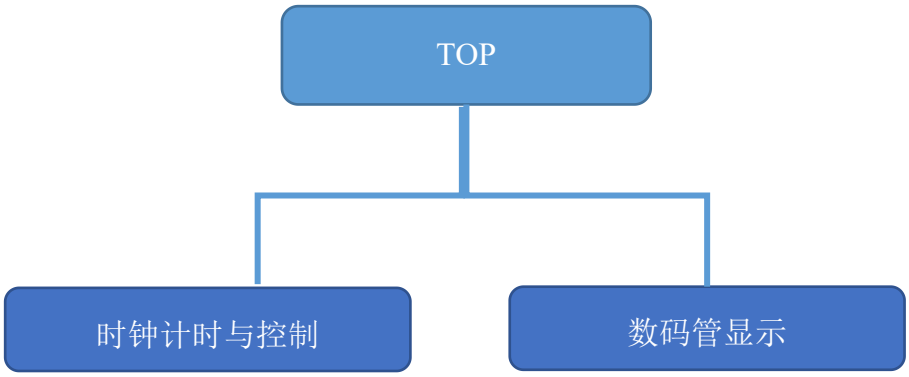
2、计时校准功能：通过对应按键控制调整分钟计时与时钟计时，调整的过

程中对应位需要闪烁；

此项功能中注意两点：①调整对应位是，数码管该位进行闪烁；②调整时注意进位；

基于上述分析我们将项目分成两个部分：

1. 时钟计时与控制。
2. 数码管显示控制。



7.4实验源码（完整源码查看 demo 源文件）

7.4.1顶层设计

输入输出信号如下表：

信号	位宽	方向	描述
clk	1	输入	外部输入时钟，PGX-Lite 7K 板卡输入时钟为 50MHz
key	3	输入	时钟校准信号输入（轻触按键）
led	1	输出	时钟秒钟跳动显示（LED 灯闪烁一次，为 1S）
smg	8	输出	数码管段选输出
dig	4	输出	数码管位选输出

```
`timescale 1ns / 1ps
`define UD #1
module top_watch
(
    input clk,
    input [2:0]key,
```

```
    output led,
    output reg [3:0]dig,
    output reg [7:0]smg
);

  parameter CLK_FRE = 26'd50_000_000;
/*=============================================================
                        复位信号的产生
=============================================================*/
reg [4:0] rstn_cnt=0;
always @(posedge clk)
begin
    if(rstn_cnt==5'h1f)
        rstn_cnt <= `UD rstn_cnt;
    else
        rstn_cnt <= `UD rstn_cnt + 1'b1;
end

wire rstn;
assign rstn = rstn_cnt[4];
/*=============================================================
                            数字时钟的产生和控制
=============================================================*/
wire [3:0] hour_h_o,hour_l_o,minutes_h_o,minutes_l_o;
wire [2:0] state_flag;
watch_data_gen #(
    .CLK_FRE      (CLK_FRE      )
) u_watch_data_gen
(
    .clk            (clk            ), //input clk,
    .rstn           (rstn           ), //input rstn,
    .key            (key            ), //input [2:0]key,
    .hour_h_o       (hour_h_o       ), //output reg [3:0]hour_h_o,
    .hour_l_o       (hour_l_o       ), //output reg [3:0]hour_l_o,
    .minutes_h_o(minutes_h_o), //output reg [3:0]minutes_h_o,
    .minutes_l_o(minutes_l_o), //output reg [3:0]minutes_l_o,
    .second_led (led            ), //output reg second_led,
    .state_flag (state_flag )    //output reg [2:0]state_flag
);
/*=============================================================
                            时钟分频
=============================================================*/
wire clk_10khz;
div_clk #(
```

```
        .CLK_FRE       (CLK_FRE       )
    )   div_clk
    (
        .clk            (clk),
        .clk_10khz (clk_10khz)
    );
    /*==========================================================
                                数码管显示
    ==========================================================*/
    reg   [1:0]sel=0;
    wire [3:0]dig0;
    wire [7:0]smg0;

    always @(posedge clk_10khz)
    begin
        sel <= `UD sel+1'b1;
    end

    seq_control seq_control_0
    (
        .clk(clk),
        .sec_en(led),
        .control_dig(state_flag),
        .sel(2'd0),
        .key(minutes_l_o),
        .dig(dig0),
        .smg(smg0)
    );

    wire [3:0]dig1;
    wire [7:0]smg1;

    seq_control seq_control_1
    (
        .clk(clk),
        .sec_en(led),
        .control_dig(state_flag),
        .sel(2'd1),
        .key(minutes_h_o),
        .dig(dig1),
        .smg(smg1)
    );

    wire [3:0]dig2;
```

```
wire [7:0]smg2;

seq_control seq_control_2
(
        .clk(clk),
        .sec_en(led),
        .control_dig(state_flag),
        .sel(2'd2),
        .key(hour_l_o),
        .dig(dig2),
        .smg(smg2)
);

wire [3:0]dig3;
wire [7:0]smg3;

seq_control seq_control_3
(
        .clk(clk),
        .sec_en(led),
        .control_dig(state_flag),
        .sel(2'd3),
        .key(hour_h_o),
        .dig(dig3),
        .smg(smg3)
);


always @(posedge clk_10khz)
begin
        if(sel==2'b00)
                dig <= `UD ~dig0;
        else if(sel==2'b01)
                dig <= `UD ~dig1;
        else if(sel==2'b10)
                dig <= `UD ~dig2;
        else if(sel==2'b11)
                dig <= `UD ~dig3;
end


always @(posedge clk_10khz)
begin
        if(sel==2'b00)
```

```
            smg <= `UD smg0;
        else if(sel==2'b01)
            smg <= `UD smg1;
        else if(sel==2'b10)
            smg <= `UD smg2;
        else if(sel==2'b11)
            smg <= `UD smg3;
    end

    endmodule
```

7.4.2数字时钟产生与控制模块设计

在此模块中我们要实现前面描述的两个主要的功能点：计时与控制；

输入输出信号如下表：

信号	位宽	方向	描述
clk	1	输入	外部输入时钟，PGX-Lite 7K 板卡输入时钟为 50MHz
rstn	1	输入	外部输入复位信号，
key	3	输入	时钟校准信号输入（轻触按键）
hour_h_o	4	输出	时钟高位计数
hour_l_o	4	输出	时钟低位计数
minutes_h_o	4	输出	分钟高位计数
minutes_l_o	4	输出	分钟低位计数
second_led	1	输出	时钟秒钟跳动显示（LED 灯闪烁一次，为 1S）
state_flag	3	输出	数码管段选输出

Module 设计的关键点如下（完整 module 查看源文件）：

```
`timescale 1ns / 1ps
`define UD #1
module watch_data_gen #(
    parameter CLK_FRE = 26'd50_000_000
)
(
    input clk,
    input rstn,
    input [2:0]key,
    output reg [3:0]hour_h_o,
```

```
        output reg [3:0]hour_l_o,
        output reg [3:0]minutes_h_o,
        output reg [3:0]minutes_l_o,
        output reg second_led,
        output reg [2:0]state_flag
);

/*==========================================================
                        基准的产生
==========================================================*/
// 1s= 20ns * 50_000_000;
reg [25:0]second_cnt;
always @(posedge clk)
begin
    if(second_cnt==CLK_FRE-1'b1)
        second_cnt <= `UD 26'd0;
    else
        second_cnt <= `UD second_cnt + 1'b1;
end
//每个 1s 闪烁一次
always @(posedge clk)
begin
    if(!rstn)
        second_led <= `UD 1'b0;
    if(second_cnt==(CLK_FRE>>1)-1'b1)
        second_led <= `UD 1'b1;
    else if(second_cnt==CLK_FRE-1'b1)
        second_led <= `UD 1'b0;
end

reg [3:0]minutes_h;
reg [3:0]minutes_l;
reg [3:0]hour_h;
reg [3:0]hour_l;

reg [3:0]minutes_l_fix=0;
reg [3:0]minutes_h_fix=0;
reg [3:0]hour_l_fix=0;
reg [3:0]hour_h_fix=0;
/*==========================================================
                    校准逻辑产生
==========================================================*/
wire [2:0]key_out;
```

```
    btn_deb #(
        .BT_WIDTH(4'd3),
        .CLK_FRE (CLK_FRE)
    )
    u_btn_deb
    (
          .clk       (clk),
          .btn_in (key),
          .btn_out(key_out)
    );
    /*=====================================================
          //key[0] -> k1 ;用于校准标记
          key_cnt = 3'd0 用于正常显示；
          key_cnt = 3'd1 用于分钟低位校准；
          key_cnt = 3'd2 用于分钟高位校准；
          key_cnt = 3'd3 用于时钟低位校准；
          key_cnt = 3'd4 用于时钟高位校准；
    =====================================================*/
    reg [2:0]key_out_reg=3'd0;
    always @(posedge clk)
    begin
          key_out_reg <= `UD key_out;
    end

    reg [2:0]key_cnt=3'd0;
    always @(posedge clk)
    begin
          if(key_cnt==3'd4 && (!key_out[0] && key_out_reg[0]))
                key_cnt <= `UD 3'd0;
          else if(!key_out[0] && key_out_reg[0])
                key_cnt <= `UD key_cnt + 1'b1;
    end
    /*=====================================================
                      key[1] 用于"+"；key[2] 用"-"
    =====================================================*/
    always @(posedge clk)
    begin
          if(key_cnt==3'd0)//校准前将分钟低位和输出值保持一致
                minutes_1_fix <= `UD minutes_1;
          else   if(!key_out[1]   &&   key_out_reg[1]   &&   key_cnt==3'd1   &&
minutes_1_fix == 4'd9)//当处于分钟校准状态时，按下"+"按键,且此时校准值已经为 9 时，再按下按键，则校准值变为 0
                minutes_1_fix <= `UD 4'd0;//"+"
          else   if(!key_out[2]   &&   key_out_reg[2]   &&   key_cnt==3'd1   &&
```

```
minutes_l_fix == 4'd0)//当处于分钟校准状态时，按下"-"按键,且此时校准值已经为 0 时，再按下按键，则校准值变为 9
                minutes_l_fix <= `UD 4'd9;//"-"
            else if(!key_out[1] && key_out_reg[1] && key_cnt==3'd1)//当处于分钟低位校准状态时，按下"+"按键,校准数值加 1；
                minutes_l_fix <= `UD minutes_l_fix + 1'b1;//"+"
            else if(!key_out[2] && key_out_reg[2] && key_cnt==3'd1)//当处于分钟低位校准状态时，按下"-"按键,校准数值减 1；
                minutes_l_fix <= `UD minutes_l_fix - 1'b1;   //"-"
        end


        always @(posedge clk)
        begin
            if(key_cnt!=3'd2)//校准前数据和输出值保持一致
                minutes_h_fix <= `UD minutes_h;
            else   if(!key_out[1]   &&   key_out_reg[1]   &&   key_cnt==3'd2   &&   minutes_h_fix == 4'd5)//当处于校准状态时，按下"+"按键,且此时校准值已经为 5 时，再按下按键，则校准值变为 0
                minutes_h_fix <= `UD 4'd0;//"+"
            else   if(!key_out[2]   &&   key_out_reg[2]   &&   key_cnt==3'd2   &&   minutes_h_fix == 4'd0)//当处于分钟校准状态时，按下"-"按键,且此时校准值已经为 0 时，再按下按键，则校准值变为 5
                minutes_h_fix <= `UD 4'd5;//"-"
            else if(!key_out[1] && key_out_reg[1] && key_cnt==3'd2)//当处于校准状态时，按下"+"按键,按下"+"按键,校准数值加 1；
                minutes_h_fix <= `UD minutes_h_fix + 1'b1;//"+"
            else if(!key_out[2] && key_out_reg[2] && key_cnt==3'd2)//当处于校准状态时，按下"-"按键,校准数值减 1；
                minutes_h_fix <= `UD minutes_h_fix - 1'b1;//"-"
        end


        always @(posedge clk)
        begin
            if(key_cnt!=3'd3 )//校准前数据赋值为 0
                hour_l_fix <= `UD 4'd0;
            else   if(!key_out[1]   &&   key_out_reg[1]   &&   key_cnt==3'd3   &&   hour_h_o != 4'd2 && hour_l_fix==4'd9)//当处于校准状态时，按下"+"按键,且此时小时的高位不为 2 且校准值已经为 9 时，再按下按键，则校准值变为 0
                hour_l_fix <= `UD 4'd0;//"+"
            else if(!key_out[1] && key_out_reg[1] && key_cnt==3'd3 && hour_h_o == 4'd2 && hour_l_fix==4'd3)//当处于校准状态时，按下"+"按键,且此时小时的高
```

```
位为 2 且校准值已经为 3 时，再按下按键，则校准值变为 0
                hour_l_fix <= `UD 4'd0;//"+"
            else  if(!key_out[2]  &&  key_out_reg[2]  &&  key_cnt==3'd3  &&
hour_h_o != 4'd2 && hour_l_fix==4'd0)//当处于校准状态时，按下"-"按键,且此时
小时的高位不为 2 时，校准位为 0，再按下按键，则校准值变为,9
                hour_l_fix <= `UD 4'd9;//"-"
            else if(!key_out[2] && key_out_reg[2] && key_cnt==3'd3 && hour_h_o
== 4'd2 && hour_l_fix==4'd0)//当处于校准状态时，按下"-"按键,且此时小时的高
位为 2 时，校准位为 0，再按下按键，则校准值变为 3
                hour_l_fix <= `UD 4'd3;//"-"
            else if(!key_out[1] && key_out_reg[1] && key_cnt==3'd3)//当处于校准
状态时，按下"+"按键,按下"+"按键,校准数值加 1；
                hour_l_fix <= `UD hour_l_fix + 1'b1;//"+"
            else if(!key_out[2] && key_out_reg[2] && key_cnt==3'd3)//当处于校准
状态时，按下"-"按键,校准数值减 1；
                hour_l_fix <= `UD hour_l_fix - 1'b1;
        end


        always @(posedge clk)
        begin
            if(key_cnt!=3'd4)//校准前数据和输出值保持一致
                hour_h_fix <= `UD hour_h;
            else if(!key_out[1] && key_out_reg[1] && key_cnt==3'd4 && hour_h_fix
== 4'd2)//当处于校准状态时，按下"+"按键,且此时小时的高位为 2，再按下按键，
则校准值变为 0
                hour_h_fix <= `UD 4'd0;//"+"
            else if(!key_out[2] && key_out_reg[2] && key_cnt==3'd4 && hour_h_fix
== 4'd0)//当处于校准状态时，按下"-"按键,且此时小时的高位为 0，再按下按键，
则校准值变为 2
                hour_h_fix <= `UD 4'd2;//"-"
            else if(!key_out[1] && key_out_reg[1] && key_cnt==3'd4 )//当处于校准
状态时，按下"+"按键,按下"+"按键,校准数值加 1；
                hour_h_fix <= `UD hour_h_fix + 1'b1;//"+"
            else if(!key_out[2] && key_out_reg[2] && key_cnt==3'd4)//当处于校准
状态时，按下"-"按键,校准数值减 1；
                hour_h_fix <= `UD hour_h_fix - 1'b1;//"-"
        end
        /*=====================================================
                                秒钟的产生:中间值
        =====================================================*/
        //秒钟计 60 次（0~59）为 1 分钟
        reg [5:0]second_flag=0;
        always @(posedge clk)
```

```
    begin
        if(second_cnt==CLK_FRE-1'b1 && second_flag==6'd59)
            second_flag <=`UD 6'd0;
        else if(second_cnt==CLK_FRE-1'b1)
            second_flag <=`UD second_flag + 1'b1;
    end
    /*==========================================================
                            分钟的产生:中间值
    ==========================================================*/
    //minutes_l gen
    always @(posedge clk)
    begin
        if(!rstn)//初始值为 0
            minutes_l <= `UD 4'd0;
        else if(key_cnt==3'd1)//校准时，分钟低位为校准值
            minutes_l <= `UD minutes_l_fix;
        else    if(second_cnt==CLK_FRE-1'b1    &&    second_flag==6'd59    &&
minutes_l==4'd9)//9 分 59 秒产生进位，低位赋值为 0
            minutes_l <= `UD 4'd0;
        else if(second_cnt==CLK_FRE-1'b1 && second_flag==6'd59)//60 秒产生
分钟的低位进位
            minutes_l <= `UD minutes_l +1'b1;
    end
    //minutes_h gen
    always @(posedge clk)
    begin
        if(!rstn)
            minutes_h <= `UD 4'd0;
        else if(key_cnt==3'd2)
            minutes_h <= `UD minutes_h_fix;
        else    if(second_cnt==CLK_FRE-1'b1    &&    second_flag==6'd59    &&
minutes_h==4'd5    && minutes_l==4'd9)//当为 59 分 59 秒的时候产生进位，分钟
的高位赋值为 0；
            minutes_h <= `UD 4'd0;
        else    if(second_cnt==CLK_FRE-1'b1    &&    second_flag==6'd59    &&
minutes_l==4'd9 )//9 分 59 秒的时候产生进位；
            minutes_h <= `UD minutes_h +1'b1;
    end
    /*==========================================================
                            小时的产生:中间值
    ==========================================================*/
    always @(posedge clk)
    begin
        if(!rstn)
```

```
                hour_l <= `UD 4'd0;
            else if(key_cnt==3'd3)
                hour_l <= `UD hour_l_fix;
            else if(hour_h!=4'd2 && hour_l==4'd9 && second_cnt==CLK_FRE-1'b1
&& second_flag==6'd59 && minutes_h==4'd5 && minutes_l==4'd9 )//9:59:59 或者
19:59:59，下一秒 hour_l 为 0；
                hour_l <= `UD 4'd0;
            else if(hour_h==4'd2 && hour_l==4'd3 && second_cnt==CLK_FRE-1'b1
&&          second_flag==6'd59              &&          minutes_h==4'd5          &&
minutes_l==4'd9 )//23:59:59;hour_l 为 0；
                hour_l <= `UD 4'd0;
            else     if(second_cnt==CLK_FRE-1'b1     &&     minutes_h==4'd5     &&
minutes_l==4'd9 && second_flag==6'd59)//XX:59:59;hour_l 加 1；
                hour_l <= `UD hour_l +1'b1;
        end

        always @(posedge clk)
        begin
            if(!rstn)
                hour_h <= `UD 4'd0;
            else if(key_cnt==3'd4)
                hour_h <= `UD hour_h_fix;
            else if(hour_h==4'd2 && hour_l==4'd3 && second_cnt==CLK_FRE-1'b1
&& minutes_h==4'd5 && minutes_l==4'd9 && second_flag==6'd59)//当时间刻度
为：23:59:59 ,hour_h 为 0
                hour_h <= `UD 4'd0;
            else      if(hour_l==4'd9      &&      second_cnt==CLK_FRE-1'b1      &&
minutes_h==4'd5 && minutes_l==4'd9 && second_flag==6'd59)//当时间刻度为：
09:59:59 or 19:59:59,hour_h 加 1
                hour_h <= `UD hour_h + 1'b1;
        end

        /*=====================================================
                            output
        =====================================================*/
        /*=====================================================
            //key[0] -> k1 ;用于校准标记
            key_cnt = 3'd0 用于正常显示；
            key_cnt = 3'd1 用于分钟低位校准；
            key_cnt = 3'd2 用于分钟高位校准；
            key_cnt = 3'd3 用于时钟低位校准；
            key_cnt = 3'd4 用于时钟高位校准；
        =====================================================*/
        always @(posedge clk)
```

```
begin
    minutes_l_o <=`UD minutes_l;
end

always @(posedge clk)
begin
    minutes_h_o <=`UD minutes_h;
end

always @(posedge clk)
begin
    hour_l_o <=`UD hour_l;
end

always @(posedge clk)
begin
    hour_h_o <=`UD hour_h;
end

always @(posedge clk)
begin
    state_flag <= `UD key_cnt;
end

endmodule
```

7.4.3时钟分频模块

```
`timescale 1ns / 1ps
`define UD #1
module div_clk #(
    parameter       CLK_FRE    = 26'd40_000_000
)
(
    input clk,
    output    clk_10khz
);
//times =20ns*5000=100us=10khz;
parameter CLK_DIV_100US = CLK_FRE/50_00;
reg [15:0]cnt;
always @(posedge clk)
begin
    if(cnt == CLK_DIV_100US - 1'b1)
```

```
            cnt<= `UD 16'd0;
        else
            cnt <= `UD cnt + 1'b1;
    end
    reg flag=1'b0;
    always @(posedge clk)
    begin
        if(cnt == (CLK_DIV_100US>>1) - 1'b1)
            flag <= `UD 1'b1;
        else if(cnt == CLK_DIV_100US - 1'b1)
            flag <= `UD 1'b0;
    end
    assign clk_10khz = flag;
    endmodule
```

7.4.4数码管显示模块设计

数码管显示模块相比前一个实验需要增加一个功能：当进入校准模式时数码管的校准位需要进行闪烁，故而引入一个 1S 的周期信号，在 1S 时间内 0.5s 正常点亮，0.5s 不点亮使得数码管闪烁；闪烁对应位需要引入按键控制输出的 dig_ctl 信号（前面代码中有描述）；

闪烁控制的模块设计如下：

```
    always @(*)
    begin
        case(control_dig)
            4'd0: //正常显示
                case(sel)
                    2'd0:dig = 4'b0001;
                    2'd1:dig = 4'b0010;
                    2'd2:dig = 4'b0100;
                    2'd3:dig = 4'b1000;
                default:dig = 4'b0000;
                endcase
            4'd1://时钟高位校准
                case(sel)
                    2'd0:begin if(sec_en) dig = 4'b0001 ;else dig = 4'b0000; end
                    2'd1:dig = 4'b0010;
                    2'd2:dig = 4'b0100;
                    2'd3:dig = 4'b1000;
                default:dig = 4'b0000;
                endcase
            4'd2://时钟低位校准
```

```
                case(sel)
                    2'd0:dig = 4'b0001;
                    2'd1:begin if(sec_en) dig = 4'b0010 ;else dig = 4'b0000; end
                    2'd2:dig = 4'b0100;
                    2'd3:dig = 4'b1000;
                default:dig = 4'b0000;
                endcase
            4'd3: //分钟高位校准
                case(sel)
                    2'd0:dig = 4'b0001;
                    2'd1:dig = 4'b0010;
                    2'd2:begin if(sec_en) dig = 4'b0100 ;else dig = 4'b0000; end
                    2'd3:dig = 4'b1000;
                default:dig = 4'b0000;
                endcase
            4'd4: //分钟低位校准
                case(sel)
                    2'd0:dig = 4'b0001;
                    2'd1:dig = 4'b0010;
                    2'd2:dig = 4'b0100;
                    2'd3:begin if(sec_en) dig = 4'b1000 ;else dig = 4'b0000; end
                default:dig = 4'b0000;
                endcase
            default:dig = 4'b0000;
        endcase
    end
```

7.5实验现象

加载后的显示结果为：数码管显示从 00：00 开始，LED1 闪烁（1 次/s）；

按轻触按键 KEY0，进入校准模式，第一次按下 KEY1，进入分钟低位计数校准调节，之后再次按下 KEY0，校准位将会往左移动 1 位，直到校准位为时钟计数高位时，按下 KEY0 将推出校准模式，进入正常计数模式；

在校准模式中按下轻触按键 KEY1 一次，对应校准位加 1，在可计数的最大值时会归 0；

在校准模式中按下轻触按键 KEY2 一次，对应校准位减 1，在减到 0 时会置位为可计数的最大值；

8.串口收发实验

8.1实验目的

盘古 PGX-Lite 7K 开发板集成了一路 USB 转串口模块，通过使用一根 USB Type C 线连接到 PC 端 USB 接口进行串口通信。

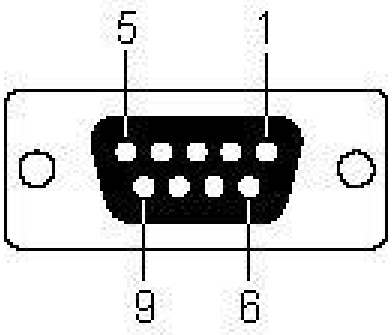
8.2实验要求

串口通信时波特率设置为 115200bps，数据格式为 1 位起始位、8 位数据位、无校验位、1 位结束位。板子每隔 1s 向串口助手发送一次 ASCII 码显示的“www.meyesemi.com”，通过串口助手向板子以十六进制形式发送数字（00~FF），LED 以二进制对应数值显示亮起。

8.3 实验原理

8.3.1串口原理

从下图我们可以看到标准串口接口是 9 根线，具体含义如下：



数据线：

TXD（pin 3）：串口数据输出(Transmit Data)

RXD（pin 2）：串口数据输入(Receive Data)

握手：

RTS（pin 7）：发送数据请求(Request to Send)

CTS（pin 8）：清除发送(Clear to Send)

DSR（pin 6）：数据发送就绪(Data Send Ready)

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

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

地线：

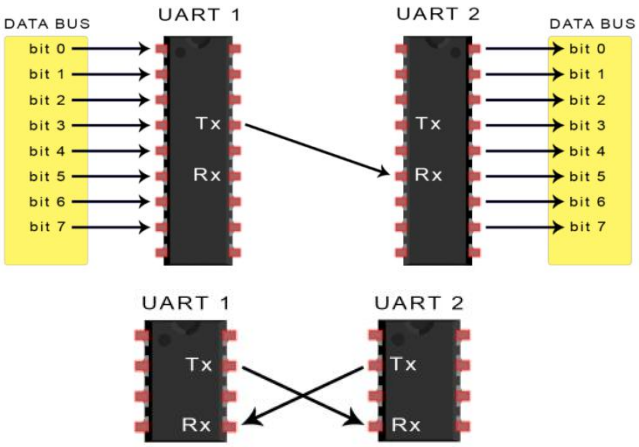
GND（pin 5）：地线

其它：

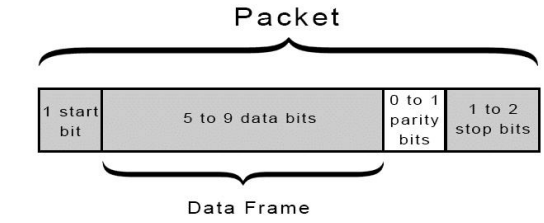
RI（pin 9）：铃声指示

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

RS232 的串口连接方式：



串口传输协议如下：



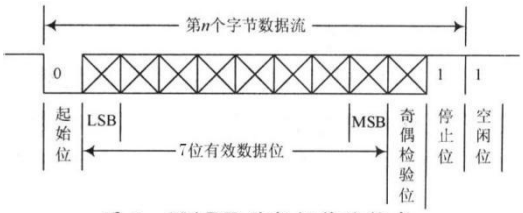


图 1　UART 的数据传输格式

起始位：先发出一个逻辑”0”信号，表示传输字符的开始。

数据位：可以是 5~8 位逻辑”0”或”1”。如 ASCII 码（7 位），扩展 BCD 码（8 位）。

校验位：数据位加上这一位后，使得“1”的位数应为偶数(偶校验)或奇数(奇

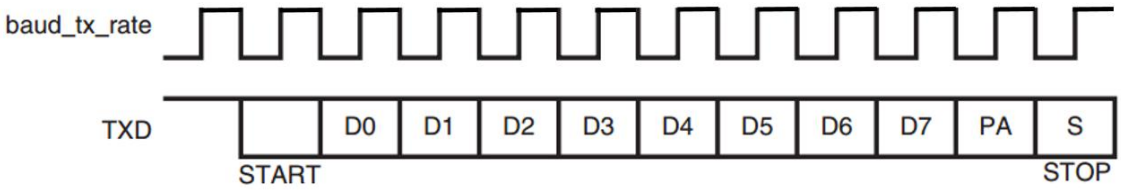
校验)。

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

空闲位：处于逻辑“1”状态，表示当前线路上没有资料传送。

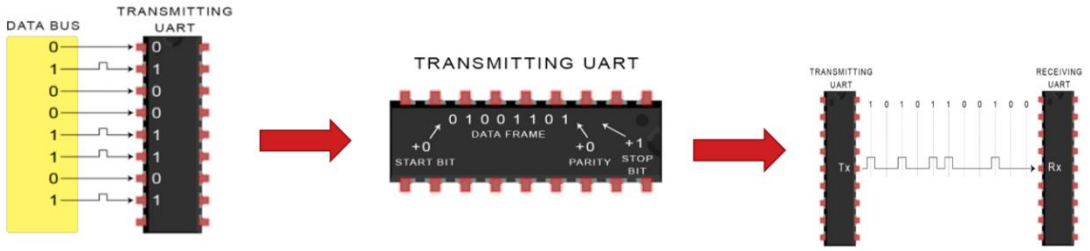
波特率：uart 中的波特率就可以认为是比特率，即每秒传输的位数(bit)。一般选波特率都会有 9600,19200,115200 等选项。其实意思就是每秒传输这么多个比特位数(bit)。

引入波特率的概念后可得到串口的传输节奏如下：

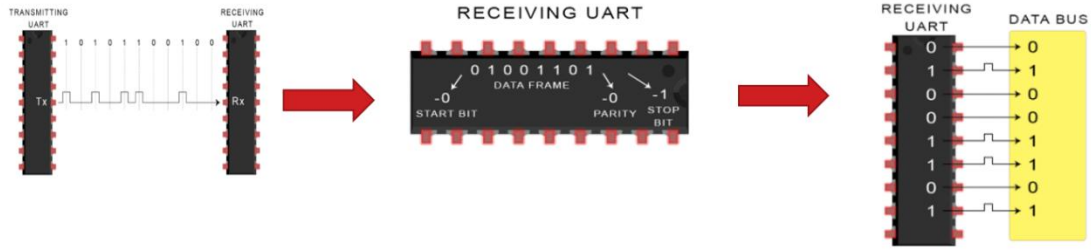


8.4串口传输步骤

8.4.1串口发送流程



8.4.2串口接收流程



8.4.3串口发送字符

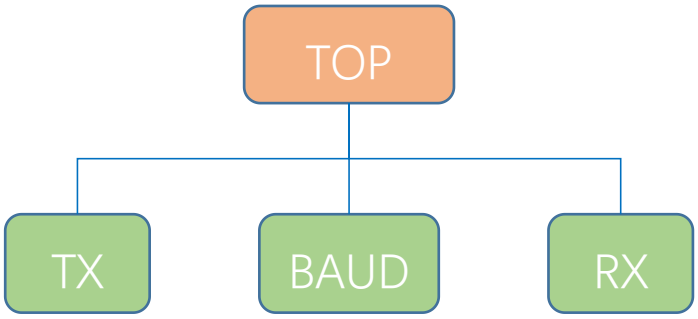
从前面串口协议中可以了解到串口每次传输可以以有 5～8bit 数据，在计算机中字符通常用 ASCII 码（7bit）表示，所以字符的发送可以用 ASCII 码发送。

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

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

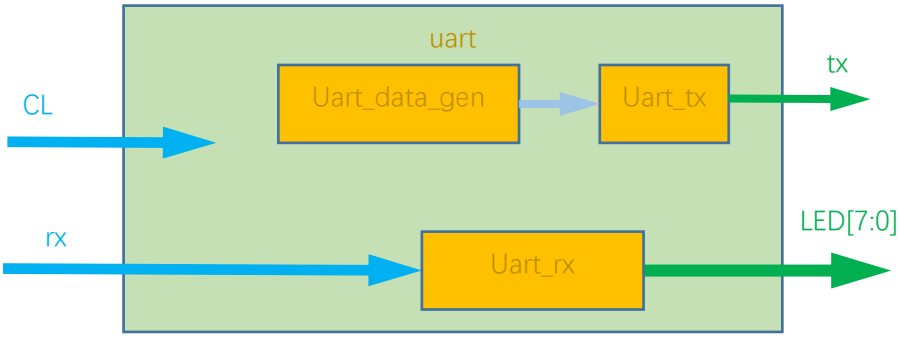
8.5 实验源码设计

从实验目的分析可将实验做如下划分：



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

上述分析仅仅搭建好 MES50HP 的与 PC 通信的桥梁 UART，传输的数据没有体现。故而需要增加发送数据模块，与接收数据模块；



8.5.1串口发射模块设计

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

代码设计如下：

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

    endmodule
```

Module 设计的关键点如下（完整 module 查看源文件）：

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

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

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


//=============================================================
        //wire and reg in the module

//=============================================================
        reg                tx_pluse_reg =0;

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

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

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

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

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


//=============================================================
==============
        //logic

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

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

        // uart  模块发送工作使能标志信号
        always @(posedge clk)
        begin
            if(~tx_pluse_reg & tx_pluse)
                tx_en <= 1'b1;
            else if(tx_state == SEND_END)
                tx_en <= 1'b0;
        end

        //division the clock to satisfy baud rate.波特周期计数器
        always @ (posedge clk)
        begin
            if(clk_div_cnt == BPS_NUM || (~tx_pluse_reg & tx_pluse))
                clk_div_cnt     <= `UD 16'h0;
```

```
            else
                clk_div_cnt     <= `UD clk_div_cnt + 16'h1;
        end

        //count the number has transmited.发送数据状态中，发送 bit 位计数，以
波特周期累加
        always @ (posedge clk)
        begin
            if(!tx_en)
                tx_bit_cnt      <= `UD 3'h0;
            else if((tx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
                tx_bit_cnt      <= `UD 3'h0;
            else if((tx_state == SEND_DATA) && (clk_div_cnt == BPS_NUM))
                tx_bit_cnt      <= `UD tx_bit_cnt + 3'h1;
            else
                tx_bit_cnt      <= `UD tx_bit_cnt;
        end


//=============================================================
==============
        //transmitter state machine

//=============================================================
==============

        //    state change  状态跳转
        always @(posedge clk)
        begin
            tx_state <= tx_state_n;
        end

        //    state change condition  状态跳转条件及规律
        always @ (*)
        begin
          case(tx_state)
            IDLE      :
            begin
                if(~tx_pluse_reg & tx_pluse)
    //触发发送做 16 个时钟周期延时后跳转到，发送 start 状态
                    tx_state_n = SEND_START;
                else
                    tx_state_n = tx_state;
            end
```

```
                SEND_START    :
                begin
                    if(clk_div_cnt == BPS_NUM)
    //发送一个波特周期的低电平后进入，发送数据状态
                        tx_state_n = SEND_DATA;
                    else
                        tx_state_n = tx_state;
                end
                SEND_DATA     :
                begin
                    if(tx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM)
    //计时 8 个波特周期后（发送了 8bit 数据），跳转到发送 stop 状态
                        tx_state_n = SEND_STOP;
                    else
                        tx_state_n = tx_state;
                end
                SEND_STOP     :
                begin
                    if(clk_div_cnt == BPS_NUM)
    //设置停止位宽为 1 个波特周期，计数发送一个波特周期的高电平，之后跳
转到发送结束状态
                        tx_state_n = SEND_END;
                    else
                        tx_state_n = tx_state;
                end
                SEND_END      :   tx_state_n = IDLE;
                default :   tx_state_n = IDLE;
            endcase
        end

        //     logical ouput    状态机输出
        always @ (posedge clk)
        begin
          if(tx_en)
          begin
              case(tx_state)
                  IDLE          :   uart_tx   <= `UD 1'h1;
    //空闲状态输出高电平
                  SEND_START :   uart_tx   <= `UD 1'h0;
    //start 状态发送一个波特周期的低电平
                  SEND_DATA   :
    //发送状态每个波特周期发送一个 bit;
                  begin
                      case(tx_bit_cnt)
```

```
                        3'h0   :   uart_tx   <= `UD tx_data[0];
                        3'h1   :   uart_tx   <= `UD tx_data[1];
                        3'h2   :   uart_tx   <= `UD tx_data[2];
                        3'h3   :   uart_tx   <= `UD tx_data[3];
                        3'h4   :   uart_tx   <= `UD tx_data[4];
                        3'h5   :   uart_tx   <= `UD tx_data[5];
                        3'h6   :   uart_tx   <= `UD tx_data[6];
                        3'h7   :   uart_tx   <= `UD tx_data[7];
                        default: uart_tx   <= `UD 1'h1;
                    endcase
                end
                SEND_STOP   :   uart_tx   <= `UD 1'h1;
//发送停止状态 输出 1 个波特周期高电平
                default     :   uart_tx   <= `UD 1'h1;
// 其他状态默认与空闲状态一致，保持高电平输出
            endcase
        end
        else
            uart_tx <= `UD 1'h1;
    end

    endmodule
```

8.5.2串口接收模块设计

串口接收模块是发射模块的逆过程，设计思路区别不大，但是有如下几点需要注意：

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

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

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

Module 设计如下：

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

module uart_rx # (
    parameter           BPS_NUM     =     16'd434
```

```
    //   设置波特率为 4800 时，     bit 位宽时钟周期个数:50MHz set 10417
40MHz set 8333
    //   设置波特率为 9600 时，     bit 位宽时钟周期个数:50MHz set 5208
40MHz set 4167
    //   设置波特率为 115200 时，  bit 位宽时钟周期个数:50MHz set 434
40MHz set 347
    )
    (
            //input ports
            input                   clk,
            input                   uart_rx,

            //output ports
            output reg [7:0]    rx_data,
            output reg              rx_en,
            output                  rx_finish
    );

        // uart rx state machine's state
        localparam    IDLE              = 4'h0;
    //空闲状态，等待开始信号到来.
        localparam    RECEIV_START = 4'h1;
    //接收 Uart 开始信号，低电平一个波特周期.
        localparam    RECEIV_DATA   = 4'h2;
    //接收 Uart 传输数据信号，此工程定义传输 8bit，每个波特周期中间位置取
值，8 个周期后跳转到 stop 状态.
        localparam    RECEIV_STOP   = 4'h3;
    //停止状态数据线是高电平，与空闲状态是一致的按照协议标准需要等待一
个停止位周期再做状态跳转.
        localparam    RECEIV_END    = 4'h4;
    //结束中转状态.

//=============================================================
==============
        //wire and reg in the module

//=============================================================
==============
        reg       [2:0]          rx_state=0;          //current state of tx state
machine.  当前状态
        reg       [2:0]          rx_state_n=0;        //next state of tx state machine.
下一个状态
        reg             [7:0]                   rx_data_reg;                //
```

```
接收数据缓冲寄存器
        reg                     uart_rx_1d;
    //save uart_rx one cycle.                保存 uart_rx 一个时钟周期
        reg                     uart_rx_2d;
    //save uart_rx one cycle.保存 uart_rx  前两个时钟周期
        wire                    start;
    //active when start a byte receive.   检测到 start 信号标志
        reg     [15:0]          clk_div_cnt;
    //count for division the clock.          波特周期计数器


//=============================================================
==============
        //logic

//=============================================================
==============

        //some control single.
        always @ (posedge clk)
        begin
            uart_rx_1d <= `UD uart_rx;
            uart_rx_2d <= `UD   uart_rx_1d;
        end

        assign start        = (!uart_rx) && (uart_rx_1d || uart_rx_2d);
        assign rx_finish = (rx_state == RECEIV_END);


        //division the clock to satisfy baud rate.波特周期计数器
        always @ (posedge clk)
        begin
            if(rx_state == IDLE || clk_div_cnt == BPS_NUM)
                clk_div_cnt     <= `UD 16'h0;
            else
                clk_div_cnt     <= `UD clk_div_cnt + 16'h1;
        end

        // receive bit data numbers
        //在接收数据状态中，接收的 bit 位计数，每一个波特周期计数加 1
        reg     [2:0]           rx_bit_cnt=0;       //the bits number has transmited.
        always @ (posedge clk)
        begin
            if(rx_state == IDLE)
```

```
                rx_bit_cnt <= `UD 3'h0;
            else if((rx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
                rx_bit_cnt <= `UD 3'h0;
            else   if((rx_state   ==   RECEIV_DATA)   &&   (clk_div_cnt   ==
BPS_NUM))
                rx_bit_cnt <= `UD rx_bit_cnt + 3'h1;
            else
                rx_bit_cnt <= `UD rx_bit_cnt;
        end

    //=================================================================
================
    //receive state machine
    //=================================================================
================
        //状态机状态跳转
        always @(posedge clk)
        begin
            rx_state <= rx_state_n;
        end

        //状态机状态跳转条件及跳转规律
        always @ (*)
        begin
          case(rx_state)
              IDLE          :
              begin
                  if(start)                                          //监测到
start 信号到来，下一状态跳转到 start 状态
                      rx_state_n = RECEIV_START;
                  else
                      rx_state_n = rx_state;
              end
              RECEIV_START      :
              begin
                  if(clk_div_cnt == BPS_NUM)                          //已完
成接收 start 标志信号
                      rx_state_n = RECEIV_DATA;
                  else
                      rx_state_n = rx_state;
              end
              RECEIV_DATA       :
              begin
```

```
                    if(rx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM) //已完
成 8bit 数据的传输
                        rx_state_n = RECEIV_STOP;
                    else
                        rx_state_n = rx_state;
                end
                RECEIV_STOP      :
                begin
                    if(clk_div_cnt == BPS_NUM)                          //已
完成接收 stop 标志信号
                        rx_state_n = RECEIV_END;
                    else
                        rx_state_n = rx_state;
                end
                RECEIV_END       :
                begin
                    if(!uart_rx_1d)                                     //数据
线重新被拉低，表示新数据传输又发送 start 标志信号，需要跳转到 start 状态
                        rx_state_n = RECEIV_START;
                    else                                                //
没有其他状况出现时，回到空闲状态，等待 start 信号的到来
                        rx_state_n = IDLE;
                end
                default      :   rx_state_n = IDLE;
            endcase
        end

        // 状态机输出
        always @ (posedge clk)
        begin
            case(rx_state)
                IDLE             ,
                RECEIV_START :                                          //在空
闲和 start 状态时将接收数据缓冲寄存器和数据使能置位；
                begin
                    rx_en <= `UD 1'b0;
                    rx_data_reg <= `UD 8'h0;
                end
                RECEIV_DATA   :
                begin
                    if(clk_div_cnt == BPS_NUM[15:1])            //在一个波特
周期的中间位置取数据线上传输的数据；
                        rx_data_reg    <= `UD {uart_rx , rx_data_reg[7:1]};   //
```

```
以循环右移的方式将 uart_rx 数据填入缓冲寄存器的最高位（Uart 传输低位在前，最后一个 bit 刚好是最高位）
                    end
                    RECEIV_STOP   :
                    begin
                        rx_en     <= `UD 1'b1;                    // 输出使能信号，表示最新的数据输出有效
                        rx_data <= `UD rx_data_reg;               // 将缓冲寄存器的值赋值给输出寄存器
                    end
                    RECEIV_END      :
                    begin
                        rx_data_reg <= `UD 8'h0;
                    end
                    default:      rx_en <= `UD 1'b0;
                endcase
            end

    endmodule
```

8.5.3串口发射控制模块设计

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

Module 如下：

```
    `timescale 1ns / 1ps
    `define UD #1
    module uart_data_gen(
        input                   clk,
        input       [7:0]       read_data,
        input                   tx_busy,
        input       [7:0]       write_max_num,
        output reg [7:0]        write_data,
        output reg              write_en
    );

        // set every second send a string,"====HELLO WORLD==="
        // 设置约每秒发送一个字符串
        reg [25:0] time_cnt=0;
        reg [ 7:0] data_num;
        always @(posedge clk)
        begin
            time_cnt <= `UD time_cnt + 26'd1;
```

```
end

// 设置串口发射工作区间
reg             work_en=0;
reg             work_en_1d=0;
always @(posedge clk)
begin
    if(time_cnt == 26'd2048)
        work_en <= `UD 1'b1;
    else if(data_num == write_max_num-1'b1)
        work_en <= `UD 1'b0;
end

always @(posedge clk)
begin
    work_en_1d <= `UD work_en;
end

// get the tx_busy‘s falling edge     获取 tx_busy 的下降沿
reg                 tx_busy_reg=0;
wire                tx_busy_f;
always @ (posedge clk) tx_busy_reg <= `UD tx_busy;

assign tx_busy_f = (!tx_busy) && (tx_busy_reg);

// 串口发射数据触发信号
reg write_pluse;
always @ (posedge clk)
begin
    if(work_en)
    begin
        if(~work_en_1d || tx_busy_f)
            write_pluse <= `UD 1'b1;
        else
            write_pluse <= `UD 1'b0;
    end
    else
        write_pluse <= `UD 1'b0;
end

always @ (posedge clk)
begin
    if(~work_en & tx_busy_f)
        data_num    <= 7'h0;
```

```
        else if(write_pluse)
            data_num     <= data_num + 8'h1;
    end

    always @(posedge clk)
    begin
        write_en <= `UD write_pluse;
    end

//  字符的对应 ASCII 码
    always @ (posedge clk)
    begin
        case(data_num)
            8'h0  ,
            8'h1  : write_data <= `UD 8'h77;// ASCII code is w
            8'h2  : write_data <= `UD 8'h77;// ASCII code is w
            8'h3  : write_data <= `UD 8'h77;// ASCII code is w
            8'h4  : write_data <= `UD 8'h2E;// ASCII code is .
            8'h5  : write_data <= `UD 8'h6D;// ASCII code is m
            8'h6  : write_data <= `UD 8'h65;// ASCII code is e
            8'h7  : write_data <= `UD 8'h79;// ASCII code is y
            8'h8  : write_data <= `UD 8'h65;// ASCII code is e
            8'h9  : write_data <= `UD 8'h73;// ASCII code is s
            8'ha  : write_data <= `UD 8'h65;// ASCII code is e
            8'hb  : write_data <= `UD 8'h6D;// ASCII code is m
            8'hc  : write_data <= `UD 8'h69;// ASCII code is i
            8'hd  : write_data <= `UD 8'h2E;// ASCII code is .
            8'he  : write_data <= `UD 8'h63;// ASCII code is c
            8'hf  : write_data <= `UD 8'h6F;// ASCII code is o
            8'h10 : write_data <= `UD 8'h6D;// ASCII code is m
            8'h11 ,
            8'h12 : write_data <= `UD 8'h0d;
            8'h13 : write_data <= `UD 8'h0a;
            default :     write_data <= `UD read_data;
        endcase
    end

endmodule
```

8.5.4串口实验顶层模块设计

目标：板子 1s 向串口助手发送一次十进制显示的“www.meyesemi.com”，

通过串口助手向板子以十六进制形式发送数字， LED 以二进制显示亮起。

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

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

具体的 module 实现如下:

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

module uart_top(
    //input ports
    input           clk,
    input           uart_rx,

    //output ports
    output  [7:0] led,
    output          uart_tx
);

    parameter       BPS_NUM = 16'd434;
    //  设置波特率为 4800 时，   bit 位宽时钟周期个数:50MHz set 10417 40MHz set 8333
    //  设置波特率为 9600 时，   bit 位宽时钟周期个数:50MHz set 5208 40MHz set 4167
    //  设置波特率为 115200 时， bit 位宽时钟周期个数:50MHz set 434 40MHz set 347 12M set 104


//================================================================
//wire and reg in the module
//================================================================

    wire            tx_busy;        //transmitter is free.
    wire            rx_finish;      //receiver is free.
    wire    [7:0]   rx_data;        //the data receive from uart_rx.

    wire    [7:0]   tx_data;
```

```
        wire                tx_en;              //enable transmit.

    //=============================================================
    //logic
    //=============================================================
        wire rx_en;

    //=============================================================
    //instance
    //=============================================================
        reg   [7:0] receive_data;
        always @(posedge clk)    receive_data <= led;
        uart_data_gen uart_data_gen(
            .clk              (   clk         ),//input                 clk,
            .read_data        (   receive_data ),//input        [7:0]   read_data,
            .tx_busy          (   tx_busy       ),//input               tx_busy,
            .write_max_num    (   8'h14         ),//input   [7:0]   write_max_num,
            .write_data       (   tx_data       ),//output reg [7:0]   write_data
            .write_en         (   tx_en         ) //output reg          write_en
        );

        //uart transmit data module.
        uart_tx #(
              .BPS_NUM        (       BPS_NUM                 )   //parameter
BPS_NUM   =     16'd434
         )
         u_uart_tx(
            .clk                    (   clk             ),// input             clk,
            .tx_data                (   tx_data         ),// input  [7:0]
tx_data,
            .tx_pluse               (   tx_en           ),// input
tx_pluse,
            .uart_tx                (   uart_tx         ),// output reg
uart_tx,
            .tx_busy                (   tx_busy         ) // output
tx_busy
        );

        //Uart receive data module.
        uart_rx #(
              .BPS_NUM              (   BPS_NUM         ) //parameter
BPS_NUM   =     16'd434
         )
```

```
    u_uart_rx (
        .clk            (  clk            ),// input                clk,
        .uart_rx        (  uart_rx        ),// input                uart_rx,
        .rx_data        (  rx_data        ),// output reg [7:0]     rx_data,
        .rx_en          (  rx_en          ),// output reg           rx_en,
        .rx_finish      (  rx_finish      ) // output               rx_finish
    );

    assign led = rx_data;

endmodule
```

8.6 实验现象

用 SSCOM 串口调试工具，波特率设置为 115200bps，数据格式为 1 位起始位、8 位数据位、无校验位、1 位结束位，用 Type-C 连接开发板与电脑后有如下现象：

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

实验现象二：

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