

版本信息：

版本

REV2018

时间

04/12/2018

ZYNQ 修炼秘籍

基于米联客系列开发板

第一季 基于 ZYNQ 的 FPGA 基础入门

电子版自学资料

常州一二三电子科技有限公司
溧阳米联电子科技有限公司
版权所有

米联客学院 04QQ 群： 516869816

米联客学院 03QQ 群： 543731097

米联客学院 02QQ 群： 86730608

米联客学院 01QQ 群： 34215299

版本	时间	描述
Rev2016	2015-07-25	第一版初稿，大部分采用 zedboard 资料
Rev2017	2017-01-31	做了重大改进，自己编写里批处理命令，方便移植
Rev2018	2017-12-16	对 2017 版本改进，修改教程 bug 同时增加更多学习课程
Rev2018	2017-04-15	对 2018-12-16 版本改进，细化基础部分内容

感谢您使用米联客开发板团队开发的 ZYNQ 开发板，以及配套教程。本教程将对之前编写的《ZYNQ 修炼秘籍》-LINUX 部分内容做出改进，并且增加新的课程内容。本教程不仅仅适合用于米联客开发板，而且可以用于其他的 ZYNQ 开发。

软件版本：VIVADO2015.4 (linux 部分安装主要用到里面的交叉编译环境)

软件版本：VIVADO2016.4 (首期代码用 2016.4,读者可以自行升级到高版本)

软件版本：VIVADO2017.4 (2017.4 预计在 2018 年 1 月官方发布软件)

版权声明：

本手册版权归常州一二三电子科技有限公司/溧阳米联电子科技有限公司所有，并保留一切权利，未经我司书面授权，擅自摘录或者修改本手册部分或者全部内容，我司有权追究其法律责任。

技术支持：

版主大神们都等着大家去提问--电子资源论坛 www.osrc.cn

微信公众平台：电子资源论坛



目录

ZYNQ 修炼秘籍.....	1
目录.....	4
【第一季】ZYNQ SOC 开机及 FPGA 基础 共 10 课时.....	7
CH01_开机程序测试.....	8
1.1 开机测试的目的.....	8
1.2 开机前准备.....	8
1.3 开机测试.....	11
CH02_ZYNQ_VIVADO 软件安装.....	14
2.1 VIVADO 软件介绍.....	14
2.2 VIVADO 软件安装(适合所有 vivado 安装).....	14
2.3 VIVADO 软件注册.....	19
2.3 本章小结.....	21
CH03_USB 下载器驱动安装及下载程序.....	22
3.1 下载器驱动的安装.....	22
3.2 下载 runled 工程的 bit 文件验证板子和下载器工作正常.....	24
3.3 下载器使用需要注意的问题.....	25
CH04_FPGA 设计 Verilog 基础（一）.....	27
4.1 Verilog HDL 代码规范.....	27
◆ 项目构架设计	27
◆ 接口时序设计规范	27
4.2 技术背景.....	30
4.3 Verilog 最基础语法.....	33
4.4 关键字.....	34
4.5 Verilog 中数值表示的方式	40
4.6 阻塞赋值和非阻塞赋值详解	40
CH05_FPGA 设计 Verilog 基础（二）.....	45
5.1 状态机设计.....	45
5.2 一段式状态机.....	46
5.3 两段式状态机.....	47
5.4 三段式状态机.....	49
CH06_FPGA 设计 Verilog 基础（三）.....	52
6.1 完成的 Test bench 文件结构.....	52
6.2 时钟激励设计.....	52
6.3 复位信号设计.....	54
6.4 特殊信号设计.....	55
6.5 仿真控制语句及系统任务描述.....	58
6.6 加法器的仿真测试文件编写	61
CH07_FPGA_RunLED 创建 VIVADO 工程实验.....	64

7.1 硬件图片.....	64
7.2 硬件原理图.....	64
7.3 新建 VIVADO 工程	64
7.4 创建工程文件.....	67
7.5 Verilog FPGA 流水灯实验.....	70
7.6 添加管脚约束文件.....	72
7.7 编译并且产生 bit 文件.....	77
7.8 下载程序.....	77
7.9 实验结果.....	79
7.10 本章小结.....	80
CH08_FPGA_Button 按钮去抖动实验.....	81
8.1 硬件介绍.....	81
8.2 时序设计.....	81
8.3 程序源码.....	82
8.4 程序分析.....	86
8.5 综合布线前仿真时序.....	87
8.6 Chipscope 在线逻辑分析仪仿真.....	87
8.7 输出结果.....	87
8.8 小结.....	87
CH09_FPGA 多路分配器设计.....	88
9.1 硬件图片.....	88
9.2 硬件原理图.....	88
9.3 介于 VIVADO 的 FPGA 设计流程	89
9.4 多路分配器设计思想.....	89
9.5 时序设计.....	90
9.6 程序源码.....	90
9.7 行为仿真.....	95
9.7.1 创建多路分频器工程.....	95
9.7.2 添加仿真文件.....	98
9.7.3 行为级仿真.....	101
9.8 综合 Synthesis	105
9.8.1 添加文件.....	105
9.8.2 综合并查看报告.....	107
9.8.3 综合时序仿真.....	107
9.9 执行 Implementation	108
9.9.1 执行并查看报告.....	108
9.9.2 布局布线后时序仿真.....	109
9.10 VIVADO 在线逻辑分析仪使用.....	110
9.10.1 IP Catalog 添加 IA ip core	110
9.10.2 逻辑分析仪抓取的信号.....	114
9.10.3 逻辑分析仪使用.....	115
9.11 小结.....	116
CH10_HDMI 接口测试	117

10.1 创建工程文件.....	117
10.2 添加工程文件.....	119
10.3 添加管脚约束文件.....	134
10.4 编译并且产生 bit 文件.....	136
10.5 下载程序.....	136
10.6 实验结果.....	137
10.7 本章小结.....	138

【第一季】ZYNQ SOC 开机及 FPGA 基础 共 10 课时

第一季课程共计 10 课时，主要讲解开机测试，JTAG 下载程序，FPGA 基础语法基础，VIVADO 软件快速入门、VGA 或者 HDMI 接口的测试。

开发人员拿到板子后第一件事情应该是对板子做一个开机测试。对于有 FPGA 基础，第一次使用 ZYNQ，第一次使用 VIVADO 软件的读者，可以把软件使用相关课程看下；对于没有 FPGA 基础的，需要把 FPGA 基础的知识好好学习下。对于熟悉 ZYNQ 软件的，也会 FPGA 开发的，可以跳过本章基础部分，直接进入后面章节学习。

CH01_开机程序测试

1.1 开机测试的目的

使用者进入正式开发前，需要对开发板各个接口进行功能测试，验证开发板功能可靠。开机测试通过后，使用者可进行后续的开发工作。

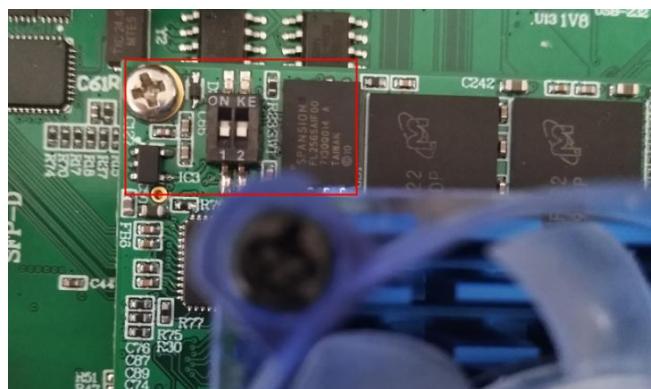
1.2 开机前准备

1、启动模式检查

检查开发板的拨码开关是否是 SD 卡启动模式，即 1-OFF, 2-OFF 状态。如果不是，请将拨码开关拨到 1-OFF, 2-OFF 状态。

表. 开发板启动模式

启动模式	开关状态
SD 卡启动/JTAG 调试模式	开关 1-OFF 开关 2-OFF
启动/JTAG 调试模式	开关 1-ON 开关 2-OFF



拨码开关

2、连接下载器

下载器连接：下载器一端通过 mini USB 线连接到电脑，另一端通过转接板连接到开发板 JTAG 接口。开发板未通电，红色的 LED 灯亮，绿色 LED 灯灭；开发板通电，红色 LED 灯亮，绿色 LED 灯亮。



开发板未通电

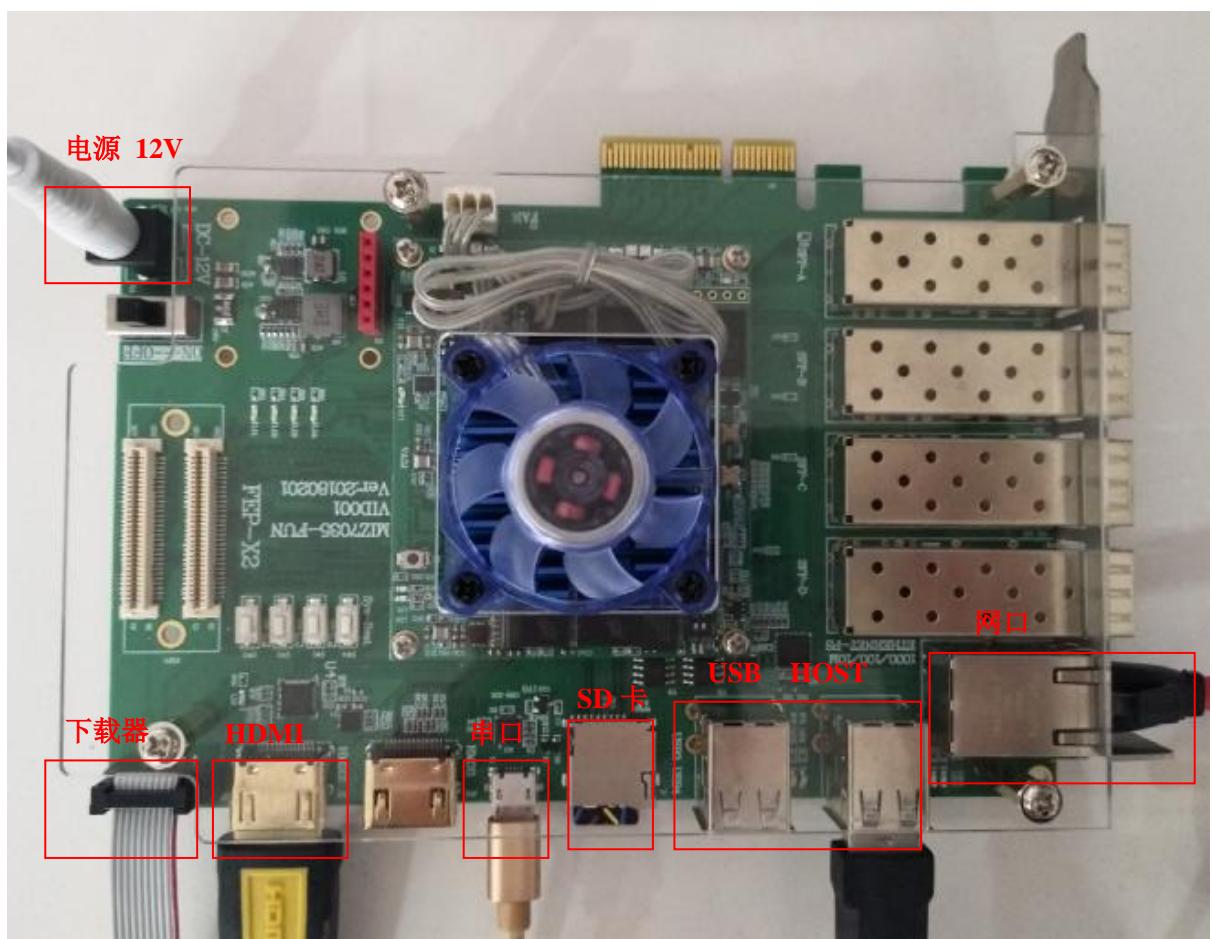


开发板通电

3、连接开发板测试端口

表. 测试端口的连接

接口	描述
电源接口	DC 12V
下载器接口	下载器
HDMI 接口	HDMI 线一端接入开发板，另一端接电脑显示 屏或 LCD 屏
串口	接入串口线
TF 卡座	插入 SD 卡（包含出厂开机测试程序）
USB HOST 接口	接鼠标
网口	接入网线

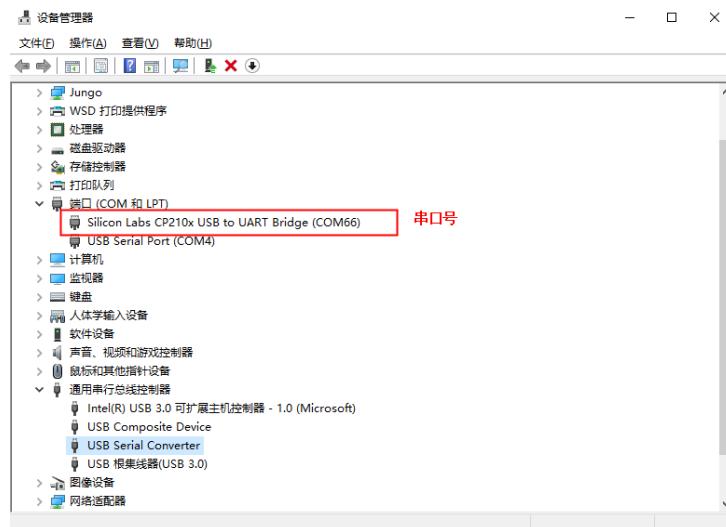


MIZ7035 开机测试连线图

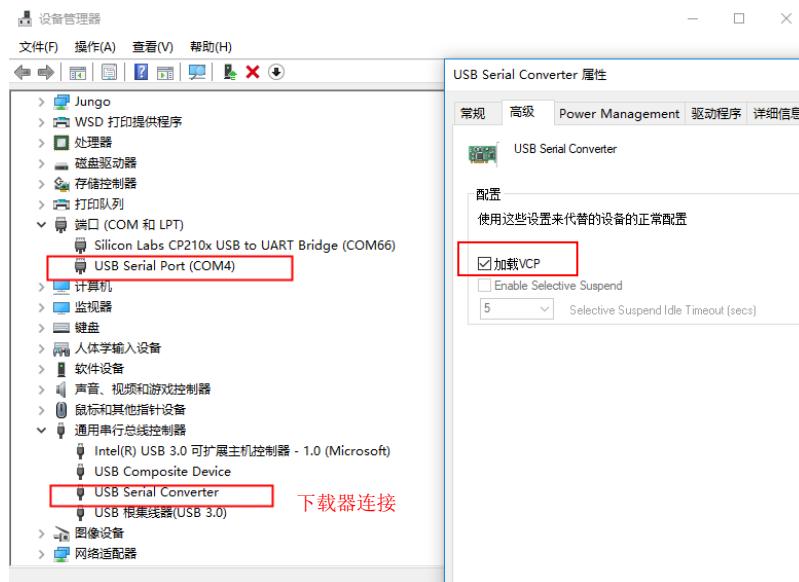
4、查看下载器和串口连接

打开电脑设备管理器，查看串口端口号和下载器连接是否正常。

注意：如果下载器驱动不识别，请参考教程中“CH03_USB 下载器驱动安装及下载程序”重新安装驱动。



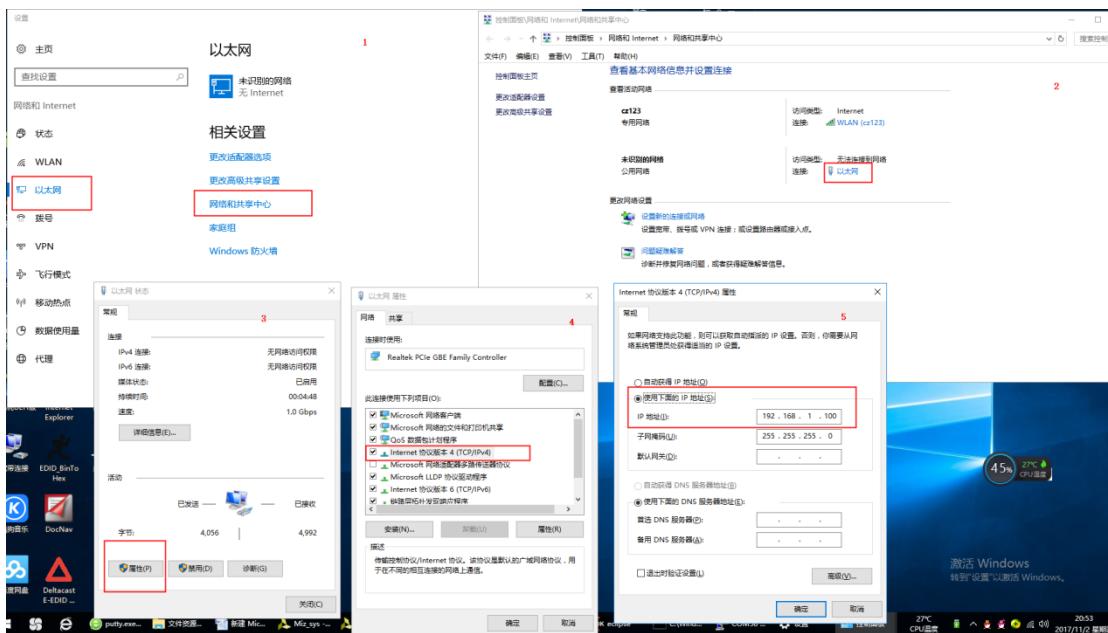
串口连接正常



下载器连接正常

5、设置本地主机 IP 地址。

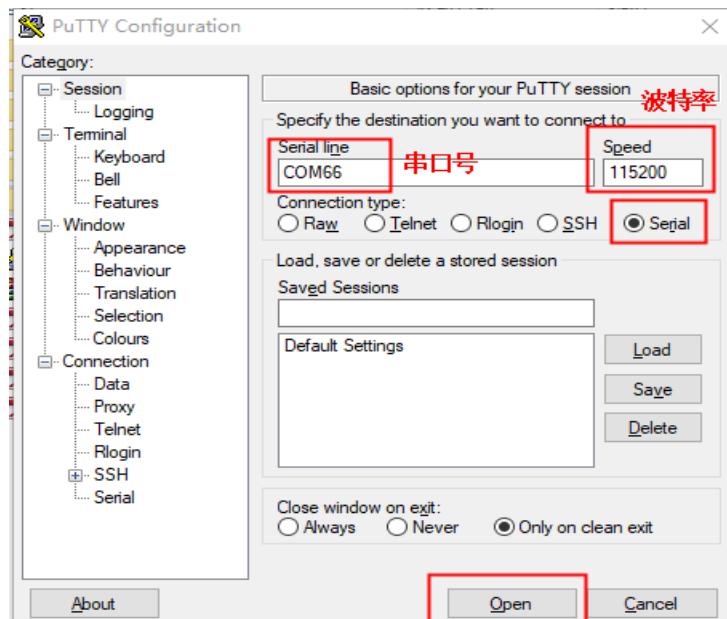
IP 地址可自定义。设置本地主机 IP 地址目的：便于区分主机和从机，开发板的 IP 地址不能与主机的 IP 地址相同。下图为 Windows 10 IP 地址修改。图中本地主机设置的 IP 地址为 192.168.1.100。



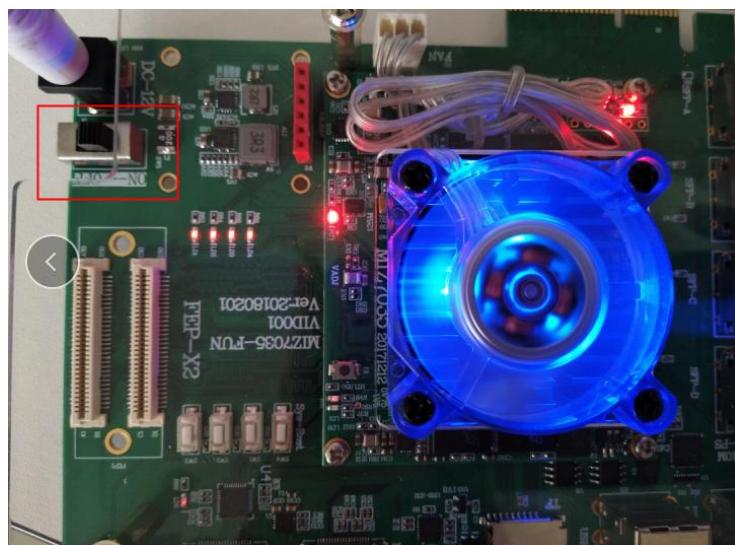
主机 IP 地址设置

1.3 开机测试

1、打开 Putty 软件。首先设置串口波特率、串口号。然后打开串口，电源开关拨到 ON。



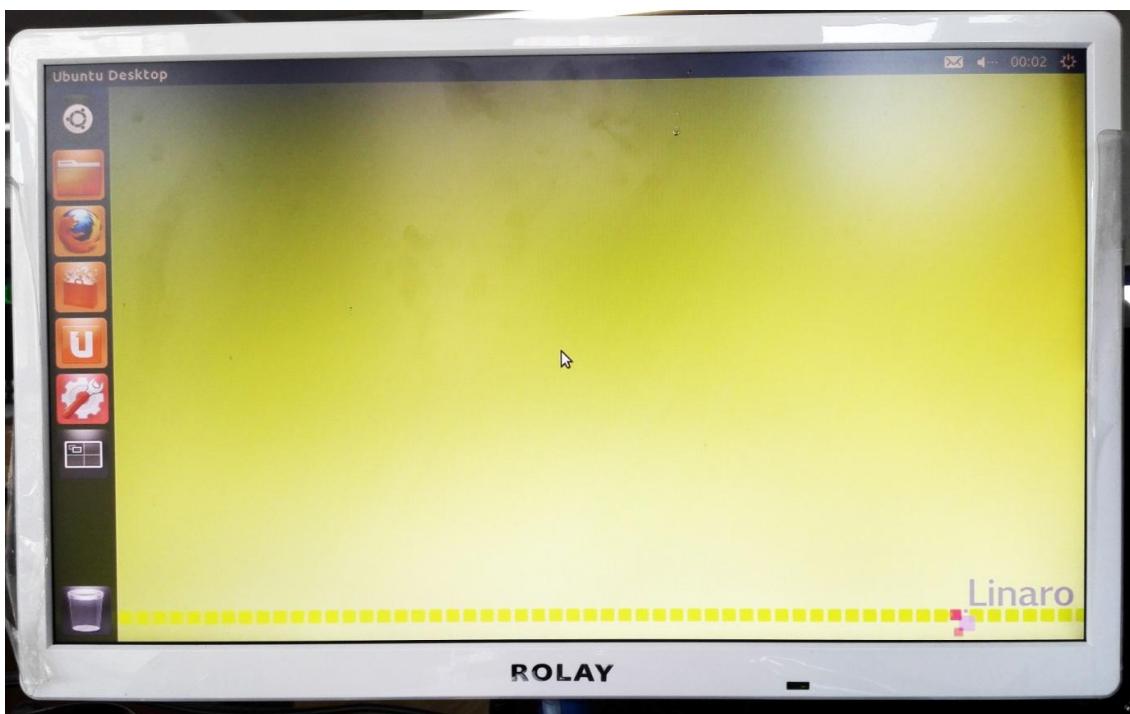
串口打开



电源打开

2、UBUNTU 系统界面

开机后，SD 卡启动，出现 UBUNTU 系统界面，表示成功进入 LINUX 系统。移动鼠标，可进行相应操控。



3、网口测试

3.1 设置开发板 IP 地址。

在软件中设置的开发板 IP 地址“192.168.1.XXX”。

操作：在 putty 中输入：ifconfig eth0 192.168.1.XXX

注意：开发板 IP 地址可自行定义，不能与本地主机 IP 地址相同。例如：主机 IP 地址是 192.168.1.100，可以在 putty 软件中输入：ifconfig eth0 192.168.1.118。

```
root@linaro-ubuntu-desktop:~# ifconfig eth0 192.168.1.118
root@linaro-ubuntu-desktop:~#
```

Putty 中设置开发板 IP 地址

3.2 测试主机与开发板的网口通信

在 PC 端打开控制台。控制台中输入命令：ping 192.168.1.XXX, 即设置的开发板 IP 地址。例如：开发板的 IP 地址是 192.168.1.118，在控制台中输入：ping 192.168.1.118。

注意：如果用 ping 不通过，应该给开发板重新设置 IP 地址，重复网口测试操作。



打开控制台

```
C:\Users\Administrator>ping 192.168.1.118

正在 Ping 192.168.1.118 具有 32 字节的数据:
来自 192.168.1.118 的回复: 字节=32 时间<1ms TTL=64

192.168.1.118 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms, 最长 = 0ms, 平均 = 0ms
```

网口测试通过

至此，开机测试全部结束。

CH02_ZYNQ_VIVADO 软件安装

2.1 VIVADO 软件介绍

“一提起 Xilinx 的开发环境，人们总是先会想起 ISE，而对 Vivado 不甚了解。其实，Vivado 是 Xilinx 公司于 2012 推出的新一代集成设计 环境。虽然目前其流行度并不高，但可以说 Vivado 代表了未来 Xilinx FPGA 开发环境的变化趋势。所以，作为一个 Xilinx FPGA 的开发使用 者，学习掌握 Vivado 是趋势，也是必然。

作为开发者，首先肯定有以下疑惑：既然已经有 ISE 存在了，为何 Xilinx 公司又花大力气去搞什么 Vivado 呢？在 Vivado Design Suite User Guide : Getting Started(UG910) 中提到，推出 Vivado 是为了提高设计者的效率，它能显著增加 Xilinx 的 28nm 工艺的可编程逻辑器件的设计、综合与 实现效率。可以推测，随着 FPGA 进入 28nm 时代，ISE 工具似乎就有些“不合时宜”了，硬件提升了，软件不提升的话，设计效率必然受影响。正是出于这一考虑，Xilinx 公司于 2008 年便开始筹划推出新一代的软件开发环境，经历 4 年时间打造出了 Vivado 工具这一巅峰之作。

必须说明的是，Vivado 并不是 ISE 的升级版，它是全新的另一个 Xilinx FPGA 的开发工具（事实上，ISE 并没有因为 Vivado 的出现而挂 掉也不可能挂掉，Vivado 2012.2 推出的同时 ISE 也更新到了 ISE14.7）。以前在 ISE 里面经常出现的像 XST、Core Generator 等工具在 Vivado 里面已经不复存在，开发者可以将 Vivado 理解为 Xilinx 为高端 FPGA 专门开发的一款开发工具。

Vivado 目前只支持 Xilinx 的 28nm 工艺的 7 系列 FPGA，包括 ZYNQ、Virtex-7 系列、Kintex-7 系列和 Artix-7 系列，不支持其 它系列的 FPGA。这不难理解，人家本身就是为高端而生的开发工具，没必要去支持低端。而 ISE14.2 支持全系列的 FPGA，这也好理解，高端酒店就是 为高富帅开的，低端酒店屌丝可进，高富帅也不会拦嘛。对于开发者，如果使用非 7 系列的 FPGA 器件，那么 ISE 是不二选择，但是如果使用 7 系列的 FPGA，Vivado 的开发效率必然完爆 ISE 了。”

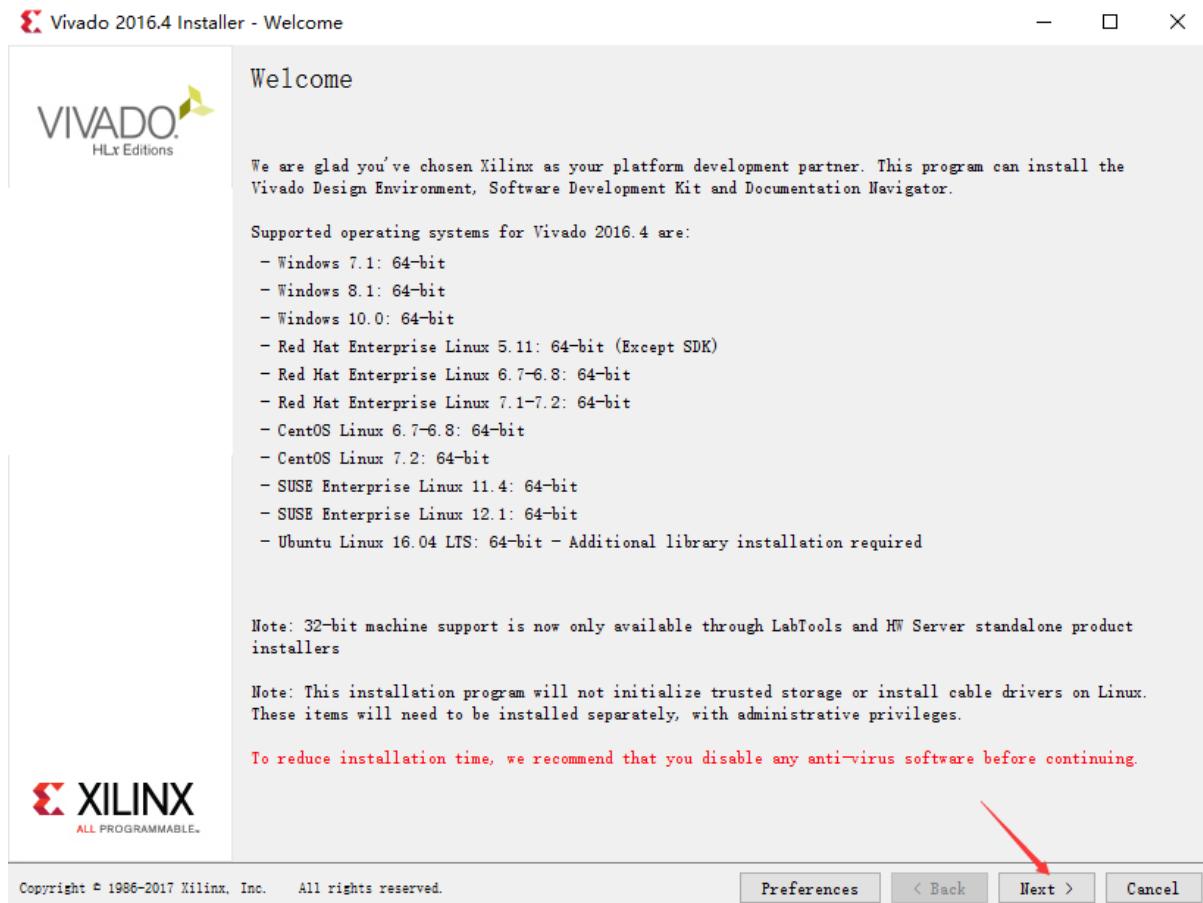
安装 vivado 的过程，其实很简单，但是需要注意一个问题，安装时一定把 SDK 选上，避免不必要的麻烦。

2.2 VIVADO 软件安装(适合所有 vivado 安装)

安装软件前请退出杀毒软件，以免造成安装失败，比如 360 安全卫士

Step1: 双击  xsetup.exe 进行程序安装

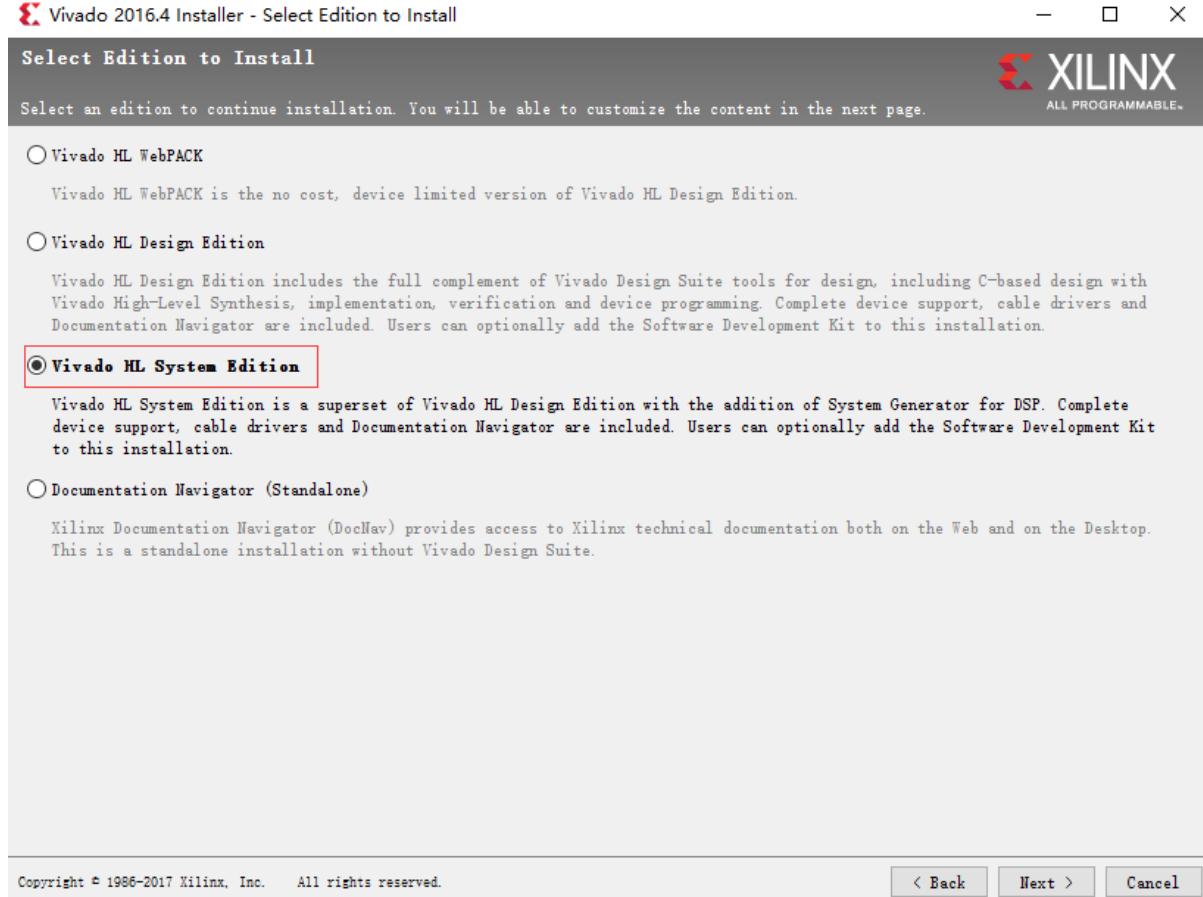
Step2: 驱动欢迎界面后单击 NEXT



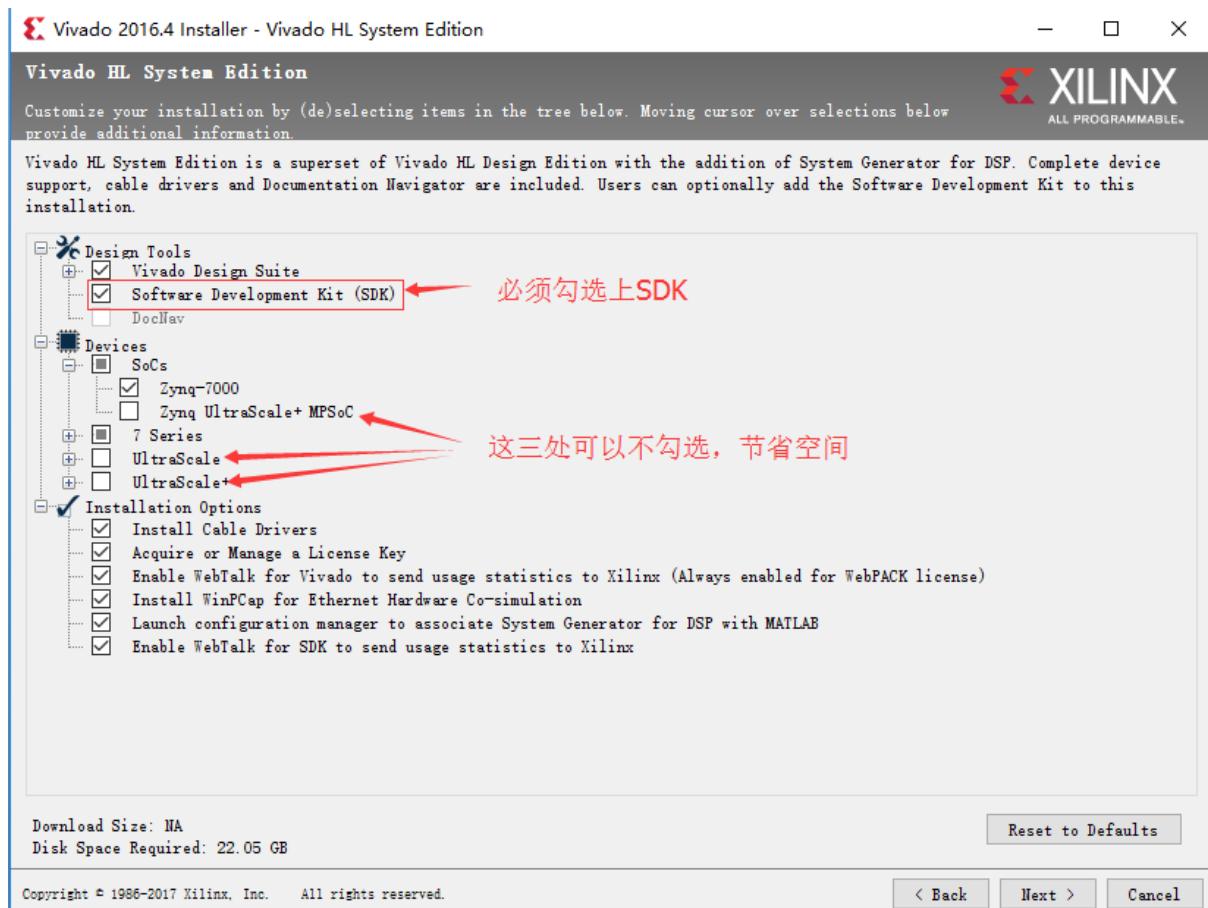
Step3:全勾选上全部，全部同意，单击NEXT



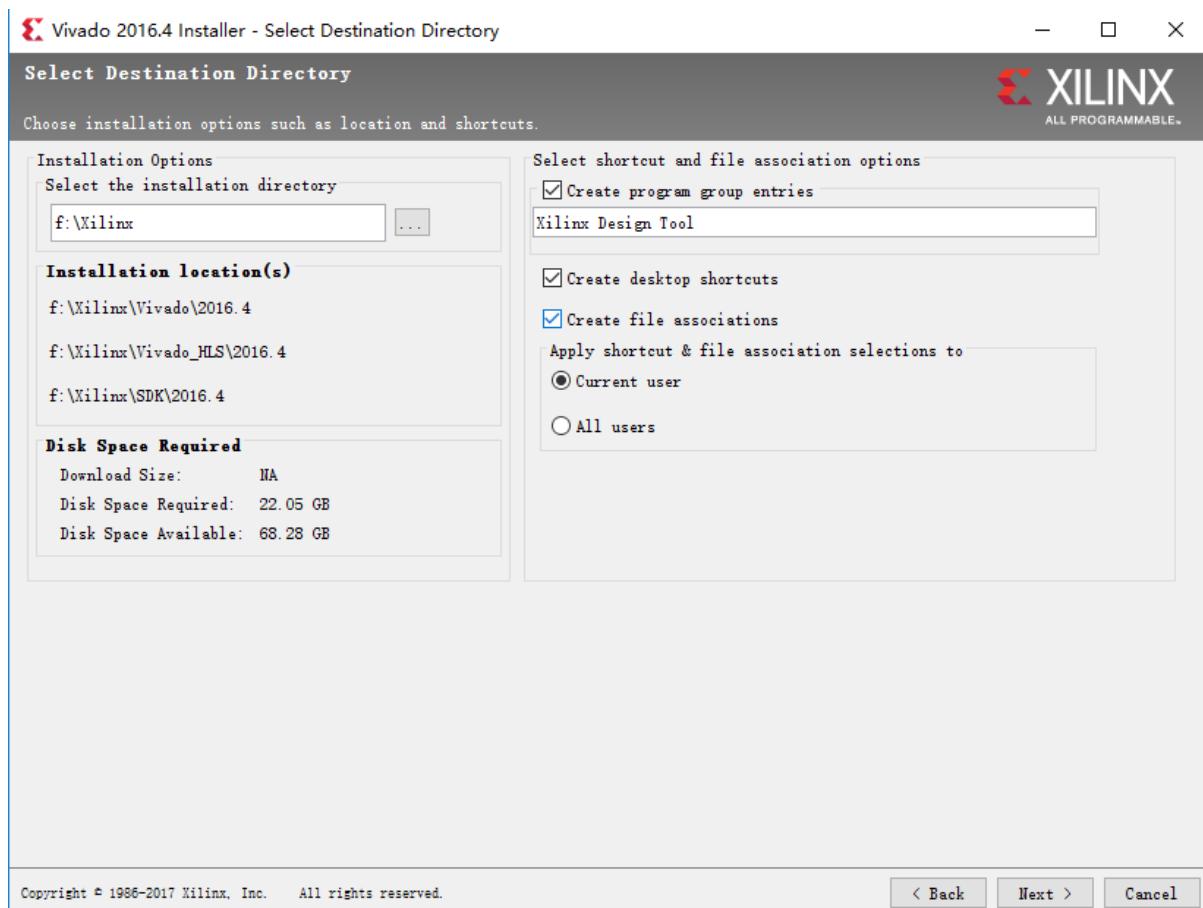
Step4:选择安装Vivado HL System Edition 单击NEXT(这个版本功能最全)



Step5:必须勾选上SDK，其他默认，然后单击NEXT



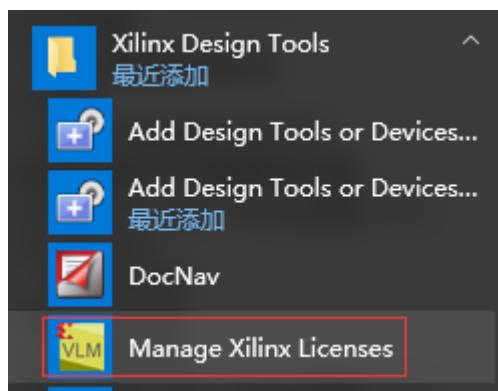
Step6:设置安装目录，然后单击NEXT进行安装



Step7:安装大概需要半个多小时，

2.3 VIVADO 软件注册

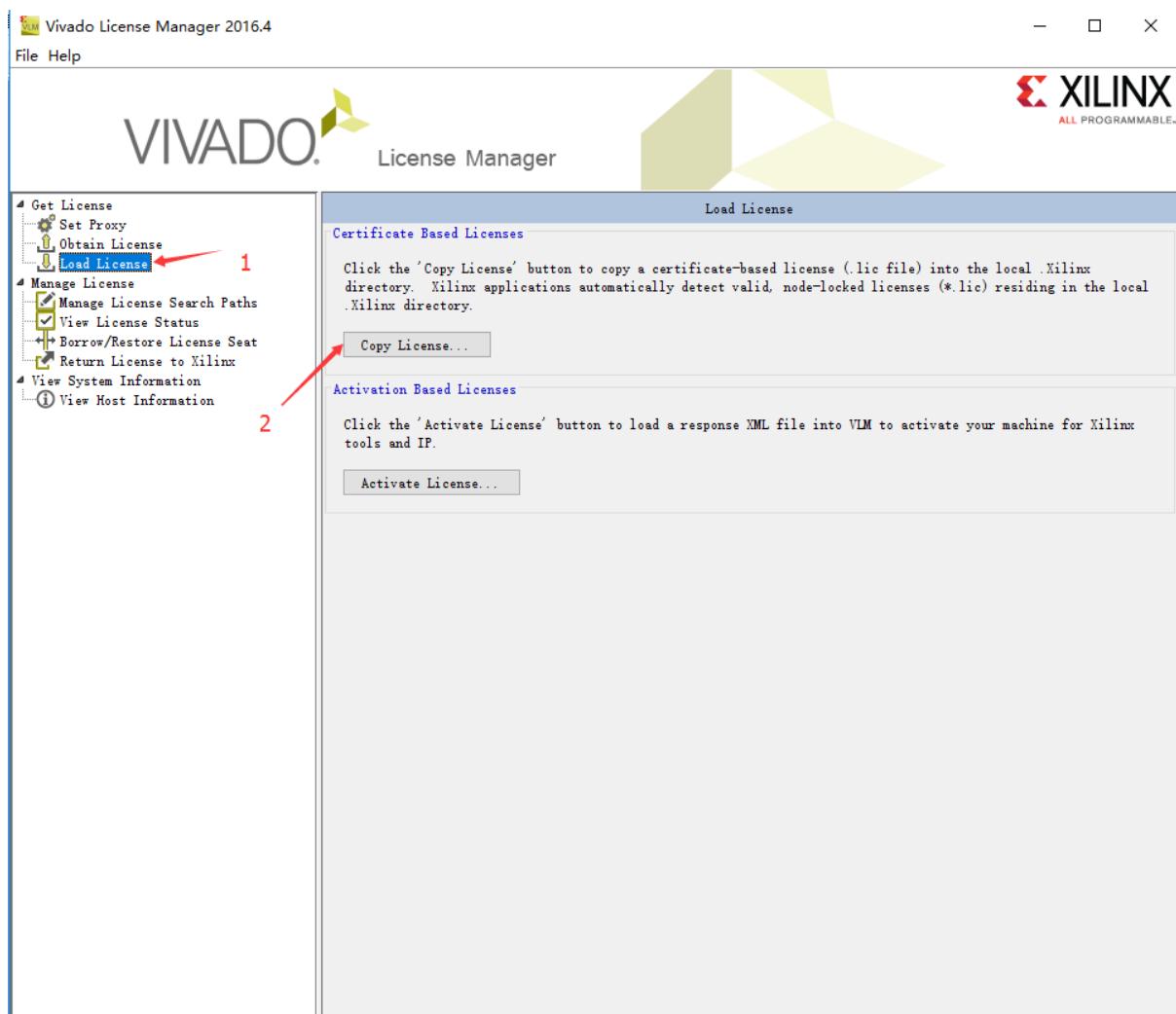
Step1:在开始菜单中找到 vivado 的文件夹，单击 Manage Xilinx license 注册



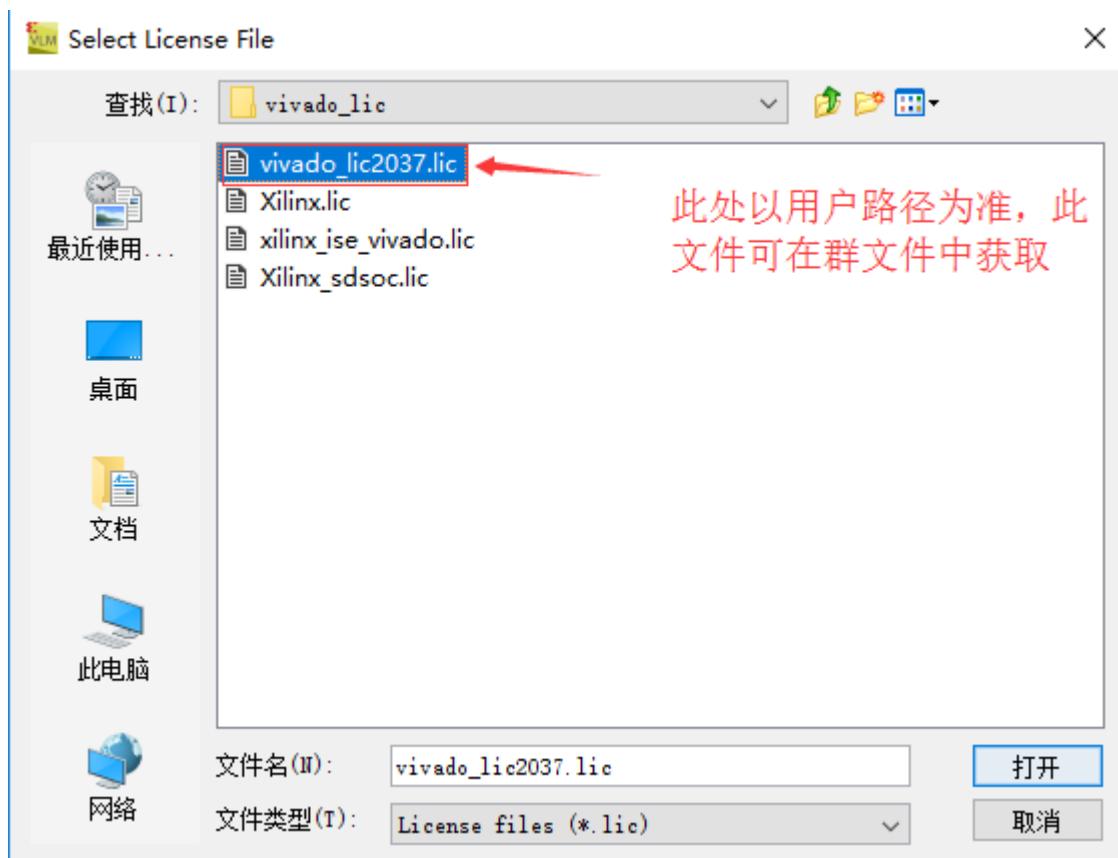
Step2:弹出下图界面后

1、单击 load license

2、单击 Copy license



Step3: Copy license 选择打开



Step4: 弹出添加成功，单击 OK 结束



2.3 本章小结

本章详细描述了如何安装 VIVAOD2016.4 以及如果加载 license,在安装的过程中一定要关闭杀毒软件，以免造成安装失败。

CH03_USB 下载器驱动安装及下载程序

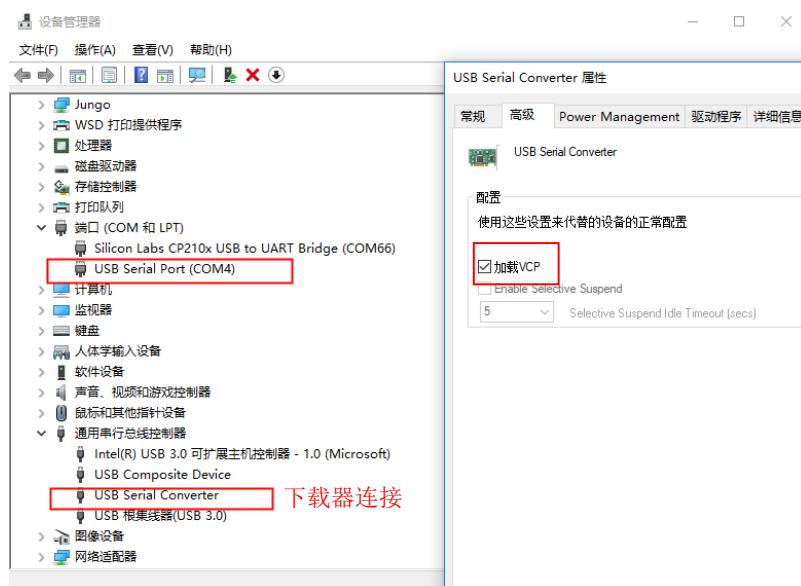
3.1 下载器驱动的安装

需要手动安装下载器驱动分为两种情况：

一种是软件自动安装驱动未成功。默认情况下安装 VIVADO 的时候会自动安装好下载器驱动程序，但是也有例外，比如杀毒软件禁止安装驱动，或者安装 VIVADO 的时候没有勾选安装驱动，这时需要手动安装。另外一种是已安装驱动，但在使用的过程中驱动破损，需要重新安装。

下面对这两种情况下下载器驱动安装进行说明，首先，**连接下载器到电脑**，查看设备管理器中下载器驱动连接状态。然后根据对应情况进行处理。

1、下载器驱动安装正常，下载器可以正常连接



下载器正常连接状态

在设备管理器查看下载器连接状态。图中为下载器正确识别状态。注意：由于开发板的串口也会识别成这样，如果开发者想知道这个设备是不是 USB JTAG 可以采取先不接通开发板串口，如果设备管理器只有这么一个设备，那就是正常识别了。

注意：在 USB Serial Converter 属性中勾选“加载 VCP”，则在端口中能看到下载器的端口号，即 USB Serial Port。不勾选则端口中不显示下载器的 COM 号。不影响实际操作，凭个人喜好。

2、下载器驱动安装不成功，手动安装下载器驱动

如果下载器驱动转件时，未安装成功，设备管理器中未显示连接，如图中所示，则需要手动安装。

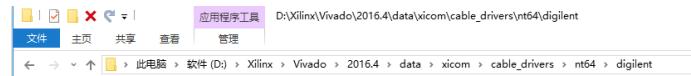


下载器安装未成功

安装方法：

- 拔掉连接到电脑的下载器
- 找到驱动安装程序的路径

笔者的 VIVADO 是安装在 D 盘下的，驱动安装程序的路径如下图：双击.exe 文件进行安装，一路 NEXT 到结束就按照好了。



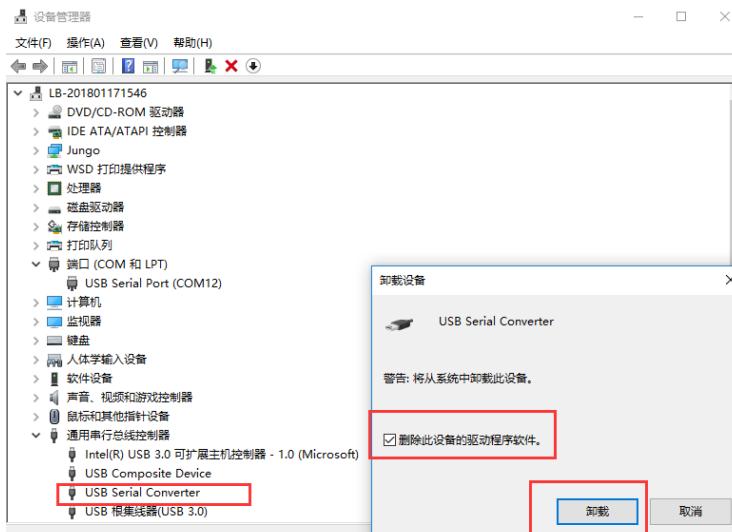
- 重新插入下载器，查看设备管理器，这个时候电脑就能识别出下载器。

3、下载器驱动损坏，需要手动重新安装

在使用的过程中，如果下载器驱动损坏，需要重新手动安装下载器。

安装方法：

- 电脑连接下载器，卸载下载器驱动



b、拔掉连接到电脑的下载器

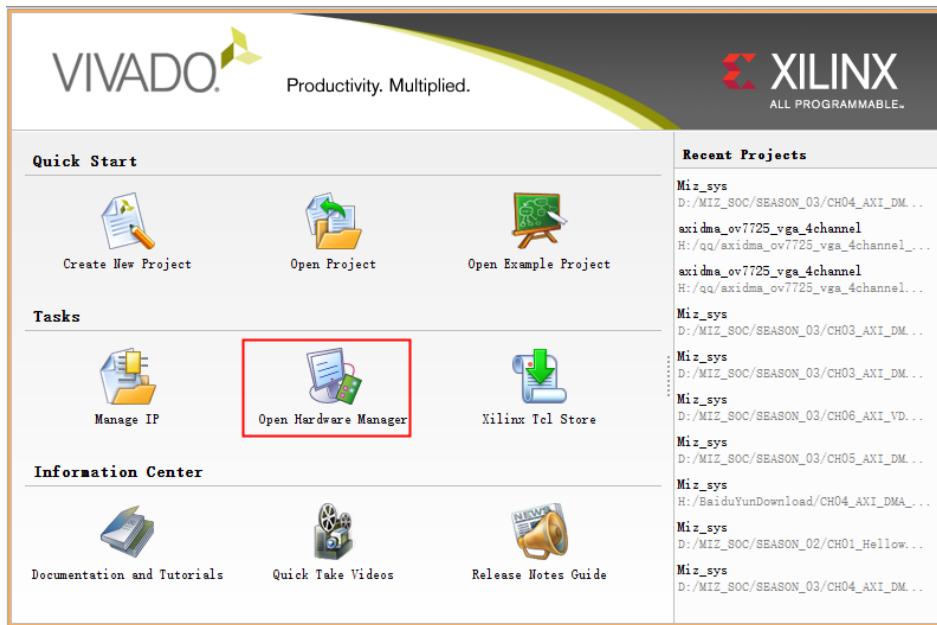
c、找到驱动安装程序的路径

笔者的 VIVADO 是安装在 D 盘下的，驱动安装程序的路径如下图：双击 install_digilent.exe 文件进行安装，一路 NEXT 到结束就按照好了。

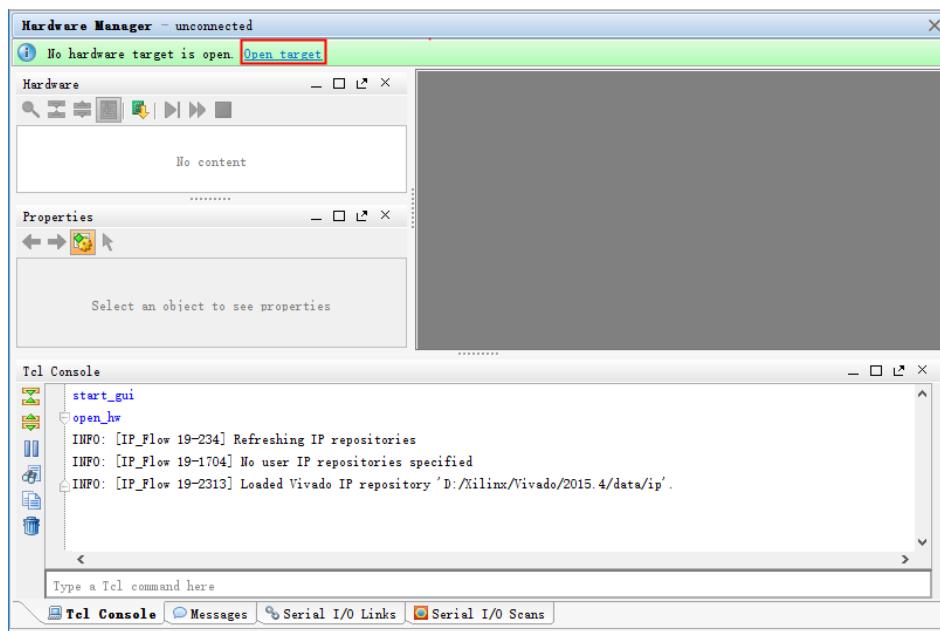
d、重新插入下载器，查看设备管理器，这个时候电脑就能识别出下载器。

3.2 下载 runled 工程的 bit 文件验证板子和下载器工作正常

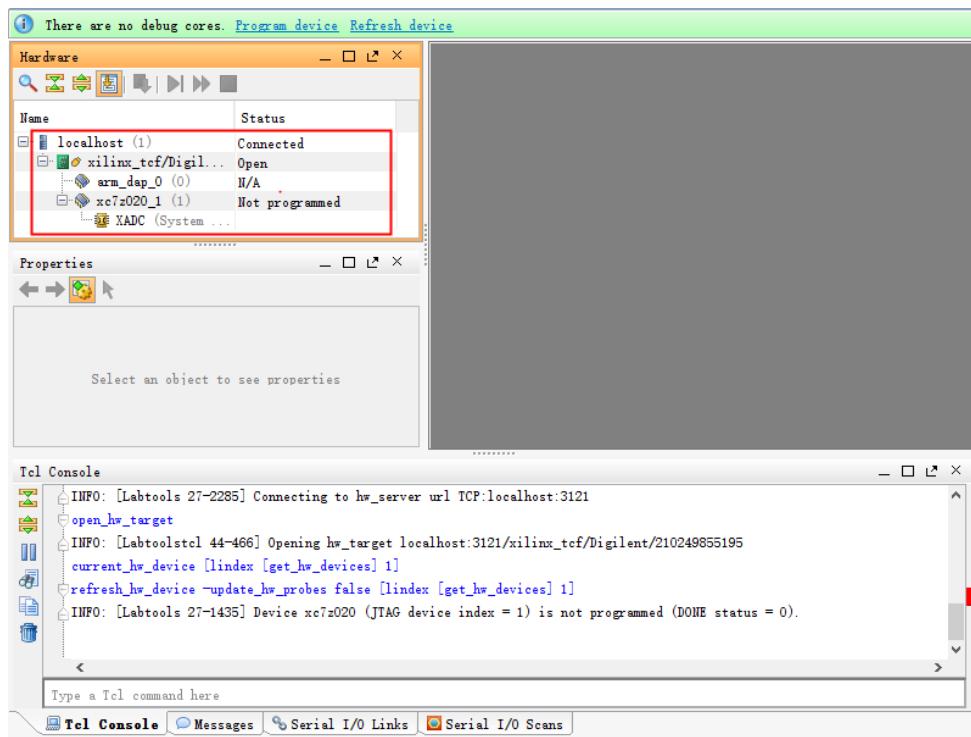
Step1:启动 VIVADO 软件，并且鼠标左击 Open Hardware Manager 图标控件



Step2: 鼠标左击 Open_target



Step3:如下图显示，正确检测到了设备



Step4:下载 CH07 的程序看运行效果

3.3 下载器使用需要注意的问题

1、下载器出厂前都经过严格测试了，出问题的几率很低，所以读者在使用的时候特别是第一次使用请对照本课程安装驱动和测试。

2、有些故障比如无法识别到芯片，也有可能是转接头的安装松动了，可以手指用力安

装紧固。

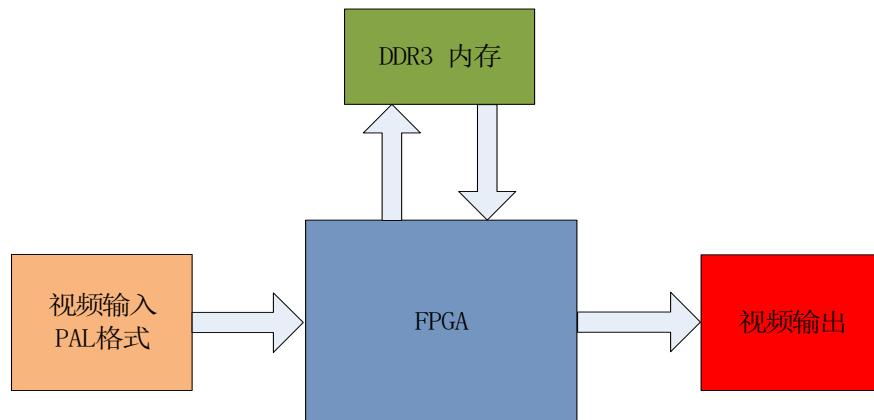
3、当打开多个 VIVADO 工程的时候并且有一个 VIVADO 工程连接了 JTAG，其他工程会连接失败。可以选择关闭已经连接的工程，或者断开连接。

4、有时候 VIVADO 工程死机了，关闭工程后，没有释放占用 USB JTAG 的系统资源也会导致，JTAG 无法识别到芯片，可以重启电脑，或者到进程管理器去关闭死机的进程。

CH04_FPGA 设计 Verilog 基础（一）

4.1 Verilog HDL 代码规范

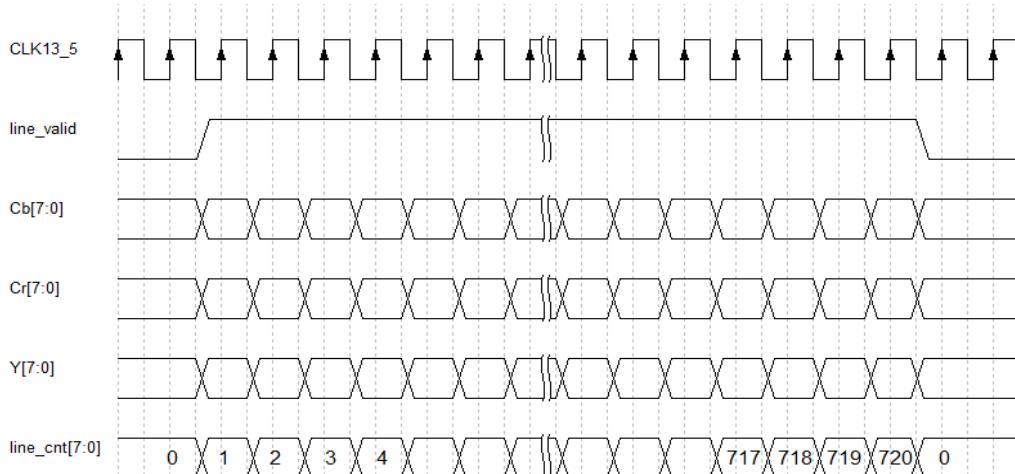
◆ 项目构架设计



项目的构架用于团队的沟通，以及项目设计的全局把控

◆ 接口时序设计规范

模块和模块之间的通过模块的接口实现关联，因此规范的时序设计，对于程序设计的过程，以及程序的维护，团队之间的沟通都是非常必要的。



◆ 命名规则

1、顶层文件

对象+功能+top

比如： video_oneline_top

2、逻辑控制文件

介于顶层和驱动层文件之间

对象+ctr

比如： ddr_ctrl.v

3、驱动程序命名

对象+功能+dri

比如： lcd_dri.v、uart_rxd_dri.v

4、参数文件命名

对象+para

比如： lcd_para.v

5、模块接口命名：文件名+u

比如 lcd_dir lcd_dir_u(.....)

6、模块接口命名：特征名+文件名+u

比如 mcb_read c3_mcb_read_u

7、程序注释说明

```
////////////////////////////////////////////////////////////////  
// Company: milinker corporation  
// Engineer:jinry tang  
// WEB:www.milinker.com  
// BBS:www.osrc.cn  
// Create Date: 07:28:50 07/31/2015  
// Design Name: FPGA STREAM  
// Module Name: FPGA_USB  
// Project Name: FPGA STREAM
```

```
// Target Devices: XC6SLX16-FTG256/XC6SLX25-FTG256 Mis603
// Tool versions: ISE14.7
// Description: CY7C68013A SLAVE FIFO communication with fpga
// Revision: V1.0
// Additional Comments:
//1) _i input
//2) _o output
//3) _n activ low
//4) _dg debug signal
//5) _r delay or register
//6) _s state machine
////////////////////////////////////////////////////////////////
```

8、端口注释

input Video_vs_i,//输入场同步入

9、信号命名

命名总体规则：对象+功能（+极性）+特性

10、时钟信号

对象+功能+特性

比如：phy_txclk_i、sys_50mhz_i

11、复位信号

对象+功能+极性+特性

比如：phy_RST_N_i、sys_RST_N_i

12、延迟信号

对象+功能+特性 1+特征 2

比如：fram_sync_i_r0、fram_sync_i_r1

13、特定功能计数器

对象+cnt

比如: line_cnt、div_cnt0、div_cnt1

功能+cnt

比如: wr_cnt、rd_cnt

对象+功能+cnt

比如: fifo_wr_cnt、mcb_wr_cnt、mem_wr_cnt

对象+对象+cnt

比如: video_line_cnt、video_fram_cnt

14、一般计数器

cnt+序号

用于不容易混淆的计数

比如: cnt0、cnt1、cnt2

15、时序同步信号

对象+功能+特性

比如: line_sync_i、fram_sysc_i

16、使能信号

功能+en

比如: wr_en、rd_en

对象+功能+en

比如: fifo_wr_en、mcb_wr_en

4.2 技术背景

大规模集成电路设计制造技术和数字信号处理技术，近三十年来，各自得到了迅速的发展。这两个表面上看来没有什么关系的技术领域实质上是紧密相关的。因为数字信号处理系统往往要进行一些复杂的数学运算和数据的处理，并且又有实时响应的要求，它们通常是由

高速专用数字逻辑系统或专用数字信号处理器所构成，电路是相当复杂的。因此只有在

高速大规模集成电路设计制造技术进步的基础上，才有可能实现真正有意义的实时数字信号处理系统。对实时数字信号处理系统的要求不断提高，也推动了高速大规模集成电路设计制造技

术的进步。现代专用集成电路的设计是借助于电子电路设计自动化（EDA）工具完成的。学习和掌握硬件描述语言（HDL）是使用电子电路设计自动化（EDA）工具的基础。

笔者建议 Verilog，虽然很多学校古董级的老师还在教 VHDL。当然 VHDL 也是要了解的，因为这门古老的语言的历史遗留问题，现在还有很多 VHDL 的模块，有的时候我们要拿来主义，所以还有必要了解下的。但是历史的车轮总是在前进，优胜劣汰。也许不久的将来 Verilog 也会被 C,C++这种高级语言代替。

为了更方面地切入主题，笔者假设，你已经学过单片机，并且掌握 C 语言。因为单片机，和 C 语言，可以说是当代大学生的一项基本能力。有了这个基础，再学习其他现代计算机编程，算法，才能达到事半功倍的效果。如果你还不会单片机和 C 语言，建议你首先学会单片机，或者 C 语言。当然，这只是笔者的建议，不会单片机，或者 C 语言，并不代表学不好 Verilog 语言。

学过单片机的都知道，我们的程序代码是一条指令一条指令来执行的。CPU 首先通过总线，读取一条指令，然后解析这条指令，再然后执行这条指令。我们写的 C 代码总是一条一条地执行。如果我们同时要处理 10 个子程序，那么 CPU 必须一个个子程序来执行。如果有些实时性较高的，如扫描下矩阵键盘，VGA 刷个屏，都需要中断来实现。如果刷屏时间比较长，就会影响到你按键的灵敏度。另外比如，我们的单片机在用串口接收数据，并且也要发送数据，同时我们的单片机要处理外部的 IO 信号，如果我们的 IO 信号非常快，并且有几百个信号，可能同一个时刻触发，很显然，如果这些信号比较快，那么我们的单片机，就没法实现了。

这是笔者简单举了两种情况，那么如果使用 FPGA 就可以很方便地解决以上问题。由于 FPGA 的并行性，不管是扫描键盘，还是扫描 VGA，都可以把它们做成独立的模块，时间上没有冲突，每个模块可以同时执行。

再比如用一个 FPGA，就可以同时完成串口的收发，以及 IO 的监控，因为 FPGA 的程序实际上就是电路，是瞬间就完成了，我们只要用 Verilog 写出来相应功能的程序模块，这些模块是同时运行的。

这样看来 FPGA 真是太强大了，太完美了。但不要高兴地太早，由于 FPGA 可以在一个时钟内，完成多条语句的赋值，但是如果赋值必须有个前后顺序呢？也就是需要一步步的完成，怎么办？如果说并行控制是 FPGA 的优点，那么顺序控制就是他的不足之处。世界上永远没有完美的东西，我们在获得一种优势的时候，往往也获得了一种劣势。但是，办法总比问题多。

■ 顺序控制的第一种办法——状态机设计

可以说，我们用 Verilog 来写程序，状态机无处不在。顾名思义，通过设计状态机，我们可以控制 Verilog 让他该快的时候快，该慢的时候慢，该做什么的时候就做什么。这才是我们想要的。状态机是很不错的东西，初学者对他望而生畏，而熟悉 Verilog 语言的人都对其会爱不释手。

■ 顺序控制的第二种方法——FPGA 中运行 CPU

FPGA 也可以运行 CPU?是的，没错，FPGA 也可以像单片机一样使用，这样我们就可以用 C 代码来一条条指令来执行了，这不是太强大了？是的，没错。关键的问题是，我们是可以把一些逻辑控制顺序复杂的事情用 C 代码来实现，而实时处理的部分仍然用 Verilog 来实现。并且那部分 Verilog 可以被 C 代码控制。Xilinx FPGA 目前支持的 CPU 有 Microblaze, ARM9,POWERPC, CortexA9 (zynq 就 Xilinx 比较新的一款片子，完美的将 CortexA9 和 FPGA 整合到一起，有兴趣的可以淘宝搜索 MiZ702) 其中 Microblaze 是一款软 CPU，是软核。ARM9, CortexA9 和 POWERPC 是硬核。这里有两个概念：

1) 软核就是用代码就能现的 CPU 核，这种核配置灵活，成本较低。但是要占用 FPGA 宝贵的资源。

2) 硬核就是一块电路，做到 FPGA 内部，方便使用，性能更高。比如 Xilinx 的 DDR 内存控制器，就是一种硬核，其运行速度非常高，我们只要做些配置，就可以方便使用。

两种核可谓各有所长。

■ FPGA 还是 ASIC

根据具体看情况而定，从我们上面的一些介绍，笔者相信你已经有了一定的判断能力了。笔者的建议是，低速场合，实时性要求的低的地方用 ASIC，有些功能用 ASIC 方便的用 ASIC，成本低的用 ASIC。排除那些可以不用 FPGA 地方，那么剩余的就要考虑是不是用 FPGA 来实现更加方便。一般来说，FPGA 程序开发相对来说要难度大一些，并且成本要高一些。

讲了这么多的背景知识，我们来看一小段代码：

```
u32sum (a,b)
{
    a=a+1;
    b=b+1;
    c=a+b;
    return c;
}
```

```
always@(posedgeclk) begin  
    a = a + 1;  
    b = b + 1;  
    c = a + b;  
end
```

同样是实现了求和，但是，C 代码需要 N 多个（很多）CPU 周期才得出结果,而用 Verilog 一个 clk 周期就计算出来了。

或许现在你还不知道为什么。没关系，下面的内存笔者讲解 Verilog 语言基础。

4.3 Verilog 最最基础语法

Verilog 和 C 在外形上有很大相识的地方，有了 C 基础背景，Verilog 看起来就并不陌生。

C 语言和 Verilog 的关键词和结构对比：

C	Verilog
sub-function	module, function, task
if-then-else	if-then-else
Case	Case
{,}	begin, end
For	For
While	While
Break	Disable
Define	Define
Int	Int
Printf	monitor, display,strobe

C 语言和 Verilog 运算符对比：

C	Verilog	功能
*	*	乘
/	/	除
+	+	加
-	-	减
%	%	取模
!	!	反逻辑
&&	&&	逻辑且
		逻辑或
>	>	大于
<	<	小于
>=	>=	大于等于
<=	<=	小于等于
==	==	等于
!=	!=	不等于
~	~	位反相
&	&	按位逻辑与
		按位逻辑或
^	^	按位逻辑异或
~^	~^	按位逻辑同或
>>	>>	右移
<<	<<	左移
?:	?:	同等於 if-else 敘述

真是太振奋人心了，一切都是这么熟悉。学好 FPGA 已经没有心理障碍了。

4.4 关键字

信号部分：

input 关键词，模块的输入信号，比如 input C1k, C1k 是外面关键输入的时钟信号；

output 关键词，模块的输出信号，比如 output [3:0]Led; 这个地方正好是一组输出信号。其中[3:0]表示 0~3 共 4 路信号。

inout 模块输入输出双向信号。这种类型，我们的例子 24LC02 中有使用。数总线的通信中，这种信号被广泛应用；

wire 关键词，线信号。例如：wire C1_Clk; 其中 C1_Clk 就是 wire 类型的信号；

线信号，三态类型，我们一般常用的线信号类型有 input, output, inout, wire；

reg 关键词，寄存器。**和线信号不同，它可以在 always 中被赋值，经常用于时序逻辑中。**比如 reg[3:0]Led; 表示了一组寄存器。

结构部分：

```
module()
...
endmodule
```

代表一个模块，我们的代码写在这个两个关键字中间

always@()括号里面是敏感信号。这里的 always@(posedge Clk) 敏感信号是 posedge Clk 含义是在上升沿的时候有效，敏感信号还可以 negedge Clk 含义是下降沿的时候有效，这种形式一般时序逻辑都会用到。还可以是*这个一符号，如果是一个*则表示一直是敏感的，一般用于组合逻辑。

assign 用来给 output, inout 以及 wire 这些类型进行连线。assign 相当于一条连线，将表达式右边的电路直接通过 wire(线)连接到左边，左边信号必须是 wire 型 (output 和 inout 属于 wire 型)。当右边变化了左边立马变化，方便用来描述简单的组合逻辑。示例：

```
wire a, b, y;  
assign y = a & b;、
```

这些语句含义上都和高级语言一样：

```
if(...)begin
```

```
.....
```

```
End
```

```
if(...)begin
```

```
.....
```

```
end
```

```
else begin
```

```
.....
```

```
End
```

```
if(...)begin
```

```
.....
```

```
end
```

```
else if(...)begin
```

```
.....
```

```
end
```

```
case(...)
```

```
.....
```

```
endcase
```

begin..... end 作用域范围，类似于 C 的大括号。用法举例：

```
always@ (posedge clk) begin
```

```
.....
```

```
end
```

符号部分：（我们这里 FALSE 为 0， TRUE 为 1）

“;” 分号用于每一句代码的结束，以表示结束，和 C 语言一样。

“:” 冒号，用在数组，和条件运算符以及 case 语句结构中。case 结构会在后面讲解。

“<=” 赋值符号，非阻塞赋值，在一个 always 模块中，所有语句一起更新。**它也可以表示小于等于，具体是什么含义编译环境根据当前编程环境判断，如果“<=”是用在一个 if 判断里如：if(a <= 10)；当然就表示小于等于了。**

“=” 阻塞赋值，或者给信号赋值，如果在 always 模块中，这条语句被立刻执行。**阻塞赋值和非阻塞赋值将再后面详细举例说明。**

“+, -, *, /, %” 是加、减、乘、除运算符号，这些使用和 C 语言基本是一样的，当你用到这些符号时，编译后会自动生成或者消耗 FPGA 原有的加法器或是乘法器等。其中符号 /, % 会消耗大量的逻辑，谨慎使用。

“<” 小于，比如 A<B 含义就是 A 和 B 比较，如果 A 小于 B 就是 TRUE，否则为 FALSE。

“<=” 小于等于，比如 A<=B 含义就是 A 和 B 比较，如果 A 小于等于 B 就是 TRUE，否则为 FALSE。

“>” 大于，比如 A>B 含义就是 A 和 B 比较，如果 A 大于 B 就是 TRUE，否则为 FALSE。

“>=” 大于等于，比如 A>=B 含义就是 A 和 B 比较，如果大于等于 B 就是 TRUE，否则为 FALSE。

“==” 等于等于，比如 A==B 含义就是 A 和 B 比较，如果 A 等于 B 就是 TRUE，否则为 FALSE。

“!=” 不等于，A!=B 含义是 A 和 B 比较，如果 A 不等于 B 就是 TRUE，否则为 FALSE.

“`>>`”右移运算符，比如 `A>>2` 表示把 A 右移 2 位。

“`<<`”左移运算符，比如 `A<<2` 表示把 A 左移 2 位。

“`~`”按位取反运算符，比如 `A=8' b1111_0000`;则`~A` 的值为 `8' b0000_1111`;

“`&`”按位与，比如 `A=8' b1111_0000;B=8' b1010_1111`;则 `A&B` 结果为 `8' b1010_0000`;

“`^`”异或运算符，比如 `A=8' b1111_0000;B=8' b1010_1111`;则 `A^B` 结果为 `8' b0101_1111`;

“`&&`”逻辑与，比如 `A==1, B==2`;则 `A&&B` 结果为 TRUE;如果 `A==1, B==0`, 则 `A&&B` 结果为 FALSE，一般用于条件判断。

`A = B ? C : D` 是一个条件运算符，含义是如果 B 为 TRUE 则把 C 连线 A, 否则把 D 连线 A。B 通常是个条件判断，用小括弧括起:

```
assign C1_Clk = (C1==25'd24999999) ? 1 : 0 ;
```

`C1_Clk`, 是一个 wire 类型的信号，当 `C1==25'd24999999` 时候，连线到 1，否则连线到 0.

“`{}`”在 Verilog 中表示拼接符，`{a, b}`这个的含义是将括号内的数按位并在一起，比如：`{1001, 1110}`表示的是 `10011110`。拼接是 Verilog 相对于其他语言的一大优势，在以后的编程中请慢慢体会。

参数部分：

`parameter` 定义一个符号 a 为常数（十进制 180 找个常量的定义等效方式）：

```
parameter a = 180;//十进制，默认分配长度 32bit(编译器默认)
parameter a = 8'd180;//十进制
parameter a = 8'haa; //十六进制
parameter a = 8'b1010_1010; //二进制
```

预处理命令

```
//-----
`include file1.v
//-----
`define X = 1;
//-----
`define Y;
`ifndef Y
    Z=1;
`else
    Z=0;
`endif
//-----
```

有的时候我们一些公共的宏参数，我们可以放在一个文件中，比如这个文件名字为 xx.v 那么我们可以`include xx.v 就可以包含找个文件中定义的一些宏参数。我还是来详细说明下吧！

话说 Verilog 的`include 和 C 语言的 include 用法是一样一样的，要说区别可能就在于那个点吧。

include 一般就是包含一个文件，对于 Verilog 这个文件里的内容无非是一些参数定义，所以这里再提几个关键字：`ifdef `define `endif（他们都带个点，呵呵）。

他们联合起来使用，确实能让你的程序多样化，就拿 VGA 程序说事吧。

首先，你可以新建一个.v 文件（可以直接新建一个 TXT，让后将后缀换成.v）其实这个后缀没所谓，.v 也是可以的，我觉得，写成.v 更能体现出这个文件的意义。

假设有个 lcd_para.v 文件，内容如下：

```
// 640 * 480
`ifdef VGA_640_480_60FPS_25MHz
`define H_FRONT 11'd16
`define H_SYNC 11'd96
`define H_BACK 11'd48
`define H_DISP 11'd640
`define H_TOTAL 11'd800
```

```

`define V_FRONT 11'd10
`define V_SYNC 11'd2
`define V_BACK 11'd33
`define V_DISP 11'd480
`define V_TOTAL 11'd525
`endif
// 800 * 600
`ifdef VGA_800_600_60MHz
`define H_FRONT 11'd56
`define H_SYNC 11'd120
`define H_BACK 11'd64
`define H_DISP 11'd800
`define H_TOTAL 11'd1040

`define V_FRONT 11'd37
`define V_SYNC 11'd6
`define V_BACK 11'd23
`define V_DISP 11'd600
`define V_TOTAL 11'd666
`endif
//-----
`define H_Start (^H_SYNC + `H_BACK)
`define H_END (^H_SYNC + `H_BACK + `H_DISP)
`define V_Start (^V_SYNC + `V_BACK)
`define V_END (^V_SYNC + `V_BACK + `V_DISP)

```

这里为 VGA 定义了两种分辨率，通过`define VGA_800_600_60MHz 或 VGA_640_480_60FPS_25MHz 或`define VGA_800_600_72FPS_50MHz 来决定使用哪种分辨率。

比如，我的 xxx.v 文件想调用 lcd_para.h，那么 xxx.v 我可以写到：

```

`define VGA_800_600_60MHz //这句要放在"lcd_para.h"的上面，不然编译不通过
`include "lcd_para.h"

```

那么 xxx.v 文件中就可以用 lcd_para.v 中的参数了，且对应是 VGA_800_600_60MHz 下的参数。

其次` include "lcd_para.v" 这个路径也有一点讲究，xxx.v 作为引用 lcd_para.v 的

文件它和 lcd_para.v 在同一文件夹下才能怎么写，就是相对路径一说了。也就是以 xxx.v 为当前路径去引索 lcd_para.v 文件的位子。所以如果他们不再一个文件夹那么请写出更详细（正确）的路径。顺便说一句，lcd_para.v 添不添加到工程是无所谓的，只要路径

对了即可，当然我还是建议添加到工程，且和.v 文件放在同一文件夹下，以方便观察和管理。

4.5 Verilog 中数值表示的方式

如果我们要表示一个十进制是 180 的数值，在 Verilog 中的表示方法如下：

二进制：8' b1010_1010; //其中“_”是为了容易观察位数，可有可无。

十进制：8' d180;

16 进制：8' hAA;

4.6 阻塞赋值和非阻塞赋值详解

说到阻塞赋值和非阻塞赋值，是很多初学者很迷惑的地方。原因是 C 语言没有可以类比的东西。

学习 FPGA 和单片机最大的区别在于，学 FPGA 时，你必须时刻都有着时钟的概念。不像单片机时钟相关性比较差，FPGA 你必须却把握每一个时钟。

首先来说说非阻塞赋值，这个在时序逻辑中随处可见：

```
reg A;
reg B;
always @(posedge clk)
begin
    A <= 1'b1;
    B <= 1'b1;
    /**
     * 或者
     */
    B <= 1'b1;
    A <= 1'b1;
    *****/
end
```

这段程序里，A 和 B 是同时被赋值的，具体是说在时钟的上升沿来的时刻，A 和 B

同时被置 1。调换 A 和 B 的上下顺序，将得到相同的结果。

接着看另外一段程序：

```
reg A;  
reg B;  
always @(posedge clk)  
begin  
    A <= 1'b1;  
end  
always @(posedge clk)  
begin  
    B <= 1'b1;  
end
```

这段程序，与第一段程序也是完全等价的，A 和 B 在同一时刻被赋值。两段程序综合出的逻辑也是完全相同的。这就是非阻塞赋值的特点，体现了 FPGA 的并行性！

接着来看阻塞赋值，它少了一个非，表示会阻塞住，那么体会下这个阻塞：

```
always @(posedge clk)  
begin  
    A = 1'b1;  
    B <= 1'b1;  
end
```

看到，上面这个程序是阻塞和非阻塞的混合使用，一般教材是极力反对这种写法的。其实只要你理解了，有的时候这种用法还能帮上大忙。只不过，不理解的话乱用会导致时序违规。

回到正题，我们这么写是为了更好的理解阻塞赋值：当时钟上升沿来临的时刻，首先 A 会被置 1，然后 B 寄存器再置 1。区别就是 A 和 B 不再同时置 1。A 要比 B 提前零点几纳秒。这样就出现了先后顺序。这个过程还是在一个时钟内完成的，但是数据到达 B 寄存器相比上面两段程序晚了那么零点几纳秒！

当我们的时钟跑的比较慢的时候，比如 50M，一个周期有 20ns，那么这么短暂的延时基本可以忽略不计，但是随着设计的复杂，以及时钟速度的提高，这样的语句就要小心。

假设，我们要计算 AB 求和再除以 2 的结果。先用非阻塞方法去实现，由于 AB 求和再除以 2 是两个步骤，而非阻塞所以的事情都在一个时钟完成，所以这里我们用状态机，将两个步骤分配到两个时钟里去完成：

```
module unblock
(
    input clk_i,
    input rst_n_i,
    output reg [4:0]result_o
);
    reg [3:0]A;
    reg [3:0]B;
    reg [4:0]C;
    reg i;
    always @(posedge clk_i )
        if(!rst_n_i)
            begin
                #2
                A <= 4'd4;
                B <= 4'd12;
                C <= 5'd0;
                result_o = 5'd0;
            end
        else begin
            #2
            C <= A + B;
            result_o <= (C >> 1);
        end
    endmodule
```

第一个时钟上升沿来临时，完成 $C \leftarrow A + B$ ；

第二个时钟来临时完成 $result \leftarrow (C >> 1)$ ；

求出结果，这个过程耗费两个时钟。（不考虑复位消耗的时钟）

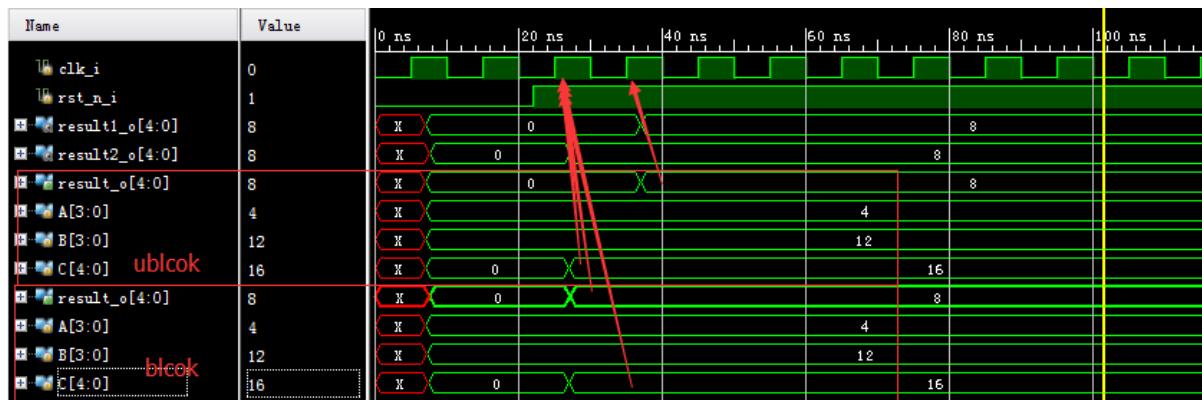
再来，用添加阻塞的方式实现：

```
module block(
    input clk_i,
    input rst_n_i,
    output reg [4:0]result_o
);

    reg [3:0]A;
    reg [3:0]B;
    reg [4:0]C;

    always @(posedge clk_i)
        if(!rst_n_i)
            begin
                #2    A = 4'd4;
                #0.2 B = 4'd12;
                #0.2 C = 5'd0;
                #0.2 result_o = 5'd0;
            end
        else begin
                #2    C = A + B;
                #0.2 result_o = (C >> 1);
            end
endmodule
```

仿真结果：



先通过阻塞的方法提前得到 C 的值，再将 C 右移 1 位，达到除以 2 的效果。整个过程耗时一个时钟。

以上的程序并没有什么实际的参考价值，但是解释清楚阻塞和非阻塞赋值，它已经做到了~~

讲到这里，笔者以最快的速度，最简单的方式，让读者学习了 Verilog 语言的语法部分。具备这些基础知识，下面笔者将带你通过代码来学习 Verilog 语言。最后，笔者提一点建议，学习 Verilog 多看别人写的优秀的代码，多看官方提供的代码和文档。其中官方提供的代码，很多时候代表了最新的用法，或者推荐的用法。读者学习，首先把最最基础的掌握好，这样，在项目中遇到了问题，也能快速学习，快速解决。

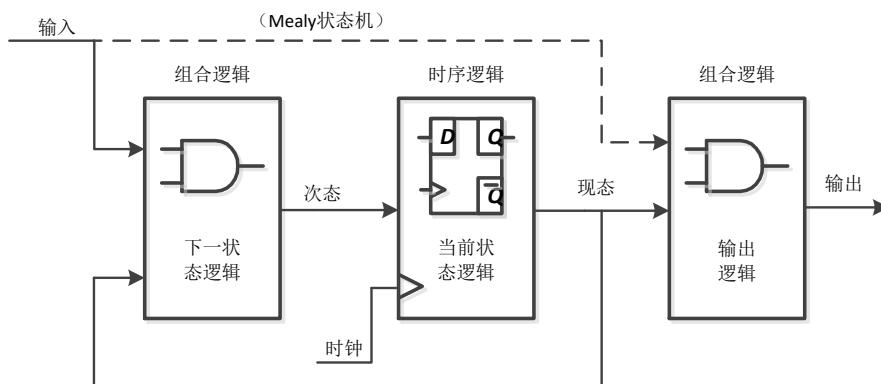
对于理论知识的学习，没必要一开始就研究得那么深刻，只是搞理论学习，对于学习 Verilog 语言，或者 FPGA 开发是不实际的，要联系理论和实践结合。多仿真，多验证，多问题，多学习，多改进。

CH05_FPGA 设计 Verilog 基础（二）

5.1 状态机设计

状态机是许多数字系统的核心部件，是一类重要的时序逻辑电路。通常包括三个部分：一是下一个状态的逻辑电路，二是存储状态机当前状态的时序逻辑电路，三是输出组合逻辑电路。通常，状态机的状态数量有限，称为有限状态机（FSM）。由于状态机所有触发器的时钟由同一脉冲边沿触发，故也称之为同步状态机。

根据状态机的输出信号是否与电路的输入有关分为 Mealy 型状态机和 Moore 型状态机。电路的输出信号不仅与电路当前状态有关，还与电路的输入有关，称为 Mealy 型状态机，而电路的输出仅仅与各触发器的状态，不受电路输入信号影响或无输入，称为 Moore 型状态机。其标准模型如下所示：



状态机的状态转移图，通常也可根据输入和内部条件画出。一般来说，状态机的设计包含下列设计步骤：

- 根据需求和设计原则，确定是 Moore 型还是 Mealy 型状态机；
- 分析状态机的所有状态，对每一状态选择合适的编码方式，进行编码；
- 根据状态转移关系和输出绘出状态转移图；
- 构建合适的状态机结构，对状态机进行硬件描述。

状态机的描述通常有三种方法，称为一段式状态机，二段式状态机和三段式状态机。状态机的描述通常包含以下四部分：

- 1) 利用参数定义语句 parameter 描述状态机各个状态名称，即状态编码。状态编码通常有很多方法包含自然二进制编码，One-hot 编码，格雷编码等；
- 2) 用时序的 always 块描述状态触发器实现状态存储；

- 3) 使用敏感表和 case 语句（也采用 if-else 等价语句）描述状态转换逻辑；
- 4) 描述状态机的输出逻辑。

下面根据状态机的三种方法，来比较各种方法的优劣。

5.2 一段式状态机

```
module detect_1(
    input clk_i,
    input rst_n_i,
    output out_o
);

reg out_r;
//状态声明和状态编码
reg [1:0] state;
parameter [1:0] S0=2'b00;
parameter [1:0] S1=2'b01;
parameter [1:0] S2=2'b10;
parameter [1:0] S3=2'b11;

always@(posedge clk_i)
begin
    if(!rst_n_i)begin
        state<=0;
        out_r<=1'b0;
    end
    else
        case(state)
            S0 :
                begin
                    out_r<=1'b0;
                    state<= S1;
                end
            S1 :
                begin
                    out_r<=1'b1;
                    state<= S2;
                end
            S2 :
                begin
                    out_r<=1'b0;
                    state<= S3;
                end
            S3 :
                begin
                    out_r<=1'b1;
                    state<= S0;
                end
        endcase
    end
endmodule
```

```
end

S1 :
begin
    out_r<=1'b1;
    state<= S2;
end

S2 :
begin
    out_r<=1'b0;
    state<= S3;
end

S3 :
begin
    out_r<=1'b1;
end

endcase

end

assign out_o=out_r;

endmodule
```

一段式状态机是应该避免使用的，该写法仅仅适用于非常简单的状态机设计，不符合组合逻辑与时序逻辑分开的原则，整个结构代码也不清晰，不利用维护和修改。

5.3 两段式状态机

```
module detect_2(
    input clk_i,
    input rst_n_i,
    output out_o
);
    reg out_r;
    //状态声明和状态编码
    reg [1:0] Current_state;
```

```
reg [1:0] Next_state;
parameter [1:0] S0=2'b00;
parameter [1:0] S1=2'b01;
parameter [1:0] S2=2'b10;
parameter [1:0] S3=2'b11;

//时序逻辑: 描述状态转换
always@(posedge clk_i)
begin
    if(!rst_n_i)
        Current_state<=0;
    else
        Current_state<=Next_state;
end

//组合逻辑:描述下一状态和输出
always@(*)
begin
    out_r=1'b0;
    case(Current_state)
        S0 :
            begin
                out_r=1'b0;
                Next_state= S1;
            end
        S1 :
            begin
                out_r=1'b1;
                Next_state= S2;
            end
        S2 :
            begin
                out_r=1'b0;
                Next_state= S3;
            end
    end
end
```

```
begin
    out_r=1'b0;
    Next_state= S3;
end

S3 :
begin
    out_r=1'b1;
    Next_state=Next_state;
end
endcase
end
assign out_o=out_r;
endmodule
```

两段式状态机采用两个 always 模块实现状态机的功能，其中一个 always 采用同步时序逻辑描述状态转移，另一个 always 采用组合逻辑来判断状态条件转移。两段式状态机是推荐的状态机设计方法。

5.4 三段式状态机

```
module detect_3(
    input clk_i,
    input rst_n_i,
    output out_o
);
    reg out_r;
    //状态声明和状态编码
    reg [1:0] Current_state;
    reg [1:0] Next_state;
    parameter [1:0] S0=2'b00;
    parameter [1:0] S1=2'b01;
    parameter [1:0] S2=2'b10;
    parameter [1:0] S3=2'b11;
```

```
//时序逻辑：描述状态转换
always@(posedge clk_i)
begin
    if(!rst_n_i)
        Current_state<=0;
    else
        Current_state<=Next_state;
end

//组合逻辑：描述下一状态
always@(*)
begin
    case(Current_state)
        S0:
            Next_state = S1;
        S1:
            Next_state = S2;
        S2:
            Next_state = S3;
        S3:
            begin
                Next_state = Next_state;
            end
        default :
            Next_state = S0;
    endcase
end

//输出逻辑：让输出 out， 经过寄存器 out_r 锁存后输出， 消除毛刺
always@(posedge clk_i)
begin
```

```
if(!rst_n_i)
    out_r<=1'b0;
else
begin
    case(Current_state)
        S0,S2:
            out_r<=1'b0;
        S1,S3:
            out_r<=1'b1;
        default :
            out_r<=out_r;
    endcase
end
end

assign out_o=out_r;
```

三段式状态机在第一个 always 模块采用同步时序逻辑方式描述状态转移，第二个 always 模块采用组合逻辑方式描述状态转移规律，第三个 always 描述电路的输出。通常让输出信号经过寄存器缓存之后再输出，消除电路毛刺。这种状态机也是比较推崇的，主要是由于维护方便，组合逻辑与时序逻辑完全独立。

CH06_FPGA 设计 Verilog 基础（三）

一个完整的设计，除了好的功能描述代码，对于程序的仿真验证是必不可少的。学会如何去验证自己所写的程序，即如何调试自己的程序是一件非常重要的事情。而 RTL 逻辑设计中，学会根据硬件逻辑来写测试程序，即 Testbench 是尤其重要的。Verilog 测试平台是一个例化的待测（MUT）模块，重要的是给它施加激励并观测其输出。逻辑模块与其对应的测试平台共同组成仿真模型，应用这个模型可以测试该模块能否符合自己的设计要求。

编写 TESTBENCH 的目的是为了对使用硬件描述语言设计的电路进行仿真验证，测试设计电路的功能、性能与设计的预期是否相符。通常，编写测试文件的过程如下：

- 产生模拟激励（波形）；
- 将产生的激励加入到被测试模块中并观察其响应；
- 将输出响应与期望值相比较。

6.1 完成的 Test bench 文件结构

通常，一个完整的测试文件其结构为

```
module Test_bench();//通常无输入无输出  
信号或变量声明定义  
逻辑设计中输入对应 reg 型  
逻辑设计中输出对应 wire 型  
使用 initial 或 always 语句产生激励  
例化待测试模块  
监控和比较输出响应  
endmodule
```

6.2 时钟激励设计

下面列举出一些常用的封装子程序，这些是常用的写法，在很多应用中都能用到。

```
/*-----  
时钟激励产生方法一： 50% 占空比时钟  
-----*/  
parameter ClockPeriod=10;
```

```
initial
begin
    clk_i=0;
    forever
        #(ClockPeriod/2) clk_i=~clk_i;
    end
/*
时钟激励产生方法二：50% 占空比时钟
*/
initial
begin
    clk_i=0;
    always #(ClockPeriod/2) clk_i=~clk_i;
end

/*
时钟激励产生方法四：产生固定数量的时钟脉冲
*/
initial
begin
    clk_i=0;
    repeat(6)
        #(ClockPeriod/2) clk_i=~clk_i;
    end

/*
时钟激励产生方法五：产生非占空比为 50% 的时钟
*/
initial
begin
    clk_i=0;
```

```
forever
begin
    #(ClockPeriod/2)-2) clk_i=0;
    #(ClockPeriod/2)+2) clk_i=1;
end
end
```

6.3 复位信号设计

```
/*
-----复位信号产生方法一：异步复位-----
*/
initial
begin
    rst_n_i=1;
    #100;
    rst_n_i=0;
    #100;
    rst_n_i=1;
end

/*
-----复位信号产生方法二：同步复位-----
*/
initial
begin
    rst_n_i=1;
    @ (negedge clk_i)
    rst_n_i=0;
    #100;           //固定时间复位
end
```

```
repeat(10) @ (negedge clk_i); //固定周期数复位
  @ (negedge clk_i)
  rst_n_i=1;
end

/*
-----  
复位信号产生方法三：复位任务封装  
-----*/
task reset;
  input [31:0] reset_time; //复位时间可调，输入复位时间
  RST_ING=0; //复位方式可调，低电平或高电平
begin
  rst_n=RST_ING; //复位中
  #reset_time; //复位时间
  rst_n_i=~RST_ING; //撤销复位，复位结束
end
endtask
```

6.4 特殊信号设计

```
/*
-----  
特殊激励信号产生描述一：输入信号任务封装  
-----*/
task i_data;
  input [7:0] dut_data;
begin
  @(posedge data_en); send_data=0;
  @(posedge data_en); send_data=dut_data[0];
  @(posedge data_en); send_data=dut_data[1];
  @(posedge data_en); send_data=dut_data[2];
```

```
@(posedge data_en); send_data=dut_data[3];
@(posedge data_en); send_data=dut_data[4];
@(posedge data_en); send_data=dut_data[5];
@(posedge data_en); send_data=dut_data[6];
@(posedge data_en); send_data=dut_data[7];
@(posedge data_en); send_data=1;
#100;
end
endtask
//调用方法: i_data(8'hXX);
/*
-----特殊激励信号产生描述二：多输入信号任务封装
-----*/
task more_input;
input [7:0] a;
input [7:0] b;
input [31:0] times;
output [8:0] c;
begin
repeat(times)          //等待 times 个时钟上升沿
    @(posedge clk_i)
        c=a+b;           //时钟上升沿 a, b 相加
end
endtask
//调用方法: more_input(x,y,t,z); //按声明顺序
/*
-----双向信号描述一：inout 在 testbench 中定义为 wire 型变量
-----*/
//为双向端口设置中间变量 inout_reg 作为 inout 的输出寄存，其中 inout 变
//量定义为 wire 型，使用输出使能控制传输方向
//inout bir_port;
```

```
wire bir_port;
reg bir_port_reg;
reg bi_port_oe;
assign bi_port=bi_port_oe ? bir_port_reg : 1'bz;
/*
-----  
双向信号描述二：强制 force  
-----*/  
//当双向端口作为输出口时，不需要对其进行初始化，而只需开通三态门  
//当双向端口作为输入时，只需要对其进行初始化并关闭三态门，初始化赋值需  
//使用 wire 型数据，通过 force 命令来对双向端口进行输入赋值  
//assign dinout=(!en) din :16'hz; 完成双向赋值  
  
initial  
begin  
    force dinout=20;  
    #200  
    force dinout=dinout-1;  
end  
/*
-----  
特殊激励信号产生描述三：输入信号产生,一次 SRAM 写信号产生  
-----*/  
initial  
begin  
    cs_n=1;           //片选无效  
    wr_n=1;           //写使能无效  
    rd_n=1;           //读使能无效  
    addr=8'hxx;       //地址无效  
    data=8'hzz;       //数据无效  
    #100;  
    cs_n=0;           //片选有效  
    wr_n=0;           //写使能有效  
    addr=8'hF1;       //写入地址
```

```
data=8'h2C;           //写入数据
#100;
cs_n=1;
wr_n=1;
#10;
addr=8'hxx;
data=8'hzz;
end

/*
-----  
Testbench 中@与 wait
-----*/
//@使用沿触发
//wait 语句都是使用电平触发
initial
begin
    start=1'b1;
    wait(en=1'b1);
    #10;
    start=1'b0;
end
```

6.5 仿真控制语句及系统任务描述

```
/*
-----  
仿真控制语句及系统任务描述
-----*/
$stop      //停止运行仿真, modelsim 中可继续仿真
$stop(n)   //带参数系统任务, 根据参数 0,1 或 2 不同, 输出仿真信息
$finish    //结束运行仿真, 不可继续仿真
```

```
$finish(n) //带参数系统任务，根据参数 0,1 或 2 不同，输出仿真信息
    //0:不输出任何信息
    //1:输出当前仿真时刻和位置
    //2:输出当前仿真时刻、位置和仿真过程中用到的 memory 以及 CPU 时间的
统计
$random      //产生随机数
$random % n //产生范围-n 到 n 之间的随机数
{$random} % n //产生范围 0 到 n 之间的随机数

/*
-----  
仿真终端显示描述
-----*/
$monitor     //仿真打印输出,大印出仿真过程中的变量，使其终端显示
/*
$monitor($time,,,"clk=%d reset=%d out=%d",clk,reset,out);
*/
$display     //终端打印字符串,显示仿真结果等
/*
$display(" Simulation start ! ");
$display(" At time %t,input is %b%b%b,output is %b",$time,a,b,en,z);

*/
$time        //返回 64 位整型时间
$stime       //返回 32 位整型时间
$realtime    //实行实型模拟时间
/*
-----  
文本输入方式：$readmemb/$readmemh
-----*/
//激励具有复杂的数据结构
//verilog 提供了读入文本的系统函数
$readmemb/$readmemh("<数据文件名>,<存储器名>");
```

```
$readmemb/$readmemh("<数据文件名>,<存储器名>,<起始地址>);  
$readmemb/$readmemh("<数据文件名>,<存储器名>,<起始地址>,<结束地址>);  
$readmemb/*读取二进制数据，读取文件内容只能包含：空白位置，注释行，二进制数  
数据中不能包含位宽说明和格式说明，每个数字必须是二进制数字。*/  
$readmemh/*读取十六进制数据，读取文件内容只能包含：空白位置，注释行，十六进  
制数  
数据中不能包含位宽说明和格式说明，每个数字必须是十六进制数字。*/  
/*当地址出现在数据文件中，格式为@hh...h,地址与数字之间不允许空白位  
置，  
可出现多个地址*/  
  
module  
    reg [7:0] memory[0:3];//声明 8 个 8 位存储单元  
    integer i;  
    initial  
        begin  
            $readmemh("mem.dat",memory);//读取系统文件到存储器中的给定地址  
            //显示此时存储器内容  
            for(i=0;i<4;i=i+1)  
                $display("Memory[%d]=%h",i,memory[i]);  
        end  
    endmodule  
  
/*mem.dat 文件内容  
@001  
AB CD  
@003  
A1  
*/  
//仿真输出为  
Memory[0] = xx;  
Memory[1] = AB;
```

```
Memory[2] = CD;  
Memory[3] = A1;
```

6.6 加法器的仿真测试文件编写

上面只例举了常用的 testbench 写法，在工程应用中基本能够满足我们需求，至于其他更为复杂的 testbench 写法，大家可参考其他书籍或资料。

这里提出以下几点建议供大家参考：

- 封装有用且常用的 testbench，testbench 中可以使用 task 或 function 对代码进行封装，下次利用时灵活调用即可；
- 如果待测试文件中存在双向信号(inout)需要注意，需要一个 reg 变量来表示输入，一个 wire 变量表示输出；
- 单个 initial 语句不要太复杂，可分开写成多个 initial 语句，便于阅读和修改；
- Testbench 说到底是依赖 PC 软件平台，必须与自身设计的硬件功能想搭配。

下面具体看一段程序：

```
module add(a,b,c,d,e); // 模块接口  
input [5:0] a; // 输入信号 a  
input [5:0] b; // 输入信号 b  
input [5:0] c; // 输入信号 a  
input [5:0] d; // 输入信号 b  
output[7:0] e; // 求和输出信号  
wire [6:0]outa1,outa2; // 定义输出网线型  
assign e = outa2+outa1; // 把两部分输出结果合并  
/*
```

通常，我们模块的调用写法如下：

被调用的模块名字- 自定义的名字- 括号内信号

这里比如括号内的信号，.ina(ina1)

这种写法最常用，信号的顺序可以调换

另外还有一种写法没可以直接这样写

```
adder u1 (ina1,inb1,outa1);
```

这种写法必须确保信号的顺序一致，这种写法几乎没有人采用

```
/*
adder u1 (.ina(a),.inb(b),.outa(outa1)); // 调用 adder 模块，自定义名字为 u1
adder u2 (.ina(c),.inb(d),.outa(outa2)); // 调用 adder 模块，自定义名字为 u2
endmodule
```

//adder 子模块

```
module adder(ina,inb,outa );// 模块接口
input [5:0] ina; // ina-输入信号
input [5:0] inb; // inb-输入信号
output [6:0] outa; // outa-输出信号
assign outa = ina + inb; // 求和
endmodule // 模块结束
```

仿真文件：

```
`timescale 1ns / 1ps
module add_tb();
reg [5:0] a;
reg [5:0] b;
reg [5:0] c;
reg [5:0] d;
wire[7:0] e;
reg [5:0] i; //中间变量
// 调用被仿真模块模块
add uut (.a(a), .b(b),.c(c),.d(d),.e(e));
initial begin // initial 是仿真用的初始化关键词
a=0;b=0;c=0;d=0; // 必须初始化输入信号
for(i=1;i<31;i=i+1) begin
#10 ;
a = i;
b = i;
```

```

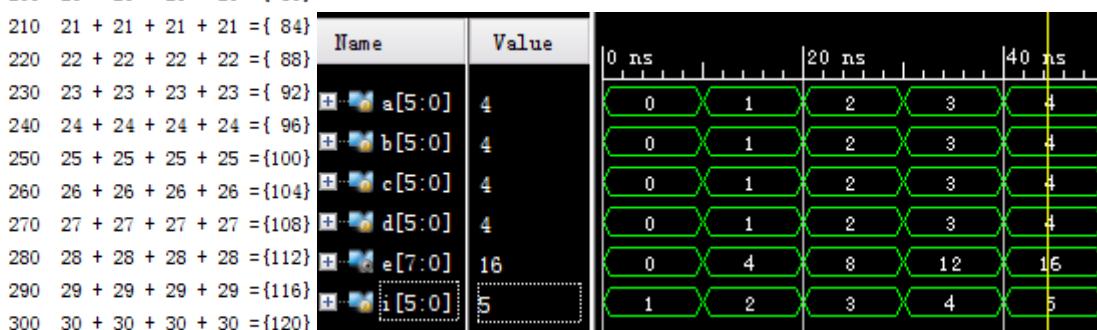
c = i;
d = i;
end // 给输入信号 a 赋值
end
initial begin
$monitor($time,,,"%d + %d + %d + %d ={%d}",a,b,c,d,e); // 信号打印输出
#500 $finish;
end
endmodule

```

```

0 0 + 0 + 0 + 0 ={ 0}
10 1 + 1 + 1 + 1 ={ 4}
20 2 + 2 + 2 + 2 ={ 8}
30 3 + 3 + 3 + 3 ={ 12}
40 4 + 4 + 4 + 4 ={ 16}
50 5 + 5 + 5 + 5 ={ 20}
60 6 + 6 + 6 + 6 ={ 24}
70 7 + 7 + 7 + 7 ={ 28}
80 8 + 8 + 8 + 8 ={ 32}
90 9 + 9 + 9 + 9 ={ 36}
100 10 + 10 + 10 + 10 ={ 40}
110 11 + 11 + 11 + 11 ={ 44}
120 12 + 12 + 12 + 12 ={ 48}
130 13 + 13 + 13 + 13 ={ 52}
140 14 + 14 + 14 + 14 ={ 56}
150 15 + 15 + 15 + 15 ={ 60}
160 16 + 16 + 16 + 16 ={ 64}
170 17 + 17 + 17 + 17 ={ 68}
180 18 + 18 + 18 + 18 ={ 72}
190 19 + 19 + 19 + 19 ={ 76}
200 20 + 20 + 20 + 20 ={ 80}

```



CH07_FPGA_RunLED 创建 VIVADO 工程实验

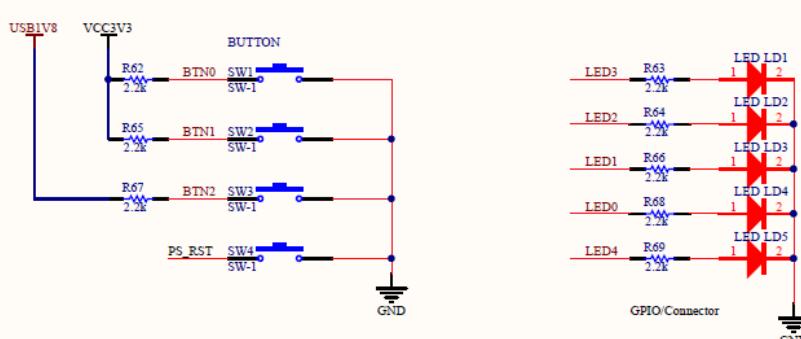
7.1 硬件图片

先来熟悉一下开发板的硬件：LED 部分及按钮部分

MIZ7035:



7.2 硬件原理图



PIN 脚定义(版本：201802):以原理图为标准

CLK_i: C8 RSTn_i(SW2): A8	LED0: B9 LED1: J10 LED2: H11 LED3: G9
------------------------------	--

对应工程：S01/CH7_Runled_New

7.3 新建 VIVADO 工程

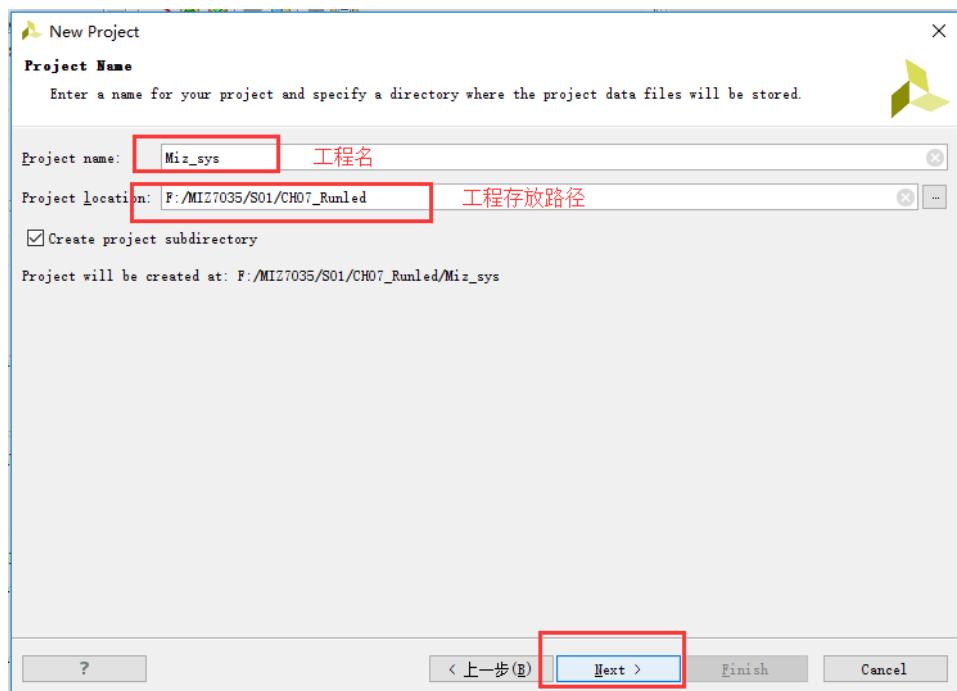
Step1:启动 VIVADO，单击 Create New Project



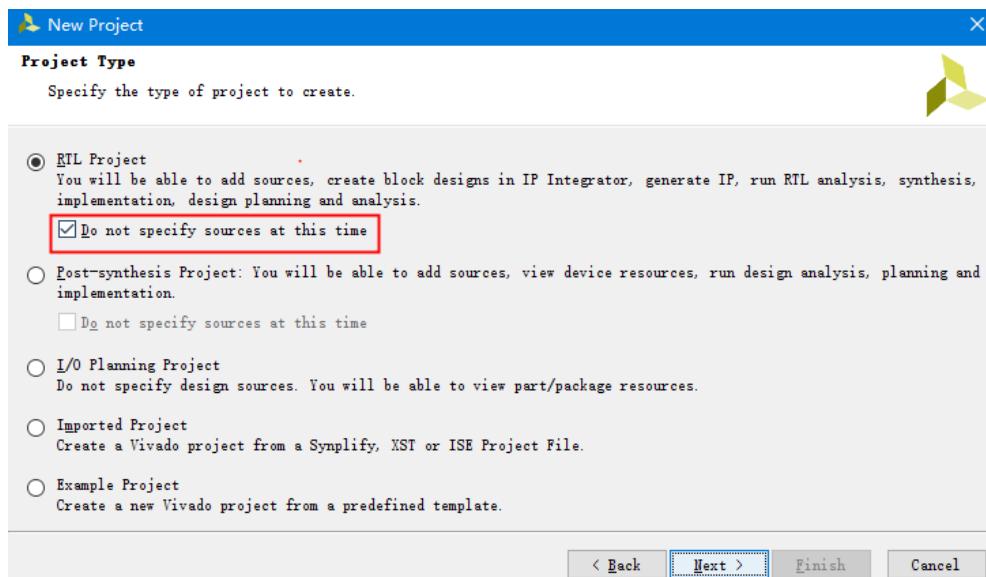
Step2:单击 NEXT



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

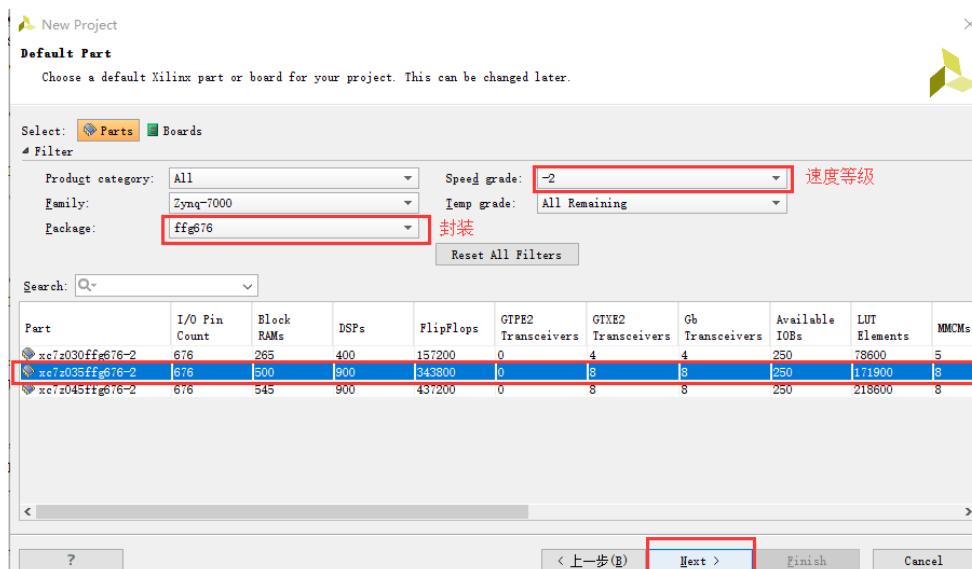


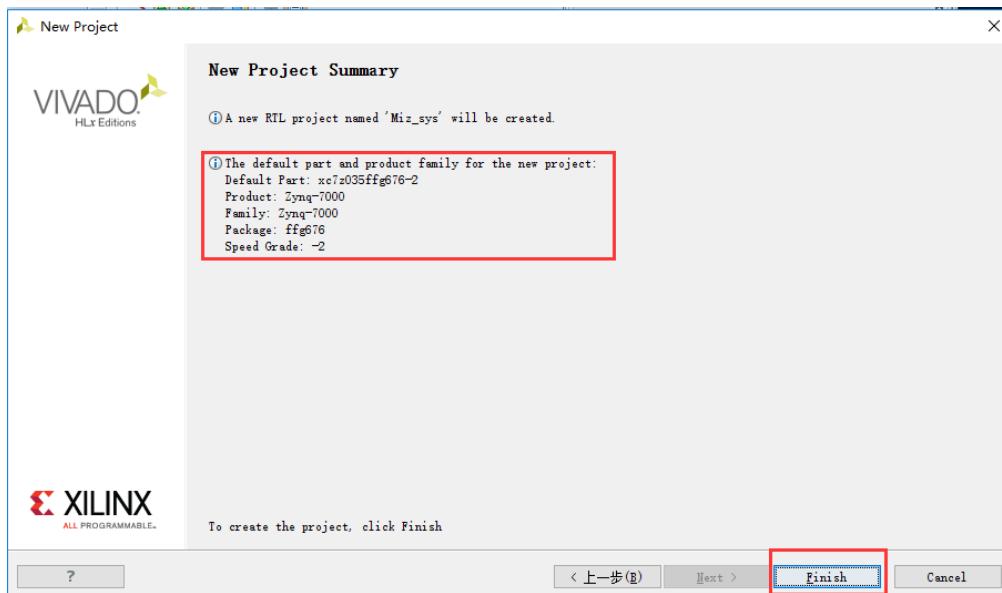
Step 4 :选择 RTL Project 并且勾选复选框，之后单击 NEXT



Step5:选择芯片的型号和封装速度等级:

MiZ7035 如下图所示设置:

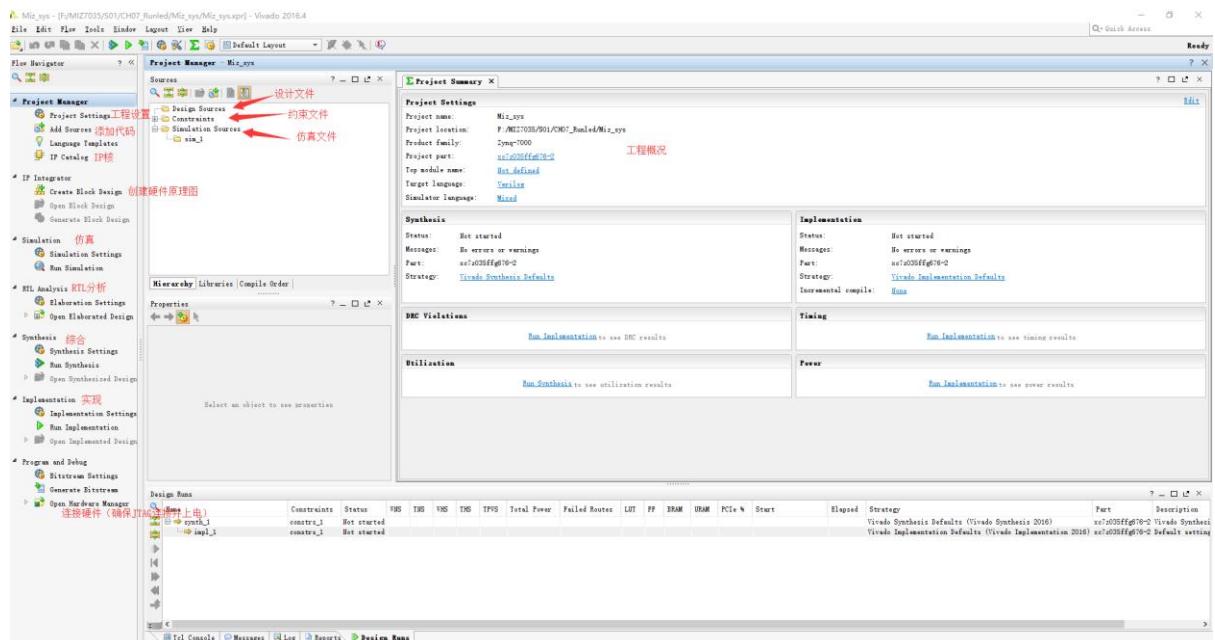




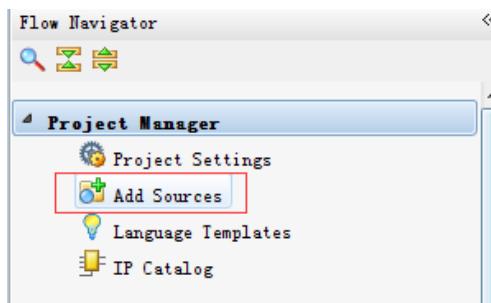
Step:6 单击 Finish 完成工程创建。

7.4 创建工程文件

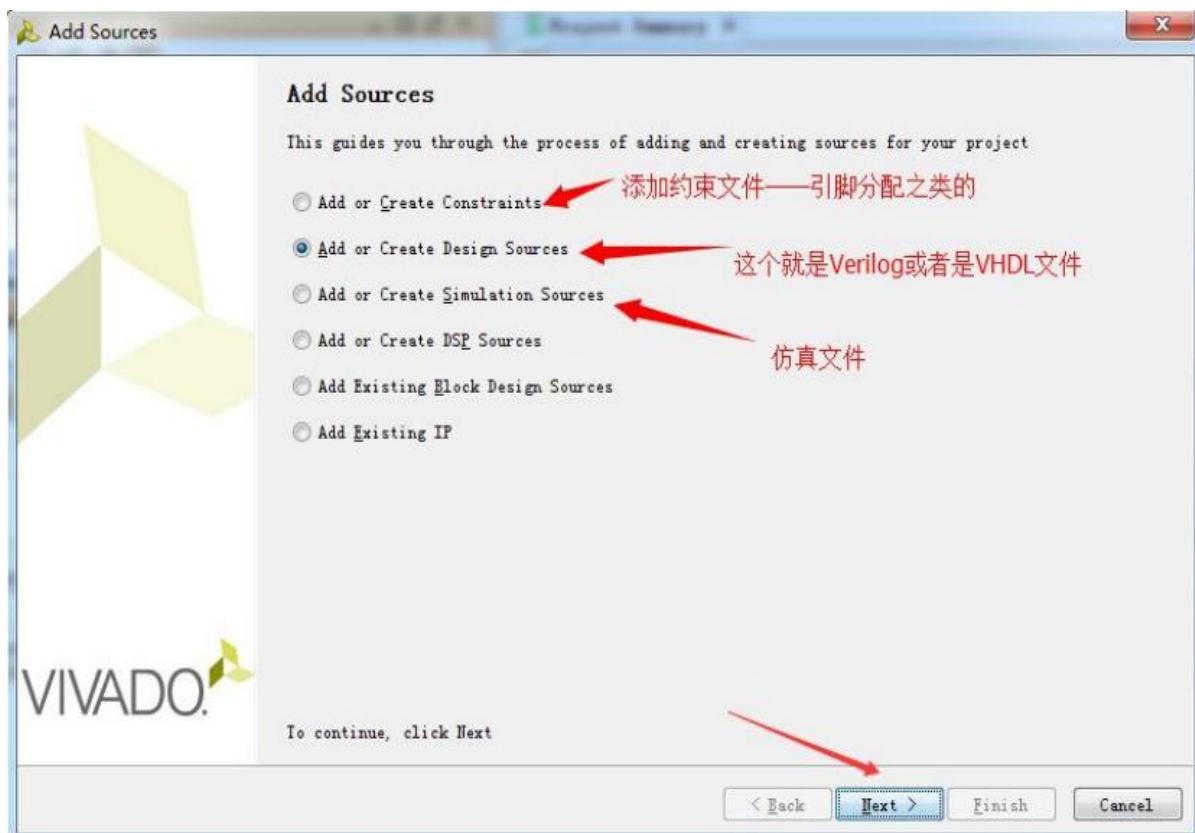
Step1: 打开 VIVADO 软件



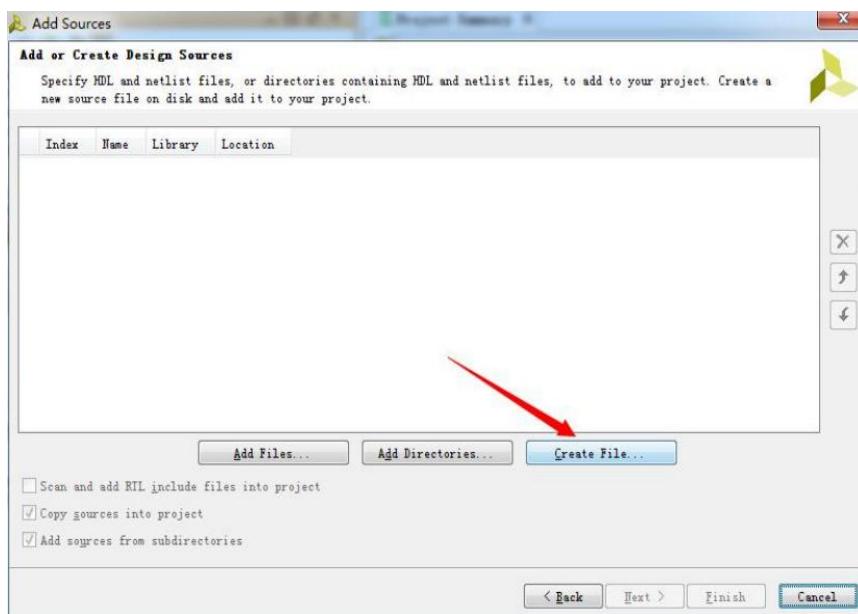
Step2: 单击 Add Sources



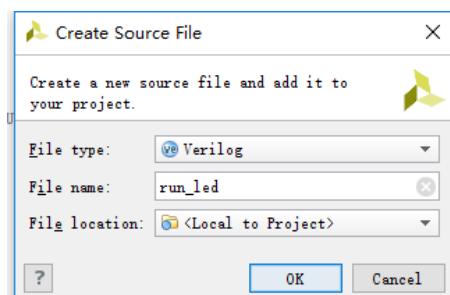
Step4: 选择单击 Add or Create Design Sources 然后单击 NEXT



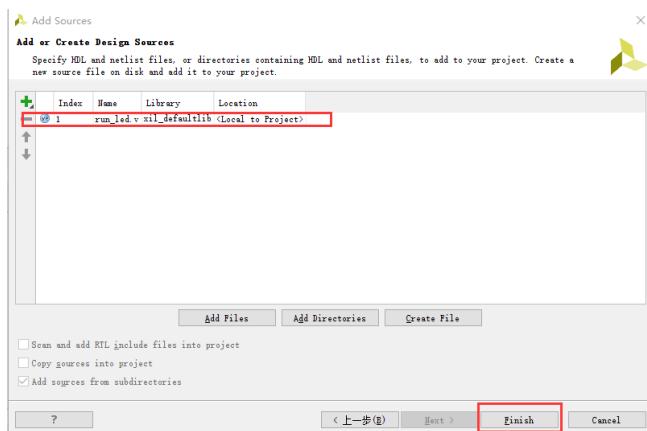
Step5: 单击 Create File 来创建文件



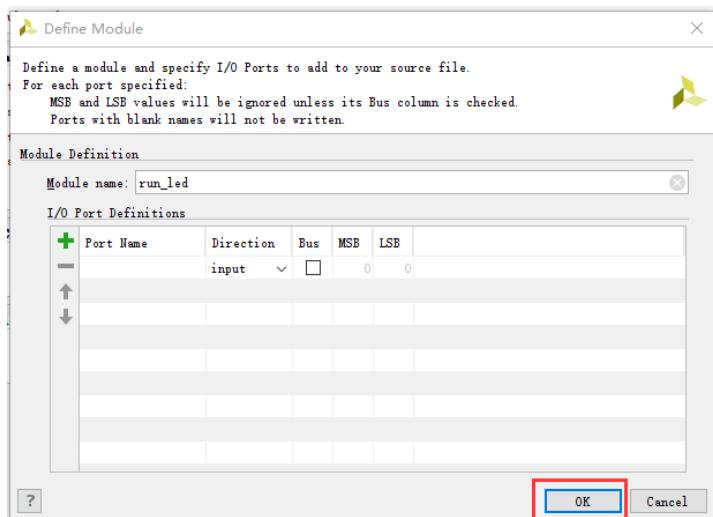
Step6: 创建一个 run_led 的文件，并且文件类型选择 Verilog



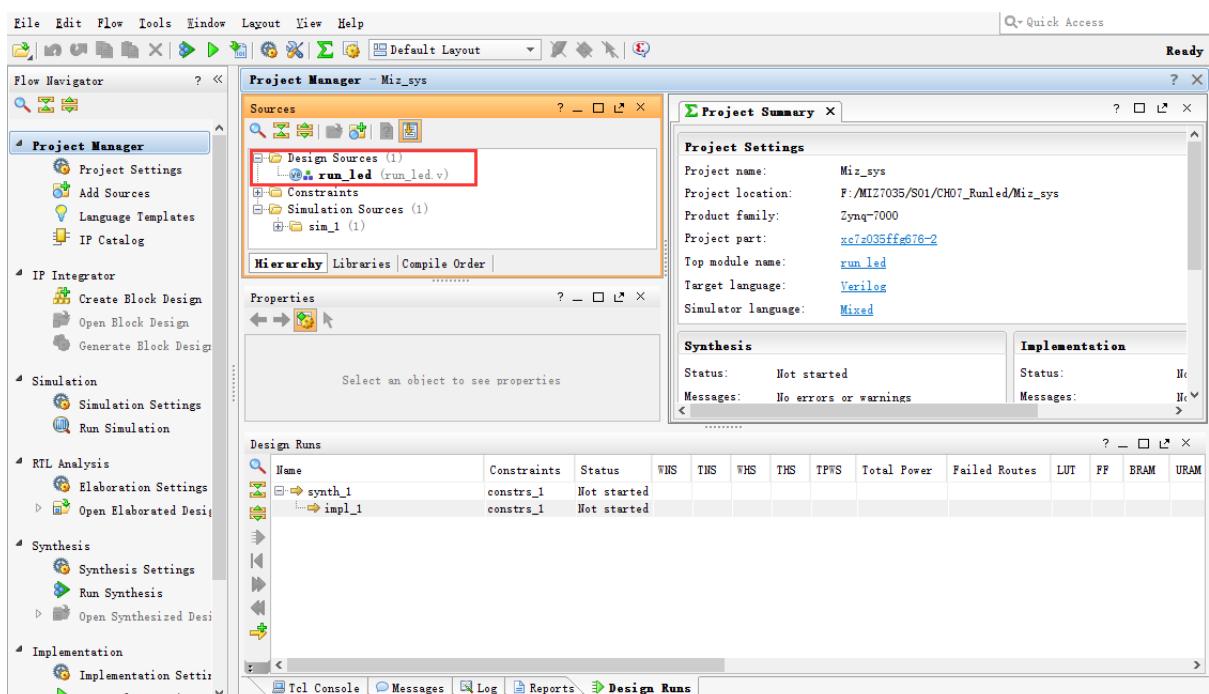
Step7: 添加完成后如下图所示之后单击 finish 完成文件的创建



Step8: 继续弹出的对话空中，可以设置一些端口，但是我们现在什么都不做。单击 OK



Step9: 创建完成后可以看到 Design Sources 文件夹中有了 run_led.v 这个文件，这个文件就是我们可以编写 verilog 程序的文件。



7.5 Verilog FPGA 流水灯实验

Step1: 双击 Led.v 打开流水程序源码如下

```
`timescale 1ns / 1ps
///////////////////////////////
// Company:
// Engineer:
```

```
//  
// Create Date: 2018/04/15 16:35:41  
// Design Name:  
// Module Name: run_led  
// Project Name:  
// Target Devices:  
// Tool Versions:  
// Description:  
//  
// Dependencies:  
//  
// Revision:  
// Revision 0.01 - File Created  
// Additional Comments:  
//  
///////////////////////////////////////////////////////////////////  
module run_led(  
    );  
endmodule
```

可以看出这是一个空的工程，我们现在要添加代码同时也要添加工程信息。

Step2: 编写程序并且添加工程信息

```
// Target Devices: XC7Z035-FFG676-2  
// Tool versions: VIVADO2016.4  
// Description: water led  
// Revision: V1.1  
// Additional Comments:  
//1) _i PIN input  
//2) _o PIN output  
//3) _n PIN active low  
//4) _dg debug signal  
//5) _r reg delay  
//6) _s state machine  
///////////////////////////////////////////////////////////////////  
module run_led(  
    input CLK_i,
```

```

input RSTn_i,
output reg [3:0]LED_o
);
reg [31:0]C0;
always @(posedge CLK_i)
if(!RSTn_i)
begin
    LED_o <= 4'b1;
    C0 <= 32'h0;
end
else
begin
    if(C0 == 32'd50_000_000)
        begin
            C0 <= 32'h0;
            if(LED_o == 4'b1000)
                LED_o <= 4'b1;
            else LED_o <= LED_o << 1;
        end
    else
        begin
            C0 <= C0 + 1'b1;
            LED_o <= LED_o;
        end
end
endmodule

```

这样我们就编写好了代码下面还要添加管脚约束文件。

7.6 添加管脚约束文件

管脚约束文件，即.XDC文件，一般情况，生成后会放在Miz_sys.srcs\constrs_1文件夹中。添加管脚约束有两种方法，一种是直接加入已经写好的约束文件；另一种是综合后，添加管脚约束。

第一种：直接加入已经写好的约束文件。

Step1：将我们提供历程中的“DOC”文件夹复制到你所建立的工程中，DOC文件夹包含已经写好的.XDC文件（如果有错误，以原理图为准）。本例中的工程是CH07_Runled，读者需要将历程中的DOC文件复制到自己建立的工程中。



例程中的DOC文件夹复制到你自己的工程中

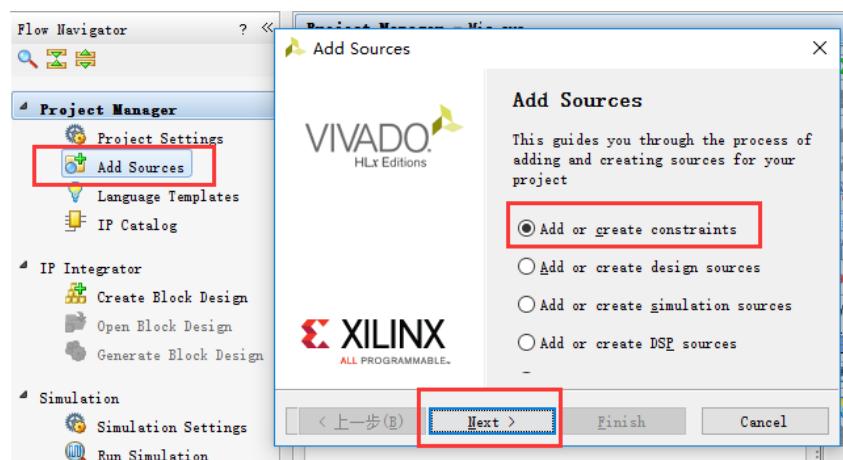
打开 DOC 文件夹，Miz_sys_pin.xdc 为本历程中的约束文件。



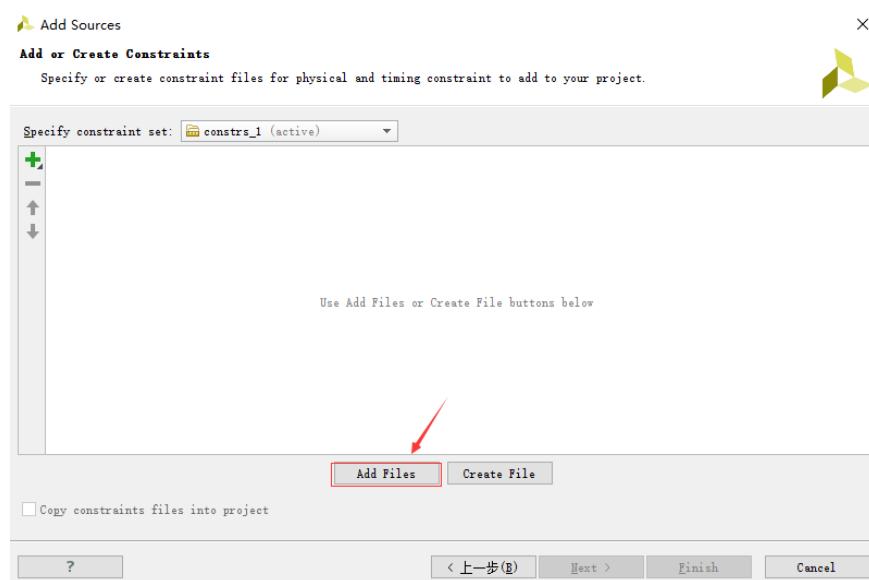
DOC 文件夹中的. XDC 文件

Step2: 单击 (和添加. v 文件一样)

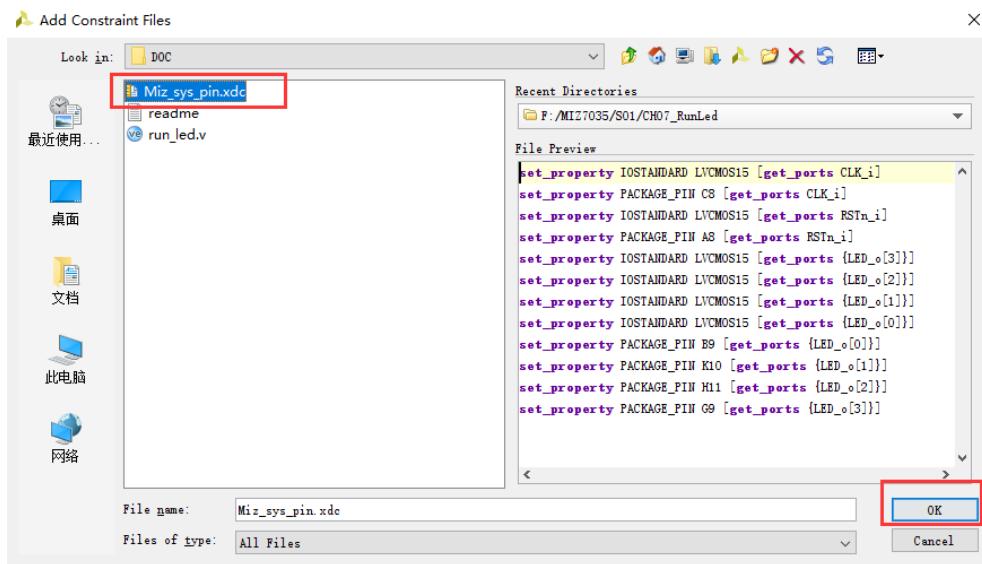
Step3: 选择 Add or create constraints 然后单击 NEXT



Step4: 单击 Add Files



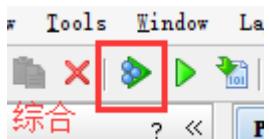
Step5: 选中 Miz_sys_pin.xdc 文件，然后点击 OK。



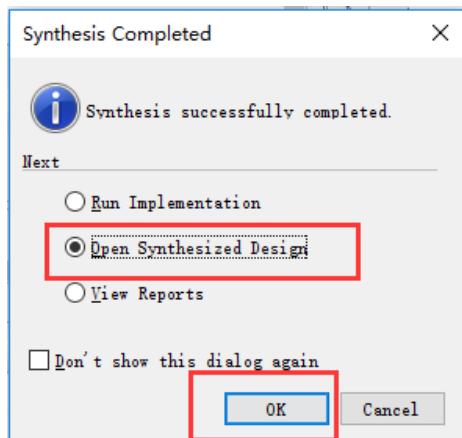
Step6:点击 Finish 完成约束文件的添加

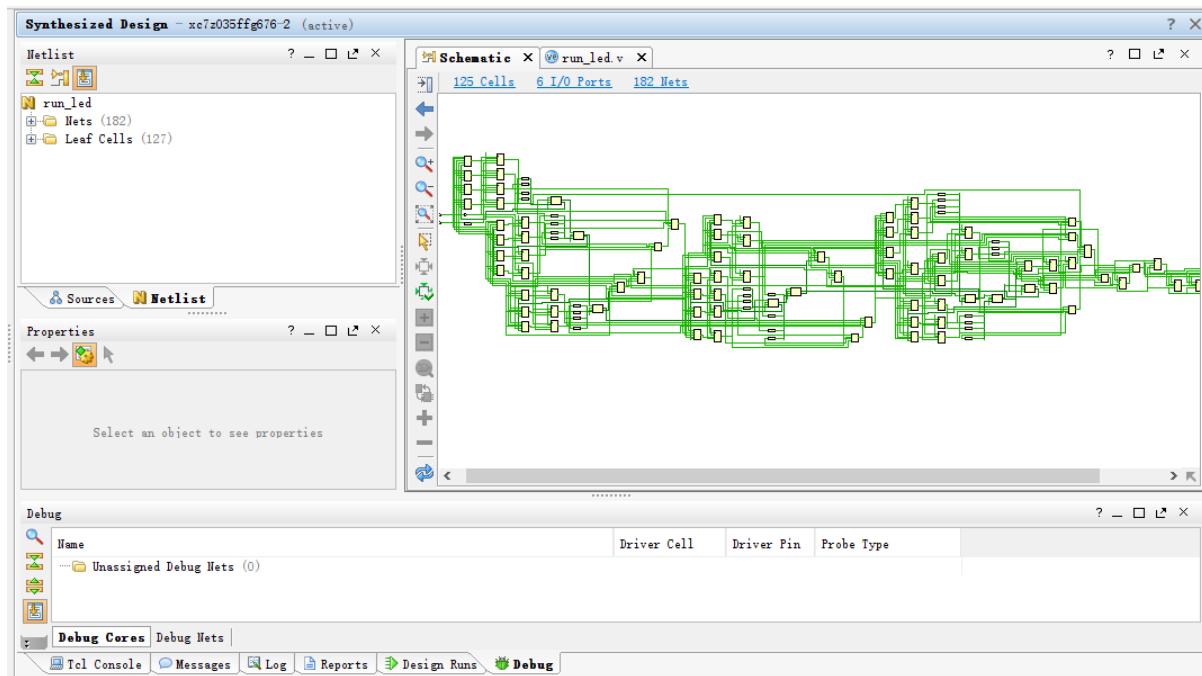
第二种：综合后，添加管脚约束。

Step1:综合工程。



Step2:打开 Open Synthesized Design。





Step3:管脚配置。

点击 I/O Ports，在 Find Results 中进行管脚配置（图 a）。Package Pin 对应 FPGA 中的管脚，I/O Std 对应电平标准（图 b）。例如要配置 RSTn_i 信号连接到 A8 管脚，电平标准是 LVCMOS15，在 Package Pin 中选择 A8 管脚，I/O Std 选择 LVCMOS15。其他管脚也做相应配置（图 c）。

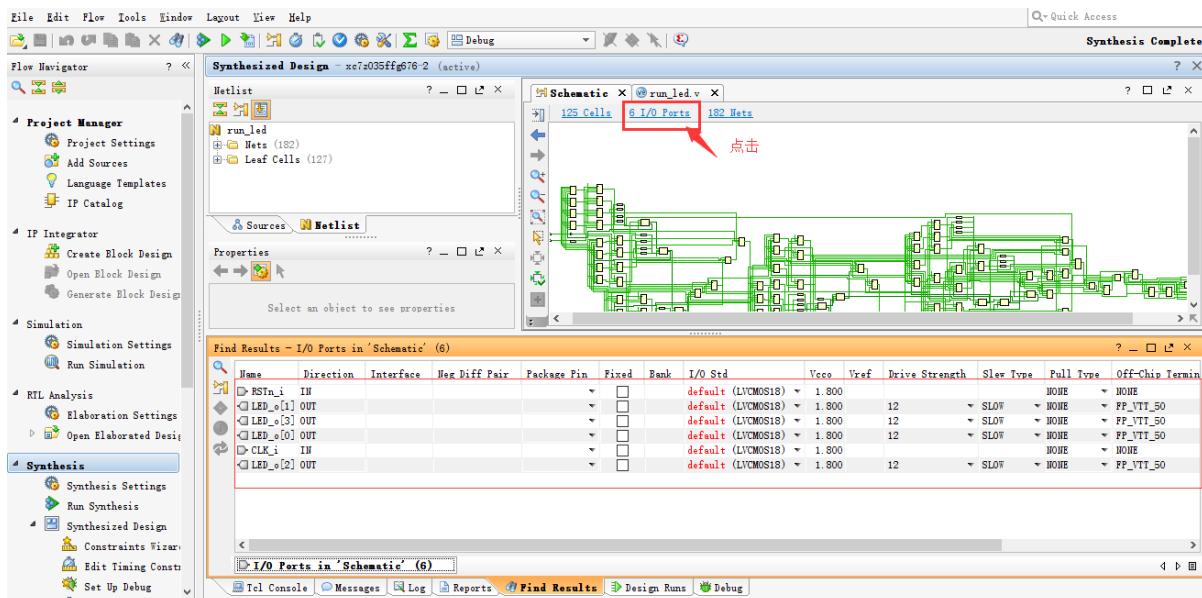


图 a

Find Results - I/O Ports in 'Schematic' (6)

Name	Direction	Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termin
RSTn_i	IN			I			LVCMS15*	1.500				HIGH	HIGH
LED_o[1]	OUT			A2			default (LVCMS18)	1.800	12	SLOW	HIGH		FP_VTT_50
LED_o[3]	OUT			A3			default (LVCMS18)	1.800	12	SLOW	HIGH		FP_VTT_50
LED_o[0]	OUT			A4			default (LVCMS18)	1.800	12	SLOW	HIGH		FP_VTT_50
CLK_i	IN			A5			default (LVCMS18)	1.800	12	SLOW	HIGH		FP_VTT_50
LED_o[2]	OUT			A7			default (LVCMS18)	1.800	12	SLOW	HIGH		FP_VTT_50
				A8									
				A9									
				A10									

管脚

电平标准

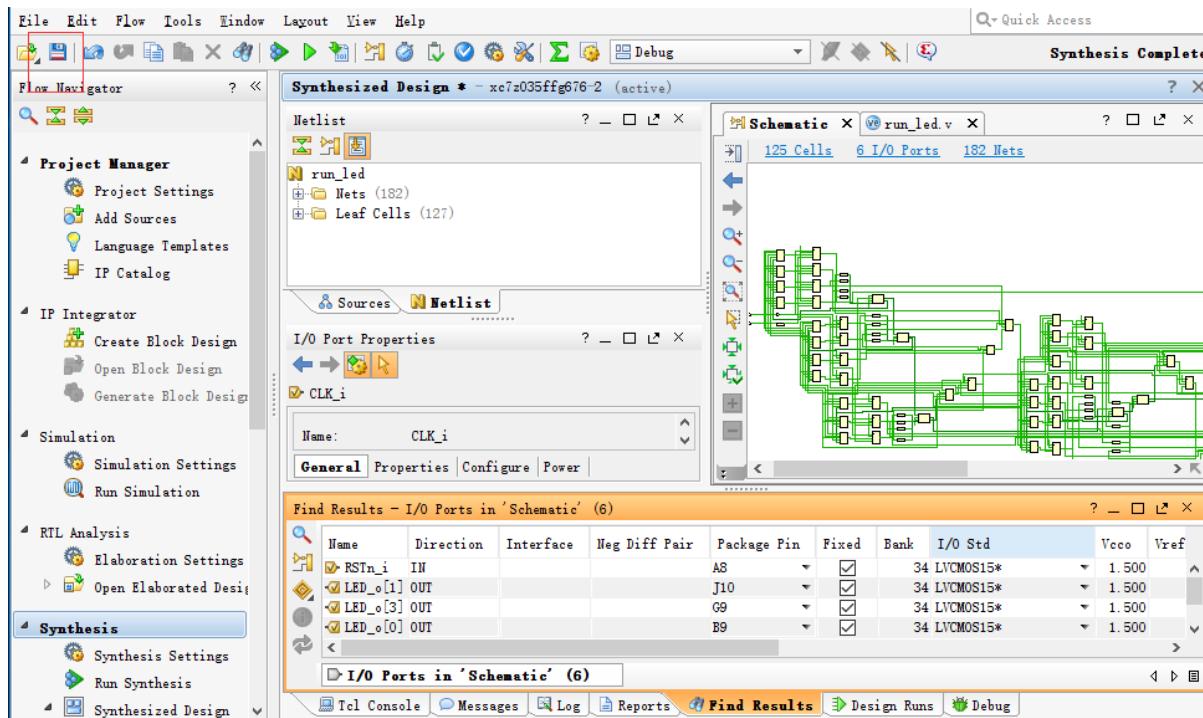
图 b

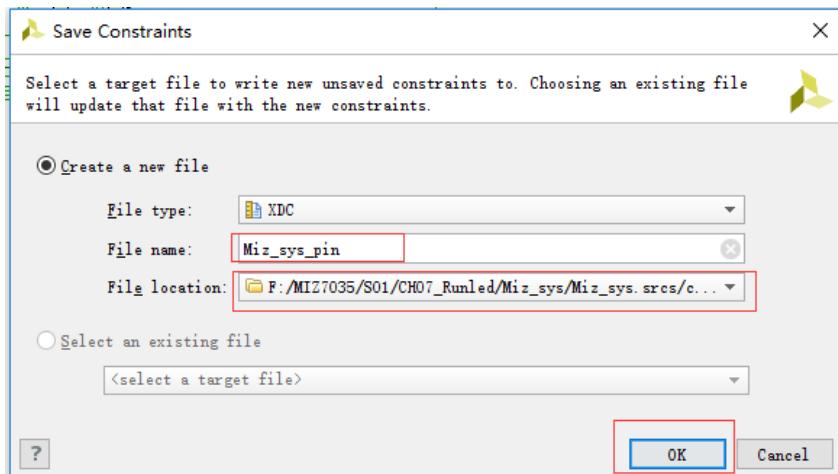
Find Results - I/O Ports in 'Schematic' (6)

Name	Direction	Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termin
RSTn_i	IN			A8	✓		34 LVCMS15*	1.500				HIGH	HIGH
LED_o[1]	OUT			J10	✓		34 LVCMS15*	1.500	12	SLOW	HIGH		FP_VTT_50
LED_o[3]	OUT			G9	✓		34 LVCMS15*	1.500	12	SLOW	HIGH		FP_VTT_50
LED_o[0]	OUT			B9	✓		34 LVCMS15*	1.500	12	SLOW	HIGH		FP_VTT_50
CLK_i	IN			J8	✓		34 LVCMS15*	1.500				HIGH	HIGH
LED_o[2]	OUT			H11	✓		34 LVCMS15*	1.500	12	SLOW	HIGH		FP_VTT_50

图 c

Step4: 保存，给 XDC 文件命名，生成.XDC 文件，并保存到 Miz_sys\MIz_sys.srcs\constrs。Constrs 文件夹是自己创建的。





Step5.XDC 文件生成结束

7.7 编译并且产生 bit 文件

Step1:单击综合

Step2:单击执行

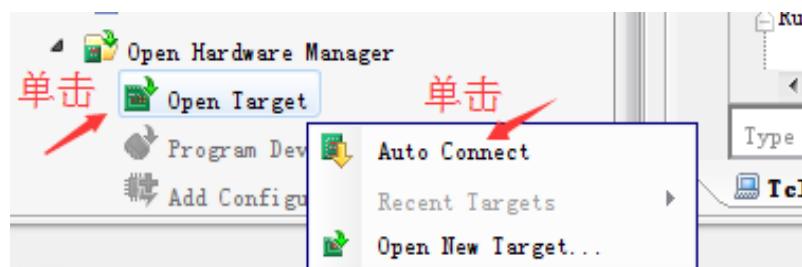
Step3:单击产生 bit



7.8 下载程序

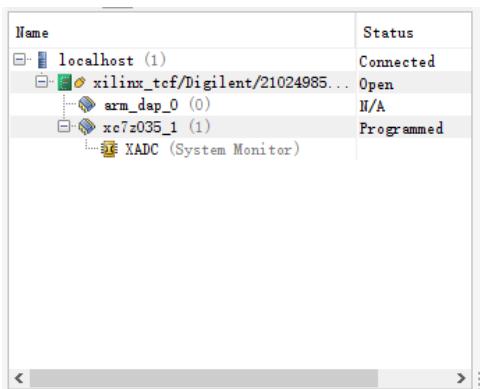
Step1:给开发板通电，并且连接下载器

Step2:单击 OpenTarget 然后单击 Auto Connect

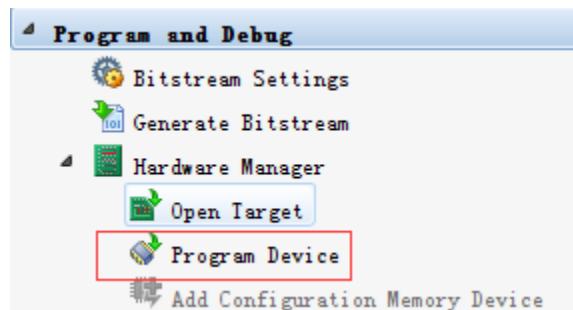


Step3:连接成功后如下图所示:

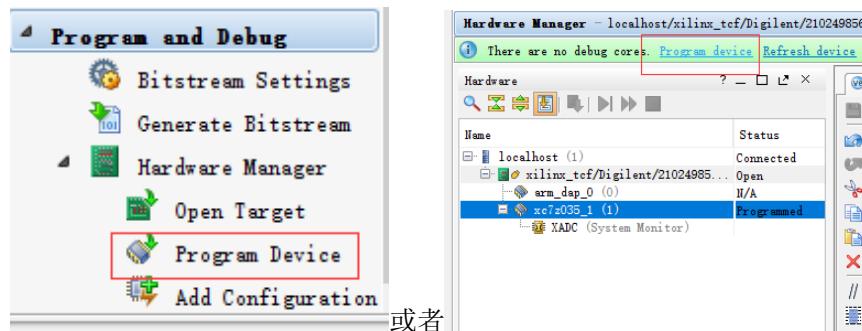
MIZ7035 如下图所示:



Step4:单击 Program Device

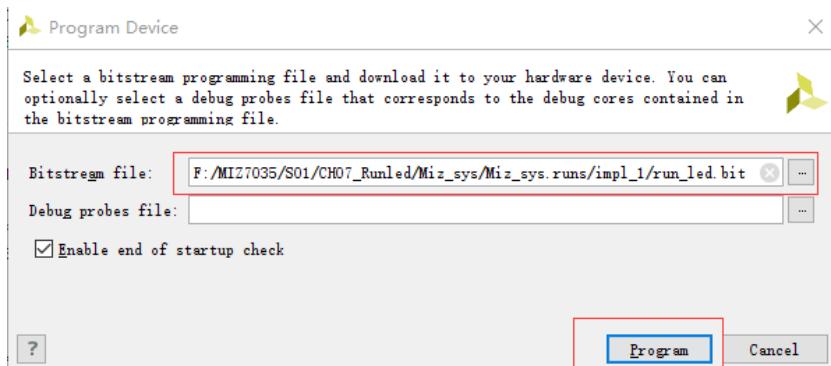


Step5:单击 Program Device 然后选择 XC7Z035。也可以从顶部单击 Program device

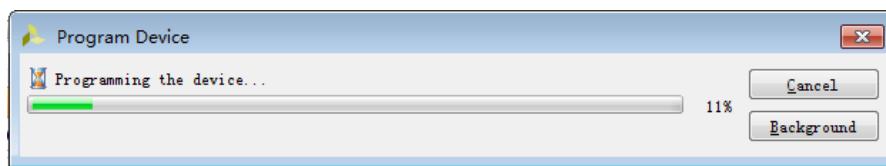


或者

Step6:弹出的对话框中有我们要下载的 Bit 文件

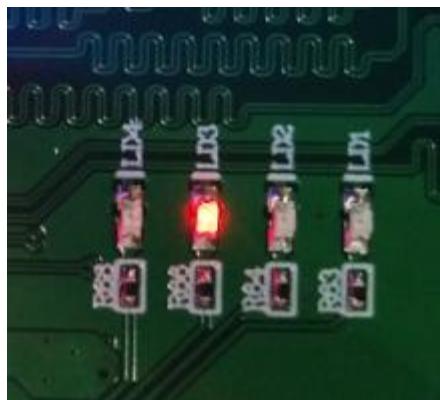


Step7: 下载过程



7.9 实验结果

下载过程下载完成后 LED 流水灯就运行起来了。



7.10 本章小结

本章详细讲解了如何创建 VIVADO 工程以及在 VIVADO 工程环境下编写纯 PL 代码的程序，并且讲解了如何添加管脚约束，时钟约束，编译程序，下载程序。通过流水灯实现这个简单的实验抛砖引玉，让大家掌握了 VIVADO 软件的使用。

CH08_FPGA_Button 按钮去抖动实验

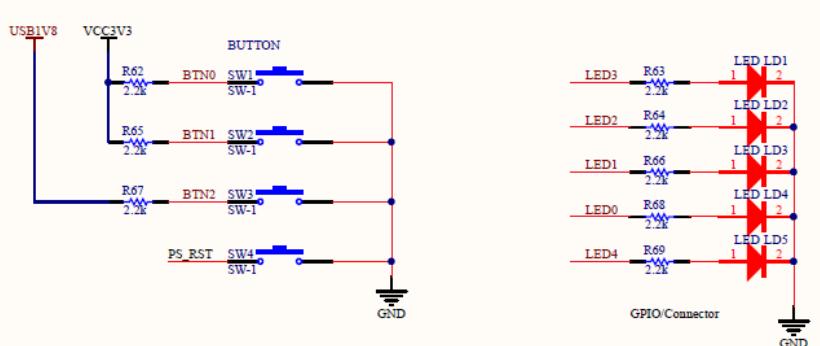
按键的消抖，是指按键在闭合或松开的瞬间伴随着一连串的抖动，这样的抖动将直接影响设计系统的稳定性，降低响应灵敏度。因此，必须对抖动进行处理，即消除抖动的影响。实际工程中，有很多消抖方案，如 RS 触发器消抖，电容充放电消抖，软件消抖。本章利用 FPGA 内部来设计消抖，即采取软件消抖。

按键的机械特性，决定着按键的抖动时间，一般抖动时间在 5ms~10ms。消抖，也意味着，每次在按键闭合或松开期间，跳过这段抖动时间，再检测按键的状态。只要通过简单的延时就可实现按键的消抖动。

8.1 硬件介绍

MiZ7035 底板中配套 4 个独立按键与 FPGA 相连，其中两个是可分配的用户按键（BTN1 和 BTN2），具体请参见 MiZ7035_Fun 原理图。由于各个按键独立，消抖过程是一样的，故本节就用板子上的一个按键 BTN0 来模拟实际环境。按键每按一次，对应的 LED 灯反转一次。即检测按键是否有闭合和断开的过程，如果有，第一次则 LED 灯点亮，第二次，则 LED 灯熄灭。

原理图如下图所示



MiZ7035 开发板底板用户按键和 4 个 LED 与 FPGA 对应的 PIN 定义：以原理图为标准

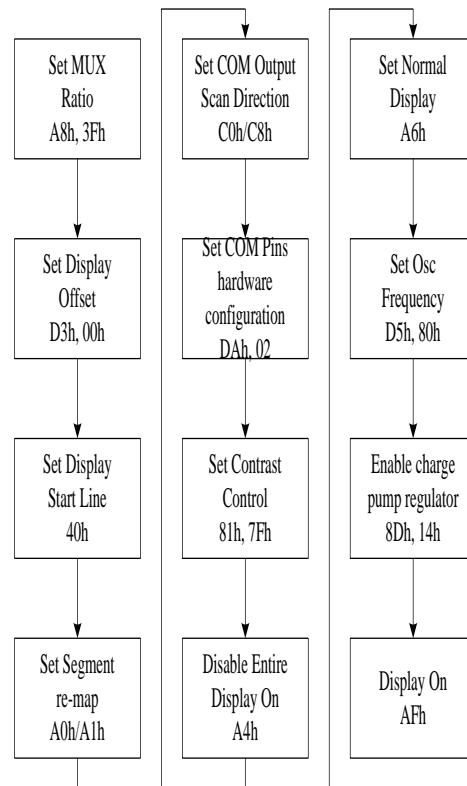
clk_i: H16 LED0: B9 LED1: J10 LED2: H11 LED3: G9	BTN0(key_i): A10 BTN1(rst_n_i): A8
--	---------------------------------------

对应工程：S01/ CH08_FPGA_Button_New

8.2 时序设计

由于按键固有的特性，在每次闭合和断开时，经过抖动-稳定-抖动-稳定的过程。因此，检测按键是否有按下过程，则要进行两次消抖处理。通过检测按键输入的值，当检测到 BTNC 为低电平时，启动计数器，做 10ms 延时，再检测一次，若 BTNC 依然为低，则说明，BTNC 被按下，设置

按键按下标志位。再次检测 BTNC，若 BTNC 为高电平，做 10ms 延时，第二次检测，若依然为高电平，则说明 BTNC 已断开，设置 BTNC 断开标志位，通过两个标志位，可以判断，BTNC 已经完成了一次闭合到断开的过程，则 led 灯反转一次。消抖基本流程如下图所示。



采用状态机来实现上面的流程是非常方便的。通过状态机的切换，按键每次由闭合到断开的过程中，分别产生 low_flag 和 high_flag，当这两个同时为高电平时，led 灯实现一次翻转。即第一次按键 led 高亮，第二次按键 led 熄灭，第三次 led 高亮……。

8.3 程序源码

```

`timescale 1ns / 1ps
//-----
/*
* 文件名字: Key_Jitter.v
* 程序描述:
* 作      者:
* 修改日期:
* 版本号:
* 版权所有: 漂阳米联电子科技有限公司
*/
//-----

```

```
module Key_Jitter(
    input clk_i,
    input rst_n_i,
    input key_i,
    output [3:0] led_o,
    output [18:0] div_cnt_tb,
    output [2:0] key_state_tb
);

localparam DELAY_Param=19'd499_999;//for project
//localparam DELAY_Param=19'd1500;      //for simulation
//localparam DELAY_Param=19'd10;       //No filter jitter

reg [3:0] led_o_r;
(*KEEP = "TRUE" *)reg [18:0] div_cnt;//10ms 去抖时间计数器
always@(posedge clk_i or negedge rst_n_i)
begin
    if(!rst_n_i)
        div_cnt<=19'd0;
    else if(div_cnt<DELAY_Param)
        div_cnt<=div_cnt+1'b1;
    else
        div_cnt<=0;
end

wire delay_10ms=(div_cnt==DELAY_Param) ? 1'b1:1'b0;

//按键检测状态机
localparam DETECTER1=3'b000;
localparam DETECTER2=3'b001;
localparam DETECTER3=3'b010;
localparam DETECTER4=3'b011;
localparam LED_DIS  =3'b100;
```

```
reg low_flag;
reg high_flag;
reg [2:0] key_state;
always@(posedge clk_i or negedge rst_n_i)
begin
    if(!rst_n_i)
        begin
            key_state<=DETECTER1;
            low_flag<=0;
            high_flag<=0;
            led_o_r<=4'b1111;
        end
    else if(delay_10ms) //每 10ms 检测一次，每次检测按键闭合和断开过程,
        led 灯翻转一次
        begin
            case(key_state)
                DETECTER1 :
                    begin
                        if(key_i!=1'b1)
                            key_state<=DETECTER2;
                        else
                            key_state<=DETECTER1;
                    end
                DETECTER2 :
                    begin
                        if(key_i!=1'b1)
                            begin
                                low_flag<=1'b1;
                                key_state<=DETECTER3;
                            end
                        else
                            begin
                                key_state<=DETECTER1;
                                low_flag<=low_flag;
                            end
                    end
            endcase
        end
    end
end
```

```
end
```

```
DETECTER3 :
```

```
begin
```

```
    if(key_i==1'b1)
```

```
        key_state<=DETECTER4;
```

```
    else
```

```
        key_state<=DETECTER3;
```

```
end
```

```
DETECTER4 :
```

```
begin
```

```
    if(key_i==1'b1)
```

```
        begin
```

```
            high_flag<=1'b1;
```

```
            key_state<=LED_DIS;
```

```
        end
```

```
    else
```

```
        begin
```

```
            high_flag<=high_flag;
```

```
            key_state<=DETECTER3;
```

```
        end
```

```
    end
```

```
LED_DIS :
```

```
begin
```

```
    if(high_flag & low_flag)
```

```
        begin
```

```
            key_state<=DETECTER1;
```

```
            led_o_r<=~led_o_r;
```

```
            high_flag<=1'b0;
```

```
            low_flag<=1'b0;
```

```
        end
```

```
    else
```

```
        begin
```

```
            led_o_r<=led_o_r;
```

```
key_state<=key_state;
high_flag<=high_flag;
low_flag<=low_flag;
end
end
default:
begin
    key_state<=DETECTER1;
    led_o_r<=0;
    high_flag<=0;
    low_flag<=0;
end
endcase
end
else
begin
    key_state<=key_state;
    led_o_r<=led_o_r;
    high_flag<=high_flag;
    low_flag<=low_flag;
end
end
assign led_o=led_o_r;
assign div_cnt_tb=div_cnt;
assign key_state_tb=key_state;
endmodule
```

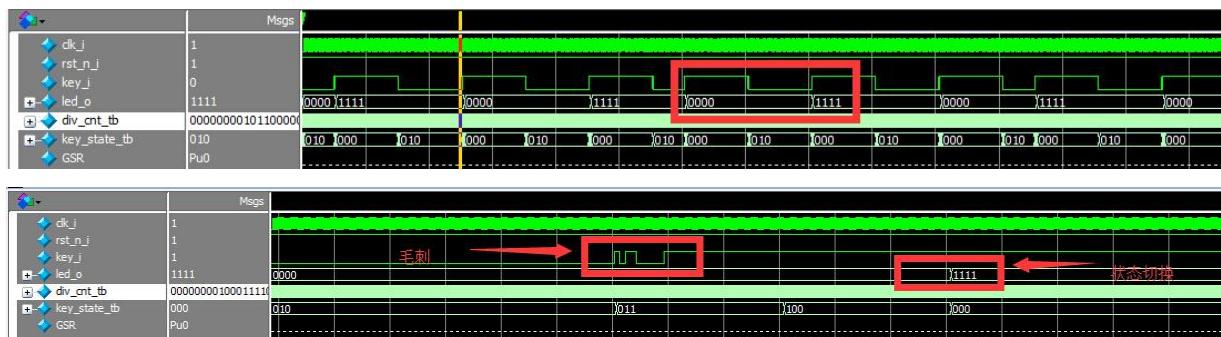
8.4 程序分析

- 程序中定义了 div_cnt 计数器，实现 10ms 消抖延时操作；
- 程序中定义了 key_state 一段式状态机，该状态机包括 5 个状态。其中前 4 个为按键闭合与断开检测与消抖状态，在完成前 4 个状态后，通过 led 灯的翻转来表现按键的闭合与断开；
- 程序中定义了两个标志位，分别为 low_flag 和 high_flag。low_flag 表示检测到按键按下，即按键闭合，high_flag 表示检测到按键弹起，即按键断开。当着两个标志为同时为 1 时，表示按键完成一次闭合与断开；
- 程序中，对状态机做了循环，来回不停检测按键状态，且每次状态的切换时间是 10ms，状态

的切换时间，也是按键消抖检测过程。

8.5 综合布线前仿真时序

为清晰地表达程序工作流程，在源代码中，添加了 `div_cnt_tb`,`div_start_tb`,`key_state_tb` 这些信号。且程序中将有意将计数器另外设置一个较小的值，这是为了减少仿真等待的时间。其仿真图如下所示。



8.6 Chipscope 在线逻辑分析仪仿真

对于消抖试验而言，逻辑分析仪不能很直观表现整个设计流程，故该节将放弃使用，直接观察按键按下，LED 灯能否正确熄灭和点亮作为试验结果。

8.7 输出结果

按键 BTN0 每按一次，LED 灯很好地熄灭和点亮。LED 灯响应无差错。为清晰的表示消抖的效果，可将延时参数设置很小，可以发现，按键有时候明明已经按下去了，LED 却无响应。

8.8 小结

按键消抖，是轻触开关必须进行的一项操作。否则，按键将无法使用。在一个工作系统当中，由于按键未消抖，甚至可能直接使系统崩溃。从上面的几个例子也可以看出，无论是按键消抖，还是跑马灯等，设计的前提是要熟练掌握 verilog 语法。后续章节中，将不在局限于单个文件。在代码量逐渐增加的基础上，将分模块分文件来实现整个系统的功能。

CH09_FPGA 多路分配器设计

在第二章的学习中，笔者带大家通过一个入门必学的流水灯实验实现，快速掌握了VIVADO 基于 FPGA 开发板的基本流程。考虑到很多初学者并没有掌握好 Vivado 下 FPGA 的开发流程，本章开始笔者讲更加详细地介绍基于 VIVADO FPGA 开发的流程规范，让读者掌全面掌握 FPGA 开发流程包括了如何仿真、综合、执行、下载到开发板测试。

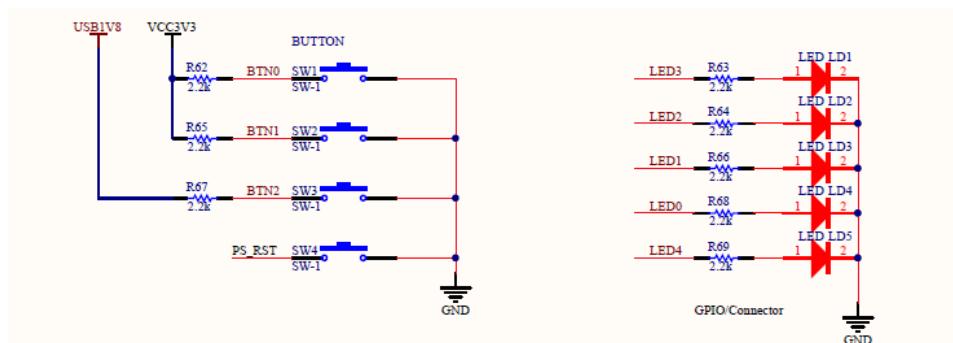
9.1 硬件图片

本章使用到的硬件和前一章一样：LED 部分及按钮部分

MiZ7035:



9.2 硬件原理图

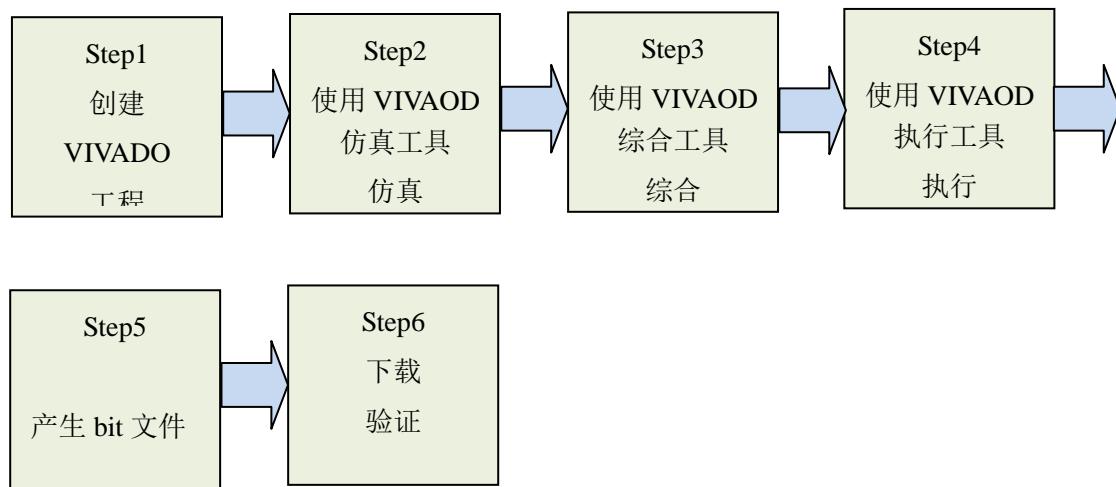


PIN 脚定义：以原理图为标准

clk_i: C8 div2_o (LED0): B9 div2hz_o (LED1): J10 div3_o (LED2): H11 div4_o (LED3): G9 div8_o:B10	BTN2:R19 BTN1(rst_n_i): A8
---	-------------------------------

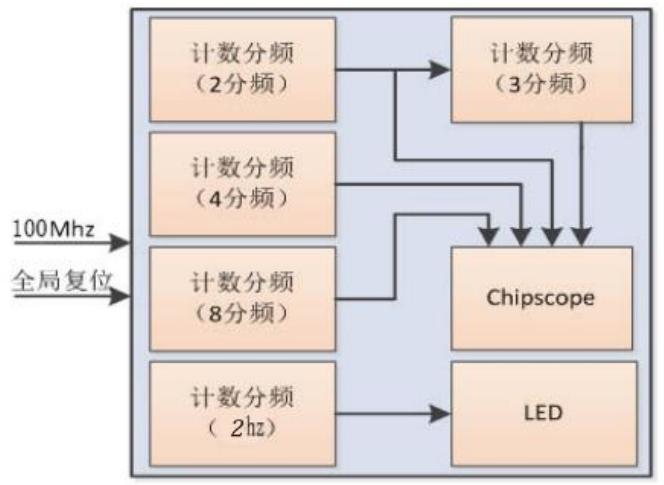
对应工程：S01/CH09_CLK_DIV_New

9.3 介于 VIVADO 的 FPGA 设计流程

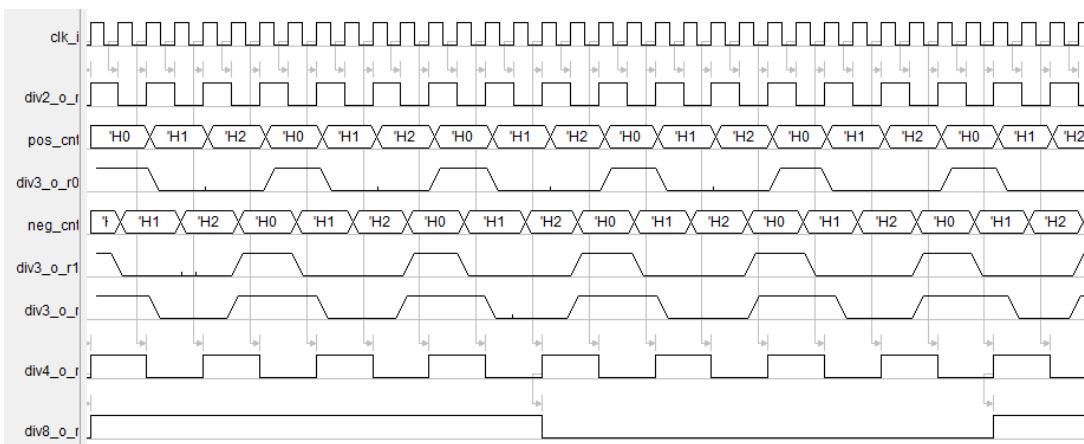


9.4 多路分配器设计思想

FPGA 输入全局时钟 100MHZ，定义合适的分频计数器，得到对应的时钟。通过 chipscope 来抓取 2 分频、3 分频、4 分频和 8 分频结果，通过板子上的 LED 灯，来显示 2HZ 的时钟。设计总体框图如下所示



9.5 时序设计



- ① 定义寄存器 `div2_o_r`, 检测输入时钟上升沿, 每次上升沿寄存器 `div2_o_r` 反转一次, 实现 2 分频。
- ② 定义寄存器 `pos_cnt[1:0]`, `neg[1:0]`, 分别检测 `div2_o_r` 的上升沿和下降沿, 检测到上升沿和下降沿时, 两个寄存器分别累加。计数到 2^d2 时, 寄存器清零。另定义两个 `div3_o_r0` 和 `div3_o_r1`, 当两个计数器小于 2^d1 时,`div3_o_r0` 和 `div3_o_r1` 均赋值为 1, 其他情况赋值为 0。由 `div3_o_r0` 和 `div3_o_r1` 组合逻辑相或即为 `div2_o_r` 进一步进行 3 分频所得的结果。
- ③ 定义位宽为 2 的寄存器 `div_cnt[1:0]`, 检测输入时钟上升沿, `div_cnt==2'b00` 或 `2'b01`, 4 分频输出寄存器 `div4_o_r` 反转, `div_cnt==2'b00` 时, 8 分频输出寄存器 `div8_o_r` 反转。
- ④ 由于输入时钟 100MHZ, 为得到 2HZ 的时钟, 需要定义计数器至少 $1000000000/1=100000000$ 。在此定义一个 26 位位宽的 `div2hz_cnt` 计数器。检测输入时钟上升沿, `div2hz_cnt==26'd24_999999` 或 `div2hz_cnt==26'd49_999999` 时, 2HZ 输出寄存器 `div2hz_o_r` 反转。

9.6 程序源码

```

`timescale 1ns / 1ps
//-----
// Target Devices: XC7Z020-FGG400
// Tool versions: VIVADO2016.4
// Description: Divider_Multiple
// Revision: V1.1
// Additional Comments:

```

```
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low
//4) _dg debug signal
//5) _r reg delay
//6) _s state machine
*/
//-----
module Divider_Multiple(
    input clk_i,
    input rst_n_i,
    output div2_o,
    output div3_o,
    output div4_o,
    output div8_o,
    output div2hz_o
);

reg div2_o_r;
always@(posedge clk_i or negedge rst_n_i)
begin
    if(!rst_n_i)
        div2_o_r<=1'b0;
    else
        div2_o_r<=~div2_o_r;
end

reg [1:0] div_cnt1;
always@(posedge clk_i or negedge rst_n_i)
begin
    if(!rst_n_i)
```

```
div_cnt1<=2'b00;  
else  
    div_cnt1<=div_cnt1+1'b1;  
end  
  
reg div4_o_r;  
reg div8_o_r;  
  
always@(posedge clk_i or negedge rst_n_i)  
begin  
    if(!rst_n_i)  
        div4_o_r<=1'b0;  
    else if(div_cnt1==2'b00 || div_cnt1==2'b10)  
        div4_o_r<=~div4_o_r;  
    else  
        div4_o_r<=div4_o_r;  
end  
  
always@(posedge clk_i or negedge rst_n_i)  
begin  
    if(!rst_n_i)  
        div8_o_r<=1'b0;  
    else if((~div_cnt1[0]) && (~div_cnt1[1]))  
        div8_o_r<=~div8_o_r;  
    else  
        div8_o_r<=div8_o_r;  
end  
  
reg [1:0] pos_cnt;  
reg [1:0] neg_cnt;  
always@(posedge div2_o_r or negedge rst_n_i)
```

```
begin
    if(!rst_n_i)
        pos_cnt<=2'b00;
    else if(pos_cnt==2'd2)
        pos_cnt<=2'b00;
    else
        pos_cnt<=pos_cnt+1'b1;
end

always@(negedge div2_o_r or negedge rst_n_i)
begin
    if(!rst_n_i)
        neg_cnt<=2'b00;
    else if(neg_cnt==2'd2)
        neg_cnt<=2'b00;
    else
        neg_cnt<=neg_cnt+1'b1;
end

reg div3_o_r0;
reg div3_o_r1;
always@(posedge div2_o_r or negedge rst_n_i)
begin
    if(!rst_n_i)
        div3_o_r0<=1'b0;
    else if(pos_cnt<2'd1)
        div3_o_r0<=1'b1;
    else
        div3_o_r0<=1'b0;
end
```

```
always@(negedge div2_o_r or negedge rst_n_i)
begin
    if(!rst_n_i)
        div3_o_r1<=1'b0;
    else if(neg_cnt<2'd1)
        div3_o_r1<=1'b1;
    else
        div3_o_r1<=1'b0;
end

reg div2hz_o_r;
reg [25:0] div2hz_cnt;

always@(posedge clk_i or negedge rst_n_i)
begin
    if(!rst_n_i)
        div2hz_cnt<=0;
    else if(div2hz_cnt<26'd50_000000)
        div2hz_cnt<=div2hz_cnt+1'b1;
    else
        div2hz_cnt<=0;
end

always@(posedge clk_i or negedge rst_n_i)
begin
    if(!rst_n_i)
        div2hz_o_r<=0;
    else if(div2hz_cnt==26'd24_999999 || div2hz_cnt==26'd49_999999)
        div2hz_o_r<=~div2hz_o_r;
    else
        div2hz_o_r<=div2hz_o_r;
end
```

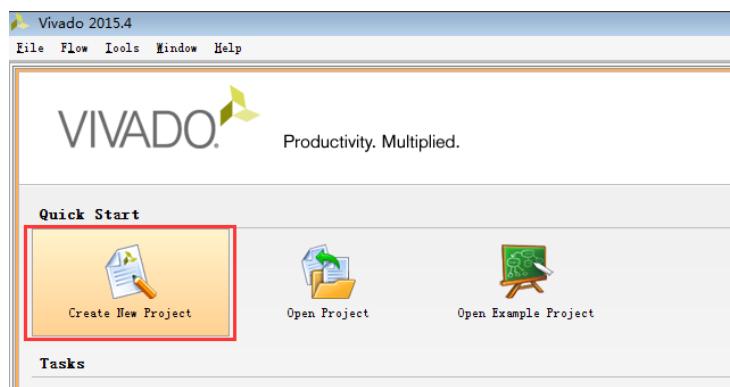
```
assign div2_o=div2_o_r;
assign div3_o=div3_o_r0 | div3_o_r1;
assign div4_o=div4_o_r;
assign div8_o=div8_o_r;
assign div2hz_o=div2hz_o_r;

ila_0 ila_0_0 (
    .clk(clk_i), // input wire clk
    .probe0(div2hz_o), // input wire [0:0] probe0
    .probe1({div2_o,div3_o,div4_o,div8_o}) // input wire [3:0] probe1
);
endmodule
```

9.7 行为仿真

9.7.1 创建多路分频器工程

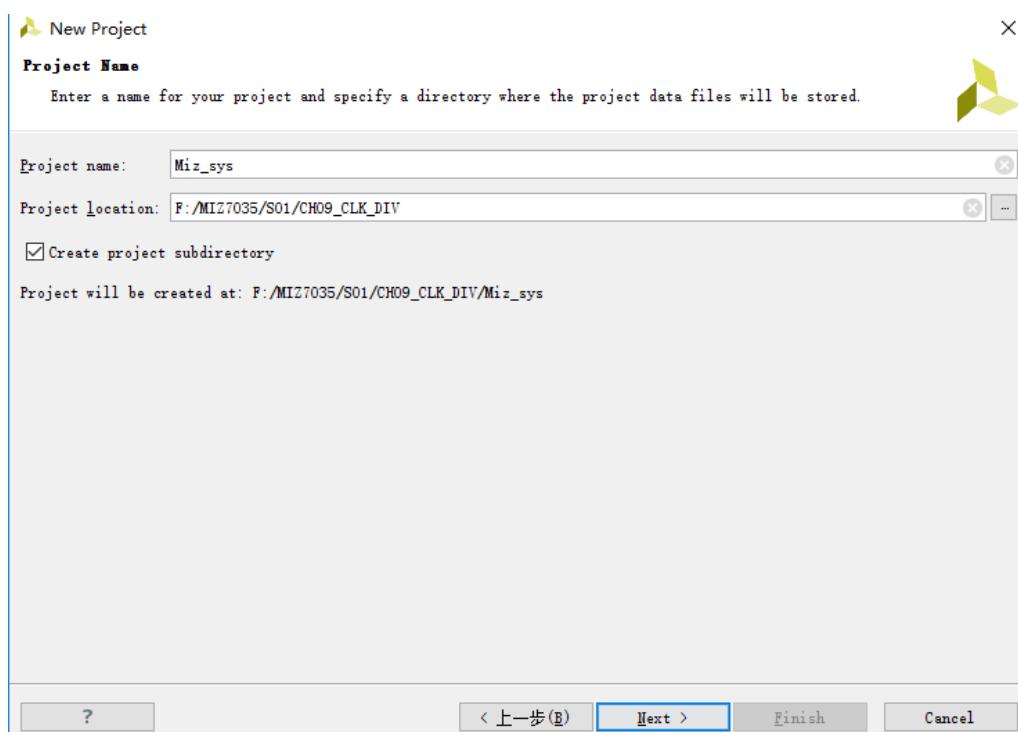
Step1: 创建工程



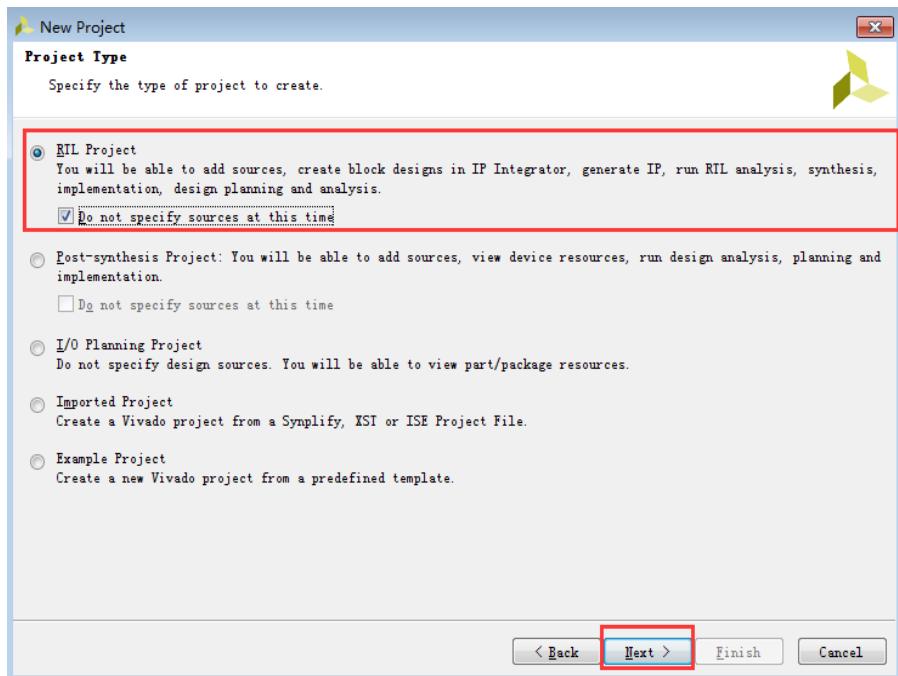
Step2: 欢迎界面直接单击 NEXT



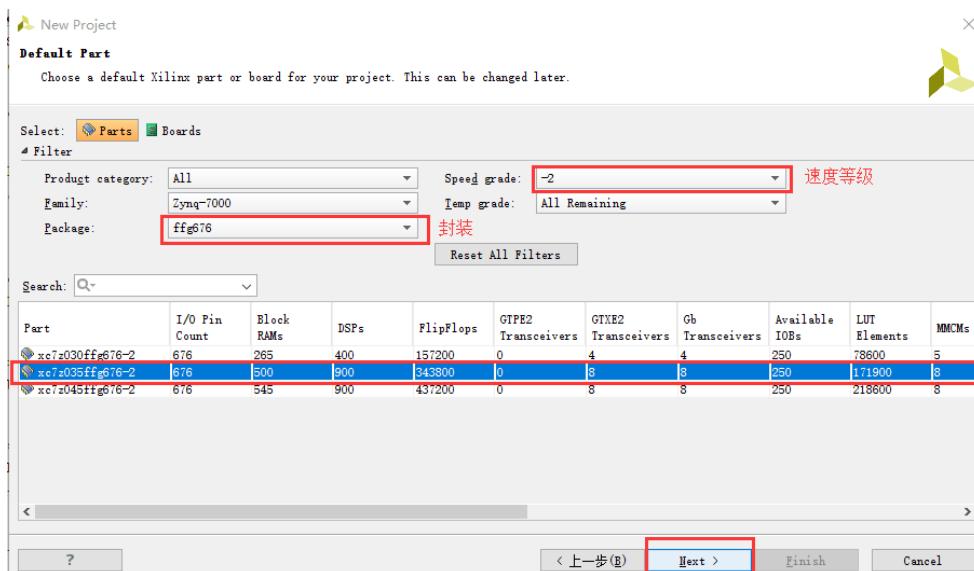
Step3:工程名字命名为 Divider_Multiple，并且设置保存的路径，单击 NEXT



Step4:新建一个 RTL 工程，并且勾选不要添加源文件，单击 NEXT



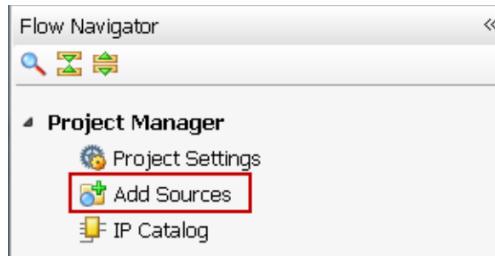
Step5: 选择芯片的型号和封装速度等级



Step6:最后单击 Finish 完成工程的创建

9.7.2 添加仿真文件

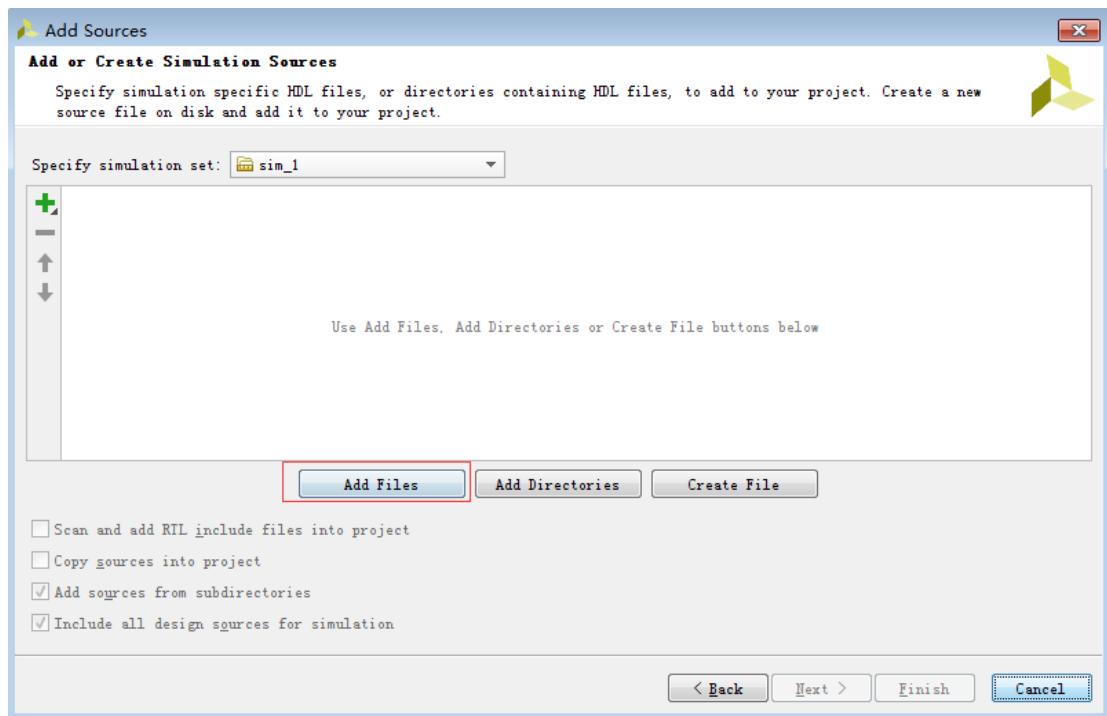
Step1:单击 Add Sources 添加仿真文件



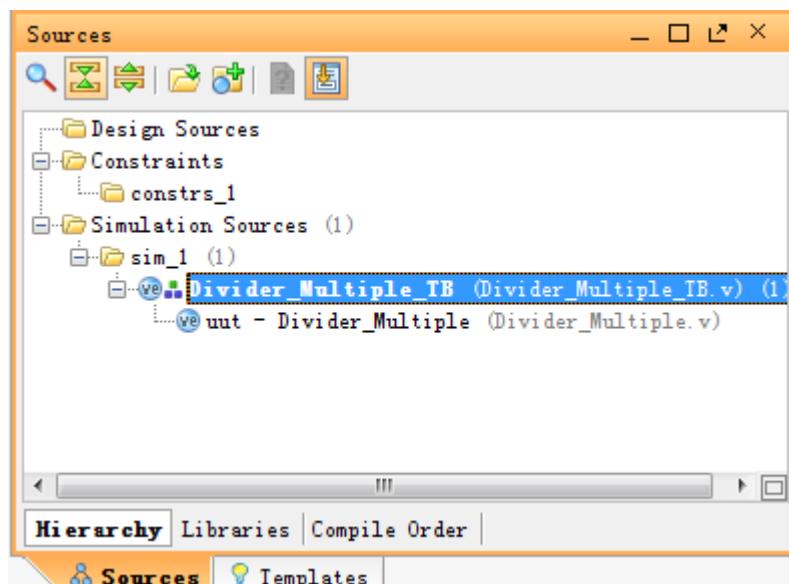
Step2:单击 Add Sources 添加仿真文件



Step3:单击 Add files 把仿真文件添加进来



Step4: 单击 Add files 把仿真文件添加进来



Step5: 仿真文件源码

```
module Divider_Multiple_top_TB;

// Inputs
reg clk_i;
reg rst_n_i;
```

```
// Outputs
wire div2_o;
wire div3_o;
wire div4_o;
wire div8_o;
wire div2hz_o;

// Instantiate the Unit Under Test (UUT)
Divider_Multiple_top uut (
    .clk_i(clk_i),
    .rst_n_i(rst_n_i),
    .div2_o(div2_o),
    .div3_o(div3_o),
    .div4_o(div4_o),
    .div8_o(div8_o),
    .div2hz_o(div2hz_o)
);

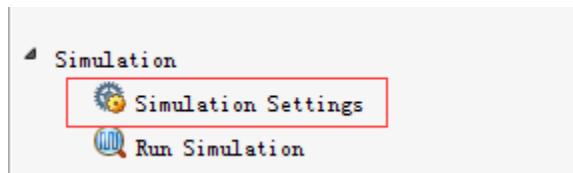
initial
begin
    // Initialize Inputs4
    clk_i = 0;
    rst_n_i = 0;

    // Wait 100 ns for global reset to finish
    #96;
    rst_n_i=1;
end

always
begin
    #5 clk_i=~clk_i;
end

endmodule
```

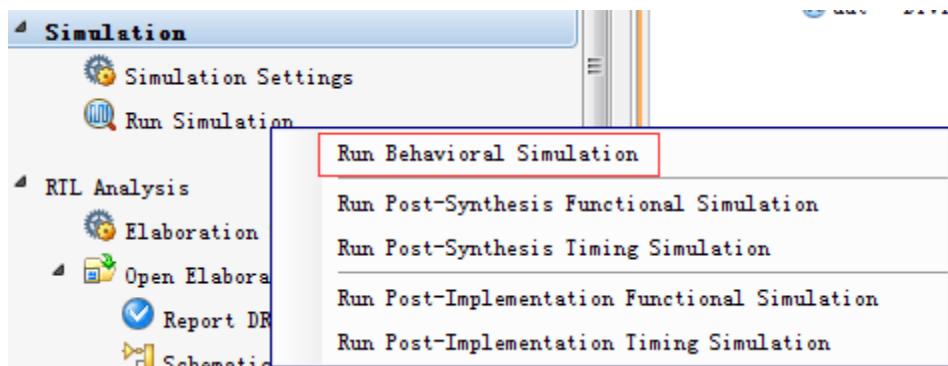
Step6:单击 Simulation Settings 对仿真参数做一些设置



Step7:单击 Simulation 设置仿真时间为 1000ms

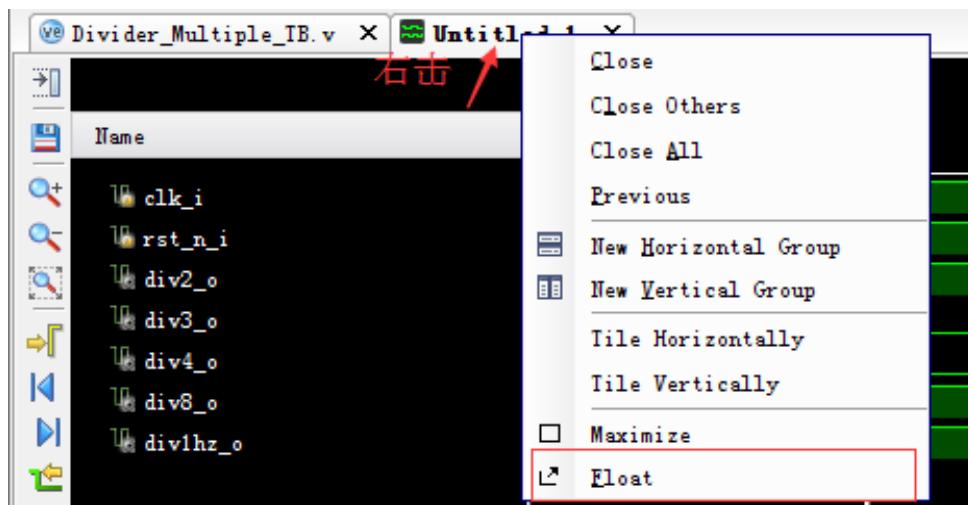
9.7.3 行为级仿真

Step1:单击 Run Simulation

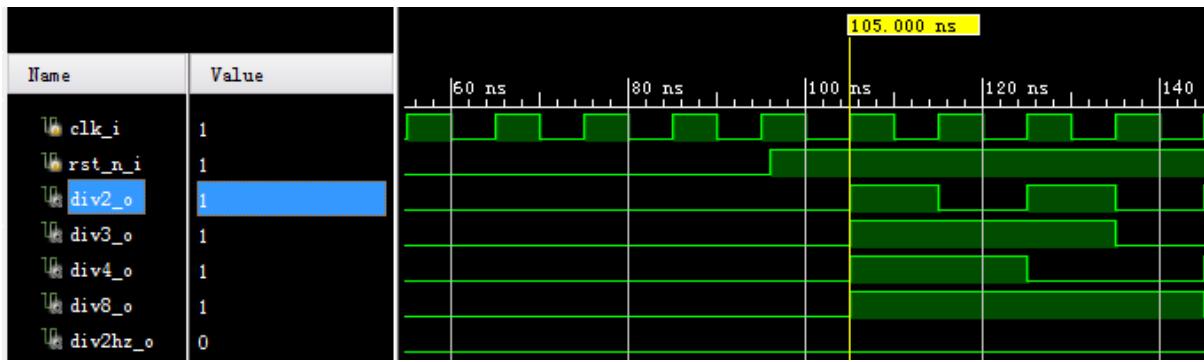


Step2:计算机 CPU 会模拟 FPGA 的运行，1000ms 运行来说通常需要几分钟时间。具体时间和 CPU 的配置有很大关系。

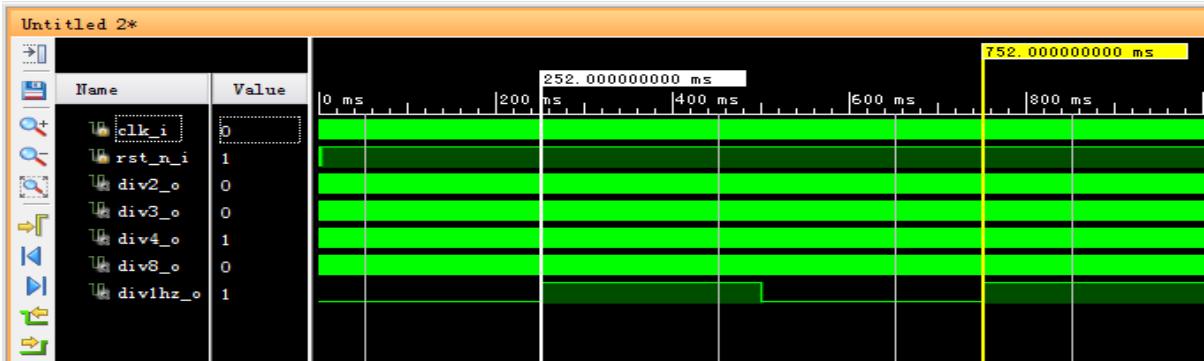
Step3:仿真结束后查看波形，为了观察方便，右击窗口选择 float



Step4:使用放大工具放大后观察

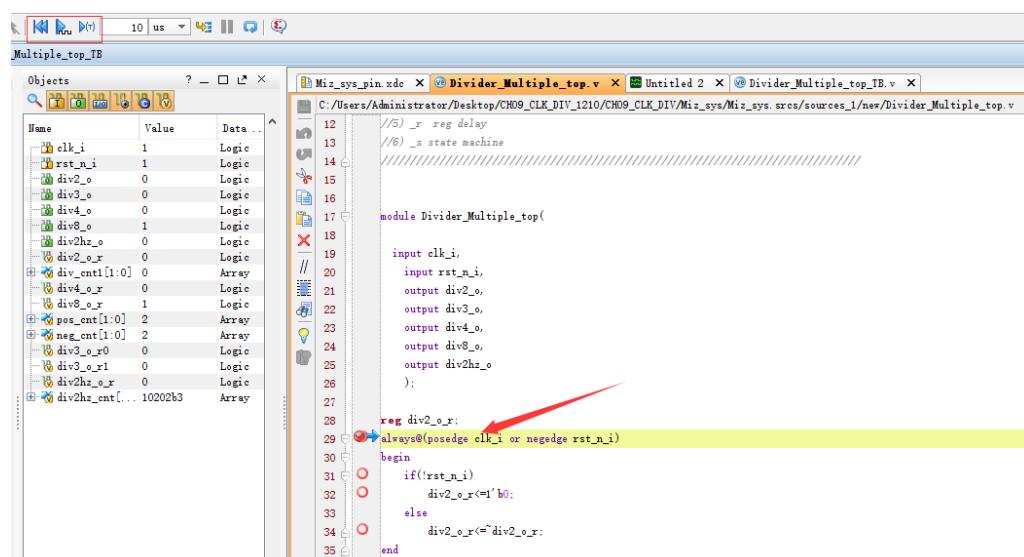


Step5: 使用放大工具放大后观察

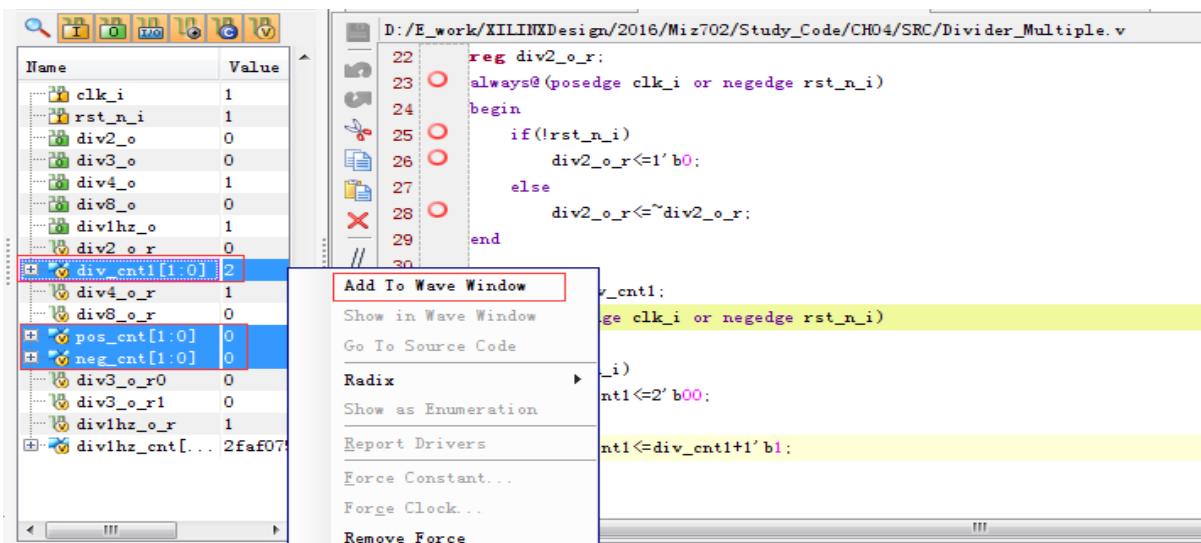


Step6: 断点观察更多信号

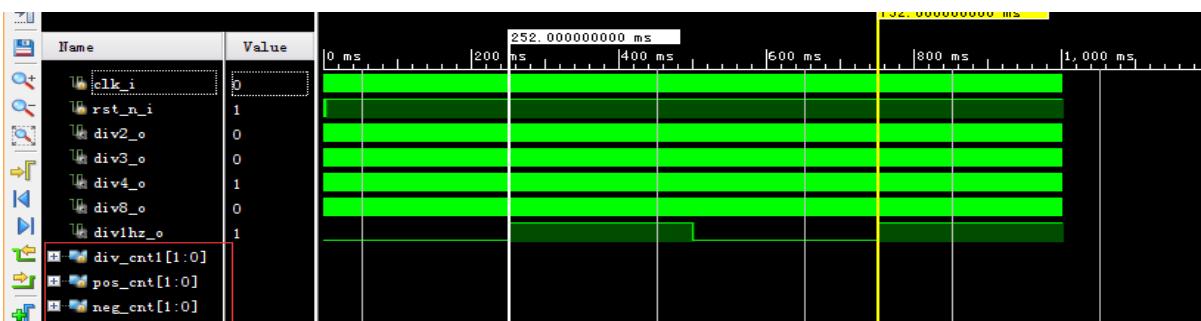
- 1、打开 divider_multiple_top.v 文件只要是显示红色圆圈的位置就是可以设置断点，单击红色圆圈。
- 2、单击运行
- 3、可以看到红色线框内有很多信号了，这些就是内部的运行信号，可以让我们观察程序更加仔细。



Step7: 用鼠标单击这个几个信号，然后右击后单击 Add To Wave Window

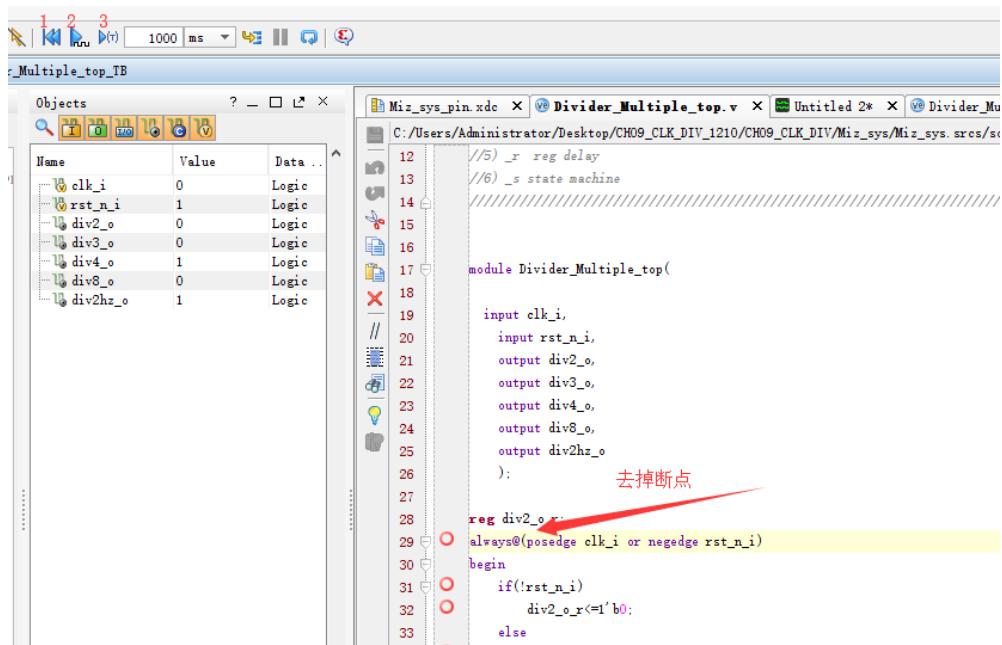


Step8:可以看到我们添加进来的三个寄存器变量

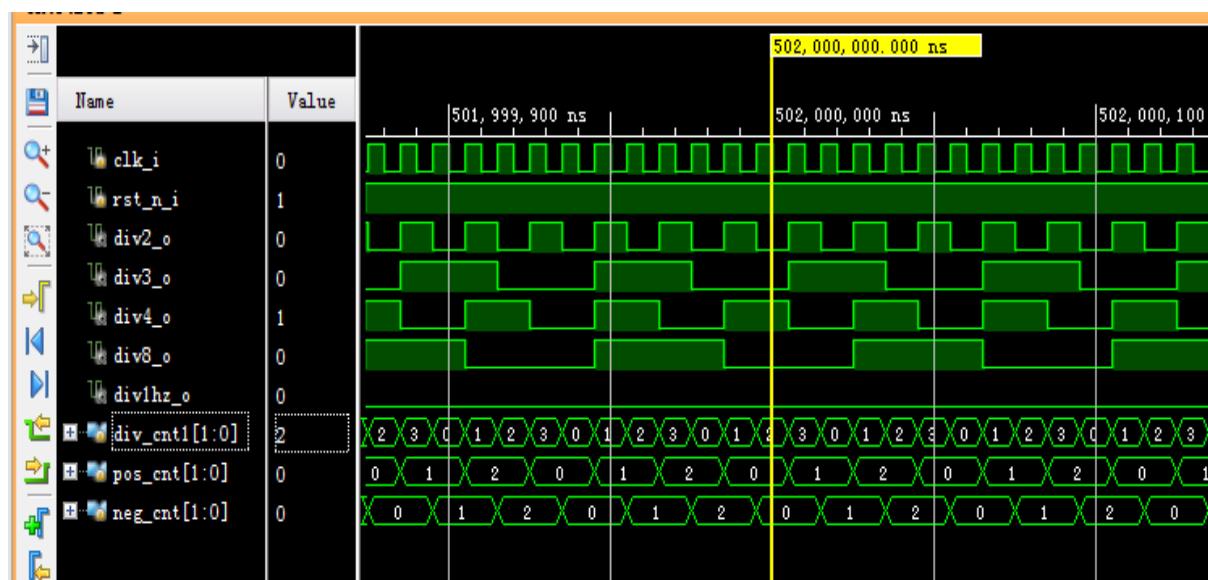


Step9:重新仿真 按钮 1 是初始化仿真 按钮 2 是仿真开始 按钮 3 是仿真到设置时间

- 1、去掉刚才设置的断点
 - 2、单击 1 处按钮重新加载初始化仿真
 - 3、设置仿真时间为 1000ms
 - 4、单击 3 处按钮

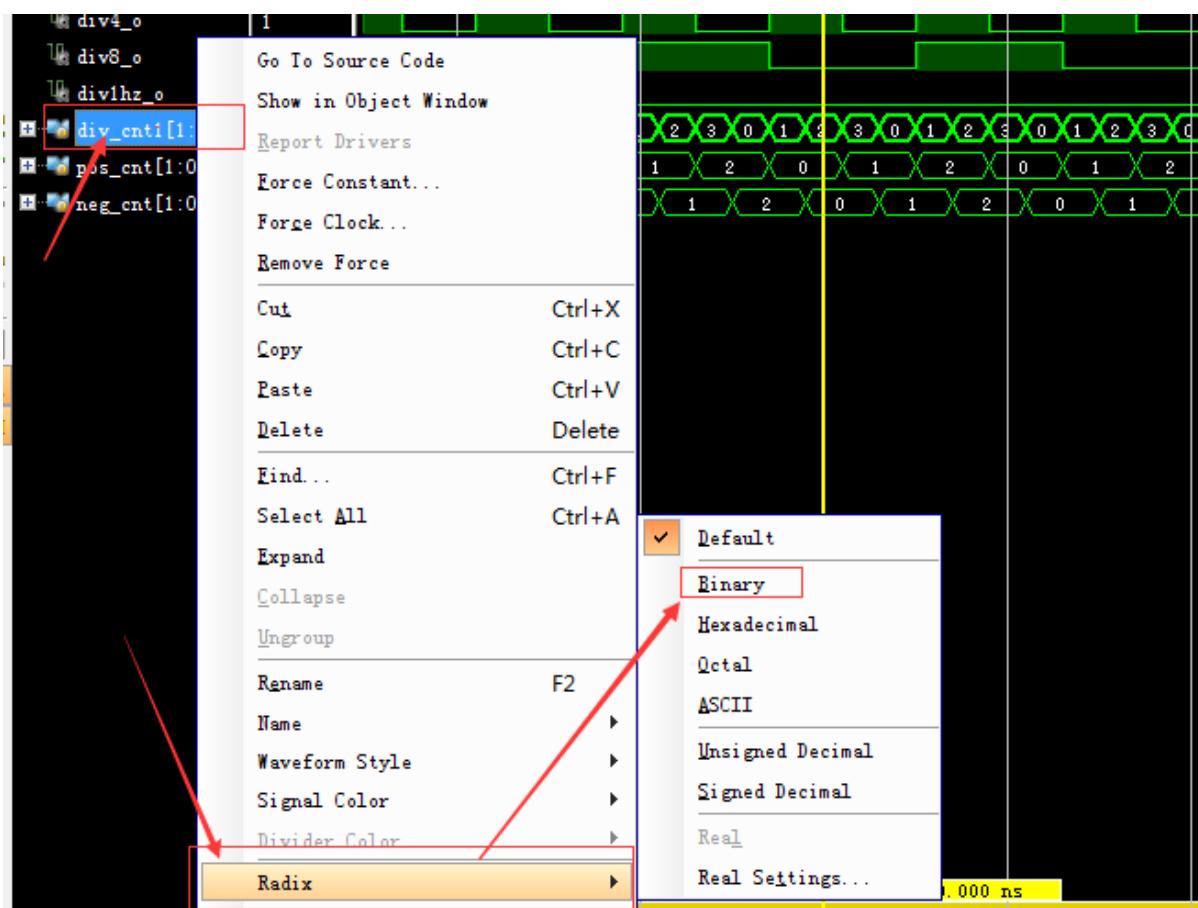


Step10:重新仿真后结果

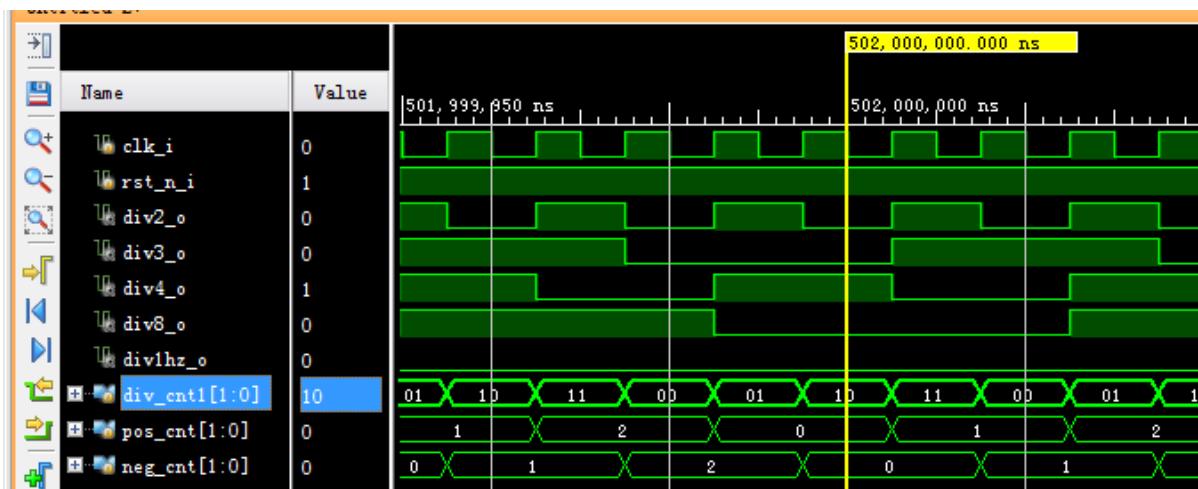


Step11:设置观察的数据类型

- 1、首先选择一个要观察的变量
- 2、右击选择 Radix
- 3、假设选择 Binary 以二进制形式观察



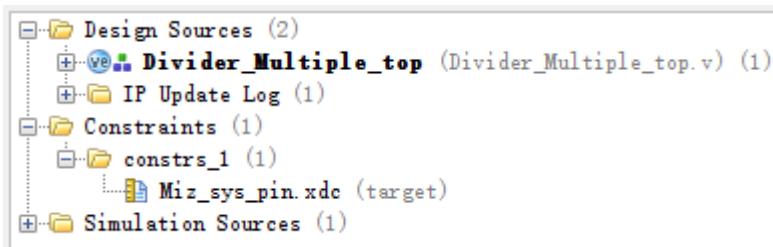
Step12:设置好后的效果



9.8 综合 Synthesis

9.8.1 添加文件

Step1:把 Divider_multiple_top.v 添加进来，并且添加 xdc 管脚约束文件



XDC 约束文件

```
set_property PACKAGE_PIN C8 [get_ports clk_i]
set_property IOSTANDARD LVCMOS15 [get_ports clk_i]

set_property PACKAGE_PIN A10 [get_ports rst_n_i]
set_property IOSTANDARD LVCMOS15 [get_ports rst_n_i]

set_property PACKAGE_PIN B9 [get_ports div2_o]
set_property IOSTANDARD LVCMOS15 [get_ports div2_o]

set_property PACKAGE_PIN J10 [get_ports div2hz_o]
set_property IOSTANDARD LVCMOS15 [get_ports div2hz_o]

set_property PACKAGE_PIN H11 [get_ports div3_o]
set_property IOSTANDARD LVCMOS15 [get_ports div3_o]

set_property PACKAGE_PIN G9 [get_ports div4_o]
set_property IOSTANDARD LVCMOS15 [get_ports div4_o]

set_property PACKAGE_PIN B10 [get_ports div8_o]
set_property IOSTANDARD LVCMOS15 [get_ports div8_o]
```

9.8.2 综合并查看报告

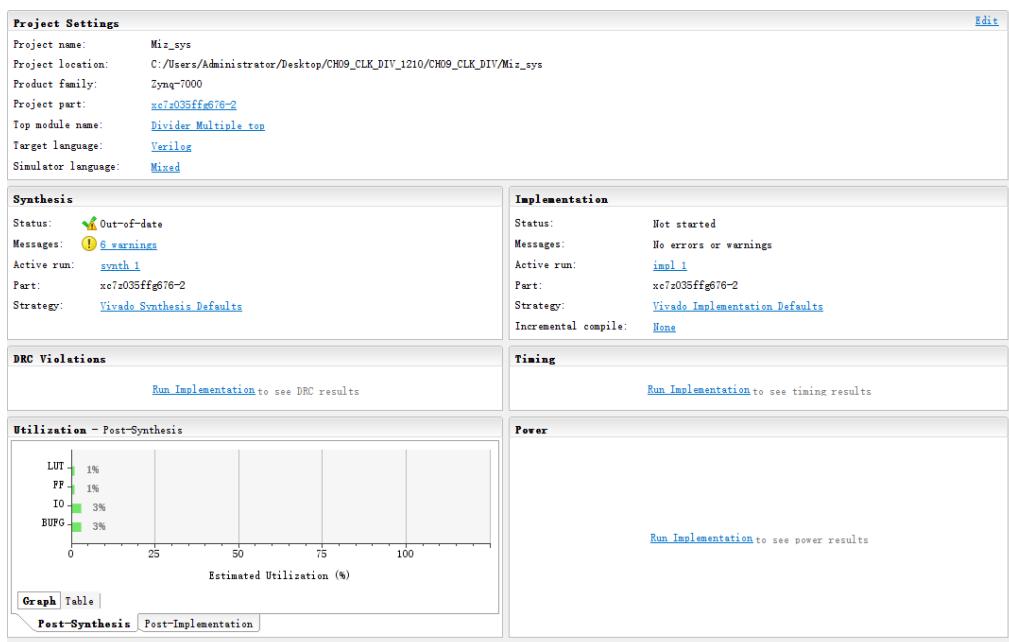
Step1:点击综合按钮



Step2:综合完成后通过查看报告看资源的利用情况

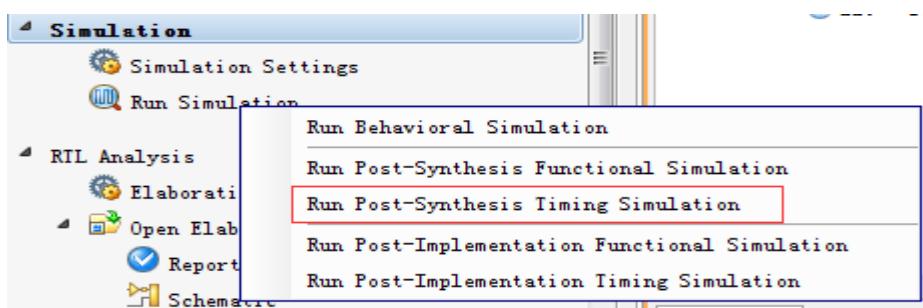


可以看到这个工程只是利用到了很少一部分资源

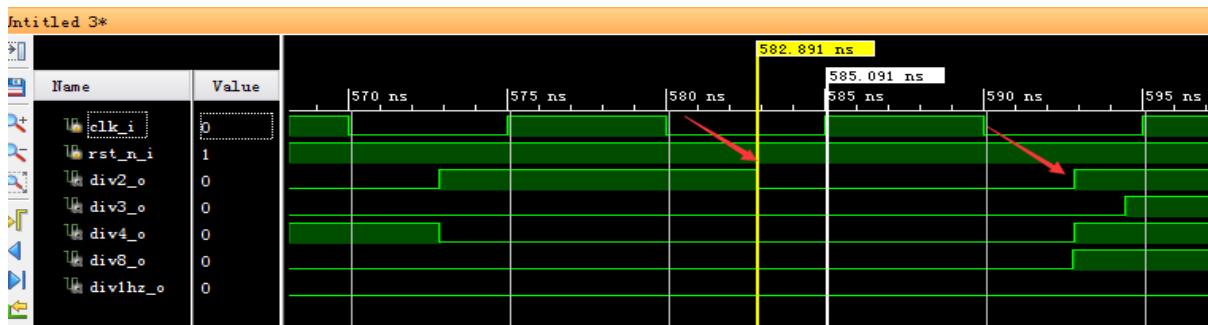


9.8.3 综合时序仿真

Step1:单击 Run Simulation 选择 Run Post-synthesis Timing Simulation



Step2:观察波形可以清晰看到综合后仿真加入了延迟更加接近实际芯片的运行情况



9.9 执行 Implementation

9.9.1 执行并查看报告

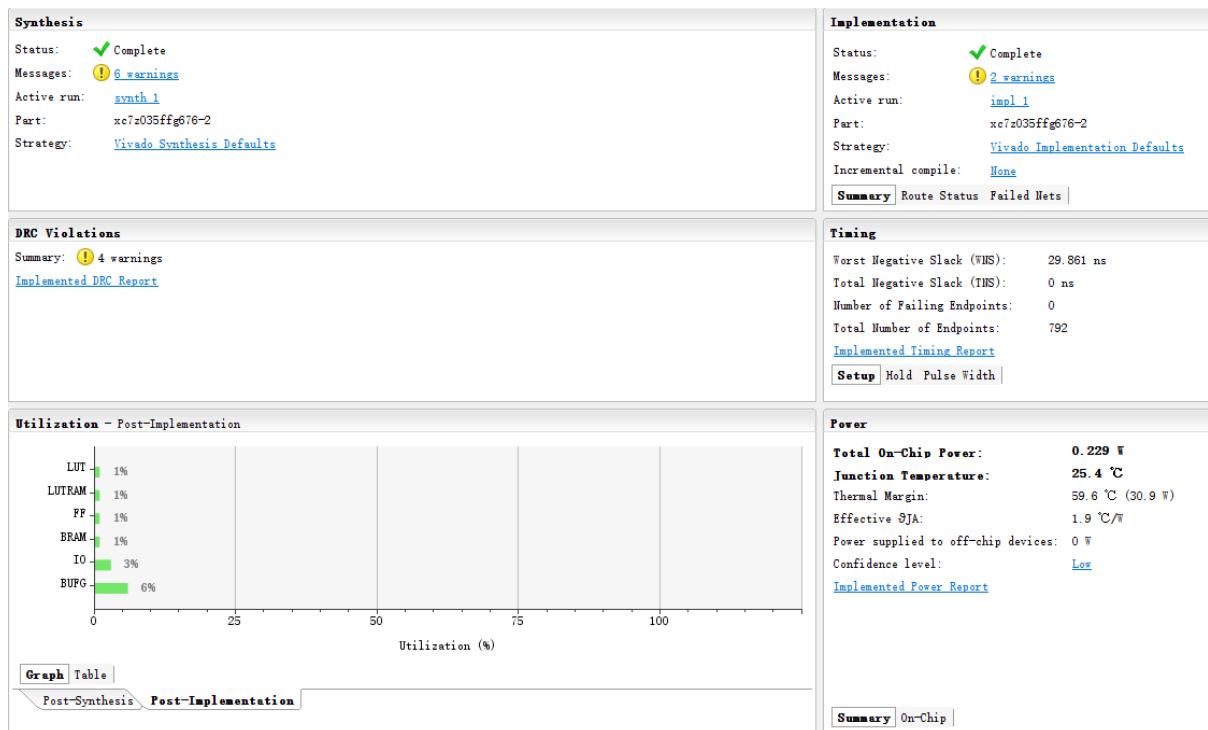
Step1:点击执行按钮



Step2:执行运行完毕后再次单击



Step3:查看执行运行完毕后的报告,执行完成后的报告比综合后的报告相比,是精确的分析和评估



Step4:点开 Table 可以看到使用的资料的具体参数

The screenshot shows two windows side-by-side. The top window is titled 'DRC Violations' and displays a summary with 4 warnings and a link to the 'Implemented DRC Report'. The bottom window is titled 'Utilization - Post-Implementation' and contains a table of resource utilization. The table has columns for Resource, Utilization, Available, and Utilization %. The data is as follows:

Resource	Utilization	Available	Utilization %
LUT	1010	17600	5.74
LUTRAM	94	6000	1.57
FF	1528	35200	4.34
BRAM	0.50	60	0.83
IO	7	100	7.00
BUFG	2	32	6.25

Below the table, there are tabs for 'Graph' and 'Table', with 'Table' being selected. At the bottom, there are tabs for 'Post-Synthesis' and 'Post-Implementation', with 'Post-Implementation' being selected.

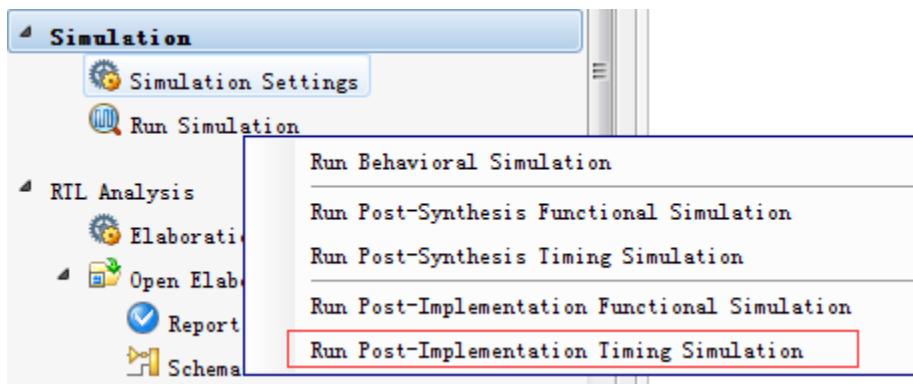
Step5:查看执行完后的时序约束报告

The screenshot shows the 'Timing - Timing Summary - timing_2' window. On the left is a tree view of timing constraints: General Information, Timer Settings, Design Timing Summary (with Clock Summary (1) expanded), Check Timing (24) (with no_clock (6), constant_clock (0), pulse_width_clock (0), unconstrained_internal (1), no_input_delay (1), no_output_delay (5), multiple_clock (0), generated_clocks (0) listed), and All user specified timing constraints are met. The main pane is titled 'Design Timing Summary' and contains tables for Setup and Hold times. The Setup table includes fields like Worst Negative Slack (WNS), Total Negative Slack (TNS), Number of Failing Endpoints, and Total Number of Endpoints. The Hold table includes fields like Worst Hold Slack (WHS), Total Hold Slack (THS), Number of Failing Endpoints, and Total Number of Endpoints. Both tables show values such as 4.453 ns, 0.263 ns, 0, 32, and 0.000 ns, 0.000 ns, 0, 33 respectively. A note at the bottom states 'All user specified timing constraints are met.'

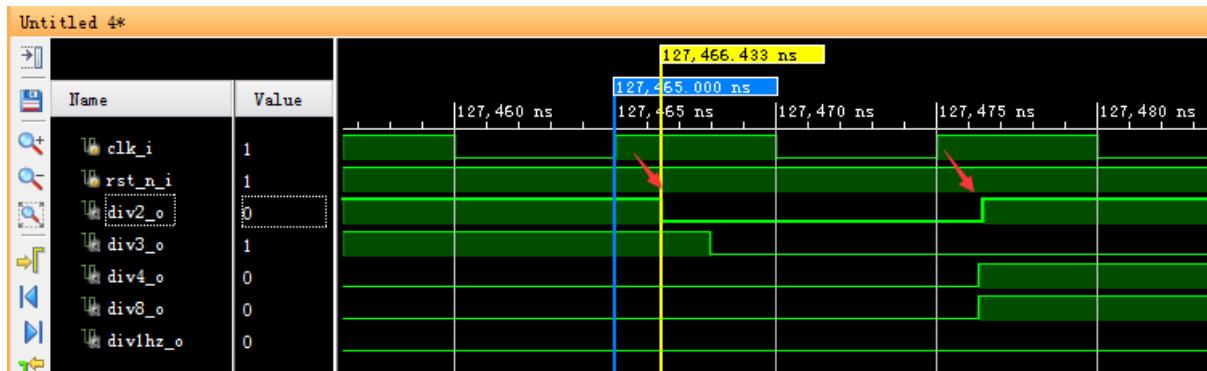
时序约束报告是 FPGA 开发中很重要的一项参数，所以必须看一下是否有违反时序约束的情况。可以看到有一些黄色的 warning。在这里不会影响我们的输出结果，因为我们这输出并没有做时序约束。但是如果要输出很严格的时序就需要加上时序约束。

9.9.2 布局布线后时序仿真

Step1:单击 Run Simulation 选择 Run Post-Implementation Timing Simulation



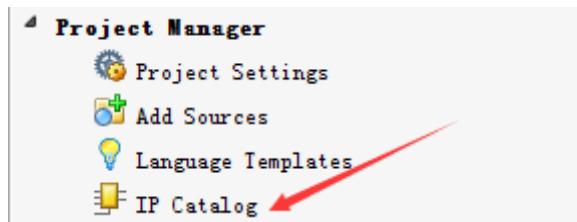
Step2: 观察波形可以清晰看到布局布线后仿真加入了延迟这要比综合后的时序更加接近真实的情况



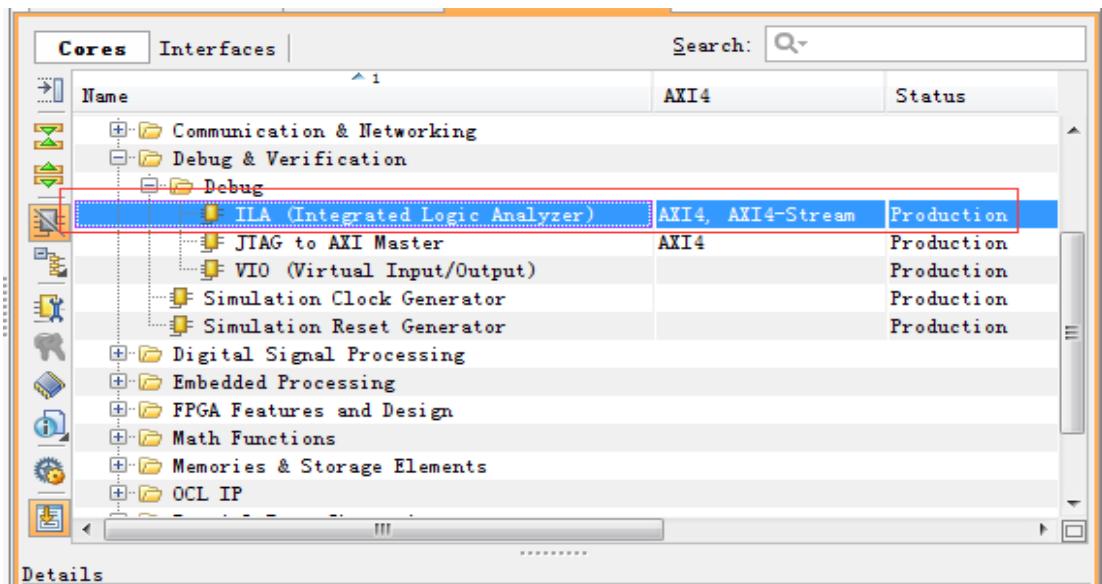
9.10 VIVADO 在线逻辑分析仪使用

9.10.1 IP Catalog 添加 ILA ip core

Step1: 单击 IP Catalog

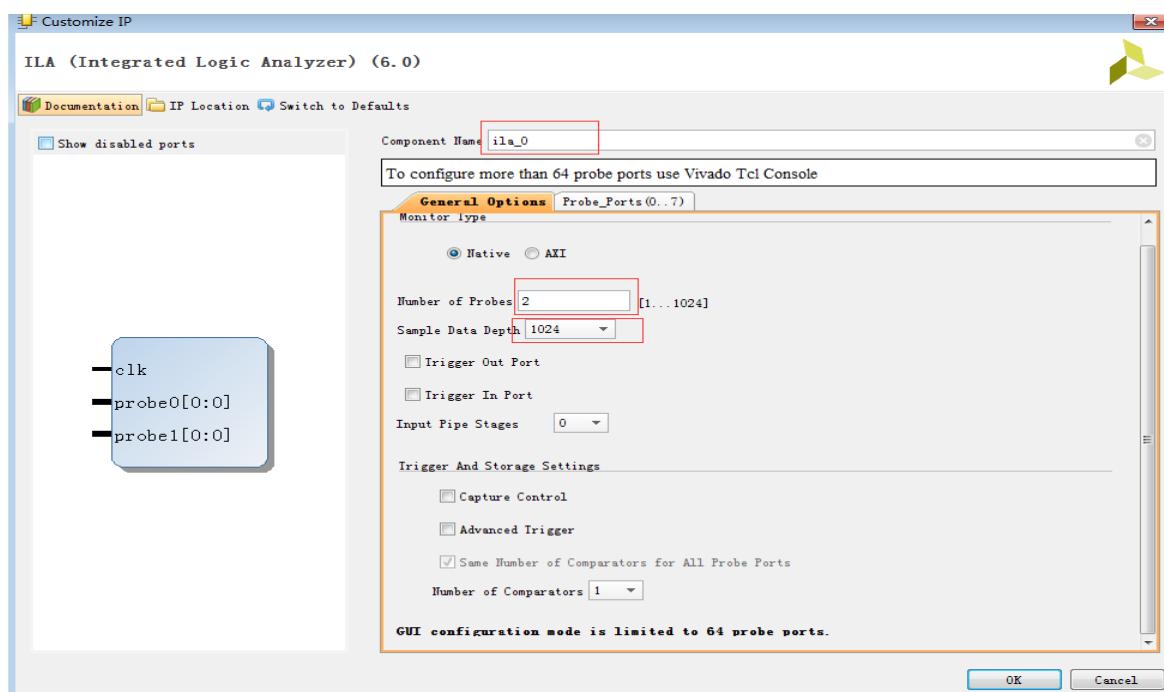


Step2: 打开 Debug & Verification > Debug -> 双击 ILA



Step3:游标 General Options 设置如下

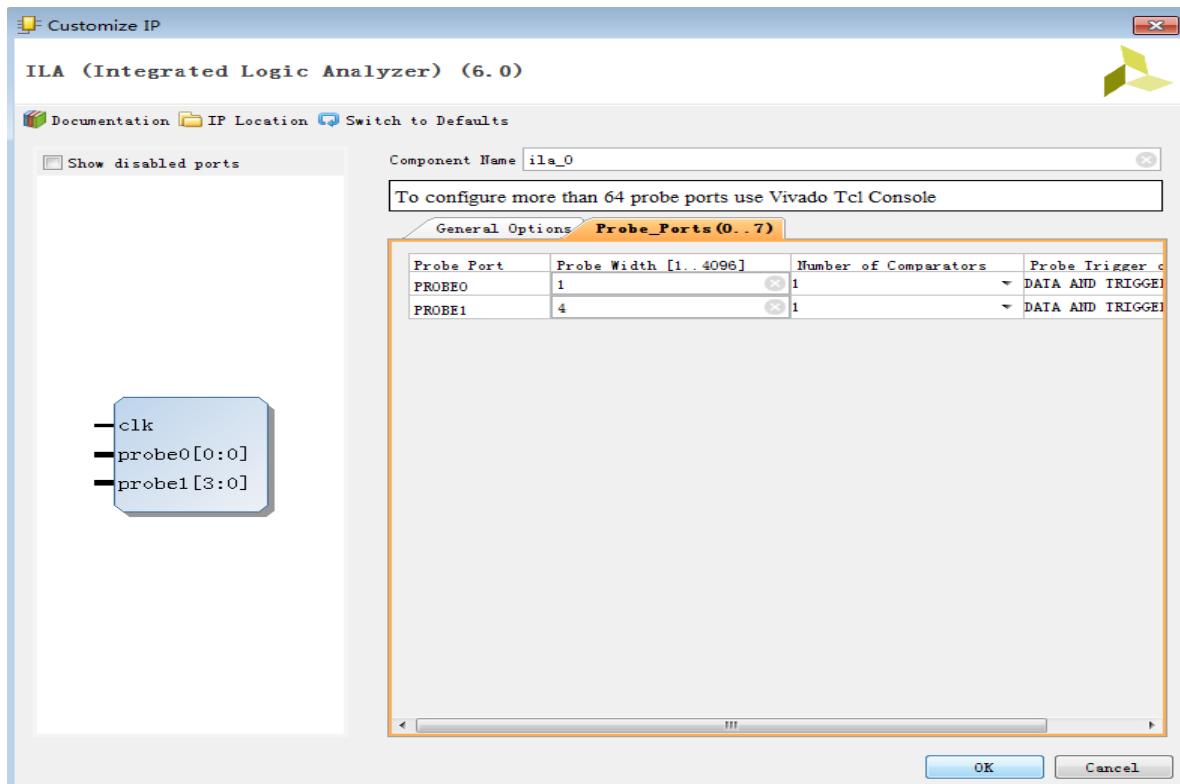
- 1、Number of probes 2 为设置需要观察信号的组为2组,因为我们准备1组放触发信号,1组放普通观察的信号
- 2、Sample Data Depth 1024 设置采样的深度,这是需要消耗 FPGA 的 BRAM 的 BRAM 越大可以设置的采样深度就越大,当然编译速度会降低。



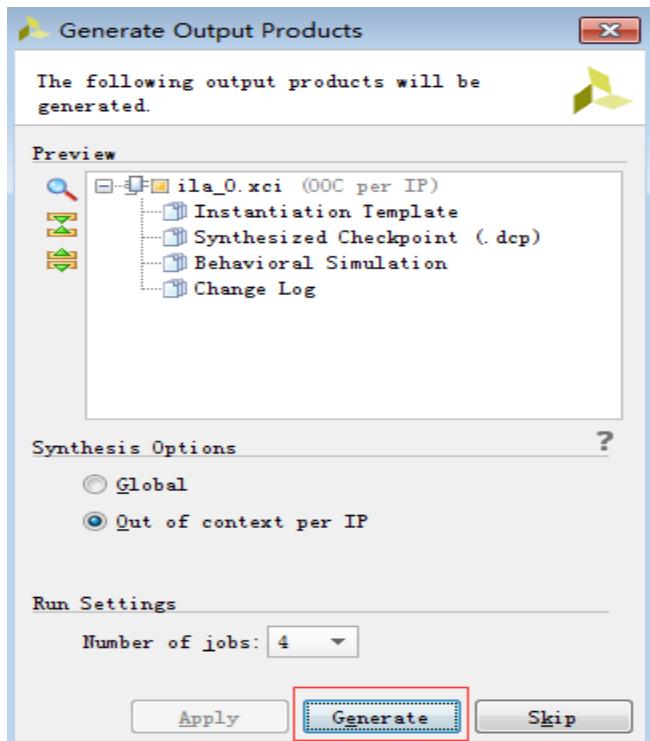
Step4:游标 Probe_Ports 设置如下

Probe Port 探针类似示波器的表笔,只是这里是在 FPGA 内部,我们设置了 Probe0 用来检查 2HZ 的信号, Probe1 用来检测另外 4 个分频信号。

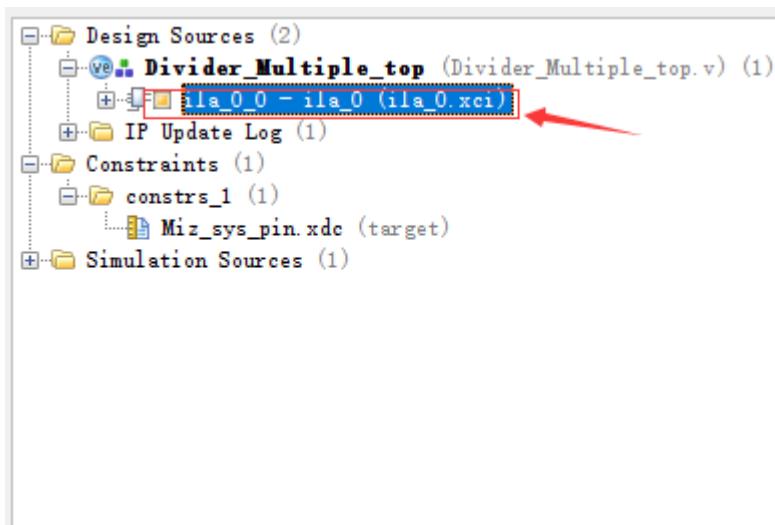
设置好后单击 OK 关闭窗口



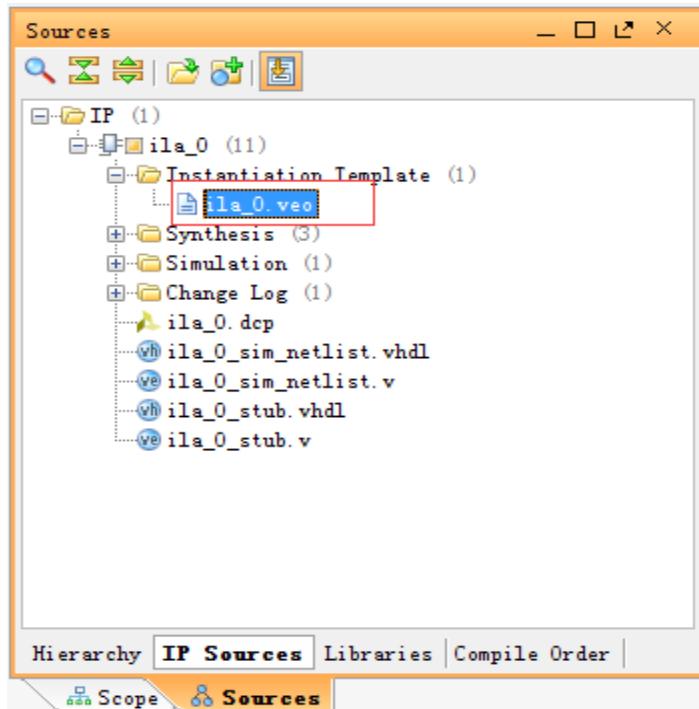
Step5: 直接单击 Generate



Step6: 可以看到 ila 这个逻辑分析仪的 IP 添加进来了



Step7:切换 IP Sources 游标下，然后双击 ila_0.veo 打开调用的接口模版



Step8:IP 接口调用模版打开后，可以看到这是一个 IP 接口，显然我们只要把需要被检测的信号根据前面的设置填进去就可以了。clk 就是采样时钟，probe0 就是 2HZ 信号，probe1 就是其他需要被观察的信号。

```

56 ila_0 your_instance_name (
57     .clk(clk), // input wire clk
58
59
60     .probe0(probe0), // input wire [0:0] probe0
61     .probe1(probe1) // input wire [3:0] probe1
62 );

```

修改，并且嵌入到顶层文件中

```

ila_0 ila_0_0 (
    .clk(clk_i), // input wire clk
    .probe0(div2hz_o), // input wire [0:0] probe0
    .probe1({div2_o,div3_o,div4_o,div8_o}) // input wire [3:0] probe1
);

```

9.10.2 逻辑分析仪抓取的信号

设置好逻辑分析仪，需要抓取的信号为

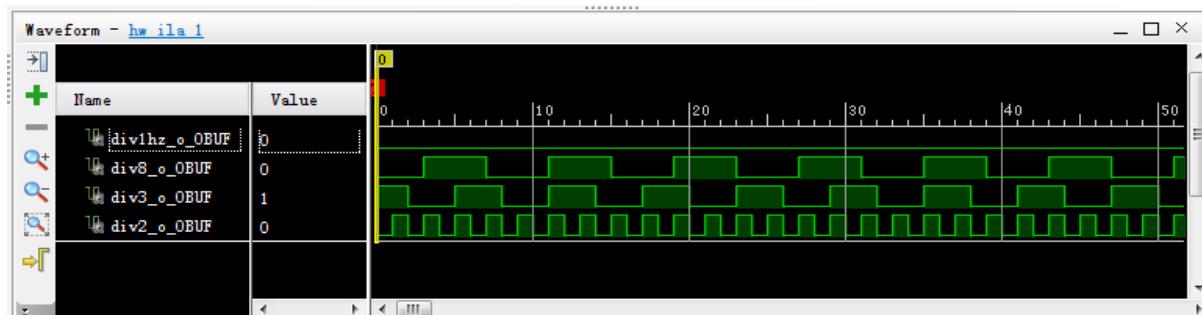
div2_o_r,

```

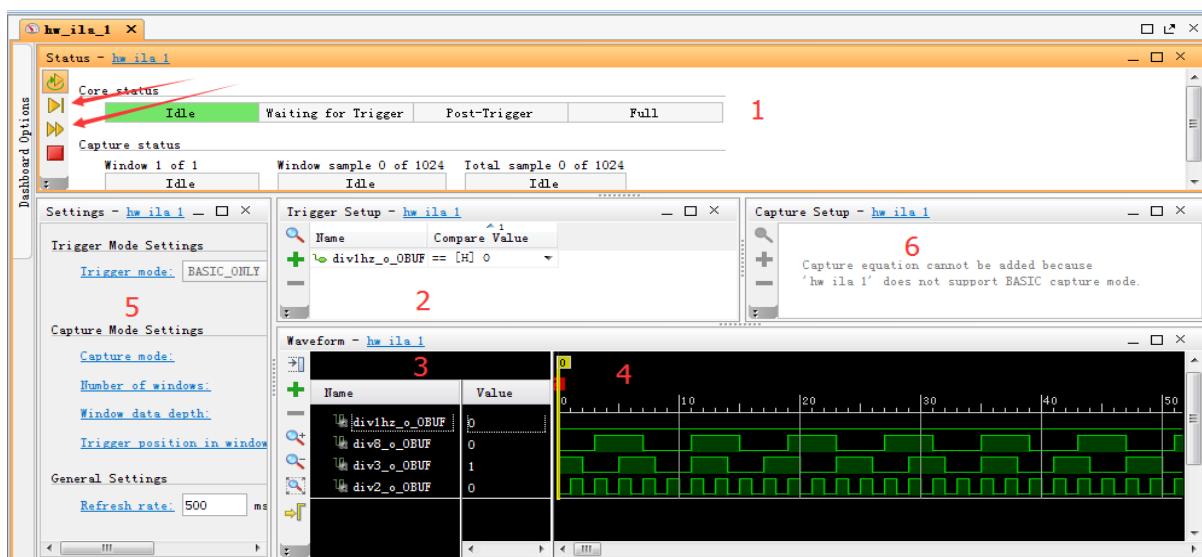
div3_o_r,
div3_o_r0,
div3_o_r1,
div4_o_r,
div8_o_r。

```

逻辑分析仪抓取的信号如下图所示。



9.10.3 逻辑分析仪使用



区域 1：设置采样的启动停止，和采样的方式

区域 2：设置触发信号

区域 3：被观察的信号名字

区域 4：被观察的信号波形

区域 5：触发模式设置

区域 6：触发设置

那么我们主要使用的有 1、2、3、4 这几个区域。

9.11 小结

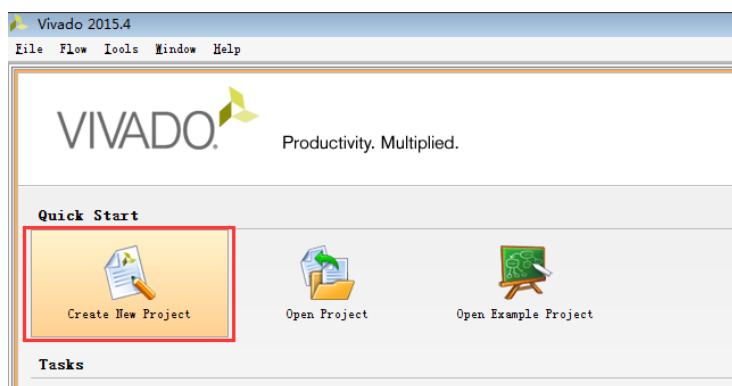
本章全面介绍了 VIVADO 的 FPGA 开发流程规范。包括了程序设计、行为仿真、综合过程、综合后时序仿真、执行过程、执行后仿真、FPGA 资源的利用情况分析、利用 VIVADO 自带的逻辑分析仪抓取信号波形，进行分析、IAL 逻辑分析仪 IP 的使用和设置。本章非常适合从 ISE 转向 VIVADO 开发的工程，或者 ALTERA 开发转向 XILINX 开发的工程师、或者没有 FPGA 开发基础的初学者。

CH10_ HDMI 接口测试

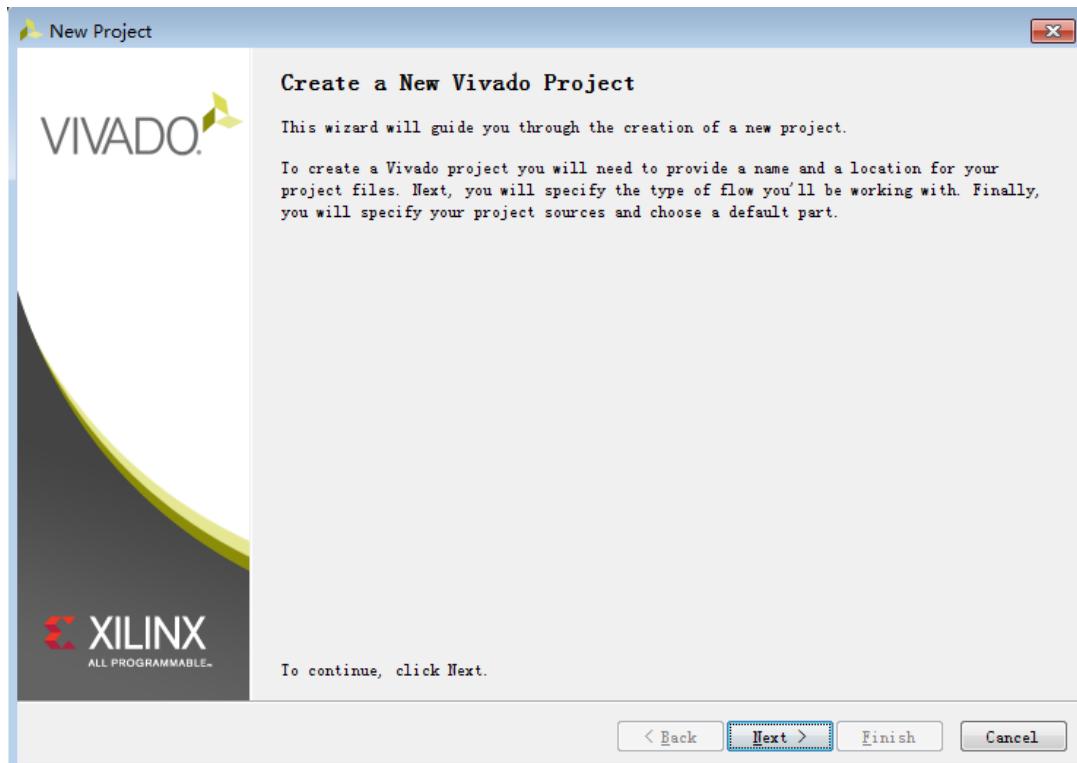
HDMI 既高清晰度多媒体接口（High Definition Multimedia Interface），这是一种数字视频/音频接口，相比较前面介绍的 VGA 接口，它传输的信息量大，色彩度高，传输速度快等显著优点。我们的 MiZ7035 开发板上面采用了 PLIO 模拟的方式来实现 HDMI 功能，这章就将带领大家使用我们封装好的 IP 核来设计一个 HDMI 接口的测试实验。

10.1 创建工程文件

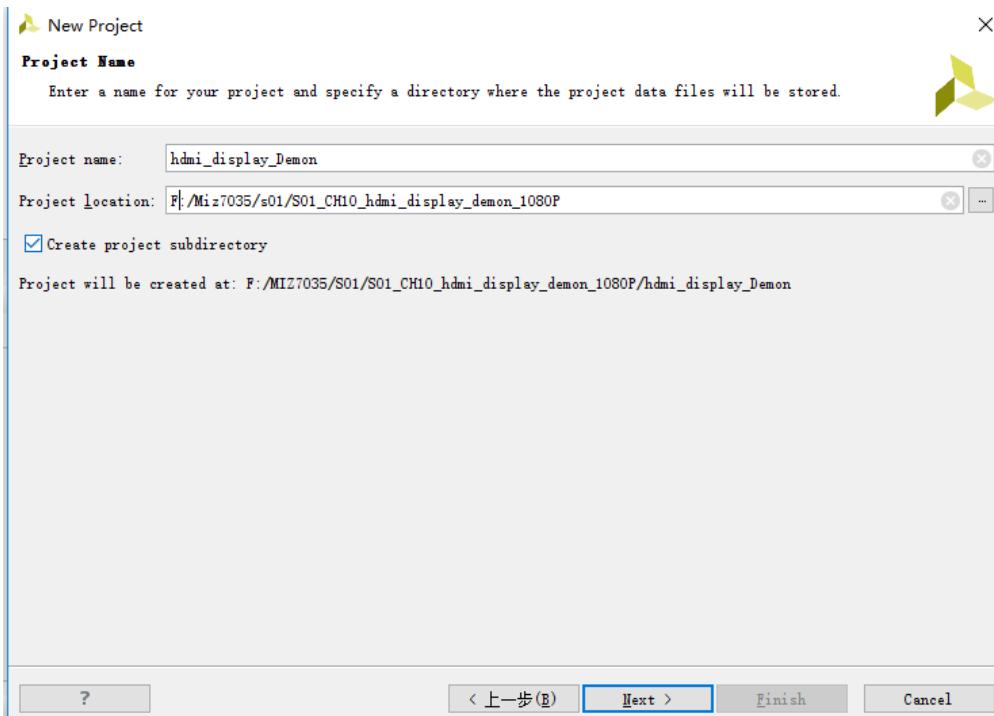
Step1: 创建工程



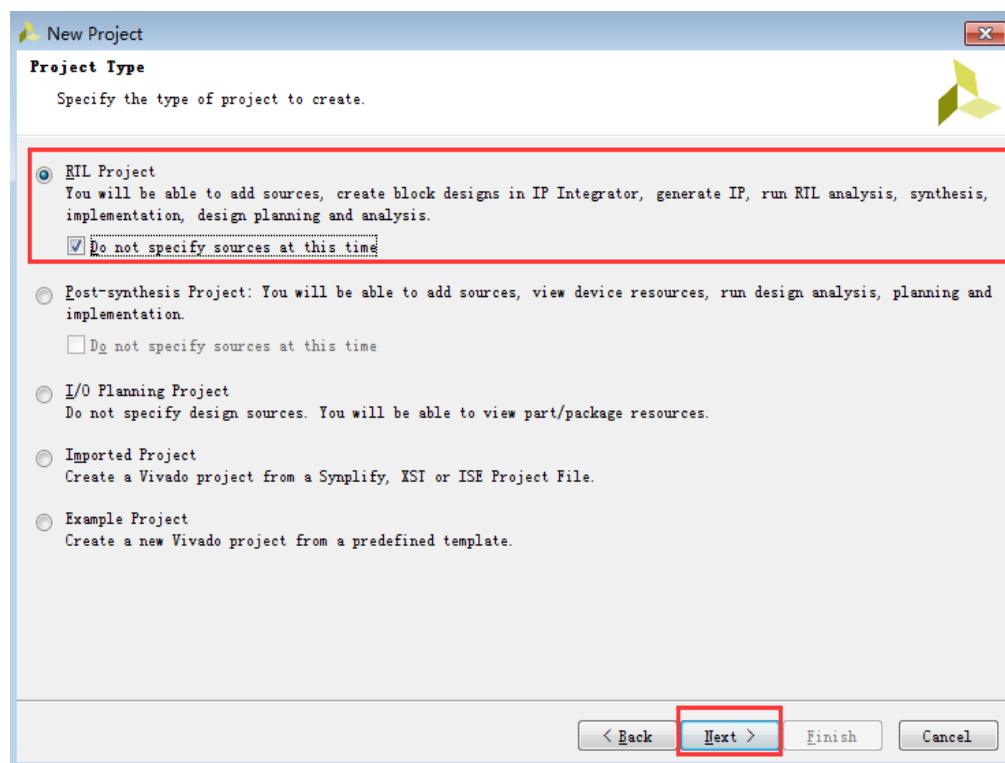
Step2: 欢迎界面直接单击 NEXT



Step3: 工程名字命名为 hdmi_display_Demon，并且设置保存的路径，单击 NEXT

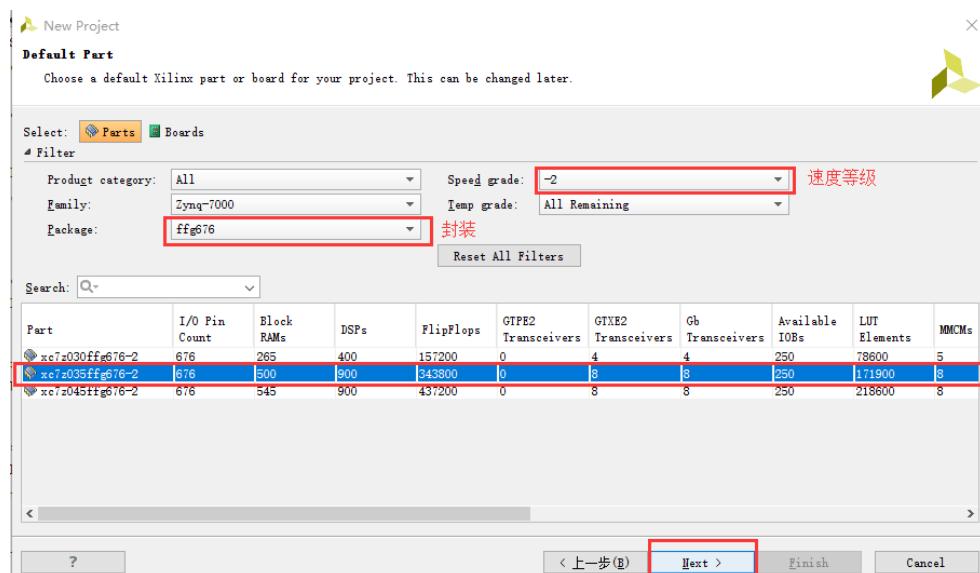


Step4: 新建一个 RTL 工程，并且勾选不要添加源文件，单击 NEXT



Step5: 选择芯片的型号和封装速度等级

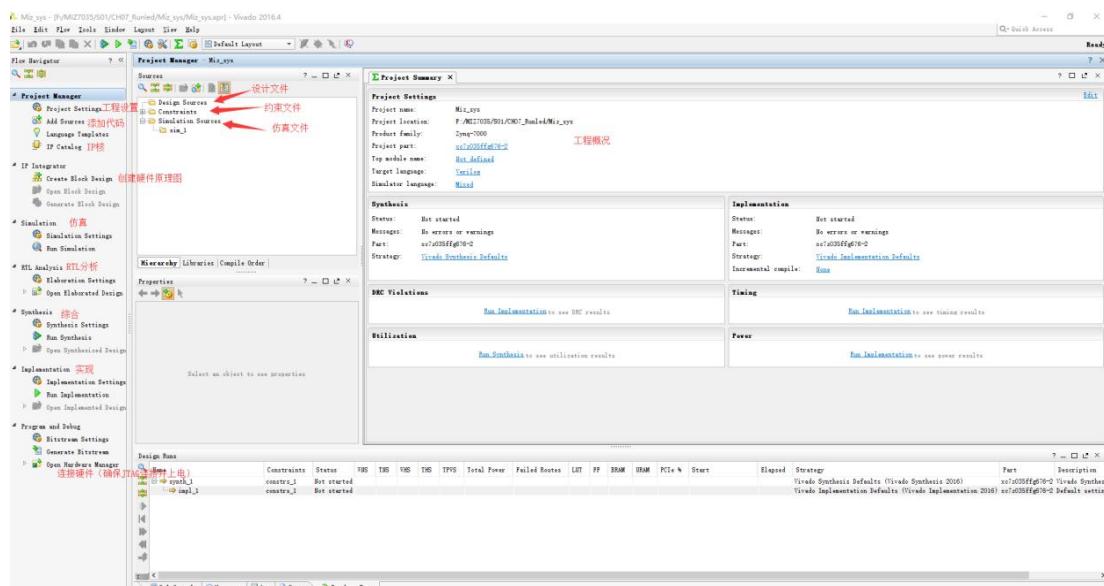
MiZ7035



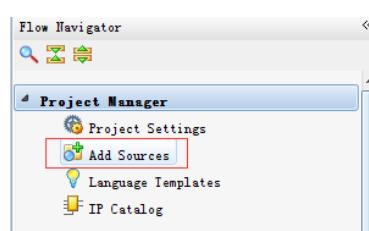
Step6:最后单击 Finish 完成工程的创建

10.2 添加工程文件

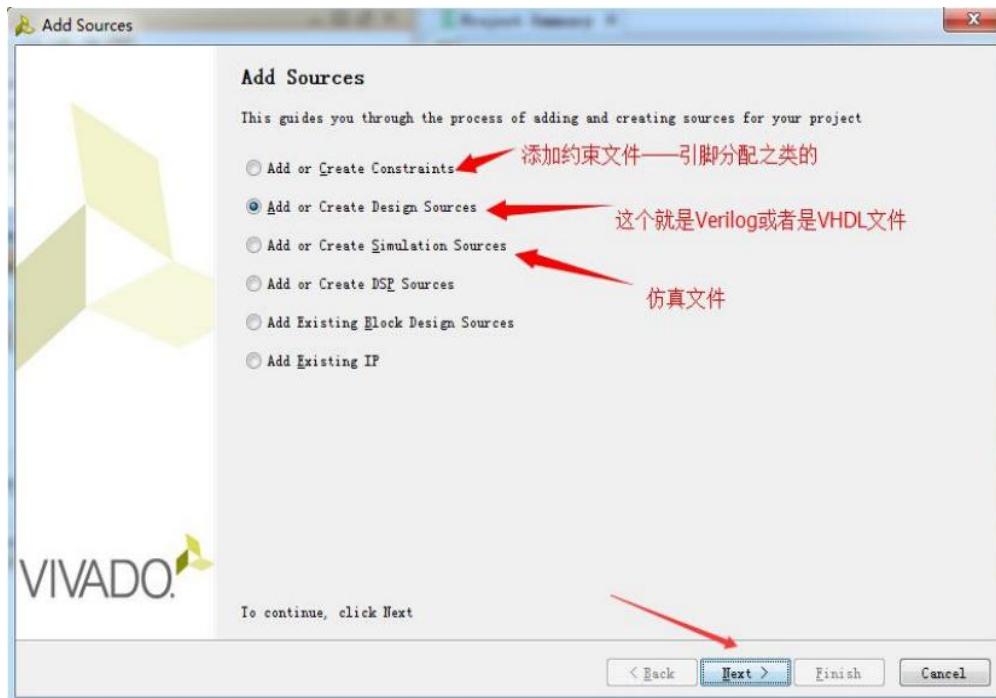
Step1:打开 VIVADO 软件



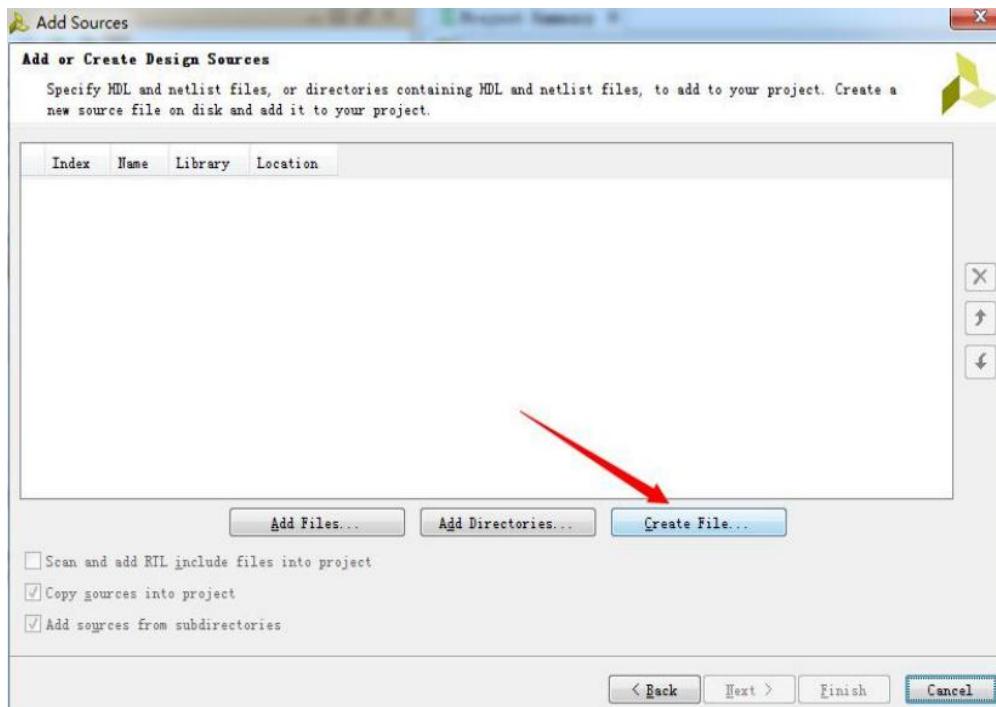
Step2:单击 Add Sources



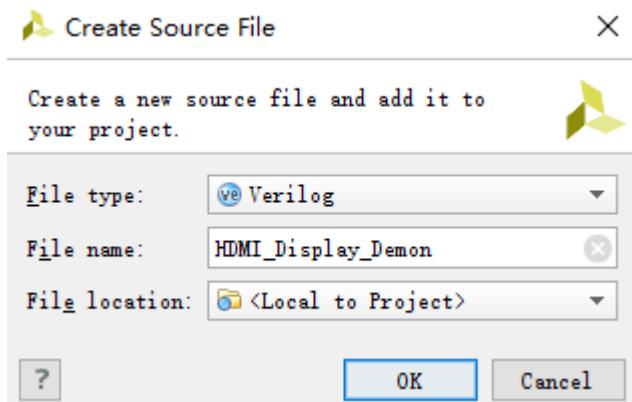
Step4: 选择单击 Add or Create Design Sources 然后单击 NEXT



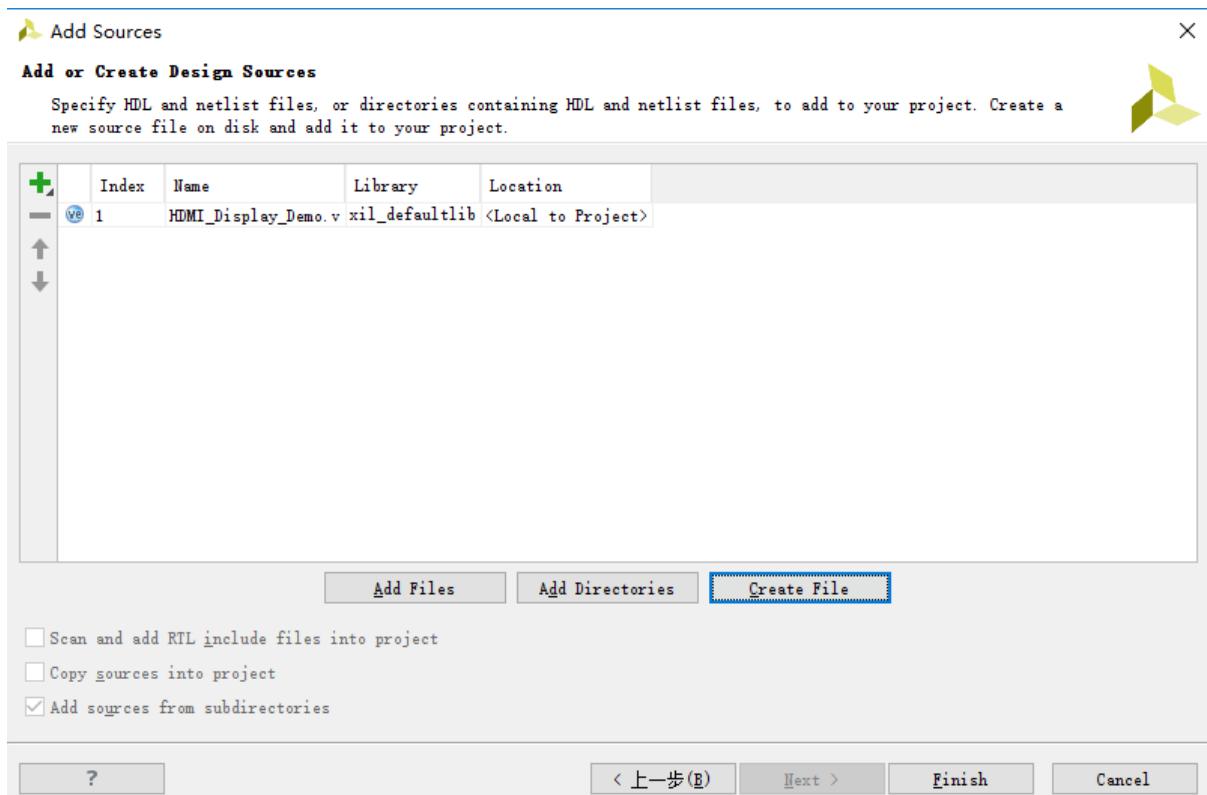
Step5: 单击 Create File 来创建文件



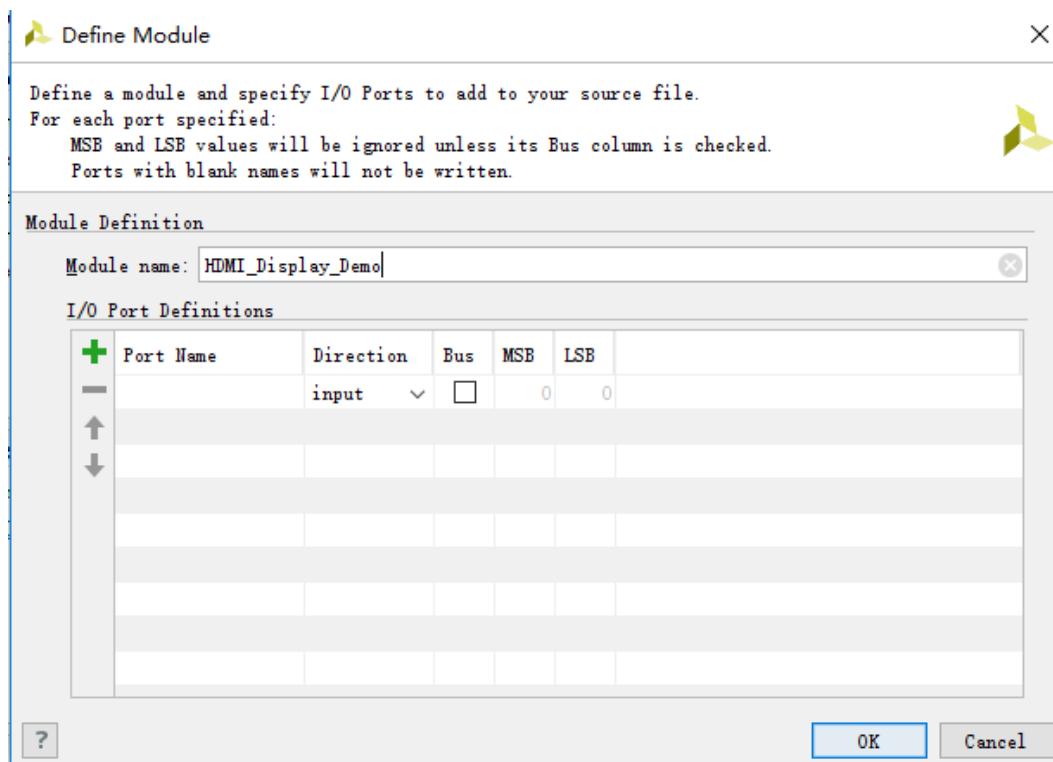
Step6: 创建一个 HDMI_Display_Demon 的文件，并且文件类型选择 Verilog



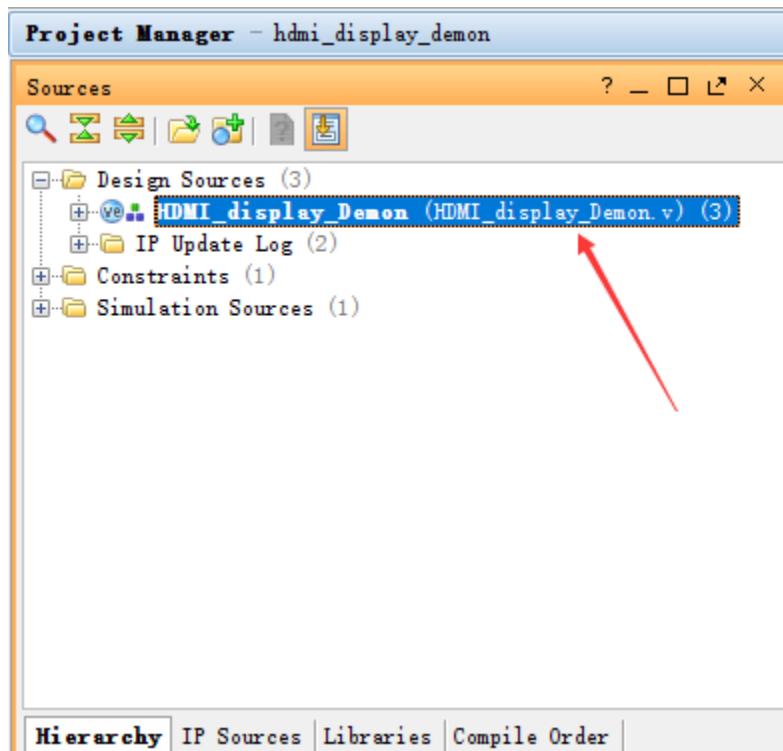
Step7:添加完成后如下图所示之后单击 finish 完成文件的创建



Step8:继续弹出的对话空中，可以设置一些端口，但是我们现在什么都不做。单击 OK



Step9: 创建完成后可以看到 Design Sources 文件夹中有了 HDMI_Display_Demon.v 这个文件



Step9: 创建完成后可以看到 Design Sources 文件夹中有了 HDMI_Display_Demon.v 这个文件，这个文件就是我们可以编写 verilog 程序的文件。

Step10: 双击 HDMI_Display_Demon.v 打开程序源码如下

```
`timescale 1ns / 1ps
///////////////////////////////
/////
// Company:
// Engineer:
//
// Create Date: 2017/02/22 14:12:56
// Design Name:
// Module Name: HDMI_Display_Demon
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
///////////////////////////////
/////

module HDMI_Display_Demon(
);

endmodule
```

可以看出这是一个空的工程，我们现在要添加代码同时也要添加工程信息。

Step11: 编写程序并且添加工程信息

```
module HDMI_display_Demon(
    input      clk_100M,
    input      KEY,
    output     HDMI_CLK_P,
    output     HDMI_CLK_N,
    output     HDMI_D2_P,
    output     HDMI_D2_N,
    output     HDMI_D1_P,
    output     HDMI_D1_N,
    output     HDMI_D0_P,
    output     HDMI_D0_N,
    output [3:0] LED
);

wire pixclk;
wire[7:0] R, G, B;
wire HS, VS, DE;
hdmi_data_gen u_hdmi_data_gen
(
    .pix_clk          (pixclk),
    .turn_mode        (KEY),
    .VGA_R            (R),
    .VGA_G            (G),
    .VGA_B            (B),
    .VGA_HS           (HS),
    .VGA_VS           (VS),
    .VGA_DE           (DE),
    .mode              (LED)
);

wire serclk;
wire lock;
wire[23:0] RGB;
assign RGB={R, G, B} ;
HDMI_FPGA_ML_0 u_HDMI
(
    .PXLCLK_I         (pixclk),
    .PXLCLK_5X_I       (serclk),
    .LOCKED_I          (lock),
    .RST_N             (1'b1),
    .VGA_HS             (HS),
    .VGA_VS             (VS),
```

```

    . VGA_DE          (DE),
    . VGA_RGB         (RGB),
    . HDMI_CLK_P      (HDMI_CLK_P),
    . HDMI_CLK_N      (HDMI_CLK_N),
    . HDMI_D2_P       (HDMI_D2_P),
    . HDMI_D2_N       (HDMI_D2_N),
    . HDMI_D1_P       (HDMI_D1_P),
    . HDMI_D1_N       (HDMI_D1_N),
    . HDMI_D0_P       (HDMI_D0_P),
    . HDMI_D0_N       (HDMI_D0_N)
);

clk_wiz_0  u_clk
(
    .clk_in1        (clk_100M),
    .resetn         (1'b1),
    .clk_out1       (pixclk),
    .clk_out2       (serclk),
    .locked         (lock)
);
endmodule

```

Step13: 按照之前的方法再添加一个 hdmi_data_gen的文件，并添加下面的程序。

```

module hdmi_data_gen
(
    input           pix_clk,
    input           turn_mode,
    output [7:0]   VGA_R,
    output [7:0]   VGA_G,
    output [7:0]   VGA_B,
    output          VGA_HS,
    output          VGA_VS,
    output          VGA_DE,
    output [3:0]   mode
);

//-----
// 水平扫描参数的设定 1280*720 60HZ
//-----

parameter H_Total     = 1650;
parameter H_Sync      = 40;
parameter H_Back      = 220;
parameter H_Active    = 1280;
parameter H_Front     = 110;
parameter H_Start     = 260;

```

```
parameter H_End      = 1540;
//-----//
// 垂直扫描参数的设定 1280*720 60HZ
//-----//
parameter V_Total    = 750;
parameter V_Sync     = 5;
parameter V_Back     = 20;
parameter V_Active   = 720;
parameter V_Front    = 5;
parameter V_Start    = 25;
parameter V_End      = 745;
reg[11:0] x_cnt;
always @(posedge pix_clk)      //水平计数
begin
    if(1'b0)
        x_cnt <= 1;
    else if(x_cnt==H_Total)
        x_cnt <= 1;
    else
        x_cnt <= x_cnt + 1;
end

reg hsync_r;
reg hs_de;
always @(posedge pix_clk)
begin
    if(1'b0)
        hsync_r <= 1'b1;
    else if(x_cnt==1)
        hsync_r <= 1'b0;
    else if(x_cnt==H_Sync)
        hsync_r <= 1'b1;

    if(1'b0)
        hs_de <= 1'b0;
    else if(x_cnt==H_Start)
        hs_de <= 1'b1;
    else if(x_cnt==H_End)
        hs_de <= 1'b0;
end

reg[11:0] y_cnt;
always @(posedge pix_clk)
begin
```

```
if(1'b0)
y_cnt <= 1;
else if(y_cnt==V_Total)
y_cnt <= 1;
else if(x_cnt==H_Total)
y_cnt <= y_cnt + 1;
end

reg vsync_r;
reg vs_de;
always @(posedge pix_clk)
begin
    if(1'b0)
        vsync_r <= 1'b1;
    else if(y_cnt==1)
        vsync_r <= 1'b0;
    else if(y_cnt==V_Sync)
        vsync_r <= 1'b1;

    if(1'b0)
        vs_de <= 1'b0;
    else if(y_cnt==V_Start)
        vs_de <= 1'b1;
    else if(y_cnt==V_End)
        vs_de <= 1'b0;
end

reg[7:0] grid_data_1;
reg[7:0] grid_data_2;
always @(posedge pix_clk) //格子图像
begin
    if((x_cnt[4]==1'b1)^ (y_cnt[4]==1'b1))
        grid_data_1 <= 8'h00;
    else
        grid_data_1 <= 8'hff;

    if((x_cnt[6]==1'b1)^ (y_cnt[6]==1'b1))
        grid_data_2 <= 8'h00;
    else
        grid_data_2 <= 8'hff;
end

reg[23:0] color_bar;
always @(posedge pix_clk)
```

```
begin
    if(x_cnt==260)
        color_bar <= 24'hff0000;
    else if(x_cnt==420)
        color_bar <= 24'h00ff00;
    else if(x_cnt==580)
        color_bar <= 24'h0000ff;
    else if(x_cnt==740)
        color_bar <= 24'hff00ff;
    else if(x_cnt==900)
        color_bar <= 24'hffff00;
    else if(x_cnt==1060)
        color_bar <= 24'h00ffff;
    else if(x_cnt==1220)
        color_bar <= 24'hfffffff;
    else if(x_cnt==1380)
        color_bar <= 24'h000000;
    else
        color_bar <= color_bar;
end

reg[16:0] key_counter;
reg[3:0] dis_mode;
assign mode=dis_mode;
always @(posedge pix_clk) //按键处理程序
begin
    if(turn_mode==1'b0)
        key_counter <= 14'b0;
    else if((turn_mode==1'b1)&(key_counter<=17'h11704))
        key_counter <= key_counter + 1'b1;

    if(key_counter==17'h11704)
        begin
            if(dis_mode==4'd12)
                dis_mode <= 4'd0;
            else
                dis_mode <= dis_mode + 1'b1;
        end
    end

reg[7:0] VGA_R_reg;
reg[7:0] VGA_G_reg;
reg[7:0] VGA_B_reg;
always @(posedge pix_clk)
```

```
begin
    if(1'b0)
        begin
            VGA_R_reg<=0;
            VGA_G_reg<=0;
            VGA_B_reg<=0;
        end
    else
        case(dis_mode)
            4'd0:begin
                VGA_R_reg<=0; //LCD 显示彩色条
                VGA_G_reg<=0;
                VGA_B_reg<=0;
            end
            4'd1:begin
                VGA_R_reg<=8'b11111111; //LCD 显示全白
                VGA_G_reg<=8'b11111111;
                VGA_B_reg<=8'b11111111;
            end
            4'd2:begin
                VGA_R_reg<=8'b11111111; //LCD 显示全红
                VGA_G_reg<=0;
                VGA_B_reg<=0;
            end
            4'd3:begin
                VGA_R_reg<=0; //LCD 显示全绿
                VGA_G_reg<=8'b11111111;
                VGA_B_reg<=0;
            end
            4'd4:begin
                VGA_R_reg<=0; //LCD 显示全蓝
                VGA_G_reg<=0;
                VGA_B_reg<=8'b11111111;
            end
            4'd5:begin
                VGA_R_reg<=grid_data_1; // LCD 显示方格 1
                VGA_G_reg<=grid_data_1;
                VGA_B_reg<=grid_data_1;
            end
            4'd6:begin
                VGA_R_reg<=grid_data_2; // LCD 显示方格 2
                VGA_G_reg<=grid_data_2;
                VGA_B_reg<=grid_data_2;
            end
        endcase
    end
```

```

色 4' d7:begin
    VGA_R_reg<=x_cnt[7:0]; //LCD 显示水平渐变
色    VGA_G_reg<=x_cnt[7:0];
    VGA_B_reg<=x_cnt[7:0];
end
色 4' d8:begin
    VGA_R_reg<=y_cnt[8:1]; //LCD 显示垂直渐变
    VGA_G_reg<=y_cnt[8:1];
    VGA_B_reg<=y_cnt[8:1];
end
变色 4' d9:begin
    VGA_R_reg<=x_cnt[7:0]; //LCD 显示红水平渐
    VGA_G_reg<=0;
    VGA_B_reg<=0;
end
变色 4' d10:begin
    VGA_R_reg<=0; //LCD 显示绿水平渐
    VGA_G_reg<=x_cnt[7:0];
    VGA_B_reg<=0;
end
变色 4' d11:begin
    VGA_R_reg<=0; //LCD 显示蓝水平渐
    VGA_G_reg<=0;
    VGA_B_reg<=x_cnt[7:0];
end
4' d12:begin
    VGA_R_reg<=color_bar[23:16]; //LCD 显示彩色条
    VGA_G_reg<=color_bar[15:8];
    VGA_B_reg<=color_bar[7:0];
end
default:begin
    VGA_R_reg<=8'b11111111; //LCD 显示全白
    VGA_G_reg<=8'b11111111;
    VGA_B_reg<=8'b11111111;
end
endcase
end

assign VGA_HS = hsync_r;
assign VGA_VS = vsync_r;

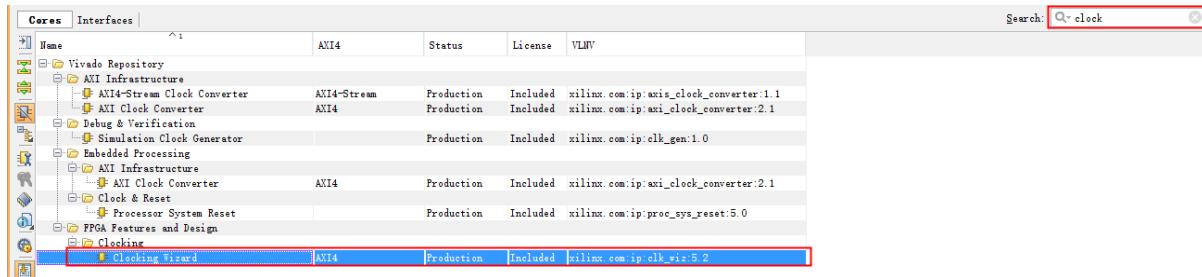
```

```

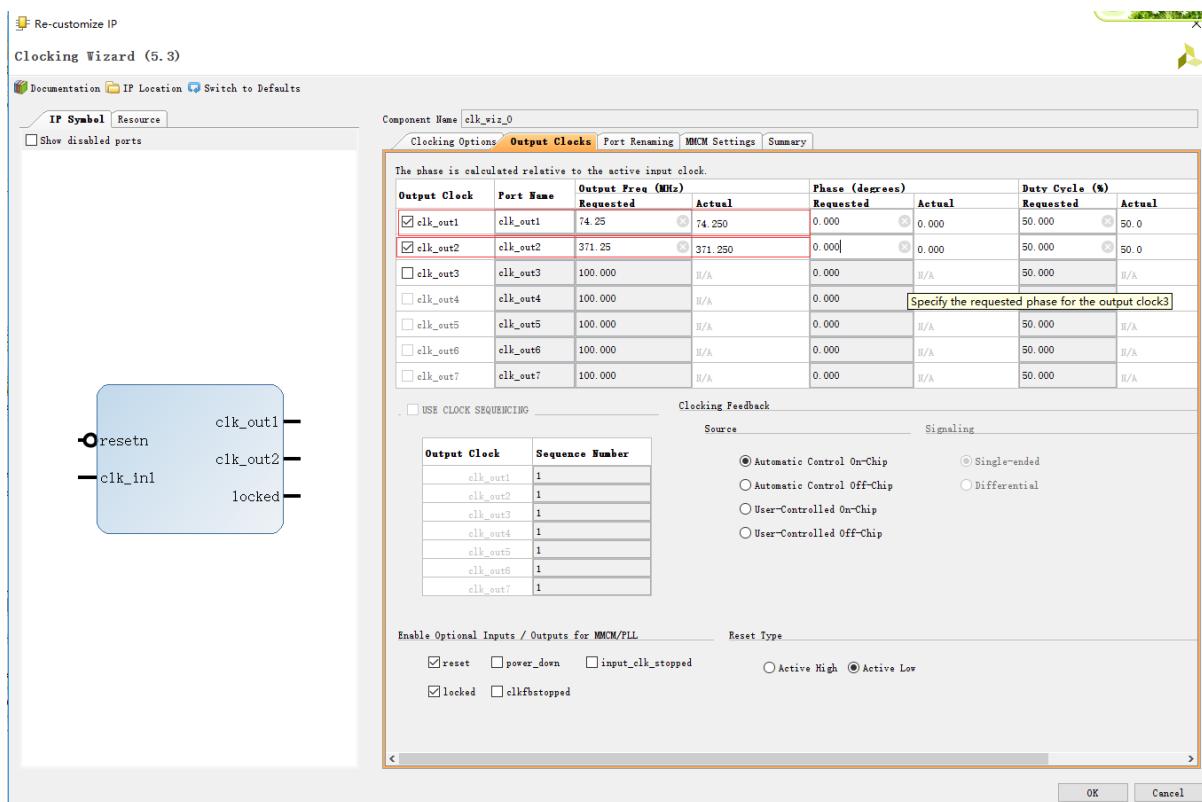
assign VGA_DE = hs_de & vs_de;
assign VGA_R = (hs_de & vs_de)?VGA_R_reg:8'h0;
assign VGA_G = (hs_de & vs_de)?VGA_G_reg:8'h0;
assign VGA_B = (hs_de & vs_de)?VGA_B_reg:8'h0;
endmodule

```

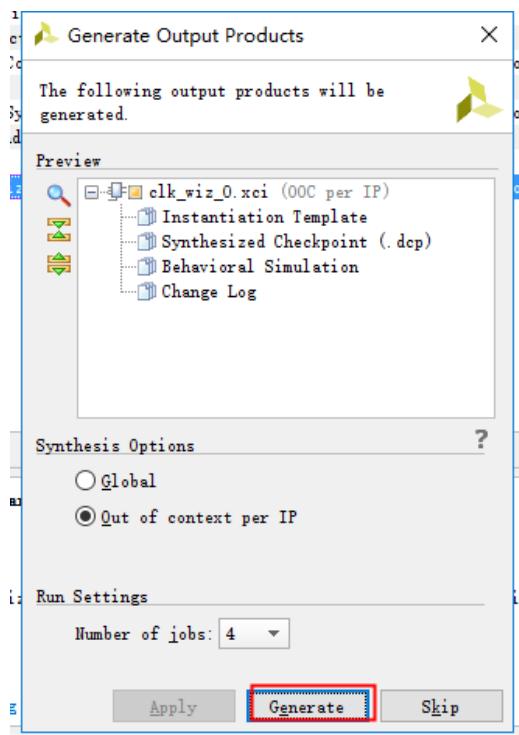
Step14: 单击 IP Catalog, 添加一个时钟管理器, 为系统提供时钟。



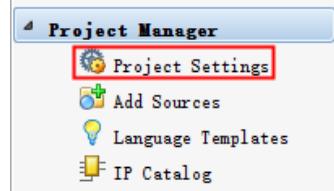
Step15: 在 output clocks 选项卡下如下图设置, 其余参数默认配置即可, 然后单击 OK。



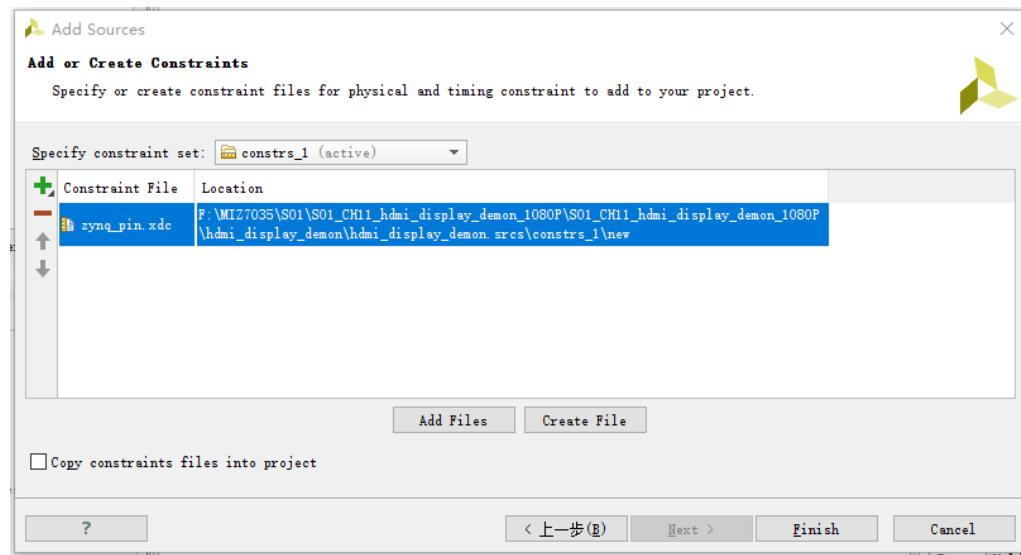
Step16: 单击 Generate。



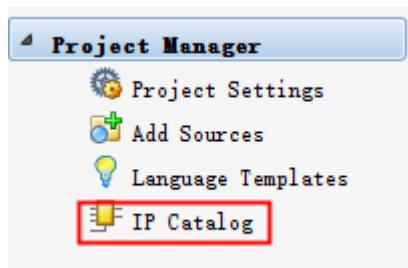
Step17: 单击 Project settings,对工程进行设置。



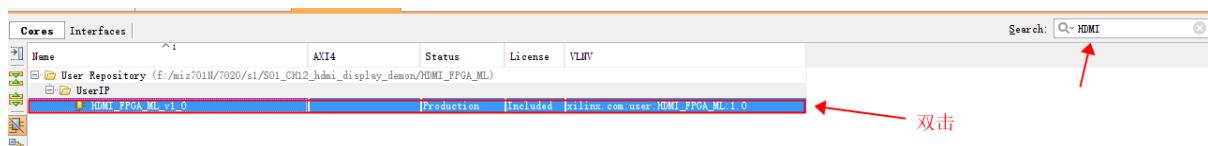
Step18: 选择 IP 选项，再选择 Repository Manager，单击+号将本章的 IP 核添加进来，最后单击 OK，完成工程的配置（IP 核在我们提供的源代码对应章节的文件包里面的 Miz_ip_lib 文件夹中找到）。



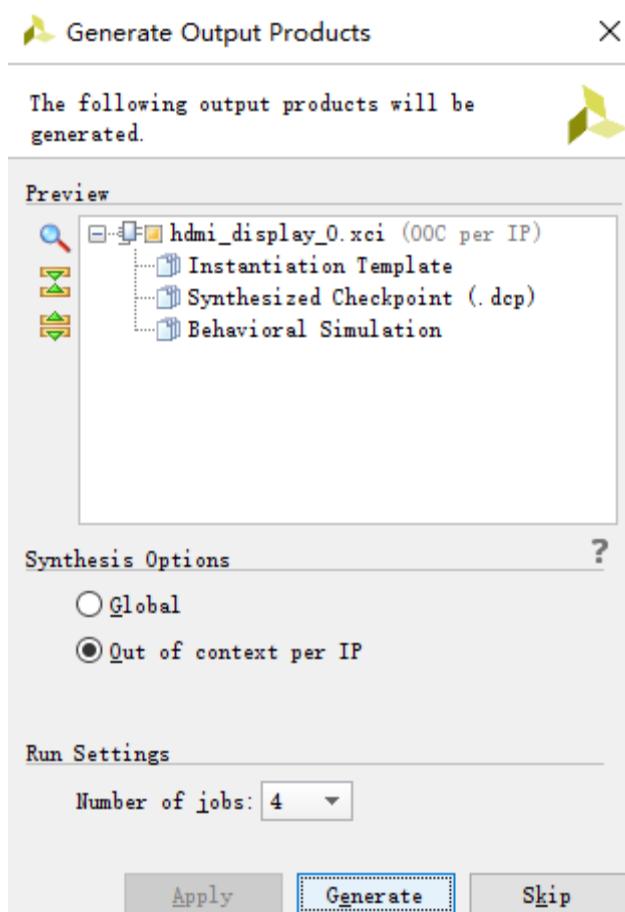
Step19: 单击  IP Catalog 添加一个 IP。



Step20: 输入 IP 的名字，双击之后单击 OK 不做任何改动，将其添加到工程当中。

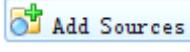


Step21: 单击 Generate。

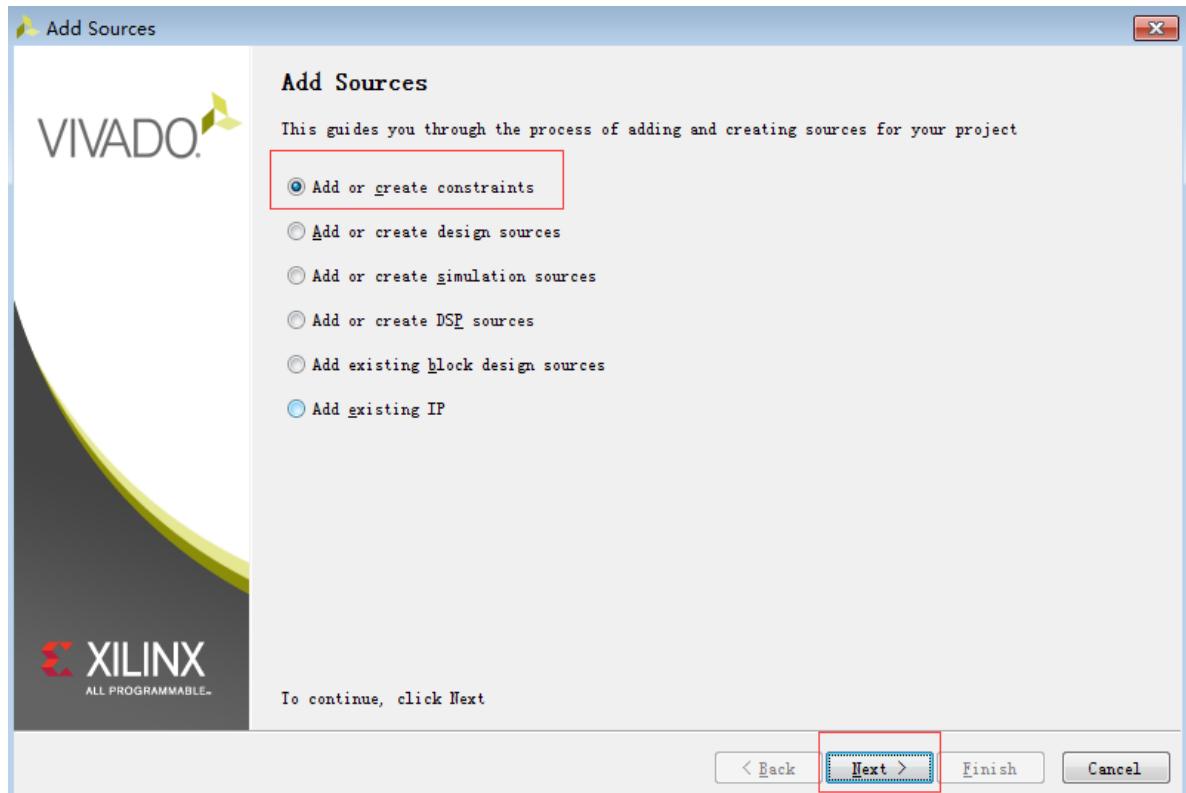


这样我们就编写好了代码下面还要添加管脚约束文件。

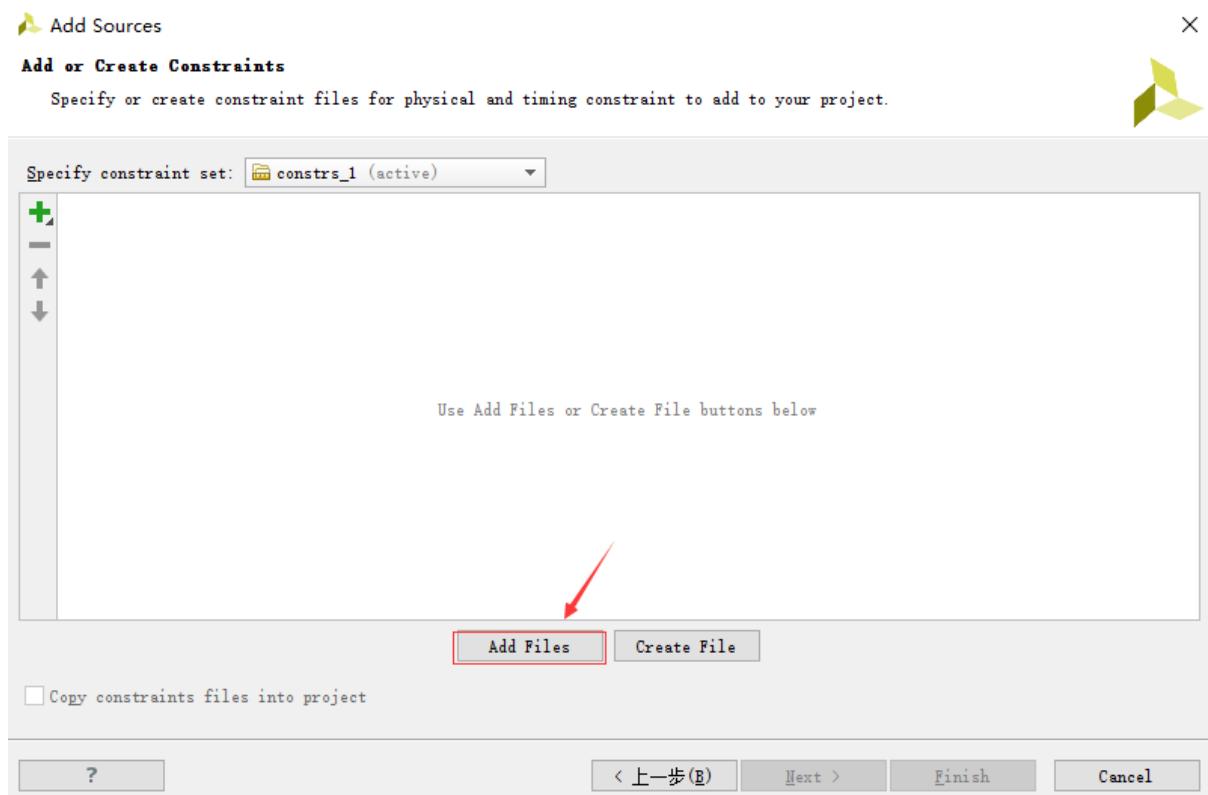
10.3 添加管脚约束文件

Step1: 单击  (和添加.v文件一样)

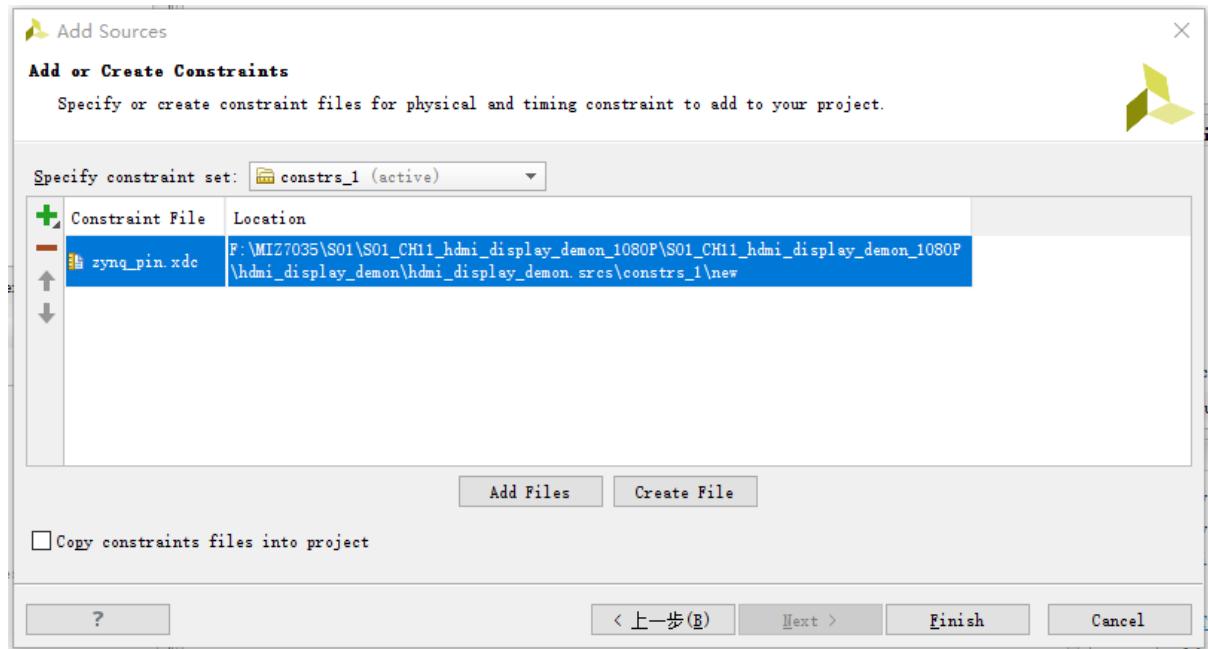
Step2: 选择 Add or create constraints 然后单击 NEXT



Step3: 单击 Add Files



Step4:在我们提供的源文件下的 DOC 文件夹下找到 XDC 文件夹:



Step5:选中 zynq_pin.xdc 文件，然后点击 OK。

Step6:点击 Finish 完成约束文件的添加

10.4 编译并且产生 bit 文件

Step1:单击综合

Step2:单击执行

Step3:单击产生 bit



10.5 下载程序

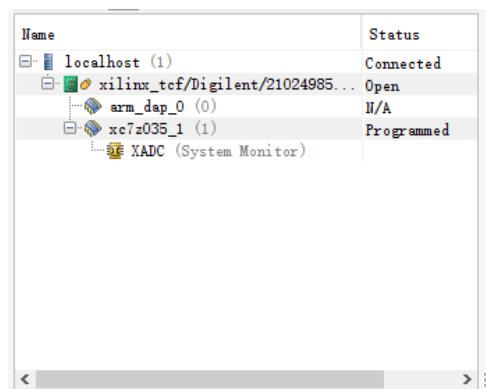
Step1:给开发板通电，并且连接下载器

Step2:单击 OpenTarget 然后单击 Auto Connect

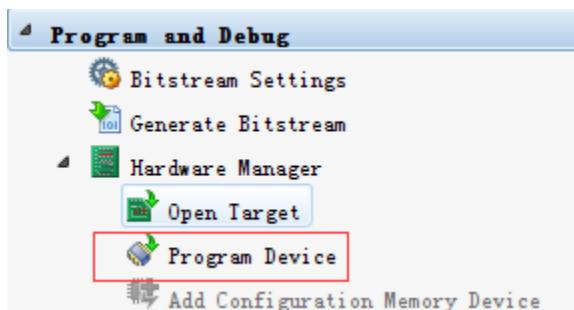


Step3:连接成功后如下图所示：

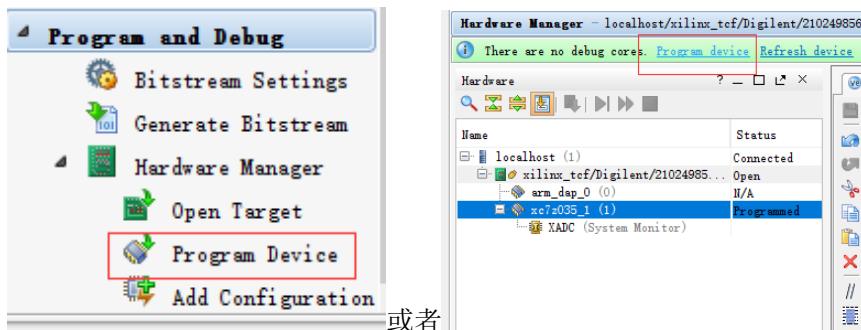
MIZ7035 如下图所示：



Step4:单击 Program Device

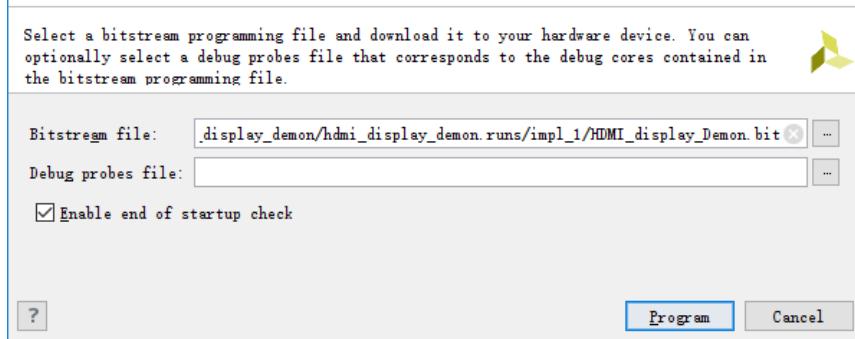


Step5:单击 Program Device 然后选择 XC7Z035。也可以从顶部单击 Program device

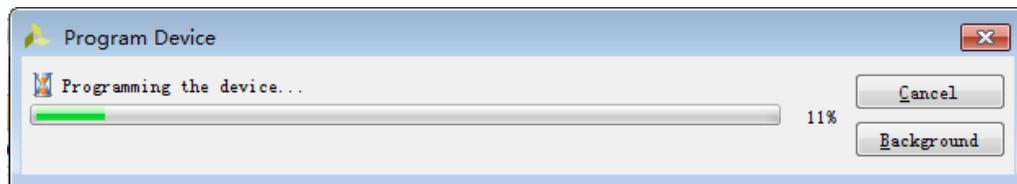


或者

Step6:弹出的对话框中有我们要下载的 Bit 文件

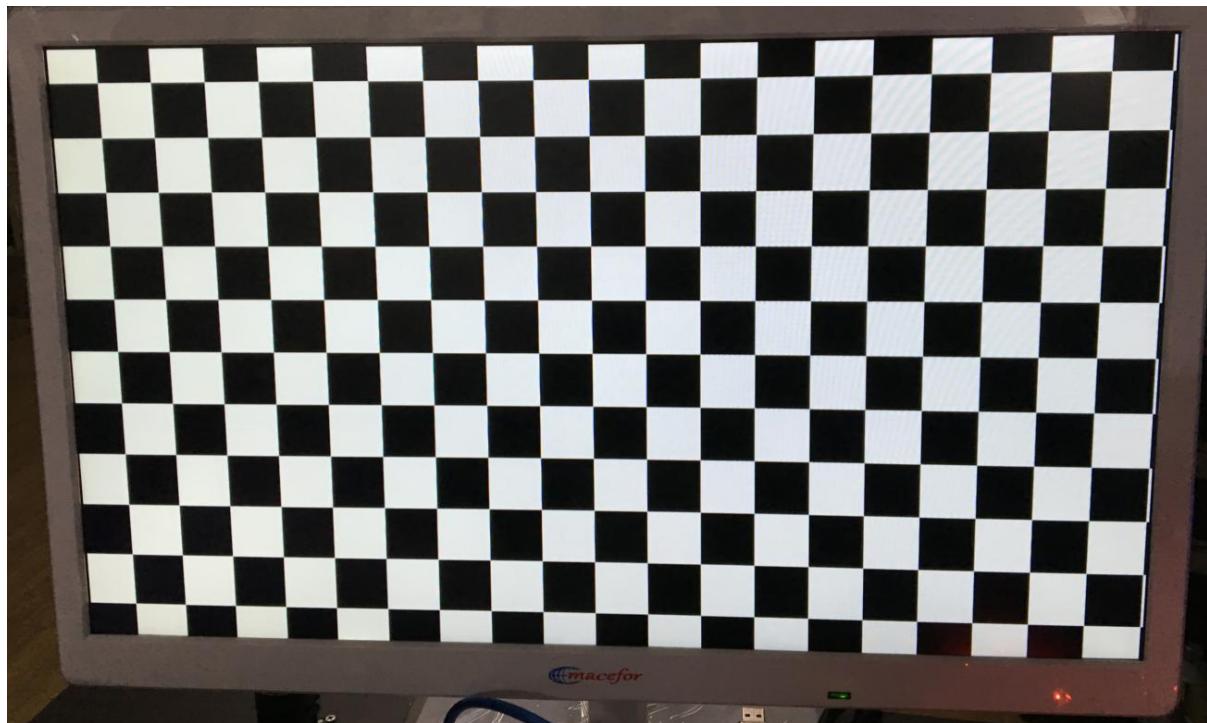


Step7: 下载过程



10.6 实验结果

下载过程下载完成后按中间的按键 VGA 会切换一次显示的图像，这些都是我们在程序部分定义的图像。



10.7 本章小结

本章详细讲解了如何创建 HDMI 接口的测试程序。大家对此消化了以后，可以尝试设计其他分辨率的测试程序

一个完美的结束
意味着一个新的开始！

www.osrc.cn

米联客
技术论坛
秀出你的风采！