
ALINX 黑金 ZYNQ 开发平台

配套教程

AX7350

2018/7/16 20:29:04



我们承诺本教程并非一劳永逸，固守不变的文档。我们会根据论坛上大家的反馈意见，以及实际的开发实践经验积累不断的修正和优化教程

文档修订记录:

版本	时间	描述
V1.01	2018/5/2	初始版本
V1.02	2018/5/16	修改部分修辞
V1.03	2018/7/16	修改部分错误

序

首先感谢大家购买芯驿电子科技（上海）有限公司出品的 ZYNQ 的开发板 AX7350！您对我们和我们产品的支持和信任,给我们增添了永往直前的信心和勇气。

“播下一粒种子，收获一片森林”，更是芯驿电子科技（上海）有限公司的美好愿望，同时我们会在黑金动力社区 <http://www.hejjin.org> 和大家一起讨论，一起学习，一起进步，一起成长。

目录

序	3
目录	4
第一章 开发板简介和检测	10
1.1 开发板简介	10
1.2 开发板检测	12
1.2.1 检测需要自备的工具	13
1.2.2 开发板线缆连接	15
1.2.3 开始测试	16
第二章 ZYNQ 简介	21
2.1 PS 和 PL 互联技术	21
2.2 ZYNQ 芯片开发流程的简介	27
2.3 学习 ZYNQ 要具备哪些技能	28
2.3.1 软件开发人员	28
2.3.2 逻辑开发人员	28
第三章 Vivado 开发环境	29
3.1 Vivado 软件介绍	29
3.2 Vivado 软件版本	29
3.3 Vivado 软件 Windows 下安装	30
第四章 PL 的 “Hello World”LED 实验	37
4.1 LED 硬件介绍	37
4.2 创建 Vivado 工程	37
4.3 创建 Verilog 文件点亮 LED	43
4.4 添加 XDC 约束文件约束管脚	48
4.5 编译工程	52
4.6 下载调试	54
第五章 HDMI 输出实验	58
5.1 硬件介绍	58
5.2 程序设计	58
5.3 添加 XDC 约束文件	60
5.4 下载调试	61
5.5 实验总结	62
第六章 可编程时钟 SI5338 实验	63
6.1 硬件介绍	63
6.2 程序设计	63

6.3 ClockBuilder Desktop 软件使用简介	63
6.4 使用 Python 转换寄存器配置文件	68
6.5 Vivado 工程建立	68
6.6 下载调试	69
6.7 实验总结	69
第六章 PL 端 DDR3 读写测试实验	70
7.1 硬件介绍	70
7.2 Vivado 工程建立	71
7.2.1 创建一个 PL 端 ddr3 测试工程	71
7.2.2 配置 ddr3 IP	72
7.2.3 添加其他测试代码	88
7.3 下载调试	88
7.4 实验总结	88
第七章 GTX 收发器误码率测试 IBERT 实验	89
8.1 硬件介绍	89
8.2 Vivado 工程建立	89
8.3 下载调试	93
8.4 实验总结	96
第八章 体验 ARM , 裸机输出 “Hello World”	97
9.1 硬件介绍	97
9.2 Vivado 工程建立	97
9.3 SDK 调试	108
9.4 实验总结	120
9.5 常见问题	120
9.5.1 通过 vivado 启动 SDK 后没有窗口弹出	120
第九章 PS 点亮 PL 的 LED 灯	122
10.1 Vivado 工程建立	122
10.2 XDC 文件约束 PL 管脚	130
10.3 SDK 程序编写	132
10.4 下载调试	135
10.5 实验总结	137
第十章 PS 定时器中断实验	138
11.1 Vivado 工程建立	138
11.2 SDK 程序编写	139
11.3 下载调试	143
11.4 实验总结	144
第十一章 PL 按键中断实验	145
12.1 Vivado 工程建立	145

12.2 下载调试	149
12.3 实验总结	155
第十三章 以太网实验 (LWIP)	156
13.1 Vivado 工程建立	156
13.1.1 PS 端的以太网配置	156
13.1.2 PL 端 AXI 以太网配置	159
13.1.3 添加约束文件	168
13.2 SDK 程序	169
13.2.1 LWIP 库修改	169
13.2.2 创建基于 LWIP 模板的 APP	175
13.3 下载调试	176
13.3.1 PL 端以太网测试	176
13.3.2 PS 端以太网测试	178
13.4 实验总结	180
第十四章 自定义 IP 实验	181
14.1 PWM 介绍	181
14.2 Vivado 工程建立	182
14.2.1 创建一个 vivado 工程	182
14.2.2 创建自定义 IP	183
14.2.3 添加自定义 IP 到工程	192
14.3 SDK 软件编写调试	195
14.4 实验总结	203
14.5 常见问题	203
14.5.1 如何知道 AXI IP 的基址	203
第十五章 使用 VDMA 驱动 HDMI 显示	205
15.1 Vivado 工程建立	205
15.2 SDK 软件编写调试	227
第十六章 固化程序	232
16.1 Vivado 工程建立	232
16.2 生成 FSBL	235
16.3 创建 BOOT 文件	239
16.4 SD 卡启动测试	243
16.5 QSPI 启动测试	244
16.6 Vivado 下烧写 QSPI	245
第十七章 PCIe ROOT 枚举测试	249
17.1 Vivado 工程建立	249
17.1.1 创建新工程	249
17.1.2 创建 block 设计	251

17.1.3 添加 AXI MM to PCIe bridge.....	260
17.1.4 添加复位模块	266
17.1.5 添加 DMA 模块.....	267
17.1.6 连接中断	268
17.1.7 添加 AXI 互联模块.....	269
17.1.8 连接时钟	273
17.1.9 连接复位信号	275
17.1.10 其他连接	276
17.1.11 端口设置	277
17.1.12 地址分配.....	280
17.1.13 创建 HDL 封装.....	282
17.1.14 添加 xdc 约束.....	282
17.1.15 关键步骤	284
17.2 SDK 下载调试.....	286
17.3 实验总结	296
第十八章 安装虚拟机和 Ubuntu 系统	297
18.1 虚拟机软件安装	297
18.2 Ubuntu 安装	298
18.2.1 安装系统	298
18.2.2 修改软件源服务器	305
18.2.3 设置 bash 为默认 sh.....	307
18.2.4 设置屏幕锁定时间	307
18.3 常见问题	308
18.3.1 虚拟机要求虚拟化支持	308
第十九章 Ubuntu 安装 Linux 版 Vivado 软件.....	310
19.1 安装 Linux 版 Vivado.....	310
19.2 权限设置	315
19.3 安装下载器驱动	315
19.4 测试 Vivado	315
19.5 常见问题	317
19.5.1 Linux 下载器下载时提示被占用	317
19.5.2 适合 ZYNQ 的交叉编译器	319
第二十章 Petalinux 工具安装	320
20.1 Petalinux 简介	320
20.2 安装必要的库	320
20.3 安装 Petalinux	321
第二十一章 NFS 服务软件安装	324
21.1 安装 NFS 服务	324

21.2 测试 NFS	326
第二十二章 使用 Petalinux 定制 Linux 系统	327
22.1 Vivado 工程	327
22.2 使用 Petalinux 建立工程	328
22.3 配置 Linux 内核	334
22.4 配置根文件系统	337
22.5 编译	338
22.6 修改 FSBL	339
22.7 生成 BOOT 文件	340
22.8 测试 Linux	341
第二十三章 使用 SDK 开发 Linux 程序	343
23.1 使用 SDK 建立 Linux 应用程序	343
23.2 通过 NFS 共享运行	346
23.3 通过 TCF-Agent 运行调试	348
23.4 TCF-Agent 问题	351
第二十四章 Linux 下 GPIO 实验	352
24.1 使用 SHELL 控制	352
24.2 使用 C 语言控制	353
24.3 实验总结	355
第二十五章 Petalinux 下的 HDMI 显示	357
25.1 Petalinux 配置	357
25.2 配置 Linux 内核	361
25.3 修改设备树	363
25.4 编译测试 Petalinux 工程	366
第二十六章 使用 Debian 8 桌面系统	369
26.1 Petalinux 配置	369
26.2 配置 Linux 内核	370
26.2.1 配置 USB WIFI 模块驱动	371
26.2.2 配置 USB 摄像头驱动	372
26.2.3 配置 PCIe NVMe SSD 驱动	372
26.3 编译测试 Petalinux 工程	373
26.4 制作 SD 卡文件系统	374
26.4.1 SD 卡修改分区	374
26.4.2 同步根文件系统到 SD 卡 EXT4 分区	379
第二十七章 PCIe SSD 应用	382
27.1 查看 PCI 设备	382
27.2 格式化 SSD	384
27.3 SSD 测速	385

第二十八章 QSPI 和 EMMC 启动 Linux	388
28.1 复制 Petalinux 工程	388
28.2 配置编译 Petalinux	389
28.3 如何烧写 EMMC	396
28.3.1 格式化并挂载 EMMC	396
28.3.2 复制文件 emmc	398
28.4 使用批处理文件快速烧写 QSPI.....	399
28.5 测试 QSPI 和 EMMC 启动.....	401

第一章 开发板简介和检测

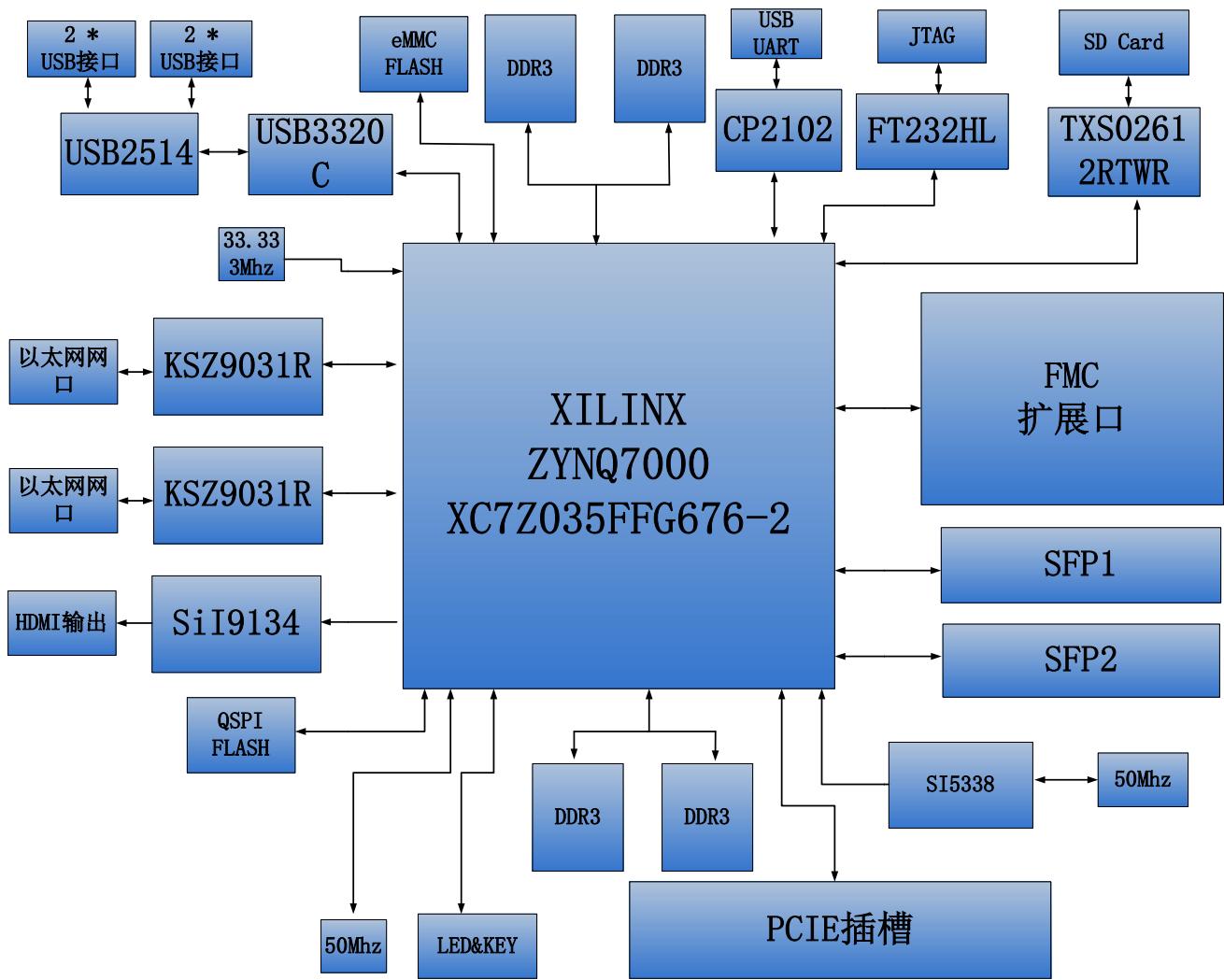
1.1 开发板简介

在这里，对这款 AX7350 ZYNQ 开发平台进行简单的功能介绍。

开发板主要由 ZYNQ7350 主芯片，4 个 DDR3，1 片 eMMC，1 个 QSPI FLASH 和一些外设接口组成。ZYNQ7350 采用 Xilinx 公司的 Zynq7000 系列的芯片，型号为 XC7Z035-2FFG676。ZYNQ7035 芯片可分成处理器系统部分 Processor System(PS)和可编程逻辑部分 Programmable Logic(PL)，在 ZYNQ7350 芯片的 PS 端和 PL 端分别挂了 2 片 DDR3，每片 DDR3 容量高达 512M 字节，使得 ARM 系统和 FPGA 系统能独立处理和存储的数据的功能。PS 端的 8GB eMMC FLASH 存储芯片和 256Mb 的 QSPI FLASH 用来静态存储 ZYNQ 的操作系统、文件系统及用户数据。

AX7350 开发板扩展了丰富的外围接口，其中包含 1 个 PCIe4 插槽、2 路光纤接口、2 路千兆以太网接口、4 路 USB2.0 HOST 接口、1 路 HDMI 输出接口，1 路 UART 串口接口、1 路 SD 卡接口、1 个 FMC 扩展接口和一些按键 LED。

下图为整个开发系统的结构示意图：



通过这个示意图，我们可以看到，我们这个开发平台所能含有的接口和功能。

- Xilinx ARM+FPGA 芯片 Zynq-7000 XC7Z035-2FFG676。
- DDR3

带有四片大容量的 512M 字节（共 2GB）高速 DDR3 SDRAM。其中两片挂载在 PS 端，可作为 ZYNQ 芯片数据的缓存，也可以作为操作系统运行的内存；另外两片挂在 PL 端，可作为 FPGA 的数据存储，图像分析缓存，数据处理。

- eMMC

PS 端挂载一片 8GB eMMC FLASH 存储芯片，用户存储操作系统文件或者其他用户数据。

- QSPI FLASH

一片 256Mbit 的 QSPI FLASH 存储芯片，可用作 ZYNQ 芯片的 Uboot 文件，系统文件和用户数据的存储；

- PCIe x4 接口

一路标准的 PCIe x8 的主机插槽用于 PCIe x4 通信，可用于连接 PCIe x4, x2, x1 的 PCIe 板卡，实现 PCIe 数据通信。支持 PCI Express 2.0 标准，单通道通信速率可高达 5GBaud。

- 2 路 SFP 光纤接口

ZYNQ 的 GTX 收发器的 2 路高速收发器连接到 2 个光模块的发送和接收，实现 2 路高速的

光纤通信接口。每路的光纤数据通信接收和发送的速度高达 10Gb/s。

- 千兆以太网接口

2 路 10/100M/1000M 以太网 RJ45 接口 ,用于和电脑或其它网络设备进行以太网数据交换。

网络接口芯片采用 Micrel 公司的 KSZ9031 工业级 GPHY 芯片 ,1 路以太网连接到 ZYNQ 芯片的 PS 端 ,1 路以太网连接到 ZYNQ 芯片的 PL 端。

- HDMI 视频输出

1 路 HDMI 视频输出接口 ,我们选用了 Silion Image 公司的 SIL9134 HDMI 编码芯片 ,最高支持 1080P@60Hz 输出 ,支持 3D 输出。

- USB2.0 HOST 接口

通过 USB Hub 芯片扩展 4 路 USB HOST 接口 ,用于连接外部的 USB 从设备 ,比如连接鼠标 ,键盘 ,U 盘等等。USB 接口采用扁型 USB 接口(USB Type A)。

- USB Uart 接口

2 路 Uart 转 USB 接口 ,用于和电脑通信 ,方便用户调试。1 路在核心板上 ,核心板独立工作是使用 ,1 路在底板上 , 整板调试时使用。串口芯片采用 Silicon Labs CP2102GM 的 USB-UAR 芯片 ,USB 接口采用 MINI USB 接口。

- Micro SD 卡座

1 路 Micro SD 卡座 ,用于存储操作系统镜像和文件系统。

- FMC 扩展口

1 个标准的 FMC LPC 的扩展口 ,可以外接 XILINX 或者我们黑金的各种 FMC 模块 (HDMI 输入输出模块 ,双目摄像头模块 ,高速 AD 模块等等) 。FMC 扩展口包含 33 对差分 IO 信号和一路高速 GTX 收发信号。

- USB JTAG 口

一路 USB JTAG 口 ,通过 USB 线及板载的 JTAG 电路对 ZYNQ 系统进行调试和下载

- 时钟

板载一个 33.333Mhz 的有源晶振 ,给 PS 系统提供稳定的时钟源 ,一个 50MHz 的有源晶振 ,为 PL 逻辑提供额外的时钟 ;另外板上有一个可编程的时钟芯片给 GTX 提供时钟源 ,为 PCIE ,光纤和 DDR 工作提供参考时钟。

- LED 灯

9 个发光二极管 LED, 1 个电源指示灯 ;1 个 DONE 配置指示灯 ;2 个串口通信指示灯 ,1 个 PS 控制 LED 灯 ,4 个 PL 控制指示灯。

- 按键

6 个按键 ,1 个复位按键 ,1 个 PS 用户按键 ,4 个 PL 用户按键。

1.2 开发板检测

拿到开发板 ,大部分人都想立即体验一下 ,看看开发板是否正常工作 ,下面我们介绍如何对开发板进行一个简单的检测。

1.2.1 检测需要自备的工具

- 1) 电脑



- 2) 支持 HDMI 的显示器，要求分辨率不小于 1920x1080



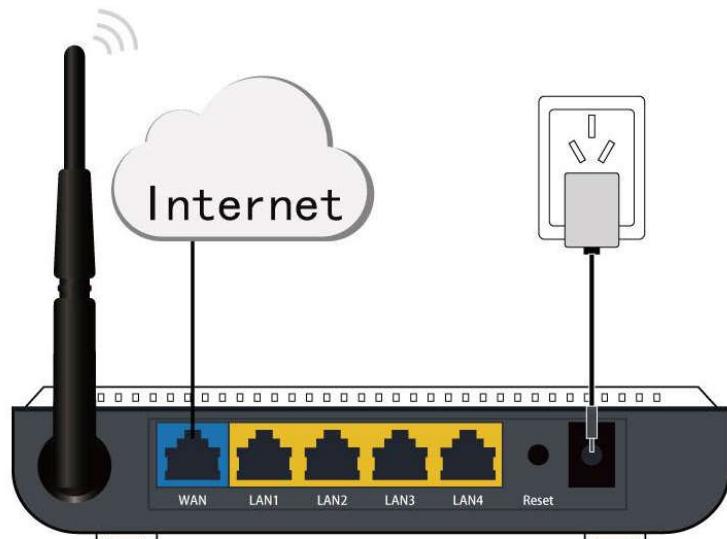
- 3) HDMI 线缆 2 条



4) USB 鼠标键盘



5) 路由器，为了测试网络，最好能连接互联网，支持 DHCP。

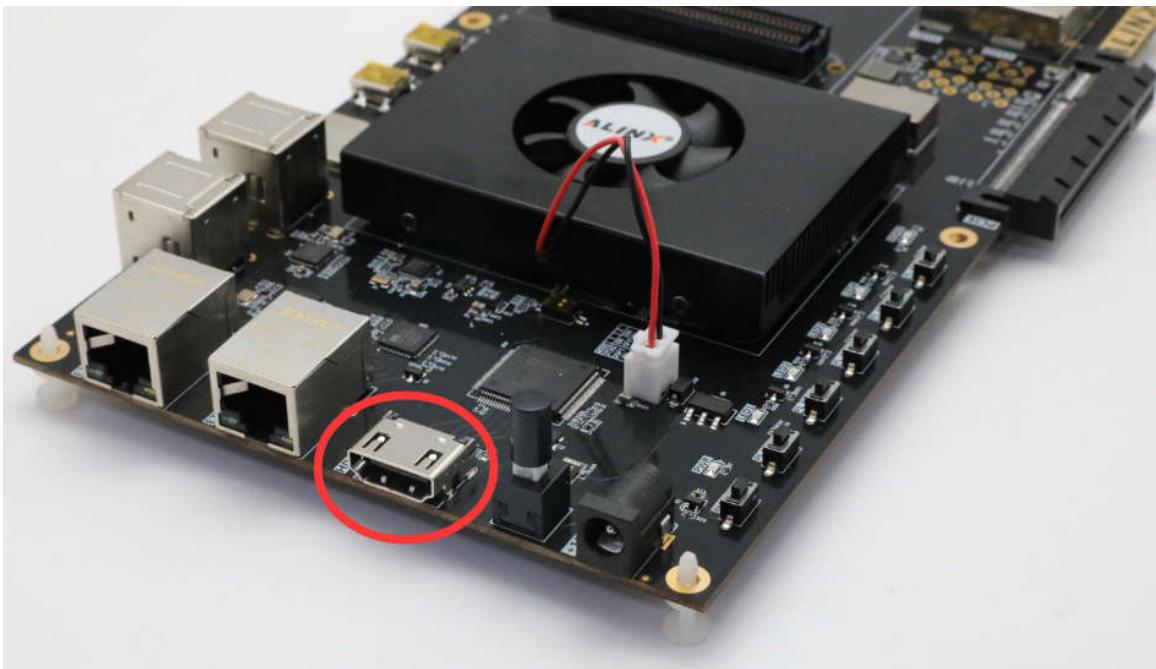


6) 网线



1.2.2 开发板线缆连接

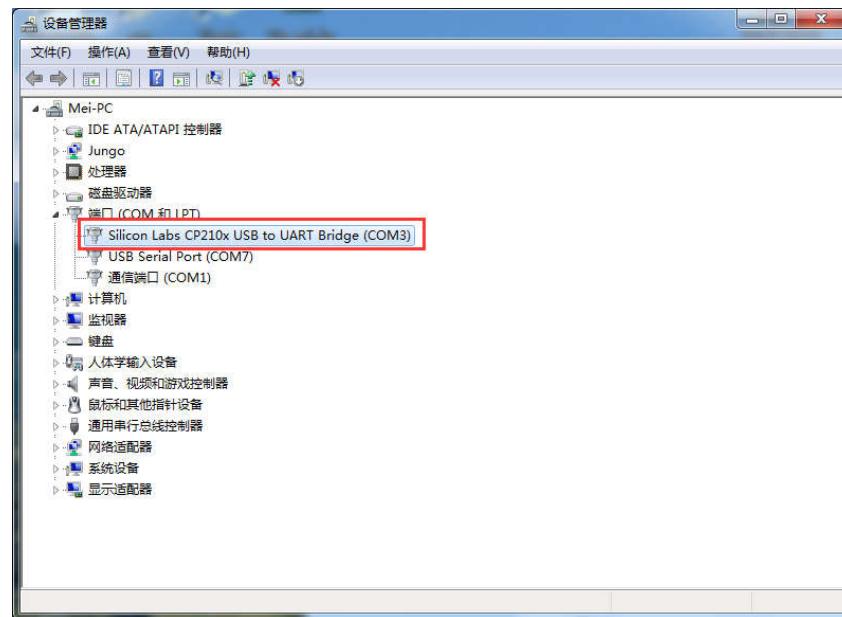
1) 连接 HDMI 显示器



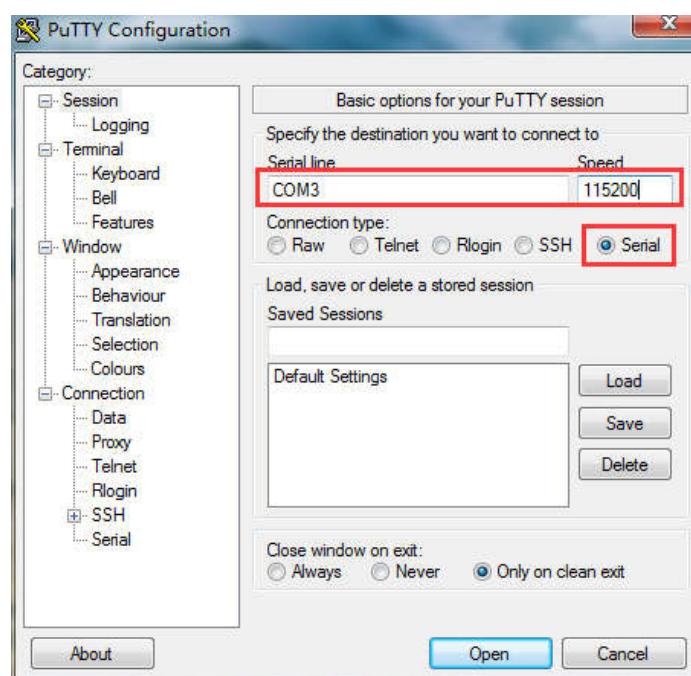
- 2) 连接 usb 转串口 , 主要用于看 ARM 打印出的一些信息
- 3) 连接 ARM 端网口 (ETH1) 到路由器
- 4) 连接电源

1.2.3 开始测试

- 1) 测试前我们需要安装 USB 转串口的驱动软件 (软件/CP210x_Windows_Drivers.zip) , 不然无法做串口通信测试。驱动安装好以后 , 用红色 USB 线连接电脑 USB 口和开发板上的 UART 口(J1)进行连接 , 然后打开电脑的设备管理器 , 设备管理器能够找到串口设备 CP210x , 我机器上映射的是 COM3 。如果不能成功安装驱动 , 可以尝试使用驱动精灵安装。



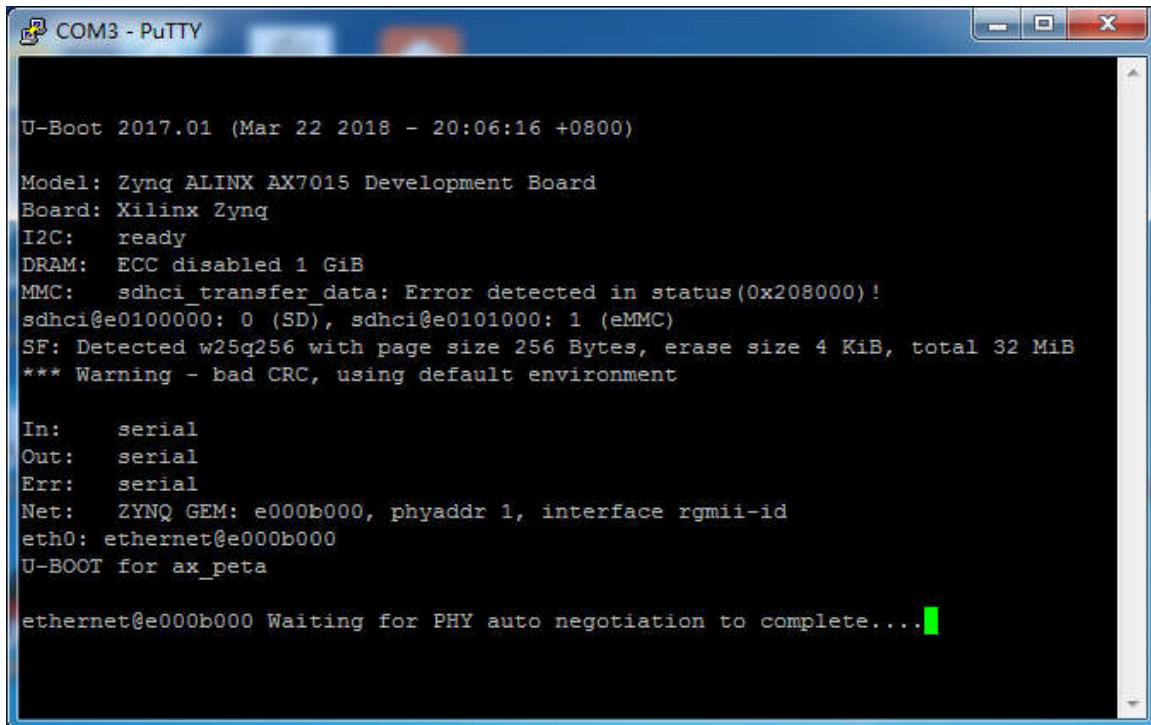
- 2) 终端工具有很多，例如 putty , teraterm, Windows 自带终端工具 , SecureCRT 等等，众多终端工具中，比较下来，还是 putty 最好用，资料（软件/ putty.exe ）为大家准备了绿色免安装的 putty 软件。
- 3) 选择 Serial , Serial line 填写 COM3 , Speed 填写 115200 , COM3 串口号根据设备管理器里显示的填写，点击 “Open”



- 4) 确定开发板启动模式是否为 SD 启动模式（默认出厂时开发板的 SD 卡插槽里有卡，启动模式默认也是 SD 卡），拨动拨码开关时用一个带尖端的工具，例如，镊子，取卡针等，轻轻

拨动。

- 5) 打开开发板上的电源开关，PuTTY 工具窗口会显示 u-boot 和 Linux 系统的启动信息。



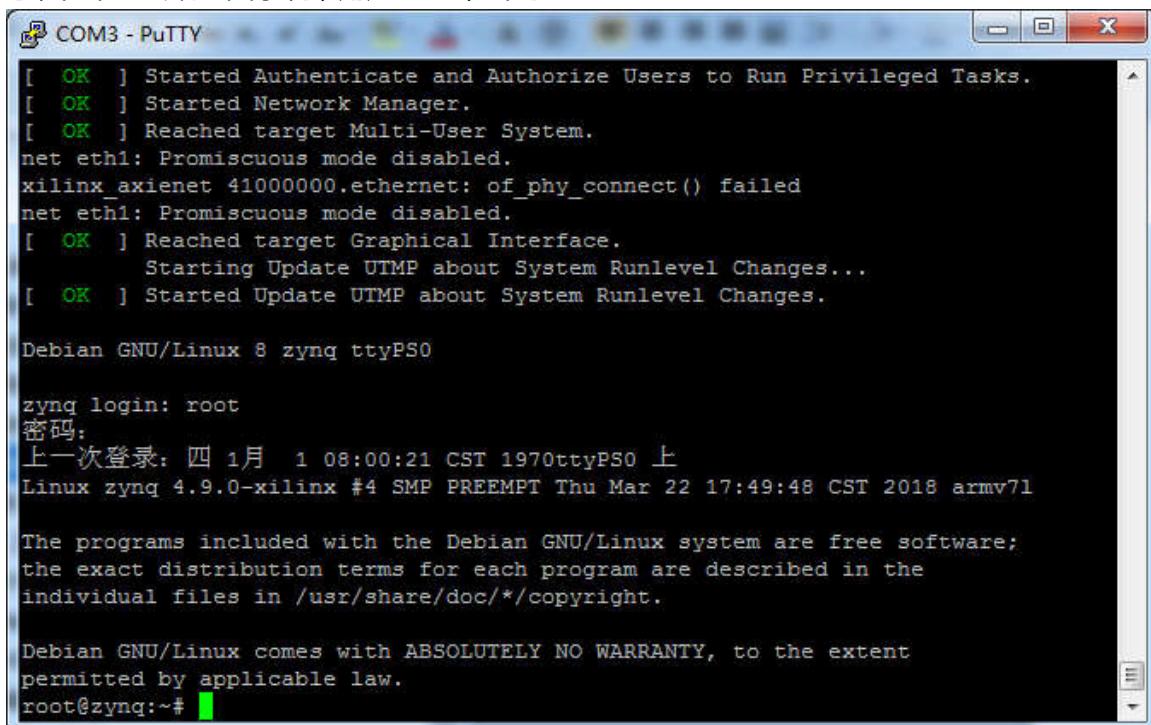
```
U-Boot 2017.01 (Mar 22 2018 - 20:06:16 +0800)

Model: Zynq ALINX AX7015 Development Board
Board: Xilinx Zynq
I2C:    ready
DRAM:  ECC disabled 1 GiB
MMC:   sdhci transfer data: Error detected in status(0x208000) !
sdhci@e0100000: 0 (SD), sdhci@e0101000: 1 (eMMC)
SF: Detected w25q256 with page size 256 Bytes, erase size 4 KiB, total 32 MiB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   ZYNQ GEM: e000b000, phyaddr 1, interface rgmii-id
eth0:  ethernet@e000b000
U-BOOT for ax_peta

ethernet@e000b000 Waiting for PHY auto negotiation to complete....
```

- 6) 可以在串口终端登陆系统，用户: root，密码: root



```
[  OK  ] Started Authenticate and Authorize Users to Run Privileged Tasks.
[  OK  ] Started Network Manager.
[  OK  ] Reached target Multi-User System.
net eth1: Promiscuous mode disabled.
xilinx_axienet 41000000.ethernet: of_phy_connect() failed
net eth1: Promiscuous mode disabled.
[  OK  ] Reached target Graphical Interface.
          Starting Update UTMP about System Runlevel Changes...
[  OK  ] Started Update UTMP about System Runlevel Changes.

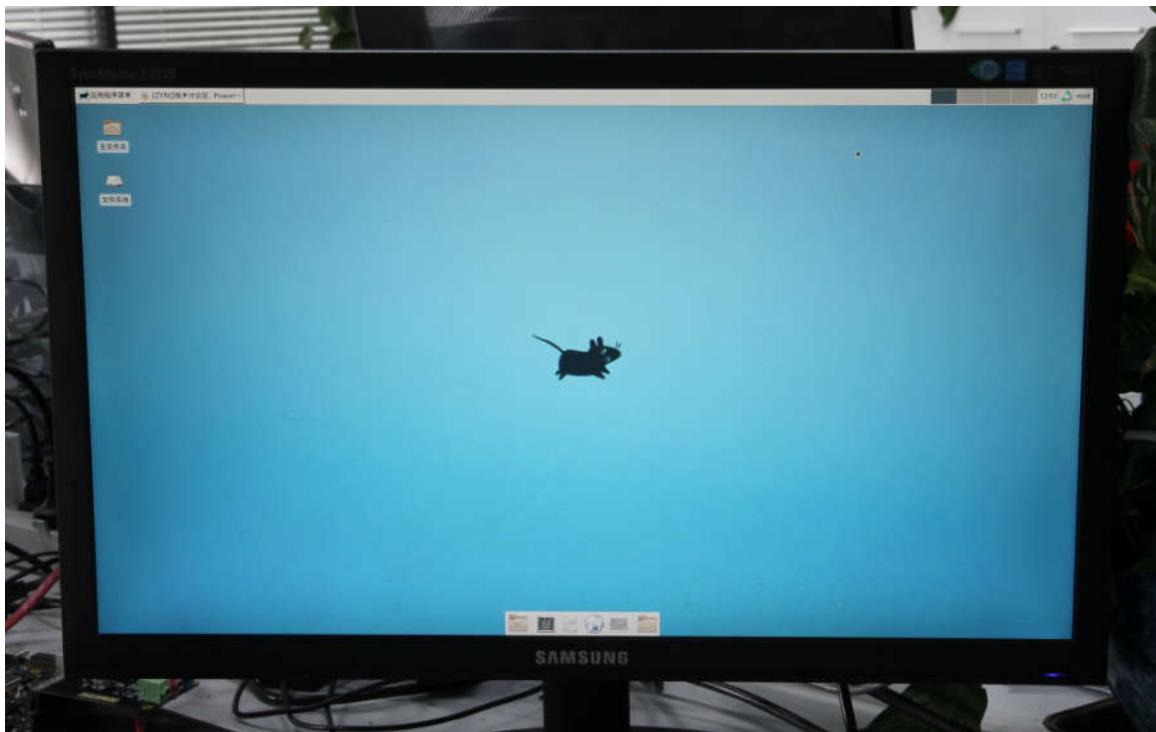
Debian GNU/Linux 8 zynq ttyPS0

zyng login: root
密码:
上一次登录: 四 1月 1 08:00:21 CST 1970ttyPS0 上
Linux zynq 4.9.0-xilinx #4 SMP PREEMPT Thu Mar 22 17:49:48 CST 2018 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@zynq:~#
```

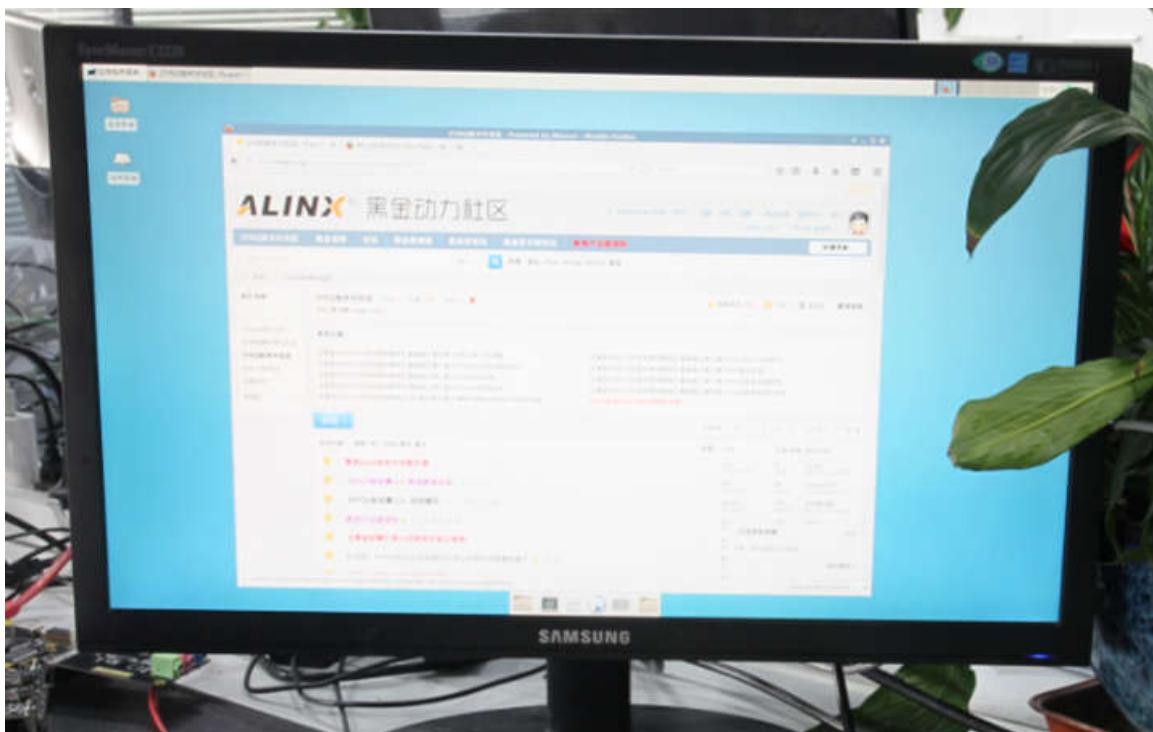
- 7) 启动完成后连接开发板 HDMI 显示器会显示 Debian 的桌面。



- 8) 这时可以使用连接到鼠标和键盘来操作了，用鼠标双击 Web 浏览器，启动浏览器时间较长，请耐心等待。



- 9) 地址栏输入网址，我们这里输入黑金动力社区的网址。正常打开，我们的开发板已经能正常上网了。

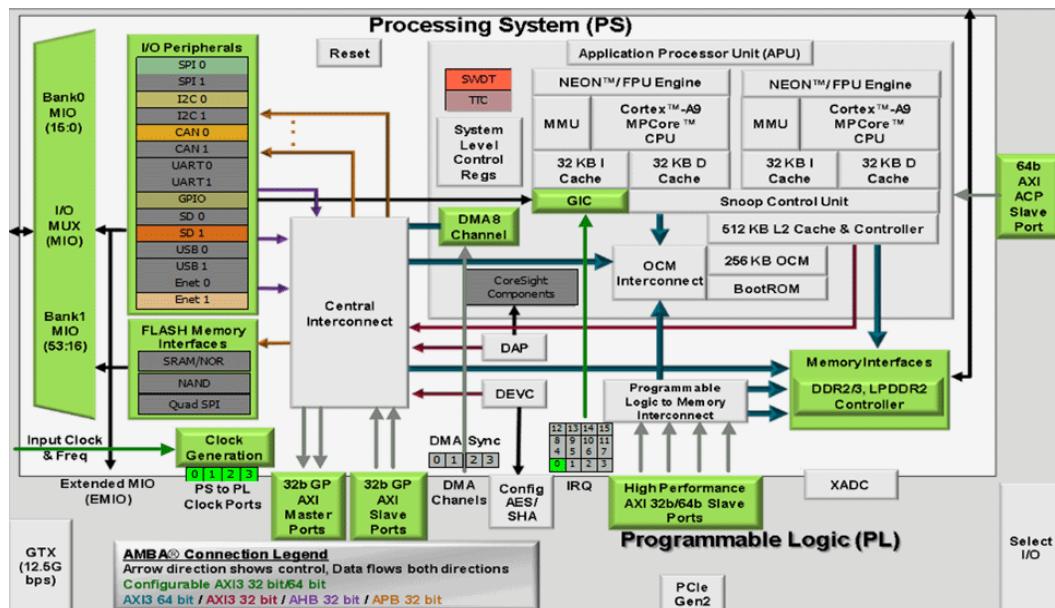


10) 开发板的简单检测到此结束。

第二章 ZYNQ 简介

Zynq 系列的亮点在于 FPGA 里包含了完整的 ARM 处理子系统 (PS), 每一颗 Zynq 系列的处理器都包含了 Cortex-A9 处理器，整个处理器的搭建都以处理器为中心，而且处理器子系统中集成了内存控制器和大量的外设，使 Cortex-A9 的核在 Zynq-7000 中完全独立于可编程逻辑单元，也就是说如果暂时没有用到可编程逻辑单元部分 (PL)，ARM 处理器的子系统也可以独立工作，这与以前的 FPGA 有本质区别，其是以处理器为中心的。

Zynq 就是两大功能块，PS 部分和 PL 部分，说白了，就是 ARM 的 SOC 部分，和 FPGA 部分。其中，PS 集成了两个 ARM Cortex™-A9 处理器，AMBA® 互连，内部存储器，外部存储器接口和外设。这些外设主要包括 USB 总线接口，以太网接口，SD/SDIO 接口，I2C 总线接口，CAN 总线接口，UART 接口，GPIO 等。



ZYNQ 芯片的总体框图

PS: 处理系统 (Processing System)，就是与 FPGA 无关的 ARM 的 SOC 的部分。

PL: 可编程逻辑 (Progarmmmable Logic)，就是 FPGA 部分。

2.1 PS 和 PL 互联技术

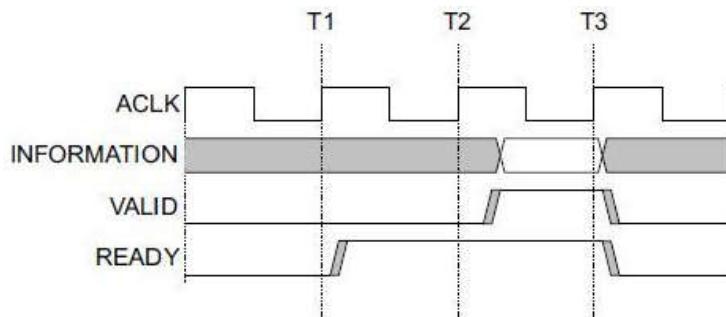
ZYNQ 作为首款将高性能 ARM Cortex-A9 系列处理器与高性能 FPGA 在单芯片内紧密结合的产品，为了实现 ARM 处理器和 FPGA 之间的高速通信和数据交互，发挥 ARM 处理器和 FPGA 的性能优势，需要设计高效的片内高性能处理器与 FPGA 之间的互连通路。因此，如何设计高效的 PL 和 PS 数据互通通路是 ZYNQ 芯片设计的重中之重，也是产品设计的成败关键之一。本节，我们就将主要介绍 PS 和 PL 的连接，让用户了解 PS 和 PL 之间连接的技术。

其实，在具体设计中我们往往不需要在连接这个地方做太多工作，我们加入 IP 核以后，系

统会自动使用 AXI 接口将我们的 IP 核与处理器连接起来，我们只需要再做一点补充就可以了。

AXI 全称 Advanced eXtensible Interface 是 Xilinx 从 6 系列的 FPGA 开始引入的一个接口协议，主要描述了主设备和从设备之间的数据传输方式。在 ZYNQ 中继续使用，版本是 AXI4，所以我们经常会看到 AXI4.0，ZYNQ 内部设备都有 AXI 接口。其实 AXI 就是 ARM 公司提出的 AMBA (Advanced Microcontroller Bus Architecture) 的一个部分，是一种高性能、高带宽、低延迟的片内总线，也用来替代以前的 AHB 和 APB 总线。第一个版本的 AXI (AXI3) 包含在 2003 年发布的 AMBA3.0 中，AXI 的第二个版本 AXI (AXI4) 包含在 2010 年发布的 AMBA 4.0 之中。

AXI 协议主要描述了主设备和从设备之间的数据传输方式，主设备和从设备之间通过握手信号建立连接。当从设备准备好接收数据时，会发出 READY 信号。当主设备的数据准备好时，会发出和维持 VALID 信号，表示数据有效。数据只有在 VALID 和 READY 信号都有效的时候才开始传输。当这两个信号持续保持有效，主设备会继续传输下一个数据。主设备可以撤销 VALID 信号，或者从设备撤销 READY 信号终止传输。AXI 的协议如图，T2 时，从设备的 READY 信号有效，T3 时主设备的 VALID 信号有效，数据传输开始。



AXI 握手时序图

在 ZYNQ 中，支持 AXI-Lite，AXI4 和 AXI-Stream 三种总线，通过表 5-1, 我们可以看到这三中 AXI 接口的特性。

接口协议	特性	应用场合
AXI4-Lite	地址/单数据传输	低速外设或控制
AXI4	地址/突发数据传输	地址的批量传输
AXI4-Stream	仅传输数据，突发传输	数据流和媒体流传输

AXI-Lite：

具有轻量级，结构简单的特点，适合小批量数据、简单控制场合。不支持批量传输，读写时一次只能读写一个字 (32bit)。主要用于访问一些低速外设和外设的控制。

AXI4：

接口和 AXI-Lite 差不多，只是增加了一项功能就是批量传输，可以连续对一片地址进行一次性读写。也就是说具有数据读写的 burst 功能。

上面两种均采用内存映射控制方式，即 ARM 将用户自定义 IP 编入某一地址进行访问，读

写时就像在读写自己的片内 RAM，编程也很方便，开发难度较低。代价就是资源占用过多，需要额外的读地址线、写地址线、读数据线、写数据线、写应答线这些信号线。

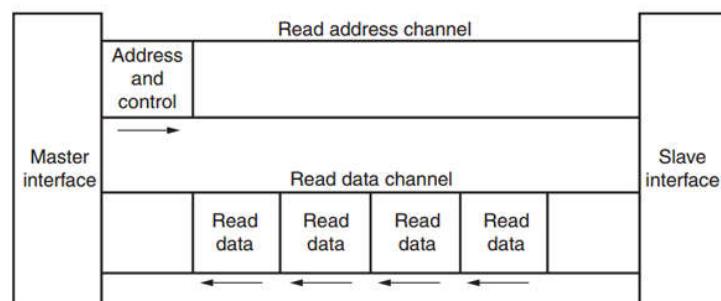
AXI-Stream：

这是一种连续流接口，不需要地址线（很像 FIFO，一直读或一直写就行）。对于这类 IP，ARM 不能通过上面的内存映射方式控制（FIFO 根本没有地址的概念），必须有一个转换装置，例如 AXI-DMA 模块来实现内存映射到流式接口的转换。AXI-Stream 适用的场合有很多：视频流处理；通信协议转换；数字信号处理；无线通信等。其本质都是针对数值流构建的数据通路，从信源（例如 ARM 内存、DMA、无线接收前端等）到信宿（例如 HDMI 显示器、高速 AD 音频输出，等）构建起连续的数据流。这种接口适合做实时信号处理。

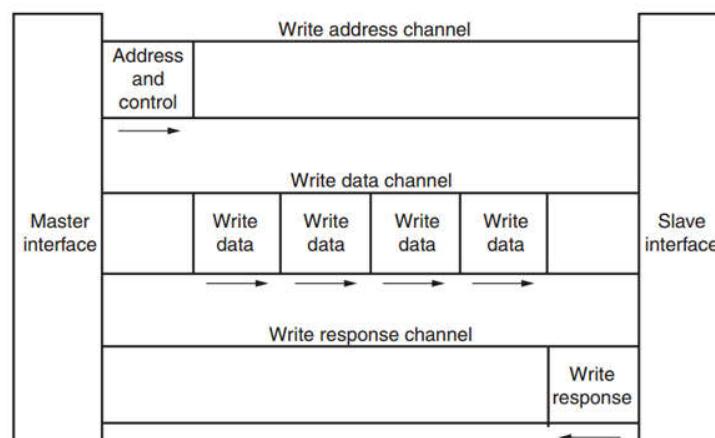
AXI4 和 AXI4-Lite 接口包含 5 个不同的通道：

- ✓ Read Address Channel
- ✓ Write Address Channel
- ✓ Read Data Channel
- ✓ Write Data Channel
- ✓ Write Response Channel

其中每个通道都是一个独立的 AXI 握手协议。下面两个图分别显示了读和写的模型：



AXI 读数据通道



AXI 写数据通道

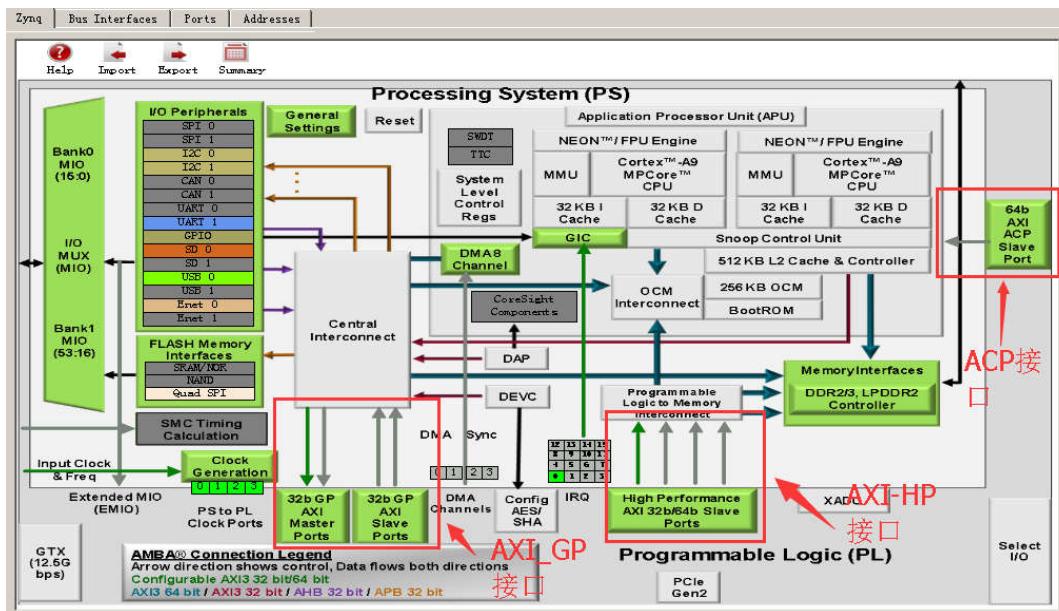
在 ZYNQ 芯片内部用硬件实现了 AXI 总线协议，包括 9 个物理接口，分别为 AXI-GP0~AXI-GP3，AXI-HP0~AXI-HP3，AXI-ACP 接口。

AXI_ACP 接口，是 ARM 多核架构下定义的一种接口，中文翻译为加速器一致性端口，

用来管理 DMA 之类的不带缓存的 AXI 外设，PS 端是 Slave 接口。

AXI_HP 接口，是高性能/带宽的 AXI3.0 标准的接口，总共有四个，PL 模块作为主设备连接。主要用于 PL 访问 PS 上的存储器（DDR 和 On-Chip RAM）

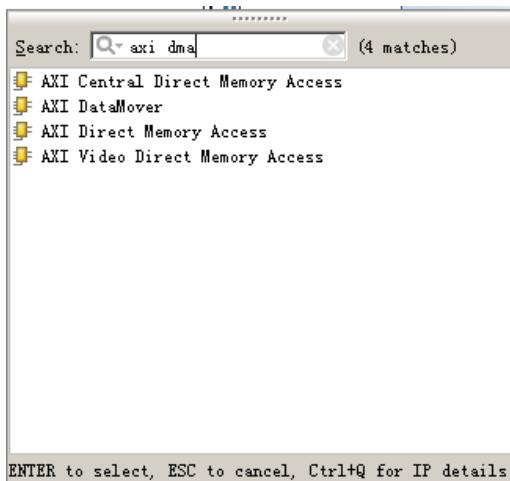
AXI_GP 接口，是通用的 AXI 接口，总共有四个，包括两个 32 位主设备接口和两个 32 位从设备接口。



可以看到，只有两个 AXI-GP 是 Master Port，即主机接口，其余 7 个口都是 Slave Port（从机接口）。主机接口具有发起读写的权限，ARM 可以利用两个 AXI-GP 主机接口主动访问 PL 逻辑，其实就是把 PL 映射到某个地址，读写 PL 寄存器如同在读写自己的存储器。其余从机接口就属于被动接口，接受来自 PL 的读写，逆来顺受。

另外这 9 个 AXI 接口性能也是不同的。GP 接口是 32 位的低性能接口，理论带宽 600MB/s，而 HP 和 ACP 接口为 64 位高性能接口，理论带宽 1200MB/s。有人会问，为什么高性能接口不做成主机接口呢？这样可以由 ARM 发起高速数据传输。答案是高性能接口根本不需要 ARM CPU 来负责数据搬移，真正的搬运工是位于 PL 中的 DMA 控制器。

位于 PS 端的 ARM 直接有硬件支持 AXI 接口，而 PL 则需要使用逻辑实现相应的 AXI 协议。Xilinx 在 Vivado 开发环境里提供现成 IP 如 AXI-DMA，AXI-GPIO，AXI-Dataover, AXI-Stream 都实现了相应的接口，使用时直接从 Vivado 的 IP 列表中添加即可实现相应功能。下图为 Vivado 下的各种 DMA IP：



下面为几个常用的 AXI 接口 IP 的功能介绍：

AXI-DMA：实现从 PS 内存到 PL 高速传输高速通道 AXI-HP<---->AXI-Stream 的转换

AXI-FIFO-MM2S：实现从 PS 内存到 PL 通用传输通道 AXI-GP<---->AXI-Stream 的转换

AXI-Datamover：实现从 PS 内存到 PL 高速传输高速通道 AXI-HP<---->AXI-Stream 的转换，只不过这次是完全由 PL 控制的，PS 是完全被动的。

AXI-VDMA：实现从 PS 内存到 PL 高速传输高速通道 AXI-HP<---->AXI-Stream 的转换，只不过是专门针对视频、图像等二维数据的。

AXI-CDMA：这个是由 PL 完成的将数据从内存的一个位置搬到另一个位置，无需 CPU 来插手。

关于如何使用这些 IP，我们会在后面的章节中举例讲到。有时，用户需要开发自己定义的 IP 同 PS 进行通信，这时可以利用向导生成对应的 IP。用户自定义 IP 核可以拥有 AXI-Lite, AXI4, AXI-Stream, PLB 和 FSL 这些接口。后两种由于 ARM 这一端不支持，所以不用。

有了上面的这些官方 IP 和向导生成的自定义 IP，用户其实不需要对 AXI 时序了解太多（除非确实遇到问题），因为 Xilinx 已经将和 AXI 时序有关的细节都封装起来，用户只需要关注自己的逻辑实现即可。

AXI 协议严格的讲是一个点对点的主从接口协议，当多个外设需要互相交互数据时，我们需要加入一个 AXI Interconnect 模块，也就是 AXI 互联矩阵，作用是提供将一个或多个 AXI 主设备连接到一个或多个 AXI 从设备的一种交换机制（有点类似于交换机里面的交换矩阵）。

这个 AXI Interconnect IP 核最多可以支持 16 个主设备、16 个从设备，如果需要更多的接口，可以多加入几个 IP 核。

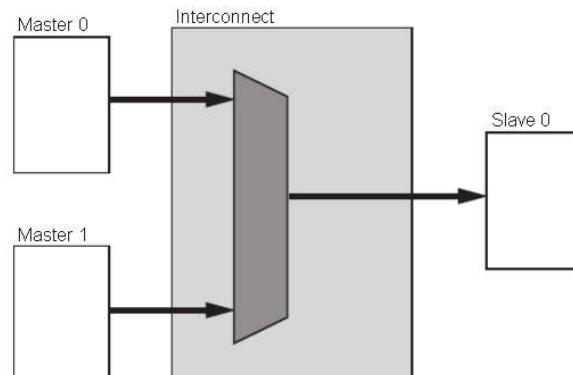
AXI Interconnect 基本连接模式有以下几种：

N-to-1 Interconnect

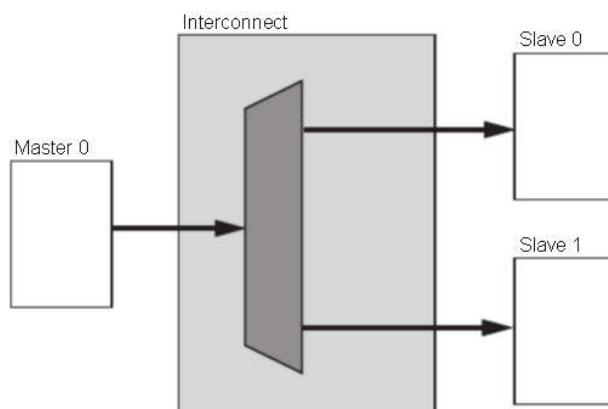
1-to-N Interconnect

N-to-M Interconnect (Crossbar Mode)

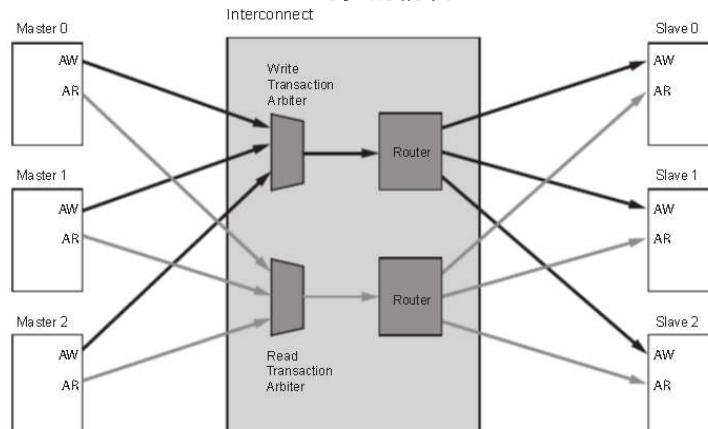
N-to-M Interconnect (Shared Access Mode)



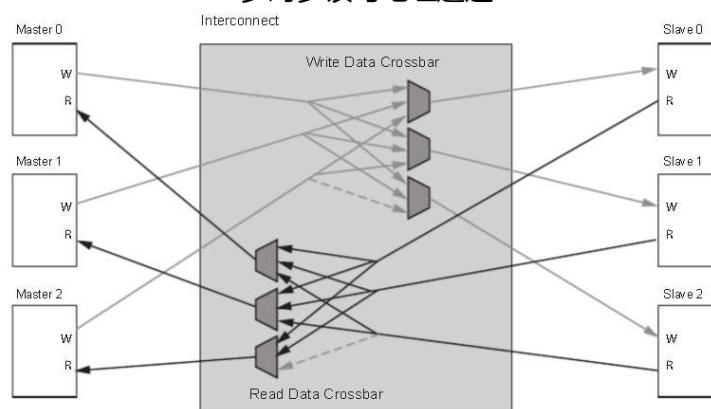
多对一的情况



一对多的情况

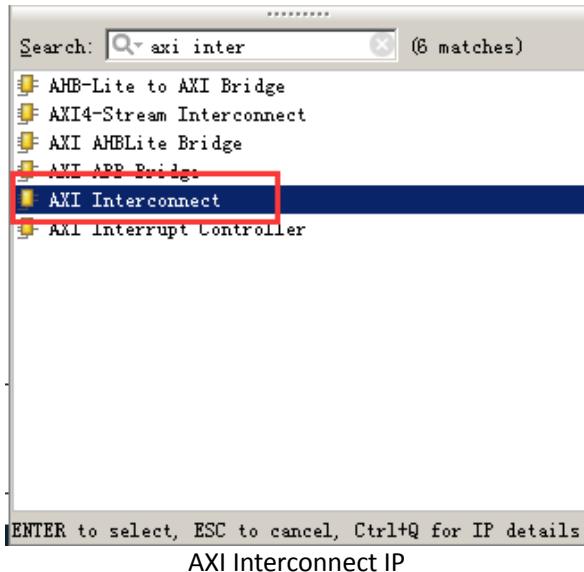


多对多读写地址通道



多对多读写数据通道

ZYNQ 内部的 AXI 接口设备就是通过互联矩阵的方式互联起来的，既保证了传输数据的高效性，又保证了连接的灵活性。Xilinx 在 Vivado 里我们提供了实现这种互联矩阵的 IP 核 axi_interconnect，我们只要调用就可以。



2.2 ZYNQ 芯片开发流程的简介

由于 ZYNQ 将 CPU 与 FPGA 集成在了一起，开发人员既需要设计 ARM 的操作系统应用程序和设备的驱动程序，又需要设计 FPGA 部分的硬件逻辑设计。开发中既要了解 Linux 操作系统，系统的构架，也需要搭建一个 FPGA 和 ARM 系统之间的硬件设计平台。所以 ZYNQ 的开发是需要软件人员和硬件人员协同设计并开发的。这既是 ZYNQ 开发中所谓的“软硬件协同设计”。

ZYNQ 系统的硬件系统和软件系统的设计和开发需要用到一下的开发环境和调试工具：
Xilinx Vivado。

Vivado 设计套件实现 FPGA 部分的设计和开发，管脚和时序的约束，编译和仿真，实现 RTL 到比特流的设计流程。Vivado 并不是 ISE 设计套件的简单升级，而是一个全新的设计套件。它替代了 ISE 设计套件的所有重要工具，比如 Project Navigator、Xilinx Synthesis Technology、Implementation、CORE Generator、Constraint、Simulator、Chipscope Analyzer、FPGA Editor 等设计工具。

Xilinx SDK (Software Development Kit)，SDK 是 Xilinx 软件开发套件(SDK),在 Vivado 硬件系统的基础上，系统会自动配置一些重要参数，其中包括工具和库路径、编译器选项、JTAG 和闪存设置，调试器连接已经裸机板支持包(BSP)。SDK 也为所有支持的 Xilinx IP 硬核提供了驱动程序。SDK 支持 IP 硬核 (FPGA 上) 和处理器软件协同调试，我们可以使用高级 C 或 C++语言来开发和调试 ARM 和 FPGA 系统，测试硬件系统是否工作正常。SDK 软件也是 Vivado 软件自带的，无需单独安装。

ZYNQ 的开发也是先硬件后软件的方法。具体流程如下：

- 1) 在 Vivado 上新建工程，增加一个嵌入式的源文件。
- 2) 在 Vivado 里添加和配置 PS 和 PL 部分基本的外设，或需要添加自定义的外设。
- 3) 在 Vivado 里生成顶层 HDL 文件，并添加约束文件。再编译生成比特流文件（*.bit）。
- 4) 导出硬件信息到 SDK 软件开发环境，在 SDK 环境里可以编写一些调试软件验证硬件和软件，结合比特流文件单独调试 ZYNQ 系统。
- 5) 在 SDK 里生成 FSBL 文件。
- 6) 在 VMware 虚拟机里生成 u-boot.elf、bootloader 镜像。
- 7) 在 SDK 里通过 FSBL 文件，比特流文件 system.bit 和 u-boot.elf 文件生成一个 BOOT.bin 文件。
- 8) 在 VMware 里生成 Ubuntu 的内核镜像文件 zImage 和 Ubuntu 的根文件系统。另外还需要对 FPGA 自定义的 IP 编写驱动。
- 9) 把 BOOT、内核、设备树、根文件系统文件放入到 SD 卡中，启动开发板电源，Linux 操作系统会从 SD 卡里启动。

以上是典型的 ZYNQ 开发流程，但是 ZYNQ 也可以单独做为 ARM 来使用，这样就不需要关系 PL 端资源，和传统的 ARM 开发没有太大区别。ZYNQ 也可以只使用 PL 部分，但是 PL 的配置还是要 PS 来完成的，就是无法通过传统的固化 Flash 方式把只要 PL 的固件固化起来。

2.3 学习 ZYNQ 要具备哪些技能

2.3.1 软件开发人员

- ✓ 计算机组成原理
- ✓ C、C++语言
- ✓ 计算机操作系统
- ✓ tcl 脚本
- ✓ 良好的英语阅读基础

2.3.2 逻辑开发人员

- ✓ 计算机组成原理
- ✓ C 语言
- ✓ 数字电路基础
- ✓ Verilog、VHDL 语言
- ✓ 良好的英语阅读基础

第三章 Vivado 开发环境

3.1 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 年开始便筹划推出新一代的软件开发环境，经历 10 年时间打造出了 Vivado 工具这一巅峰之作。

3.2 Vivado 软件版本

Vivado 的软件版本在不断的升级中，到目前为止最新的软件版本已经是 2017.4 了。因为 ZYNQ 开发板的所有例程和教程我们都在 Vivado 2017.4 的开发环境中完成。为了避免软件版本版本的原因而导致一些无法解释的问题，还是希望大家学习过程中与我们保持同步。用户使用之前需要安装 Vivado 2017.4 的软件。因为 Vivado 软件比较大，我们没有提供光盘安装文件，只提供下载链接，另外用户也可以到 Xilinx 的官网下载，官网下载需要注册相关账号。

Vivado 软件的 Xilinx 官方下载地址：

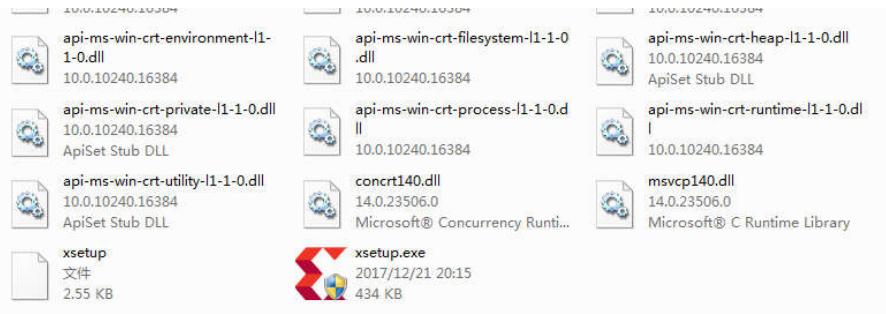
<http://china.xilinx.com/support/download.html>

The screenshot shows the Xilinx website's download section for the Vivado Design Suite. A sidebar on the left lists versions from 2017.4 down to 2016.4. The main content area displays the 'Vivado 通用信息 - 2017.4' and 'Vivado Design Suite - HLx 版本 - 2017.4 Full Product Installation' sections. The second section provides download details and links to support documents.

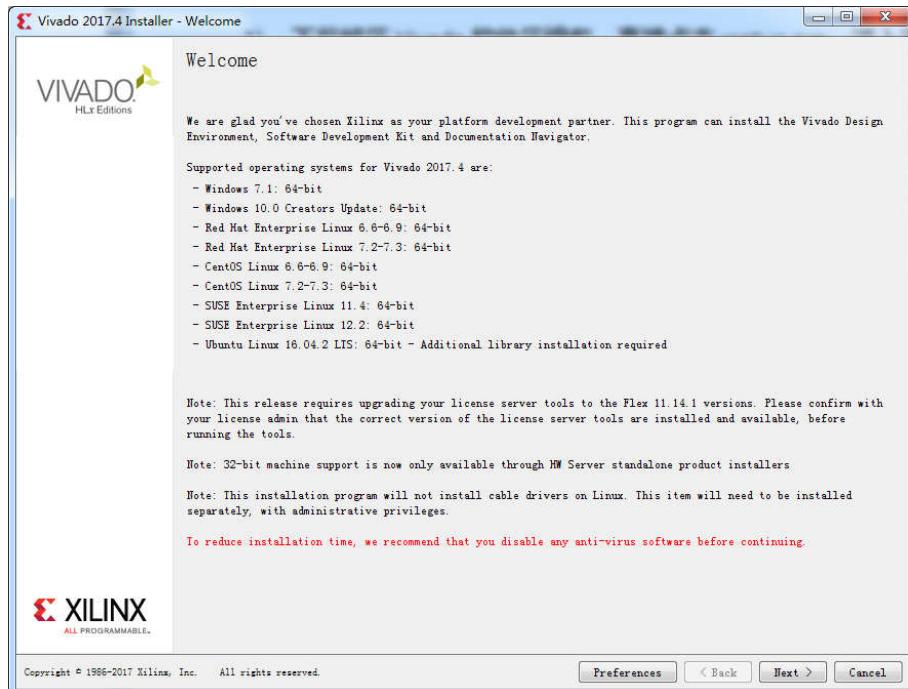
Vivado 提供了 Linux 版和 Windows 版，还提供二合一版本，我们这里使用二合一版本，既能满足 Windows 开发又能满足 Linux 开发，Vivado 要求操作系统必须是 64 位。

3.3 Vivado 软件 Windows 下安装

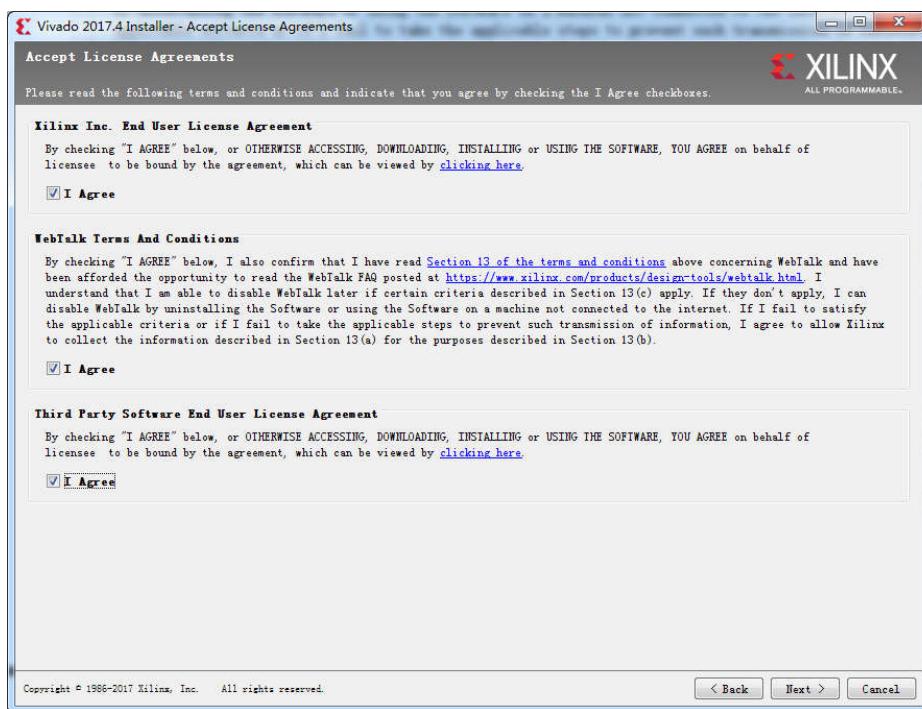
- 1) 下载解压 Vivado 软件压缩包，直接点击 xsetup.exe，进入安装，不过为了更好的安装，请关闭杀毒软件，各种电脑管家



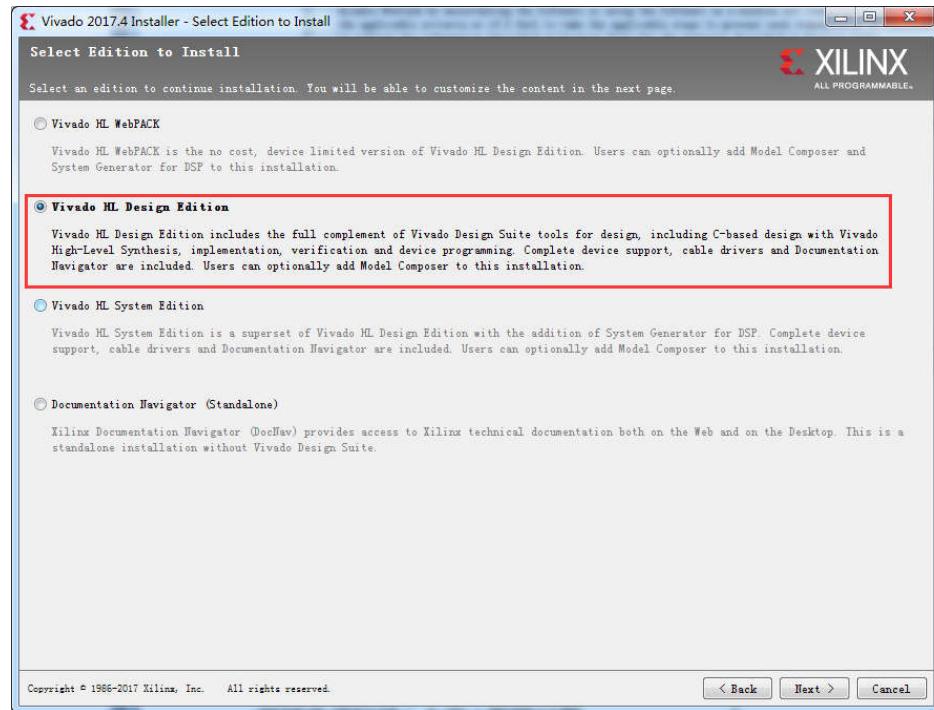
- 2) 如果提示版本更新，我们忽略更新，点击 “Continue”
- 3) 点击 “next” 进行安装，可以看到 Vivado 对系统要求，



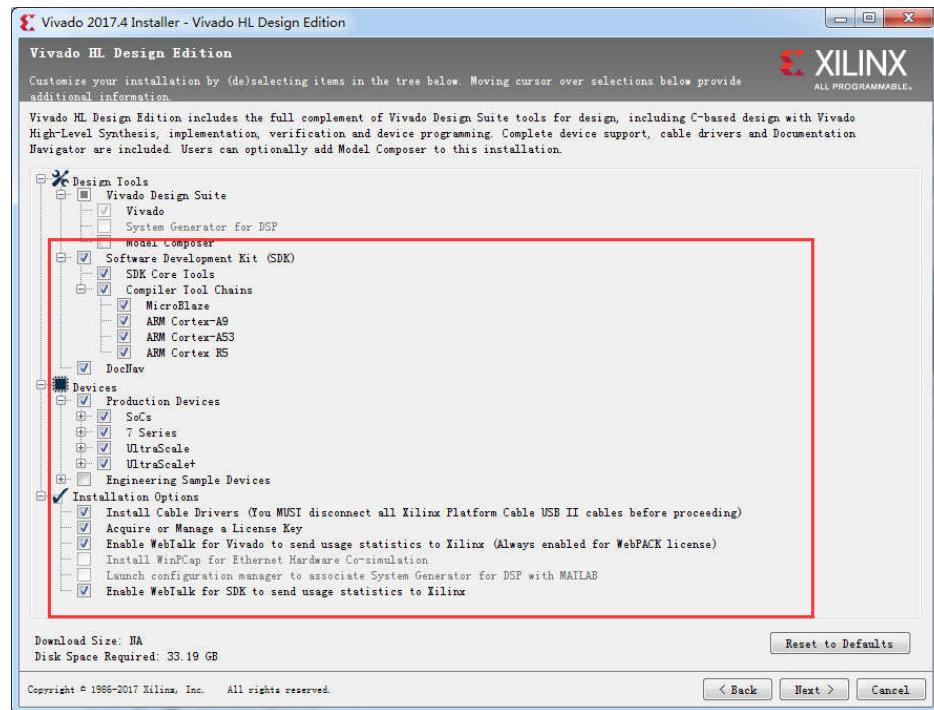
4) 点击 “I Agree” 接受各个条款



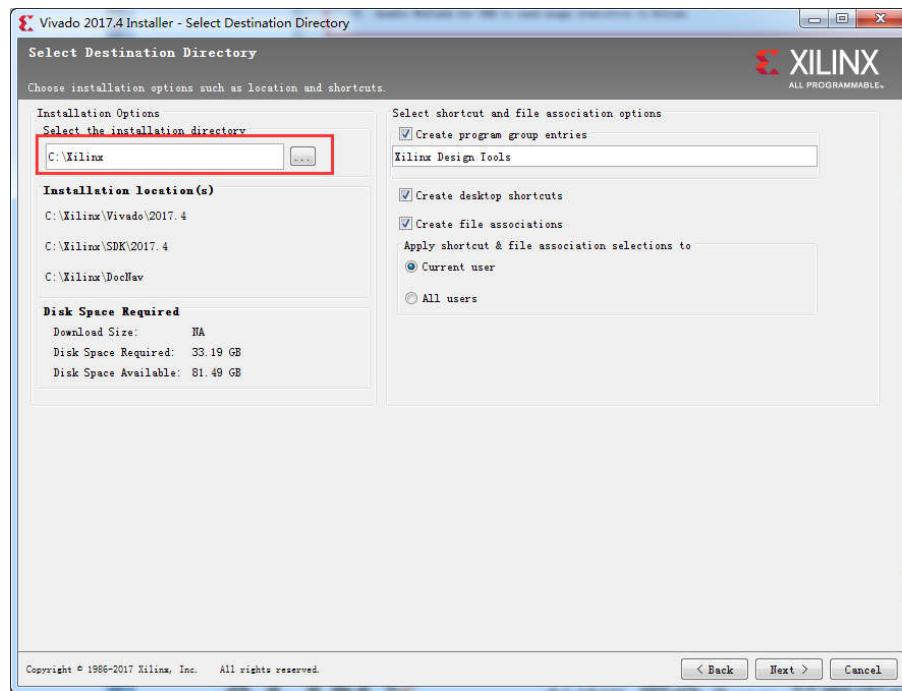
5) 选择 “Vivado HL Design Edition”



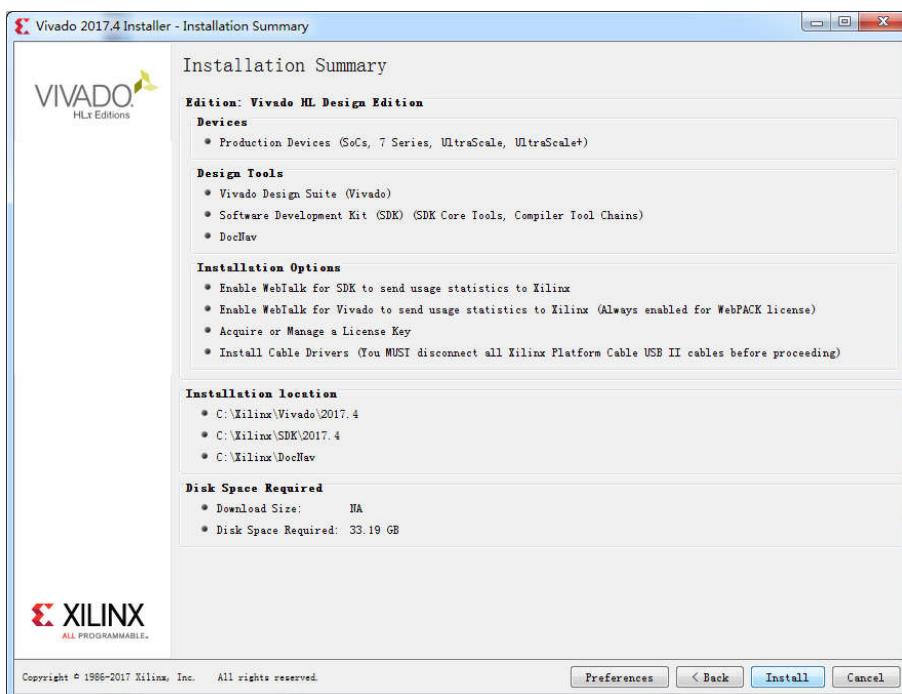
6) 这里使用默认配置，点击“next”



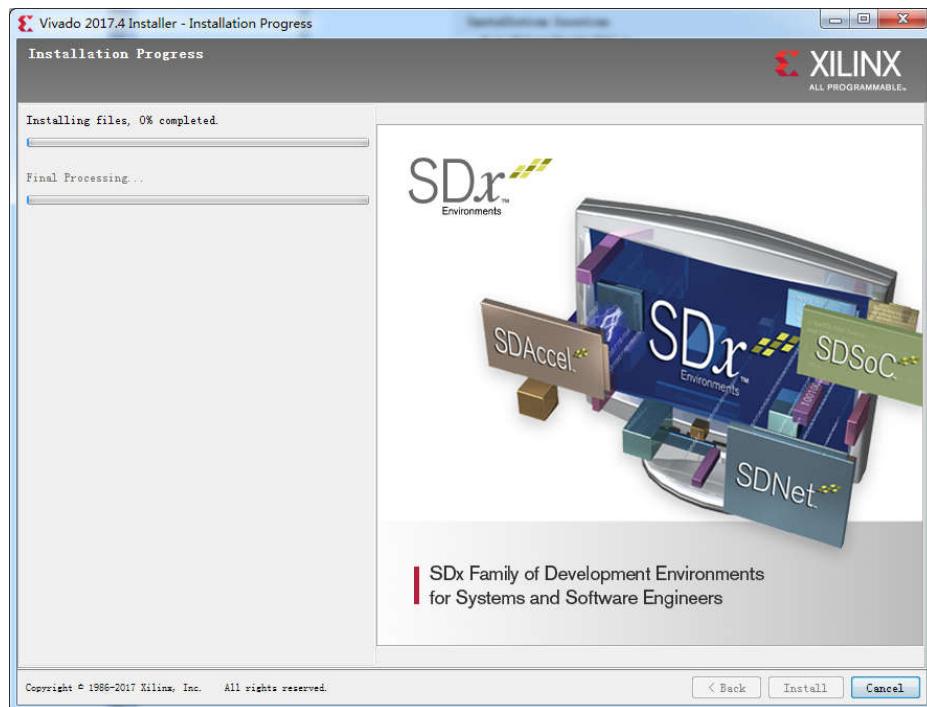
7) 安装路径这里没有修改，安装路径不能有中文、空格等特殊字符，同时电脑的用户名不要是中文、带空格的名称。可以看到 Vivado 对硬盘大小的要求，大约 33G。



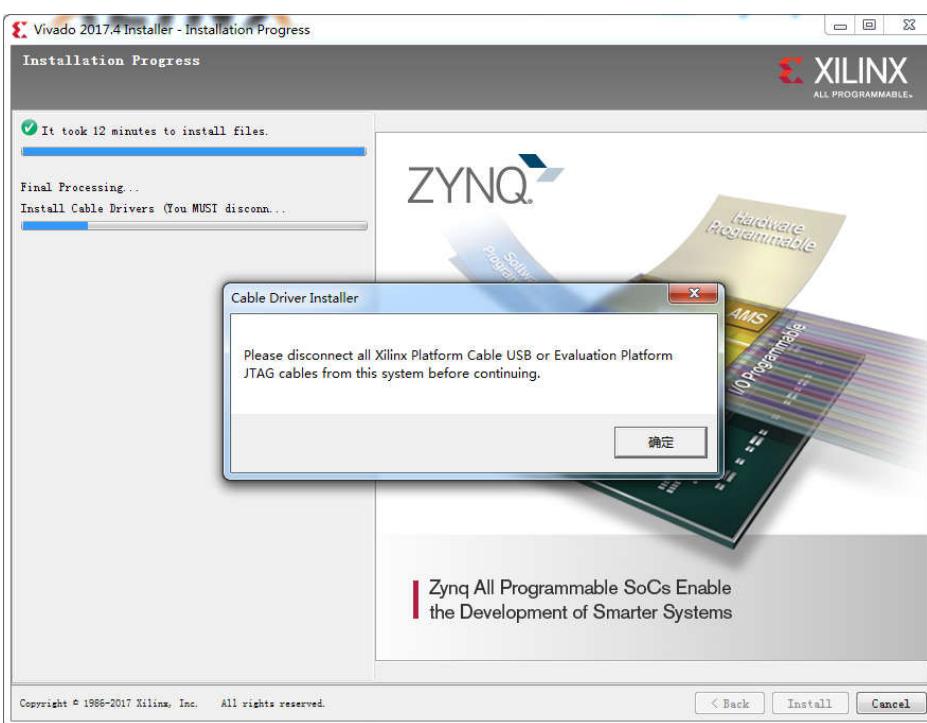
8) 点击 “Install” 安装



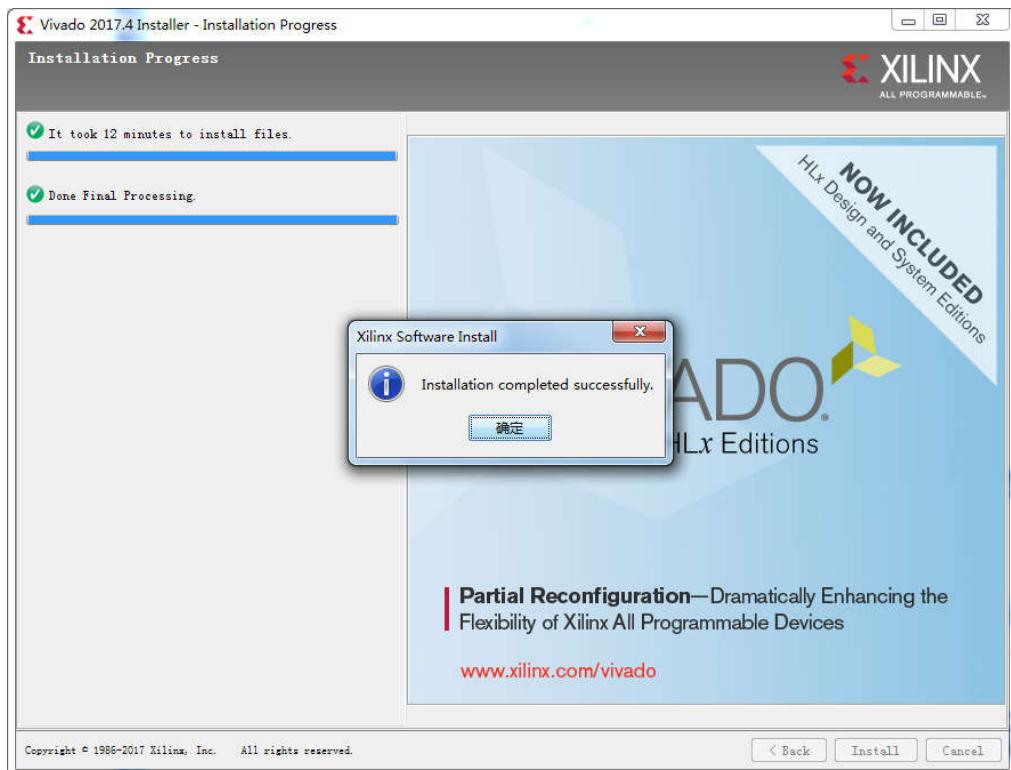
9) 等待安装，时间较长，如果没有关闭杀毒软件和电脑管家，安装过程可能会被拦截，导致安装软件后不能使用



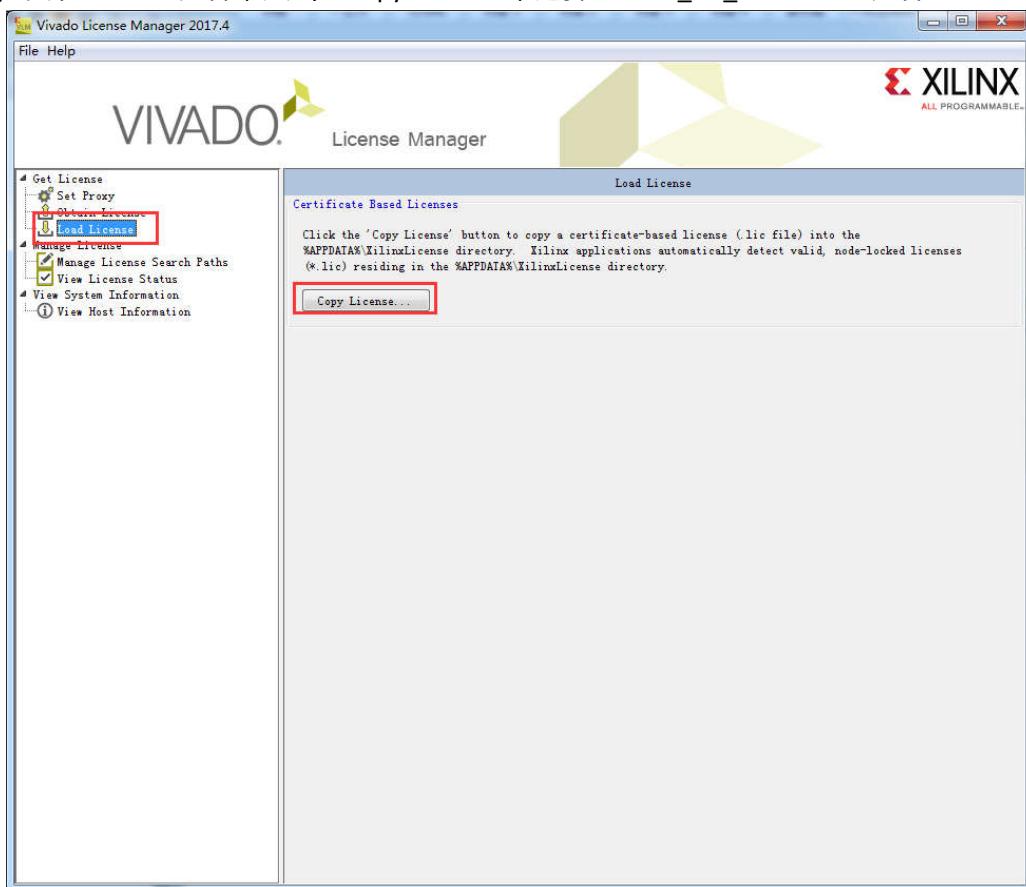
10) 这个时候提示我们断开下载器或者开发板的 JTAG 线，点“确定”



11) 提示安装成功



12) 安装 License 文件，点击 “Copy License”，选择 “xilinx_ise_vivado.lic” 文件。



13) 可以看到安装成功，要提醒，电脑用户名不要有中文和空格



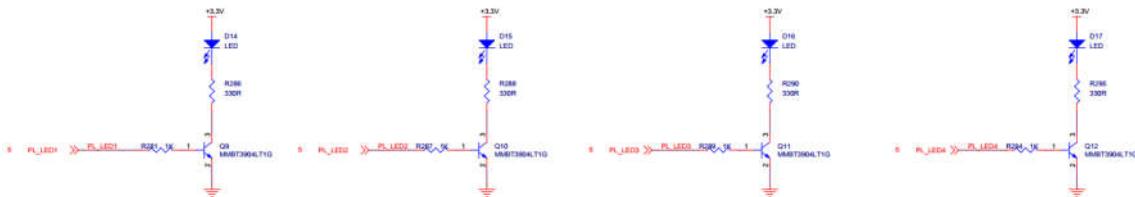
第四章 PL 的 “Hello World” LED 实验

实验 Vivado 工程为 “led”。

对于 ZYNQ 来说 PL(FPGA)开发是至关重要的 ,这也是 ZYNQ 比其他 ARM 的有优势的地方 ,可以定制化很多 ARM 端的外设 ,在定制 ARM 端的外设之前先让我们通过一个 LED 例程来熟悉 PL (FPGA)的开发流程 ,熟悉 Vivado 软件的基本操作 ,这个开发流程和不带 ARM 的 FPGA 芯片完全一致。

在本例程中 ,我们要做的是 LED 灯控制实验 ,每秒钟控制开发板上的 LED 灯翻转一次 ,实现亮、灭、亮、灭的控制。会控制 LED 灯 ,其它外设也慢慢就会了。

4.1 LED 硬件介绍



PL 端只能直接控制 PL 端的 LED , PS 端的外设是无法直接控制的 ,我们可以看到 4 颗 LED 通过三极管连接到 3.3V 电源 ,当三极管导通 LED 就会亮 ,只要连接到三极管的 IO 为高电平 ,三极管就会导通。无论是学习 FPGA 还是学习 ARM ,基本的硬件知识还是要掌握 ,例如看原理图。

从原理图我们可以知道四个 LED 对应的 ZYNQ 芯片的管脚情况如下 :

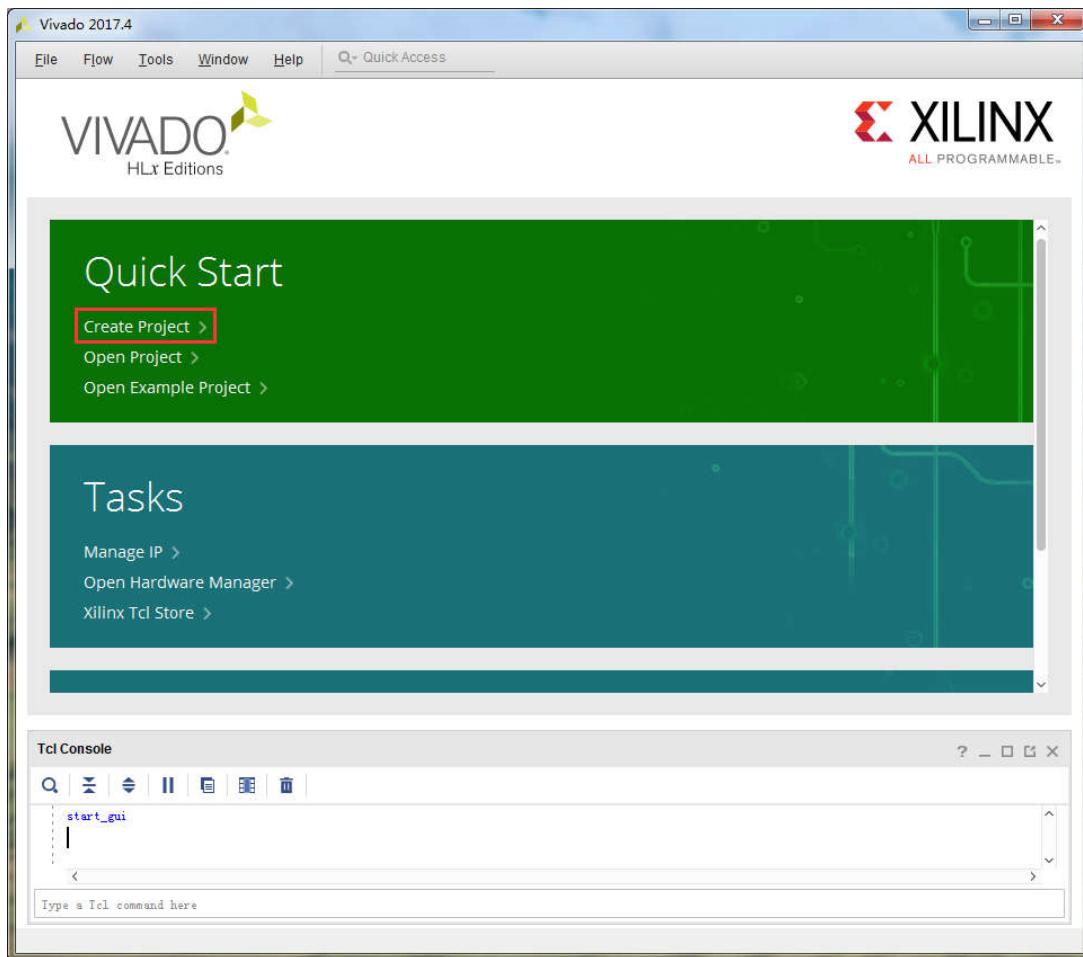
- PL_LED1 —— F5
- PL_LED2 —— E5
- PL_LED3 —— G5
- PL_LED4 —— G6

4.2 创建 Vivado 工程

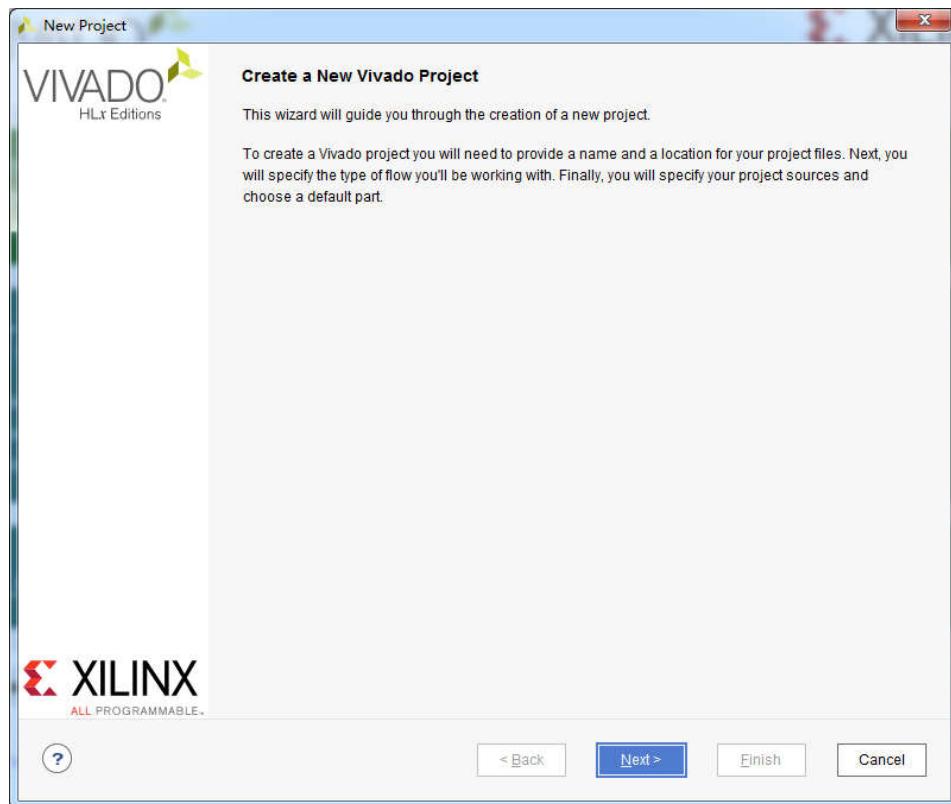
- 1) 启动 Vivado , 在 Windows 中可以通过双击 Vivado 快捷方式启动



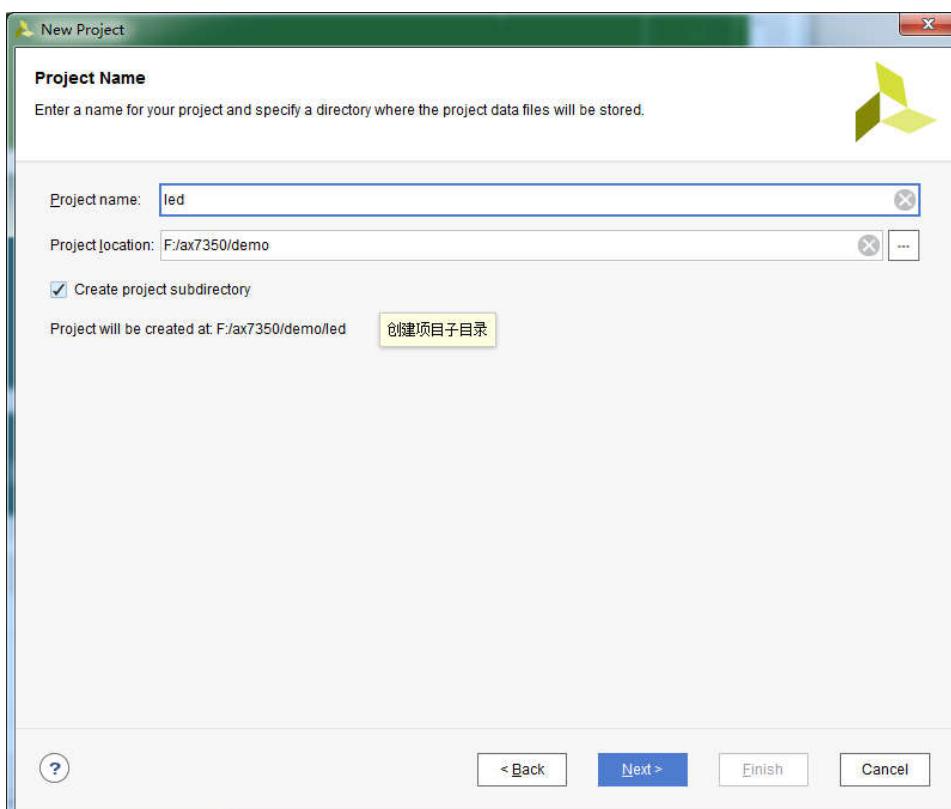
- 2) 在 Vivado 2017.4 开发环境里点击 “Create New Project” , 创建一个新的工程。



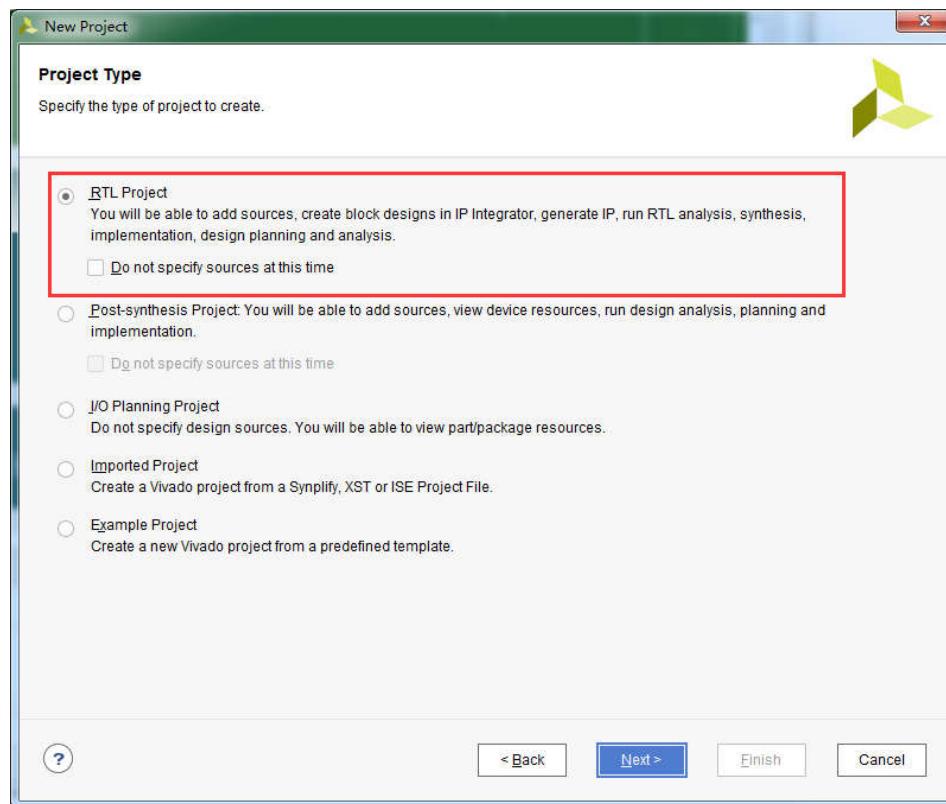
- 3) 弹出一个建立新工程的向导 , 点击 “Next”



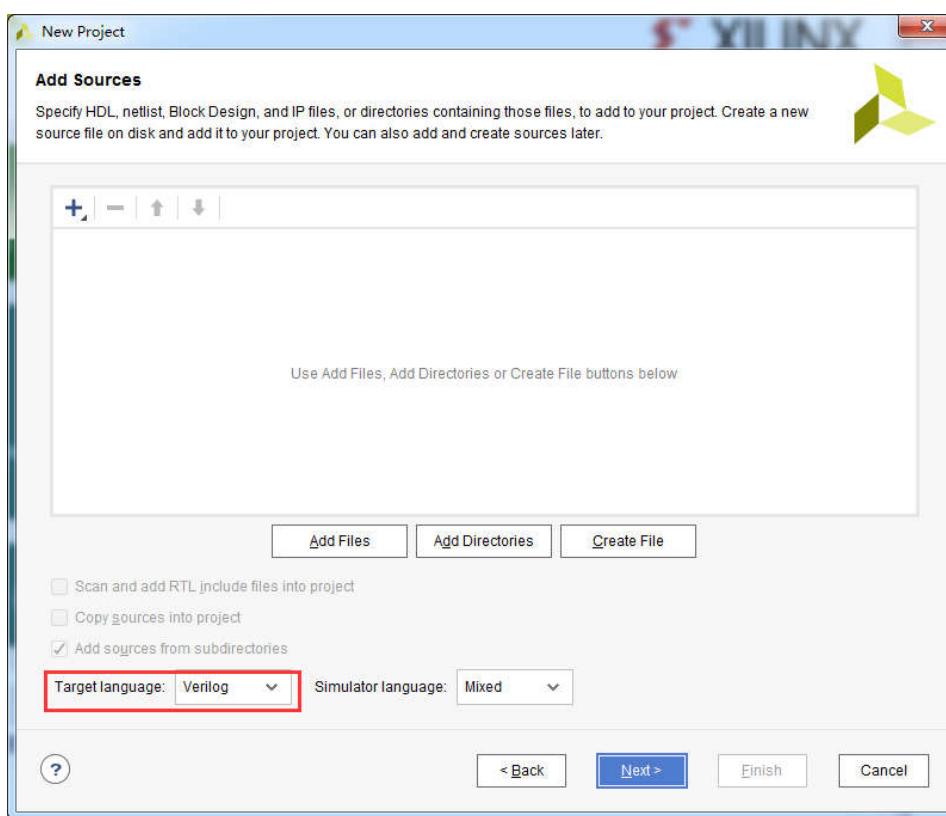
- 4) 在弹出的对话框中输入工程名和工程存放的目录，我们这里取一个 led 的工程名。需要注意工程路径 “Project location” 不能有中文空格，路径名称也不能太长。



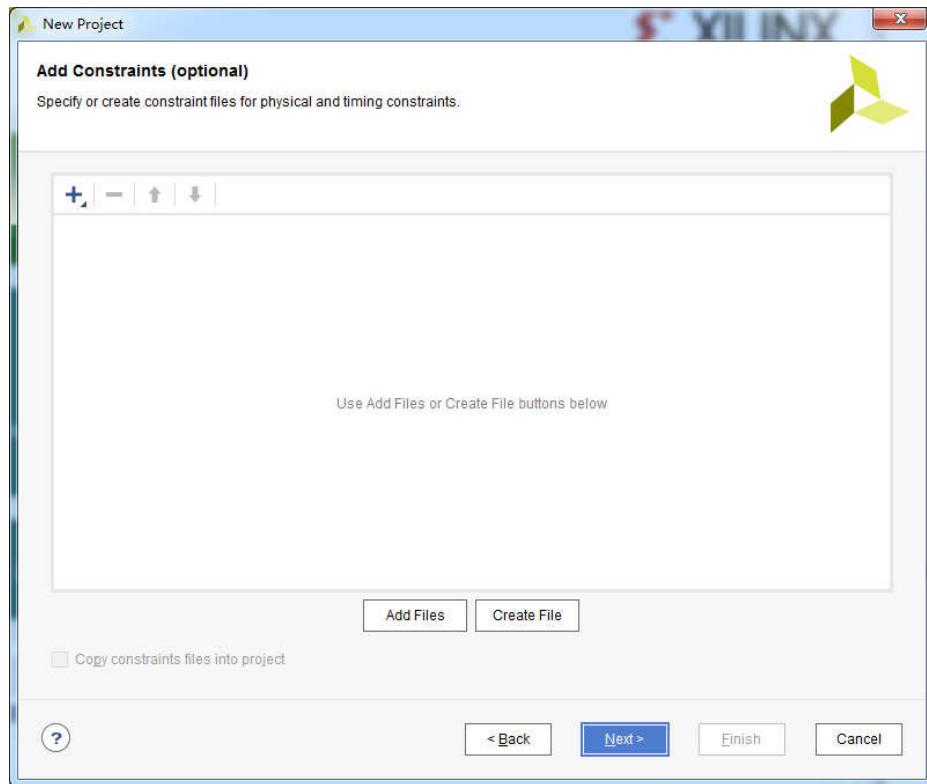
5) 在工程类型中选择 “RTL Project”



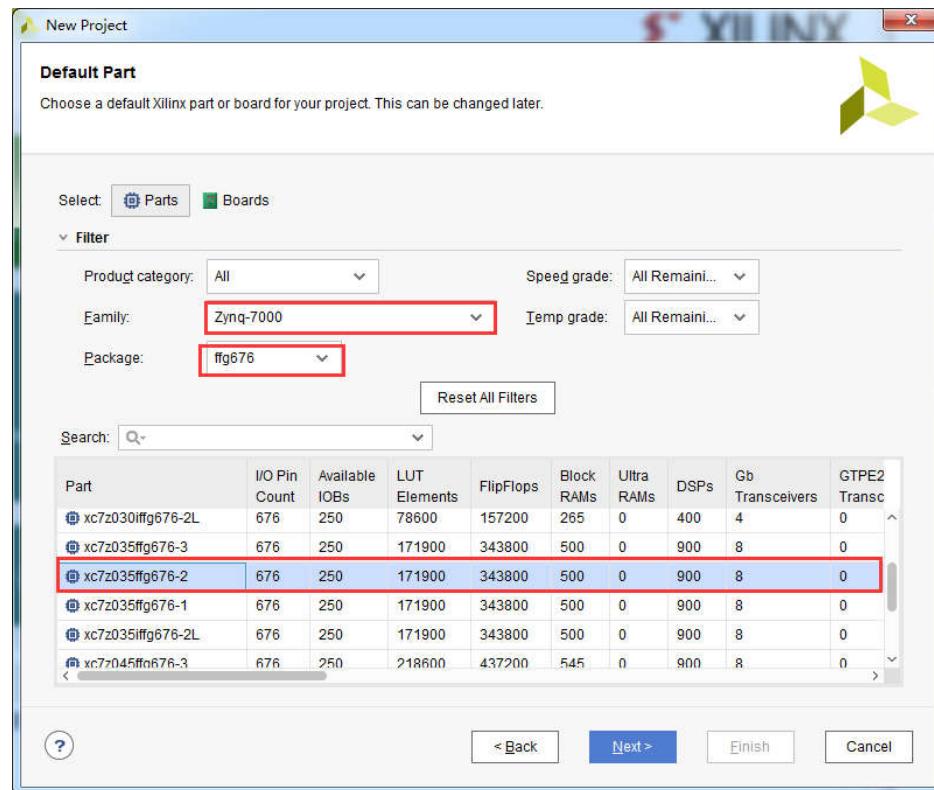
6) 目标语言 “Target language” 选择 “Verilog”



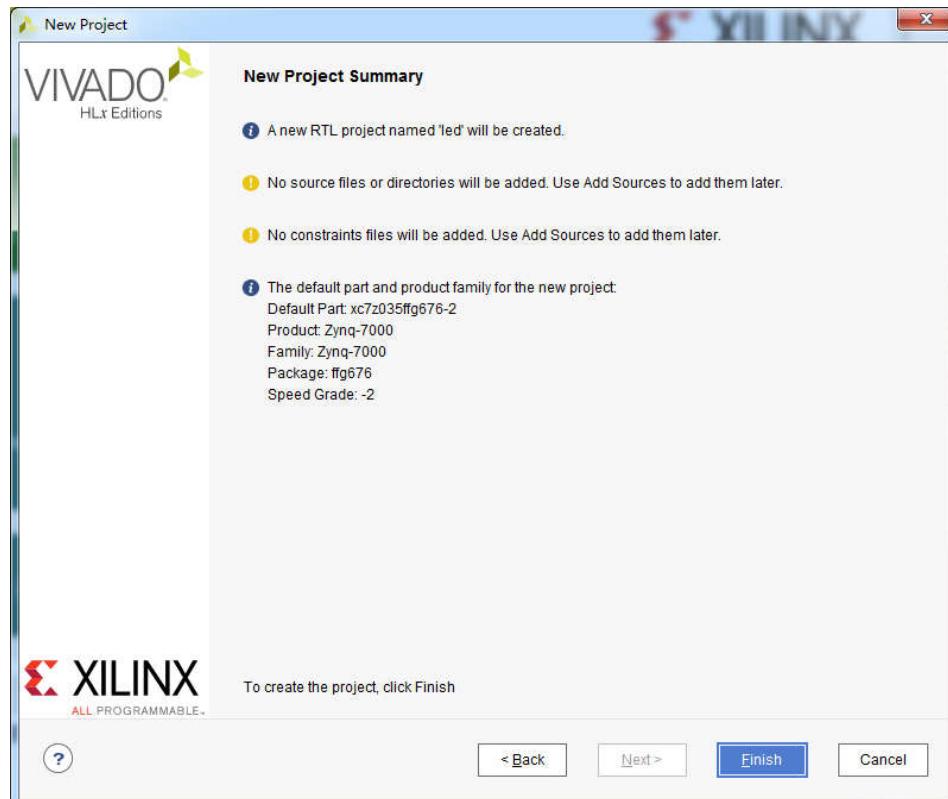
7) 点击 “Next” , 不添加任何文件



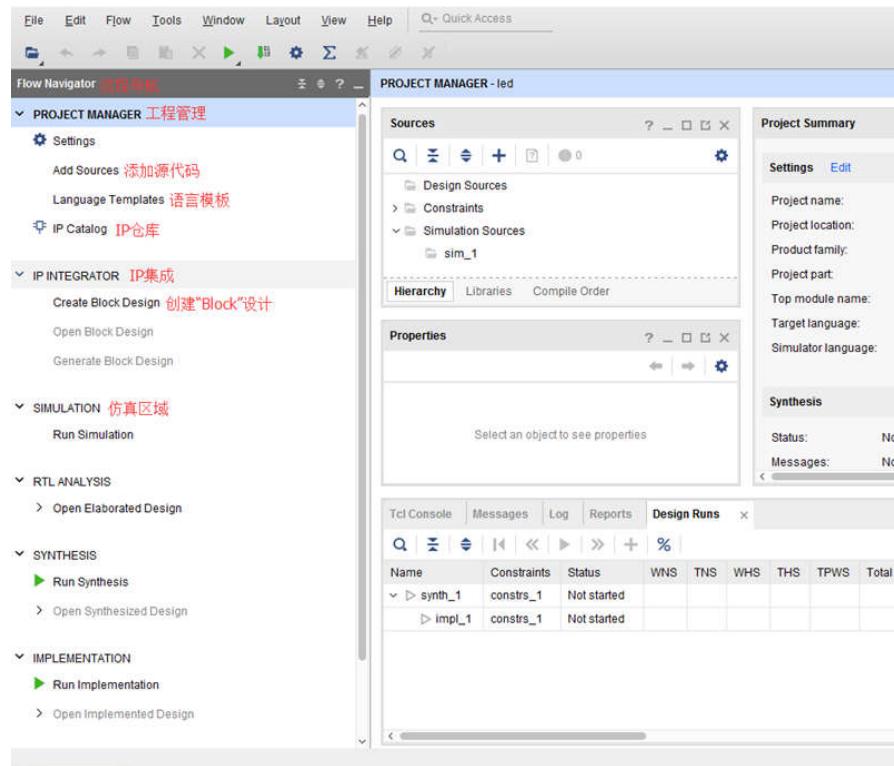
8) 在 “Default Part” 选项中 , 器件家族 “Family” 选择 “Zynq-7000” , 封装类型 “Package” 选择 “ffg676” , 减少我们选择范围。在下拉列表中选择 “xc7z035ffg672-2” , “-2” 表示速率等级 , 数字越大 , 性能越好。



9) 点击“Finish”就可以完成以后名为“led”工程的创建。

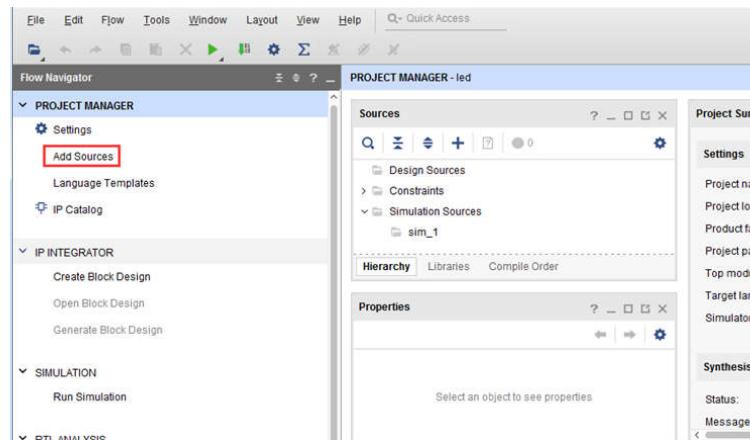


10) Vivado 软件界面

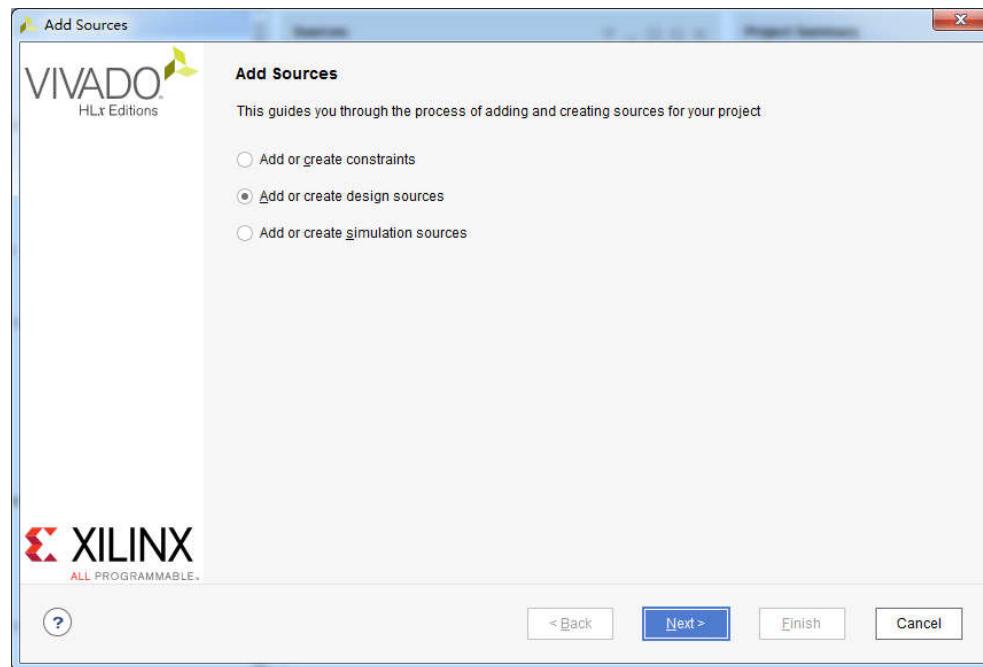


4.3 创建 Verilog 文件点亮 LED

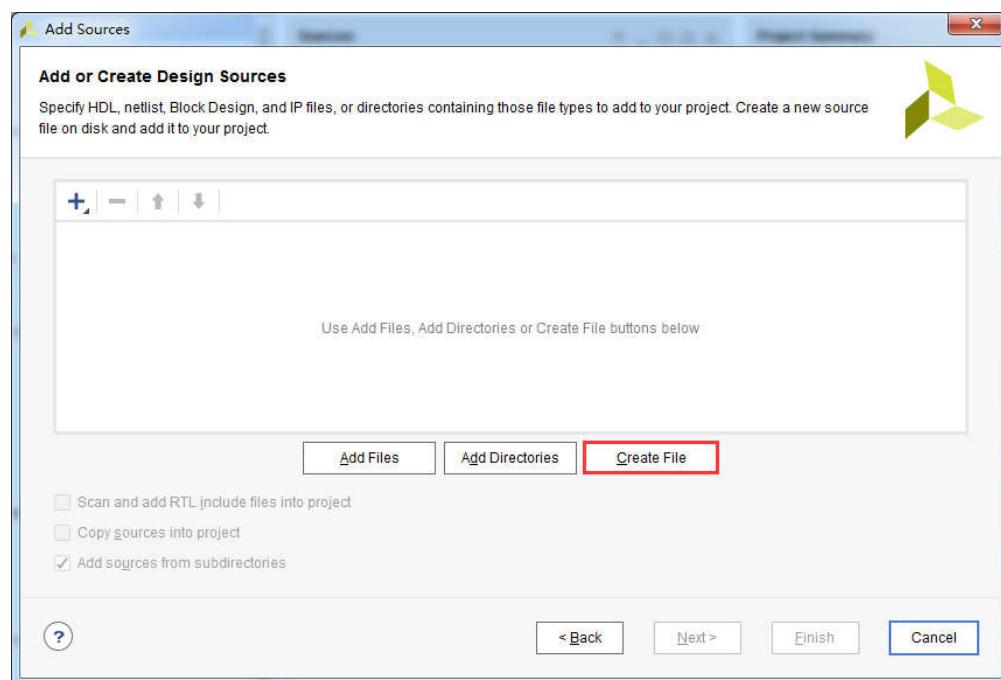
- 1) 点击 Project Manager 下的 Add Sources 图标 (或者使用快捷键 Alt+A)



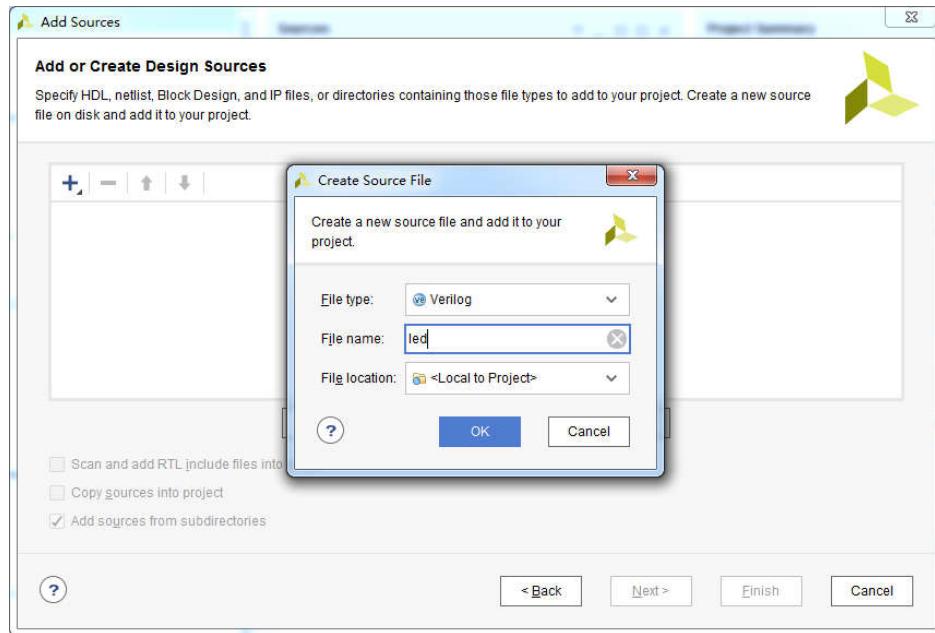
- 2) 选择添加或创建设计源文件 “Add or create design sources” ,点击 “Next”



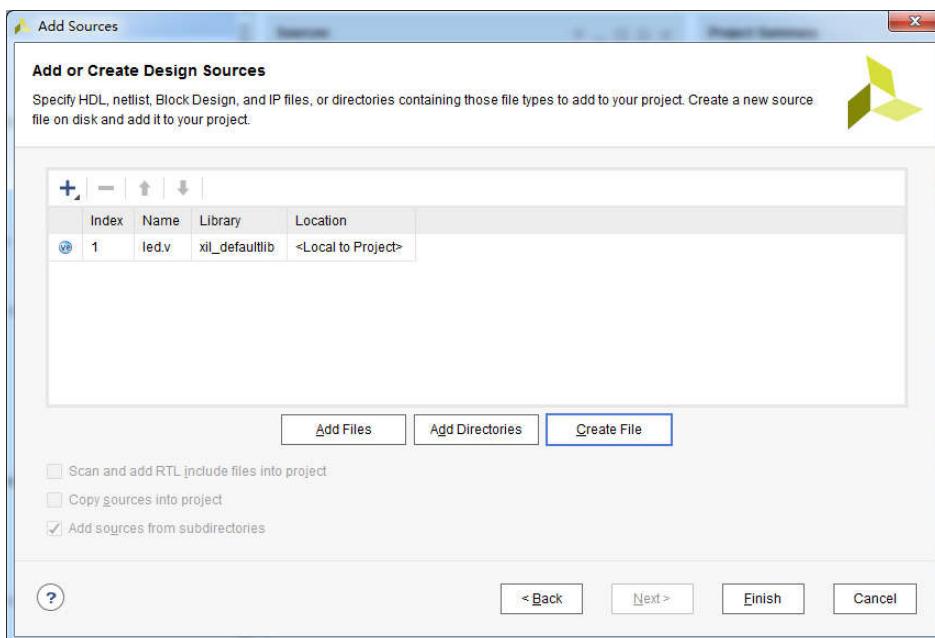
3) 选择创建文件 “Create File”



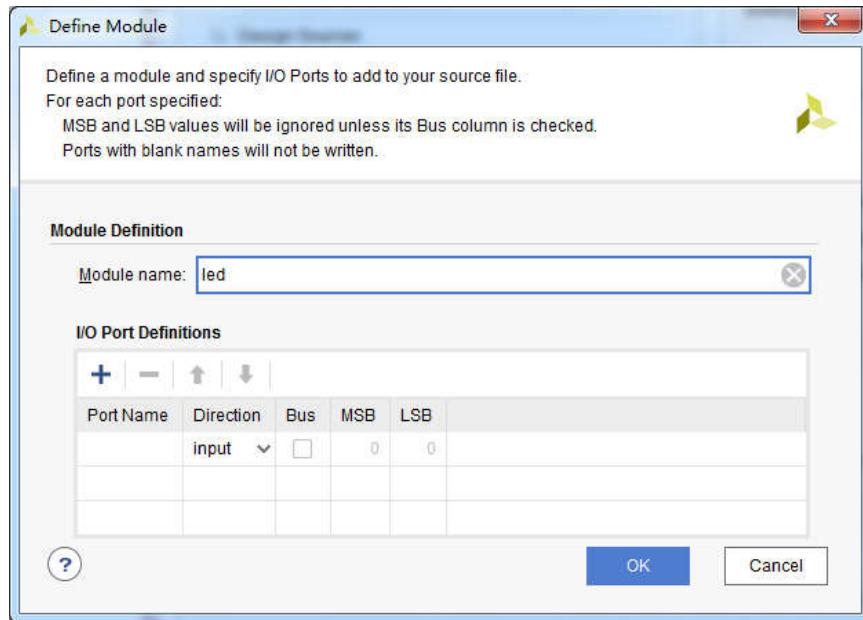
4) 文件名 “File name” 设置为 “led” , 点击 “OK”



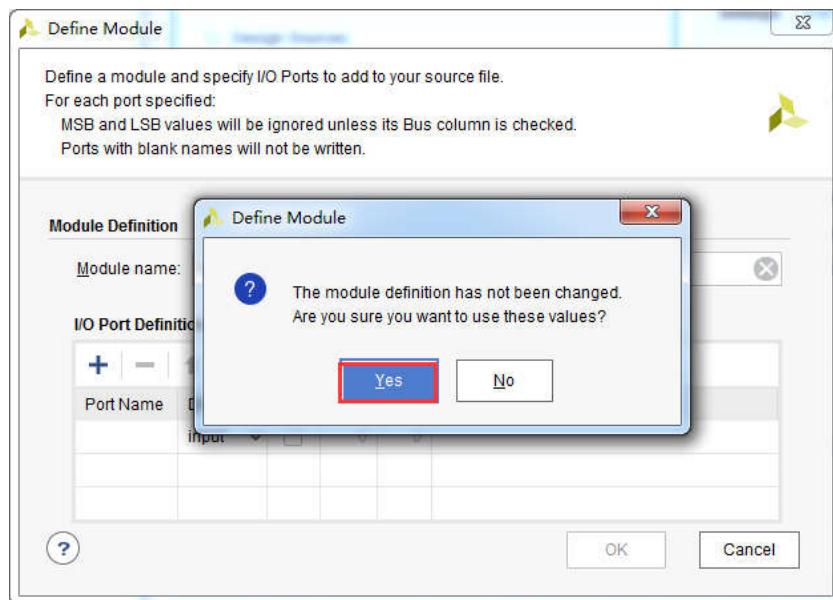
5) 点击 “Finish” ,完成 “led.v” 文件添加



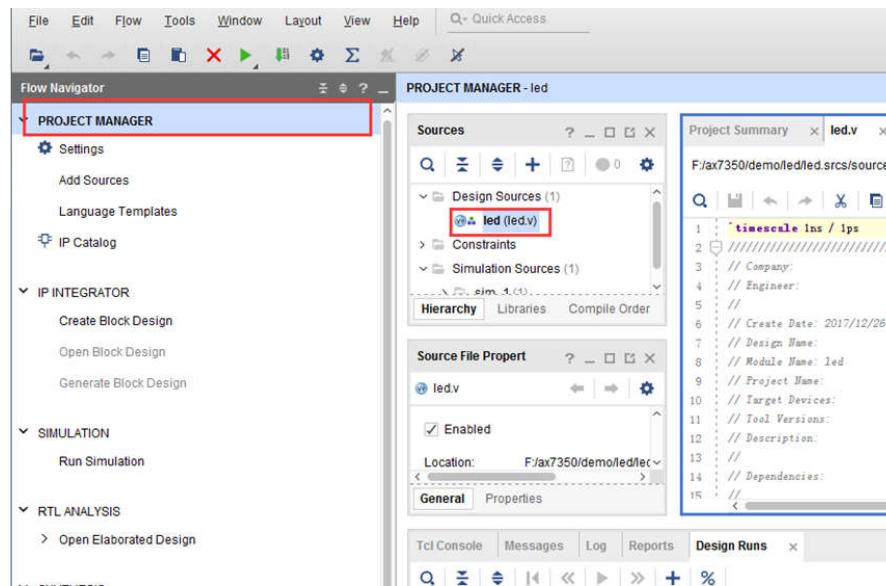
6) 在弹出的模块定义 “Define Module” ,中可以指定 “led.v” 文件的模块名称 “Module name” , 这里默认不变为 “led” , 还可以指定一些端口 , 这里暂时不指定 , 点击 “OK”。



7) 在弹出的对话框中选择 “Yes”



8) 双击 “led.v” 可以打开文件，然后编辑



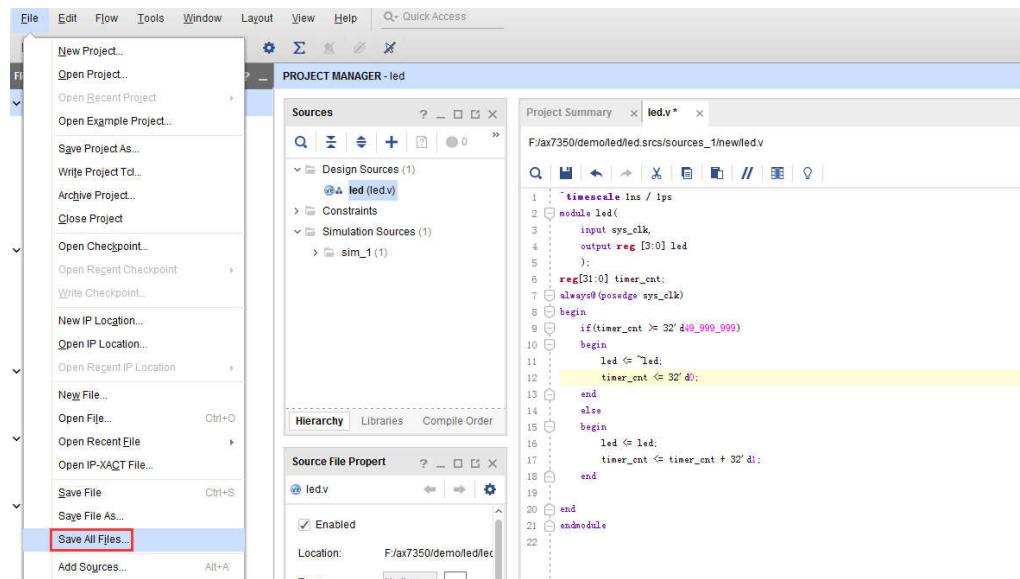
- 9) 编写 “led.v” ,这里定义了一个 32 位的寄存器 timer, 用于循环计数 0~49999999(1 秒钟), 计数到 49999999(1 秒)的时候, 寄存器 timer 变为 0 ,并翻转四个 LED。这样原来 LED 是灭的话 , 就会点亮 , 如果原来 LED 为亮的话 , 就会熄灭。编写好后的代码如下 :

```

`timescale 1ns / 1ps
module led(
    input sys_clk,
    output reg [3:0] led
);
reg[31:0] timer_cnt;
always@(posedge sys_clk)
begin
    if(timer_cnt >= 32'd49_999_999)
        begin
            led <= ~led;
            timer_cnt <= 32'd0;
        end
    else
        begin
            led <= led;
            timer_cnt <= timer_cnt + 32'd1;
        end
end
endmodule

```

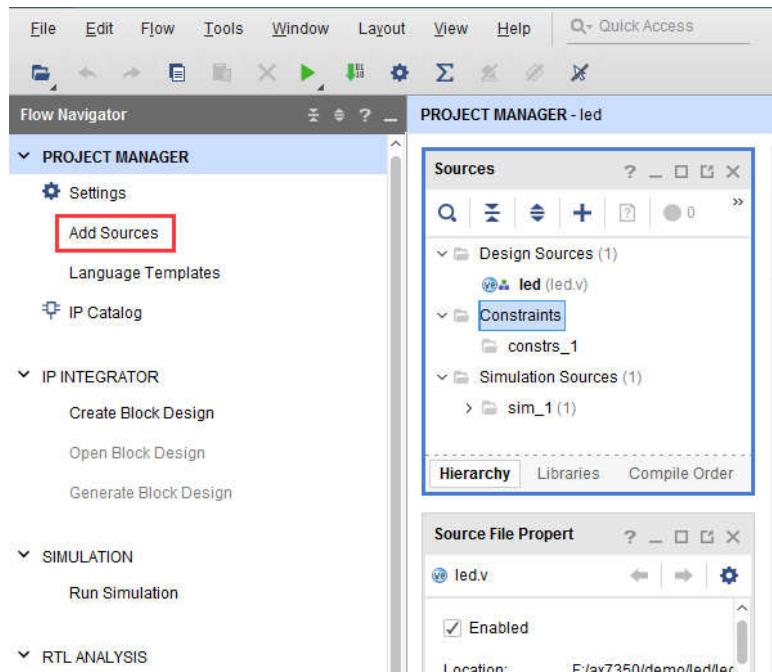
- 10) 编写好代码后保存,点击菜单 File -> Save All Files



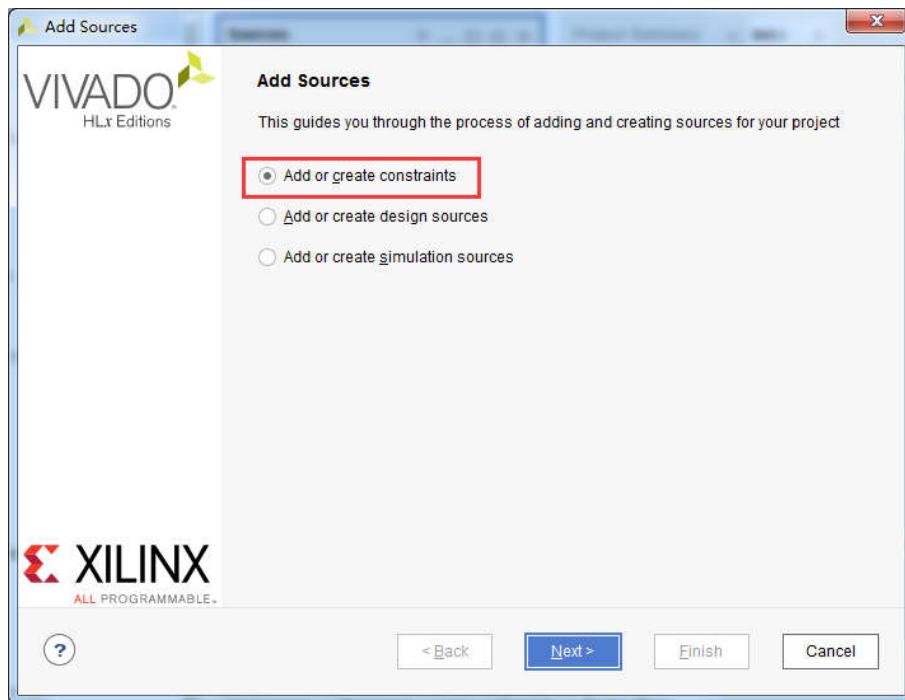
4.4 添加 XDC 约束文件约束管脚

Vivado 使用的约束文件格式为 xdc 文件, xdc 文件里主要是完成管脚的约束, 时钟的约束, 以及组的约束。这里我们需要对 led.v 程序中的输入输出端口分配到 FPGA 的真实管脚上, 这需要准备一个 FPGA 的引脚绑定文件.xdc 并添加到工程中。

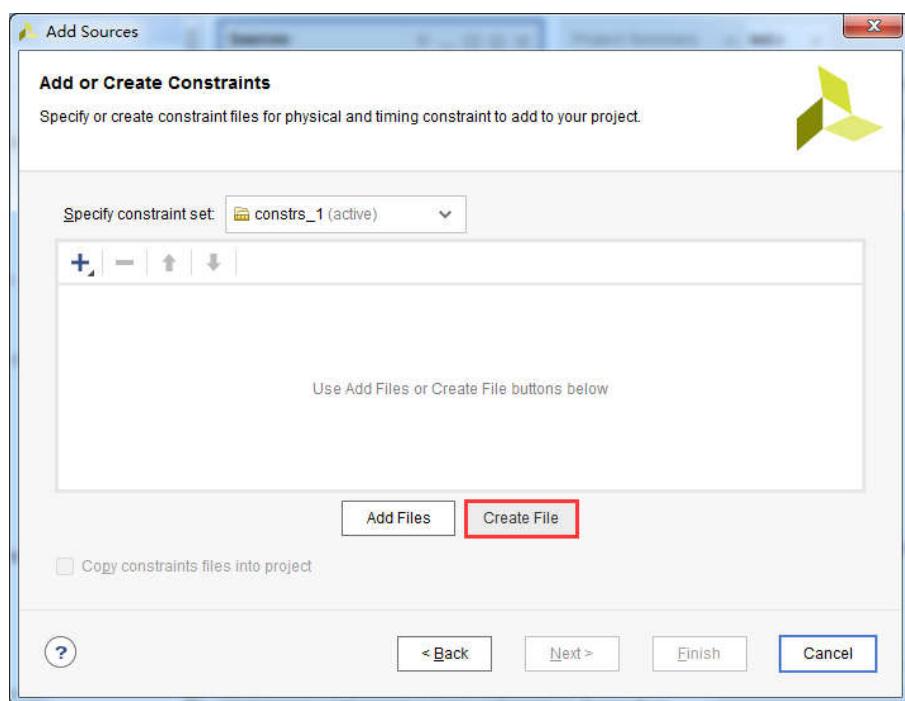
1) 点击 Project Manager 下的 Add Sources



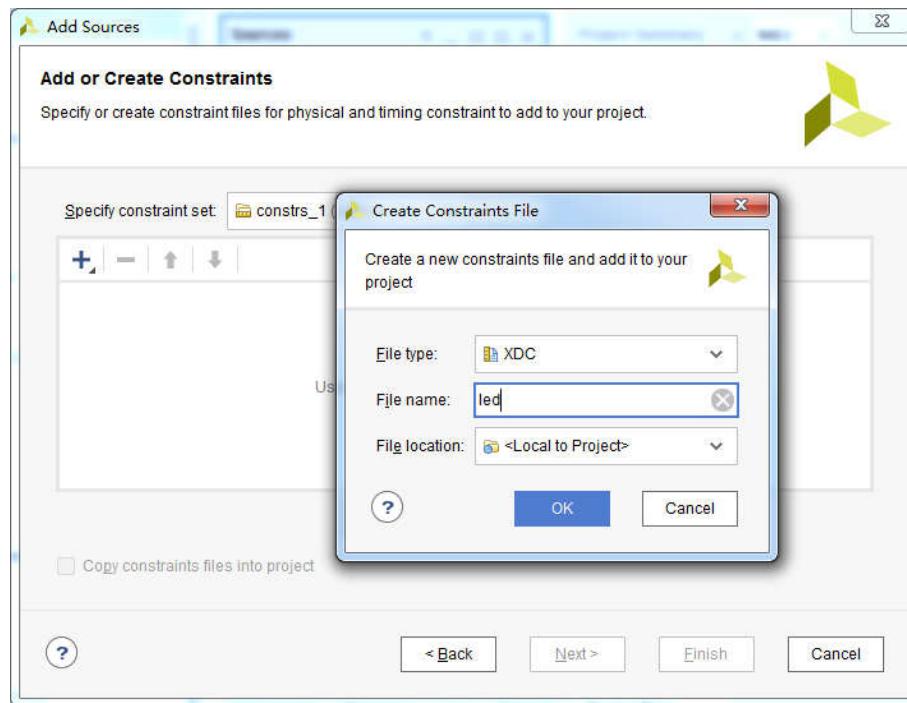
2) 选择 “Add or create constraints” 选项，点击 “Next”



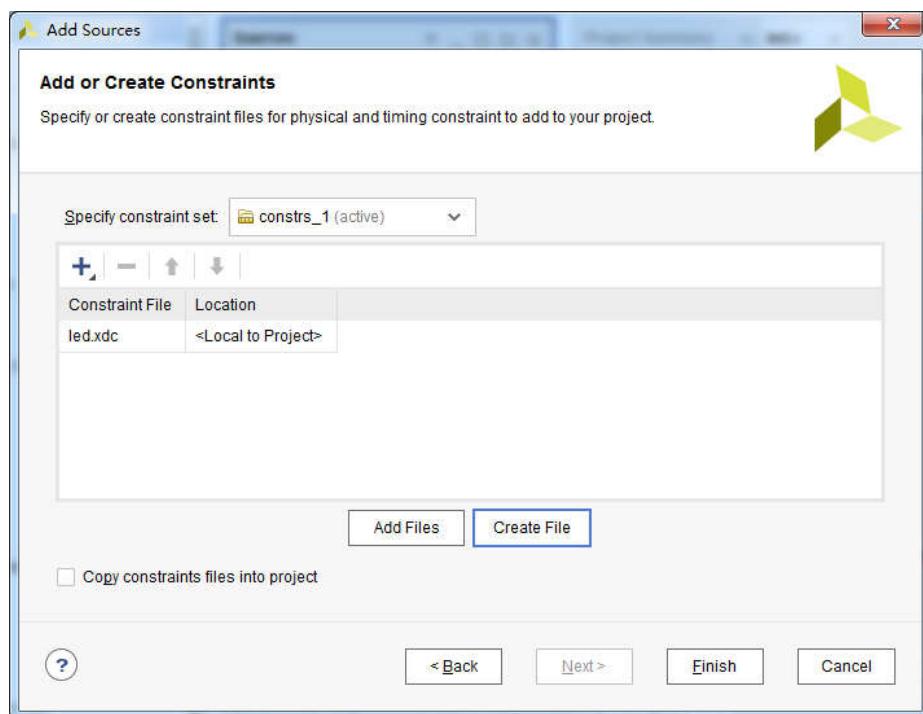
3) 点击 “Create File” 按钮



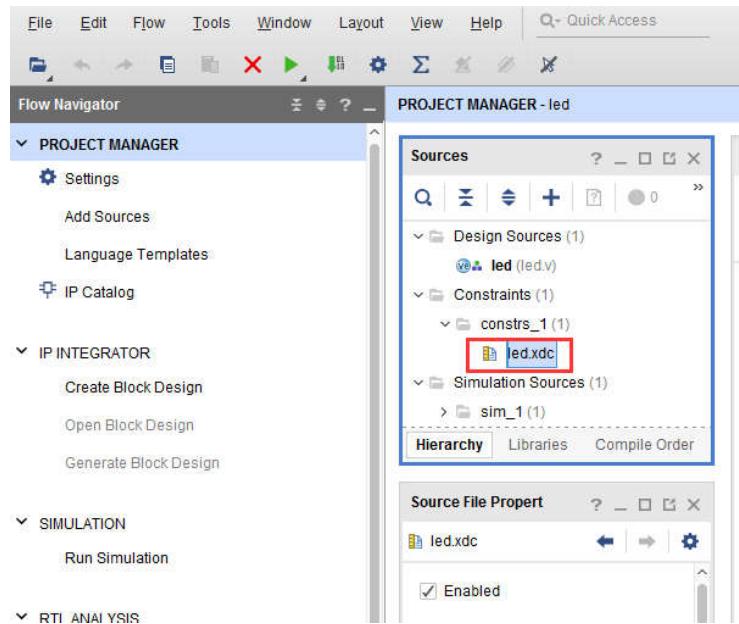
4) 在弹出的对话框里选择 File type 是 XDC, “File name” 为 “led”, 点击 OK 按钮



5) 点击“Finish”完成



6) 这时在 Project Manager 界面下的 Constraints 目录的 constrs_1 目录下已经有了一个“led.xdc”文件



7) 双击打开这个 led.xdc 文件，在这个文件里添加以下的引脚定义

```
set_property IOSTANDARD LVCMOS15 [get_ports {led[3]}]
set_property IOSTANDARD LVCMOS15 [get_ports {led[2]}]
set_property IOSTANDARD LVCMOS15 [get_ports {led[1]}]
set_property IOSTANDARD LVCMOS15 [get_ports {led[0]}]
set_property IOSTANDARD LVCMOS18 [get_ports sys_clk]
set_property PACKAGE_PIN J14 [get_ports sys_clk]
set_property PACKAGE_PIN F5 [get_ports {led[0]}]
set_property PACKAGE_PIN E5 [get_ports {led[1]}]
set_property PACKAGE_PIN G5 [get_ports {led[2]}]
set_property PACKAGE_PIN G6 [get_ports {led[3]}]
```

下面来介绍一下最基本的 XDC 编写的语法，普通 IO 口只需约束引脚号和电压，管脚约束如下：

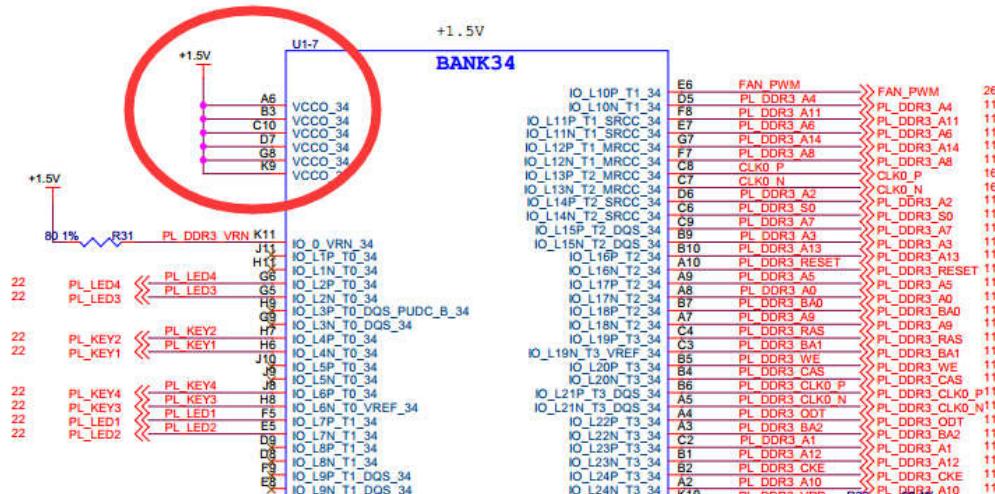
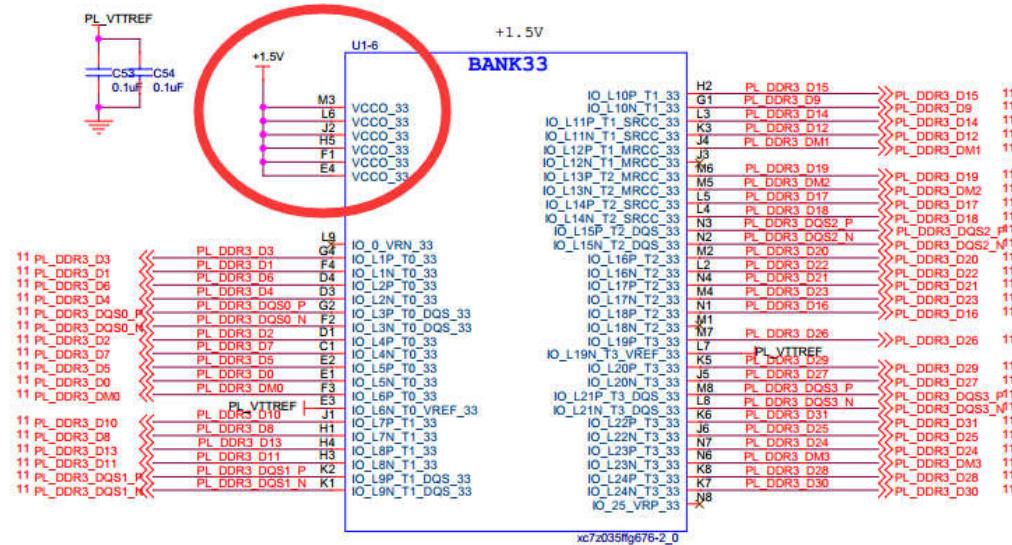
set_property PACKAGE_PIN "引脚编号" [get_ports "端口名称"]

电平信号的约束如下：

set_property IOSTANDARD "电平标准" [get_ports "端口名称"]

这里需要注意文字的大小写，端口名称是数组的话用{}括起来，端口名称必须和源代码中的名字一致，且端口名字不能和关键字一样。

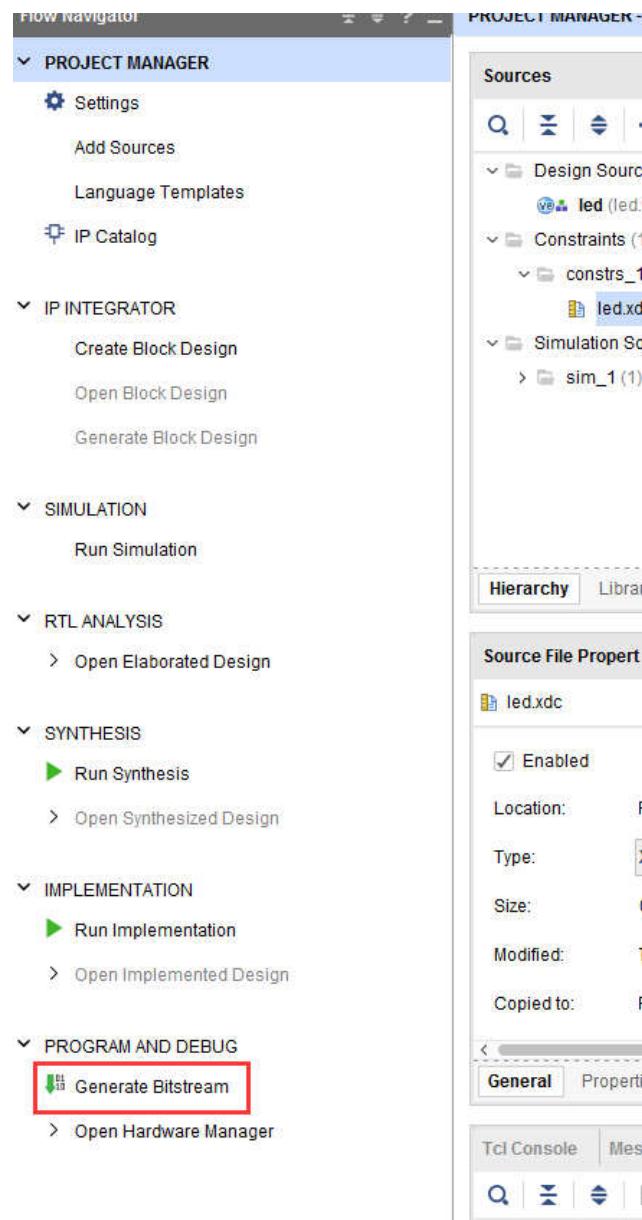
电平标准中 LVCMOS15 后面的数字指 FPGA 的 BANK 电压，LED 所在 BANK 电压为 1.5 伏，所以电平标准为“LVCMOS15”时钟输入 FPGA 的 BANK 电压为 1.8V，所以电平标准为“LVCMOS18”。



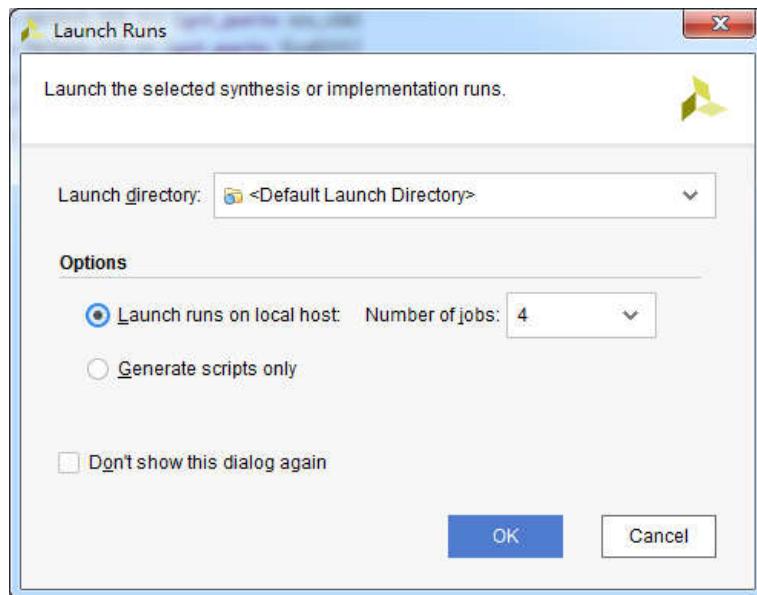
8) 完成后选择菜单 File -> Save all files 保存所有文件

4.5 编译工程

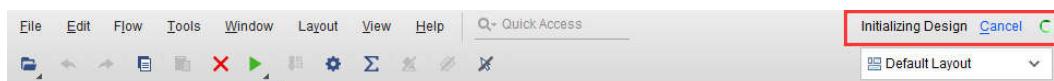
- 1) 编译的过程可以细分为综合、布局布线、生成 bit 文件等，这里我们直接点击“Generate Bitstream”，直接生成 bit 文件。



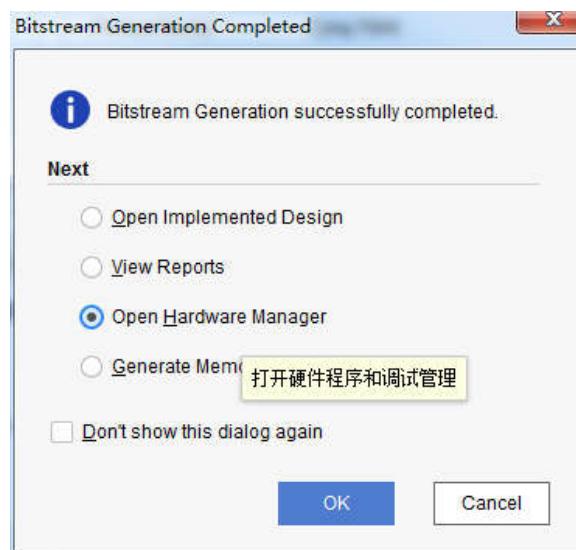
- 2) 在弹出的对话框中可以选择任务数量，这里和 CPU 核心数有关，一般数字越大，编译越快，点击“OK”



- 3) 这个时候开始编译 ,可以看到右上角有个状态信息 ,在编译过程中可能会被杀毒软件、电脑管家拦截运行 ,导致无法编译或很长时间没有编译成功。



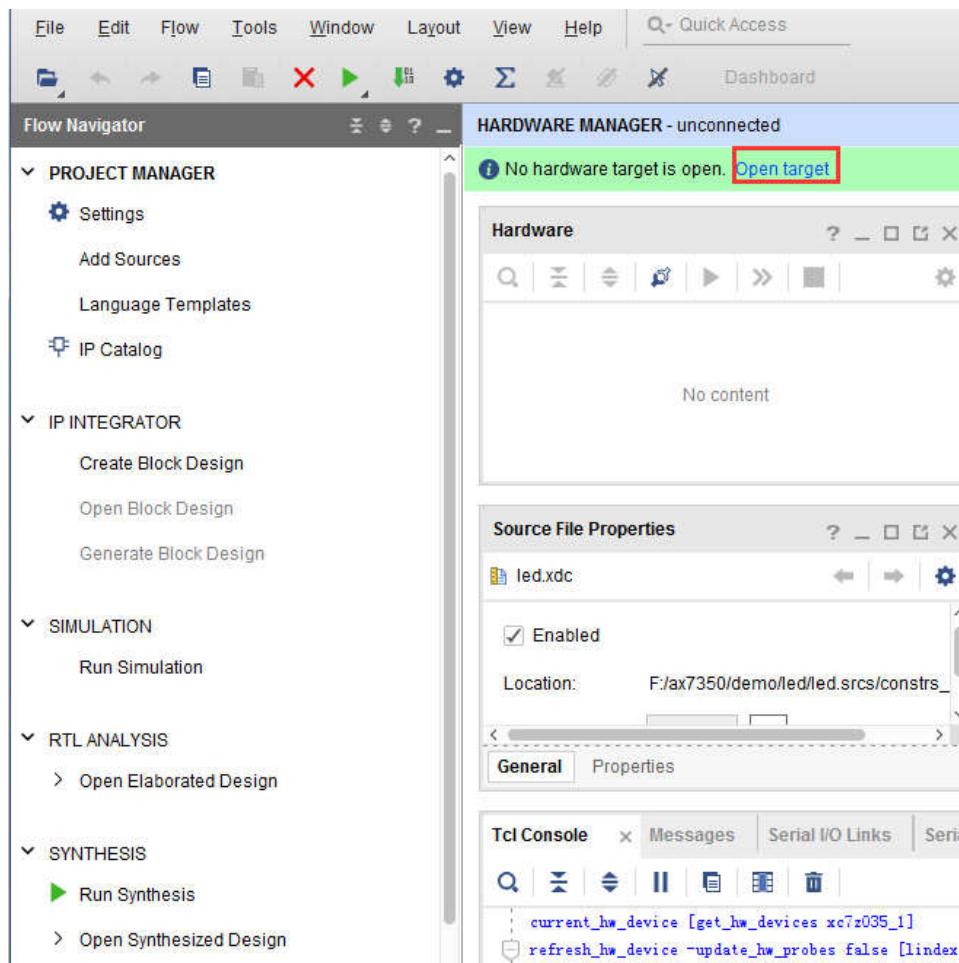
- 4) 编译中没有任何错误 ,编译完成 ,弹出一个对话框让我们选择后续操作 ,这里选项 “Open Hardware Manager” ,点击 “OK” ,当然 ,也可以选择 “Cancel” ,然后在左边导航栏选择 “Open Hardware Manager”



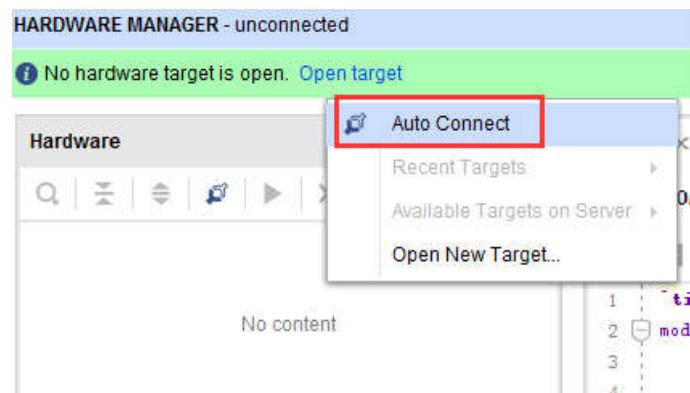
4.6 下载调试

- 1) 连接好开发板的 JTAG 接口到 PC 的 USB ,给开发板上电

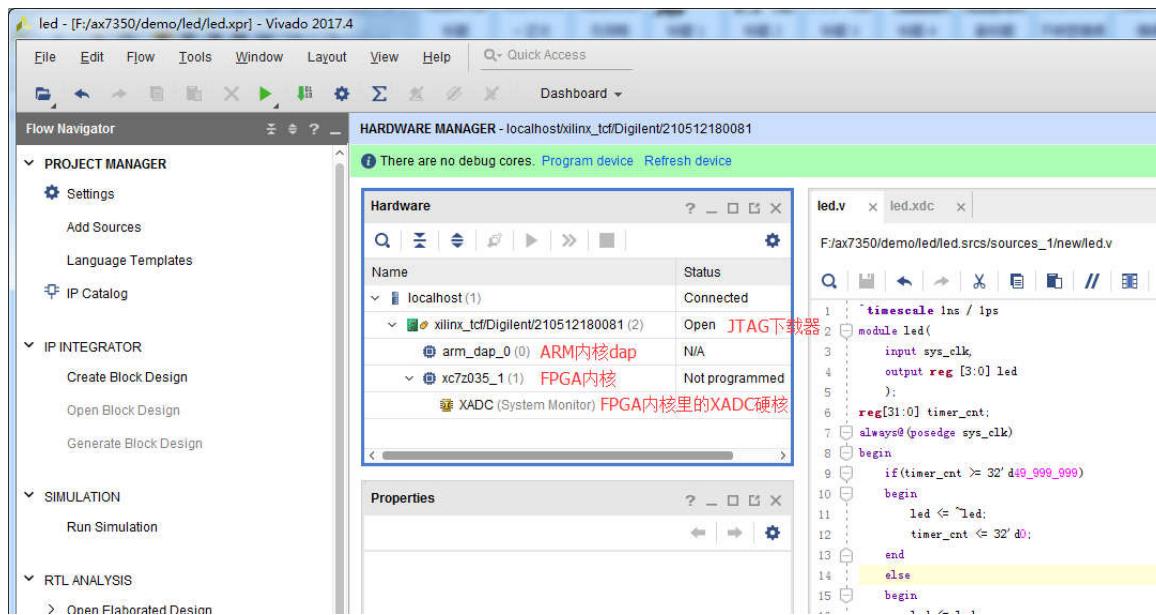
在 “HARDWARE MANAGER” 界面点击 “Open target”



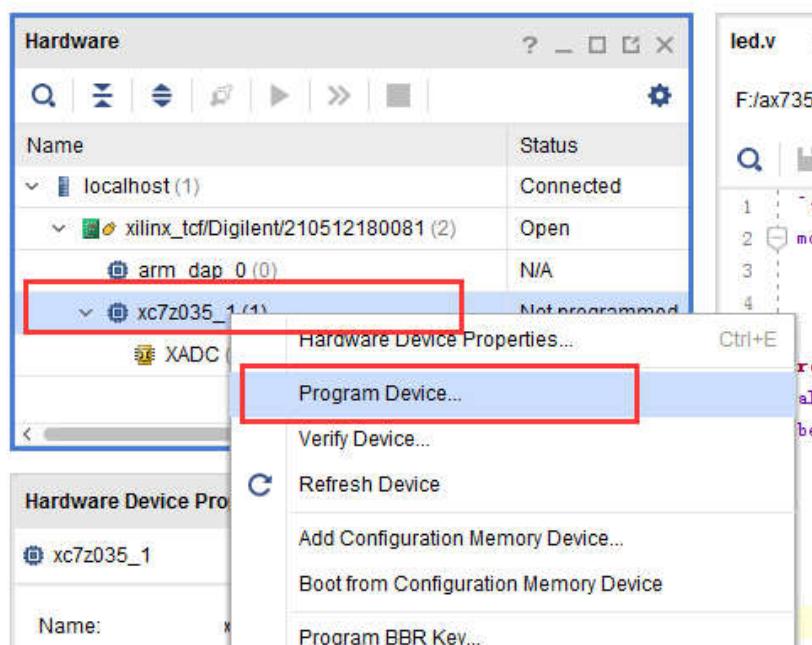
2) 点击 “Auto Connect”



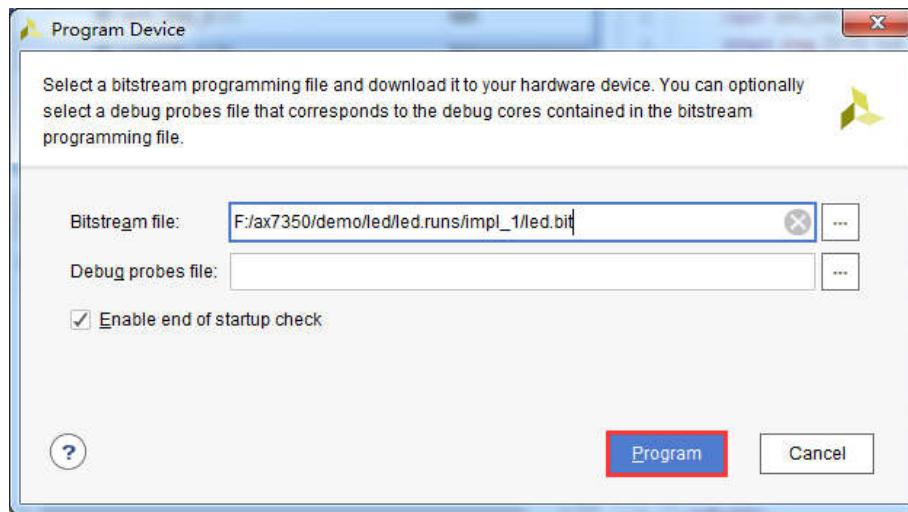
3) 可以看到 JTAG 扫描到 arm 和 FPGA 内核，还有一个 XADC，可以检测系统电压、温度



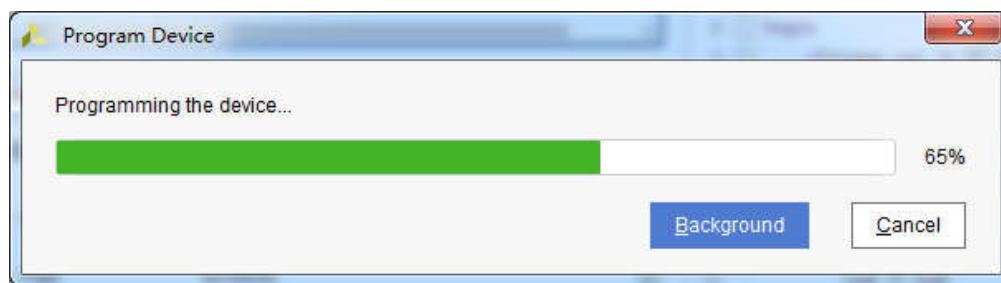
4) 选择 xc7z035_1，右键 “Program Device...”



5) 在弹出窗口中点击 “Program”



6) 等待下载



- 7) 下载完成以后，我们可以看到 4 颗 LED 开始每秒变化一次。到此为止 Vivado 简单流程体验完成。后面的章节会介绍如果把程序烧录到 Flash，需要 PS 系统的配合才能完成，只有 PL 的工程不能直接烧写 Flash。

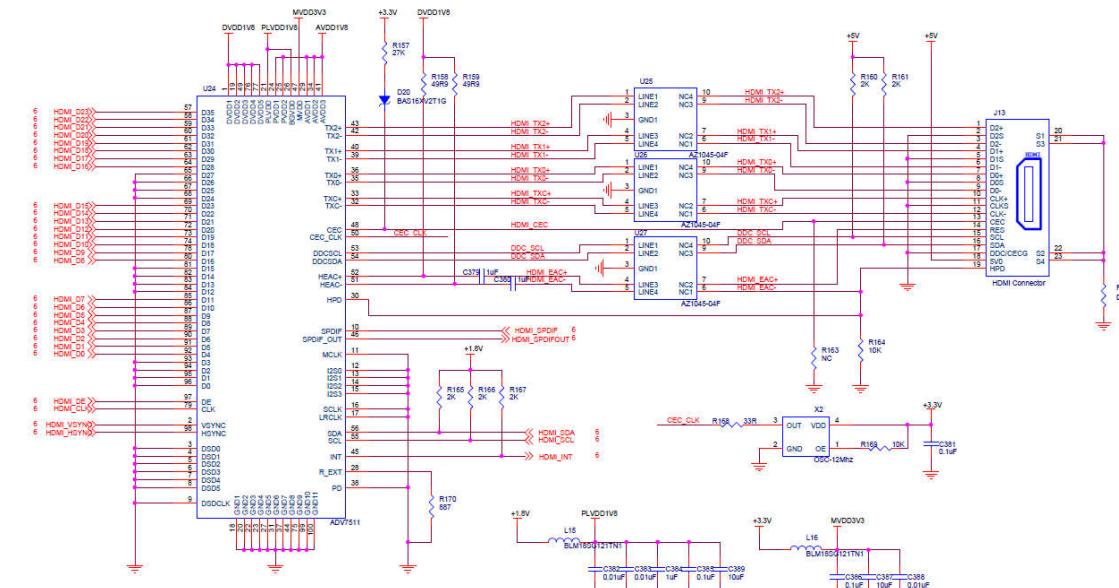
第五章 HDMI 输出实验

实验 Vivado 工程为 “hdmi_output_test”。

前面我们介绍了 led 闪灯实验，只是为了了解 Vivado 的基本开发流程，本章这个实验相对 LED 闪灯实验复杂点，做一个 HDMI 输出的彩条，这也是我们后面学习显示、视频处理的基础。实验还不涉及到 PS 系统，从实验设计可以看出如果要非常好的使用 ZYNQ 芯片，需要很好的 FPGA 基础知识。

5.1 硬件介绍

开发板使用 ADV7511 做为 HDMI 编码芯片，将 24 位 RGB 编码输出 TMDS 差分信号。ADV7511 功能强大，本实验只使用其中一小部分，将 RGB24 视频数据显示出来即可。



ADV7511 芯片需要通过 I2C 总线配置寄存器才能正常工作，从原理图中可以看出 I2C 总线连接到 PL 端的 IO，可以通过 PL 直接配置。

5.2 程序设计

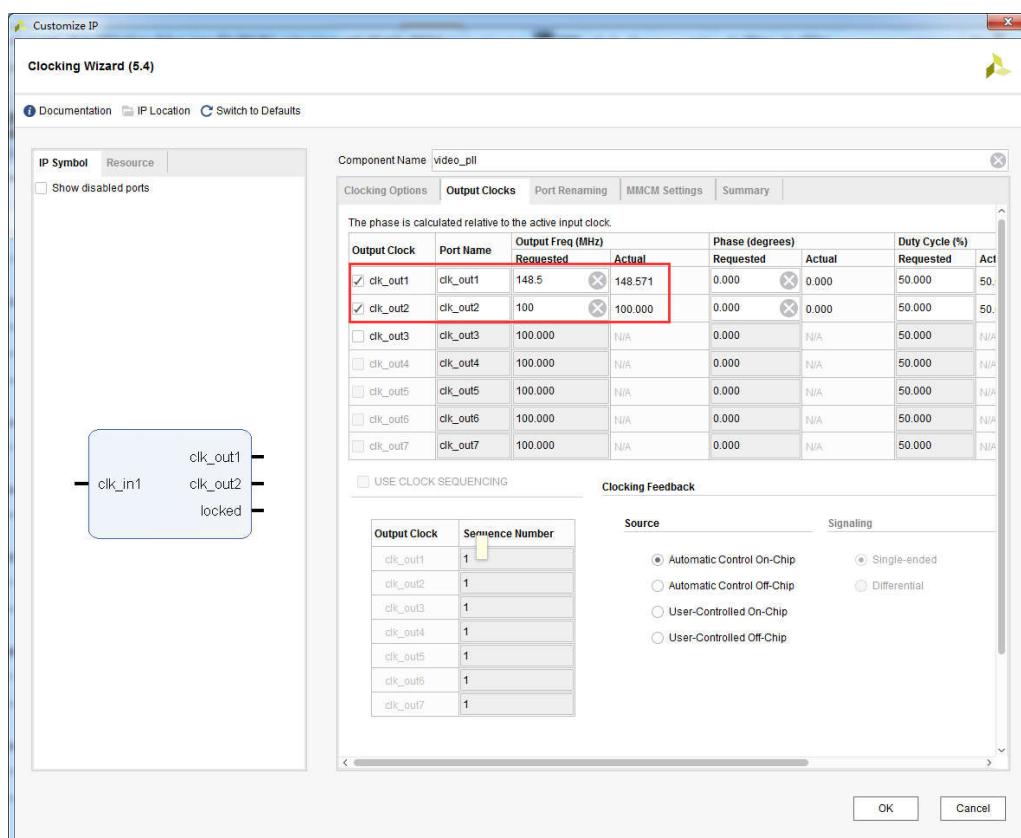
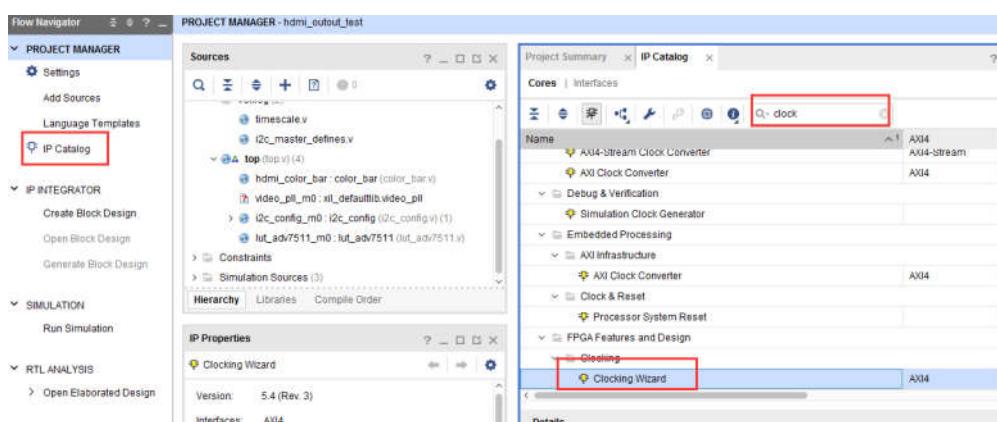
本实验实现通过 HDMI 显示彩条，实验中设计了视频时序发生和彩条发生模块 “color_bar.v” , I2C Master 寄存器配置模块 “i2c_config.v” , 配置数据查找表模块 “lut_adv7511.v”。

具体代码在这里不再一一介绍了，大家自己去看。下面针对每个模块实现的功能给大家做一下简介：

顶层模块 top.v 是项目的顶层文件，主要是实例化 4 个子模块（时钟模块 video_pll, 彩条生成模块 color_bar 和 I2C 配置模块 i2c_config 和配置查找表模块 lut_adv7511。

彩条产生模块 color_bar.v 是产生 8 种颜色的 VGA 格式的彩条，彩条分别为白、黄、青、绿、紫、红、蓝和黑。产生分辨率为 1920x1080 刷新率为 60Hz 的彩条，也就是所谓的 1080P 的高清视频图像。所以这个模块会输出 R (8 位) G (8 位) B (8 位) 图像信号、行同步、列同步和数据有效信号。

时钟模块 video_pll 调用的是一个 Xilinx 提供的时钟 IP，通过输入的系统时钟产生一个 100Mhz 时钟和一个 1080P 的像素时钟 148.5Mhz。生成时钟 IP 的方法是点击 Project Manager 目录下的 IP Catalog, 再选择 FPGA Features and Design->Clocking->Clocking Wizard 图标。



5.3 添加 XDC 约束文件

添加以下的约束文件到项目中，在约束文件里添加了时钟和 HDMI 相关的管脚。

```
#Clock signal
set_property IOSTANDARD LVCMOS18 [get_ports sys_clk]
set_property PACKAGE_PIN J14 [get_ports sys_clk]
create_clock -period 20.000 -name sys_clk -waveform {0.000 10.000} [get_ports sys_clk]

set_property PACKAGE_PIN K13 [get_ports hdmi_clk]
set_property PACKAGE_PIN G16 [get_ports {hdmi_d[0]}]
set_property PACKAGE_PIN E16 [get_ports {hdmi_d[1]}]
set_property PACKAGE_PIN J15 [get_ports {hdmi_d[2]}]
set_property PACKAGE_PIN E15 [get_ports {hdmi_d[3]}]
set_property PACKAGE_PIN F15 [get_ports {hdmi_d[4]}]
set_property PACKAGE_PIN G15 [get_ports {hdmi_d[5]}]
set_property PACKAGE_PIN F14 [get_ports {hdmi_d[6]}]
set_property PACKAGE_PIN H14 [get_ports {hdmi_d[7]}]
set_property PACKAGE_PIN J13 [get_ports {hdmi_d[8]}]
set_property PACKAGE_PIN K12 [get_ports {hdmi_d[9]}]
set_property PACKAGE_PIN B11 [get_ports {hdmi_d[10]}]
set_property PACKAGE_PIN C12 [get_ports {hdmi_d[11]}]
set_property PACKAGE_PIN D13 [get_ports {hdmi_d[12]}]
set_property PACKAGE_PIN A12 [get_ports {hdmi_d[13]}]
set_property PACKAGE_PIN C13 [get_ports {hdmi_d[14]}]
set_property PACKAGE_PIN A13 [get_ports {hdmi_d[15]}]
set_property PACKAGE_PIN D14 [get_ports {hdmi_d[16]}]
set_property PACKAGE_PIN D15 [get_ports {hdmi_d[17]}]
set_property PACKAGE_PIN A14 [get_ports {hdmi_d[18]}]
set_property PACKAGE_PIN B14 [get_ports {hdmi_d[19]}]
set_property PACKAGE_PIN A15 [get_ports {hdmi_d[20]}]
set_property PACKAGE_PIN B15 [get_ports {hdmi_d[21]}]
set_property PACKAGE_PIN D16 [get_ports {hdmi_d[22]}]
set_property PACKAGE_PIN B16 [get_ports {hdmi_d[23]}]
set_property PACKAGE_PIN K15 [get_ports hdmi_de]
set_property PACKAGE_PIN C11 [get_ports hdmi_hs]
#set_property PACKAGE_PIN C17 [get_ports hdmi_nreset]
set_property PACKAGE_PIN B12 [get_ports hdmi_vs]
set_property PACKAGE_PIN A17 [get_ports hdmi_scl]
set_property PACKAGE_PIN C16 [get_ports hdmi_sda]

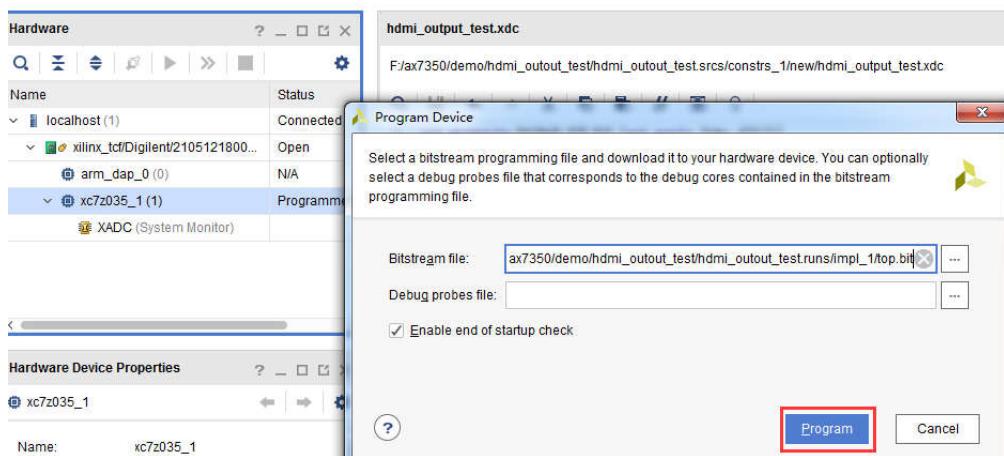
set_property IOSTANDARD LVCMOS18 [get_ports hdmi_scl]
set_property IOSTANDARD LVCMOS18 [get_ports hdmi_sda]
set_property IOSTANDARD LVCMOS18 [get_ports {hdmi_d[*]}]
set_property IOSTANDARD LVCMOS18 [get_ports hdmi_clk]
set_property IOSTANDARD LVCMOS18 [get_ports hdmi_de]
set_property IOSTANDARD LVCMOS18 [get_ports hdmi_vs]
set_property IOSTANDARD LVCMOS18 [get_ports hdmi_hs]
```

```
set_property PULLUP true [get_ports hdmi_scl]
set_property PULLUP true [get_ports hdmi_sda]

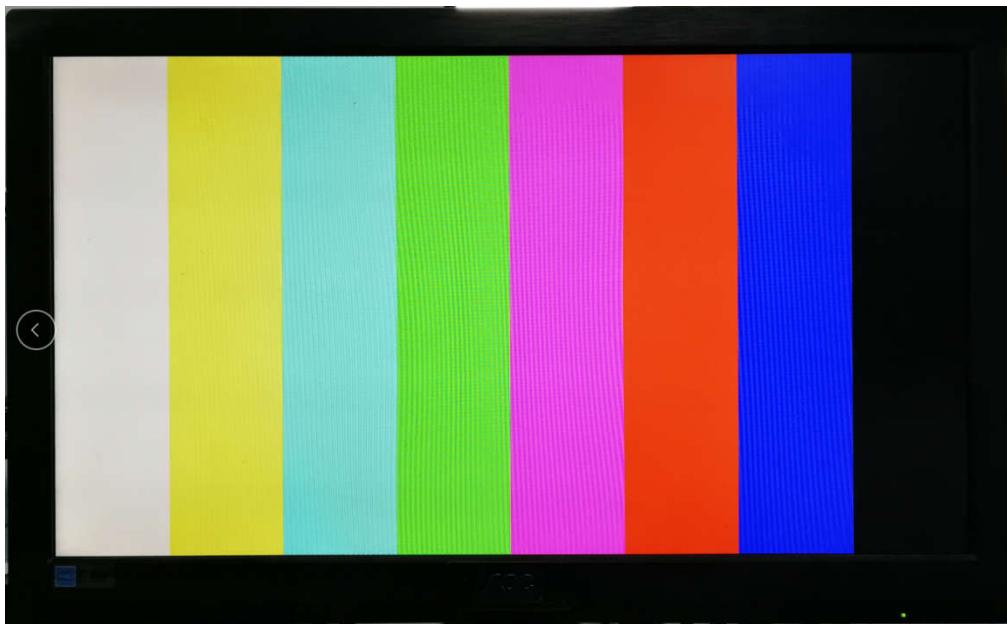
set_property SLEW FAST [get_ports {hdmi_d[*]}]
set_property DRIVE 8 [get_ports {hdmi_d[*]}]
set_property SLEW FAST [get_ports hdmi_clk]
set_property SLEW FAST [get_ports hdmi_de]
set_property SLEW FAST [get_ports hdmi_hs]
set_property SLEW FAST [get_ports hdmi_scl]
set_property SLEW FAST [get_ports hdmi_sda]
set_property SLEW FAST [get_ports hdmi_vs]
```

5.4 下载调试

保存工程并编译生成 bit 文件，连接 HDMI 接口到 HDMI 显示器，需要注意，这里使用 1920x1080@60Hz，请确保自己的显示器支持这个分辨率。



下载后显示器显示如下图像



5.5 实验总结

本实验初步接触到视频显示，涉及到视频知识，这不是 zynq 学习的重点，所以没有详细介绍，但 zynq 在视频处理领域用途广泛，需要学习者有良好的基础知识。实验中仅仅使用 PL 来驱动 HDMI 芯片，包括 I2C 寄存器配置，当然 I2C 的配置还是使用 PS 来配置比较合适。

第六章 可编程时钟 SI5338 实验

实验 Vivado 工程为 “si5338_in3_pl_test”。

开发板上有一个 4 通道差分输出可编程时钟芯片 SI5338，可以为 PL、收发器模块提供高质量的时钟，而且输出频率是可以动态编程的。

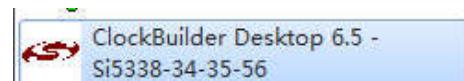
6.1 硬件介绍

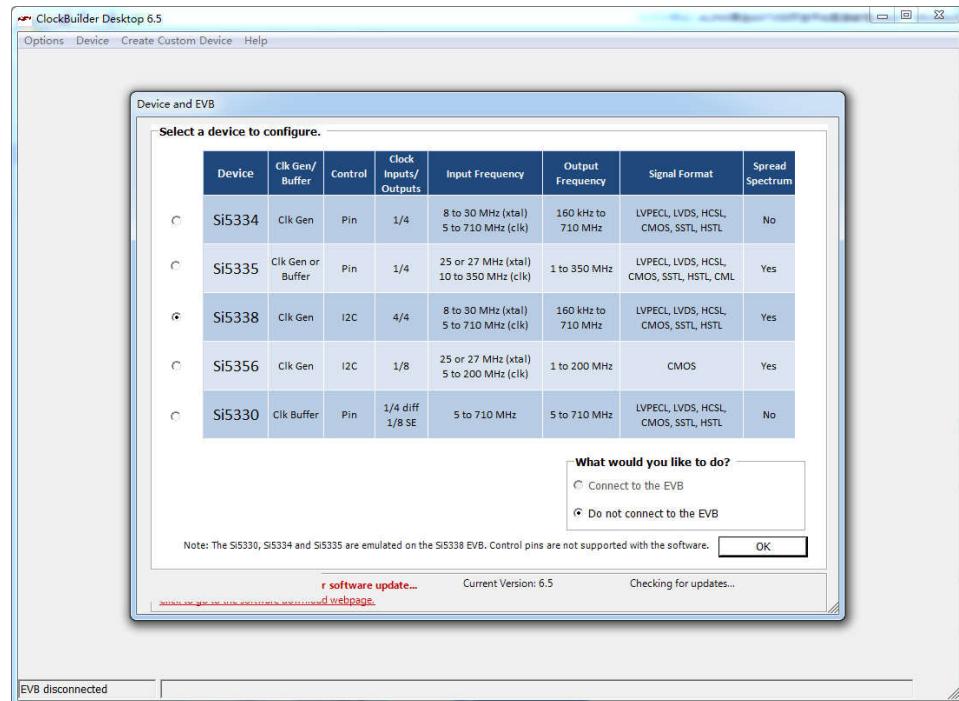
6.2 程序设计

SI5338 通过 I2C 来配置，配置寄存器的来源是通过软件 “ClockBuilder Desktop”，这个软件是 Silicon Laboratories 公司提供。然后我们使用一个 Python 脚本把 “ClockBuilder Desktop” 软件生成的 C 语言头文件 “register_map.h” 转换为一个 rom 初始化文件。“si5338.vhd” 模块是用来配置 SI5338，通过 LED 闪烁来判断 si5338 输出时钟是否正常。

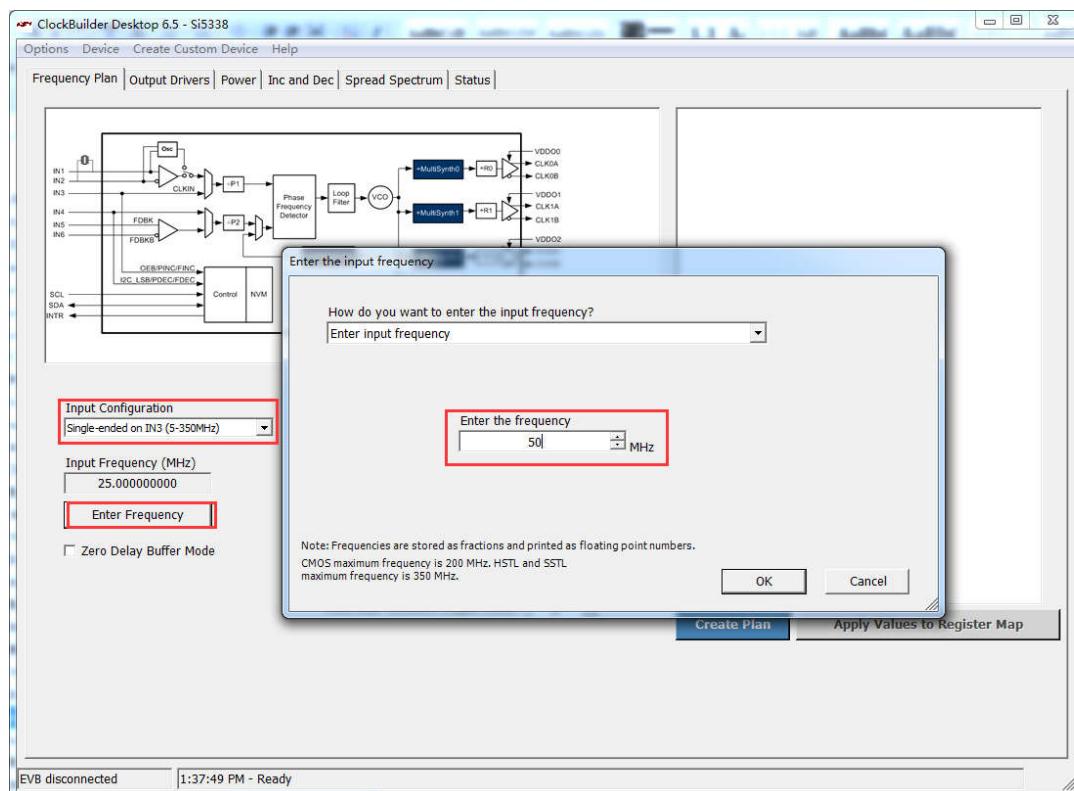
6.3 ClockBuilder Desktop 软件使用简介

启动 “ClockBuilder Desktop” 软件，器件选择 Si5338，选择 “Do not connect to the EVB”

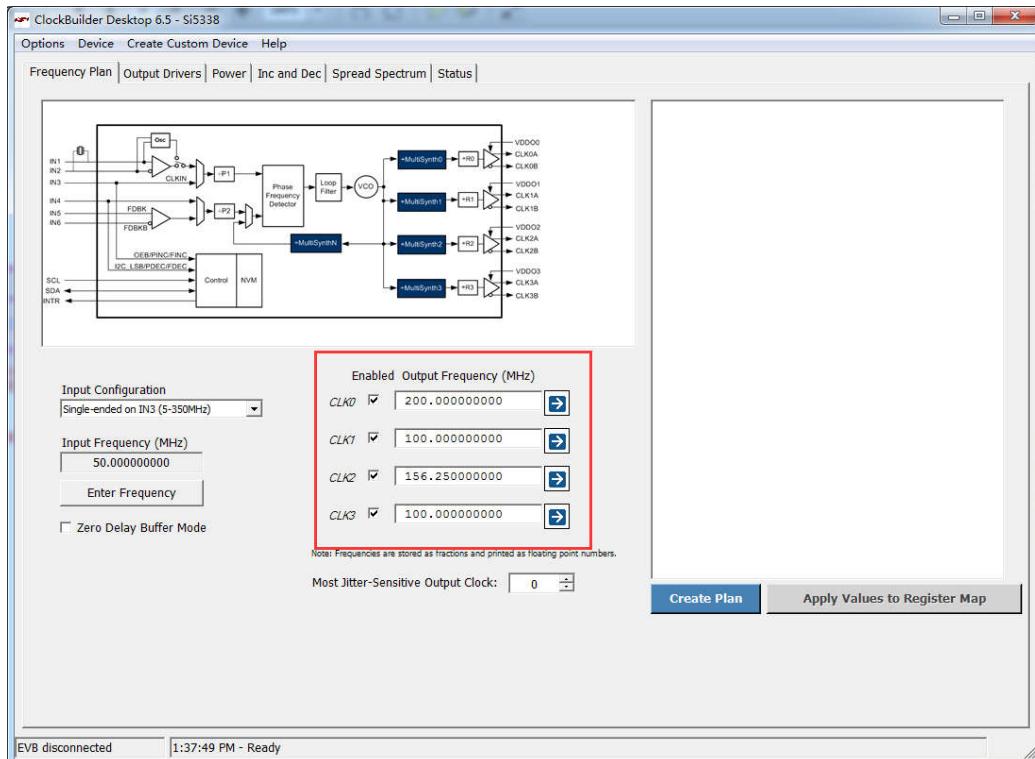




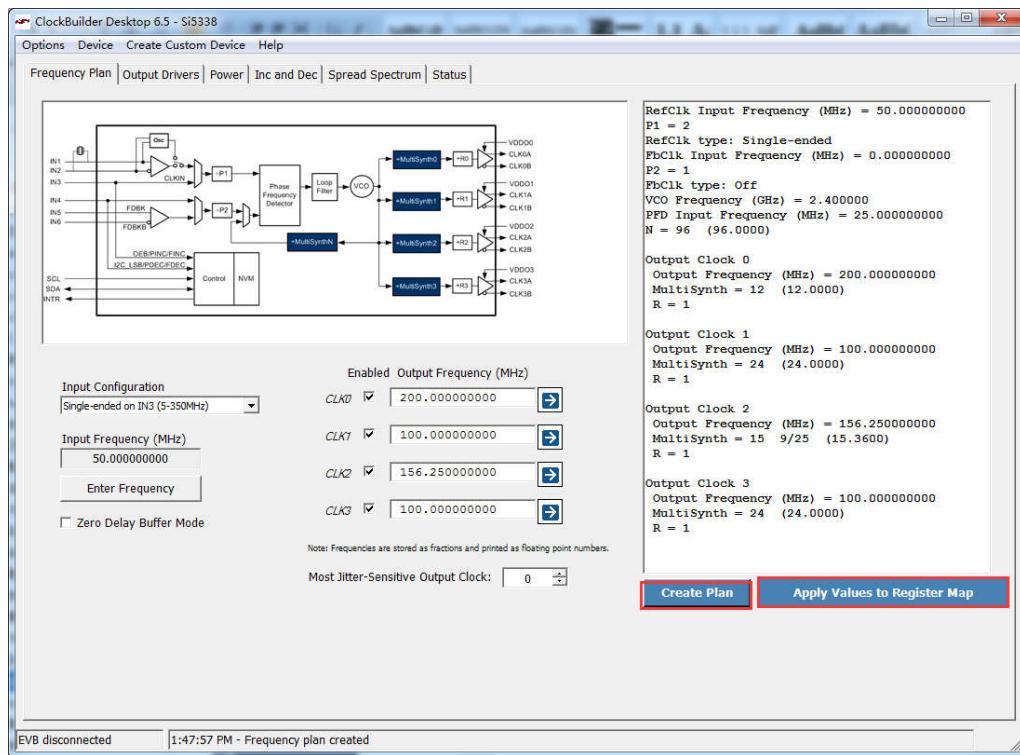
输入配置 “Input Configuration” 选择 “Single-ended on IN3(5-350MHz)” , 输入频率 “Enter the frequency” 填写 50 , 点击 OK



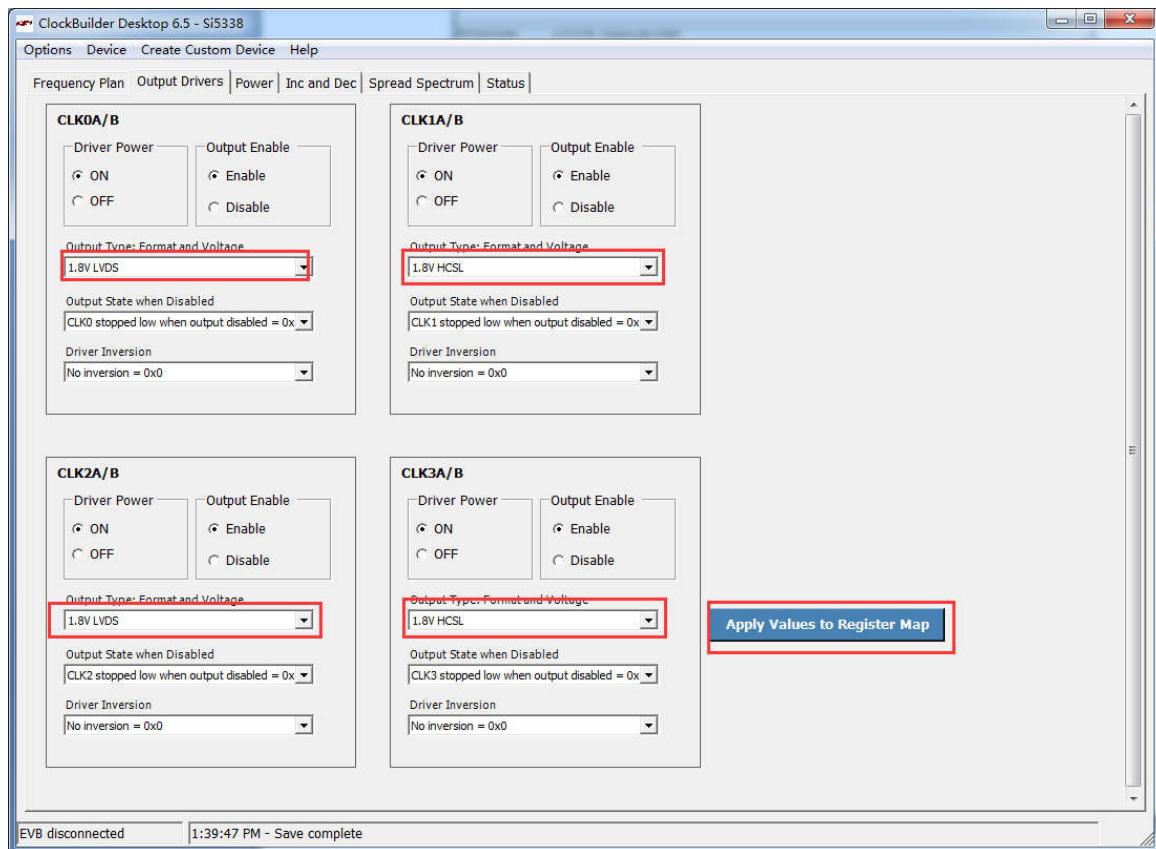
配置 4 通道输出频率 , CLK0 为 200MHz , 用于 DDR3 , CLK1 和 CLK3 100MHz 用于 PCIe , CLK2 为 156.25MHz , 用于 SFP 光通信。



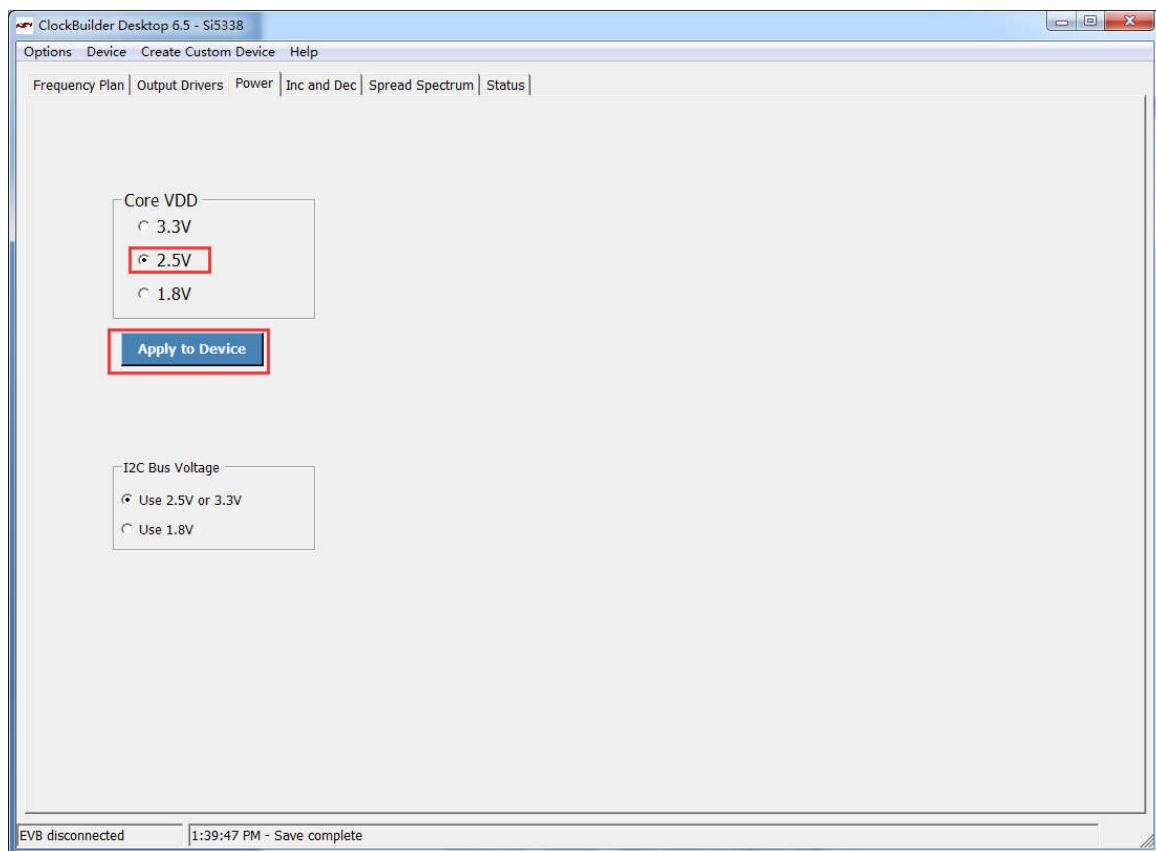
点击 “Create Plan” 后再点击 “Apply Values to Register Map”



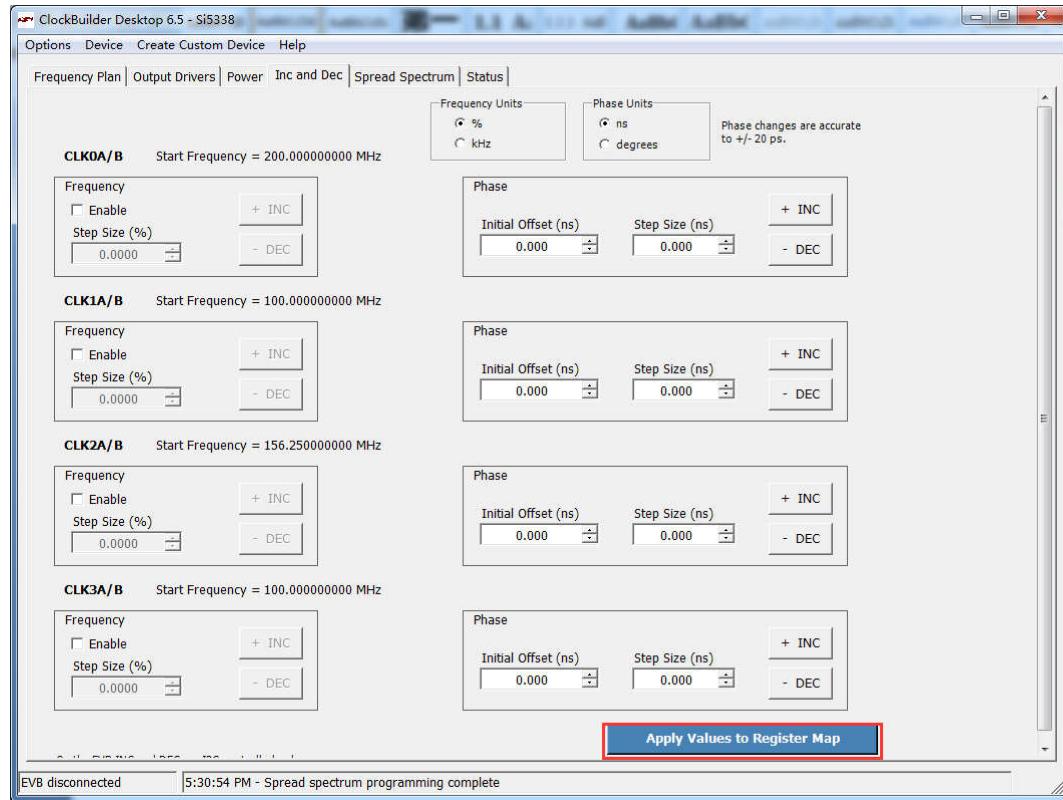
在 “Output Drivers” 标签页 ,设置 CLK0A/B 和 CLK2A/B 输出为 1.8V LVDS ,CLK1A/B 和 CLK3A/B 输出为 1.8V HCSL , 其他默认 , 点击 “Apply Values to Register Map”



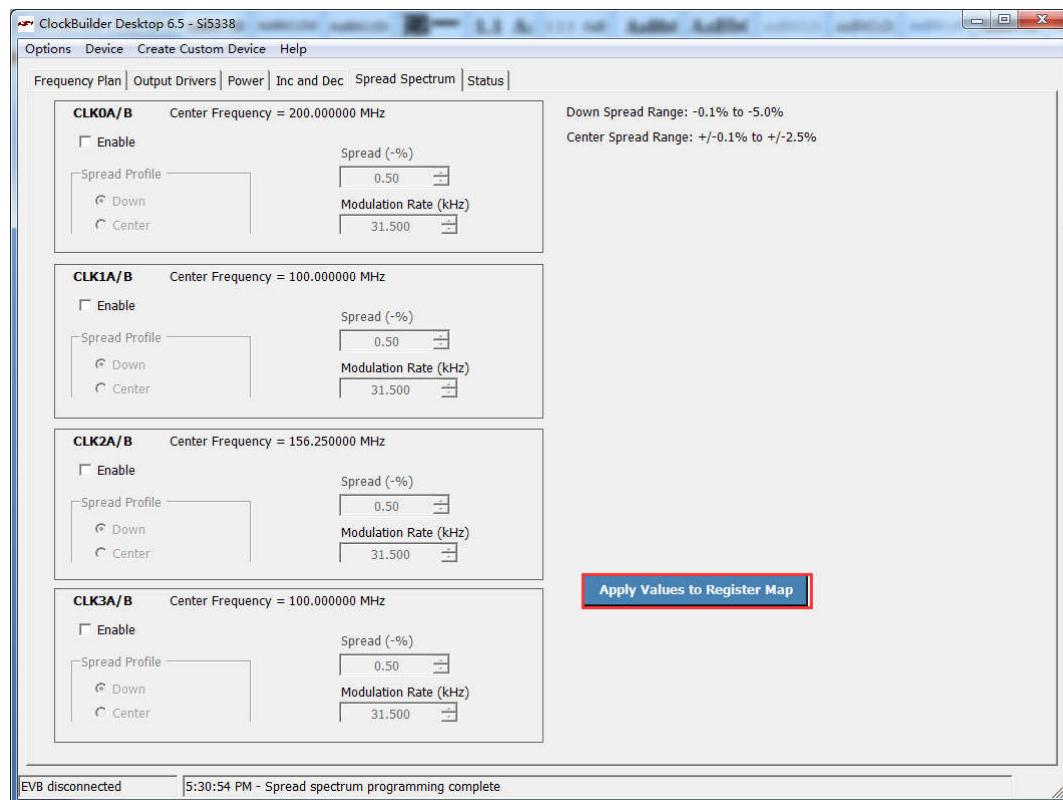
在“Power”页，默认配置，点击“Apply to Device”



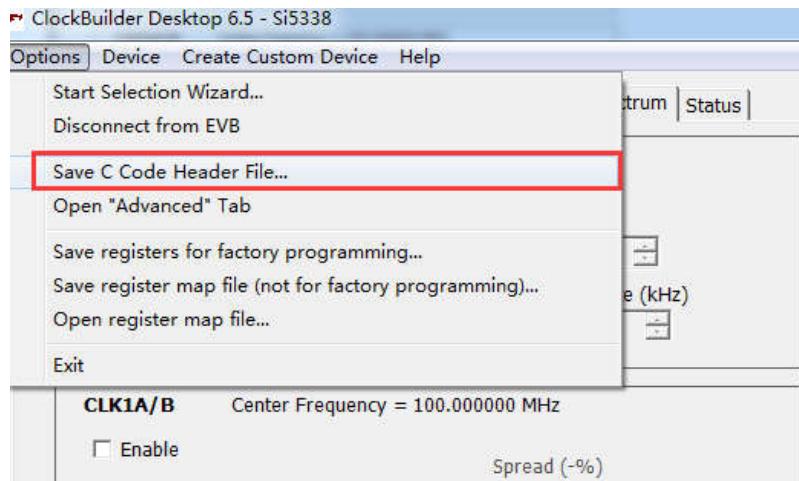
"Inc and Dec" 页点击 "Apply Values to Register Map"



"Spread Spectrum" 页点击 "Apply Values to Register Map"

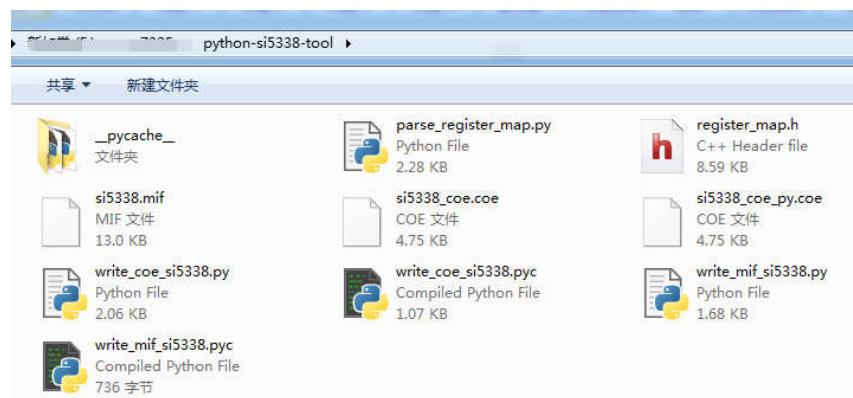


点击 “Options->Save C Code Header File...” ,保存一个 “register_map.h” 文件，这个文件后面会用到。



6.4 使用 Python 转换寄存器配置文件

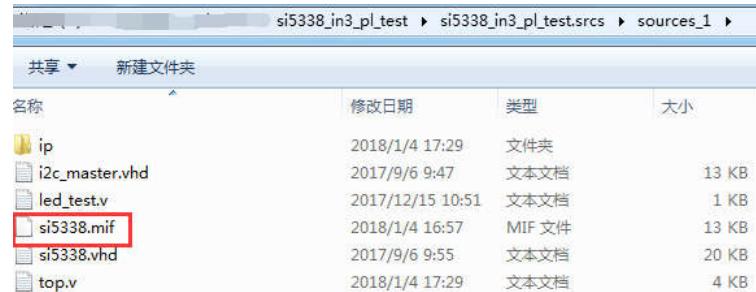
芯驿电子提供了一个 Python 脚本，用于把 “register_map.h” 文件转换为一个 mif 文件，mif 文件可以被 VHDL 读取并初始化一个 ROM，用于配置 SI5338 的寄存器，安装好 Python 环境后可以直接点击 “parse_register_map.py” ，这个时候会生成一个 “si5338.mif” 文件。



6.5 Vivado 工程建立

在前面的教程中详细介绍了如何建立一个工程，这里不再复述，可以直接参考芯驿电子提供的已经建立好的工程。

在工程的源代码目录可以看到 “si5338.mif” 文件，这个文件就是通过 Python 生成的 ROM 初始化文件。



6.6 下载调试

生成 bit 文件以后，使用 JTAG 下载到开发板中，可以看到 LED1、LED2、LED3 都会同频闪烁，LED4 常亮，按 PL_KEY1 复位 LED1、LED2、LED3 状态。

6.7 实验总结

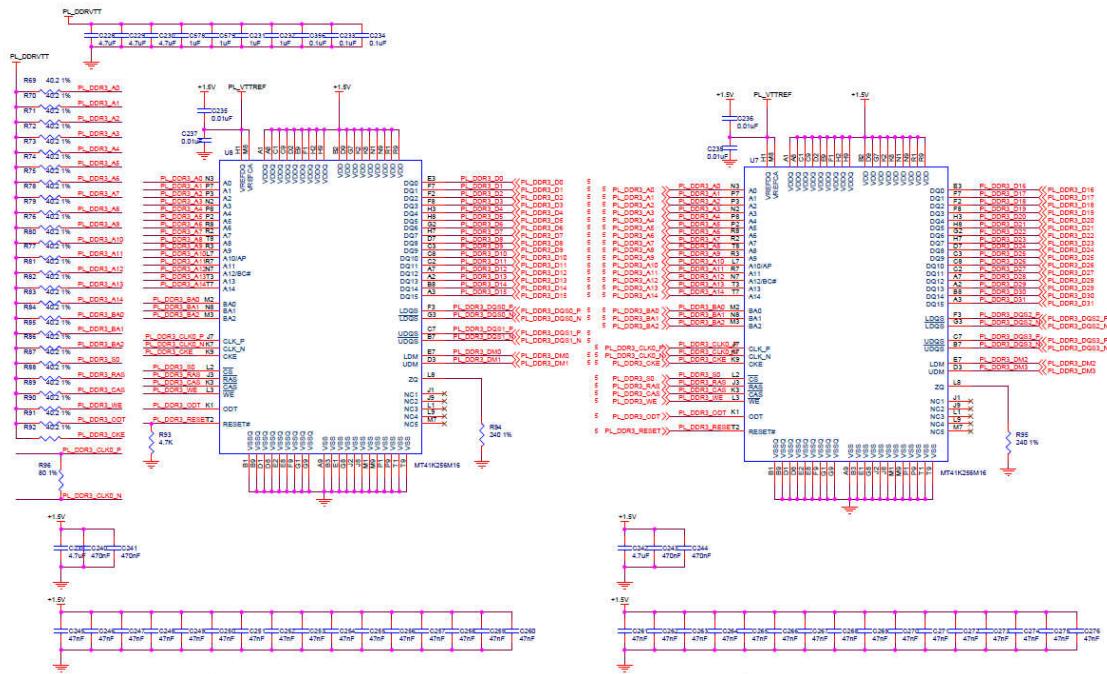
SI5338 为 FPGA 提供了 3 路差分时钟，而是是可配置的，在后面的高速收发器 GTX 实验、PCIe 实验中都会用到，本实验 I2C 同样使用 PL 来配置，后面的实验会使用 PS 来配置。

第七章 PL 端 DDR3 读写测试实验

实验 Vivado 工程为 “ddr3_pl_test”。

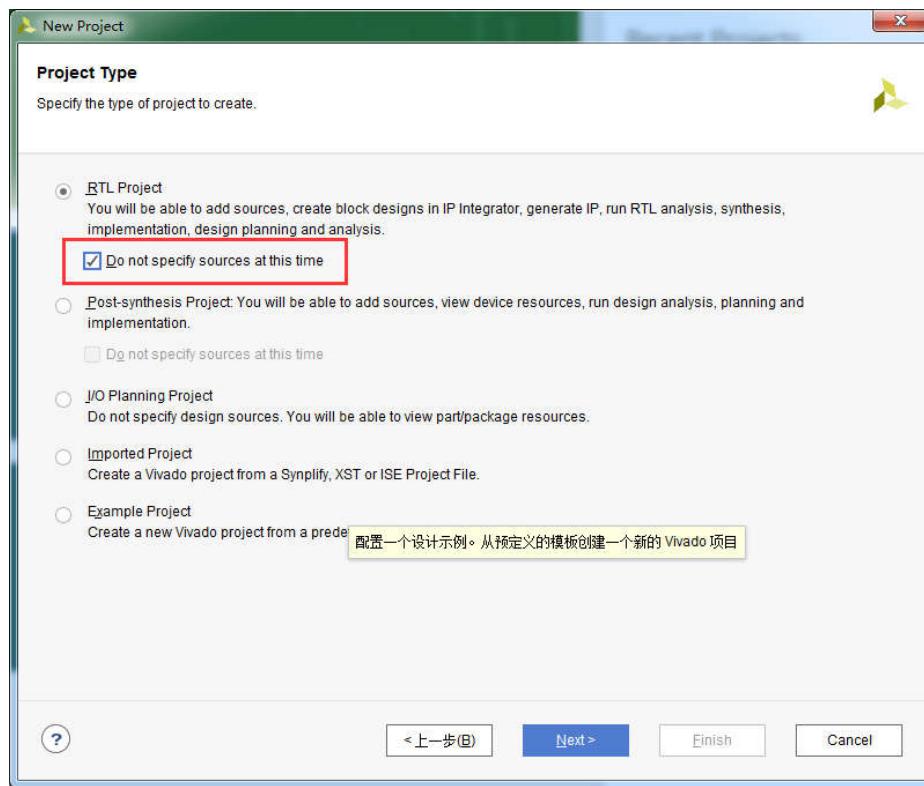
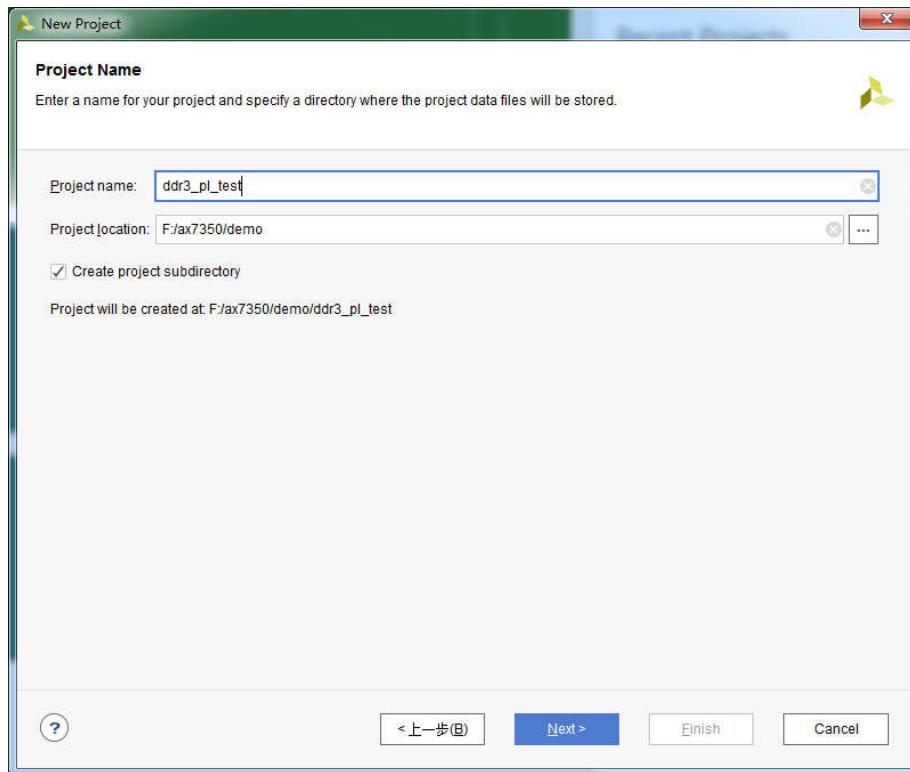
7.1 硬件介绍

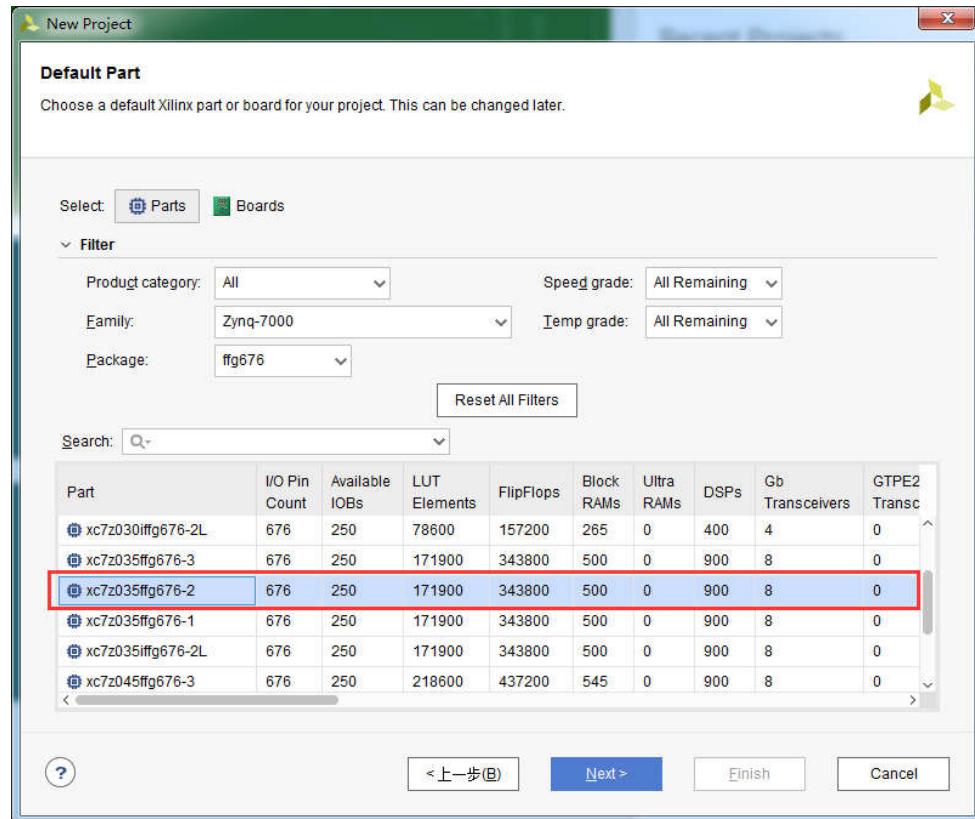
开发板的 PL 端有 2 颗 16bit ddr3，这很大程度方便我们移植以前的 FPGA 工程到 ZYNQ 系统中，同时也提供了更大的带宽。



7.2 Vivado 工程建立

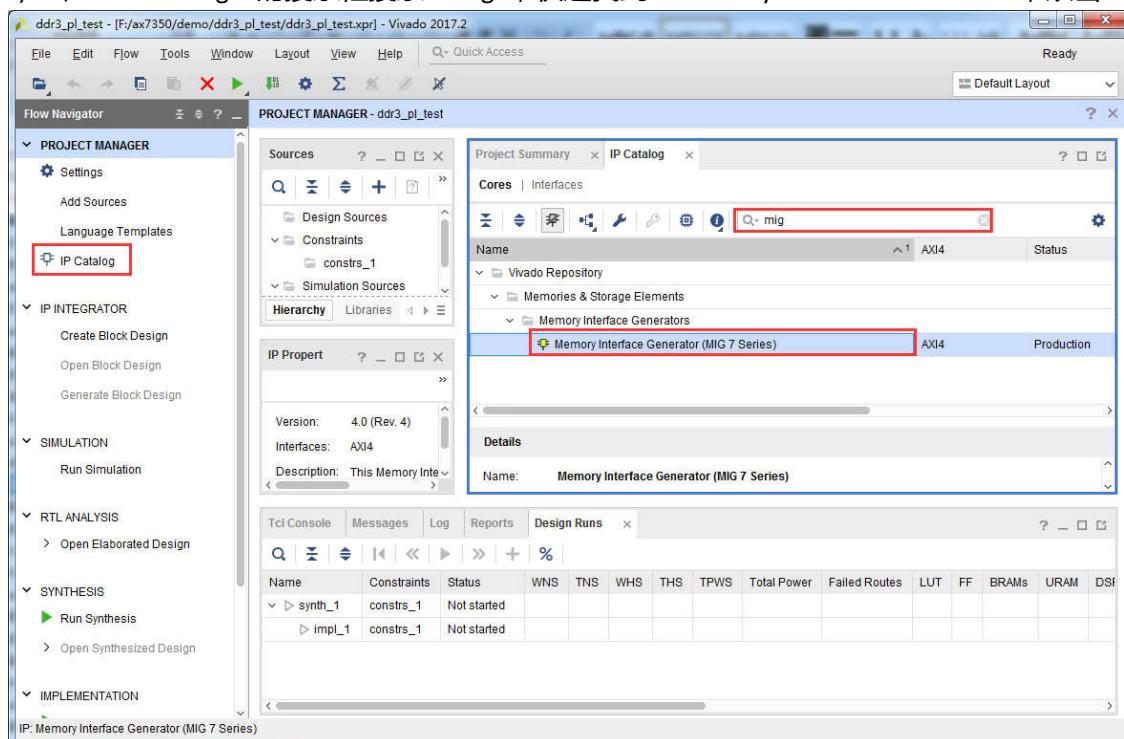
7.2.1 创建一个 PL 端 ddr3 测试工程



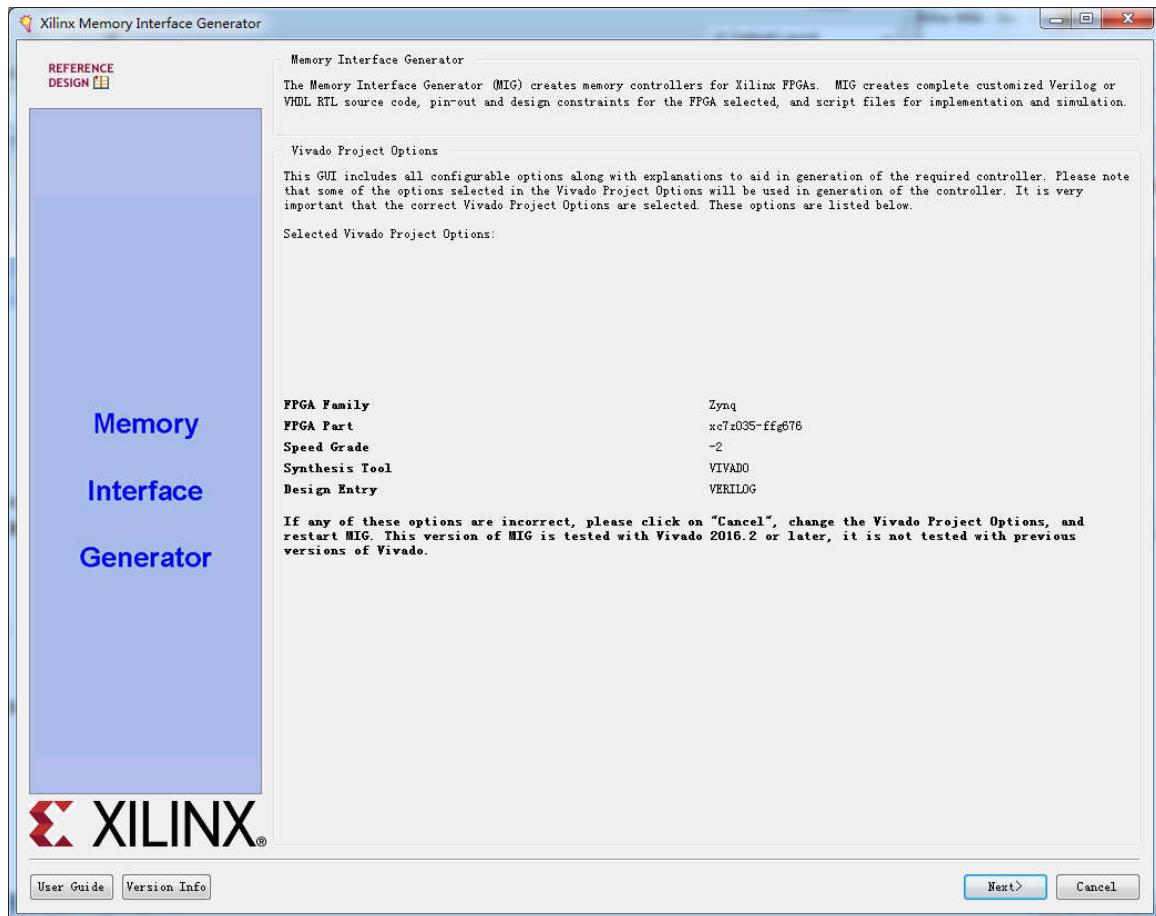


7.2.2 配置 ddr3 IP

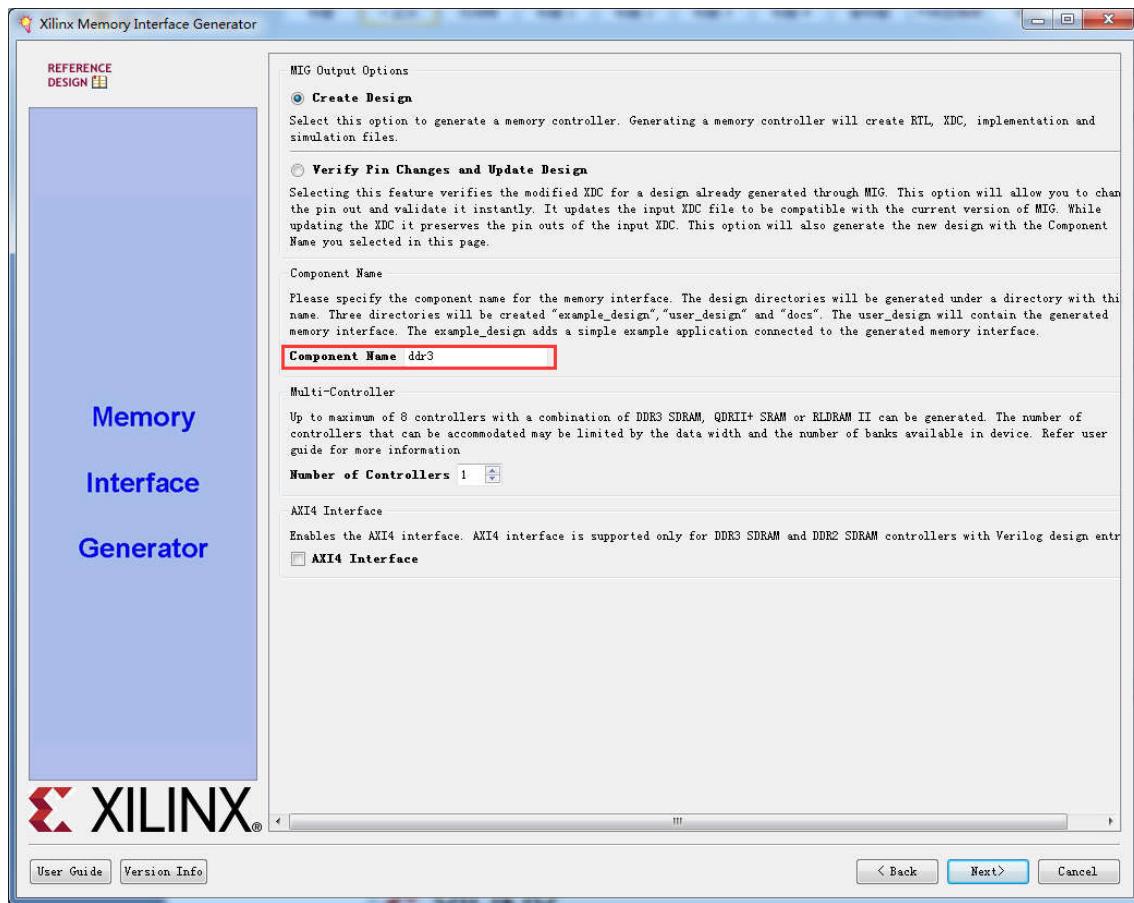
1) 在 “IP Catalog” 的搜索框搜索 “mig”，快速找到 “Memory Interface Generator”，双击



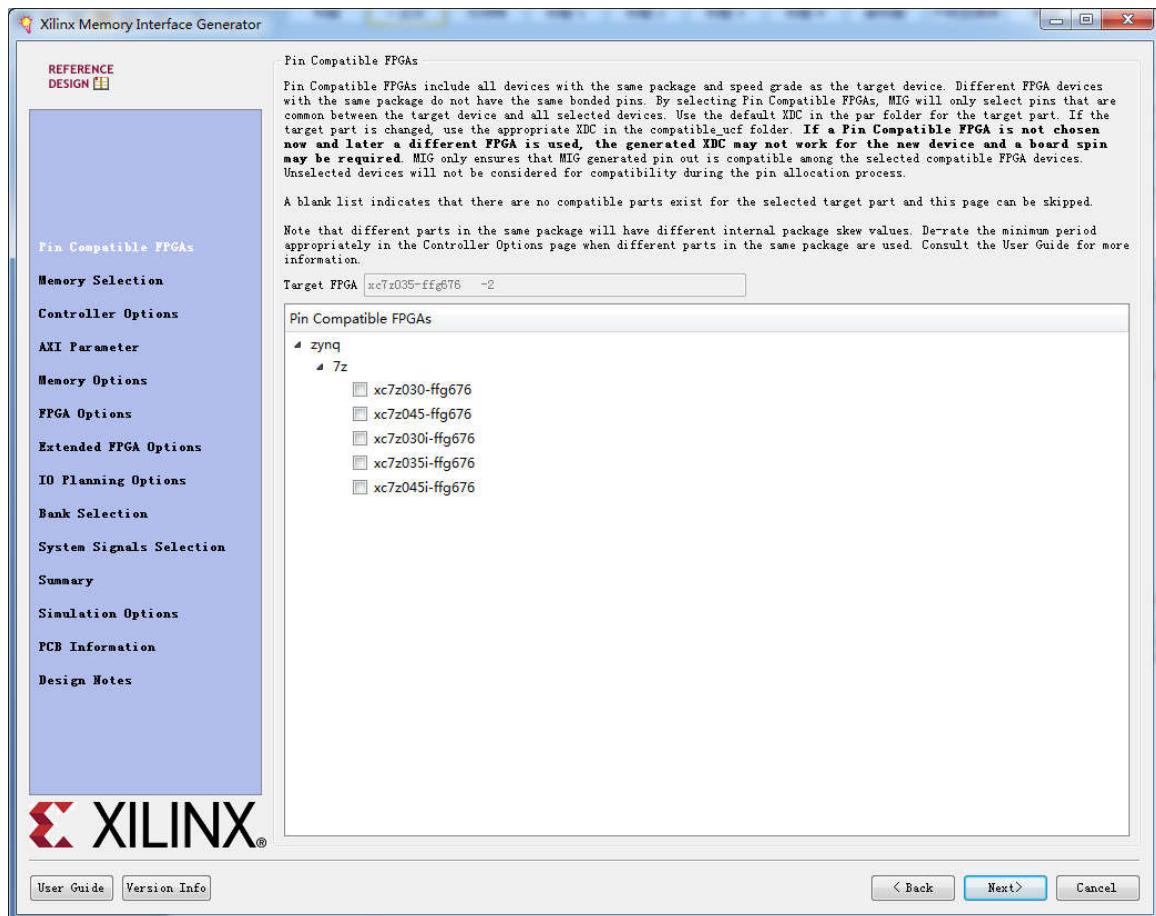
2) 点击 “Next”



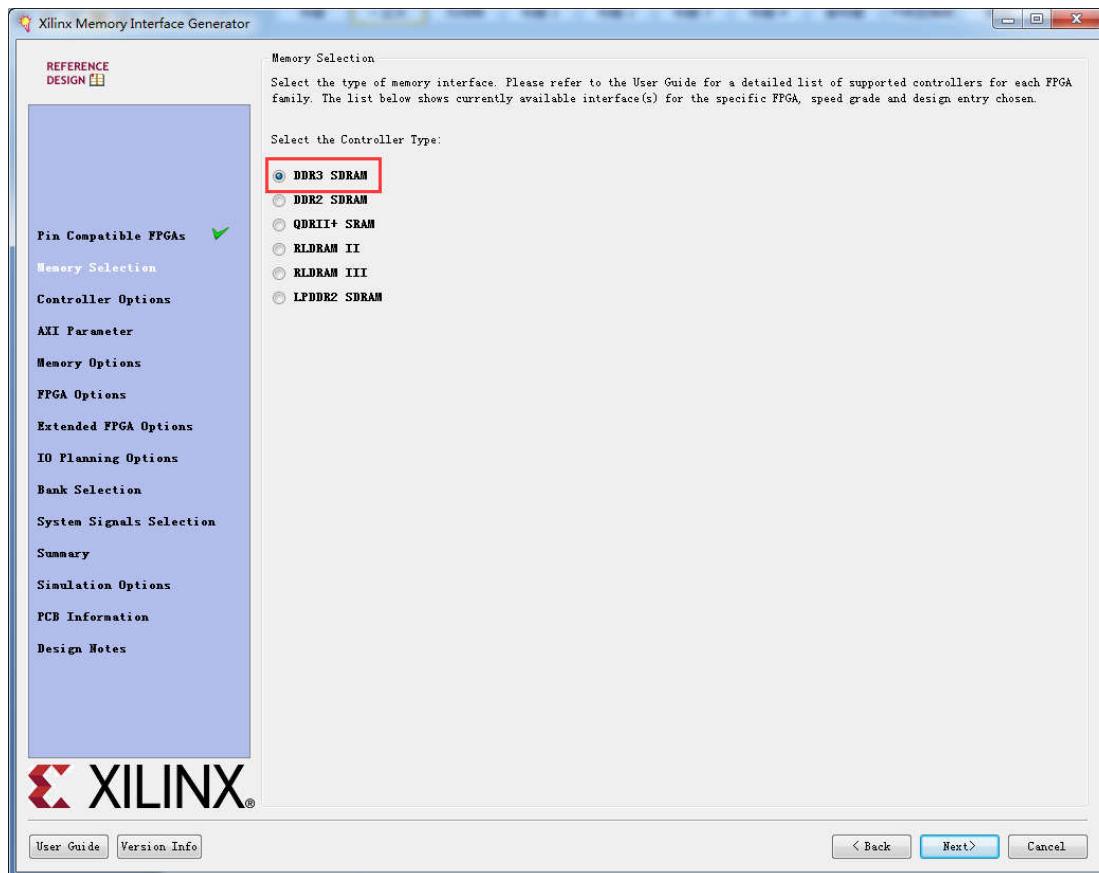
3) “Component Name” 修改为 “ddr3” , 以后我们例化 ddr3 就可以 , 点击 “Next”



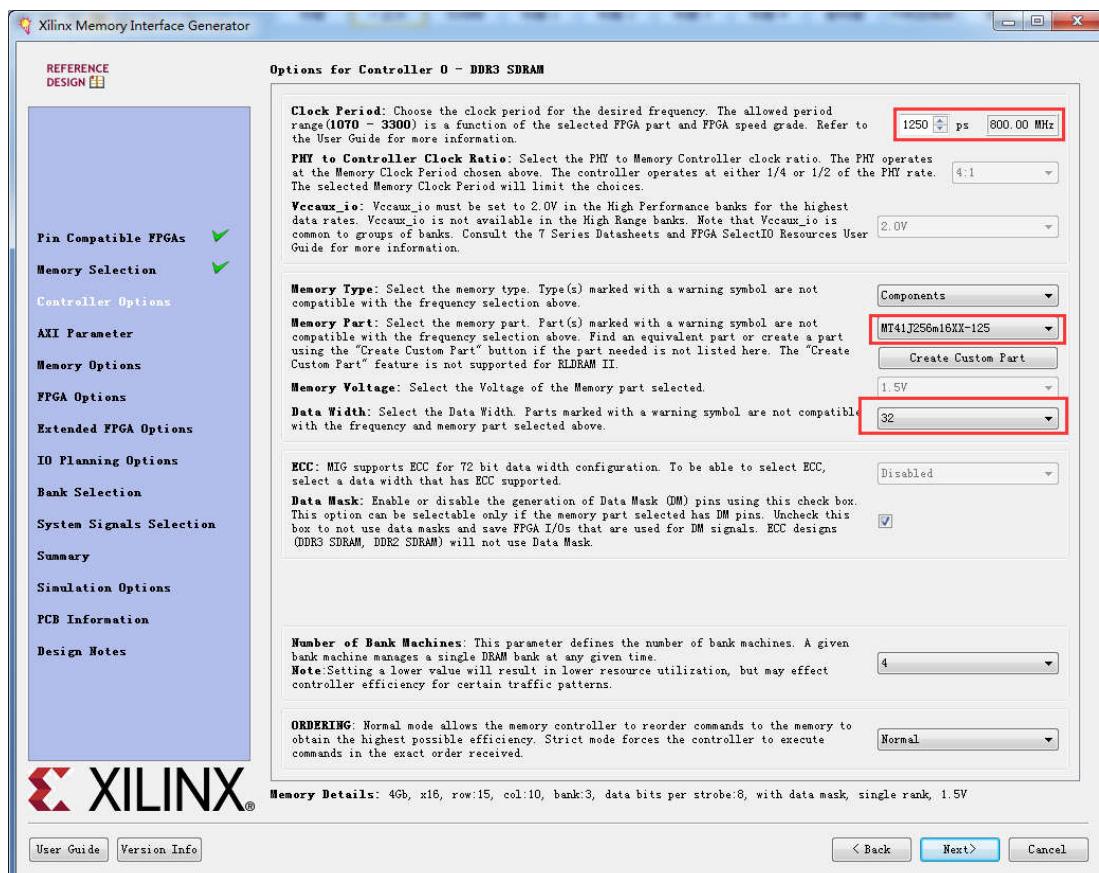
4) 点击 “Next”



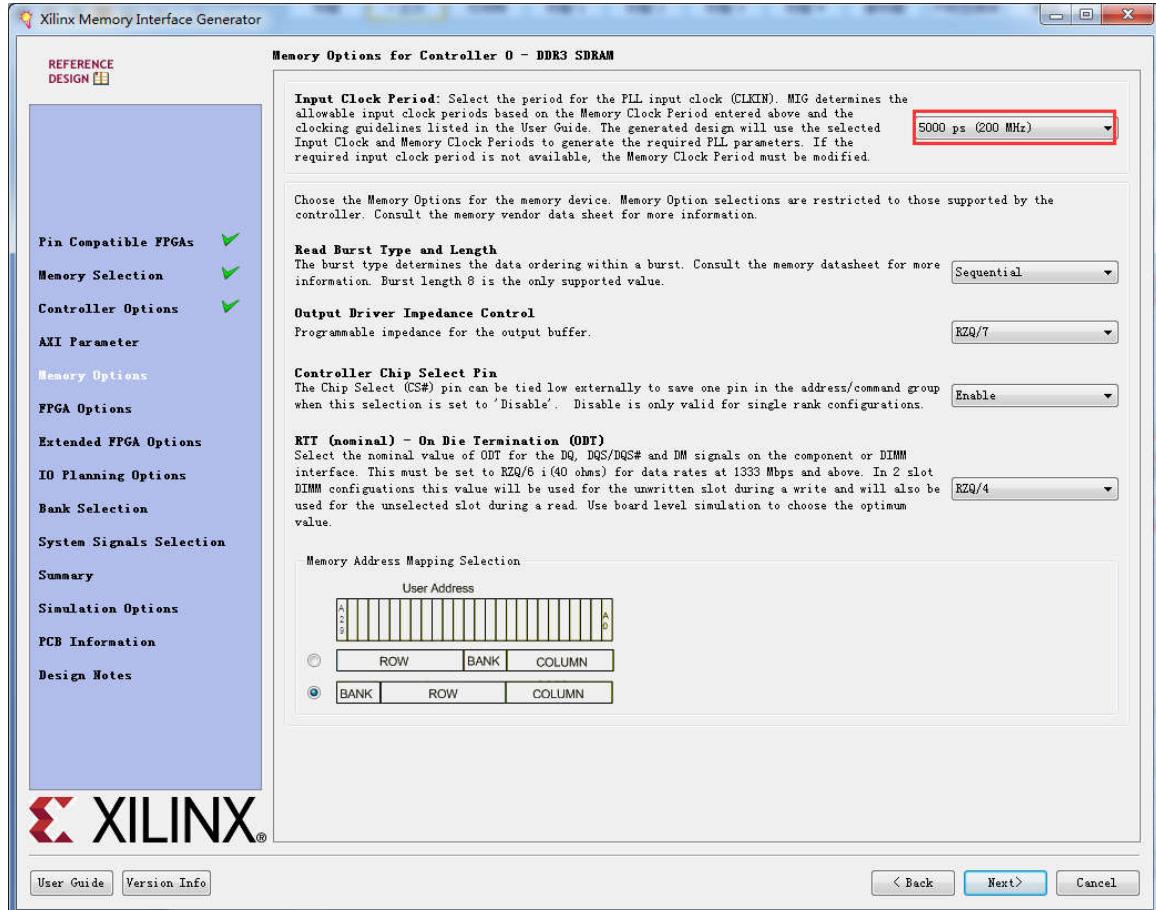
5) 控制器类型选择 “DDR3 SDRAM” , 点击 “Next”



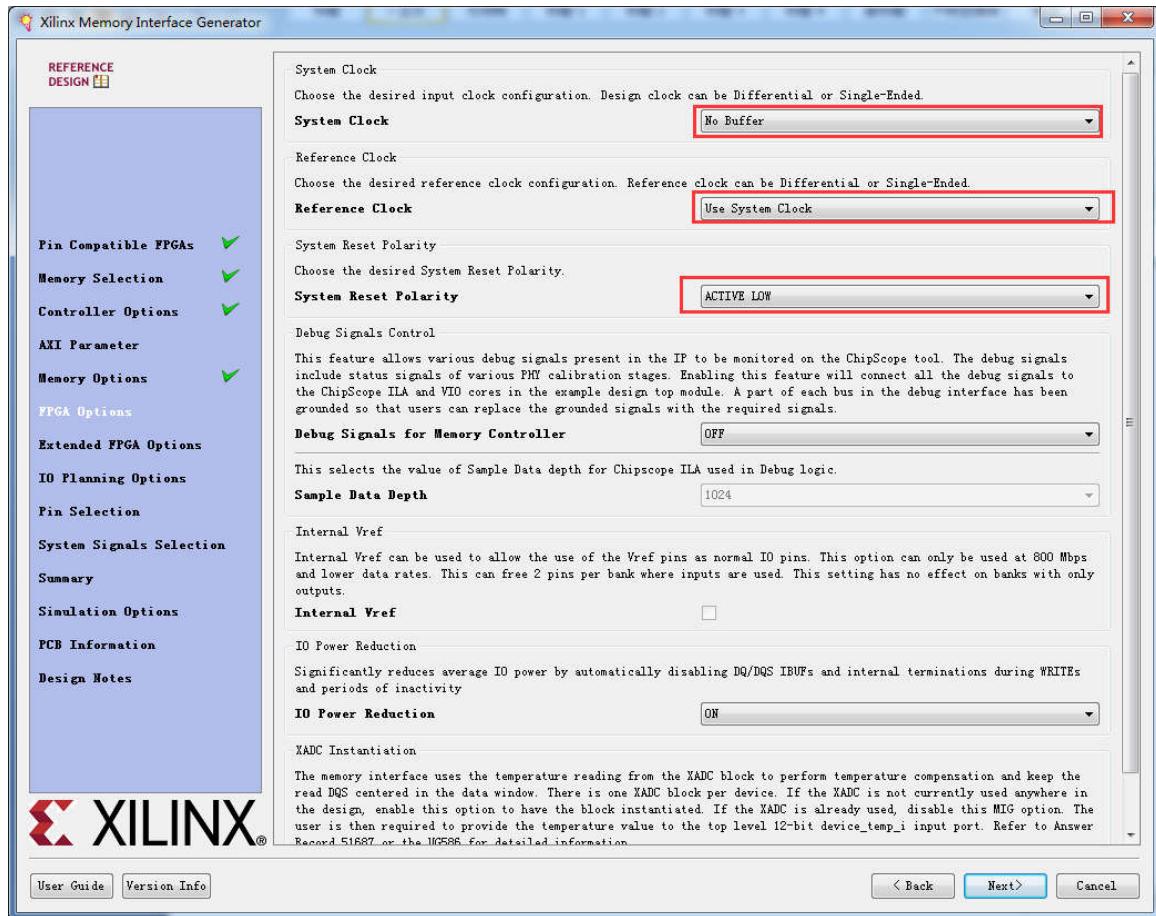
6) “Memory Part” 选择 “MT41J256m16XX-125”，“Data Width” 选择 32



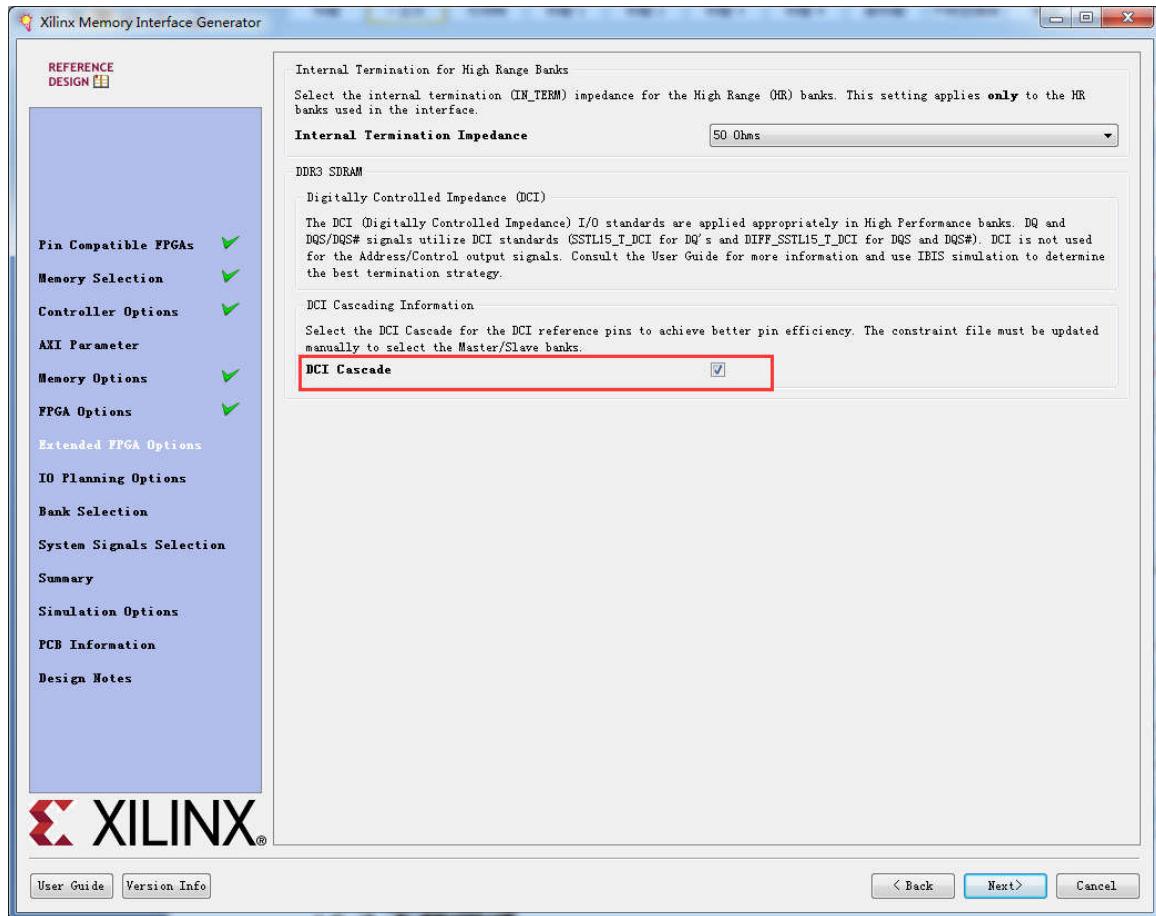
7) “Input Clock Period” 选择 5000ps (200MHz)



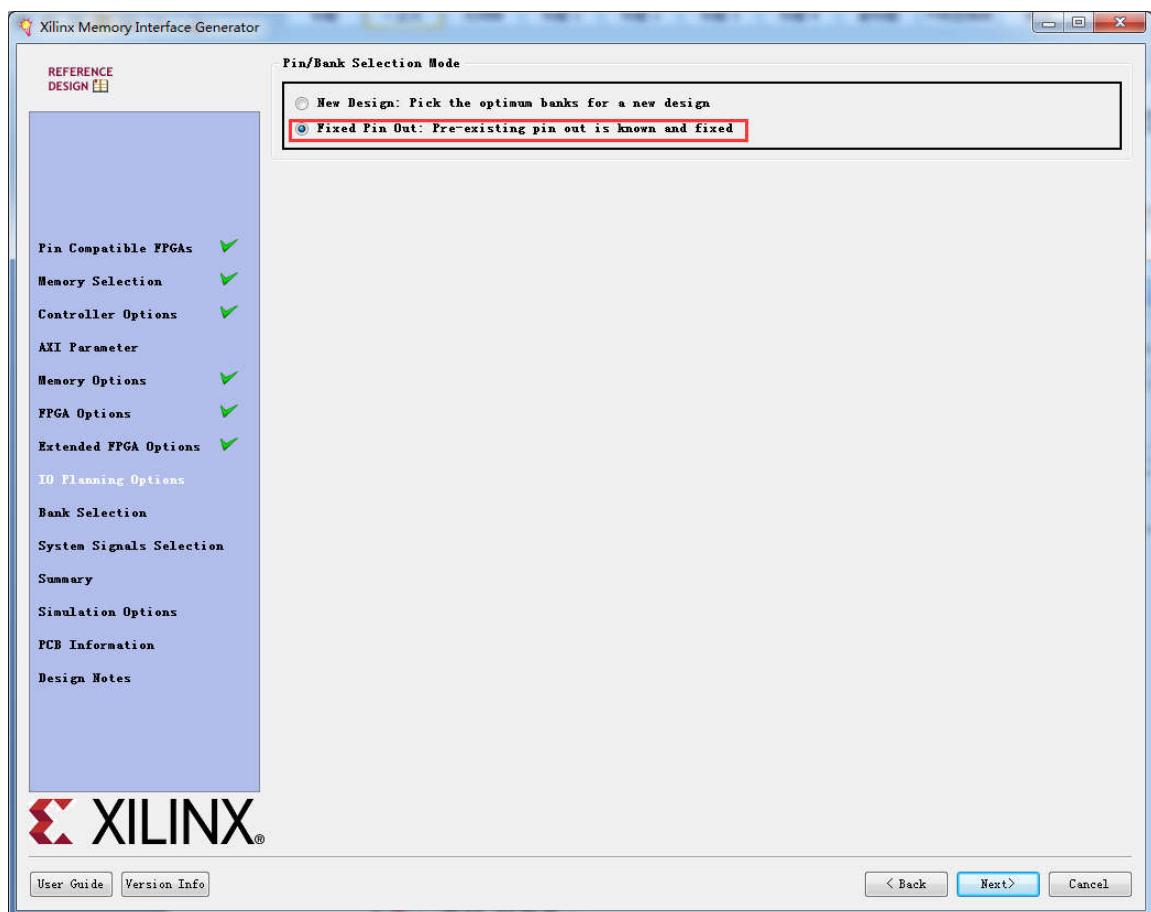
8) “System Clock” 选择 “No Buffer” , “Reference Clock” 选择 “Use System Clock” , “System Reset Polarity” 选择 “ACTIVE LOW” , 点击 “Next”



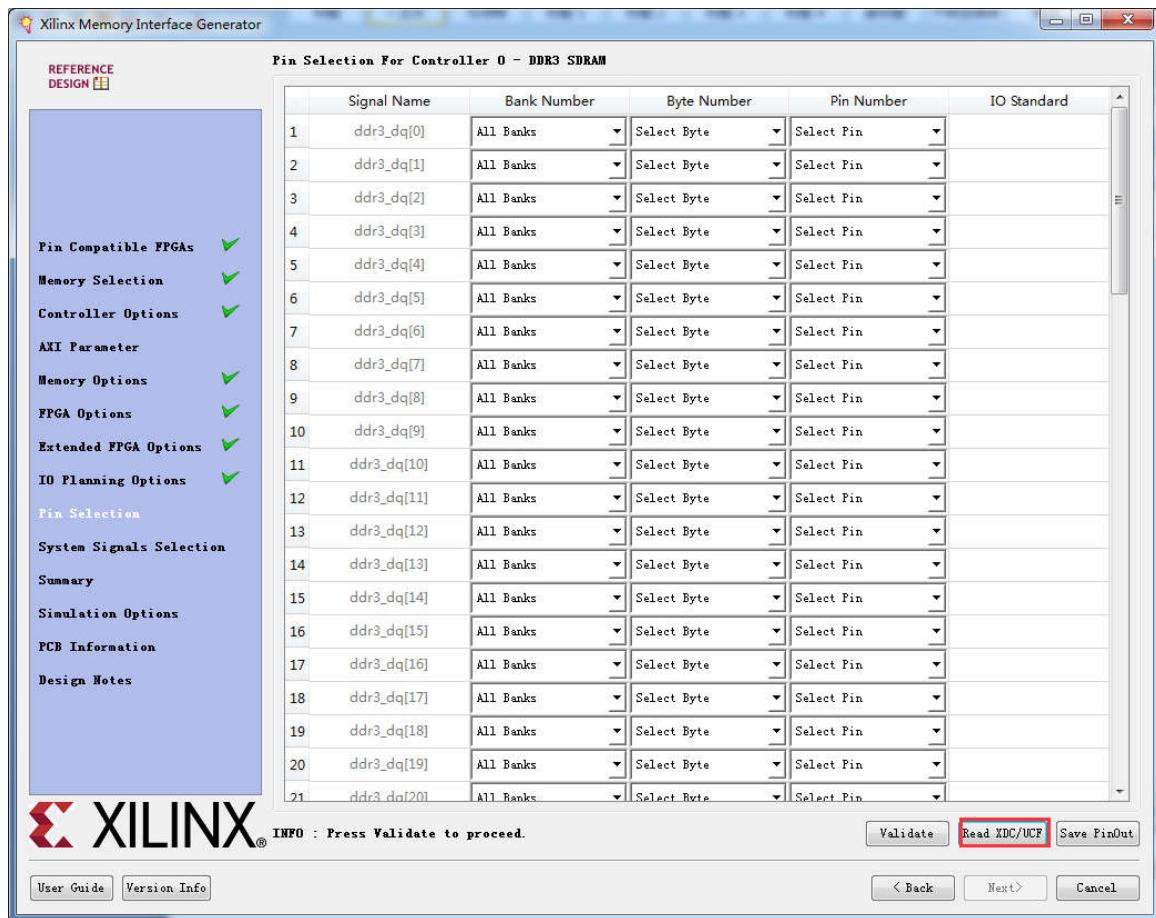
9) 使能 DCI Cascade , 点击 “Next”



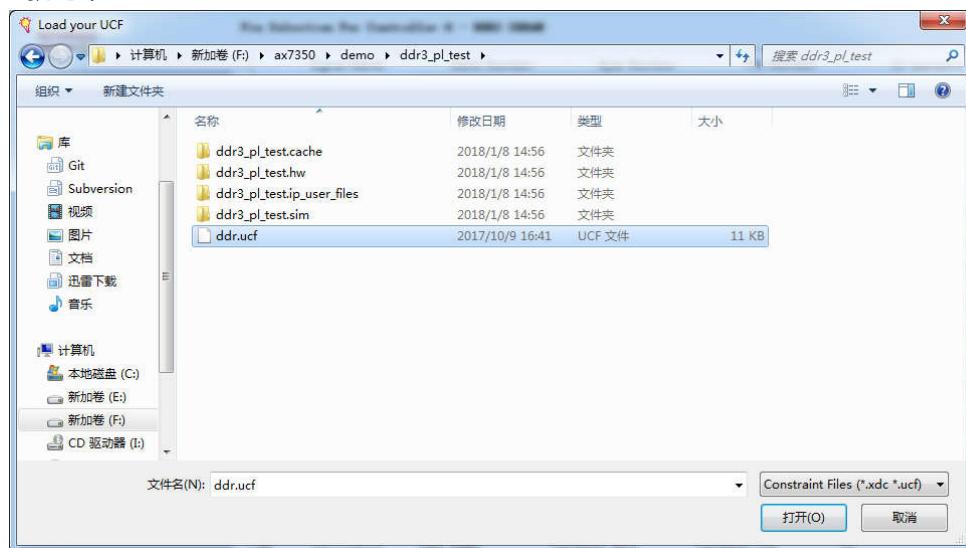
10) 选择 “Fixed Pin Out : Pre-existing pin out is known and fixed”



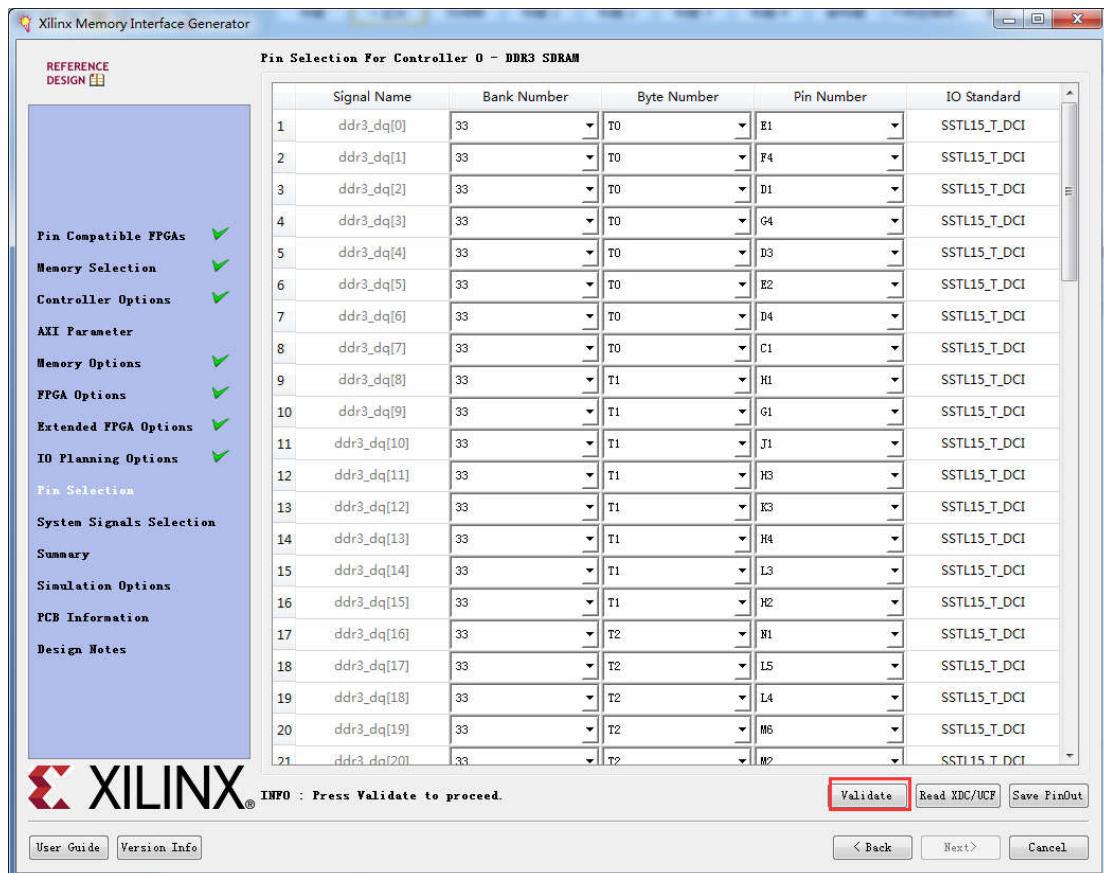
11) 点击 “Read XDC/UCF”



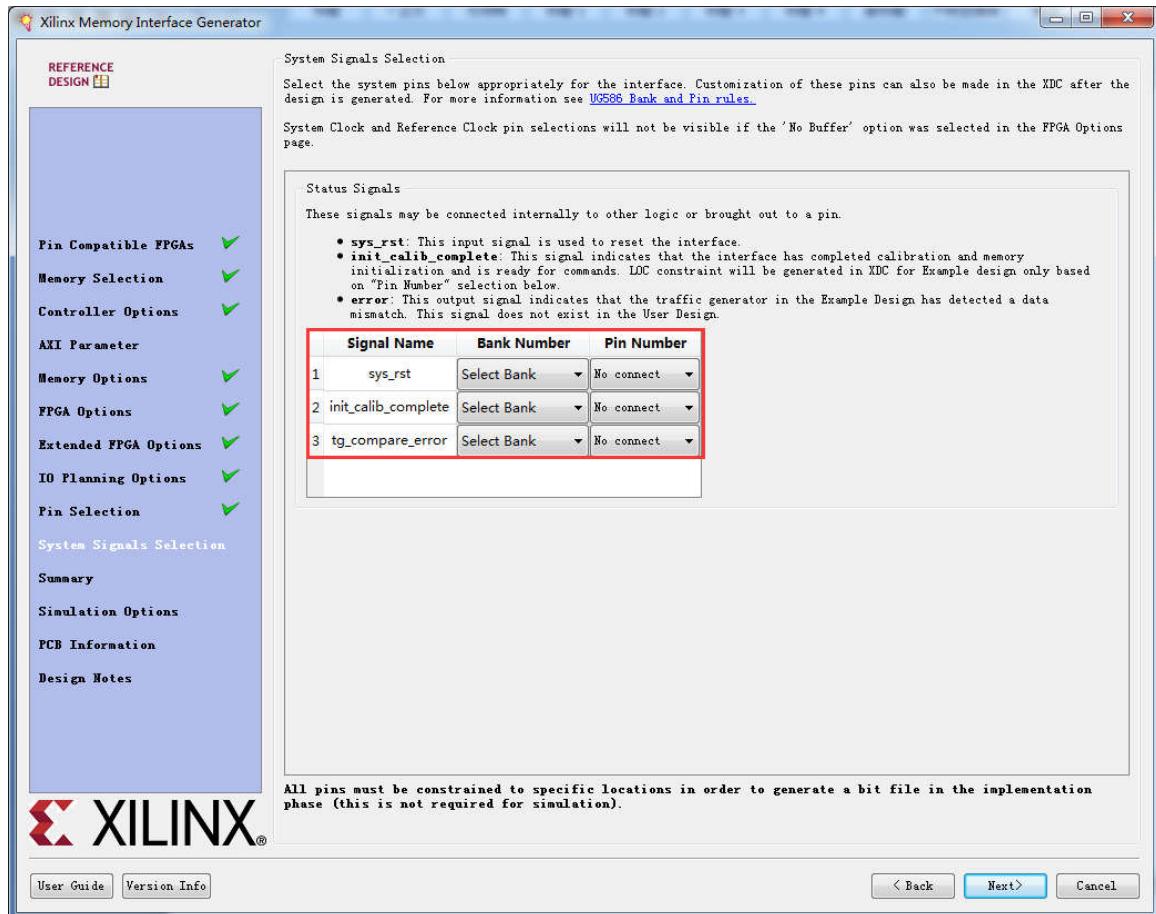
12) 选择 ddr.ucf, 这里可以选择工程里已经存在的 XDC 文件，只要包含 ddr3 的管脚分配信息就可以。



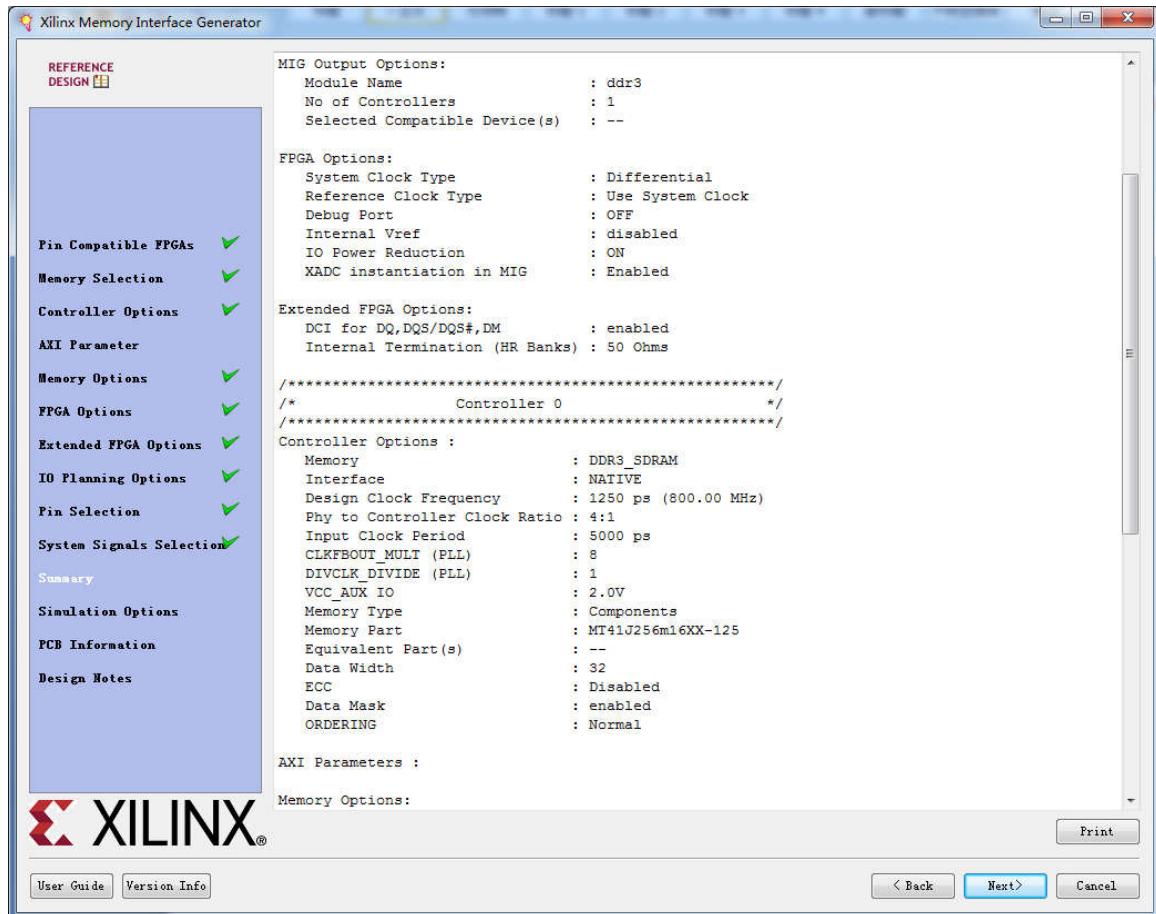
13) 点击 “Validate”



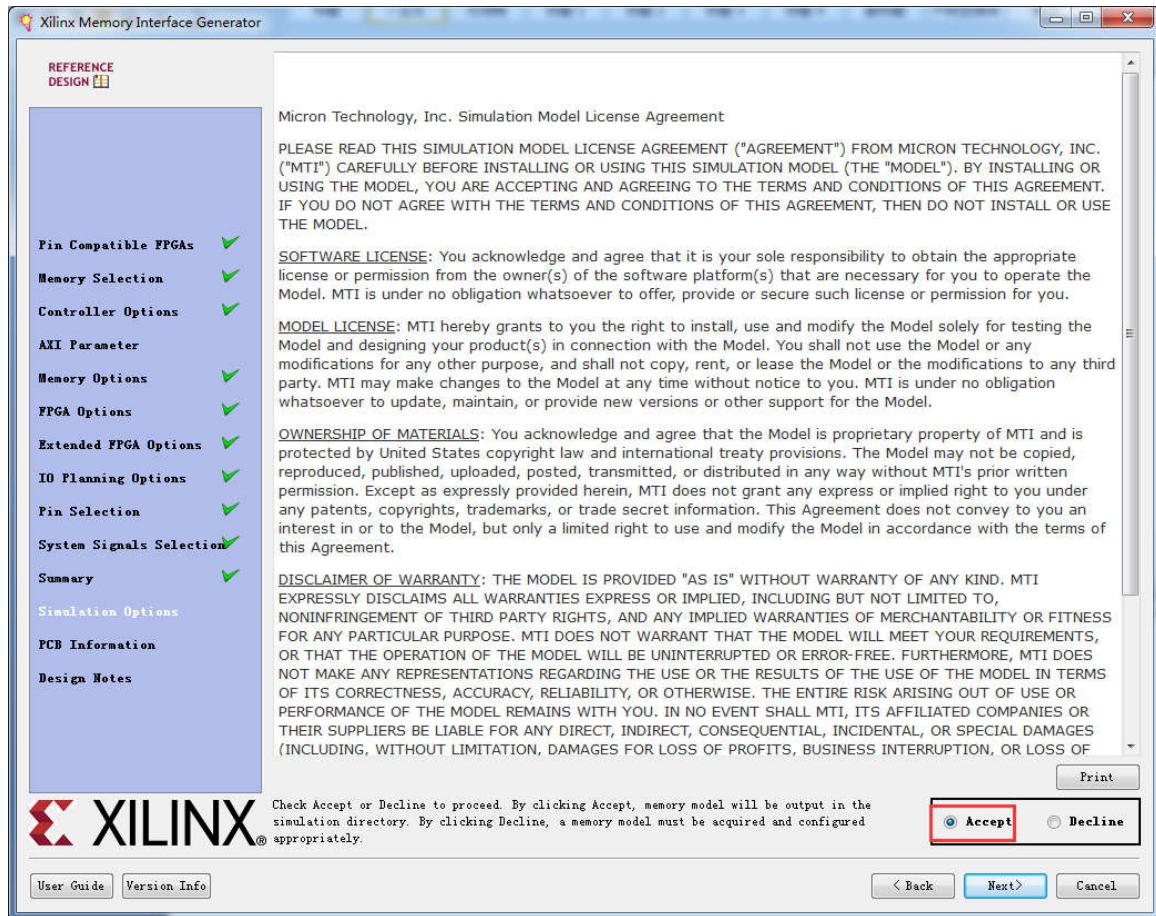
14) 选择测试输出的管脚，这里保持默认，不配置



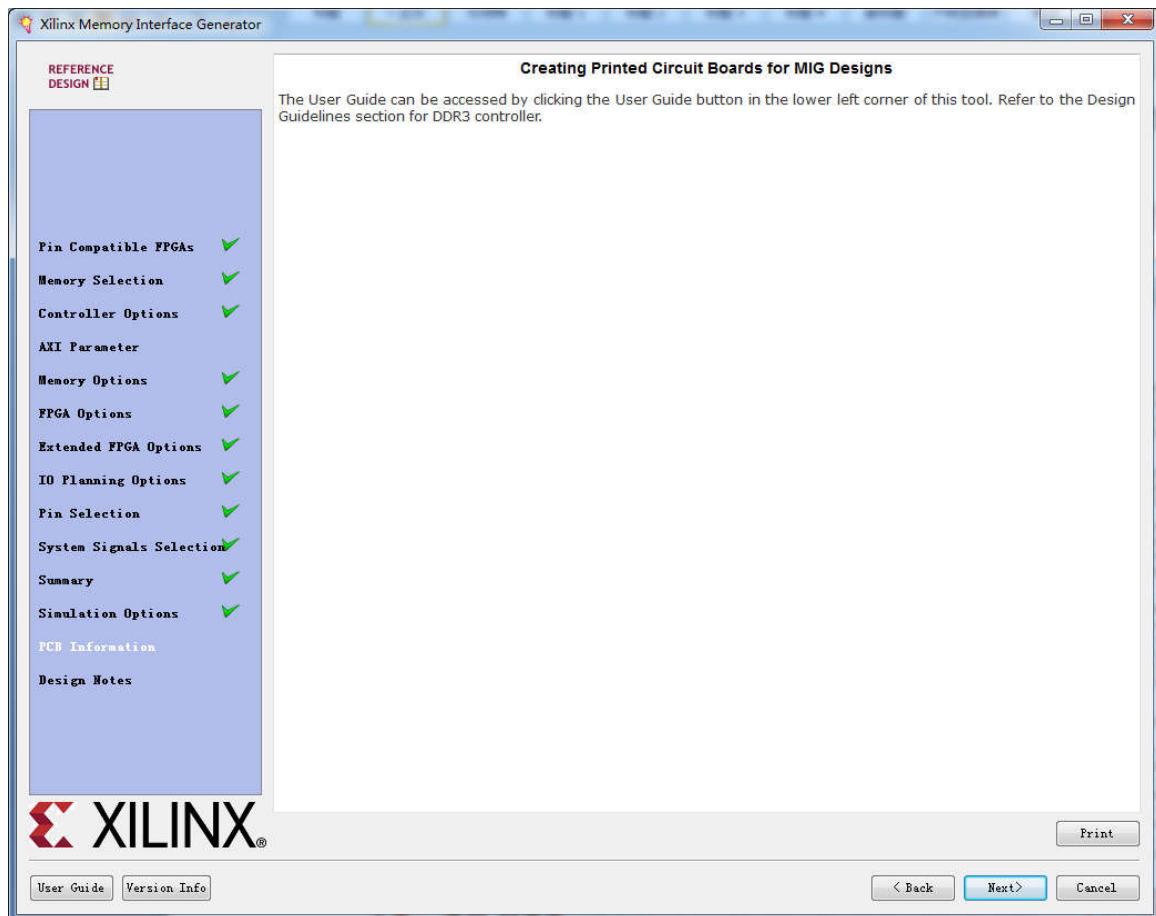
15) 点击 “Next”



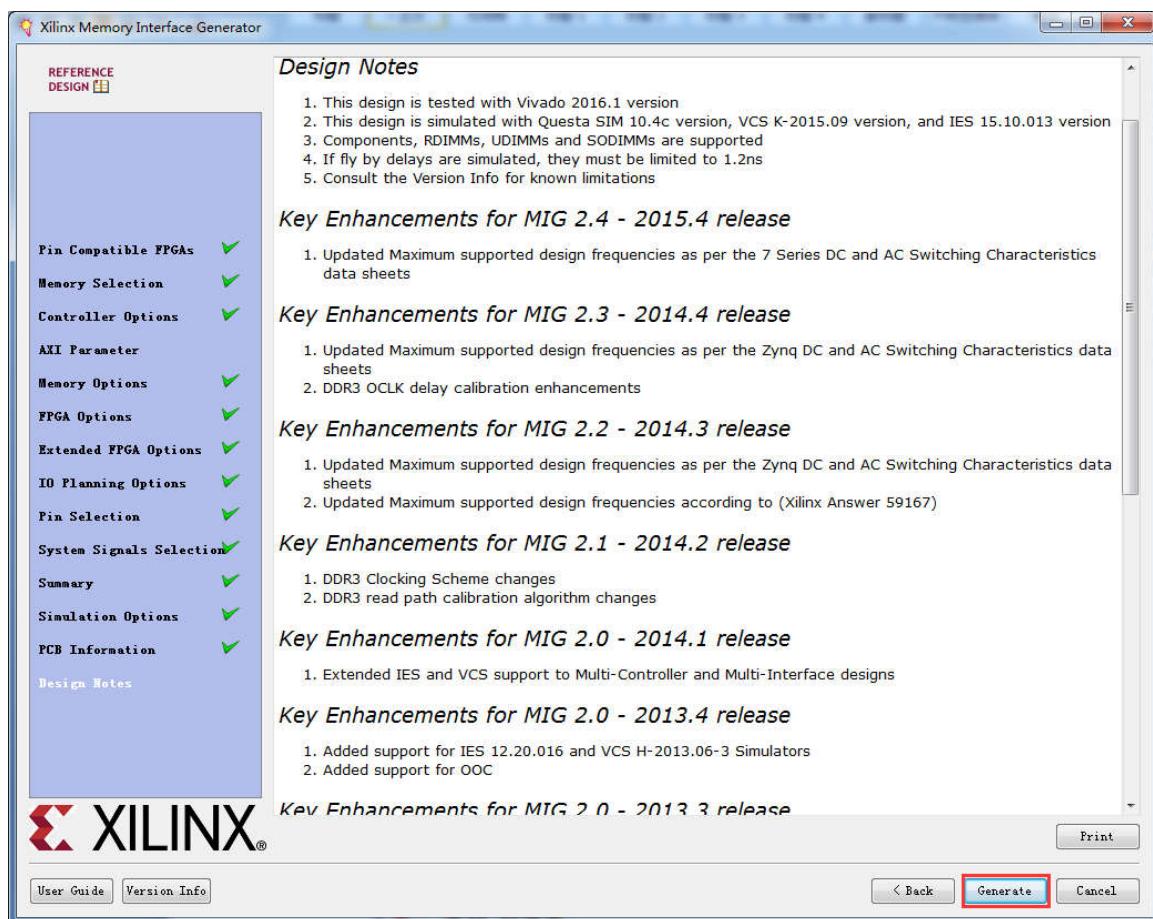
16) 点击 “Access” 接受条款



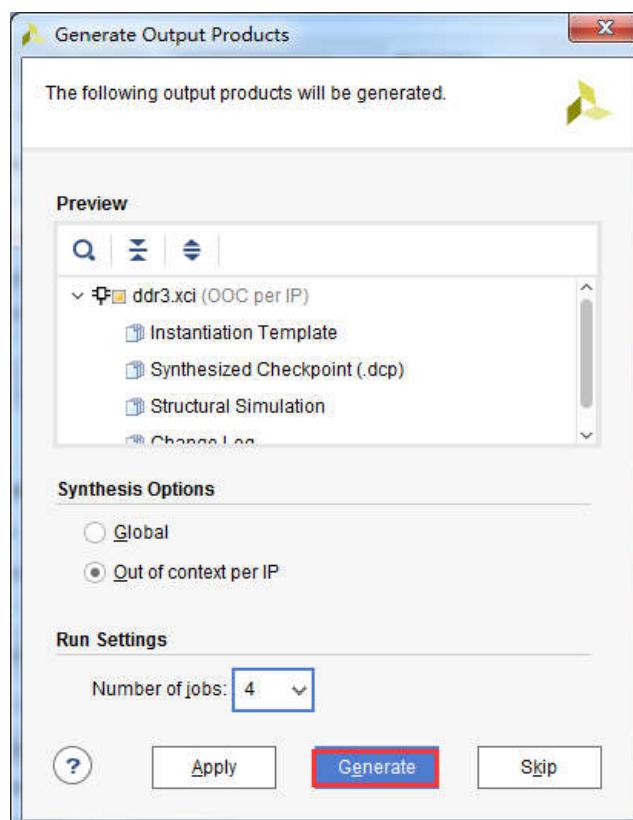
17) 点击 “Next”



18) 点击 “Generate”

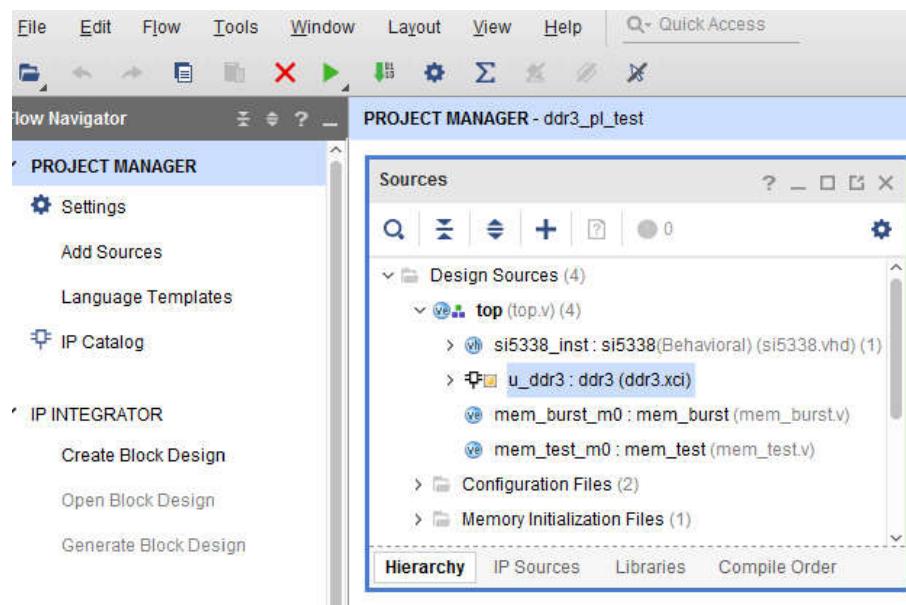


19) 在弹出的“Generate Output Products”中选择“Generate”



7.2.3 添加其他测试代码

其他代码主要功能是配置 si5338，读写 ddr3 并比较数据是否一致，这里不做详细介绍，可参考工程代码。



7.3 下载调试

生成 bit 文件以后，使用 JTAG 下载到开发板，我们可以通过 LED 来观察 ddr3 测试情况，LED1 亮表示 si5338 配置完成，LED2 亮表示 ddr3 读写有错误，LED3 亮表示 ddr3 控制器初始化完成，LED4 闪烁表示 ddr3 测试程序在运行。

7.4 实验总结

本实验通过 PL 端 Verilog 代码直接读写 ddr3，通过 LED 来显示测试结果，我们也可以把 ddr3 配置成 AXI 接口，这样方便和 ARM 系统完成数据交互。

第八章 GTX 收发器误码率测试 IBERT 实验

实验 Vivado 工程为 “ibert_test”。

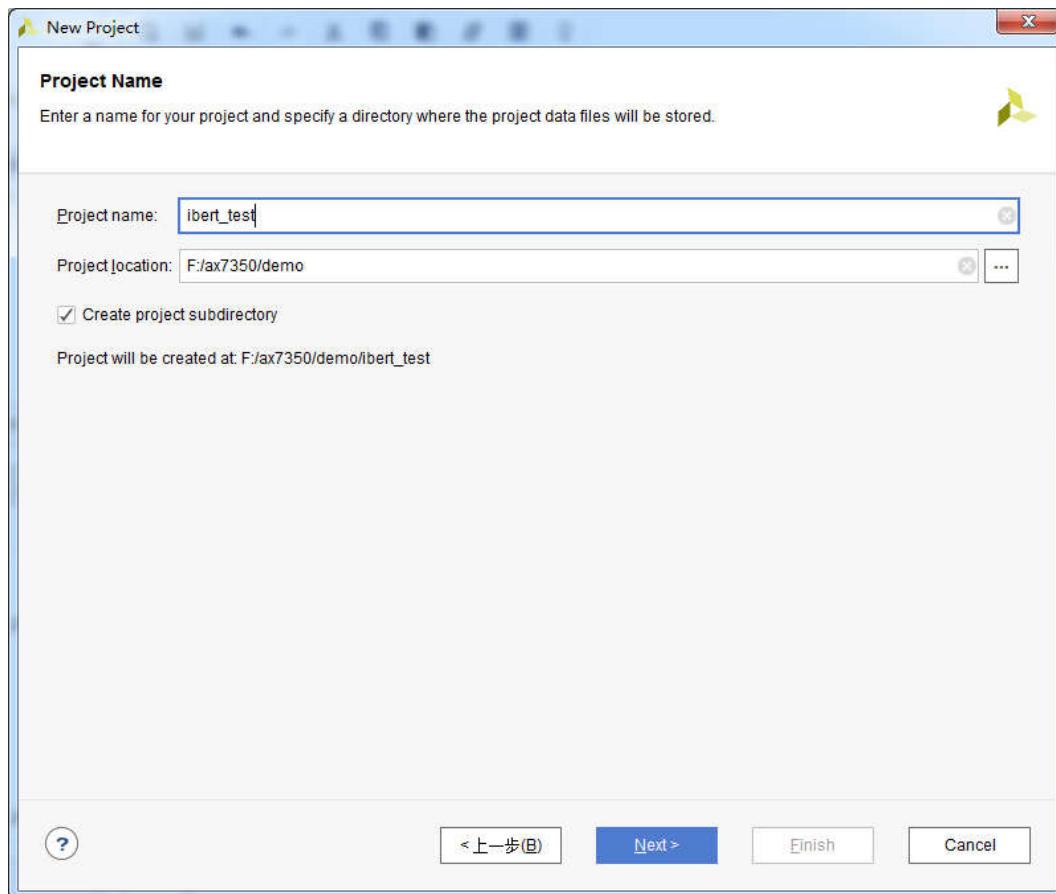
Vivado 软件为我们提供了强大的误码率测试器 IBERT，不但可以测试误码率还能测试眼图，给我们使用高速收发器带来很大的便利，本实验做个抛砖引玉，简单介绍 IBERT 的使用。

8.1 硬件介绍

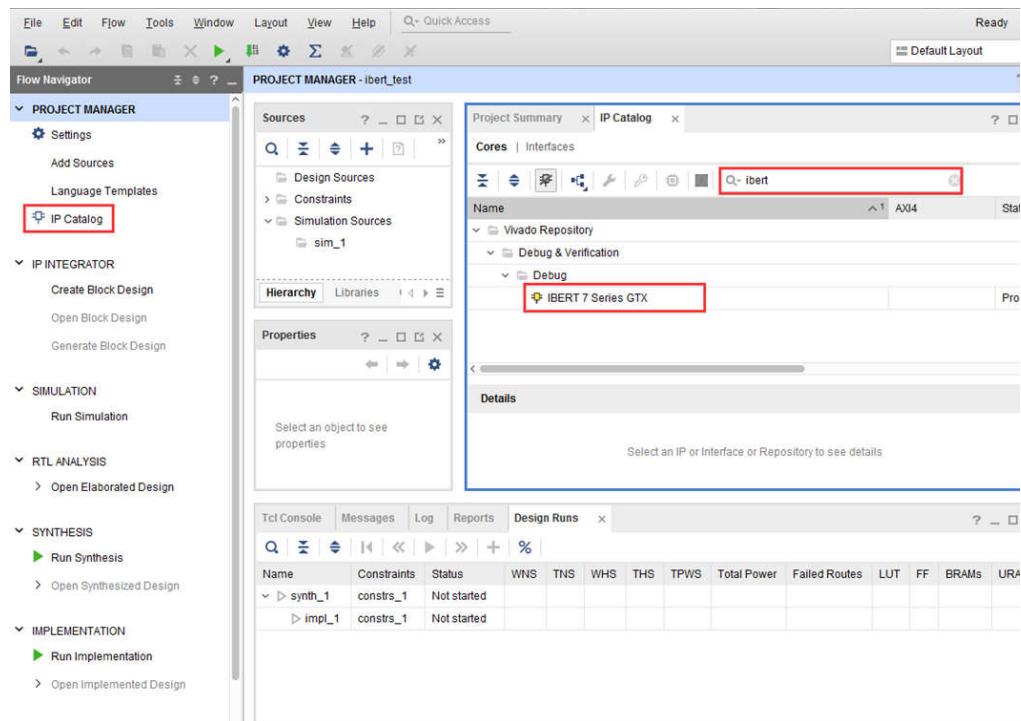
使用 IBERT 测试误码率和眼图必须有个收发环通的硬件，开发板上有 2 个 SFP 光纤接口，本实验把 2 个光接口收发连接，形成 2 个收发环通链路。

8.2 Vivado 工程建立

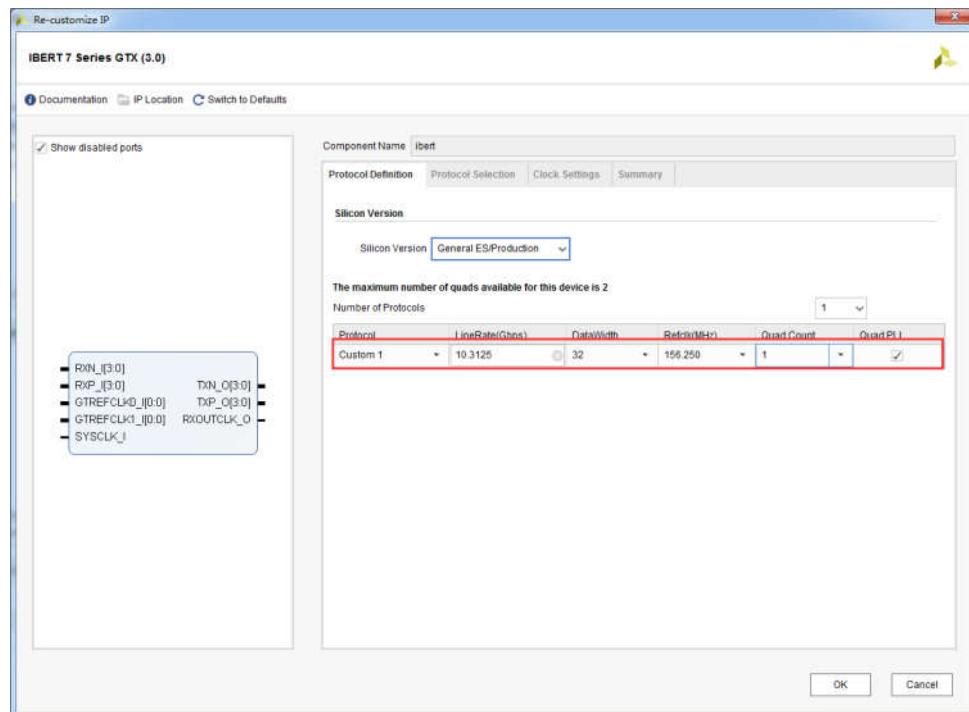
- 1) 新建一个工程名为 “ibert_test”



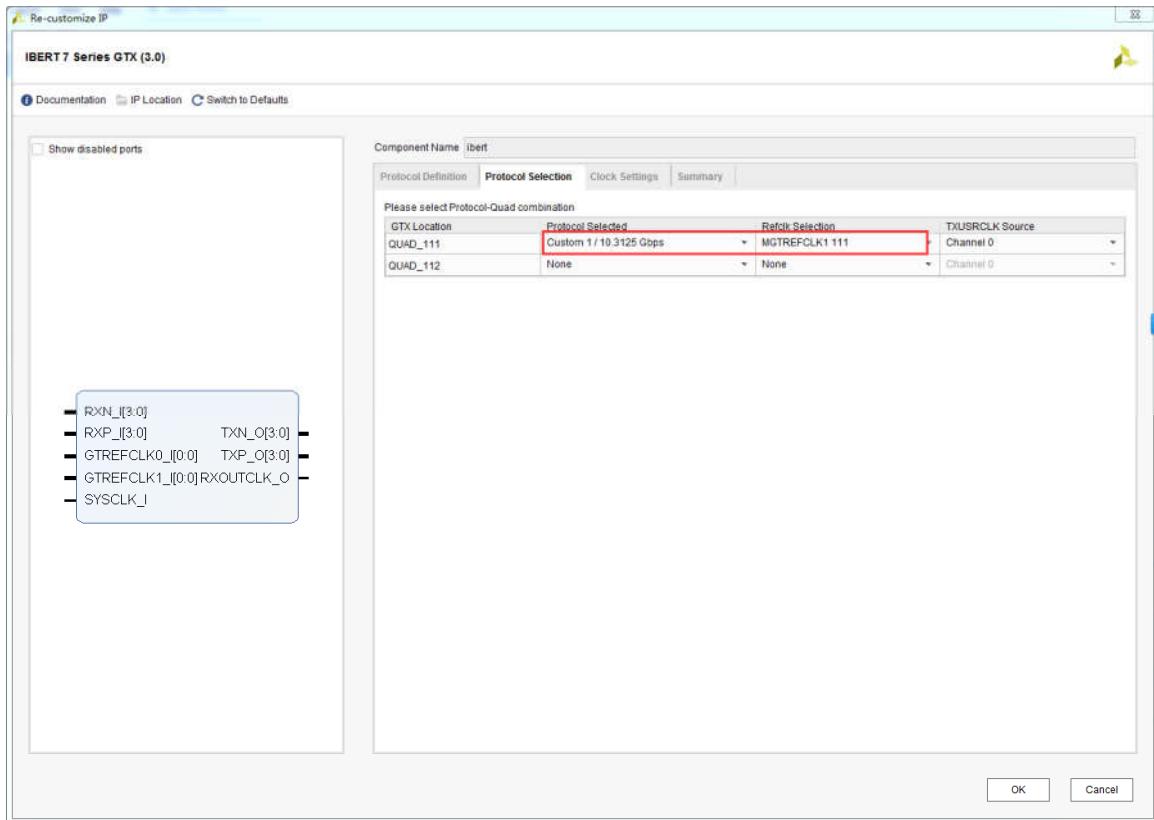
- 2) 在 “IP Catalog” 中搜索 “ibert” 快速找到 “IBERT 7 Series GTX”，双击



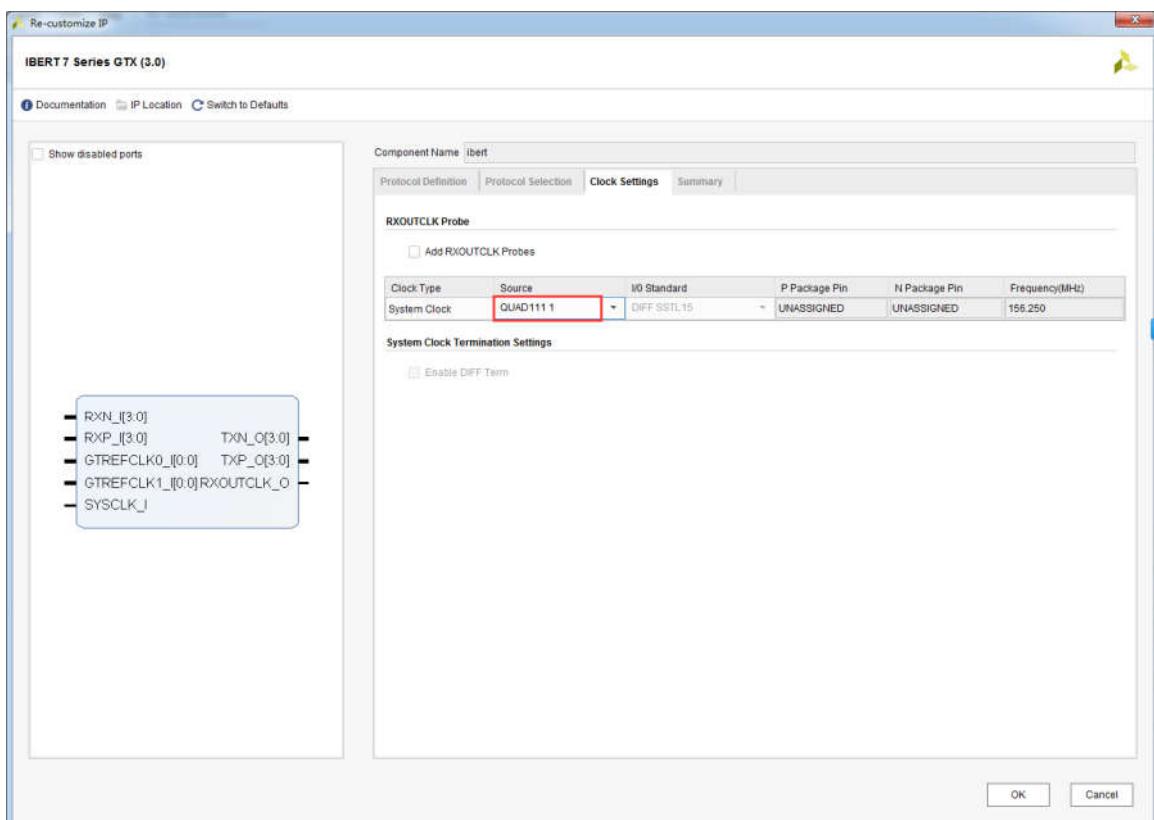
- 3) “Component Name” 改为 “ibert” , LineRate 填写 10.3125G , 这是芯片支持的最大速率 , Refclk (MHZ) 选择 156.250



- 4) 在 “Protocol Selected” 中选择 “Custom 1/10.3125Gbps” , “Refclk Selection” 选择 “MGTRREFCLK1 111”

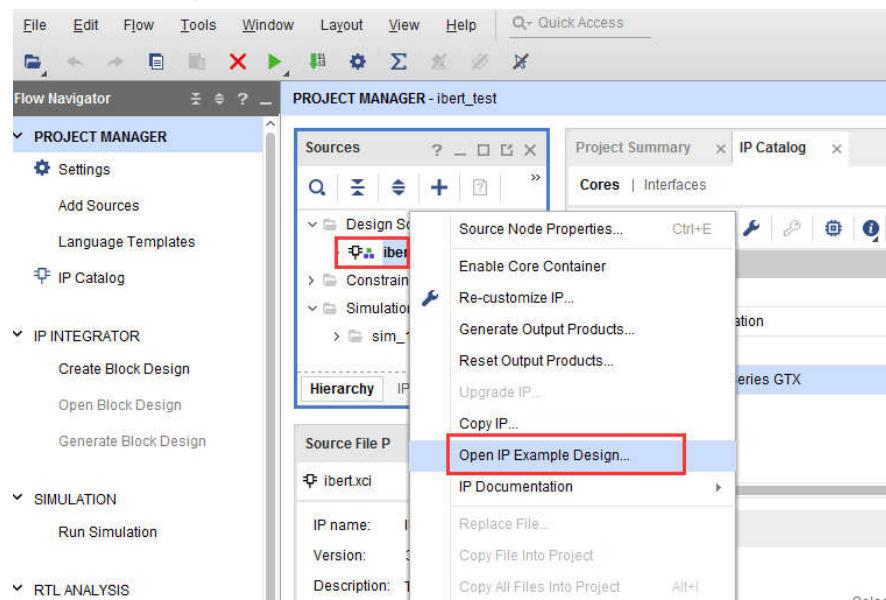


- 5) 在“Clock Settings”页选择“Source”选择“QUAD111 1”

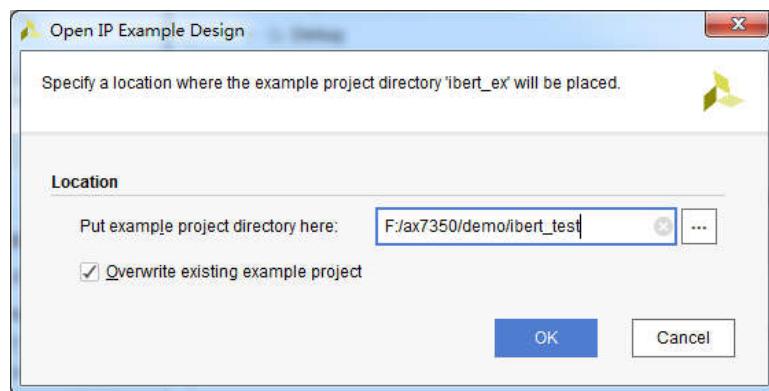


- 6) 生成 IP core 后等待一段时间后选择 IP , 右键 “Open IP Example Design...” ,为 IP 生成一个测

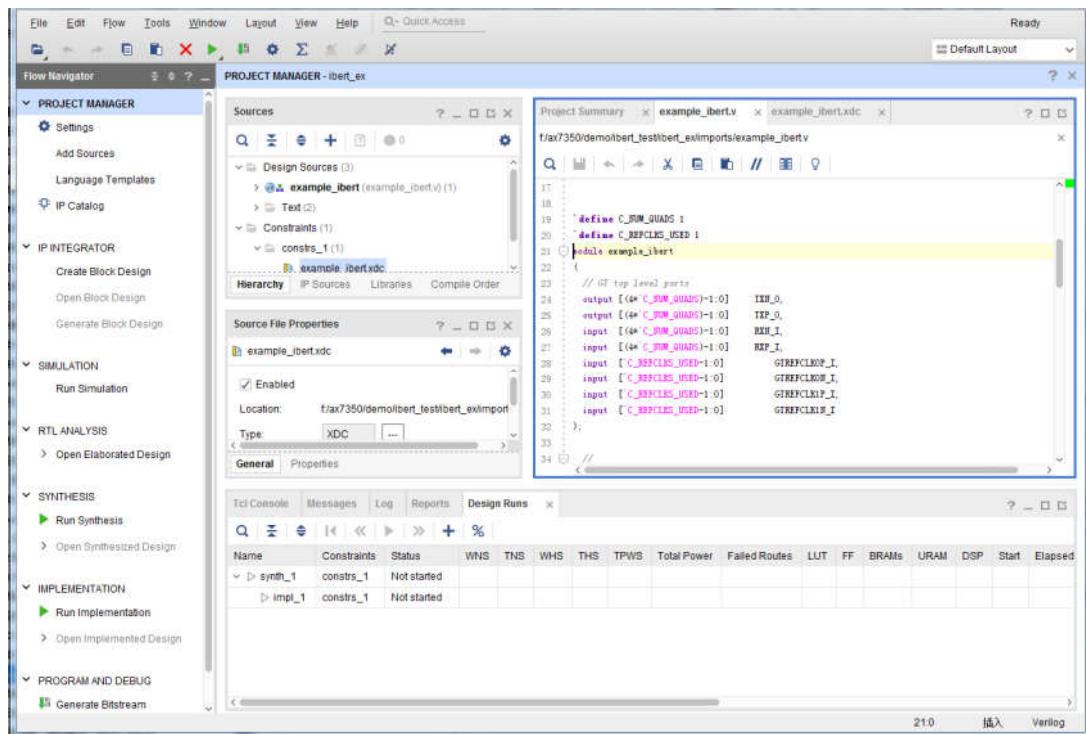
试例程，很多 IP 都带有测试例程，包括前面讲解的 ddr3 控制器。



7) 选择测试例程的路径，点击 “OK”

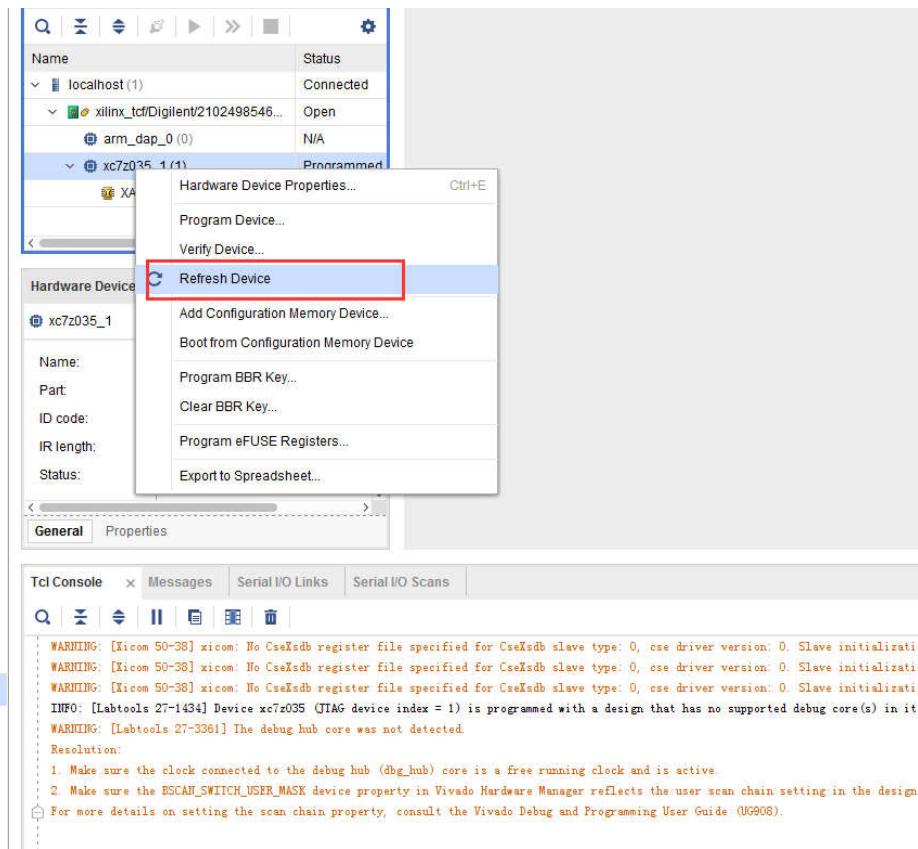


8) Vivado 会自动生成一个工程并打开，这个例程可能需要我们修改才能正常使用，本实验中添加了 SFP 光发送使能，si5338 配置程序，具体修改参考提供的工程。

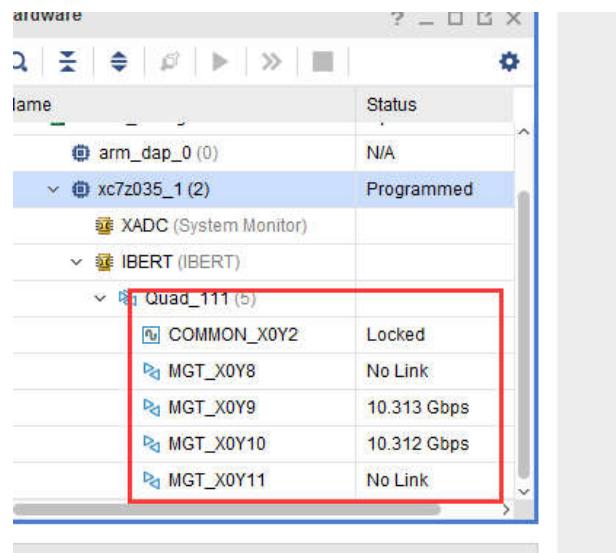


8.3 下载调试

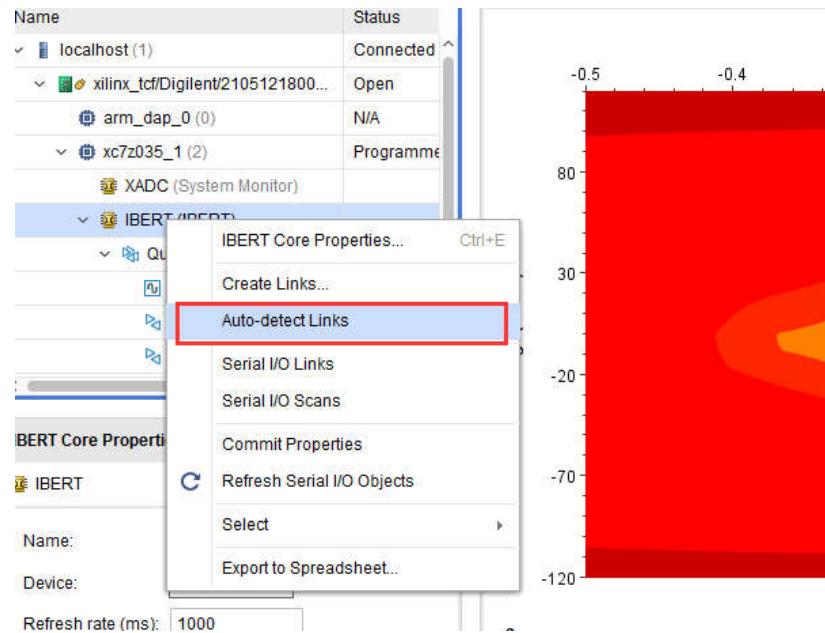
- 1) 先使用插入光模块，然后使用光纤将 2 个光口对接，给开发板上电
- 2) 将修改后的例程工程编译生成 bit 文件，使用 JTAG 下载到开发板，然后刷新设备



3) 刷新完成以后的情况，可以看到有 2 个链路锁定，速度 10.313Gbps。



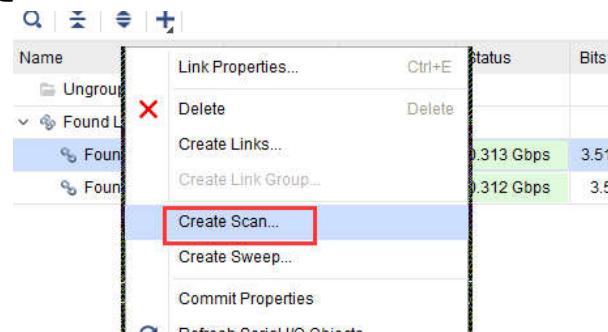
4) 如果没有连接信息，可以右键点击 “Auto-detect Links”



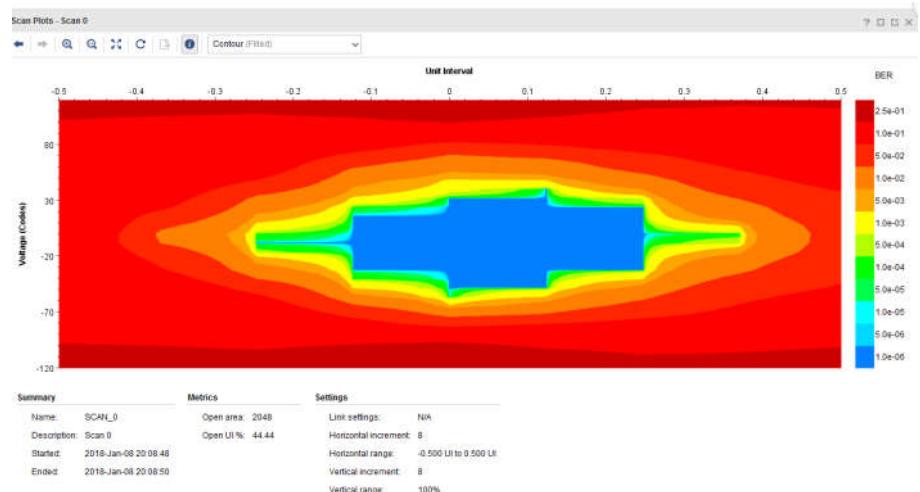
- 5) 点击“Serial I/O Links”，可以看到 Errors 都是 0，如果不是 0，可以点击“Reset”，重新开始测试。

Serial I/O Links											
Name	TX	RX	Status	Bits	Errors	BER	BERT Re...	TX Pattern	RX Pattern	TX Pre-Cursor	TX Post-Cursor
Ungrouped Links (0)											
Found Links (2)											
Found 0	MGT_X0Y10/TX	MGT_X0Y9/RX	10.312 Gbps	3.983E10	0E0	2.51E...	Reset	PRBS 7-bit	PRBS 7-bit	1.67 dB (00111)	0.68 dB (00011)
Found 1	MGT_X0Y9/TX	MGT_X0Y10/RX	10.313 Gbps	3.984E10	0E0	2.51E-11	Reset	PRBS 7-bit	PRBS 7-bit	1.67 dB (00111)	0.68 dB (00011)

- 6) 选择一个链路，右键“Create Scan...”



- 7) 默认配置出来的眼图，注意：使用不同的软件版本，测量眼图可能会有差异。



8.4 实验总结

IBERT 提供了强大的调试功能，本实验仅仅展示了部分功能，更详细的使用参考 xilinx 文档 pg132。

第九章 体验 ARM ,裸机输出“Hello World”

实验 Vivado 工程为 “ps_hello”。

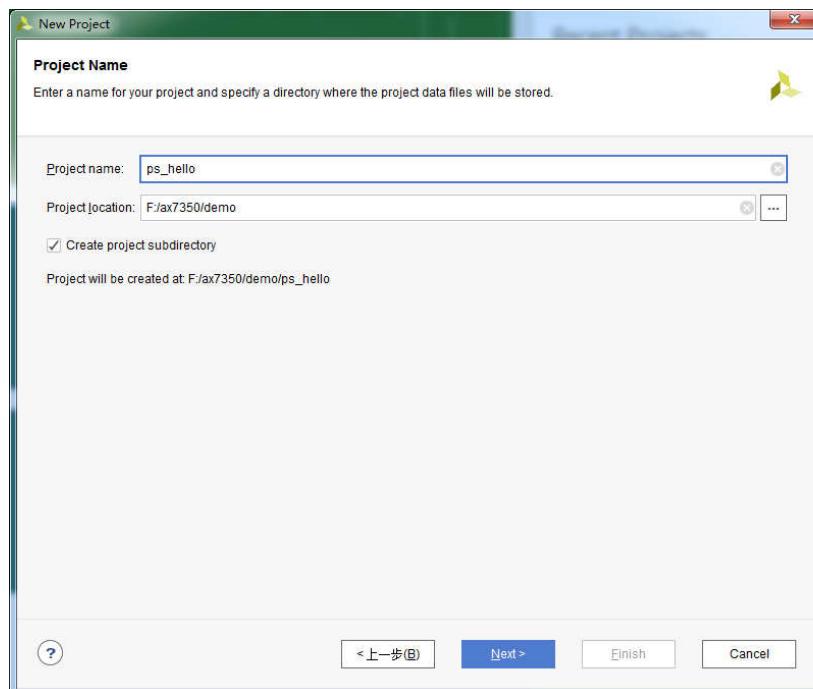
前面的实验都是在 PL 端进行的，可以看到和普通 FPGA 开发流程没有任何区别，ZYNQ 的主要优势就是 FPGA 和 ARM 的合理结合，这对开发人员提出了更高的要求。从本章开始，我们开始使用 ARM，也就是我们说的 PS，本章我们使用一个简单的串口打印来体验一下 Vivado SDK 和 PS 端的特性。

9.1 硬件介绍

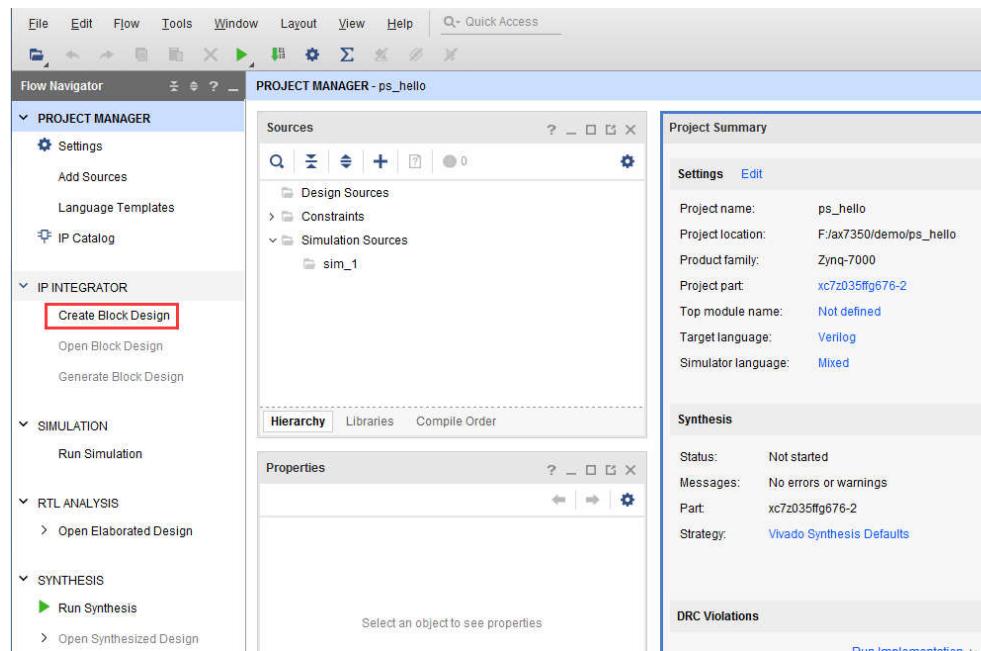
我们从原理图中可以看到 ZYNQ 芯片分为 PL 和 PS，PS 端的 IO 分配相对是固定的，不能任意分配，而且不需要在 Vivado 软件里分配管脚，虽然本实验仅仅使用了 PS，但是还要建立一个 Vivado 工程，用来配置 PS 管脚。

9.2 Vivado 工程建立

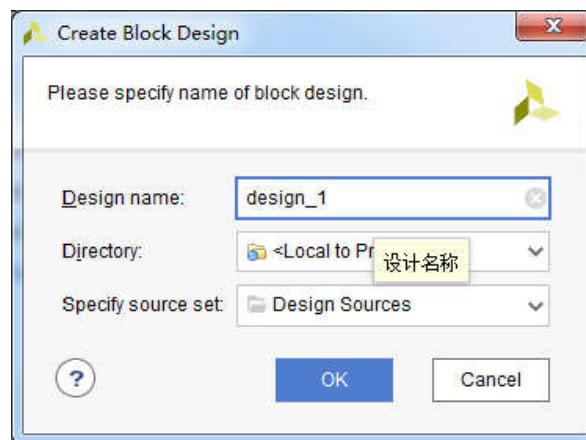
- 1) 创建一个名为 “ps_hello” 的工程



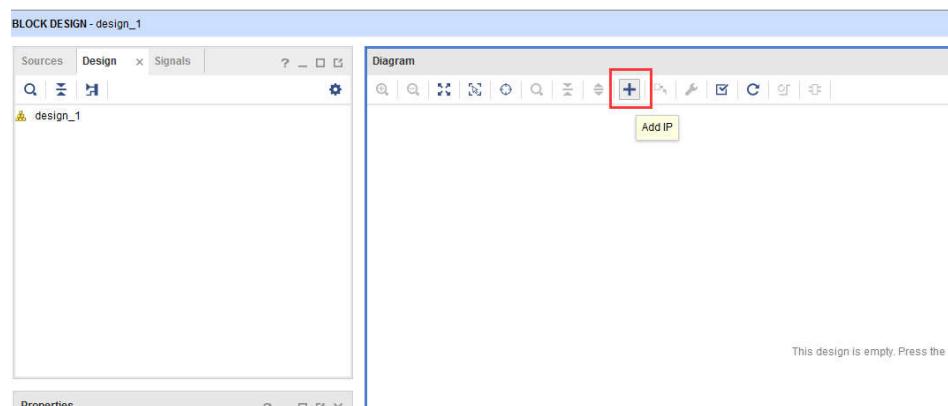
- 2) 点击 “Create Block Design”，创建一个 Block 设计



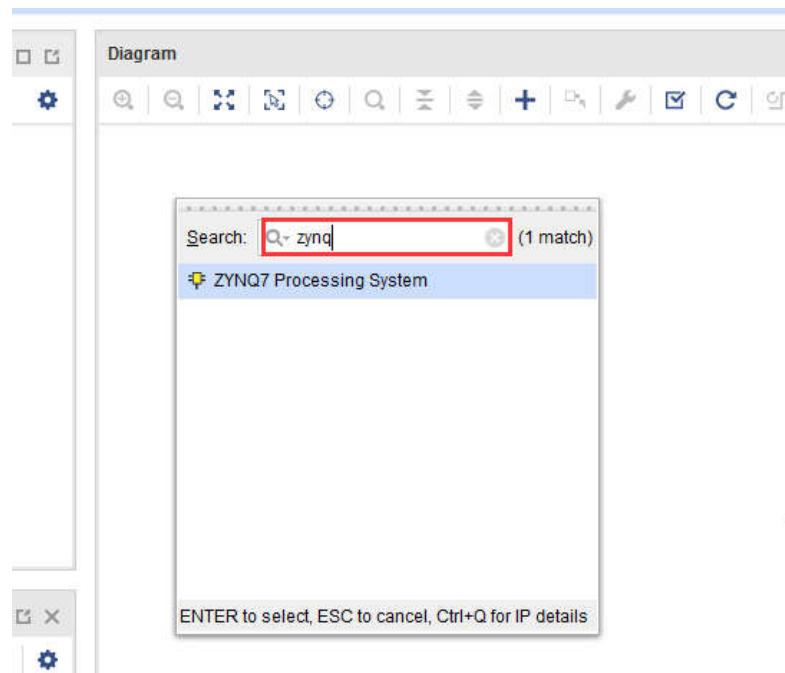
- 3) “Design name” 这里不做修改，保持默认 “design_1”，这里可以根据需要修改，不过名字要尽量简短，否则在 Windows 下编译会有问题。



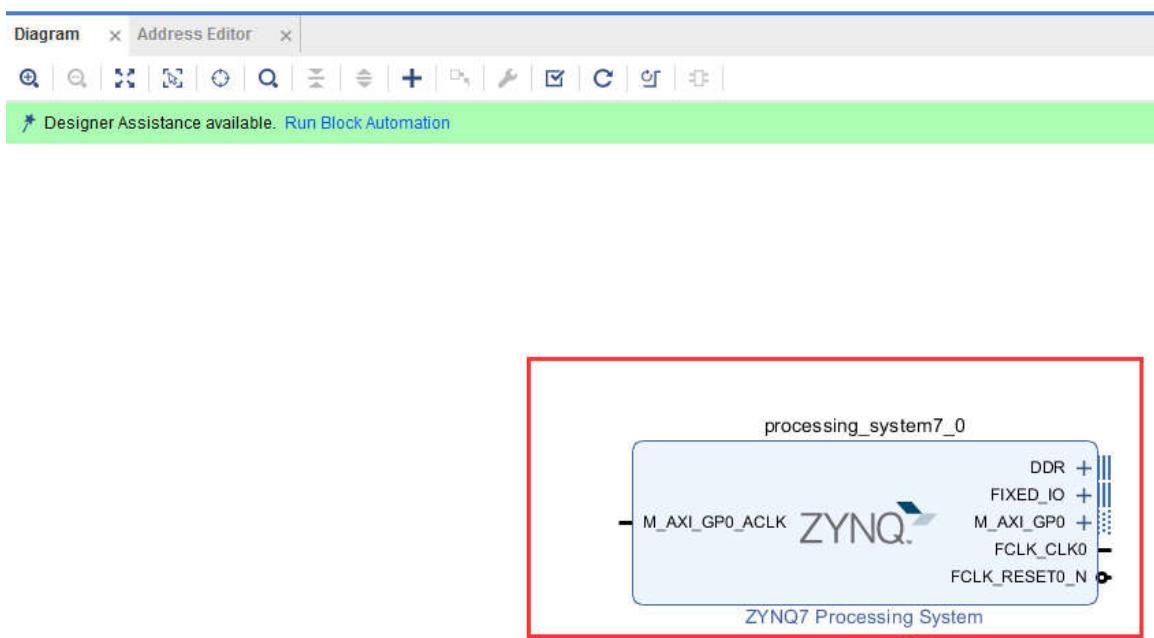
- 4) 点击 “Add IP” 快捷图标



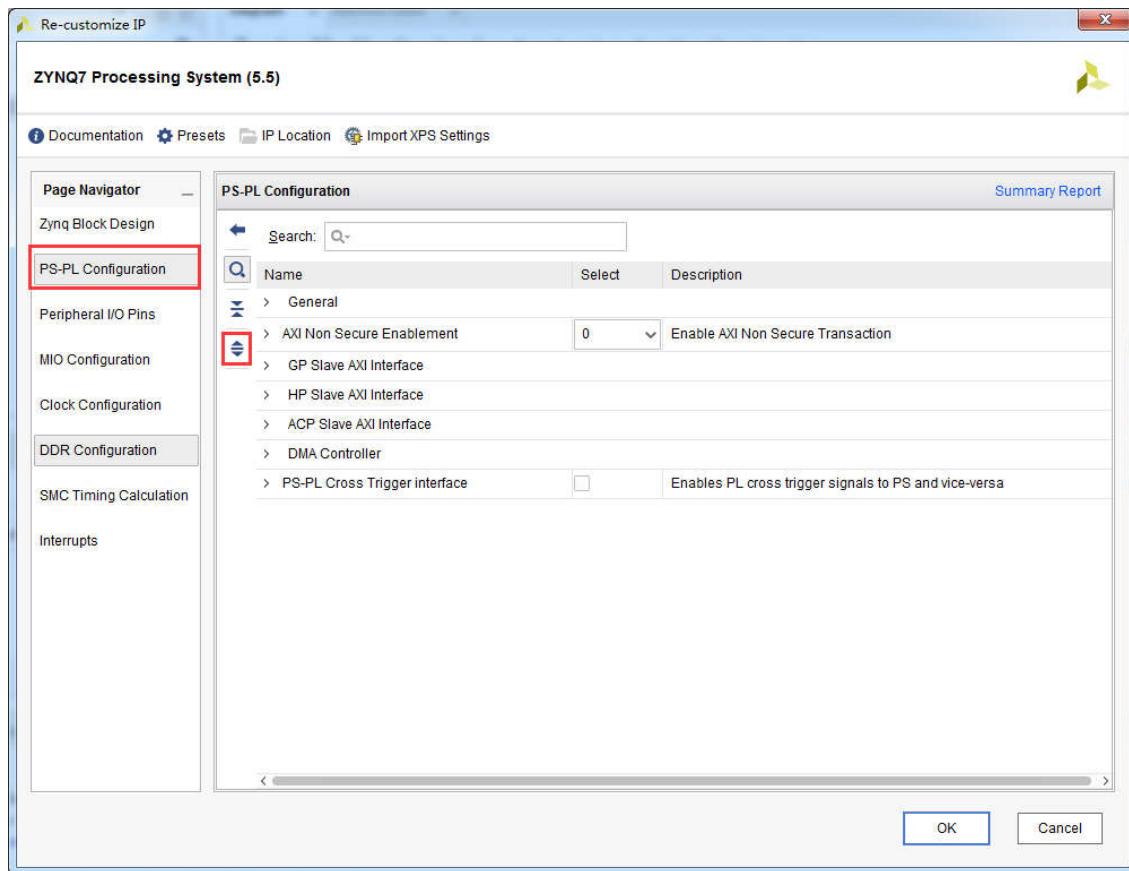
- 5) 搜索 “zynq” , 在搜索结果列表中双击 “ZYNQ7 Processing System”



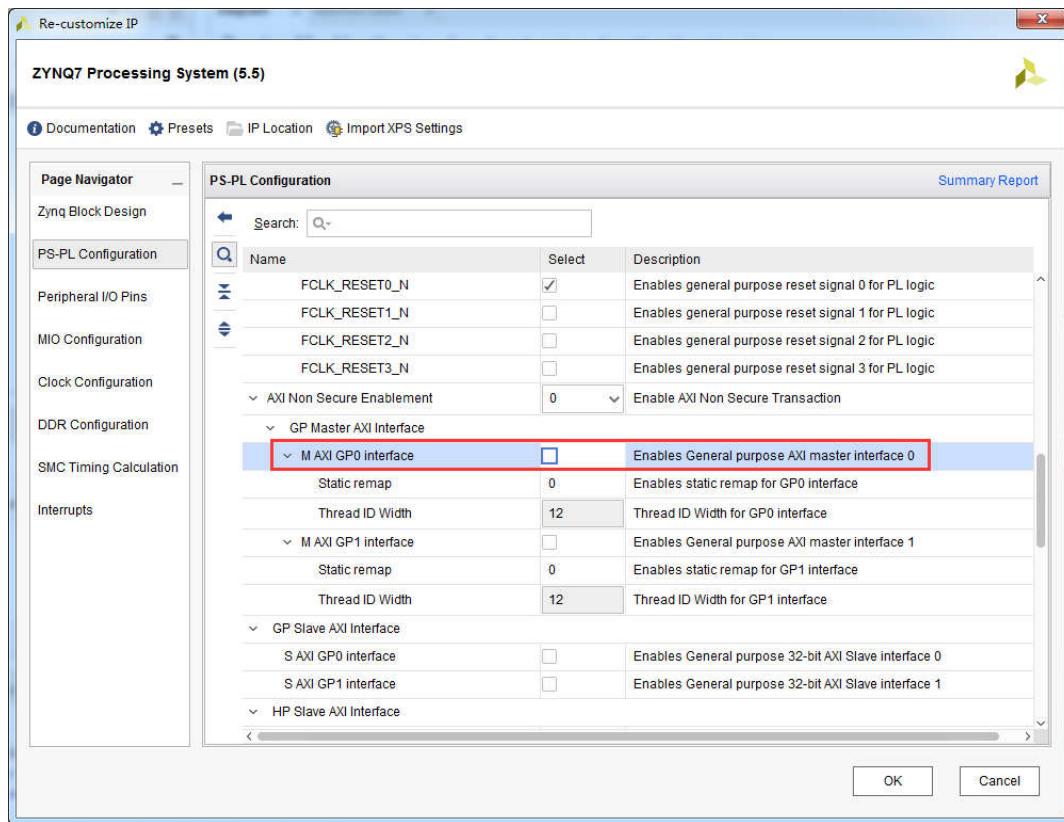
- 6) 双击 Block 图中的 “processing_system7_0”，配置相关参数



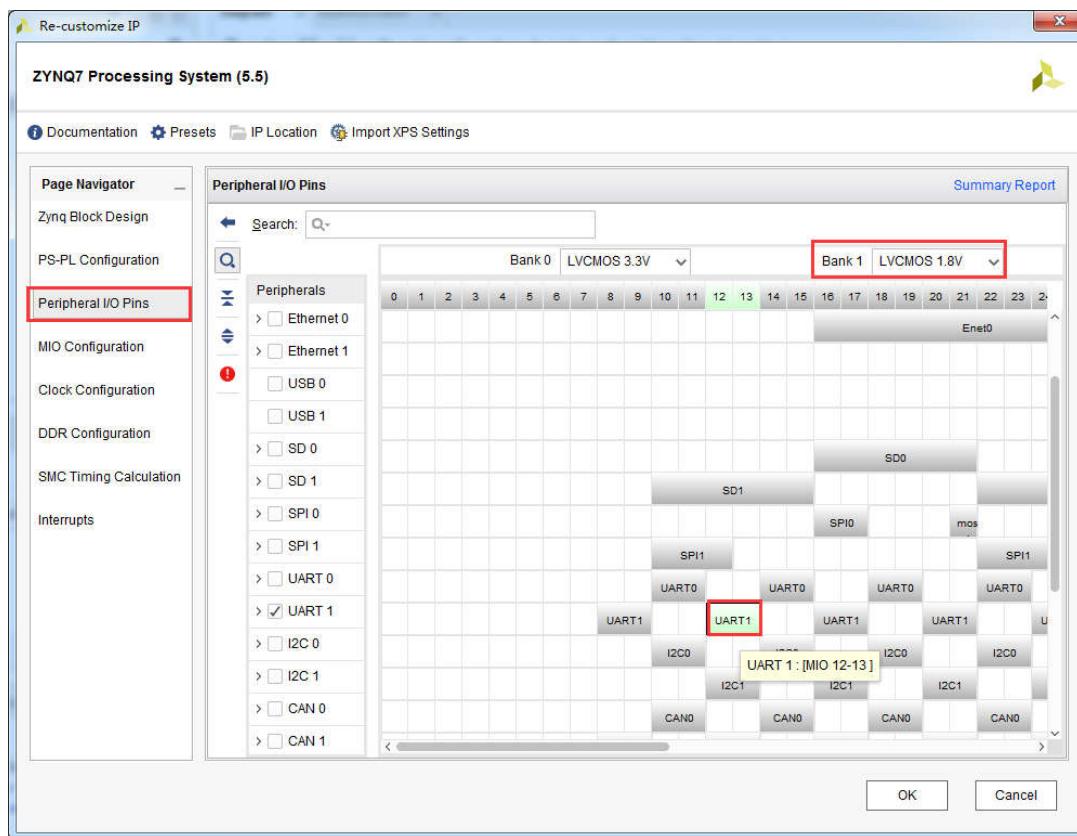
- 7) 在 “PS-PL Configuration” 选项中展开所以项目



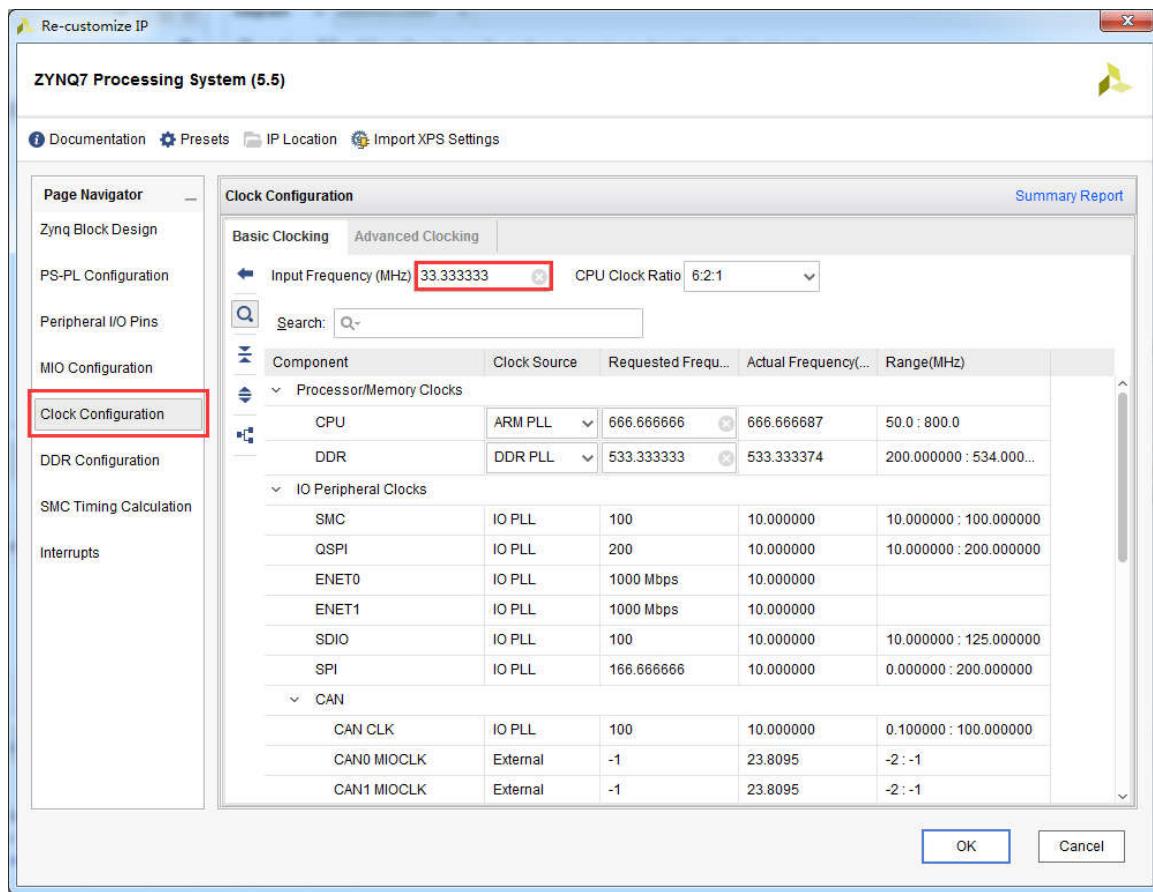
- 8) 取消“M AXI GPO interface”接口，这个接口可以扩展PL端的AXI接口外设，所以PL如果要和PS进行数据交互，都要按照AXI总线协议进行，xilinx为我们提供了大量的AXI接口的IP核。



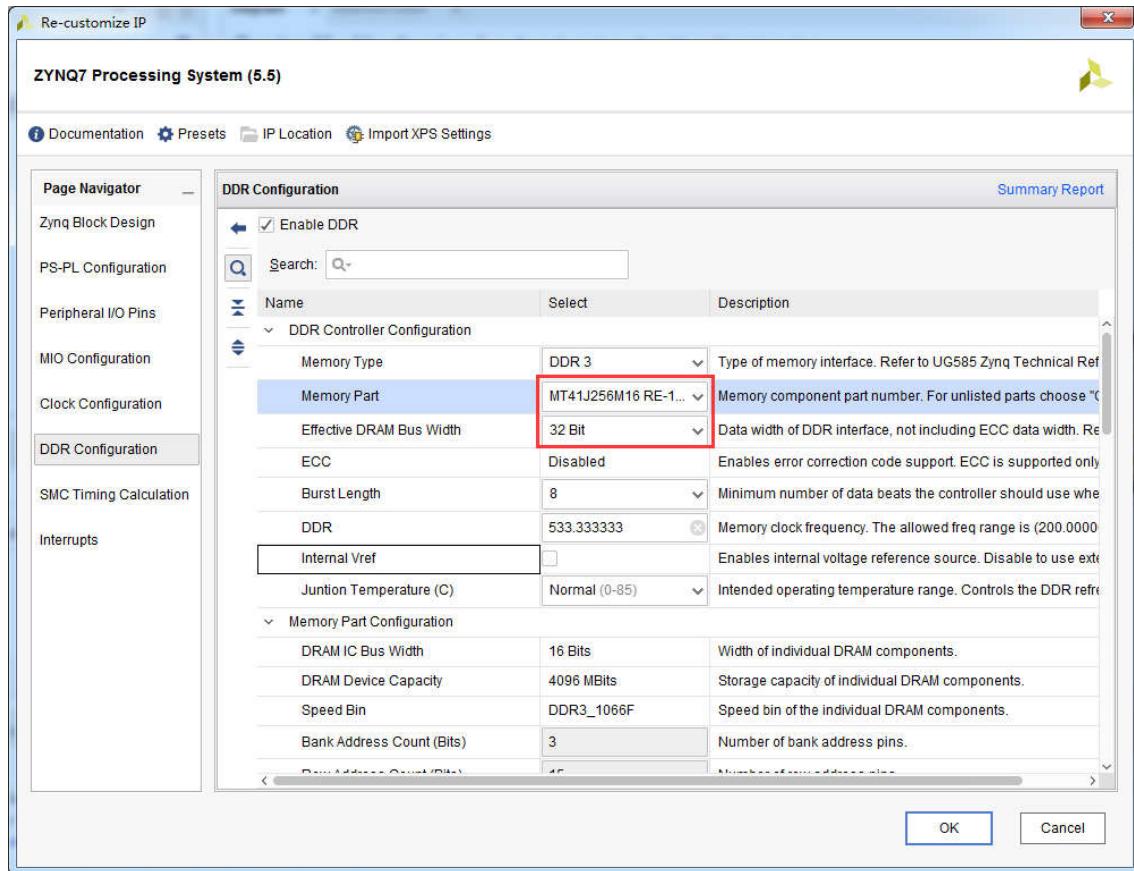
- 9) 从原理图中我们可以找到串口连接在 PS 的 MIO12-MIO13 上，所以在 “Peripheral I/O Pins” 选项中使能 UART1(MIO12-13) ,Bank 0 电压选择 “LVCMS 3.3V” ,Bank 1 电压选择 “LVCMS 1.8 V” , 本实验仅仅使用了一个串口功能，这里就不再使能其他设备。



- 10) 在“Clock Configuration”选项卡中我们可以配置 PS 时钟输入频率，这里默认是 33.333333，和板子上一致，不用修改，CPU 频率默认 666.666666MHz，这里也不修改。同时 PS 还可以给 PL 端提供 4 路时钟，频率可以配置，这里不需要，所以保持默认即可。



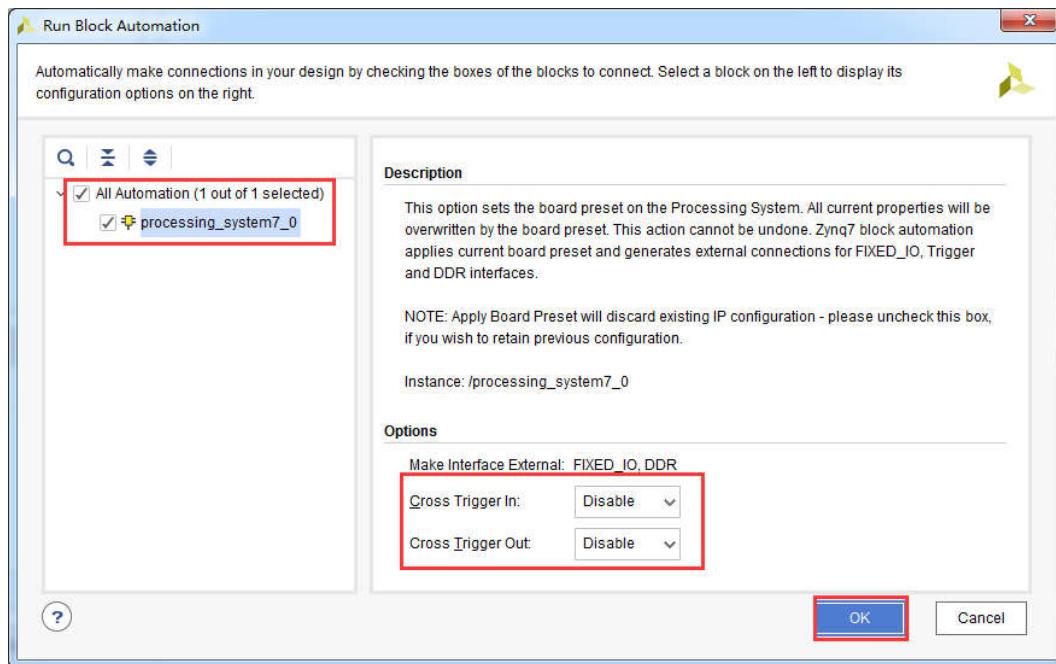
- 11) 在“DDR Configuration”选项卡中可以配置 PS 端 ddr 的参数，“Memory Part”选择“MT41J256M16 RE-125”，“Effective DRAM Bus Width”，选择“32 Bit”，到此配置完成，点击“OK”



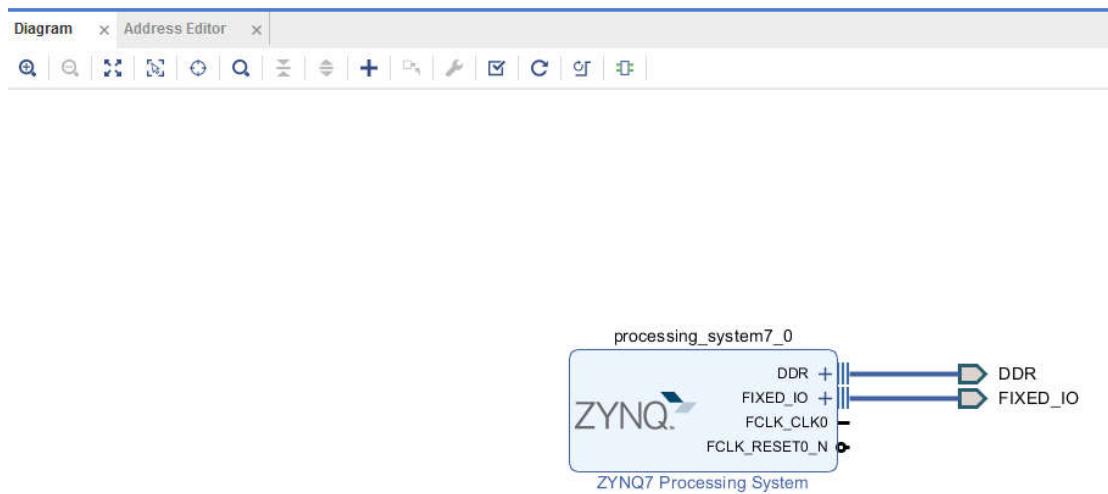
12) 点击“Run Block Automation”，vivado 软件会自动完成一些导出端口的任务



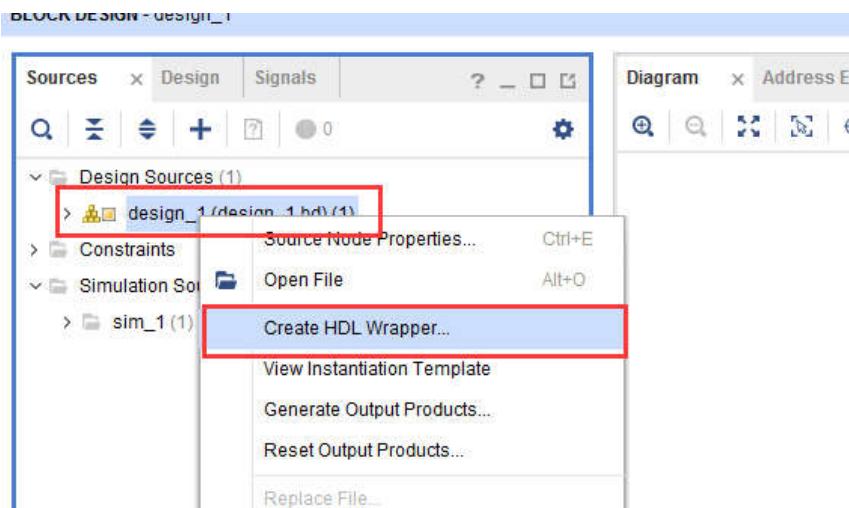
13) 点击“OK”



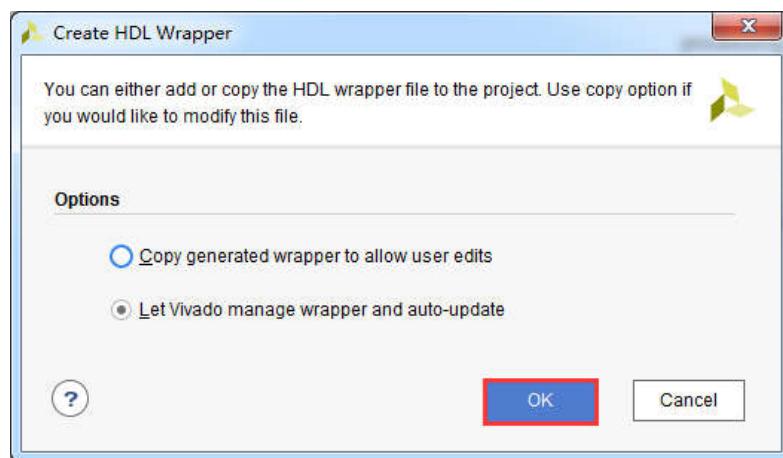
- 14) 点击“OK”以后我们可以看到 PS 端导出一些管脚 ,包括 DDR 还有 FIXED_IO ,按键 “Ctrl + s” 保存设计



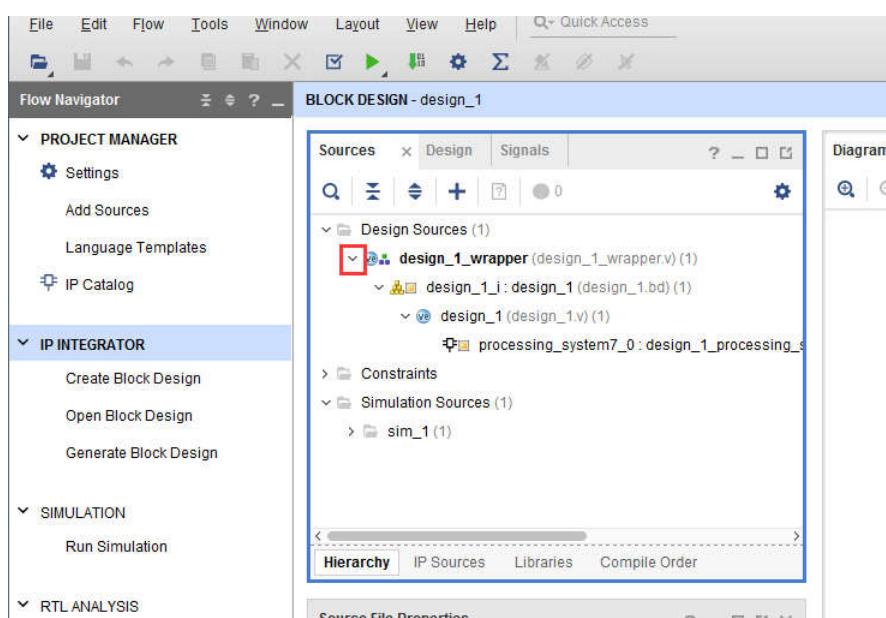
- 15) 选择 Block 设计 , 右键 “Create HDL Wrapper...” , 创建一个 Verilog 或 VHDL 文件



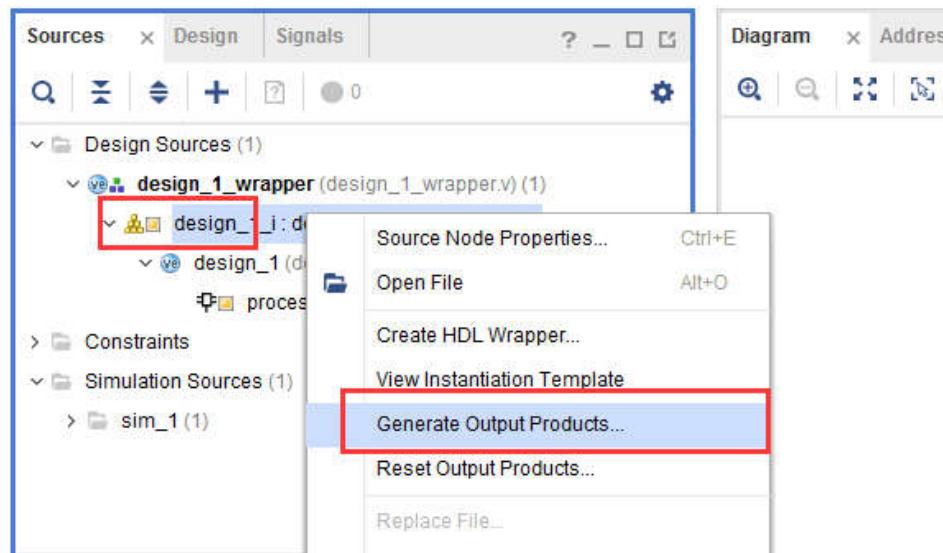
16) 保持默认选项，点击“OK”



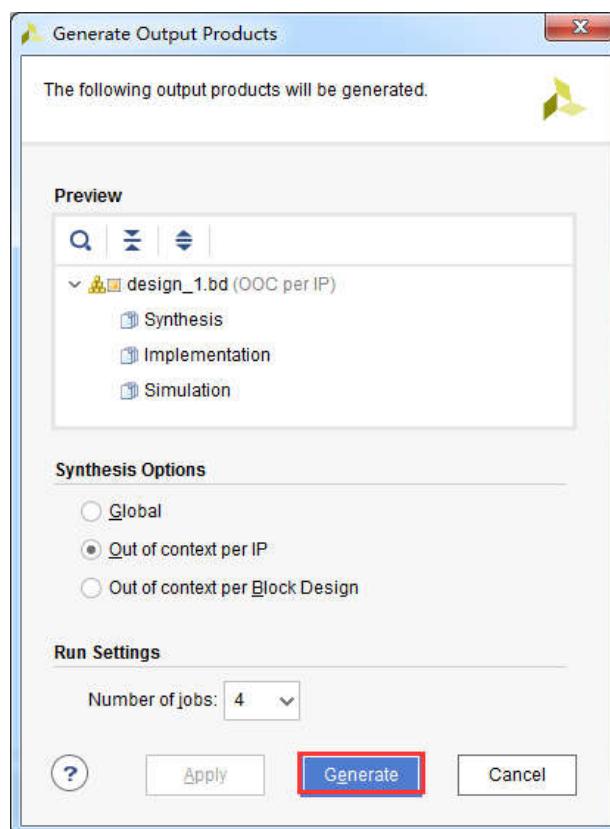
17) 展开设计可以看到 PS 被当成一个普通 IP 来使用。



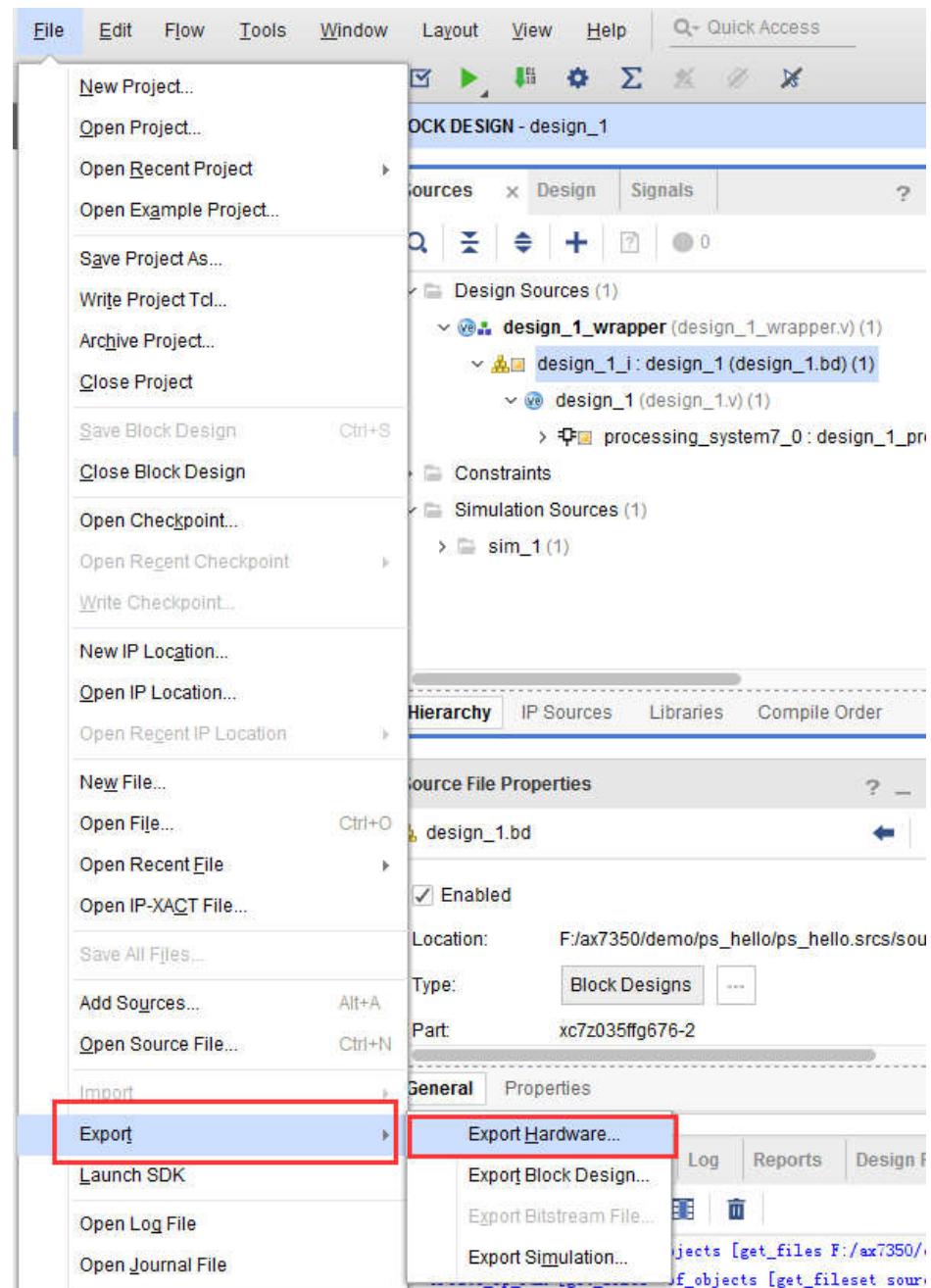
18) 选择 block 设计，右键 “Generate Output Products”



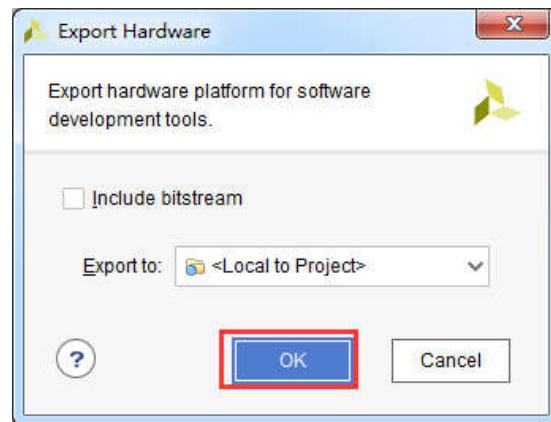
19) 点击 “Generate”



20) 在菜单栏 “File -> Export -> Export Hardware...” 导出硬件信息，这里就包含了 PS 了的配置信息。

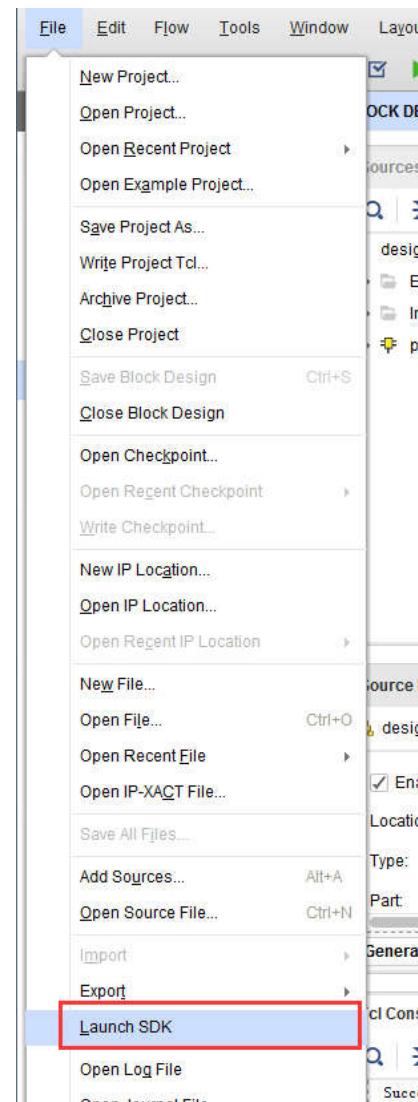


- 21) 在弹出的对话框中点击“OK”，因为实验仅仅是使用了 PS 的串口，不需要 PL 参与，这里就没有使能“Include bitstream”

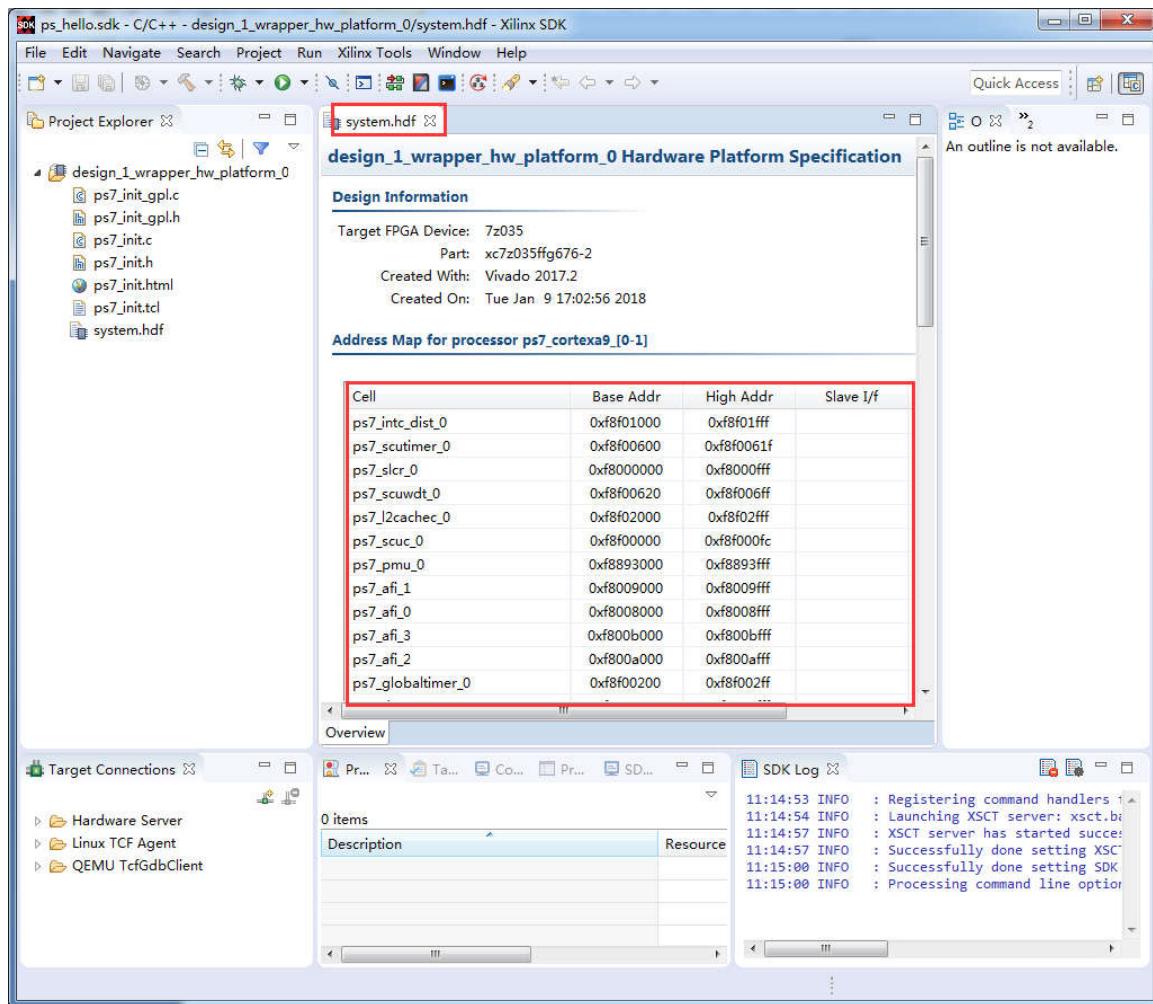


9.3 SDK 调试

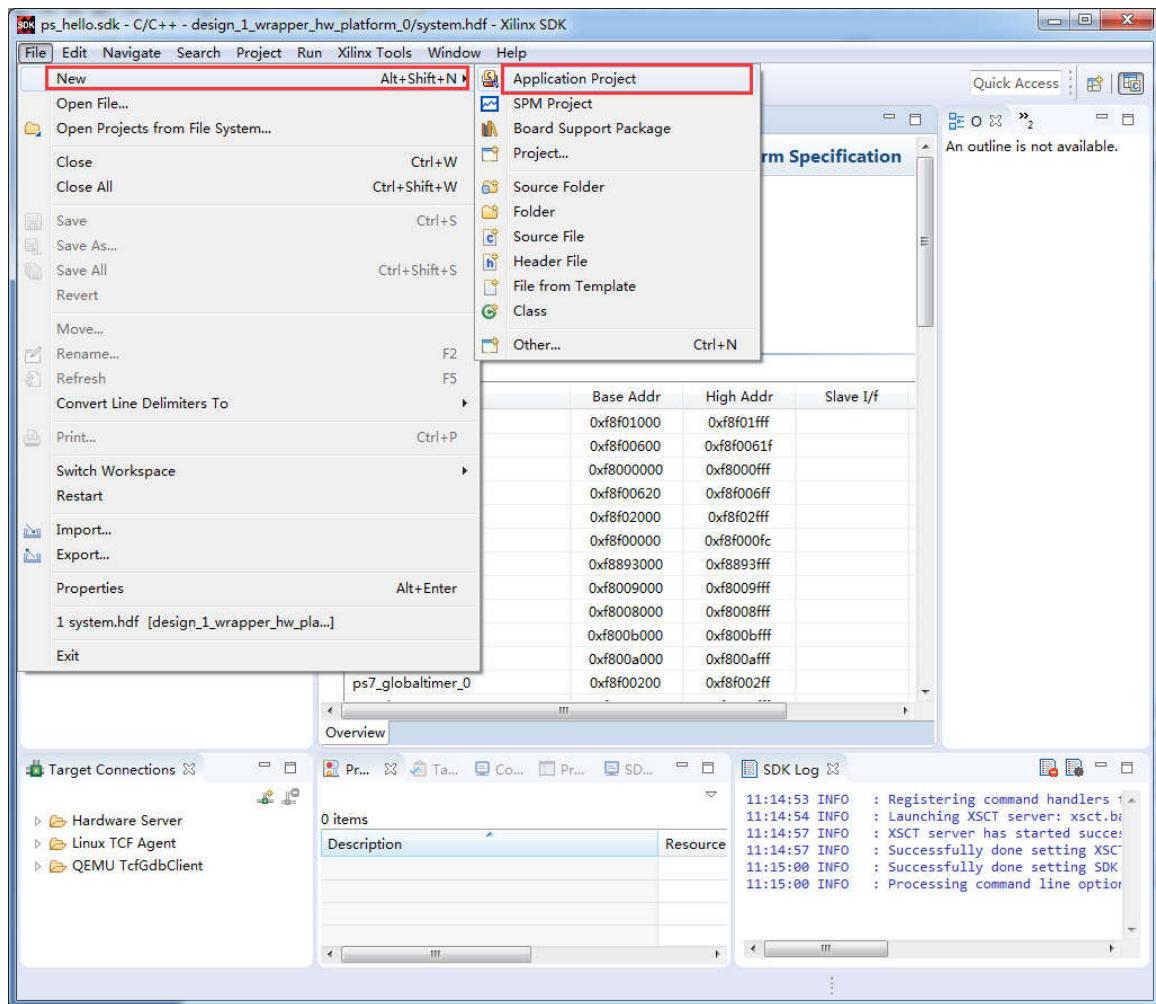
- 1) 点击 Vivado 菜单 "File -> Launch SDK"，启动 SDK



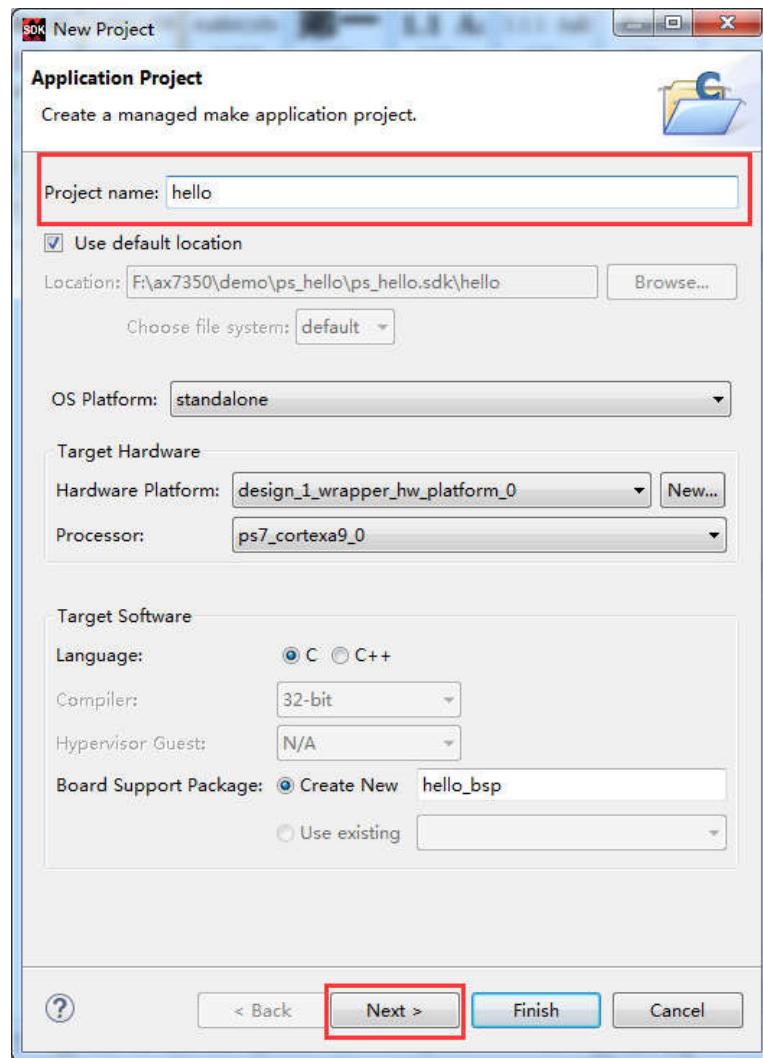
- 2) 启动 SDK 后我们会看到一个文件夹，有一个名为“system.hdf”文件，这个文件就包含了 Vivado 硬件设计的信息，可以给软件开发使用，也可以看到 PS 端外设的寄存器列表。



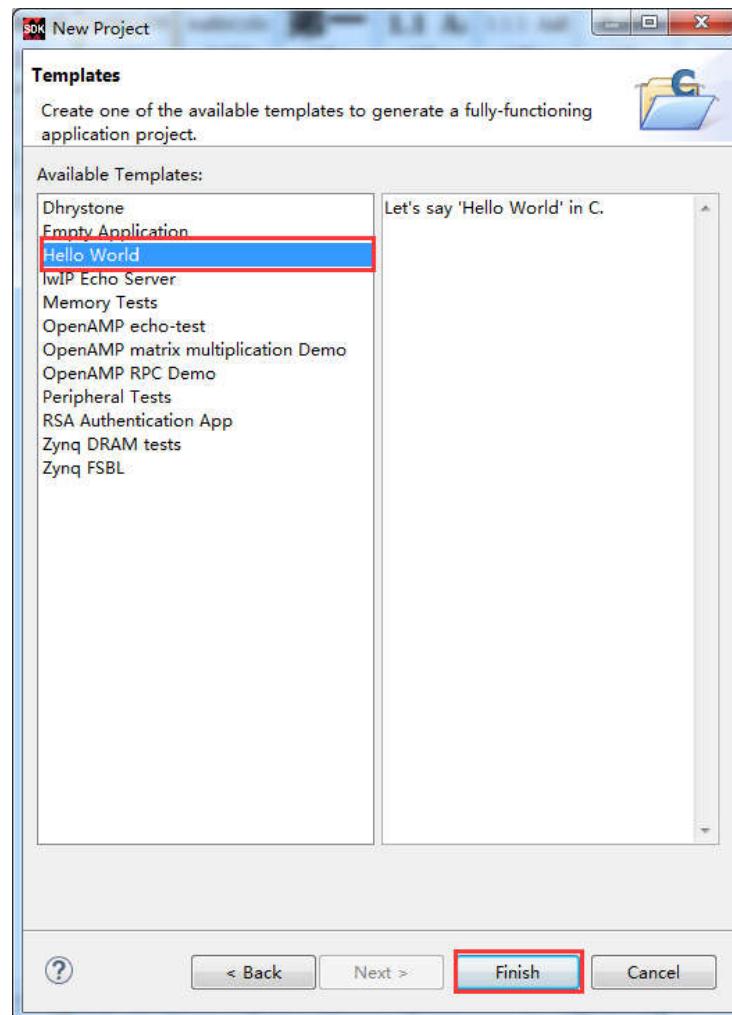
- 3) 在 SDK 的菜单 “New -> Application Project”，建立一个 APP 工程



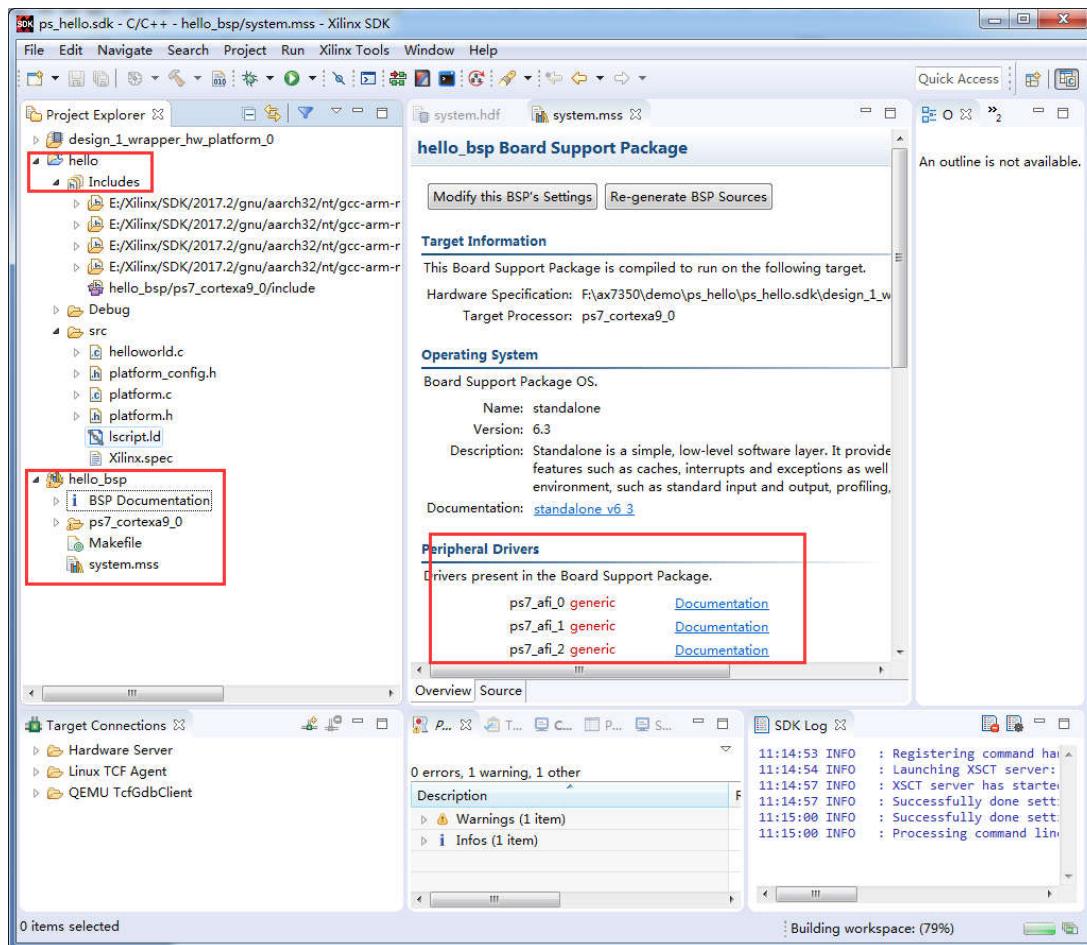
- 4) “Project name” 填写 “hello”，其他默认，点击 “Next”



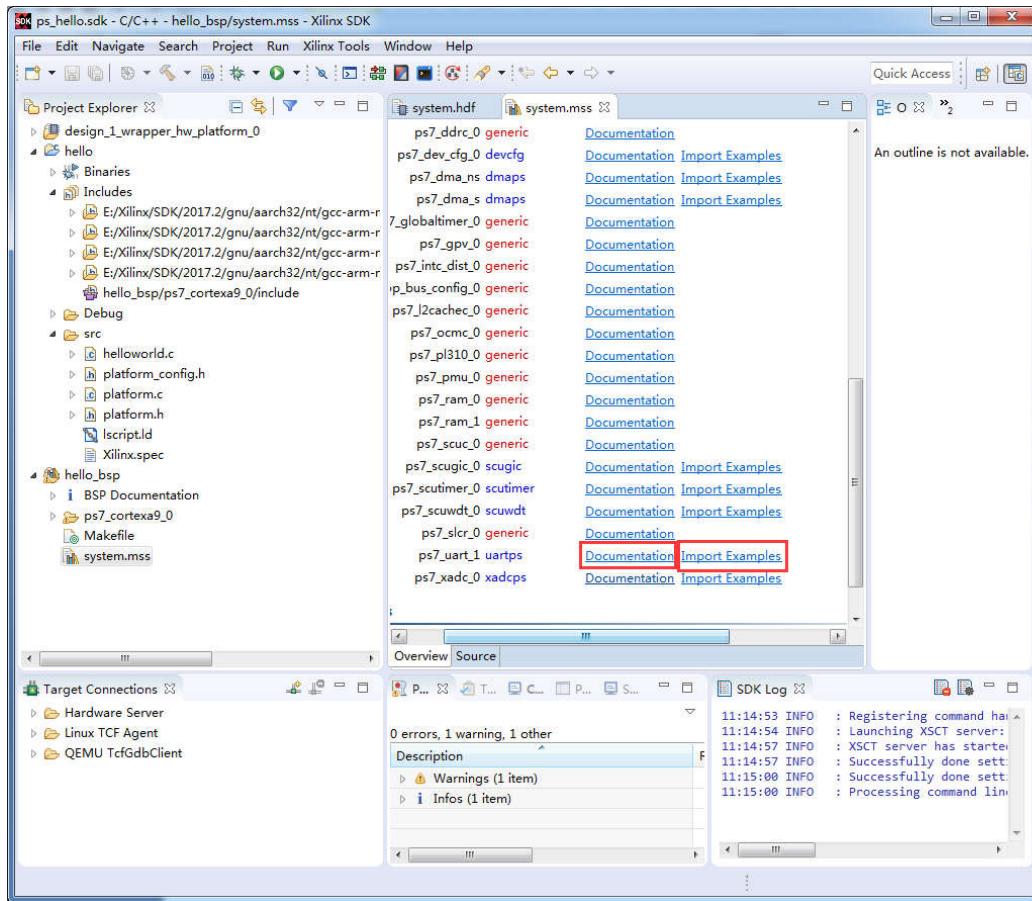
5) 模板选择 “Hello World” , 点击 “Finish”



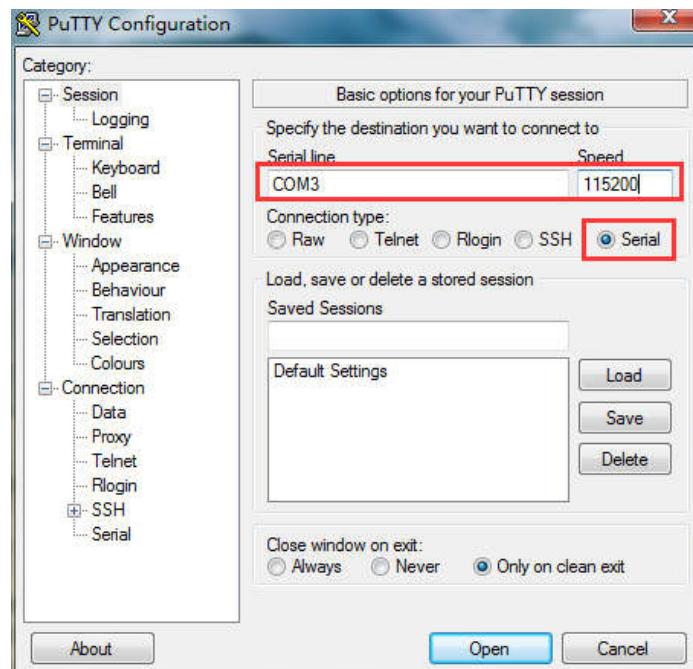
- 6) 可以看到 SDK 创建了一个 “hello” 目录，还有一个 “hello_bsp” 的目录，在 “hello_bsp” 目录中可以找到很多有用的信息，其中有 “BSP Documentation” 包含了一些 PS 外设的 API 说明。



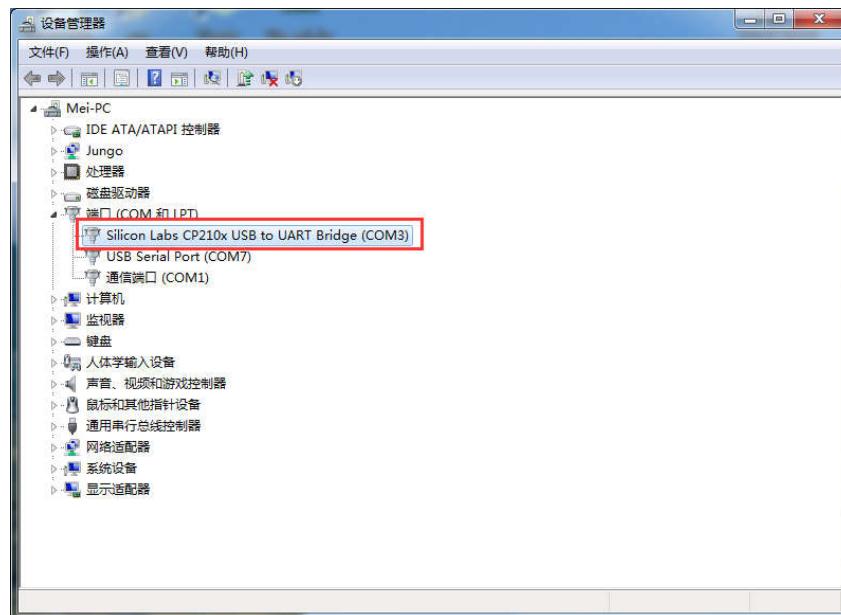
- 7) 双击“system.mss”，还可以看到有些PS外设还提供了例程，这是用来了解学习这些外设的第一手资料。



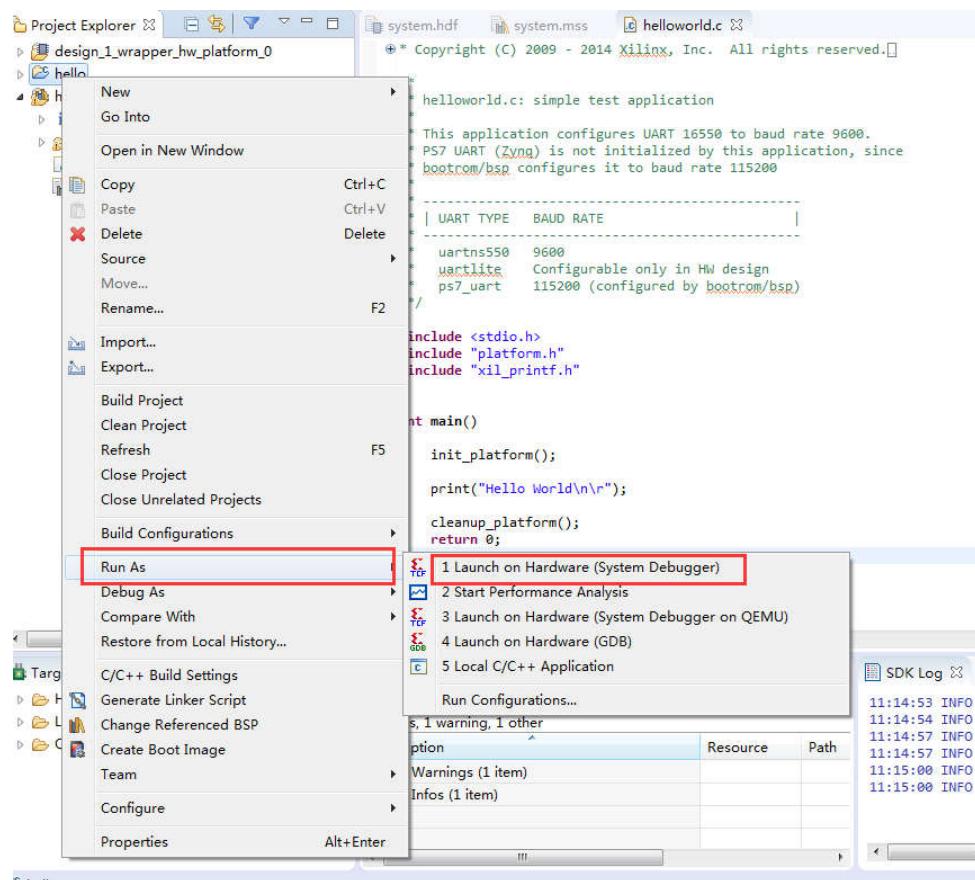
- 8) 连接 JTAG 的 USB 线、UART 的 USB 线到 PC
- 9) 使用 PuTTY 软件做为串口终端调试工具，PuTTY 是一个免安装的小软件



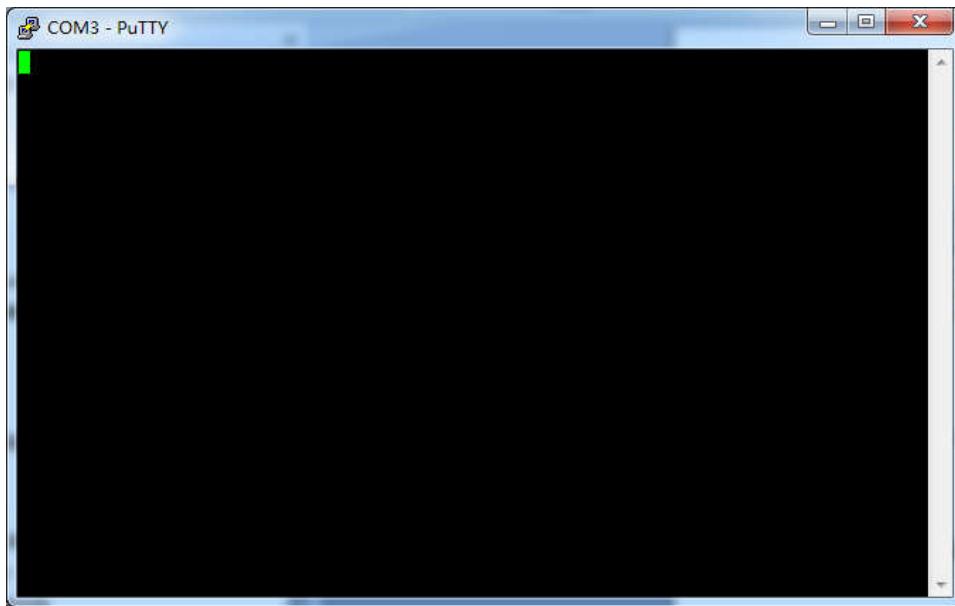
- 10) 选择 Serial , Serial line 填写 COM3 , Speed 填写 115200 , 串口号根据设备管理器里显示的填写 , 点击 “Open”



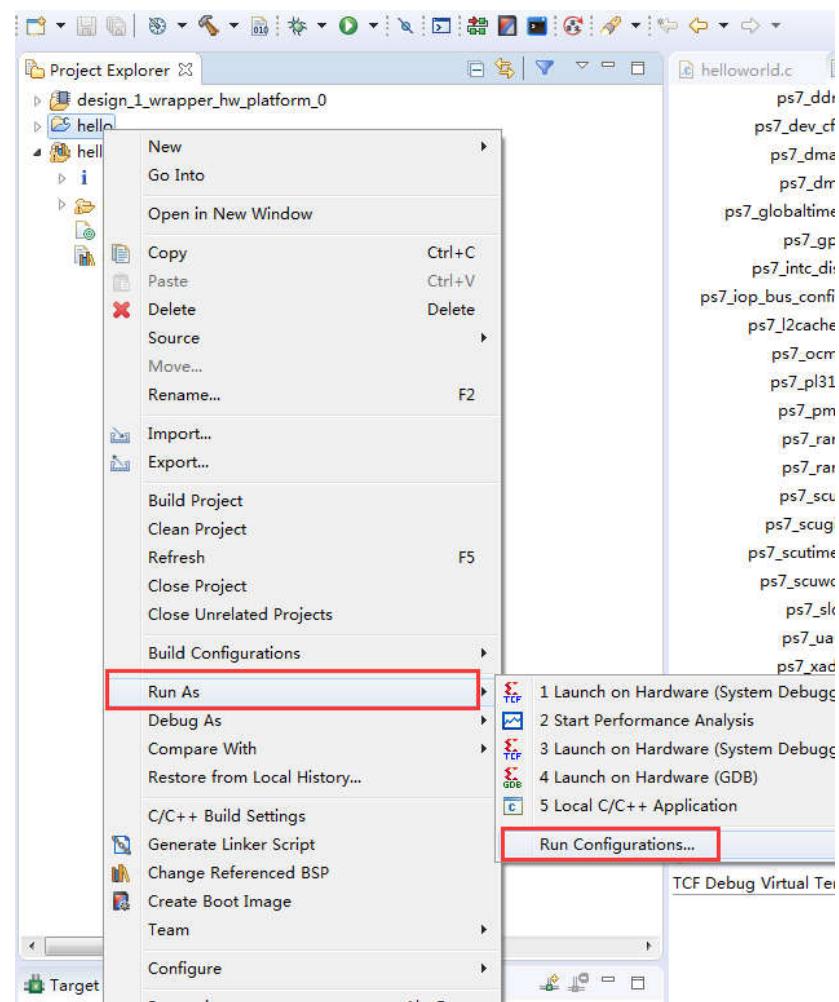
- 11) 给开发板上电，准备运行程序，开发板出厂时带有程序，**这里可以把运行模式选择 JTAG 模式，然后重新上电**。选择“hello”，右键，可以看到很多选项，本实验要用到这里的“Run as”，就是把程序运行起来，“Run as”里又有很对选项，选择第一个“Launch on Hardware(System Debugger)”，使用系统调试，直接运行程序。



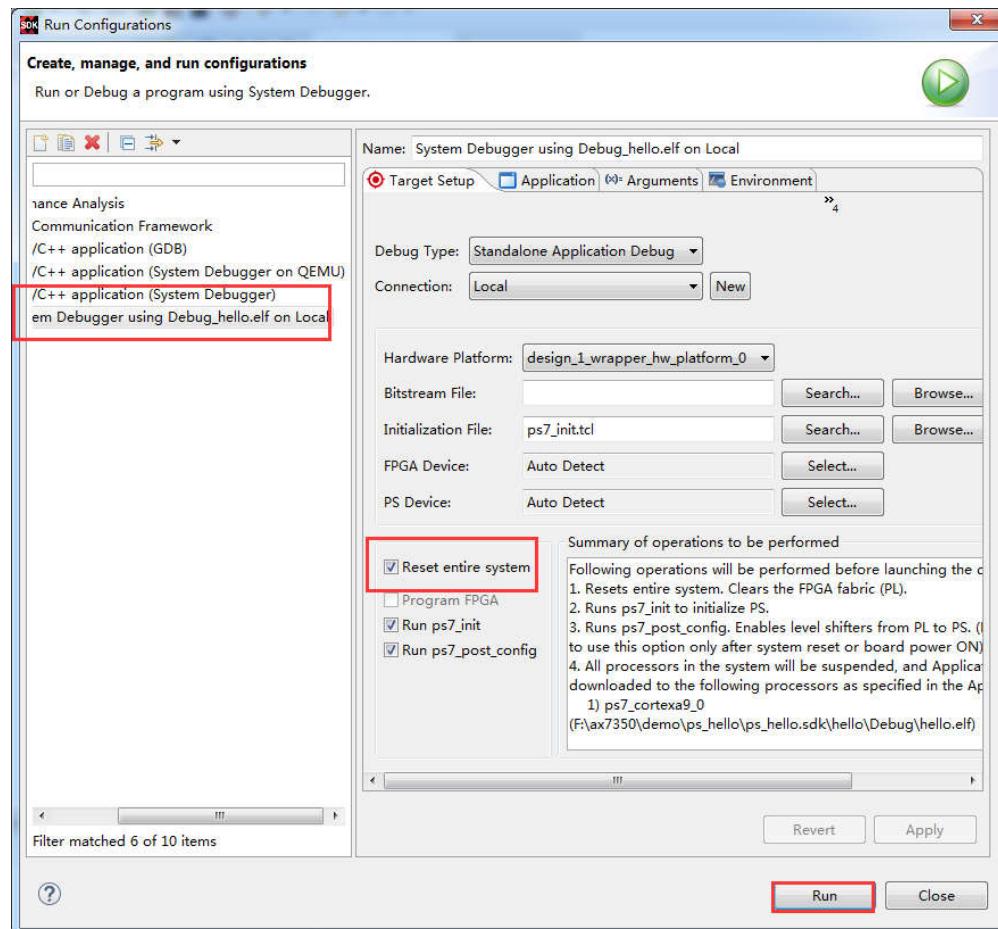
12) 这个时候观察 PuTTY 软件，可能有输出显示，也可能没有



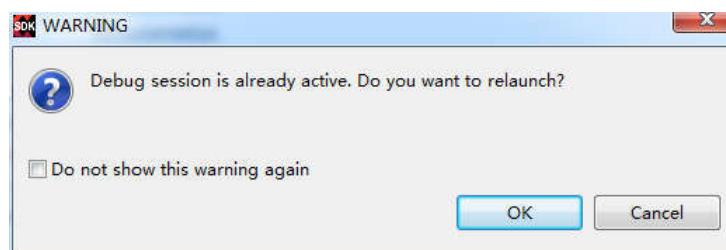
13) 为了保证系统的可靠调试，需要添加一个配置，右键 “Run As -> Run Configuration...”



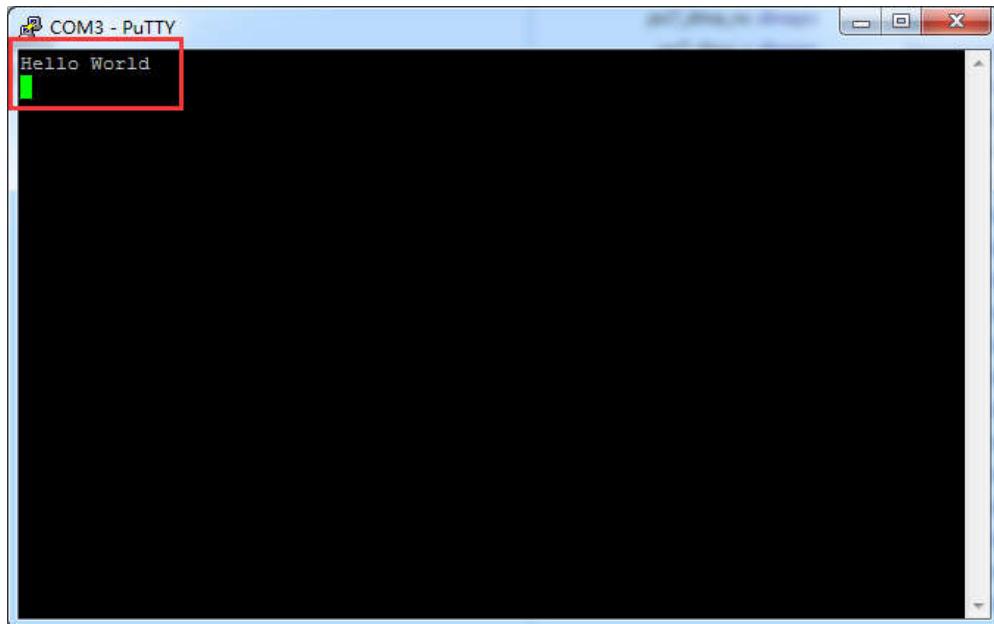
14) 选择“Reset entire system”复位整个系统，如果系统中还有 PL 设计，还必须选择“Program FPGA”，再次点击“Run”



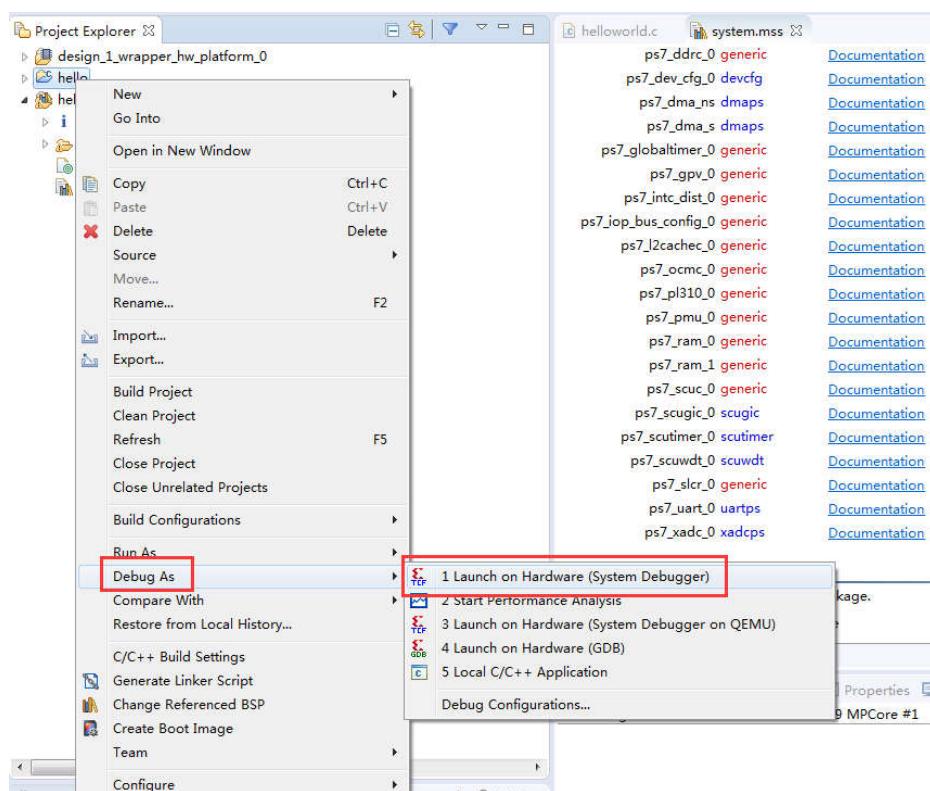
15) 点击“OK”，确认重新运行



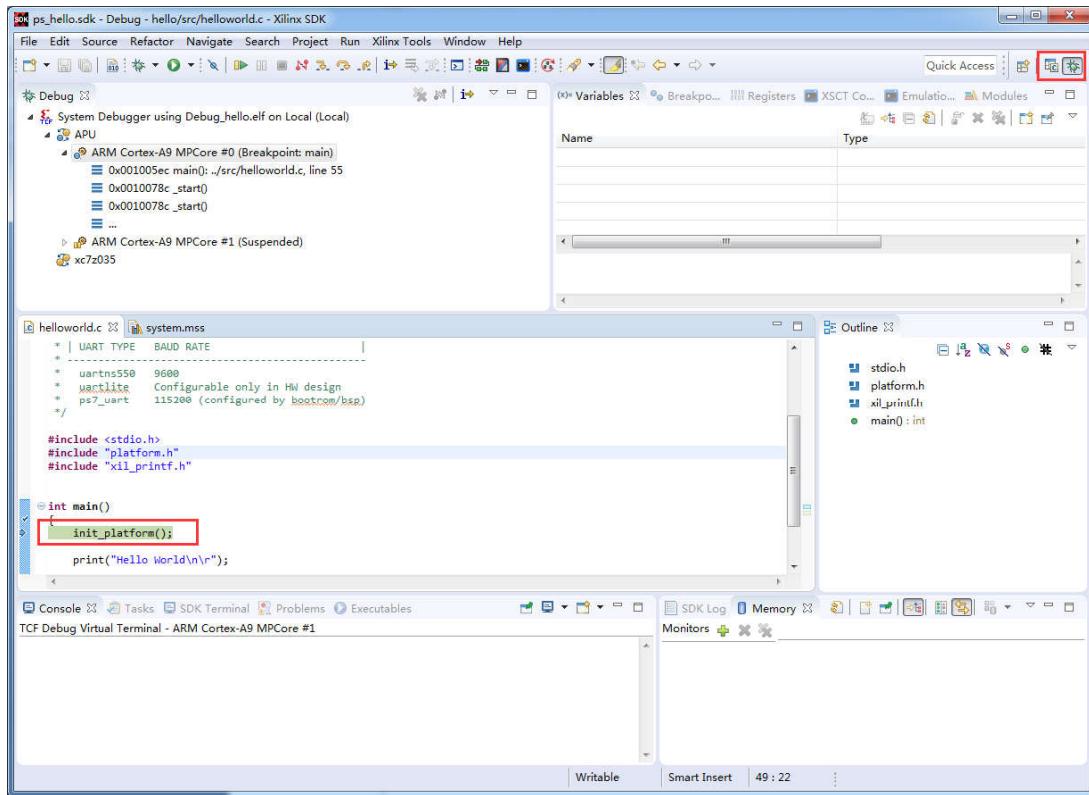
16) 这次就可以看到熟悉“Hello World”显示出来了



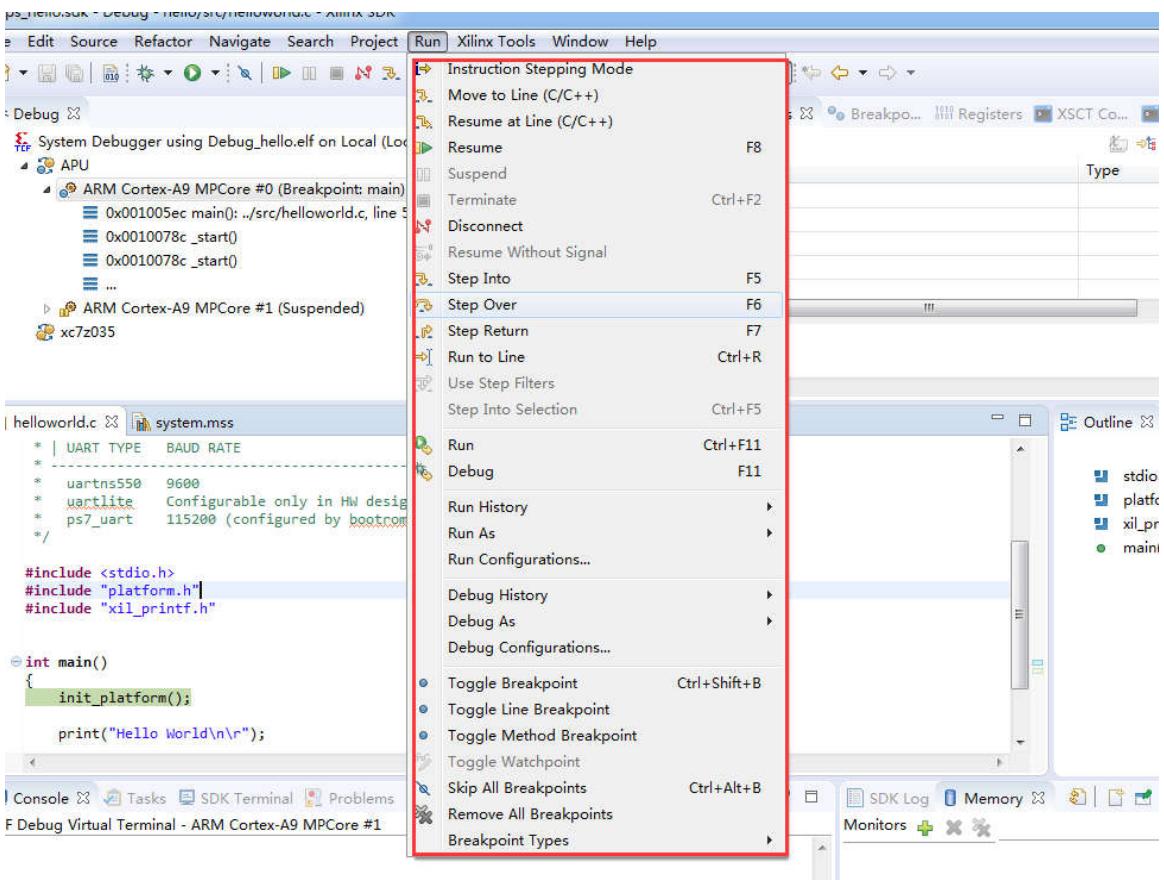
17) 除了“Run As”，还可以“Debug As”，这样可以设置断点，单步运行



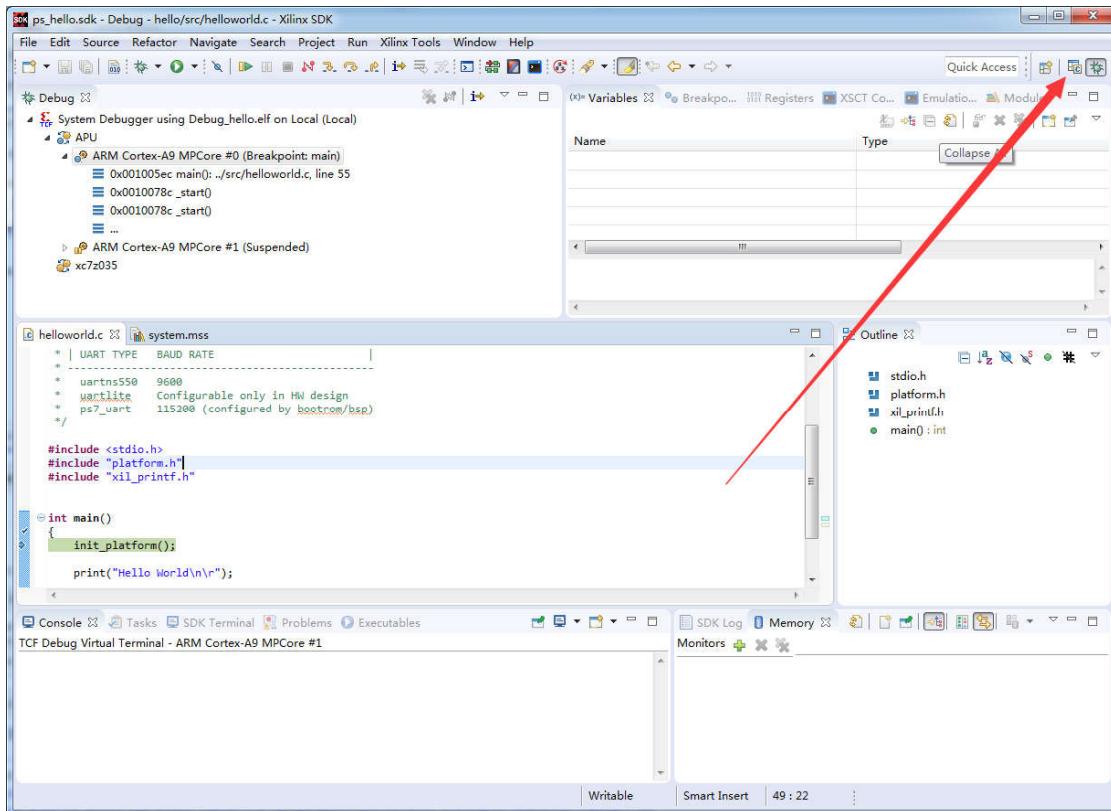
18) 进入 Debug 模式



19) 和其他 C 语言开发 IDE 一样，可以逐步运行、设置断点等



20) 右上角可以切换 IDE 模式



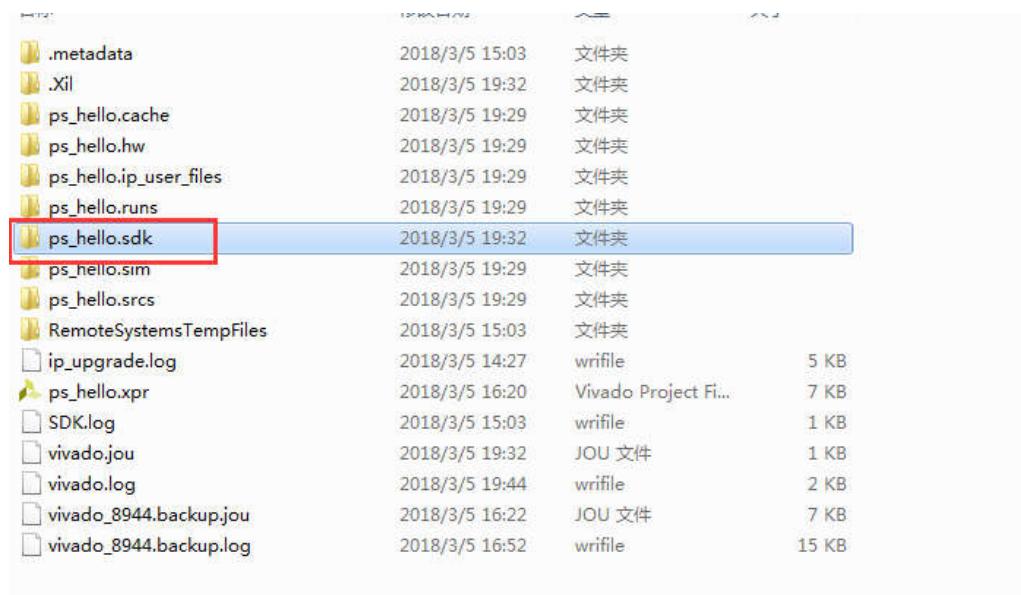
9.4 实验总结

本实验使用一个简单的 Hello World 讲解了 SDK 的使用，SDK 拥有很强大的功能，不能一一讲解，在我们不断的使用中逐渐掌握。

9.5 常见问题

9.5.1 通过 vivado 启动 SDK 后没有窗口弹出

- 1) 安装 Vivado 软件的时候一定要安装 SDK
- 2) 在启动 SDK 软件前就有 sdk 目录，可能会导致无法启动 SDK，删除这个目录再试



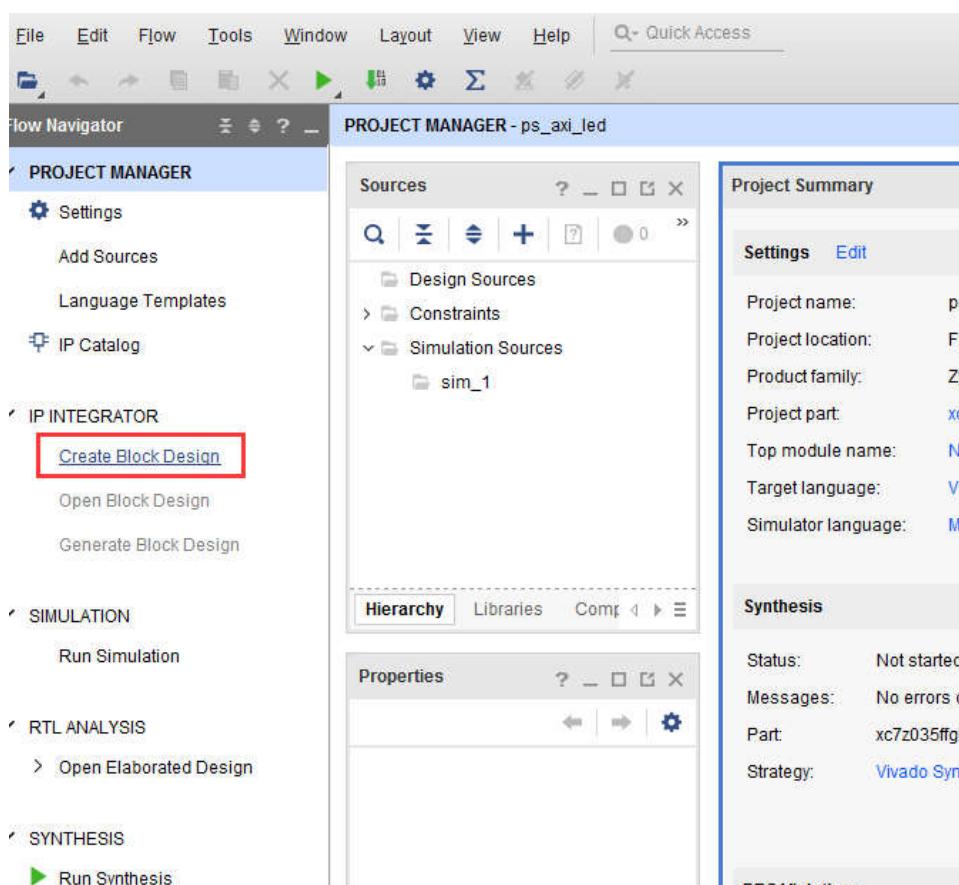
第十章 PS 点亮 PL 的 LED 灯

实验 Vivado 工程为 “ps_axi_led”。

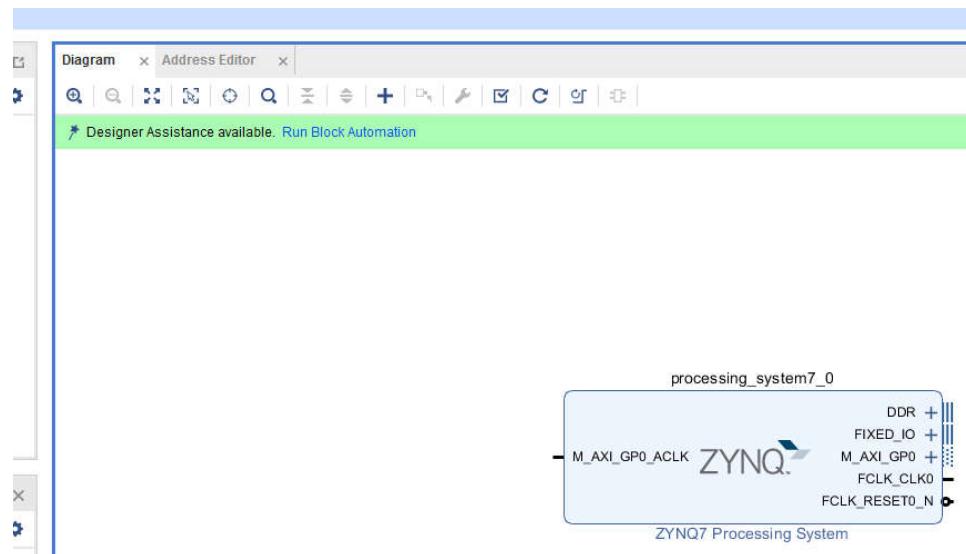
使用 zynq 最大的疑问就是如何把 PS 和 PL 结合起来使用，在其他的 SOC 芯片中一般都会有 GPIO，本实验使用一个 AXI GPIO 的 IP 核，让 PS 端通过 AXI 总线控制 PL 端的 LED 灯，实验虽然简单，不过可以让我们了解 PL 和 PS 是如何结合的。

10.1 Vivado 工程建立

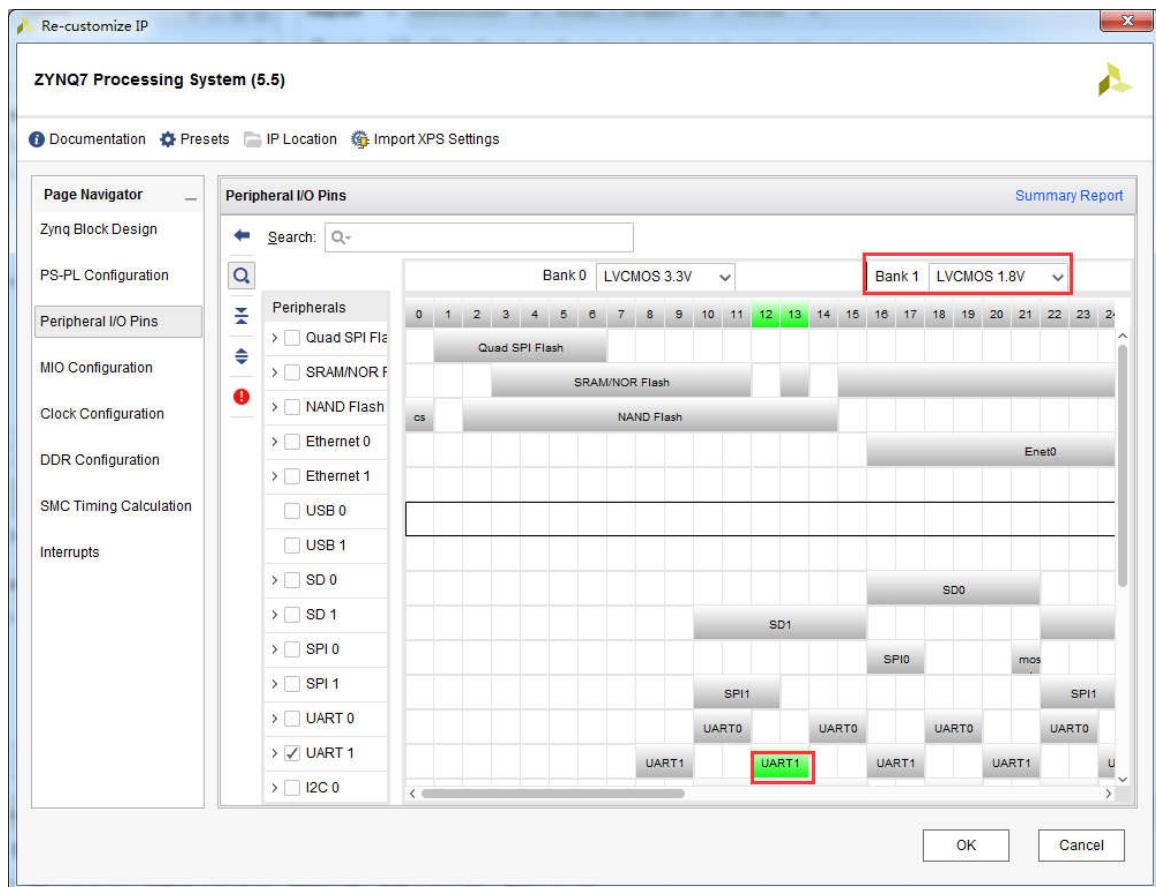
- 1) 建立一个名为 “ps_axi_led” Vivado 工程，表示 PS 通过 AXI 总线控制 LED 灯
- 2) 创建一个 Block 设计



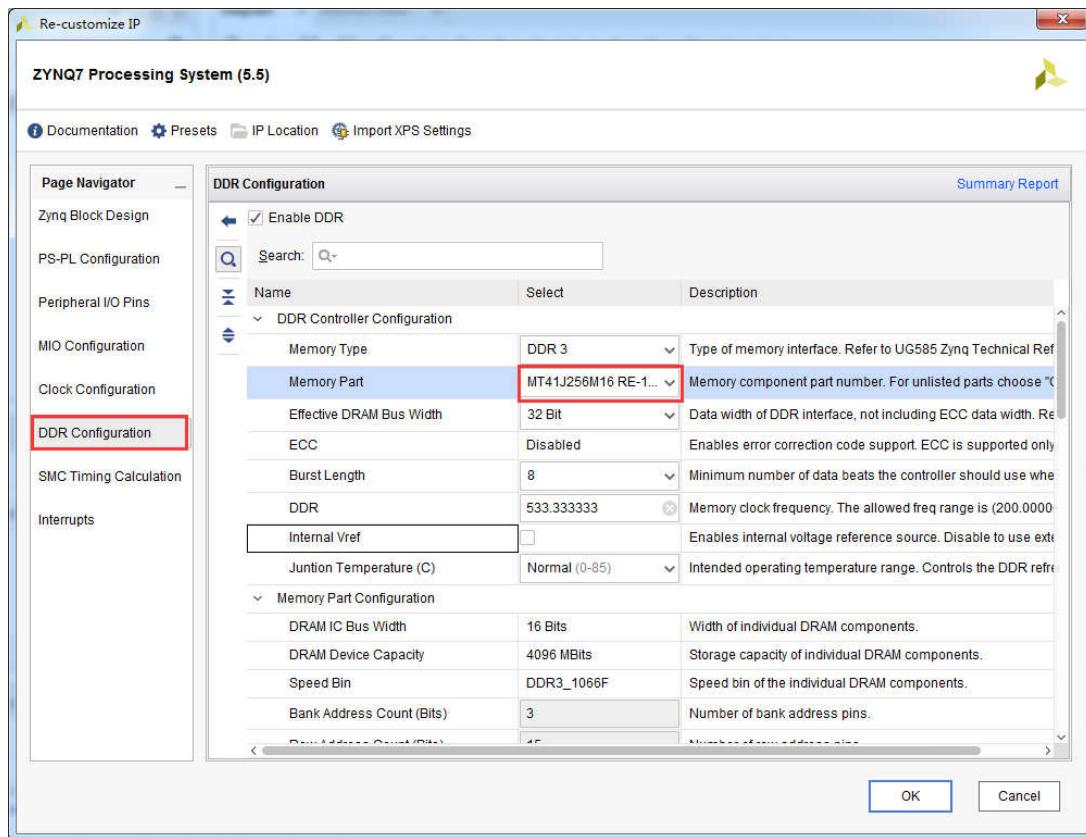
- 3) 添加 ZYNQ 处理器



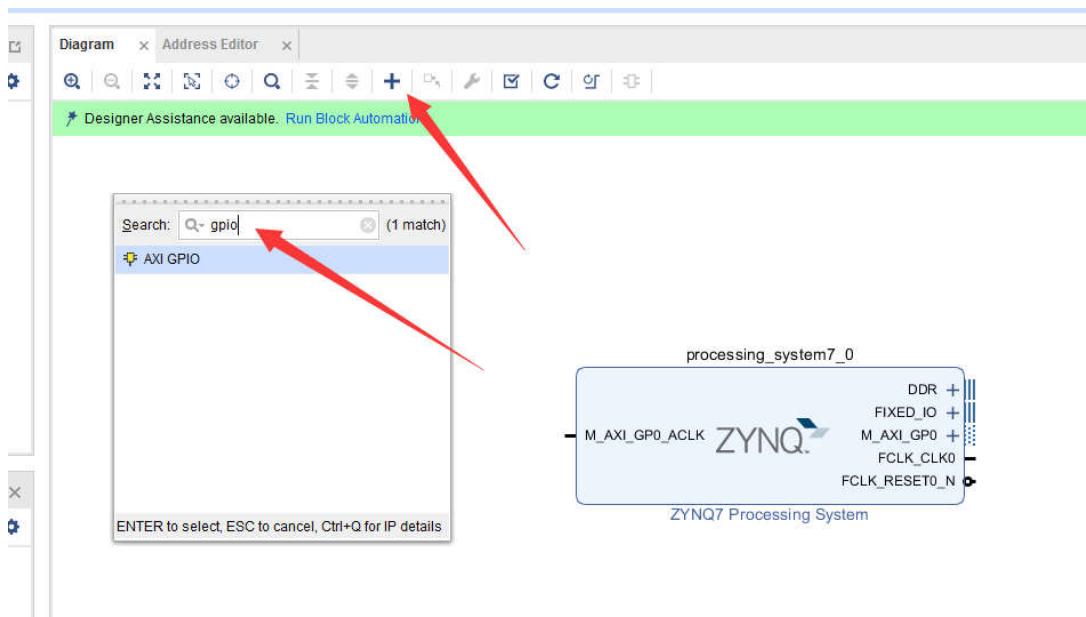
4) 配置 Bank 电平标准，使能串口



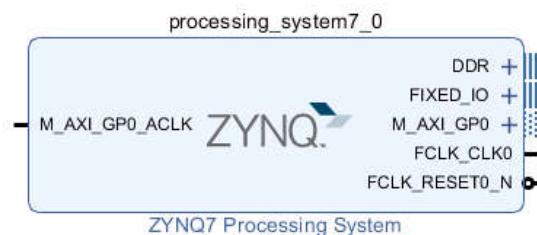
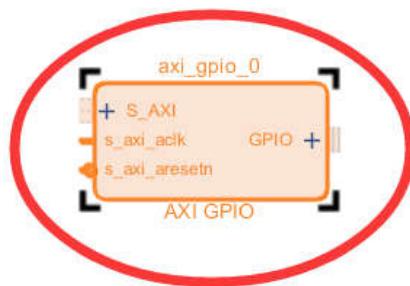
5) 配置 DDR3 型号为 “MT41J256M16 RE-125”



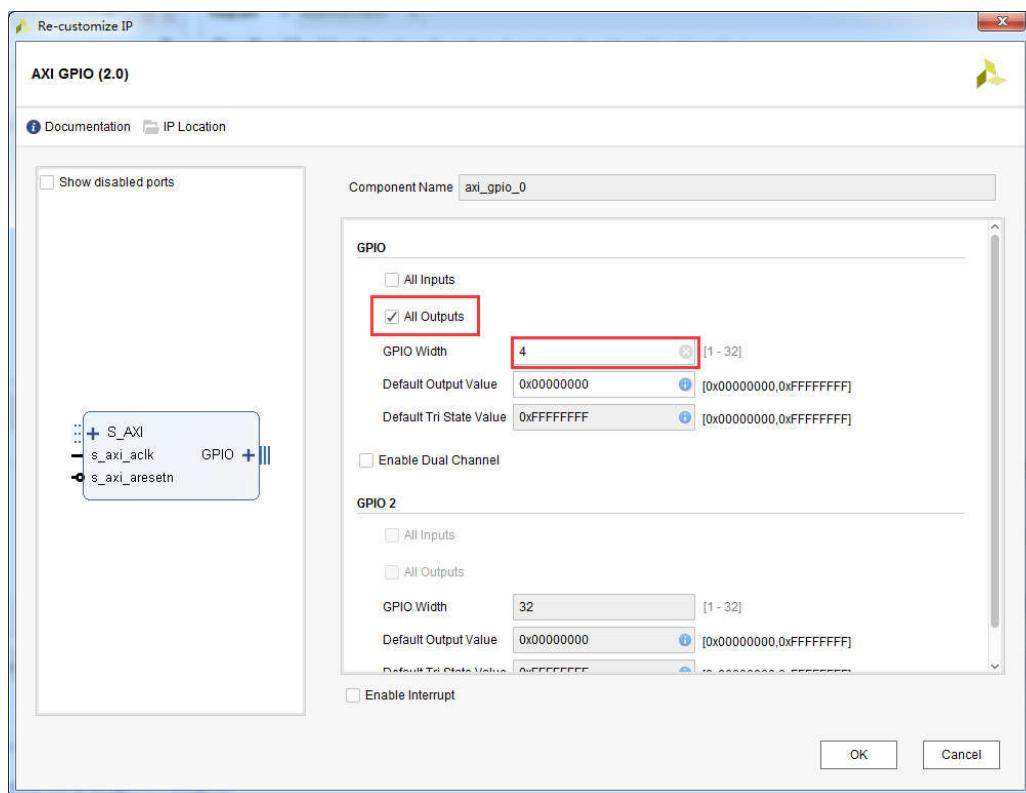
6) 添加一个 AXI GPIO 的 IP 核



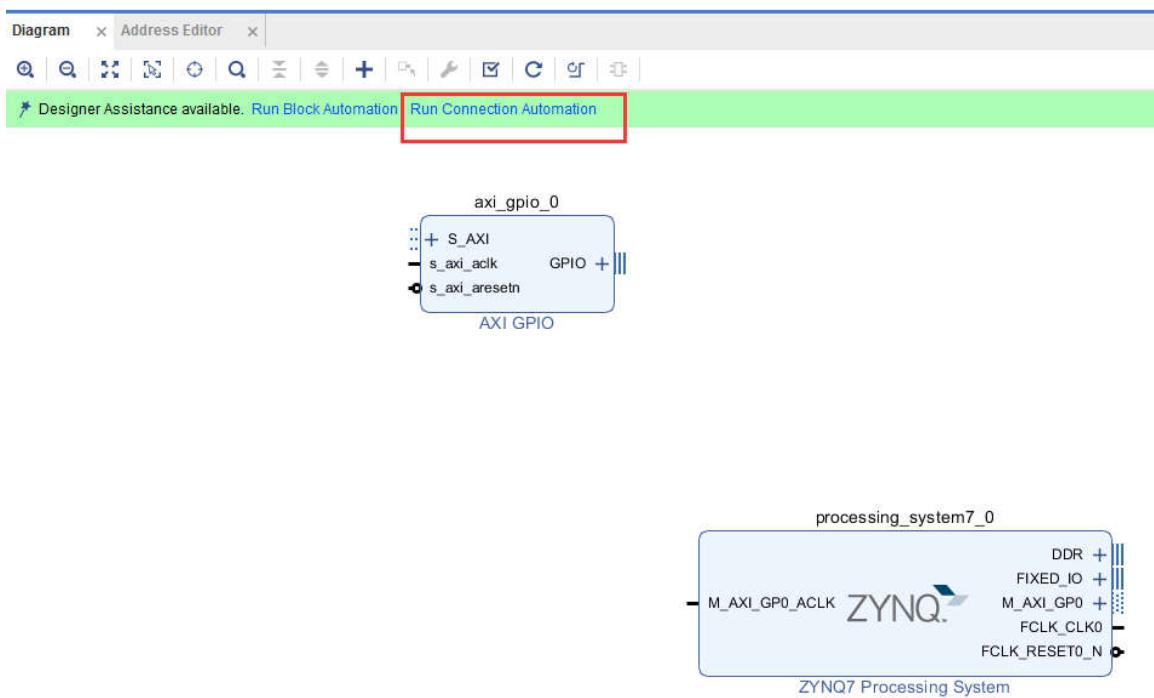
7) 双击刚才添加的“axi_gpio_0”配置参数



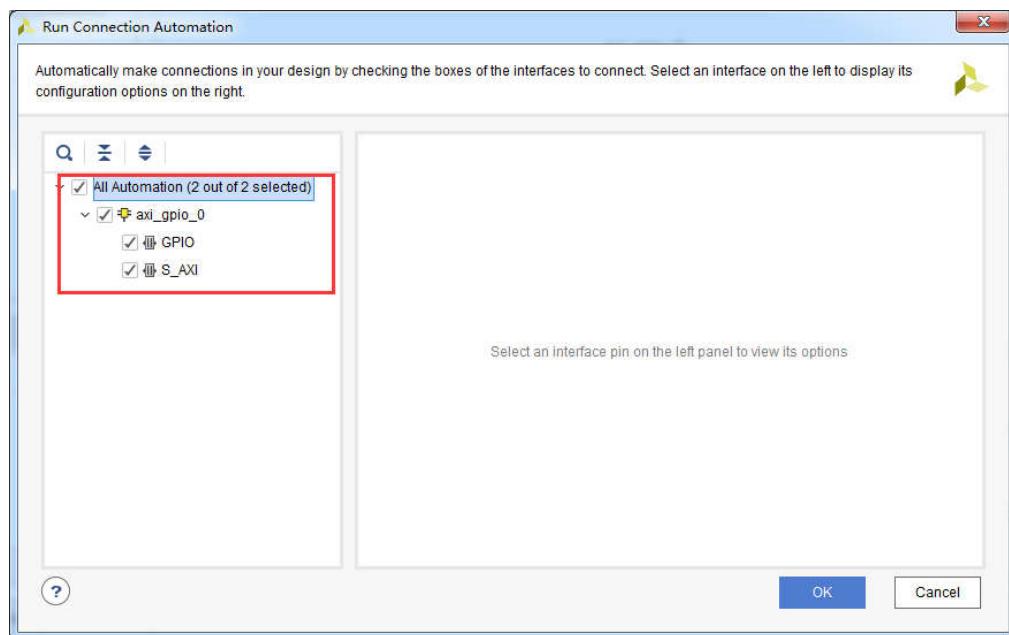
- 8) 选择 “All Outputs” , 因为这里控制 LED , 只要输出就可以了 , “GPIO Width” 填 4 , 控制 4 颗 LED , 点击 OK



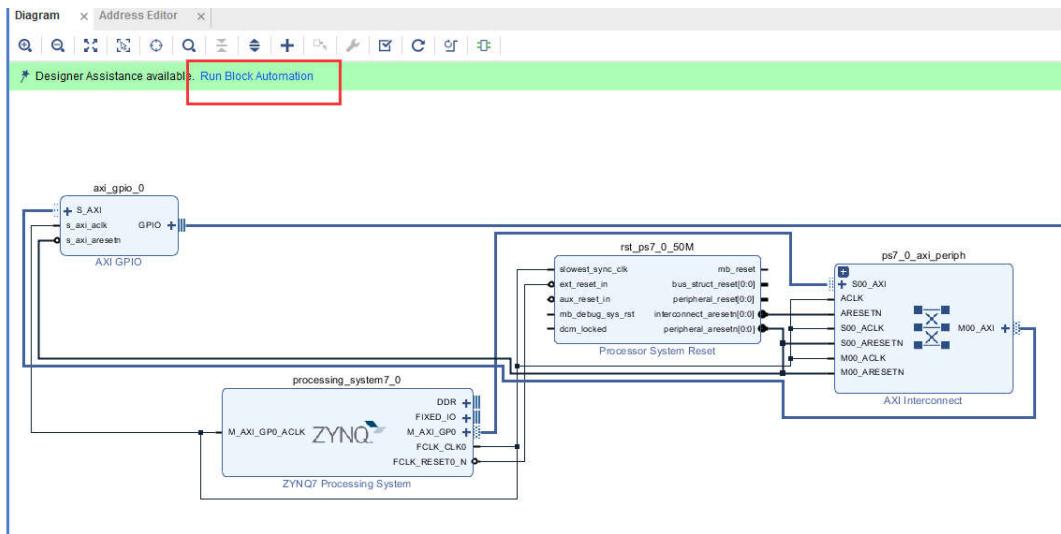
- 9) 点击 “Run Connection Automation” , 可以完成部分自动连线



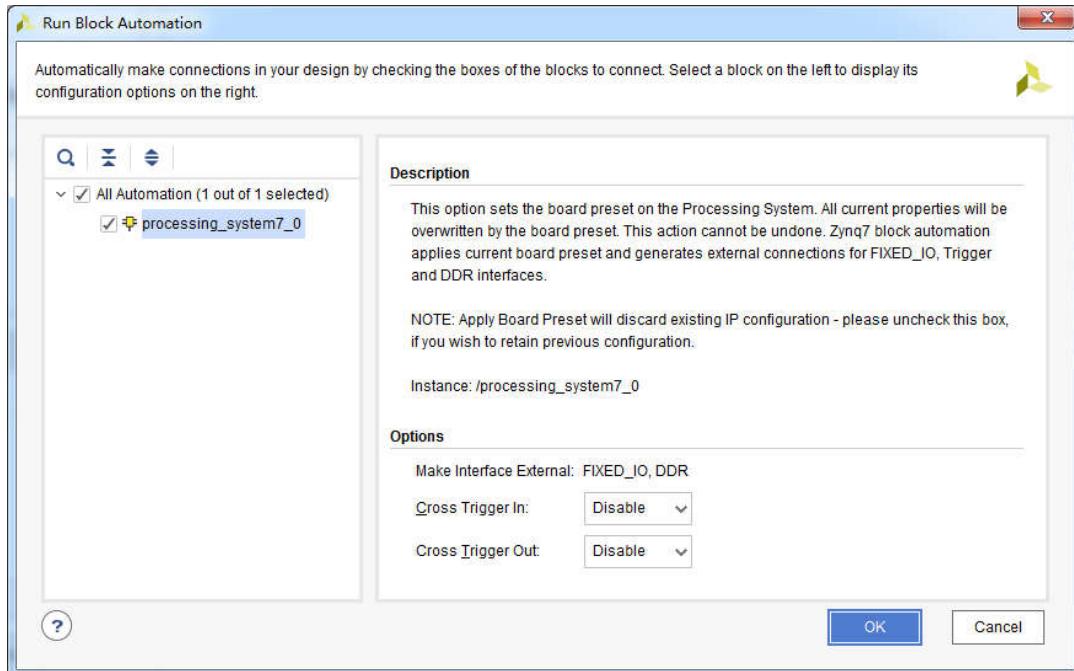
10) 选择要自动连接的端口，这里全选，点击 OK



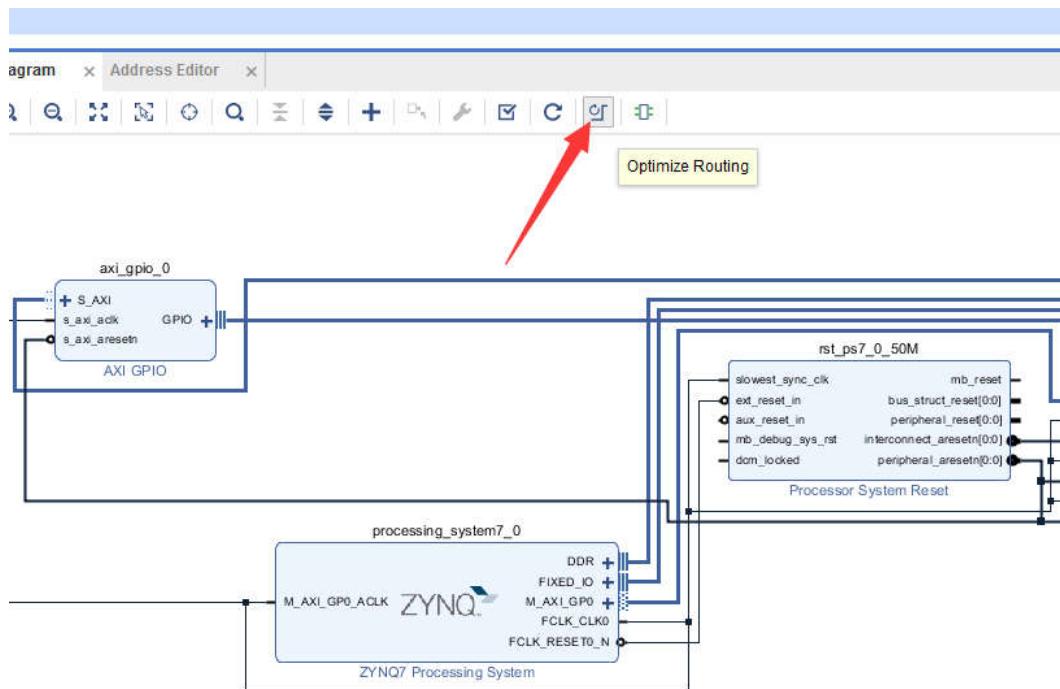
11) 点击 “Run Block Automation”



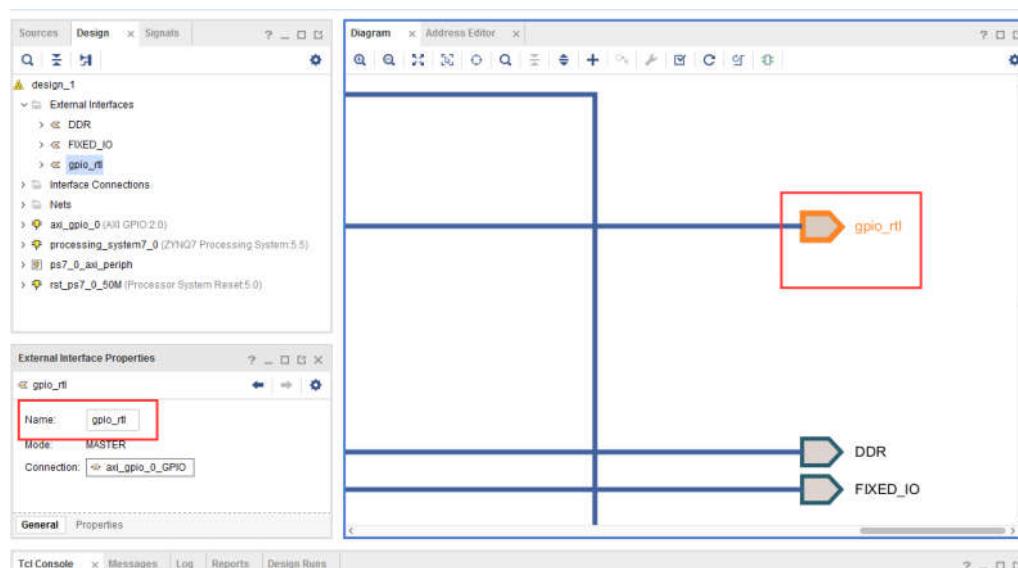
12) 点击 “OK”



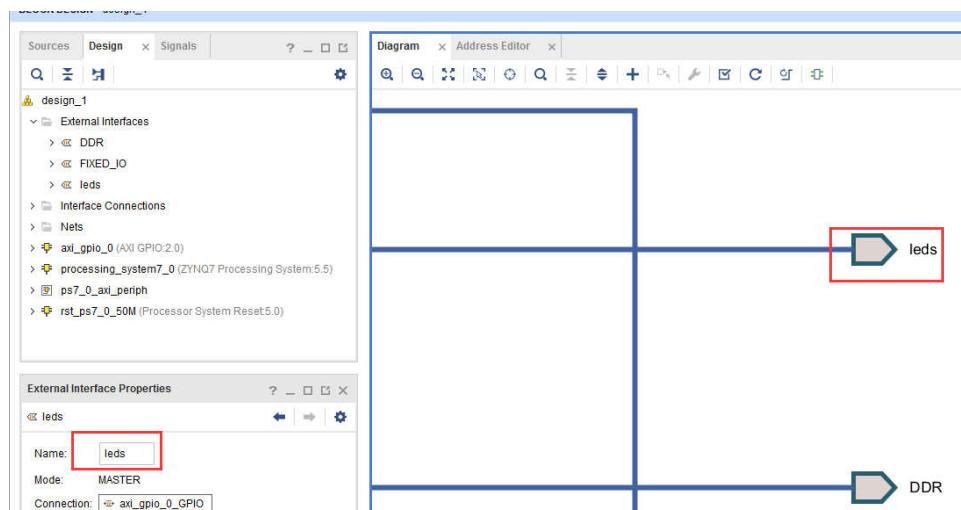
13) 点击 “Optimize Routing”，可以优化布局



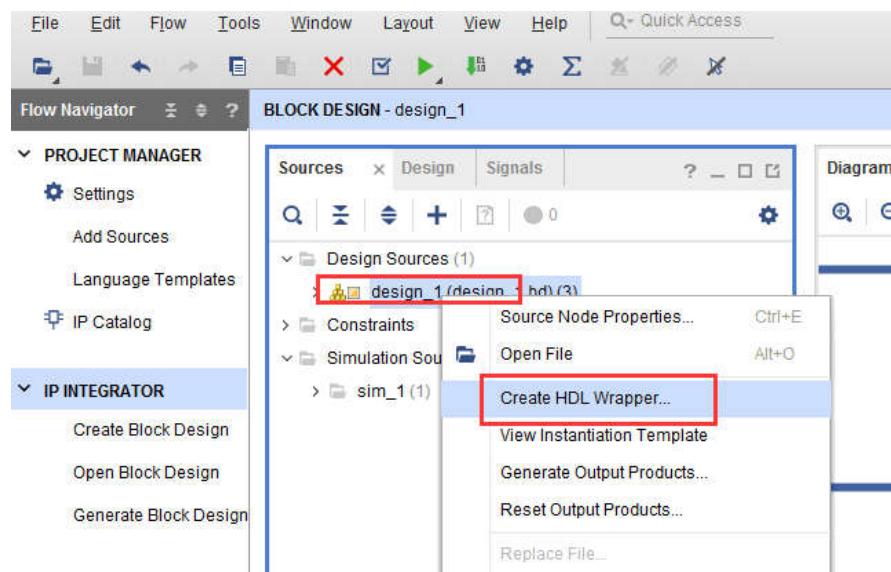
14) 修改 GPIO 端口的名称



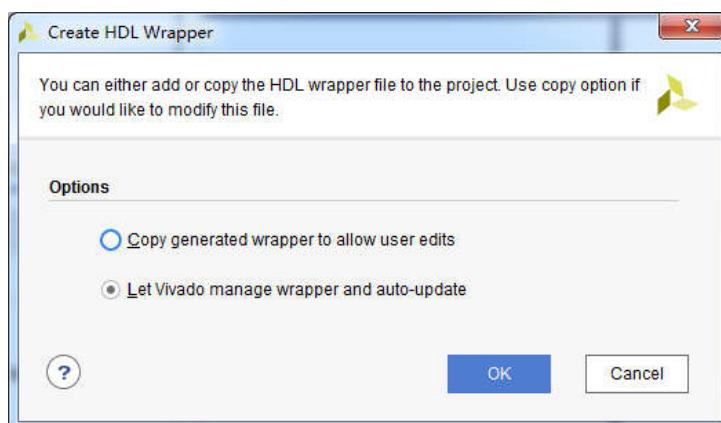
15) 名称修改为 leds



16) 创建 HDL 文件



17) 点击 “OK”



18) 在生成的 Verilog 文件中，可以看到有个 “leds_tri_o” 的输出端口，要为他们分配管脚

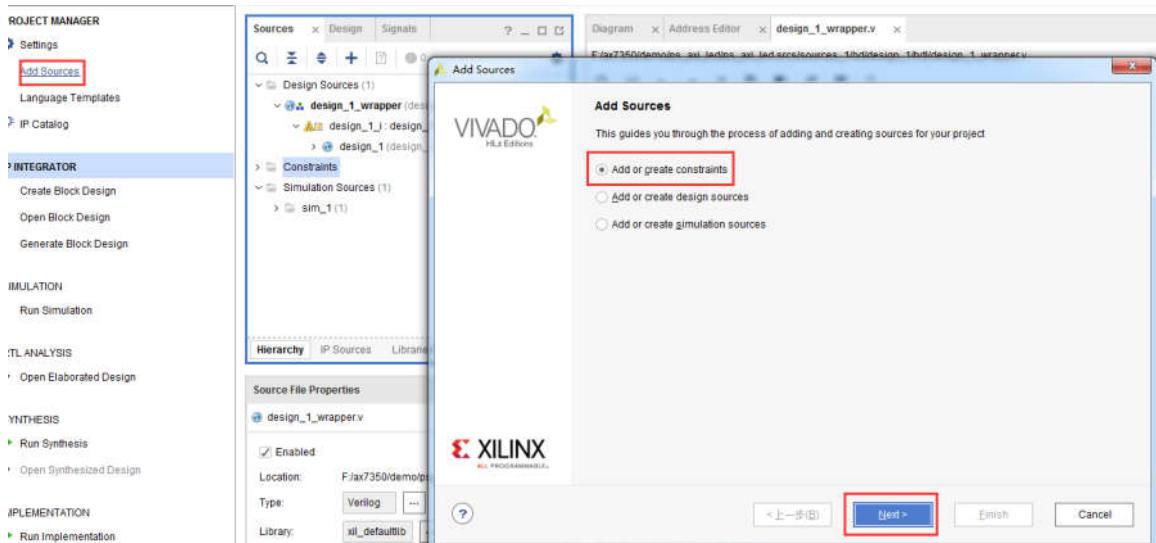
```

28    FIXED_IO_ddr_vrn,
29    FIXED_IO_ddr_vrp,
30    FIXED_IO_mio,
31    FIXED_IO_ps_clk,
32    FIXED_IO_ps_porb,
33    FIXED_IO_ps_rstb,
34    leds_tri_o
35    inout [14:0]DDR_addr;
36    inout [2:0]DDR_ba;
37    inout DDR_cas_n;
38    inout DDR_ck_n;
39    inout DDR_ck_p;
40    inout DDR_cke;
41    inout DDR_csn;
42    inout [3:0]DDR_dm;
43    inout [31:0]DDR_dq;
44    inout [3:0]DDR_dqs_n;
45    inout [3:0]DDR_dqs_p;
46    inout DDR_odt;
47    inout DDR_ras_n;
48    inout DDR_reset_n;
49    inout DDR_we_n;
50    inout FIXED_IO_ddr_vrn;
51    inout FIXED_IO_ddr_vrp;
52    inout [53:0]FIXED_IO_mio;
53    inout FIXED_IO_ps_clk;
54    inout FIXED_IO_ps_porb;
55    inout FIXED_IO_ps_rstb;
56    output [3:0]leds_tri_o;

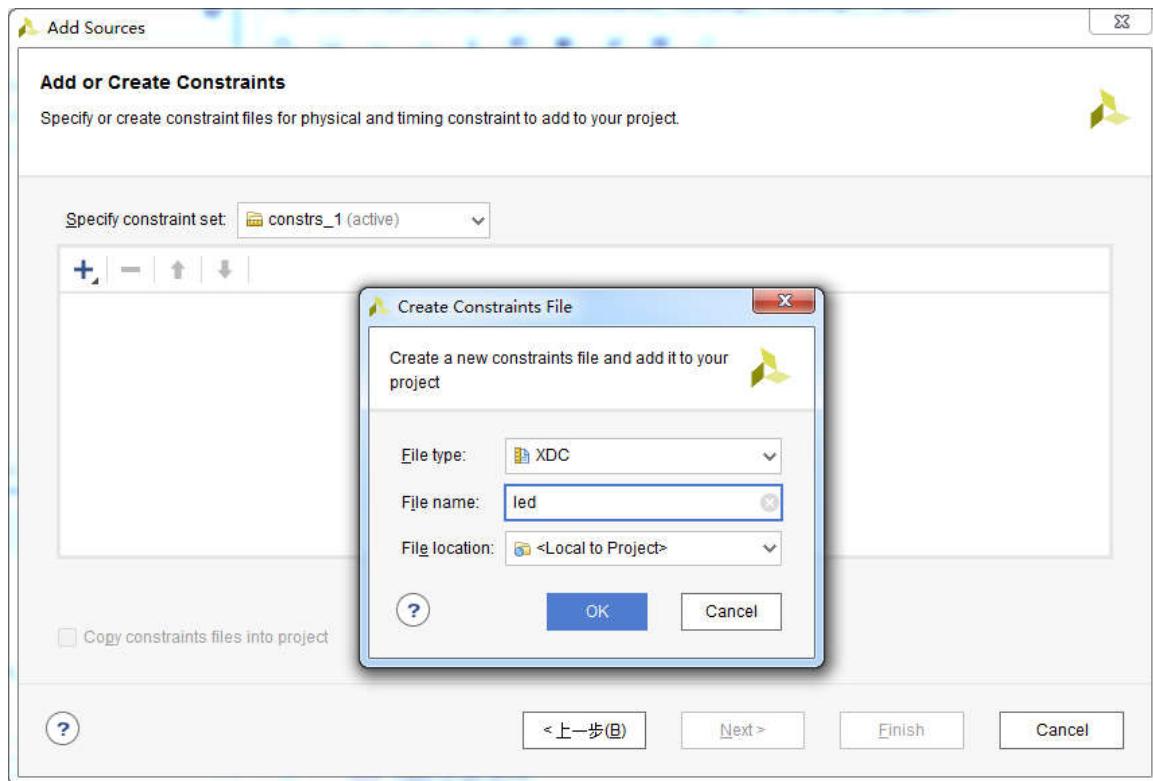
```

10.2 XDC 文件约束 PL 管脚

1) 创建一个新的 xdc 约束文件

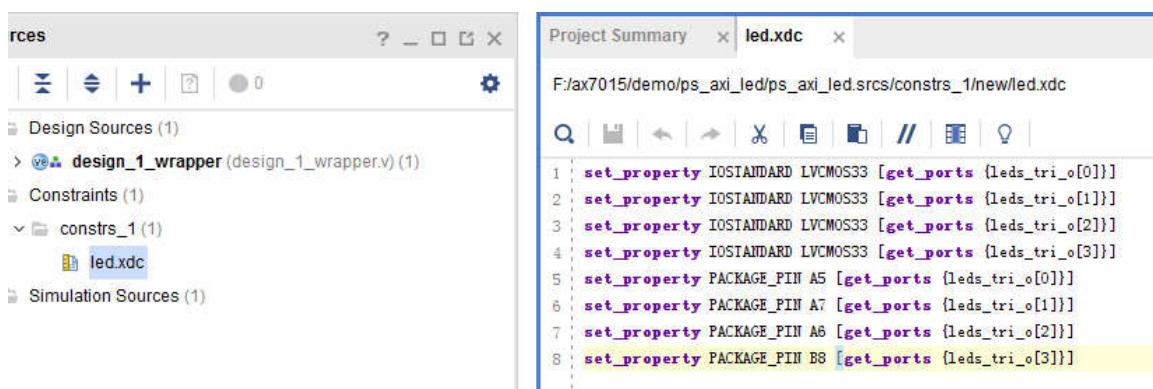


2) 文件名称为 led



3) led.xdc 添加一下内容，端口名称一定要和顶层文件端口一致

```
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[3]}]
set_property PACKAGE_PIN A5 [get_ports {leds_tri_o[0]}]
set_property PACKAGE_PIN A7 [get_ports {leds_tri_o[1]}]
set_property PACKAGE_PIN A6 [get_ports {leds_tri_o[2]}]
set_property PACKAGE_PIN B8 [get_ports {leds_tri_o[3]}]
```

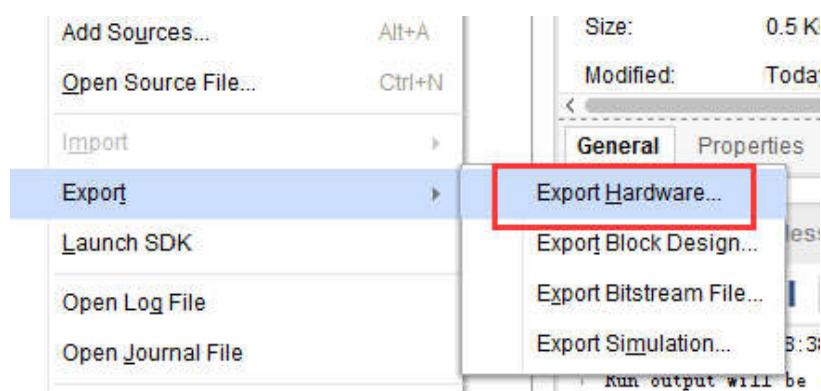


10.3 SDK 程序编写

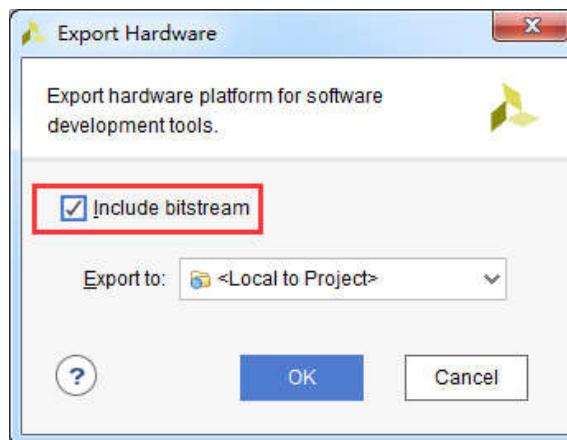
1) 生成 bit 文件



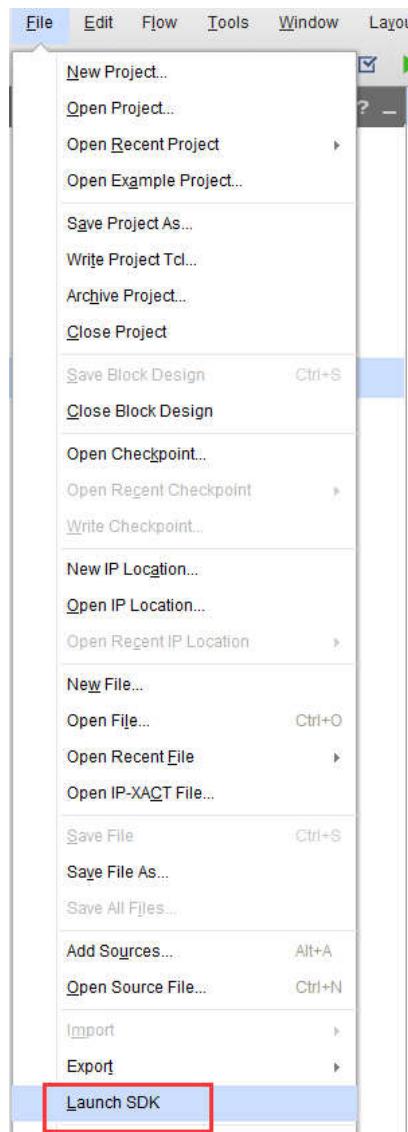
2) 导出硬件



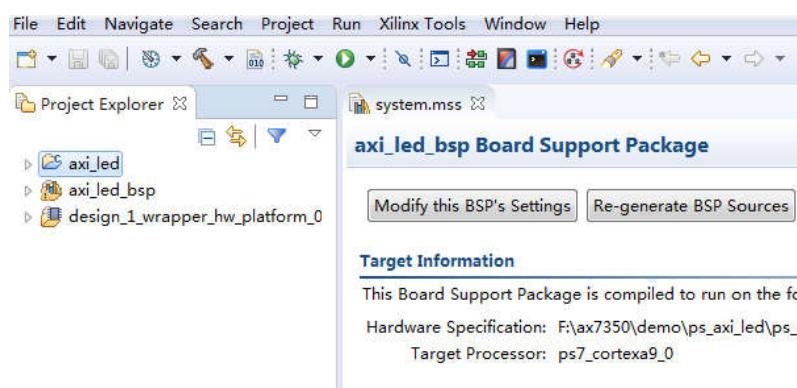
3) 因为要用到 PL，所以选择 “Include bitstream”，点击 “OK”



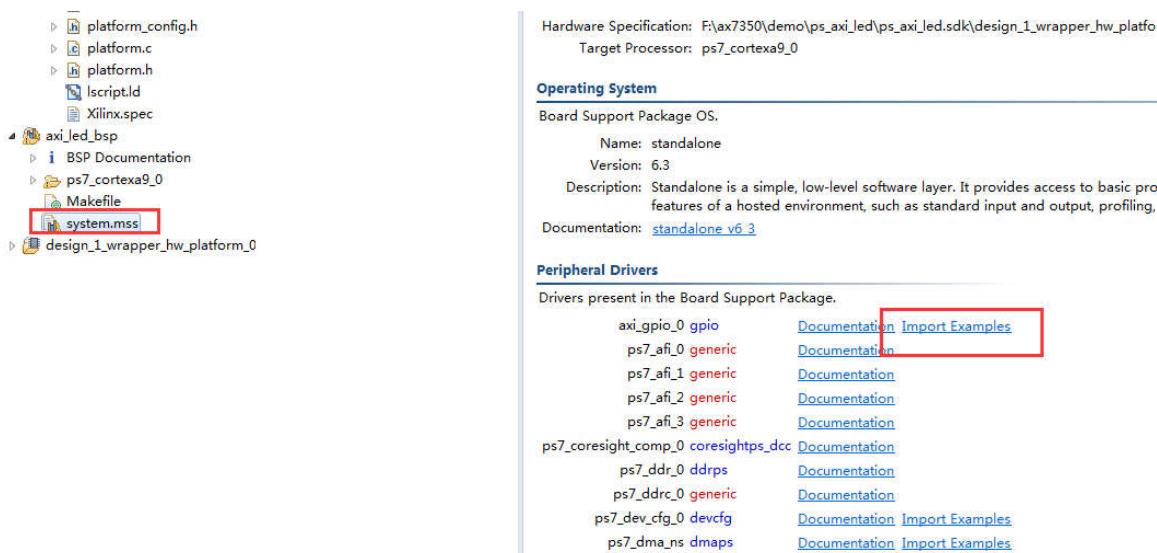
4) 运行 SDK



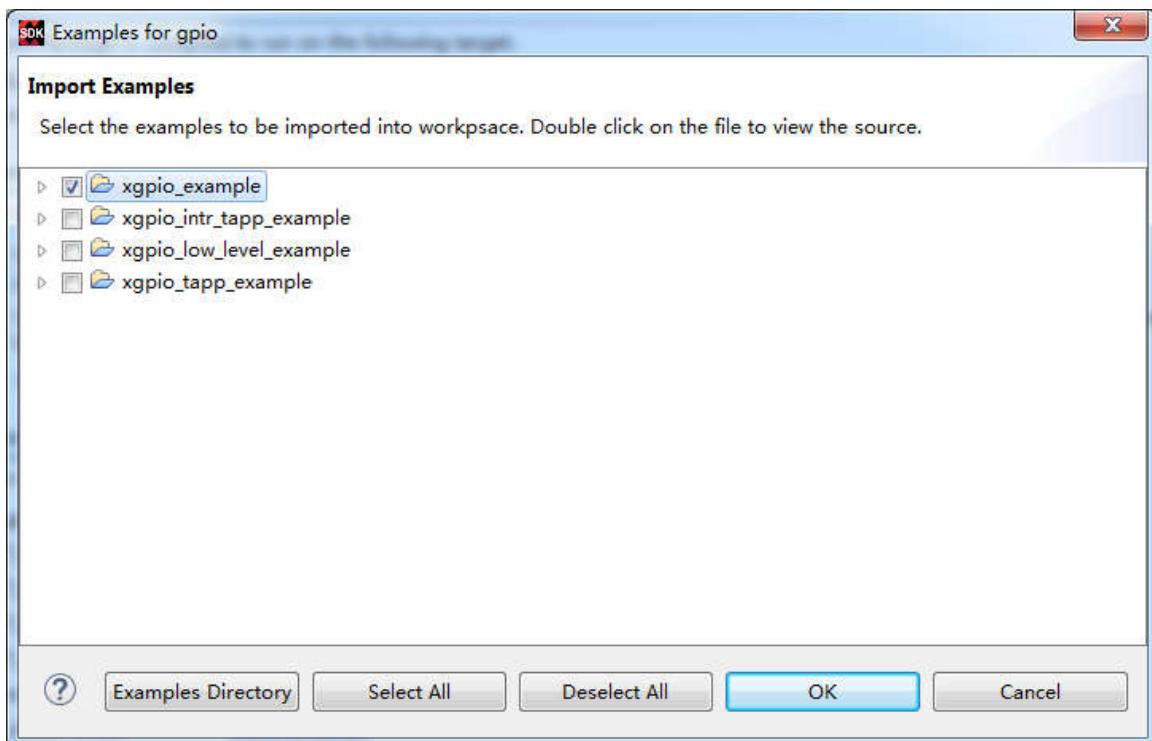
- 5) 创建一个名为“axi_led”的APP，工程模板选择Hello World



- 6) 面对一个不熟悉 AXI GPIO，我们如何控制呢？我们可以尝试一下 SDK 自带的例程
7) 双击“system.mss”，找到“axi_gpio_0”，这里可以点击“Documentation”来看相关文档，
这里就不演示，点击“Import Examples”



- 8) 在弹出的对话框中有多个例程，从名称中可以猜个大概，这里选第一个“xgpio_example”



- 9) 可以看到例程比较简单，短短几行代码，完成了 AXI GPIO 的操作

```

system.mss xgpio_example.c
=====
* @param None
* @return XST_FAILURE to indicate that the GPIO Initialization had
* failed.
*
* @note This function will not return if the test is running.
*****
int main(void)
{
    int Status;
    volatile int Delay;

    /* Initialize the GPIO driver */
    Status = XGpio_Initialize(&Gpio, GPIO_EXAMPLE_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        xil_printf("Gpio Initialization Failed\r\n");
        return XST_FAILURE;
    }

    /* Set the direction for all signals as inputs except the LED output */
    XGpio_SetDataDirection(&Gpio, LED_CHANNEL, ~LED);

    /* Loop forever blinking the LED */
    while (1) {
        /* Set the LED to High */
        XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, LED);

        /* Wait a small amount of time so the LED is visible */
        for (Delay = 0; Delay < LED_DELAY; Delay++);

        /* Clear the LED bit */
        XGpio_DiscreteClear(&Gpio, LED_CHANNEL, LED);

        /* Wait a small amount of time so the LED is visible */
        for (Delay = 0; Delay < LED_DELAY; Delay++);
    }

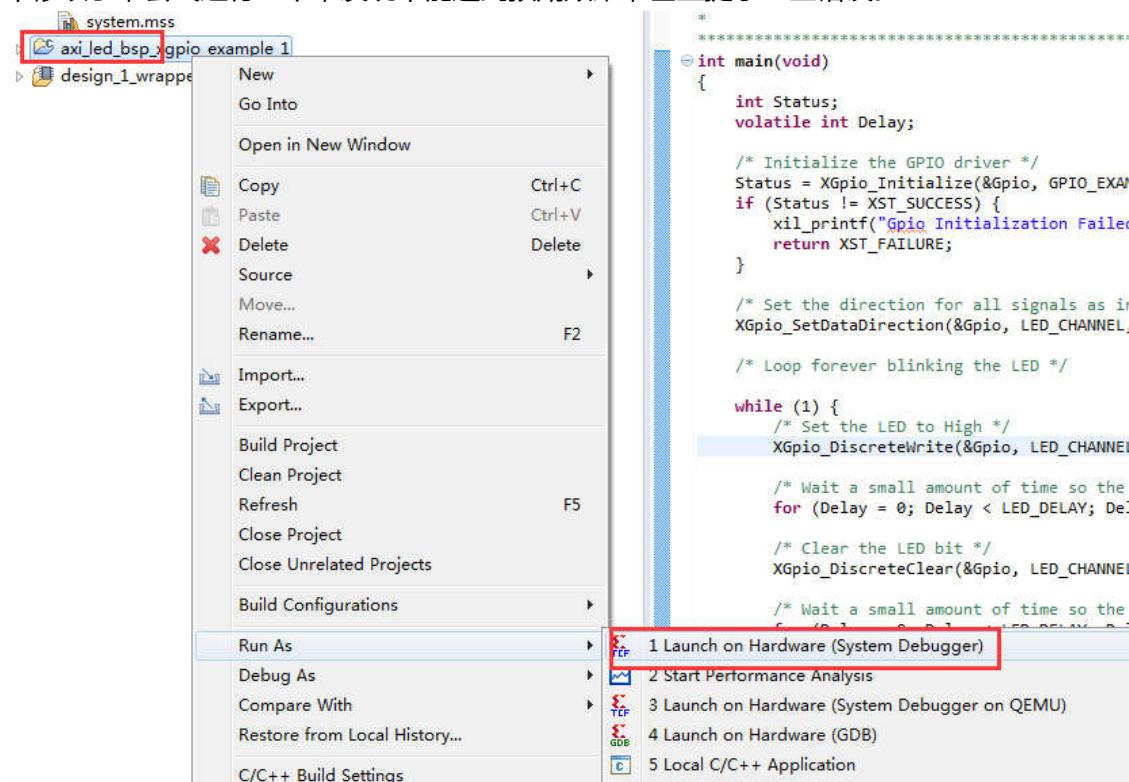
    xil_printf("Successfully ran Gpio Example\r\n");
    return XST_SUCCESS;
}

```

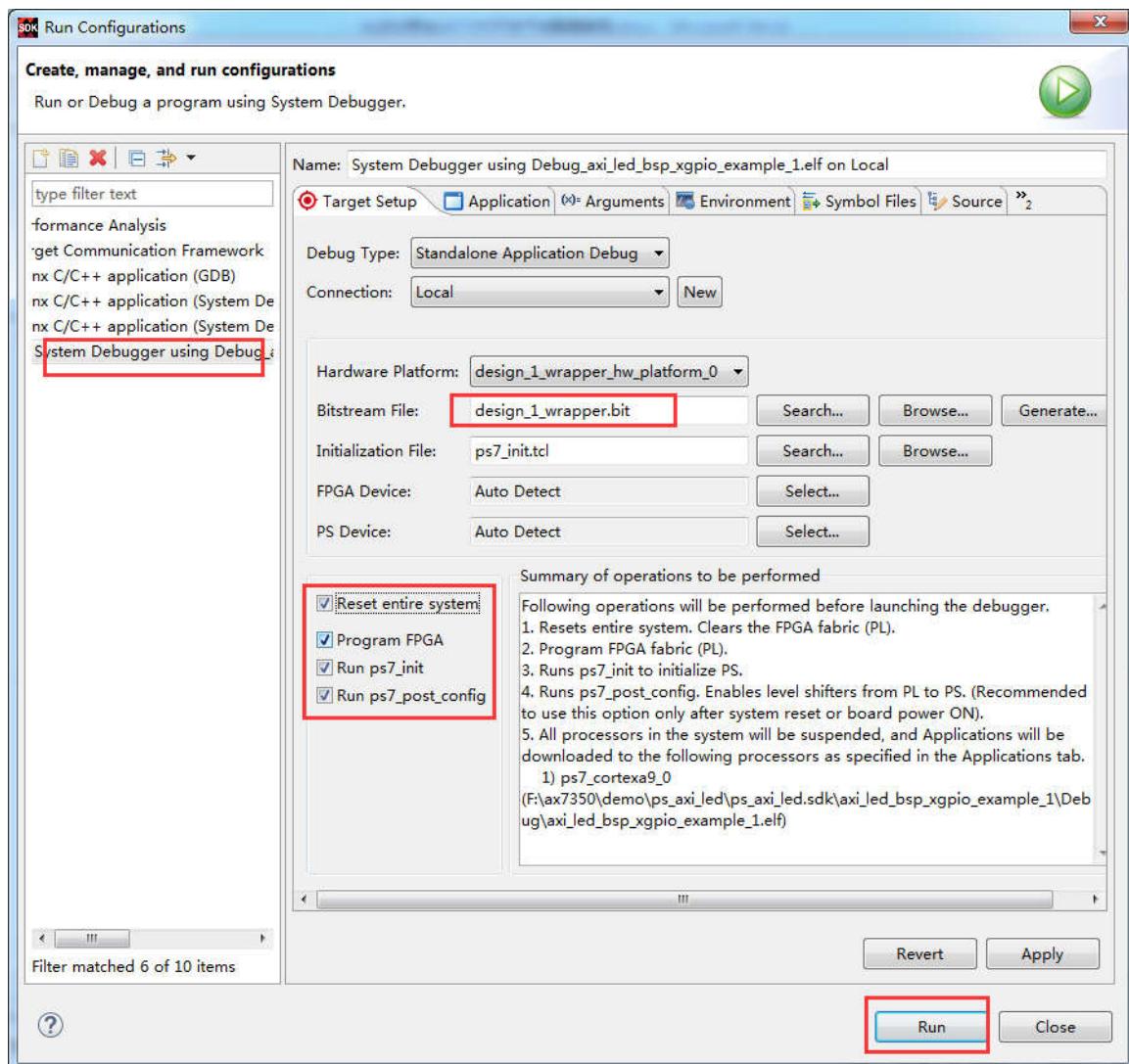
里面用到很多 GPIO 相关的 API 函数，通过文档可以了解详细，也可以选中该函数，按“F3”查看具体定义。如果有了这些信息你还不能理解如何使用 AXI GPIO，说明你需要补充 C 语言基础。

10.4 下载调试

- 虽然 SDK 可以提供一些例程，但有一部分例程是需要自己修改的，这个简单的 LED 例程就不修改了，尝试运行一下，发现不能达到预期效果，甚至提示一些错误。



- 2) 前面的教程已经提到，“Run As” 最好复位系统，有 PL 的设计要“Program FPGA”，如果你的 PL 多次修改，别忘了重新导出硬件。按照下图配置后再次运行，可以看到开发板 LED1 快速闪烁。



- 3) 修改代码让 4 个 LED 灯都闪烁

```
/*-----  
***** Include Files *****  
-----*/  
  
#include "xparameters.h"  
#include "xgpio.h"  
#include "xil_printf.h"  
  
/*----- Constant Definitions -----*/  
  
#define LED 0x0F /* Assumes bit 0 of GPIO is connected to an LED */  
  
/*  
 * The following constants map to the XPAR parameters created in the  
 * xparameters.h file. They are defined here such that a user can easily  
 * change all the needed parameters in one place.  
 */  
#define GPIO_EXAMPLE_DEVICE_ID XPAR_GPIO_0_DEVICE_ID  
  
/*  
 * The following constant is used to wait after an LED is turned on to make  
 * sure the LED is fully illuminated. This value is typically 100ms.  
 */
```

10.5 实验总结

通过实验我们了解到 PS 可以通过 AXI 总线控制 PL，但似乎没有体现出 ZYNQ 的优势，因为对于控制 LED 灯，无论是 ARM 还是 FPGA，都可以轻松完成，但是如果把 LED 换成串口呢，控制 100 路串口通信，8 路以太网等应用，我想还没有哪个 SOC 能完成这种功能，只有 ZYNQ 可以，这就是 ZYNQ 和普通 SOC 的不同之处。

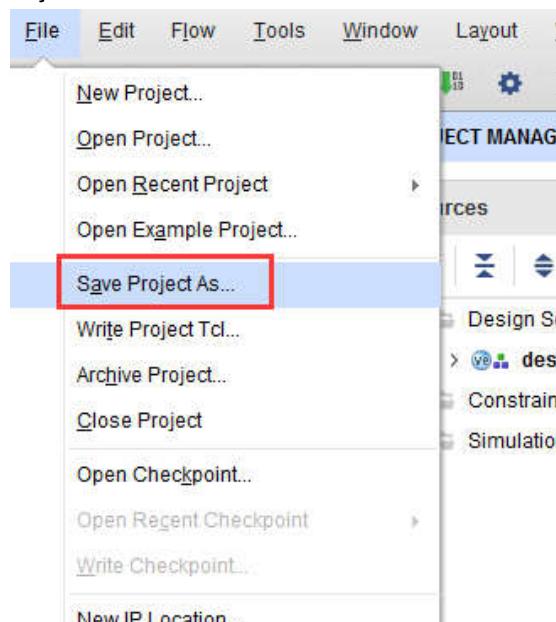
第十一章 PS 定时器中断实验

实验 Vivado 工程为 “ps_timer”。

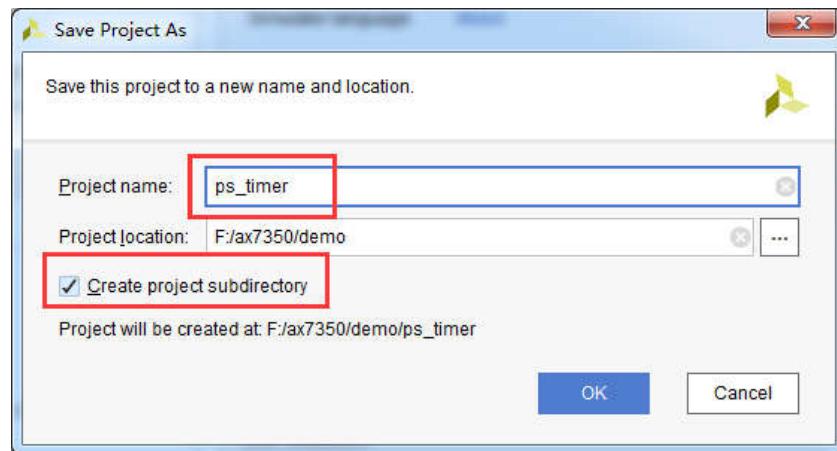
很多 SOC 内部都会有定时器，ZYNQ 的 PS 也有，对于 ZYNQ 内到底有什么外设，这些外设有什么特性，都是开发者必须关心的，因此建议经常阅读 xilinx 文档 UG585。

11.1 Vivado 工程建立

- 1) 反复建立 Vivado 工程 其中有大量的重复性工作 最简单的解决方法就是把工程复制一份，再修改一下，打开工程 “ps_hello”
- 2) 点击菜单 “File ->Save Project As...”

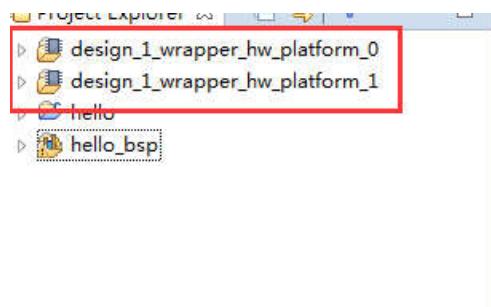


- 3) 在弹出的对话框中填写新的工程名 “ps_timer”，选择创建工程子目录，PS 里的定时器，因为不需要管脚输出，就不用配置管脚了

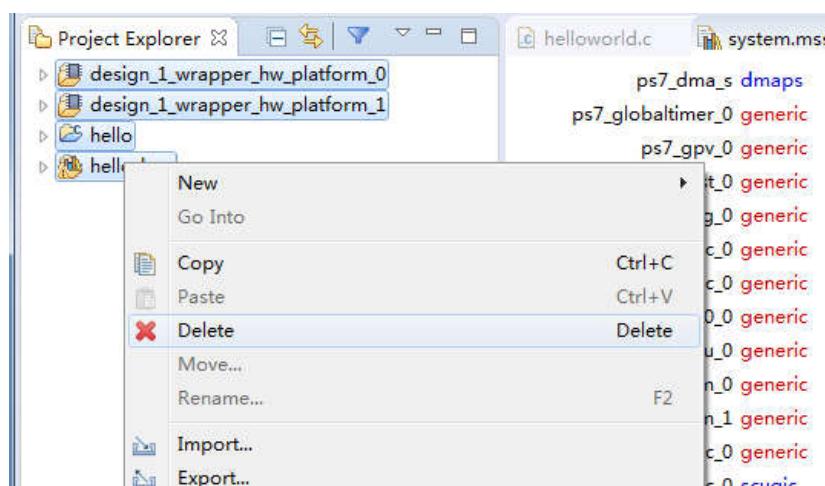


11.2 SDK 程序编写

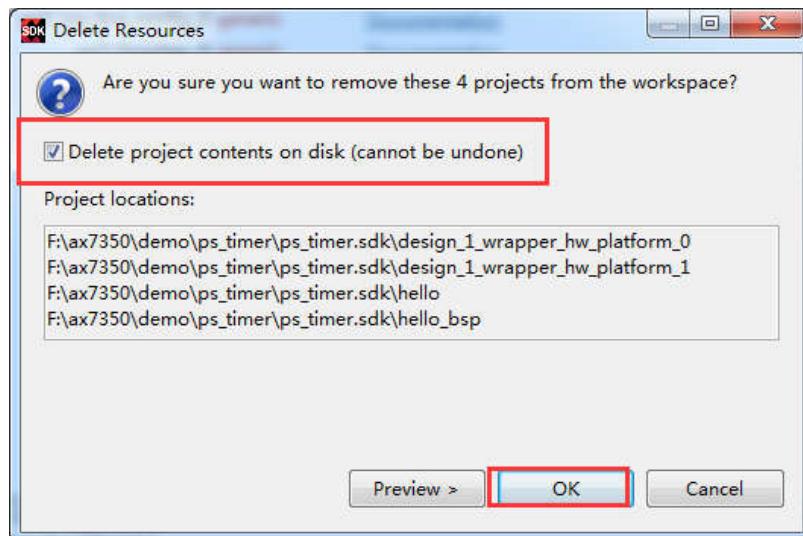
1) 运行 SDK，可以看到，和前面的例程不同，这里又多出了一个硬件平台信息文件夹



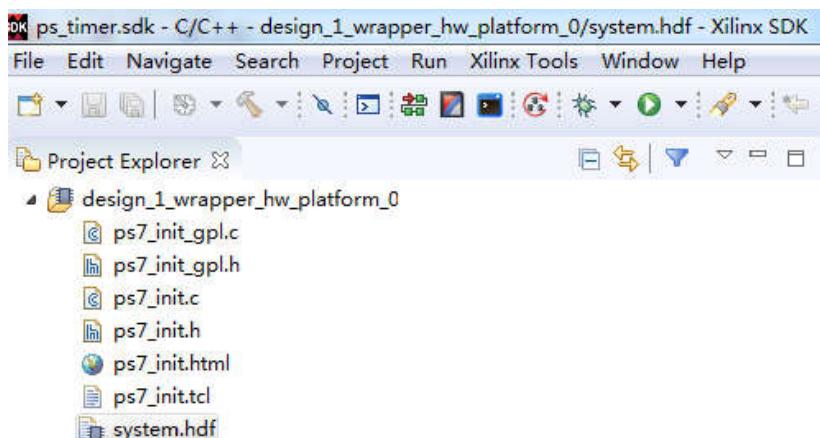
2) 使用别人的 SDK 工程时也会有类似的现象出现，这里我们都给删除



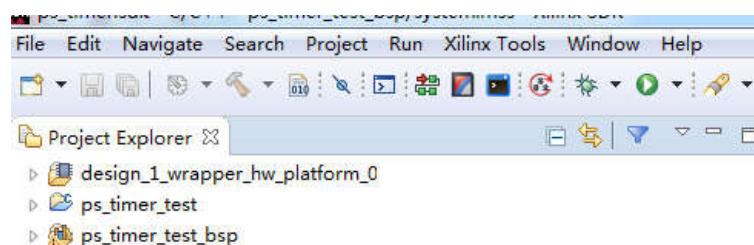
3) 文件也删除



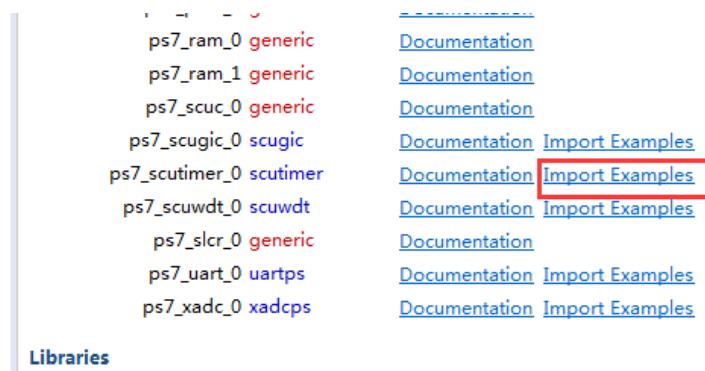
- 4) 在 Vivado 里重新运行 SDK , 可以看到又有一个新的硬件平台信息



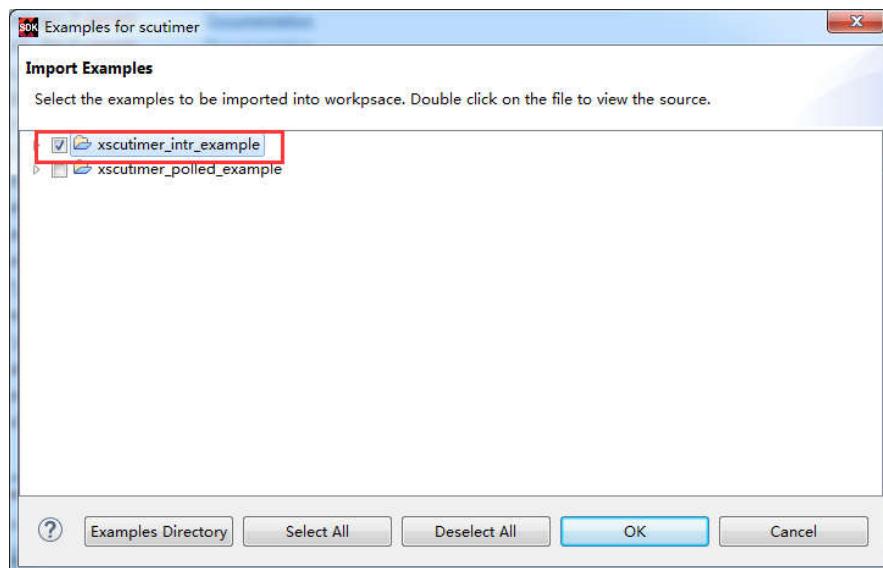
- 5) 重新建立一个工程 , 名字就叫 “ps_timer_test” , 模板还是 Hello World



- 6) 现在又到了写代码的时候了 , 又有了无从下手的感觉 , 不知道定时器怎么用 , 也不知道中断怎么用 , 还是用老方法 , 来看看例子



- 7) 非常幸运，有一个定时器中断的例子，怎么就知道这个例子就是中断的例子呢？是通过“intr”猜测的，所以，基本功很重要，不然你练找例程都不会。



下面就是阅读代码，然后修改代码了，当然，可能一下不能完全理解这些代码，只能在以后的应用中去反复练习

- 8) 本实验设计一个 1 秒定时器中断一次，然后打印出信息，30 秒后结束，首先修改计数器最大值，修改为 CPU 频率的一半，也就是计数器的时钟频率值，这样就会 1 秒中断一次

```
***** Include Files *****/
#include "xparameters.h"
#include "xscutimer.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_printf.h"

***** Constant Definitions *****/
/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are only defined here such that a user can easily
 * change all the needed parameters in one place.
 */
#ifndef TESTAPP_GEN
#define TIMER_DEVICE_ID      XPAR_XSCUTIMER_0_DEVICE_ID
#define INTC_DEVICE_ID        XPAR_SCUGIC_SINGLE_DEVICE_ID
#define TIMER_IRPT_INTR      XPAR_SCUTIMER_INTR
#endif
#define TIMER_LOAD_VALUE     (XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ/2 - 1)

***** Type Definitions *****/
***** Macros (Inline Functions) Definitions *****/
***** Function Prototypes *****
```

9) 修改计数次数 3 改为 30

```
LastTimerExpired = TimerExpired;
/*
 * If it has expired a number of times, then stop the timer
 * counter and stop this example.
 */
if (TimerExpired == 30) {
    XScuTimer_Stop(TimerInstancePtr);
    break;
}
/*
 * Disable and disconnect the interrupt system.
 */
TimerDisableIntrSystem(IntcInstancePtr, TimerIntrId);
```

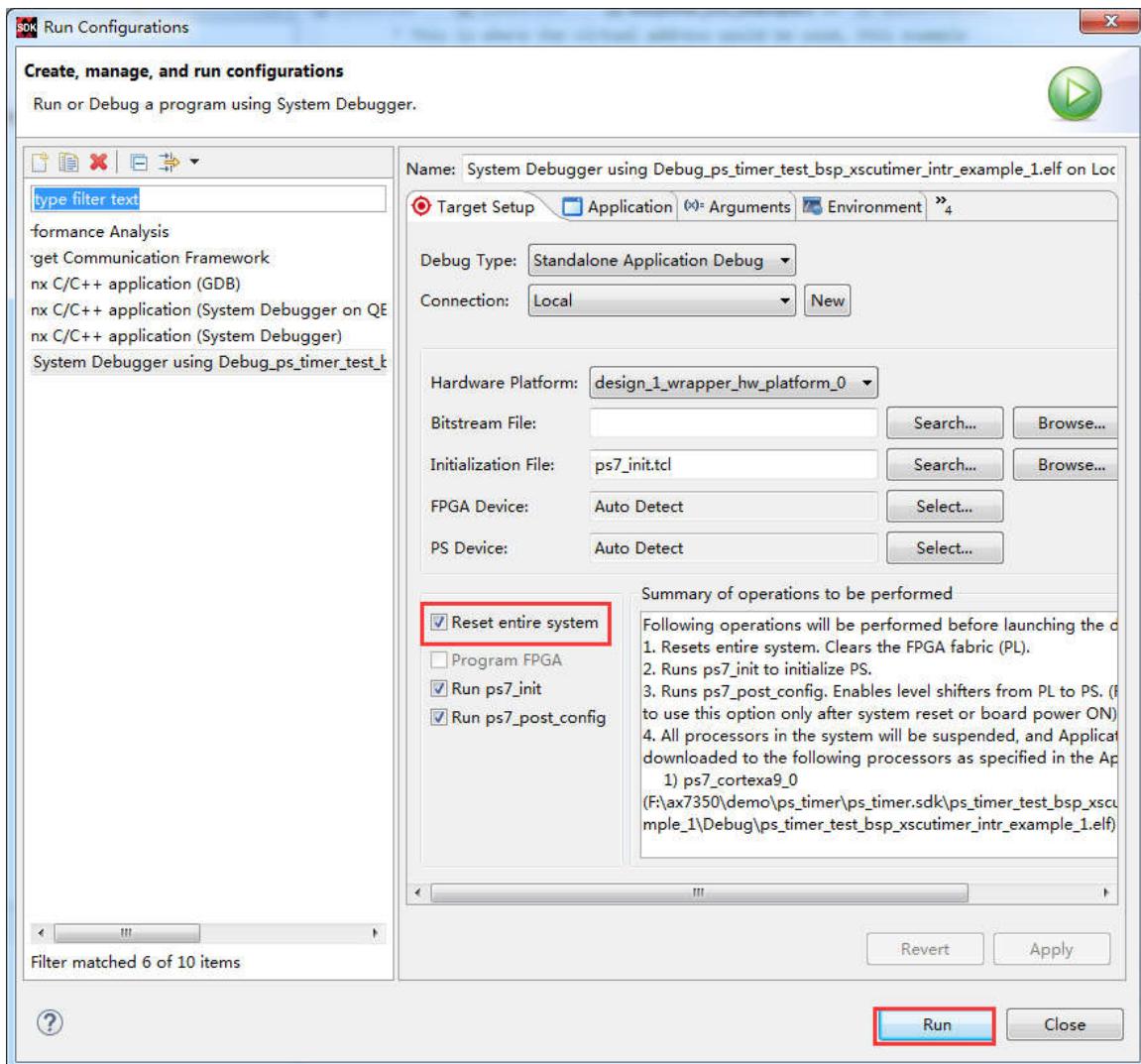
10) 添加打印信息

```
*****
@ static void TimerIntrHandler(void *CallBackRef)
{
    XScuTimer *TimerInstancePtr = (XScuTimer *) CallBackRef;

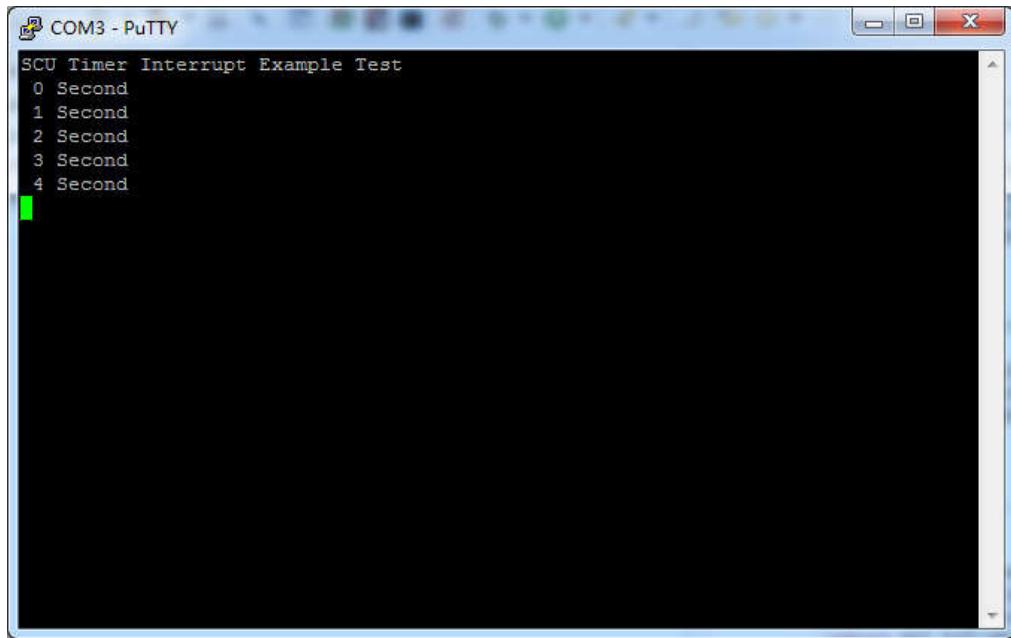
    /*
     * Check if the timer counter has expired, checking is not necessary
     * since that's the reason this function is executed, this just shows
     * how the callback reference can be used as a pointer to the instance
     * of the timer counter that expired, increment a shared variable so
     * the main thread of execution can see the timer expired.
     */
    if (XScuTimer_IsExpired(TimerInstancePtr)) {
        XScuTimer_ClearInterruptStatus(TimerInstancePtr);
        printf("%d Second\n\r",TimerExpired);
        TimerExpired++;
        if (TimerExpired == 30) {
            XScuTimer_DisableAutoReload(TimerInstancePtr);
        }
    }
}
```

11.3 下载调试

- 1) 打开 PuTTY 串口终端
- 2) 下载调试程序的方法前面教程已经讲解，不再复述



- 3) 和我们预期一样，串口没秒会输出一句信息



11.4 实验总结

实验中通过简单的修改 SDK 的例程 , 就完成了定时器 , 中断的应用 , 看似简单的操作 , 可蕴含了丰富的知识 , 我们需要非常了解定时器的原理、中断的原理 , 这些基本知识是学习好 ZYNQ 的必要条件。

第十二章 PL 按键中断实验

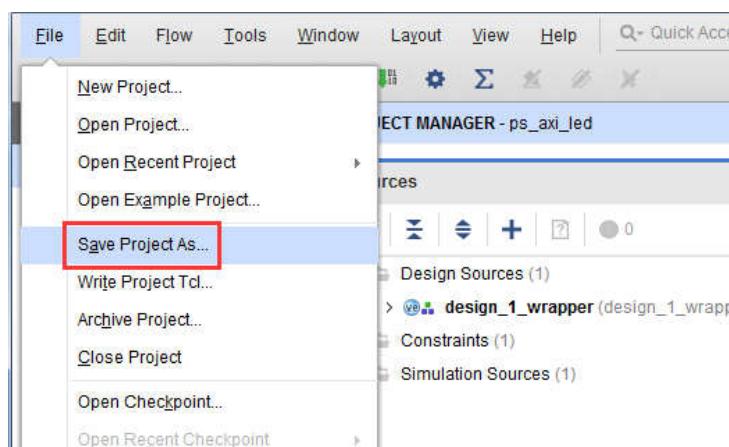
实验 Vivado 工程为 “ps_axi_key”。

前面的定时器中断实验的中断属于 PS 内部的中断，本实验中断来自 PL，PS 最大可以接收 16 个来自 PL 的中断信号，都是上升沿或高电平触发。本实验用按键中断来控制 LED。

Source	Interrupt Name	IRQ ID#	Status Bits (mpcore Registers)	Required Type	PS-PL Signal Name	I/O
PL	PL [2:0]	63:61	spi_status_0[31:29]	Rising edge/ High level	IRQF2P[2:0]	Input
	PL [7:3]	68:64	spi_status_1[4:0]	Rising edge/ High level	IRQF2P[7:3]	Input
Timer	TTC 1	71:69	spi_status_1[7:5]	High level	~	~
DMAC	DMAC[7:4]	75:72	spi_status_1[11:8]	High level	IRQP2F[27:24]	Output
IOP	USB 1	76	spi_status_1[12]	High level	IRQP2F[7]	Output
	Ethernet 1	77	spi_status_1[13]	High level	IRQP2F[6]	Output
	Ethernet 1 Wake-up	78	spi_status_1[14]	Rising edge	IRQP2F[5]	Output
	SDIO 1	79	spi_status_1[15]	High level	IRQP2F[4]	Output
	I2C 1	80	spi_status_1[16]	High level	IRQP2F[3]	Output
	SPI 1	81	spi_status_1[17]	High level	IRQP2F[2]	Output
	UART 1	82	spi_status_1[18]	High level	IRQP2F[1]	Output
	CAN 1	83	spi_status_1[19]	High level	IRQP2F[0]	Output
PL	PL [15:8]	91:84	spi_status_1[27:20]	Rising edge/ High level	IRQF2P[15:8]	Input
SCU	Parity	92	spi_status_1[28]	Rising edge	~	~
Reserved	~	95:93	spi_status_1[31:29]	~	~	~

12.1 Vivado 工程建立

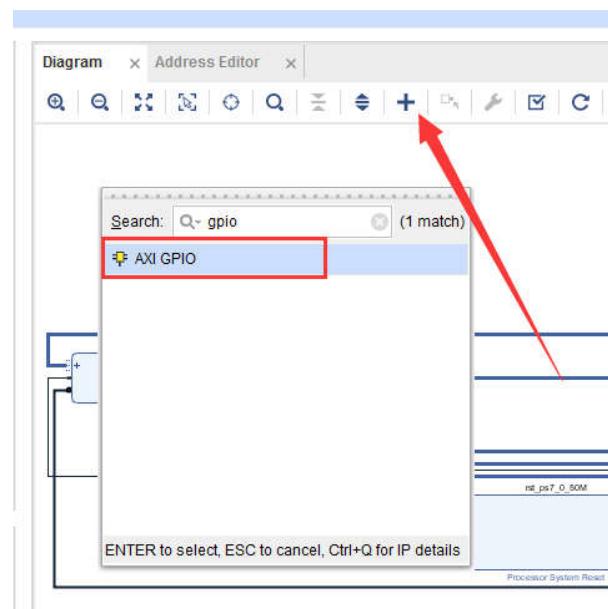
- 1) 本实验所用的 Vivado 工程只需要在 “ps_axi_led” 这个工程上添加用于按键输入的 AXI GPIO 就可以，点击菜单 “File -> Save Project As...”



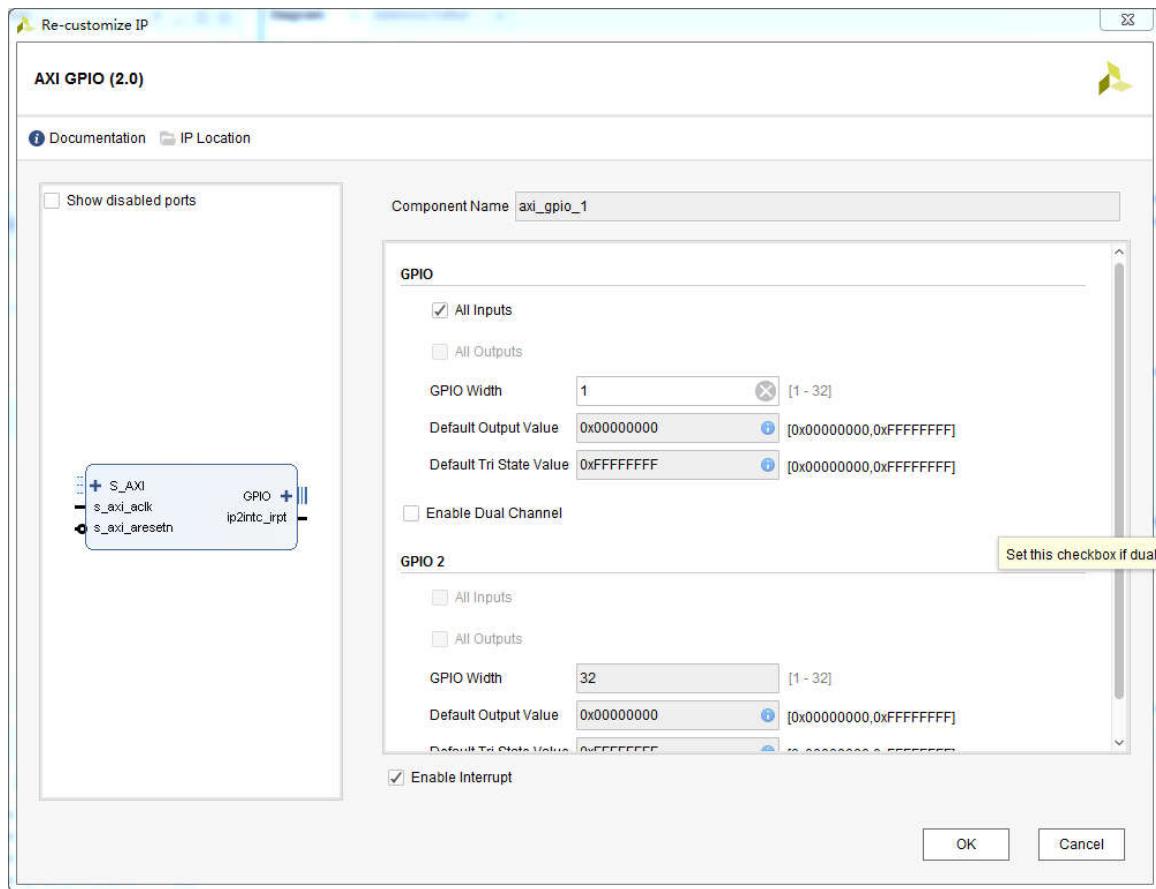
- 2) 新的工程名为 “ps_axi_key”



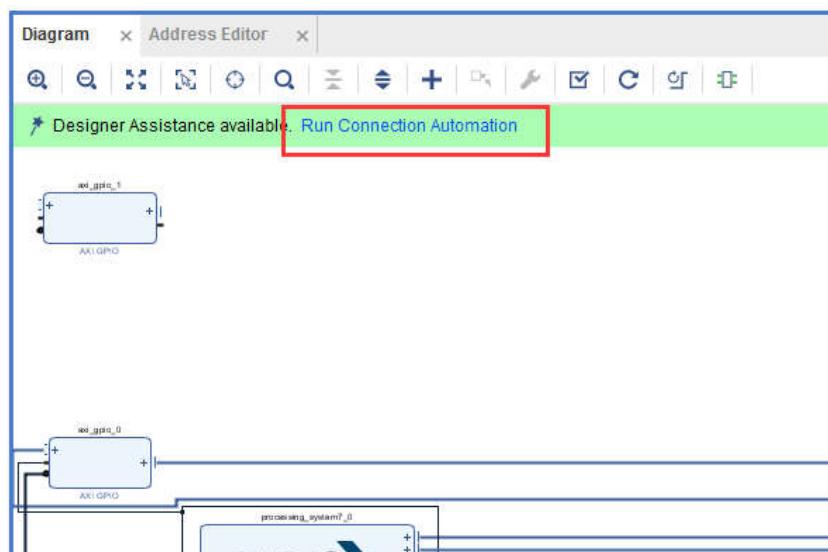
3) 添加一个 AXI GPIO



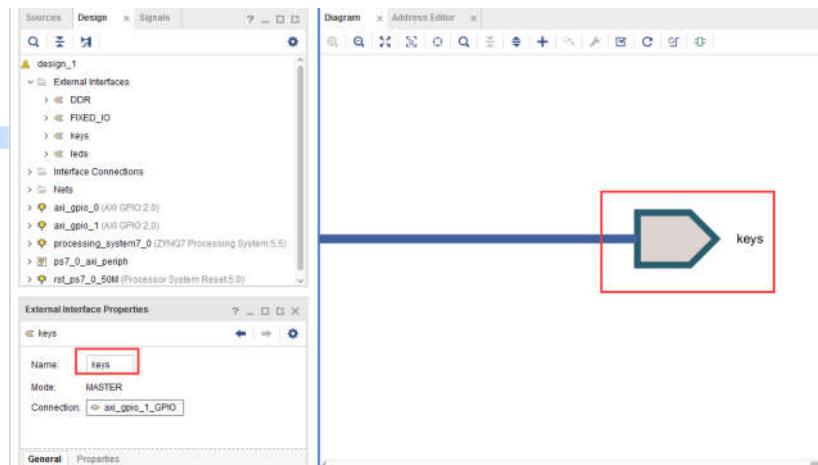
4) 配置 GPIO 参数，都为输入，宽度为 1，使能中断



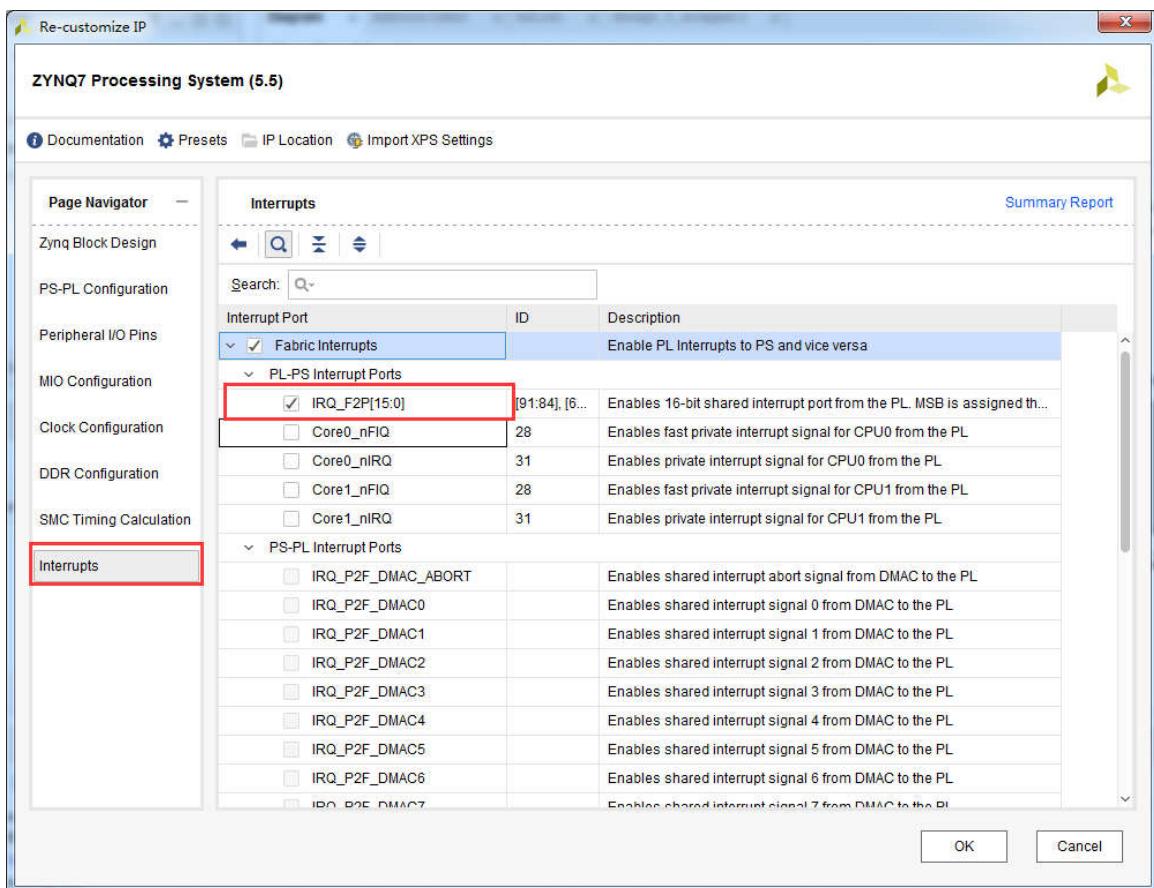
5) 使用自动连接



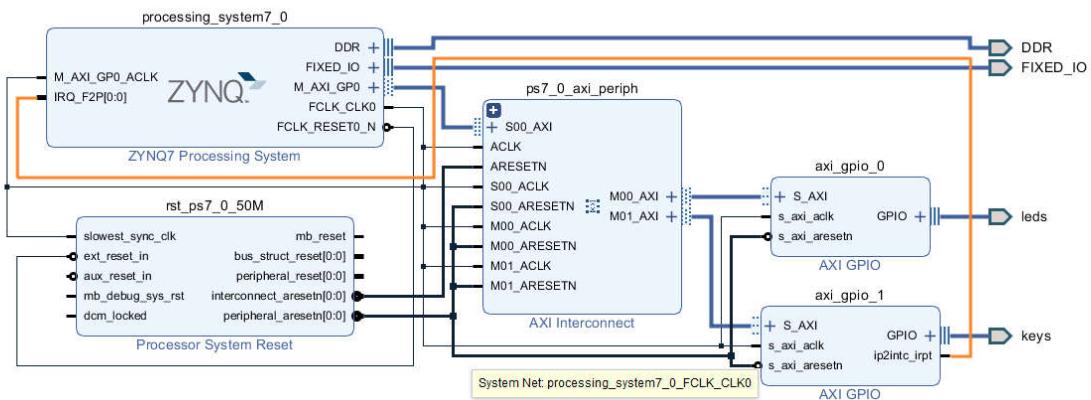
6) 再把端口名称改为 keys



7) 配置 ZYNQ 处理器的中断，勾选 IRQ_F2P

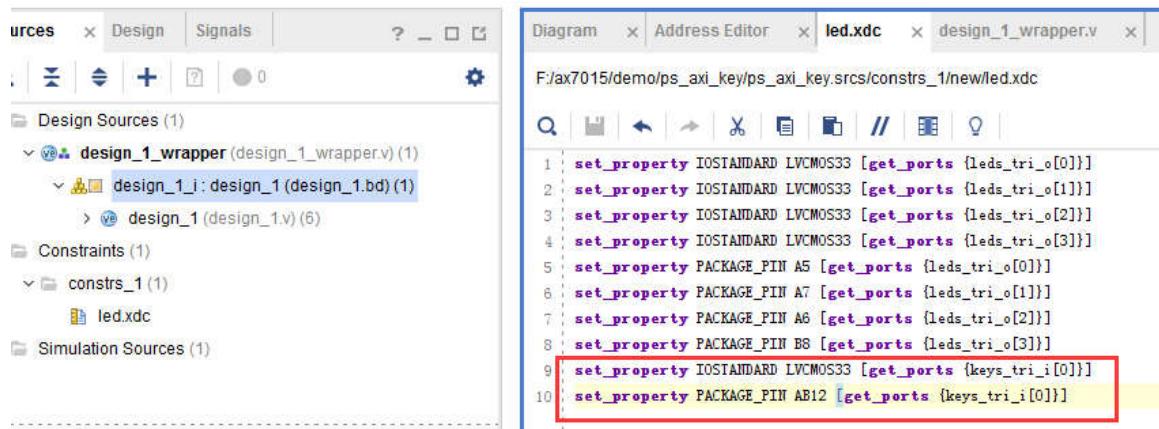


8) 连接 ip2intc_irpt 到 IRQ_F2Q



9) 修改 xdc 约束文件

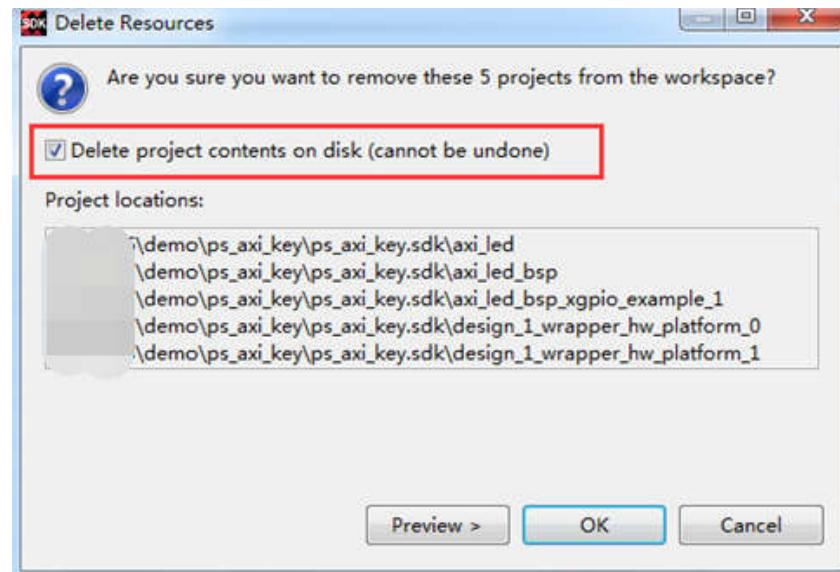
```
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[3]}]
set_property PACKAGE_PIN A5 [get_ports {leds_tri_o[0]}]
set_property PACKAGE_PIN A7 [get_ports {leds_tri_o[1]}]
set_property PACKAGE_PIN A6 [get_ports {leds_tri_o[2]}]
set_property PACKAGE_PIN B8 [get_ports {leds_tri_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {keys_tri_i[0]}]
set_property PACKAGE_PIN AB12 [get_ports {keys_tri_i[0]}]
```



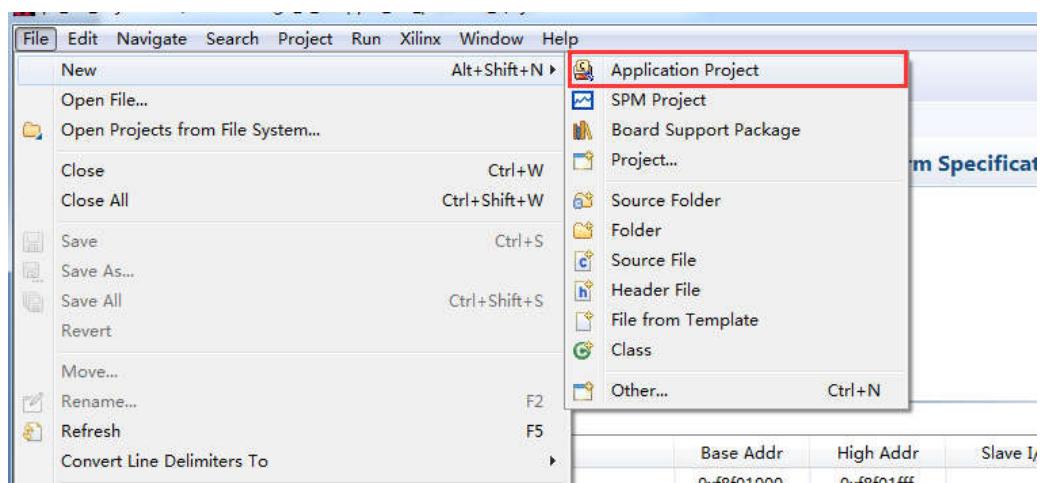
10) 保存设计，编译生成 bit 文件

12.2 下载调试

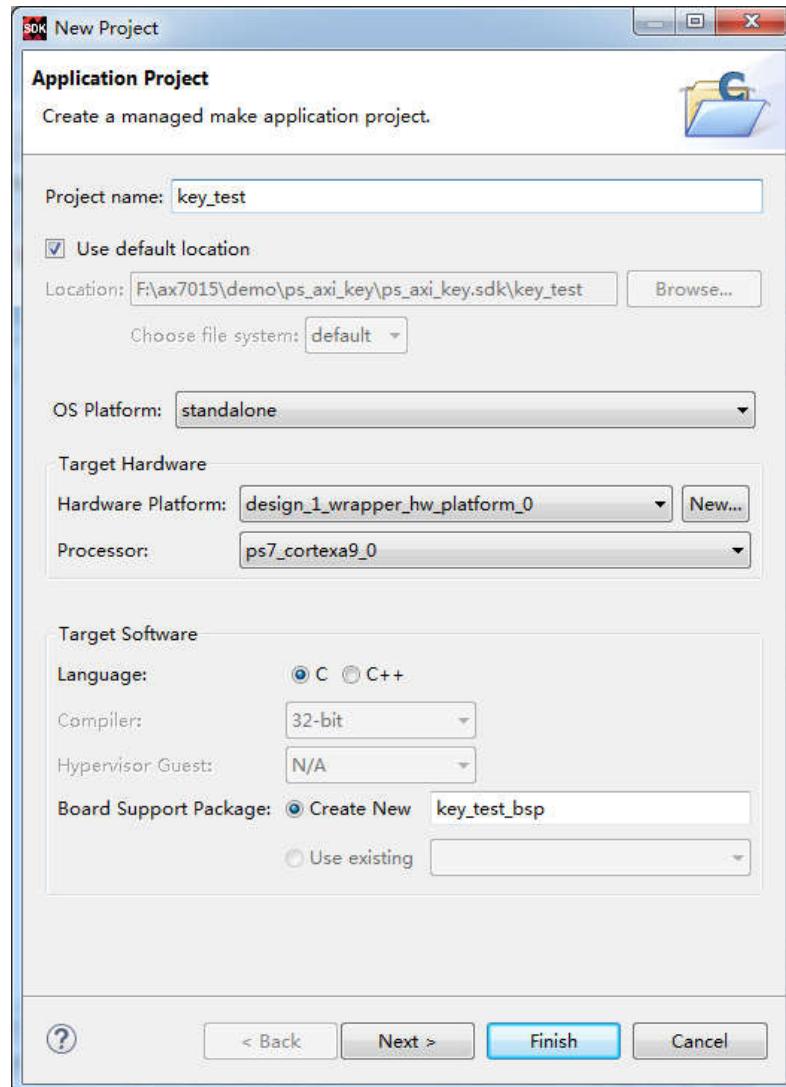
- 运行 SDK，由于是从其他工程复制而来，SDK 下有很多我们不需要的文件，全部删除，然后关闭 SDK 重新运行。



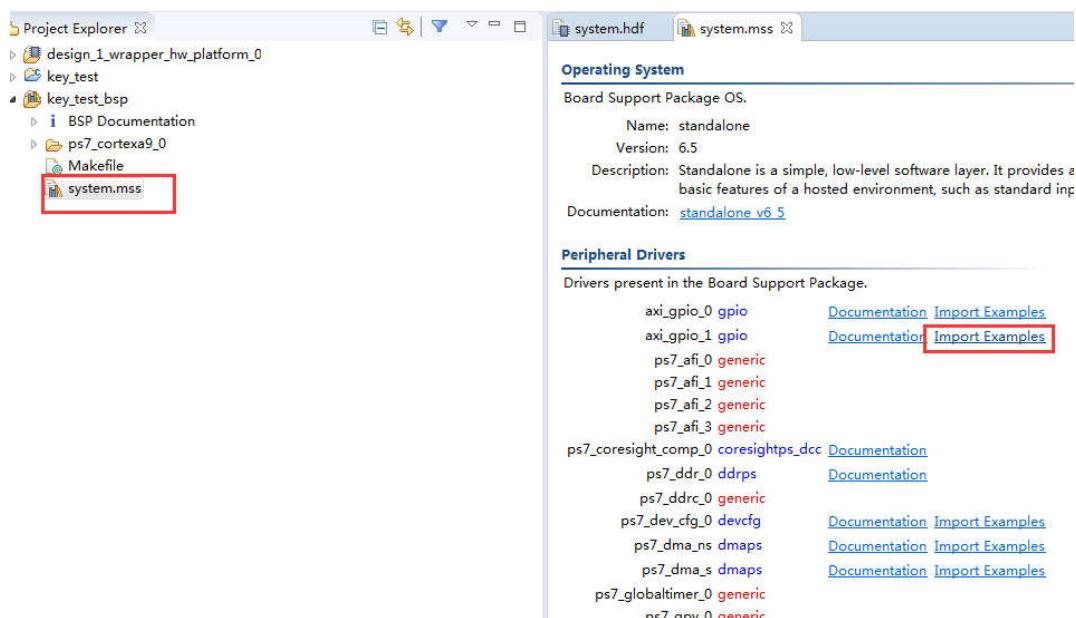
2) 新建一个 APP



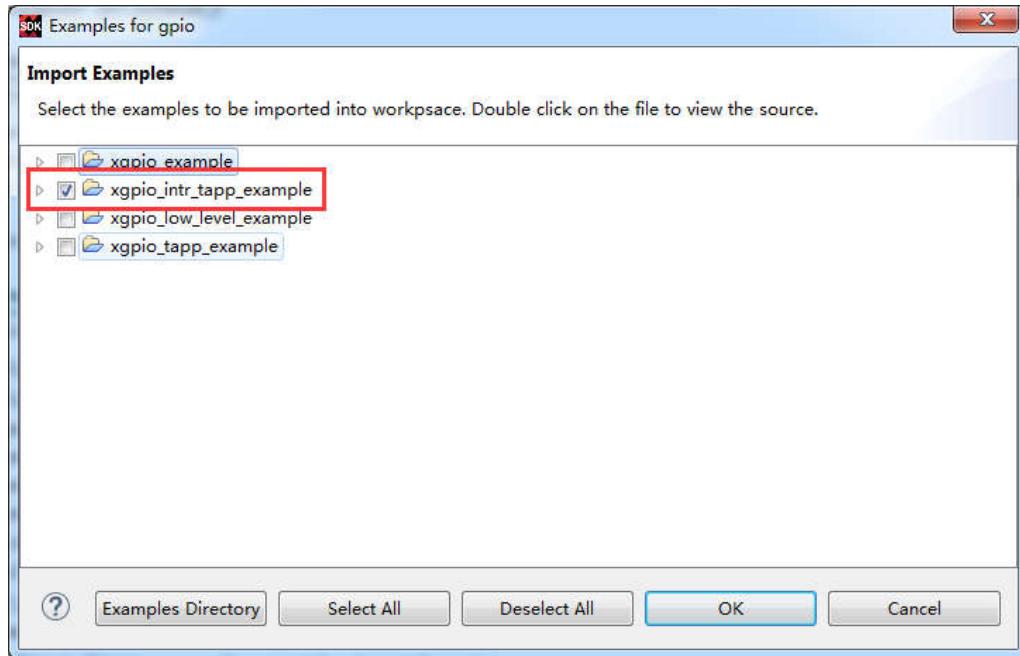
3) 设置 Project name 为 "key_test"



4) 和前面的教程一样，在不熟悉 SDK 程序编写的情况下，我们尽量使用 SDK 自带例程来修改



5) 选择 “xgpio_intr_tapp_example”



6) 导入例程以后有未定义的错误，我们需要修改部分代码

```

design_1_wrapper_hw_platform_0
key_test
key_test_bsp
  BSP Documentation
  ps7_cortexa9_0
    Makefile
    system.mss
key_test_bsp_xgpio_intr_tapp_example_1

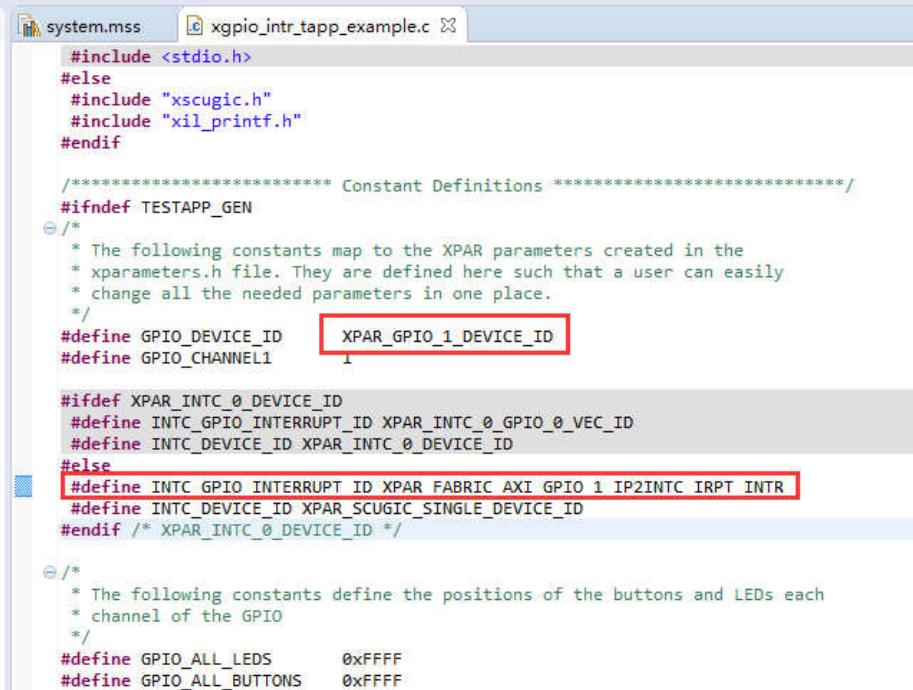
#include <stdio.h>
#ifndef TESTAPP_GEN
#define GPIO_DEVICE_ID      XPAR_GPIO_0_DEVICE_ID
#define GPIO_CHANNEL1        1
#endif
#define XPAR_INTC_0_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID XPAR_INTC_0_GPIO_0_VEC_ID
#define INTC_DEVICE_ID XPAR_INTC_0_DEVICE_ID
#define XPAR_INTC_0_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
#endif /* XPAR_INTC_0_DEVICE_ID */

#define GPIO_ALL_LEDS      0xFFFF
#define GPIO_ALL_BUTTONS   0xFFFF

#define BUTTON_CHANNEL     1 /* Channel 1 of the GPIO Device */
#define LED CHANNEL        2 /* Channel 2 of the GPIO Device */

```

7) 按下图修改 GPIO 和中断号的宏定义

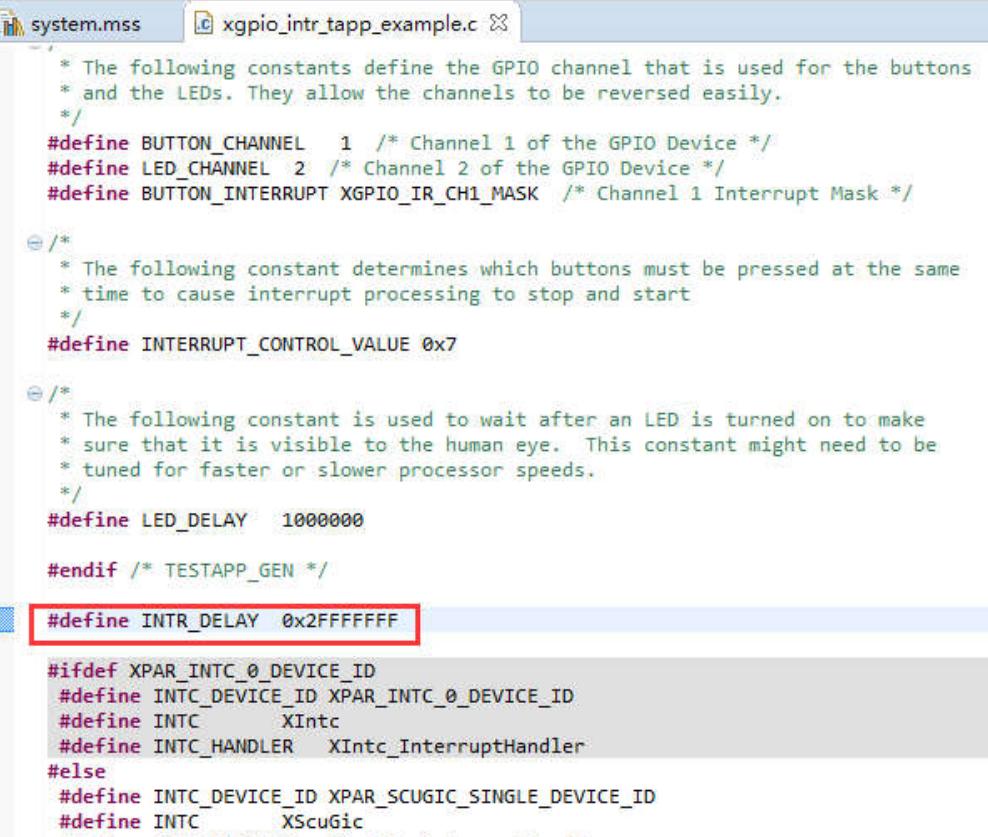


```

system.mss xgpio_intr_tapp_example.c
#include <stdio.h>
#ifndef TESTAPP_GEN
#define XPAR_GPIO_1_DEVICE_ID
#define GPIO_CHANNEL1 1
#endif
/* The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are defined here such that a user can easily
 * change all the needed parameters in one place.
 */
#define GPIO_DEVICE_ID XPAR_GPIO_1_DEVICE_ID
#define GPIO_CHANNEL1 1
#define XPAR_INTC_0_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID XPAR_INTC_0_GPIO_0_VEC_ID
#define INTC_DEVICE_ID XPAR_INTC_0_DEVICE_ID
#ifndef XPAR_INTC_0_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_1_IP2INTC_IRPT_INTR
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
#endif /* XPAR_INTC_0_DEVICE_ID */
/* The following constants define the positions of the buttons and LEDs each
 * channel of the GPIO
 */
#define GPIO_ALL_LEDS 0xFFFF
#define GPIO_ALL_BUTTONS 0xFFFF

```

8) 修改测试延时时间，让我们有足够的时间去按按键



```

system.mss xgpio_intr_tapp_example.c
/* The following constants define the GPIO channel that is used for the buttons
 * and the LEDs. They allow the channels to be reversed easily.
 */
#define BUTTON_CHANNEL 1 /* Channel 1 of the GPIO Device */
#define LED_CHANNEL 2 /* Channel 2 of the GPIO Device */
#define BUTTON_INTERRUPT XGPIO_IR_CH1_MASK /* Channel 1 Interrupt Mask */

/* The following constant determines which buttons must be pressed at the same
 * time to cause interrupt processing to stop and start
 */
#define INTERRUPT_CONTROL_VALUE 0x7

/* The following constant is used to wait after an LED is turned on to make
 * sure that it is visible to the human eye. This constant might need to be
 * tuned for faster or slower processor speeds.
 */
#define LED_DELAY 1000000

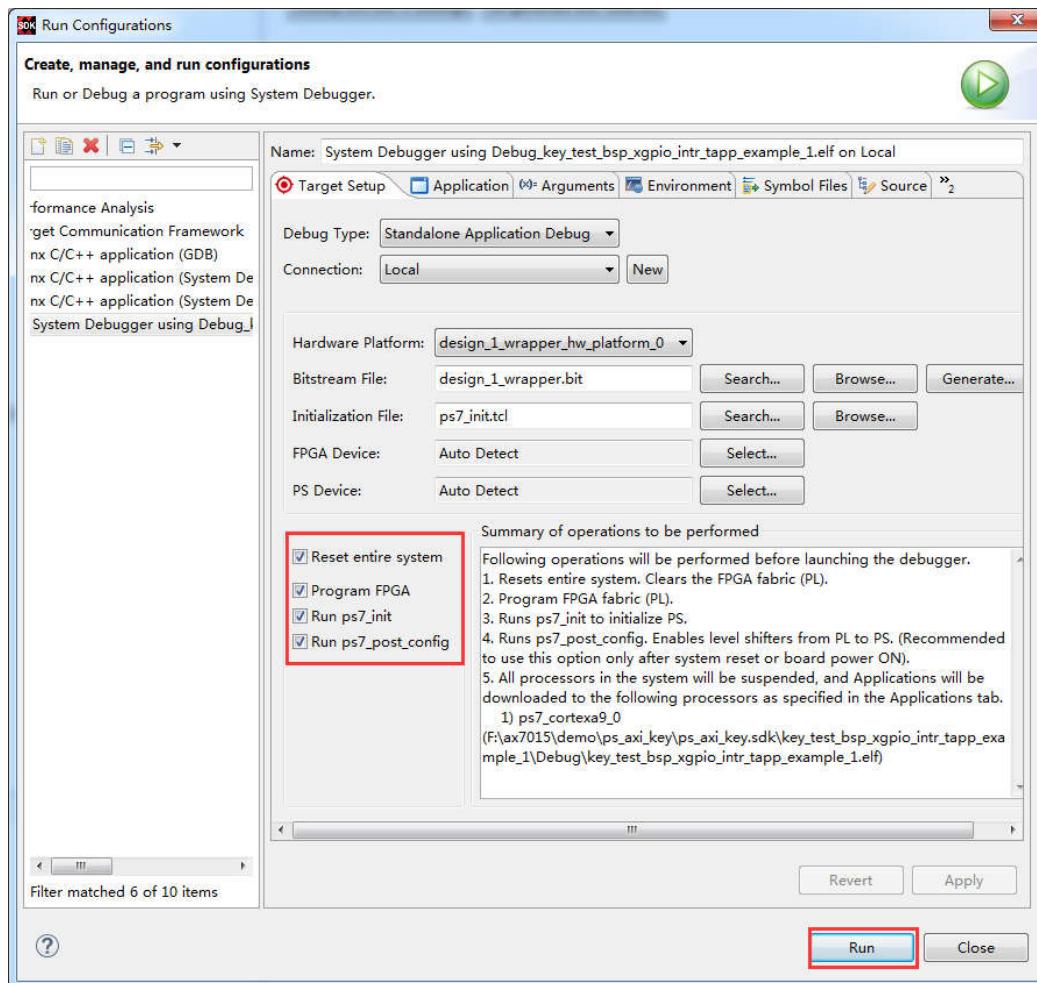
#endif /* TESTAPP_GEN */

#define INTR_DELAY 0x2FFFFFFF

#endif XPAR_INTC_0_DEVICE_ID
#define INTC_DEVICE_ID XPAR_INTC_0_DEVICE_ID
#define INTC_XIntc
#define INTC_HANDLER XIntc_InterruptHandler
#else
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
#define INTC_XScuGic
#define INTC_HANDLER XScuGic_InterruptHandler

```

9) 打开串口终端，运行程序



- 10) 如果一直不按按键，串口显示 “No button pressed.”，如果按下 “PL_KEY” 显示 “Successfully ran Gpio Interrupt Tapp Example”。

```

Press button to Generate Interrupt
Press button to Generate Interrupt
No button pressed.
Press button to Generate Interrupt
No button pressed.
Press button to Generate Interrupt
Successfully ran Gpio Interrupt Tapp Example
Press button to Generate Interrupt
No button pressed.
Press button to Generate Interrupt
Successfully ran Gpio Interrupt Tapp Example
Press button to Generate Interrupt
No button pressed.
Press button to Generate Interrupt
Successfully ran Gpio Interrupt Tapp Example

```

12.3 实验总结

PL 端可以给 PS 发送中断信号，这提高了 PL 和 PS 数据交互的效率，在需要大数量、低延时的应用中需要用到中断处理。

第十三章 以太网实验 (LWIP)

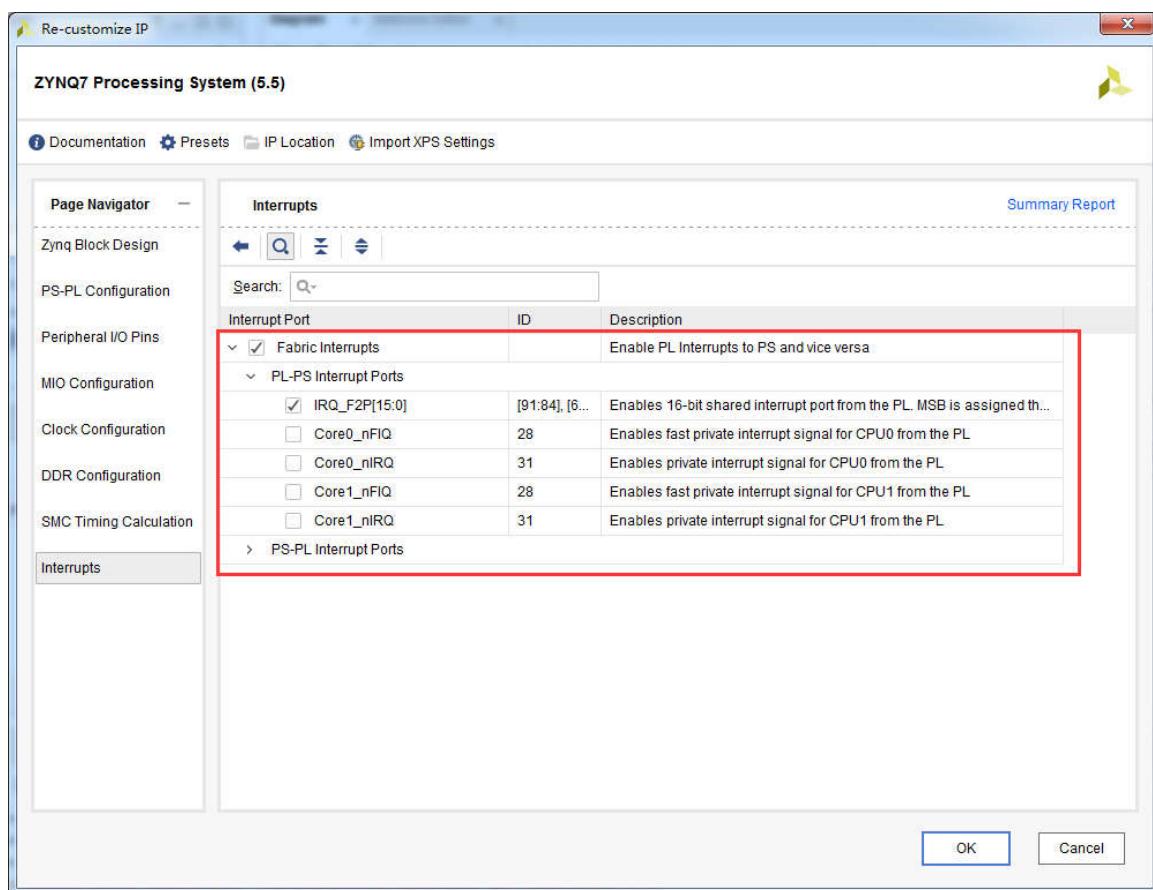
实验 Vivado 工程为 “net_test”。

开发板有 2 路千兆以太网，都是通过 RGMII 接口连接，其中一路连接到 PS 端，PS 端集成了 2 个 EMAC 控制器，可以由 ARM 直接使用千兆以太网，另外一路接在 PL 端，可以由 FPGA 控制，本实验演示如何使用 SDK 自带的 LWIP 模板进行千兆以太网 TCP 通信。

LWIP 虽然是轻量级协议栈，但如果从来没有使用过，使用起来会有一定的困难，建议先熟悉 LWIP 的相关知识。

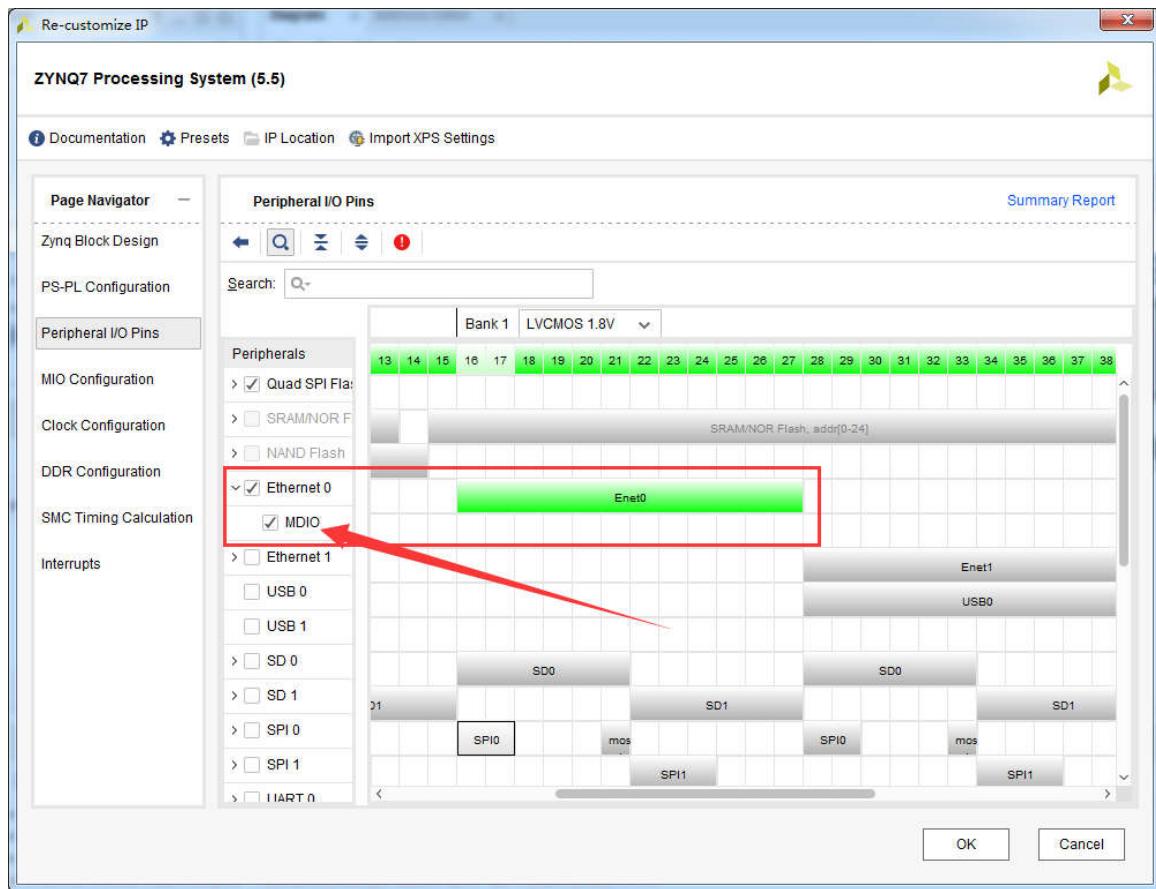
13.1 Vivado 工程建立

新建一个 “net_test” vivado 工作，添加 ZYNQ，按照前面的教程配置串口，并且勾选中断。详细参数可以参考例程附带的 vivado 工程。

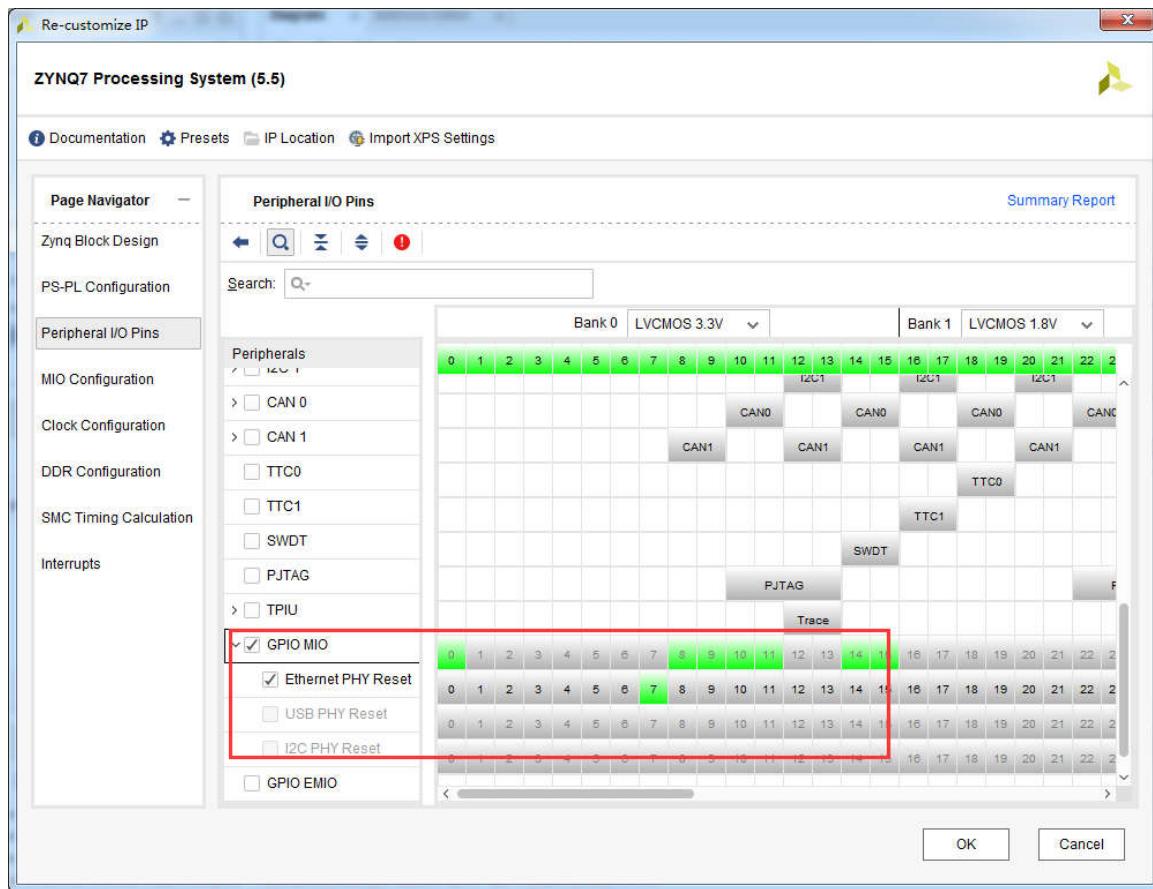


13.1.1 PS 端的以太网配置

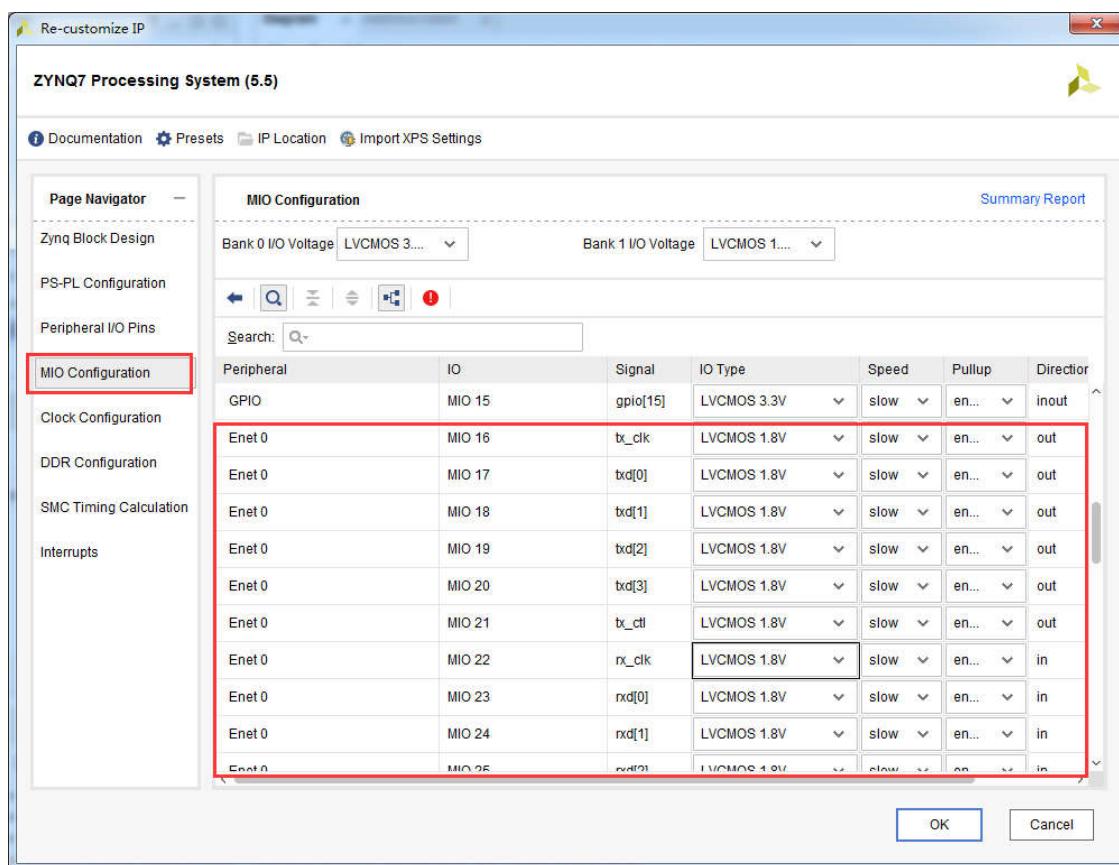
- 1) 使能 “Enet0” 和 “MDIO”



- 2) 勾选 GPIO MIO , 选择 Ethernet PHY Reset 为 MIO7

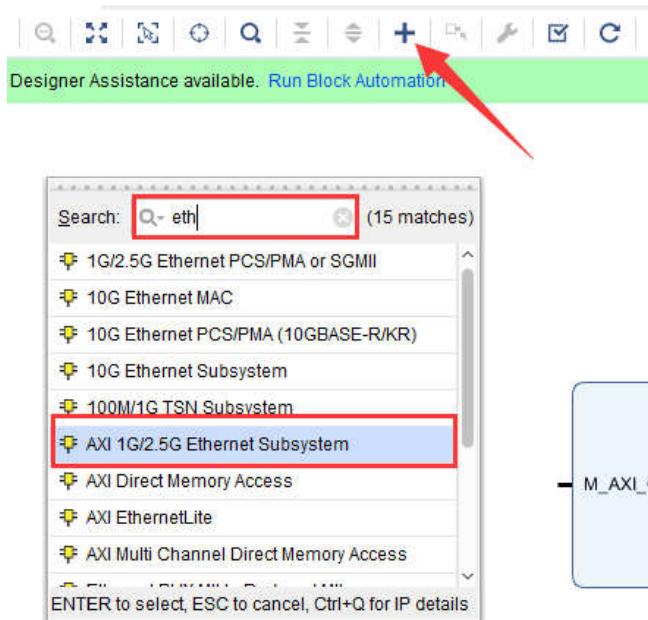


3) 保持 Enet0 的电平标准为 LVC MOS18

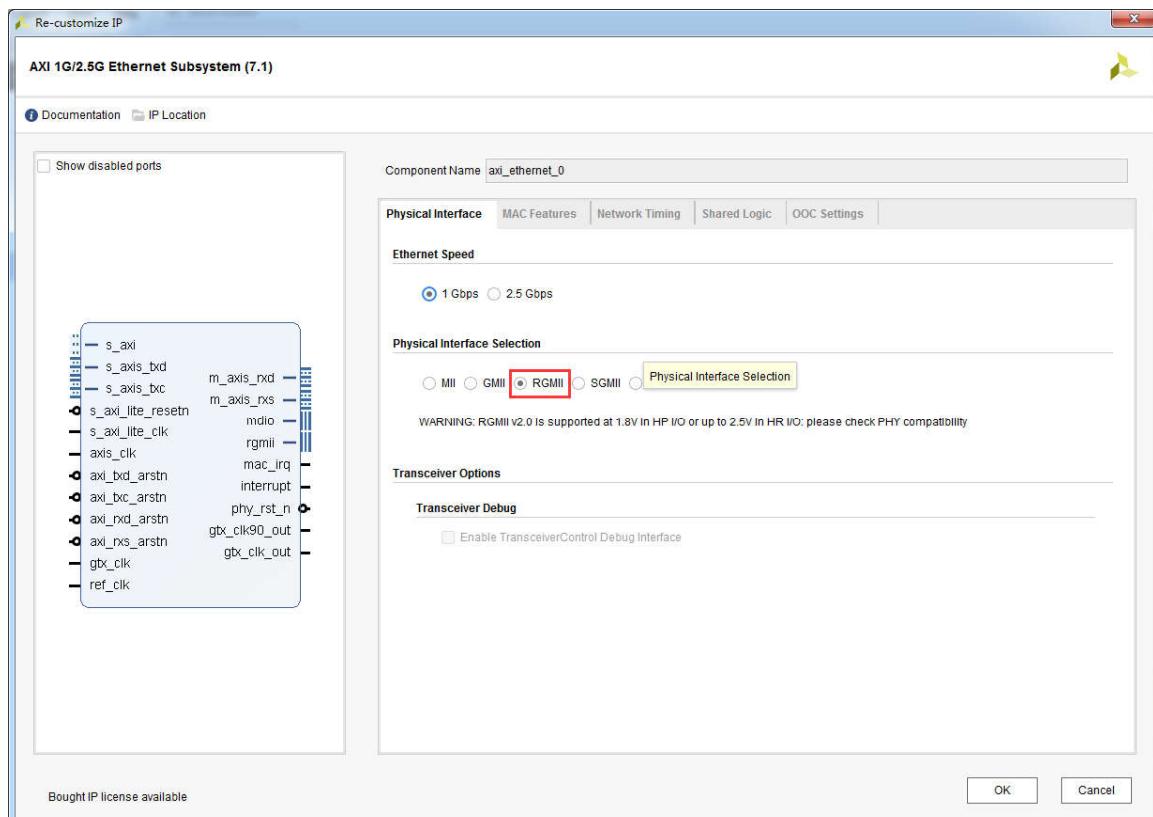


13.1.2 PL 端 AXI 以太网配置

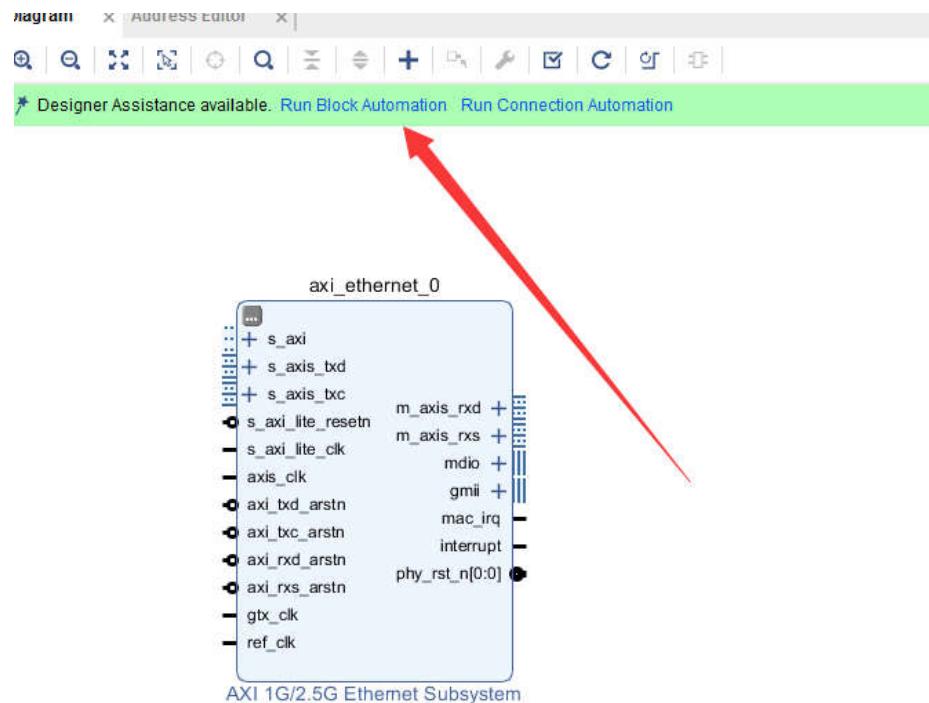
- 1) 搜索 “eth” , 添加一个 “AXI 1G/2.5G Ethernet Subsystem”



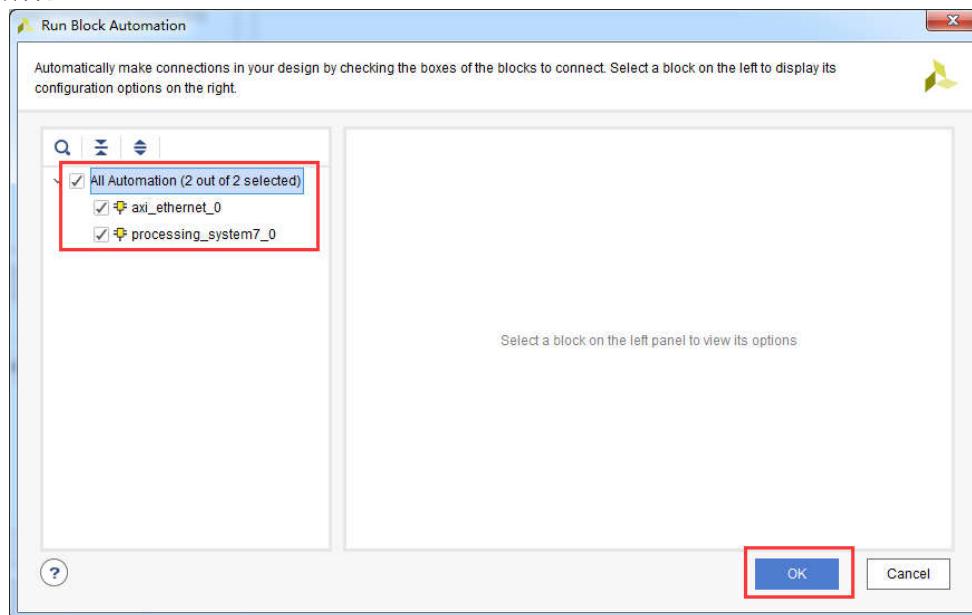
- 2) 双击刚才添加的模块 , 修改参数 , 物理接口选择 “RGMII” , 其他参数默认



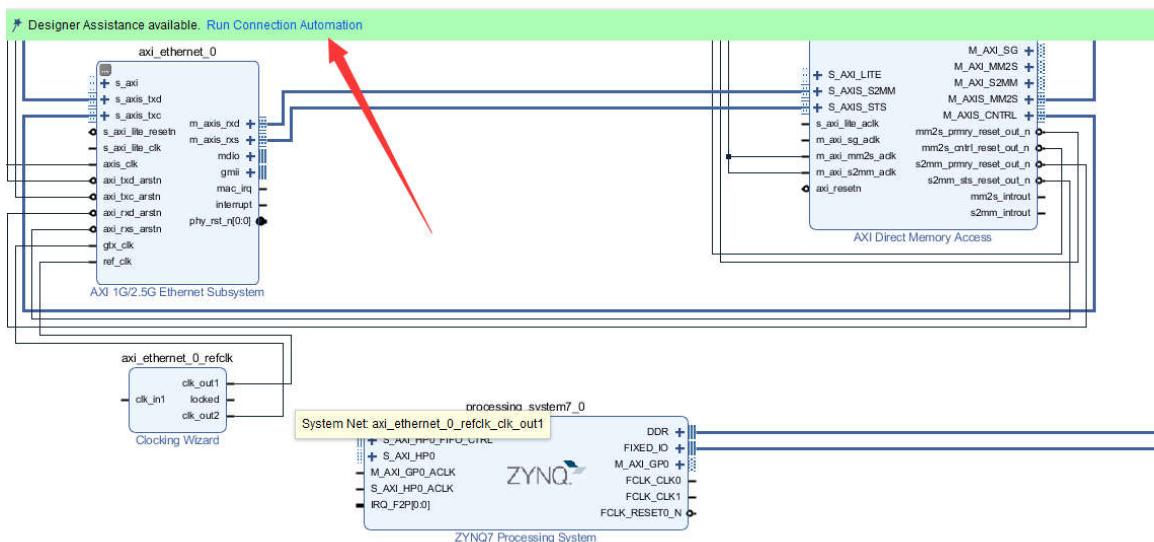
- 3) 点击 “Run Block Automation”



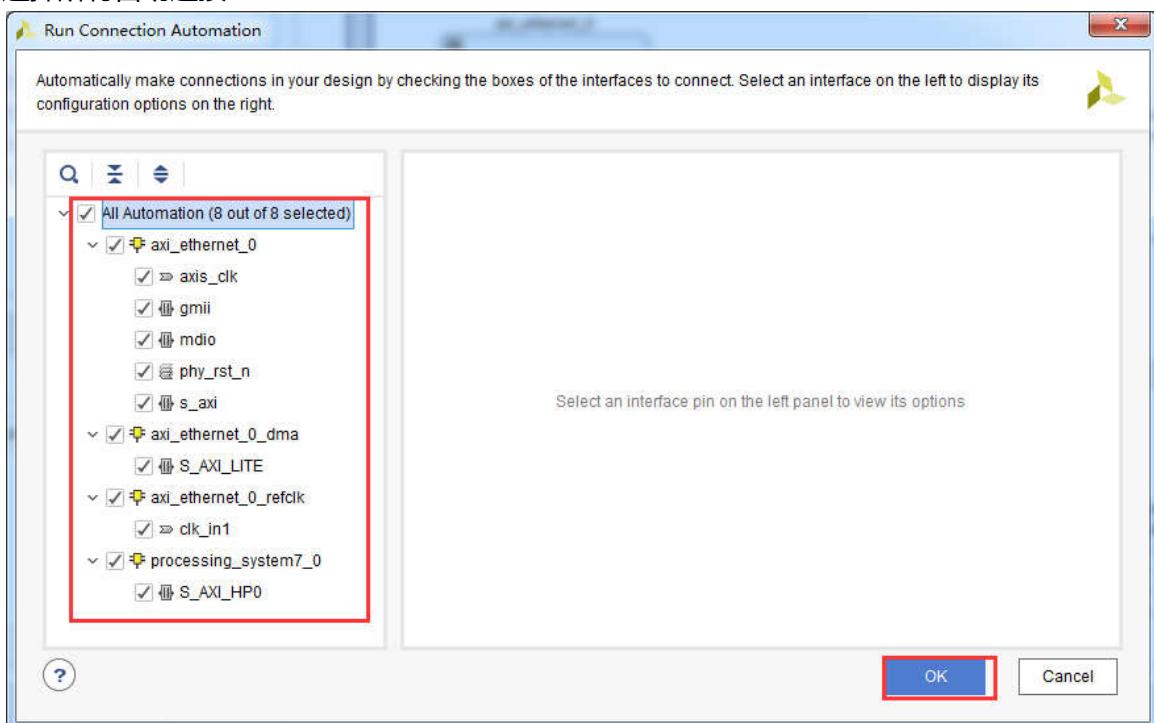
4) 选择所有



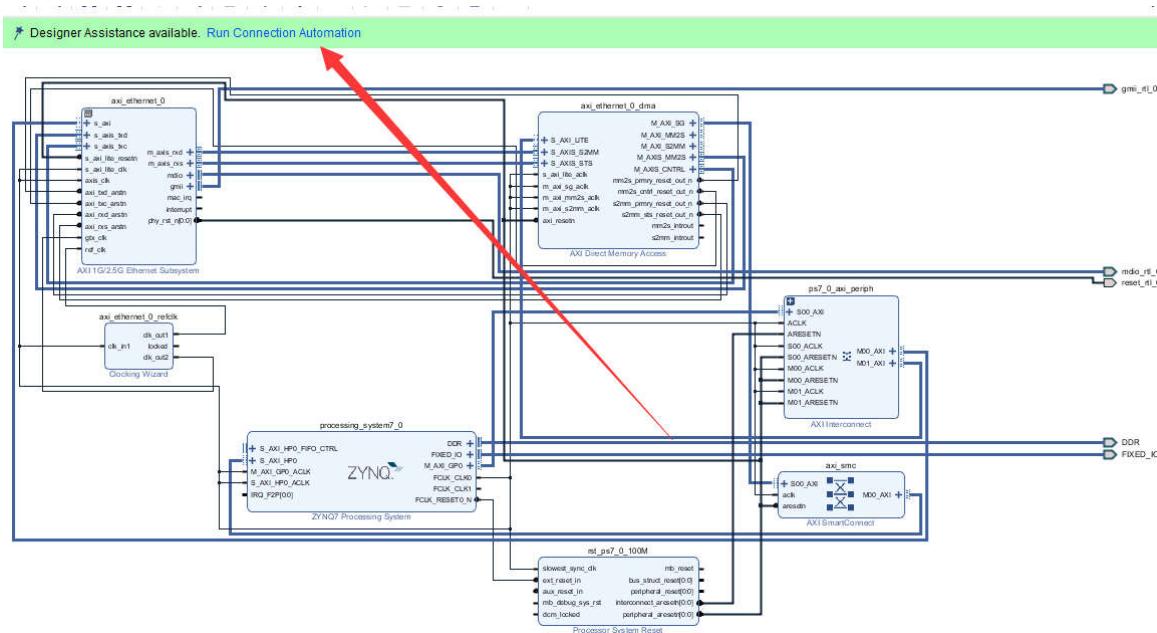
5) 这个时候可以 Vivado 自动插入了很多模块，点击 “Run Connection Automation” 自动连线和端口



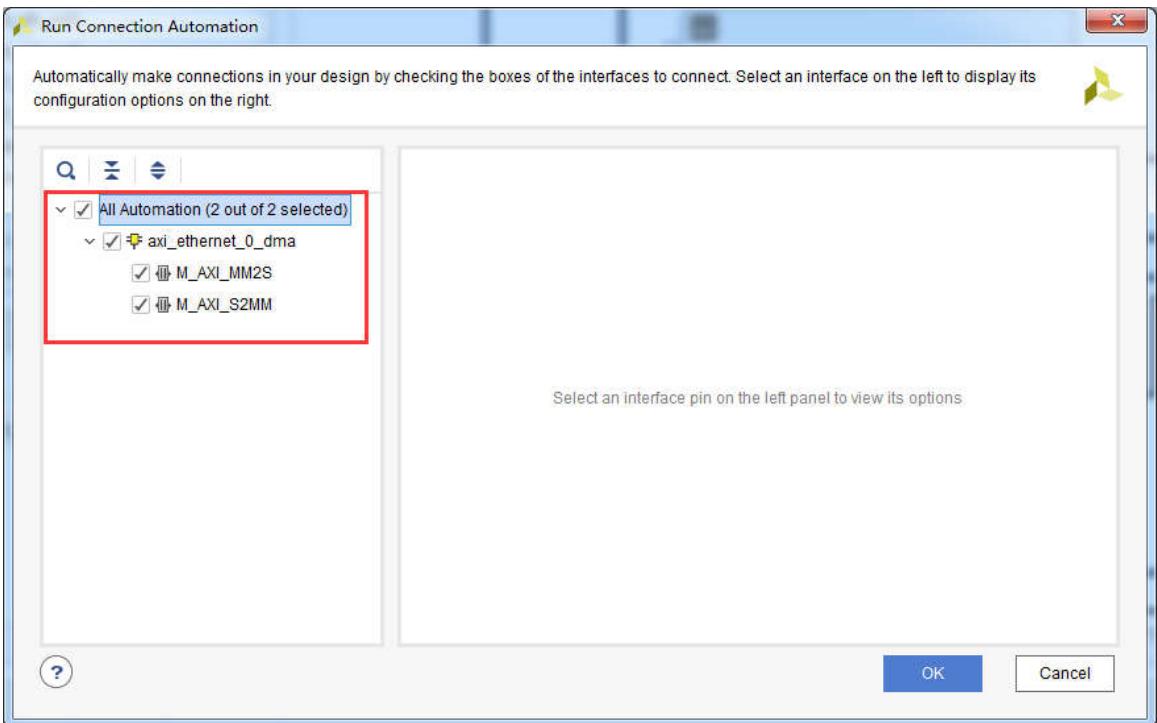
6) 选择所有自动连接



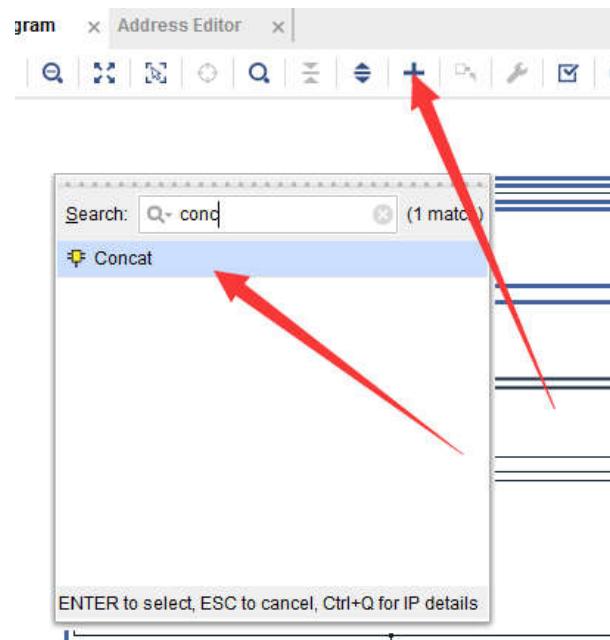
7) 然后一次并没有完成连线，再次点击“Run Connection Automation”



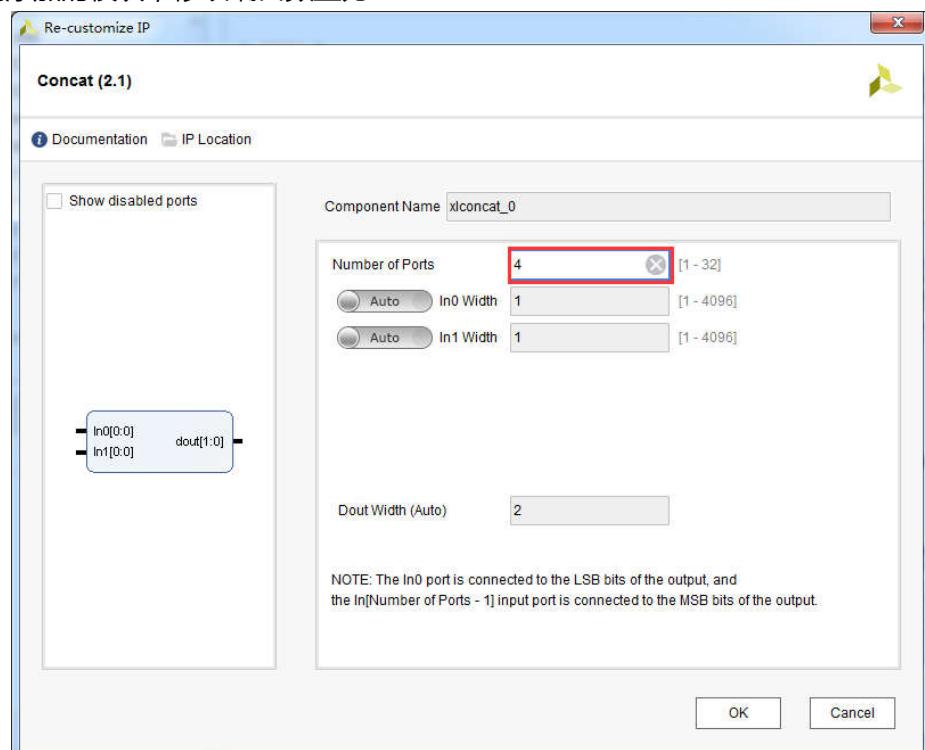
8) 选择所有



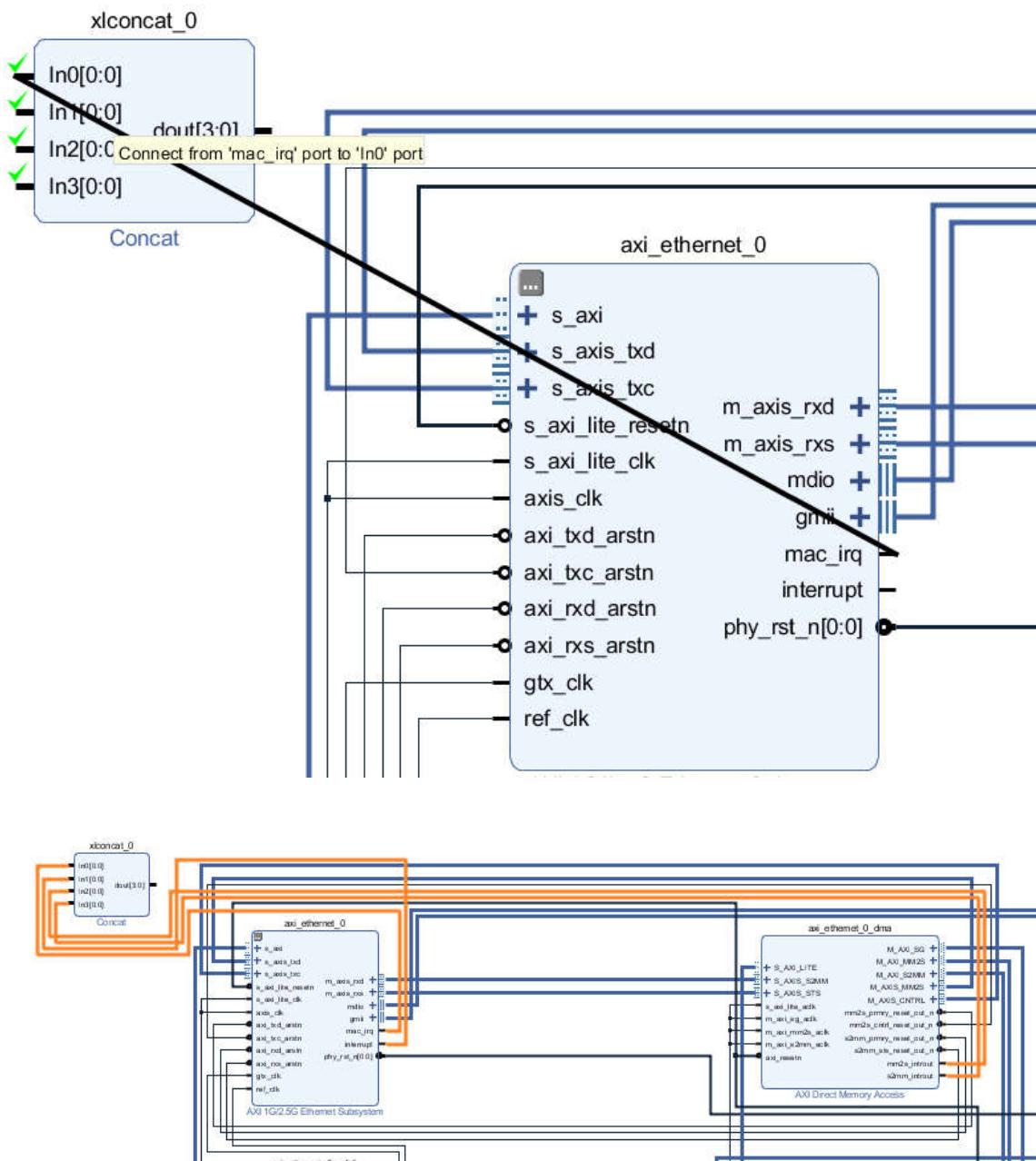
9) 处理中断，搜索“conc”添加“Concat”



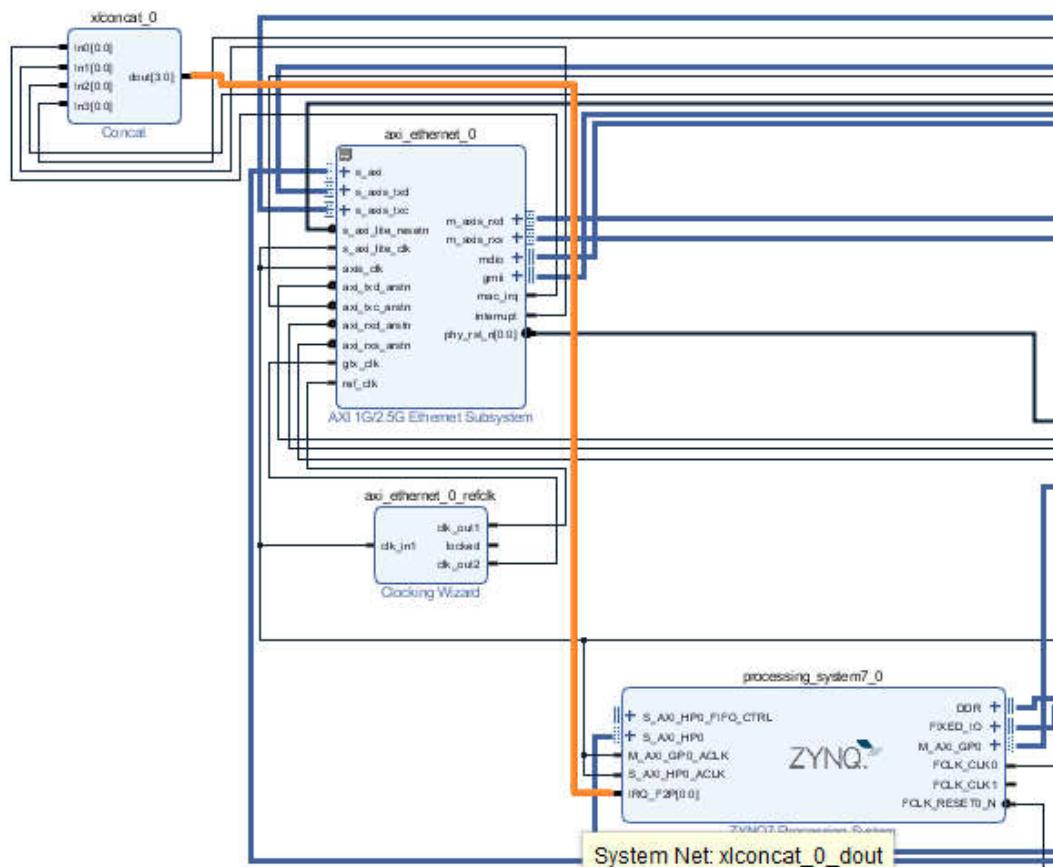
10) 双击刚添加的模块，修改端口数量为“4”



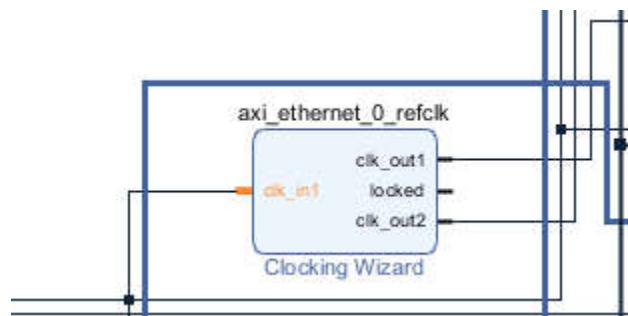
11) 将 4 个中断信号连接起来



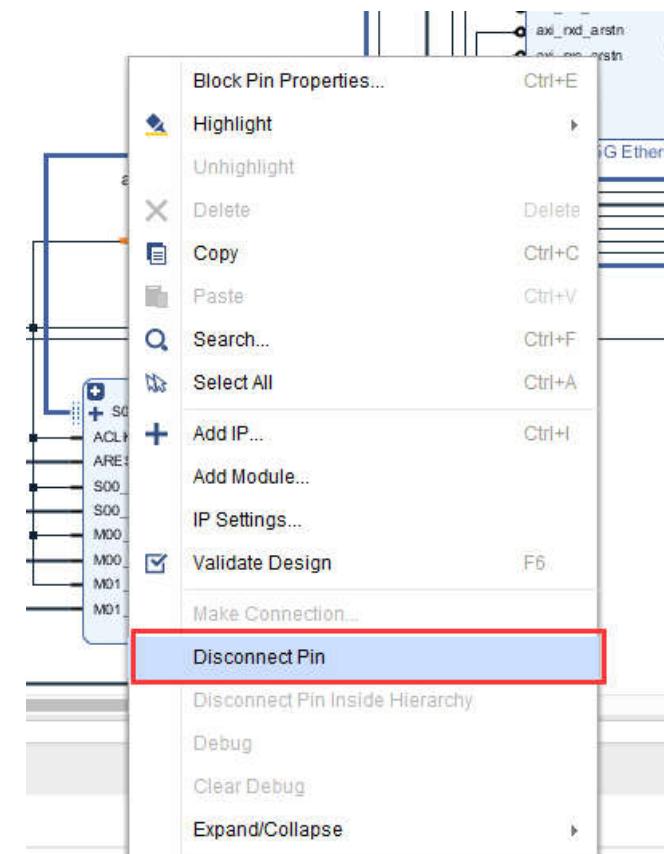
12) 然后和 ZYNQ 的 “IRQ_P2P” 接口相连



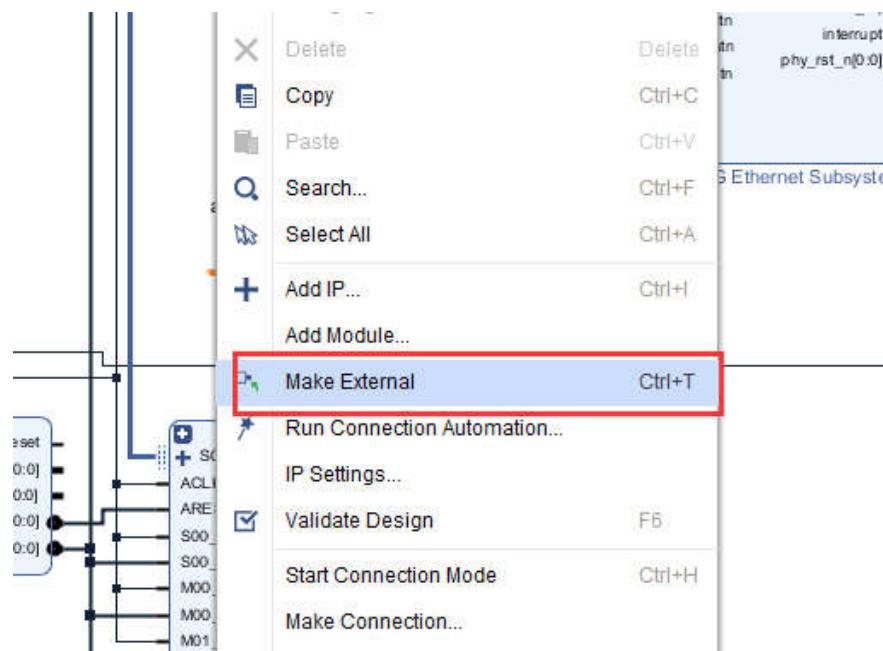
13) 处理以太网模块参考时钟，以太网模块需要一个 200Mhz 时钟和一个 125Mhz 时钟，我们修改为通过外部 50Mhz 的晶振输入然后通过 PLL 模块倍频，选择 “`axi_ethernet_0_refclk`” 的 “`clk_in1`” 脚，右键



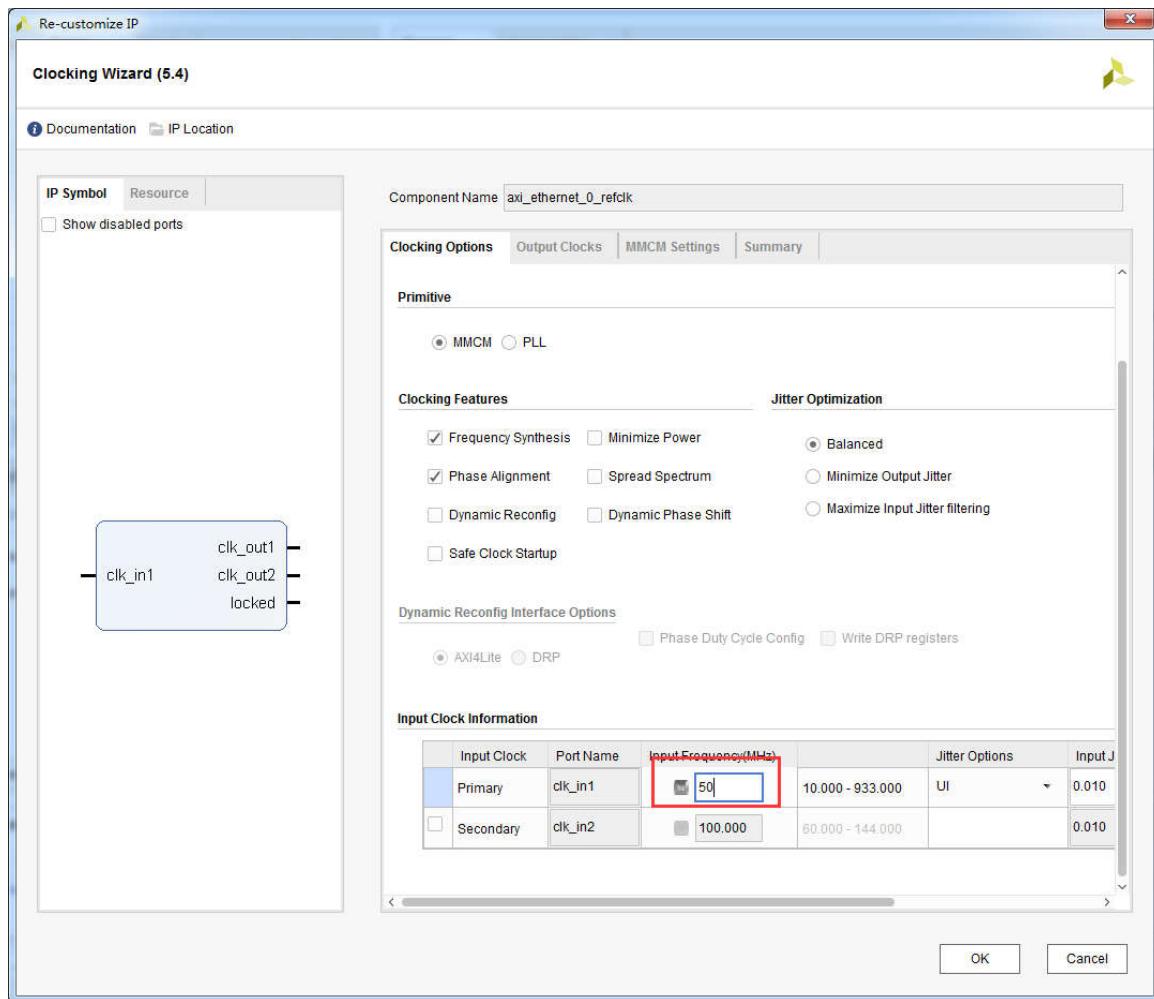
14) 选择 “Disconnect Pin”



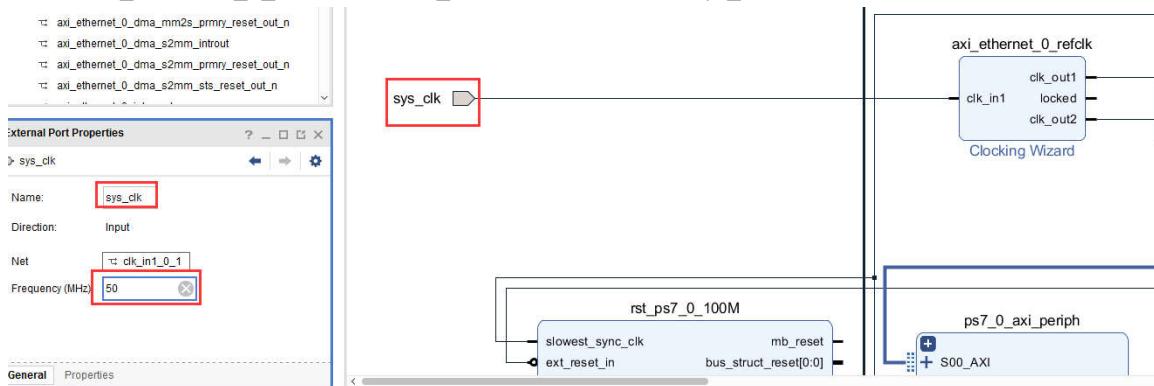
15) 然后右键 “Make External”



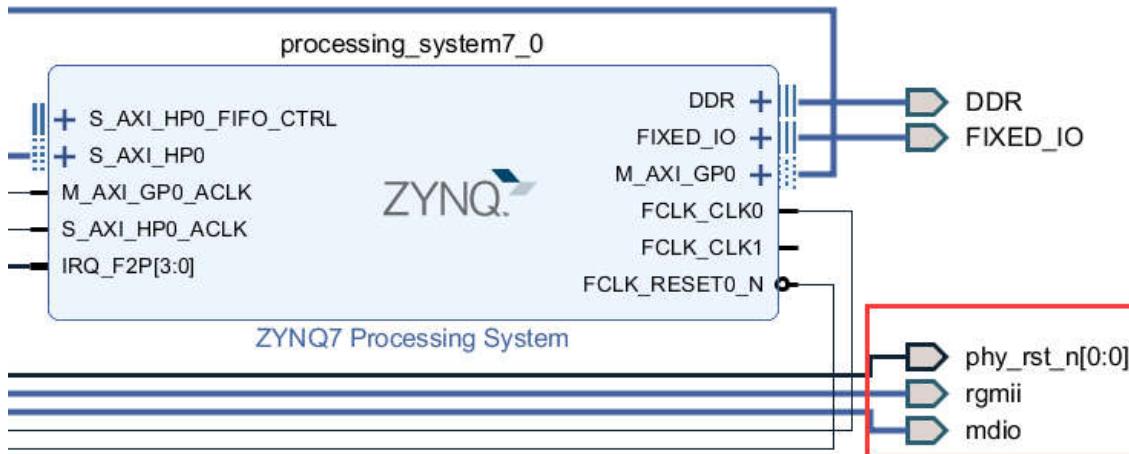
16) 双击 “axi_ethernet_0_refclk” 修改时钟输入频率为 50Mhz



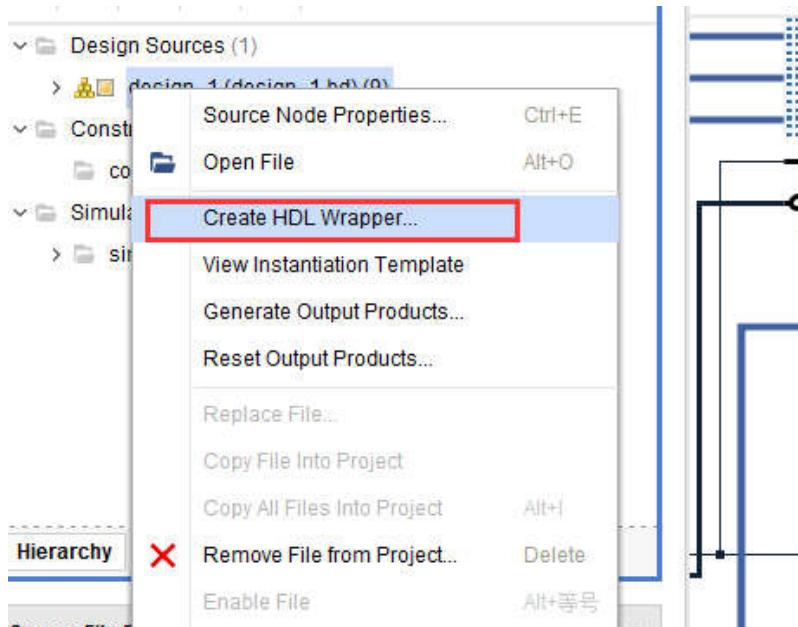
17) 修改 “axi_ethernet_0_refclk”的“clk_in1”的端口名称为 “sys_clk” ,并把频率改为 “50Mhz”



18) 修改其他端口的名称



19) 创建 HDL 文件



13.1.3 添加约束文件

```

set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]
set_property BITSTREAM.CONFIG.UNUSEDPIN PULLUP [current_design]

set_property PACKAGE_PIN J14 [get_ports sys_clk]
set_property IOSTANDARD LVCMOS18 [get_ports sys_clk]

set_property PACKAGE_PIN H16 [get_ports {mdio_mdc      }]
set_property PACKAGE_PIN H13 [get_ports {mdio_mdio_io   }]
set_property PACKAGE_PIN H12 [get_ports {phy_RST_n     }]
set_property PACKAGE_PIN G14 [get_ports {rgmii_rxc    }]

```

```
set_property PACKAGE_PIN E13 [get_ports {rgmii_rx_ctl}]  
set_property PACKAGE_PIN F13 [get_ports {rgmii_rd[0]}]  
set_property PACKAGE_PIN F12 [get_ports {rgmii_rd[1]}]  
set_property PACKAGE_PIN E12 [get_ports {rgmii_rd[2]}]  
set_property PACKAGE_PIN G11 [get_ports {rgmii_rd[3]}]  
set_property PACKAGE_PIN D11 [get_ports {rgmii_txc}]  
set_property PACKAGE_PIN E11 [get_ports {rgmii_tx_ctl}]  
set_property PACKAGE_PIN F10 [get_ports {rgmii_td[0]}]  
set_property PACKAGE_PIN G10 [get_ports {rgmii_td[1]}]  
set_property PACKAGE_PIN D10 [get_ports {rgmii_td[2]}]  
set_property PACKAGE_PIN E10 [get_ports {rgmii_td[3]}]  
  
set_property IOSTANDARD LVCMOS18 [get_ports {mdio_mdc}]  
set_property IOSTANDARD LVCMOS18 [get_ports {mdio_mdio_io}]  
set_property IOSTANDARD LVCMOS18 [get_ports {phy_rst_n}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_rxc}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_rx_ctl}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_rd[0]}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_rd[1]}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_rd[2]}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_rd[3]}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_txc}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_tx_ctl}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_td[0]}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_td[1]}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_td[2]}]  
set_property IOSTANDARD LVCMOS18 [get_ports {rgmii_td[3]}]
```

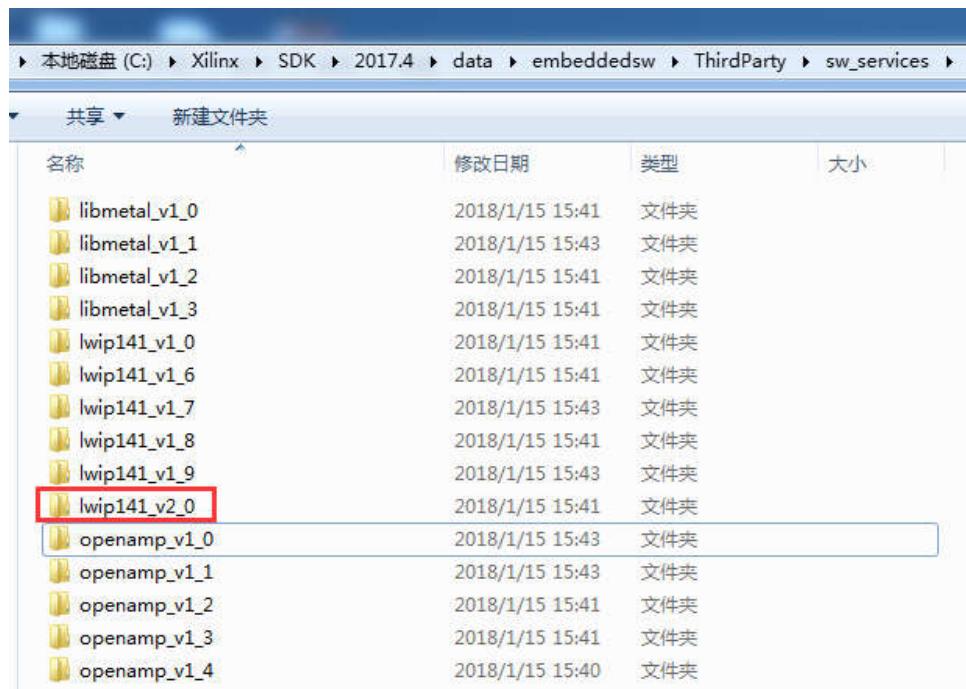
20) 编译生成 bit 文件，然后导出硬件信息，启动 SDK

13.2 SDK 程序

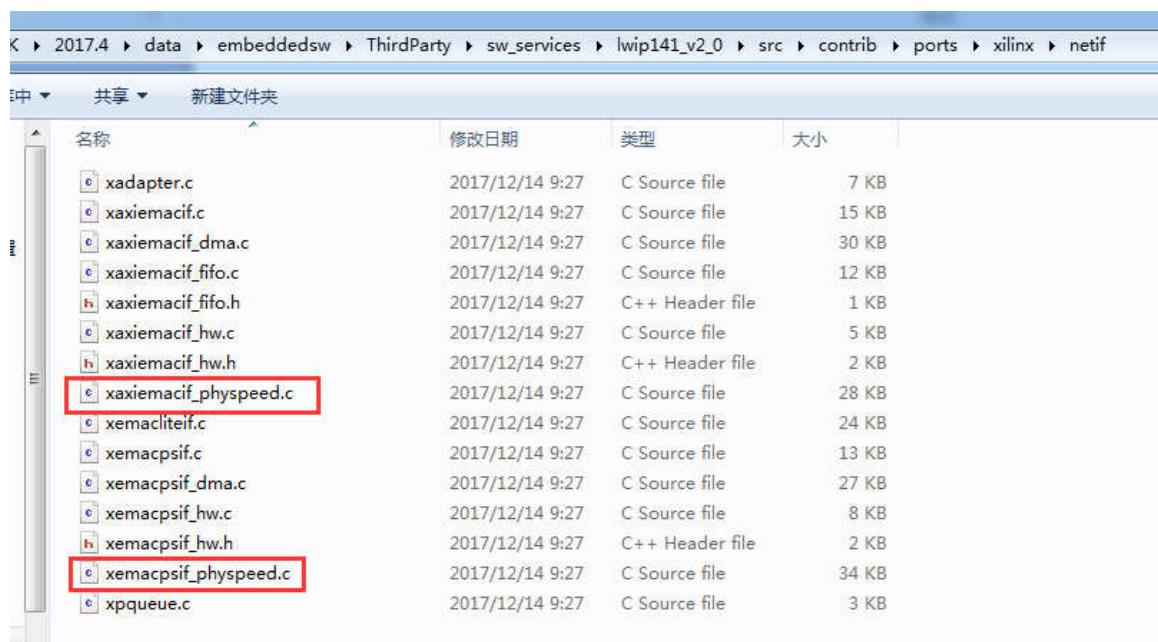
13.2.1 LWIP 库修改

由于自带的 LWIP 库只能识别部分 phy 芯片，如果开发板所用的 phy 芯片不在默认支持范围内，要修改库文件。也可以直接使用修改过的库替换原有的库。

1) 找到库文件目录 “X:\Xilinx\SDK\2017.4\data\embeddedsw\ThirdParty\sw_services”



- 2) 找到要修改的文件目录 “lwip141_v2_0\src\contrib\ports\xilinx\netif” 中文件 “xaxiemacif_physpeed.c” 和 “xemacpsif_physpeed.c” 要修改。



- 3) 修改 “xaxiemacif_physspeed.c” 文件，添加相关宏定义

```

#define IEEE_MMD_ACCESS_CTRL_DEVAD_MASK      0x1F
#define IEEE_MMD_ACCESS_CTRL_PIDEVAD_MASK    0x801F
#define IEEE_MMD_ACCESS_CTRL_NOPIDEVAD_MASK  0x401F

#define PHY_RO_ISOLATE                      0x0400
#define PHY_DETECT_REG                      1
#define PHY_IDENTIFIER_1_REG                2
#define PHY_IDENTIFIER_2_REG                3
#define PHY_DETECT_MASK                     0x1808
#define PHY_MARVELL_IDENTIFIER            0x0141
#define PHY_TI_IDENTIFIER                 0x2000

/* Marvel PHY flags */
#define MARVEL_PHY_IDENTIFIER              0x141
#define MARVEL_PHY_MODEL_NUM_MASK        0x3F0
#define MARVEL_PHY_88E1111_MODEL         0xC0
#define MARVEL_PHY_88E1116R_MODEL        0x240
#define PHY_88E1111_RGMII_RX_CLOCK_DELAYED_MASK 0x0080

/* TI PHY Flags */
#define TI_PHY_DETECT_MASK                0x796D
#define TI_PHY_IDENTIFIER                 0x2000
#define TI_PHY_DP83867_MODEL             0xA231
#define DP83867_RGMII_CLOCK_DELAY_CTRL_MASK 0x0003
#define DP83867_RGMII_TX_CLOCK_DELAY_MASK 0x0030
#define DP83867_RGMII_RX_CLOCK_DELAY_MASK 0x0003

/* TI DP83867 PHY Registers */
#define DP83867_R32_RGMIICTL1           0x32
#define DP83867_R86_RGMIIDCTL          0x86

#define MICREL_PHY_IDENTIFIER            0x22
#define MICREL_PHY_KSZ9031_MODEL        0x220

#define TI_PHY_REGCR                   0xD
#define TI_PHY_ADDDR                   0xE

```

4) 添加 phy 速度获取函数

```

unsigned int get_phy_speed_ksz9031(XAxiEthernet *xaxiemacp, u32 phy_addr)
{
    u16 control;
    u16 status;
    u16 partner_capabilities;
    xil_printf("Start PHY autonegotiation \r\n");

    XAxiEthernet_PhWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
    XAxiEthernet_PhRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
    //control |= IEEE_RGMII_TXRX_CLOCK_DELAYED_MASK;
    control &= ~(0x10);
    XAxiEthernet_PhWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, control);

    XAxiEthernet_PhWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

    XAxiEthernet_PhRead(xaxiemacp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
    control |= IEEE_ASYMMETRIC_PAUSE_MASK;
    control |= IEEE_PAUSE_MASK;
    control |= ADVERTISE_100;
    control |= ADVERTISE_10;
    XAxiEthernet_PhWrite(xaxiemacp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

    XAxiEthernet_PhRead(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                        &control);
    control |= ADVERTISE_1000;
    XAxiEthernet_PhWrite(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                        control);

    XAxiEthernet_PhWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    XAxiEthernet_PhRead(xaxiemacp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,
                        &control);
    control |= (7 << 12); /* max number of gigabit attempts */
    control |= (1 << 11); /* enable downshift */
    XAxiEthernet_PhWrite(xaxiemacp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,

```

```

        control);
XAxiEthernet_PhRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
control |= IEEE_STAT_AUTONEGOTIATE_RESTART;

XAxiEthernet_PhWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

XAxiEthernet_PhRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
control |= IEEE_CTRL_RESET_MASK;
XAxiEthernet_PhWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

while (1) {
    XAxiEthernet_PhRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    if (control & IEEE_CTRL_RESET_MASK)
        continue;
    else
        break;
}
xil_printf("Waiting for PHY to complete autonegotiation.\r\n");

XAxiEthernet_PhRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);
while ( !(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE) ) {
    sleep(1);
    XAxiEthernet_PhRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET,
                         &status);
}

xil_printf("autonegotiation complete \r\n");

XAxiEthernet_PhRead(xaxiemacp, phy_addr, 0x1f, &partner_capabilities);

if ( (partner_capabilities & 0x40) == 0x40)/* 1000Mbps */
    return 1000;
else if ( (partner_capabilities & 0x20) == 0x20)/* 100Mbps */
    return 100;
else if ( (partner_capabilities & 0x10) == 0x10)/* 10Mbps */
    return 10;
else
    return 0;
}

```

5) 修改函数 “get_IEEE_phy_speed” , 添加对 KSZ9031 的支持。

```

unsigned get_IEEE_phy_speed(XAxiEthernet *xaxiemacp)
{
    u16 phy_identifier;
    u16 phy_model;
    u8 phytype;

#ifndef XPAR_AXIETHERNET_0_BASEADDR
    u32 phy_addr = detect_phy(xaxiemacp);
#endif

    /* Get the PHY Identifier and Model number */
    XAxiEthernet_PhRead(xaxiemacp, phy_addr, PHY_IDENTIFIER_1_REG, &phy_identifier);
    XAxiEthernet_PhRead(xaxiemacp, phy_addr, PHY_IDENTIFIER_2_REG, &phy_model);

    /* Depending upon what manufacturer PHY is connected, a different mask is
     * needed to determine the specific model number of the PHY. */
    if (phy_identifier == MARVEL_PHY_IDENTIFIER) {
        phy_model = phy_model & MARVEL_PHY_MODEL_NUM_MASK;

        if (phy_model == MARVEL_PHY_88E1116R_MODEL) {
            return get_phy_speed_88E1116R(xaxiemacp, phy_addr);
        } else if (phy_model == MARVEL_PHY_88E1111_MODEL) {
            return get_phy_speed_88E1111(xaxiemacp, phy_addr);
        }
    } else if (phy_identifier == TI_PHY_IDENTIFIER) {
        phy_model = phy_model & TI_PHY_DP83867_MODEL;
        phytype = XAxiEthernet_GetPhysicalInterface(xaxiemacp);

        if (phy_model == TI_PHY_DP83867_MODEL && phytype == XAE_PHY_TYPE_SGMII) {
            return get_phy_speed_TI_DP83867_SGMII(xaxiemacp, phy_addr);
        }

        if (phy_model == TI_PHY_DP83867_MODEL) {
            return get_phy_speed_TI_DP83867(xaxiemacp, phy_addr);
        }
    }
}

```

```

else if(phy_identifier == MICREL_PHY_IDENTIFIER)
{
    xil_printf("Phy %d is KSZ9031\n\r", phy_addr);
    get_phy_speed_ksz9031(xaxiemacp, phy_addr);
}
else {
    LWIP_DEBUGF(NETIF_DEBUG, ("XAxiEthernet get_IEEE_physpeed: Detected PHY with unknown
identifier/model.\r\n"));
}
#endif
#ifdef PCM_PMA_CORE_PRESENT
    return get_phy_negotiated_speed(xaxiemacp, phy_addr);
#endif
}

```

6) 修改 “xemacpsif_physpeed.c” 文件添加宏定义

```

#define IEEE_PAGE_ADDRESS_REGISTER          22
#define IEEE_CTRL_1GBPS_LINKSPEED_MASK    0x2040
#define IEEE_CTRL_LINKSPEED_MASK          0x0040
#define IEEE_CTRL_LINKSPEED_1000M         0x0040
#define IEEE_CTRL_LINKSPEED_100M          0x2000
#define IEEE_CTRL_LINKSPEED_10M           0x0000
#define IEEE_CTRL_RESET_MASK              0x8000

#define IEEE_SPEED_MASK                  0xC000
#define IEEE_SPEED_1000                 0x8000
#define IEEE_SPEED_100                  0x4000

#define IEEE_CTRL_RESET_MASK             0x8000
#define IEEE_CTRL_AUTONEGOTIATE_ENABLE   0x1000
#define IEEE_STAT_AUTONEGOTIATE_COMPLETE 0x0020
#define IEEE_STAT_AUTONEGOTIATE_RESTART  0x0200
#define IEEE_RGMII_TXRX_CLOCK_DELAYED_MASK 0x0030
#define IEEE_ASYMMETRIC_PAUSE_MASK       0x0800
#define IEEE_PAUSE_MASK                 0x0400
#define IEEE_AUTONEG_ERROR_MASK         0x8000

#define PHY_DETECT_REG                  1
#define PHY_IDENTIFIER_1_REG            2
#define PHY_IDENTIFIER_2_REG            3
#define PHY_DETECT_MASK                0x1808
#define PHY_MARVELL_IDENTIFIER        0x0141
#define PHY_TI_IDENTIFIER              0x2000
#define PHY_XILINX_PCS_PMA_ID1        0x0174
#define PHY_XILINX_PCS_PMA_ID2        0x0C00

#define MICREL_PHY_IDENTIFIER          0x22
#define MICREL_PHY_KSZ9031_MODEL       0x220

#define XEMACPS_GMII2RGMII_SPEED1000_FD 0x140
#define XEMACPS_GMII2RGMII_SPEED100_FD   0x2100
#define XEMACPS_GMII2RGMII_SPEED10_FD    0x100
#define XEMACPS_GMII2RGMII_DEF_NUM     0x10

```

7) 添加 phy 速度获取函数

```

static u32_t get_phy_speed_ksz9031(XEmacPs *xemacpsp, u32_t phy_addr)
{
    u16_t temp;
    u16_t control;
    u16_t status;
    u16_t status_speed;
    u32_t timeout_counter = 0;
    u32_t temp_speed;
    u32_t phyregtemp;

    xil_printf("Start PHY autonegotiation \r\n");

    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
    XEmacPs_PhphyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
    control |= IEEE_RGMII_TXRX_CLOCK_DELAYED_MASK;
    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, control);
}

```

```

XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
control |= IEEE_ASYMMETRIC_PAUSE_MASK;
control |= IEEE_PAUSE_MASK;
control |= ADVERTISE_100;
control |= ADVERTISE_10;
XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                &control);
control |= ADVERTISE_1000;
XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                 control);

XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,
                &control);
control |= (7 << 12); /* max number of gigabit attempts */
control |= (1 << 11); /* enable downshift */
XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,
                 control);
XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
control |= IEEE_STAT_AUTONEGOTIATE_RESTART;
XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
control |= IEEE_CTRL_RESET_MASK;
XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

while (1) {
    XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    if (control & IEEE_CTRL_RESET_MASK)
        continue;
    else
        break;
}

XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);

xil_printf("Waiting for PHY to complete autonegotiation.\r\n");

while ( !(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE) ) {
    sleep(1);
    XEmacPs_PhRead(xemacpsp, phy_addr,
                    IEEE_COPPER_SPECIFIC_STATUS_REG_2, &temp);
    timeout_counter++;

    if (timeout_counter == 30) {
        xil_printf("Auto negotiation error \r\n");
        return;
    }
    XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);
}
xil_printf("autonegotiation complete \r\n");

XEmacPs_PhRead(xemacpsp, phy_addr, 0x1f,
                &status_speed);

if ( (status_speed & 0x40) == 0x40)/* 1000Mbps */
    return 1000;
else if ( (status_speed & 0x20) == 0x20)/* 100Mbps */
    return 100;
else if ( (status_speed & 0x10) == 0x10)/* 10Mbps */
    return 10;
else
    return 0;
return XST_SUCCESS;
}

```

8) 修改函数 “get_IIEEE_phy_speed”，添加对 KSZ9031 的支持

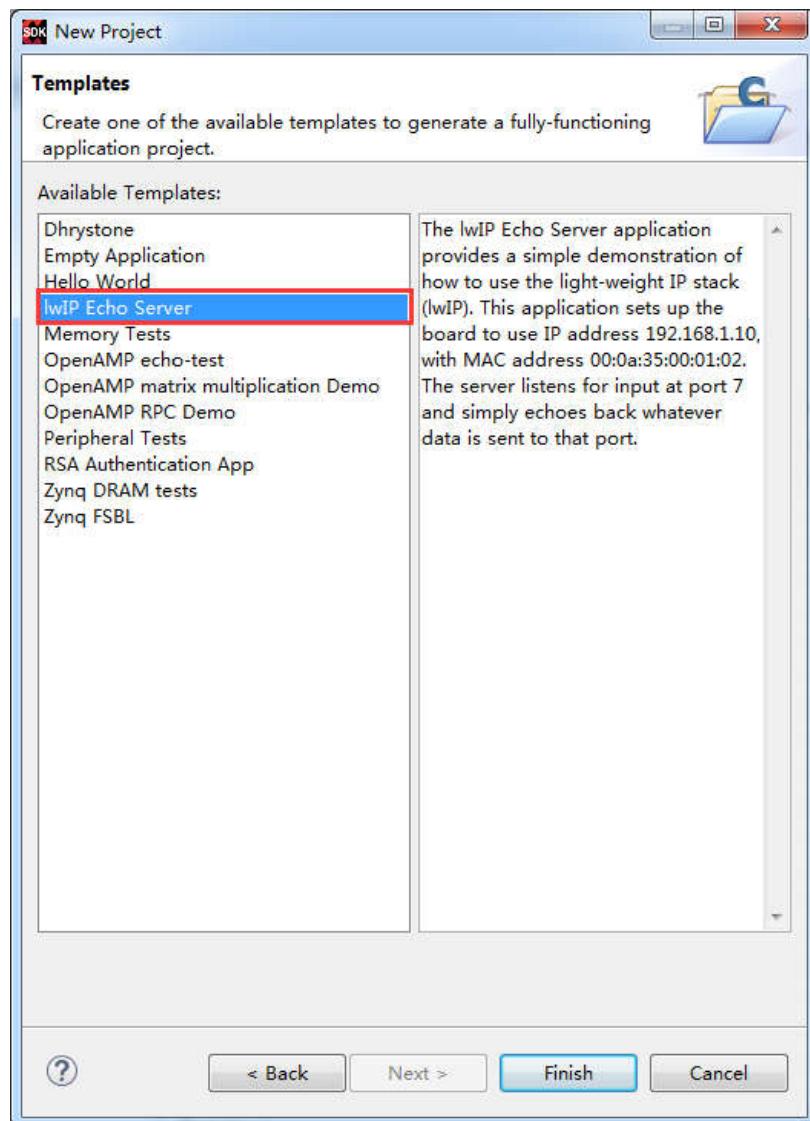
```

static u32_t get_IIEEE_phy_speed(XEmacPs *xemacpsp, u32_t phy_addr)
{
    u16_t phy_identity;
    u32_t RetStatus;

```

```
XEmacPs_PhysRead(xemacpsp, phy_addr, PHY_IDENTIFIER_1_REG,  
    &phy_identity);  
  
if(phy_identity == MICREL_PHY_IDENTIFIER)  
{  
    RetStatus = get_phy_speed_ksz9031(xemacpsp, phy_addr);  
}  
else if (phy_identity == PHY_TI_IDENTIFIER) {  
    RetStatus = get_TI_phy_speed(xemacpsp, phy_addr);  
} else {  
    RetStatus = get_Marvell_phy_speed(xemacpsp, phy_addr);  
}  
  
return RetStatus;  
}
```

13.2.2 创建基于 LWIP 模板的 APP

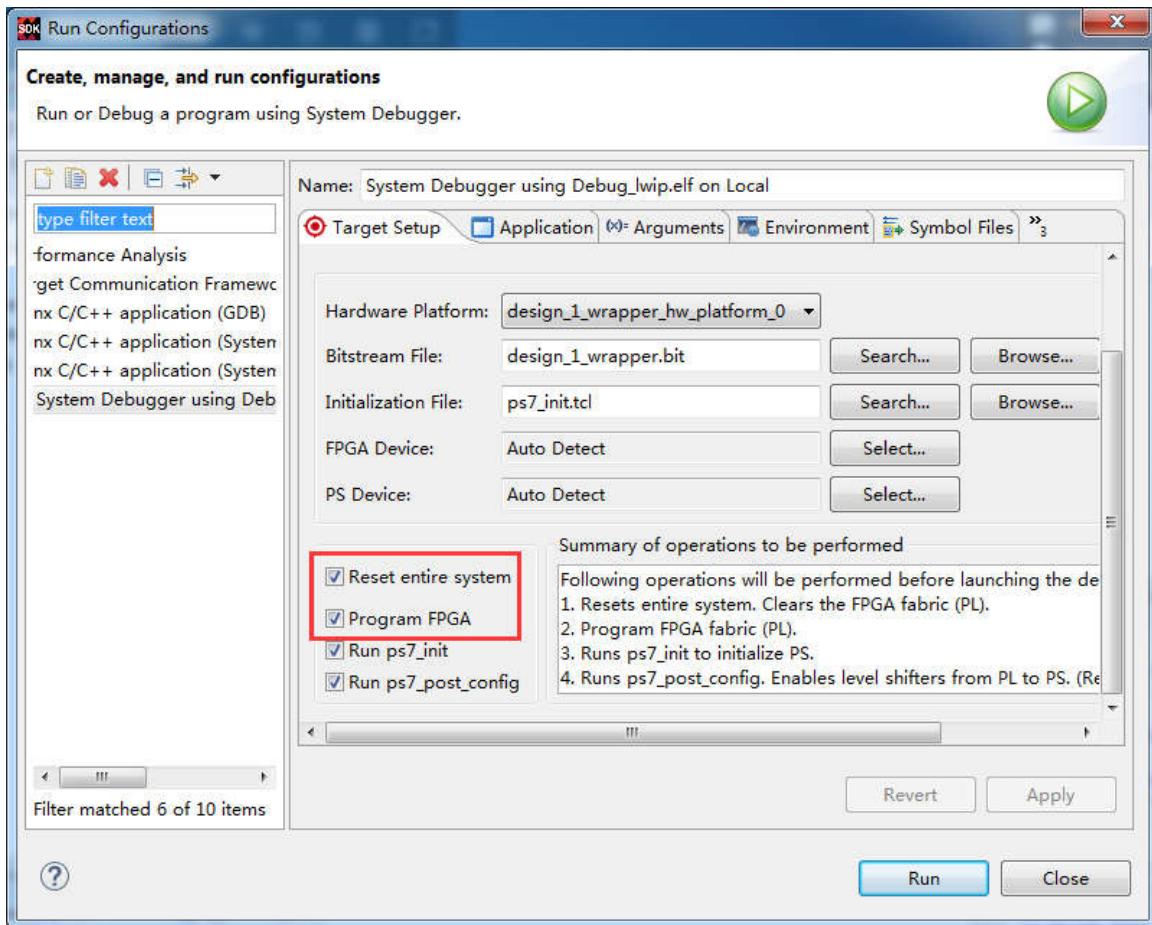


13.3 下载调试

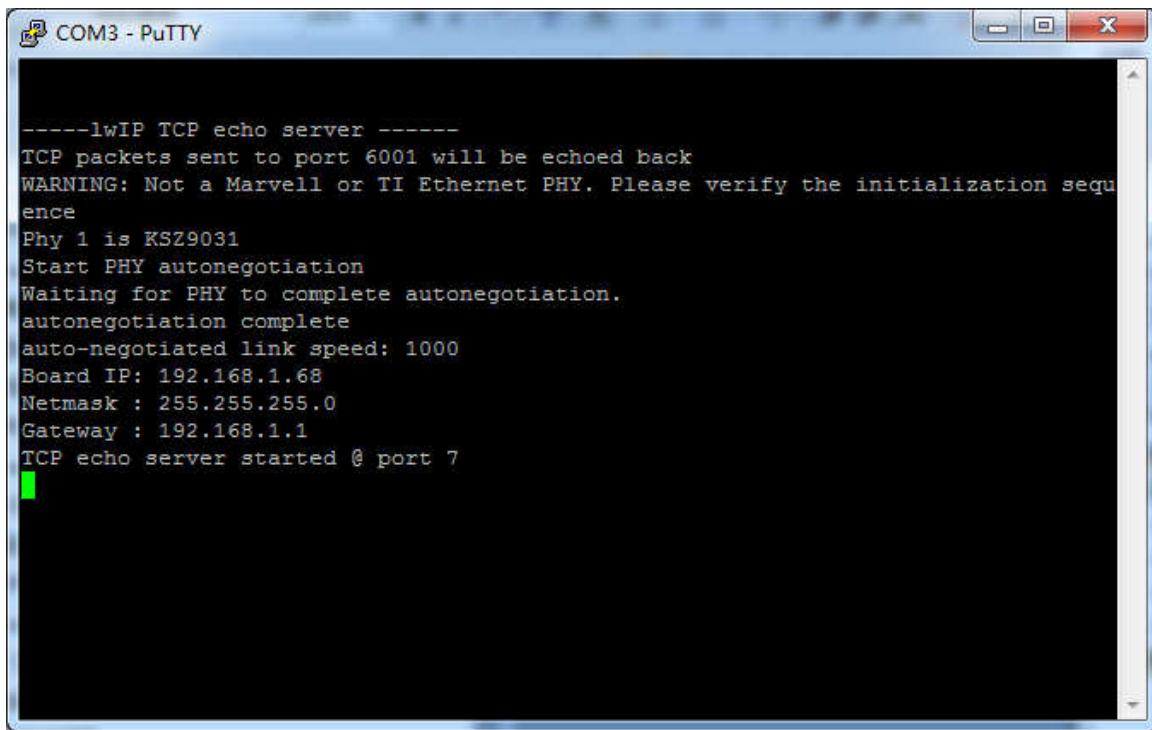
如果系统中既有 PS 以太网控制器，又有 PL 端 AXI 以太网控制器，LWIP 模板默认会选择 PL 端 AXI 以太网控制器，我们先测试 PL 端以太网，测试环境要求有一台支持 dhcp 的路由器，开发板连接路由器可以自动获取 IP 地址，实验主机和开发板在一个网络，可以相互通信。

13.3.1 PL 端以太网测试

- 1) 连接串口打开串口调试终端，连接好 PL 端以太网网线到路由器（ETH2）
- 2) 运行 SDK

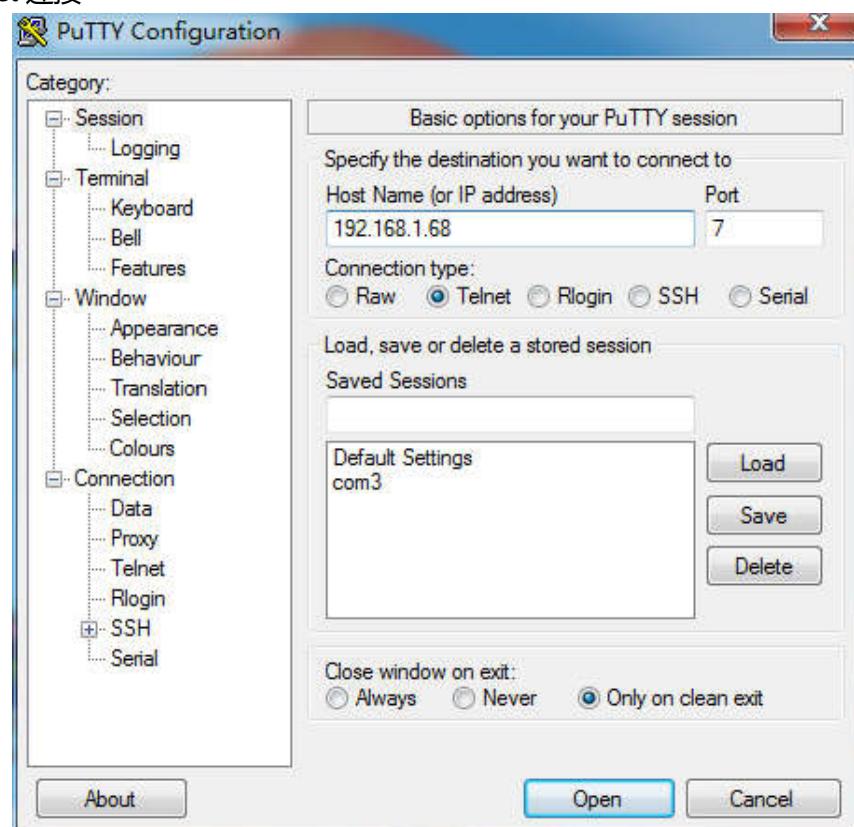


- 3) 可以看到串口打印出一些信息，可以看到自动获取到地址为“192.168.1.68”，连接速度 1000Mbps，tcp 端口为 7

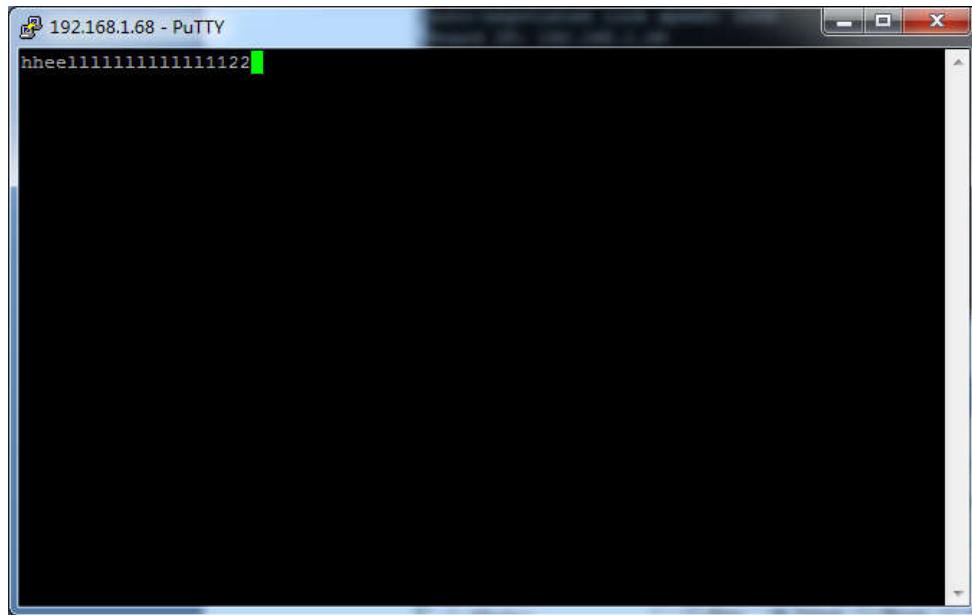


```
----lwIP TCP echo server ----
TCP packets sent to port 6001 will be echoed back
WARNING: Not a Marvell or TI Ethernet PHY. Please verify the initialization sequence
Phy 1 is KS29031
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
auto-negotiated link speed: 1000
Board IP: 192.168.1.68
Netmask : 255.255.255.0
Gateway : 192.168.1.1
TCP echo server started @ port 7
```

4) 使用 telnet 连接



5) 当输入一个字符时，开发板返回相同字符

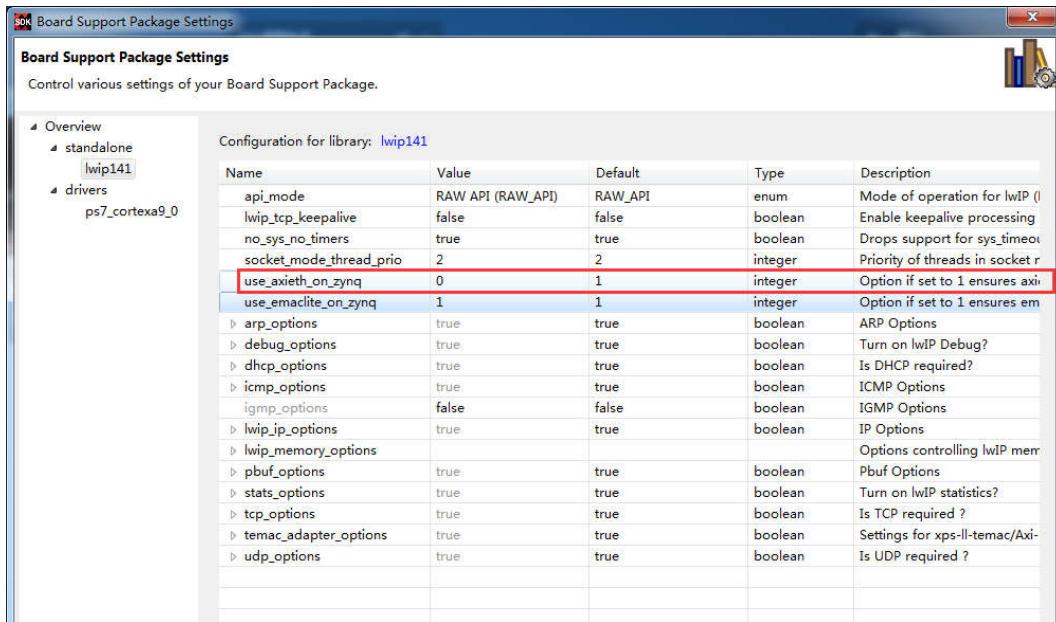


13.3.2 PS 端以太网测试

1) 修改 BSP 设置

The image shows two side-by-side windows. On the left is a file explorer-style interface showing a project structure. It includes a root folder "design_1_wrapper_hw_platform_0" and a subfolder "lwip" which contains "Binaries", "Includes", "Debug", and "src". Inside "src" is a folder "lwip_bsp" containing "BSP Documentation", "ps7_cortexa9_0" (which has "Makefile" and "system.mss" files), and "system.mss" is highlighted with a red box. On the right is a configuration interface titled "Iwip_bsp Board Support Package". It has a "Target Information" section with "Hardware Specification: F:\ax7350\test\ps_net\ps_net.sd" and "Target Processor: ps7_cortexa9_0". Below that is an "Operating System" section with "Board Support Package OS." and "Name: standalone". A button "Modify this BSP's Settings" is highlighted with a red box.

2) “use_axieth_on_zynq” 修改为 0，使用 PS 以太网



3) 修改 “platform_config.h” 文件

```

#ifndef __PLATFORM_CONFIG_H_
#define __PLATFORM_CONFIG_H_

#define USE_SOFTETH_ON_ZYNQ 0
//#define PLATFORM_EMAC_BASEADDR XPAR_AXI_ETHERNET_0_BASEADDR
#define PLATFORM_EMAC_BASEADDR XPAR_PS7_ETHERNET_0_BASEADDR
#define PLATFORM_ZYNQ

#endif

```

4) 网线连接 PS 端以太网到路由器

5) 运行程序，观察串口输出

```
Phy 1 is KSZ9031
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
auto-negotiated link speed: 1000
Board IP: 192.168.1.68
Netmask : 255.255.255.0
Gateway : 192.168.1.1
TCP echo server started @ port 7

-----lwIP TCP echo server -----
TCP packets sent to port 6001 will be echoed back
WARNING: Not a Marvell or TI Ethernet PHY. Please verify the initialization sequence
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 1: 1000
Board IP: 192.168.1.68
Netmask : 255.255.255.0
Gateway : 192.168.1.1
TCP echo server started @ port 7
```

13.4 实验总结

通过实验我们更加深刻了解到 SDK 程序的开发，通过简单修改例程已经不能满足需求，有时候还要修改库文件。

第十四章 自定义 IP 实验

实验 Vivado 工程为 “custom_pwm_ip”。

Xilinx 官方为大家提供了很多 IP 核，在 Vivado 的 IP Catalog 中可以查看这些 IP 核，用户在构建自己的系统中，不可能只使用 Xilinx 官方的免费 IP 核，很多时候需要创建属于自己的用户 IP 核，创建自己的 IP 核有很多好处，例如系统设计定制化、设计复用，可以在 IP 核中加入 license，有偿提供给别人使用；简化系统设计和缩短设计时间。用 ZYNQ 系统设计 IP 核，最常用的就是使用 AXI 总线将 PS 同 PL 部分的 IP 核连接起来。本实验将为大家介绍如何在 Vivado 中构建 AXI 总线类型的 IP 核，此 IP 核用来产生一个 PWM，用这个控制开发板上的 LED，做一个呼吸灯的效果。

14.1 PWM 介绍

我们经常使用 PWM 来控制 LED，蜂鸣器等，通过调节脉冲的占空比来调节 LED 的亮度。

在其他开发板中我们使用过的一个 pwm 模块如下：

```
////////////////////////////////////////////////////////////////
//                                                       //
// Author: meisq                                         //
//         msq@qq.com                                     //
//         ALINX(shanghai) Technology Co.,Ltd           //
// hejin                                              //
// WEB: http://www.alinx.cn/                           //
// BBS: http://www.hejin.org/                          //
//                                                       //
// Copyright (c) 2017,ALINX(shanghai) Technology Co.,Ltd   //
// All rights reserved                                  //
//                                                       //
// This source file may be used and distributed without restriction provided   //
// that this copyright statement is not removed from the file and that any      //
// derivative work contains the original copyright notice and the associated    //
// disclaimer.                                         //
//                                                       //
//=====
// Description: pwm model
//   pwm out period = frequency(pwm_out) * (2 ** N) / frequency(clk);
//
//=====
// Revision History:
// Date       By        Revision     Change Description
//-----
// 2017/5/3   meisq     1.0        Original
//*****`timescale 1ns / 1ps
module ax_pwm
#(
  parameter N = 32 //pwm bit width
)
(
  input      clk,
  input      rst,
  input[N - 1:0]period,
  input[N - 1:0]duty,
  output     pwm_out
);

reg[N - 1:0] period_r;
reg[N - 1:0] duty_r;
reg[N - 1:0] period_cnt;
reg pwm_r;
assign pwm_out = pwm_r;
always@(posedge clk or posedge rst)
begin
  if(rst==1)
```

```

begin
    period_r <= { N {1'b0} };
    duty_r <= { N {1'b0} };
end
else
begin
    period_r <= period;
    duty_r <= duty;
end
end

always@(posedge clk or posedge rst)
begin
    if(rst==1)
        period_cnt <= { N {1'b0} };
    else
        period_cnt <= period_cnt + period_r;
end

always@(posedge clk or posedge rst)
begin
    if(rst==1)
        begin
            pwm_r <= 1'b0;
        end
    else
        begin
            if(period_cnt >= duty_r)
                pwm_r <= 1'b1;
            else
                pwm_r <= 1'b0;
        end
    end
endmodule

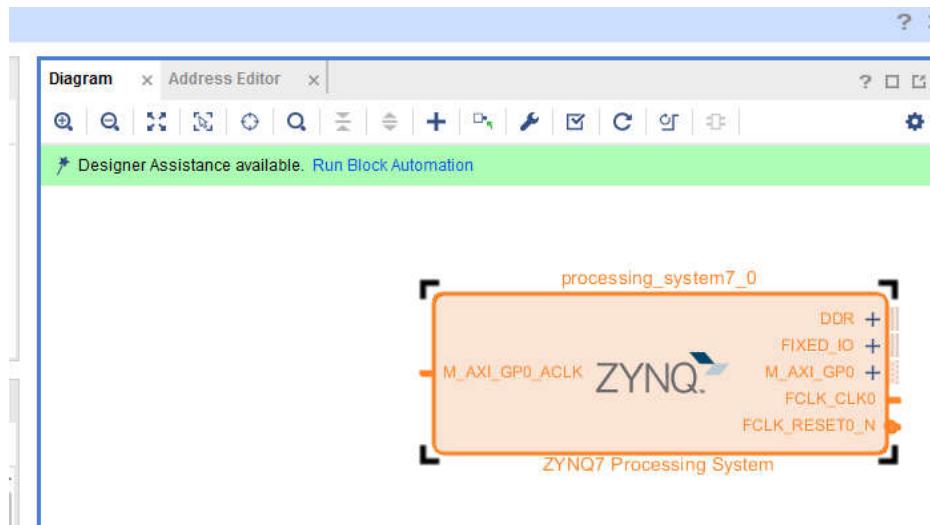
```

可以看到这个 PWM 模块需要 2 个参数 “period”、“duty” 来控制频率和占空比，我们需要设计一些寄存器来控制这些参数，这里需要使用 AXI 总线，PS 通过 AXI 总线来读写寄存器。

14.2 Vivado 工程建立

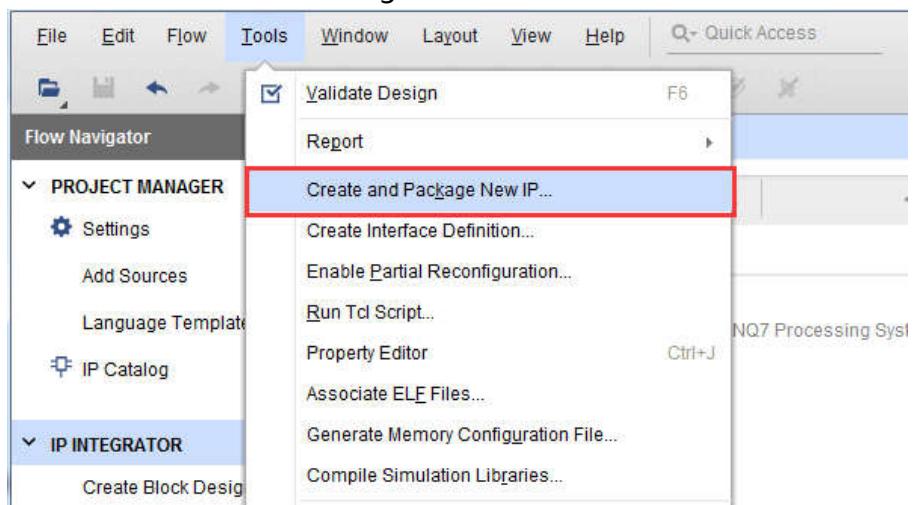
14.2.1 创建一个 vivado 工程

创建一个名为 “custom_pwm_ip” 工程，添加 zynq PS 系统，并配置参数，具体方法可以参考前面方法

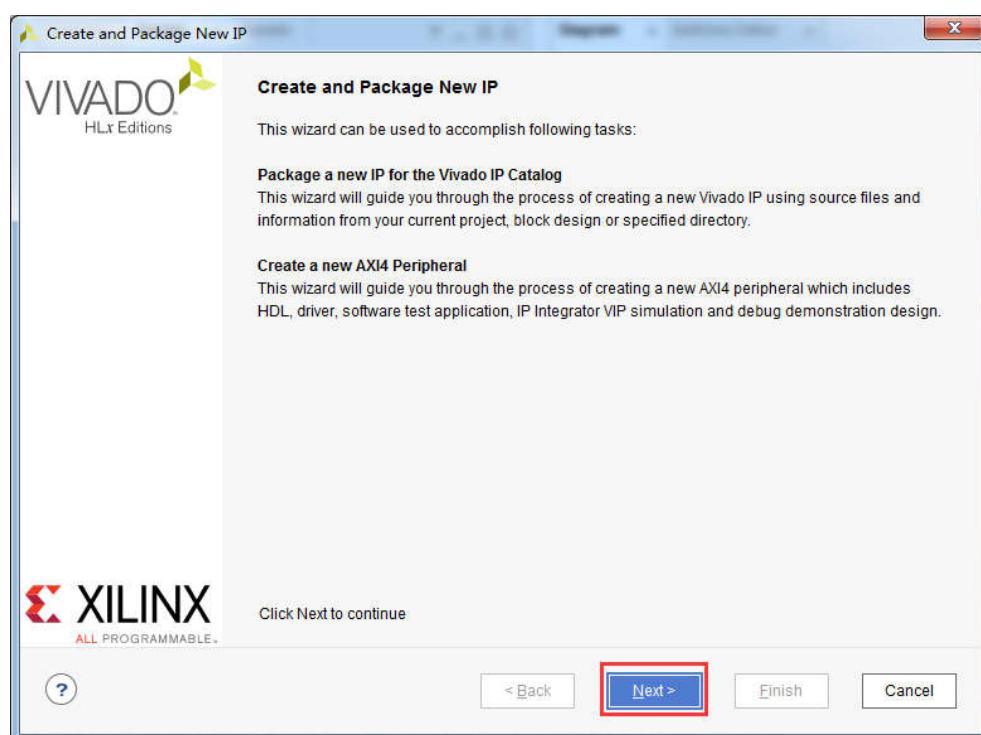


14.2.2 创建自定义 IP

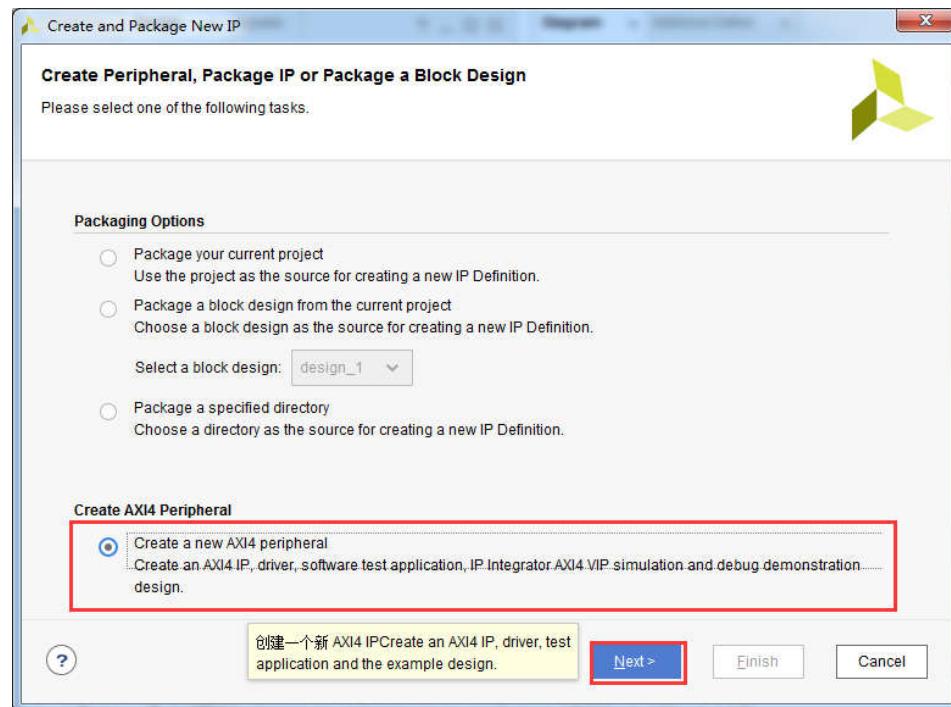
1) 点击菜单 “Tools->Create and Package IP...”



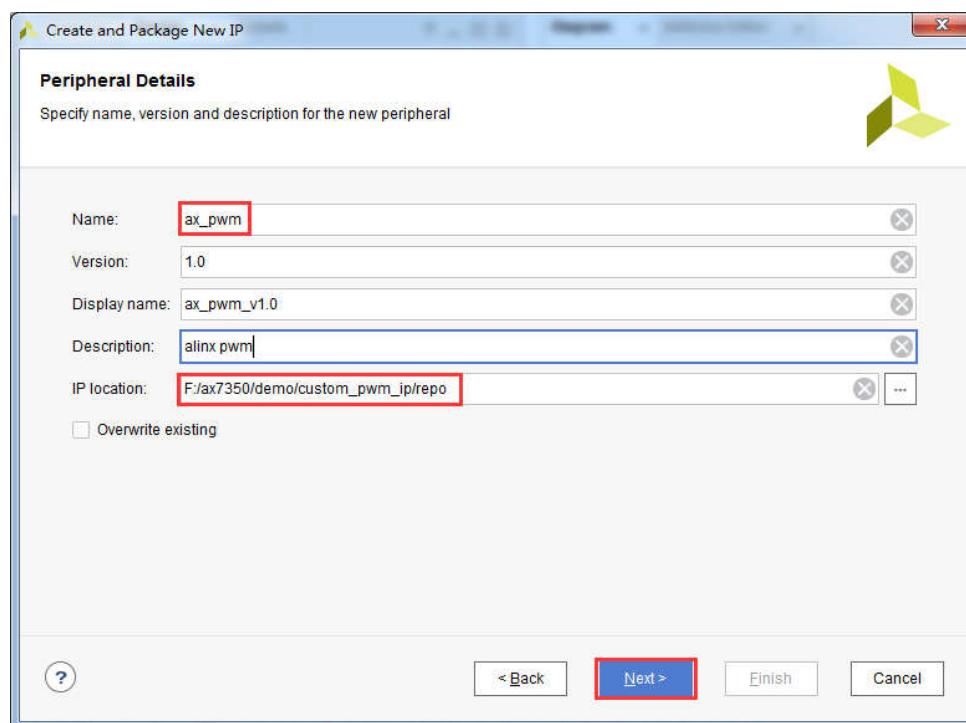
2) 选择 “Next”



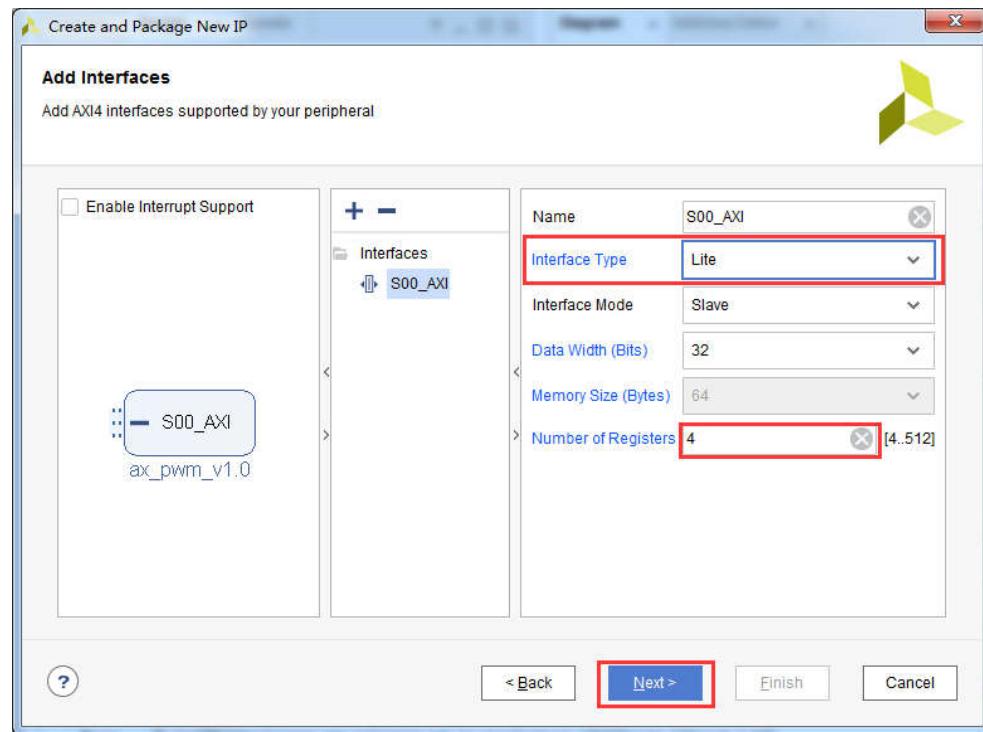
3) 选择创建一个新的 AXI4 设备



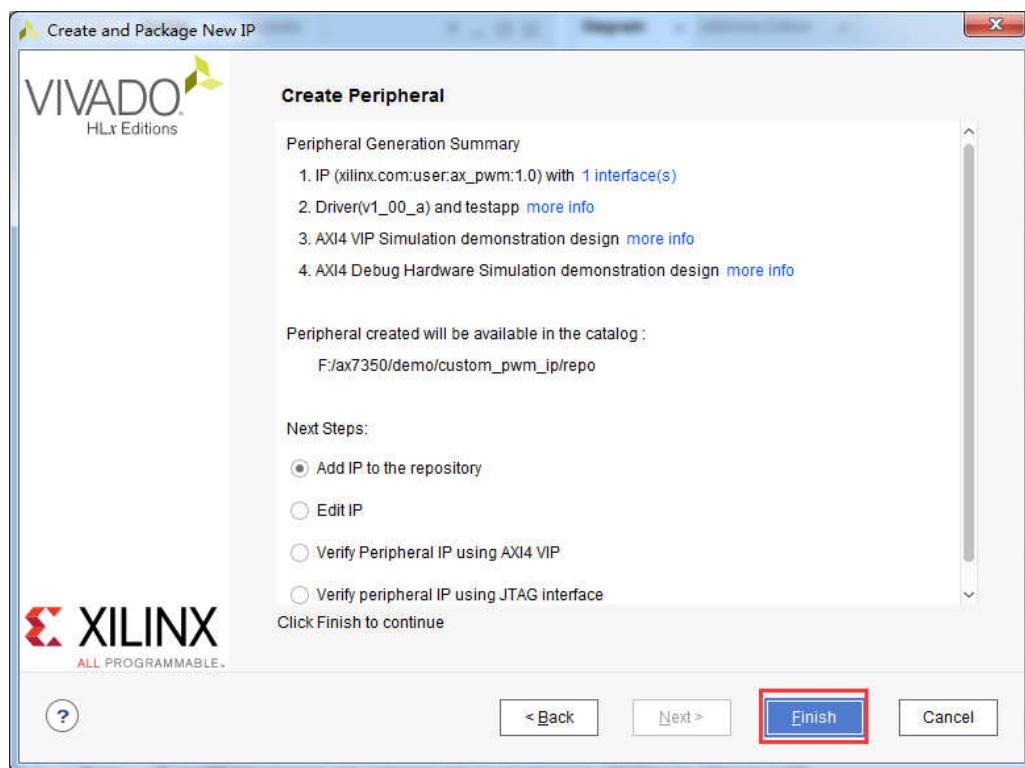
- 4) 名称填写 “ax_pwm” ,描述填写 “alinx pwm” ,然后选择一个合适的位置用来放 IP



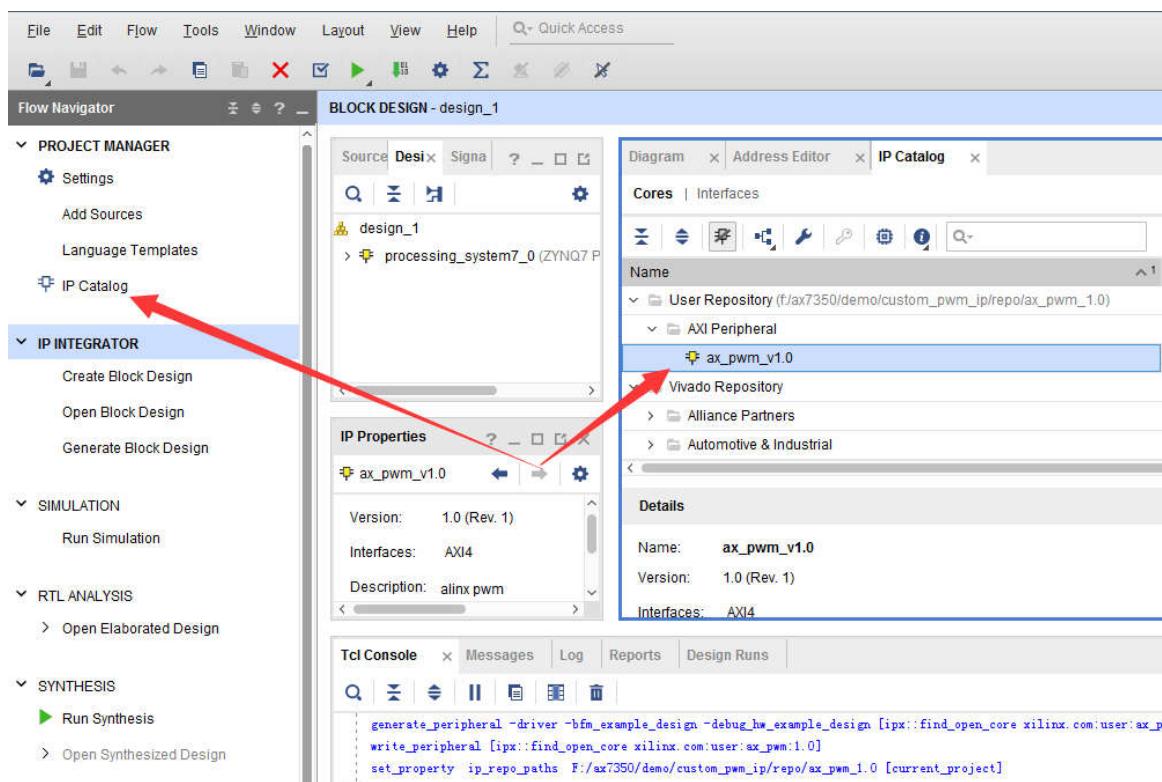
- 5) 下面参数可以指定接口类型、寄存器数量等，这里不需要修改，使用 AXI Lite Slave 接口，4 个寄存器。



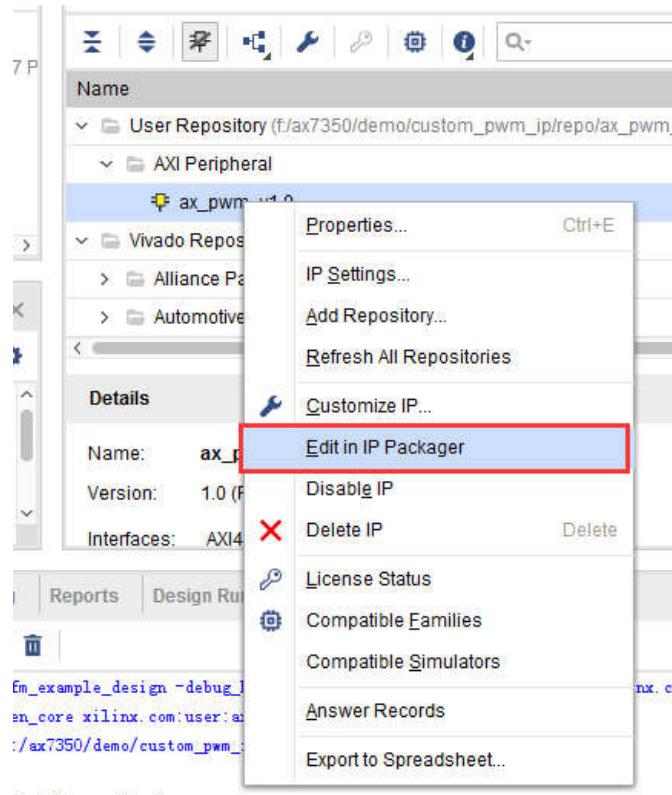
6) 点击“Finish”完成IP的创建



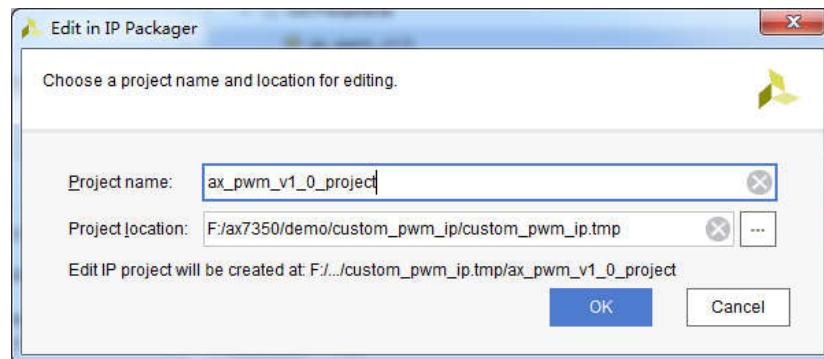
7) 在“IP Catalog”中可以看到刚才创建的IP



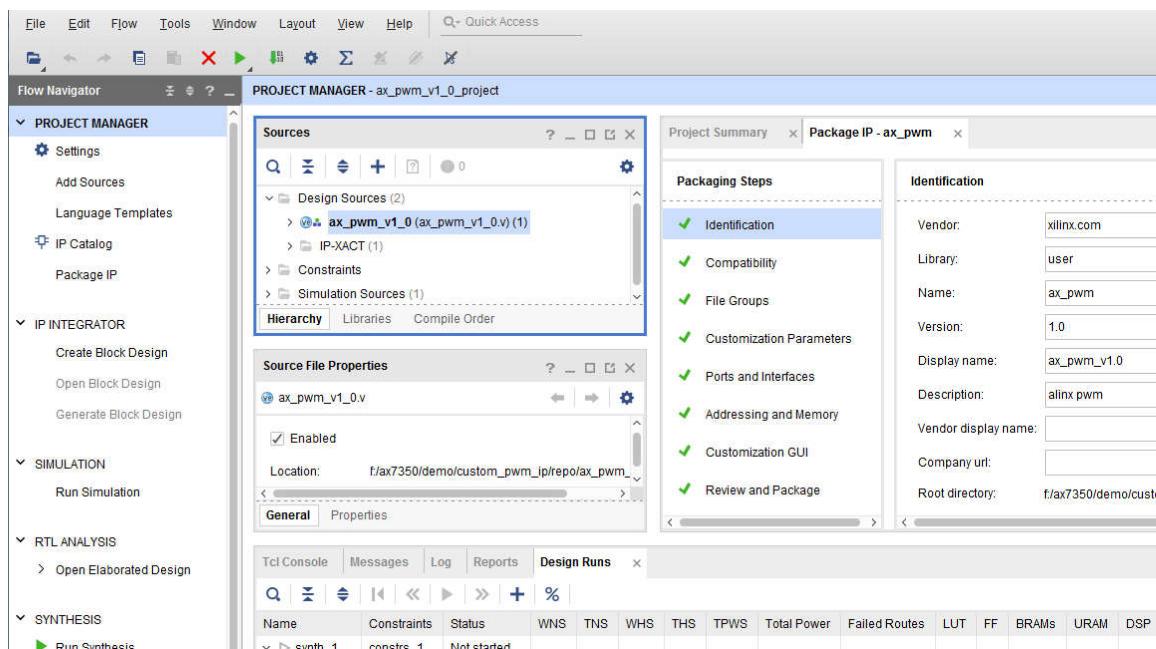
- 8) 这个时候的 IP 只有简单的寄存器读写功能，我们需要修改 IP，选择 IP，右键 “Edit in IP Packager”



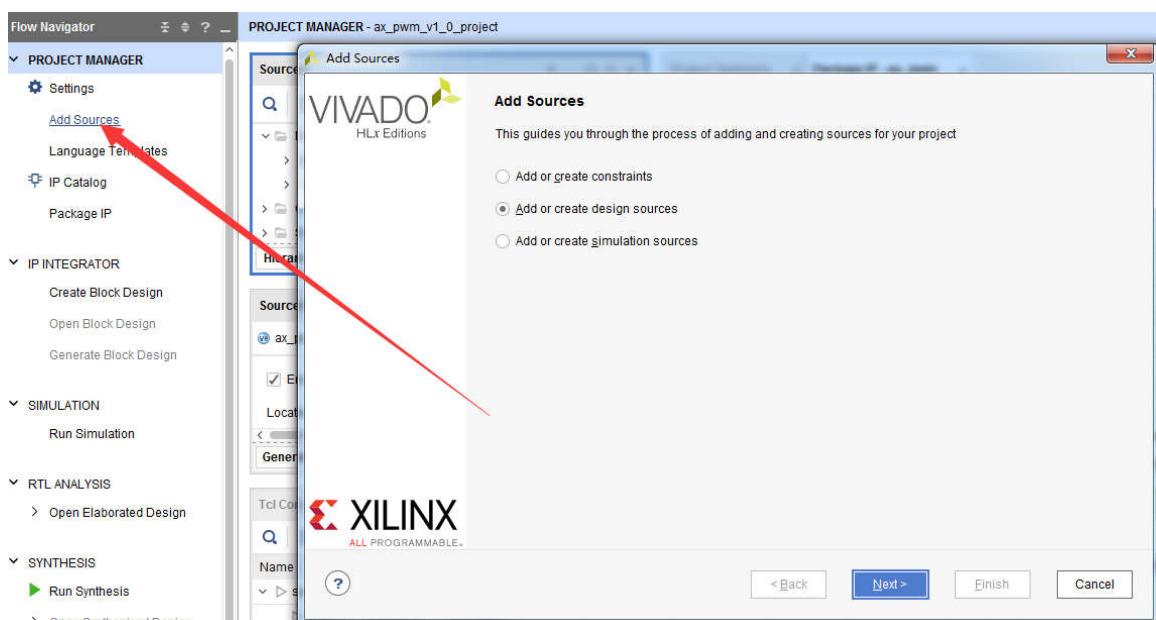
- 9) 这是弹出一个对话框，可以填写工程名称和路径，这里默认，点击 “OK”



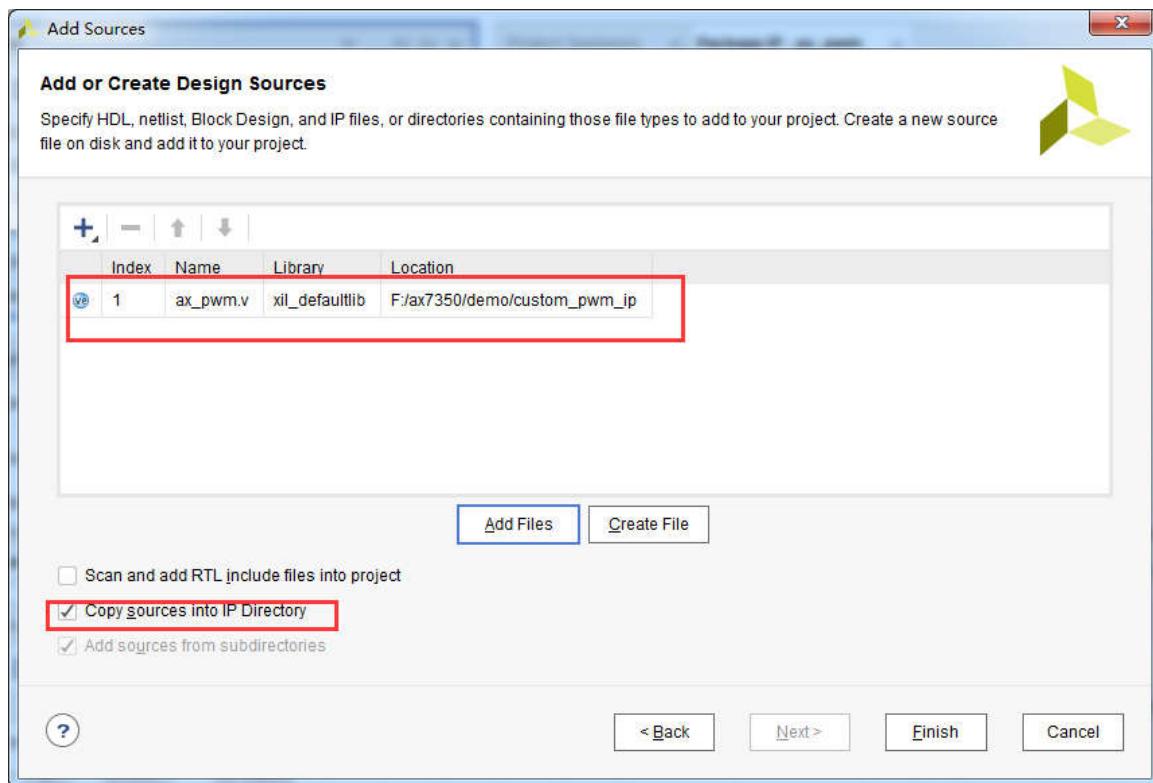
10) Vivado 打开了一个新的工程



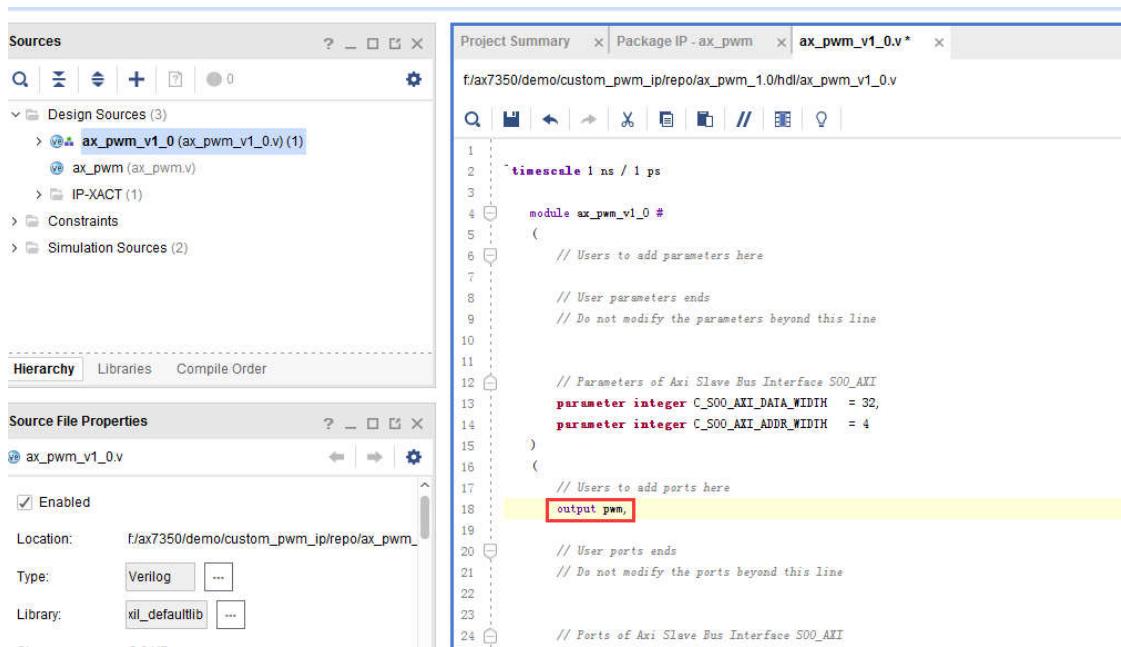
11) 添加 PWM 功能的核心代码



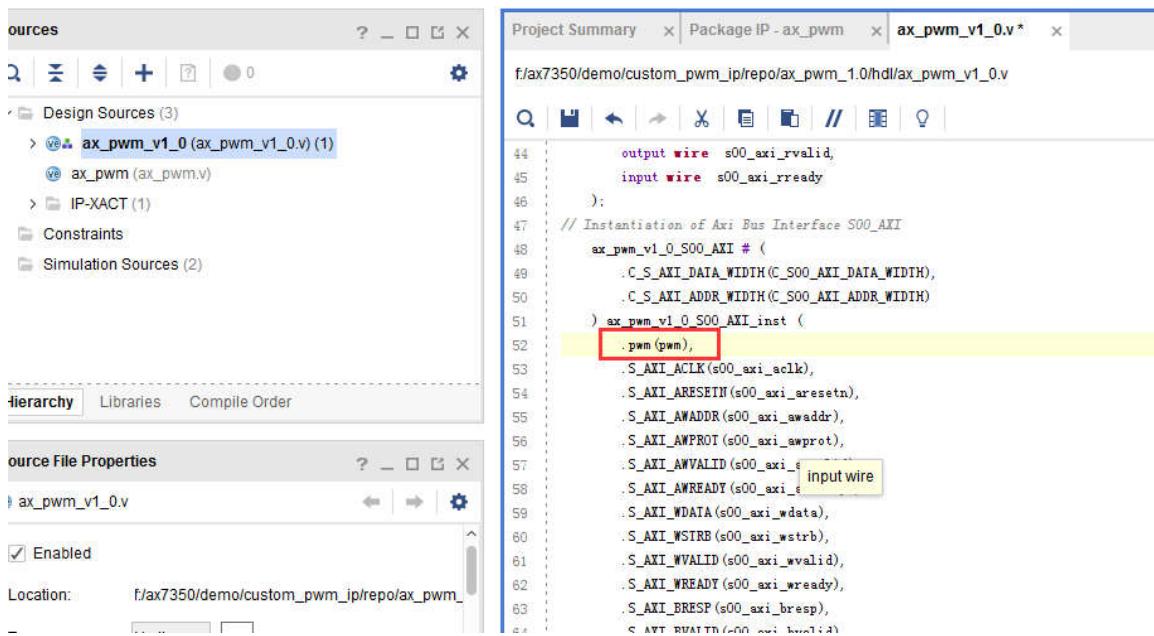
12) 添加代码时选择复制代码到 IP 目录



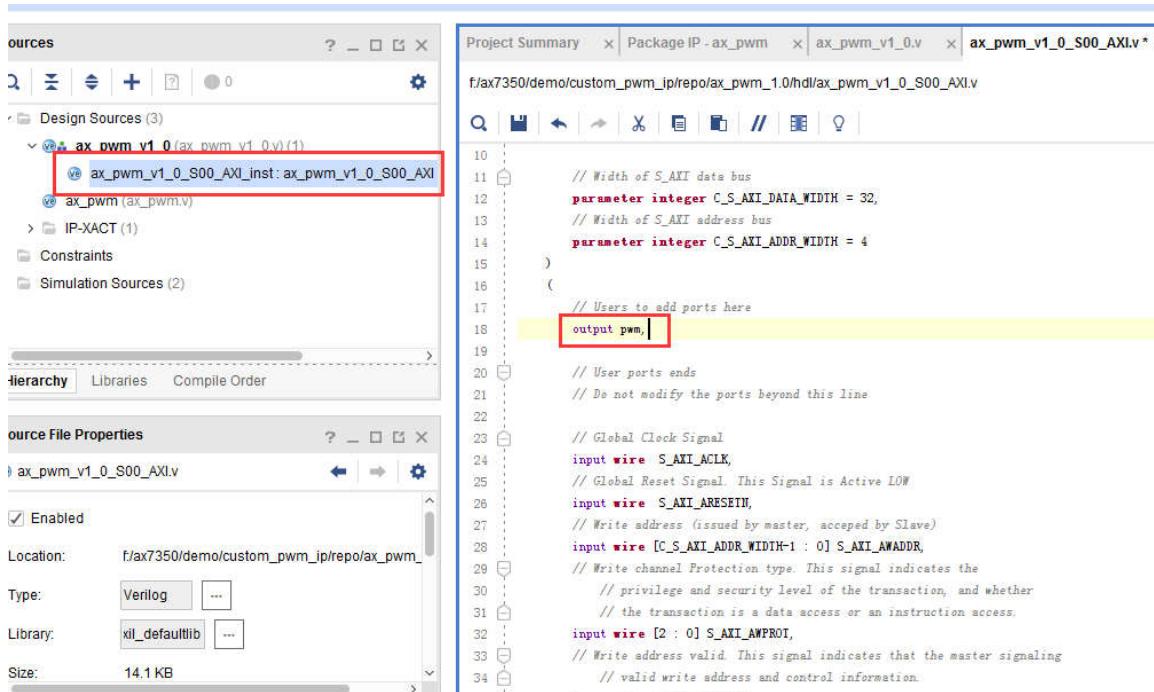
13) 修改 “ax_pwm_v1_0.v” , 添加一个 pwm 输出端口



14) 修改 “ax_pwm_v1_0.v” , 在例化 “ax_pwm_V1_0_S00_AXI” , 中添加 pwm 端口的例化



- 15) 修改 “ax_pwm_v1_0_s00_AXI.v” 文件，添加 pwm 端口，这个文件是实现 AXI4 Lite Slave 的核心代码



- 16) 修改“ax_pwm_v1_0_s00_AXI.v”文件 ,例化 pwm 核心功能代码 将寄存器 slv_reg0 和 slv_reg1 用于 pwm 模块的参数控制。

```

PROJECT MANAGER - ax_pwm_v1_0_project
Sources
Design Sources (3)
  ax_pwm_v1_0 (ax_pwm_v1_0.v) (1)
    ax_pwm_v1_0_S00_AXI_inst: ax_pwm_v1_0_S00_AXI
      ax_pwm (ax_pwm.v)
      IP-XACT (1)
  Constraints
  Simulation Sources (2)

Hierarchy Libraries Compile Order

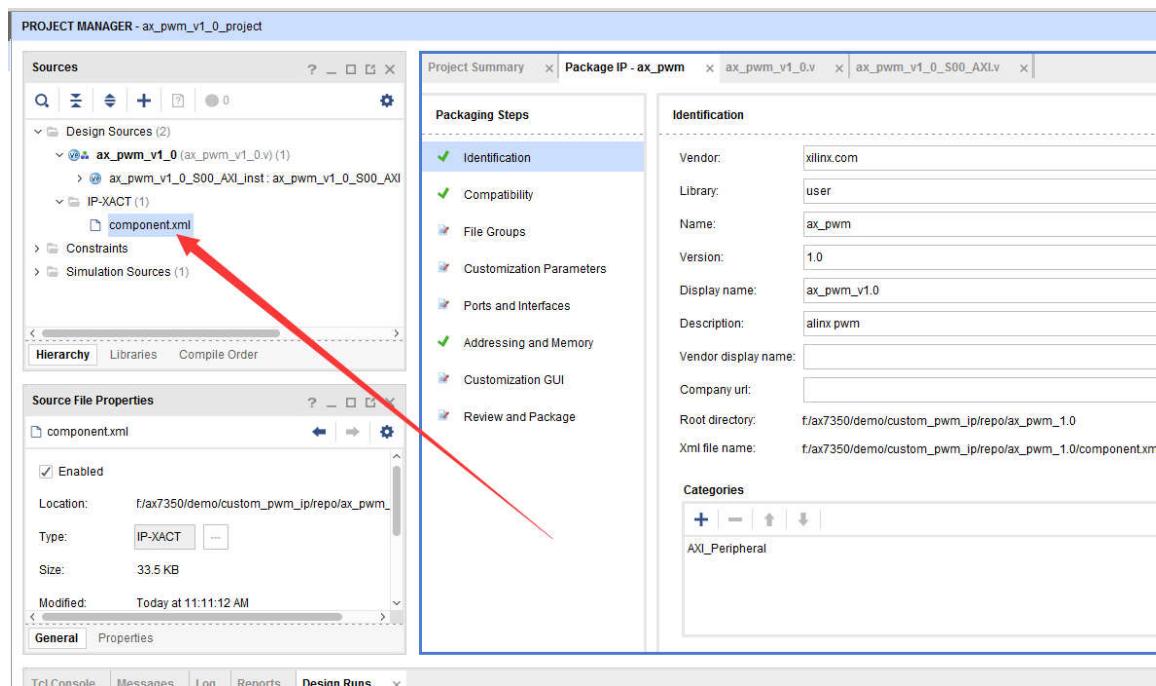
Source File Properties
  ax_pwm_v1_0_S00_AXI.v
    Enabled: checked
    Location: f:/ax7350/demo/custom_pwm_ip/repo/ax_pwm...
    Type: Verilog
    Library: xil_defaultlib
    Size: 14.1 KB

Project Summary x Package IP - ax_pwm x ax_pwm_v1_0.v x ax_pwm_v1_0_S00_AXI.v *

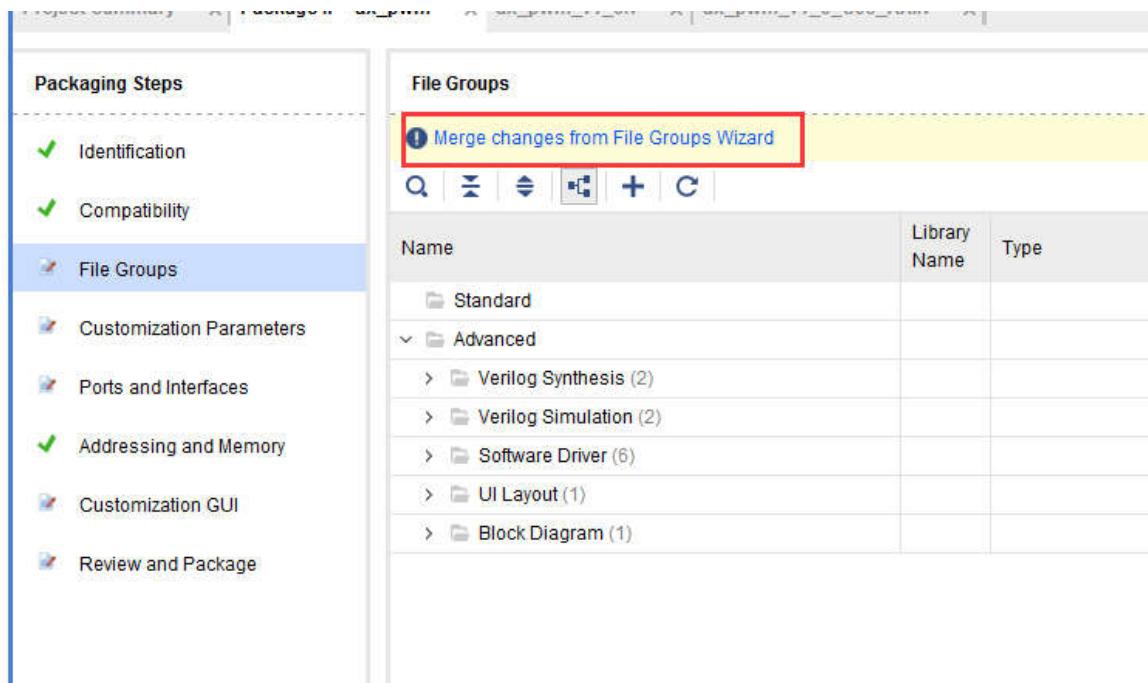
f:/ax7350/demo/custom_pwm_ip/repo/ax_pwm_1.0/hdl/ax_pwm_v1_0_S00_AXI.v
388 end
389 else
390 begin
391 // When there is a valid read address ($_AXI_ARVALID) with
392 // acceptance of read address by the slave ($xil_arready),
393 // output the read data
394 if ($lv_raddr)
395 begin
396   axi_rdata <= reg_data_out; // register read data
397 end
398 end
399 end
400
401 // Add user logic here
402 ax_pwm ax_pwm_m0
403 (
404   .clk(S_AXI_ACLK),
405   .rst(S_AXI_ARESETN),
406   .period(slv_reg0),
407   .duty(slv_reg1),
408   .pwm_out(pwm)
409 );
410
411 // User logic ends
412
413 endmodule

```

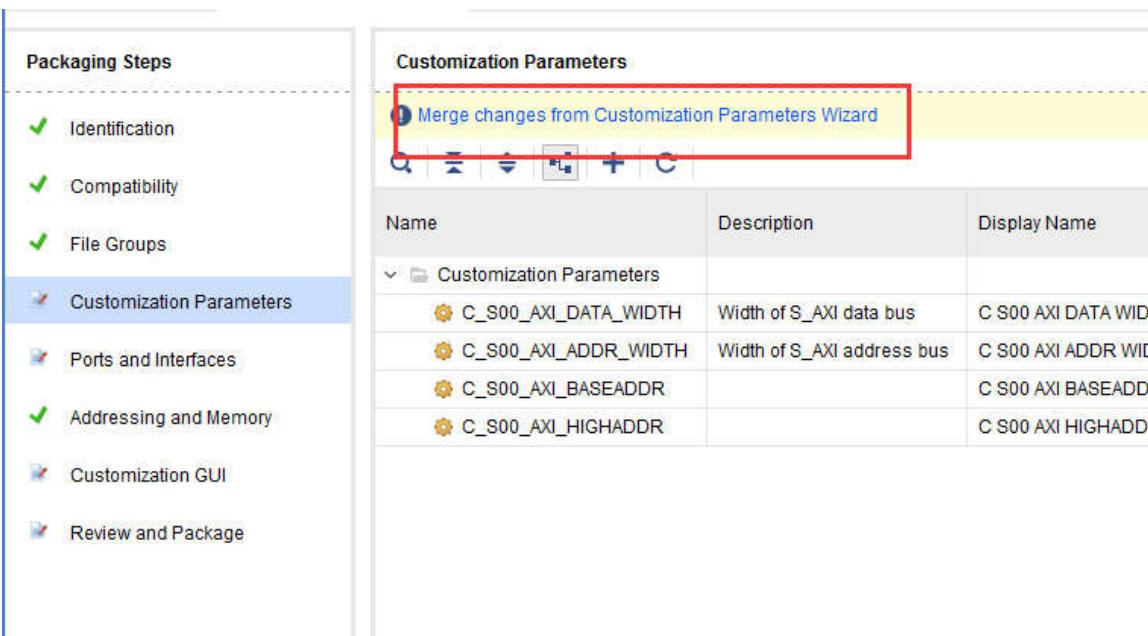
17) 双击“component.xml”文件



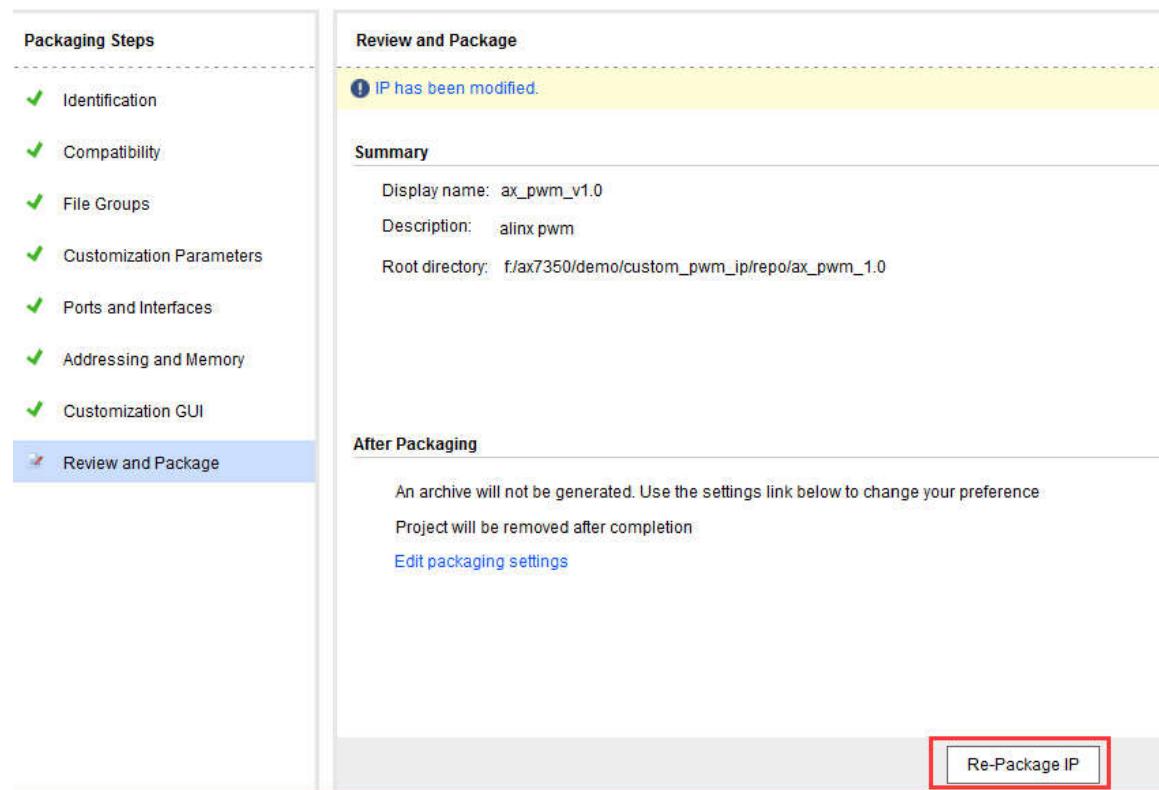
18) 在“File Groups”选项中点击“Merge changes from File Groups Wizard”



- 19) 在“Customization Parameters”选项中点击“Merge changes from Customization Parameters Wizard”

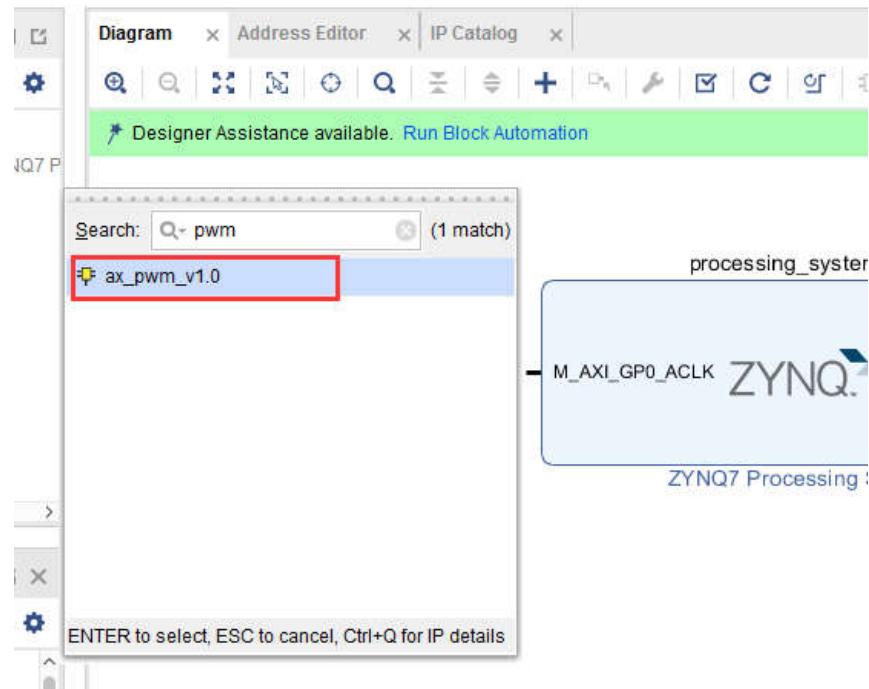


- 20) 点击“Re-Package IP”完成IP的修改

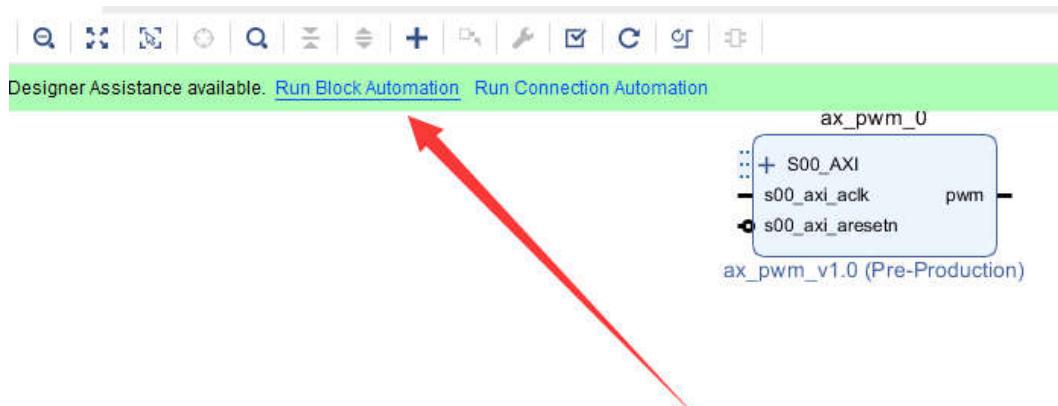


14.2.3 添加自定义 IP 到工程

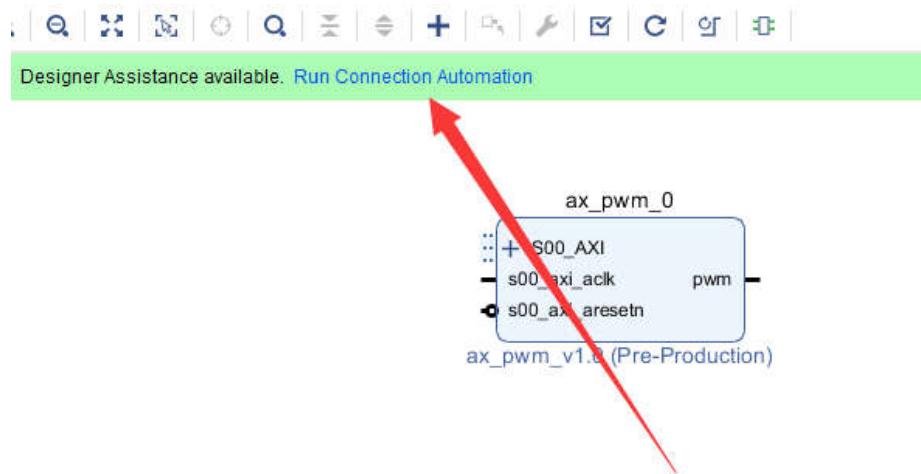
- 1) 搜索 “pwm” , 添加 “ax_pwm_v1.0”



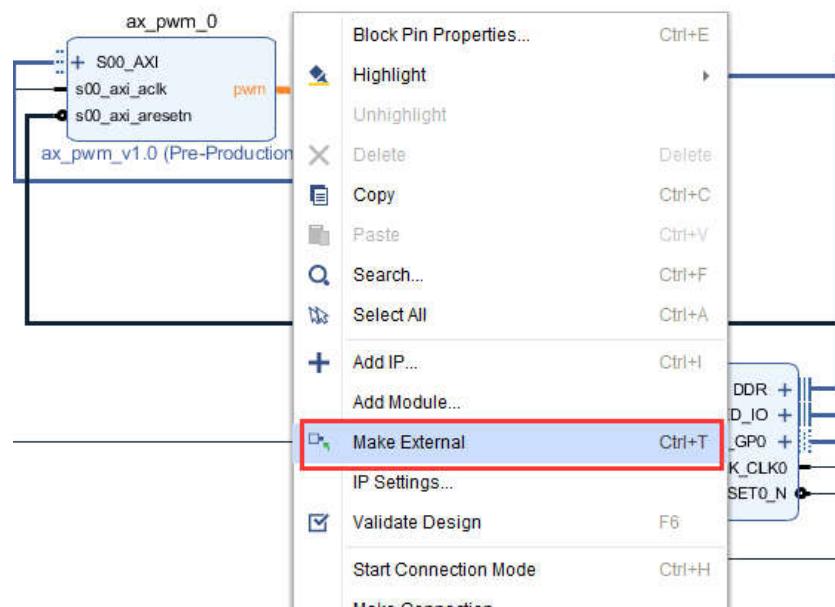
- 2) 点击 “Run Block Automation”

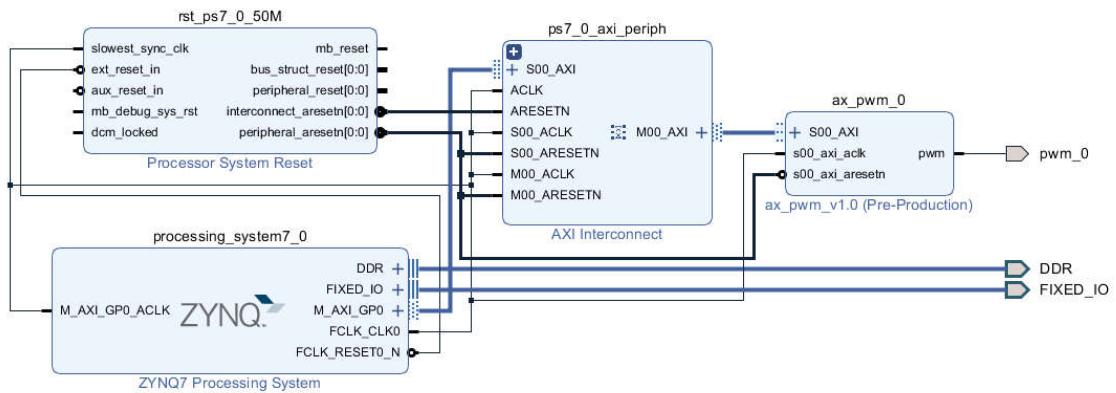


3) 点击 “Run Connection Automation”

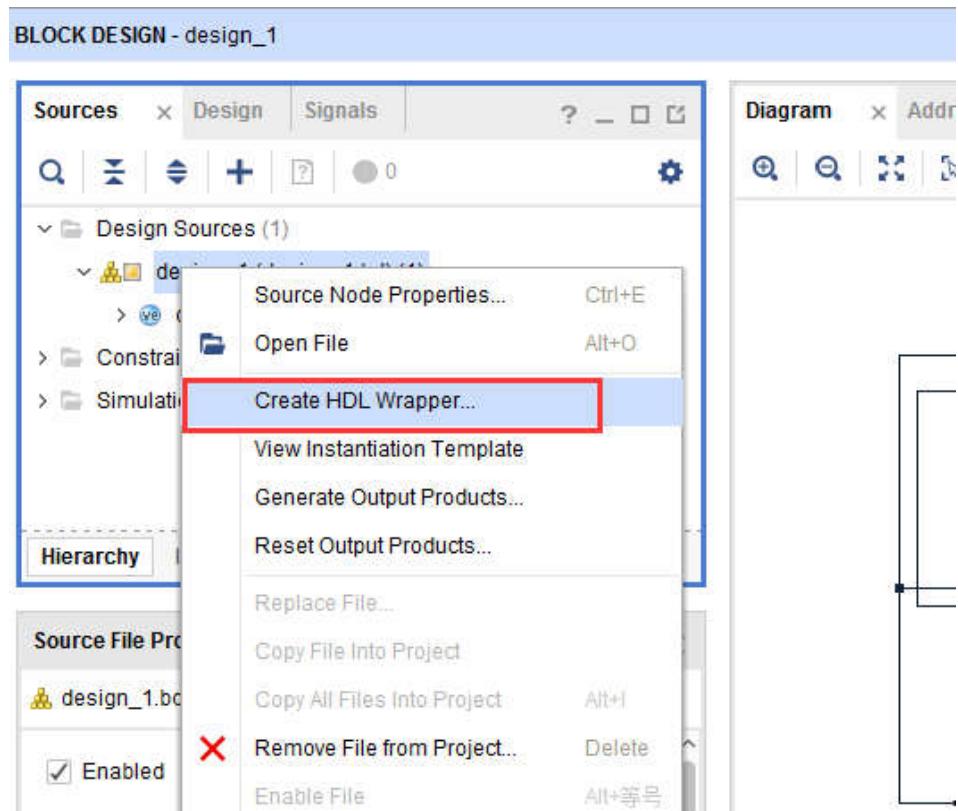


4) 导出 pwm 端口



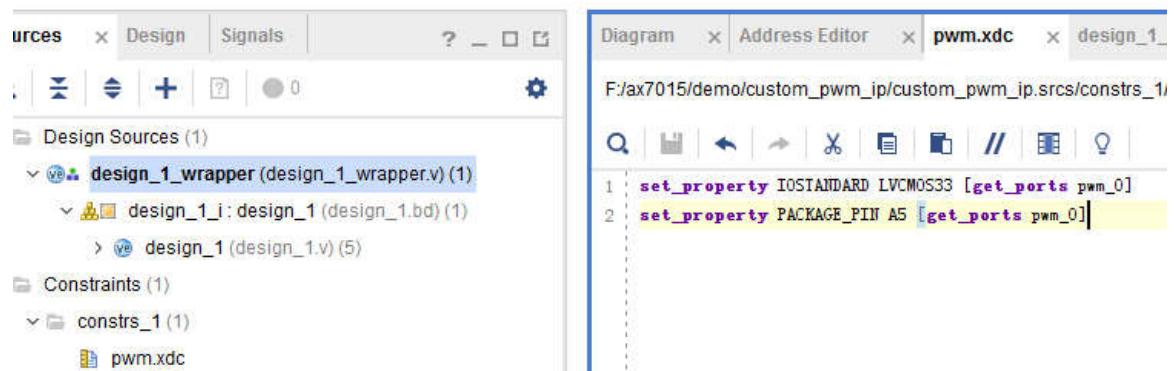


5) 创建 HDL 文件



6) 添加 xdc 文件分配管脚，把 pwm_0 输出端口分配给 PL_LED1，做一个呼吸灯

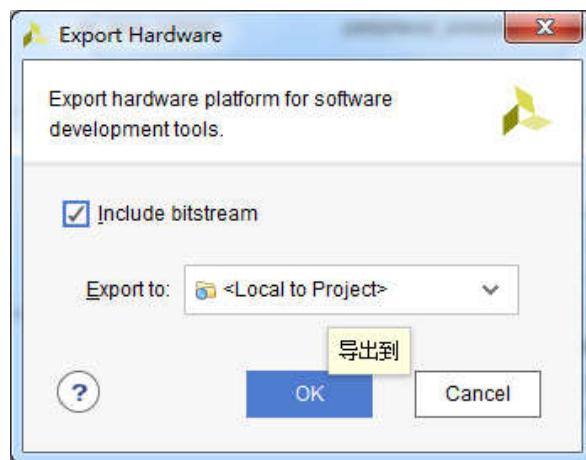
```
set_property IOSTANDARD LVCMOS33 [get_ports pwm_0]
set_property PACKAGE_PIN A5 [get_ports pwm_0]
```



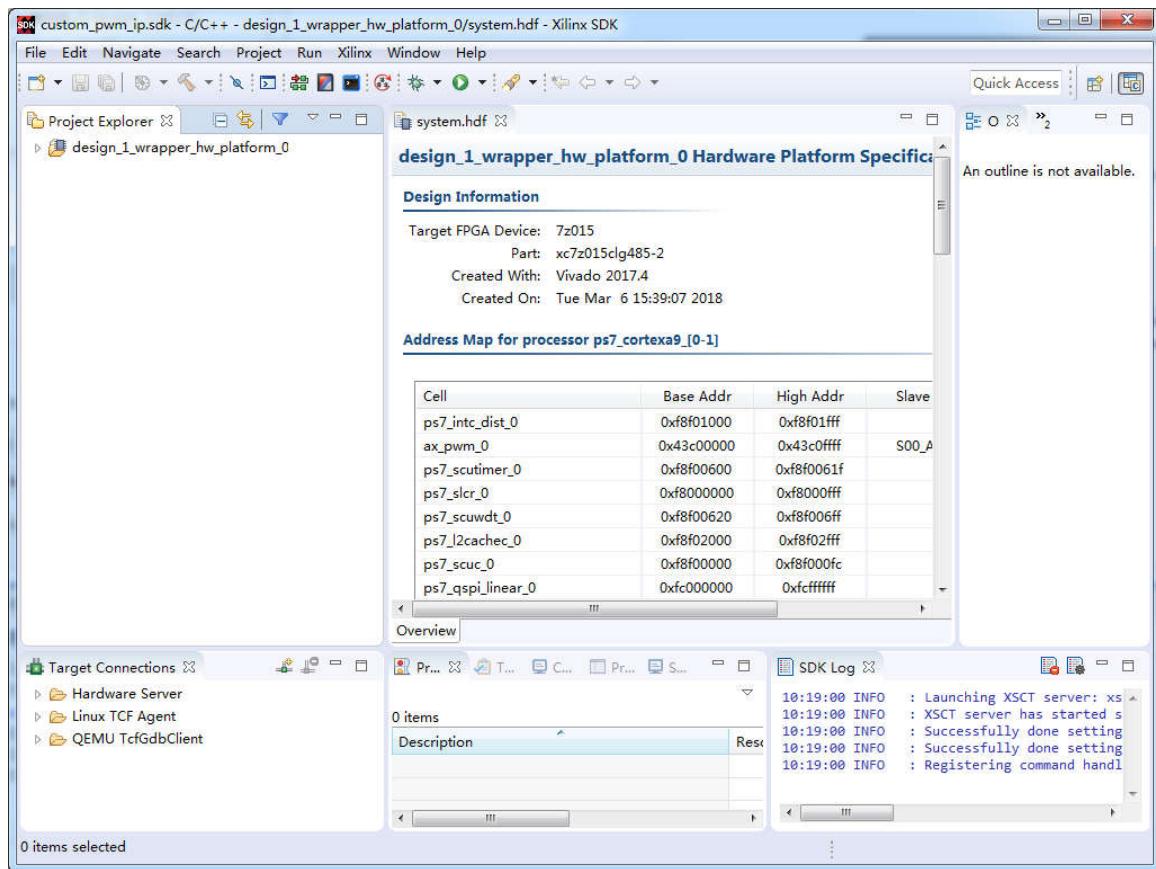
7) 编译生成 bit 文件

14.3 SDK 软件编写调试

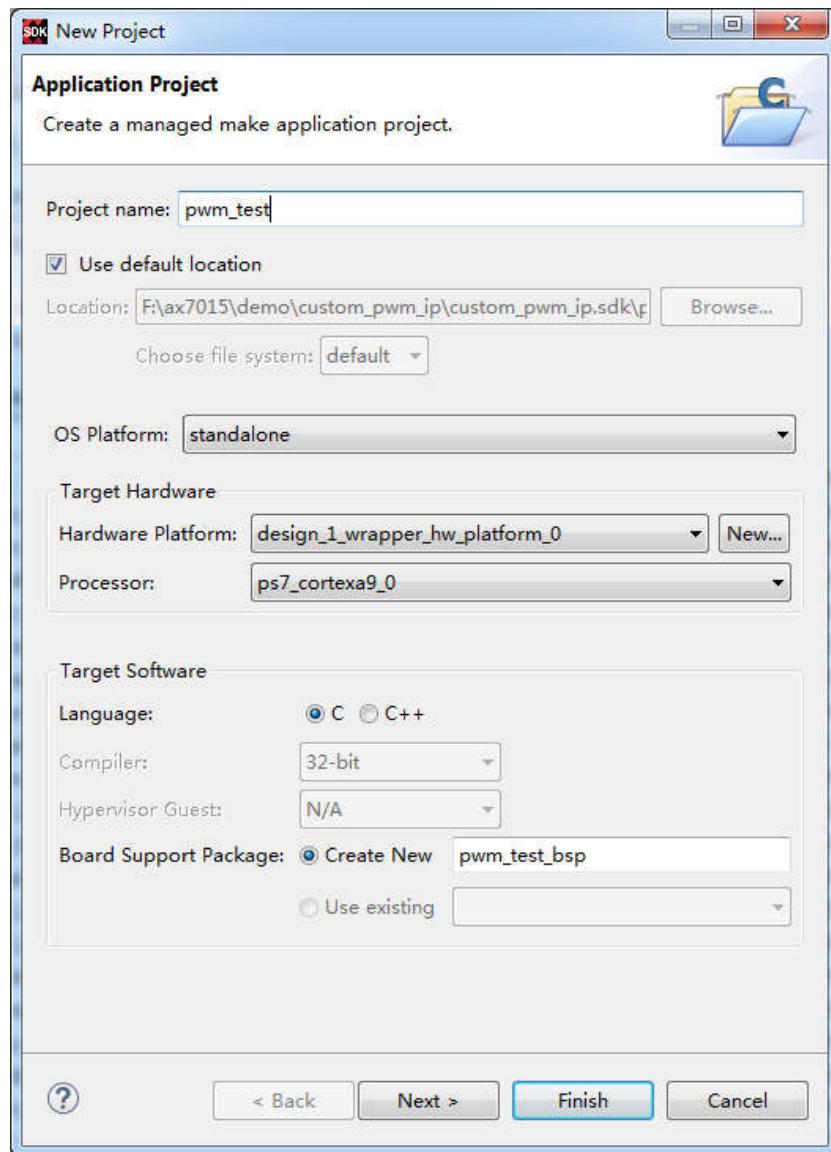
1) 导出硬件



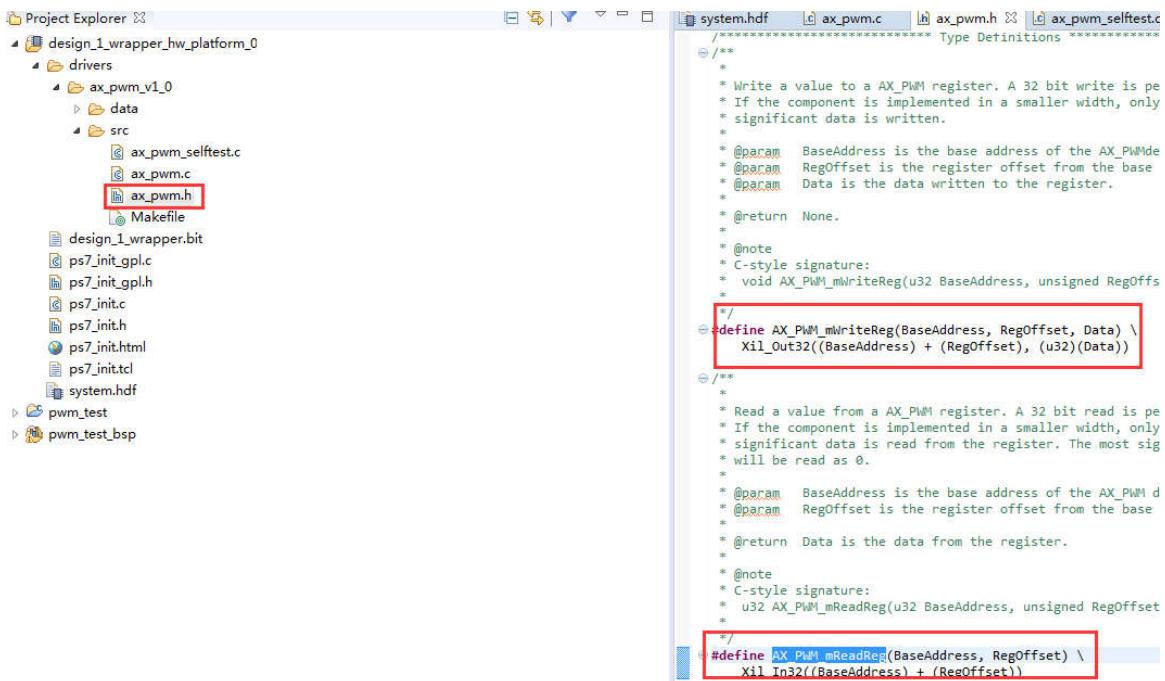
2) 启动 SDK



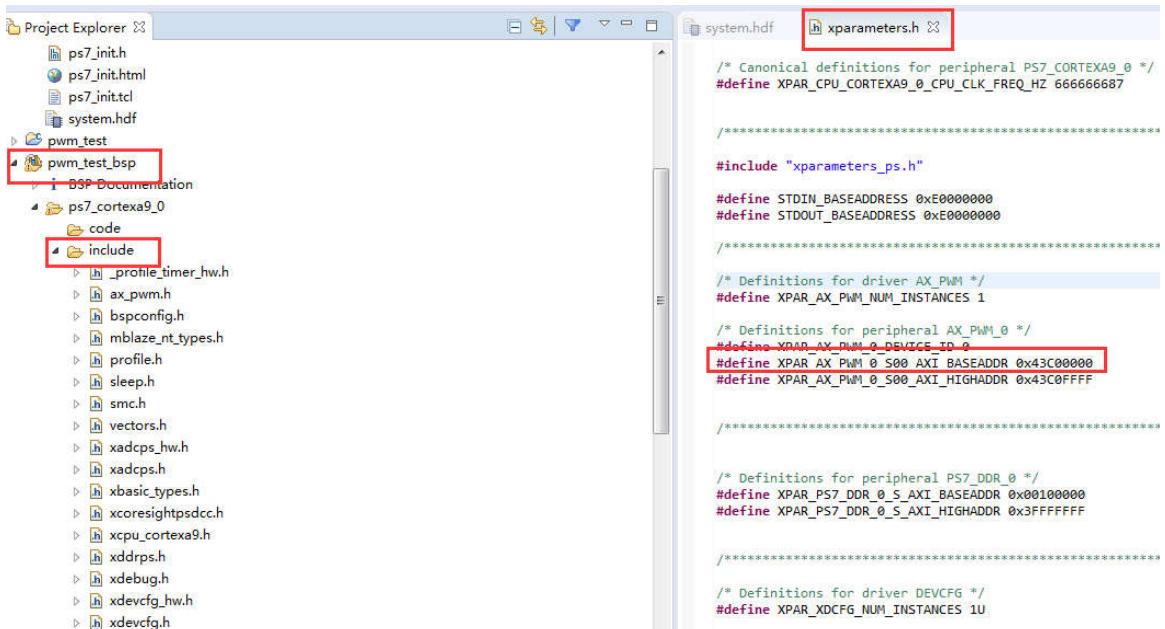
3) 新建 APP , 模板选择 “Hello World”



- 4) 前面的例都是使用 xilinx 的 IP , xilinx 大多都提供一套 API , 对于这个自定义 IP , 我们需要自己开发 , 先看看 APP 的目录下的资源 , 可以找到一个 ax_pwm.h 的文件 , 这个文件里包含里对自定义 IP 寄存器的读写宏定义



- 5) 在 bsp 里找到 “xparameters.h” 文件，这个非常重要的文件，里面找到了自定 IP 的寄存器基地址，可以找到自定义 IP 的基地址。



- 6) 有个寄存器读写宏和自定义 IP 的基地址，我们开始编写代码，测试自定义 IP，我们先通过写寄存器 AX_PWM_S00_AXI_SLV_REG0_OFFSET，控制 PWM 输出频率，然后通过写寄存器 AX_PWM_S00_AXI_SLV_REG1_OFFSET 控制 PWM 输出的占空比。

```

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "ax_pwm.h"
#include "xil_io.h"
#include "xparameters.h"
#include "sleep.h"

```

```

unsigned int duty;

int main()
{
    init_platform();

    print("Hello World\n\r");

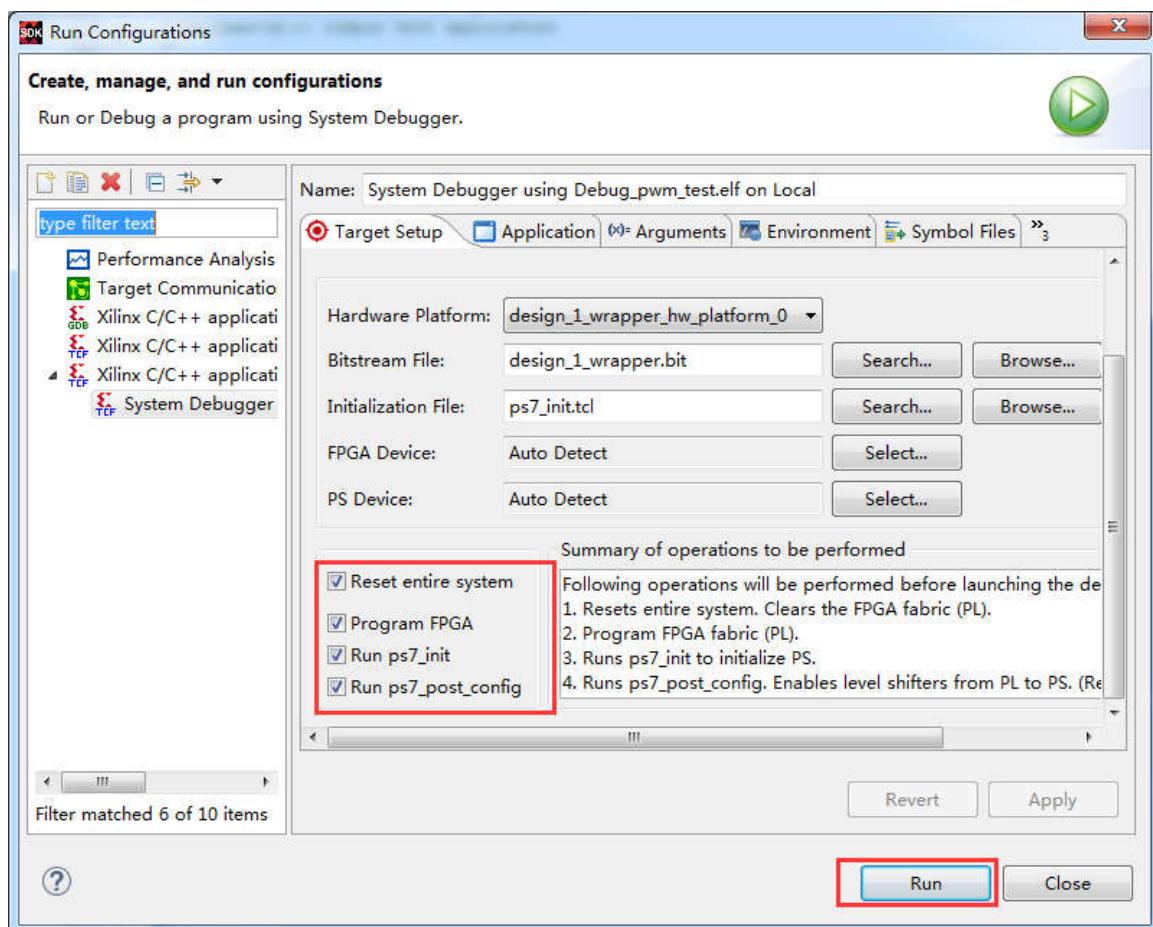
    //pwm_out period = frequency(pwm_out) * (2 ** N) / frequency(clk);
    AX_PWM_mWriteReg(XPAR_AX_PWM_0_S00_AXI_BASEADDR, AX_PWM_S00_AXI_SLV_REG0_OFFSET, 17179); //200hz

    while (1) {
        for (duty = 0; duty < 0xffffffff; duty = duty + 500) {
            AX_PWM_mWriteReg(XPAR_AX_PWM_0_S00_AXI_BASEADDR, AX_PWM_S00_AXI_SLV_REG1_OFFSET, duty);
            usleep(10000);
        }
    }

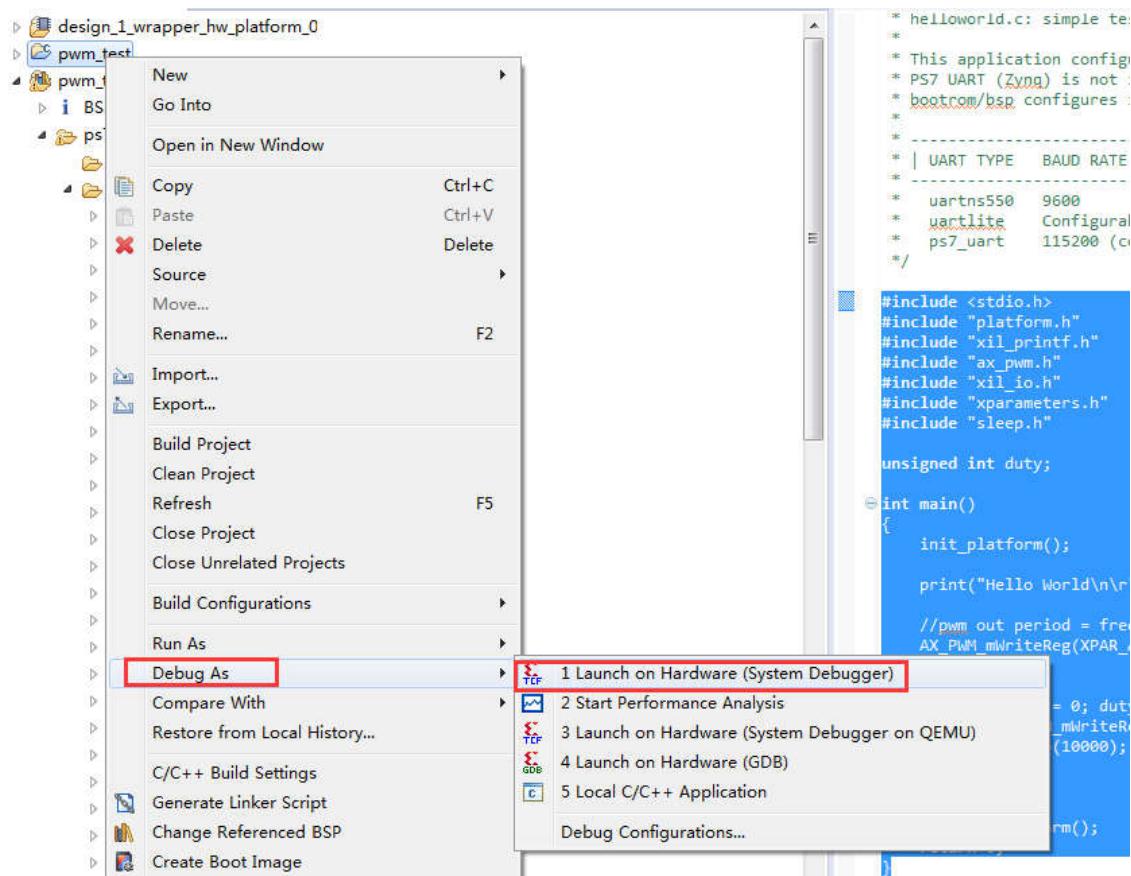
    cleanup_platform();
    return 0;
}

```

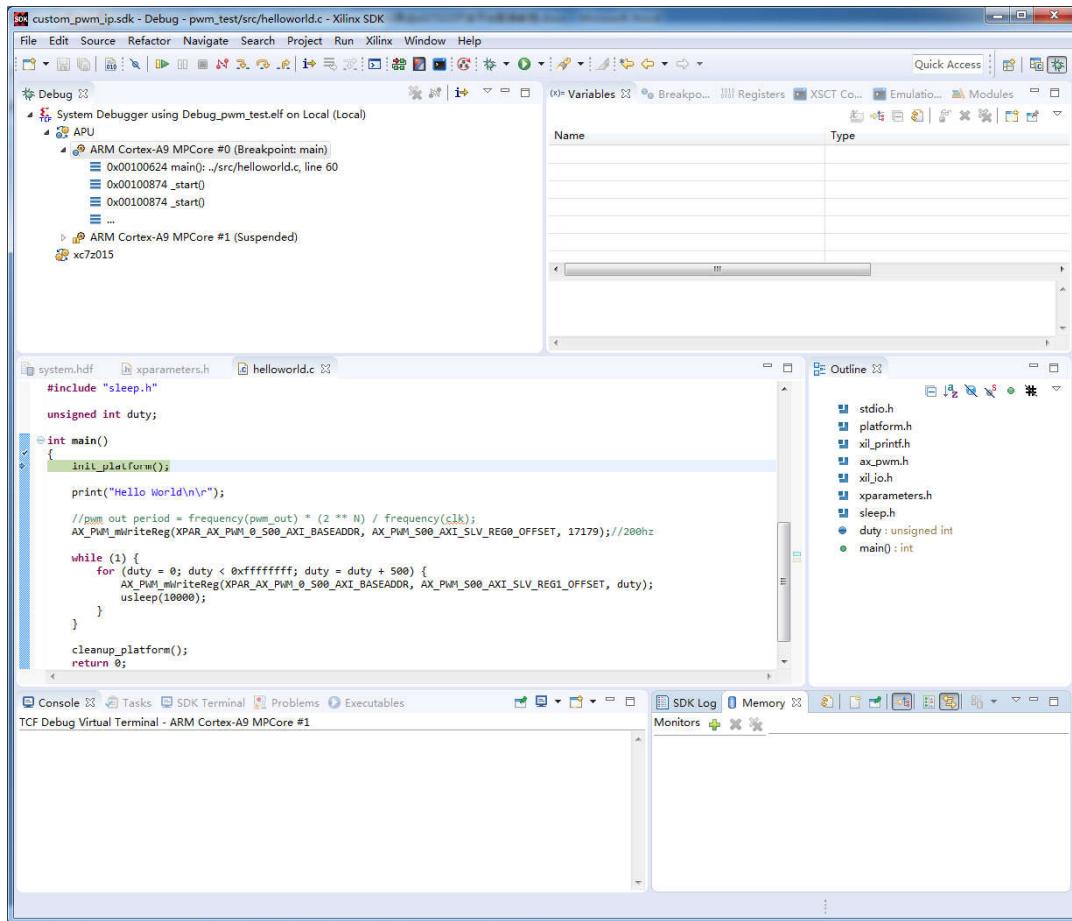
- 7) 通过运行代码，我们可以看到 PL_LED1 呈现出一个呼吸灯的效果。



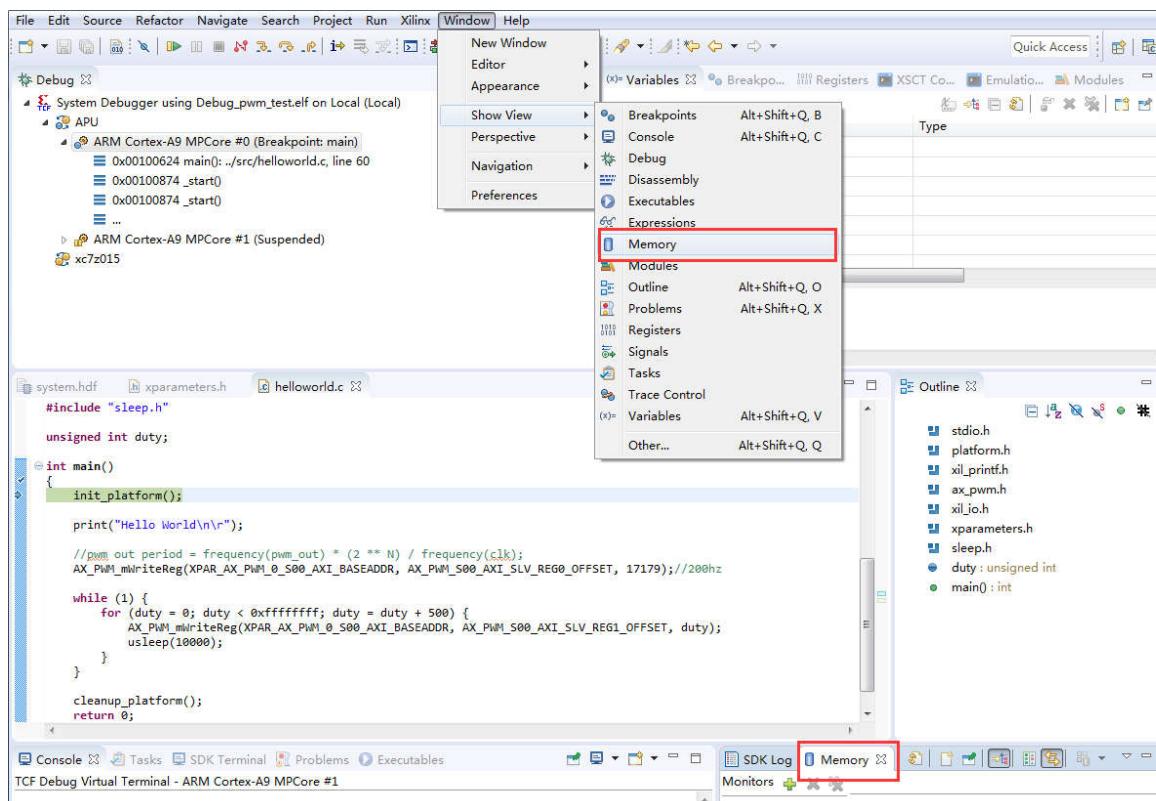
- 8) 通过 debug，我们来查看一下寄存器



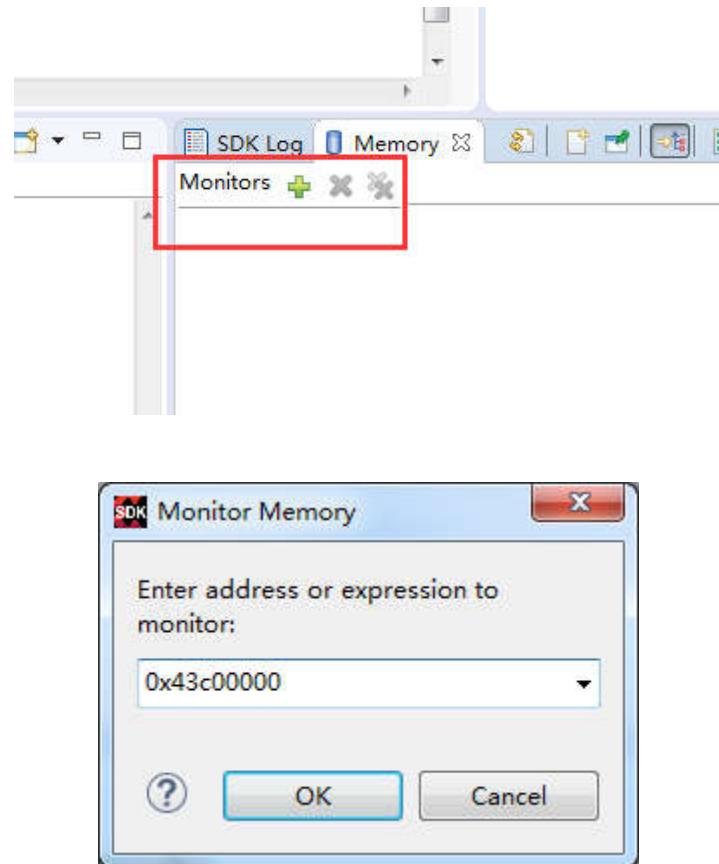
9) 进入 debug 状态，按 “F6” 可以单步运行。



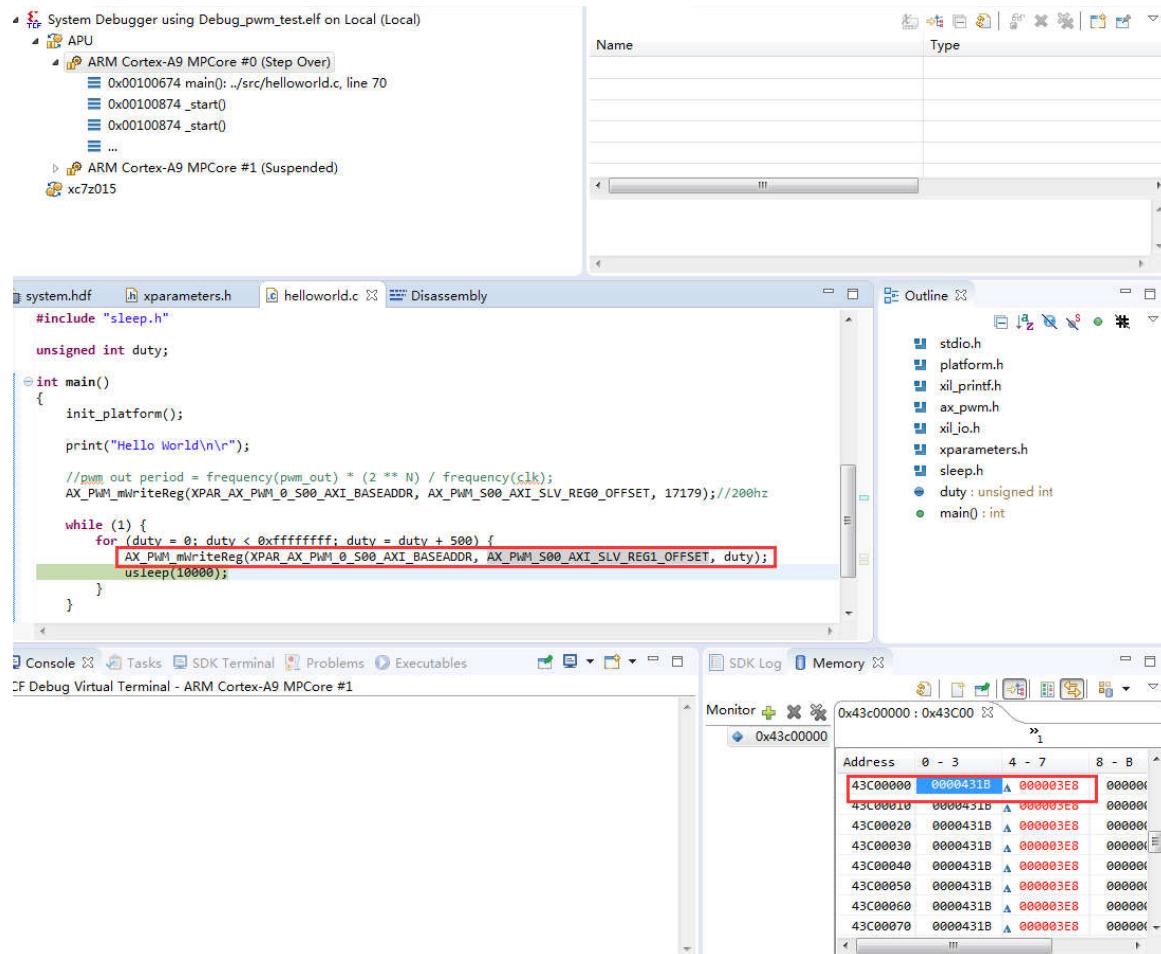
10) 通过菜单可以查看“Memory”窗口



11) 添加一个监视地址 “0x43c00000”



12) 单步运行，观察变化



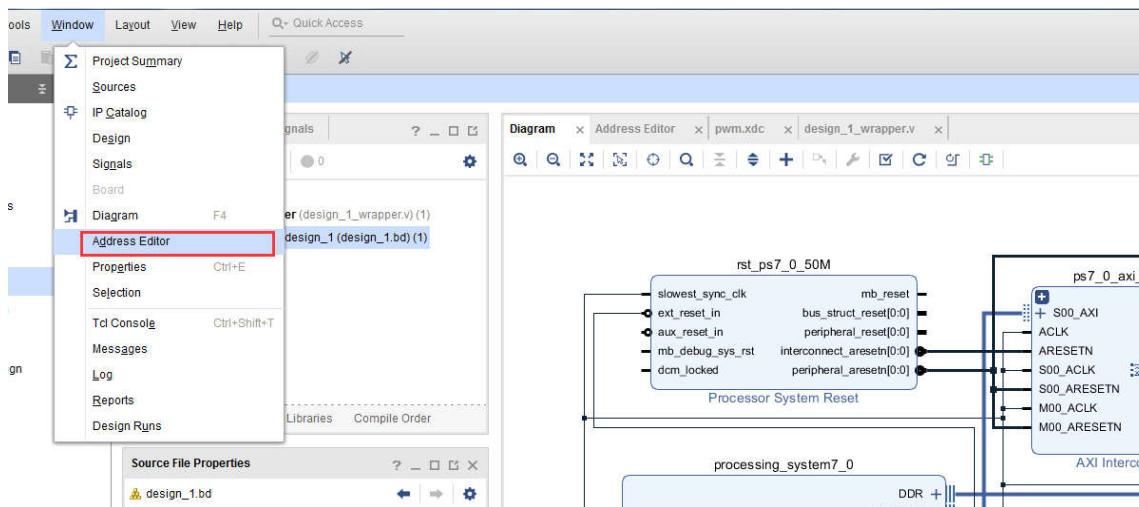
14.4 实验总结

通过本实验我们掌握了更多的 SDK 调试技巧，掌握了 ARM + FPGA 开发的核心内容，就是 ARM 和 FPGA 数据交互。

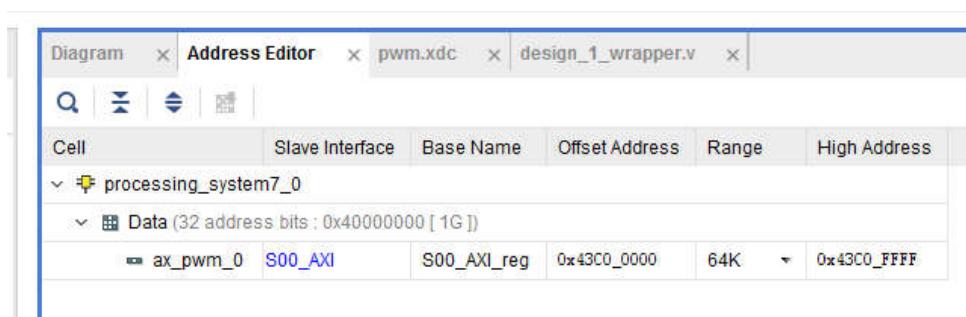
14.5 常见问题

14.5.1 如何知道 AXI IP 的基地址

- 1) 如下图所示，打开 “Address Editor”，可以看到地址分配情况



2) 地址一般是 Vivado 自动分配，我们也可以修改地址



第十五章 使用 VDMA 驱动 HDMI 显示

实验 Vivado 工程为 “vdma_hdmi_out”。

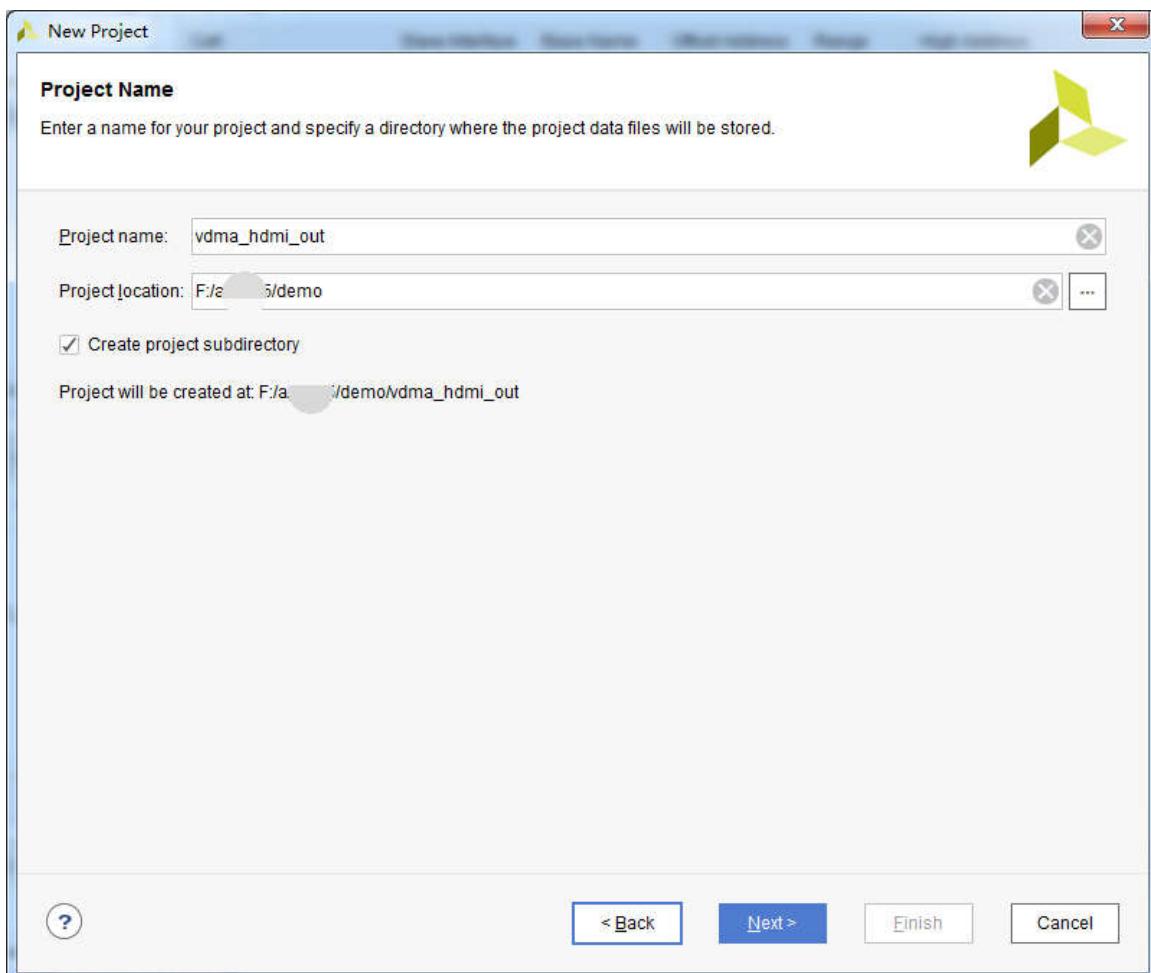
PS 没有集成显示控制系统，需要借助 PL 来实现，实现的方案有很多，但是都离不开 DMA 系统，DMA 系统可以完成显示数据从 ddr3 读出到显示器的显示，降低 CPU 的开销，VDMA 是 xilinx 开发的特殊 DMA，专门用于视频输入输出，是学习 xilinx FPGA 视频处理的重要内容。

前面的 HDMI 显示数据是 PL 内部产生的，这个实验中显示数据是 PS 生成的，然后 PL 通过 VDMA 送给 HDMI 接口。

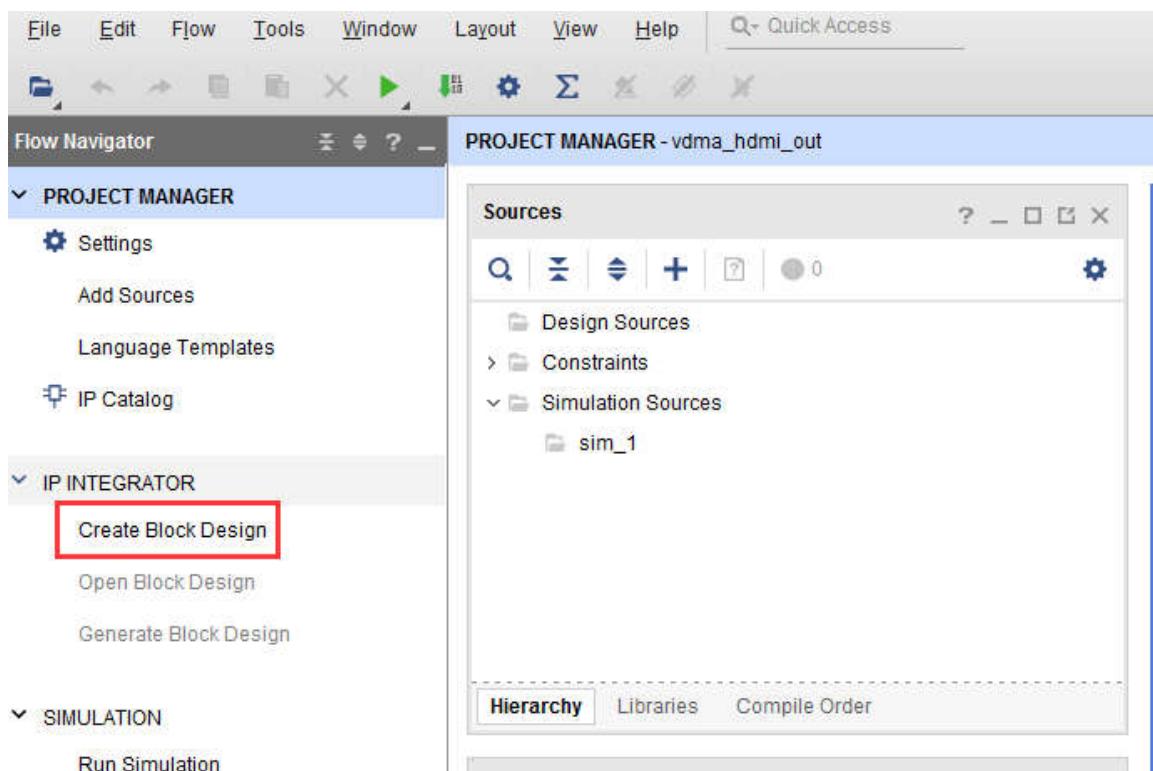
15.1 Vivado 工程建立

由于 VDMA 显示是一个非常重要的内容，本实验会详细介绍 Vivado 的搭建过程。

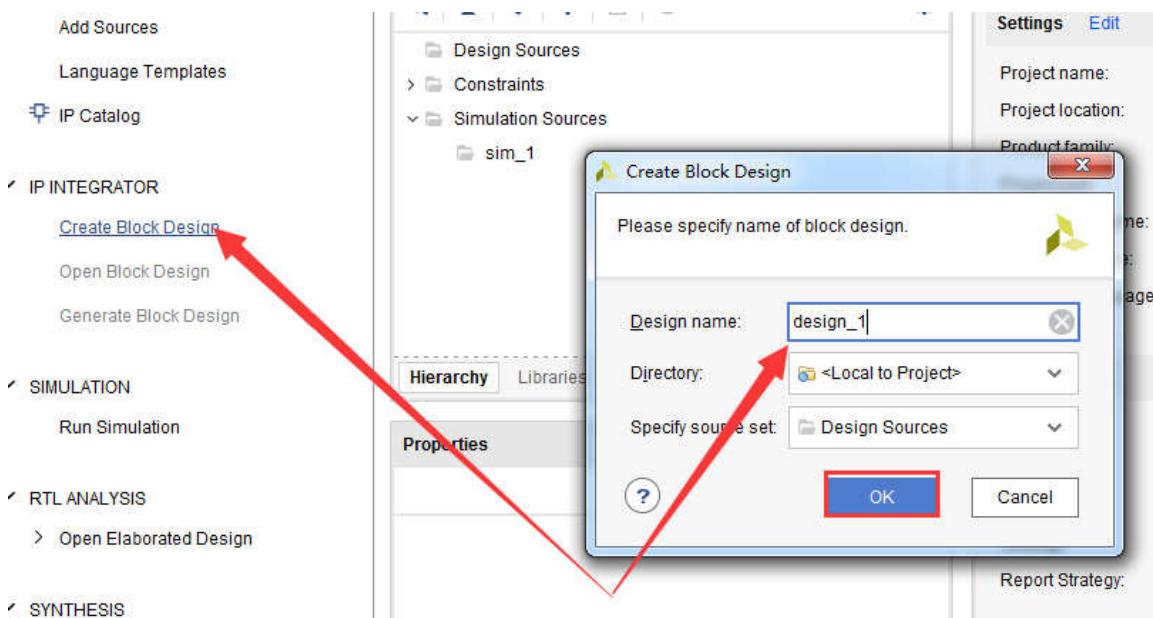
- 1) 新建一个名为 “vdma_hdmi_out” 工程。



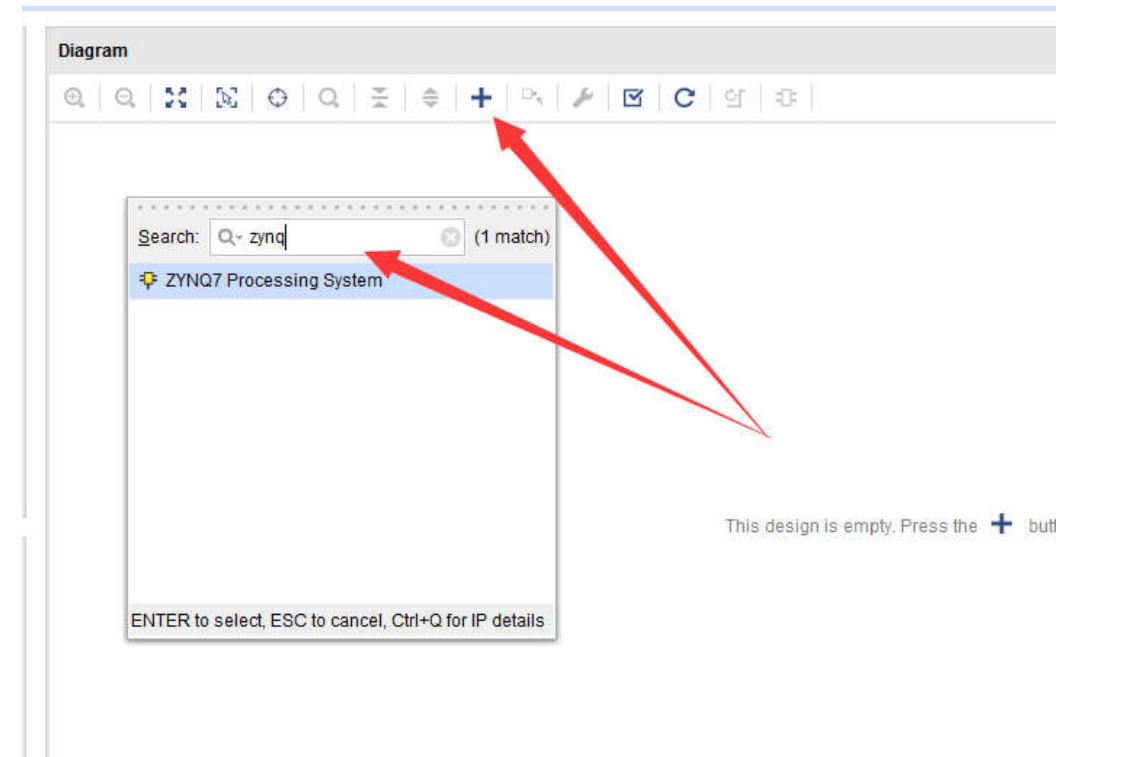
- 2) 创建一个 Block 设计



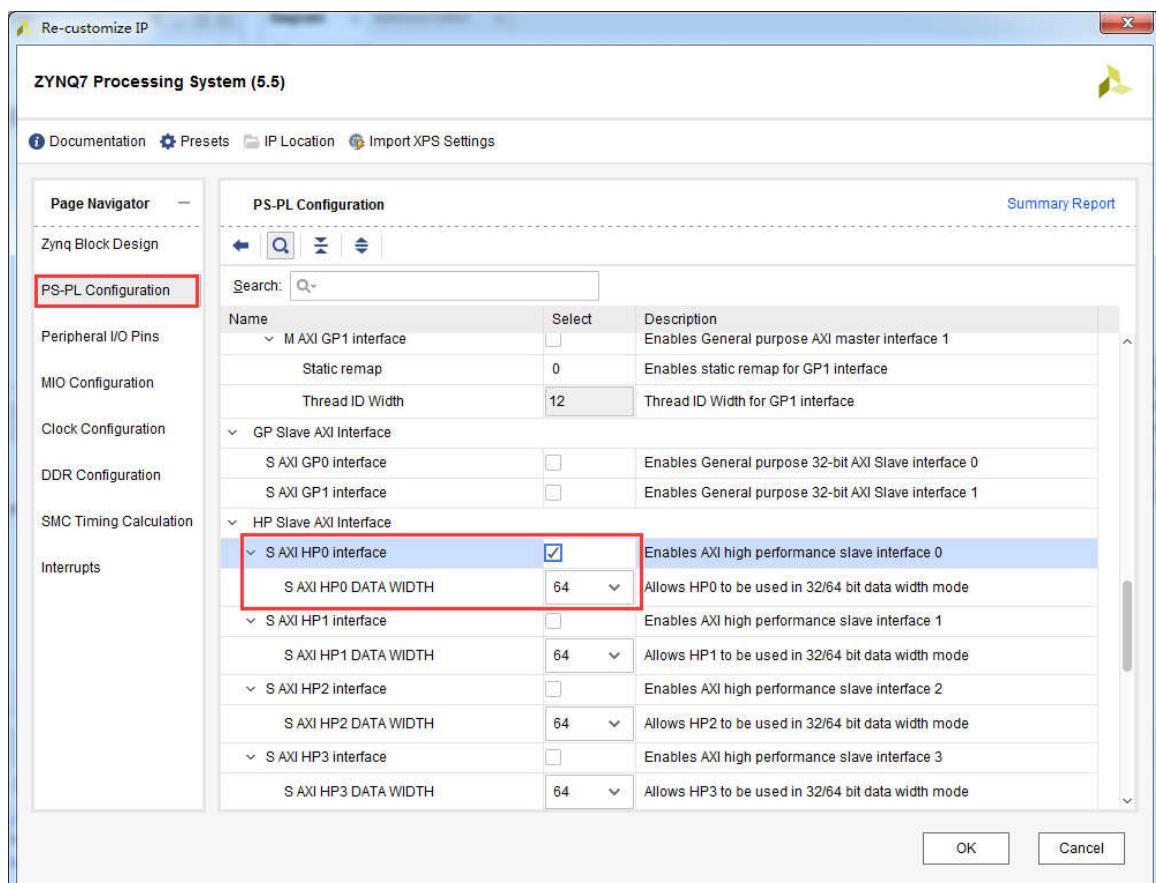
3) 设计名称保持默认不变



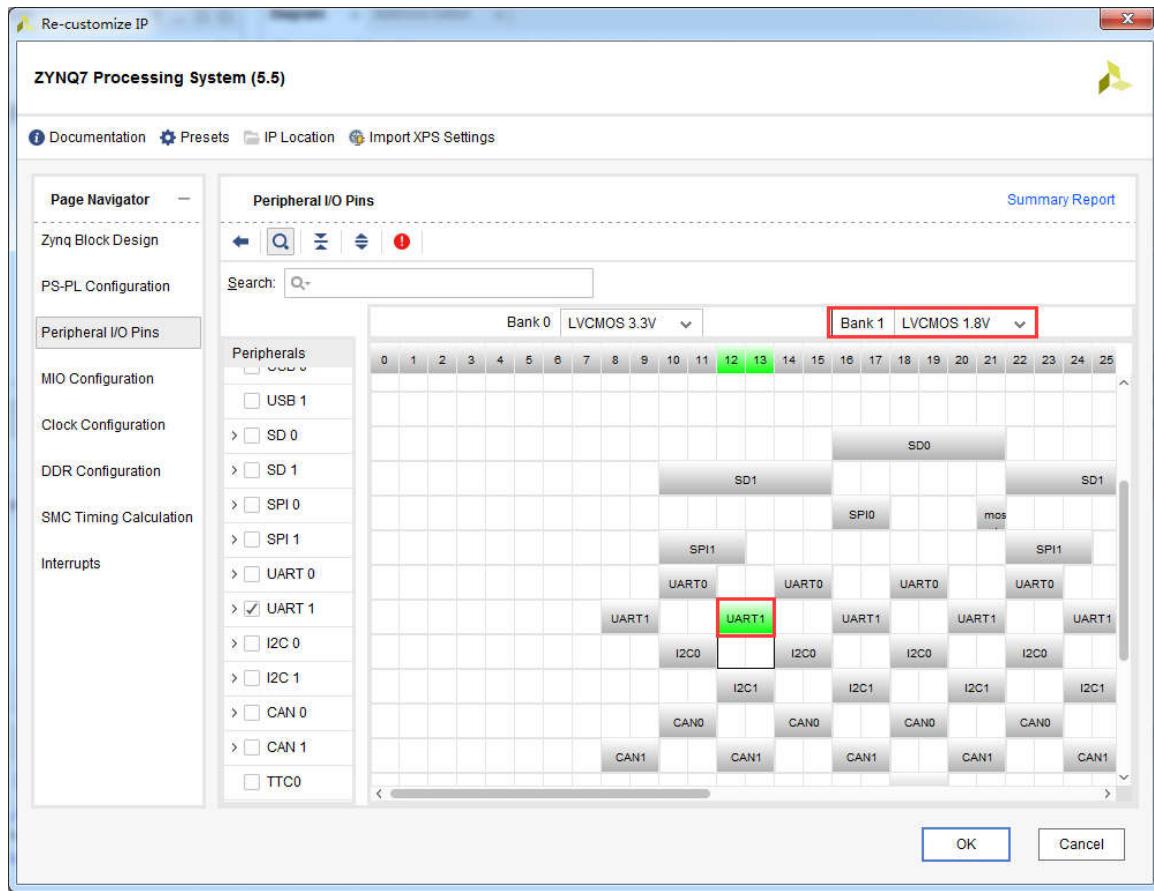
4) 添加 ZYNQ 处理器



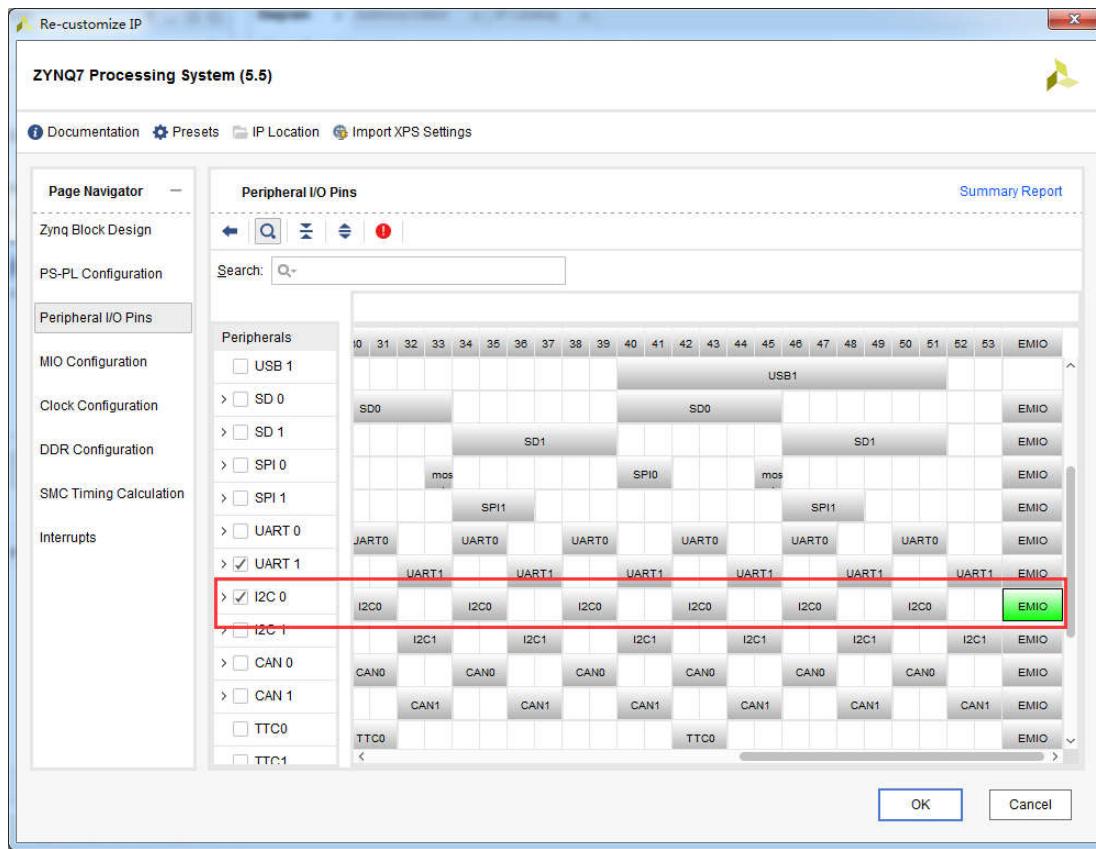
5) 配置 ZYNQ 参数，使能 HPO 接口，用于 VDMA 快速读取 ddr。



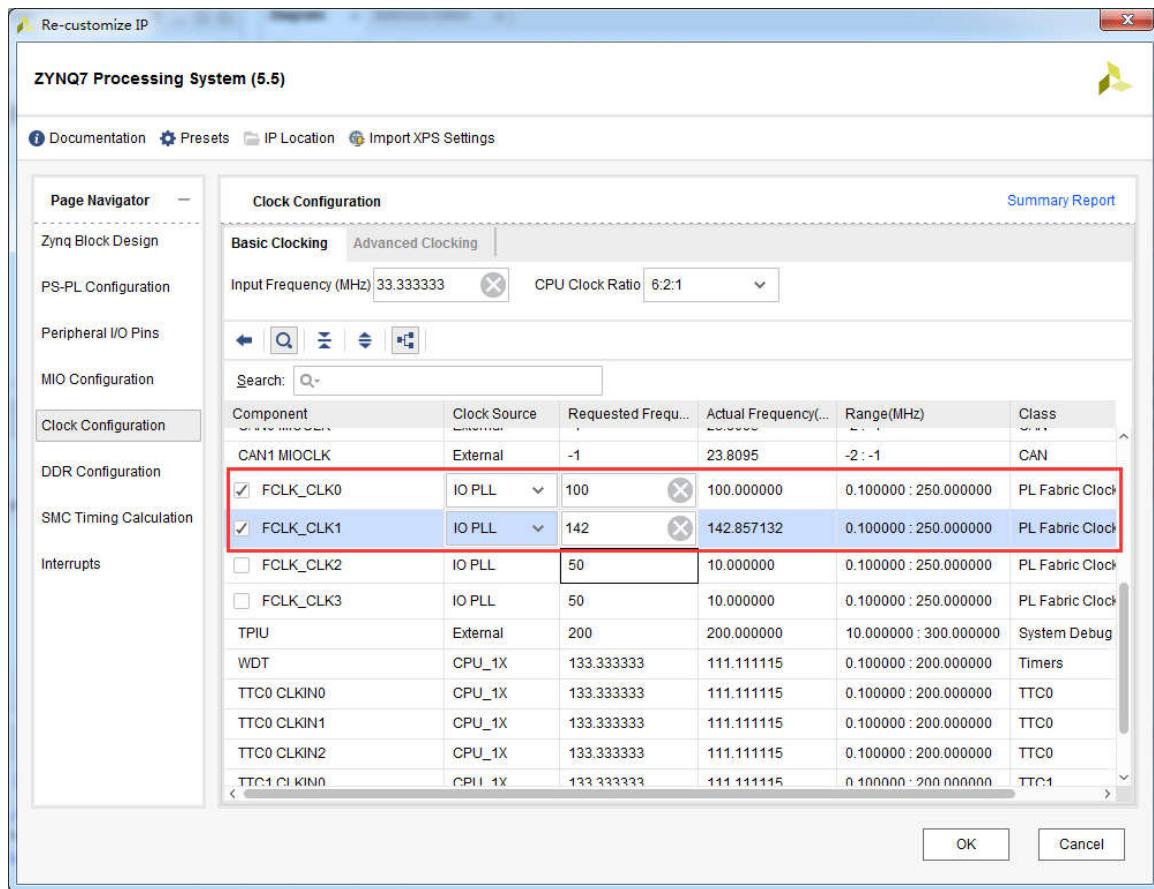
- 6) 配置 Bank 电平标准，Bank0 为 LVCMS 3.3V，Bank1 为 LVCMS 1.8V，使能串口



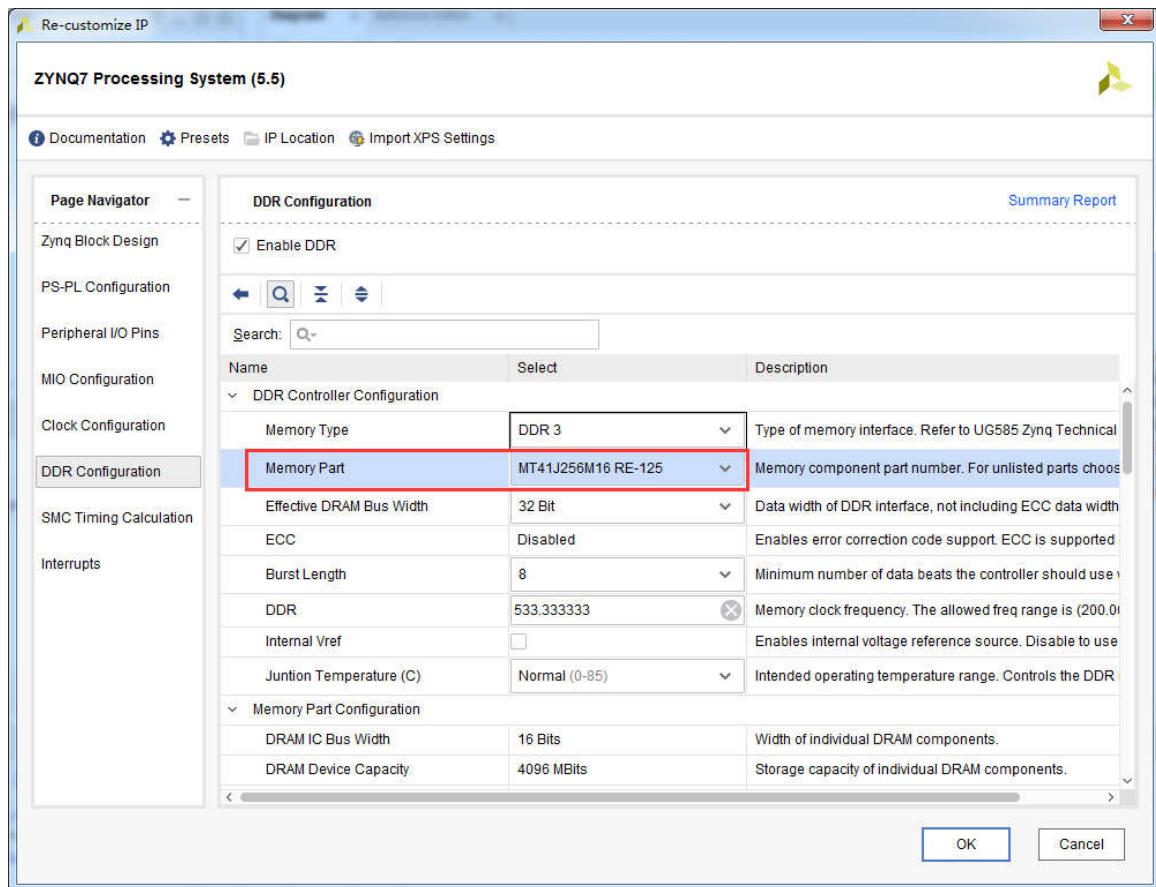
- 7) 使能 I2C0，并且选择 EMIO，这样可以把 I2C 连接到 PL 端。



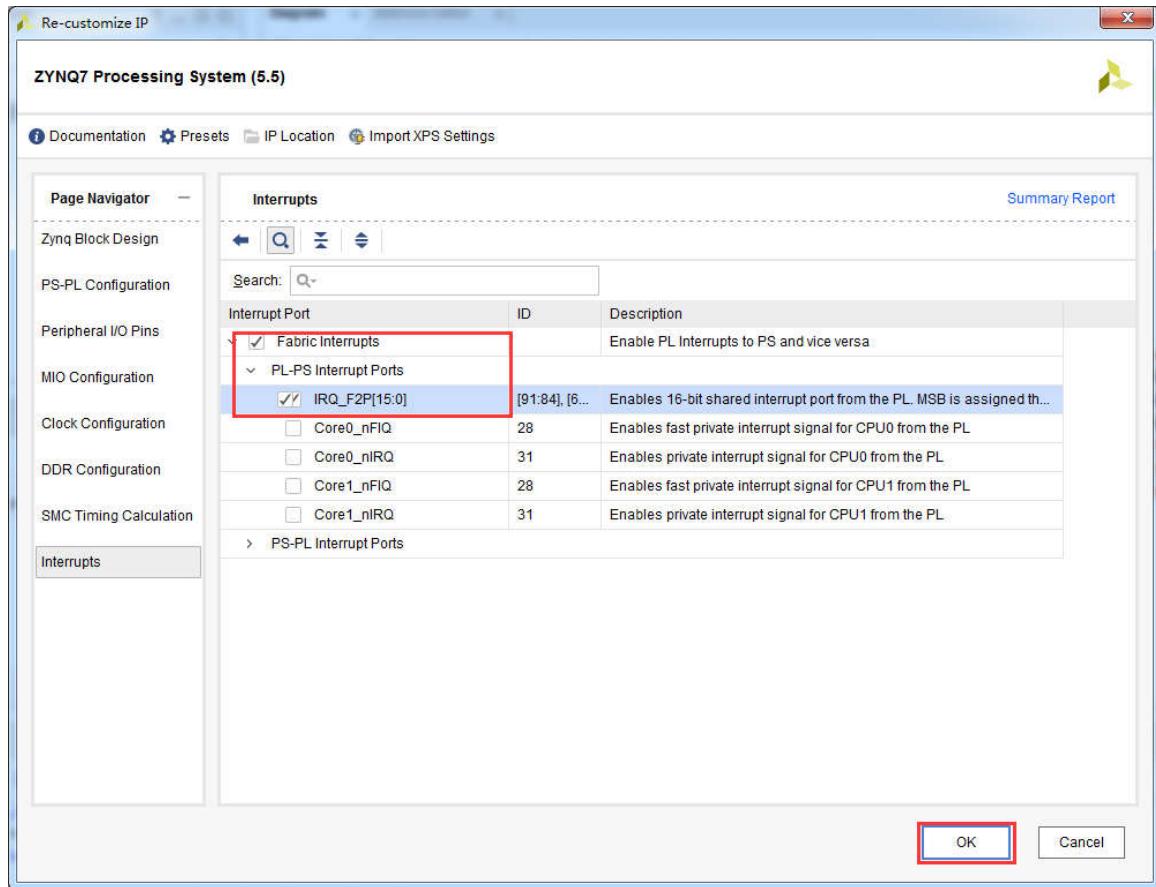
- 8) 配置时钟 , FCLK_CLK0 配置为 100Mhz , FCLK_CLK1 配置为 142Mhz , 这个时钟用于 VDMA 读取数据。



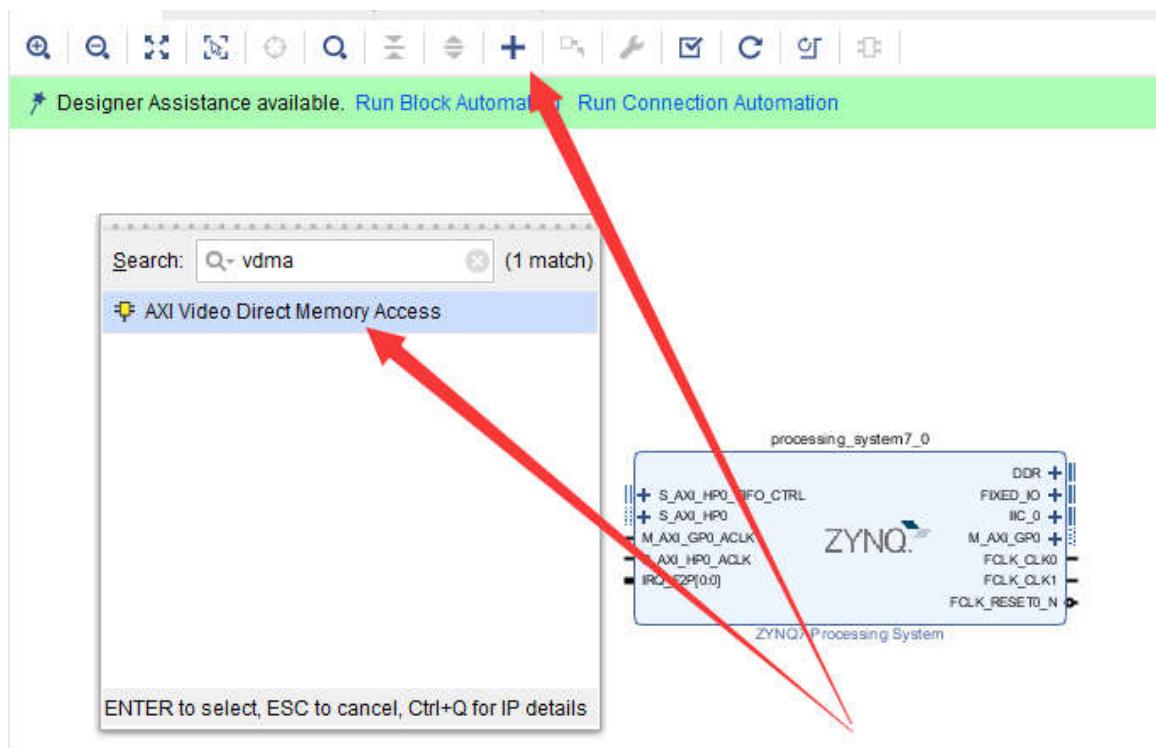
9) 配置 ddr3 , 选择 MT41256M16 RE-125



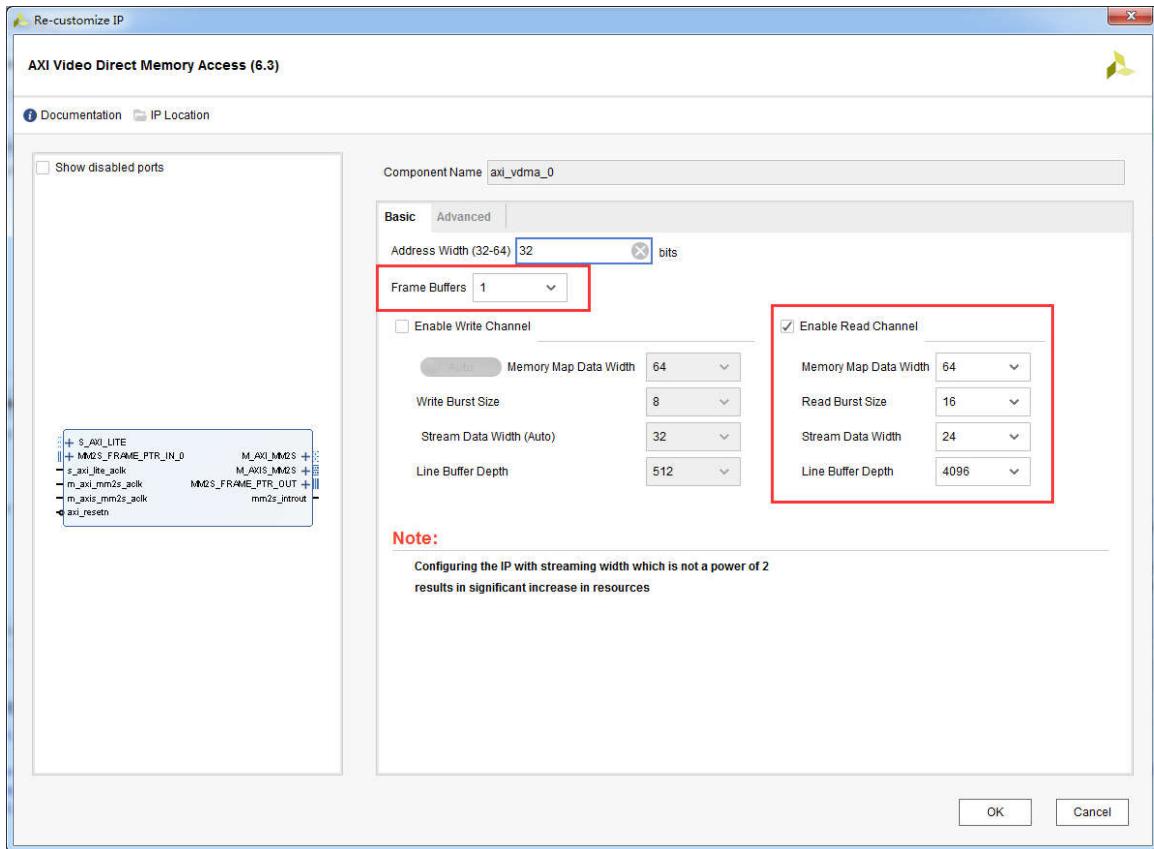
10) 配置中断，使能 IRQ_F2P，接收 PL 端的中断



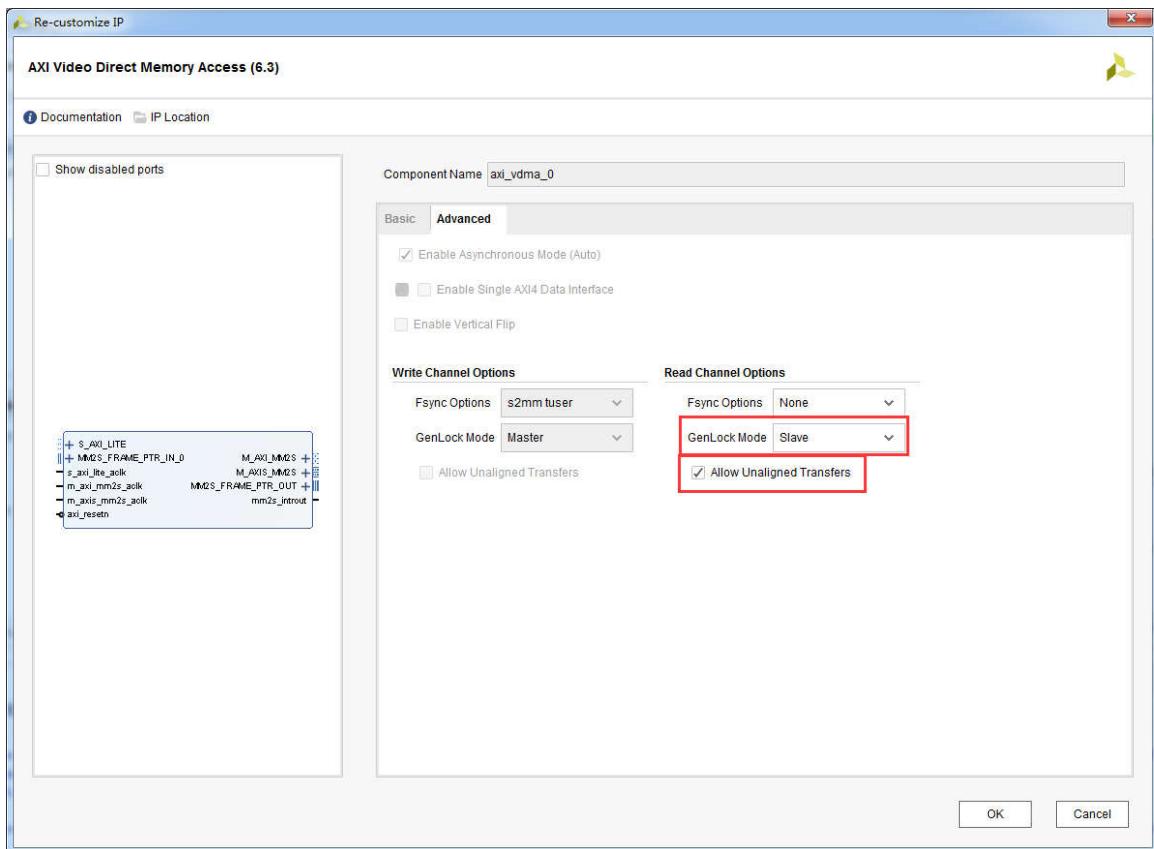
11) 添加 VDMA IP



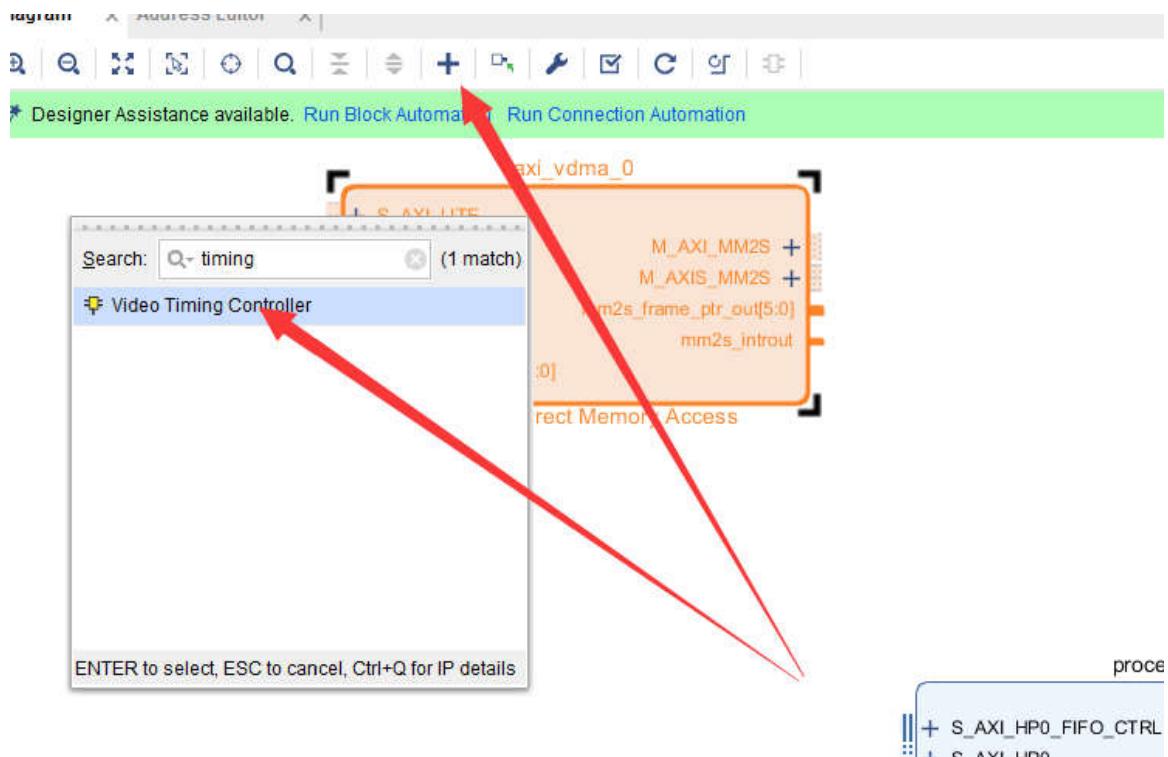
12) 按照下图配置 VDMA 基本参数



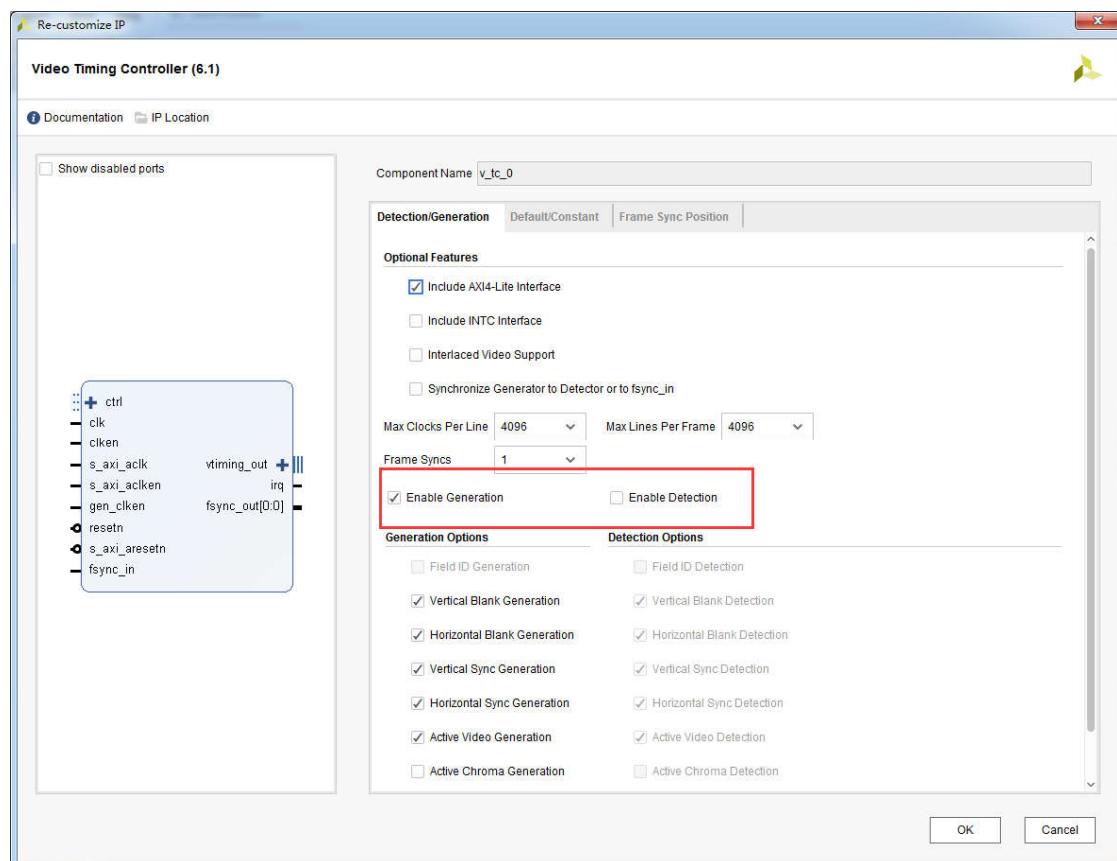
13) 配置 VDMA 高级参数



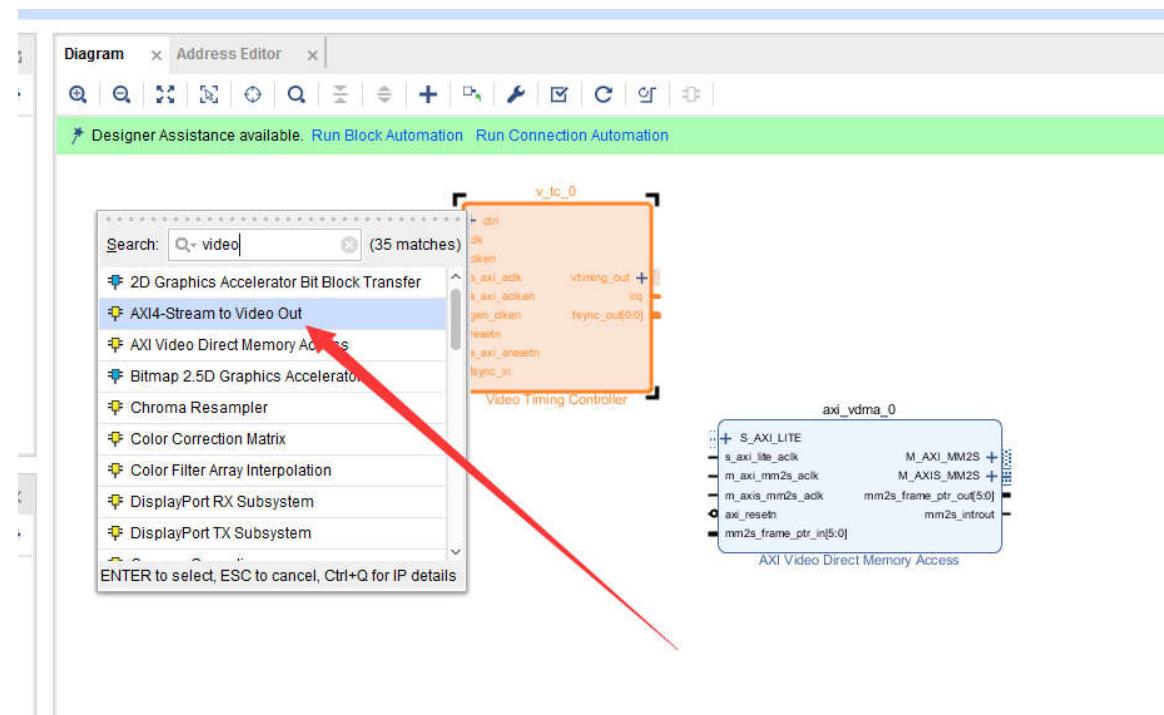
14) 添加视频时序控制器



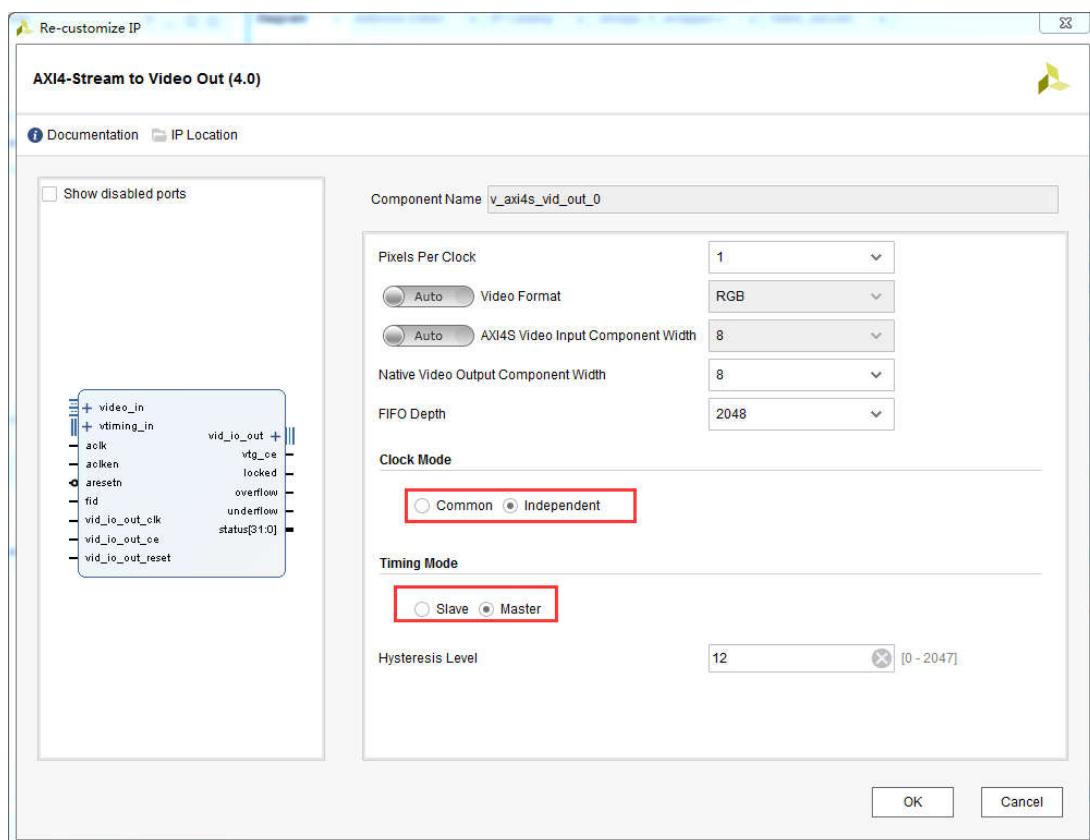
15) 配置视频时序控制器参数



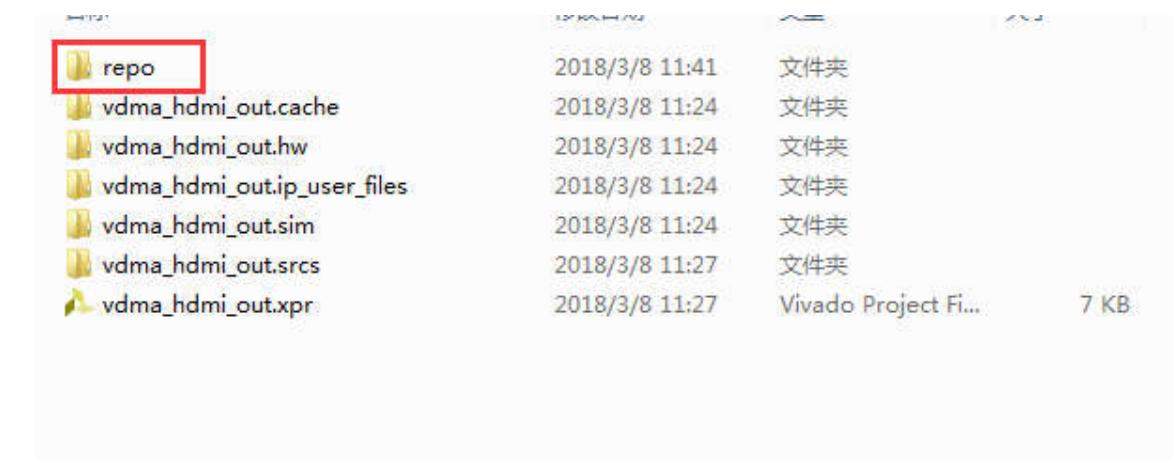
16) 添加 AXI 流转视频输出控制器



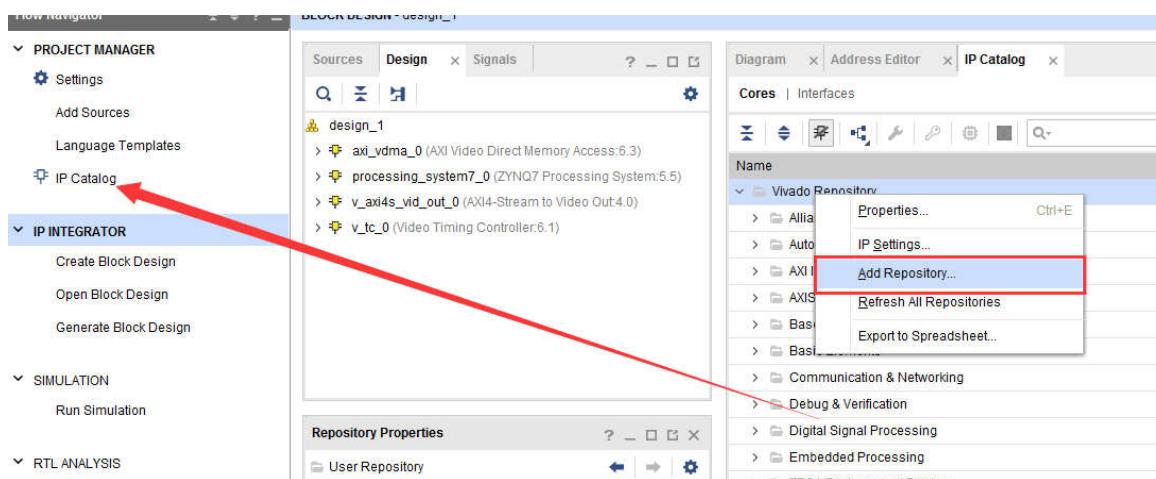
17) 配置 AXI 流转视频输出控制器参数



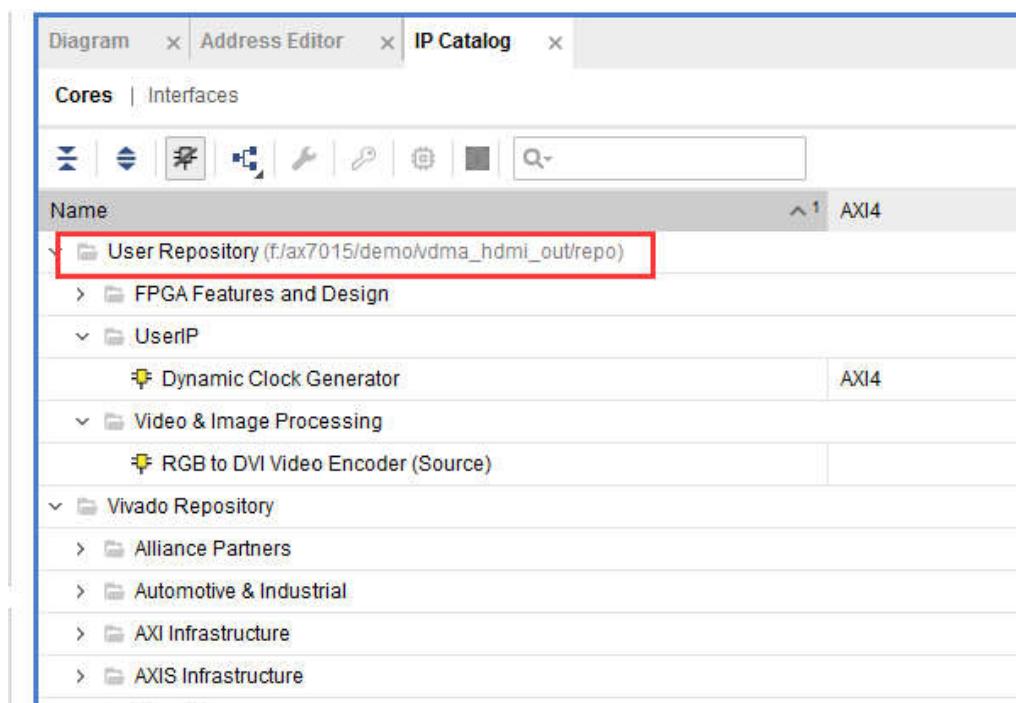
18) 由于视频有很多分辨率，各种分辨的时钟频率不相同，需要使用一个动态时钟控制器，这个 IP 来自开源软件，找到例程里的 repo 目录，复制到自己的目录下



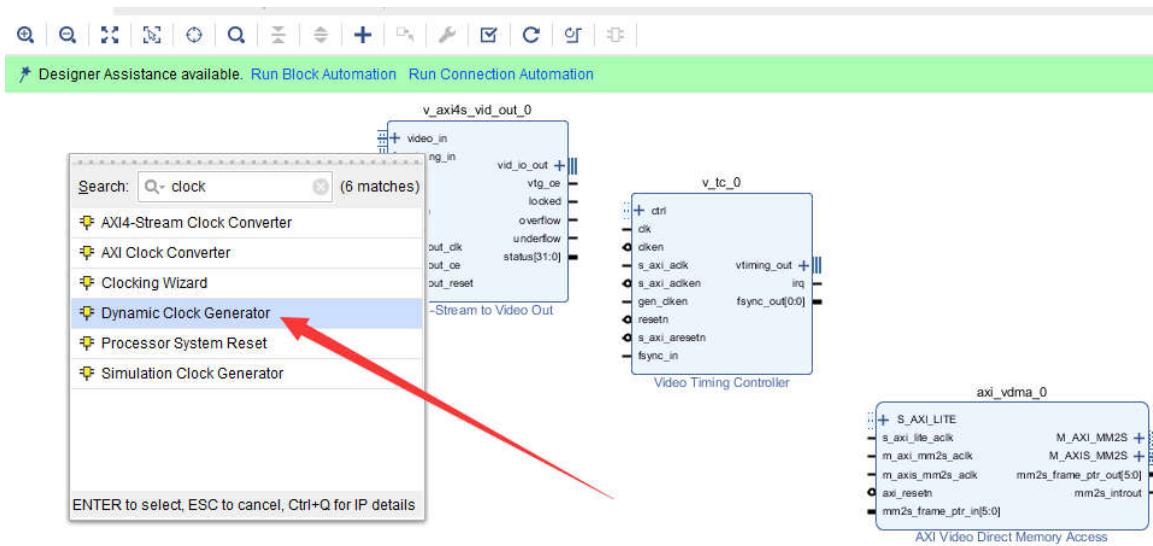
19) 添加 IP 仓库



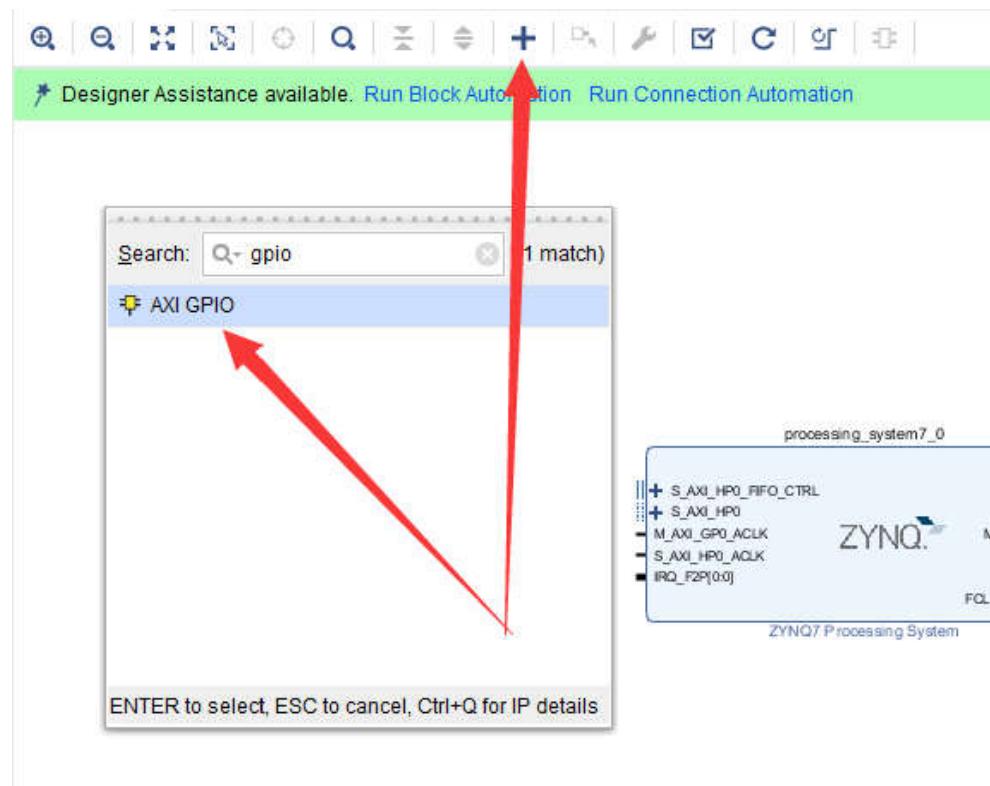
20) 添加完成以后可以看到很多 IP



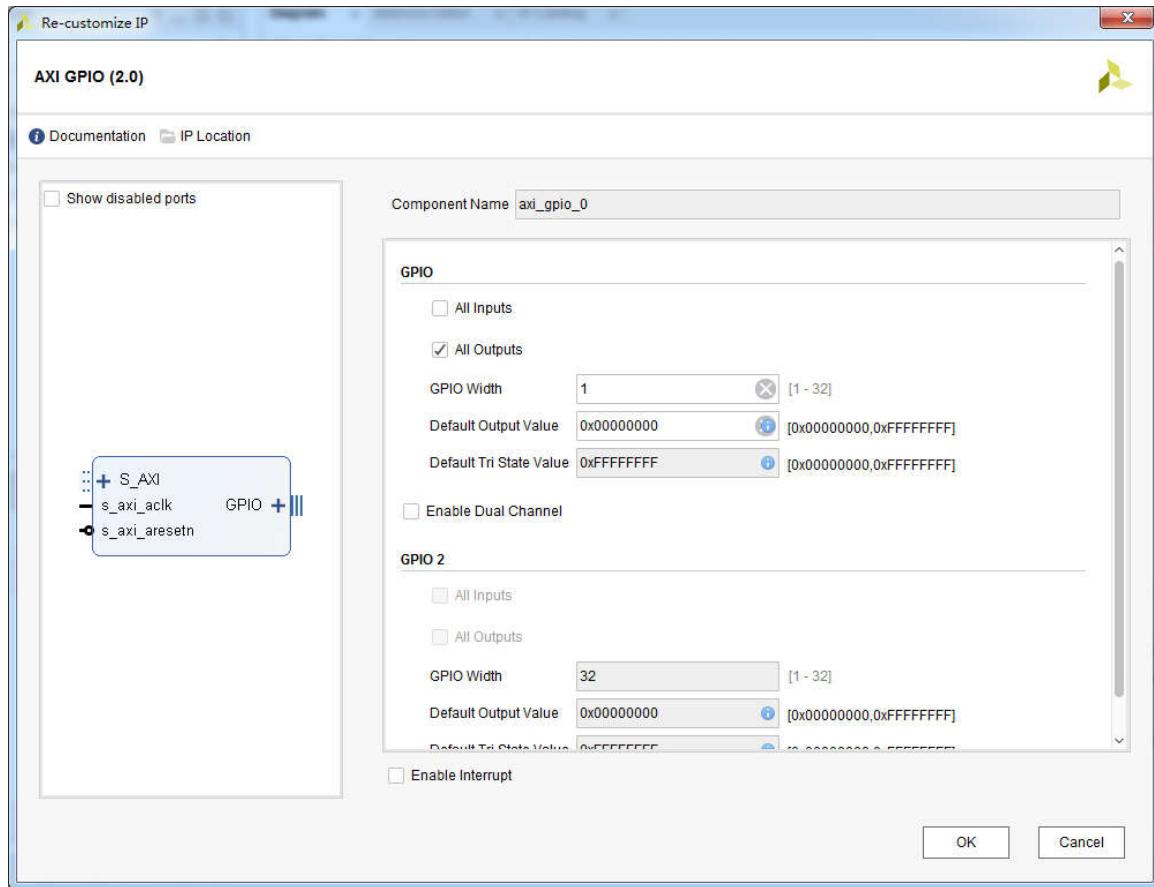
21) 添加动态时钟控制器



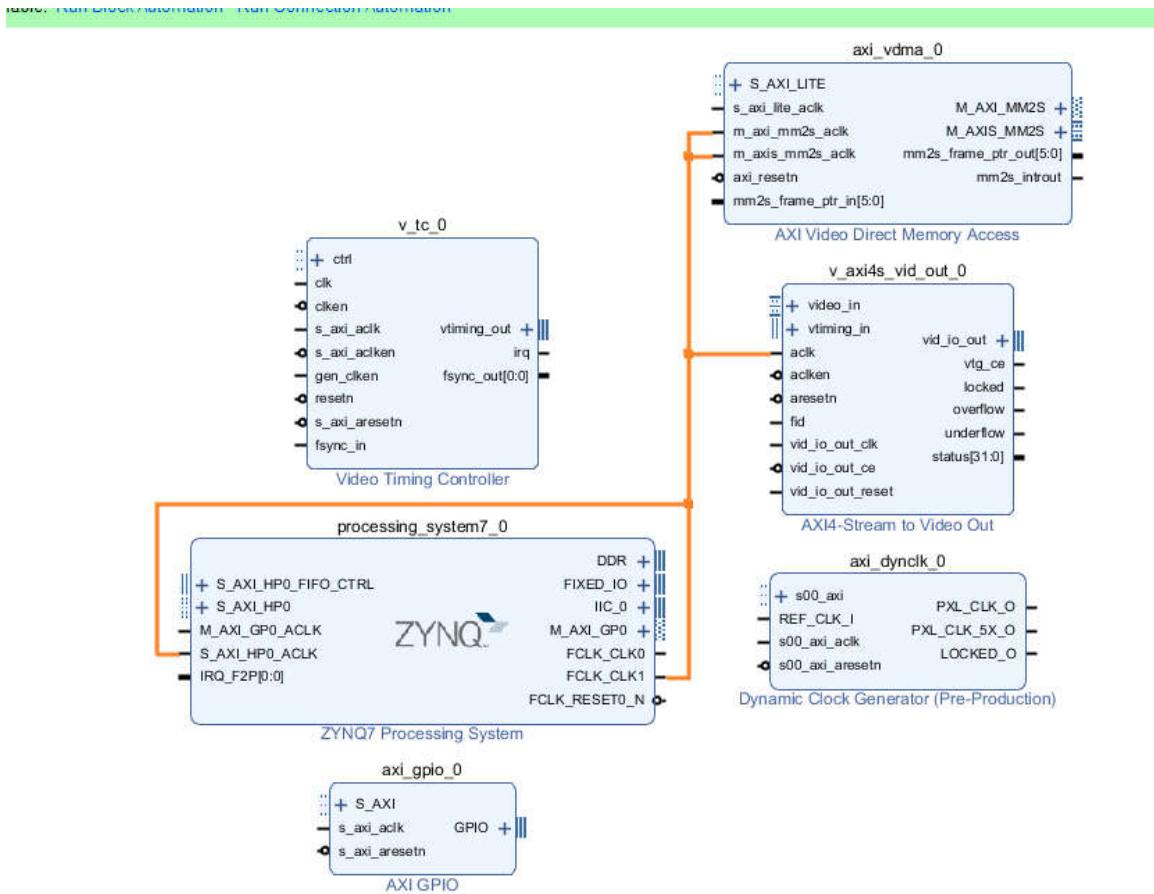
22) 添加一个 GPIO，用于 HDMI 模块复位



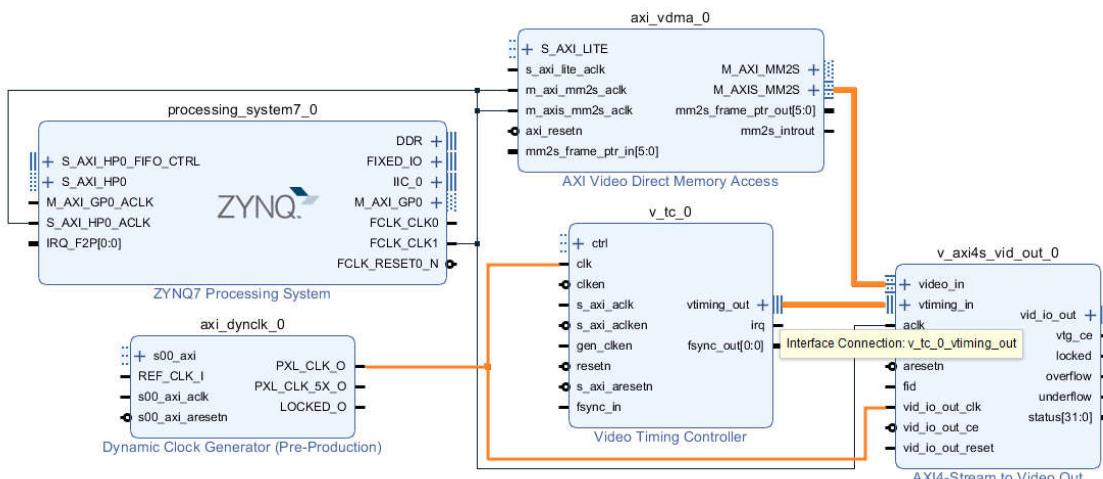
23) 配置 GPIO 参数

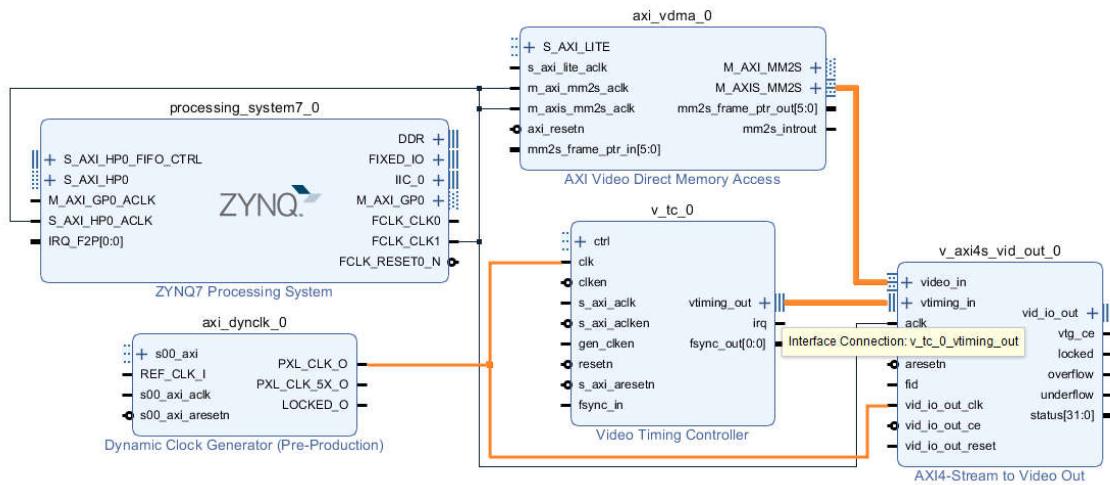


24) 连接 Vivado 可能无法自动连接的时钟信号

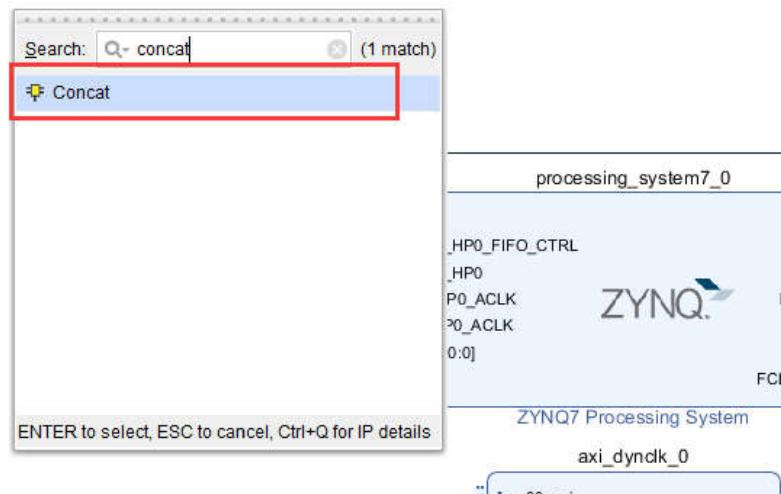
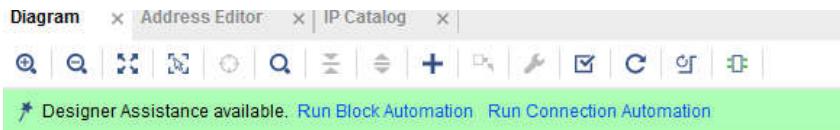


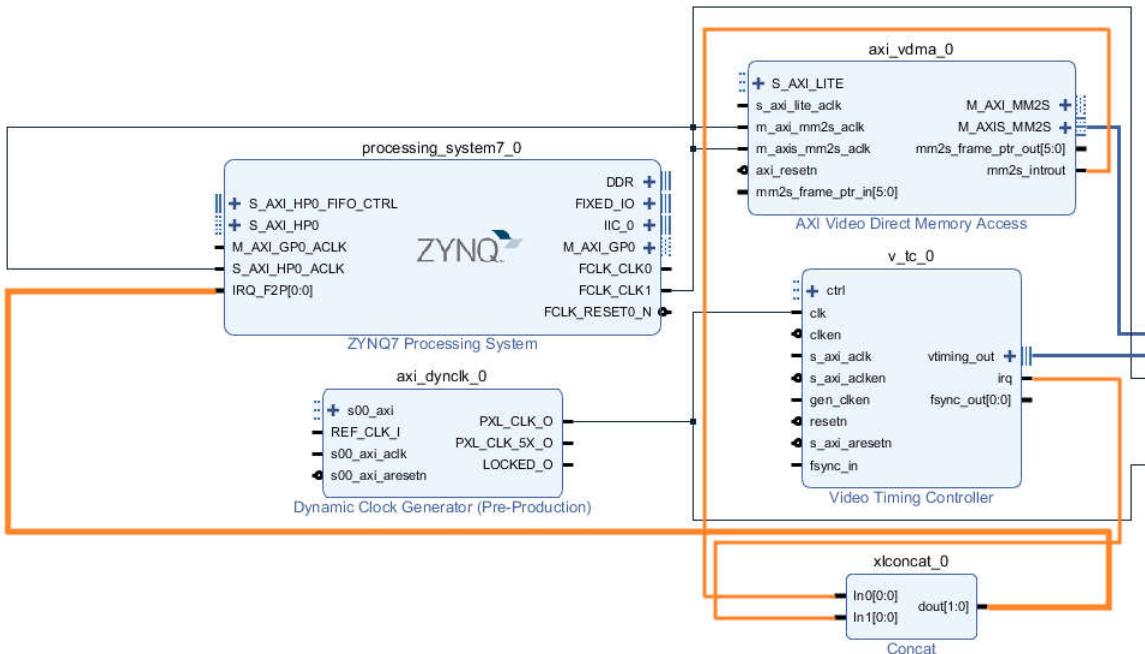
25) 连接其他一些关键信号



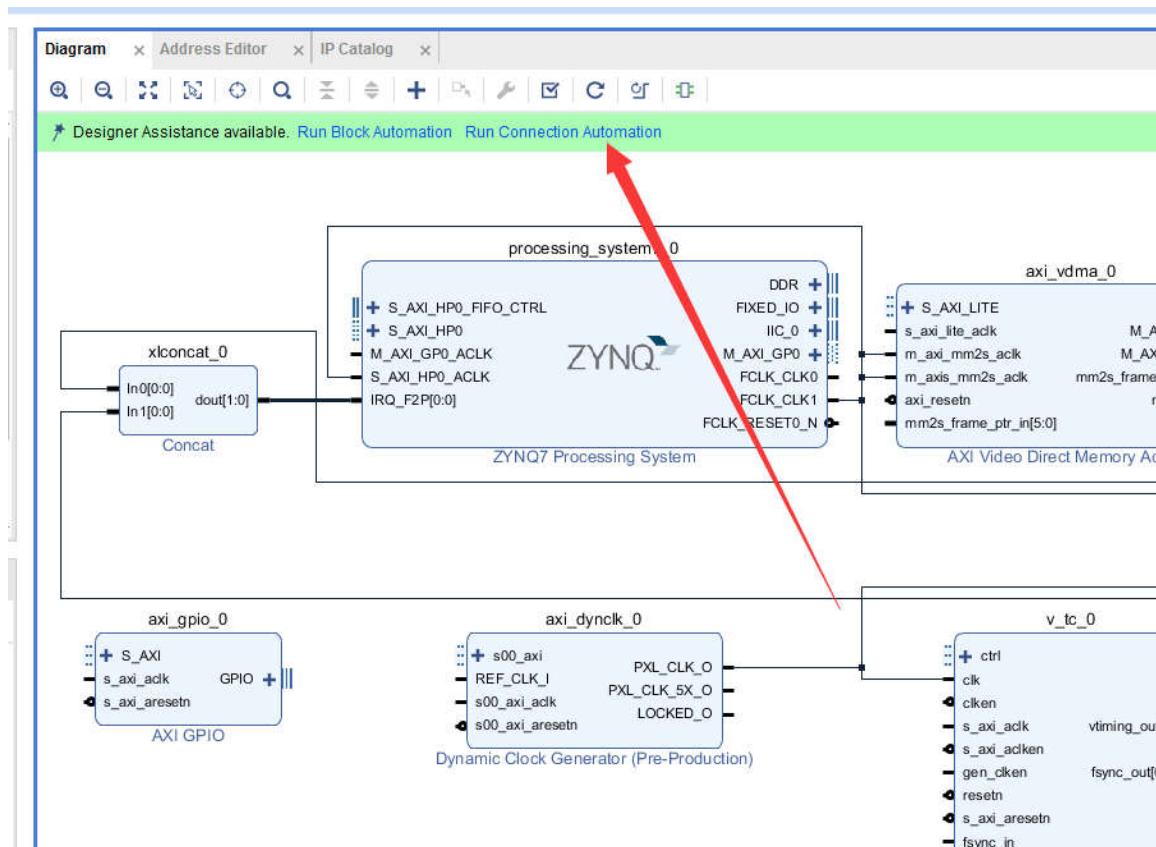


26) 连接中断信号，需要先添加一个 Concat IP，用于信号连接

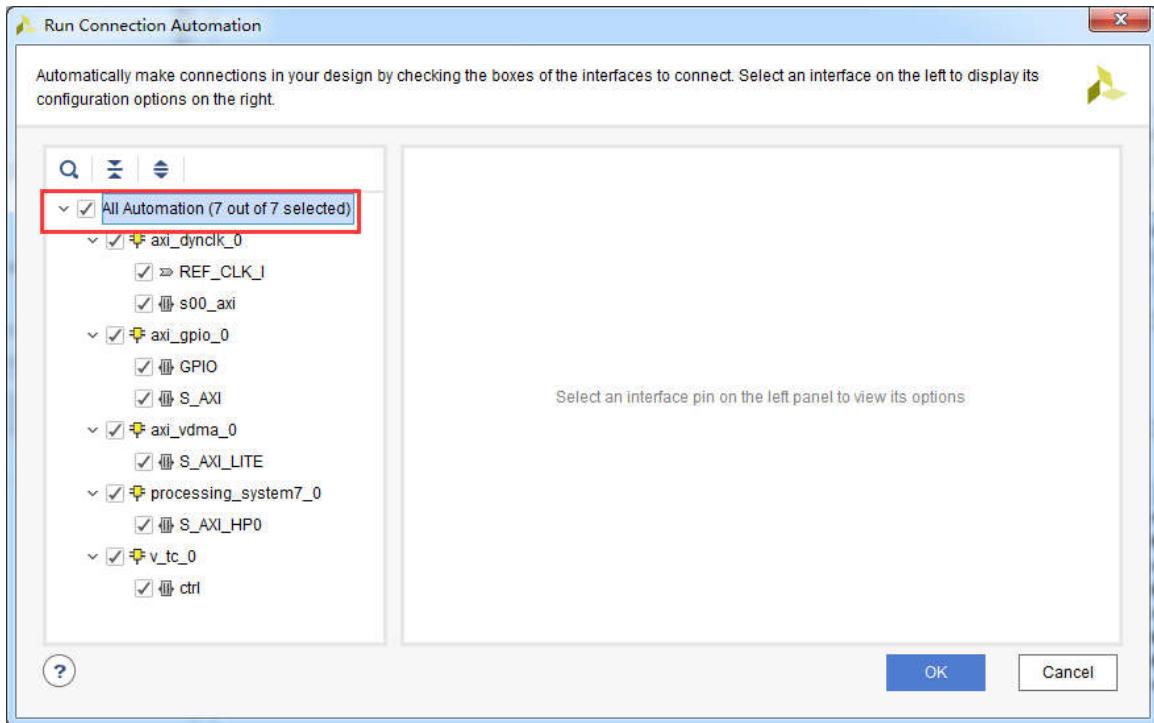




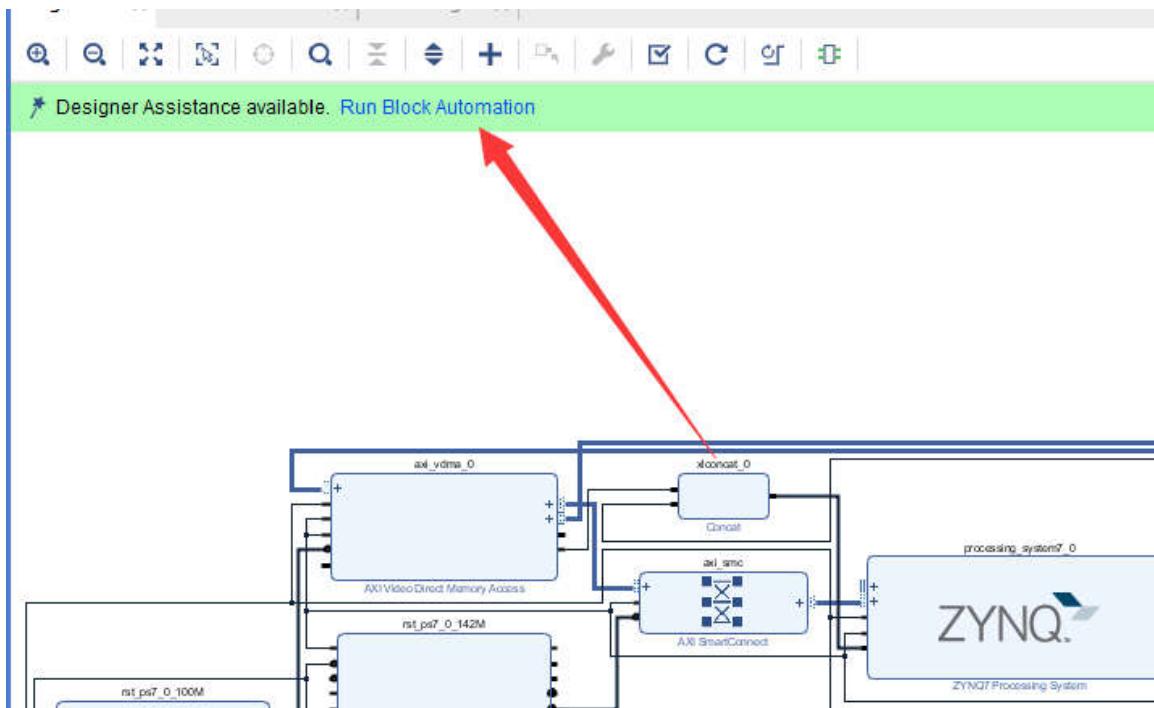
27) 使用 Vivado 自动连接功能，完成剩下的线连接



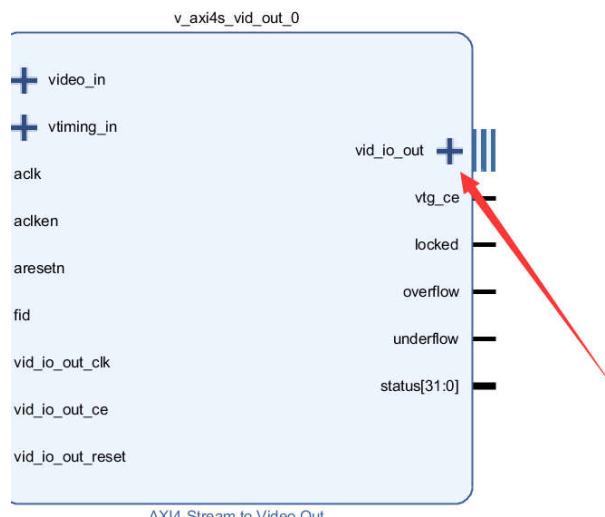
28) 选择所有模块自动连接



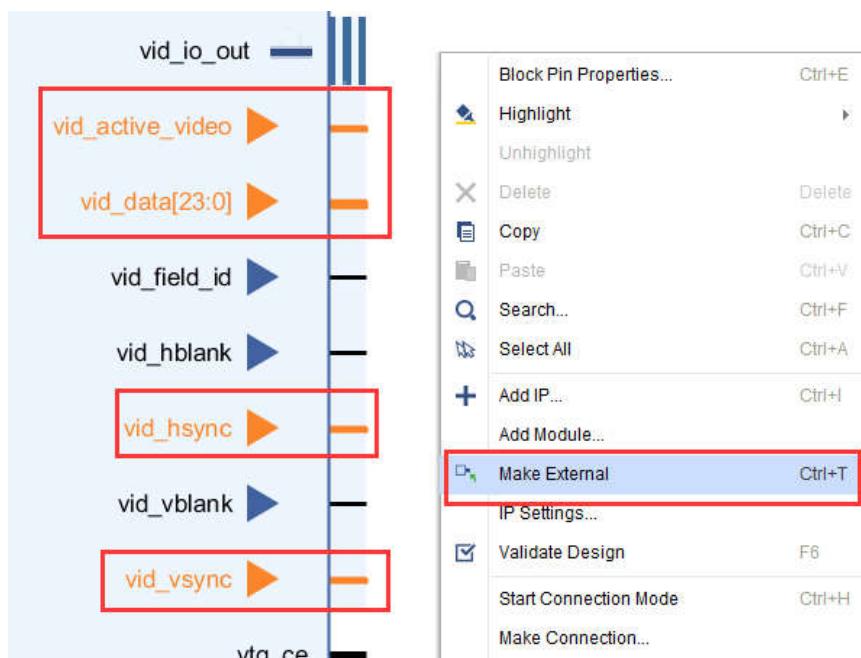
29) 运行“Run Block Automation”完成一些必要的端口导出



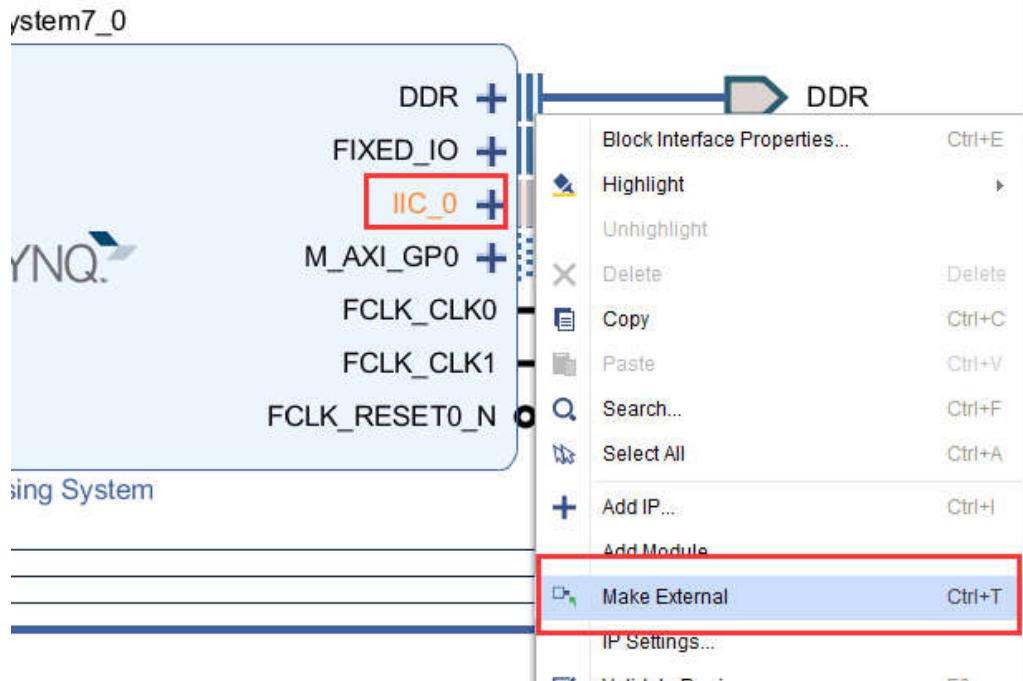
30) 展开 vid_io_out 端口



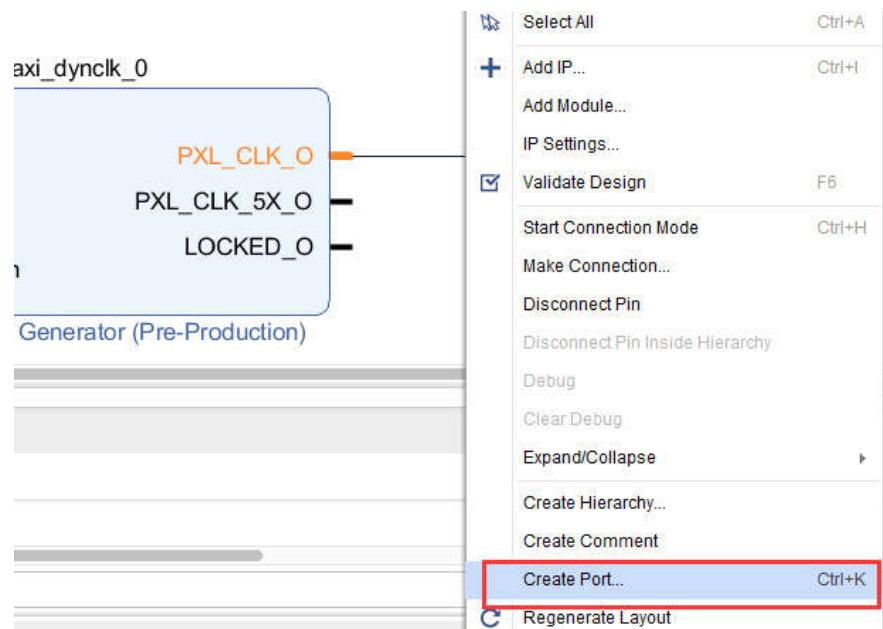
31) 选择我们需要的端口导出



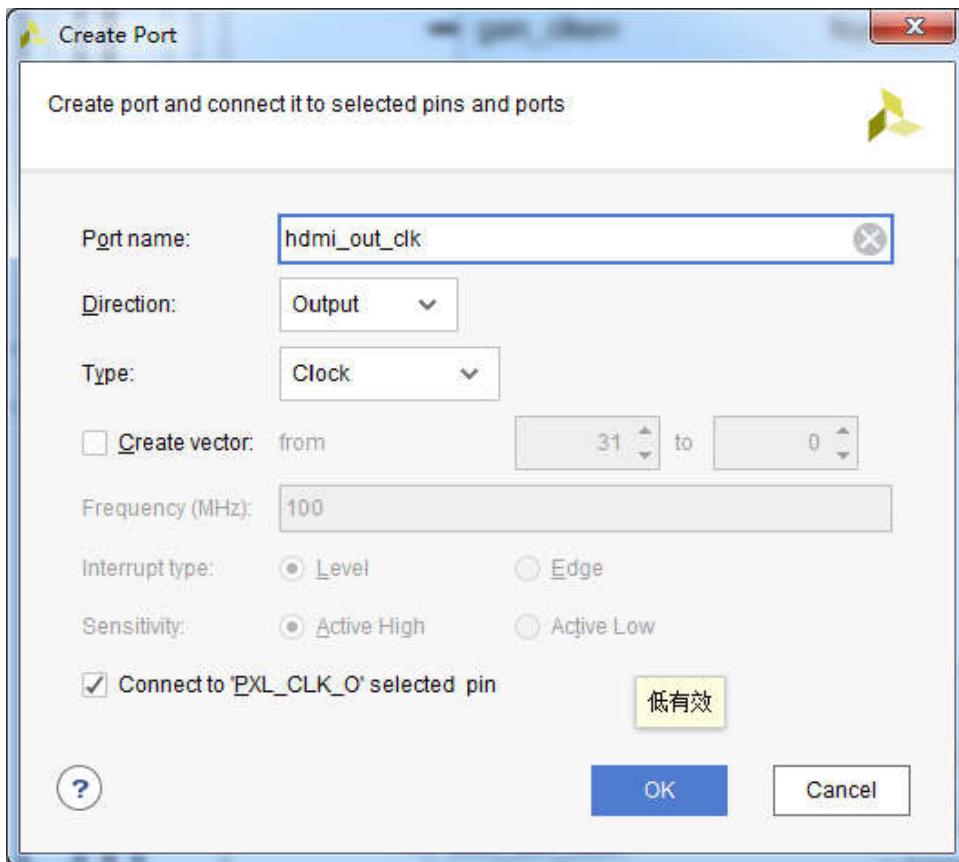
32) 导出 IIC_0 端口



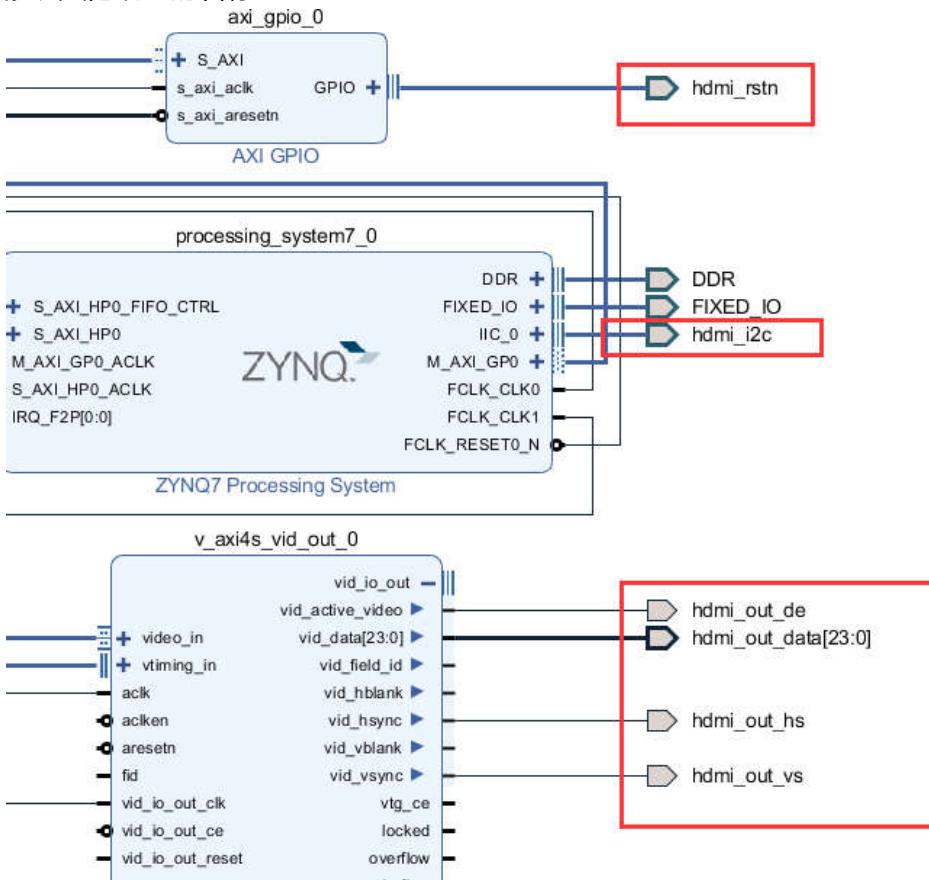
33) 导出视频时钟端口



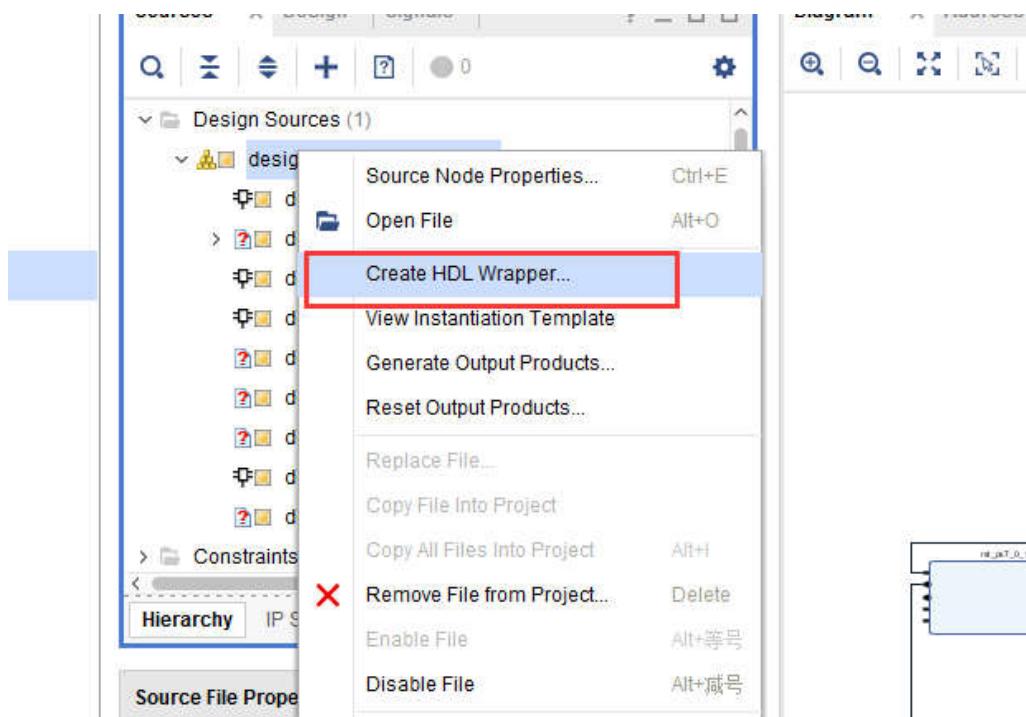
34) 名称修改为 hdmi_out_clk



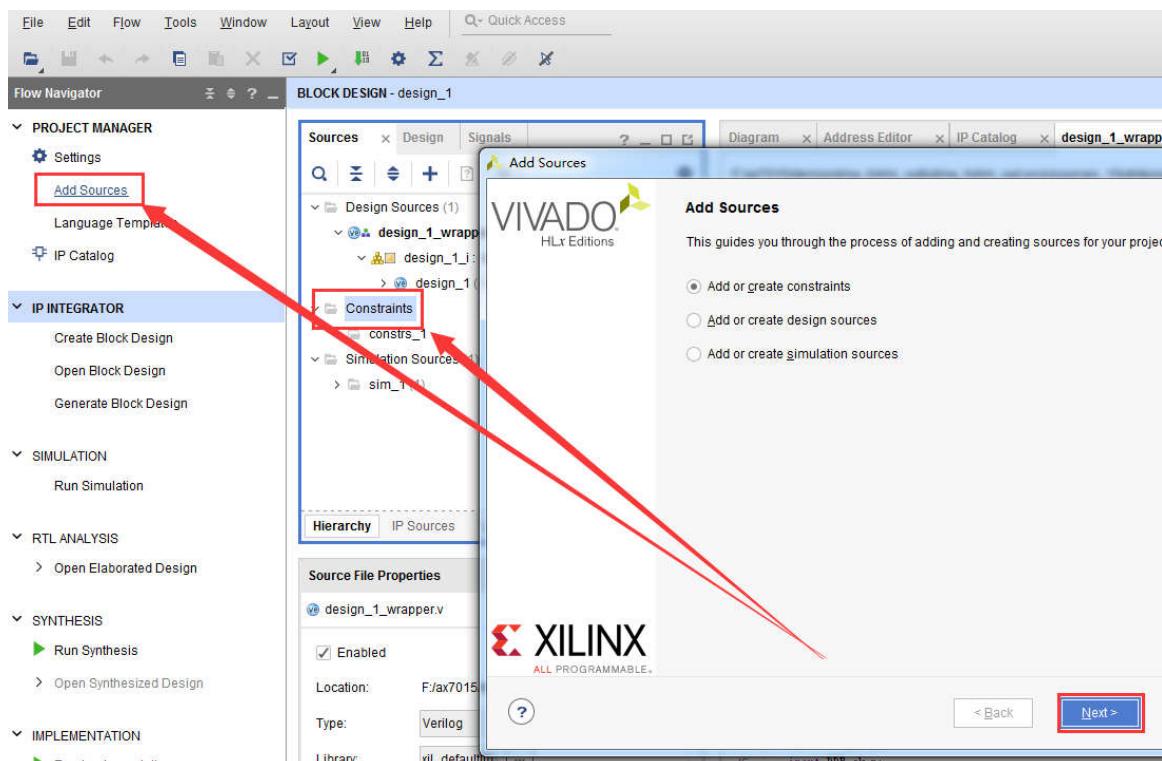
35) 修改其他端口的名称



36) 保存设计后按 F6 检查设计，没有问题后创建 HDL 文件



37) 添加 HDMI 输出的 xdc 文件，约束管脚



38) xdc 文件内容如下

```
set_property PACKAGE_PIN H1 [get_ports hdmi_out_clk]
set_property PACKAGE_PIN G3 [get_ports {hdmi_out_data[0]}]
```

```
set_property PACKAGE_PIN H3 [get_ports {hdmi_out_data[1]}]
set_property PACKAGE_PIN H4 [get_ports {hdmi_out_data[2]}]
set_property PACKAGE_PIN G7 [get_ports {hdmi_out_data[3]}]
set_property PACKAGE_PIN G8 [get_ports {hdmi_out_data[4]}]
set_property PACKAGE_PIN G1 [get_ports {hdmi_out_data[5]}]
set_property PACKAGE_PIN H5 [get_ports {hdmi_out_data[6]}]
set_property PACKAGE_PIN H6 [get_ports {hdmi_out_data[7]}]
set_property PACKAGE_PIN G4 [get_ports {hdmi_out_data[8]}]
set_property PACKAGE_PIN F4 [get_ports {hdmi_out_data[9]}]
set_property PACKAGE_PIN F5 [get_ports {hdmi_out_data[10]}]
set_property PACKAGE_PIN E5 [get_ports {hdmi_out_data[11]}]
set_property PACKAGE_PIN G6 [get_ports {hdmi_out_data[12]}]
set_property PACKAGE_PIN F6 [get_ports {hdmi_out_data[13]}]
set_property PACKAGE_PIN E7 [get_ports {hdmi_out_data[14]}]
set_property PACKAGE_PIN F7 [get_ports {hdmi_out_data[15]}]
set_property PACKAGE_PIN D3 [get_ports {hdmi_out_data[16]}]
set_property PACKAGE_PIN C3 [get_ports {hdmi_out_data[17]}]
set_property PACKAGE_PIN C4 [get_ports {hdmi_out_data[18]}]
set_property PACKAGE_PIN D5 [get_ports {hdmi_out_data[19]}]
set_property PACKAGE_PIN C5 [get_ports {hdmi_out_data[20]}]
set_property PACKAGE_PIN C6 [get_ports {hdmi_out_data[21]}]
set_property PACKAGE_PIN E8 [get_ports {hdmi_out_data[22]}]
set_property PACKAGE_PIN D8 [get_ports {hdmi_out_data[23]}]
set_property PACKAGE_PIN G2 [get_ports hdmi_out_de]
set_property PACKAGE_PIN E4 [get_ports hdmi_out_hs]
set_property PACKAGE_PIN L4 [get_ports {hdmi_rstn_tri_o[0]}]
set_property PACKAGE_PIN E3 [get_ports hdmi_out_vs]
set_property PACKAGE_PIN J8 [get_ports hdmi_i2c_scl_io]
set_property PACKAGE_PIN K8 [get_ports hdmi_i2c_sda_io]

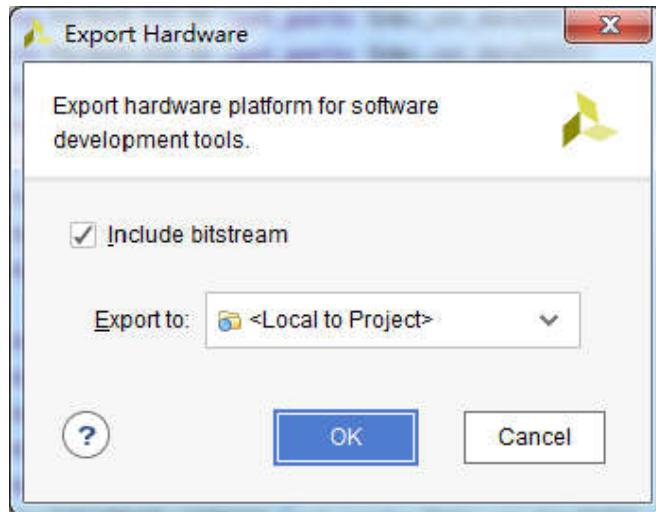
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_i2c_scl_io]
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_i2c_sda_io]
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_out_clk]
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_out_de]
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_out_hs]
set_property IOSTANDARD LVCMOS33 [get_ports {hdmi_out_data[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_out_vs]
set_property IOSTANDARD LVCMOS33 [get_ports {hdmi_rstn_tri_o[0]}]

set_property SLEW FAST [get_ports {hdmi_out_data[*]}]
set_property DRIVE 8 [get_ports {hdmi_out_data[*]}]
set_property SLEW FAST [get_ports hdmi_out_clk]
set_property SLEW FAST [get_ports hdmi_out_de]
set_property SLEW FAST [get_ports hdmi_out_hs]
set_property SLEW FAST [get_ports hdmi_out_vs]
```

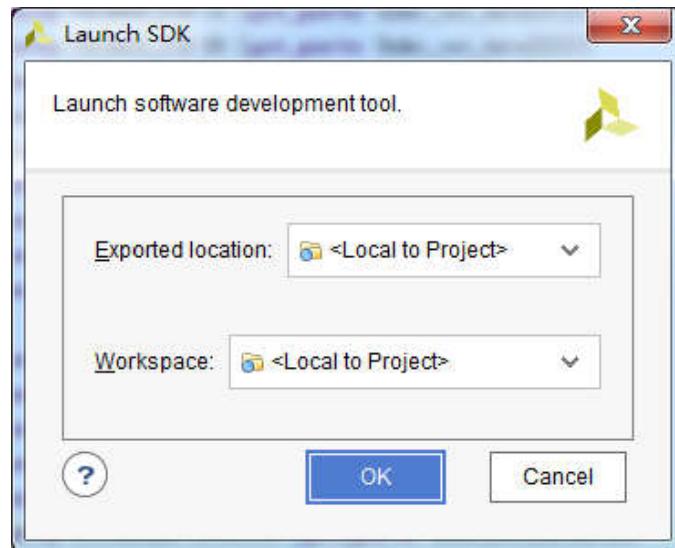
39) 编译生成 bit 文件

15.2 SDK 软件编写调试

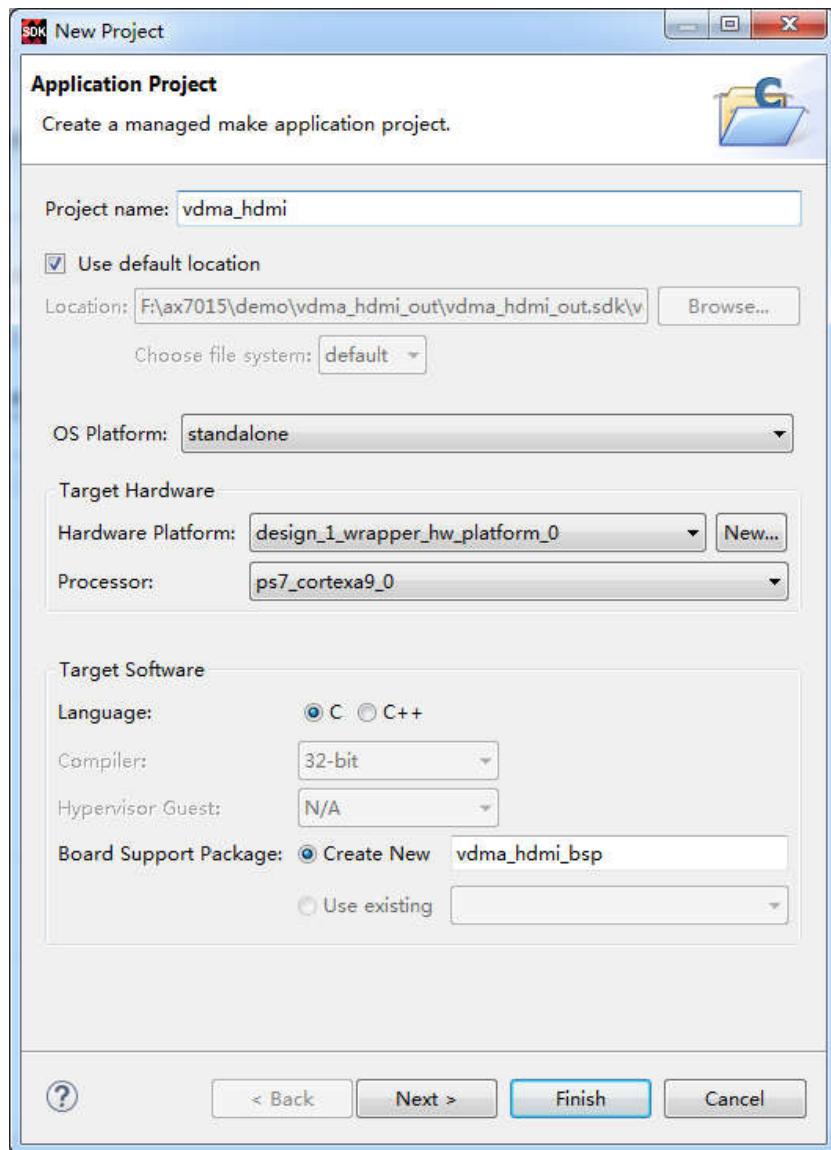
1) 导出硬件



2) 运行 SDK



3) 新建一个名为 vdma_hdmi 的 APP



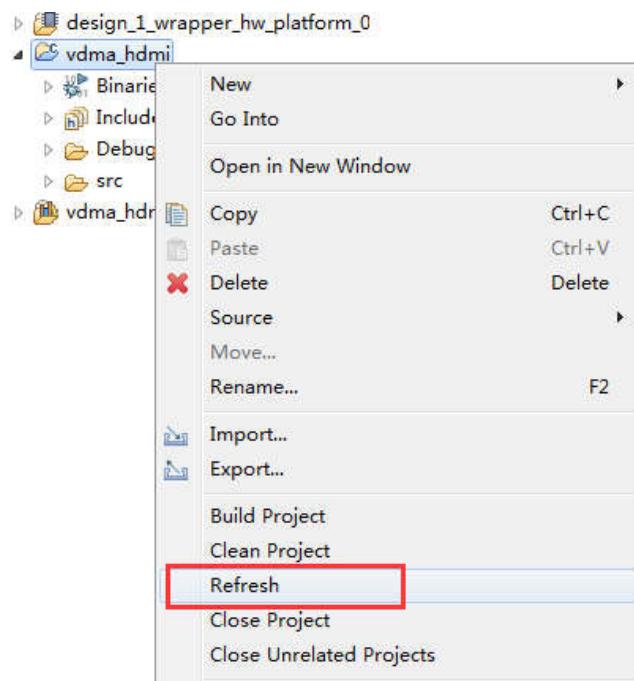
- 4) 由于程序文件较多，不再具体介绍，直接复制例程的源代码。删除 src 目录下的文件，使用例程的 src 目录文件代替

新加卷 (F:) ▶ a: ▶ demo ▶ vdma_hdmi_out ▶ vdma_hdmi_out.sdk ▶ vdma_hdmi ▶ src ▶

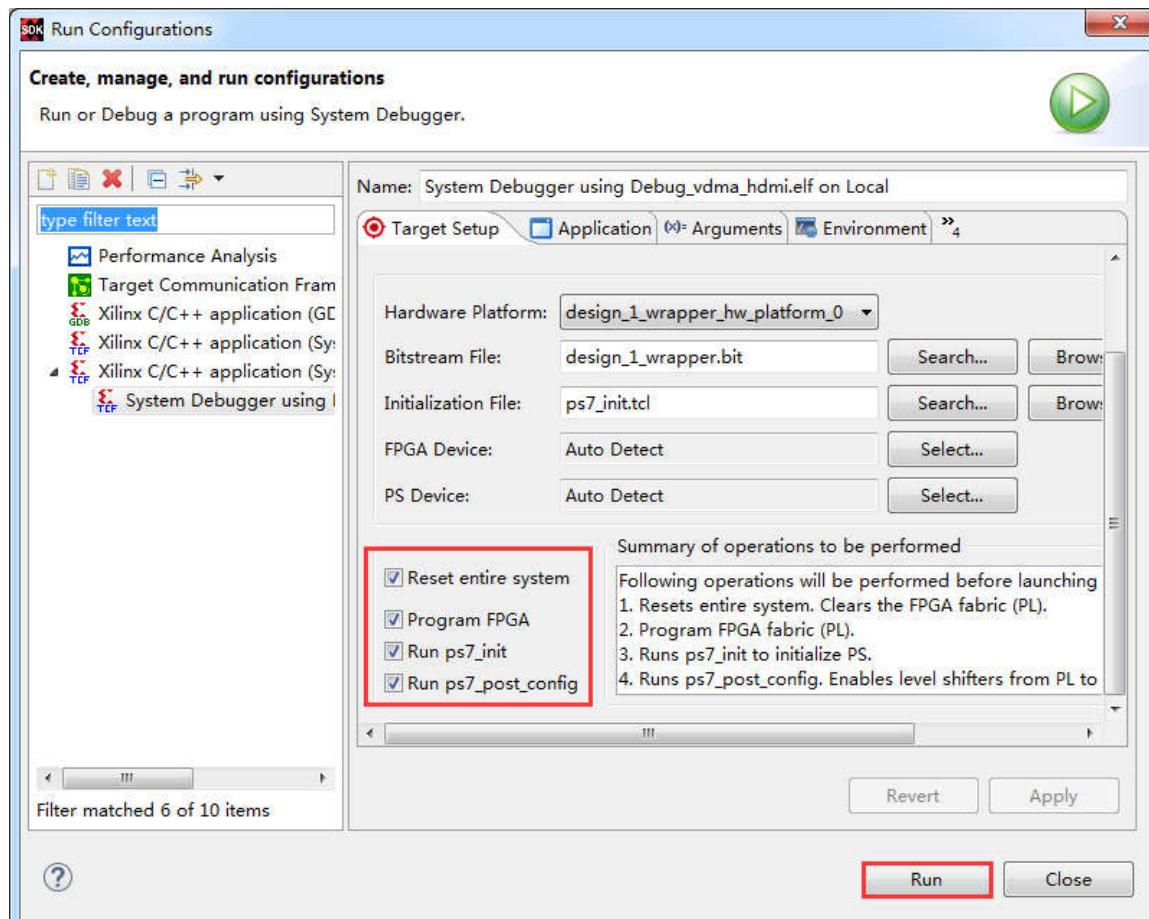
共享 新建文件夹

名称	修改日期	类型	大小
display_ctrl	2018/3/8 16:01	文件夹	
dynclk	2018/3/8 16:01	文件夹	
i2c	2018/3/8 16:01	文件夹	
display_demo.h	2018/3/8 16:54	C++ Header file	3 KB
lscript.ld	2018/3/8 15:31	LD 文件	6 KB
main.c	2018/3/8 16:53	C Source file	8 KB
pic_800_600.h	2017/1/14 22:27	C++ Header file	7,208 KB
Xilinx.spec	2018/3/8 15:31	SPEC 文件	1 KB

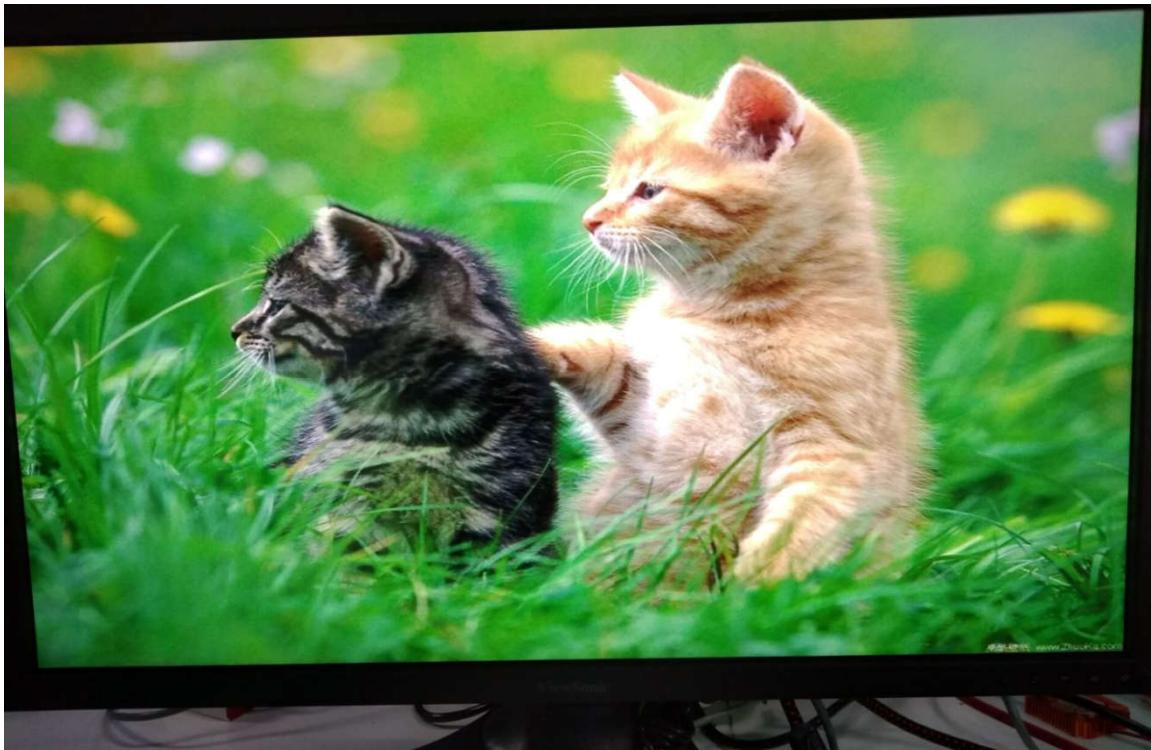
5) 在 SDK 下刷新



6) 连接 HDMI 输出端口到显示器，编译运行



7) 显示器显示出一幅图片



第十六章 固化程序

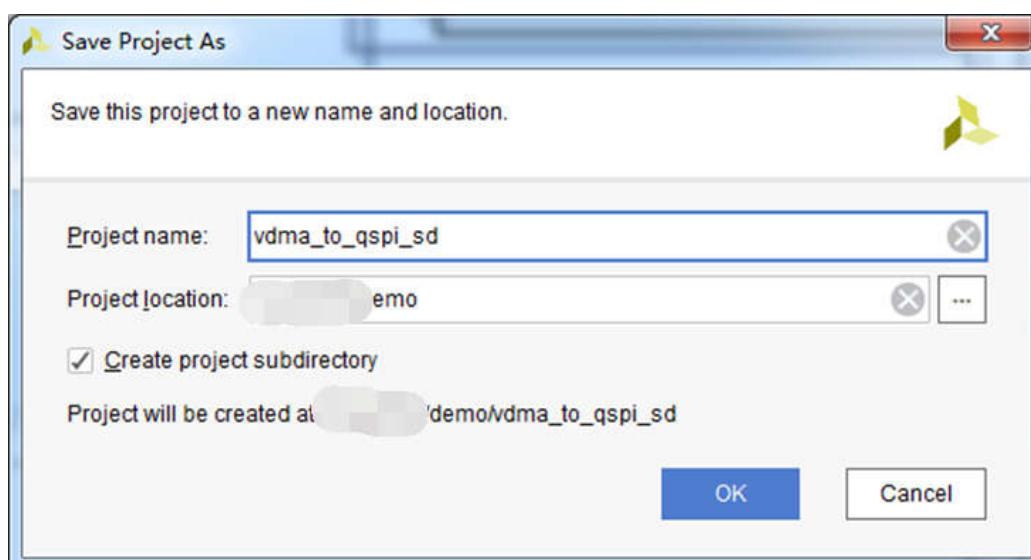
实验 Vivado 工程为 “vdma_to_qspi_sd”。

做个这么多试验，都是通过 JTAG 调试的，怎么把程序放在 SD 卡或者烧写到 QSPI Flash 里运行？首先 PL 必须有 PS 配置，所以不能像以往的 FPGA 烧录方法直接烧录到 Flash，本实验讲解如何固化程序。

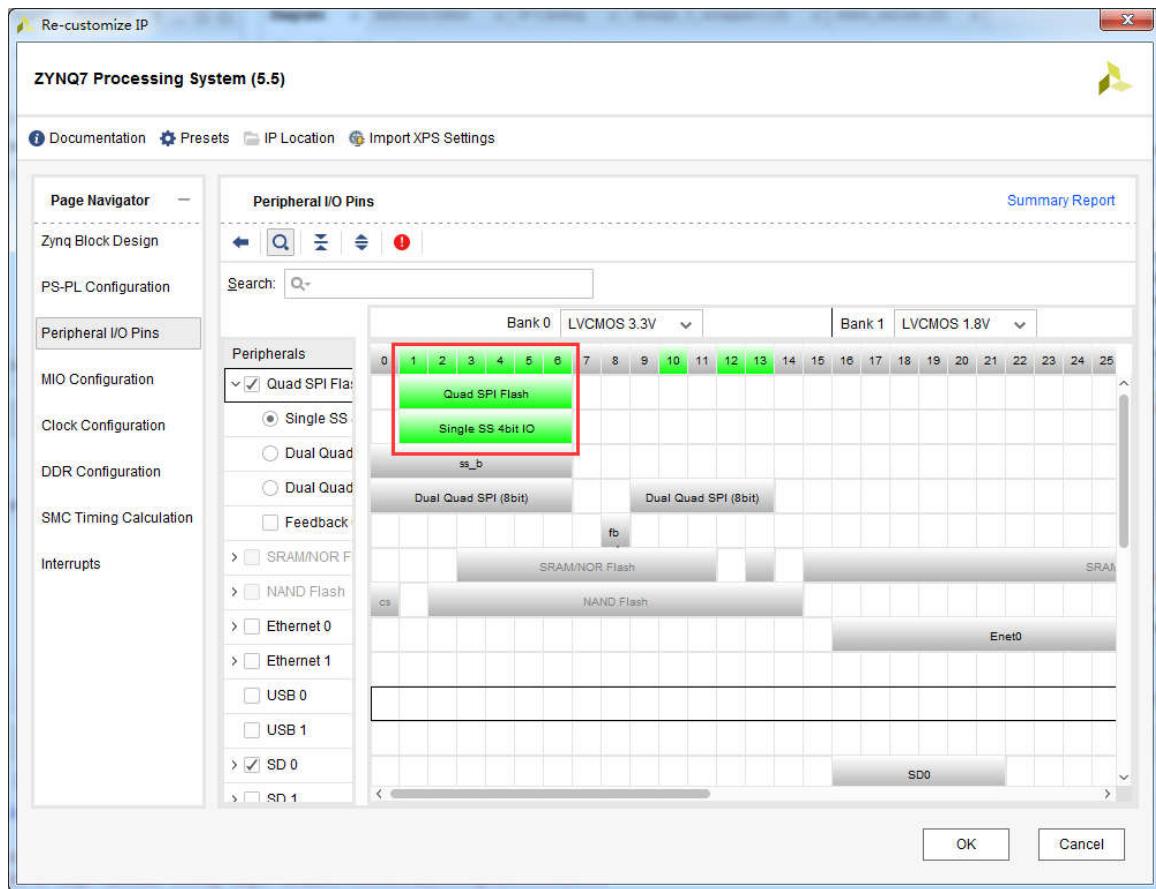
16.1 Vivado 工程建立

本实验选择 VDMA 测试工程来固化，在建立 VDMA 测试工程时，我们没有使能 QSPI 和 SD 卡，要固化程序必须使能 QSPI 或 SD 卡。

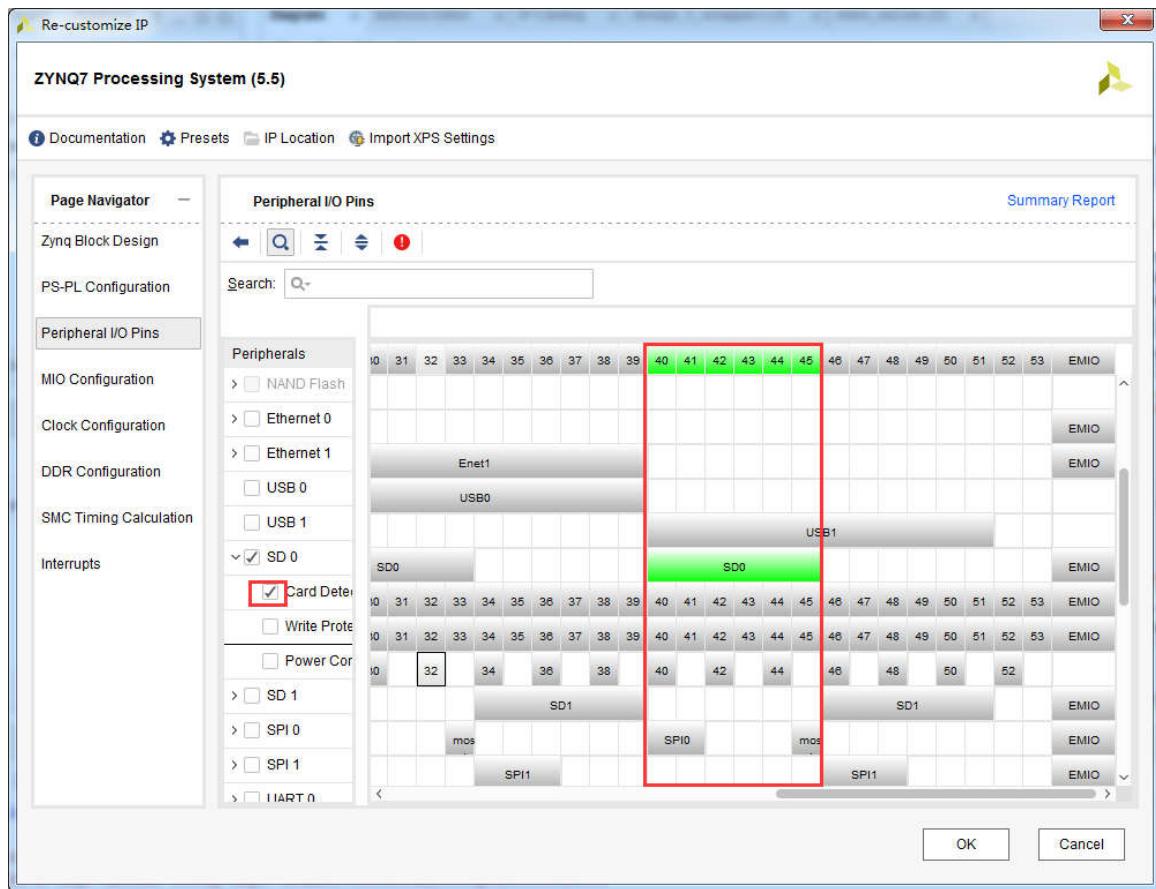
- 1) 将 VDMA 测试工程另存一份，用于固化程序实验，名称改为 “vdma_to_qspi_sd”



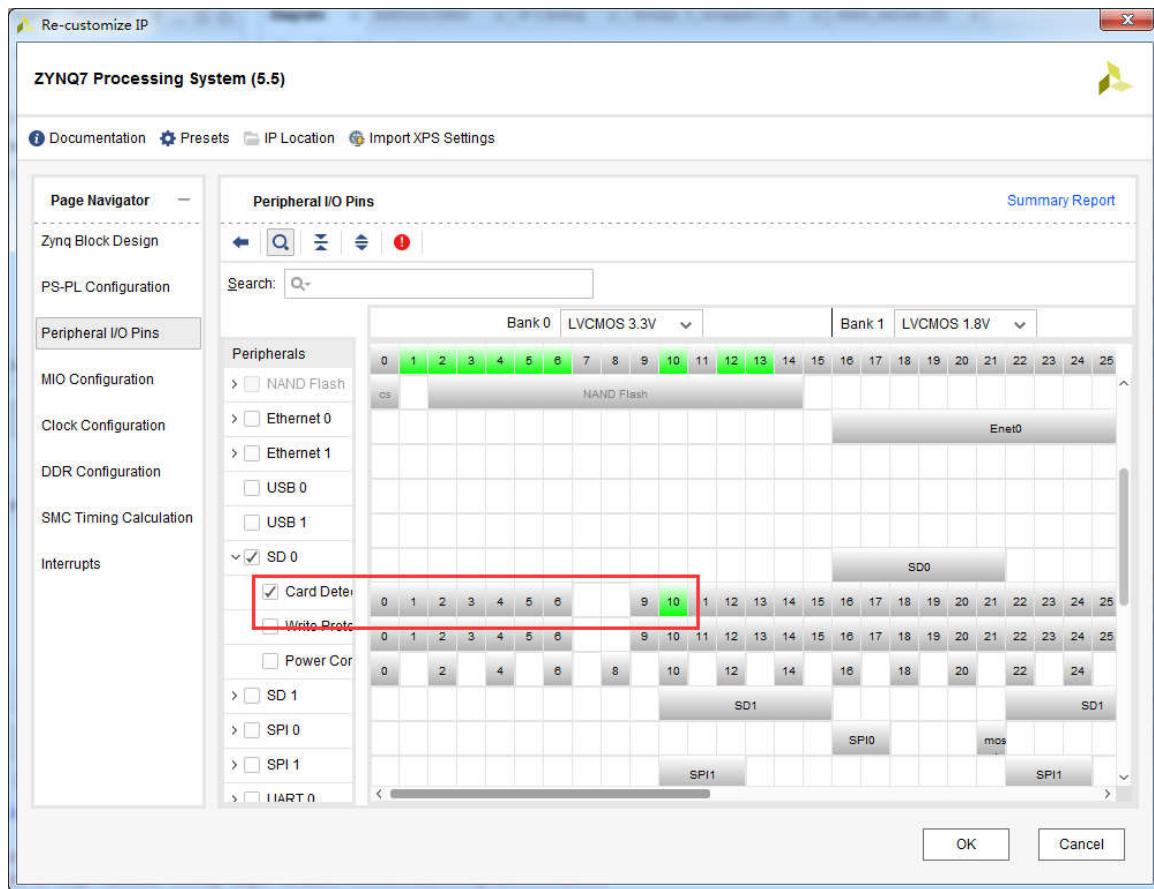
- 2) 添加 QSPI，使用 MIO1-6



- 3) 添加 SD0 控制器，使用 MIO40-45，使用 TF 卡接口



4) 添加 SD 卡的卡检测管脚 MIO10

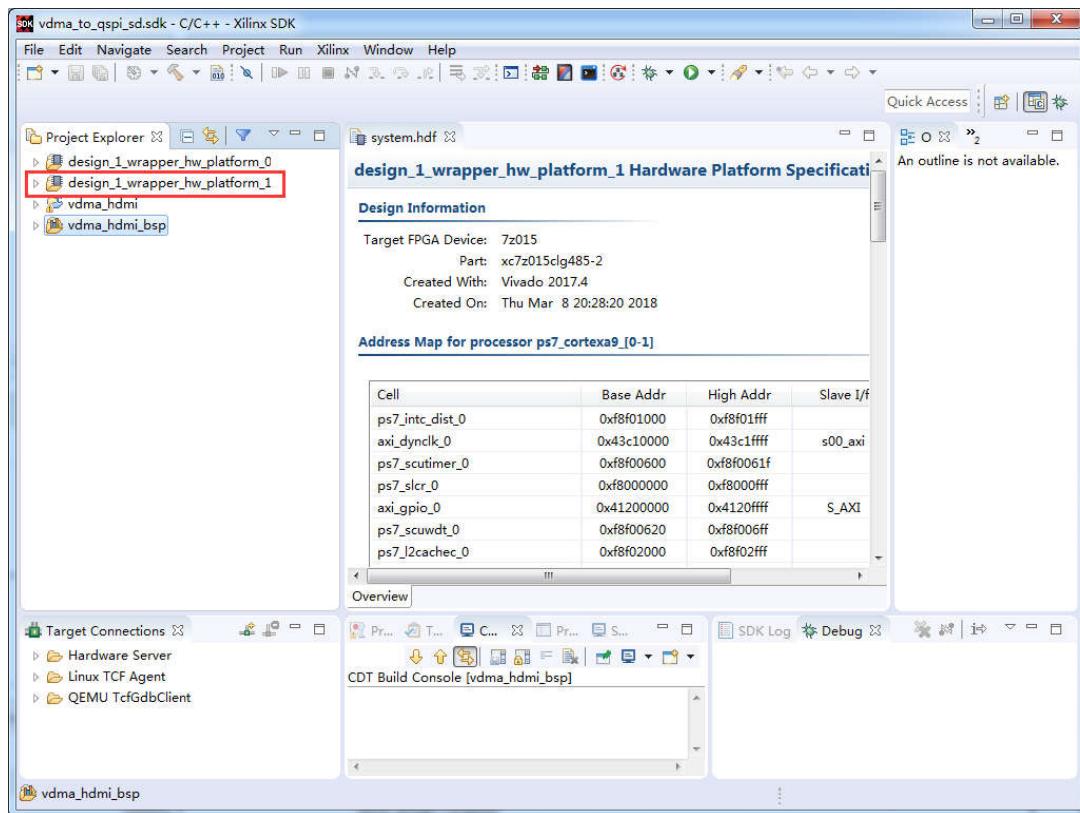


- 5) 保存设计，编译生成 bit 文件，再次导出硬件

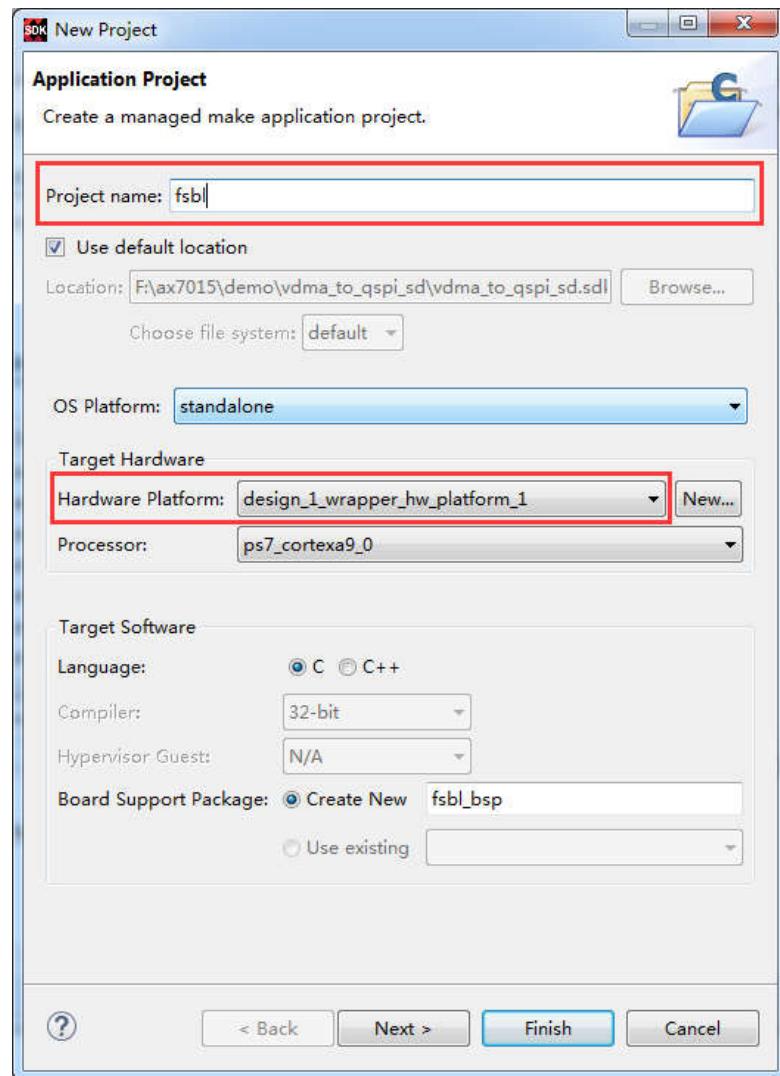
16.2 生成 FSBL

FSBL 是一个二级引导程序，完成 MIO 的分配、DDR 控制器初始化、SD、QSPI 控制器初始化，配置 FPGA，然后加载用户程序。

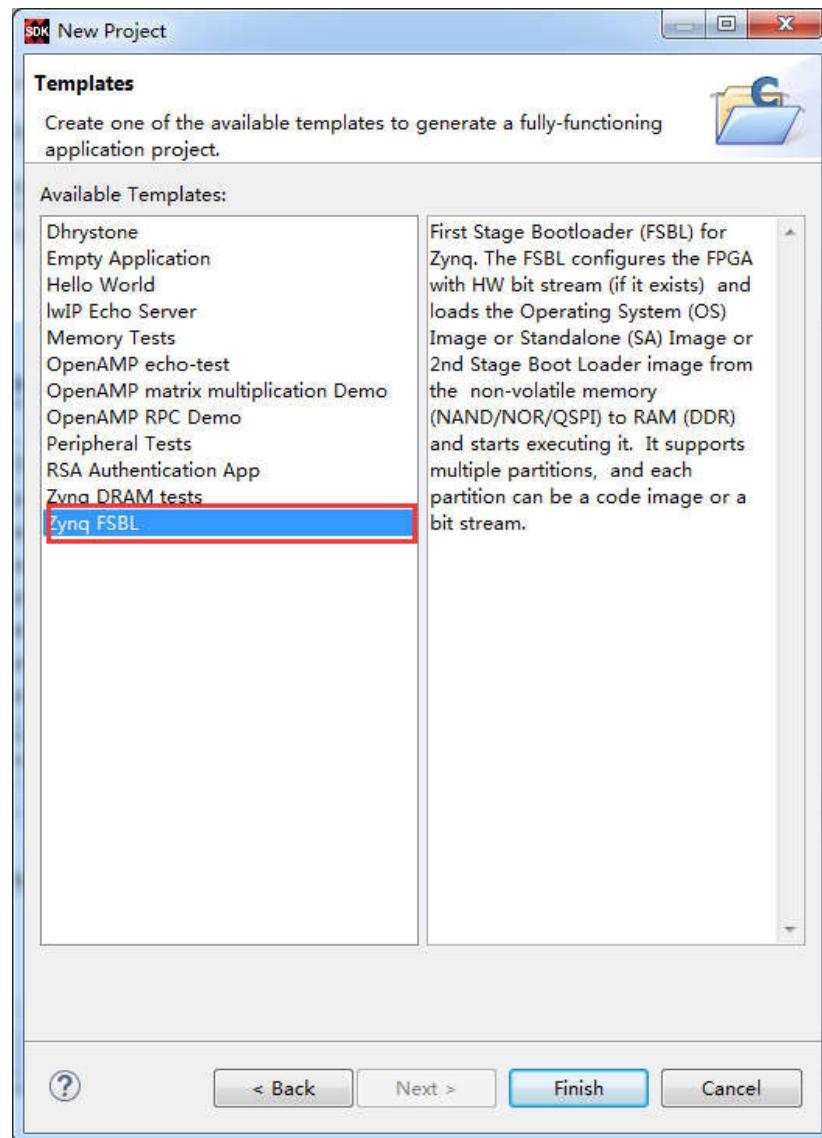
- 1) 启动 SDK 软件，由于是从其他工程复制而来，以前就是 SDK 工程，由于路径变化，又多出一个 hw_platform_1



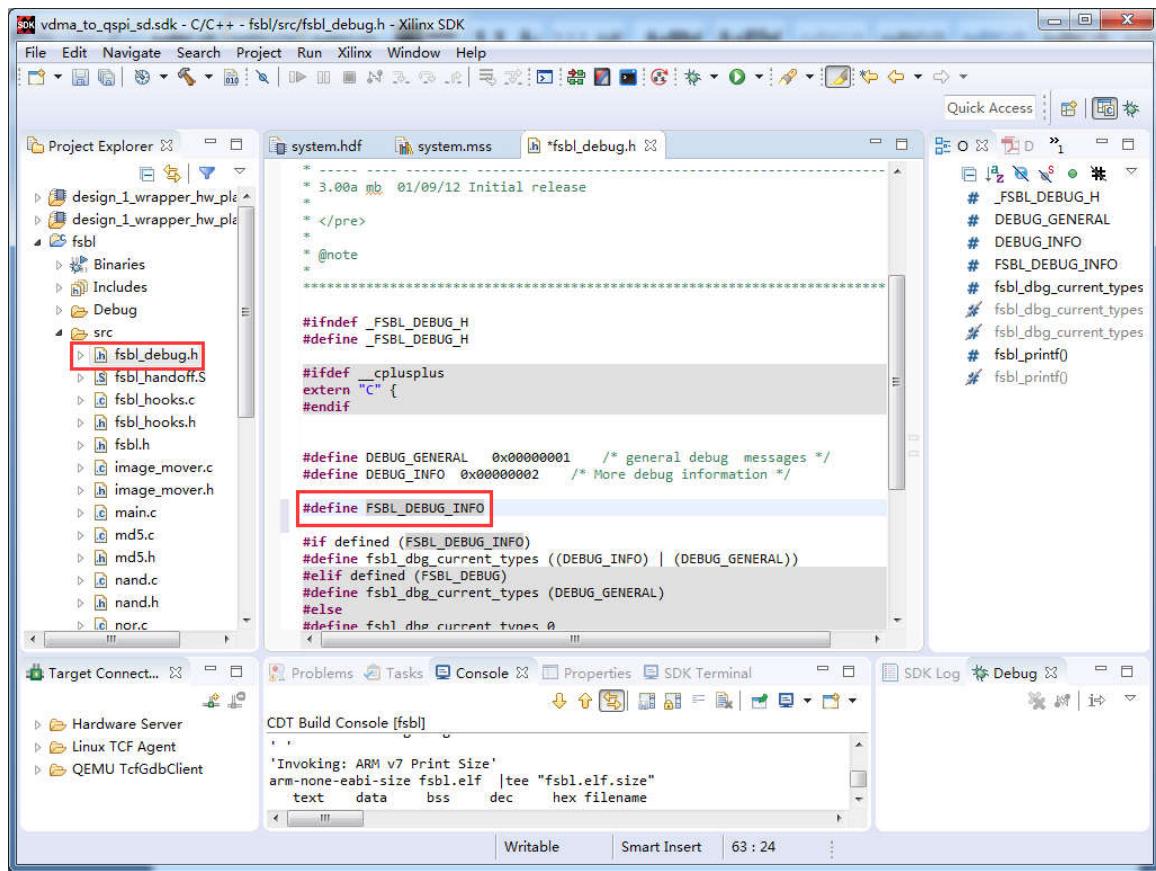
- 2) 新建一个名为 fsbl 的 APP , 特别注意硬件平台选择最新的那个



3) 模板选择 Zynq FSBL



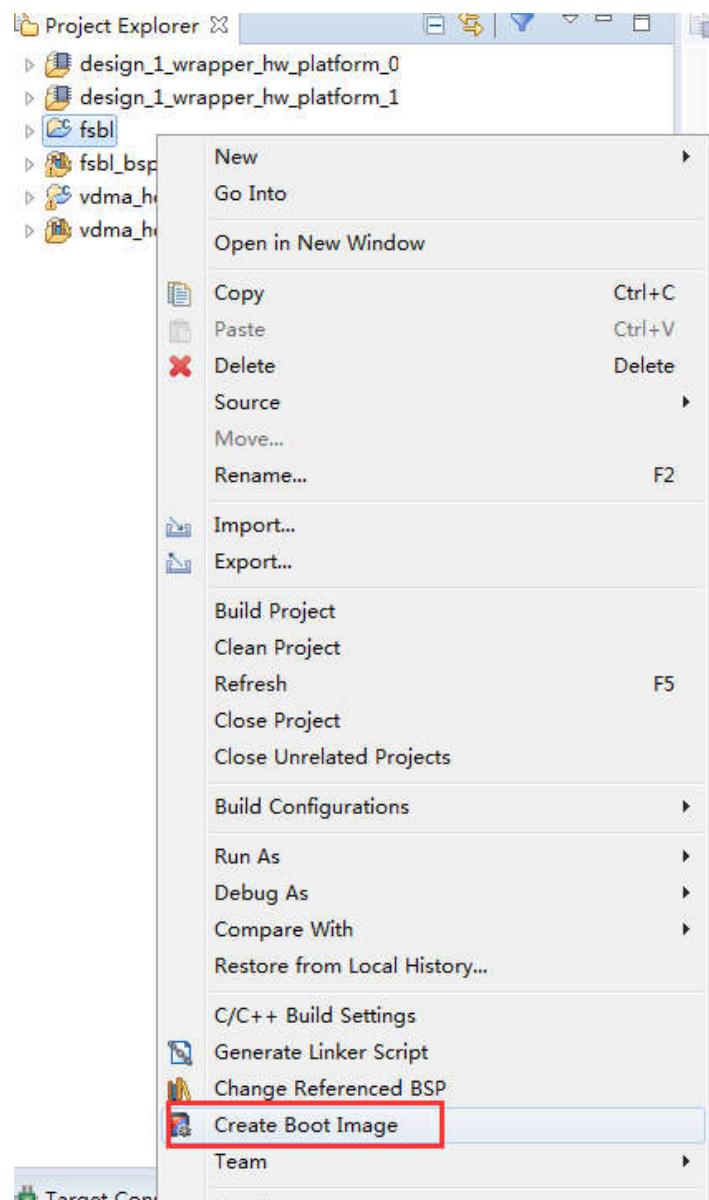
- 4) 添加调试宏定义 `FSBL_DEBUG_INFO`，可以在启动输出 FSBL 的一些状态信息，有利于调试，但是会导致启动时间变长。



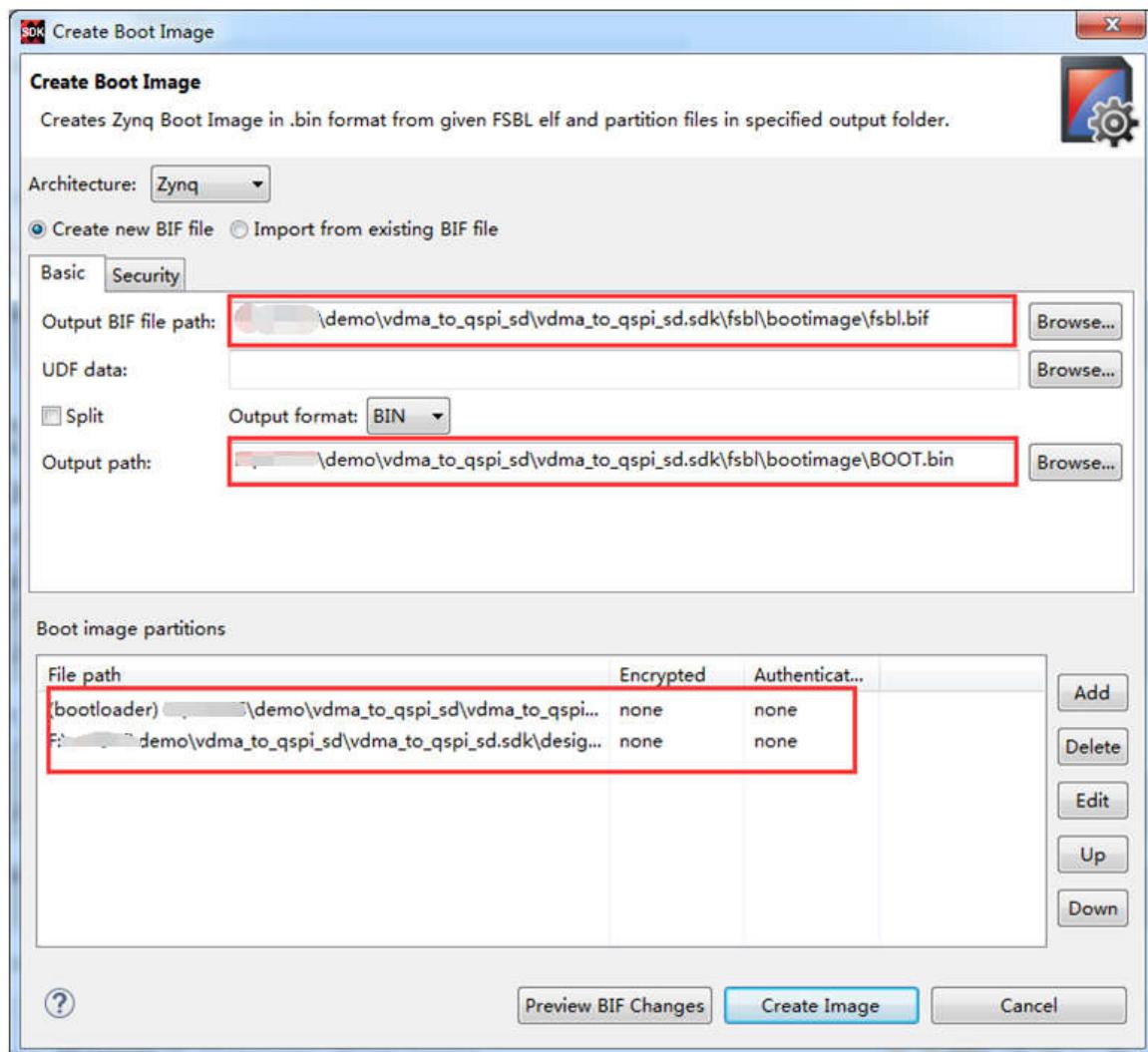
5) SDK 默认会自动编译，生成 fsbl.elf 文件

16.3 创建 BOOT 文件

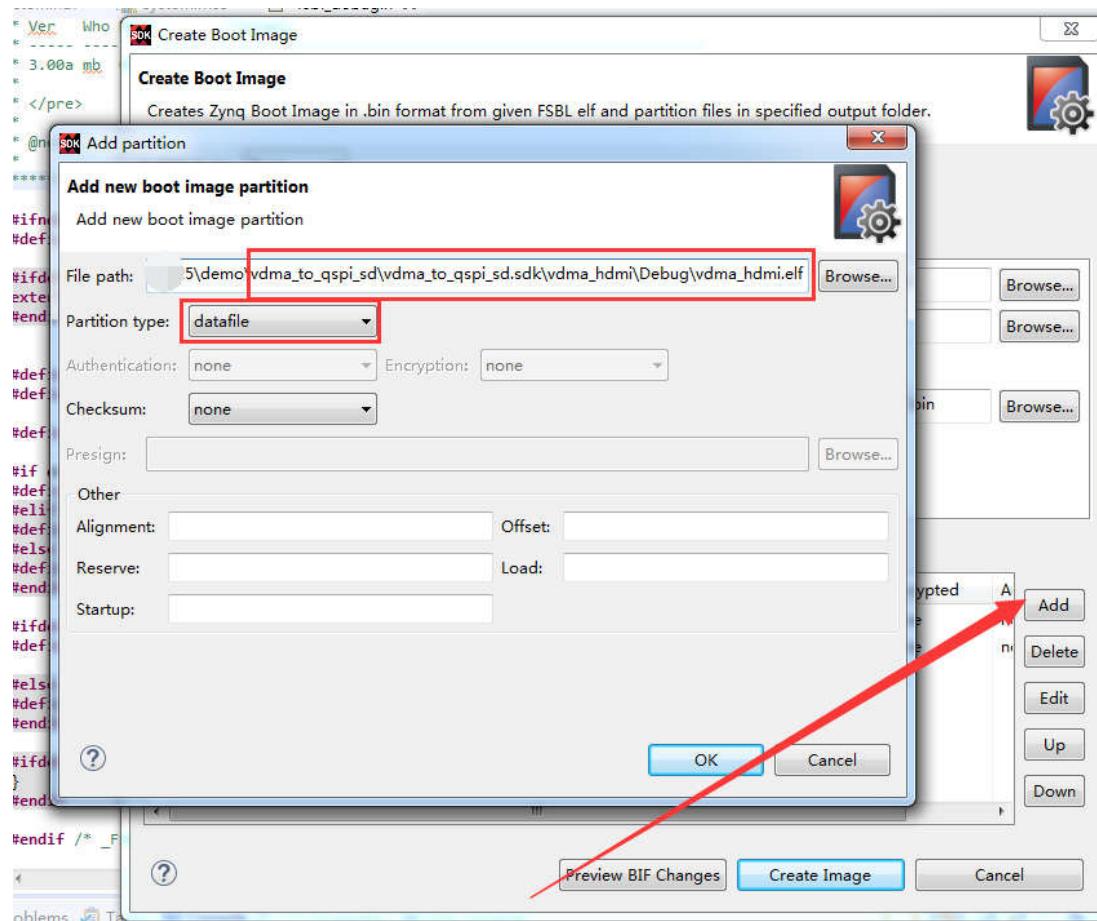
1) 选择 fsbl 工程，右键选择 Create Boot Image



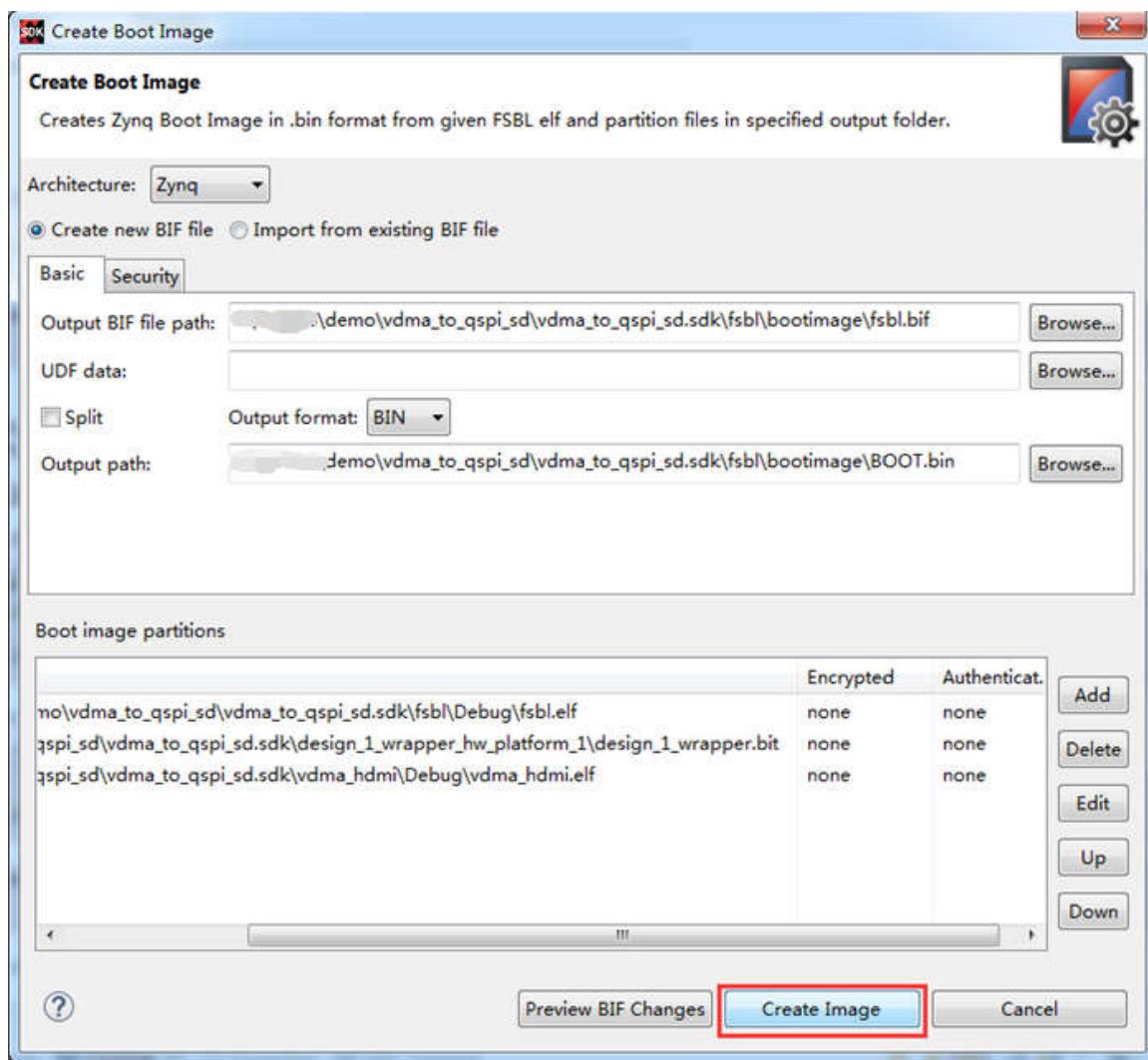
- 2) 弹出的窗口中可以看到生成的 BIF 文件路径，BIF 文件是生成 BOOT 文件的配置文件，还有生成的 BOOT.bin 文件路径，BOOT.bin 文件是我们需要的启动文件，可以放到 SD 卡启动，也可以烧写到 QSPI Flash。



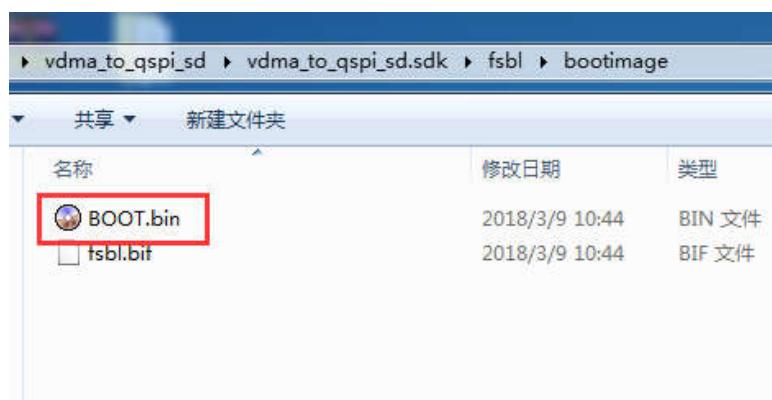
- 3) 在 Boot image partitions 列表中有要合成的文件，第一个文件一定是 bootloader 文件，就是上面生成的 fsbl.elf 文件，第二个文件是 FPGA 配置文件，现在点击 Add 添加我们的 VDMA 测试程序 vdma_hdmi.elf



4) 点击 Create Image 生成

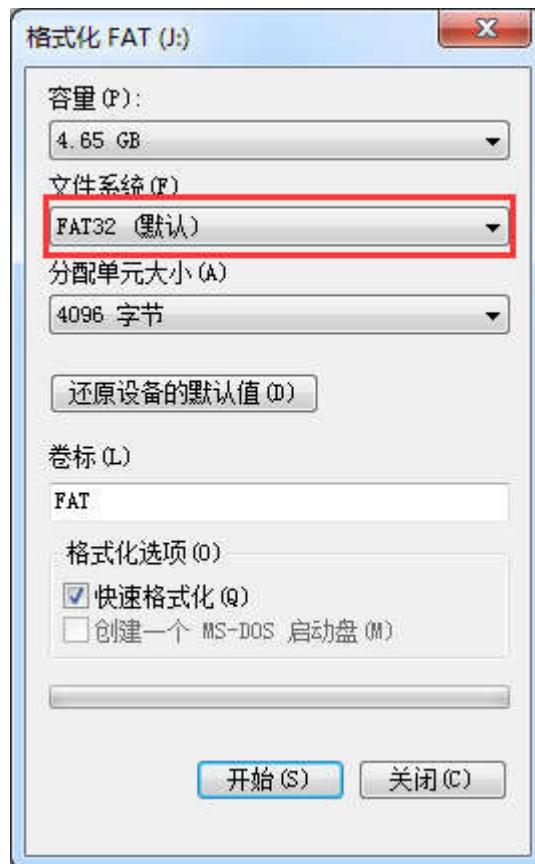


5) 在生成的目录下可以找到 BOOT.bin 文件



16.4 SD 卡启动测试

1) 格式化 SD 卡，只能格式化为 FAT32 格式，其他格式无法启动



2) 放入 BOOT.bin 文件，放在根目录



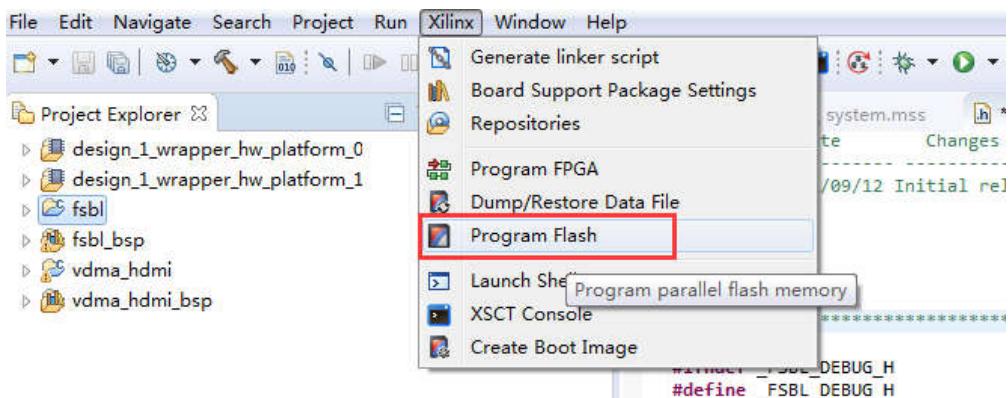
3) SD 卡插入开发板的 SD 卡插槽

4) 启动模式调整为 SD 卡启动

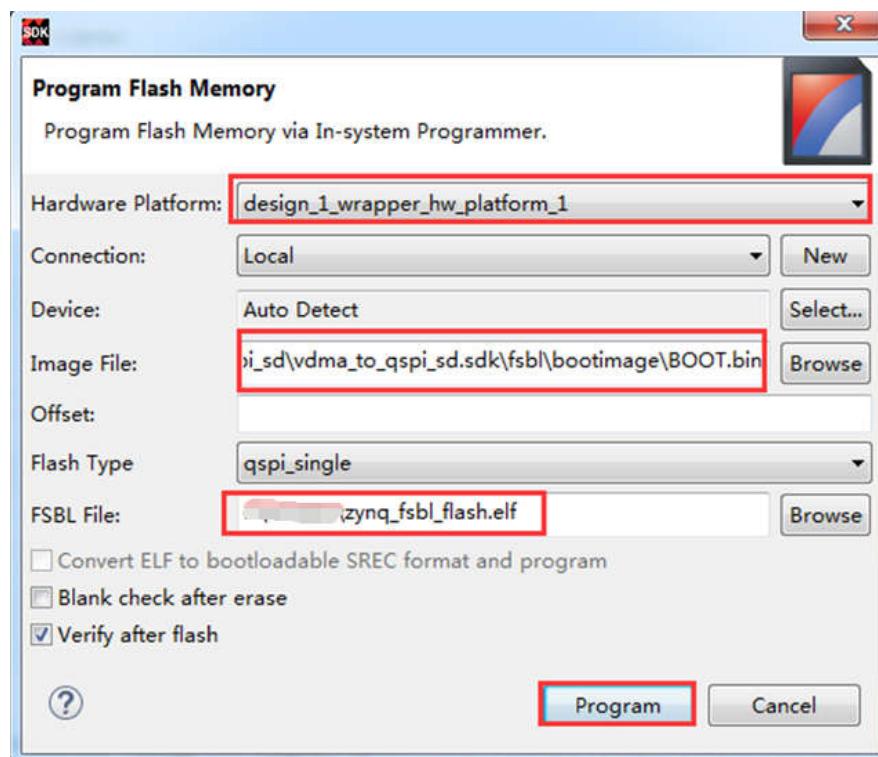
5) 插上 HDMI 显示器，给开发板上电，可以看到显示器显示了小猫的图片

16.5 QSPI 启动测试

1) 在 SDK 菜单 Xilinx -> Program Flash



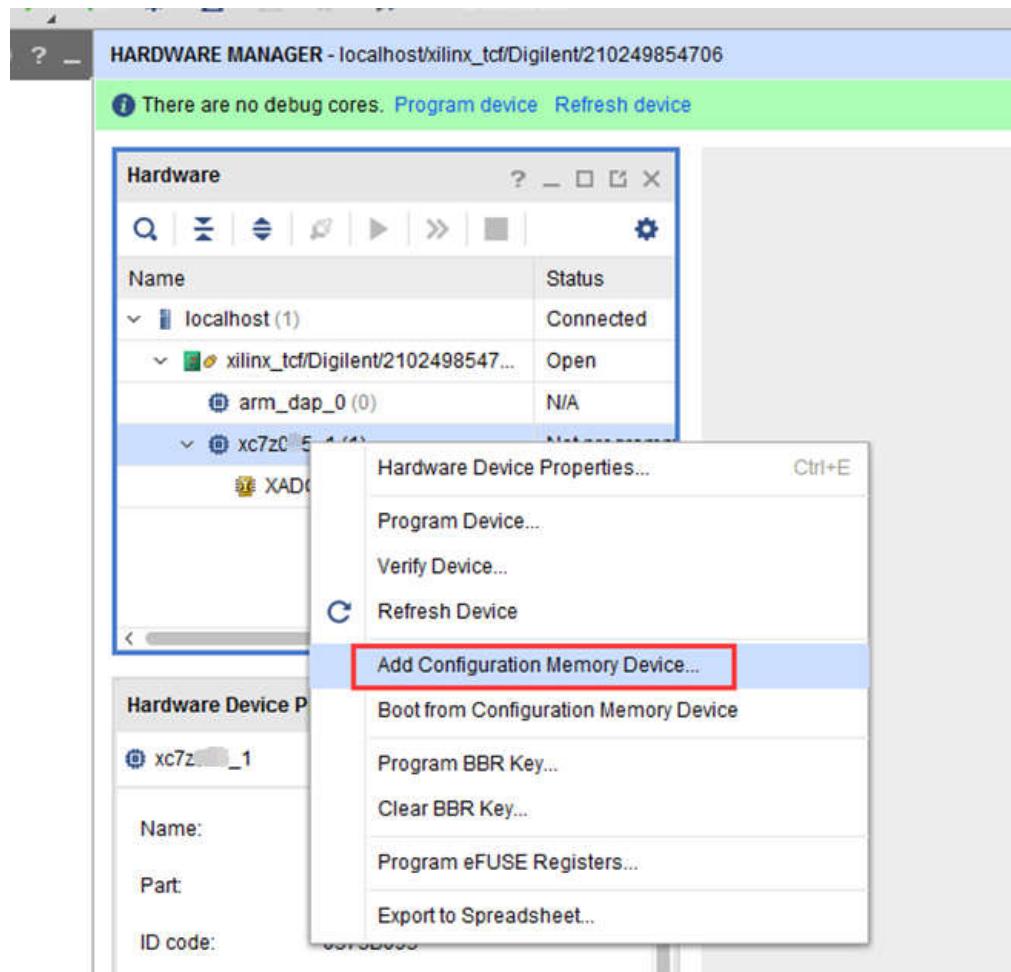
- 2) Hardware Platform 选择最新的，Image File 文件选择要烧写的 BOOT.bin，FSBL file 选择芯驿电子定制的特别版本 fsbl.elf，只有用这个 fsbl 才能烧写。
- 3)



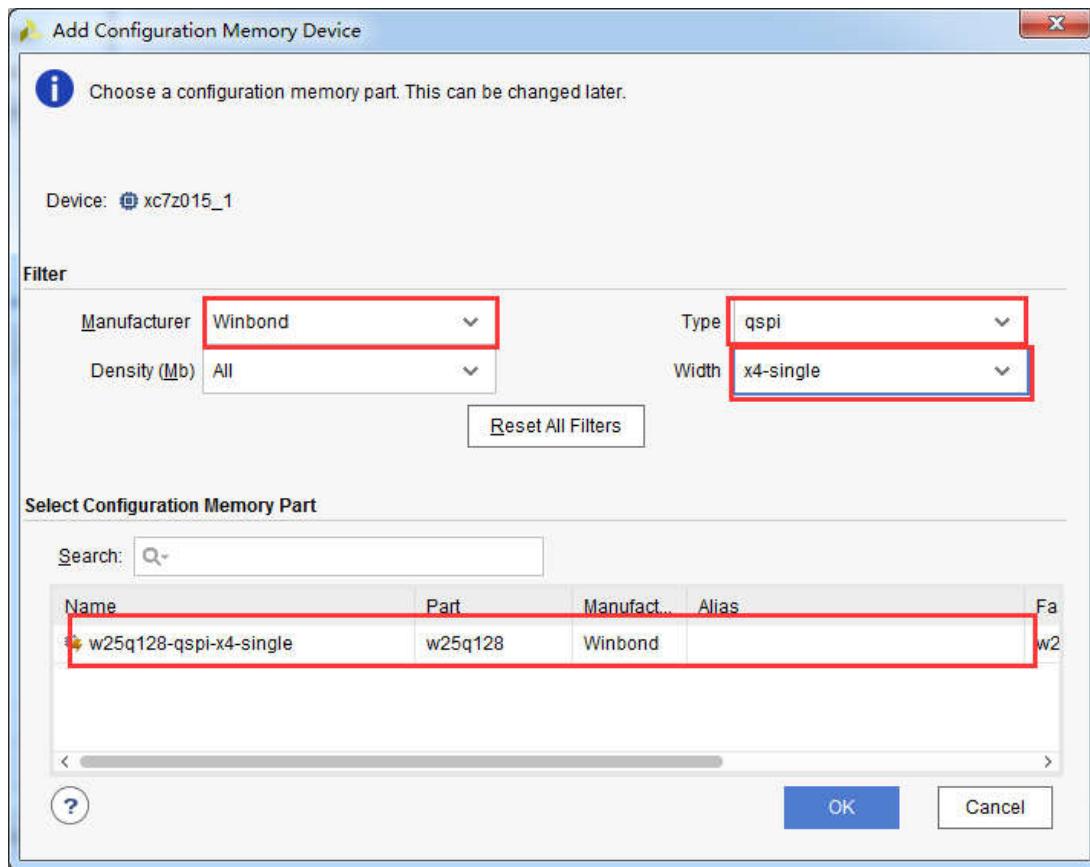
- 4) 点击 Program 等待烧写完成
- 5) 设置启动模式为 QSPI，再次启动，可以看到显示器有显示输出

16.6 Vivado 下烧写 QSPI

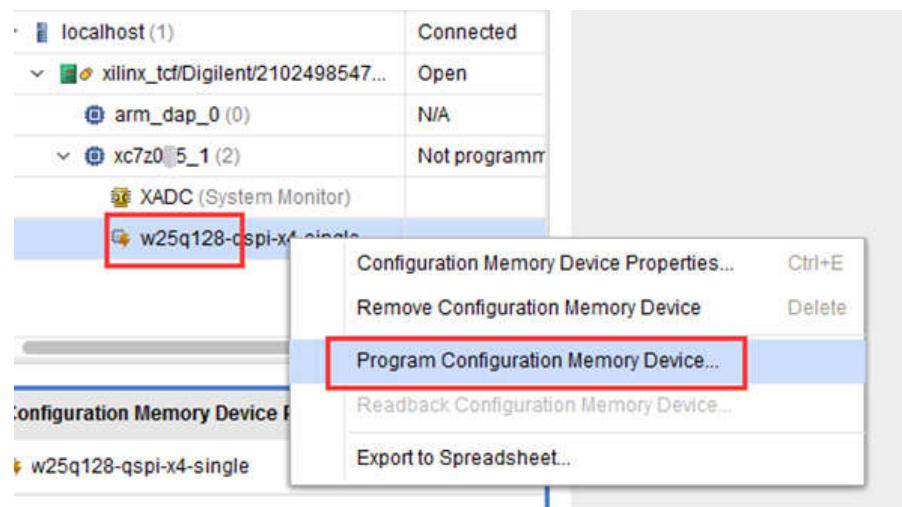
- 1) 在 HARDWARE MANGER 下选择器件，右键 Add Configuration Memory Device



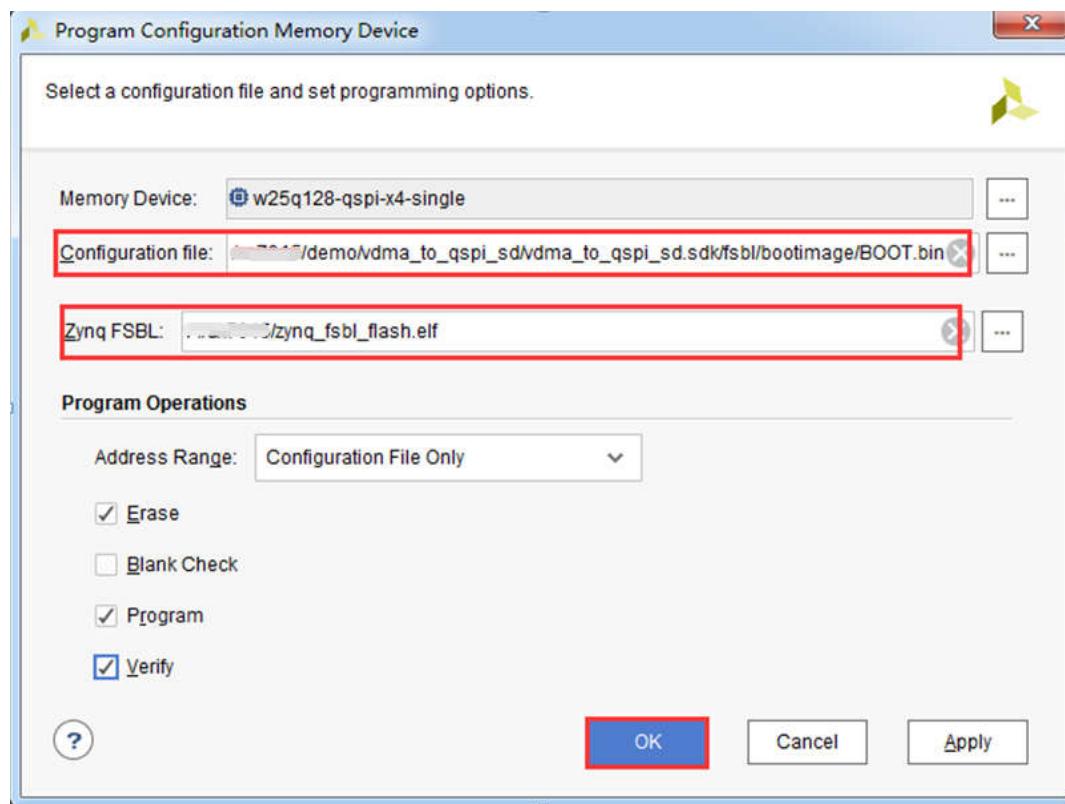
- 2) 选择尝试 Winbond , 类型选择 qspi , 宽度选择 x4-single , 这时候出现一个 w25q128 , 开发板使用 w25q256 , 但是不影响烧录。



- 3) 右键选择编程文件



- 4) 选择要烧写的文件和芯驿电子定制的 fsbl 文件 , 就可以烧写了 , 如果烧写时不是 JTAG 启动模式 , 软件会给出一个警告 , 所以建议烧写 QSPI 的时候设置到 JTAG 启动模式



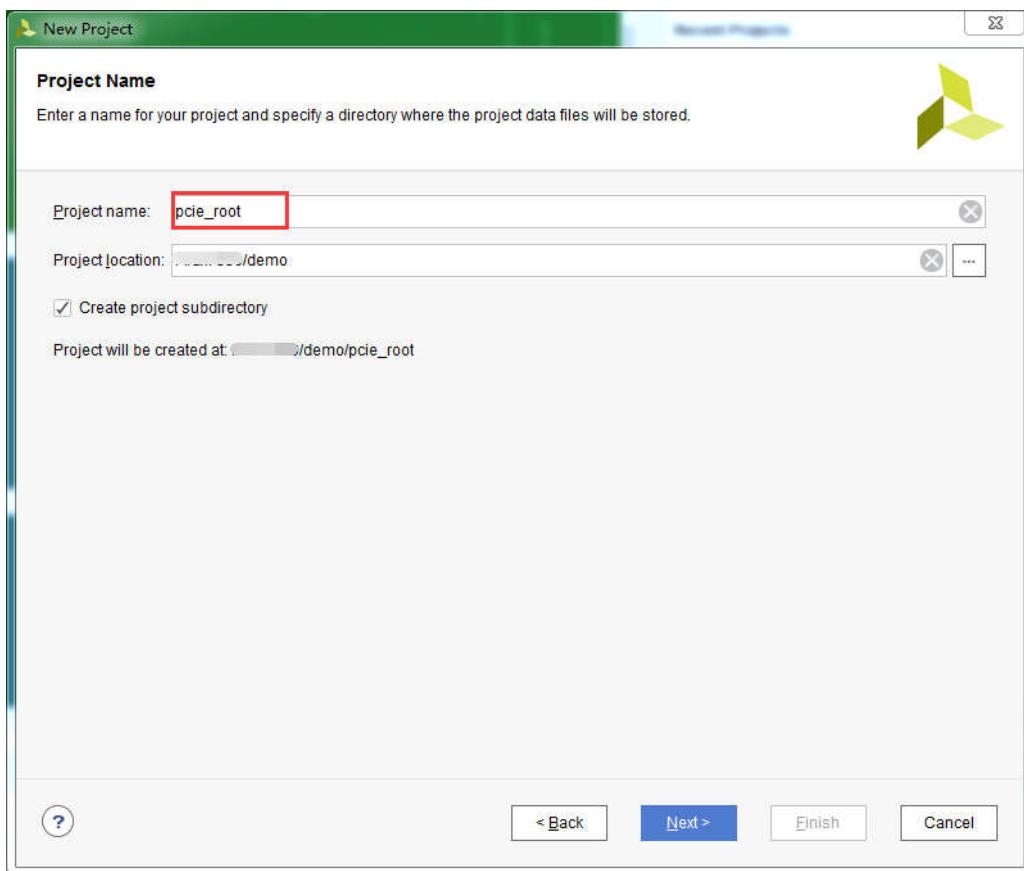
第十七章 PCIe ROOT 枚举测试

实验 Vivado 工程为 “pcie_root”。

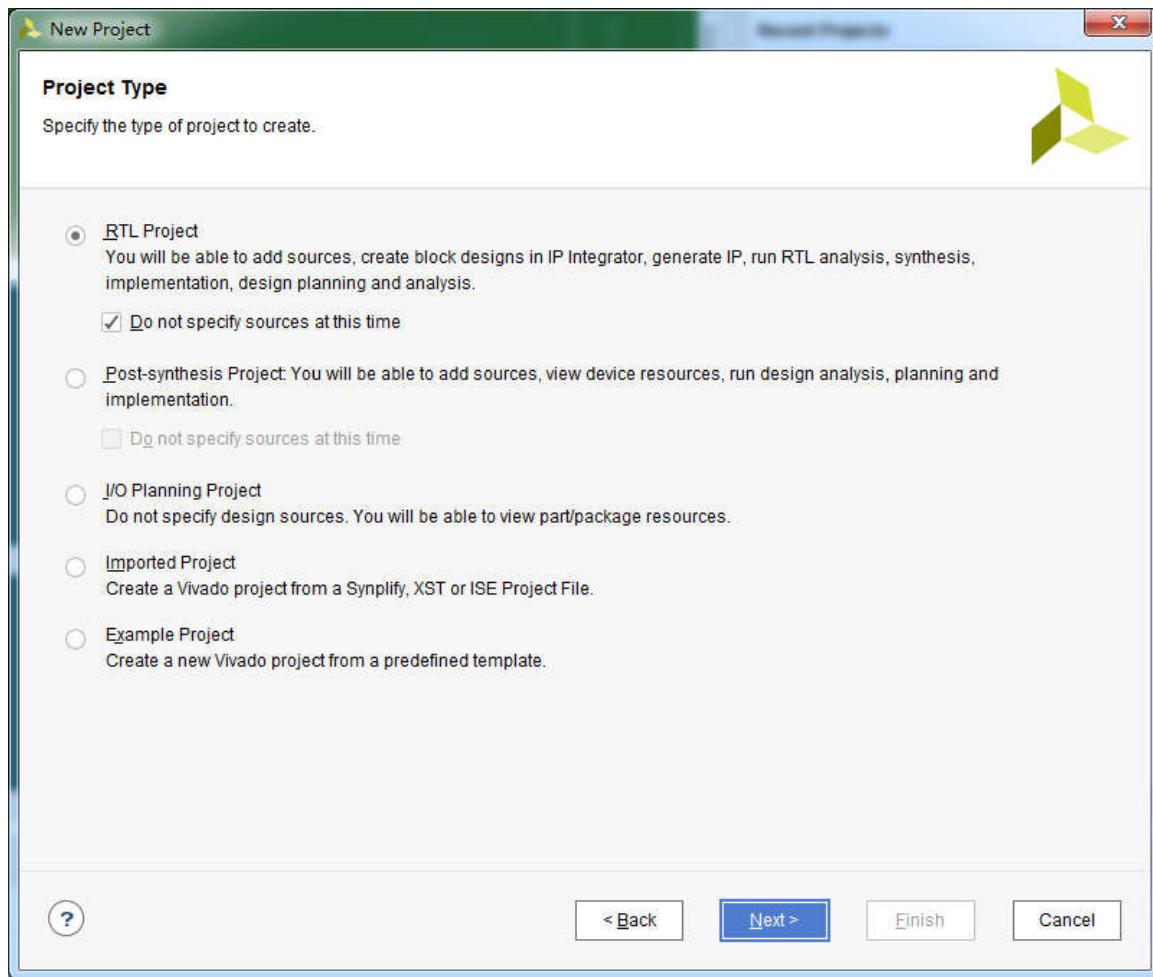
17.1 Vivado 工程建立

17.1.1 创建新工程

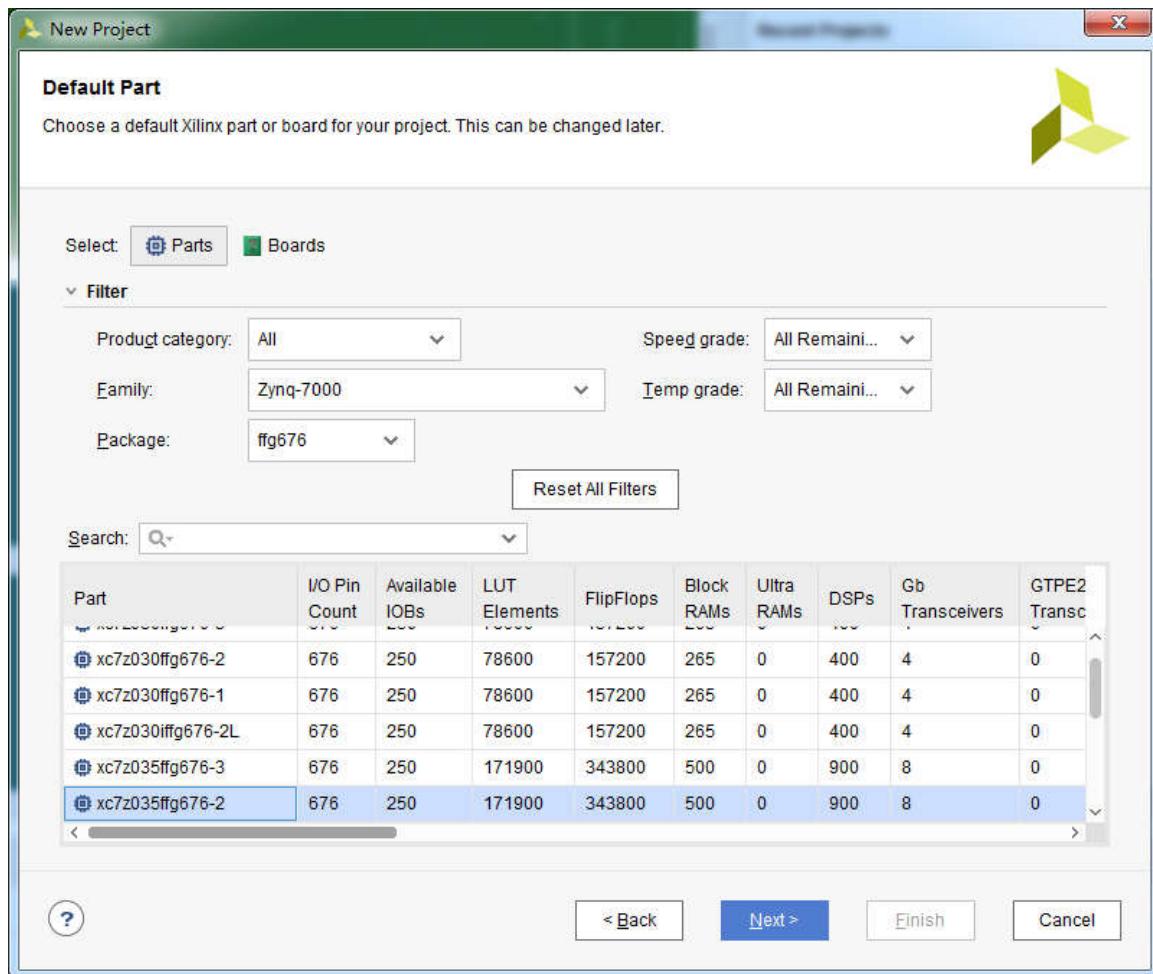
- 1) 新建一个工程名称为 “pcie_root” ,选择新建一个文件夹 , 点击 “Next”



- 2) 工程类型选择 “RTL Project” , 同时选择 “Do not specify sources at this time” , 点击 “Next”



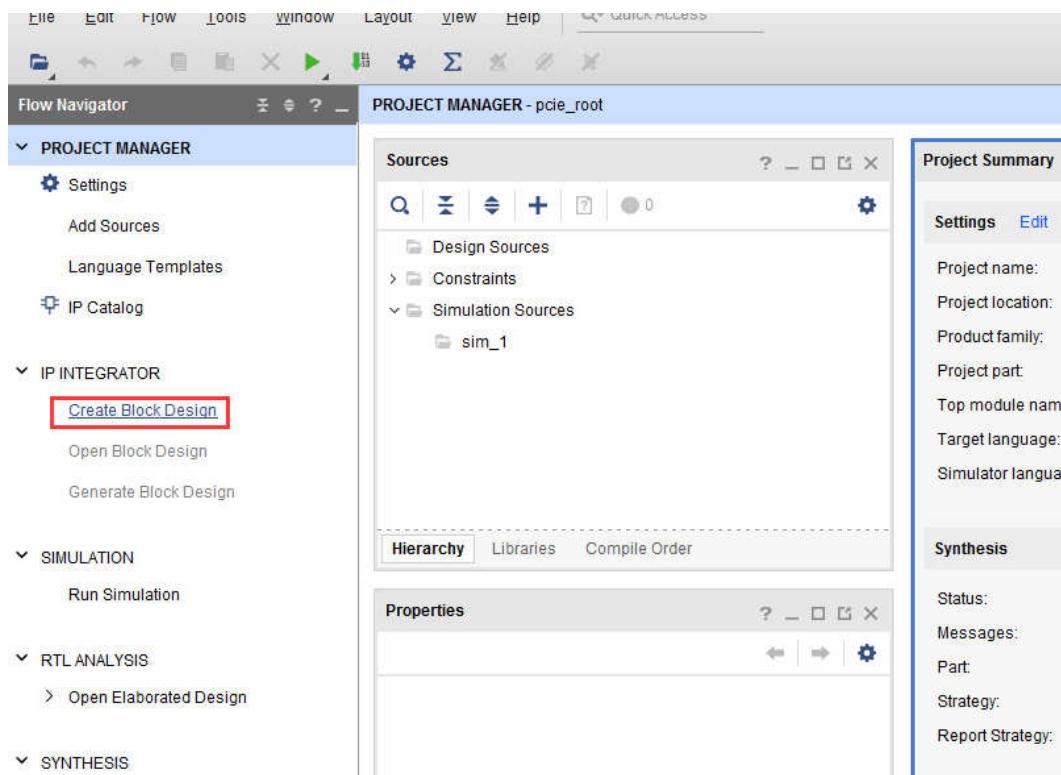
- 3) “Parts” 器件家族 (Family) 选择 “Zynq-7000” , 封装 (Package) 选择 “ffg676” , 然后选择 “xc7z035ffg676-2” , 点击 “Next”



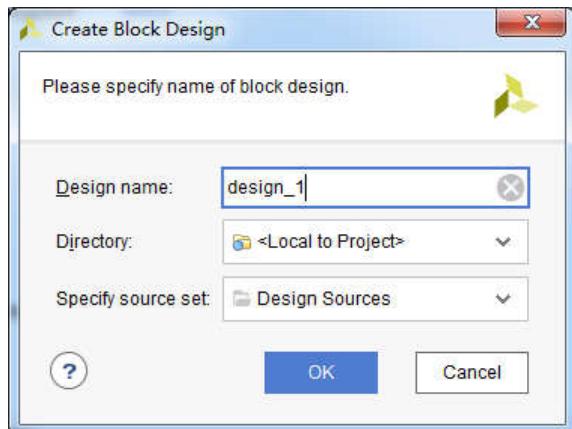
4) 点击“Finish”完成工程向导

17.1.2 创建 block 设计

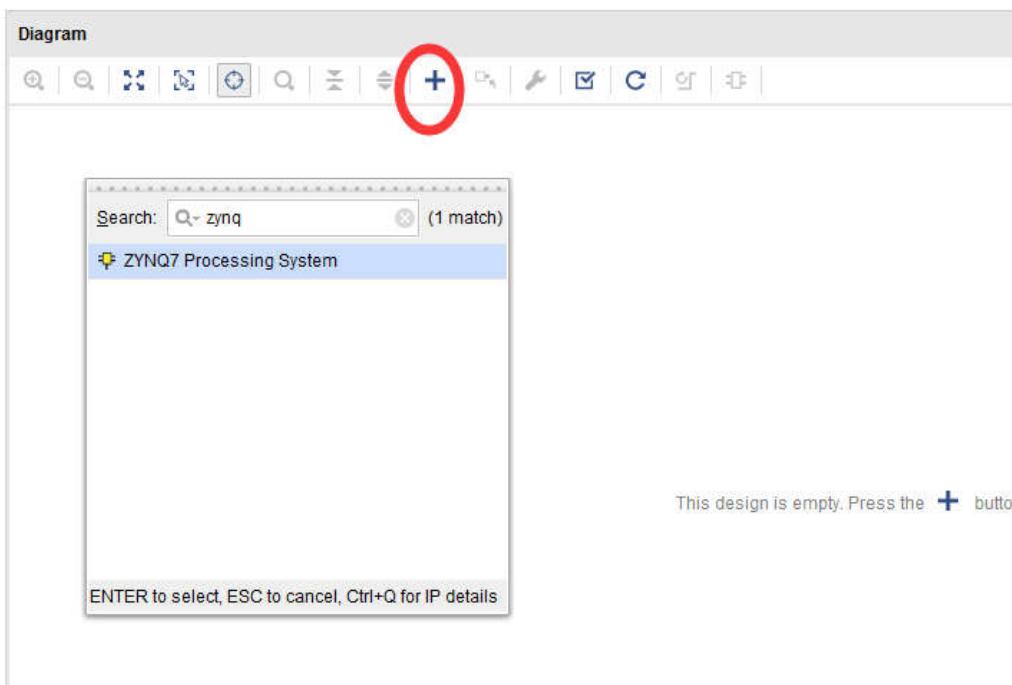
1) 从 Vivado 流程导航窗口中点击“Create Block Design”



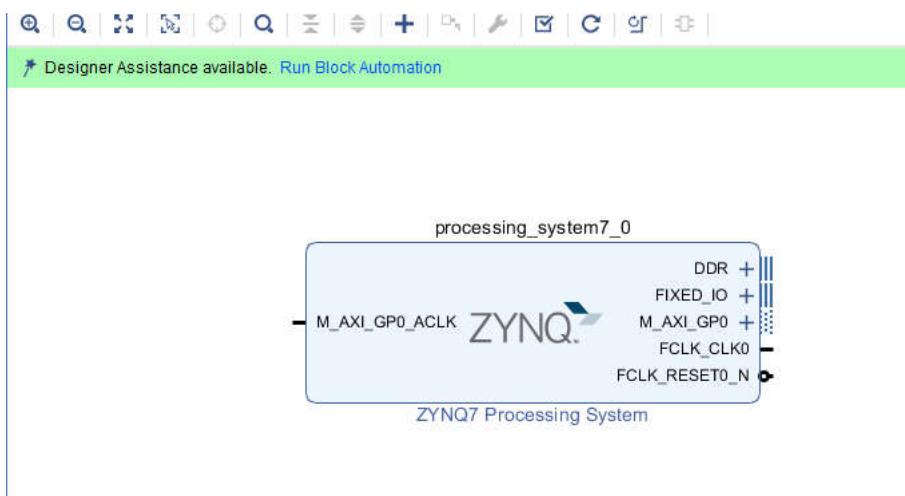
- 2) 给 Block 设计起名字，这里保持 “design_1” 的默认名称，点击 “OK”



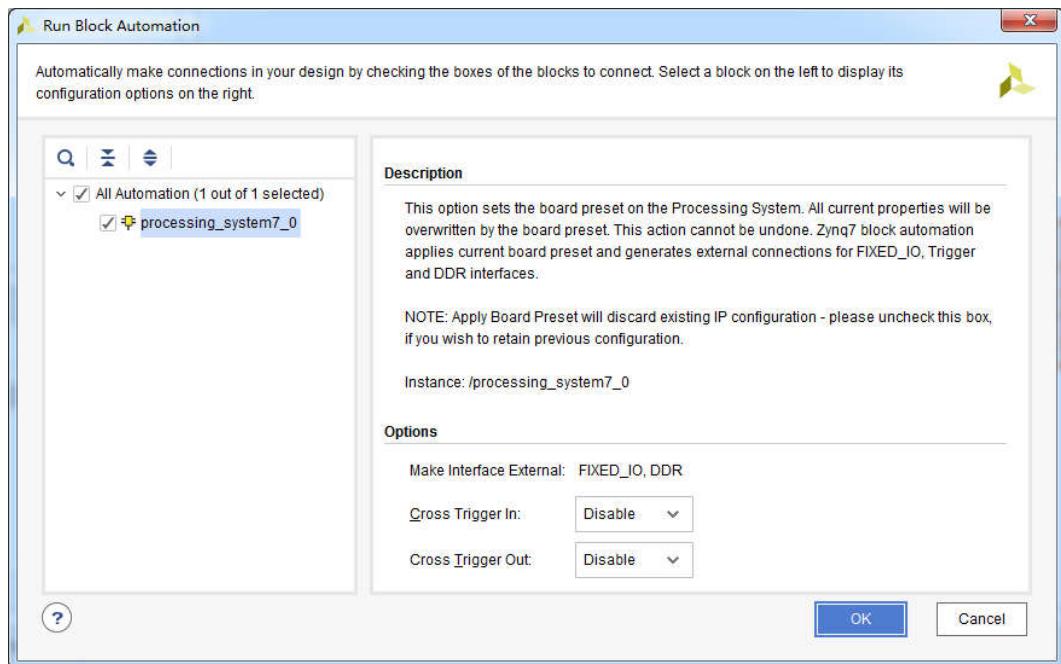
- 3) 空白设计打开后，点击 “Add IP” 图标，在 IP 仓库中选择双击 “ZYNQ7 Processing System”



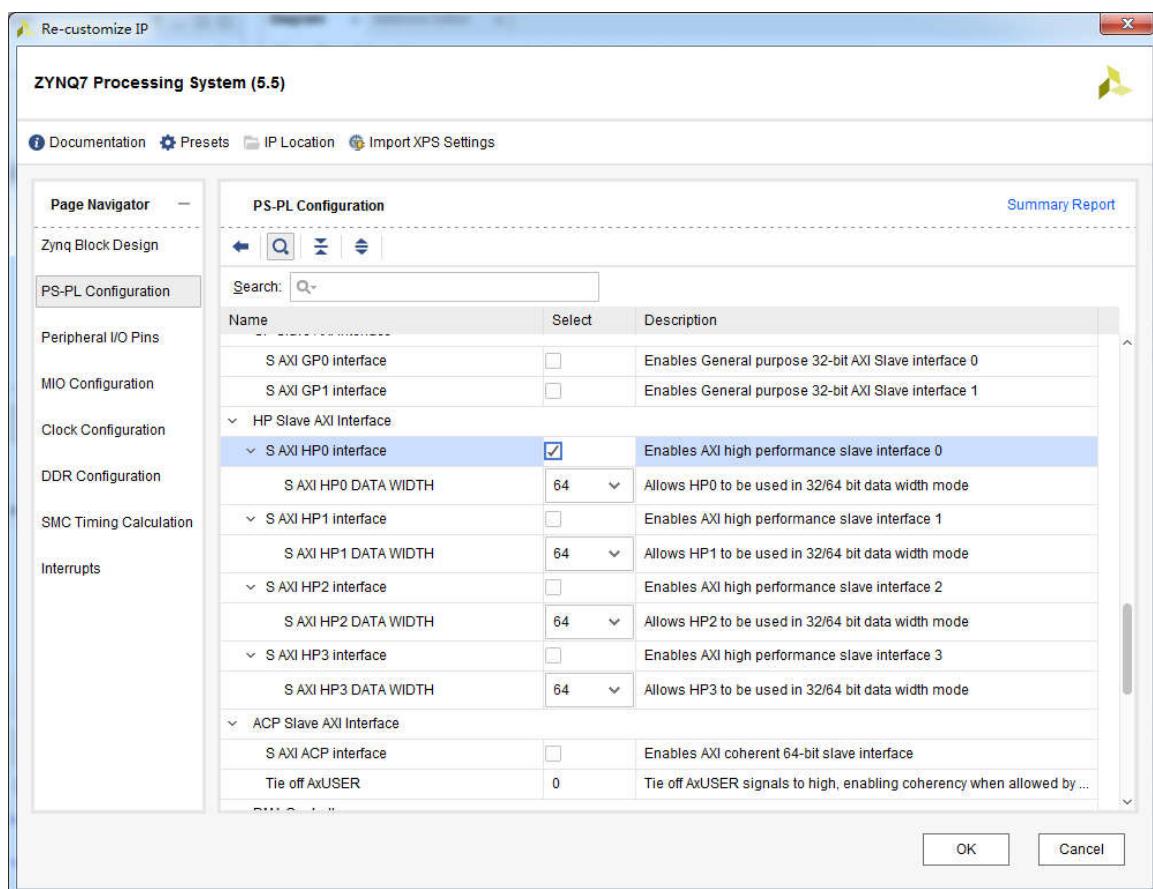
- 4) ZYNQ PS block 被添加到 block 设计中，点击 “Run Block Automation” 来配置目标硬件



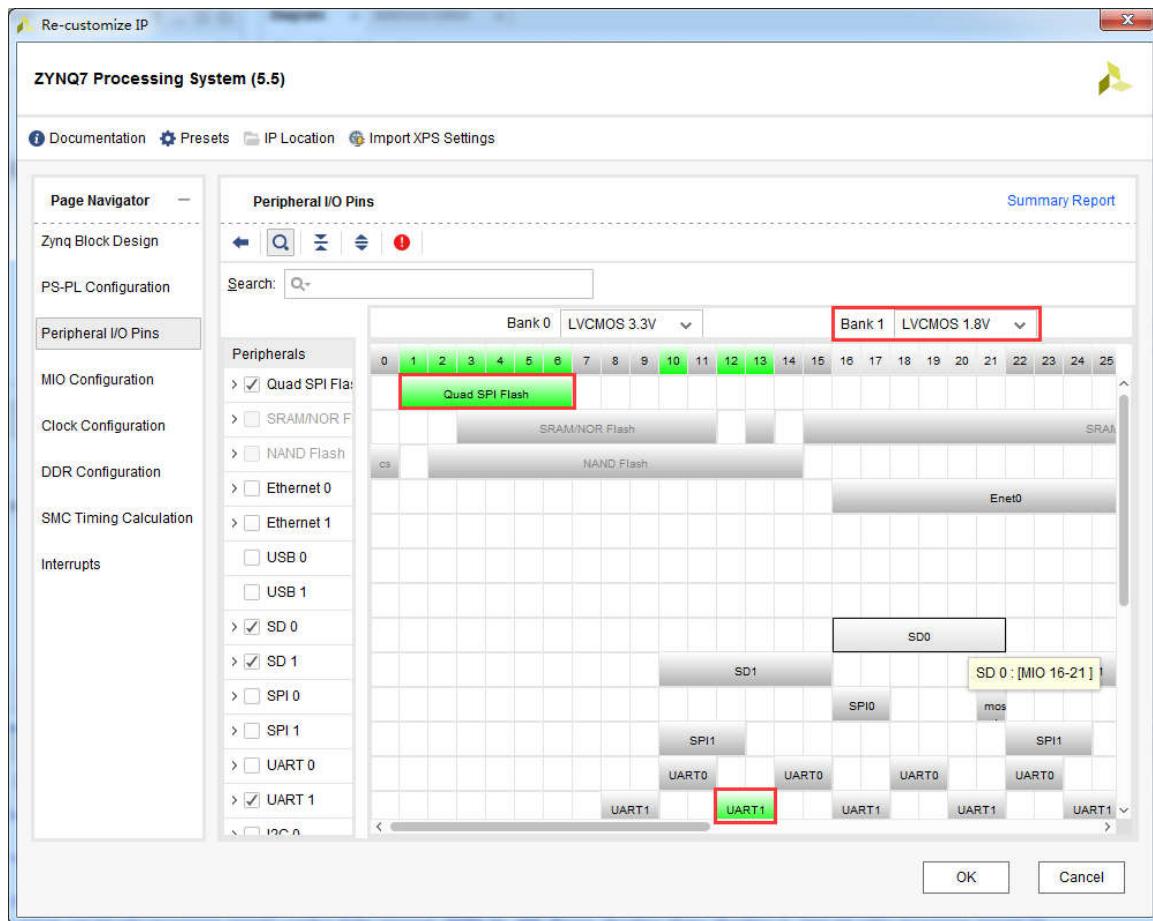
- 5) 用默认设置，点击 “OK”



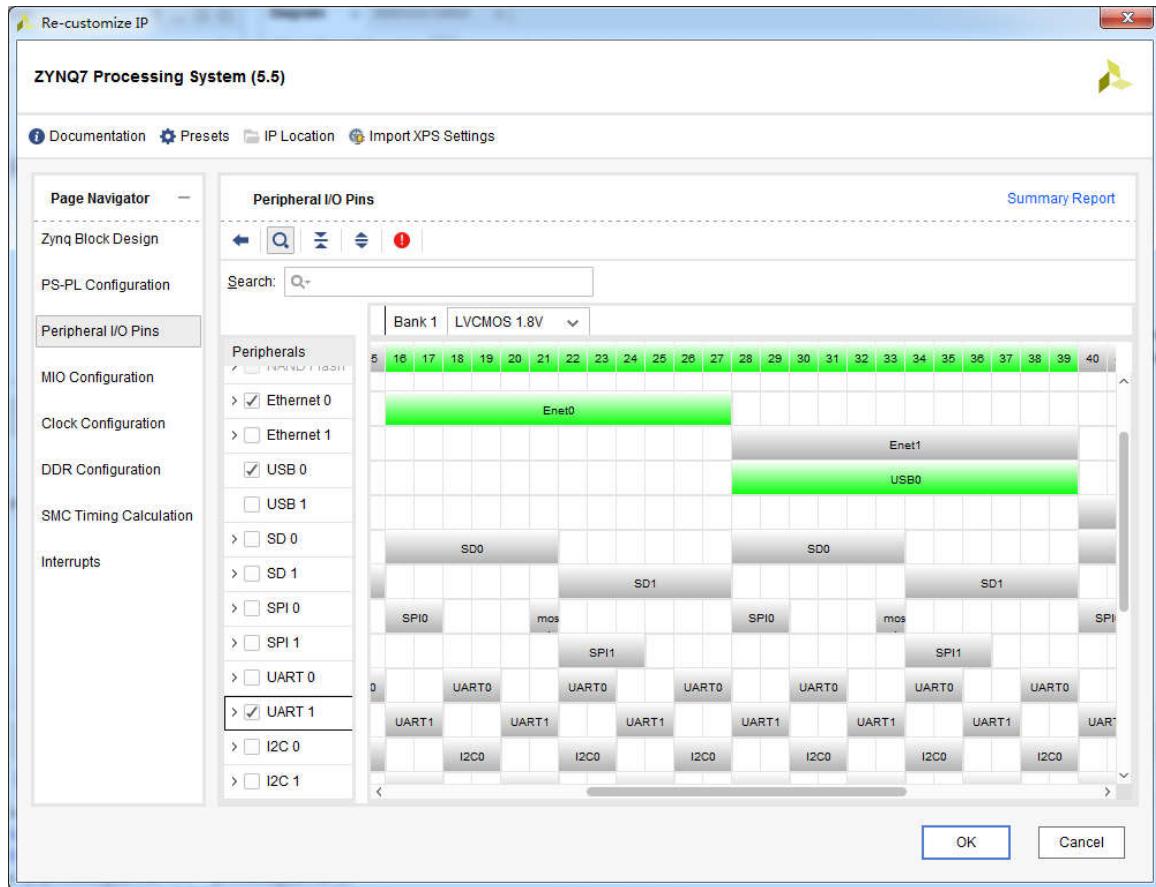
- 6) 双击 “processing_system7_0” 配置 ZYNQ PS
- 7) 在 “PS-PL Configuration” 选项卡中，使能 HP Slave AXI 接口 “S AXI HPO interface”，HP 端口可以让外部 AXI Master 接口高性能访问 ZYNQ PS ddr3 内存。



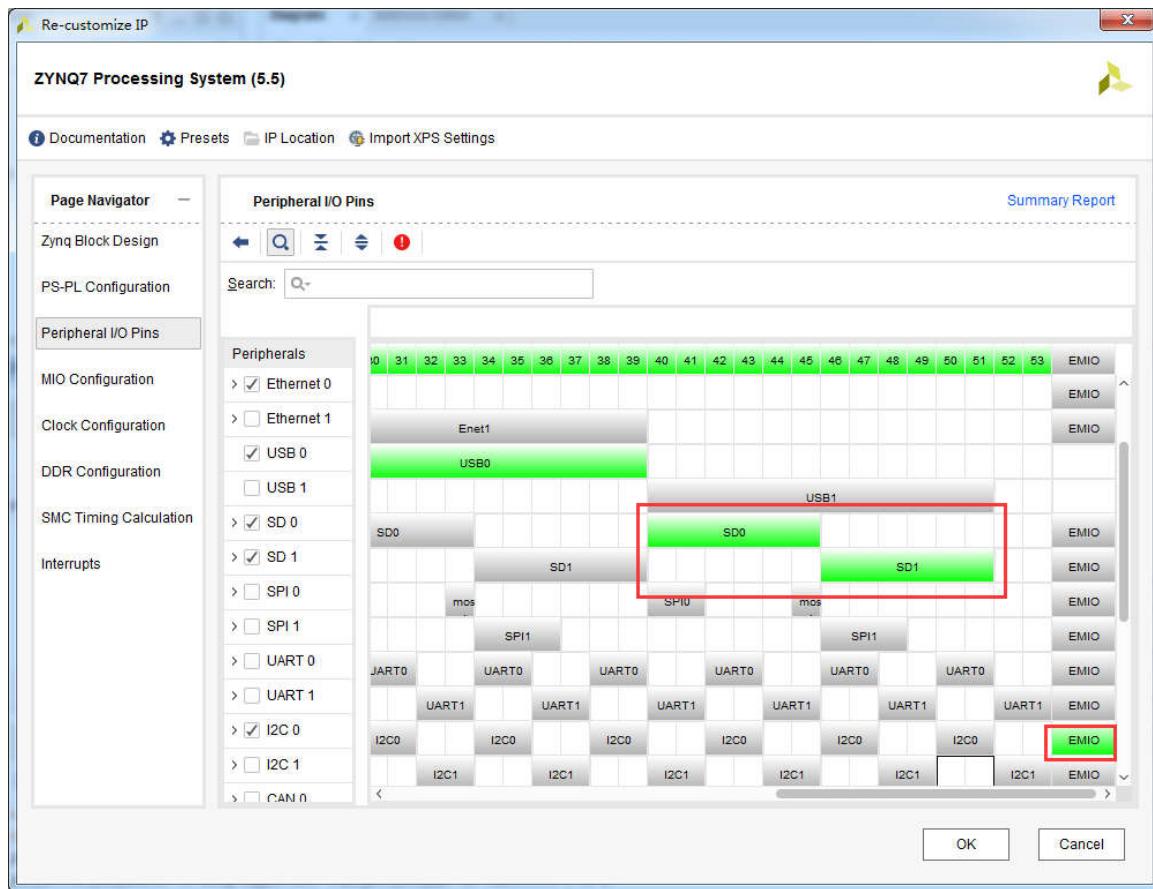
- 8) 在“Peripheral I/O Pins”选项中配置 Bank 1 电平标准为“LVCMS 1.8V”，使能 Quad SPI Flash，IO 使用 MIO1-MIO6，使能 UART1，IO 使用 MIO12-MIO13



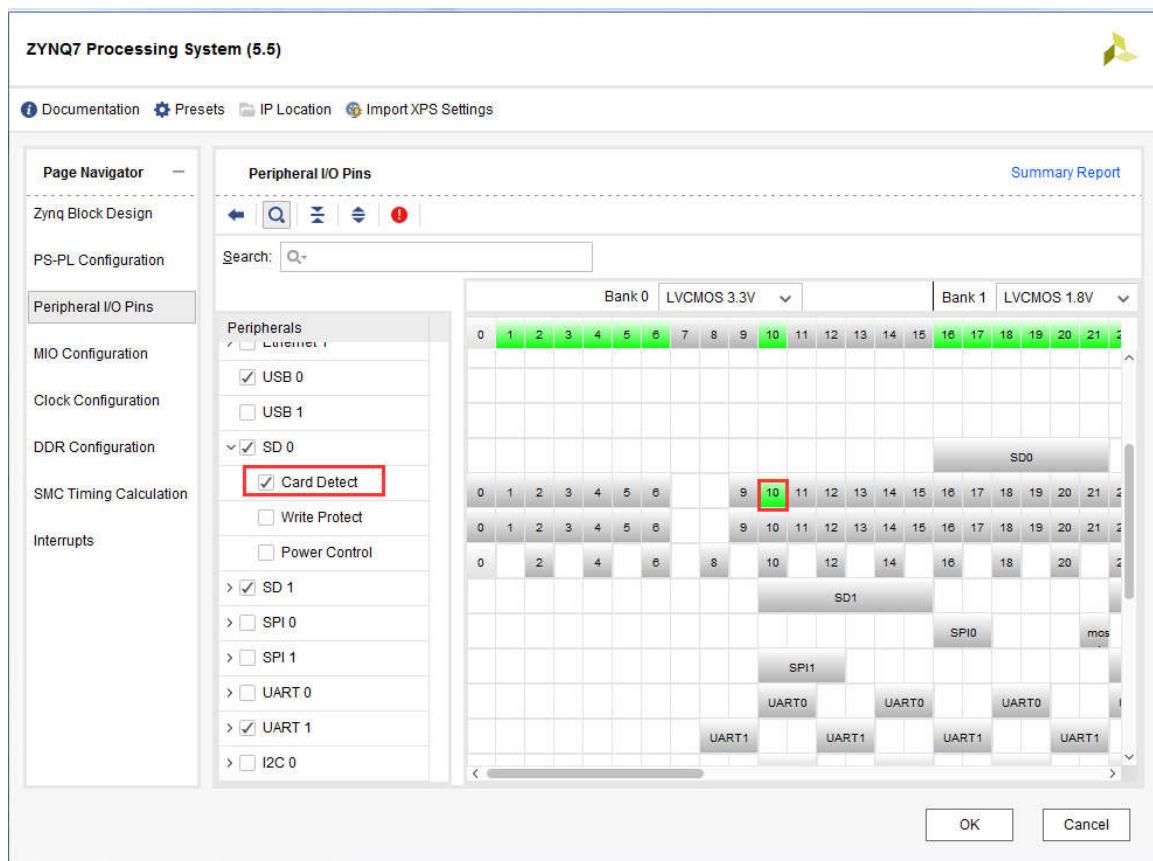
- 9) 使能 Ethernet 0，IO 使用 MIO16-MIO27，使能 USB0，IO 使用 MIO28-MIO39，



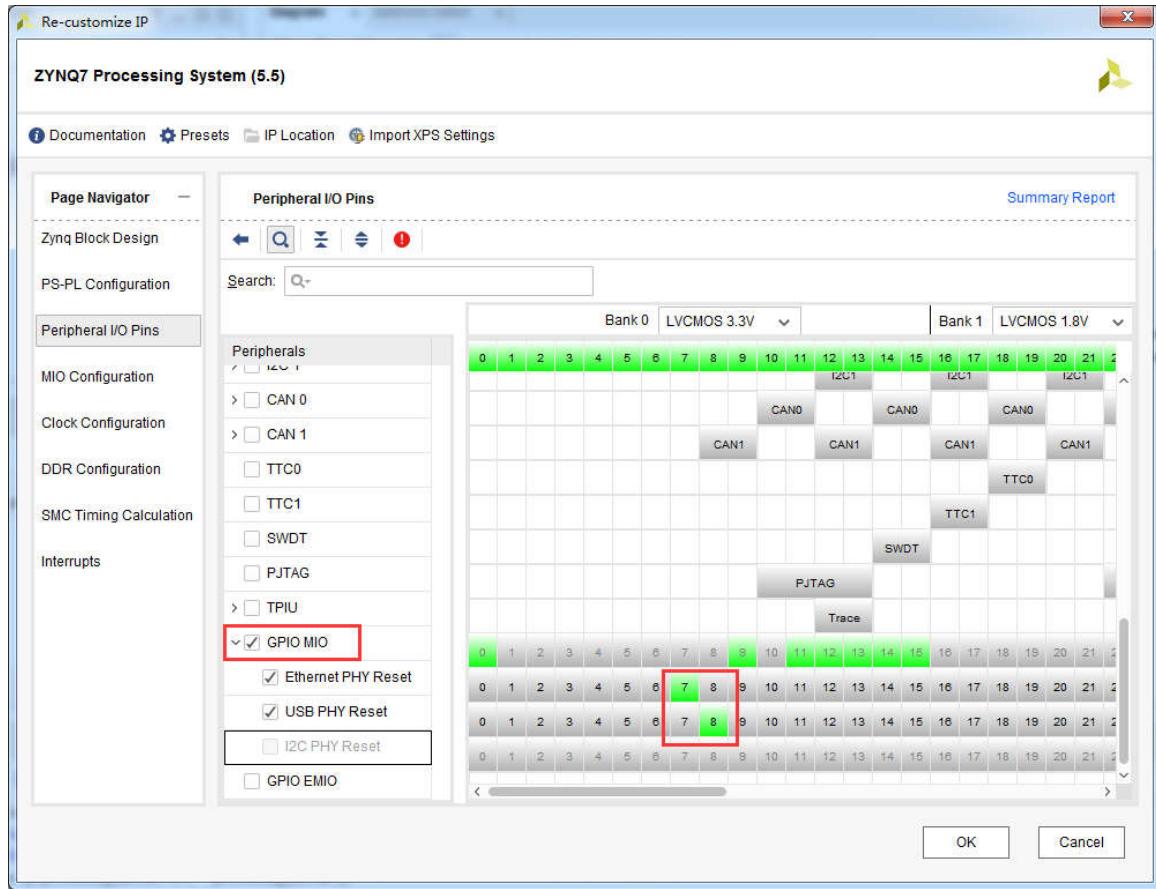
- 10) 使能 Ethernet 0 的 MDIO , IO 使用 MIO52-MIO53 , 使能 SD 0 , IO 使用 MIO40-MIO45 , 使能 SD 1 , IO 使用 MIO47-MIO51 , 使能 I2C 0 , IO 通过 PL 扩展 , 选择 EMIO



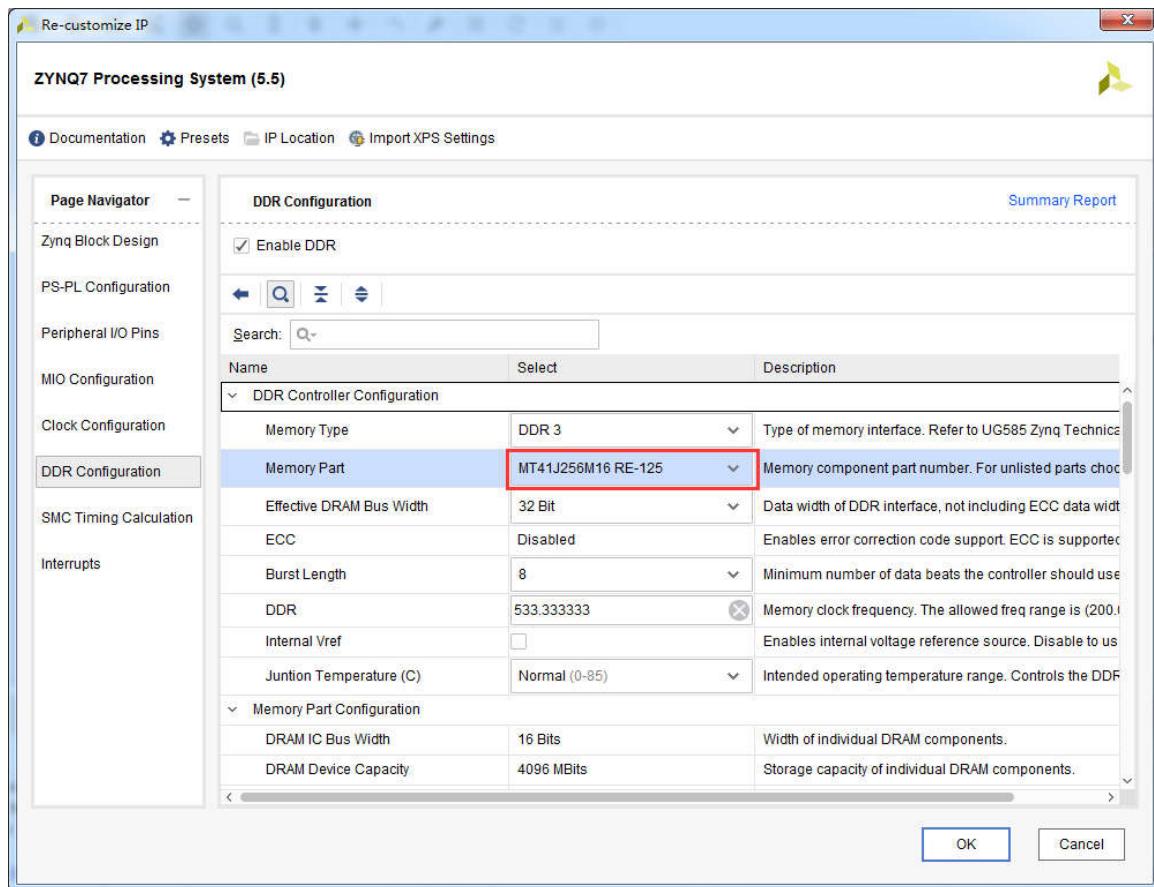
11) SD 0 的 Card Detect 选择 MIO 10



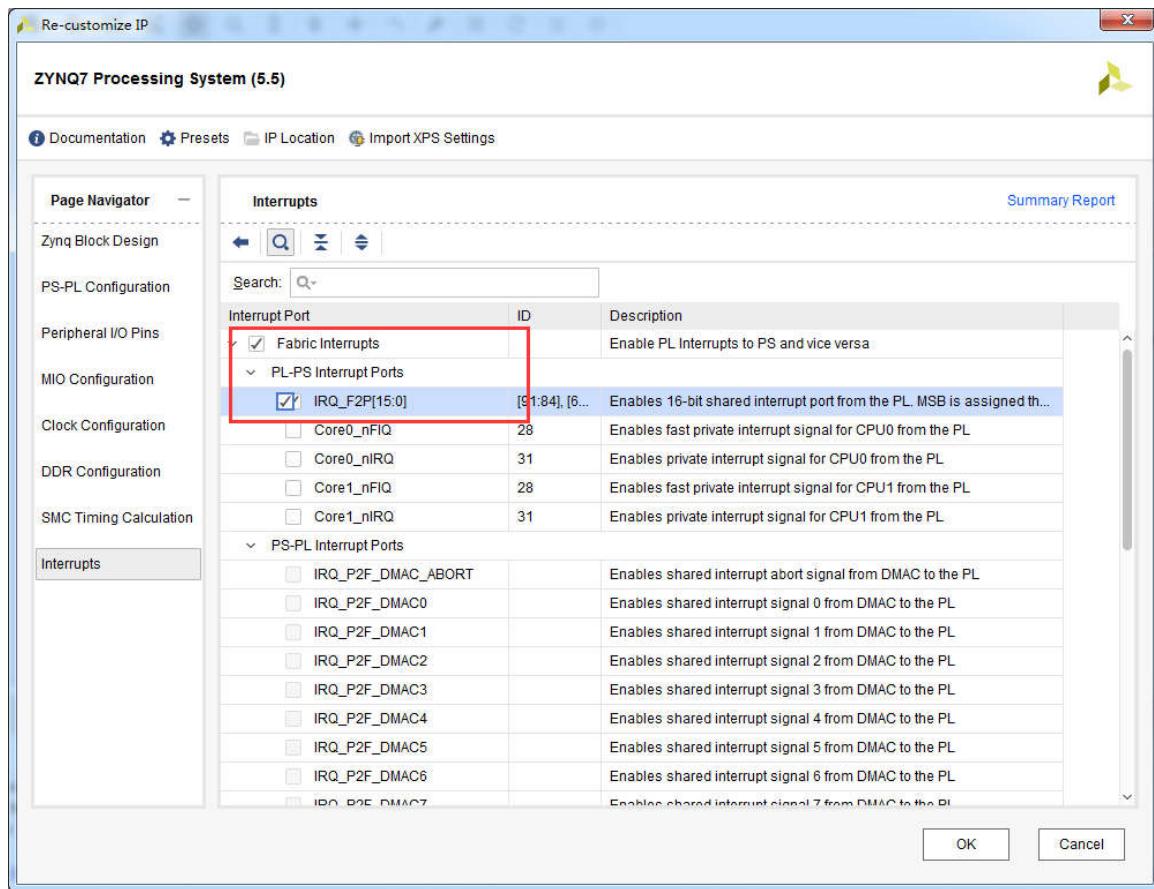
12) 使能 GPIO MIO，选择 Ethernet PHY Reset 为 MIO7 , USB PHY Reset 为 MIO8



13) 在 DDR Configuration 选择中选择 Memory Part 为 “MT41J256M16 RE-125”

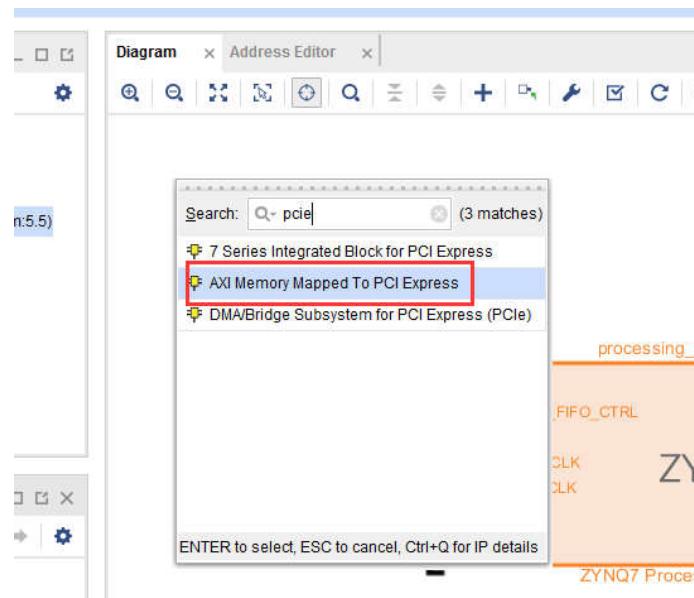


14) 在 Interrupts 选项中使能 “Fabric Interrupts” , 再使能 IRQ_F2Q[15:0] , 点击 OK 完成配置



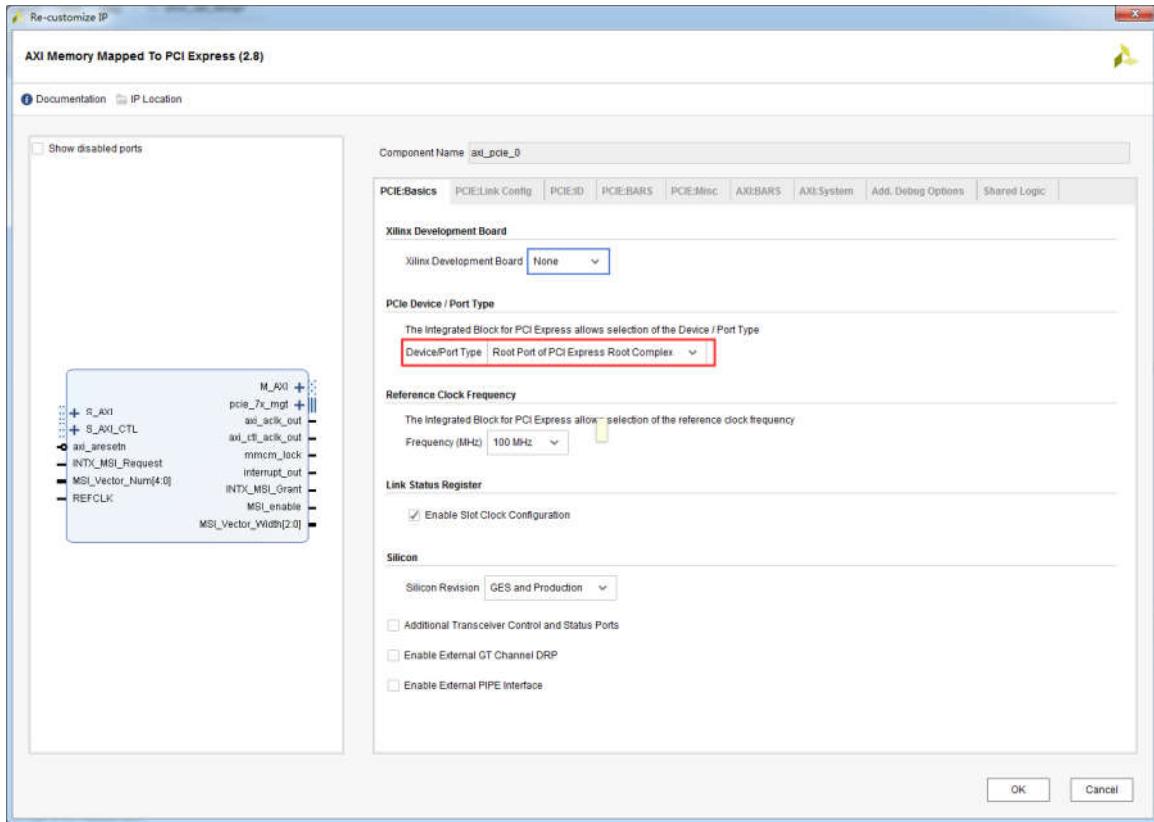
17.1.3 添加 AXI MM to PCIe bridge

1) 从 IP 仓库添加 “AXI Memory Mapped to PCI Express” block 到设计中

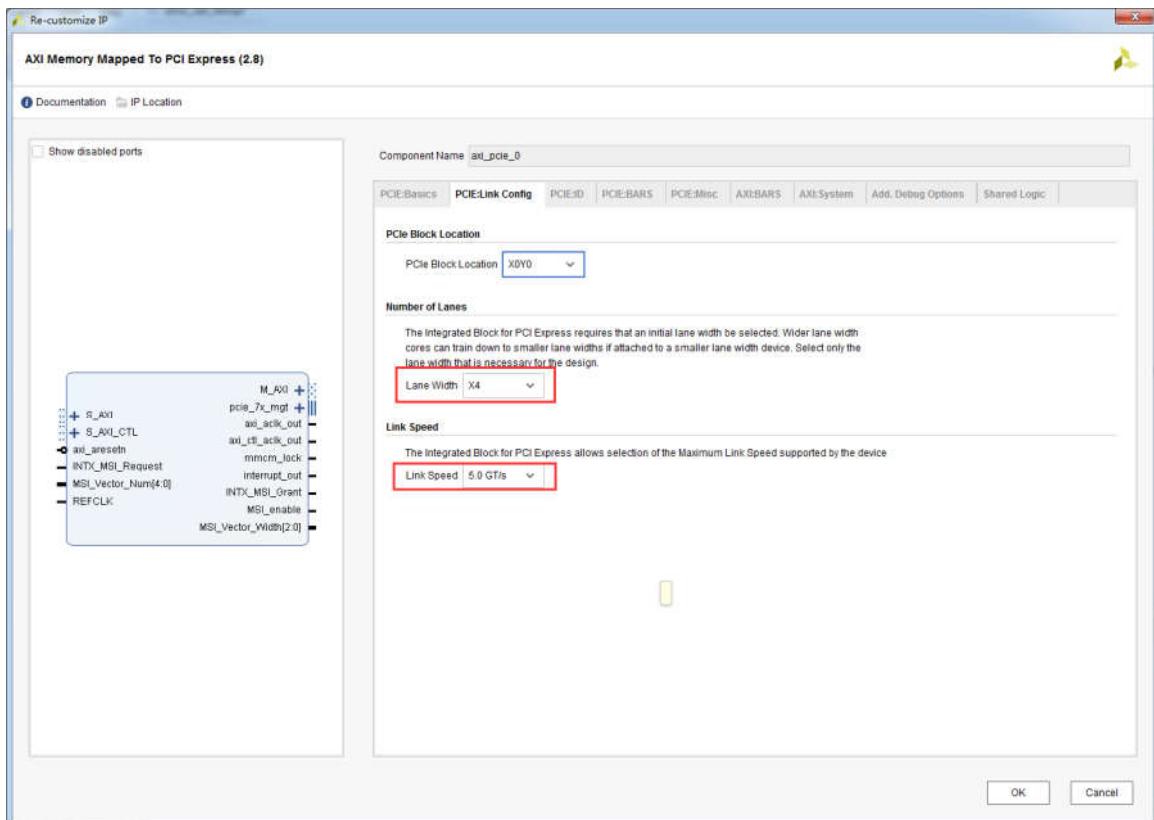


2) 双击 “axi_pcnie_0” 配置参数

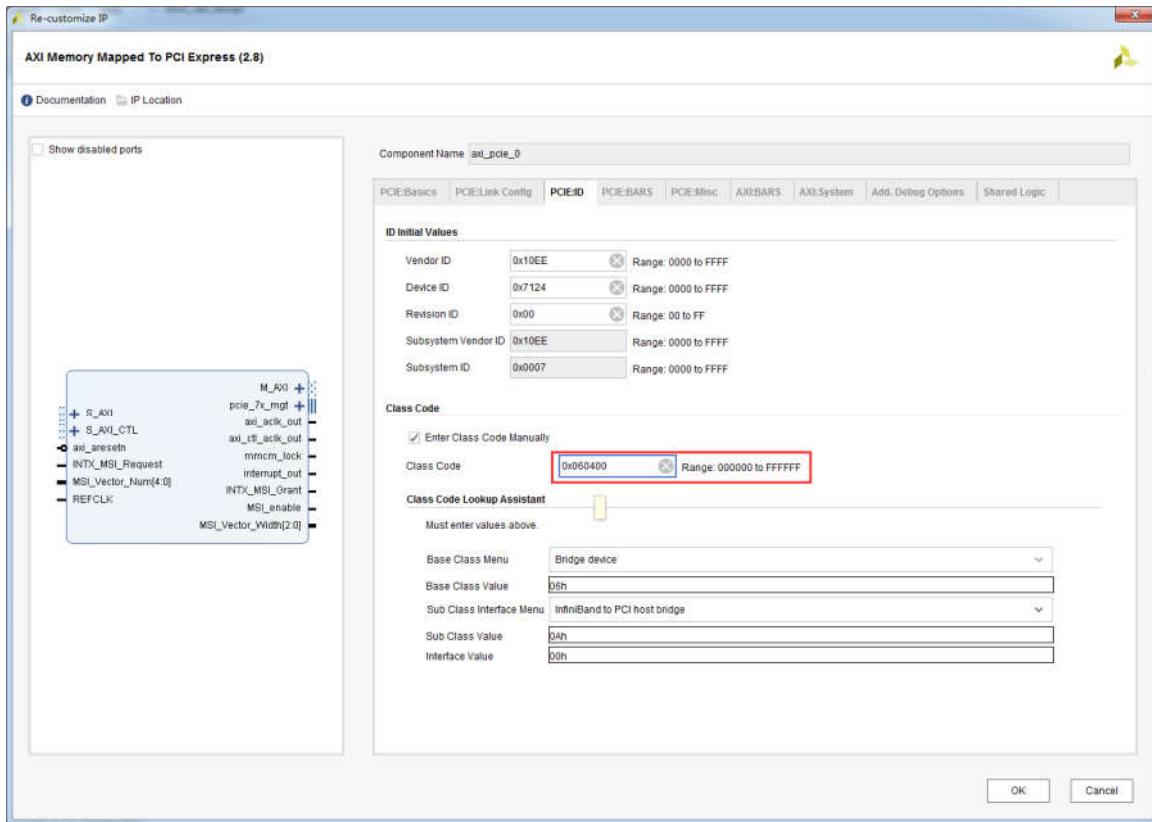
- 3) 在“PCIE:Basics”配置选项中，端口类型选择“Root Port of PCI Express Root Complex”



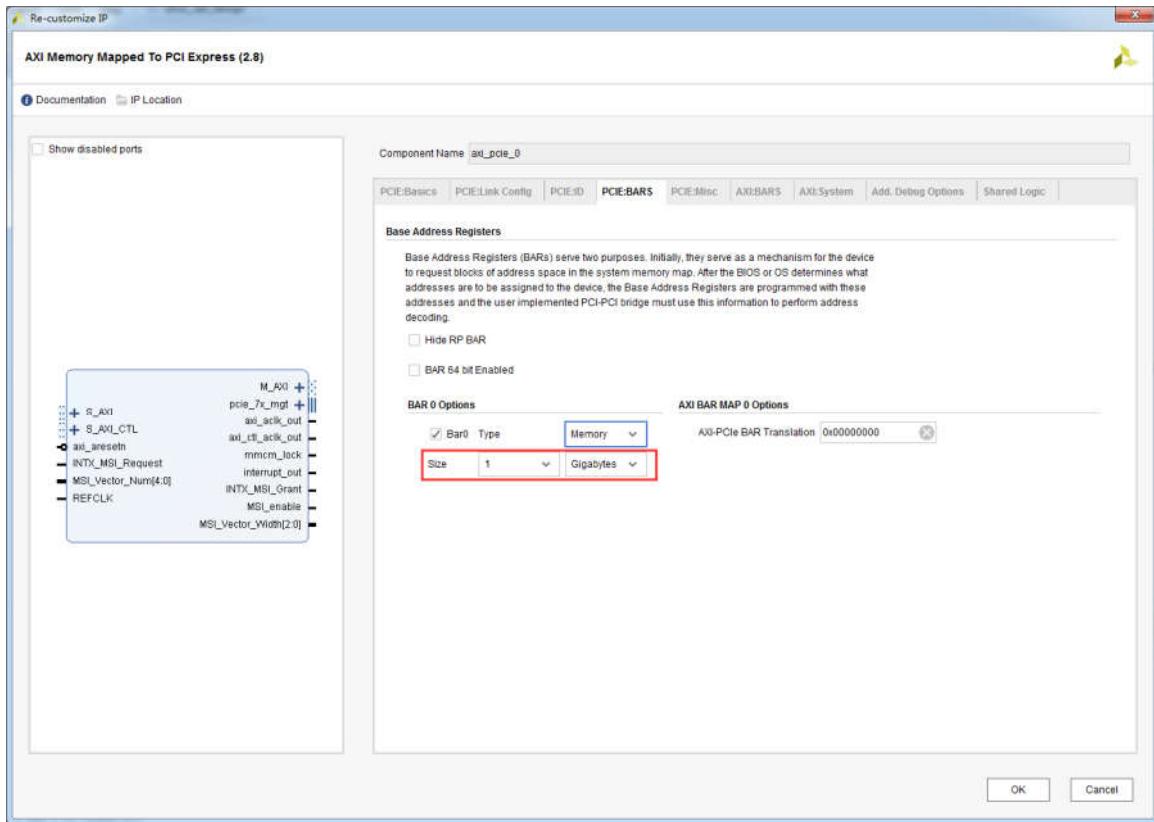
- 4) 在“PCIE:Link Config”选项中，“Lane Width”选择 X4，“Link speed”选择 5 GT/s



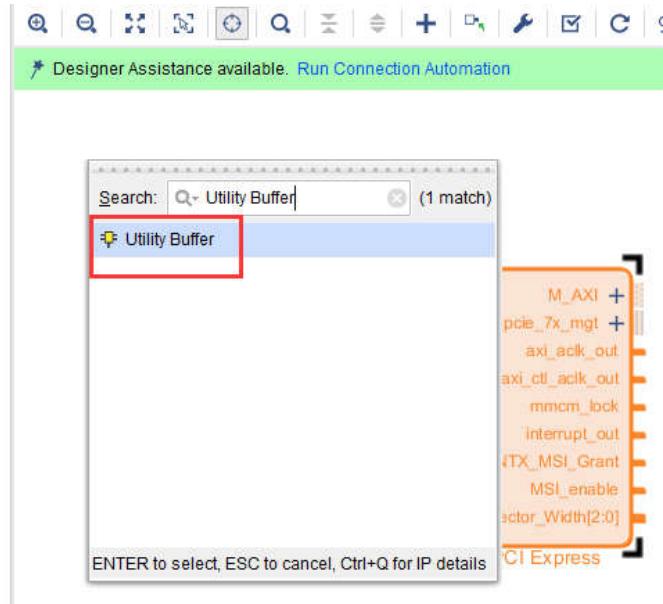
- 5) 在“PCIE:ID”选项中，“Class Code”填写 0x060400. 这是为了以后在 Linux 下能正确的使用驱动程序。



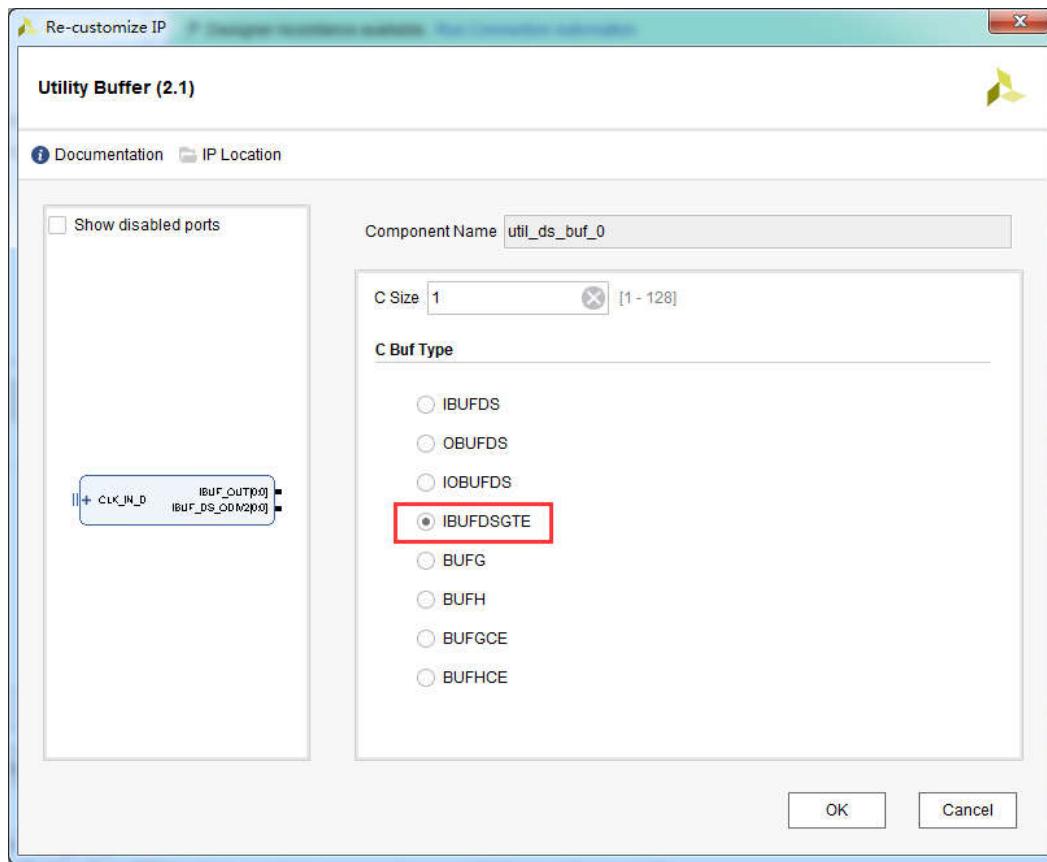
- 6) 在“PCIE:BARS”项中，设置 BAR 0 类型为“Memory”，大小为 1 Gigabytes，其他参数都保持默认，点击“OK”



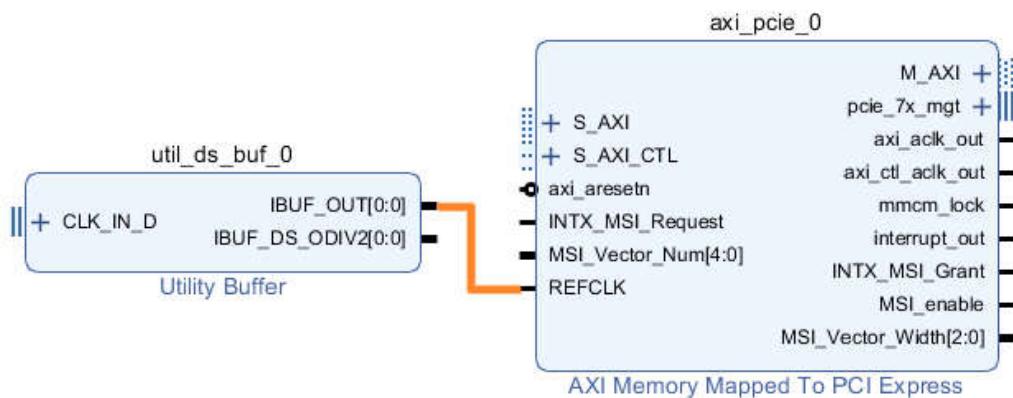
7) 添加一个“Utility Buffer”到 block 设计



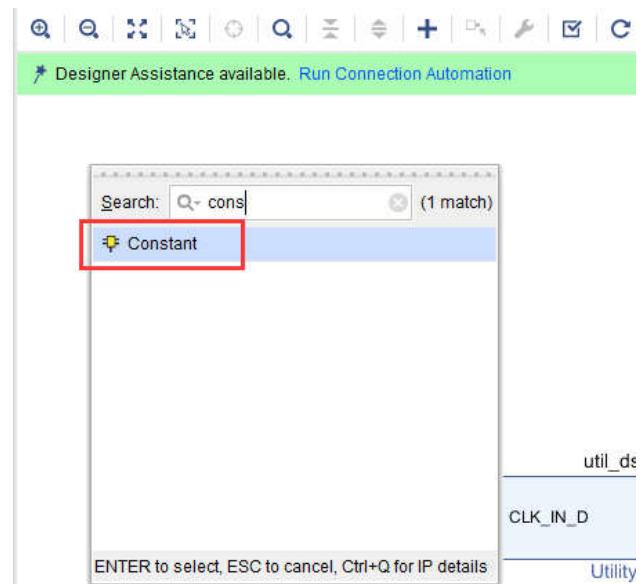
8) 双击“util_ds_buf_0”，选择“C Buf Type”为“IBUFDSGTE”，这种类似是收发器专用参考差分时钟 BUFFER，开发板上来自 S15338 输出，频率为 100Mhz



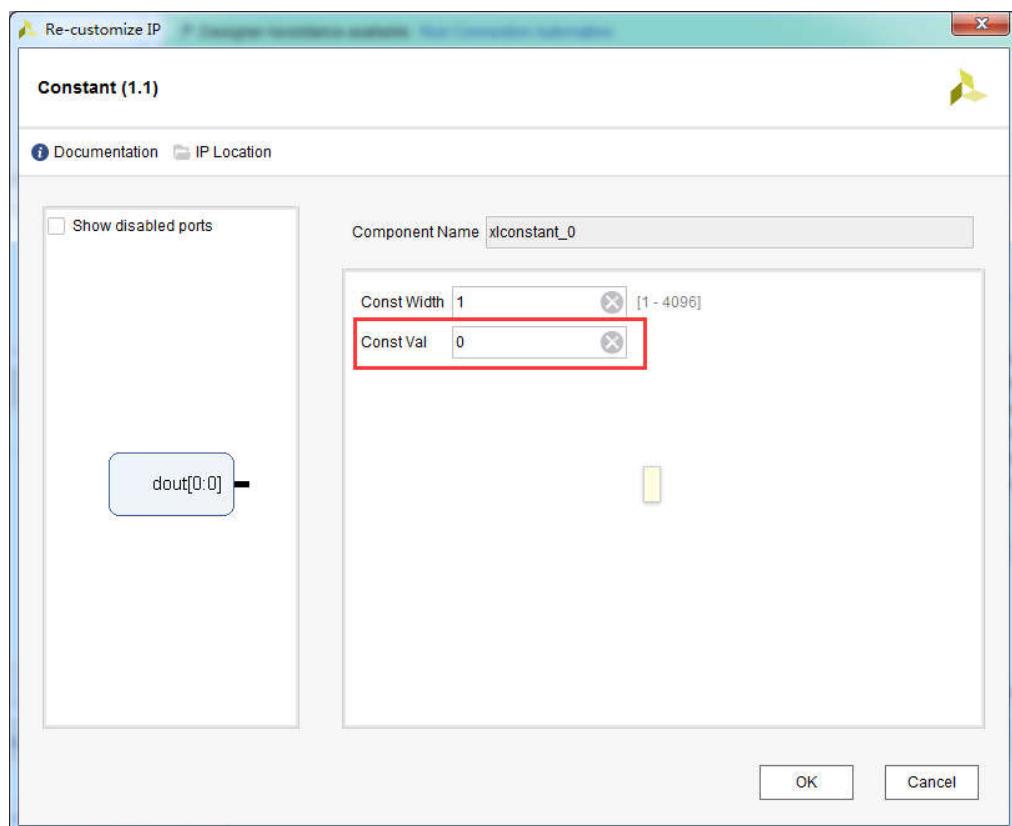
9) 连接 utility buffer 的输出 “IBUF_OUT” 到 AXI-PCIe 模块的 “REFCLK” 输入。



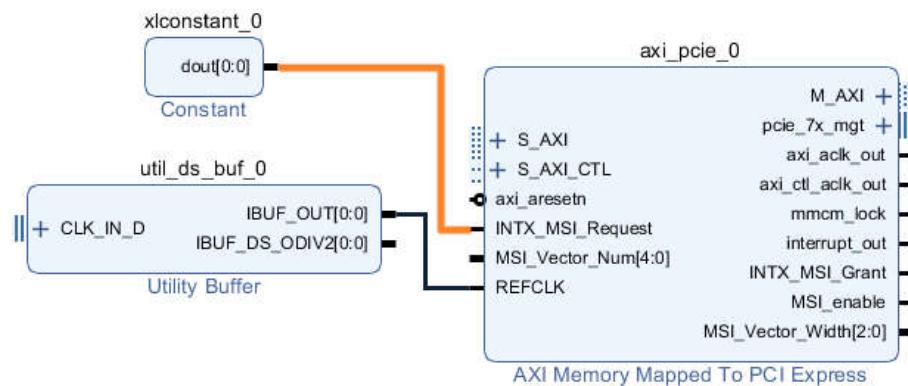
10) 添加一个 “Constant” 模块到设计中



11) 双击 “Constant” 配置参数 Const Val 修改 “0”



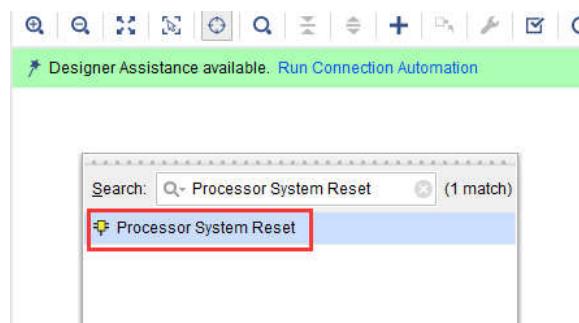
12) 把 xlconstant_0 输出连接到 axi_pcie_0 的输入 INTX_MSI_Request



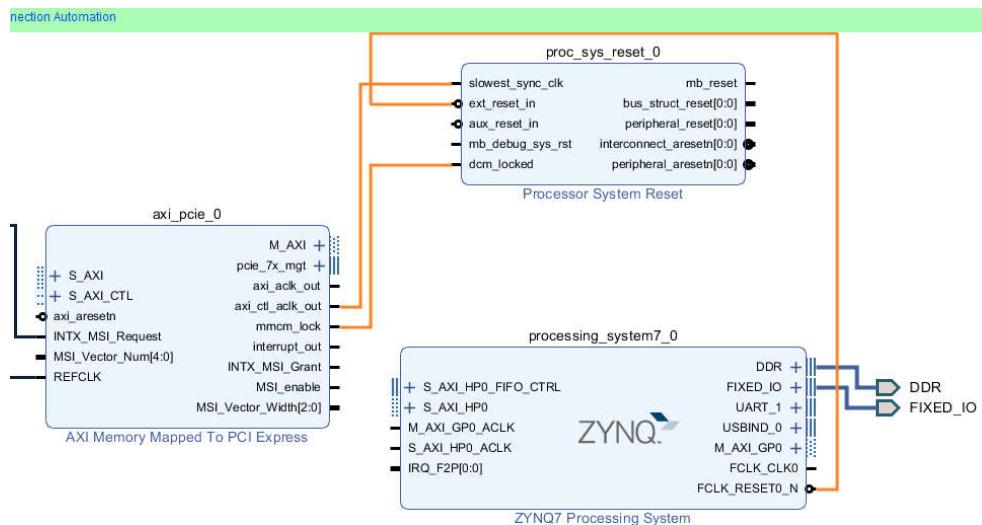
17.1.4 添加复位模块

设计中有 2 个时钟，都是来自 AXI-PCIe bridge 模块的输出，分别为 “`axi_aclk_out`”、“`axi_ctl_aclk_out`”，一个用于 PCIe 数据传输，一个用于模块的寄存器控制，所以要添加 2 个复位模块。

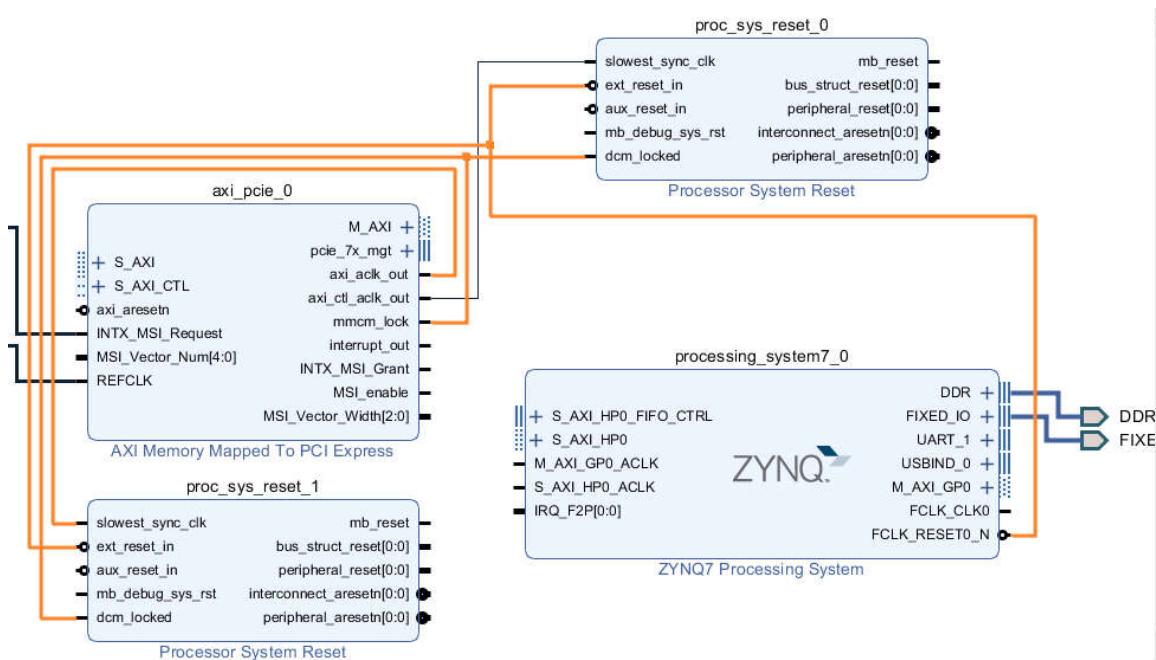
- 1) 添加 “Processor System Reset” 到设计中，默认名称为 “`proc_sys_reset_0`”



- 2) 把 “`axi_pcie_0`” 模块的输出 “`axi_ctl_aclk_out`” 连接到 “`proc_sys_reset_0`” 模块的输入 “`slowest_sync_clk`”，把 “`axi_pcie_0`” 模块的输出 “`mmcm_lock`” 连接到 “`proc_sys_reset_0`” 模块的输入 “`dcm_locked`” ,把模块 “`processing_system7_0`” 的输出 “`FCLK_RESET0_N`” 连接到模块 “`proc_sys_reset_0`” 的输入 “`ext_reset_in`”。

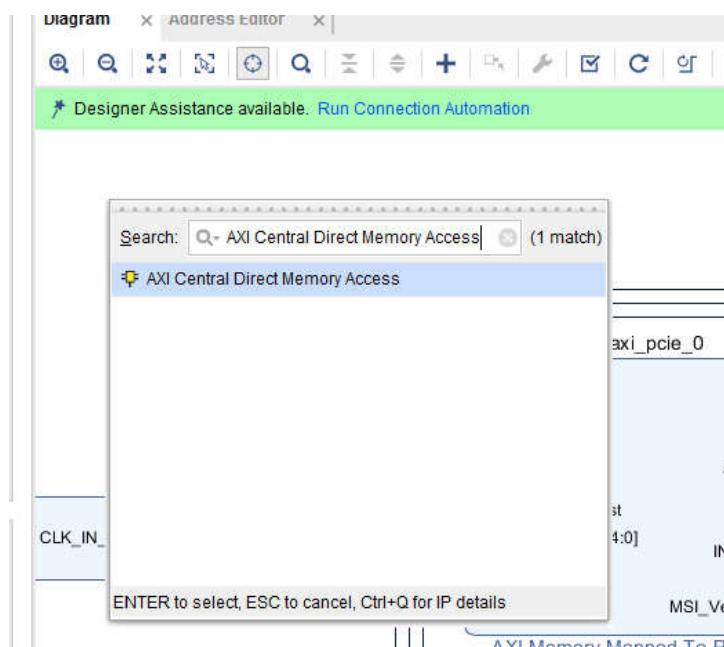


- 3) 添加“Processor System Reset”到设计中，默认名称为“proc_sys_reset_1”
- 4) 把“axi_pcie_0”模块的输出“axi_aclk_out”连接到“proc_sys_reset_1”模块的输入“slowest_sync_clk”，把“axi_pcie_0”模块的输出“mmcm_lock”连接到“proc_sys_reset_1”模块的输入“ext_reset_in”,把模块“processing_system7_0”的输出“FCLK_RESET0_N”连接到模块“proc_sys_reset_1”的输入“aux_reset_in”。

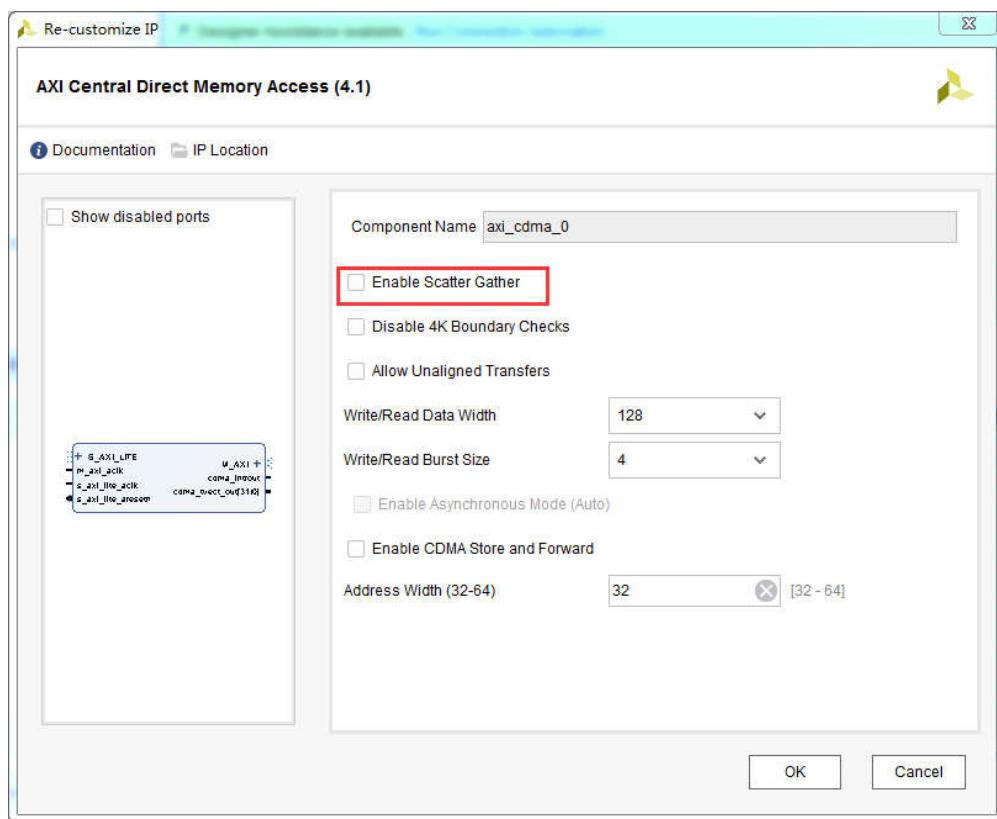


17.1.5 添加 DMA 模块

- 1) 添加一个“AXI Central Direct Memory Access”到设计中

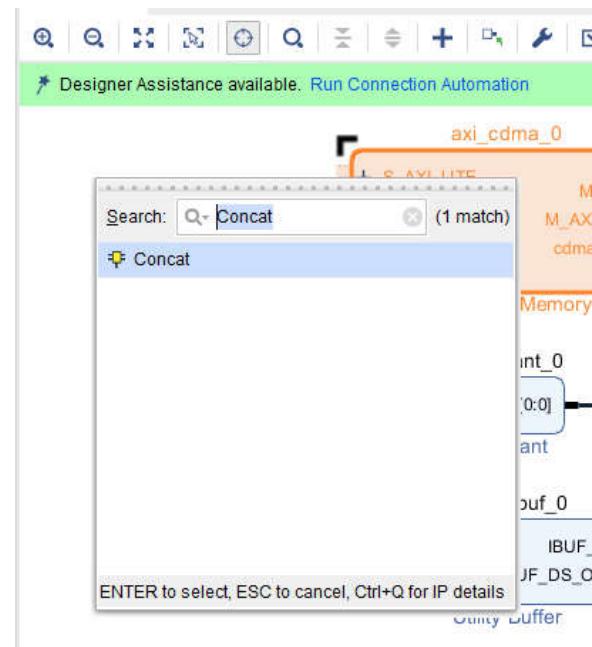


- 2) 双击“axi_cdma_0” 打开配置窗口，“Write/Read Data Width” 选择 128，不要勾选 “Enable Scatter Gather”

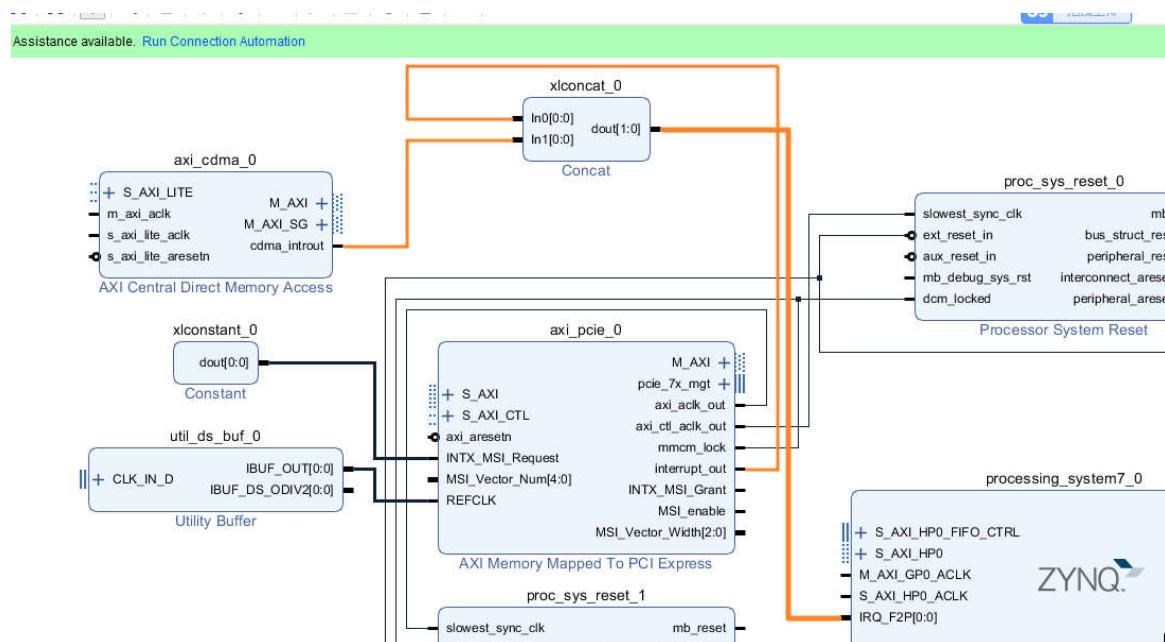


17.1.6 连接中断

- 1) 添加一个“Concat”到设计中

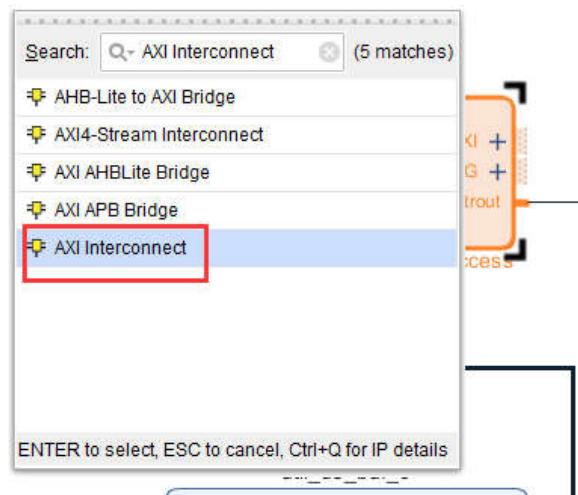


- 2) 连接 axi_pcie_0 模块的中断到 xlconcat_0 模块的 In0 端口，连接 axi_cdma_0 模块的中断到 xlconcat_0 模块的 In1 端口，连接 xlconcat_0 模块的输出 dout 到 ZYNQ PS 处理器 IRQ_F2P 端口。

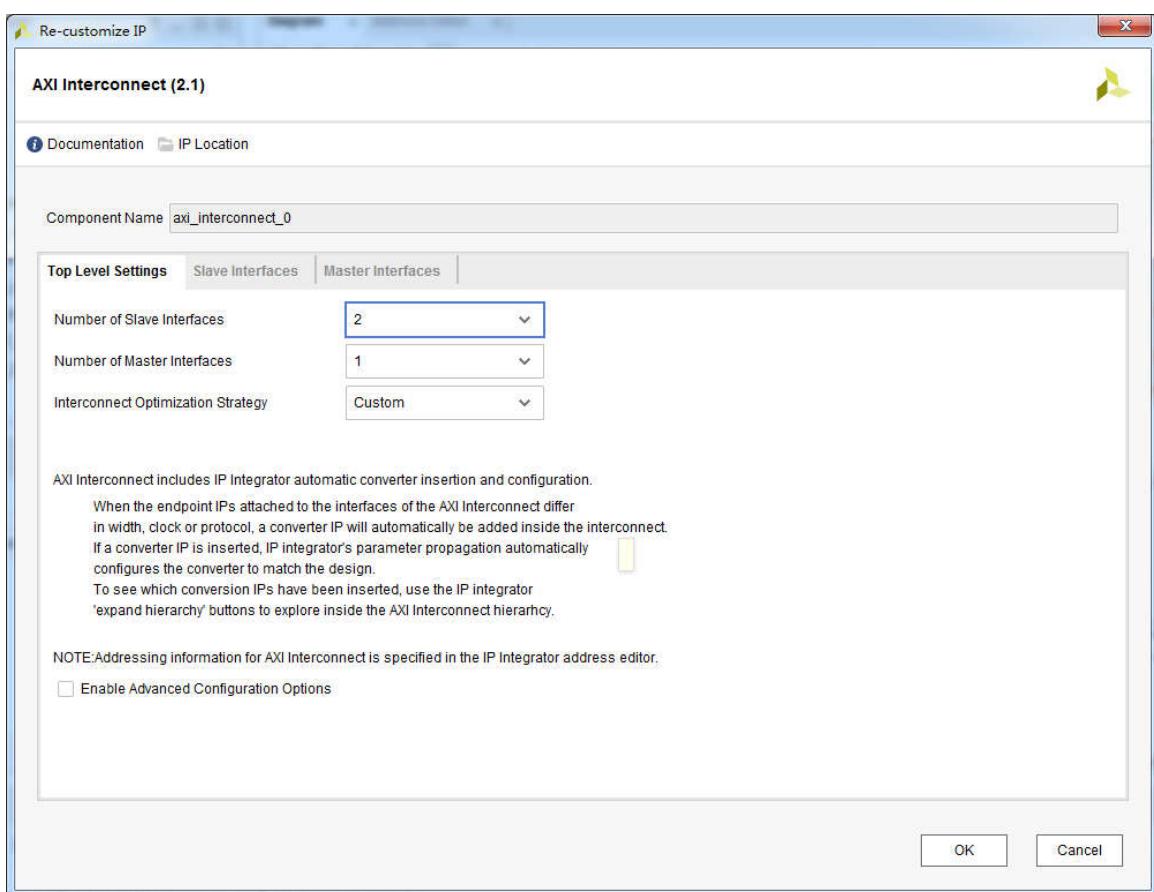


17.1.7 添加 AXI 互联模块

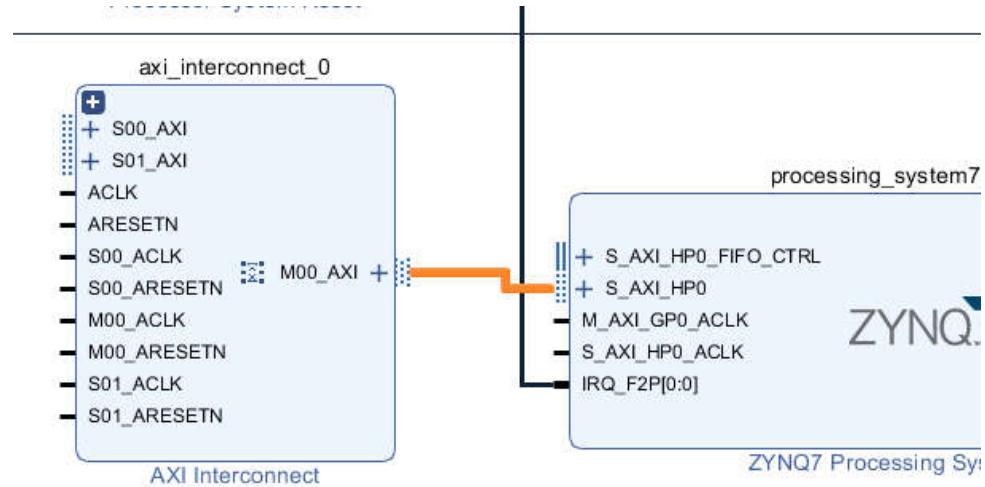
- 1) 添加一个“AXI Interconnect”模块到设计中，默认名称为“axi_interconnect_0”



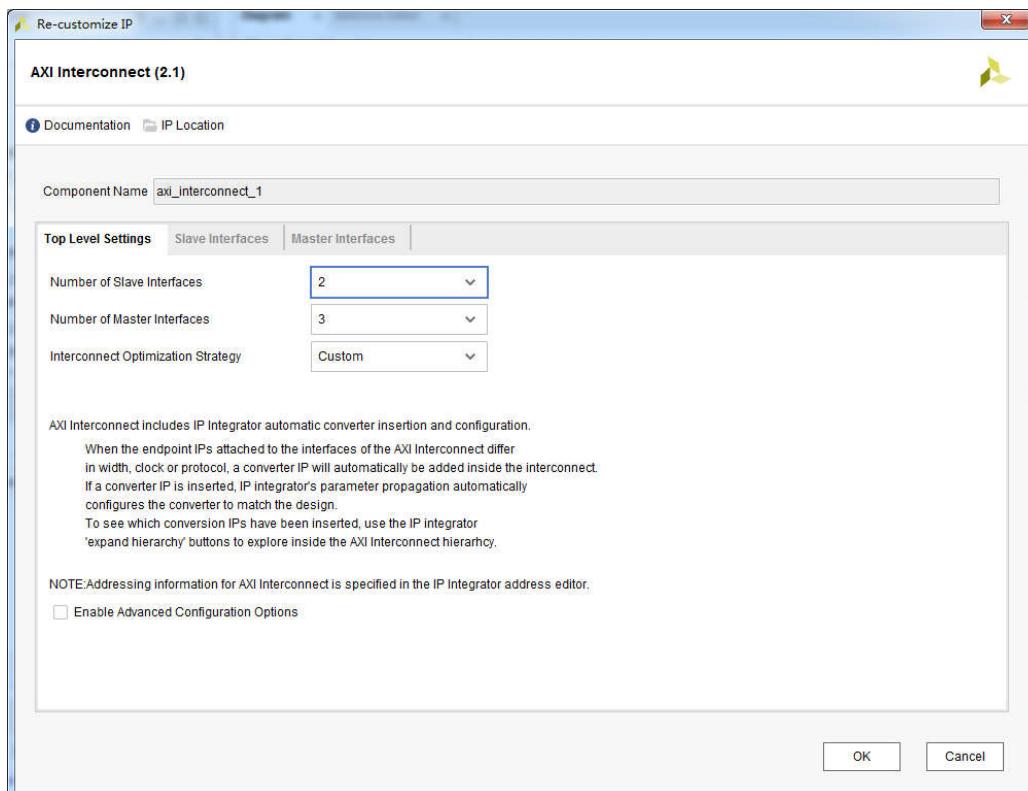
- 2) 双击 “axi_interconnect_0” , 配置参数修改为 2 个 Slave 端口 , 1 个 Master 端口



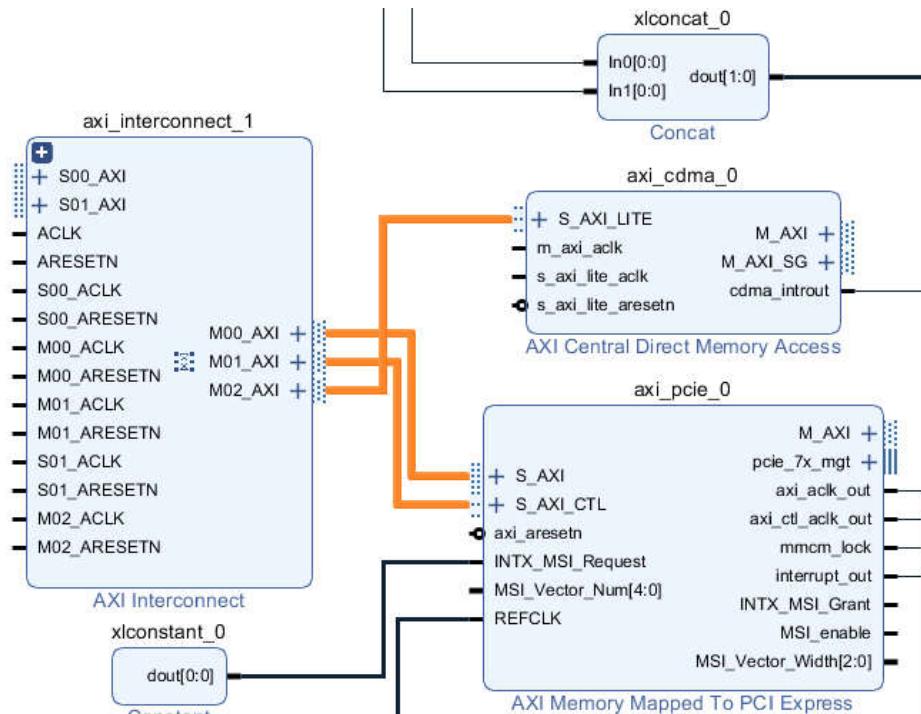
- 3) 连接 “axi_interconnect_0” 模块的 M00_AXI 端口到 ZYNQ PS 处理器的 S_AXI_HPO 端口



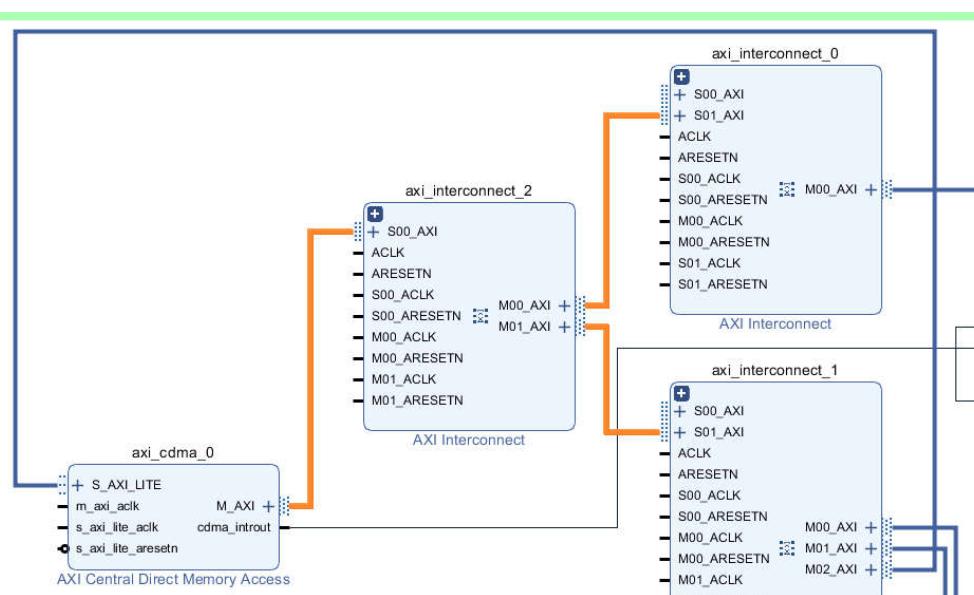
- 4) 添加一个“AXI Interconnect”模块到设计中，默认名称为“axi_interconnect_1”
- 5) 双击“axi_interconnect_1”，配置参数修改为 2 个 Slave 端口，3 个 Master 端口



- 6) 连接“axi_interconnect_1”模块的“M00_AXI”端口到 AXI-PCIe 模块的“S_AXI”端口，连接“axi_interconnect_1”模块的“M01_AXI”端口到 AXI-PCIe 模块的“S_AXI_CTL”端口，连接“axi_interconnect_1”模块的“M02_AXI”端口到 CDMA 模块的“S_AXI_LITE”端口

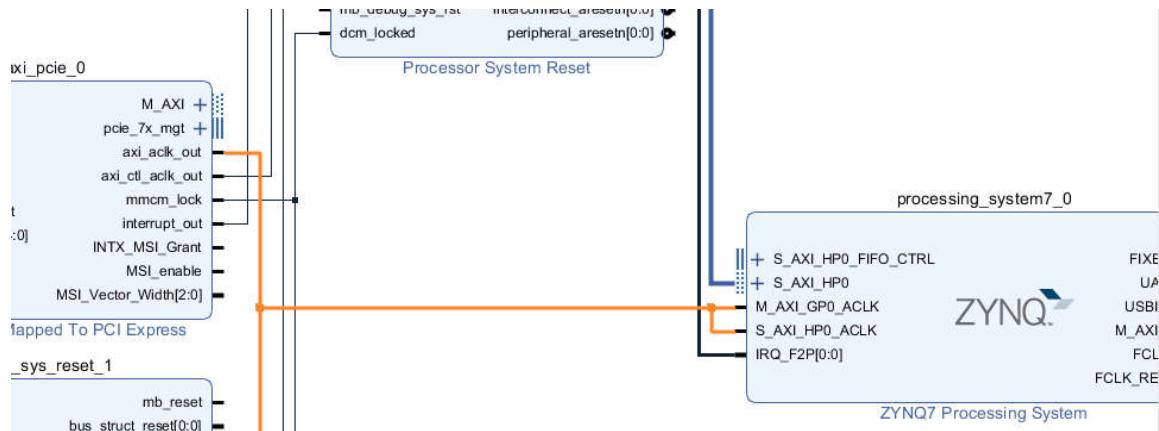


- 7) 添加一个“AXI Interconnect”模块到设计中，默认名称为“axi_interconnect_2”
- 8) 连接“axi_interconnect_2”模块的“M00_AXI”端口到“axi_interconnect_0”模块的“S01_AXI”端口，连接“axi_interconnect_2”模块的“M01_AXI”端口到“axi_interconnect_1”模块的“S01_AXI”端口，连接“axi_interconnect_2”模块的“S00_AXI”端口到“axi_cdma_0”模块的“M_AXI”端口

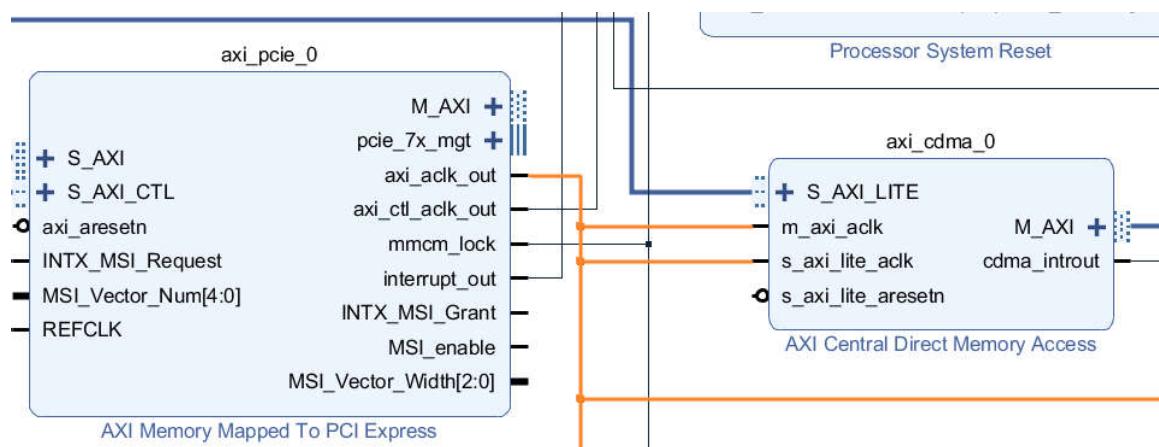


17.1.8 连接时钟

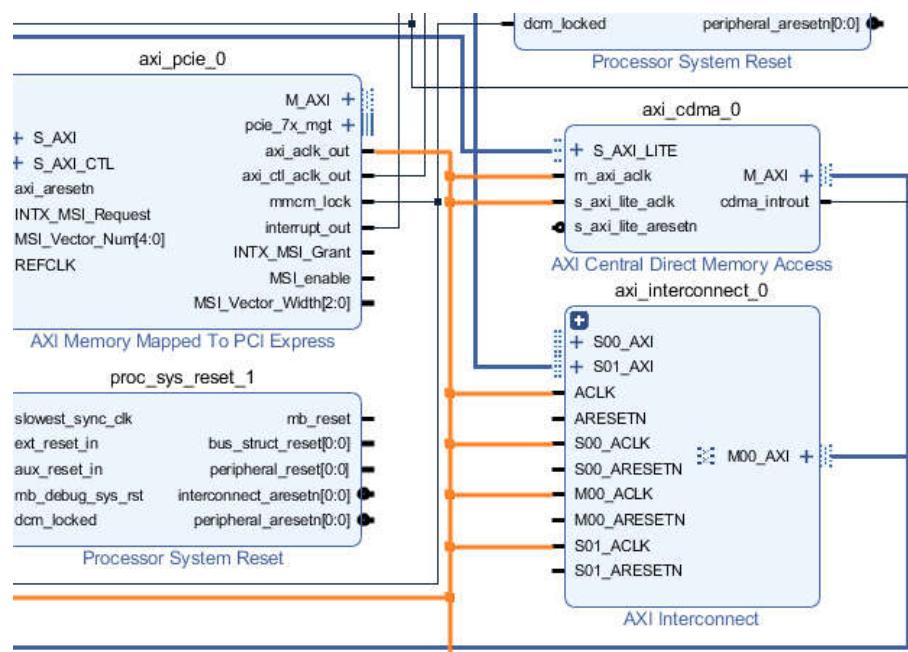
- 1) 连接 “axi_aclk_out”时钟到 Zynq PS 的输入 “M_AXI_GP0_ACLK” 和 “S_AXI_HP0_ACLK”



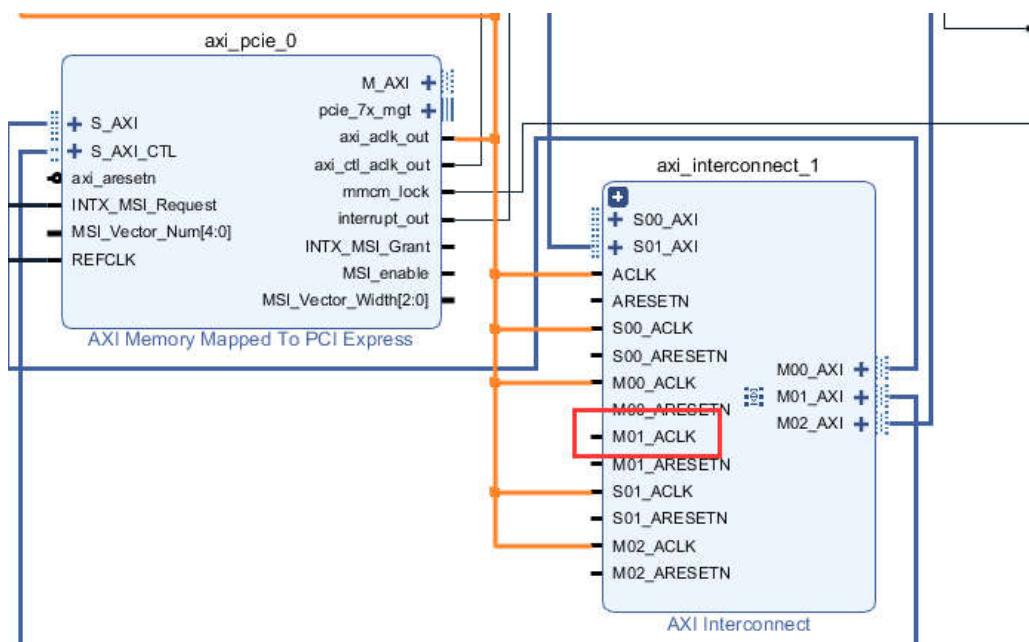
- 2) 连接 “axi_aclk_out”时钟到 “axi_cdma_0”的输入 “m_axi_aclk” 和 “s_axi_lite_aclk”



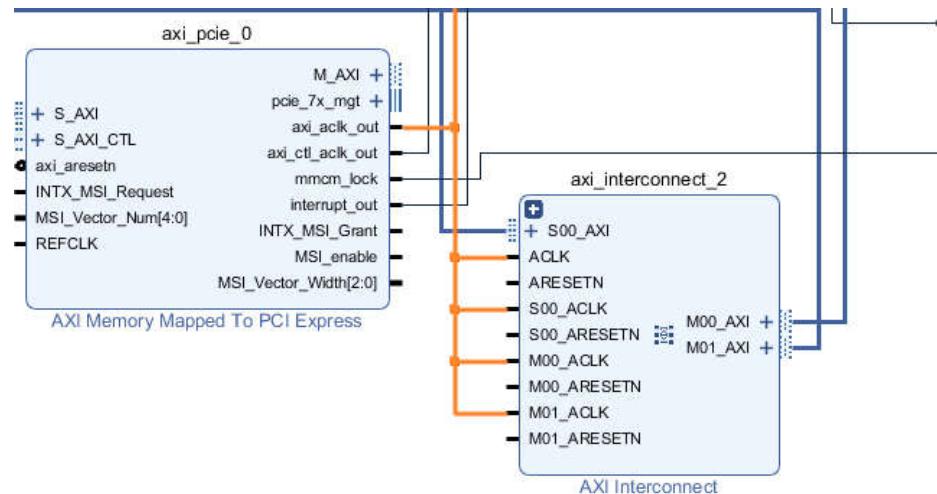
- 3) 连接 “axi_interconnect_0” 所有时钟到 “axi_aclk_out”



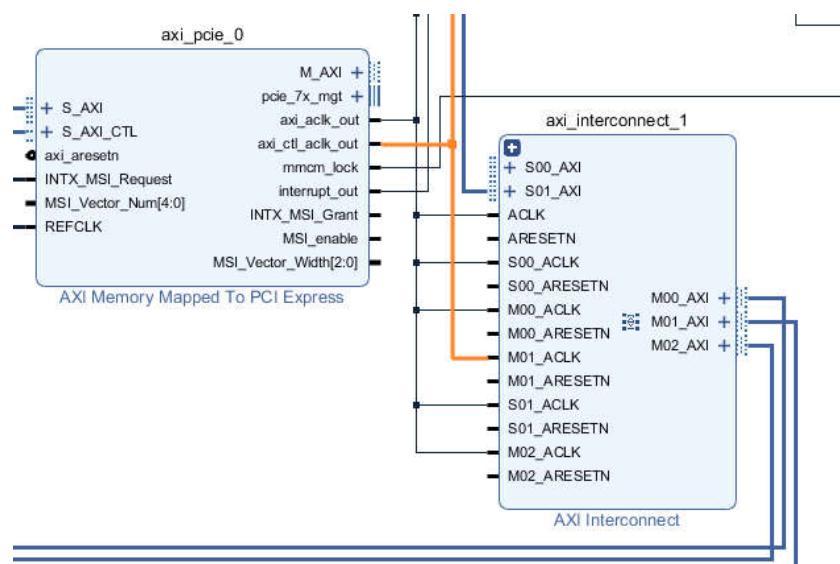
4) 连接 “axi_interconnect_1” 除了 “M01_ACLK” 的所有时钟到 “axi_aclk_out”



5) 连接 “axi_interconnect_2” 所有时钟到 “axi_aclk_out”

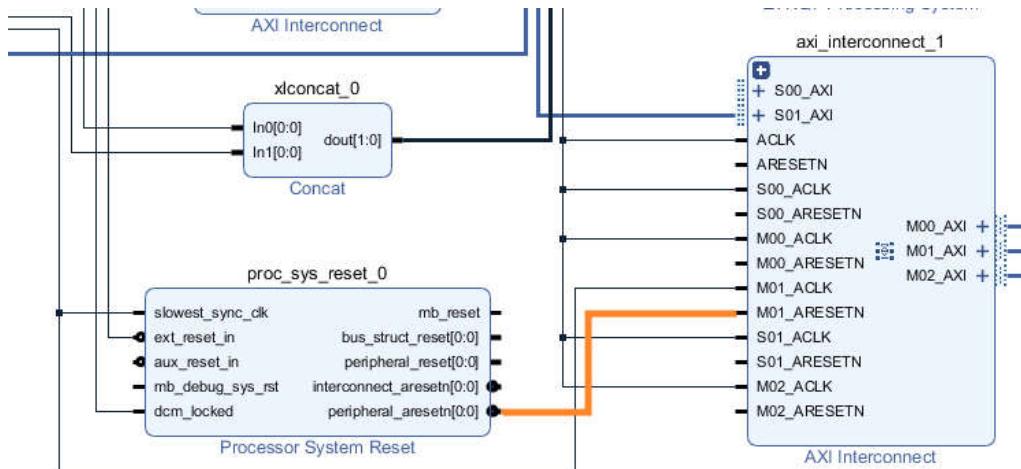


6) 连接 “axi_interconnect_1” 的时钟 “M01_ACLK” 到 “axi_ctl_aclk_out”

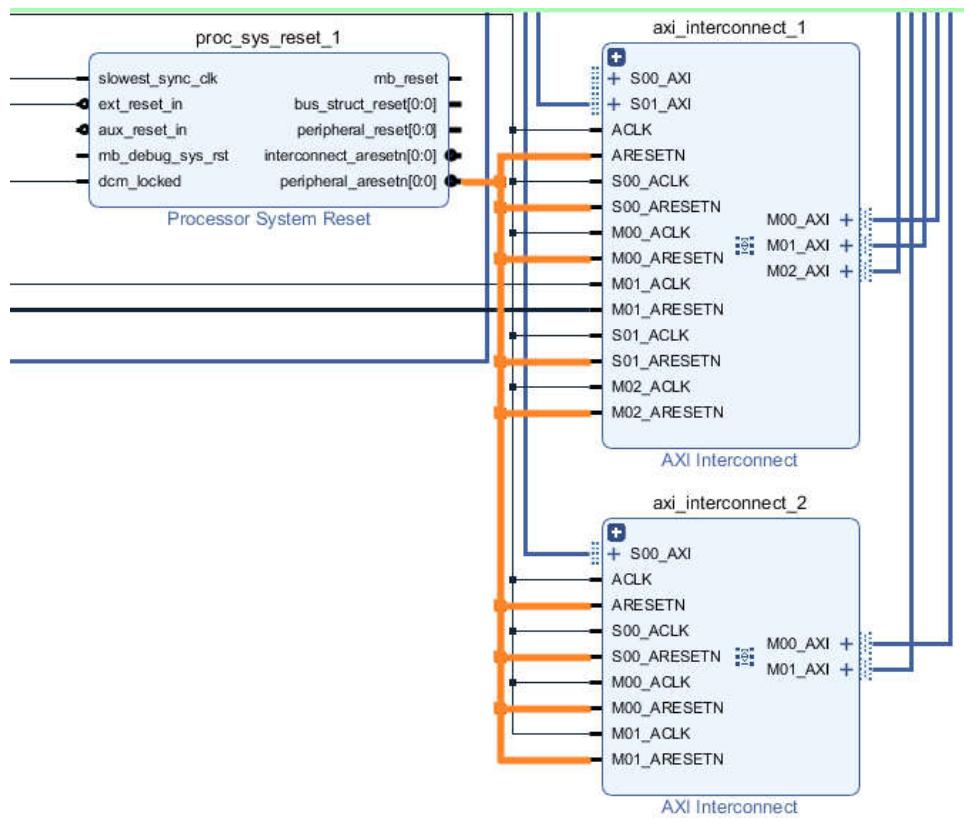


17.1.9 连接复位信号

1) 连接复位模块 “proc_sys_reset_0” 的输出 “peripheral_aresetn” 到 AXI 互联模块 “axi_interconnect_1”的复位输入 “M01_ARESETN”。

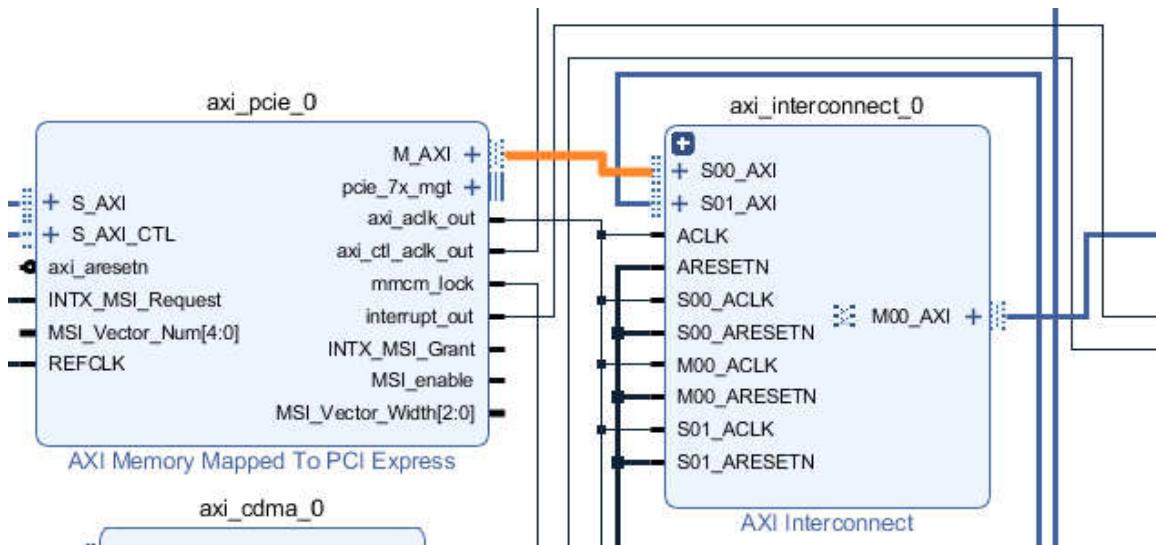


- 2) 连接复位模块 “proc_sys_reset_1” 的输出 “peripheral_aresetn” 到 AXI 互联模块 “axi_interconnect_1”的其他复位输入 ,连接到 “axi_interconnect_0”、“axi_interconnect_2” 的全部复位输入。

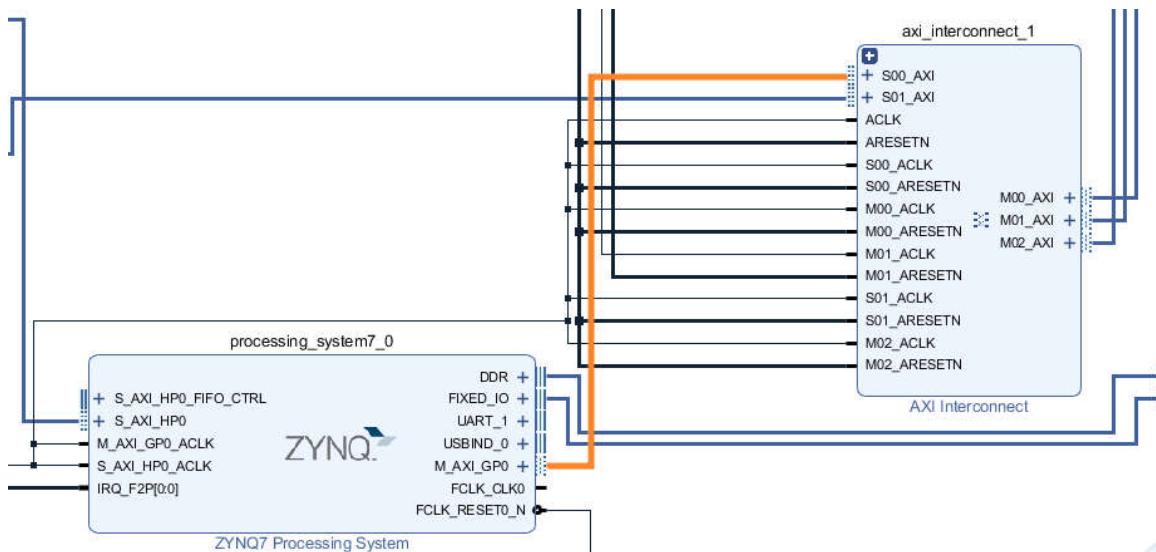


17.1.10 其他连接

- 1) 连接 “axi_pcie_0” 的 “M_AXI” 到 “axi_interconnect_0” 的 “S00_AXI”

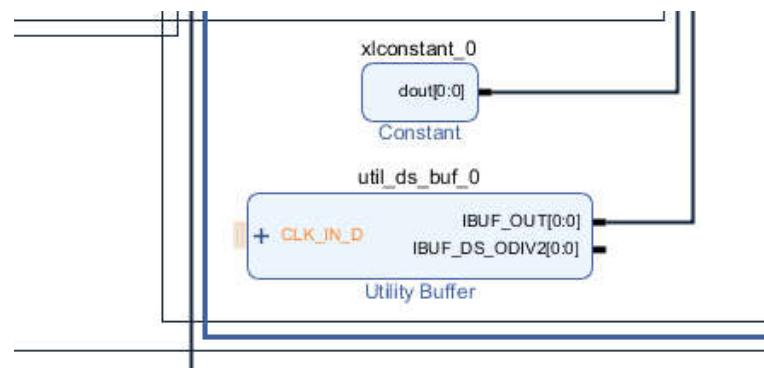


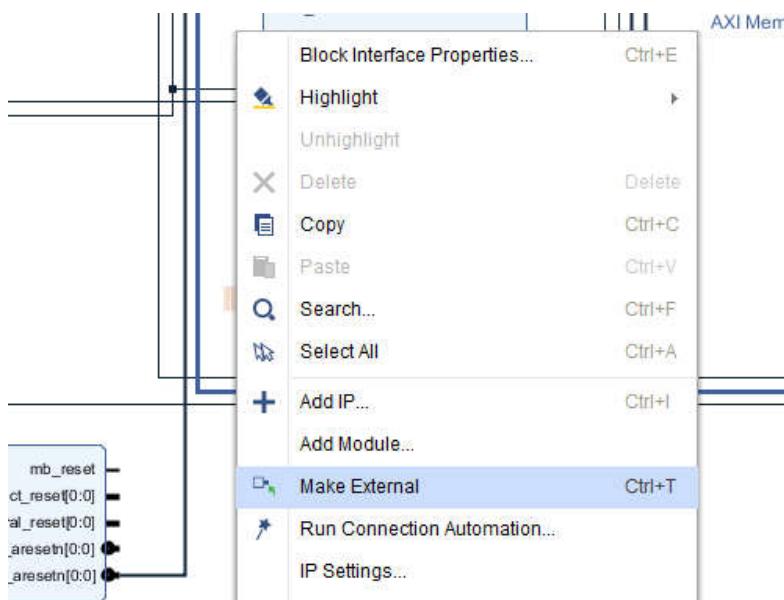
2) 连接 ZYNQ 处理器的 “M_AXI_GPO” 到 “axi_interconnect_1”的 “S00_AXI”



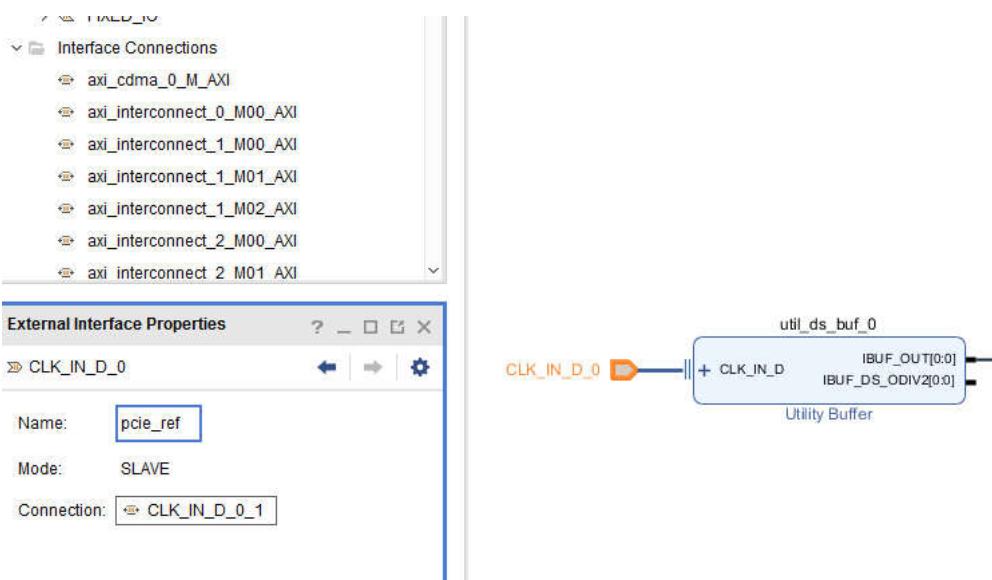
17.1.11 端口设置

1) util_ds_buf_0 差分时钟输入，右键 “Make External”

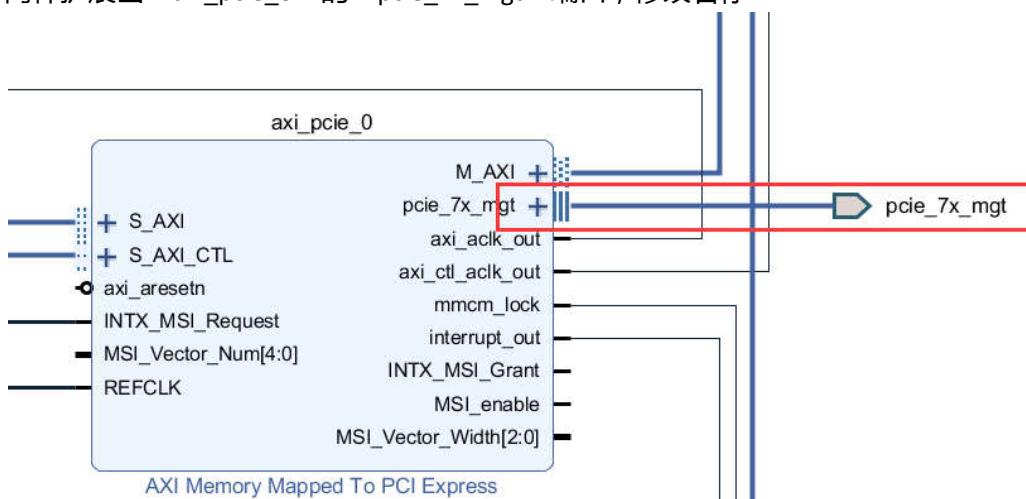




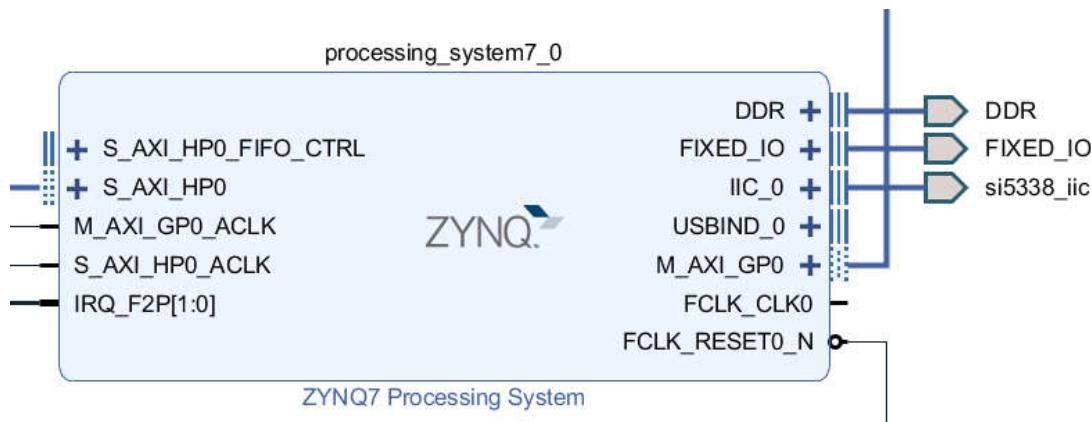
2) 修改名称为 “pcie_ref”



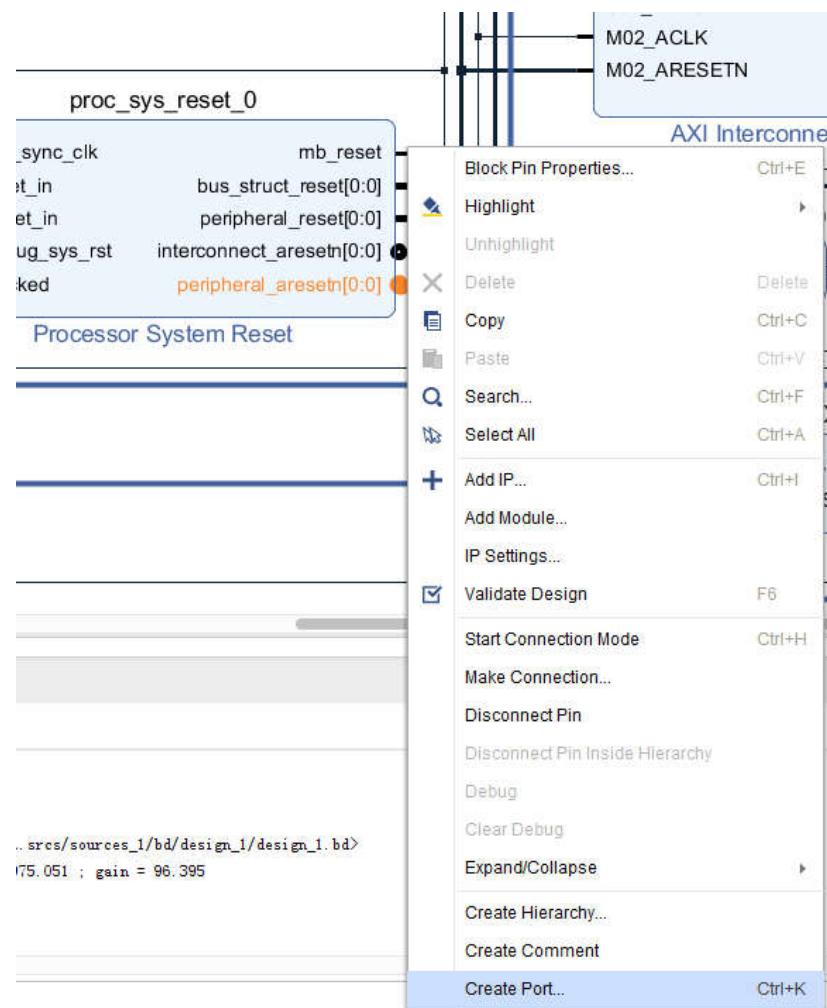
3) 同样扩展出 “axi_pcie_0”的“pcie_7x_mgt”端口，修改名称



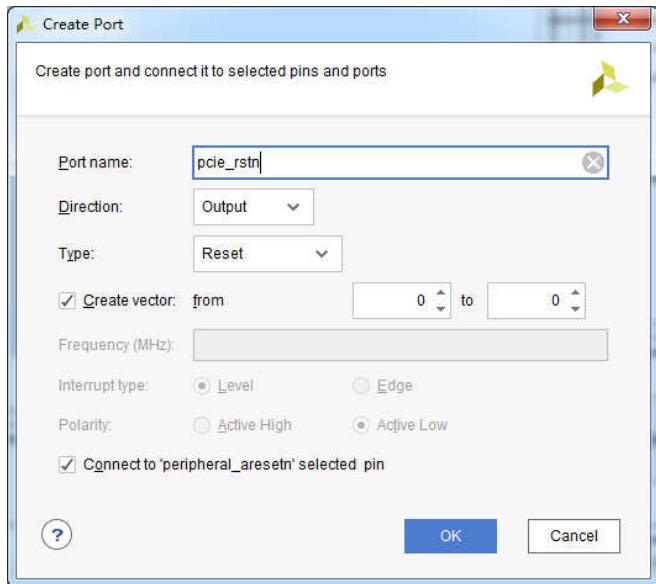
- 4) 将 IIC_0 端口引出，端口名为“si5338_iic”，用于配置 SI5338



- 5) 选择复位模块“proc_sys_reset_0”的输出“peripheral_aresetn” ,右键然后选择“Create Port...”

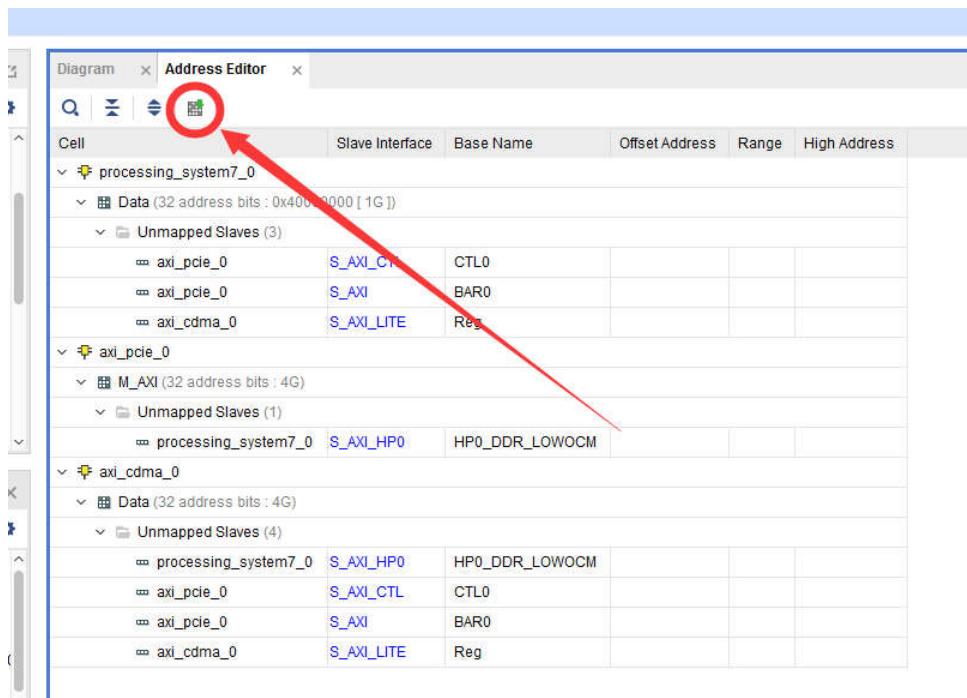


- 6) 端口名称填写“pcie_rstn”

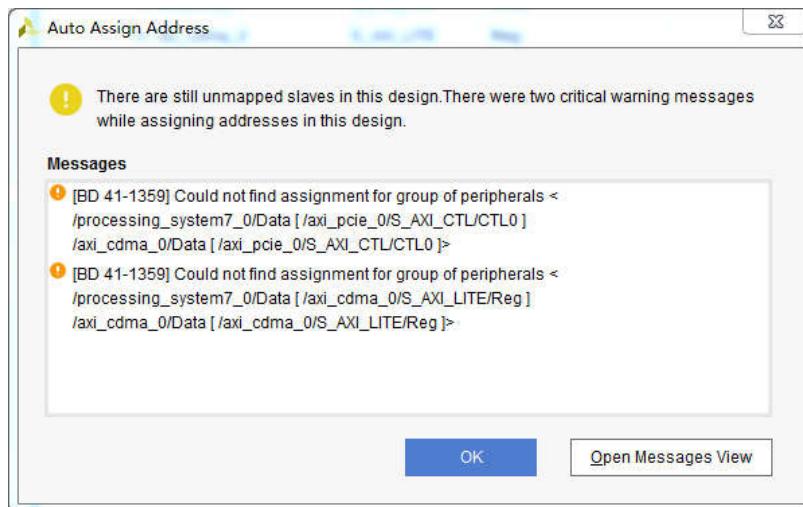


17.1.12 地址分配

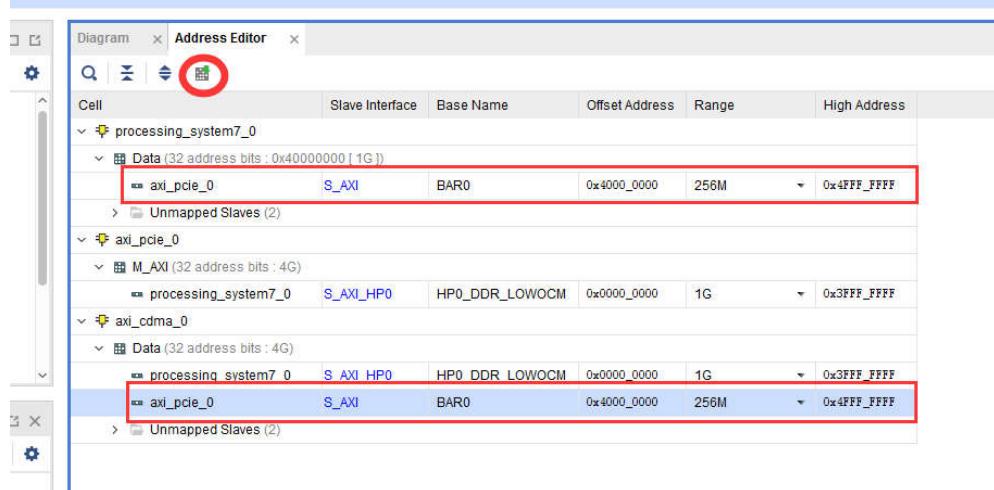
7) 在“Address Editor”窗口点击自动分配图标



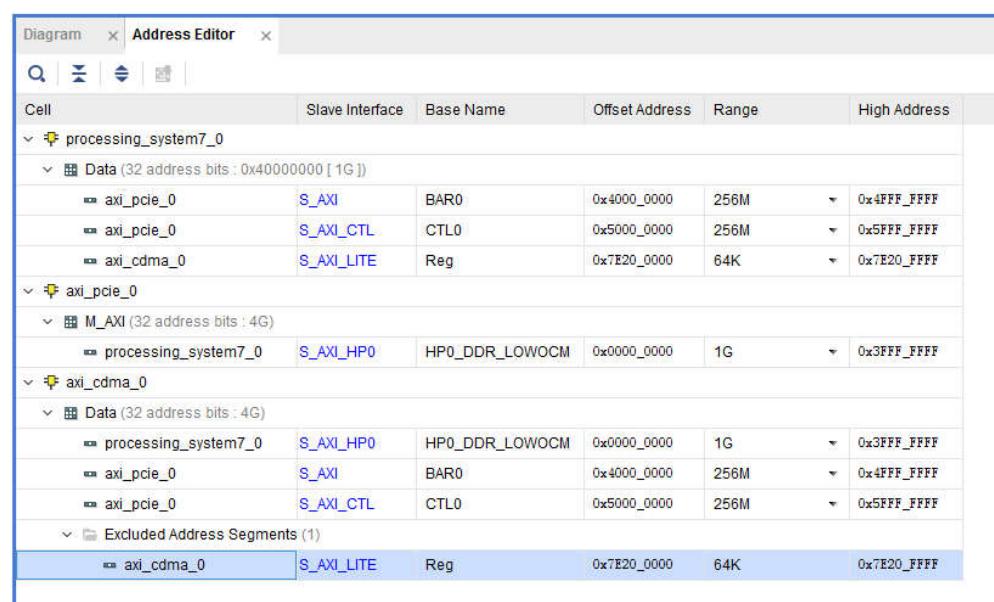
8) 出现一个警告



9) 把 axi_pcie_0 的 BAR0 地址范围改为 256M 后，重新点击自动分配

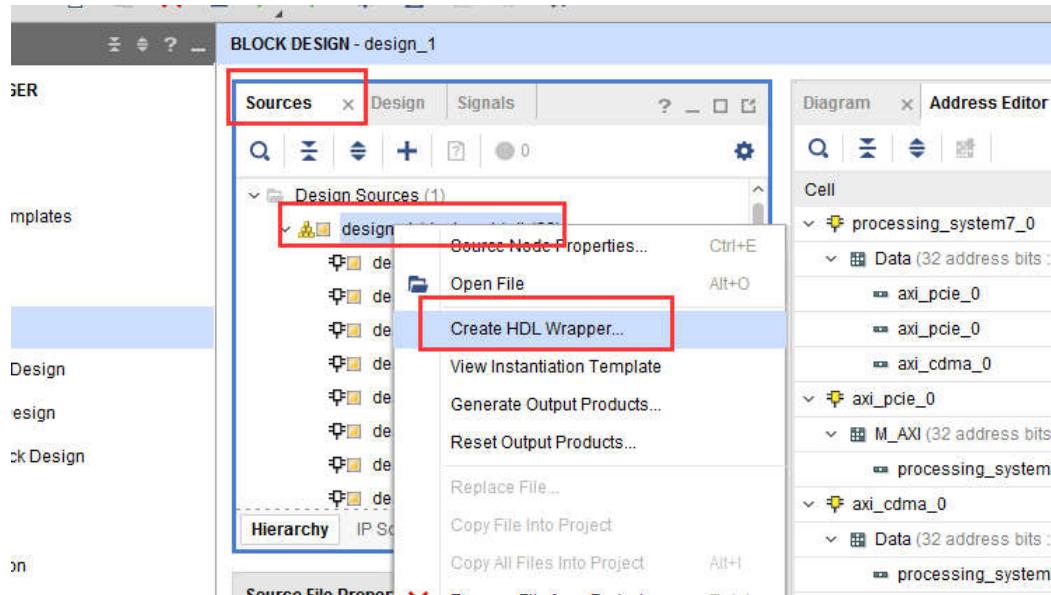


10) 重新分配以后的地址，然后保存 block 设计

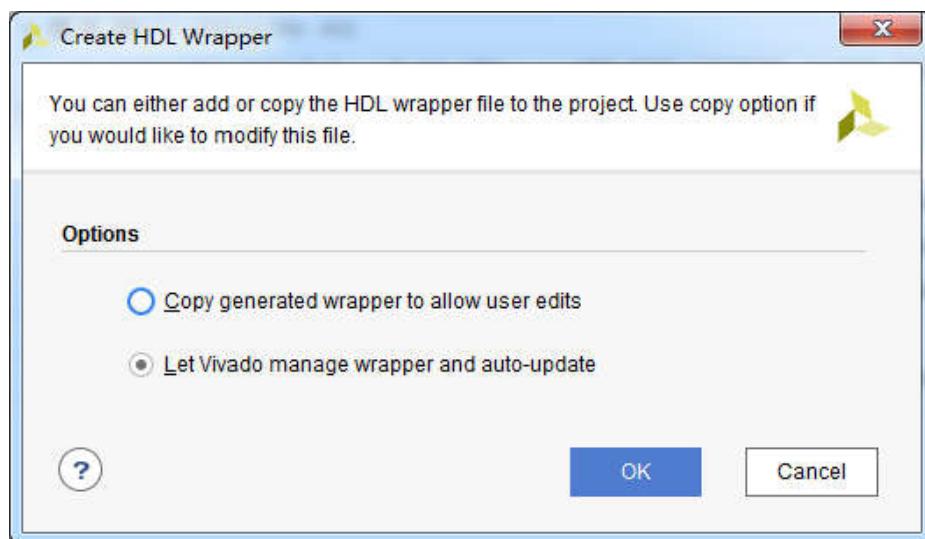


17.1.13 创建 HDL 封装

- 在 “Sources” 选项卡中选择 Block 设计右键，然后 “Create HDL wrapper”

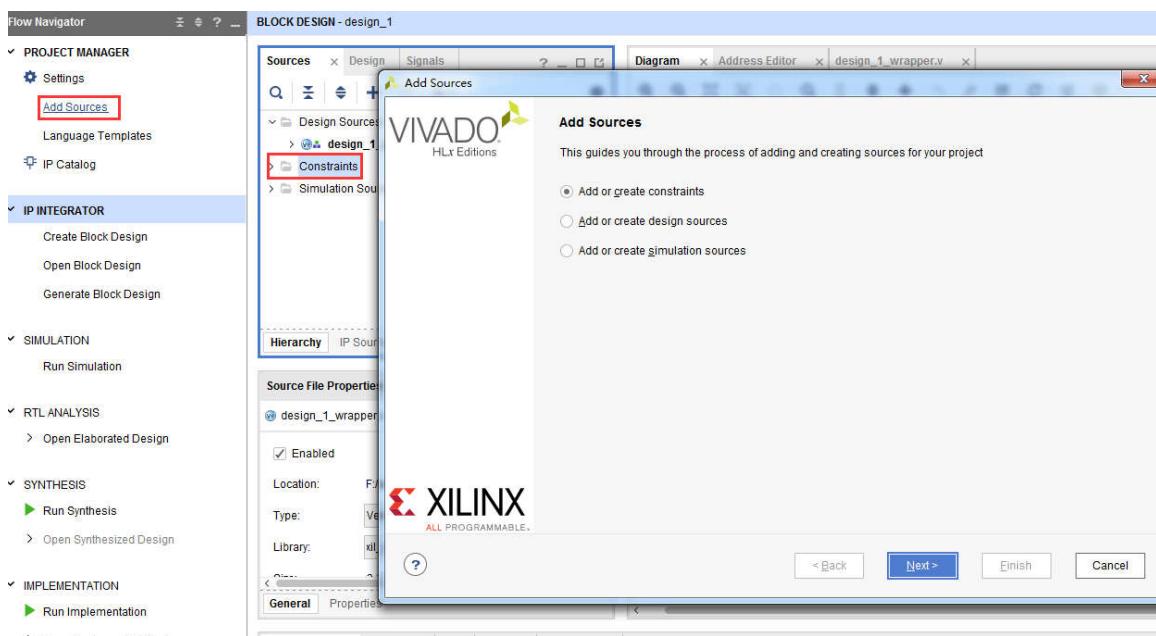


- 点击 “OK”

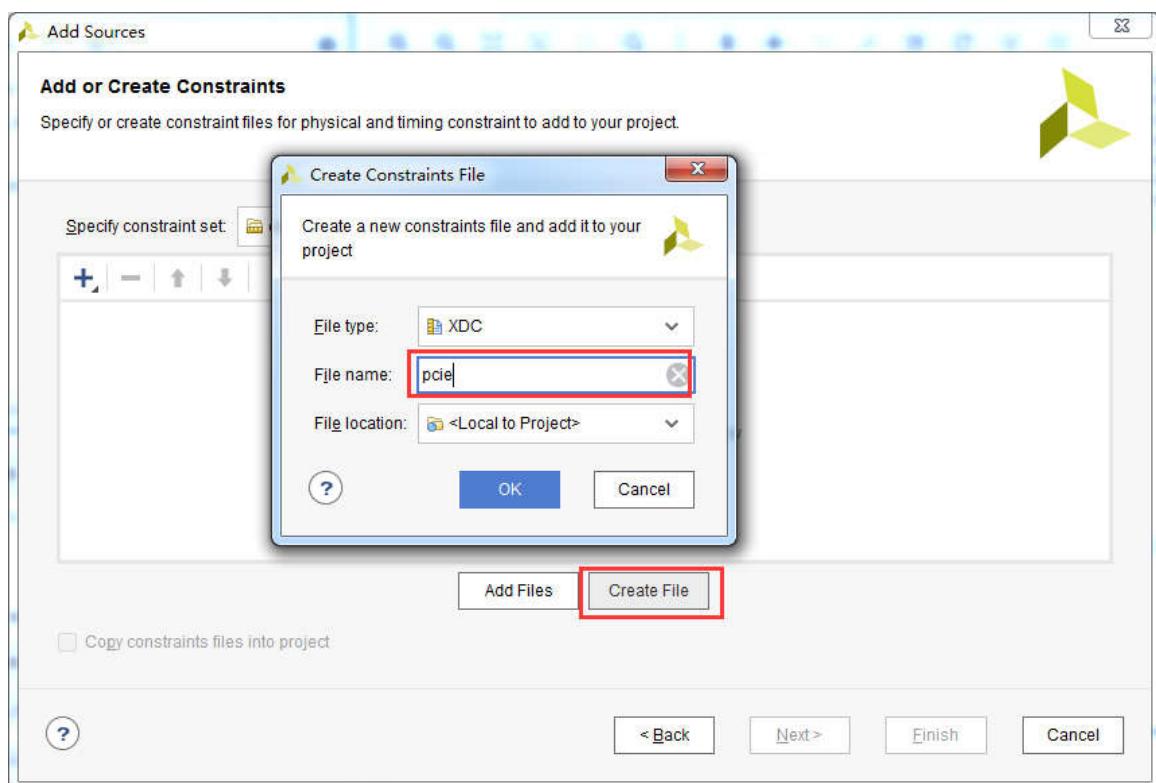


17.1.14 添加 xdc 约束

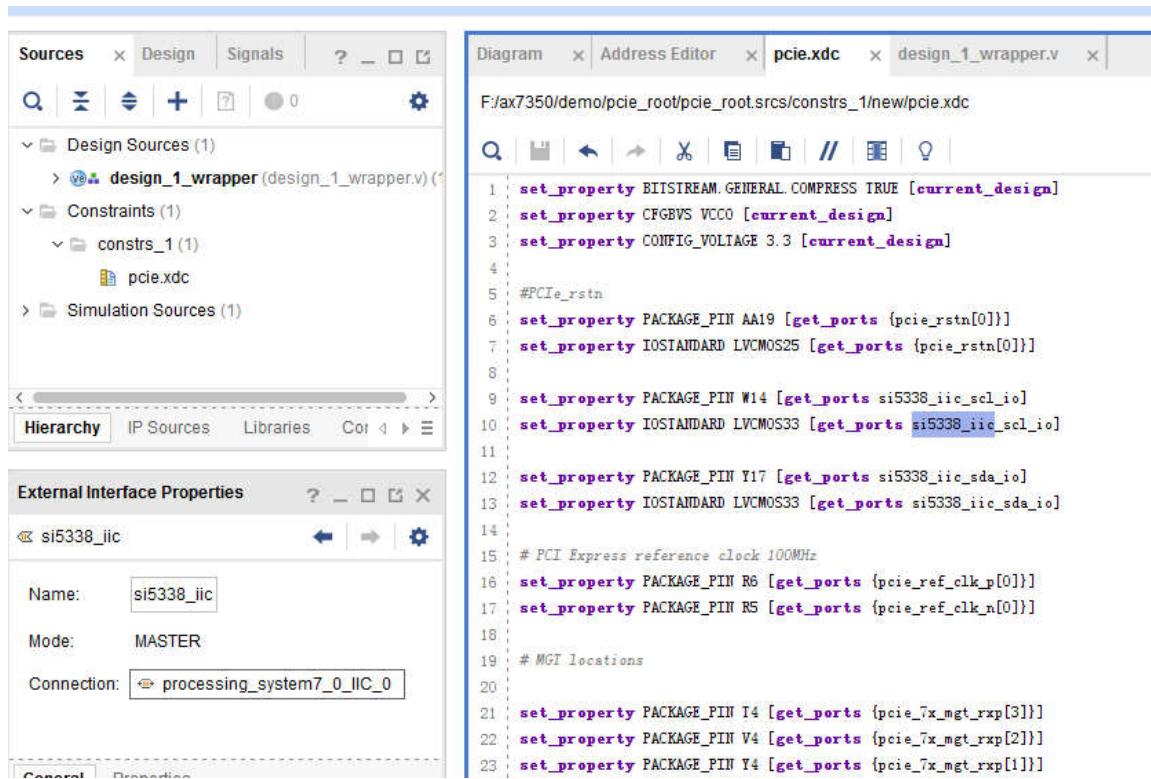
- 在约束选项下点击 “Add Sources”，在弹出的窗口选择 “Add or create constraints”



2) 点击“Create”，名称填写 pcie，然后点击“Finish”



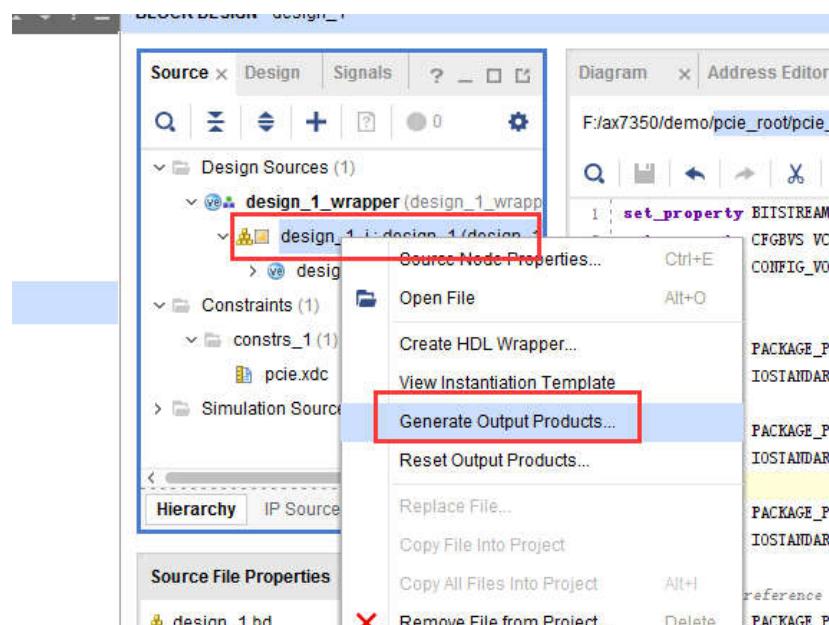
3) 修改 xdc 文件，文件内容不再给出，在开发板给的工程里可以找到
(pcie_root/pcie_root.srcs/constrs_1/new/pcie.xdc)



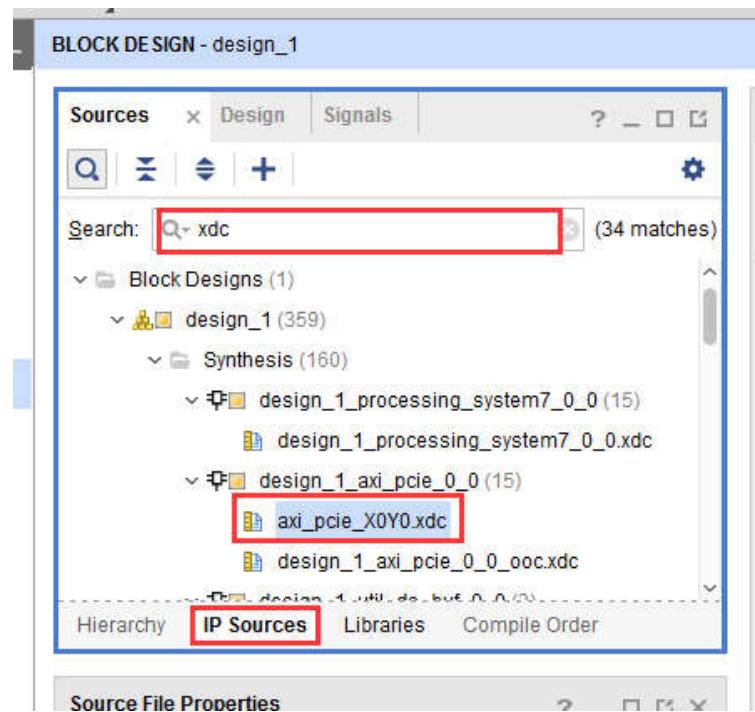
17.1.15 关键步骤

pcie 模块会自动带上管脚约束信息，但是和我们的硬件设计是不相符的，这个时候需要我们把自带的 xdc 文件添加不使用属性。

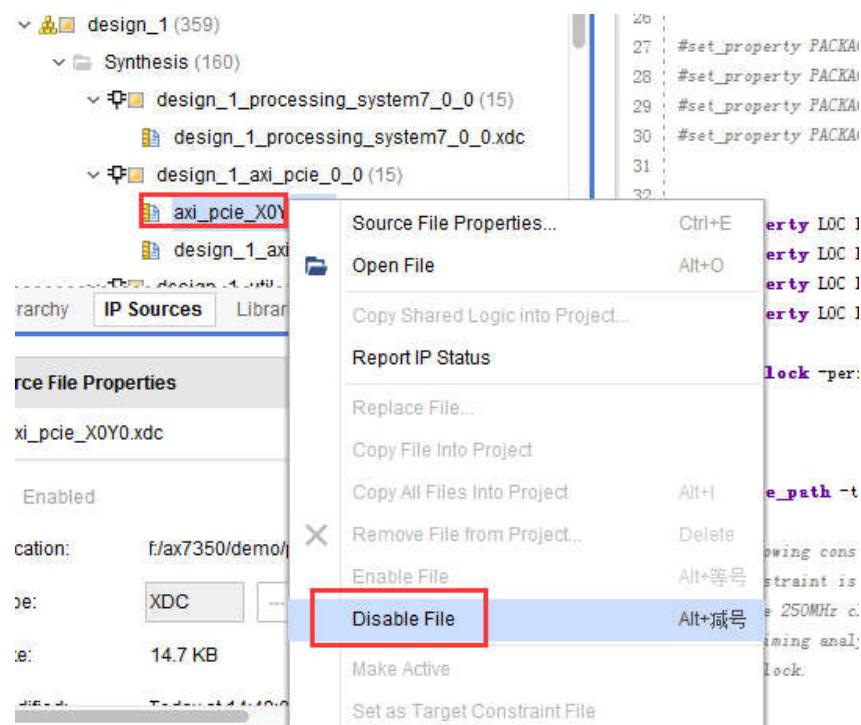
- 在模块设计 “design_1_i” 右键选择 “Generate Output Products...”



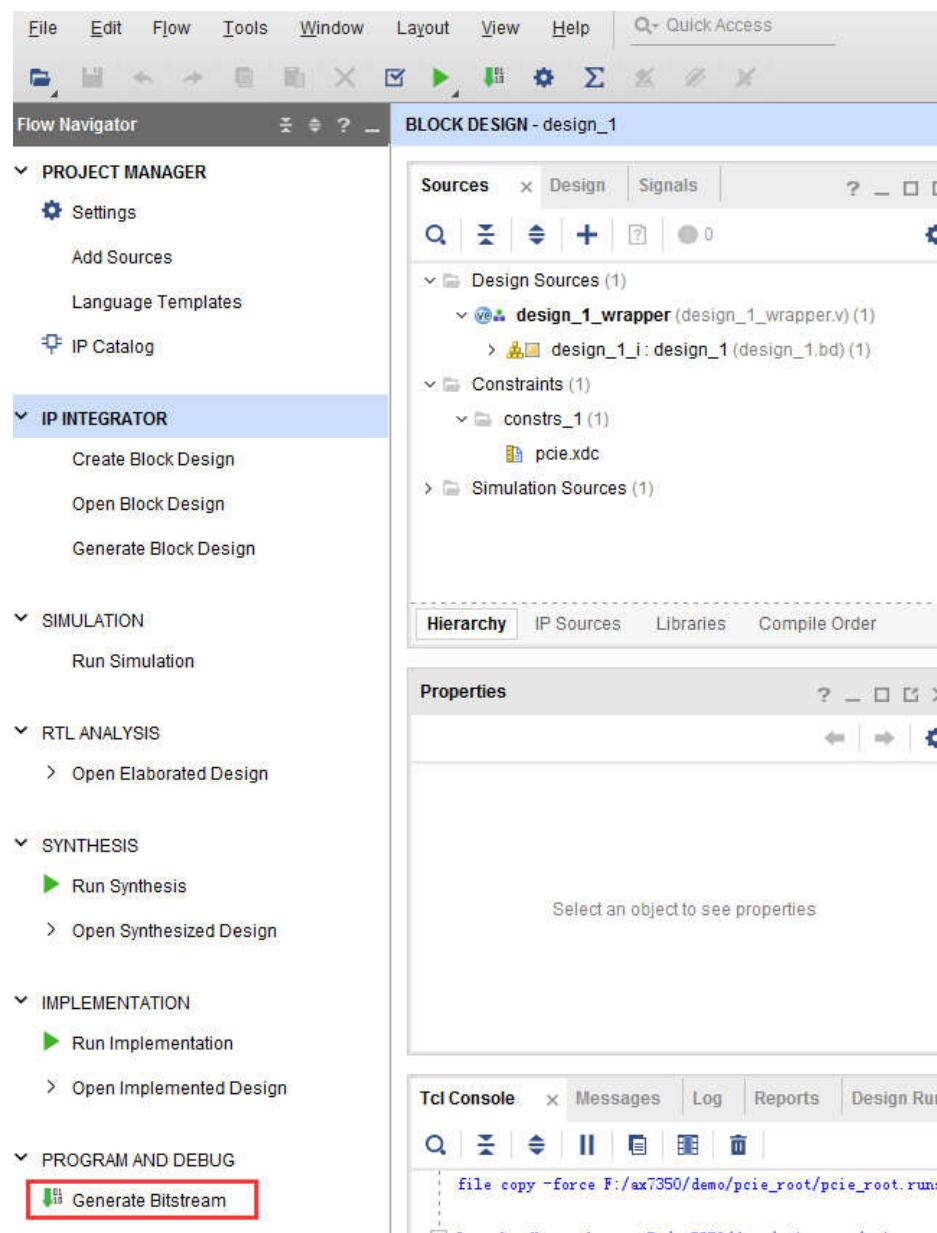
- 在 “IP Sources” 选项卡中搜索 xdc 文件，axi_PCIE_XOYO.xdc 就是我们要找的文件



3) 选择这个文件然后右键 “Disable File”

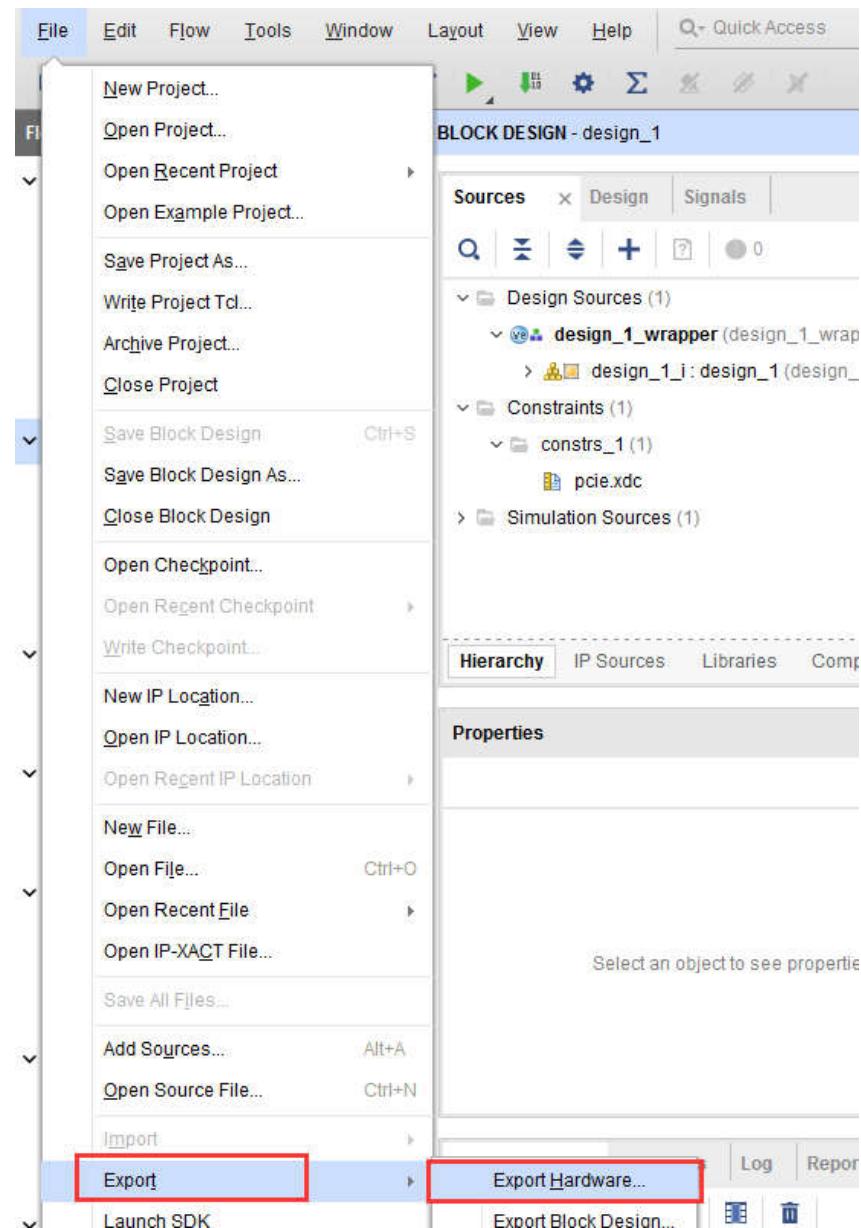


4) 编译生成 bit 文件

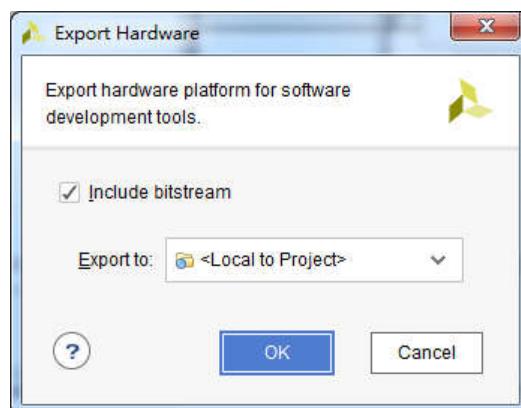


17.2 SDK 下载调试

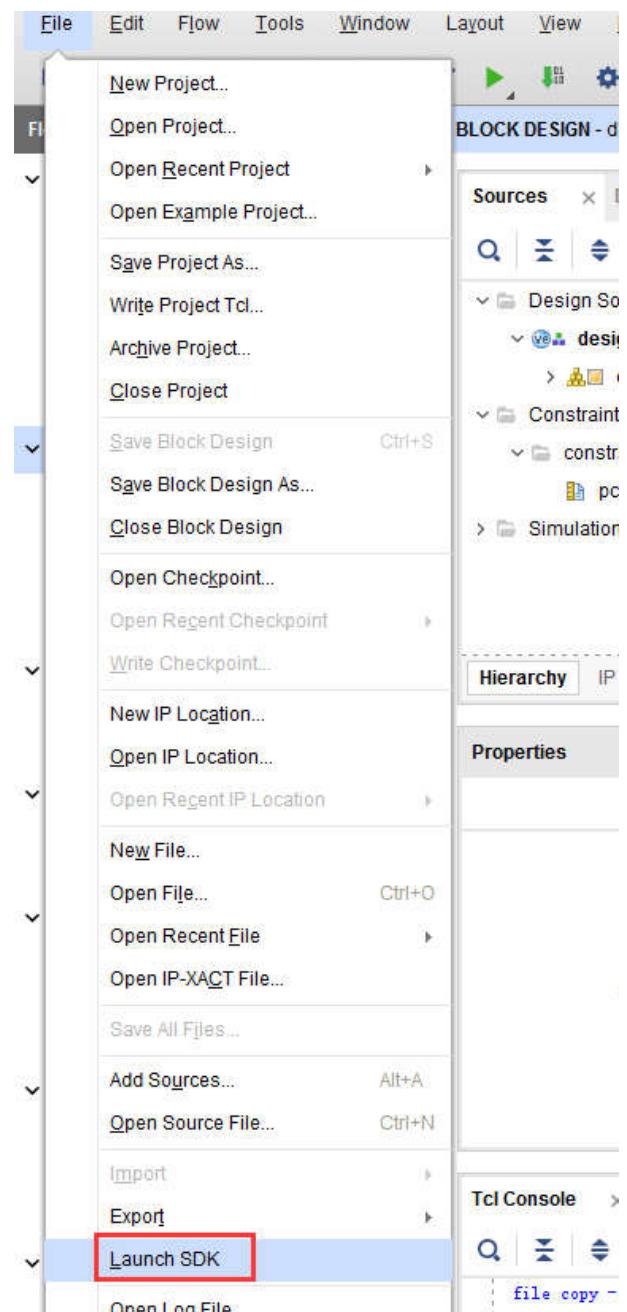
1) 导出硬件



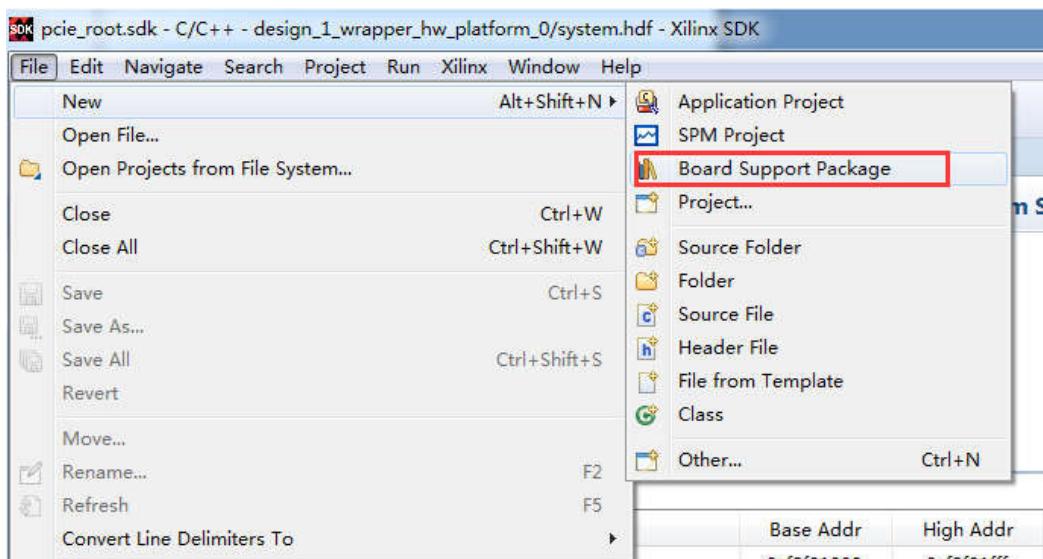
2) 选择包含 bit 文件



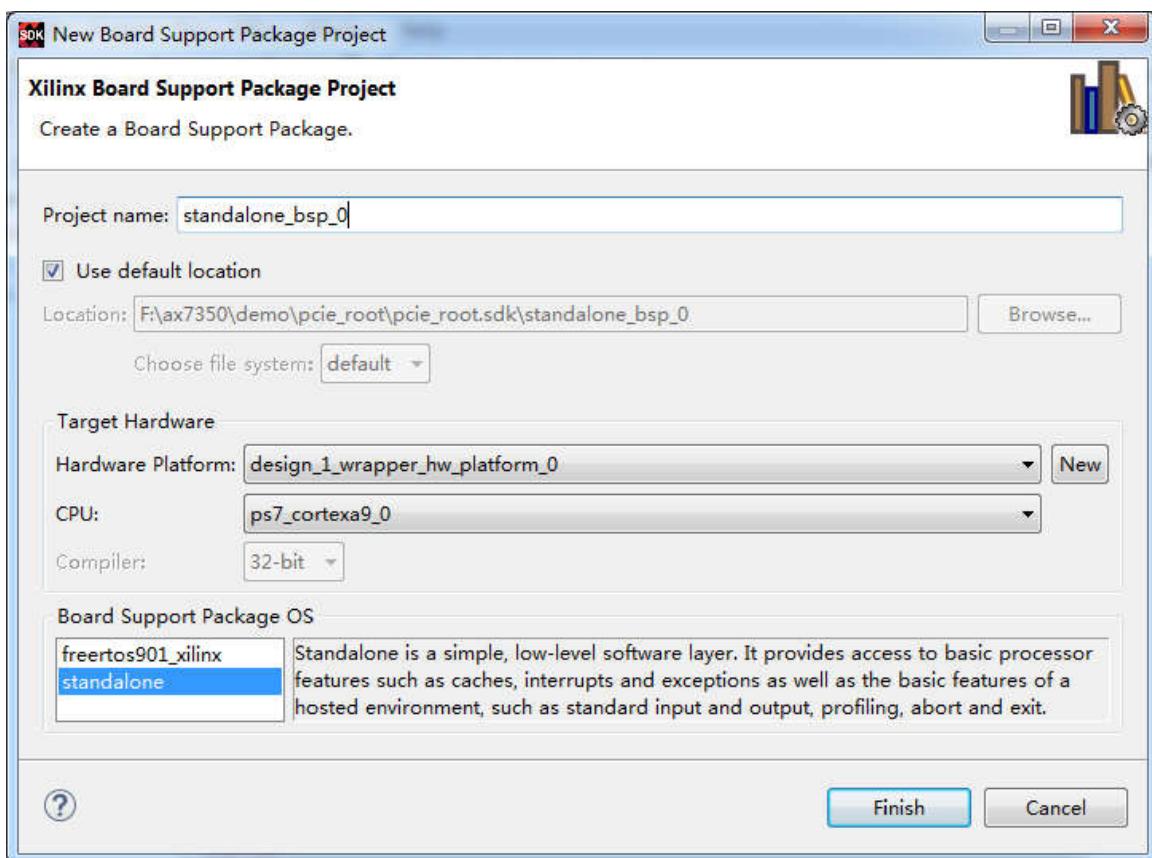
3) 运行 SDK



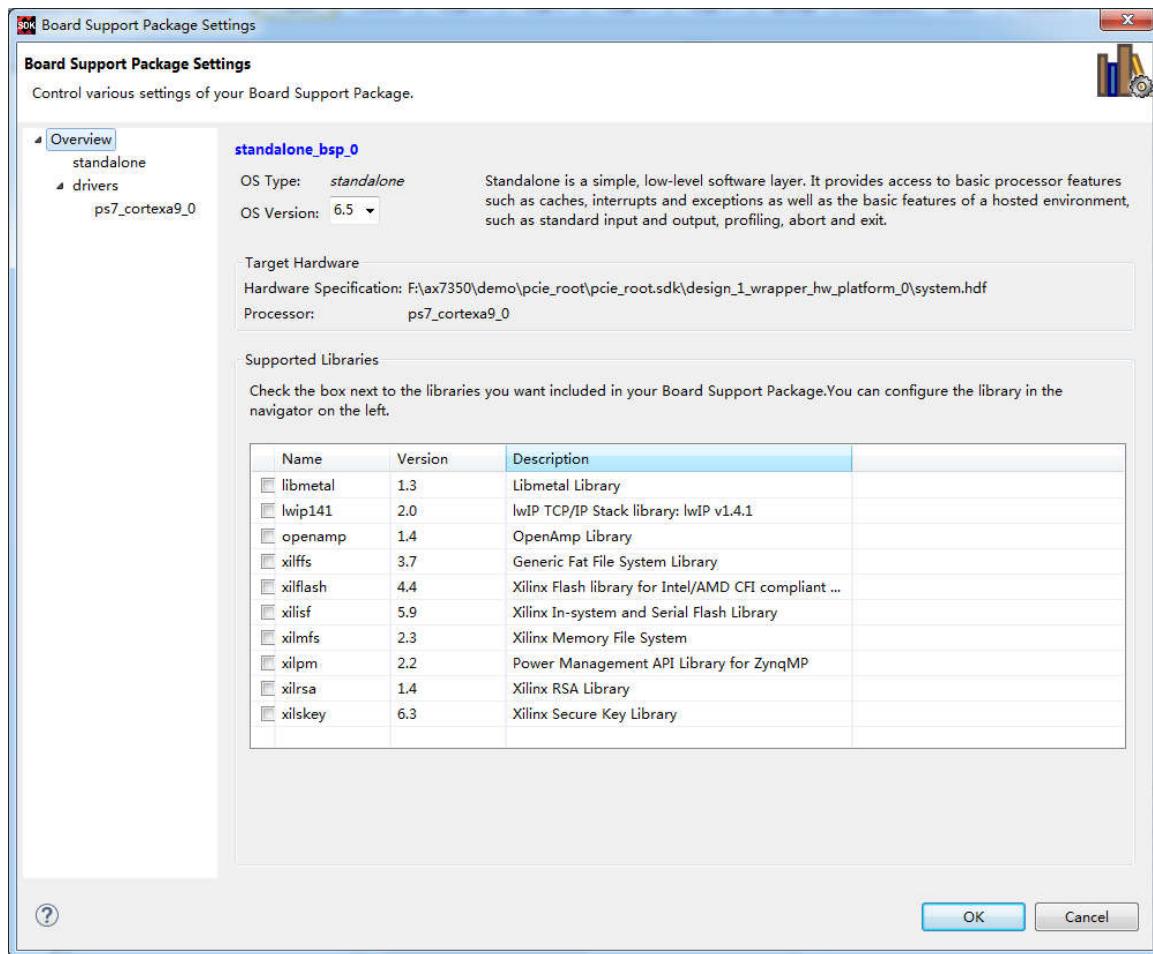
4) 新建一个 BSP



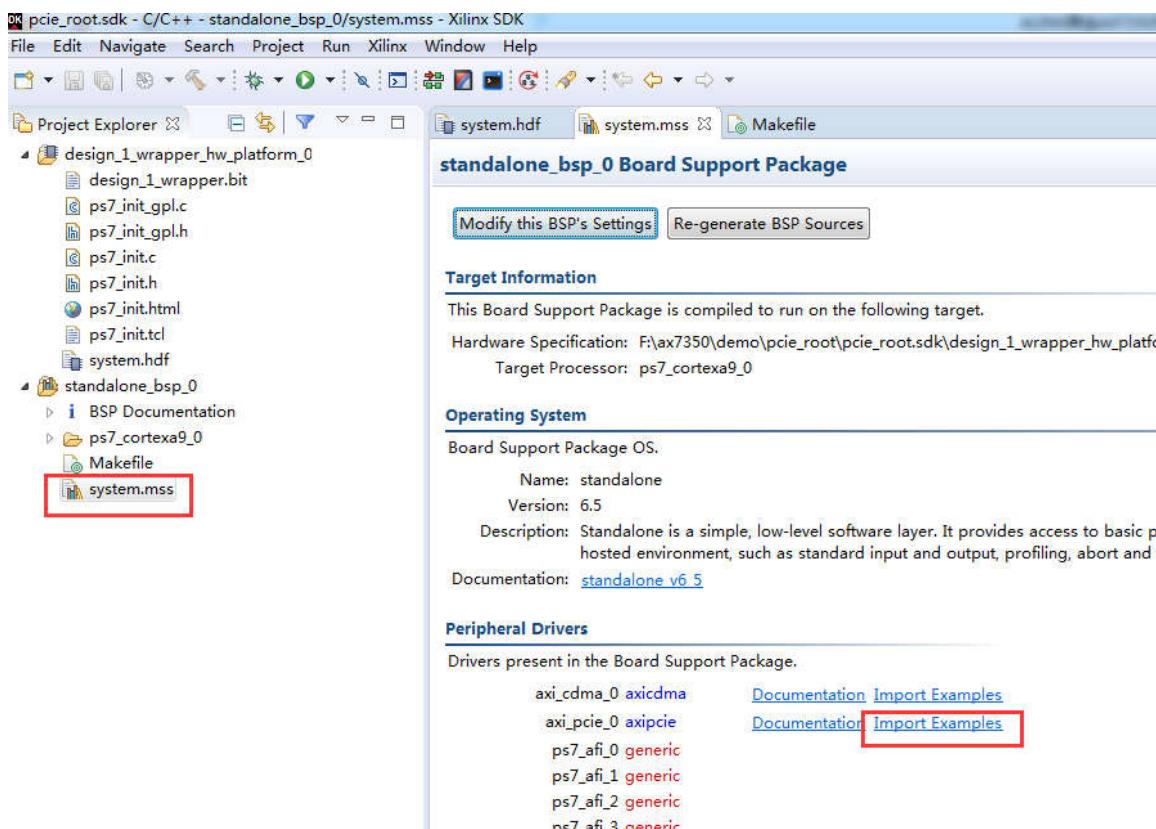
- 5) 保持默认的工程名，“Board Support Package OS”选择“standalone”



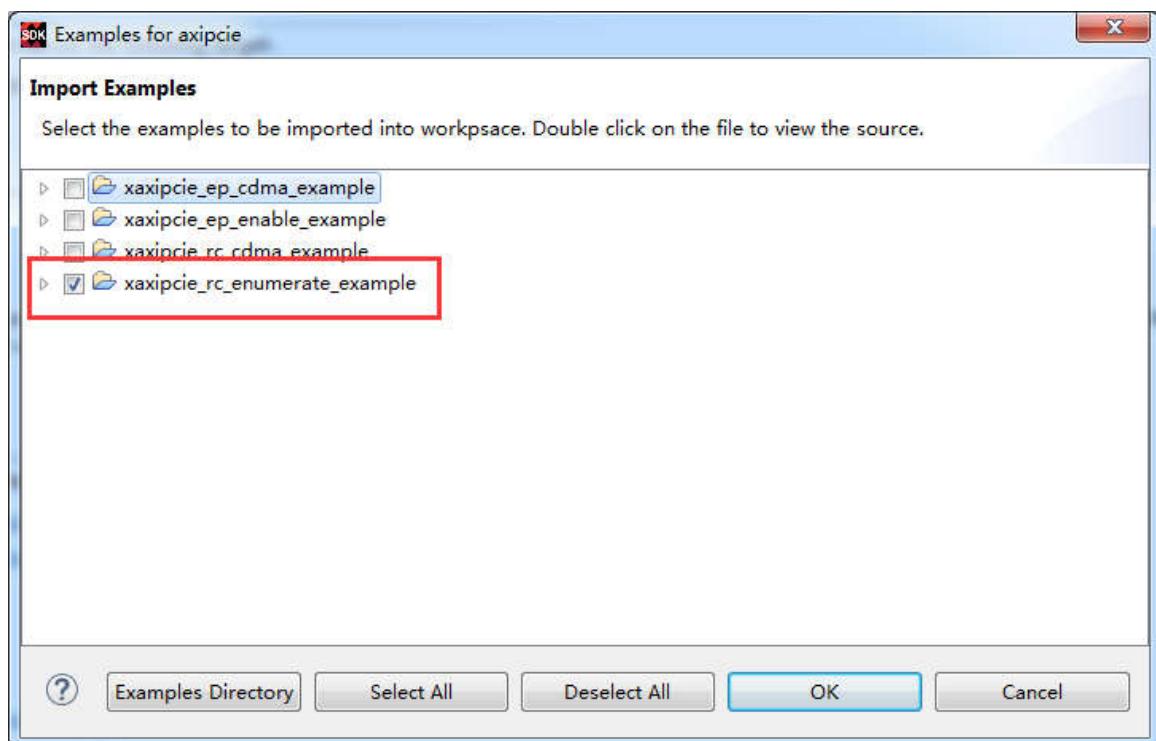
- 6) 保持默认配置，选择“OK”



- 7) 双击 “system.mss” ,在 “axi_pcie_0” 这一行选择 “Import Examples” , 导入一个例程



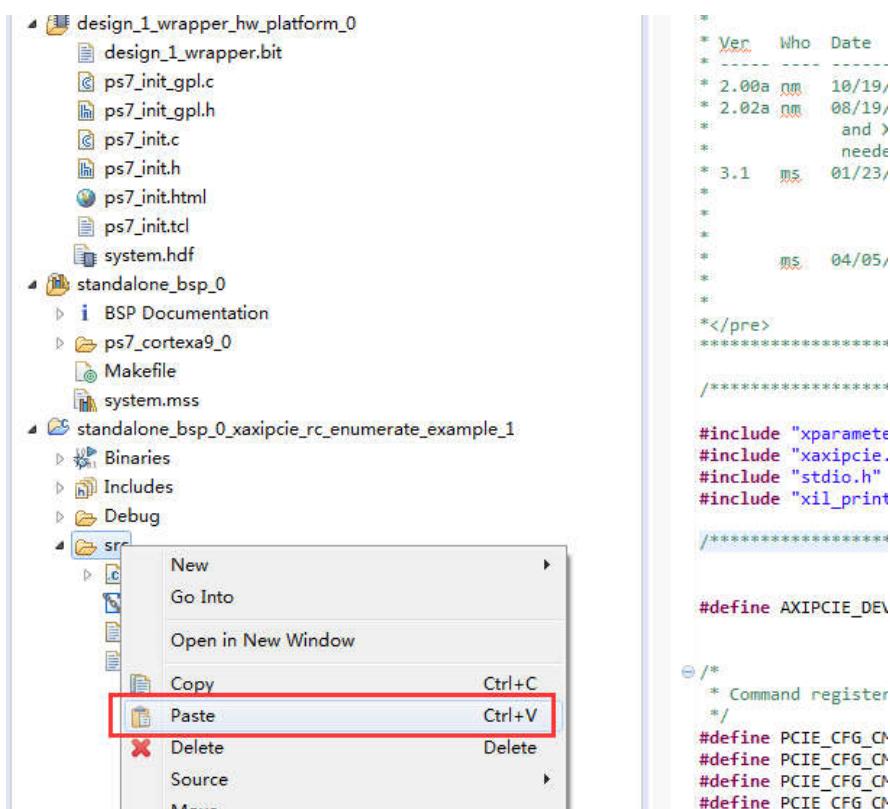
8) 选择 “xaxipcie_rc_enumerate_example”



9) 复制文件用于 si5338 的配置，相关文件可以到开发板例程的 sdk 目录找到

新建文件夹			
名称	修改日期	类型	大小
lscript.ld	2018/4/10 15:57	LD 文件	7 KB
platform.c	2015/11/18 7:06	C Source file	4 KB
platform.h	2015/11/18 7:06	C++ Header file	2 KB
platform_config.h	2017/12/14 17:47	C++ Header file	1 KB
README.txt	2018/4/10 15:57	文本文档	1 KB
register_map.h	2018/4/6 13:39	C++ Header file	9 KB
si5338.c	2017/2/16 17:17	C Source file	4 KB
si5338.h	2016/6/1 20:13	C++ Header file	1 KB
xaxipcie_rc_enumerate_example.c	2018/4/10 16:01	C Source file	14 KB
Xilinx.spec	2018/4/10 15:57	SPEC 文件	1 KB

10) 在 sdk 的工程目录选择 src 目录，右键 “Paste” 把上面复制的三个文件粘贴到 src 目录



11) 修改 “xaxipcie_rc_enumerate_example.c” 文件，添加 si5338 的相关配置，先添加头文件

```

/*
 *      needed for enumeration
 * 3.1 ms 01/23/17 Added xil_printf statement in main function to
 *      ensure that "Successfully ran" and "Failed" strings
 *      are available in all examples. This is a fix for
 *      CR-965028.
 * ms 04/05/17 Added tabspace for return statements in functions
 *      for proper documentation while generating doxygen.
 */

/*</pre>
*****
***** Include Files *****/
#include "xparameters.h" /* Defines for XPAR constants */
#include "xaxipcie.h" /* XAxiPcie level 1 interface */
#include "stdio.h"
#include "xil_printf.h"
#include "sleep.h"
#include "si5338.h"

***** Constant Definitions *****/
#define AXI_PCIE_DEVICE_ID XPAR_AXI_PCIE_0_DEVICE_ID

/*
 * Command register offsets
 */
#define PCIE_CFG_CMD_IO_EN 0x00000001 /* I/O access enable */
#define PCIE_CFG_CMD_MEM_EN 0x00000002 /* Memory access enable */
#define PCIE_CFG_CMD_BUSM_EN 0x00000004 /* Bus master enable */
#define PCIE_CFG_CMD_PARITY 0x00000040 /* parity errors response */
#define PCIE_CFG_CMD_SERR_EN 0x00000100 /* SERR report enable */

/*
 * PCIe Configuration registers offsets
*/

```

12) 在 PcieInitRootComplex 函数中添加 si5338 的配置

```

/*
 * @param AxiPciePtr is a pointer to an instance of XAxiPcie data
 * structure represents a root complex IP.
 * @param DeviceId is AXI PCIe IP unique ID
 *
 * @return
 * - XST_SUCCESS if successful.
 * - XST_FAILURE if unsuccessful.
 *
 * @note None.
 */

int PcieInitRootComplex(XAxiPcie *AxiPciePtr, u16 DeviceId)
{
    int Status;
    u32 HeaderData;
    u32 InterruptMask;
    u8 BusNumber;
    u8 DeviceNumber;
    u8 FunctionNumber;
    u8 PortNumber;

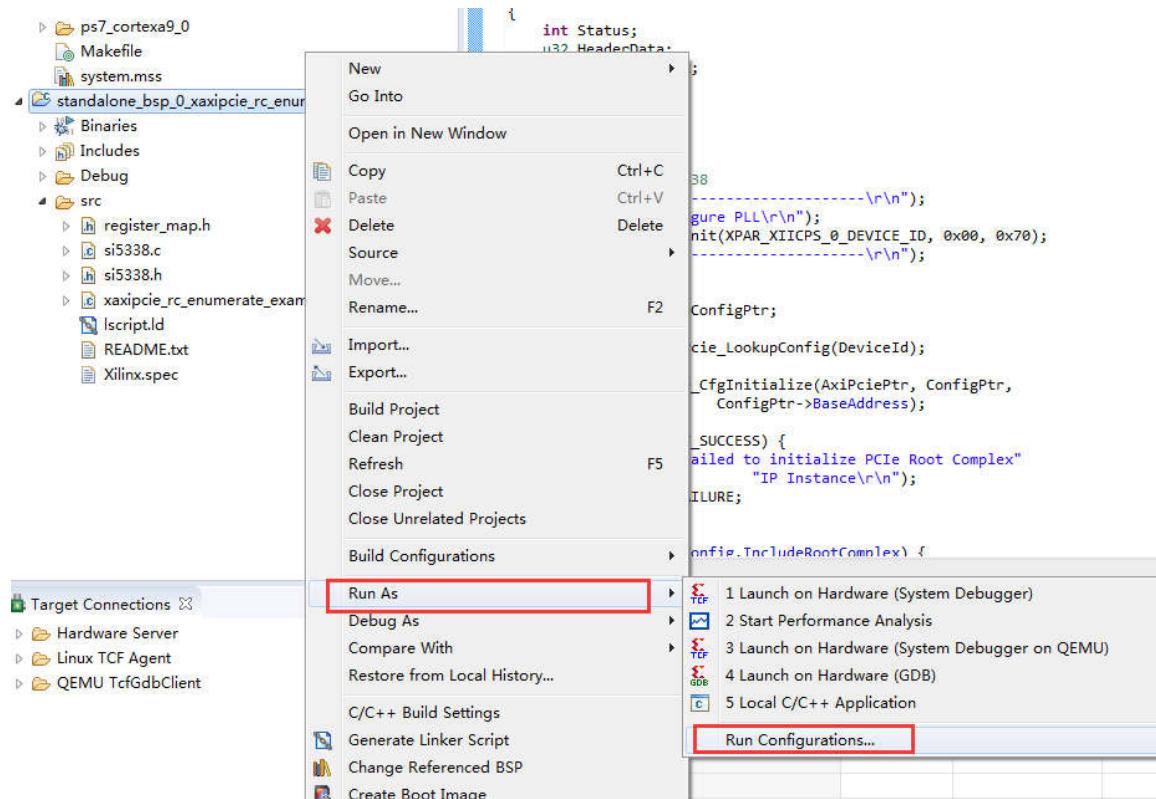
    // Configure Si5338
    xil_printf("-----\r\n");
    xil_printf("Configure PLL\r\n");
    Status = si5338_iinit(XPAR_XIICPS_0_DEVICE_ID, 0x00, 0x70);
    xil_printf("-----\r\n");
    usleep(100*1000);

    XAxiPcie_Config *ConfigPtr;
    ConfigPtr = XAxiPcie_LookupConfig(DeviceId);
    Status = XAxiPcie_CfgInitialize(AxiPciePtr, ConfigPtr,
                                    ConfigPtr->BaseAddress);

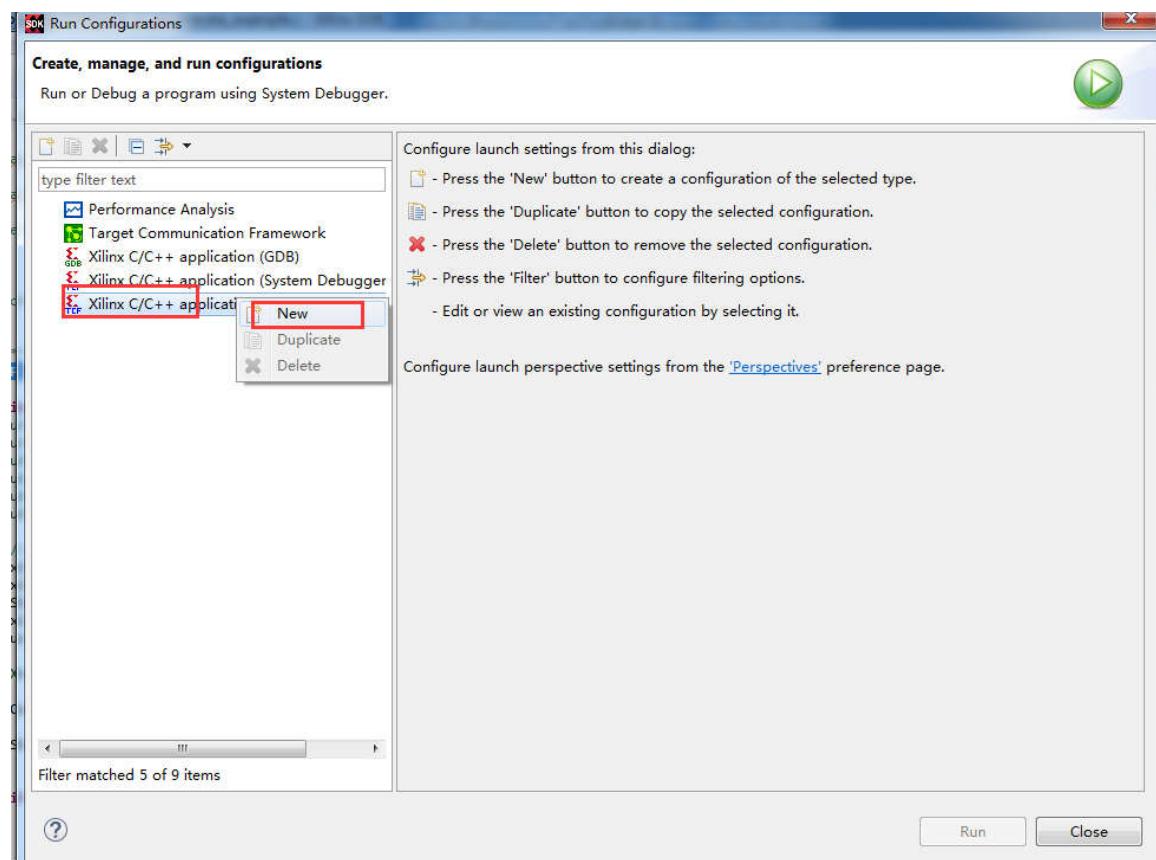
    if (Status != XST_SUCCESS) {
        ...
    }
}

```

13) 打开运行配置

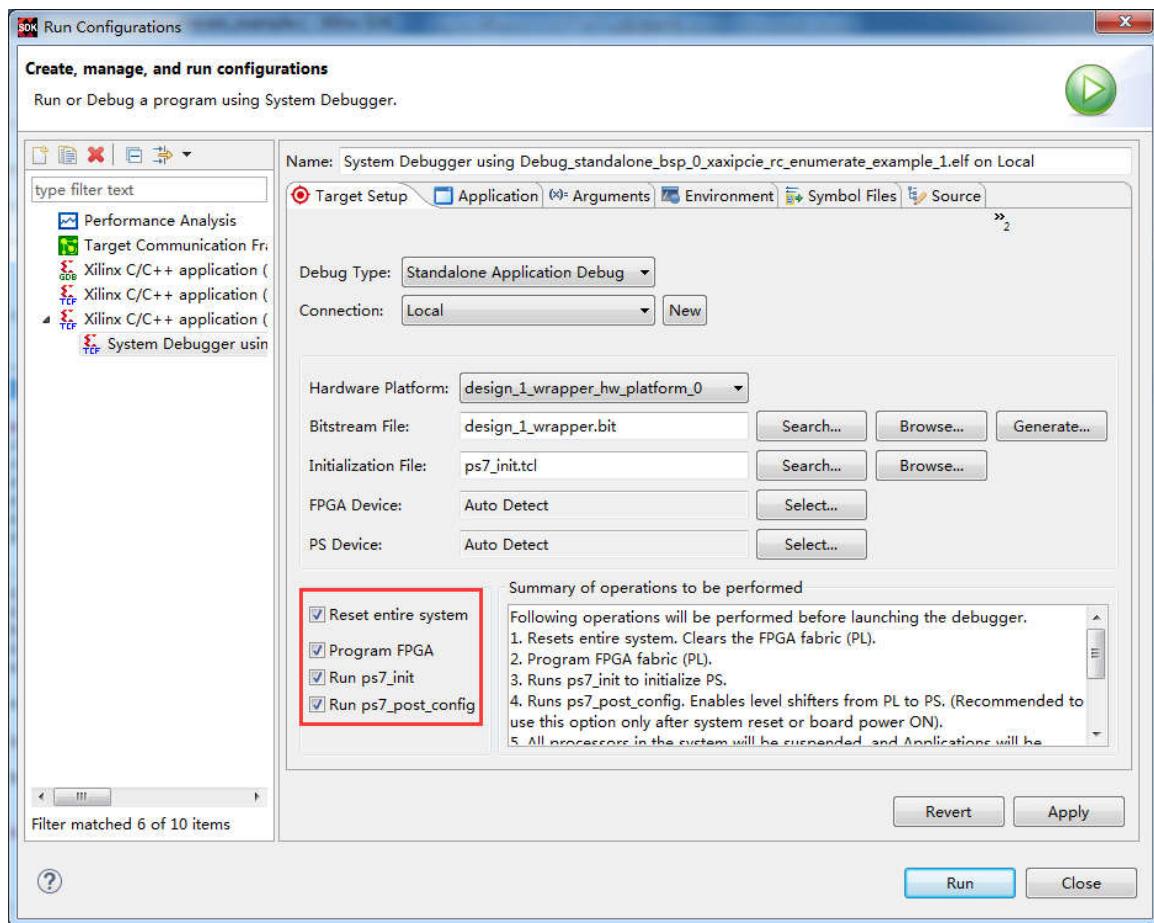


14) 选择 “Xilinx C/C++ application (System Debugger)” 右键 “New”



15) 在 PCIe 插槽插入 PCIe 设备，例如 SSD，网卡等，连接串口调试终端，选择复位整个系统并

编程 FPGA , 开发板上电后点击 “Run”



16) 串口终端软件输入了一些信息，包括 Si5338 配置，PCIe 设备的一些信息

```
Configure PLL
Si5338 Rev 1 Initialization      Done
-----
Interrupts currently enabled are   0
Interrupts currently pending are   0
Interrupts currently enabled are   0
Interrupts currently pending are   0
Link is up
Bus Number is 00
Device Number is 00
Function Number is 00
Port Number is 00
PCIe Local Config Space is 100147 at register CommandStatus
PCIe Local Config Space is 70100 at register Prim Sec. Bus
Root Complex IP Instance has been successfully initialized
Start Enumeration of PCIe Fabric on This System
PCIeBus is 00
PCIeDev is 00
PCIeFunc is 00
Vendor ID is 10EE
This is a Bridge
PCIeBus is 01
PCIeDev is 00
PCIeFunc is 00
Vendor ID is 144D
This is an End Point
End Point has been enabled
End of Enumeration of PCIe Fabric on This system
Successfully ran Axipcie rc enumerate Example
```

17.3 实验总结

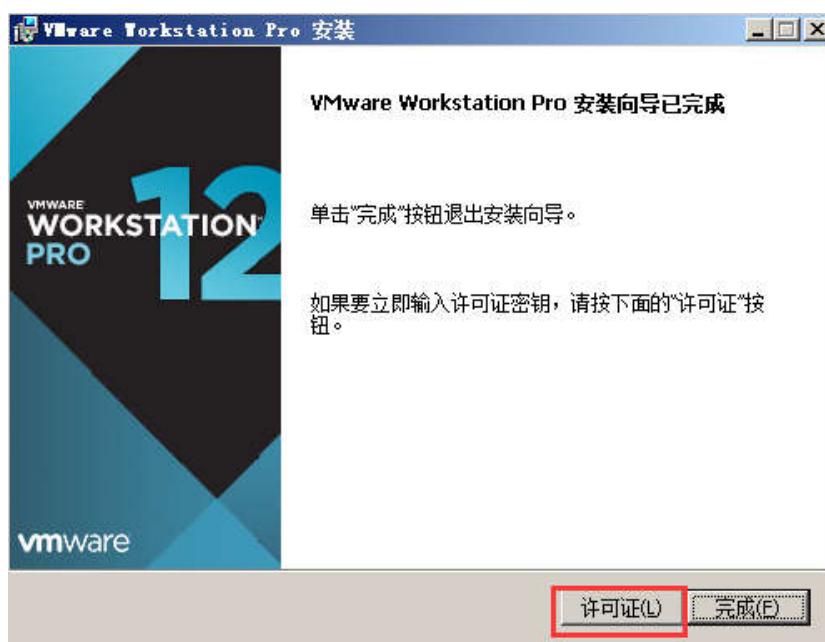
本实验建立了 PCIe Root 的 Vivado 工程，然后通过 SDK 裸机程序测试了 PCIe 枚举设备的功能，为我们 PCIe Root 应用打下基础，后面教程我们会通过 Linux 来使用 PCIe Root 功能。

第十八章 安装虚拟机和 Ubuntu 系统

后面的教程要涉及到嵌入式 Linux 开发，一般需要一台 Linux 操作系统主机，用来编译 u-boot 或者 Linux-kernel。在 Windows 操作系统的电脑上安装个虚拟机，再在虚拟机上安装 Linux 操作系统是最便捷的方法了。

18.1 虚拟机软件安装

我们提供的虚拟机的安装软件版本为 VMware-workstation-full 12.1.1。用户可以在我们提供的资料里找到，双击“VMware-workstation-full-12.1.1-3770994.exe”图标开始安装。因为安装比较简单，所以具体的安装步骤我们这里不做具体介绍了，用户只要按照默认安装项一直点“Next”按键来进行安装。安装完成的最后一个界面里，我们需要选择许可证来输入一个 VMware12 的系列号。





安装完成后，桌面上有 VMware Workstation Pro 的图标。



18.2 Ubuntu 安装

18.2.1 安装系统

安装好虚拟机后，就要在虚拟机上安装 Linux 操作系统了。鉴于 ubuntu Linux 桌面操作系统的安装以及配置较为简单，所以我们选择了 ubuntu 桌面操作系统。

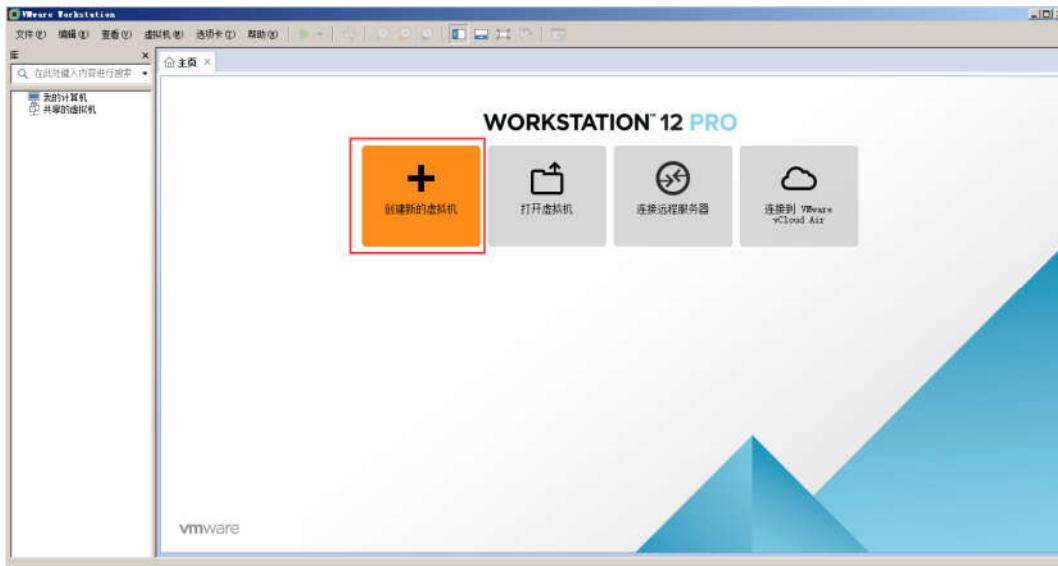
本教程使用 Ubuntu 16.04.3 LTS 64 位操作系统

下载地址：<http://releases.ubuntu.com/16.04/ubuntu-16.04.3-desktop-amd64.iso>

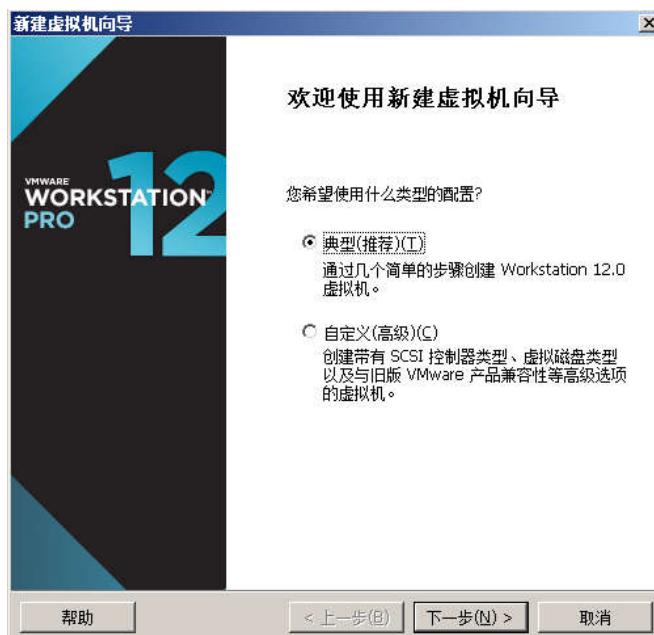
程和例程就是基于该版本。

ubuntu 安装步骤如下：

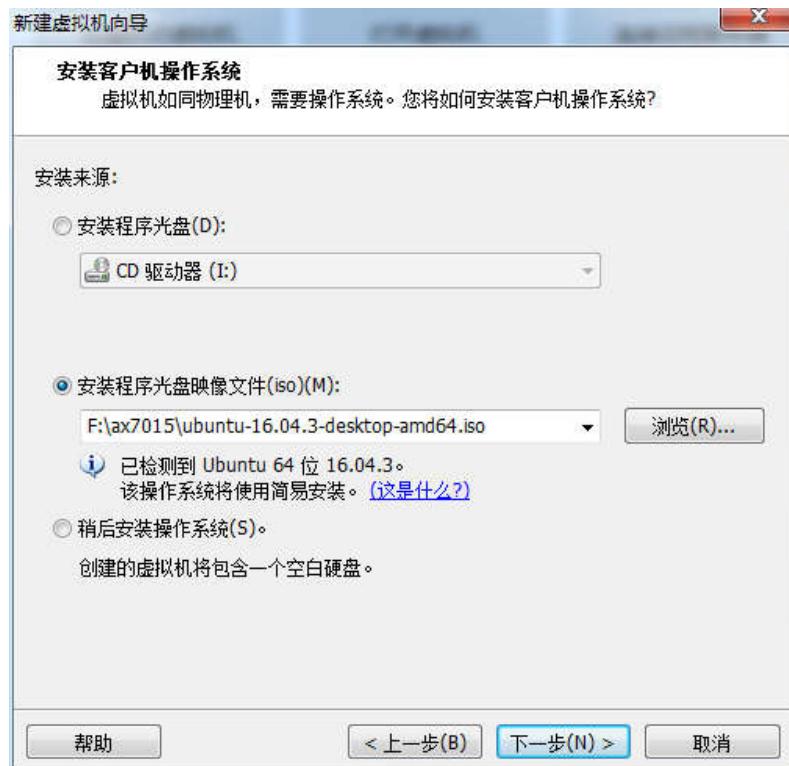
- 1) 双击桌面的 VMware Workstation Pro 的图标，然后在 VMware 工作界面上点击 “创建新的虚拟机” 图标。



2) 选择典型，下一步。



3) 选择"安装程序光盘映像文件(iso)"项，然后点击浏览找到 ubuntu 的光盘映像文件“ubuntu-16.04.3-desktop-amd64.iso”。



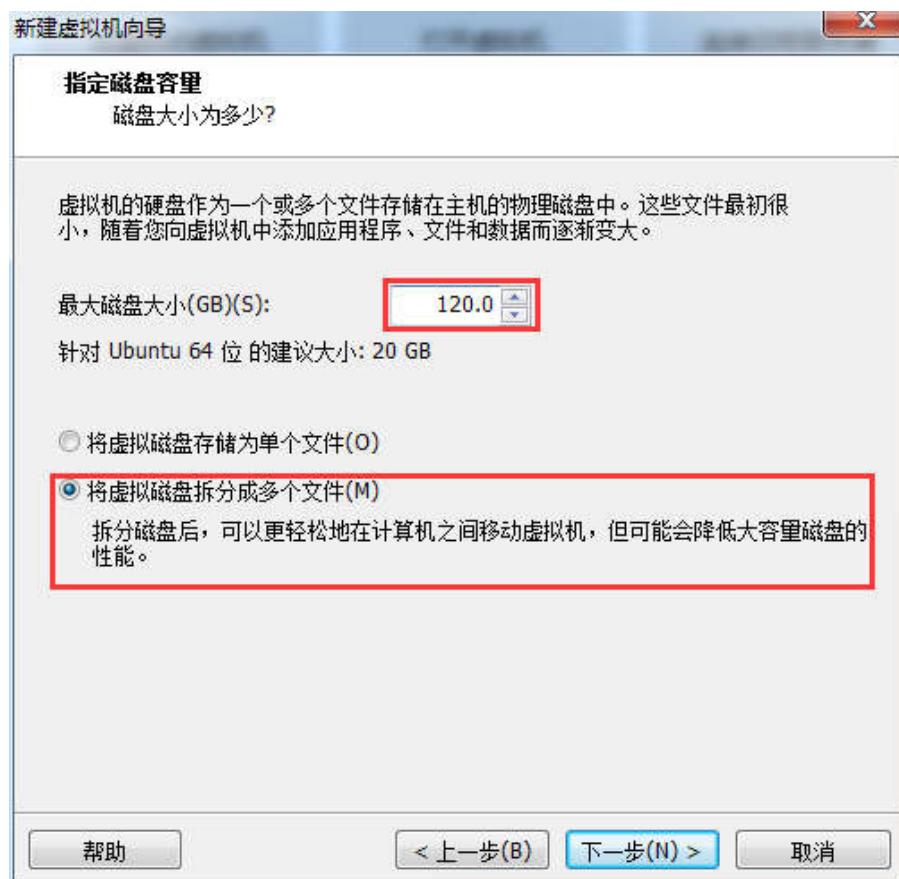
- 4) 在虚拟机向导里输入虚拟机的全名，用户名和密码。这里的全名，用户名和密码用户可以自行设置。



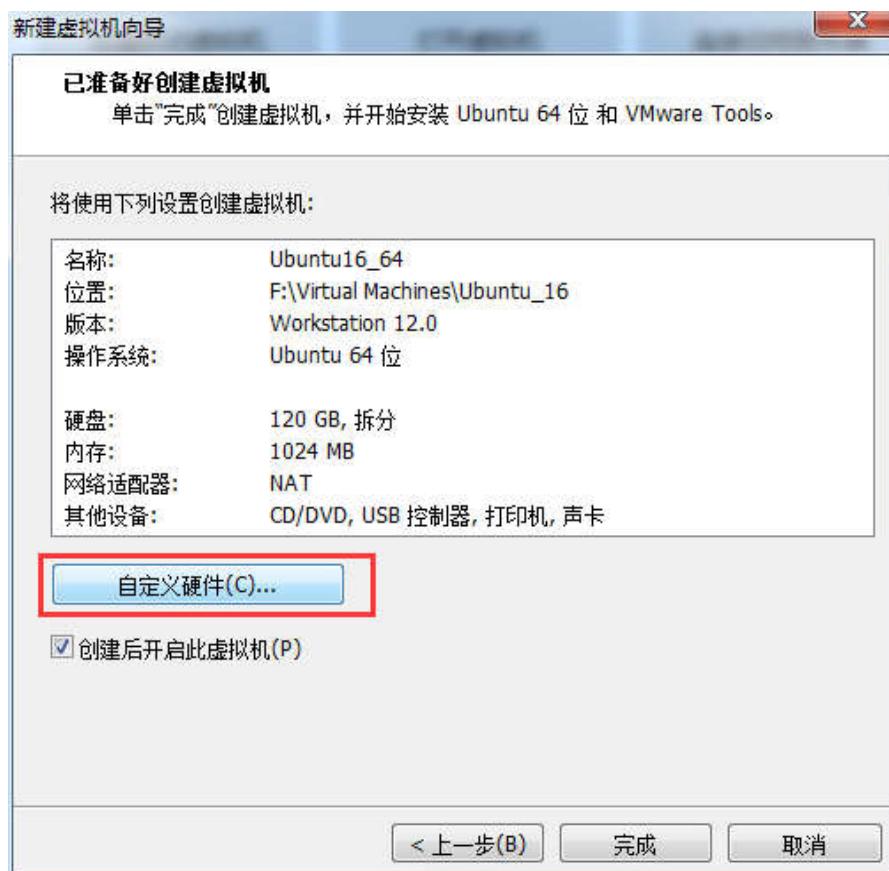
- 5) 虚拟机名称可以自己修改，安装位置需要选择安装到硬盘空间比较充足的磁盘



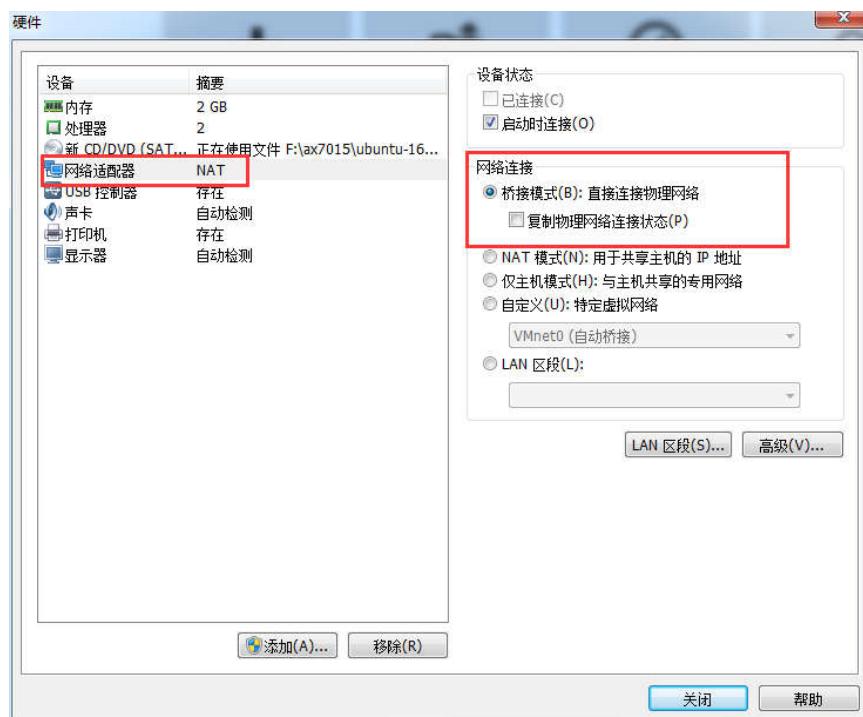
- 6) 设置最大的磁盘大小为 120G，我们需要在虚拟机里安装软件，这里预留空间大一些。用户可以根据自己的硬盘空间选择合适的空间尺寸，**建议大于等于 120G。**



7) 选择自定义硬件



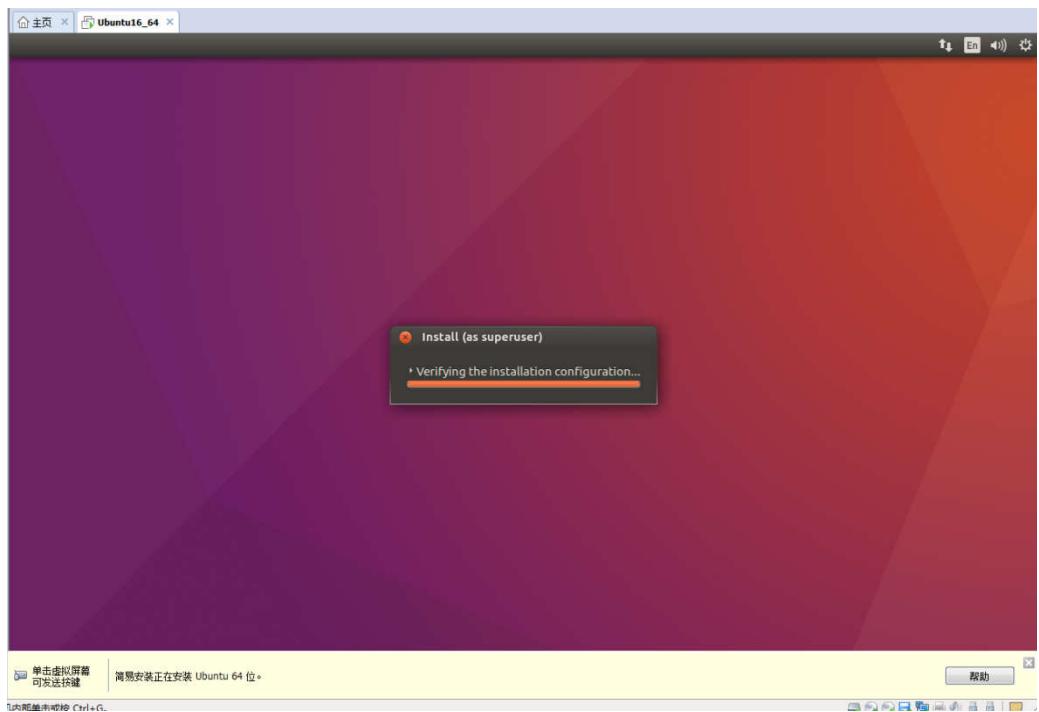
8) 可以根据修改修改内存大小和处理器核心，网络适配器选项，网络连接选择桥接模式



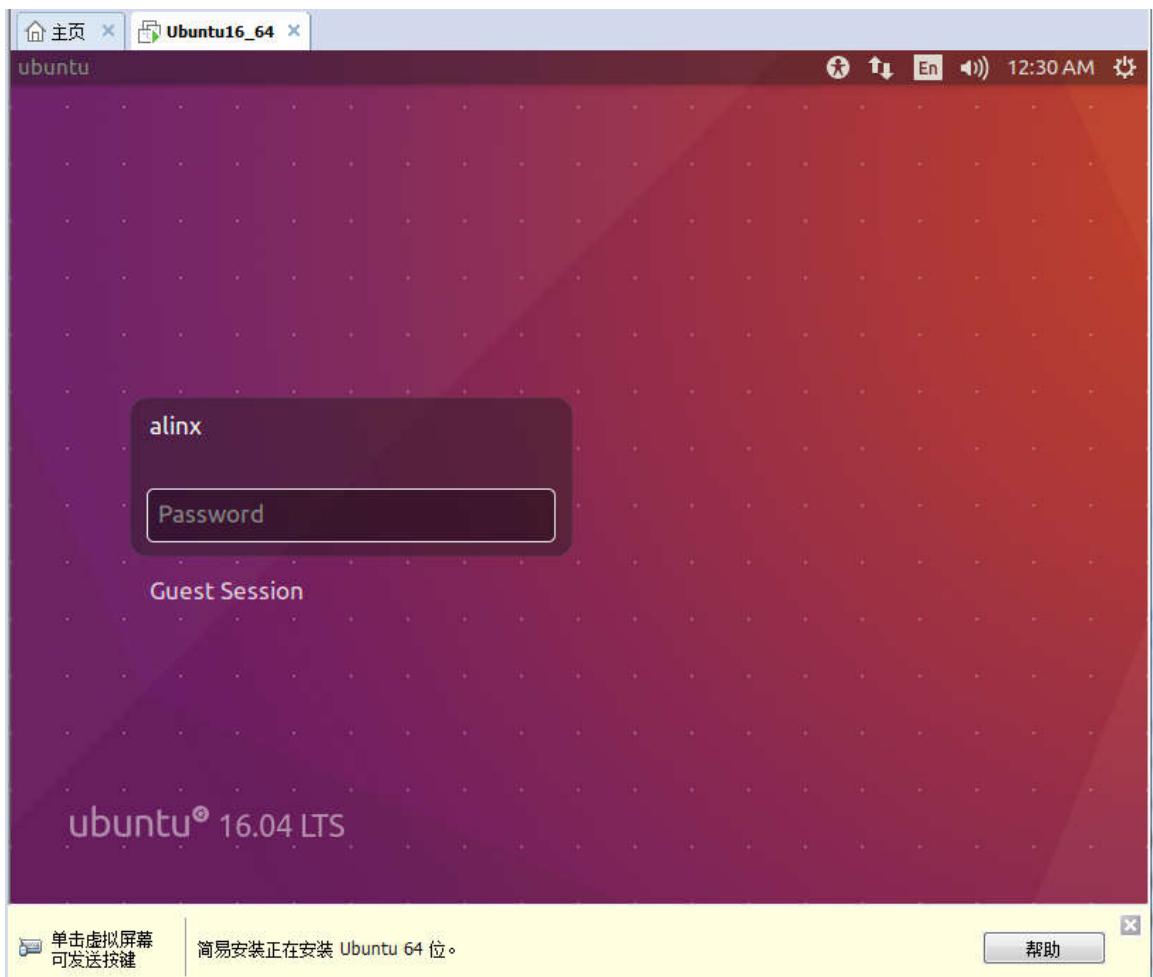
9) 点击完成就开始安装 Ubuntu 了



10) 安装过程比较慢，要等待一段时间

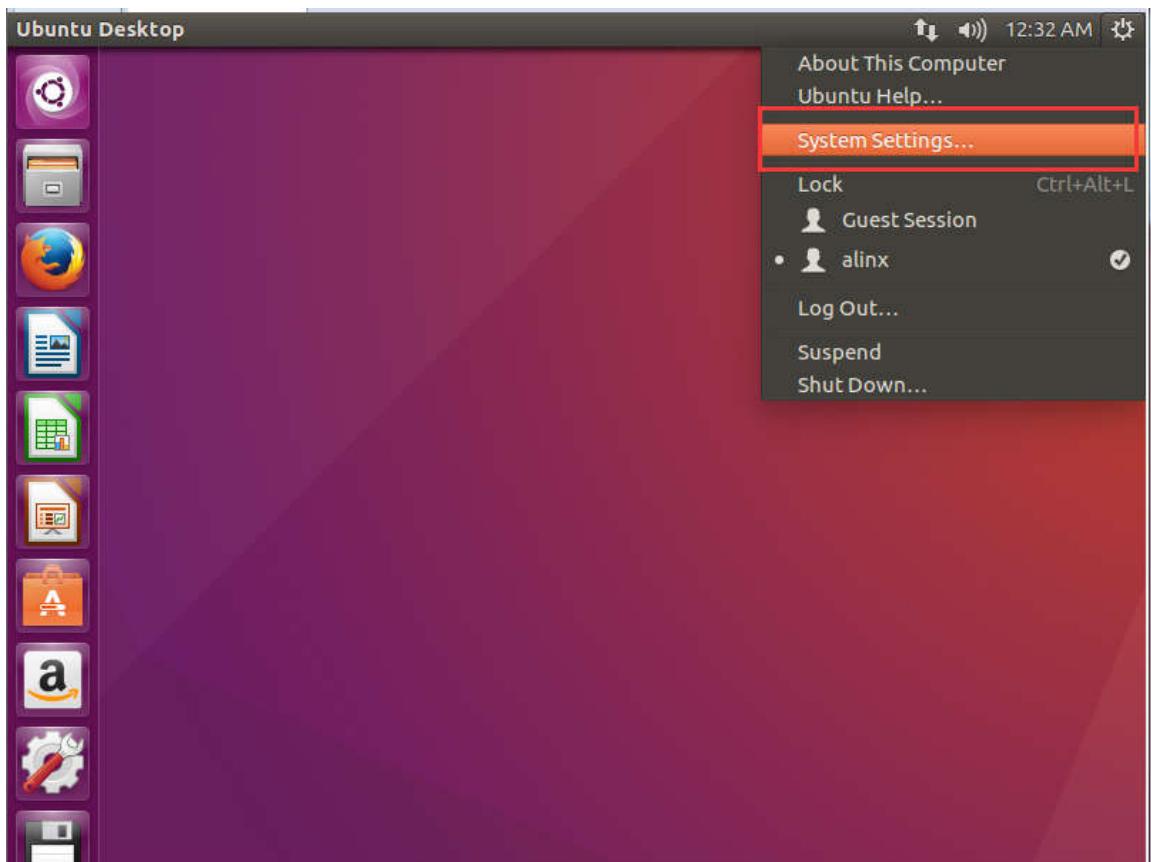


11) 安装完成以后进入系统

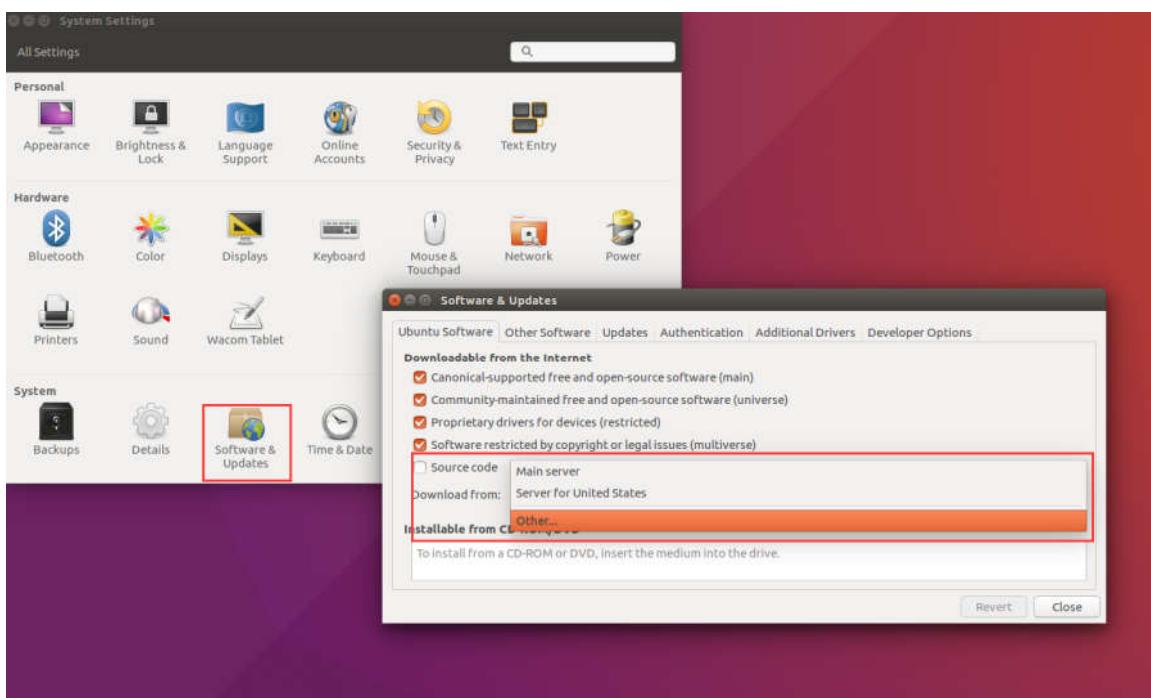


18.2.2 修改软件源服务器

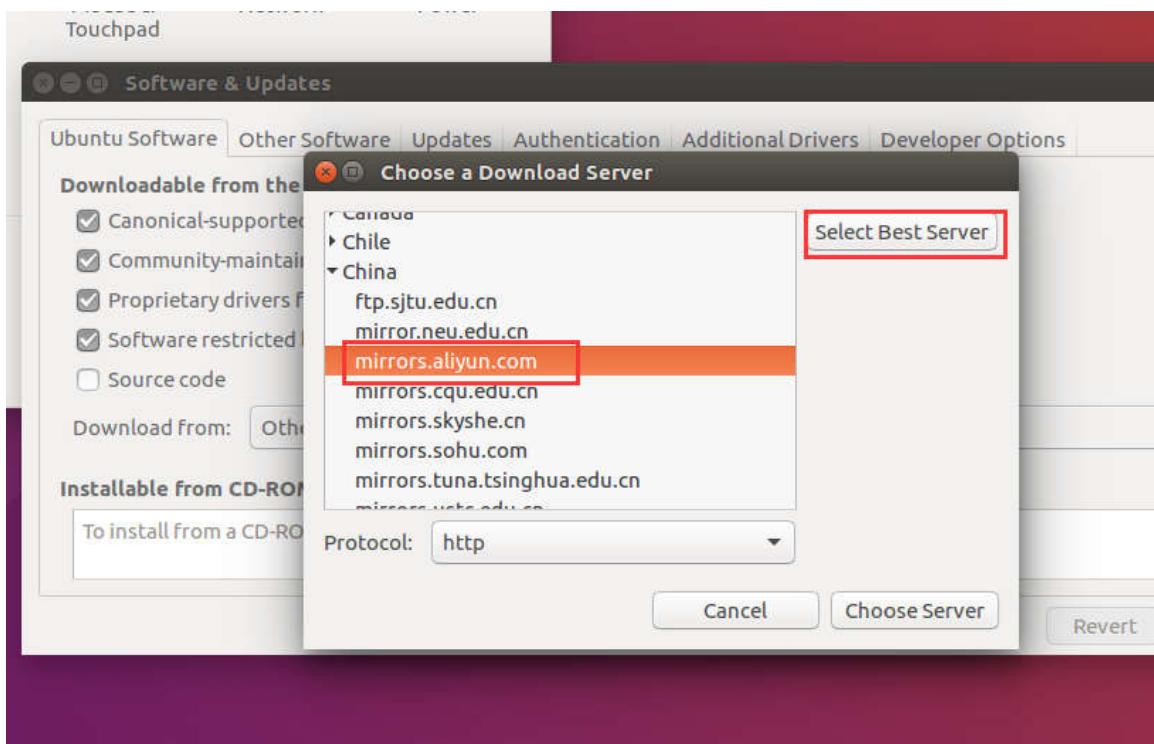
1) 为了以后安装软件方便，我们要设置一下软件源，点击系统设置



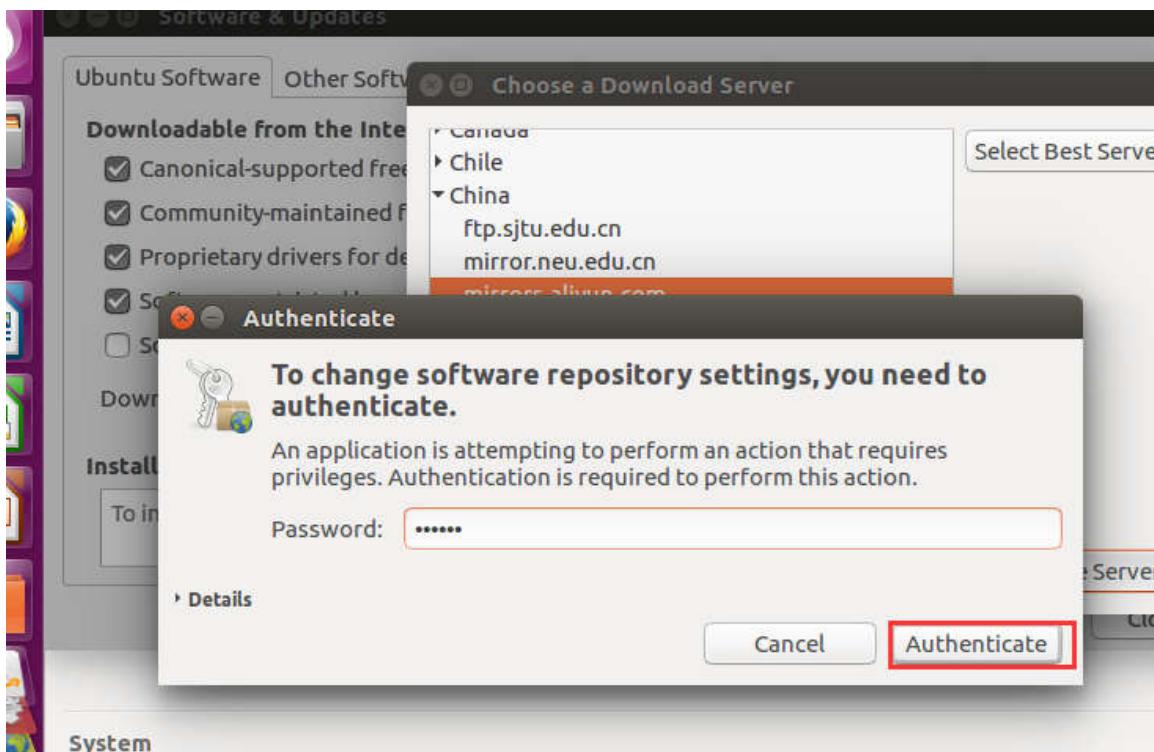
2) 在 “Software & Updates” 中选择 “Other...”



- 3) 点击 “Select Best Server” , 可以测试出一个最快的服务器 , 然后选择 “Choose Server” , 这些操作都是基于虚拟机能够连接互联网的情形。

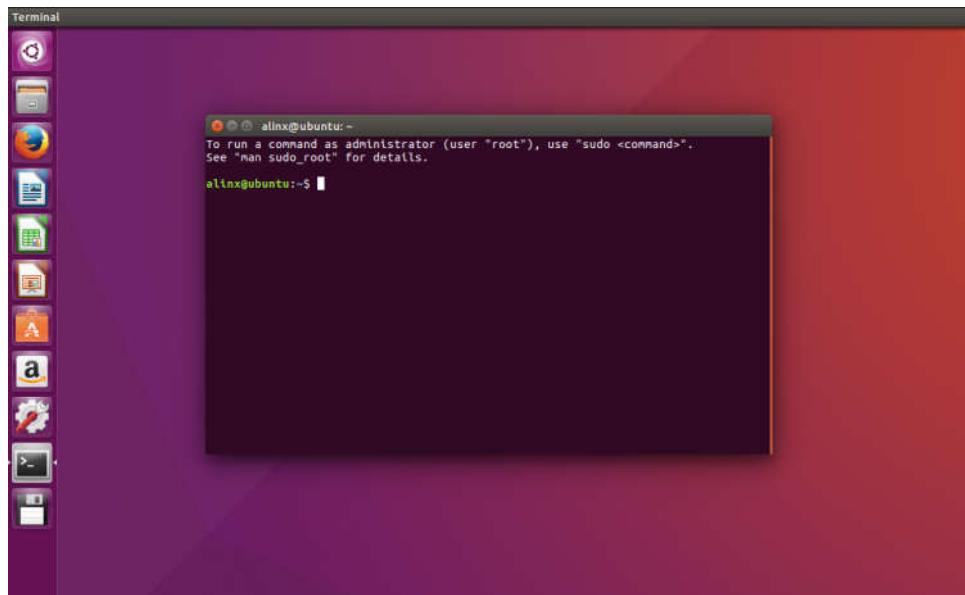


- 4) 输入密码 , 完成软件源修改



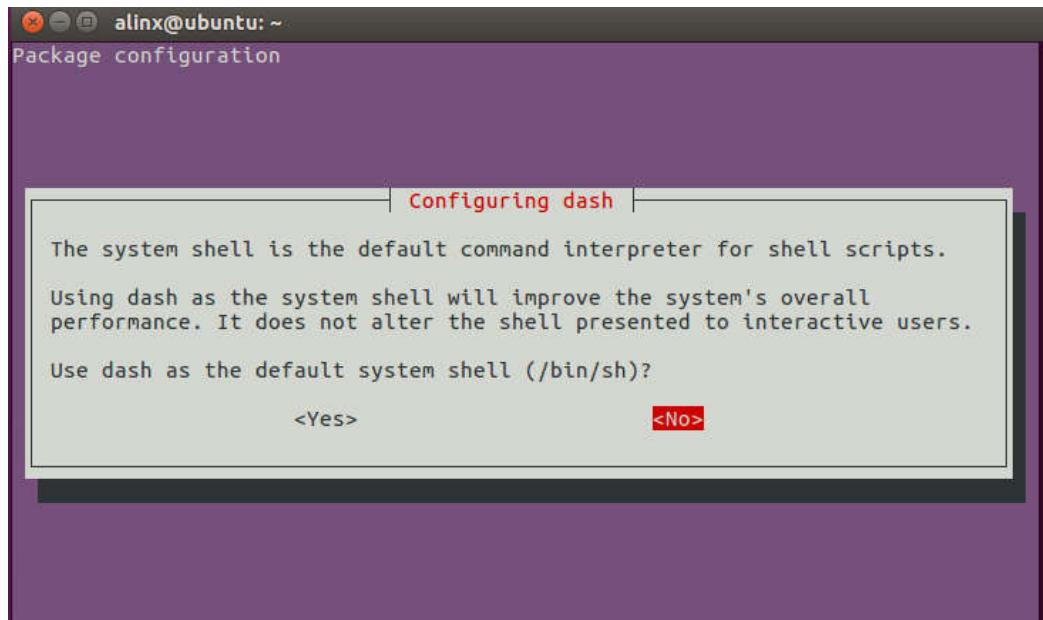
18.2.3 设置 bash 为默认 sh

1) Ctrl+Alt+T 打开终端



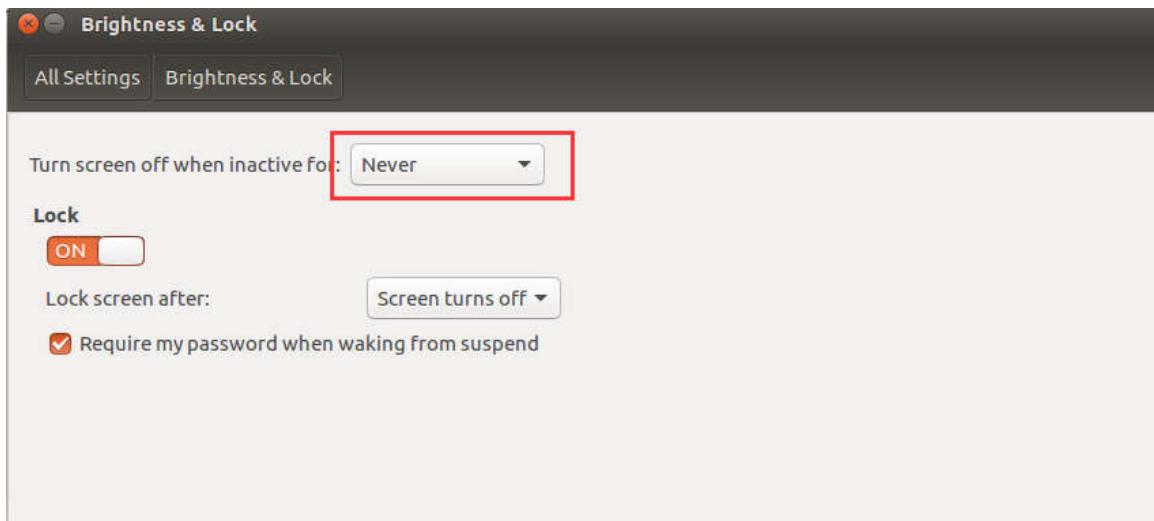
2) 输入命令，Configuring dash 选择 “No”，回车确认

```
sudo dpkg-reconfigure dash
```



18.2.4 设置屏幕锁定时间

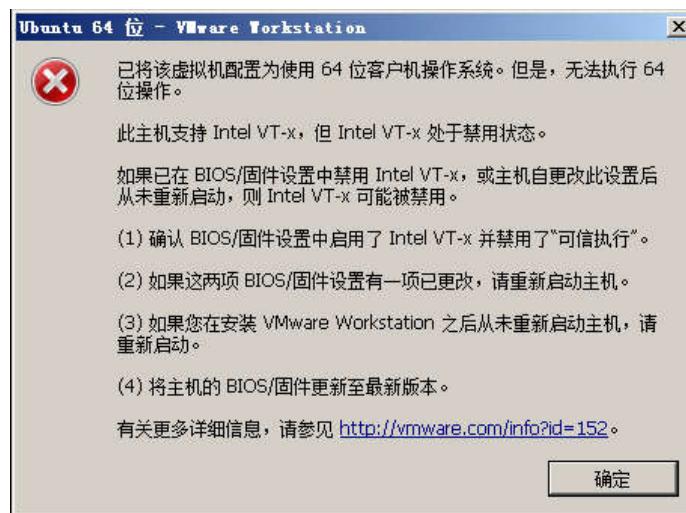
为了能复制大文件到 Ubuntu 系统，我们取消屏幕锁定



18.3 常见问题

18.3.1 虚拟机要求虚拟化支持

- 1) 如果安装 Ubuntu 弹出以下的错误信息框的话，用户需要重启电脑，进入 BIOS 里进行设置。



重启电脑后，进入到 BIOS 里，找到 Inter 虚拟化技术这一项，点击开启。不同的主板，可能名字不太一样。

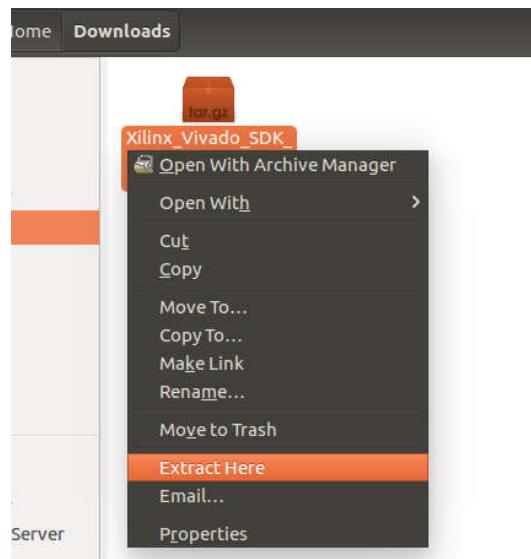


第十九章 Ubuntu 安装 Linux 版 Vivado 软件

虽然 Windows 下的 Vivado 软件可以解决大部分问题，可是偶尔我们还要使用 Linux 版本的 Vivado，特别是 SDK，我们可以交叉编译很多应用程序。

19.1 安装 Linux 版 Vivado

- 1) 复制安装文件到虚拟机 Ubuntu，解压文件

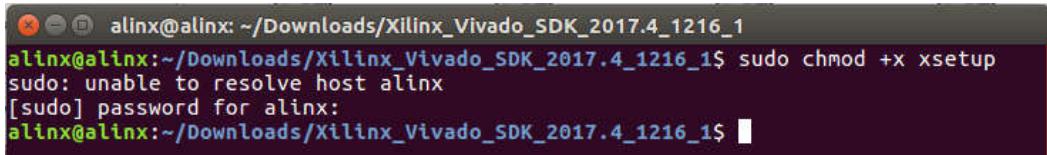


- 2) 使用终端进入解压后的

A screenshot of a terminal window with a dark background. The terminal prompt shows 'alinx@alinx: ~/Downloads/Xilinx_Vivado_SDK_2017.4_1216_1\$'. The user has entered the command 'cd ~/Downloads/Xilinx_Vivado_SDK_2017.4_1216_1' and is awaiting the results.

- 3) 运行命令

```
sudo chmod +x xsetup
```



```
alinx@alinx: ~/Downloads/Xilinx_Vivado_SDK_2017.4_1216_1
alinx@alinx:~/Downloads/Xilinx_Vivado_SDK_2017.4_1216_1$ sudo chmod +x xsetup
sudo: unable to resolve host alinx
[sudo] password for alinx:
alinx@alinx:~/Downloads/Xilinx_Vivado_SDK_2017.4_1216_1$
```

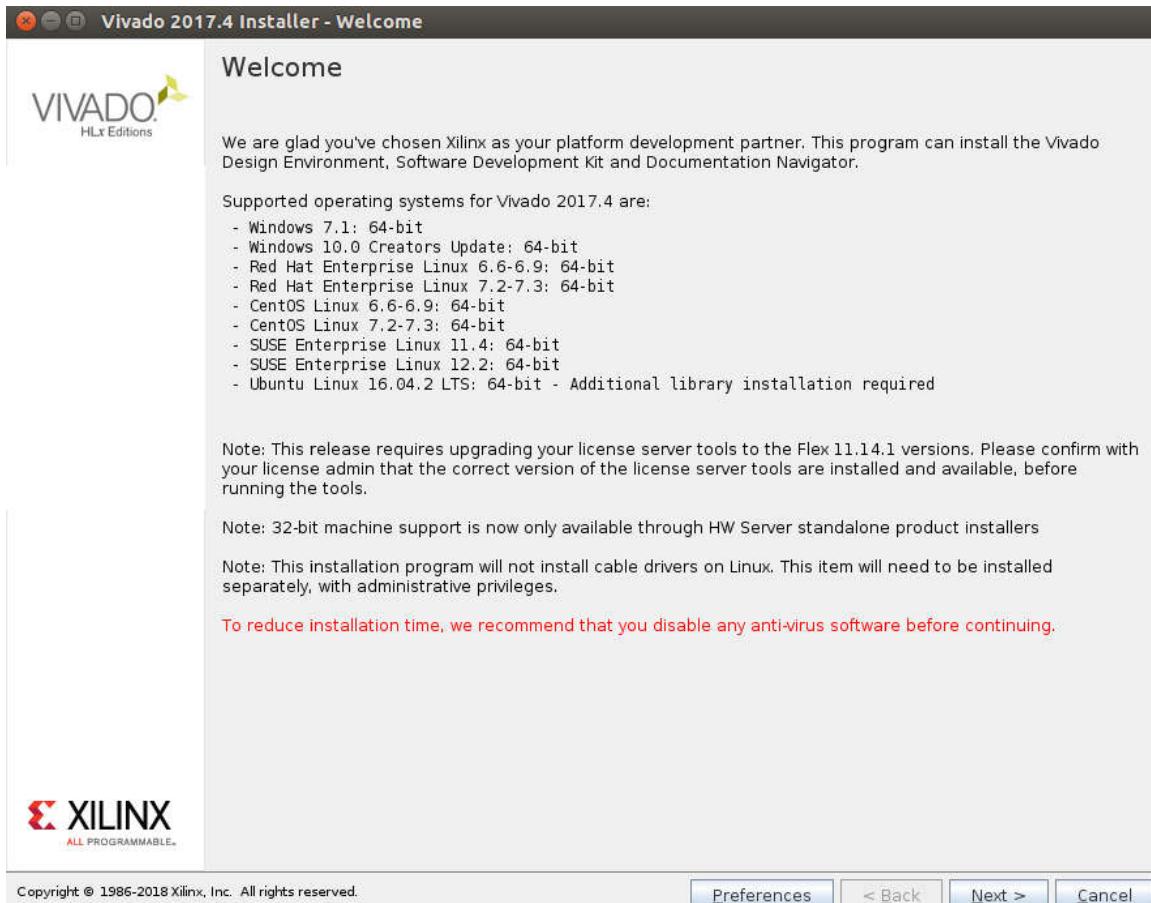
- 4) 运行命令，开始安装

```
sudo ./xsetup
```

- 5) 如果弹出这些窗口，都点击“Ignore”



- 6) 安装过程要求我们关闭杀毒软件



7) 同意所有条款

Accept License Agreements

Please read the following terms and conditions and indicate that you agree by checking the I Agree checkboxes.

Xilinx Inc. End User License Agreement

By checking "I AGREE" below, or OTHERWISE ACCESSING, DOWNLOADING, INSTALLING or USING THE SOFTWARE, YOU AGREE on behalf of licensee to be bound by the agreement, which can be viewed by [clicking here](#).

I Agree

WebTalk Terms And Conditions

By checking "I AGREE" below, I also confirm that I have read [Section 13 of the terms and conditions](#) above concerning WebTalk and have been afforded the opportunity to read the WebTalk FAQ posted at <https://www.xilinx.com/products/design-tools/webtalk.html>. I understand that I am able to disable WebTalk later if certain criteria described in Section 13(c) apply. If they don't apply, I can disable WebTalk by uninstalling the Software or using the Software on a machine not connected to the internet. If I fail to satisfy the applicable criteria or if I fail to take the applicable steps to prevent such transmission of information, I agree to allow Xilinx to collect the information described in Section 13(a) for the purposes described in Section 13(b).

I Agree

Third Party Software End User License Agreement

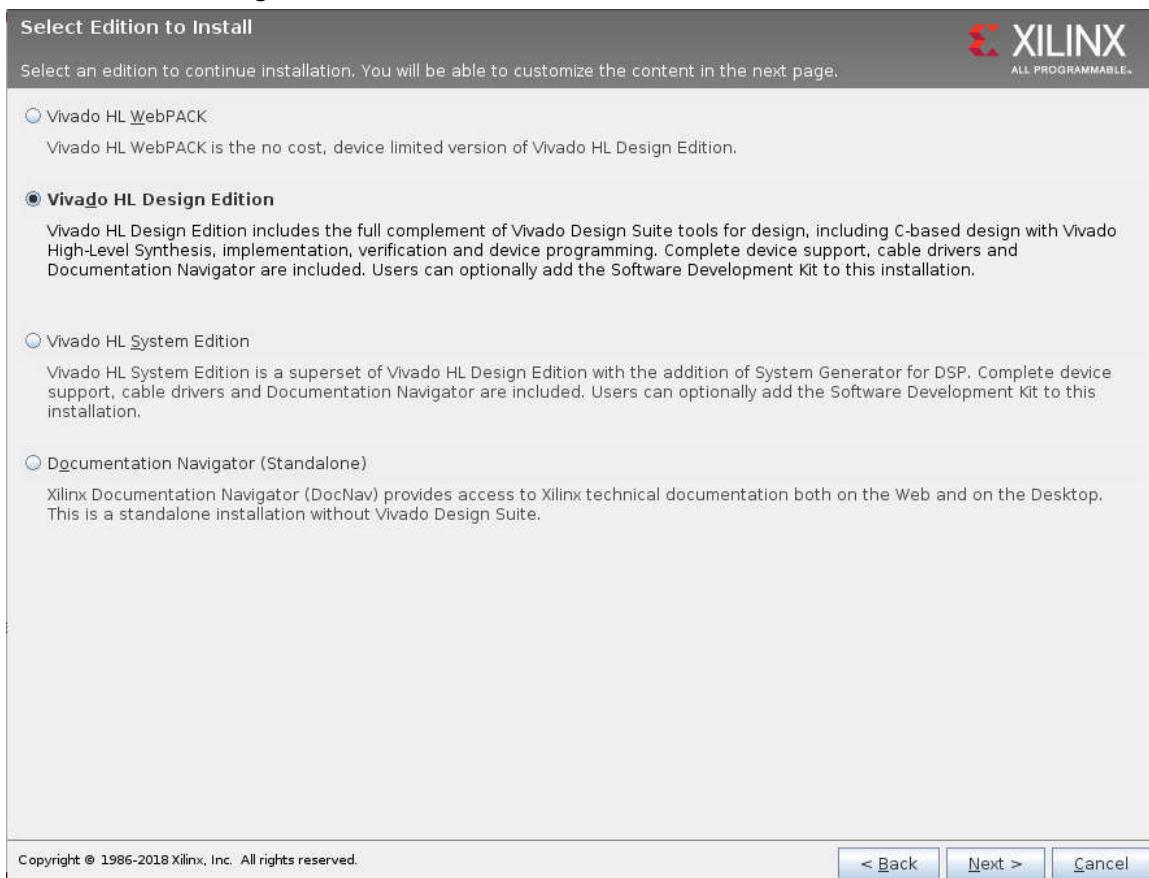
By checking "I AGREE" below, or OTHERWISE ACCESSING, DOWNLOADING, INSTALLING or USING THE SOFTWARE, YOU AGREE on behalf of licensee to be bound by the agreement, which can be viewed by [clicking here](#).

I Agree

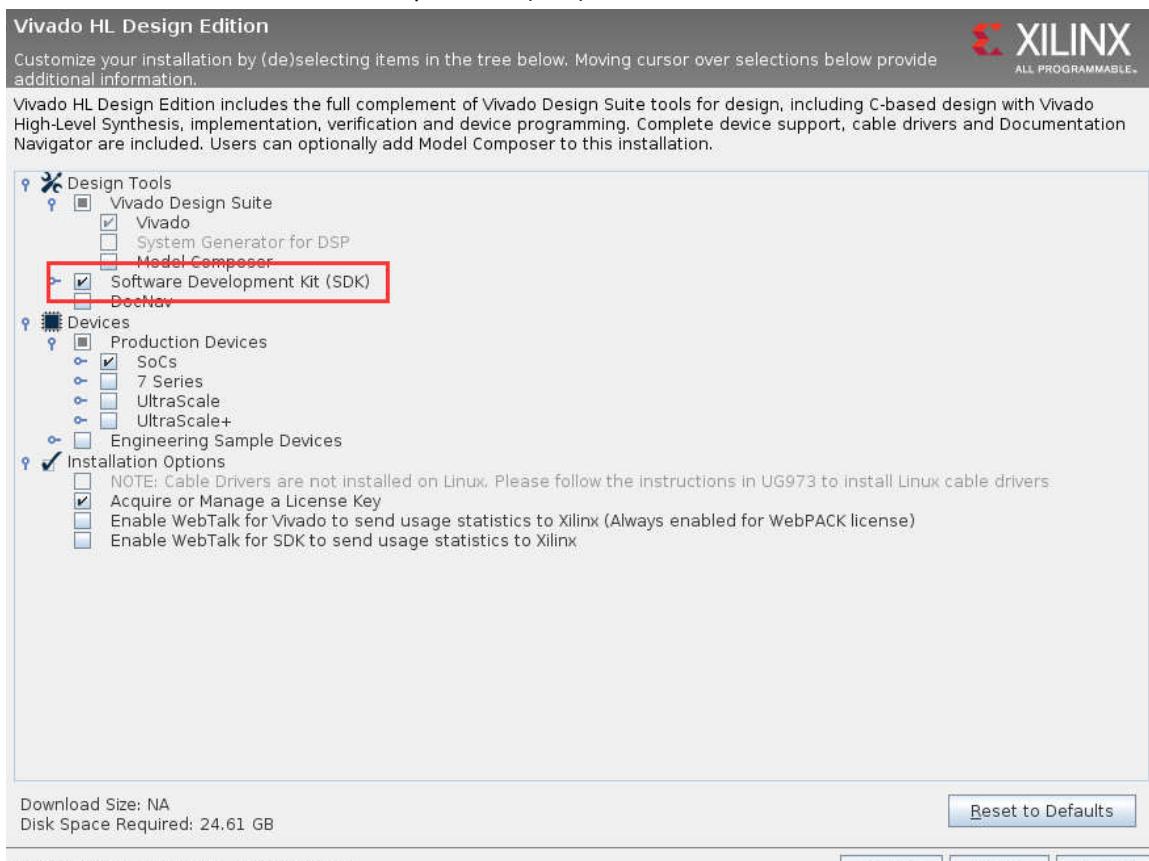
Copyright © 1986-2018 Xilinx, Inc. All rights reserved.

< Back Next > Cancel

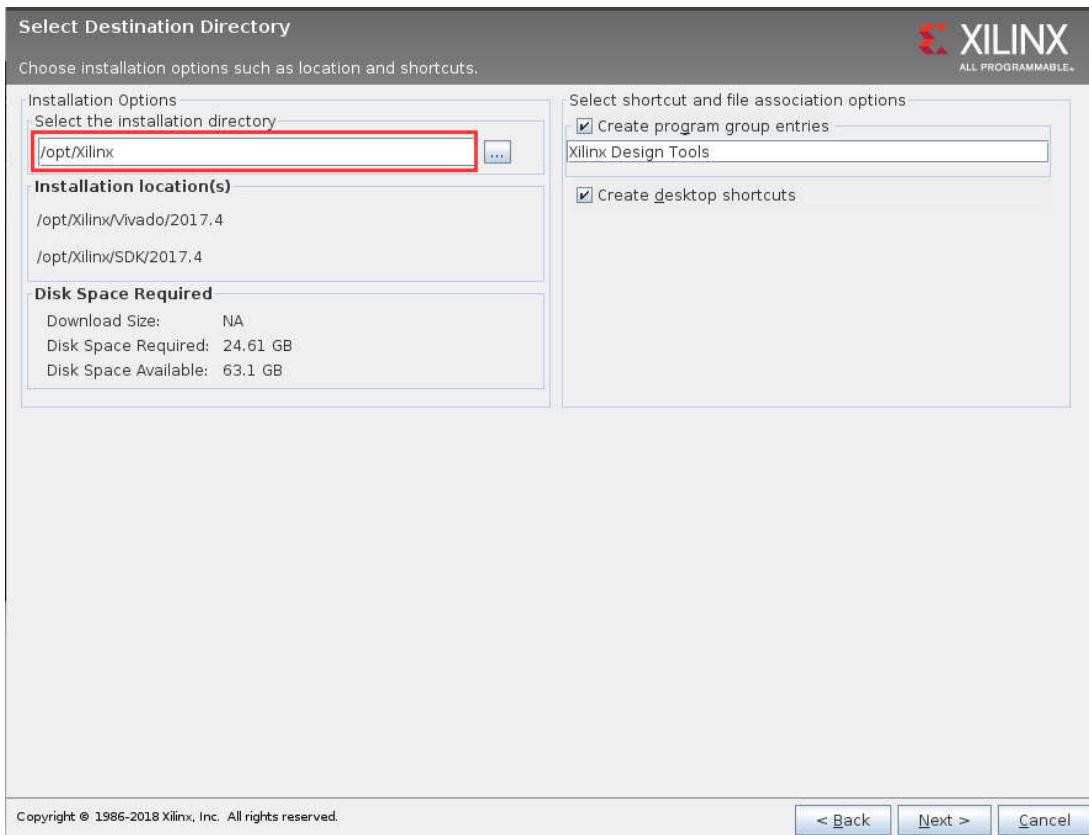
8) 选择 “Vivado HL Design Edition”



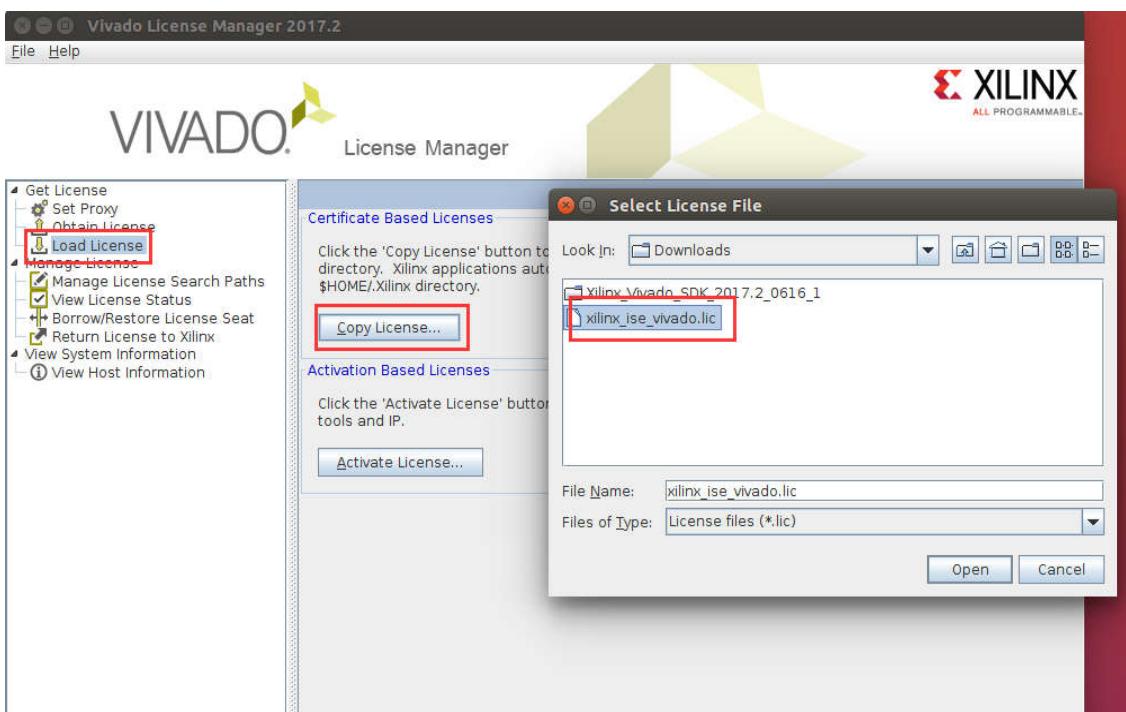
9) 这里一定要选择 “Software Development Kit(SDK)”



10) 安装路径使用默认路径



11) 点击 Copy License 安装 “lic” 文件



19.2 权限设置

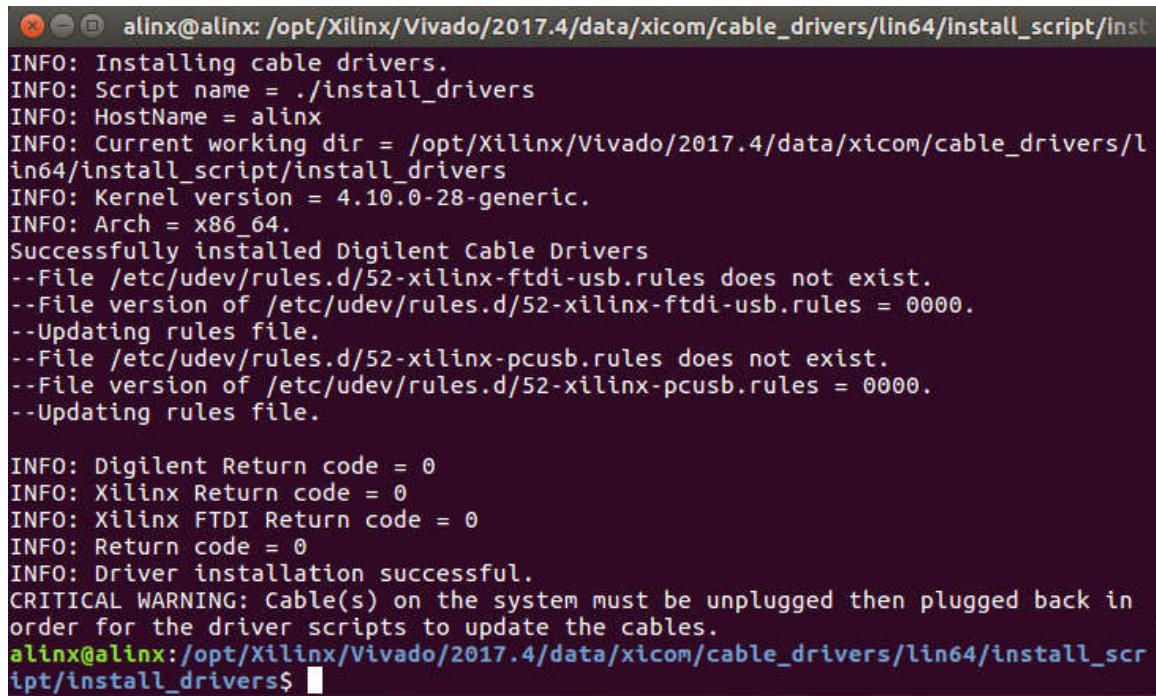
运行命令添加运行权限

```
sudo chmod 777 -R /opt/Xilinx/  
sudo chmod 777 -R ~/Xilinx/
```

19.3 安装下载器驱动

运行下列命令安装下载器驱动

```
cd /opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/install_drivers/  
sudo ./install_drivers
```



```
alinx@alinx: /opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/install_drivers$ ./install_drivers  
INFO: Installing cable drivers.  
INFO: Script name = ./install_drivers  
INFO: HostName = alinx  
INFO: Current working dir = /opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/install_drivers  
INFO: Kernel version = 4.10.0-28-generic.  
INFO: Arch = x86_64.  
Successfully installed Digilent Cable Drivers  
--File /etc/udev/rules.d/52-xilinx-ftdi-usb.rules does not exist.  
--File version of /etc/udev/rules.d/52-xilinx-ftdi-usb.rules = 0000.  
--Updating rules file.  
--File /etc/udev/rules.d/52-xilinx-pcusb.rules does not exist.  
--File version of /etc/udev/rules.d/52-xilinx-pcusb.rules = 0000.  
--Updating rules file.  
  
INFO: Digilent Return code = 0  
INFO: Xilinx Return code = 0  
INFO: Xilinx FTDI Return code = 0  
INFO: Return code = 0  
INFO: Driver installation successful.  
CRITICAL WARNING: Cable(s) on the system must be unplugged then plugged back in  
order for the driver scripts to update the cables.  
alinx@alinx:/opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/install_drivers$
```

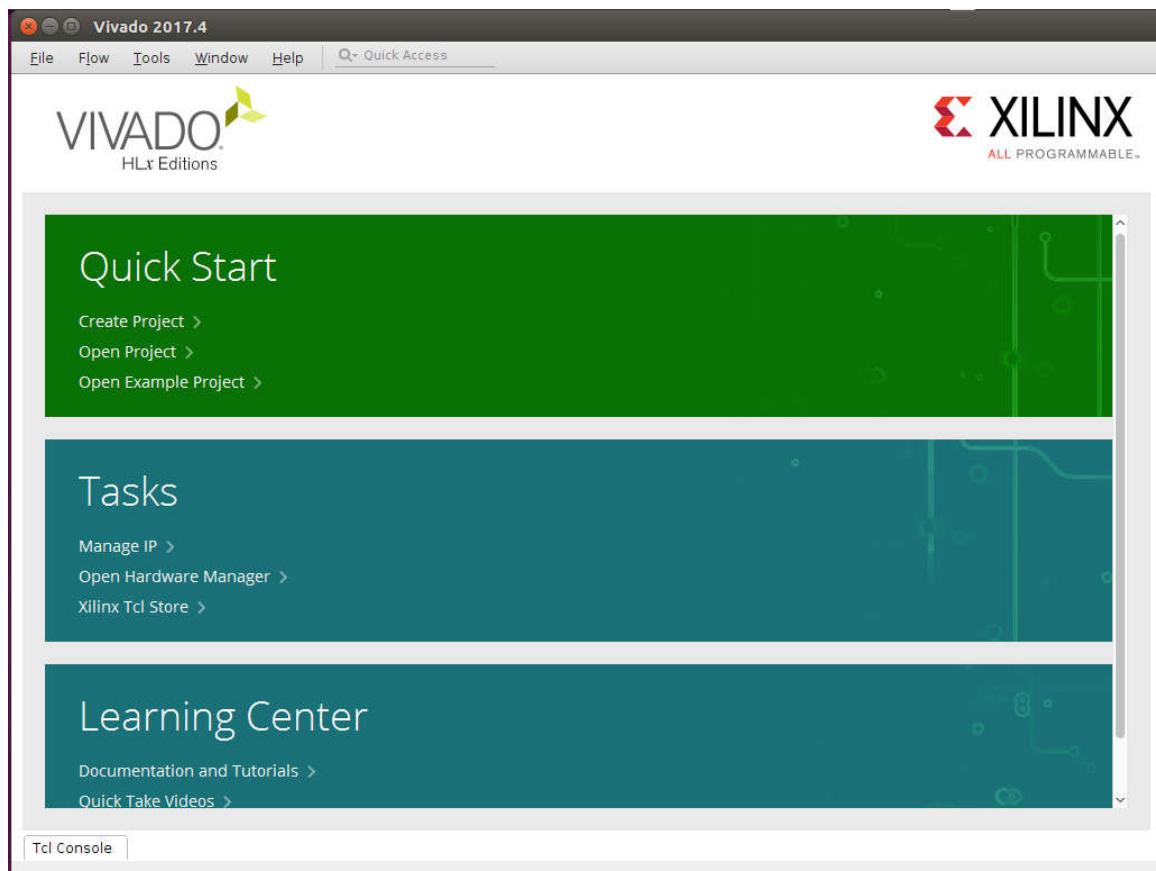
19.4 测试 Vivado

- 1) 运行下列命令，启动 Vivado

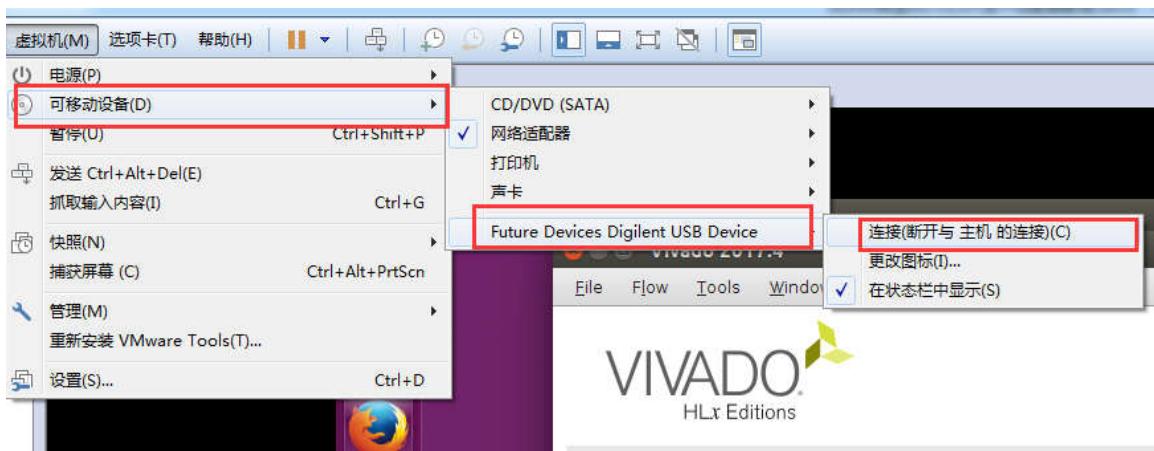
```
source /opt/Xilinx/Vivado/2017.4/settings64.sh  
vivado &
```

```
alinx@alinx: /opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/install_xicom_cable_drivers.sh
--File /etc/udev/rules.d/52-xilinx-pcusb.rules does not exist.
--File version of /etc/udev/rules.d/52-xilinx-pcusb.rules = 0000.
--Updating rules file.

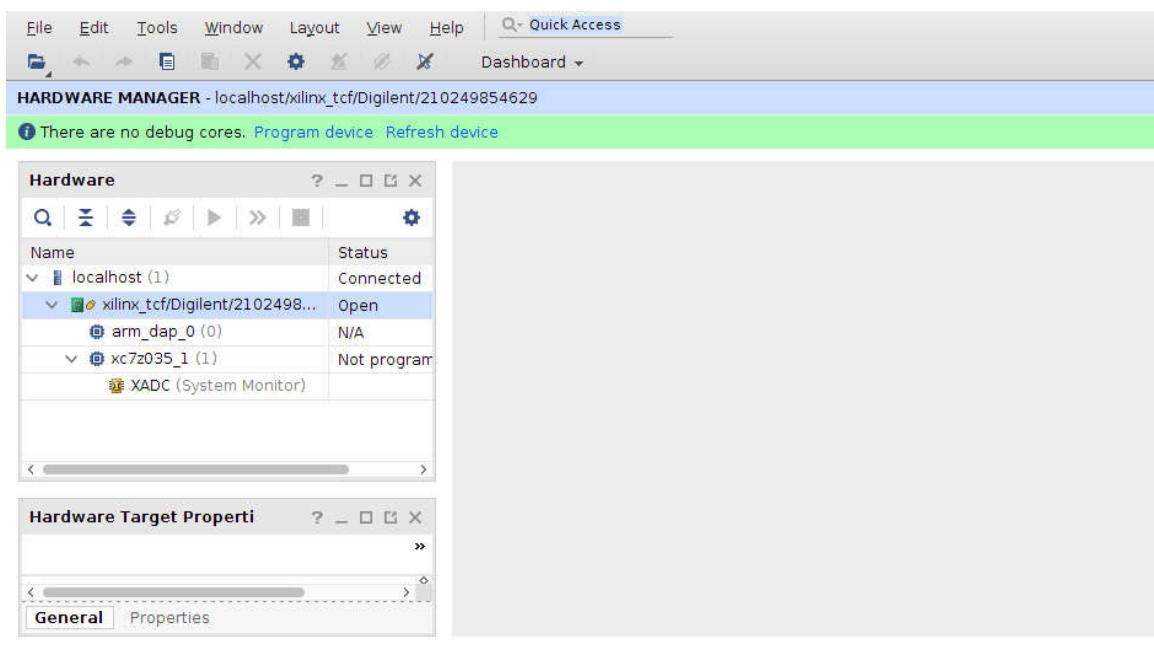
INFO: Digilent Return code = 0
INFO: Xilinx Return code = 0
INFO: Xilinx FTI Return code = 0
INFO: Return code = 0
INFO: Driver installation successful.
CRITICAL WARNING: Cable(s) on the system must be unplugged then plugged back in
order for the driver scripts to update the cables.
alinx@alinx:/opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/install_xicom_cable_drivers$ source /opt/Xilinx/Vivado/2017.4/settings64.sh
alinx@alinx:/opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/install_xicom_cable_drivers$ vivado &
[1] 7754
alinx@alinx:/opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/install_xicom_cable_drivers$ **** Vivado v2017.4 (64-bit)
**** SW Build 2086221 on Fri Dec 15 20:54:30 MST 2017
**** IP Build 2085800 on Fri Dec 15 22:25:07 MST 2017
** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.
```



2) 连接下载器到虚拟机



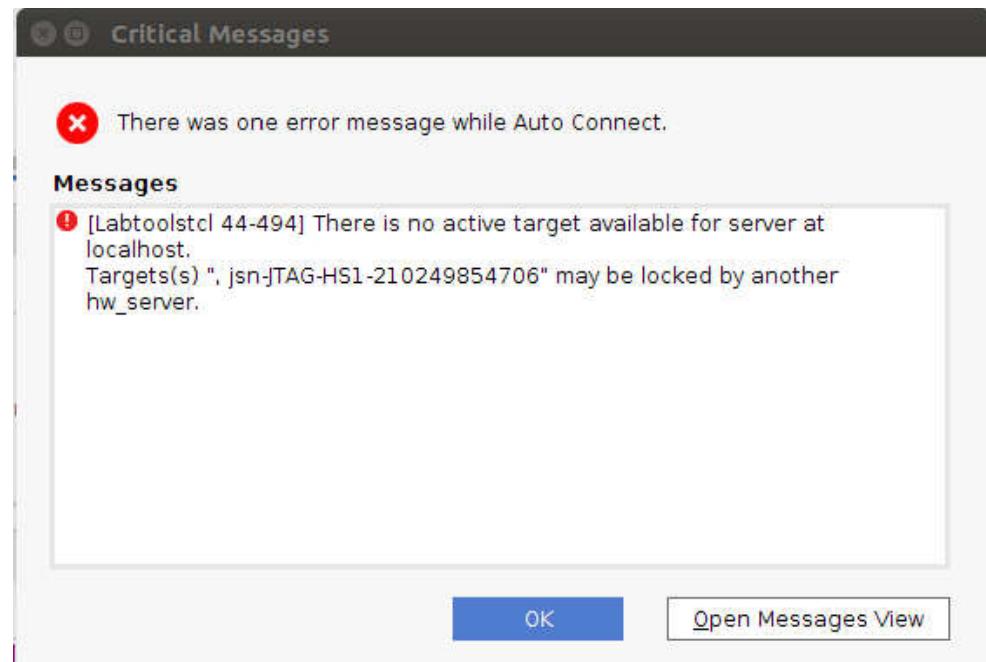
- 3) 连接开发板和下载器，使用“Open Hardware Manager”测试，正常情况下可以发现芯片，说明 Vivado 和下载器驱动安装成功



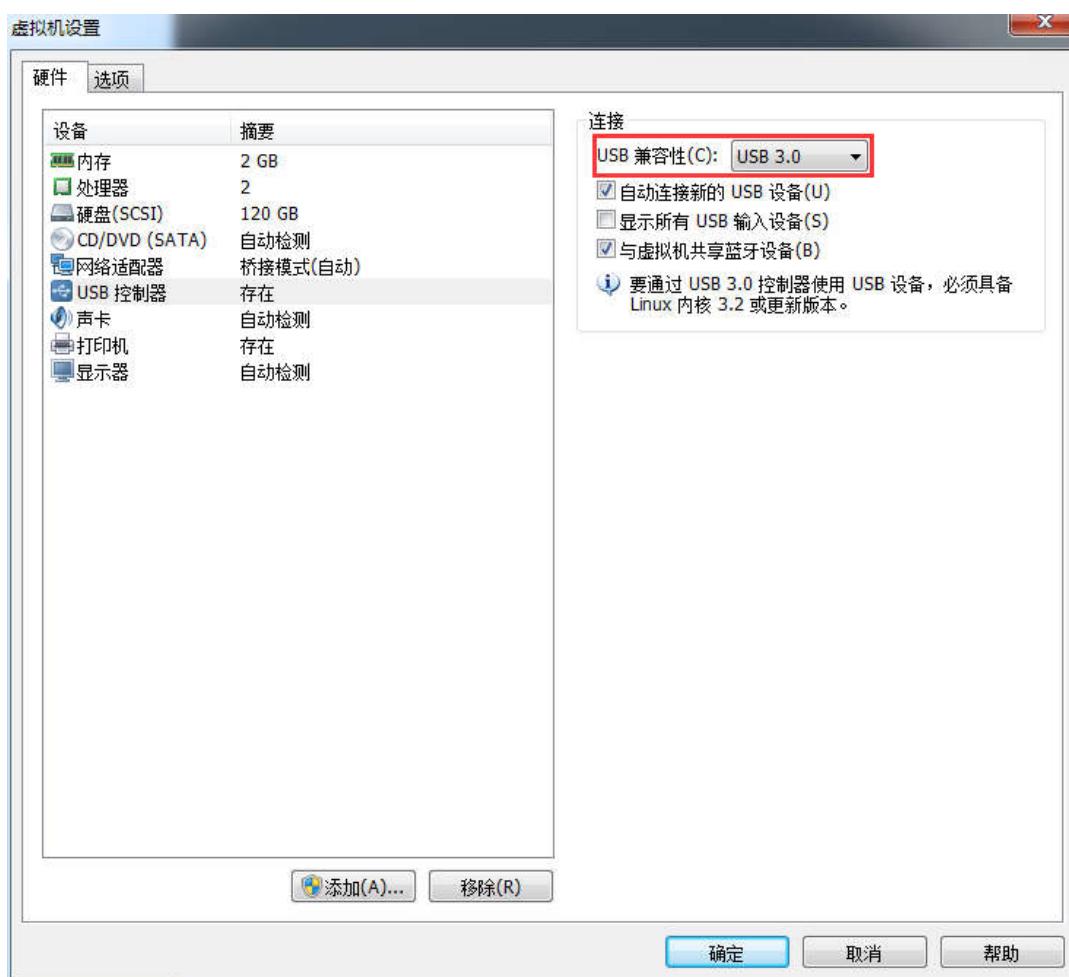
19.5 常见问题

19.5.1 Linux 下载器下载时提示被占用

- 1) 测试硬件时，能发现下载器，但是有个错误



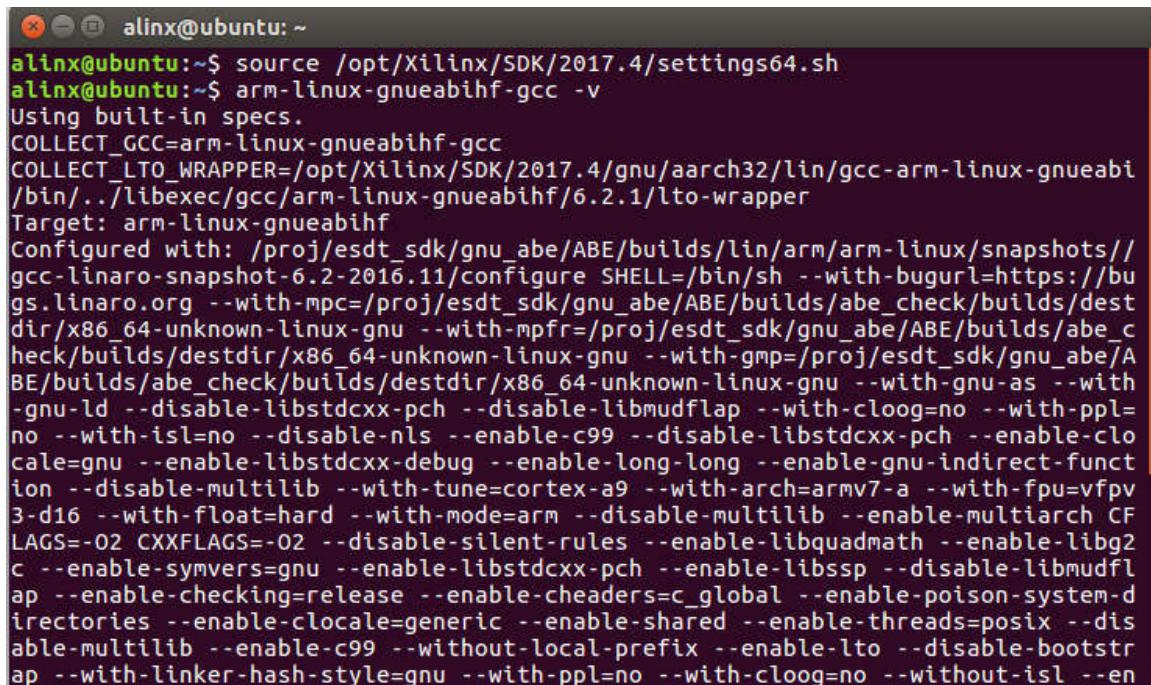
有些主板需要设置 USB 的兼容性，关闭虚拟机的 Ubuntu，设置 USB 兼容性到“USB 3.0”，再次尝试，如果还不能使用下载器，只能使用 Windows 版本下载了。



19.5.2 适合 ZYNQ 的交叉编译器

在安装 Vivado 的时候已经选择安装 SDK 了，SDK 里就包含了交叉编译器
arm-linux-gnueabihf-gcc。

```
source /opt/Xilinx/SDK/2017.4/settings64.sh  
arm-linux-gnueabihf-gcc -v
```



```
alinx@ubuntu:~$ source /opt/Xilinx/SDK/2017.4/settings64.sh  
alinx@ubuntu:~$ arm-linux-gnueabihf-gcc -v  
Using built-in specs.  
COLLECT_GCC=arm-linux-gnueabihf-gcc  
COLLECT_LTO_WRAPPER=/opt/Xilinx/SDK/2017.4/gnu/aarch32/lin/gcc-arm-linux-gnueabi  
/bin/../libexec/gcc/arm-linux-gnueabihf/6.2.1/lto-wrapper  
Target: arm-linux-gnueabihf  
Configured with: /proj/esdt_sdk/gnu_abe/ABE/builds/lin/arm/arm-linux/snapshots//  
gcc-linaro-snapshot-6.2-2016.11/configure SHELL=/bin/sh --with-bugurl=https://bu  
gs.linaro.org --with-mpc=/proj/esdt_sdk/gnu_abe/ABE/builds/abe_check/builds/dest  
dir/x86_64-unknown-linux-gnu --with-mpfr=/proj/esdt_sdk/gnu_abe/ABE/builds/abe_c  
heck/builds/destdir/x86_64-unknown-linux-gnu --with-gmp=/proj/esdt_sdk/gnu_abe/A  
BE/builds/abe_check/builds/destdir/x86_64-unknown-linux-gnu --with-gnu-as --with  
-gnu-ld --disable-libstdcxx-pch --disable-libmudflap --with-cloog=no --with-ppl=  
no --with-isl=no --disable-nls --enable-c99 --disable-libstdcxx-pch --enable-clo  
cale-gnu --enable-libstdcxx-debug --enable-long-long --enable-gnu-indirect-funct  
ion --disable-multilib --with-tune=cortex-a9 --with-arch=armv7-a --with-fpu=vfpv  
3-d16 --with-float=hard --with-mode=arm --disable-multilib --enable-multiarch CF  
LAGS=-O2 CXXFLAGS=-O2 --disable-silent-rules --enable-libquadmath --enable-libg2  
c --enable-symvers=gnu --enable-libstdcxx-pch --enable-libssp --disable-libmudfl  
ap --enable-checking=release --enable-headers=c_global --enable-poison-system-d  
irectories --enable-clocale=generic --enable-shared --enable-threads=posix --dis  
able-multilib --enable-c99 --without-local-prefix --enable-lto --disable-bootstr  
ap --with-linker-hash-style=gnu --with-ppl=no --with-cloog=no --without-isl --en
```

第二十章 Petalinux 工具安装

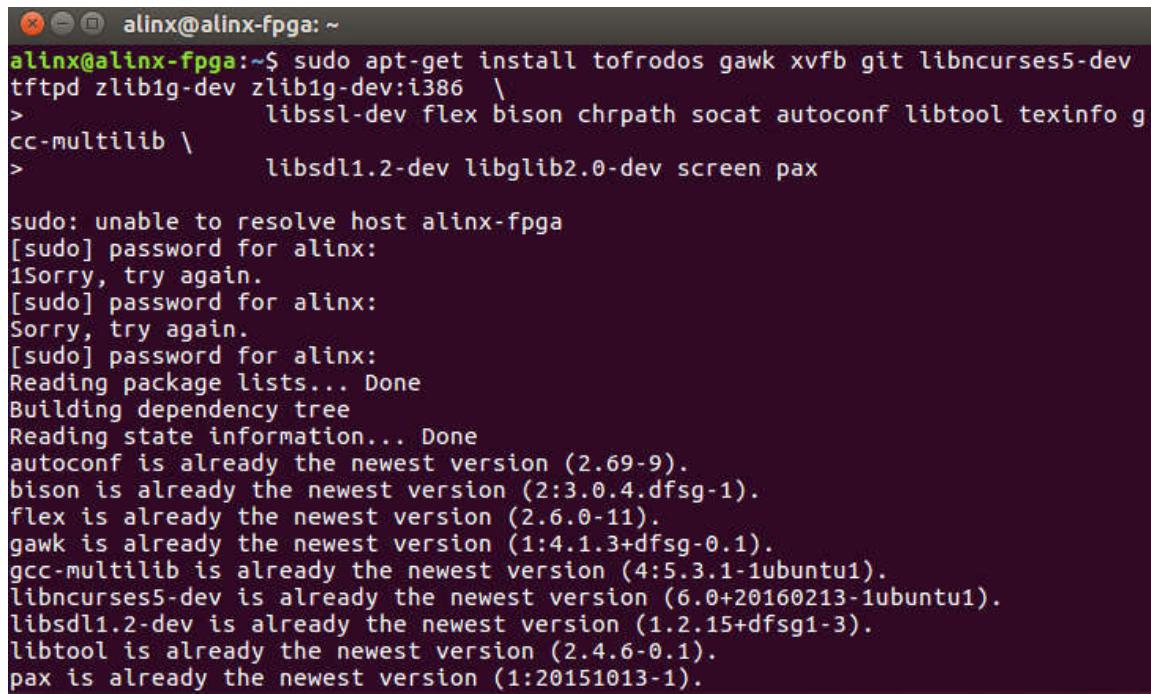
20.1 Petalinux 简介

petalinux 并不是一个特殊 Linux 内核，而是一套开发环境配置的工具，降低 uboot、内核、根文件系统的配置的工作量，可以从 Vivado 的导出硬件信息自动完成相关软件的配置。

20.2 安装必要的库

- 1) 运行下面命令安装库

```
sudo apt-get install tofrodos gawk xvfb git libncurses5-dev tftpd zlib1g-dev zlib1g-dev:i386 \
libssl-dev flex bison chrpath socat autoconf libtool texinfo gcc-multilib \
libsdl1.2-dev libglib2.0-dev screen pax
```



```
alinx@alinx-fpga: ~
alinx@alinx-fpga: ~$ sudo apt-get install tofrodos gawk xvfb git libncurses5-dev
tftpd zlib1g-dev zlib1g-dev:i386 \
> libssl-dev flex bison chrpath socat autoconf libtool texinfo g
cc-multilib \
> libsdl1.2-dev libglib2.0-dev screen pax

sudo: unable to resolve host alinx-fpga
[sudo] password for alinx:
1Sorry, try again.
[sudo] password for alinx:
Sorry, try again.
[sudo] password for alinx:
Reading package lists... Done
Building dependency tree
Reading state information... Done
autoconf is already the newest version (2.69-9).
bison is already the newest version (2:3.0.4.dfsg-1).
flex is already the newest version (2.6.0-11).
gawk is already the newest version (1:4.1.3+dfsg-0.1).
gcc-multilib is already the newest version (4:5.3.1-1ubuntu1).
libncurses5-dev is already the newest version (6.0+20160213-1ubuntu1).
libsdl1.2-dev is already the newest version (1.2.15+dfsg1-3).
libtool is already the newest version (2.4.6-0.1).
pax is already the newest version (1:20151013-1).
```

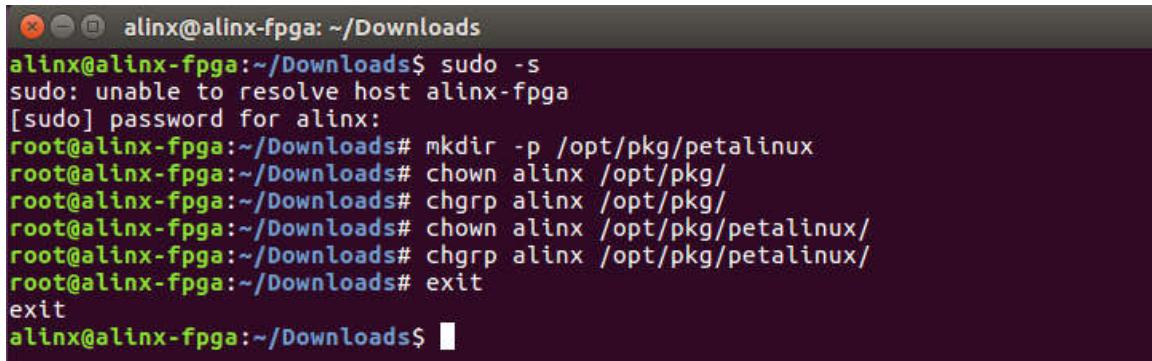
- 2) 配置 tftp server，如果不需要从 TFTP 启动，这一步可选

```
sudo -s
apt-get install tftpd-hpa
chmod a+w /var/lib/tftpboot/
reboot
```

20.3 安装 Petalinux

- 1) 运行一下命令做安装准备

```
sudo -s  
mkdir -p /opt/pkg/petalinux  
chown <your_user_name> /opt/pkg/  
chgrp <your_user_name> /opt/pkg/  
chgrp <your_user_name> /opt/pkg/petalinux/  
chown <your_user_name> /opt/pkg/petalinux/  
exit
```



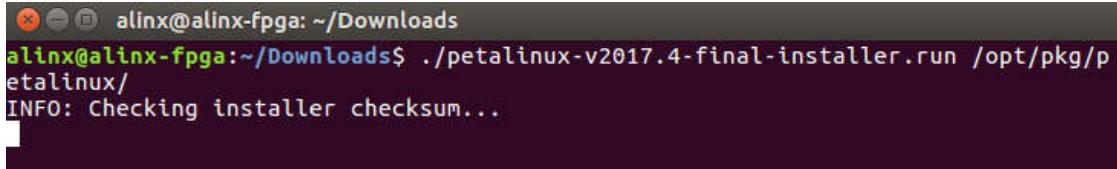
```
alinx@alinx-fpga: ~/Downloads$ sudo -s  
sudo: unable to resolve host alinx-fpga  
[sudo] password for alinx:  
root@alinx-fpga:~/Downloads# mkdir -p /opt/pkg/petalinux  
root@alinx-fpga:~/Downloads# chown alinx /opt/pkg/  
root@alinx-fpga:~/Downloads# chgrp alinx /opt/pkg/  
root@alinx-fpga:~/Downloads# chown alinx /opt/pkg/petalinux/  
root@alinx-fpga:~/Downloads# chgrp alinx /opt/pkg/petalinux/  
root@alinx-fpga:~/Downloads# exit  
exit  
alinx@alinx-fpga:~/Downloads$
```

- 2) 给安装文件添加运行权限

```
sudo chmod +x petalinux-v2017.4-final-installer.run
```

- 3) 开始安装

```
./petalinux-v2017.4-final-installer.run /opt/pkg/petalinux/
```



```
alinx@alinx-fpga: ~/Downloads$ ./petalinux-v2017.4-final-installer.run /opt/pkg/petalinux/  
INFO: Checking installer checksum...
```

- 4) 按回车查看协议内容

```
linux/
INFO: Checking installer checksum...
INFO: Extracting PetaLinux installer...

LICENSE AGREEMENTS

PetaLinux SDK contains software from a number of sources. Please review
the following licenses and indicate your acceptance of each to continue.

You do not have to accept the licenses, however if you do not then you may
not use PetaLinux SDK.

Use PgUp/PgDn to navigate the license viewer, and press 'q' to close
Press Enter to display the license agreements.
```

5) 按 q 退出协议内容

```
XILINX, INC.
END USER LICENSE AGREEMENT FOR PETALINUX TOOLS

CAREFULLY READ THIS END USER LICENSE AGREEMENT FOR PETALINUX TOOLS ("AGREEMENT")
. BY CLICKING THE "ACCEPT" OR "AGREE" BUTTON, ENTERING <93>YES<94> OR <93>Y<94>
TO ACCEPT THIS AGREEMENT, OR OTHERWISE ACCESSING, DOWNLOADING, INSTALLING OR USING THE SOFTWARE, YOU AGREE ON BEHALF OF LICENSEE TO BE BOUND BY THIS AGREEMENT.

IF LICENSEE DOES NOT AGREE TO ALL OF THE TERMS AND CONDITIONS OF THIS AGREEMENT,
DO NOT CLICK THE "ACCEPT" OR "AGREE" BUTTON, ENTER <93>YES<94> OR <93>Y<94>, OR
ACCESS, DOWNLOAD, INSTALL OR USE THE SOFTWARE.

1. Definitions

"Bitstream" means a machine-executable, binary form of a core used to program a Xilinx Device.

"Licensee" means the individual, corporation or other legal entity who has downloaded and installed the Software.

>User" means a specific human being who is identified by Licensee as a person who is authorized to use the applicable Software on behalf of Licensee. In cases
```

/tmp/tmp.Wt4jhy0kpU./etc/license/Petalinux_EULA.txt

6) 按 y 同意协议内容

```
linux/
INFO: Checking installer checksum...
INFO: Extracting PetaLinux installer...

LICENSE AGREEMENTS

PetaLinux SDK contains software from a number of sources. Please review
the following licenses and indicate your acceptance of each to continue.

You do not have to accept the licenses, however if you do not then you may
not use PetaLinux SDK.

Use PgUp/PgDn to navigate the license viewer, and press 'q' to close
Press Enter to display the license agreements
Do you accept Xilinx End User License Agreement? [y/N] > ■
```

7) 在安装过程中会弹出 License , 按 “q” 退出 , 然后按 “y” 同意。

WebTalk Terms and Conditions

By indicating I accept this WebTalk notice, I also confirm that I have read Section 13 of the terms and conditions above concerning WebTalk and have been afforded the opportunity to read the WebTalk FAQ posted at <http://www.xilinx.com/webtalk>. I understand that I am able to disable WebTalk later if certain criteria described in Section 13(c) apply. If they don't apply, I can disable WebTalk by uninstalling the Software or using the Software on a machine not connected to the internet. If I fail to satisfy the applicable criteria or if I fail to take the applicable steps to prevent such transmission of information, I agree to allow Xilinx to collect the information described in Section 13(a) for the purposes described in Section 13(b).

```
/tmp/tmp.Wt4jhy0kpU./etc/license/WebTalk_notice.txt (END)
```

```
INFO: Checking installer checksum...
INFO: Extracting PetaLinux installer...
```

LICENSE AGREEMENTS

PetaLinux SDK contains software from a number of sources. Please review the following licenses and indicate your acceptance of each to continue.

You do not have to accept the licenses, however if you do not then you may not use PetaLinux SDK.

Use PgUp/PgDn to navigate the license viewer, and press 'q' to close

```
Press Enter to display the license agreementsq
Do you accept Xilinx End User License Agreement? [y/N] > y
Do you accept Webtalk Terms and Conditions? [y/N] > y
Do you accept Third Party End User License Agreement? [y/N] > y
INFO: Checking installation environment requirements...
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
INFO: Installing PetaLinux...
■
```

第二十一章 NFS 服务软件安装

NFS (Network FileSystem , 网络文件系统) 是由 SUN 公司发展，并于 1984 年推出的技术，用于在不同机器，不同操作系统之间通过网络互相分享各自的文件。NFS 设计之初就是为了在不同的系统间使用，所以它的通讯协议设计与主机及操作系统无关。

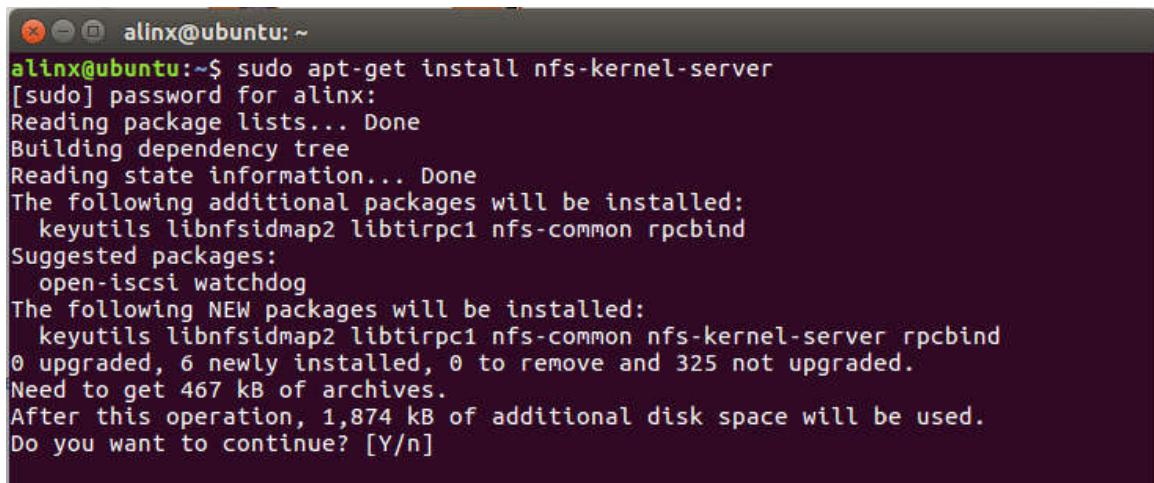
NFS 分服务器和客户机，当使用远端文件时只要用 mount 命令就可把远端 NFS 服务器上的文件系统挂载在本地文件系统之下，操作远程文件与操作本地文件没有不同。NFS 服务器所共享文件或目录记录在 /etc(exports) 文件中。

嵌入式 Linux 开发中，会经常使用 NFS，目标系统通常作为 NFS 客户机使用，Linux 主机作为 NFS 服务器。在目标系统上通过 NFS，将服务器的 NFS 共享目录挂载到本地，可以直接运行服务器上的文件。在调试系统驱动模块以及应用程序，NFS 都是十分必要的，并且 Linux 还支持 NFS 根文件系统，能直接从远程 NFS root 启动系统，这对嵌入式 Linux 根文件系统裁剪和集成也是十分有必要的。

21.1 安装 NFS 服务

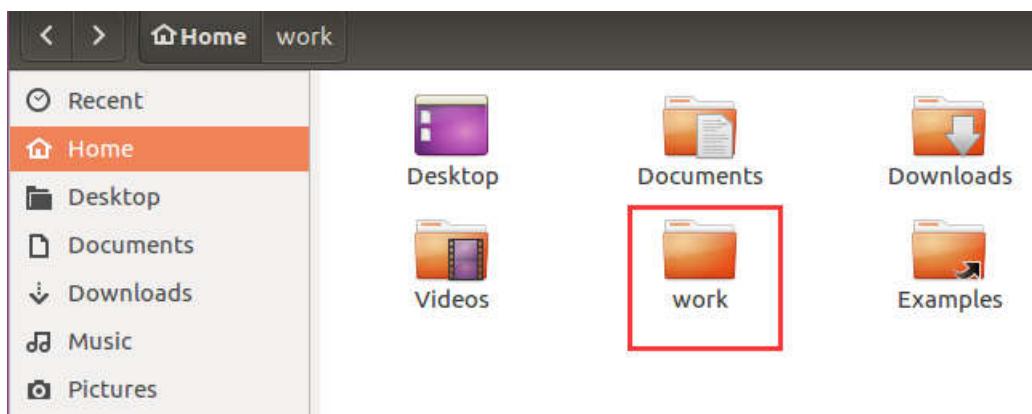
- 1) 通过下面的命令安装 NFS 服务器

```
sudo apt-get install nfs-kernel-server
```



```
alinx@ubuntu:~$ sudo apt-get install nfs-kernel-server
[sudo] password for alinx:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  keyutils libnfsidmap2 libtirpc1 nfs-common rpcbind
Suggested packages:
  open-iscsi watchdog
The following NEW packages will be installed:
  keyutils libnfsidmap2 libtirpc1 nfs-common nfs-kernel-server rpcbind
0 upgraded, 6 newly installed, 0 to remove and 325 not upgraded.
Need to get 467 kB of archives.
After this operation, 1,874 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

- 2) 新建一个 work 目录做为 NFS 的一个工作目录，以后我们可以把交叉编译的程序放在这个目录里，开发板可以很方便共享到这个目录里的文件。



- 3) 使用下面命令编辑/etc/exports 文件，配置 NFS 服务路径

```
sudo gedit /etc/exports
```

```
alinx@ubuntu:~$ sudo gedit /etc/exports
[sudo] password for alinx:
(gedit:60516): IBUS-WARNING **: The owner of /home/alinx/.config/ibus/bus is not
root!
```

- 4) 在尾部添加/home/alinx/work *(rw,sync,no_root_squash,no_subtree_check) , 配置 /home/alinx/work 目录为 NFS 的一个工作目录。

```
Open exports /etc
# /etc/exports: the access control list for filesystems which may be exported
#           to NFS clients.  See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes      hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)
#
/home/alinx/work *(rw,sync,no_root_squash,no_subtree_check)
```

- 5) 执行下面命令重启 rpcbind 服务。nfs 是一个 RPC 程序，使用它前，需要映射好端口，通过 rpcbind 设定

```
sudo /etc/init.d/rpcbind restart
```

- 6) 执行下面命令重启 nfs 服务

```
sudo /etc/init.d/nfs-kernel-server restart
```

21.2 测试 NFS

- 1) 通过下面命令挂载 NFS，在本机将 NFS 工作路径挂载在/mnt 目录

```
mount -t nfs 127.0.0.1:/home/alinx/work /mnt
```

- 2) 进入/mnt,新建一个 test 目录测试一下，可以在/home/alinx/work 目录同步看到 test 文件夹

```
cd /mnt  
mkdir test
```

第二十二章 使用 Petalinux 定制 Linux 系统

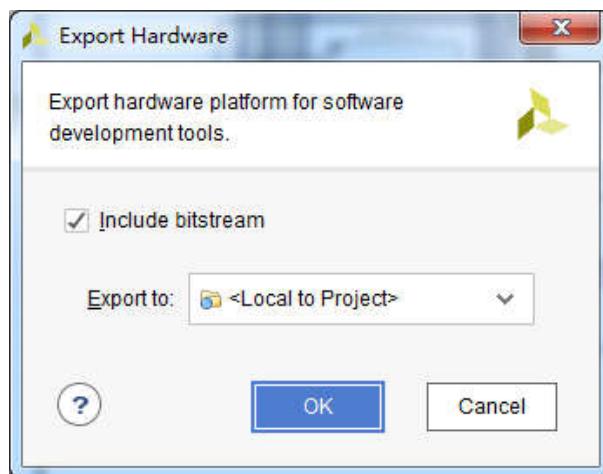
实验 Vivado 工程为 “linux_base”。

前面的教程中我们搭建好了 Petalinux 环境，本教程主要演示如何使用 Petalinux。需要注意的是，本实验只有 Linux 主机可以连接互联网的情况下才能完成。

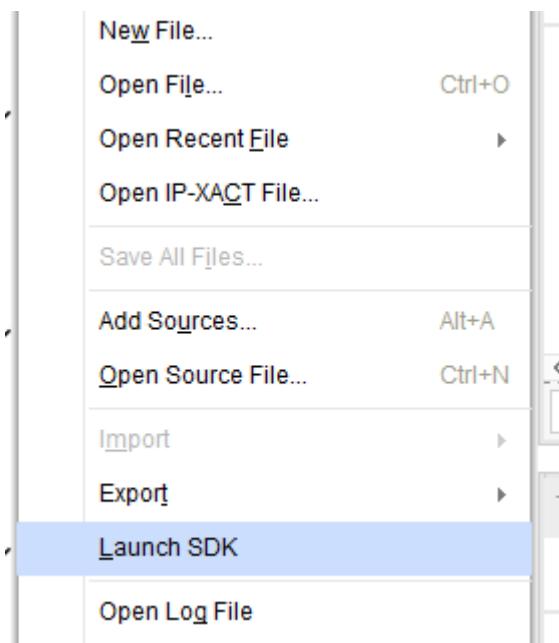
22.1 Vivado 工程

使用 Petalinux 可以非常方便地定制嵌入式 Linux 系统，只需要 Vivado 软件把硬件信息导出，然后 Petalinux 根据这些信息来配置 uboot，内核、文件系统等，Vivado 工程建立的步骤在前面的实验中大量讲解，这里不再讲解。

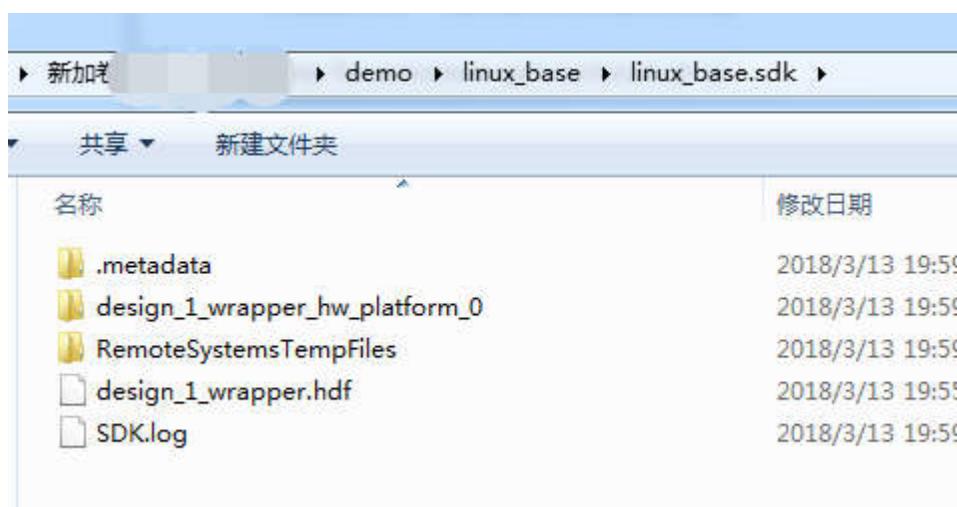
- 1) 编译生成 bit 文件
- 2) 导出硬件信息



- 3) 运行 SDK

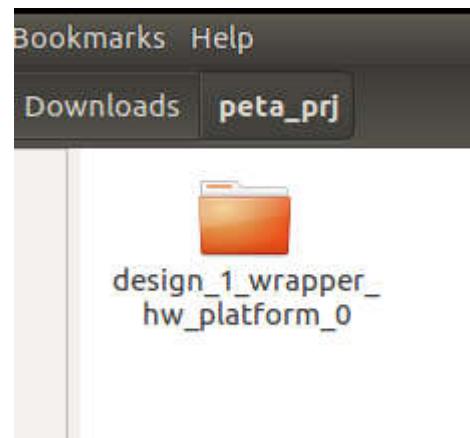


- 4) 在 vivado 的工程目录下会有一个*.sdk 的目录，下面有一个“design_1_wrapper_hw_platform_0”目录，这个目录就包含了 petalinux 使用的文件。



22.2 使用 Petalinux 建立工程

- 1) 把 “design_1_wrapper_hw_platform_0” 目录复制到 Linux 主机中



2) 打开终端，进入工作目录

```
alinx@ubuntu: ~/Downloads/peta_prj
alinx@ubuntu:~/Downloads/peta_prj$
```

3) 设置 petalinux 环境变量，运行下面命令

```
source /opt/pkg/petalinux/settings.sh
```

```
alinx@ubuntu: ~/Downloads/peta_prj
alinx@ubuntu:~/Downloads/peta_prj$ source /opt/pkg/petalinux/settings.sh
PetaLinux environment set to '/opt/pkg/petalinux'
WARNING: /bin/sh is not bash!
bash is PetaLinux recommended shell. Please set your default shell to bash.
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
WARNING: No tftp server found - please refer to "PetaLinux SDK Installation Guide" for its impact and solution
alinx@ubuntu:~/Downloads/peta_prj$
```

4) 运行下面命令设置 vivado 环境变量

```
source /opt/Xilinx/Vivado/2017.4/settings64.sh
```

```
alinx@ubuntu:~/Downloads/peta_prj$ source /opt/pkg/petalinux/settings.sh
Petalinux environment set to '/opt/pkg/petalinux'
WARNING: /bin/sh is not bash!
bash is Petalinux recommended shell. Please set your default shell to bash.
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
WARNING: No tftp server found - please refer to "Petalinux SDK Installation Guide" for its impact and solution
alinx@ubuntu:~/Downloads/peta_prj$ source /opt/Xilinx/Vivado/2017.4/settings64.sh
alinx@ubuntu:~/Downloads/peta_prj$
```

- 5) 使用下面命令创建一个 petalinux 工程，工程名为 ax_peta，这个时候 petalinux 会自动创建一个名为 ax_peta 的工程。

```
petalinux-create --type project --template zynq --name ax_peta
```

The terminal window shows the creation of a new Petalinux project named 'ax_peta'. A red box highlights the newly created folder 'ax_peta' in the file browser on the left.

```
alinx@ubuntu:~/Downloads/peta_prj$ source /opt/pkg/petalinux/settings.sh
Petalinux environment set to '/opt/pkg/petalinux'
WARNING: /bin/sh is not bash!
bash is Petalinux recommended shell. Please set your default shell to bash.
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
WARNING: No tftp server found - please refer to "Petalinux SDK Installation Guide" for its impact and solution
alinx@ubuntu:~/Downloads/peta_prj$ source /opt/Xilinx/Vivado/2017.4/settings64.sh
alinx@ubuntu:~/Downloads/peta_prj$ petalinux-create --type project --template zynq --name ax_peta
INFO: Create project: ax_peta
INFO: New project successfully created in /home/alinx/Downloads/peta_prj/ax_peta
alinx@ubuntu:~/Downloads/peta_prj$
```

- 6) 使用下面的命令进入 petalinux 工作目录

```
cd ax_peta
```

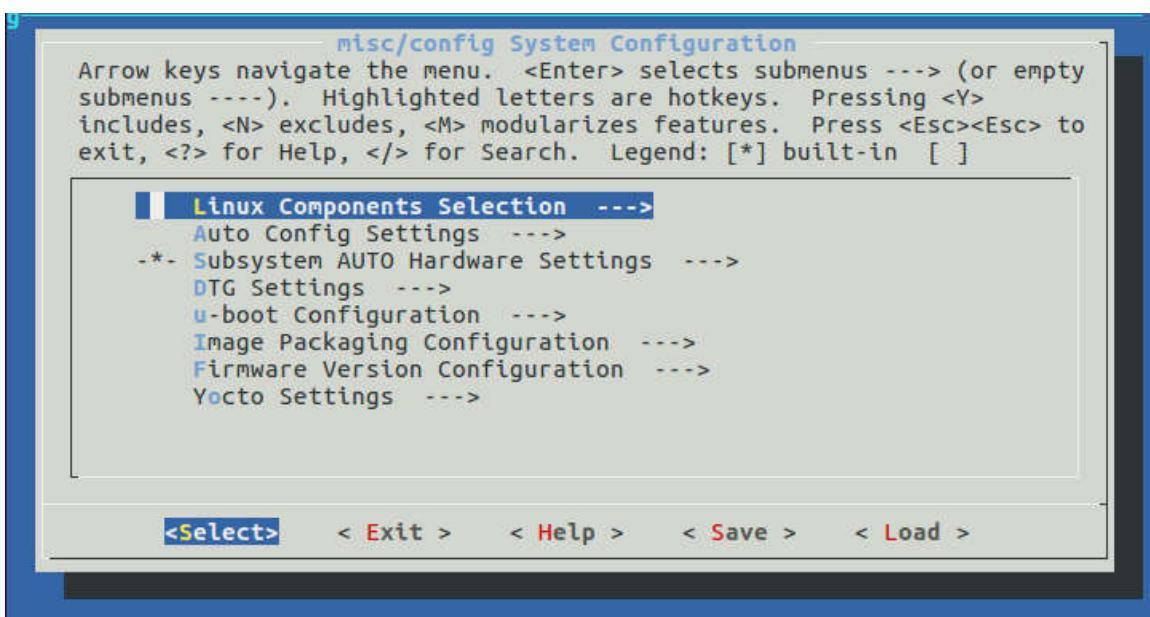
```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
alinx@ubuntu:~/Downloads/peta_prj$ source /opt/pkg/petalinux/settings.sh
Petalinux environment set to '/opt/pkg/petalinux'
WARNING: /bin/sh is not bash!
bash is Petalinux recommended shell. Please set your default shell to bash.
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
WARNING: No tftp server found - please refer to "Petalinux SDK Installation Guide" for its impact and solution
alinx@ubuntu:~/Downloads/peta_prj$ source /opt/Xilinx/Vivado/2017.4/settings64.sh
alinx@ubuntu:~/Downloads/peta_prj$ petalinux-create --type project --template zynq --name ax_peta
INFO: Create project: ax_peta
INFO: New project successfully created in /home/alinx/Downloads/peta_prj/ax_peta
alinx@ubuntu:~/Downloads/peta_prj$ cd ax_peta/
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

- 7) 使用下面命令配置 Petalinux 工程的硬件信息，“`../design_1_wrapper_hw_platform_0`” 目录就是 vivado 导出的硬件信息。

```
petalinux-config --get-hw-description ../design_1_wrapper_hw_platform_0
```

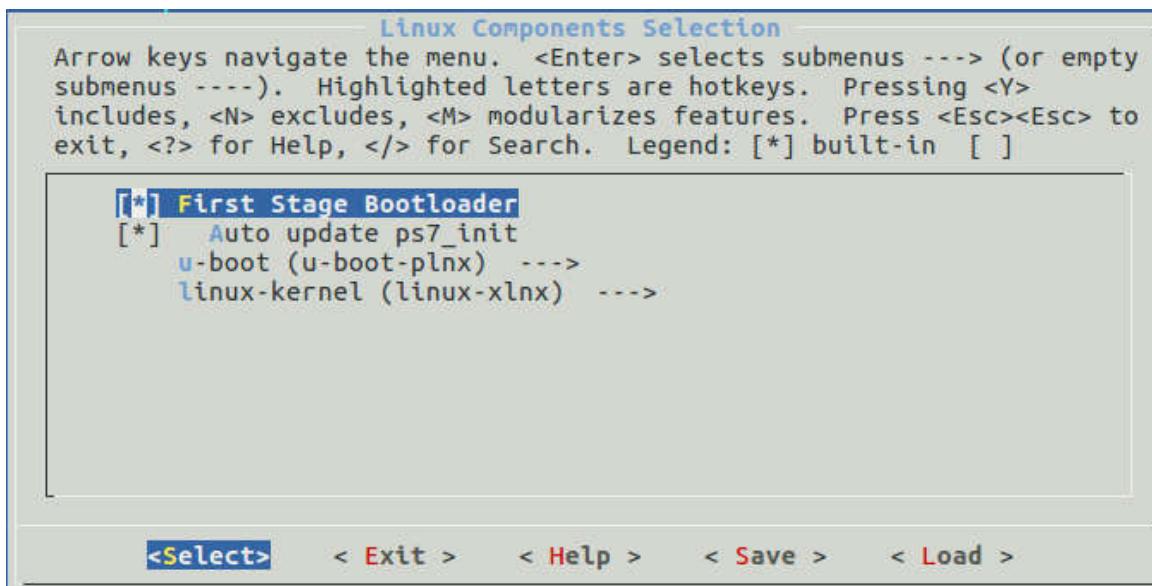
```
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-config --get-hw-description
../design_1_wrapper_hw_platform_0/
INFO: Getting hardware description...
[INFO] generating Kconfig for project
```

- 8) 在弹出一个窗口里可以配置 petalinux 工程，如果配置过后想再次配置，可以运行命令“`petalinux-config`”来配置。

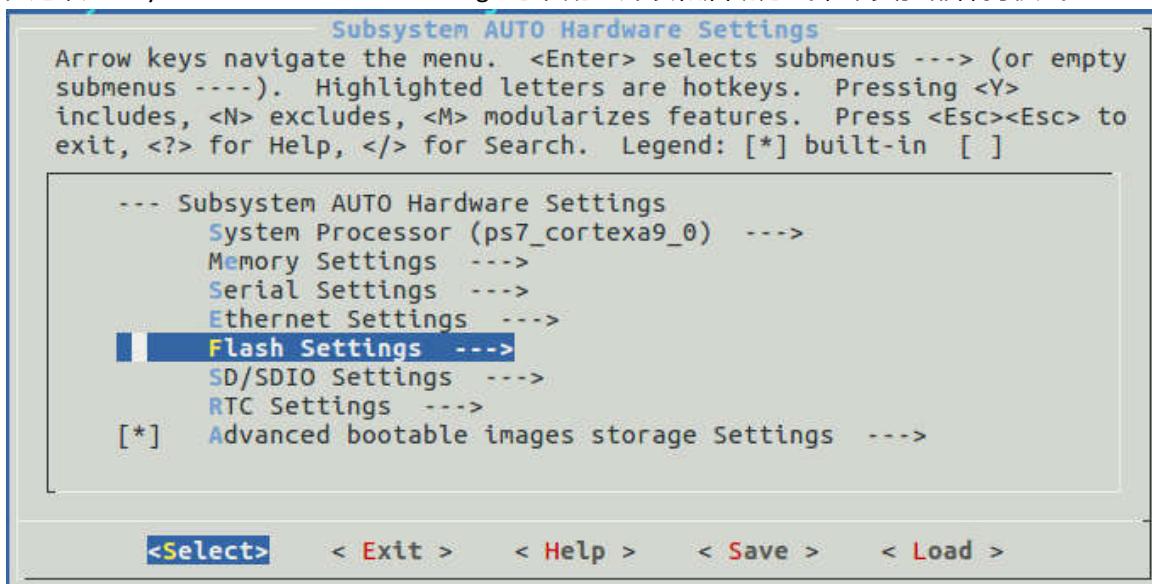


- 9) 在选项 Linux Components Selection 中可以配置 uboot 和 Linux 内核的来源，默认是 git 上下

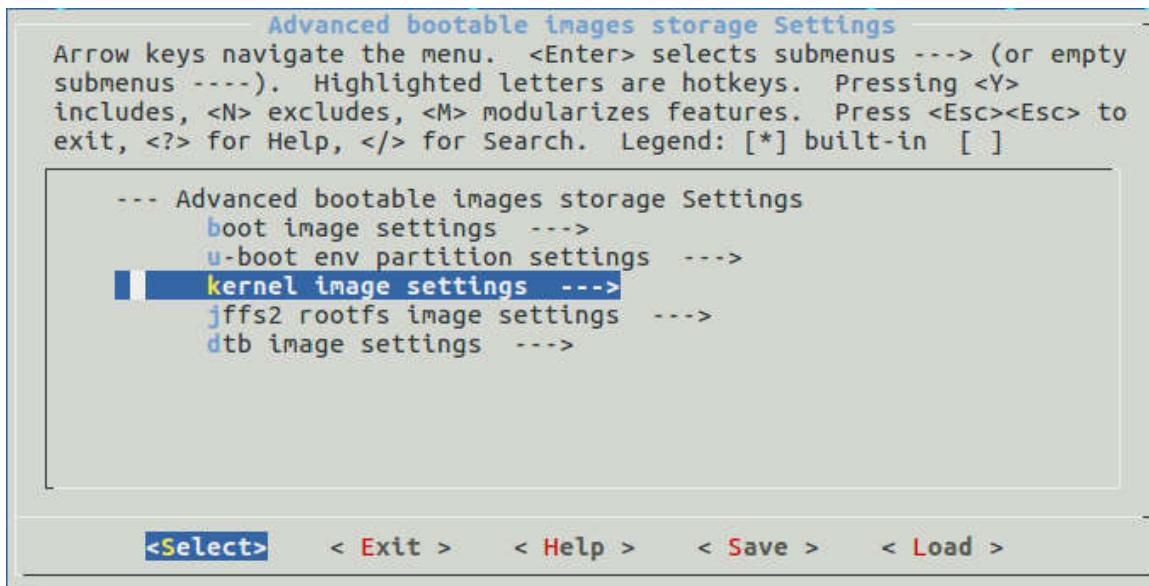
载的，需要 Linux 主机连接互联网才能下载。本实验保持默认配置。



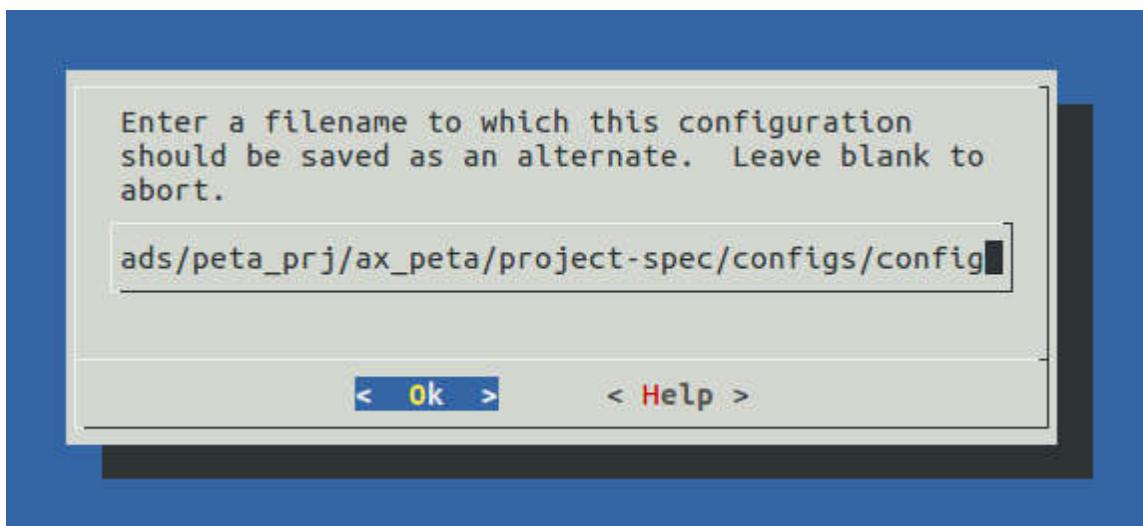
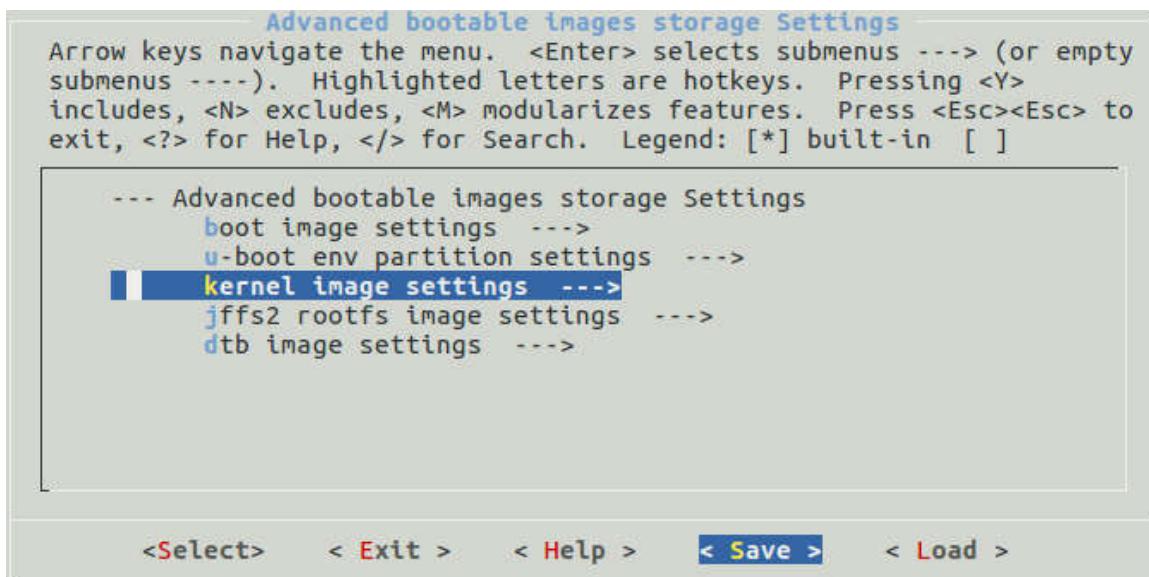
- 10) 在选项 Subsystem AUTO Hardware Settings 可以配置外设和启动方式，本实验都保持模式



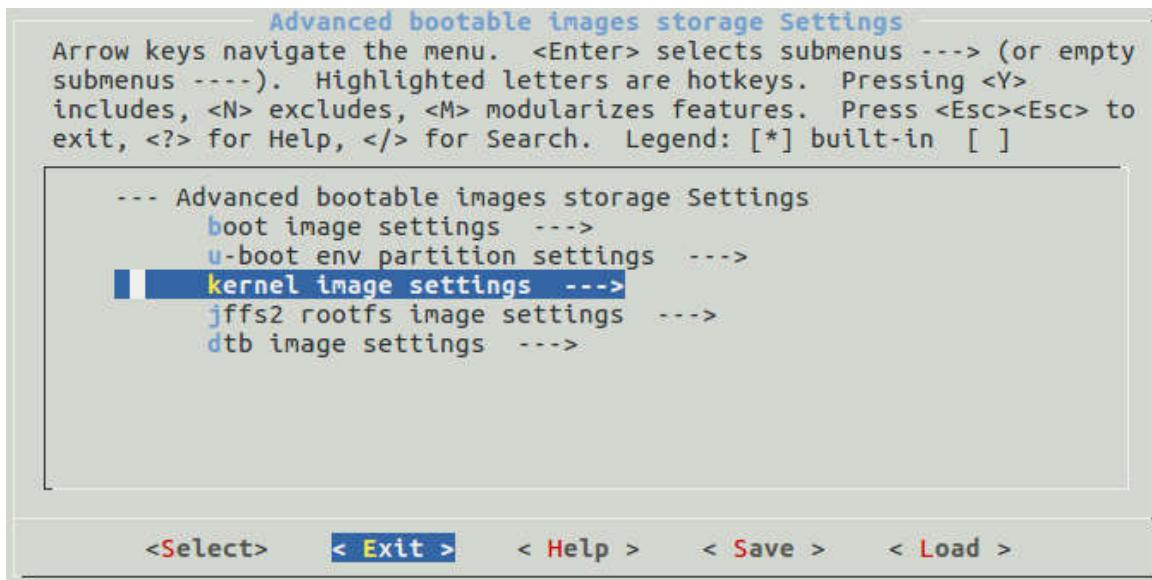
- 11) 在 Advanced bootable images storage Settings 选项中配置启动方式，默认从 sd 卡启动，为了调试这里保持默认从 sd 卡启动，如果需要制作一个从 QSPI flash 启动的嵌入式 Linux，可以在那里配置。



12) 配置完成后保持设置，本实验基本都是默认配置



13) 退出配置界面



14) 开始一个比较漫长的等待

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
INFO: Getting hardware description...
[INFO] generating Kconfig for project

[INFO] menuconfig project
/home/alinkx/Downloads/peta_prj/ax_peta/build/misc/config/Kconfig.syshw:30:warning: defaults for choice values not supported
/home/alinkx/Downloads/peta_prj/ax_peta/build/misc/config/Kconfig:568:warning: config symbol defined without type

*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.

[INFO] sourcing bitbake
[INFO] generating plnxtool conf
[INFO] generating meta-plnx-generated layer
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_peta
~/Downloads/peta_prj/ax_peta
[INFO] generating machine configuration
[INFO] generating bbappends for project . This may take time !
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_peta
```

22.3 配置 Linux 内核

1) 使用下面命令配置内核，运行命令后又要等待很长一段时间

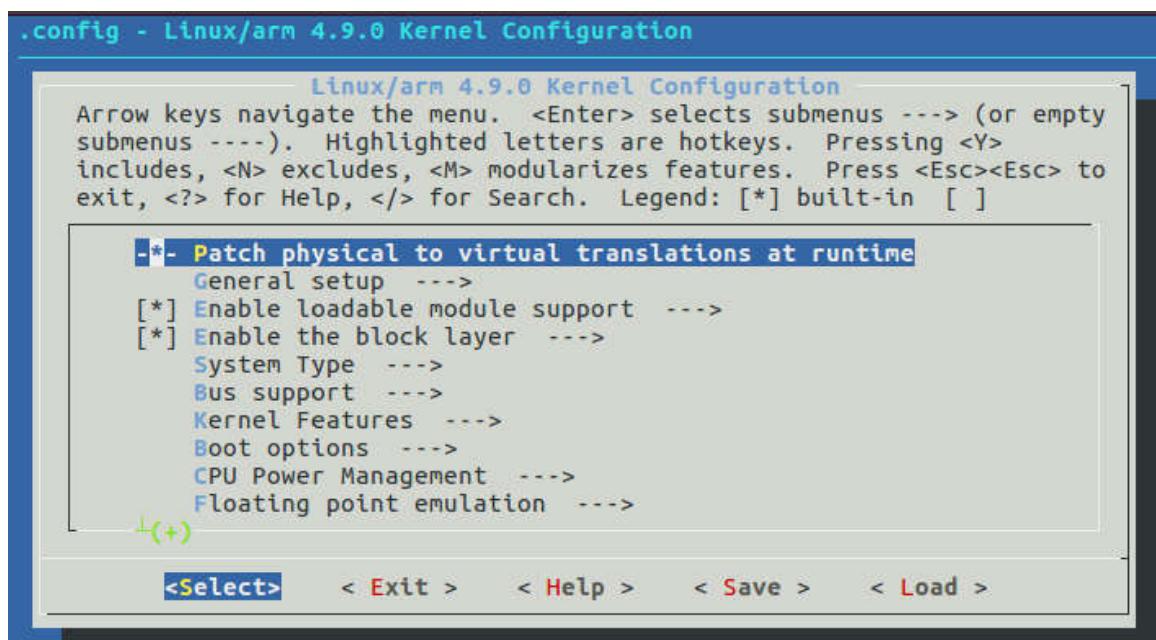
```
petalinux-config -c kernel
```

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
eta
~/Downloads/peta_prj/ax_peta
[INFO] generating u-boot configuration files

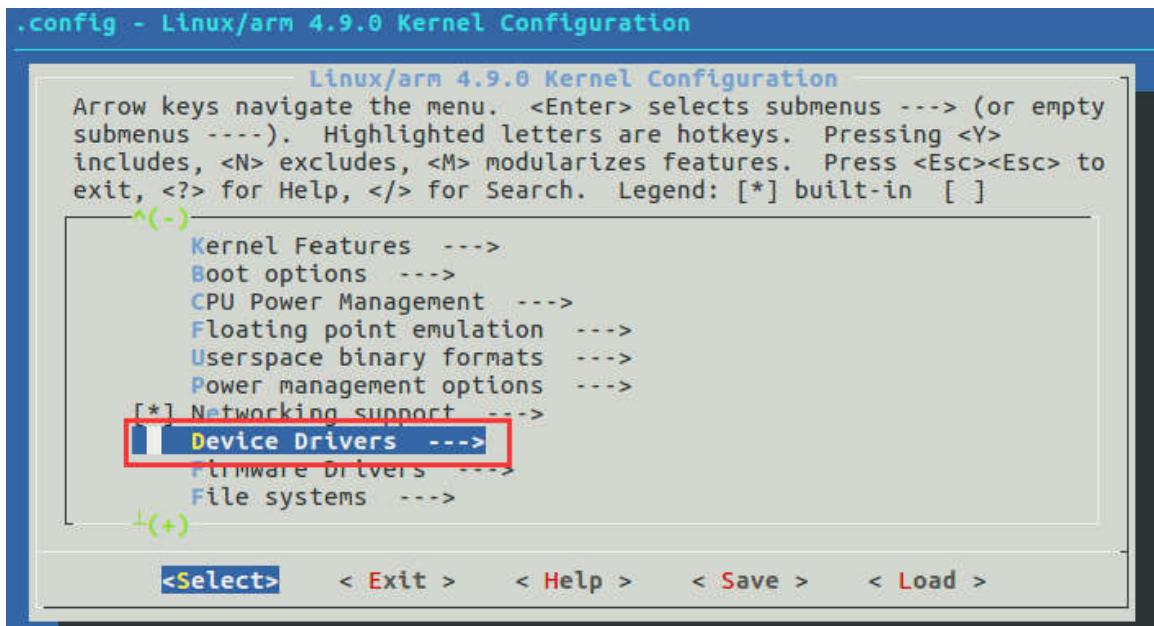
[INFO] generating kernel configuration files
[INFO] generating kconfig for Rootfs
Generate rootfs kconfig
[INFO] oldconfig rootfs
[INFO] generating petalinux-user-image.bb
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-config -c kernel
[INFO] generating Kconfig for project

[INFO] sourcing bitbake
[INFO] generating plnxtool conf
[INFO] generating meta-plnx-generated layer
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_peta
~/Downloads/peta_prj/ax_peta
[INFO] generating machine configuration
[INFO] configuring: kernel
[INFO] generating kernel configuration files
[INFO] bitbake virtual/kernel -c menuconfig
Parsing recipes: 14% |#####| ETA: 0:01:29
                                         | ETA: 0:01:31
```

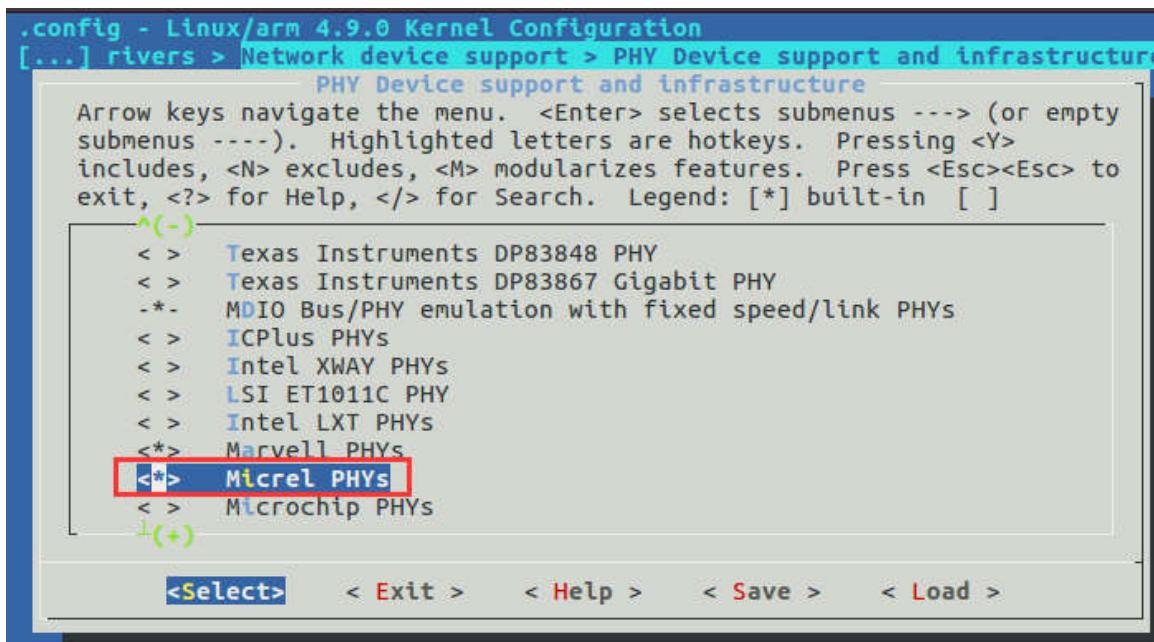
- 2) 等待一段时间后弹出配置内核的配置界面



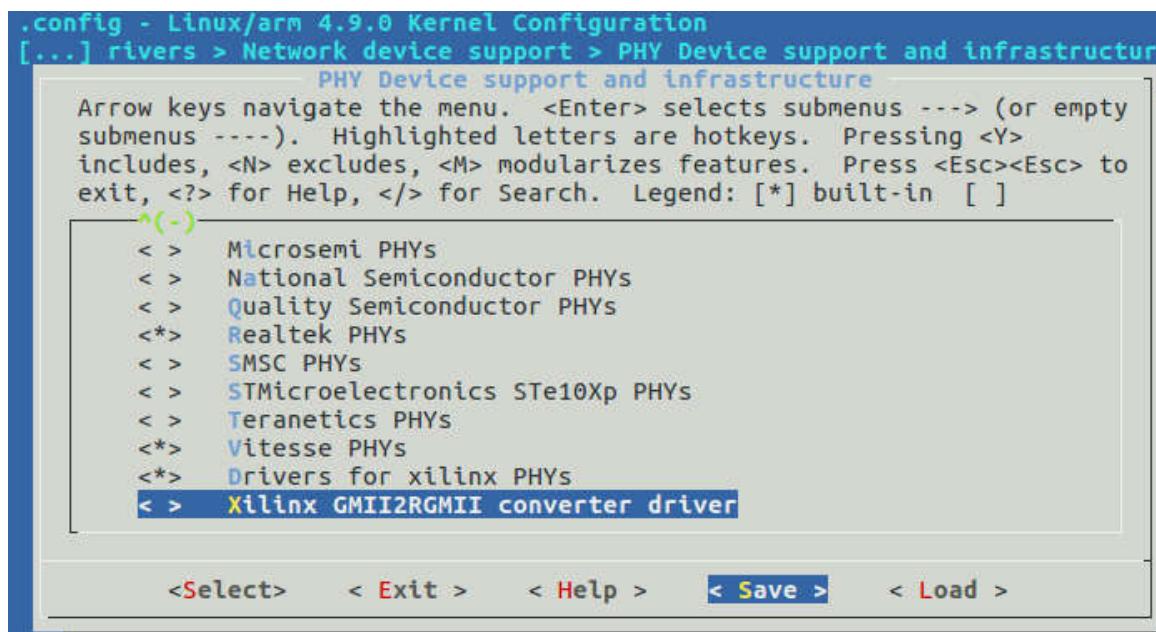
- 3) 由于以太网的 phy 芯片驱动默认没有打开，需要配置驱动，在选项 Device Drivers 配置驱动



- 4) 然后到 Network device support > PHY Device support and infrastructure , 选中 Micrel PHYs , 按 Y 键。



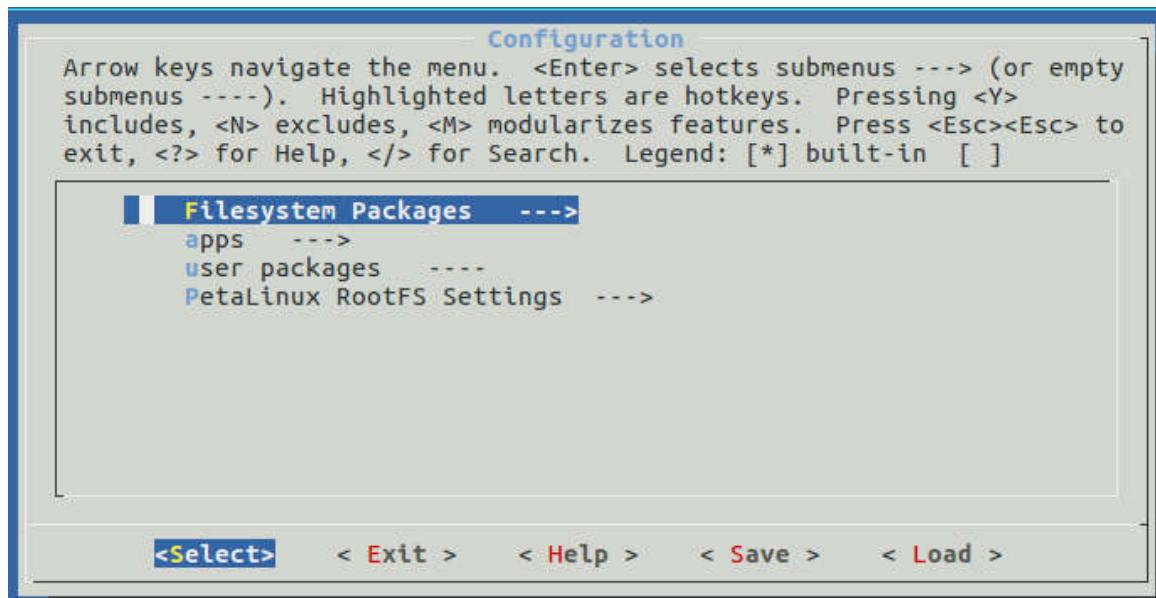
- 5) 其他的不需要再配置 , 保存配置并退出



22.4 配置根文件系统

运行下面的命令配置根文件系统,可以根据需求来配置根文件系统,本实验保持默认配置。

```
petalinux-config -c rootfs
```



22.5 编译

- 1) 使用下面命令配置编译 uboot、内核、根文件系统、设备树等。

```
petalinux-build
```

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
~/Downloads/peta_prj/ax_peta
[INFO] generating machine configuration
[INFO] configuring: rootfs
[INFO] generating kconfig for Rootfs
Generate rootfs kconfig
[INFO] menuconfig rootfs

*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.

[INFO] generating petalinux-user-image.bb
[INFO] successfully configured rootfs
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-build
[INFO] building project
[INFO] sourcing bitbake
INFO: bitbake petalinux-user-image
Loading cache: 100% |#####| Time: 0:00:00
Loaded 3257 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:03
Parsing of 2466 .bb files complete (2434 cached, 32 parsed). 3259 targets, 226 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 44% |#####| ETA: 0:00:06
```

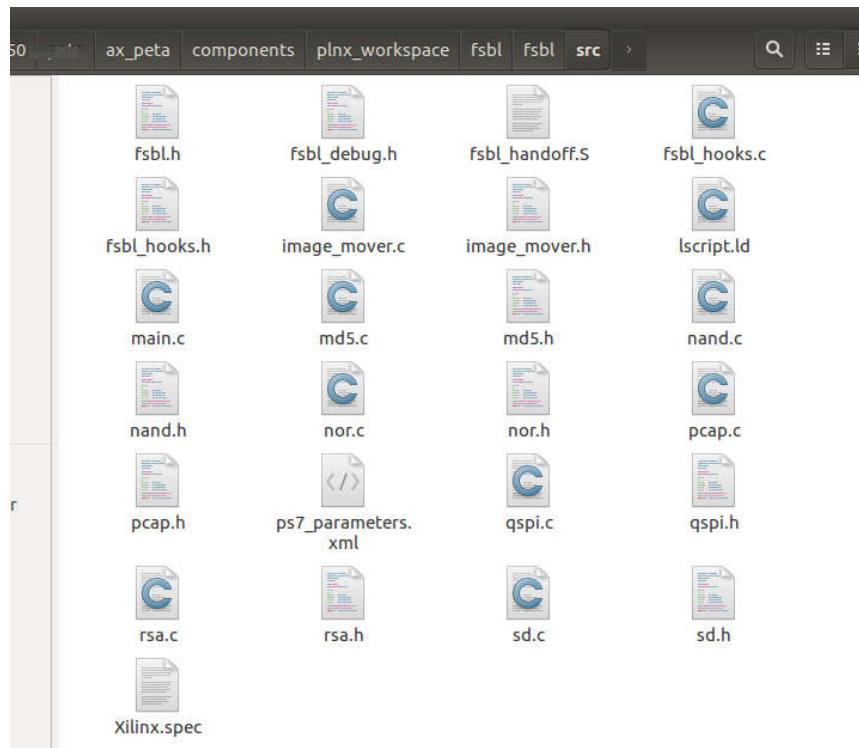
- 2) 编译完成

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
[INFO] successfully configured rootfs
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-build
[INFO] building project
[INFO] sourcing bitbake
INFO: bitbake petalinux-user-image
Loading cache: 100% |#####| Time: 0:00:00
Loaded 3257 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:03
Parsing of 2466 .bb files complete (2434 cached, 32 parsed). 3259 targets, 226 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#####| Time: 0:00:08
Checking sstate mirror object availability: 100% |#####| Time: 0:32:53
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
fsbl-2017.4+gitAUTOINC+77448ae629-r0 do_compile: NOTE: fsbl: compiling from external source tree /opt/pkg/petalinux/tools/hsm/data/embeddedsw
NOTE: Tasks Summary: Attempted 2444 tasks of which 1884 didn't need to be rerun and all succeeded.
INFO: Copying Images from deploy to images
INFO: Creating images/linux directory
NOTE: Failed to copy built images to tftp dir: /tftpboot
[INFO] successfully built project
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

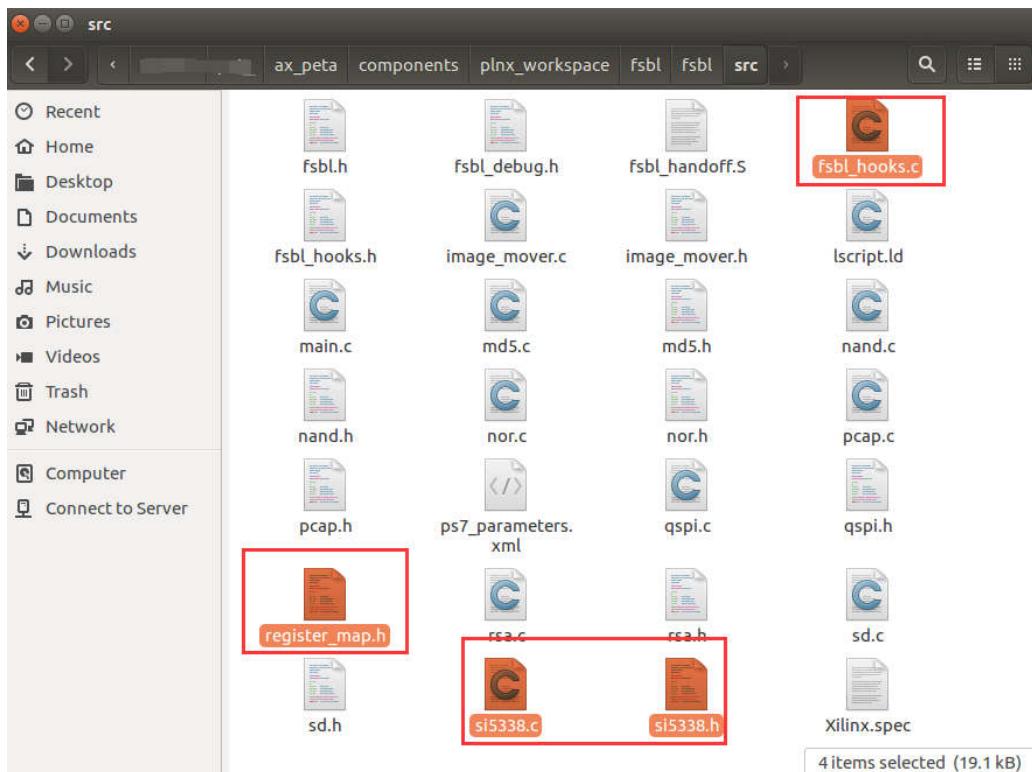
22.6 修改 FSBL

由于开发板使用一个 SI5338 来产生差分时钟，PCIe Root 模块使用了这个时钟，所以我们要在 Linux 系统启动前配置好 SI5338，这里要修改 FSBL 代码。

- 1) 找到 FSBL 源文件，在工程目录的 components/plnx_workspace/fsbl/fsbl/src



- 2) 把 fsbl_hooks.c 文件替换为修改过后的文件，复制 si5338.c、si5338.h 和 register_map.h 文件
复制到 src 目录



3) 再次使用下面命令编译

```
petalinux-build
```

22.7 生成 BOOT 文件

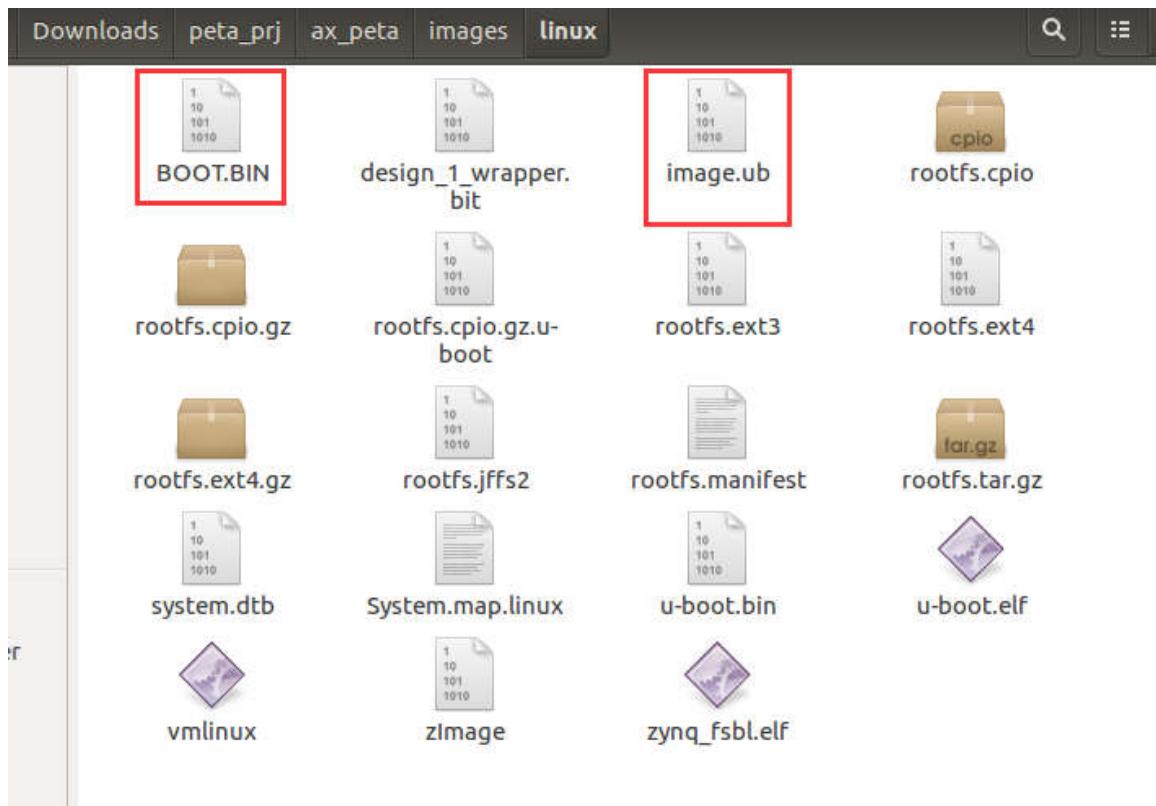
运行下面命令生成 BOOT 文件，注意空格和短线

```
petalinux-package --boot --fsbl ./images/linux/zynq_fsbl.elf  
--fpga ./images/linux/design_1_wrapper.bit --u-boot --force
```

```
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-package --boot --fsbl ./ima  
ges/linux/zynq_fsbl.elf --fpga ./images/linux/design_1_wrapper.bit --uboot --for  
ce  
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/zyn  
q_fsbl.elf"  
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/des  
ign_1_wrapper.bit"  
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/u-b  
oot.elf"  
INFO: Generating zynq binary package BOOT.BIN...  
INFO: Binary is ready.  
WARNING: Unable to access the TFTPBOOT folder /tftpboot!!!  
WARNING: Skip file copy to TFTPBOOT folder!!!  
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

22.8 测试 Linux

- 1) 将工程目录 images -> linux 目录中的 BOOT.BIN 和 image.ub 复制到 sd 卡 , 复制前最好先格式化一下 sd 卡 , 然后插到开发板上 , 开发板设置到 sd 卡启动



- 2) 打开串口终端 , 启动开发板

```

update-rc.d: /etc/init.d/run-postinsts exists during rc.d purge (continuing)
  Removing any system startup links for run-postinsts ...
  /etc/rcS.d/S99run-postinsts
INIT: Entering runlevel: 5
Configuring network interfaces... IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
udhcpc (v1.24.1) started
Sending discover...
Sending discover...
Sending discover...
No lease, forking to background
done.
Starting Dropbear SSH server: Generating key, this may take a while...
Public key portion is:
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCOdWDQcYL7eEXy5zFIkNEQ/nc9jL6uOHocJZ+EIZO
ZyA1LVCKvvSXsQeaTy2FtdKuj8+2H6eFo3OL/bcCst2twhA7njTjA+mwtZ7D83H0HdKx1+xWEuk6S12
Fingerprint: md5 38:1d:b7:ba:35:d7:52:50:3b:2f:d9:06:7c:60:f4:79
dropbear.
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting syslogd/klogd: done
Starting tcf-agent: OK

PetaLinux 2017.4 ax_peta /dev/ttys0
ax_peta login: root

```

- 3) 使用 root 登录，默认密码 root , ETH1 (PS) 插上网线后 (路由器支持自动获取 IP) , 使用 ifconfig 命令可以看到网络状态。

```

root@ax_peta:~# macb e000b000.ethernet eth0: link up (1000/Full)
IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready

root@ax_peta:~# ifconfig
eth0      Link encap:Ethernet HWaddr 00:0A:35:00:1E:53
          inet addr:192.168.1.46  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20a:35ff:fe00:1e53%lo/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:20 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4350 (4.2 KiB)  TX bytes:1902 (1.8 KiB)
          Interrupt:29 Base address:0xb000

lo       Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1%1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@ax_peta:~#

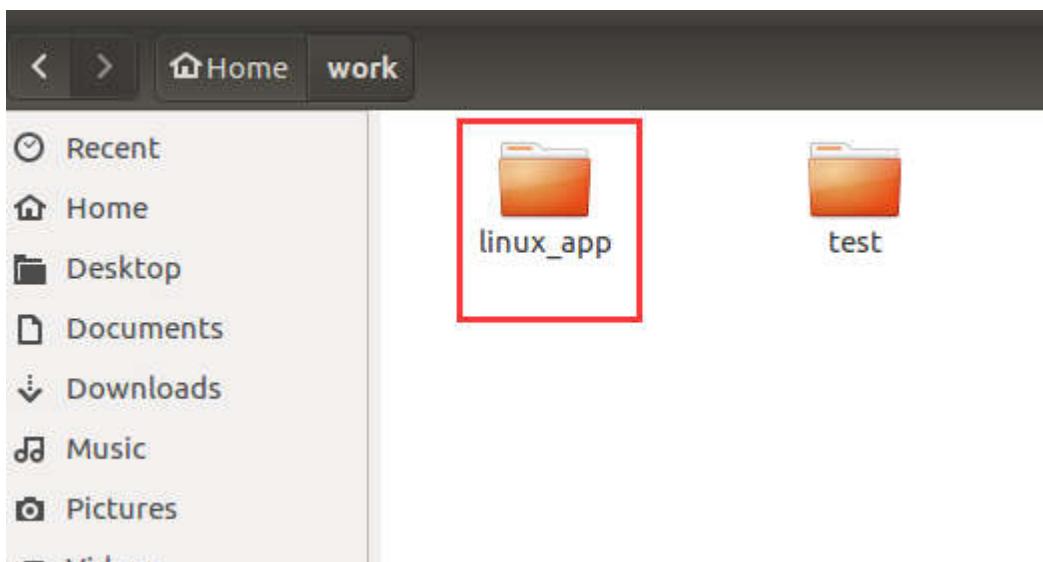
```

第二十三章 使用 SDK 开发 Linux 程序

前面的教程中我们使用 petalinux 制作了一个嵌入式 Linux 系统，本实验要做一个 Linux 应用程序，然后在开发板上运行。本实验需要使用上面的实验中做好的 Linux 运行环境。

23.1 使用 SDK 建立 Linux 应用程序

- 1) 在/home/alinx/work 建立目录 linux_app，用于 SDK 的工作空间

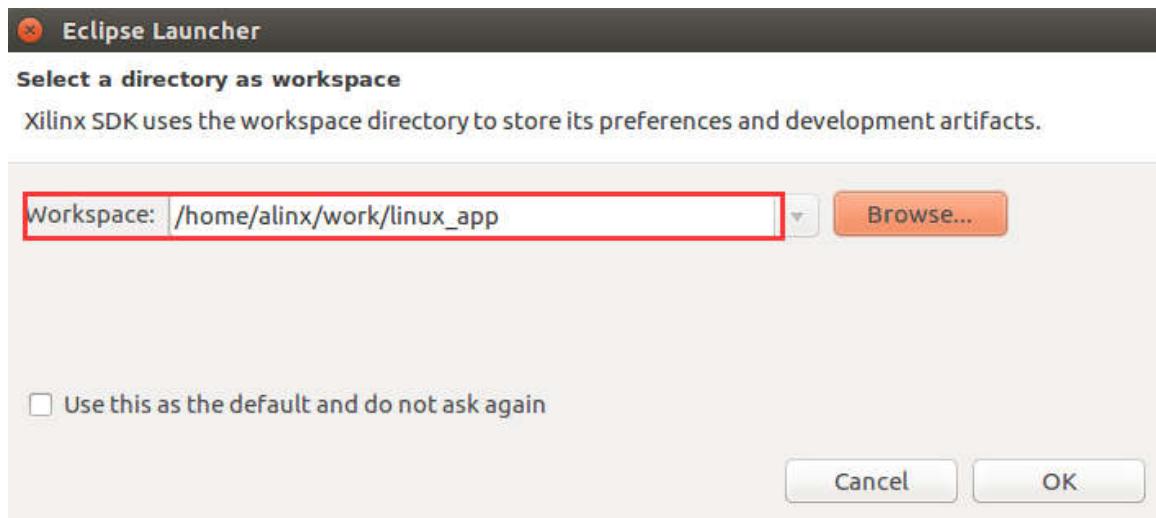


- 2) 设置 Vivado 环境变量，运行 SDK

```
source /opt/Xilinx/Vivado/2017.4/settings64.sh  
xsdk
```

```
alinx@ubuntu:~/work$ source /opt/Xilinx/Vivado/2017.4/settings64.sh  
alinx@ubuntu:~/work$ xsdk  
***** Xilinx Software Development Kit  
***** SDK v2017.4 (64-bit)  
**** SW Build 2086221 on Fri Dec 15 20:54:30 MST 2017  
** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.  
  
Launching SDK with command /opt/Xilinx/SDK/2017.4/eclipse/lnx64.o/eclipse -vmargs  
-Xms64m -Xmx4G -Dorg.eclipse.swt.internal.gtk.cairoGraphics=false  
alinx@ubuntu:~/work$
```

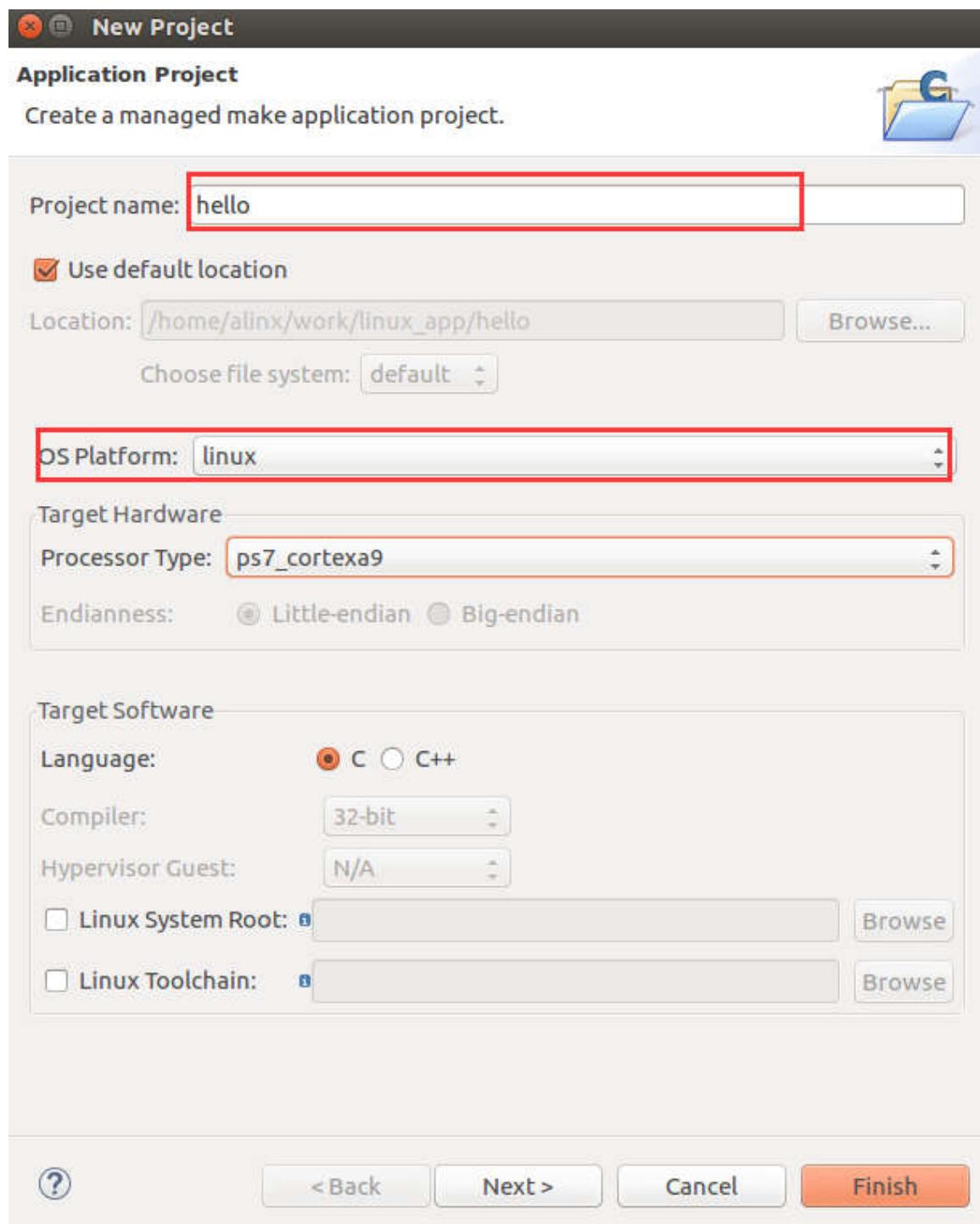
- 3) 工作空间选/home/alinx/work/linux_app



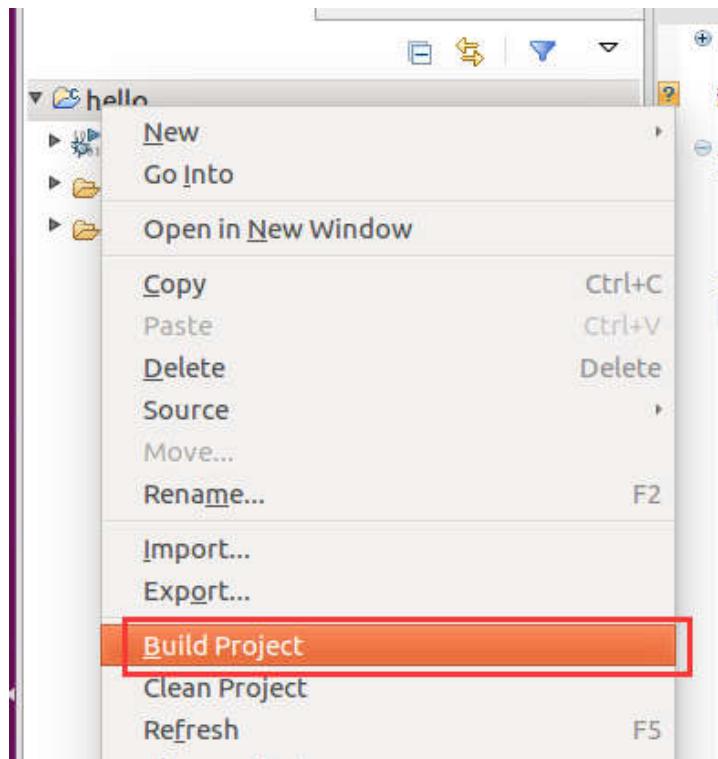
4) 选择 Create Application Project



5) 工程名填写 hello , OS Platform 选择 linux



6) Build Project



23.2 通过 NFS 共享运行

- 1) 开发板插上网线 (PS、ETH1 , 需要路由器支持自动获取 IP) , 上电 , 挂载 Linux 主机 NFS , 其中 192.168.1.77 为主机地址 , /home/alinx/work 为 NFS 目录 , /mnt 为开发板目录。这里要求主机和开发板在一个网段。

```
mount -t nfs -o nolock 192.168.1.77:/home/alinx/work /mnt
```

```

Sending select for 192.168.1.46...
Lease of 192.168.1.46 obtained, lease time 86400
/etc/udhcpc.d/50default: Adding DNS 192.168.1.1
done.
Starting Dropbear SSH server: Generating key, this may take a while...

Public key portion is:
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB3fwPFZmbH0qhjkpV+J/zQsBk5nYuFDON9kbBkN7dl
B4MEEC/2aV7AmR0S7SFQ1JZedVCh2YBREBy1mBV2ld+Up125nGQnwwSOLj7T6kiFLU5fxRqDSHfxoOu5
nQq4ck7yS5kggDbHE/vUD1yyE22J1AwcARDI91VRYCsv/aoumLWoEL3y85VLIwsrkNPUsz8L3sXKICauv
IAshRYbkxc2zzsc4HkMGJZ/NXL2MC527taECp5y2DmXHdEUTIDSvpXLH1cFf9PRFzjhaEoJpXgAu3thJ
AGTza2ZPDmghv5MTuAr5KVpgu2yTizcmoMDeZWTmdpnh+OtqjU29FrEvhhx1 root@ax_peta
Fingerprint: md5 f0:36:14:f7:ad:e9:48:56:87:71:80:b0:d7:f7:dd:6a
dropbear.
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting syslogd/klogd: done
Starting tcf-agent: OK

Petalinux 2017.4 ax_peta /dev/ttys0

ax_peta login: root
Password:
root@ax_peta:~# mount -t nfs -o noblock 192.168.1.77:/home/alinx/work /mnt
root@ax_peta:~#

```

- 2) 进入/mnt/linux_app_hello/Debug 目录，运行 hello.elf，我们可以看到打印出 Hello World。

```

cd /mnt/linux_app_hello/Debug
./hello.elf

```

```

Public key portion is:
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB3fwPFZmbH0qhjkpV+J/zQsBk5nYuFDON9kbBkN7dl
B4MEEC/2aV7AmR0S7SFQ1JZedVCh2YBREBy1mBV2ld+Up125nGQnwwSOLj7T6kiFLU5fxRqDSHfxoOu5
nQq4ck7yS5kggDbHE/vUD1yyE22J1AwcARDI91VRYCsv/aoumLWoEL3y85VLIwsrkNPUsz8L3sXKICauv
IAshRYbkxc2zzsc4HkMGJZ/NXL2MC527taECp5y2DmXHdEUTIDSvpXLH1cFf9PRFzjhaEoJpXgAu3thJ
AGTza2ZPDmghv5MTuAr5KVpgu2yTizcmoMDeZWTmdpnh+OtqjU29FrEvhhx1 root@ax_peta
Fingerprint: md5 f0:36:14:f7:ad:e9:48:56:87:71:80:b0:d7:f7:dd:6a
dropbear.
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting syslogd/klogd: done
Starting tcf-agent: OK

Petalinux 2017.4 ax_peta /dev/ttys0

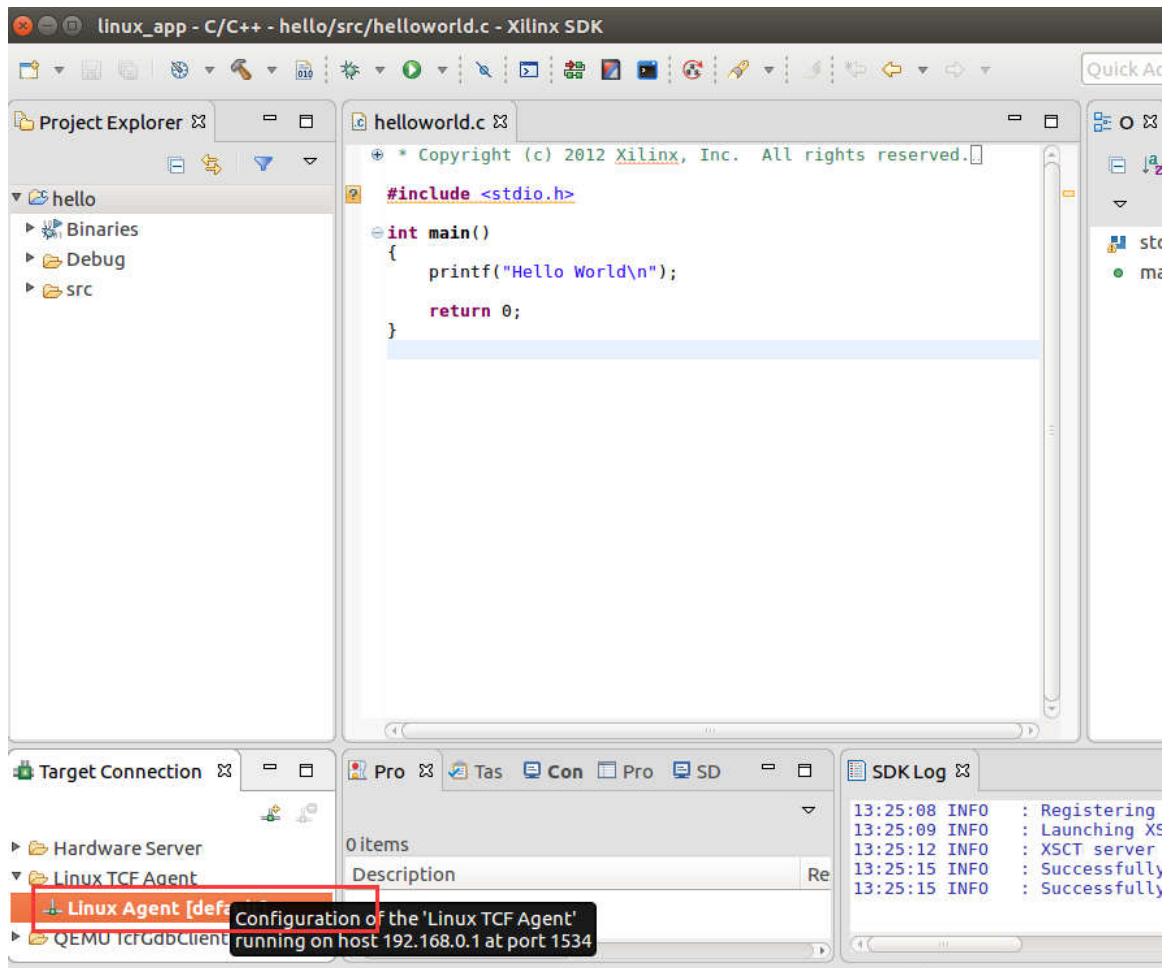
ax_peta login: root
Password:
root@ax_peta:~# mount -t nfs -o noblock 192.168.1.77:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# cd linux_app/hello/Debug/
root@ax_peta:/mnt/linux_app/hello/Debug# random: crng init done

root@ax_peta:/mnt/linux_app/hello/Debug# ./hello.elf
Hello World
root@ax_peta:/mnt/linux_app/hello/Debug#

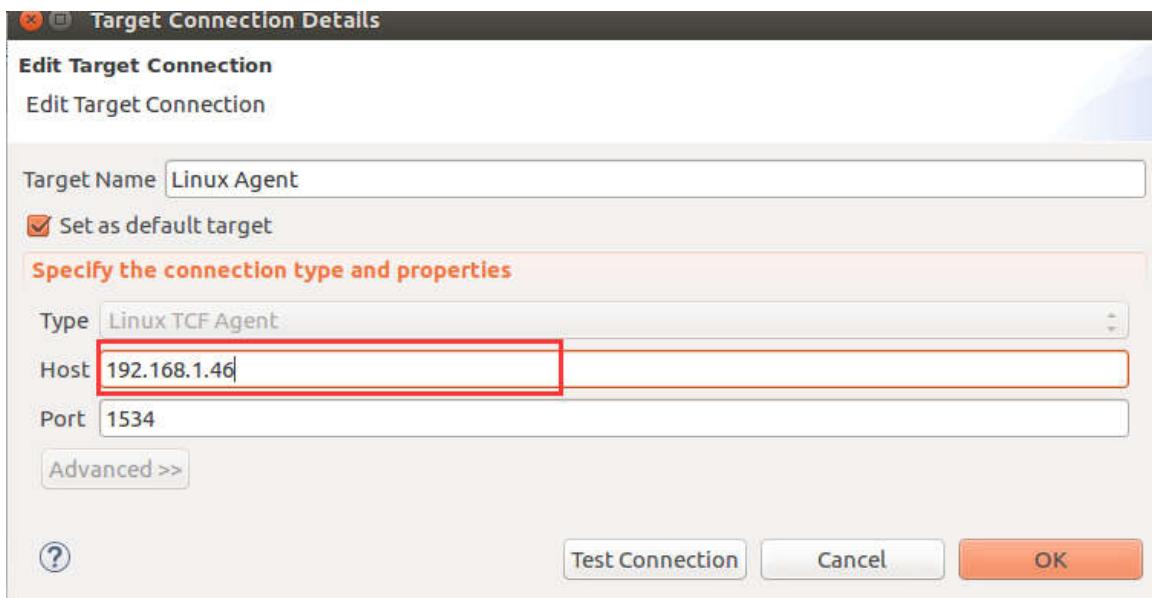
```

23.3 通过 TCF-Agent 运行调试

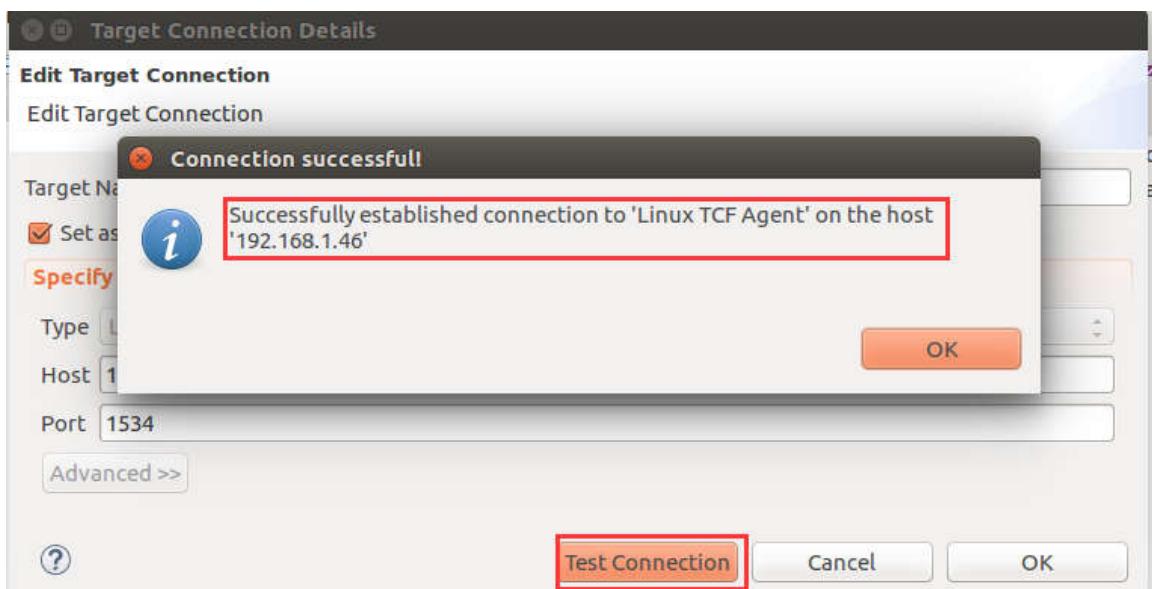
- 1) 双击 Linux Agent，配置连接参数



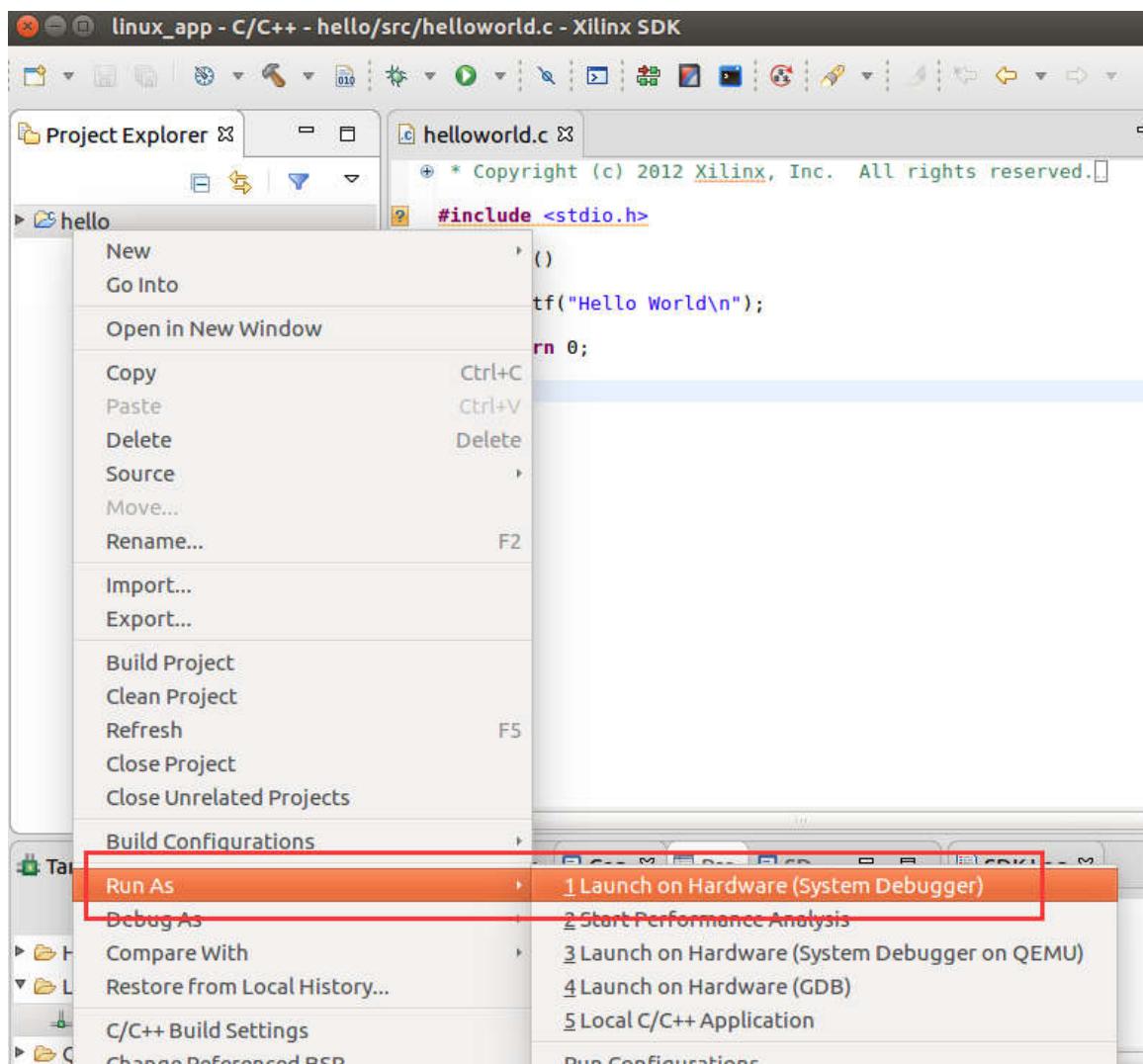
- 2) 填上 Host 的 IP 地址，这里填写的就是开发板的 IP 地址



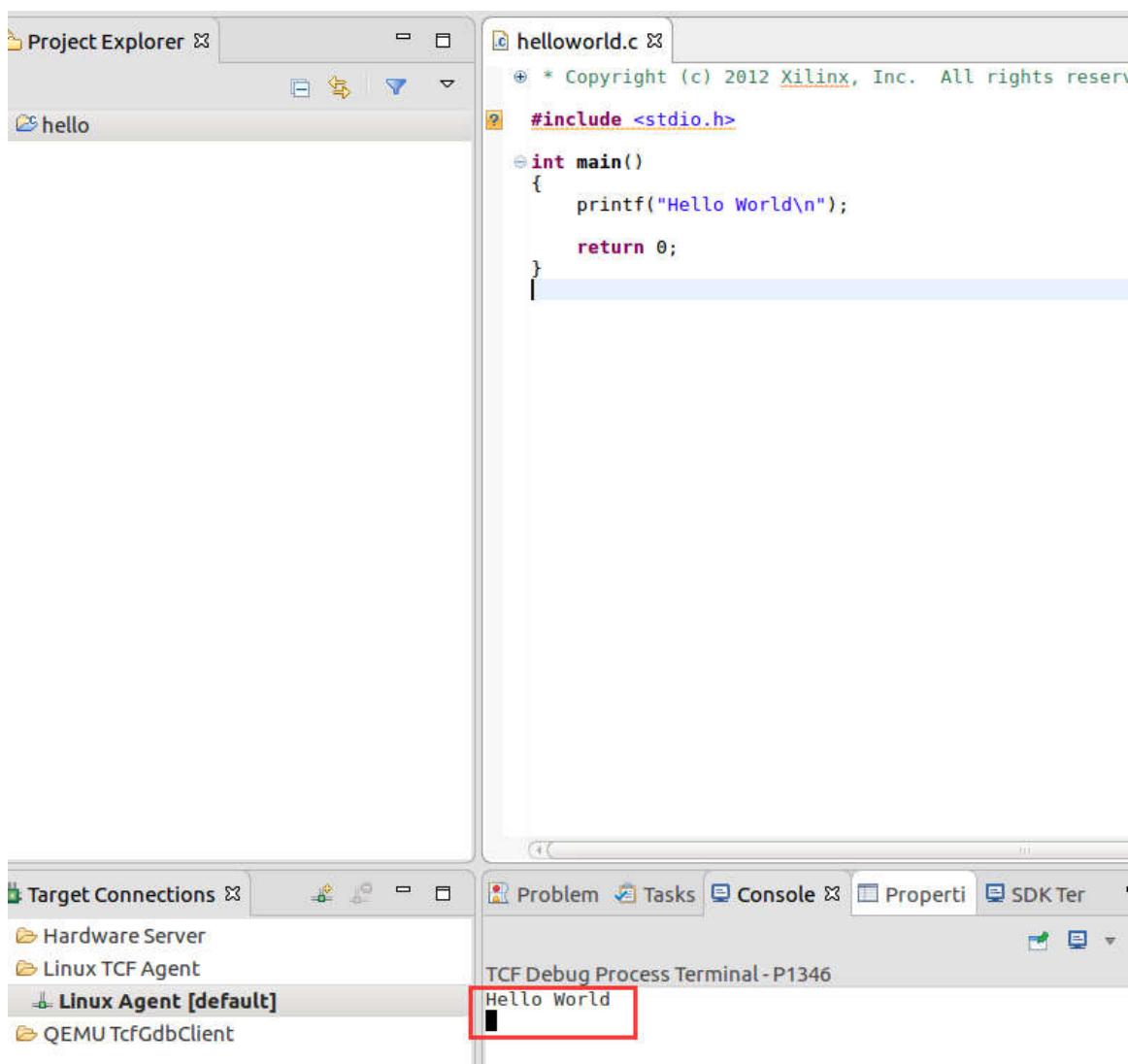
- 3) 点击 Test Connection ,如果测试连接成功 ,表示 Linux TCF Agent 服务正在运行 ,可以运行调试



- 4) 选择工程 ,右键运行



5) 在 debug 终端可以看到 Hello World 输出



23.4 TCF-Agent 问题

TCF-Agent 虽然可以很方便的运行调试 Linux 应用程序，也不需要 NFS 支持，但是对调试多线程程序支持不是很好，在应用程序崩溃时也不能很好的恢复调试环境，需要重启开发板，所以一直不常用。

第二十四章 Linux 下 GPIO 实验

前面的教程介绍如何使用 SDK 编写一个 zynq 版本的 helloworld 实验，本实验介绍如何控制 zynq 端外设，实验使用 GPIO 来举例，ZYNQ 的 GPIO 可以分为 2 种，一种是 PS 端自带的 GPIO，一种是使用 PL 实现的 GPIO，在建立 Vivado 工程时添加了 Xilinx 的 GPIO IP，大部分 Xilinx 提供的 IP 核在 Linux 下都已经有驱动，而且很多默认配置都是可以用的，像 AXI GPIO 驱动不需要在内核中再配置就可以使用。

在 <http://www.wiki.xilinx.com/Linux+Drivers> 页面我们可以找到所有 Linux 下 Xilinx 的驱动，例如 GPIO 驱动如下图所示，有些驱动给出了详细的用法。

GPIO	Zynq and Zynq Ultrascale+ MPSoC	GPIO Driver	Yes	drivers/gpio/gpio-zynq.c
GPIO	axi_gpio	AXI GPIO Driver	Yes	drivers/gpio/gpio-xilinx.c
HDMI Clocks	SI5324 Clock Multiplier/Jitter Attenuator	CCF SI5324 Driver	No	hdmi-modules/clk/*

在 GPIO 驱动详细页面 <http://www.wiki.xilinx.com/Linux%20GPIO%20Driver> 中介绍了 GPIO 驱动使用范围，设备树范例，以及如何写程序。

24.1 使用 SHELL 控制

Linux 提供了强大的 SHELL 功能，也是学习 Linux 必须掌握的技能，对于不熟悉 Linux 的命令和 SHELL 是令人头疼的，但是为了更好地学习 ZYNQ，必须熟悉掌握 SHELL，本教程不会去详细讲解 Linux 和 SHELL 的使用。

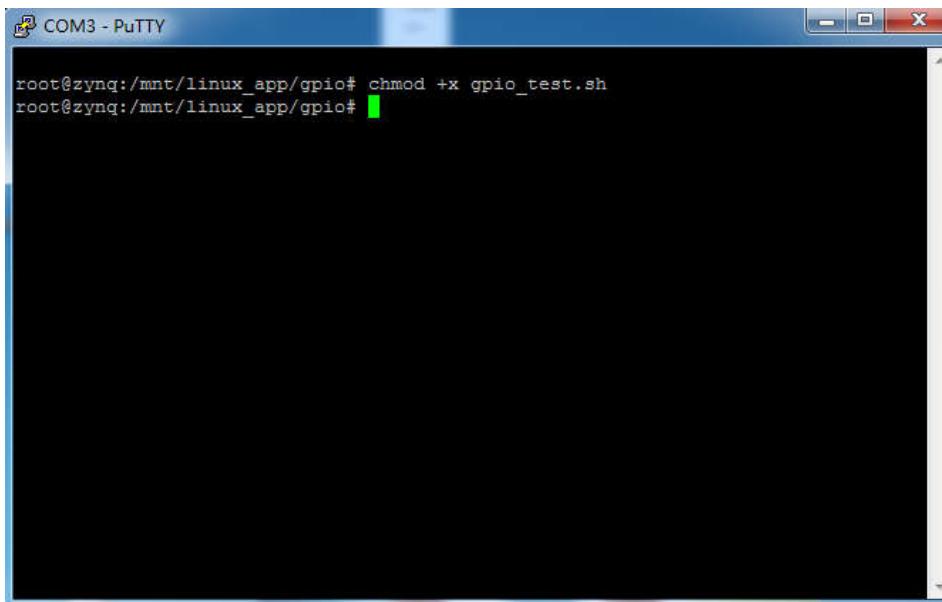
gpio_test.sh 文件内容如下，gpio_test 函数会根据参数来 export 一个 GPIO，然后一个 for 循环 3 次，每次先写 0 再写 1，调用了 5 次 gpio_test，依次点亮 5 个 LED，启动 898 是 PS 端的，其他是 PL 端的。

```
#!/bin/sh
gpio_test() {
    gpio=$1
    echo $gpio > /sys/class/gpio/export
    echo out > /sys/class/gpio/gpio${gpio}/direction
    for i in $(seq 1 3)
    do
        echo 0 >/sys/class/gpio/gpio${gpio}/value
        sleep 1
        echo 1 >/sys/class/gpio/gpio${gpio}/value
        sleep 1
    done
    echo $gpio > /sys/class/gpio/unexport
}
gpio_test 898
gpio_test 1016
gpio_test 1017
gpio_test 1018
gpio_test 1019
```

我们可以通过挂载 NFS 来运行这个 SHELL。

如果 SHELL 不能运行，可以先添加运行权限，命令如下：

```
chmod +x gpio_test.sh
```



24.2 使用 C 语言控制

大部分情况我们都需要使用 C 语言来控制外设，在 Xilinx 的 wiki 页面

<http://www.wiki.xilinx.com/GPIO%20User%20Space%20App> 我们找到一段 GPIO 测试代码，代码内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

// The specific GPIO being used must be setup and replaced thru
// this code. The GPIO of 898 is in the path of most the sys dirs
// and in the export write.
//
// Figuring out the exact GPIO was not totally obvious when there
// were multiple GPIOs in the system. One way to do is to go into
// the gpiochips in /sys/class/gpio and view the label as it should
// reflect the address of the GPIO in the system. The name of the
// the chip appears to be the 1st GPIO of the controller.
//
// The export causes the gpio898 dir to appear in /sys/class/gpio.
// Then the direction and value can be changed by writing to them.

// The performance of this is pretty good, using a nfs mount,
// running on open source linux,
// the GPIO can be toggled about every 1sec.
// The following commands from the console setup the GPIO to be
// exported, set the direction of it to an output and write a 1
// to the GPIO.
//
// bash> echo 898 > /sys/class/gpio/export
// bash> echo out > /sys/class/gpio/gpio898/direction
// bash> echo 1 > /sys/class/gpio/gpio898/value

// if sysfs is not mounted on your system, the you need to mount it
// bash> mount -t sysfs sysfs /sys
```

```
// the following bash script to toggle the gpio is also handy for
// testing
//
// while [ 1 ]; do
//   echo 1 > /sys/class/gpio/gpio898/value
//   echo 0 > /sys/class/gpio/gpio898/value
// done

// to compile this, use the following command
// gcc gpio.c -o gpio

// The kernel needs the following configuration to make this work.
//
// CONFIG_GPIO_SYSFS=y
// CONFIG_SYSFS=y
// CONFIG_EXPERIMENTAL=y
// CONFIG_GPIO_XILINX=y

int main()
{
    int valuefd, exportfd, directionfd;
    printf("GPIO test running...\n");

    // The GPIO has to be exported to be able to see it
    // in sysfs

    exportfd = open("/sys/class/gpio/export", O_WRONLY);
    if (exportfd < 0)
    {
        printf("Cannot open GPIO to export it\n");
        exit(1);
    }

    write(exportfd, "898", 4);
    close(exportfd);

    printf("GPIO exported successfully\n");

    // Update the direction of the GPIO to be an output

    directionfd = open("/sys/class/gpio/gpio898/direction", O_RDWR);
    if (directionfd < 0)
    {
        printf("Cannot open GPIO direction it\n");
        exit(1);
    }

    write(directionfd, "out", 4);
    close(directionfd);

    printf("GPIO direction set as output successfully\n");

    // Get the GPIO value ready to be toggled

    valuefd = open("/sys/class/gpio/gpio898/value", O_RDWR);
    if (valuefd < 0)
    {
        printf("Cannot open GPIO value\n");
        exit(1);
    }

    printf("GPIO value opened, now toggling...\n");

    // toggle the GPIO as fast as possible forever, a control c is needed
    // to stop it

    while (1)
    {
        write(valuefd, "1", 2);
        sleep(1);
        write(valuefd, "0", 2);
        sleep(1);
    }
}
```

这一次我们不再使用 SDK 来编译，源代码命名为 “gpio.c” ,运行下面命令编译代码

```
source /opt/Xilinx/Vivado/2017.4/settings64.sh
arm-linux-gnueabihf-gcc  gpio.c -o gpio
```

编译完成以后会生成一个 gpio 的文件，不像在 Windows , Linux 下对扩展名要求不是很严

格，gpio 文件就是一个 elf 文件。

运行 gpio，可以看到 PS 端 LED 不断闪烁，说明这个 898 就是 PS 端第一个 LED。

```
root@zynq:/mnt/linux_app/gpio# ./gpio
GPIO test running...
GPIO exported successfully
GPIO direction set as output successfully
GPIO value opened, now toggling...
```

如何确定 GPIO 的这个编号？

通过下面命令，我们可以看到有 gpiochip898 gpiochip1016 gpiochip1020，说明有三个 GPIO 控制器，数字是控制器 GPIO 基数。

```
ls /sys/class/gpio
```

```
root@zynq:/mnt/linux_app/gpio# ls /sys/class/gpio
export gpio898 gpiochip1016 gpiochip1020 gpiochip898 unexport
root@zynq:/mnt/linux_app/gpio#
```

如何确定和物理 GPIO 的关系？

通过下面命令，来确定 GPIO1016 和物理 GPIO 的关系，可以看到这个 gpio 在设备树里的节点是 “/amba_pl/gpio@41210000”，通过设备树的节点我们可以确定是哪一个物理 GPIO。

```
cat /sys/class/gpio/gpiochip1016/label
```

```
root@zynq:/sys/class/gpio/gpiochip1016# cat /sys/class/gpio/gpiochip1016/label
/amba_pl/gpio@41210000
root@zynq:/sys/class/gpio/gpiochip1016#
```

24.3 实验总结

本实验重点在于如何通过 Xilinx 给的资料来学习 ZYNQ，技术资料更新较快，只有紧跟芯片厂商提供的最新资料才能获取到最新最好的技术。后续教程中 PCIe 驱动、PL 端以太网驱动都是 Xilinx 提供，这些资料都可以通过 wiki 获取。

如果使用一个非 xilinx 的 IP，或者自己写的 IP，那就要自己开发驱动程序，这对没做过 Linux 驱动的开发人员来说是一个挑战，所以我们尽可能使用 Xilinx 的 IP 来搭建系统，优点就是不用

开发 Linux 驱动，缺点就是不够灵活，如果 IP 有问题或者驱动有问题，无法快速定位问题。

第二十五章 Petalinux 下的 HDMI 显示

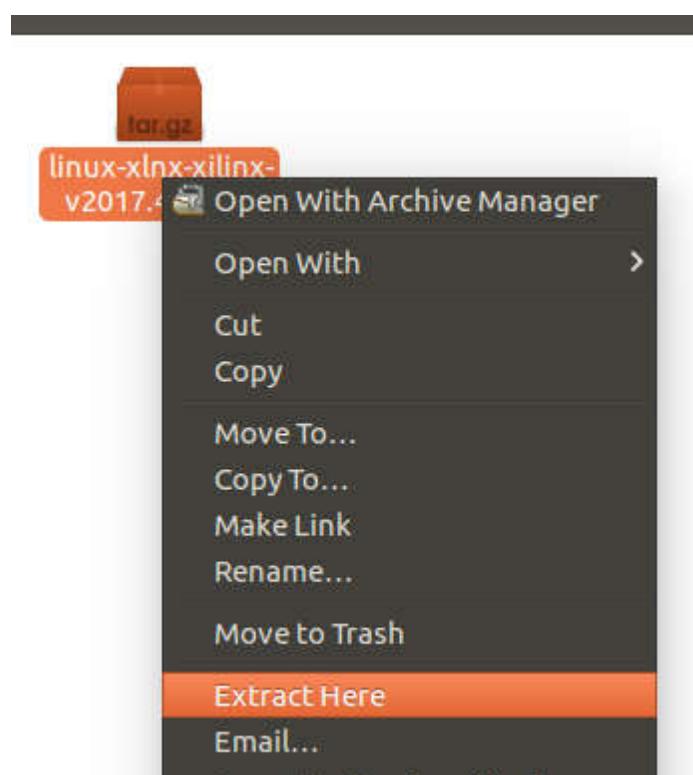
前面的教程中我们已经体验了使用 Petalinux 开发嵌入式 Linux 系统，但我们使用的功能仅仅是 Petalinux 冰山一角，Petalinux 的强大功能需要我们不断的探索，本实验讲解如何使用一个自己的内核来做 Linux，这样我们可以在内核里添加很多驱动，例如 HDMI 显示。

开发板使用的 HDMI 接口芯片是 ADV7511，芯驿电子将 Xilinx 提供的内核中加入了 HDMI 接口芯片的驱动。

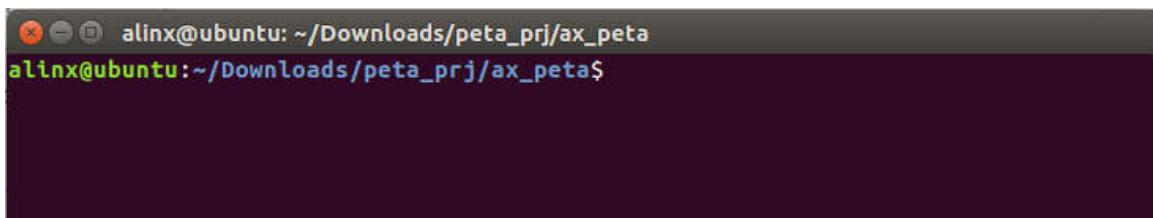
25.1 Petalinux 配置

本实验还是在前面实验的 Petalinux 工程修改，需要先掌握前面的实验内容。

- 1) 把内核源文件复制到 Linux 主机，然后解压，解压后的内核目录就是本实验 Petalinux 要用到的内核。



- 2) 打开终端，进入前面实验中的 Petalinux 工程目录



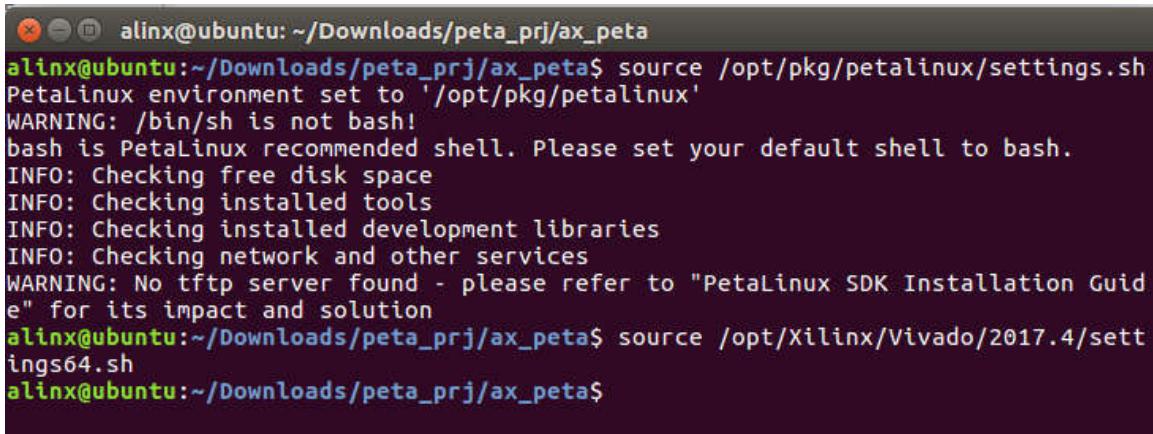
```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

3) 设置 petalinux 环境变量，运行下面命令

```
source /opt/pkg/petalinux/settings.sh
```

4) 运行下面命令设置 vivado 环境变量

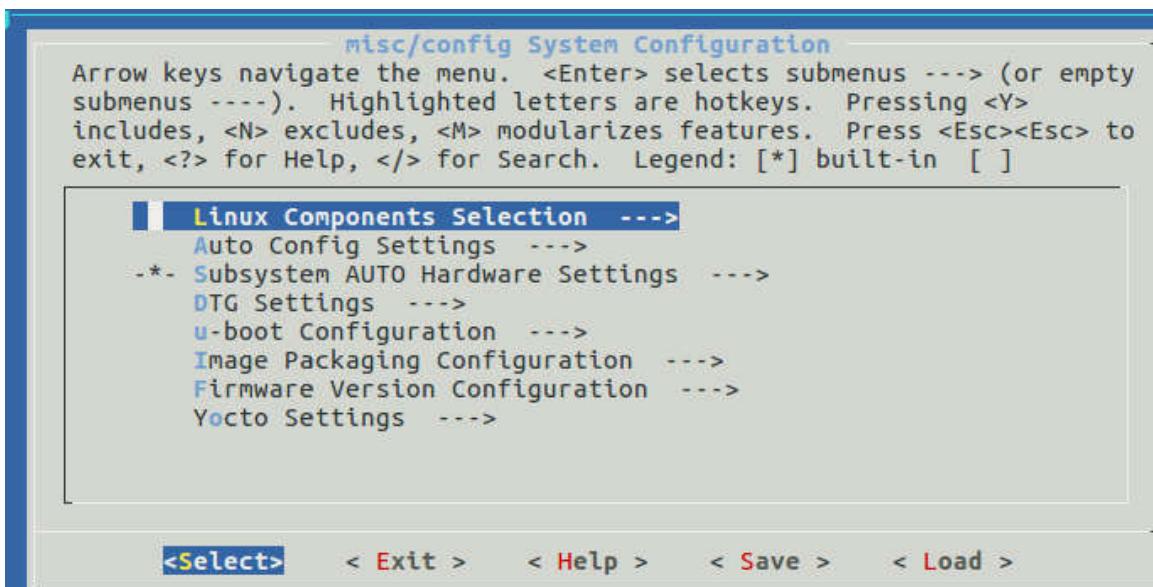
```
source /opt/Xilinx/Vivado/2017.4/settings64.sh
```



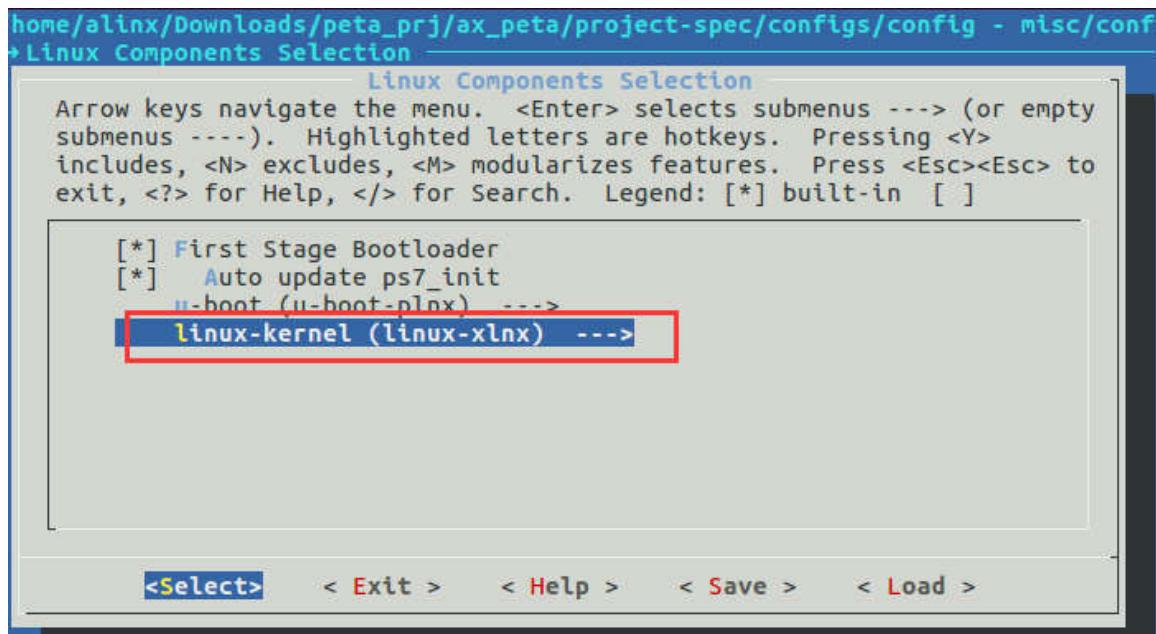
```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ source /opt/pkg/petalinux/settings.sh
PetaLinux environment set to '/opt/pkg/petalinux'
WARNING: /bin/sh is not bash!
bash is PetaLinux recommended shell. Please set your default shell to bash.
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
WARNING: No tftp server found - please refer to "PetaLinux SDK Installation Guide" for its impact and solution
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ source /opt/Xilinx/Vivado/2017.4/settings64.sh
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

5) 使用下面命令重新配置 petalinux

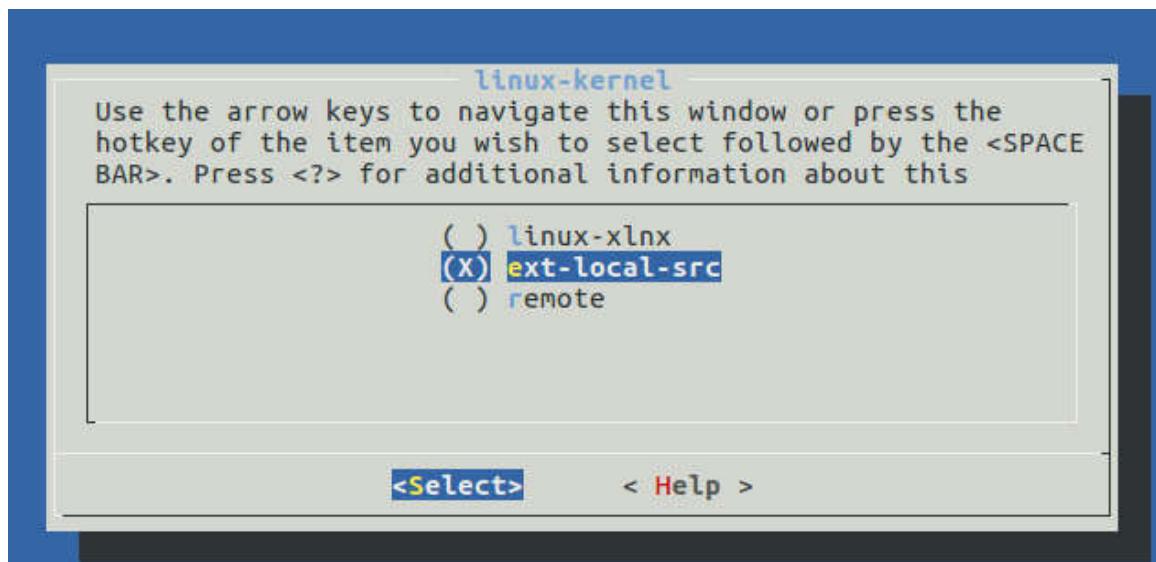
```
petalinux-config
```



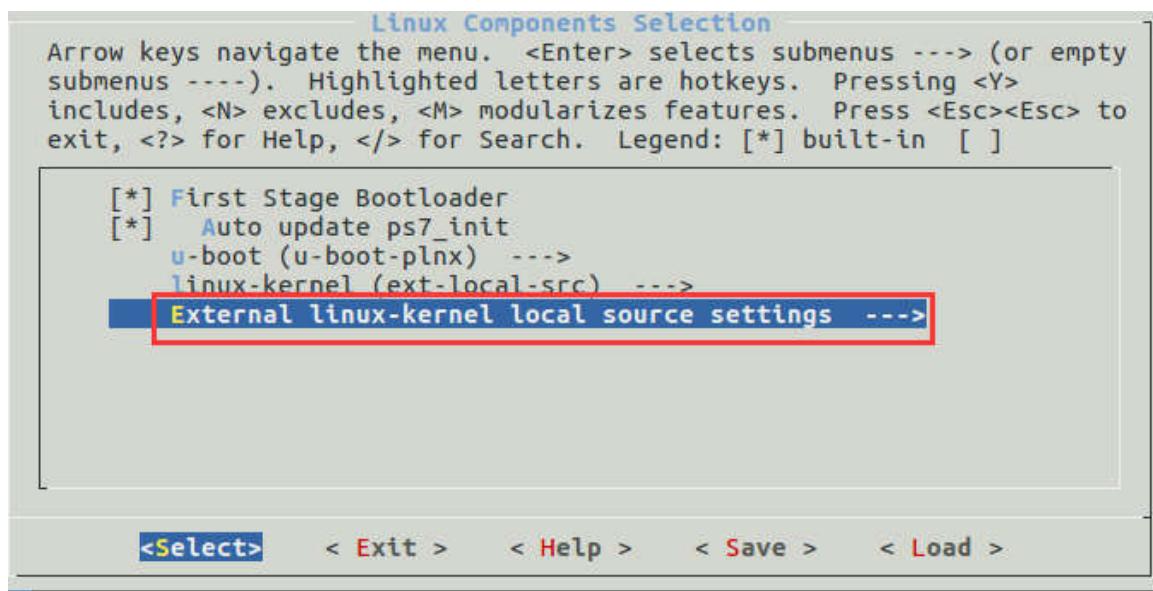
- 6) 选择 Linux Components Selection ---> linux-kernel (linux-xlnx) --->



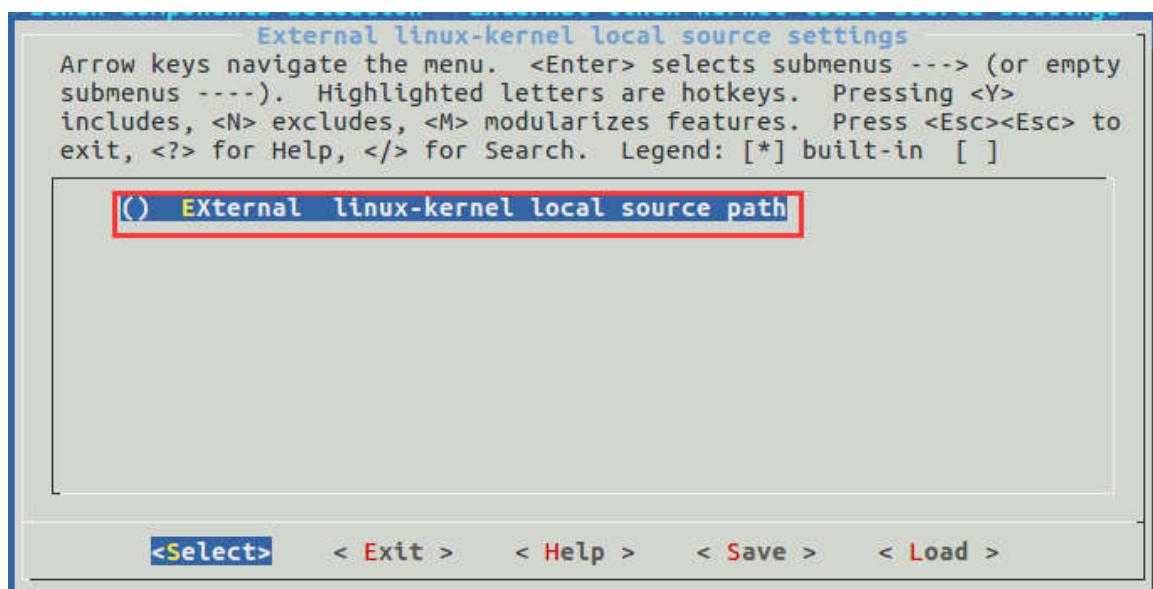
- 7) 选择 ext-local-src 按空格键



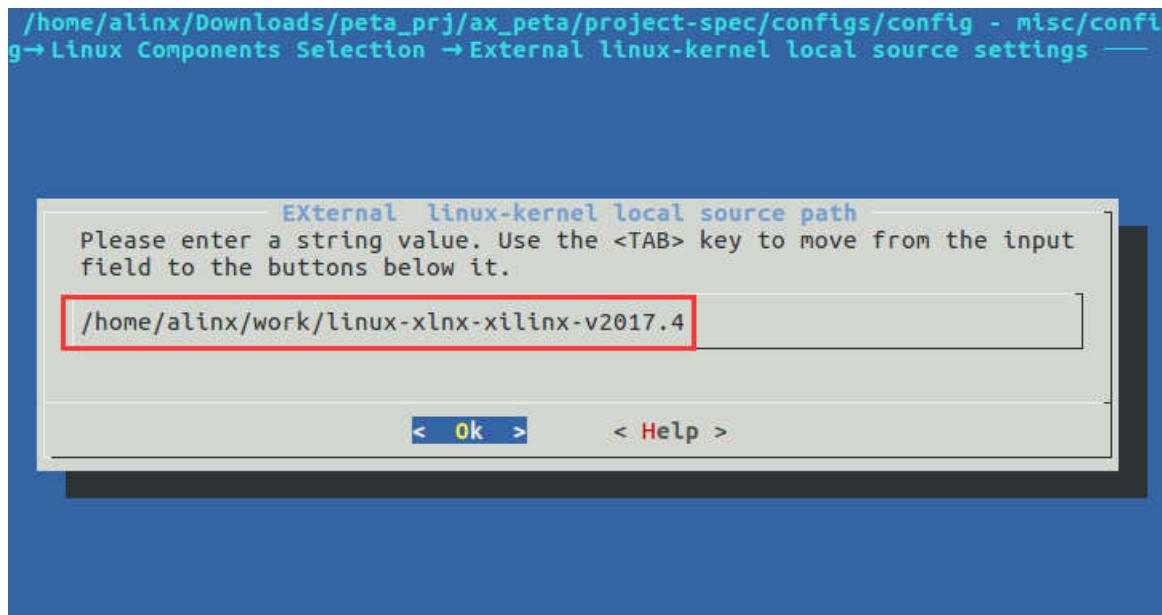
- 8) 选择 External linux-kernel local source settings --->



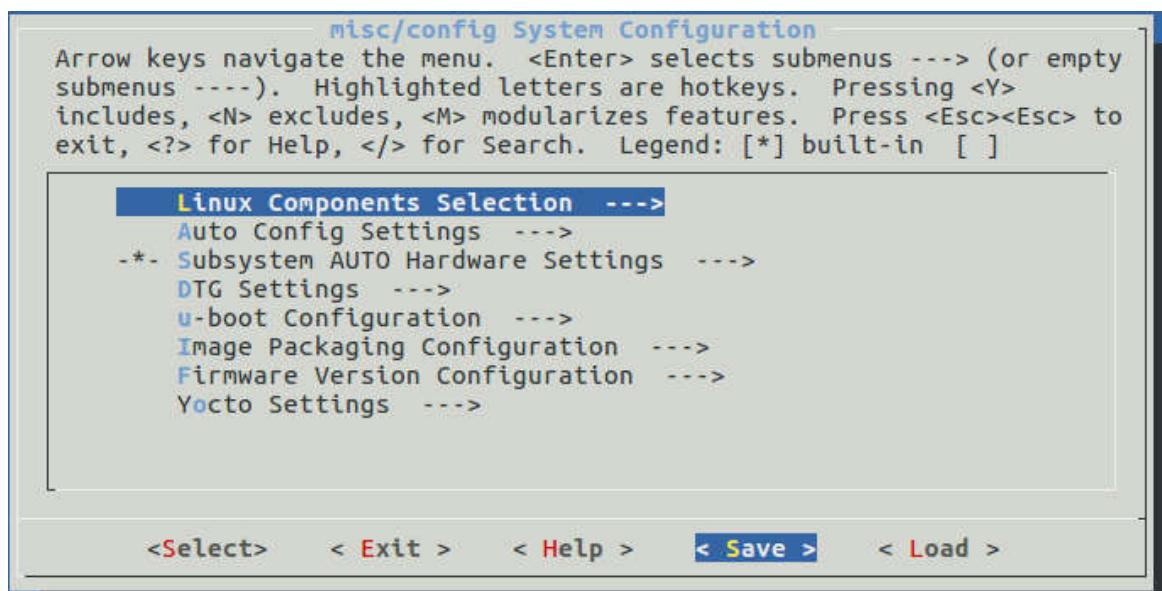
选择 External linux-kernel local source path



- 9) 填上 Linux 内核源码的路径 /home/ilinx/work/linux-xlnx-xilinx-v2017.4



10) 然后保存退出



25.2 配置 Linux 内核

1) 运行下面的命令配置内核

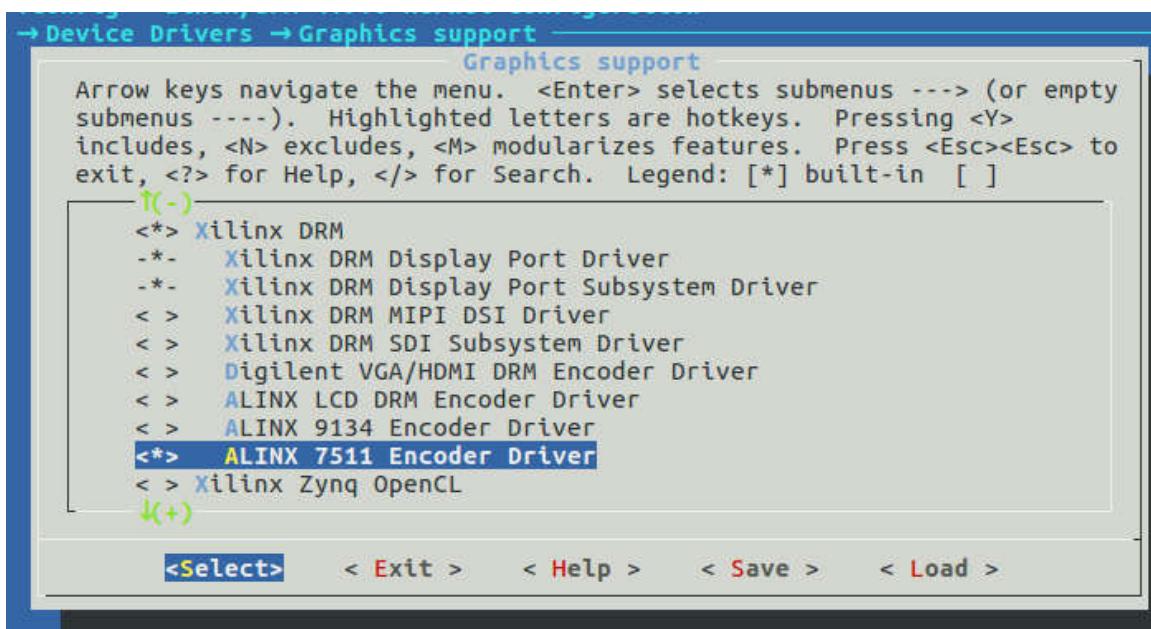
```
petalinux-config -c kernel
```

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
*** Execute 'make' to start the build or try 'make help'.

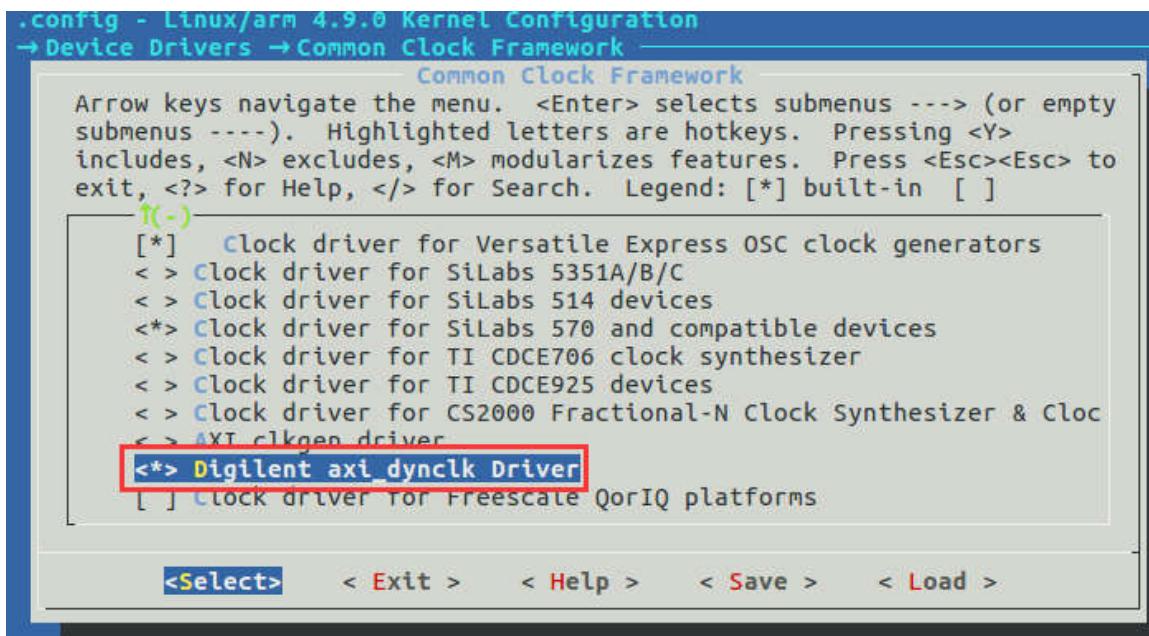
[INFO] sourcing bitbake
[INFO] generating plnxtool conf
[INFO] generating meta-plnx-generated layer
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_peta
~/Downloads/peta_prj/ax_peta
[INFO] generating machine configuration
[INFO] generating bbappends for project . This may take time !
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_peta
~/Downloads/peta_prj/ax_peta
[INFO] generating u-boot configuration files

[INFO] generating kernel configuration files
[INFO] generating kconfig for Rootfs
Generate rootfs kconfig
[INFO] oldconfig rootfs
[INFO] generating petalinux-user-image.bb
[INFO] successfully configured project
webtalk failed:Petalinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get Petalinux usage statistics!
alink@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-config -c kernel
```

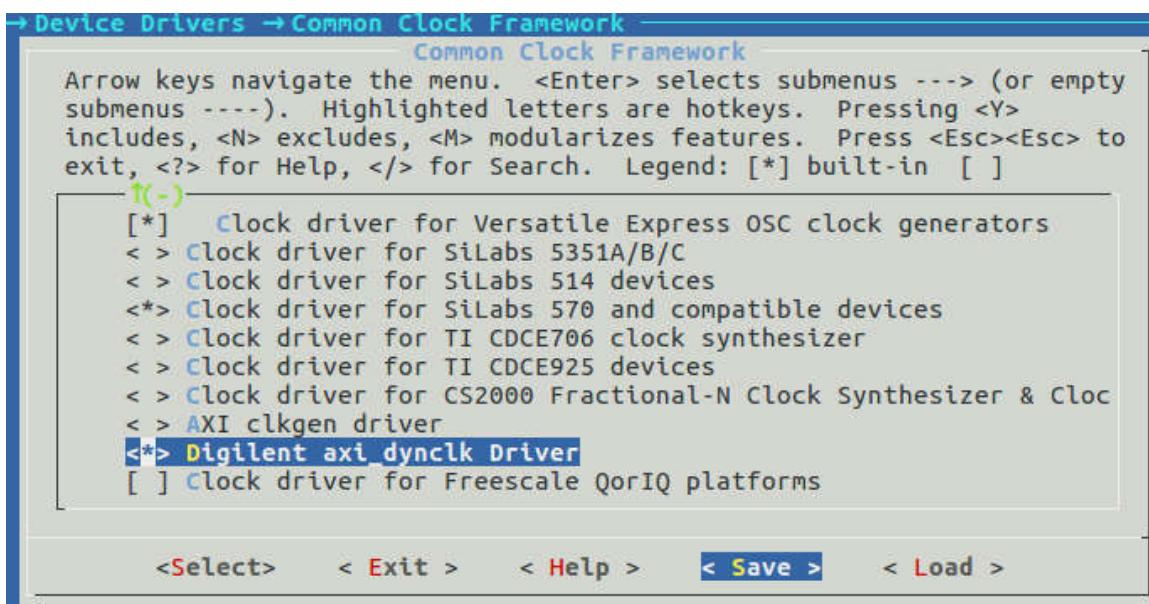
- 2) 在弹出的窗口中，进入 Device Drivers → Graphics support，选择 ALINX 7511 Encoder Driver 项按 y。



- 3) 在 Device Drivers → Common Clock Framework 选项中选择 Digilent axi_dynclk Driver 按 y



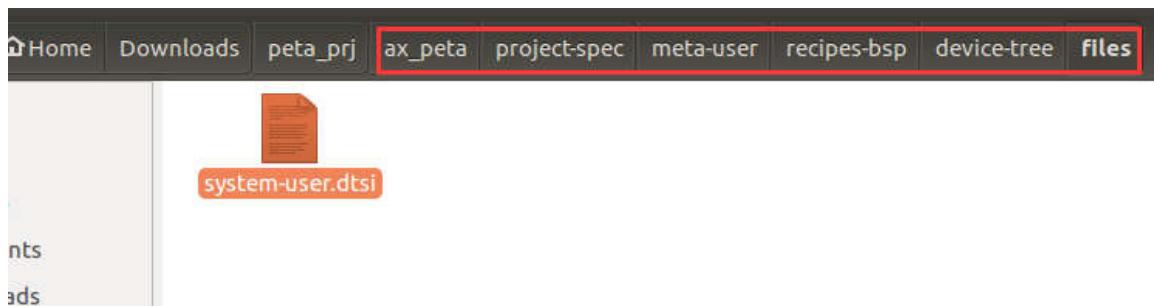
4) 保存并退出



25.3 修改设备树

设备树是描述设备信息的一种格式化文本，每个驱动要求的设备树节点也是不同的，对于没接触过设备的开发人员，需要慢慢熟悉。

1) 打开 petalinux 工程文件中的名为 system-user.dtsi 的文件



2) 修改设备树内容如下

```
/include/ "system-conf.dtsi"

/ {
    model = "Zynq ALINX AX7350 Development Board";
    compatible = "alinx,zynq-ax7350", "xlnx,zynq-7000";
    aliases {
        ethernet0 = "&gem0";
        ethernet1 = "&xaxi_ethernet_0";
        serial0 = "&uart1";
    };

    usb_phy0: usb_phy@0 {
        compatible = "ulpi-phy";
        #phy-cells = <0>;
        reg = <0xe0002000 0x1000>;
        view-port = <0x0170>;
        drv-vbus;
    };

};

&i2c1 {
    clock-frequency = <100000>;
};

&usb0 {
    usb-phy = <&usb_phy0>;
};

&sdhci0 {
    u-boot,dm-pre-reloc;
};

&uart1 {
    u-boot,dm-pre-reloc;
};

&flash0 {
    compatible = "micron,m25p80", "w25q256", "spi-flash";
};
```

```
&gem0 {
    phy-handle = <&ethernet_phy>;
    ethernet_phy: ethernet-phy@1 {
        reg = <1>;
        device_type = "ethernet-phy";
    };
};

&axi_ethernet_0 {
    local-mac-address = [00 0a 35 00 01 22];
    phy-handle = <&phy1>;
    xlnx,has-mdio = <0x1>;
    phy-mode = "rgmii";
    mdio {
        phy1: phy@1 {
            device_type = "ethernet-phy";
            reg = <1>;
        };
    };
};

&amba_pl {

    ax_encoder_0:ax_7511_encoder {
        compatible = "ax_7511,drm-encoder";
        ax_7511,edid-i2c = <&i2c1>;
    };

    xilinx_drm {
        compatible = "xlnx,drm";
        xlnx,vtc = <&v_tc_0>;
        xlnx,connector-type = "HDMI-A";
        xlnx,encoder-slave = <&ax_encoder_0>;
        clocks = <&axi_dynclk_0>;
        planes {
            xlnx,pixel-format = "rgb888";
            plane0 {
                dmas = <&axi_vdma_0 0>;
                dma-names = "dma";
            };
        };
    };
};

&axi_dynclk_0 {
    compatible = "digilent,axi-dynclk";
    #clock-cells = <0>;
    clocks = <&clkc 15>;
};

&v_tc_0 {
    compatible = "xlnx,v-tc-5.01.a";
};
```

```
system-user.dtsi
/include/ "system-conf.dtsi"

/ {
    model = "Zynq ALINX AX7350 Development Board";
    compatible = "alinx,zynq-ax7350", "xlnx,zynq-7000";
    aliases {
        ethernet0 = "&gem0";
        ethernet1 = "&axi_ethernet_0";
        serial0 = "&uart1";
    };
    usb_phy0: usb_phy@0 {
        compatible = "ulpi-phy";
        #phy-cells = <0>;
        reg = <0xe0002000 0x1000>;
        view-port = <0x0170>;
        drv-vbus;
    };
};

&i2c1 {
    clock-frequency = <100000>;
};

&usb0 {
    usb-phy = <&usb_phy0>;
};

&sdhci0 {
    u-boot,dm-pre-reloc;
};

&uart1 {
    u-boot,dm-pre-reloc;
};

&flash0 {
    compatible = "micron,m25p80", "w25q256", "spi-flash";
};

&gem0 {
    phy-handle = <&ethernet phy>;
```

25.4 编译测试 Petalinux 工程

- 1) 使用下面命令配置编译 uboot、内核、根文件系统、设备树等。

```
petalinux-build
```

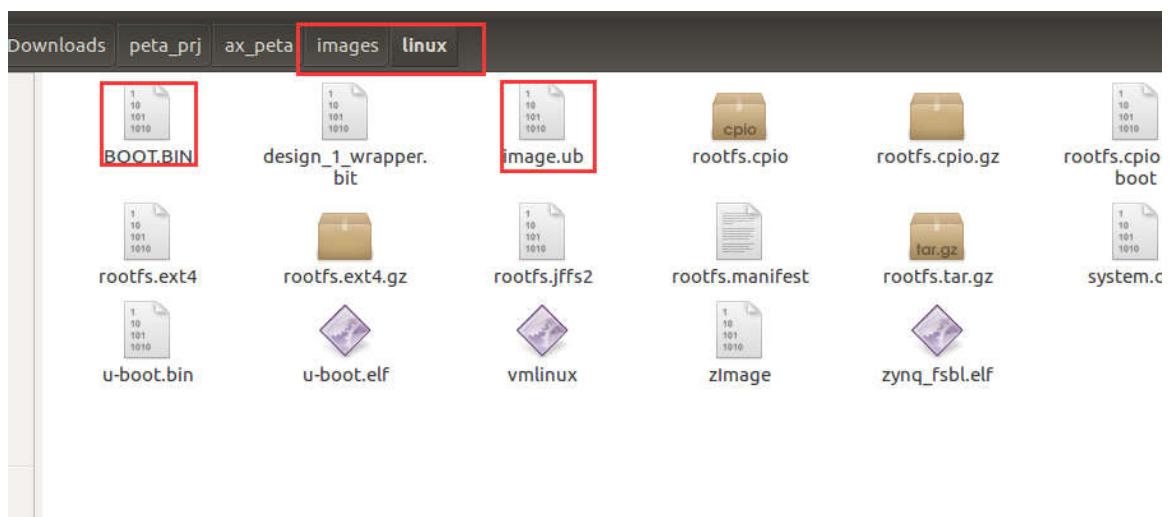
```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
Loading cache: 100% |#####
Loaded 3256 entries from dependency cache.
Parsing recipes: 100% |#####
Parsing of 2466 .bb files complete (2433 cached, 33 parsed). 3259 targets, 226 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#####
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 2 tasks of which 0 didn't need to be rerun and all succeeded.
[INFO] successfully configured kernel
webtalk failed:PetaLinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-build
[INFO] building project
[INFO] sourcing bitbake
INFO: bitbake petalinux-user-image
Loading cache: 100% |#####
Loaded 3256 entries from dependency cache.
Parsing recipes: 100% |#####
Parsing of 2466 .bb files complete (2433 cached, 33 parsed). 3259 targets, 226 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 18% |##### | ETA: 0:00:08
```

- 2) 运行下面命令生成 BOOT 文件，注意空格和短线

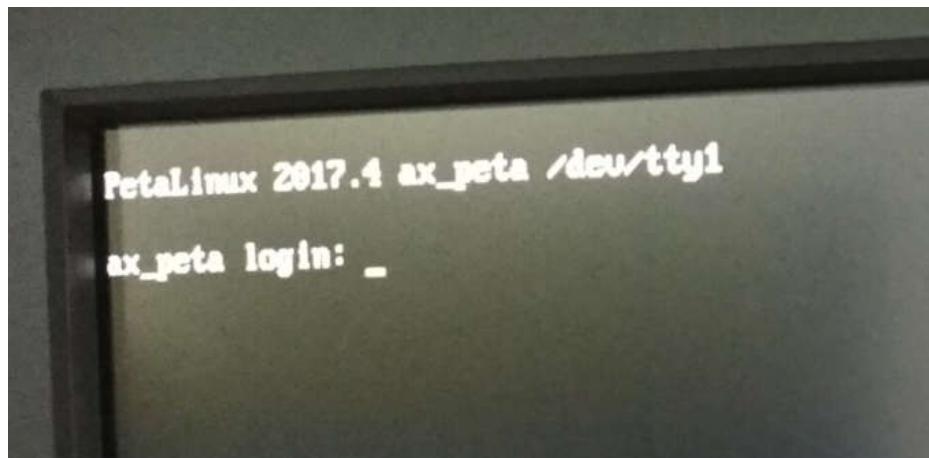
```
petalinux-package --boot --fsbl ./images/linux/zynq_fsbl.elf
--fpga ./images/linux/design_1_wrapper.bit --uboot --force
```

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
from external source tree /home/alinx/work/linux-xlnx-xilinx-v2017.4
NOTE: Tasks Summary: Attempted 2439 tasks of which 2367 didn't need to be rerun and all succeeded.
INFO: Copying Images from deploy to images
NOTE: Failed to copy built images to tftp dir: /tftpboot
[INFO] successfully built project
webtalk failed:PetaLinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-package --boot --fsbl ./images/linux/zynq_fsbl.elf --fpga ./images/linux/design_1_wrapper.bit --uboot --force
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/zynq_fsbl.elf"
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/design_1_wrapper.bit"
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/u-boot.elf"
INFO: Generating zynq binary package BOOT.BIN...
INFO: Binary is ready.
WARNING: Unable to access the TFTPBOOT folder /tftpboot!!!
WARNING: Skip file copy to TFTPBOOT folder!!!
webtalk failed:Invalid tool in the statistics file:petalinux-yocto!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

- 3) 把 BOOT.bin 和 iamge.ub 复制到 sd 中，设置开发板 sd 模式启动，插上 HDMI 显示器，启动开发板。



4) 显示器会显示出如下内容



第二十六章 使用 Debian 8 桌面系统

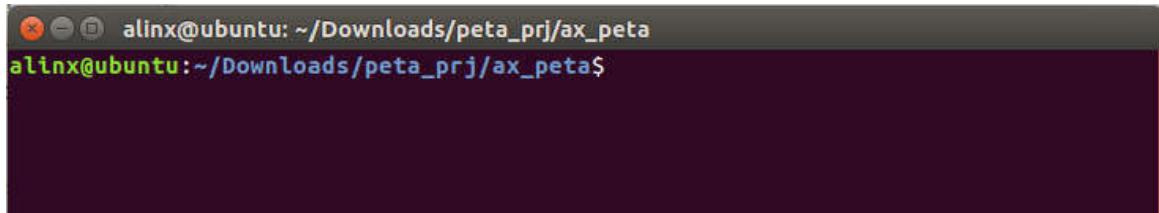
前面的教程中我们学习了如何使用 Petalinux 定制一个嵌入式 Linux 系统以及 HDMI 显示，本实验将做一个基于 Debian 8 的根文件系统，根文件系统的制作比较复杂，本章不再讲解，直接使用制作好的 Debian 8 根文件系统。

26.1 Petalinux 配置

由于 Debian 8 根文件系统比较大，QSPI flash 是无法放下的，只能放在 sd 卡或 emmc 里，所以我们要配置 Petalinux。

本实验还是在前面实验的 Petalinux 工程修改，需要先掌握前面的实验内容。

- 1) 打开终端，进入前面实验中的 Petalinux 工程目录



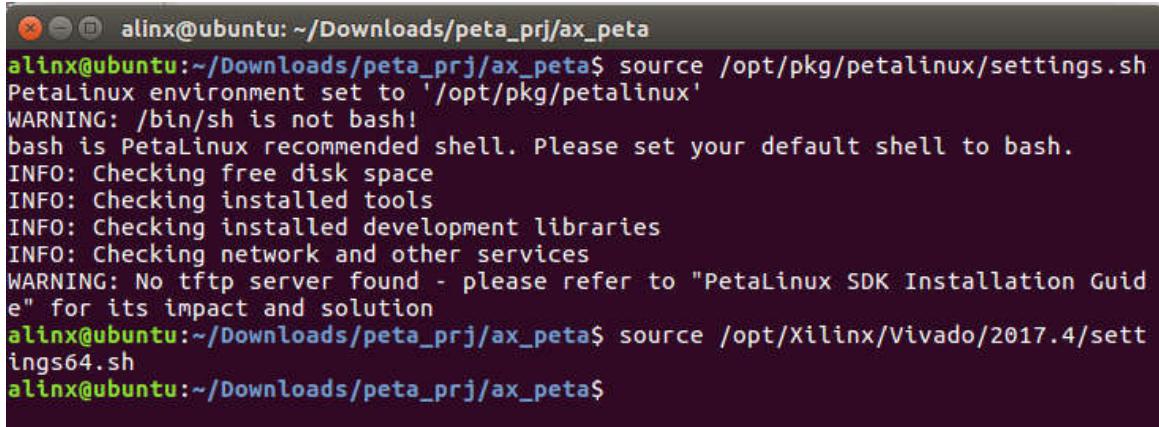
```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

- 2) 设置 petalinux 环境变量，运行下面命令

```
source /opt/pkg/petalinux/settings.sh
```

- 3) 运行下面命令设置 vivado 环境变量

```
source /opt/Xilinx/Vivado/2017.4/settings64.sh
```

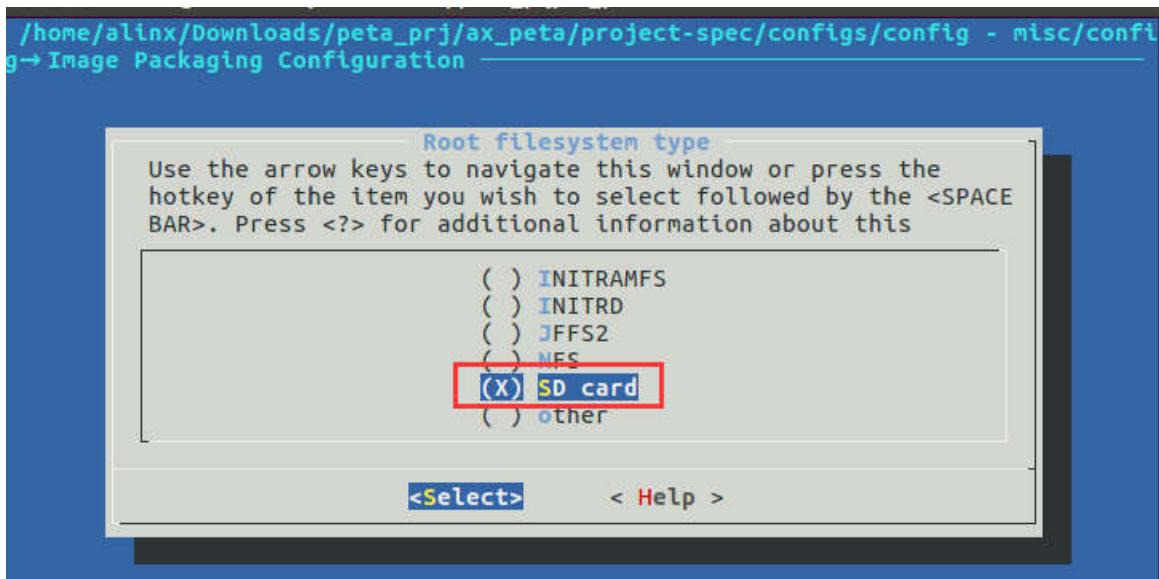


```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ source /opt/pkg/petalinux/settings.sh
PetaLinux environment set to '/opt/pkg/petalinux'
WARNING: /bin/sh is not bash!
bash is PetaLinux recommended shell. Please set your default shell to bash.
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
WARNING: No tftp server found - please refer to "PetaLinux SDK Installation Guide" for its impact and solution
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ source /opt/Xilinx/Vivado/2017.4/settings64.sh
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

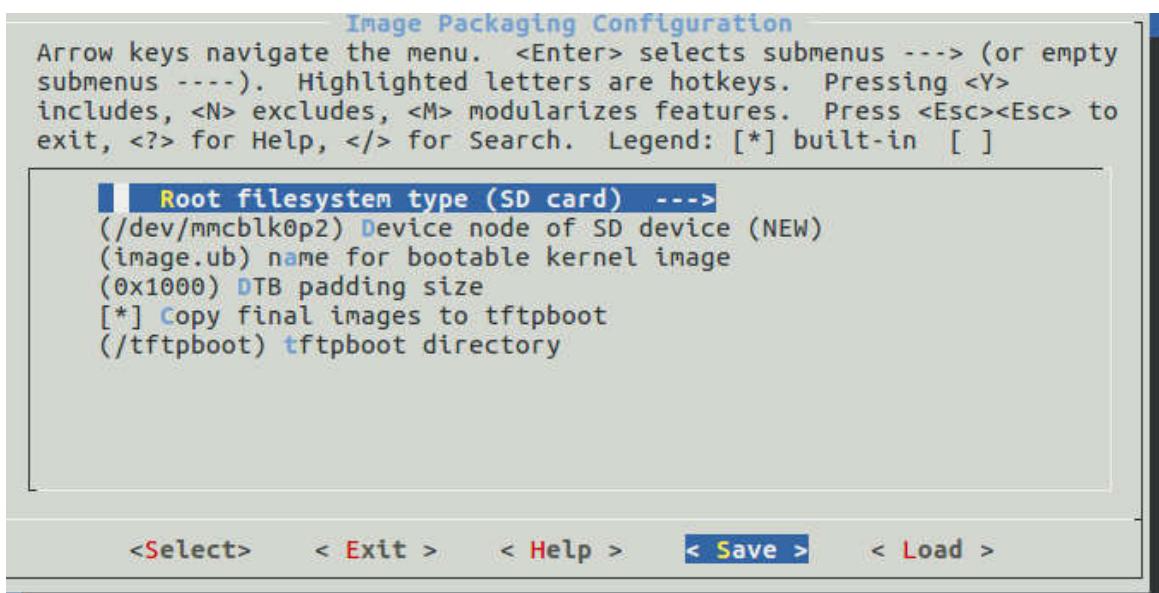
- 4) 使用下面命令重新配置 petalinux

petalinux-config

- 5) 在 Image Packaging Configuration --> Root filesystem type 选项中选择 SD card , 把根文件系统放在 SD 卡中。



- 6) 保持并退出



26.2 配置 Linux 内核

前面的教程中配置 HDMI 输出相关的驱动，这里继续配置一些其他驱动。

- 1) 运行下面的命令配置内核

```
petalinux-config -c kernel
```

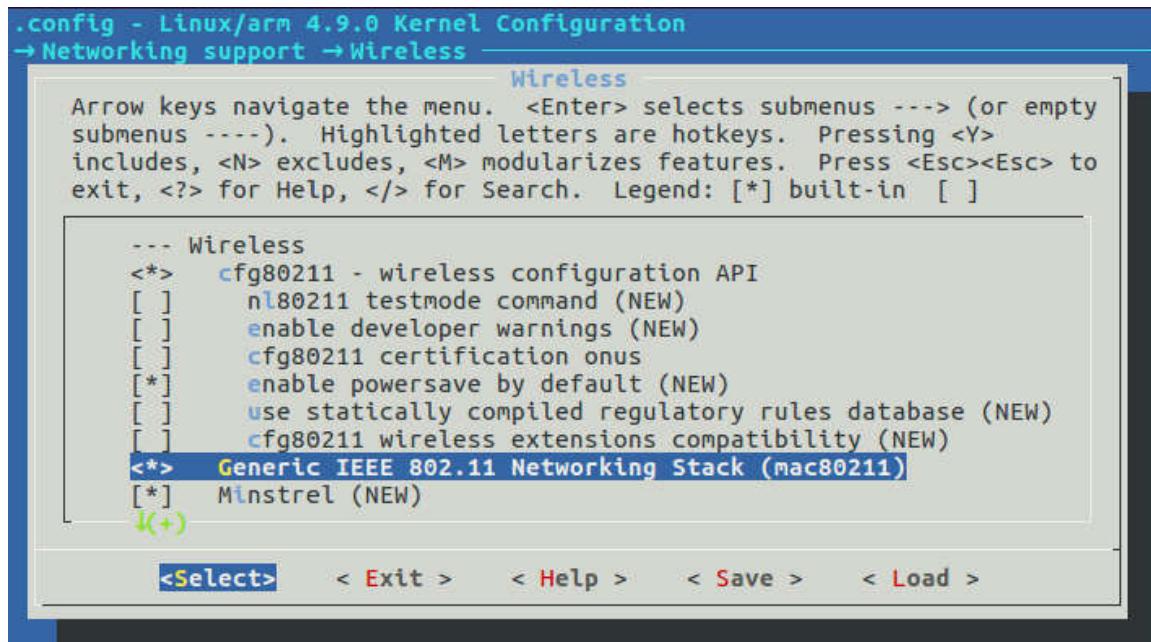
```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
*** Execute 'make' to start the build or try 'make help'.

[INFO] sourcing bitbake
[INFO] generating plnxtool conf
[INFO] generating meta-plnx-generated layer
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_peta
~/Downloads/peta_prj/ax_peta
[INFO] generating machine configuration
[INFO] generating bbappends for project . This may take time !
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_peta
~/Downloads/peta_prj/ax_peta
[INFO] generating u-boot configuration files

[INFO] generating kernel configuration files
[INFO] generating kconfig for Rootfs
Generate rootfs kconfig
[INFO] oldconfig rootfs
[INFO] generating petalinux-user-image.bb
[INFO] successfully configured project
webtalk failed:Petalinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get Petalinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-config -c kernel
```

26.2.1 配置 USB WIFI 模块驱动

- 在选项 Networking Support -> Wireless -> 选项中先选择 cfg80211 - wireless configuration API , 再使能 Generic IEEE 802.11 Networking Stack (mac80211)



- 在选项 Device Drivers -> Network device support -> Wireless LAN -> Realtek rtlwifi family of devices 选择 Realtek RTL8192CU/RTL8188CU USB Wireless Network Adapter

```
.config - Linux/arm 4.9.0 Kernel Configuration
[...] twork device support → Wireless LAN → Realtek rtlwifi family of devices
    Realtek rtlwifi family of devices
        Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
        submenus ----). Highlighted letters are hotkeys. Pressing <Y>
        includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to
        exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]
    --- Realtek rtlwifi family of devices
    <-> Realtek RTL8192CE/RTL8188CE Wireless Network Adapter (NEW)
    <-> Realtek RTL8192SE/RTL8191SE PCIe Wireless Network Adapter (
    <-> Realtek RTL8192DE/RTL8188DE PCIe Wireless Network Adapter (
    <-> Realtek RTL8723AE PCIe Wireless Network Adapter (NEW)
    <-> Realtek RTL8723BE PCIe Wireless Network Adapter (NEW)
    <-> Realtek RTL8188EE Wireless Network Adapter (NEW)
    <-> Realtek RTL8192EE Wireless Network Adapter (NEW)
    <-> Realtek RTL8821AE/RTL8812AE Wireless Network Adapter (NEW)
<*> Realtek RTL8192CU/RTL8188CU USB Wireless Network Adapter
↓(+)

<Select> < Exit > < Help > < Save > < Load >
```

26.2.2 配置 USB 摄像头驱动

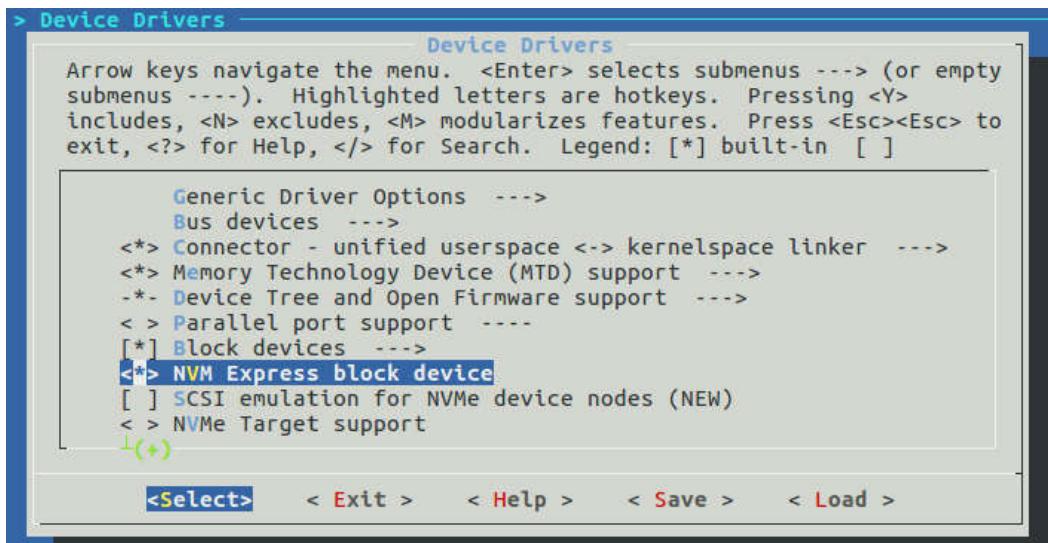
- 在选项 Device Drivers ---> Multimedia support ---> Media USB Adapters ---> 中使能 USB Video Class (UVC)

```
.config - Linux/arm 4.9.0 Kernel Configuration
→ Device Drivers → Multimedia support → Media USB Adapters
    Media USB Adapters
        Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
        submenus ----). Highlighted letters are hotkeys. Pressing <Y>
        includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to
        exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]
    --- Media USB Adapters
    *** Webcam devices ***
    <*> USB Video Class (UVC)
    [*]   UVC input events device support (NEW)
    <M>   GSPCA based webcams (NEW) --->
    <->   USB Philips Cameras (NEW)
    <->   CPiA2 Video For Linux (NEW)
    <->   USB ZR364XX Camera support (NEW)
    <->   USB Syntek DC1125 Camera support (NEW)
    <->   USB Sensoray 2255 video capture device (NEW)
↓(+)

<Select> < Exit > < Help > < Save > < Load >
```

26.2.3 配置 PCIe NVMe SSD 驱动

- PCIe 驱动默认已经配置好了，这里不需要再配置，只需配置 SSD 驱动
- 在 Device Drivers 中选择<*> NVM Express block device



- 3) 保持并退出

26.3 编译测试 Petalinux 工程

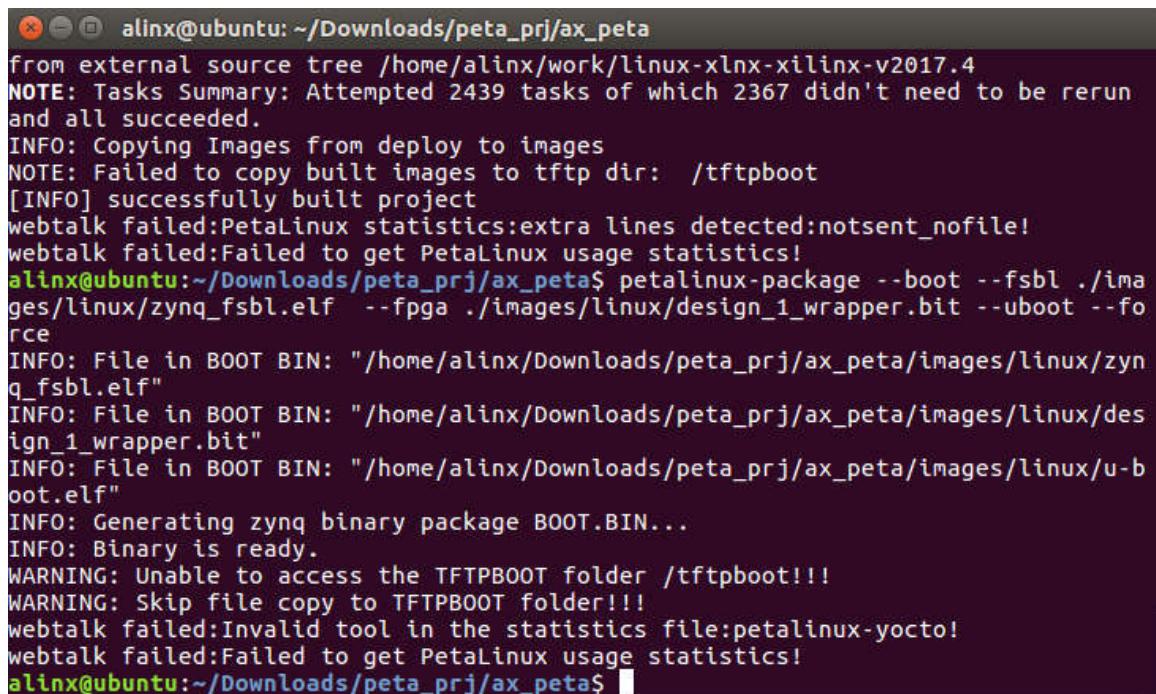
- 1) 使用下面命令配置编译 uboot、内核、根文件系统、设备树等。

```
petalinux-build
```

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
Loading cache: 100% [########################################] Time: 0:00:00
Loaded 3256 entries from dependency cache.
Parsing recipes: 100% [########################################] Time: 0:00:03
Parsing of 2466 .bb files complete (2433 cached, 33 parsed). 3259 targets, 226 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% [########################################] Time: 0:00:04
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 2 tasks of which 0 didn't need to be rerun and all succeeded.
[INFO] successfully configured kernel
webtalk failed:Petalinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get Petalinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-build
[INFO] building project
[INFO] sourcing bitbake
INFO: bitbake petalinux-user-image
Loading cache: 100% [########################################] Time: 0:00:01
Loaded 3256 entries from dependency cache.
Parsing recipes: 100% [########################################] Time: 0:00:03
Parsing of 2466 .bb files complete (2433 cached, 33 parsed). 3259 targets, 226 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 18% [##] | ETA: 0:00:08
```

- 2) 运行下面命令生成 BOOT 文件，注意空格和短线

```
petalinux-package --boot --fsbl ./images/linux/zynq_fsbl.elf
--fpga ./images/linux/design_1_wrapper.bit --u-boot --force
```



```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
from external source tree /home/alinx/work/linux-xlnx-xilinx-v2017.4
NOTE: Tasks Summary: Attempted 2439 tasks of which 2367 didn't need to be rerun
and all succeeded.
INFO: Copying Images from deploy to images
NOTE: Failed to copy built images to tftp dir: /tftpboot
[INFO] successfully built project
webtalk failed:PetaLinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-package --boot --fsbl ./images/linux/zynq_fsbl.elf --fpga ./images/linux/design_1_wrapper.bit --u-boot --force
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/zynq_fsbl.elf"
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/design_1_wrapper.bit"
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/u-boot.elf"
INFO: Generating zynq binary package BOOT.BIN...
INFO: Binary is ready.
WARNING: Unable to access the TFTPBOOT folder /tftpboot!!!
WARNING: Skip file copy to TFTPBOOT folder!!!
webtalk failed:Invalid tool in the statistics file:petalinux-yocto!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

26.4 制作 SD 卡文件系统

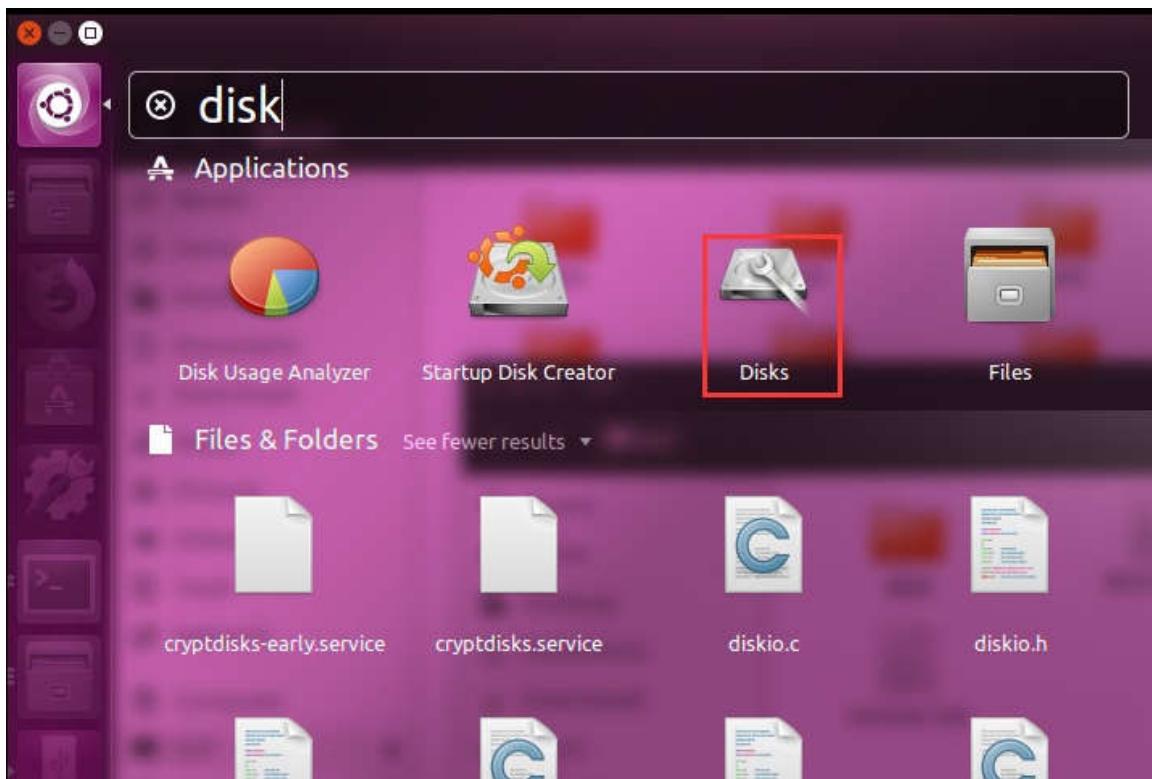
制作 SD 卡文件系统会导致 SD 卡里内容丢失，请先做好备份。

26.4.1 SD 卡修改分区

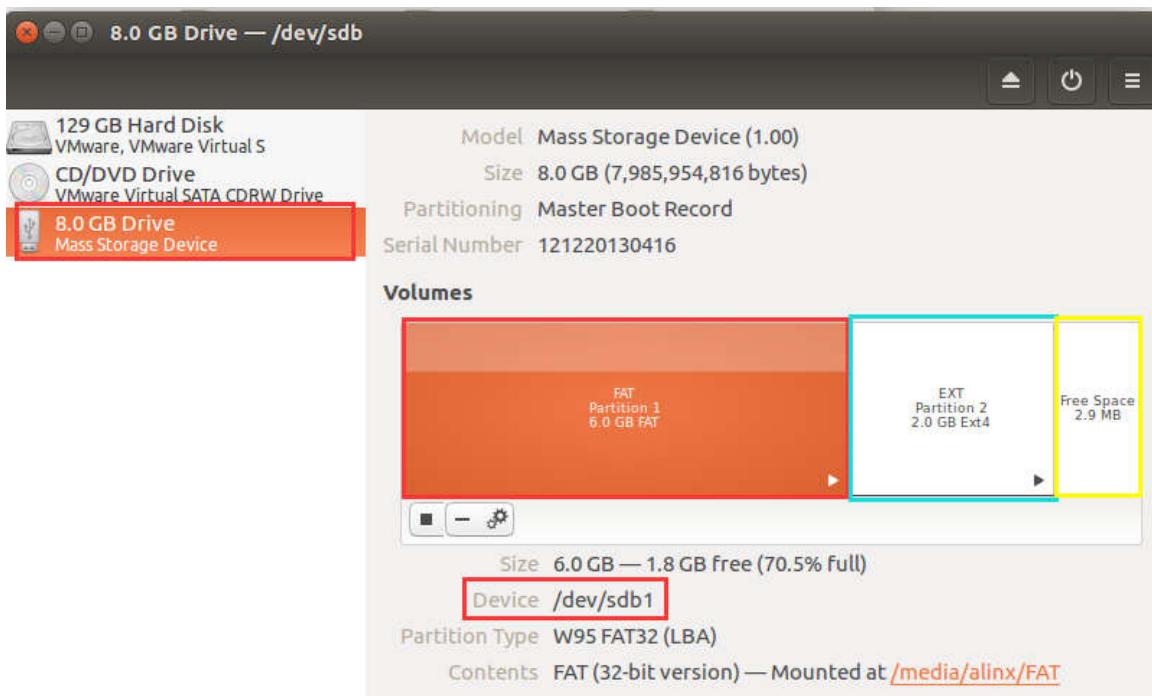
- 1) 把开发板的 sd 卡插入读卡器，然后插入电脑的 USB 口
- 2) 连接到虚拟机 Linux 中



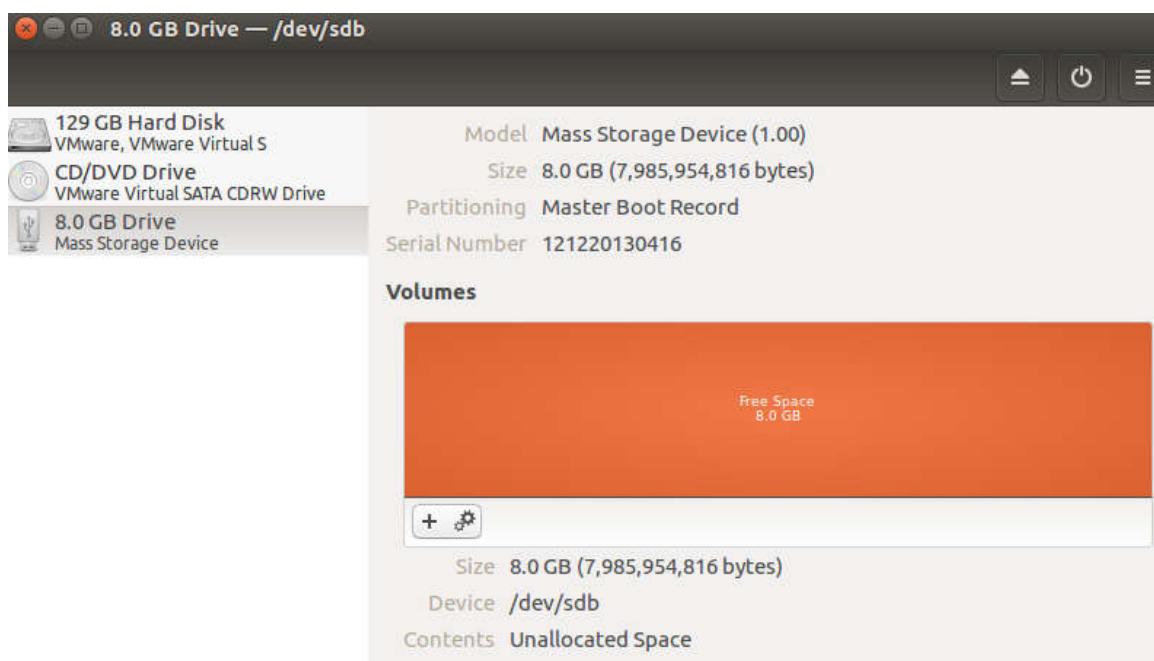
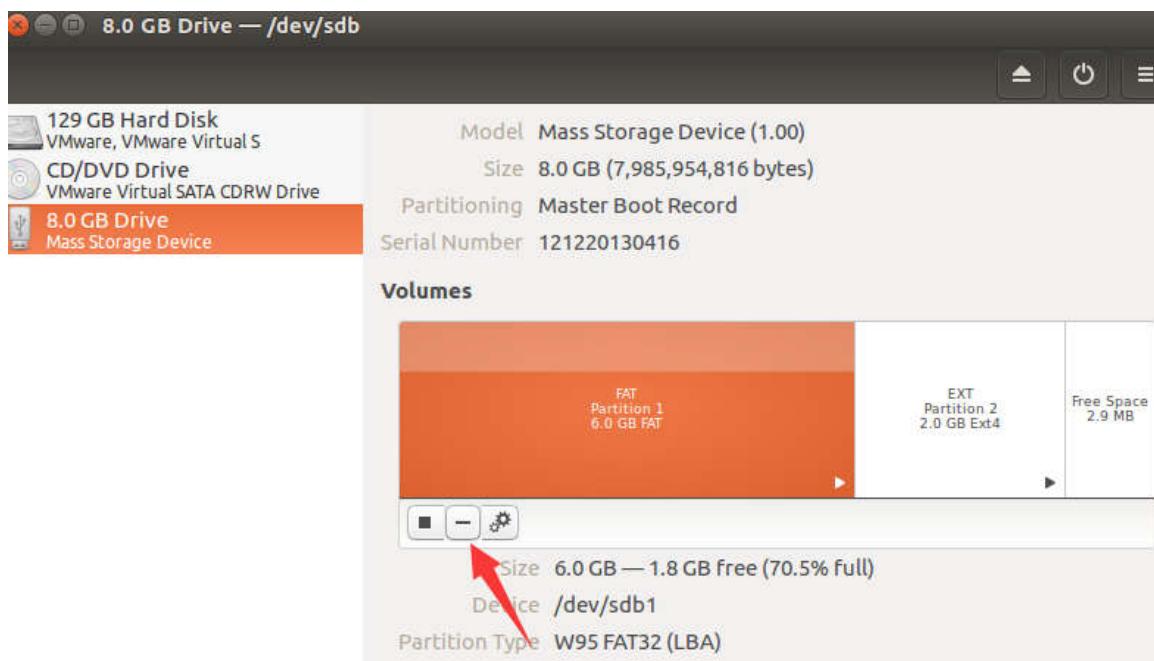
- 3) 在 ubuntu 的搜索路径中，输入 disk，会出现 Disks 的图标



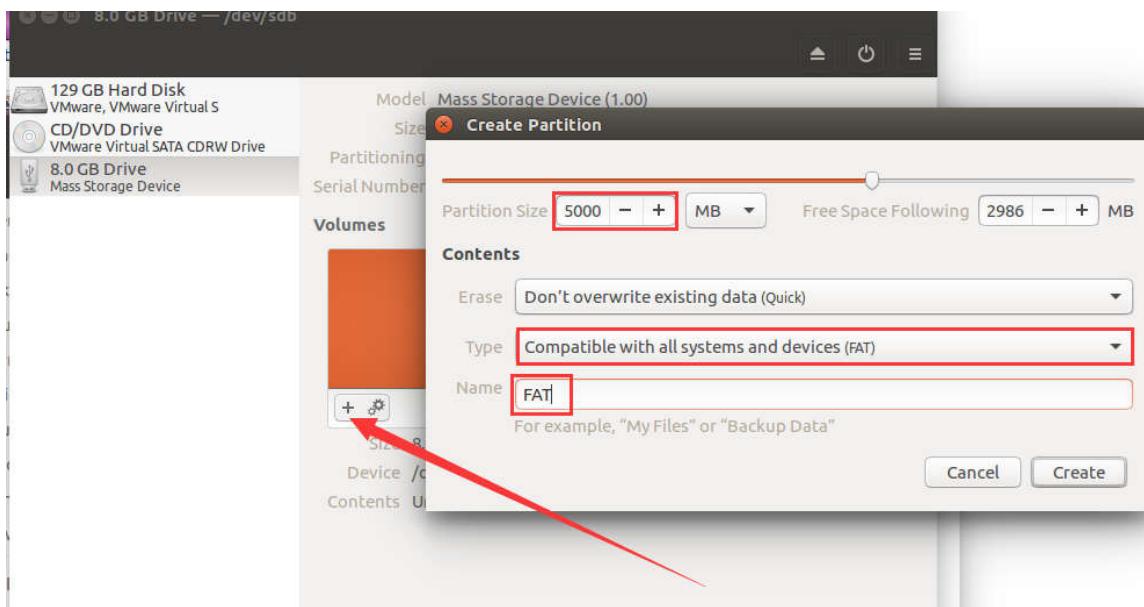
- 4) 鼠标点击 Disks 图标，出现"Disks"的对话框，本实验中 SD 卡已经分为 2 个分区，一个名为 FAT，一个名为 EXT，这里要重新分区。



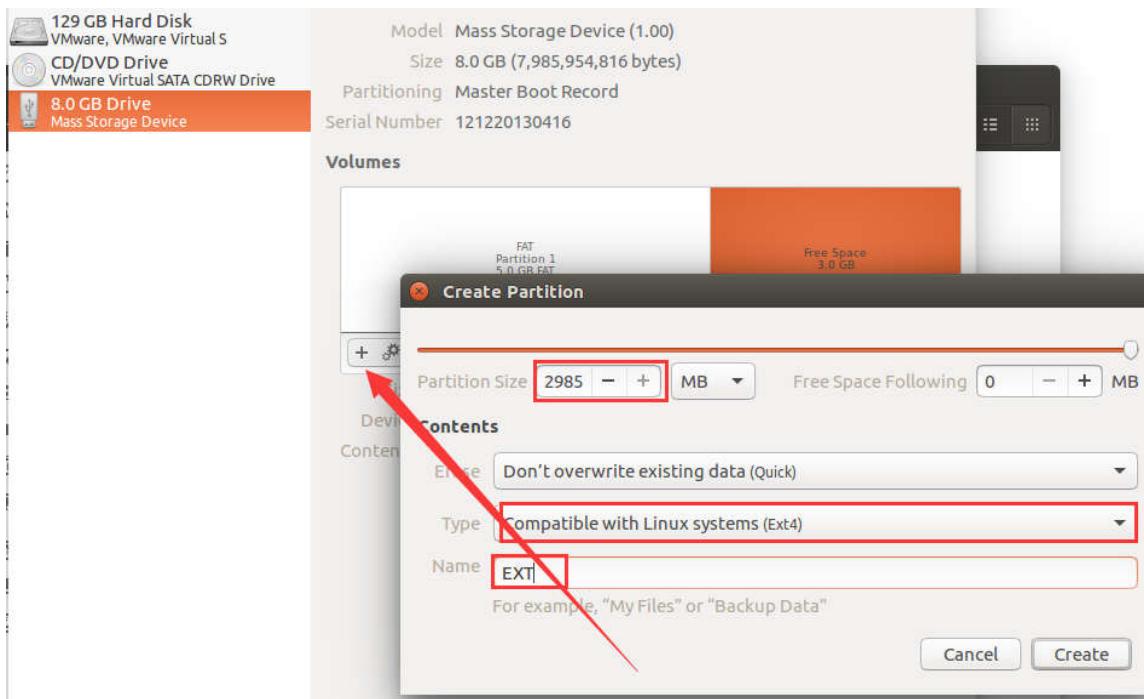
- 5) 选择每个分区下面的删除分区图标，删除所有分区。



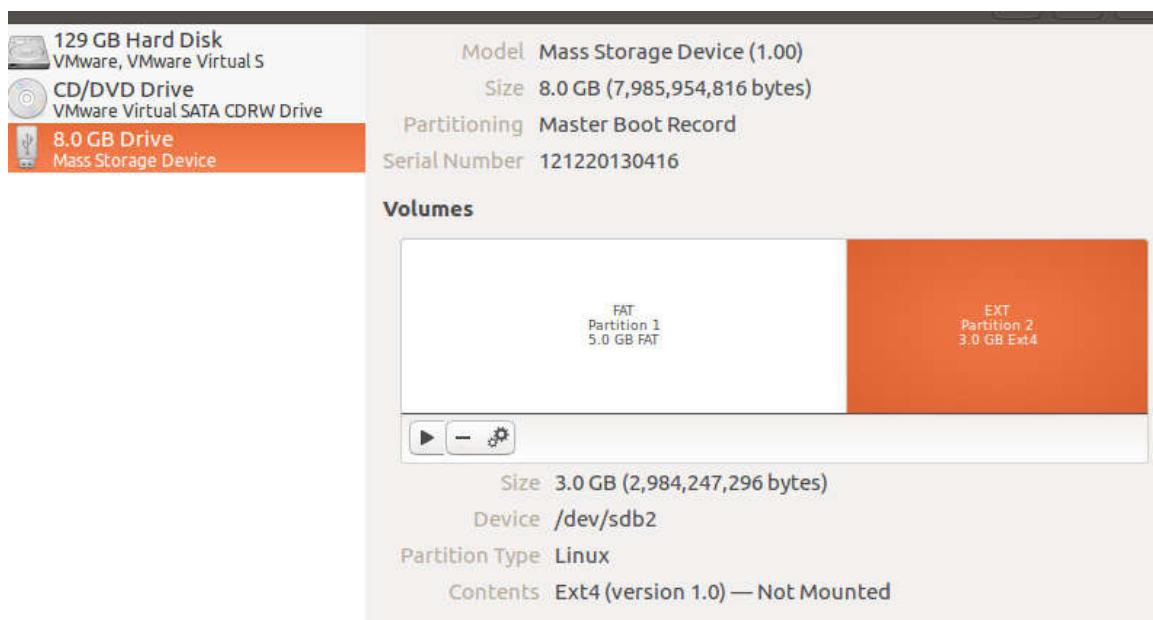
- 6) 点击添加分区的图标，添加第一个分区，本实验填写 5000MB，格式为 FAT，用于存放 ZYNQ 的启动文件 BOOT.bin 和内核文件、设备树，名称为 FAT



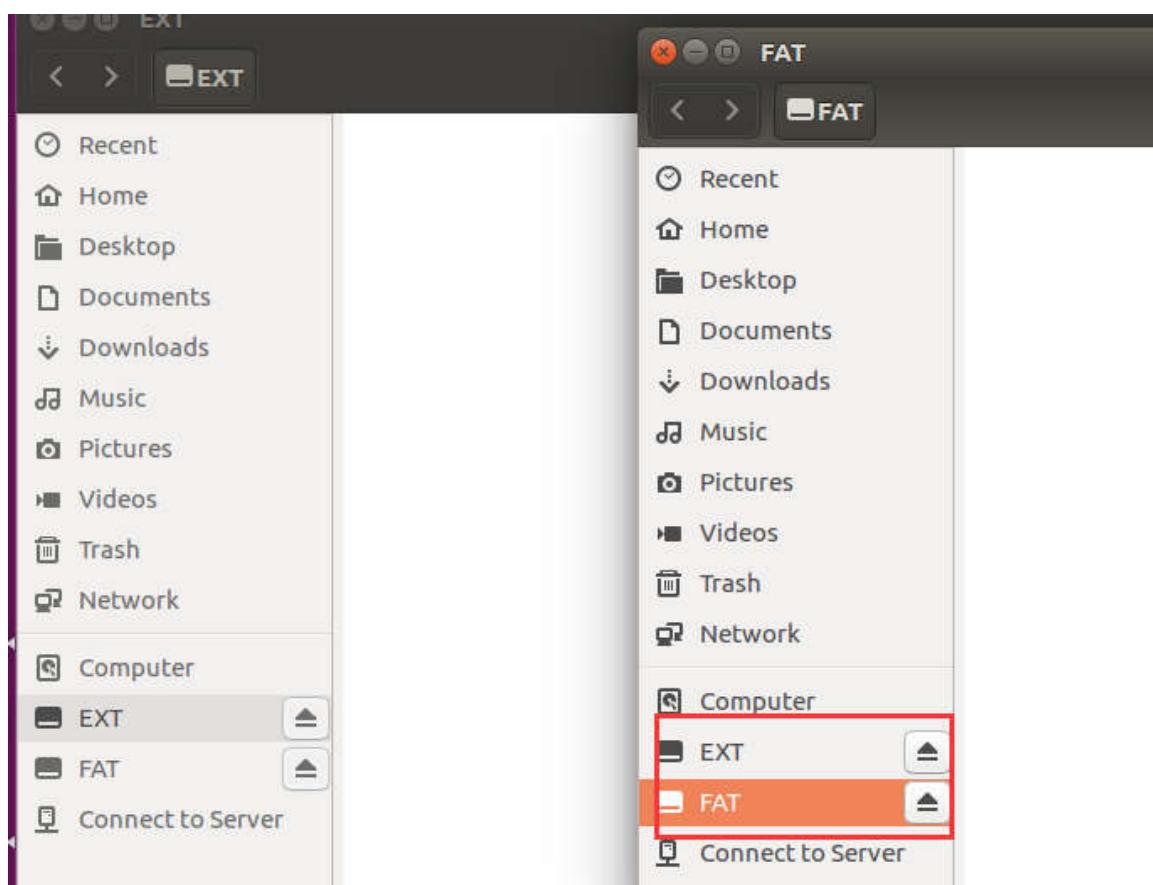
7) 创建第二个分区，用于存放根文件系统，格式为 EXT4，名称为 EXT



8) 创建分区完成后重新插入 sd 卡

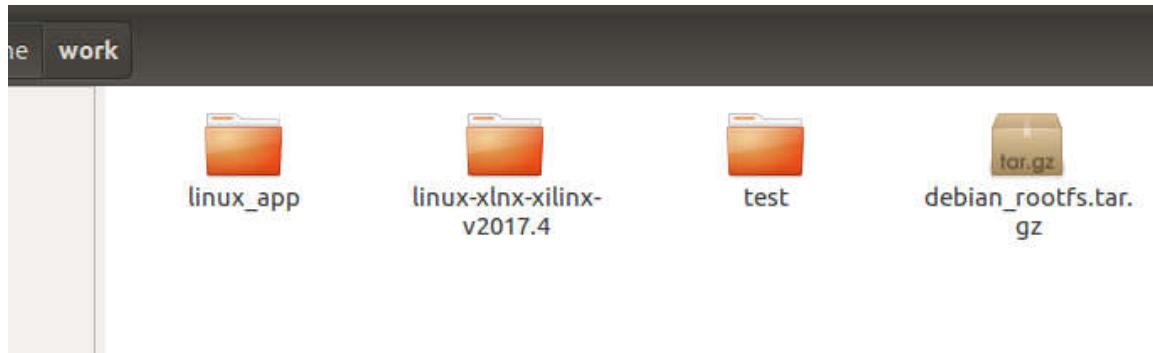


9) 系统会自动挂载分区，并弹出窗口



26.4.2 同步根文件系统到 SD 卡 EXT4 分区

- 1) 复制根文件系统的压缩包到 Linux 主机（实验复制在 “/home/alinx/work” 目录下）



- 2) 输入解压缩命令“sudo tar zxvpf debian_rootfs.tar.gz”，解压缩文件系统。解压缩可能需要几分钟的时间。

```
alinx@ubuntu:~/work$ sudo tar zxvpf debian_rootfs.tar.gz
```

- 3) 我们需要把这些文件系统的所有文件复制到 SD 卡的 EXT 分区的根目录下，在命令窗口输入命令“cd debian_rootfs”先进入根文件系统的目录。

```
alinx@ubuntu:~/work/debian_rootfs
debian_rootfs/usr/share/zoneinfo/Australia/Adelaide
debian_rootfs/usr/share/zoneinfo/Australia/Eucla
debian_rootfs/usr/share/zoneinfo/Australia/Darwin
debian_rootfs/usr/share/zoneinfo/Australia/Brisbane
debian_rootfs/usr/share/zoneinfo/Australia/North
debian_rootfs/usr/share/zoneinfo/Australia/NSW
debian_rootfs/usr/share/zoneinfo/Australia/Yancowinna
debian_rootfs/usr/share/zoneinfo/Australia/Canberra
debian_rootfs/usr/share/zoneinfo/Australia/West
debian_rootfs/usr/share/zoneinfo/Australia/Queensland
debian_rootfs/usr/share/zoneinfo/Australia/Perth
debian_rootfs/usr/share/zoneinfo/Australia/Sydney
debian_rootfs/usr/share/zoneinfo/Australia/Tasmania
debian_rootfs/usr/share/zoneinfo/Jamaica
debian_rootfs/usr/share/zoneinfo/GB-Eire
debian_rootfs/usr/share/zoneinfo/PST8PDT
debian_rootfs/usr/share/zoneinfo/EST
debian_rootfs/usr/share/zoneinfo/CST6CDT
debian_rootfs/usr/share/zoneinfo/Hongkong
debian_rootfs/usr/share/zoneinfo/localtime
debian_rootfs/usr/share/zoneinfo/Poland
debian_rootfs/mnt/
alinx@ubuntu:~/work$ cd debian_rootfs
alinx@ubuntu:~/work/debian_rootfs$
```

- 4) 在命令窗口输入“sudo rsync -av ./ /media/alinx/EXT”(/media/alinx/EXT 为 SD 卡 EXT4 分区的路径，可能会有不同，请根据自己实际情况修改)，开始同步当前目录到 SD 卡的 EXT 分区根目录，同步可能需要十几分钟的时间。

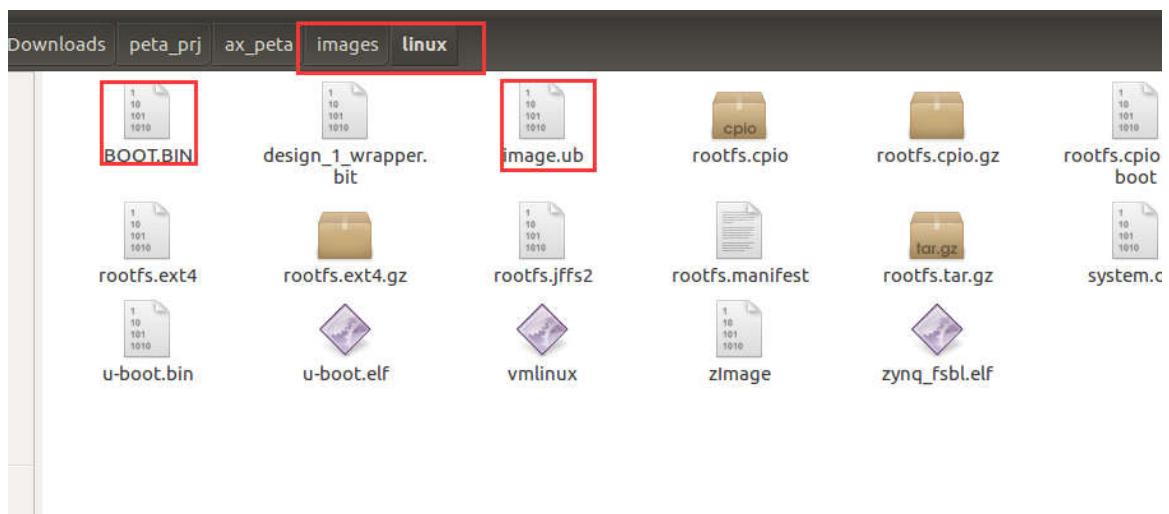
```

var/log/dpkg.log
var/log/faillog
var/log/fontconfig.log
var/log/lastlog
var/log/wtmp
var/log/apt/
var/log/apt/history.log
var/log/apt/term.log
var/log/fsck/
var/log/fsck/checkfs
var/log/fsck/checkroot
var/log/ntpstats/
var/mail/
var/opt/
var/spool/
var/spool/mail -> ../mail
var/spool/cron/
var/spool/cron/crontabs/
var/spool/rsyslog/
var/tmp/

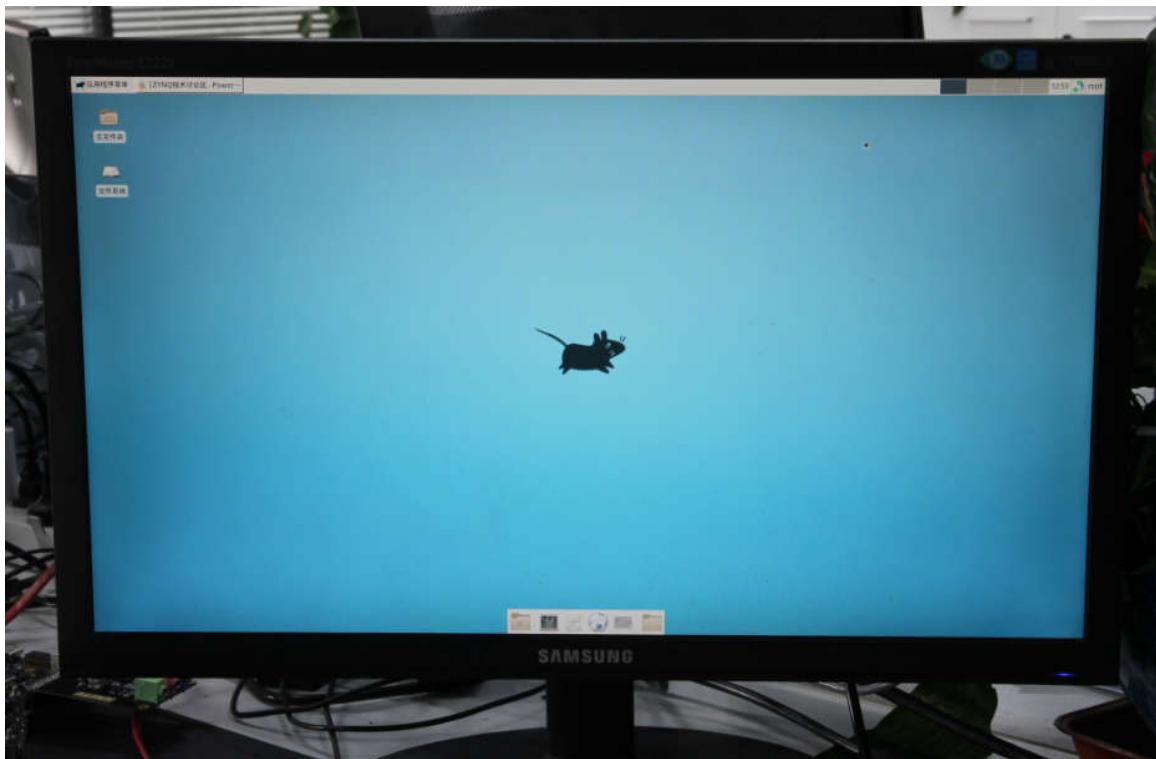
sent 1,117,338,469 bytes received 809,451 bytes 5,414,759.90 bytes/sec
total size is 1,114,115,109 speedup is 1.00
alinx@ubuntu:~/work/debian_rootfs$ sudo rsync -av ./ /media/alinx/EXT

```

- 5) 当命令行中重新出现命令提示符时，表示同步的过程结束。
- 6) 把 BOOT.bin 和 iamge.ub 复制到 sd 的 FAT32 分区(第一分区)中 ,设置开发板 sd 模式启动，插上 HDMI 显示器，启动开发板。



- 7) SD 卡制作完成后 ,把制作好后的 SD 卡插入到开发板的 SD 卡插槽内。连接 USB 串口线，连接 HDMI 显示器，开发板上电后在 HDMI 显示器上会显示 Debian 的操作系统的界面。另外在串口工具里我们可以看到操作系统启动的过程, 运行 u-boot 之后开始运行 Linux, 账号: root , 密码 : root



第二十七章 PCIe SSD 应用

在 vivado 工程里配置 PCIe , 内核驱动里也添加了 SSD 的驱动 , 本章学习如何 PCIe 接口的 SSD。教程都是基于前面教程已经完成的 Debian 8 系统。

如下面图片展示 , SSD 硬盘需要一个 PCIe 转 NVMe 转接板。



27.1 查看 PCI 设备

- 1) 给开发板 ETH1 连接到路由器 , 能上互联网 , 因为要在线安装一些软件
- 2) 登录开发板系统 , 使用下面命令安装 pci 工具

```
apt-get install pciutils
```

```
[ OK ] Reached target Graphical Interface.
Starting Update UTMP about System Runlevel Changes...
[ OK ] Started Update UTMP about System Runlevel Changes.
net eth1: Promiscuous mode disabled.
xilinx_axienet 41000000.ethernet eth1: Link is Down

Debian GNU/Linux 8 zyng ttyPS0

zyng login: macb e000b000.ethernet eth0: link up (1000/Full)

Debian GNU/Linux 8 zyng ttyPS0

zyng login: root
密码:
上一次登录: 五 4月 27 11:08:55 CST 2018ttyPS0 上
Linux zyng 4.9.0-xilinx #5 SMP PREEMPT Tue Apr 24 19:34:40 CST 2018 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

root@zyng:~# apt-get install pciutils
```

- 3) 运行下面命令查看 pcie 设备，可以看到有个 “Non-Volatile memory controller: Samsung Electronics Co Ltd Device a804” 设备，这个就是 SSD 硬盘。

lspci

```
root@zyng:~# lspci
00:00.0 PCI bridge: Xilinx Corporation Device 7124
01:00.0 Non-Volatile memory controller: Samsung Electronics Co Ltd Device a804
```

- 4) 运行下面命令,可以看到 SSD 没有格式化 ,大小 232.9G

lsblk

```
root@zyng:~# lsblk
NAME      MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
mtdblock2    31:2    0 10.5M  0 disk
mtdblock0    31:0    0   5M  0 disk
mmcblk1boot0 179:16   0   4M  1 disk
mmcblk1     179:8    0  7.3G  0 disk
└─mmcblk1p1 179:9    0   3G  0 part /emmc1
└─mmcblk1p2 179:10   0  4.3G  0 part /emmc2
mmcblk1rpmb 179:32   0   4M  0 disk
mtdblock3    31:3    0 16.4M  0 disk
mtdblock1    31:1    0 128K  0 disk
mmcblk1boot1 179:24   0   4M  1 disk
nvme0n1     259:0    0 232.9G 0 disk
└─nvme0n1p0 179:0    0   7.4G  0 disk
└─mmcblk0p2 179:2    0   2G  0 part /
└─mmcblk0p1 179:1    0   5.2G  0 part /sd
```

27.2 格式化 SSD

- 1) 使用下面命令格式化 SSD

```
fdisk /dev/nvme0n1
```

```
root@zynq:~# fdisk /dev/nvme0n1

Welcome to fdisk (util-linux 2.25.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0x151972a6.

Command (m for help):
```

- 2) 输入 n 回车，创建新分区，输入 p 回车，然后再输入 1 回车，创建第一分区，然后按回车
保持默认分区大小，最好输入 w 输入数据到磁盘

```
Welcome to fdisk (util-linux 2.25.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0x151972a6.

Command (m for help): n
Partition type
  p  primary (0 primary, 0 extended, 4 free)
  e  extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-488397167, default 2048): vgaarb: this pci device is not a vg
a device

Last sector, +sectors or +size{K,M,G,T,P} (2048-488397167, default 488397167):
+232943M

Created a new partition 1 of type 'Linux' and of size 232.9 GiB.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
  nvme0n1: p1
Syncing disks.

root@zynq:~# nvme0n1: p1
```

- 3) 使用下面命令创建文件系统格式化分区

```
mkfs -t ext4 /dev/nvme0n1p1
```

```
root@zynq:~# mkfs -t ext4 /dev/nvme0n1p1
mke2fs 1.42.12 (29-Aug-2014)
Discarding device blocks: 完成
Creating filesystem with 61049390 4k blocks and 15269888 inodes
Filesystem UUID: c7e98d5f-18a2-4936-bc52-1b3c4b4bd5eb
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
    4096000, 7962624, 11239424, 20480000, 23887872

Allocating group tables: 完成
正在写入inode表: 完成
Creating journal (32768 blocks): 完成
Writing superblocks and filesystem accounting information: 完成

root@zynq:~#
```

- 4) 创建一个文件夹，然后挂载 SSD 分区

```
mkdir /media/nvme
mount /dev/nvme0n1p1 /media/nvme
```

```
root@zynq:~# mkdir /media/nvme
root@zynq:~# mount /dev/nvme0n1p1 /media/nvme
EXT4-fs (nvme0n1p1): mounted filesystem with ordered data mode. Opts: (null)
root@zynq:~#
```

- 5) 建立一个文件夹测试一下

```
cd /media/nvme
mkdir test
sync
```

```
root@zynq:~# cd /media/nvme
root@zynq:/media/nvme# mkdir test
root@zynq:/media/nvme# sync
root@zynq:/media/nvme#
```

27.3 SSD 测速

- 1) 写入速度测试，写入 2.1GB 数据，速度 139MB/S

```
dd if=/dev/zero of=/dev/nvme0n1p1 bs=2M count=1000
```

```
root@zynq:/media/nvme# dd if=/dev/zero of=/dev/nvme0n1p1 bs=2M count=1000
记录了1000+0 的读入
记录了1000+0 的写出
2097152000字节(2.1 GB)已复制, 15.0342 秒, 139 MB/秒
root@zynq:/media/nvme#
```

- 2) 读取速度测试，读取 2.1G，速度 142MB/S

```
dd if=/dev/nvme0n1p1 of=/dev/null bs=2M count=1000
```

```
root@zynq:/media/nvme# dd if=/dev/nvme0n1p1 of=/dev/null bs=2M count=1000
记录了1000+0 的读入
记录了1000+0 的写出
2097152000字节 (2.1 GB) 已复制, 14.7306 秒, 142 MB/秒
root@zynq:/media/nvme#
```


第二十八章 QSPI 和 EMMC 启动 Linux

前面教程讲解的 Linux 都是在 SD 里启动，本实验讲解如何使用 Petalinux 制作一个从 QSPI Flash 和 EMMC 启动的 Linux。

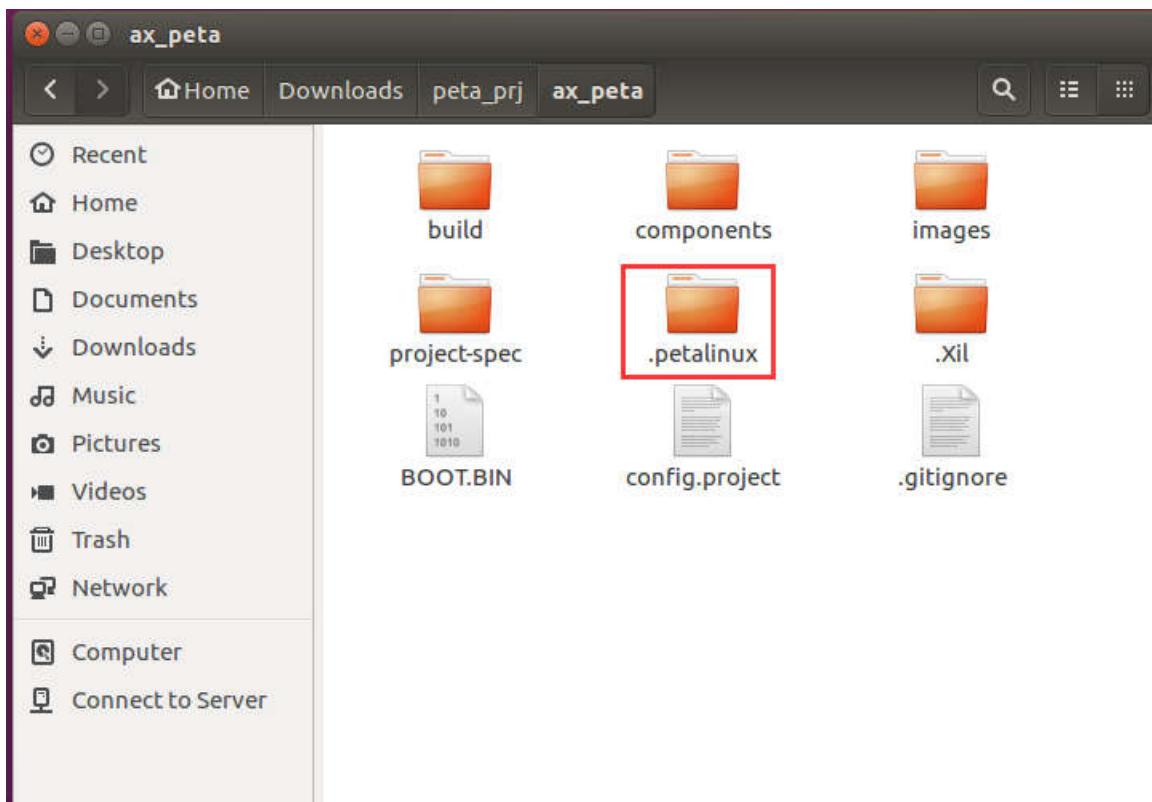
QSPI 有效可用的大小为 16MBytes，很多情况下不够存放 BOOT.bin, image.ub 文件，我们可以把 BOOT.bin 文件放在 QSPI Flash, image.ub 文件放在 EMMC，本实验只是演示如何灵活应用 QSPI 和 EMMC，如何存放这些文件还是要根据具体的应用来决定。

需要注意 ZYNQ7000 不能直接从 EMMC 启动。

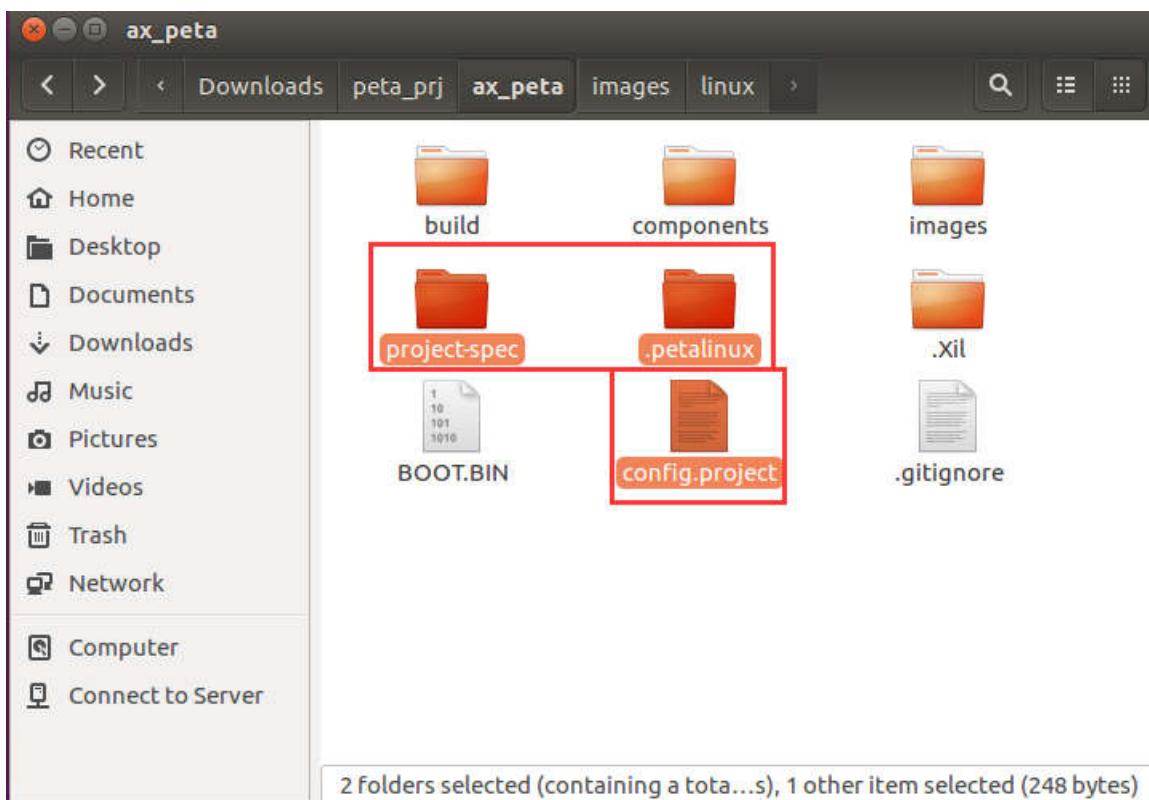
28.1 复制 Petalinux 工程

前面的教程中我们已经使用 Petalinux 做了 SD 卡启动的各种实验，我们想保留 SD 启动的工程，但是又不想新建一个工程，我们可以把老的工程复制一份。

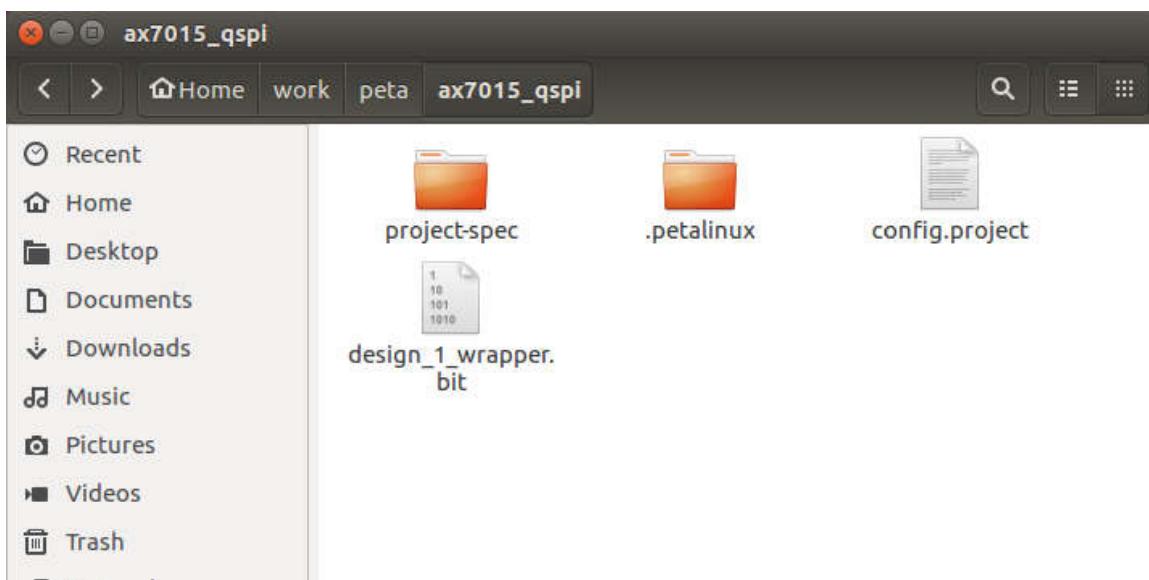
- 1) 在工程目录同时按键 **Ctrl +H**，显示隐藏文件



- 2) 把 project-spec、.petalinux、config.project 复制到一个新的目录，做一个新的 Petalinux 工程



- 3) 再把 images/linux 目录下的 bit 文件也复制到新的工程目录下，用于合成带 PL 配置的 BOOT



28.2 配置编译 Petalinux

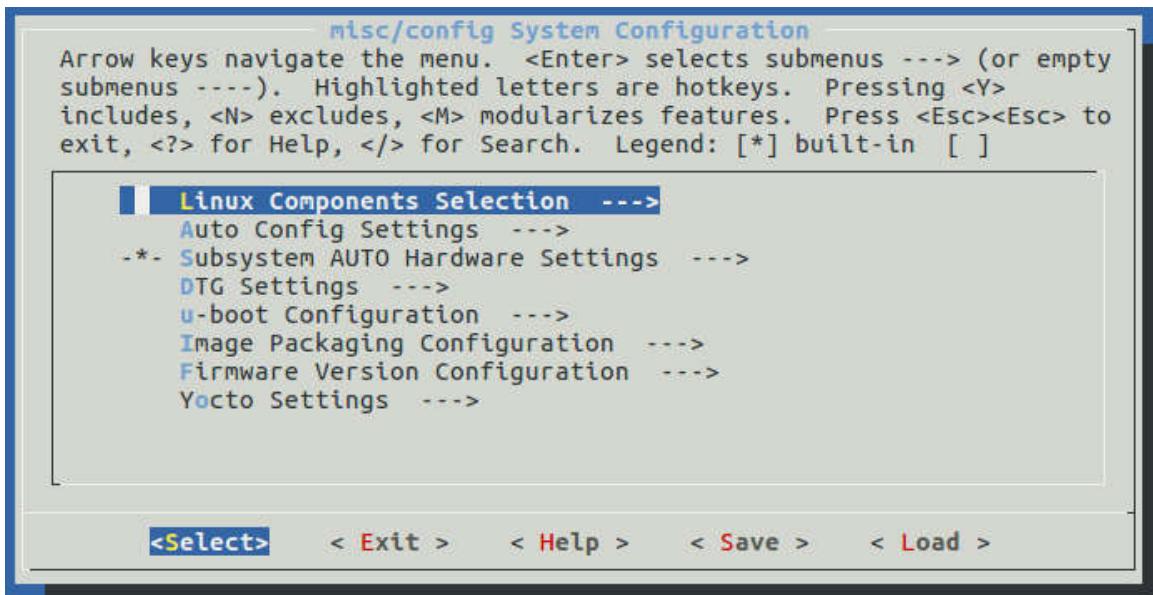
- 1) 使用下面命令设置环境变量

```
source /opt/pkg/petalinux/settings.sh  
source /opt/Xilinx/Vivado/2017.4/settings64.sh
```

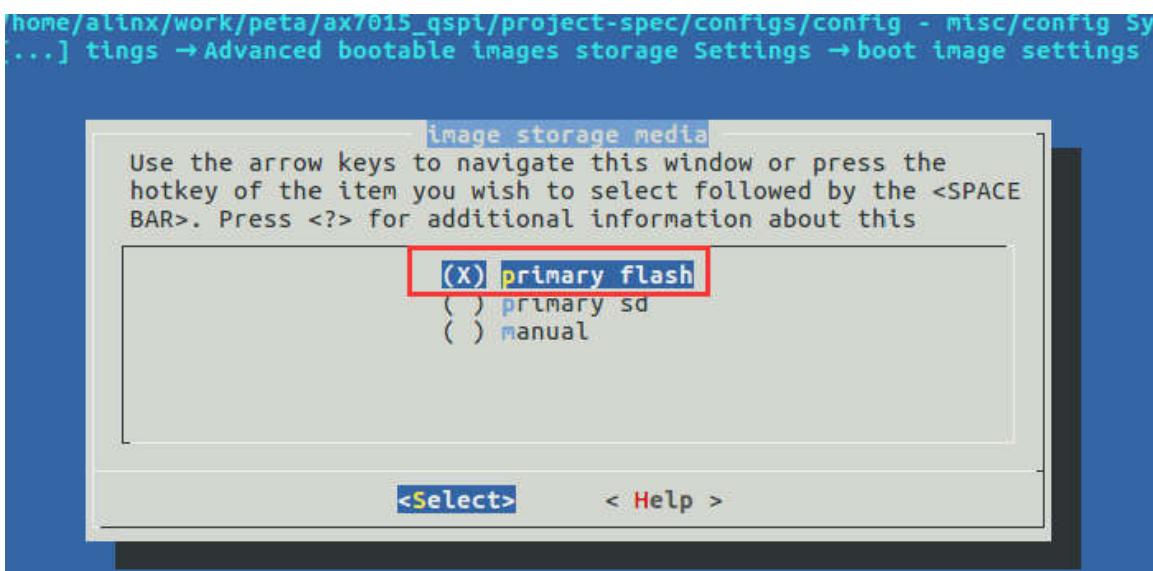
```
alinx@ubuntu:~/work/peta/ax7015_qspi$ source /opt/pkg/petalinux/settings.sh
```

```
alinx@ubuntu:~/work/peta/ax7015_qspi$ source /opt/Xilinx/Vivado/2017.4/settings64.sh
```

2) 使用 petalinux-config 命令配置 Petalinux

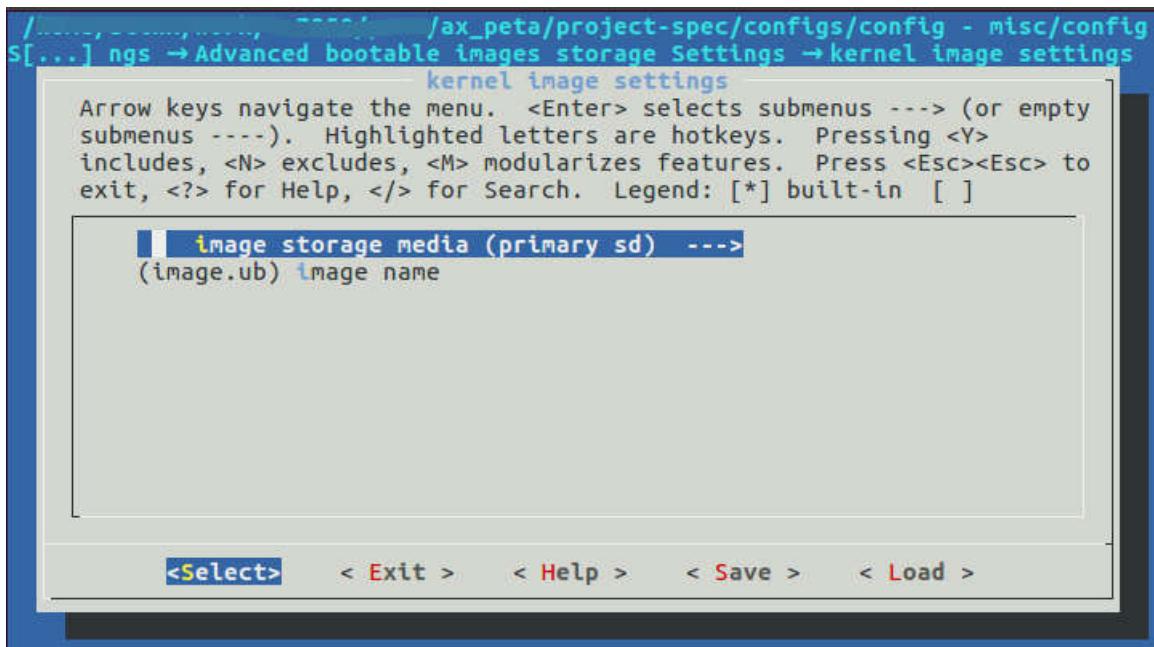


3) 在 Subsystem AUTO Hardware Settings ---> Advanced bootable images storage Settings ---> boot image settings ---> image storage media 选项中选择 primary flash。表示启动文件 BOOT.bin 要放在 QSPI 启动。

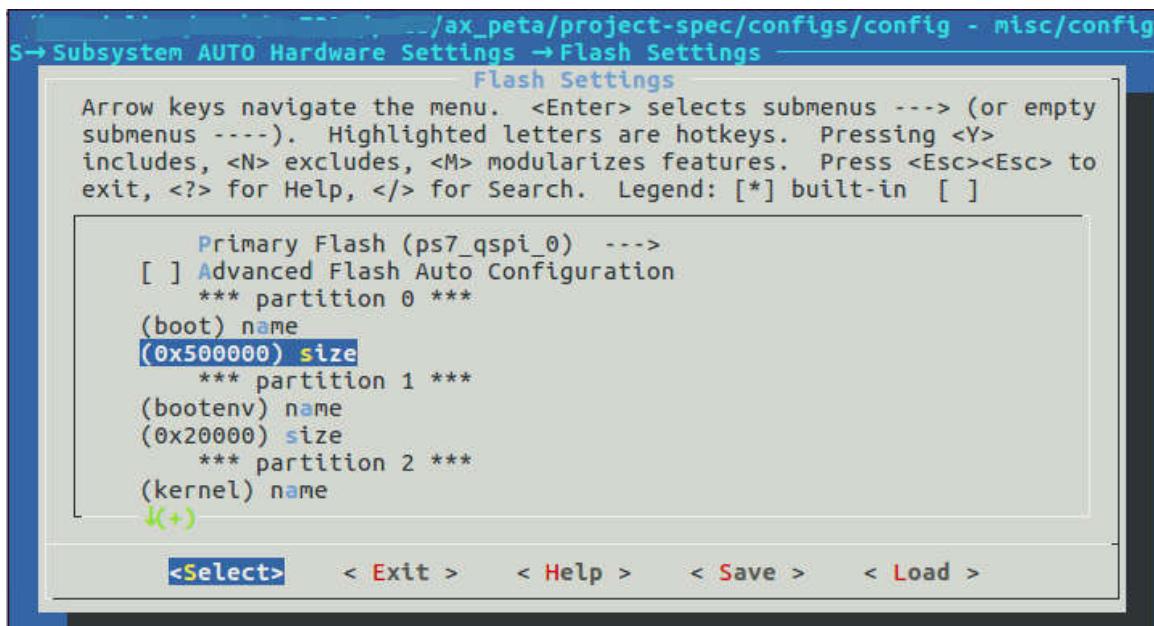


4) 在 Subsystem AUTO Hardware Settings ---> Advanced bootable images storage Settings --->

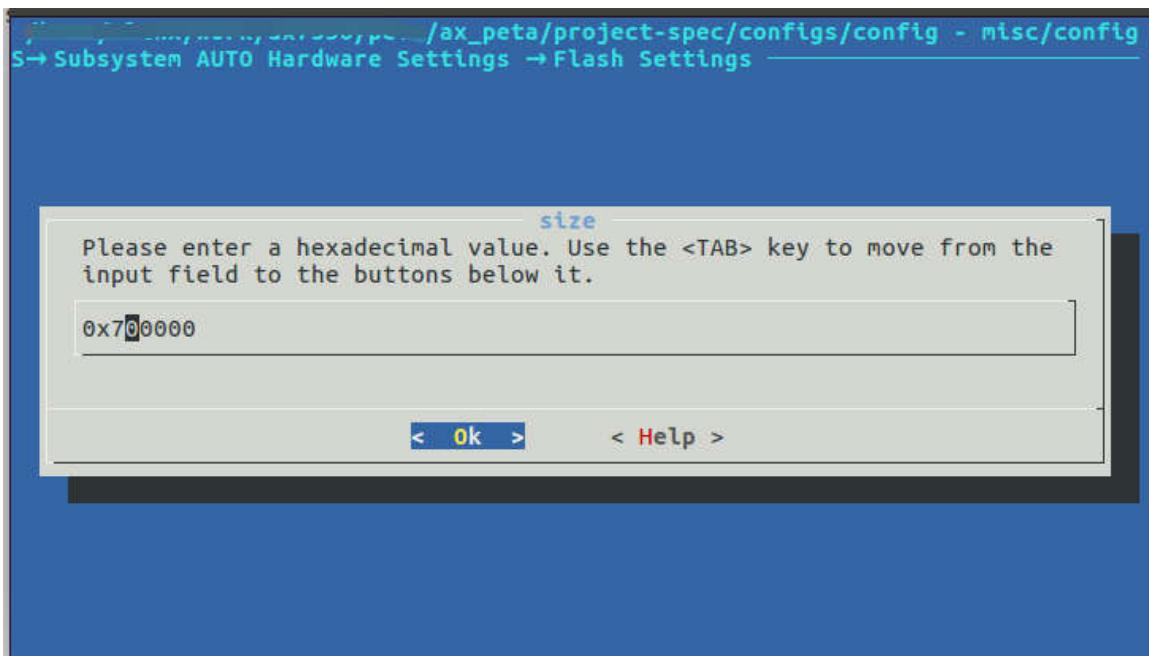
kernel image settings ---> image storage media 选项中选择 primary sd



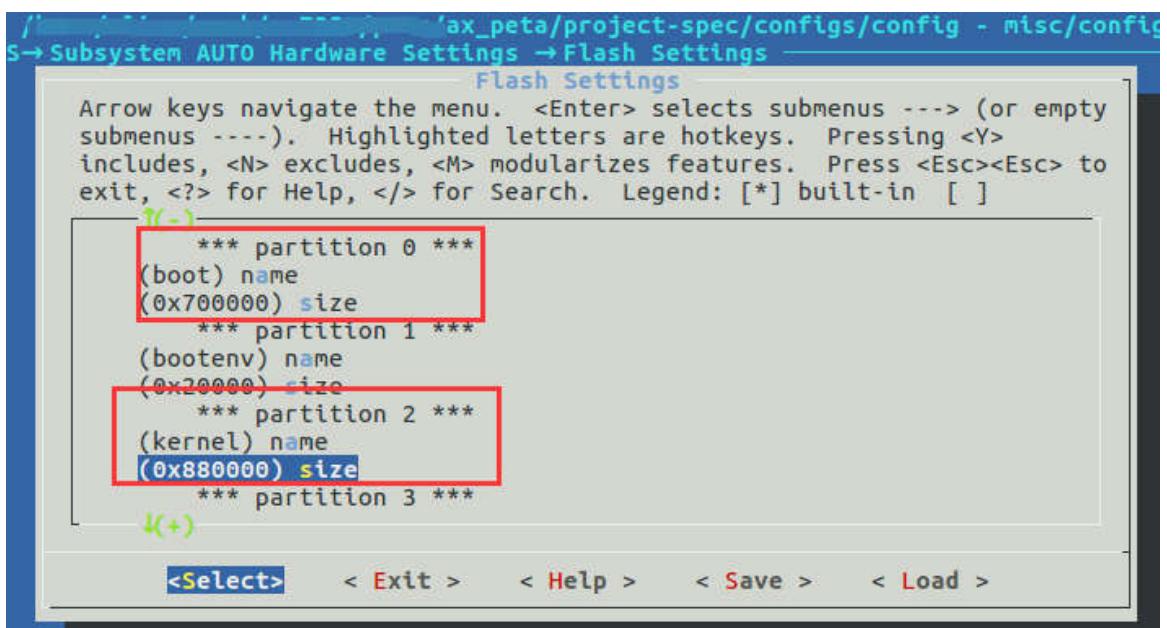
5) Subsystem AUTO Hardware Settings → Flash Settings 中可以修改 QSPI flash 的分区。



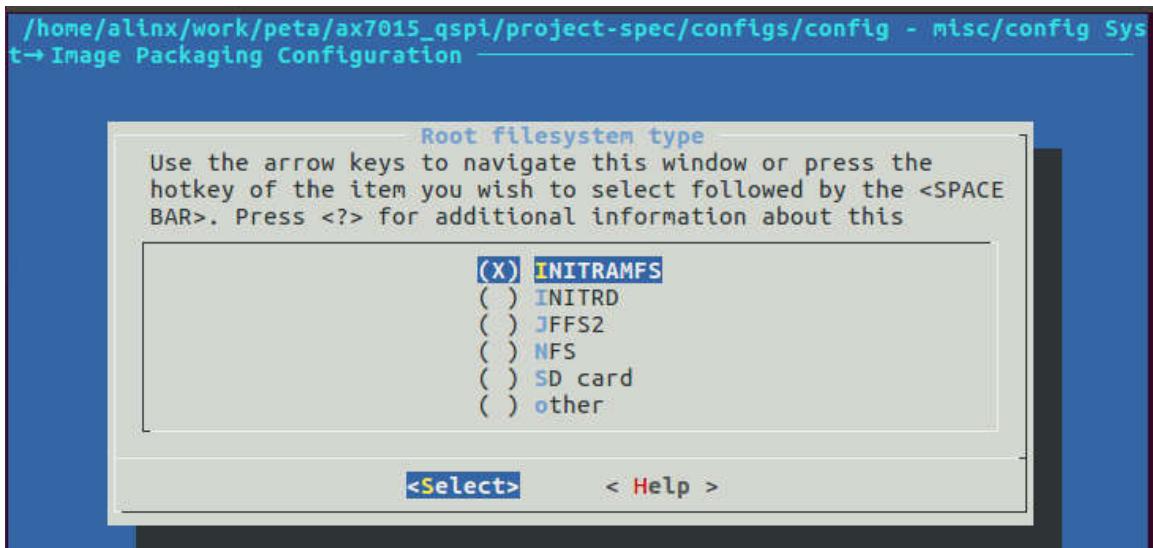
6) 选择 size 后回车进入编辑模式



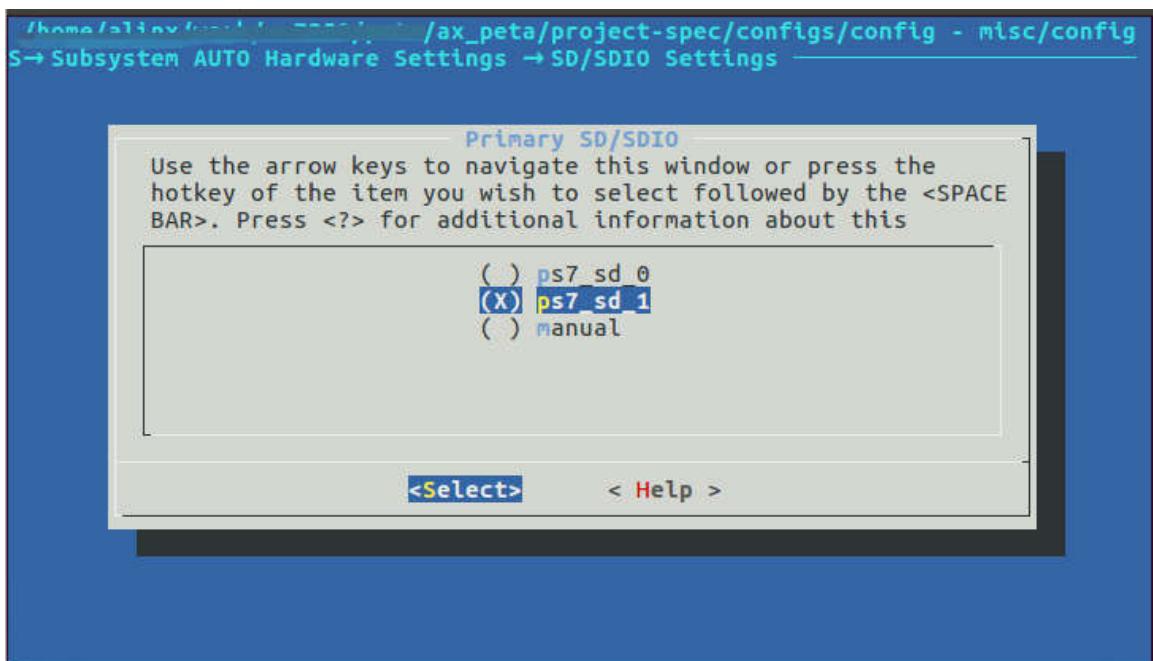
- 7) 按照图中修改 boot 的大小



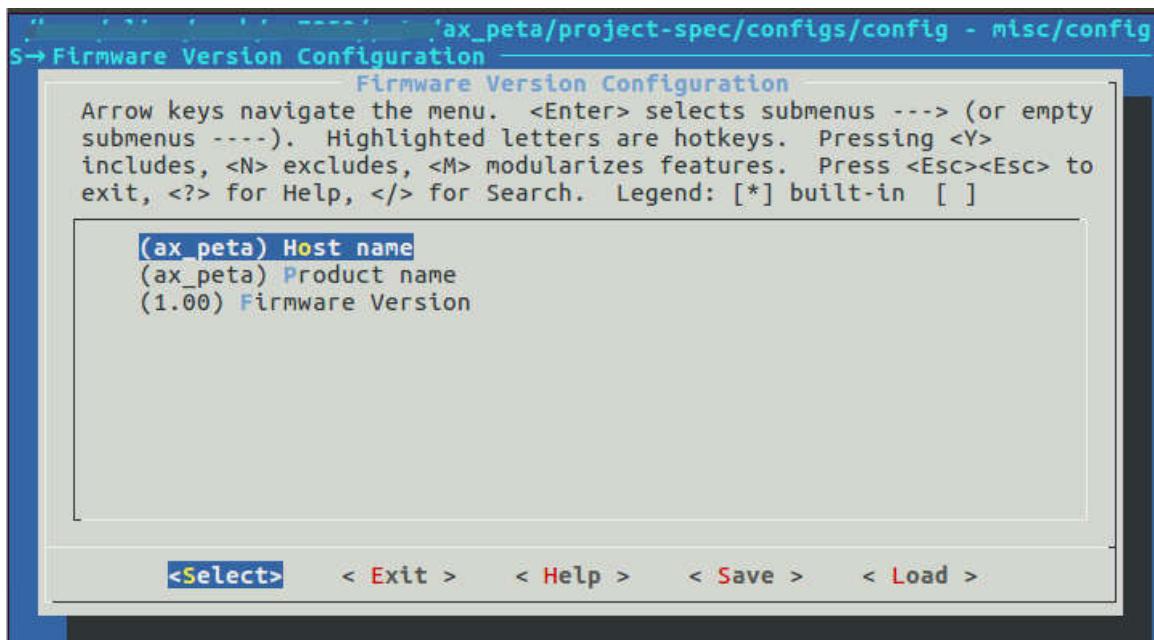
- 8) 在 Image Packaging Configuration ---> Root filesystem type 选择 INITRAMFS , 使用 RAM 类型的根文件系统



- 9) 在 Subsystem AUTO Hardware Settings → SD/SDIO Settings 中选择 ps7_sd_1 , 因为 EMMC 在 sd1 控制器上。

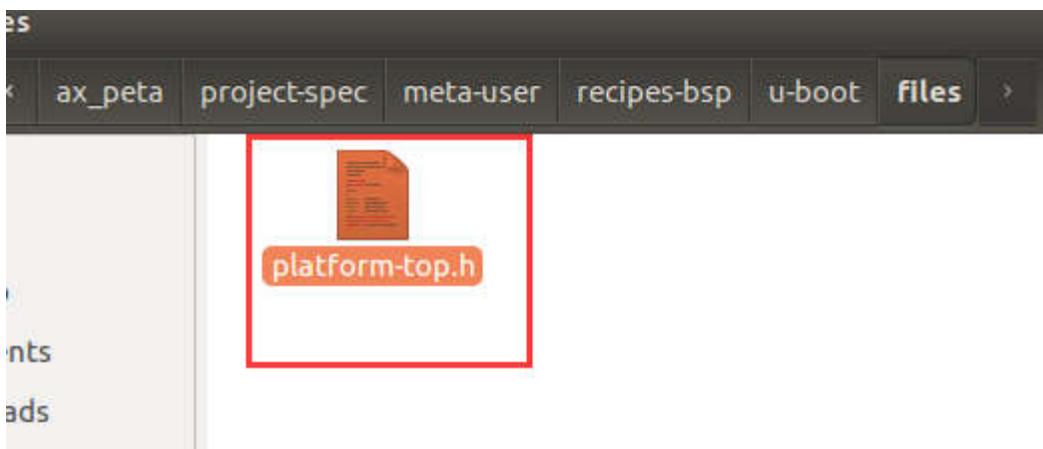


- 10) 在 Firmware Version Configuration ---> 中可以修改 Host name 等信息



11) 保存配置

12) 修改 uboot 配置文件 (project-spec/meta-user/recipes-bsp/u-boot/files/platform-top.h)



13) platform-top.h 修改后内容为

```
#include <configs/platform-auto.h>
#define CONFIG_SYS_BOOTM_LEN 0xF000000

/*Required for uartless designs */
#ifndef CONFIG_BAUDRATE
#define CONFIG_BAUDRATE 115200
#endif
#ifndef CONFIG_DEBUG_UART
#define CONFIG_DEBUG_UART
#endif
#endif

/* sdio - ps7_sd_1 */
#define CONFIG_ZYNQ_SDHCI1 0xE0101000

#define CONFIG_ZYNQ_HISPD_BROKEN

/* Extra U-Boot Env settings */
#define CONFIG_EXTRA_ENV_SETTINGS \
    SERIAL_MULTI \
```

```
CONSOLE_ARG \
PSSERIAL0 \
"progemmc=" \
"if fatload mmc 0:1 0x20000000 image.ub; then " \
"echo Copying Linux image.ub to eMMC ...; " \
"echo Loading image.ub from SD card into memory ...; " \
"fatload mmc 0:1 0x20000000 image.ub; " \
"echo Copying image.ub from memory to eMMC partition ...; " \
"fatwrite mmc 1:1 0x20000000 image.ub ${filesize}; " \
"echo Copying Linux image.ub to eMMC Complete ...;" \
"fi\0" \
"nc=setenv stdout nc;setenv stdin nc;\0" \
"ethaddr=00:0a:35:00:1e:53\0" \
"importbootenv=echo \"Importing environment from SD ...\"; " \
"env import -t ${loadbootenv addr} ${filesize}\0" \
"loadbootenv=load mmc $sdbootdev:$partid ${loadbootenv_addr} ${bootenv}\0" \
"sd_uEnvtxt_existance_test=test -e mmc $sdbootdev:$partid /uEnv.txt\0" \
"uenvboot=" \
"if run sd_uEnvtxt_existance_test; then " \
"run loadbootenv; " \
"echo Loaded environment from ${bootenv}; " \
"run importbootenv; " \
"fi\0" \
"sdboot=echo boot Petalinux; run uenvboot ; mmcinfo && fatload mmc 0 ${netstart} ${kernel_img} \
&& bootm \0" \
"autoload=no\0" \
"clobstart=0x10000000\0" \
"netstart=0x10000000\0" \
"dtbnetstart=0x1800000\0" \
"loadaddr=0x10000000\0" \
"bootsize=0x640000\0" \
"bootstart=0x0\0" \
"boot_img=BOOT.BIN\0" \
"load_boot=tftpboot ${clobstart} ${boot_img}\0" \
"update_boot=setenv img boot; setenv psize ${bootsize}; setenv installcmd \"install_boot\"; run \
load_boot test_img; setenv img; setenv psize; setenv installcmd\0" \
"sd_update_boot=echo Updating boot from SD; mmcinfo && fatload mmc 0:1 ${clobstart} ${boot_img} \
&& run install_boot\0" \
"install_boot=sf probe 0 && sf erase ${bootstart} ${bootsize} && " \
"sf write ${clobstart} ${bootstart} ${filesize}\0" \
"bootenvsize=0x2000\0" \
"bootenvstart=0x640000\0" \
"eraseenv=sf probe 0 && sf erase ${bootenvstart} ${bootenvsize}\0" \
"jffs2_img=rootfs.jffs2\0" \
"load_jffs2=tftpboot ${clobstart} ${jffs2_img}\0" \
"update_jffs2=setenv img jffs2; setenv psize ${jffs2size}; setenv installcmd \"install_jffs2\"; \
run load_jffs2 test_img; setenv img; setenv psize; setenv installcmd\0" \
"sd_update_jffs2=echo Updating jffs2 from SD; mmcinfo && fatload mmc 0:1 ${clobstart} ${jffs2_img} \
&& run install_jffs2\0" \
"install_jffs2=sf probe 0 && sf erase ${jffs2start} ${jffs2size} && " \
"sf write ${clobstart} ${jffs2start} ${filesize}\0" \
"kernel_img=image.ub\0" \
"load_kernel=tftpboot ${clobstart} ${kernel_img}\0" \
"update_kernel=setenv img kernel; setenv psize ${kernelsize}; setenv installcmd \
\"install_kernel\"; run load_kernel ${installcmd}; setenv img; setenv psize; setenv installcmd\0" \
"install_kernel=mmcinfo && fatwrite mmc 0 ${clobstart} ${kernel_img} ${filesize}\0" \
"cp_kernel2ram=mmc dev 1 && fatload mmc 1 ${netstart} ${kernel_img}\0" \
"dtb_img=system.dtb\0" \
"load_dtb=tftpboot ${clobstart} ${dtb_img}\0" \
"update_dtb=setenv img dtb; setenv psize ${dtbsize}; setenv installcmd \"install_dtb\"; run \
load_dtb test_img; setenv img; setenv psize; setenv installcmd\0" \
"sd_update_dtb=echo Updating dtb from SD; mmcinfo && fatload mmc 0:1 ${clobstart} ${dtb_img} && \
run install_dtb\0" \
"fault=echo ${img} image size is greater than allocated place - partition ${img} is NOT UPDATED\0" \
"test_crc;if imi ${clobstart}; then run test_img; else echo ${img} Bad CRC - ${img} is NOT UPDATED; \
fi\0" \
"test_img=setenv var \"if test ${filesize} -gt ${psize}\\"; then run fault\\; else run \
${installcmd}\\; fi\\; run var; setenv var\0" \
"netboot=tftpboot ${netstart} ${kernel_img} && bootm\0" \
"default_bootcmd=run progemmc; run uenvboot; run cp_kernel2ram && bootm ${netstart}\0" \
""
```

14) 编译

```
*** Execute 'make' to start the build or try 'make help'.

[INFO] sourcing bitbake
[INFO] generating plnxtool conf
[INFO] generating meta-plnx-generated layer
~/work/peta/ax7015_qspi/build/misc/plnx-generated ~/work/peta/ax7015_qspi
~/work/peta/ax7015_qspi
[INFO] generating machine configuration
[INFO] generating bbappends for project . This may take time !
~/work/peta/ax7015_qspi/build/misc/plnx-generated ~/work/peta/ax7015_qspi
~/work/peta/ax7015_qspi
[INFO] generating u-boot configuration files

[INFO] generating kernel configuration files
[INFO] generating kconfig for Rootfs
Generate rootfs kconfig
[INFO] oldconfig rootfs
[INFO] generating petalinux-user-image.bb
[INFO] successfully configured project
webtalk failed:Petalinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get Petalinux usage statistics!
alinx@ubuntu:~/work/peta/ax7015_qspi$ petalinux-build
```

15) 使用下面命令合成 BOOT

```
petalinux-package --boot --fsbl ./images/linux/zynq_fsbl.elf --fpga design_1_wrapper.bit --u-boot
--force
```

28.3 如何烧写 EMMC

28.3.1 格式化并挂载 EMMC

EMMC 为板子的芯片，如果是批量生产，需要使用 EMMC 量产设备进行烧写，本实验使用前面做好的 Debian 8 系统，格式化并挂载 EMMC。

建立一个格式化并挂载 EMMC 的 SHELL 文件，本实验中把 EMMC 分 2 个区，一个格式化为 FAT32，用于存放 image.ub，另一个格式化为 EXT4，用于存放 Linux 应用程序。

```
#!/bin/sh

EMMC_P1_PATH=/media/emmc_p1
EMMC_P2_PATH=/media/emmc_p2
if [ ! -d $EMMC_P1_PATH ]
then
    mkdir $EMMC_P1_PATH
fi

if [ ! -d $EMMC_P2_PATH ]
then
    mkdir $EMMC_P2_PATH
fi

mount /dev/mmcblk1p1 $EMMC_P1_PATH
ret=$?
if [ $ret -ne 0 ]; then
    echo "Start format /dev/mmcblk1"
    dd if=/dev/zero of=/dev/mmcblk1 bs=512 count=1
```

```

sync
echo "format finished!"
echo "Step1:Parting the disks..."
fdisk /dev/mmcblk1 <<EOF
n
p
1
+30720
n
c
2

wq
EOF

partprobe &> /dev/null
echo "Part finished!"
echo "Step2:Formating disks..."
mkfs.vfat /dev/mmcblk1p1 &> /dev/null
mkfs.ext4 /dev/mmcblk1p2 &> /dev/null
echo "Format finished!"
mount /dev/mmcblk1p1 $EMMC_P1_PATH
mount /dev/mmcblk1p2 $EMMC_P2_PATH
else
    mount /dev/mmcblk1p2 $EMMC_P2_PATH
fi

```

把 format_mount_emmc.sh 文件放在 sd 卡，然后通过 sd 卡启动开发板，默认情况下 Debian 8 会把 sd 卡的第一分区挂载到/sd 目录

```

Debian GNU/Linux 8 zynq ttyPS0

zynq login: root
密码:
上一次登录: 四 1月 1 08:00:57 CST 1970ttyPS0 上
Linux zynq 4.9.0-xilinx #5 SMP PREEMPT Tue Apr 24 19:34:40 CST 2018 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@zynq:~# cd /sd
root@zynq:/sd# ls
00_base.zip  course_s1.zip  linux_app  startup.sh
BOOT.BIN      image.ub       readme.txt
root@zynq:/sd#

```

运行 format_mount_emmc.sh 文件

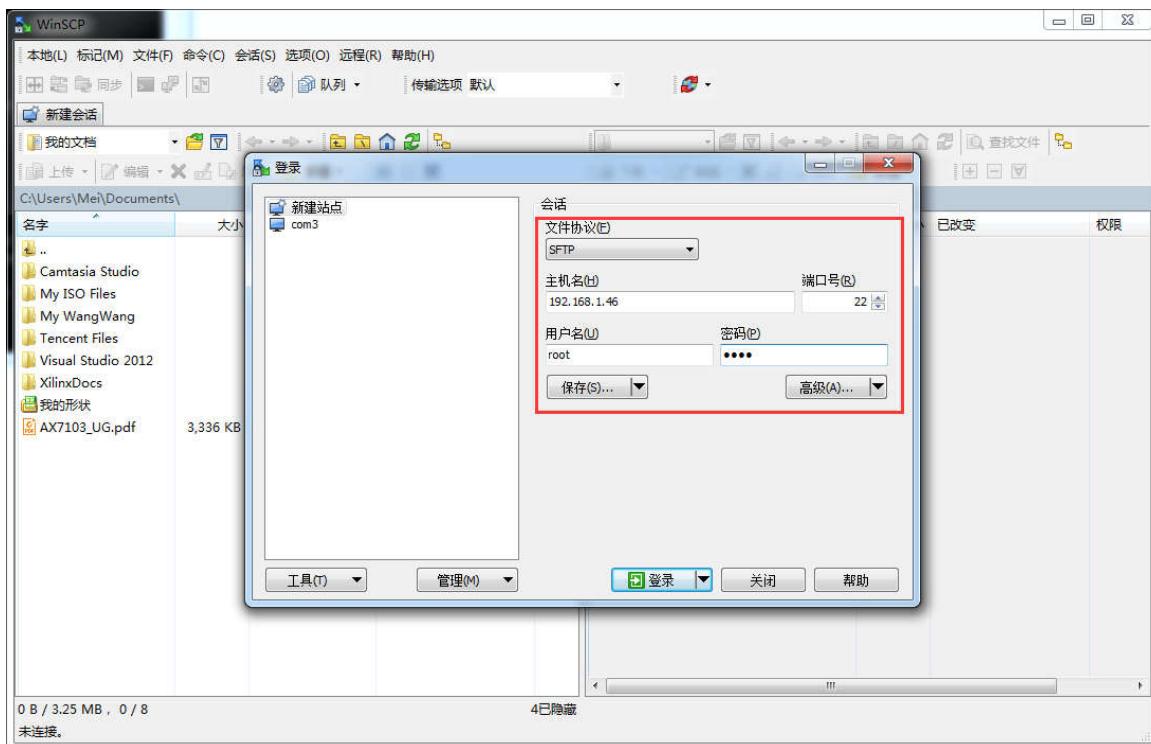
```

root@zynq:/sd# cd linux_app/
root@zynq:/sd/linux_app# ls
format_mount_emmc.sh
root@zynq:/sd/linux_app# ./format_mount_emmc.sh
FAT-fs (mmcblk1p1): Volume was not properly unmounted. Some data may be corrupt.
Please run fsck.
EXT4-fs (mmcblk1p2): recovery complete
EXT4-fs (mmcblk1p2): mounted filesystem with ordered data mode. Opts: (null)
root@zynq:/sd/linux_app#

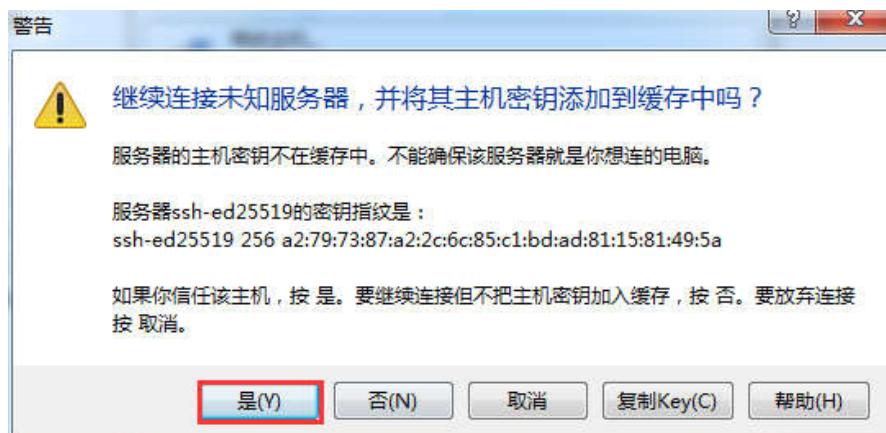
```

28.3.2 复制文件 emmc

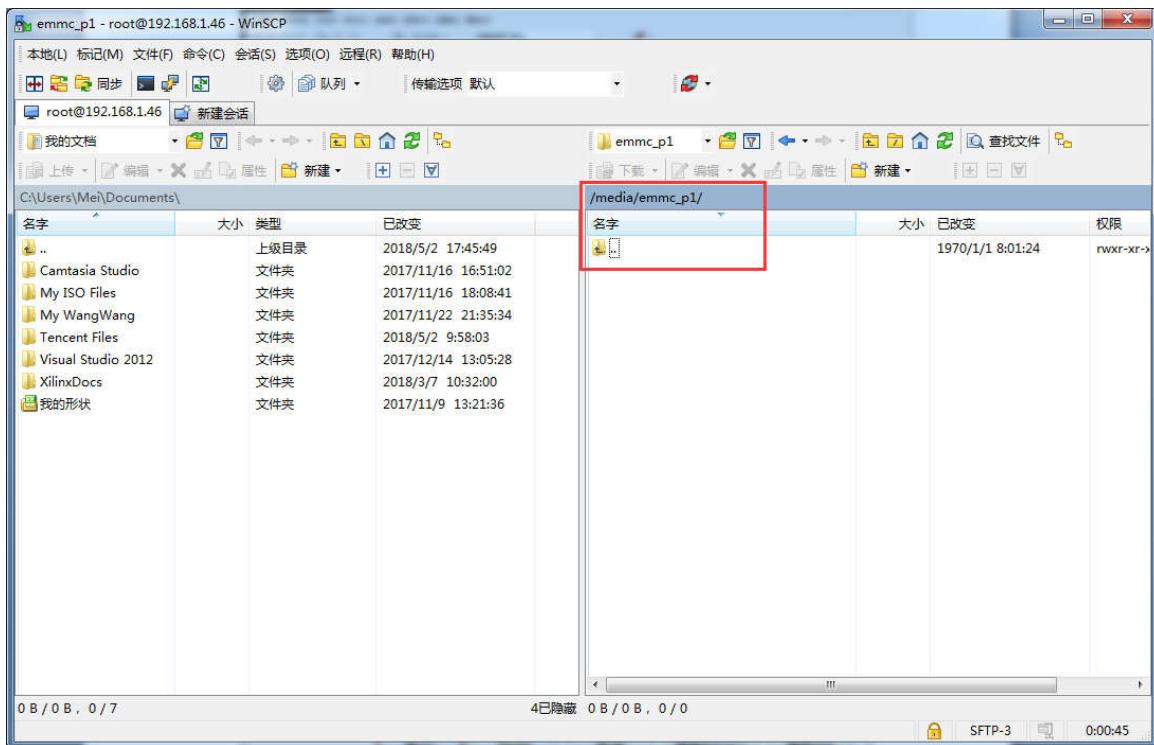
- 1) 大量复制文件到 emmc 时我们需要更快捷的工具，本实验通过 WinSCP 软件
- 2) 使用 sftp 协议，主机名填写开发板的 IP 地址，端口名 22，用户名 root，密码 root，登录到系统中



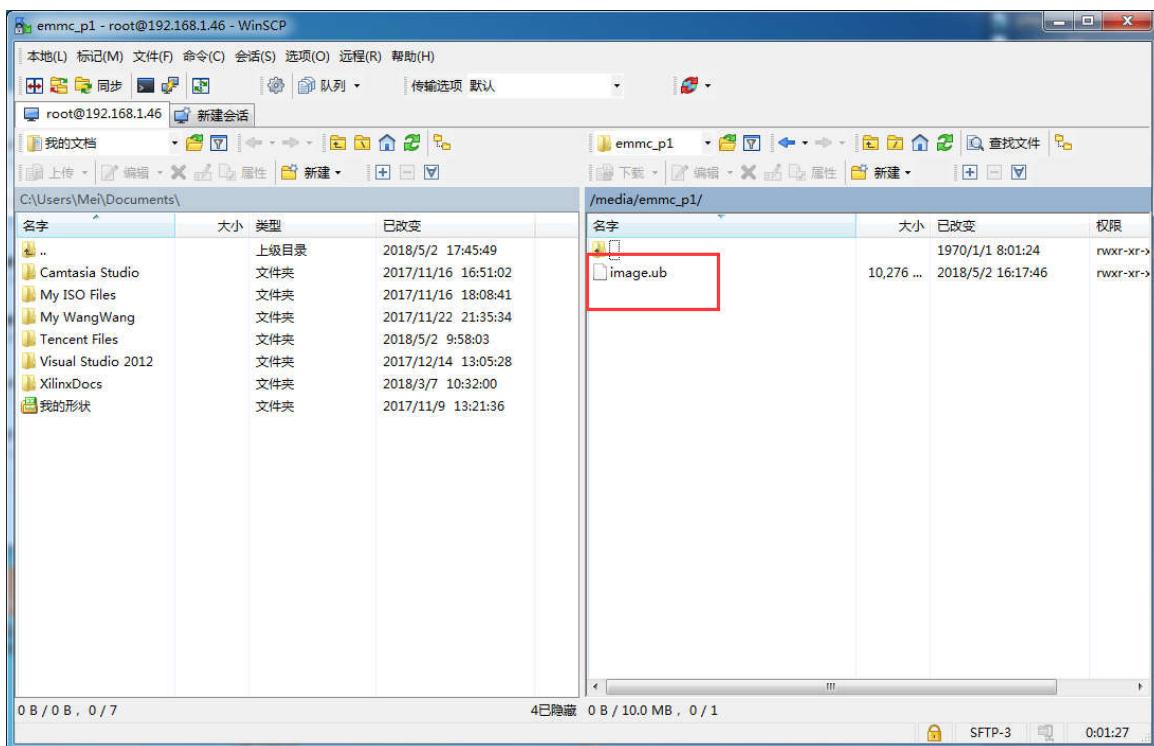
- 3) 选择是



- 4) 找到/media/emmc_p1 文件夹



5) 可以直接把我们要的文件拖拽到目标文件夹中



6) image.ub 文件已经放入 EMMC , 下面需要把 BOOT.bin 文件烧写到 QSPI

28.4 使用批处理文件快速烧写 QSPI

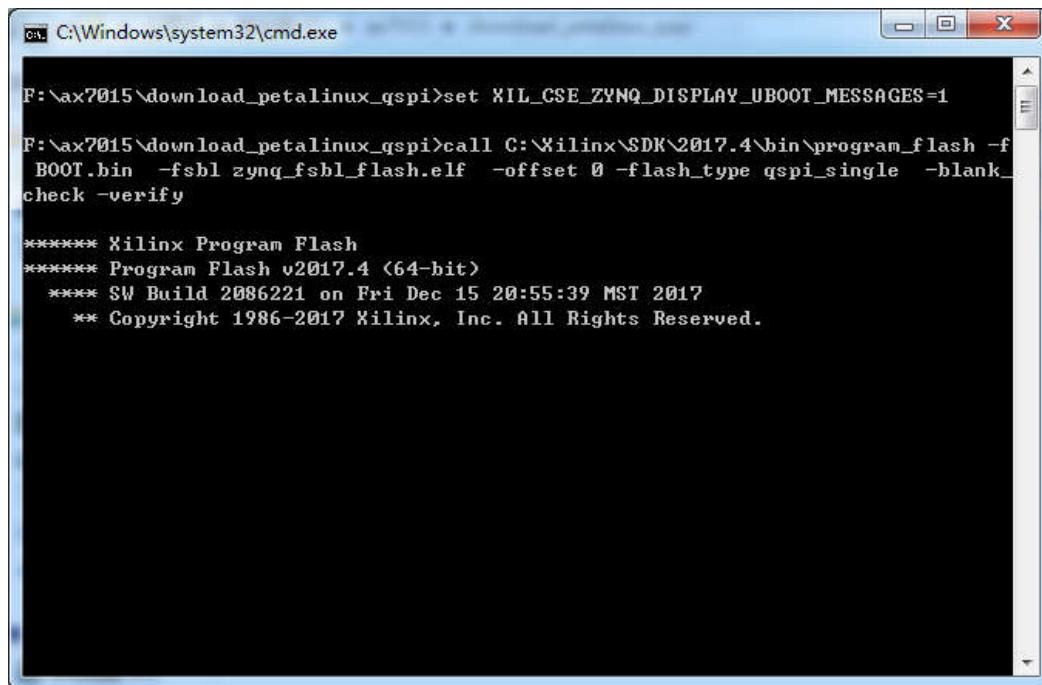
- 1) 新建一个 program_qspi.txt 文本文件，扩展名改为 bat，内容填写如下，其中 set XIL_CSE_ZYNQ_DISPLAY_UBOOT_MESSAGES=1 设置显示烧写过程中的 uboot 打印信息，C:\Xilinx\SDK\2017.4\bin\program_flash 为我们工具路径，按照安装路径适当修改，-f 为要烧写的文件，-fsbl 为要烧写使用的 fsbl 文件，-blank_check -verify 为校验选项。

```
set XIL_CSE_ZYNQ_DISPLAY_UBOOT_MESSAGES=1
call C:\Xilinx\SDK\2017.4\bin\program_flash -f BOOT.bin -fsbl
zynq_fsbl_flash.elf -offset 0 -flash_type qspi_single -blank_check
-verify
pause
```

- 2) 把要烧录的 BOOT.bin、fsbl、bat 文件放在一起

名称	修改日期	类型	大小
BOOT.BIN	2018/3/27 20:25	BIN 文件	14,879 KB
program_qspi.bat	2018/3/22 11:04	Windows 批处理...	1 KB
zynq_fsbl_flash.elf	2018/1/5 22:04	ELF 文件	181 KB

- 3) 插上 JTAG 线后上电，双击 bat 文件即可烧写 flash。



28.5 测试 QSPI 和 EMMC 启动

- 1) 把启动模式设置到 QSPI，然后启动开发板，登录系统，可以看到系统正常系统。

```
Public key portion is:  
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQ Cf0mQ8fXbQhKa576z907iSARyIEikjkSy0++H5H/NG  
9PKHkAf/7n3oTXK6T5hSSe7CqFn1k9y7SrS4RvUuQYLyjf6Y+huvq8jmny0UdFe4kqedgy9FgEaV0RQ  
aofoE4QnvLId9Uf4uiR1JqbP8uNLdCHGZtzYhn44qRHCNIkw/GAryk2DFZdaIRzRjh2ODLc20req3zp/  
ewo62h/M22DYTsLY1eHfBtCsDaKnDpvhpr5/5R194JiH4JYoiG0ZH+8Bg2JB0H08d1XYIpfwKhgkJfmH  
rSBk3ALc3rb06hAiYrLrkedY+0hUobACQomLpiiAQ44feTzZU5TB0pcjDHRh root@ax_peta  
Fingerprint: md5 d0:37:2f:0e:19:a4:5a:67:96:64:3c:41:9d:20:65:c4  
dropbear.  
hwclock: can't open '/dev/misc/rtc': No such file or directory  
Starting syslogd/klogd: done  
Starting tcf-agent: OK  
  
PetaLinux 2017.4 ax_peta /dev/ttys0  
  
ax_peta login: root  
Password:  
root@ax_peta:~# df  
Filesystem 1K-blocks Used Available Use% Mounted on  
devtmpfs 503284 4 503280 0% /dev  
tmpfs 515104 116 514988 0% /run  
tmpfs 515104 44 515060 0% /var/volatile  
/dev/mmcblk1p1 3139576 10280 3129296 0% /run/media/mmcblk1p1  
/dev/mmcblk1p2 4286636 8760 4037084 0% /run/media/mmcblk1p2  
root@ax_peta:~#
```