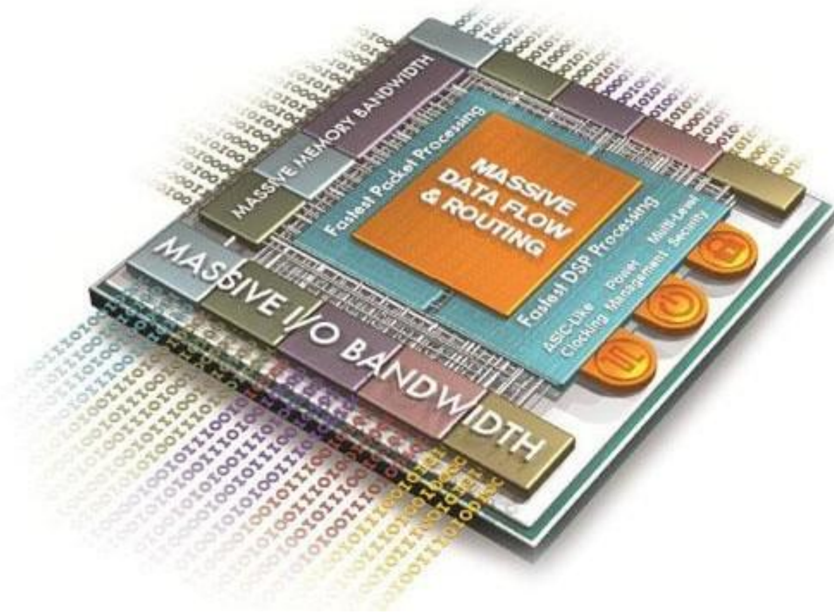
FPGA EGO1 开发平台

实验指导书



E–ELEMENTS

依元素科技

安全使用规范

- 使用扩展接口扩展电路应用前请关闭电路板总开关，避免损坏器件。
- 电路板建议在绝缘平台上使用，否则可能引起电路板损坏。
- 电路使用时应防止静电。
- 液晶显示器件或模块结雾时，不要通电工作，防止电极化学反应，产生断线。
- 电源正负极、输入/输出端口定义时需谨慎，避免应接反引起开发板的损坏。
- 保持电路板的表面清洁。
- 小心轻放，避免不必要的硬件损伤

目录

实验一：熟悉开发板和VIVADO 软件 1
一、 实验目的 1
二、 实验内容 1
三、 实验要求 1
四、 开发板介绍 1
五、 实验步骤 2
实验二：组合逻辑电路设计 15
一、 实验目的 15
二、 实验内容 15
三、 实验要求 15
四、 实验步骤 19
五、 实验结果 21
实验三：时序逻辑电路设计 21
一、 实验目的 21
二、 实验内容 21
三、 实验要求 21
四、 实验步骤 21
五、 实验结果 30
实验四：状态机 32
一、 实验目的 32
二、 实验内容 32
三、 实验要求 32
四、 实验步骤 32
五、 实验结果 37
实验五：模块化调用 38
一、 实验目的 38
二、 实验内容 38

四、 实验步骤 38
实验六：数码管显示 41
一、 实验目的 41
二、 实验内容 41
三、 实验要求 41
四、 实验背景知识 41
五、 实验方案及实现 43
六、 实验结果 45
实验七：交通灯 47
一、 实验目的 47
二、 实验内容 47
三、 实验要求 47
四、 实验方案及实现 47
五、 实验结果 52
实验八：秒表的设计 54
一、 实验目的 54
二、 实验内容 54
三、 实验要求 54
四、 实验方案及实现 54
五、 实验结果 57
实验九: 蜂鸣器演奏实验 59
一、 实验目的 59
二、 实验内容 59
三、 实验要求 59
四、 实验背景知识 59
五、 实验结果 64
实验十：字符型 LCM 驱动 65
一、 实验目的 65
二、 实验内容 65

四、 实验背景知识 65
五、 实验程序实现 69
六、 实验结果 75
实验十一：VGA 76
一、 实验目的 76
二、 实验内容 76
三、 实验要求 76
四、 实验背景知识 76
五、 实验结果 80
实验十二：PS/2 接口控制 81
一、 实验目的 81
二、 实验内容 81
三、 实验要求 81
四、 实验背景知识 81
五、 实验方案及实现： 84
六、 实验结果 90
实验十三：IP 核调用 91
一、 实验目的 91
二、 实验内容 91
三、 实验要求 91
四、 实验步骤 91

实验一：熟悉开发板和 VIVADO 软件

一、 实验目的

1. 熟悉 EGO1 开发板；
2. 熟悉 VIVADO 的编译环境；
3. 了解在 VIVADO 环境下运用 Verilog HDL 语言的编程开发流程，包括源程序的编写、编译、模拟仿真及程序下载。

二、 实验内容

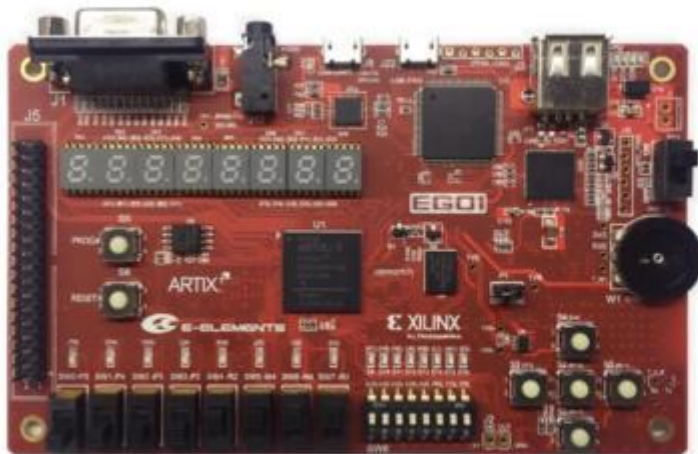
1. VIVADO 环境下源程序的编写、编译
2. 模拟仿真
3. 程序下载

三、 实验要求

1. 在 VIVADO 环境下完成对简单电路工作情况的仿真模拟；
2. 完成配置程序的下载，并在 EGO1 开发板上对程序进行最终验证。

四、开发板介绍

EGO1 FPGA 口袋实验平台



功能特性：

- 采用 Xilinx Artix-7 XC7A35T芯片
- 配置方式：USB-JTAG/SPI Flash
- 高达100MHz 的内部时钟速度
- 存储器：2Mbit SRAM
  N25Q032A SPI Flash
- 通用IO：Switch ：x8、LED：x16、Button：x5、DIP：x8
- 通用扩展IO：32pin
- 音视频/显示：
  7段数码管：x8
  VGA视频输出接口
  Audio音频接口
- 通信接口：UART：USB转UART、Bluetooth：蓝牙模块
- 模拟接口： DAC：8-bit分辨率、XADC：2路12bit 1Msps ADC

五、 实验步骤

1．介绍在 VIVADO 环境下的编程开发流程

(1) 启动 VIVADO。如图 1.1 所示：

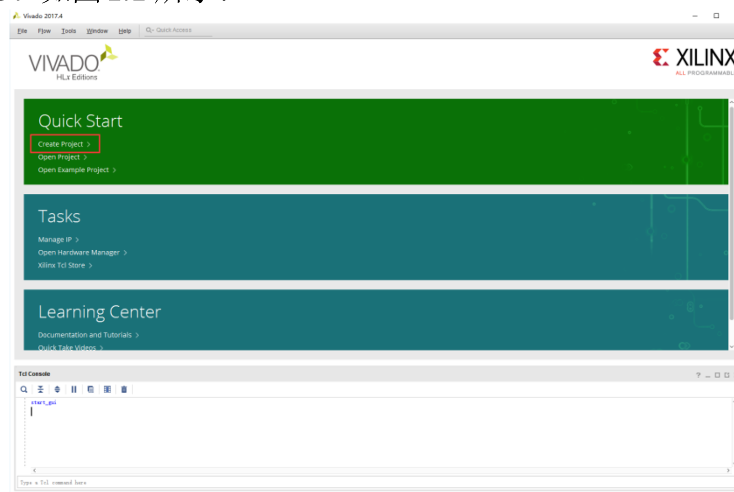


图 1.1、VIVADO 窗口界面

(2) 利用向导，建立一个新项目。

- 在New Project菜单中点击Next。
- 填写所要新建的工程名。如这里的工程名：led，工程所在位置：E:\SC_Vivado\list_Vivado，然后点击Next。如图1.2所示：

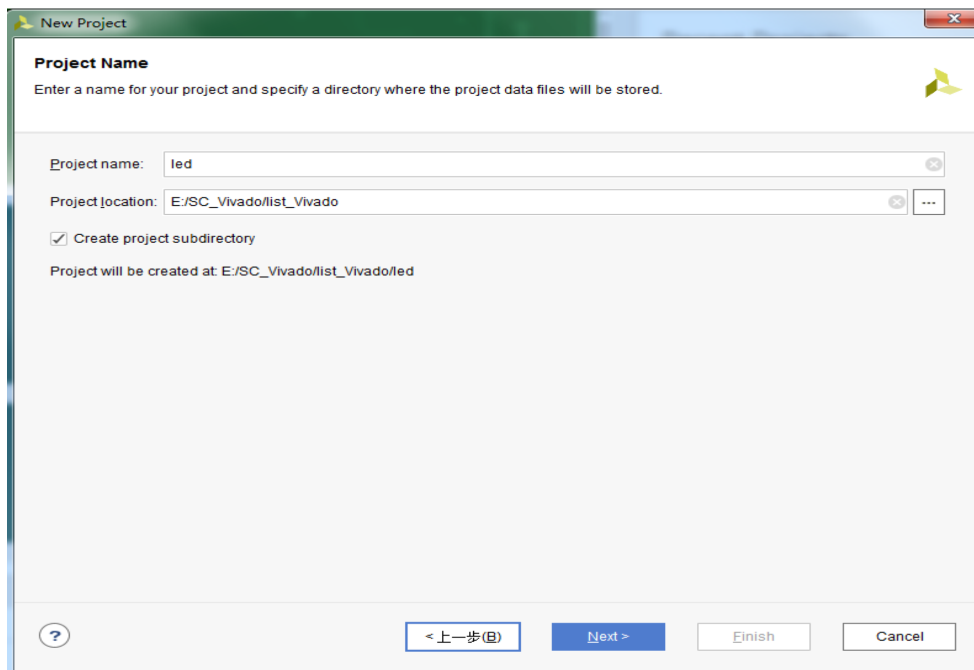


图 1.2、 VIVADO 项目名称、路径设定窗口

- 选择创建RTL Project，勾选Do not specify source at this time,跳过添加文件步骤，选择完成后点击Next进入下一步。如图 1.3 所示：

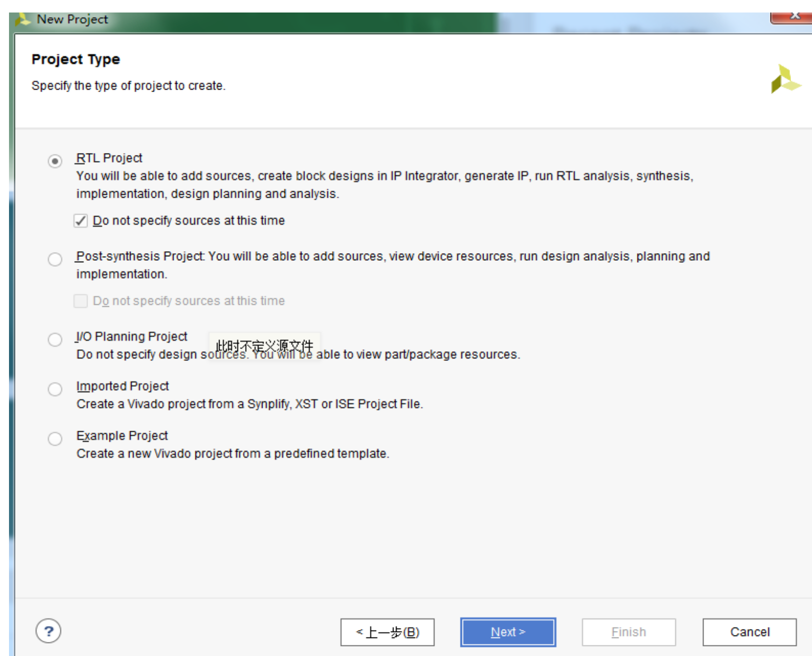


图 1.3、项目类型

- 器件的选择是和实验平台的硬件相关的，根据我们的 EGO1 实验开发板，它使用的是 xc7a35tcsg324-1 的器件，找到相应的器件，如图 1.4 所示：

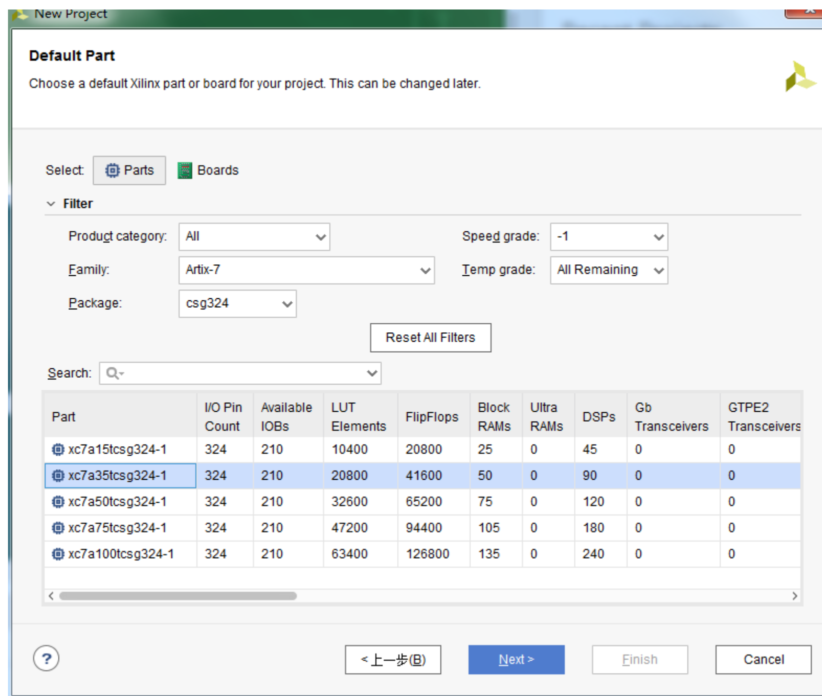


图 1.4、 VIVADO 中器件选择窗口

- 在New Project Summary界面检查新建的设计内容是否符合我们需求，确认无误后，点击Finish完成。如图 1.5 所示：

图 1.5、新建设计内容完成窗口

- 点击Finish后自动进入创建完毕后的空白工程。如图1.6所示：

图 1.6、空白工程窗口

- VIVADO 中包含完整的文本编辑程序（Text Editor），在此用 Verilog HDL 来编写源程序。新建一个 Verilog HDL 文件，可以通过右击 Design sourse 选择 Add Sourse，如图 1.7 所示：

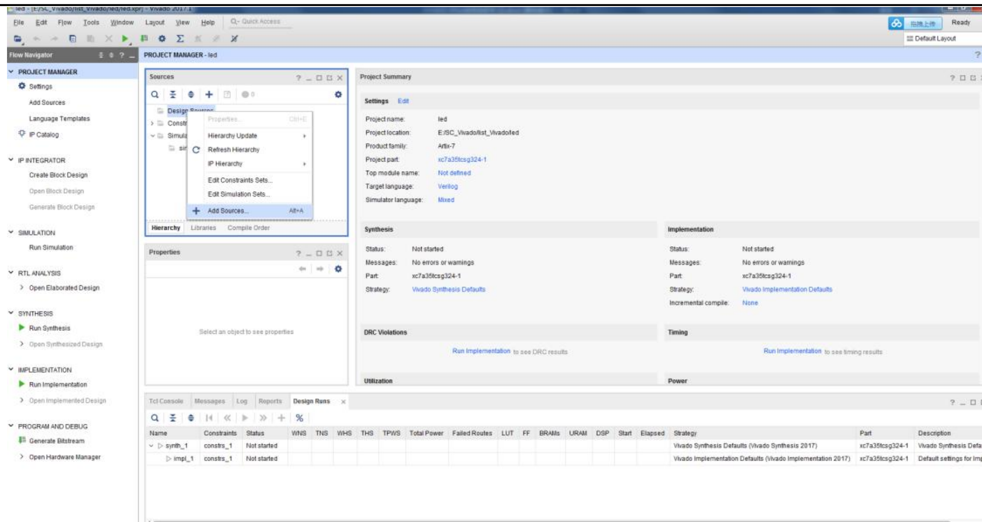


图 1.7、添加源文件

- 选择第二个选项，如图 1.8 所示：

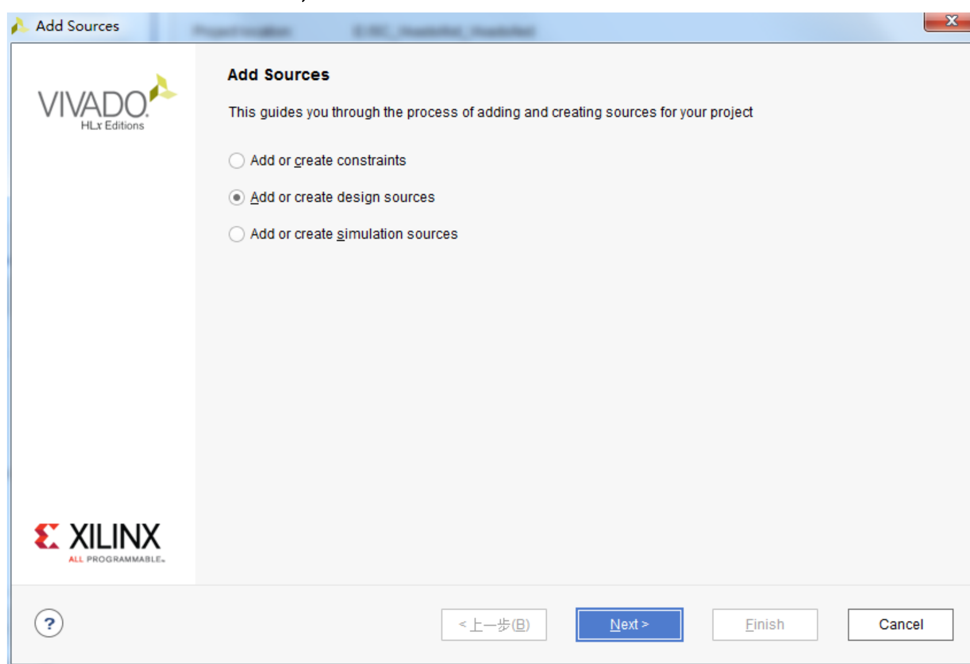


图 1.8、选择文件类型

- 选择Create Files输入led，点击OK，确认led.v添加进去后，选择Finish完成设计文件添加。如图1.9所示：

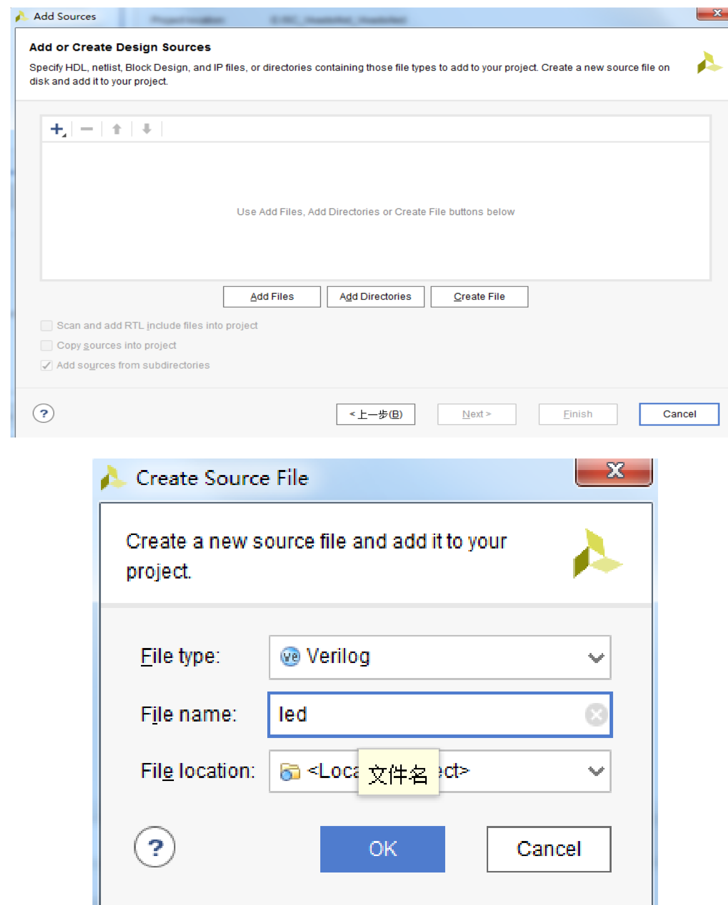


图 1.9、创建文件

- 创建完成点击 Finish，如图 1.10 所示：

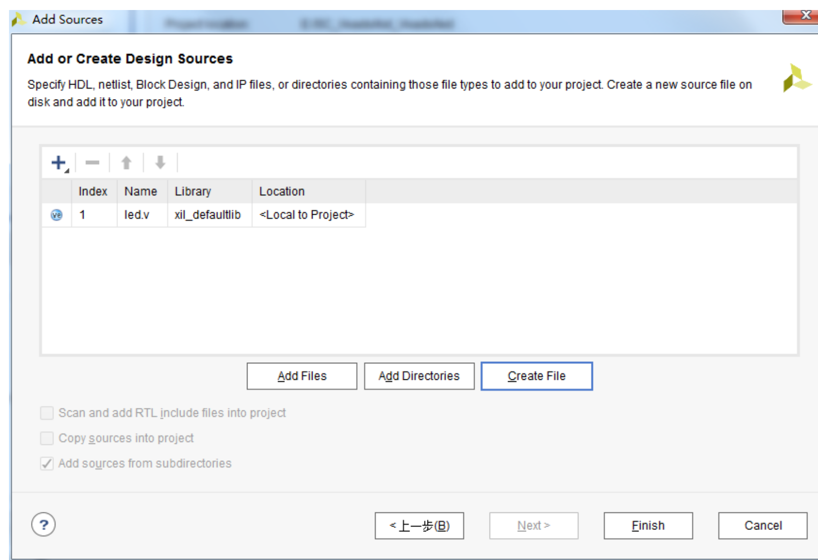


图 1.10、创建完成

- 填写模块名称和端口，如图 1.11 所示：

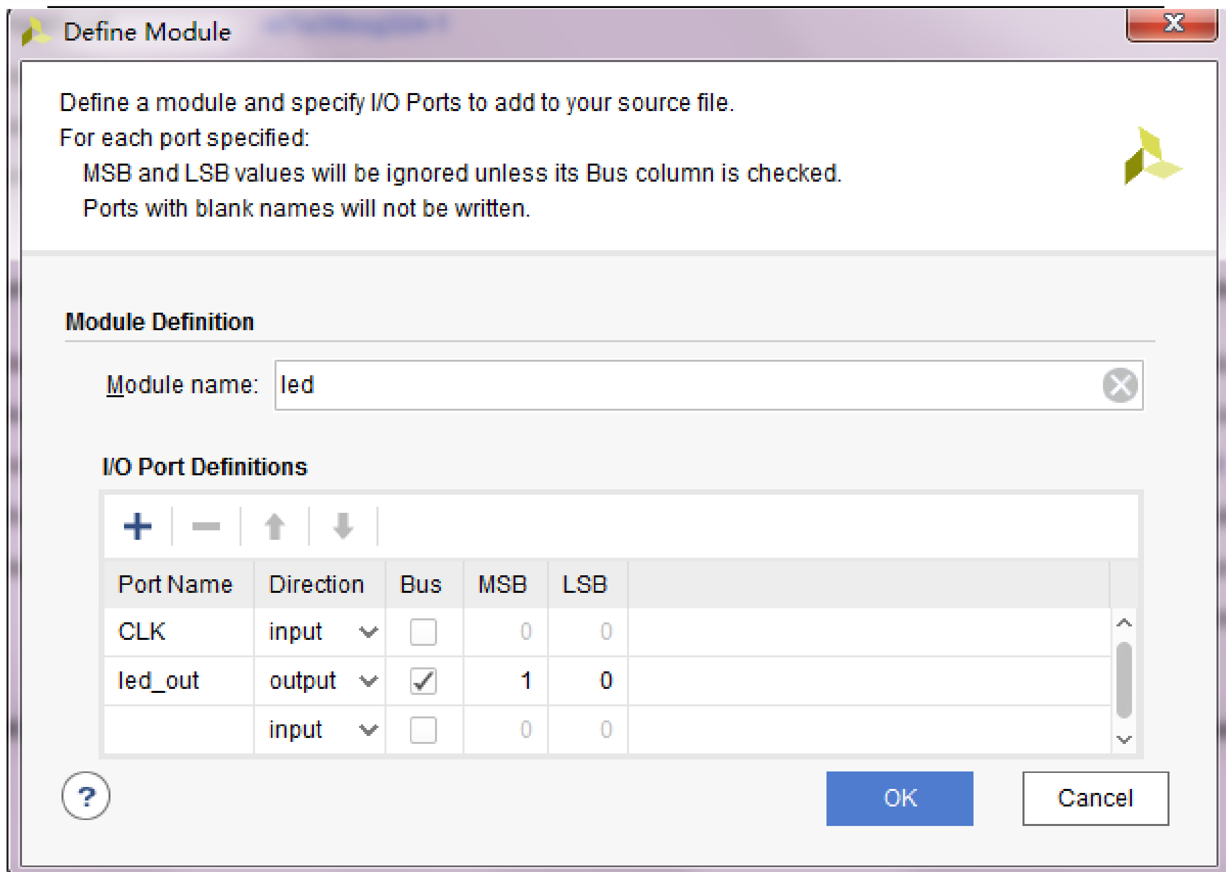


图 1.11、 Define Module 窗口

(1) Verilog HDL 程序输入。

在用户区 Verilog HDL 文件窗口中输入源程序，保存时文件名与实体名保持一致。如图1.12所示：

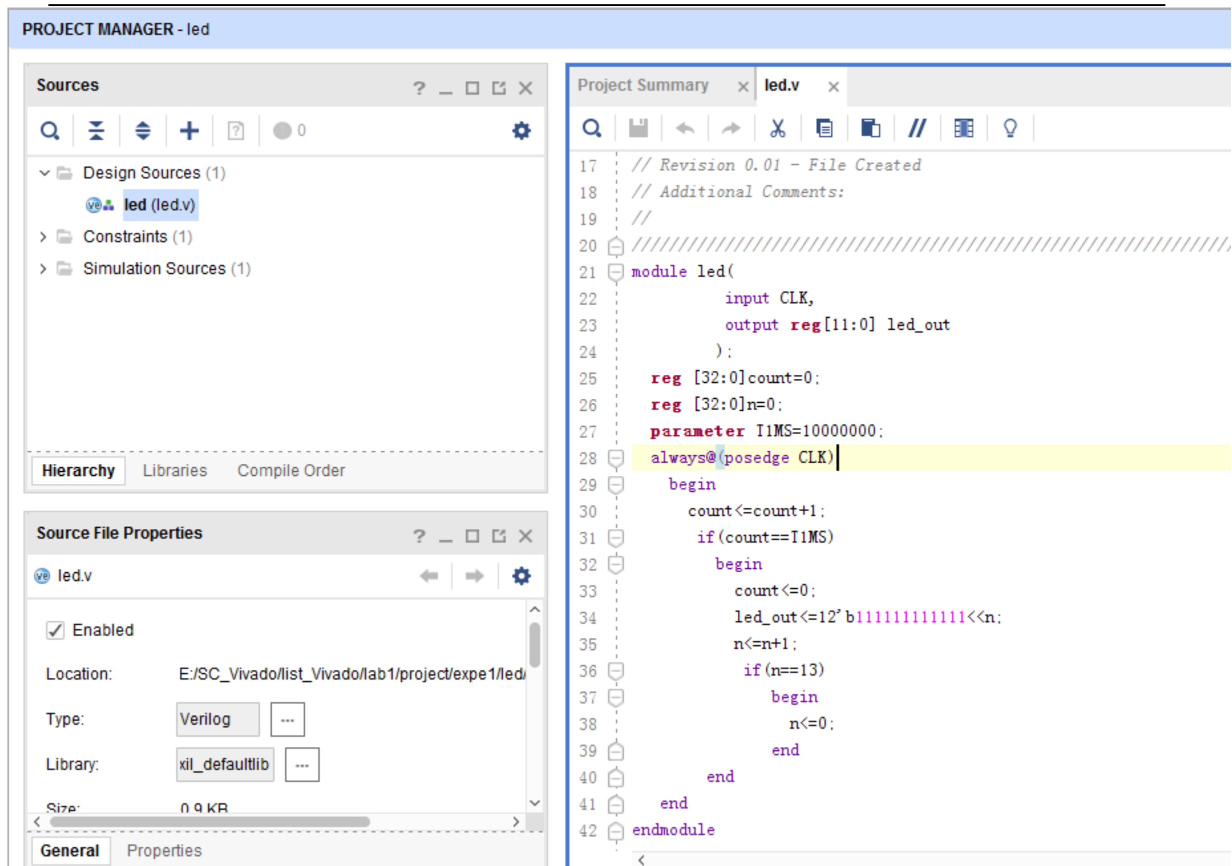


图 1.12、 Verilog 代码编辑窗口

编辑代码如下：

```
module led(
        input CLK,
        output reg[11:0] led_out
       );
 reg [32:0]count=0;
 reg [32:0]n=0;
 parameter T1MS=10000000;
 always@(posedge CLK)
   begin
     count<=count+1;
      if(count==T1MS)
        begin
          count<=0;
          led_out<=12'b111111111111<<n;
          n<=n+1;
           if(n==13)
              begin
                n<=0;
              end
        end
   end
endmodule
```

(4) VIVADO 程序编译。

- 完成 Synthesize 的综合编译。如图 1.13 所示：

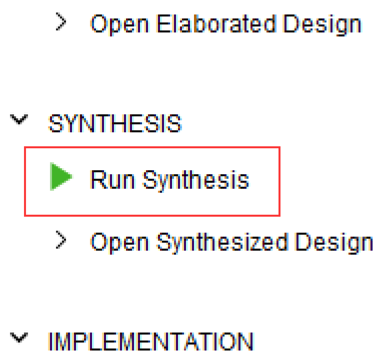


图 1.13、 Verilog 代码 Synthesize 综合编译

- 编译成功后双击 Schematic 可以查看 RTL 级电路图。如图 1.14 所示：

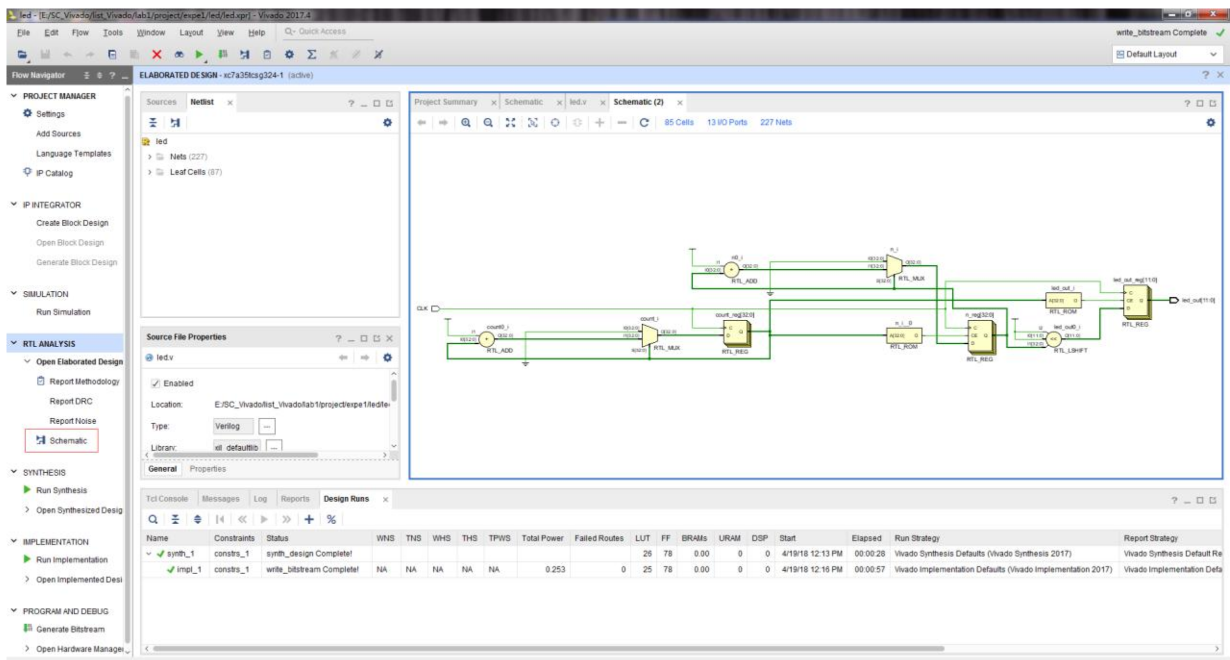


图 1.14、 RTL 级电路图

(5) 分配引脚。

- 右击约束子目录下文件夹，选择 Add Sourse...，如图 1.15 所示：

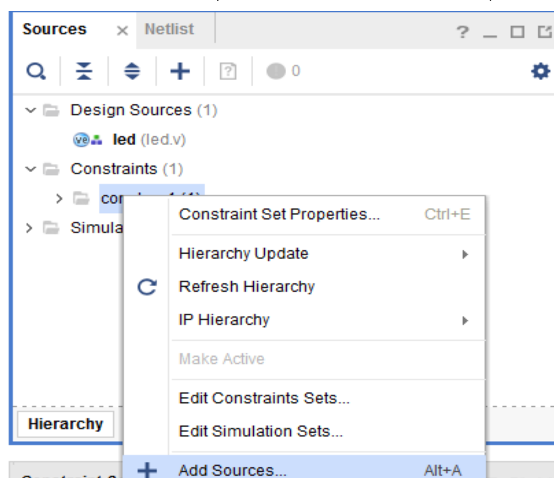


图 1.15、 添加引脚分配文件

- 选择第一项 Add or create constraints,点击 Next
- 选择 Create File...弹出下面的窗口，填写约束文件的文件名 led。如图 1.16 所示：

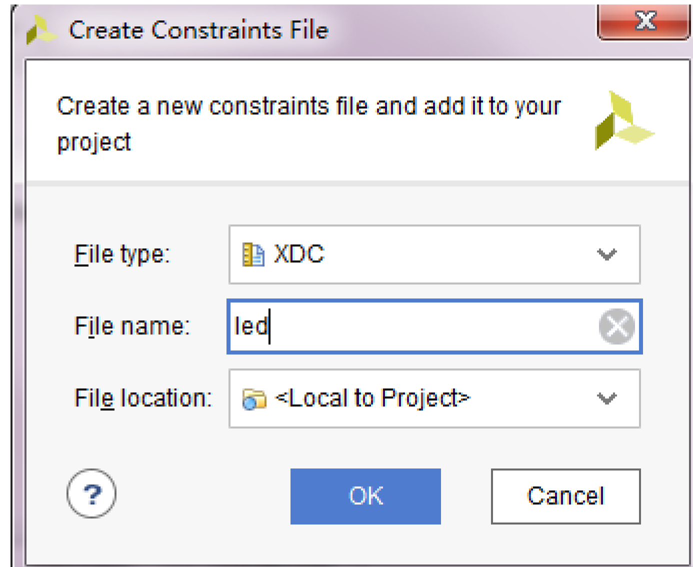


图 1.16、创建约束文件

约束文件如下：

```
set_property PACKAGE_PIN P17 [get_ports CLK]
set_property PACKAGE_PIN K3 [get_ports {led_out[0]}]
set_property PACKAGE_PIN M1 [get_ports {led_out[1]}]
set_property PACKAGE_PIN L1 [get_ports {led_out[2]}]
set_property PACKAGE_PIN K6 [get_ports {led_out[3]}]
set_property PACKAGE_PIN J5 [get_ports {led_out[4]}]
set_property PACKAGE_PIN H5 [get_ports {led_out[5]}]
set_property PACKAGE_PIN H6 [get_ports {led_out[6]}]
set_property PACKAGE_PIN K1 [get_ports {led_out[7]}]
set_property PACKAGE_PIN K2 [get_ports {led_out[8]}]
set_property PACKAGE_PIN J2 [get_ports {led_out[9]}]
set_property PACKAGE_PIN J3 [get_ports {led_out[10]}]
set_property PACKAGE_PIN H4 [get_ports {led_out[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[11]}]
```

(6) 完成实现之后，与综合相同可以选择Open Implemented Design打开实现后的设计，或者直接点击Generate Bitstream。如图1.17所示：

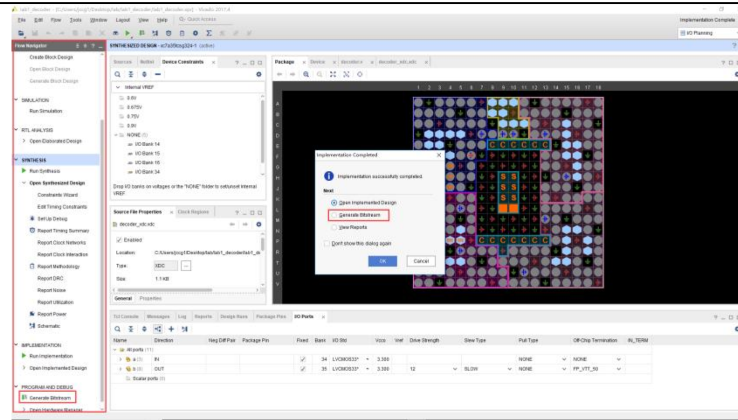


图 1.17、生成二进制文件

- Bitstream文件生成完毕后，选择Open Hardware Manager。如图1.18所示：

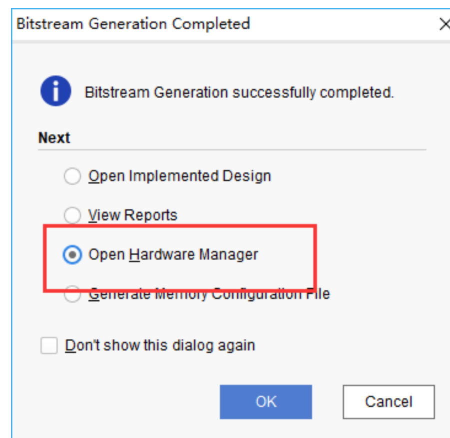


图 1.18、Hardware Manager

- 将EGO1与电脑连接后，点击Open Target->Auto Connect 连接板卡。如图1.19所示：

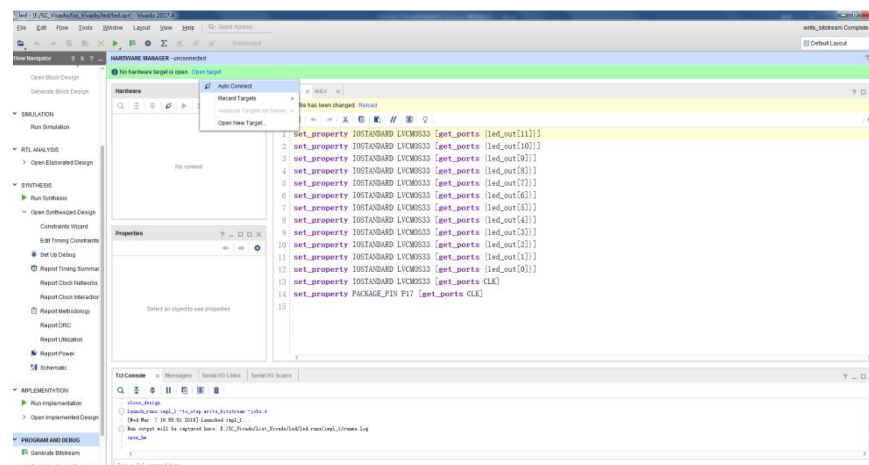


图 1.18、连接设备

- 连接成功后，右击 FPGA 芯片选择 Program Device。如图 1.19 所示：

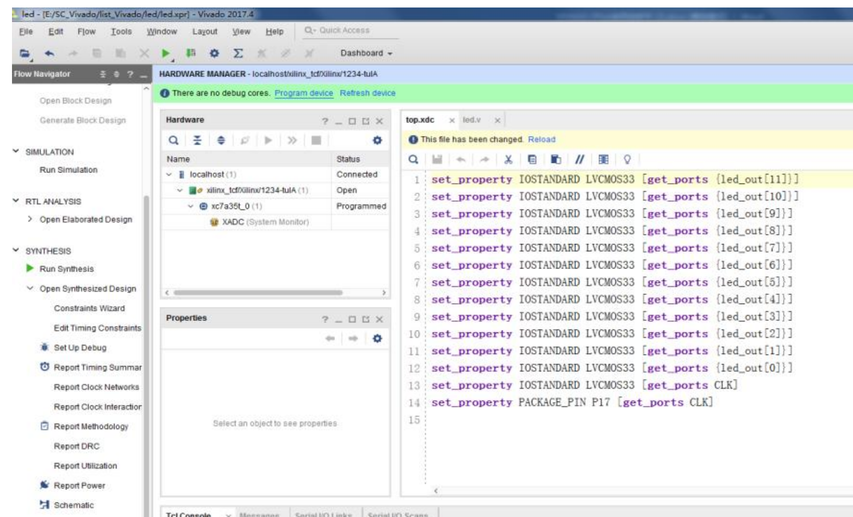


图 1.19、对开发板编程

下载完成 OK，开发板即可演示。

接下来介绍如何将程序烧录到 ROM 里，这样程序就能掉电不丢失。先生成 bin 文件，点击 Settings，选择 Bitstream，勾选 bin_file，点击 Apply，点击 OK。如图 1.20 所示：

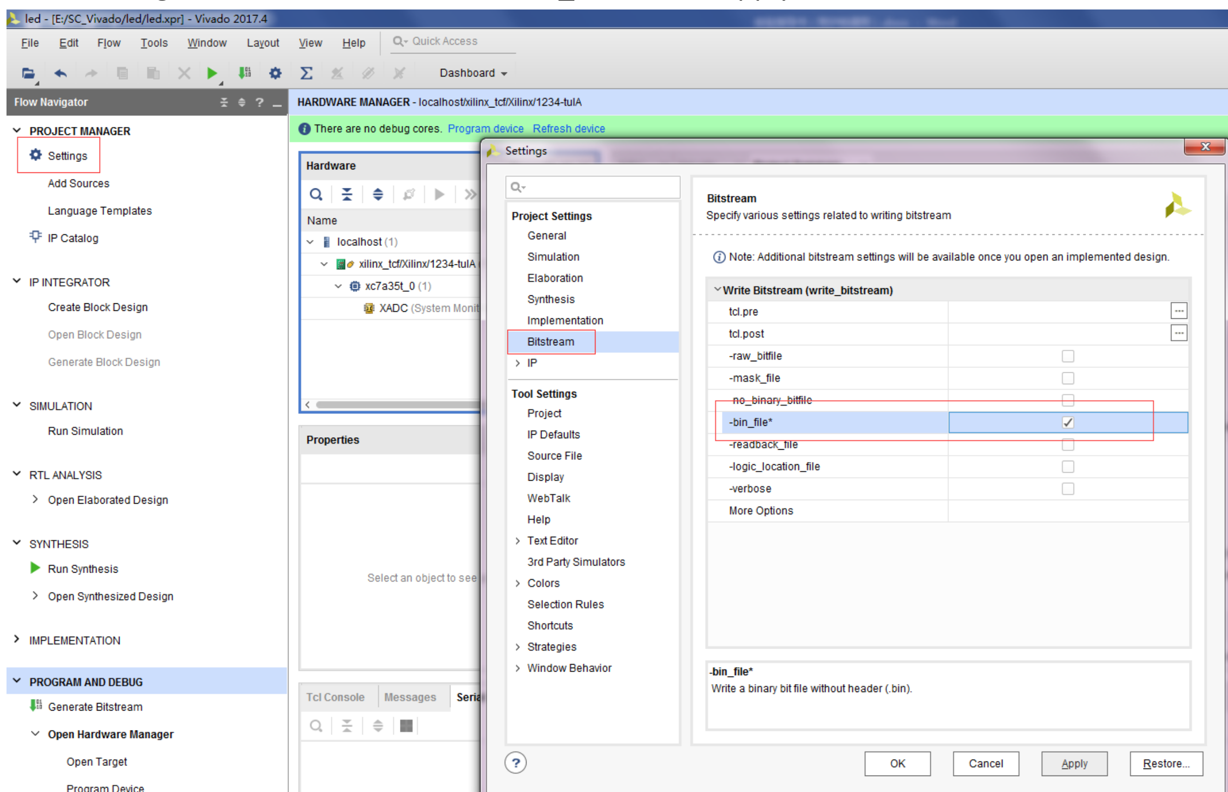


图 1.20、 生成 bin 文件

- 点击 Generate Bitstream。
- 右击 FPGA 芯片选择 Add Configuration Memory Device。如图 1.21 所示：

led - [E:/SC_Vivado/list_Vivado/lab1/project/expe1/led/led.xpr] - Vivado 2017.4

File Edit Flow Tools Window Layout View Help Quick Access

Dashboard

Flow Navigator

PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog

IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design

SIMULATION
Run Simulation

RTL ANALYSIS
Open Elaborated Design
Report Methodology

HARDWARE MANAGER - localhost/xilinx_tcf/Xilinx/1234-tulA

There are no debug cores. Program device Refresh device

Hardware

Name	Status
localhost (1)	Connected
xilinx_tcf/Xilinx/1234-tulA (1)	Open
xc7a35	
XAD	

Hardware Device P
xc7a35t_0
Name:
Part:

led.v
17 // Revi
18 // Addi
19 //
20 //////

Hardware Device Properties... Ctrl+E
Program Device...
Verify Device...
Refresh Device
Add Configuration Memory Device...
Boot from Configuration Memory Device
Program BBR Key...
Clear BBR Key...
Program eFUSE Registers...
Export to Spreadsheet...

图 1.21、 ROM 烧写

- 选择 flash 芯片型号，如图 1.22 所示：

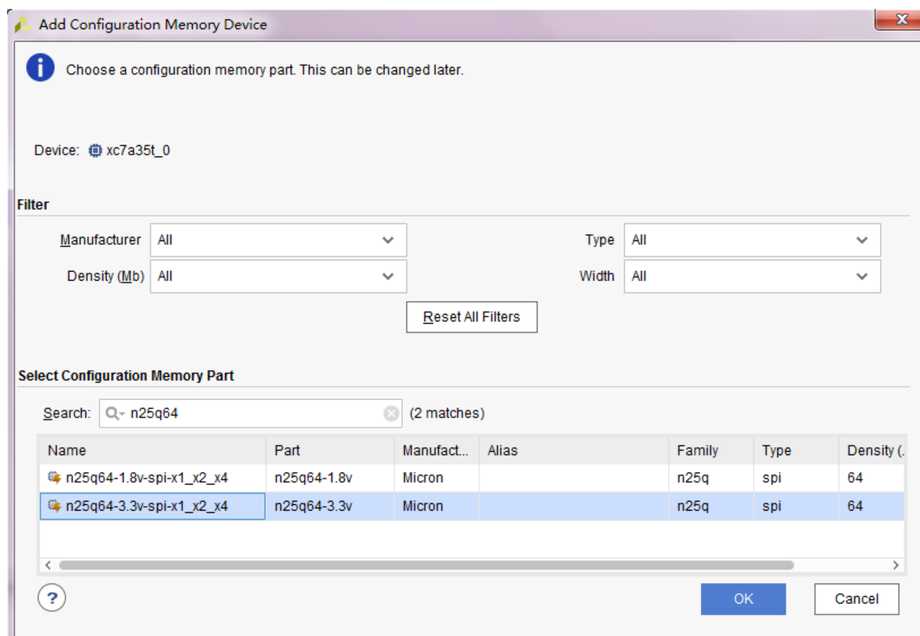


图 1.22、 选择芯片型号

- 在弹出的窗口选择配置文件为前面生成的.bin 文件

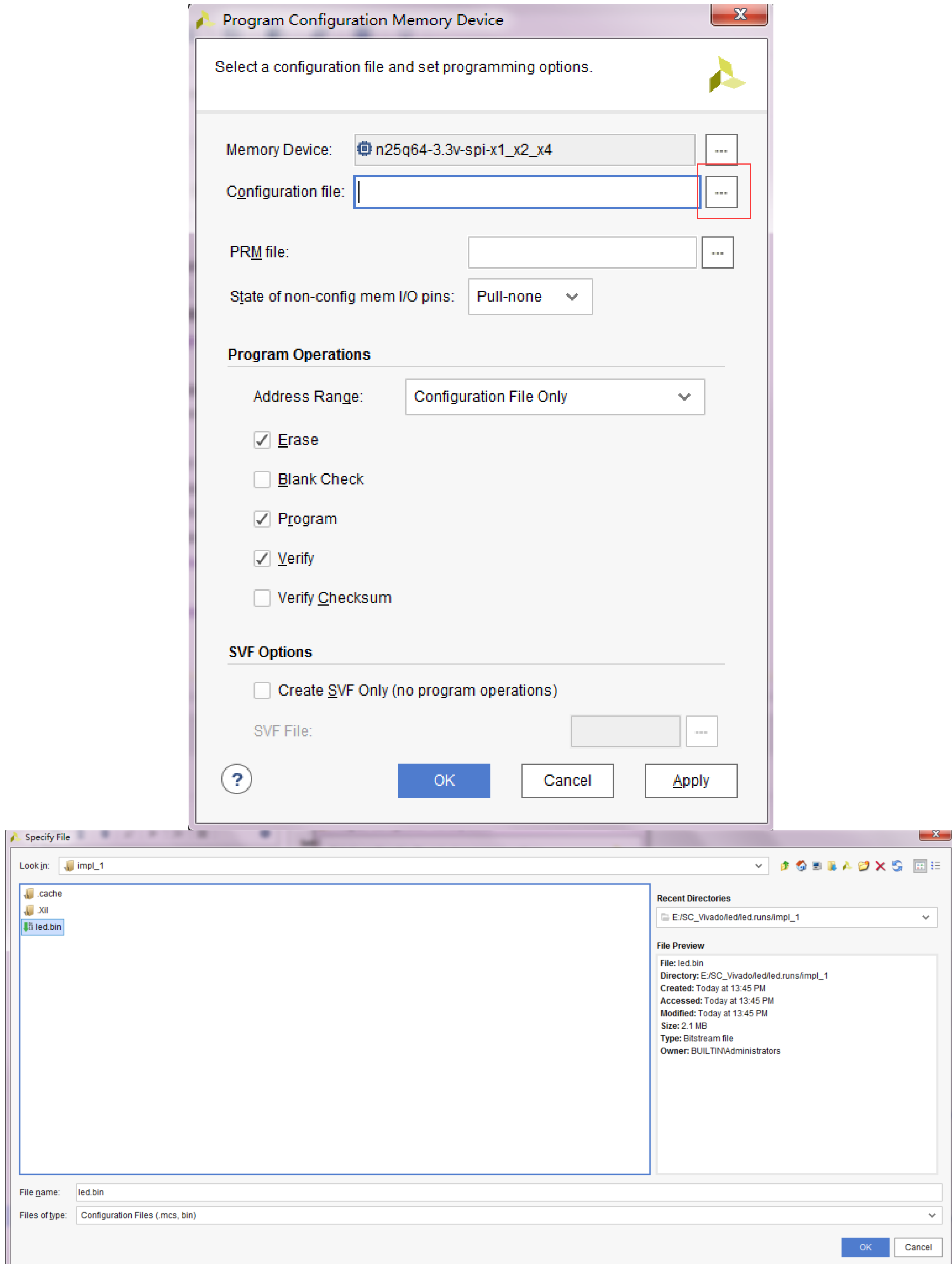


图 1.23、选择 bin 文件

选定文件并完成编程后点击 OK，完成 Flash programming。

实验二：组合逻辑电路设计

一、 实验目的

1. 学习 Verilog HDL 基本语法；
2. 巩固 VIVADO 环境下的 Verilog HDL 编程设计的基础。

二、 实验内容

1. 实现以下组合逻辑功能：编码 / 译码器，比较器，全加器。

三、 实验要求

1. 在 PC 机上完成相应的时序仿真，对结果进行分析；
2. 完成下载，在实验板上对程序进行验证。

四、 实验步骤

1. 编码器的实现

编码器通常分为两大类： 普通编码器和优先编码器。 其中普通编码器就是对某一个给定时刻只能对一个输入信号进行编码的编码器， 它的输入端口不允许同一时刻出现两个以上的有效输入信号； 优先编码器就是对某一个给定时刻只对优先级最高的输入信号进行编码的编码器，它的输入端口允许多个输入信号同时有效。

现以编码器为例，介绍普通编码器的 Verilog HDL 语言程序设计。通常，四至二线编码器的逻辑电路符号如图 2.1 所示，真值表如表 2.1 所示。不难看出该编码器的工作原理为：编码器将对四个输入信号进行编码操作，然后以两位二进制码的形式输出，这里输入信号为低电平有效。

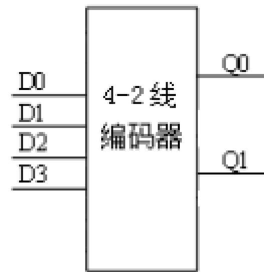


图 2.1、 四至二线编码器的电路符号

表 2.1、四至二线编码器的真值表

D3	D2	D1	D0	Q1	Q0
0	1	1	1	1	1
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	0	0	0

具体操作过程如下：

（1） 利用项目向导，建立一个新项目，建议工程名为 expe2。

（2） 新建一个 Verilog HDL 文件，并输入源程序：

```
module   encoder4_2(q,d);
input[3:0]d;
output[1:0]   q;
reg[1:0] q;
    always@(d) begin
    case(d)
    4'b0111:   q<=2'b11;
    4'b1011:   q<=2'b10;
    4'b1101:   q<=2'b01;
    4'b1110:   q<=2'b00;
    default:   q<=2'bzz;
endcase
end
endmodul
e
```

a． 对源程序进行语法检查并编译。

b． 对项目进行时序逻辑功能仿真。

c． 分配管脚。（管脚分配可参照实验结果分）

d． 下载。

2． 比较器的实现

数字比较器的设计，通常依据两组二进制数码的数值大小来进行比较，即

a>b、a=b 或 a<b，这三种情况有一种值为真。比较器的电路符号如图 2.2。

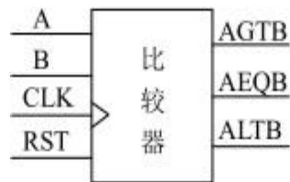


图 2.2、比较器电路符号

各引脚说明： A、B：皆为二位信号； CLK：时钟脉冲输入； RST：清除控制。

AGTB ：当 A>B 时，其值为 1，否则为 0；

AEQB ：当 A=B 时，其值为 1，否则为 0；

ALTB：当 A<B 时，其值为 1，否则为 0。

其操作过程同译码器的实现，这里不再赘述。注意顶层文件名一定要设为 comp。

源程序如下：

```
module comp(CLK,RST,A,B,AGTB,ALTB,AEQB);
input CLK,RST;
input[1:0] A,B;
output    AGTB,ALTB,AEQB; reg
          AGTB,ALTB,AEQB;
always    @(posedge CLK or negedge RST)
  begin
  if(!RST)
    begin
    AGTB<=0;
    AEQB<=0;
    ALTB<=0;
    en
  d else
    begin
    if(A>B)
      begin
      AGTB<=1;
      AEQB<=0;
      ALTB<=0;
      end
    else if(A==B)
      begin
      AGTB<=0;
      AEQB<=1;
      ALTB<=0;
```

```
        en
      d else
        begin
        AGTB<=0;
        AEQB<=0;
        ALTB<=1;
        en
      d end
    end
endmodul
e
```

3．全加器的实现

全加器其实就是考虑到进位的加法器。一位全加器的电路符号如图 2.3 所示，真值表如表 2.2 所示。

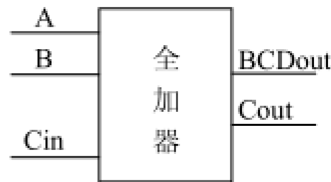


图 2.3、全加器电路符号 表 2.2、 一位全加器真值表

全加器输入			全加器输出	
A	B	Cin	BCDout	Cout
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

具体操作步骤不再一一给出。这里仅给出一位全加器的源程序。

源程序如下：

```
module ful_adder(cout,sum,a,b,cin);
input   a,b;
input   cin;
output   sum;
output   cout;
reg   sum;
reg   cout;
always   @(a or b or cin)
```

begin

```
    {cout,sum}=a+b+cin;
    end
endmodule
```

五、 实验结果

1． 编码器

管脚分配如下表：

程序中管脚名	实际管脚	说明
D0	N4	拨动开关 SW1
D1	M4	拨动开关 SW2
D2	R2	拨动开关 SW3
D3	P2	拨动开关 SW4
Q0	K2	LED 0
Q1	J2	LED 1

实验结果如下表：

拨动开关 1 脚	拨动开关 1 脚	拨动开关 1 脚	拨动开关 1 脚	拨动开关 1 脚	拨动开关 1 脚
0	1	1	1	亮	亮
1	0	1	1	暗	亮
1	1	0	1	亮	暗
1	1	1	0	暗	暗

2． 比较器：

管脚分配如下表：

程序中管脚名	实际管脚	说明
CLK	P17	全局时钟脚
A（1）	N4	拨动开关 SW1
A（0）	M4	拨动开关 SW2
B（1）	R2	拨动开关 SW3
B（0）	P2	拨动开关 SW4
RST	P15	按键 BTNC
AGTB	K2	LED 0
AEQB	J2	LED 1
ALTB	J3	LED 2

实验结果如下表：

按键 K1	拨动开关 1~4 脚	LED D1	LED D2	LED D3
按下	X	暗	暗	暗
未按下	开关[1-2] > 开关[3-4]	亮	暗	暗
	开关[1-2] = 开关[3-4]	暗	亮	暗
	开关[1-2] < 开关[3-4]	暗	暗	亮

3. 全加器

管脚分配如下表：

程序中管脚名	实际管脚	说明
A	N4	拨动开关 SW1
B	M4	拨动开关 SW2
CIN	R2	拨动开关 SW3
SUM	K2	LED 0
COUT	J2	LED 1

实验结果如下：

这里完成的是二进制加法：sum=A+B+Cin。另外，Cout 为进位输出位，请按照上述表达式，自己检验实验结果是否正确。

相关说明：

(1) 本实验电路板中的 LED 灯共阴极连接应用，当输入高电平‘1’时，LED 亮；

(2) 拨动开关靠近数字标称端输出为高‘1’。

实验三：时序逻辑电路设计

一、 实验目的

1. 理解触发器和计数器的概念，掌握这些时序器件的 Verilog HDL 语言程序设计的方法。

二、 实验内容

1. 触发器（D 型）；
2. 计数器（递增、递减）。

三、 实验要求

1. 在 VIVADO 环境下进行时序仿真；
2. 完成下载，在实验板上对程序进行验证，必要时可用示波器对波形进行观察。

四、 实验步骤

1. D 触发器的实现

在各种复杂的数字电路中， 不但需要对输入信号进行算术运算和逻辑运算，还经常需要将这些信号和运算结果保存起来。因此，需要使用具有记忆功能的基本逻辑单元，能够存储一位信号的基本单元电路就被称为触发器。根据电路结构形式和 控制方式的不同，可以将触发器分为 D 触发器、JK 触发器、T 触发器等等。这里只介绍常用的 D 型触发器，其他类型触发器请有兴趣的同学自己实现。

在数字电路中，D 触发器是最为简单也是最为常用的一种基本时序逻辑电路，它是构成数字电路系统的基础。大体可分为如下几类：基本的 D 触发器；同步复位的 D 触发器；异步复位的 D 触发器；同步置位/复位的 D 触发器； 异步置位/复位的 D 触发器

下面先分别介绍各个 D 触发器的具体工作原理，然后再介绍具体操作步骤。

（1） 基本的 D 触发器

在数字电路中，一个基本的上升沿 D 触发器的逻辑电路符号如图 3.1 所示，其功能表如表 3.1 所示。

根据下面的电路符号和功能表不难看出，一个基本的 D 触发器的工作原理为：当时钟信号的上升沿到来时，输入端口 D 的数据将传递给输出端口 Q 和输出端口 Q。在此，输出端口 Q 和输出端口 Q 除了反相之外，其他特性都是相同的。

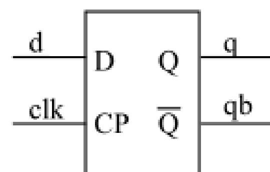


图 3.1 、电路符号

表 3.1、D 触发器的功能表

D	CP	Q	$\overline{Q}$
X	0	保持	保持
X	1	保持	保持
0	上升沿	0	1
1	上升沿	1	0

下面给出具体操作过程：

a． 利用向导，建立一个新项目，工程名为 expe3。

b． 新建一个 Verilog HDL 文件，并输入源程序：

```
module   async_rddf(clk, d,q,qb);
input      clk, d;
output      q,qb;
reg          q,qb;
always    @(posedge clk) begin
        q<=d;
        qb<=~d;
end
endmodul
e
```

c． 对源程序进行语法检查和编译；

d． 进行时序仿真；

在Simulation Source 上右击，在弹出的菜单中点击 Add Source，在弹出的对话框中选择 Add or create simulaton sources，如图3.2 所示：

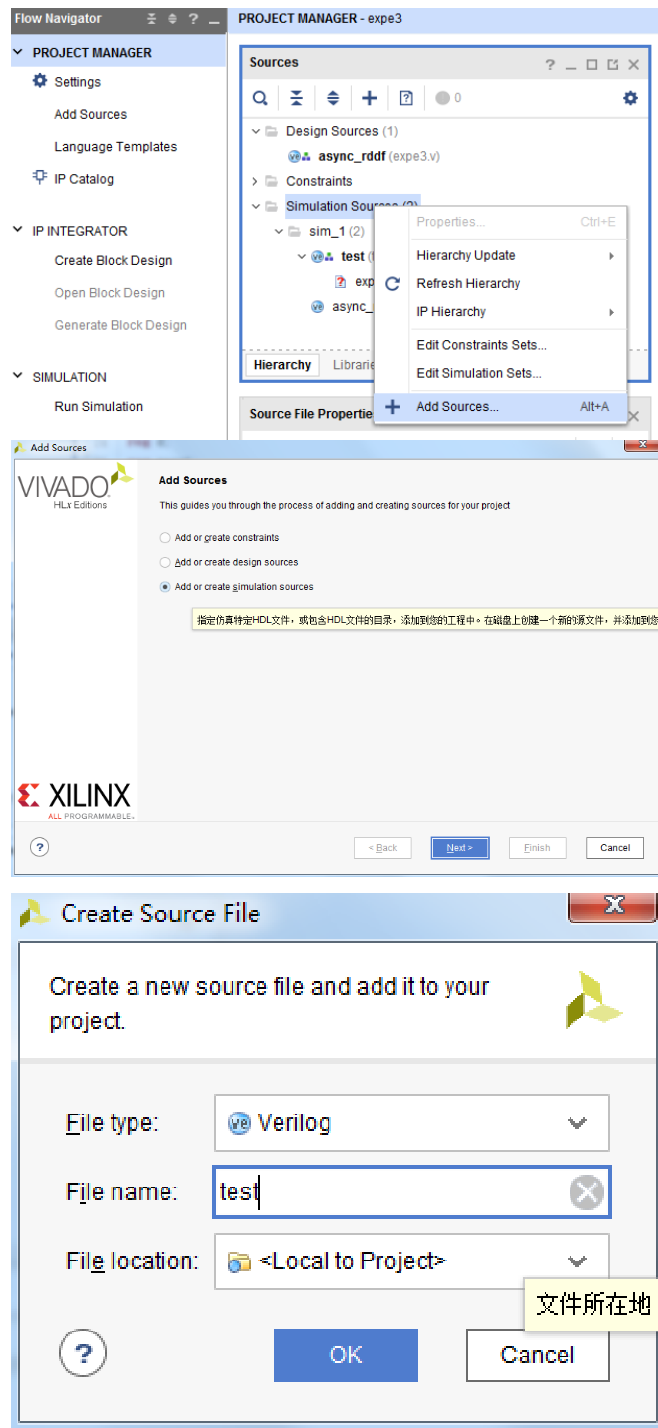
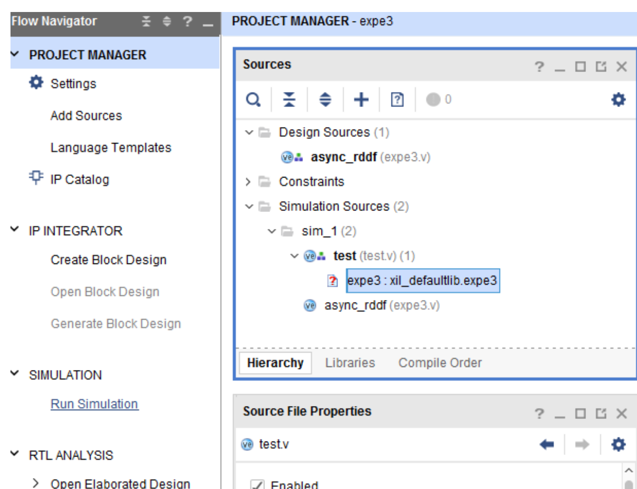


图3.2 创建仿真源文件

创建完成后，输入仿真程序如下：

```
module test;
    //
    Inputs
    reg  clk;
    reg d;
    //
    Outputs
    wire q;
    wire qb;
    // Instantiate the Unit Under Test (UUT)
    async_rddf uut (
        .clk(clk),
        .d(d),
        .q(q),
        .qb(qb)
    );
    initial begin
        // Initialize Inputs
        clk = 0;
        d = 0;
        // Wait 100 ns for global reset to
        finish #100;
        // Add stimulus here
    end
    always #20 clk=~clk;
    always #30 d=~d;
endmodule
```

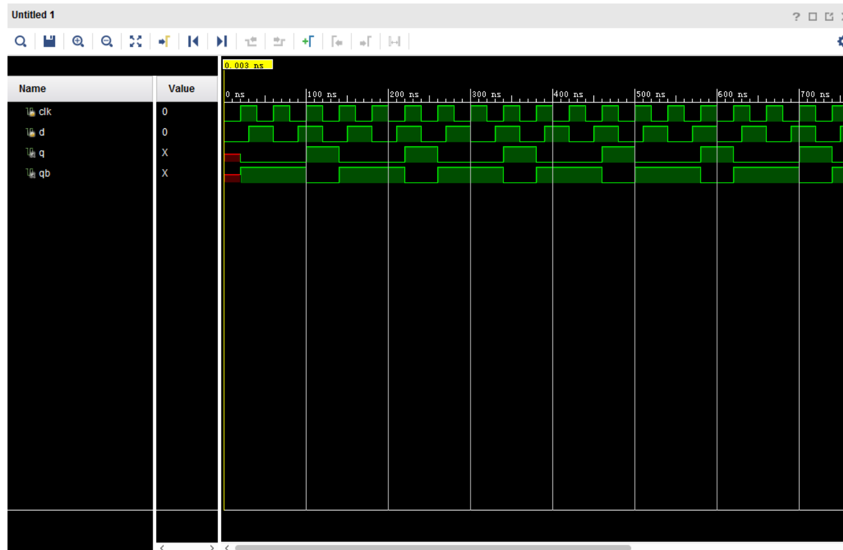
输入完成后点击左侧的 Run Simulation 进行仿真



仿真结果如图所示。

e. 分配管脚；

f. 下载。



（2） 同步复位的 D 触发器

在数字电路中，一种常见的带有同步复位控制端口的上升沿 D 触发器的逻辑电路符号如图 3.3 所示，它的功能表如表 3.2 所示。不难看出，只有在时钟信号的上升沿到来并且复位控制端口的信号有效时，D 触发器才进行复位操作， 即将输出端口 Q 的值置为逻辑 0，而把输出端口 Q 的值置为逻辑 1。

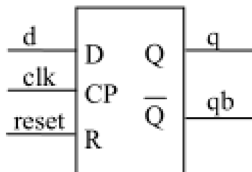


图 3.3 D 触发器电路符号

表 3.2 D 触发器的功能表

R	D	CP	Q	$\overline{Q}$
0	X	上升沿	0	1
1	X	0	保持	保持
1	X	1	保持	保持
1	0	上升沿	0	1
1	1	上升沿	1	0

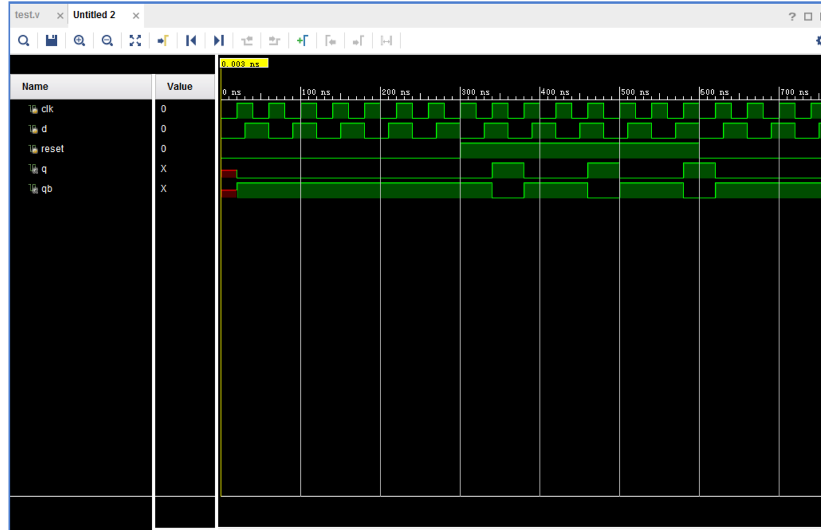
源程序如下：

```
module
        sync_rddf(clk,reset,d,q,qb);
input   clk,reset,d;
output   q,qb;
reg q,qb;
always    @(posedge    clk) begin
   if(!reset) begin
      q<=0;
      qb<=1;
```

```
end
else begin
```

```
        q<=d;
        qb<=~d;
      en
    d end
    endmodule
```

仿真结果如下：



仿真结果说明：

当复位信号 reset 为高时，同步复位 D 触发器与基本 D 触发器所实现的功能一致。

（3） 异步复位的 D 触发器

常见的带有异步复位控制端口的上升沿 D 触发器的逻辑电路符号如图 3.4 所示，它的功能表如表 3.3 所示。不难看出，只要复位控制端口的信号有效，D 触发器就会立即进行复位操作。可见，这时的复位操作是与时钟信号无关的。

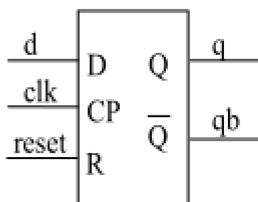


图 3.3、D 触发器电路符

表 3.2、D 触发器的功能表

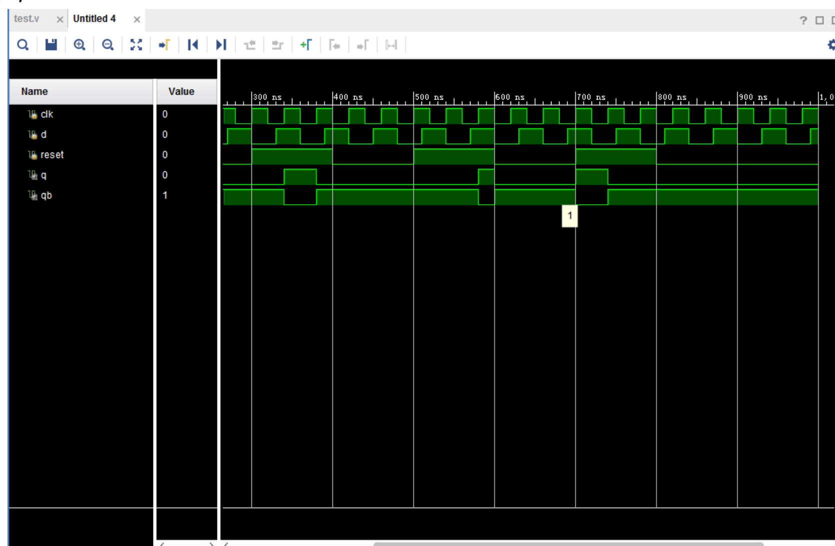
R	D	CP	Q	$\overline{Q}$
0	X	上升沿	0	1
1	X	0	保持	保持
1	X	1	保持	保持
1	0	上升沿	0	1
1	1	上升沿	1	0

源程序如下：

```
module  async_rddf(clk,reset,d,q,qb);
input clk,reset,d;
```

```
output q,qb;
reg    q,qb;
always    @(posedge clk or negedge reset)
    begin if(!reset) begin
        q<=0;
        qb<=1;
    end
    else begin
        q<=d;
        qb<=~d;
    en
d end
endmodule
```

仿真结果如下：



仿真结果说明：

观察同步复位 D 触发器与异步复位 D 触发器的仿真结果，其区别是显而易见的：如果不考虑器件本身的延迟，异步复位 D 触发器的 reset 信号为 0 时，输出信号 q 直接复位，不受时钟信号的影响。

（4） 同步置位/复位的 D 触发器

同时带有置位控制和复位控制端口的 D 触发器也是经常使用的，同样它也具有同步异步两种方式。这里我们给出同步置位/复位的 D 触发器的源程序及仿真结果，请读者根据已经介绍的内容自己实现异步置位/复位的 D 触发器。

带有同步置位/复位端口的上升沿 D 触发器的逻辑电路符号如图 3.5 所示，它的功能表如表 3.4 所示。不难看出，只有在时钟信号的上升沿到来并且同步置位/复位端口的信号有效时，D 触发器才可以进行置位或者复位操作。

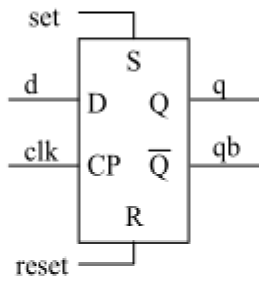


图 3.5、 电路符号

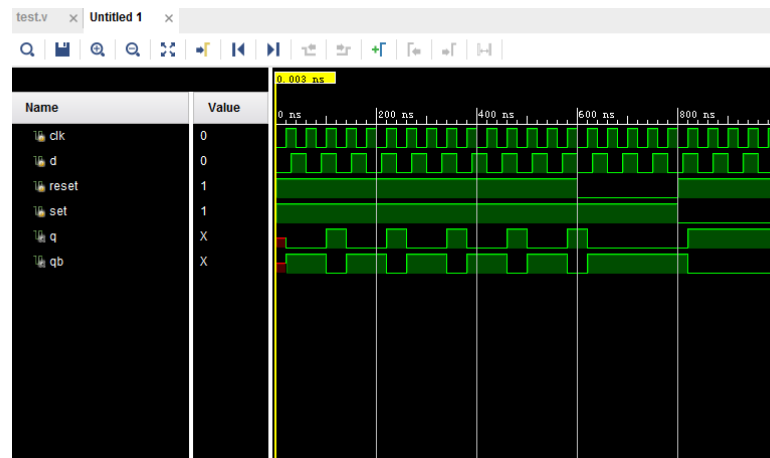
表 3.4、D 触发器的功能表

S	R	D	CP	Q	$\overline{Q}$
0	1	X	上升沿	1	0
1	0	X	上升沿	0	1
1	1	X	0	保持	保持
1	1	0	上升沿	0	1
1	1	1	上升沿	1	0

源程序如下：

```
module   sync_rsddf(clk,reset,set,d,q,qb);
input clk,reset,set;
input d;
output q,qb;
reg   q,qb;
always @(posedge clk) begin
   if(!set && reset) begin
      q<=1;
      qb<=0;
   end
   else if(set && !reset) begin
      q<=0;
      qb<=1;
   end
   else begin
      q<=d;
      qb<=~d;
   en
d end
endmodule
```

仿真结果如下：



2． 计数器的实现

（1） 加法计数器

加法计数器的动作是， 每次时钟脉冲信号 clk 为上升沿时， 计数器会将计数值加 1。 以图 3.6 为例， 它是 2bits 的计数器，所以计数值（由 Q1Q0 组成），依次是 0，1，2，3，0，1...，周而复始。

在图 3.6 的波形图里，透露了这样几个信息：

i. 一个两 bit 计数器，它所能计数的范围是 0~3（22-1）。同理，n bits 的计数器所能计数的范围是 0~2n -1。

ii. 分别由 Q0、Q1 得到的波形频率是时钟脉冲信号 clk 的 1/2、1/4，亦即是将时钟脉冲信号的 clk 频率除 2、除 4。因此图 3.6 又常被称为除 4 计数器。

iii. 由上讨论推广可知，n bits 计数器可获得的信号之多是频率除 2n 的结果。

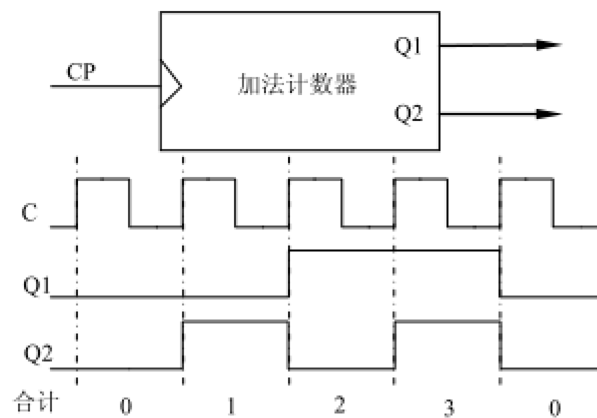
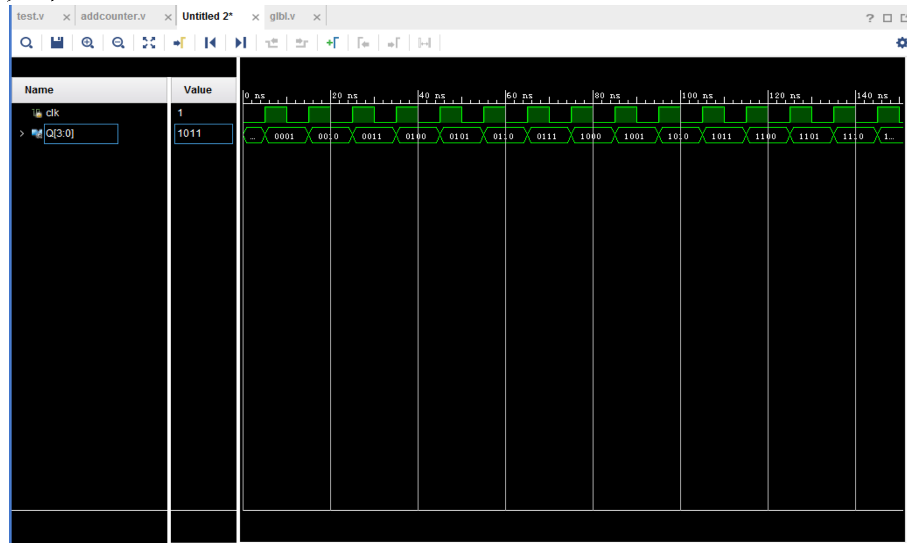


图 3.6、加法计数器的相关波形

源程序如下：

```
module addcounter(clk,Q) ;
input clk ;
output[3 :0] Q ;
reg[3 :0] Q = 4'b0000;
always @(posedge clk)begin
Q<=Q+1 ;
end
endmodul
e
```

仿真结果如下：



（2） 减法计数器

减法计数器的计数方式改成 15，14...0，因为仅是和加法计数器的技术方向不同，其它完全是一样的，因此，减法计数器的 Verilog HDL 语言描述，只需要将前面加法计数器的程序稍作修改即可。在此不再多说。

五、 实验结果

时序逻辑电路不像组合逻辑电路那样可以通过有限的 LED 灯、 七段码来指示实验结果， 从而验证硬件描述语言的正确性。因此对于一些快速的信号时序检查，我们需要借助其它仪器设备，如信号发生器、示波器等来进行设计验证。

1. D 触发器

（1） 基本 D 触发器

管脚分配如下表：

程序中管脚名	实际管脚	说明
d	B16	扩展口exp_io[0]
clk	P17	全局时钟
q	A13	扩展口exp_io[2]
qb	A15	扩展口exp_io[1]

除了时钟的分配是固定的，其它管脚分配可以自己选择，这里这样安排只是为了与信号发生器和示波器连接方便。将信号发生器接到 B18，将示波器的两通道的引脚分别接到 F13 和 B13。对输入输出波形进行观察，比较。

（2） 同步/同步复位 D 触发器

管脚分配如下表：

程序中管脚名	实际管脚	说明
D	B16	扩展口exp_io[0]
Clk	P17	全局时钟
RESET	N4	拨动开关 SW1
Q	B17	扩展口exp_io[16]
QB	A15	扩展口exp_io[1]

将信号发生器接到 B18，将示波器的两通道的引脚分别接到 F13 和 B13。对输入输出波形进行观察，比较。

（3） 同步置位/复位的 D 触发器

管脚分配如下表：

程序中管脚名	实际管脚	说明
D	B16	扩展口exp_io[0]
Clk	P17	全局时钟
RESET	N4	拨动开关 SW1
Q	B17	扩展口exp_io[16]
QB	A15	扩展口exp_io[1]
SET	A16	扩展口exp_io[17]

将信号发生器接到 B18，将示波器的两通道的引脚分别接到 F13 和 B13。对输入输出波形进行观察，比较。

（4） 计数器

管脚分配如下表：

程序中管脚名	实际管脚	说明
Clk	P17	全局时钟
Q0	B16	扩展口exp_io[0]
Q1	B17	扩展口exp_io[16]
Q2	A15	扩展口exp_io[1]
Q3	A16	扩展口exp_io[17]

一般示波器只有两个通道，所以要一起观察五个信号是不可能的，建议进行如下操作：首先将示波器的两通道的引脚分别接到 B8 和 B2，观察 Q0 是否是 clk 的二分频；然后再以 Q0 为参照，观察和比较 Q1、Q2、Q3 与 Q0 的频率、相位关系。

实验四：状态机

一、 实验目的

1. 对有限状态机(FSM)做初步了解。

二、 实验内容

1. Gray 编码和 One-hot 编码两种状态机；
2. 触发器部分和组合逻辑部分结合与分开两种状态机。

三、 实验要求

1 对程序中状态和输出稍作修改，在 VIVADO 环境下进行时序仿真；
2 下载至实验板，观察结果。

四、 实验步骤

有限状态机是由寄存器组和组合逻辑构成的硬件时序电路，其状态（即由寄存器组的 1 和 0 的组合状态所构成的有限个状态）只可能在同一时钟跳变沿的情况下才能从一个状态转向另一个状态，究竟转向哪一状态还是留在原状态不但取决于各个输入值，还取决于当前所在状态。（这里指的是米里 Mealy 型有限状态机，而莫尔 Moore 型有限状态机究竟转向哪一状态只取决于当前状态。）

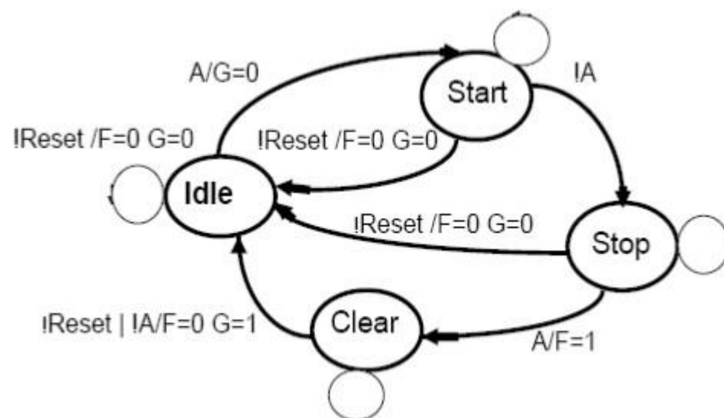


图 4.1、用三种不同编码所实现的状态图

【例1】采用 Gray 编码的状态机

源程序：

```
module ztj(clock,reset,a,g,f
        );
input  a,reset,clock;
output g,f;
reg g,f;
reg[1:0]  state;
parameter start=2'b00,
          stop  =2'b01,
          clear=2'b10,
          idle =2'b11;
always@(posedge clock)
begin
if(!reset)
        begin
        state<=idle;
        f<=0; g<=0;
        end
else
        begin
        case(state)
            start:if(!a)
                        state<=stop;
                        else
                        state<=start;
            stop :if(a)
                            begin
                            state<=clear;
                            f<=1;
                            end
                        els
                        e   state<=stop;
            clear:if(!a)
                            begin
                            state<=idle;
                            f<=0; g<=1;
                            end

                        else  state<=clear;
            idle :if(a)
                            begin
```

```
                    state<=start;
                    g<=0;
                    end
                els
                e   state<=idle;
        endcas
    en  e
end d
endmodule
```

【例2】采用One-hot 编码的状态机

源程序：

```
module fsm (Clock, Reset, A, F, G);
input Clock, Reset, A; output F,G;
reg F,G;
reg [3:0] state ;
parameter Idle = 4'b1000,
Start = 4'b0100,
Stop = 4'b0010,
Clear = 4'b0001;
always @(posedge Clock)
begin
  if (!Reset)
  begin
    state <= Idle; F<=0; G<=0;
  end
  else
  begin
    case (state)
    Idle: begin
      if (A) begin
        state  <= Start;
        G<=0;
      end
      else state <= Idle;
    end
    Start: begin
      if (!A) state <= Stop;
      else state <= Start;
    end
    Stop: begin
      if (A) begin
```

```
state <= Clear;
F <= 1;
```

```
          end
          else state <= Stop;
        end
        Clear: begin
          if (!A) begin
            state <=Idle;
            F<=0; G<=1;
          end
          else state <= Clear;
        end
        default: state <=Idle;
        endcase
    en
d end
endmodule
```

例 1 中采用 Gray 编码，例 2 中采用的是 One-hot 编码。究竟采用哪一种编码好要看具体情况而定。对于用 FPGA 实现的有限状态机建议采用 One-hot 码，因为虽然采用 One-hot 编码多用了两个触发器，但所用组合电路可省下许多，因而使电路的速度和可靠性有显著提高，而总的单元数并无显著增加。采用了 One-hot 编码后有了多余的状态，就有一些不可到达的状态，为此在 CASE 语句的最后需要增加 default 分支项，以确保多余状态能回到 Idle 状态。

【例 3】利用状态机编写的流水灯

源程序：

```
module  led(clk,data,sw);
input  clk,sw; output[3:0]
data;
reg clk1s;
parameter  max=5000000;
reg[1:0] state=2'b00;
reg[30:0] n;
reg[3:0] data;
always @(posedge clk)begin
  if(n==max)begin
    if(!clk1s)clk1s<=1'b1;
    else clk1s<=1'b0;
    n<=0;
    end
  else n<=n+1;
```

```
end
always @(posedge clk1s)begin
```

```
    case(state)
    2'b00:begi
    n
       state<=2'b01;
       if(sw)begin
             data<=4'b1000;
       end
       else begin
             data<=4'b0111;
       en
    d end
    2'b01:begin
       state<=2'b10;
       if(sw)begin
             data<=4'b0100;
    end
       else begin
             data<=4'b1011;
       en
    d end
    2'b10:begin
       state<=2'b11;
       if(sw)begin
             data<=4'b0010;
       end
       else begin
             data<=4'b1101;
       en
    d end
    2'b11:begin
       state<=2'b00;
       if(sw)begin
             data<=4'b0001;
       end
       else begin
             data<=4'b1110;
       end
    end
    endcas
    e
 end
 endmodul
```

e

例三是采用 Gray 码编码的流水灯程序，通过拨码开关控制，可以显示两种流水灯。

五、 实验结果

三种编码方式的仿真波形是一致的，如下图：

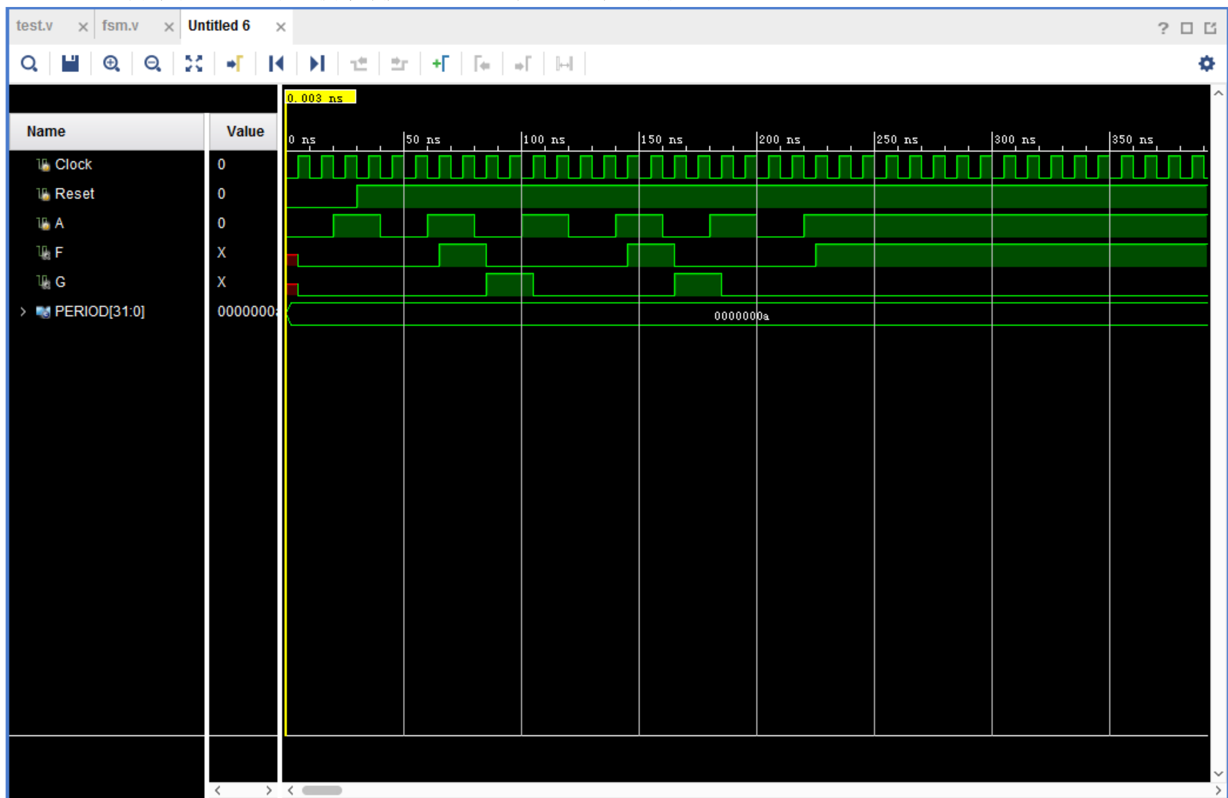


图 4.2、三种编码方式的仿真波形

流水灯的实验结果也完全符合要求，通过拨码开关的选择可以显示两种方式的流水灯。

管脚分配表：

程序中管脚名	实际管脚	说明
SW	N4	拨动开关 SW1
DATA[0]	K2	LED 0
DATA[1]	J2	LED 1
DATA[2]	J3	LED 2
DATA[3]	H4	LED 3

实验结果对照表：

拨动开关 1 脚	LED D1	LED D2	LED D3	LED D4
1	从左至右逐次只亮一个灯			
0	从左至右逐次只灭一个灯			

实验五：模块化调用

一、 实验目的

1 对Verilog HDL 的模块化设计做初步了解；

2 体会主流设计 “自顶向下”设计思想。

二、 实验内容

1. 实现顶层文件调用其他模块。

三、 实验要求

1. 在接下来的实验中熟练掌握模块化调用。

四、 实验步骤

以下为程序：

程序一：

```
module mux1(a,b,c);
input a,b;
output c;
assign c=a&b;
endmodule
```

程序二：

```
module mux2(b,c,d);
input b,c;
output d;
assign d=c&b;
endmodule
```

程序三：

```
module mux3(d,c,e);
input d,c;
output e;
assign e=c&d;
endmodule
```

以上三个程序都是最简单的程序，也是我们这次用到的子模块；

下面是主模块：（顶层文件）

```
module  mux_top(a,b,e);
input a,b;
output  e;
wire b;
wire c;
wire d;
//对于第一个模块的调用 其中 mux1 为子模块名称，mux_one 为在顶层文件中引用的名称。
mux1 mux_one (.a(a),.b(b),.c(c));
//====================================
mux2 mux_two (.b(b),.c(c),.d(d));
//====================================
mux3 mux_three (.d(d),.c(c),.e(e));
endmodule
```

在软件中使用结果：

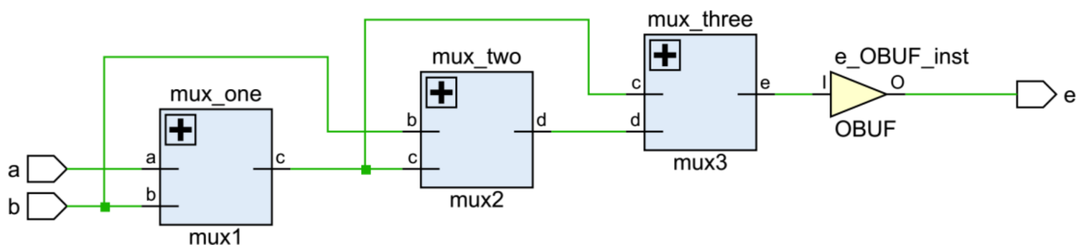
1.建立工程；

2.分别编写子模块和主模块的代码；

3.主模块自动设为顶层文件；（右击设置）

4.编译文件；

5.RTL 视图；



实验六：数码管显示

一、 实验目的

1. 学习动态数码管的工作原理；
2. 实现对EGO1开发板四位动态数码管的控制；

二、 实验内容

实现对EGO1开发板四位动态数码管的控制，使其能够正常工作；

三、 实验要求

在EGO1开发板上显示想要的数字。

四、 实验背景知识

1. LED 数码管基础知识

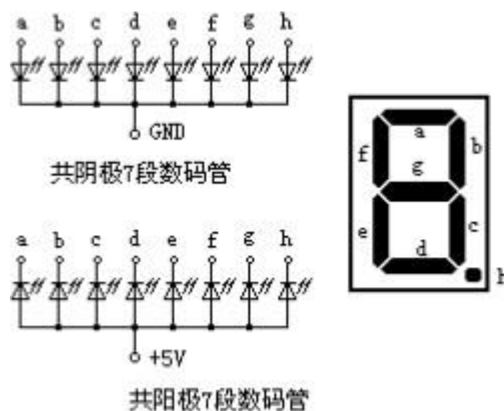


图 6.1、数码管原理图

在数码管上显示数字就是将相应的段位点亮组成要显示的数字，共阴数码管的码值表如下所示，‘1’代表相应的管脚输出高电平，点亮相应段位，‘0’代表相应的管脚输出低电平，不点亮相应段位。

表 6.1 、共阴数码管对应的码值表

显示	a	b	c	d	e	f	g	dp(h)
0	1	1	1	1	1	1	0	0
1	0	1	1	0	0	0	0	0
2	1	1	0	1	1	0	1	0
3	1	1	1	1	0	0	1	0
4	0	1	1	0	0	1	1	0
5	1	0	1	1	0	1	1	0
6	1	0	1	1	1	1	1	0
7	1	1	1	0	0	0	0	0
8	1	1	1	1	1	1	1	0
9	1	1	1	1	0	1	1	0

2． 动态数码管原理

EGO1 开发板上使用的是共阴极动态数码管， 这种数码管有四个共阴极分别选通对应的每位数码管， 四位数码管的八个段码脚连接在一起。其硬件连接图如图5.2所示。

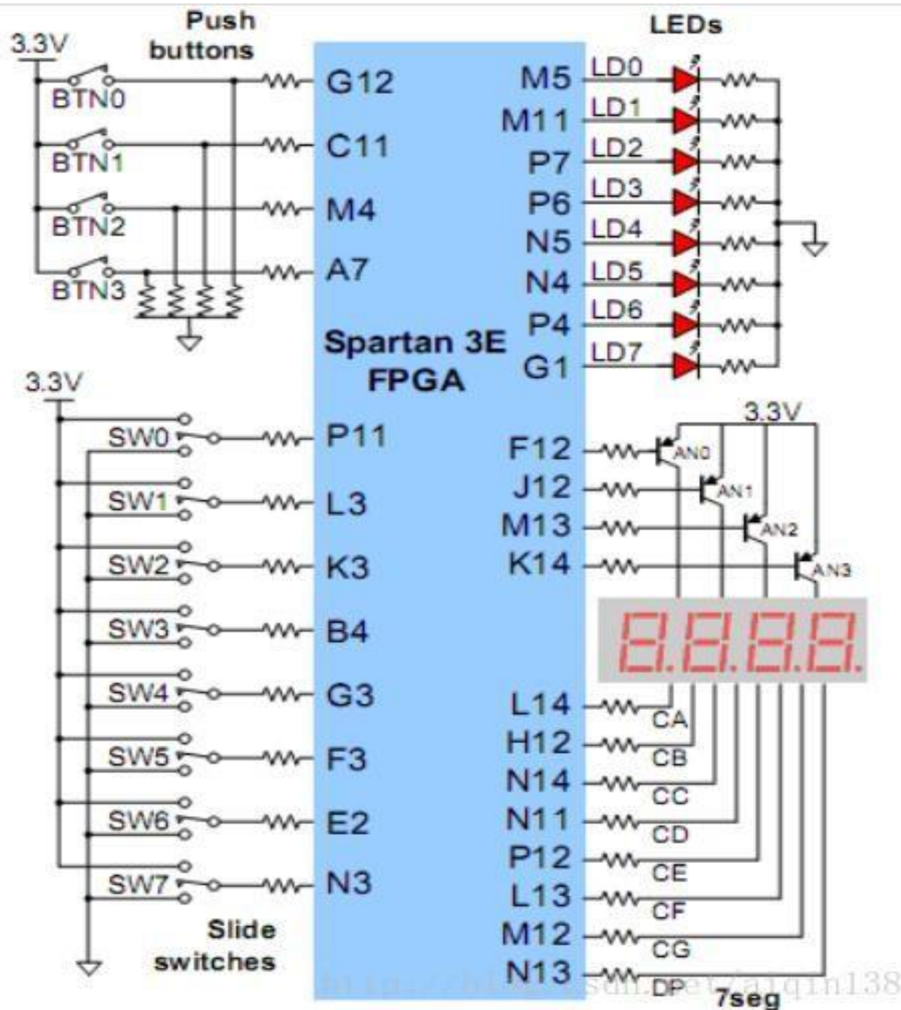


图 6.2、 四位动态数码管硬件连接图

动态数码管显示的原理是： 每次选通其中一位， 送出这位要显示的内容，然后一段时间后选通下一位送出对应数据，4 个数码管这样依次选通并送出相应的数据，结束后再重复进行。这样只要选通时间选取的合适，由于人眼的视觉暂留，数码管看起来就是连续显示的。

五、 实验方案及实现

1、数码管显示的设计共分3个模块：

（1） 数码管封装模块

（2） 数码管设计模块

（3） 顶层模块

- 数码管封装模块代码：

```
//数码管 ip 核
module smg_ip_model(clk,data,sm_wei,sm_duan);
input clk;
input [15:0] data;
output [3:0] sm_wei;
output [7:0] sm_duan;
//----------------------------------------------------------
//分频
integer clk_cnt;
reg clk_400Hz;
always @(posedge clk)
if(clk_cnt==32'd100000)
begin clk_cnt <= 1'b0; clk_400Hz <= ~clk_400Hz;end
else
clk_cnt <= clk_cnt + 1'b1;
//----------------------------------------------------------
//位控制
reg [3:0]wei_ctrl=4'b1110;
always @(posedge clk_400Hz)
wei_ctrl <= {wei_ctrl[2:0],wei_ctrl[3]};
//段控制
reg [3:0]duan_ctrl;
always @(wei_ctrl)
case(wei_ctrl)
4'b1110:duan_ctrl=data[3:0];
4'b1101:duan_ctrl=data[7:4];
4'b1011:duan_ctrl=data[11:8];
```

```
4'b0111:duan_ctrl=data[15:12];
default:duan_ctrl=4'hf;
endcase
//--------------------------------------------------------------
//解码模块
reg  [7:0]duan;
always @(duan_ctrl)
case(duan_ctrl)
4'h0:duan=8'b0011_1111;//0
4'h1:duan=8'b0000_0110;//1
4'h2:duan=8'b0101_1011;//2
4'h3:duan=8'b0100_1111;//3
4'h4:duan=8'b0110_0110;//4
4'h5:duan=8'b0110_1101;//5
4'h6:duan=8'b0111_1101;//6
4'h7:duan=8'b0000_0111;//7
4'h8:duan=8'b0111_1111;//8
4'h9:duan=8'b0110_1111;//
9
4'ha:duan=8'b0111_0111;//
a
4'hb:duan=8'b0111_1100;//
b
4'hc:duan=8'b0011_1001;//
c
4'hd:duan=8'b0101_1110;//
d
4'he:duan=8'b0111_1000;//
e
4'hf:duan=8'b0111_0001;//
f
// 4'hf:duan=8'b1111_1111;//不显
示default : duan =
8'b0011_1111;//0 endcase
//--------------------------------------------------------------
assign sm_wei =~wei_ctrl;
assign sm_duan = duan;
endmodule
```

- 数码管设计模块

```
//测试数码管 ip
module  test(clk,data);
input clk;
```

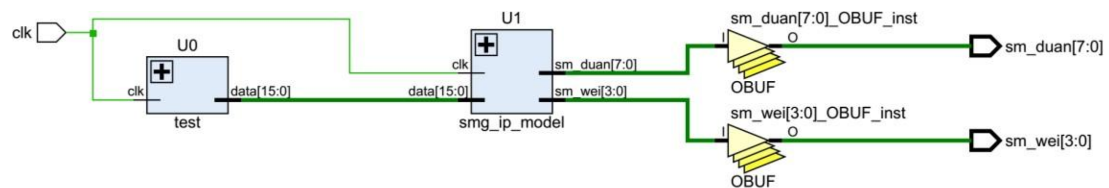
```
output [15:0]data;
//-----------------------------------------------------------
//分频 1Hz
reg clk_1Hz;
integer clk_1Hz_cnt;
always @(posedge clk)
if(clk_1Hz_cnt==32'd25000000-1)
begin clk_1Hz_cnt <= 1'b0; clk_1Hz <= ~clk_1Hz;end
else
```

```
clk_1Hz_cnt <= clk_1Hz_cnt + 1'b1;
//----------------------------------------------------------
//循环显示 0-9
reg [39:0]disp=40'h1234567890;
reg [15:0]data;
always @(posedge clk_1Hz)
begin
disp <= {disp[35:0],disp[39:36]};
data <= disp[39:24];
end
endmodul
e
```

- 顶层模块

```
//顶层模块
module smg_ip(clk,sm_wei,sm_duan);
input clk;
output [3:0]sm_wei;
output  [7:0]sm_duan;
//----------------------------------------------------------
wire [15:0]data;
wire [3:0]sm_wei;
wire [7:0]sm_duan;
//----------------------------------------------------------
test U0 (.clk(clk),.data(data));
smg_ip_model U1 (.clk(clk),.data(data),.sm_wei(sm_wei),.sm_duan(sm_duan));
endmodule
```

2、数码管显示实验的RTL视图：



六、 实验结果

1.将程序写好，编译通过后分配好管脚（管脚分配如下所示），然后再次编译生成下载文件。

约束文件：

```
set_property PACKAGE_PIN P17 [get_ports clk] set_property
PACKAGE_PIN G2 [get_ports {sm_wei[0]}] set_property
PACKAGE_PIN C2 [get_ports {sm_wei[1]}] set_property
PACKAGE_PIN C1 [get_ports {sm_wei[2]}] set_property
PACKAGE_PIN H1 [get_ports {sm_wei[3]}] set_property
PACKAGE_PIN B4 [get_ports {sm_duan[0]}] set_property
PACKAGE_PIN A4 [get_ports {sm_duan[1]}] set_property
PACKAGE_PIN A3 [get_ports {sm_duan[2]}] set_property
PACKAGE_PIN B1 [get_ports {sm_duan[3]}] set_property
PACKAGE_PIN A1 [get_ports {sm_duan[4]}] set_property
PACKAGE_PIN B3 [get_ports {sm_duan[5]}] set_property
PACKAGE_PIN B2 [get_ports {sm_duan[6]}] set_property
PACKAGE_PIN D5[get_ports {sm_duan[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports clk] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_wei[0]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_wei[1]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_wei[2]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_wei[3]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[0]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[1]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[2]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[3]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[4]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[5]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[6]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[7]}]
```

2.将程序下载到开发板上，上电后观察数码管情况。如果不能达到效果检查源程序，改好后再试。

实验七：交通灯

一、 实验目的

1. 综合运用 Verilog HDL 语言进行时序设计

二、 实验内容

1. 编写时间控制程序，利用交通灯实验子板，实现东西，南北向的交通灯计数并亮灯的程序；

2．子板实现所有显示方面的功能，包括十进制倒数计数和红绿黄三色灯的轮流点亮。

三、 实验要求

1. 两个方向各种灯亮的时间能够进行设置和修改；

2. 交通灯控制器的状态表见表 7-1

表 7-1 交通灯控制器的状态转换表

东西方向			南北方向		
绿灯	黄灯	红灯	绿灯	黄灯	红灯
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0

四、 实验方案及实现

1、为了在八段数码管上正确显示十进制数据，设计一个函数，程序即上述实验五中的数码管封装模块，请同学自行参考并编写，RTL 视图如图 7.1 所示。

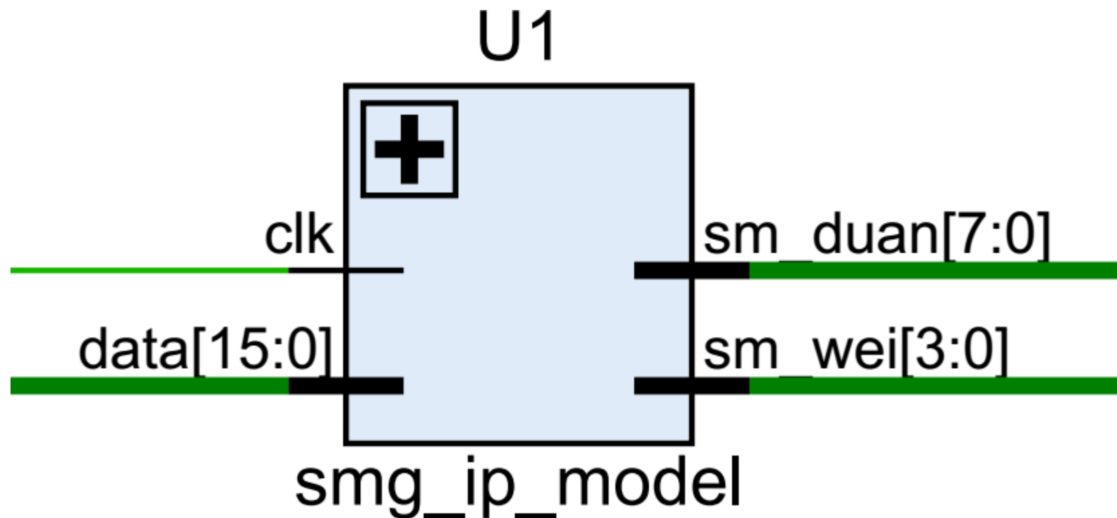


图 7.1、数码管封装模块RTL

2、主程序使用 case 语句，根据变量 Stage 的值确定当前状态，并加以执行（程序功能如下）：

Stage=2’b00 代表南北方向绿灯亮

Stage=2’b01 代表东西方向绿灯亮

Stage=2’b10 代表南北方向绿灯转红灯期间的黄灯亮

Stage=2’b11 代表东西方向绿灯转红灯期间的黄灯亮绿

灯亮灯的时间为 20S；

黄灯亮灯的时间为 10S；

主程序如下：

```
module test(clk,data,out_LED3_NS,out_LED3_WE);
input clk;
output [15:0]data;
output [2:0] out_LED3_NS,out_LED3_WE;
//分频 1Hz
reg clk_1Hz;
integer clk_1Hz_cnt;
always @(posedge clk)
if(clk_1Hz_cnt==32'd25000000-1)
begin clk_1Hz_cnt <= 1'b0; clk_1Hz <= ~clk_1Hz;end
else
clk_1Hz_cnt <= clk_1Hz_cnt + 1'b1;
//--------------------------------------------------------
//循环显示 0-9
```

```
reg [15:0]data;
reg[2:0] out_LED3_NS,out_LED3_WE;
reg[3:0] Time_10=2'd2,Time_1=2'd0;
reg[1:0] Stage=2'b00;
always @(posedge clk_1Hz)
begin
case(Stage)
    2'b00:
    begin//NS pass
    if((Time_10==0) & (Time_1==0)) begin
        Stage<=2'b11;
        Time_10<=4'd1;
        Time_1 <=4'd0;
        end
    else
        begin
        if(Time_1==0)
            begin
            Time_1<=4'd9;
            Time_10<=Time_10-1;
            end
        else
            begin
            Time_1<=Time_1-1;
            end
        end
    data[15:8]<={Time_10,Time_1};
    data[7:0]<={Time_10,Time_1};
    out_LED3_NS<=3'b001;
    out_LED3_WE<=3'b100;
    end
    2'b01:
    begin//WE pass
    if((Time_10==0) & (Time_1==0)) begin
        Stage<=2'b10;
        Time_10<=4'd1;
        Time_1 <=4'd0;
        end
    els
    e   begin
        if(Time_1==0)
            begin
```

```
                Time_1<=4'd9;
                Time_10<=Time_10-1;
                end
        else
                begin
                Time_1<=Time_1-1;
        end     end
data[15:8]<={Time_10,Time_1};
data[7:0]<={Time_10,Time_1};
out_LED3_WE<=3'b001;
out_LED3_NS<=3'b100;
end
2'b10:
begin//Yellow to NS pass
if((Time_10==0) & (Time_1==0))
        begin
        Stage<=2'b00;
        Time_10<=4'd2;
        Time_1 <=4'd0;
        end
els
e       begin
        if(Time_1==0)
                begin
                Time_1<=4'd9;
                Time_10<=Time_10-1;
                end
        else
                begin
                Time_1<=Time_1-1;
                end
        end
data[15:8]<={Time_10,Time_1};
data[7:0]<={Time_10,Time_1};
out_LED3_NS<=3'b010;
out_LED3_WE<=3'b010;
end
2'b11:
begin//Yellow to WE pass
if((Time_10==0) & (Time_1==0))
        begin
        Stage<=2'b01;
```

```
Time_10<=4'd2;
```

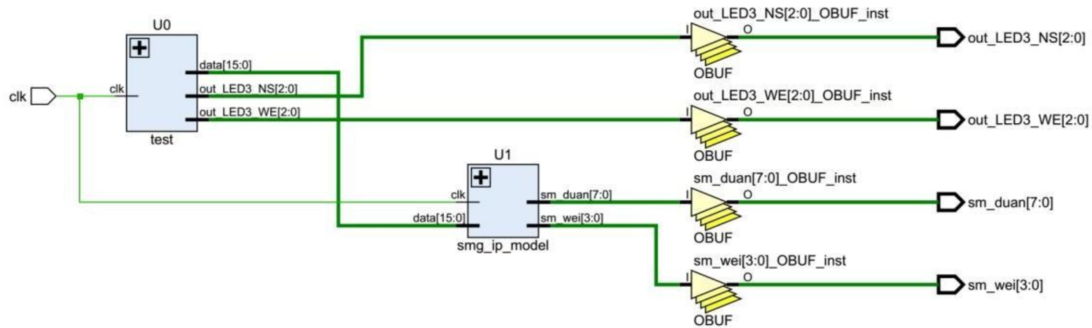
```
        Time_1 <=4'd0;
        end
    els
    e   begin
        if(Time_1==0)
            begin
            Time_1<=4'd9;
            Time_10<=Time_10-1;
            end
        else
            begin
            Time_1<=Time_1-1;
            end
        end
    data[15:8]<={Time_10,Time_1};
    data[7:0]<={Time_10,Time_1};
    out_LED3_NS<=3'b010;
    out_LED3_WE<=3'b010;
    end
    default:
        begin
        Stage<=2'b00;
        Time_10<=4'd2;
        Time_1<=4'd0;
        end
endcase
end
endmodul
e
```

3、顶层文件如下：

```
module smg(clk,sm_wei,sm_duan,out_LED3_NS,out_LED3_WE);
input clk;
output [3:0]sm_wei;
output  [7:0]sm_duan;
output [2:0] out_LED3_NS,out_LED3_WE;
//---------------------------------------------------------
wire [15:0]data;
wire [3:0]sm_wei;
wire [7:0]sm_duan;
wire [2:0] out_LED3_NS,out_LED3_WE;
//---------------------------------------------------------
test U0 (.clk(clk),.data(data),.out_LED3_NS(out_LED3_NS),.out_LED3_WE(out_LED3_WE));
```

```
smg_ip_model U1 (.clk(clk),.data(data),.sm_wei(sm_wei),.sm_duan(sm_duan));
endmodule
```

4、总体设计的 RTL 视图：



五、 实验结果

1、本实验是在交通灯实验子板上实现的，管脚分配如下

set_property PACKAGE_PIN P17 [get_ports clk] set_property
PACKAGE_PIN G2 [get_ports {sm_wei[0]}] set_property
PACKAGE_PIN C2 [get_ports {sm_wei[1]}] set_property
PACKAGE_PIN C1 [get_ports {sm_wei[2]}] set_property
PACKAGE_PIN H1 [get_ports {sm_wei[3]}] set_property
PACKAGE_PIN B4 [get_ports {sm_duan[0]}] set_property
PACKAGE_PIN A4 [get_ports {sm_duan[1]}] set_property
PACKAGE_PIN A3 [get_ports {sm_duan[2]}] set_property
PACKAGE_PIN B1 [get_ports {sm_duan[3]}] set_property
PACKAGE_PIN A1 [get_ports {sm_duan[4]}] set_property
PACKAGE_PIN B3 [get_ports {sm_duan[5]}] set_property
PACKAGE_PIN B2 [get_ports {sm_duan[6]}] set_property
PACKAGE_PIN D5 [get_ports {sm_duan[7]}]
set_property PACKAGE_PIN K2 [get_ports {out_LED3_NS[2]}]
set_property PACKAGE_PIN J2 [get_ports {out_LED3_NS[1]}]
set_property PACKAGE_PIN J3 [get_ports {out_LED3_NS[0]}]
set_property PACKAGE_PIN H4 [get_ports {out_LED3_WE[2]}]
set_property PACKAGE_PIN J4 [get_ports {out_LED3_WE[1]}]
set_property PACKAGE_PIN G3 [get_ports{out_LED3_WE[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports clk] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_wei[0]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_wei[1]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_wei[2]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_wei[3]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[0]}] set_property
IOSTANDARD LVCMOS33 [get_ports {sm_duan[1]}] set_property
IOSTANDARD LVCMOS33 [get_ports{sm_duan[2]}]

```
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out_LED3_NS[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out_LED3_NS[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out_LED3_NS[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out_LED3_WE[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out_LED3_WE[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {out_LED3_WE[0]}]
```

2、将实验板上电，下载程序到 FPGA 芯片中。

3、观察实验结果。

实验八：秒表的设计

一、 实验目的

1. 学习动态数码管的工作原理；
2. 实现 FPGA 对四位动态数码管的控制；
3. 熟悉模块化编程的操作流程。

二、 实验内容

1. 实现 FPGA 对四位动态数码管的控制，使其能够正常工作；
2. 应用四位动态数码管做为显示器件设计一个简单秒表。

三、 实验要求

1. 秒表的最小计时单位为 0.1s；
2. 设计的秒表能够实现暂停和继续计时的功能。

四、 实验方案及实现

1、秒表的设计共分 3 个模块：

（1）、 数码管显示模块

（2）、 计时模块

（3）、 顶层模块

- 数码管显示模块
  程序即上述实验五中的数码管封装模块，请同学自行参考并编写。
- 计时模块

```
module time_counter(clk,key1,rst,data);
input clk;
input key1;
input rst;
output [15:0] data;
reg [15:0] data;
reg clk_1Hz;
integer clk_1Hz_cnt;
```

```
always @(posedge clk)
if(clk_1Hz_cnt==32'd25000000-1)
begin clk_1Hz_cnt <= 1'b0;
clk_1Hz <= ~clk_1Hz;
end
else
clk_1Hz_cnt <= clk_1Hz_cnt + 1'b1;

reg[3:0]  time_1=0,time_10=0,time_100=0,time_1000=0;
always@(posedge clk_1Hz)
begin
    if(rst) begin
    time_1<=0;
    time_10<=0;
    time_100<=0;
    time_1000<=0;
    data[15:0] <={time_1000,time_100,time_10,time_1};
end
    els
    e    begin
            if(key1==0)
            begin
                if(time_1 < 4'b1001)
                    begin
                    time_1 <= time_1+1'b1;
                    data[15:0] <={time_1000,time_100,time_10,time_1};
                    end
                else if(time_10 < 4'b1001)
                    begin
                    time_10 <= time_10+1'b1;
                    time_1 <= 4'b0000;
                    data[15:0] <={time_1000,time_100,time_10,time_1};
                    end
                else if(time_100 < 4'b1001)
                    begin
                    time_100 <= time_100+1'b1;
                    time_10 <= 4'b0000;
                    time_1 <= 4'b0000;
                    data[15:0] <={time_1000,time_100,time_10,time_1};
                    end
                else if(time_1000 < 4'b1001)
                    begin
```

```
time_10000 <= time_10000+1'b1;
```

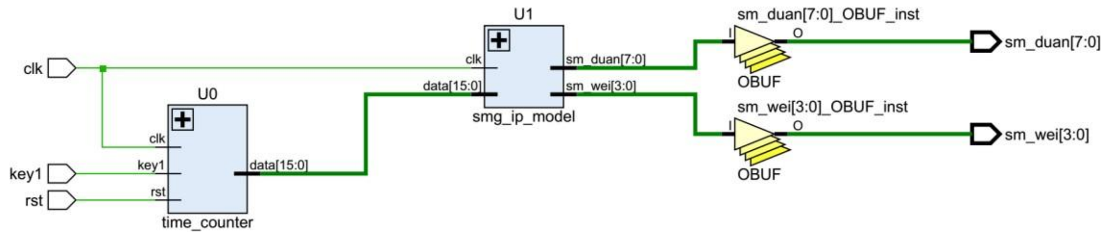
```
                        time_100  <=  4'b0000;
                        time_10   <=  4'b0000;
                        time_1 <= 4'b0000;
                        data[15:0] <={time_1000,time_100,time_10,time_1};
                        end
                    els
                    e   begin
                        time_1000 <= 4'b0000;
                        time_100  <=  4'b0000;
                        time_10   <=  4'b0000;
                        time_1 <= 4'b0000;
                        data[15:0] <={time_1000,time_100,time_10,time_1};
                        end

                    end
                else
                    begin
                    time_1000 <= time_1000;
                    time_100   <= time_100;
                    time_10    <= time_10;
                    time_1     <= time_1;
                    data[15:0] <={time_1000,time_100,time_10,time_1};
                    end
            en
        end d
endmodule
```

● 顶层模块

```
module  mb(clk,key1,rst,sm_wei,sm_duan);
input clk;
input key1;
input rst;
output [3:0]sm_wei;
output  [7:0]sm_duan;
//---------------------------------------------------------
wire [15:0]data;
wire [3:0]sm_wei;
wire [7:0]sm_duan;
//---------------------------------------------------------
time_counter U0 (.clk(clk),.rst(rst),.key1(key1),.data(data));
smg_ip_model U1 (.clk(clk),.data(data),.sm_wei(sm_wei),.sm_duan(sm_duan));
endmodule
```

2、总体设计的 RTL 视图如下：



五、 实验结果

1、将程序写好，编译通过后分配好管脚（管脚分配如下所示），然后再次编译生成下载文件。

```
set_property PACKAGE_PIN P17 [get_ports clk]
set_property PACKAGE_PIN R11 [get_ports key1] set_property
PACKAGE_PIN P15 [get_ports rst] set_property PACKAGE_PIN G2
[get_ports {sm_wei[0]}] set_property PACKAGE_PIN C2
[get_ports {sm_wei[1]}] set_property PACKAGE_PIN C1
[get_ports {sm_wei[2]}] set_property PACKAGE_PIN H1
[get_ports {sm_wei[3]}] set_property PACKAGE_PIN B4
[get_ports {sm_duan[0]}] set_property PACKAGE_PIN A4
[get_ports {sm_duan[1]}] set_property PACKAGE_PIN A3
[get_ports {sm_duan[2]}] set_property PACKAGE_PIN B1
[get_ports {sm_duan[3]}] set_property PACKAGE_PIN A1
[get_ports {sm_duan[4]}] set_property PACKAGE_PIN B3
[get_ports {sm_duan[5]}] set_property PACKAGE_PIN B2
[get_ports {sm_duan[6]}] set_property PACKAGE_PIN D5
[get_ports {sm_duan[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports clk] set_property
IOSTANDARD LVCMOS33 [get_ports key1] set_property IOSTANDARD
LVCMOS33 [get_ports rst] set_property IOSTANDARD LVCMOS33
[get_ports {sm_wei[0]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_wei[1]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_wei[2]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_wei[3]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_duan[0]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_duan[1]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_duan[2]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_duan[3]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_duan[4]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_duan[5]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_duan[6]}] set_property IOSTANDARD LVCMOS33
[get_ports {sm_duan[7]}]
```

2、将程序下载到开发板上，上电后观察秒表计数情况，按住 key1 是否停止计时，按下 key2 后是否继续计时。如果不能达到效果检查源程序，改好后再试。

实验九：蜂鸣器演奏实验

一、 实验目的

1. 使用 FPGA 控制蜂鸣器演奏乐曲梁祝中的一段；
2. 初步学会利用结构建模方法设计程序。

二、 实验内容

1. 利用时钟分频进行音调和音长的设定；
2. 利用整个程序，播放梁祝乐曲；
1. 让 LED 灯随着音乐的节拍显示，分高中低三种频率显示。

三、 实验要求

1. 将时钟频率分别分成 6MHz 和4Hz 两种；
2. 利用功能框图建立整个程序，清晰的播放出梁祝乐曲。

四、 实验背景知识

乐曲演奏的两个基本参数是每个音符的频率值（音调）及其持续的时间（音长）。因此只要控制输出到扬声器的激励信号的频率和持续时间，就可以发出连续的音乐声。

1、音调的控制

频率的高低决定音调的高低。简谱中从低音1 到高音1 的每个音名对应的频率如下表所示。

音名	频率(Hz)	音名	频率(Hz)	音名	频率(Hz)
低音 1	261.6	中音 1	523.3	高音 1	1046.5
低音 2	293.7	中音 2	587.3	高音 2	1174.7
低音 3	329.6	中音 3	659.3	高音 3	1318.5
低音 4	349.2	中音 4	698.5	高音 4	1396.9
低音 5	392	中音 5	784	高音 5	1568
低音 6	440	中音 6	880	高音 6	1760
低音 7	493.9	中音 7	987.8	高音 7	1975.5

考虑到如果基频过低，则由于分频比太小，造成四舍五入后误差较大；如果基频过高，虽然误差减小了，但是分频数变大。综合以上两个因素，选择5MHz 作为基频。由于实验板上没有5MHz 的时钟频率，所以必须先分频。

本实验要演奏的《梁祝》，各音阶频率及分频比见下表。

音名	分频比	预置数	音名	分频比	预置数
低音3	7585	8798	中音2	4257	12126
低音5	6378	10005	中音3	3792	12591
低音6	5682	10701	中音5	3189	13194
低音7	5062	11321	中音6	2841	13542
低音1	9557	6826	高音1	2389	13994

为了减小输出的偶次谐波分量，输出到扬声器的波形应为对称方波，因此在扬声器前要加一个二分频。表中给出了各音阶频率时计数器不同的预置数。采用加载预置数实现分频的方法比采用反馈复零法节约资源，实现起来也容易些。

对于乐曲中的休止符，只要令分频系数设为0，即初始值为2e14-1=16383 即可，此时扬声器将不会发出声。

2、音长的控制

音符的持续时间须根据乐曲的速度及每个音符的节拍数来确定。

本例演奏的梁祝片段，最短的音符为四分音符，如果将全音符的持续时间设为1s 的话，则只需要再提供一个4Hz的时钟频率即可产生四分音符的时长。

演奏电路的原理图如下图所示：

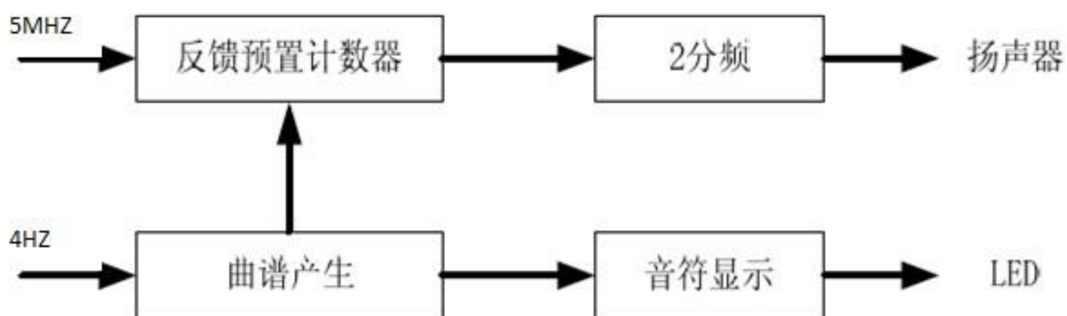


图9.1 演奏电路原理示意图

3、乐曲演奏主要源程序：

```
module song(clk,reset,speaker);
input clk;
input reset;
output speaker;
//output[3:0] high,med,low;
```

```
reg[3:0] high,med,low;
reg[13:0] divider,origin;
reg[7:0] counter;
reg[2:0] cnt;
reg[23:0] cnt1;
reg speaker;
wire carry;
assign carry=(divider==16383);
reg clk_5MHZ;
always@(posedge clk or posedge reset)
begin
if(reset)
    begin
    cnt<=0;
    clk_5MHZ<=0;
    end
else
    begin
    if(cnt==4)
        begin
        cnt<=0;
        clk_5MHZ<=~clk_5MHZ;
        end
    else
    cnt<=cnt+1;
    end
end

reg clk_4HZ;
always@(posedge clk or posedge reset)
begin
if(reset)
    begin
    cnt1<=0;
    clk_4HZ<=0;
    end
else
    begin
    if(cnt1==6249999)
        begin
        cnt1<=0;
```

```
clk_4HZ<=~clk_4HZ;

end
```

```
        else
                cnt1<=cnt1+1;
        end
en
d

always @(posedge clk_5MHZ)
begin
if(carry)  divider=origin;
else  divider=divider+1;
end
always @(posedge carry)
begin
speaker=~speaker;
end


always @(posedge clk_4HZ)
begin
case({high,med,low})
'b000000000011: origin=8798;
'b000000000101: origin=10005;
'b000000000110: origin=10701;
'b000000000111: origin=11321;
'b000000010000: origin=6826;
'b000000100000: origin=12126;
'b000000110000: origin=12591;
'b000001010000: origin=13194;
'b000001100000: origin=13542;
'b000100000000: origin=13994;
'b000000000000: origin=16383;
endcas
e end
always @(posedge clk_4HZ)
begin
if(counter==63) counter=0;
else counter=counter+1;
case(counter)
0: {high,med,low}='b000000000011;
1: {high,med,low}='b000000000011;
2: {high,med,low}='b000000000011;
```

3: {high,med,low}='b000000000011;

4: {high,med,low}='b000000000101;

```
5: {high,med,low}='b000000000101;
6: {high,med,low}='b000000000101;
7: {high,med,low}='b000000000110;
8: {high,med,low}='b000000010000;
9: {high,med,low}='b000000010000;
10: {high,med,low}='b000000010000;
11: {high,med,low}='b000000100000;
12: {high,med,low}='b000000000110;
13: {high,med,low}='b000000010000;
14: {high,med,low}='b000000000101;
15: {high,med,low}='b000000000101;
16: {high,med,low}='b000001010000;
17: {high,med,low}='b000001010000;
18: {high,med,low}='b000001010000;
19: {high,med,low}='b000100000000;
20: {high,med,low}='b000001100000;
21: {high,med,low}='b000001010000;
22: {high,med,low}='b000000110000;
23: {high,med,low}='b000001010000;
24: {high,med,low}='b000000100000;
25: {high,med,low}='b000000100000;
26: {high,med,low}='b000000100000;
27: {high,med,low}='b000000100000;
28: {high,med,low}='b000000100000;
29: {high,med,low}='b000000100000;
30: {high,med,low}='b000000100000;
31: {high,med,low}='b000000100000;
32: {high,med,low}='b000000100000;
33: {high,med,low}='b000000100000;
34: {high,med,low}='b000000100000;
35: {high,med,low}='b000000110000;
36: {high,med,low}='b000000000111;
37: {high,med,low}='b000000000111;
38: {high,med,low}='b000000000110;
39: {high,med,low}='b000000000110;
40: {high,med,low}='b000000000101;
41: {high,med,low}='b000000000101;
42: {high,med,low}='b000000000101;
43: {high,med,low}='b000000000110;
44: {high,med,low}='b000000010000;
45: {high,med,low}='b000000010000;
46: {high,med,low}='b000000100000;
```

47: {high,med,low}='b000000100000;

48: {high,med,low}='b000000000011;

```
49: {high,med,low}='b000000000011;
50: {high,med,low}='b000000010000;
51: {high,med,low}='b000000010000;
52: {high,med,low}='b000000000110;
53: {high,med,low}='b000000000101;
54: {high,med,low}='b000000000110;
55: {high,med,low}='b000000010000;
56: {high,med,low}='b000000000101;
57: {high,med,low}='b000000000101;
58: {high,med,low}='b000000000101;
59: {high,med,low}='b000000000101;
60: {high,med,low}='b000000000101;
61: {high,med,low}='b000000000101;
62: {high,med,low}='b000000000101;
63: {high,med,low}='b000000000101;
endcase
end
endmodul
e
```

五、 实验结果

1、将程序写好，编译通过后分配好管脚（管脚分配如下所示），然后再次编译生成下载文件。

```
set_property PACKAGE_PIN P17 [get_ports clk]
set_property PACKAGE_PIN P15 [get_ports rst]
set_property PACKAGE_PIN B16 [get_ports speaker]// 外接5V的蜂鸣器

set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports rst]
set_property IOSTANDARD LVCMOS33 [get_ports speaker]
```

2、将程序下载到开发板上，上电后听蜂鸣器的发音（由于 50MHZ 在分频的过程中存在误差，设置的基频为 5MHZ 也不够好，同时在计算每个音符的分频比和预置数时采用四舍五入的方法，导致音乐最后的播放效果较差，有比较大的失真，望同学们在此基础上改进）。

实验十：字符型 LCM 驱动

一、 实验目的

1. 实现 FPGA 对LCM 的控制；
2. 了解 LCM 的工作时序和LCM 控制器相关指令。

二、 实验内容

1. 实现 FPGA 对LCM 的控制，使LCM 能够正常工作；
2. 实现在LCD1602的计时显示。

三、 实验要求

1. 实现LCD1602的动态显示。

四、 实验背景知识

1. 实验平台介绍

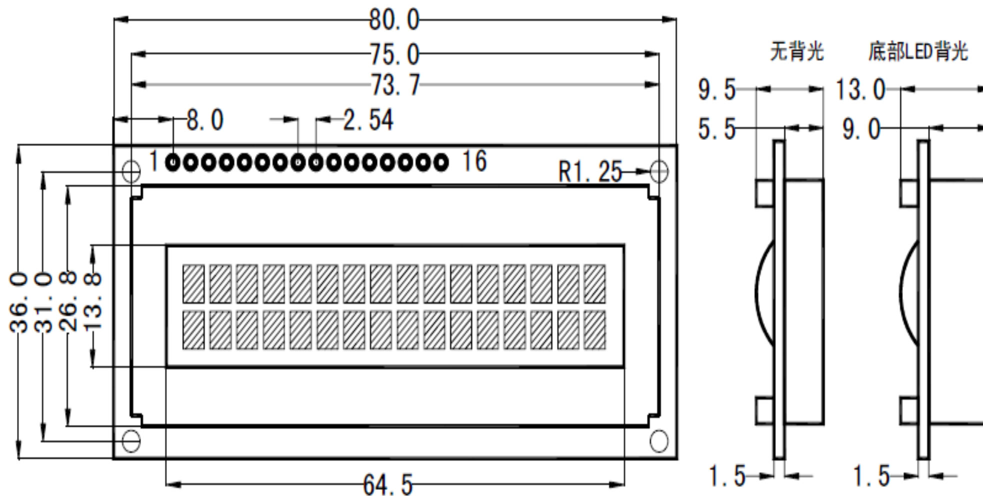
该实验的平台为：实验母板和LCM 模块。

LCM 是指液晶显示驱动模块，它由三部分组成，包括LCD 控制器、LCD 驱动器和LCD 显示装置。其中，LCD 控制器用于与芯片进行沟通，LCD 驱动器负责点亮LCD 显示装置。目前的LCM 模块一般将LCD 控制器、LCD 驱动器集成到一块IC 芯片上。

本实验通过编程，由母板提供时钟及其它必要的控制信号及数据信号，实现与LCD 控制器的沟通。

2. LCD1602简要介绍

LCD1602，根据名称可以知道，就是能显示2行，每行16个字符的液晶，只能显示字母，数字和符号等字符，不能显示汉字，图片。如下图：



市面上卖的的LCD1602操作基本上都是相同的，只是带不带背光之分。其控制芯片都是HD44780及其兼容芯片，所以控制接口都是一样，控制时序可以说是68并口时序。1LCD602控制线主要有4根：

（1）RS：数据/指令选择端，当RS = 0，写指令；当RS = 1，写数据。

（2）RW：读/写选择端，当RW = 0，写指令/数据;当RW = 1，读状态/数据。

（3）EN：使能端，下降沿使指令/数据生效。

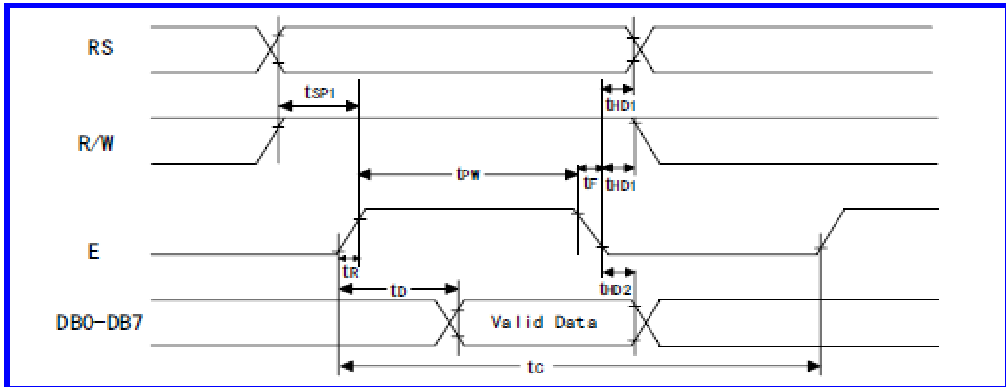
（4）Data[7:0]：8根并行数据口。

外引接口：

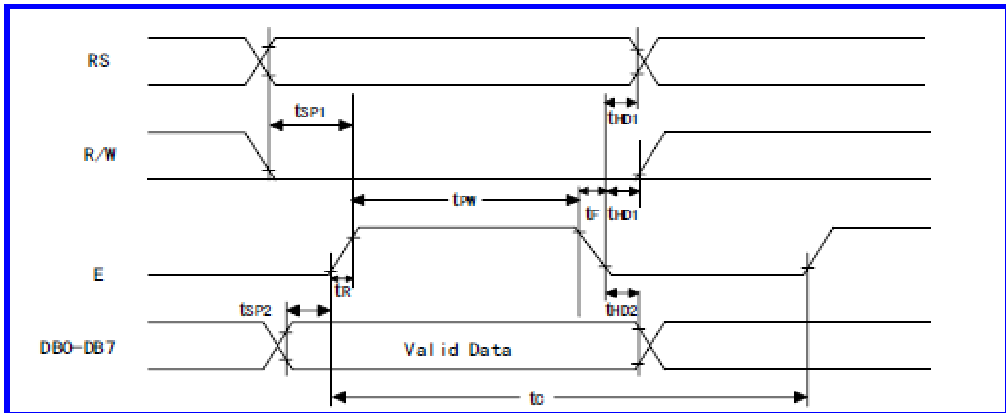
编号	符号	引脚说明	编号	符号	引脚说明
1	VSS	电源地	9	D2	Data I/O
2	VDD	电源正极	10	D3	Data I/O
3	VL	液晶显示偏压信号	11	D4	Data I/O
4	RS	数据/命令选择端（H/L）	12	D5	Data I/O
5	R/W	读/写选择端（H/L）	13	D6	Data I/O
6	E	使能信号	14	D7	Data I/O
7	D0	Data I/O	15	BLA	背光源正极
8	D1	Data I/O	16	BLK	背光源负极

操作时序图：

1. 读操作时序



2. 写操作时序



控制指令：

6 1602 液晶模块内部的控制器共有 11 条控制指令，如下表所示

序号	指令	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	清显示	0	0	0	0	0	0	0	0	0	1
2	光标返回	0	0	0	0	0	0	0	0	1	*
3	置输入模式	0	0	0	0	0	0	0	1	I/D	S
4	显示开/关控制	0	0	0	0	0	0	1	D	C	B
5	光标或字符移位	0	0	0	0	0	1	S/C	R/L	*	*
6	置功能	0	0	0	0	1	DL	N	F	*	*
7	置字符发生存贮器地址	0	0	0	1	字符发生存贮器地址					
8	置数据存贮器地址	0	0	1	显示数据存贮器地址						
9	读忙标志或地址	0	1	BF	计数器地址						
10	写数到 CGRAM 或 DDRAM）	1	0	要写的数据内容							
11	从 CGRAM 或 DDRAM 读数	1	1	读出的数据内容							

指令方面只讲解一下显示模式设置指令0x38,0x31的区别。其实模式设置指令就是上图中的指令6：0x38：设置8位格式,2行,5*7；0x31：设置8位格式,2行,5*7。为什么要介绍0x31呢，一般单片机驱动LCD1602都是0x38的？

由于一般的LCD1602 都是VDD = 5V驱动的， 而有些FPGA开发板上的

LCD1602接口是由3.3V供电的。也就是VDD = 3.3V，这样就会引起供电不足的问题，所以经过试验得到，当VDD = 3.3V时，显示模式设置指令写入0x38时，LCD1602显示很暗，看不到；进而改为0x31时，只显示1行，LCD1602就正常显示了。这个要引起注意,下面实验的代码就是只显示1行的。其他指令详解请查看数据手册。

3. FPGA 驱动LCD1602 思路

FPGA驱动LCD1602，其实就是通过同步状态机模拟单片机驱动LCD1602，由并行模拟单步执行，状态过程就是先初始化LCD1602，然后写地址，最后写入显示数据。

（1）首先，要明白LCD1602是慢速器件。如果直接用FPGA 外接的几十兆时钟直接驱动肯定是不行的，所以要对FPGA时钟进行分频驱动，或者计数延时使能驱动。

这里实验采用的计数延时使能驱动，代码中通过计数器定时得出lcd_clk_en信号线驱动。要注意的是不同厂家生产的LCD1602的时序延时都不同，但大多数都是纳秒级的，实验采用的是间隔500ns使能驱动，最好延时长一些比较可靠，这个可以自己尝试修正。

（2）LCD1602 的初始化过程需要明白。大家估计都用单片机驱动过LCD1602，这里FPGA驱动LCD1602的初始化过程也是一样的。主要是以下4条指令的配置：

- 显示模式设置Mode_Set：8’h38
- 显示开/关及光标设置Cursor_Set：8’h0c
- 显示地址设置Address_Set：8’h06
- 清屏设置Clear_Set：8'h01

这里需要注意是写指令,所以RS = 0，并且写完指令后，EN下降沿使能。

（3）初始化完成后，还需要写入地址，第一行初始地址：8’h80；第二行初始地址：8’h80 + 8”h40 = 8’hc0。这里RS = 0，并且写完地址后，EN下降沿使能。

（4）写入地址后，就可以显示字符啦。但需要注意LCD1602写入设置地址

指令8’h06后，地址是随每写入一个数据后，默认自加一的。这个一定要明白， 不然作动态显示时，就会出现问题。一定要把握实验的数据是要显示在哪个位置，而LCD1602写入地址是会默认地址指针加一的。这里RS = 1，并且写完数据后，EN下降沿使能。

（5）由于实验要求动态显示，所以数据要刷新。这里由于我们采用的是同步状态机模拟LCD1602的控制时序，所以在显示完最后的数据后，状态要跳回写入地址状态，以便进行动态刷新。这个很重要，不只是保证刷新，更是保证地址没有偏移。

五、 实验程序实现

1. 主程序：

```
module lcd1602_driver(input clk,          //100M
                      input rst_n,
                         output lcd_p,       //Backlight Source +
                         output lcd_n,       //Backlight Source -
                         output reg lcd_rs,       //0:write order; 1:write data
                         output lcd_rw,              //0:write data;   1:read
                         data output reg lcd_en,                        //negedge
                         output reg [7:0] lcd_data);

//------------------lcd1602 order---------------------------
parameter       Mode_Set        =    8'h31,
                 Cursor_Set    =    8'h0c,
                 Address_Set   =   8'h06,
                 Clear_Set     =    8'h01;

/****************************LCD1602 Display Data****************************/
wire [7:0] data0,data1; //counter data
wire [7:0] addr;      //write address
//------------------------------1s counter--------------------------------
reg [31:0] cnt1;
reg [7:0] data_r0,data_r1;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
            cnt1 <= 1'b0;
            data_r0 <= 1'b0;
```

```
                        data_r1 <= 1'b0;
                end
        else if(cnt1==32'd50000000)
                begin
                        if(data_r0==8'd9)
                                begin
                                        data_r0 <= 1'b0;
                                        if(data_r1==8'd9)
                                                data_r1 <= 1'b0;
                                        els
                                        e       data_r1 <= data_r1 + 1'b1;
                                en
                        else    d
                                data_r0 <= data_r0 + 1'b1;
                        cnt1 <= 1'b0;
                en
        els     d
        e
                cnt1 <= cnt1 + 1'b1;
end

assign data0 = 8'h30 + data_r0 ;
assign data1 = 8'h30 + data_r1 ;

//------------------address-----------------
assign addr = 8'h80;

/****************************LCD1602 Driver****************************/
//----------------------lcd1602 clk_en--------------------
reg [31:0] cnt;
reg lcd_clk_en;
always @(posedge clk or negedge rst_n)
begin
        if(!rst_n)
                begin
                        cnt <= 1'b0;
                        lcd_clk_en <= 1'b0;
                end
        else if(cnt == 32'h24999)
                                        //500u
                s begin
                        lcd_clk_en <= 1'b1;
```

```
            cnt <= 1'b0;
        end
    else
```

```
            begin
                cnt <= cnt + 1'b1;
                lcd_clk_en <= 1'b0;
            end
end

//----------------------lcd1602 display state-----------------------------------------
reg [4:0] state;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
            state <= 1'b0;
            lcd_rs <= 1'b0;
            lcd_en <= 1'b0;
            lcd_data <= 1'b0;
        end
    else if(lcd_clk_en)
        begin
            case(state)
                //------------------init_state--------------------
                5'd0: begin
                        lcd_rs <= 1'b0;
                        lcd_en <= 1'b1;
                        lcd_data <= Mode_Set;
                        state <= state + 1'd1;
                        end
                5'd1: begin
                        lcd_en <=  1'b0;
                        state <= state + 1'd1;
                        end
                5'd2: begin
                        lcd_rs <= 1'b0;
                        lcd_en <= 1'b1;
                        lcd_data <= Cursor_Set;
                        state <= state + 1'd1;
                        end
                5'd3: begin
                        lcd_en <=  1'b0;
                        state <= state + 1'd1;
                        end
                5'd4: begin
```

```
lcd_rs <= 1'b0;
lcd_en <= 1'b1;
```

```
lcd_data <= Address_Set;
state <= state + 1'd1;
en
d 5'd5: begin
lcd_en <=  1'b0;
state <= state + 1'd1;
end
5'd6: begin
lcd_rs <= 1'b0;
lcd_en <= 1'b1;
lcd_data <= Clear_Set;
state <= state + 1'd1;
end
5'd7: begin
lcd_en <=  1'b0;
state <= state + 1'd1;
end

//-------------------work state-------------------
5'd8: begin
lcd_rs <= 1'b0;
lcd_en <= 1'b1;
lcd_data <= addr;      //write
addr state <= state + 1'd1;
en
d 5'd9: begin
lcd_en <=  1'b0;
state <= state + 1'd1;
end
5'd10: begin
lcd_rs <= 1'b1;
lcd_en <= 1'b1;
lcd_data <= "C";      //write
data state <= state + 1'd1;
 en
d 5'd11: begin
lcd_en <=  1'b0;
state <= state + 1'd1;
end
5'd12: begin
lcd_rs <= 1'b1;
lcd_en <= 1'b1;
```

```
lcd_data <= "n";      //write
data state <= state + 1'd1;
```

```
        en
d 5'd13: begin
            lcd_en <=  1'b0;
            state <= state + 1'd1;
            end
5'd14: begin
            lcd_rs <= 1'b1;
            lcd_en <= 1'b1;
            lcd_data <= "t";      //write
            data state <= state + 1'd1;
             en
d 5'd15: begin
            lcd_en <=  1'b0;
            state <= state + 1'd1;
            end
5'd16: begin
            lcd_rs <= 1'b1;
            lcd_en <= 1'b1;
            lcd_data <= ":";      //write
            data state <= state + 1'd1;
             en
d 5'd17: begin
            lcd_en <=  1'b0;
            state <= state + 1'd1;
            end
5'd18: begin
            lcd_rs <= 1'b1;
            lcd_en <= 1'b1;
            lcd_data <= data1;     //write data: tens
            digit state <= state + 1'd1;
             en
d 5'd19: begin
            lcd_en <=  1'b0;
            state <= state + 1'd1;
            end
5'd20: begin
            lcd_rs <= 1'b1;
            lcd_en <= 1'b1;
            lcd_data <= data0;     //write data: single
            digit state <= state + 1'd1;
            en
d 5'd21:
```

```
begin
        lcd_en <= 1'b0;
        state <= 5'd8;
```

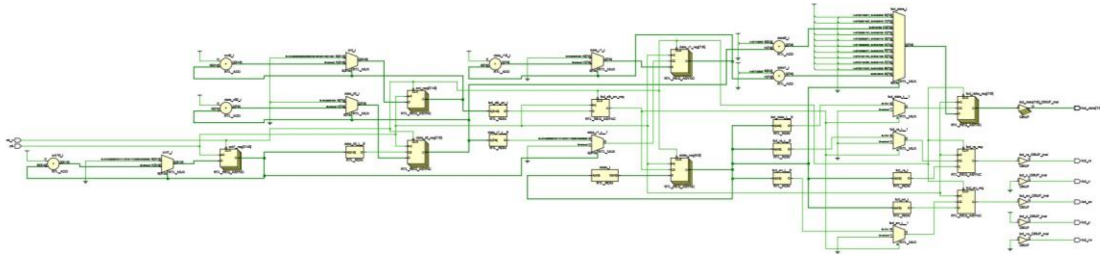
```
                        end
                    default: state <= 5'bxxxxx;
                endcase
            en
end         d

assign lcd_rw = 1'b0;     //only write

//-----------------backlight driver---------------
assign lcd_n = 1'b0;
assign lcd_p = 1'b1;

endmodule
```

2. 总体设计的RTL视图



六、 实验结果

1、将程序写好，编译通过后分配好管脚（管脚分配如下所示），然后再次编译生成下载文件。

```
set_property PACKAGE_PIN P17 [get_ports clk]
set_property PACKAGE_PIN B16 [get_ports {lcd_data[7]}]
set_property PACKAGE_PIN B17 [get_ports {lcd_data[6]}]
set_property PACKAGE_PIN A15 [get_ports {lcd_data[5]}]
set_property PACKAGE_PIN A16 [get_ports {lcd_data[4]}]
set_property PACKAGE_PIN A13 [get_ports {lcd_data[3]}]
set_property PACKAGE_PIN A14 [get_ports {lcd_data[2]}]
set_property PACKAGE_PIN B18 [get_ports {lcd_data[1]}]
set_property PACKAGE_PIN A18 [get_ports {lcd_data[0]}]
set_property PACKAGE_PIN F13 [get_ports lcd_en] set_property
PACKAGE_PIN F14 [get_ports lcd_p] set_property PACKAGE_PIN
B13 [get_ports lcd_n] set_property PACKAGE_PIN B14 [get_ports
lcd_rw] set_property PACKAGE_PIN D14 [get_ports lcd_rs]
set_property PACKAGE_PIN P15 [get_ports rst_n]

set_property IOSTANDARD LVCMOS33 [get_ports clk] set_property
IOSTANDARD LVCMOS33 [get_ports {lcd_data[7]}] set_property
IOSTANDARD LVCMOS33 [get_ports {lcd_data[6]}] set_property
IOSTANDARD LVCMOS33 [get_ports {lcd_data[5]}] set_property
IOSTANDARD LVCMOS33 [get_ports {lcd_data[4]}] set_property
IOSTANDARD LVCMOS33 [get_ports {lcd_data[3]}] set_property
IOSTANDARD LVCMOS33 [get_ports {lcd_data[2]}] set_property
IOSTANDARD LVCMOS33 [get_ports {lcd_data[1]}] set_property
IOSTANDARD LVCMOS33 [get_ports {lcd_data[0]}] set_property
IOSTANDARD LVCMOS33 [get_ports lcd_en] set_property IOSTANDARD
LVCMOS33 [get_ports lcd_p] set_property IOSTANDARD LVCMOS33
[get_ports lcd_n] set_property IOSTANDARD LVCMOS33 [get_ports
lcd_rw] set_property IOSTANDARD LVCMOS33 [get_ports lcd_rs]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
```

实验十一：VGA

一、 实验目的

1. 了解VGA 显示的基本原理和实现方法；
2. 学会使用verilog 语言编制VGA 显示的接口程序。

二、 实验内容

1. 掌握VGA 显示通信的基本原理和实现方法；
2. 理解编制的VGA 接口程序的实现方法和使用。

三、 实验要求

1. 掌握已编好的VGA 程序，根据提供的思路自主编制VGA 显示的例程。

四、 实验背景知识

1. VGA 的概念

VGA 的英文全称是Video Graphic Array，即显示绘图阵列。VGA 支持在640X480 的较高分辨率下同时显示16种色彩或256 种灰度，同时在320X240 分辨率下可以同时显示256 种颜色。肉眼对颜色的敏感远大于分辨率，所以即使分辨率较低图像依然生动鲜明。VGA 由于良好的性能迅速开始流行，厂商们纷纷在VGA 基础上加以扩充，如将显存提高至1M 并使其支持更高分辨率如800X600 或1024X768，这些扩充的模式就称之为VESA（Video Electronics Standards Association，视频电子标准协会）的Super VGA 模式，简称SVGA，现在的显卡和显示器都支持SVGA 模式。不管是VGA 还是SVGA，使用的连线都是15针的梯形插头，传输模拟信号。

2. VGA 的接口信号

目前大多数计算机与外部显示设备之间都是通过模拟VGA 接口连接，计算机内部以数字方式生成的显示图像信息，被显卡中的数字/模拟转换器转变为R、

G、B三原色信号和行、场同步信号，信号通过电缆传输到显示设备中。本例中，VGA 接口是标准的15 针接口，有五个接口信号，如下表。

HS	行同步信号
VS	场同步信号
R	红色信号
G	绿色信号
B	蓝色信号

表11.1、VGA 接口信号定义

3. 行同步和场同步

为了实现发送端与接受端图像各点一一正确对应，发送端与接收端的扫描必须同步。同步脉冲是周期稳定，边沿陡峭的脉冲。按我国电视标准，行同步脉冲的频率等于行频为15.625KHZ，行周期为64us。在电视技术中常以64us 作为时间单位，并以H 表示，即1H＝64us。场同步脉冲频率等于场频为50HZ，场周期为20ms，即312.5H。行同步脉冲宽度为4.7us 左右，场同步脉冲宽度为2.5～3H。

五、 实验方案

如上图，主程序有4 个输入信号和5 个输出信号。

输入信号分别为：clk、switch。clk 是时钟信号，switch 是选择模式信号，分别对应横彩条、竖彩条及两种棋盘格。输出信号在上面已经介绍过了，不再赘述。

程序源码：

```
module vga( clock, switch, disp_RGB, hsync, vsync );
input clock; //系统输入时钟 100MHz
input [1:0]switch;
output [2:0]disp_RGB; //VGA 数据输出
output hsync; //VGA 行同步信号output
vsync; //VGA 场同步信号
reg [9:0] hcount; //VGA 行扫描计数器
reg [9:0] vcount; //VGA 场扫描计数器
reg [2:0] data;
reg [2:0] h_dat;
reg [2:0] v_dat;
reg flag;
reg [1:0]cnt;
wire hcount_ov;
wire vcount_ov;
wire dat_act;
```

```
wire  hsync;
wire  vsync;
reg vga_clk;
//VGA 行、场扫描时序参数表
parameter hsync_end = 10'd95,
hdat_begin = 10'd143,
hdat_end = 10'd783,
hpixel_end = 10'd799,
vsync_end = 10'd1,
vdat_begin = 10'd34,
vdat_end = 10'd514,
vline_end = 10'd524;
always @(posedge clock)
begin
if(cnt==3)
    cnt <= 0;
  else
    cnt <= cnt + 1;
end
always @(posedge clock)
begin
  if(cnt < 2)
    vga_clk <= 1;
  else
    vga_clk <= 0;
end
//***********************VGA 驱动部分*****************************//行扫描
always @(posedge vga_clk)
begin
if (hcount_ov)
hcount <= 10'd0;
else
hcount <= hcount + 10'd1;
end
assign hcount_ov = (hcount == hpixel_end);
//场扫描

always @(posedge vga_clk)
begin
if (hcount_ov)
begin
if (vcount_ov)
vcount <= 10'd0;
```

else

```
vcount <= vcount + 10'd1;
end
end
assign vcount_ov = (vcount == vline_end);
//数据、同步信号输
assign dat_act = ((hcount >= hdat_begin) && (hcount < hdat_end))
&& ((vcount >= vdat_begin) && (vcount < vdat_end));
assign hsync = (hcount > hsync_end);
assign vsync = (vcount > vsync_end); assign
disp_RGB = (dat_act) ? data : 3'h00;
//************************显示数据处理部分*****************************//
always @(posedge vga_clk)
  begin
    case(switch[1:0])
      2'd0: data <= h_dat; //选择横彩条
      2'd1: data <= v_dat; //选择竖彩条
      2'd2: data <= (v_dat ^ h_dat); //产生棋盘格
      2'd3: data <= (v_dat ~^ h_dat); //产生棋盘格
    endcase
  end
always @(posedge vga_clk) //产生竖彩条
  begin
    if(hcount < 223)
      v_dat <= 3'h7; //白
    else if(hcount < 303)
      v_dat <= 3'h6; //黄
    else if(hcount < 383)
      v_dat <= 3'h5; //青
    else if(hcount < 463)
      v_dat <= 3'h4; //绿
    else if(hcount < 543)
      v_dat <= 3'h3; //紫
    else if(hcount < 623)
      v_dat <= 3'h2; //红
    else if(hcount < 703)
      v_dat <= 3'h1; //蓝
    else
      v_dat <= 3'h0; //黑
  end
always @(posedge vga_clk) //产生横彩条
  begin
    if(vcount < 94)
      h_dat <= 3'h7; //白
    else if(vcount < 154)
      h_dat <= 3'h6; //黄
    else if(vcount < 214)
      h_dat <= 3'h5; //青
    else if(vcount < 274)
      h_dat <= 3'h4; //绿
    else if(vcount < 334)
      h_dat <= 3'h3; //紫
```

```
      else if(vcount < 394)
        h_dat <= 3'h2; //红
      else if(vcount < 454)
        h_dat <= 3'h1; //蓝
      else
        h_dat <= 3'h0; //黑
   end
endmodu
```

五、 实验结果

1、将程序写好，编译通过后分配好管脚（管脚分配如下所示），然后再次编译生成下载文件。

```
set_property PACKAGE_PIN P17 [get_ports CLK]
set_property PACKAGE_PIN N4 [get_ports {switch[1]}]
set_property PACKAGE_PIN R1 [get_ports {switch[0]}]
set_property PACKAGE_PIN C7 [get_ports {disp_RGB[2]}]
set_property PACKAGE_PIN B6 [get_ports {disp_RGB[1]}]
set_property PACKAGE_PIN F5 [get_ports {disp_RGB[0]}]
set_property PACKAGE_PIN D7 [get_ports hsync] set_property
PACKAGE_PIN C4 [get_ports vsync]
set_property IOSTANDARD LVCMOS33 [get_ports CLK] set_property
IOSTANDARD LVCMOS33 [get_ports {switch[1]}] set_property
IOSTANDARD LVCMOS33 [get_ports {switch[0]}] set_property
IOSTANDARD LVCMOS33 [get_ports {disp_RGB[2]}] set_property
IOSTANDARD LVCMOS33 [get_ports {disp_RGB[1]}] set_property
IOSTANDARD LVCMOS33 [get_ports {disp_RGB[0]}] set_property
IOSTANDARD LVCMOS33 [get_ports hsync] set_property IOSTANDARD
LVCMOS33 [get_ports vsync]
```

2.将程序下载到实验板上，接好VGA 线，观察显示内容。

实验十二：PS/2 接口控制

一、 实验目的

1. 了解PS/2键盘的工作原理；
2. 学会使用verilog 语言编制PS/2的接口控制程序。

二、 实验内容

1. 通过Verilog编程实现在EGO1开发板上对键盘接口控制，通过键盘输入实现对LED流水灯的控制(按下W时，LED向右奔跑，按下X时，LED向左奔跑，按下ctrl时左右交换);
2. 理解编制的PS/2接口控制程序的实现方法和使用。

三、 实验要求

1. 掌握已编好的PS/2程序，根据提供的思路自主编制PS/2输入在LCD或者VAG上显示的例程。

四、 实验背景知识

1. PS/2协议和键盘扫描码

（1） PS/2键盘履行一种双向同步串行协议。换句话说每次数据线上发送一位数据，并且每在时钟线上发一个脉冲就被读入，键盘可以发送数据到主机，而主机也可以发送数据到设备。但主机总是在总线上有优先权，它可以在任何时候抑制来自于键盘的通信，只要把时钟拉低即可。

本实例要编写一个能实现PS/2端口功能的Verilog程序。首先我们要了解PS/2端口的结构与管脚功能定义，如表12.1所示。

表 12.11 PS/2 端口结构及管脚定义

端口结构		管脚定义	
		1	数据
		2	未实现，保留
		3	电源地
		4	电源，+5V
		5	时钟
插头	插座	6	未实现，保留

可以看到，PS/2里面只有一个数据口，若要分辨很多按键就需要一个高效率的分辨方法。键盘的处理器花费很多的时间来扫描或监视按键矩阵。如果它发现有键被按下，释放或按住键盘，将发送扫描码的信息包到计算机。PS/2的时序如图3：

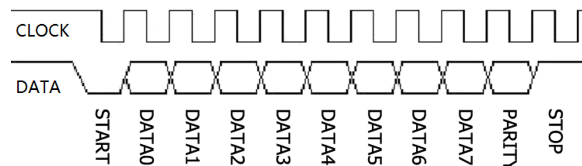


图3 PS/2时序图

当PS/2设备需要传输数据时，会在PS/2时钟（CLOCK）的第一个下降沿拉低PS/2数据线（DATA），表示一帧PS/2数据传输的开始。接着是8个数据位（都是在时钟的下降沿有效），然后是奇偶校验位，最后是停止位（高电平）。主机在接收完一帧数据后，需要将接收到的串行数据转换成并行数据，这就是PS/2键盘的扫描码。

（2） PS/2键盘扫描码有两种不同的类型：通码(make code)和断码(break code)。当按键被按下或持续按住时，键盘会发送该键的通码；而当一个键被释放时，键盘会发送该键的断码。根据键盘按键扫描码的不同,可将按键分为3类：

- 第1类按键：通码为一个字节，断码为0xF0+通码的形式。如A键,其通码为0x1C；断码为0xF0 0x1C。
- 第2类按键：通码为两字节0xE0+0xXX形式，断码为0xE0+0xF0+0xXX形式。如Right Ctrl键，其通码为0xE0 0x14；断码为0xE0 0xF0 0x14。

- 第3类特殊按键：有两个，Print Screen键，其通码为0xE0 0x12 0xE0 0x7C；断码为0xE0 0xF0 0x7C 0xE0 0xF0 0x12。Pause键，其通码为0xE1 0x14 0x77 0xE1 0xF0 0xl4 0xF0 0x77；断码为空。

每个键一整套的通断码组成了扫描码集，图1中包含了键盘上面大部分按键的扫描码。

图 1 键盘扫描码

（3）信号从键盘输入通过PS/2端口的数据线输入的过程。首先键盘要检测数据线和时钟线是否都为高，只有它们都处在高的状态才可以写数据。从键盘发送到主机的数据在时钟信号的下降沿（当时钟从高变到低）的时候被读取。

键盘主要使用一种每帧包含11位的串行协议：第一位是起始位，始终为“0”；接下来是8位数据位，排列顺序是由低到高；再后面是奇偶校验位；最后是结束位，始终为“1”。

2. PS/2键盘解码模块

（1）PS/2键盘事件响应流程如图6：

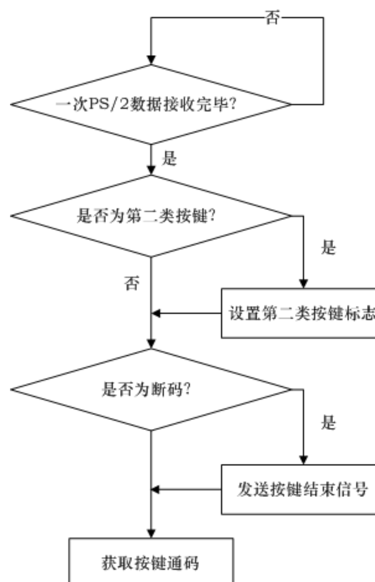


图 6 PS/2 键盘事件响应流程图

当有按键按下时，PS/2解码模块先将按键通码保存，然后一直等待按键松开，直到接收按键断码，此时一次按键事件结束。接着，按照第二套按键扫描码的规定，将按键通码转换为ASCII码，并产生一个按键事件信号，等待其他模块处理。

五、 实验方案及实现：

程序的设计共分3个模块：

- PS/2 接口控制模块

```
module
ps2 (
        CLK,
        RSTn,
        iCe,
        pclk,
        pdata,
        scan_code,
        rdy
);

input CLK;      //100M 时钟信号
input RSTn;     //复位信号
input  pclk;  //PS2  接口时钟信号
input  pdata;  //PS2  接口数据信号
input iCe;
wire wGatedClk;
reg rCe;
output reg[15:0] scan_code;  // 1byte 键值，只做简单的按键扫描
wire   oPsState;               //键盘当前状态，ps2_state=1 表示有键被按下

//-----------------------------------------
reg rPsClk0,rPsClk1,rPsClk2;  //ps2k_clk 状态寄存器

//wire pos_ps2k_clk; // ps2k_clk 上升沿标志位
wire neg_ps2k_clk; // ps2k_clk 下降沿标志位

always @ (posedge CLK)
begin
        rCe <= iCe;
```

```
end
assign wGatedClk = rCe & CLK;

always @ (posedge wGatedClk or negedge RSTn) begin
        if(!RSTn) begin
                        rPsClk0  <=  1'b0;
                        rPsClk1  <=  1'b0;
                        rPsClk2 <= 1'b0;
                end
        else begin                          //锁存状态，进行滤
                        波rPsClk0 <= pclk;
                         rPsClk1 <= rPsClk0;
                         rPsClk2 <= rPsClk1;
                end
end

assign neg_ps2k_clk = ~rPsClk1 & rPsClk2;     //下降沿

//-----------------------------------------
reg[7:0] ps2_byte_r;      //PC 接收来自 PS2 的一个字节数据存储器
reg[7:0] temp_data;      //当前接收数据寄存
器reg[3:0] num;          //计数寄存器

output  rdy;
reg rdy;
always @ (posedge wGatedClk or negedge RSTn) begin
        if(!RSTn) begin
                        num <= 4'd0;
                        temp_data <= 8'd0;
                end
        else if(neg_ps2k_clk) begin            //检测到ps2k_clk 的下降沿
                        case (num)
                                4'd0:           num <= num+1'b1;
                                4'd1:           begin
                                                    num <= num+1'b1;
                                                    temp_data[0] <= pdata;
                                                    //bit0 end
                                4'd2:           begin
                                                    num <= num+1'b1;
                                                    temp_data[1] <= pdata;
                                                    //bit1 end
                                4'd3:           begin
                                                    num <= num+1'b1;
```

```
temp_data[2] <= pdata;
//bit2
```

```
                                        end
                        4'd4:           begin
                                        num <= num+1'b1;
                                        temp_data[3] <= pdata;
                                        //bit3 end
                        4'd5:           begin
                                        num <= num+1'b1;
                                        temp_data[4] <= pdata;
                                        //bit4 end
                        4'd6:           begin
                                        num <= num+1'b1;
                                        temp_data[5] <= pdata;
                                        //bit5 end
                        4'd7:           begin
                                        num <= num+1'b1;
                                        temp_data[6] <= pdata;
                                        //bit6 end
                        4'd8:           begin
                                        num <= num+1'b1;
                                        temp_data[7] <= pdata;
                                                                //bit
                                        7 end
                        4'd9:           begin
                                        num  <=  num+1'b1;    //奇偶校验
位，不做处理
                                        end
                        4'd10:
                        begin           num <= 4'd0;    // num 清零
                                        end

                        default: ;
            end         endcase
end

always @ (posedge wGatedClk or negedge RSTn)
        begin
                if(!RSTn)
                        begin
                                rdy <= 1'b0;
                        end
                                                                        end
```

else if(num ==
4'd10)

r

d

y

<

=

1

'

b

1

;

e

l

s

e

r

d

y

<

=

1

'

b

0

;

```
reg key_f0;       //松键标志位，置1表示接收到数据8'hf0，再接收到下一个数据后清零
reg ps2_state_r; //键盘当前状态，ps2_state_r=1表示有键被按下
reg[20:0] k;
always @ (posedge wGatedClk or negedge  RSTn) begin     //接收数据的相应处理，这里只对1byte的键值进行处理
            if(!RSTn) begin
                        key_f0 <= 1'b0;
                        ps2_state_r <= 1'b0;
                  end
            else if(num==4'd9) begin     //刚传送完一个字节数据
                        if(temp_data == 8'hf0) key_f0 <= 1'b1;
                        else begin
                                    if(!key_f0) begin    //说明有键按
                                          下if(k>=4000)
                                          begin
                                          ps2_state_r <= 1'b1;
                                          ps2_byte_r <= temp_data;
                                          end   //锁存当前键值
                                          else
                                          k<=k+1'b1;
                                          end
                                    else if(key_f0) begin
                                                ps2_state_r <= 1'b0;
                                                key_f0 <= 1'b0;
                                                k<=0;
                                          end
                              end
                  end
end
reg[15:0] ps2_asci;    //接收数据的相应ASCII码
always @ (ps2_byte_r)
begin
 /*case (ps2_byte_r)   //键值转换为ASCII码，这里做的比较简单，只处理字母
                  8'h5a: ps2_asci <= 16'hfffe;
                  8'h71: ps2_asci <= 16'hfffd;
                  8'h70: ps2_asci <= 16'hfffb;
                  8'h66: ps2_asci <= 16'hfff7;
                  8'h79: ps2_asci <= 16'hffef;
                  8'h7a: ps2_asci <= 16'hffdf;
                  8'h72: ps2_asci <= 16'hffbf;
                  8'h69: ps2_asci <= 16'hff7f;
```

```
8'h74: ps2_asci <= 16'hfeff;
8'h73: ps2_asci <= 16'hfdff;
```

```
8'h6b: ps2_asci <= 16'hfbff;
8'h7d: ps2_asci <= 16'hf7ff;
8'h75: ps2_asci <= 16'hefff;
8'h6c: ps2_asci <= 16'hdfff;
8'h7b: ps2_asci <= 16'hbfff;
8'h7c: ps2_asci <= 16'h7fff;
default:ps2_asci<=16'hffff ;
endcase*/
ps2_asci <= {8'hff,ps2_byte_r[7:0]};
end
assign oPsState = ps2_state_r;
//assign scan_code = ps2_asci;
always@(posedge  wGatedClk)
begin
    if(oPsState==1)
    scan_code <= ps2_asci;
    else
    scan_code <=16'hffff;
end
endmodule
```

- 命令测试模块

```
module
cmd_control_module (
    CLK, RSTn,
        PS2_Done_Sig, PS2_Data,
        Data_Out
);

    input CLK;
        input RSTn;
        input   PS2_Done_Sig;
        input   [7:0]PS2_Data;
        output  [3:0]Data_Out;
        /****************************
        / reg [3:0]rData;

        always @ ( posedge CLK or negedge RSTn )
            if( !RSTn )
                begin
                    rData <= 4'b0001;
                end
        else if( PS2_Done_Sig )
```

```
                    case( PS2_Data )
                         8'h1d:
                        rData <= { rData[2:0], rData[3] };

                        8'h22:
                        rData <= { rData[0], rData[3:1] };

                        8'h14:
                        rData <= { rData[0], rData[1], rData[2], rData[3] };

                        endcase

        /***********************************
        / assign Data_Out = rData;
        /***********************************
    / endmodule
```

- 顶层模块

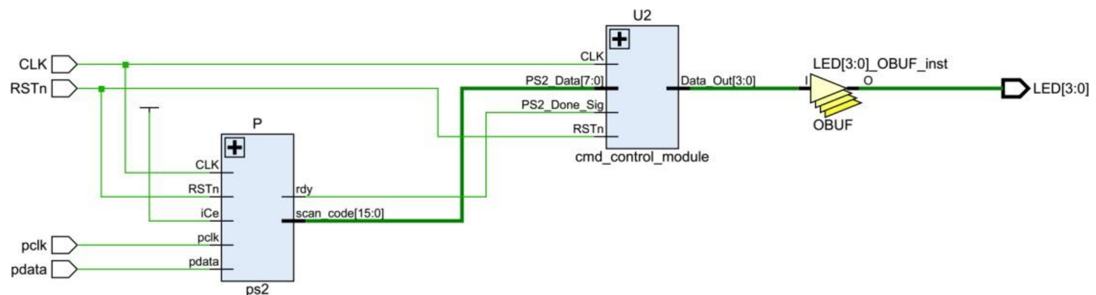
```
module top(
        pclk,
        pdata,
        CLK,
        RSTn,
        LED
);
          input CLK, RSTn;
        input pclk;                   // PS_2 clock input
        input pdata;                  //    PS_2    pdata
        input  wire[7:0]  scan_code;      //  Scan_code
        output wire rdy;             //     pdata      ready
        output

        ps2 P(
                .CLK(CLK),
                .RSTn(RSTn),
                .iCe(1'b1),
                .pclk(pclk),
                .pdata(pdata),
                .scan_code(scan_code),
                .rdy(rdy)
                );
        output [3:0] LED;
        cmd_control_module
```

```
U2 (
    .CLK( CLK ),
```

```
                    .RSTn( RSTn ),
                    .PS2_Done_Sig( rdy ), // input - from U1
                    .PS2_Data( scan_code ),     // input - from U1
                    .Data_Out( LED )             // output - to top
        );
endmodule
```

（3）总体设计的RTL视图



六、 实验结果

1、将程序写好，编译通过后分配好管脚（管脚分配如下所示），然后再次编译生成下载文件。

```
set_property PACKAGE_PIN P17 [get_ports CLK]
set_property PACKAGE_PIN K5 [get_ports pclk]
set_property PACKAGE_PIN L4 [get_ports pdata]
set_property PACKAGE_PIN P15 [get_ports RSTn]
set_property PACKAGE_PIN H4 [get_ports {LED[3]}]
set_property PACKAGE_PIN J3 [get_ports {LED[2]}]
set_property PACKAGE_PIN J2 [get_ports {LED[1]}]
set_property PACKAGE_PIN K2 [get_ports {LED[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports pclk]
set_property IOSTANDARD LVCMOS33 [get_ports pdata]
set_property IOSTANDARD LVCMOS33 [get_ports RSTn]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
```

2.将程序下载到实验板上，自行通过扩展接口接上PS/2接口线，按键W、X或Ctrl看LED灯的亮灭情况。（由于按键存在抖动，会存在按一下LED灯跳2下的情况）

实验十三：IP 核调用

一、 实验目的

Vivado 中有很多 IP 核可以直接使用，例如数学运算（乘法器、除法器、浮点运算器等）、信号处理（FFT、DFT、DDS 等）。IP 核类似编程中的函数库（例如 C 语言中的 printf() 函数，可以直接调用，非常方便，大大加快了开发速度。

二、 实验内容

1. 实现系统 IP 核调用。

三、 实验要求

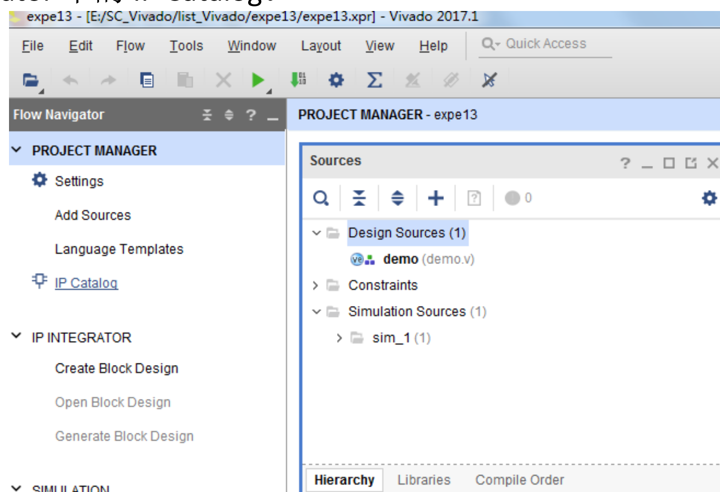
1. 在接下来的实验中熟练掌握系统 IP 核调用。

四、 实验步骤

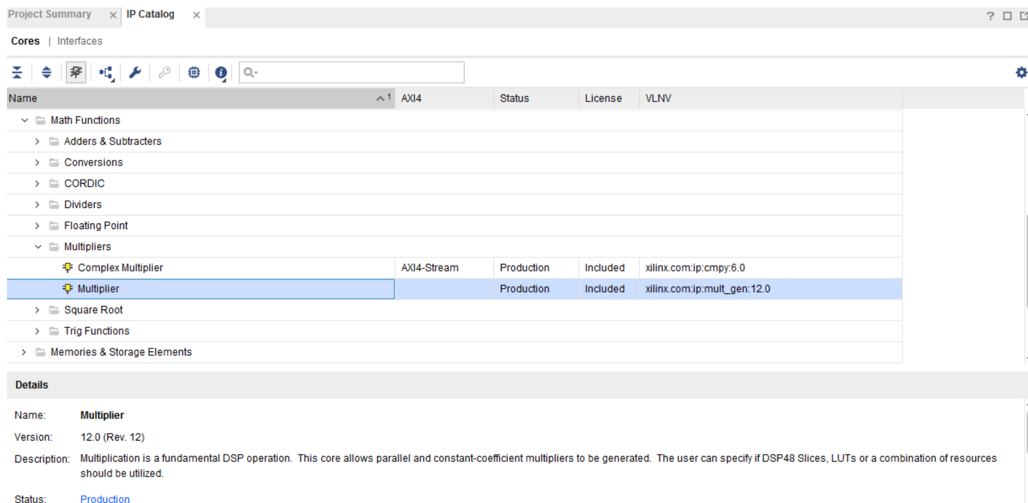
创建IP核

这里简单举一个乘法器的 IP 核使用实例，使用 Verilog 调用。首先新建工程，新建 demo.v 顶层模块。

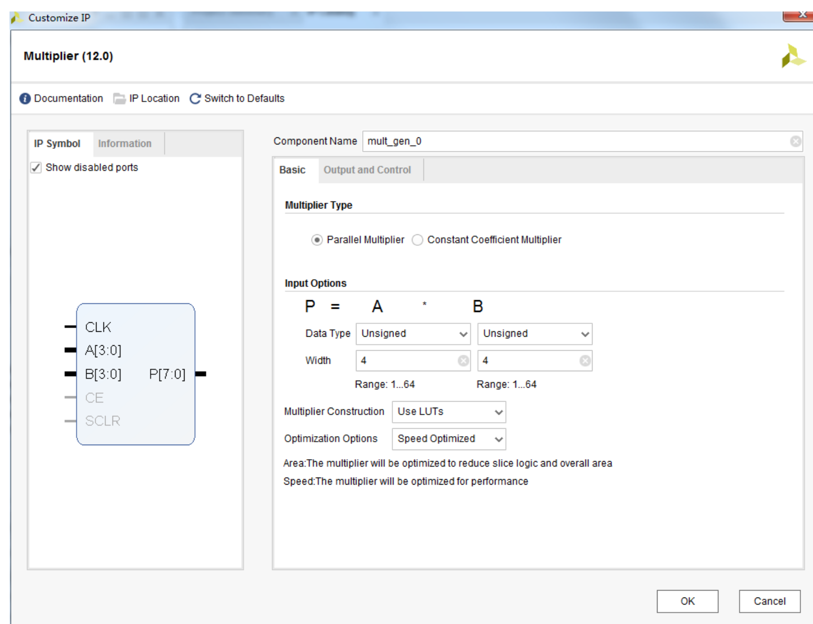
点击 Flow Navigator 中的 IP Catalog。



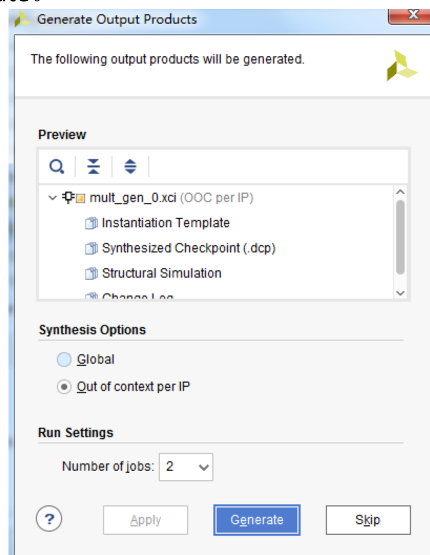
选择 Math Functions 下的Multiplier，即乘法器，并双击。



将弹出 IP 核的参数设置对话框。点击左上角的 Documentation，可以打开这个 IP 核的使用手册查阅。这里直接设置输入信号 A 和B 均为4 位无符号型数据，其他均为默认值，点击OK。

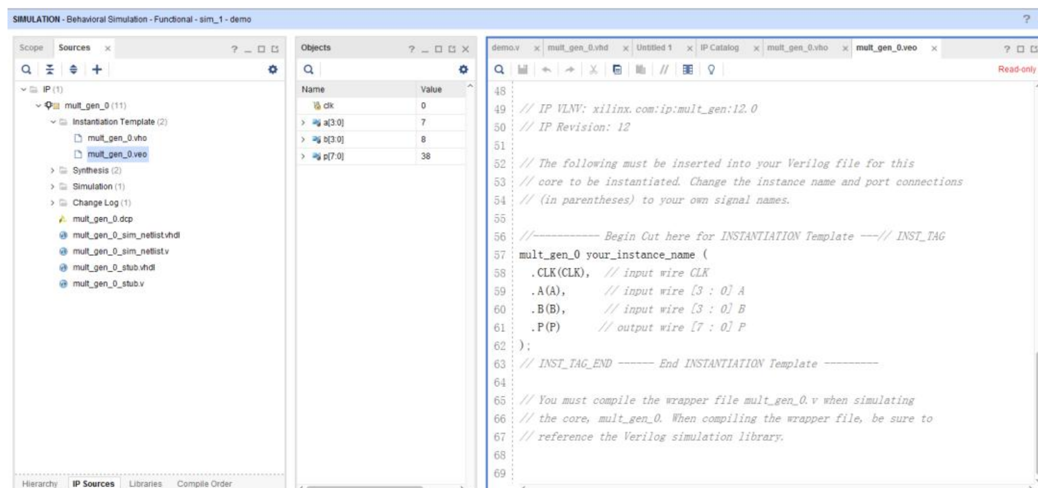


稍后弹出的窗口，点击Generate。



调用IP核

选择IP Sources，展开并选择mult_gen_0 - Instantiation Template - mult_gen_0.veo，可以打开实例化模板文件。如图，这段代码就是使用Verilog调用这个IP核的示例代码。



行为仿真验证

将示例代码复制到demo.v文件中，并进行修改，最终如下。代码中声明了无符号型的4位变量a和b，分别赋初值7、8，作为乘数使用；无符号型的8位变量p，用于保存计算结果。clk为Testbench编写的周期20ns的时钟信号；mult_gen_0 mul(...)语句实例化了mult_gen_0类型的模块对象mul，并将clk、a、b、p作为参数传入。

```
module demo(
);

reg clk = 0;
always #10 clk = ~clk;

wire [3:0] a = 7;
wire [3:0] b = 8;
wire [7:0] p;

mult_gen_0 mul (
  .CLK(clk),   // input wire CLK
  .A(a),       // input wire [3 : 0] A
  .B(b),       // input wire [3 : 0] B
  .P(p)        // output wire [7 : 0] P
);

endmodule
```

以demo为顶层模块，启动行为仿真，即可输出波形。设置a、b、p显示为无符号十进制（右击选择Radix - Unsigned Decimal）。如图，可以看到a=7, b=8，第一个时钟上升沿后p = a * b = 56。