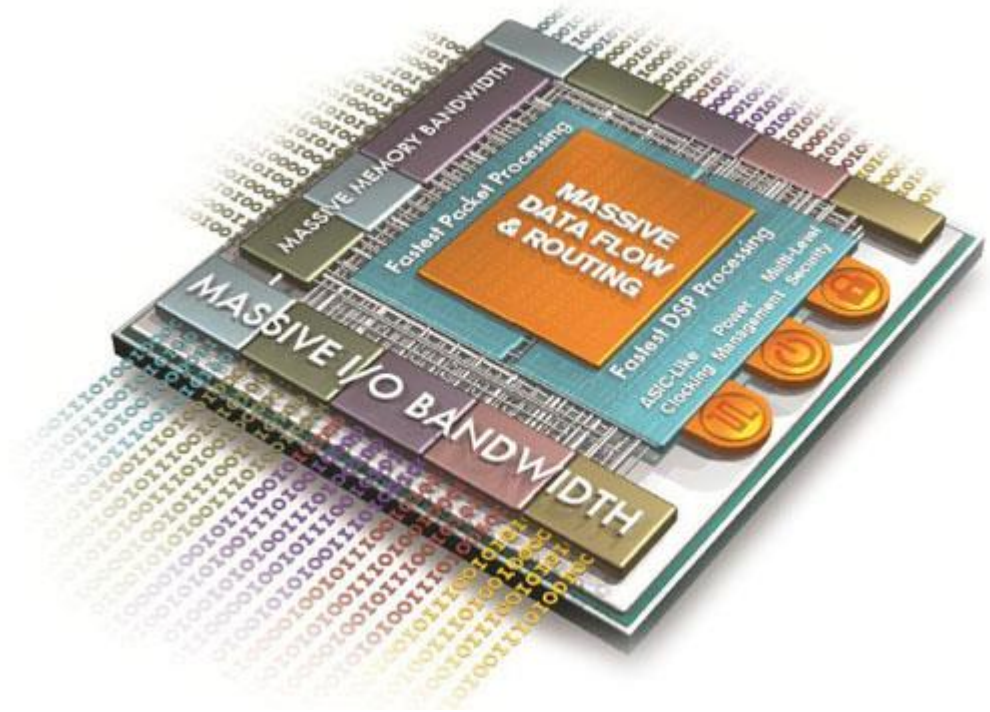
FPGA Basys3 开发实验指导书



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 VIVADO 编译环境（一）

一、 实验目的

1. 熟悉 VIVADO 的编译环境；
2. 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 Basys3 开发板上对程序进行最终验证。

四、 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：

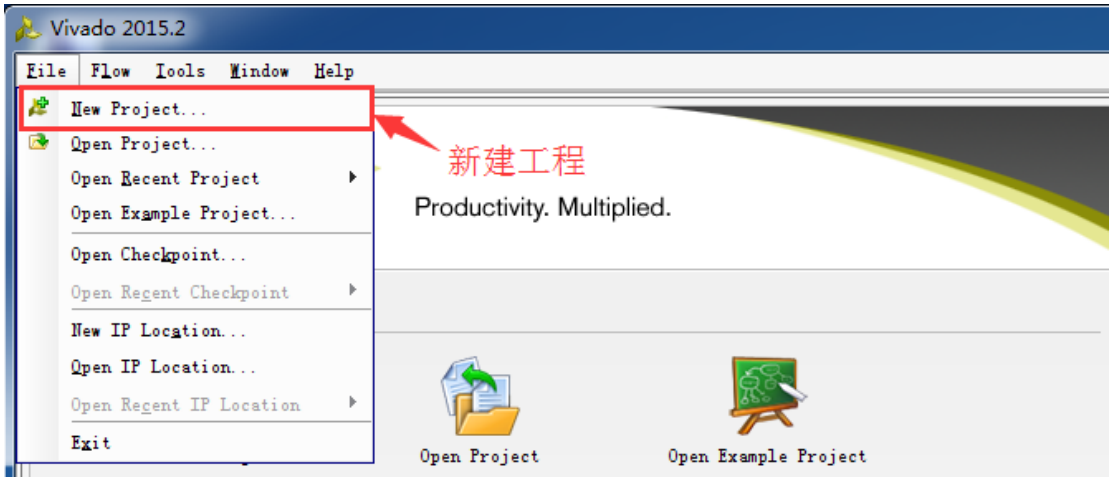


图 1.1、VIVADO 窗口界面

(2) 利用向导，建立一个新项目。

- 在 File 菜单中选择 New Project 选项启动项目向导。
- 填写所要新建的工程名。如这里的工程名：led，工程所在位置：C:/Workspace/Vivado，然后点击 Next。如图 1.2 所示：

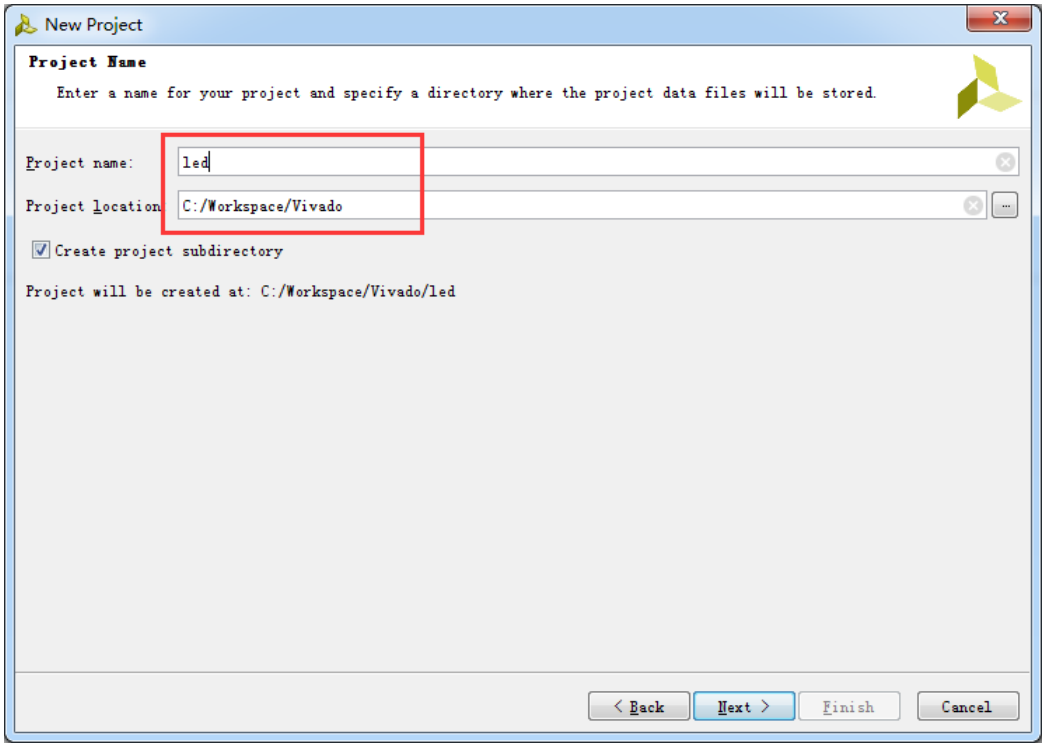


图 1.2、VIVADO 项目名称、路径设定窗口

- 在 File 选择项目类型，如图 1.3 所示：

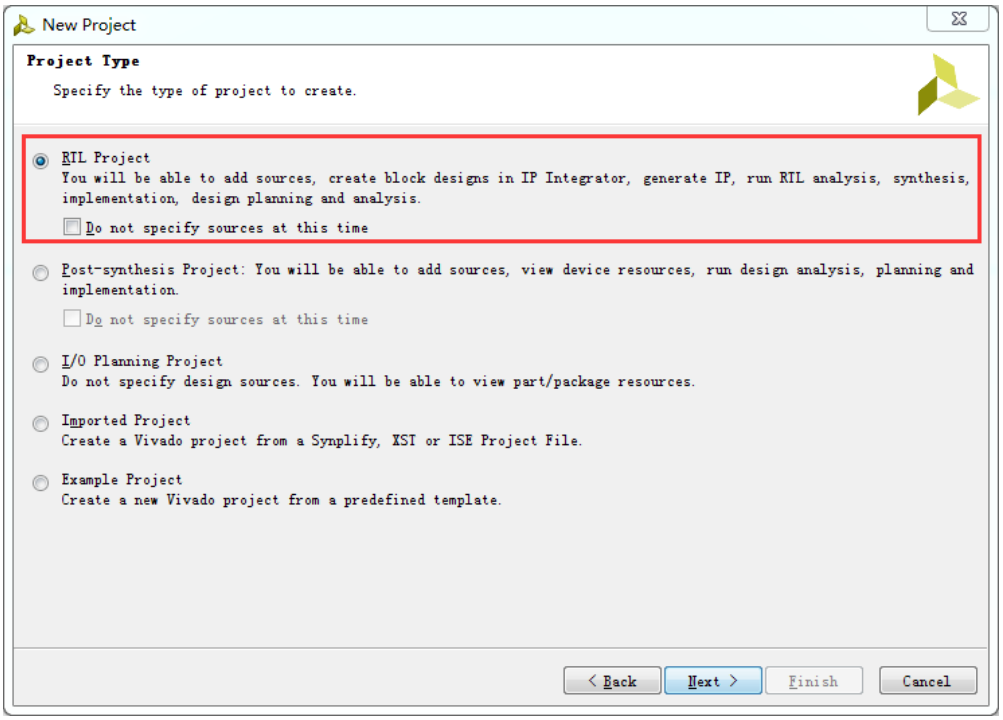


图 1.3、项目类型

- 由于我们是新建工程，所以此处默认没有可以添加的源文件，并且设置编程语言和仿真语 言均设置为 Verilog。点击 Next，如图 1.4 所示：

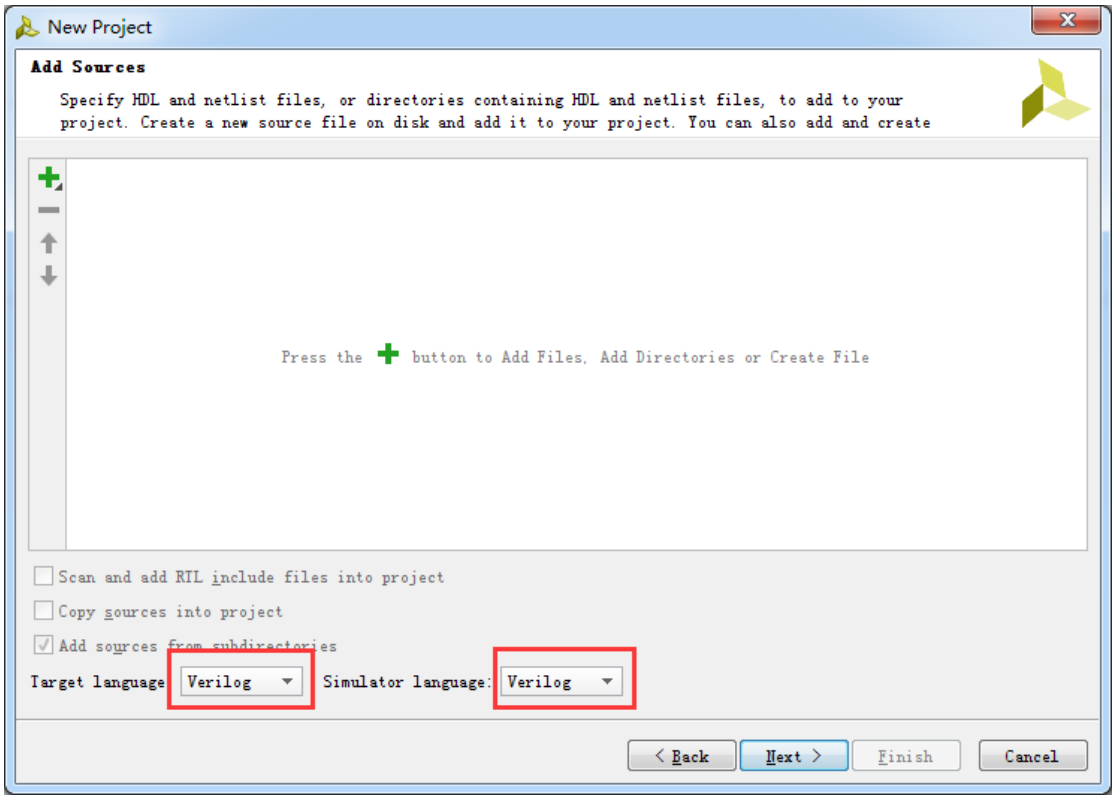


图 1.4、语言选择

- 没有可以添加的 IP，所以不添加 IP,直接点击 Next
- 也没有可以添加的约束文件，所以不添加，直接点击 Next
- 器件的选择是和实验平台的硬件相关的，根据我们的 Basys3 实验开发板，它使用的是 xc7a35tcpg236-1 的器件，找到相应的器件，如图 1.5 所示：

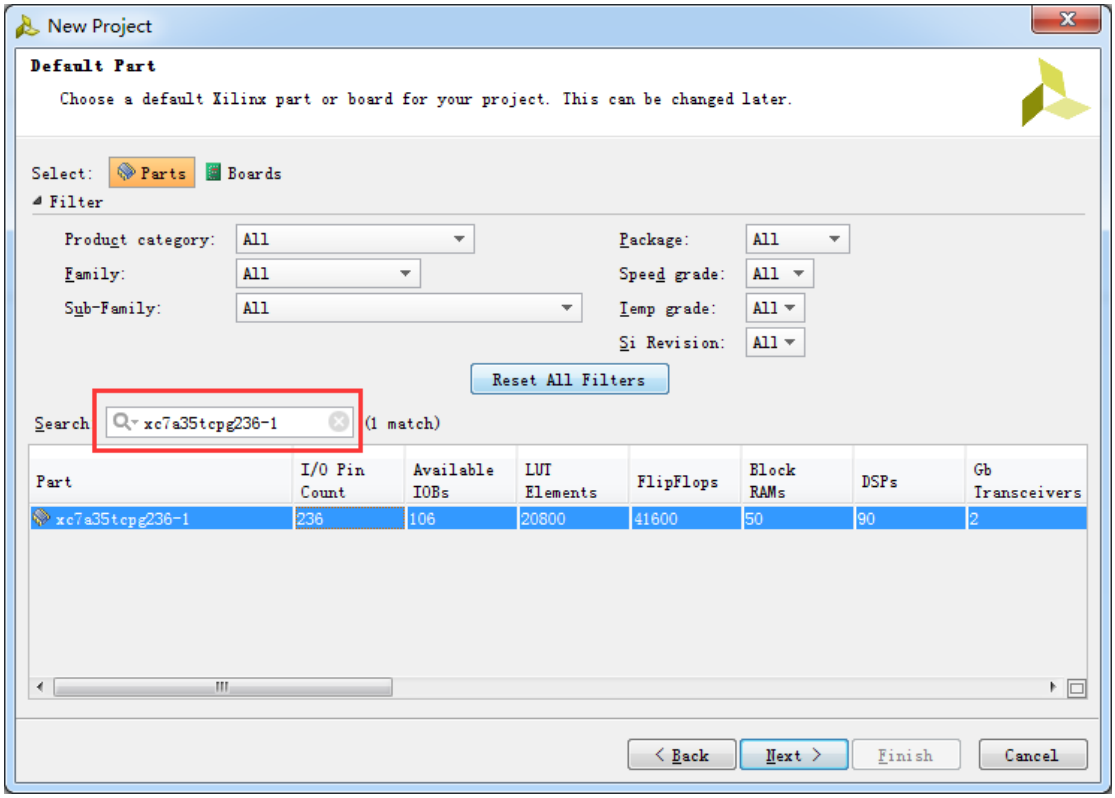


图 1.5、VIVADO 中器件选择窗口

- 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 1.6 所示：

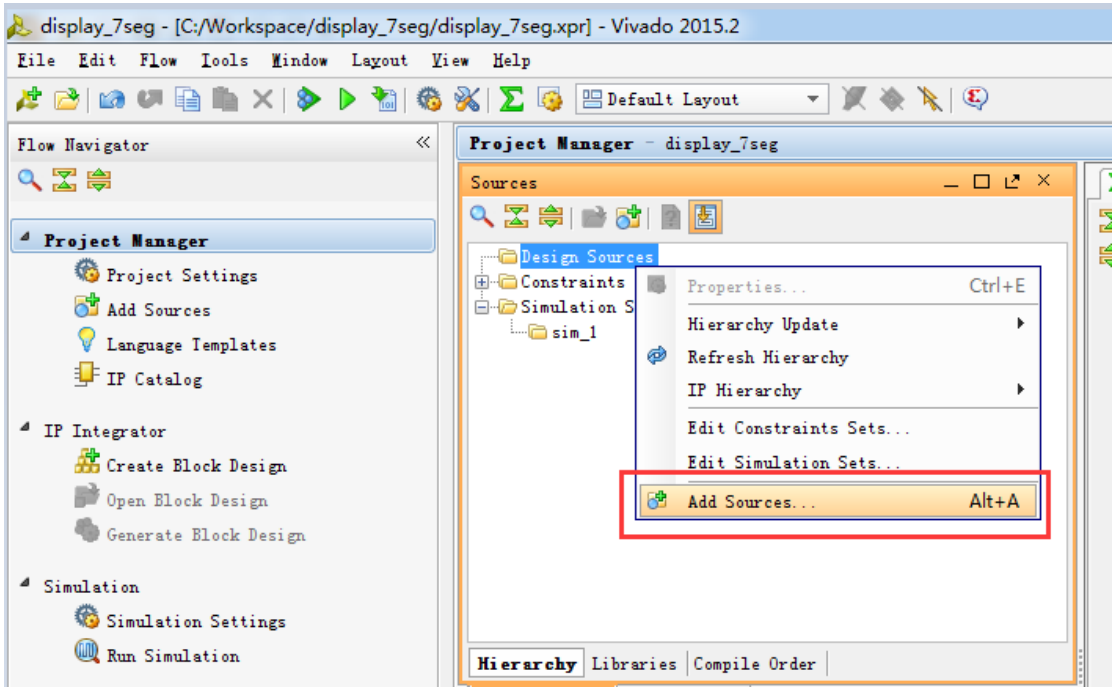


图 1.6、添加源文件

- 选择第二个选项，如图 1.7 所示：

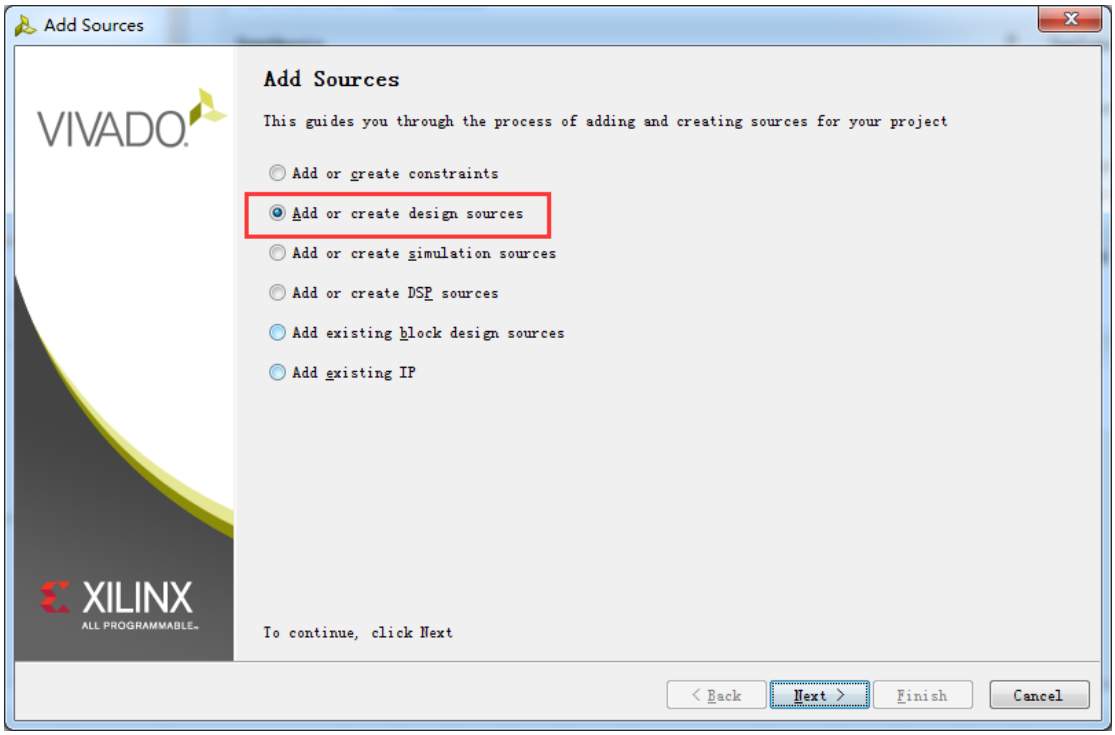
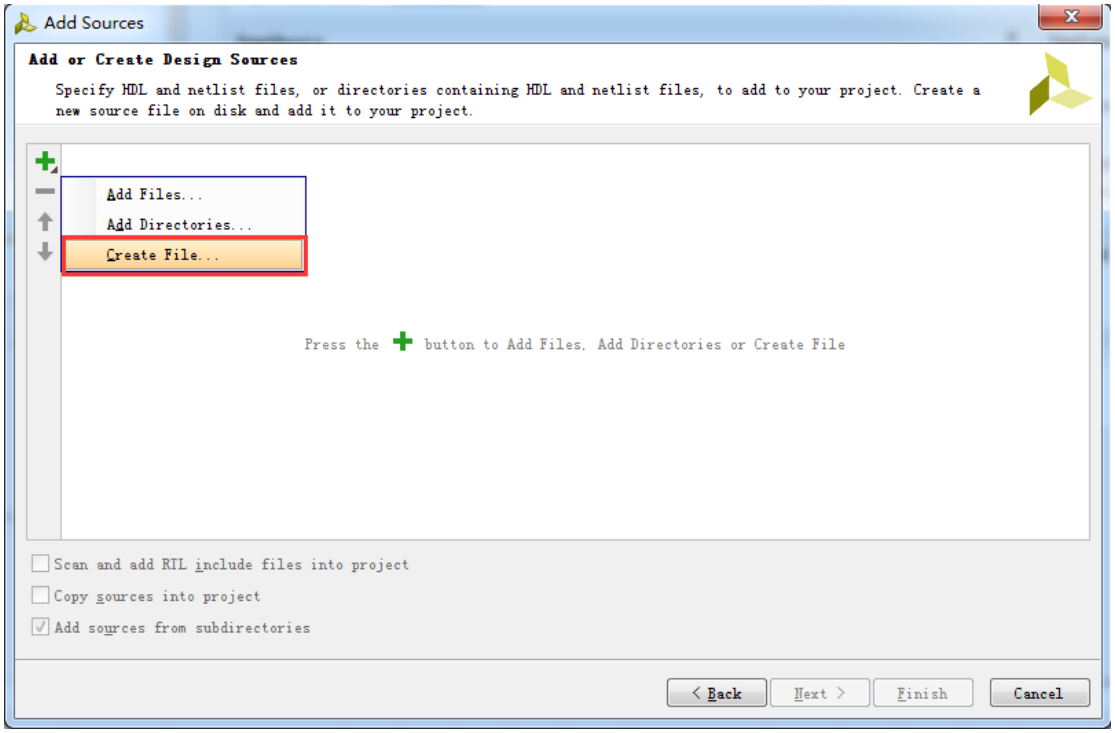


图 1.7、选择文件类型

- 选择 Create File…在弹出下面窗口填写新建源文件名称为 display_7seg



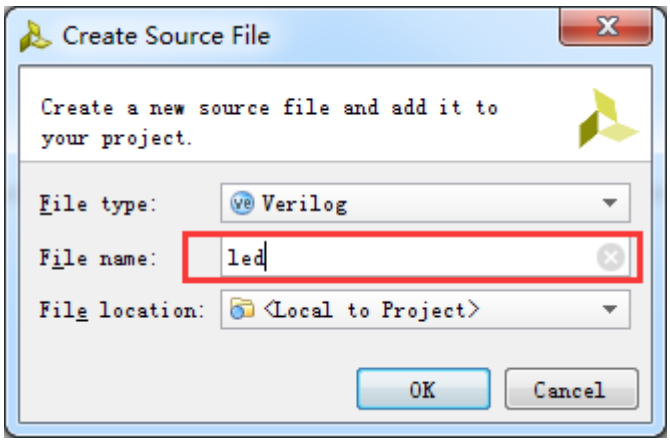


图 1.8、创建文件

- 创建完成点击 Finish

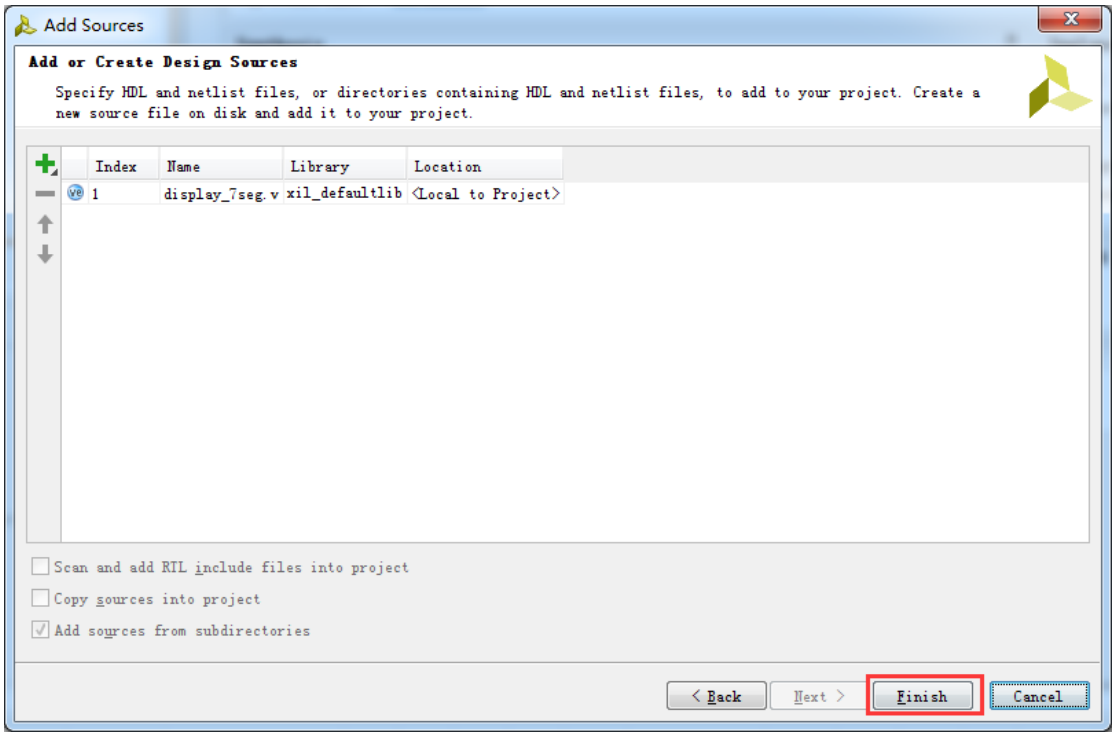


图 1.9、创建完成

- 填写模块名称和端口，如图 1.10 所示：

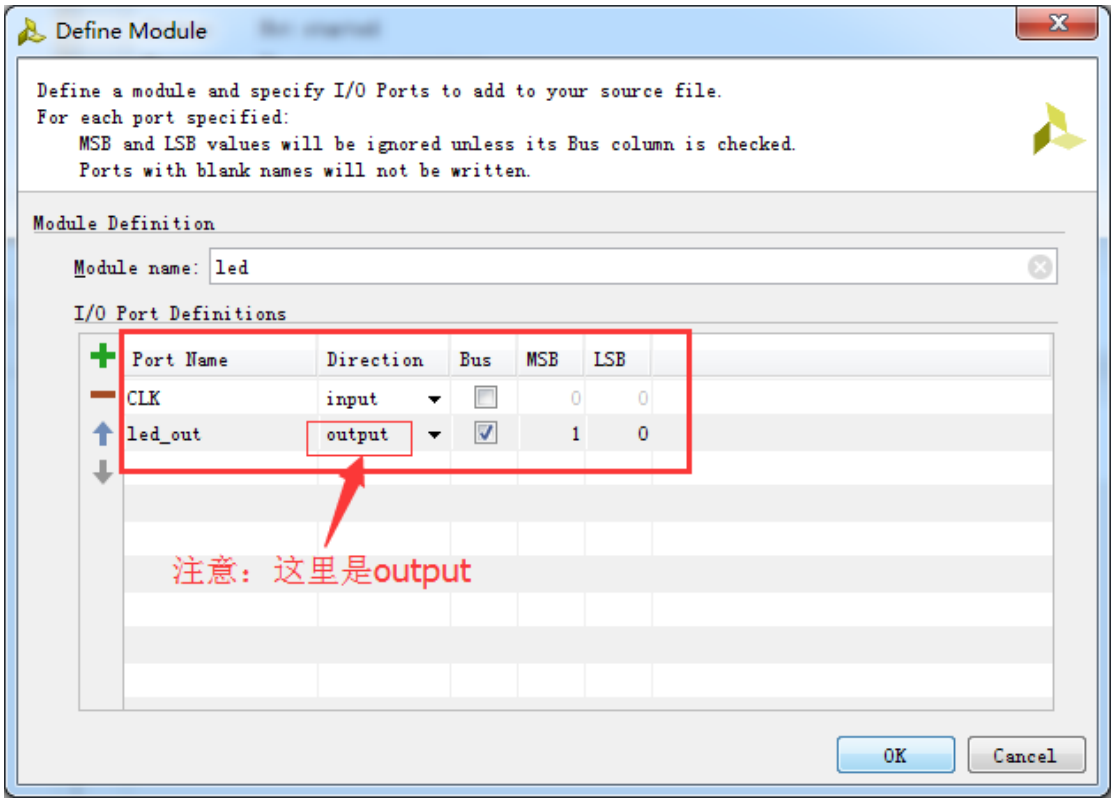


图 1.10、Define Module 窗口

(3) 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 1.11 所示：

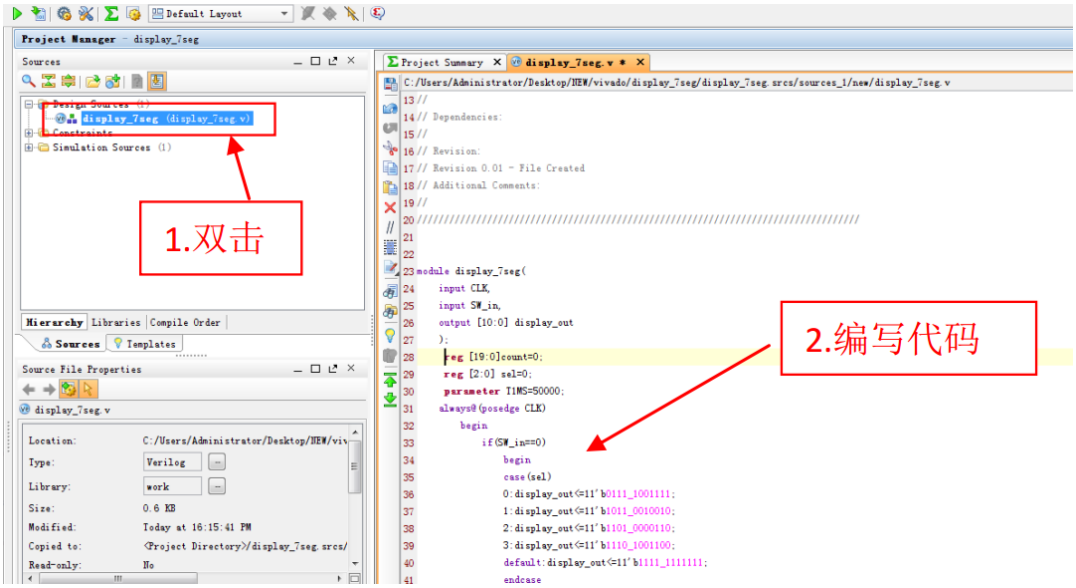


图 1.11、Verilog 代码编辑窗口

编辑代码如下：

```
module led(
input CLK,
output reg[1:0] led_out
);
reg [32:0]count=0;
parameter T1MS=50000000;
always@(posedge CLK)
begin
count<=count+1;
if(count==T1MS)
begin
count<=0;
end
if(count<25000000)
begin
led_out<=2'b01;
end
else
begin
led_out<=2'b10;
end
end
Endmodule
```

(4)VIVADO 程序编译。

- 完成 Synthesize 的综合编译。如图 1.12 所示：

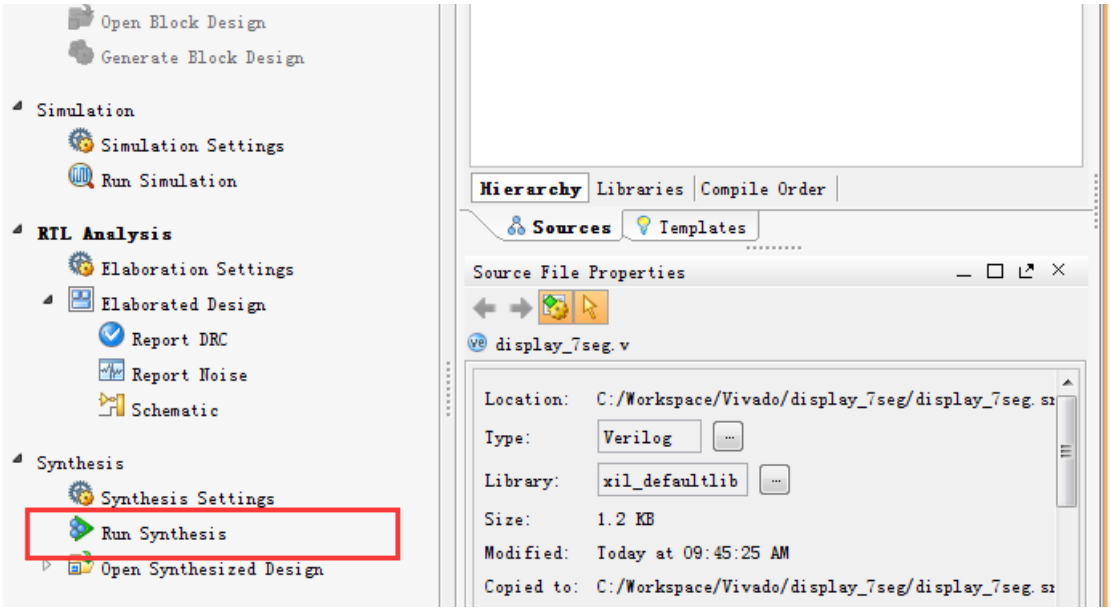


图 1.12、Verilog 代码 Synthesize 综合编译

- 编译成功后双击 Schematic 可以查看 RTL 级电路图。如图 1.13 所示：

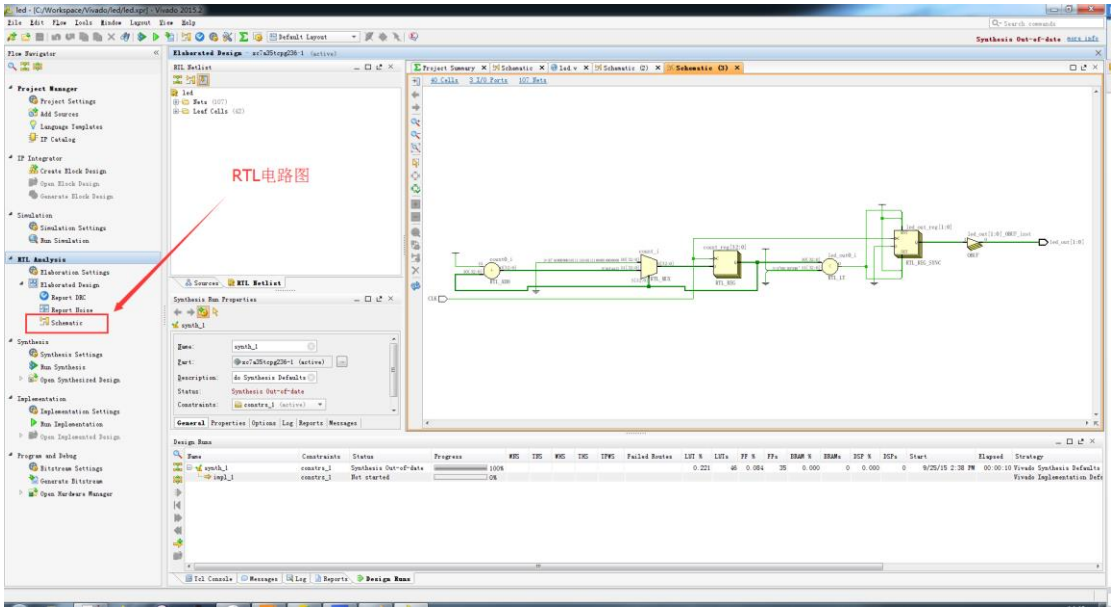


图 1.13、RTL 级电路图

(4)分配引脚。

- 右击约束子目录下文件夹，选择 Add Sourse…，如图 1.14 所示：

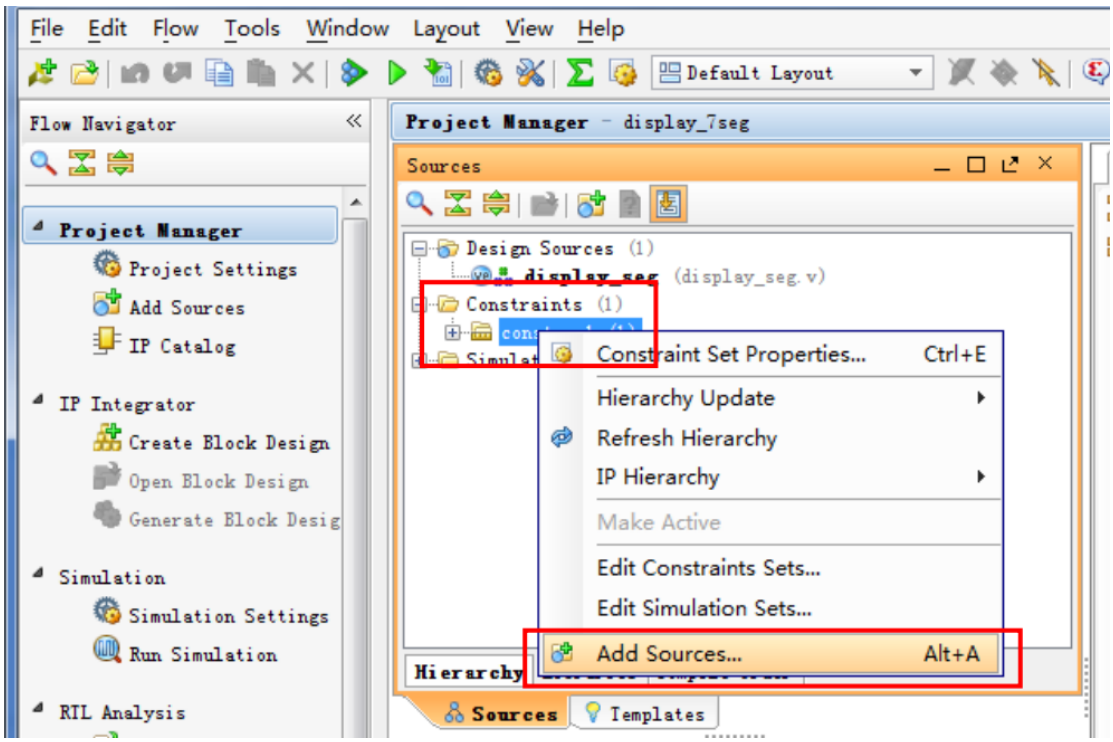


图 1.14、RTL 级电路图

- 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

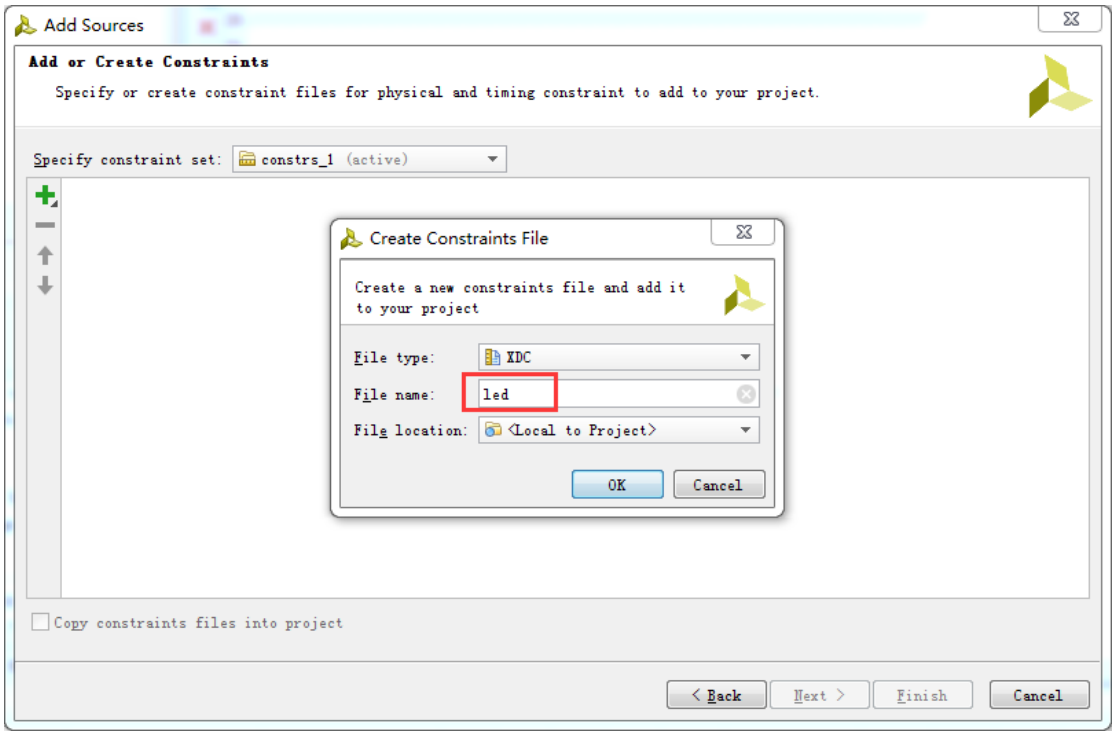


图 1.15、创建约束文件

编写约束文件如下：

```
set_property PACKAGE_PIN W5 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports CLK]
set_property PACKAGE_PIN U16 [get_ports {led_out[0]}]
set_property PACKAGE_PIN L1 [get_ports {led_out[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led_out[1]}]
```

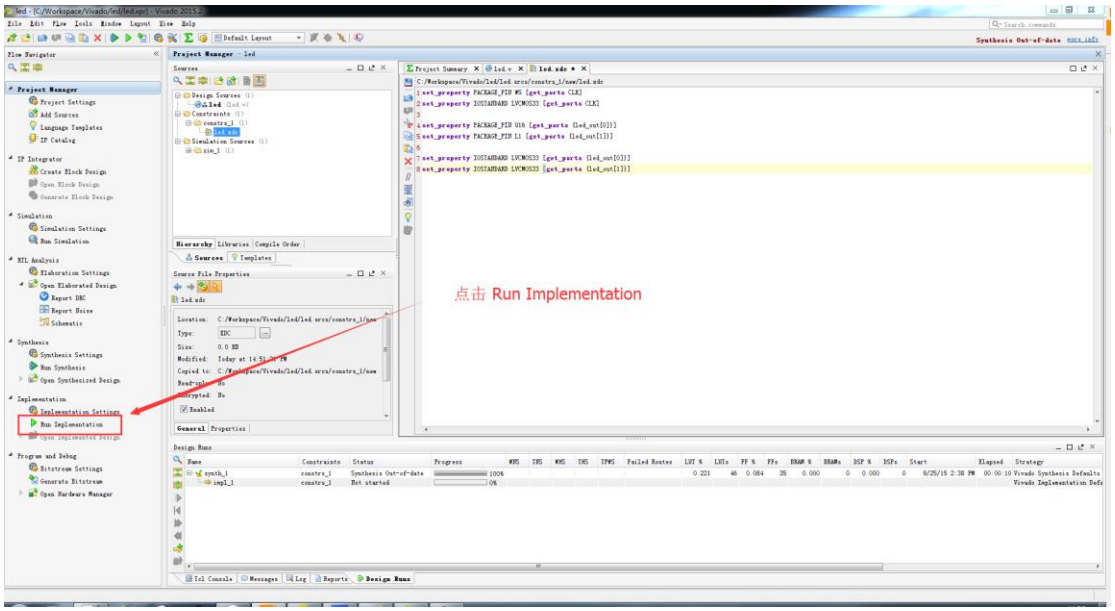


图 1.16、Impementation

- 在上图中单击 Run implementation，运行完成后弹出下面窗口：

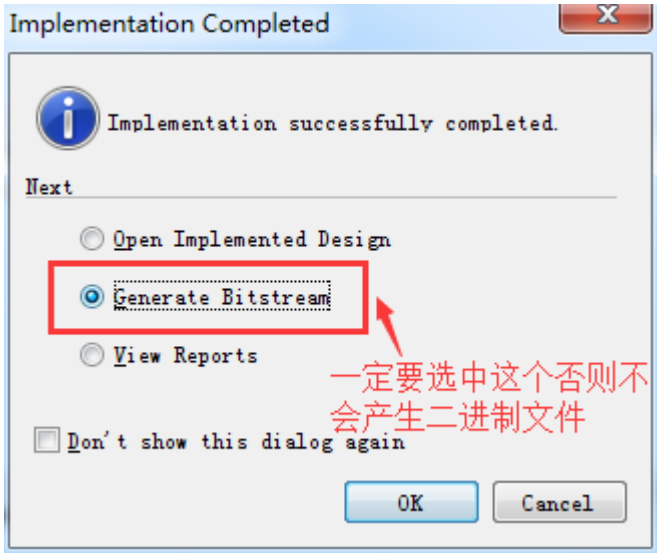


图 1.17、生成二进制文件

- 运行完成，选择 Open Hardware Manager

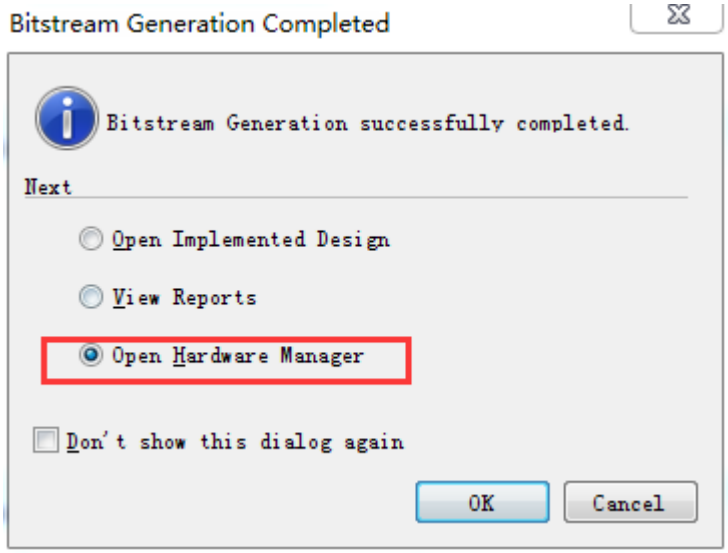


图 1.18、Hardware Manager

- 将实验板通过 USB 连接至电脑，然后点击 Auto Connect

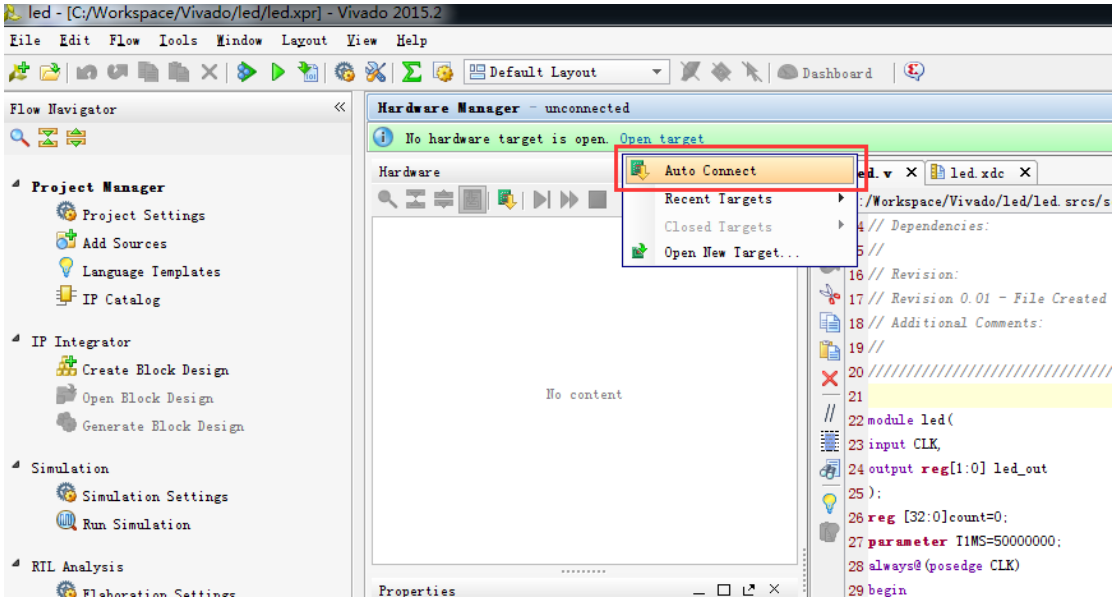


图 1.19、连接设备

- 连接成功后，右击 FPGA 芯片选择 Program Device

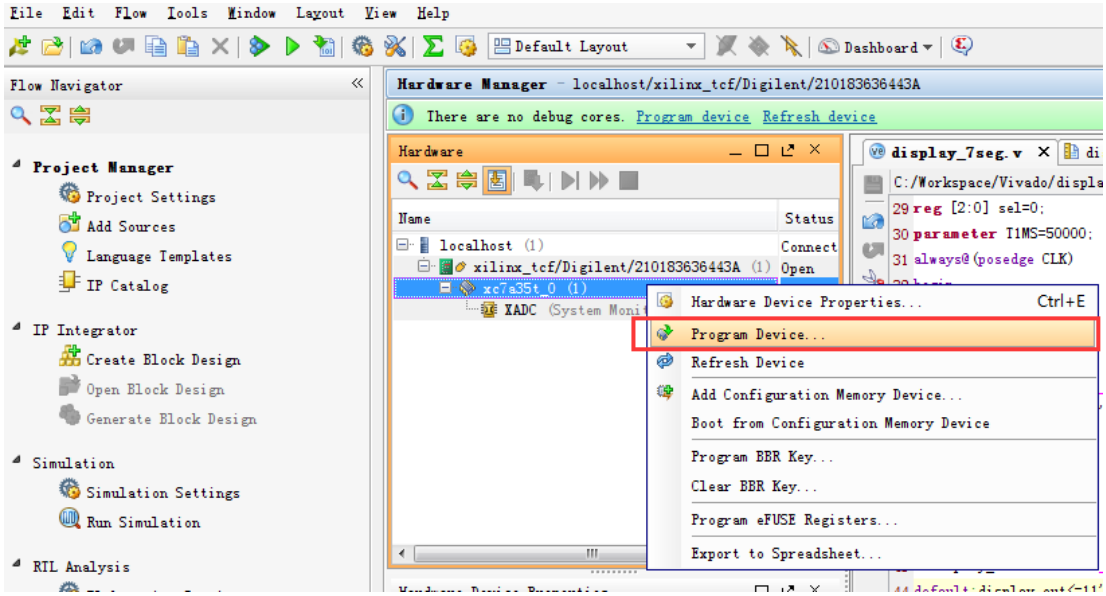


图 1.20、对开发板编程

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。
右击 FPGA 芯片选择 Add Configuration Memory Device

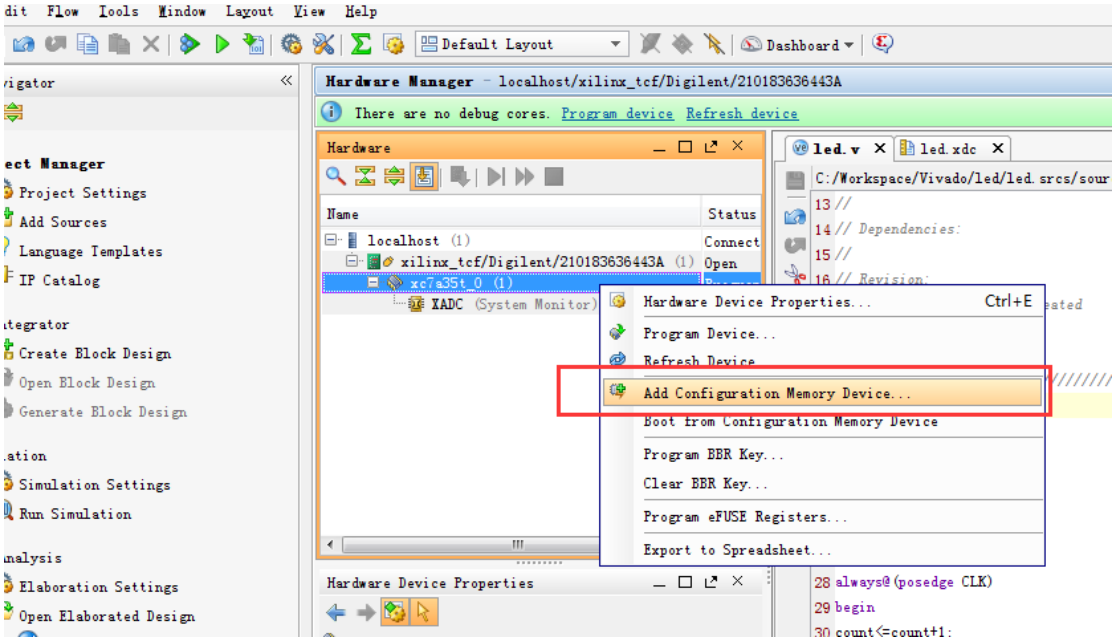


图 1.21、ROM 烧写

- 选择 flash 芯片型号，如图 1.20 所示：

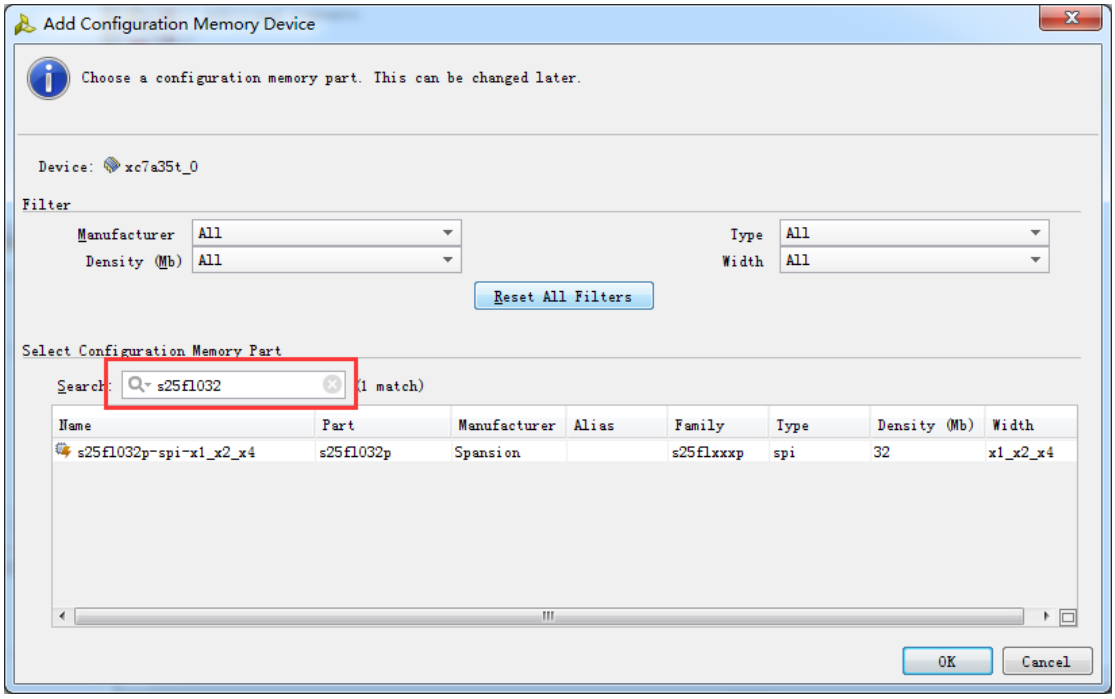


图 1.22、选择芯片型号

- 在弹出的窗口选择配置文件为前面生成的.bit 文件

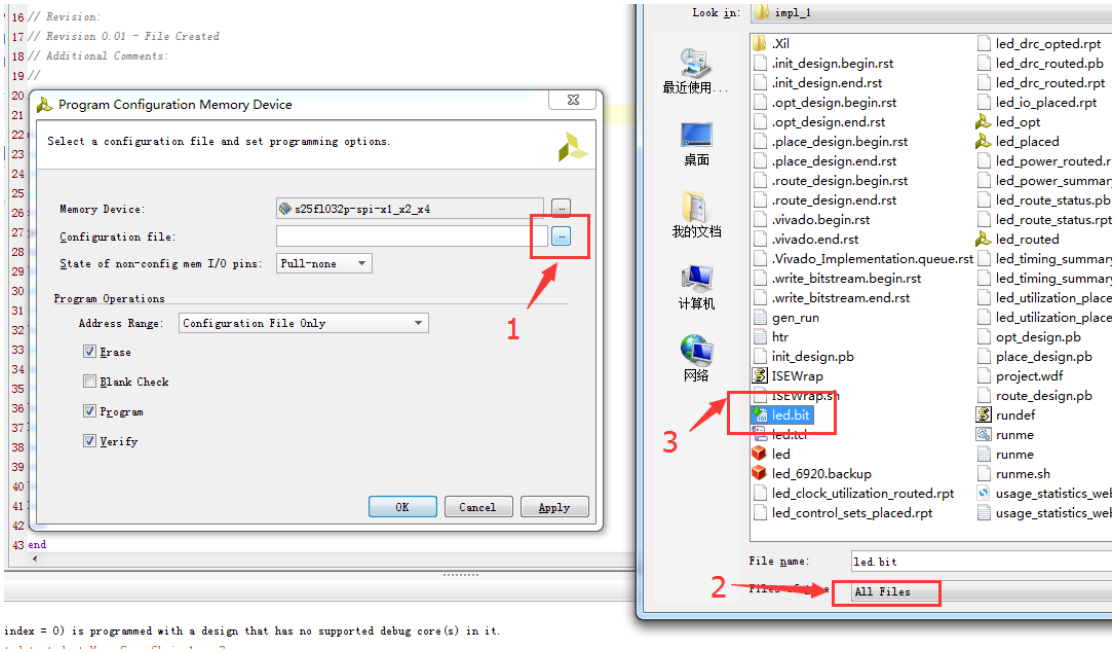


图 1.23、选择二进制文件

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。

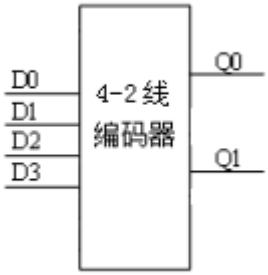


图 2.1、 四至二线编码器的电路符号

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
endmodule
```

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部分）

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。

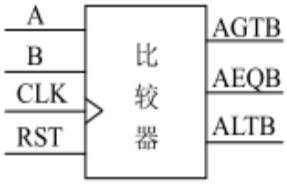


图 2.2、比较器电路符号

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;
    end
  else
    begin
    if(A>B)
      begin
      AGTB<=1;
      AEQB<=0;
      ALTB<=0;
      end
    else if(A==B)
      begin
      AGTB<=0;
      AEQB<=1;
      ALTB<=0;
```

```
        end
      else
        begin
        AGTB<=0;
        AEQB<=0;
        ALTB<=1;
        end
      end
  end
endmodule
```

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。

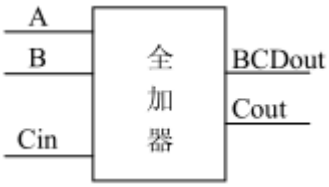


图 2.3、全加器电路符号

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	V16	拨动开关 SW1
D1	W16	拨动开关 SW2
D2	W17	拨动开关 SW3
D3	W15	拨动开关 SW4
Q0	E19	LED LD1
Q1	U19	LED LD2

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	暗	暗
1	0	1	1	亮	暗
1	1	0	1	暗	亮
1	1	1	0	亮	亮

2. 比较器：

管脚分配如下表：

程序中管脚名	实际管脚	说明
CLK	W5	全局时钟脚
A（1）	V16	拨动开关 SW1
A（0）	W16	拨动开关 SW2
B（1）	W17	拨动开关 SW3
B（0）	W15	拨动开关 SW4
RST	U18	按键 BTNC
AGTB	E19	LED LD1
AEQB	U19	LED LD2
ALQB	V19	LED LD3

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	V16	拨动开关 SW1
B	W16	拨动开关 SW2
CIN	W17	拨动开关 SW3
SUM	E19	LED LD1
COUT	U19	LED LD2

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。

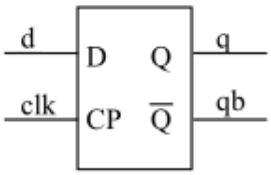


图 3.1 、电路符号

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
endmodule
```

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：

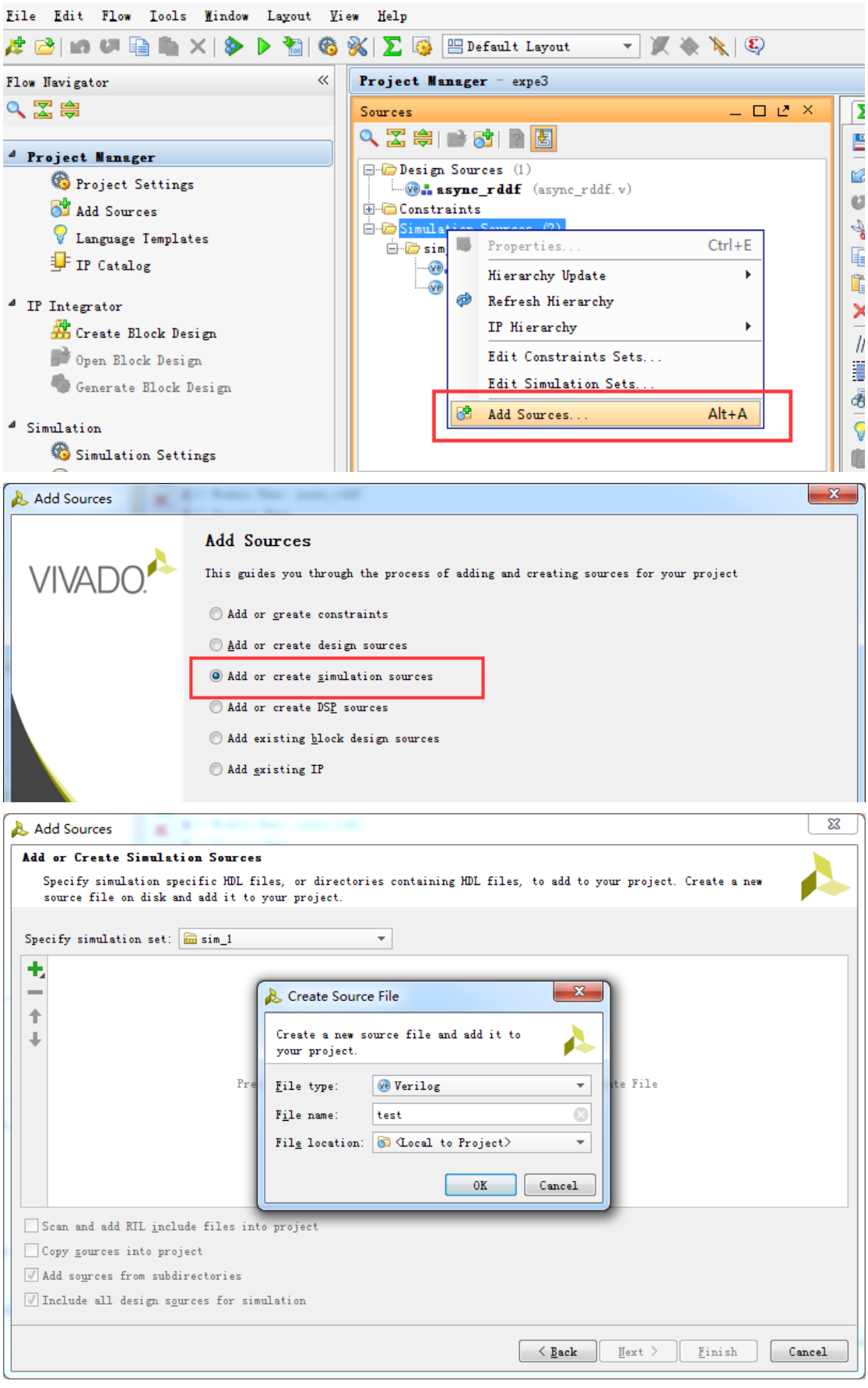
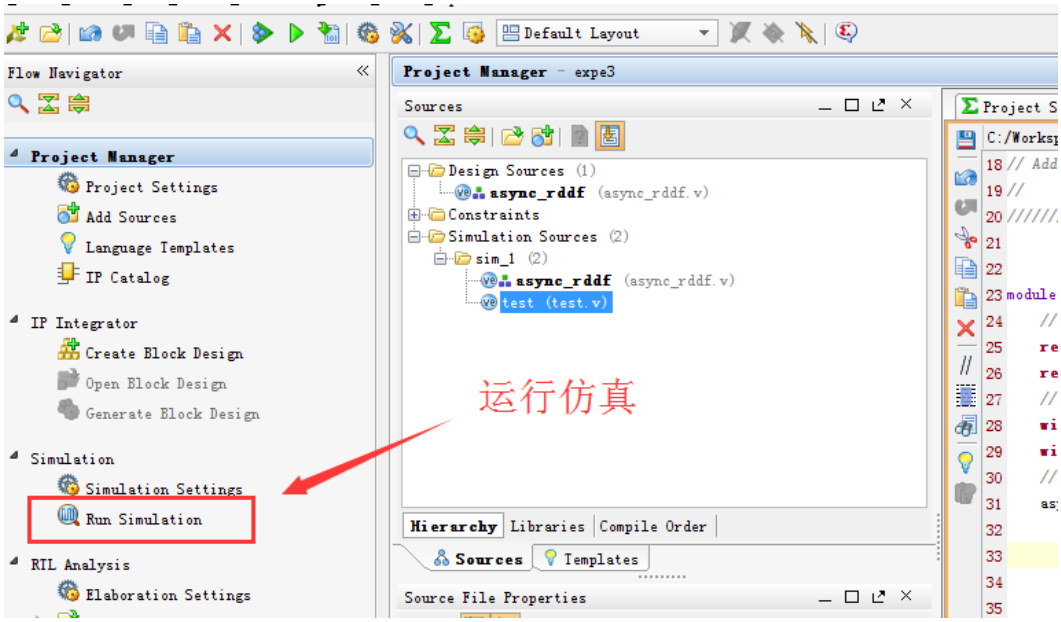


图 3.2 创建仿真源文件

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
```

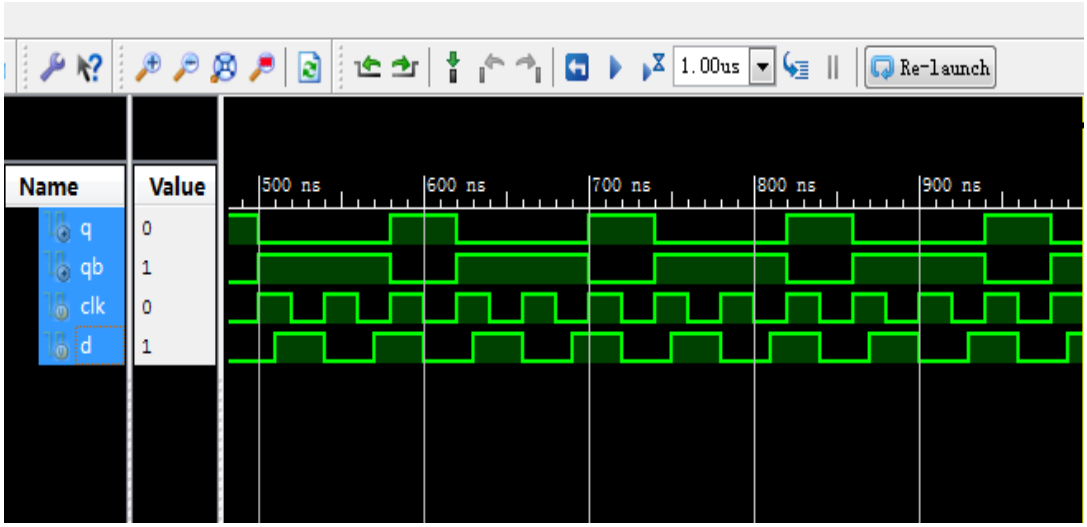
输入完成后点击左侧的 Run Simulation 进行仿真



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。

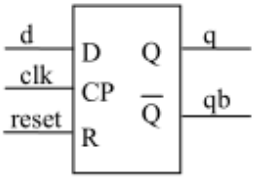


图 3.3 D 触发器电路符号

表 3.2 D 触发器的功能表

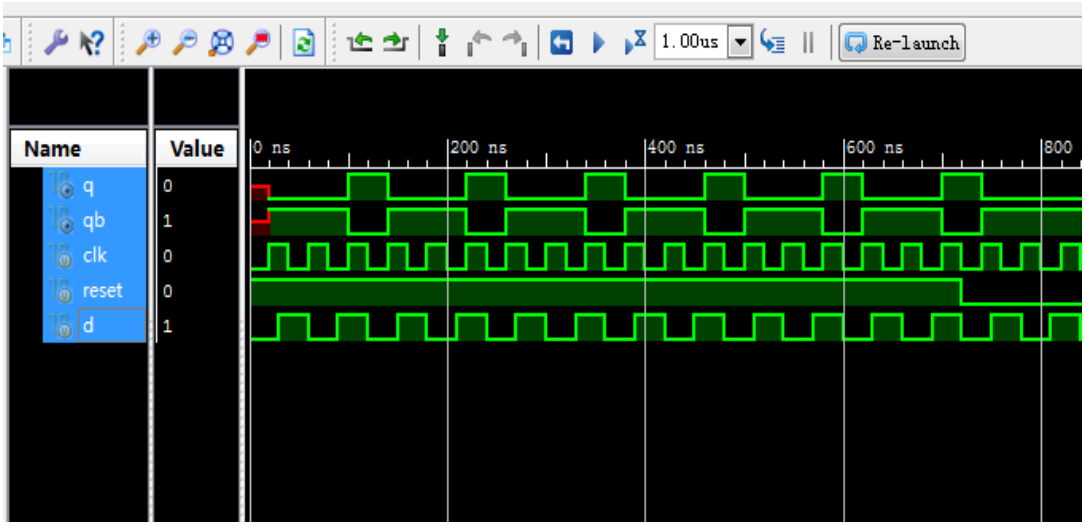
R	D	CP	Q	$\overline{Q}$
0	X	上升沿	0	1
1	X	0	保持	保持
1	X	1	保持	保持
1	0	上升沿	0	1
1	1	上升沿	1	0

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;
        end
    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。

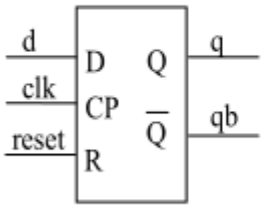


图 3.3、D 触发器电路符

表 3.2、D 触发器的功能表

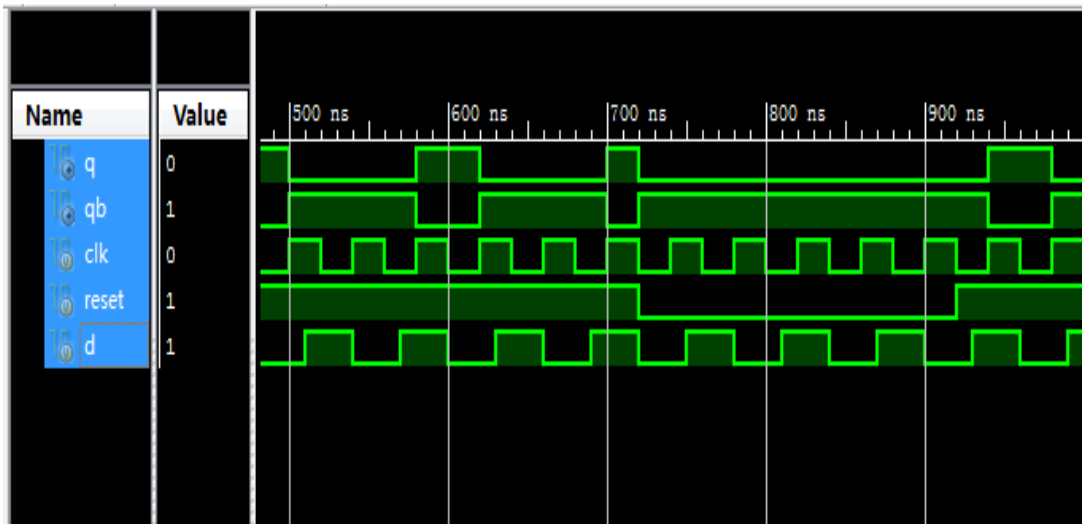
R	D	CP	Q	$\overline{Q}$
0	X	上升沿	0	1
1	X	0	保持	保持
1	X	1	保持	保持
1	0	上升沿	0	1
1	1	上升沿	1	0

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;
   end
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。

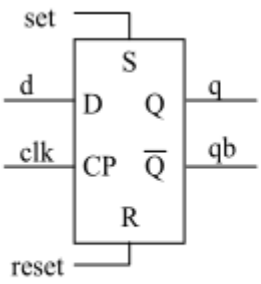


图 3.5、 电路符号

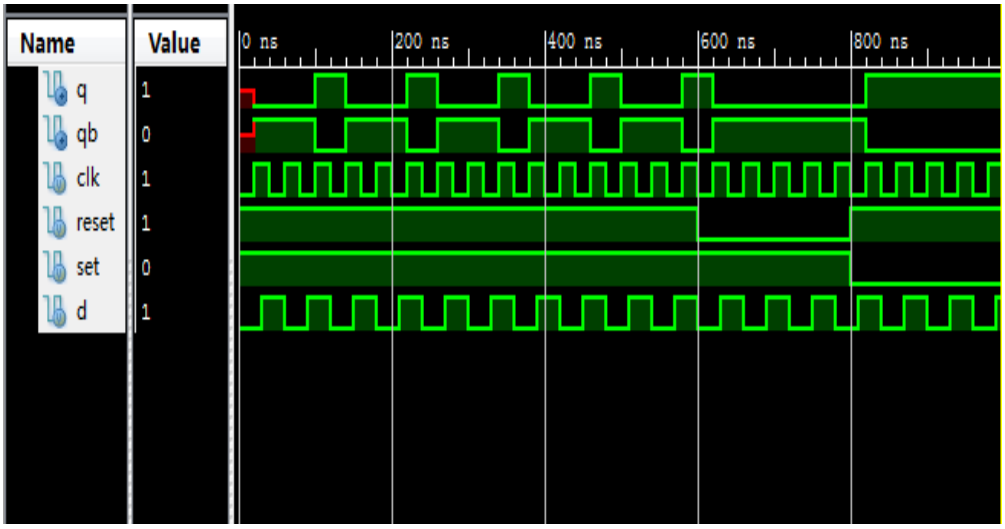
表 3.4、D 触发器的功能表

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;
   end
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。

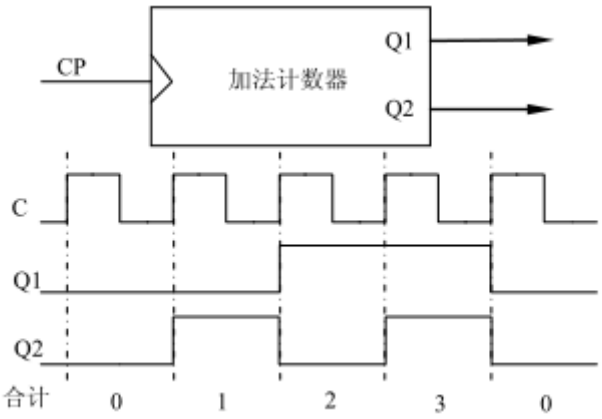


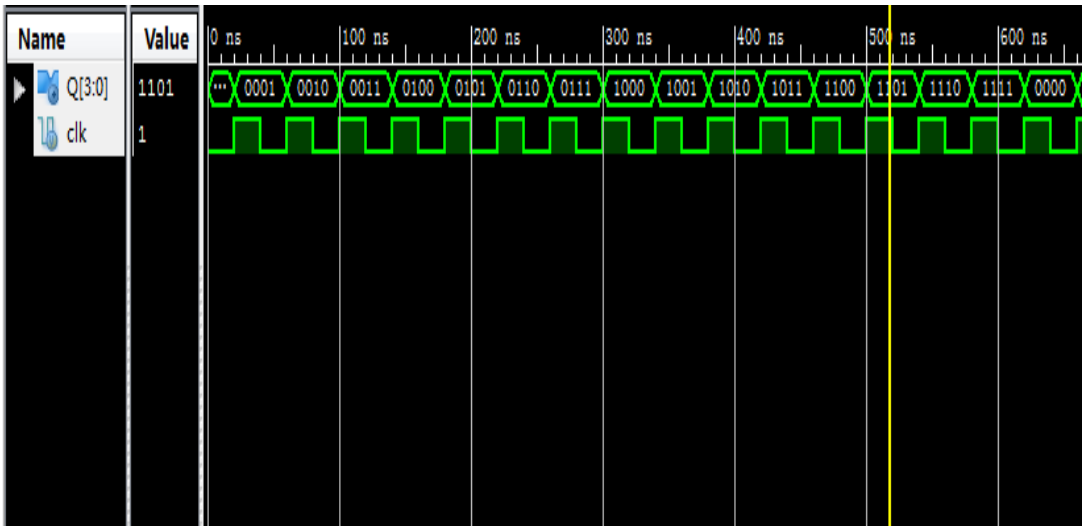
图 3.6、加法计数器的相关波形

源程序如下：

```
module addcounter(clk,Q) ;
```

```
input clk ;
output[3 :0] Q ;
reg[3 :0] Q ;
always @(posedge clk)begin
Q<=Q+1 ;
end
endmodule
```

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	L2	扩展口 JA 的 JA1
clk	W5	全局时钟
q	J2	扩展口 JA 的 JA2
qb	G2	扩展口 JA 的 JA3

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 B2，将示波器的两通道的引脚分别接到 A3 和 J3。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	L2	扩展口 JA 的 JA1
Clk	W5	全局时钟
RESET	V16	拨动开关 SW1
Q	J2	扩展口 JA 的 JA2
QB	G2	扩展口 JA 的 JA3

将信号发生器接到 B2，将示波器的两通道的引脚分别接到 A3 和 J3。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	L2	扩展口 JA 的 JA1
Clk	W5	全局时钟
RESET	V16	拨动开关 SW1
Q	J2	扩展口 JA 的 JA2
QB	G2	扩展口 JA 的 JA3

将信号发生器接到 B2，将示波器的两通道的引脚分别接到 A3 和 J3。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	W5	全局时钟
Q0	L2	扩展口 JA 的 JA1
Q1	J2	扩展口 JA 的 JA2
Q2	G2	扩展口 JA 的 JA3
Q3	H1	扩展口 JA 的 JA4

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。）

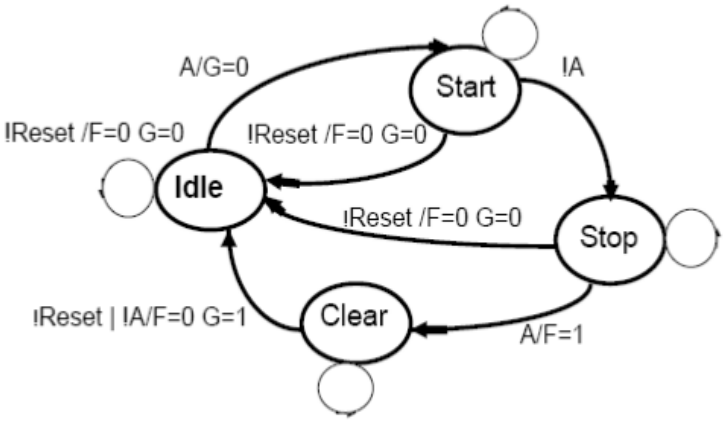


图 4.1、用三种不同编码所实现的状态图

【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
                    else
                        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
                else
                    state<=idle;
            endcase
        end
    end
    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
    end
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:begin
      state<=2'b01;
      if(sw)begin
            data<=4'b1000;
      end
      else begin
            data<=4'b0111;
      end
    end
    2'b01:begin
      state<=2'b10;
      if(sw)begin
            data<=4'b0100;
    end
      else begin
            data<=4'b1011;
      end
    end
    2'b10:begin
      state<=2'b11;
      if(sw)begin
            data<=4'b0010;
      end
      else begin
            data<=4'b1101;
      end
    end
    2'b11:begin
      state<=2'b00;
      if(sw)begin
            data<=4'b0001;
      end
      else begin
            data<=4'b1110;
      end
    end
    endcase
  end
  endmodule
```

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：

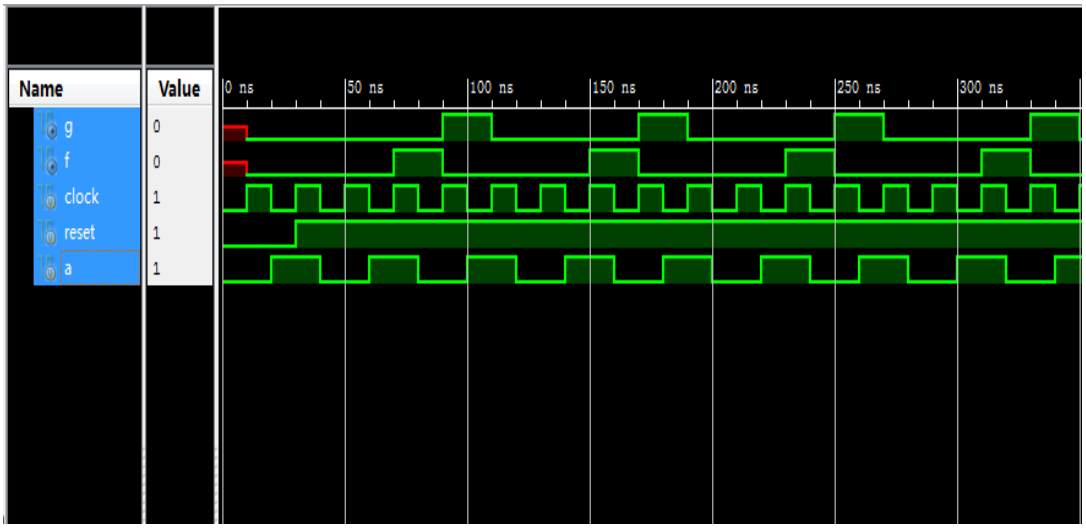


图 4.2、三种编码方式的仿真波形

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	V17	拨动开关 SW0
DATA[1]	E19	LED LD1
DATA[2]	U19	LED LD2
DATA[3]	V19	LED LD3
DATA[4]	W18	LED LD4

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）

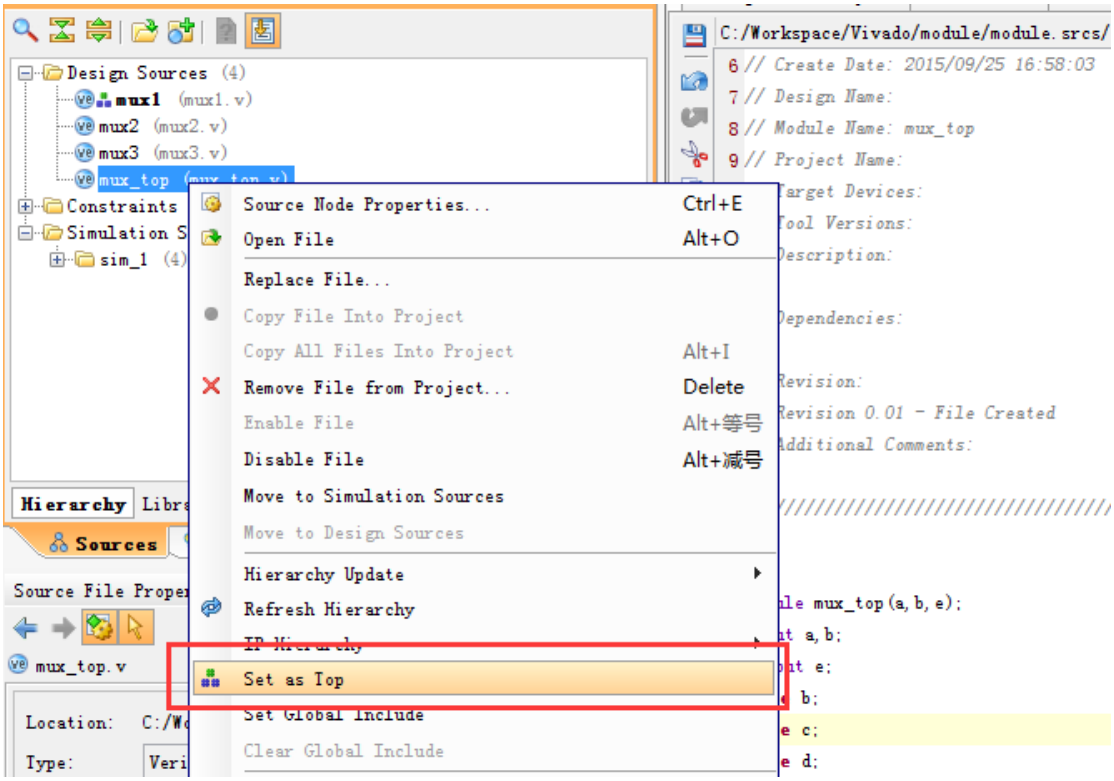
```
module mux_top(a,b,e);
input a,b;
output e;
wire b;
wire c;
wire d;
//对于第一个模块的调用 其中 mux1 为子模块名称，mux_one 为在顶层文件中引用的名称。
mux1 mux_one (.a(a),.b(b),.c(c));
//=====================================
mux2 mux_two (.b(b),.c(c),.d(d));
//=====================================
mux3 mux_three (.d(d),.c(c),.e(e));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

在软件中使用结果：

1.建立工程；

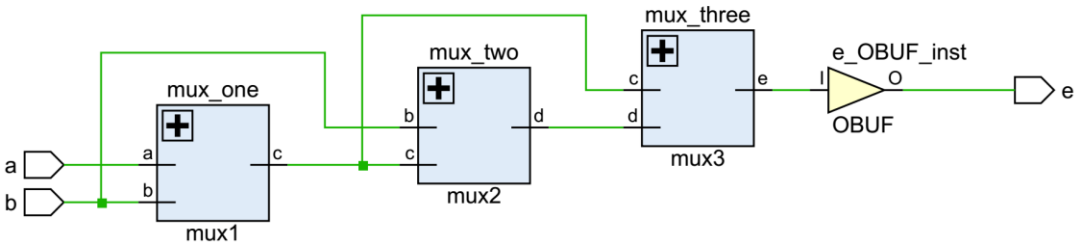
2.分别编写子模块和主模块的代码；

3.设置主模块为顶层问价； （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Basys3开发板四位动态数码管的控制；

二、 实验内容

实现对Basys3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Basys3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

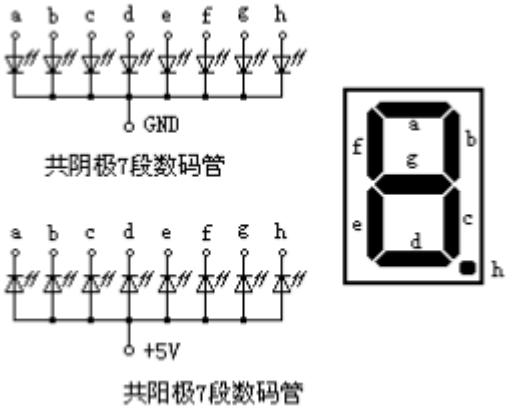


图 6.1、数码管原理图

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

Basys3 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。

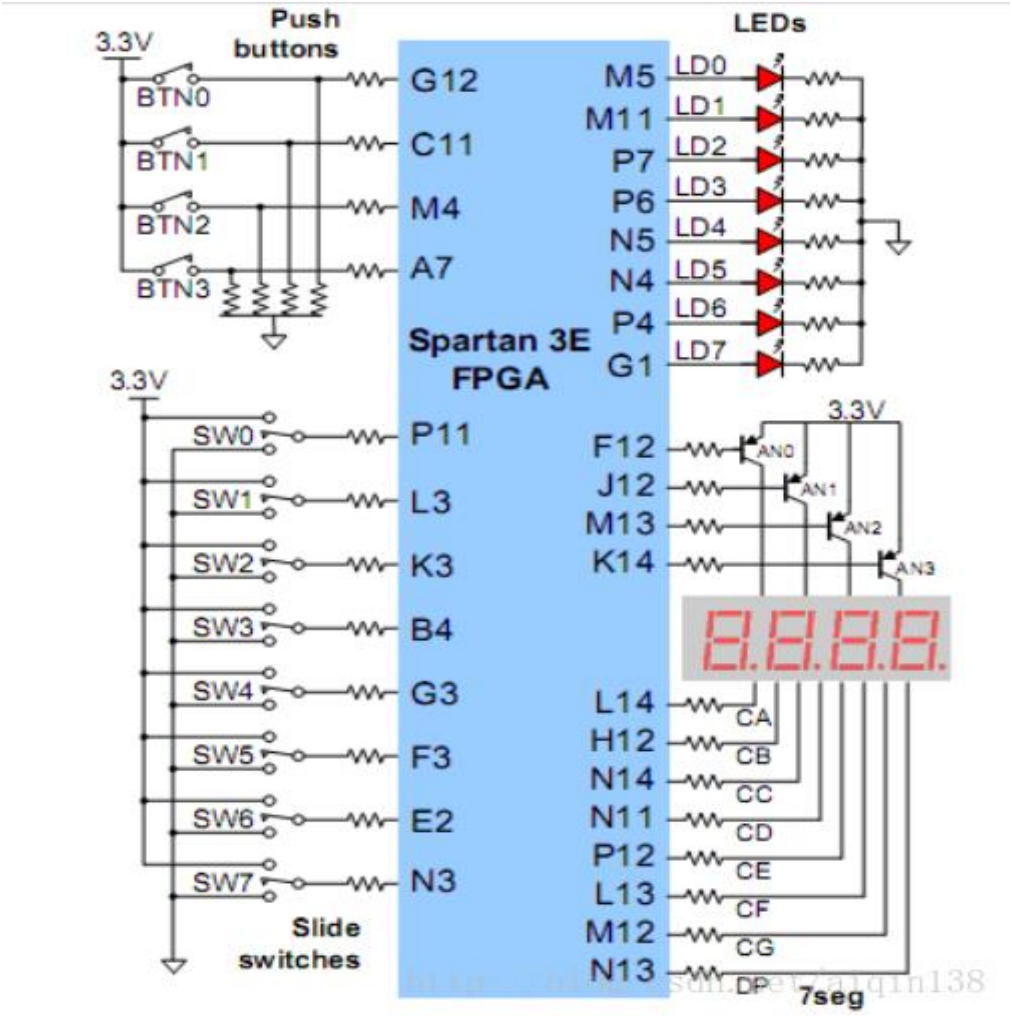


图 6.2、 四位动态数码管硬件连接图

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'b1100_0000;//0
4'h1:duan=8'b1111_1001;//1
4'h2:duan=8'b1010_0100;//2
4'h3:duan=8'b1011_0000;//3
4'h4:duan=8'b1001_1001;//4
4'h5:duan=8'b1001_0010;//5
4'h6:duan=8'b1000_0010;//6
4'h7:duan=8'b1111_1000;//7
4'h8:duan=8'b1000_0000;//8
4'h9:duan=8'b1001_0000;//9
4'ha:duan=8'b1000_1000;//a
4'hb:duan=8'b1000_0011;//b
4'hc:duan=8'b1100_0110;//c
4'hd:duan=8'b1010_0001;//d
4'he:duan=8'b1000_0111;//e
4'hf:duan=8'b1000_1110;//f
// 4'hf:duan=8'b1111_1111;//不显示
default : duan = 8'b1100_0000;//0
endcase
//-------------------------------------------------------
assign sm_wei = wei_ctrl;
assign sm_duan = duan;
endmodule
```

- 数码管设计模块

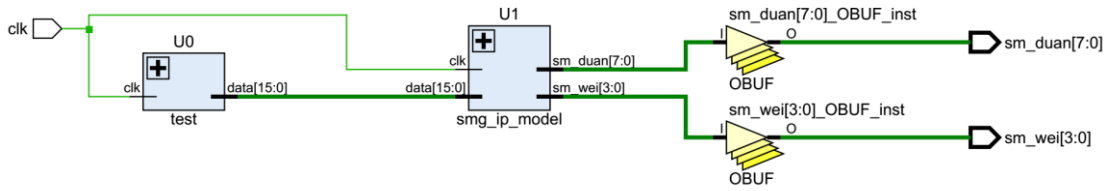
```
//测试数码管 ip
module test(clk,data);
input clk;
output [15:0]data;
//-------------------------------------------------------
//分频 1Hz
reg clk_1Hz;
integer clk_1Hz_cnt;
always @(posedge clk)
if(clk_1Hz_cnt==32'd25000000-1)
begin clk_1Hz_cnt <= 1'b0; clk_1Hz <= ~clk_1Hz;end
else
```

```
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
endmodule
```

- 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 W5 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property PACKAGE_PIN W4 [get_ports {sm_wei[3]}]
set_property PACKAGE_PIN V4 [get_ports {sm_wei[2]}]
set_property PACKAGE_PIN U4 [get_ports {sm_wei[1]}]
set_property PACKAGE_PIN U2 [get_ports {sm_wei[0]}]
set_property PACKAGE_PIN W7 [get_ports {sm_duan[0]}]
set_property PACKAGE_PIN W6 [get_ports {sm_duan[1]}]
set_property PACKAGE_PIN U8 [get_ports {sm_duan[2]}]
set_property PACKAGE_PIN V8 [get_ports {sm_duan[3]}]
set_property PACKAGE_PIN U5 [get_ports {sm_duan[4]}]
set_property PACKAGE_PIN V5 [get_ports {sm_duan[5]}]
set_property PACKAGE_PIN U7 [get_ports {sm_duan[6]}]
set_property PACKAGE_PIN V7 [get_ports {sm_duan[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[0]}]
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。