

版本信息：  
版本  
REV2018  
时间  
04/12/2017

# ZYNQ 修炼秘籍

## 基于米联客系列开发板

第二季 基于 ZYNQ 的 SOC 入门基础

电子版自学资料

常州一二三电子科技有限公司  
溧阳米联电子科技有限公司  
版权所有  
米联客学院 03QQ 群： 516869816  
米联客学院 03QQ 群： 543731097  
米联客学院 02QQ 群： 86730608  
米联客学院 01QQ 群： 34215299



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

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

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

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

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

#### 版权声明：

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

#### 技术支持：

版主大神们都等着大家去提问--电子资源论坛 [www.osrc.cn](http://www.osrc.cn)

微信公众平台：电子资源论坛



# 目录

<b>ZYNQ 修炼秘籍.....</b>	<b>1</b>
<b>目录.....</b>	<b>4</b>
<b>【第二季】ZYNQ SOC 入门基础 共 16 课时 .....</b>	<b>8</b>
<b>CH01_Hello World 实验 .....</b>	<b>9</b>
1.1 最小系统分析.....	9
1.2 硬件电路分析.....	9
1.3 创建一个 VIVADO 工程.....	10
1.4 导出 SOC 硬件到 SDK.....	20
1.5 Hello World 实验.....	21
1.6 MemTest 内存测试程序.....	30
1.7 DRAMTest 内存测试程序 .....	32
1.8 LWIP 协议对千兆网口测试 .....	33
1.9 使用快捷按钮调试.....	36
1.10 本章小结.....	36
<b>CH02_MIO 实验 .....</b>	<b>37</b>
2.1 GPIO 简介 .....	37
2.1.1 GPIO 的控制寄存器地址空间.....	38
2.1.2 MIO 内部构造分析 .....	41
2.1.3 EMIO 的特性.....	42
2.2 电路分析及实验预期.....	42
2.3 ZYNQ 核的添加及配置.....	42
2.4 新建 LED_Flash SDK 工程 .....	43
2.5 程序分析.....	47
2.6 本章小结.....	54
<b>CH03_EMIO 实验.....</b>	<b>55</b>
3.1 EMIO 和 MIO 的对比介绍 .....	55
3.2 电路分析与实验现象 .....	55
3.3 创建 VIVADO 工程.....	55
3.4 创建约束文件 .....	57
3.5 产生 bit 文件并导入到 SDK 中 .....	58
3.6 程序分析.....	64
3.7 本章小结.....	65
<b>CH04_User_IP 实验 .....</b>	<b>66</b>
4.1 创建 IP.....	66
4.2 调用自定义 IP .....	69
4.3 导入到 S D K .....	71
4.4 本章小结.....	72
<b>CH05_UBOOT 实验 .....</b>	<b>73</b>

5.1 什么是固化.....	73
5.2 固化的流程.....	73
5.3 固化准备.....	73
5.4 zynq 的从 SD 卡的启动的过程.....	74
5.5 zynq 启动模式位的选择 .....	74
5.6 BOOT.bin 制作过程详解 .....	75
5.7 从 Quad-SPI 启动.....	84
5.8 本章小结.....	86
CH06_XADC 实验.....	87
6.1 实验概述.....	87
6.2 新建一个 VIVADO 工程.....	87
6.3 加载到 SDK.....	88
6.4 函数介绍.....	90
6.5 本章小结.....	90
CH07_ZYNQ PL 中断请求 .....	91
7.1 ZYNQ 中断介绍.....	91
7.1.1 ZYNQ 中断框图.....	91
7.1.2 ZYNQ CPU 软件中断 (SGI).....	92
7.1.3 ZYNQ CPU 私有端口中断.....	93
7.2 搭建硬件地址.....	95
7.3 加载到 SDK.....	97
7.4 程序分析.....	98
7.5 本章小结.....	106
CH08_ZYNQ 定时器中断实验 .....	107
8.1 中断原理.....	107
8.1.1 软件中断(SGI).....	107
8.1.2 共享中断 SPI.....	107
8.1.3 私有中断 (PPI) .....	108
8.1.4 私有定时器.....	108
8.2 搭建硬件工程.....	108
8.3 加载到 SDK.....	110
8.4 程序分析.....	112
8.5 本章小结.....	118
CH09_UART 串口中断实验.....	119
9.1 加载到 SDK.....	119
9.2 程序分析.....	120
9.3 本章小结.....	128
CH10_User GPIO 实验 .....	129
10.1 创建 IP.....	129
10.2 搭建硬件工程.....	148
10.3 加载到 SDK.....	149
10.4 程序分析.....	150
10.4 本章小结.....	151

CH11_ZYNQ 软硬调试高级技巧 .....	152
11.1 方案框架.....	152
11.2 硬件工程搭建.....	152
11.3 加载到 SDK.....	161
11.4 本章小结.....	165
CH12_AXI_Lite 总线详解.....	166
12.1 前言.....	166
12.2 AXI 总线与 ZYNQ 的关系.....	166
12.3 AXI 总线和 AXI 接口以及 AXI 协议.....	166
12.3.1 AXI 总线概述.....	166
12.3.2 AXI 接口介绍.....	167
12.3.3 AXI 协议概述.....	168
12.3.4 AXI 协议之握手协议.....	169
12.4 AXI4-Lite 详解 .....	171
12.4.1 AXI4-Lite 源码查看 .....	171
12.4.2 AXI-Lite 源码分析 .....	174
12.5 观察 AXI4-Lite 总线信号 .....	181
12.6 加载到 SDK.....	185
12.7 本章小结.....	188
CH13_AXI_PWM 实验.....	189
13.1 自定义 IP 的封装 .....	189
13.2 用户 IP 的修改 .....	191
13.3 搭建硬件工程.....	200
13.4 加载到 SDK.....	203
13.5 程序分析.....	205
13.6 本章小结.....	206
CH14_EMIO_OLED 实验 (MZ7XB 可跳过本章) .....	207
14.1 板载 OLED 硬件原理 .....	207
14.1.1 硬件电路简析.....	207
14.1.2 SSD1306 简介 .....	208
14.2 OLED 驱动开发思路解析 .....	209
14.2.1 SPI 接口 .....	209
14.2.2 SSD1306 控制 .....	209
14.2.2 Frame Buffer 显示机制 .....	213
14.2.3 像素操作函数.....	213
14.2.4 其他 API 的实现 .....	213
14.3 OLED 驱动方案实现 .....	214
14.4 点阵式 OLED 显示原理 .....	214
14.4.1 OLED 简介 .....	214
14.4.2 点阵式显示设备显示原理 .....	214
14.4.3 字模的获取 .....	215
14.5 硬件搭建.....	219
14.6 导入到 SDK.....	220

14.7 本章小结.....	221
CH15_AXI_OLED 实验(MZ7XB 可跳过本章).....	222
15.1 自定义 IP 的封装 .....	222
15.2 SSD1306_OLED_ML 用户 IP 的修改 .....	224
15.3 OLED 硬件控制器关键状态机 .....	262
15.4 硬件工程搭建.....	274
15.5 导入到 SDK.....	275
15.6 本章小结.....	275
CH16 等精度频率计实验 .....	276
16.1 等精度频率计原理.....	276
16.1.1 引言 .....	276
16.1.2 频率测量原理.....	276
16.1.3 脉冲计数法.....	276
16.1.4 周期测频法.....	277
16.1.5 多周期同步测频原理及误差分析 .....	277
16.2 等精度频率计设计.....	278
16.2.1 PS 寄存器功能划分 .....	278
16.2.2 具体实现.....	279
16.2.3 频率计 PL 部分代码设计 .....	279
16.3 硬件工程搭建.....	281
16.4 导入到 SDK.....	282
16.5 误差分析.....	283
16.6 本章小结.....	283

## 【第二季】ZYNQ SOC 入门基础 共 16 课时

第二季课程共计 16 课时。学习重点包括 MIO、EMIO 的使用，中断资源的使用，熟悉了解 ZYNQ 中断的库函数，学会推导 XILINX SDK 中断函数的构架，掌握 AXI-LITE 总线协议，掌握自定义 IP 的创建，封装。掌握 VIVADO 软件的调试技巧等。

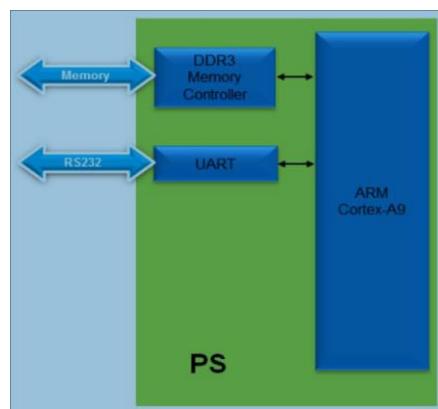
## CH01\_Hello World 实验

ZYNQ 是一款 SOC 芯片，在前面第一季的学习当中，我们只是粗略的学习了 ZYNQ 的 PL 部分，对于 ZYNQ 最突出的功能，其内部的双核 Cortex-A9 内核并未使用到。从本章开始，我们就将开始学习 ZYNQ 的 SOC 学习。

本章将带领大家搭建一个最小系统，在此基础上，对我们的板子上的一些硬件进行测试，通过本章，你将掌握如何创建一个 SOC 工程与 SDK 软件的基本使用。

### 1.1 最小系统分析

这张图展示了我们需要构建的最小系统。并且下面的嵌入式实验会基于这个最小系统进行添加外设。



本实验中将会只使用到 PS 部分资源包括了 ARM Cortex-A9、DDR3 内存、一个 UART 串口。这就是我们的最小系统。首先我们程序会加载到 DDR 内存中，然后 CPU 一条一条执行，那么执行的情况我们可以通过串口打印观察。

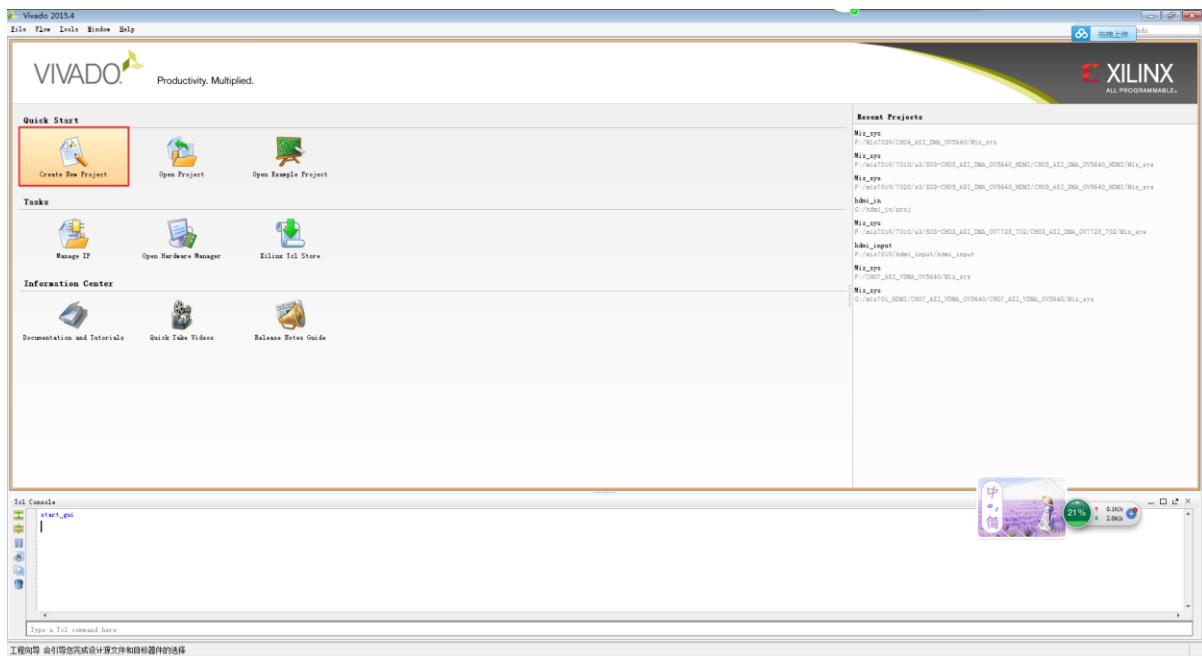
### 1.2 硬件电路分析

MIZ7035 开发板 PS 端 DDR 容量为 1GB，PL 端 DDR 为 1GB，本实验只使用 PS 端的 DDR。下图为 MIZ7035 的核心板。

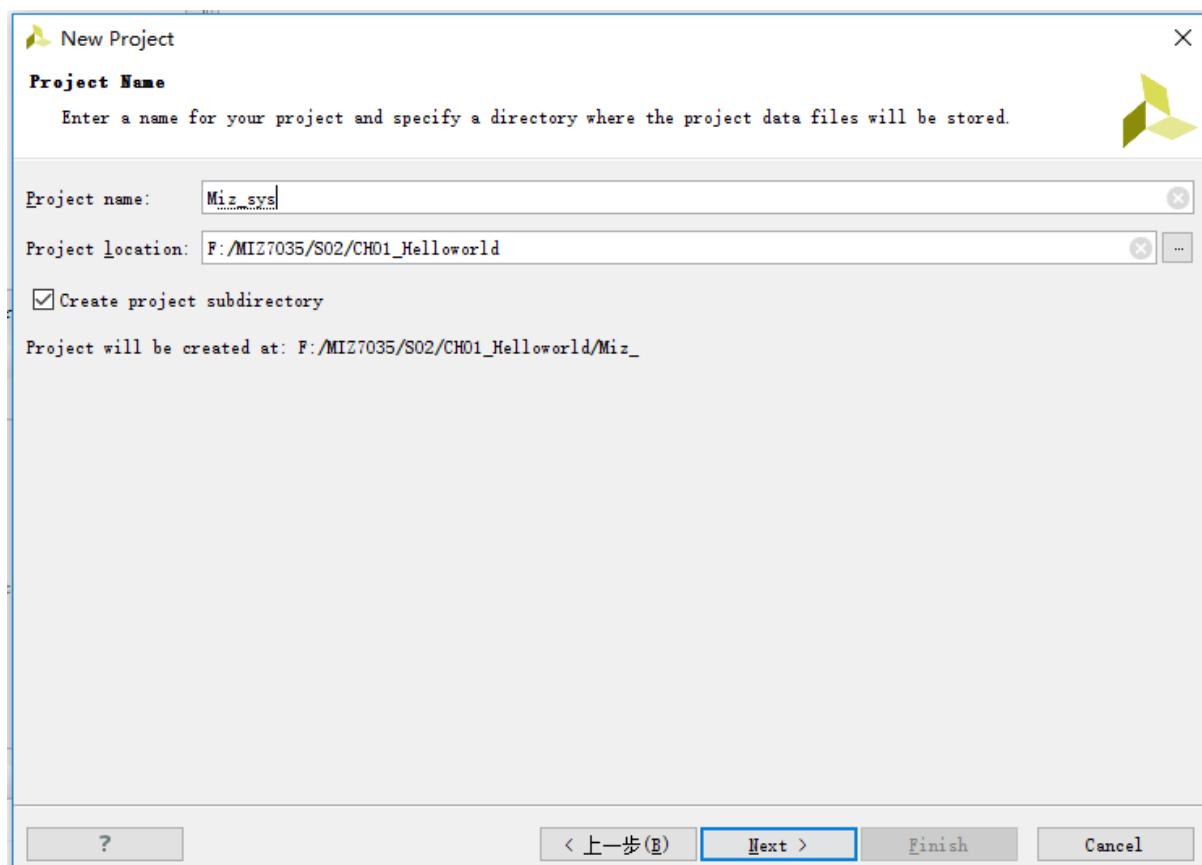


## 1.3 创建一个 VIVADO 工程

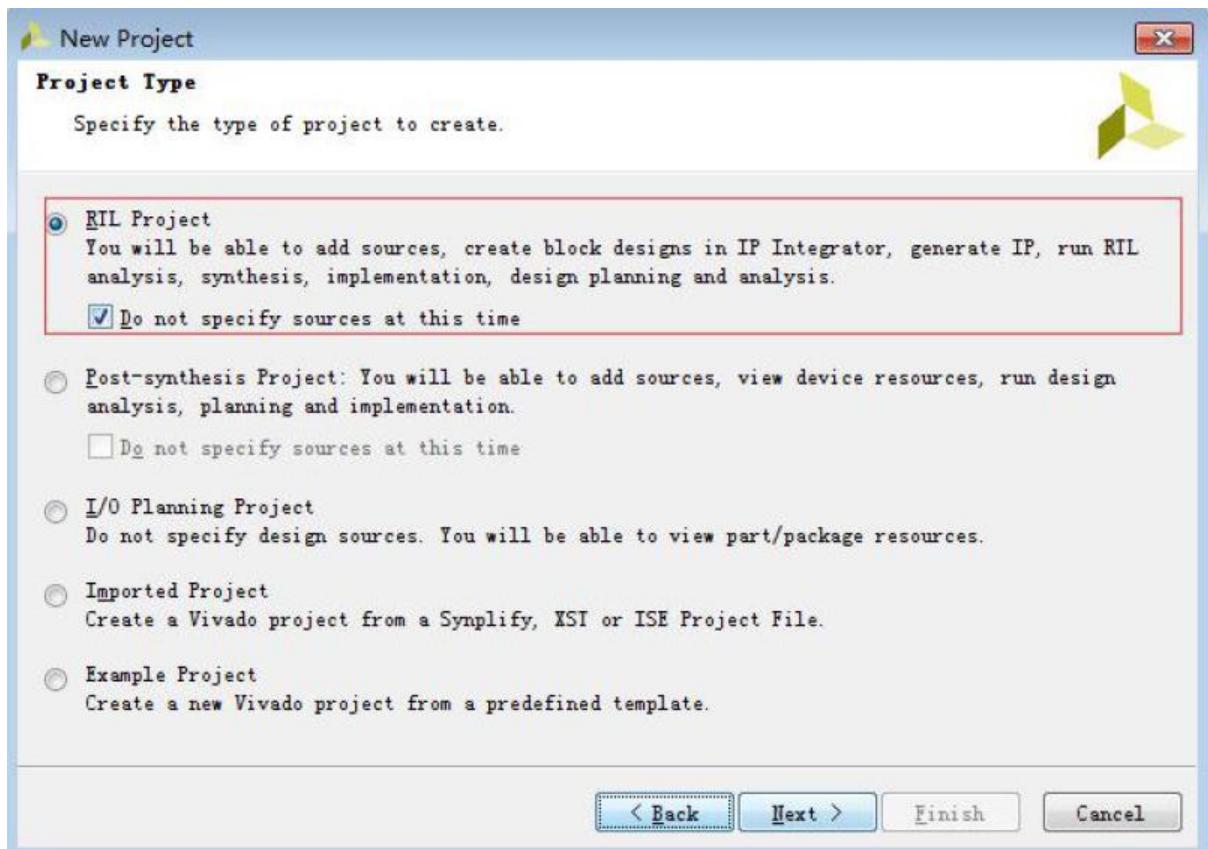
Step1：在打开的VIVADO软件界面，单击Create New Project。



Step2：单击NEXT，在弹出的窗口中输入工程名和选择保存路径，然后单击Next。

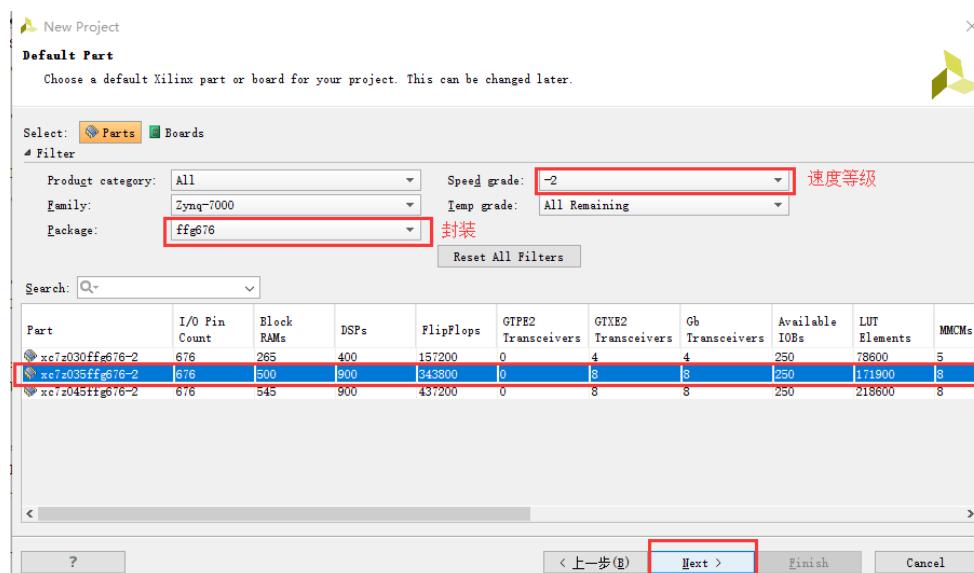


Step3：



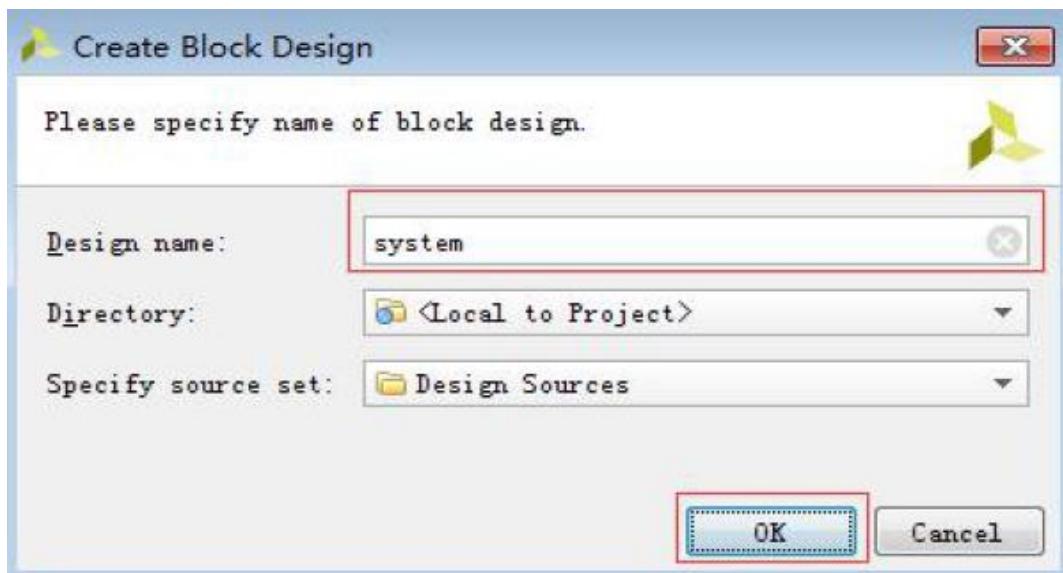
Step4: 选择芯片类型, 然后单击OK。

MiZ7035 如下图所示设置:

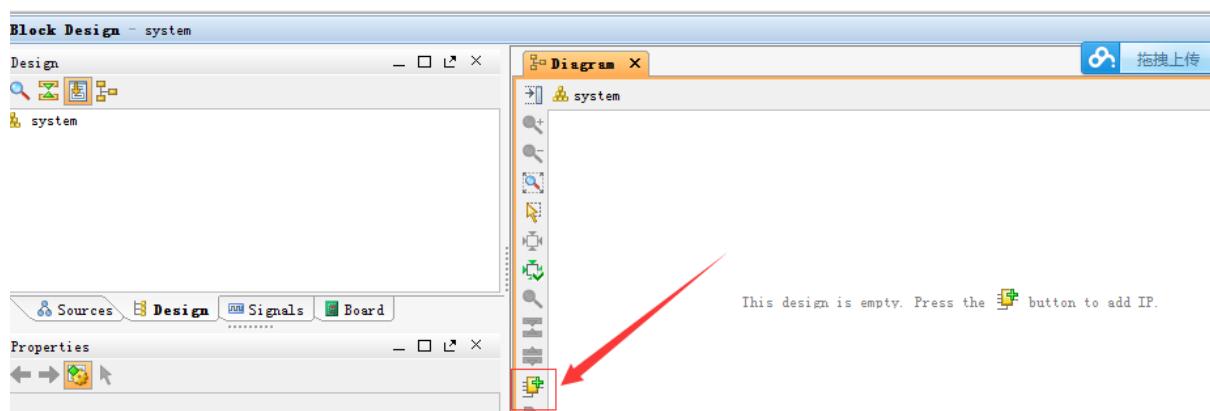


Step5: 单击Finish完成工程的创建

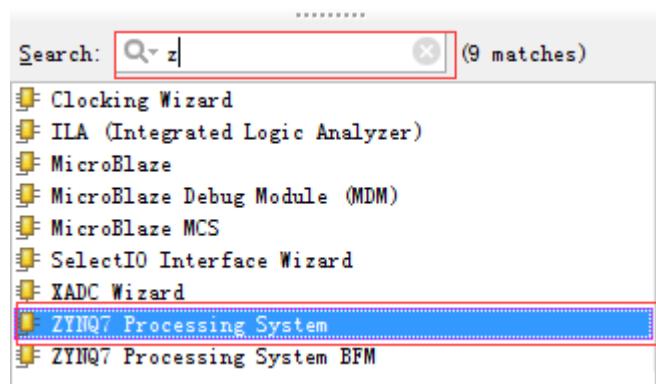
Step6: 单击Create Block Design, 输入System。



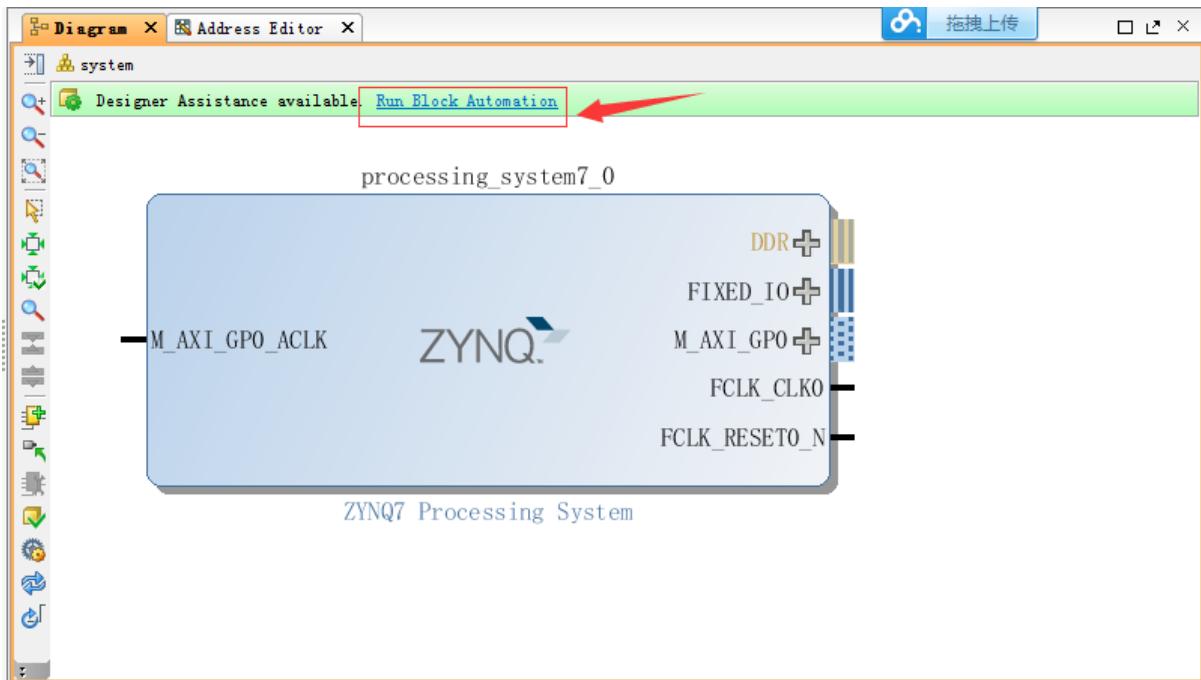
Step7: 单击下图中 添加IP按钮



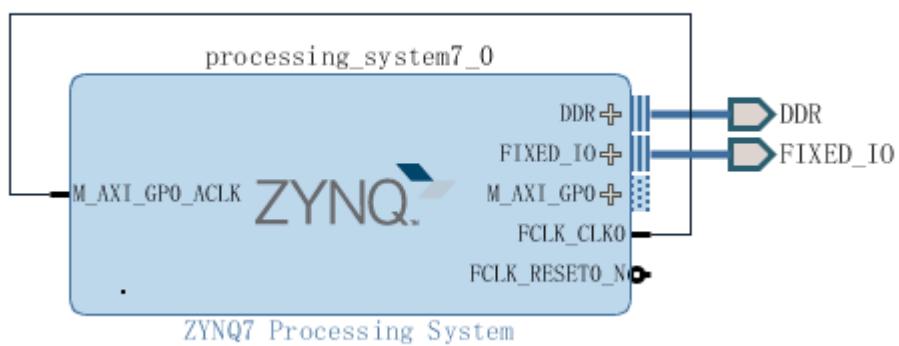
Step8: 搜索单词z选择ZYNQ7 Processing System, 然后双击



Step9: 添加进来了ZYNQ CPU IP, 然后单击Run Block Automation , 直接单击OK。

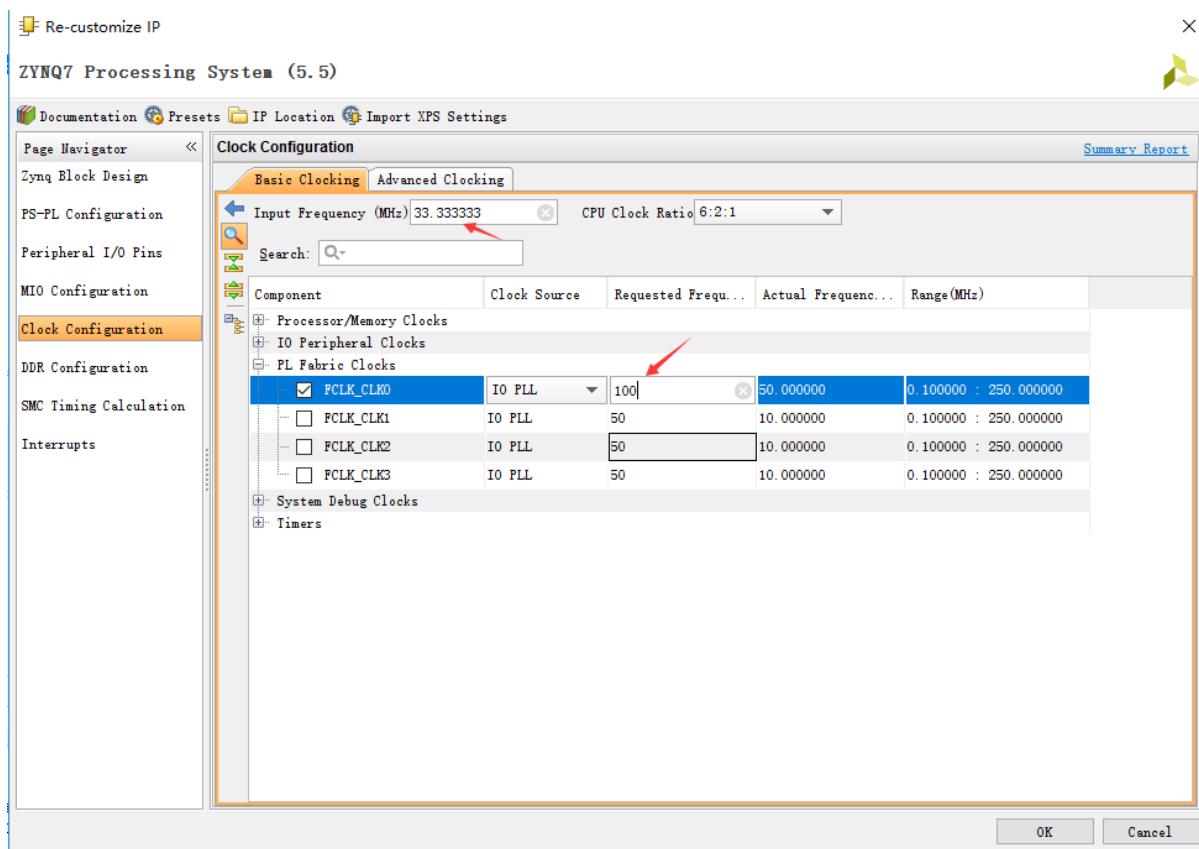


Step10：在Block文件中，我们进行连线，将鼠标放在引脚处，鼠标变成铅笔后进行拖拽，连线如下图所示。连线的作用就是把PS的时钟可以接入PL部分，当然这里我们暂时用不到PL部分的资源。



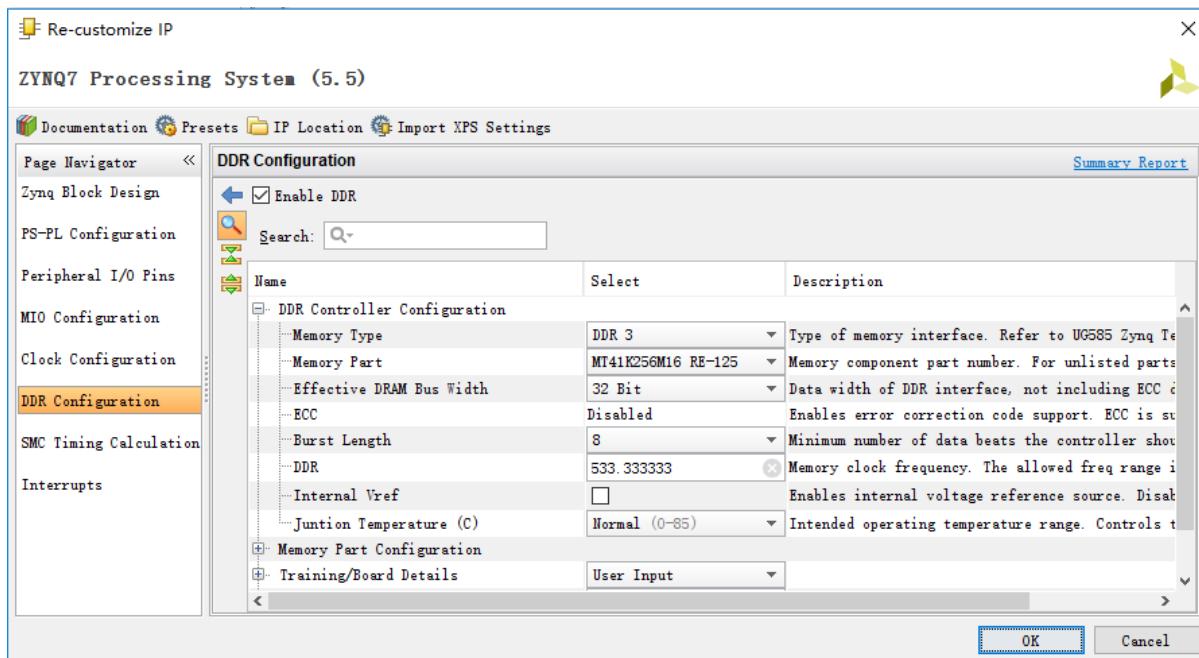
Step11：双击ZYNQ CPU IP，对其进行设置，使其对应我们的硬件设置。在此部分，我们需要做的就是修改时钟频率，内存类型和接口输出。需要注意的是，如果时钟频率与内存类型与我们的硬件不一致时，SDK中的程序会崩溃，运行不过来。这在后续的调试中，是一个小技巧。正确的配置是成功的必要条件。

**PS时钟配置：**

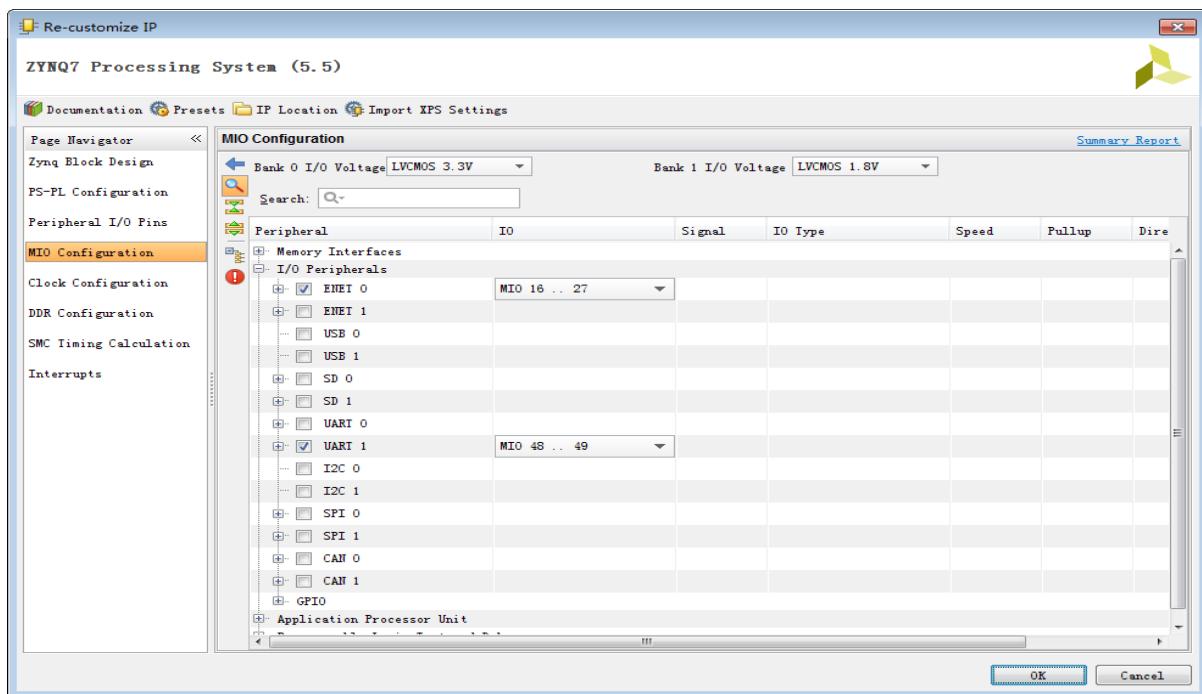


内存型号配置 (PS: MiZ7035开发板内存型号配置为MT41K256M16 RE-125) :

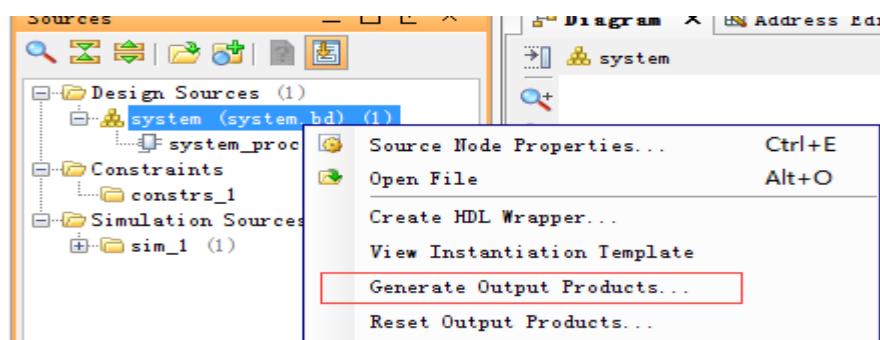
MiZ7035内存型号配置如下:



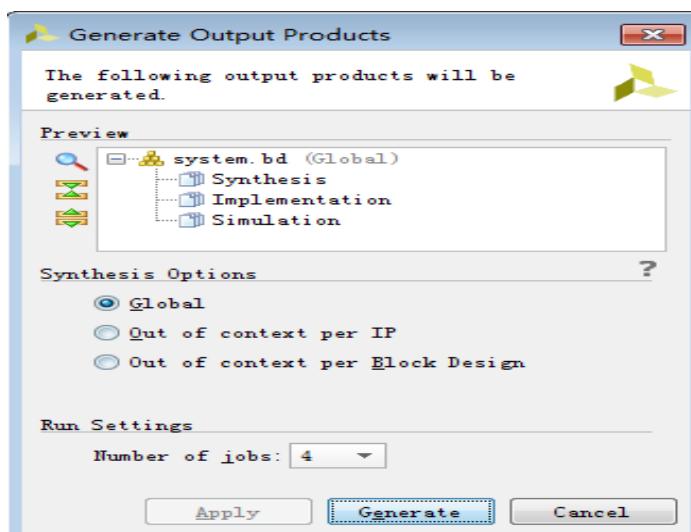
Step12: 设置外扩接口，之后点击OK。



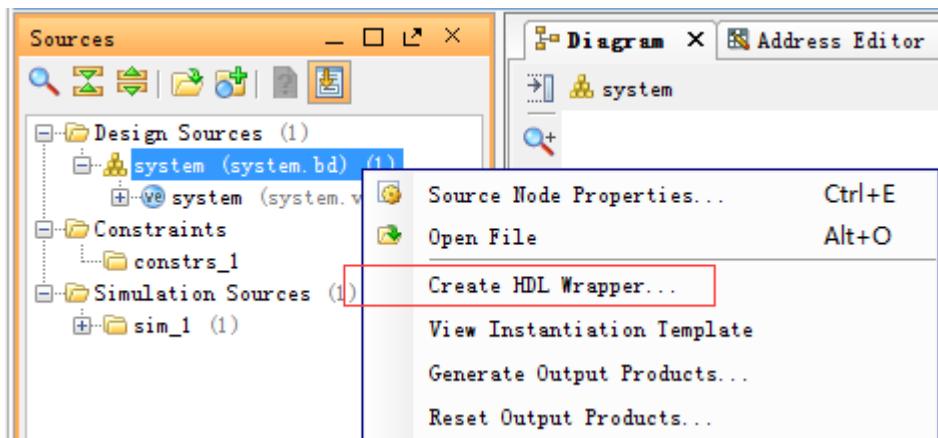
Step13: 右击 system.bd, 单击Generate Output Products。



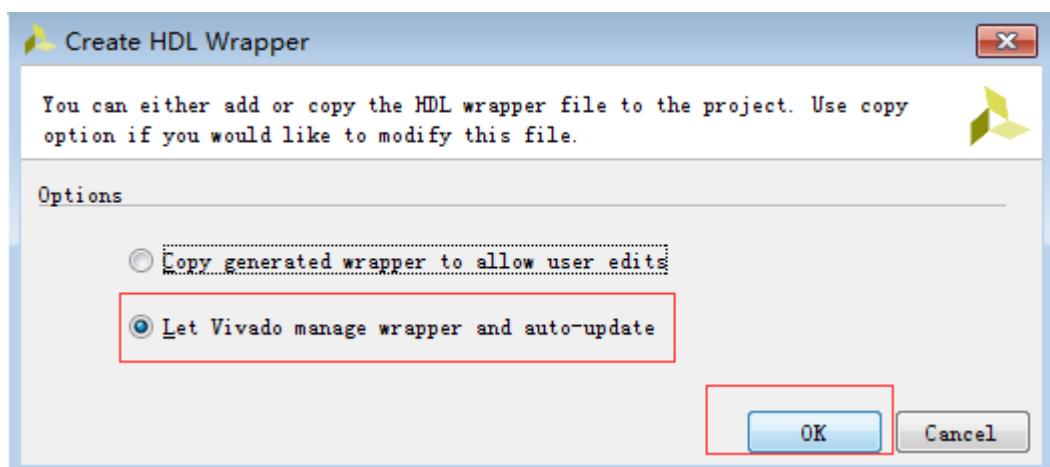
Step14: 支部操作会产生执行、仿真、综合的文件，可以看出来最后的硬件设计步骤还是回到了我们前面的FPGA开发上来了。



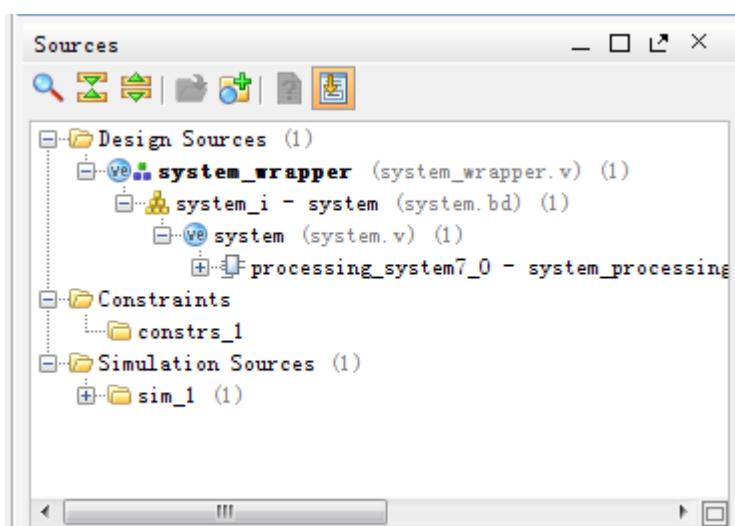
Step15:右击 system.bd 选择 Create HDL Wrapper 这步的作用是产生顶层的 HDL 文件



Step16:选择 Leave Let Vivado manager wrapper and auto-update 然后单击 OK



Step17:之后我看下源码的层次结构，可以看到 system\_wrapper.v 就是顶层文件，调用了 CPU.



Step18:查看 system\_wrapper.v 源码

```
//Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.  
//-----  
//Tool Version: Vivado v.2015.4 (win64) Build 1412921 Wed Nov 18 09:43:45 MST 2015  
//Date      : Thu Mar 24 22:07:47 2016  
//Host      : PC201603040001 running 64-bit Service Pack 1  (build 7601)  
//Command   : generate_target system_wrapper.bd  
//Design    : system_wrapper  
//Purpose   : IP block netlist  
//-----  
`timescale 1 ps / 1 ps  
  
module system_wrapper  
(DDR_addr,  
 DDR_ba,  
 DDR_cas_n,  
 DDR_ck_n,  
 DDR_ck_p,  
 DDR_cke,  
 DDR_cs_n,  
 DDR_dm,  
 DDR_dq,  
 DDR_dqs_n,  
 DDR_dqs_p,  
 DDR_odt,  
 DDR_ras_n,  
 DDR_reset_n,  
 DDR_we_n,  
 FIXED_IO_ddr_vrn,  
 FIXED_IO_ddr_vrp,  
 FIXED_IO_mio,  
 FIXED_IO_ps_clk,
```

```
    FIXED_IO_ps_porb,  
    FIXED_IO_ps_srstb);  
  
inout [14:0]DDR_addr;  
inout [2:0]DDR_ba;  
inout DDR_cas_n;  
inout DDR_ck_n;  
inout DDR_ck_p;  
inout DDR_cke;  
inout DDR_cs_n;  
inout [3:0]DDR_dm;  
inout [31:0]DDR_dq;  
inout [3:0]DDR_dqs_n;  
inout [3:0]DDR_dqs_p;  
inout DDR_odt;  
inout DDR_ras_n;  
inout DDR_reset_n;  
inout DDR_we_n;  
inout FIXED_IO_ddr_vrn;  
inout FIXED_IO_ddr_vrp;  
inout [53:0]FIXED_IO_mio;  
inout FIXED_IO_ps_clk;  
inout FIXED_IO_ps_porb;  
inout FIXED_IO_ps_srstb;  
  
wire [14:0]DDR_addr;  
wire [2:0]DDR_ba;  
wire DDR_cas_n;  
wire DDR_ck_n;  
wire DDR_ck_p;  
wire DDR_cke;  
wire DDR_cs_n;
```

```
wire [3:0]DDR_dm;
wire [31:0]DDR_dq;
wire [3:0]DDR_dqs_n;
wire [3:0]DDR_dqs_p;
wire DDR_odt;
wire DDR_ras_n;
wire DDR_reset_n;
wire DDR_we_n;
wire FIXED_IO_ddr_vrn;
wire FIXED_IO_ddr_vrp;
wire [53:0]FIXED_IO_mio;
wire FIXED_IO_ps_clk;
wire FIXED_IO_ps_porb;
wire FIXED_IO_ps_srstb;
system system_i
  (.DDR_addr(DDR_addr),
   .DDR_ba(DDR_ba),
   .DDR_cas_n(DDR_cas_n),
   .DDR_ck_n(DDR_ck_n),
   .DDR_ck_p(DDR_ck_p),
   .DDR_cke(DDR_cke),
   .DDR_cs_n(DDR_cs_n),
   .DDR_dm(DDR_dm),
   .DDR_dq(DDR_dq),
   .DDR_dqs_n(DDR_dqs_n),
   .DDR_dqs_p(DDR_dqs_p),
   .DDR_odt(DDR_odt),
   .DDR_ras_n(DDR_ras_n),
   .DDR_reset_n(DDR_reset_n),
   .DDR_we_n(DDR_we_n),
   .FIXED_IO_ddr_vrn(FIXED_IO_ddr_vrn),
```

```

.FIXED_IO_ddr_vrp(FIXED_IO_ddr_vrp),
.FIXED_IO_mio(FIXED_IO_mio),
.FIXED_IO_ps_clk(FIXED_IO_ps_clk),
.FIXED_IO_ps_porb(FIXED_IO_ps_porb),
.FIXED_IO_ps_srstb(FIXED_IO_ps_srstb));

endmodule

```

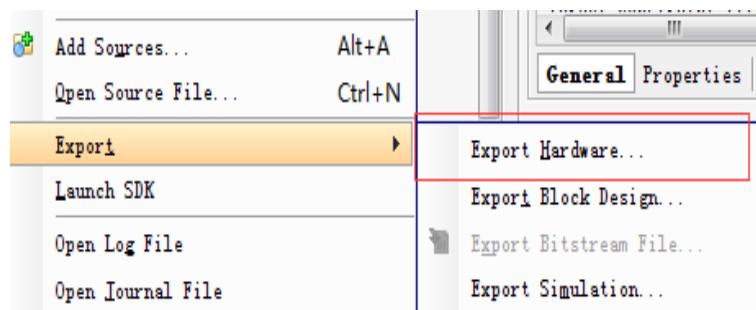
可以看到顶层文件的源码调用了 CPU 接口，所有外设的接口也都是通过顶层文件引出来的。



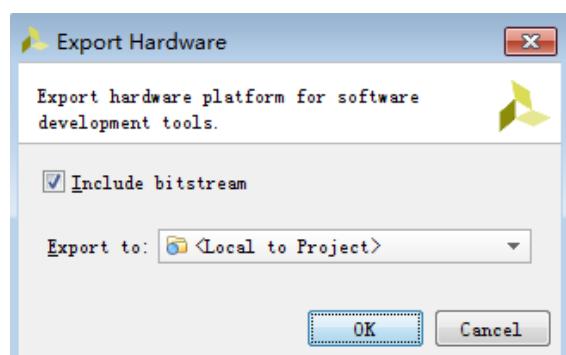
Step19:执行->产生 bit 文件。

## 1.4 导出 SOC 硬件到 SDK

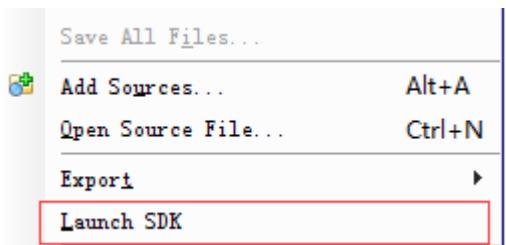
Step1: File->Export->Export Hardware



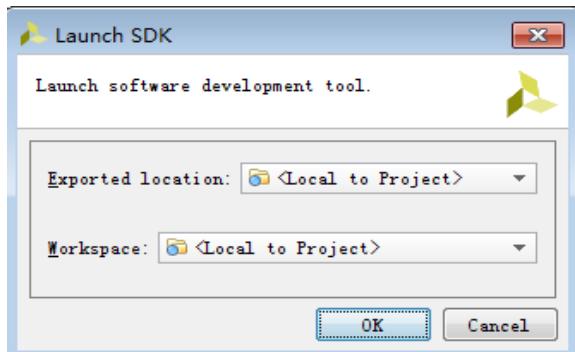
Step2:勾选 Include bitstream 直接单击 OK



Step3:File->Launch SDK 加载到 SDK



Step4:单击 OK

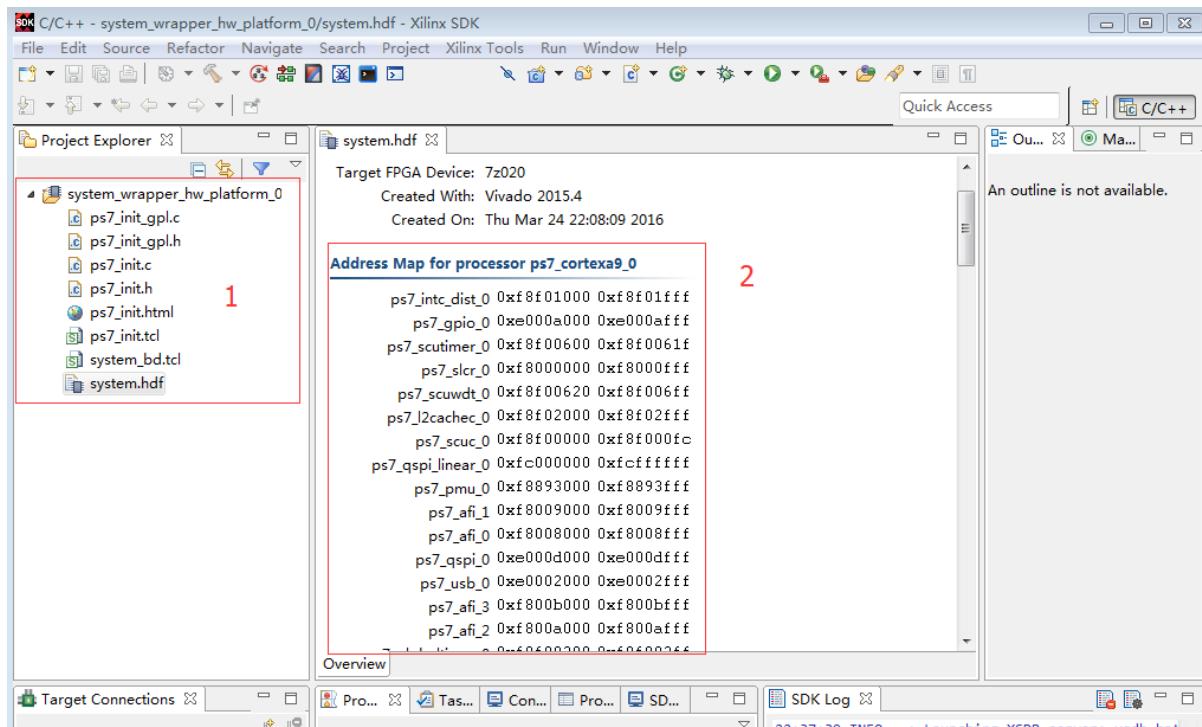


## 1.5 Hello World 实验

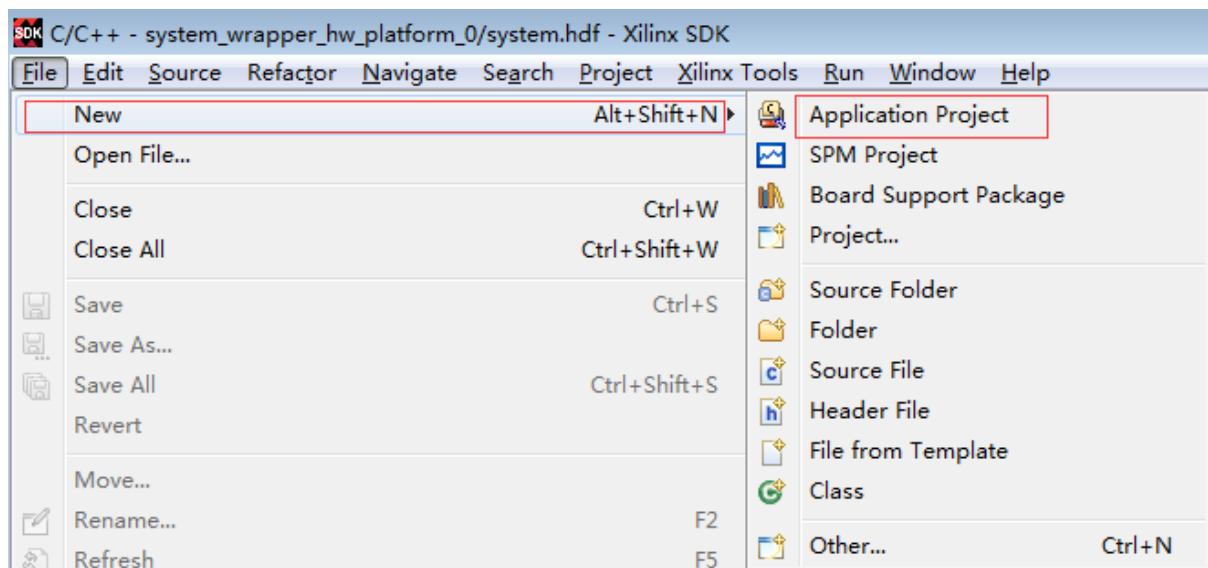
Step1:导出完成后如下图

1、硬件部分，这部分就是从 VIVADO 定制好的 SOC 硬件

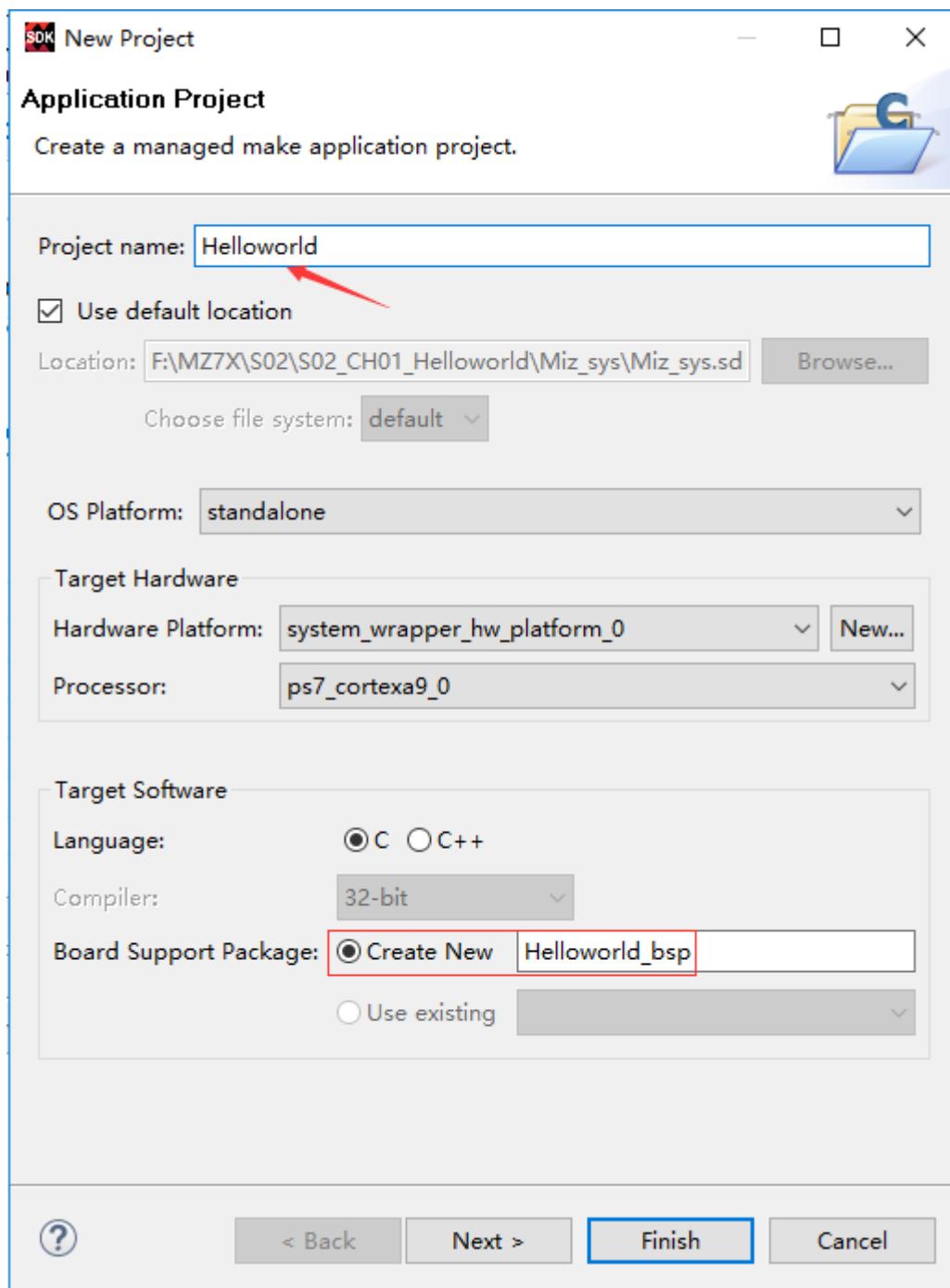
2、这部分是硬件的地址空间分配



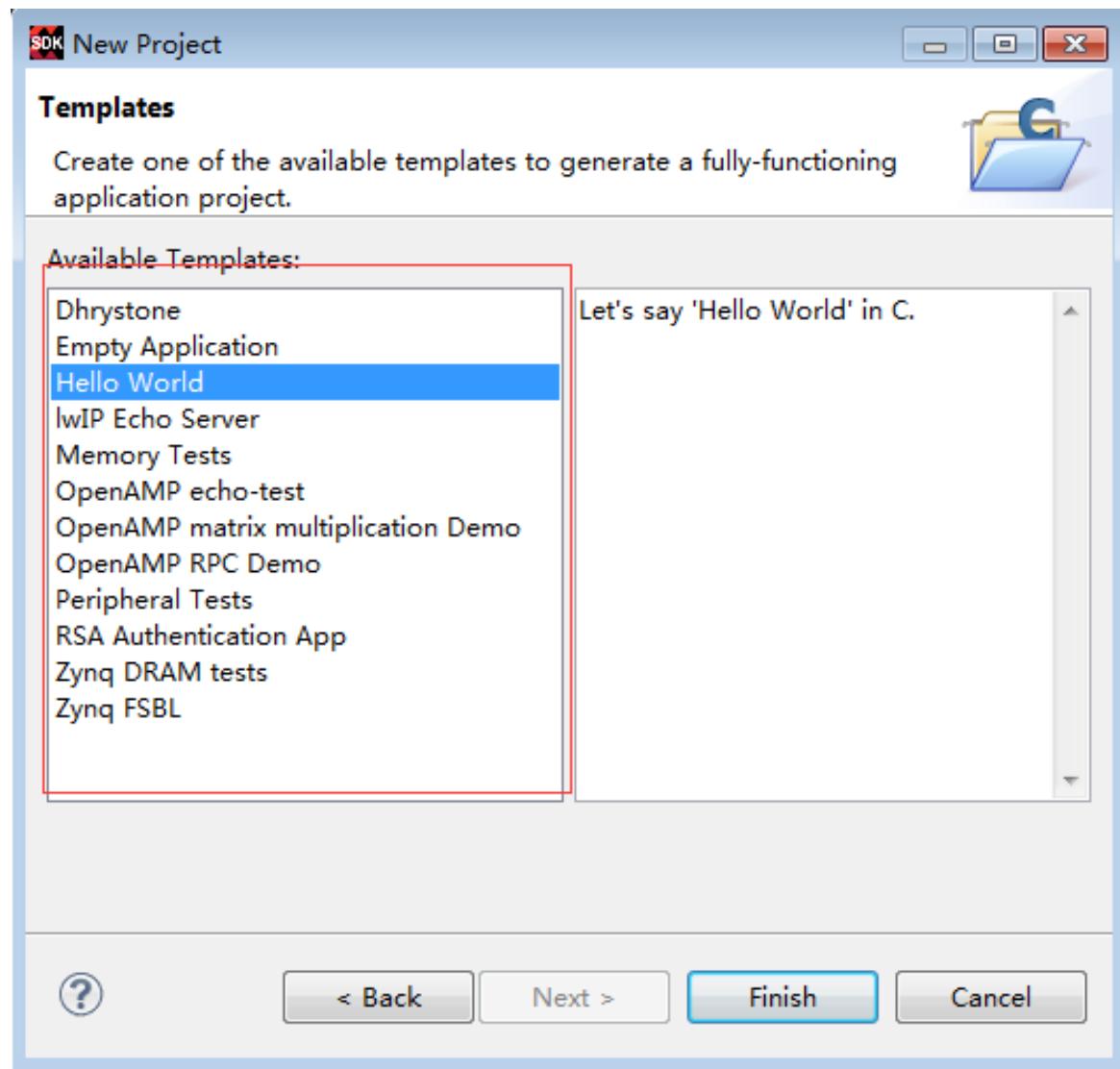
Step2:选择 File->New->Application Project



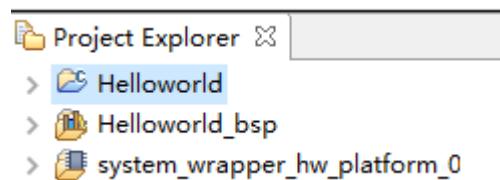
Step3:工程命名为 HelloWorld,然后单击 NEXT



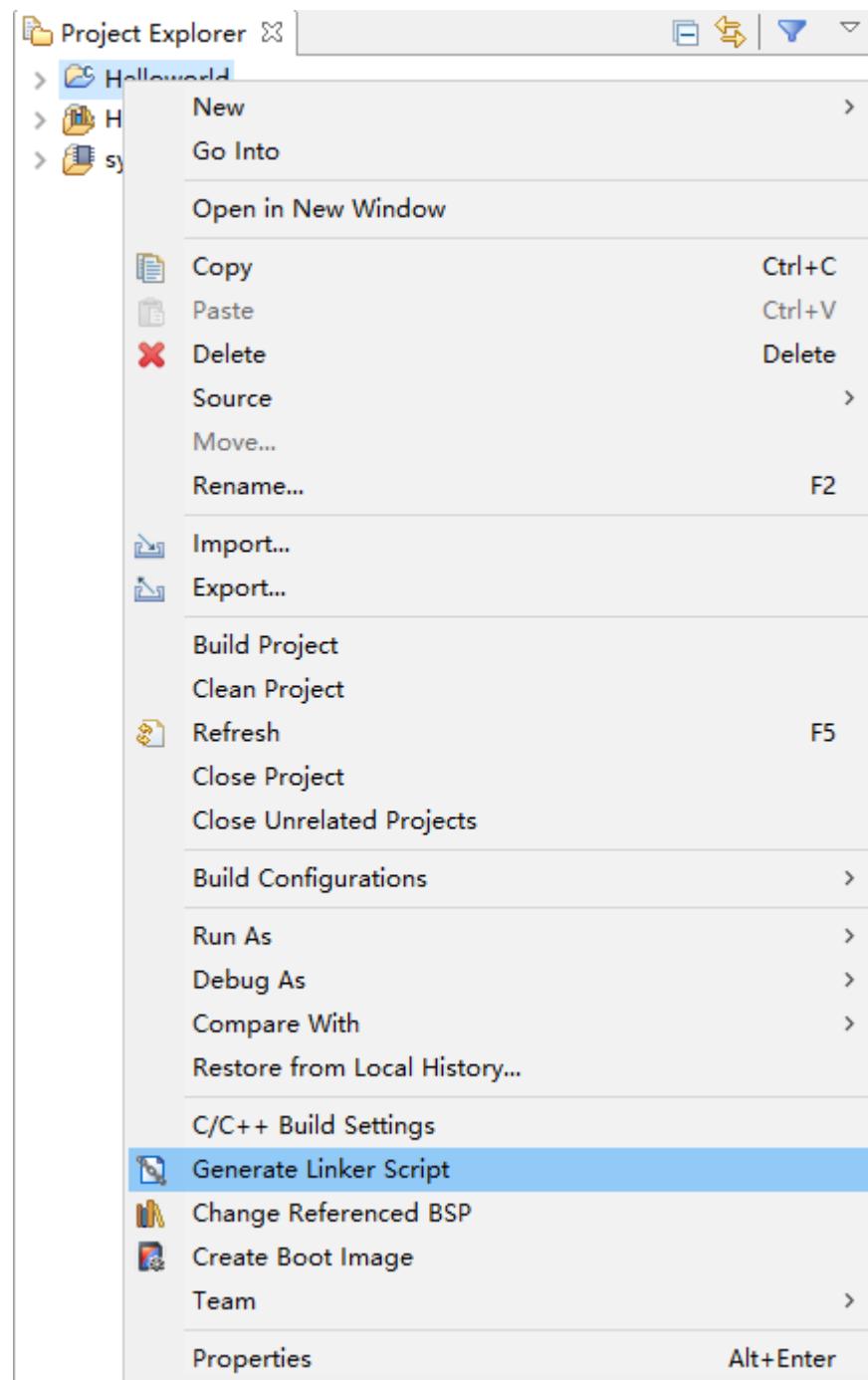
Step4:系统里面有很多自带的测试程序，本次就用自带的 Helloworld 程序做测试，单击 Finish



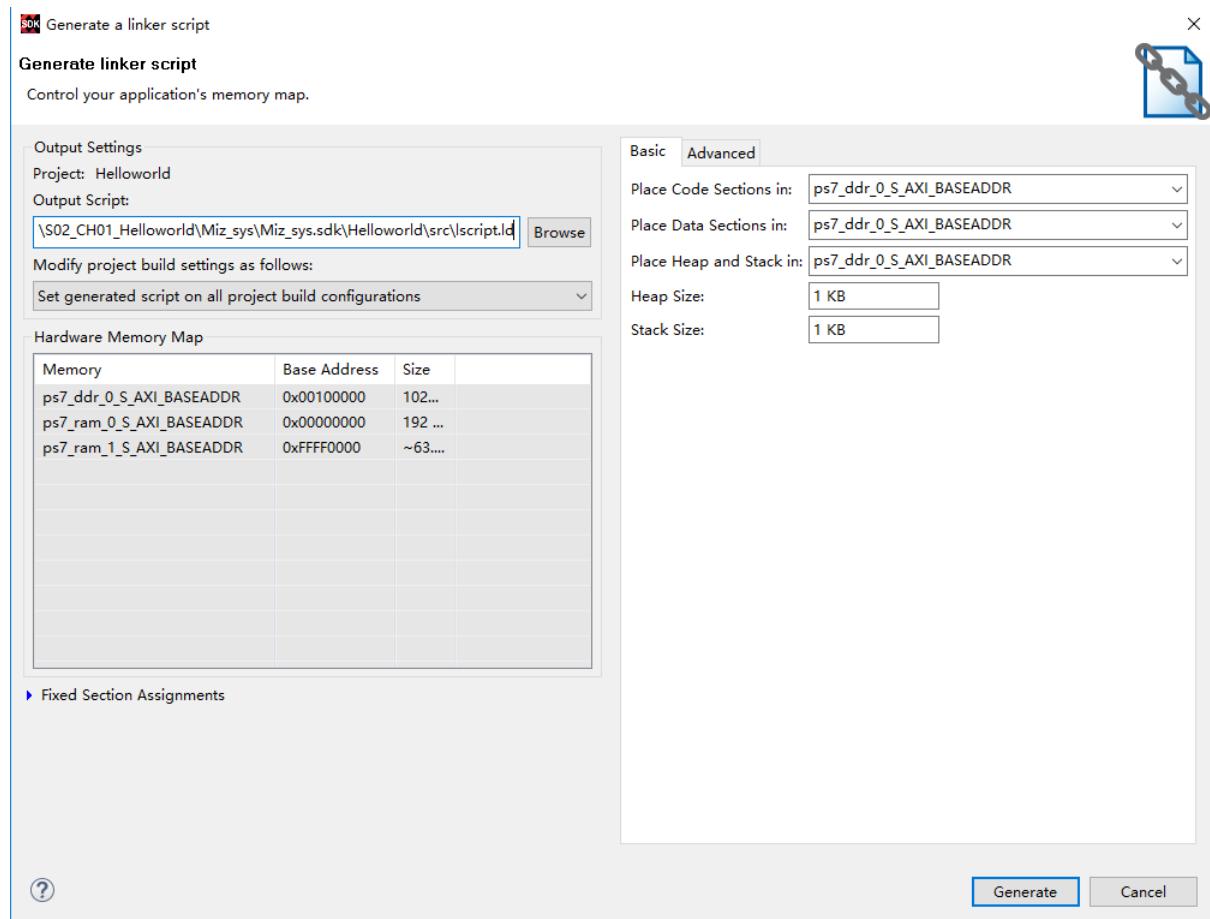
Step5:完成后



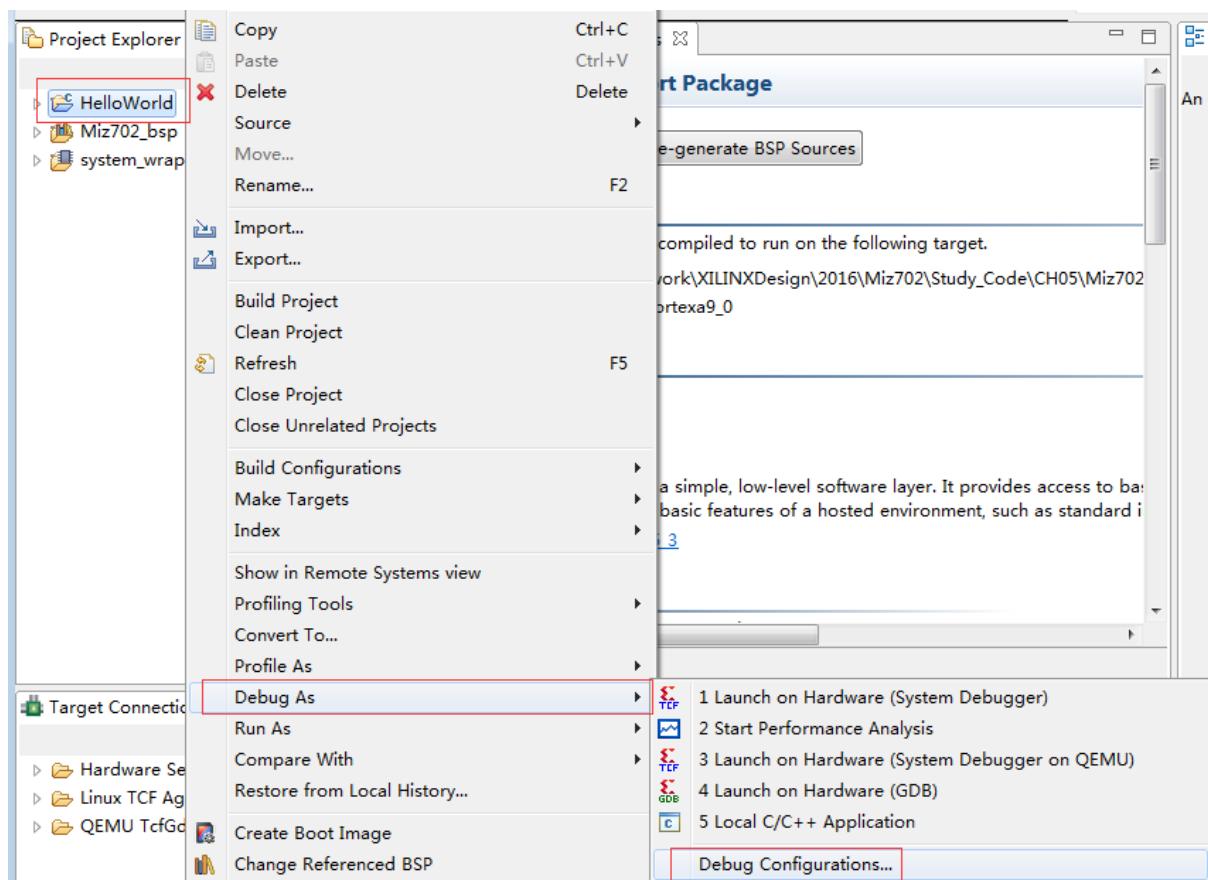
Step6:右击 HelloWorld->Generate linker Script



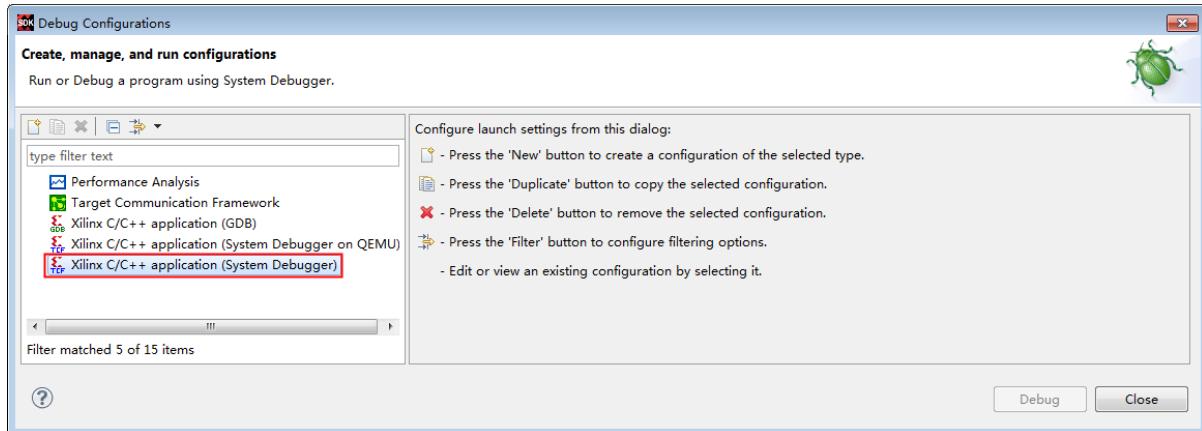
Step7:可以看到所有可用内存的情况，代码、数据、堆栈运行所在内存的情况。不做人为改动，关闭。



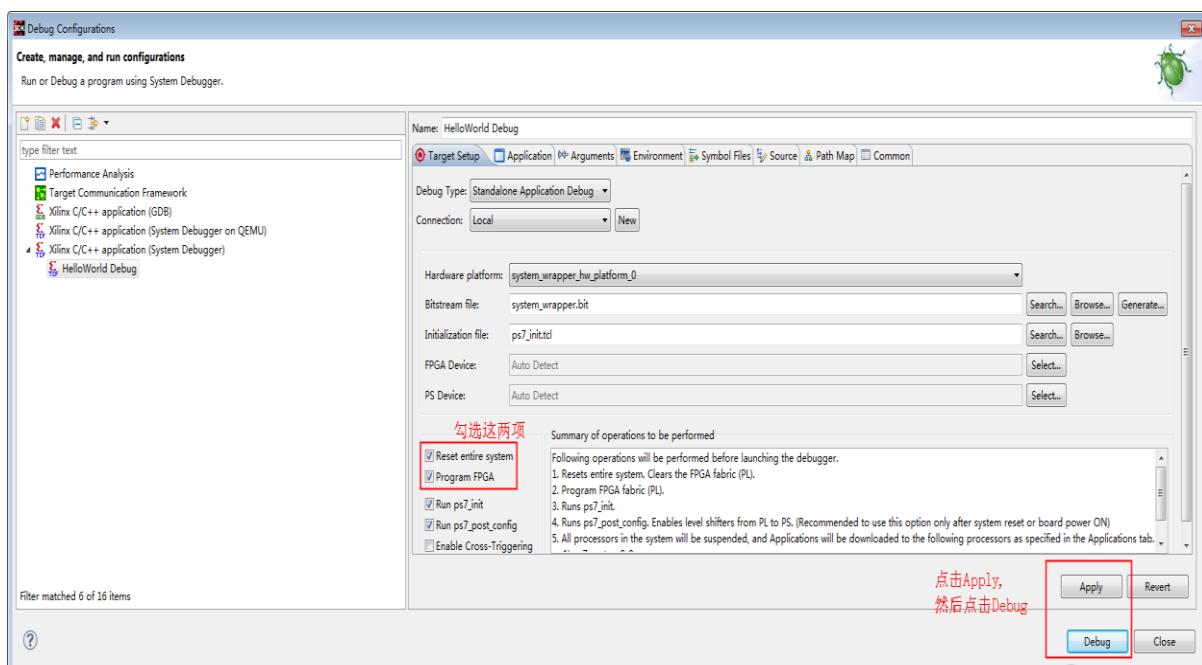
Step8:右击 HelloWorld->



Step9:双击这个位置新建

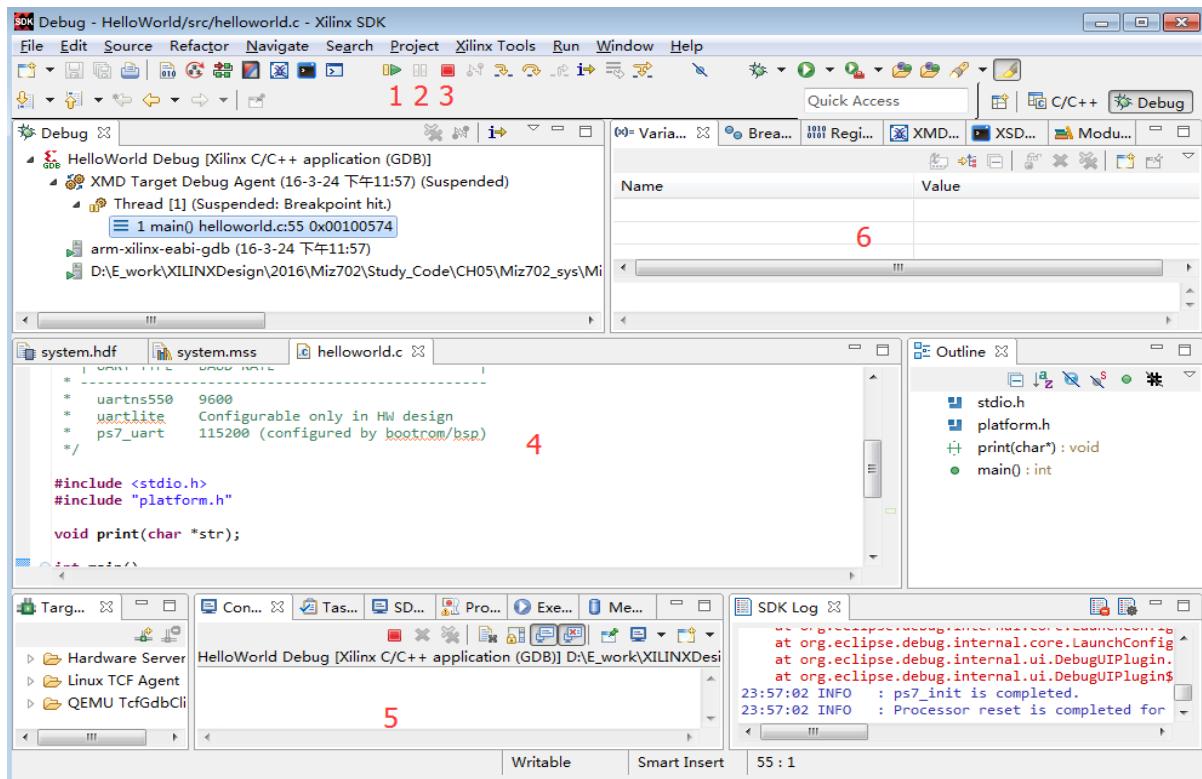


Step10:然后进行如下设置

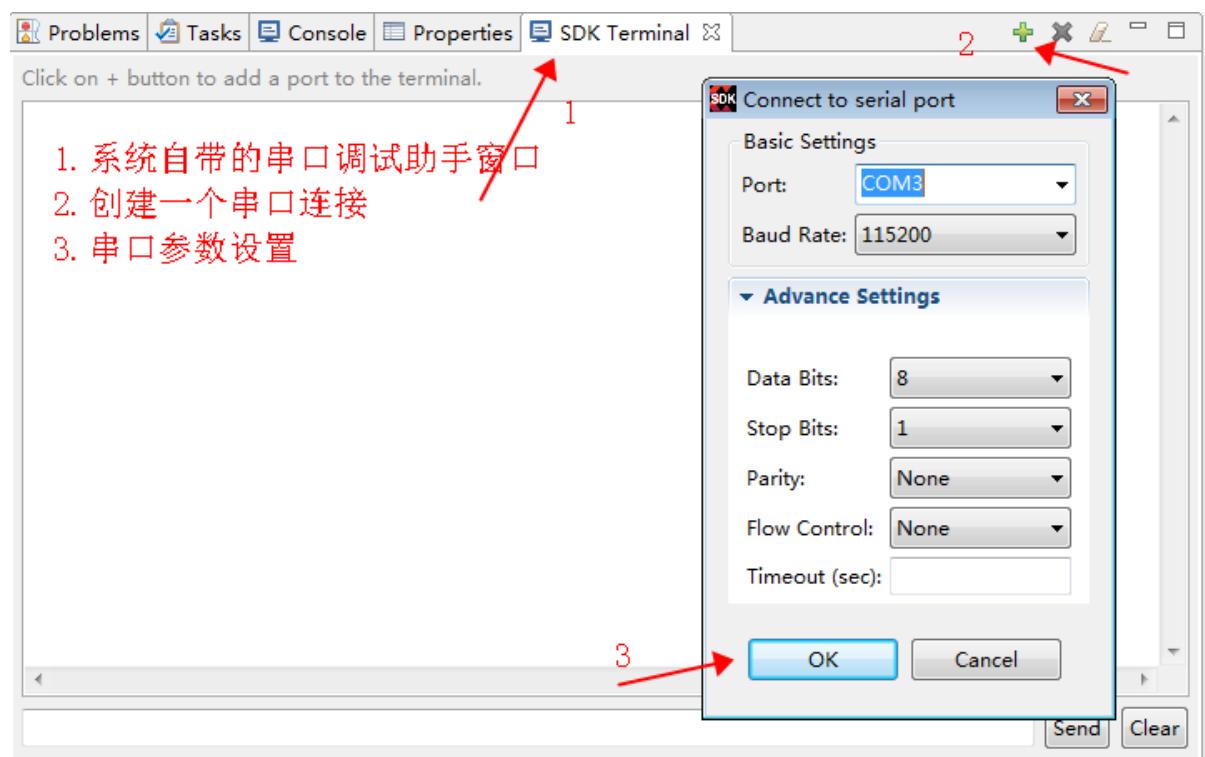


Step11:进入 SDK 调试界面

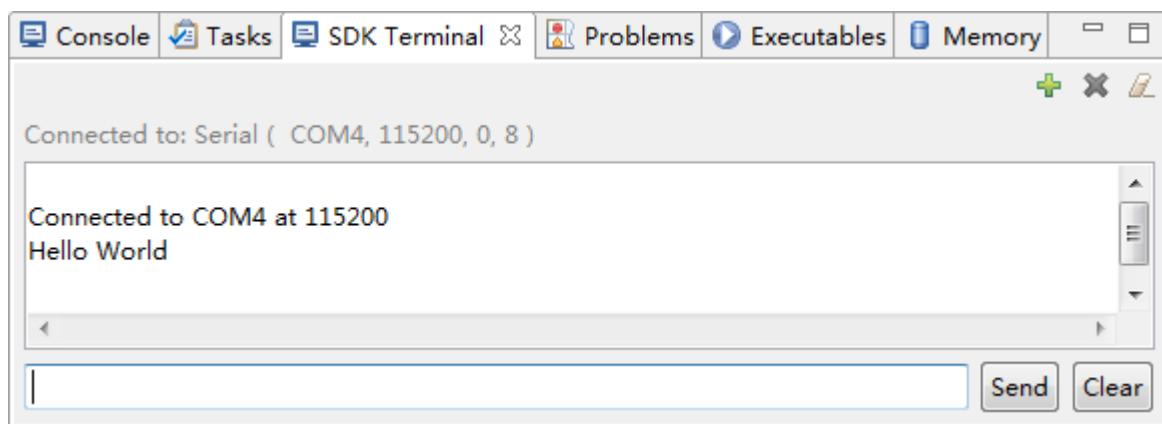
1、启动 2、暂停 3、停止 4、代码 5、信息控制台 6、调试变量



Step12:启用系统自带的串口调试助手，进行相关的设置。

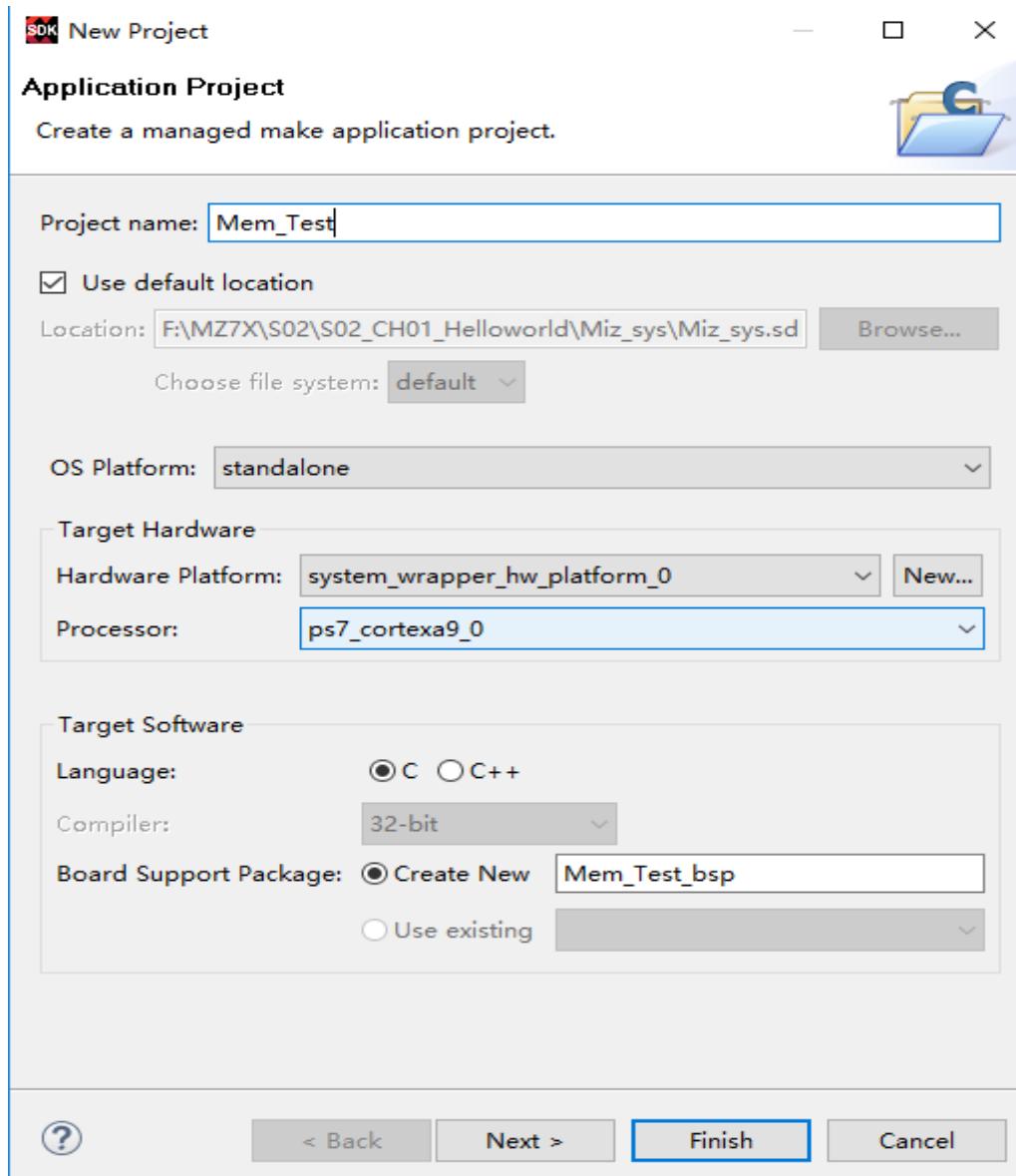


Step13:单击运行输出结果

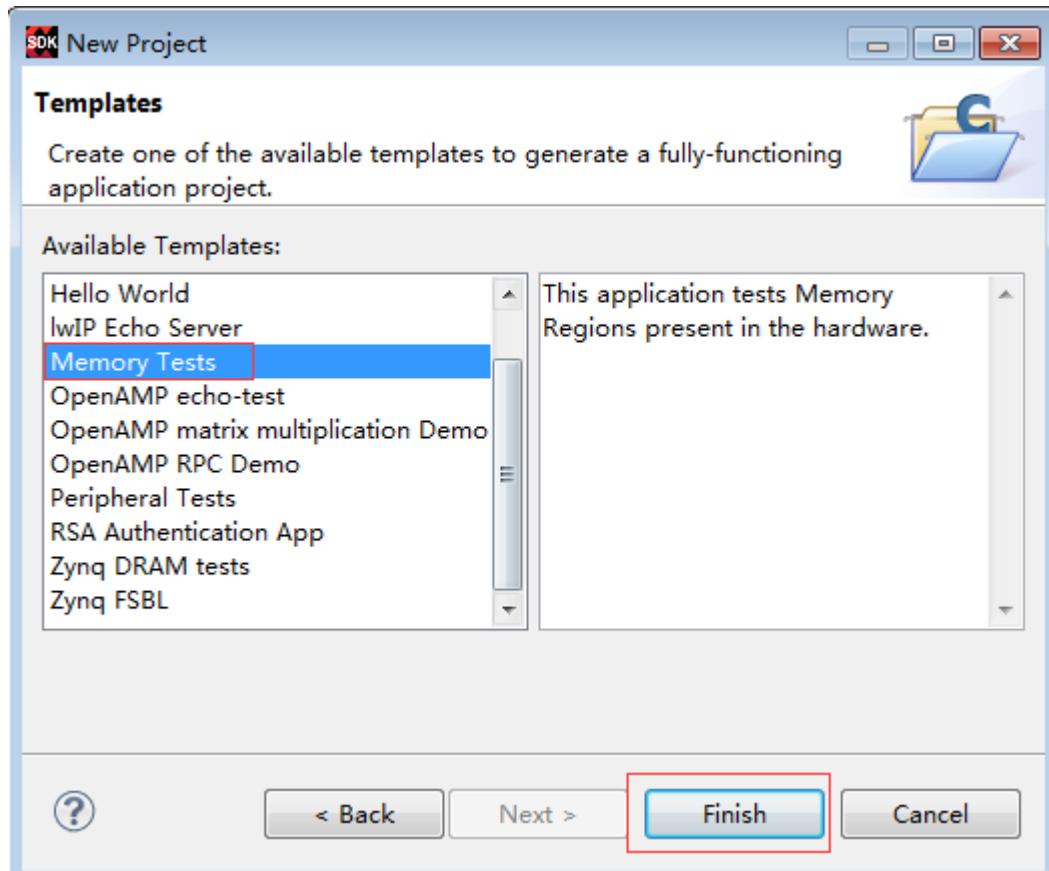


## 1.6 MemTest 内存测试程序

Step1:新建一个名为 MemTest 的工程



Step2:仍然采用自带的测试函数测试

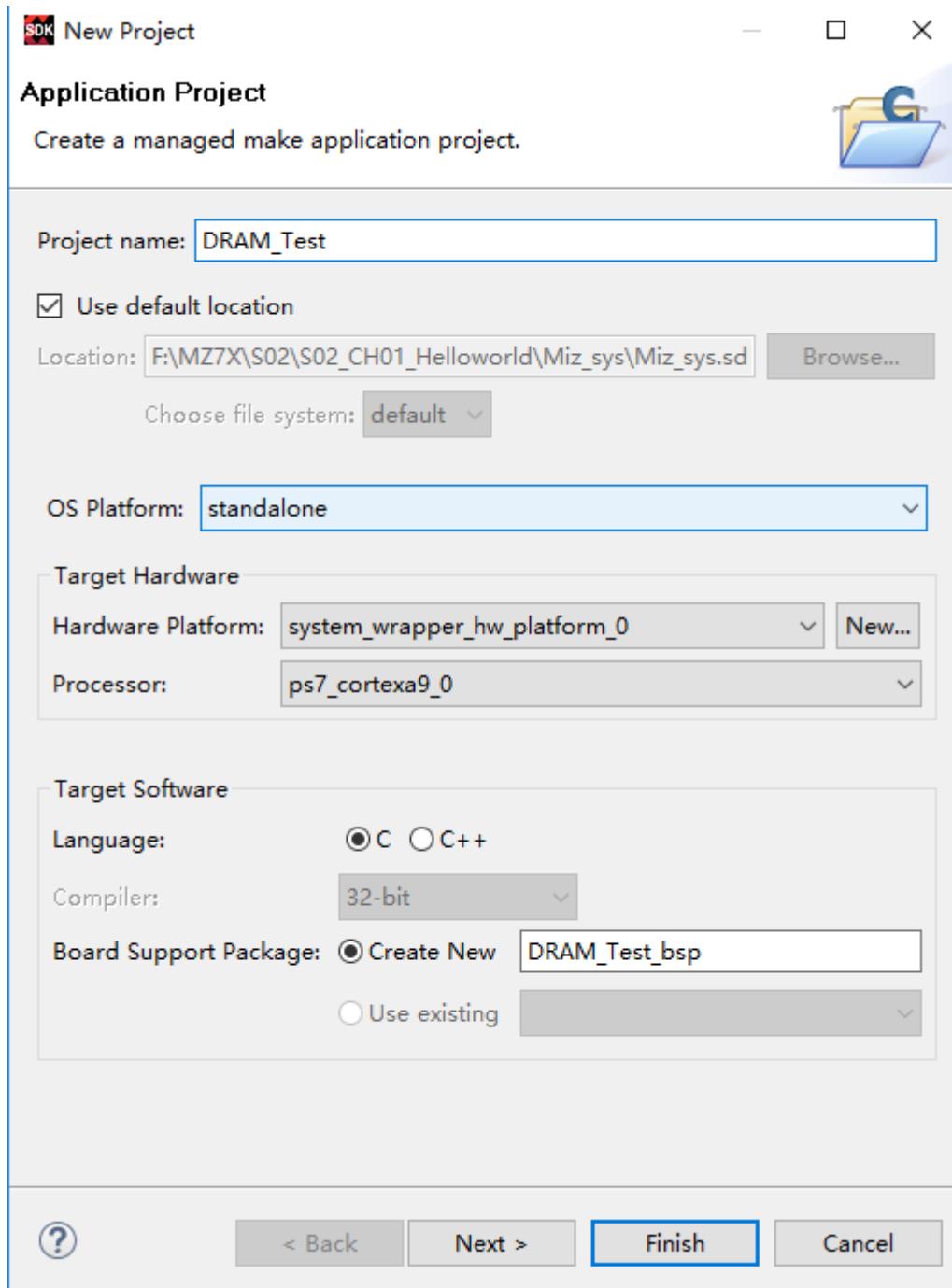


Step3: 测试结果

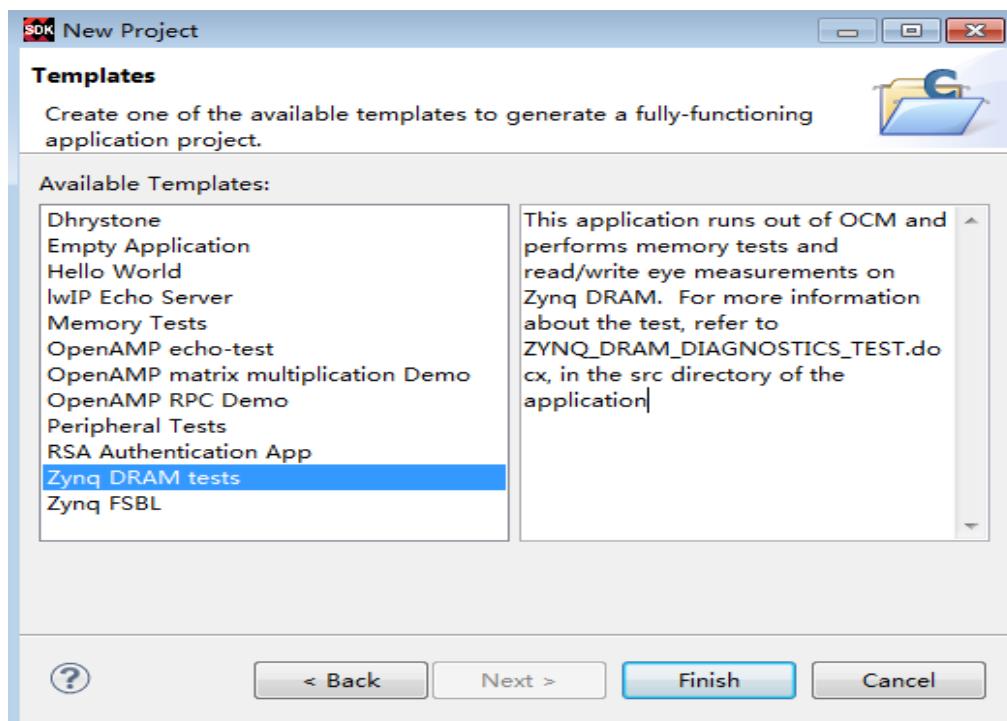
```
--Starting Memory Test Application--  
NOTE: This application runs with D-Cache disabled. As a result, cacheline requests will be aligned to memory boundaries.  
Testing memory region: ps7_ddr_0  
    Memory Controller: ps7_ddr  
        Base Address: 0x0001000000  
        Size: 0x1fff000000 bytes  
        32-bit test: PASSED!  
        16-bit test: PASSED!  
        8-bit test: PASSED!  
Testing memory region: ps7_ram_1  
    Memory Controller: ps7_ram  
        Base Address: 0xfffff00000  
        Size: 0x00000fe00 bytes  
        32-bit test: PASSED!  
        16-bit test: PASSED!  
        8-bit test: PASSED!  
--Memory Test Application Complete--
```

## 1.7 DRAMTest 内存测试程序

Step1:新建一个名为 MemTest 的工程



Step3:新建一个名为 MemTest 的工程



#### Step4: 测试结果

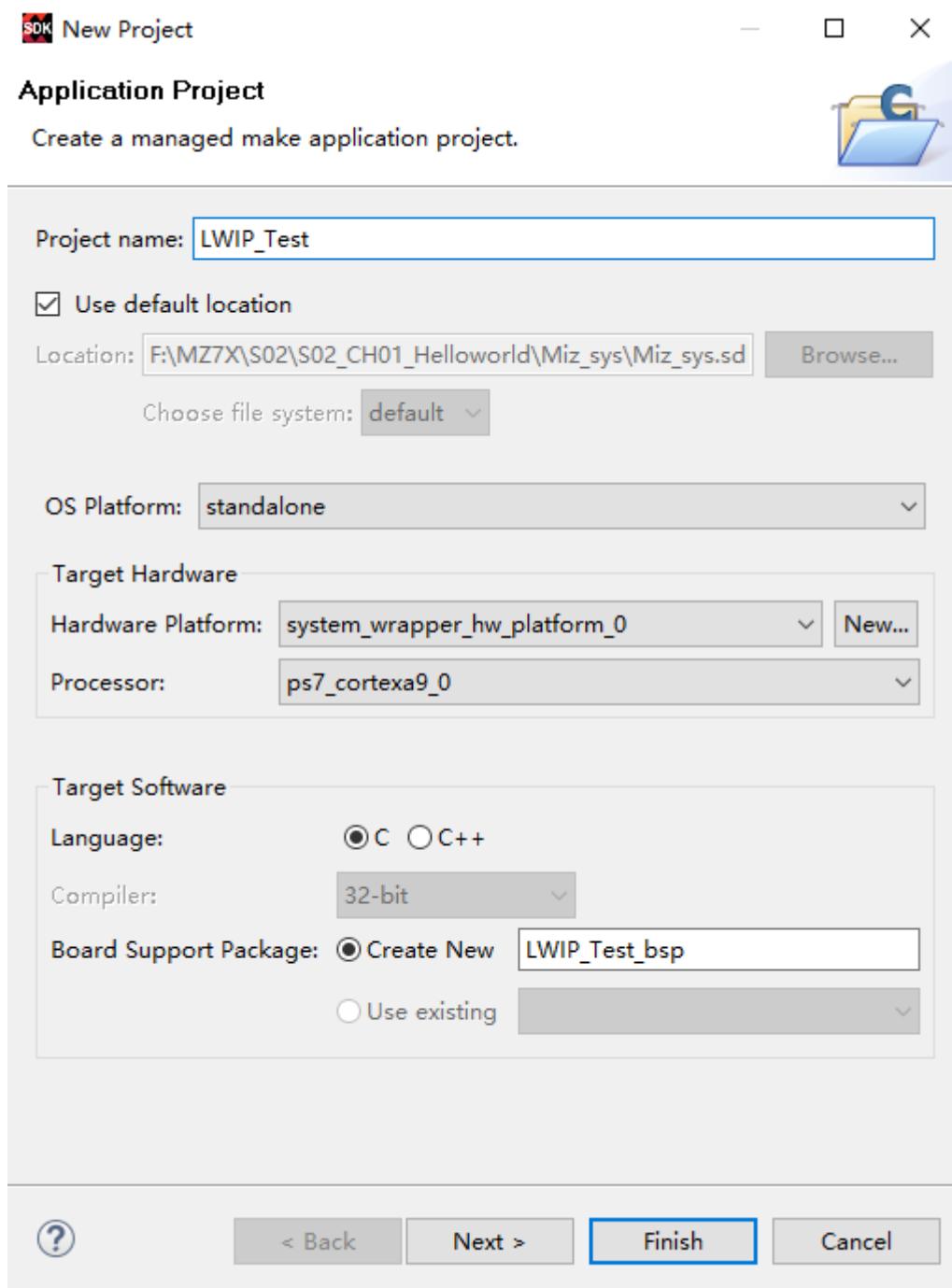
根据提示可以在控制台中输入相关序号按回车进行（r,i 测试会有一部分错误，还以和程序空间有关系）

```

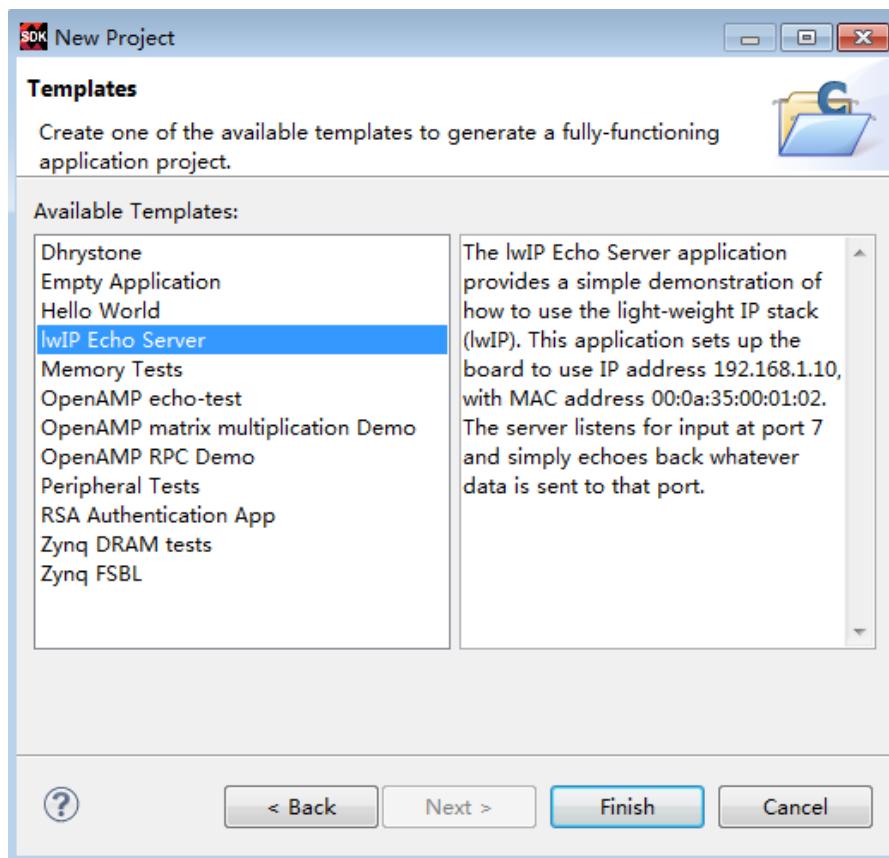
----- ZYNQ DRAM DIAGNOSTICS TEST -----
Select one of the options below:
## Memory Test ##
Bus Width = 32, XADC Temperature = 47.0328
's' - Test 1MB length from address 0x100000
'1' - Test 32MB length from address 0x100000
'2' - Test 64MB length from address 0x100000
'3' - Test 128MB length from address 0x100000
'4' - Test 255MB length from address 0x100000
'5' - Test 511MB length from address 0x100000
'6' - Test 1023MB length from address 0x100000
## Read Data Eye Measurement Test
'r' - Measure Read Data Eye
## Write Data Eye Measurement Test
'i' - Measure Write Data Eye
Other options for Write Eye Data Test:
  'f' - Fast Mode: Toggles Fast mode - ON/OFF
  'c' - Centre Mode: Toggles Centre mode - ON/OFF
  'e' - Vary the size of memory test for Read/Write Eye Measurement tests
## Data Cache Enable / Disable Option:
  'z' - D-Cache Enable / Disable
## Other options
  'v' - Verbose Mode ON/OFF
5| 
```

## 1.8 LWIP 协议对千兆网口测试

Step1: 新建一个名为 LWIP\_Test 的工程



Step2:选择 LWIP Echo Server 之后单击 Finish



Step3:运行之后的串口打印信息

```
-----lwIP TCP echo server -----
TCP packets sent to port 6001 will be echoed back
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
DHCP Timeout
Configuring default IP of 192.168.1.10
Board IP: 192.168.1.10
Netmask : 255.255.255.0
Gateway : 192.168.1.1
TCP echo server started @ port 7
```

Step4:用网络助手实现回传测试



## 1.9 使用快捷按钮调试



使用这两个图标，一个是 debug 一个是运行模式可以方便调试。

## 1.10 本章小结

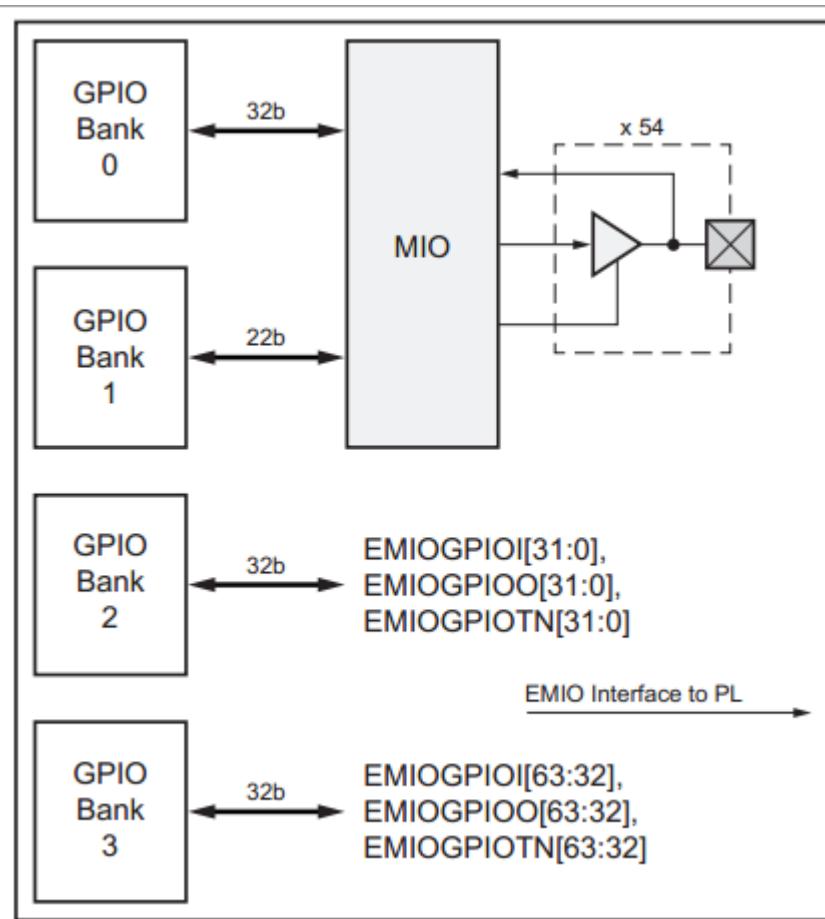
本章详细讲解了定制一个 SOC 最小系统，并且运行了自带的 HelloWorld 工程、MemTest 内存测试工程、DRAMTest 内存测试工程、LWIP 网络协议工程对千兆网口测试。本章让初学者可以搭建一个最小的 SOC 系统，并且教会读者利用软件自动的工程对 SOC 的基本外设进行测试。希望大家多多操作，熟练掌握如何创建 VIVADO 工程，懂得如何根据自己的硬件平台配置 ZYNQ CPU IP，下章我们将不在对这些进行详细的讲解。

## CH02\_MIO 实验

### 2.1 GPIO 简介

Zynq7000 系列芯片有 54 个 MIO(multiuse I/O)，它们分配在 GPIO 的 Bank0 和 Bank1 隶属于 PS 部分，这些 IO 与 PS 直接相连。不需要添加引脚约束，MIO 信号对 PL 部分是透明的，不可见。所以对 MIO 的操作可以看作是纯 PS 的操作。

GPIO 的控制和状态寄存器基址为：0xE000\_A000，我们 SDK 下软件操作底层都是对于内存地址空间的操作。



Bank0:MIO[31:0]

Bank1:MIO[52:53]

Bank2:EMIO[31:0]

Bank3:EMIO[63:32]

## 2.1.1 GPIO 的控制寄存器地址空间

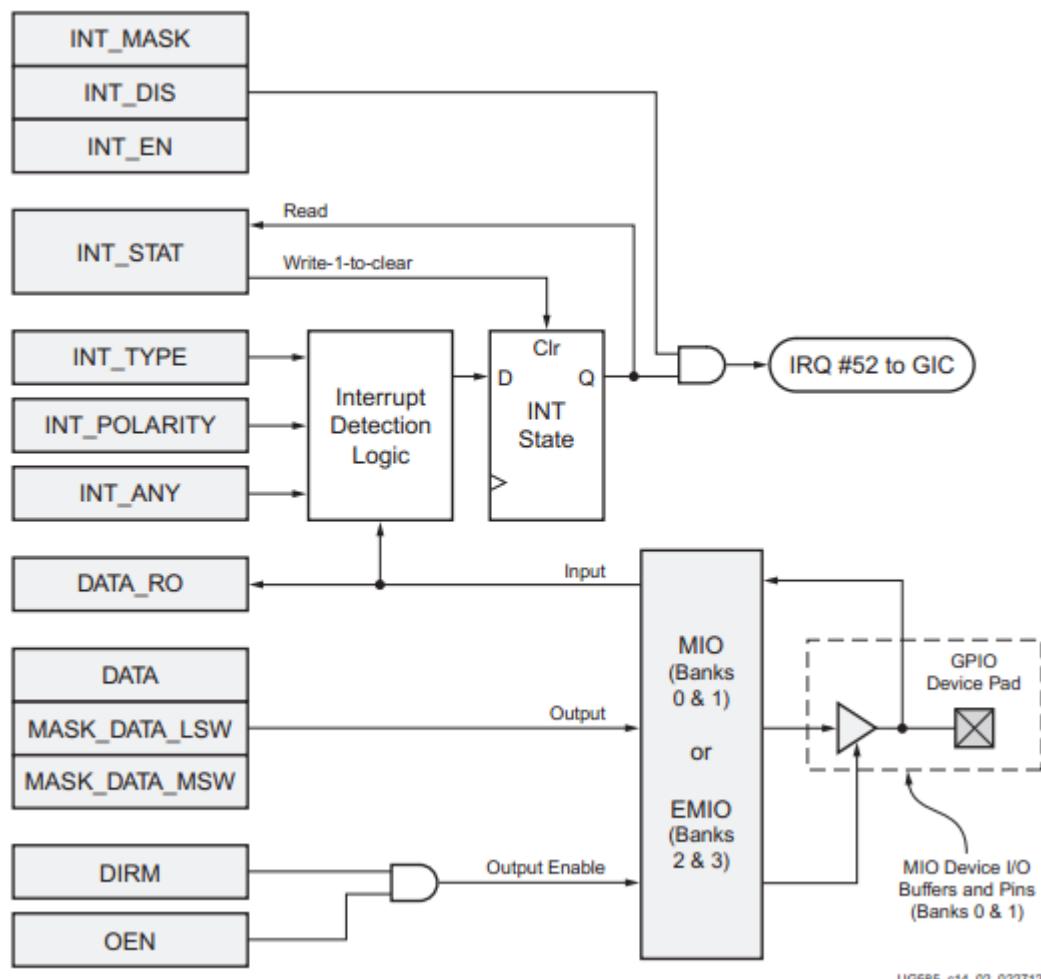
我们在 SDK 下操作的时候底层都是对这些寄存器的操作，具体的相关参数请参考技术手册 ug585-Zynq-7000-TRM.pdf

Register Name	Address	Width	Type	Reset Value	Description
<a href="#">MASK_DATA_0_LSW</a>	0x00000000	32	mixed	x	Maskable Output Data (GPIO Bank0, MIO, Lower 16bits)
<a href="#">MASK_DATA_0_MSB</a>	0x00000004	32	mixed	x	Maskable Output Data (GPIO Bank0, MIO, Upper 16bits)
<a href="#">MASK_DATA_1_LSW</a>	0x00000008	32	mixed	x	Maskable Output Data (GPIO Bank1, MIO, Lower 16bits)
<a href="#">MASK_DATA_1_MSB</a>	0x0000000C	22	mixed	x	Maskable Output Data (GPIO Bank1, MIO, Upper 6bits)
<a href="#">MASK_DATA_2_LSW</a>	0x00000010	32	mixed	0x00000000	Maskable Output Data (GPIO Bank2, EMIO, Lower 16bits)
<a href="#">MASK_DATA_2_MSB</a>	0x00000014	32	mixed	0x00000000	Maskable Output Data (GPIO Bank2, EMIO, Upper 16bits)
<a href="#">MASK_DATA_3_LSW</a>	0x00000018	32	mixed	0x00000000	Maskable Output Data (GPIO Bank3, EMIO, Lower 16bits)
<a href="#">MASK_DATA_3_MSB</a>	0x0000001C	32	mixed	0x00000000	Maskable Output Data (GPIO Bank3, EMIO, Upper 16bits)
<a href="#">DATA_0</a>	0x00000040	32	rw	x	Output Data (GPIO Bank0, MIO)
<a href="#">DATA_1</a>	0x00000044	22	rw	x	Output Data (GPIO Bank1, MIO)
<a href="#">DATA_2</a>	0x00000048	32	rw	0x00000000	Output Data (GPIO Bank2, EMIO)
<a href="#">DATA_3</a>	0x0000004C	32	rw	0x00000000	Output Data (GPIO Bank3, EMIO)
<a href="#">DATA_0_RO</a>	0x00000060	32	ro	x	Input Data (GPIO Bank0, MIO)
<a href="#">DATA_1_RO</a>	0x00000064	22	ro	x	Input Data (GPIO Bank1, MIO)
<a href="#">DATA_2_RO</a>	0x00000068	32	ro	0x00000000	Input Data (GPIO Bank2, EMIO)
<a href="#">DATA_3_RO</a>	0x0000006C	32	ro	0x00000000	Input Data (GPIO Bank3, EMIO)

Register Name	Address	Width	Type	Reset Value	Description
<a href="#">DIRM_0</a>	0x00000204	32	rw	0x00000000	Direction mode (GPIO Bank0, MIO)
<a href="#">OEN_0</a>	0x00000208	32	rw	0x00000000	Output enable (GPIO Bank0, MIO)
<a href="#">INT_MASK_0</a>	0x0000020C	32	ro	0x00000000	Interrupt Mask Status (GPIO Bank0, MIO)
<a href="#">INT_EN_0</a>	0x00000210	32	wo	0x00000000	Interrupt Enable/Unmask (GPIO Bank0, MIO)
<a href="#">INT_DIS_0</a>	0x00000214	32	wo	0x00000000	Interrupt Disable/Mask (GPIO Bank0, MIO)
<a href="#">INT_STAT_0</a>	0x00000218	32	wtc	0x00000000	Interrupt Status (GPIO Bank0, MIO)
<a href="#">INT_TYPE_0</a>	0x0000021C	32	rw	0xFFFFFFFF	Interrupt Type (GPIO Bank0, MIO)
<a href="#">INT_POLARITY_0</a>	0x00000220	32	rw	0x00000000	Interrupt Polarity (GPIO Bank0, MIO)
<a href="#">INT_ANY_0</a>	0x00000224	32	rw	0x00000000	Interrupt Any Edge Sensitive (GPIO Bank0, MIO)
<a href="#">DIRM_1</a>	0x00000244	22	rw	0x00000000	Direction mode (GPIO Bank1, MIO)
<a href="#">OEN_1</a>	0x00000248	22	rw	0x00000000	Output enable (GPIO Bank1, MIO)
<a href="#">INT_MASK_1</a>	0x0000024C	22	ro	0x00000000	Interrupt Mask Status (GPIO Bank1, MIO)
<a href="#">INT_EN_1</a>	0x00000250	22	wo	0x00000000	Interrupt Enable/Unmask (GPIO Bank1, MIO)
<a href="#">INT_DIS_1</a>	0x00000254	22	wo	0x00000000	Interrupt Disable/Mask (GPIO Bank1, MIO)
<a href="#">INT_STAT_1</a>	0x00000258	22	wtc	0x00000000	Interrupt Status (GPIO Bank1, MIO)
<a href="#">INT_TYPE_1</a>	0x0000025C	22	rw	0x003FFFFF	Interrupt Type (GPIO Bank1, MIO)
<a href="#">INT_POLARITY_1</a>	0x00000260	22	rw	0x00000000	Interrupt Polarity (GPIO Bank1, MIO)
<a href="#">INT_ANY_1</a>	0x00000264	22	rw	0x00000000	Interrupt Any Edge Sensitive (GPIO Bank1, MIO)
<a href="#">DIRM_2</a>	0x00000284	32	rw	0x00000000	Direction mode (GPIO Bank2, EMIO)
<a href="#">OEN_2</a>	0x00000288	32	rw	0x00000000	Output enable (GPIO Bank2, EMIO)

Register Name	Address	Width	Type	Reset Value	Description
<a href="#">INT_MASK_2</a>	0x0000028C	32	ro	0x00000000	Interrupt Mask Status (GPIO Bank2, EMIO)
<a href="#">INT_EN_2</a>	0x00000290	32	wo	0x00000000	Interrupt Enable/Unmask (GPIO Bank2, EMIO)
<a href="#">INT_DIS_2</a>	0x00000294	32	wo	0x00000000	Interrupt Disable/Mask (GPIO Bank2, EMIO)
<a href="#">INT_STAT_2</a>	0x00000298	32	wtc	0x00000000	Interrupt Status (GPIO Bank2, EMIO)
<a href="#">INT_TYPE_2</a>	0x0000029C	32	rw	0xFFFFFFFF	Interrupt Type (GPIO Bank2, EMIO)
<a href="#">INT_POLARITY_2</a>	0x000002A0	32	rw	0x00000000	Interrupt Polarity (GPIO Bank2, EMIO)
<a href="#">INT_ANY_2</a>	0x000002A4	32	rw	0x00000000	Interrupt Any Edge Sensitive (GPIO Bank2, EMIO)
<a href="#">DIRM_3</a>	0x000002C4	32	rw	0x00000000	Direction mode (GPIO Bank3, EMIO)
<a href="#">OEN_3</a>	0x000002C8	32	rw	0x00000000	Output enable (GPIO Bank3, EMIO)
<a href="#">INT_MASK_3</a>	0x000002CC	32	ro	0x00000000	Interrupt Mask Status (GPIO Bank3, EMIO)
<a href="#">INT_EN_3</a>	0x000002D0	32	wo	0x00000000	Interrupt Enable/Unmask (GPIO Bank3, EMIO)
<a href="#">INT_DIS_3</a>	0x000002D4	32	wo	0x00000000	Interrupt Disable/Mask (GPIO Bank3, EMIO)
<a href="#">INT_STAT_3</a>	0x000002D8	32	wtc	0x00000000	Interrupt Status (GPIO Bank3, EMIO)
<a href="#">INT_TYPE_3</a>	0x000002DC	32	rw	0xFFFFFFFF	Interrupt Type (GPIO Bank3, EMIO)
<a href="#">INT_POLARITY_3</a>	0x000002E0	32	rw	0x00000000	Interrupt Polarity (GPIO Bank3, EMIO)
<a href="#">INT_ANY_3</a>	0x000002E4	32	rw	0x00000000	Interrupt Any Edge Sensitive (GPIO Bank3, EMIO)

## 2.1.2 MIO 内部构造分析



UG685\_c14\_02\_022712

**DATA\_RO:** 此寄存器使能软件观察 PIN 脚，当 GPIO 被配置成输出的时候，这个寄存器的值会反应输出的 PIN 脚情况。

**DATA:**此寄存器控制输出到 GPIO 的值，读这个寄存器的值可以读到最后一次写入该寄存器的值。

**MASK\_DATA\_LSW:**位操作寄存器，写入 GPIO 低 16bit 其他没有改变的位置保存原先的状态

**MASK\_DATA\_MSW:**位操作寄存器，写入 GPIO 高 16bit 其他没有改变的位置保存原先的状态

**DIRM:**此寄存器控制输出的开关，当 DIRM[x]==0 时候，禁止输出

**OEN:** 输出使能，当 OEN[x]==0 的时候输出关闭，PIN 脚处于三态

因此，如果要读 IO 状态就得读 DATA\_RO 的值，如果是对某一位进行操作就是写 MASK\_DATA\_LSW/MASK\_DATA\_MSW

具体的相关参数请参考技术手册 ug585-Zynq-7000-TRM.pdf

### 2.1.3 EMIO 的特性

与 MIO 大部分类似但是一下几点需要注意下

- EMIO 在 PL 部分，输入与 OEN 寄存器无关，当 DIRM 设置为 0 的时候设置为输入可以读 DATA\_RO 寄存器获取数据。
- 输出不能设置成三态，当 DIRM 设置为 1 的时候为输出，写入 DATA 寄存器或者 MASK\_DATA\_LSW/MASK\_DATA\_MSW 寄存器
- EMIOTN[x]=DIRM[x] & OEN[x]，实现输出的控制。

具体的相关参数请参考技术手册 ug585-Zynq-7000-TRM.pdf

## 2.2 电路分析及实验预期

在米联系列的开发板上有一个 MIO 是与开发板上的一个 LD9 相连的，这个 MIO 就是 MI07。实验通过操作该 MIO 来实现 LD9 的闪烁。

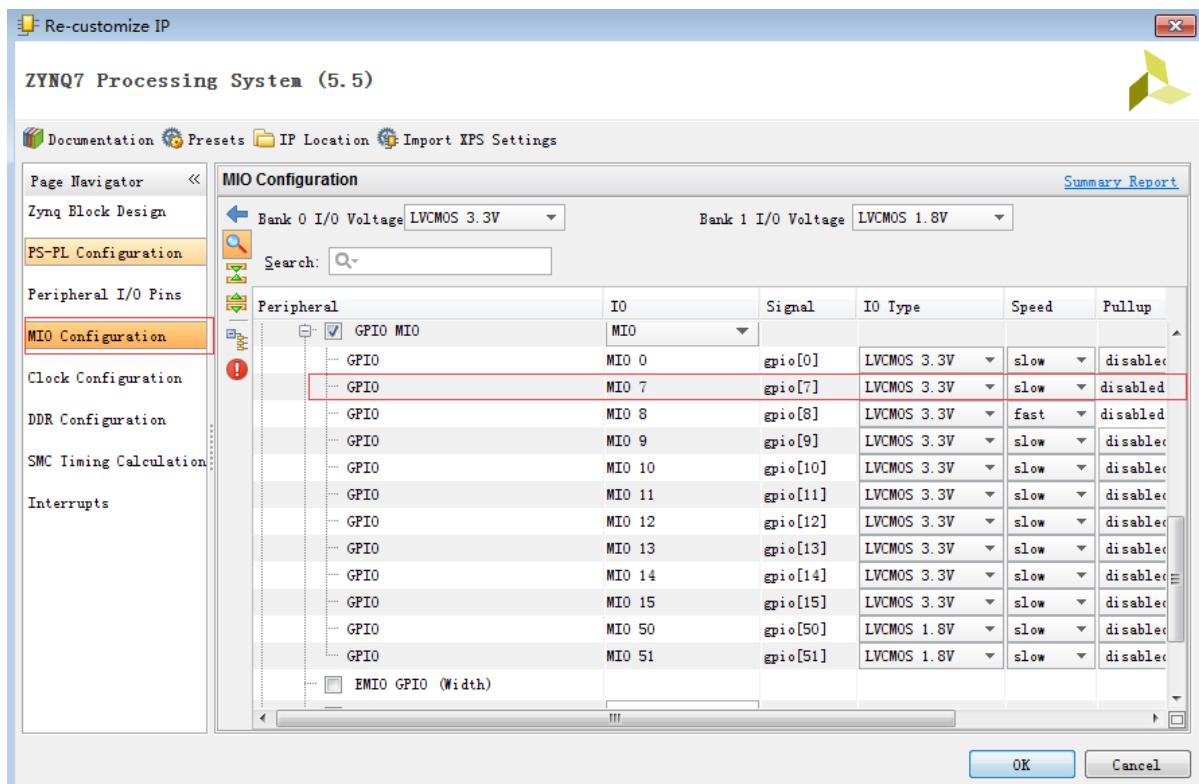
## 2.3 ZYNQ 核的添加及配置

Step1:新建一个名为 Miz\_sys 的工程，正确配置芯片型号，还未掌握的请参照上一章进行设置。

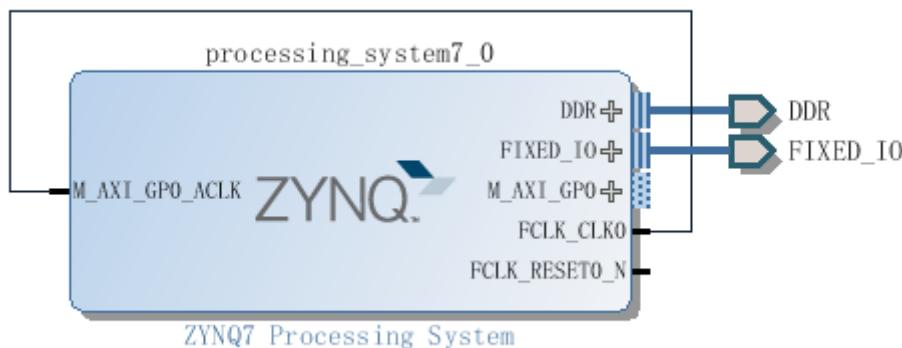
Step2：单击 Create Block Design，创建一个 BD 文件，并命名为 System。

Step3:加入一个 ZYNQ CPU IP，根据自己的产品型号，正确配置时钟频率与内存类型，尚未掌握的请重新温习上一章内容。

Step4：由于本章需要用到 MIO 接口，因此需要确保 MIO 选项被勾选（默认已勾选）。



Step5:单击 OK 后退出，系统整体电路如下。



Step6: 右击 system.bd, 单击 Generate Output Products。

Step7: 右击 system.bd 选择 Create HDL Wrapper 产生顶层的 HDL 文件。

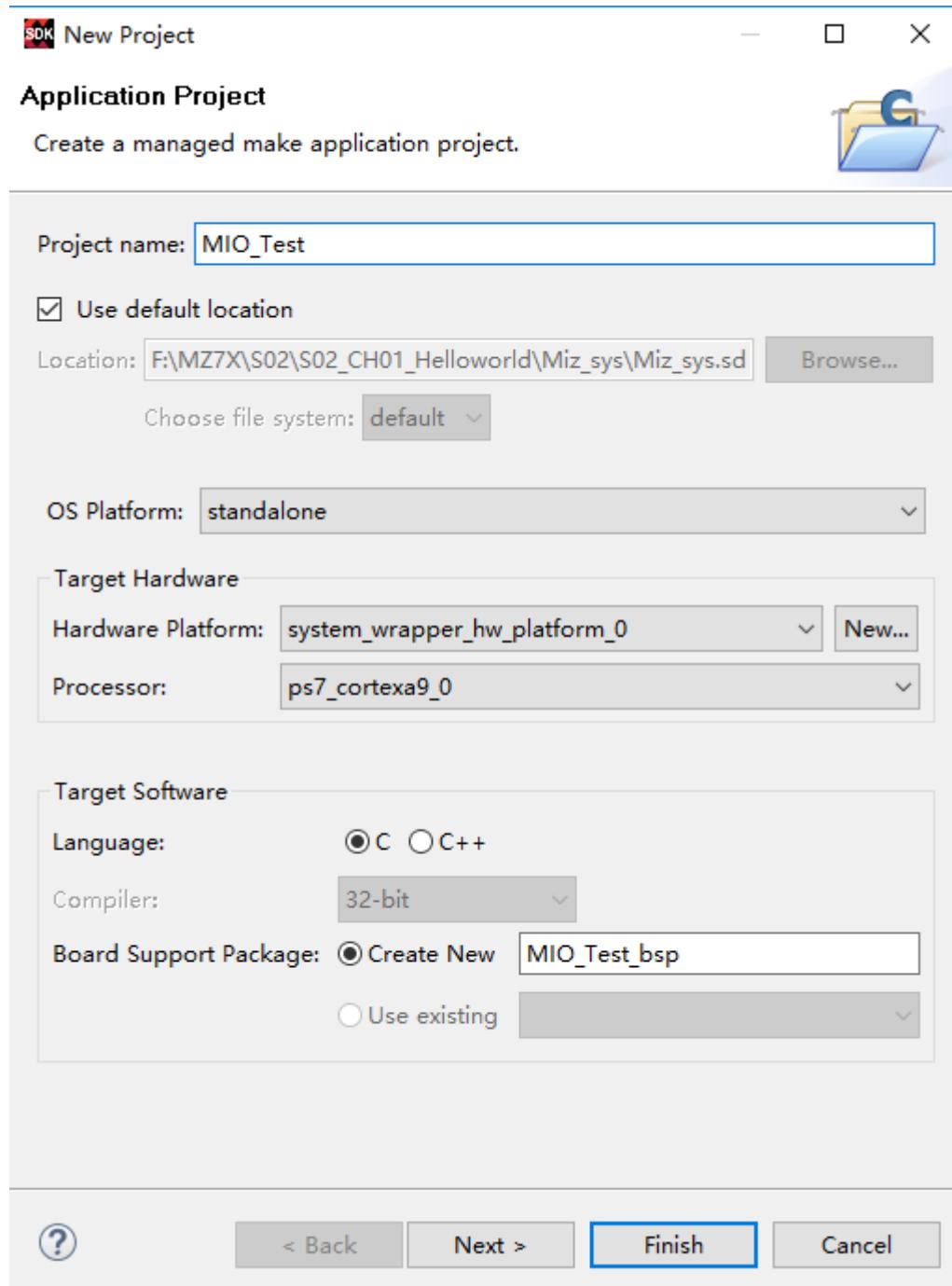
Step8: File->Export->Export Hardware。

Step9: 勾选 Include bitstream 直接单击 OK。

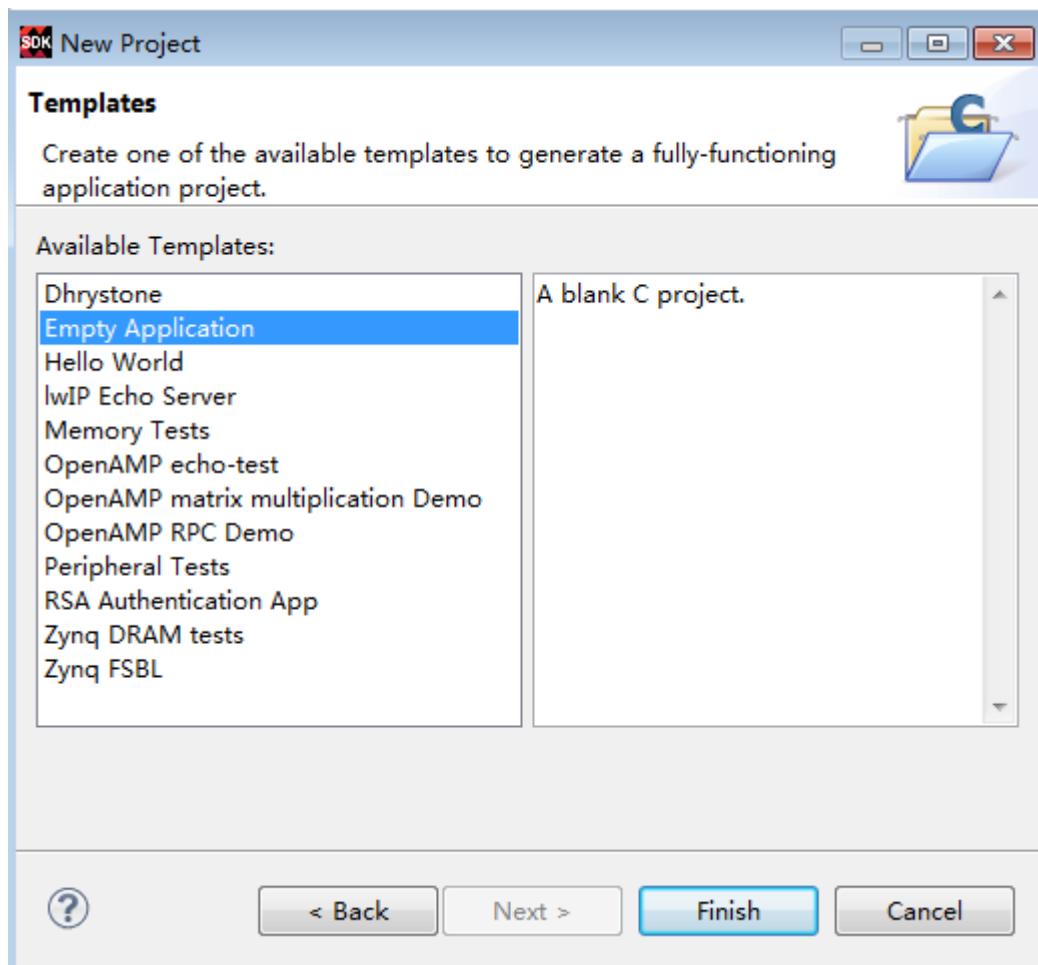
Step10: File->Launch SDK 加载到 SDK，单击 OK。

## 2.4 新建 LED\_Flash SDK 工程

Step1: 在 SDK 界面中，新建一个名为 MIO\_Test 的工程



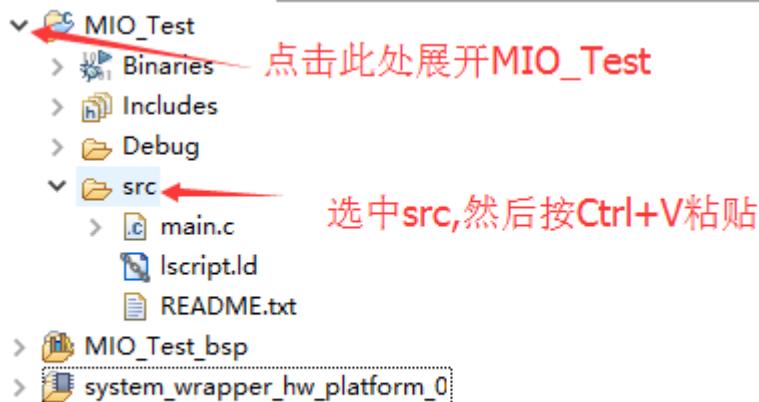
Step2:建立一个空的工程



Step3：在我们提供的源程序文件夹中找到 DOC 文件夹下的 C\_Driver 文件夹，将其中的设计文件复制一下。



Step4：点击 MIO\_Test 旁边的箭头使其展开，然后选中 src, 按下 Ctrl+V 快捷键完成粘贴

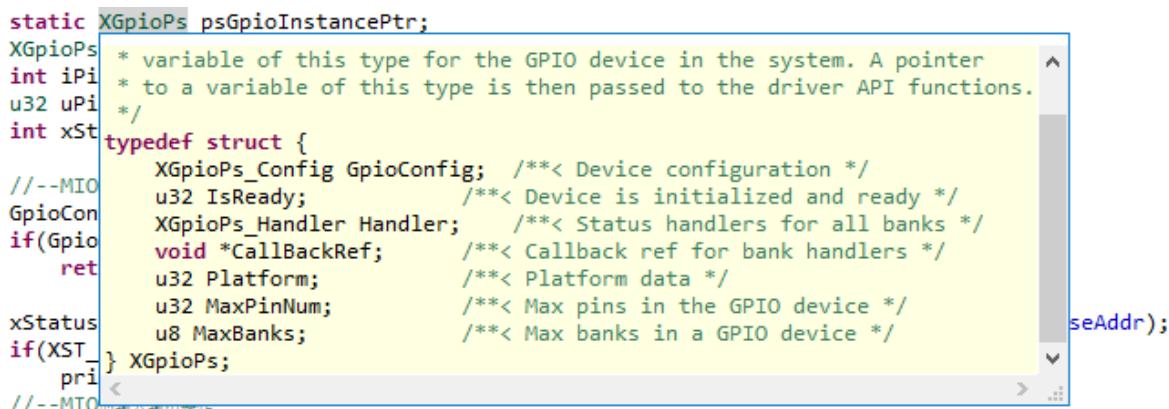


## 2.5 程序分析

接下来我们对整个程序做一个分析。

`static XGpioPs psGpioInstancePtr;` 这是一个指针实例，指向的我们添加进来的 GPIO 端口。

绿色标识的一个结构体（SDK 中结构体都用绿色标识）`XGpiops`，我们将鼠标停留在这个结构体上面，就可以看到它里面所包含的内容。



从这个图上可以看到，这个结构体上包含了 GPIO 的一些参数，分别是：设备的配置、设备是否初始化并准备好、所有状态的处理程序、块处理程序的回调、设备数据、GPIO 的最大 pin 数量和 GPIO 的最大的 bank 数量。

`XGpioPs_Config* GpioConfigPtr;` 也是一个指针实例，按照刚才介绍的方法我们来查看下它的释义。

```
/*
 * This typedef contains configuration information for a device.
 */
typedef struct {
    u16 DeviceId;          /* Unique ID of device */
    u32 BaseAddr;          /* Register base address */
} XGpioPs_Config;
```

从中可以看出，此结构体存放的是 GPIO 的设备地址和基地址。

iPinNumber 这个参数，是告知程序，操作的 MIO 是哪一个，因为我们要操作的是 MI07，所以这里所以这里的 iPinNumber 等于 7，在后一章的 EMI0 中也有这个参数，具体怎么算请参看下一节内容，这里就做个铺垫吧。

```
GpioConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
if(GpioConfigPtr == NULL)
    return XST_FAILURE;
```

这段程序是一段

查找 GPIO 配置程序。XGpiops\_Lookupconfig()这个函数是一个 xilinx 官方提供的 GPIO 的查找配置的函数，程序的参数为要查找的 GPIO 的基地址。基地址可从 xparameters.h 中查看，单击 BSP 支持包（此处为 MI0\_Test\_bsp）的小三角形，选择 Ps7\_Cortexa9\_0 文件夹下的 include 文件夹，在其中找到 xparameters.h，双击打开它。若未找到，则

在主界面下的 System.mss 界面点击 **Re-generate BSP Sources**，重新生成 BSP 支持包，此时只要耐心等待即可。如下图所示。

```
/* Definitions for peripheral PS7_RAM_0 */
#define XPAR_PS7_RAM_0_S_AXI_BASEADDR 0x00000000
#define XPAR_PS7_RAM_0_S_AXI_HIGHADDR 0x0003FFFF

/* Definitions for peripheral PS7_RAM_1 */
#define XPAR_PS7_RAM_1_S_AXI_BASEADDR 0xFFFFC0000
#define XPAR_PS7_RAM_1_S_AXI_HIGHADDR 0xFFFFFFF

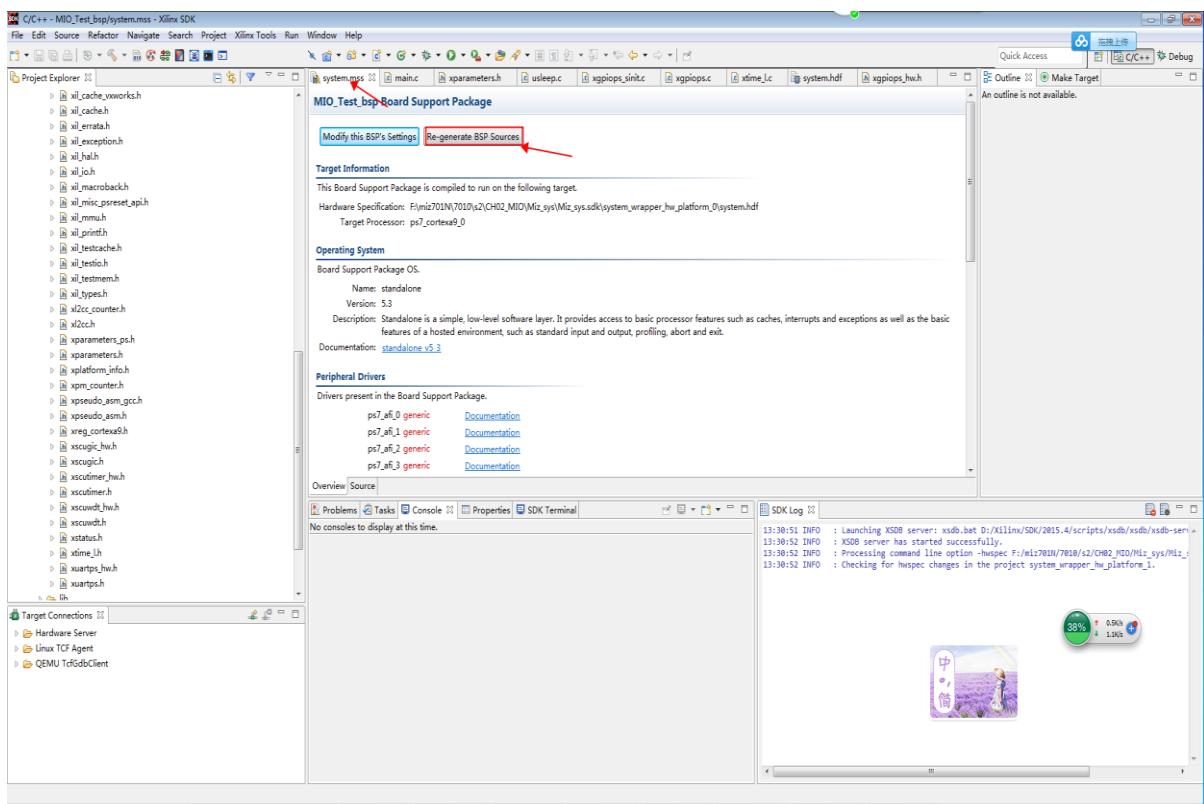
/* Definitions for peripheral PS7_SCUC_0 */
#define XPAR_PS7_SCUC_0_S_AXI_BASEADDR 0xF8F00000
#define XPAR_PS7_SCUC_0_S_AXI_HIGHADDR 0xF8F000FC

/* Definitions for peripheral PS7_SLCR_0 */
#define XPAR_PS7_SLCR_0_S_AXI_BASEADDR 0x80000000
#define XPAR_PS7_SLCR_0_S_AXI_HIGHADDR 0x80000FFF

/*****************************************/
/* Definitions for driver GPIOPS */
#define XPAR_XGPIOPS_NUM_INSTANCES 1

/* Definitions for peripheral PS7_GPIO_0 */
#define XPAR_PS7_GPIO_0_DEVICE_ID 0
#define XPAR_PS7_GPIO_0_BASEADDR 0xE000A000
#define XPAR_PS7_GPIO_0_HIGHADDR 0xE000AFFF

/*****************************************/
```



此处我们用到的就是 `XPAR_PS7_GPIO_0_DEVICE_ID`。这段话的整体意思就是查找 GPIO 的配置，然后判断其是否为空，若为空则返回查找失败。

```
xStatus = XGpioPs_CfgInitialize(&psGpioInstancePtr, GpioConfigPtr, GpioConfigPtr->BaseAddr);
if(XST_SUCCESS != xStatus)
    print(" PS GPIO INIT FAILED \n\r");
```

上图这段程序也是跟刚才大同小异，完成的是 gpio 配置的初始化工作，如果初始化不成功的话，将通过串口打印出一串初始化失败的通知信息，在此就不再去对其详细的分析。

本章中具体来看看 `XGpioPs_SetDirectionPin(&psGpioInstancePtr, iPinNumber, uPinDirection); //配置MIO输出方向`

这个函数，因为此函数中涉及到了一些 ZYNQ 中 GPIO 的硬件结构，将鼠标停留在这个函数上，按 F3 查看其函数定义。

```

/****************************************************************************
**
* Set the Direction of the specified pin.
*
* @param InstancePtr is a pointer to the XGpioPs instance.
* @param Pin is the pin number to which the Data is to be written.
*           Valid values are 0-117 in Zynq and 0-173 in Zynq Ultrascale+ MP.
* @param Direction is the direction to be set for the specified pin.
*           Valid values are 0 for Input Direction, 1 for Output Direction.
*
* @return None.
*/
void XGpioPs_SetDirectionPin(XGpioPs *InstancePtr, u32 Pin, u32 Direction)
{
    u8 Bank;
    u8 PinNumber;
    u32 DirModeReg;

    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
    Xil_AssertVoid(Pin < InstancePtr->MaxPinNum);
    Xil_AssertVoid(Direction <= (u32)1);

    /* Get the Bank number and Pin number within the bank. */
    XGpioPs_GetBankPin((u8)Pin, &Bank, &PinNumber);

    DirModeReg = XGpioPs_ReadReg(InstancePtr->GpioConfig.BaseAddr,
                                ((u32)(Bank) * XGPIOPS_REG_MASK_OFFSET) +
                                XGPIOPS_DIRM_OFFSET);

    if (Direction != (u32)0) { /* Output Direction */
        DirModeReg |= ((u32)1 << (u32)PinNumber);
    } else { /* Input Direction */
        DirModeReg &= ~((u32)1 << (u32)PinNumber);
    }

    XGpioPs_WriteReg(InstancePtr->GpioConfig.BaseAddr,
                    ((u32)(Bank) * XGPIOPS_REG_MASK_OFFSET) +
                    XGPIOPS_DIRM_OFFSET, DirModeReg);
}

```

从上图方框圈出的地方我们可以看到此程序给出的功能说明，它完成的是指定 pin 脚的方向设置。这个程序中，首先它有一个读取 bank 号的子程序：

```

/* Get the Bank number and Pin number within the bank. */
XGpioPs_GetBankPin((u8)Pin, &Bank, &PinNumber);

```

我们将鼠标停留在这个函数之上，按 F3 查看下它是具体怎样来查找 bank 号的。

```

* @return None.
*
* @note None.
*
*****
void XGpioPs_GetBankPin(u8 PinNumber, u8 *BankNumber, u8 *PinNumberInBank)
{
    u32 XGpioPsPinTable[6] = {0};
    u32 Platform = XGetPlatform_Info();

    if (Platform == XPLAT_ZYNQ_ULTRA_MP) {
        /*
         * This structure defines the mapping of the pin numbers to the banks when
         * the driver APIs are used for working on the individual pins.
         */

        XGpioPsPinTable[0] = (u32)25; /* 0 - 25, Bank 0 */
        XGpioPsPinTable[1] = (u32)51; /* 26 - 51, Bank 1 */
        XGpioPsPinTable[2] = (u32)77; /* 52 - 77, Bank 2 */
        XGpioPsPinTable[3] = (u32)109; /* 78 - 109, Bank 3 */
        XGpioPsPinTable[4] = (u32)141; /* 110 - 141, Bank 4 */
        XGpioPsPinTable[5] = (u32)173; /* 142 - 173 Bank 5 */

        *BankNumber = 0U;
        while (*BankNumber < 6U) {
            if (PinNumber <= XGpioPsPinTable[*BankNumber]) {
                break;
            }
            (*BankNumber)++;
        }
    } else {
        XGpioPsPinTable[0] = (u32)31; /* 0 - 31, Bank 0 */
        XGpioPsPinTable[1] = (u32)53; /* 32 - 53, Bank 1 */
        XGpioPsPinTable[2] = (u32)85; /* 54 - 85, Bank 2 */
        XGpioPsPinTable[3] = (u32)117; /* 86 - 117 Bank 3 */

        *BankNumber = 0U;
        while (*BankNumber < 4U) {
            if (PinNumber <= XGpioPsPinTable[*BankNumber]) {
                break;
            }
            (*BankNumber)++;
        }
    }

    if (*BankNumber == (u8)0) {
        *PinNumberInBank = PinNumber;
    } else {
        *PinNumberInBank = (u8)((u32)PinNumber %
                               (XGpioPsPinTable[*BankNumber] - (u8)1) + (u32)1);
    }
}
/** @} */

```

上图中方框圈出的地方就是程序查找 bank 号的。一开始程序先判断了 ZYNQ 的类型，在本章第一节 GPIO 简介中我们知道，7010 和 7020 其实是有四个 bank 的，因此当程序执行后，其实程序是执行 else 部分的程序的。此时再来看看 else 部分的程序。程序首先给出了四个 bank 的 bank 号的最大值，然后初始化了 bank 号为 0，接下来的 while 语句限制了 bank 的最大数量为 4。接下来用 pin 的序号从 bank0 到 bank4 逐个比对，若是此时 pin 的序号小于或等于当前 bank 的最大值，则可以判断出 pin 是属于这个 bank 的，跳出 while 语句，否则 bank 号进行自加操作直到得出 bank 号。接下来又是一个 if 语句，判断 bank 号是否为 bank0，若是则将 pinnumber 直接赋值，否则经过计算一段公式得出 pinnumber。

接下来回到 XGpioPs\_SetDirectionPin 函数分析其他的子程序，在获取了 bank 号之后，是一

这

```

DirModeReg = XGpioPs_ReadReg(InstancePtr->GpioConfig.BaseAddr,
    ((u32)(Bank) * XGPIOPS_REG_MASK_OFFSET) +
    XGPIOPS_DIRM_OFFSET);

```

个读取寄存器的程序

里重点观察第二个参数，这是一个任务寄存器偏移+DIRM\_OFFSET 的参数，此时我们可打开 xilinx 的编程手册 ug585-zynq-7000-TRM(接下来的内容中我们将将其简称为 ug585)，来具体看看这个是个什么东西。

复制 DIRM，在 ug585 中查找到这么一段话：

- **DIRM:** Direction Mode. This controls whether the I/O pin is acting as an input or an output. Since the input logic is always enabled, this effectively enables/disables the output driver. When DIRM[x]==0, the output driver is disabled.

此时得知这其实就是一个方向寄存器，当它等于 0 的时候输出被禁止，只有输入被运行，也就是此时是作为输入用的，等于 1 时做输出用。这在 GPIO 的通道示意图中也能发现有这个部分构成。

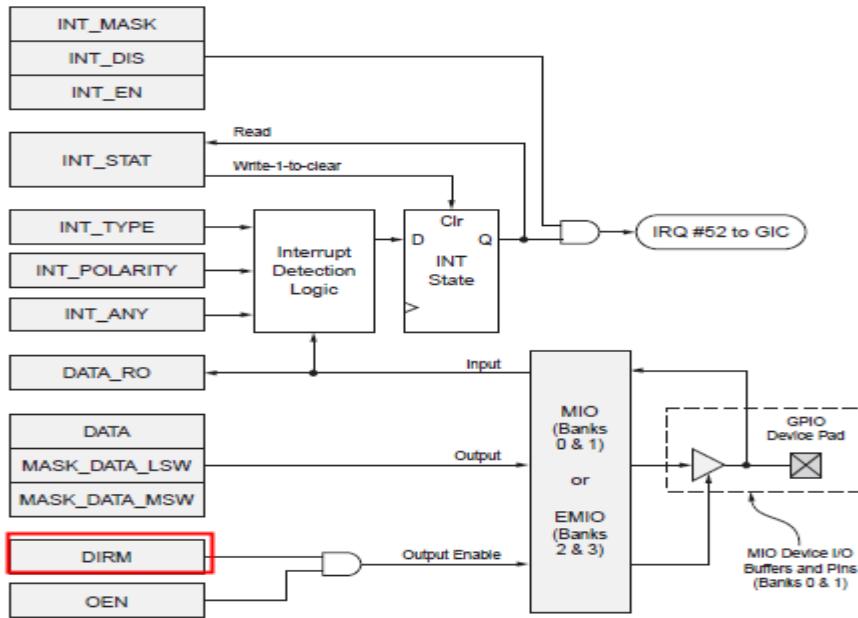


Figure 14-2: GPIO Channel

回到 `XGpioPs_SetDirectionPin` 的分析，再得到了 bank 号与要写哪个寄存器的地址后，接下来的 `if else` 语句就是对这对 pinbumer 这一位单独做一些操作，最后把方向寄存器的值写入到读出的那个寄存器当中。

回到 `main.c` 的分析当中，接下来的 `XGpioPs_SetOutputEnablePin` 函数，其原理与设置方向函数的原理是一样的，我们就不再深层次对其进行分析，它完成的功能在程序中也有注释。

最后，我们看到对单个位操作的函数 `XGpioPs_WritePin`，它与之前的程序结构也是大体一致的，它的三个参数分别为 `gpio` 的基地址、要操作的 MIO 号和写入的数据。按下 F3 查看一下它的定义。

```
void XGpioPs_WritePin(XGpioPs *InstancePtr, u32 Pin, u32 Data)
{
    u32 RegOffset;
    u32 Value;
    u8 Bank;
    u8 PinNumber;
    u32 DataVar = Data;

    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
    Xil_AssertVoid(Pin < InstancePtr->MaxPinNum);

    /* Get the Bank number and Pin number within the bank. */
    XGpioPs_GetBankPin((u8)Pin, &Bank, &PinNumber);

    if (PinNumber > 15U) {
        /* There are only 16 data bits in bit maskable register. */
        PinNumber -= (u8)16;
        RegOffset = XGPIOPS_DATA_MSB_OFFSET;
    } else {
        RegOffset = XGPIOPS_DATA_LSB_OFFSET;
    }

    /* Get the 32 bit value to be written to the Mask/Data register where
     * the upper 16 bits is the mask and lower 16 bits is the data.
     */
    DataVar &= (u32)0x01;
    Value = ~((u32)1 << (PinNumber + 16U)) & ((DataVar << PinNumber) | 0xFFFF0000U); // MIO7=1=FFFF7FFFF&FFFF8000; // MIO7=0=FFF7FFFF&FFF8000;
    XGpioPs_WriteReg(InstancePtr->GpioConfig.BaseAddr,
                    ((u32)(Bank) * XGPIOPS_DATA_MASK_OFFSET) +
                    RegOffset, Value);
}
```

上图中，我们直接看到方框圈起来的部分，此处我们观测到有两个陌生的偏移，此时我们可在 ug585 中查看一下它们具体是什么意思。

- **MASK\_DATA\_LSW:** This register enables more selective changes to the desired output value. Any combination of up to 16 bits can be written. Those bits that are not written are unchanged and hold their previous value. Reading from this register returns the previous value written to either DATA or MASK\_DATA\_[LSW,MSW]; it does not return the current value on the device pin. This register avoids the need for a read-modify-write sequence for unchanged bits.
- **MASK\_DATA\_MSB:** This register is the same as MASK\_DATA\_LSW, except it controls the upper16 bits of the bank.

Register Name	Address	Width	Type	Reset Value	Description
<a href="#">MASK DATA_0_LSW</a>	0x00000000	32	mixed	x	Maskable Output Data (GPIO Bank0, MIO, Lower 16bits)
<a href="#">MASK DATA_0_MSB</a>	0x00000004	32	mixed	x	Maskable Output Data (GPIO Bank0, MIO, Upper 16bits)
<a href="#">MASK DATA_1_LSW</a>	0x00000008	32	mixed	x	Maskable Output Data (GPIO Bank1, MIO, Lower 16bits)
<a href="#">MASK DATA_1_MSB</a>	0x0000000C	22	mixed	x	Maskable Output Data (GPIO Bank1, MIO, Upper 6bits)
<a href="#">MASK DATA_2_LSW</a>	0x00000010	32	mixed	0x00000000	Maskable Output Data (GPIO Bank2, EMIO, Lower 16bits)
<a href="#">MASK DATA_2_MSB</a>	0x00000014	32	mixed	0x00000000	Maskable Output Data (GPIO Bank2, EMIO, Upper 16bits)
<a href="#">MASK DATA_3_LSW</a>	0x00000018	32	mixed	0x00000000	Maskable Output Data (GPIO Bank3, EMIO, Lower 16bits)
<a href="#">MASK DATA_3_MSB</a>	0x0000001C	32	mixed	0x00000000	Maskable Output Data (GPIO Bank3, EMIO, Upper 16bits)
<a href="#">DATA_0</a>	0x00000040	32	rw	x	Output Data (GPIO Bank0, MIO)
<a href="#">DATA_1</a>	0x00000044	22	rw	x	Output Data (GPIO Bank1, MIO)
<a href="#">DATA_2</a>	0x00000048	32	rw	0x00000000	Output Data (GPIO Bank2, EMIO)
<a href="#">DATA_3</a>	0x0000004C	32	rw	0x00000000	Output Data (GPIO Bank3, EMIO)

此时可以得知，这两个分别是要写入数据的高 16 位偏移量和低 16 位偏移量。此时即可得知这段程序是通过判断 pinNumber 的值来决定寄存器偏移量是用高 16 位偏移量还是低 16 位偏移量。

此时再看 XGpioPs\_SetOutputEnablePin 函数的接下来的这段程序：

```

/*
 * Get the 32 bit value to be written to the Mask/Data register where
 * the upper 16 bits is the mask and lower 16 bits is the data.
 */
DataVar &= (u32)0x01;
Value = ~((u32)1 << (PinNumber + 16U)) & ((DataVar << PinNumber) | 0xFFFF0000U); //MIO7=1=FFF7FFFF&FFF8000; //MIO7=0=FFF7FFFF&FFF0000;
XGpioPs_WriteReg(InstancePtr->GpioConfig.BaseAddr,
    ((u32)(Bank) * XGPIOPS_DATA_MASK_OFFSET) +
    RegOffset, Value);

```

这段程序完成的就是向指定 MIO 写入某个值的操作。我们分析一下这段程序，比如我们要向 MIO7 写入 1，程序一开始已经把要写入的值赋值给了 DataVar，在此段程序程

序一开始又将 DataVar 与 0x01 与操作，此操作后 DataVar 的值还是为 1。

接下来的 value 就是要写入寄存器的值，我们来看看它是怎么操作的。

`~((u32)1 << (PinNumber + 16U))` 这里的意思为把 PinNumber 加上 16（也就是把 pinNumber 移到高 16 位）赋值为 1，然后再取反，执行完后这一段的值为<sup>~</sup>(80000) h，也就是(FFFF7FFF) h。

再看后半段 `((DataVar << PinNumber) | 0xFFFF0000U)`，之前已经得到 DataVar 的值为 1，因此这里的意思为把 pinNumber 位赋值为 1，再与 FFFF0000 或操作，执行完这一段的值为 (80) h | (FFFF0000) h，也就是 (FFFF0080) h，整句执行完之后就是 (FFF7FFF) h & (FFFF0080) h = (FFF70080) h。也就是此时 Value 的值为 FFF70080。

XGpioPs\_WriteReg 这个函数就是往寄存器中写入数据，按下 F3 查看函数定义。如下图所示。

```
/*
 */
* This macro writes to the given register.
* @param BaseAddr is the base address of the device.
* @param RegOffset is the offset of the register to be written.
* @param Data is the 32-bit value to write to the register.
* @return None.
* @note None.
*/
#define XGpioPs_WriteReg(BaseAddr, RegOffset, Data) \
    xil_Out32((BaseAddr) + (u32)(RegOffset), (u32)(Data))
```

从图上可知，第一个参数为设备的基地址，第二个参数为偏移量，此处为 0，第三个参数为要写入寄存器的数据。

另外程序还可直接使用寄存器函数对 MIO 进行操作，其用法参照我们之前的分析，寄存器函数操作如下所示：

```
XGpioPs_WriteReg(0xE000A000, 0x00000000, 0xFF7FFFFF&0xFFFF0080);
usleep(500000); //延时
XGpioPs_WriteReg(0xE000A000, 0x00000000, 0xFF7FFFFF&0xFFFF0000);
usleep(500000); //延时
```

按照之前我们讲过的方法，大家可自行对库函数进行分析。

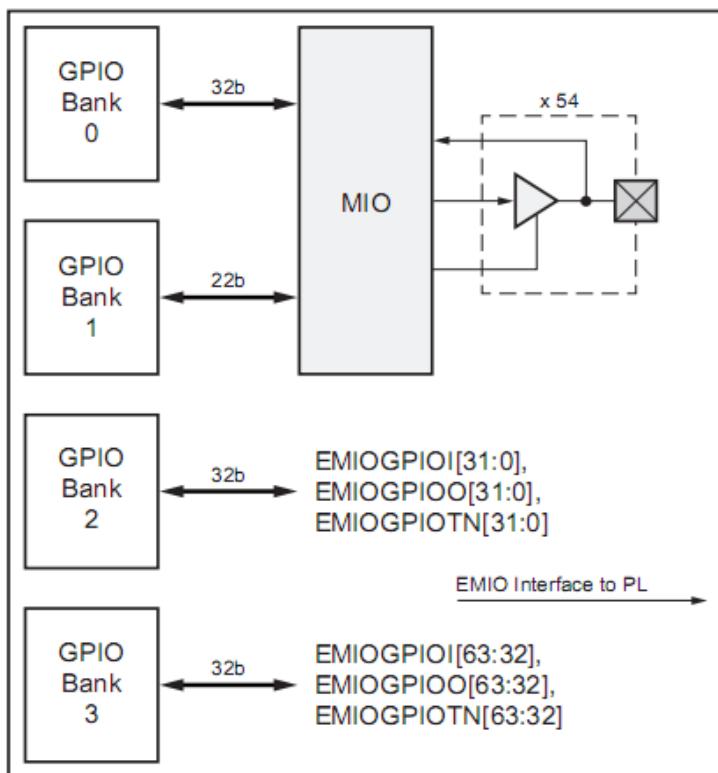
## 2.6 本章小结

本章讲解了 ZYNQ 芯片的 GPIO 的一些知识，然后通过使用 SDK 进行编程点亮一个 LED。同时分析了程序的代码。测试结果说明了，库函数使用方便，但是效率低下，寄存器效率高，但是使用不方便。因此在设计系统的时候如何优化是需要综合考虑的。

## CH03\_EMIO 实验

### 3.1 EMIO 和 MIO 的对比介绍

上次讲到 MIO 的使用，初步熟悉了 EDK 的使用，这次就来说说 EMIO 的使用。如你所见 zynq 的 GPIO，分为两种，MIO(multiuse I/O)和 EMIO(extendable multiuse I/O)



MIO 分配在 bank0 和 bank1 直接与 PS 部分相连，EMIO 分配在 bank2 和 bank3 和 PL 部分相连。除了 bank1 是 22-bit 之外，其他的 bank 都是 32-bit。所以 MIO 有 53 个引脚可供我们使用，而 EMIO 有 64 个引脚可供我们使用。

使用 EMIO 的好处就是，当 MIO 不够用时，PS 可以通过驱动 EMIO 控制 PL 部分的引脚，接下来就来详细介绍下 EMIO 的使用。

EMIO 的使用和 MIO 的使用其实是非常相似的。区别在于，EMIO 的使用相当于，是一个 PS + PL 的结合使用的例子。所以，EMIO 需要分配引脚，以及编译综合生成 bit 文件。

### 3.2 电路分析与实验现象

本节我们将使用 Miz 系列开发的 LED，通过 SDK 操作 EMIO 来控制 LED 灯的流水操作。

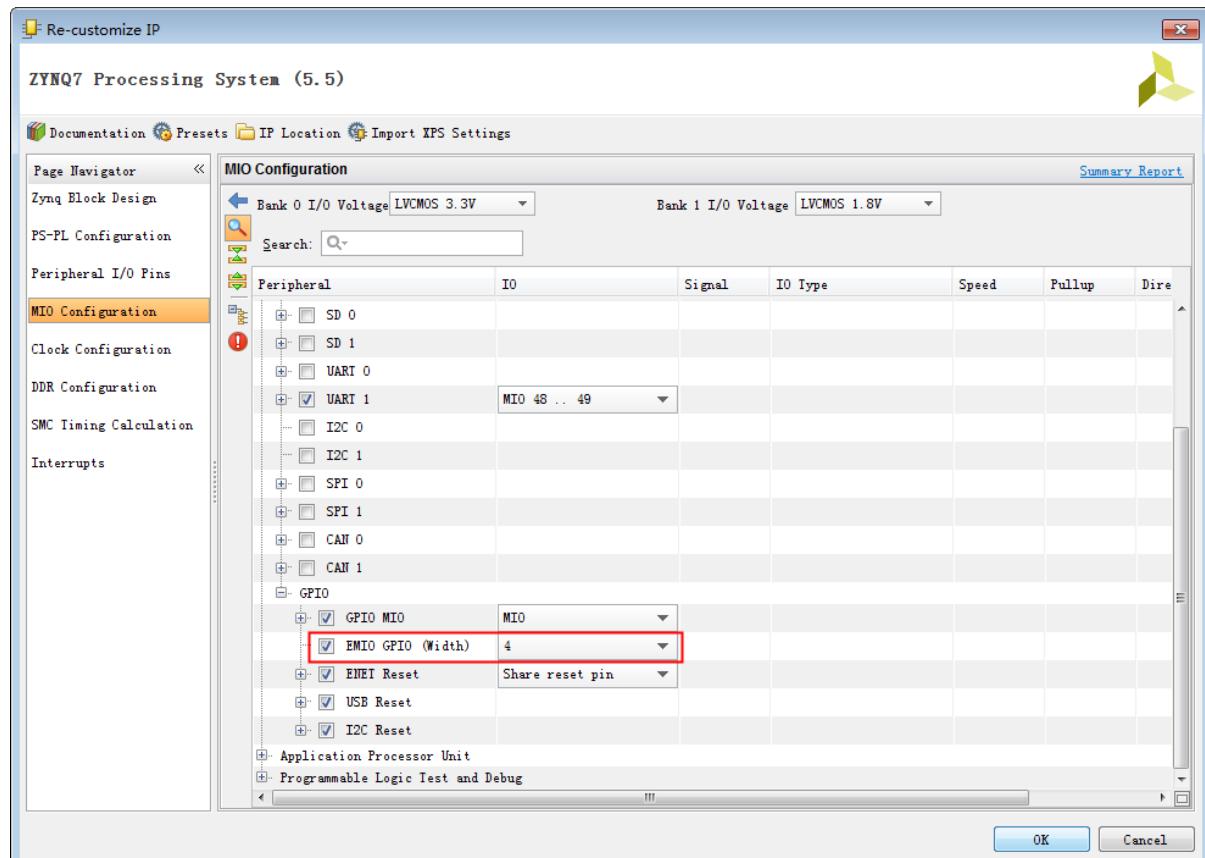
### 3.3 创建 VIVADO 工程

Step1:新建一个名为为 Miz\_sys 的工程，芯片类型根据自身情况设置。

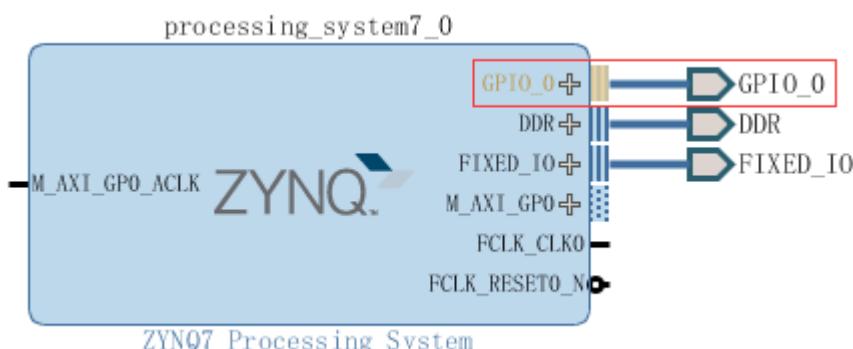
Step2: 创建一个 BD 文件，并命名为 system。

Step3: 添加 ZYNQ7 Processing System，根据自己的硬件类型配置好输入时钟频率与内存型号。

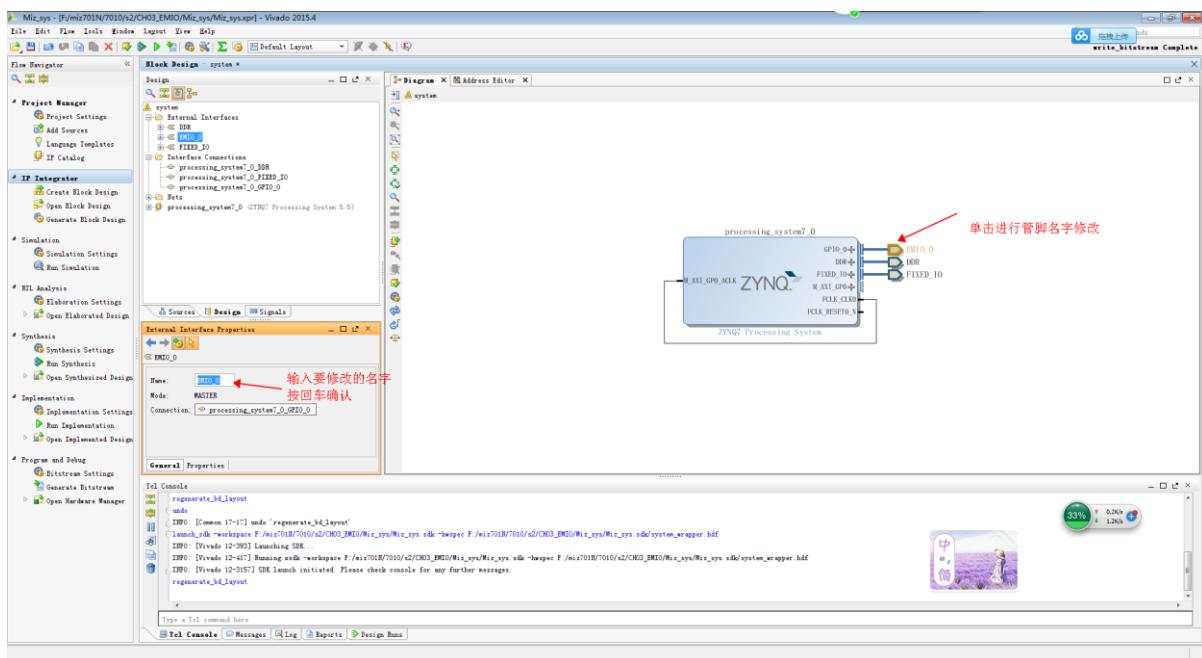
Step4: 在 MIO Configuration 选项卡，再看到 I/O Peripherals 中的 GPIO 一栏，勾选上其中的 EMIO 一栏，并选择 4 位引脚输出（最多可以选择 64 位，但是这个使用只需要 4 位足够了。）



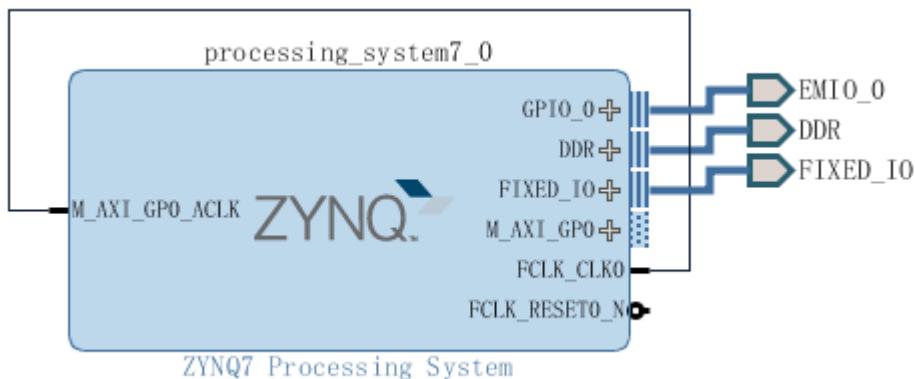
Step5: 单击 OK，仔细观察发现的 zynq 核心多出一组引脚名为 GPIO\_0，这个正是我们刚刚设置的一组 EMIO，我们右击该引脚，选择 make external 把 GPIO\_0 引脚引出（或者单击该引脚处，按快捷键 Ctrl +t，也可以将引脚引出）。效果如下图所示：



Step6: 单击 GPIO\_0，将其修改为 EMIO\_0，如下图所示：



Step7：接着，将如下两引脚连接起来，其实就是给 M\_AXI\_GPO\_ACLK 提供一个时钟。

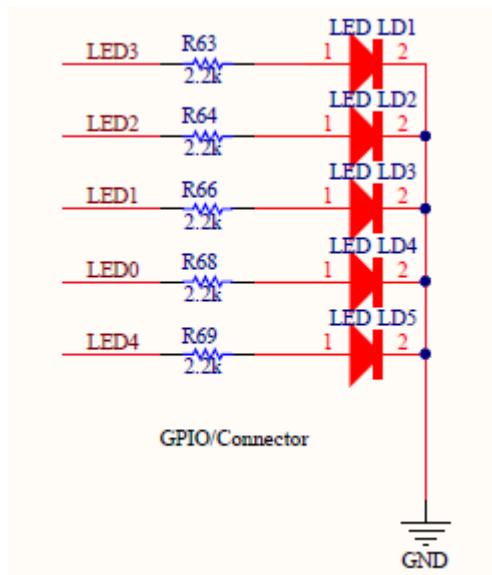


Step8：右键单击 Block 文件，文件选择 Generate the Output Products。

Step9：单击 Block 文件，选择 Create a HDL wrapper，根据 Block 文件内容产生一个 HDL 的顶层文件，并选择让 vivado 自动完成。

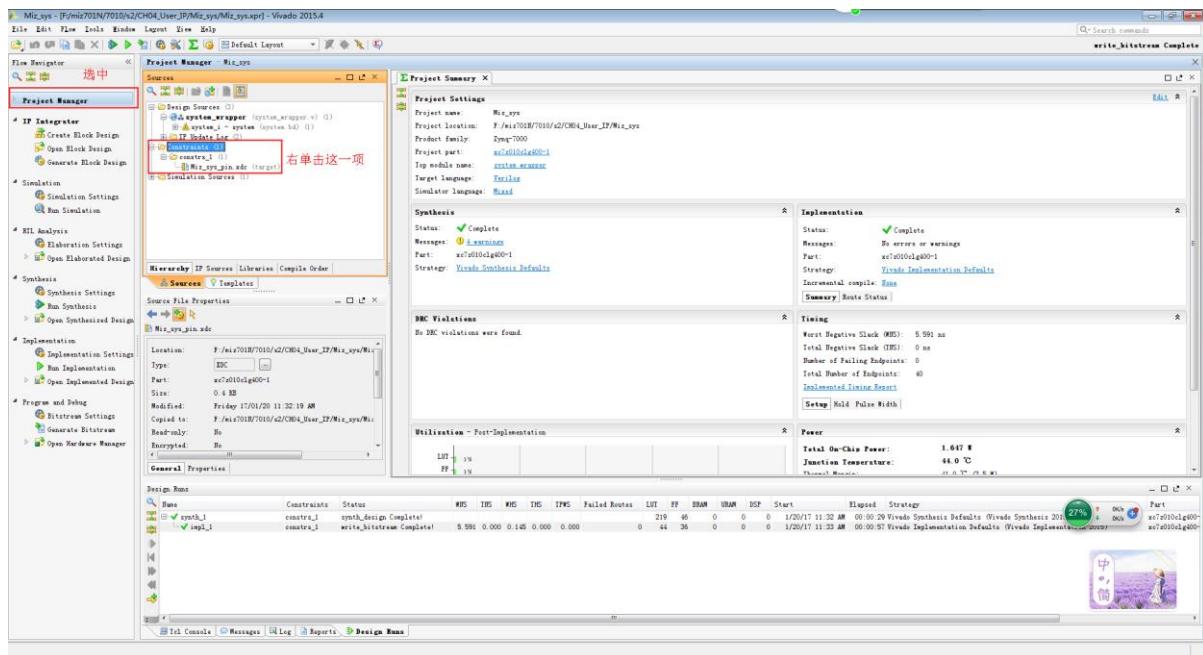
### 3.4 创建约束文件

根据自身的硬件，对芯片的引脚进行分配，首先打开我们提供的原理图文件，，MZ7X 的 LED 部分原理图如下所示：



此处我们选择 LED0–LED3 分配给 EMIO。

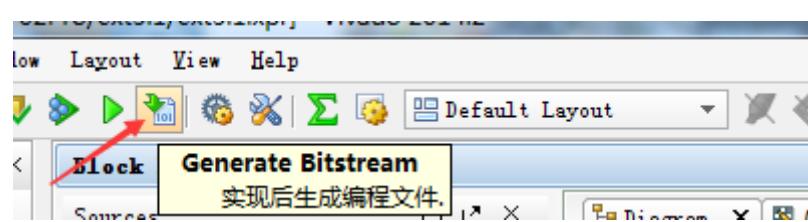
Step1：选中 Project manager，然后右单击 Constraints，选择 Add Sources。



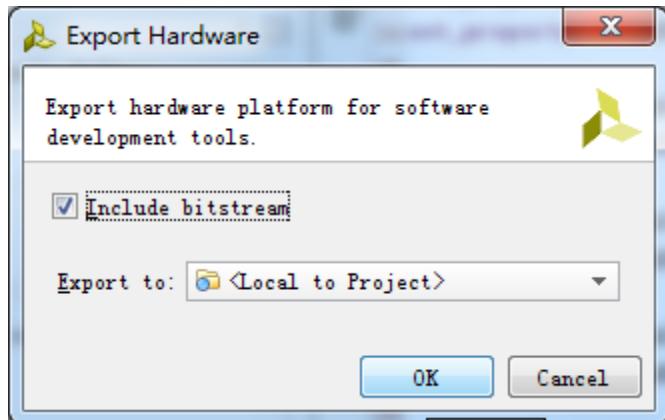
Step2:按照之前介绍的方法，在我们提供的源程序包的 DOC 文件夹下找到 XDC 文件夹，将其中的约束文件添加到工程当中来。

### 3.5 产生 bit 文件并导入到 SDK 中

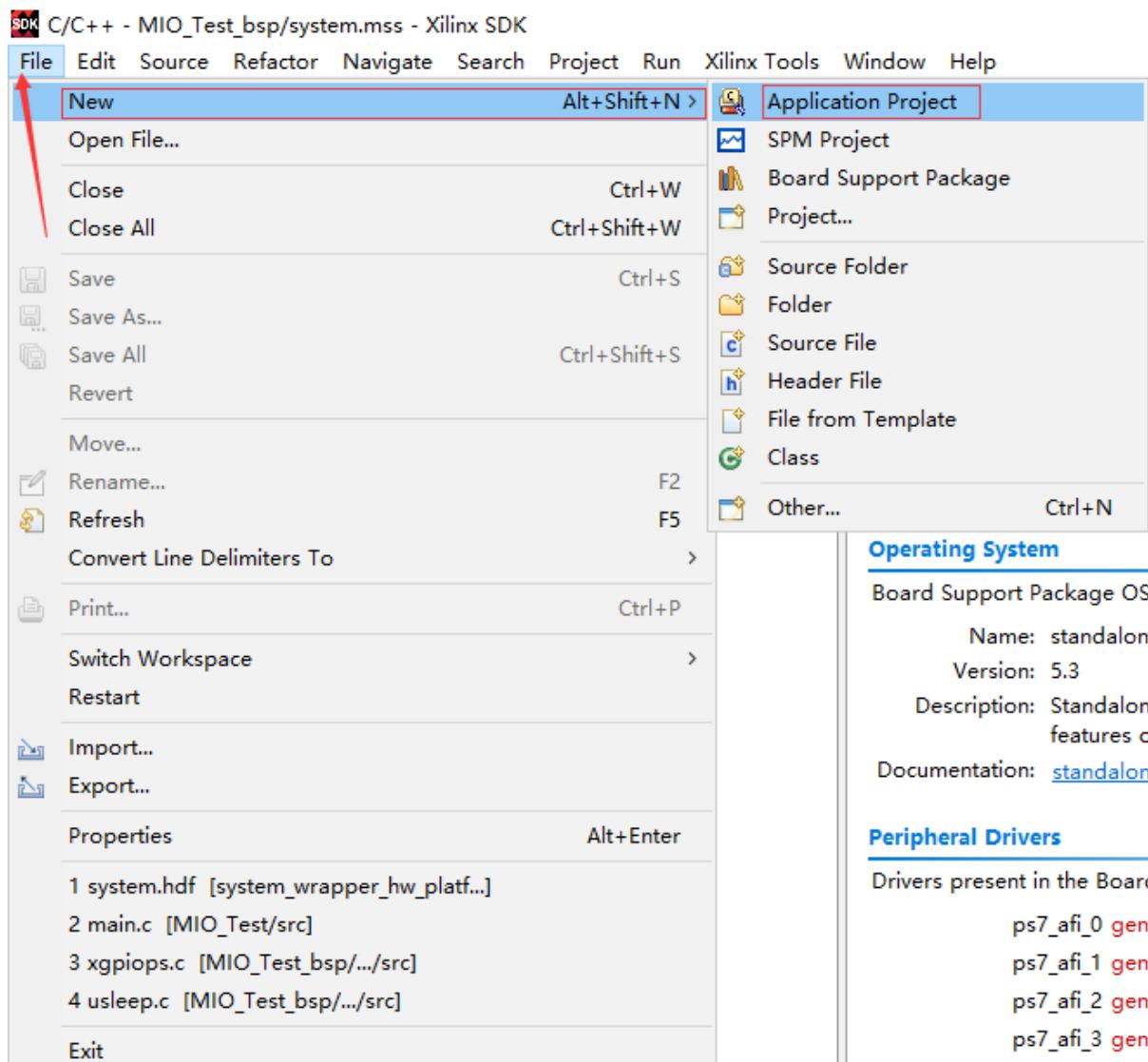
Step1：生成 bit 文件。



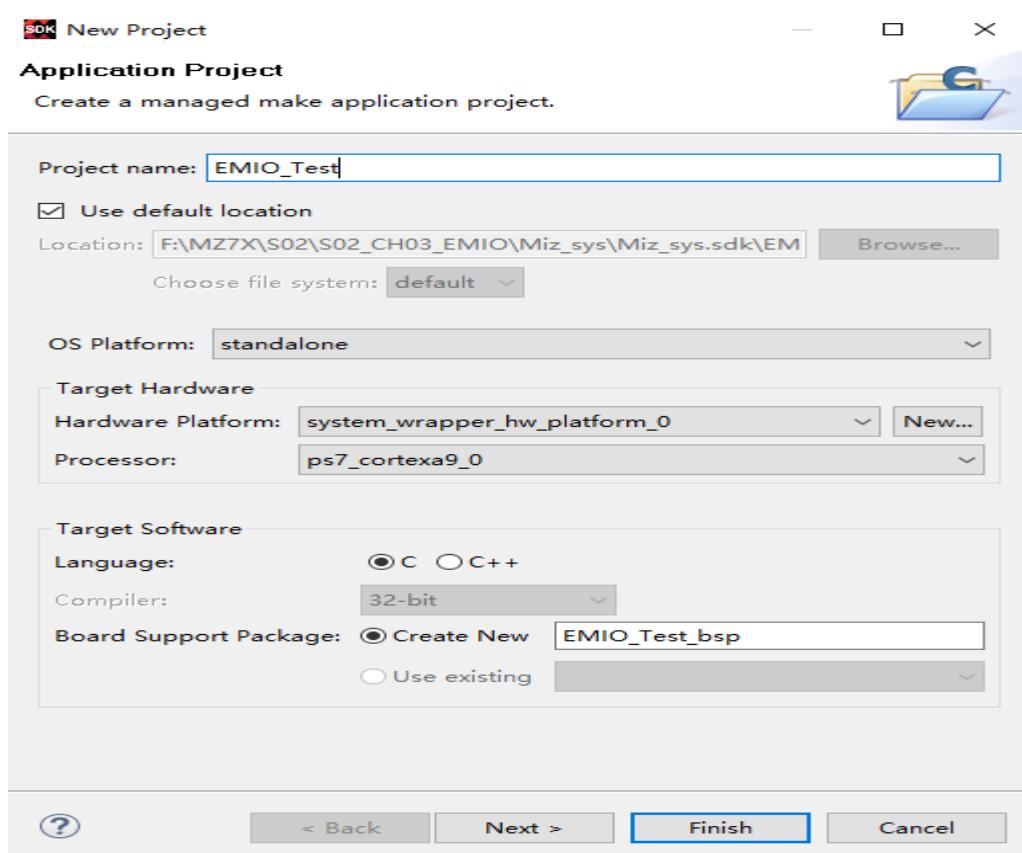
Step2: 导出到硬件。



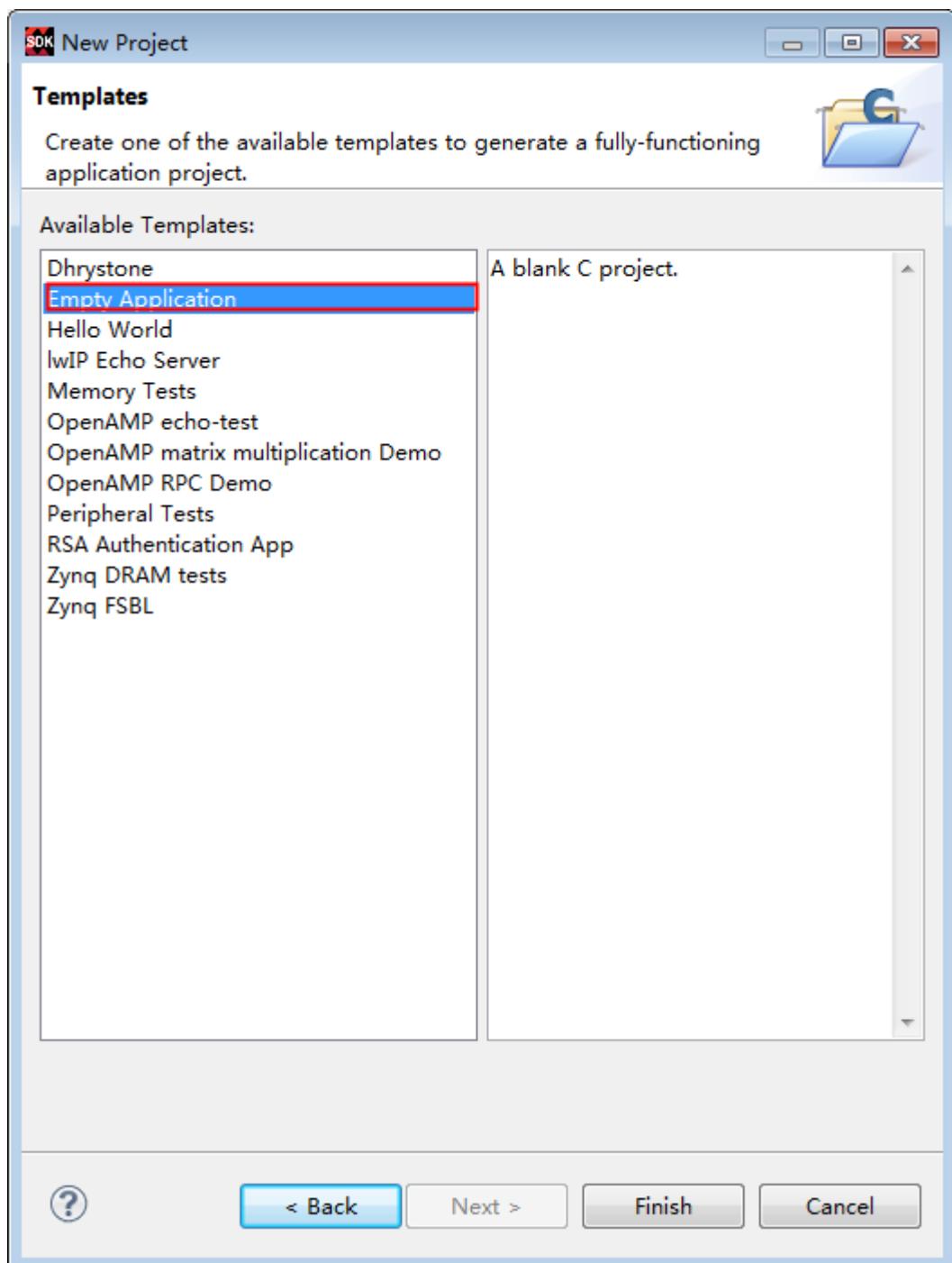
Step3: 打开 SDK, 单击 File-New-Application project。



Step4:输入工程名字，此处命名为 EMIO\_Test，单击 Next。



Step5: 选择 Empty Application, 创建一个空的工程, 单击 Finish 完成创建。



Step6: 在我们提供的源程序文件夹中找到 DOC 文件夹下的 C\_Driver 文件夹, 将其中的设计文件复制一下。

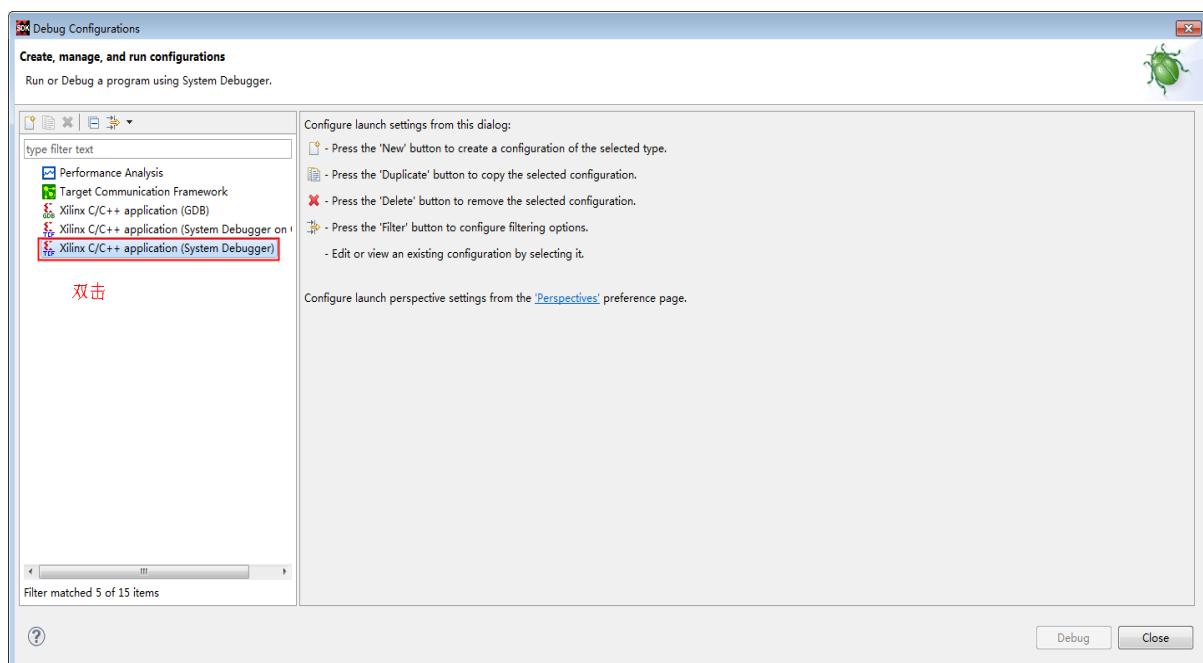


Step7: 点击 EMIO\_Test 旁边的箭头使其展开，然后选中 src, 按下 Ctrl+V 快捷键完成粘贴。

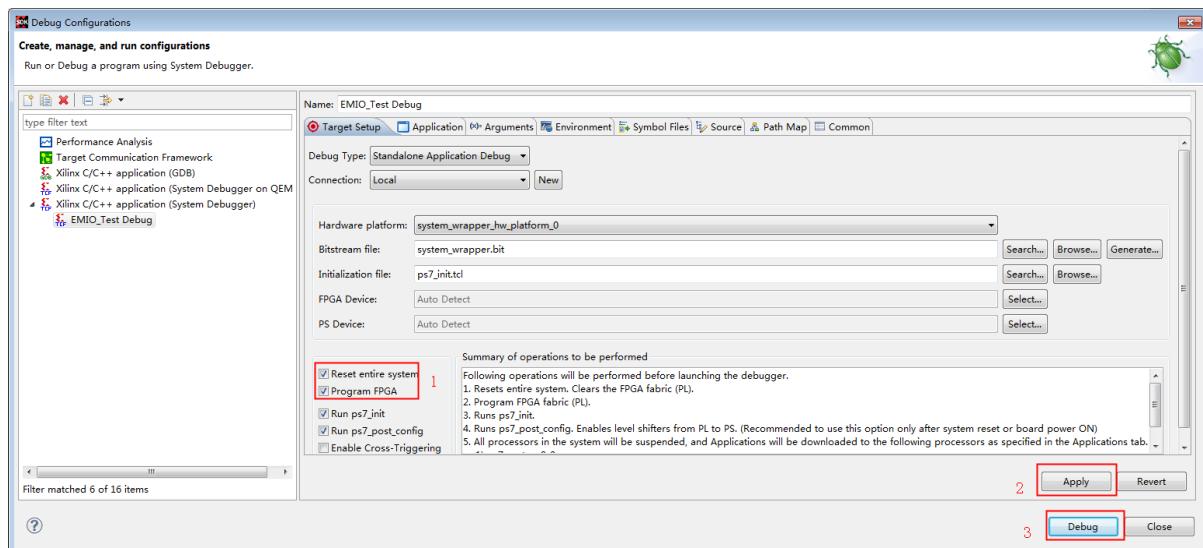


Step9: 右击工程，选择 Debug as ->Debug configuration。

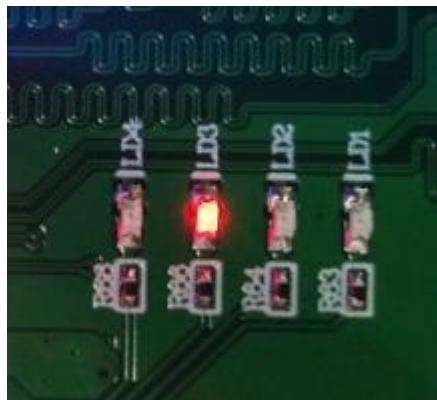
Step10: 选中 system Debugger,双击创建一个系统调试。



Step11: 设置系统调试。

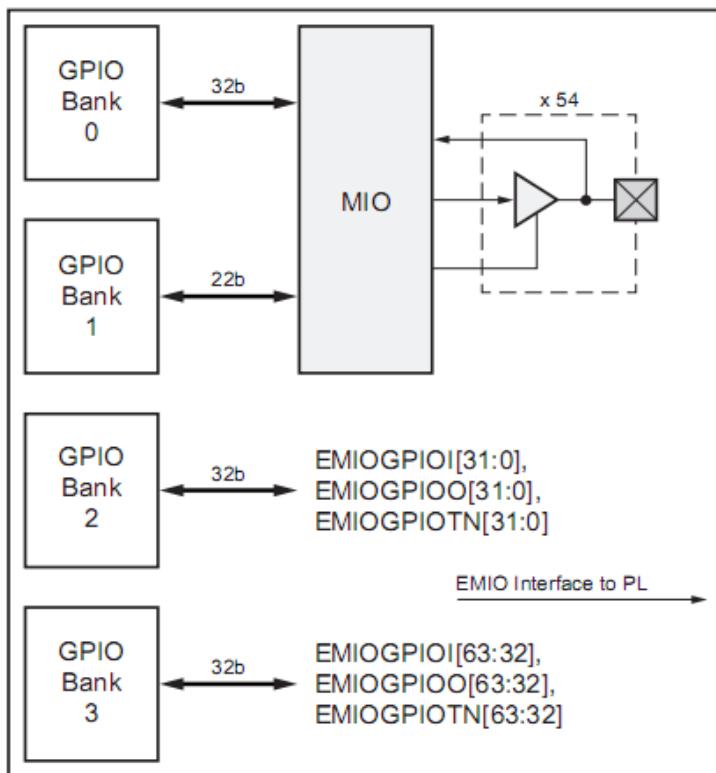


Step12：单击窗口上的运行按钮，运行程序，可看到 LED 的流水操作。



### 3. 6 程序分析

本章程序与第二章 MIO 基本上是一模一样的，如果还有不懂得地方请返回去查看第二章程序的分析，这里不再重复的讲解。这里需要注意的是本章程序中为什么要定义成 54 开头呢？答案如下图所示：



因为 MIO 和 EMIO 是同一编号的 MIO 共 54 个，从 0~53。而从 54 开始就开始是 EMIO 了的范围了。之前我们应出了 4 个引脚 emio\_0\_tri\_io[0]~emio\_0\_tri\_io[3],他们其实就依次对应 54~57 这几个序号，同时也对应了我们开发板上的 4 个 LED（这是引脚约束的结果）。

### 3.7 本章小结

通过本章的学习，我们掌握了在 MIO 不够使用的情况下，通过 PL 部分扩展 EMIO 增加 IO 的使用量。并且通过一个简单的例子演示了如何添加 EMIO IP 并且启动 SDK 通过 JTAG 下载调试的方法。

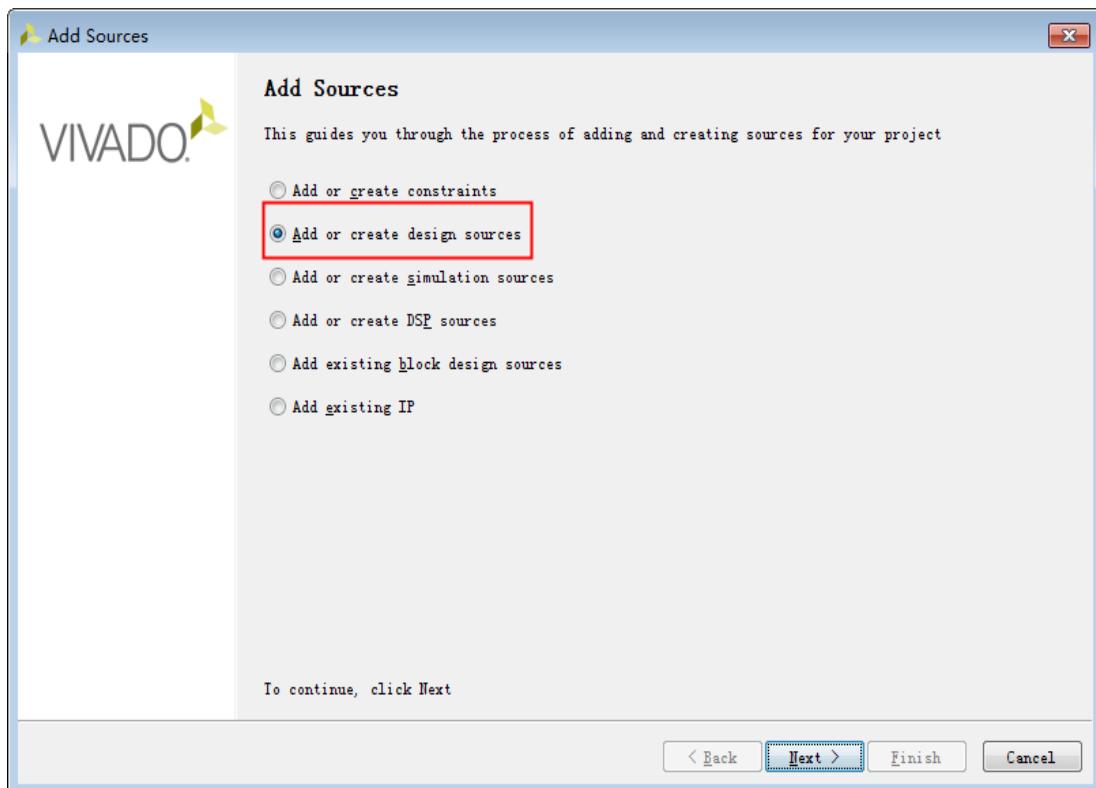
## CH04\_User\_IP 实验

### 4.1 创建 IP

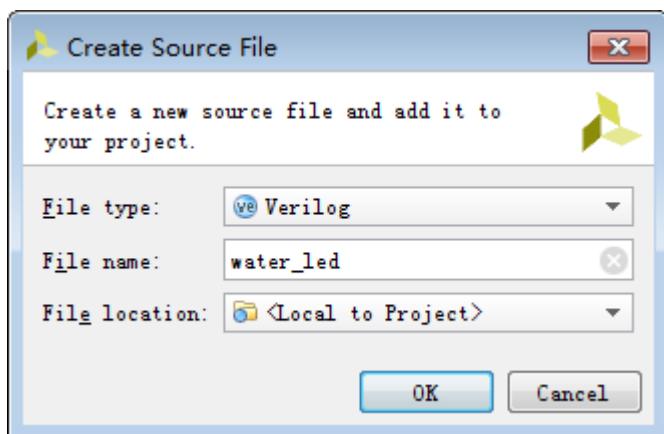
在之前的教程中，我们通过 MIO 与 EMIO 来控制 LED，所使用的也是官方的 IP，实际当中，官方提供的 IP 不可能涵盖到方方面面，用户需要自己编写硬件描述语言，然后将其封装成 IP 来使用，本节就将详细的讲解如何在 VIVADO 中创建用户自定义的 IP。

Step1：打开 VIVADO 软件，新建一个工程。

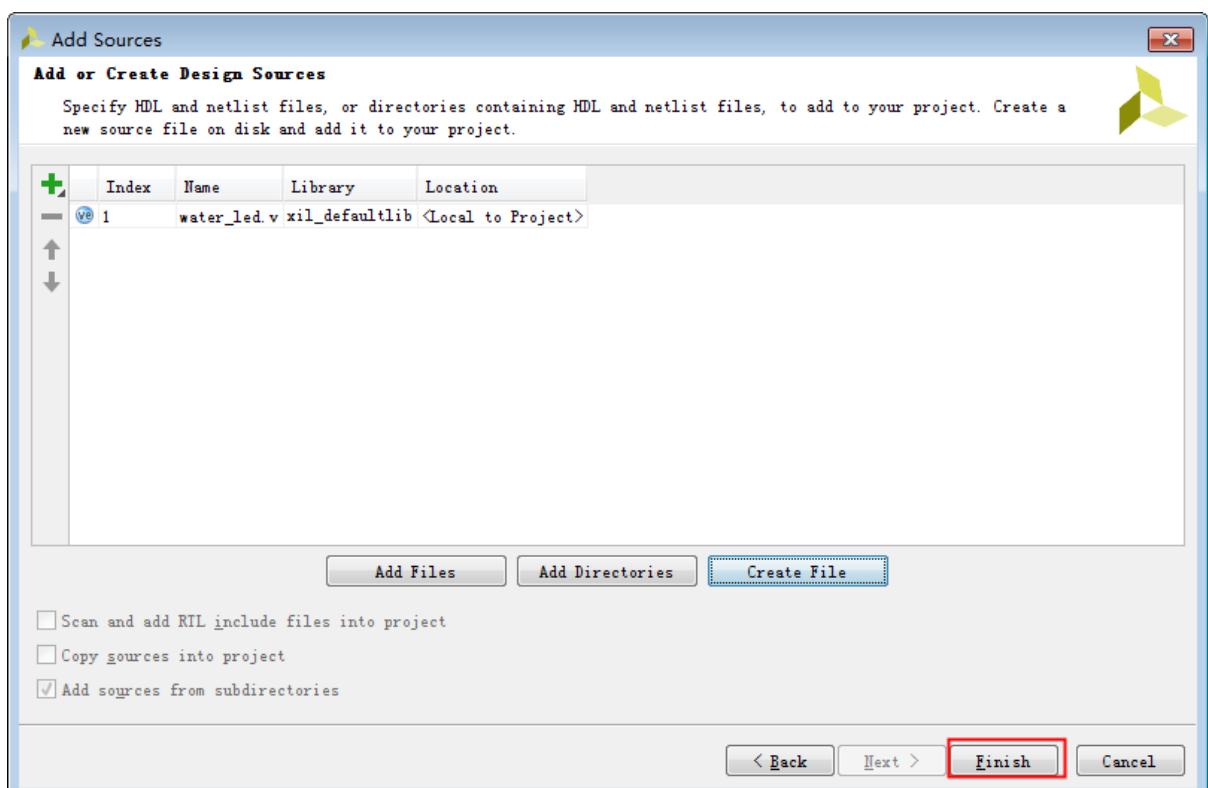
Step2：单击 Add Source，选择 Add or Create design Sources,然后单击 Next。



Step3：单击 Create File，输入文件名，单击 OK。



Step4: 单击 Finish, 完成 Verilog 文件的创建。

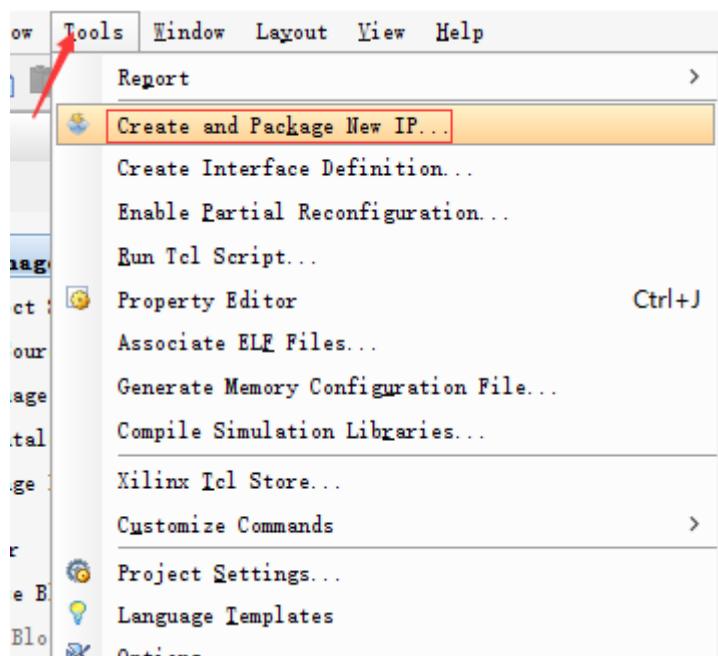


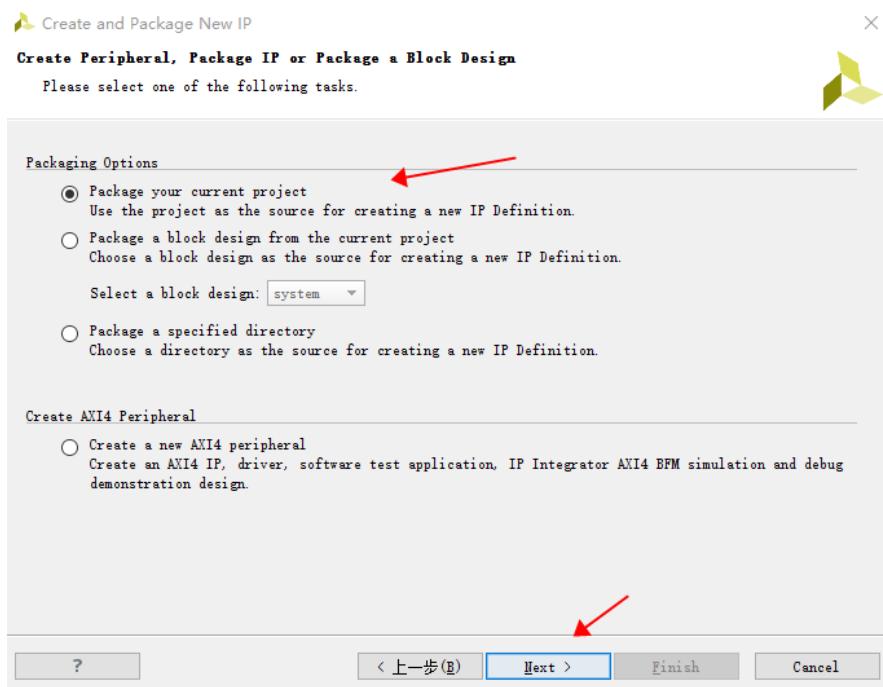
Step5: 将以下代码复制入文本编辑区内。

```
module LED_ML(
    input CLK_i,//100MHZ
    input RSTn_i,
    output reg [3:0]LED_o
);
    reg [31:0]C0;
    always @(posedge CLK_i)
```

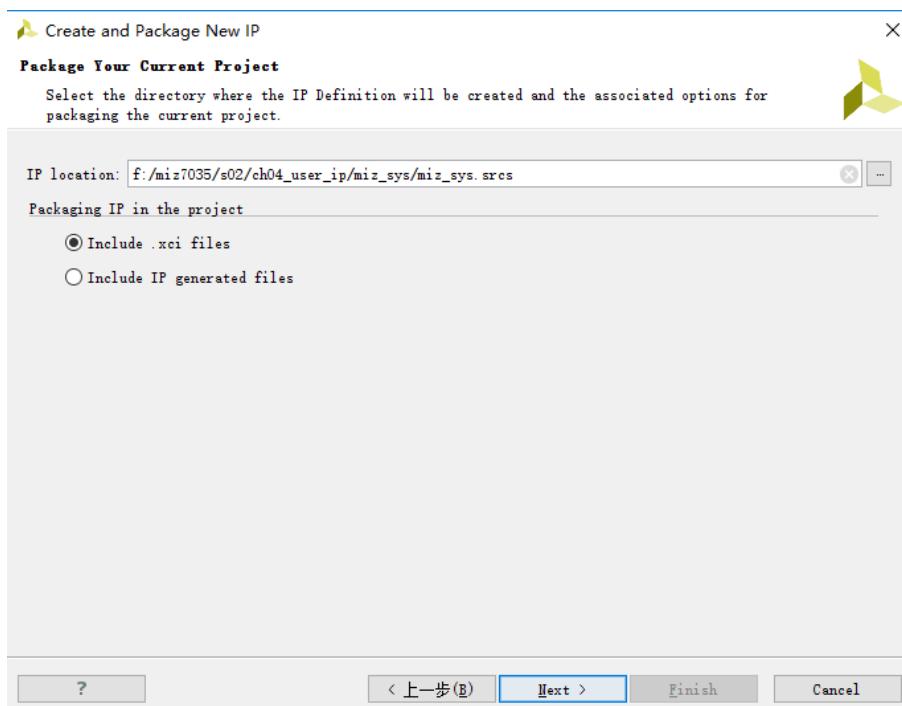
```
if(!RSTn_i)
begin
LED_o <= 4'b0001;
C0 <= 32'h0;
end
else
begin
if(C0 == 32'd49_999_999)//1s
begin
C0 <= 32'h0;
if(LED_o == 4'b1000)
LED_o <= 4'b0001;
else LED_o <= LED_o << 1;
end
else begin C0 <= C0 + 1'b1; LED_o <= LED_o; end
end
endmodule
```

Step6: 单击 Tools—>Create and package IP，单击 Next。





Step7: 选择 IP 的保存路径，单击 Next。

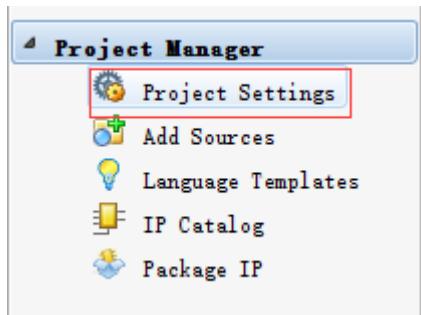


Step8: 单击 Finish 完成封装。

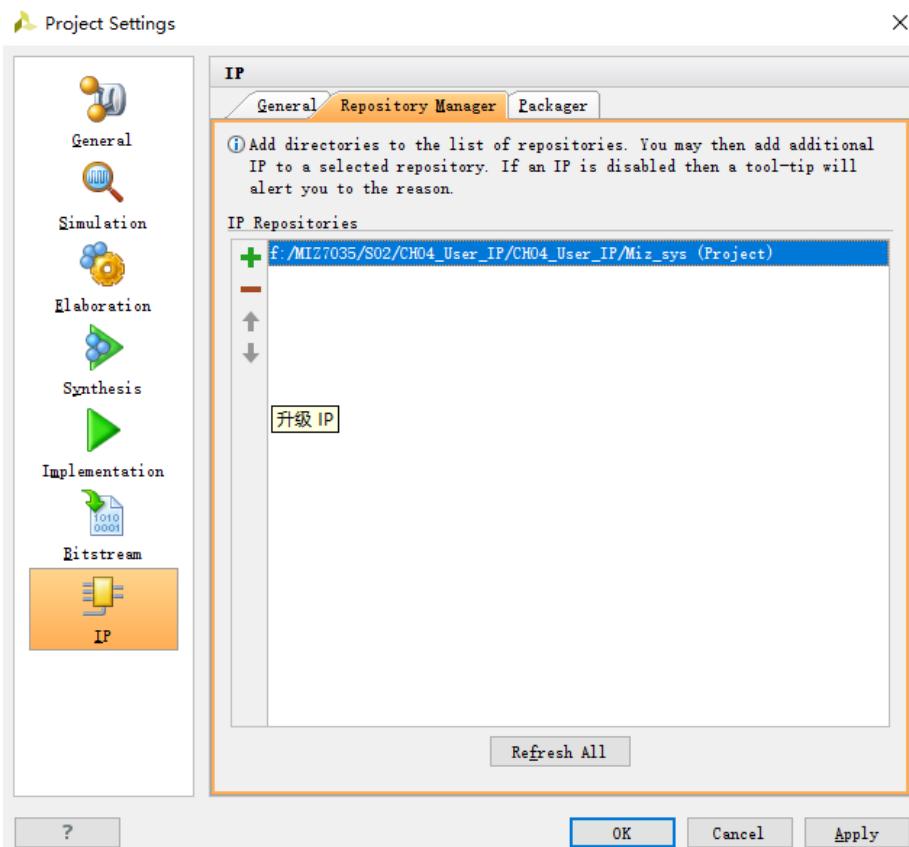
## 4. 2 调用自定义 IP

Step1: 另外新建一个 VIVADO 工程，根据自己的开发板正确配置芯片型号。

Step2: 在 Project manager 区中单击 Project settings。



Step3: 选择 IP 设置区中的 repository manager, 。

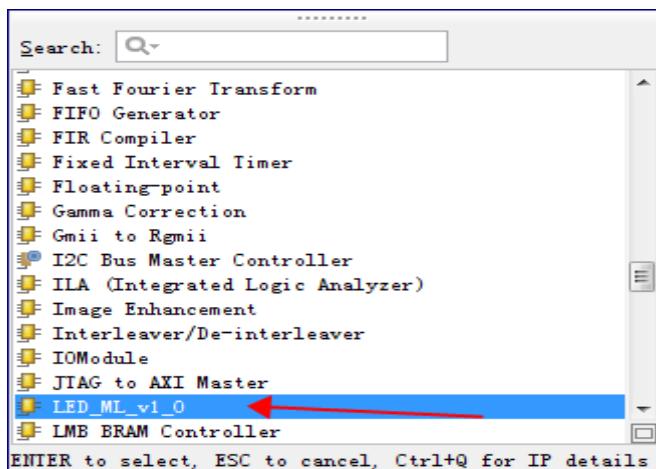


Step4: 单击+号图标, 将上一节封装的 IP 的路径存放进去, 单击 OK。

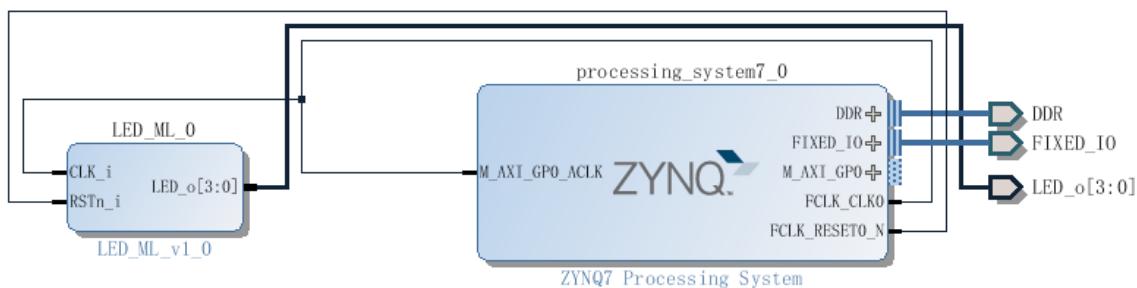
Step5: 新建一个 BD 文件, 输入文件名, 完成创建。

Step6: 向 BD 文件中添加一个 ZYNQ Processing system, 根据自身硬件完成 PS 时钟和内存型号的配置 (还未熟练的复习一下第二章和第三章)。

Step7: 单击添加 IP 图标, 输入上一节我们自定义 IP 的模块名, 将其添加入 BD 文件中。



Step8:按如下电路图完成模块间的连线。



Step9: 右键单击 Block 文件，文件选择 Generate the Output Products。

Step10: 右键单击 Block 文件，选择 Create a HDL wrapper，根据 Block 文件内容产生一个 HDL 的顶层文件，并选择让 vivado 自动完成。

Step11: 选中 Project manager，然后右单击 Constraints，选择 Add Sources。

Step12: 输入文件名，完成创建，将上一章 EMIO 的约束文件 copy 进去。

Step11: 产生 bit 文件。

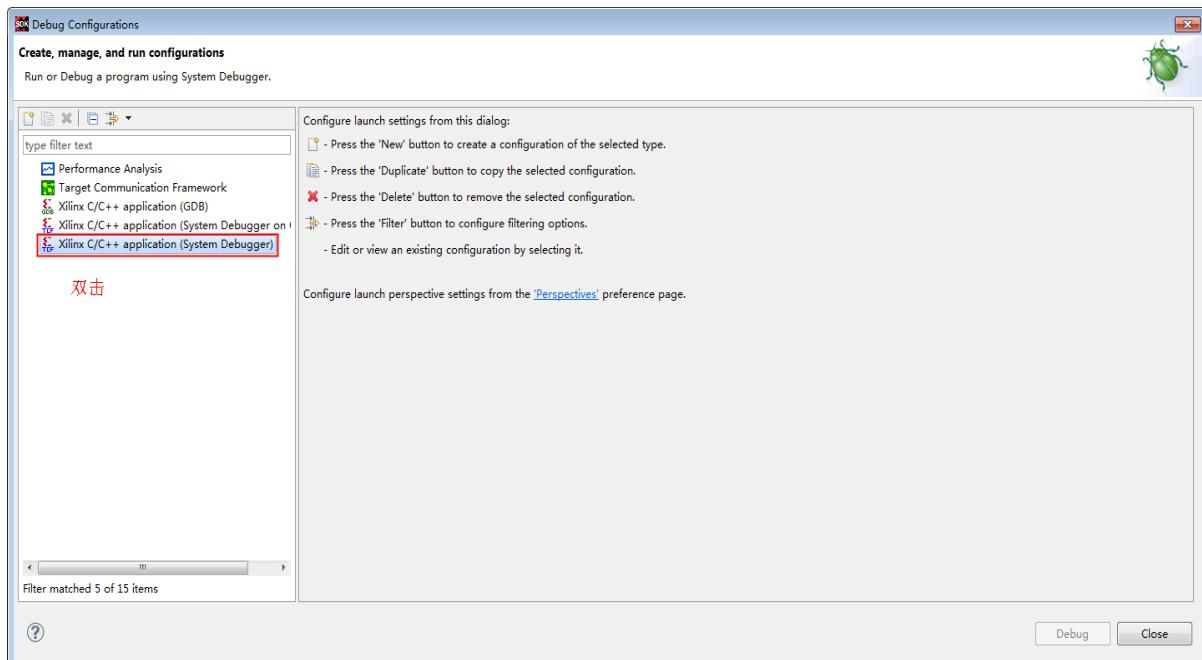
### 4.3 导入到 S D K

由于自定义的 IP 的时钟输入来自于 ZYNQ Processing system, 源时钟是使用的 PS 的时钟，因此需要启动 SDK 整个系统才能启动，而自定义 IP 不需要由 SDK 进行配置，因此我们可以按照前几节讲过的内容，在 S D K 端建立一个 Hello World 工程跑起来就能让自定义 IP 跑起来。

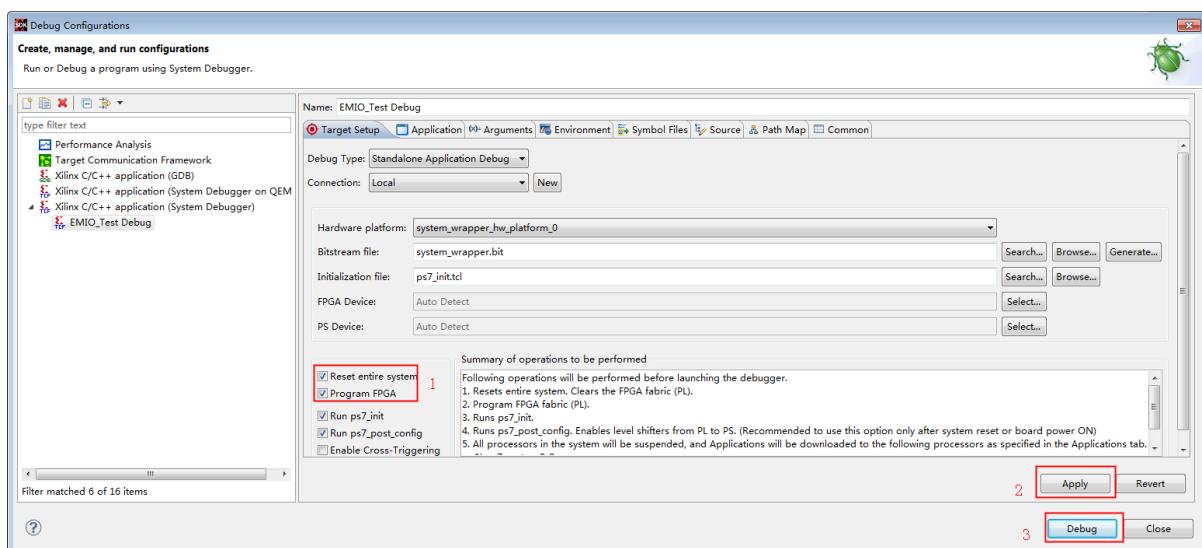
Step1: 创建一个 Hello World 工程。

Step2: 右击工程，选择 Debug as ->Debug configuration。

Step3: 选中 system Debugger, 双击创建一个系统调试。



Step7：设置系统调试。



Step8：单击窗口上的运行按钮，运行程序，可看到 LED 的流水操作。

#### 4.4 本章小结

本章主要介绍了如下在 VIVADO 下创建一个自定义的 IP，内容比较简单，需要注意的是如果工程中使用的源时钟是为 PS 时钟的话，是需要启动 SDK 系统才能正常工作的，若是系统使用到了 ZYNQ Processing System，则系统使用的是 PS 时钟，这是一个判断的依据。在 ZYNQ 的开发中，创建自定义 IP 是一项基本功，还未熟练掌握的要勤加练习。

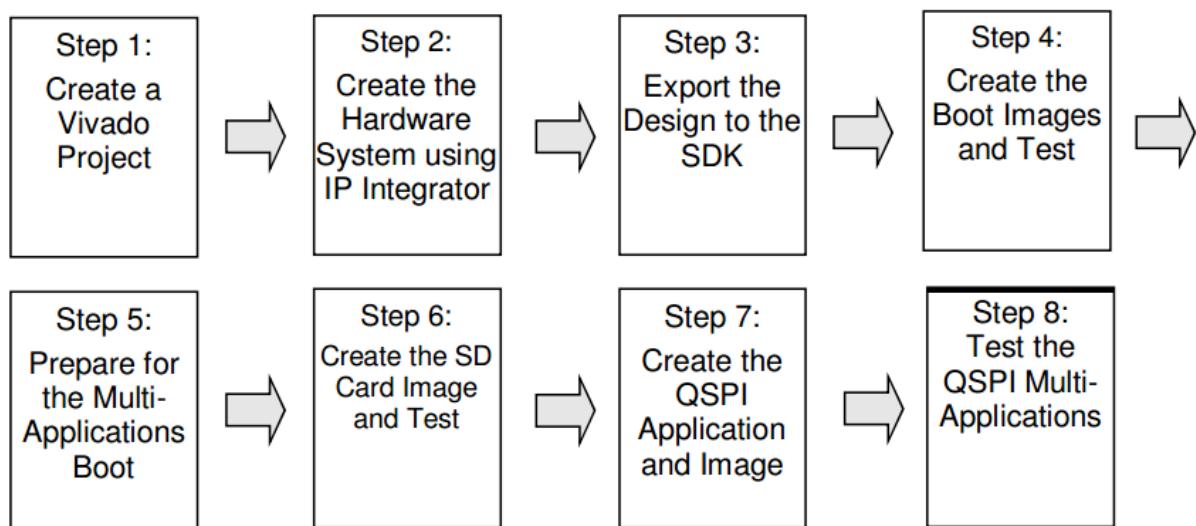
## CH05\_UBOOT 实验

### 5.1 什么是固化

我们前几章的程序都是通过 JTAG 先下载 bit 流文件，再下载 elf 文件，之后点击 Run As 来运行的程序。JTAG 的方法是通过 TCL 脚本来初始化 PS，然后用 JTAG 收发信息，可用于在线调试。但是这样只要一断电，程序就丢失了。还得全部重新来过。

本章介绍通过制作镜像文件，将镜像文件拷贝到 SD 卡，然后将拨码开关拨到 SD 启动，那么每次断电之后程序都会自动从 SD 启动，程序就别固化，而不会掉电丢失了。

### 5.2 固化的流程



### 5.3 固化准备

《第四章 ZYNQ User IP 的使用》实验其实就是一个最简单的“PS + PL”运用的体现。如果我们想固化这个程序，及为这个程序做一个镜像文件，制作改镜像需要哪些材料呢？

首先，想到的两个文件就是 PL 部分需要的 bit 文件，以及 PS 需要的 elf 文件。但是仅仅是这两个文件是远远不够的。我们还需要一段代码吧 bit 文件以及 elf 文件安置好。那么这段代码就是大名鼎鼎的 FSBL.elf。

所以要制作这样一个镜像文件我们需要：FSBL.elf、bit、elf。

最后得到一个等式就是：BOOT.bin = FSBL.elf+该工程.bit+该工程.elf。该工程的 bit 文件和 elf 文件在我们的程序编译完之后都有了，关键是这个 FSBL.elf 这么那里找？不用担心，FSBL.elf 文件 xilinx 找就为我们准备好了，我们可以利用 SDK 生成它。再次之前，我们先简单了解一下 zynq 的启动的过程。

## 5.4 zynq 的从 SD 卡的启动的过程

和大多数 arm 启动过程一样，这个启动过程也分为 3 个阶段，这三个阶段分别称之为阶段 0、阶段 1 和阶段 2。

阶段 0：即传统的 BootROM 过程，zynq 芯片里有个 rom 里面固化了一段不可修改的程序，只有 zynq 一上电，这段程序就会执行，它将对 zynq 的 NAND、NOR、SD 等基本外设控制器进行初始化。把 SD 卡这类易失的存储器件初始化好了之后，就会把其中的程序拷贝到 zynq 的 OCM (On-chip memory)，那么这个被拷贝到片上 RAM 执行的程序就是我们今天要制作的文件——BOOT.bin。

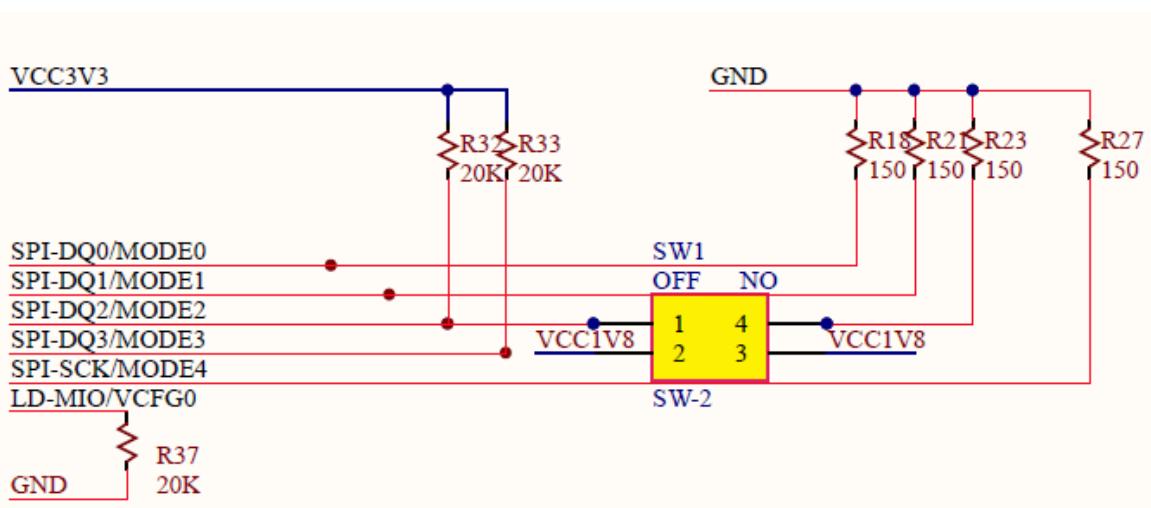
阶段 1：BOOT.bin 加载到 OCM 上就开始执行了，之前说过 BOOT.bin 其实就是由 FSBL.elf+该工程.bit+该工程.e1f 构成。而阶段 1 要做的就是：首先配置 PS 部分，PS 完成初始化后，会去配置 PL 部分，最后还可以去加载阶段 2 的代码。

阶段 2：这一阶段是可选的，主要是为了完成 Linux 系统启动过程。我们这次是还是裸奔，所以暂时不需要。

## 5.5 zynq 启动模式位的选择

这里有个疑问，众所周知 zynq 具有多种启动方式：NOR, NAND, Quad-SPI, SD Card 以及 JTAG 。zynq 如何判断到底从哪里启动呢？事实上，当上电后，zynq 会根据模式管脚的设定选用 boot 的方式。而这个管脚的设定是通过核心板上的拨码开关 (MZ702A 的拨码开关在核心板上)。

MZ7X 模式选择通过拨码开关来实现，当拨码开关 ON 状态接通到 GND 否则接通到 3V3.



MiZ7035 通过拨码开关设置 MIO 的电平状态

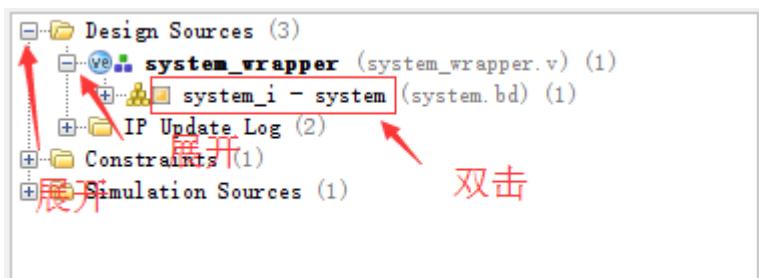
我们的开发板默认拨码的顺序，就是默认的 SD 卡启动，具体模式位应该如何选择如下表所示：

表. 开发板启动模式

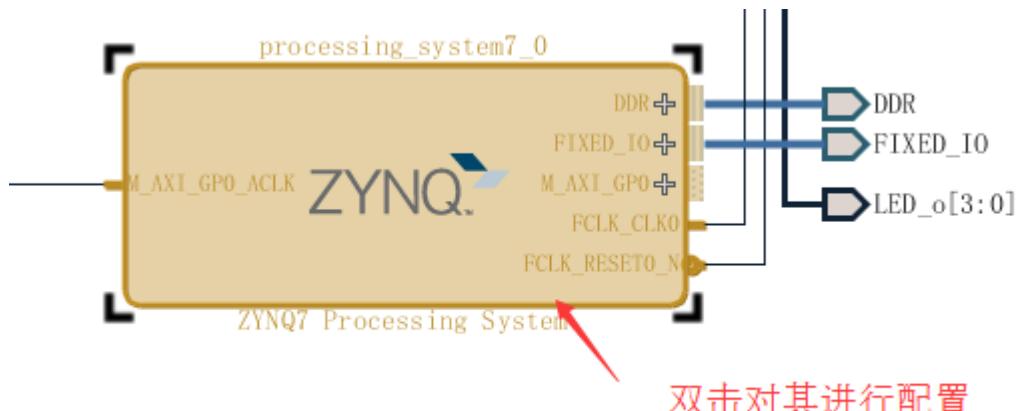
启动模式	开关状态
SD 卡启动/JTAG 调试模式	开关 1-OFF 开关 2-OFF
启动/JTAG 调试模式	开关 1-ON 开关 2-OFF

## 5.6 BOOT.bin 制作过程详解

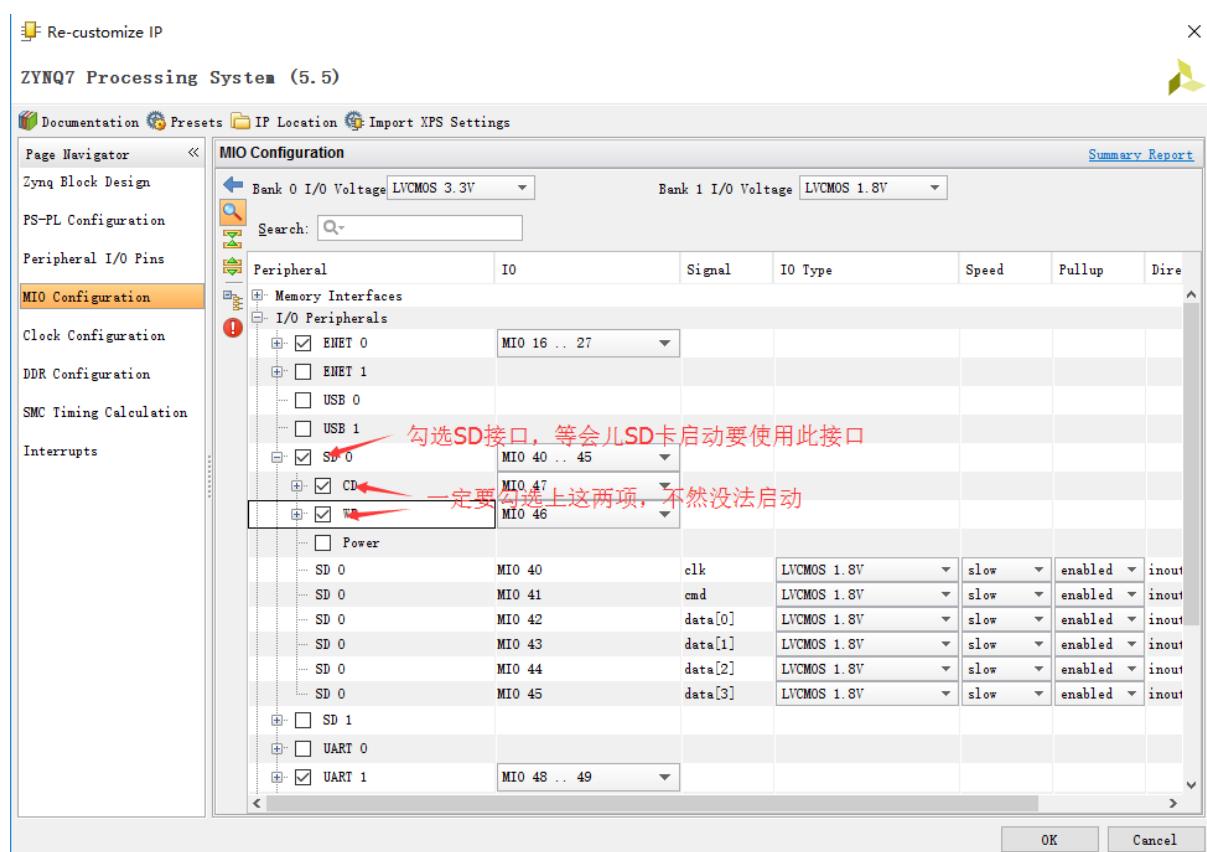
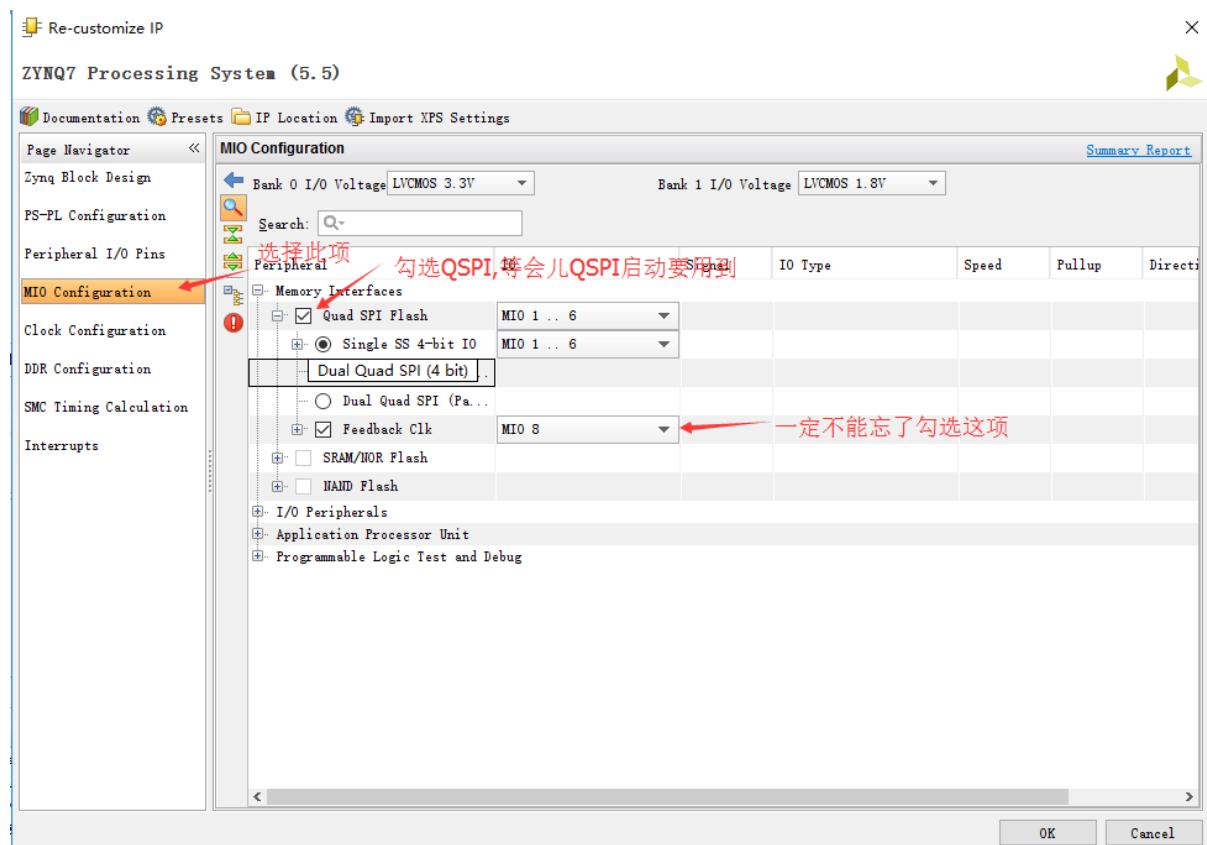
Step1：打开上一章的工程，然后打开硬件原理图。

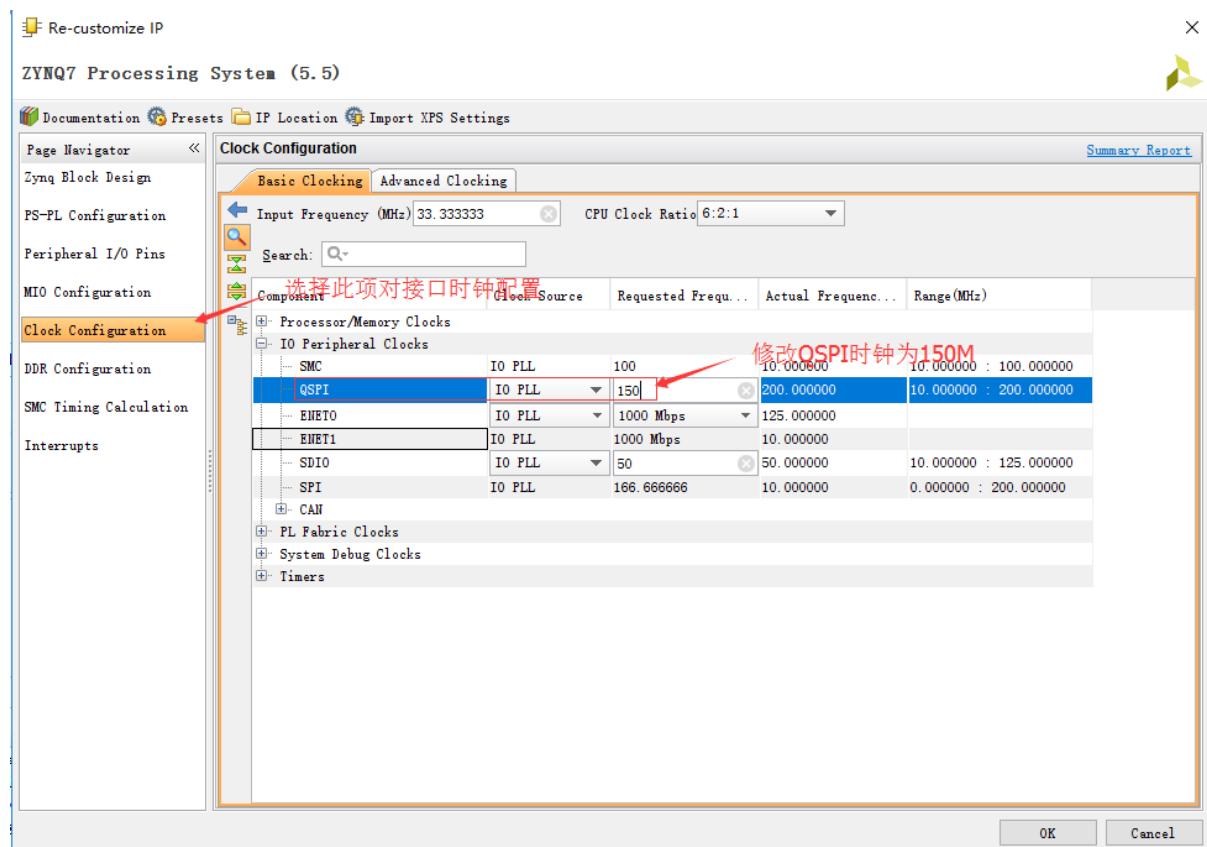


Step2: 双击 ZYNQ Processing System, 对其进行配置：



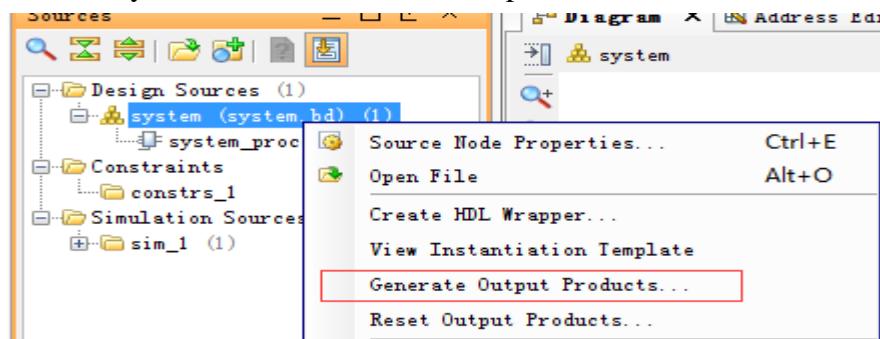
Step3: 选择 MIO Configuration 选项，然后如下图所示配置：

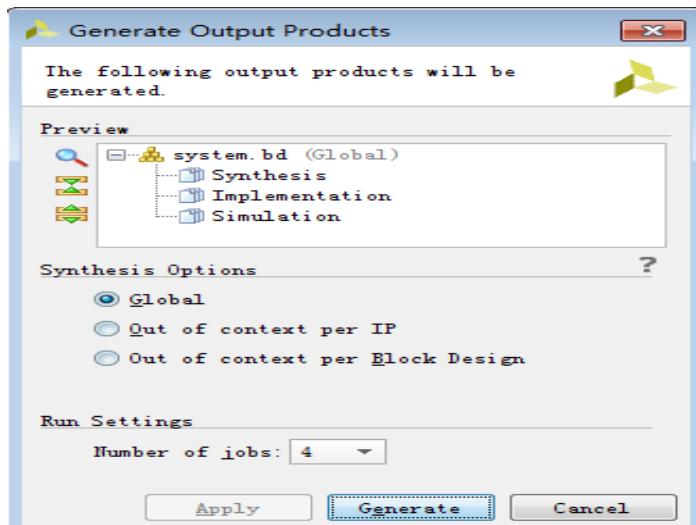




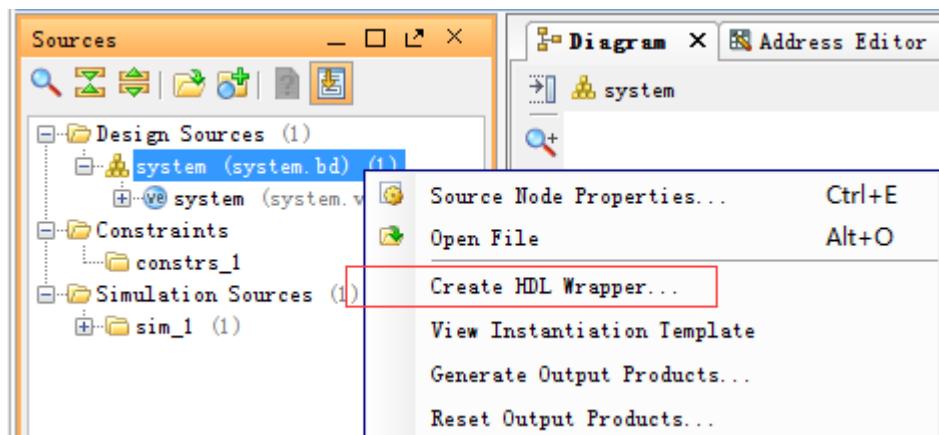
Step4:点击 OK 完成配置。

Step5: 右击 system.bd, 单击Generate Output Products。

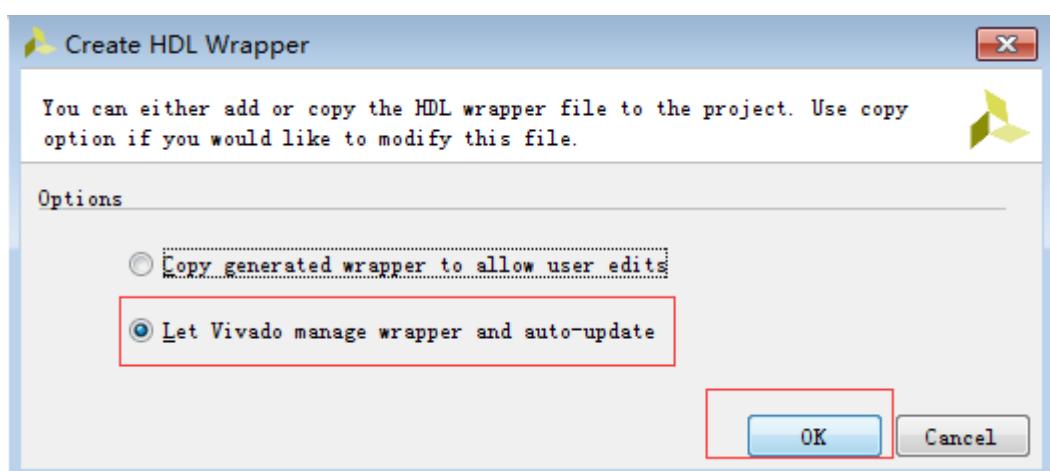




Step6: 右击 system.bd 选择 Create HDL Wrapper 这步的作用是产生顶层的 HDL 文件



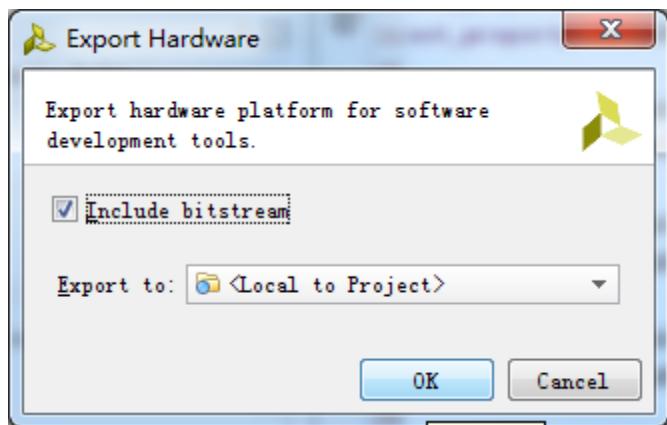
Step7: 选择 Leave Let Vivado manager wrapper and auto-update 然后单击 OK



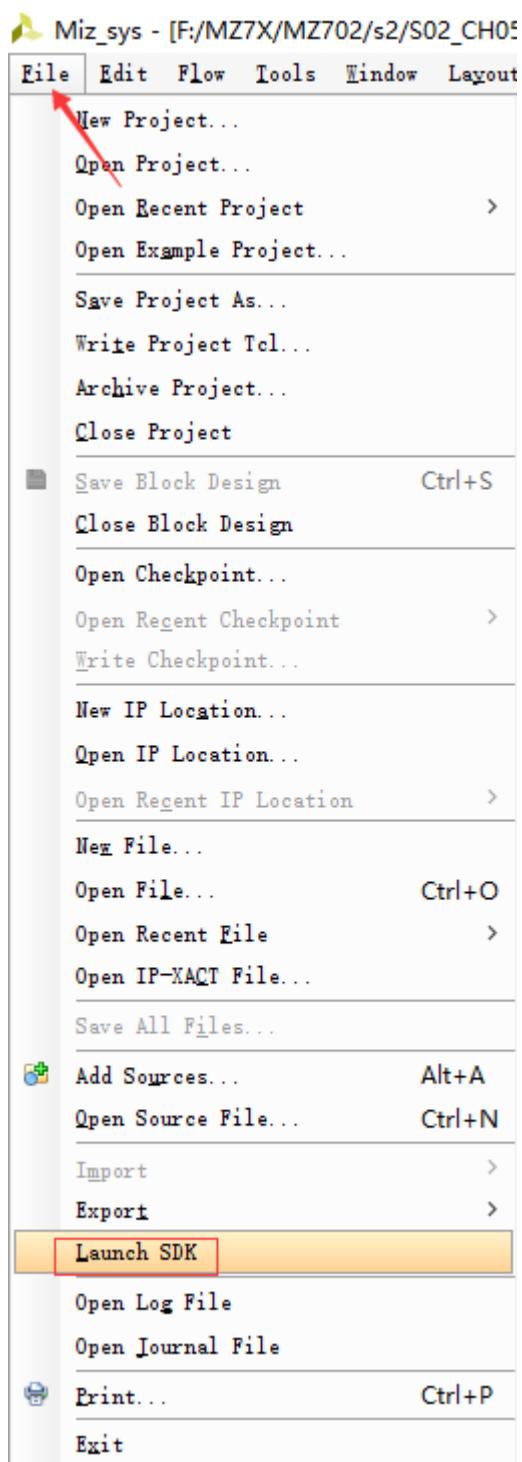
Step8: 生成 Bit 文件。



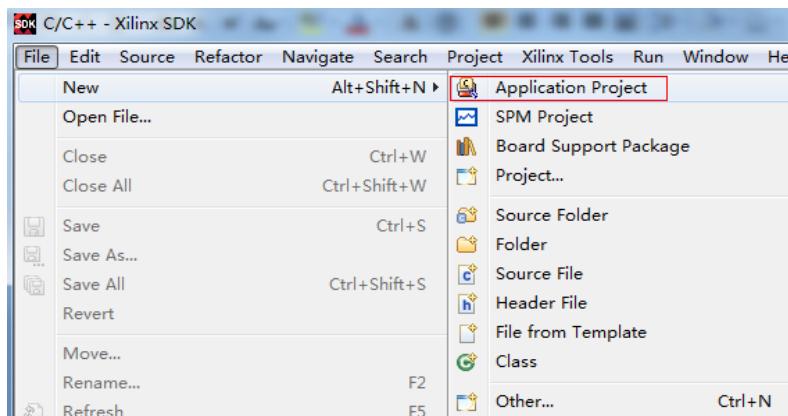
Step9:导出到硬件。



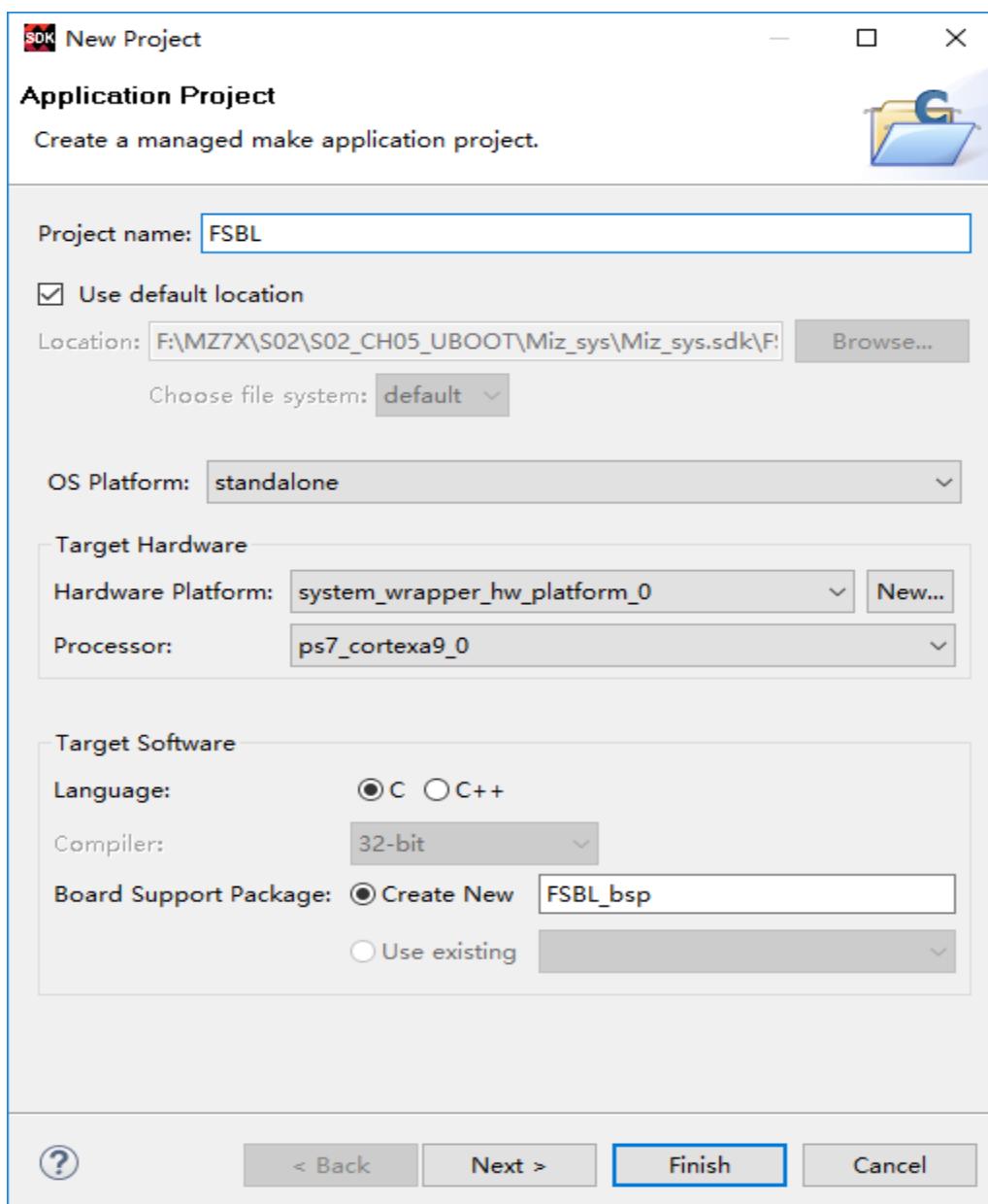
Step10:启动 SDK



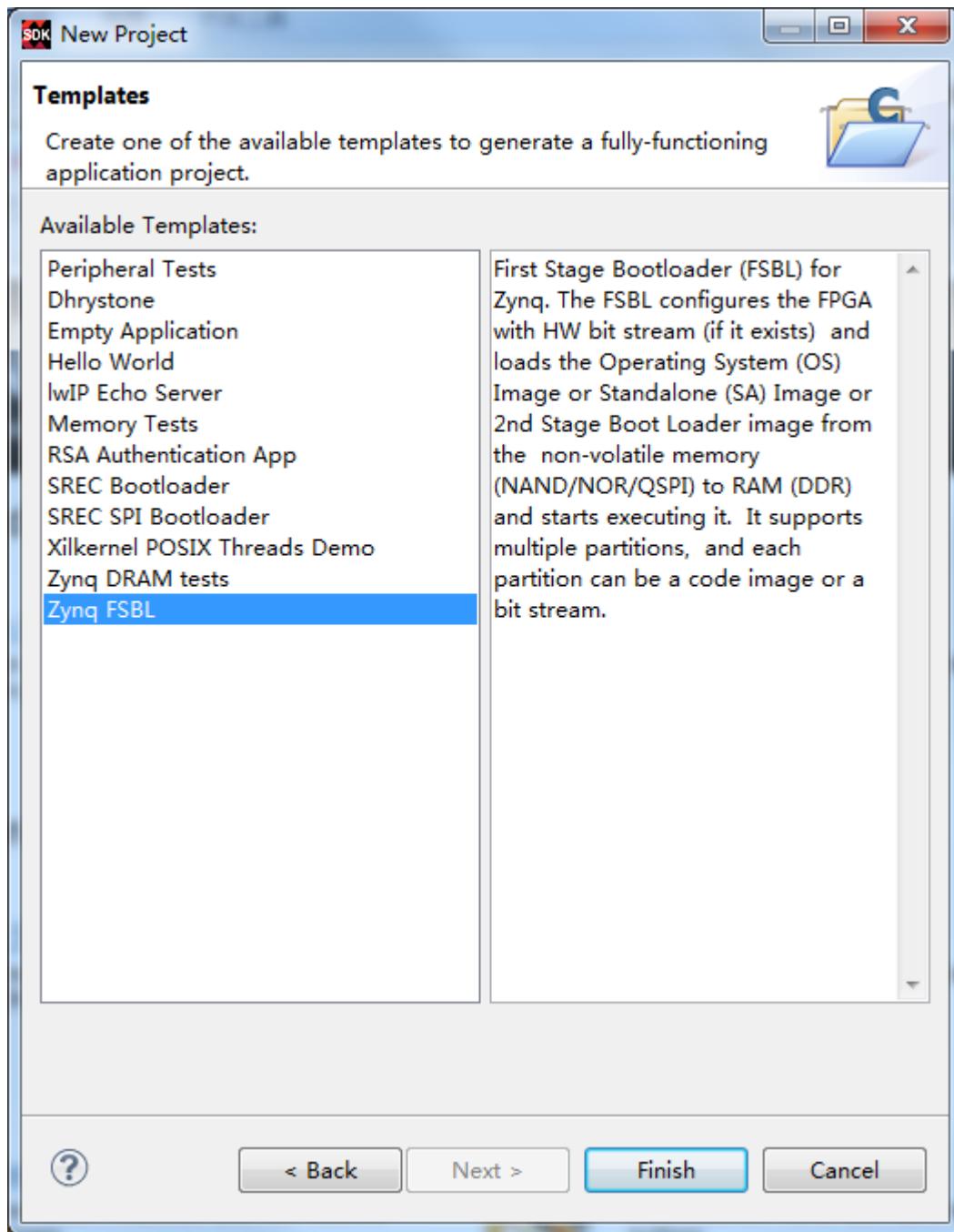
Step11：新建一个应用工程



Step12: 填写工程名，点击 Next

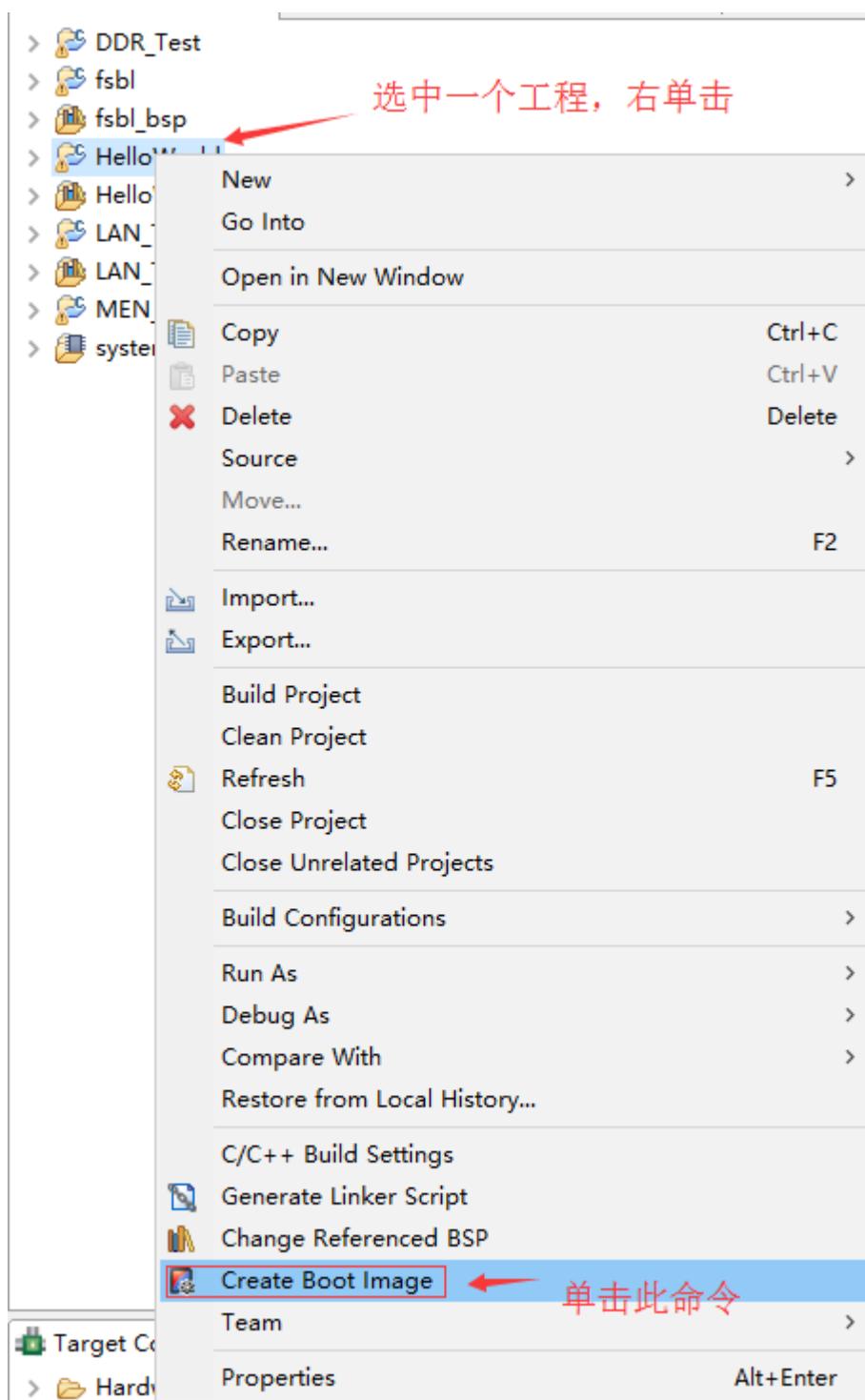


Step13: 现在工程类型为 Zynq FSBL

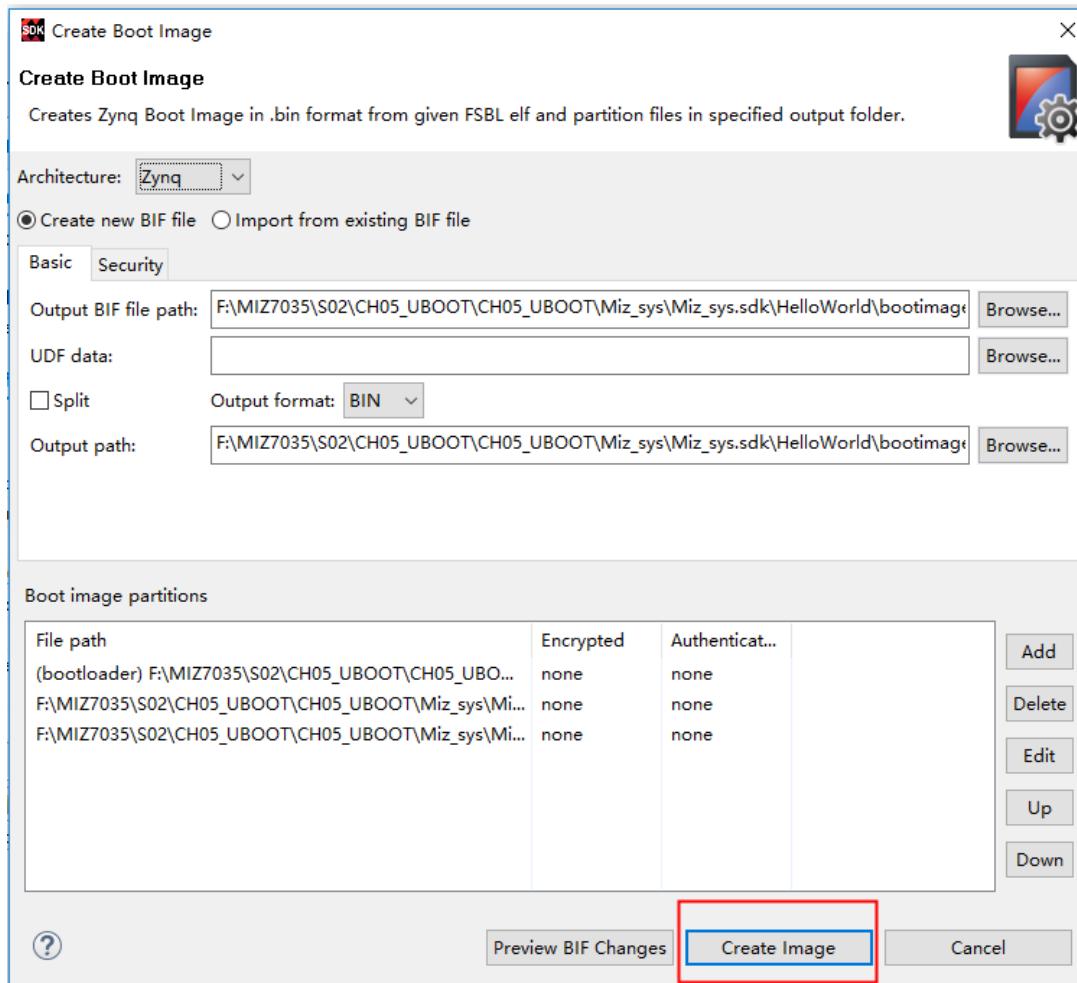


Step14：按快捷键 Ctrl+B 编译一下工程。

Step15：选中其中一个工程（记住是哪一个工程，等会儿好找 Boot.bin），然后右单击，选择如下命令：



Step16: 在新窗口中直接单击 Create Image 即可完成 Boot.bin 的创建，此文件可作为 SD 卡启动文件和 SPI 启动文件。



在之前设定的文件夹下找到，BOOT.bin 并且将其拷到 SD 卡中，再把 SD 卡插到开发板，打开电源，和上次工程出现的现象重现了，这次断电之后，程序也不会消失了~~~  
最后提醒下放大 SD 卡的 bin 文件，一定得叫 BOOT.bin，不然不识别。

## 5.7 从 Quad-SPI 启动

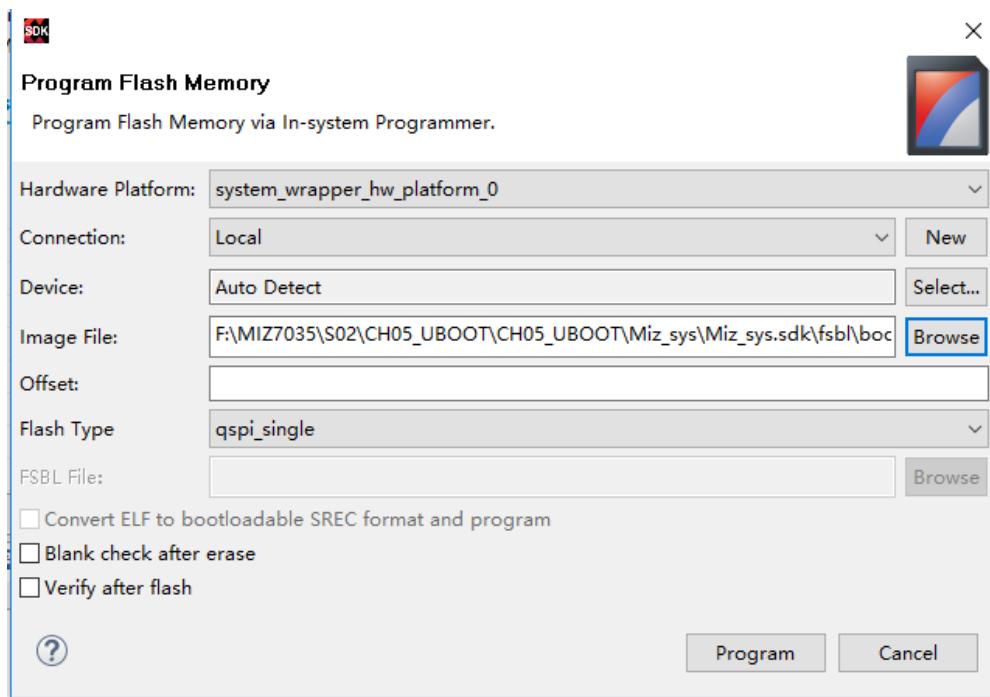
Step1: 设置配置模式

表. 开发板启动模式

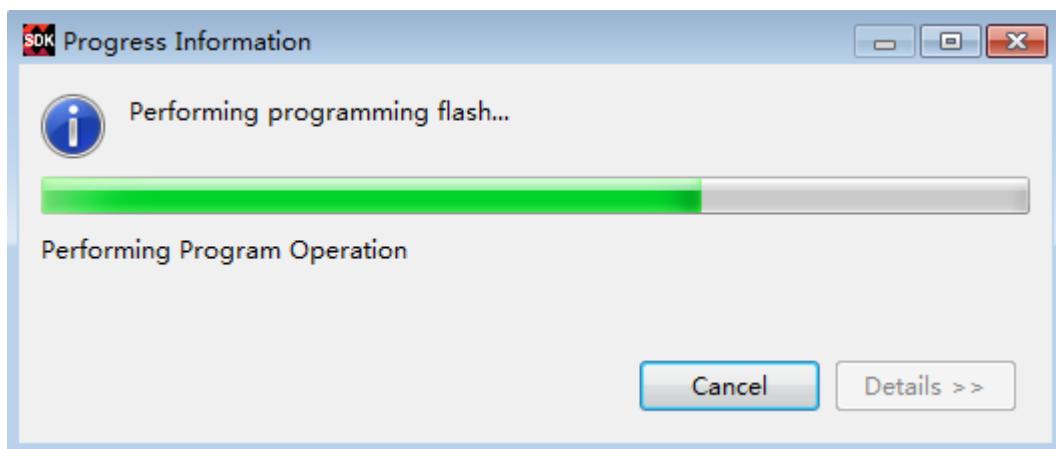
启动模式	开关状态
SD 卡启动/JTAG 调试模式	开关 1-OFF 开关 2-OFF
启动/JTAG 调试模式	开关 1-ON 开关 2-OFF

Step2: 给开发板通电，同时连接串口到 PC（不是必须的可以不连接）

Step3: 选择 Xilinx Tools > Program Flash



Step4: 下载过程，需要几分钟时间



Step5: 下载过程，输出的情况

```
CortexA9 Processor Configuration
-----
Version.....0x00000003
User ID.....0x00000000
No of PC Breakpoints.....6
No of Addr/Data Watchpoints.....4
Processor Reset .... DONE
SF: Detected S25FL256S_64K with page size 256 Bytes, erase size 64 Ki
Performing Erase Operation...
Erase Operation successful.
INFO: [Xicom 50-44] Elapsed time = 10 sec.
Performing Program Operation...
0%.....Program Operation successful.
INFO: [Xicom 50-44] Elapsed time = 168 sec.

Flash Operation Successful
```

Step6: 下载完成后断电重启，就能看到从 QSPI FLASH 加载了

## 5.8 本章小结

本章详细讲解了 SD 卡下 UBOOT 的制作过程和如何编程 QSPI FLASH。这样固化后程序就不容易丢失了。

## CH06\_XADC 实验

### 6.1 实验概述

这次借助 zynq 的内嵌的 XADC 来采集 zynq 内部的一些参数：

- VCCINT: 内部PL核心电压
- VCCAUX: 辅助PL电压
- VREFP: XADC正参考电压
- VREFN: XADC负参考电压
- VCCBram: PL BRAM电压
- VCCPInt: PS内部核心电压
- VCCPAux: PS辅助电压
- VCCDdr: DDR RAM的工作电压

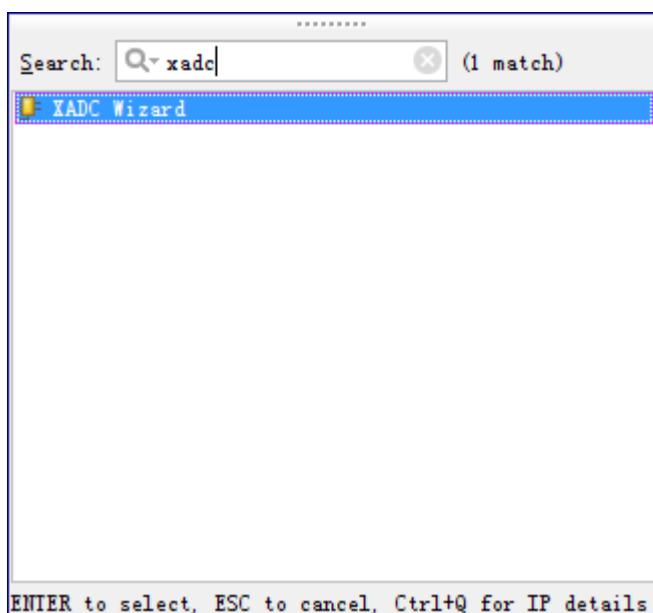
### 6.2 新建一个 VIVADO 工程

Step1:新建一个名为为 Miz\_sys 的工程，芯片类型根据自身情况设置。

Step2:创建一个 BD 文件，并命名为 system。

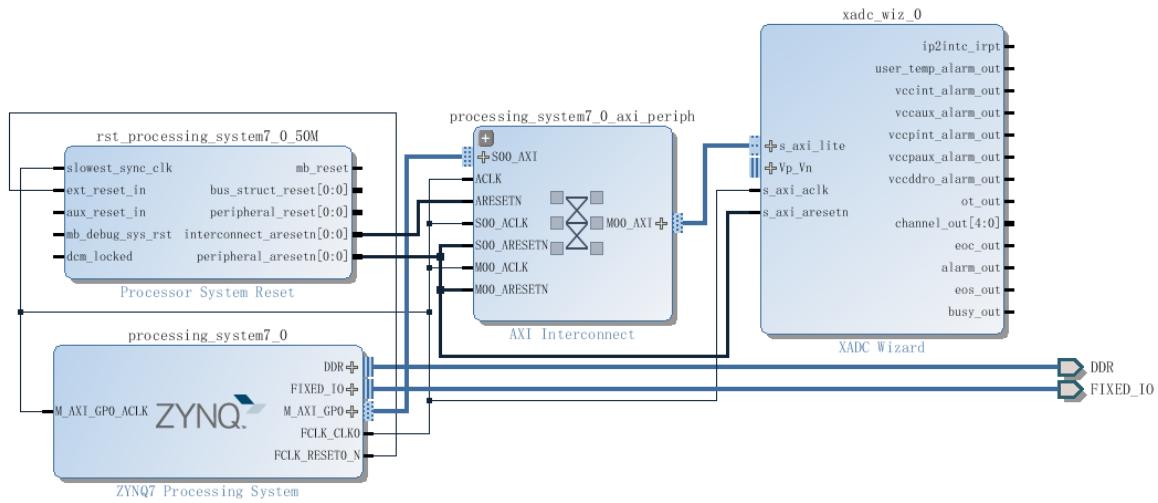
Step3:添加 ZYNQ7 Processing System，根据自己的硬件类型配置好输入时钟频率与内存型号。

Step4：单击添加 IP 按钮，输入 Xadc 添加一个 XADC 的 IP 进入 BD 文件中。



Step5：无需对 XADC IP 进行任何配置，直接单击 run connection automation，然后按

OK 完成整体电路的设计。



Step6: 右键单击 Block 文件，文件选择 Generate the Output Products。

Step7: 右键单击 Block 文件，选择 Create a HDL wrapper，根据 Block 文件内容产生一个 HDL 的顶层文件，并选择让 vivado 自动完成。

Step8: 生成 Bit 文件。

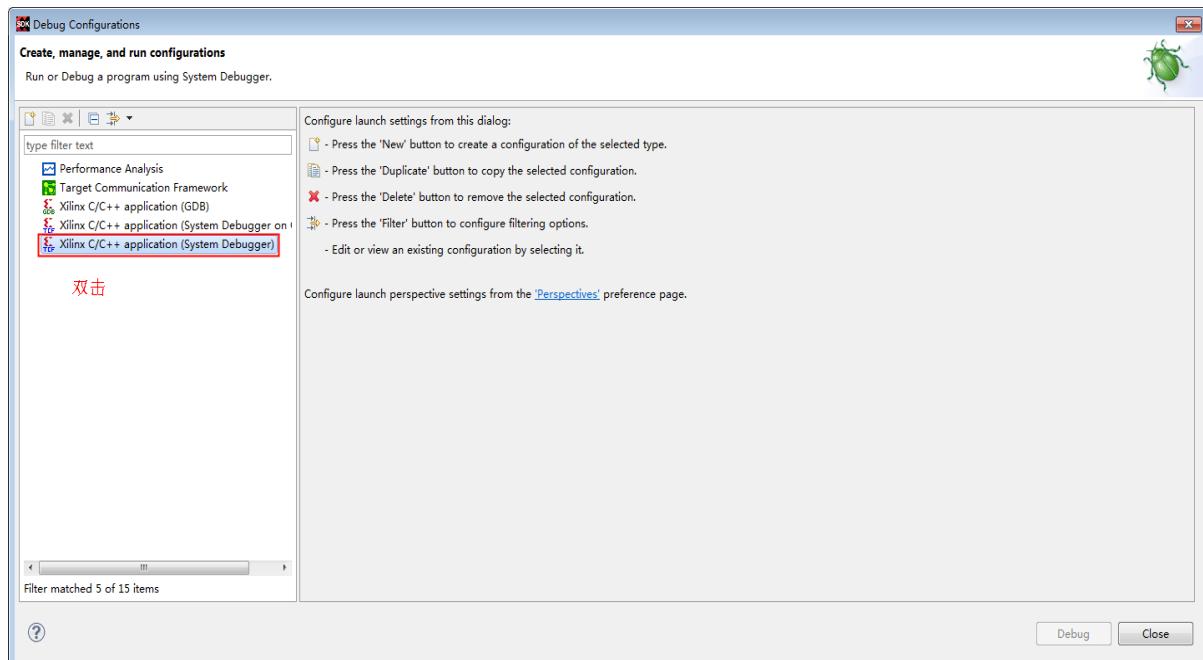
## 6.3 加载到 SDK

Step1: 导出硬件。

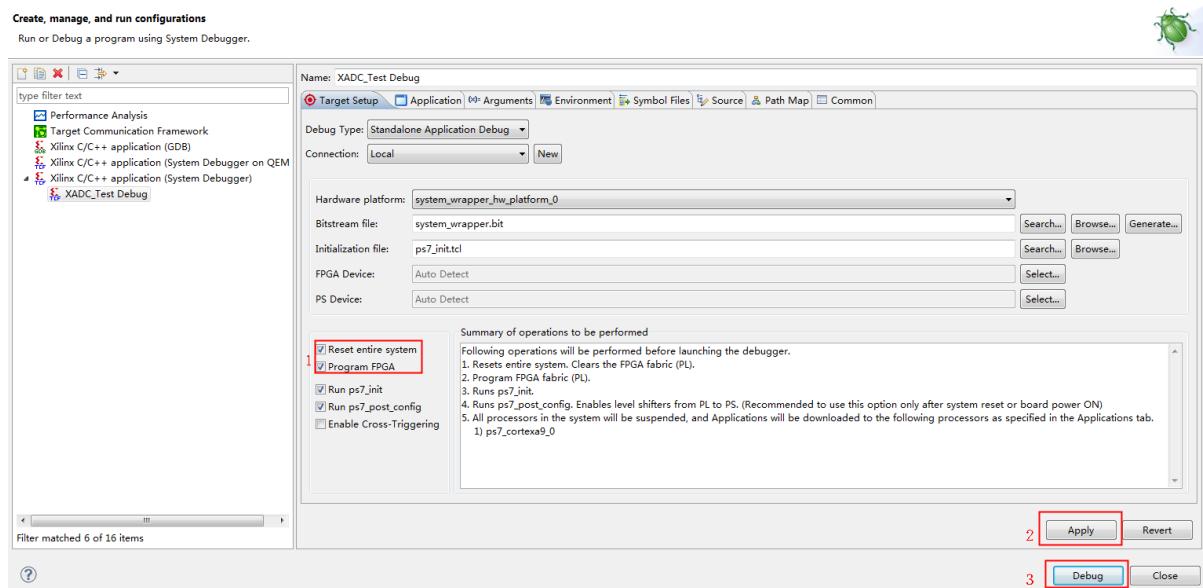
Step2: 在我们提供的源程序文件夹中找到 DOC 文件夹下的 C\_Driver 文件夹，将其中的设计文件复制一下。

Step3: 右击工程，选择 Debug as ->Debug configuration。

Step4: 选中 system Debugger, 双击创建一个系统调试。



### Step5：设置系统调试。



打开系统自带的窗口调试助手，点击运行按钮开始运行程序。



系统运行结果如下图所示：

```
Raw VccInt 21784 Real VccInt 0.997192
Raw VccAux 39416 Real VccAux 1.804321
Raw VccBram 21768 Real VccBram 0.996460
Raw VccPInt 21681 Real VccPInt 0.992477
Raw VccPAux 39394 Real VccPAux 1.803314
Raw VccDDR 32265 Real VccDDR 1.476974
Raw Temp 40884 Real Temp 41.251343
Raw VccInt 21784 Real VccInt 0.997192
Raw VccAux 39416 Real VccAux 1.804321
Raw VccBram 21768 Real VccBram 0.996460
Raw VccPInt 21681 Real VccPInt 0.992477
Raw VccPAux 39394 Real VccPAux 1.803314
Raw VccDDR 32265 Real VccDDR 1.476974
Raw Temp 40872 Real Temp 41.159058
Raw VccInt 21784 Real VccInt 0.997192
Raw VccAux 39416 Real VccAux 1.804321
Raw VccBram 21768 Real VccBram 0.996460
Raw Vcc
```

## 6.4 函数介绍

1. Use the "XAdcPs\_SelfTest()" 这个自检就不用说了
2. Use "XAdcPs\_SetSequencerMode()"这个是设置采样模式。
3. Use "XAdcPs\_SetAlarmEnables()"这个是设置采样值报警的，直接关闭，不需要报警
4. Use "XAdcPs\_SetSeqInputModule()" 这个是设置输入模式的
5. Use "XAdcPs\_SetSeqChEnables()" 这个是使能采样通道的

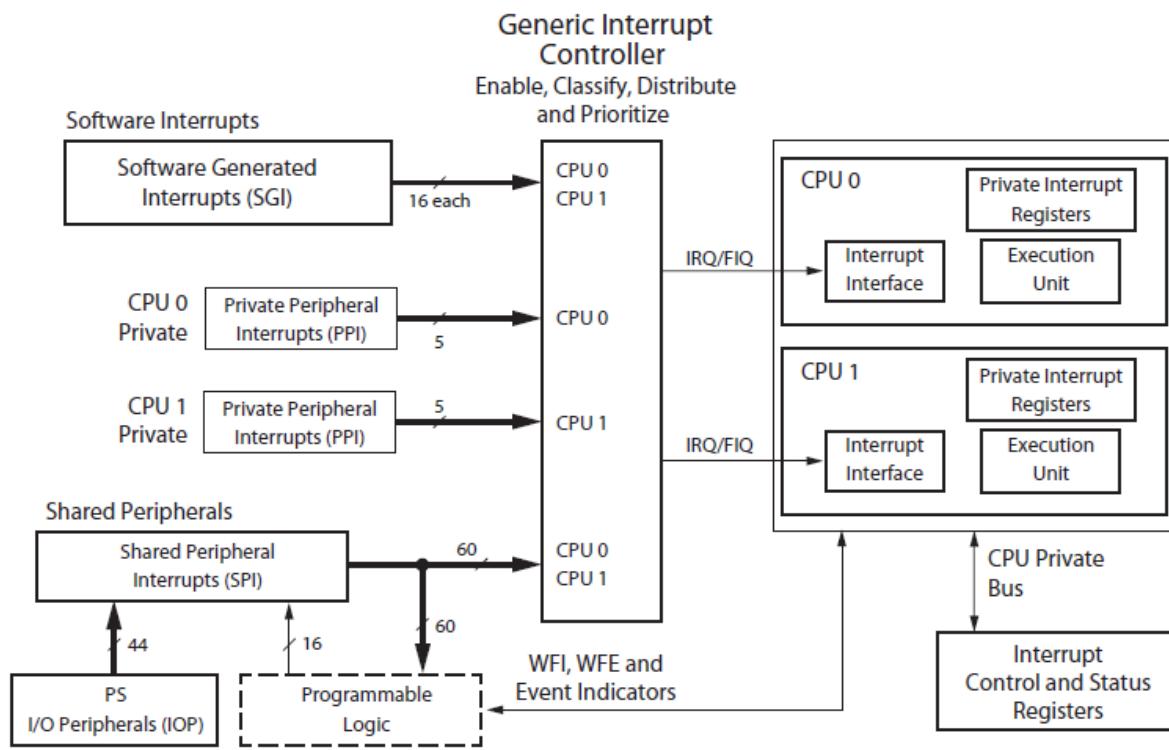
## 6.5 本章小结

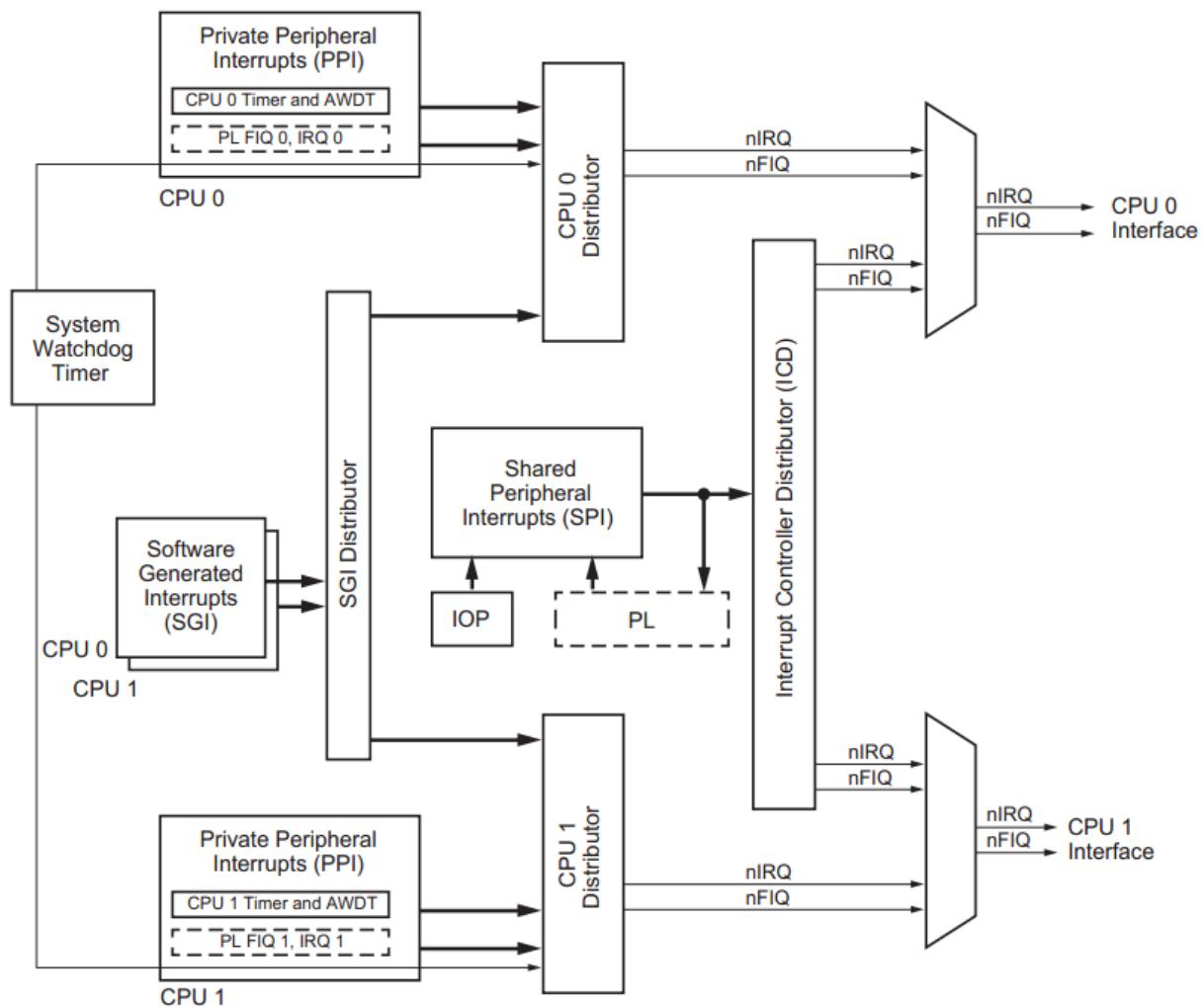
本章讲解了如果采集片上电压以及温度的方法，这个实验在实际工程运用中可以通过测试自生的电压和稳定判断系统是否可以正常工作，是否要做出一些报警之类的行动。

## CH07\_ZYNQ PL 中断请求

### 7.1 ZYNQ 中断介绍

#### 7.1.1 ZYNQ 中断框图





可以看到本例子中 PL 到 PS 部分的中断经过 ICD 控制器分发器后同时进入 CPU1 和 CPU0。从下面的表格中可以看到中断向量的具体值。PL 到 PS 部分一共有 20 个中断可以使用。其中 4 个是快速中断。剩余的 16 个是本章中涉及了，可以任意定义。如下表所示。

Type	PL Signal Name	I/O	Destination
PL to PS Interrupts	IRQF2P[7:0]	I	SPI: Numbers [68:61].
	IRQF2P[15:8]	I	SPI: Numbers [91:84].
	IRQF2P[19:16]	I	PPI: nFIQ, nIRQ (both CPUs).
PS to PL Interrupts	IRQP2F[27:0]	O	PI Logic. These signals are received from the I/O peripherals and are forwarded to the interrupt controller. These signals are also provided as outputs to the PL.

### 7.1.2 ZYNQ CPU 软件中断 (SGI)

ZYNQ 2 个 CPU 都具备各自 16 个软件中断。

IRQ ID#	Name	SGI#	Type	Description
0	Software 0	0	Rising edge	A set of 16 interrupt sources that are private to each CPU that can be routed to up to 16 common interrupt destinations where each destination can be one or more CPUs.
1	Software 1	1	Rising edge	
~	...	~	...	
15	Software 15	15	Rising edge	

### 7.1.3 ZYNQ CPU 私有端口中斷

这些中断都是固定死的，不能修改。这里有 2 个 PL 到 CPU 的快速中断 nFIQ

IRQ ID#	Name	PPI#	Type	Description
26:16	Reserved	~	~	Reserved
27	Global Timer	0	Rising edge	Global timer
28	nFIQ	1	Active Low level (active High at PS-PL interface)	Fast interrupt signal from the PL: CPU0: IRQF2P[18] CPU1: IRQF2P[19]
29	CPU Private Timer	2	Rising edge	Interrupt from private CPU timer
30	AWDT{0, 1}	3	Rising edge	Private watchdog timer for each CPU
31	nIRQ	4	Active Low level (active High at PS-PL interface)	Interrupt signal from the PL: CPU0: IRQF2P[16] CPU1: IRQF2P[17]

## 7.1.4 ZYNQ PS 和 PL 共享中断

Source	Interrupt Name	IRQ ID#	Status Bits (mpcore Registers)	Required Type	PS-PL Signal Name	I/O
APU	CPU 1, 0 (L2, TLB, BTAC)	33:32	spi_status_0[1:0]	Rising edge	~	~
	L2 Cache	34	spi_status_0[2]	High level	~	~
	OCM	35	spi_status_0[3]	High level	~	~
Reserved	~	36	spi_status_0[3]	~	~	~
PMU	PMU [1,0]	38, 37	spi_status_0[6:5]	High level	~	~
XADC	XADC	39	spi_status_0[7]	High level	~	~
DevC	DevC	40	spi_status_0[8]	High level	~	~
SWDT	SWDT	41	spi_status_0[9]	Rising edge	~	~
Timer	TTC 0	44:42	spi_status_0[12:10]	High level	~	~
DMAC	DMAC Abort	45	spi_status_0[13]	High level	IRQP2F[28]	Output
	DMAC [3:0]	49:46	spi_status_0[17:14]	High level	IRQP2F[23:20]	Output
Memory	SMC	50	spi_status_0[18]	High level	IRQP2F[19]	Output
	Quad SPI	51	spi_status_0[19]	High level	IRQP2F[18]	Output
Reserved	~	~	~	Always driven Low	IRQP2F[17]	Output
IOP	GPIO	52	spi_status_0[20]	High level	IRQP2F[16]	Output
	USB 0	53	spi_status_0[21]	High level	IRQP2F[15]	Output
	Ethernet 0	54	spi_status_0[22]	High level	IRQP2F[14]	Output
	Ethernet 0 Wake-up	55	spi_status_0[23]	Rising edge	IRQP2F[13]	Output
	SDIO 0	56	spi_status_0[24]	High level	IRQP2F[12]	Output
	I2C 0	57	spi_status_0[25]	High level	IRQP2F[11]	Output
	SPI 0	58	spi_status_0[26]	High level	IRQP2F[10]	Output
	UART 0	59	spi_status_0[27]	High level	IRQP2F[9]	Output
	CAN 0	60	spi_status_0[28]	High level	IRQP2F[8]	Output
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]	~	~	~

共享中断就是 PL 的中断可以发送给 PS 处理。上图中，黄色区域就是 16 个 PL 的中断，它们可以设置为高电平或者低电平触发。

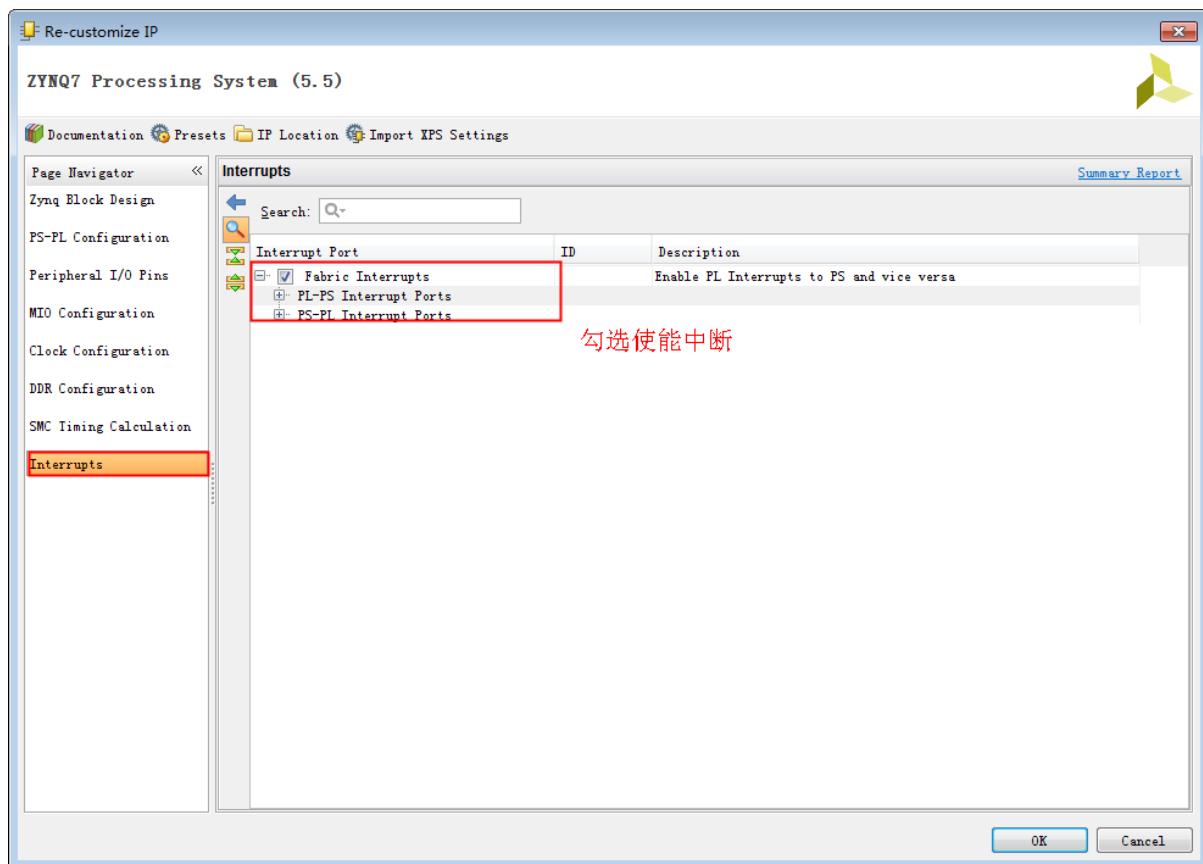
## 7.2 搭建硬件地址

Step1:新建一个名为为 Miz\_sys 的工程，芯片类型根据自身情况设置。

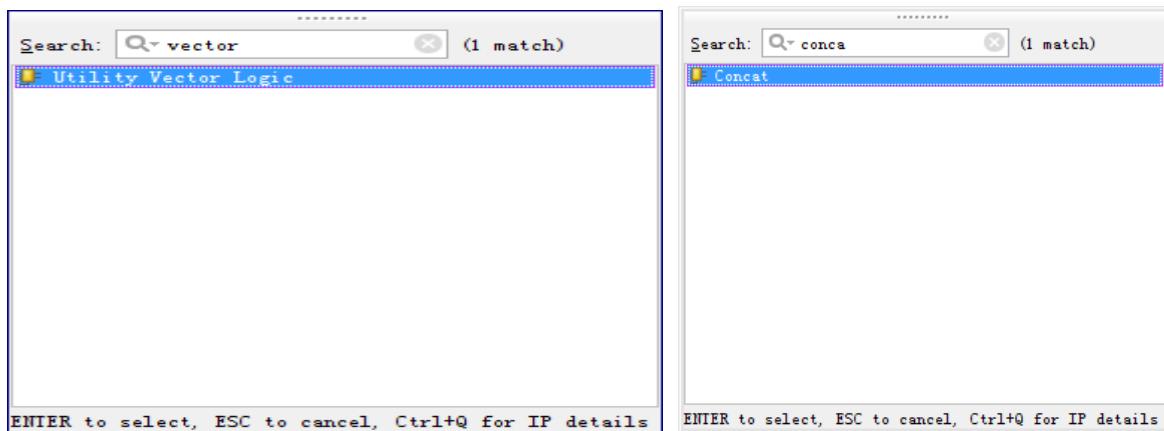
Step2:创建一个 BD 文件，并命名为 system。

Step3:添加 ZYNQ7 Processing System，根据自己的硬件类型配置好输入时钟频率与内存型号。

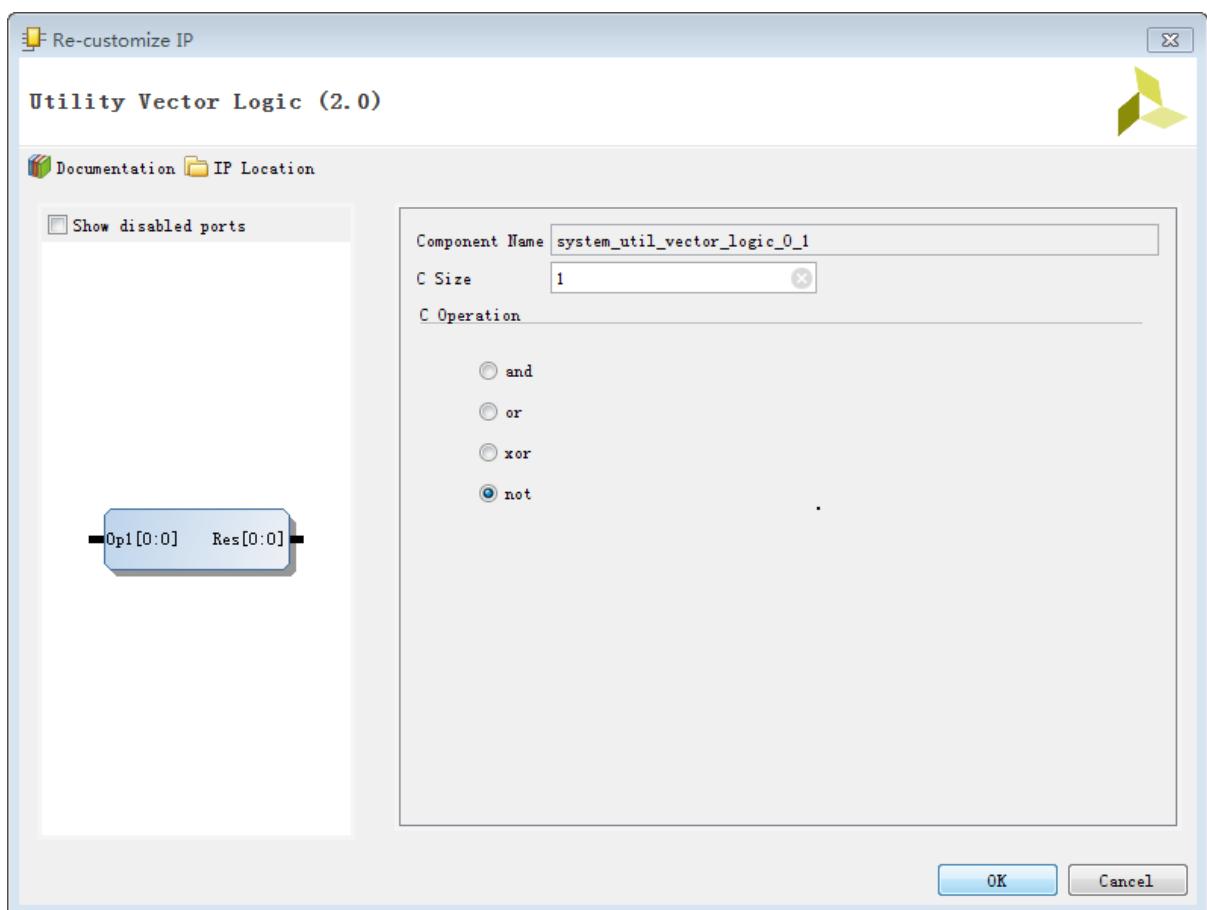
Step4: 在 ZYNQ7 Processing System 配置窗口中，使能中断，单击 OK 完成配置。



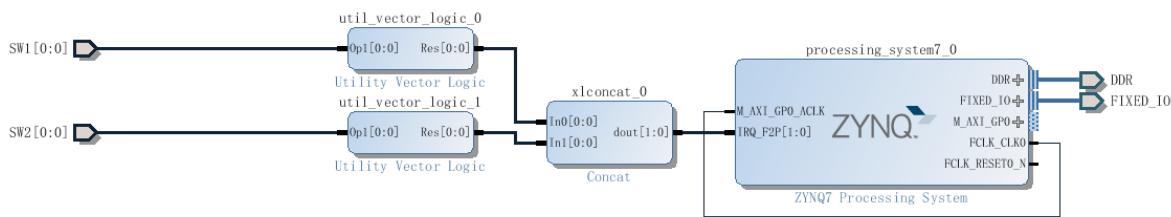
Step5:单击添加 IP 按钮，添加两个逻辑门模块和一个 concat IP。



Step6: 双击逻辑门模块，将其配置为非功能。



Step7: 按以下电路，完善整体电路。



Step8: 右键单击 Block 文件，文件选择 Generate the Output Products。

Step9: 右键单击 Block 文件，选择 Create a HDL wrapper，根据 Block 文件内容产生一个 HDL 的顶层文件，并选择让 vivado 自动完成。

Step10: 添加约束文件，在我们提供的源程序包的 DOC 文件夹下找到 XDC 文件夹，将其中的约束文件添加到工程当中来。

Step10: 生成 Bit 文件。

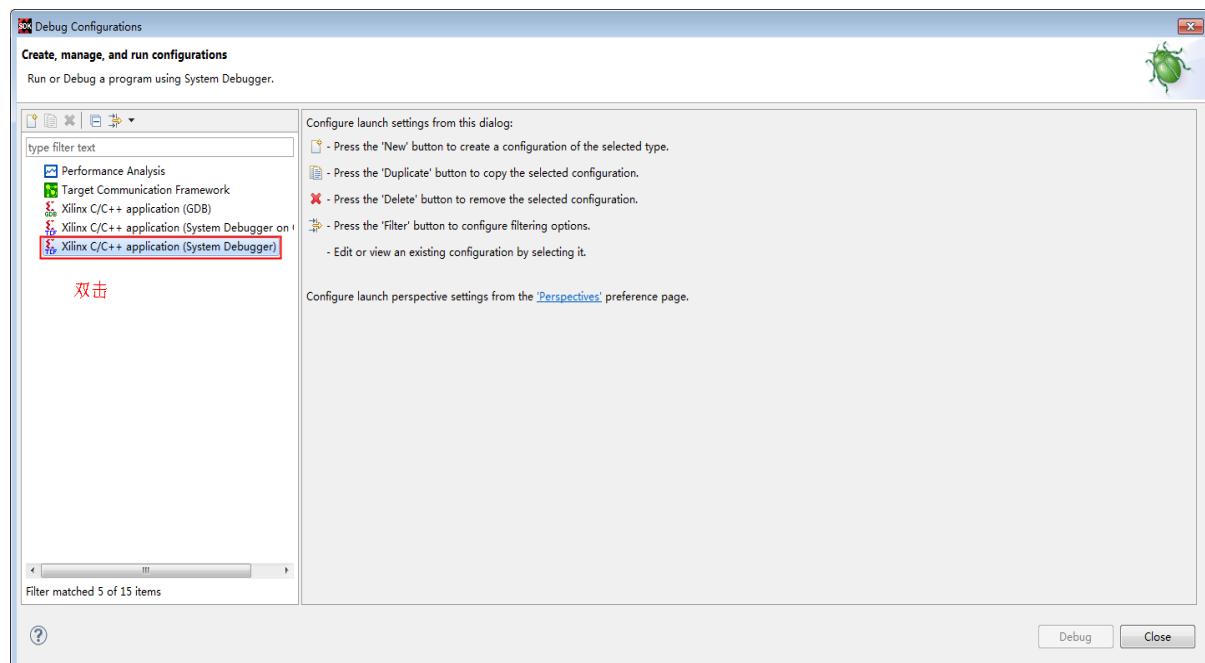
## 7.3 加载到 SDK

Step1: 导出硬件。

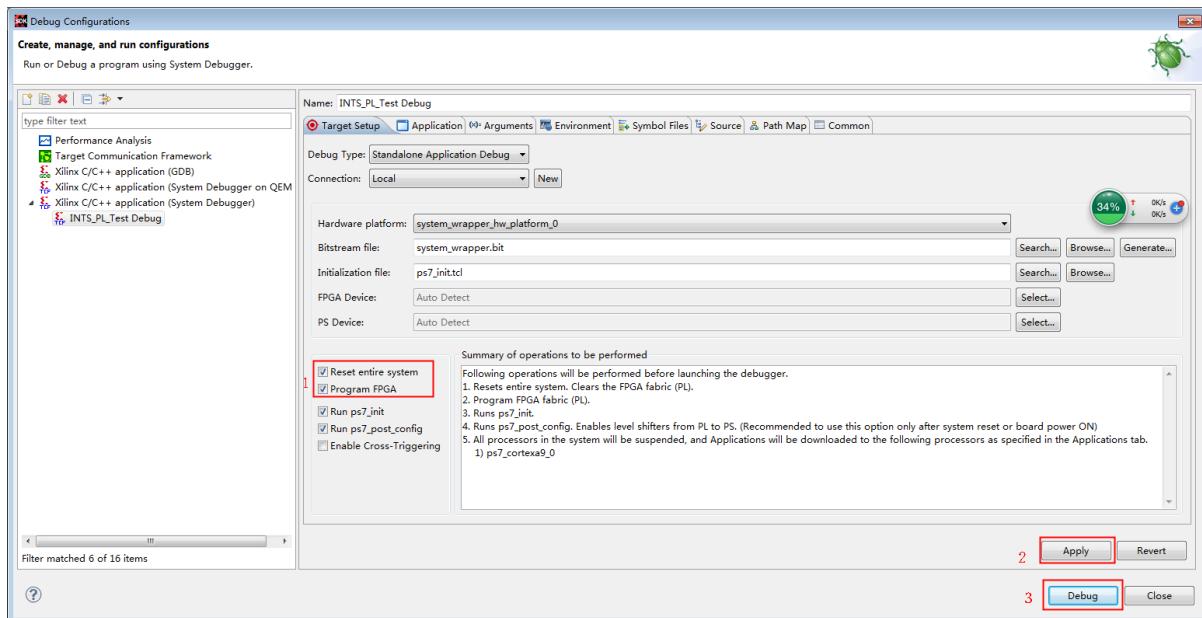
Step2: 新建一个空 SDK 工程，并将我们提供的设计文件复制到工程当中来（不熟悉的参考本季第 2、3 章 SDK 部分）。

Step3: 右击工程，选择 Debug as ->Debug configuration。

Step4: 选中 system Debugger, 双击创建一个系统调试。



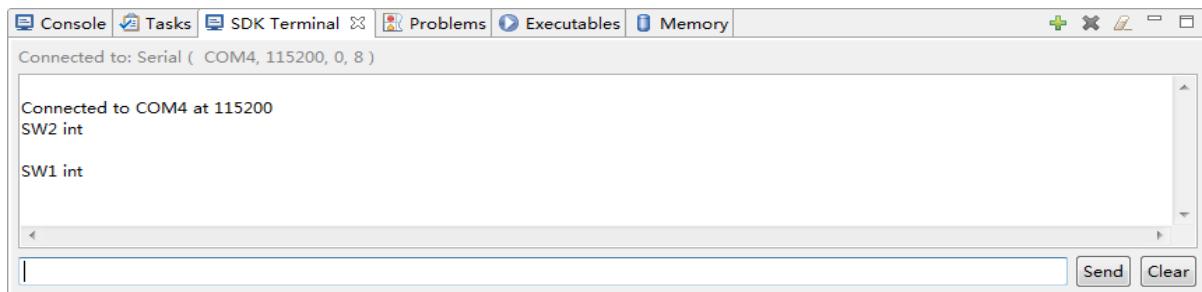
Step5: 设置系统调试。



打开系统自带的窗口调试助手，点击运行按钮开始运行程序。



系统运行结果如下图所示：



## 7.4 程序分析

接下来，我们对本章节的程序做一个详细的分析。还是先从 `main` 函数开始分析。第一句打印标题我们略过，直接看到这一句 `IntcInitFunction(INTC_DEVICE_ID);`，这个函数只带了一个参数，我们选中这个参数，直接按 F3 跟踪一下这个参数。

```
#define INTC_DEVICE_ID          XPAR_PS7_SCUGIC_0_DEVICE_ID
```

从上图可以看到，这个参数是系统的中断的设备 ID 基地址的宏定义，也就是中断的地址。

我们返回 `main` 函数当中，选中这个函数，按 F3 对其跟踪，查看一下此函数的定义。

```

int IntcInitFunction(u16 DeviceId)
{
    XScuGic_Config *IntcConfig;
    int status;

    // Interrupt controller initialisation
    IntcConfig = XScuGic_LookupConfig(DeviceId);
    status = XScuGic_CfgInitialize(&INTCInst, IntcConfig, IntcConfig->CpuBaseAddress);
    if(status != XST_SUCCESS) return XST_FAILURE;

    // Call to interrupt setup
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                                &INTCInst);

    Xil_ExceptionEnable();

    // Connect SW1~SW3 interrupt to handler
    status = XScuGic_Connect(&INTCInst,
                           SW1_INT_ID,
                           (Xil_ExceptionHandler)SW_intr_Handler,
                           (void *)1);
    if(status != XST_SUCCESS) return XST_FAILURE;

    status = XScuGic_Connect(&INTCInst,
                           SW2_INT_ID,
                           (Xil_ExceptionHandler)SW_intr_Handler,
                           (void *)2);
    if(status != XST_SUCCESS) return XST_FAILURE;

    // Set interrupt type of SW1~SW3 to rising edge
    IntcTypeSetup(&INTCInst, SW1_INT_ID, INT_TYPE_RISING_EDGE);
    IntcTypeSetup(&INTCInst, SW2_INT_ID, INT_TYPE_RISING_EDGE);

    // Enable SW1~SW3 interrupts in the controller
    XScuGic_Enable(&INTCInst, SW1_INT_ID);
    XScuGic_Enable(&INTCInst, SW2_INT_ID);

    return XST_SUCCESS;
}

```

程序一开头还是定义了一些要用到的指针和变量。接下来是一个跟第二章讲过的相似的一个查找设备配置的程序，带的参数为设备 ID,也就是看我们的中断向量是否存在，感兴趣的可以选中这个程序，按下 F3 查看其定义。

```

XScuGic_Config *XScuGic_LookupConfig(u16 DeviceId)
{
    XScuGic_Config *CfgPtr = NULL;
    u32 Index;

    for (Index=0U; Index < (u32)XPAR_SCUGIC_NUM_INSTANCES; Index++) {
        if (XScuGic_ConfigTable[Index].DeviceId == DeviceId) {
            CfgPtr = &XScuGic_ConfigTable[Index];
            break;
        }
    }

    return (XScuGic_Config *)CfgPtr;
}

```

接下来依然是一个状态检测，这是 xilinx 初始化的老套路，当执行完这一句后，系统会对我们的中断做一些初始化，如果初始化成功，会返回一个 XST\_SUCCESS 的标志。当未检测到返回到这

个初始化成功的标志时，系统会返回一个 XST\_FAILURE 标志。

接下来是一个中断注册函数 Xil\_ExceptionRegisterHandler,按照之前讲过的方法，查看其函数定义。

```
*****
/** 
 * Makes the connection between the Id of the exception source and the
 * associated Handler that is to run when the exception is recognized. The
 * argument provided in this call as the Data is used as the argument
 * for the Handler when it is called.
 *
 * @param    exception_id contains the ID of the exception source and should
 *           be in the range of 0 to XIL_EXCEPTION_ID_LAST.
 *           See xil_exception_l.h for further information.
 * @param    Handler to the Handler for that exception.
 * @param    Data is a reference to Data that will be passed to the
 *           Handler when it gets called.
 *
 * @return   None.
 *
 * @note    None.
 */
void Xil_ExceptionRegisterHandler(u32 Exception_id,
                                  Xil_ExceptionHandler Handler,
                                  void *Data)
{
    XExc_VectorTable[Exception_id].Handler = Handler;
    XExc_VectorTable[Exception_id].Data = Data;
}
```

从上面可以看到这个函数是把中断的句柄和中断的参数放到了两个数组当中，选中这个数组按下 F3 来看看这个数组。

```
XExc_VectorTableEntry XExc_VectorTable[XIL_EXCEPTION_ID_LAST + 1] =
{
    {Xil_ExceptionNullHandler, NULL},
    {Xil_ExceptionNullHandler, NULL},
    {Xil_ExceptionNullHandler, NULL},
    {Xil_PrefetchAbortHandler, NULL},
    {Xil_DataAbortHandler, NULL},
    {Xil_ExceptionNullHandler, NULL},
    {Xil_ExceptionNullHandler, NULL},
};
```

可以看到这个数组的结构如上图所示，它是由一个结构体定义的，这个结构体定义如下图所示：

```
typedef struct {
    Xil_ExceptionHandler Handler;
    void *Data;
} XExc_VectorTableEntry;
```

接下来看到这段程序：

```
status = XScuGic_Connect(&INTCInst,
                         SW1_INT_ID,
                         (Xil_ExceptionHandler)SW_intr_Handler,
                         (void *)1);
if(status != XST_SUCCESS) return XST_FAILURE;

status = XScuGic_Connect(&INTCInst,
                         SW2_INT_ID,
                         (Xil_ExceptionHandler)SW_intr_Handler,
                         (void *)2);
if(status != XST_SUCCESS) return XST_FAILURE;
```

通过上图中的程序，可以连接到我们的中断，我们查看下其定义。

```

/****************************************************************************
 * Makes the connection between the Int_Id of the interrupt source and the
 * associated handler that is to run when the interrupt is recognized. The
 * argument provided in this call as the CallBackRef is used as the argument
 * for the handler when it is called.
 *
 * @param InstancePtr is a pointer to the XScuGic instance.
 * @param Int_Id contains the ID of the interrupt source and should be
 *           in the range of 0 to XSCUGIC_MAX_NUM_INTR_INPUTS - 1
 * @param Handler to the handler for that interrupt.
 * @param CallBackRef is the callback reference, usually the instance
 *           pointer of the connecting driver.
 *
 * @return
 *           - XST_SUCCESS if the handler was connected correctly.
 *
 * @note
 *
 * WARNING: The handler provided as an argument will overwrite any handler
 * that was previously connected.
 */
s32 XScuGic_Connect(XScuGic *InstancePtr, u32 Int_Id,
                     Xil_InterruptHandler Handler, void *CallBackRef)
{
    /*
     * Assert the arguments
     */
    Xil_AssertNonvoid(InstancePtr != NULL);
    Xil_AssertNonvoid(Int_Id < XSCUGIC_MAX_NUM_INTR_INPUTS);
    Xil_AssertNonvoid(Handler != NULL);
    Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

    /*
     * The Int_Id is used as an index into the table to select the proper
     * handler
     */
    InstancePtr->Config->HandlerTable[Int_Id].Handler = Handler;
    InstancePtr->Config->HandlerTable[Int_Id].CallBackRef = CallBackRef;

    return XST_SUCCESS;
}

```

上图可以看到方框中的语句把我们的中断的句柄和一个指针变量传递了进来，也就是把 `XScuGic_Connect` 函数的最后两个函数传递了进来。此时我们返回继续查看 `XScuGic_Connect` 函数，我们发现中断的句柄其实是个指针函数，也就是说当程序被执行的时候，其实被调用的是这个指针函数，此时我们跟踪这个指针函数，查看它具体做了些什么。

```

/*
 * This function is the primary interrupt handler for the driver. It must be
 * connected to the interrupt source such that it is called when an interrupt of
 * the interrupt controller is active. It will resolve which interrupts are
 * active and enabled and call the appropriate interrupt handler. It uses
 * the Interrupt Type information to determine when to acknowledge the interrupt.
 * Highest priority interrupts are serviced first.
 *
 * This function assumes that an interrupt vector table has been previously
 * initialized. It does not verify that entries in the table are valid before
 * calling an interrupt handler.
 *
 * @param InstancePtr is a pointer to the XScuGic instance.
 * @return None.
 * @note None.
 */
void XScuGic_InterruptHandler(XScuGic *InstancePtr)
{
    u32 InterruptID;
    u32 IntIDFull;
    XScuGic_VectorTableEntry *TablePtr;

    /* Assert that the pointer to the instance is valid
     */
    Xil_AssertVoid(InstancePtr != NULL);

    /*
     * Read the int_ack register to identify the highest priority interrupt ID
     * and make sure it is valid. Reading Int_Ack will clear the interrupt
     * in the GIC.
     */
    IntIDFull = XScuGic_CPUReadReg(InstancePtr, XSCUGIC_INT_ACK_OFFSET);
    InterruptID = IntIDFull & XSCUGIC_ACK_INTID_MASK;

    if(XSCUGIC_MAX_NUM_INTR_INPUTS < InterruptID){
        goto IntrExit;
    }

    /*
     * If the interrupt is shared, do some locking here if there are multiple
     * processors.
     */
    /*
     * If pre-emption is required:
     * Re-enable pre-emption by setting the CPSR I bit for non-secure ,
     * interrupts or the F bit for secure interrupts
     */

    /*
     * If we need to change security domains, issue a SMC instruction here.
     */

    /*
     * Execute the ISR. Jump into the Interrupt service routine based on the
     * IRQSource. A software trigger is cleared by the ACK.
     */
    TablePtr = &(InstancePtr->Config->HandlerTable[InterruptID]);
    if(TablePtr != NULL) {
        TablePtr->Handler(TablePtr->CallBackRef);
    }

    IntrExit:
    /*
     * Write to the EOI register, we are all done here.
     * Let this function return, the boot code will restore the stack.
     */
    XScuGic_CPUWriteReg(InstancePtr, XSCUGIC_EOI_OFFSET, IntIDFull);

    /*
     * Return from the interrupt. Change security domains could happen here.
     */
}

```

通过程序开头 xilinx 给出的这个程序的注释可以知道：这个函数是基本的中断驱动函数。它必须连接到中断源，以便在中断控制器的中断激活时被调用。它将解决哪些中断是活动的和启用的，并调用适当的中断处理程序。它使用中断类型信息来确定何时确认中断。首先处理最高优先级的中断。此函数假定中断向量表已预先初始化。它不会在调用中断处理程序之前验证表中的条目是否有效。

上面讲到的这个中断向量表其实也就是下图所示的部分。

```

    void Xil_ExceptionRegisterHandler(u32 Exception_id,
                                     Xil_ExceptionHandler Handler,
                                     void *Data)
    {
        XExc_VectorTable[Exception_id].Handler = Handler;
        XExc_VectorTable[Exception_id].Data = Data;
    }
}

```

这部分在刚才已经进行了讲解了，此时我们就可以清楚的知道这就是一个中断向量表了。

回到基本的中断驱动函数的分析，看到下面的一段程序：

```

/*
 * Read the int_ack register to identify the highest priority interrupt ID
 * and make sure it is valid. Reading Int_Ack will clear the interrupt
 * in the GIC.
 */
IntIDFull = XScuGic_CPUReadReg(InstancePtr, XSCUGIC_INT_ACK_OFFSET);
InterruptID = IntIDFull & XSCUGIC_ACK_INTID_MASK;

if(XSCUGIC_MAX_NUM_INTR_INPUTS < InterruptID){
    goto IntrExit;
}

```

通过注释我们知道了这个程序是读取 `int_ack` 寄存器以识别最高优先级的中断 ID，并确保其有效。读取 `Int_Ack` 将清除 GIC 中的中断。然后看看读出来的中断 ID 是否大于最大的中断值。查看下这个最大的中断值。

```

#ifndef __ARM_NEON__
#define XSCUGIC_MAX_NUM_INTR_INPUTS 95U /* Maximum number of interrupt defined by Zynq */
#else
#define XSCUGIC_MAX_NUM_INTR_INPUTS 195U /* Maximum number of interrupt defined by Zynq Ultrascale MP */
#endif

```

从上图中圈出的地方可以看到，当使用 ZYNQ 的时候，最大有 95 个中断可以供我们使用。当读出来的这个中断值大于 95U 的话，就直接跳转到异常处理程序部分：

```

IntrExit:
/*
 * Write to the EOI register, we are all done here.
 * Let this function return, the boot code will restore the stack.
*/
XScuGic_CPUWriteReg(InstancePtr, XSCUGIC_EOI_OFFSET, IntIDFull);

/*
 * Return from the interrupt. Change security domains could happen here.
*/

```

这里的意思也就相当于恢复中断寄存器，相当于出栈。

当读出来的中断值是正常的话，就会查找这个中断的中断向量表，如果这个向量表不是非空的话，就开始处理这个中断，也就是开始执行之前的连接中断的函数。此部分程序如下：

```

/*
 * Execute the ISR. Jump into the Interrupt service routine based on the
 * IRQSource. A software trigger is cleared by the ACK.
*/
TablePtr = &(InstancePtr->Config->HandlerTable[InterruptID]);
if(TablePtr != NULL) {
    TablePtr->Handler(TablePtr->CallBackRef);
}

```

上图中的 `Tableptr` 指向的 `CallBackRef` 其实就是我们连接中断函数定义的无符号的数字，如下图所示。

```
// Connect SW1~SW3 interrupt to handler
status = XScuGic_Connect(&INTCInst,
                        SW1_INT_ID,
                        (Xil_ExceptionHandler)SW_intr_Handler,
                        (void *)1);
if(status != XST_SUCCESS) return XST_FAILURE;

status = XScuGic_Connect(&INTCInst,
                        SW2_INT_ID,
                        (Xil_ExceptionHandler)SW_intr_Handler,
                        (void *)2);
if(status != XST_SUCCESS) return XST_FAILURE;
```

为了验证我们的猜想，我们可以把这里的数字改成其他的值进行验证。

回到主程序当中，接着看到这段函数：

```
IntcTypeSetup(&INTCInst, SW1_INT_ID, INT_TYPE_RISING_EDGE);
IntcTypeSetup(&INTCInst, SW2_INT_ID, INT_TYPE_RISING_EDGE);
```

这段程序把中断的触发类型设置为了上升沿触发。

```
// Enable SW1~SW3 interrupts in the controller
XScuGic_Enable(&INTCInst, SW1_INT_ID);
XScuGic_Enable(&INTCInst, SW2_INT_ID);
```

这段程序使能了中断。

整段程序下来，那么主要是执行了哪个函数呢？通过上面的分析，我们可以判定其实是下面这个函数：

```
status = XScuGic_Connect(&INTCInst,
                        SW1_INT_ID,
                        (Xil_ExceptionHandler)SW_intr_Handler,
                        (void *)1);
```

这个函数的方框部分其实是个指针函数，我们可以跟踪看一下其定义。

```
static void SW_intr_Handler(void *param)
{
    int sw_id = (int)param;
    printf("Sw%d int\n\r", sw_id);
}
```

一开始，它将传递进来的指针传递给了 sw\_id，然后会打印哪个按钮初始化，其实也就是哪个中断被触发了。

接下来，我们再对中断的一些寄存器做一些分析。在中断设置里的一些寄存器是比较重要的，我们就来分析一下中断设置里的寄存器。

```

void IntcTypeSetup(XScuGic *InstancePtr, int intId, int intType)
{
    int mask;

    intType &= INT_TYPE_MASK;
    mask = XScuGic_DistReg(InstancePtr, INT_CFG0_OFFSET + (intId/16)*4);
    mask &= ~INT_TYPE_MASK << ((intId%16)*2);
    mask |= intType << ((intId%16)*2);
    XScuGic_DistWriteReg(InstancePtr, INT_CFG0_OFFSET + (intId/16)*4, mask);
}

Explore Macro Expansion - 3 step(s)
int IntcInitFunc:#define INT_CFG0_OFFSET 0x00000C00
{
    XScuGic_Config
    int status;
    XScuGic_DistReadReg(InstancePtr, INT_CFG0_OFFSET . . .
}

```

The screenshot shows a code editor with a tooltip expanded over a line of code. The tooltip title is "Explore Macro Expansion - 3 step(s)". It contains two columns: "Original" and "Fully Expanded". The "Original" column shows the macro definition "#define INT\_CFG0\_OFFSET 0x00000C00". The "Fully Expanded" column shows the expanded code: "((Xil\_In32(((InstancePtr)->Config->DistBaseAddress) + ((0x00000C00 + (intId/16)\*4))))". A red box highlights the macro definition in the original code.

将鼠标停留在图上圈出的函数上，SDK 会跳出关于这个函数的信息，在跳出的窗口中左边是我们圈出的这个函数的定义，右边则是在执行过程中实际运行的程序。我们拷贝出右边这个函数来分析一下：

((Xil\_In32(((InstancePtr)->Config->DistBaseAddress)) + ((0x00000C00 + (intId/16)\*4))))

红色部分是一个指针，它调用了 config 里的一个地址 DisBaseAddress，后半部分我们可以断定这是一个寄存器地址，因为这个函数就是一个读取中断寄存器的函数。此时，我们跟踪一下这个函数。

```

/*
 *
 * Read the given Distributor Interface register
 *
 * @param InstancePtr is a pointer to the instance to be worked on.
 * @param RegOffset is the register offset to be read
 *
 * @return The 32-bit value of the register
 *
 * @note
 * C-style signature:
 *     u32 XScuGic_DistReadReg(XScuGic *InstancePtr, u32 RegOffset)
 */
#define XScuGic_DistReadReg(InstancePtr, RegOffset) \
(XScuGic_ReadReg(((InstancePtr)->Config->DistBaseAddress), (RegOffset)))

```

此时，我们就知道了第一个参数是一个指向要处理的中断的指针，第二个是寄存器偏移。我们就来计算一下这个寄存器偏移。首先我们来看看中断的基址是多少（也就是红色部分指向的地址）。

```

/* Definitions for peripheral PS7_SCUGIC_0 */
#define XPAR_PS7_SCUGIC_0_DEVICE_ID 0
#define XPAR_PS7_SCUGIC_0_BASEADDR 0xF8F00100
#define XPAR_PS7_SCUGIC_0_HIGHADDR 0xF8F001FF
#define XPAR_PS7_SCUGIC_0_DIST_BASEADDR 0xF8F01000

```

在 xparameters.h 中，找到了中断的基地址，如图中方框部分，为 F8F01000。IntId 就是定义的哪个按钮将被初始化，此处以 SW1 为例，SW1 的 ID 为 #define SW1\_INT\_ID 61，等于 61，此时可以算出：寄存器的地址 = F8F01000 + ((0x00000C00 + (61/16)\*4)) = F8F01C0C。打开 ug585，查看一下这个寄存器是什么功能。

Relative Address	0x00001C0C
Absolute Address	0xF8F01C0C
Width	32 bits
Access Type	rw
Reset Value	0x00000000
Description	Interrupt Configuration Register 3

#### Register ICDICFR3 Details

The ICDICFR 3 register control the interrupt sensitivity of the Shared Peripheral Interrupts (SPI), IRQ ID #48 to ID #63. This register has two bits per interrupt. This two bit field is either equal to 01 (high-level active) or equal to 11 (rising-edge sensitive). The LSB is always 1 because only one CPU will handle a SPI interrupt, regardless of the number of CPUs targeted.

Refer to UG585 TRM Section 7.2.3 Shared Peripheral Interrupts (SPI) for the required sensitivity type for the SPI interrupts. The SPI interrupts must match the expected sensitivity. Interrupts from the PL may be high-level or rising edge sensitive; this must be coordinated with the PL hardware and software.

Field Name	Bits	Type	Reset Value	Description
config_63 (GIC_INT_CFG)	31:30	rw	0x0	Configuration for interrupt ID#63 01: high-level active 11: rising-edge The lower bit is read-only and is always 1.
config_62 (GIC_INT_CFG)	29:28	rw	0x0	Configuration for interrupt ID#62 01: high-level active 11: rising-edge The lower bit is read-only and is always 1.
config_61 (GIC_INT_CFG)	27:26	rw	0x0	Configuration for interrupt ID#61 01: high-level active 11: rising-edge The lower bit is read-only and is always 1.

从图中我们可以看出这是一个设置中断触发方式的寄存器，01 的时候，高电平触发，11 的时候，上升沿触发。从上表中可以看到每个中断 ID 都由两位表示，而寄存器又是 32 位数据，因此，可以算出总共我们可以设置 16 个中断 ID，这也是程序中为什么要除以 16 的原因。接下来看到 Intcsetup 的下一句。`mask &= ~(INT_TYPE_MASK << (intId%16)*2);` 当执行完这一句后，我们来计算一下寄存器地址变为了多少？在前面的定义中，我们找到 INT\_TYPE\_MASK 的值，`#define INT_TYPE_MASK 0x03`，因此可以计算出此时寄存器的值为：`F8F01C0C & (~C000000) = F0F01C0C`。下一句又是一个运算，这次我们直接计算：`F0F01C0C | 3FFFFFF = F3FF1C0C`。也就是说最终写入寄存器的值是这个值。可以对照 ug585 查看配置的信息。

其他的寄存器设置的分析方法与上面的一致，在此就不再反复讲解了。

## 7.5 本章小结

本章学习了外部中断，通过 PL 传递开发板按键的中断，然后在 PS 接受处理中断。

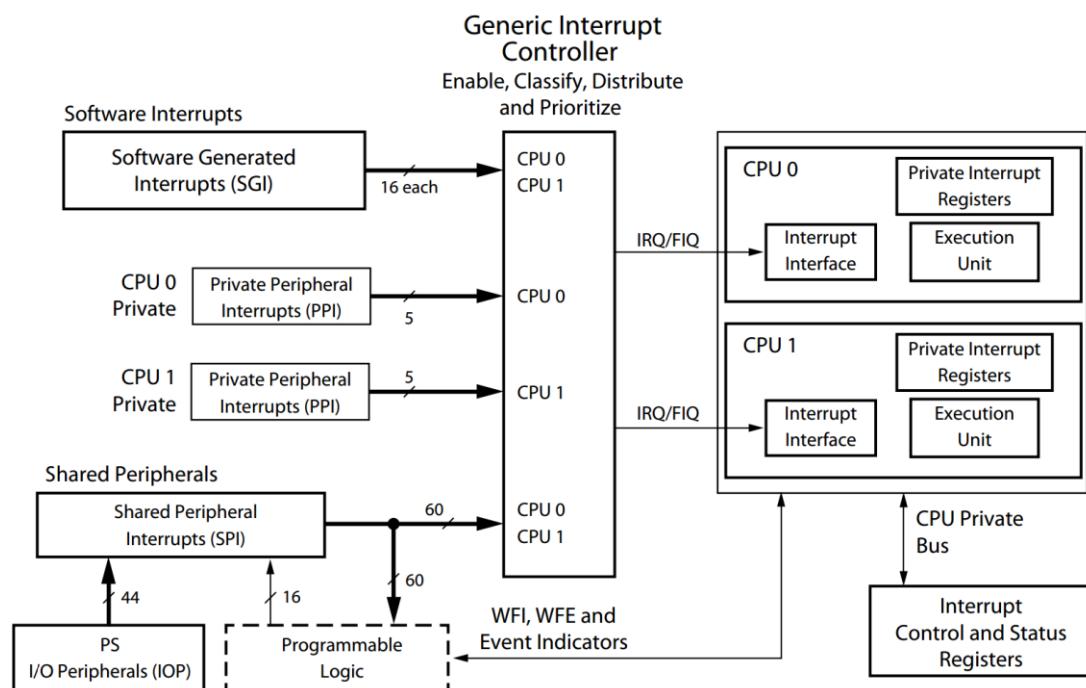
## CH08\_ZYNQ 定时器中断实验

上一章实现了 PS 接受来自 PL 的中断，本章将在 ZYNQ 的纯 PS 里实现私有定时器中断。每隔一秒中断一次，在中断函数里计数加 1，通过串口打印输出。

### 8.1 中断原理

中断对于保证任务的实时性非常必要，在 ZYNQ 里集成了中断控制器 GIC(Generic Interrupt Controller). GIC 可以接受 I/O 外设中断 IOP 和 PL 中断，将这些中断发给 CPU。

中断体系结构框图图下：



#### 8.1.1 软件中断(SGI)

SGI 通过写 ICDSGIR 寄存器产生 SGI.

#### 8.1.2 共享中断 SPI

通过 PS 和 PL 内各种 I/O 和存储器控制器产生。

### 8.1.3 私有中断（PPI）

包含：全局定时器，私有看门狗定时器，私有定时器以及来自 PL 的 FIQ/IRQ。本文主要介绍 PPI，其它的请参考官方手册 ug585\_Zynq\_7000\_TRM.pdf。

ZYNQ 每个 CPU 链接 5 个私有外设中断，所有中断的触发类型都是固定不变的。并且来自 PL 的快速中断信号 FIQ 和中断信号 IRQ 反向，然后送到中断控制器因此尽管在 ICDICFR1 寄存器内反应的他们是低电平触发，但是 PS-PL 接口中为高电平触发。如图所示：

IRQ ID#	Name	PPI#	Type	Description
26:16	Reserved	~	~	Reserved
27	Global Timer	0	Rising edge	Global timer
28	nFIQ	1	Active Low level (active High at PS-PL interface)	Fast interrupt signal from the PL: CPU0: IRQF2P[18] CPU1: IRQF2P[19]
29	CPU Private Timer	2	Rising edge	Interrupt from private CPU timer
30	AWDT{0, 1}	3	Rising edge	Private watchdog timer for each CPU
31	nIRQ	4	Active Low level (active High at PS-PL interface)	Interrupt signal from the PL: CPU0: IRQF2P[16] CPU1: IRQF2P[17]

### 8.1.4 私有定时器

zynq 中每个 ARM core 都有自己的私有定时器，私有定时器的工作频率为 CPU 的一半，比如 MZ702A 的 ARM 工作频率为 666MHZ，则私有定时器的频率为 333MHz.

私有定时器的特性如下：

- (1) 32 位计数器，达到零时产生一个中断
- (2) 8 位预分频计数器，可以更好的控制中断周期
- (3) 可配置一次性或者自动重加载模式
- (4) 定时器时间可以通过下式计算：

定时时间 = 1/定时器频率 \* (预加载值+1)

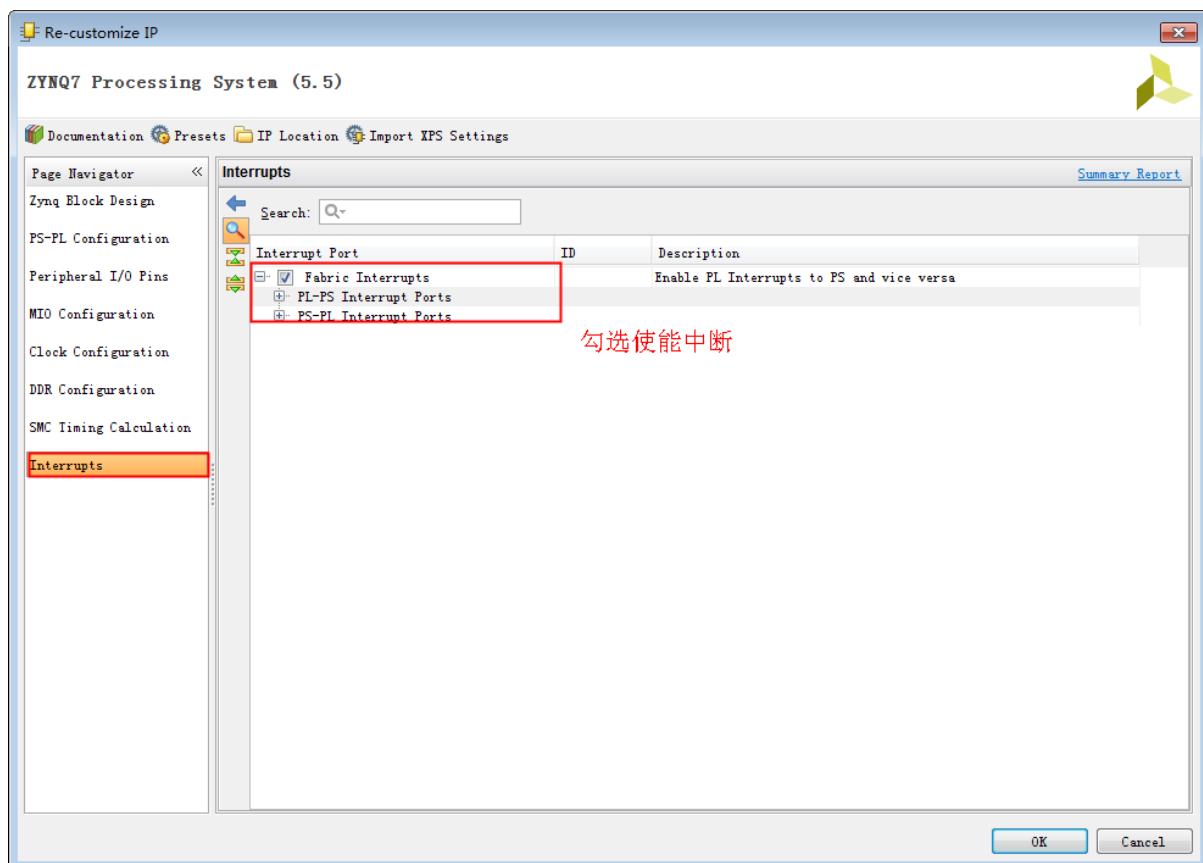
## 8.2 搭建硬件工程

Step1:新建一个名为为 Miz\_sys 的工程，芯片类型根据自身情况设置。

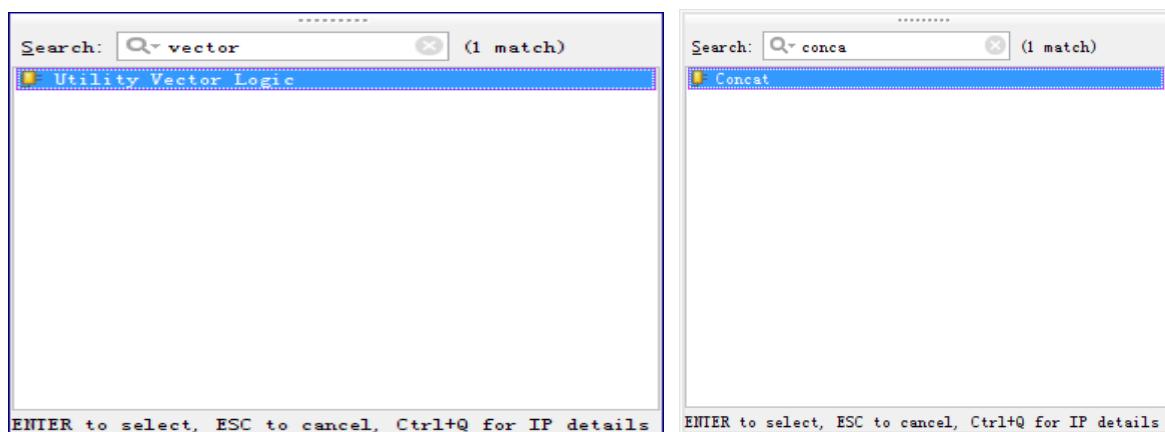
Step2:创建一个 BD 文件，并命名为 system。

Step3:添加 ZYNQ7 Processing System，根据自己的硬件类型配置好输入时钟频率与内存型号。

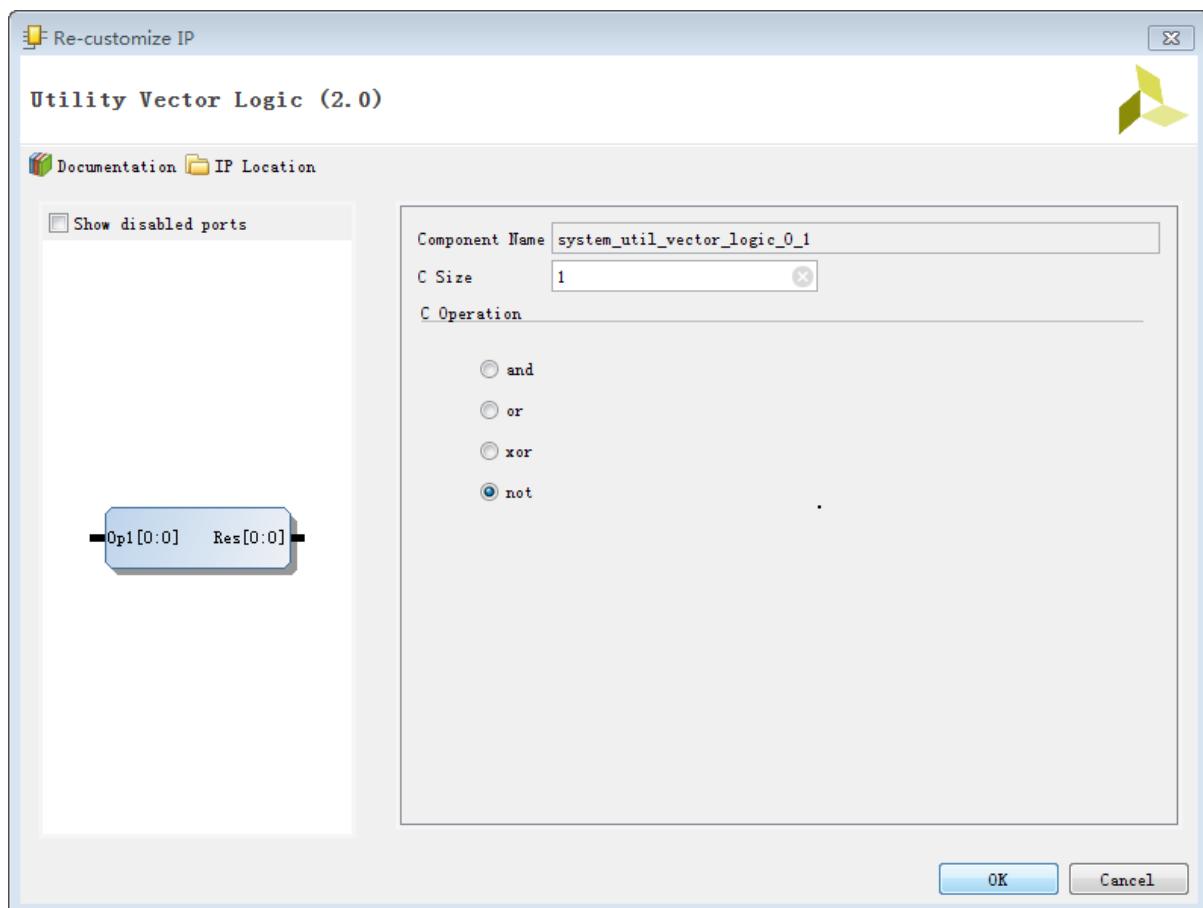
Step4：在 ZYNQ7 Processing System 配置窗口中，使能中断，单击 OK 完成配置。



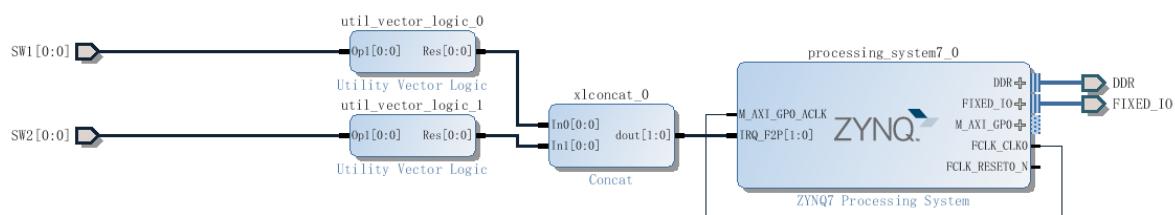
Step5:单击添加 IP 按钮，添加两个逻辑门模块和一个 concat IP。



Step6: 双击逻辑门模块，将其配置为非功能。



Step7: 按以下电路，完善整体电路。



Step8: 右键单击 Block 文件，文件选择 Generate the Output Products。

Step9: 右键单击 Block 文件，选择 Create a HDL wrapper，根据 Block 文件内容产生一个 HDL 的顶层文件，并选择让 vivado 自动完成。

Step10: 添加约束文件，在我们提供的源程序包的 DOC 文件夹下找到 XDC 文件夹，将其中的约束文件添加到工程当中来。

Step10: 生成 Bit 文件。

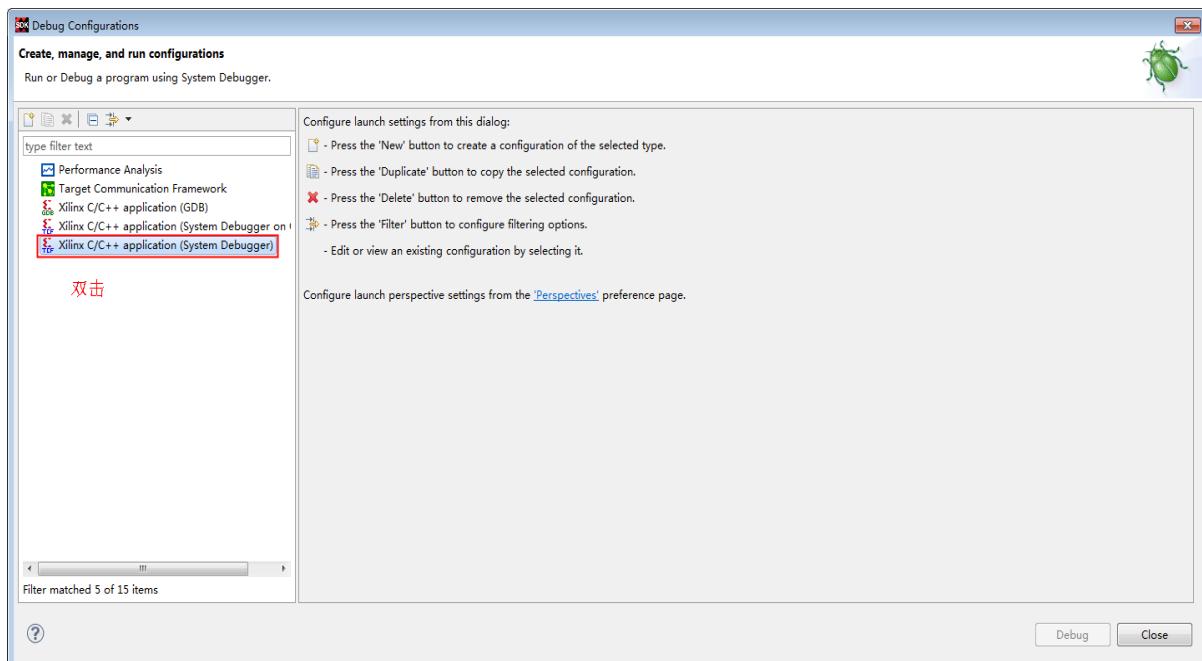
### 8.3 加载到 SDK

Step1: 导出硬件。

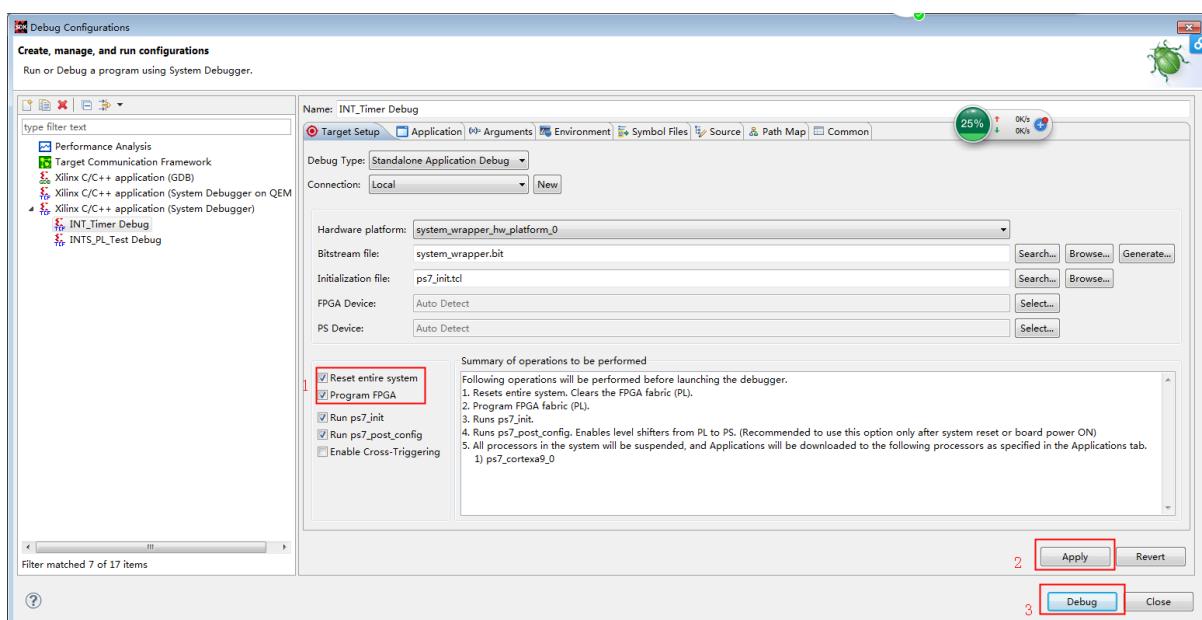
Step2：新建一个空 SDK 工程，并将我们提供的设计文件复制到工程当中来（不熟悉的参考本季第 2、3 章 SDK 部分）。

Step3：右击工程，选择 Debug as ->Debug configuration。

Step4：选中 system Debugger，双击创建一个系统调试。



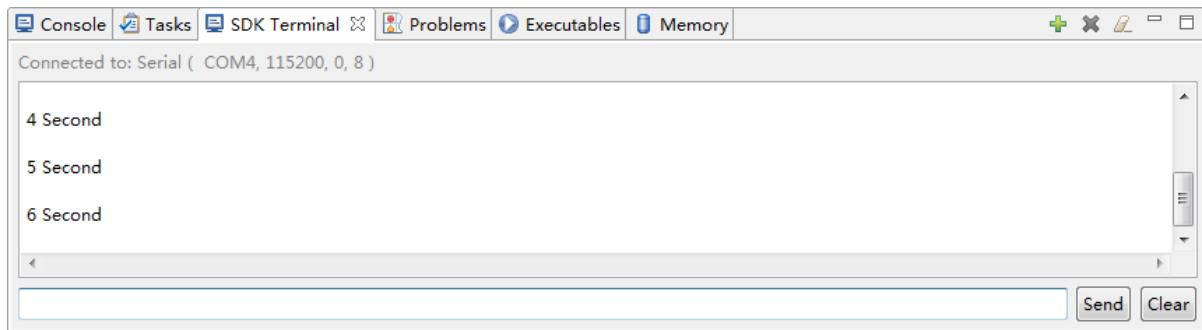
Step5：设置系统调试。



打开系统自带的窗口调试助手，点击运行按钮开始运行程序。



系统运行结果如下图所示：



## 8.4 程序分析

本章的程序讲解依然是从 main 函数处开始。首先我们看看整个程序的结构。

```
int main()
{
    XScuTimer_Config *TMRConfigPtr;      //timer config
    printf("-----START-----\n");
    //私有定时器初始化
    TMRConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
    XScuTimer_CfgInitialize(&Timer, TMRConfigPtr, TMRConfigPtr->BaseAddr);
    //set up the interrupts
    SetupInterruptSystem(&Intc, &Timer, TIMER_IRQ_INTR); //#
    //加数计数周期，私有定时器的时钟为CPU的一般，为333MHZ，如果计数1S，加数值为1sx(333x1000x1000)(1/s)-1=0x13D92D3F
    XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
    //自动装载
    XScuTimer_EnableAutoReload(&Timer);
    //启动定时器
    XScuTimer_Start(&Timer);
    while(1);

    return 0;
}
```

程序一开始的指针和测试程序就不再啰嗦了。接下来的查找配置程序也与我们上一章 PL\_PS 中断是一样的，只是换了个函数名字与基址而已。还未掌握的可以看看我们上一章的分析。

接下来看到定时器的初始化程序，直接跟踪这个程序，查看其定义。如下图所示：

```
/*
 * Initialize a specific timer instance/driver. This function must be called
 * before other functions of the driver are called.
 */
*Xil_Status XScuTimer_CfgInitialize(XScuTimer *InstancePtr,
                                     XScuTimer_Config *ConfigPtr, u32 EffectiveAddress)
{
    s32 Status;
    Xil_AssertNonvoid(InstancePtr != NULL);
    Xil_AssertNonvoid(ConfigPtr != NULL);

    /*
     * If the device is started, disallow the initialize and return a
     * status indicating it is started. This allows the user to stop the
     * device and reinitialize, but prevents a user from inadvertently
     * initializing.
     */
    if (InstancePtr->IsStarted != XIL_COMPONENT_IS_STARTED) {
        /*
         * Copy configuration into the instance structure.
         */
        InstancePtr->Config.DeviceId = ConfigPtr->DeviceId;

        /*
         * Save the base address pointer such that the registers of the block
         * can be accessed and indicate it has not been started yet.
         */
        InstancePtr->Config.BaseAddr = EffectiveAddress;
        InstancePtr->IsStarted = (u32)0;

        /*
         * Indicate the instance is ready to use, successfully initialized.
         */
        InstancePtr->IsReady = XIL_COMPONENT_IS_READY;
    }
    Status = (s32)XST_SUCCESS;
}
else {
    Status = (s32)XST_DEVICE_IS_STARTED;
}
```

Xilinx 官方提供的程序，开头都会给出程序的功能和参数的注释，若是不懂程序是什么意思，不妨先看看这些。从图片上的程序功能注释来看，这是一个指定定时器的初始化函数，在这个定时器被其他函数调用之前，这个函数必须先被调用。也就是说必须先进行定时器的初始化，定时器才能正常的使用。接下来看到程序部分。程序一开始用了一个特定的函数来检测传递进来的参数是否

是空的，如果是，则不能正常跳转到下一个语句。

接下来的一句是检测定时器是否已经开始了（也就是有没有初始化成功），如果没有，就跳到 if 中的语句里面。否则，返回一个已经初始化了的标志。接下来我们看到 if 语句里面的程序。

```

/*
 * Copy configuration into the instance structure.
 */
InstancePtr->Config.DeviceId = ConfigPtr->DeviceId;

/*
 * Save the base address pointer such that the registers of the block
 * can be accessed and indicate it has not been started yet.
 */
InstancePtr->Config.BaseAddr = EffectiveAddress;

InstancePtr->IsStarted = (u32)0;

/*
 * Indicate the instance is ready to use, successfully initialized.
 */
InstancePtr->IsReady = XIL_COMPONENT_IS_READY;

Status =(s32)XST_SUCCESS;

```

一开始，程序把配置指针中的设备 id 拷贝进入了定时器的实例结构中的 DeviceId。接着把程序的最后一个参数 EffectiveAddress（可以猜到这个是一个基地址，具体是什么现在还不知晓）也传递到了定时器的实例结构中的 BaseAddr，紧接着把实例结构里的 IsStarted 标志置为 0，再之后把实例结构中的 IsReady 标志置为 XIL\_COMPONENT\_IS\_READY。最后再给 Status 变量赋值为 XST\_SUCCUSS。可以看出来，定时器的一系列的初始化都是围绕着这个实例结构来的。那么，我们就来看看这个实例结构到底是什么？我们在主函数中找到这个实例结构。

`static XScuTimer Timer;` 在这里，这个实例结构是指向一个结构体的，我们来看看这个结构体的内容。

```

typedef struct {
    XScuTimer_Config Config; /*< Hardware Configuration */
    u32 IsReady;           /*< Device is initialized and ready */
    u32 IsStarted;         /*< Device timer is running */
} XScuTimer;

```

可以看到，这个结构体中又包含了一个结构体，我们再继续看看它包含的这个结构体。

```

/**
 * This typedef contains configuration information for the device.
 */
typedef struct {
    u16 DeviceId;        /*< Unique ID of device */
    u32 BaseAddr;        /*< Base address of the device */
} XScuTimer_Config;

```

此处，我们发现这两个结构体中的内容正好是我们刚才初始化程序中配置的那些参数。接下来，我们再来看看这些参数是如何来的。这就得看到刚才我们提到过的查找配置程序了。

```

TMRConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
XScuTimer_CfgInitialize(&Timer, TMRConfigPtr, TMRConfigPtr->BaseAddr);

```

这些参数就是通过查找配置这个程序获取的。我们回过来看看这个程序。

```

XScuTimer_Config *XScuTimer_LookupConfig(u16 DeviceId)
{
    XScuTimer_Config *CfgPtr = NULL;
    u32 Index;

    for (Index = 0U; Index < XPAR_XSCUTIMER_NUM_INSTANCES; Index++) {
        if (XScuTimer_ConfigTable[Index].DeviceId == DeviceId) {
            CfgPtr = &XScuTimer_ConfigTable[Index];
            break;
        }
    }

    return (XScuTimer_Config *)CfgPtr;
}

```

从上图可以看到，这些配置是存放在一个数组当中的，让我们继续查看一下数组。

```
XScuTimer_Config XScuTimer_ConfigTable[] =
{
    {
        XPAR_PS7_SCUTIMER_0_DEVICE_ID,
        XPAR_PS7_SCUTIMER_0_BASEADDR
    }
};
```

图中的两个对象，是我们 parameters.h 中系统自动生成的定时器的设备地址和基址。只要我们在硬件电路上添加了定时器，那这两个参数就会自动被添加，定时器的参数也将会自动生成。

回到 main 函数的分析，接下来的是一个建立中断的函数，这个函数带了三个参数：第一个参数指向了中断控制器，第二个指向的是定时器，第三个是中断号。将鼠标放在中断号上面时，我们可以发现中断号为 29。我们可以在 ug585 的中断部分查看一下中断号 29 是什么类型的中断。

Table 7-3: Private Peripheral Interrupts (PPI)

IRQ ID#	Name	PPI#	Type	Description
26:16	Reserved	~	~	Reserved
27	Global Timer	0	Rising edge	Global timer
28	nFIQ	1	Active Low level (active High at PS-PL interface)	Fast interrupt signal from the PL: CPU0: IRQF2P[18] CPU1: IRQF2P[19]
29	CPU Private Timer	2	Rising edge	Interrupt from private CPU timer
30	AWDT{0, 1}	3	Rising edge	Private watchdog timer for each CPU
31	nIRQ	4	Active Low level (active High at PS-PL interface)	Interrupt signal from the PL: CPU0: IRQF2P[16] CPU1: IRQF2P[17]

可以看到这是定时器中断，上升沿触发的。这样定义是有一定依据的。这段程序与上一章 PL\_PS 中断是差不多的，我们上一章对其进行过详细的分析，大家可参照上一章介绍的方法对其进行分析。

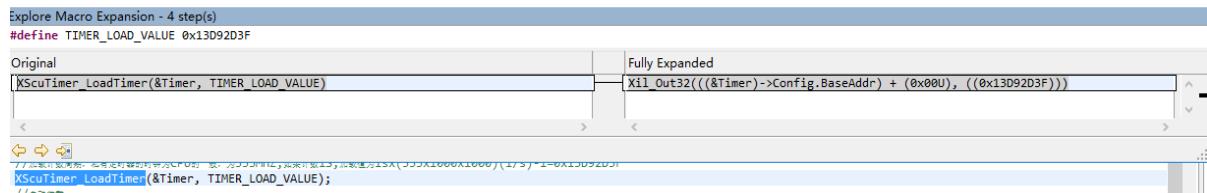
回到 main 函数，接下来的这句是本章程序中的核心部分。它将程序的定时时间设置为了 1 秒。那么，系统是如何做到定时一秒的呢？定时器时间可以通过下式计算：定时时间 = 1/定时器频率 \* (预加载值+1)，则可以推算出：预加载值=定时时间\*定时器频率-1。定时时间是已知的，如果再知道定时器频率则可以计算出加载的值，查看 xilinx 的编程指导手册 ug585-zynq-7000-TRM 的定时器篇得知：

### 8.3.1 Clocking

The GTC is always clocked at 1/2 of the CPU frequency (CPU\_3x2x).

定时器频率为处理器频率的一半，比如 Mz702 的 ARM 工作频率为 666MHz，则私有定时器的频率为 333MHz，则加载值为  $1*333\_000\_000*(1/s) - 1 = 0x13D92D3F$ 。

回到 main 函数的分析，当我们把鼠标停留在装载加载值函数 XScuTimer\_LoadTimer 上时，SDK 会显示关于这个函数的一些信息。



我们看到这个函数的原函数是向一个寄存器地址中写入了预加载值，我们计算一下这个寄存器地址。原函数的第一个参数我们刚才提到过，就是那个实例结构中的基址，也就是定时器的基地址。我们在 xparameters.h 中找到它。

```
/* Definitions for peripheral PS7_SCUTIMER_0 */
#define XPAR_PS7_SCUTIMER_0_DEVICER_ID 0
#define XPAR_PS7_SCUTIMER_0_BASEADDR 0xF8F00600
#define XPAR_PS7_SCUTIMER_0_HIGHADDR 0xF8F0061F
```

此时我们就可以计算了： $F8F00600 + 00 = F8F00600$ 。在 ug585 中查找一下这个寄存器地址，看看这个寄存器是干什么用的。

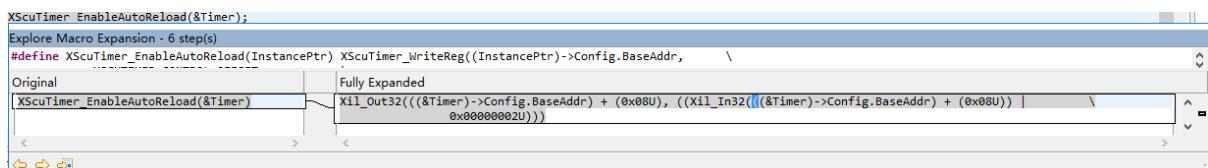
#### Register ([mpcore](#)) Private\_Timer\_Load\_Register

Name	Private_Timer_Load_Register
Software Name	TIMER_LOAD
Relative Address	0x00000600
Absolute Address	0xF8F00600
Width	32 bits
Access Type	rw
Reset Value	0x00000000
Description	Private Timer Load Register

#### Register Private\_Timer\_Load\_Register Details

Field Name	Bits	Type	Reset Value	Description
	31:0	rw	0x0	The Timer Load Register contains the value copied to the Timer Counter Register when it decrements down to zero with auto reload mode enabled. Writing to the Timer Load Register means that you also write to the Timer Counter Register.

可以看到，这个寄存器就是个装载预加载值的寄存器。接着看到 main 函数的下一句。



这段程序与上一句差不多一致，我们通过分析寄存器，看看这段程序完成的功能。这段程序的寄存器地址为： $F8F00600 + 08 = F8F00608$ 。这段程序写入的数据为： $(F8F00600 + 08) | 0x00000002 = F8F0060A$ 。查找 ug585 看看寄存器的功能。

#### Register ([mpcore](#)) Private\_Timer\_Control\_Register

Name	Private_Timer_Control_Register
Software Name	TIMER_CONTROL

Relative Address	0x000000608
Absolute Address	0xF8F00608
Width	32 bits
Access Type	rw
Reset Value	0x00000000
Description	Private Timer Control Register

#### Register Private\_Timer\_Control\_Register Details

Field Name	Bits	Type	Reset Value	Description
SBZP	31:16	rw	0x0	UNK/SBZP.
Prescaler (PRESCALER)	15:8	rw	0x0	The prescaler modifies the clock period for the decrementing event for the Counter Register. See Calculating timer intervals on page 4-2 for the equation.\n
UNK_SBZP	7:3	rw	0x0	UNK/SBZP.
IRQ_Enable (IRQ_ENABLE)	2	rw	0x0	If set, the interrupt ID 29 is set as pending in the Interrupt Distributor when the event flag is set in the Timer Status Register.
Auto_reload (AUTO_RELOAD)	1	rw	0x0	1'b0 = Single shot mode. Counter decrements down to zero, sets the event flag and stops. 1'b1 = Auto-reload mode. Each time the Counter Register reaches zero, it is reloaded with the value contained in the Timer Load Register.
Timer_Enable (ENABLE)	0	rw	0x0	Timer enable 1'b0 = Timer is disabled and the counter does not decrement. All registers can still be read and written 1'b1 = Timer is enabled and the counter decrements normally

这个寄存器是一个预加载值控制寄存器，通过写入我们上面分析出的那个数据，把中断的预加载值设置为了自动装载模式（也就是中断一次过后，系统又会自动的装入初值，不用人工载入初值），也就是图中用方框圈出的部分。

回到 main 函数，讲解最后一个函数，启动定时器的函数。还是先跟踪一下这个函数。

```
/****************************************************************************
 * Start the timer.
 * @param    InstancePtr is a pointer to the XScuTimer instance.
 * @return   None.
 * @note    None.
 */
void XScuTimer_Start(XScuTimer *InstancePtr)
{
    u32 Register;

    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

    /*
     * Read the contents of the Control register.
     */
    Register = XScuTimer_ReadReg(InstancePtr->Config.BaseAddr,
                                 XSCUTIMER_CONTROL_OFFSET);

    /*
     * Set the 'timer enable' bit in the register.
     */
    Register |= XSCUTIMER_CONTROL_ENABLE_MASK;

    /*
     * Update the Control register with the new value.
     */
    XScuTimer_WriteReg(InstancePtr->Config.BaseAddr,
                       XSCUTIMER_CONTROL_OFFSET, Register);

    /*
     * Indicate that the device is started.
     */
    InstancePtr->IsStarted = XIL_COMPONENT_IS_STARTED;
}
```

可以看出来，这也是一個通过读写寄存器的方式来操作定时器的过程，我们依然来分析一下寄存器。

```

Register = XScuTimer_ReadReg(InstancePtr->Config.BaseAddr,
                             XSCUTIMER_CONTROL_OFFSET); //F8F00600+08=F8F00608

Explore Macro Expansion - 2 step(s)
#define XSCUTIMER_CONTROL_OFFSET 0x08U

Original
XScuTimer_ReadReg(InstancePtr->Config.BaseAddr,
                  XSCUTIMER_CONTROL_OFFSET)
      |
      +-- Fully Expanded
          Xil_In32((InstancePtr->Config.BaseAddr) + (0x08U))
  
```

之前的一些初始化程序就跳过不再讲解了，直接看到上图所示的程序，这个程序的分析与我们刚才讲的装载初值的方法是一样的，这里我们可以直接计算此程序读出的寄存器地址：F8F00600+08=F8F00608。

```

Register |= XSCUTIMER_CONTROL_ENABLE_MASK;
/* * Update the register with the new
   value */
  
```

这一句的意思就是把刚才得到的寄存器的地址与 0x00000001 或操作。此时寄存器地址为：F8F00608 | 0x00000001U =F8F00609。

```

XScuTimer_WriteReg(InstancePtr->Config.BaseAddr,
                   XSCUTIMER_CONTROL_OFFSET, Register);

Explore Macro Expansion - 2 step(s)
#define XSCUTIMER_CONTROL_OFFSET 0x08U

Original
XScuTimer_WriteReg(InstancePtr->Config.BaseAddr,
                   XSCUTIMER_CONTROL_OFFSET, Register)
      |
      +-- Fully Expanded
          Xil_Out32((InstancePtr->Config.BaseAddr) + (0x08U), (Register))
  
```

这里我们发现，上图中这个函数的源程序中，第一个参数即为我们第一次得到的寄存器地址，写入的数据为第二次得到的寄存器地址。也就是说向 F8F00608 这个寄存器里写入数据 F8F00609。查看 ug585，看看这么配置是什么意思。

Relative Address	0x000000608
Absolute Address	0xF8F00608
Width	32 bits
Access Type	rw
Reset Value	0x00000000
Description	Private Timer Control Register

Register Private\_Timer\_Control\_Register Details

Field Name	Bits	Type	Reset Value	Description
SBZP	31:16	rw	0x0	UNK/SBZP.
Prescaler (PRESCALER)	15:8	rw	0x0	The prescaler modifies the clock period for the decrementing event for the Counter Register. See Calculating timer intervals on page 4-2 for the equation.\
UNK_SBZP	7:3	rw	0x0	UNK/SBZP.
IRQ_Enable (IRQ_ENABLE)	2	rw	0x0	If set, the interrupt ID 29 is set as pending in the Interrupt Distributor when the event flag is set in the Timer Status Register.
Auto_reload (AUTO_RELOAD)	1	rw	0x0	1'b0 = Single shot mode. Counter decrements down to zero, sets the event flag and stops. 1'b1 = Auto-reload mode. Each time the Counter Register reaches zero, it is reloaded with the value contained in the Timer Load Register.
Timer_Enable (ENABLE)	0	rw	0x0	Timer enable 1'b0 = Timer is disabled and the counter does not decrement. All registers can still be read and written 1'b1 = Timer is enabled and the counter decrements normally

此时就可以清晰的知晓，通过控制这个寄存器的最后一位，就可以控制定时器的工作与否，刚才我们写入的是 F8F00609，将最后一位置 1，也就是启动了定时器。

## 8.5 本章小结

中断对于实时系统是非常重要的，可以说说是实时性的保障吧。本章简要介绍了 ZYNQ 的中断原理和中断类型，详细介绍了私有定时器，建立了完整的工程进行测试。

## CH09\_UART 串口中断实验

本章的 UART 中断将在之前 PL\_PS 中断和定时器中断上推导出来，因此本章有点难度，如果前两章还不是很熟悉的话，需要返回到前面两章把这两章的内容再次消化一下，再来学习本章的内容。本章的硬件工程可以直接使用定时器中断的硬件工程，因此此次试验就直接到 SDK 软件部分。

### 9.1 加载到 SDK

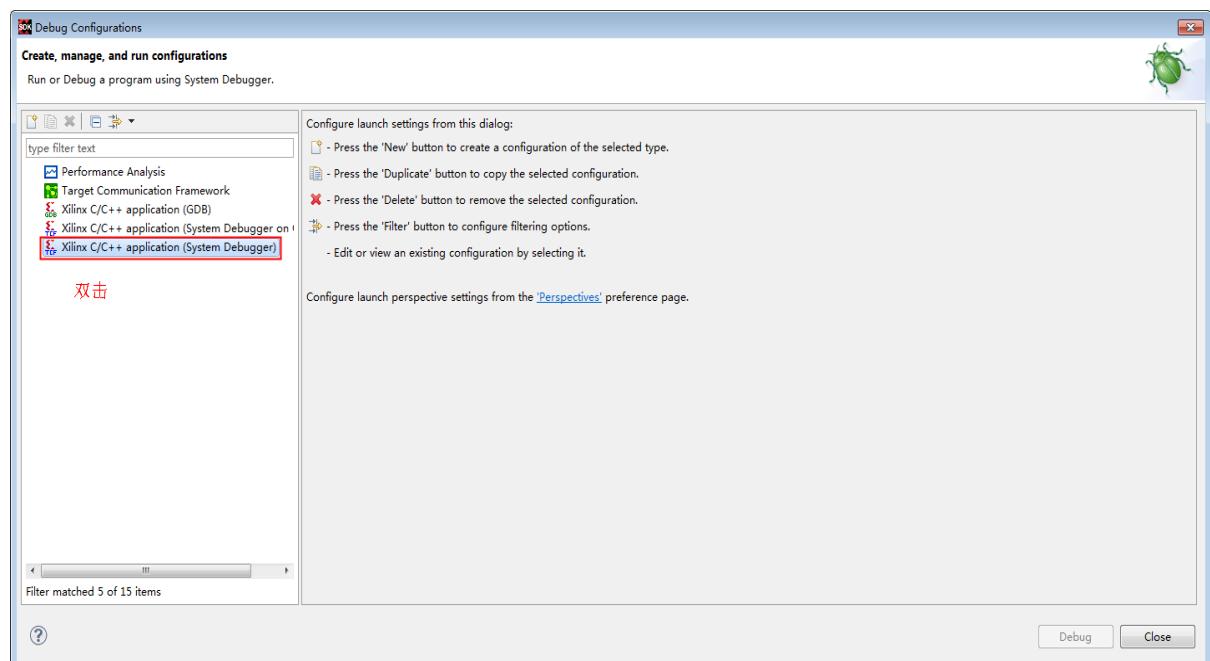
Step1：打开定时器中断的工程。

Step2：导出硬件。

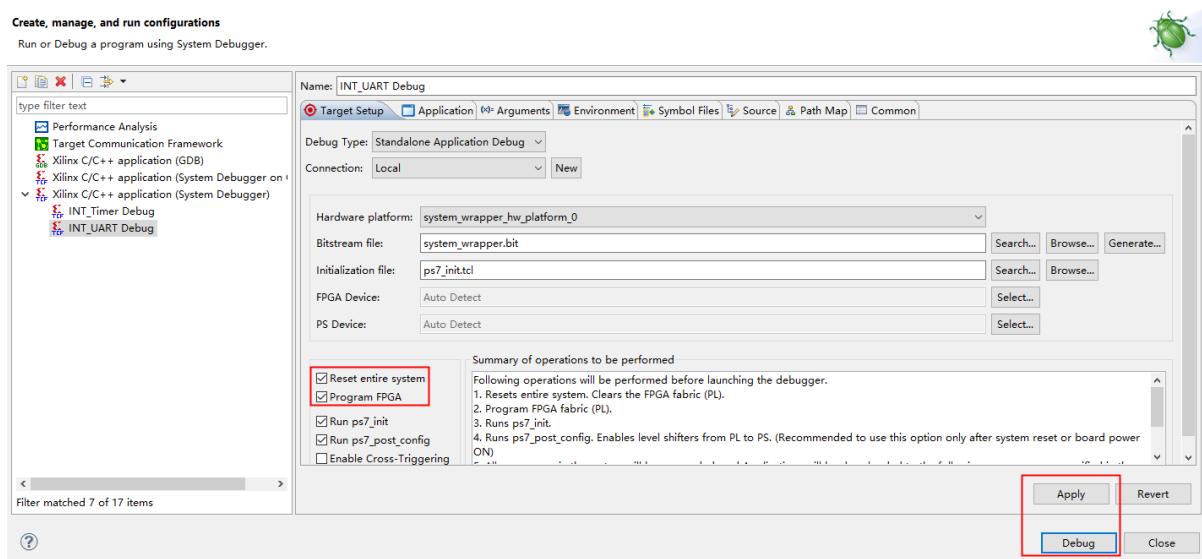
Step3：新建一个空 SDK 工程，并将我们提供的设计文件复制到工程当中来。

Step4：右击工程，选择 Debug as ->Debug configuration。

Step5：选中 system Debugger, 双击创建一个系统调试。



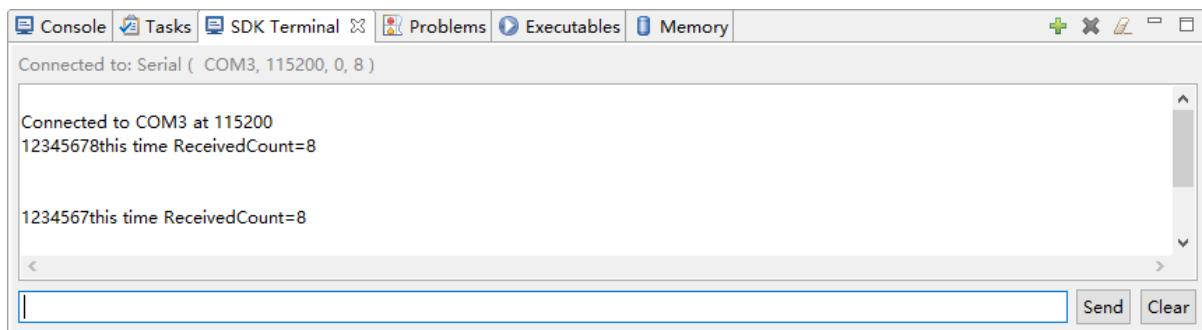
Step6：设置系统调试。



打开系统自带的窗口调试助手，点击运行按钮开始运行程序。



系统运行结果如下图所示：



## 9.2 程序分析

本章的程序与之前两章的程序都大同小异，一些函数都在我们之前两章中看到和介绍过。

首先我们先介绍下面三个宏定义。

```
//timer info
#define UART_DEVICE_ID      XPAR_PS7_UART_1_DEVICE_ID
#define INTC_DEVICE_ID      XPAR_SCUGIC_SINGLE_DEVICE_ID
#define UART_IRPT_INTR     XPAR_XUARTPS_1_INTR
```

第一个是我们的 UART 的设备 ID，第二个是我们中断的设备 ID，第三个是 UART 的中断号。把鼠标停留在 UART 的中断号上，按下 F3 跟踪它，经过两次跟踪后，得到 UART 的中断号如下图所示：

```
#define XPS_I2C1_INT_ID      80U
#define XPS_SPI1_INT_ID       81U
#define XPS_UART1_INT_ID      82U
#define XPS_CAN1_INT_ID       83U
```

我们可以在 ug585 中查看一下中断号 82 是否是串口中断。

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]	~	~	~

可以看到，确实是串口中断，高电平触发。

再来看看 main 函数中的内容。首先依然是通过查找配置程序来获取串口的硬件配置。我们跟踪这个程序，看看他获取的配置是什么。

```
/****************************************************************************
 * Looks up the device configuration based on the unique device ID. The table
 * contains the configuration info for each device in the system.
 *
 * @param    DeviceId contains the ID of the device
 *
 * @return   A pointer to the configuration structure or NULL if the
 *          specified device is not in the system.
 *
 * @note     None.
 *
 ****/
XUartPs_Config *XUartPs_LookupConfig(u16 DeviceId)
{
    XUartPs_Config *CfgPtr = NULL;

    u32 Index;

    for (Index = 0U; Index < (u32)XPAR_XUARTPS_NUM_INSTANCES; Index++) {
        if (XUartPs_ConfigTable[Index].DeviceId == DeviceId) {
            CfgPtr = &XUartPs_ConfigTable[Index];
            break;
        }
    }

    return (XUartPs_Config *)CfgPtr;
}
/** @} */
```

这个程序还是从一个配置表数组中查找的配置文件，继续往下剥离，看一看这个数组中的内容。

```
XUartPs_Config XUartPs_ConfigTable[] =
{
    {
        XPAR_PS7_UART_1_DEVICE_ID,
        XPAR_PS7_UART_1_BASEADDR,
        XPAR_PS7_UART_1_UART_CLK_FREQ_HZ,
        XPAR_PS7_UART_1_HAS_MODEM
    }
};
```

可以看到，这个数组里存放的是 UART 的设备 ID, UART 的基地址，时钟频率和一个不知道什么作用的对象。后两个参数是我们没用到的，因此就略过了。前两个都是我们在硬件工程中添加了中断后，系统自动生成的。

接下来还是一个熟悉的函数，对 UART 进行了初始化。可以看到这个函数的第一个参数指向了定义的 UART 指针，我们就跟踪一下这个指针。

`static XUartPs_Uart; //uart` 我们发现它指向了一个结构体，那么我们继续跟踪看看结构体中内容。

```
typedef struct {
    XUartPs_Config Config; /* Configuration data structure */
    u32 InputClockHz; /* Input clock frequency */
    u32 IsReady; /* Device is initialized and ready */
    u32 BaudRate; /* Current baud rate */

    XUartPsBuffer SendBuffer;
    XUartPsBuffer ReceiveBuffer;

    XUartPs_Handler Handler;
    void *CallBackRef; /* Callback reference for event handler */
    u32 Platform;
} XUartPs;
```

这个结构体中的内容比较多，第一个对象是我们 UART 硬件的一些配置，它指向的是一个结构体。那么就来看看这个结构体吧。

```
typedef struct {
    u16 DeviceId; /* Unique ID of device */
    u32 BaseAddress; /* Base address of device (IPIF) */
    u32 InputClockHz; /* Input clock frequency */
    s32 ModemPinsConnected; /* Specifies whether modem pins are connected
                             * to MIO or FMI0 */
} XUartPs_Config;
```

可以看到，这些就是刚才我们查找配置程序获取到的硬件参数。

回到 XUartPs 结构体的分析。第二个对象是输入时钟频率，第三个是设备是否初始化并准备好，第四个是波特率，第五个是两个 buffer,一个发送的一个接收的。挑选一个参看一下。

```
/* Keep track of state information about a data buffer in the interrupt mode. */
typedef struct {
    u8 *NextBytePtr;
    u32 RequestedBytes;
    u32 RemainingBytes;
} XUartPsBuffer;
```

接着第七个是一个 Hander,第八个是一个回掉函数，最后一个 platform 具体是什么意思不得而知。

回到初始化程序。我们来看看这个函数与之前有什么不同了。

```
s32 XUartPs_CfgInitialize(XUartPs *InstancePtr,
                           XUartPs_Config * Config, u32 EffectiveAddr)
{
    s32 Status;
    u32 ModeRegister;
    u32 BaudRate;

    /* Assert validates the input arguments */
    Xil_AssertNonvoid(InstancePtr != NULL);
    Xil_AssertNonvoid(Config != NULL);

    /* Setup the driver instance using passed in parameters */
    InstancePtr->Config.BaseAddress = EffectiveAddr;
    InstancePtr->Config.InputClockHz = Config->InputClockHz;
    InstancePtr->Config.Modem PinsConnected = Config->Modem PinsConnected;

    /* Initialize other instance data to default values */
    InstancePtr->Handler = XUartPs_StubHandler;

    InstancePtr->SendBuffer.NextBytePtr = NULL;
    InstancePtr->SendBuffer.RemainingBytes = 0U;
    InstancePtr->SendBuffer.RequestedBytes = 0U;

    InstancePtr->ReceiveBuffer.NextBytePtr = NULL;
    InstancePtr->ReceiveBuffer.RemainingBytes = 0U;
    InstancePtr->ReceiveBuffer.RequestedBytes = 0U;

    /* Initialize the platform data */
    InstancePtr->Platform = XGetPlatform_Info();

    /* Flag that the driver instance is ready to use */
    InstancePtr->IsReady = XIL_COMPONENT_IS_READY;

    /*
     * Set the default baud rate here, can be changed prior to
     * starting the device
     */
    BaudRate = (u32)XUARTPS_DFT_BAUDRATE;
    Status = XUartPs_SetBaudRate(InstancePtr, BaudRate);
    if (Status != (s32)XST_SUCCESS) {
        InstancePtr->IsReady = 0U;
    } else {
```

```

    /*
     * Set up the default data format: 8 bit data, 1 stop bit, no
     * parity
     */
    ModeRegister = XUartPs_ReadReg(InstancePtr->Config.BaseAddress,
                                    XUARTPS_MR_OFFSET);

    /* Mask off what's already there */
    ModeRegister &= (~((u32)XUARTPS_MR_CHARLEN_MASK |
                      (u32)XUARTPS_MR_STOPMODE_MASK |
                      (u32)XUARTPS_MR_PARITY_MASK));

    /* Set the register value to the desired data format */
    ModeRegister |= ((u32)XUARTPS_MR_CHARLEN_8_BIT |
                     (u32)XUARTPS_MR_STOPMODE_1_BIT |
                     (u32)XUARTPS_MR_PARITY_NONE);

    /* Write the mode register out */
    XUartPs_WriteReg(InstancePtr->Config.BaseAddress, XUARTPS_MR_OFFSET,
                     ModeRegister);

    /* Set the RX FIFO trigger at 8 data bytes. */
    XUartPs_WriteReg(InstancePtr->Config.BaseAddress,
                     XUARTPS_RXWM_OFFSET, 0x08U);

    /* Set the RX timeout to 1, which will be 4 character time */
    XUartPs_WriteReg(InstancePtr->Config.BaseAddress,
                     XUARTPS_RXTOUT_OFFSET, 0x01U);

    /* Disable all interrupts, polled mode is the default */
    XUartPs_WriteReg(InstancePtr->Config.BaseAddress, XUARTPS_IDR_OFFSET,
                     XUARTPS_IXR_MASK);

    Status = XST_SUCCESS;
}
return Status;
}

```

一开始是一长串的初始化，如下图所示：

```

s32 Status;
u32 ModeRegister;
u32 BaudRate;

/* Assert validates the input arguments */
Xil_AssertNonvoid(InstancePtr != NULL);
Xil_AssertNonvoid(Config != NULL);

/* Setup the driver instance using passed in parameters */
InstancePtr->Config.BaseAddress = EffectiveAddr;
InstancePtr->Config.InputClockHz = Config->InputClockHz;
InstancePtr->Config.ModemPinsConnected = Config->ModemPinsConnected;

/* Initialize other instance data to default values */
InstancePtr->Handler = XUartPs_StubHandler;

InstancePtr->SendBuffer.NextBytePtr = NULL;
InstancePtr->SendBuffer.RemainingBytes = 0U;
InstancePtr->SendBuffer.RequestedBytes = 0U;

InstancePtr->ReceiveBuffer.NextBytePtr = NULL;
InstancePtr->ReceiveBuffer.RemainingBytes = 0U;
InstancePtr->ReceiveBuffer.RequestedBytes = 0U;

```

接下来的这个函数是一个用于判断芯片类型的函数。

```
u32 XGetPlatform_Info()
{
    u32 reg;
#if defined (ARMR5) || (__aarch64__)
    return XPLAT_ZYNQ_ULTRA_MP;
#elif defined (__microblaze__)
    return XPLAT_MICROBLAZE;
#else
    return XPLAT_ZYNQ;
#endif
}
```

接下来，程序将 Instance(也就是我们的 UART 硬件)的标志设置为 XIL\_COMPONENT\_IS\_READY，表明此时 UART 已经可以使用了。

```
/* Flag that the driver instance is ready to use */
InstancePtr->IsReady = XIL_COMPONENT_IS_READY;
```

接下来，程序将 UART 的波特率设置为了 115200。

```
/
BaudRate = (u32)XUARTPS_DFT_BAUDRATE;
Status = XUartPs_SetBaudRate(InstancePtr, BaudRate);
if (Status != (s32)XIL_COMPONENT_MACRO_EXPANSION) {
    InstancePtr->BaudRate = 115200U;
} else {
```

接下来的这一句是读取 UART 的模式寄存器。

```
ModeRegister = XUartPs_ReadReg(InstancePtr->Config.BaseAddress,
                                XUARTPS_MR_OFFSET);
```

我们可以来看看读取的什么内容，把鼠标停放在这个函数的上方，看到函数显示出了这个函数的原函数。



与我们定时器实验中讲到的读写寄存器的函数差不多，第一个参数是 UART 的基地址，这在我们一开始的分析中就提到过，我们反回去看看 UART 的基地址是多少。

```
XUartPs_Config XUartPs_ConfigTable[] =
{
    {
        XPAR_PS7_UART_1_DEVICE_ID,
        XPAR_PS7_UART_1_BASEADDR,
        Macro Expansion -> UART_CLK_FREQ_HZ,
        0xE0001000 -> HAS_MODEM
    }
};
```

可以知道，此处的基地址为 0xE0001000，直接计算：E0001000+0x0004=E0001004。打开 ug585 查看下这个寄存器的介绍。

Register ([UART](#)) mode\_reg0

Name	mode_reg0
Software Name	MR
Relative Address	0x00000004
Absolute Address	uart0: 0xE0000004 uart1: 0xE0001004
Width	32 bits
Access Type	mixed
Reset Value	0x00000000
Description	UART Mode Register

## Register mode\_reg0 Details

The UART Mode register defines the setup of the data format to be transmitted or received. If this register is modified during transmission or reception, data validity cannot be guaranteed.

Field Name	Bits	Type	Reset Value	Description
reserved	31:12	ro	0x0	Reserved, read as zero, ignored on write.
reserved	11	rw	0x0	Reserved. Do not modify.
reserved	10	rw	0x0	Reserved. Do not modify.

Field Name	Bits	Type	Reset Value	Description
CHMODE	9:8	rw	0x0	Channel mode: Defines the mode of operation of the UART. 00: normal 01: automatic echo 10: local loopback 11: remote loopback
NBSTOP	7:6	rw	0x0	Number of stop bits: Defines the number of stop bits to detect on receive and to generate on transmit. 00: 1 stop bit 01: 1.5 stop bits 10: 2 stop bits 11: reserved
PAR	5:3	rw	0x0	Parity type select: Defines the expected parity to check on receive and the parity to generate on transmit. 000: even parity 001: odd parity 010: forced to 0 parity (space) 011: forced to 1 parity (mark) 1xx: no parity
CHRL	2:1	rw	0x0	Character length select: Defines the number of bits in each character. 11: 6 bits 10: 7 bits 0x: 8 bits
CLKS (CLKSEL)	0	rw	0x0	Clock source select: This field defines whether a pre-scaler of 8 is applied to the baud rate generator input clock. 0: clock source is uart_ref_clk 1: clock source is uart_ref_clk/8

可以看到，这是一个 UART 的模式寄存器，通过这个寄存器可以设置串口的数据位宽，有无停止位和奇偶校验位等信息。

再来看看下一句程序。这句是对刚才读出的寄存器的一个运算。

```
/* Mask off what's already there */
ModeRegister &= (~((u32)XUARTPS_MR_CHARLEN_MASK |
    (u32)XUARTPS_MR_STOPMODE_MASK |
    (u32)XUARTPS_MR_PARITY_MASK));
```

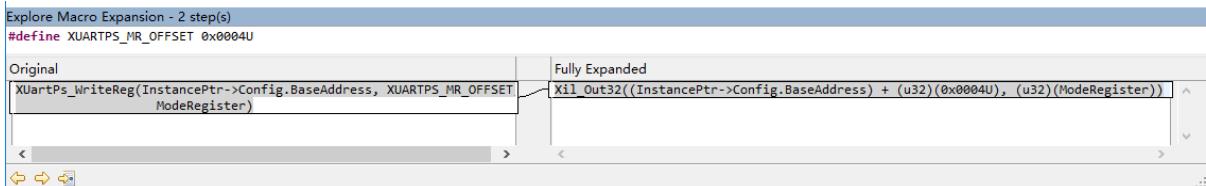
首先得到方框中这三个参数的值。这里我们已经查看程序得知这三个值分别为：6, A0, 38。然后进行运算：ModeRegister=E0001004 & (~(6|A0|38))=E0001004 & 11 =0。

```
/* Set the register value to the desired data format */
ModeRegister |= ((u32)XUARTPS_MR_CHARLEN_8_BIT |
    (u32)XUARTPS_MR_STOPMODE_1_BIT |
    (u32)XUARTPS_MR_PARITY_NONE); //0|(0|0|20)=20
```

接下来的这一句也是一个运算，不多讲，直接运算。ModeRegister=0|(0|20)=20。

```
/* Write the mode register out */
XUartPs_WriteReg(InstancePtr->Config.BaseAddress, XUARTPS_MR_OFFSET,
    ModeRegister);
```

这段程序就是一个写寄存器的功能了。看看这个函数的原函数。



由此得出，这个函数读写的地址为刚才模式寄存器的地址，写入的数据就是运算得出的 20h。参照刚才 ug585 里的模式寄存器说明，显而易见，经过这段程序之后，把 UART 设置为了 8 个数据位，1 个停止位和无奇偶校验位的模式。

接下来的还有 3 个写寄存器的程序，分析方法与刚才的一致。这里就只给出它们实现的功能。分别是：设置 UART 的 RX FIFO 在 8bit 处触发、设置 UART 的超时为 1（4 个字符时间）、禁止所有中断轮询模式为默认的样式。

回到 main 函数的分析当中，接下来的函数实现的是建立起中断的功能，这个函数在我们上一章也进行过详细的讲解。这里我们关注一下下面这个函数。

```

static void UartIntrHandler(void *CallBackRef)
{
    XUartPs *InstancePtr = (XUartPs *) CallBackRef;
    u32 IsrStatus;

    u32 ReceivedCount=0;

    u32 CsrRegister;
    /*
     * Read the interrupt ID register to determine which
     * interrupt is active
     */
    IsrStatus = XUartPs_ReadReg(InstancePtr->Config.BaseAddress,
                                XUARTPS_IMR_OFFSET); //e0001000+10=regaddr=e0001010

    IsrStatus &= XUartPs_ReadReg(InstancePtr->Config.BaseAddress,
                                XUARTPS_ISR_OFFSET); //e0001000+14=regaddr=e0001014

    /* Dispatch an appropriate handler. */
    if((IsrStatus & ((u32)XUARTPS_RXR_RXOVR | (u32)XUARTPS_RXR_RXEMPTY |
        (u32)XUARTPS_RXR_RXFULL)) != (u32)0) {

        CsrRegister = XUartPs_ReadReg(InstancePtr->Config.BaseAddress, //判断FIFO触发标准位
                                       XUARTPS_SR_OFFSET); //e0001000+2c=regaddr=e000102c

        while((CsrRegister & XUARTPS_SR_RXEMPTY)== (u32)0){ //读取FIFO中所有数据
            //InstancePtr->ReceiveBuffer.NextBytePtr[ReceivedCount] = //每次循环读取1byte
            ;

            XUartPs_WriteReg(InstancePtr->Config.BaseAddress, //每次循环发送读取到的数据
                             XUARTPS_FIFO_OFFSET,
                             XUartPs_ReadReg(InstancePtr->Config.
                                             BaseAddress,
                                             XUARTPS_FIFO_OFFSET));

            ReceivedCount++; //计数
            CsrRegister = XUartPs_ReadReg(InstancePtr->Config.BaseAddress,
                                          XUARTPS_SR_OFFSET);
        }
    }
    printf("this_time_ReceivedCount=%d\r\n",ReceivedCount);
    XUartPs_WriteReg(InstancePtr->Config.BaseAddress, XUARTPS_ISR_OFFSET,
                     IsrStatus);
}

```

当我们运行 XScuGic\_Connect 这个函数的时候，实际运行的就是这个回调函数。这个函数也是真正实现 UART 发送与接收功能的函数。可以看到这个程序是通过读写寄存器的方式来工作的，我们可以用刚才我们讲到的方法对其进行分析，在程序中，我们也给出了分析的过程。大家可以认真的去看一看。

### 9.3 本章小结

本章主要详细的分析了 UART 中断的实现过程，通过本章，我们重点需要掌握的是怎样分析一个问题的方法。通过这几张中断部分的讲解，我们应该做到对中断部分得心应手的程度。

## CH10\_ User GPIO 实验

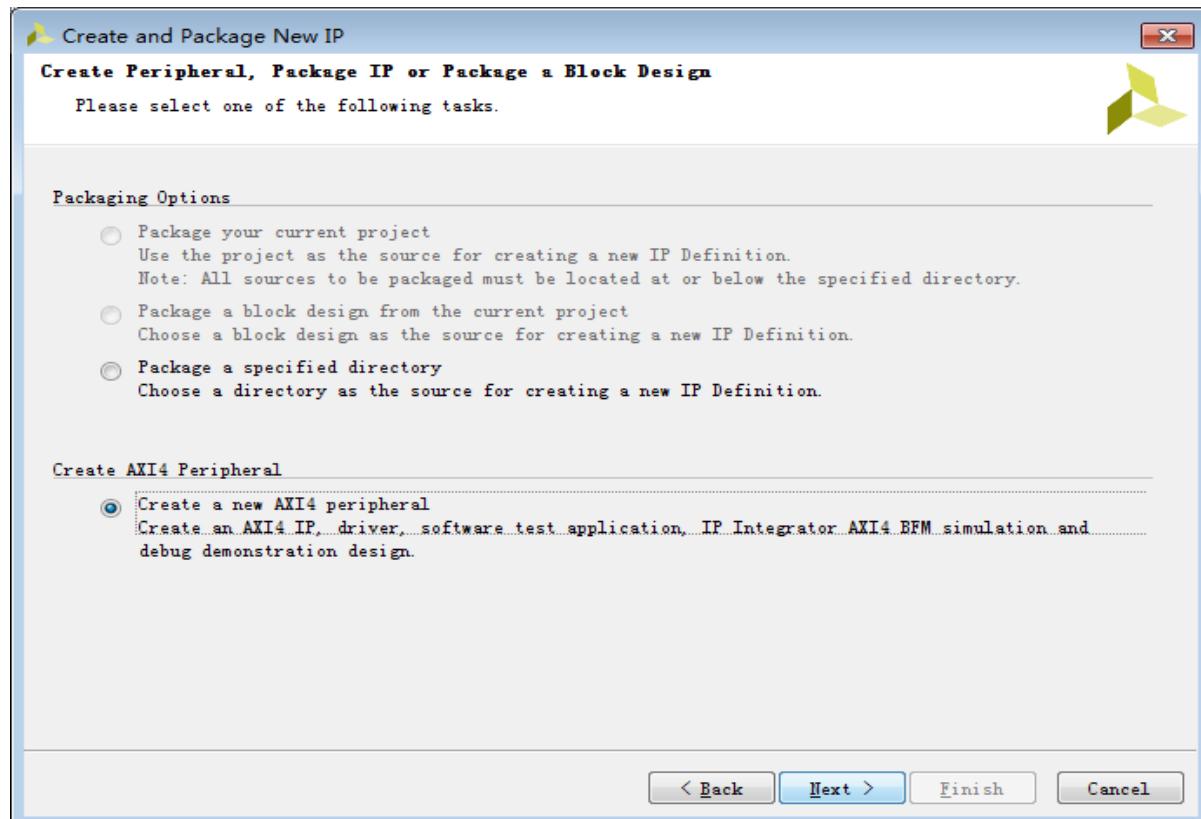
在之前的第四章课程中，我们详细的讲解了如何在 VIVADO 软件下封装一个简单的流水灯程序。在 ZYNQ 开发过程中，有时候我们可能会需要与 ARM 硬核进行通信，在这种情况下，可能就需要用到更高速的接口与 ARM 通信。本章就将讲解如何创建一个基于高速的 AXI 总线的 IP。本章将带领大家创建一个带 AXI 总线接口的自定义 GPIO 模拟的流水灯实验。通过这种方法，我们可以在 GPIO 资源缺乏的情况下，利用 PL 的资源来扩充 GPIO 资源。

### 10.1 创建 IP

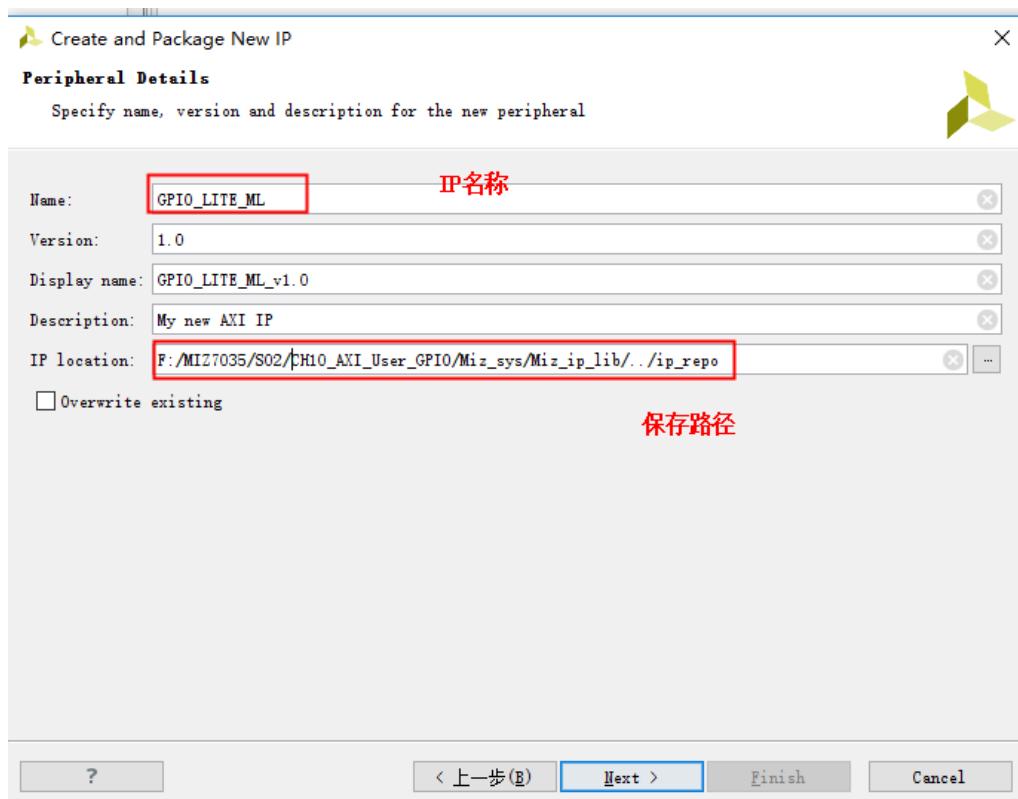
Step1：打开 VIVADO 软件，新建一个工程。

Step2：单击 Tools 菜单下的 Create and package IP。

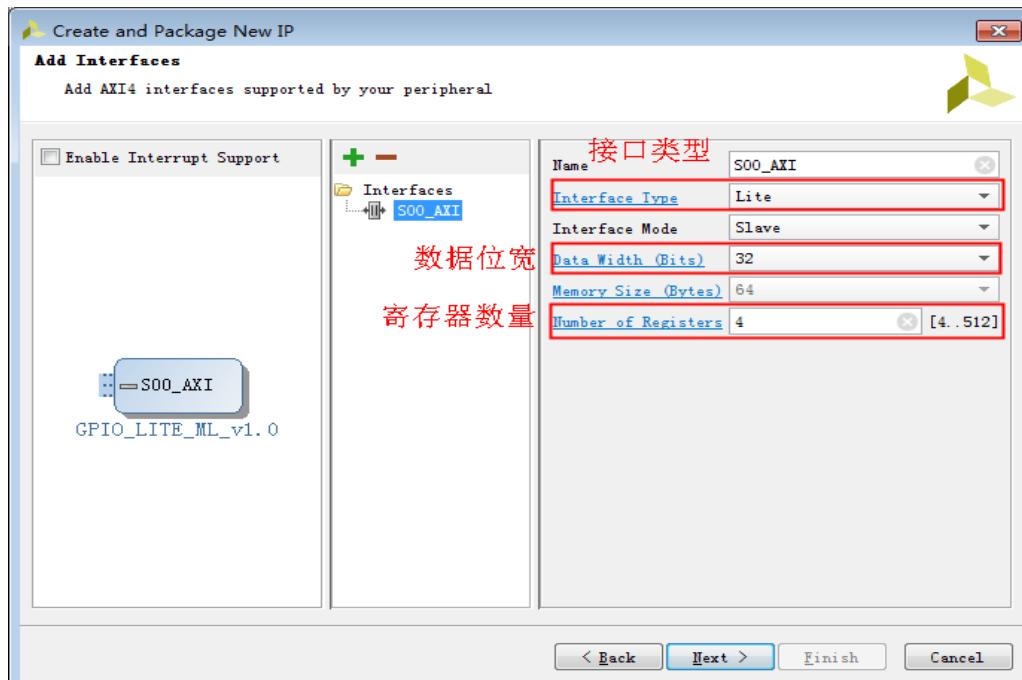
Step3：单击 Next，选择 Create a new AXI4 peripheral，单击 Next。



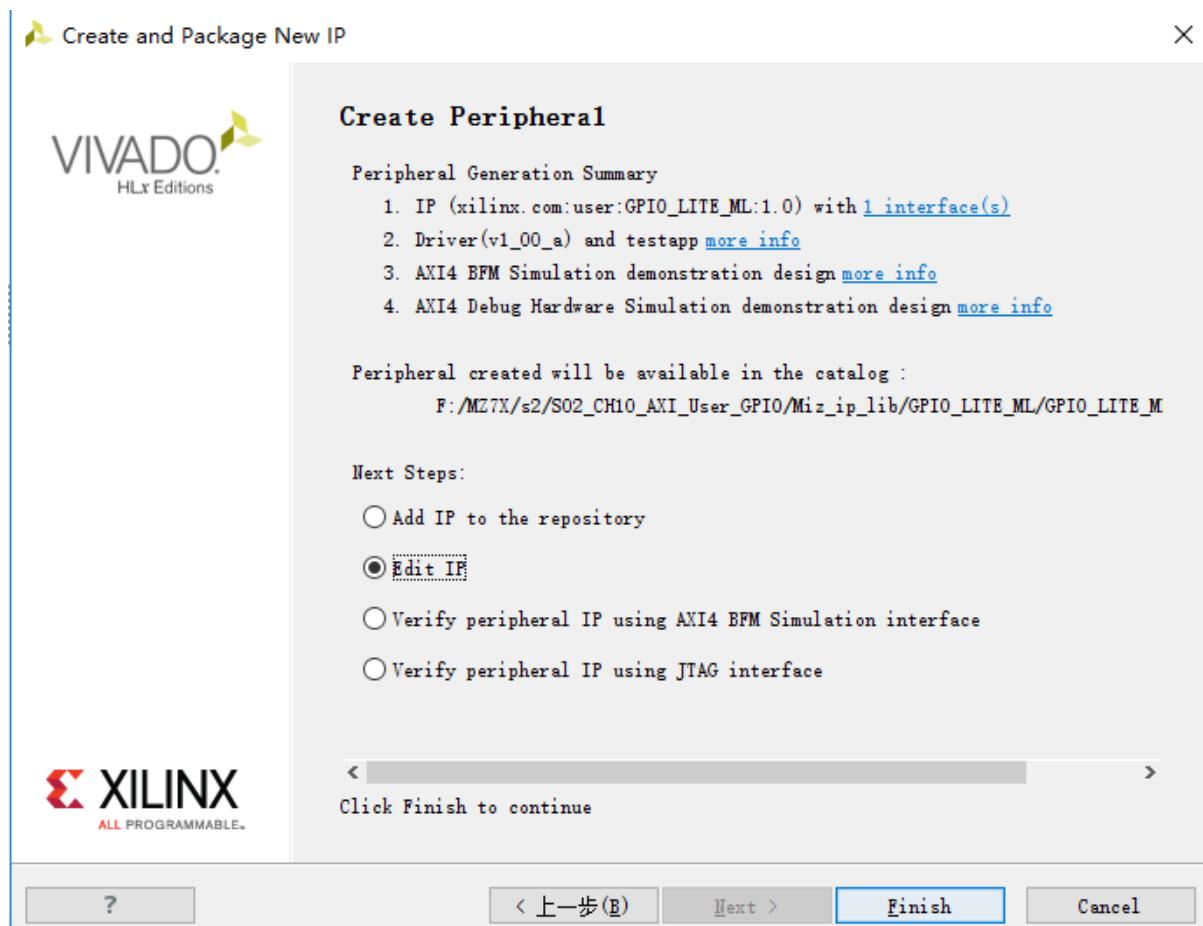
Step4：输入要创建的 IP 名字，此处我们命名为 GPIO\_LITE\_ML,选择好保存路劲,单击 Next。



Step5: 选择接口类型为 lite,数据位宽为 32 位, 寄存器数量为 4, 然后单击 next。



Step6:选择 Edit IP, 然后选择 Finish 按钮将打开一个新的编辑 IP 的工程。



Step7: 选中 Project Manager, 双击 GPIO\_LITE\_ML\_v1\_0\_S00\_inst, 用以下程序替换原来的程序。

```
`timescale 1 ns / 1 ps

module GPIO_LITE_ML_v1_0_S00_AXI #
(
    // Users to add parameters here

    // User parameters ends
    // Do not modify the parameters beyond this line

    // Width of S_AXI data bus
    parameter integer C_S_AXI_DATA_WIDTH      = 32,
    // Width of S_AXI address bus
    parameter integer C_S_AXI_ADDR_WIDTH      = 4
)
```

```
(  
    // Users to add ports here  
    output wire [7:0]GPIO_LED,  
    // User ports ends  
    // Do not modify the ports beyond this line  
  
    // Global Clock Signal  
    input wire  S_AXI_ACLK,  
    // Global Reset Signal. This Signal is Active LOW  
    input wire  S_AXI_ARESETN,  
    // Write address (issued by master, acceped by Slave)  
    input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,  
    // Write channel Protection type. This signal indicates the  
        // privilege and security level of the transaction, and whether  
        // the transaction is a data access or an instruction access.  
    input wire [2 : 0] S_AXI_AWPROT,  
    // Write address valid. This signal indicates that the master signaling  
        // valid write address and control information.  
    input wire  S_AXI_AWVALID,  
    // Write address ready. This signal indicates that the slave is ready  
        // to accept an address and associated control signals.  
    output wire  S_AXI_AWREADY,  
    // Write data (issued by master, acceped by Slave)  
    input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,  
    // Write strobes. This signal indicates which byte lanes hold  
        // valid data. There is one write strobe bit for each eight  
        // bits of the write data bus.  
    input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,  
    // Write valid. This signal indicates that valid write  
        // data and strobes are available.  
    input wire  S_AXI_WVALID,  
    // Write ready. This signal indicates that the slave  
        // can accept the write data.  
    output wire  S_AXI_WREADY,  
    // Write response. This signal indicates the status  
        // of the write transaction.
```

```
output wire [1 : 0] S_AXI_BRESP,  
    // Write response valid. This signal indicates that the channel  
    // is signaling a valid write response.  
output wire  S_AXI_BVALID,  
    // Response ready. This signal indicates that the master  
    // can accept a write response.  
input wire   S_AXI_BREADY,  
    // Read address (issued by master, accepted by Slave)  
input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,  
    // Protection type. This signal indicates the privilege  
    // and security level of the transaction, and whether the  
    // transaction is a data access or an instruction access.  
input wire [2 : 0] S_AXI_ARPROT,  
    // Read address valid. This signal indicates that the channel  
    // is signaling valid read address and control information.  
input wire  S_AXI_ARVALID,  
    // Read address ready. This signal indicates that the slave is  
    // ready to accept an address and associated control signals.  
output wire  S_AXI_ARREADY,  
    // Read data (issued by slave)  
output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,  
    // Read response. This signal indicates the status of the  
    // read transfer.  
output wire [1 : 0] S_AXI_RRESP,  
    // Read valid. This signal indicates that the channel is  
    // signaling the required read data.  
output wire  S_AXI_RVALID,  
    // Read ready. This signal indicates that the master can  
    // accept the read data and response information.  
input wire   S_AXI_RREADY  
);  
  
// AXI4LITE signals  
reg [C_S_AXI_ADDR_WIDTH-1 : 0] axi_awaddr;  
reg      axi_awready;  
reg      axi_wready;
```

```
reg [1 : 0]      axi_bresp;
reg          axi_bvalid;
reg [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_araddr;
reg          axi_arready;
reg [C_S_AXI_DATA_WIDTH-1 : 0]  axi_rdata;
reg [1 : 0]      axi_rresp;
reg          axi_rvalid;

// Example-specific design signals
// local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
// ADDR_LSB is used for addressing 32/64 bit registers/memories
// ADDR_LSB = 2 for 32 bits (n downto 2)
// ADDR_LSB = 3 for 64 bits (n downto 3)
localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
localparam integer OPT_MEM_ADDR_BITS = 1;
-----
//-- Signals for user logic register space example
-----
//-- Number of Slave Registers 4
reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg0;
reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg1;
reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg2;
reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg3;
wire  slv_reg_rden;
wire  slv_reg_wren;
reg [C_S_AXI_DATA_WIDTH-1:0]    reg_data_out;
integer   byte_index;

// I/O Connections assignments

assign S_AXI_AWREADY  = axi_awready;
assign S_AXI_WREADY   = axi_wready;
assign S_AXI_BRESP    = axi_bresp;
assign S_AXI_BVALID   = axi_bvalid;
assign S_AXI_ARREADY  = axi_arready;
assign S_AXI_RDATA    = axi_rdata;
```

```
assign S_AXI_RRESP = axi_rresp;
assign S_AXI_RVALID = axi_rvalid;
// Implement axi_awready generation
// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
// de-asserted when reset is low.

always @(posedge S_AXI_ACLK)
begin
if (S_AXI_ARESETN == 1'b0)
begin
    axi_awready <= 1'b0;
end
else
begin
    if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
        begin
            // slave is ready to accept write address when
            // there is a valid write address and write data
            // on the write address and data bus. This design
            // expects no outstanding transactions.
            axi_awready <= 1'b1;
        end
    else
        begin
            axi_awready <= 1'b0;
        end
    end
end

// Implement axi_awaddr latching
// This process is used to latch the address when both
// S_AXI_AWVALID and S_AXI_WVALID are valid.

always @(posedge S_AXI_ACLK)
begin
```

```
if ( S_AXI_ARESETN == 1'b0 )
begin
    axi_awaddr <= 0;
end
else
begin
    if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
        begin
            // Write Address latching
            axi_awaddr <= S_AXI_AWADDR;
        end
    end
end

// Implement axi_wready generation
// axi_wready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
// de-asserted when reset is low.

always @(posedge S_AXI_ACLK )
begin
if ( S_AXI_ARESETN == 1'b0 )
begin
    axi_wready <= 1'b0;
end
else
begin
    if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID)
        begin
            // slave is ready to accept write data when
            // there is a valid write address and write data
            // on the write address and data bus. This design
            // expects no outstanding transactions.
            axi_wready <= 1'b1;
        end
    end
end
else
```

```
begin
    axi_wready <= 1'b0;
end
end
end

// Implement memory mapped register select and write logic generation
// The write data is accepted and written to memory mapped registers when
// axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes are
used to
// select byte enables of slave registers while writing.
// These registers are cleared when reset (active low) is applied.
// Slave register write enable is asserted when valid address and data are available
// and the slave is ready to accept the write address and write data.
assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID;

always @(*(posedge S_AXI_ACLK))
begin
if ( S_AXI_ARESETN == 1'b0 )
begin
    slv_reg0 <= 0;
    slv_reg1 <= 0;
    slv_reg2 <= 0;
    slv_reg3 <= 0;
end
else begin
    if (slv_reg_wren)
begin
    case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        2'h0:
            for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                    // Respective byte enables are asserted as per write strobes
                    // Slave register 0
                    slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];

```

```
        end

    2'h1:
        for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
            if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                // Respective byte enables are asserted as per write strobes
                // Slave register 1
                slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
            end
    2'h2:
        for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
            if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                // Respective byte enables are asserted as per write strobes
                // Slave register 2
                slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
            end
    2'h3:
        for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
            if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                // Respective byte enables are asserted as per write strobes
                // Slave register 3
                slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
            end
        default : begin
            slv_reg0 <= slv_reg0;
            slv_reg1 <= slv_reg1;
            slv_reg2 <= slv_reg2;
            slv_reg3 <= slv_reg3;
        end
    endcase
end
end
end
```

```
// Implement write response logic generation
// The write response and response valid signals are asserted by the slave
// when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
// This marks the acceptance of address and indicates the status of
// write transaction.

always @(posedge S_AXI_ACLK)
begin
    if (S_AXI_ARESETN == 1'b0)
        begin
            axi_bvalid <= 0;
            axi_bresp <= 2'b0;
        end
    else
        begin
            if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready && S_AXI_WVALID)
                begin
                    // indicates a valid write response is available
                    axi_bvalid <= 1'b1;
                    axi_bresp <= 2'b0; // 'OKAY' response
                end
                // work error responses in future
            end
        else
            begin
                if (S_AXI_BREADY && axi_bvalid)
                    //check if bready is asserted while bvalid is high)
                    //there is a possibility that bready is always asserted high)
                    begin
                        axi_bvalid <= 1'b0;
                    end
            end
        end
    end

// Implement axi_arready generation
```

```
// axi_already is asserted for one S_AXI_ACLK clock cycle when
// S_AXI_ARVALID is asserted. axi_awready is
// de-asserted when reset (active low) is asserted.
// The read address is also latched when S_AXI_ARVALID is
// asserted. axi_araddr is reset to zero on reset assertion.

always @(posedge S_AXI_ACLK)
begin
    if (S_AXI_ARESETN == 1'b0)
        begin
            axi_already <= 1'b0;
            axi_araddr  <= 32'b0;
        end
    else
        begin
            if (~axi_already && S_AXI_ARVALID)
                begin
                    // indicates that the slave has accepted the valid read address
                    axi_already <= 1'b1;
                    // Read address latching
                    axi_araddr  <= S_AXI_ARADDR;
                end
            else
                begin
                    axi_already <= 1'b0;
                end
        end
    end

    // Implement axi_arvalid generation
    // axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
    // S_AXI_ARVALID and axi_already are asserted. The slave registers
    // data are available on the axi_rdata bus at this instance. The
    // assertion of axi_rvalid marks the validity of read data on the
    // bus and axi_rresp indicates the status of read transaction.axi_rvalid
    // is deasserted on reset (active low). axi_rresp and axi_rdata are
```

```
// cleared to zero on reset (active low).
always @(posedge S_AXI_ACLK)
begin
if (S_AXI_ARESETN == 1'b0)
begin
    axi_rvalid <= 0;
    axi_rresp  <= 0;
end
else
begin
if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
begin
    // Valid read data is available at the read data bus
    axi_rvalid <= 1'b1;
    axi_rresp  <= 2'b0; // 'OKAY' response
end
else if (axi_rvalid && S_AXI_RREADY)
begin
    // Read data is accepted by the master
    axi_rvalid <= 1'b0;
end
end
end

// Implement memory mapped register select and read logic generation
// Slave register read enable is asserted when valid address is available
// and the slave is ready to accept the read address.
assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
always @(*)
begin
    // Address decoding for reading registers
    case (axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB])
        2'h0 : reg_data_out <= slv_reg0;
        2'h1 : reg_data_out <= slv_reg1;
        2'h2 : reg_data_out <= slv_reg2;
        2'h3 : reg_data_out <= slv_reg3;
```

```
default : reg_data_out <= 0;
endcase
end

// Output register or memory read data
always @(posedge S_AXI_ACLK)
begin
if (S_AXI_ARESETN == 1'b0)
begin
    axi_rdata <= 0;
end
else
begin
    // When there is a valid read address (S_AXI_ARVALID) with
    // acceptance of read address by the slave (axi_arready),
    // output the read data
    if (slv_reg_rden)
begin
    axi_rdata <= reg_data_out;      // register read data
end
end
end

// Add user logic here

assign GPIO_LED[7:0] = slv_reg0[7:0];
// User logic ends

endmodule
```

以上程序与生成的程序基本一致，只是添加了一个用户输出端口和用户逻辑。修改部分如下图所示：

```

module GPIO_LITE_ML_v1_0_S00_AXI #
(
    // Users to add parameters here
    // User parameters ends
    // Do not modify the parameters beyond this line

    // Width of S_AXI data bus
    parameter integer C_S_AXI_DATA_WIDTH = 32,
    // Width of S_AXI address bus
    parameter integer C_S_AXI_ADDR_WIDTH = 4
)
(
    // Users to add ports here
    output wire [7:0]GPIO_LED,
    // User ports ends
    // Do not modify the ports beyond this line

    // Global Clock Signal
    input wire S_AXI_ACLK,
    // Global Reset Signal. This Signal is Active LOW
    input wire S_AXI_ARESETN,
    // Write address (issued by master, accepted by Slave)
    input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
    // Write channel Protection type. This signal indicates the
    // privilege and security level of the transaction, and whether
    // the transaction is a data access or an instruction access.
    input wire [2 : 0] S_AXI_AWPROT,
    // Output register or memory read data
    always @ (posedge S_AXI_ACLK )
    begin
        if (S_AXI_ARESEIN == 1'b0 )
            begin
                axi_rdata <= 0;
            end
        else
            begin
                // When there is a valid read address (S_AXI_ARVALID) with
                // acceptance of read address by the slave (axi_arready),
                // output the read data
                if (slv_reg_rden)
                    begin
                        axi_rdata <= reg_data_out;      // register read data
                    end
            end
    end
    // Add user logic here
    assign GPIO_LED[7:0] = slv_reg0[7:0];
    // User logic ends
endmodule

```

最后的用户逻辑将 slv\_reg0 的值赋值给了用户输出逻辑，当我们向 slv\_reg0 写入数据的时候，也就相当于向 GPIO\_LED 赋值。

Step8：双击 GPIO\_LITE\_ML 文件，用以下程序替换原来的程序。

```
`timescale 1 ns / 1 ps

module GPIO_LITE_ML #
(
    // Users to add parameters here

    // User parameters ends
    // Do not modify the parameters beyond this line

    // Parameters of Axi Slave Bus Interface S00_AXI
    parameter integer C_S00_AXI_DATA_WIDTH = 32,
    parameter integer C_S00_AXI_ADDR_WIDTH = 4
)
(
    // Users to add ports here
    output wire [7:0]GPIO_LED,
    // User ports ends
    // Do not modify the ports beyond this line

    // Ports of Axi Slave Bus Interface S00_AXI
    input wire    s00_axi_aclk,
    input wire    s00_axi_aresetn,
    input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_awaddr,
    input wire [2 : 0] s00_axi_awprot,
    input wire    s00_axi_awvalid,
    output wire   s00_axi_awready,
    input wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_wdata,
    input wire [(C_S00_AXI_DATA_WIDTH/8)-1 : 0] s00_axi_wstrb,
    input wire    s00_axi_wvalid,
    output wire   s00_axi_wready,
    output wire [1 : 0] s00_axi_bresp,
    output wire   s00_axi_bvalid,
    input wire    s00_axi_bready,
```

```
input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_araddr,
input wire [2 : 0] s00_axi_arprot,
input wire s00_axi_arvalid,
output wire s00_axi_arready,
output wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_rdata,
output wire [1 : 0] s00_axi_rresp,
output wire s00_axi_rvalid,
input wire s00_axi_rready
);

// Instantiation of Axi Bus Interface S00_AXI
GPIO_LITE_ML_v1_0_S00_AXI #(
.C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
.C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
) GPIO_LITE_ML_v1_0_S00_AXI_inst (
.S_AXI_ACLK(s00_axi_aclk),
.S_AXI_ARESETN(s00_axi_aresetn),
.S_AXI_AWADDR(s00_axi_awaddr),
.S_AXI_AWPROT(s00_axi_awprot),
.S_AXI_AWVALID(s00_axi_awvalid),
.S_AXI_AWREADY(s00_axi_awready),
.S_AXI_WDATA(s00_axi_wdata),
.S_AXI_WSTRB(s00_axi_wstrb),
.S_AXI_WVALID(s00_axi_wvalid),
.S_AXI_WREADY(s00_axi_wready),
.S_AXI_BRESP(s00_axi_bresp),
.S_AXI_BVALID(s00_axi_bvalid),
.S_AXI_BREADY(s00_axi_bready),
.S_AXI_ARADDR(s00_axi_araddr),
.S_AXI_ARPROT(s00_axi_arprot),
.S_AXI_ARVALID(s00_axi_arvalid),
.S_AXI_ARREADY(s00_axi_arready),
.S_AXI_RDATA(s00_axi_rdata),
.S_AXI_RRESP(s00_axi_rresp),
.S_AXI_RVALID(s00_axi_rvalid),
.S_AXI_RREADY(s00_axi_rready),
//user port
```

```
.GPIO_LED(GPIO_LED)
);

// Add user logic here

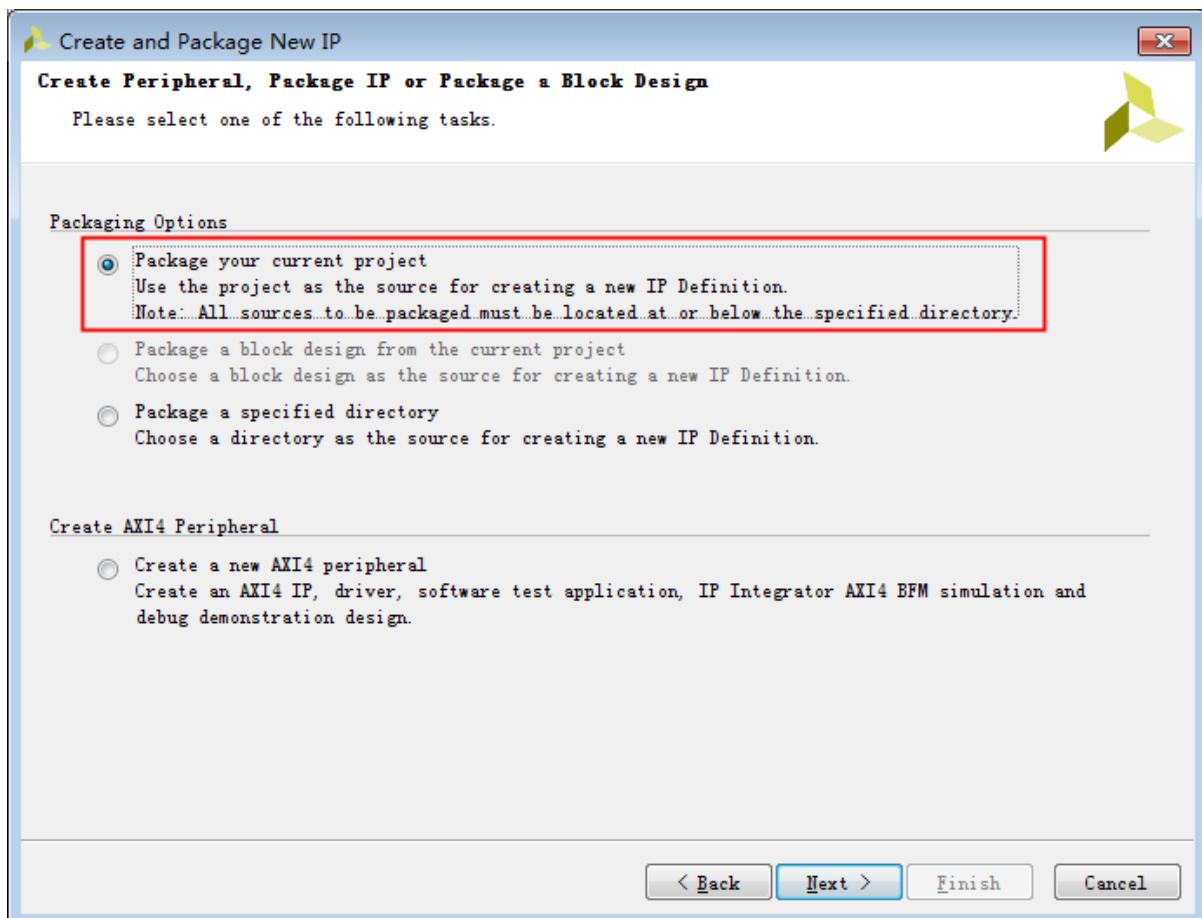
// User logic ends

endmodule
```

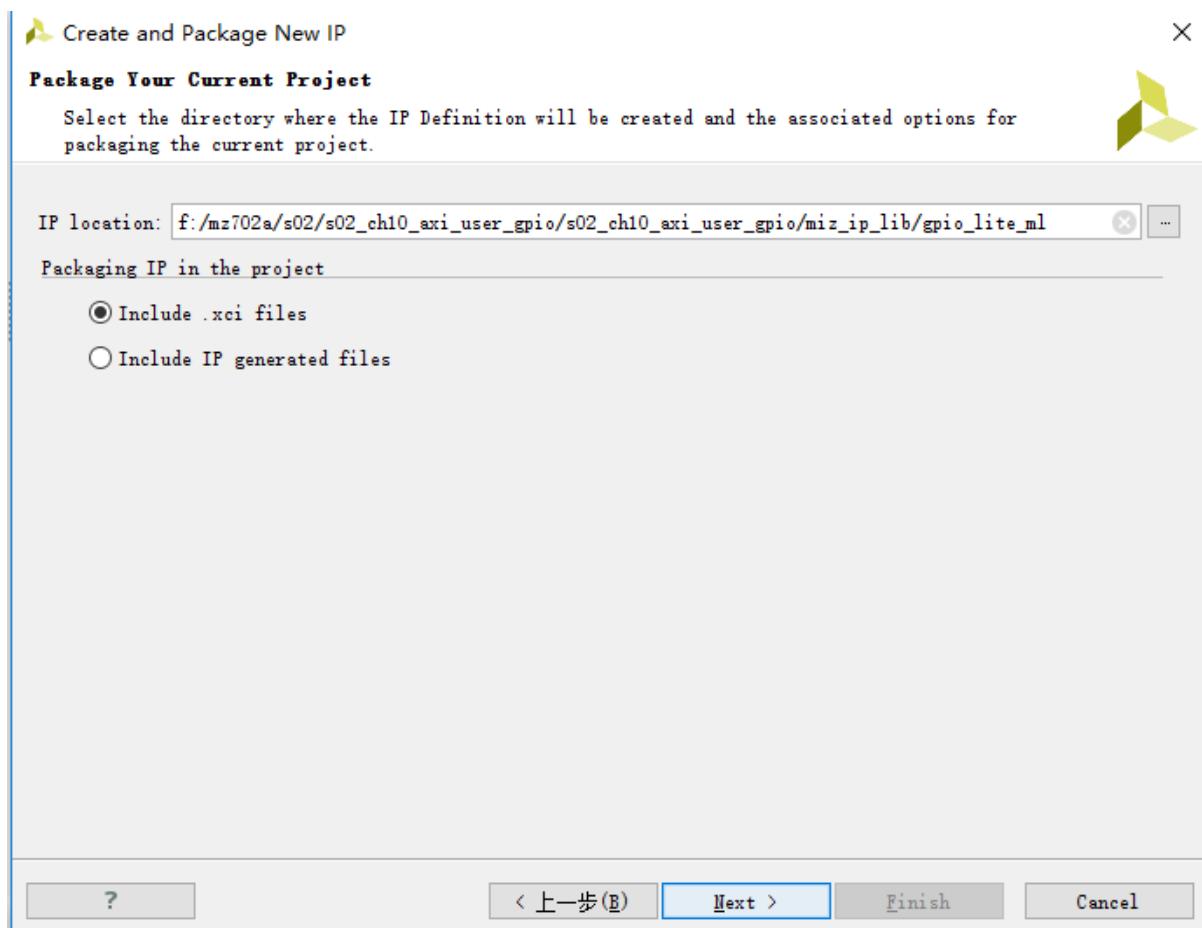
以上的程序也只是在原来的程序的基础上增加了一个用户端口而已，并无什么大的改变。

Step9：单击 Tools 菜单下的 Create and package IP 命令，重新封装 IP。

Step10：单击 Next，选择第一项，单击 Next。

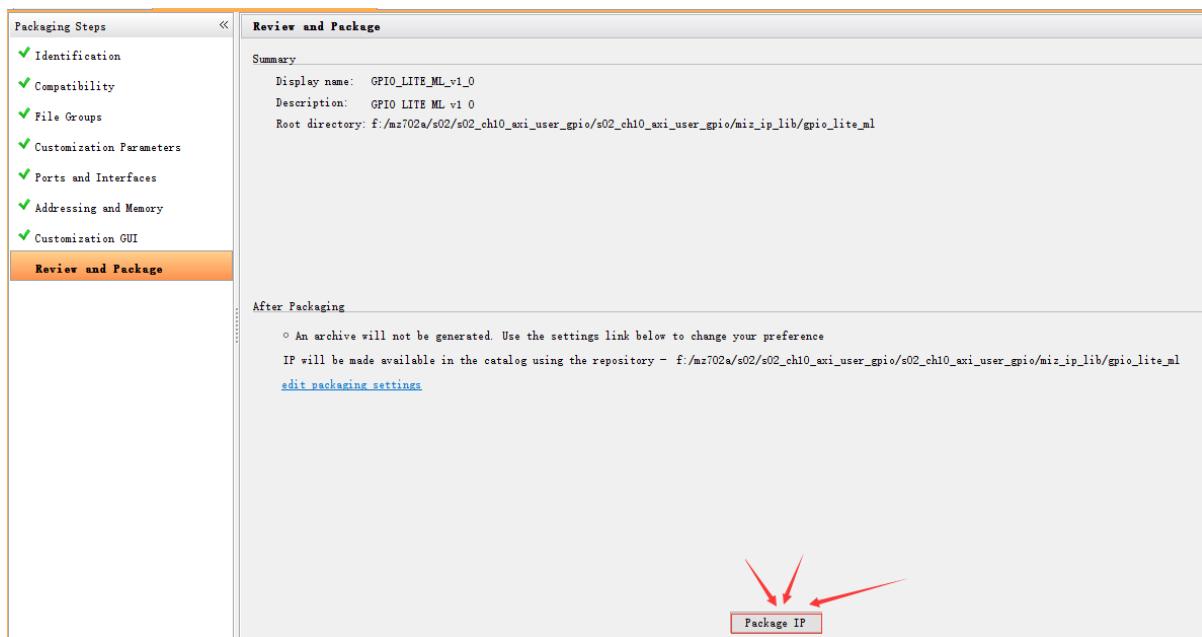


Step11：选择保存的路径，单击 Next。



Step12: 选择 Overwrite, 然后单击 Finish。

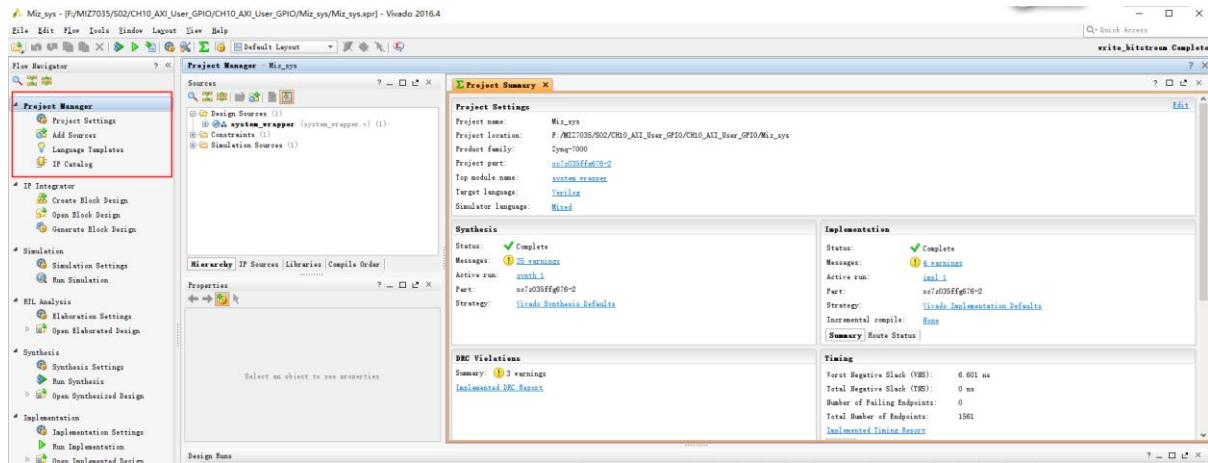
Step13: 在新弹出的窗口中, 我们注意到有一个警告, 直接忽略它, 选择 Review and package IP 选项, 单击底部的 package IP 按钮完成 IP 的创建。



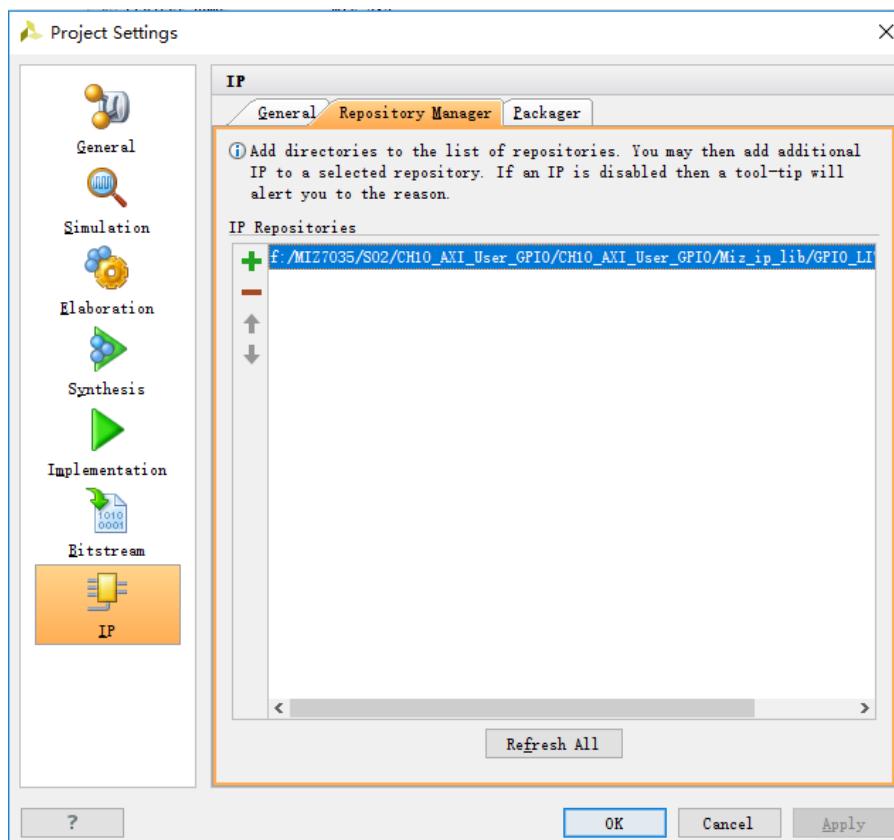
## 10.2 搭建硬件工程

Step1：另外新建一个 VIVADO 工程，根据自己的开发板正确配置芯片型号。

Step2：在 Project manager 区中单击 Project settings。



Step3：选择 IP 设置区中的 repository manager, 将上一节我们封装好的 IP 的路劲添加进去。

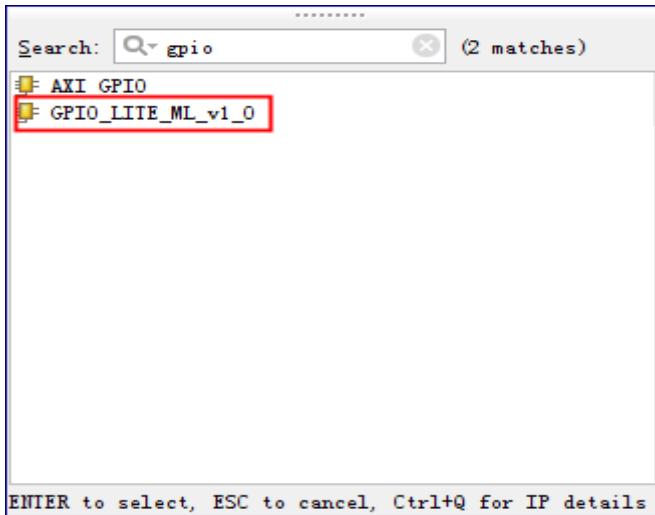


Step4：单击+号图标，将上一节封装的 IP 的路劲存放进去，单击 OK。

Step5：新建一个 BD 文件，输入文件名，完成创建。

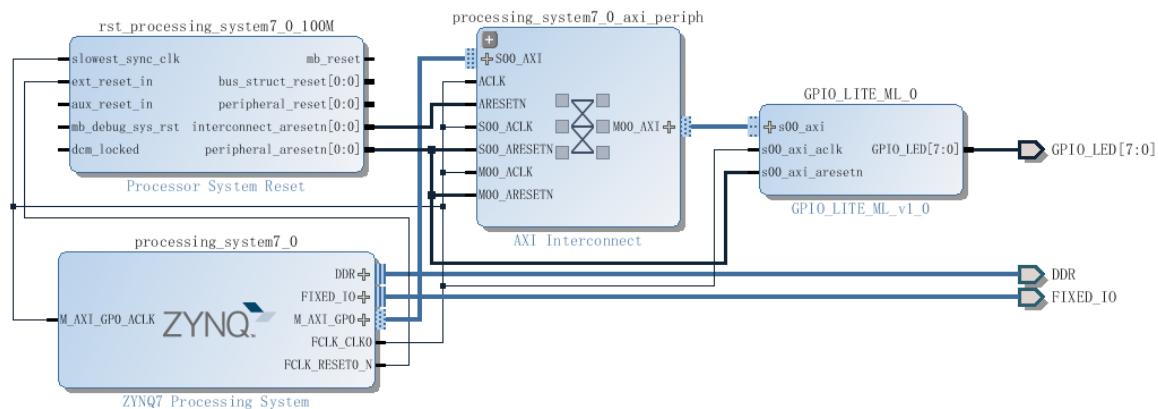
Step6：向 BD 文件中添加一个 ZYNQ Processing system, 根据自身硬件完成 IP 的配置。

Step7: 单击添加 IP 图标, 输入上一节我们自定义 IP 的模块名, 将其添加入 BD 文件中。



Step8: 直接点击 Run connection automation。

Step9: 选中 GPIO\_LED 端口, 按 Ctrl+T 引出端口, 整体硬件电路如下。



Step10: 右键单击 Block 文件, 文件选择 Generate the Output Products。

Step9: 右键单击 Block 文件, 选择 Create a HDL wrapper, 根据 Block 文件内容产生一个 HDL 的顶层文件, 并选择让 vivado 自动完成。

Step10: 添加约束文件, 在我们提供的源程序包的 DOC 文件夹下找到 XDC 文件夹, 将其中的约束文件添加到工程当中来。

Step11: 生成 bit 文件。

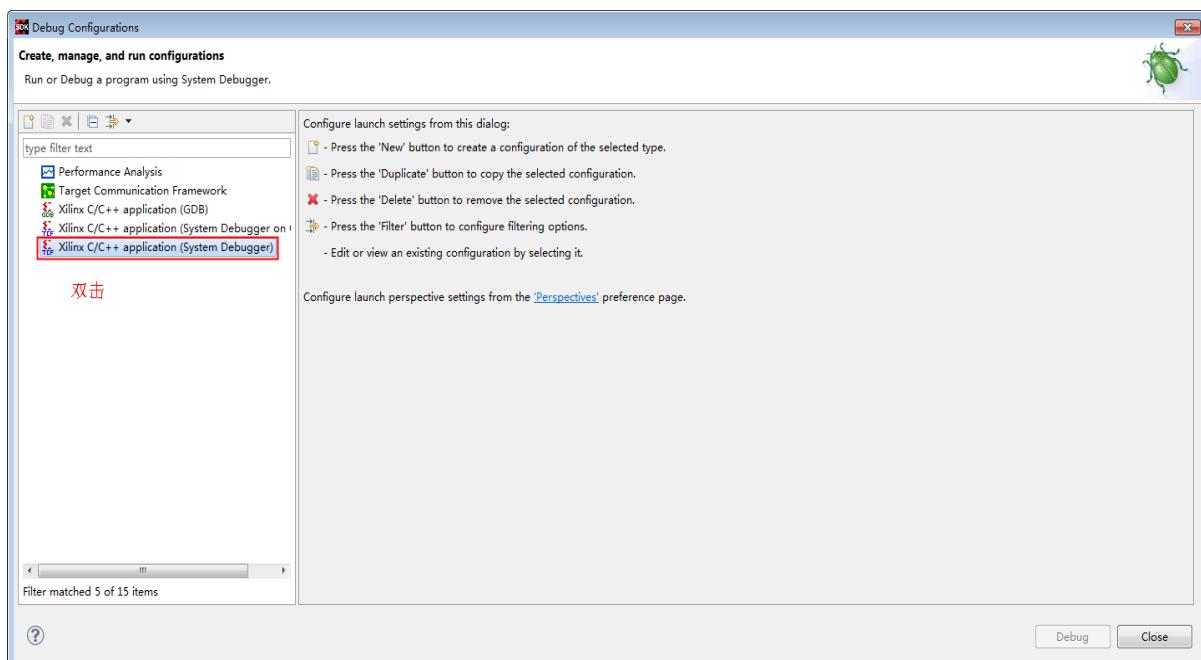
## 10.3 加载到 SDK

Step1: 导出硬件。

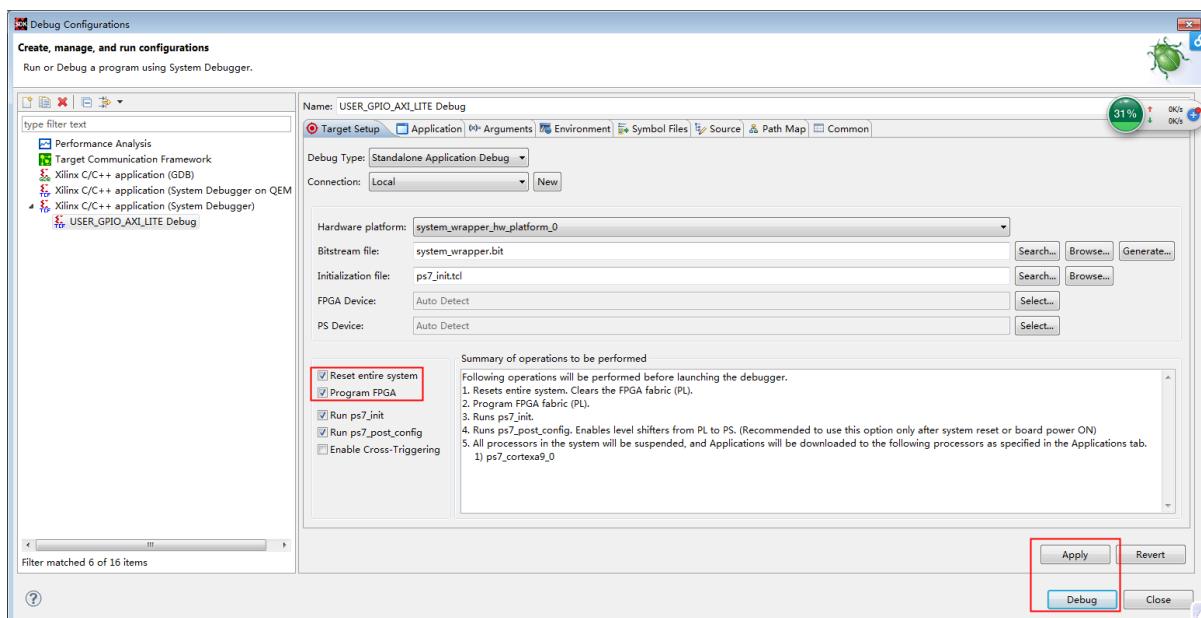
Step2: 新建一个空 SDK 工程, 并将我们提供的设计文件复制到工程当中来。

Step3: 右击工程, 选择 Debug as ->Debug configuration。

Step4: 选中 system Debugger, 双击创建一个系统调试。



### Step5：设置系统调试。



点击运行按钮开始运行程序，在开发板上四个 LED 循环流水操作。



## 10.4 程序分析

XGpio\_axi\_WriteReg() 函数实现的是向 AXI 的寄存器中写入数据，它的三个参数分别为地址、偏移量和数据。需要注意的是此处的偏移量，AXI 的相邻寄存器偏移量相差 4 个字节，默认 slv\_reg0 的偏移量是 0，因此，可以推导出 slv\_reg1, slv\_reg2 的偏移量分别为 4 和 8，本章中，我们只用到了 slv\_reg0，所以偏移量为 0。

## 10.4 本章小结

本章介绍了一种创建 AXI 总线高速接口的方法，在实际开发中，有非常重要的意义，大家可以根据这种方法，自行设计其他带 AXI 总线的 IP。

## CH11\_ZYNQ 软硬调试高级技巧

软件和硬件的完美结合才是 SOC 的优势和长处，那么开发 ZYNQ 就需要掌握软件和硬件开发的调试技巧，这样才能同时分析软件或者硬件的运行情况，找到问题，最终解决。那么本章将通过一个简单的例子带大家使用 vivado+SDK 进行系统的调试。

### 11.1 方案框架

这个实验中，我们将在上一章工程的基础上添加一个名为 MATH\_IP 的 Custom IP，并且添加 Mark Debug 观察 AXI4-Lite 总线上的工作情况，添加 VIO CORE 观察 MATH\_IP 的工作情况，添加 ILA CORE 观察 LED 的 PIN 脚输出情况。

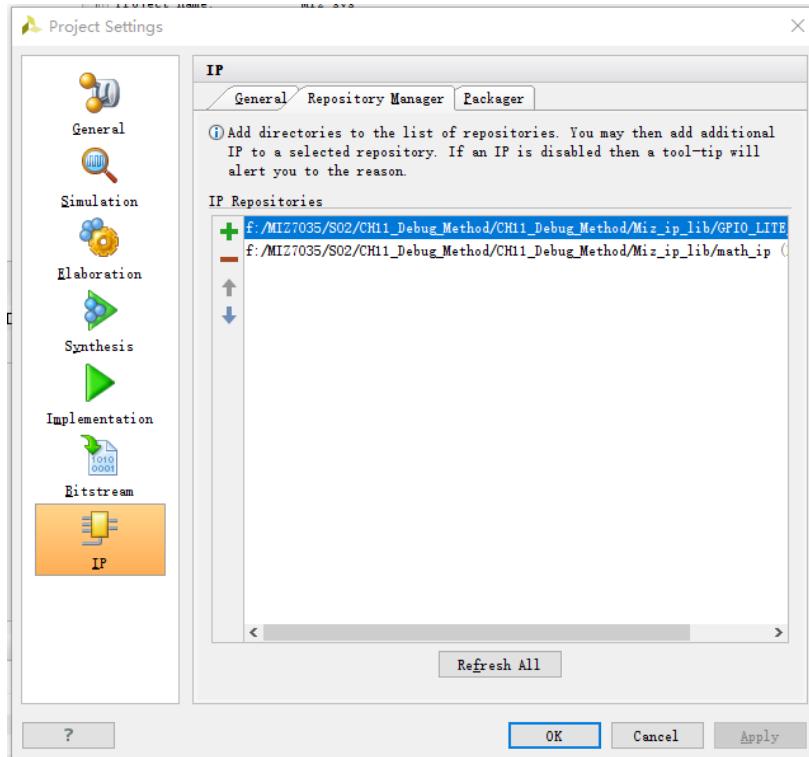
### 11.2 硬件工程搭建

Step1：做好备份后，直接打开上一章节的硬件工程。

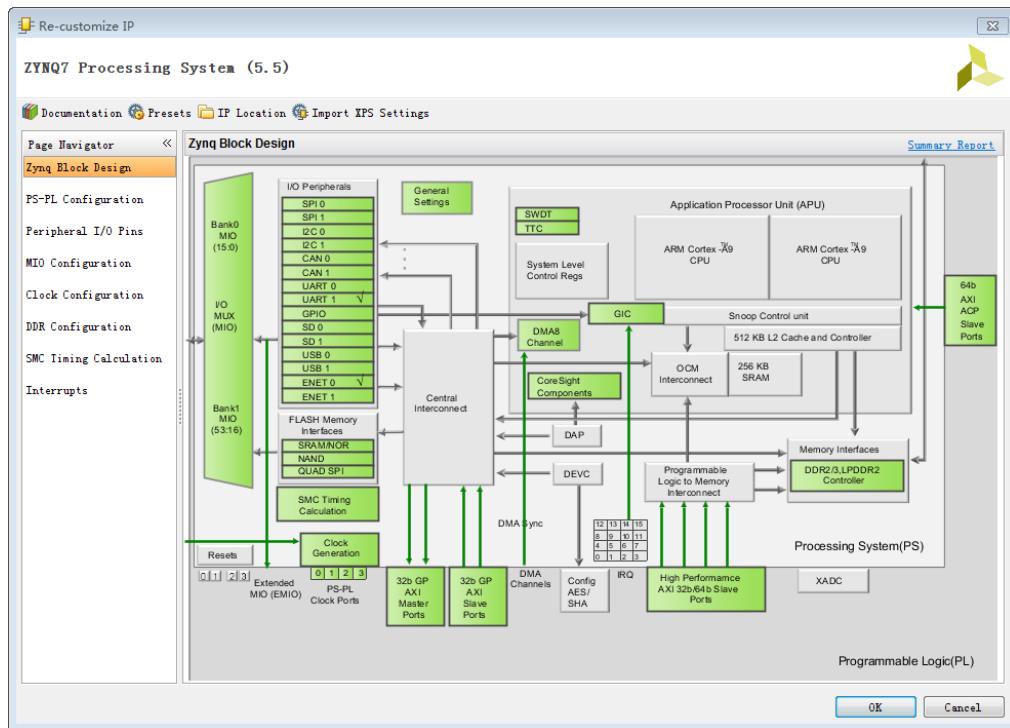
Step2：在 Project manager 区中单击 Project settings。

Step3：选择 IP 设置区中的 repository manager。

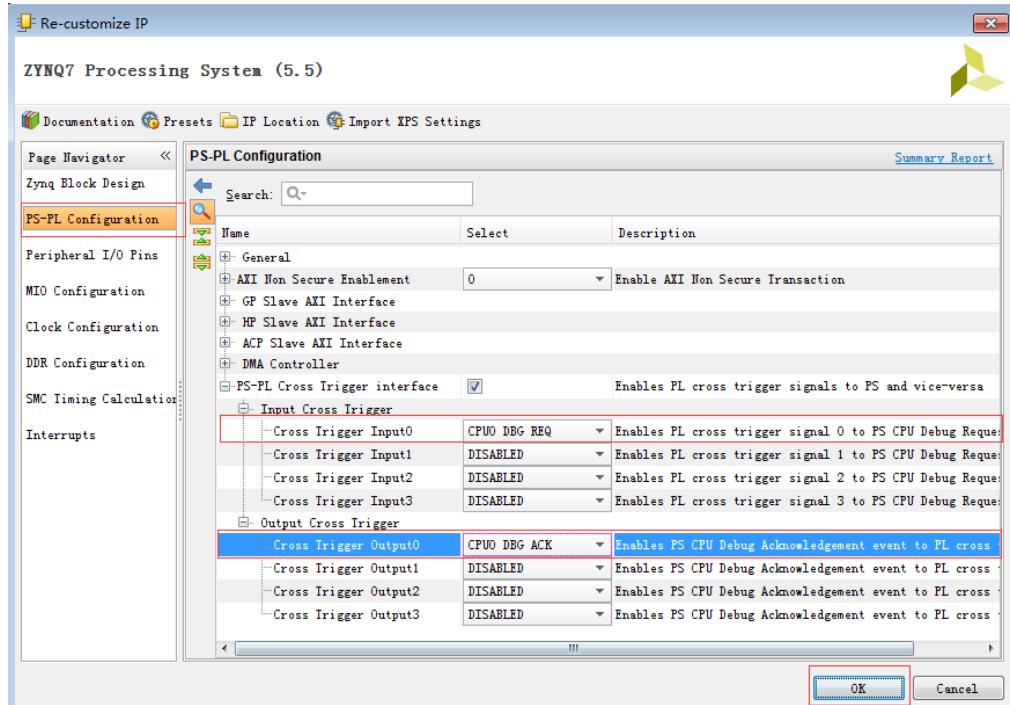
Step4：单击+号图标，将 math\_ip\_0 的路径添加进去（math\_ip 可在我们附带的第十一章程序文件夹中的 Miz\_ip\_lib 文件夹中找到），单击 OK。



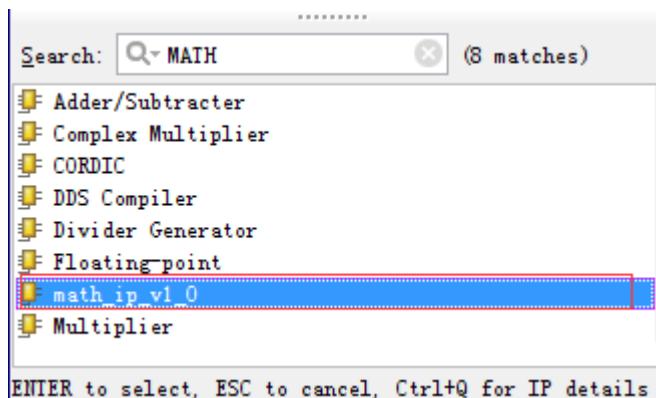
Step5：双击 ZYNQ processing System 图标，配置 IP。



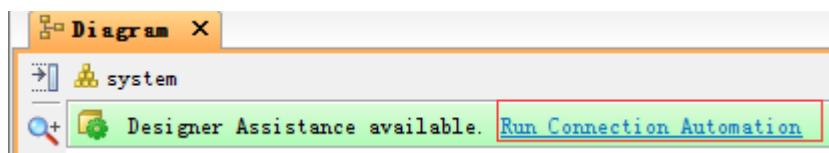
Step6: 展开 PS-PL Cross Trigger interface > Input Cross Trigger, Cross Trigger Input 0 设置为: CPU0 DBG REQ、Output Cross Trigger 设置为 CPU0 DBG ACK, 单击 OK 完成修改。



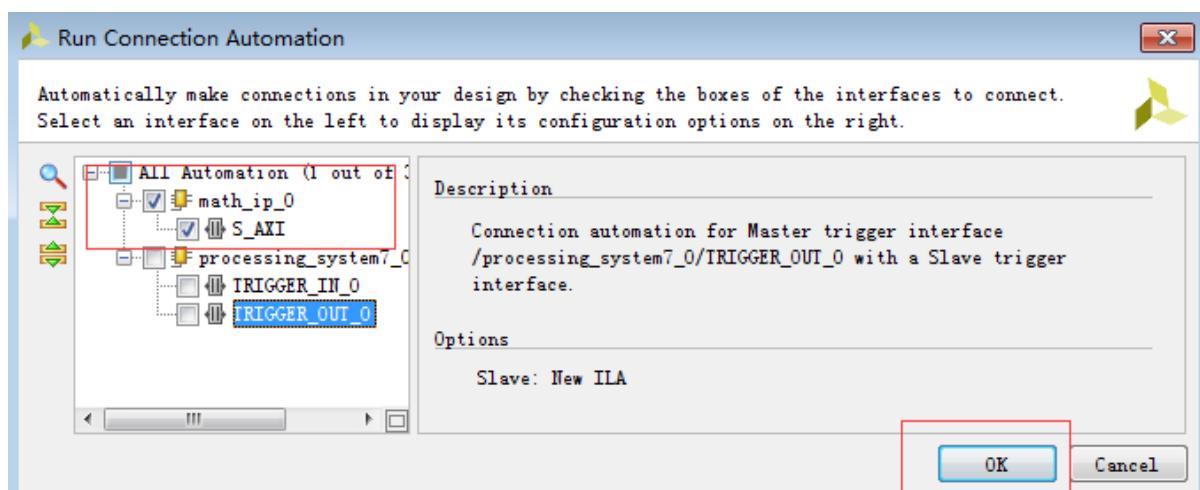
Step7: 单击 IP icon 搜索单词“math”之后双击添加 IPCORE。



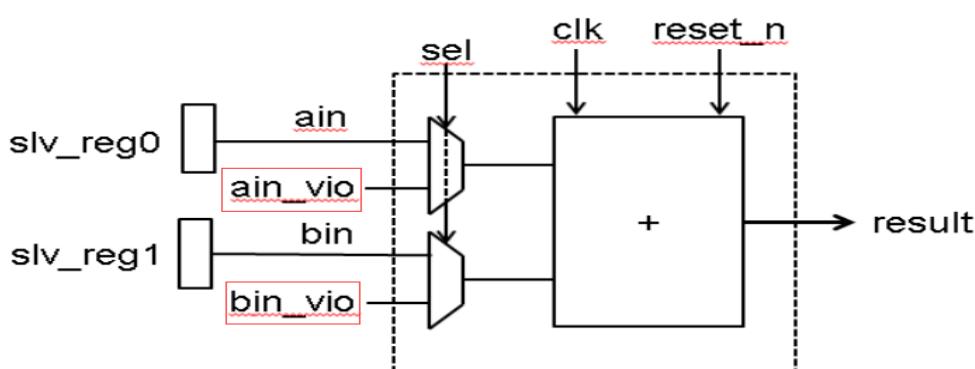
Step8: 单击 Click on Run Connection Automation。



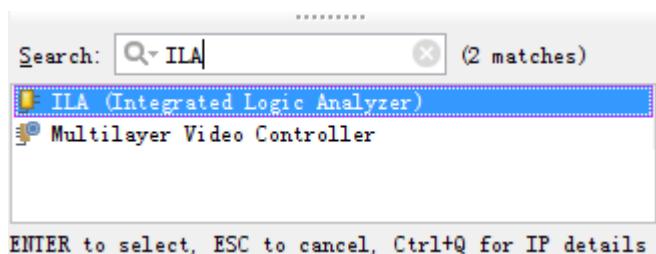
Step9: 勾选 math\_ip\_0 and S\_AXI 之后单击 OK。



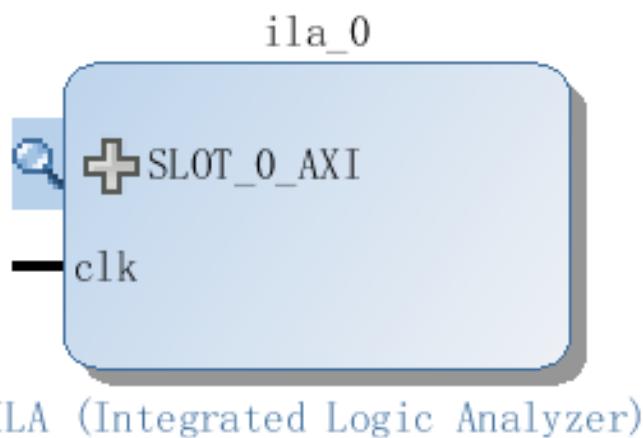
这个 math\_ip 实际上是一个简单的硬件加法器。虽然这个简单的加法器在这里没有实用意义,但是如果换成了硬件算法,那么就具备实用价值了。红色的方框内 ain\_vio 和 bin\_vio 是我们准备通过逻辑分析抓去的观察信号。



Step10: 单击 IP icon  添加 ila CORE

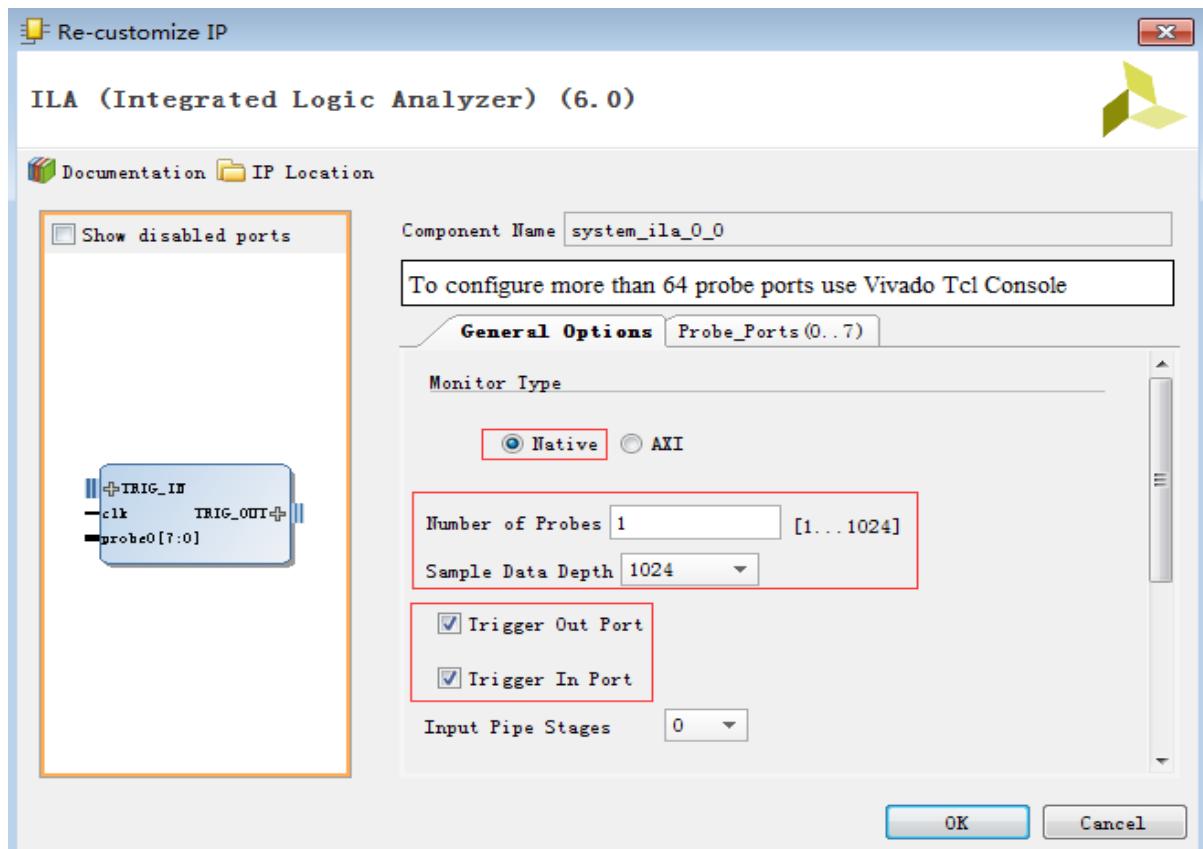


Step11: 双击打开 ILA CORE

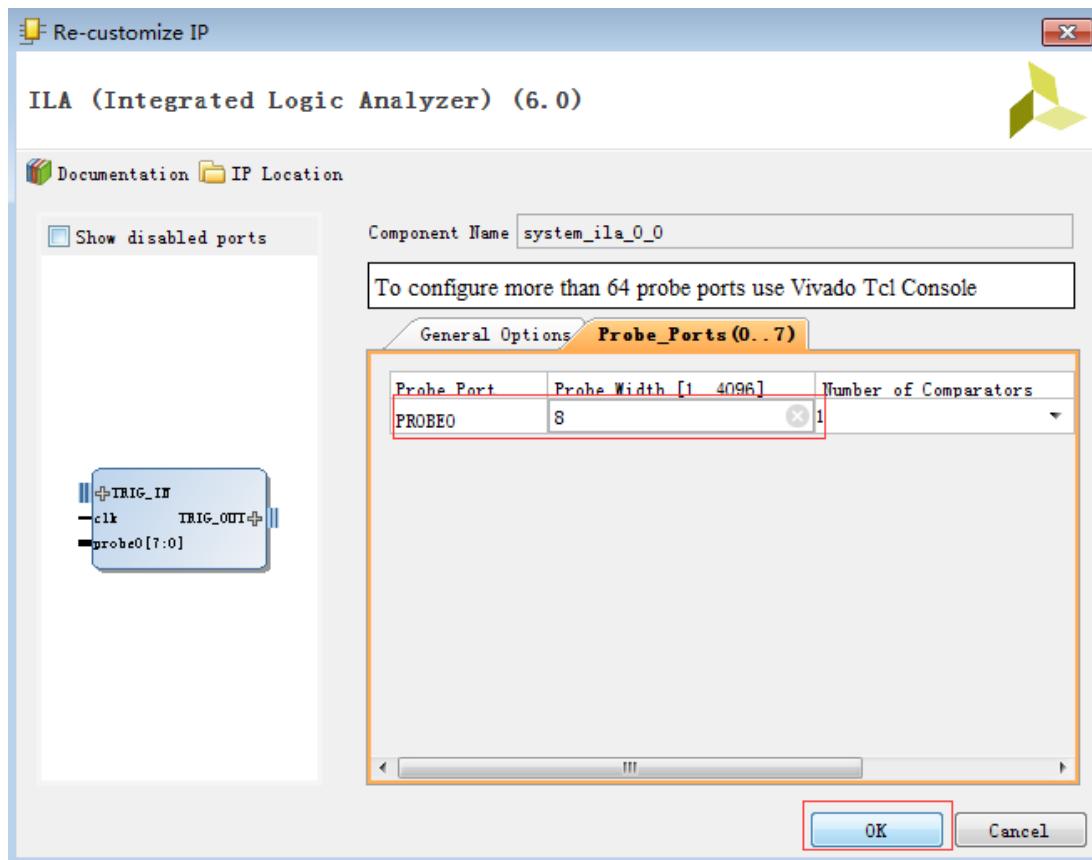


Step12: 双击打开 ILA CORE

General Options 设置如下



Probe\_Ports 设置如下,之后单击 OK

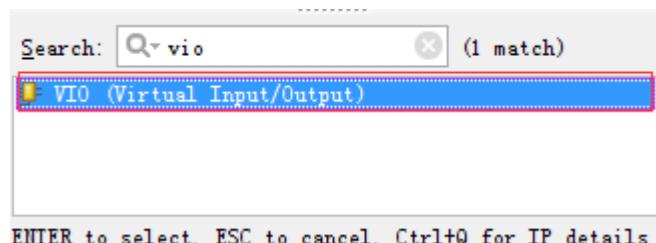


Step13: 连接 Probe0 到 GPIO\_LED。

Step14: 连接 CLK 接口到 FCLK\_CLK0 接口

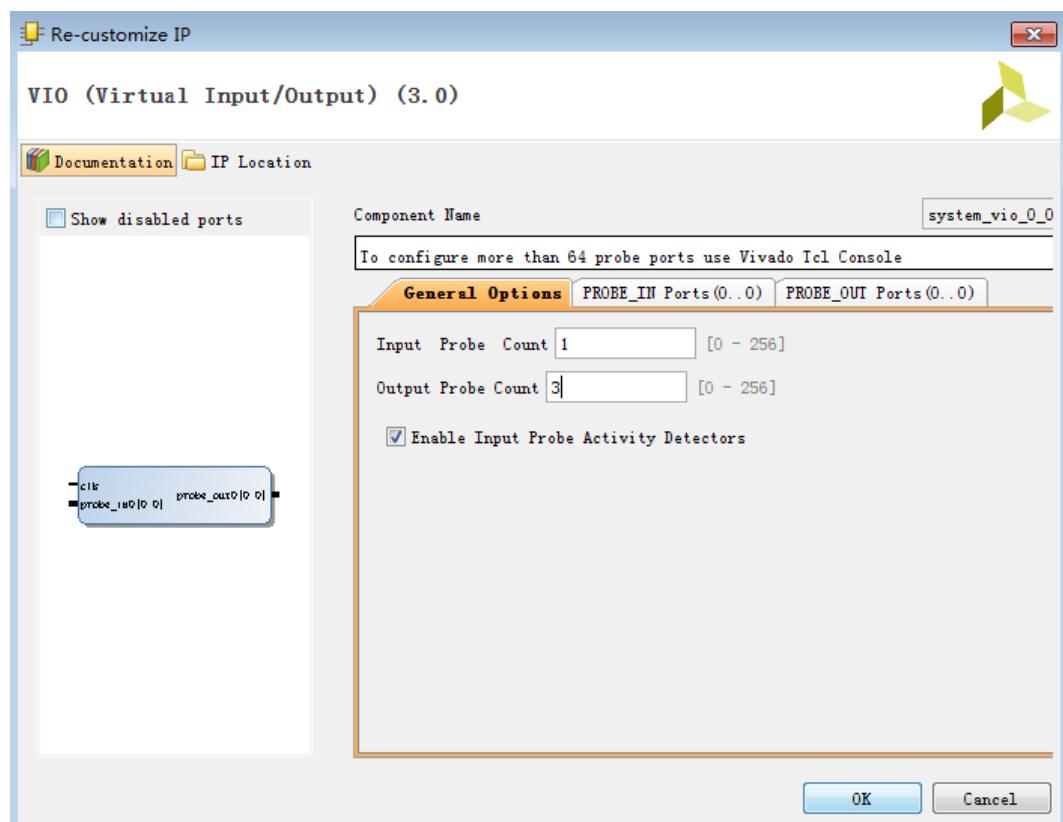
Step15: 连接 TRIGG\_IN 和 TRIGGER\_OUT\_0、TRIG\_OUT 和 TRIGGER\_IN\_0

Step16: 添加 IP icon  添加 vio。

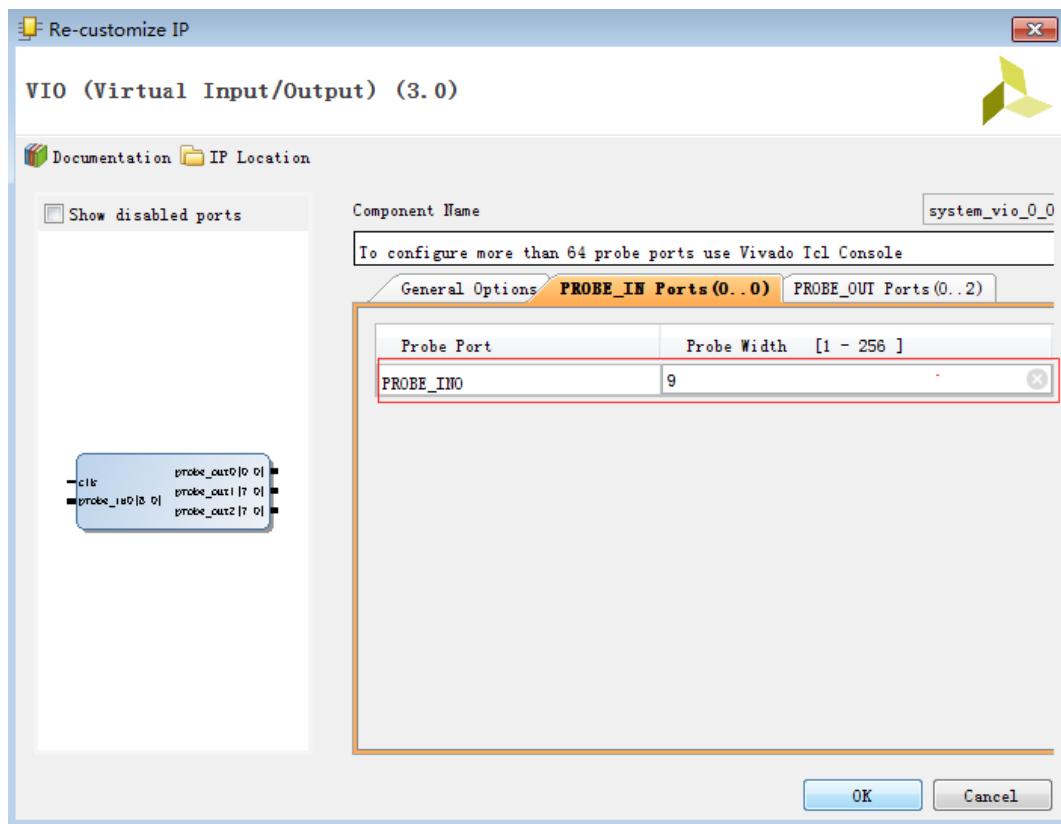


Step17: 双击 VIO core 修改参数

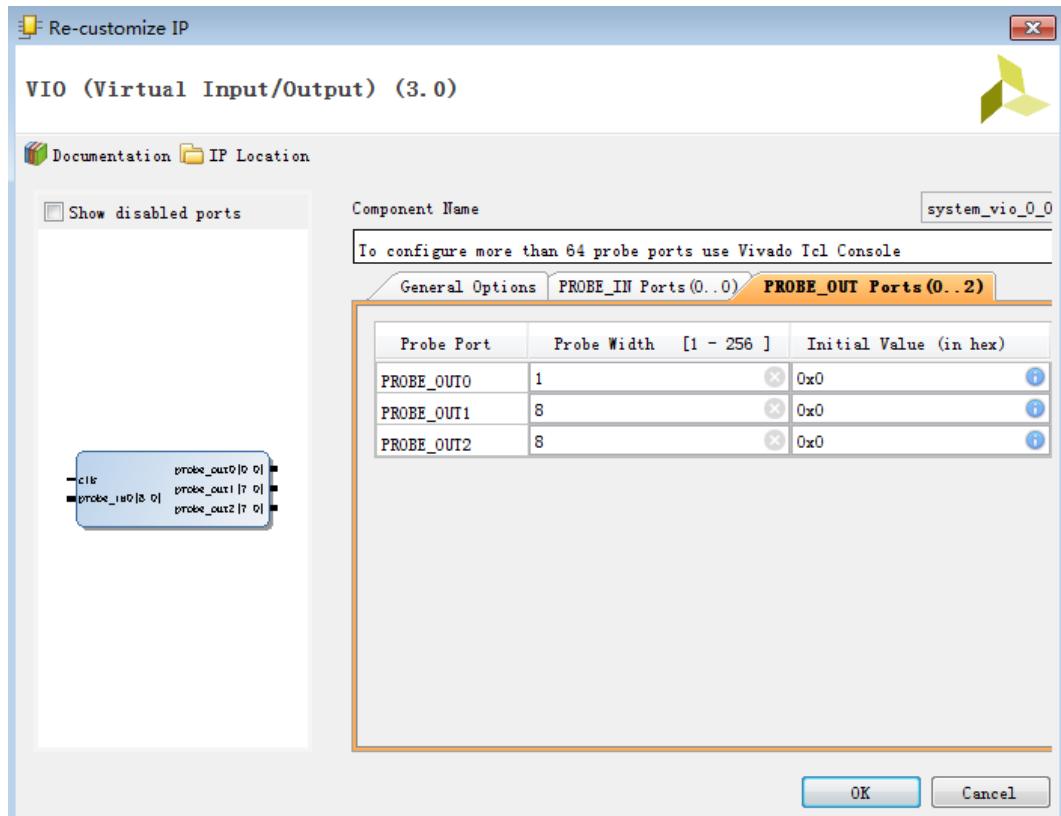
General Options 设置如下，输入 probe 为 1 输出为 3



Probe\_in 设置位宽为 9



Probe\_out0 设置位宽: 1; Probe\_out1 设置位宽: 8; Probe\_out2 设置位宽: 8;



Step18:连接

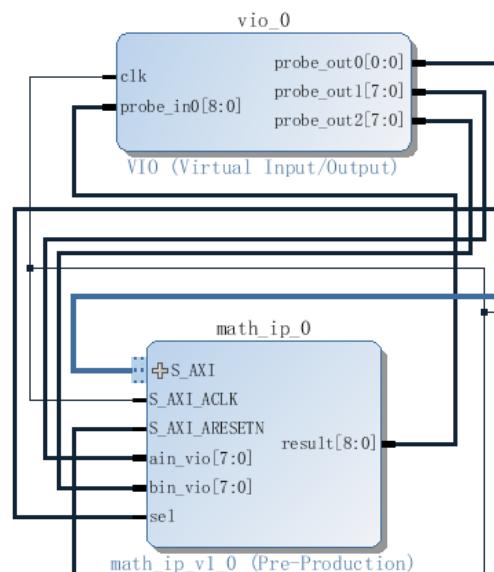
PROBE\_IN -> result

PROBE\_OUT0 -> sel

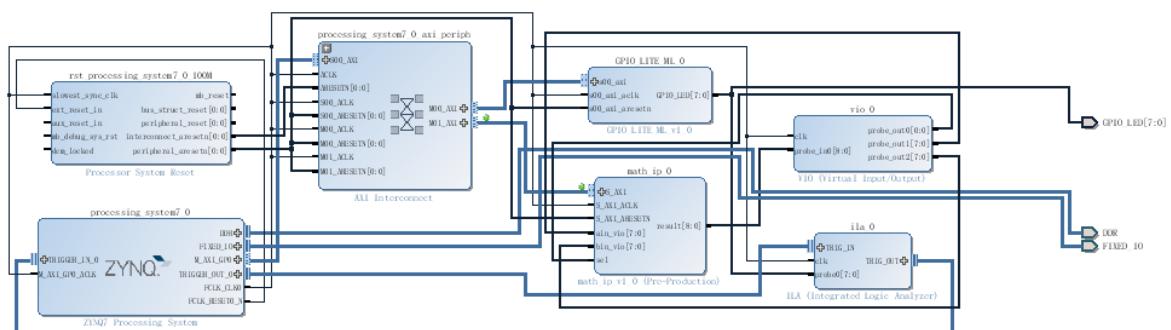
PROBE\_OUT1 -> ain\_vio

PROBE\_OUT2 -> bin\_vio

CLK-> FCLK\_CKLO

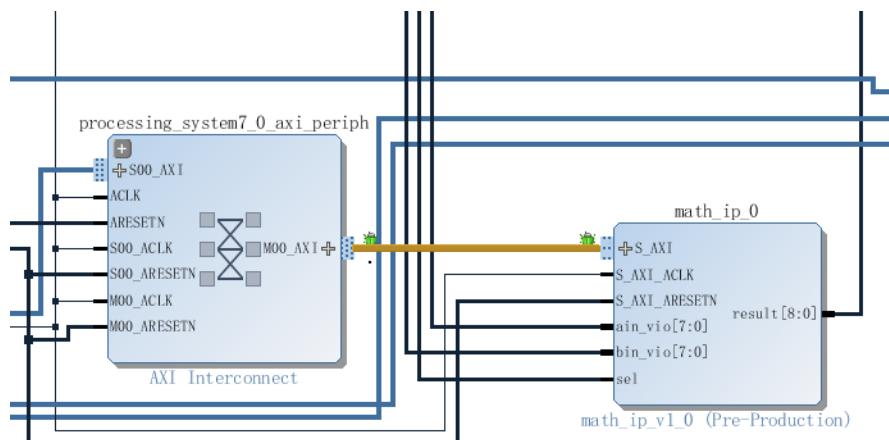


Step19: 连接好的系统整体电路。



Step20: 选中 AXI Interconnect 和 math\_0 CORE 之间的 S\_AXI 总线

Step21: 右击选择 Mark Debug



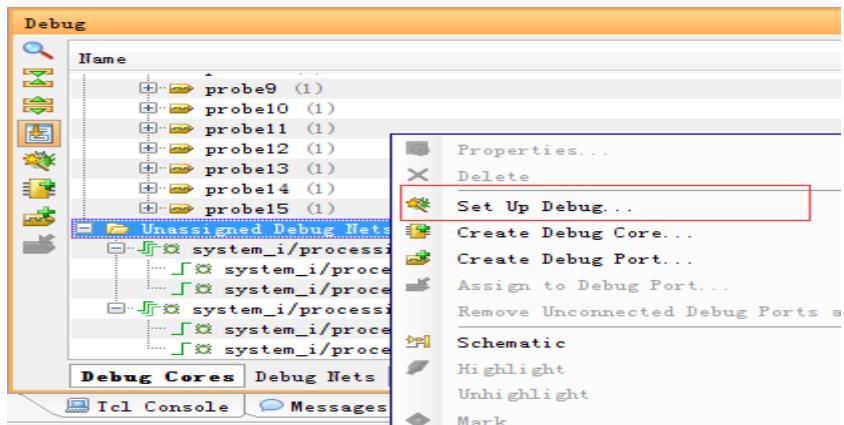
Step22:接下来依然是,右键单击 Block 文件,文件选择 Generate the Output Products。

Step23:继续右键单击 Block 文件, 选择 Create a HDL wrapper, 根据 Block 文件内容产生一个 HDL 的顶层文件, 并选择让 vivado 自动完成。

Step24:单击 Run Synthesis,如果有 Save 对话框弹出选择保存。

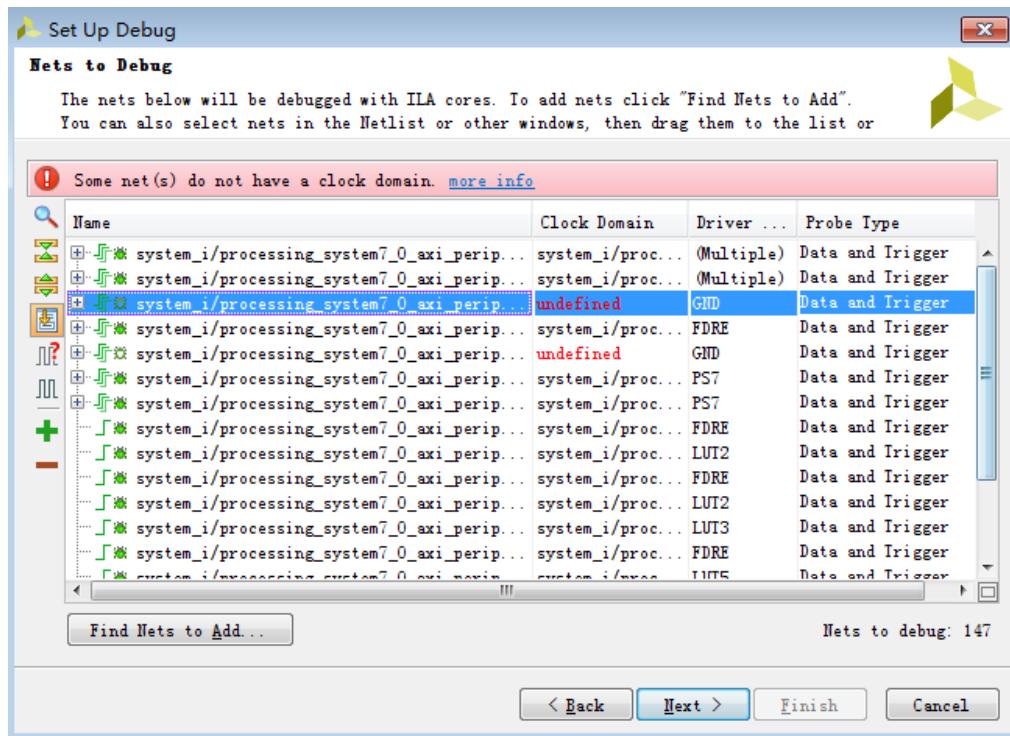
Step25:综合结束后选择 Synthesized Design option 单击 OK。

Step26:在如下对话框中找到 Unassigned debug nets(如果对话框没有出现选择 菜单->Window > Debug)



Step27:右击 Unassigned Debug Nets 选择 Set up Debug... 之后单击 Next

Step28:删除红色错误的信号然后单击 Next 到结束



Step29:生成 Bit 文件。

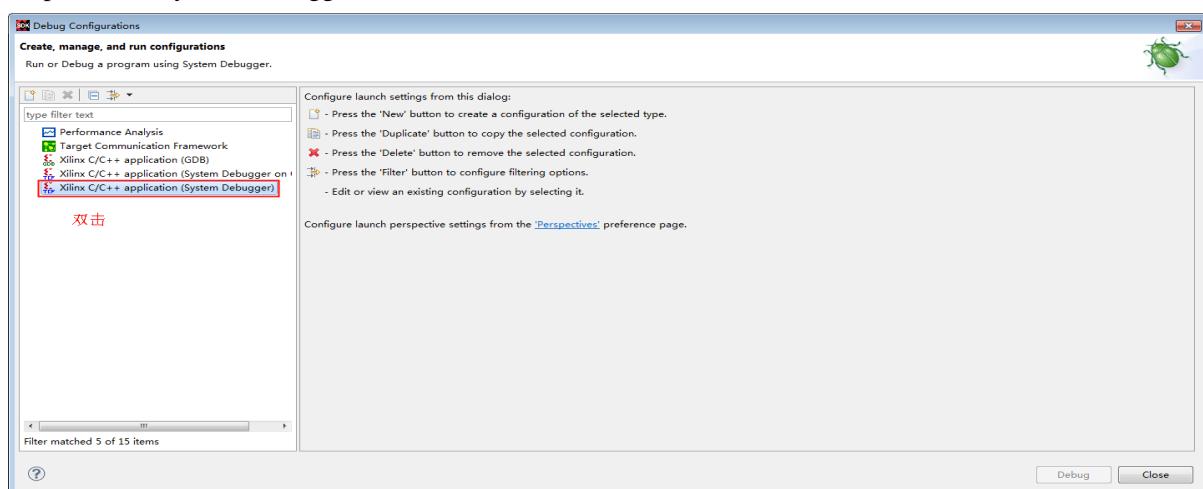
### 11.3 加载到 SDK

Step1: 导出硬件。

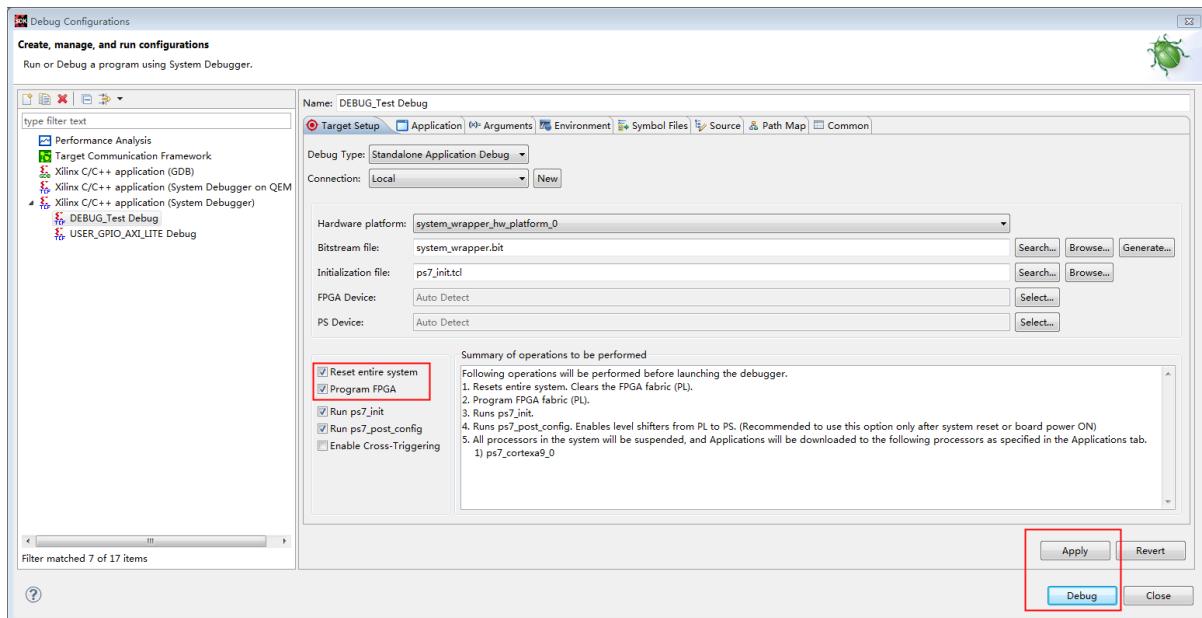
Step2: 新建一个空 SDK 工程，并将我们提供的设计文件复制到工程当中来。

Step3: 右击工程，选择 Debug as ->Debug configuration。

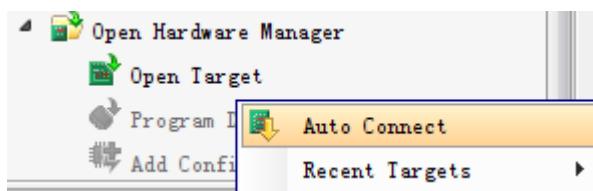
Step4: 选中 system Debugger,双击创建一个系统调试。



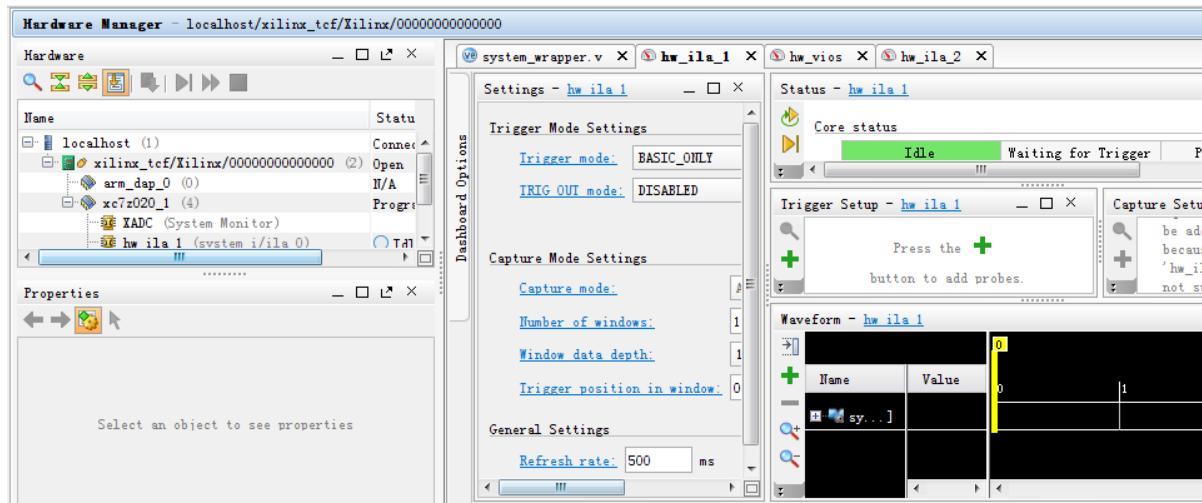
Step5: 设置系统调试。



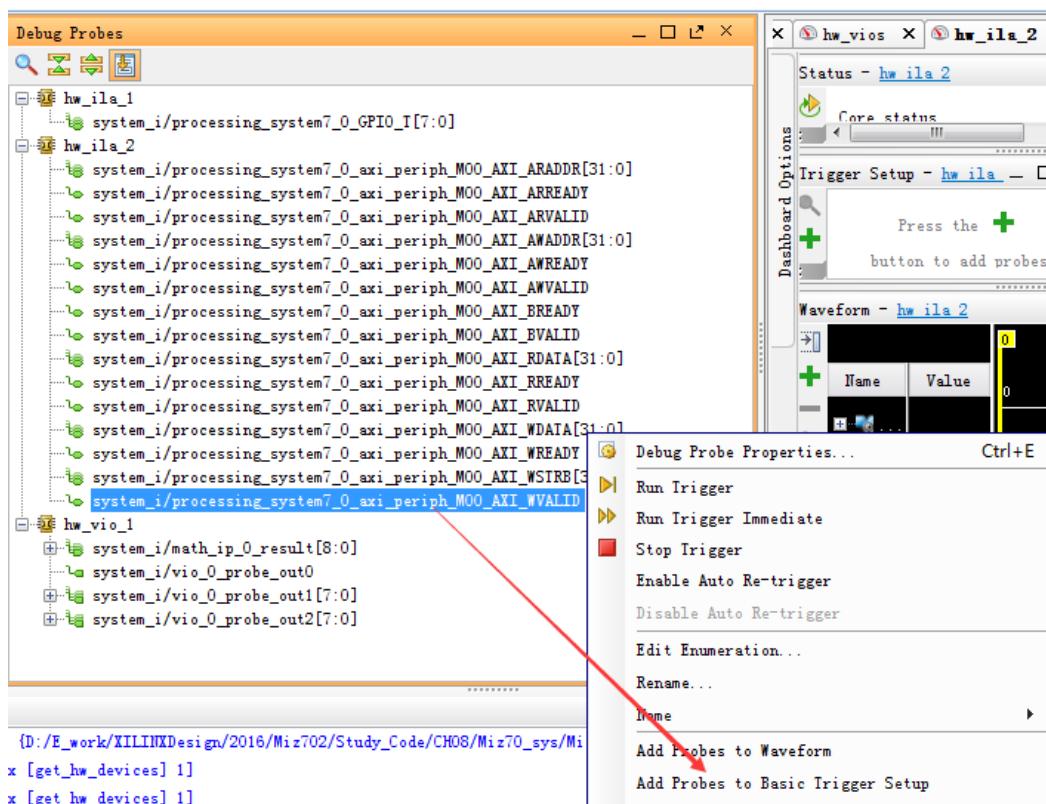
Step6:回到VIVADO 单击 Open Target->Auto Connect



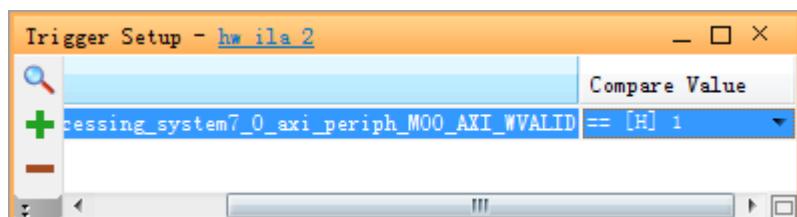
Step7:加载完成后的界面



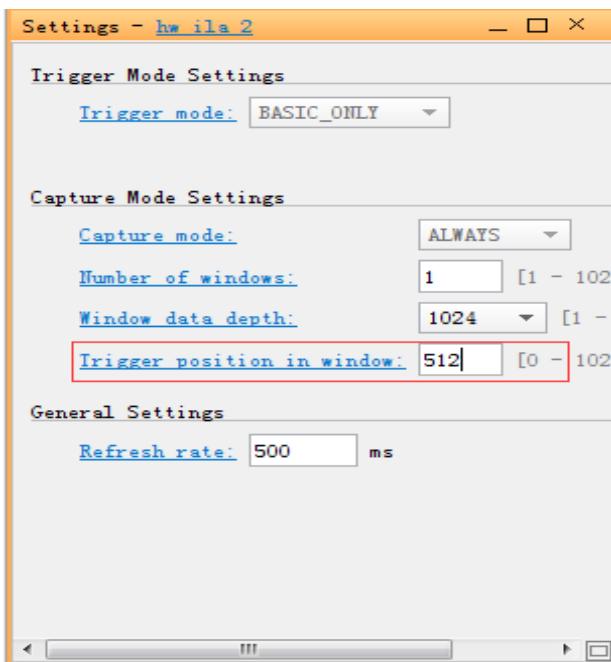
Step8:选择菜单->window->Debugprobes 选择 AXI\_WVALID 做为触发信号



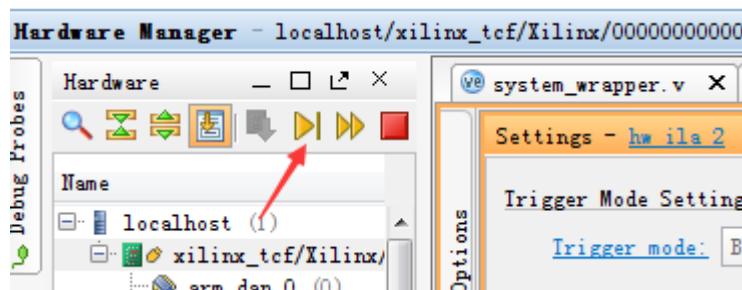
Step9: 设置触发条件为 1



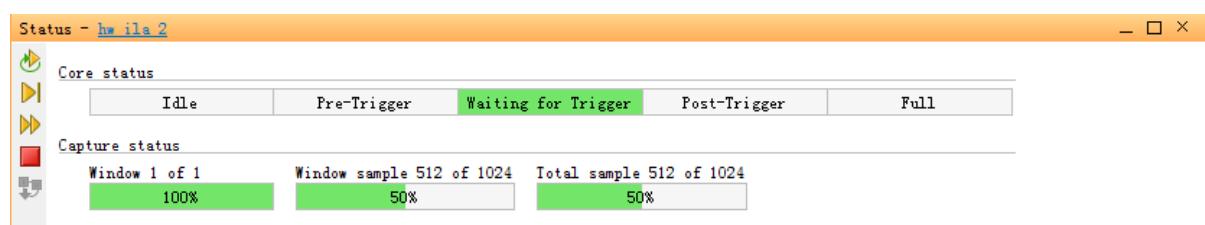
Step10: 设置触发位置为 512



Step11:单击箭头所指向启动触发

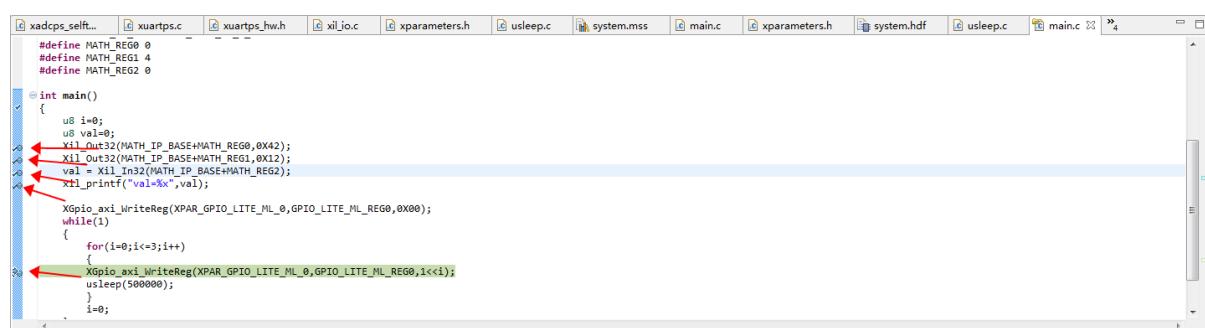


Step12:进入等待触发状态

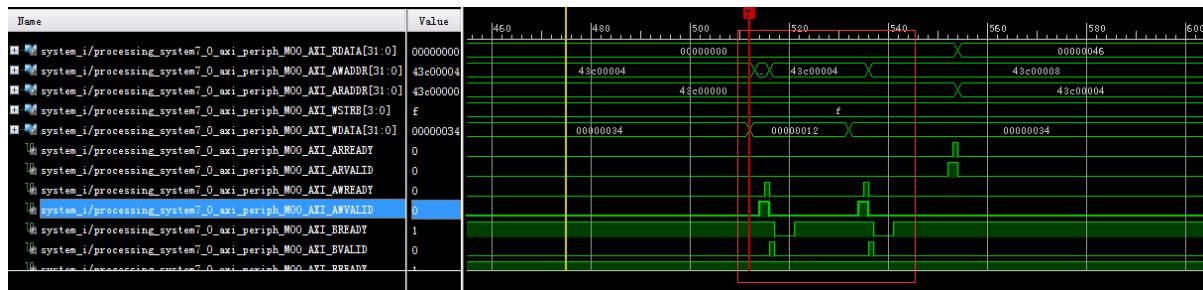


Step13: 打开系统自带的串口调试软件。

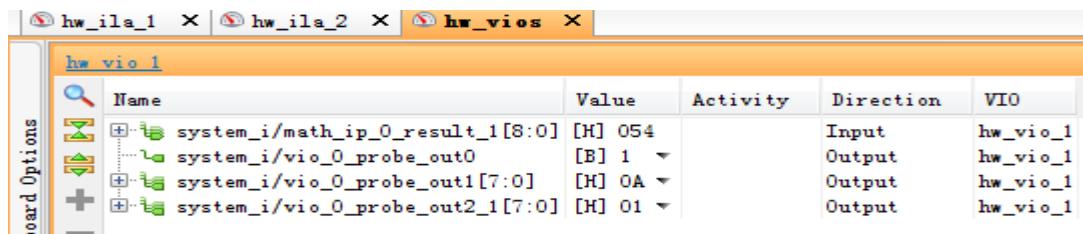
Step14: 在以下位置加入断点（在图中位置双击即可加入断点），方便调试。



Step15: 单击运行 后 VIVADO HW\_ILA2 窗口采集到波形输出, 可以看到 AXI 总线的工作时序。

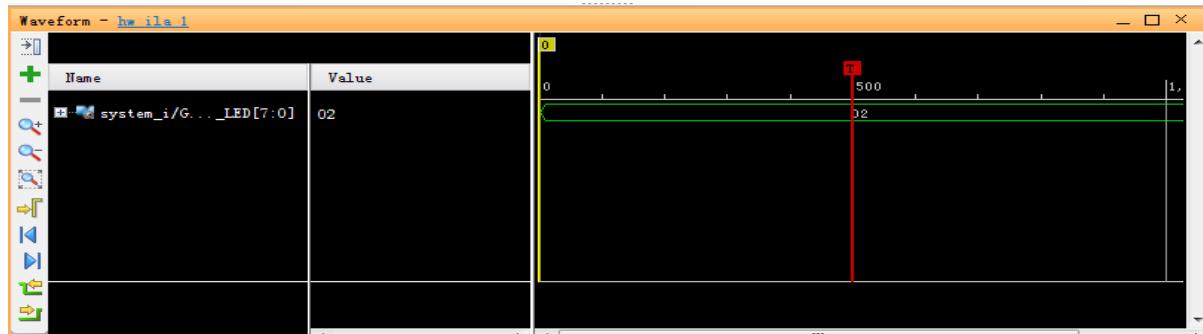


Step16: 同时可以观察到 VIO 核采集到的数据



Step17: 当再次单击 后控制台输出 0X54

Step18: HW\_ILA1 窗口采集到的数据是 GPIO\_LED 的值为 0x02, 同时可观察到开发板上的 LED2 亮起。



## 11.4 本章小结

在这个实验中, 笔者添加了一个用户自定义的 IP CORE 并且通过使用 VIO CORE 观察其数据。通过 ILA CORE 观察 AXI 总线的通信时序情况, 以及 EMIO 的输出情况。其中难点就是 SDK 和 VIVAOD 的联合调试。

## CH12\_AXI\_Lite 总线详解

### 12.1 前言

ZYNQ拥有ARM+FPGA这个神奇的架构，那么ARM和FPGA究竟是如何进行通信的呢？本章通过剖析AXI总线源码，来一探其中的秘密。

### 12.2 AXI 总线与 ZYNQ 的关系

AXI(Advanced eXtensible Interface)本是由ARM公司提出的一种总线协议，Xilinx从6系列的FPGA开始对AXI总线提供支持，此时AXI已经发展到了AXI4这个版本，所以当你用到Xilinx的软件的时候看到的都是“AXI4”的IP，如Vivado打包一个AXI IP的时候，看到的都是Create a new AXI4 peripheral。

到了ZYNQ就更不必说了，AXI总线更是应用广泛，双击查看ZYNQ的IP核的内部配置，随处可见AXI的身影。

### 12.3 AXI 总线和 AXI 接口以及 AXI 协议

总线、接口和协议，这三个词常常被联系在一起，但是我们心里要明白他们的区别。

总线是一组传输通道，是各种逻辑器件构成的传输数据的通道，一般由数据线、地址线、控制线等构成。接口是一种连接标准，又常常被称之为物理接口。

协议就是传输数据的规则。

#### 12.3.1 AXI 总线概述

在ZYNQ中有支持三种AXI总线，拥有三种AXI接口，当然用的都是AXI协议。其中三种AXI总线分别为：

AXI4：(For high-performance memory-mapped requirements.) 主要面向高性能地址映射通信的需求，是面向地址映射的接口，允许最大256轮的数据突发传输；

AXI4-Lite：(For simple, low-throughput memory-mapped communication) 是一个轻量级的地址映射单次传输接口，占用很少的逻辑单元。

AXI4-Stream：(For high-speed streaming data.) 面向高速流数据传输；去掉了地址项，允许无限制的数据突发传输规模。

首先说AXI4总线和AXI4-Lite总线具有相同的组成部分：

- (1) 读地址通道，包含ARVALID, ARADDR, ARREADY信号；
- (2) 读数据通道，包含RVALID, RDATA, RREADY, RRESP信号；
- (3) 写地址通道，包含AWVALID, AWADDR, AWREADY信号；
- (4) 写数据通道，包含WVALID, WDATA, WSTRB, WREADY信号；
- (5) 写应答通道，包含BVALID, BRESP, BREADY信号；
- (6) 系统通道，包含：ACLK, ARESETN信号。

AXI4总线和AXI4-Lite总线的信号也有他的命名特点：

读地址信号都是以AR开头 (A: address; R: read)

写地址信号都是以AW开头 (A: address; W: write)

读数据信号都是以R开头 (R: read)

写数据信号都是以W开头 (W: write)

应答型号都是以B开头 (B: back (answer back) )

了解到总线的组成部分以及命名特点，那么在后续的实验中您将逐渐看到他们的身影。每个信号的作用暂停不表，放在后面一一介绍。

而AXI4-Stream总线的组成有：

- (1) ACLK信号：总线时钟，上升沿有效；
- (2) ARESETN信号：总线复位，低电平有效
- (3) TREADY信号：从机告诉主机做好传输准备；
- (4) TDATA信号：数据，可选宽度32, 64, 128, 256bit
- (5) TSTRB信号：每一bit对应TDATA的一个有效字节，宽度为TDATA/8
- (6) TLAST信号：主机告诉从机该次传输为突发传输的结尾；
- (7) TVALID信号：主机告诉从机数据本次传输有效；
- (8) TUSER信号：用户定义信号，宽度为128bit。

对于AXI4-Stream总线命名而言，除了总线时钟和总线复位，其他的信号线都是以T字母开头，后面跟上一个有意义的单词，看清这一点后，能帮助读者记忆每个信号线的意义。如TVALID = T+单词Valid（有效），那么读者就应该立刻反应该信号的作用。每个信号的具体作用，在后面分析源码时再做分析

### 12.3.2 AXI 接口介绍

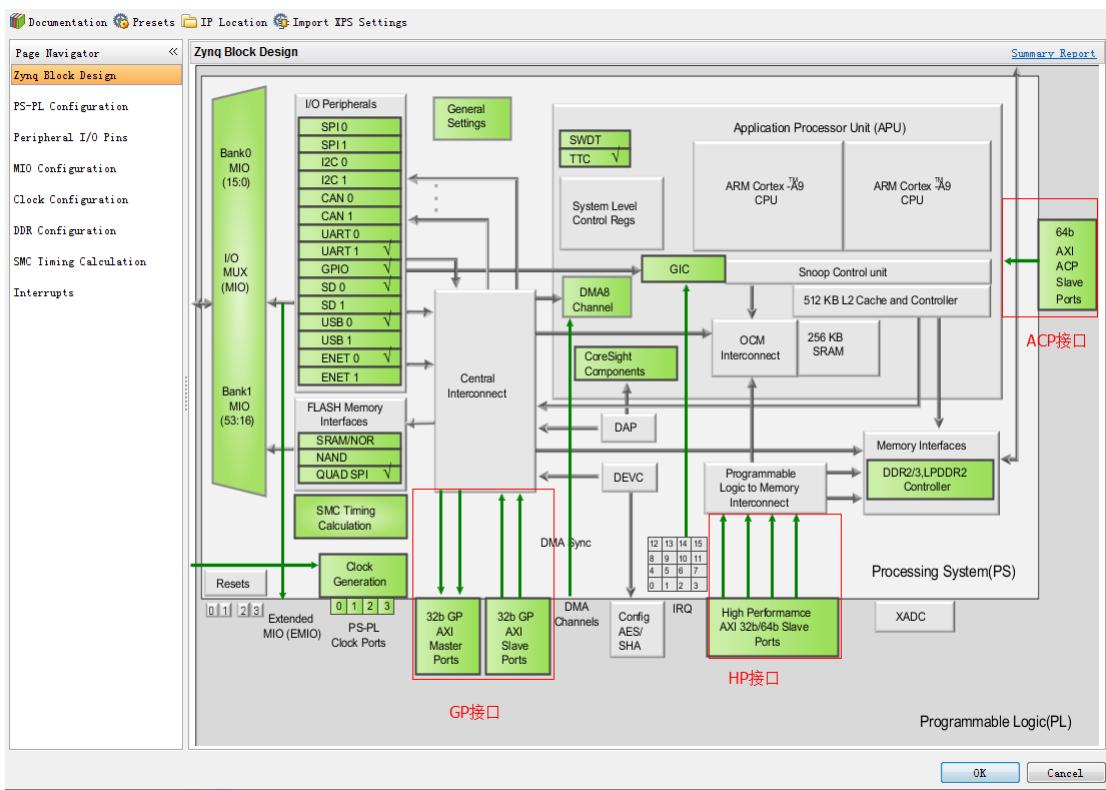
三种AXI接口分别是：

AXI-GP接口（4个）：是通用的AXI接口，包括两个32位主设备接口和两个32位从设备接口，用过改接口可以访问PS中的片内外设。

AXI-HP接口（4个）：是高性能/带宽的标准的接口，PL模块作为主设备连接（从下图中箭头可以看出）。主要用于PL访问PS上的存储器（DDR和On-Chip RAM）

AXI-ACP接口（1个）：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。

我们可以双击查看ZYNQ的IP核的内部配置，就能发现上述的三种接口，图中已用红色方框标记出来，我们可以清楚的看出接口连接与总线的走向：

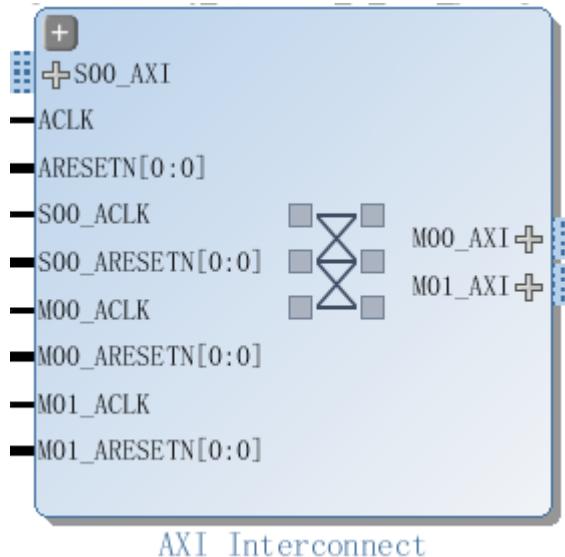


### 12.3.3 AXI 协议概述

讲到协议不可能说是撇开总线单讲协议，因为协议的制定也是要建立在总线构成之上的。虽然说AXI4, AXI4-Lite, AXI4-Stream都是AXI4协议，但是各自细节上还是不同的。

总的来说，AXI总线协议的两端可以分为分为主（master）、从（slave）两端，他们之间一般需要通过一个AXI Interconnect相连接，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。当我们添加了zynq以及带AXI的IP后再进行自动连线时vivado会自动帮我们添加上这个IP，大家应该是不陌生了。

AXI Interconnect的主要作用是，当存在多个主机以及从机器时，AXI Interconnect负责将它们联系并管理起来。由于AXI支持乱序发送，乱序发送需要主机的ID信号支撑，而不同的主机发送的ID可能相同，而AXI Interconnect解决了这一问题，他会对不同主机的ID信号进行处理让ID变得唯一。

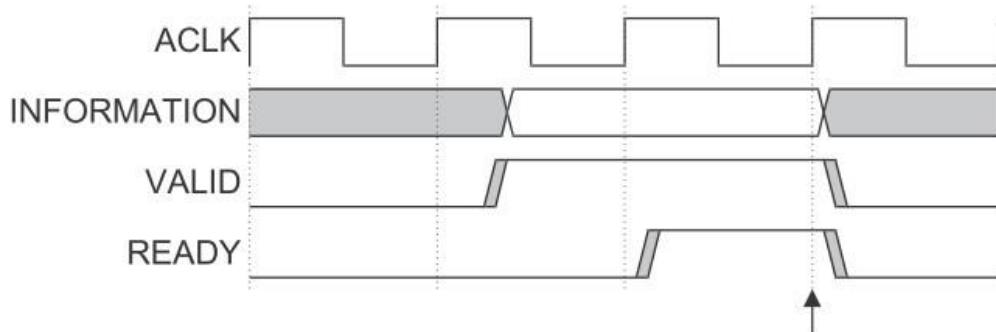


AXI协议将读地址通道，读数据通道，写地址通道，写数据通道，写响应通道分开，各自通道都有自己的握手协议。每个通道互不干扰却又彼此依赖。这也是AXI高效的原因之一。

#### 12.3.4 AXI 协议之握手协议

AXI4 所采用的是一种 READY, VALID 握手通信机制，简单来说主从双方进行数据通信前，有一个握手的过程。传输源产生 VLAID 信号来指明何时数据或控制信息有效。而目地源产生 READY 信号来指明已经准备好接受数据或控制信息。传输发生在 VALID 和 READY 信号同时为高的时候。VALID 和 READY 信号的出现有三种关系。

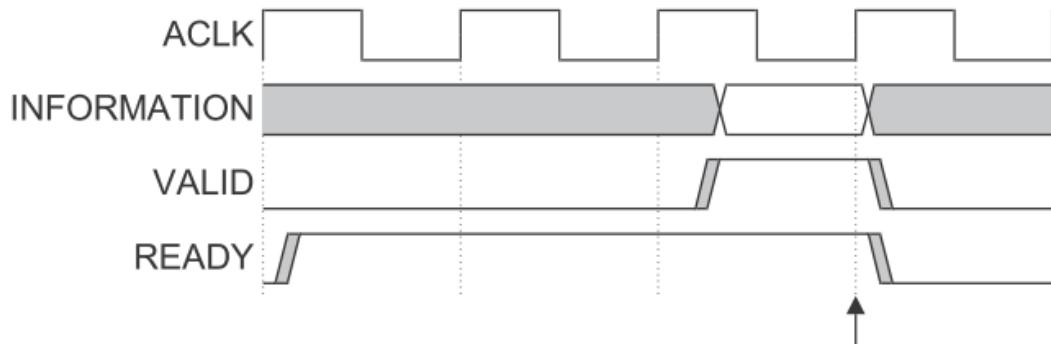
(1) VALID 先变高 READY 后变高。时序图如下：



**Figure 3-1 VALID before READY handshake**

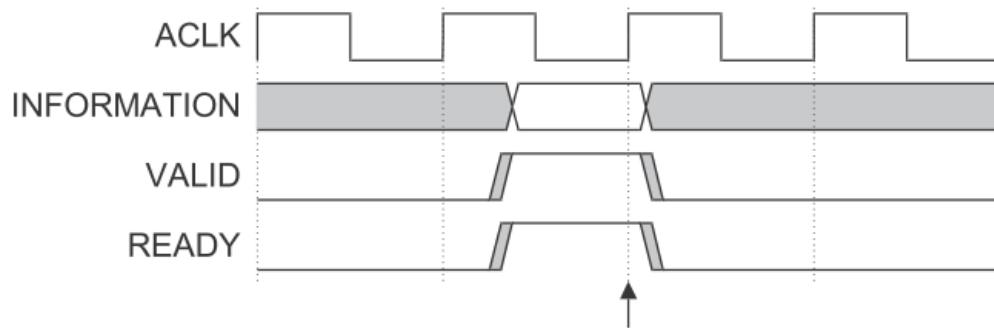
在箭头处信息传输发生。

(2) READY 先变高 VALID 后变高。时序图如下：



同样在箭头处信息传输发生。

(3) VALID 和 READY 信号同时变高。时序图如下：



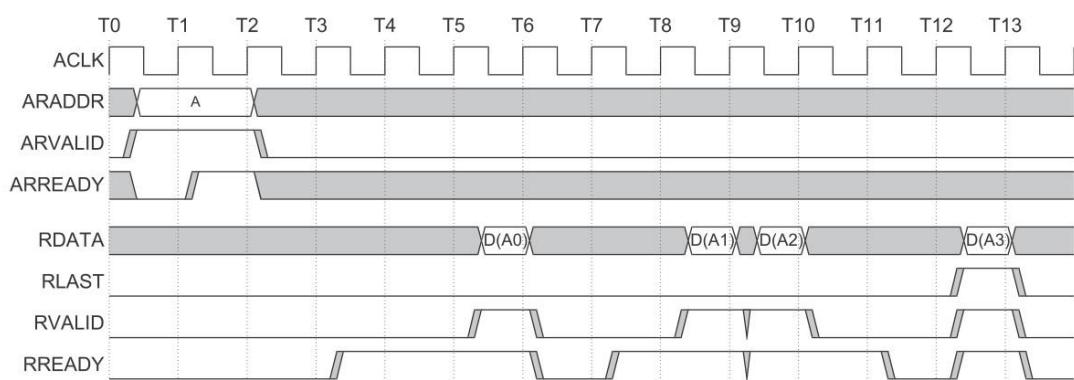
**Figure 3-3 VALID with READY handshake**

在这种情况下，信息传输立马发生，如图箭头处指明信息传输发生。

需要强调的是，AXI的五个通道，每个通道都有握手机制，接下来我们就来分析一下AXI-Lite的源码来更深入的了解AXI机制。

### 12.3.5 突发式读写

1、突发式读的时序图如下：



**Figure 1-4 Read burst**

当地址出现在地址总线后，传输的数据将出现在读数据通道上。设备保持 VALID 为低直到读数据有效。为了表明一次突发式读写的完成，设备用 RLAST 信号来表示最后一个被传输的数据。

2、突发式写时序图如下：

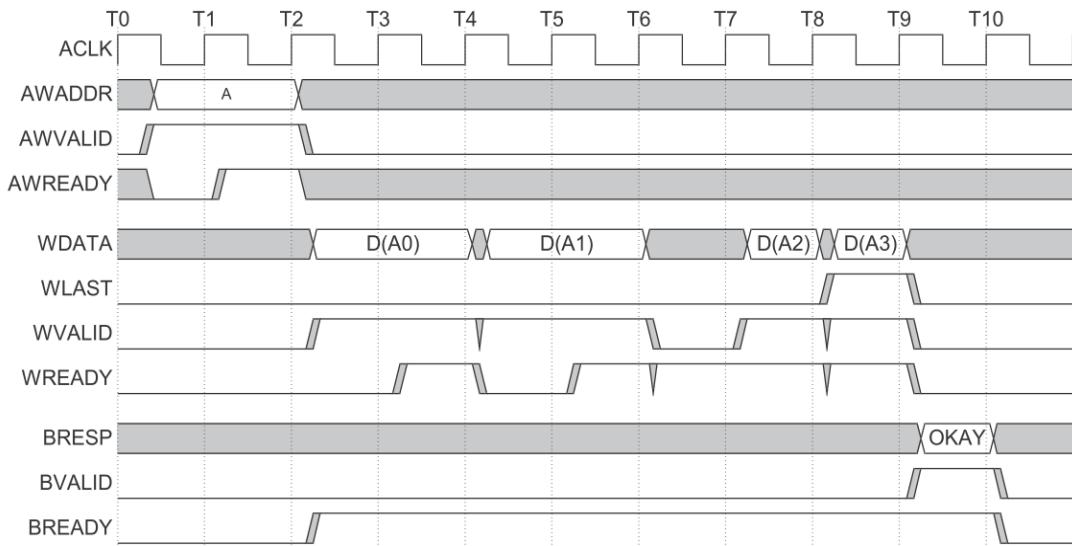


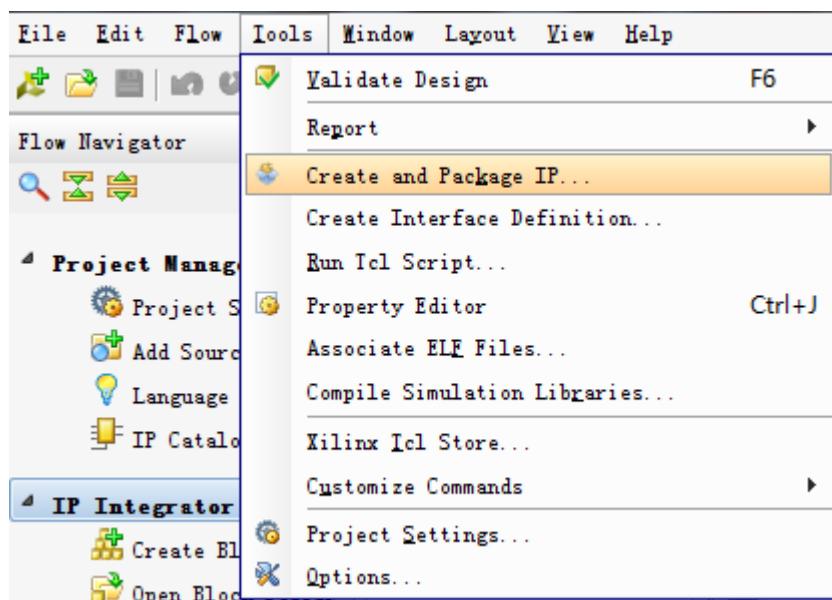
Figure 1-6 Write burst

这一过程的开始时，主机发送地址和控制信息到写地址通道中，然后主机发送每一个写数据到写数据通道中。当主机发送最后一个数据时，WLAST 信号就变为高。当设备接收完所有数据之后他将一个写响应发送回主机来表明写事务完成。

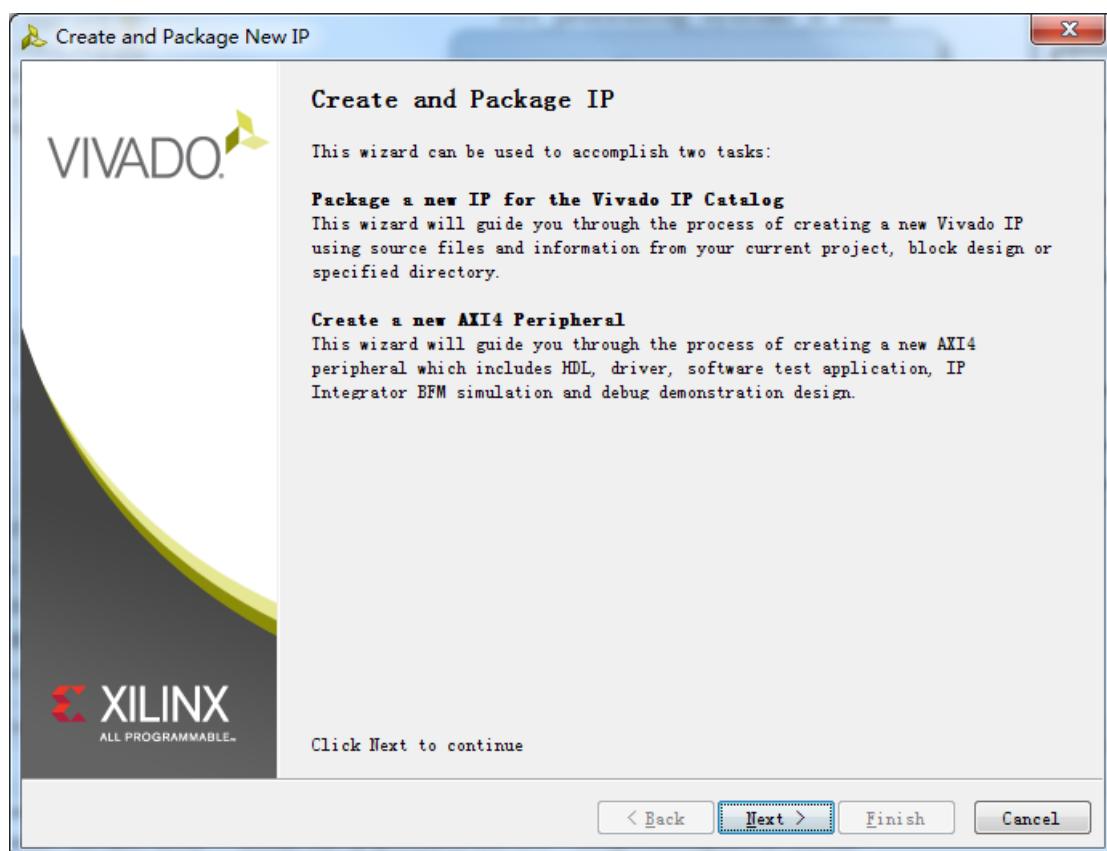
## 12.4 AXI4-Lite 详解

### 12.4.1 AXI4-Lite 源码查看

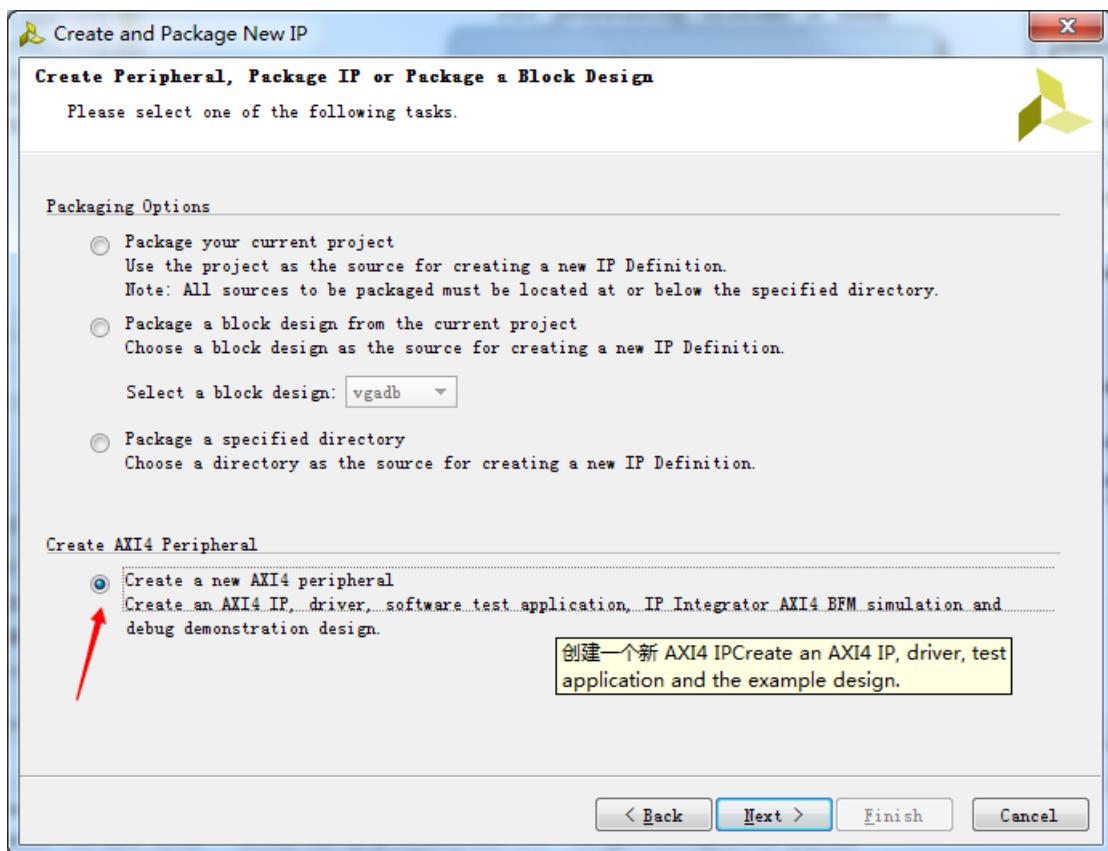
Step1：要看到AXI-Lite的源码，我们先要自定义一个AXI-Lite的IP，新建工程之后，选择，菜单栏->Tools->Create and Package IP：



Step2: 选择Next

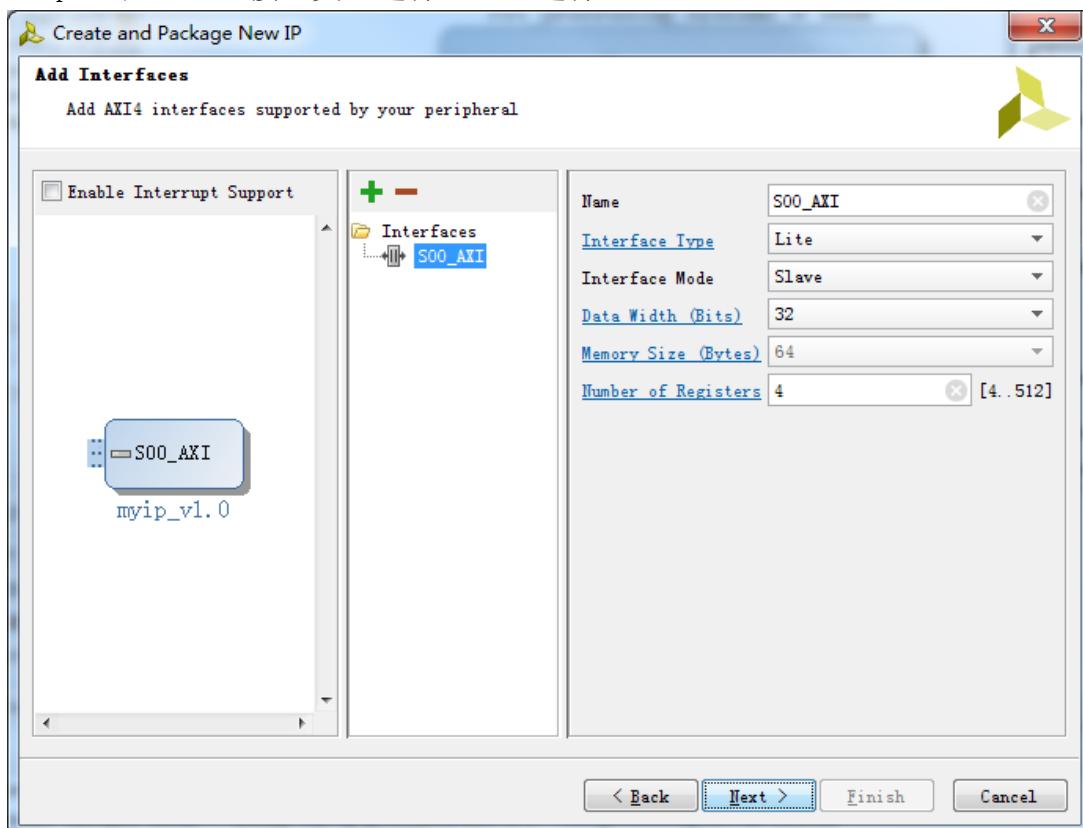


Step3: 选择Create AXI4 Peripheral, 然后Next:

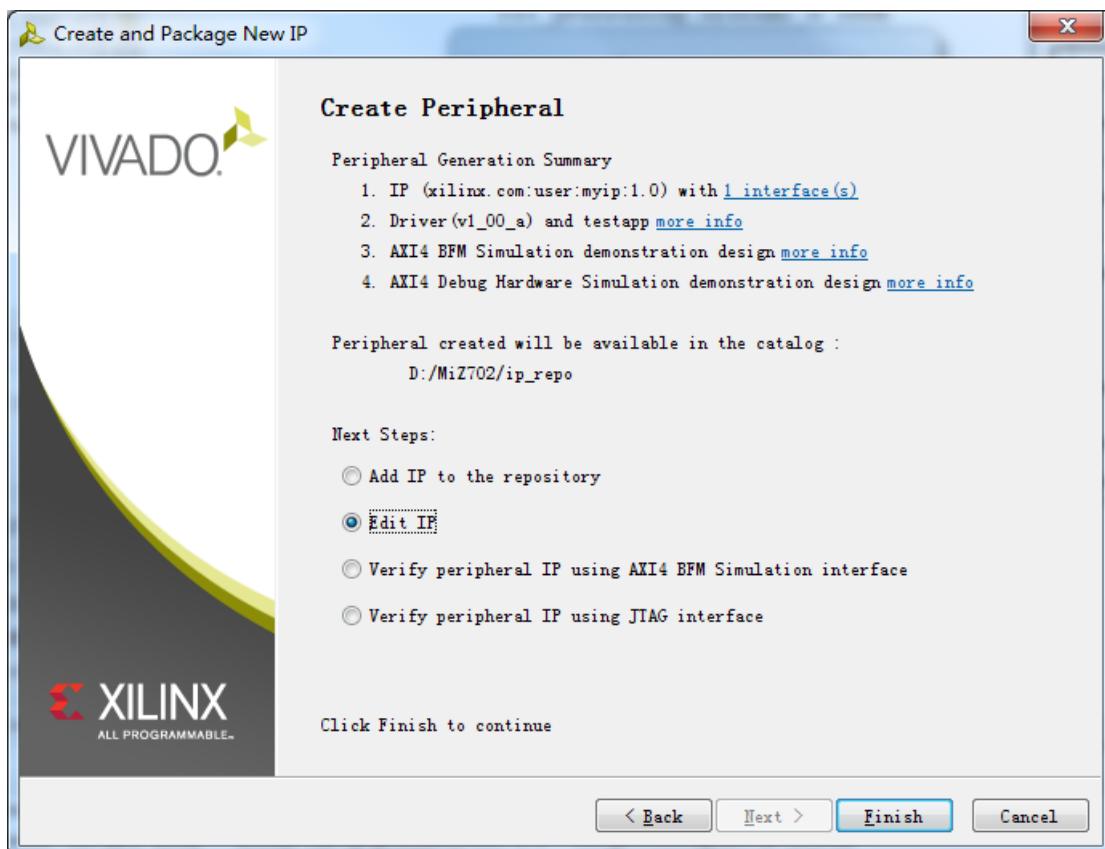


Step4: 给模块命名，保存，然后Next

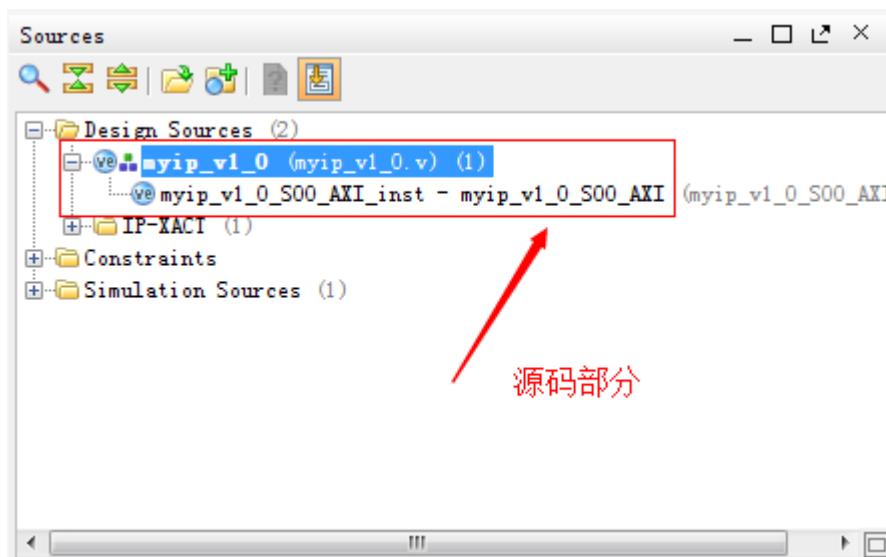
Step5: 注意这里接口类型选择Lite，选择Next:



Step6: 选择Edit IP, 点击Finish:



Step7: 此后, Vivado会新建一个工程, 专门编辑该IP, 通过该工程, 我们就可以看到Vivado为我们生成的AXI-Lite的操作源码:



## 12.4.2 AXI-Lite 源码分析

当打开顶层文件的时, 映入眼帘的是一堆AXI的信号, 这些信号是否似曾相识?

```

input wire s00_axi_aclk,
input wire s00_axi_aresetn,
input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_awaddr,
input wire [2 : 0] s00_axi_awprot,
input wire s00_axi_awvalid,
output wire s00_axi_awready,
input wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_wdata,
input wire [(C_S00_AXI_DATA_WIDTH/8)-1 : 0] s00_axi_wstrb,
input wire s00_axi_wvalid,
output wire s00_axi_wready,
output wire [1 : 0] s00_axi_bresp,
output wire s00_axi_bvalid,
input wire s00_axi_bready,
input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_araddr,
input wire [2 : 0] s00_axi_arprot,
input wire s00_axi_arvalid,
output wire s00_axi_arready,
output wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_rdata,
output wire [1 : 0] s00_axi_rresp,
output wire s00_axi_rvalid,
input wire s00_axi_rready

```

没错笔者曾在《AXI总线概述》这节中提到了他们，这次通过源码分析再次隆重介绍它们。

地址通道		数据通道	
读通道	ARVALID	读地址有效。此信号表明该信道此时能有效读出地址和控制信息	RVALID 读数据有效。此信号表明该信道此时能有效读出数据
	ARADDR	读地址	RDATA 读数据
	ARREADY	读地址准备好了。该信号指示从器件准备好接受一个地址和相关联的控制信号	RREADY 读数据准备好了。该信号指示从器件准备好接收数据
	ARPROT	保护类型。这个信号表示该事务的特权和安全级别，并确定是否该事务是一个数据存取或指令的访问	RRESP 读取响应。这个信号表明读事务处理的状态。

	地址通道		数据通道		应答通道	
写通道	AWVALID	写地址有效。这个信号表示该主信号有效的写地址和控制信息。	WVALID	写有效。这个信号表示有效的写数据和选通信号都可用。	BVALID	写响应有效。此信号表明写命令的有效写入响应。
	AWADDR	写地址	WDATA	写数据	BREADY	响应准备。该信号指示在主主机可以接受一个响应信号
	AWREADY	写地址准备好了。该信号指示从器件准备好接受一个地址和相关联的控制信号	WSTRB	写选通。这个信号表明该字节通道持有效数据。每一bit对应 WDATA 一个字节	BRESP	写响应。这个信号表示写事务处理的状态。
	AWPROT	写通道保护类型。这个信号表示该事务的特权和安全级别，并确定是否该事务是一个数据存取或指令的访问	WREADY	写准备好了。该信号指示从器件可以接受写数据。		

Vivado为我们生成的AXI-Lite的操作源码，是一个例子，我只需要读懂他，然后稍加修改，就可以为我们所用。

我们先来看一段WDATA相关的代码：

```
always @(posedge S_AXI_ACLK)
begin
    if (S_AXI_ARESETN == 1'b0)
        begin
            slv_reg0 <= 0;
            slv_reg1 <= 0;
            slv_reg2 <= 0;
            slv_reg3 <= 0;
        end
    else begin
        if (slv_reg_wren)
            begin
                case (axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB])
                    2'h0:
                        for (byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1)
                            if (S_AXI_WSTRB[byte_index] == 1) begin

```

```

// Respective byte enables are asserted as per write strobes
// Slave register 0
slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
end

2'h1:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
if ( S_AXI_WSTRB[byte_index] == 1 ) begin
// Respective byte enables are asserted as per write strobes
// Slave register 1
slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
end

2'h2:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
if ( S_AXI_WSTRB[byte_index] == 1 ) begin
// Respective byte enables are asserted as per write strobes
// Slave register 2
slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
end

2'h3:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
if ( S_AXI_WSTRB[byte_index] == 1 ) begin
// Respective byte enables are asserted as per write strobes
// Slave register 3
slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
end

default : begin
    slv_reg0 <= slv_reg0;
    slv_reg1 <= slv_reg1;
    slv_reg2 <= slv_reg2;
    slv_reg3 <= slv_reg3;
end

endcase
end
end
end

```

这段程序的作用是，当PS那边向AXI4-Lite总线写数据时，PS这边负责将数据接收  
到寄存器slv\_reg。而slv\_reg寄存器有0~3共4个。至于赋值给哪一个由  
axi\_awaddr[ADDR\_LSB+OPT\_MEM\_ADDR\_BITS:ADDR\_LSB]决定，根据宏定义其实就是由  
axi\_awaddr[3:2]（写地址中不仅包含地址，而且包含了控制位，这里的[3:2]就是控  
制位）决定赋值给哪个slv\_reg。

PS调用写函数时，如果不做地址偏移的话，axi\_awaddr[3:2]的值默认是为0的，举  
个例子，如果我们自定义的IP的地址被映射为0x43C00000，那么我们  
Xil\_Out32(0x43C00000, Value)写的就是slv\_reg0的值。如果地址偏移4位，如

Xil\_Out32(0x43C00000 + 4, Value) 写的就是slv\_reg1的值，依次类推。

分析时只关注slv\_reg0（其他结构上也是一模一样的）：

```
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
```

其中，C\_S\_AXI\_DATA\_WIDTH的宏定义的值为32，也就是数据位宽，S\_AXI\_WSTRB就是写选通信号，S\_AXI\_WDATA就是写数据信号。

存在于for循环中的最关键的一句：

slv\_reg0[(byte\_index\*8) +: 8] <= S\_AXI\_WDATA[(byte\_index\*8) +: 8];

当byte\_index = 0的时候这句话就等价于：

slv\_reg0[7:0] <= S\_AXI\_WDATA[7:0];

当byte\_index = 1的时候这句话就等价于：

slv\_reg0[15:8] <= S\_AXI\_WDATA[15:8];

当byte\_index = 2的时候这句话就等价于：

slv\_reg0[23:16] <= S\_AXI\_WDATA[23:16];

当byte\_index = 3的时候这句话就等价于：

slv\_reg0[31:24] <= S\_AXI\_WDATA[31:24];

也就是说，只有当写选通信号为1时，它所对应S\_AXI\_WDATA的字节才会被读取。

读懂了这段话之后，我们就知道了，如果我们想得到PS写到总线上的数据，我们只需要读取slv\_reg0的值即可。

那如果，我们想写数据到总线让PS读取该数据，我们该怎么做呢？我们继续来看有关RADTA读数据代码：

```
// Output register or memory read data
always @(posedge S_AXI_ACLK)
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_rdata <= 0;
        end
    else
        begin
            // When there is a valid read address (S_AXI_ARVALID) with
            // acceptance of read address by the slave (axi_arready),
            // output the read data
            if (slv_reg_rden)
                begin
                    axi_rdata <= reg_data_out;      // register read data
                end
        end
end
```

观察可知，当PS读取数据时，程序会把reg\_data\_out复制给axi\_rdata（RADTA读数

据）。我们继续追踪reg\_data\_out：

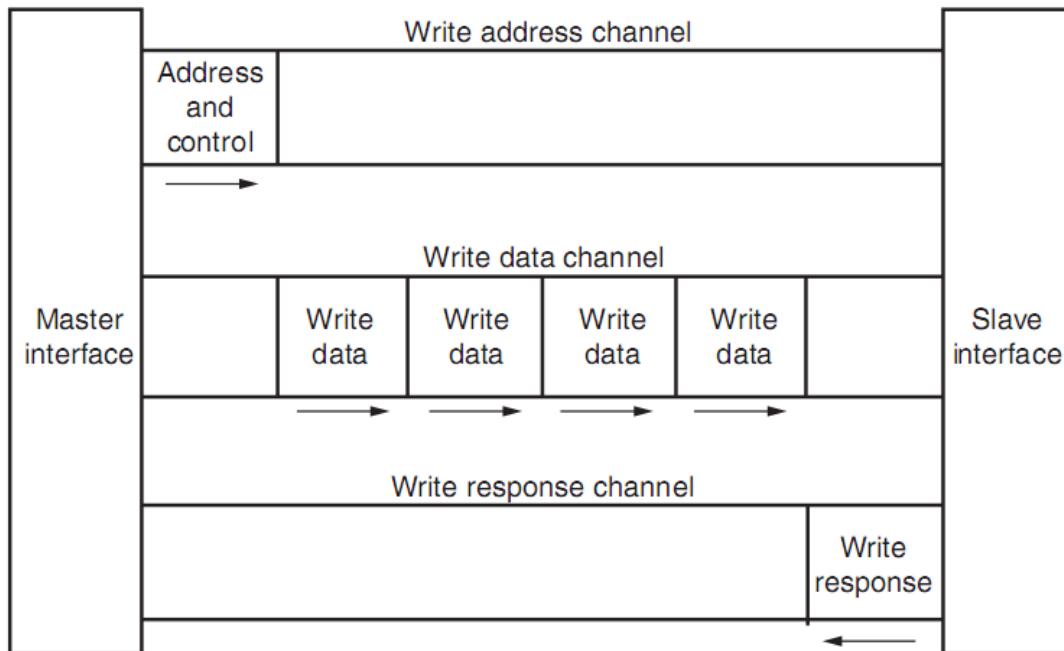
```
always @(*)
begin
    // Address decoding for reading registers
    case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        2' h0 : reg_data_out <= slv_reg0;
        2' h1 : reg_data_out <= slv_reg1;
        2' h2 : reg_data_out <= slv_reg2;
        2' h3 : reg_data_out <= slv_reg3;
        default : reg_data_out <= 0;
    endcase
end
```

和前面分析的一样此时通过判断axi\_awaddr[3:2]的值来判断将那个值给reg\_data\_out上，同样当PS调用读取函数时，这里axi\_awaddr[3:2]默认是0，所以我们只需要把slv\_reg0替换成我们自己数据，就可以让PS通过总线读到我们提供的数据。

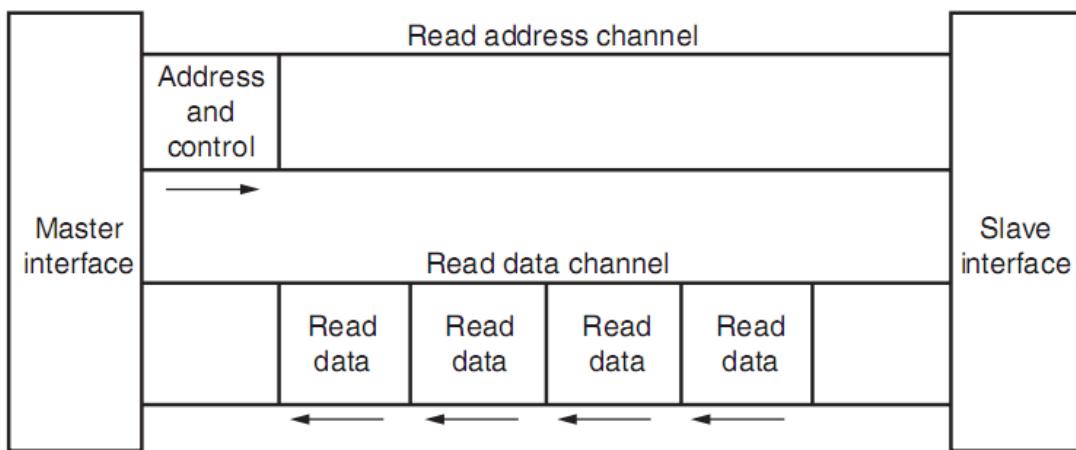
这里可能有的读者会问了，slv\_reg0不是总线写过来的数据吗？因为笔者说过这个程序是Vivado为我们提供的例子，它这么做无非是想验证我写出去的值和我读进入的值相等。但是他怎么写确实会对初看代码的人造成困扰。

最后笔者提出一个问题，为什么写通道要比读通道多了一列应答通道，这是为什么呢？

首先，你要知道这个应答信号是干什么用的？



写应答，主要是回复主机你这个写过程是没有问题的，那读为什么不需要这个过程呢？



这时因为主机在读取数据时，从机可以直接通过读数据通道给主机反馈信息，因此就没有必要再来开辟一个单独的应答通道了。

### 小结：

如果我们想读AXI4\_Lite总线上的数据时，只需关注slv\_reg的数据，我们可自行添加一段代码，如：

```

reg [11:0]rlcd_rgb;
always @(posedge S_AXI_ACLK)
begin
    if (S_AXI_ARESETN == 1'b0)
        begin
            rlcd_rgb <= 12'd0;
        end
    else
        begin
            rlcd_rgb <= slv_reg0[11:0];
        end
    end
assign lcd_rgb = rlcd_rgb;

```

如果我们想对AXI4\_Lite信号写数据时，我们只需修改对reg\_data\_out的赋值，如：

```

//写总线测试修改!!!!!!!
wire[31:0]wlcd_xy;// = {10'd0,lcd_xy};
assign wlcd_xy = {10'd0,lcd_xy};
assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
always @(*)
begin
    // Address decoding for reading registers
    case (axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB])
        2'h0 : reg_data_out <= wlcd_xy;//slv_reg0;

```

```

2' h1  : reg_data_out <= slv_reg1;
2' h2  : reg_data_out <= slv_reg2;
2' h3  : reg_data_out <= slv_reg3;
default : reg_data_out <= 0;
endcase
end

```

最后强调下如果我们自定义的IP的地址被映射为0x43C00000, 那么我们Xil\_Out32(0x43C00000, Value)写的就是slv\_reg0的值。如果地址偏移4位, 如Xil\_Out32(0x43C00000 + 4, Value)写的就是slv\_reg1的值, 依次类推。

目前这里只有4个寄存器, 那是因为之前选择的是4个, 其实我们可以定义的更多:



在ps的头文件里可以看到我们自定义的IP的地址是有个范围的

```
#define XPAR_MYIPFREQUENCY_0_S00_AXI_BASEADDR 0x43C00000
#define XPAR_MYIPFREQUENCY_0_S00_AXI_HIGHADDR 0x43C0FFFF
```

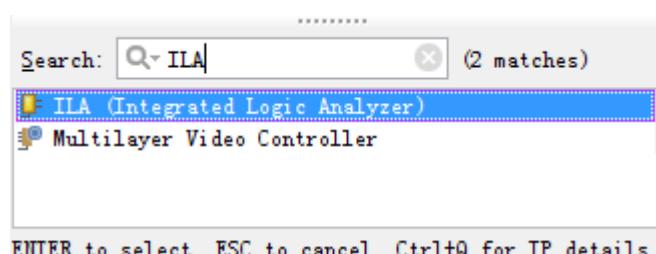
理论上只要基址 + 偏移量不要超过HIGHADDR即可。

## 12.5 观察 AXI4-Lite 总线信号

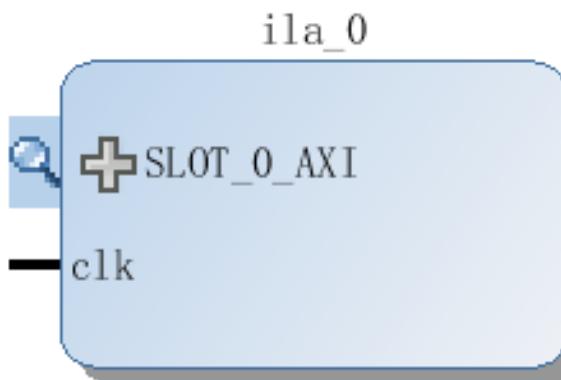
在第十章, 我们封装了一个AXI\_Lite的GPIO, 通过本章的分析, 我们在第十章工程的基础上通过添加一个ila核的方式, 来具体看看AXI\_Lite总线的信号。

Step1: 做好第十章工程的备份, 然后直接打开第十章的工程。

Step2: 单击 IP icon 添加 ila CORE



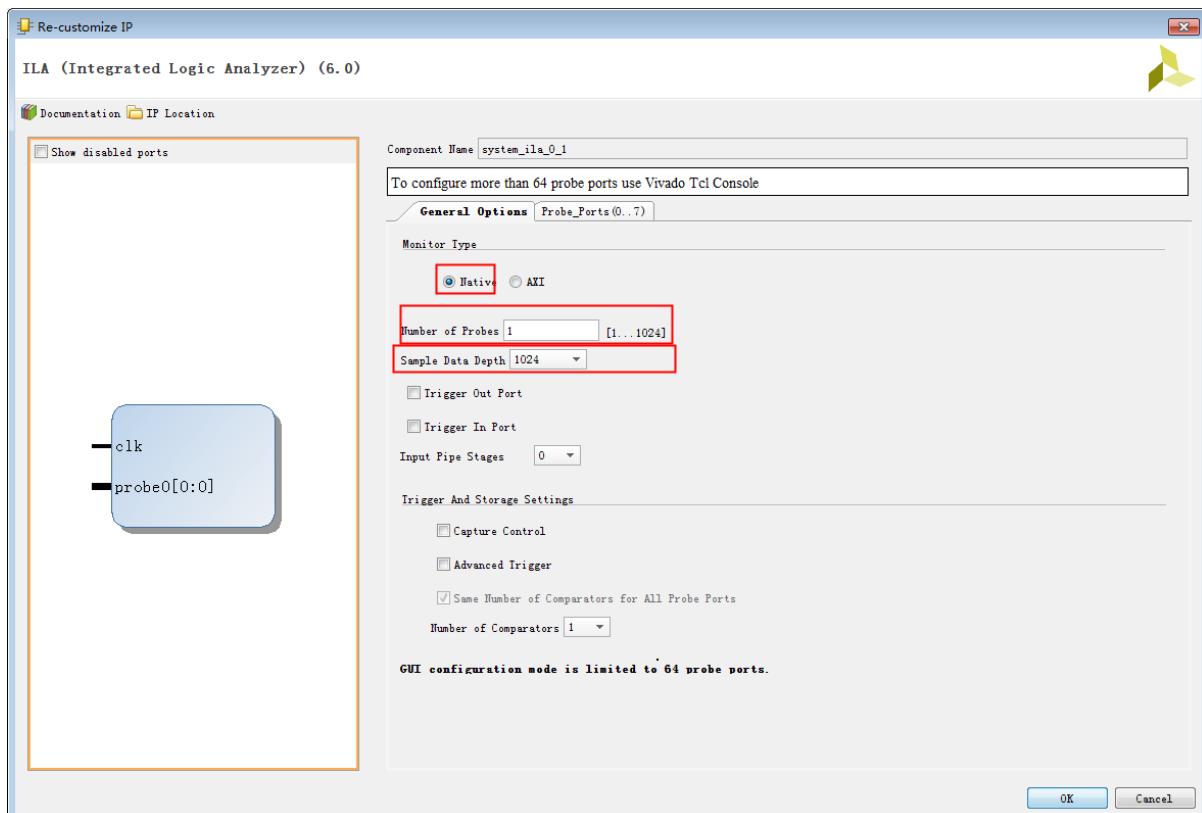
Step3: 双击打开 ILA CORE



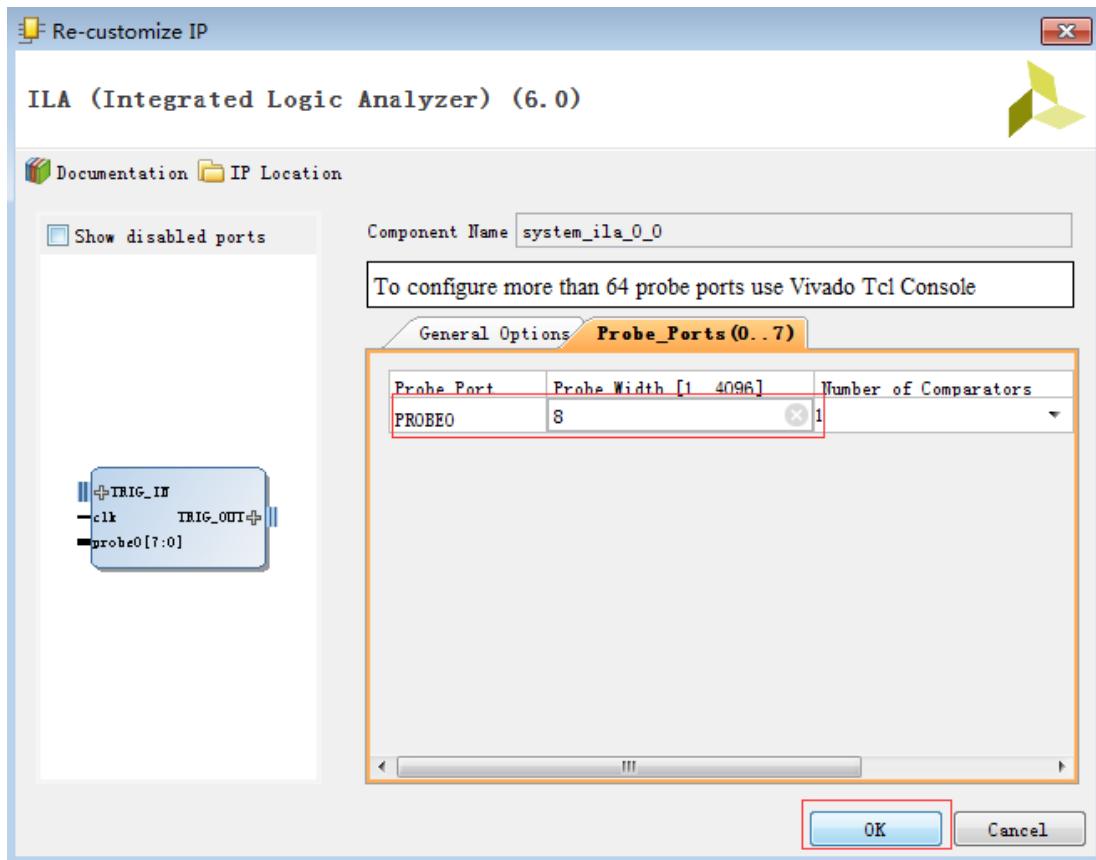
ILA (Integrated Logic Analyzer)

Step4: 双击打开 ILA CORE

General Options 设置如下



Probe\_Ports 设置如下,之后单击 OK

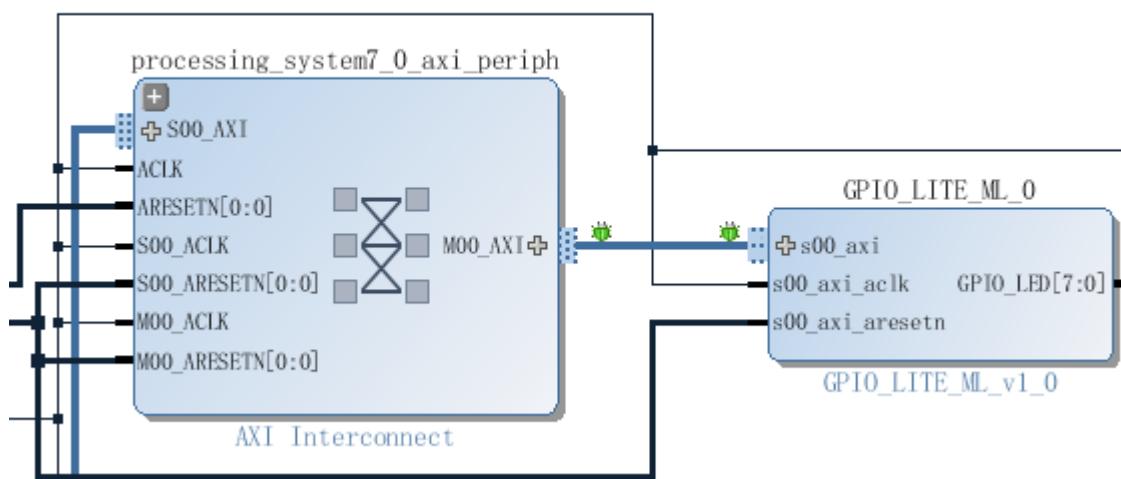


Step5: 连接 Probe0 到 GPIO\_LED。

Step6: 连接 CLK 接口到 FCLK\_CLK0 接口

Step7: 选中 Processing\_System7\_0\_axi\_periph 和 GPIO\_LITE\_ML\_0 之间的 S\_AXI 总线。

Step8: 右击选择 Mark Debug



Step9: 接下来依然是，右键单击 Block 文件，文件选择 Generate the Output

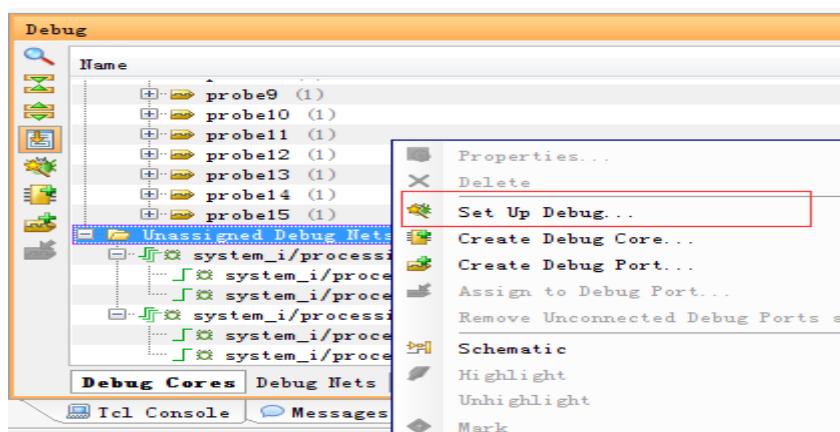
Products。

Step10:继续右键单击 Block 文件，选择 Create a HDL wrapper，根据 Block 文件内容产生一个 HDL 的顶层文件，并选择让 vivado 自动完成。

Step11:单击 Run Synthesis,如果有 Save 对话框弹出选择保存。

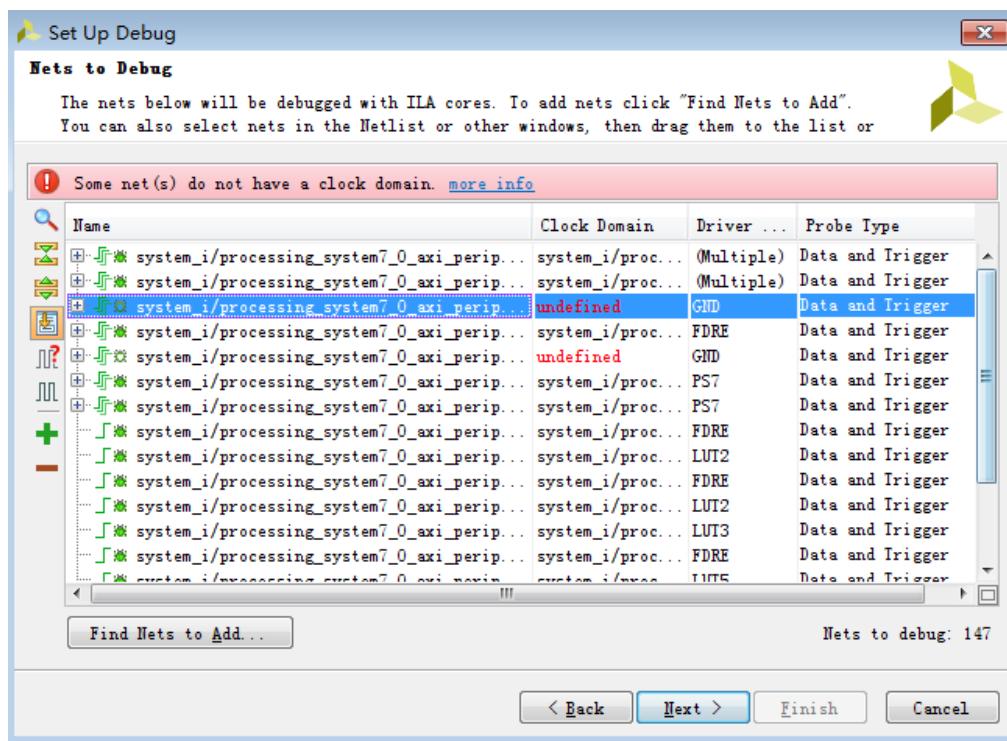
Step12:综合结束后选择 Synthesized Design option 单击 OK。

Step13:在如下对话框中找到 Unassigned debug nets(如果对话框没有出现选择 菜单->Window > Debug)



Step14:右击 Unassigned Debug Nets 选择 Set up Debug... 之后单击 Next

Step15:删除红色错误的信号然后单击 Next 到结束



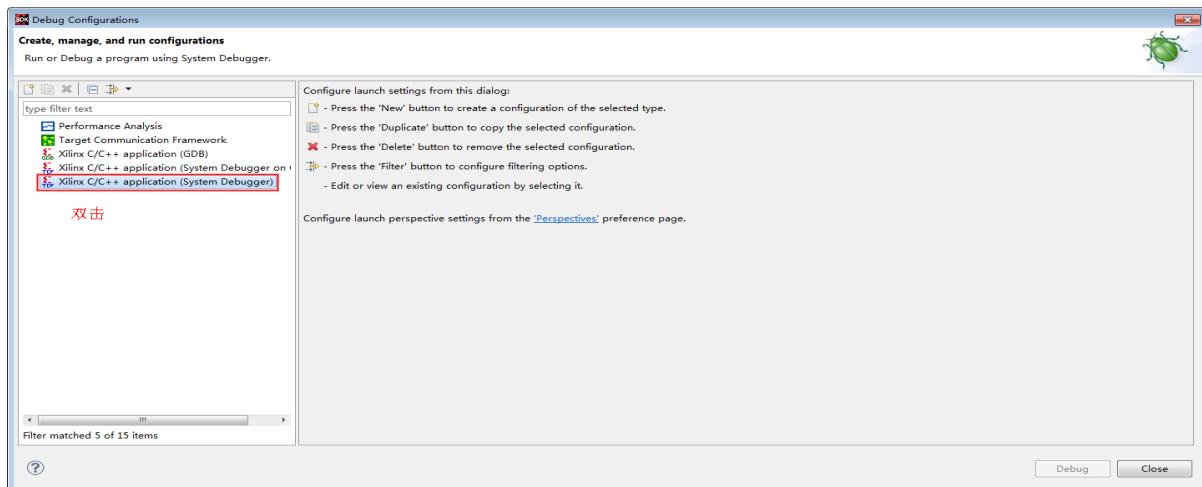
Step16:生成 Bit 文件。

## 12.6 加载到 SDK

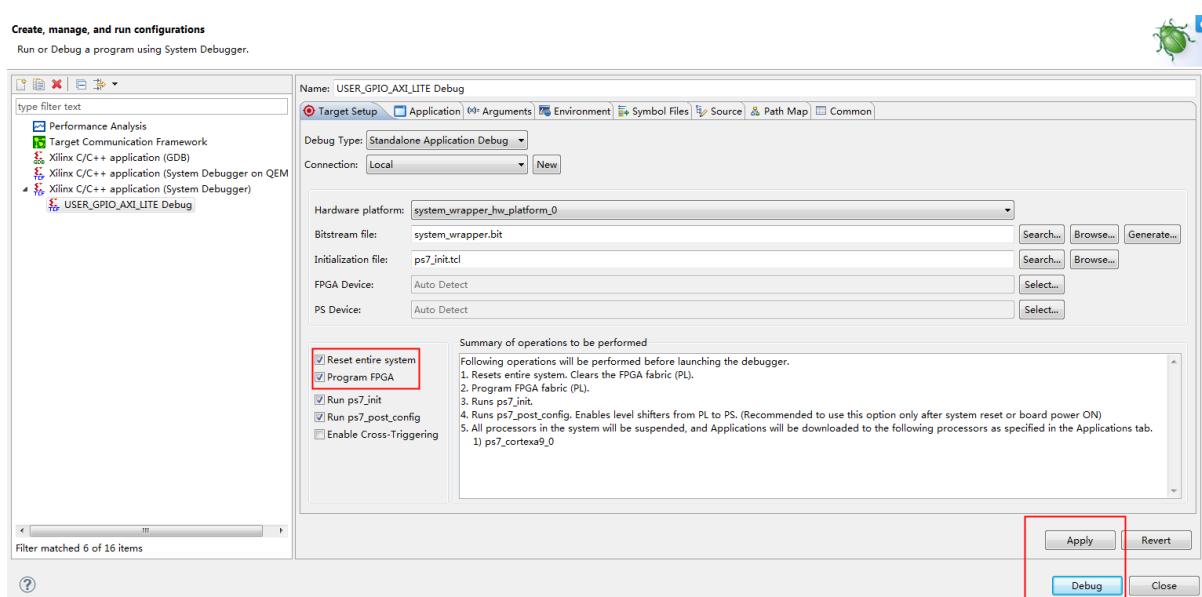
Step1: 导出硬件。

Step2: 右击工程，选择 Debug as ->Debug configuration。

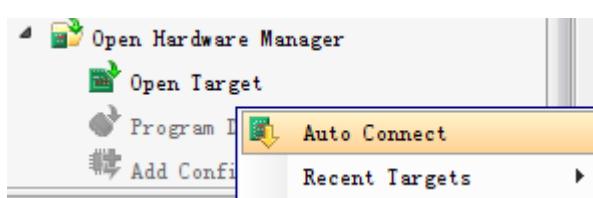
Step3: 选中 system Debugger,双击创建一个系统调试。



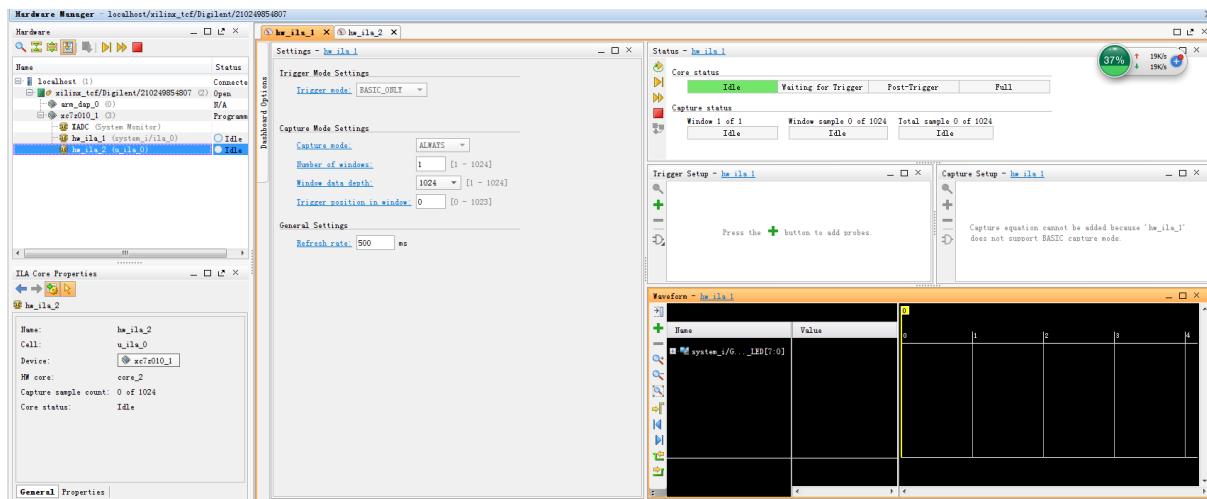
Step4: 设置系统调试。



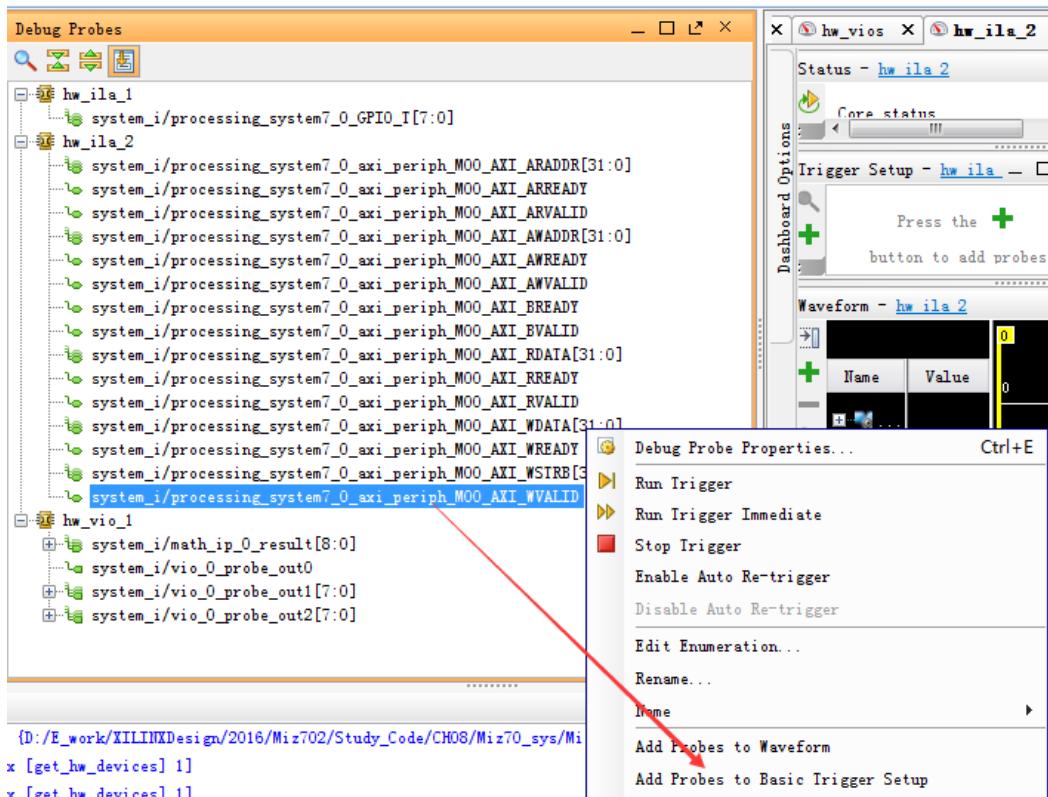
Step5:回到 VIVADO 单击 Open Target->Auto Connect



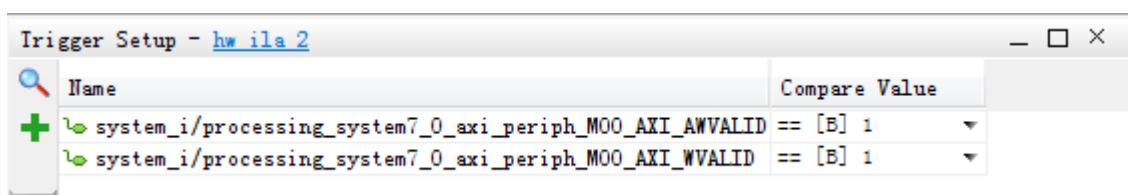
### Step6:加载完成后的界面



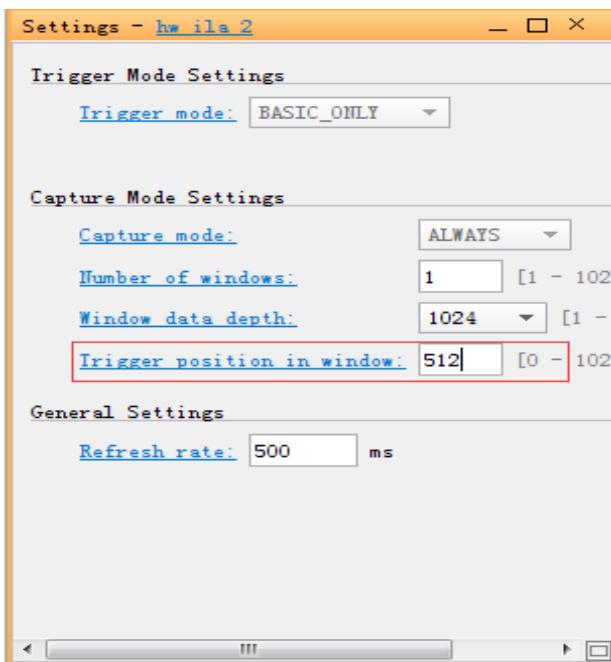
Step7:选择菜单->window->Debugprobes 选择 AXI\_WVALID 和 AXI\_AWVALID 做为触发信号



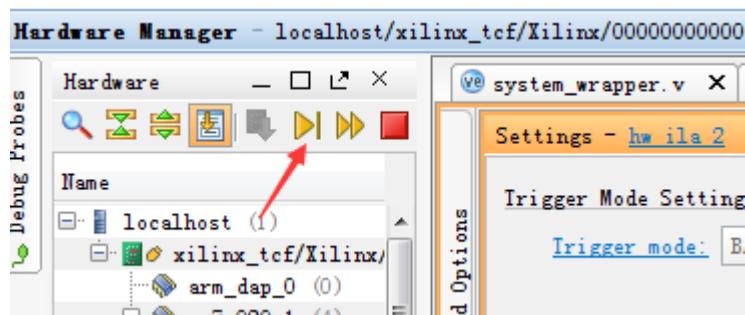
Step8:设置触发条件为 1



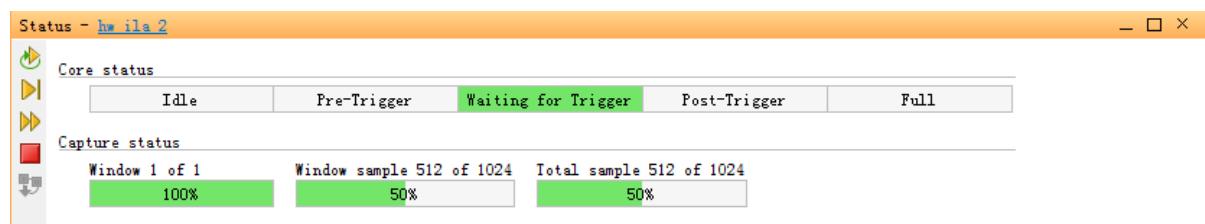
Step9:设置触发位置为 512



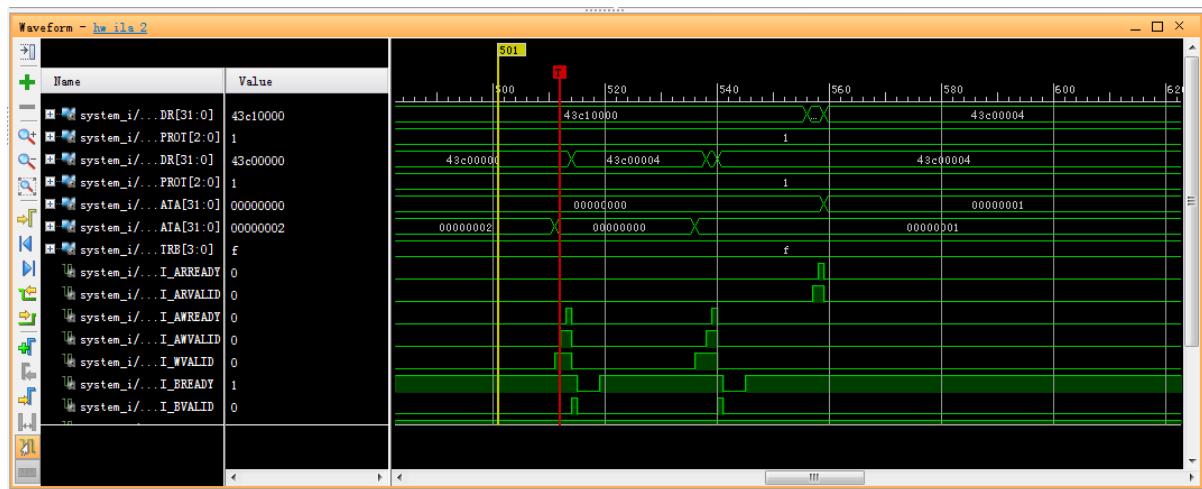
Step10:单击箭头所指向启动触发



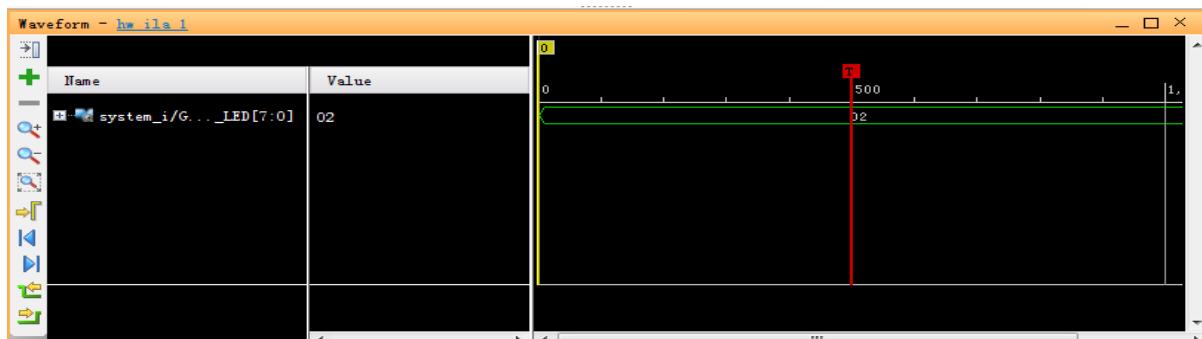
Step11:进入等待触发状态



Step12: 单击运行 | 后 VIVADO HW\_ILA2 窗口采集到波形输出，可以看到 AXI 总线的工作时序。



Step13:HW\_ILA1 窗口采集到的数据是 GPIO\_LED 的值为 0x02，同时可观察到开发板上的 LED2 亮起。



## 12.7 本章小结

通过本章的学习，我们首先得认识到总线和接口以及协议的区别，其次通过分析 AXI4-Lite，AXI4-Stream，AXI4总线的从机代码，对AXI协议有一定的认识，那么在后面学习AXI的一些IP时就不会有恐惧的心理。

最后，我们再理一理AXI总线和AXI接口的关系。在ZYNQ中，支持AXI4-Lite，AXI4 和AXI4-Stream三种总线协议，这前面已经说过了，要注意的是PS与PL之间的接口

(AXI-GP接口，AXI-HP接口以及AXI-ACP接口)却只支持AXI-Lite和AXI协议这两种总线协议。也就是说PL这边的AXI-Stream的接口是不能直接与PS对接的，需要经过AXI4或者AXI4-Lite的转换。比如后面将用到的VDMA IP，它就实现了在PL内部AXI4到AXI-Stream的转换，VDMA利用的接口就是AXI-HP接口。

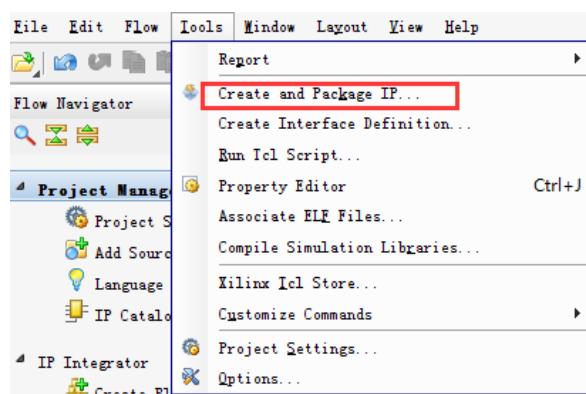
## CH13\_AXI\_PWM 实验

当学习了上一章的协议介绍内容后，开发基于这些协议的方案已经不是什么难事了，关键的一点就是从零到有的突破了。本章就以 AXI-Lite 总线实现 8 路 LED 自定义 IP 作为第一验证 AXI-Lite 总线应用的方案，带领大家快速进入实战状态。

### 13.1 自定义 IP 的封装

Step1：新建一个名为 Miz\_sys 空的工程。

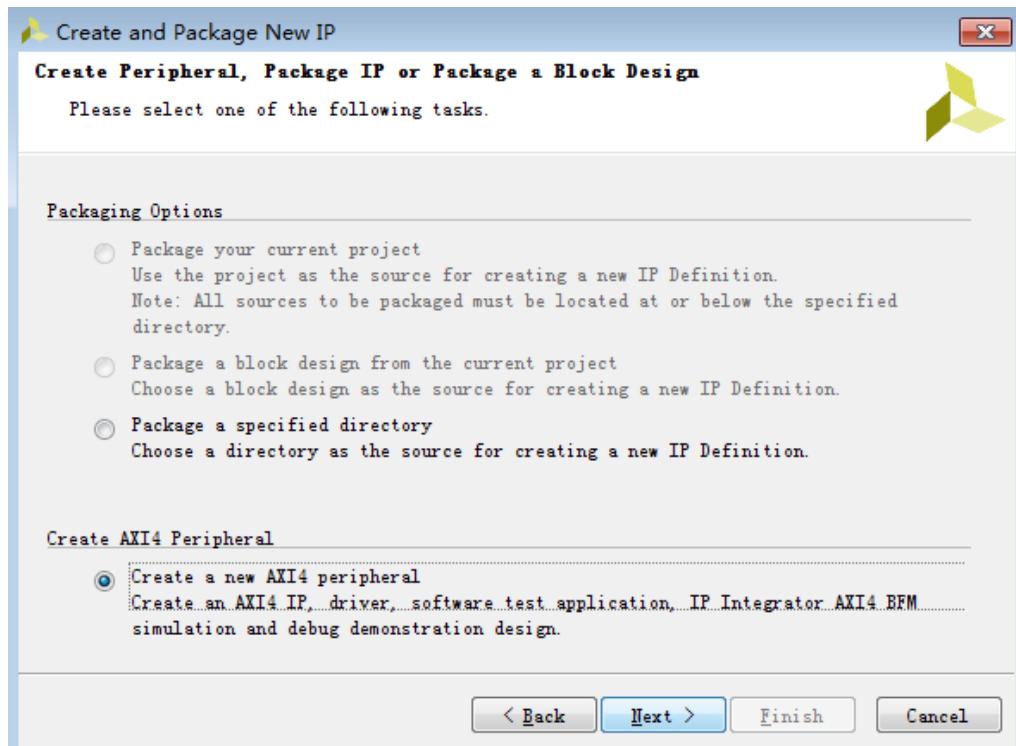
Step2：选择 Tools Create and Package IP 创建 IP



Step3:单击 NEXT

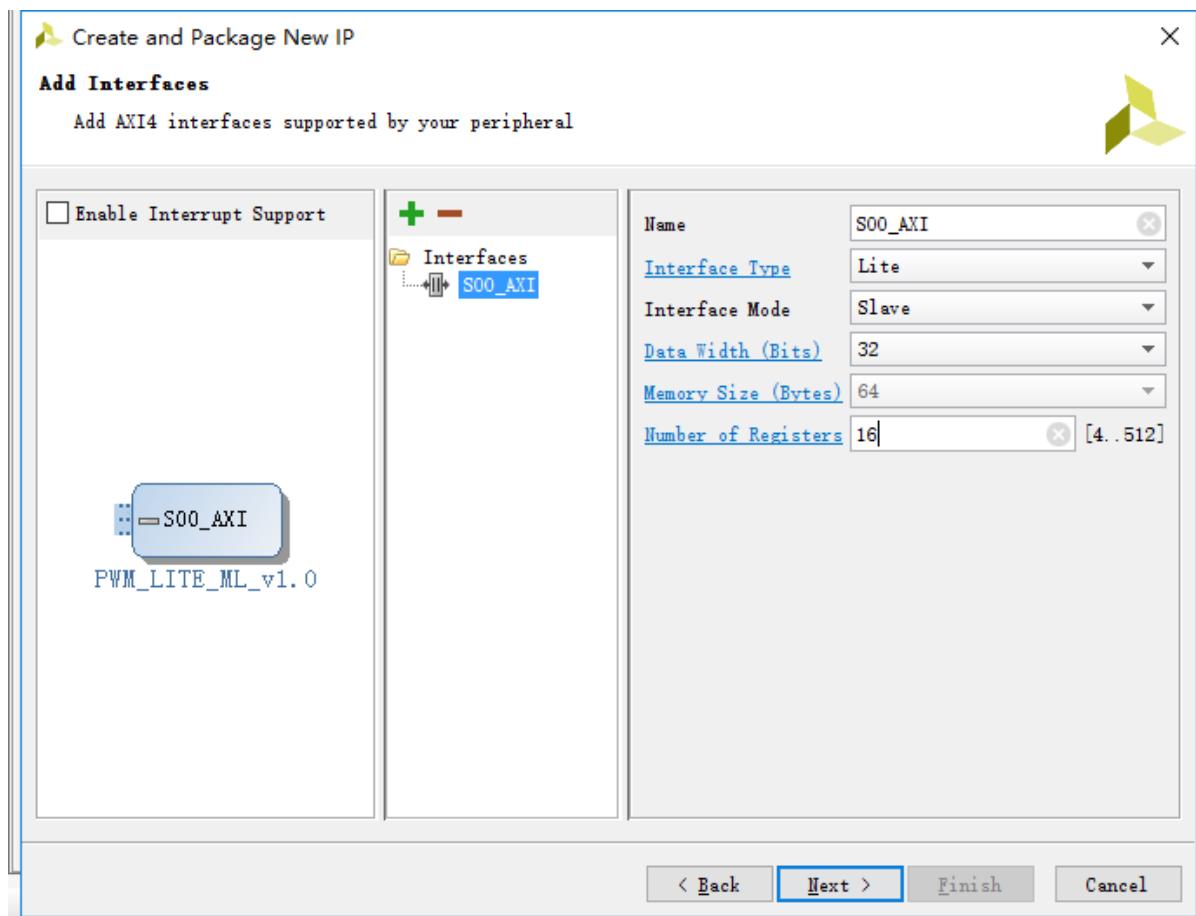


Step4:由于我们需要挂在到总线上，因此创建一个带 AXI 总线的用户 IP

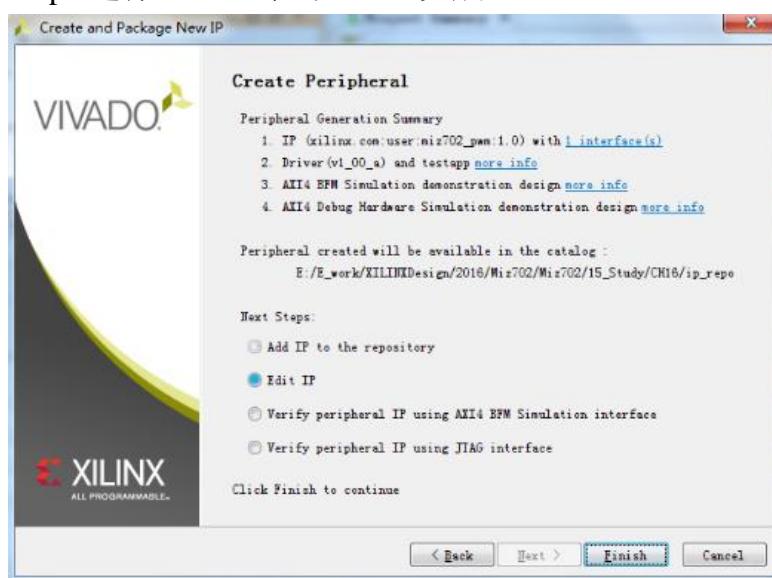


Step5: 设置 IP 的名字为 PWM\_LITE\_ML 版本号默认，并且记住 IP 的位置

Step6: 设置总线形式为 Lite 总线，Lite 总线是简化的 AXI 总线消耗的资源少，当然性能也是比完全版的 AXI 总线差一点，但是由于音频的速度并不高，因此采用 Lite 总线就够了，设置寄存器数量为 16，因为后面我们需要用到 16 个寄存器。



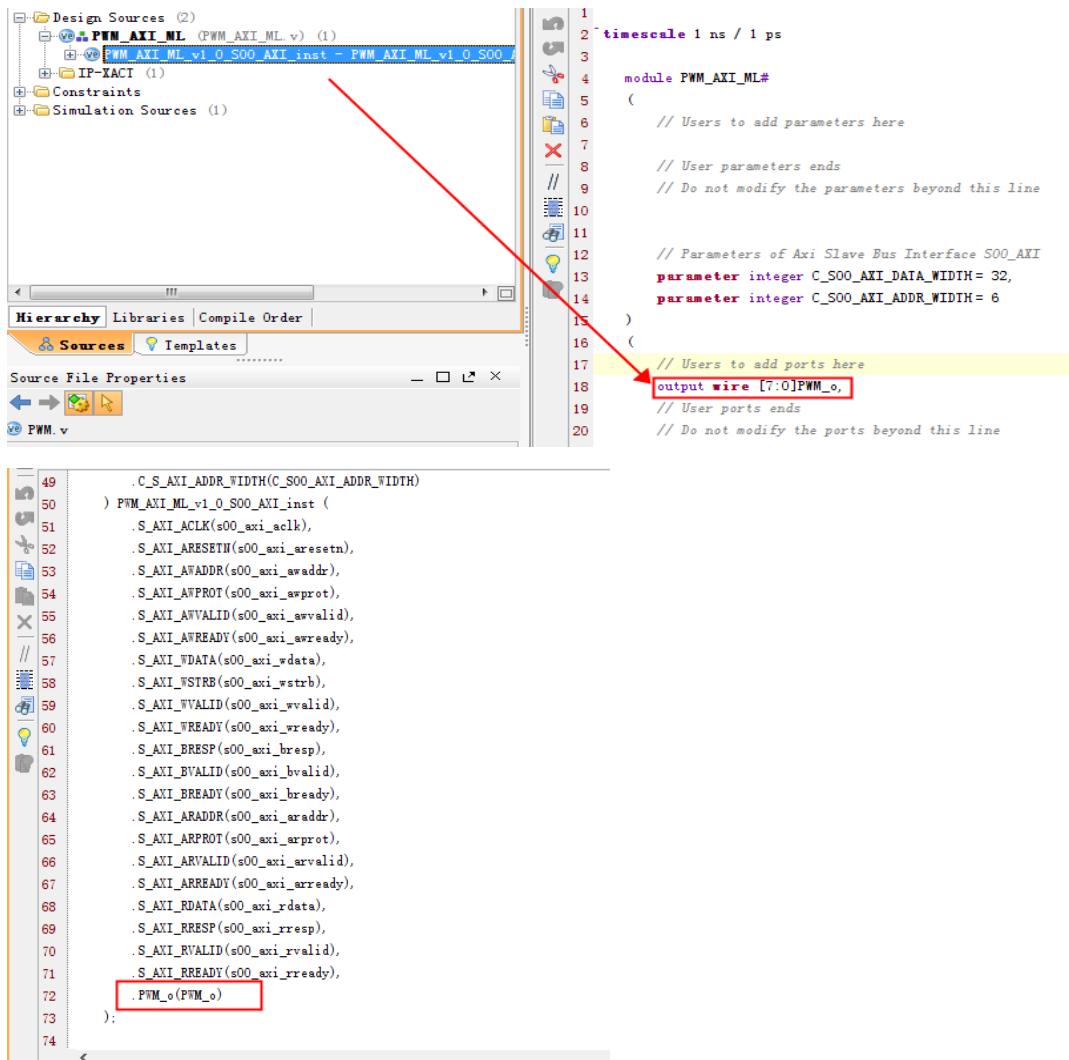
Step7:选择 Edit IP 单击 Finish 完成



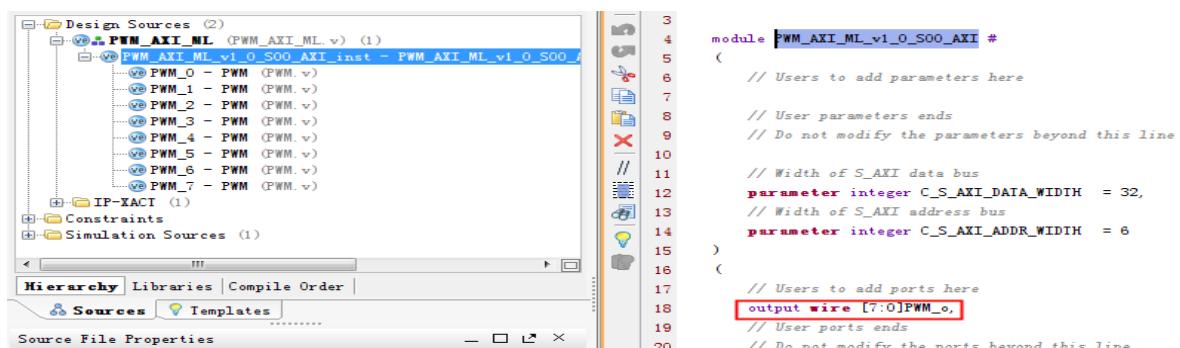
## 13.2 用户 IP 的修改

IP 创建完成后，并不能立马使用，还需要做一些修改。

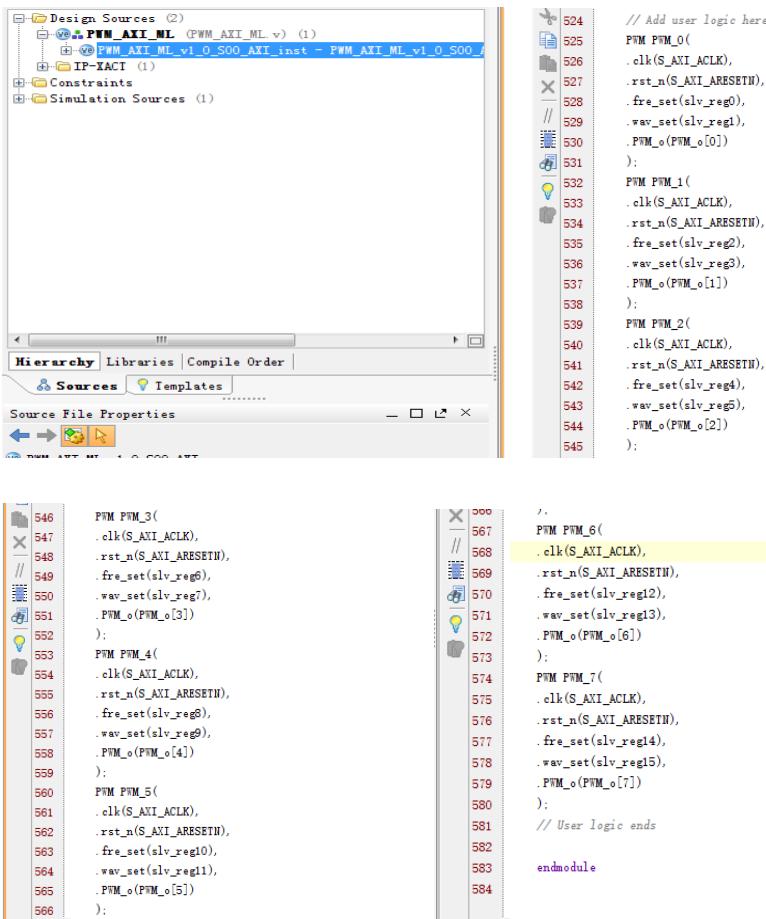
Step1:打开 PWM\_AXI\_ML.v 文件在以下位置修改:



Step2:修改 PWM\_AXI\_ML\_v1\_0\_S00\_AXI.v 的端口部分



Step3:slv\_reg0-slv\_reg5 为 PS 部分写入 PL 的寄存器。通过这个 16 个寄存器的值,我们可以控制 PWM 的占空比。



Step3:下面这段代码就是 PS 写 PL 部分的寄存器，一共有 16 个寄存器

```
always @(\posedge S_AXI_ACLK)
begin
if ( S_AXI_ARESETN == 1'b0 )
begin
    slv_reg0 <= 0;
    slv_reg1 <= 0;
    slv_reg2 <= 0;
    slv_reg3 <= 0;
    slv_reg4 <= 0;
    slv_reg5 <= 0;
    slv_reg6 <= 0;
    slv_reg7 <= 0;
    slv_reg8 <= 0;
    slv_reg9 <= 0;
    slv_reg10 <= 0;
    slv_reg11 <= 0;
    slv_reg12 <= 0;
    slv_reg13 <= 0;
    slv_reg14 <= 0;
```

```
    slv_reg15 <= 0;
end
else begin
    if (slv_reg_wren)
        begin
            case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
                4'h0:
                    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                            // Respective byte enables are asserted as per write strobes
                            // Slave register 0
                            slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                        end
                4'h1:
                    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                            // Respective byte enables are asserted as per write strobes
                            // Slave register 1
                            slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                        end
                4'h2:
                    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                            // Respective byte enables are asserted as per write strobes
                            // Slave register 2
                            slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                        end
                4'h3:
                    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                            // Respective byte enables are asserted as per write strobes
                            // Slave register 3
                            slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
                        end
                4'h4:
                    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
                        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                            // Respective byte enables are asserted as per write strobes
```

```
// Slave register 4
    slv_reg4[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
end
4'h5:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 5
        slv_reg5[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
4'h6:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 6
        slv_reg6[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
4'h7:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 7
        slv_reg7[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
4'h8:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 8
        slv_reg8[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
4'h9:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 9
        slv_reg9[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
```

```
4'hA:  
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =  
byte_index+1 )  
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin  
            // Respective byte enables are asserted as per write strobes  
            // Slave register 10  
            slv_reg10[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];  
        end  
4'hB:  
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =  
byte_index+1 )  
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin  
            // Respective byte enables are asserted as per write strobes  
            // Slave register 11  
            slv_reg11[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];  
        end  
4'hC:  
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =  
byte_index+1 )  
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin  
            // Respective byte enables are asserted as per write strobes  
            // Slave register 12  
            slv_reg12[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];  
        end  
4'hD:  
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =  
byte_index+1 )  
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin  
            // Respective byte enables are asserted as per write strobes  
            // Slave register 13  
            slv_reg13[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];  
        end  
4'hE:  
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =  
byte_index+1 )  
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin  
            // Respective byte enables are asserted as per write strobes  
            // Slave register 14  
            slv_reg14[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];  
        end  
4'hF:  
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =  
byte_index+1 )
```

```

if ( S_AXI_WSTRB[byte_index] == 1 ) begin
    // Respective byte enables are asserted as per write strobes
    // Slave register 15
    slv_reg15[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
end
default : begin
    slv_reg0 <= slv_reg0;
    slv_reg1 <= slv_reg1;
    slv_reg2 <= slv_reg2;
    slv_reg3 <= slv_reg3;
    slv_reg4 <= slv_reg4;
    slv_reg5 <= slv_reg5;
    slv_reg6 <= slv_reg6;
    slv_reg7 <= slv_reg7;
    slv_reg8 <= slv_reg8;
    slv_reg9 <= slv_reg9;
    slv_reg10 <= slv_reg10;
    slv_reg11 <= slv_reg11;
    slv_reg12 <= slv_reg12;
    slv_reg13 <= slv_reg13;
    slv_reg14 <= slv_reg14;
    slv_reg15 <= slv_reg15;
end
endcase
end
end
end

```

Step4:新建一个 PWM.v 文件实现 PWM 输出。

```

module PWM(
    input clk,
    input rst_n,
    input [31:0]fre_set,
    input [31:0]wav_set,
    output  PWM_o
);

reg [31:0]fre_cnt;
always @(posedge clk)begin
    if(rst_n==1'b0)begin
        fre_cnt <=32'd0;
    end
    else begin
        if(fre_cnt<fre_set) begin

```

```

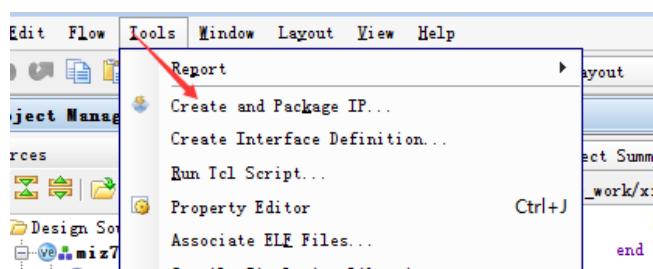
    fre_cnt <= fre_cnt+1'b1;
end
else begin
    fre_cnt<=32'd0;
end
end
end

assign PWM_o = (wav_set>fre_cnt);

endmodule

```

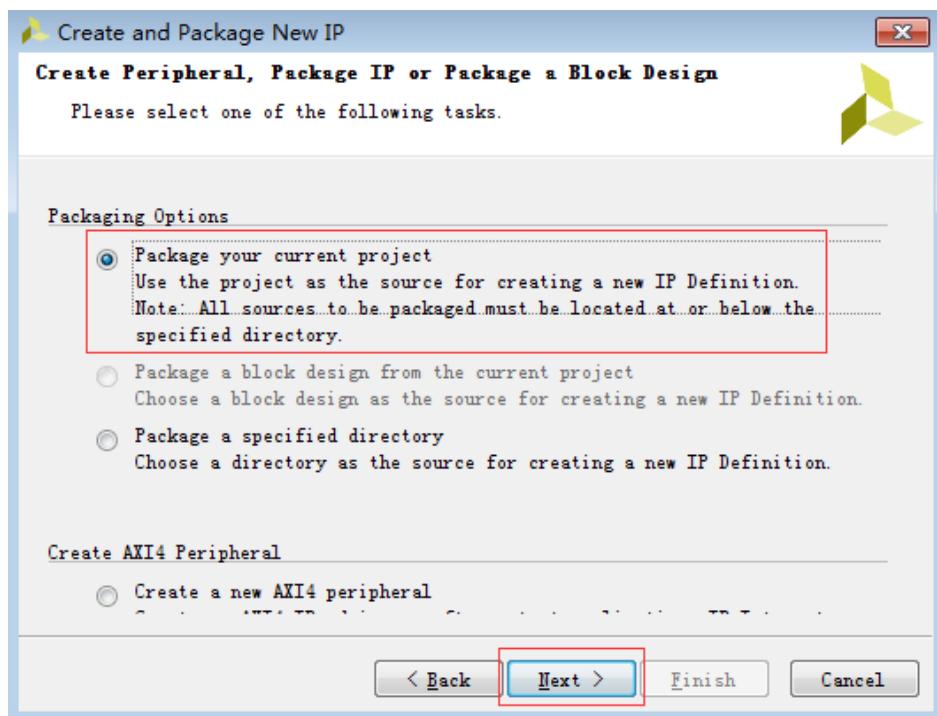
Step5:修改完成后重新封装一次自定义 IP



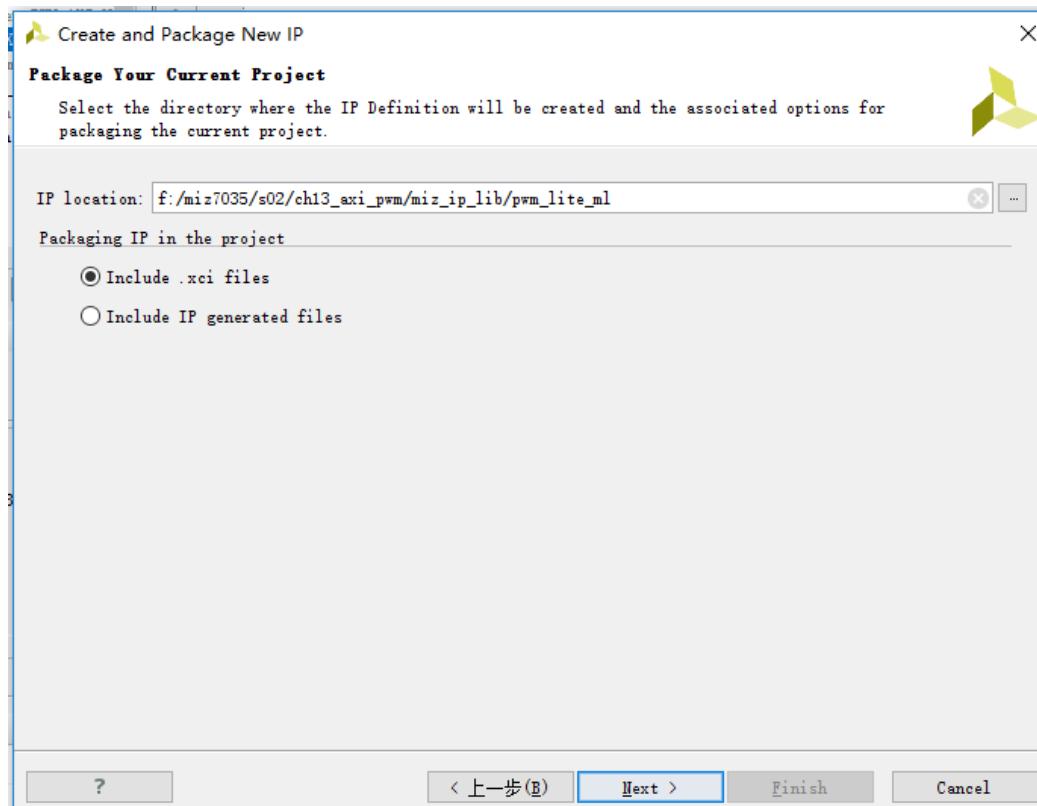
Step6:单击 NEXT



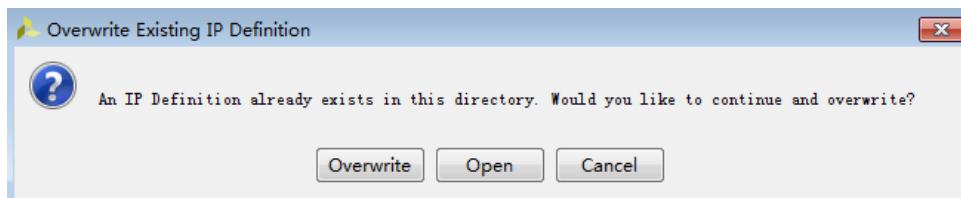
Step7:和第一次不同，这次选择第一个单选框然后单击 NEXT



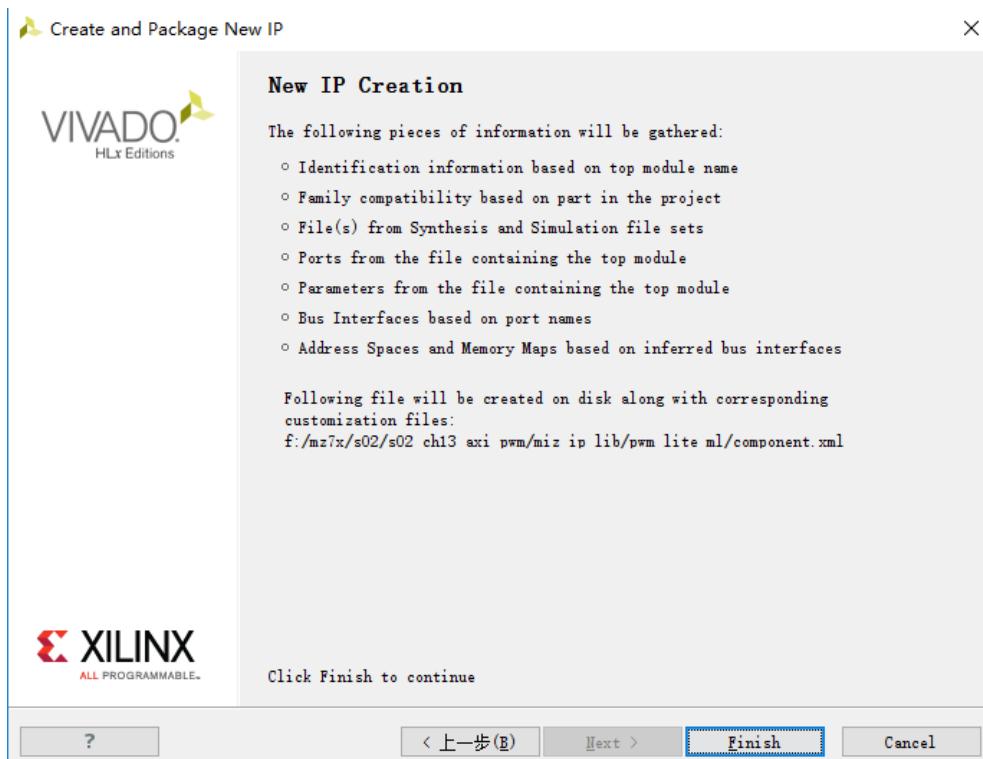
Step8:选择第一个单选框，然后单击 NEXT



Step9:点击 Overwrite



Step10:点击 Finish 到此自定义 IP 结束



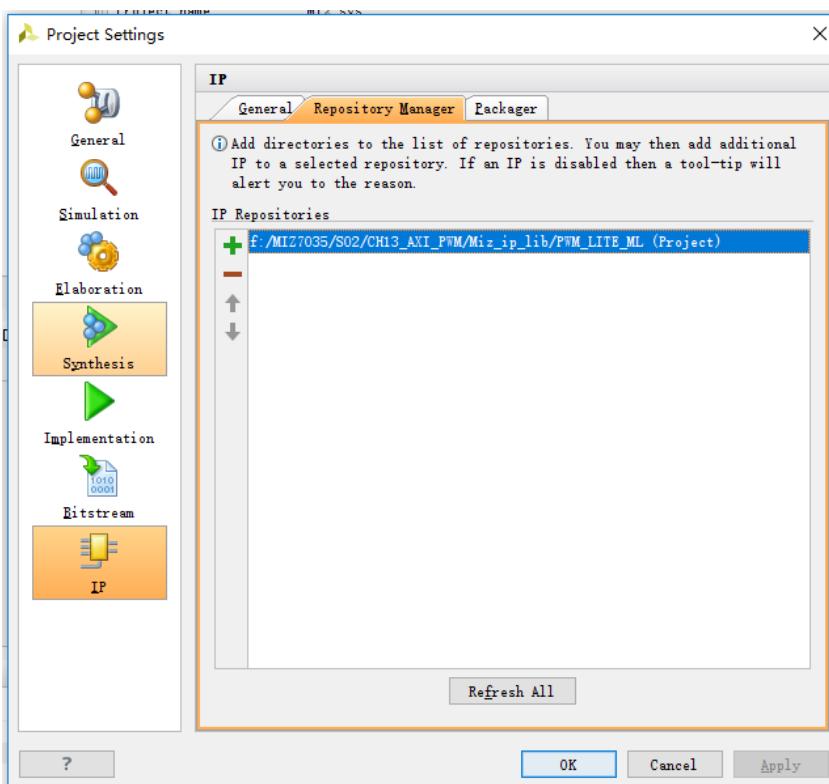
### 13.3 搭建硬件工程

Step1: 另外新建一个 VIVADO 工程, 根据自己的开发板正确配置芯片型号。

Step2: 在 Project manager 区中单击 Project settings。

Step3: 选择 IP 设置区中的 repository manager, 将上一节我们封装好的 IP 的路径添加进去。

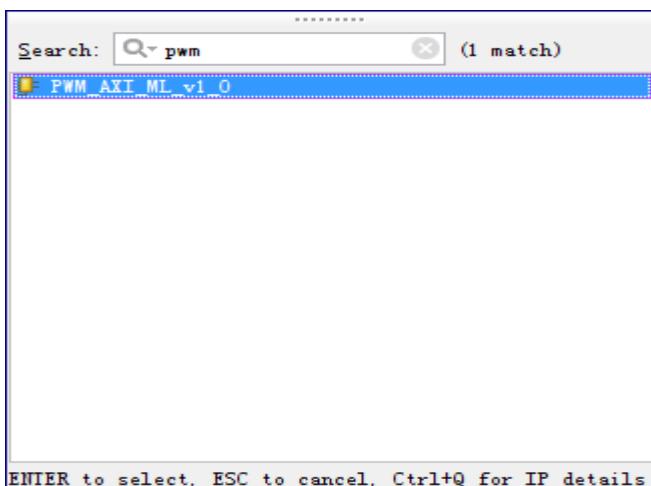
Step4: 单击+号图标, 将上一节封装的 IP 的路劲存放进去, 单击 OK。



Step5: 新建一个 BD 文件，输入文件名，完成创建。

Step6: 向 BD 文件中添加一个 ZYNQ Processing system,根据自身硬件完成 IP 的配置。

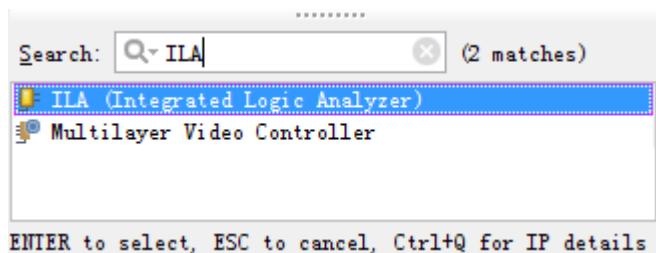
Step7: 单击添加 IP 图标，输入上一节我们自定义 IP 的模块名，将其添加入 BD 文件中。



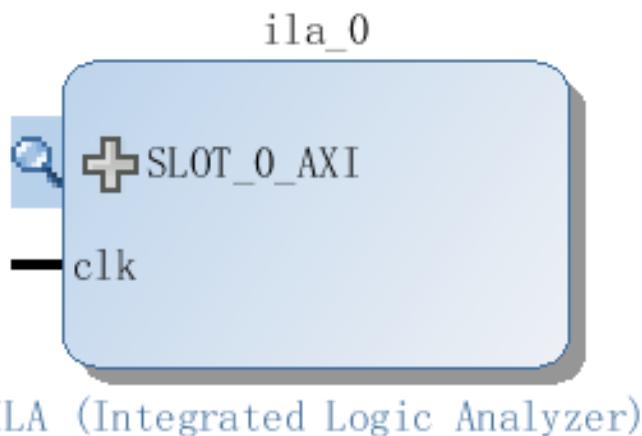
Step8: 直接点击 Run connection automation，然后单击 OK。

Step9: 选中 PWM\_o，按 Ctrl+T 组合键引出端口。

Step10: 单击 IP icon  添加 ila CORE

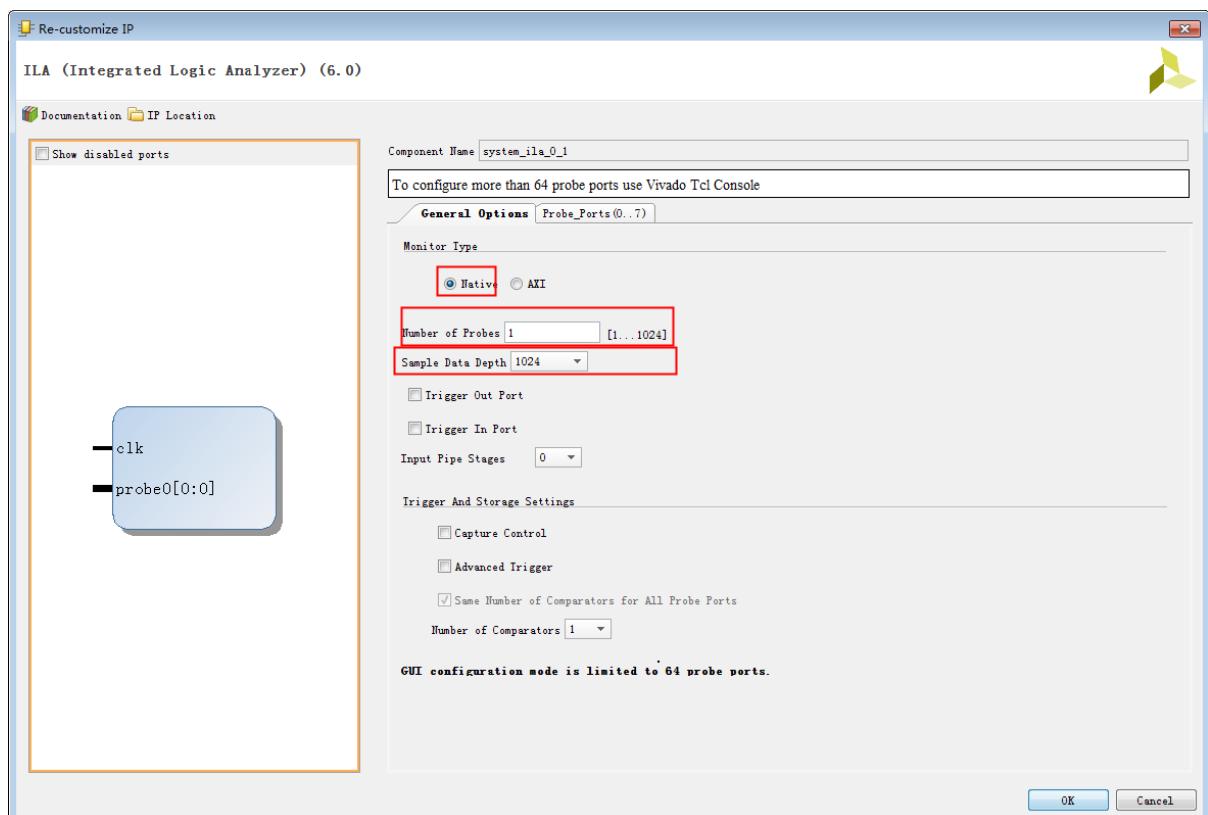


Step11: 双击打开 ILA CORE

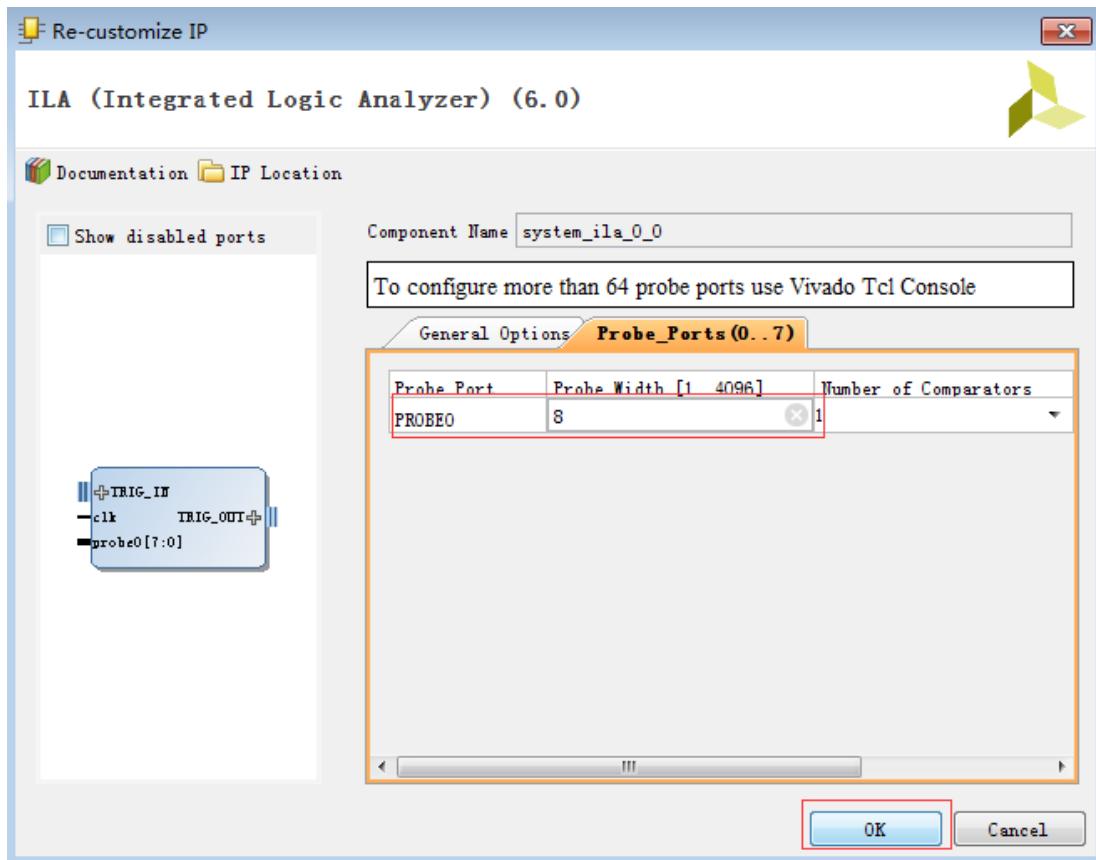


Step12: 双击打开 ILA CORE

General Options 设置如下



Probe\_Ports 设置如下,之后单击 OK



Step13: 连接 Probe0 到 PWM\_o。

Step14: 连接 CLK 接口到 FCLK\_CLK0 接口。

Step15: 右键单击 Block 文件，文件选择 Generate the Output Products。

Step16: 右键单击 Block 文件，选择 Create a HDL wrapper，根据 Block 文件内容产生一个 HDL 的顶层文件，并选择让 vivado 自动完成。

Step17: 将我们提供的约束文件添加到工程当中来。

Step18: 生成 bit 文件。

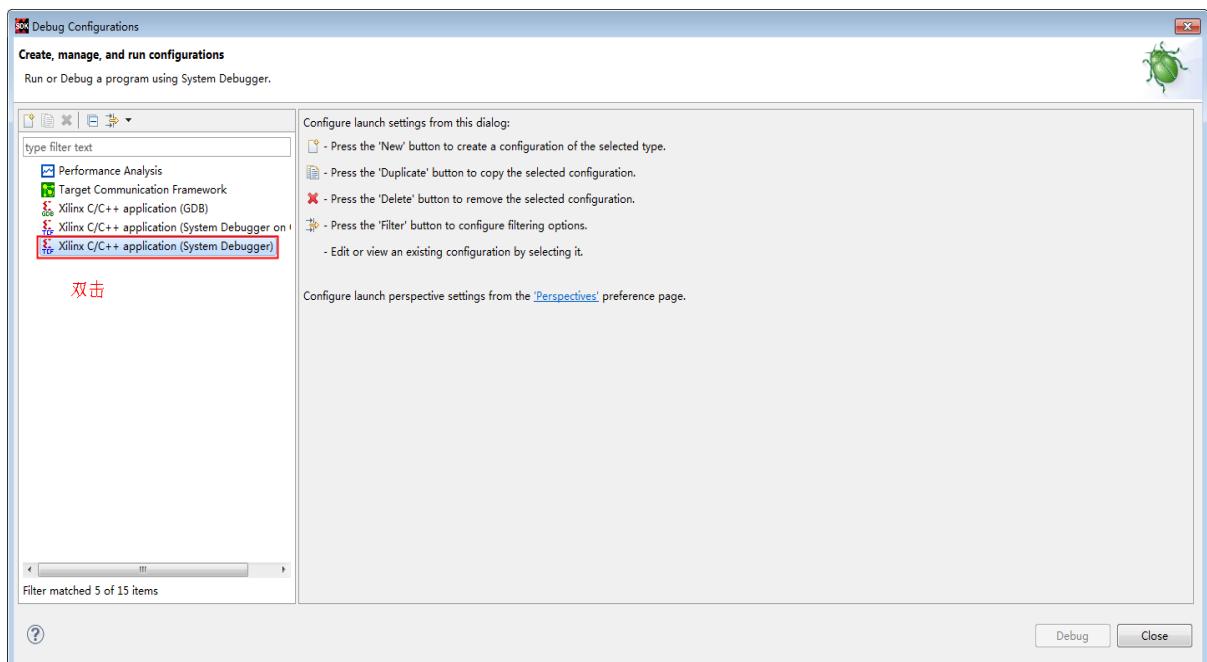
## 13.4 加载到 SDK

Step1: 导出硬件。

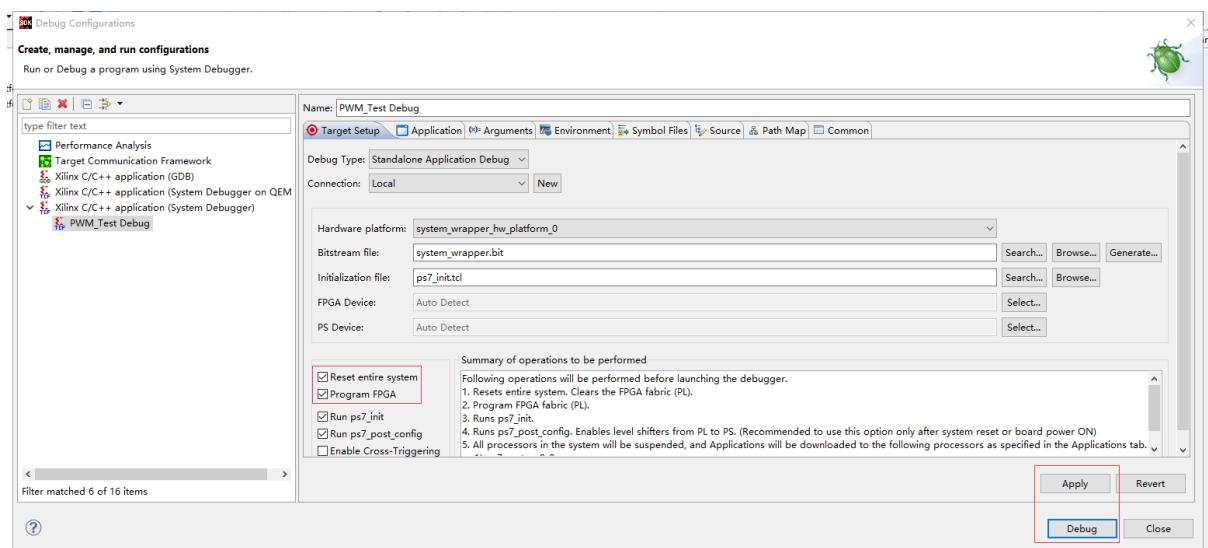
Step2: 新建一个空 SDK 工程，并将我们提供的设计文件复制到工程当中来。

Step3: 右击工程，选择 Debug as ->Debug configuration。

Step4: 选中 system Debugger，双击创建一个系统调试。

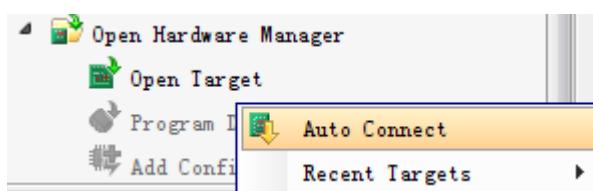


### Step5: 设置系统调试。

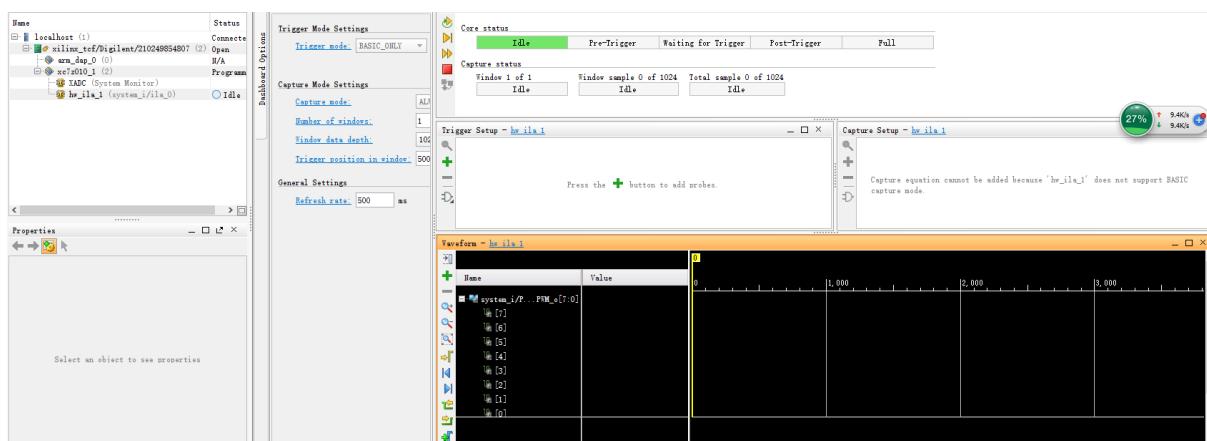


Step6: 单击运行程序按钮 运行程序。

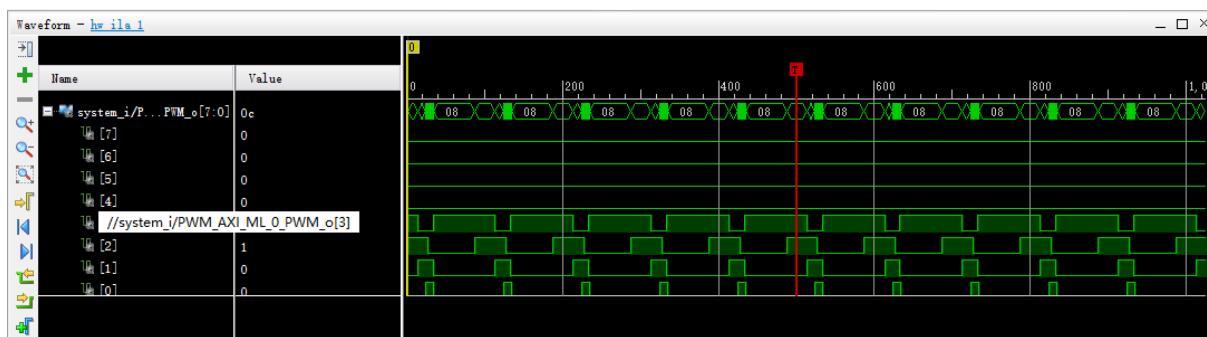
Step7: 回到 VIVADO 单击 Open Target->Auto Connect



Step8: 加载完成后的界面



Step9: 单击箭头所指向启动触发, 窗口显示采集到的信号波形。

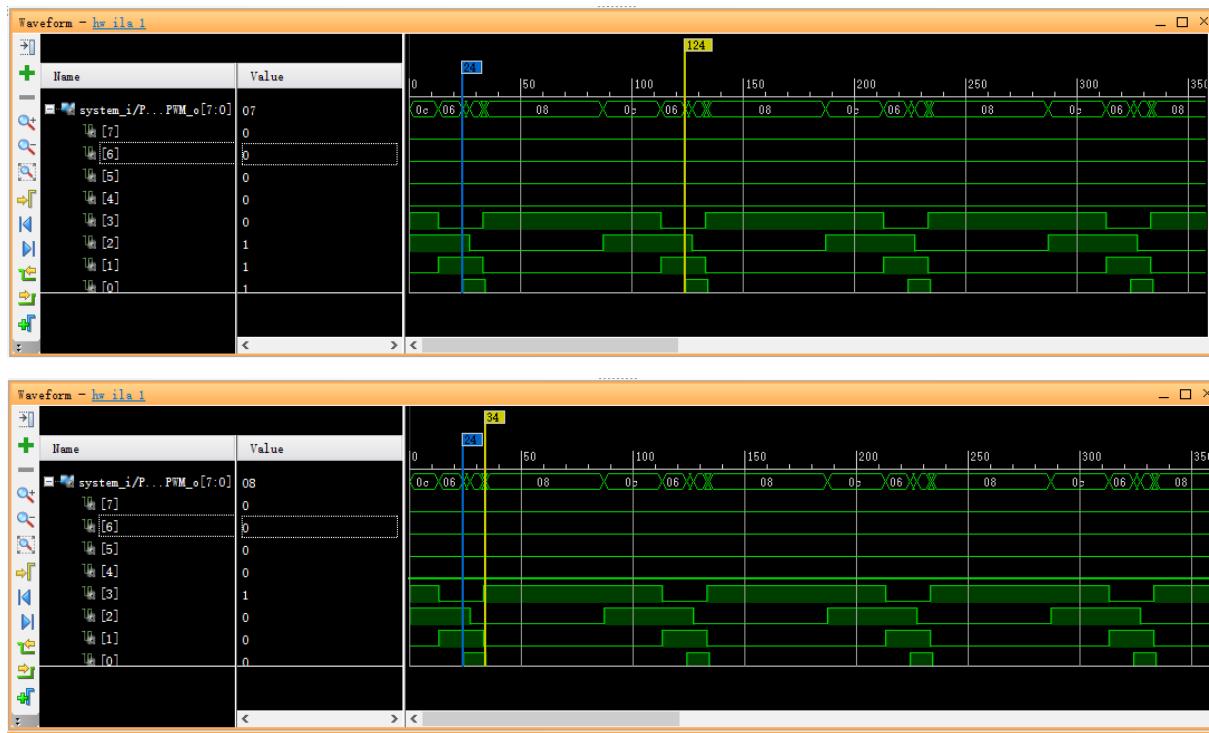


## 13.5 程序分析

Main 函数中, 根据之前章节的讲解我们可知, 此处是分别向 AXI 总线的寄存器中写入数据。在 13.2 小节里, 我们观察到, `pwm_o[0]` 的频率设置和波形设置是通过 `slv_reg0` 和 `slv_reg1` 控制的。

```
// Add user logic here
PWM PWM_O(
    .clk(S_AXI_ACLK),
    .rst_n(S_AXI_ARESETN),
    .fre_set(slv_reg0), // Red arrow pointing here
    .wav_set(slv_reg1), // Red arrow pointing here
    .PWM_o(PWM_o[0])
);
```

由程序可知, 程序中设置的 `pwm_o[0]` 的频率和波形频率分别为 99 和 10, 我们在 ila 中实际测量一下看看波形是否正确 (将黄色测量线拖放到某一点, 然后点击 , 可设立一个参考点)。



由上图可知，我们写入的数据和实际的输出是完全一致的，验证了我们的想法。

## 13.6 本章小结

本章实现了第一个实现具体功能的 8 路 PWM，通过点亮 LED 可以看到效果。这个简单的工程充分体现了 SOC 的优势。CPU 无需参与就可以让 8 路 PWM 持续输出，这个输出是有 PL 控制的

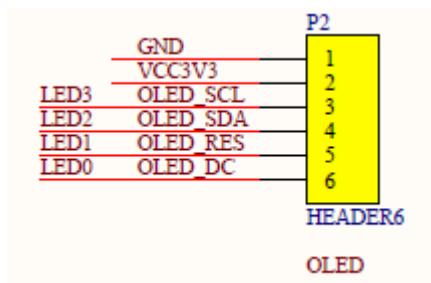
## CH14\_EMIO\_OLED 实验

### 14.1 板载 OLED 硬件原理

MIZ7035 开发板板载 OLED 的型号是 UG-2864HSWEG04，分辨率为 128\*64，接口类型为 4 线 SPI，控制芯片为 SSD1306。本小节，首先简要分析开发板 OLED 相关的硬件电路，然后对 SSD1306 控制器进行介绍，为后续的驱动开发做好铺垫。

#### 14.1.1 硬件电路简析

MZ7XA OLED 接口电路如下图所示。



关键引脚具体说明如下表所示。

引脚名称	详细描述
SCLK	串行时钟线。总线上的数据传输是通过时钟驱动的。每个 bit 的传输都发生在 SCLK 的上升沿。
SDIN	串行数据线。输入数据 (MSB 最先传输) 在 SCLK 上升沿被锁存，在最后一个时钟周期将 8 位串行数据转换为一个 byte 的并行数据。
D/C	数据/命令控制。高电平表示总线上上传输的是数据，低电平表示总线上上传输的是命令。
RES	复位信号。该信号被拉低时，芯片执行复位操作。
CS	片选信号。低电平有效。
VCC	面板驱动电压源。
VDD	控制器电压源。
VSS	地线。
VBAT	内部 DC/DC 电压转换器供电电源。

从原理图中可以看出片选信号 CS 通过电阻短接到 GND，因此该信号是一直有效的；OLED-RES、OLED-DC、OLED-SCLK、OLED-SDIN 直接连接到 Zynq GPIO，其中 RES 和 DC 信号低电平有效；PIN7 VDD 和 PIN5 VBAT 是高电平有效的，但是并非直接连接至 Zynq GPIO，而是通过 PMOS 管进行驱动。根据 PMOS 管的导通特性可以知道，当 OLED\_VBAT

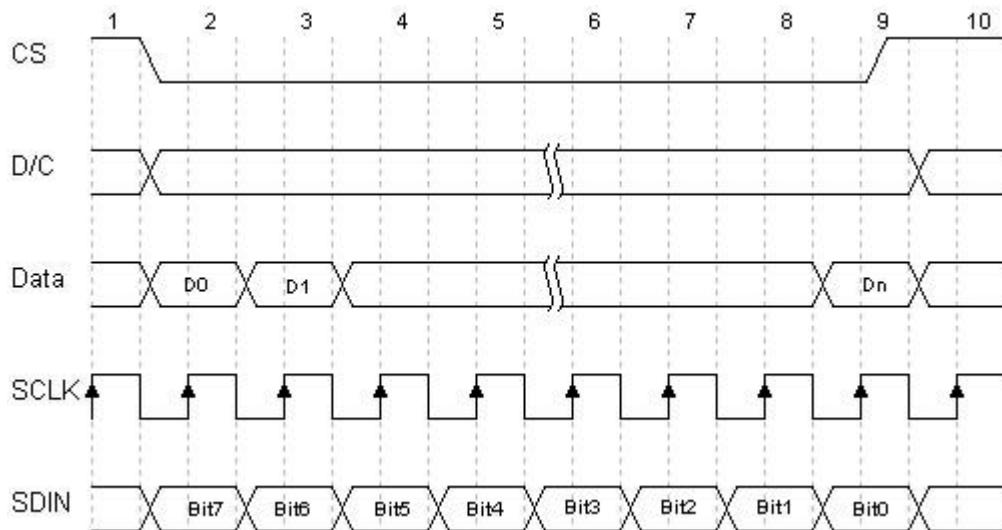
和 OLED-VDD 为低电平时，3.3V 的电压才会送到 VBAT 和 VDD，换句话说，对于 Zynq 而言，VBAT 和 VDD 是低电平有效。市面上大多是将 VBAT 和 VDD 直接连接到高电平，这样就不需要额外的控制，但是功耗也相对高一些。Miz702 和 Miz702N 开发板将 VBAT 和 VDD 连接到 Zynq GPIO，可以通过软件控制 OLED 的通、断电，可以降低整个板子的功耗。

### 14.1.2 SSD1306 简介

SSD1306 是一块内置 CMOS OLED/PLED 驱动控制器的 IC 芯片，芯片可以驱动共阴型 OLED 面板。芯片内部包含晶振、显示 RAM、对比度控制模块以及 256 级亮度控制模块，大大降低了外围元器件数量和功耗。MCU 可以通过 6800/8000 并行接口，I2C 接口或者 SPI 接口实现对 SSD1306 的控制。

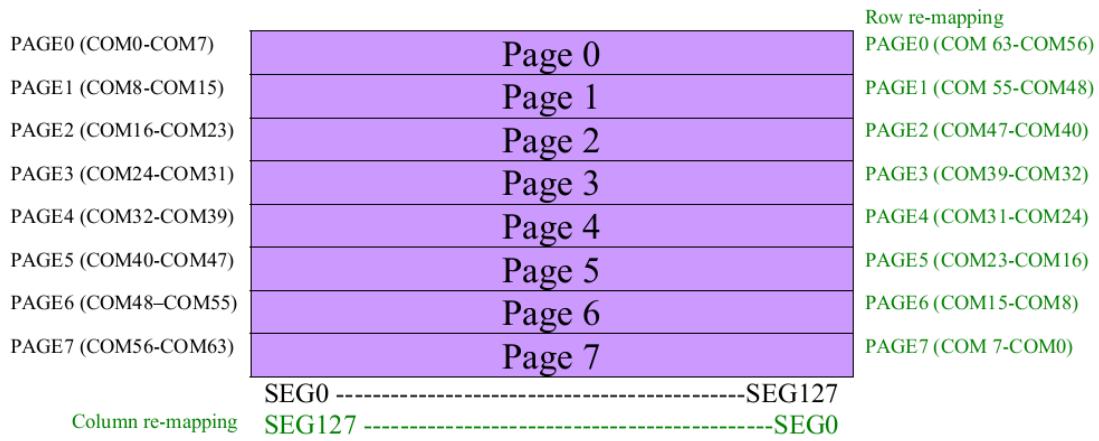
板载 OLED 接口为 4 线串行（SPI）方式，工作在模式下，需要注意的地方有以下几点：

- 使用的信号有以下几个：CS，RES，DC，SCLK，SDIN，各信号作用请参照上一小节，此处不再重复。
- 只能往模块写数据而不能读数据。
- 每个数据长度均为 8 位，在 SCLK 的上升沿，数据从 SDIN 移入到 SSD1306，并且是高位在前的。
- 写操作的时序如下图所示。



4 线 SPI 模式就介绍到这里，时序图是十分重要的，驱动程序和 SPI 相关的函数就是对这个时序图设计的“翻译”。读者在为自己的项目设计电路时，如果用到其他几种接口方式，请自行阅读 SSD1306 数据手册。

接下来，我们介绍一下模块的显存，SSD1306 的显存总共为 128\*64bit 大小，SSD1306 将这些显存分为了 8 页，其对应关系如下：



可以看出，SSD1306 的每页包含了 128 个字节，总共 8 页，这样刚好是 128\*64 的点阵大小。

## 14.2 OLED 驱动开发思路解析

### 14.2.1 SPI 接口

Zynq 和 OLED 通过 SPI 总线连接，想要实现对 OLED 的控制，就必须按照 SPI 接口规范完成数据的传输，相应的我们在驱动实现时要设计出 SPI 接口函数。主要接口函数有以下几个：

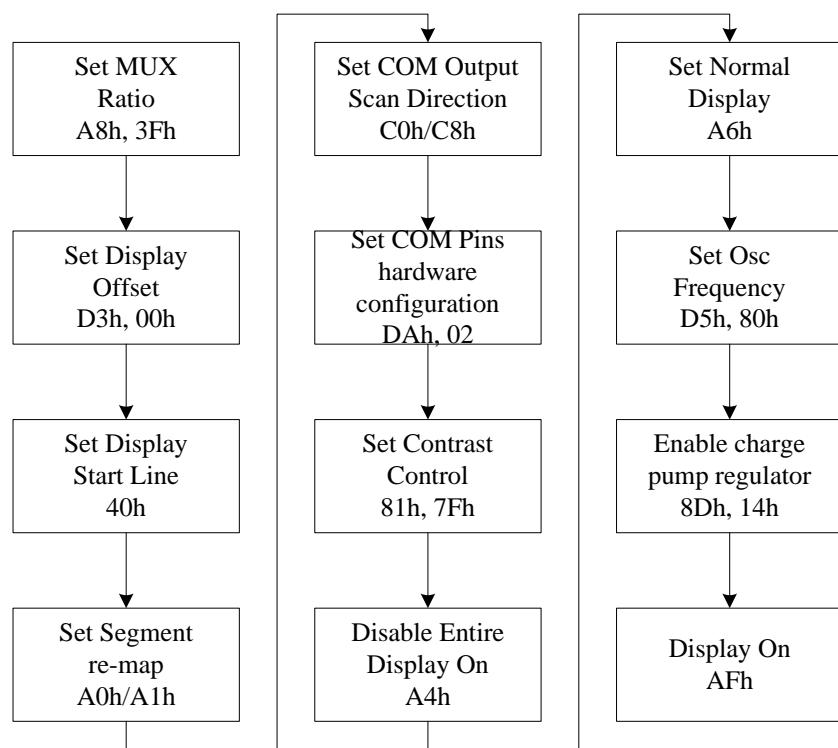
- 写命令
- 写数据

这部分实现难度不大，在驱动实现基础篇参考源码，再结合 18.3.2 的时序图，很容易就能够理解。

### 14.2.2 SSD1306 控制

对 SSD1306 的控制是通过 SPI 接口实现的，实现了基本的写命令和写数据操作之后，就可以轻松地完成 SSD1306 的控制，常用的控制函数有：

- SSD1306 初始化，初始化流程如下图所示：



- 开启显示
- 关闭显示

在实现 SSD1306 的控制之前，有必要了解 SSD1306 常用控制命令，命令分为两种，一种是单字节命令；另一种是非单字节指令，第一个字节是命令字，接下来的一个或多个字节是配置项。现将命令按使用类型分类描述如下：

命令表单 (D/C#=0, R/W#(WR#) = 0, E(RD#=1) 特殊状态除外)

### 1、基本命令

D/C	Hex	D7	D6	D5	D4	D3	D2	D1	D0	命令	描述
0	81 A[7:0]	1 A <sub>7</sub>	0 A <sub>6</sub>	0 A <sub>5</sub>	0 A <sub>4</sub>	0 A <sub>3</sub>	0 A <sub>2</sub>	0 A <sub>1</sub>	1 A <sub>0</sub>	设置对比度	双字节命令, 1~256级对比度可选, 对比度随值增加。 (复位值 = 0x7f)
0	A4/A5	1	0	0	0	0	1	0	X <sub>0</sub>	全部显示开	A4h, X <sub>0</sub> = 0 : 恢复内存内容显示(默认), 输出内存中的内容 A5h, X <sub>0</sub> = 1 : 开显示, 输出无视内存的内容
0	A6/A7	1	0	0	0	0	1	1	X <sub>0</sub>	设置正常 / 逆显示	A6, X[0]= 0: 正常显示(默认) RAM为0: 显示面板关 RAM为1: 显示面板开 A7 X[0]= 1: 逆显示 RAM为0: 显示面板开 RAM为1: 显示面板关
0	AE/AF	1	0	0	0	1	1	1	X <sub>0</sub>	设置显示开 / 关	AE: X[0]= 0: 关显示(默认) AE: X[0]= 1: 在正常模式显示

## 2、寻址设置命令表

D/C	Hex	D7	D6	D5	D4	D3	D2	D1	D0	命令	描述
0	26/27	0	0	1	0	0	1	1	X0	连续水平滚动	26小时, X[0]= 0, 右向水平滚动
0	A[7:0]	0	0	0	0	0	0	0	0	平滚动	27 h, X[0]= 1, 左向水平滚动
0	B[2:0]	*	*	*	*	*	B2	B1	B0	设置	(水平滚动1列)
0	C[2:0]	*	*	*	*	*	C2	C1	C0		[7:0]:虚拟字节(设置为00 h)
0	D[2:0]	*	*	*	*	*	D2	D1	D0		B(2:0):定义开始页面地址
0	E[7:0]	0	0	0	0	0	0	0	0		0~7 PAGE0 ~ PAGE7
0	F[7:0]	1	1	1	1	1	1	1	1		C(2:0):设置每个滚动步骤之间的时间间隔的帧频 000 b - 5帧100 b - 3帧 001 b - 64帧101 b - 4帧 010 b - 128帧110 b - 25帧 011 b - 256帧111 b - 2帧 D(2:0):定义最终页面地址 0~7 PAGE0 ~ PAGE7 D(2:0)的值必须大于或等于B(2:0) E[7:0]:虚拟字节(设置为00 h) F[7:0]:虚拟字节(设置为FFh)
0	2E	0	0	1	0	1	1	1	0	禁用滚动	
0	2F	0	0	1	0	1	1	1	1	激活滚动	

D/C	Hex	D7	D6	D5	D4	D3	D2	D1	D0	命令	描述
0	00~0F	0	0	0	0	X3	X2	X1	X0	设置低的列开始地址页面寻址模式	设置列的低咬起始地址注册页面使用X(握)寻址模式数据位。最初的显示行寄存器复位后重置为0000 b。 请注意 (1) 该命令只是页面寻址模式
0	10~1F	0	0	0	1	X3	X2	X1	X0	设定更高的列开始地址页面寻址模式	设置列的高咬起始地址注册页面使用X(握)寻址模式数据位。最初的显示行寄存器复位后重置为0000 b。 请注意 1) 这个命令只是页面寻址模式
0	20	0	0	1	0	0	0	0	0	设置内存寻址模式	A[1:0]= 00, 水平寻址模式
0	A[1:0]	*	*	*	*	*	*	A1	A0		A[1:0]= 01, 垂直的寻址模式
0	21	0	0	1	0	0	0	0	1	设置列	A[1:0]= 10, 页面寻址模式(重置) A[1:0]= 11, 无效 设置列开始和结束地址

0 0	A[6:0] B[6:0]	*	A <sub>6</sub> B <sub>6</sub>	A <sub>5</sub> B <sub>5</sub>	A <sub>4</sub> B <sub>4</sub>	A <sub>3</sub> B <sub>3</sub>	A <sub>2</sub> B <sub>2</sub>	A <sub>1</sub> B <sub>1</sub>	A <sub>0</sub> B <sub>0</sub>	地址	A[6:0]:列起始地址, 范围:0 - 127 (默认值 = 0) B[6:0]:列结束地址范围:0 - 127 (默认值 = 127) 注: (1)该命令只是为水平或垂直寻址模式。
0 0 0	22 A[2:0] B[2:0]	0 * *	0 * *	1 * *	0 * *	0 * *	0 A <sub>2</sub> B <sub>2</sub>	1 A <sub>1</sub> B <sub>1</sub>	0 A <sub>0</sub> B <sub>0</sub>	设置页面地址	页面设置开始和结束地址 A[2:0]:页面起始地址, 范围:0-7 (默认值= 0 ) B[2:0]:页面结束地址, 范围:0-7 (默认值= 7 ) 注: (1)该命令只是为水平或垂直寻址模式。
0	B0~B7	1	0	1	1	0	X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>	设置页面开始 页面地址 寻址 模式	设置GDDR4页面的起始地址 (PAGE0 ~ PAGE7) 页面寻址模式, 使用X[2:0]。 请注意 (1)该命令只是页面寻址模式

### 3、硬件配置表(面板分辨率&设计相关)命令

0	40~7F	0	1	X <sub>5</sub>	X <sub>4</sub>	X <sub>3</sub>	X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>	设置显示开始行	设置显示RAM的显示起始行地址0 -> 63, 使用X <sub>5</sub> X <sub>4</sub> X <sub>3</sub> X <sub>2</sub> X <sub>1</sub> X <sub>0</sub> 。 在复位后起始行地址为0。
0	A0/A1	1	0	1	0	0	0	0	X <sub>0</sub>	设置段重映射	A0, X[0]= 0:列地址0映射到 SEG0(默认值) A1 X[0]= 1:列地址127映射到SEG0
0 0	A8 A[5:0]	1 * *	0 * *	1 A <sub>5</sub>	0 A <sub>4</sub>	1 A <sub>3</sub>	0 A <sub>2</sub>	0 A <sub>1</sub>	0 A <sub>0</sub>	设置多种比例	MUX比率设置为N + 1 MUX N = A[5:0]:从16MUX到64MUX , 复位 值= 111111 b(即63 d、64 mux) A[5:0]:值0到14是无效的。
0	C0/C8	1	1	0	0	X <sub>3</sub>	0	0	0	设置COM输出扫描方向	C0:X[3]= 0:正常模式(默认值)扫描 COM0->COM(N - 1) C8:X[3]= 1:重映射模式。扫描 COM0(N - 1)->COM0 其中N是MUX比率值
0 0	D3 A[5:0]	1 * *	1 * A <sub>5</sub>	0 A <sub>4</sub>	1 A <sub>3</sub>	0 A <sub>2</sub>	1 A <sub>1</sub>	1 A <sub>0</sub>	设置显示补偿	设置COM垂直移动 0->63 复位后的值为0。	
0 0	DA A[5:4]	1 * *	1 * A <sub>5</sub>	0 A <sub>4</sub>	1 A <sub>4</sub>	0 0	0 0	1 0	1 0	设置COM脚	A[4]= 0, 连续COM脚配置 A[4]= 1, (默认), 可选择COM脚配置 A[5]= 0, (默认), 禁用COM左/右重 映射 A[5]= 1, COM左/右可重映射

#### 4、电荷泵命令表

0 0	8D A[7:0]	1 *	0 *	0 0	0 1	1 0	1 A <sub>2</sub>	0 0	1 0	电荷泵 设置	A[2]= 0, 禁用电荷泵(复位) A[2]= 1, 在显示时使能电荷泵 请注意: 在下列的命令序列之前电荷泵必须启用: 0x8d; 电荷泵设置 0x14, 使能电荷泵 0xAF; 开显示
--------	--------------	--------	--------	--------	--------	--------	---------------------	--------	--------	-----------	--

所以的详细指令可以查阅《SSD1306 说明书》。

#### 14.2.2 Frame Buffer 显示机制

SSD1306 显存是按字节方式写入的，如果我们使用只写方式操作模块，每次要写 8 个点，因此在显示过程中，必须把要点亮的点所在的字节的每个位的状态都搞清楚，否则写入的数据就会覆盖掉之前的状态，造成显示错误。在可读的模式下，在写入之前，可以对待写入字节进行读取，修改需要操作的位之后再写入显存，虽然 1 读 2 改 3 写的操作方式耗时较多，但是不会出现显示错误。

在介绍 SSD1306 时已经说过，对于 3 线或 4 线 SPI 模式，模块是不支持读的。为了解决上述问题，采用的办法是在利用软件创建一个显示缓冲区 frame\_buffer[128][4]，共 512 个字节，也就是 128\*32 个位，对应了 OLED 整个显示区域。在每次修改的时候，只是修改软件内的 frame\_buffer，修改完成之后，一次性把 frame\_buffer 内的数据写入到 SSD1306 内部显存。当然这个方法也有坏处，就是对于那些 SRAM 很小的单片机（比如 51 系列）就比较麻烦了。

#### 14.2.3 像素操作函数

建立起 frame buffer 的显示机制后，只需要能够绘制和擦除像素的函数，就可以点亮和熄灭 OLED 面板上任意一个 LED 了。

像素操作函数并不是必须的，但是可以大大提高驱动的灵活性。比如我们要显示的不是中英文字符这种规律简单的图形，而是用某种算法描绘出来的图形，例如椭圆、正弦波等，采用和字符显示类似的查表操作就不见得是明智的选择了，所以像素操作函数就有了一定的必要性。

此外，像素操作函数为顶层 API 函数提供了一个唯一的 OLED 绘图接口函数，在移植 OLED 驱动时，只要不改动该函数的接口，就不会影响和绘图相关功能，从而便于驱动程序的移植和维护。

#### 14.2.4 其他 API 的实现

虽然提供像素操作函数，就可以实现对 OLED 的操作，但是为了方便用户进行二次

开发，有必要设计一些常见的 API，常用的有以下几个：

- 英文字符显示
- 英文字符串显示
- 中文字符显示

## 14.3 OLED 驱动方案实现

Zynq 与传统 FPGA 最大的区别是芯片内置了 ARM Cortex-A9 双核 CPU，因此基于 Zynq 的设计比基于普通 SoC 或者基于 FPGA 的设计有更多的选择，本小节给出一种实现方案，给读者提供一些设计思路。

基础方案，主要针对那些对 FPGA 开发不太熟悉，从传统 SOC 开发转型 Zynq 开发的设计人员。方案的主要工作均由 PS 完成，涉及 FPGA 的开发很少。

熟悉单片机开发的人都知道，用 IO 模拟总线时序是开发时常用的手段，当然，这是因为单片机资源有限，没有相应的总线接口控制器。随着 MCU 的内置资源越来越丰富，IO 模拟总线时序的方法就显得没那么有必要了。但是 MCU 内置接口控制器也有其缺点和限制，例如硬件接口固定等，外设接口一旦不能完全匹配控制器接口，就不能轻松地使用 MCU 内置接口控制器了。也正是由于这个原因，本方案没有采用 Zynq 的 SPI 控制器，而是采用 EMIO 对总线时序进行模拟。

## 14.4 点阵式 OLED 显示原理

### 14.4.1 OLED 简介

OLED，即有机发光二极管（Organic Light-Emitting Diode），又称为有机电激光显示（Organic Electroluminescence Display, OELD）。OLED 由于同时具备自发光，不需背光源、对比度高、厚度薄、视角广、反应速度快、可用于挠曲性面板、使用温度范围广、构造及制程较简单等优异之特性，被认为是下一代的平面显示器新兴应用技术。

LCD 都需要背光，而 OLED 不需要，因为它是自发光的。这样同样的显示，OLED 效果要来得好一些。OLED 的尺寸难以大型化，但是分辨率确可以做到很高。

### 14.4.2 点阵式显示设备显示原理

在数字世界中，所有数据归根结底都是以 0 和 1 的方式存在的。那么点阵式显示设备是如何将字符、汉字等信息显示出来的呢？抛开 OLED 这种高大上的词不谈，先来看一下最简单的点阵 LED。如下图所示，要显示出图形，只要按照一定的方式点亮点阵上的一部分“点”就可以了，LED 的亮和灭就对应着 1 和 0。



OLED 的显示原理在本质上是相同的，只不过是 LED 间的间隙很小，密度很大，从而显示效果也比上图中的点阵 LED 好很多。对于字符而言，这种表征了点阵开关状态的数据，被抽象成了一个术语，叫做字模。例如英文字母“A”和中文字符“你”的字模信息，如下面两幅图所示。

英文字模	位代码	字模信息
	0 0 0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0 0 0	0x00
	0 0 0 1 0 0 0 0 0	0x10
	0 0 1 1 1 0 0 0 0	0x38
	0 1 1 0 1 1 0 0 0	0x6c
	1 1 0 0 0 1 1 0 0	0xc6
	1 1 0 0 0 1 1 0 0	0xc6
	1 1 1 1 1 1 1 0 0	0xfe
	1 1 0 0 0 1 1 0 0	0xc6
	1 1 0 0 0 1 1 0 0	0xc6
	1 1 0 0 0 1 1 0 0	0xc6
	1 1 0 0 0 1 1 0 0	0xc6
	0 0 0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0 0 0	0x00

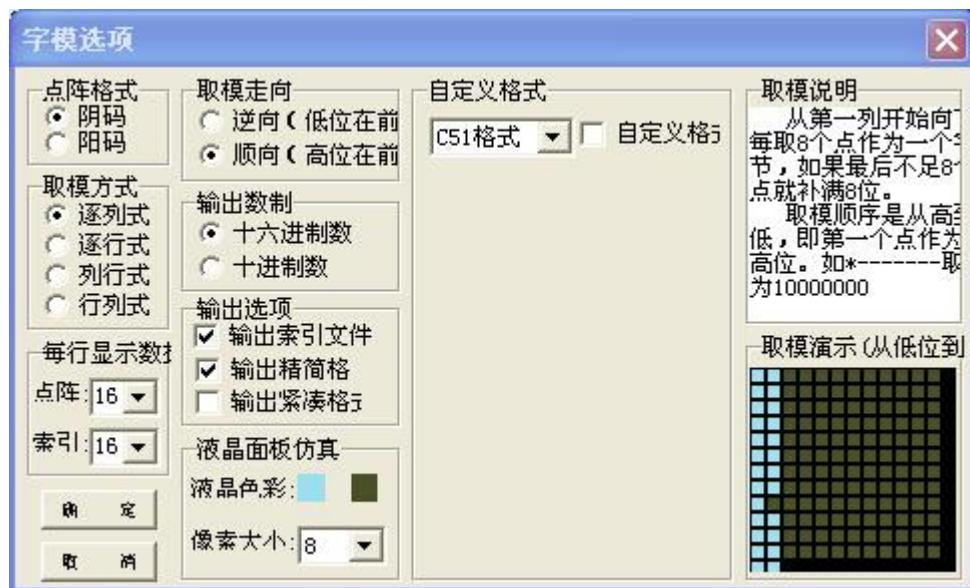
  

中文字模	位代码	字模信息
	0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0	0x08, 0x80
	0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0	0x08, 0x80
	0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0	0x08, 0x80
	0 0 0 1 0 0 0 1 1 1 1 1 1 1 1 0	0x11, 0xfe
	0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0	0x11, 0x02
	0 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0	0x32, 0x04
	0 1 0 1 0 1 0 0 0 0 1 0 0 0 0 0	0x54, 0x20
	0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0	0x10, 0x20
	0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 0	0x10, 0xa8
	0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 0	0x10, 0xa4
	0 0 0 1 0 0 0 1 0 0 1 0 0 1 1 0	0x11, 0x26
	0 0 0 1 0 0 1 0 0 0 1 0 0 0 1 0	0x12, 0x22
	0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0	0x10, 0x20
	0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0	0x10, 0x20
	0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0	0x10, 0xa0
	0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0	0x10, 0x40

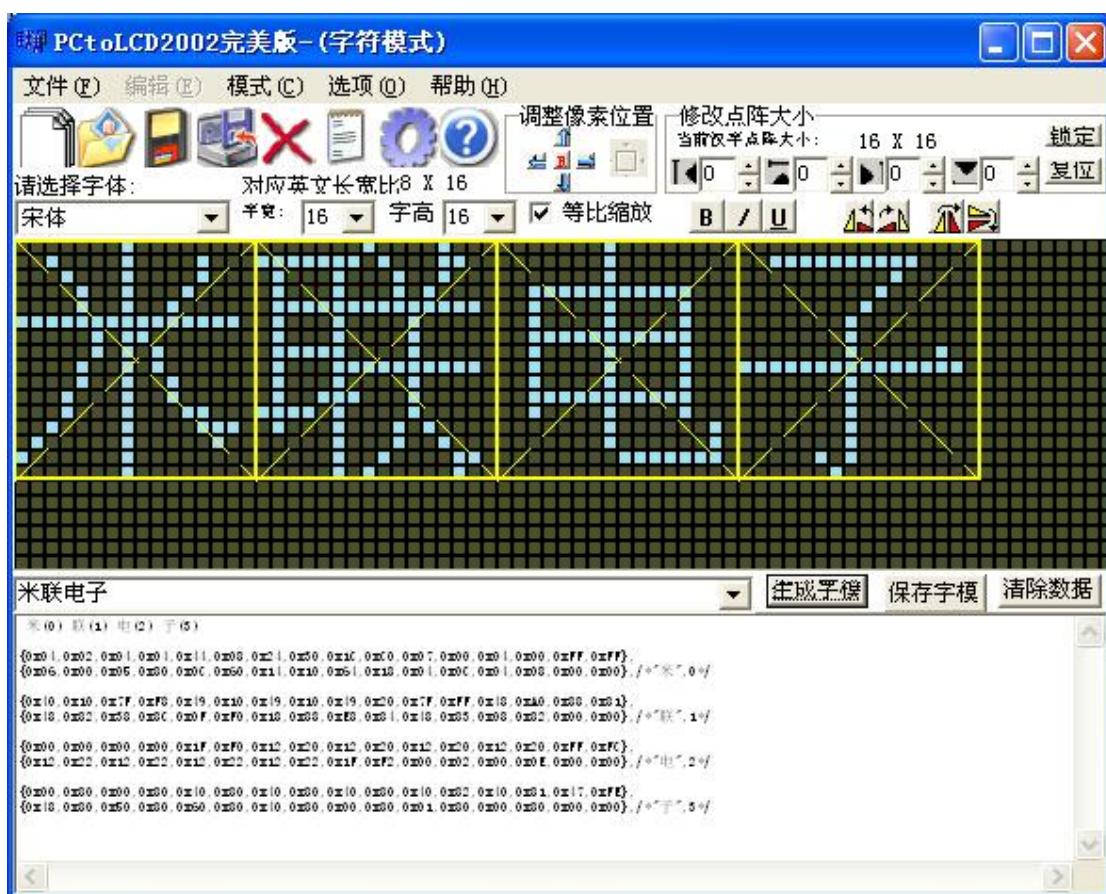
#### 14.4.3 字模的获取

网络上有很多字模获取软件，笔者选用的是 PCtoLCD2002。

点击选项，进入下图所示的参数设置界面，根据自己的需求进行参数设置。



设置好参数后，在字符框中输入字符，然后点击生成字模，就可以获取到字模信息了。如下图所示。



为了更透彻地理解显示原理，笔者首先编写了一个简单的测试程序：

```
const unsigned char HanZi[4][32] =
```

```
{  
// 米(0) 联(1) 电(2) 子(3)  
  
{0x01,0x00,0x21,0x08,0x11,0x08,0x09,0x10,0x09,0x20,0xFF,0xFE,0x05,0x80,0x05,0x40,  
0x09,0x40,0x09,0x20,0x11,0x20,0x11,0x18,0x21,0xE,0x41,0x04,0x81,0x00,0x01,0x00},/*" 米,0*/  
  
{0x01,0x08,0xFE,0x8C,0x44,0x48,0x44,0x50,0x7F,0xFE,0x44,0x20,0x44,0x20,0x7C,0x20,  
0x47,0xFE,0x44,0x20,0x4E,0x20,0xF4,0x20,0x44,0x50,0x04,0x48,0x04,0x86,0x05,0x04},/*" 联,1*/  
  
{0x01,0x00,0x01,0x00,0x01,0x00,0x3F,0xF8,0x21,0x08,0x21,0x08,0x3F,0xF8,0x21,0x08,  
0x21,0x08,0x21,0x08,0x3F,0xF8,0x21,0x08,0x01,0x02,0x01,0x02,0x00,0xFE,0x00,0x00},/*" 电,2*/  
  
{0x00,0x00,0x3F,0xF0,0x00,0x20,0x00,0x40,0x00,0x80,0x01,0x00,0x01,0x00,0x01,0x04,  
0xFF,0xFE,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,0x05,0x00,0x02,0x00},/*" 子,3*/  
};  
  
// led_matrix_disp_test.cpp : Defines the entry point for the console application.  
#include "stdafx.h"  
#include "font.h"  
  
int main(int argc, char* argv[])  
{  
    char i = 0;  
    unsigned char ch_l = 0x0;  
    unsigned char ch_r = 0x0;  
    unsigned char row = 0x0;           // 行  
    unsigned char col = 0x0;          // 列  
  
    for(i=0;i<4;i++)                // 四个汉字  
    {
```

```
for(row=0;row<16;row++)           // 逐行打印
{
    ch_l = HanZi[i][2*row];      // 字符左半边字模
    ch_r = HanZi[i][2*row+1];    // 字符右半边字模
    // 绘制左半边
    for(col=0;col<8;col++)
    {
        if(ch_l&0x80)
            printf("%d",1);
        else
            printf(" ");
    }

    ch_l = ch_l<<1;
}

// 绘制右半边
for(col=0;col<8;col++)
{
    if(ch_r&0x80)
        printf("%d",1);
    else
        printf(" ");

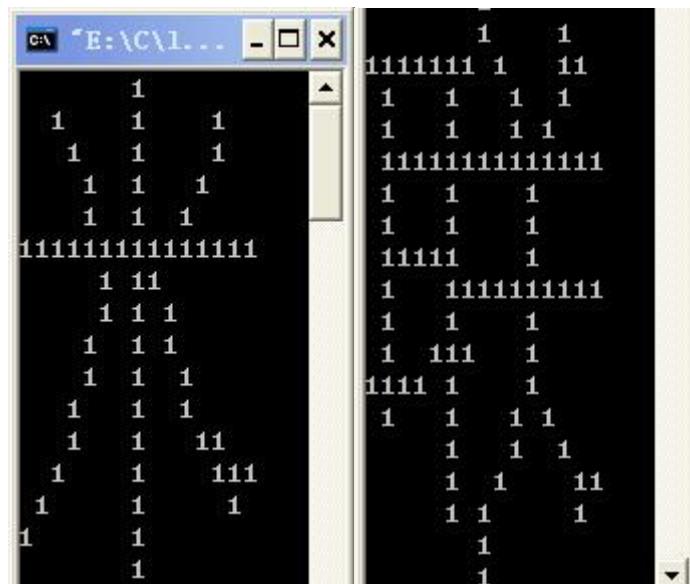
    ch_r = ch_r<<1;
}

// 换行，开始绘制下一行
printf("\n");
}

}

return 0;
}
```

测试结果如下图所示：



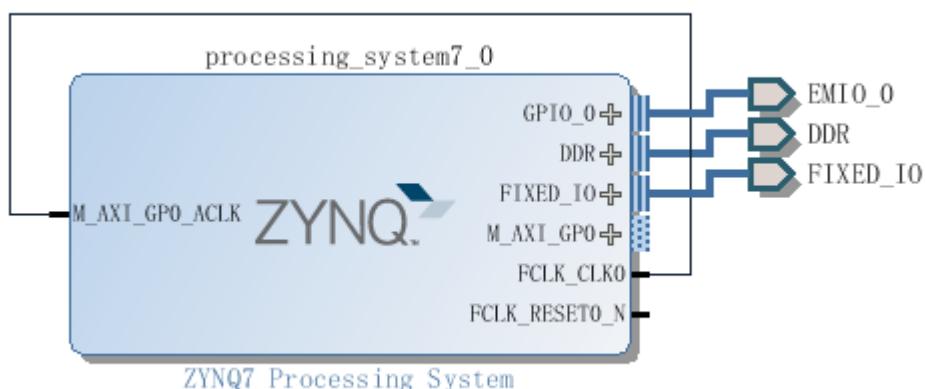
这个测试程序采用的字模是从左至右、从上到下的方式获取的，这是因为要照顾到打印函数的特性，程序难度不大，此处不再逐句解释。后续我们设计的 OLED 驱动虽然和本程序有所区别，但思想上是相同的。

## 14.5 硬件搭建

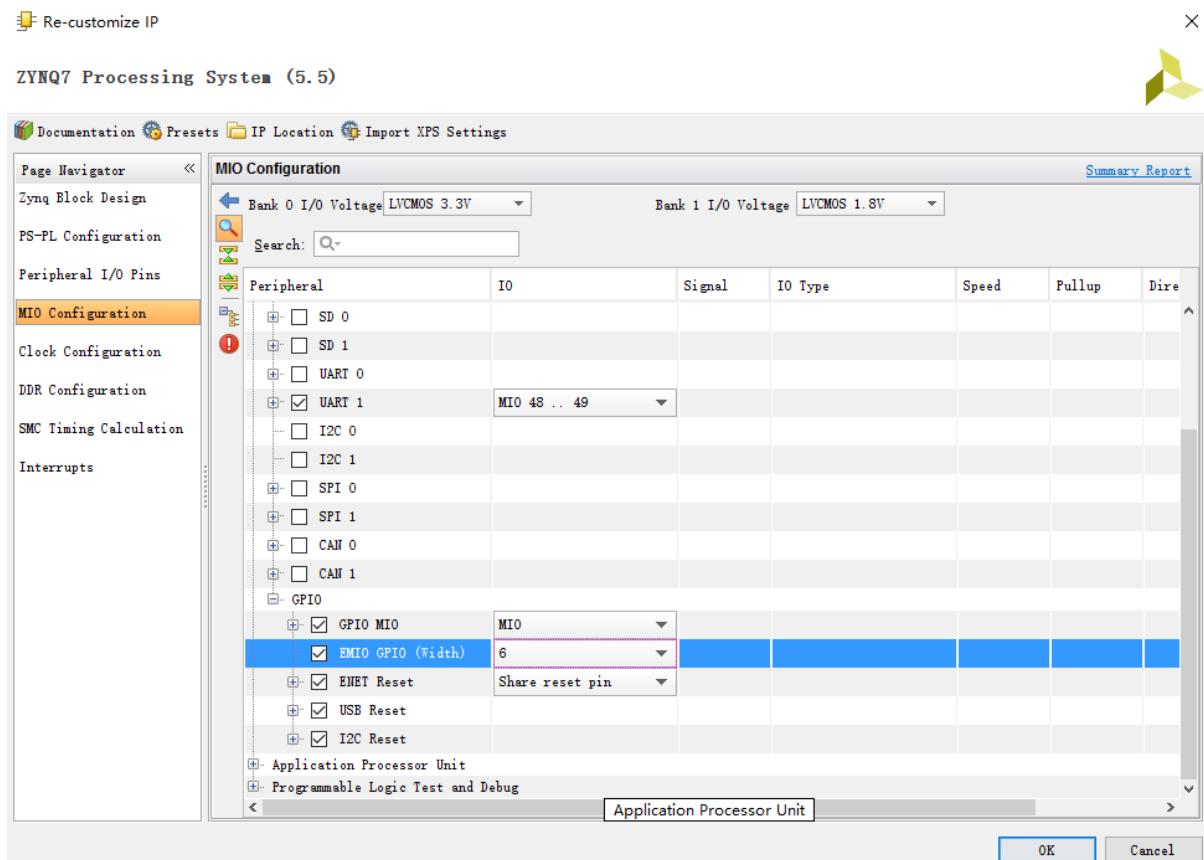
本章的硬件电路与第三章基本一致，因此做好备份后，我们直接使用第三章的工程，对其进行一些细微的修改即可。

Step1：做好备份后，打开第三章的工程。

Step2:双击 ZYNQ Processing System 图标，对其进行一些修改。



Step3:展开 MIO configuration-I/O peripherals-GPIO,将 EMIO 的数量改为 6。



Step4: 右键单击 Block 文件，文件选择 Generate the Output Products。

Step5: 右键单击 Block 文件，选择 Create a HDL wrapper，根据 Block 文件内容产生一个 HDL 的顶层文件，并选择让 vivado 自动完成。

Step6: 添加约束文件，在我们提供的源程序包的 DOC 文件夹下找到 XDC 文件夹，将其中的约束文件添加到工程当中来。Step7:生成 bit 文件。

## 14.6 导入到 SDK

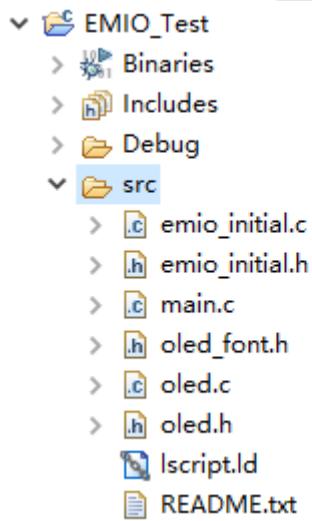
Step1:导出硬件。

Step2:选中第三章的 main.c 文件，右单击，选择 Delete 删除文件。

Step3: 将我们提供的设计文件复制到工程当中来

MIZ7035 > S02 > CH14_EMIO_OLED > CH14_EMIO_OLED > DOC > C_driver			
名称	修改日期	类型	大小
emio_initial.c	2016/11/13 星期...	C 文件	3 KB
emio_initial.h	2016/11/13 星期...	H 文件	1 KB
lscript.ld	2016/11/13 星期...	LD 文件	7 KB
main.c	2016/11/13 星期...	C 文件	1 KB
oled.c	2016/11/13 星期...	C 文件	12 KB
oled.h	2016/11/13 星期...	H 文件	3 KB
oled_font.h	2016/11/13 星期...	H 文件	12 KB
README	2016/11/13 星期...	文本文档	1 KB

Step4: 展开 EMIO\_Test, 在 Src 下按 Ctrl+V 将所有文件粘贴过来。



Step5: 右击工程, 选择 Debug as ->Debug configuration。

Step6: 选中 system Debugger, 双击创建一个系统调试。

Step7: 设置系统调试。

Step8: 单击运行程序按钮 运行程序, 此时可在 OLED 上观察到滚动显示我们定义的字符。

## 14.7 本章小结

本次试验搭进行了 OLED 的驱动, 可以用 OLED 方便的现实必要信息的现实, 例如开发板的运行信息, 时间信息等等。

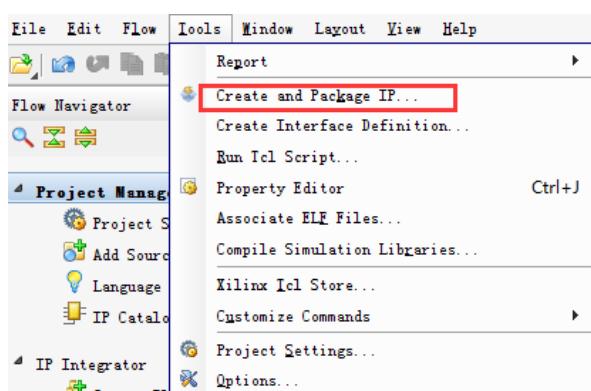
## CH15\_AXI\_OLED 实验

在上一个例子中，主要是以软件功能为主，采用了软件模拟 SPI 时序进行控制 OLED。这样做好处是灵活，但是牺牲了效率。本章采用的方式是让 SPI 驱动由 Verilog 实现，字库也是保存到了 PL 部分的 BRAM 中。这种方式是减轻了 CPU 负担，提高了 CPU 效率。缺点是没有上一章的方法灵活。

### 15.1 自定义 IP 的封装

Step1：新建一个名为 Miz\_sys 空的工程。

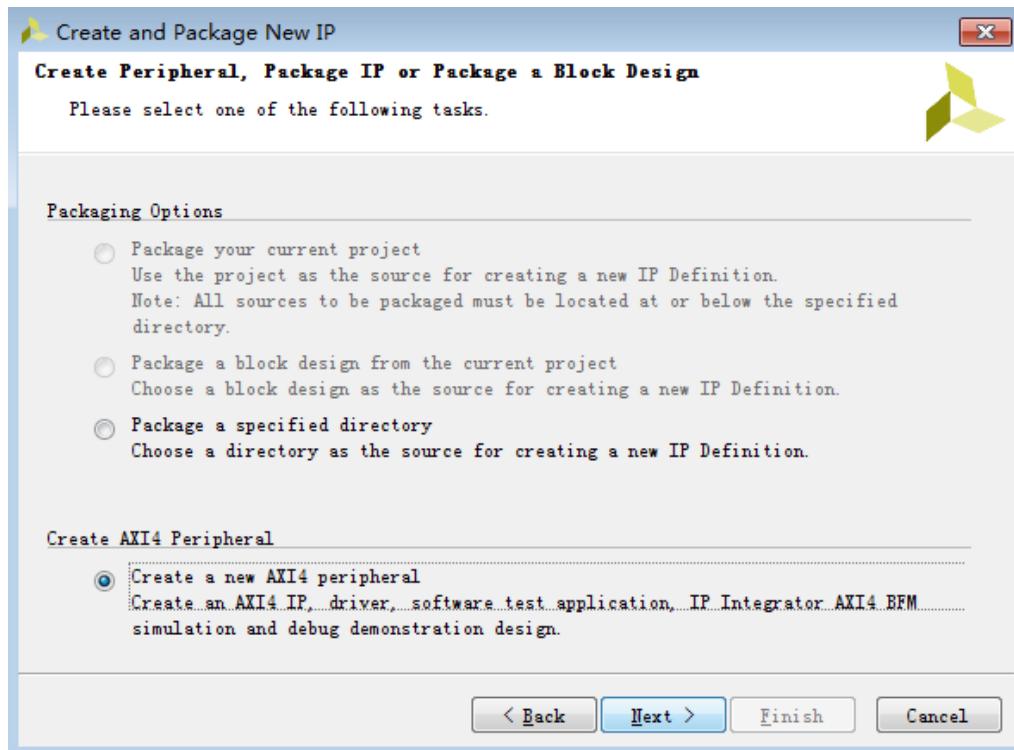
Step2：选择 Tools Create and Package IP 创建 IP



Step3:单击 NEXT

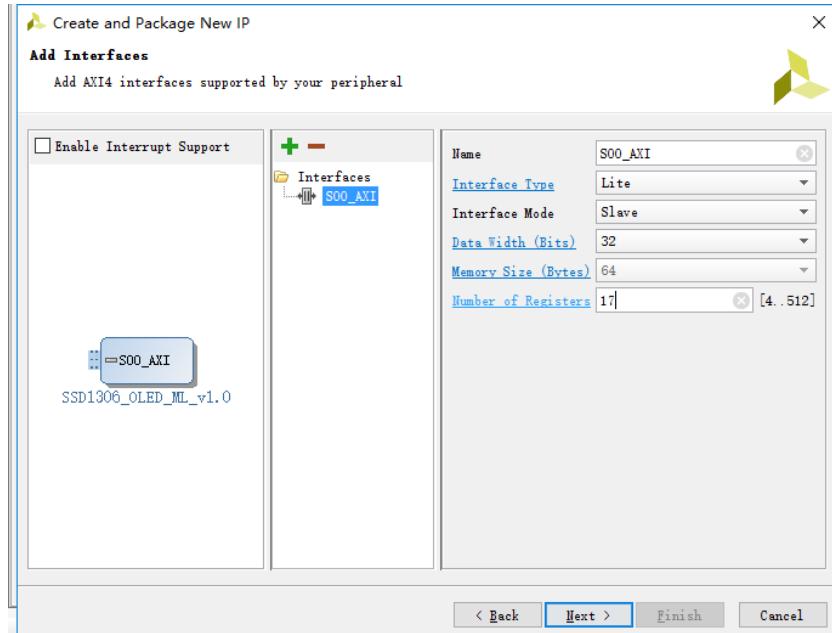


Step4:由于我们需要挂在到总线上，因此创建一个带 AXI 总线的用户 IP

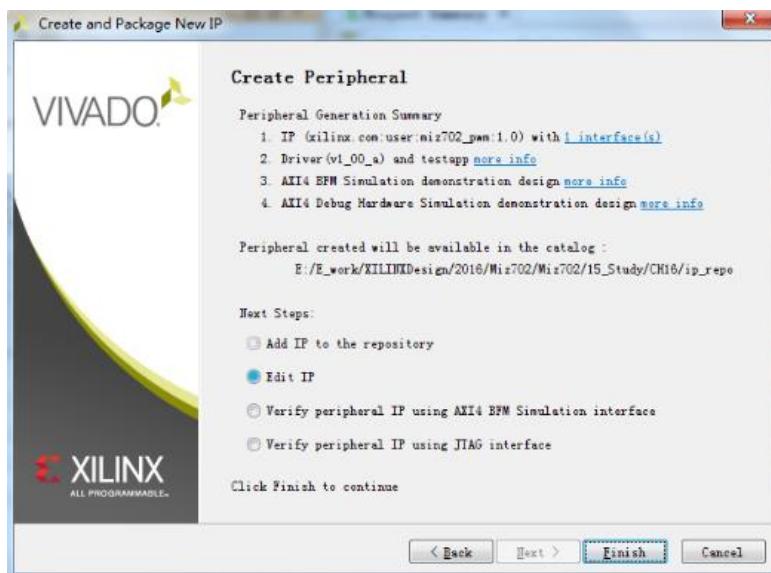


Step5:设置IP的名字为SSD1306\_OLED\_ML版本号默认，并且记住IP的位置

Step6:设置总线形式为Lite总线，Lite总线是简化的AXI总线消耗的资源少，当然性能也是比完全版的AXI总线差一点，但是由于音频的速度并不高，因此采用Lite总线就够了，设置寄存器数量为17，因为后面我们需要用到17个寄存器。



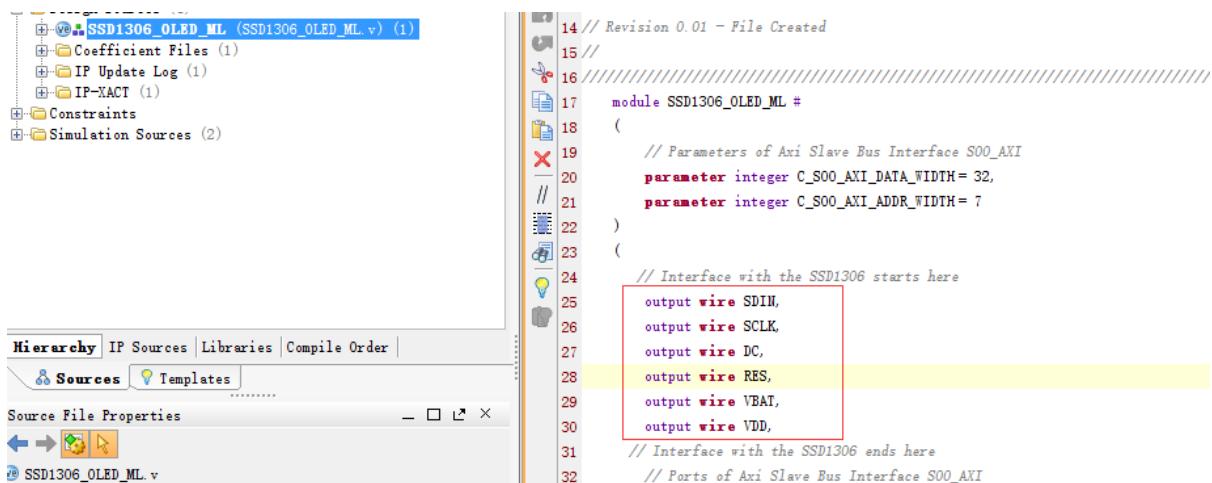
Step7:选择Edit IP单击Finish完成

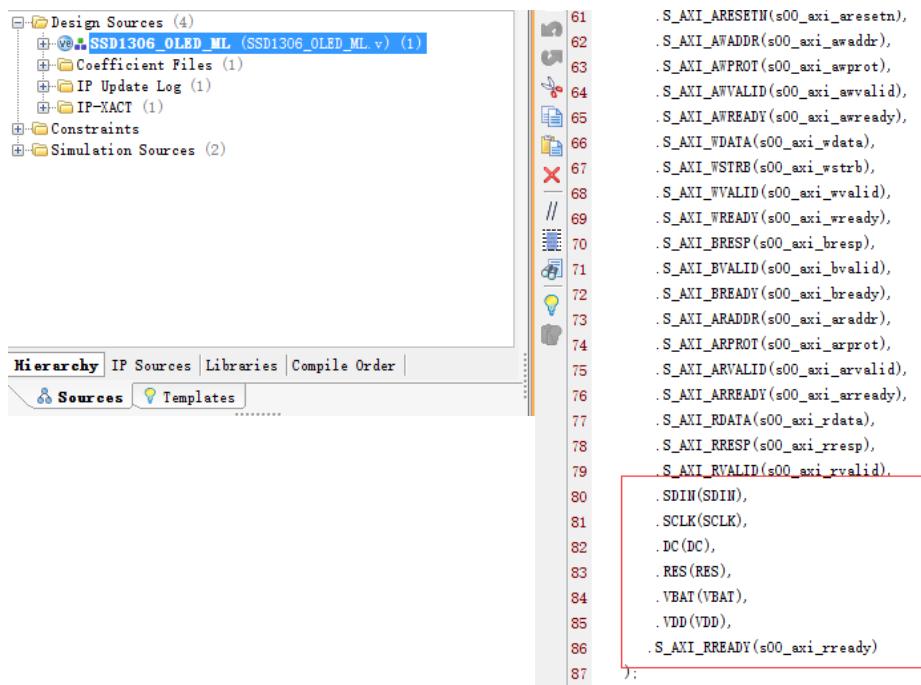


## 15.2 SSD1306\_OLED\_ML 用户 IP 的修改

IP 创建完成后，并不能立马使用，还需要做一些修改。

Step1: 打开 SSD1306\_OLED\_ML.v 文件在以下位置修改:





Step2:用以下程序替代 SSD1306\_OLED\_ML\_v1\_0\_S00\_AXI.v。

```

`timescale 1 ns / 1 ps
///////////////////////////////
//
//
// Create Date: 06:13:25 08/18/2014
// Module Name: SSD1306_OLED_v1_0_S00_AXI
// Project Name: SSD1306_OLED
// Target Devices: Zynq
// Tool versions: Vivado 16.4 (64-bits)
// Description: The core is a slave AXI peripheral with 17 software-accessed registers.
// registers 0-16 are used for data, register 17 is the control register
//
// Revision: 1.0 - SSD1306_OLED_v1_0_S00_AXI completed
// Revision 0.01 - File Created
//
/////////////////////////////
module SSD1306_OLED_v1_0_S00_AXI #
(
    // Width of S_AXI data bus
    parameter integer C_S_AXI_DATA_WIDTH = 32,
    // Width of S_AXI address bus
    parameter integer C_S_AXI_ADDR_WIDTH = 7
)

```

```
(

// Interface with the SSD1306 starts here
    //SPI Data In (MOSI)
    output SDIN,
    //SPI Clock
    output SCLK,
    //Data_Command Control
    output DC,
    //Power Reset
    output RES,
    //Battery Voltage Control - connected to field-effect transistors-active low
    output VBAT,
    // Logic Voltage Control - connected to field-effect transistors-active low
    output VDD,

// Interface with the SSD1306 ends here

    // Global Clock Signal
    input wire S_AXI_ACLK,
    // Global Reset Signal. This Signal is Active LOW
    input wire S_AXI_ARESETN,
    // Write address (issued by master, acceped by Slave)
    input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
    // Write channel Protection type. This signal indicates the
        // privilege and security level of the transaction, and whether
        // the transaction is a data access or an instruction access.
    input wire [2 : 0] S_AXI_AWPROT,
    // Write address valid. This signal indicates that the master signaling
        // valid write address and control information.
    input wire S_AXI_AWVALID,
    // Write address ready. This signal indicates that the slave is ready
        // to accept an address and associated control signals.
    output wire S_AXI_AWREADY,
    // Write data (issued by master, acceped by Slave)
    input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
    // Write strobes. This signal indicates which byte lanes hold
        // valid data. There is one write strobe bit for each eight
        // bits of the write data bus.
    input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,
    // Write valid. This signal indicates that valid write
        // data and strobes are available.
    input wire S_AXI_WVALID,
```

```
// Write ready. This signal indicates that the slave
    // can accept the write data.
output wire  S_AXI_WREADY,
// Write response. This signal indicates the status
    // of the write transaction.
output wire [1 : 0] S_AXI_BRESP,
// Write response valid. This signal indicates that the channel
    // is signaling a valid write response.
output wire  S_AXI_BVALID,
// Response ready. This signal indicates that the master
    // can accept a write response.
input wire  S_AXI_BREADY,
// Read address (issued by master, accepted by Slave)
input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
// Protection type. This signal indicates the privilege
    // and security level of the transaction, and whether the
    // transaction is a data access or an instruction access.
input wire [2 : 0] S_AXI_ARPROT,
// Read address valid. This signal indicates that the channel
    // is signaling valid read address and control information.
input wire  S_AXI_ARVALID,
// Read address ready. This signal indicates that the slave is
    // ready to accept an address and associated control signals.
output wire  S_AXI_ARREADY,
// Read data (issued by slave)
output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
// Read response. This signal indicates the status of the
    // read transfer.
output wire [1 : 0] S_AXI_RRESP,
// Read valid. This signal indicates that the channel is
    // signaling the required read data.
output wire  S_AXI_RVALID,
// Read ready. This signal indicates that the master can
    // accept the read data and response information.
input wire  S_AXI_RREADY
);

// AXI4LITE signals
reg [C_S_AXI_ADDR_WIDTH-1 : 0] axi_awaddr;
reg      axi_awready;
reg      axi_wready;
reg [1 : 0]    axi_bresp;
reg      axi_bvalid;
```

```
reg [C_S_AXI_ADDR_WIDTH-1 : 0] axi_araddr;
reg      axi_arready;
reg [C_S_AXI_DATA_WIDTH-1 : 0]  axi_rdata;
reg [1 : 0]      axi_rresp;
reg      axi_rvalid;

// Example-specific design signals
// local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
// ADDR_LSB is used for addressing 32/64 bit registers/memories
// ADDR_LSB = 2 for 32 bits (n downto 2)
// ADDR_LSB = 3 for 64 bits (n downto 3)
localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
localparam integer OPT_MEM_ADDR_BITS = 4;
//-----
//-- Signals for user logic register space example
//-----
//-- Number of Slave Registers 17
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg0;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg1;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg2;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg3;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg4;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg5;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg6;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg7;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg8;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg9;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg10;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg11;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg12;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg13;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg14;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg15;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg16;
wire slv_reg_rden;
wire slv_reg_wren;
reg [C_S_AXI_DATA_WIDTH-1:0] reg_data_out;
integer byte_index;

// I/O Connections assignments

assign S_AXI_AWREADY = axi_arready;
assign S_AXI_WREADY  = axi_wready;
```

```
assign S_AXI_BRESP = axi_bresp;
assign S_AXI_BVALID = axi_bvalid;
assign S_AXI_ARREADY = axi_arready;
assign S_AXI_RDATA = axi_rdata;
assign S_AXI_RRESP = axi_rresp;
assign S_AXI_RVALID = axi_rvalid;
// Implement axi_awready generation
// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
// de-asserted when reset is low.
```

```
//
```

---

Parameters, Registers, and Wires

---

```
//
```

```
//Current overall state of the state machine
reg [143:0] current_state;
//State to go to after the SPI transmission is finished
reg [111:0] after_state;
//State to go to after the set page sequence
reg [142:0] after_page_state;
//State to go to after sending the character sequence
reg [95:0] after_char_state;
//State to go to after the UpdateScreen is finished
reg [39:0] after_update_state;

//Variable that contains what the screen will be after the next UpdateScreen state
reg [7:0] current_screen[0:3][0:15];
```

```
//Variable assigned to the SSD1306 interface
reg temp_dc = 1'b0;
reg temp_res = 1'b1;
reg temp_vbat = 1'b1;
reg temp_vdd = 1'b1;
assign DC = temp_dc;
assign RES = temp_res;
assign VBAT = temp_vbat;
assign VDD = temp_vdd;
```

```
----- Variables used in the Delay Controller Block -----
wire [11:0] temp_delay_ms; //amount of ms to delay
```

```
reg temp_delay_en = 1'b0; //Enable signal for the delay block
wire temp_delay_fin; //Finish signal for the delay block
assign temp_delay_ms = (after_state == "DispContrast1") ? 12'h074 : 12'h014;

//----- Variables used in the SPI controller block -----
reg temp_spi_en = 1'b0; //Enable signal for the SPI block
reg [7:0] temp_spi_data = 8'h00; //Data to be sent out on SPI
wire temp_spi_fin; //Finish signal for the SPI block

//----- Variables used in the characters libtray -----
reg [7:0] temp_char; //Contains ASCII value for character
reg [10:0] temp_addr; //Contains address to BYTE needed in memory
wire [7:0] temp_dout; //Contains byte outputted from memory
reg [1:0] temp_page; //Current page
reg [3:0] temp_index; //Current character on page

//----- Variables used in the reset and synchronization circuitry -----
reg init_first_r = 1'b1; // Initilaize only one time
reg clear_screen_i = 1'b1; // Clear the screen on start up
reg ready = 1'b0; // Ready flag
reg RST_internal =1'b1;
reg[11:0] count =12'h000;
wire RST_IN;
wire RST=1'b0; // dummy wire - can be connected as a port to provide external reset to the circuit
integer i = 0;
integer j = 0;
assign RST_IN = (RST || RST_internal);

//----- Core commands assignments start -----
wire Display_c;
wire Clear_c;
assign Display_c = slv_reg16[0];
assign Clear_c =slv_reg16[1];

//----- Core commands assignments end -----

always @(posedge S_AXI_ACLK )
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_awready <= 1'b0;
        end

```

```
else
begin
if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
begin
    // slave is ready to accept write address when
    // there is a valid write address and write data
    // on the write address and data bus. This design
    // expects no outstanding transactions.
    axi_awready <= 1'b1;
end
else
begin
    axi_awready <= 1'b0;
end
end

// Implement axi_awaddr latching
// This process is used to latch the address when both
// S_AXI_AWVALID and S_AXI_WVALID are valid.

always @(posedge S_AXI_ACLK)
begin
if (S_AXI_ARESETN == 1'b0)
begin
    axi_awaddr <= 0;
end
else
begin
    if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
    begin
        // Write Address latching
        axi_awaddr <= S_AXI_AWADDR;
    end
end
end

// Implement axi_wready generation
// axi_wready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
// de-asserted when reset is low.

always @(posedge S_AXI_ACLK)
```

```
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_wready <= 1'b0;
        end
    else
        begin
            if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID)
                begin
                    // slave is ready to accept write data when
                    // there is a valid write address and write data
                    // on the write address and data bus. This design
                    // expects no outstanding transactions.
                    axi_wready <= 1'b1;
                end
            else
                begin
                    axi_wready <= 1'b0;
                end
        end
    end

    // Implement memory mapped register select and write logic generation
    // The write data is accepted and written to memory mapped registers when
    // axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes are
    used to
    // select byte enables of slave registers while writing.
    // These registers are cleared when reset (active low) is applied.
    // Slave register write enable is asserted when valid address and data are available
    // and the slave is ready to accept the write address and write data.
    assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID;

    always @(posedge S_AXI_ACLK)
    begin
        if ( S_AXI_ARESETN == 1'b0 )
            begin
                slv_reg0 <= 0;
                slv_reg1 <= 0;
                slv_reg2 <= 0;
                slv_reg3 <= 0;
                slv_reg4 <= 0;
                slv_reg5 <= 0;
                slv_reg6 <= 0;
            end
    end
```

```
slv_reg7 <= 0;
slv_reg8 <= 0;
slv_reg9 <= 0;
slv_reg10 <= 0;
slv_reg11 <= 0;
slv_reg12 <= 0;
slv_reg13 <= 0;
slv_reg14 <= 0;
slv_reg15 <= 0;
slv_reg16 <= 0;
end
else begin
  if (slv_reg_wren)
    begin
      case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        5'h00:
          for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
            if ( S_AXI_WSTRB[byte_index] == 1 ) begin
              // Respective byte enables are asserted as per write strobes
              // Slave register 0
              slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
            end
        5'h01:
          for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
            if ( S_AXI_WSTRB[byte_index] == 1 ) begin
              // Respective byte enables are asserted as per write strobes
              // Slave register 1
              slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
            end
        5'h02:
          for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
            if ( S_AXI_WSTRB[byte_index] == 1 ) begin
              // Respective byte enables are asserted as per write strobes
              // Slave register 2
              slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
            end
        5'h03:
          for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
            if ( S_AXI_WSTRB[byte_index] == 1 ) begin
```

```
// Respective byte enables are asserted as per write strobes
// Slave register 3
    slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
end
5'h04:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 4
        slv_reg4[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
5'h05:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 5
        slv_reg5[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
5'h06:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 6
        slv_reg6[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
5'h07:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 7
        slv_reg7[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
5'h08:
for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 8
        slv_reg8[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
```

```
        end
5'h09:
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
            // Respective byte enables are asserted as per write strobes
            // Slave register 9
            slv_reg9[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
        end
5'h0A:
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
            // Respective byte enables are asserted as per write strobes
            // Slave register 10
            slv_reg10[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
        end
5'h0B:
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
            // Respective byte enables are asserted as per write strobes
            // Slave register 11
            slv_reg11[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
        end
5'h0C:
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
            // Respective byte enables are asserted as per write strobes
            // Slave register 12
            slv_reg12[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
        end
5'h0D:
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
            // Respective byte enables are asserted as per write strobes
            // Slave register 13
            slv_reg13[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
        end
5'h0E:
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
```

```
byte_index+1 )
    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
        // Respective byte enables are asserted as per write strobes
        // Slave register 14
        slv_reg14[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
    end
5'h0F:
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
            // Respective byte enables are asserted as per write strobes
            // Slave register 15
            slv_reg15[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
        end
5'h10:
    for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index =
byte_index+1 )
        if ( S_AXI_WSTRB[byte_index] == 1 ) begin
            // Respective byte enables are asserted as per write strobes
            // Slave register 16
            slv_reg16[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
        end
    default : begin
        slv_reg0 <= slv_reg0;
        slv_reg1 <= slv_reg1;
        slv_reg2 <= slv_reg2;
        slv_reg3 <= slv_reg3;
        slv_reg4 <= slv_reg4;
        slv_reg5 <= slv_reg5;
        slv_reg6 <= slv_reg6;
        slv_reg7 <= slv_reg7;
        slv_reg8 <= slv_reg8;
        slv_reg9 <= slv_reg9;
        slv_reg10 <= slv_reg10;
        slv_reg11 <= slv_reg11;
        slv_reg12 <= slv_reg12;
        slv_reg13 <= slv_reg13;
        slv_reg14 <= slv_reg14;
        slv_reg15 <= slv_reg15;
        slv_reg16 <= slv_reg16;
    end
endcase
end
```

```
end
end

// Implement write response logic generation
// The write response and response valid signals are asserted by the slave
// when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
// This marks the acceptance of address and indicates the status of
// write transaction.

always @(posedge S_AXI_ACLK)
begin
    if (S_AXI_ARESETN == 1'b0)
        begin
            axi_bvalid <= 0;
            axi_bresp <= 2'b0;
        end
    else
        begin
            if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready && S_AXI_WVALID)
                begin
                    // indicates a valid write response is available
                    axi_bvalid <= 1'b1;
                    axi_bresp <= 2'b0; // 'OKAY' response
                end
                // work error responses in future
        end
    else
        begin
            if (S_AXI_BREADY && axi_bvalid)
                //check if bready is asserted while bvalid is high)
                //there is a possibility that bready is always asserted high)
                begin
                    axi_bvalid <= 1'b0;
                end
        end
    end
end

// Implement axi_arready generation
// axi_arready is asserted for one S_AXI_ACLK clock cycle when
// S_AXI_ARVALID is asserted. axi_awready is
// de-asserted when reset (active low) is asserted.
// The read address is also latched when S_AXI_ARVALID is
// asserted. axi_araddr is reset to zero on reset assertion.
```

```
always @(posedge S_AXI_ACLK)
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_arready <= 1'b0;
            axi_araddr  <= 32'b0;
        end
    else
        begin
            if (~axi_arready && S_AXI_ARVALID)
                begin
                    // indicates that the slave has accepted the valid read address
                    axi_arready <= 1'b1;
                    // Read address latching
                    axi_araddr  <= S_AXI_ARADDR;
                end
            else
                begin
                    axi_arready <= 1'b0;
                end
        end
    end

    // Implement axi_arvalid generation
    // axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
    // S_AXI_ARVALID and axi_arready are asserted. The slave registers
    // data are available on the axi_rdata bus at this instance. The
    // assertion of axi_rvalid marks the validity of read data on the
    // bus and axi_rresp indicates the status of read transaction.axi_rvalid
    // is deasserted on reset (active low). axi_rresp and axi_rdata are
    // cleared to zero on reset (active low).
    always @(posedge S_AXI_ACLK)
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            axi_rvalid <= 0;
            axi_rresp  <= 0;
        end
    else
        begin
            if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
                begin
```

```
// Valid read data is available at the read data bus
axi_rvalid <= 1'b1;
axi_rresp  <= 2'b0; // 'OKAY' response
end
else if (axi_rvalid && S_AXI_RREADY)
begin
    // Read data is accepted by the master
    axi_rvalid <= 1'b0;
end
end
end

// Implement memory mapped register select and read logic generation
// Slave register read enable is asserted when valid address is available
// and the slave is ready to accept the read address.
assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
always @(*)
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            reg_data_out <= 0;
        end
    else
        begin
            // Address decoding for reading registers
            case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
                5'h00  : reg_data_out <= slv_reg0;
                5'h01  : reg_data_out <= slv_reg1;
                5'h02  : reg_data_out <= slv_reg2;
                5'h03  : reg_data_out <= slv_reg3;
                5'h04  : reg_data_out <= slv_reg4;
                5'h05  : reg_data_out <= slv_reg5;
                5'h06  : reg_data_out <= slv_reg6;
                5'h07  : reg_data_out <= slv_reg7;
                5'h08  : reg_data_out <= slv_reg8;
                5'h09  : reg_data_out <= slv_reg9;
                5'h0A  : reg_data_out <= slv_reg10;
                5'h0B  : reg_data_out <= slv_reg11;
                5'h0C  : reg_data_out <= slv_reg12;
                5'h0D  : reg_data_out <= slv_reg13;
                5'h0E  : reg_data_out <= slv_reg14;
                5'h0F  : reg_data_out <= slv_reg15;
                5'h10  : reg_data_out <= slv_reg16;
```

```
    default : reg_data_out <= 0;
endcase
end
end

// Output register or memory read data
always @(posedge S_AXI_ACLK)
begin
if (S_AXI_ARESETN == 1'b0)
begin
    axi_rdata <= 0;
end
else
begin
    // When there is a valid read address (S_AXI_ARVALID) with
    // acceptance of read address by the slave (axi_arready),
    // output the read data
    if (slv_reg_rden)
begin
    begin
        axi_rdata <= reg_data_out;      // register read data
    end
end
end
end

//
```

===== Implementation =====

```
SpiCtrl SPI_COMP(
    .CLK(S_AXI_ACLK),
    .RST(RST_IN),
    .SPI_EN(temp_spi_en),
    .SPI_DATA(temp_spi_data),
    .SDO(SDIN),
    .SCLK(SCLK),
    .SPI_FIN(temp_spi_fin)
);
```

```
Delay DELAY_COMP(
    .CLK(S_AXI_ACLK),
    .RST(RST_IN),
```

```
.DELAY_MS(temp_delay_ms),
.DELAY_EN(temp_delay_en),
.DELAY_FIN(temp_delay_fin)
);

charLib CHAR_LIB_COMP(
    .clka(S_AXI_ACLK),
    .addra(temp_addr),
    .douta(temp_dout)
);

// State Machine
always @ (posedge S_AXI_ACLK) begin
    if(RST_IN == 1'b1) begin
        current_state <= "Idle";
        temp_res <= 1'b0;
    end
    else begin
        temp_res <= 1'b1;

        case(current_state)

            // Idle State
            "Idle" : begin
                if(init_first_r == 1'b1) begin
                    temp_dc <= 1'b0; // DC=0 "Commands" , DC=1 "Data"
                    current_state <= "VddOn";
                    init_first_r <= 1'b0; // Don't go over the initialization
more than once
                end

                else begin
                    current_state <= "WaitRequest";
                end
            end

            // Initialization Sequence
            // This should be done only one time when Zedboard starts
            "VddOn" : begin // turn the power on the logic of the display
                temp_vdd <= 1'b0; // remember the power FET transistor for VDD
is active low
                current_state <= "Wait1";
            end
        endcase
    end
end
```

```
end

// 3
"Wait1" : begin
    after_state <= "DispOff";
    current_state <= "Transition3";
end

// 4
"DispOff" : begin
    temp_spi_data <= 8'hAE; // 0xAE= Set Display OFF
    after_state <= "SetClockDiv1";
    current_state <= "Transition1";
end

// 5
"SetClockDiv1" : begin
    temp_spi_data <= 8'hD5; //0xD5
    after_state <= "SetClockDiv2";
    current_state <= "Transition1";
end

// 6
"SetClockDiv2" : begin
    temp_spi_data <= 8'h80; // 0x80
    after_state <= "MultiPlex1";
    current_state <= "Transition1";
end

// 7
"MultiPlex1" : begin
    temp_spi_data <= 8'hA8; //0xA8
    after_state <= "MultiPlex2";
    current_state <= "Transition1";
end

// 8
"MultiPlex2" : begin
    temp_spi_data <= 8'h1F; // 0x1F
    after_state <= "ChargePump1";
    current_state <= "Transition1";
end
```

```
// 9
"ChargePump1" : begin // Access Charge Pump Setting
    temp_spi_data <= 8'h8D; //0x8D
    after_state <= "ChargePump2";
    current_state <= "Transition1";
end

// 10
"ChargePump2" : begin // Enable Charge Pump
    temp_spi_data <= 8'h14; // 0x14
    after_state <= "PreCharge1";
    current_state <= "Transition1";
end

// 11
"PreCharge1" : begin // Access Pre-charge Period Setting
    temp_spi_data <= 8'hD9; // 0xD9
    after_state <= "PreCharge2";
    current_state <= "Transition1";
end

// 12
"PreCharge2" : begin //Set the Pre-charge Period
    temp_spi_data <= 8'hFF; // 0xF1
    after_state <= "VCOMH1";
    current_state <= "Transition1";
end

// 13
"VCOMH1" : begin //Set the Pre-charge Period
    temp_spi_data <= 8'hDB; // 0xF1
    after_state <= "VCOMH2";
    current_state <= "Transition1";
end

// 14
"VCOMH2" : begin //Set the Pre-charge Period
    temp_spi_data <= 8'h40; // 0xF1
    after_state <= "DispContrast1";
    current_state <= "Transition1";
end
```

```
// 15
"DispContrast1" : begin //Set Contrast Control for BANK0
    temp_spi_data <= 8'h81; // 0x81
    after_state <= "DispContrast2";
    current_state <= "Transition1";
end

// 16
"DispContrast2" : begin
    temp_spi_data <= 8'hF1; // 0x0F
    after_state <= "InvertDisp1";
    current_state <= "Transition1";
end

// 17
"InvertDisp1" : begin
    temp_spi_data <= 8'hA0; // 0xA1
    after_state <= "InvertDisp2";
    current_state <= "Transition1";
end

// 18
"InvertDisp2" : begin
    temp_spi_data <= 8'hC0; // 0xC0
    after_state <= "ComConfig1";
    current_state <= "Transition1";
end

// 19
"ComConfig1" : begin
    temp_spi_data <= 8'hDA; // 0xDA
    after_state <= "ComConfig2";
    current_state <= "Transition1";
end

// 20
"ComConfig2" : begin
    temp_spi_data <= 8'h02; // 0x02
    after_state <= "VbatOn";
    current_state <= "Transition1";
end
```

```
// 21

"VbatOn" : begin
    temp_vbat <= 1'b0;
    current_state <= "Wait3";
end

// 22
"Wait3" : begin
    after_state <= "ResetOn";
    current_state <= "Transition3";
end

// 23
"ResetOn" : begin
    temp_res <= 1'b0;
    current_state <= "Wait2";
end

// 24
"Wait2" : begin
    after_state <= "ResetOff";
    current_state <= "Transition3";
end

// 25
"ResetOff" : begin
    temp_res <= 1'b1;
    current_state <= "WaitRequest";
end

// ***** END Initialization sequence but without turnning the
display on *****

// Main state
"WaitRequest" : begin
    if(Display_c == 1'b1) begin
        current_state <= "ClearDC";
        after_page_state <= "ReadRegisters";
        temp_page <= 2'b00;
    end
    else if ((Clear_c==1'b1) || (clear_screen_i == 1'b1)) begin
```

```
        current_state <= "ClearDC";
        after_page_state <= "ClearScreen";
        temp_page <= 2'b00;
    end

    else begin
        current_state<="WaitRequest"; // keep looping in the
WaitRequest state until you receive a command

        if ((clear_screen_i == 1'b0) && (ready ==1'b0)) begin // this part is only executed once, on start-up
            temp_spi_data <= 8'hAF; // 0xAF // Dispaly ON
            after_state <= "WaitRequest";
            current_state <= "Transition1";
            temp_dc<=1'b0;
            ready <= 1'b1;
        end
    end

end

//Update Page states
//1. Sets DC to command mode
//2. Sends the SetPage Command
//3. Sends the Page to be set to
//4. Sets the start pixel to the left column
//5. Sets DC to data mode
"ClearDC" : begin
    temp_dc <= 1'b0;
    current_state <= "SetPage";
end

"SetPage" : begin
    temp_spi_data <= 8'b00100010;
    after_state <= "PageNum";
    current_state <= "Transition1";
end

"PageNum" : begin
    temp_spi_data <= {6'b000000,temp_page};
    after_state <= "LeftColumn1";
    current_state <= "Transition1";

```

```
end

"LeftColumn1" : begin
    temp_spi_data <= 8'b00000000;
    after_state <= "LeftColumn2";
    current_state <= "Transition1";
end

"LeftColumn2" : begin
    temp_spi_data <= 8'b00010000;
    after_state <= "SetDC";
    current_state <= "Transition1";
end

"SetDC" : begin
    temp_dc <= 1'b1;
    current_state <= after_page_state;
end

"ClearScreen" : begin
    for(i = 0; i <= 3 ; i=i+1) begin
        for(j = 0; j <= 15 ; j=j+1) begin
            current_screen[i][j] <= 8'h20;
        end
    end
    after_update_state <= "WaitRequest";
    current_state <= "UpdateScreen";
end

"ReadRegisters" : begin
    // Page0
    current_screen[0][0]<=slv_reg0[7:0];
    current_screen[0][1]<=slv_reg0[15:8];
    current_screen[0][2]<=slv_reg0[23:16];
    current_screen[0][3]<=slv_reg0[31:24];
    current_screen[0][4]<=slv_reg1[7:0];
    current_screen[0][5]<=slv_reg1[15:8];
    current_screen[0][6]<=slv_reg1[23:16];
    current_screen[0][7]<=slv_reg1[31:24];
    current_screen[0][8]<=slv_reg2[7:0];
    current_screen[0][9]<=slv_reg2[15:8];

```

```
current_screen[0][10]<=slv_reg2[23:16];
current_screen[0][11]<=slv_reg2[31:24];
current_screen[0][12]<=slv_reg3[7:0];
current_screen[0][13]<=slv_reg3[15:8];
current_screen[0][14]<=slv_reg3[23:16];
current_screen[0][15]<=slv_reg3[31:24];
//Page1
current_screen[1][0]<=slv_reg4[7:0];
current_screen[1][1]<=slv_reg4[15:8];
current_screen[1][2]<=slv_reg4[23:16];
current_screen[1][3]<=slv_reg4[31:24];
current_screen[1][4]<=slv_reg5[7:0];
current_screen[1][5]<=slv_reg5[15:8];
current_screen[1][6]<=slv_reg5[23:16];
current_screen[1][7]<=slv_reg5[31:24];
current_screen[1][8]<=slv_reg6[7:0];
current_screen[1][9]<=slv_reg6[15:8];
current_screen[1][10]<=slv_reg6[23:16];
current_screen[1][11]<=slv_reg6[31:24];
current_screen[1][12]<=slv_reg7[7:0];
current_screen[1][13]<=slv_reg7[15:8];
current_screen[1][14]<=slv_reg7[23:16];
current_screen[1][15]<=slv_reg7[31:24];
//Page2
current_screen[2][0]<=slv_reg8[7:0];
current_screen[2][1]<=slv_reg8[15:8];
current_screen[2][2]<=slv_reg8[23:16];
current_screen[2][3]<=slv_reg8[31:24];
current_screen[2][4]<=slv_reg9[7:0];
current_screen[2][5]<=slv_reg9[15:8];
current_screen[2][6]<=slv_reg9[23:16];
current_screen[2][7]<=slv_reg9[31:24];
current_screen[2][8]<=slv_reg10[7:0];
current_screen[2][9]<=slv_reg10[15:8];
current_screen[2][10]<=slv_reg10[23:16];
current_screen[2][11]<=slv_reg10[31:24];
current_screen[2][12]<=slv_reg11[7:0];
current_screen[2][13]<=slv_reg11[15:8];
current_screen[2][14]<=slv_reg11[23:16];
current_screen[2][15]<=slv_reg11[31:24];
//Page3
current_screen[3][0]<=slv_reg12[7:0];
current_screen[3][1]<=slv_reg12[15:8];
```

```
current_screen[3][2]<=slv_reg12[23:16];
current_screen[3][3]<=slv_reg12[31:24];
current_screen[3][4]<=slv_reg13[7:0];
current_screen[3][5]<=slv_reg13[15:8];
current_screen[3][6]<=slv_reg13[23:16];
current_screen[3][7]<=slv_reg13[31:24];
current_screen[3][8]<=slv_reg14[7:0];
current_screen[3][9]<=slv_reg14[15:8];
current_screen[3][10]<=slv_reg14[23:16];
current_screen[3][11]<=slv_reg14[31:24];
current_screen[3][12]<=slv_reg15[7:0];
current_screen[3][13]<=slv_reg15[15:8];
current_screen[3][14]<=slv_reg15[23:16];
current_screen[3][15]<=slv_reg15[31:24];

after_update_state <= "WaitRequest";
current_state <= "UpdateScreen";
end

//UpdateScreen State
//1. Gets ASCII value from current_screen at the current page and the
current spot of the page
//2. If on the last character of the page transition update the page
number, if on the last page(3)
//           then the updateScreen go to "after_update_state" after
"UpdateScreen" : begin

    temp_char <= current_screen[temp_page][temp_index];

    if(temp_index == 'd15) begin

        temp_index <= 'd0;
        temp_page <= temp_page + 1'b1;
        after_char_state <= "ClearDC";

        if(temp_page == 2'b11) begin
            after_page_state <= after_update_state;
            clear_screen_i<=1'b0;
        end
        else begin
            after_page_state <= "UpdateScreen";
        end
    end
end
```

```
else begin

    temp_index <= temp_index + 1'b1;
    after_char_state <= "UpdateScreen";

end

current_state <= "SendChar1";

end

//Send Character States
//1. Sets the Address to ASCII value of char with the counter appended
to the end
//2. Waits a clock for the data to get ready by going to ReadMem and
ReadMem2 states
//3. Send the byte of data given by the block Ram
//4. Repeat 7 more times for the rest of the character bytes
"SendChar1" : begin
    temp_addr <= {temp_char, 3'b000};
    after_state <= "SendChar2";
    current_state <= "ReadMem";
end

"SendChar2" : begin
    temp_addr <= {temp_char, 3'b001};
    after_state <= "SendChar3";
    current_state <= "ReadMem";
end

"SendChar3" : begin
    temp_addr <= {temp_char, 3'b010};
    after_state <= "SendChar4";
    current_state <= "ReadMem";
end

"SendChar4" : begin
    temp_addr <= {temp_char, 3'b011};
    after_state <= "SendChar5";
    current_state <= "ReadMem";
end

"SendChar5" : begin
```

```
temp_addr <= {temp_char, 3'b100};  
after_state <= "SendChar6";  
current_state <= "ReadMem";  
end  
  
"SendChar6" : begin  
    temp_addr <= {temp_char, 3'b101};  
    after_state <= "SendChar7";  
    current_state <= "ReadMem";  
end  
  
"SendChar7" : begin  
    temp_addr <= {temp_char, 3'b110};  
    after_state <= "SendChar8";  
    current_state <= "ReadMem";  
end  
  
"SendChar8" : begin  
    temp_addr <= {temp_char, 3'b111};  
    after_state <= after_char_state;  
    current_state <= "ReadMem";  
end  
  
"ReadMem" : begin  
    current_state <= "ReadMem2";  
end  
  
"ReadMem2" : begin  
    temp_spi_data <= temp_dout;  
    current_state <= "Transition1";  
end  
  
// SPI transitions  
// 1. Set SPI_EN to 1  
// 2. Waits for SpiCtrl to finish  
// 3. Goes to clear state (Transition5)  
"Transition1" : begin  
    temp_spi_en <= 1'b1;  
    current_state <= "Transition2";  
end
```

```
"Transition2" : begin
    if(temp_spi_fin == 1'b1) begin
        current_state <= "Transition5";
    end
end

// Delay Transitions
// 1. Set DELAY_EN to 1
// 2. Waits for Delay to finish
// 3. Goes to Clear state (Transition5)
"Transition3" : begin
    temp_delay_en <= 1'b1;
    current_state <= "Transition4";
end

"Transition4" : begin
    if(temp_delay_fin == 1'b1) begin
        current_state <= "Transition5";
    end
end

// Clear transition
// 1. Sets both DELAY_EN and SPI_EN to 0
// 2. Go to after state
"Transition5" : begin
    temp_spi_en <= 1'b0;
    temp_delay_en <= 1'b0;
    current_state <= after_state;
end

default : current_state <= "Idle";

endcase
end
end

// Internal reset generator
always @(posedge S_AXI_ACLK) begin
if (RST_IN == 1'b1)
    count<=count+1'b1;
    if (count == 12'hFFF) begin
        RST_internal <=1'b0;
```

```
        end  
    end  
  
endmodule
```

Step3:添加一个 SPI 控制器源码 SpiCtrl.v 文件，代码如下所示:

```
`timescale 1ns / 1ps  
////////////////////////////////////////////////////////////////////////  
//  
//  
//  
//  
//  
// Create Date:      12:12:51 08/04/2014  
// Module Name:     SpiCtrl  
// Project Name:    AXIOLED  
// Target Devices: Zynq  
// Tool versions:   Vivado 16.4 (64-bits)  
// Description: Spi block that sends SPI data formatted SCLK active low with  
//                 SDO changing on the falling edge  
//  
// Revision: 1.0 - SPI completed  
// Revision 0.01 - File Created  
//  
////////////////////////////////////////////////////////////////////////  
module SpiCtrl(  
    CLK,  
    RST,  
    SPI_EN,  
    SPI_DATA,  
    SDO,  
    SCLK,  
    SPI_FIN  
);  
  
//  
=====  
//                                         Port Declarations  
//  
=====  
    input CLK;  
    input RST;  
    input SPI_EN;
```

```
input [7:0] SPI_DATA;
output SDO;
output SCLK;
output SPI_FIN;

//=====

// Parameters, Registers, and Wires
//=====

wire SDO, SCLK, SPI_FIN;

reg [39:0] current_state = "Idle";           // Signal for state machine

reg [7:0] shift_register = 8'h00;             // Shift register to shift out SPI_DATA saved when SPI_EN
was set

reg [3:0] shift_counter = 4'h0;               // Keeps track how many bits were sent
wire clk_divided;                           // Used as SCLK
reg [4:0] counter = 5'b00000;                // Count clocks to be used to divide CLK
reg temp_sdo = 1'b1;                         // Tied to SDO

reg falling = 1'b0;                          // signal indicating that the clk has just fell

//=====

// Implementation
//=====

assign clk_divided = ~counter[4];
assign SCLK = clk_divided;
assign SDO = temp_sdo;

assign SPI_FIN = (current_state == "Done") ? 1'b1 : 1'b0;

// State Machine
always @(posedge CLK) begin
    if(RST == 1'b1) begin                      // Synchronous RST
        current_state <= "Idle";
    end
    else begin
        case(current_state)
```

```
// Wait for SPI_EN to go high
"Idle" : begin
    if(SPI_EN == 1'b1) begin
        current_state <= "Send";
    end
end

// Start sending bits, transition out when all bits are sent and SCLK is high
"Send" : begin
    if(shift_counter == 4'h8 && falling == 1'b0) begin
        current_state <= "Done";
    end
end

// Finish SPI transmission wait for SPI_EN to go low
"Done" : begin
    if(SPI_EN == 1'b0) begin
        current_state <= "Idle";
    end
end

default : current_state <= "Idle";

endcase
end
end
// End of State Machine

// Clock Divider
always @(posedge CLK) begin
    // start clock counter when in send state
    if(current_state == "Send") begin
        counter <= counter + 1'b1;
    end
    // reset clock counter when not in send state
    else begin
        counter <= 5'b00000;
    end
end
// End Clock Divider
```

```

// SPI_SEND_BYT, sends SPI data formatted SCLK active low with SDO changing on the
falling edge
always @(posedge CLK) begin
    if(current_state == "Idle") begin
        shift_counter <= 4'h0;
        // keeps placing SPI_DATA into shift_register so that when state goes to send it
has the latest SPI_DATA
        shift_register <= SPI_DATA;
        temp_sdo <= 1'b1;
    end
    else if(current_state == "Send") begin
        // if on the falling edge of Clk_divided
        if(clk_divided == 1'b0 && falling == 1'b0) begin
            // Indicate that it is passed the falling edge
            falling <= 1'b1;
            // send out the MSB
            temp_sdo <= shift_register[7];
            // Shift through SPI_DATA
            shift_register <= {shift_register[6:0],1'b0};
            // Keep track of what bit it is on
            shift_counter <= shift_counter + 1'b1;
        end
        // on SCLK high reset the falling flag
        else if(clk_divided == 1'b1) begin
            falling <= 1'b0;
        end
    end
end

endmodule

```

这是一个很好用的 SPI 控制器，只要通过设置 SPI\_EN,SPI\_DATA,信号就能发送数据了，这个代码初学者可以当作一个 verilog 的例子学习下，仔细分析下 SPI 的工作时序。

Step4:添加一个毫秒延迟模块 Delay.v 文件

```

`timescale 1ns / 1ps
///////////////////////////////
//
//
//
//
// Create Date: 12:12:51 08/04/2014
// Module Name: Delay
// Project Name: ZedboardOLED

```

```
// Target Devices: Zynq
// Tool versions: Vivado 14.2 (64-bits)
// Description: Creates a delay of DELAY_MS ms
//
// Revision: 1.0
// Revision 0.01 - File Created
//
////////////////////////////////////////////////////////////////////////
module Delay(
    CLK,
    RST,
    DELAY_MS,
    DELAY_EN,
    DELAY_FIN
);

//
=====

//                                         Port Declarations
//
=====

input CLK;
input RST;
input [11:0] DELAY_MS;
input DELAY_EN;
output DELAY_FIN;

//
=====

//                                         Parameters, Registers, and Wires
//
=====

wire DELAY_FIN;

reg [31:0] current_state = "Idle";                      // Signal for state machine
reg [16:0] clk_counter = 17'b0000000000000000;           // Counts up on every rising edge of CLK
reg [11:0] ms_counter = 12'h000;                         // Counts up when clk_counter =
100,000

//
=====

//                                         Implementation
=====
```

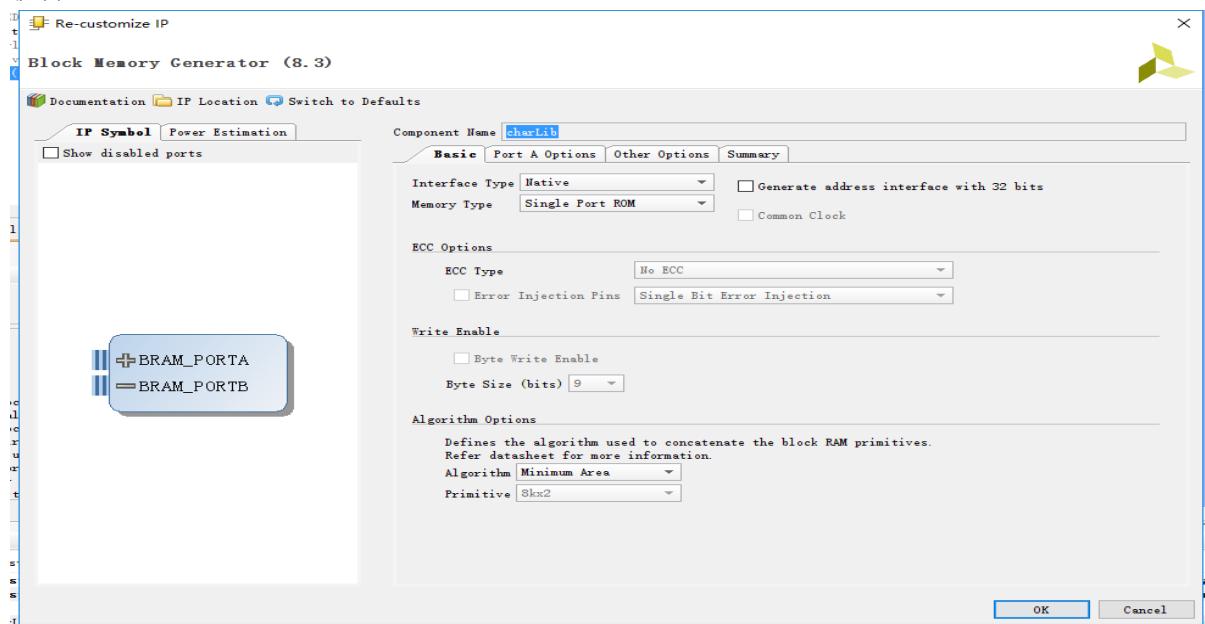
```
//  
=====  
assign DELAY_FIN = (current_state == "Done" && DELAY_EN == 1'b1) ? 1'b1 : 1'b0;  
  
// State Machine  
always @(posedge CLK) begin  
    // When RST is asserted switch to idle (synchronous)  
    if(RST == 1'b1) begin  
        current_state <= "Idle";  
    end  
    else begin  
        case(current_state)  
  
            "Idle" : begin  
                // Start delay on DELAY_EN  
                if(DELAY_EN == 1'b1) begin  
                    current_state <= "Hold";  
                end  
            end  
  
            "Hold" : begin  
                // Stay until DELAY_MS has occurred  
                if(ms_counter == DELAY_MS) begin  
                    current_state <= "Done";  
                end  
            end  
  
            "Done" : begin  
                // Wait until DELAY_EN is deasserted to go to IDLE  
                if(DELAY_EN == 1'b0) begin  
                    current_state <= "Idle";  
                end  
            end  
  
            default : current_state <= "Idle";  
  
        endcase  
    end  
end  
// End State Machine  
  
// Creates ms_counter that counts at 1KHz
```

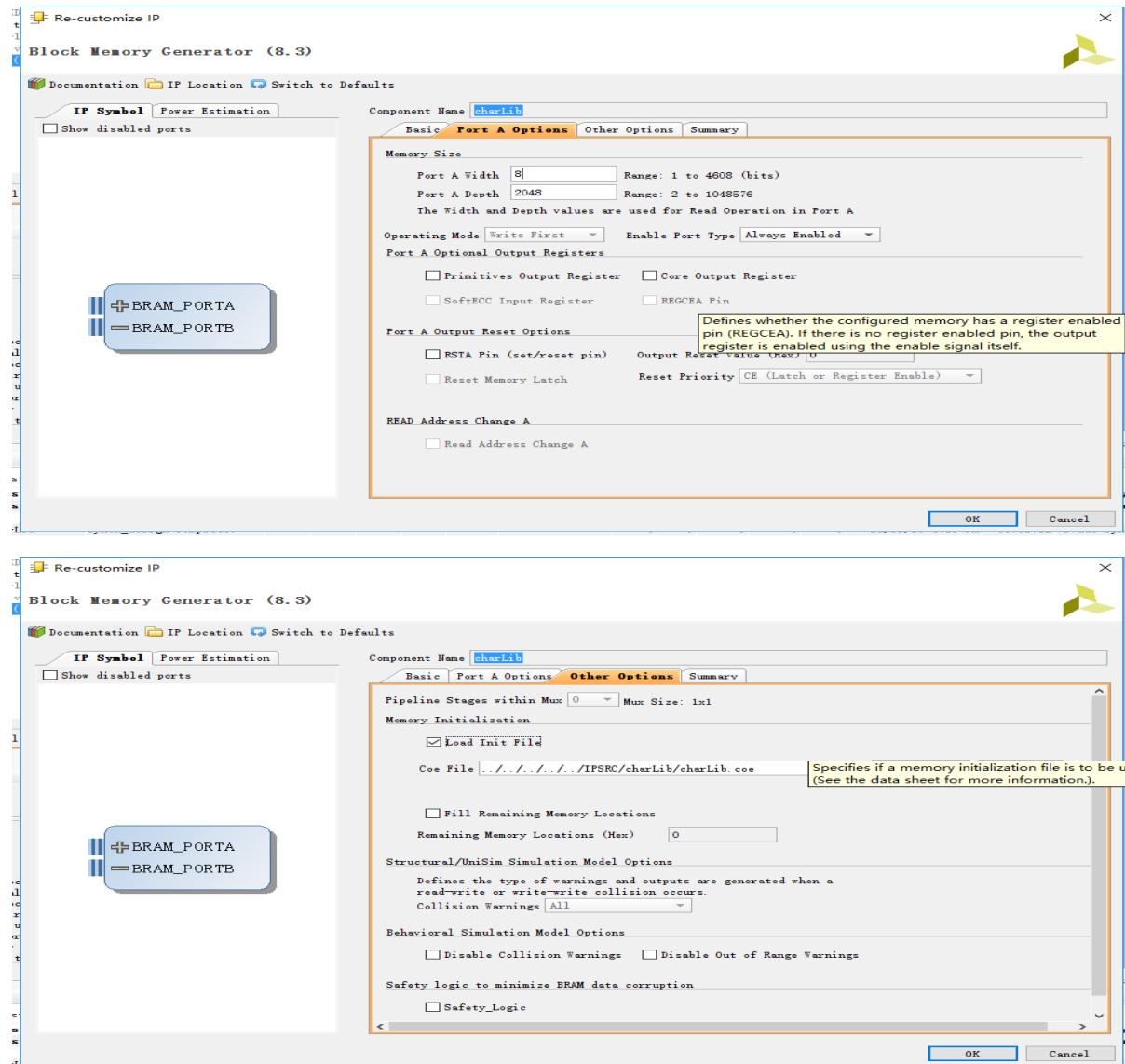
```

// CLK_DIV
always @(posedge CLK) begin
    if(current_state == "Hold") begin
        if(clk_counter == 17'b11000011010100000) begin // 100,000
            clk_counter <= 17'b00000000000000000000;
            ms_counter <= ms_counter + 1'b1; // increments at
1KHz
        end
    end begin
        clk_counter <= clk_counter + 1'b1;
    end
    else begin
        clk_counter <= clk_counter + 1'b1;
    end
end begin
// If not in the hold state reset counters
clk_counter <= 17'b00000000000000000000;
ms_counter <= 12'h000;
end
endmodule

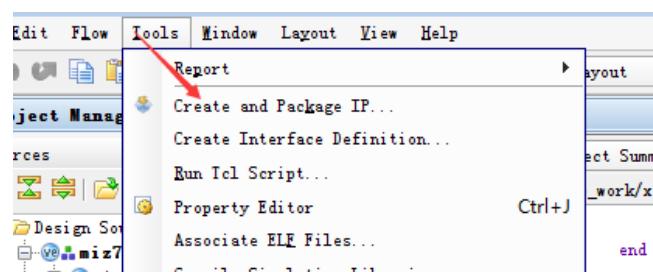
```

Step5:添加一个 Block ROM IP,按下图进行设置。ROM 的 coe 文件可在我们提供的源代码程序包中获得。





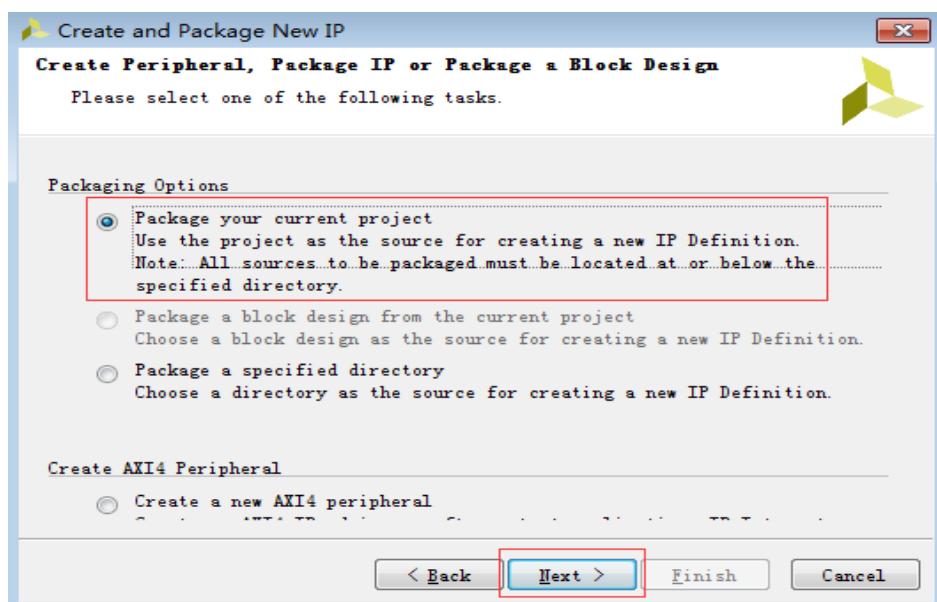
Step6:修改完成后重新封装一次自定义IP



Step7:单击NEXT

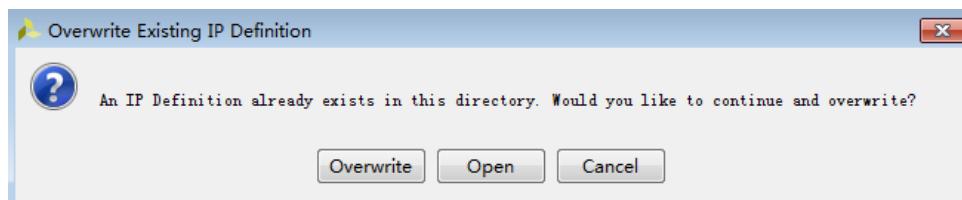


Step8:和第一次不同，这次选择第一个单选框然后单击 NEXT

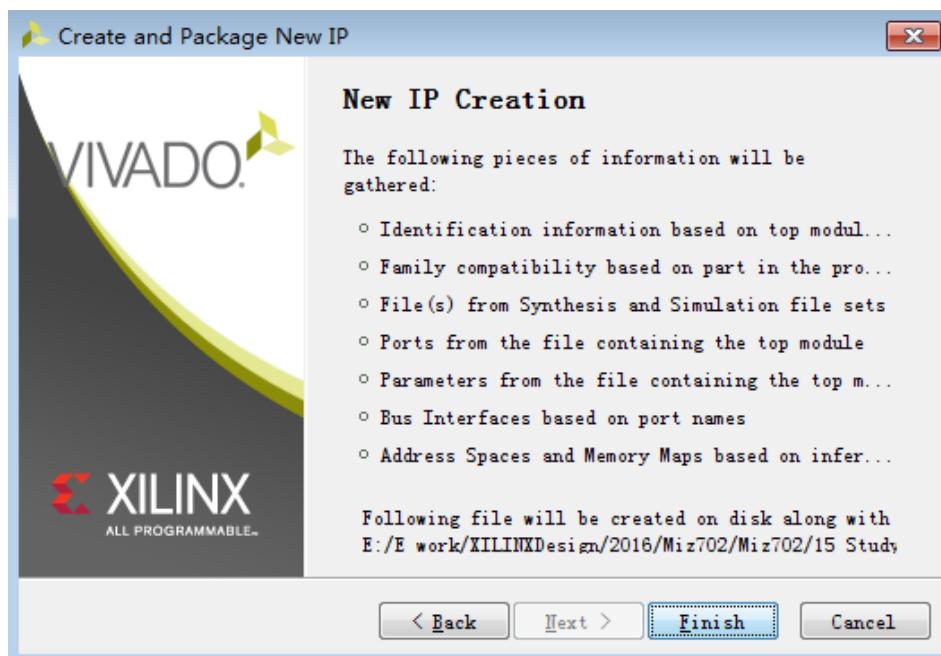


Step9:选择 Include.xci file，然后单击 NEXT

Step10:点击 Overwrite



Step11:点击 Finish 到此自定义 IP 结束



### 15.3 OLED 硬件控制器关键状态机

```
always @(posedge S_AXI_ACLK) begin
    if(RST_IN == 1'b1) begin
        current_state <= "Idle";
        temp_res <= 1'b0;
    end
    else begin
        temp_res <= 1'b1;

        case(current_state)
            // Idle State
            "Idle" : begin
                if(init_first_r == 1'b1) begin
                    temp_dc <= 1'b0; // DC=0 "Commands" , DC=1 "Data"
                    current_state <= "VddOn";
                    init_first_r <= 1'b0; // Don't go over the initialization
more than once
                end

                else begin
                    current_state <= "WaitRequest";
                end
            end
        endcase
    end
end
```

```
// Initialization Sequence
// This should be done only one time when Zedboard starts
"VddOn" : begin // turn the power on the logic of the display
    temp_vdd <= 1'b0; // remember the power FET transistor for VDD
is active low
    current_state <= "Wait1";
end

// 3
"Wait1" : begin
    after_state <= "DispOff";
    current_state <= "Transition3";
end

// 4
"DispOff" : begin
    temp_spi_data <= 8'hAE; // 0xAE= Set Display OFF
    after_state <= "SetClockDiv1";
    current_state <= "Transition1";
end

// 5
"SetClockDiv1" : begin
    temp_spi_data <= 8'hD5; //0xD5
    after_state <= "SetClockDiv2";
    current_state <= "Transition1";
end

// 6
"SetClockDiv2" : begin
    temp_spi_data <= 8'h80; // 0x80
    after_state <= "MultiPlex1";
    current_state <= "Transition1";
end

// 7
"MultiPlex1" : begin
    temp_spi_data <= 8'hA8; //0xA8
    after_state <= "MultiPlex2";
    current_state <= "Transition1";
end

// 8
```

```
"MultiPlex2" : begin
    temp_spi_data <= 8'h1F; // 0x1F
    after_state <= "ChargePump1";
    current_state <= "Transition1";
end

// 9
"ChargePump1" : begin // Access Charge Pump Setting
    temp_spi_data <= 8'h8D; //0x8D
    after_state <= "ChargePump2";
    current_state <= "Transition1";
end

// 10
"ChargePump2" : begin // Enable Charge Pump
    temp_spi_data <= 8'h14; // 0x14
    after_state <= "PreCharge1";
    current_state <= "Transition1";
end

// 11
"PreCharge1" : begin // Access Pre-charge Period Setting
    temp_spi_data <= 8'hD9; // 0xD9
    after_state <= "PreCharge2";
    current_state <= "Transition1";
end

// 12
"PreCharge2" : begin //Set the Pre-charge Period
    temp_spi_data <= 8'hFF; // 0xFF
    after_state <= "VCOMH1";
    current_state <= "Transition1";
end

// 13
"VCOMH1" : begin //Set the Pre-charge Period
    temp_spi_data <= 8'hDB; // 0xF1
    after_state <= "VCOMH2";
    current_state <= "Transition1";
end

// 14
```

```
"VCOMH2" : begin //Set the Pre-charge Period
    temp_spi_data <= 8'h40; // 0xF1
    after_state <= "DispContrast1";
    current_state <= "Transition1";
end

// 15
"DispContrast1" : begin //Set Contrast Control for BANK0
    temp_spi_data <= 8'h81; // 0x81
    after_state <= "DispContrast2";
    current_state <= "Transition1";
end

// 16
"DispContrast2" : begin
    temp_spi_data <= 8'hF1; // 0x0F
    after_state <= "InvertDisp1";
    current_state <= "Transition1";
end

// 17
"InvertDisp1" : begin
    temp_spi_data <= 8'hA0; // 0xA1
    after_state <= "InvertDisp2";
    current_state <= "Transition1";
end

// 18
"InvertDisp2" : begin
    temp_spi_data <= 8'hC0; // 0xC0
    after_state <= "ComConfig1";
    current_state <= "Transition1";
end

// 19
"ComConfig1" : begin
    temp_spi_data <= 8'hDA; // 0xDA
    after_state <= "ComConfig2";
    current_state <= "Transition1";
end
```

```
// 20
"ComConfig2" : begin
    temp_spi_data <= 8'h02; // 0x02
    after_state <= "VbatOn";
    current_state <= "Transition1";
end

// 21

"VbatOn" : begin
    temp_vbat <= 1'b0;
    current_state <= "Wait3";
end

// 22
"Wait3" : begin
    after_state <= "ResetOn";
    current_state <= "Transition3";
end

// 23
"ResetOn" : begin
    temp_res <= 1'b0;
    current_state <= "Wait2";
end

// 24
"Wait2" : begin
    after_state <= "ResetOff";
    current_state <= "Transition3";
end

// 25
"ResetOff" : begin
    temp_res <= 1'b1;
    current_state <= "WaitRequest";
end
// ***** END Initialization sequence but without turnning the
display on *****

// Main state
"WaitRequest" : begin
    if(Display_c == 1'b1) begin
```

```
        current_state <= "ClearDC";
        after_page_state <= "ReadRegisters";
        temp_page <= 2'b00;
    end
    else if ((Clear_c==1'b1) || (clear_screen_i == 1'b1)) begin

        current_state <= "ClearDC";
        after_page_state <= "ClearScreen";
        temp_page <= 2'b00;
    end
    else begin
        current_state<="WaitRequest"; // keep looping in the
WaitRequest state until you receive a command

        if ((clear_screen_i == 1'b0) && (ready ==1'b0)) begin // this part is only executed once, on start-up
            temp_spi_data <= 8'hAF; // 0xAF // Dispaly ON
            after_state <= "WaitRequest";
            current_state <= "Transition1";
            temp_dc<=1'b0;
            ready <= 1'b1;
        end
    end
end

//Update Page states
//1. Sets DC to command mode
//2. Sends the SetPage Command
//3. Sends the Page to be set to
//4. Sets the start pixel to the left column
//5. Sets DC to data mode
"ClearDC" : begin
    temp_dc <= 1'b0;
    current_state <= "SetPage";
end

"SetPage" : begin
    temp_spi_data <= 8'b00100010;
    after_state <= "PageNum";
    current_state <= "Transition1";
```

```
end

"PageNum" : begin
    temp_spi_data <= { 6'b000000,temp_page };
    after_state <= "LeftColumn1";
    current_state <= "Transition1";
end

"LeftColumn1" : begin
    temp_spi_data <= 8'b00000000;
    after_state <= "LeftColumn2";
    current_state <= "Transition1";
end

"LeftColumn2" : begin
    temp_spi_data <= 8'b00010000;
    after_state <= "SetDC";
    current_state <= "Transition1";
end

"SetDC" : begin
    temp_dc <= 1'b1;
    current_state <= after_page_state;
end

"ClearScreen" : begin
    for(i = 0; i <= 3 ; i=i+1) begin
        for(j = 0; j <= 15 ; j=j+1) begin
            current_screen[i][j] <= 8'h20;
        end
    end
    after_update_state <= "WaitRequest";
    current_state <= "UpdateScreen";
end

"ReadRegisters" : begin
    // Page0
    current_screen[0][0]<=slv_reg0[7:0];
    current_screen[0][1]<=slv_reg0[15:8];
    current_screen[0][2]<=slv_reg0[23:16];
    current_screen[0][3]<=slv_reg0[31:24];

```

```
current_screen[0][4]<=slv_reg1[7:0];
current_screen[0][5]<=slv_reg1[15:8];
current_screen[0][6]<=slv_reg1[23:16];
current_screen[0][7]<=slv_reg1[31:24];
current_screen[0][8]<=slv_reg2[7:0];
current_screen[0][9]<=slv_reg2[15:8];
current_screen[0][10]<=slv_reg2[23:16];
current_screen[0][11]<=slv_reg2[31:24];
current_screen[0][12]<=slv_reg3[7:0];
current_screen[0][13]<=slv_reg3[15:8];
current_screen[0][14]<=slv_reg3[23:16];
current_screen[0][15]<=slv_reg3[31:24];
//Page1
current_screen[1][0]<=slv_reg4[7:0];
current_screen[1][1]<=slv_reg4[15:8];
current_screen[1][2]<=slv_reg4[23:16];
current_screen[1][3]<=slv_reg4[31:24];
current_screen[1][4]<=slv_reg5[7:0];
current_screen[1][5]<=slv_reg5[15:8];
current_screen[1][6]<=slv_reg5[23:16];
current_screen[1][7]<=slv_reg5[31:24];
current_screen[1][8]<=slv_reg6[7:0];
current_screen[1][9]<=slv_reg6[15:8];
current_screen[1][10]<=slv_reg6[23:16];
current_screen[1][11]<=slv_reg6[31:24];
current_screen[1][12]<=slv_reg7[7:0];
current_screen[1][13]<=slv_reg7[15:8];
current_screen[1][14]<=slv_reg7[23:16];
current_screen[1][15]<=slv_reg7[31:24];
//Page2
current_screen[2][0]<=slv_reg8[7:0];
current_screen[2][1]<=slv_reg8[15:8];
current_screen[2][2]<=slv_reg8[23:16];
current_screen[2][3]<=slv_reg8[31:24];
current_screen[2][4]<=slv_reg9[7:0];
current_screen[2][5]<=slv_reg9[15:8];
current_screen[2][6]<=slv_reg9[23:16];
current_screen[2][7]<=slv_reg9[31:24];
current_screen[2][8]<=slv_reg10[7:0];
current_screen[2][9]<=slv_reg10[15:8];
current_screen[2][10]<=slv_reg10[23:16];
current_screen[2][11]<=slv_reg10[31:24];
current_screen[2][12]<=slv_reg11[7:0];
```

```
current_screen[2][13]<=slv_reg11[15:8];
current_screen[2][14]<=slv_reg11[23:16];
current_screen[2][15]<=slv_reg11[31:24];
//Page3
current_screen[3][0]<=slv_reg12[7:0];
current_screen[3][1]<=slv_reg12[15:8];
current_screen[3][2]<=slv_reg12[23:16];
current_screen[3][3]<=slv_reg12[31:24];
current_screen[3][4]<=slv_reg13[7:0];
current_screen[3][5]<=slv_reg13[15:8];
current_screen[3][6]<=slv_reg13[23:16];
current_screen[3][7]<=slv_reg13[31:24];
current_screen[3][8]<=slv_reg14[7:0];
current_screen[3][9]<=slv_reg14[15:8];
current_screen[3][10]<=slv_reg14[23:16];
current_screen[3][11]<=slv_reg14[31:24];
current_screen[3][12]<=slv_reg15[7:0];
current_screen[3][13]<=slv_reg15[15:8];
current_screen[3][14]<=slv_reg15[23:16];
current_screen[3][15]<=slv_reg15[31:24];

after_update_state <= "WaitRequest";
current_state <= "UpdateScreen";
end

//UpdateScreen State
//1. Gets ASCII value from current_screen at the current page and the
current spot of the page
//2. If on the last character of the page transition update the page
number, if on the last page(3)
//           then the updateScreen go to "after_update_state" after
"UpdateScreen" : begin

    temp_char <= current_screen[temp_page][temp_index];

    if(temp_index == 'd15) begin

        temp_index <= 'd0;
        temp_page <= temp_page + 1'b1;
        after_char_state <= "ClearDC";

        if(temp_page == 2'b11) begin
            after_page_state <= after_update_state;
```

```
        clear_screen_i<=1'b0;
    end
    else begin
        after_page_state <= "UpdateScreen";
    end
end
else begin

    temp_index <= temp_index + 1'b1;
    after_char_state <= "UpdateScreen";

end

current_state <= "SendChar1";

end

//Send Character States
//1. Sets the Address to ASCII value of char with the counter appended
to the end
//2. Waits a clock for the data to get ready by going to ReadMem and
ReadMem2 states
//3. Send the byte of data given by the block Ram
//4. Repeat 7 more times for the rest of the character bytes
"SendChar1" : begin
    temp_addr <= {temp_char, 3'b000};
    after_state <= "SendChar2";
    current_state <= "ReadMem";
end

"SendChar2" : begin
    temp_addr <= {temp_char, 3'b001};
    after_state <= "SendChar3";
    current_state <= "ReadMem";
end

"SendChar3" : begin
    temp_addr <= {temp_char, 3'b010};
    after_state <= "SendChar4";
    current_state <= "ReadMem";
end

"SendChar4" : begin
```

```
temp_addr <= {temp_char, 3'b011};  
after_state <= "SendChar5";  
current_state <= "ReadMem";  
end  
  
"SendChar5" : begin  
    temp_addr <= {temp_char, 3'b100};  
    after_state <= "SendChar6";  
    current_state <= "ReadMem";  
end  
  
"SendChar6" : begin  
    temp_addr <= {temp_char, 3'b101};  
    after_state <= "SendChar7";  
    current_state <= "ReadMem";  
end  
  
"SendChar7" : begin  
    temp_addr <= {temp_char, 3'b110};  
    after_state <= "SendChar8";  
    current_state <= "ReadMem";  
end  
  
"SendChar8" : begin  
    temp_addr <= {temp_char, 3'b111};  
    after_state <= after_char_state;  
    current_state <= "ReadMem";  
end  
  
"ReadMem" : begin  
    current_state <= "ReadMem2";  
end  
  
"ReadMem2" : begin  
    temp_spi_data <= temp_dout;  
    current_state <= "Transition1";  
end  
  
// SPI transitions  
// 1. Set SPI_EN to 1  
// 2. Waits for SpiCtrl to finish  
// 3. Goes to clear state (Transition5)
```

```
"Transition1" : begin
    temp_spi_en <= 1'b1;
    current_state <= "Transition2";
end

"Transition2" : begin
    if(temp_spi_fin == 1'b1) begin
        current_state <= "Transition5";
    end
end

// Delay Transitions
// 1. Set DELAY_EN to 1
// 2. Waits for Delay to finish
// 3. Goes to Clear state (Transition5)
"Transition3" : begin
    temp_delay_en <= 1'b1;
    current_state <= "Transition4";
end

"Transition4" : begin
    if(temp_delay_fin == 1'b1) begin
        current_state <= "Transition5";
    end
end

// Clear transition
// 1. Sets both DELAY_EN and SPI_EN to 0
// 2. Go to after state
"Transition5" : begin
    temp_spi_en <= 1'b0;
    temp_delay_en <= 1'b0;
    current_state <= after_state;
end

default : current_state <= "Idle";

endcase
end
end
```

```
// Internal reset generator
always @(posedge S_AXI_ACLK) begin
if (RST_IN == 1'b1)
    count<=count+1'b1;
if (count == 12'hFFF) begin
    RST_internal <=1'b0;
end
end
```

这个状态机实现了 OLED 的通电控制、初始化、以及字符的显示。

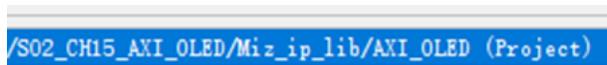
## 15.4 硬件工程搭建

Step1：另外新建一个 VIVADO 工程，根据自己的开发板正确配置芯片型号。

Step2：在 Project manager 区中单击 Project settings。

Step3：选择 IP 设置区中的 repository manager, 将上一节我们封装好的 IP 的路劲添加进去。

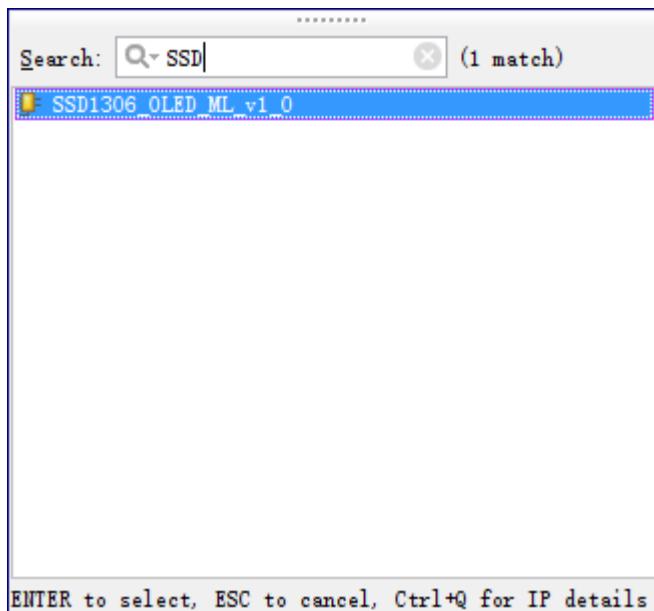
Step4：单击+号图标，将上一节封装的 IP 的路劲存放进去，单击 OK。



Step5：新建一个 BD 文件，输入文件名，完成创建。

Step6：向 BD 文件中添加一个 ZYNQ Processing system, 根据自身硬件完成 IP 的配置。

Step7：单击添加 IP 图标，输入上一节我们自定义 IP 的模块名，将其添加入 BD 文件中。



Step8：直接点击 Run connection automation，然后单击 OK。

Step9：选中 SSD1306 控制 IP 的输出端口，按 Ctrl+T 组合键引出端口。

Step10：右键单击 Block 文件，文件选择 Generate the Output Products。

Step11：右键单击 Block 文件，选择 Create a HDL wrapper，根据 Block 文件内容产生一个 HDL 的顶层文件，并选择让 vivado 自动完成。

Step12：将我们提供的约束文件添加到工程当中来。

Step13：生成 bit 文件。

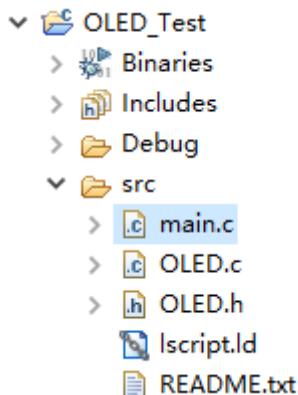
## 15.5 导入到 SDK

Step1:导出硬件。

Step2:新建一个名为 OLED\_Test 的空白工程。

Step3:打开我们提供的源程序包，在第二季，第 15 章的文件夹中，将 SDK 所有的文件复制过来。

Step4:展开 OLED\_Test，在 Src 下按 Ctrl+V 将所有文件粘贴过来。



Step5：右击工程，选择 Debug as ->Debug configuration。

Step6：选中 system Debugger,双击创建一个系统调试。

Step7：设置系统调试。

Step8:单击运行程序按钮 运行程序，此时可在 OLED 上观察到滚动显示我们定义的字符。

## 15.6 本章小结

本章的方案虽然不及 14 章的功能强大，但是可以提高 CPU 的工作效率，充分发挥 PL 的硬件资源的能力，减轻 CPU 的负担。

两种方案各有优缺点，前者很好地平衡了 PS 和 PL 部分的工作，但是功能单一，只能显示字符；后者未能合理使用 PL 资源，但是灵活度高、功能强大。读者可以尝试将两种方案进行融合，取长补短，设计出更优秀的方案。

## CH16 等精度频率计实验

在了解了 AXI 总线之后，今天我自己动手设计一个带 AXI4-Lite 总线的 IP，来完成频率计的实验。

频率计虽然小，但是也算五脏俱全，涉及到 zynq 的方方面面，比如：

- A) PL 部分逻辑设计
- B) 自定义 AXI4-Lite 的 IP 的建立
- C) 通过 AXI4-Lite 总线实现 PS 与 PL 间的数据传递
- D) PS 控制输入输出外设

### 16.1 等精度频率计原理

#### 16.1.1 引言

传统的数字频率测量方法有脉冲计数法和周期测频法，但这两种方法分别适合测量高频和低频信号，具有较大的局限性。多周期同步测频法以脉冲计数法为基础，并对之进行改进，实现了全频段的等精度测量，且测量精度大大提高，因此多周期同步测频法在目前测频系统中得到越来越广泛的应用。很多文献对多周期同步测频法的等精度测量原理有所介绍，但多数文献都是从测频控制模块的结构和测频波形出发，对测频原理进行论述。就我的亲身感触而言，这种阐述方式并不能帮助读者很快很好地理解频率计的原理（也有可能是本人比较笨>\_<），因此，本文以脉冲计数法为基础，对之进行逐步改进得到多周期同步测频法，即等精度测频法，个人觉得这种逐步深入的方法可以更好地理解决精度频率计的原理。

#### 16.1.2 频率测量原理

所谓频率，就是周期性信号在单位时间内变化的次数。频率测量的方法有很多种，在模拟电路中有比较测频法，响应测频法，游标法等；在数字电路中，有基于脉冲计数测频原理的直接测频法、周期测频法、在直接测频法的基础上发展起来的多周期同步测频法和全同步数字测频法。本小节简单介绍计数测频法和周期测频法，重点分析多周期同步测频法的工作原理。

#### 16.1.3 脉冲计数法

脉冲计数法原理：在预置的闸门时间  $T_{pr}$  内对被测脉冲信号进行计数，得到脉冲数  $N_x$ ，通过公式  $F_x=N_x/T_{pr}$  可计算出单位时间内脉冲个数，即被测信号的频率。

该方法测量误差来源于闸门时间  $T_{pr}$  和计数值  $N_x$ ，且被测信号频率  $F_x$  与闸门开启时间  $T_{pr}$  越大，测频精度越高。因此，该方法适合于高频率信号的测量。

### 16.1.4 周期测频法

预置测频闸门开启时间  $T_{pr}$  等于被测信号的周期  $T_x$ ，通过计数器在闸门时间  $T_{pr}$  内基准时钟信号进行计数，若得到的基准时钟信号脉冲个数为  $N_x$ ，且基准时钟周期为  $T$ ，则可按公式  $T_x = T * N_x$  计算出待测信号的周期  $T_x$ ，然后换算得到被测信号频率。

该方法的测量误差来源于基准时钟信号和计数误差，且测量相对误差与被测频率  $F_x$  成正比，与基准时钟频率  $F$  成反比。所以，当被测信号频率越低，基准时钟频率越高时，周期测频法的测量精度越高。

### 16.1.5 多周期同步测频原理及误差分析

用范围，但不能兼顾高低频等精度的测量要求。多周期同步频率测量法以脉冲计数测频法为基础，实现了闸门信号与被测信号的同步，从而解决了上述问题，实现了测量全频段的等精度测量。

从脉冲计数测频法原理可以看出，该方法闸门信号与被测信号不同步，也就是说在时间轴上两路信号随机出现，相对位置具有随机性。因此即使在相同的闸门时间内，被测脉冲计数结果也不一定相同，闸门时间大于  $N * T_{testclk}$  时，越接近  $(N+1) * T_{testclk}$ ，误差越大。为了解决这个问题，利用 D 触发器使闸门信号在被测信号的上升沿产生动作，这样以来测量的实际门控时间刚好是被测信号周期的整数倍，这样就消除了被测信号引起的 1 个周期的误差。

这里还是给个时序图，解释一下引入 D 触发器为何能消除被测信号引起的 1 个周期的误差。

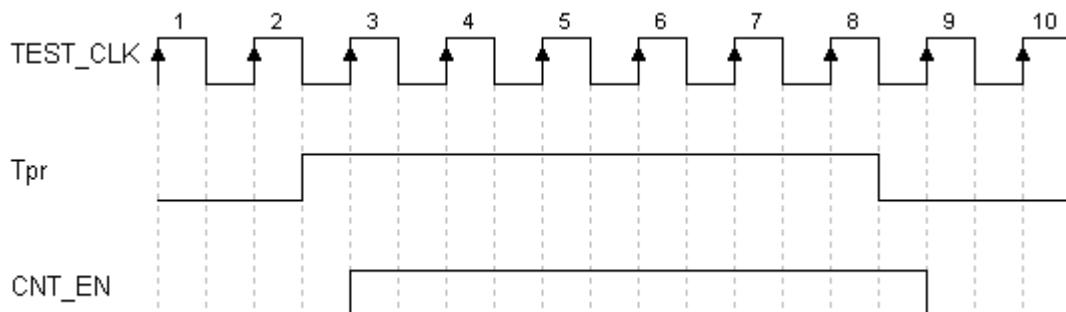


图 1 Tpr 处理后成为 CNT\_EN

由于引入了 D 触发器，CNT\_EN 不会在  $T_{pr}$  发生变化时立即变化，而是在 TestClk 上升沿到来时才发生变化，从而保证 CNT\_EN 刚好是 TEST\_Clk 的整数倍。测频法和测周法的原理和误差分析如果不明白，自己画个图试试，可以很好地帮助理解。

解决问题的同时，产生了新的问题：实际闸门时间与预置闸门时间不相等，因此需要获取实际闸门时间。为解决这一问题，引入另一计数器和标准时钟信号。在测量被测信号频率的同时，对标准时钟脉冲进行计数，通过计算即可得到实际闸门时间。这样就得到多周期同步频率计的主要结构，如图 2 所示。

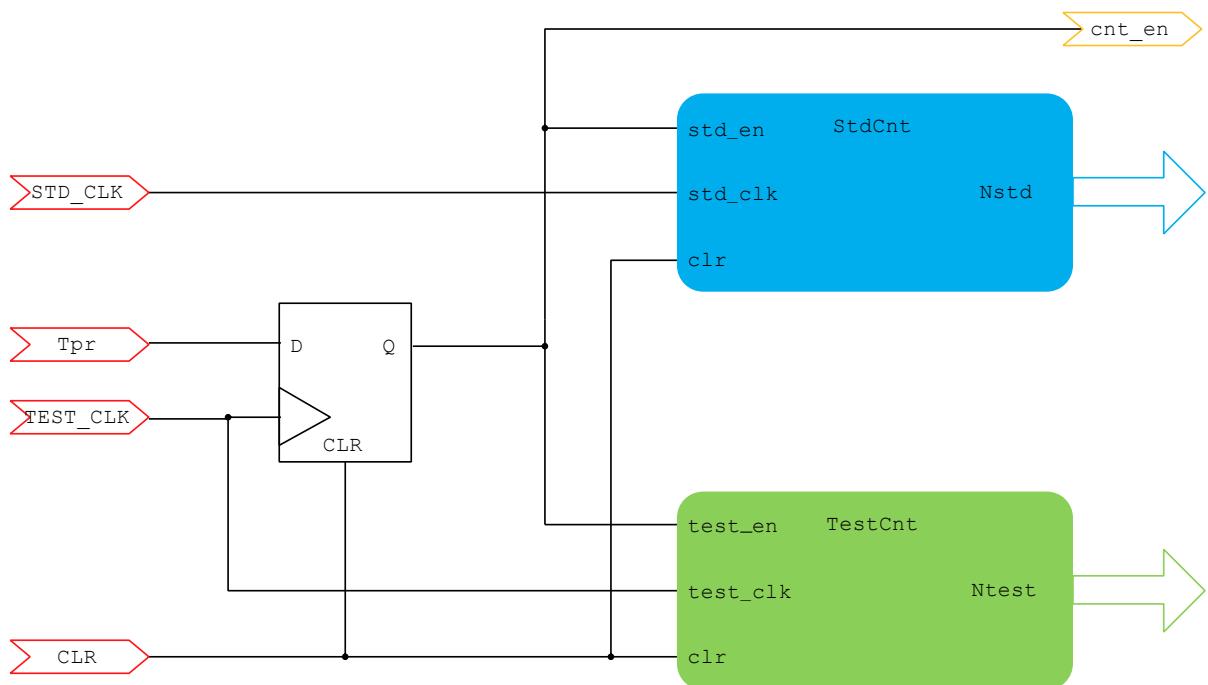


图 2 测频主控模块结构图

其中，STD\_CLK 为标准时钟；Tpr 为预置门控信号；TEST\_CLK 为待测信号；CLR 为计数清零信号。

在计数允许时间内，同时对标准信号和被测信号进行计数，由于两个计数器计数时间相等，从而得到公式（1）。

$$N_{std}/F_{std} = N_{test}/F_{test} \quad \text{公式(1)}$$

其中  $N_{std}$  为标准时钟计数值； $F_{std}$  为标准时钟频率； $N_{test}$  为待测信号计数值； $F_{test}$  为待测信号频率，由公式（1）可知待测频率为  $F_{test}=F_{std}*N_{test}/N_{std}$ 。

由于未对标准时钟进行同步计数，所以测量结果会产生  $\pm 1$  个标准信号脉冲的误差。

从以上论述可以得出如下结论：

待测信号频率  $F_{test}$  的相对测量误差与待测信号频率无关。

增大 Tpr 或提高  $F_{std}$ ，可以增大  $N_{std}$ ，减少测量误差，提高测量精度。

标准频率误差为  $\Delta F_{std}/F_{std}$ 。测试电路可采用高频率稳定度和高精度的恒温可微调的晶体振荡器作标准频率发生电路从而进一步降低测频误差。

## 16.2 等精度频率计设计

### 16.2.1 PS 寄存器功能划分

reg0：控制寄存器 0 (offset: 0x00)

Bit	功能
Bit31~bit2	保留

Bit1	闸门信号 Tpr (高时 打开闸门)
Bit0	复位/清零信号 clr (低有效)

reg1: 数据寄存器 Nstd (offset: 0x04)

Bit	功能
Bit31~bit0	标准时钟计数值

reg2: 数据寄存器 Ntest (offset: 0x08)

Bit	功能
Bit31~bit0	待测信号计数值

## 16.2.2 具体实现

本文方案实现亦分为两部分，一是计数值的获取，该部分由测频控制模块（PL 实现）完成；二是结果的计算及显示，该部分工作由 PS 完成。采用 M1z 系列开发板板载的 100MHz 时钟信号作为标准信号，可使测量的最大相对误差小于或等于 10<sup>-8</sup>。

## 16.2.3 频率计 PL 部分代码设计

测频主要控制部分结构图在原理篇已经给出，该结构并不复杂，且所用元件较为常见。因此可以自行编码实现，也可以调用元件库实现。

这部分涉及到创建基于 AXI4-Lite 总线的 IP 核，方法参见前面章节内容

根据之前的分析，PL 部分我们需要在闸门型号打开时，我们需要对标准时钟 StdClock 以及待测时钟 TestClock 分别进行计数。闸门信号关闭时停在计算，并把计数值存放到寄存器中等待 PS 通过 AXI4-Lite 总线读取数据。

在自定义 AXI4-Lite IP 内部添加用户逻辑如下：

```
reg clr;
reg Tpr;
reg[31:0] Nstd;
reg[31:0] Ntest;
```

```
reg [11:0]rlcd_rgb;

always @(posedge S_AXI_ACLK)
begin
    if ( S_AXI_ARESETN == 1'b0 )
        begin
            clr <= 1'd0;
            Tpr <= 1'd0;
        end
    else
        begin
            clr <= slv_reg0[0];
            Tpr <= slv_reg0[1];
        end
end

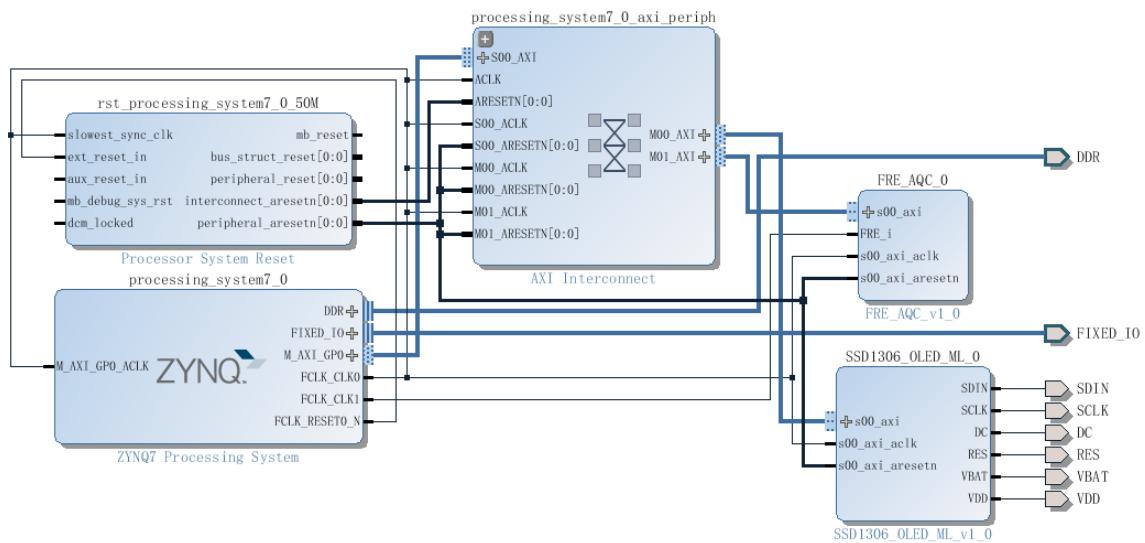
always @(posedge S_AXI_ACLK)
if(clr == 1'b0)
begin
    Nstd <= 32'd0;
end
else if(Tpr == 1'b1)
begin
    Nstd <= Nstd + 1'b1;
end
else
begin
    Nstd <= Nstd;
end
```

```
//-----  
  
always @(posedge FRE_i)  
  if(clr == 1'b0)  
    begin  
      Ntest <= 32'd0;  
    end  
  else if(Tpr == 1'b1)  
    begin  
      Ntest <= Ntest + 1'b1;  
    end  
  else  
    begin  
      Ntest <= Ntest;  
    end  
  
// User logic ends
```

这里的测试时钟是 FRE\_i，后续我们可以观察 PS 那边计算的结果。

### 16.3 硬件工程搭建

本章工程比较简单，在上一章 AXI\_OLED 的工程的基础上添加一个上一节封装的 IP 和用 PS 端输出一个测试时钟即可，完成的硬件工程如下图所示：



完善工程后，生成 Bit 文件即可。

## 16.4 导入到 SDK

Step1:导出硬件。

Step2:用以下程序替换之前 main.c 中的内容。

```
/*
 * main.c
 *
 * Created on: 2016 年 7 月 1 日
 * Author: Administrator
 */

#include <stdio.h>
#include "xbasic_types.h"
#include "OLED.h"
#include "sleep.h"
#include "xparameters.h"
void print(char *str);
#define FRE_AQC_BASE XPAR_FRE_AQC_0_BASEADDR

int main()
{
    char str[16]="";
    u32 fre_std,fre_test;
    double fre_val;
    oled_fresh_en();//enable oled print
    print_message("frequency test",0);
    while(1)

```

```
{  
    Xil_Out32(FRE_AQC_BASE,0);  
    usleep(10);  
    Xil_Out32(FRE_AQC_BASE,3);  
    usleep(100000);  
    fre_std =Xil_In32(FRE_AQC_BASE+4);  
    fre_test =Xil_In32(FRE_AQC_BASE+8);  
    fre_val =(double)fre_test/fre_std*100;  
    sprintf(str,"f=% .4lfMHZ",fre_val);  
    print_message(str,1);  
  
    Xil_Out32(FRE_AQC_BASE,0);  
    usleep(10);  
    Xil_Out32(FRE_AQC_BASE,3);  
    usleep(1000);  
    fre_std =Xil_In32(FRE_AQC_BASE+4);  
    fre_test =Xil_In32(FRE_AQC_BASE+8);  
    fre_val =(double)fre_test/fre_std*100;  
    sprintf(str,"f=% .4lfMHZ",fre_val);  
    print_message(str,2);  
    sleep(1);  
}  
  
return 0;  
}
```

Step3：右击工程，选择 Debug as ->Debug configuration。

Step4：选中 system Debugger,双击创建一个系统调试。

Step5：设置系统调试。

Step6:单击运行程序按钮  运行程序，此时可在 OLED 上观察到测量的频率值。

## 16.5 误差分析

单击运行程序后，在 OLED 上我们看到测得的频率为 23.6M，此时查看 ZYNQ Processing System 的输出频率为 23MHZ，实际为 23.2558，基本满足功能要求。

## 16.6 本章小结

计算在 PS 部分进行很简单，就是一个除法，成功的关键在于 AXI 总线通信无误。

通过本章的学习主要是培养读者设计 IP 的思路，如何划分寄存器功能。以及如何将任务合理的分配给 PL 以及 PS，让其发挥各自的优势。

一个完美的结束  
意味着一个新的开始！

[www.osrc.cn](http://www.osrc.cn)

米联客  
技术论坛  
秀出你的风采！