FII-PRA006/010 使用说明及实验指导书

yvonne
北京智能芯片科技有限公司

文档版本控制

文档版本	日期	修改内容记录
V1.0	2019/7/10	完成第一版，请大家帮助提意见。
V1.1	2019/7/12	需要下板的实验，在 X.4 下板验证中，每个实验都增加了实验现象以及对应的照片

目录

FII-PRA006/010 使用说明及实验指导书.....0
工程文件目录.....5
第一部分 FII-PRA006/010 开发系统简介.....1
1、 系统的设计目标.....1
2、 系统资源.....1
3、人机交互开关.....1
4、软件开发系统.....1
5、实物展示.....2
第二部分 FII-PRA006 主要硬件资源的使用与 FPGA 开发试验.....4
实验一 LED 流水灯设计.....5
1.1 实验目的.....5
1.2 实验要求.....5
1.3 实验内容.....5
1.3.1 LED 灯简介.....5
1.3.2 硬件设计.....5
1.3.3 程序设计.....6
1.4 实验验证.....15
1.4.1 下板验证前的准备工作.....15
1.4.2 下板验证方法及步骤.....17
实验二 SignalTap 实验.....19
2.1 实验目的.....19
2.2 实验要求.....19
2.3 实验内容.....19
2.3.1 开关简介和 SignalTap 简介.....19
2.3.2 硬件设计.....19
2.3.3 程序设计.....20
2.4 SignalTap Logic Analyzer 的使用及实验验证.....20
实验三 数码管显示.....23
3.1 实验目的.....23
3.2 实验要求.....23
3.3 实验内容.....23
3.3.1 数码管简介.....23
3.3.2 硬件设计.....24
3.3.3 程序设计.....25
3.4 Flash 的运用及实验验证.....28
实验四 Block/SCH 实验.....32
4.1 实验目的.....32
4.2 实验要求.....32
4.3 实验内容.....32
4.4 实验验证.....35
实验五 按键消抖实验.....36
5.1 实验目的.....36

5.2 实验要求.....36
5.3 实验内容.....36
5.3.1 按键和消抖原理简介.....36
5.3.2 硬件设计.....37
5.3.3 程序设计.....37
5.4 实验验证.....41
实验六 乘法器的使用与 modelsim 仿真.....43
6.1 实验目的.....43
6.2 实验要求.....43
6.3 实验内容.....43
6.3.1 程序介绍.....43
6.4 Modelsim 的使用及实验验证.....45
实验七 十六进制数到 BCD 码转换及应用.....53
7.1 实验目的.....53
7.2 实验要求.....53
7.3 实验内容.....53
7.2.1 十六进制数转 BCD 码原理简介.....53
7.2.2 程序介绍.....55
7.4 十六进制数到 BCD 码转换的应用.....56
7.5 实验验证.....58
实验总结与思考.....59
实验八 ROM 的使用.....60
8.1 实验目的.....60
8.2 实验要求.....60
8.3 实验内容.....60
8.3.1 程序介绍.....60
8.4 实验验证.....63
实验总结与思考.....64
实验九 利用双端口 RAM 实现帧数据的读写.....65
9.1 实验目的.....65
9.2 实验要求.....65
9.3 实验内容.....65
9.3.1 程序介绍.....65
9.3 实验验证.....73
实验总结与思考.....75
实验十 异步串行口设计与实验.....76
10.1 实验目的.....76
10.2 实验要求.....76
10.3 实验内容.....76
10.3.1 USB 转串口芯片（FT2232）简介.....76
10.3.2 硬件设计.....77
10.3.3 程序介绍.....77
10.4 实验验证.....82
实验十一 IIC 协议传输.....84

11.1 实验目的.....84
11.2 实验要求.....84
11.3 实验内容.....84
11.3.1 EEPROM 简介与 IIC 协议简介.....84
11.3.2 硬件介绍.....85
11.3.3 程序介绍.....85
11.4 实验验证.....94
实验十二 AD、DA 实验.....96
12.1 实验目的.....96
12.2 实验要求.....96
12.3 实验内容.....96
12.3.1 AD 转换芯片（PCF8591）简介.....96
12.3.2 硬件设计.....97
12.3.3 程序介绍.....97
12.4 实验验证.....99
实验十三 VGA 实验.....102
13.1 实验目的.....102
13.2 实验要求.....102
13.2 实验内容.....102
13.2.1 VGA 简介.....102
13.3.2 硬件设计.....103
13.3.3 程序介绍.....104
13.4 实验验证.....106
实验十四 以太网实验.....107
14.1 实验目的.....107
14.2 实验要求.....107
14.3 实验内容.....107
14.3.1 实验原理简介.....107
14.3.2 硬件设计.....109
14.3.3 程序设计.....109
14.4 实验验证.....126

工程文件目录

实验一：流水灯设计
实验二：开关输入及显示
实验三：十进制计数器
实验四：Quartus II 图形化自定向下的设计方法
实验五：block_debouncing
实验六：乘法器的设计与仿真
实验七：HEX_BCD，HEX_BCD_mult
实验八：ROM 的接口设计与应用
实验九：双口 RAM 的接口设计与使用
实验十：异步串行口的设计与应用
实验十一：EEPROM 接口设计及读写应用
实验十二：ADC/DAC 实验
实验十三：VGA 原理与图像发生器设计
实验十四：Ethernet

第一部分　FII-PRA006/010 开发系统简介

1、系统的设计目标

本系统设计的主要目的是配合 Intel Quartus 完成 FPGA 学习、开发与实验。主器件采用 Intel Cyclone10 10CL006YE144C8G 或 10CL010YE144C8G，目前是 Intel 最新一代 FPGA 器件。可完成主要学习和开发项目如下：

（1）基础的 FPGA 设计训练。

（2）SOPC（NIOS II）系统的构建及训练。

（3）身为功能完善的开发板的同时，也可以作为下载器使用，为 Altera，Xilinx，RISC-V 提供下载服务。

2、系统资源

（1）串行 flash

SPI 接口：serial flash（16M bytes）

（2）串行 EEPROM

（3）千兆以太网：100/1000 Mbps

（4）USB 转串行接口：USB-UART　bridge

3、人机交互开关

（1）8 个拨动开关

（2）5 个按键

4 个按键定义为（up，down，left，right）

（3）8 个 LED 灯

（4）6 个 7 段位数码管

（5）I2C 总线接口

（6）USB to UART 接口

（7）JTAG 编程接口

（8）2 个 12 脚 GPIO 连接口，符合 PMOD 接口标准

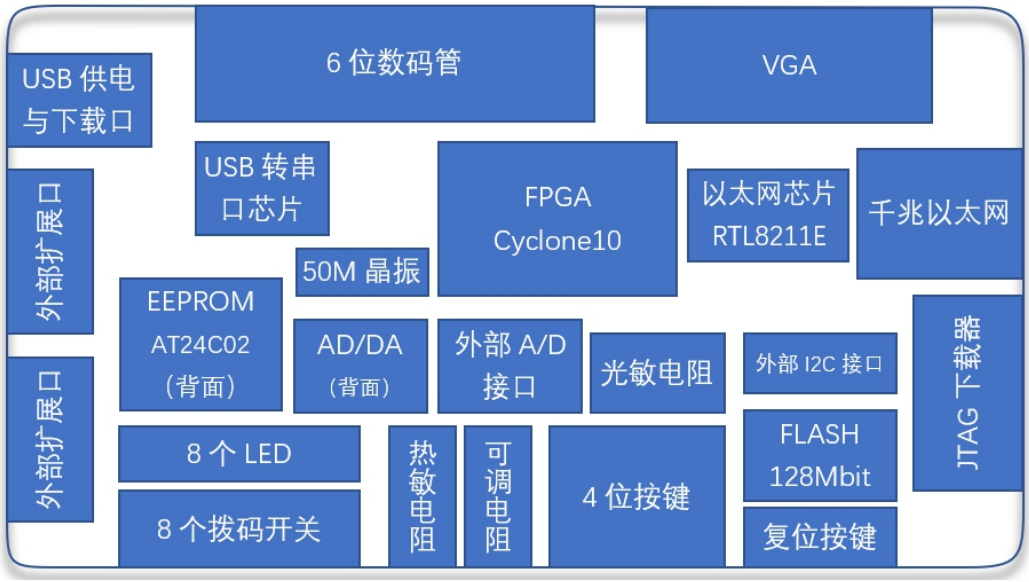
（9）VGA 接口

4、软件开发系统

Quartus 18.0 及后续版本，用于开发 FPGA， Nios-II SOPC

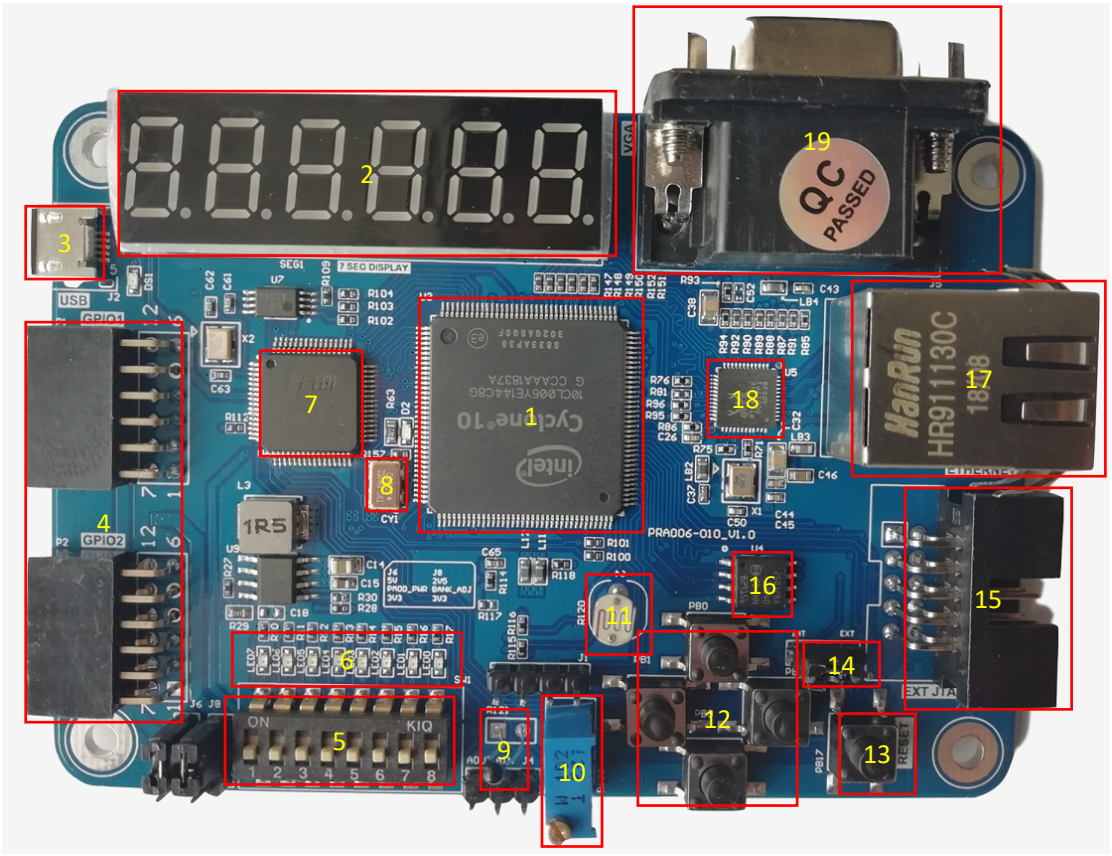
5、实物展示

（1）FII-PRA006 系统框图如图一所示：



图一　FII-PRA006 系统框图

（2）FII-PRA006 实物如图二所示



图二　FII-PRA006 系统框图

（3）对应图二实物图，开发板主要器件具体如下：

1、10CL006YE144C8G 或 10CL006YE144C8G

2、6 位供养数码管

3、USB 供电与下载接口
4、外部扩展接口 GPIO
5、8 个拨码开关
6、8 个 LED 灯
7、USB 转串口芯片（FT2232）
8、50MHz 晶振
9、热敏电阻
10、电位器
11、光敏电阻
12、4 个按键（up，down，right，down）
13、复位按键（Reset）
14、JTAG 下载器功能转换接口
15、JTAG 下载接口
16、Flash（N25Q128A，128M bit/16M bytes）
17、以太网接口
18、以太网 PHY 芯片（RTL8211E-VB）
19、VGA 接口
20、实物背面 U8：AD/DA 转换芯片（PCF8591）
21、实物背面 U13：EEPROM（AT24C02N）

第二部分　FII-PRA006 主要硬件资源的使用与 FPGA 开发试验

本部分主要通过 FPGA 的开发实例逐渐引导用户学会 FPGA 程序的开发以及板载硬件的使用，同时结合实例由浅入深的介绍开发系统软件 Quartus II 使用。本部分所涉及的开发练习内容如下。

实验一　LED 流水灯设计

1.1　实验目的

（1）　练习开发软件 Quartus II xx 的使用，新项目的建立，以及系统资源 IP Core 的使用；
（2）　熟练 Verilog HDL 程序的编写，达成良好的代码书写风格；
（3）　掌握分频的设计，实现流水灯设计；
（4）　结合硬件资源进行 FPGA 的管脚分配，并实现实际下板检验；
（5）　观察实验现象，总结实验结果。

1.2　实验要求

（1）　运用全部 LED 灯，复位时全部点亮；
（2）　结束复位，LED 灯由低到高（由右向左）依次点亮；
（3）　每个 LED 灯亮 1 秒钟时间；
（4）　最后一个（最高位）灯点亮后，下一次回到第一个（最低位）灯，实现循环；

1.3　实验内容

1.3.1　LED 灯简介

LED 灯，全称 Light-Emitting Diode Light，别名发光二极管。

它的特点是工作电流小，可靠性高，寿命长。发展至今，LED 灯种类很多，如图 1-1 所示，FII-PRA006/010 采用的是图中红圈内的 LED 灯。

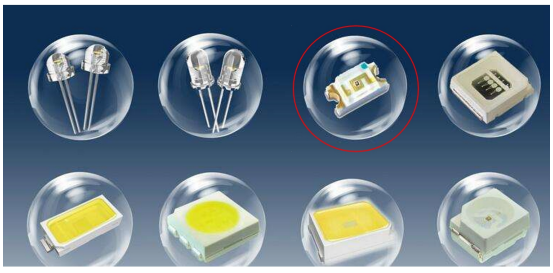


图 1-1　不同种类的 LED 灯

1.3.2　硬件设计

板载 8 位 LED 灯实物图如图 1-2 所示，LED 灯原理图如图 1-3 所示（红色框），可以看到本次实验板的 LED 灯模块采用的是 8 个共阳极 LED 灯，通过 270R 电阻与 Vcc 3.3V 相连，负极通过 N 沟道场效应管（绿色框）接地，场效应管栅极通过 4.7K 电阻接地，FPGA 通过 SW_LED 端（蓝色框）与场效应管栅极相连，因此当 FPGA 置 1，输出高电平时，场效应管导通，然后 LED 等导通，LED 灯点亮。



图 1-2　8 位 LED 灯实物图

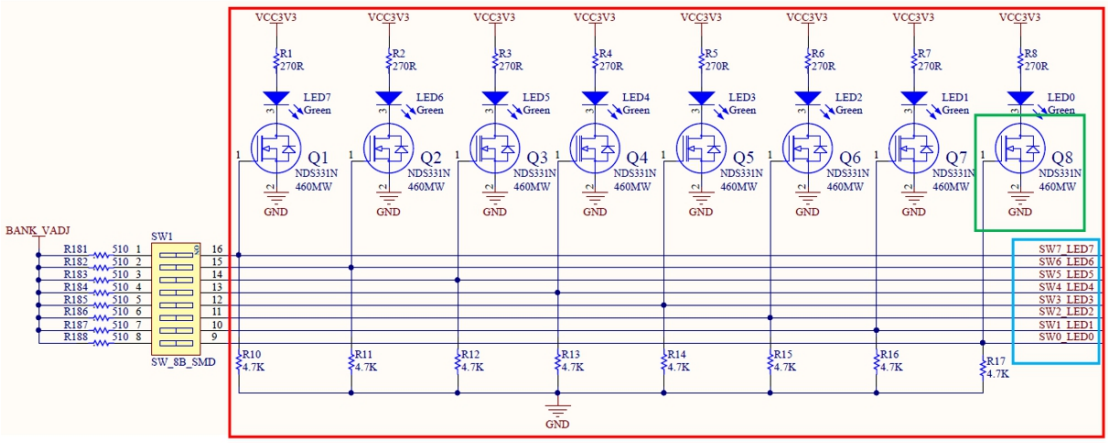


图 1-3　原理图 LED 部分

1.3.3　程序设计

1.3.3.1　系统框图

LED 流水灯系统框图如图 1-4 所示：

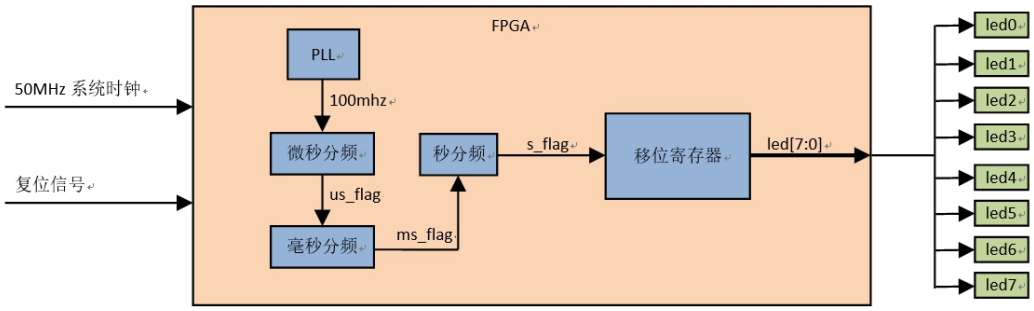


图 1-4　led 流水灯系统框图

1.3.3.2　程序介绍

编写程序之前，首先简单介绍一下我们使用的开发环境，如何创建一个工程。以 Quartus 18.1 版本综合器为例，具体工程建立步骤如图 1-5 至 1-10 所示。

（1）　如图 1-5，打开 Quartus 后，可以直接点击屏幕中间的 New Project Wizard 新建工程，也可以在工具栏里点击 File 后新建工程，或者快捷键 Ctrl+N 后按提示新建工程。

（2）　如图 1-6 所示，选择正确的工程路径，工程命名为 LED_shifting，建议路径简单易查找，方便以后的查看与调用。

（3）　如图 1-7 所示，可直接添加一些提前写好的文件，由于是新建工程，因此直接点击 Next 执行下一步操作。

（4）　如图 1-8 所示，选择正确的 FPGA 芯片型号，板载芯片型号为 10CL006YE144C8G 或

10CL010YE144C8G，在 Family 处选择 Cyclone 10 LP，Pin count 处选择 144，Core speed grade 处选择 8，可帮我们缩小选择范围，快速找到目标型号。

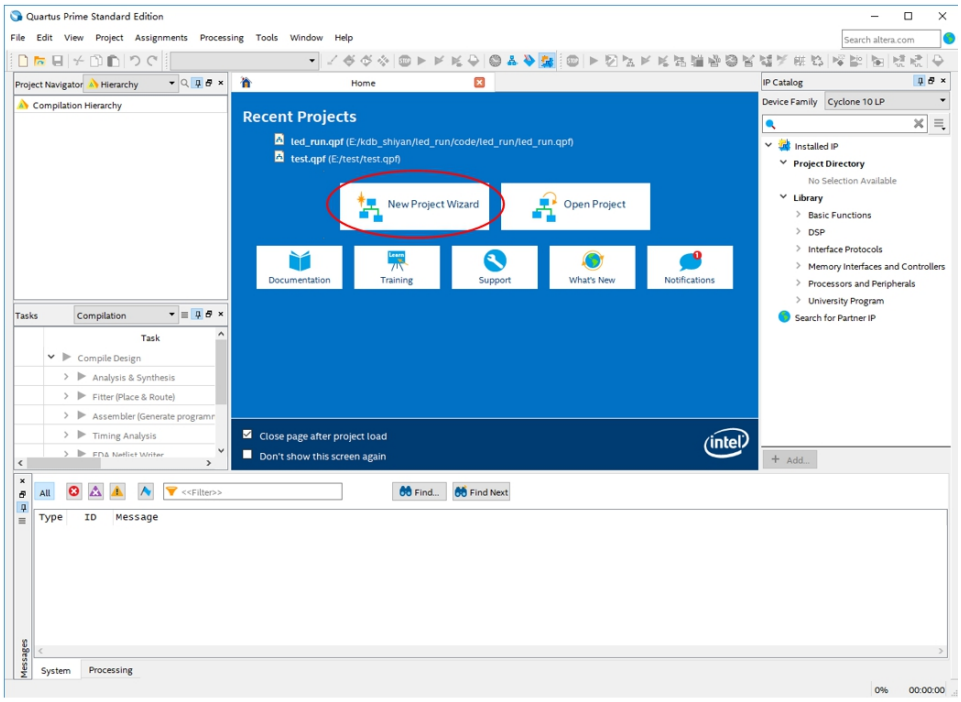


图 1-5　启动后的 Quartus II 18.1 界面

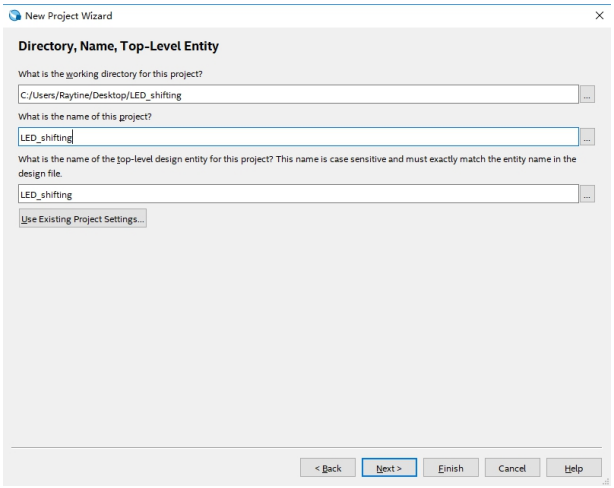


图 1-6　工程文件的路径选择及命名

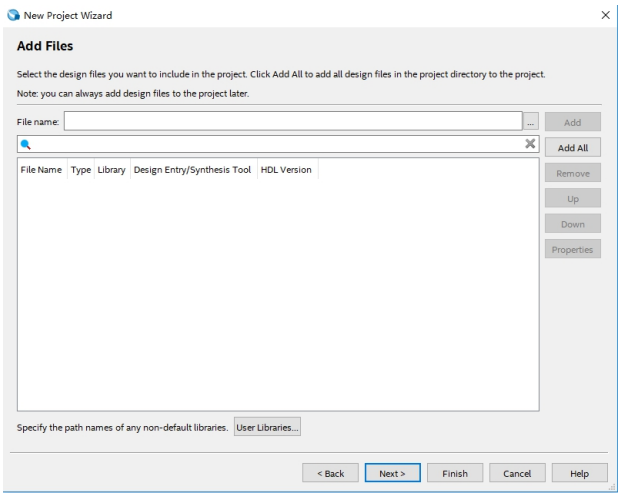


图 1-7　添加文件

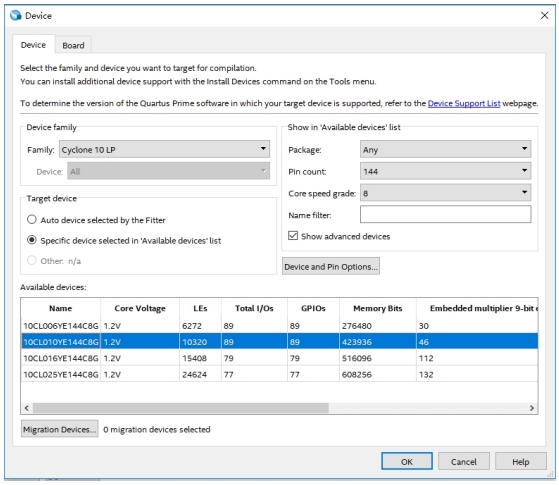


图 1-8　器件选择

（5）　如图 1-9 所示，进行 EDA 工具的选择，在这里暂时不做任何操作，直接使用 Quartus II 自带的 EDA 工具即可

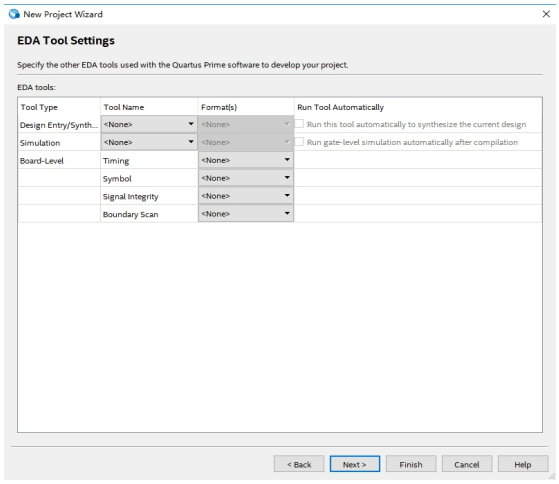


图 1-9　EDA 工具选择

（6）　点击 Next 进入下一个界面后选择 Finish 完成项目工程的建立。

（7）　点击 File→New 或使用快捷键 Ctrl+N，弹出如图 1-10 所示对话框，建立一个程序文件（注意：Verilog HDL File），进行代码的书写。应该注意程序名和工程名的统一性，并保存在正确的路径（文件夹）下。

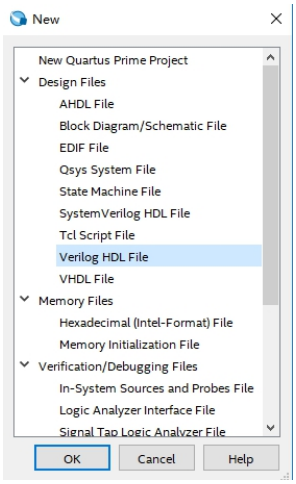


图 1-10　建立程序文件（LED_shifting.v）

准备工作就绪后，开始编写程序，具体介绍如下。

第一步：主程序框架的建立（接口设计）

```
module Led_shifting(
    input                   rst,
    input                   clk,
    output    reg   [7:0]   led
);
endmodule
```

本次实验的输入信号有 50MHz 的系统时钟 clk，复位信号 rst，以及输出信号 led；8 个 led 灯采用 led [7:0] 的多位宽形式定义，减小代码量，方便代码编写。

第二步：IP Core 的调用，PLL 模块的建立与使用

（1） 如图 1-11 所示，在主界面右边 IP catalog 选项栏中，找到 ALTPLL，

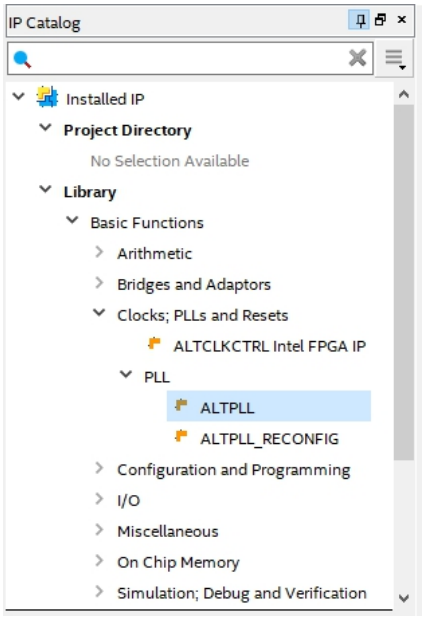


图 1-11 IP 目录

（2） 如图 1-12 所示，双击 ALTPLL，在弹出的对话框中输入 PLL 模块的名称。这里给出名称为 PLL1。注意选择类型为 Verilog 语言类型。

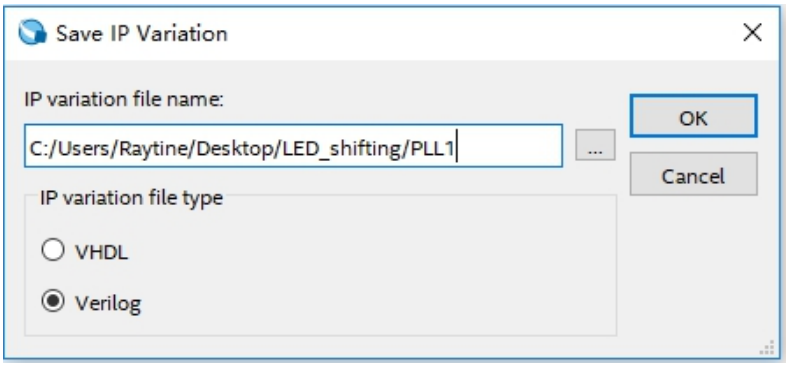


图 1-12 PLL 的命名与保存

（3） 如图 1-13 所示，完成上一步操作后，进入详细的设置界面。Inclk0 为 PLL 的输入时钟，由开发板提供，应与系统时钟保持一致，设为 50MHz；PLL 反馈路径设置为正常模

式（In normal mode）这里涉及到的高级特性，请使用阅读帮助文件；PLL 补偿的输出时钟为 C0；设置完成后点击 Next 进行下一步设置。

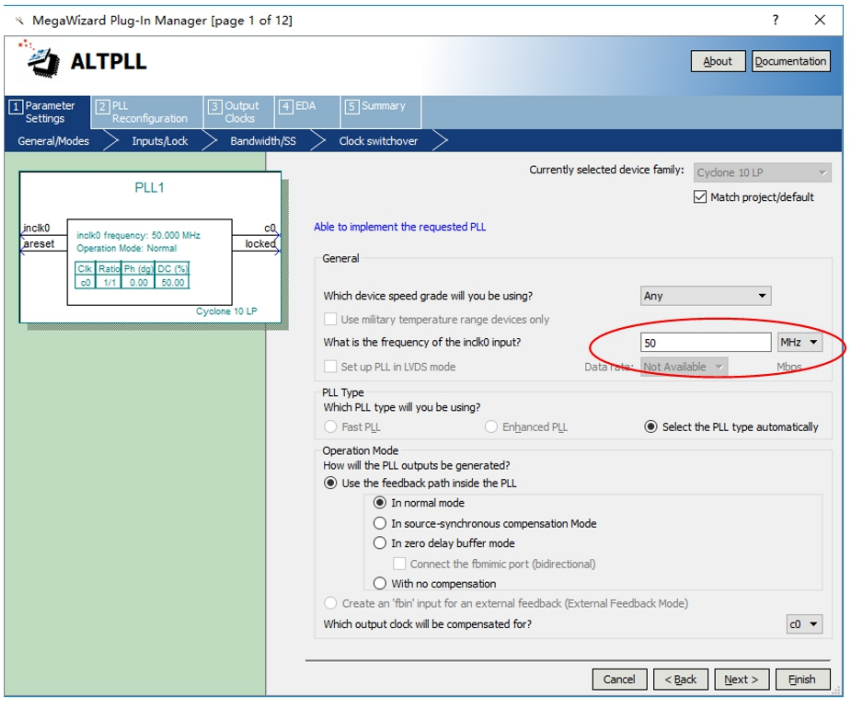


图 1-13 PLL 设置 1 （输入时钟设置）

（4） 如图 1-14 所示，是 PLL 异步复位（areset)控制和捕获锁定（LOCKED)状态的设置，本次实验按照图中默认方式选择即可。

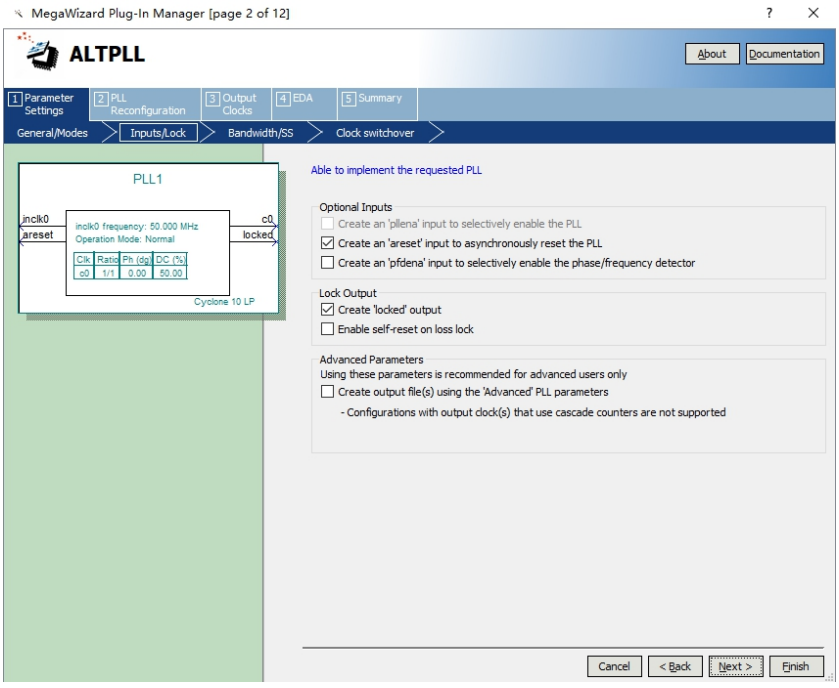


图 1-14 PLL 设置 2

（5） 接下来的三个设置页面里的内容均按默认设置执行即可。

（6） 如图 1-15 所示，是对 PLL 输出时钟 Output Clocks 的设置，一共能够输出 5 个不同的时钟 clk c0～clk c4，本次实验只用到一个，对 clk c0 进行设置，其它默认不使用即可。设置输出频率为 100Mhz，相位偏移（clock phase shift）为 0，占空比（clock duty cycle）为 50%，

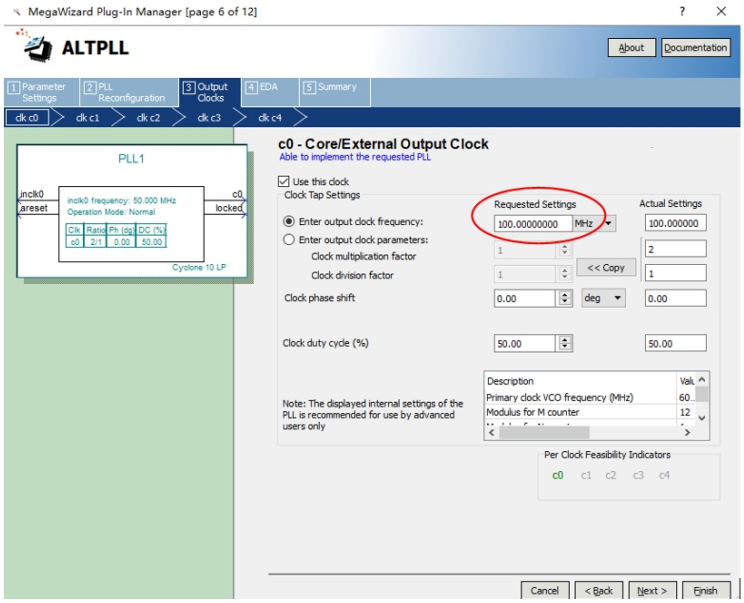


图 1-15　PLL 设置 3（输出时钟设置）

（7）　EDA 设置默认即可。

（8）　如图 1-16 所示，输出文件类型设置选择.bsf（后续采用图形符号设计时会用到）文件和.v 文件，其他按默认值设置，点击 Finish 完成设置。

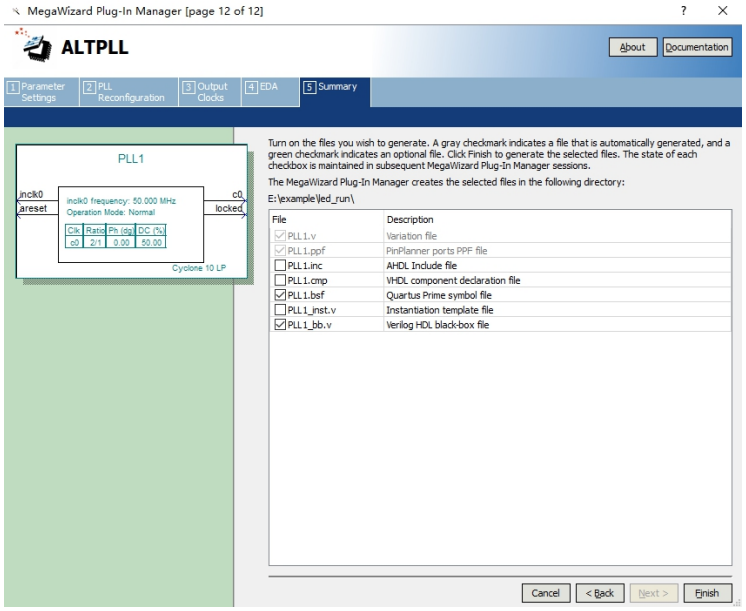


图 1-16　PLL 设置 4 输出文件类型设置

（9）　如图 1-17 所示，在项目界面的 Project Navigator 的类型框中选择 file（默认为项目层次 Hierarchy）

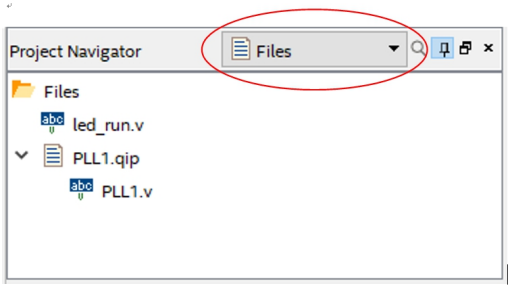


图 1-17　PLL1.v 文件位置

（10）　如图 1-18 所示，点击 PLL1.v，主窗口会显示出 PLL 的内容，找到模块名和端口列表，复制到顶层，进行实例化。

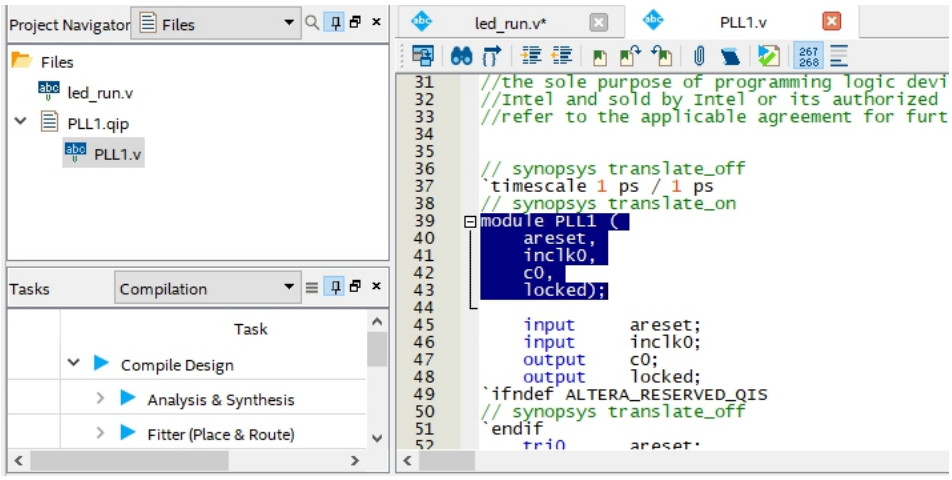


图 1-18　PLL1.v 文件内容

系统上电，pll_locked 信号在 PLL 锁定（稳定工作）前的值为 0，PLL 锁定后 pll_locked 拉高，正常输出时钟信号 sys_clk。锁相环例化如下：

```
wire            sys_clk;
wire            pll_locked;
PLL1 PLL1_inst
(
    .areset             (1'b0),
    .inclk0             (clk),
    .c0                 (sys_clk),
    .locked             (pll_locked)
)
```

（11）　sys_rst 作为分频部分的复位信号，ext_rst 作为流水灯部分的复位信号，在时钟 sys_clk 的沿驱动下，经过一级寄存器同步复位。

```
reg             sys_rst;
reg             ext_rst;
always @ (posedge sys_clk)
begin
    sys_rst <= !pll_locked;
    ext_rst <= rst;
end
```

第三步：分频器的设计

我们采用经 PLL 输出的 100MHz 时钟作为系统时钟，实验要求流水灯的移动速度为 1 秒钟，采用分频设计，首先经过微妙分频得到 1us，然后进过毫秒分频得到 1ms，最后经过秒分频得到 1s 钟。

（1）　微秒分频

```
reg     [6:0]   us_cnt;
```

```
reg                 us_f;
always @ (posedge sys_clk)
begin
    if (sys_rst) begin
        us_cnt <= 0;
        us_f <= 1'b0;
    end
    else begin
        us_f <= 1'b0;
        if (us_cnt == 99) begin
            us_cnt <= 0;
            us_f <= 1'b1;
        end
        else
            us_cnt <= us_cnt + 1'b1;
    end
end
```

100MHz 时钟一个周期是 10ns，1us 需要 100 个时钟周期，即 100 个 10ns，因此定义一个微秒计数器 us_cnt [6:0]，一个微秒脉冲信号 us_f。复位时计数器清零，在每一个时钟上升沿，计数器加一，当计数器等于 99 时，经过了 1us 时间，此时将微秒脉冲信号 us_f 拉高。这样，每经过 1us，这个模块就会产生一个脉冲信号。

（2） 毫秒分频

同理 1ms 等于 1000 个 1us，因此定义一个毫秒计数器 ms_cnt [9:0]，一个微秒脉冲信号 ms_f。

```
reg     [9:0]   ms_cnt;
reg             ms_f;
always @ (posedge sys_clk)
begin
    if (sys_rst) begin
        ms_cnt <= 0;
        ms_f <= 1'b0;
    end
    else begin
        ms_f <= 1'b0;
        if (us_f) begin
            if (ms_cnt == 999) begin
                ms_cnt <= 0;
                ms_f <= 1'b1;
            end
            else
                ms_cnt <= ms_cnt + 1'b1;
        end
    end
```

```
end
```

（3） 秒分频

同理 1s 等于 1000 个 1ms，因此定义一个秒计数器 s_cnt [9:0]，一个秒脉冲信号 s_f。当三个计数器同时计满，此时时间进过 1s 钟，发出秒脉冲信号。

```
reg      [9:0]   s_cnt;
reg              s_f;
always @ (posedge sys_clk) begin
    if (sys_rst) begin
        s_cnt <= 0;
        s_f <= 1'b0;
    end
    else begin
        s_f <= 1'b0;
        if (ms_f) begin
            if (s_cnt == 999) begin
                s_cnt <=0;
                s_f <= 1'b1;
            end
            else
                s_cnt <= s_cnt + 1'b1;
            end
    end
end
```

第四步：流水灯设计

复位时，8 个 LED 灯全亮，所以输出 led 的值为 8’hff，接下来 LED 灯要流动起来，所以首先点亮最低位 LED 灯，此时 led 的值为 8’b0000_0001，当秒脉冲信号到来，点亮下一个 LED 灯，此时 led 的值为 8’b0000_0010，由此可见，只要将“1”高电平循环向左移即可，采用位拼接的形式实现，即 led <= {led[6:0], led[7]}。

```
always @ (posedge sys_clk)
begin
    if (ext_rst)
        led <= 8'hff;
    else begin
        if (led == 8'hff)
            led <= 8'b0000_0001;
        else if (s_f)
            led <= {led[6:0], led[7]};
    end
end
```

1.4 实验验证

1.4.1 下板验证前的准备工作



图 1-19 部分功能介绍

如图 1-19 所示，程序编写完成之后需要进行分析综合，检查对错。点击分析综合图标即可完成，或者用快捷键 Ctrl+K 即可，管脚分配是是将各个信号与 FPGA 管脚相绑定，全编译是将整个工程生成下板时使用的编程文件，并再次检查错误，点击下板图标，按提示将工程下板。点击分析综合图标，完成后 Quartus 会自动生成一个报告，如图 1-20 所示，报告具体内容这里不予详细介绍。

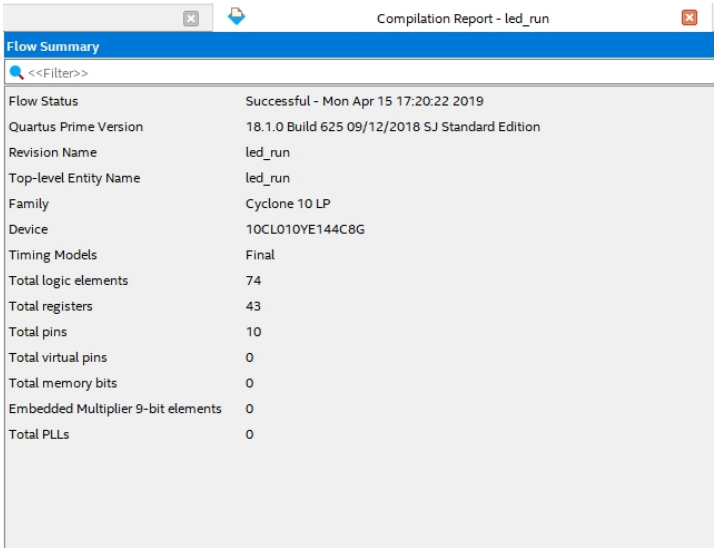


图 1-20 编译报告

检查修改至没有错误，即可开始下板验证，在此之前，需要进行管脚分配，管脚具体分配如图表 1-1 所示。

表 1-1 LED 灯实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
clk	C10_50MCLK	91	输入时钟
rst	KEY3	10	复位按键
led[7]	SW7_LED7	77	Led 灯第 7 位
led[6]	SW6_LED6	76	Led 灯第 6 位
led[5]	SW5_LED5	75	Led 灯第 5 位
led[4]	SW4_LED4	74	Led 灯第 4 位
led[3]	SW3_LED3	87	Led 灯第 3 位
led[2]	SW2_LED2	86	Led 灯第 2 位
led[1]	SW1_LED1	83	Led 灯第 1 位
led[0]	SW0_LED0	80	Led 灯第 0 位

点击管脚分配图标，打开管脚分配窗口，如图 1-21 所示。双击每个管脚对应的 location

栏，直接输入对应管脚编号回车确定即可，也可以点击下拉键找到对应的管脚，但相对较慢。需要注意的是，图 1-21 中 I/O standard 栏，这里显示的内容是每个 I/O 口的电压标准，通过原理图中的 BANK 电压以及设计要求进行确定，本次实验中的 I/O 电压应该选择为 3.3V，双击 I/O standard 栏，点击下拉键，如图 1-22 所示，选择我们需要的电压标准即可。也可以在选择芯片型号的时候提前设置好默认电压标准，点击图 1-8 中 Device and Pin Options→Voltage→Default I/O standard 进行设置。

管脚分配完成，如图 1-23 所示。然后点击全编译，通过后，即可进行下板验证。

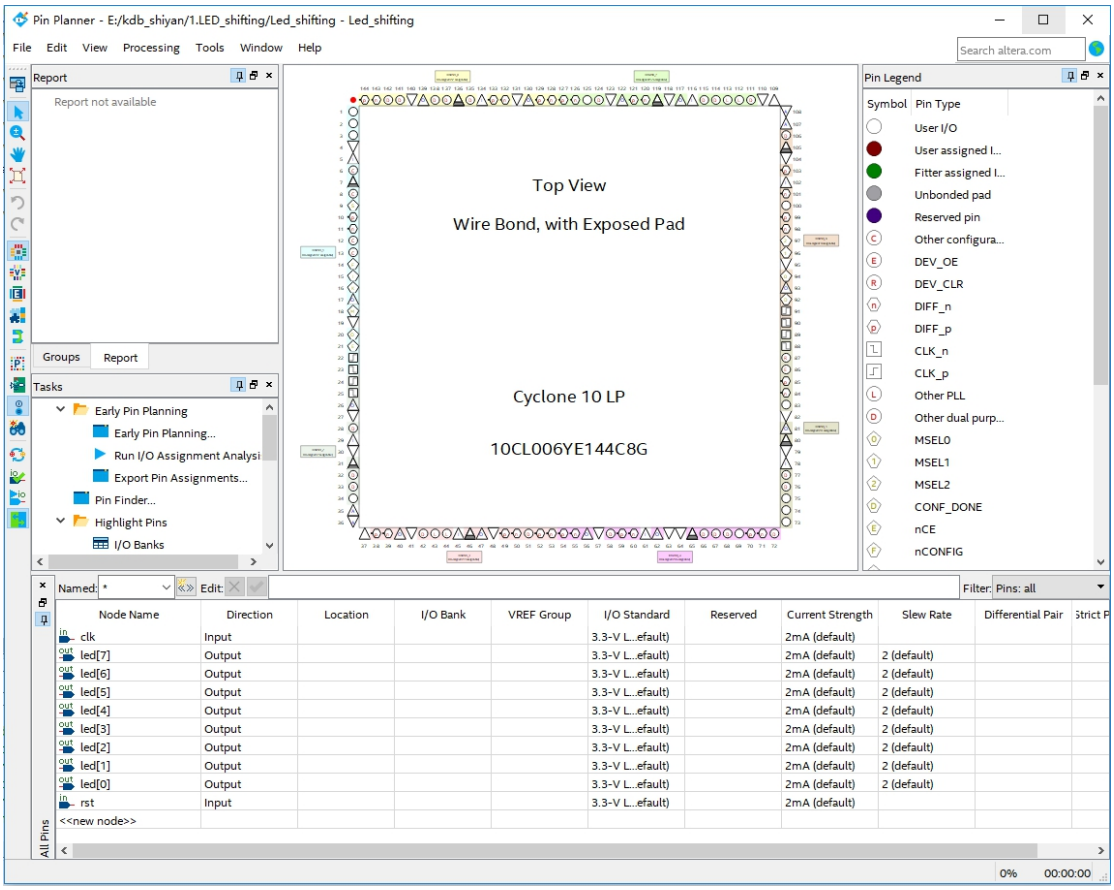


图 1-21　管脚分配窗口

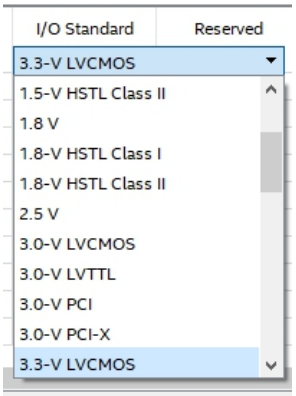


图 1-22　I/O 电压选择

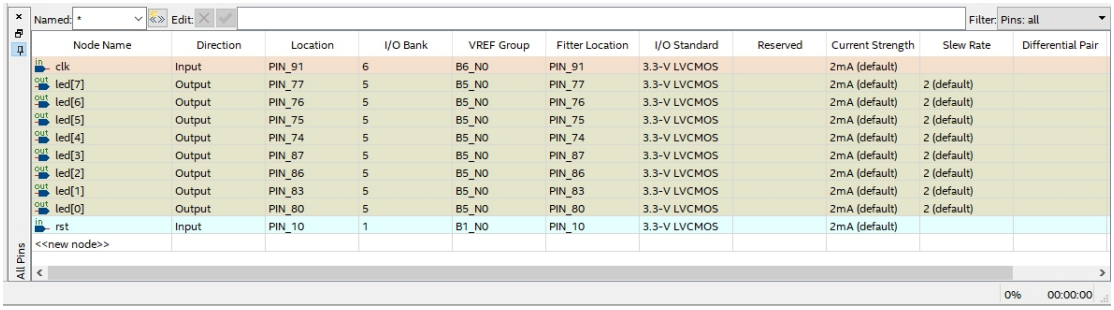


图 1-23　完成管脚分配

1.4.2　下板验证方法及步骤

下板验证之前，需要对 Quartus 进行一些设置，具体请参考附录文件夹中的“JTAG 下载器使用说明-ALTERA”，按照说明设置按成后，点击下载键打开下载窗口，如图 1-24 所示。

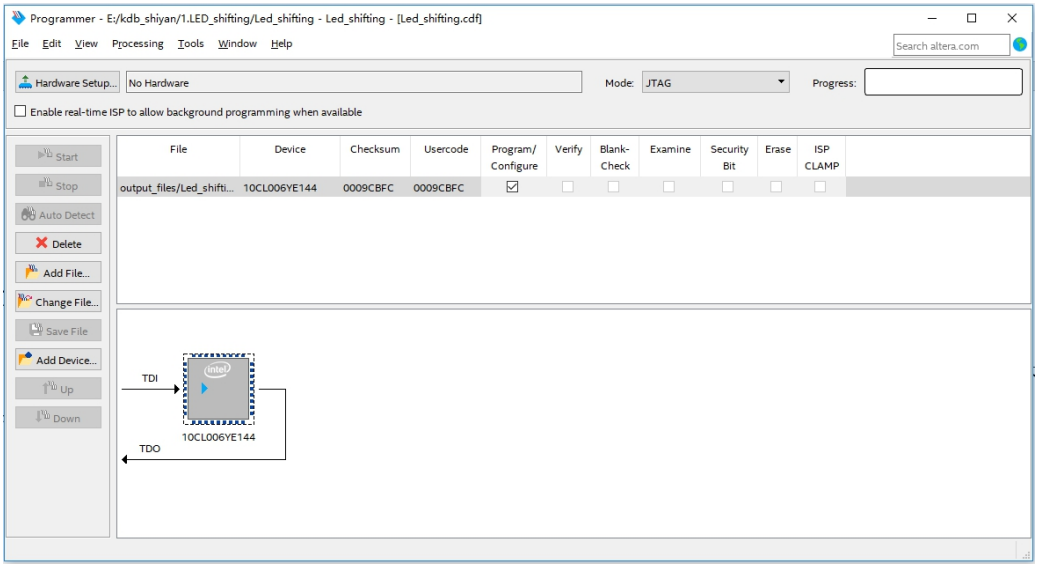


图 1-24　下载窗口

将开发板与主机连接后，点击 Hardware Setup，选择我们的开发板，如图 1-25 所示。

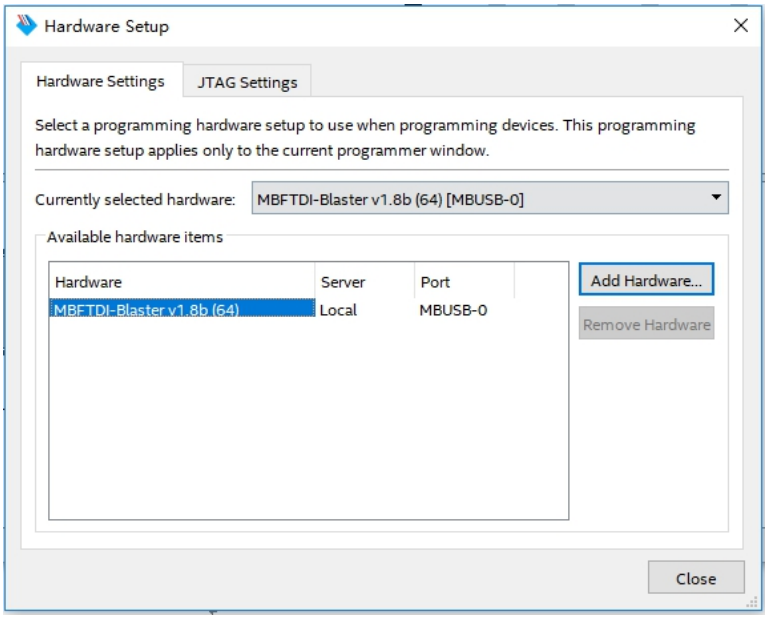


图 1-25　硬件选择

点击 Start 开始下载，如图 1-26 所示，Progress 显示 100%（Successful），即下载完成。

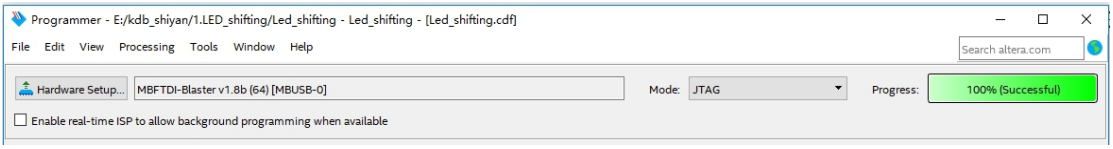


图 1-26　下载成功

如图 1-27 所示，LED 灯从低到高循环点亮，间隔一秒钟。

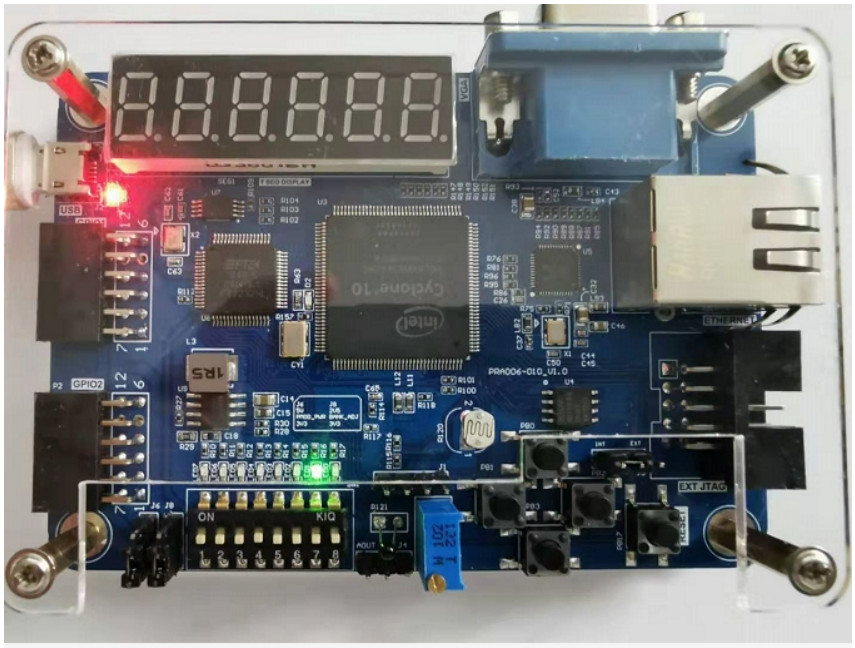


图 1-27　流水灯实验现象

实验二　SignalTap 实验

2.1　实验目的

（1）　继续练习开发板硬件的使用；
（2）　练习 Quartus 中 SignalTap Logic Analyzer 的使用；
（3）　学会对捕捉到的信号进行分析。

2.2　实验要求

通过 SignalTap 的使用，对开发板山的开关信号进行捕捉并分析。

2.3　实验内容

2.3.1　开关简介和 SignalTap 简介

（1）　开关简介

板载开关为 8 位拨码开关，如图 2-1 所示，也叫 DIP 开关，地址开关等，是一款用来操作控制的地址开关，采用的是 0/1 的二进制编码原理。

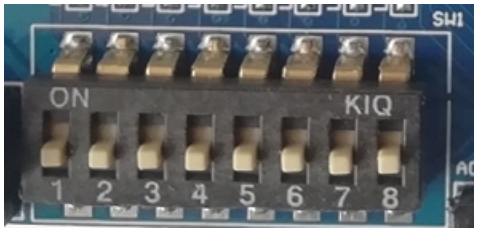


图 2-1　拨码开关实物图

（2）　SignalTap 简介

SignalTap，即逻辑分析仪，使用嵌入式逻辑分析器将信号数据送往 SignalTap 中，当系统正常工作时，对内部节点信号和 I/O 引脚进行实时分析

2.3.2　硬件设计

开关部分原理图如图 2-2 所示（红色框），八位开关一端同时与 BANK_VADJ 端相连，为高电平，一端分别连接一个 LED 灯，并与相连 LED 灯的控制端 SW_LED 端相连。当拨码开关选通时，高电平导通场效应管，LED 灯亮，并且同时向 FPGA 输入一个高电平信号。

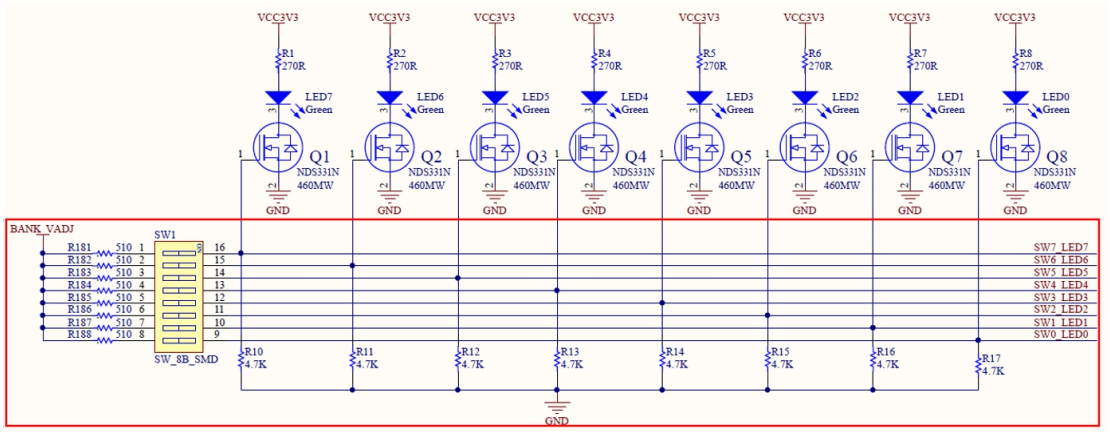


图 2-2　原理图开关部分

2.3.3　程序设计

第一步：主程序框架的建立（接口设计）

```
module SW_LED(
    input              clk,
    input       [7:0]  sw,
    output reg  [7:0]  led
);
endmodule
```

实验输入信号有系统时钟 clk，频率 50Mhz，8 位开关 SW，高有效，以及输出 8 位 LED 灯。

2.4　SignalTap Logic Analyzer 的使用及实验验证

第一步：管脚分配

管脚分配如表 2-1 所示。

表 2-1　开关试验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
clk	C10_50MCLK	91	输入时钟
rst	KEY3	10	复位按键
SW[7]	SW7_LED7	77	拨码开关第 7 位
SW[6]	SW6_LED6	76	拨码开关第 6 位
SW[5]	SW5_LED5	75	拨码开关第 5 位
SW[4]	SW4_LED4	74	拨码开关第 4 位
SW[3]	SW3_LED3	87	拨码开关第 3 位
SW[2]	SW2_LED2	86	拨码开关第 2 位
SW[1]	SW1_LED1	83	拨码开关第 1 位
SW[0]	SW0_LED0	80	拨码开关第 0 位

第二步：　SignalTap II 启动及基本设置

菜单 Tools→SignalTap II logic Analyzer，

（1） 在图 2-3 中选择 Setup(Data,Setup)进入设置界面

（2） 设置 JTAG 调试器，点击 Setup 设成与下载器一样的类型

（3） 设置器件类型（Scan Chain）

（4） 设置 SOF Manager：如图设置成刚编译生成的*.SOF

（5） 时钟和储存深度设置如图 2-4 所示

点击图 2-4 所示位置添加时钟，如图 2-5，在时钟设置对话框中：Filter 选择 SignalTap ：pre-synthesis → List，选择需要的时钟信号，在 PLL1: PLL1_INST 中选中 c0,移到右边框中。

其它设置按照图 2-2 所列设置即可。（SignalTap II 的高级使用，用户可以阅读帮助文件）

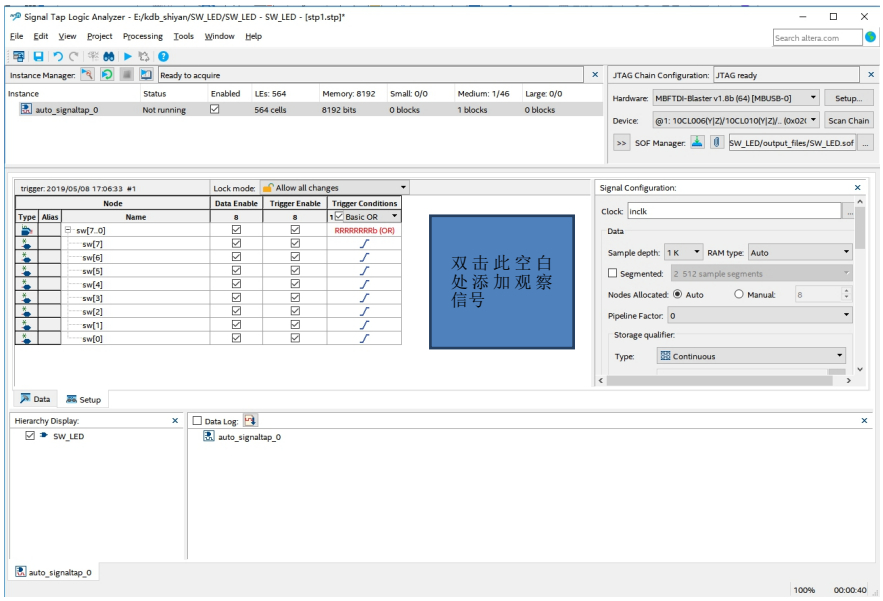


图 2-3 SignalTap 设置界面

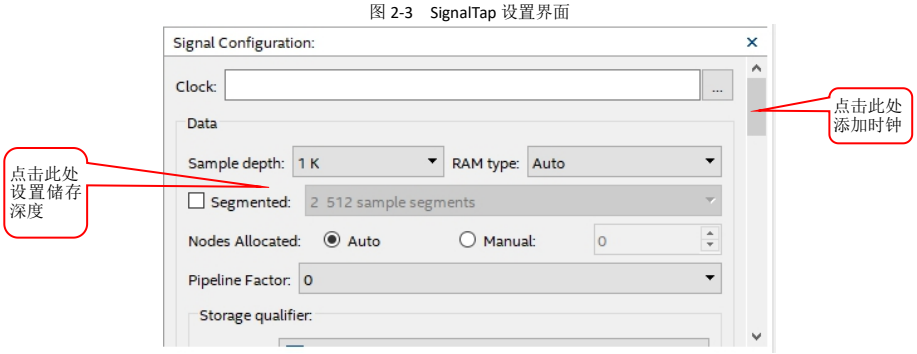


图 2-4 时钟信号及存储深度设置

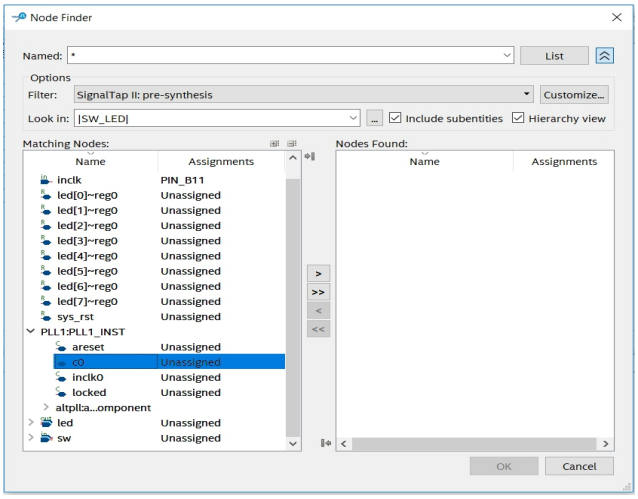


图 2-5 时钟信号选择对话框

第三步：添加观察信号

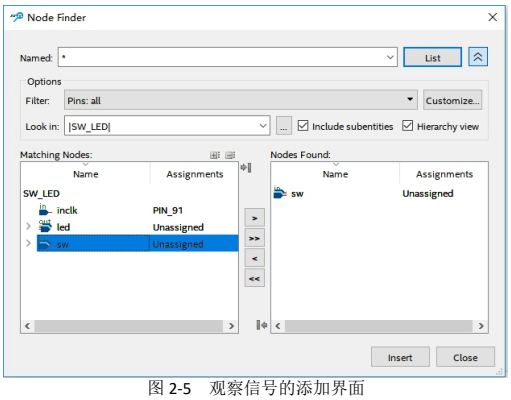


图 2-5　观察信号的添加界面

如图 2-3 所示，双击空白处添加观察信号，添加界面如图 2-6 所示，在左侧选中想要观察的信号，添加至右侧，点击 Insert 即可，添加完成后 Save 保存并重新编译。

第四步：观察信号设置

对于待观察的信号，我们还需要对其进行一些设置，是上升沿触发，还是下降沿触发，还是不关心等情况，需要手动调整，方法如图 2-6 所示。

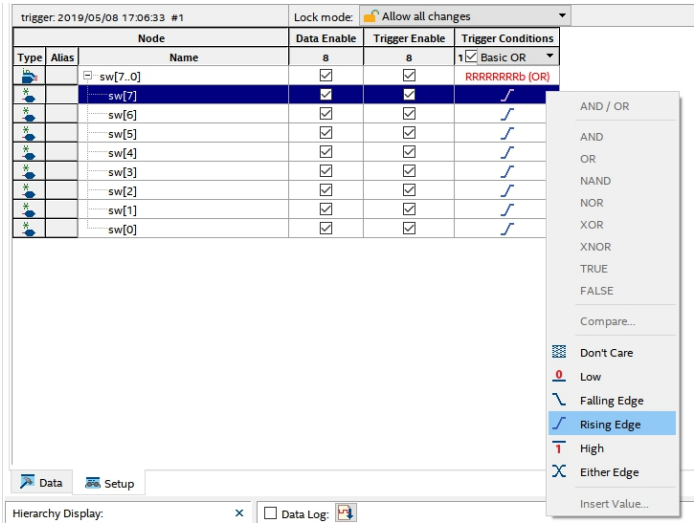


图 2-6　Trigger 信号设置

第五步：观察结果

如图 2-7 所示，点击 Run Analysis,观察 SignalTap 的输出结果。

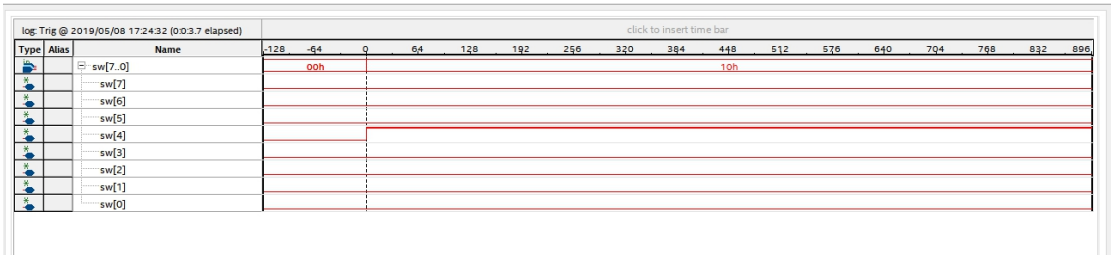


图 2-7　测试结果

经过分析，将开关 SW[4]接通后，其信号为高电平，对应的 LED 灯将点亮。

修改 Trigger 条件，测试不同 Trigger 条件下的输出结果，分析并总结。

实验三　数码管显示

3.1　实验目的

（1）　复习实验一，熟练掌握 PLL 的配置，分频器的设计，以及工程的验证；
（2）　学习 BCD 码计数器；
（3）　数码管的译码显示设计；
（4）　学会将工程烧写到开发板的串行 FLASH 中；

3.2　实验要求

（1）　数码管低两位显示秒，中间两位显示分钟，最高两位显示小时
（2）　小数点保持熄灭状态，暂不予考虑

3.3　实验内容

3.3.1　数码管简介

数码管的一种是半导体发光器件，数码管可分为七段数码管和八段数码管，区别在于八段数码管比七段数码管多一个用于显示小数点的发光二极管单元 DP（decimal point），其基本单元是发光二极管。板载数码管为六位一体的八段数码管如图 3-1 所示，其结构如图 3-2 所示。



图 3-1　板载数码管实物图

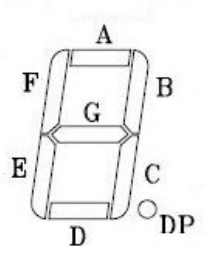


图 3-2　数码管的段结构

我们使用的是共阳极数码管，共阳极顾名思义，就是将发光二极管的阳极连接在一起，如图 3-3 所示。因此需要 FPGA 控制发光二极管阴极为低电平，点亮二极管，并显示对应的信息。六位共阳极八段数码管，把控制哪一位点亮的信号称之为位选信号，每个数码管显示的内容，称之为段选信号，对应真值表如表 3-1 所示。

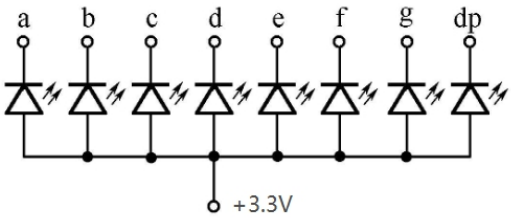


图 3-3　共阳极数码管原理图

表 3-1　共阳极八段数码管段选信号真值表

段选信号	DP	G	F	E	D	C	B	A
•	0	1	1	1	1	1	1	1
0	1	1	0	0	0	0	0	0
1	1	1	1	1	1	0	0	1
2	1	0	1	0	0	1	0	0
3	1	0	1	1	0	0	0	0
4	1	0	0	1	1	0	0	1
5	1	0	0	1	0	0	1	0
6	1	0	0	0	0	0	1	0
7	1	1	1	1	1	0	0	0
8	1	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0
A	1	0	0	0	1	0	0	0
B	1	0	0	0	0	0	1	1
C	1	1	0	0	0	1	1	0
D	1	0	1	0	0	0	0	1
E	1	0	0	0	0	1	1	0
F	1	0	0	0	1	1	1	0

数码管的显示有两种方式，静态显示和动态显示。

静态显示：每个数码管的段选都要连接一个 8 位数据线，控制并保持显示的字形，直到下一个段选信号到来。优点是驱动简单，缺点是占用 I/O 资源过多。

动态显示：将所有数码管的段选线并联在一起，由位选线控制哪一位数码有效并点亮。通过发光二极管的余晖效应和人眼的视觉暂留效应，在一定频率下，数码管看起来是持续点亮的。优点是节省 I/O 资源，缺点是驱动比较复杂，亮度没有静态显示高。

本次实验，采用动态扫描的方式驱动数码管。

3.3.2　硬件设计

数码管原理图如图 3-4 所示，阳极通过 P 沟道型场相应管与 VCC 相连，因此当位选信号 SEG_3V3_D[0:5]为低电平 0 时，场效应管导通，数码管阳极为高电平；阴极（段选信号）SEG_PA，SEG_PB，SEG_PC，SEG_PD，SEG_PE，SEG_PF，SEG_PG，SEG_DPZ 直接与 FPGA 相连，由 FPGA 直接控制，因此，当位选信号为 0，段选信号也为 0 时，数码管被点亮。

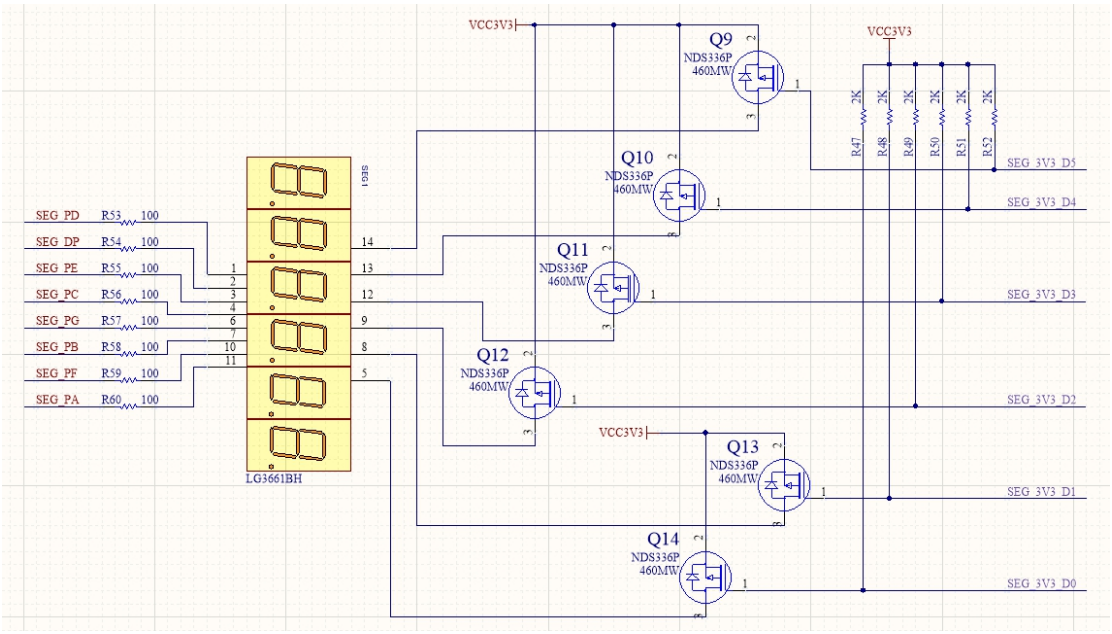


图 3-4　原理图数码管部分

3.3.3　程序设计

3.3.3.1　系统框图

数码管显示系统框图如图 3-5 所示。

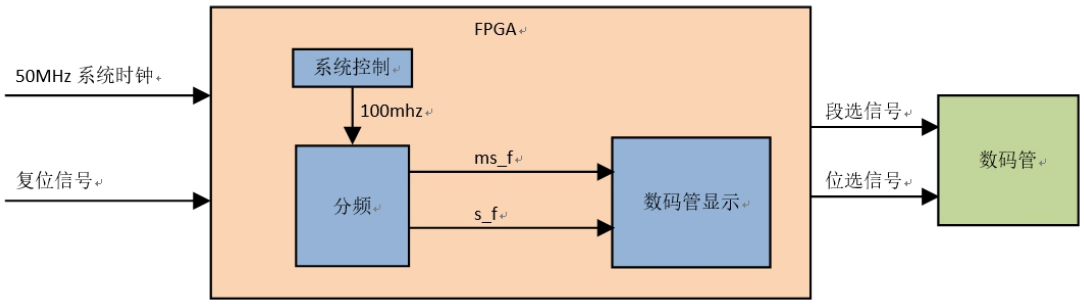


图 3-5　数码管显示系统框图

顶层模块 RTL 视图如图 3-6 所示。

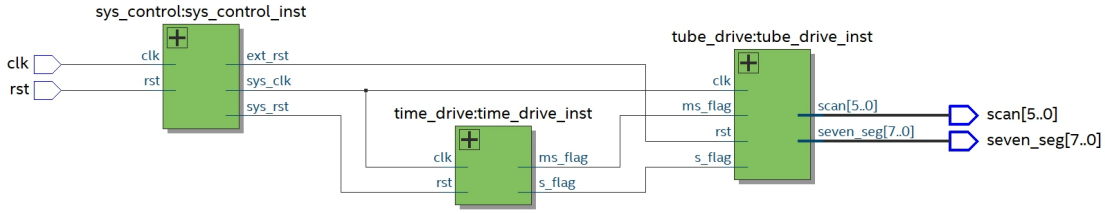


图 3-6　RTL 视图

3.3.3.2　程序介绍

第一步：主程序框架的建立（接口设计）

```
module BCD_counter (
      input                clk,
      input                rst,
      output      [7:0]    seven_seg,
      output      [5:0]    scan
);
endmodule
```

输入信号有时钟和复位信号，输出信号为段选信号 seven_seg 和位选新信号 scan。

第二步：系统控制模块

```
//例化锁相环
PLL  PLL_inst
(
      .areset              (1'b0),
      .inclk0              (clk),
      .c0                  (sys_clk),
      .locked              (locked)
);

//复位信号
always @ (posedge sys_clk)
begin
      sys_rst <= !locked;
      ext_rst <= rst;
end
```

在第一个子模块（系统控制模块）中，输入时钟为系统 50MHz 时钟，经过锁相环出输一个 100MHz，作为其他子模块的工作时钟，同时将锁相环锁定信号取反作为系统复位信号，经按键复位寄存一拍作为外部硬件复位信号。

第三步：分频模块

参考实验一，输出一个毫秒脉冲信号和一个秒脉冲信号，作为数码管驱动模块的输入信号。

第四步：数码管驱动模块

（1） 计数部分

计数部分与分频模块类似，通过秒脉冲信号分别计时 60 秒，60 分钟，24 小时，当时间到 23 小时 59 分 59 秒，计数器全部清零，相当于一天时间。

（2） 数码管动态扫描部分

```
reg      [3:0]      count_sel;
reg      [2:0]      scan_state;
always @ (posedge clk)
begin
      if (rst) begin
```

```
            scan            <= 6'b111_111;
            count_sel       <= 4'd0;
            scan_state      <= 0;
        end
        else case (scan_state)
            0    :
            begin
                scan        <= 6'b111_110;
                count_sel<= counta;
                if (ms_f)
                    scan_state <= 1;
            end

            1    :
            begin
                scan        <= 6'b111_101;
                count_sel<= countb;
                if (ms_f)
                    scan_state      <= 2;
            end

            2    :
            begin
                scan        <= 6'b111_011;
                count_sel<= countc;
                if (ms_f)
                    scan_state <= 3;
            end

            3    :
            begin
                scan        <= 6'b110_111;
                count_sel<= countd;
                if (ms_f)
                    scan_state <= 4;
            end

            4    :
            begin
                scan        <= 6'b101_111;
                count_sel<= counte;
                if (ms_f)
                    scan_state <= 5;
            end
```

```
            5    :
            begin
                scan <= 6'b011_111;
                count_sel<= countf;
                if (ms_f)
                    scan_state <= 0;
            end
            default    :    scan_state <= 0;
        endcase
end
```

数码管的动态扫描通过状态机来实现，一共六位数码管，需要六个状态即可，定义状态机 scan_state[2:0]，在不同状态下显示对应的内容 count_sel。复位时，六个数码管全部熄灭，跳转到 0 状态。在毫秒脉冲的驱动下，以 1 毫秒时间动态扫描数码管：

0 状态下点亮第 0 位数码管，并显示秒的个位；
1 状态下点亮第 1 位数码管，并显示秒的十位；
2 状态下点亮第 2 位数码管，并显示分钟的个位；
3 状态下点亮第 3 位数码管，并显示分钟的十位；
4 状态下点亮第 4 位数码管，并显示小时的个位；
5 状态下点亮第 5 位数码管，并显示小时的十位。

第五部分：段码显示部分

```
always @ (*)
begin
    case (count_sel)
        0    :    seven_seg_r <= 7'b100_0000;
        1    :    seven_seg_r <= 7'b111_1001;
        2    :    seven_seg_r <= 7'b010_0100;
        3    :    seven_seg_r <= 7'b011_0000;
        4    :    seven_seg_r <= 7'b001_1001;
        5    :    seven_seg_r <= 7'b001_0010;
        6    :    seven_seg_r <= 7'b000_0010;
        7    :    seven_seg_r <= 7'b111_1000;
        8    :    seven_seg_r <= 7'b000_0000;
        9    :    seven_seg_r <= 7'b001_0000;
        default : seven_seg_r <= 7'b100_0000;
    endcase
end
```

对照表 3-1，将需要显示的字符与段码相对应起来，小数点置高熄灭，然后以拼接的形式组成最终的输出段选信号。

3.4 Flash 的运用及实验验证

第一步：管脚分配

管脚分配如表 3-1 所示

表 3-1　数码管显示实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
clk	C10_50MCLK	91	输入时钟
rst	KEY3	10	复位按键
scan[5]	SEG_3V3_D5	124	位选信号第 5 位
scan[4]	SEG_3V3_D4	127	位选信号第 4 位
scan[3]	SEG_3V3_D3	129	位选信号第 3 位
scan[2]	SEG_3V3_D2	141	位选信号第 2 位
scan[1]	SEG_3V3_D1	142	位选信号第 1 位
scan[0]	SEG_3V3_D0	136	位选信号第 0 位
seven_seg[7]	SEG_PA	128	段选信号第 7 位
seven_seg[6]	SEG_PB	135	段选信号第 6 位
seven_seg[5]	SEG_PC	138	段选信号第 5 位
seven_seg[4]	SEG_PD	126	段选信号第 4 位
seven_seg[3]	SEG_PE	125	段选信号第 3 位
seven_seg[2]	SEG_PF	133	段选信号第 2 位
seven_seg[1]	SEG_PG	137	段选信号第 1 位
seven_seg[0]	SEG_DP	132	段选信号第 0 位

第二步：全编译

第三步：将程序固化到 Flash 上

板载 Flash（N25Q128A）是一个串行 Flash 芯片，可存储 128Mbit 的内容，对于我们学习过程中的工程来说，绰绰有余。Flash 原理图如图 3-7 所示。

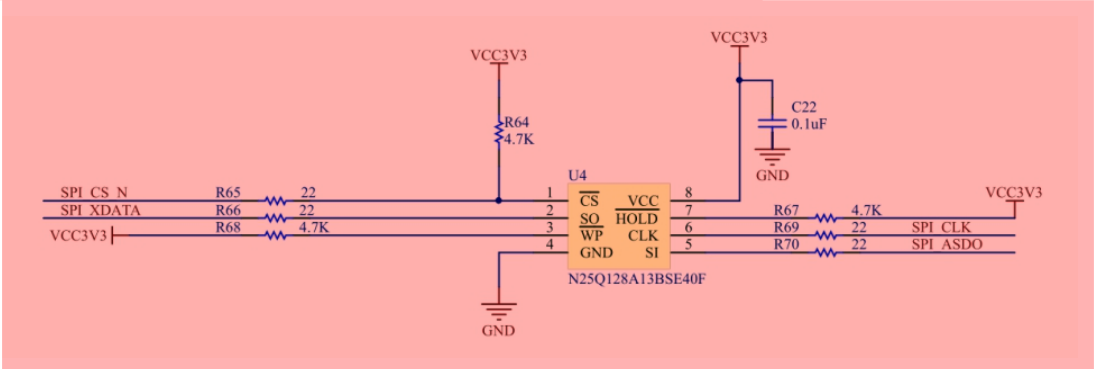


图 3-7　原理图 Flash 部分

Flash 的作用就是将程序烧录保存在开发板上，断电之后程序不消失，开发板下次通电之后可直接使用，实际学习生活过程中更加实用。在 SPI_CLK 时钟的驱动下，FPGA 通过 SPI_ASDO 线将程序烧录到 Flash 中保存起来，重新上电后，FPGA 通过 SPI_XDATA 将程序重新读取到 FPGA 中进行试验。

Flash 具体配置过程如下：

（1）　菜单 File→Convert programming files，如图 3-8；

（2）　选项设置

1)　选择 JTAG Indirect configuration File（.Jic）

2)　Configuration Device:EPCQ 128A（兼容开发板 N25Q128A）

3) Mode:Active serial

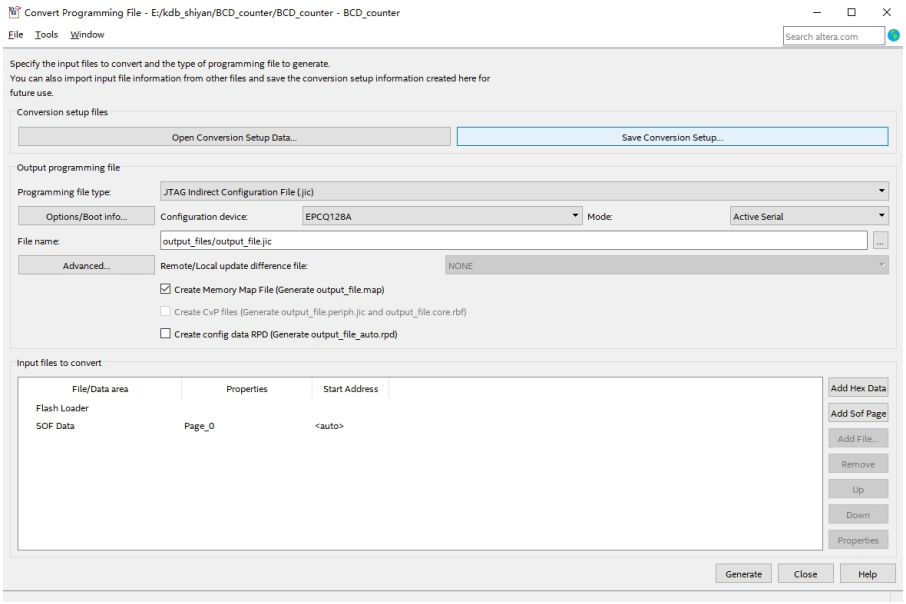


图 3-8 .jic 文件设置

（3） 点击 Advanced 按钮，按图 3-9 进行设置

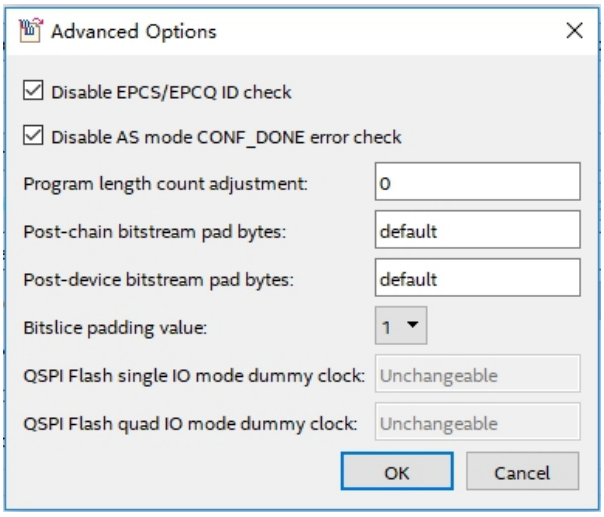


图 3-9 Advanced 选项设置

（4） 添加转换文件，如图 3-10 所示

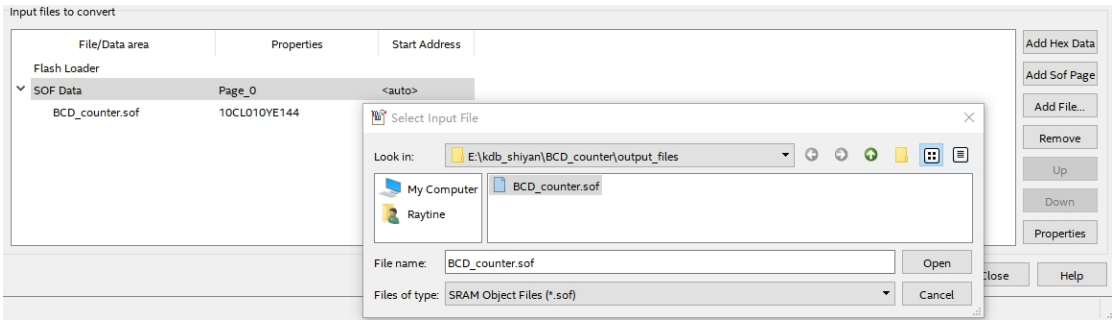


图 3-10 添加转换文件

（5） 添加设备，如图 3-11 所示

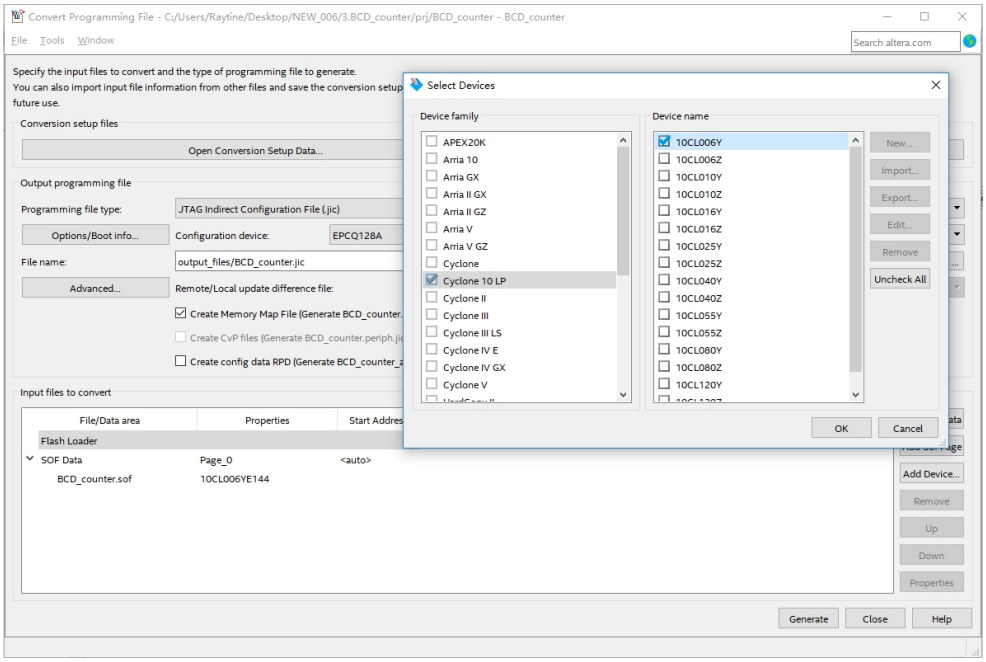


图 3-11　添加设备

（6）　点击 Generate，生成 BCD_counter.jic 文件

（7）　与之前的下板验证操作一致，选择正确的文件（.jic）下板

第四步：上电验证

如图 3-12 所示，重新上电后，FPGA 自动将 Flash 中的程序读入 FPGA 中并运行。

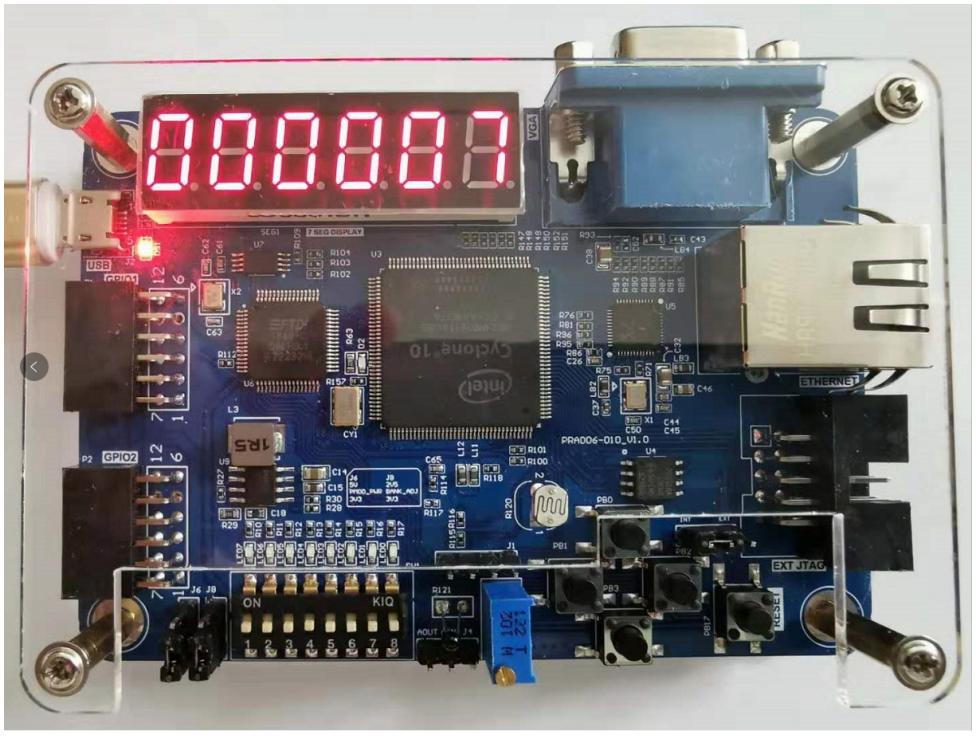


图 3-12　数码管实验现象

实验四　Block/SCH 实验

4.1　实验目的

（1）　复习 Quartus 中新建 FPGA 项目，器件选择，PLL 创建，PLL 频率设定，Verilog 的树形层次结构的设计，SignalTapII 的使用
（2）　掌握图形法从上到下的设计方法
（3）　结合 BCD_counter 项目实现数字钟的显示
（4）　观察实验结果

4.2　实验要求

运用图形设计工程。

4.3　实验内容

本次实验主要是掌握另一种设计方法，其他设计内容与实验三基本一致，不再详细介绍，下面介绍模块化设计方法。

（1）　新建项目 :File→New Project Wizard…
项目名：block_counter
选择器件(10cl010YE144c8G)
（2）　新建文件：File→New，选择 Blcok Diagram/Schematic File，如图 4-1 所示。

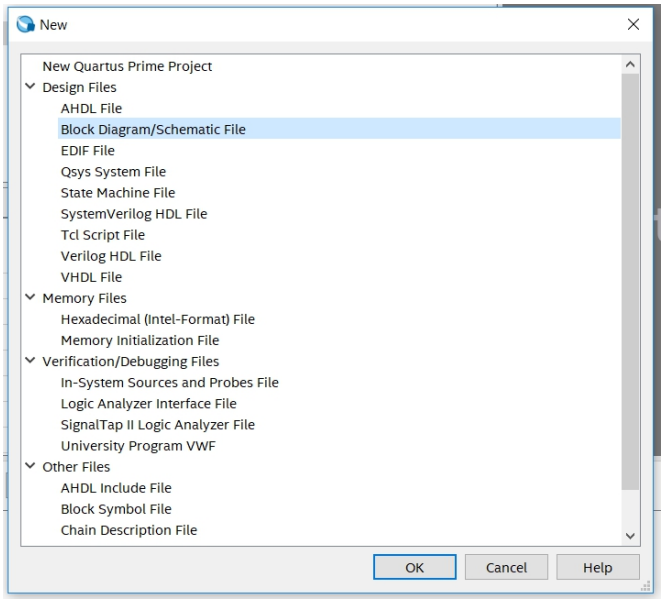


图 4-1　新建文件

（3）　新建如图 4-2 所示，中间部分为图形设计区，可以进行 Block/SCH 设计
1)　存储文件名为 block_counter.bdf
2)　双击图形设计区的空白处，添加符号

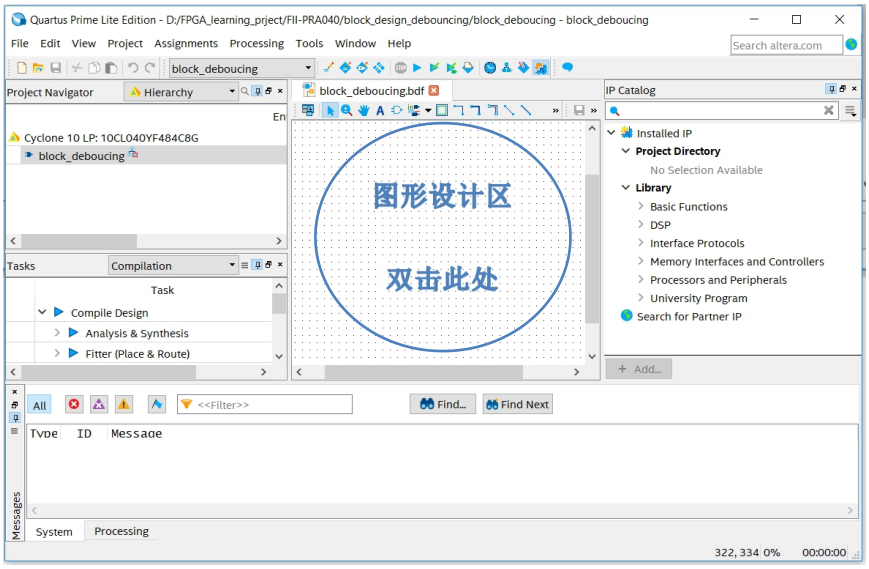


图 4-2　图形设计界面

（4）　图形编辑

双击图形设计区，在 libraries 选择合适的库和器件

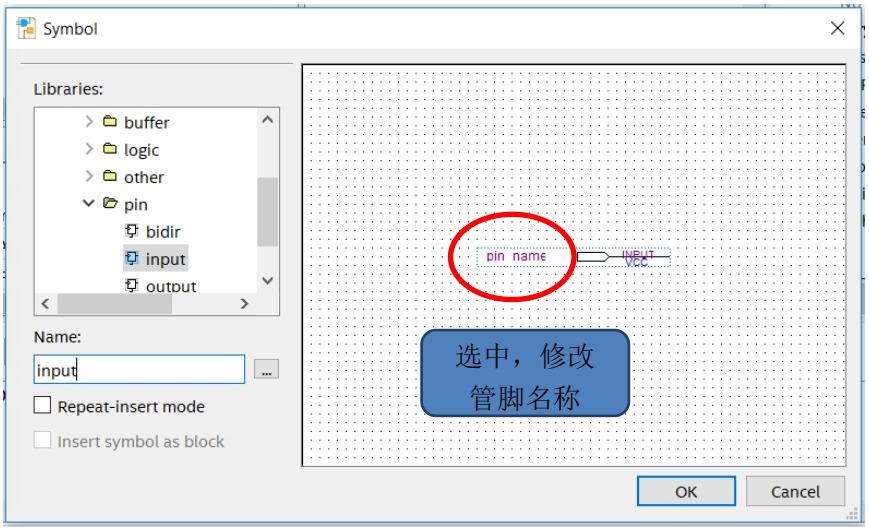


图 4-3 输入图形符号

（5）　添加输入、输出，并修改输入、输出名称

（6）　添加自定义的符号

1) 新建一个 block/sch 文件，存储为 PLL_sys.bdf
2) 添加添加 PLL IP ，参照实验一
3) 选择生成的文件中包含 PLL1.bsf 文件
4) 在 PLL_sys.bdf 文件中的空白区双击，选择刚生成的 PLL1 符号，添加到该文件中，如图 4-4 所示

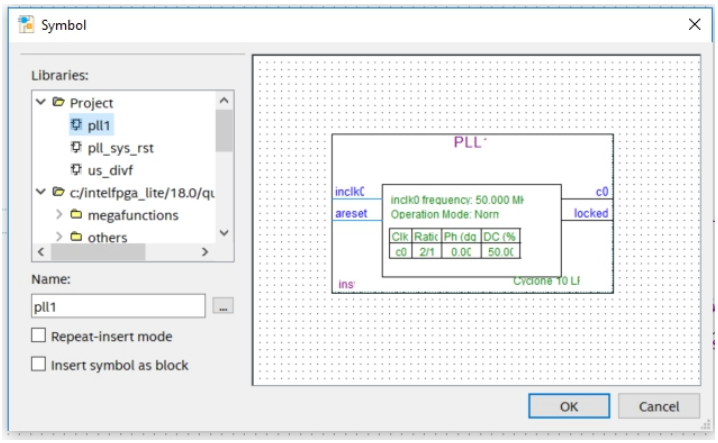


图 4-4　在图形编辑界面中调用 IP catalog 中生成的符号

5)　继续添加其它符号，input,output,dff,GND 等并连线，如图 4-5。

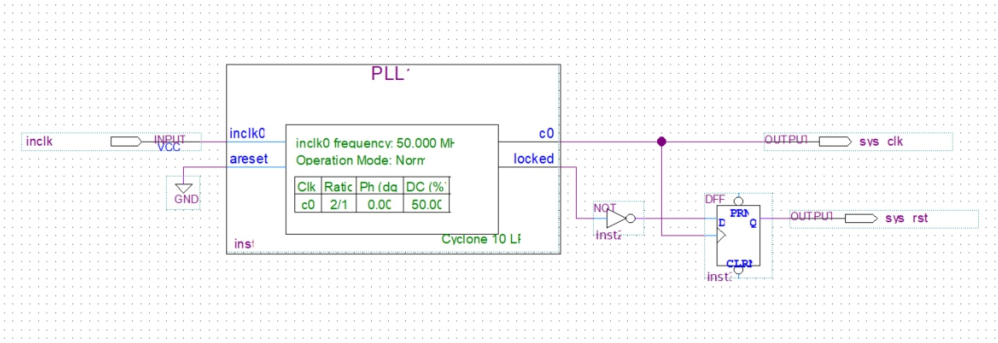


图 4-5　器件连线

（7）　将新建的文件重新创建图形编辑的符号，以便后续设计使用

1)　File→Create/Update→Create Symbol file for Current File，如图 4-6 所示

2)　生成 PLL_sys.bsf

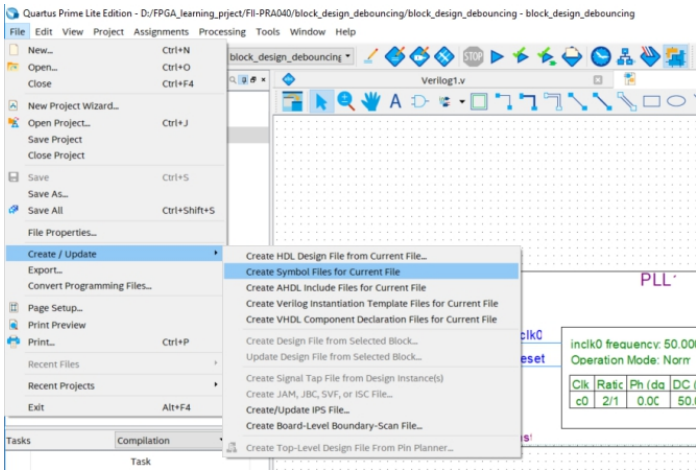


图 4-6　为当前文件创建符号文件（symbol file　*.bsf）

（8）　创建分频模块

1)　新建一个分频器的 verilog 文件 div_us（代码参考附录工程文件）

2)　将 PLL 输出时钟作为自身输入时钟，将 100Mhz 的时钟分频成 1Mhz 的时钟

3)　重复（7），创建 div_us.bsf

4)　新建 1000 分频的 verilog 文件：div_1000f.v

5)　创建 div_1000f.bsf 符号

（9）　创建输出脉冲 us，ms，second 模块，如图 4-7 所示，具体实现参考代码以及实验

一、三的分频设计

1) 新建 block/sch 文件 block_div，在 block_div.bdf 中添加已设计好的图形符号文件

2) 重复（7）创建 block_div.bsf 符号

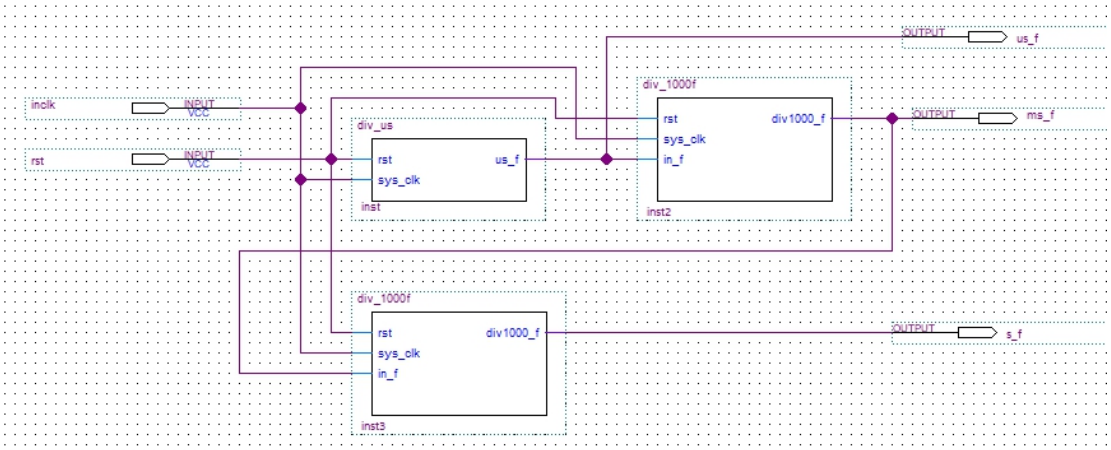


图 4-7 block/sch 设计的 us, ms, second 脉冲

（10） 新建 verilog 文件 bcd_counter.v，设计时、分计数器，并创建 bsf 符号，参考实验三，将其中一部分分频用（9）中 block_div 实现，

（11） 组合各个*.bsf, 完成数字钟的设计（block_counter.bdf），如图 4-8 所示

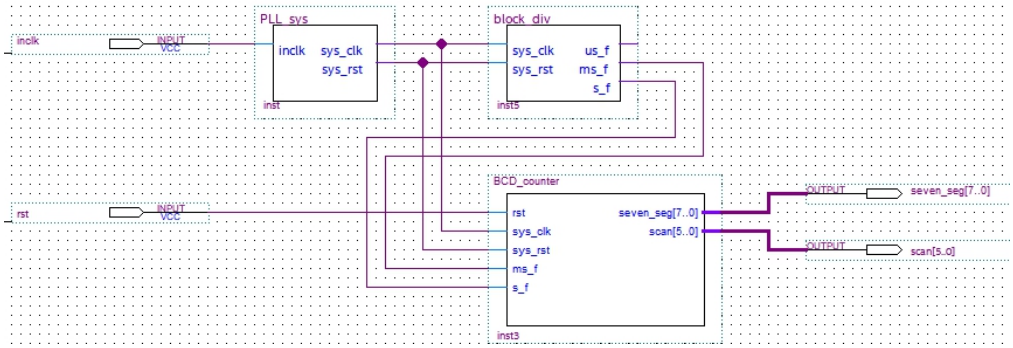


图 4-8 BDF 设计的数字钟

4.4 实验验证

管脚分配，编译，下载验证与实验三一致，可参考实验三，此处省略。

实验五　按键消抖实验

5.1　实验目的

（1）　复习流水灯设计过程
（2）　学习按键去抖原理，以及自适应程序设计
（3）　学习 Fii-PRA006/010 按键硬件电路的连接及使用
（4）　综合应用按键去抖与其它符合程序设计

5.2　实验要求

（1）　通过按键控制流水灯的移动
（2）　每按一次按键，流水灯移动一位
（3）　当按下左移按键是，流水灯向左移动，按下右移按键，流水灯向右移动

5.3　实验内容

5.3.1　按键和消抖原理简介

（1）　按键简介

板载按键为常见的机械按键，按下有效，松开后自动弹起。共五个，分别为 PB0（up），PB1（left），PB2（right），PB3（down）和一个系统复位按键 RESET。如图 5-1 所示。

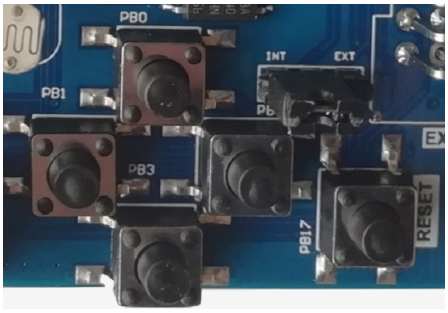


图 5-1　按键实物图

（2）　消抖原理简介

只要用到机械按键，我们就必须考虑他的不稳定性。通常按键所用的都是机械弹性开关，当机械触点断开、闭合时，由于机械触点的弹性作用，一个按键开关在闭合时不会马上就稳定的接通，在断开时也不会一下子彻底断开，而是在闭合和断开的瞬间伴随了一连串的抖动，如图 5-1 所示。

按键稳定闭合时间长短是由操作人员决定的，通常都会在 100ms 以上，刻意快速按的话能达到 40-50ms 左右，很难再低了。抖动时间是由按键的机械特性决定的，一般都会在几毫秒到几十毫秒之间，为了确保程序对按键的一次闭合或者一次断开只响应一次，必须进行按键的消抖处理。当检测到按键状态变化时，不是立即去响应动作，而是先等待闭合或断开稳定后再进行处理。按键消抖可分为硬件消抖和软件消抖。

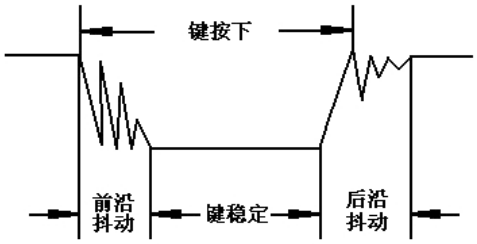


图 5-2　按键抖动原理

在绝大多数情况下，我们是用软件即程序来实现消抖的。最简单的消抖原理，就是当检测到按键状态变化后，先等待一个 10ms 左右的延时时间，让抖动消失后再进行一次按键状态检测，如果与刚才检测到的状态相同，就可以确认按键已经稳定的动作了。这种检测在传统的软件设计中使用比较广泛。但是随着按键使用次数增加，或不同质量的按键其反应也会不同。如果延时过短抖动不能滤除干净，延时过长又影响按键的灵敏度。

5.3.2　硬件设计

按键原理图如图 5-3 所示，可以看到按键一侧（P1,P2）与 VCC 相连，另一侧（P3,P4）与 FPGA 相连，同时通过一个 1K 电阻接地。正常状态下，按键悬空，此时按键 P3 处电位为 0，因此按键向 FPGA 输入值为 0；当按键按下时，按键两侧导通，此时按键 P3 处电位为 VCC3.3V，因此按键向 FPGA 输入值为 1。所以板载开关为高有效。

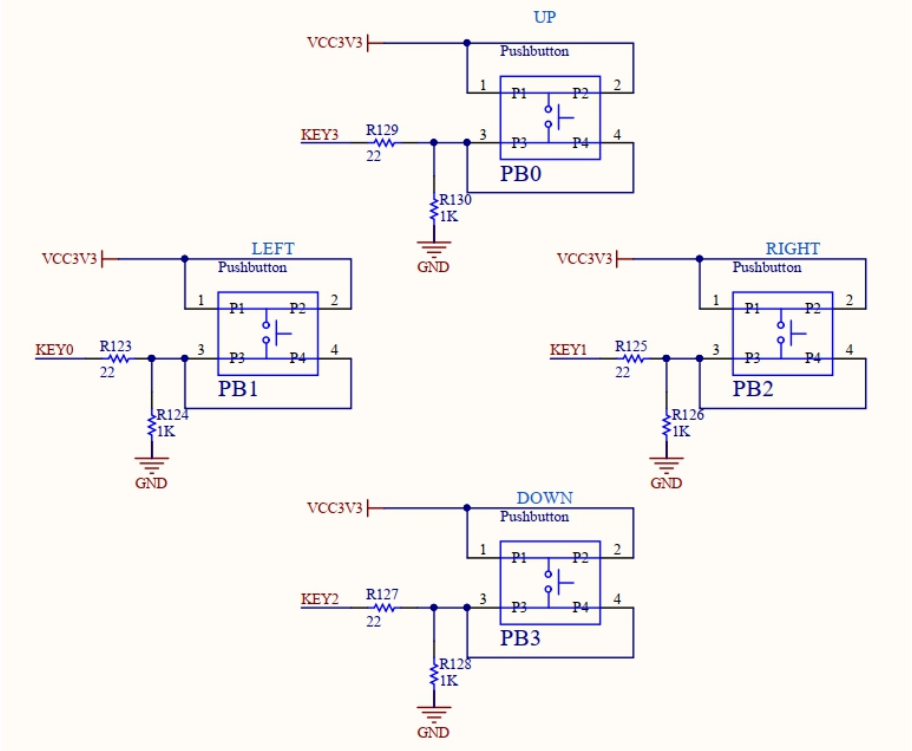


图 5-3　原理图按键部分

5.3.3　程序设计

5.3.3.1　系统框图

按键控制流水灯系统框图如图 5-4 所示。

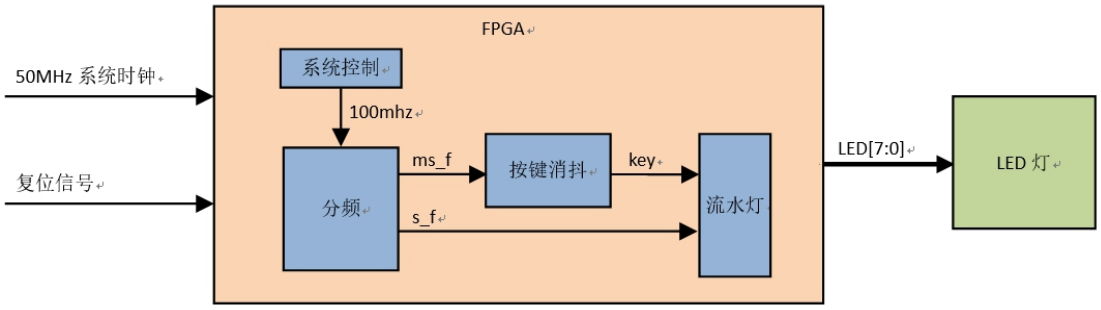


图 5-4　按键控制流水灯系统框图

顶层模块 RTL 视图如图 5-5 所示。

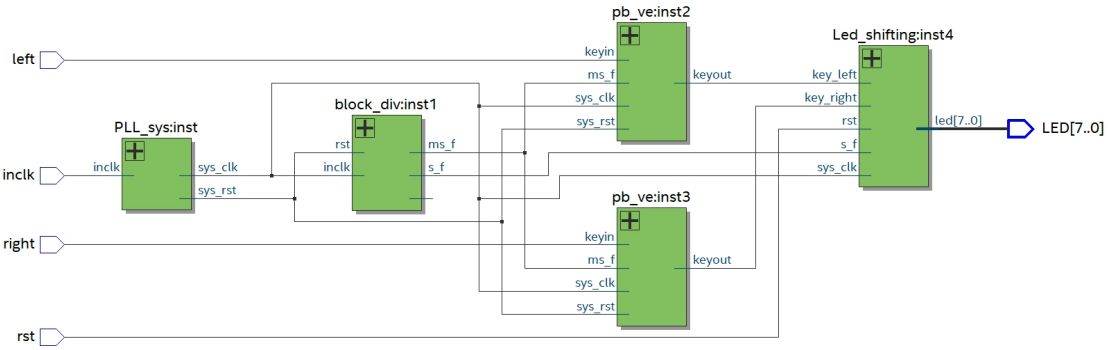


图 5-5　RTL 视图

顶层设计如图 5-6 所示。

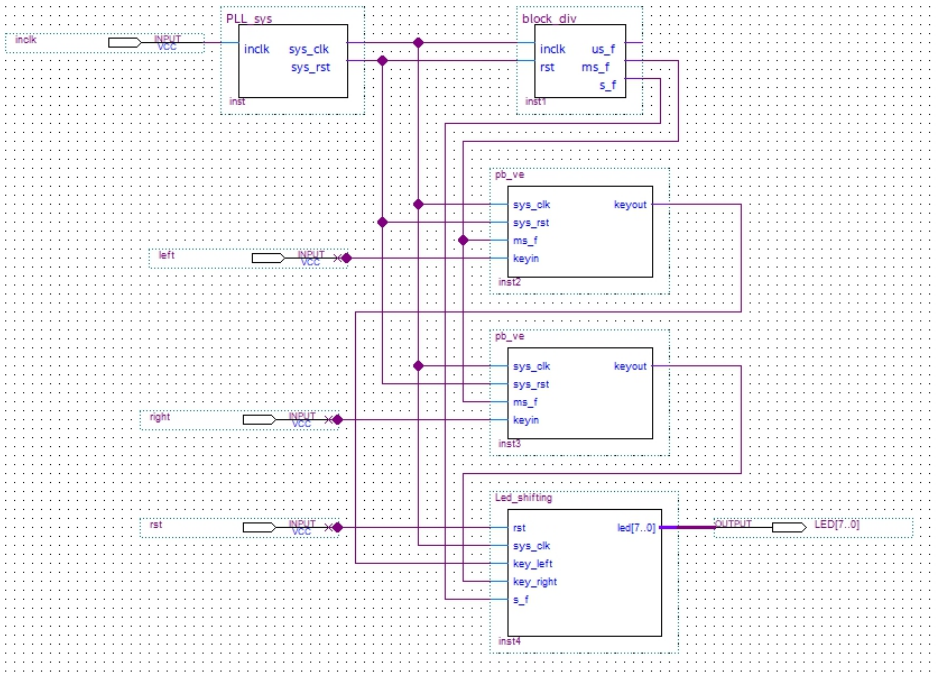


图 5-6　顶层设计

5.3.3.2　程序介绍

分频模块和 LED 显示模块参考之前的实验，对新增部分按键消抖模块进行介绍。

```
module pb_ve (
    input                   sys_clk,
```

```
    input                   sys_rst,
    input                   ms_f,
    input                   keyin,
    output                  keyout
);
    reg                     keyin_r;
    reg                     keyout_r;
    reg             [1:0]   ve_key_st;
    reg             [3:0]   ve_key_count;
always @ (posedge sys_clk)
begin
    keyin_r <= keyin;
end
always @ (posedge sys_clk)
begin
    if (sys_rst) begin
        keyout_r        <= 1'b0;
        ve_key_count    <= 0;
        ve_key_st       <= 0;
    end
    else case (ve_key_st)
        0   :
        begin
            keyout_r        <= 1'b0;
            ve_key_count    <= 0;
            if (keyin_r)
                ve_key_st <= 1;
        end
        1   :
        begin
            if (!keyin_r)
                ve_key_st <= 0;
            else begin
                if (ve_key_count == 10)
                    ve_key_st <= 2;
                else if (ms_f)
                    ve_key_count <= ve_key_count + 1'b1;
            end
        end
        2   :
        begin
            ve_key_count    <= 0;
            if (!keyin_r)
                ve_key_st <= 3;
```

```
            end
            3     :
            begin
                if (keyin_r)
                    ve_key_st <= 2;
                else begin
                    if (ve_key_count == 10) begin
                        ve_key_st       <= 0;
                        keyout_r  <= 1'b1;
                    end
                    else if (ms_f)
                        ve_key_count <= ve_key_count + 1'b1;
                end
            end
            default    :      ;
        endcase
end
   assign keyout = keyout_r;

endmodule
```

本次实验介绍的是一种自适应的按键去抖方法：在探测到按键状态变化时开始计时，在10ms 内如果状态发生变化说明抖动存在，返回初始状态，清除延时计数器，重新探测按键状态，直到延时计数器计满到 10ms 为止。按键的按下与释放采用相同的去抖方法。流程图如图 5-7 所示。

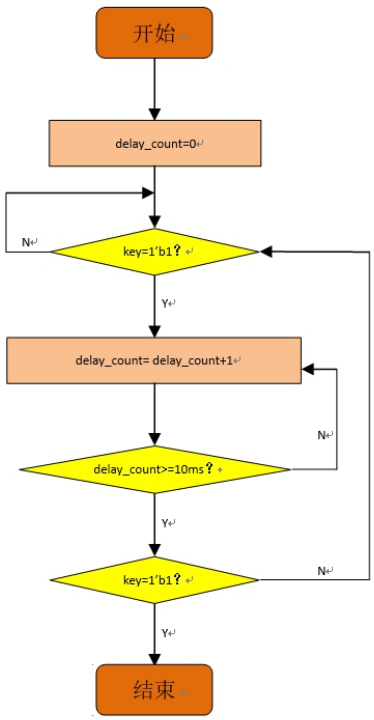


图 5-7　按键去抖流程图

其中状态 0、1 对按键按下状态去抖；2、3 对按键释放去抖。在完成整个去抖后，该程序输出一个同步时钟脉冲。

5.4 实验验证

第一步：分配管脚

管脚分配如表 5-1 所示

表 5-1 按键消抖实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
left	KEY1	3	左移信号
right	KEY2	7	右移信号
rst	KEY3	10	复位信号
led[7]	SW7_LED7	77	Led 灯第 7 位
led[6]	SW6_LED6	76	Led 灯第 6 位
led[5]	SW5_LED5	75	Led 灯第 5 位
led[4]	SW4_LED4	74	Led 灯第 4 位
led[3]	SW3_LED3	87	Led 灯第 3 位
led[2]	SW2_LED2	86	Led 灯第 2 位
led[1]	SW1_LED1	83	Led 灯第 1 位
led[0]	SW0_LED0	80	Led 灯第 0 位

第二步：下板验证

管脚分配完成后进行全编译，通过后进行下板验证。实验现象如图 7-8 至 5-10 所示。

程序下板后，LED 灯全灭，按下复位键，LED 灯全亮，如图 7-8 所示。

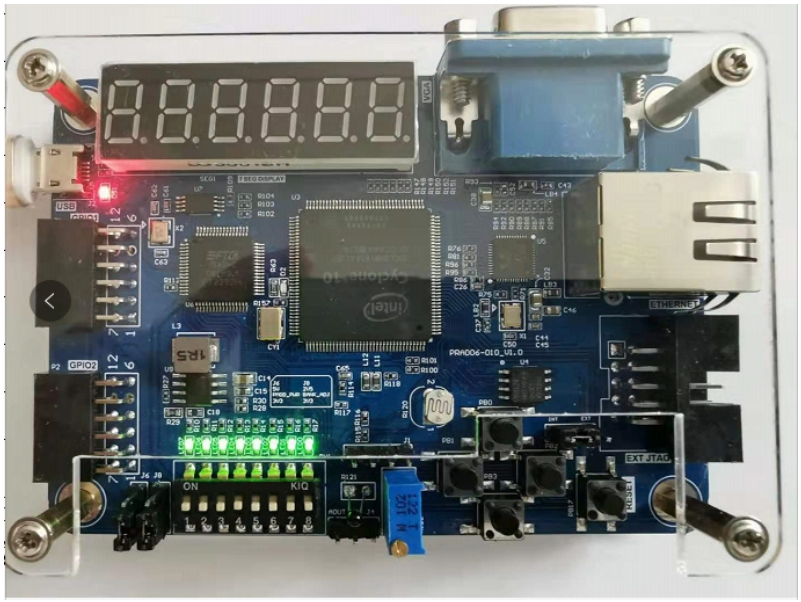


图 7-8 按键消抖实验现象（复位状态）

按下 right 右移按键后，最高位 LED 灯点亮，如图 7-9 所示。

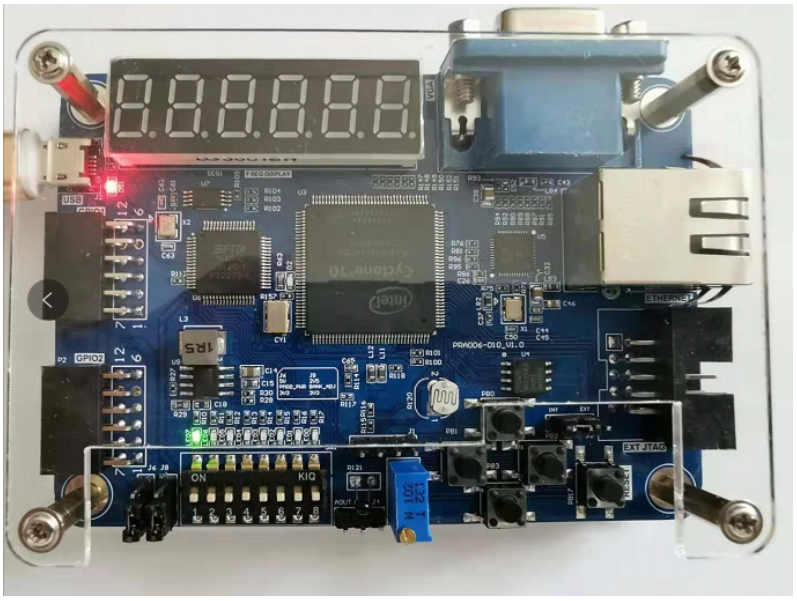


图 7-9　按键消抖实验现象（按一次右移按键）

再次按下右移按键，LED 灯向右移动一位。如图 7-10 所示。

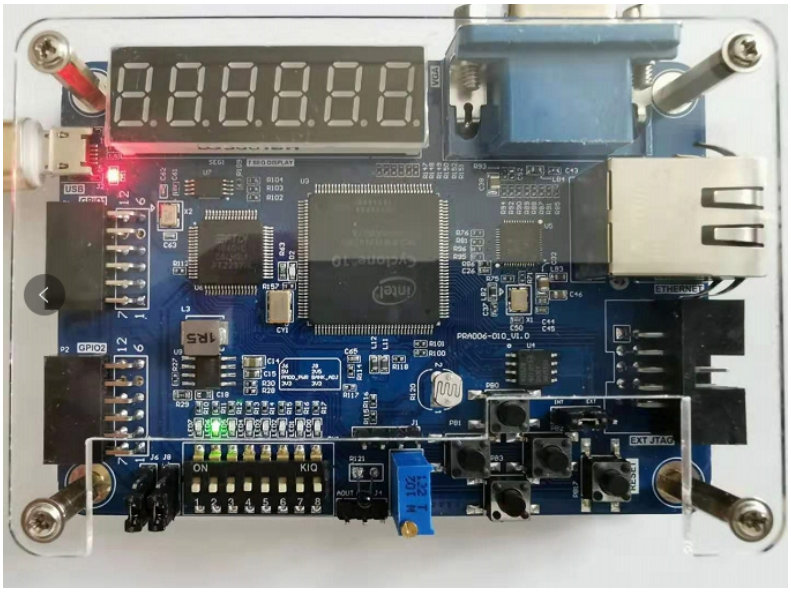


图 7-10　按键消抖实验现象（再按一次下右移按键）

实验六　乘法器的使用与 modelsim 仿真

6.1　实验目的

（1）　学会乘法器的使用
（2）　使用 modelsim 仿真设计输出

6.2　实验要求

（1）　8x8 的乘法器，第一个输入数值为 8 位开关，第二个输入值为 8 位计数器的输出
（2）　在 modelsim 中观察输出结果
（3）　用四位数码管观察计算结果

6.3　实验内容

由于学习使用对仿真工具和新 IP 核的，因此无简介和硬件设计部分。

6.3.1　程序介绍

Modelsim 是一款 HDL 语言仿真软件。可以将我们的程序进行模拟和仿真，达到检验和纠错的功能。使用 Modelsim 仿真时，与之前的实验有所不同，在建立工程的时候需要用将使用的仿真工具在 EDA 工具选择窗口进行添加，如图 6-1 所示。

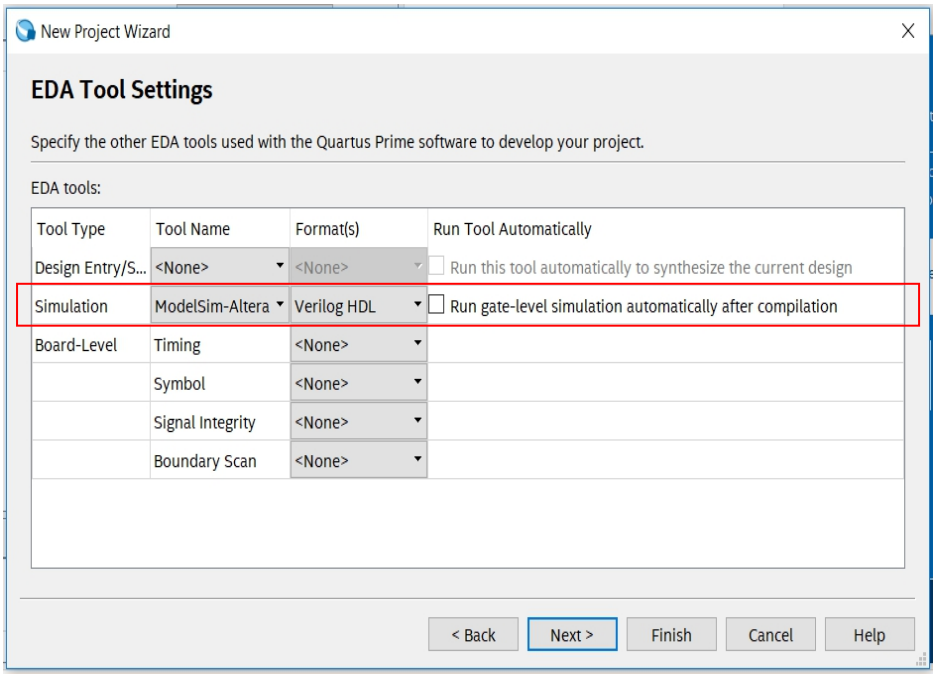


图 6-1　EDA 工具设置

程序中只用到了一个计数器，一个 PLL 和一个乘法器，这里只对乘法器进行介绍。

第一步：主程序框架的建立

```
module mult_sim (
    input               inclk,
    input               rst,
    input       [7:0]   sw,
    output      [15:0]  mult_res,
    output  reg [7:0]   count
);
```

将开关的值作为乘法器第一个输入，将计数器的值第二个输入，并将计算结果输出。

第二步：乘法器 IP 核设置步骤如下：

（1） 添加 LPM_MULT IP（IP Catalog→Library→Basic Functions→Arithmatic→LPM_MULT）保存好路径后，弹出乘法器的设置窗口，如图 6-2 所示，按要求将两个输入数据的尾款设为八位。

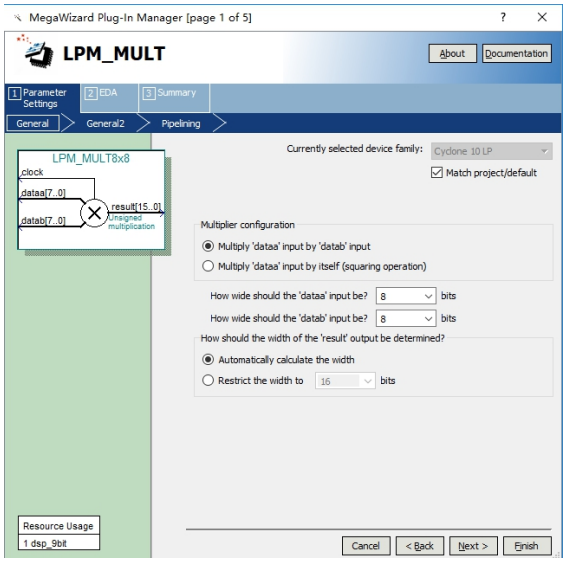


图 6-2 mult 设置 1+

（2） 选择输入数据为 unsigned，如图 6-3 所示

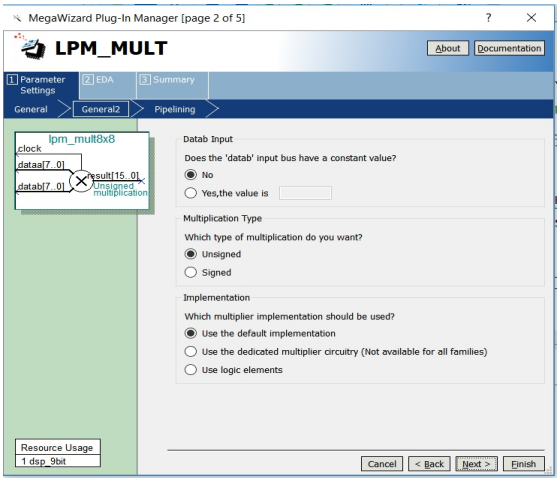


图 6-3 mult 设置 2

（3） 选择带 pipeline（流水线），用以加快运行速度，如图 6-4 所示

（4） 其他默认设置

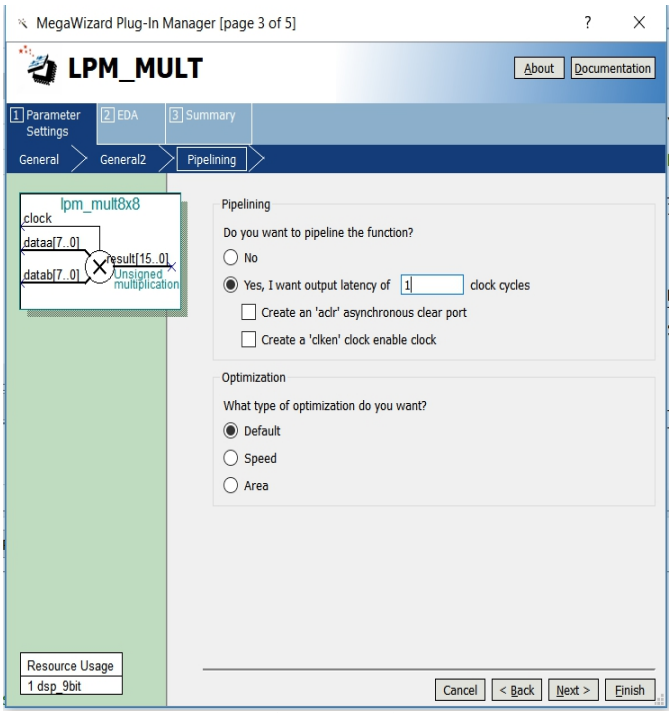


图 6-4 mult 设置 3

8x8 乘法器例化如下

```
reg             sys_clk;
mult_8x8 mult_8x8_inst (
    .clock          (sys_clk),
    .dataa          (sw),
    .datab          (count),
    .result         (mult_res)
);
```

6.4 Modelsim 的使用及实验验证

本次实验我们使用 Modelsim 进行模拟仿真来验证试验。

方法一：基于波形输入的仿真

（1） 点击菜单栏 Tools→Option，如图 6-5 指定路径，点击 OK

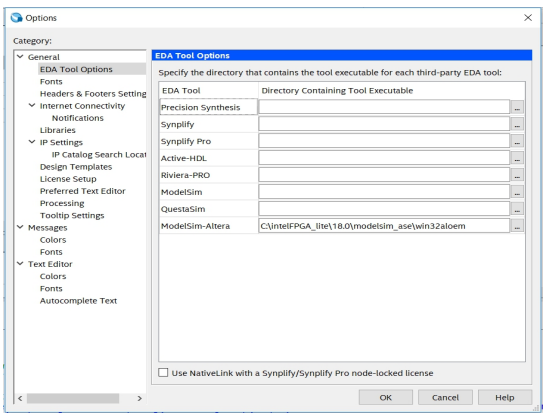


图 6-5 指定 Modelsim-Altera 的路径

（2） 点击菜单栏 Tools→Run Simulation Tool→RTL Simulation 启动 modelsim，如图 6-6

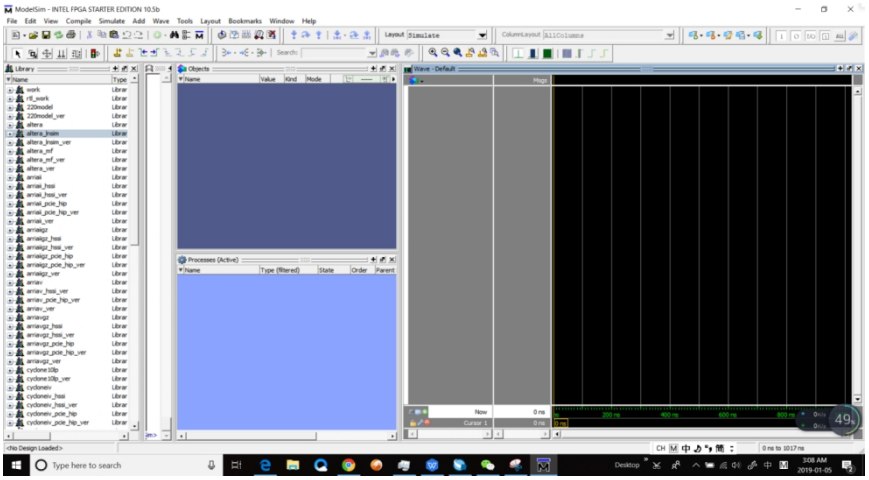


图 6-6 modelsim 窗口

（3） 设置 modelsim

1) 菜单栏 Simulate→Start Simulation

2) 添加 libraries，如图 6-7

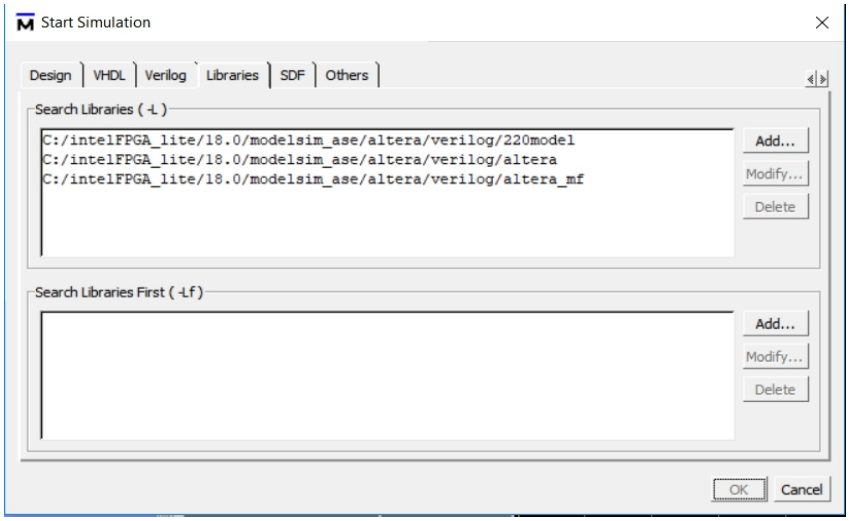


图 6-7 添加仿真库

3) 在 Design 标签选择仿真项目 mult_sim，点击 OK，如图 6-8

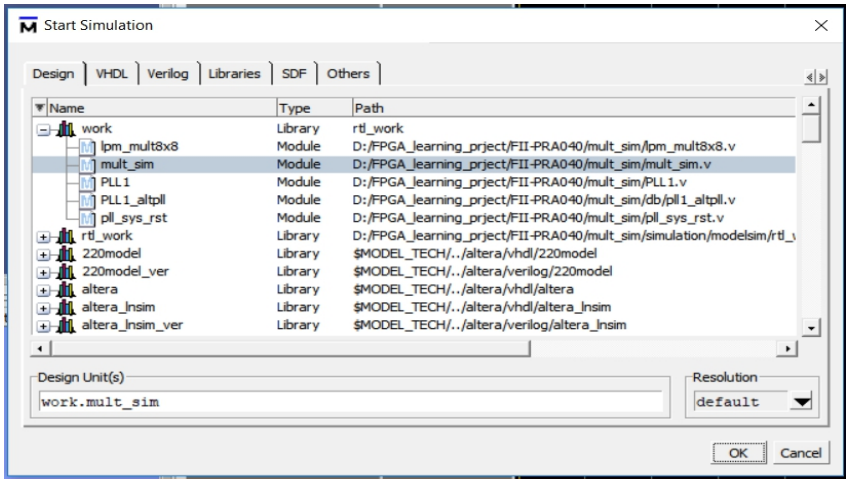


图 6-8 选择要仿真的项目

4) 选择 objects 窗口中的信号拖到 wave 窗口，如图 6-9

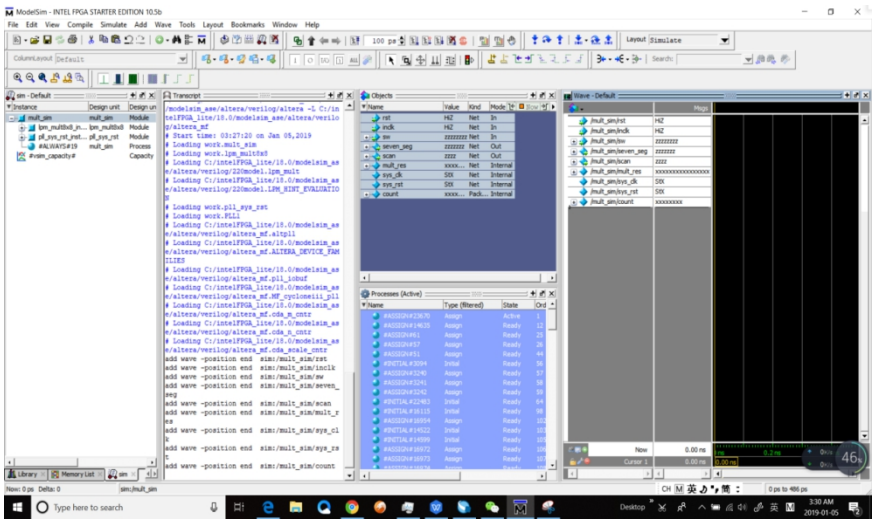


图 6-9 添加观察信号

5) 设置 wave 窗口中的输入信号，右键弹出窗口如图 6-10

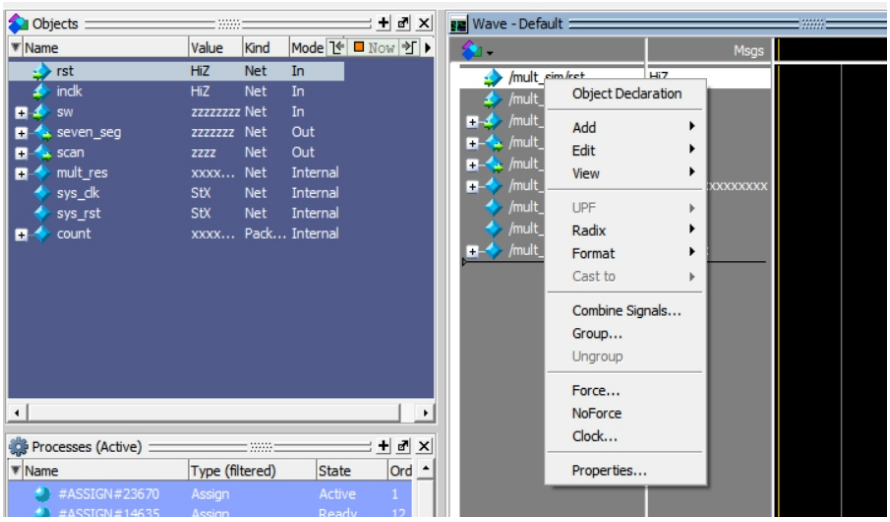


图 6-10 信号设置

6) 逻辑信号选择 force，时钟信号选择 clock

A. rst 信号设置，如图 6-11

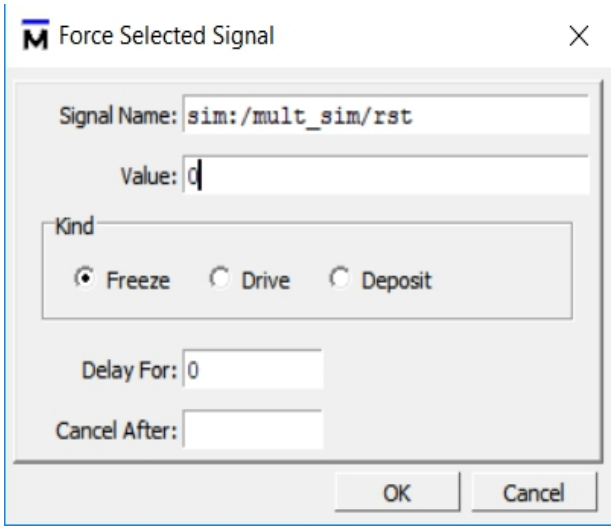


图 6-11 rst 信号设置

B. Inclk 信号设置，如图 6-12

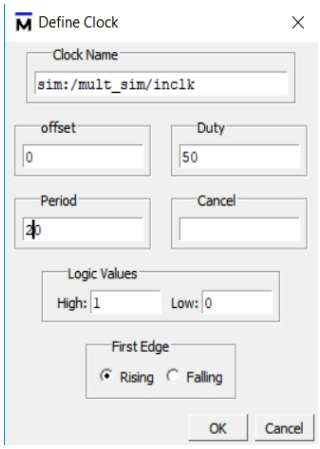


图 6-12 Inclk 信号设置

C. sw 信号设置，如图 6-13

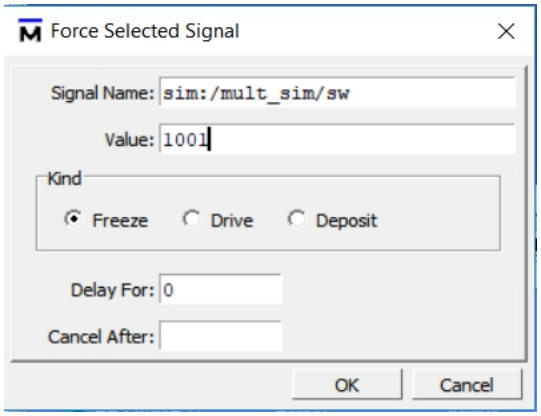


图 6-13 sw 信号设置

7) 运行仿真，工具栏如图 6-14，设置仿真时间 100ns（蓝色圈），点击红色圈内图标运行仿真

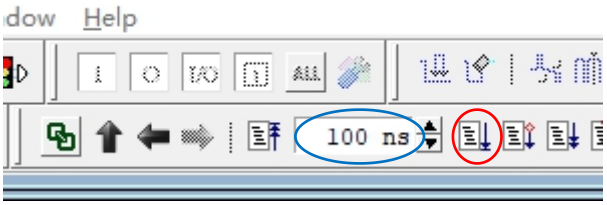


图 6-14 仿真时间设置

8) 观察结果，如图 6-15 所示，

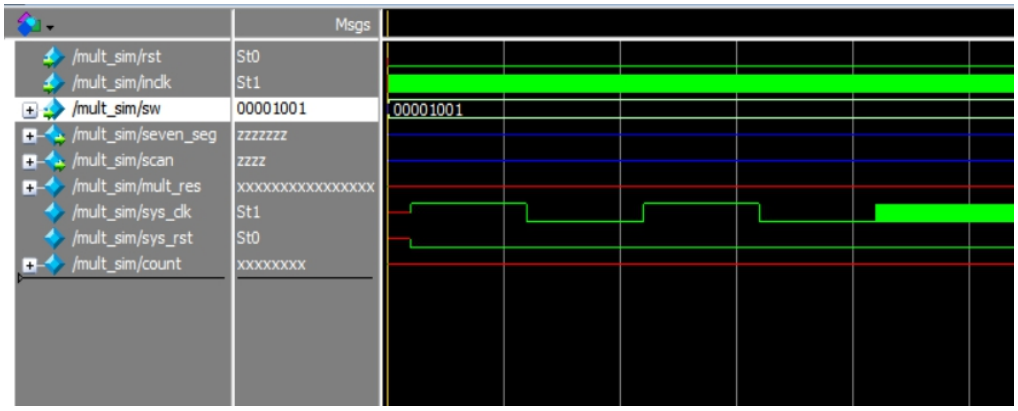


图 6-15 仿真结果

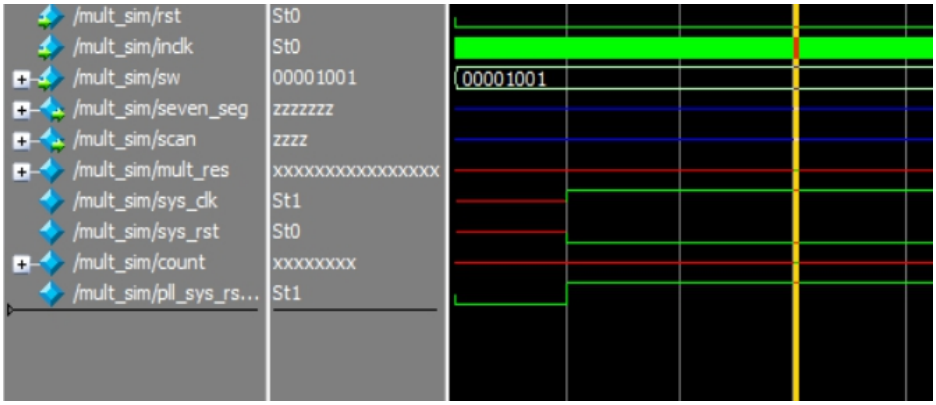
9) 结果分析

A. 数器并没有按预定的方式计数，而是保持了未知 XXXX

B. sys_rst 没有复位信号出现，从开始 X 到 0

C. 添加 pll_locked 信号到波形图中，重新仿真

D. 从图 6-16 中可以看出，由于在 PLL 锁定之前 sys_clk 出现了上升沿，此时 PLL_locked 信号恰好由低到高转换，因此没有形成可靠的复位



6-16 再次仿真结果

E. 解决方案

a. 定义 sys_rst 时设为 1’b1；

b. 利用外部 rst 提供复

这里采用方案 a，修改代码如下：

```
module pll_sys_rst(
    input           inclk,
    output          sys_clk,
    output  reg     sys_rst = 1'b1
);
```

10) 重新编译仿真

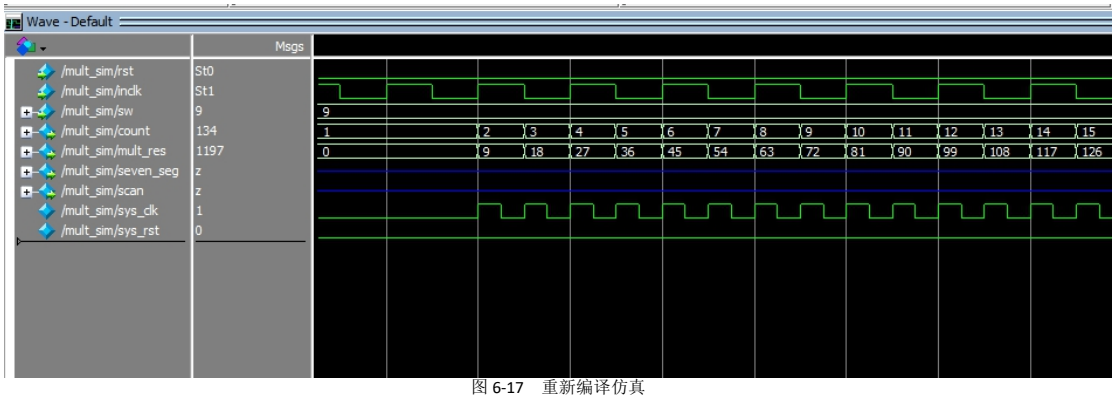


图 6-17 重新编译仿真

由于波形编辑效率较低，因此介绍另一种比较常用的仿真方法：编写仿真激励文件进行仿真。

方法二：编写仿真激励文件进行仿真

（1） 新建仿真激励文件，文件名为 tb_mult.v

（2） 具体程序如下：

```
`timescale 1ns/1ps
module tb_mult;
    reg             rst;
    reg             clk;
    reg     [7:0]   sw;
    wire    [7:0]   count;
    wire    [15:0]  mult_res;

//将需要仿真的工程例化进激励文件
mult_sim S1(
    .rst            (rst),
    .inclk          (clk),
    .sw             (sw),
    .count          (count),
    .mult_res       (mult_res)
);
//定义仿真所需要的时钟，将结果以文本形式显示出来
always begin
    #10 clk = ~clk;
    $monitor ("%d * %d = %d", count, sw, mult_res);
end
//设置仿真条件
initial begin
    rst = 0;
    clk = 1;

    #10 sw = 20;
    #10 sw = 50;
    #10 sw = 100;
    #10 sw = 101;
    #10 sw = 102;
    #10 sw = 103;
    #10 sw = 104;
    #50 sw = 105;
    //停止信号
    #1000 $stop;
end
endmodule
```

编写激励文件时，首先要在开头标注仿真的时间单位，本次实验为 1ns，然后将需要仿真的工程例化到激励文件中，定义好时钟周期后以及仿真条件，并在一定时间后停止仿真，本次仿真经过 1000 个时钟周期后停止。

编写完成，进行编译，编译通过后，将激励文件添加到 Modelsim 中进行仿真，具体步

骤如下：

A. 点击菜单栏 Assignments→settings 如图 6-19 所示

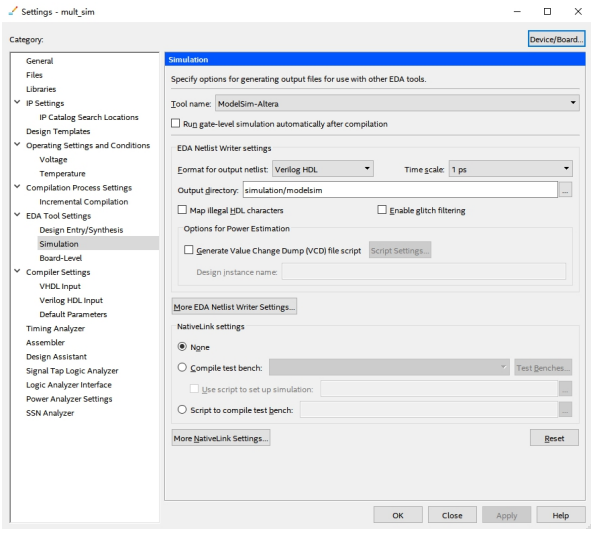


图 6-19 仿真设置 1

B. 选择 Compile test bench，点击 Test Benches 添加 tb 仿真文件，如图 6-20 所示

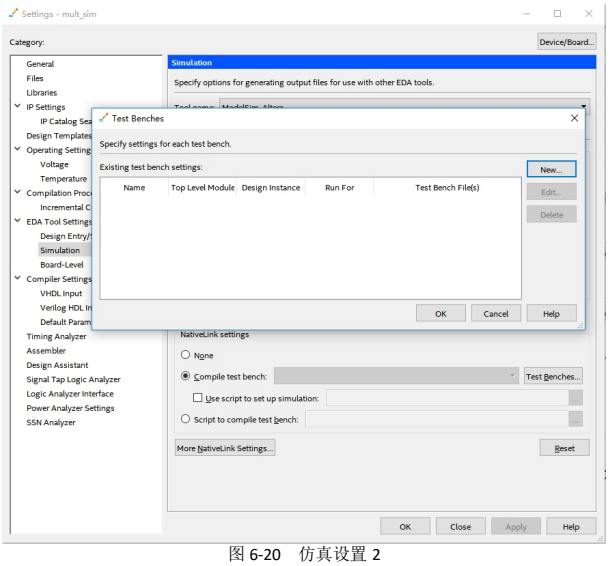


图 6-20 仿真设置 2

C. 点击 New，如图 6-21 所示，输入名称（与 tb 文件保持一致）

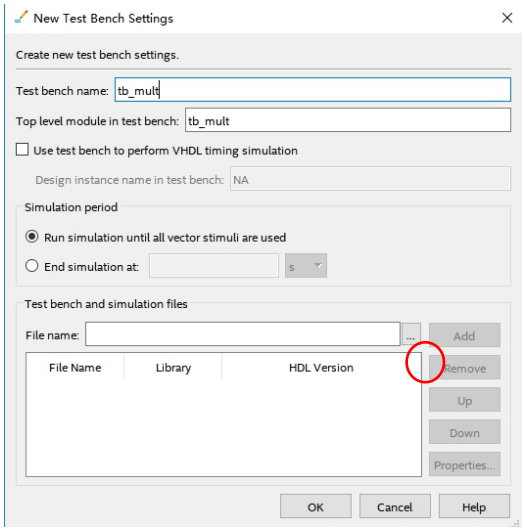


图 6-21 仿真设置 3

D. 点击图 6-21 红色圈内…添加仿真文件，找到之前写好的 tb_mult.v 文件

E. 打开，点击 Add 添加，如图 6-22 所示，然后点击 OK（三次），完成设置。

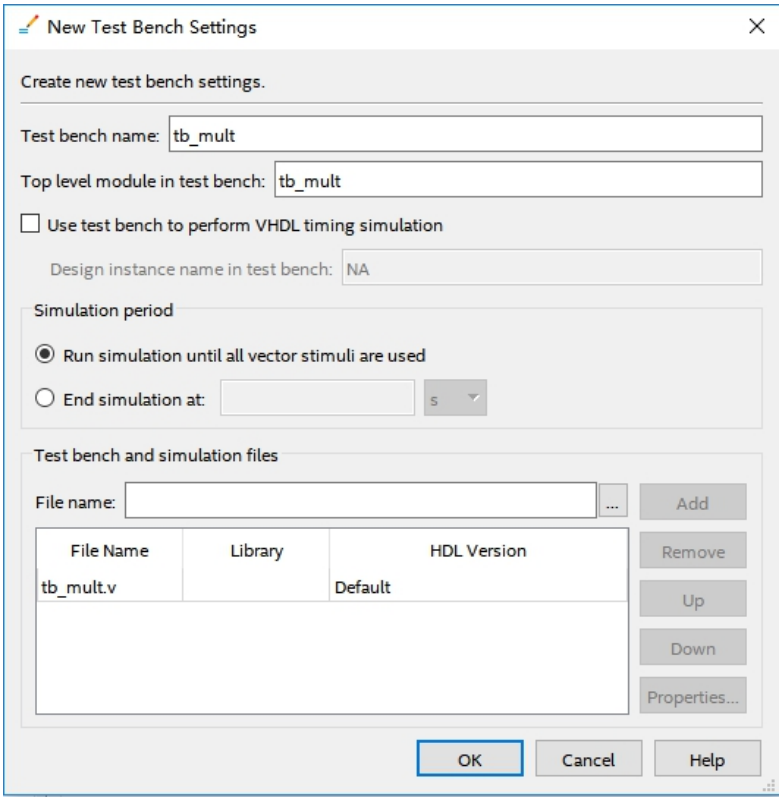


图 6-22 仿真设置 4

（3） 点击菜单 Tools→Run Simulation Tool→RTL Simulation，打开 modelsim。运行结果如图 6-23 所示。

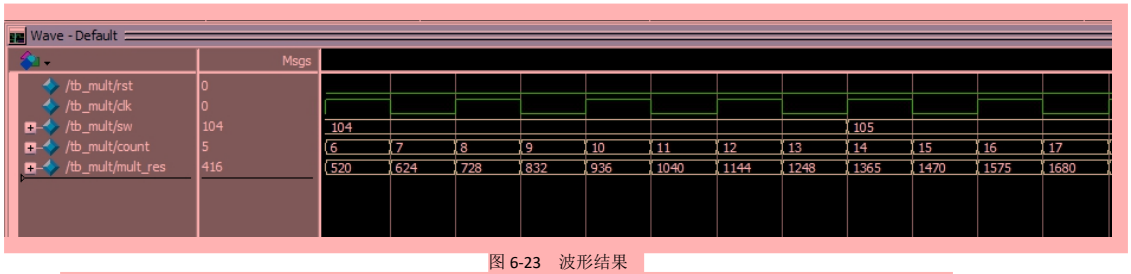


图 6-23 波形结果

达到一定延时后在 Trascript 出口显示运行结果的文本信息，如图 6-24。

因为运算结果会比输入晚一个时钟周期，所以文本信息中，乘数和结果会错一行，看起来不匹配，但不影响对实验结果的分析。

```
#    1 * 102 =     0
#    2 * 103 =   103
#    3 * 103 =   206
#    4 * 104 =   312
#    5 * 104 =   416
#    6 * 104 =   520
#    7 * 104 =   624
#    8 * 104 =   728
#    9 * 104 =   832
```

图 6-24 文本显示运行结果信息

实验七　十六进制数到 BCD 码转换及应用

7.1 实验目的

（1）　学会二进制数转 BCD 码（bin_to_bcd）
（2）　学会十六进制数转 BCD 码（hex_to_bcd）

7.2　实验要求

结合实验六，将运算结果显示到数码管上。

7.3　实验内容

7.2.1　十六进制数转 BCD 码原理简介

由于 16 进制数显示不直观，因此在实际开发中往往采用 10 进制显示输出。

人眼识别数字速度较慢，因此 16 进制到 10 进制的显示也不需要太快，一般采用两种方法进行性

（1）　减计数方法：

在同步时钟的控制下将 16 进制数做循环减 1，直到减为 0。同时设计合适的 BCD 码 10 进制计数器，当 16 进制的数减为 0 时，BCD 计数器恰好得到与 16 进制数相等的数值进行显示。

（2）　快速运算方法：移位加 3 运算得到，实现方法如下，

1) 设定表达的 10 进制最大值，假定要将 16 位 2 进制数值(4 位 16 进制）转换为 10 进制，数值最大可以表达 65535。先定义 5 个四位 2 进制数单元 ten_thousands, thousands, hundreds, tens, ones 来容纳计算结果
2) 将 16 进制数逐次左移 1 位，并将移出的部分放入定义好的变量中，依次判断 ten_thousands, thousands, hundreds, tens, ones 的单元中是否大于等于 5，如果大于 5，则将对应的位加 3，直到 16 位移位完毕，则得到相应的结果。
   注意：在移到最后一位时不做加 3 处理，直接得到运算结果
3) 十六进制数到 BCD 码转换的原理
   假设 ABCD 为 4 位 2 进制数（可能为个位，10 位或 100 位等），将其调整为 BCD 码。由于整个计算都是在逐次移位中实现的，移动一位后会得到 ABCDE（E 是从低位移过来的,其数值为 0 或 1）。此时应判断该值是否大于等于 10，如果大于等于 10，则该值加 6 将其调整到 10 以内，并将进位位移到高 4 位 BCD 码上。这里采用移动前调整，先判断 ABCD 是否大于等于 5（10 的一半），如果大于 5 就做加 3（6 的一半），然后进行移位。
   例如 ABCD=0110（十进制 6）
   A. 移位后为 1100（12），大于 1001（十进制 9）
   B. 加 0110（十进制 6），ABCD=0010,进位位为 1，结果表达为十进制 12

C. 采用移位前处理，ABCD=0110(6),大于等于 5，加 3
D. ABCD=1001（9），左移一位
E. ABCD=0010，移出的一位作为高四位 BCD 的最低位
F. 由于移出的位为 1，ABCD=0010（2），结果也位 10 进制的 12
G. 可以看出运算结果一致
H. 需要注意（1）先判断，再加 3，最后移位（2）如果有多个 BCD 码同时存在，则多个 BCD 码都要先判断是否需要加 2 调整再移位。

（3） 第一种方法比较简单，使用者可以自行练习，下面主要以第二种方法为主介绍。

例 1：二进制转 BCD，过程如图 7-1 所示

100's	10's	1's	Binary	Operation
			1010 0010	162
		1	010 0010	<< #1
		10	10 0010	<< #2
		101	0 0010	<< #3
		1000		add 3
	1	0000	0010	<< #4
	10	0000	010	<< #5
	100	0000	10	<< #6
	1000	0001	0	<< #7
	1011			add 3
1	0110	0010		<< #8
1	6	2		

图 7-1 例 1，bin_to_bcd

例 2：十六进制转 BCD，过程如图 7-2 所示

Operation	Hundreds	Tens	Units	Binary	
HEX				F	F
Start				1 1 1 1	1 1 1 1
Shift 1			1	1 1 1 1	1 1 1
Shift 2			1 1	1 1 1 1	1 1
Shift 3			1 1 1	1 1 1 1	1
Add 3			1 0 1 0	1 1 1 1	1
Shift 4		1	0 1 0 1	1 1 1 1	
Add 3		1	1 0 0 0	1 1 1 1	
Shift 5		1 1	0 0 0 1	1 1 1	
Shift 6		1 1 0	0 0 1 1	1 1	
Add 3		1 0 0 1	0 0 1 1	1 1	
Shift 7	1	0 0 1 0	0 1 1 1	1	
Add 3	1	0 0 1 0	1 0 1 0	1	
Shift 8	1 0	0 1 0 1	0 1 0 1		
BCD	2	5	5		

http://blog.csdn.net/l1200503028

图 7-2 hex_to_bcd

7.2.2 程序介绍

第一步：主程序框架的建立

```
module HEX_BCD (

    input           [15:0] hex,
    output  reg     [3:0]  ones,
    output  reg     [3:0]  tens,
    output  reg     [3:0]  hundreds,
    output  reg     [3:0]  thousands,
    output  reg     [3:0]  ten_thousands
);
```

输入一个 16 位二进制数 hex，最大可以表示十进制 65535，因此输出个位 ones，十位 tens，百位 hundreds，千位 thousands，以及万位 ten_thousands。

第二步：快速算法的实现

```
reg         [15:0]      hex_reg;
integer                 i;
always @ (*)
begin
    hex_reg = hex;
    ones = 0;
    tens = 0;
    hundreds = 0;
    thousands = 0;
    ten_thousands = 0;

    for (i = 15; i >= 0; i = i-1) begin
        if(ten_thousands >= 5)
        ten_thousands = ten_thousands + 3;
        if(thousands >= 5)
            thousands = thousands + 3;
        if(hundreds >= 5)
            hundreds = hundreds + 3;
        if(tens >= 5)
            tens = tens + 3;
        if(ones >= 5)
            ones = ones + 3;

        ten_thousands = ten_thousands << 1;
        ten_thousands[0] = thousands[3];
        thousands = thousands << 1;
        thousands[0] = hundreds[3];
```

```
            hundreds = hundreds << 1;
            hundreds[0] = tens[3];
            tens = tens << 1;
            tens[0]= ones[3];
            ones = ones << 1;
            ones[0] = hex_reg[15];
            hex_reg = {hex_reg[14:0], 1'b0};
        end
end
```

参照图 7-2，程序前一部分为判断计算部分，大于等于 5 则加 3，后一部分为移位部分。

第三步：验证

参考实验六，采用 Modelsim 进行仿真，仿真条件设置如下：

```
initial   begin
      hex = 0 ;
      repeat (20) begin
      #10;
      hex = {$random}%20000;
      #10;
      end
end
```

开始时 16 位二进制数等于 0，然后延时 10ns，16 位二进制数取小于 20000 的随机数，延时 10ns 以后结束，整个过程重复 20 遍。

对 modelsim 设置完成，添加完激励文件后，进行仿真，结果如图 7-3 所示

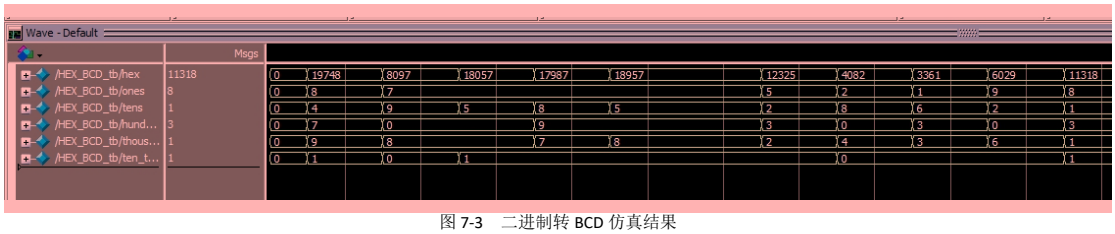


图 7-3　二进制转 BCD 仿真结果

注意与思考：

A. 上例中的赋值符号全部使用“=”而不是“<=”符号，为什么？

B. 由于整个程序采用组合逻辑设计，因此在调用该程序时应注意和调用者中其它模块的时序同步。

7.4　十六进制数到 BCD 码转换的应用

（1）　继续完成实验六的乘法器，并将结果以 10 进制的方式用数码管显示，每经过 1S 钟，数码管上的计算结果改变一次，实验需要用到分频，数码管显示，乘法器以及十六进制数转 BCD 码。

（2）　编译，打开 Compilation Report 观察 timing Analysis，

1) Fmax Summary　83.71Mhz，如图 7-4

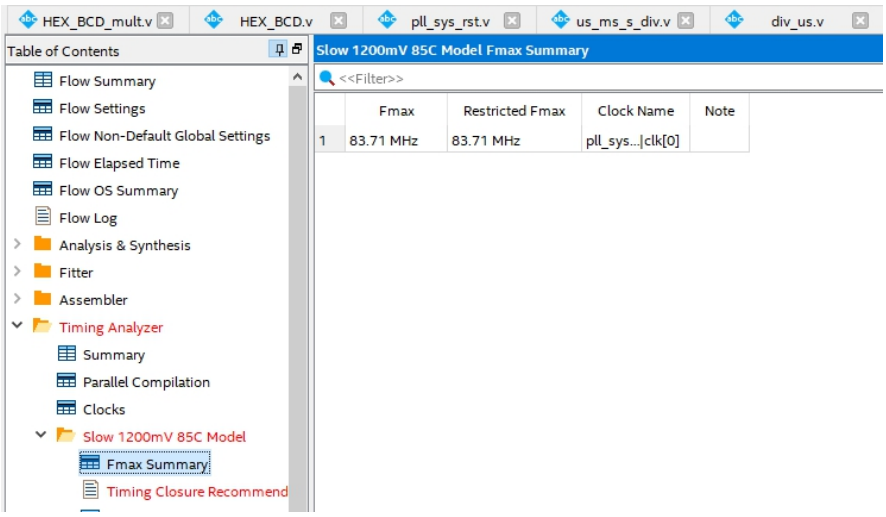


图 7-4　Fmax Summary

2)　Setup Memory

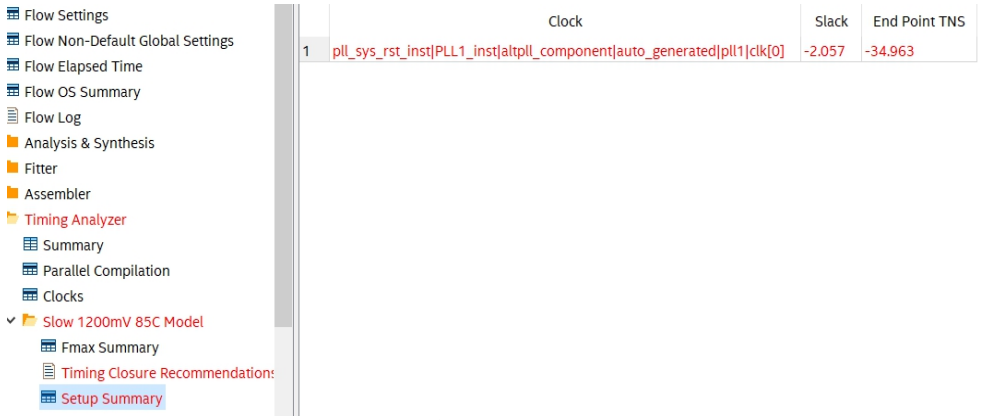


图 7-5　Setup time summary

3)　Timing Closure Recommendation　如图 7-6

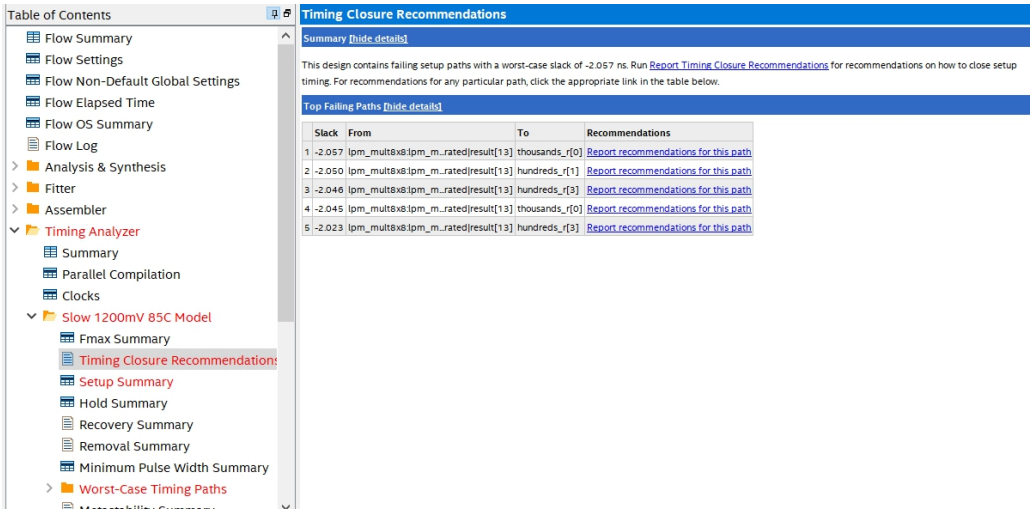


图 7-6　Timing Analysis

4)　从上面三项指标，可以看出上面的程序设计并不满足时序要求。同时也可以看到，最大的延时路径是乘法器的输出到 HEX_BCD 的延时。

有 3 种解决办法：

A.　降低时钟频率

B.　提高 HEX_BCD 的时序，增加流水线

C. 在外围插入流水线隔离（可以减小一部分延迟）

增加流水线的方式，后续实验中介绍，由于 HEX_BCD 的功能主要用来做人机界面显示，对速度要求较低，这里采用降频的方式进行。

（3） 修改 PLL 增加一个 20M 频率输出（BCD_clk）。

（4） 重新编译，观察时序结果

（5） 管脚锁定，编译，下载到 FII-PRA006/010 开发板上，进行测试

7.5 实验验证

第一步：分配管脚

管脚分配表如表 7-1 所示。

表 7-1 十六进制数转 BCD 实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
clk	C10_50MCLK	91	输入时钟
rst	KEY3	10	复位按键
scan[5]	SEG_3V3_D5	124	位选信号第 5 位
scan[4]	SEG_3V3_D4	127	位选信号第 4 位
scan[3]	SEG_3V3_D3	129	位选信号第 3 位
scan[2]	SEG_3V3_D2	141	位选信号第 2 位
scan[1]	SEG_3V3_D1	142	位选信号第 1 位
scan[0]	SEG_3V3_D0	136	位选信号第 0 位
seven_seg[7]	SEG_PA	128	段选信号第 7 位
seven_seg[6]	SEG_PB	135	段选信号第 6 位
seven_seg[5]	SEG_PC	138	段选信号第 5 位
seven_seg[4]	SEG_PD	126	段选信号第 4 位
seven_seg[3]	SEG_PE	125	段选信号第 3 位
seven_seg[2]	SEG_PF	133	段选信号第 2 位
seven_seg[1]	SEG_PG	137	段选信号第 1 位
seven_seg[0]	SEG_DP	132	段选信号第 0 位
SW[7]	SW7_LED7	77	拨码开关第 7 位
SW[6]	SW6_LED6	76	拨码开关第 6 位
SW[5]	SW5_LED5	75	拨码开关第 5 位
SW[4]	SW4_LED4	74	拨码开关第 4 位
SW[3]	SW3_LED3	87	拨码开关第 3 位
SW[2]	SW2_LED2	86	拨码开关第 2 位
SW[1]	SW1_LED1	83	拨码开关第 1 位
SW[0]	SW0_LED0	80	拨码开关第 0 位

第二步：下板验证

管脚分配完成后进行全编译，通过后进行下板验证。实验现象如图 7-7 所示。拨码开关输入的值为 00001010，十进制为 10，计数器不断在累加，因此显示结果也一直以 10 累加变化。

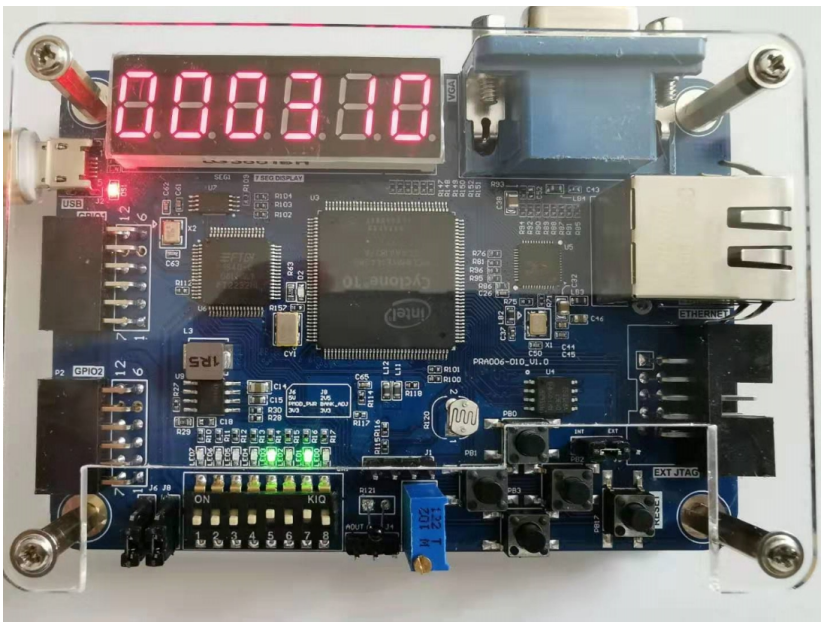


图 7-7　BCD 码转换实验现象

实验总结与思考

（1）　超过 16 位的 2 进制转 BCD 码如何实现；
（2）　什么是同步时钟，如何处理异步时钟；
（3）　学习如何根据实际需求，设计满足时序要求的电路。

实验八　ROM 的使用

8.1　实验目的

（1）　学习 FPGA 内部 memory block 的
（2）　学习*.mif 的格式，以及学会如何编辑 mif 文件来配置 ROM 的内容
（3）　学会 RAM 的使用，以及 RAM 的读写

8.2　实验要求

（1）　设计 16 数据输出的 ROM，地址范围 0-255
（2）　接口位 8 位开关输入，作为 ROM 的寻址
（3）　码管显示 ROM 的内容，要求将 16 进制数转换为 BCD 输出

8.3　实验内容

8.3.1　程序介绍

本次实验是在实验 7 的基础上进行，引用实验 7 的内容，因此只介绍 IP 核 ROM 部分。

（1）　在 IP Catalog 选择 ROM: 1-PORT, IP variation file type 选择 verilog，如图 8-1 所示

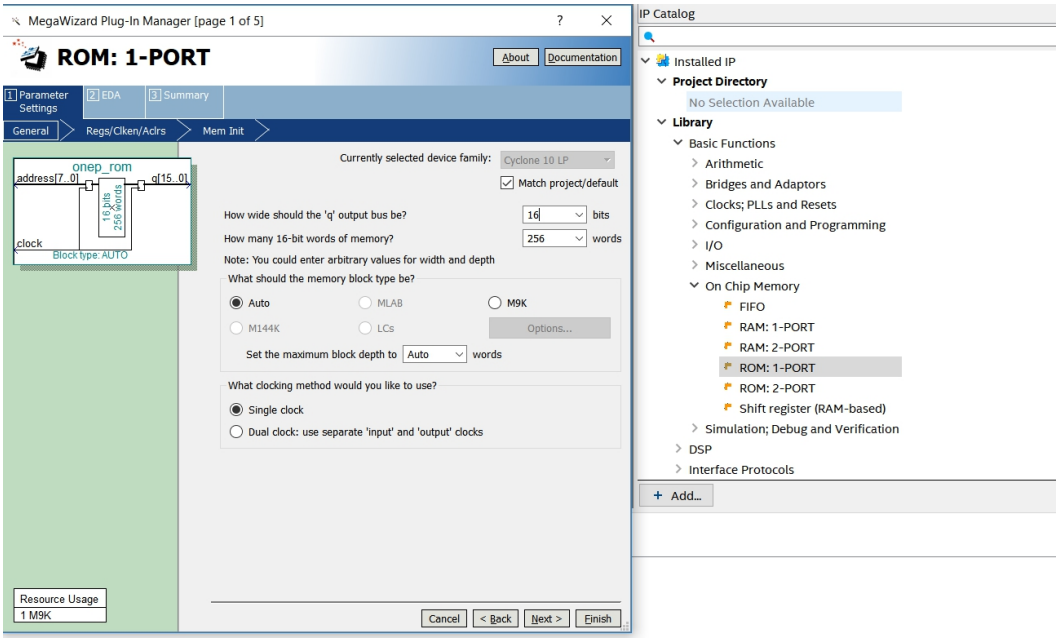


图 8-1　ram IP 核调用

（2）　按缺省设置进行，在如图 8-2 所示的设置界面中红色椭圆圈内，需要添加一个 ROM 的初始化文件.mif 文件，（图片中已经添加完毕）

（3）　创建顶层文件 rom.mif

1)　生成 rom.mif 文件，file→new→Memory Initialization File，如图 8-3 所示

2) 如图 8-4 所示，在红色圈内修改存储深度（Number of words）和字节宽度（Word size），

3) 如图 8-5 所示红色椭圆线标示的区域，在行列的地址区，右键可以选择输入数据和地址显示的格式，如 16 进制，8 进制，2 进制，有符号数，无符号数等

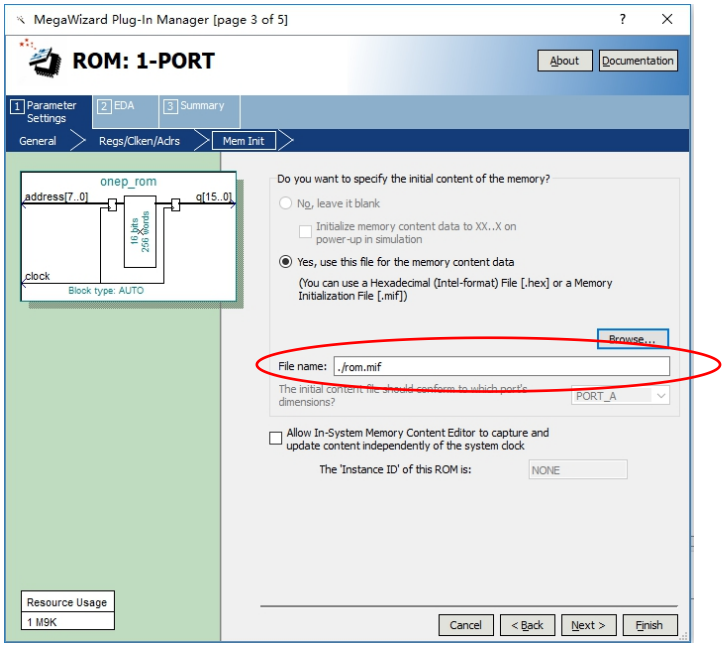


图 8-2 ROM 设置

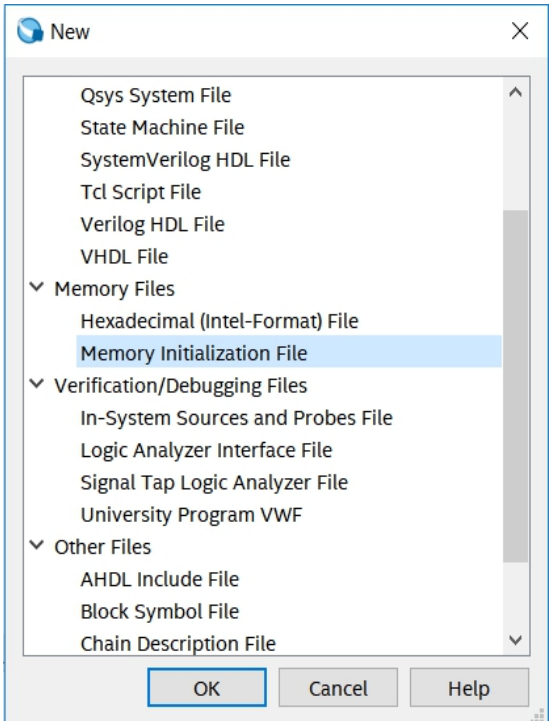


图 8-3 新建.mif 文件

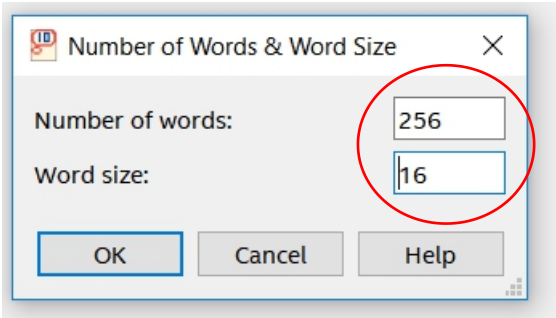


图 8-4 .mif 文件设置 1

Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	0000	0900	FFFF	00FF	0200	0044	0027	04F6
008	0000	0000	0000	0000	0000	0000	0000	0000
010	0000	0000	0000	0000	0000	0000	0000	0000
018	0000	0000	0000	0000	0000	0000	0000	0000
020	0000	0000	0000	0000	0000	0000	0000	0000
028	0000	0000	0000	0000	0000	0000	0000	0000
030	0000	0000	0000	0000	0000	0000	0000	0000
038	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000
048	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000
058	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000
068	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000
078	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000

图 8-5 .mif 文件设置 2

4) 完成对 rom IP 和的设置后，对 rom.mif 进行数据的填写，为了方便验证，从低字节到高字节以递增的形式存储和地址一样的数据，如图 8-6 所示，起始地址 0，终止地址 255（之前设置的地址深度为 256），初始值为 0，步进为 1。

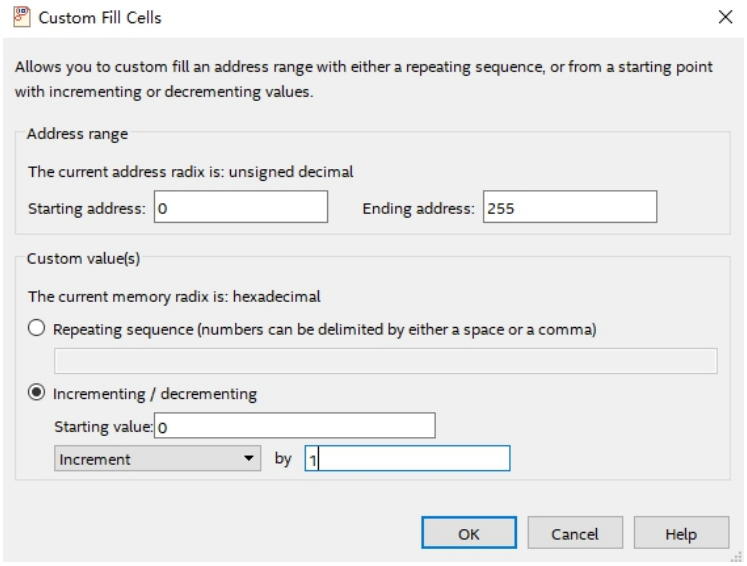
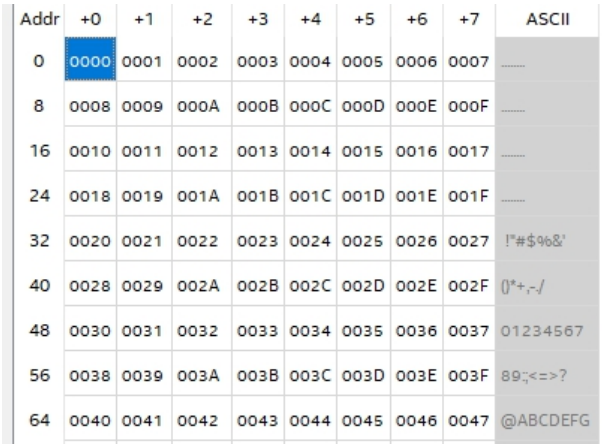


图 8-6 rom.mif 进行数据填写

5) 设置完成之后，系统将自动填写数据，如图 8-7 所示。



Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0000	0001	0002	0003	0004	0005	0006	0007	.....
8	0008	0009	000A	000B	000C	000D	000E	000F	.....
16	0010	0011	0012	0013	0014	0015	0016	0017	.....
24	0018	0019	001A	001B	001C	001D	001E	001F	.....
32	0020	0021	0022	0023	0024	0025	0026	0027	!"#$%&'
40	0028	0029	002A	002B	002C	002D	002E	002F	()*+,-./
48	0030	0031	0032	0033	0034	0035	0036	0037	01234567
56	0038	0039	003A	003B	003C	003D	003E	003F	89:;<=>?
64	0040	0041	0042	0043	0044	0045	0046	0047	@ABCDEFG

图 8-7　自动填写后的部分数据

（4）　引用实验七中十六进制数转 BCD 码的实验，将 ROM 中的数据 rom_q 经过一级寄存器同步后，显示在数码管上

ROM 例化如下：

```
reg   [15:0] rom_q_r;

always @ (posedge BCD_clk)
      rom_q_r<=rom_q;

HEX_BCD HEX_BCD_inst(
      .hex              (rom_q_r),
      .ones             (ones),
      .tens             (tens),
      .hundreds         (hundreds),
      .thousands        (thousands),
      .ten_thousands    (ten_thousands)
);
onep_rom    onep_rom_dut(
      .address          (sw),
      .clock            (sys_clk),
      .q                (rom_q)
);
```

8.4　实验验证

管脚分配与实验 7 一致。编译完成后下板验证。如图 8-8 所示。当拨码开关为 10010100 时，十进制为 148，即表示我们将读出 rom 中第 148 字节内的内容，数码管显示为 148，与我们存入的数据一致。

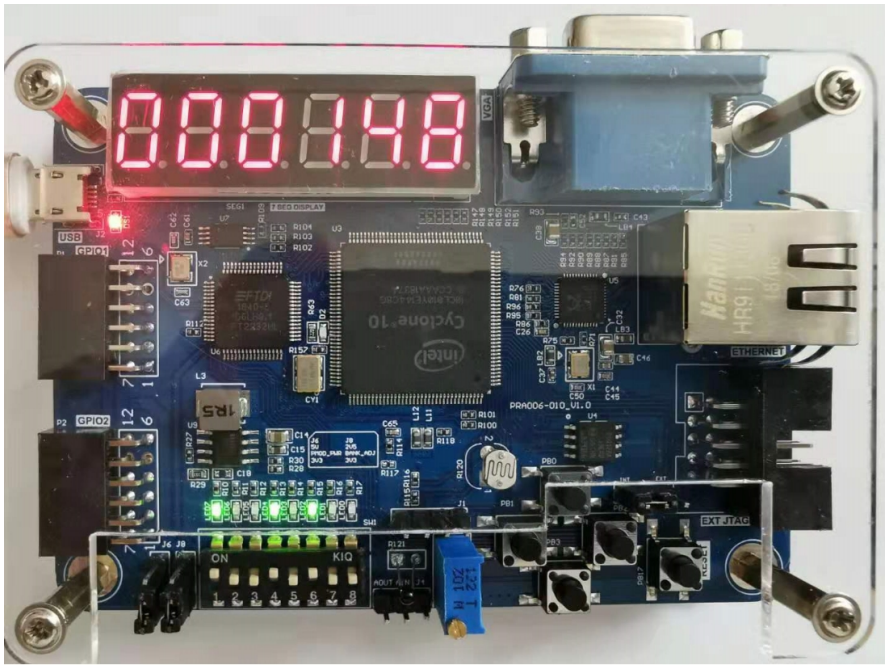


图 8-8　rom 实验现象

实验总结与思考

1. 如何利用 ROM 的初始化文件实现译码，如 6 位 7 段数码管的译码及扫描的译码
2. 编写 mif 文件形成正弦波，余弦波形等函数发生器
3. 综合应用，结合 ROM 和 PWM 的特点形成 SPWM 调制波形

实验九　利用双端口 RAM 实现帧数据的读写

9.1　实验目的

（1）　学会双端口 RAM 的配置及使用
（2）　学会使用同步时钟控制帧结构的同步
（3）　学会使用异步时钟控制帧结构的同步

9.2 实验要求

（1）　使用 signaltap II 观察同步时钟帧的同步结构
（2）　双口 RAM 的扩展使用
（3）　设计三段状态机的使用
（4）　设计一个 16 位数据帧

1) 数据由 8 位计数器生成：Data={~counta,counta}
2) 由开关输入数据帧的 ID（7 位最大表达 128 个不同的数据帧）
3) 16 位 checksum 提供数据校验
   A. 16 位 checksum 累加，丢弃进位位
   B. checksum 取补后，追加到帧数据之后
4) 由 parameter 提供可配置的数据长度 data_len
5) 封包：当数据和 checksum 包写到双口 RAM 后，最后将 userID， 帧长度和 valid 标志写到双口 RAM 的特定位置，memory 的结构如表 1 所示

Wr_addr	数据/标志	Rd_addr
8'hff	{valid,ID,data_len}	8'hff
...	N/A	...
8'hnn+2	N/A	8'hnn+2
8'hnn+1	~checksum+1	8'hnn+1
8'hnn	datann	8'hnn
...	....	...
8'h01	Data1	8'h01
8'h00	Data0	8'h00

表 1　memory 结构

6) 读、写按约定的顺序进行

握手信号为 valid，这个标志为读写同步提供了可能，因此在程序设计中一定要处理好该信号的准确性。

9.3　实验内容

9.3.1　程序介绍

第一步：主程序框架的建立

```
module frame_ram
#(parameter data_len=250)
(
input                   inclk,
input                   rst,
input           [6:0]   sw,
output    reg   [6:0]   oID,
output    reg           rd_done,
output    reg           rd_err
);
```

第二步：状态机的定义

```
parameter       [2:0]       mema_idle=0,
                            mema_init=1,
                            mema_pipe0=2,
                            mema_read0=3,
                            mema_read1=4,
                            mema_wr_data=5,
                            mema_wr_chsum=6,
                            mema_wr_done=7;

parameter       [2:0]       memb_idle=0,
                            memb_init=1,
                            memb_pipe0=2,
                            memb_read0=3,
                            memb_read1=4,
                            memb_rd_data=5,
                            memb_rd_chsum=6,
                            memb_rd_done=7;
```

第三步：其他定义：

```
时钟变量定义
wire                    sys_clk;
wire                    BCD_clk;
wire                    sys_rst;
reg                     ext_clk;
双口 RAM 接口定义
reg         [7:0]       addr_a;
reg         [15:0]      data_a;
reg                     wren_a;
wire        [15:0]      q_a;
reg         [7:0]       addr_b;
reg                     wren_b;
```

```
wire        [15:0]      q_b;
写状态机部分变量定义
reg         [6:0]       user_id;
reg         [7:0]       wr_len;
reg         [15:0]      wr_chsum;
wire                    wr_done;
reg         [7:0]       counta;
wire        [7:0]       countb;
assign countb=~counta;
reg         [15:0]      rd_chsum;
reg         [7:0]       rd_len;
reg         [15:0]      rd_data;
reg                     ext_rst;
reg         [2:0]       sta;
reg         [2:0]       sta_nxt,;
读状态机部分变量定义
reg         [15:0]      rd_chsum;
reg         [7:0]       rd_len;
reg         [15:0]      rd_data;
reg         [2:0]       stb;
reg         [2:0]       stb_nxt;
```

第四步：生成双口 RAM，PLL

```
dp_ram dp_ram_inst
(
    .address_a          (addr_a),
    .address_b          (addr_b),
    .clock              (sys_clk),
    .data_a             (data_a),
    .data_b             (16'b0),
    .wren_a             (wren_a),
    .wren_b             (wren_b),
    .q_a                (q_a),
    .q_b                (q_b)
);

pll_sys_rst pll_sys_rst_inst
(
    .inclk              (inclk),
    .sys_clk            (sys_clk),
    .BCD_clk            (BCD_clk),
    .sys_rst            (sys_rst)
);
```

RAM 宽度 16 位，深度为 256。PLL 输入 50MHz 时钟，输出 100M 作为其他模块工作时钟，20M 用来驱动数码管显示。

第五步：数据生成计数器

```
always @ (posedge sys_clk)
if(sys_rst) begin
      counta     <= 0;
      user_id     <= 0;
end
else begin
      counta <=counta + 1;
      user_id <= sw;
end
```

第六步：写状态机

```
assign wr_done = (wr_len == (data_len - 1'b1));
//思考为什么采用 wr_len==data_len-1，而不是 wr_len==data_len
//第一段
always @ (posedge sys_clk)
begin
      if (sys_rst) begin
            sta = mema_idle;
      end
      else
            sta = sta_nxt;
end
//第二段
always @ (*)
begin
      case (sta)
            mema_idle     :     sta_nxt = mema_init;
            mema_init     :     sta_nxt = mema_pipe0;
            mema_pipe0    :     sta_nxt = mema_read0;
            mema_read0    :
            begin
                  if (!q_a[15])
                        sta_nxt = mema_read1;
                  else
                        sta_nxt = sta;
            end
            mema_read1    :
            begin
                  if (!q_a[15])
                        sta_nxt = mema_wr_data;
```

```
                else
                    sta_nxt = sta;
            end
            mema_wr_data    :
            begin
                if (wr_done)
                    sta_nxt = mema_wr_chsum;
                else
                    sta_nxt = sta;
            end
            mema_wr_chsum   :   sta_nxt = mema_wr_done;
            mema_wr_done    :   sta_nxt = mema_init;
            default   :   sta_nxt   =   mema_idle;
        endcase
end
```

```
//第三段
always @ (posedge sys_clk)
begin
    case (sta)
        mema_idle    :
        begin
            addr_a      <= 8'hff;
            wren_a      <= 1'b0;
            data_a      <= 16'b0;
            wr_len      <= 8'b0;
            wr_chsum    <= 0;
        end
        mema_init, mema_pipe0, mema_read0, mema_read1      :
        begin
            addr_a      <= 8'hff;
            wren_a      <= 1'b0;
            data_a      <= 16'b0;
            wr_len      <= 8'b0;
            wr_chsum    <= 0;
        end
        mema_wr_data    :
        begin
            addr_a      <= addr_a + 1'b1;
            wren_a      <= 1'b1;
            data_a      <= {countb, counta};
            wr_len      <= wr_len + 1'b1;
            wr_chsum    <= wr_chsum + {countb, counta};
        end
```

```
            mema_wr_chsum   :
            begin
                addr_a    <= addr_a + 1'b1;
                wr_len    <= wr_len + 1'b1;
                wren_a    <= 1'b1;
                data_a    <= (~wr_chsum) + 1'b1;
            end
            mema_wr_done    :
            begin
                addr_a    <= 8'hff;
                wren_a    <= 1'b1;
                data_a    <= {1'b1, user_id, wr_len};
            end
            default   :     ;
      endcase
end
```

写顺序：

1. 读取 8’hff 地址的标志（控制字），如果 valid=1’b0,程序进入下一步，否则等待
2. 地址加 1，8’hff+1 恰好为零，从 0 地址开始写数据并计算 checksum
3. 判断是否达到预定的数据长度，达到转下一步，否则继续写数据并计算 checksum
4. checksum 取补，写入存储器中
5. 在地址 8’hff 中写入控制字，封包

第七步：读状态机

```
//第一段
always @ (posedge sys_clk)
begin
      if (ext_rst) begin
            stb = memb_idle;
      end
      else
            stb = stb_nxt;
end
//第二段
always @ (*)
begin
      case (stb)
            memb_idle     :     stb_nxt = memb_init;
            memb_init     :     stb_nxt = memb_pipe0;
            memb_pipe0    :     stb_nxt = memb_read0;
            memb_read0    :
            begin
                  if (q_b[15])
```

```
                    stb_nxt = memb_read1;
                else
                    stb_nxt = memb_init;
            end
            memb_read1  :
            begin
                if (q_b[15])
                    stb_nxt = memb_rd_data;
                else
                    stb_nxt = memb_init;
            end
            memb_rd_data      :
            begin
                if(rd_done)
                    stb_nxt = memb_rd_chsum;
                else
                    stb_nxt = stb;
            end
            memb_rd_chsum   :     stb_nxt = memb_rd_done;

            memb_rd_done    :     stb_nxt = memb_init;
            default   :     stb_nxt = memb_idle;
        endcase
end
```

```
//第三段，实际操作需在时钟的边沿驱动下进行
always @ (posedge sys_clk)
begin
    case (stb)
        memb_idle    :
        begin
            addr_b          <= 8'hff;
            rd_data         <= 0;
            rd_chsum        <= 0;
            wren_b          <= 1'b0;
            rd_len          <= 8'b0;
            oID             <= 7'b0;
            rd_err          <= 1'b0;
        end
        memb_init    :
        begin
            addr_b          <= 8'hff;
            rd_data         <= 0;
            rd_chsum        <= 0;
```

```
            wren_b          <= 1'b0;
            rd_len          <= 8'b0;
            oID             <= 7'b0;
            rd_err          <= 1'b0;
        end
        memb_pipe0  :
        begin
            addr_b    <= 8'b0;
        end
        memb_read0  :
        begin
        if (q_b[15])
            addr_b          <= addr_b + 1'b1;
        else
            addr_b          <= 8'hff;
            rd_data         <= 0;
            rd_chsum        <= 0;
            wren_b          <= 1'b0;
            rd_len          <= 8'b0;
            oID             <= 7'b0;
        end
        memb_read1  :
        begin
        if(q_b[15])
            addr_b          <= addr_b + 1'b1;
        else
            addr_b          <= 8'hff;
            rd_data         <= 0;
            rd_chsum        <= 0;
            wren_b          <= 1'b0;
            rd_len          <= q_b[7:0];
            oID             <= q_b[14:8];
        end
        memb_rd_data      :
        begin
            addr_b          <= addr_b + 1'b1;
            rd_data         <= q_b;
            rd_chsum        <= rd_chsum + rd_data;
            wren_b          <= 1'b0;
            rd_len          <= rd_len - 1'b1;
        end
        memb_rd_chsum   :
        begin
            addr_b          <= 8'hff;
```

```
                wren_b          <= 1'b0;
                if (|rd_chsum)
                    rd_err      <= 1'b1;
            end
            memb_rd_done    :
            begin
                addr_b          <= 8'hff;
                wren_b          <= 1'b1;
            end
            default    :     ;
        endcase
end
```

读顺序

1. Idle 为复位后的状态
2. Init: 初始化,置地址为 8'hff
3. Rd_pipe0:加一个 latency（由于读地址和数据都经过锁存）。地址+1，形成读数据流水结构
4. Read0:置地址为 8'hff, 读控制字并判断 valid 位是否有效
   a. 如果 valid=1'b1,地址+1，跳到下一步
   b. 如果 valid=1'b0,表示数据包还没有准备就绪，地址设置成 8'hff，返回到 init 状态
5. Read1: 再次读控制字
   a. 如果 valid=1'b1,地址+1，ID 和数据长度赋给相应变量并跳到下一步
   b. 如果 valid=1'b0,表示数据包还没有准备就绪，地址设置成 8'hff, 返回到 init 状态
6. Rd_data：
   a. 读数据付给数据变量
   b. 计算 checksum，data_len-1
   c. 判读 data_len 是否为 0
      i. 0：已读完所有数据，跳下一步
      ii. 非 0，继续本状态的操作
7. rd_chsum：读 checksum 的值，并计算最后一步 checksum。判读数据的正确性，设置 rd_err 的标志
8. rd_done: 最后一步清除存储器中的 valid 标志，为下个数据包开放写使能

9.3 实验验证

第一步：分配管脚

管脚分配如表 9-2 所示。

表 9-2 帧数据读写实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
Inclk	C10_50MCLK	91	输入时钟

rst	KEY2	10	复位信号
sw[6]	SW6_LED6	76	开关输入 6
sw[5]	SW5_LED5	75	开关输入 5
sw[4]	SW4_LED4	74	开关输入 4
sw[3]	SW3_LED3	87	开关输入 3
sw[2]	SW2_LED2	86	开关输入 2
sw[1]	SW1_LED1	83	开关输入 1
sw[0]	SW0_LED0	80	开关输入 0

第二步：用 signaltap 观察双口 RAM 的读写结果

（1） 为了便于观察读写状态机协同结果，这里将数据长度改为 4，重新编译下载。长数据的测试，用户可以自己进行测试。

```
module frame_ram
#(parameter data_len=4)
(
input                   inclk,
input                   rst,
input          [6:0]    sw,
output    reg  [6:0]    oID,
output    reg           rd_done,
output    reg           rd_err
);
```

（2） Signaltap 的实测结果，如图 9-1

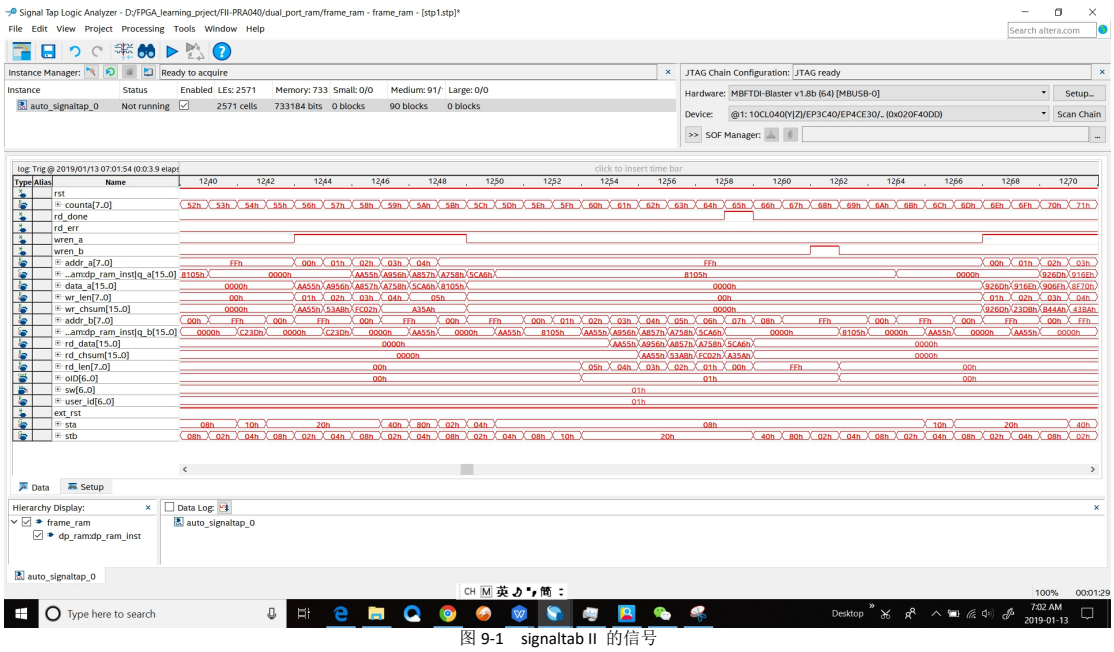


图 9-1 signaltab II 的信号

（3） 观察测试结果

1) 观察通过双端口 RAM 的握手机制

A. 是否实现写数据包封包后，读取才开始进行
B. 在读取整个数据包完成前，写数据包是否阻塞

2) 观察外部接口信号和状态
A. rd_done, rd_err
设置 rd_err=1，或上升沿为触发信号，观察是否捕捉到错误信号
B. 观察 wren_a,wren_b 信号与状态机跳转是否严格配合，达到设计要求

实验总结与思考

（1） 回顾设计要求，如何将一个实际需求经过分析，逐渐建立数字控制与状态机的模型并进行设计
（2） 在状态机的第三段修改成 if...else 模型并实现
（3） 注重思考如果读、写时钟不同，变成异步机制，如何控制握手
（4） 根据上面例子的理解，思考双口 RAM 如何在数据采集，异步通信，嵌入式 CPU 接口,DSP 芯片接口方面的应用
（5） 如何利用双口 RAM 构建 ITCM，DTCM 为将来设计 CPU 做准备

实验十　异步串行口设计与实验

10.1　实验目的

由于异步串行口在工业控制，通信以及软件调试中使用非常普遍，因此在 FPGA 开发中也是具有非常重要的地位

（1）　学习异步串行口通信的基本原理，握手机制，数据贞的结构
（2）　掌握异步采样技巧
（3）　复习数据包的贞结构
（4）　学习 FIFO 的使用
（5）　与 PC 的常用调试软件联合调试（SSCOM,teraterm 等）

10.2　实验要求

（1）　设计收发全双工异步通信接口 Tx，Rx
（2）　波特率 11520bps，8bit 数据，1 个起始位，1 个或 2 个停止位
（3）　接收缓冲（Rx FIFO）,发送缓冲（Tx FIFO）
（4）　形成数据包
（5）　数据包解析

10.3　实验内容

10.3.1　USB 转串口芯片（FT2232）简介

USB 是一种应用于 PC 领域的接口技术，用于规范电脑与外部设备的连接和通讯。USB 接口可支持设备的即插即用和热插拔功能，还可连接多达 127 种外设，如鼠标、调制解调器和键盘等。但目前还有大量外围设备和软件使用 RS232 串口标准，为了充分利用 USB 的优势，设计 USB-RS232 转换器以便通过 USB 总线传输 RS232 数据。这里提出一种基于 FT2-232H 的 USB-RS232 转换模块设计方案。

板载串口转换芯片为 FT2232，它可以实现 USB 到两个串行 UART、FIFO、Bit-Bang IO 接口，并且还可以配成 MPSSE，I2C，JTAG 及 SPI 总线。FT2232H 有 2 个独立的 UART / FIFO 控制器，用于控制异步数据、245 FIFO 数据、光电隔离(高速串口)或在安装程序命令时控制 Bit-Bang 模式；USB 协议引擎控制和管理 UTMI PHY 与 FIFO 之间的接口，负责电源管理和 USB 协议规范；双端口 FIFO TX 缓冲区(4KB)存储存储来自主机 PC 的数据，并通过多用途的 UART，FIFO 控制器使用数据；双端口 FIFO RX 缓冲区(4KB)存储来自多用途 UART/FOFO 控制器的数据，然后再将数据送至主机；复位发生器模块上电时为设备内部电路提供可靠的供电复位。RESETn 输入引脚允许外部设备重置 FT2232H。如果不使用 RESETn，应将其连接到 VCCIO(+3.3 V)；独立波特率发生器提供×16 或×10 的时钟信号给 UART，通过 1 个 14 位比例因子和 4 个寄存位提供最佳的波特率调谐，决定 UART 可编程设置的波特率。

10.3.2 硬件设计

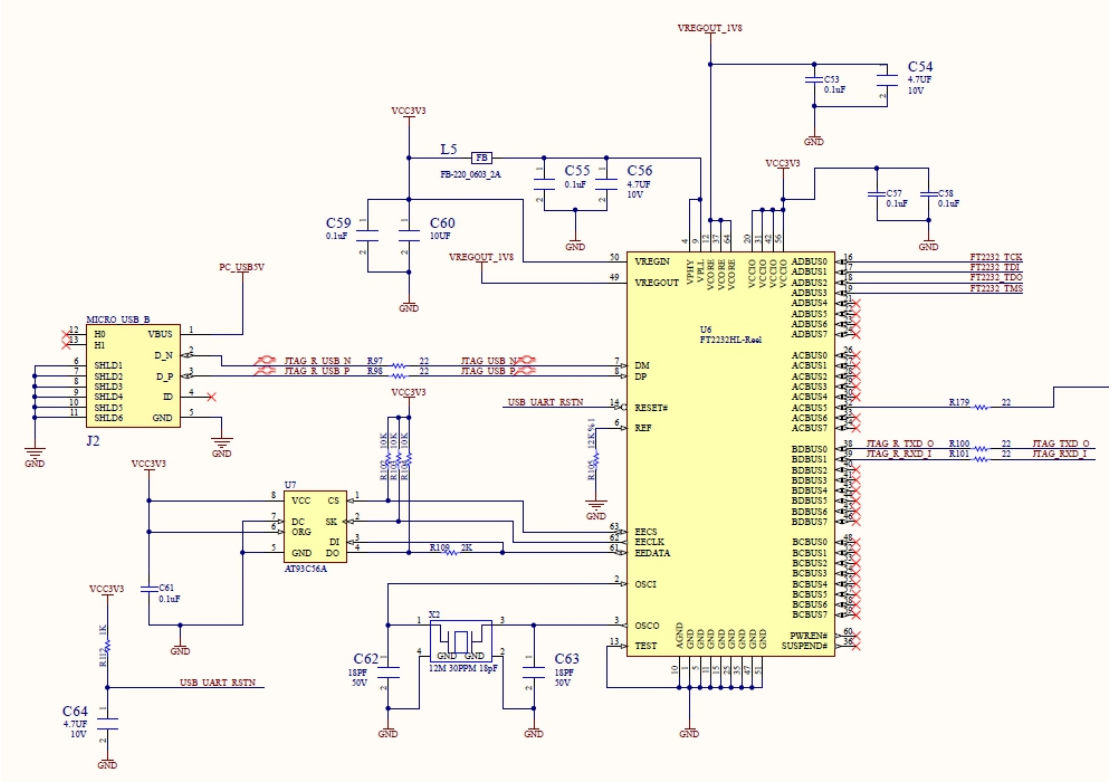


图 10-1 原理图串口部分

USB 串口转换的原理如图 10-1 所示，J2 为 USB 接口，在下载程序的时候，通过 USB_N,USB_P 两根线输入到 FT2232，经内部处理后由 JTAG_TXD_O 输入到 FPGA 中。在串口实验中，PC 端到 FPGA 端的发送路径与下载程序一致，FPGA 到 PC 端的接收路径是 FPGA 将数据由 JTAG_RXD_I 传输至 FT2232，经内部处理后经 USB 接口发送到 PC 端，以此实现串口通讯的接收与发送。

10.3.3 程序介绍

第一步：主程序架构建立

```
module uart_top
(
    input           inclk,
    input           rst,
    input   [1:0]   baud_sel,
    input           tx_wren,
    input           tx_ctrl,
    input   [7:0]   tx_data,
    input           tx_done,
    output          txbuf_rdy,

    input           rx_rden,
    output  [7:0]   rx_byte,
    output          rx_byte_rdy,
    output          sys_rst,
```

```
    output          sys_clk,
    input           rx_in,
    output          tx_out
);
```

这里有许多的握手信号，带 tx 前缀的为发送部分信号，带 rx 前缀的为接收部分信号。

第二步：新建波特率生成器文件

（1） 输入时钟 7.3728MHz(115200 的 64 倍）。实际为 7.377049MHz，这是由于 PLL 的系数是整数分频造成的，不过误差不大，在异步通信中可以通过停止位调节过来。如图 10-2 所示。

精细的解决方案

A. 采用两级 PLL 实现，得到更精细的频率

B. 停止位设为 2 位，可以有效消除误差

本次实验暂不做处理，默认输入频率为 7.3728MHz

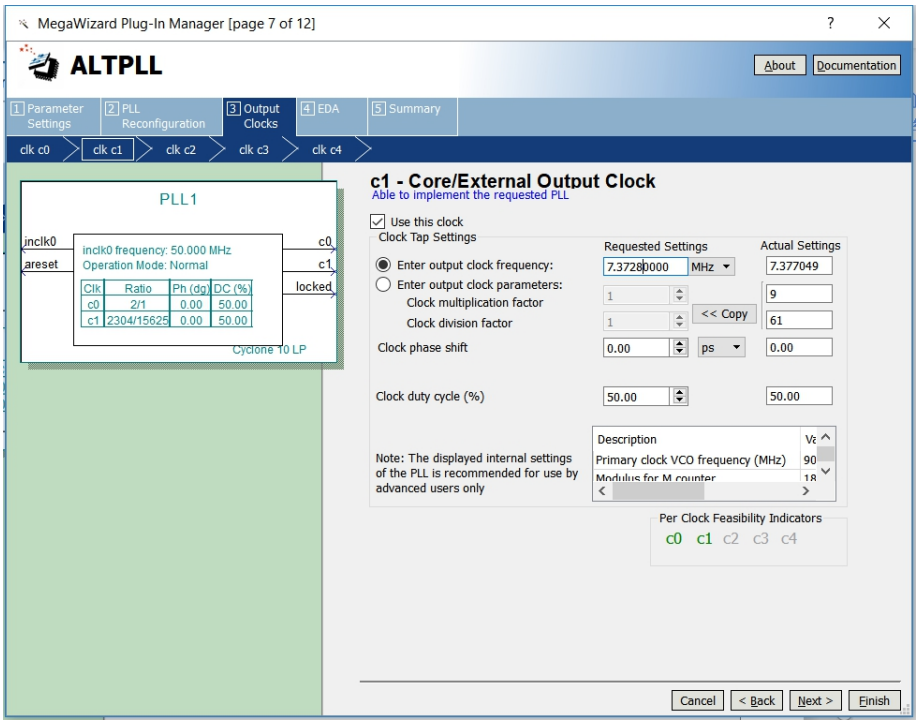


图 10-2 PLL 设置

C. 支持输出波特率 115200，57600，38400，19200 四种

D. 默认的波特率为 115200

（2） 波特率设计源文件

```
//发送波特率，时钟分频选择
wire [8:0] frq_div_tx;
assign frq_div_tx    =     (baud_sel == 2'b00) ? 9'd63:
                           (baud_sel == 2'b01) ? 9'd127:
                           (baud_sel == 2'b10) ? 9'd255:9'd511;
reg   [8:0] count_tx=9'd0;
always @ (posedge inclk)
```

```
if (rst) begin
      count_tx  <= 9'd0;
      baud_tx   <= 1'b0;
end
else begin
      if (count_tx == frq_div_tx) begin
            count_tx  <= 9'd0;
            baud_tx   <= 1'b1;
      end
      else begin
            count_tx  <= count_tx + 1'b1;
            baud_tx   <= 1'b0;
      end
end
```

设置了四个不同的档位对波特率进行选择，对应第二步（1）。接收部分波特率与发送部分一致类似。

第三步：设计发送缓冲区文件 tx_buf

（1） 8 位 FIFO，深度 256,读写时钟分离，写满标志，读空标志

（2） 接口与握手

1) rst 复位信号
2) wr_clk 写时钟
3) tx_clk 发送时钟
4) 8 位写数据 tx_data
5) wr_en 写使能
6) ctrl 写入的内容是数据还是控制字
7) rdy 缓冲区 ready，可以接受下一个数据贞

发送缓冲区例化文件

```
tx_buf
#(.TX_BIT_LEN(8),.STOP_BIT(2))
tx_buf_inst
(
.sys_rst              (sys_rst),
.uart_rst             (uart_rst),
.wr_clk               (sys_clk),
.tx_clk               (uart_clk),
.tx_baud              (tx_baud),
.tx_wren              (tx_wren),
.tx_ctrl              (tx_ctrl),
.tx_datain            (tx_data),
.tx_done              (tx_done),
.txbuf_rdy            (txbuf_rdy),
.tx_out               (tx_out)
```

```
);
```

（1） 串行发送、接口与握手文件设计

1) 接口设计

A. tx_rdy，发送空闲，可以接受新的 8bit 数据

B. tx_en，发送数据使能，传递给发送模块 8bit 数据使能信号

C. tx_data，待发送的 8bit 数据

D. tx_clk ，发送时钟

E. tx_baud， 发送波特率

2) 例化

```
tx_transmit
#(.DATA_LEN(TX_BIT_LEN),
   .STOP_BIT(STOP_BIT)
)
tx_transmit_inst
(
.tx_rst              (uart_rst),
.tx_clk              (tx_clk),
.tx_baud        (tx_baud),
.tx_en               (tx_en),
.tx_data        (tx_data),
.tx_rdy              (trans_rdy),
.tx_out              (tx_out)
);
```

（2） 编写激励文件对发送部分进行仿真。（tb_uart）

（3） 发送部分 modelsim 仿真波形，如图 10-3 所示。

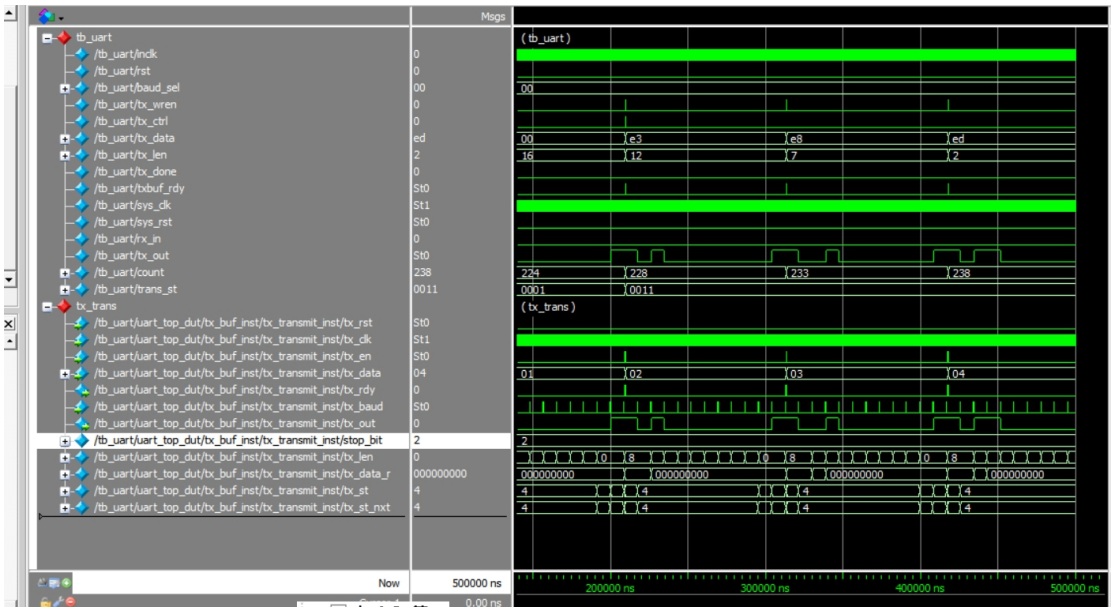


图 10-3 串口发送 moselsim 仿真波形

（4） 扩展设计（扩展内容只留给用户思考与练习，这里不在累述）

1) 设计发送器支持 5，6，7，8bit 的 PHY

2) 支持奇偶校验

（5） 以上各个步骤的设置涉及到 FIFO，PLL 等，参看 uart_top 的项目文件

第四步：UART 接受文件的设计

（1） rx_phy.v 的设计

1) 设计策略和步骤

A. 采用 8 倍采样：因此 rx_baud 与 tx_baud 是不同的，这里采样 rx_band=8*tx_band

B. 采用判断的方式实现接收数据的判决

在采样值计数完成后判断数据计数器是否大于 4

2) 接收数据的步骤：

A. 同步：是指如何从接收的 0101...找到起始位，sync_dtc

B. 接收起始位(start)

C. 循环接收 8bit 数据

D. 接收停止位（判断是一个停止位，还是两个停止位）

a. 判断停止位是否正确

i. 正确，跳转 B

ii. 错误，跳转 A，重新同步

b. 不判断，直接跳转 B，本设计采用不判断的方案

（2） rx_buf 的设计

1) 设计策略与步骤

A. 添加 256 深度，8 位的 fifo

a. 读写时钟分离

b. 异步清零（内部同步）

c. 数据先于出现 rdreq 出现在读端口

B. 步骤：

a. 初始化：fifo，rx_phy

b. 等待：FIFO 满信号（wrfull）为 0

c. 写：由 rx_phy 的：rx_phy_byte，rx_phy_rdy 触发

d. 写结束

e. 返回 b. 继续等待

f. rx_buf.v 源程序（参考工程文件）

g. 接收仿真激励

内容和步骤

i. tx,rx 回环测试(assign rx_in=tx_out)

ii. 继续使用 TX 部分的激励文件

iii. 编写 rx 的激励部分

h. Modelsim 仿真，如图 10-4 所示

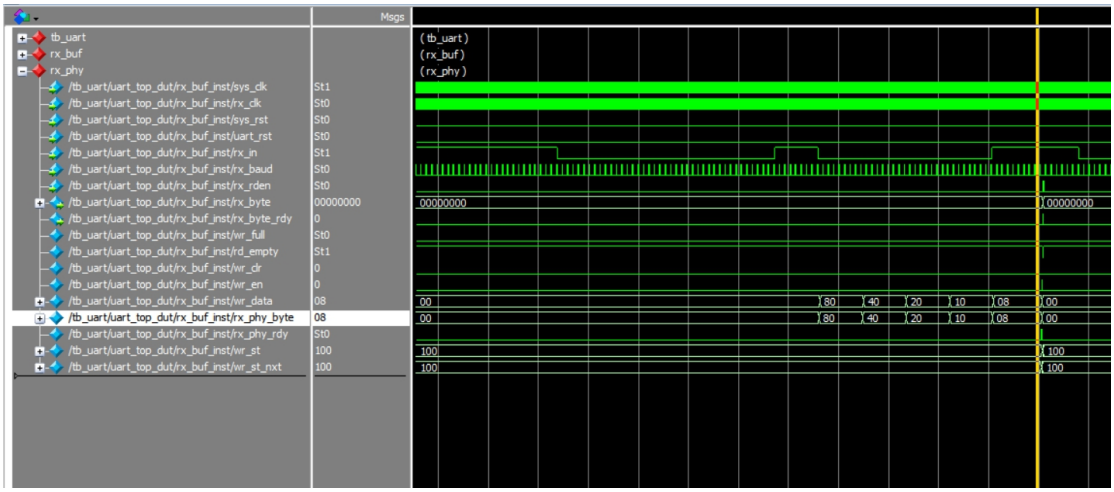


图 10-4　rx_phy 波形图

i. 思考与扩展

I. 修改程序完成 5、6、7、8bit 的设计

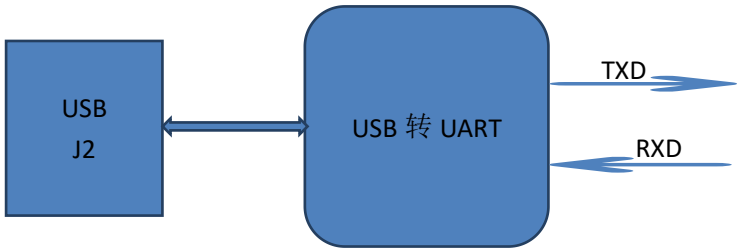
II. 完成接收端 rx_phy 的 start,stop 出错时的重新同步的设计

III. 完成 rx_buf 的收据贞的分析与打包

IV. 改判多采样为数据 180°对齐的采样方式进行设计，比较 FPGA 资源，时序以及数据恢复效果

10.4 实验验证

（1） 硬件接口，FII-PRA006/010 开发板已经集成 USB 到串口的转换



（2） FPGA 与 UART 管脚映射表 10-1 所示

表 10-1　fpga 与 uart 关键映射表

UART 名称	RXD	TXD
原理图名称	JTAG_TXD_O	JTAG_RXD_I
FPGA 管脚	144	143

（3） 编写硬件测试文件

1) 测试方案，开发板 J2 连接主机 USB 接口
2) 使用测试软件如 teraterm,SSCOM3 等，也可以自己编写一个串口通讯程序（C#,C++,JAVA,Python...）
3) PC 按照一定格式发送数据
4) 测试端采用计数器的方式生成一定格式的数据
5) 编写测试程序 hw_tb_uart，并将 uart_top 例化进去。
6) 将此程序设为顶层，例化之前的程序，即可下板验证。

（4） 分配管脚：

锁定管如表 10-2 所示

表 10-2 串行口实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
Inclk	C10_50MCLK	91	输入时钟
rst	KEY2	10	复位信号
rx_in	TAG_RXD_I	144	串口输入数据
tx_out	TAG_TXD_O	143	串口发送数据

（5） 实验现象

1) 观察 PC 收到的数据，如图 10-5 所示。

2) 用 signaltapII 观察 FPGA 收到的数据

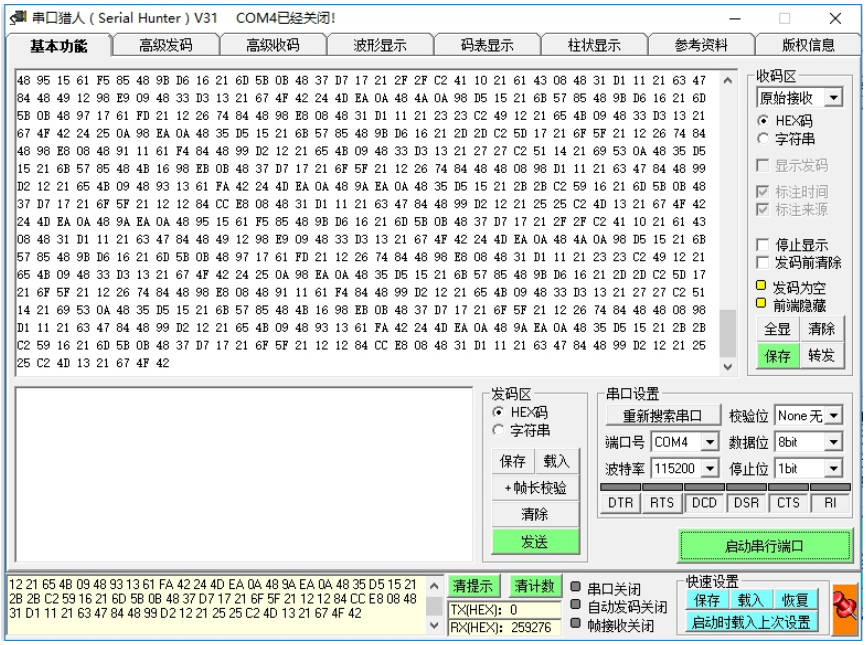


图 10-5 发送数据在上位机上显示实验现象

（6） 接收部分请同学们自己动手练习。

实验十一 IIC 协议传输

11.1 实验目的

（1） 学习异 IIC 总线的基本原理，掌握 IIC 通信协议
（2） 掌握读写 EEPROM 的方法
（3） 使用逻辑分析仪联合调试

11.2 实验要求

（1） 正确通过 FPGA 往 EEPROM 中任意一个地址（本例程写入 8'h03 地址的寄存器）写入一个数字（本例程通过（SW7~SW0）改变写入的 8bit 数据值）。写入之后，并且成功读出该写入数据。读出数据直接显示在数码管上。
（2） 下载程序进入 FPGA 按 LEFT 键,执行数据写入 eeprom 操作，按 RIGHT 键读出刚才写入数据。
（3） 通过读取数码管上显示的值判断读取的值正确与否。数码管显示值与写入值一致，则正确，否则不正确。
（4） 用 Signaltab II 分析内部数据的正确性，与数码管的显示情况对比验证。

11.3 实验内容

11.3.1 EEPROM 简介与 IIC 协议简介

（1） EEPROM 简介

EEPROM（Electrically Erasable Programmable read only memory）是指带电可擦可编程只读寄存器。是一中掉电后数据不丢失的存储芯片。

在实验板上有一个 IIC 接口的的 EEPROM 芯片 24LC02，容量大小为 256bytes。由于其掉电后数据不丢失的特性，用户可以用来存储一些硬件设置数据或者用户信息。

（2） IIC 的整体时序协议如下

1) 总线空闲状态：SDA，SCL 均为高电平
2) IIC 协议开始：SCL 保持高电平，SDA 则由高电平跳转为低电平，产生一个起始信号
3) IIC 读写数据阶段：包括数据的串行输入输出和数据接收方发出的应答型号
4) IIC 的传输结束位：SCL 为高电平，SDA 由低电平跳转为高电平，产生一个结束标志位。具体的时序图如下图 11-1 所示
5) SDA 在 SCL 为高时必须保持不变，仅 SCL 为低时改变

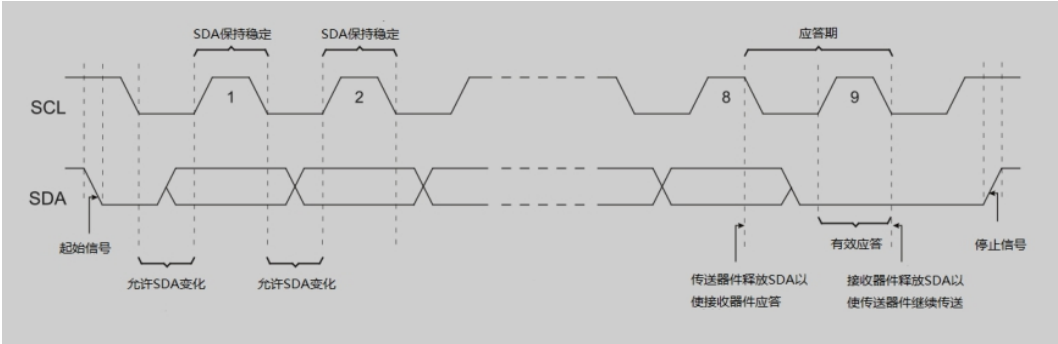


图 11-1　IIC 协议时序图

11.3.2　硬件介绍

每个 IIC 器件都有一个器件地址，有的器件地址在出厂时，就由厂家固定好了（具体数据可以找厂家的数据手册），有的确定了高几位，低位可以由使用者自行依据使用需求配置。开发板所采用的芯片 EEPROM 芯片 24LC02 的高四位地址已由元器件厂商固定为 1010，低三位在开发板链接如下图，所以器件地址为 1010000。IIC 器件原理图（eeprom）如图 11-2 所示。

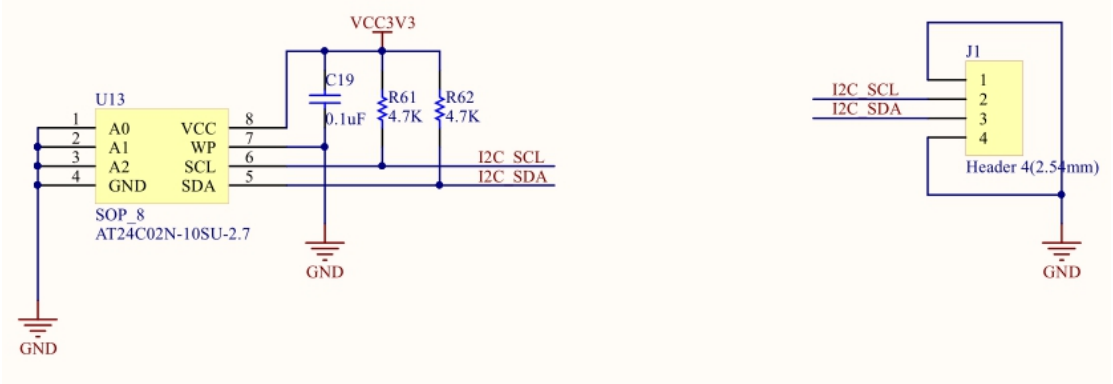


图 11-2　IIC 器件原理图 eeprom 部分

Eeprom 通过 I2C_SCL 时钟线与 I2C_SDA 数据线两根线与 FPGA 实现数据的读写。

11.3.3　程序介绍

本次实验共有两个主要模块，I2C 的读写模块与 LED 的显示模块；这里主要介绍前者。

第一步：主程序框架的建立

```
module iic_com(
    input               clk,
    input               rst,
    input       [7:0]   data,
    input               sw1,sw2,
    inout               scl,
    inout               sda,
    output  reg         iic_done,
    output      [7:0]   dis_data
);
```

输入的 8 位数据 data 是需要写入 Eeprom 中的数据，有 8 位拨码开关提供。

第二步：创建时钟 I2C_CLK

```
reg         [2:0]       cnt;
reg         [8:0]       cnt_delay;
reg                     scl_r;
reg                     scl_link ;
always @ (posedge clk or posedge rst)
begin
    if (rst)
        cnt_delay <= 9'd0;
    else if (cnt_delay == 9'd499)
        cnt_delay <= 9'd0;
    else
        cnt_delay <= cnt_delay + 1'b1;
end
always @ (posedge clk or posedge rst)
begin
    if (rst)
        cnt <= 3'd5;
    else begin
        case (cnt_delay)
            9'd124:   cnt <= 3'd1;        //cnt=1:scl
            9'd249:   cnt <= 3'd2;        //cnt=2:scl
            9'd374:   cnt <= 3'd3;        //cnt=3:scl
            9'd499:   cnt <= 3'd0;        //cnt=0:scl
            default: cnt<=3'd5;
        endcase
    end
end

`define SCL_POS         (cnt==3'd0)         //cnt=0:scl
`define SCL_HIG         (cnt==3'd1)         //cnt=1:scl
`define SCL_NEG         (cnt==3'd2)         //cnt=2:scl
`define SCL_LOW         (cnt==3'd3)         //cnt=3:scl

always @ (posedge clk or posedge rst)
begin
    if (rst)
        scl_r <= 1'b0;
    else if (cnt == 3'd0)
        scl_r <= 1'b1;
    else if (cnt == 3'd2)
        scl_r <= 1'b0;
```

```
end

assign scl = scl_link ? scl_r: 1'bz ;
```

首先，用系统 50MHZ 时钟分频得到一个周期为 10us 的 100KHz 时钟，作为 I2C 协议的传输时钟，然后通过计数器，定义了时钟的上升沿，高状态，下降沿和低状态，为后续数据的读写以及 I2C 协议的开始于结束做准备。最后一句话的意思是定义一个数据有效信号，只有当信号为高，即数据有效的时候，I2C 时钟再回有效，否则为高阻态。这也是按照 I2C 传输协议设置的。

第三步：I2C 传输的具体实现

```
`define    DEVICE_READ      8'b1010_0001
`define    DEVICE_WRITE     8'b1010_0000
`define    BYTE_ADDR        8'b0000_0011

parameter     IDLE      = 4'd0;
parameter     START1    = 4'd1;
parameter     ADD1      = 4'd2;
parameter     ACK1      = 4'd3;
parameter     ADD2      = 4'd4;
parameter     ACK2      = 4'd5;
parameter     START2    = 4'd6;
parameter     ADD3      = 4'd7;
parameter     ACK3      = 4'd8;
parameter     DATA      = 4'd9;
parameter     ACK4      = 4'd10;
parameter     STOP1     = 4'd11;
parameter     STOP2     = 4'd12;

reg       [7:0]     db_r;
reg       [7:0]     read_data;
reg       [3:0]     cstate;
reg                 sda_r;
reg                 sda_link;
reg       [3:0]     num;

always @ (posedge clk or posedge rst)
begin
    if (rst) begin
        cstate          <= IDLE;
        sda_r           <= 1'b1;
        scl_link    <= 1'b1;
        sda_link    <= 1'b1;
        num             <= 4'd0;
```

```
        read_data       <= 8'b0000_0000;
        cnt_5ms         <= 20'h00000;
        iic_done  <= 1'b0;
    end
    else case (cstate)
        IDLE :
        begin
            sda_link  <= 1'b1;
            scl_link    <= 1'b1;
            iic_done  <= 1'b0 ;
            if (sw1_r || sw2_r) begin
                db_r<= `DEVICE_WRITE;
                cstate      <= START1;
            end
            else cstate <= IDLE;
        end
        START1    :
        begin
            if (`SCL_HIG) begin
                sda_link  <= 1'b1;
                sda_r             <= 1'b0;
                num               <= 4'd0;
                cstate            <= ADD1;
            end
            else cstate <= START1;
        end
        ADD1      :
        begin
            if (`SCL_LOW) begin
                if (num == 4'd8) begin
                    num               <= 4'd0;
                    sda_r             <= 1'b1;
                    sda_link  <= 1'b0;
                    cstate            <= ACK1;
                end
                else begin
                    cstate            <= ADD1;
                    num               <= num + 1'b1;
                    case (num)
                        4'd0 : sda_r <= db_r[7];
                        4'd1 : sda_r <= db_r[6];
                        4'd2 : sda_r <= db_r[5];
                        4'd3 : sda_r <= db_r[4];
                        4'd4 : sda_r <= db_r[3];
```

```
                        4'd5 : sda_r <= db_r[2];
                        4'd6 : sda_r <= db_r[1];
                        4'd7 : sda_r <= db_r[0];
                        default : ;
                    endcase
                end
            end
            else cstate <= ADD1;
        end
        ACK1    :
        begin
            if (`SCL_NEG) begin
                cstate    <= ADD2;
                db_r<= `BYTE_ADDR;
            end
            else cstate    <= ACK1;
        end
        ADD2    :
        begin
            if (`SCL_LOW) begin
                if (num == 4'd8) begin
                    num          <= 4'd0;
                    sda_r        <= 1'b1;
                    sda_link   <= 1'b0;
                    cstate       <= ACK2;
                end
                else begin
                    sda_link <= 1'b1;
                    num <= num+1'b1;
                    case (num)
                        4'd0 : sda_r <= db_r[7];
                        4'd1 : sda_r <= db_r[6];
                        4'd2 : sda_r <= db_r[5];
                        4'd3 : sda_r <= db_r[4];
                        4'd4 : sda_r <= db_r[3];
                        4'd5 : sda_r <= db_r[2];
                        4'd6 : sda_r <= db_r[1];
                        4'd7 : sda_r <= db_r[0];
                        default : ;
                    endcase
                        cstate <= ADD2;
                    end
                end
            else cstate <= ADD2;
```

```
            end
            ACK2      :
            begin
                if (`SCL_NEG) begin
                    if (sw1_r) begin
                        cstate      <= DATA;
                        db_r<= data_tep;
                    end
                    else if (sw2_r) begin
                        db_r<= `DEVICE_READ;
                        cstate      <= START2;
                    end
                end
                else cstate <= ACK2;
            end
            START2    :
            begin
                if (`SCL_LOW) begin
                    sda_link   <= 1'b1;
                    sda_r            <= 1'b1;
                    cstate           <= START2;
                end
                else if (`SCL_HIG) begin
                    sda_r            <= 1'b0;
                    cstate           <= ADD3;
                end
                else cstate          <= START2;
            end
            ADD3      :
            begin
                if (`SCL_LOW) begin
                    if (num == 4'd8) begin
                        num              <= 4'd0;
                        sda_r            <= 1'b1;
                        sda_link   <= 1'b0;
                        cstate           <= ACK3;
                    end
                    else begin
                        num <= num + 1'b1;
                        case (num)
                            4'd0 : sda_r <= db_r[7];
                            4'd1 : sda_r <= db_r[6];
                            4'd2 : sda_r <= db_r[5];
                            4'd3 : sda_r <= db_r[4];
```

```
                        4'd4 : sda_r <= db_r[3];
                        4'd5 : sda_r <= db_r[2];
                        4'd6 : sda_r <= db_r[1];
                        4'd7 : sda_r <= db_r[0];
                        default: ;
                    endcase

                end
            end
            else cstate <= ADD3;
        end
        ACK3    :
        begin
            if (`SCL_NEG) begin
                cstate      <= DATA;
                sda_link    <= 1'b0;
            end
            else cstate     <= ACK3;
        end
        DATA    :
        begin
            if (sw2_r) begin
                if (num <= 4'd7) begin
                    cstate <= DATA;
                    if(`SCL_HIG) begin
                        num <= num + 1'b1;
                        case (num)
                            4'd0 : read_data[7] <= sda;
                            4'd1 : read_data[6] <= sda;
                            4'd2 : read_data[5] <= sda;
                            4'd3 : read_data[4] <= sda;
                            4'd4 : read_data[3] <= sda;
                            4'd5 : read_data[2] <= sda;
                            4'd6 : read_data[1] <= sda;
                            4'd7 : read_data[0] <= sda;
                            default: ;
                        endcase
                    end
                end
                else if((`SCL_LOW) && (num == 4'd8)) begin
                    num     <= 4'd0;
                    cstate  <= ACK4;
                end
                else cstate     <= DATA;
```

```
            end
            else if (sw1_r) begin
                sda_link <= 1'b1;
                if (num <= 4'd7) begin
                    cstate <= DATA;
                    if (`SCL_LOW) begin
                        sda_link   <= 1'b1;
                        num            <= num + 1'b1;
                        case (num)
                            4'd0 : sda_r <= db_r[7];
                            4'd1 : sda_r <= db_r[6];
                            4'd2 : sda_r <= db_r[5];
                            4'd3 : sda_r <= db_r[4];
                            4'd4 : sda_r <= db_r[3];
                            4'd5 : sda_r <= db_r[2];
                            4'd6 : sda_r <= db_r[1];
                            4'd7 : sda_r <= db_r[0];
                            default: ;
                        endcase
                    end
                end
                else if ((`SCL_LOW) && (num==4'd8)) begin
                    num            <= 4'd0;
                    sda_r          <= 1'b1;
                    sda_link   <= 1'b0;
                    cstate         <= ACK4;
                end
                else cstate <= DATA;
            end
        end
        ACK4    :
        begin
            if (`SCL_NEG)
                cstate <= STOP1;
            else
                cstate <= ACK4;
        end
        STOP1   :
        begin
            if (`SCL_LOW) begin
                sda_link   <= 1'b1;
                sda_r          <= 1'b0;
                cstate         <= STOP1;
            end
```

```
                else if (`SCL_HIG) begin
                    sda_r           <= 1'b1;
                    cstate          <= STOP2;
                end
                else cstate         <= STOP1;
            end
            STOP2    :
            begin
                if (`SCL_NEG) begin
                    sda_link  <= 1'b0;
                    scl_link  <= 1'b0;
                end
                else if (cnt_5ms==20'h3fffc) begin
                    cnt_5ms         <= 20'h00000;
                    iic_done  <= 1;
                    cstate          <= IDLE;
                end
                else begin
                    cstate          <= STOP2;
                    cnt_5ms         <= cnt_5ms + 1'b1;
                end
            end
            default: cstate <= IDLE;
        endcase
end
```

整个过程采用状态机实现。复位时为空闲状态 IDLE，同时数据线 sda_r 拉高，时钟和数据都有效，即 scl_link，sda_link 为高；计数器 num 清零，读出数据 read_data 为 0，5ms 延时计数器清零，I2C 传输结束信号 iic_done 为低无效。

（1） 空闲状态 IDLE：当接收到读使能或者写使能信号 sw1_r || sw2_r，将写控制字赋值给中间变量 db_r <= `DEVICE_WRITE，并跳转到起始状态START1；

（2） 起始状态 START1：在时钟信号为高德状态下将数据线拉低，产生 I2C 传输的起始信号，并跳转到器件地址状态 ADD1；

（3） 器件地址状态 ADD1：将写控制字（器件地址加一位 0）按 MSB（高位优先）传输完成后，将 sda_link 拉低释放数据总线为高阻态，并跳转到第一次应答状态 ACK1，等待从机（EEPROM）的应答信号

（4） 第一次应答状态 ACK1：如果数据线被拉低则证明从机正常接收数据，否则数据没有写进到 Eeprom 中去，接下来是重新写还是停止由实验者决定，这里没有暂时不做任何判断及处理，直接跳转至写寄存器地址状态 ADD2，并将数据写入的地址 BYTE_ADDR 赋值给中间变量（本次实验将数据写入第三个寄存器中，即 BYTE_ADDR = 0000_0011）

（5） 寄存器地址状态 ADD2：与（3）相同，将寄存器地址传输给从机并跳转到第二次应答状态 ACK2

（6） 第二次应答状态 ACK2：此时急需要判断，如果是写状态 sw1，则跳转至数据传输状态 DATA，同时将写入的数据赋值给中间变量，如果是读状态 sw2，则跳转至第二次

开始状态 START2，并将读控制字赋值给中间变量

（7） 第二次开始状态 START2：与（2）相同产生一个起始信号，并跳转到读寄存器地址状态 ADD3

（8） 读寄存器地址状态 ADD3：将需要读出数据的寄存器地址传输完毕后跳转至第三次应答状态 ACK3S

（9） 第三次应答状态 ACK3：直接跳转到数据传输状态 DATA，在读状态下，需要读出的数据会紧跟着寄存器地址直接被读出来

（10） 数据传输状态 DATA：这里需要判断，如果是读状态，则直接将数据输出，如果是写状态则将需要写入的数据传输给数据线 SDA，两种状态都需要跳转至第四次应答状态 ACK4

（11） 第四次应答状态 ACK4：直接跳转至停止传输转态 STOP1

（12） 停止传输转态 STOP1：在时钟线为高的情况下拉高数据线，产生一个停止信号，并跳转到传输完成状态 STOP2

（13） 传输完成状态 STOP2：将时钟线和数据线全部释放，并经过 5ms 延时后重新回到空闲状态 IDLE 等待下一次传输指令的到来。这是因为 Eeprom 规定连续两次读写操作之间的间隔不得小于 5ms。

11.4 实验验证

第一步：分配管脚

管脚分配如表 11-1 所示

表 11-1 IIC 协议传输实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
clk	C10_50MCLK	91	系统时钟 50M
rst	KEY3	11	复位,默认低
sm_db[0]	SEG_PA	132	数码管段选 a
sm_db [1]	SEG_PB	137	数码管段选 b
sm_db [2]	SEG_PC	133	数码管段选 c
sm_db [3]	SEG_PD	125	数码管段选 d
sm_db [4]	SEG_PE	126	数码管段选 e
sm_db [5]	SEG_PF	138	数码管段选 f
sm_db [6]	SEG_PG	135	数码管段选 g
sm_db [7]	SEG_DP	125	数码管段选 h
sm_cs1_n	SEG_3V3_D1	142	数码管段选 2
sm_cs2_n	SEG_3V3_D0	136	数码管段选 1
data[0]	SW0	80	拨码开关输入
data[1]	SW1	83	拨码开关输入
data[2]	SW2	86	拨码开关输入
data[3]	SW3	87	拨码开关输入
data[4]	SW4	74	拨码开关输入
data[5]	SW5	75	拨码开关输入
data[6]	SW6	76	拨码开关输入
data[7]	SW7	77	拨码开关输入

sw1	KEY0	3	写入 eeprom 按键
sw2	KEY1	7	读出 eeprom 按键
scl	I2C_SCL	R20	eeprom 随路时钟
sda	I2C_SDA	R21	eeprom 数据线

第二步：下板验证

管脚分配完成后进行全编译，通过后进行下板验证。

程序下板后，按 LEFT 键将 SW7~SW0 所代表的 8bit 值，写入 eeprom.再按 RIGHT 键读取刚才写入位置的的值。观察实验板上数码管显示的值与写入 eeprom 地址 8'h03 寄存器的值（SW7~SW0）的一致性(本次实验写入 8'h34)。读出来的值显示在数码管上。实验现象如图 11-3 所示。

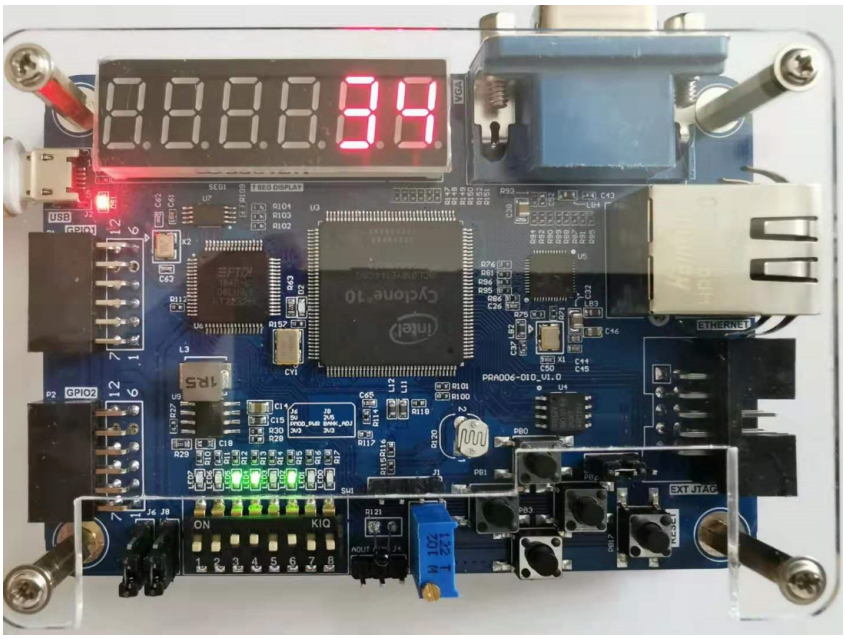


图 11-3　EEPROM 实验现象

实验十二　AD、DA 实验

12.1　实验目的

由于在现实世界中，所有的自然存在的信号都是模拟信号，而在实际工程中读取并且处理的都为数字信号。所以存在一个自然信号和工信号相互转化的过程（数模转化：DAC，，模数转化：ADC）。本实验的目的如下：

（1）　学习 AD 转化的相关理论

（2）　复习上一个实验学习到的 IIC 协议知识，对实验板上的 PCF8591 写入数据

（3）　从 PCF8591 中读取 AD 采集的值，并且将取到的值转化为实际值，用数码管显示来

12.2　实验要求

（1）　利用该芯片的 ADC 口进行模数转换，正确配置芯片，采集实验板上三个可变电阻（电位器，光敏电阻，热敏电阻）变化，并将采集的电压值经过数码管显示出来

（2）　下板验证，与电阻特性进行对比，验证结果的正确性

12.3　实验内容

12.3.1　AD 转换芯片（PCF8591）简介

PCF8591 是一个单片集成、单独供电、低功耗、8-bit CMOS 数据获取器件。PCF8591 具有 4 个模拟输入、1 个模拟输出和 1 个串行 I²C 总线接口。PCF8591 的 3 个地址引脚 A0, A1 和 A2 可用于硬件地址编程，在 PCF8591 器件上输入输出的地址、控制和数据信号都是通过双线双向 I2C 总线以串行的方式进行传输。有关 IIC 总线协议的内容请参考上实验十一。发送完器件地址信息和读写控制字后，就发送控制字信息。

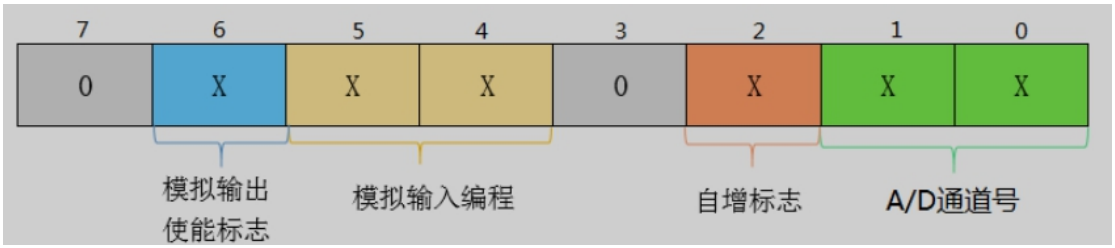


图 12-1　PCF8591 控制字示意图

具体的控制字信息如图 12-1 所示。D1-D0 用于四个通道设置，D2 自动增益选择，1 有效，D5-D4 决定模拟量输入选择，D0 为模拟输出使能，D7，D3 为预留项，为 0。发送到 PCF8951 的第二个字节将被存储到控制寄存器中，用于控制器件功能。控制寄存器的高半字节用于允许模拟输出，将和模拟输入编程为单端或差分输入。低半字节选择一个由高半字节定义的模拟输入通道。如果自动增量标志置 1，每次 A/D 转换后通道号将自动增加。

本次实验利用拨码开关（SW1，SW0）选择输入通道作为 AD 采集输入通道，具体通道信息如表 12-1 所示。把控制信息配置为 8'h40，即为模拟输出，并默认为 00 通道，即默认情况下显示的是光敏电阻电压值。

表 12-1　通道信息对照表

SW1，SW0	通道选择	采集对象
00	0	光敏电阻电压值
01	1	热敏电阻电压值
10	2	可调变阻器电压值

12.3.2　硬件设计

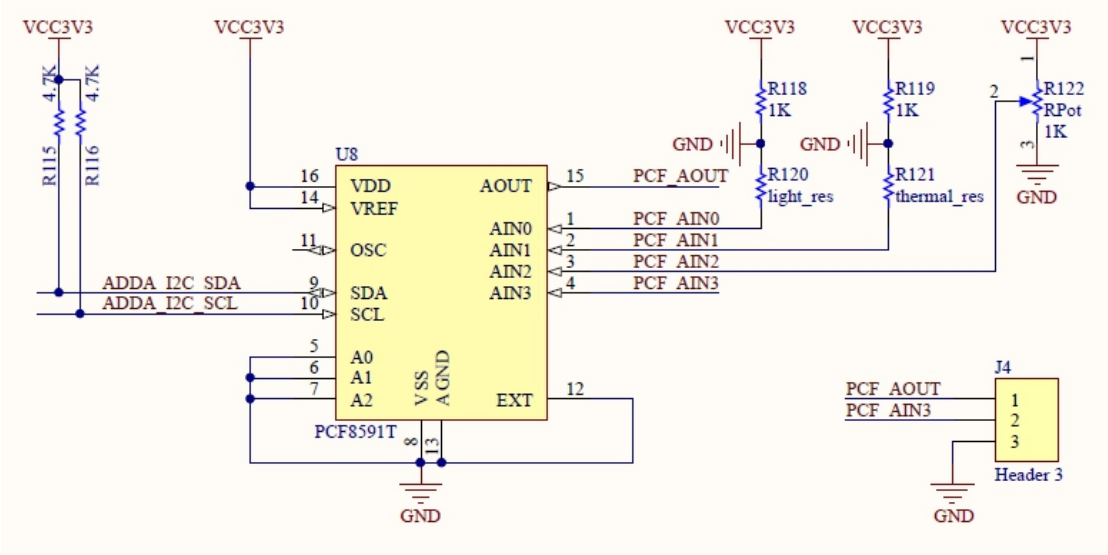


图 12-2　原理图 AD/DA 转换部分

用 PCF8591 进行 AD/DA 转换的原理图部分如图 12-2 所示。I2C 总线经过两个上拉电阻，在不工作的时候拉高。A0，A1，A2 接地，因此器件地址为 7’b1010000，模拟输入通道 AIN0 接光敏电阻，AIN1 接热敏电阻，AIN3 接电位器。当选择好通道后，FPGA 将通过数据总线 ADDA_I2C_SLC 把 PCF8591 中的值读到 FPGA 中进行运算处理。

12.3.3　程序介绍

本次实验同样是用 I2C 总线控制 PCF8951 芯片，因此程序与实验十一基本一致。本次只介绍不同的地方。

```
reg         [7:0]       read_data_temp [15:0];
reg         [11:0]      dis_data_temp;
always @ (posedge clk)
   dis_data_temp <= read_data_temp[0] + read_data_temp[1] + read_data_temp[2]
                    + read_data_temp[3] + read_data_temp[4] + read_data_temp[5]
                    + read_data_temp[6] + read_data_temp[7] + read_data_temp[8]
                    + read_data_temp[9] + read_data_temp[10] + read_data_temp[11]
                    + read_data_temp[12] + read_data_temp[13] + read_data_temp[14]
                    + read_data_temp[15];
always @ (posedge clk )
      dis_data <= dis_data_temp >> 4;
integer     i;
always @ (posedge clk or posedge rst)
begin
```

```
    if (rst) begin
        for (i=0;i<16;i=i+1)
            read_data_temp[i] <= 8'h00;
    end
    else if (iic_done) begin
        for (i=0;i<15;i=i+1)
            read_data_temp[i+1] <= read_data_temp[i];
            read_data_temp[0] <= read_data ;
        end
    else begin
        for (i=0;i<16;i=i+1)
            read_data_temp[i] <= read_data_temp[i];
    end
end
```

整个之一部分的作用，就是芯片在不断采集电阻两端的电压值的时候，由于一系列不稳定因素，会导致电压值的不稳定，因此输出的值将会出现较大误差，因此我们每次采集 16 组数据，然后求其平均值，将其结果输出作为此时电阻两端的电压值。再由电压值的变化，就可以判断组织的比那话规律。如光敏电阻，光强越大电压值越小，阻值越小，符合光敏电阻特性；热敏电阻温度越高电压值越小，组织越小，符合光敏电阻特性；电位器顺时针旋转，电压增大，阻值增大，逆时针旋转电压减小，阻值减小。

AD 芯片最总输出的是一个 8 位的数字量，但实质上他还不是我们所需要的电压值。他是将量程的电压值量化成 256 份（8 为二进制数可表示 256 个数），因此在数码管显示的时候，我们需要作进一步的计算和转换工作。

```
    parameter     V_REF = 12'd3300;
    reg           [19:0]    num_t;
    reg           [19:0]    num1;
    wire          [3:0]     data0;
    wire          [3:0]     data1;
    wire          [3:0]     data2;
    wire          [3:0]     data3;
    wire          [3:0]     data4;
    wire          [3:0]     data5;
    assign   data5 = num1 / 17'd100000;
    assign   data4 = num1 / 14'd10000 % 4'd10;
    assign   data3 = num1 / 10'd1000 % 4'd10 ;
    assign   data2 = num1 / 7'd100 % 4'd10 ;
    assign   data1 = num1 / 4'd10 % 4'd10 ;
    assign   data0 = num1 % 4'd10;

always @ (posedge clk)
  num_t <= V_REF * dis_data;
```

```
always @(posedge clk or posedge rst)
begin
    if(rst) begin
        num1 <= 20'd0;
    end
    else
        num1 <= num_t >> 4'd8;
end
```

我们的 VCC 为 3.3V，因此电阻电压最大 3.3V，所以将 8 位数据 dis_data 乘 3300 扩大 1000 倍赋值给 num_t，方便显示和观察，num_t 再缩小 256 倍（左移 8 位）赋值给 num1 与 PCF8951 量化的 256 份相对应，此时的 num1 就是此时电阻两端赌赢的电压值的 1000 倍。我们将他的每一位显示到数码管上，有高位到低位（data5 到 data0），在千位时（data3）加上小数点，则此时数码管山显示的值就是此时电阻两端的电压值，并精确到小数点后 3 位。

12.4 实验验证

第一步：分配管脚

管脚分配如表 12-2 所示。

表 12-2 AD 转换实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
clk	C10_50MCLK	91	系统时钟 50M
rst	KEY3	10	复位，默认低
sm_db[0]	SEG_PA	132	数码管段选 a
sm_db [1]	SEG_PB	137	数码管段选 b
sm_db [2]	SEG_PC	133	数码管段选 c
sm_db [3]	SEG_PD	125	数码管段选 d
sm_db [4]	SEG_PE	126	数码管段选 e
sm_db [5]	SEG_PF	138	数码管段选 f
sm_db [6]	SEG_PG	135	数码管段选 g
sm_db [7]	SEG_DP	128	数码管段选 h
data[0]	SW0	80	拨码开关输入
data[1]	SW1	83	拨码开关输入
data[2]	SW2	86	拨码开关输入
data[3]	SW3	87	拨码开关输入
data[4]	SW4	74	拨码开关输入
data[5]	SW5	75	拨码开关输入
data[6]	SW6	76	拨码开关输入
data[7]	SW7	77	拨码开关输入
scl	ADDA_I2C_SCL	53	PCF8591 时钟线
sda	ADDA_I2C_SDA	52	PCF8591 数据线
sel[0]	SEG_3V3_D0	136	数码管位选
sel[1]	SEG_3V3_D1	142	数码管位选

sel[2]	SEG_3V3_D2	141	数码管位选
sel[3]	SEG_3V3_D3	129	数码管位选
sel[4]	SEG_3V3_D4	127	数码管位选
sel[5]	SEG_3V3_D5	124	数码管位选

第二步：下板验证

管脚分配完成后进行全编译，通过后进行下板验证。

当默认状态下，即通道选择为 00 时，数码管显示当前环境亮度状态下，光敏电阻两端的电压值为 1.469V，如图 12-3 所示。

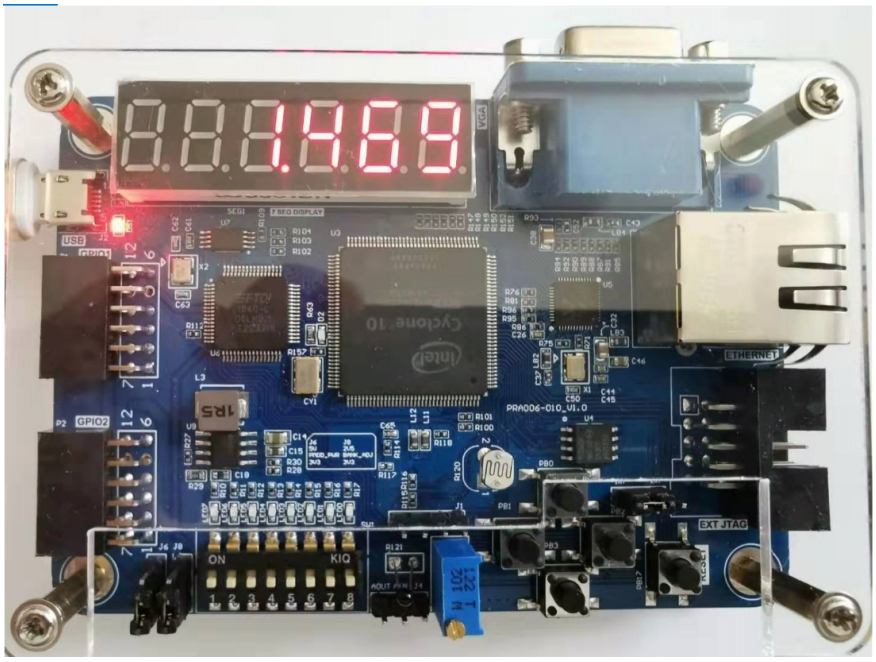


图 12-3　光敏电阻实验现象

当通道选择为 01 时，数码管显示当前环境温度下，热敏电阻两端的电压值为 2.926V，如图 12-4 所示。

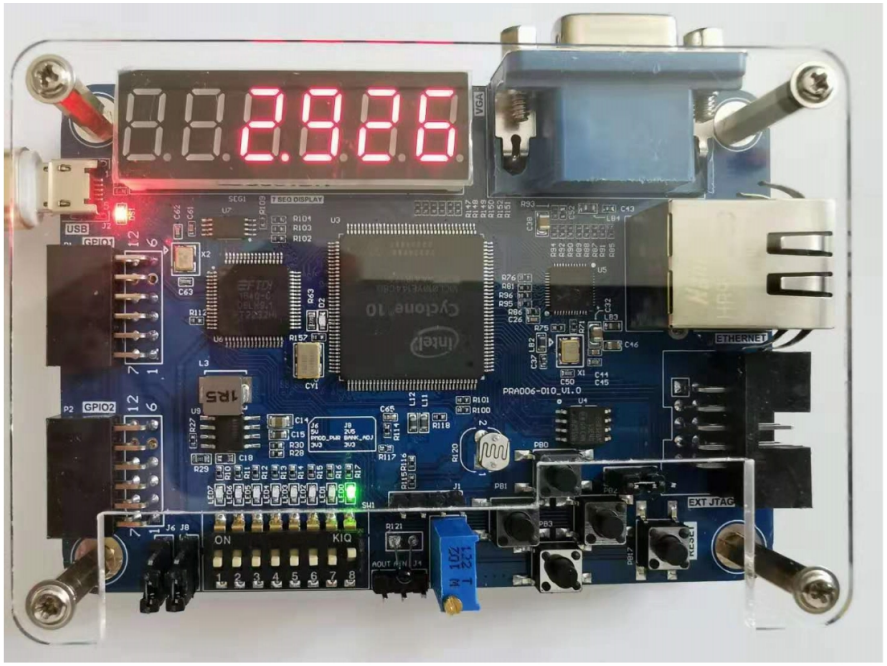


图 12-4　热敏电阻实验现象

当通道选择为 10 时，数码管显示当前阻值状态下，电位器两端的电压为 V，如图 12-5 所示。

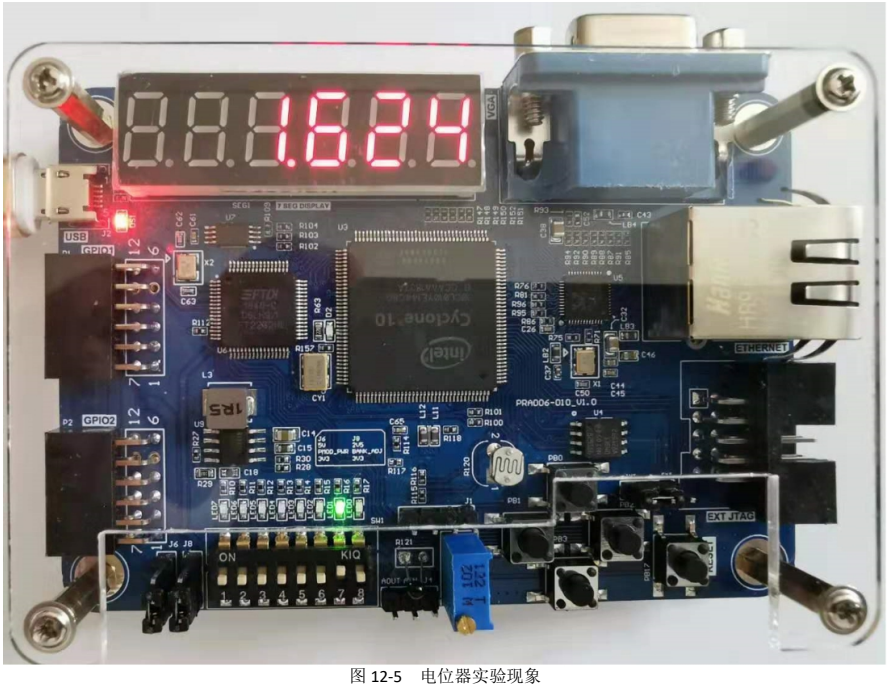


图 12-5　电位器实验现象

实验十三 VGA 实验

13.1 实验目的

（1） 掌握 VGA 实现原理
（2） 自己设计一个简单的 VGA 图像显示

13.2 实验要求

在屏幕上实现彩条状图形。

13.2 实验内容

13.2.1 VGA 简介

VGA（Video Graphics Array，视频图形阵列），是 IBM 于 1987 年提出的一个使用类比讯号的电脑显示标准。VGA 是最多制造商所共同支持的一个低标准，个人电脑在加载自己的独特驱动程式之前，都必须支持 VGA 的标准。

在显示器上 VGA 扫描方式分为逐行扫描和隔行扫描：逐行扫描是扫描从屏幕左上角一点开始，从左向右逐点扫描，每扫描完一行，电子束回到屏幕的左边下一行的起始位置，在这期间，CRT 对电子束进行消隐，每行结束时，用行同步信号进行同步；当扫描完所有的行，形成一帧，用场同步信号进行场同步，并使扫描回到屏幕左上方，同时进行场消隐，开始下一帧。隔行扫描是指电子束扫描时每隔一行扫一线，扫完一屏后再返回来扫描剩下的线，隔行扫描的显示器闪烁快速，可能会使使用者眼睛疲劳（本实验采用逐行扫描的方式），时序图如图 13-1，13-2 所示。

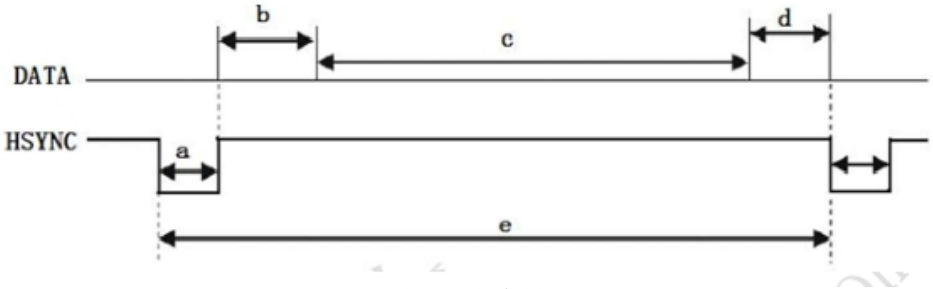


图 13-1 列同步时序

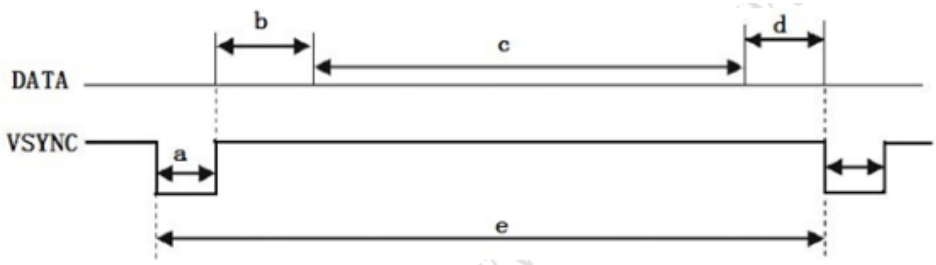


图 13-2 行同步时序

VGA 中定义行时序和列时序都需要同步脉冲（a 段），显示后沿（b 段），显示时序段（c 段）和显示前沿（d 段）四部分。VGA 工业标准显示模式要求：行同步、列同步都为负极性，即同步脉冲要求是负脉冲。由 VGA 行时序可知：每一行都有一个负极性行同步脉冲（a 段），是数据行的结束标志，同时也是下一行的开始标志。在同步脉冲之后为显示后沿（b 段），在显示时序段（c 段）显示器为亮的过程，RGB 数据驱动一行上的每一个像素点，从而显示一行。在一行的最后为显示前沿（d 段）。在显示时间段之外没有图像投射到屏幕，而是插入消隐信号。同步脉冲、显示后沿和显示前沿都是在行消隐间隔内，当消隐有效时，RGB 信号无效，屏幕不显示数据。

VGA 的列时序与行时序分析基本一致。

VGA 也有许多的显示标准，本次实验我们采用 640*480*60 的标准。标准概况如图 13-3 所示。

显示模式	时钟 (MHz)	列时序（列数）					行时序（行数）				
		a	b	c	d	e	a	b	c	d	e
640*480*60	25. 175	96	48	640	16	800	2	33	480	10	525
640*480*75	31. 5	64	120	640	16	840	3	16	480	1	500
800*600*60	40. 0	128	88	800	40	1056	4	23	600	1	628
800*600*75	49. 5	80	160	800	16	1056	3	21	600	1	625
1024*768*60	65	136	160	1024	24	1344	6	29	768	3	806
1024*768*75	78. 8	176	176	1024	16	1312	3	28	768	1	800
1280*1024*60	108. 0	112	248	1280	48	1688	3	38	1024	1	1066
1280*800*60	83. 64	136	200	1280	64	1680	3	24	800	1	828
1440*900*60	106. 47	152	232	1440	80	1904	3	28	900	1	932

图 13-3　VGA 显示标准

以本实验的显示标准 640*480*60Hz 为例。（640 为列数，480 为行数，60Hz 为刷新一屏的频率）。行时序：屏幕对应的行数为 525（a+b+c+d=e 段），其中 480（c 段）为显示行；每行均有行同步信号（a 段），为 2 个行周期的低电平。列时序：每个显示行包括 800 列（a+b+c+d=e 段），其中 640（c 段）为有效显示区，每一行有一个行同步信号（a 段）为 96 个行周期的低电平。

13.3.2　硬件设计

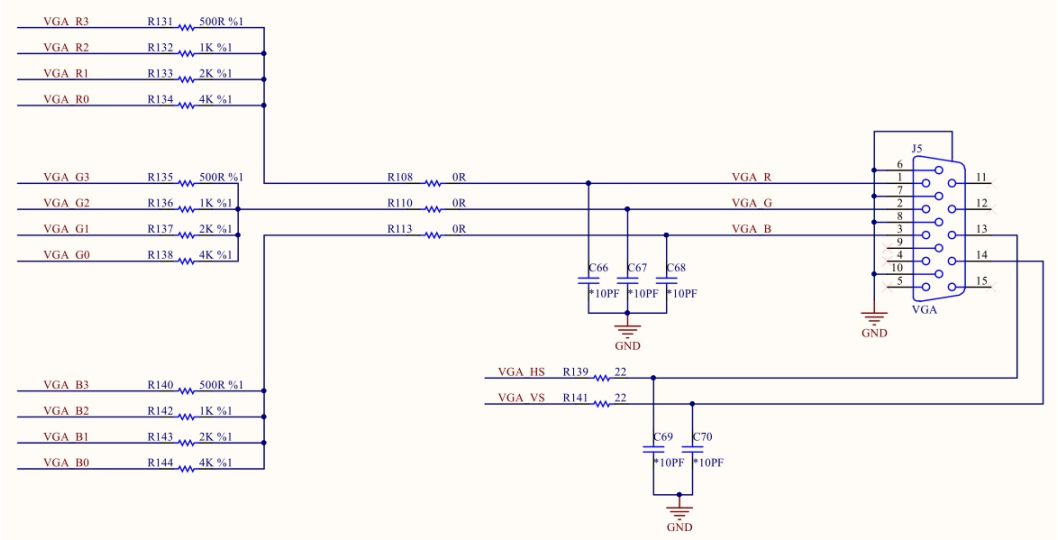


图 13-4　PCB 原理图

VGA 显示的原理图如图 13-4 所示。FPGA 直接控制 12 根数据线和两根控制线，VGA_R3-R0 代表红色分量，VGA_G3-G0 代表绿色分量，VGA_B3-B0 代表蓝色分量，以及行同步信号 VGA_HS 和场同步信号 VGA_VS。

13.3.3 程序介绍

第一步：主程序框架建立

```
module vga_test(
    input               clk,
    input               rst,
    output              vga_hs,
    output              vga_vs,
    output      [11:0]  vga_data
);
```

输入有 50MHz 时钟和复位信号，输出行同步信号 vga_has，场同步信号 vga_vs 和 12 位数据线 vga_data[11:0]。

第二步：时钟分频模块

按照我们的显示模式，采用锁相环输出时钟。

第三步：控制模块

控制模块的主要任务是让像素点显示到有效区域内。首先按照显示标准定义一些量。

```
parameter   H_SYNC    =   10'd96;
parameter   H_BACK    =   10'd48;
parameter   H_DISP    =   10'd640;
parameter   H_FRONT   =   10'd16;
parameter   H_TOTAL   =   10'd800;

parameter   V_SYNC    =   10'd2;
parameter   V_BACK    =   10'd33;
parameter   V_DISP    =   10'd480;
parameter   V_FRONT   =   10'd10;
parameter   V_TOTAL   =   10'd525;
```

带有 H 前缀对应图 13-3 中列时序的五段，值为每段像素点的个数，带有 V 前缀的对应图 13-3 中行时序的五段，值为每段的行数。

```
reg         [9:0]   cnt_h;
reg         [9:0]   cnt_v;
wire                vga_en;
wire                data_req;
assign vga_hs = (cnt_h <= H_SYNC - 1'b1) ? 1'b0 : 1'b1;
assign vga_vs = (cnt_v <= V_SYNC - 1'b1) ? 1'b0 : 1'b1;
```

```
assign vga_rgb = vga_en ? pixel_data : 16'd0;

assign vga_en = (((cnt_h >= H_SYNC+H_BACK) && (cnt_h < H_SYNC+H_BACK+H_DISP))
                        &&((cnt_v >= V_SYNC+V_BACK) && (cnt_v < V_SYNC+V_BACK+V_DISP)))
                        ?   1'b1 : 1'b0;
assign data_req = (((cnt_h >= H_SYNC+H_BACK-1'b1) && (cnt_h < H_SYNC
                       +H_BACK+H_DISP-1'b1))
                && ((cnt_v >= V_SYNC+V_BACK) && (cnt_v < V_SYNC+V_BACK+V_DISP)))
                         ?   1'b1 : 1'b0;
assign pixel_xpos = data_req ? (cnt_h - (H_SYNC + H_BACK - 1'b1)) : 10'd0;
assign pixel_ypos = data_req ? (cnt_v - (V_SYNC + V_BACK - 1'b1)) : 10'd0;
```

这一部分内容通过计数器，vga_en 将图像的有效区域固定出来，pixel_xpos 和 pixel_ypos 将有效区域内的每个像素点从左上角起给予一个确定坐标，以方便后续填入相应的颜色信息。仅有在有效区域内，才有颜色输出。

第四步：显示模块

显示模块主要任务书将显示区域分块，并且将每块内按要求填入颜色

```
parameter     H_DISP    = 10'd640;
parameter     V_DISP    = 10'd480;
localparam    WHITE     = 16'b11111_111111_11111;
localparam    BLACK     = 16'b00000_000000_00000;
localparam    RED       = 16'b11111_000000_00000;
localparam    GREEN     = 16'b00000_111111_00000;
localparam    BLUE      = 16'b00000_000000_11111;

always @(posedge vga_clk or negedge sys_rst_n)
begin
    if (!sys_rst_n)
        pixel_data <= 16'd0;
    else begin
        if ((pixel_xpos >= 0) && (pixel_xpos <= (H_DISP/5)*1))
            pixel_data <= WHITE;
        else if ((pixel_xpos >= (H_DISP/5)*1) && (pixel_xpos < (H_DISP/5)*2))
            pixel_data <= BLACK;
        else if ((pixel_xpos >= (H_DISP/5)*2) && (pixel_xpos < (H_DISP/5)*3))
            pixel_data <= RED;
        else if ((pixel_xpos >= (H_DISP/5)*3) && (pixel_xpos < (H_DISP/5)*4))
            pixel_data <= GREEN;
        else
            pixel_data <= BLUE;
    end
end
```

首先定义了五种颜色白，黑，红，绿，蓝，然后跟据像素坐标，将显示区域分成五块竖状条，分别将五种颜色填入其中。

13.4 实验验证

第一步：分配管脚

管脚分配如表 13-1 所示。

表 13-1 VGA 实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
clk	C10_50MCLK	91	系统时钟 50M
rst	KEY3	10	复位，默认低
vga_data[11]	VGA_R3	105	图像红色分量
vga_data[10]	VGA_R2	103	
vga_data[9]	VGA_R1	101	
vga_data[8]	VGA_R0	100	
vga_data[7]	VGA_G3	113	图像绿色分量
vga_data[6]	VGA_G2	112	
vga_data[5]	VGA_G1	111	
vga_data[4]	VGA_G0	106	
vga_data[3]	VGA_B3	120	图像蓝色分量
vga_data[2]	VGA_B3	119	
vga_data[1]	VGA_B2	115	
vga_data[0]	VGA_B1	114	
vga_hs	VGA_HS	98	行同步信号
vga_vs	VGA_VS	99	场同步信号

第二步：下板验证

管脚分配完成后进行全编译，通过后进行下板验证。实验现象如图 13-5 所示，屏幕被分成五个竖条，分别是白色，黑色，红色，绿色和蓝色。



图 13-5 VGA 实验现象

实验十四 以太网实验

14.1 实验目的

（1） 了解什么是以太网，以及其工作原理
（2） 熟悉不同接口类型（MII，GMII，RGMII）之间的关系以及优缺点（我们的开发板采用 RGMII）
（3） 结合开发板，完成数据的传输与接收，并验证

14.2 实验要求

（1） 进行回环测试，检查硬件是否正常工作
（2） 进行数据的就收验证
（3） 进行数据的发送验证

14.3 实验内容

14.3.1 实验原理简介

以太网是一种基带局域网技术，以太网通信是一种使用同轴电缆作为网络媒体，采用载波多路访问和冲突检测机制的通信方式，数据传输速率达到 1Gbit/s，可满足非持续性网络数据传输的需要。以太网接口作为一种互联型接口，当下应用非常的广泛。千兆以太网 MII 接口类型很多，常用 GMII 和 RGMII。

MII 接口一共有 16 根线，如图 14-1 所示。

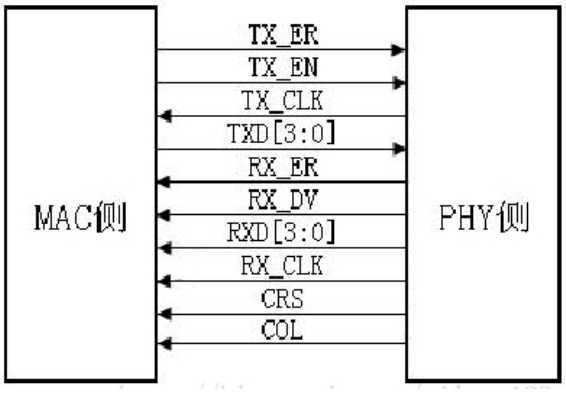


图 14-1 MII 接口

RXD(Receive Data)[3:0]：数据接收信号，共 4 根信号线；

TX_ER(Transmit Error)：发送数据错误提示信号，同步于 TX_CLK，高电平有效，表示 TX_ER 有效期内传输的数据无效。对于 10Mbps 速率下，TX_ER 不起作用；

RX_ER(Receive Error)： 接收数据错误提示信号，同步于 RX_CLK，高电平有效，表示 RX_ER 有效期内传输的数据无效。对于 10Mbps 速率下，RX_ER 不起作用；

TX_EN(Transmit Enable)： 发送使能信号，只有在 TX_EN 有效期内传的数据才有效；

RX_DV(Reveive Data Valid)： 接收数据有效信号，作用类型于发送通道的 TX_EN；

TX_CLK：发送参考时钟，100Mbps 速率下，时钟频率为 25MHz，10Mbps 速率下，时钟频率为 2.5MHz，注意，TX_CLK 时钟的方向是从 PHY 侧指向 MAC 侧的，因此此时钟是由 PHY 提供的；

RX_CLK：接收数据参考时钟，100Mbps 速率下，时钟频率为 25MHz，10Mbps 速率下，时钟频率为 2.5MHz。RX_CLK 也是由 PHY 侧提供的；

CRS：Carrier Sense，载波侦测信号，不需要同步于参考时钟，只要有数据传输，CRS 就有效，另外，CRS 只有 PHY 在半双工模式下有效；

COL：Collision Detectd，冲突检测信号，不需要同步于参考时钟，只有 PHY 在半双工模式下有效。

GMII 接口如图 14-2 所示。

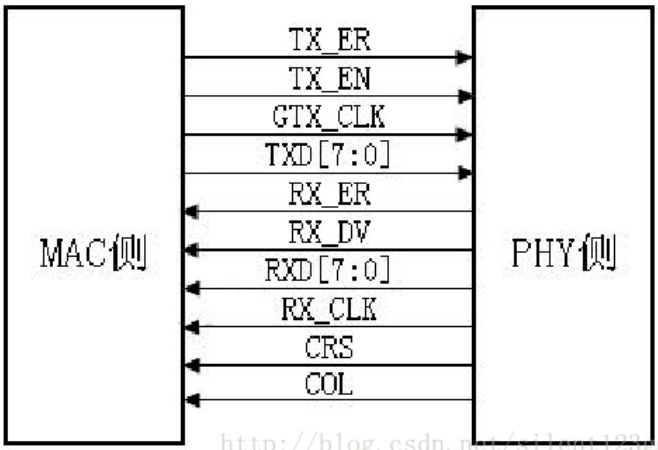


图 14-2 GMII 接口

与 MII 接口相比，GMII 的数据宽度由 4 位变为 8 位，GMII 接口中的控制信号如 TX_ER、TX_EN、RX_ER、RX_DV、CRS 和 COL 的作用同 MII 接口中的一样，发送参考时钟 GTX_CLK 和接收参考时钟 RX_CLK 的频率均为 125MHz(1000Mbps/8=125MHz)。

在这里有一点需要特别说明下，那就是发送参考时钟 GTX_CLK，它和 MII 接口中的 TX_CLK 是不同的，MII 接口中的 TX_CLK 是由 PHY 芯片提供给 MAC 芯片的，而 GMII 接口中的 GTX_CLK 是由 MAC 芯片提供给 PHY 芯片的。两者方向不一样。

在实际应用中，绝大多数 GMII 接口都是兼容 MII 接口的，所以，一般的 GMII 接口都有两个发送参考时钟：TX_CLK 和 GTX_CLK(两者的方向是不一样的，前面已经说过了)，在用作 MII 模式时，使用 TX_CLK 和 8 根数据线中的 4 根。

RGMII 接口如图 14-3 所示。

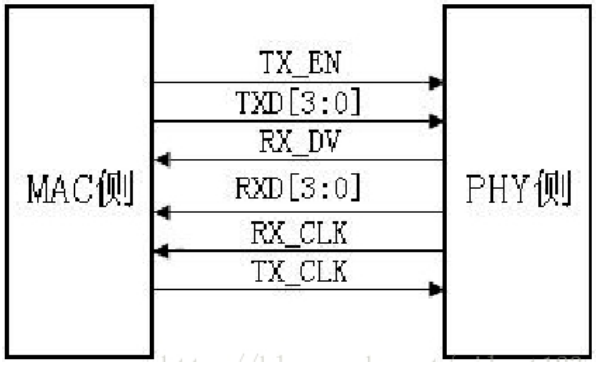


图 14-3 RGMII 接口

RGMII 即 Reduced GMII，是 GMII 的简化版本，将接口信号线数量从 24 根减少到 14 根(COL/CRS 端口状态指示信号，这里没有画出)，时钟频率仍旧为 125MHz，TX/RX 数据宽度从

8 为变为 4 位，为了保持 1000Mbps 的传输速率不变，RGMII 接口在时钟的上升沿和下降沿都采样数据。在参考时钟的上升沿发送 GMII 接口中的 TXD[3:0]/RXD[3:0]，在参考时钟的下降沿发送 GMII 接口中的 TXD[7:4]/RXD[7:4]。RGMI 同时也兼容 100Mbps 和 10Mbps 两种速率，此时参考时钟速率分别为 25MHz 和 2.5MHz。

TX_EN 信号线上传送 TX_EN 和 TX_ER 两种信息，在 TX_CLK 的上升沿发送 TX_EN，下降沿发送 TX_ER；同样的，RX_DV 信号线上也传送 RX_DV 和 RX_ER 两种信息，在 RX_CLK 的上升沿发送 RX_DV，下降沿发送 RX_ER。

14.3.2 硬件设计

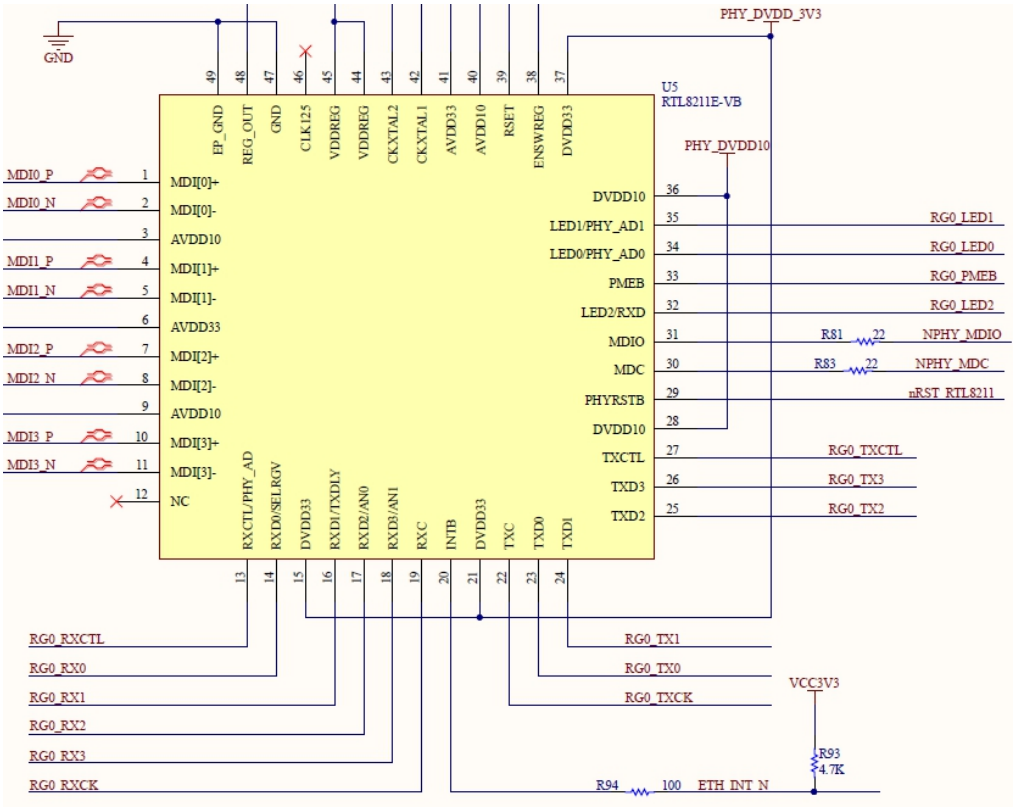


图 14-4 原理图 RTL8211E-VB 部分

实验板上采用 RTL8211E-VB 芯片组成千兆以太网模块，原理图如图 14-4 所示。PHY 芯片通过接收和发送两组信号与 FPGA 相连接。接收组信号前缀为 RG0_RX，发送组信号前缀为 RG0TX，分别有控制信号 CTL，时钟信号 CK 和四个年数据信号 3-0 组成。RG0_LED0 和 RG0_LED1 分别连接网口黄色信号灯和绿色信号灯。同时，FPGA 可以通过时钟线 NPHY_MDC 和数据线 NPHY_MDIO 对 PHY 芯片进行配置。

14.3.3 程序设计

（1） 回环测试设计（test1）

第一步：程序介绍

回环测试很简单，只需要将输入的数据直接输出即可。

```
module test1 (
    input                   rst,
```

```
    input                   rxc,
    input                   rxdv,
    input       [3:0]       rxd,
    output                  txc,
    output                  txen,
    output      [3:0]       txd,
);

    assign    txd  = rxd;
    assign    txen = rxdv;
    assign    txc  = rxc;

endmodule
```

（注意：本次实验中每个程序都含有一个 smi_ctrl 模块，在文件夹 config 中，是对 PHY 芯片的设置模块，以便于解决有些电脑不能正常连接网口的问题，暂时不做详细解释）

第二步：分配管脚

管脚分配如表 14-1 所示。

表 14-1　以太网实验管脚映射表

程序信号名	网络标号	FPGA 管脚	端口说明
rxc	RG0_RXCK	24	输入数据时钟
rxdv	RG0_RXCTL	34	输入数据控制信号
rxd[3]	RG0_RX3	28	输入数据第 3 位
rxd[2]	RG0_RX2	31	输入数据第 2 位
rxd[1]	RG0_RX1	32	输入数据第 1 位
rxd[0]	RG0_RX0	33	输入数据第 0 位
txc	RG0_TXCK	43	输出数据时钟
txen	RG0_TXCTL	42	输出数据控制信号
txd[3]	RG0_TX3	46	输出数据第 3 位
txd[2]	RG0_TX2	49	输出数据第 2 位
txd[1]	RG0_TX1	50	输出数据第 1 位
txd[0]	RG0_TX0	51	输出数据第 0 位
e_mdc	NPHY_MDC	38	配置时钟
e_mdio	NPHY_MDIO	39	配置数据

验证之前（默认自己的 PC 网卡是千兆网卡，否则请自行更换），需要首先确认自己的 PC 端 MAC 地址，如果不知道，就在 DOS 命令窗口，用 ipconfig -all 命令查看一下，以本机为例，如图 14-5 所示。

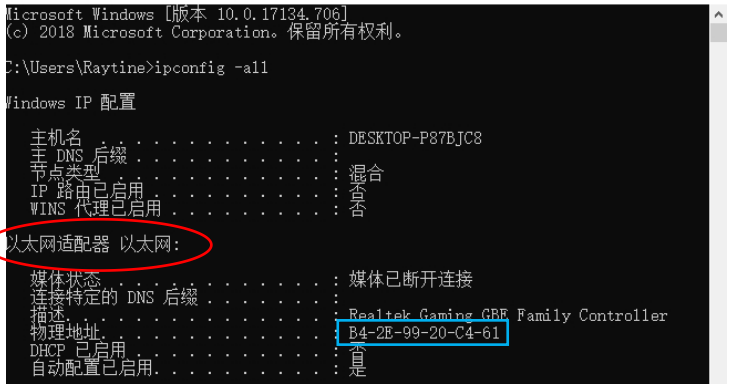


图 14-5　PC 端 MAC 地址信息

为方便后续实验，我们需要给 PC 端一个固定的 IP 地址，以本次实验为例，配置为 192.168.0.3（可以修改，但必须保证与实验的发送模块目标 IP 地址一致），如图 14-6 所示：



图 14-6　配置 PC 端 IP 地址

由于本次试验没有 ARP 协议内容（绑定开发板 IP 地址和 MAC 地址），因此需要通过 DOS 命令窗口人为绑定，本次试验将 IP 设为 192.168.0.2，MAC 地址设为 00-0A-35-01-FE-C0，（可以喜好自行更换）如图 14-7，方法如下：（注意：以管理员身份运行 DOS 命令窗口）

运行命令：ARP -s 192.168.0.2 00-0A-35-01-FE-C0

查看绑定结果：arp -a

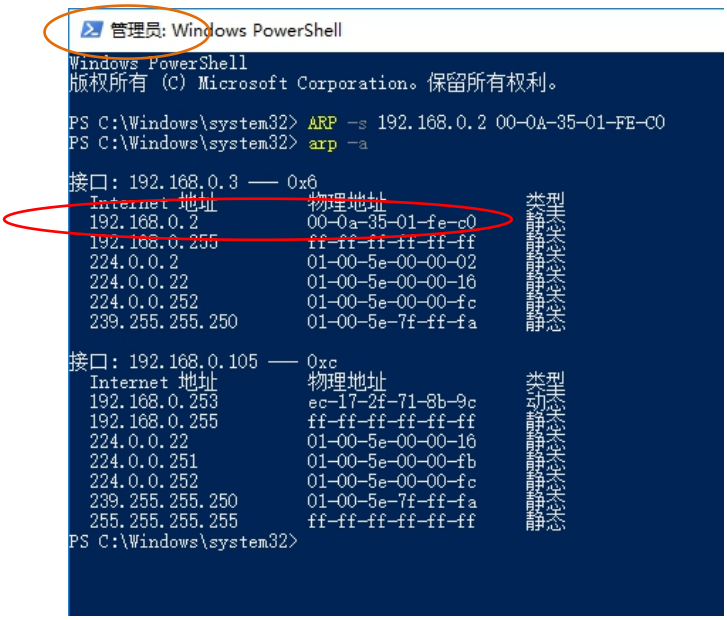


图 14-7　地址绑定方法 1

如果运行 ARP 命令时出现了失败，可以换一种方法绑定，如图 14-8：

1) 输入 netsh i i show in 命令查看本地连接的编号，如本次使用的电脑的“5”
2) 输入 netsh -c “i i” add neighbors 5（编号） “192.168.0.2” “00-0A-35-01-FE-C0”
3) 输入 arp -a 查看绑定结果

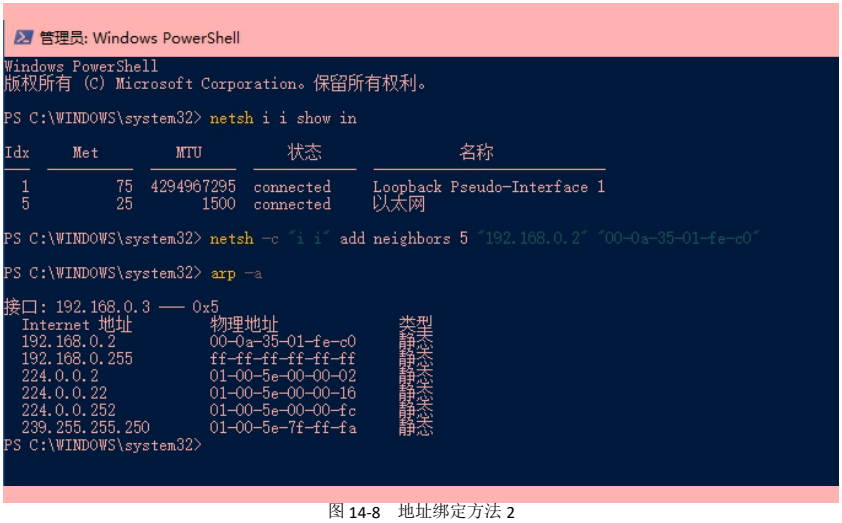


图 14-8 地址绑定方法 2

准备就绪后，我们同样用 DOS 命令窗口进行连通检测，如图 14-9 所示。Ping 是 Windows 系列自带的一个可执行命令。利用它可以检查网络是否能够连通，用好它可以很好地帮助我们分析判定网络故障。应用格式：Ping IP 地址。（除本机意外的任意地址）

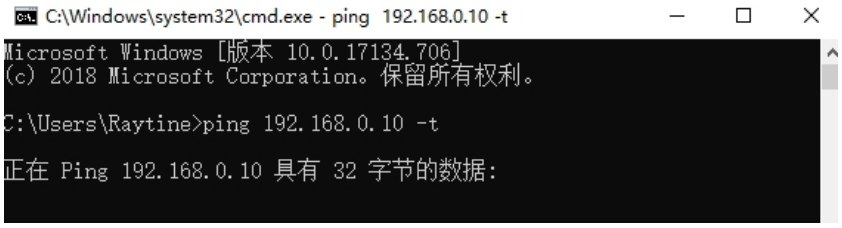


图 14-9 发送数据

打开 SignalTap，发送完命令后，如图 14-10 所示，我们可以看到数据正常，硬件完好。

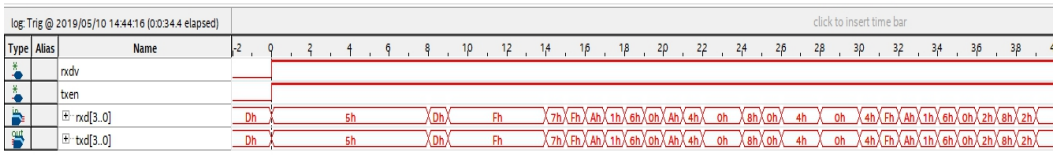


图 14-10 SignalTap II 数据捕捉

（2） 特殊 IP 核配置（test2）

由于是 RGMII 接口，数据为双边沿 4bit 数据，因此在 FPGA 内部进行数据处理时，需要转成 8bit 数据，我们调用 IP core（ddio_in）来实现，同时内部数据处理之后，再经由 IP core（ddio_out）将 8bit 数据转成双滚边沿 4bit 数据传出，需要注意的是，考虑到使能信号与数据信号同步的问题，因此将使能信号一同输入 ddio 进行转换，具体设置如图如图 14-11，14-12 所示。

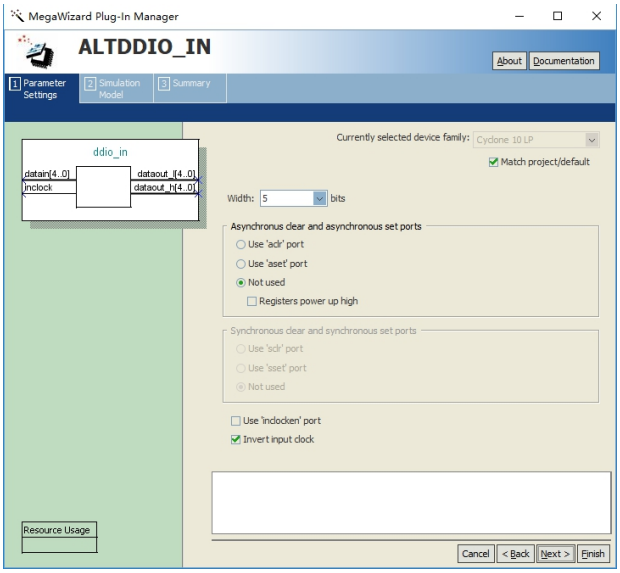


图 14-11　ddio_in 设置

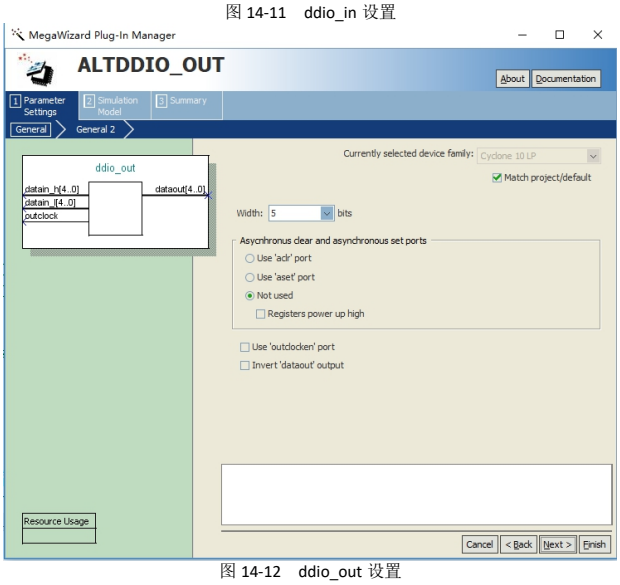


图 14-12　ddio_out 设置

考虑到 PHY 芯片提供的时钟的驱动能力比较差，所以经过锁相环处理，与以往不同的是，输入时钟 rxc 选择同源输入，如图 14-13 所示，输出 C0 时钟 ddio_clk 作为两个 ddio IP 核的驱动时钟，如图 14-14 所示，输出 C1 时钟 txc 作为数据的发送时钟（需要注意的是由于硬件电路原因以及时序原因，这里需要将 txc 做 90° 的相位差）如图 14-15 所示。

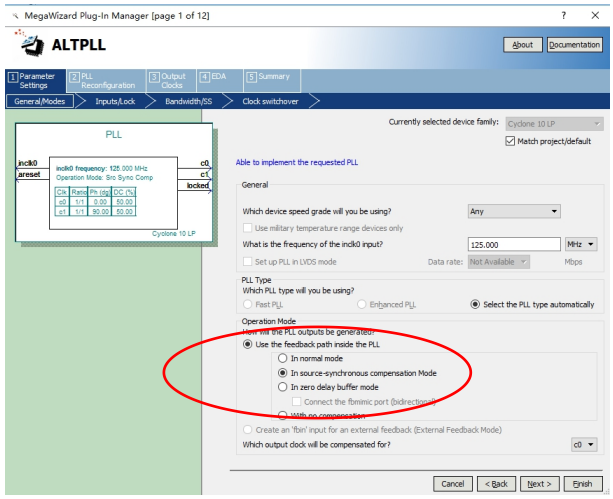


图 14-13　PLL 输入时钟设置

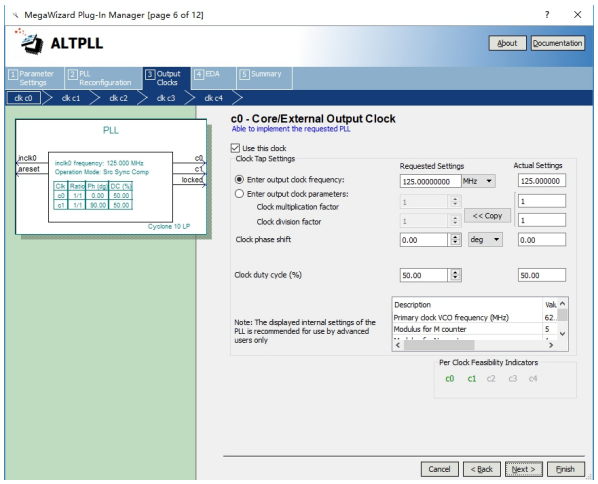


图 14-14　PLL 输出时钟（c0）设置

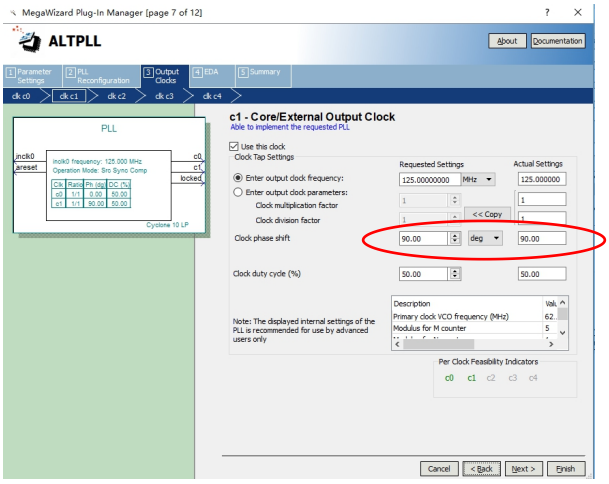


图 14-15　PLL 输出时钟（c1）设置

将三个 IP 核实例化到之前的回环测试中，进行数据传输正确性测试，（需要注意的是有序时序原因，需要将 ddio_out 输入数据相颠倒，具体参考工程文件（test2））。这次我们采用一个网络调试助手小程序作为辅助测试工具。下板，验证。

如图 14-16 所示，在设置好正确的地址和数据类型后，我们通过上位机发送检测信息（love you!）。通过 Wireshark 进行数据包的抓取，如图 14-17 所示，可以看到，数据正确的传输回 PC 端。

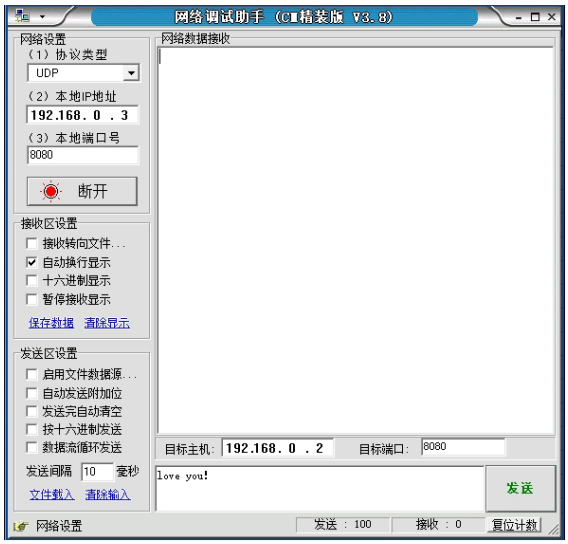


图 4-16　上位机发送测试数据

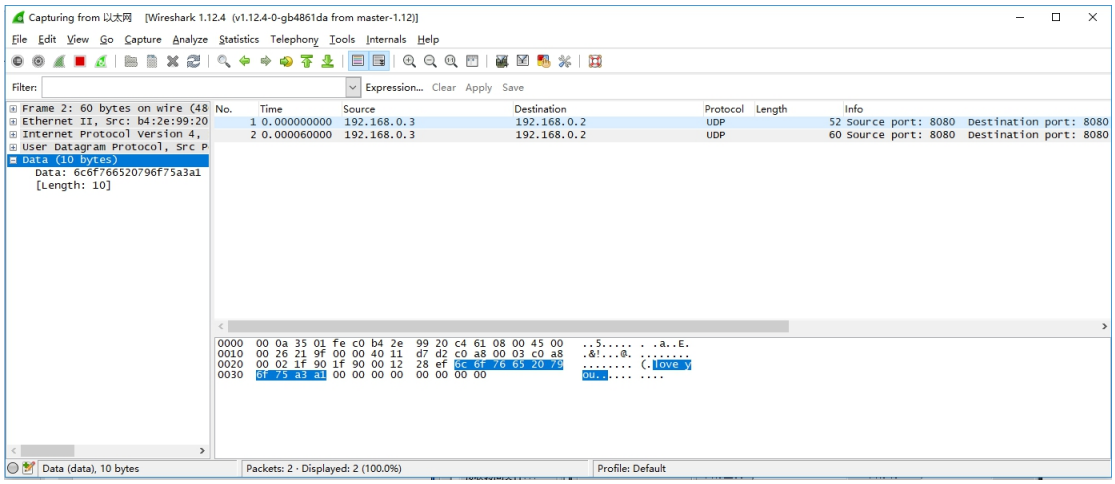


图 14-17　PC 端数据正确接收

（3）　完整以太网数据传输设计

完整的以太网数据传输，需要有数据的接收部分，以及数据的发送部分，为方便实验，我们将 PC 端发送的数据首先存到 ram 里，再经由发送端读取后发送给 PC 端，中间的一系列数据解包和打包参考工程文件“ethernet”。下面对每一个模块进行简单的介绍。

1)　数据接收模块（ip_receive）

这个模块所要解决的问题就是对数据帧进行检测识别，将有效的数据帧进行解包，并将真正的数据存入 ram 中。

```
always @ (posedge clk) begin
    if (clr) begin
        rx_state <= idle;
        data_receive <= 1'b0;
    end
    else
    case (rx_state)

        idle  :
        begin
            valid_ip_P <= 1'b0;
            byte_counter <= 3'd0;
            data_counter <= 10'd0;
            mydata <= 32'd0;
            state_counter <= 5'd0;
            data_o_valid <= 1'b0;
            ram_wr_addr <= 0;
            if (e_rxdv == 1'b1) begin
                if (datain[7:0] == 8'h55) begin                    //接收到第一个 55//
                    rx_state <= six_55;
                    mydata <= {mydata[23:0], datain[7:0]};
                end
                else
                    rx_state <= idle;
```

```
                end
            end

            six_55    :
            begin                                                    //接收 6 个 0x55//
                if ((datain[7:0] == 8'h55) && (e_rxdv == 1'b1)) begin
                    if (state_counter == 5) begin
                        state_counter <= 0;
                        rx_state <= spd_d5;
                    end
                    else
                        state_counter <= state_counter + 1'b1;
                end
                else
                    rx_state <= idle;
            end

            spd_d5    :
            begin                                                    //接收 1 个 0xd5//
                if ((datain[7:0] == 8'hd5) && (e_rxdv == 1'b1))
                    rx_state <= rx_mac;
                else
                    rx_state <= idle;
            end

            rx_mac    :
            begin                                  //接收目标 mac address 和源 mac address
                if (e_rxdv == 1'b1) begin
                    if (state_counter < 5'd11) begin
                        mymac <= {mymac[87:0], datain};
                        state_counter <= state_counter + 1'b1;
                    end
                    else begin
                        board_mac <= mymac[87:40];
                        pc_mac <= {mymac[39:0], datain};
                        state_counter <= 5'd0;
                        if((mymac[87:72] == 16'h000a) && (mymac[71:56] == 16'h3501) &&
(mymac[55:40] == 16'hfec0))     //判断目标 MAC Address 是否为本 FPGA
                            rx_state <= rx_IP_Protocol;
                        else
                            rx_state <= idle;
                    end
                end
                else
```

```
                rx_state <= idle;
        end

        rx_IP_Protocol  :
        begin                                           //接收 2 个字节的 IP TYPE
            if (e_rxdv == 1'b1) begin
                if (state_counter < 5'd1) begin
                    myIP_Prtcl <= {myIP_Prtcl[7:0], datain[7:0]};
                    state_counter <= state_counter+1'b1;
                end
                else begin
                    IP_Prtcl <= {myIP_Prtcl[7:0],datain[7:0]};
                    valid_ip_P <= 1'b1;
                    state_counter <= 5'd0;
                    rx_state <= rx_IP_layer;
                end
            end
            else
                rx_state <= idle;
        end

        rx_IP_layer     :
        begin                                       //接收 20 字节的 udp 虚拟包头,ip address
            valid_ip_P <= 1'b0;
            if (e_rxdv == 1'b1) begin
                if (state_counter < 5'd19) begin
                    myIP_layer <= {myIP_layer[151:0], datain[7:0]};
                    state_counter <= state_counter + 1'b1;
                end
                else begin
                    IP_layer <= {myIP_layer[151:0], datain[7:0]};
                    state_counter <= 5'd0;
                    rx_state <= rx_UDP_layer;
                end
            end
            else
                rx_state <= idle;
        end

        rx_UDP_layer  :
        begin                                   //接受 8 字节 UDP 的端口号及 UDP 数据包长
            rx_total_length <= IP_layer[143:128];
            pc_IP <= IP_layer[63:32];
            board_IP <= IP_layer[31:0];
```

```
            if (e_rxdv == 1'b1) begin
                if (state_counter < 5'd7) begin
                    myUDP_layer <= {myUDP_layer[55:0], datain[7:0]};
                    state_counter <= state_counter + 1'b1;
                end
                else begin
                    UDP_layer <= {myUDP_layer[55:0], datain[7:0]};
                    rx_data_length <= myUDP_layer[23:8];        //UDP 数据包的长度
                    state_counter <= 5'd0;
                    rx_state <= rx_data;
                end
            end
            else
                rx_state <= idle;
        end

        rx_data   :
        begin                                                   //接收 UDP 的数据
            if (e_rxdv == 1'b1) begin
                if (data_counter == rx_data_length-9) begin             //存最后的数据
                    data_counter <= 0;
                    rx_state <= rx_finish;
                    ram_wr_addr <= ram_wr_addr + 1'b1;
                    data_o_valid <= 1'b1;                                   //写 RAM

                    if (byte_counter == 3'd3) begin
                        data_o <= {mydata[23:0], datain[7:0]};
                        byte_counter <= 0;
                    end
                    else if (byte_counter==3'd2) begin
                        data_o <= {mydata[15:0], datain[7:0],8'h00};       //不满 32bit,补 0
                        byte_counter <= 0;
                    end
                    else if (byte_counter==3'd1) begin
                        data_o <= {mydata[7:0], datain[7:0], 16'h0000};   //不满 32bit,补 0
                                byte_counter <= 0;
                    end
                    else if (byte_counter==3'd0) begin
                        data_o <= {datain[7:0], 24'h000000};          //不满 32bit,补 0
                        byte_counter <= 0;
                    end
                end
                else begin
                    data_counter <= data_counter + 1'b1;
```

```
                        if (byte_counter < 3'd3) begin
                            mydata <= {mydata[23:0], datain[7:0]};
                            byte_counter <= byte_counter + 1'b1;
                            data_o_valid <= 1'b0;
                        end
                        else begin
                            data_o <= {mydata[23:0], datain[7:0]};
                            byte_counter <= 3'd0;
                            data_o_valid <= 1'b1;          //接受 4bytes 数据,写 ram 请求
                            ram_wr_addr <= ram_wr_addr+1'b1;
                        end
                    end
                end
                else
                    rx_state <= idle;
            end

            rx_finish  :
            begin
                data_o_valid <= 1'b0;                          //added for receive test//
                data_receive <= 1'b1;
                rx_state <= idle;
            end

            default    :    rx_state <= idle;
        endcase
end
```

接收模块，就是将接收到的数据进行一步步解析。

idle 状态：如果接收到 55，则跳转到 six_55 状态。

six_55 状态：如果继续收到连续六个 55，则跳转到 spd_d5 状态，否则返回 idle 状态。

spd_d5 状态：如果接继续接收到 d5，则证明收到了完整的数据包前导码 55_55_55_55_55_55_55_d5，并跳转到 rx_mac 转态，否则返回 idle 转态。

rx_mac 状态：这一部分是对目标 MAC 地址和源 MAC 地址的判断，如果匹配则跳转到 rx_IP_Protocol 状态，否则返回 idle 状态重新发送。

rx_IP_Protocol 状态：判断数据包的类型和长度，跳转到 rx_IP_layer 状态。

rx_IP_layer 状态：接收 20 字节的 udp 虚拟包头以及 IP 地址，跳转到 rx_UDP_layer 状态

rx_UDP_layer 状态：接收 8 字节 UDP 的端口号及 UDP 数据包长，跳转到 rx_data 状态

rx_data 状态：接收 UDP 的数据，跳转到 rx_finish 状态，

rx_finish 状态：一包数据接收完成，跳转到 idle 状态等待下一包数据的到来。

2) 数据发送模块（ip_send）

这个模块的主要内容是将 ram 中的数据读出，并以正确的数据包协议类型（UDP）将数据打包并发送出去。在发送之前，还要进行数据的校验，用 CRC 校验。

```
initial begin
    tx_state <= idle;
    //定义 IP 包头
    preamble[0] <= 8'h55;                    //7 个前导码 55,一个帧开始符 d5
    preamble[1] <= 8'h55;
    preamble[2] <= 8'h55;
    preamble[3] <= 8'h55;
    preamble[4] <= 8'h55;
    preamble[5] <= 8'h55;
    preamble[6] <= 8'h55;
    preamble[7] <= 8'hD5;

    mac_addr[0] <= 8'hB4;                    //目的 MAC 地址 ff-ff-ff-ff-ff-ff, 全 ff 为广播
包
    mac_addr[1] <= 8'h2E;                    //目的 MAC 地址 B4-2E-99-20-C4-61, 为本次
实验使用的 PC 端地址，调试阶段根据自己 PC 端的值改变内容
    mac_addr[2] <= 8'h99;
    mac_addr[3] <= 8'h20;
    mac_addr[4] <= 8'hC4;
    mac_addr[5] <= 8'h61;

    mac_addr[6] <= 8'h00;                    //源 MAC 地址 00-0A-35-01-FE-C0
    mac_addr[7] <= 8'h0A;                    //根据自己的需求修改为指定的值
    mac_addr[8] <= 8'h35;
    mac_addr[9] <= 8'h01;
    mac_addr[10]<= 8'hFE;
    mac_addr[11]<= 8'hC0;

    mac_addr[12]<= 8'h08;                    //0800: IP 包类型
    mac_addr[13]<= 8'h00;

    i<=0;
end
```

这一部分定义了数据包的前导码，PC 端 MAC 地址，开发板 MAC 地址以及 IP 包类型，需要注意的是在实际实验中，PC 端 MAC 地址需要根据自己的电脑进行修改，PC 端 MAC 地址需要根据自己绑定的地址进行修改，但前提是都不洗保证一致性，否则后续实验会接收不到数据。

```
always @ (posedge clk) begin
case (tx_state)
    idle  :
    begin
        e_txen <= 1'b0;
```

```
            crcen <= 1'b0;
            crcre <= 1;
            j <= 0;
            dataout <= 0;
            ram_rd_addr <= 1;
            tx_data_counter <= 0;
            if (time_counter == 32'h04000000) begin          //等待延迟，每隔一段时间发送一
个数据包
            tx_state <= start;
            time_counter <= 0;
        end
        else
            time_counter <= time_counter + 1'b1;
end

start :
begin               //IP header
        ip_header[0] <= {16'h4500, tx_total_length};          //版本号：4；  包头长度：20；IP
包总长
        ip_header[1][31:16] <= ip_header[1][31:16]+1'b1;      //包序列号
        ip_header[1][15:0] <= 16'h4000;                       //Fragment offset
        ip_header[2] <= 32'h80110000;                         //mema[2][15:0] 协议：17(UDP)
        ip_header[3] <= 32'hc0a80002;                         //192.168.0.2 源地址
        ip_header[4] <= 32'hc0a80003;                         //192.168.0.3 目的地址广播地址
        ip_header[5] <= 32'h1f901f90;                         //2 个字节的源端口号和 2 个字节
的目的端口号
        ip_header[6] <= {tx_data_length, 16'h0000};           //2 个字节的数据长度和 2 个字节
的校验和（无）
        tx_state <= make;
end

make      :
begin                           //生成包头的校验和
        if (i == 0) begin
            check_buffer <= ip_header[0][15:0] + ip_header[0][31:16] +
                            ip_header[1][15:0] + ip_header[1][31:16] +
                            ip_header[2][15:0] + ip_header[2][31:16] +
                            ip_header[3][15:0] + ip_header[3][31:16] +
                            ip_header[4][15:0] + ip_header[4][31:16];
            i <= i + 1'b1;
        end
        else if(i == 1) begin
            check_buffer[15:0] <= check_buffer[31:16] + check_buffer[15:0];
            i <= i+1'b1;
```

```
        end
        else begin
            ip_header[2][15:0] <= ~check_buffer[15:0];        //header checksum
            i <= 0;
            tx_state <= send55;
        end
end

send55    :
begin                                          //发送 8 个 IP 前导码:7 个 55, 1 个 d5
        e_txen <= 1'b1;                                      //GMII 数据发送有效
        crcre <= 1'b1;                                       //reset crc
        if(i == 7) begin
            dataout[7:0] <= preamble[i][7:0];
            i <= 0;
            tx_state <= sendmac;
        end
        else begin
            dataout[7:0] <= preamble[i][7:0];
            i <= i + 1'b1;
        end
end

sendmac :
begin                              //发送目标 MAC address 和源 MAC address 和 IP 包类型
        crcen <= 1'b1;             //crc 校验使能，crc32 数据校验从目标 MAC 开始
        crcre <= 1'b0;
        if (i == 13) begin
            dataout[7:0] <= mac_addr[i][7:0];
            i <= 0;
            tx_state <= sendheader;
        end
        else begin
            dataout[7:0] <= mac_addr[i][7:0];
            i <= i + 1'b1;
        end
end

sendheader    :
begin                                                  //发送 7 个 32bit 的 IP  包头
        datain_reg <= datain;                          //准备需要发送的数据
        if(j == 6) begin
            if(i == 0) begin
                dataout[7:0] <= ip_header[j][31:24];
```

```
                i <= i + 1'b1;
            end
            else if(i == 1) begin
                dataout[7:0] <= ip_header[j][23:16];
                i <= i + 1'b1;
            end
            else if(i == 2) begin
                dataout[7:0] <= ip_header[j][15:8];
                i <= i + 1'b1;
            end
            else if(i == 3) begin
                dataout[7:0] <= ip_header[j][7:0];
                i <= 0;
                j <= 0;
                tx_state <= senddata;
            end
        end
        else begin
            if(i == 0) begin
                dataout[7:0] <= ip_header[j][31:24];
                i <= i + 1'b1;
            end
            else if(i == 1) begin
                dataout[7:0] <= ip_header[j][23:16];
                i <= i + 1'b1;
            end
            else if(i == 2) begin
                dataout[7:0] <= ip_header[j][15:8];
                i <= i + 1'b1;
            end
            else if(i == 3) begin
                dataout[7:0] <= ip_header[j][7:0];
                i <= 0;
                j <= j + 1'b1;
            end
        end
end

senddata :
begin                                                   //发送 UDP 数据包
        if(tx_data_counter == tx_data_length - 9) begin     //发送最后的数据
            tx_state <= sendcrc;
                if (i == 0) begin
                    dataout[7:0] <= datain_reg[31:24];
```

```
                    i <= 0;
                end
                else if (i == 1) begin
                    dataout[7:0] <= datain_reg[23:16];
                    i <= 0;
                end
                else if (i == 2) begin
                    dataout[7:0] <= datain_reg[15:8];
                    i <= 0;
                end
                else if (i == 3) begin
                    dataout[7:0] <= datain_reg[7:0];
                    datain_reg <= datain;                    //准备数据
                    i <= 0;
                end
        end
        else begin                                           //发送其它的数据包
            tx_data_counter <= tx_data_counter+1'b1;
                if (i == 0) begin
                    dataout[7:0] <= datain_reg[31:24];
                    i <= i + 1'b1;
                    ram_rd_addr <= ram_rd_addr + 1'b1;       //RAM 地址加 1，提前让 RAM
输出数据
                end
                else if (i == 1) begin
                    dataout[7:0] <= datain_reg[23:16];
                    i <= i + 1'b1;
                end
                else if (i == 2) begin
                    dataout[7:0] <= datain_reg[15:8];
                    i <= i + 1'b1;
                end
                else if (i == 3) begin
                    dataout[7:0] <= datain_reg[7:0];
                    datain_reg <= datain;                    //准备数据
                    i <= 0;
                end
        end
end

sendcrc   :
begin                                                        //发送 32 位的 crc 校验
        crcen <= 1'b0;
        if (i == 0) begin
```

```
            dataout[7:0] <= {~crc[24], ~crc[25], ~crc[26], ~crc[27], ~crc[28], ~crc[29], ~crc[30],
~crc[31]};
            i <= i + 1'b1;
        end
        else begin
            if (i == 1) begin
                dataout[7:0] <= {~crc[16], ~crc[17], ~crc[18], ~crc[19], ~crc[20], ~crc[21],
~crc[22], ~crc[23]};
                    i <= i + 1'b1;
            end
            else if (i == 2) begin
                dataout[7:0] <= {~crc[8], ~crc[9], ~crc[10], ~crc[11], ~crc[12], ~crc[13], ~crc[14],
~crc[15]};
                    i <= i + 1'b1;
            end
            else if (i == 3) begin
                dataout[7:0] <= {~crc[0], ~crc[1], ~crc[2], ~crc[3], ~crc[4], ~crc[5], ~crc[6],
~crc[7]};
                i <= 0;
                tx_state <= idle;
            end
        end
end

default   :     tx_state <= idle;
endcase
end
```

idle 状态：等待延时，每隔一段时间发送一个数据包，跳转到 start 状态。
start 状态：发送数据包头，跳转到 make 状态。
make 状态：生成包头的校验和，跳转到 send55 状态。
send55 状态：发送 8 个前导码，跳转到 sendmac 状态。
sendmac 状态：发送目标 MAC 地址，源 MAC 地址和 IP 包类，跳转到 sendheader 状态。
sendheader 状态：发送 7 个 32bit 的 IP 包头，跳转到 senddata 状态。
senddata 状态：发送 UDP 数据包，跳转到 sendcrc 状态。
sendcrc 状态：发送 32 位的 crc 校验，返回 idle 状态。
这样，整个一包数据发送结束，返回 idle 状态等待下一包数据的发送。

3) CRC 校验模块（crc）

一个 IP 数据包的 CRC32 校验是在目标 MAC Address 开始计算的，一直计算到一个包的最后一个数据为止。以太网的 CRC32 的 verilog 的算法和多项式可以在以下网站中直接生成：http://www.easics.com/webtools/crctool

4) UDP 数据测试模块（UDP）

这个模块只需要将前三个子模块实例化到一起即可，注意每个连接的正确性。

5) 顶层模块设置（ethernet）

将 PLL，ddio_in，ddio_out，ram 以及 UDP 模块实例化到顶层，并在 ram 中提前存入特定的信息（Welcome To ZGZNXP World！），在没有数据输入的时候，FPGA 一直发送此信息，有数据输入后，发送接收到的数据。具体实现参考工程文件。

14.4 实验验证

管脚分配测试程序 test1 一致。

下板验证之前，需要注意的是必须将自己的 PC 端 IP 地址和开发板 MAC 地址确定好，并对应起来，否则将收不到数据。

将编译好的工程下载到开发板中，可以看到，如图 14-18 所示，FPGA 一直在向 PC 端发送信息。在 Wireshark 也可以看到整个发送的数据包，如图 14-19 所示。



图 14-18 发送特定信息

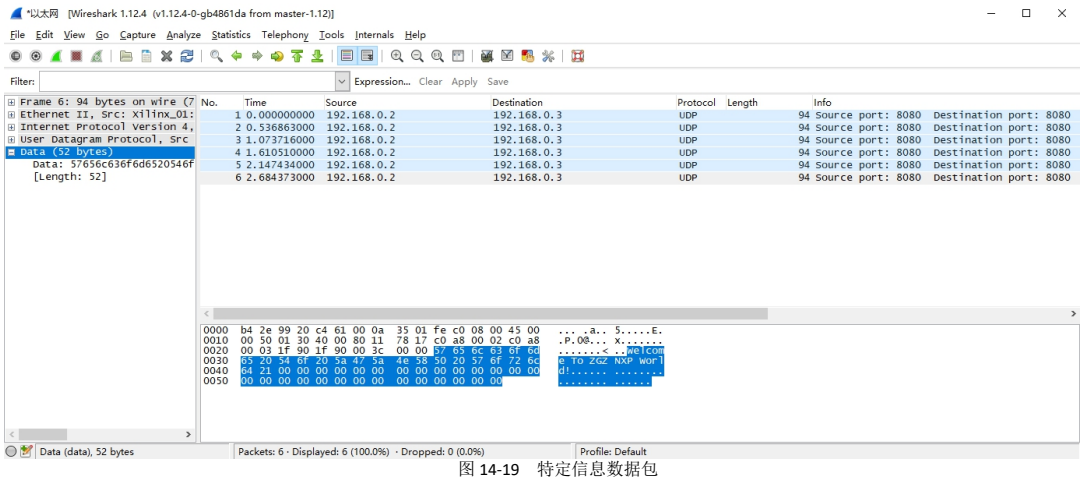


图 14-19 特定信息数据包

当 PC 端向 FPGA 发送数据时，如图 14-20 所示，可以看到在一整个数据包到达 FPGA，并且紧接着 FPGA 将接收到的数据重新打包后发送给 PC 端。如图 14-21 所示，同时网络助手也同时准确接收到发送数据信息，如图 14-22 所示。同样的通过 SignalTap 我们可以看到

接受到的数据的写入过程，如图 14-23 所示。

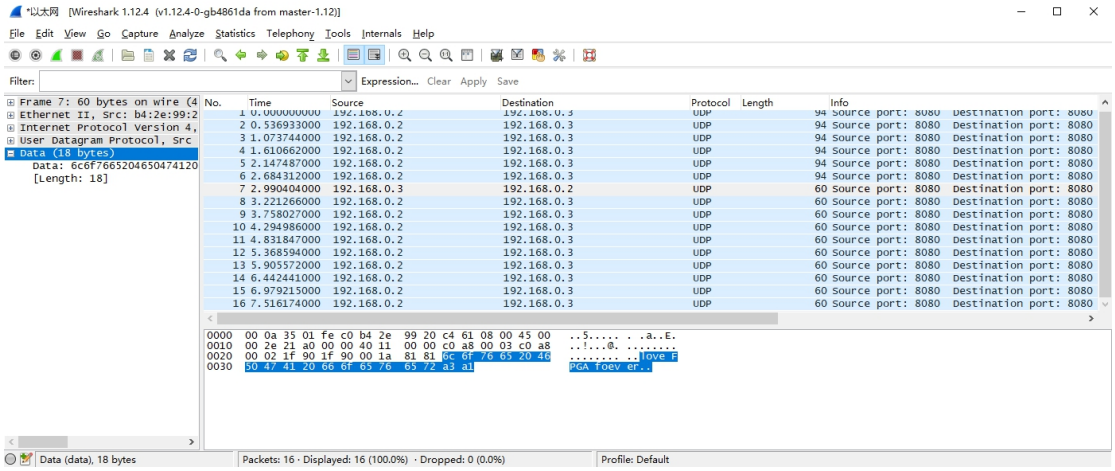


图 14-20　PC 发送数据包

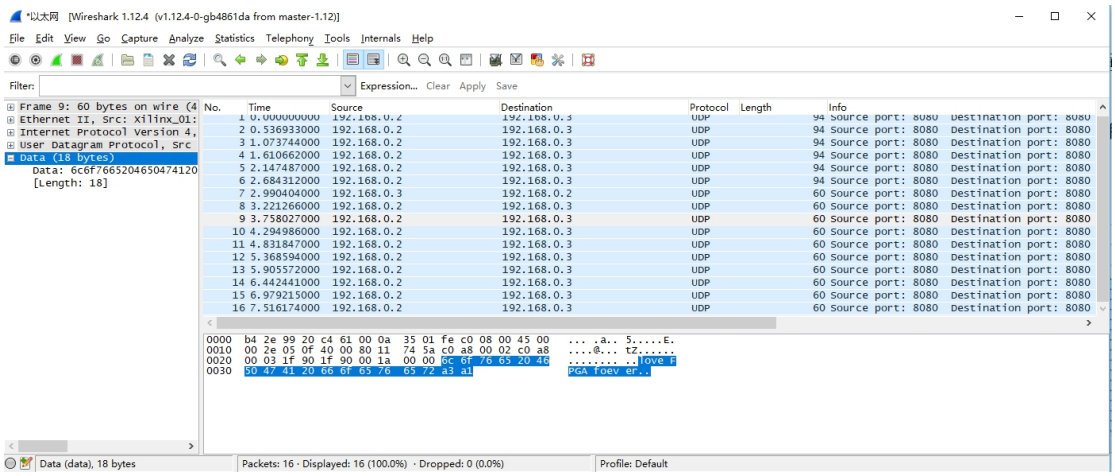


图 14-21　FPGA 将接收到的数据重新打包发送给 PC 端

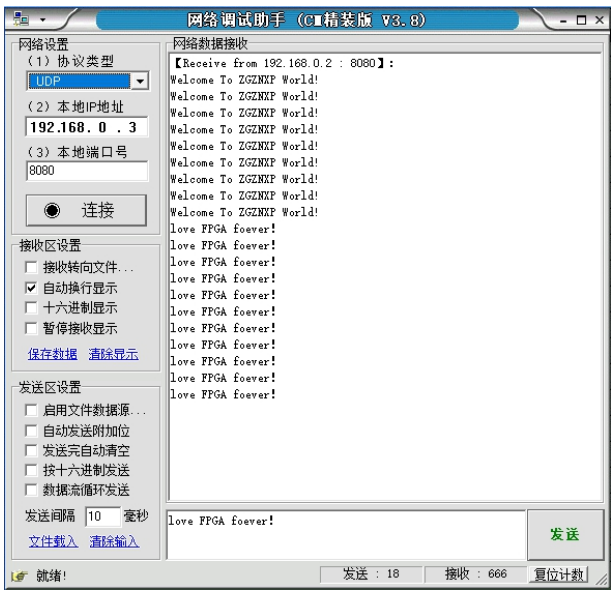


图 14-22　FPGA 发送给 PC 端接受到的信息



图 14-23　FPGA 结束数据并存入 ram 过程

需要注意的是， 以太网 Ethernet II 规定，以太网帧数据域部分最小为 46 字节，也就是以太网帧最小是 6＋6＋2＋46＋4＝64。除去 4 个字节的 FCS，因此，抓包时就是 60 字节。当数据字段的长度小于 46 字节时，MAC 子层就会在数据字段的后面填充以满足数据帧长不小于 64 字节。在用 UDP 局域网通信时，经常发生“Hello World”来进行测试，但是“Hello World”并不满足最小有效数据(64-46)的要求，为什么小于 18 个字节，对方仍然可用收到呢？因为在链路层的 MAC 子层中会进行数据补齐，不足 18 个字节的用 0 补齐。但当服务器在公网，客户端在内网，发生小于 18 个字节的数据，就会出现接收端收不到数据的情况。因此如果出现接收不到的情况，将发送的信息增加到 18 个字节以上即可。