

ZYNQ SOC 修炼秘籍

(2017 网手版)

本次更新日期 2017 年 5 月 10 日

本书已经更新 1183 页 还未完结

网络连载更新。 . .

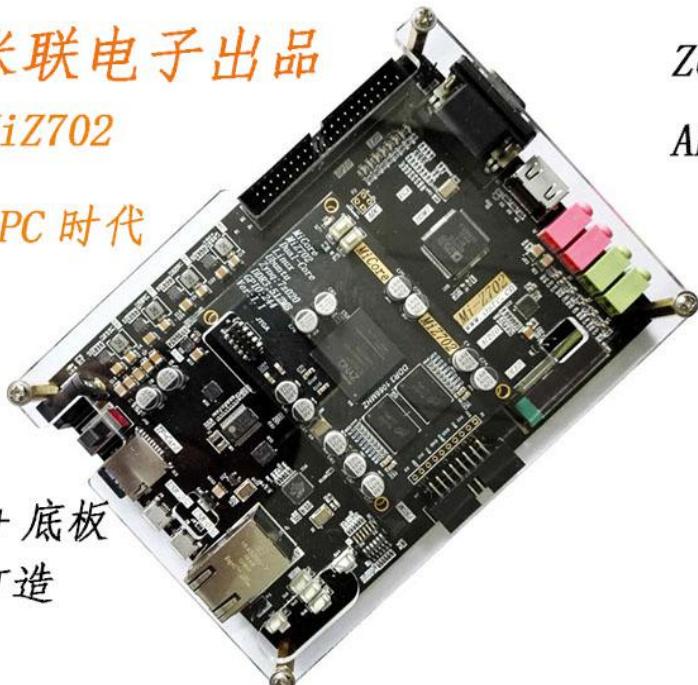
常州一二三电子 / 泰州米联电子 / 南京米联电子 /
互联网教育事业部
ZYNQ 资料开发团队
排版： 汤全元

MiZ702

南京米联电子出品
型号 : MiZ702

开启 SOPC 时代
新模式

核心板 + 底板
倾情打造



Zedboard 兼容
ARM A9 双核

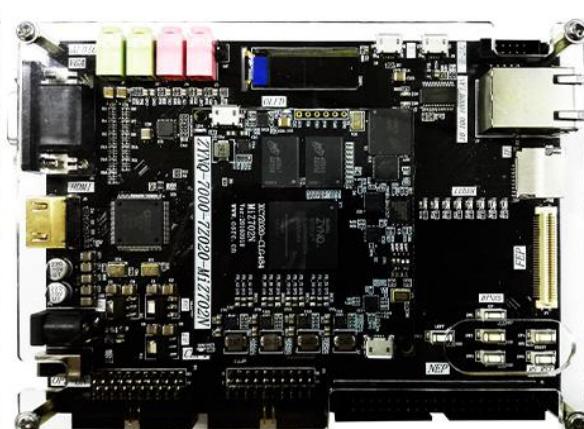
- ARM A9双核667M
- 512MB 内存
- HDMI1.4输出
- RGMII千兆网口
- USB2.0 高速
- USB转串口
- VGA输出
- TF接口
- OLED 显示器
- 音频接口
- 支持子卡
- 专业电源管理

MiZ702N

南京米联电子出品

1GB 内存 XC7Z020-CLG484-1I

型号 : MiZ702N 【MIZ702 升级】8GB EMMC ARM A9 双核



核心板 + 底板 高速接插件

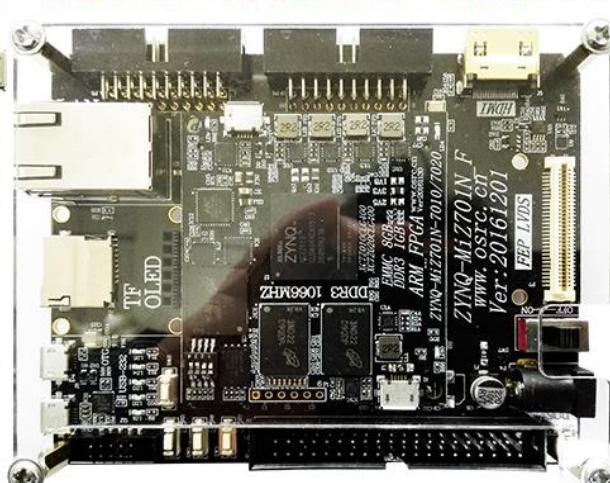
- ARM A9双核667M
- 1024MB 内存
- 8GB EMMC(板载)
- HDMI (ADV7511)
- RGMII千兆网口
- USB2.0 高速
- USB转串口
- VGA (565) 输出
- TF接口
- 音频接口
- 支持子卡
- 专业电源管理

MiZ701N

南京米联电子出品

1GB 内存 XC7Z010/020-CLG400

型号 : MiZ701N 【MIZ701 升级】8GB EMMC ARM A9 双核



核心板 + 底板 高速接插件

- ARM A9双核667M
- 1024MB 内存
- 8GB EMMC(板载)
- HDMI (IO 模拟)
- RGMII千兆网口
- USB2.0 高速
- USB转串口
- TF接口
- VGA (子卡) 输出
- 音频接口(子卡)
- 支持其他子卡
- 专业电源管理

版本	时间	描述
Rev1.0	2015-07-25	第一版初稿
Rev1.1	2016-03-31	更新 26 章节
Rev1.2	2016-04-10	更新 linux 系统定制相关教程
Rev1.3	2016-04-31	更新裸机部分 23 章及操作系统部分章节
Rev1.4	2016-05-08	更新裸机部分 24 章及操作系统部分章节
Rev1.5	2016-06-05	修复第 15 章 自定义 IP 生成的 bug
Rev1.6	2016-06-21	修复了第三章状态机的错误代码，提供了多个仿真例子
Rev1.7	2016-06-27	重新调整了文档结构内容排布更加合理 ,增加了里利理论部分的代码分析。
Rev1.8	2016-07-12	第三章中关于阻塞和非阻塞视频讲解概念混淆的纠正。
REV1.9	2016-08-10	增加 OV7725 IP VTG IP Video Out IP 使用 讲解 AXI-Stream 协议 和 VDMA IP 使用
REV2.0	2016-08-14	修改目录顺序把 VGA 接口部分的讲解放到 HDMI 之前 讲解 XILINX 自带的标准视频类 IP 的使用包括 Video in IP /VDMA IP /VTC IP /Video Out IP 给出了彩条测试，内存显示 图片的测试。

		封装了 OV7725 自定义 IP 实现图片显示 封装了 OV5640 自定义 IP 实现图片显示
REV2.1	2017-02-28	重大更新,对之前的例子进行了完善，并且增加了很多新例子， 删除一些不必要，不常用的例子。
REV2.2	2017-05-10	重大更新，第三季 SOC 裸机更新到 17 课时；第四季 LINUX 更 新到第七课时；第五季节更新到 11 课时；大量实战例子

感谢您使用南京米联团队开发的 MiZ7(MIZ701N/MIZ702/MIZ702N)开发板，在使用开发板前请认真阅读本手册，并且掌握如何正确使用开发板，不合理的操作会导致开发板损坏。

此手册不断更新中，请下载最新版本。

软件版本：VIVADO2015.4

使用本手册提供的 VIVADO 版本或者到赛灵思官网下载 2015.4 版本

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

版权声明：

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

技术支持：

版主大神们都等着大家去提问--电子资源论坛 www.osrc.cn

微信公众平台：电子资源论坛



目录

目录.....	7
【第一季】ZYNQ SOC 开机及 FPGA 基础 共 12 课时.....	25
S01_CH01_开机程序测试.....	26
1.1 MIZ701N 开机测试连线图.....	26
1.2 MIZ702N 开机测试连线图.....	27
1.3 MIZ702 开机测试连线图.....	28
1.5 UBUNTU 系统界面.....	29
1.7 网口测试.....	31
1.8 美图欣赏.....	32
S01_CH02_ZYNQ_VIVADO 软件安装.....	38
2.1 VIVADO 软件介绍.....	38
2.2 VIVADO 软件安装(适合所有 vivado 安装).....	38
2.3 VIVADO 软件注册.....	42
2.3 本章小结.....	44
S01_CH03_USB 下载器驱动安装及下载程序.....	45
3.1 下载器驱动的安装.....	45
3.2 下载 runled 工程的 bit 文件验证板子和下载器工作正常.....	46
3.3 下载器使用需要注意的问题.....	47
4.1 Verilog HDL 代码规范.....	48
◆ 项目构架设计.....	48
◆ 接口时序设计规范.....	48
4.2 技术背景.....	51
4.3 Verilog 最基础语法.....	54
4.4 关键字.....	55
4.5 Verilog 中数值表示的方式.....	61
4.6 阻塞赋值和非阻塞赋值详解.....	61
S01_CH05_FPGA 设计 Verilog 基础 (二)	66
5.1 状态机设计.....	66
5.2 一段式状态机.....	67
5.3 两段式状态机.....	68
5.4 三段式状态机.....	70
S01_CH06_FPGA 设计 Verilog 基础 (三)	73
6.1 完成的 Test bench 文件结构.....	73
6.2 时钟激励设计.....	73
6.3 复位信号设计.....	75
6.4 特殊信号设计.....	76
6.5 仿真控制语句及系统任务描述.....	79
6.6 加法器的仿真测试文件编写.....	82
S01_CH07_FPGA_RunLED 创建 VIVADO 工程实验.....	85

7.1 硬件图片.....	85
7.2 硬件原理图.....	85
7.3 新建 VIVADO 工程.....	86
7.4 创建工程文件.....	89
7.5 Verilog FPGA 流水灯实验.....	93
7.6 添加管脚约束文件.....	95
7.7 编译并且产生 bit 文件.....	98
7.8 下载程序.....	98
7.9 实验结果.....	100
7.10 本章小结.....	101
S01_CH08_FPGA_Button 按钮去抖动实验.....	102
8.1 硬件介绍.....	102
8.2 时序设计.....	103
8.3 程序源码.....	103
8.4 程序分析.....	108
8.5 综合布线前仿真时序.....	108
8.6 Chipscope 在线逻辑分析仪仿真.....	108
8.7 输出结果.....	108
8.8 小结.....	108
S01_CH09_FPGA 多路分配器设计.....	110
9.1 硬件图片.....	110
9.2 硬件原理图.....	110
9.3 介于 VIVADO 的 FPGA 设计流程.....	111
9.4 多路分配器设计思想.....	111
9.5 时序设计.....	112
9.6 程序源码.....	112
9.7 行为仿真.....	117
9.7.1 创建多路分频器工程.....	117
9.7.2 添加仿真文件.....	121
9.7.3 行为级仿真.....	125
9.8 综合 Synthesis.....	129
9.8.1 添加文件.....	129
9.8.2 综合并查看报告.....	131
9.8.3 综合时序仿真.....	131
9.9 执行 Implementation.....	132
9.9.1 执行并查看报告.....	132
9.9.2 布局布线后时序仿真.....	133
9.10 VIVADO 在线逻辑分析仪使用.....	134
9.10.1 IP Catalog 添加 IAP ip core.....	134
9.10.2 逻辑分析仪抓取的信号.....	138
9.10.3 逻辑分析仪使用.....	139
9.11 小结.....	140
S01_CH10_VGA 接口测试.....	141

10.1 硬件介绍.....	141
10.2 时序分析.....	143
10.3 新建 VIVADO 工程.....	144
10.4 创建工程文件.....	149
10.5 添加管脚约束文件.....	163
10.6 编译并且产生 bit 文件.....	169
10.7 下载程序.....	169
10.8 实验结果.....	171
10.9 本章小结.....	172
S01_CH11_ADV7511 HDMI 接口测试.....	173
11.1 ADV7511 概述.....	173
11.1.1 硬件特性.....	173
11.1.2 视频输入.....	173
11.1.3 支持的输出格式.....	174
11.1.4 视频接口信号采样.....	174
11.1.5 功能框图.....	176
11.1.6 寄存器空间.....	176
11.2 硬件电路分析.....	177
11.3 创建工程文件.....	178
11.4 添加管脚约束文件.....	183
11.5 编译并且产生 bit 文件.....	187
11.6 下载程序.....	187
11.7 实验结果.....	189
S01_CH12_PL IO 口模拟 HDMI 接口测试.....	190
12.1 创建工程文件.....	190
12.4 添加管脚约束文件.....	200
12.5 编译并且产生 bit 文件.....	201
12.6 下载程序.....	201
12.7 实验结果.....	203
【第二季】ZYNQ SOC 入门基础 共 16 课时.....	205
S02_CH01_Hello World 实验.....	206
1.1 最小系统分析.....	206
1.6 MemTest 内存测试程序.....	229
1.7 DRAMTest 内存测试程序.....	231
1.8 LWIP 协议对千兆网口测试.....	232
1.9 使用快捷按钮调试.....	235
1.10 本章小结.....	235
S02_CH02_MIO 实验.....	236
2.1 GPIO 简介.....	236
2.1.1 GPIO 的控制寄存器地址空间.....	237
2.1.2 MIO 内部构造分析.....	240
2.1.3 EMIO 的特性.....	241
2.2 电路分析及实验预期.....	241

2.3 ZYNQ 核的添加及配置.....	241
2.4 新建 LED_Flash SDK 工程.....	242
2.5 程序分析.....	246
2.6 本章小结.....	253
S02_CH03_EMIO 实验.....	254
3.1 EMIO 和 MIO 的对比介绍.....	254
3.2 电路分析与实验现象.....	254
3.3 创建 VIVADO 工程.....	254
3.4 创建约束文件.....	256
3.5 产生 bit 文件并导入到 SDK 中.....	258
3.6 程序分析.....	264
3.7 本章小结.....	264
S02_CH04_User_IP 实验.....	266
4.1 创建 IP.....	266
4.2 调用自定义 IP.....	270
4.3 导入到 S D K	272
4.4 本章小结.....	273
S02_CH05_UBOOT 实验.....	274
5.1 什么是固化.....	274
5.2 固化的流程.....	274
5.3 固化准备.....	274
5.4 zynq 的从 SD 卡的启动的过程.....	275
5.5 zynq 启动模式位的选择.....	275
5.6 BOOT.bin 制作过程详解.....	276
5.7 从 Quad-SPI 启动.....	282
5.8 本章小结.....	283
S02_CH06_XADC 实验.....	284
6.1 实验概述.....	284
6.2 新建一个 VIVADO 工程.....	284
6.3 加载到 SDK.....	285
6.4 函数介绍.....	291
6.5 本章小结.....	291
S02_CH07_ZYNQ PL 中断请求.....	292
7.1 ZYNQ 中断介绍.....	292
7.1.1 ZYNQ 中断框图.....	292
7.1.2 ZYNQ CPU 软件中断 (SGI).....	293
7.1.3 ZYNQ CPU 私有端口中断.....	294
7.2 搭建硬件地址.....	296
7.3 加载到 SDK.....	298
7.4 程序分析.....	302
7.5 本章小结.....	309
S02_CH08_ZYNQ 定时器中断实验.....	310
8.1 中断原理.....	310

8.1.1 软件中断(SGI).....	310
8.1.2 共享中断 SPI.....	310
8.1.3 私有中断 (PPI)	311
8.1.4 私有定时器.....	311
8.2 搭建硬件工程.....	311
8.3 加载到 SDK.....	314
8.4 程序分析.....	317
8.5 本章小结.....	323
S02_CH09_UART 串口中断实验.....	324
9.1 加载到 SDK.....	324
9.2 程序分析.....	328
9.3 本章小结.....	336
S02_CH10_User GPIO 实验.....	337
10.1 创建 IP.....	337
10.2 搭建硬件工程.....	357
10.3 加载到 SDK.....	359
10.4 程序分析.....	361
10.4 本章小结.....	361
11.1 方案框架.....	362
11.2 硬件工程搭建.....	362
11.3 加载到 SDK.....	372
11.4 本章小结.....	377
S02_CH12_AXI_Lite 总线详解.....	378
12.1 前言.....	378
12.2 AXI 总线与 ZYNQ 的关系.....	378
12.3 AXI 总线和 AXI 接口以及 AXI 协议.....	378
12.3.1 AXI 总线概述.....	378
12.3.2 AXI 接口介绍.....	379
12.3.3 AXI 协议概述.....	380
12.3.4 AXI 协议之握手协议.....	381
12.4 AXI4-Lite 详解.....	383
12.4.1 AXI4-Lite 源码查看.....	383
12.4.2 AXI-Lite 源码分析.....	387
12.5 观察 AXI4-Lite 总线信号.....	394
12.6 加载到 SDK.....	398
12.7 本章小结.....	401
S02_CH13_AXI_PWM 实验.....	402
13.1 自定义 IP 的封装.....	402
13.2 miz702_pwm 用户 IP 的修改.....	405
13.3 搭建硬件工程.....	414
13.4 加载到 SDK.....	417
13.5 程序分析.....	420
13.6 本章小结.....	420

S02_CH14_EMIO_OLED 实验.....	421
14.1 板载 OLED 硬件原理.....	421
14.1.1 硬件电路简析.....	421
14.1.2 SSD1306 简介.....	422
14.2 OLED 驱动开发思路解析.....	423
14.2.1 SPI 接口.....	423
14.2.2 SSD1306 控制.....	424
14.2.2 Frame Buffer 显示机制.....	427
14.2.3 像素操作函数.....	428
14.2.4 其他 API 的实现.....	428
14.3 OLED 驱动方案实现.....	428
14.4 点阵式 OLED 显示原理.....	428
14.4.1 OLED 简介.....	428
14.4.2 点阵式显示设备显示原理.....	429
14.4.3 字模的获取.....	430
14.5 硬件搭建.....	433
14.6 导入到 SDK.....	435
14.7 本章小结.....	436
S02_CH15_AXI_OLED 实验.....	437
15.1 自定义 IP 的封装.....	437
15.2 SSD1306_OLED_ML 用户 IP 的修改.....	440
15.3 OLED 硬件控制器关键状态机.....	478
15.4 硬件工程搭建.....	489
15.5 导入到 SDK.....	492
15.6 本章小结.....	493
S02_CH16 等精度频率计实验.....	494
16.1 等精度频率计原理.....	494
16.1.1 引言.....	494
16.1.2 频率测量原理.....	494
16.1.3 脉冲计数法.....	495
16.1.4 周期测频法.....	495
16.1.5 多周期同步测频原理及误差分析.....	495
16.2 等精度频率计设计.....	497
16.2.1 PS 寄存器功能划分.....	497
16.2.2 具体实现.....	498
16.2.3 频率计 PL 部分代码设计.....	498
16.3 硬件工程搭建.....	500
16.4 导入到 SDK.....	500
16.5 误差分析.....	502
16.6 本章小结.....	502
【第三季】ZYNQ SOC 裸奔应用方案共 18 课时.....	503
S03_CH01_AXI_DMA_LOOP 环路测试.....	504
1.1 概述.....	504

1.2 搭建硬件系统.....	504
1.2.1 新建 VIVADO 工程.....	504
1.2.2 创建 VIVADO 硬件构架.....	507
1.3 PS 部分软件分析.....	515
1.3.1 新建 SDK 工程.....	515
1.3.2 main.c 源码的分析.....	517
1.3.3 dma_intr.c 源码分析.....	520
1.3.4 dam_intr.h 文件分析.....	525
1.4 测试结果.....	527
S03_CH02_AXI_DMA PL 发送数据到 PS.....	529
1.1 概述.....	529
1.2 系统构架框图.....	529
1.2.1 ZYNQ IP 的设置.....	529
1.3 PS 部分.....	532
1.4 测试结果.....	534
S03_CH03_AXI_DMA_OV7725 摄像头采集系统.....	536
3.1 概述.....	536
3.2 系统构架.....	536
3.2.1 构架方案图.....	536
3.2.2 构 BLOCK 模块化设计方案图.....	537
3.3 vid in IP 介绍.....	537
3.3.1 OV_Sensor_ML 自定义 IP 模块.....	537
3.3.2 vid in IP 模块.....	545
3.3.2 VID_IN IP 接口信号的定义.....	546
3.4 VTC IP 的分析.....	552
3.4.1 VTC IP 的参数介绍.....	552
3.4.2 VTC IP 接口信号的定义.....	555
3.4.3 VTC IP 配置寄存器.....	559
3.4.5 设置 VTC IP.....	566
3.6 PLL 时钟设置.....	567
3.7 VID_OUT IP 的分析.....	568
3.7.1 VID_OUT 的参数介绍.....	568
3.7.2 VID_OUT IP 接口信号的定义.....	569
3.8 FPGA 实现的用户逻辑代码.....	572
3.8.1 关键信号 1.....	572
3.8.2 关键信号 2.....	573
3.8.3 关键信号 3.....	573
3.8.4 部分关键代码.....	573
3.9 PS 部分.....	575
3.9.1 DMA 中断函数部分分析.....	575
3.9.2 main.c 文件.....	580
3.10 实验效果.....	582
S03_CH04_AXI_DMA_OV5640 摄像头采集系统.....	583

4.1 概述.....	583
4.2 系统构架.....	583
4.2.1 构架方案图.....	583
4.2.2 构 BLOCK 模块化设计方案图.....	584
4.3 vid in IP 介绍.....	584
4.3.1 OV_Sensor_ML 自定义 IP 模块.....	584
4.3.2 vid in IP 模块.....	592
4.3.2 VID_IN IP 接口信号的定义.....	593
4.4 VTC IP 的分析.....	599
4.4.1 VTC IP 的参数介绍.....	599
4.4.2 VTC IP 接口信号的定义.....	602
4.4.3 VTC IP 配置寄存器.....	606
4.4.5 设置 VTC IP.....	613
4.6 PLL 时钟设置.....	614
4.7 VID_OUT IP 的分析.....	615
4.7.1 VID_OUT 的参数介绍.....	615
4.7.2 VID_OUT IP 接口信号的定义.....	616
4.8 FPGA 实现的用户逻辑代码.....	619
4.8.1 关键信号 1.....	619
4.8.2 关键信号 2.....	620
4.8.3 关键信号 3.....	620
4.8.4 部分关键代码.....	620
4.9 PS 部分.....	622
4.9.1 DMA 中断函数部分分析.....	622
4.9.2 main.c 文件.....	627
4.10 实验效果.....	629
S03_CH05_AXI_DMA_HDMI 图像输出.....	630
5.1 概述.....	630
5.2 系统构架.....	630
5.2.1 构架方案图.....	630
5.2.2 构 BLOCK 模块化设计方案图.....	631
5.3 vid in IP 介绍.....	631
5.3.1 OV_Sensor_ML 自定义 IP 模块.....	631
5.3.2 vid in IP 模块.....	639
5.3.2 VID_IN IP 接口信号的定义.....	640
5.4 VTC IP 的分析.....	646
5.4.1 VTC IP 的参数介绍.....	646
5.4.2 VTC IP 接口信号的定义.....	649
5.4.3 VTC IP 配置寄存器.....	653
5.4.5 设置 VTC IP.....	660
5.6 PLL 时钟设置.....	661
5.7 VID_OUT IP 的分析.....	662
5.7.1 VID_OUT 的参数介绍.....	662

5.7.2 VID_OUT IP 接口信号的定义.....	664
5.8 FPGA 实现的用户逻辑代码.....	667
5.8.1 关键信号 1.....	667
5.8.2 关键信号 2.....	667
5.8.3 关键信号 3.....	668
5.8.4 部分关键代码.....	668
5.9 PS 部分.....	670
5.9.1 DMA 中断函数部分分析.....	670
5.9.2 main.c 文件.....	675
5.10 实验效果.....	677
S03_CH06_AXI_VDMA_OV7725 摄像头采集系统.....	678
6.1 为什么要用 VDMA.....	678
6.1.1 什么是帧缓存.....	678
6.1.2 双缓冲机制.....	678
6.1.3 Zynq 硬件架构.....	680
6.1.4 VDMA 的作用.....	680
6.2 VDMA 概述.....	681
6.3 VDMA 详细介绍.....	682
6.3.1 接口.....	682
6.3.2 VDMA 帧存格式.....	683
22.3.3 读写通道工作时序.....	683
6.3.4 寄存器.....	684
6.3.5 帧同步选项.....	690
6.3.6 Genlock 同步机制.....	690
6.4 使用 VDMA.....	692
6.4.1 IP 核配置.....	692
6.4.2 软件控制流程.....	693
6.5 搭建 VDMA 图像系统.....	694
6.5.1 构架方案图.....	694
6.5.2 构 BLOCK 模块化设计方案图.....	695
6.6 PS 部分.....	695
6.6.1 main 函数.....	695
6.6.2 vdma_api.c 函数.....	697
StartTransfer 启动 VDMA 读写通道.....	703
6.7 测试结果.....	704
S03_CH07_AXI_VDMA_OV5640 摄像头采集系统.....	705
7.1 概述.....	705
7.2 搭建 VDMA 图像系统.....	705
7.2.1 构架方案图.....	705
7.2.2 构 BLOCK 模块化设计方案图.....	706
7.3 PS 部分.....	706
7.4 测试结果.....	708
S03_CH08_DMA_LWIP 以太网传输.....	709

8.1 概述.....	709
8.2 搭建硬件系统.....	709
8.2.1 系统构架.....	709
8.2.1 启用 HP 接口.....	709
8.2.2 启用 PL 到 PS 的中断资源.....	710
8.2.3 启动 PS 部分的以太网接口.....	710
8.2.4 时钟的设置.....	710
8.2.5 DMA IP 配置.....	711
8.2.6 GPIO 的配置.....	711
8.2.7 配置 axi_data_fifo_0.....	712
8.2.8 设置 S_AXIS 接口.....	712
8.2.9 地址空间映射.....	713
8.3 FPGA 的发送代码.....	713
8.4 PS 部分 BSP 设置.....	715
8.4.1 SDK 工程 BSP 设置.....	715
8.4.2 lwip 函数库设置.....	715
8.5 PS 部分程序分析.....	717
8.5.1 main.c 分析.....	717
8.5.2 AXI DMA 数据传输过程.....	720
8.6 连接测试.....	724
S03_CH09_DMA_4_Video_Switch 视频切换系统.....	728
9.1 概述.....	728
9.2 修改 OV_Sensor_ML 摄像头采集 IP.....	728
9.3 搭建硬件系统.....	730
9.3.1 系统图.....	730
9.3.2 OV_Sensor_ML IP 接线图.....	731
9.3.3 vid_in IP 的接线图.....	732
9.3.4 DMA 和 FIFO 通路.....	732
9.3.5 vid_out IP 的通路.....	733
9.3.6 AXI HP 通道和 DMA 中断.....	733
9.3.7 DMA IP 的设置.....	734
9.3.8 时钟管理模块.....	735
9.3.9 VTC 图像时序发生模块.....	735
9.4 FPGA 四路输入以及图像切换源码分析.....	735
9.4.1 按钮输入去抖代码.....	735
9.4.2 DMA 4 路视频输入的 FPGA 代码.....	736
9.4.3 DMA 输出通道.....	738
9.5 4 路视频切换 DMA C 处理源码分析.....	740
9.5.1 main.c 源码.....	740
9.5.2 dma_intr.h 源码.....	744
9.5.3 dma_intr.c 中断接收源码.....	747
9.5.4 dma_intr.c 中断发送源码.....	750
9.6 本章小结.....	754

S03_CH10_DMA_4_Video_Stitch 视频拼接系统.....	755
10.1 概述.....	755
10.2 修改 OV_Sensor_ML 摄像头采集 IP.....	755
10.3 搭建硬件系统.....	757
10.3.1 系统图.....	757
10.3.2 OV_Sensor_ML IP 接线图.....	757
10.3.3 vid_in IP 的接线图.....	759
10.3.4 DMA 和 FIFO 通路.....	759
10.3.5 vid_out IP 的通路.....	760
10.3.6 AXI HP 通道和 DMA 中断.....	760
10.3.7 DMA IP 的设置.....	761
10.3.8 VTC 图像时序发生模块.....	762
10.4 FPGA 四路输入以及图像拼接源码分析.....	762
10.4.1 图像常量参数.....	762
10.4.2 DMA 4 路视频输入的 FPGA 代码.....	763
10.4.3 DMA 输出通道.....	765
10.5 4 路视频切换 DMA C 处理源码分析.....	767
10.5.4.1 main.c 源码.....	767
10.5.4.2 dma_intr.h 源码.....	772
10.5.4.3 dma_intr.c 中断接收源码.....	775
10.5.4.4 dma_intr.c 中断发送源码.....	778
10.6 测试结果.....	782
S03_CH11_基于 TCP 的 QSPI Flash bin 文件网络烧写.....	783
11.1 概述.....	783
11.2 基本原理.....	783
11.3 Bin 文件.....	783
11.4 QSPI Flash.....	784
11.5 驱动程序.....	785
11.5.1 建立 TCP Server.....	785
11.5.2 lwip 库设置.....	785
11.5.3 程序解析.....	786
11.5.4 接收保存 BOOT.bin 文件.....	787
11.5.5 烧写 QSPI Flash.....	787
11.5.6 TCP 调试信息输出.....	788
11.6 网络调试助手操作方法.....	788
11.6.1 发送 bin 文件.....	788
11.6.7 发送启动 Flash 烧写命令.....	789
11.7 Bin 文件更新验证.....	791
11.8 待改进之处.....	791
S03_CH12_基于 UDP 的 QSPI Flash bin 文件网络烧写.....	792
12.1 概述.....	792
12.2 基本原理.....	792

12.2.1 Bin 文件.....	792
12.2.2 QSPI Flash.....	792
12.3 驱动程序.....	792
12.3.1 main 函数.....	792
12.3.2 建立 UDP 连接.....	793
12.3.3 lwip 库设置.....	793
12.3.4 程序解析.....	794
12.3.5 接收保存 BOOT.bin 文件.....	794
12.3.6 烧写 QSPI Flash.....	794
12.3.7 UDP 调试信息输出.....	794
12.4 网络调试助手操作方法.....	795
12.4.1 发送 bin 文件.....	795
12.4.2 发送启动 Flash 烧写命令.....	795
12.5 Bin 文件更新验证.....	797
12.6 待改进之处.....	797
S03_CH13_ZYNQ A9 TCP UART 双核 AMP 例程.....	798
13.1 概述.....	798
13.2 基本原理.....	798
13.2.1 软件中断.....	798
13.2.2 共享内存通信.....	799
13.2.3 双核 BOOT.....	799
13.3 驱动程序.....	800
13.3.3 CORE0 工程.....	800
13.4 CORE1 工程.....	802
13.4.1 main 函数.....	802
13.4.2 初始化软件中断.....	802
13.4.3 响应软件中断.....	802
13.4.4 共享内存数据读出.....	803
13.4.5 触发软件中断.....	803
13.5 工程创建及设置关键步骤.....	803
13.6 工程调试关键步骤.....	805
13.7 网络调试助手操作方法.....	805
13.8 生成 BOOT.bin.....	807
13.9 双核 BOOT 验证.....	808
S03_CH14_通过 BRAM 进行 PS 和 PL 间的数据交互.....	809
14.1 概述.....	809
14.2 基本原理.....	809
14.3 PL 部分设计.....	810
14.3.1 IP 连线图.....	810
14.3.2 PS 配置.....	810
14.3.3 AXI BRAM Controller.....	810
14.3.4 Block Memory Generator.....	811
14.3.5 AXI GPIO.....	813

14.4 逻辑设计.....	814
14.4.1 BRAM 读时序.....	814
14.4.2 BRAM 写时序.....	815
14.5 PS 程序设计.....	815
14.5.1 main 函数.....	815
14.5.2 GPIO 输入输出.....	815
14.5.3 BRAM 数据写入.....	816
14.5.4 BRAM 数据读出.....	816
14.6 程序测试.....	817
14.7 课后习题.....	818
S03_CH15_EMIO 光电通信-FEP 子卡的使用.....	819
15.1 概述.....	819
15.2 基本原理.....	819
15.15.1 88E1512.....	819
15.15.2 88E1512 RGMII 接口时序.....	821
15.3 PL 部分设计.....	823
15.3.1 IP 连线图.....	823
15.3.2 ZYNQ PS 设置.....	823
15.3.3 GMII to RGMII.....	824
15.3.4 时序约束.....	826
15.3.4 IO 口.....	829
15.4 PS 程序设计.....	830
15.4.1 LWIP 库修改.....	830
15.4.2 创建工程.....	835
15.5 程序测试.....	838
15.5.1 电口测试.....	838
15.5.2 光口测试.....	839
S04_CH16_PL_AXI_ETH 光电网络通信.....	842
16.1 概述.....	842
16.2 基本原理.....	842
16.2.1 88E1512.....	842
16.2.2 88E1512 RGMII 接口时序.....	844
16.3 PL 部分设计.....	847
16.3.1 IP 连线图.....	847
16.3.2 ZYNQ PS 设置.....	847
16.3.3 AXI 1G/2.5G Ethernet Subsystem.....	848
16.3.4 AXI Direct Memory Access.....	850
16.3.5 PL 至 PS 的中断.....	852
16.3.6 时序约束.....	853
16.3.7 IO 口.....	857
16.4 PS 程序设计.....	857
16.4.1 LWIP 库修改.....	857
16.4.2 创建工程.....	860

16.5 程序测试.....	862
16.5.1 电口测试.....	862
16.5.2 光口测试.....	864
S03_CH17 基于μGUI 的触摸屏 GUI 界面设计.....	866
17.1 概述.....	866
17.2 基本原理.....	866
17.3 LCD 触摸屏.....	866
17.3.1 液晶屏.....	868
17.3.2 触摸屏.....	869
17.3.3 触摸屏唤醒.....	869
17.3.4 触摸中断.....	869
17.3.5 触摸信息获取.....	870
17.3.6 触摸屏与 Miz ZYNQ 开发板的接口.....	873
17.4 μ GUI 概述.....	874
17.4.1 μGUI 库移植.....	874
17.4.2 颜色空间.....	876
17.4.3 字体大小.....	876
17.5 PL 逻辑框架.....	876
17.5.1 PS 设置.....	877
17.5.2 GUI 界面显示.....	877
17.5.3 AXI PWM.....	882
17.5.4 AXI GPIO.....	883
17.5.5 Clocking Wizard.....	884
17.5.6 IO 口.....	885
17.6 PS 程序设计.....	887
17.6.1 main 函数.....	888
17.6.2 时钟重配置.....	889
17.6.3 PWM 信号输出.....	892
17.6.4 GPIO 输入输出.....	892
17.6.5 I2C 读取触摸信息.....	893
17.6.6 GUI 界面显示.....	893
17.6.6.1 AXI VDMA.....	894
17.6.6.2 显示时序设置.....	894
17.6.7 定时器.....	894
17.6.8 GUI 界面设计.....	895
17.6.8.1 GUI 初始化.....	896
17.6.8.2 窗口 1 设计.....	896
17.6.8.3 窗口 2 设计.....	899
17.6.8.4 窗口 3 设计.....	900
17.6.8.5 窗口 4 设计.....	901
17.6.8.6 窗口 5 设计.....	903
17.7 注意事项.....	904
17.7.1 更改 GUI 分辨率.....	904

17.7.2 SDK 路径设置.....	904
17.7.3 miz701n 的 LD5 不受 GUI 控制.....	904
第四季 LINUX 系统开发开发共计 16 课时.....	905
S04_CH01_搭建工程移植 LINUX/测试 EMMC/VGA.....	906
1.1 概述：	906
1.2 LINUX 开发环境搭建.....	906
1.2.1 虚拟机环境配置（提供下载虚拟机已经完成）	906
1.2.2 下载资源.....	907
1.3 VIVADO 工程的搭建.....	907
1.3.1 VIVADO 硬件工程构架.....	907
1.3.2 时钟设置.....	908
1.4 PS 设置.....	912
1.4.1 PS SDK 测试显示器输出.....	912
1.4.2 测试效果 缺图.....	914
1.4.3 新建 FSBL 工程.....	914
1.4.4 产生设备树.....	915
1.5 编译 u-boot、kernel、设备树和文件系统.....	916
1.5.1 批处理文件.....	916
1.5.2 修改设备树.....	918
1.5.3 添加 framebuffer 驱动.....	920
1.5.4 执行 mk_kernel.sh 编译内核.....	922
1.5.5 执行 mk_bootloader.sh 编译 uboot.....	923
1.5.6 制作 UBOOT.BIN.....	923
1.6 EMMC 8GB 内存测试(MIZ702 不支持).....	923
1.7 测试 framebuffer.....	925
S04_CH02_工程移植 ubuntu 并一键制作启动盘.....	928
2.1 概述.....	928
2.2 搭建硬件系统.....	928
2.3 一键制作.....	928
2.4 运行结果.....	929
S04_CH03_QSPI 烧写 LINUX 系统.....	930
3.1 概述.....	930
3.2 搭建硬件系统.....	930
3.3 修改内核文件.....	930
3.3 编译内核及 uboot.....	933
3.4 制作 qspi 镜像.....	933
3.5 安装 screen.....	934
3.6 一件烧写 QSPI FLASH 1.....	935
3.7 烧写 QSPI FLASH 2.....	936
S04_CH04_自动挂载 8GB EMMC 板载内存.....	939
4.1 概述.....	939
4.2 执行 source setup_env.sh.....	939

4.3 修改 zynq-7000.dtsi 文件.....	939
4.4 设置 mount_emmc.sh 批处理命令的开机启动.....	941
4.5 烧写程序到 QSPI FLASH.....	943
4.6 验证测试.....	943
4.7 思考为什么.....	944
S04_CH05_在线升级 QSPI 镜像(U 盘方式).....	946
5.1 概述.....	946
5.2 执行 source setup_env.sh.....	946
5.3 烧写程序到 QSPI FLASH.....	946
5.4 查看系统根目录.....	946
5.5 基于 U 盘在线升级.....	947
S04_CH06_hello_linux.....	950
6.1 概述.....	950
6.2 执行 source setup_env.sh.....	950
6.3 SD 卡手动运行 hello 程序.....	950
6.4 EMMC 卡手动运行 hello 程序.....	952
S04_CH07_Hello_Qt 在开发板上的运行.....	953
7.1 概述.....	953
7.2 搭建交叉编译环境.....	953
7.2.1 使用批处理命令搭建交叉编译环境.....	953
7.2.2 setup_env.sh 批处理文件源码.....	958
7.2.3 get_qt_sources.sh 批处理文件源码.....	960
7.2.4 mk_qt_img.sh 批处理文件源码.....	961
7.2.4 init.sh 文件.....	964
7.2.5 测试结果.....	964
7.3 在 PC 端 LINUX 安装 qt5.8.0.....	965
7.4 QtE LINUX PC 端创建工程.....	970
7.5 对 QtE 设置交叉编译.....	977
7.6 测试结果.....	982
第五季 HSL 算法基础入门 12 课时.....	983
S05_CH01_搭建 Modelsim 和 Vivado 联合调试环境.....	984
1.1 概述.....	984
1.2 使用 GUI 编译仿真库.....	984
1.3 使用命令行编译仿真库.....	986
1.4 HLS 简单介绍.....	987
1.4.1 OpenCV 和 HLS 视频库.....	988
1.4.2 AXI4 流和视频接口.....	989
1.4.3 OpenCV 到 RTL 代码转换的流程.....	990
1.5 本章小结.....	990
S05_CH02_shift_led 实验.....	991
2.1 概述.....	991
2.2 工程创建、仿真及优化.....	991
2.2.1 工程创建.....	991

2.2.2 代码综合.....	996
2.2.3 代码优化.....	999
2.2.4 仿真实现.....	1001
2.3 HLS 代码封装.....	1007
2.4 硬件平台实现.....	1008
2.5 本章小结.....	1015
S05_CH03_ImageLoad 实验.....	1016
3.1 概述.....	1016
3.2 图片数据的获取.....	1016
3.3 视频流文件的载入.....	1018
3.4 外部摄像头的调用.....	1020
3.5 工程创建与验证.....	1021
3.6 本章小结.....	1029
S05_CH04_Skin_Dection 实验.....	1030
4.1 肤色检测原理及应用.....	1030
4.2 检测算法实现.....	1030
4.2.1 工程创建.....	1030
4.2.2 代码综合.....	1034
4.2.3 代码优化.....	1038
4.3 仿真测试.....	1043
4.4 本章小结.....	1044
S05_CH05_Sobel 算子硬件实现(一)_HLS 实现.....	1045
5.1 Sobel 原理介绍.....	1045
5.2 Sobel 算子在 HLS 上的实现.....	1046
5.2.1 工程创建.....	1046
5.2.2 代码优化及仿真.....	1050
5.2.3 工程封装.....	1056
5.3 代码详解.....	1057
5.3 本章小结.....	1062
S05_CH06_Sobel 算子硬件实现(二)_硬件验证.....	1063
6.1 系统硬件设计.....	1063
6.2 硬件工程创建.....	1064
6.3 导入到 SDK.....	1068
6.4 程序分析.....	1075
S05_CH07_基于 Hough 变换的圆检测.....	1078
7.1 Hough 变换原理介绍.....	1078
7.1.1 Hough 变换直线检测.....	1078
7.1.2 Hough 变换圆检测.....	1079
7.1.3 Hough 变换圆检测算法实现流程.....	1080
7.2 Hough 在 HLS 上的实现.....	1080
7.2.1 工程创建.....	1081
7.2.2 仿真及优化.....	1089
7.3 程序分析.....	1094

7.4 本章小结.....	1097
S05_CH08_傅里叶变换的 HLS 实现.....	1098
8.1 FFT 原理介绍.....	1098
8.2 HLS 实现.....	1101
8.3 Vivado 模块例化及 IP 封包.....	1105
8.4 本章小结.....	1117
S05_CH09_OTSU 自适应二值化.....	1118
9.1 OTSU 自适应二值化原理简介.....	1118
9.2 HLS 实现.....	1118
9.2.1 工程创建.....	1118
9.2.2 仿真及优化.....	1123
9.3 硬件工程实现（待更新）.....	1123
9.4 程序分析.....	1123
S05_CH10_音频滤波.....	1124
10.1 ADAU1761 简介.....	1124
10.1.1 ADAU1761 收发时序.....	1124
10.1.2 ADAU1761 时钟和采样率.....	1125
10.2 硬件平台的搭建.....	1126
10.3 导入到 SDK.....	1142
10.4 程序分析.....	1150
10.5 本章小结.....	1157
S05_CH11_快速角点检测的 HLS 实现.....	1158
11.1 角点定义.....	1158
11.2 角点检测算法.....	1159
11.2.1 Moravec 角点检测算法.....	1159
11.2.2 Harris 角点检测.....	1159
11.2.3 FAST 角点检测算法.....	1160
11.3 HLS 实现.....	1162
11.3.1 工程创建.....	1162
11.3.2 仿真及优化.....	1164
11.4 硬件工程创建.....	1170
11.4.1 硬件平台搭建.....	1170
11.4.2 导入到 SDK.....	1171
11.5 程序分析.....	1179
11.6 本章小结.....	1179
S05_CH12_HLS 车牌识别（待更新）.....	1180
附录(常见问题汇总).....	1181
一、工具篇.....	1181
1.1 HLS 中文注释乱码问题解决方案.....	1181
1.2 代码字体大小修改.....	1182
二、设计篇.....	1183
2.1 hls::stream 仿真警告.....	1183
2.2 仿真时使用 cvShowImage() 函数但是没有任何错误提示仿真界面直接关闭.....	1183

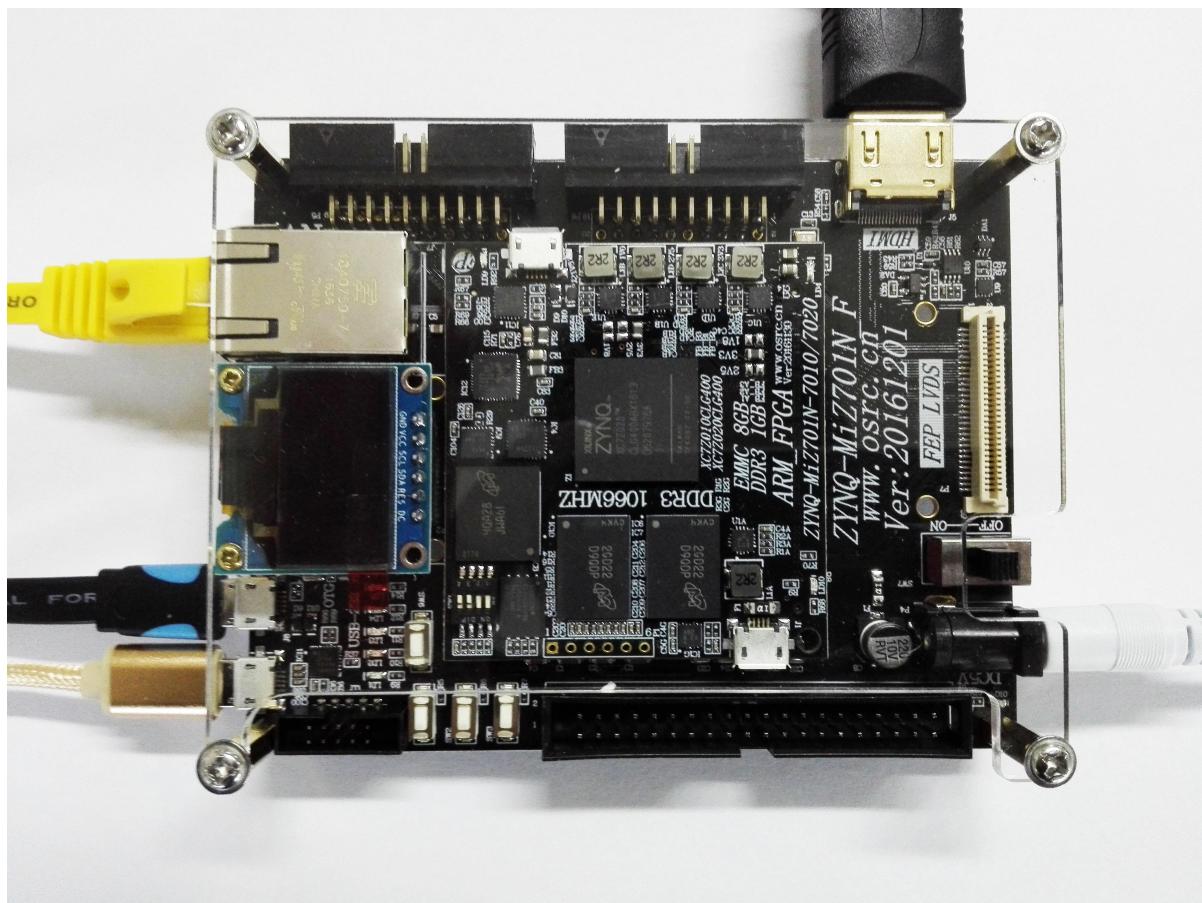
【第一季】ZYNQ SOC 开机及 FPGA 基础 共 12 课时

第一季课程共计 12 课时，主要讲解开机测试，JTAG 下载程序，FPGA 基础语法基础，VIVADO 软件快速入门、VGA 或者 HDMI 接口的测试。

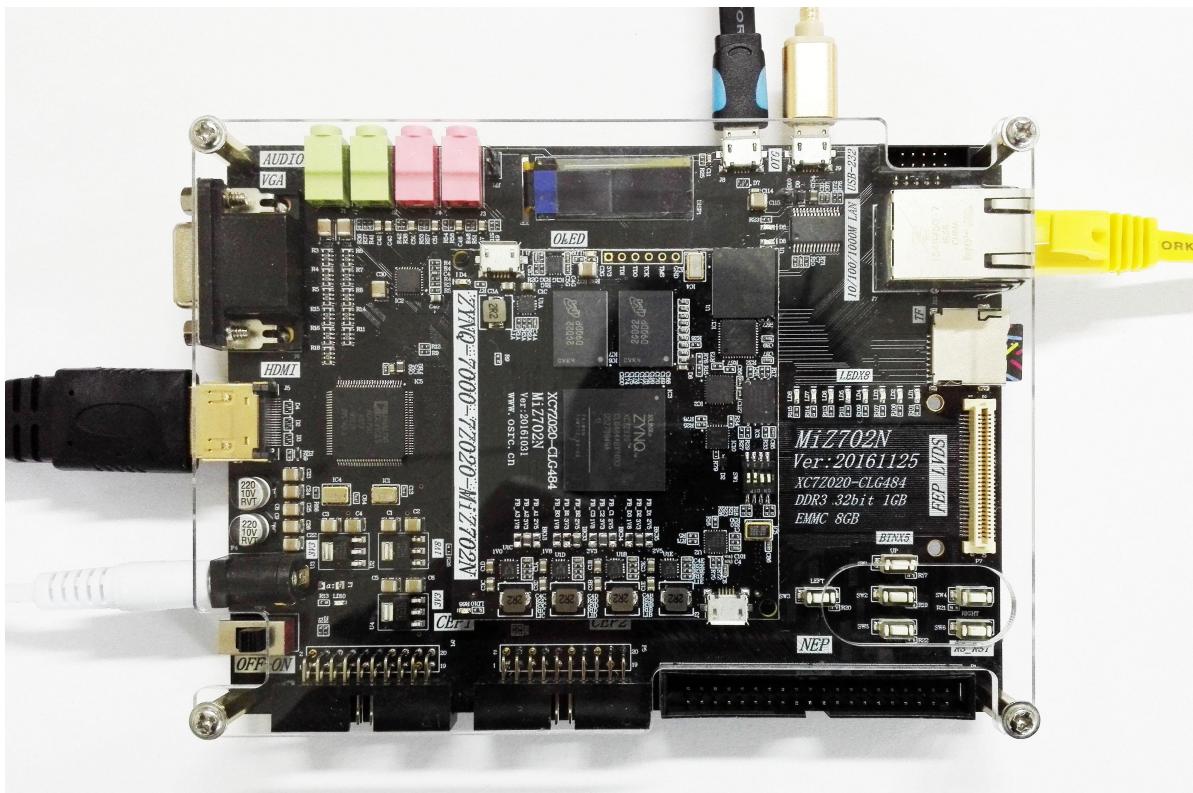
对于所有读者拿到板子后第一件事情应该是对板子做一个开机测试。对于有 FPGA 基础，第一次使用 ZYNQ，第一次使用 VIVADO 软件的读者，可以把软件使用相关课程看下；对于没有 FPGA 基础的，需要把 FPGA 基础的知识好好学习下。对于熟悉 ZYNQ 软件的，也会 FPGA 开发的，可以跳过本章基础部分，直接进入后面章节学习。

S01_CH01_开机程序测试

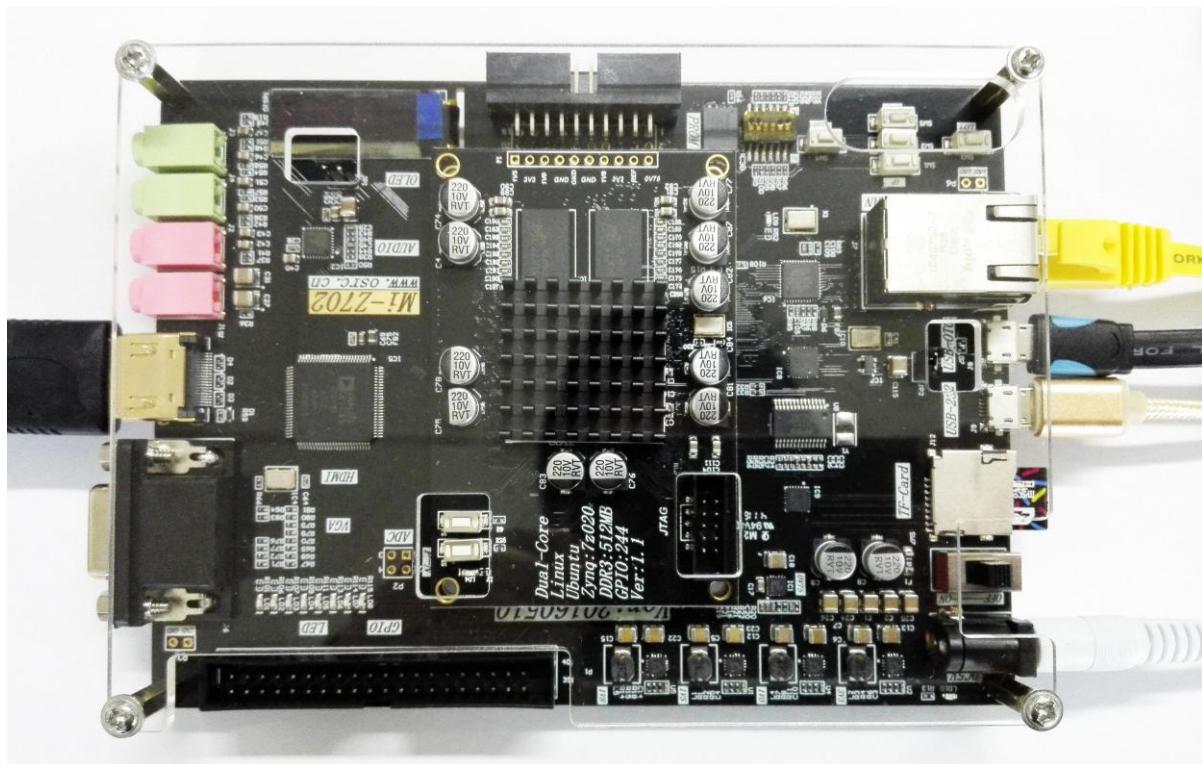
1.1 MIZ701N 开机测试连线图



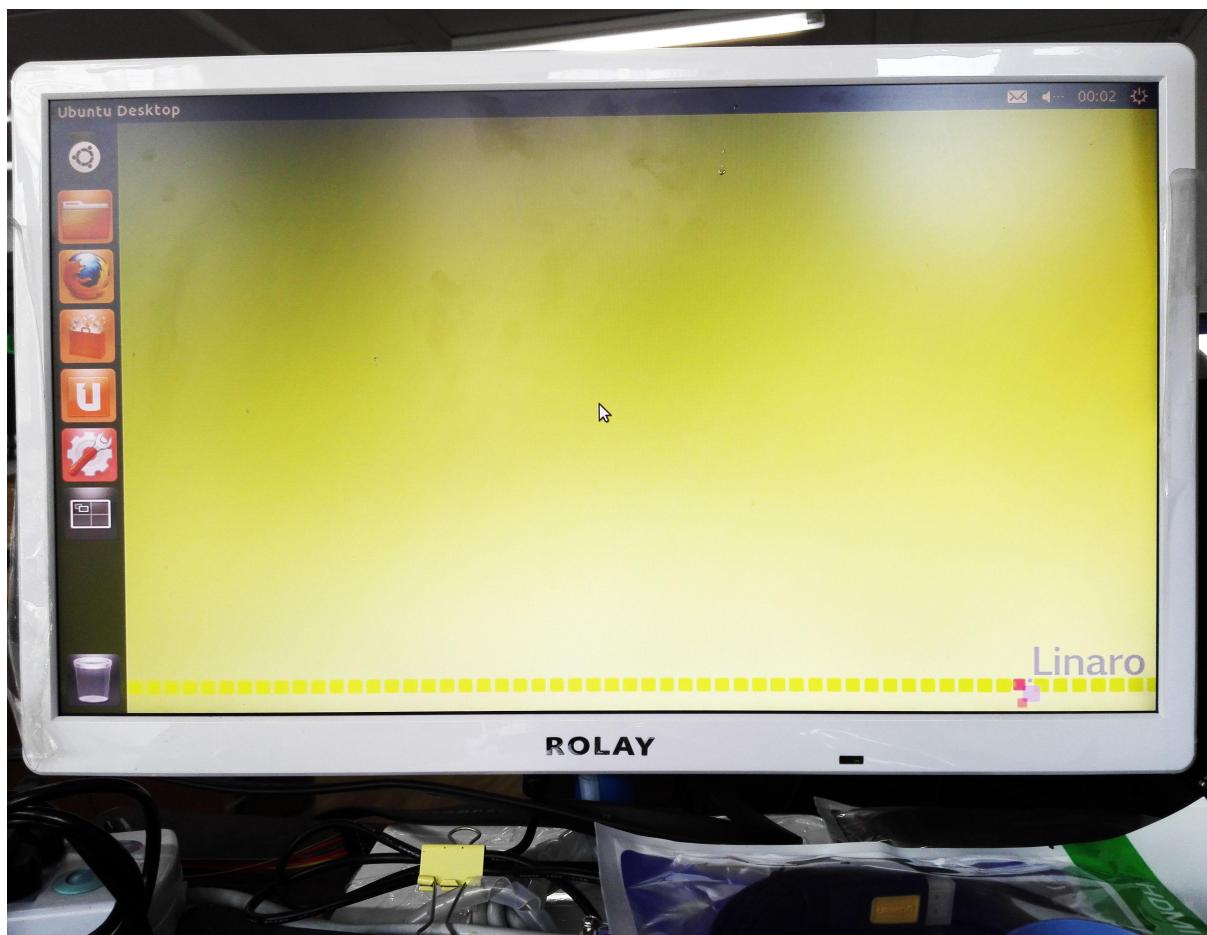
1.2 MIZ702N 开机测试连线图



1.3 MIZ702 开机测试连线图

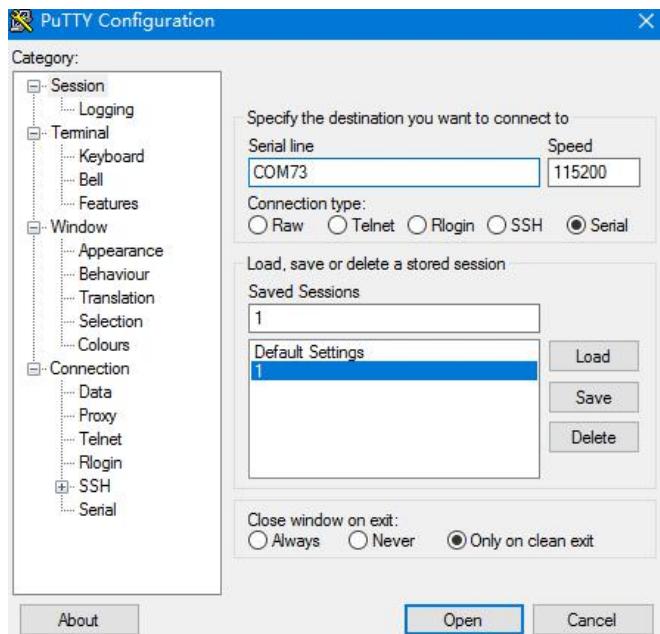


1.5 UBUNTU 系统界面



1.6 串口输出

打开 putty 软件，并且打开开发板对应串口波特率 115200



```

ci_hdrc ci_hdrc.0: new USB bus registered, assigned bus number 1
ci_hdrc ci_hdrc.0: USB 2.0 started, EHCI 1.00
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
mousedev: PS/2 mouse device common for all mice
i2c /dev entries driver
EDAC MC: ECC not enabled
Xilinx Zynq CpuIdle Driver started
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhci-pltfm: SDHCI platform and OF driver helper
mmc0: SDHCI controller on e0100000.sdhci [e0100000.sdhci] using DMA
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
NET: Registered protocol family 10
sit: IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
Registering SWP/SWPB emulation handler
hctosys: unable to open rtc device (rtc0)
ALSA device list:
No soundcards found.
Waiting for root device /dev/mmcblk0p2...
mmc0: new SDHC card at address 0001
mmcblk0: mmc0:0001 0000 7.44 GiB
mmcblk0: p1 p2
EXT4-fs (mmcblk0p2): 8 orphan inodes deleted
EXT4-fs (mmcblk0p2): recovery complete
EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 179:2.
devtmpfs: mounted
Freeing unused kernel memory: 1024K (c0800000 - c0900000)
init: ureadahead main process (658) terminated with status 5
Last login: Thu Jan  1 00:00:07 UTC 1970 on tty1
cat: /var/lib/update-notifier/fsck-at-reboot: No such file or directory
run-parts: /etc/update-motd.d/98-fsck-at-reboot exited with return code 1
Welcome to Linaro 12.11 (GNU/Linux 4.6.0-xilinx armv7l)

* Documentation: https://wiki.linaro.org/

0 packages can be updated.
0 updates are security updates.

root@linaro-ubuntu-desktop:~#

```

1.7 网口测试

Step1: 设置本地主机 IP 地址



Step2: 设置开发板 IP 地址在 putty 软件中输入

```
ifconfig eth0 192.168.1.118
```

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

Step3: 在 PC 中打开控制台 ping 192.168.1.118

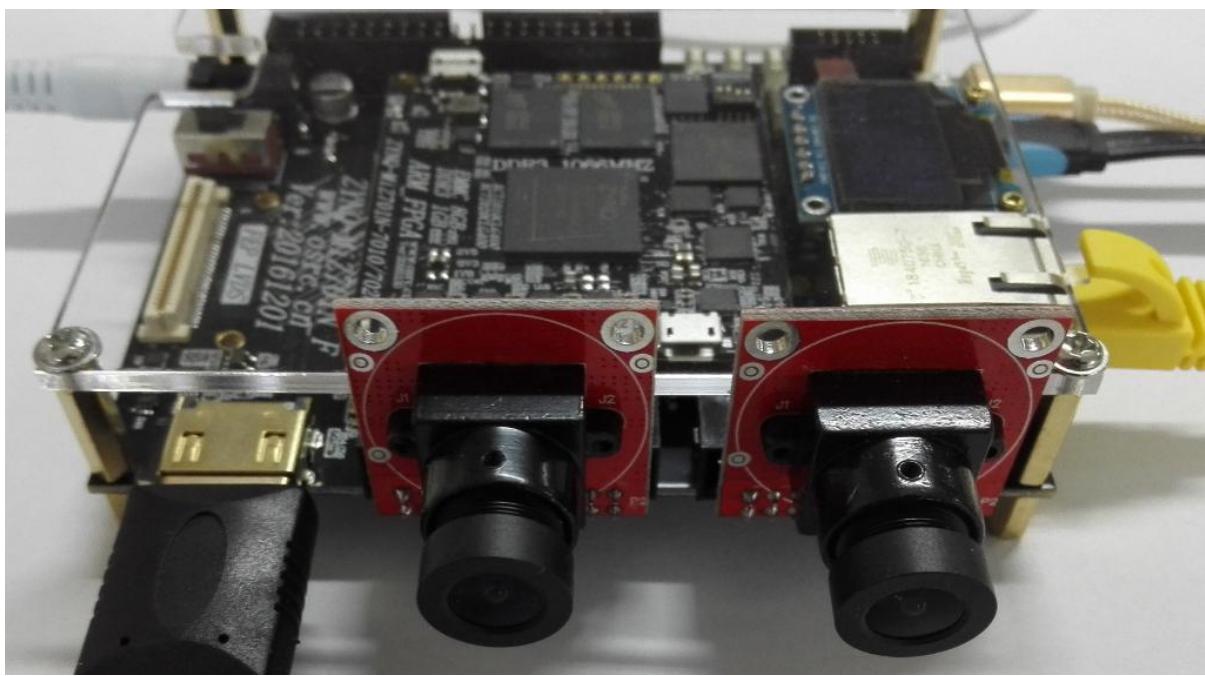
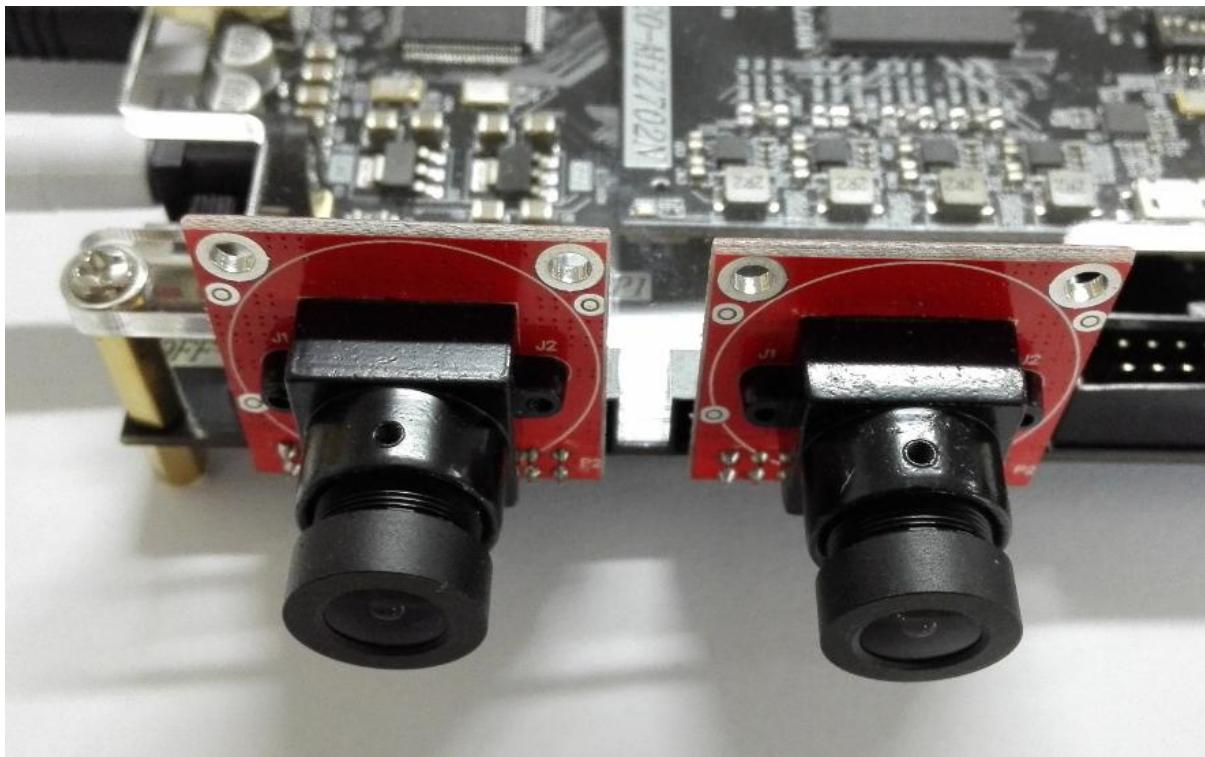
```
C:\Users\Administrator>ping 192.168.1.118

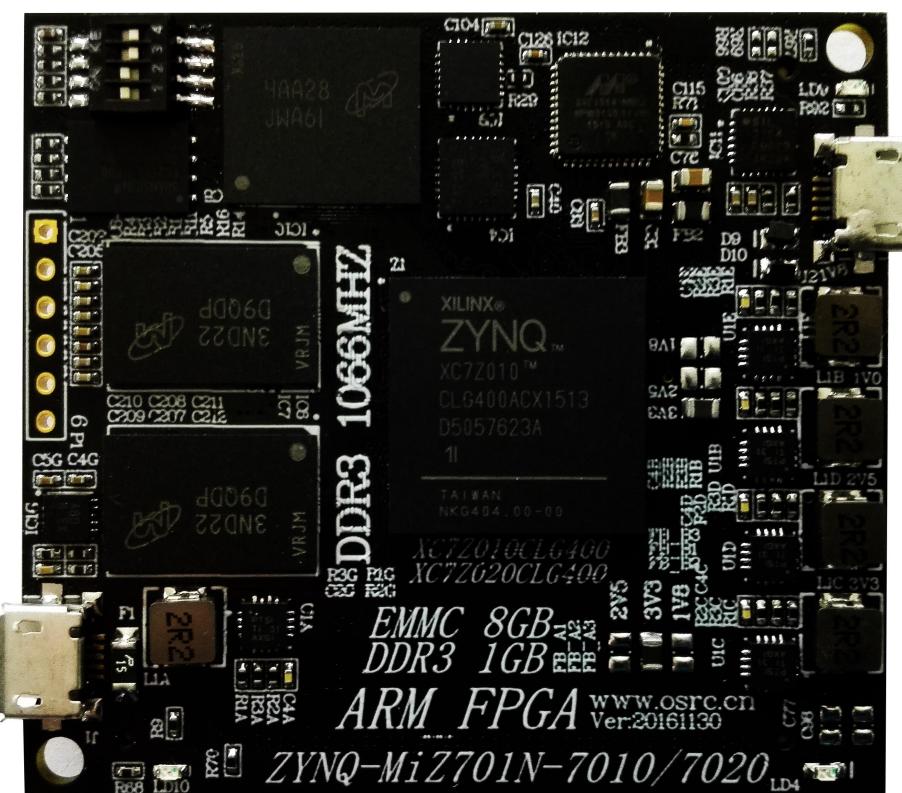
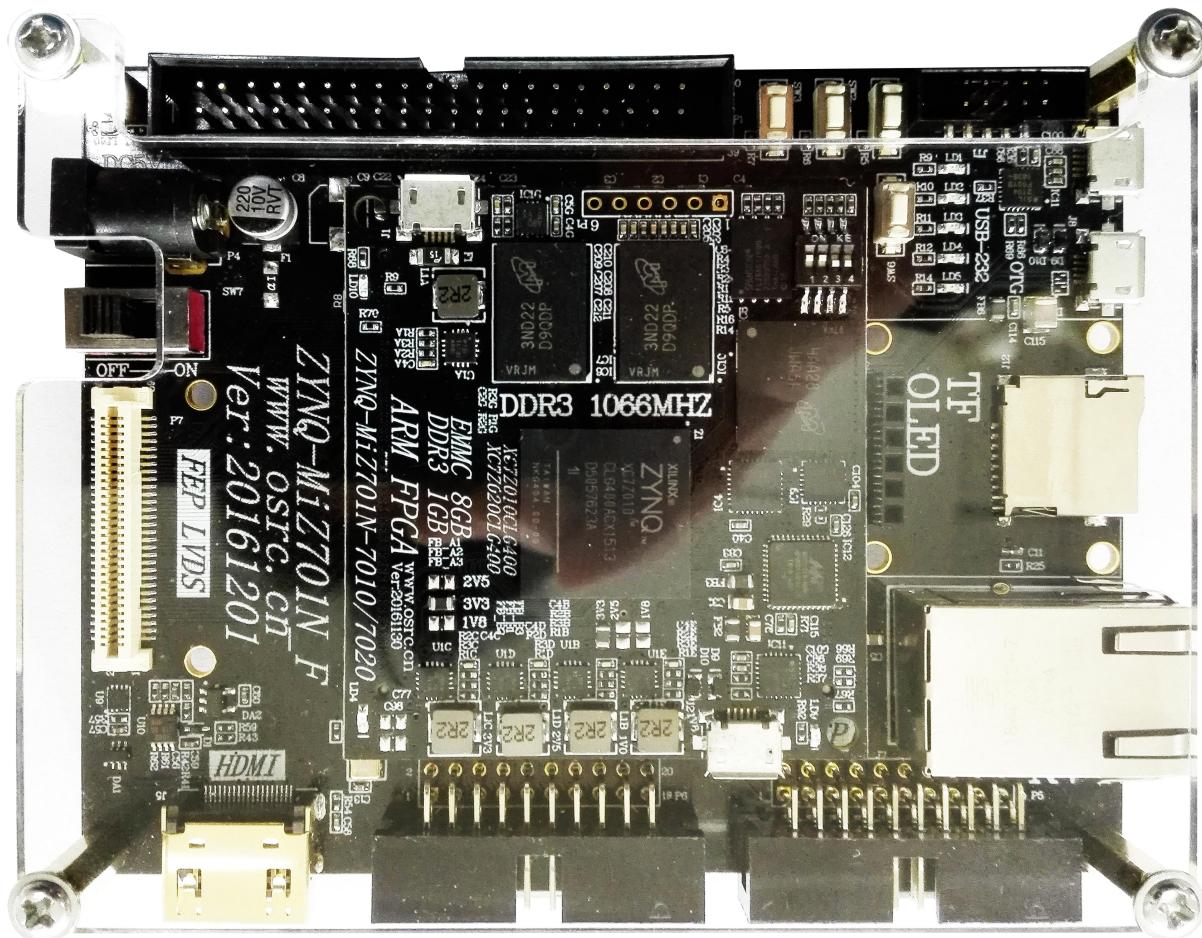
正在 Ping 192.168.1.118 具有 32 字节的数据:
来自 192.168.1.118 的回复: 字节=32 时间<1ms TTL=64

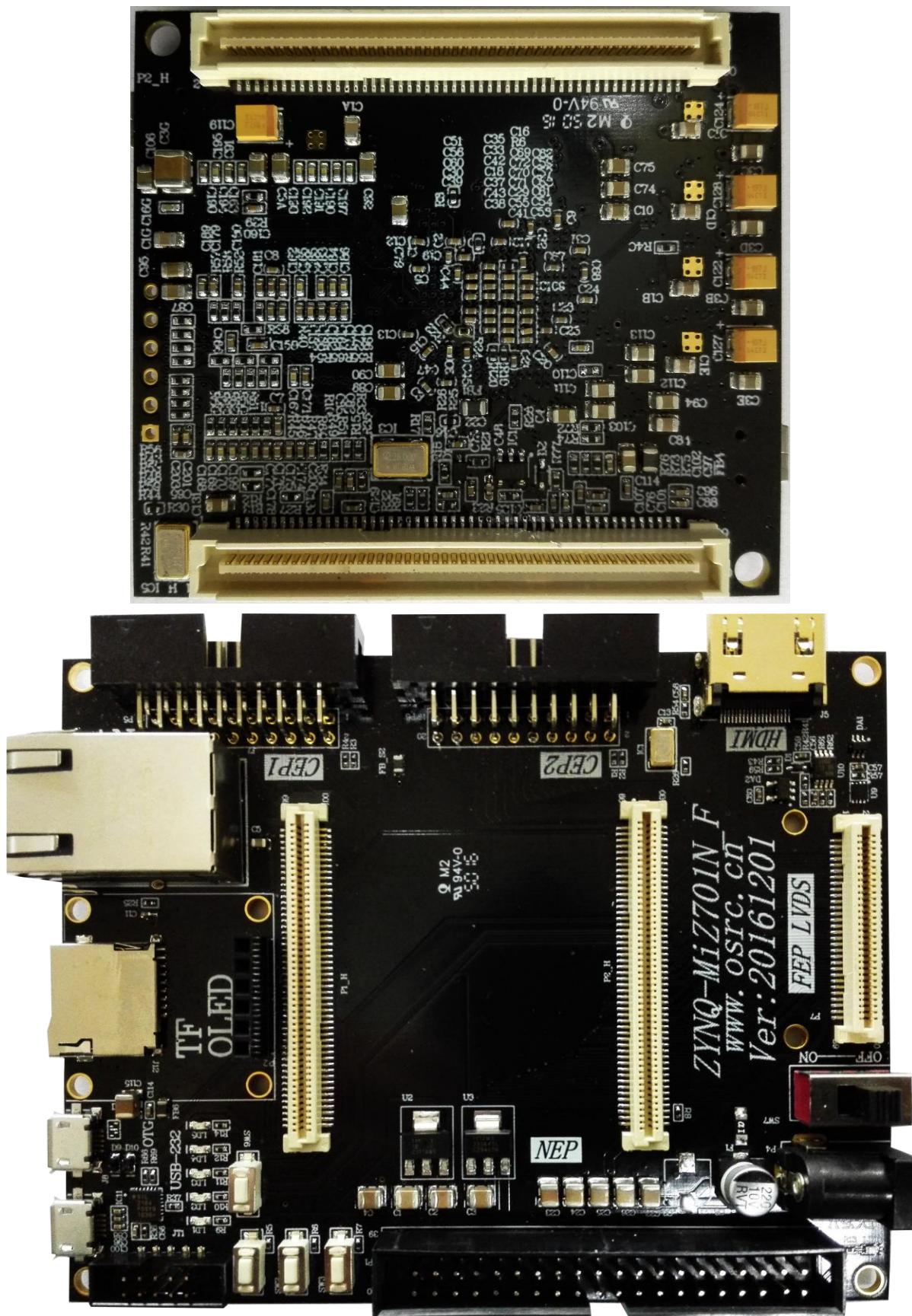
192.168.1.118 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms, 最长 = 0ms, 平均 = 0ms
```

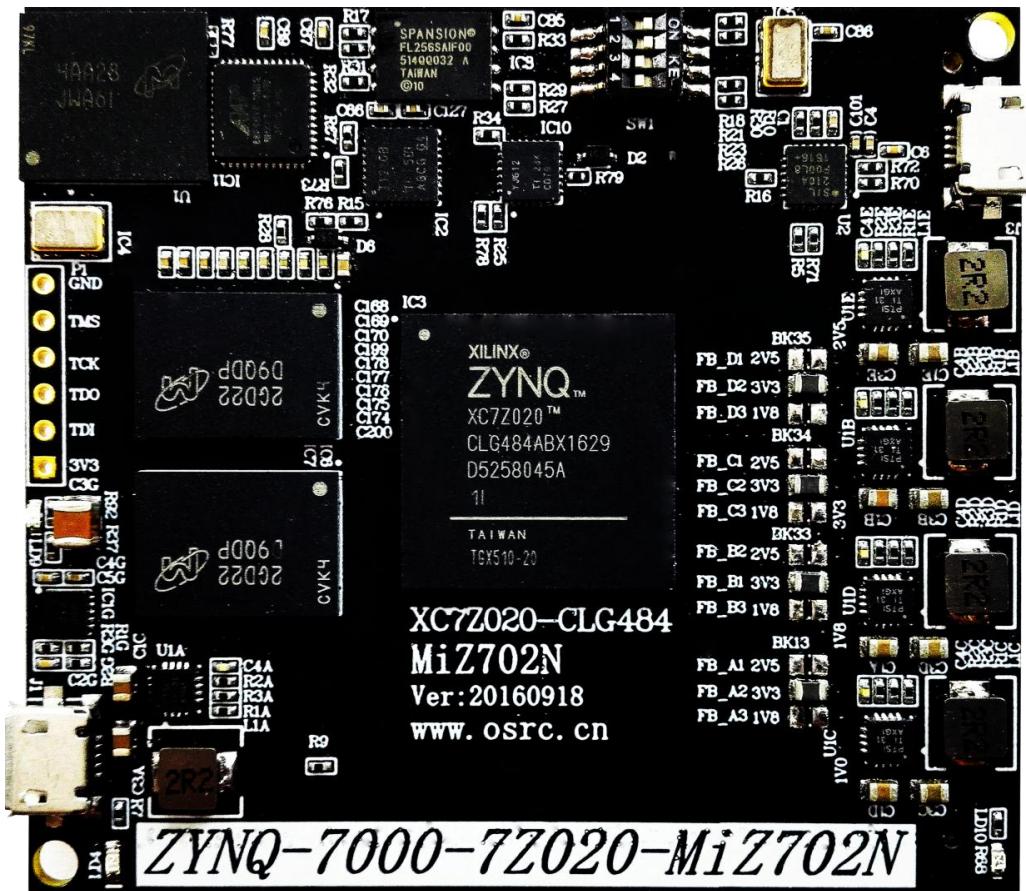
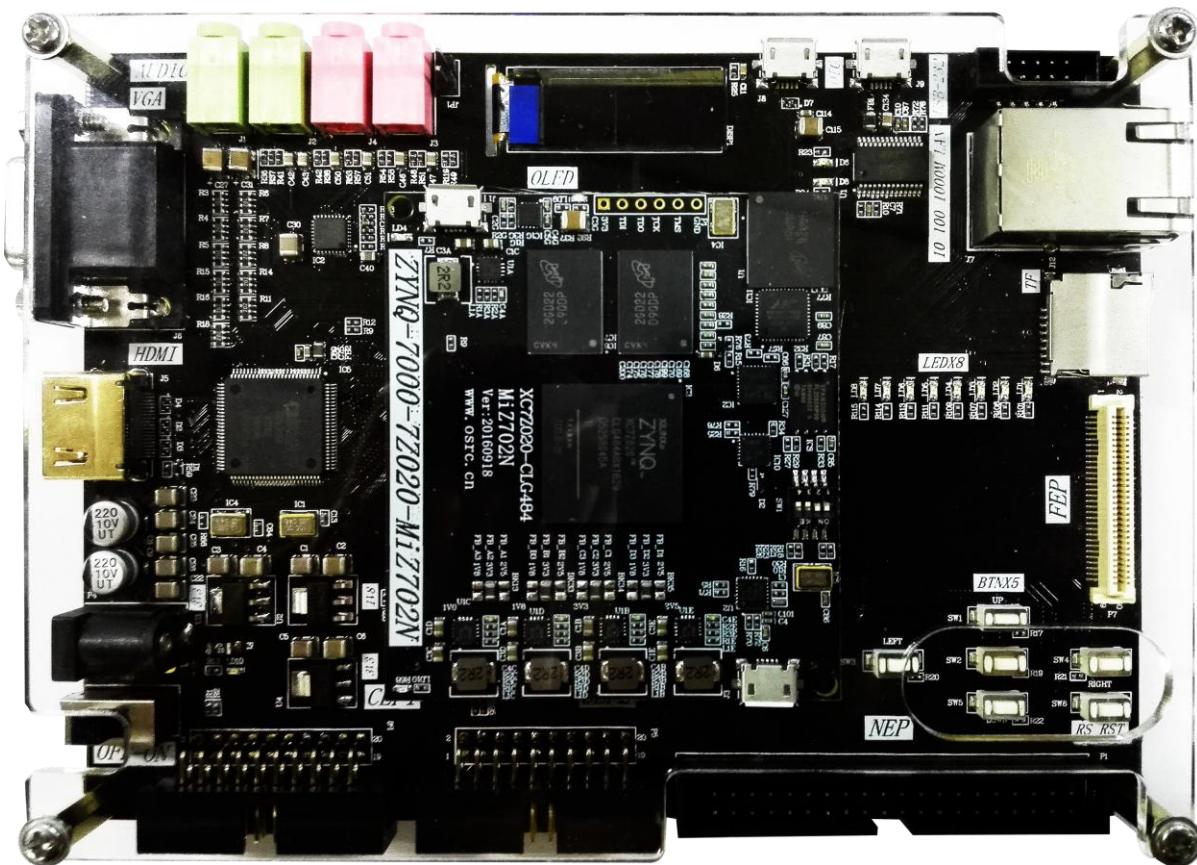
读者如果 ping 不通可以给开发板换一个 IP 地址试试。

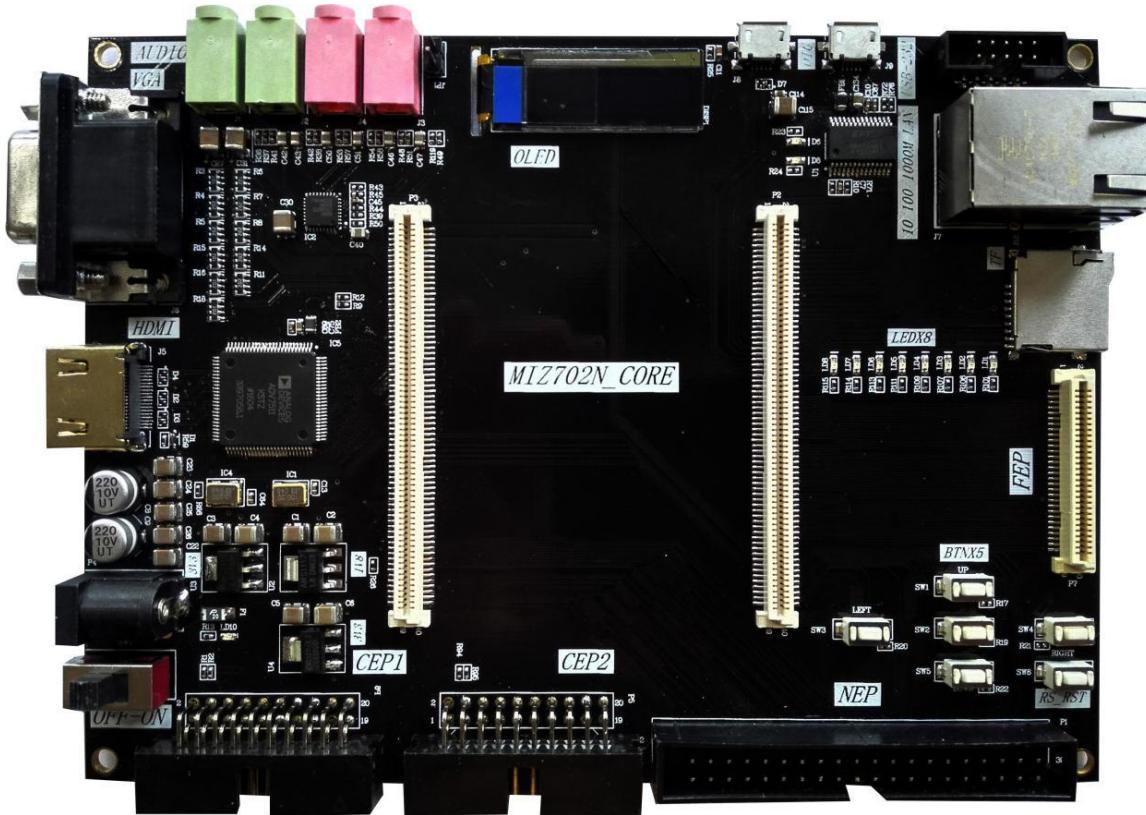
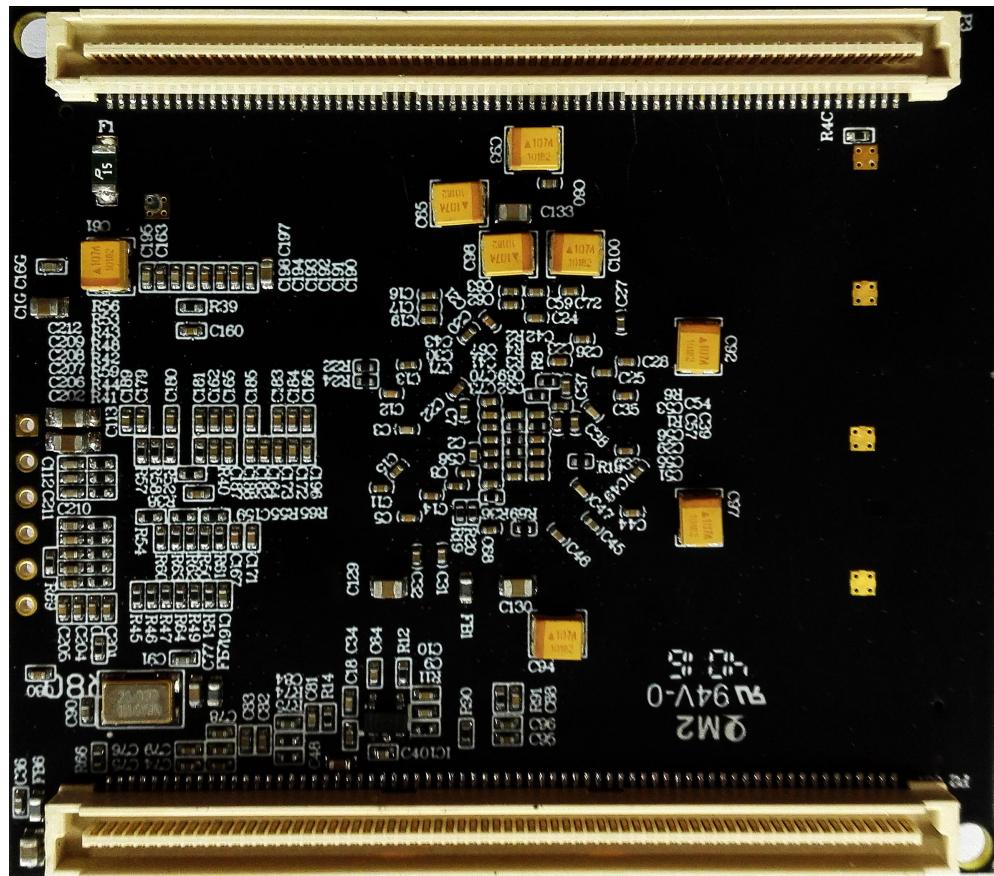
1.8 美图欣赏













S01_CH02_ZYNQ_VIVADO 软件安装

2.1 VIVADO 软件介绍

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

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

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

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

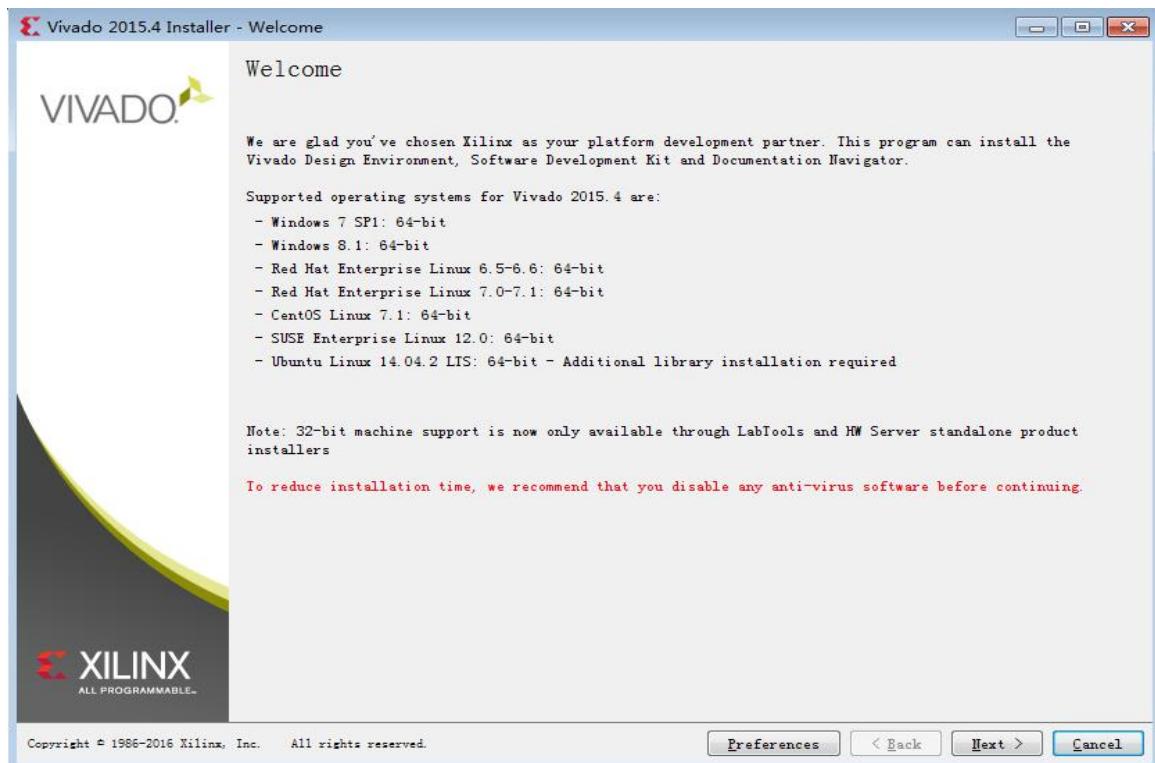
安装 vivado 的过程，其实很简单，但是需要注意一个问题，安装时一定把 SDK 选上，避免不必要的麻烦。

2.2 VIVADO 软件安装(适合所有 vivado 安装)

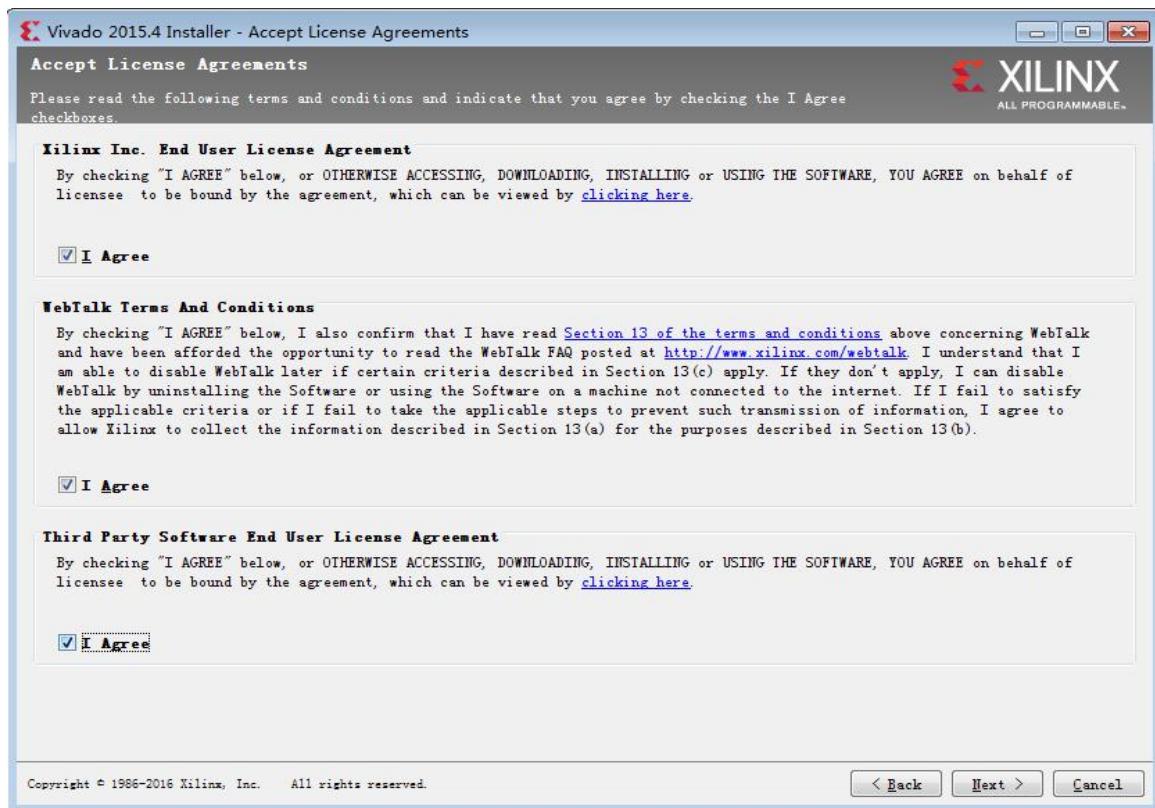
安装软件前请退出杀毒软件，以免造成安装失败，比如 360 安全卫士

Step1: 双击  xsetup.exe 进行程序安装

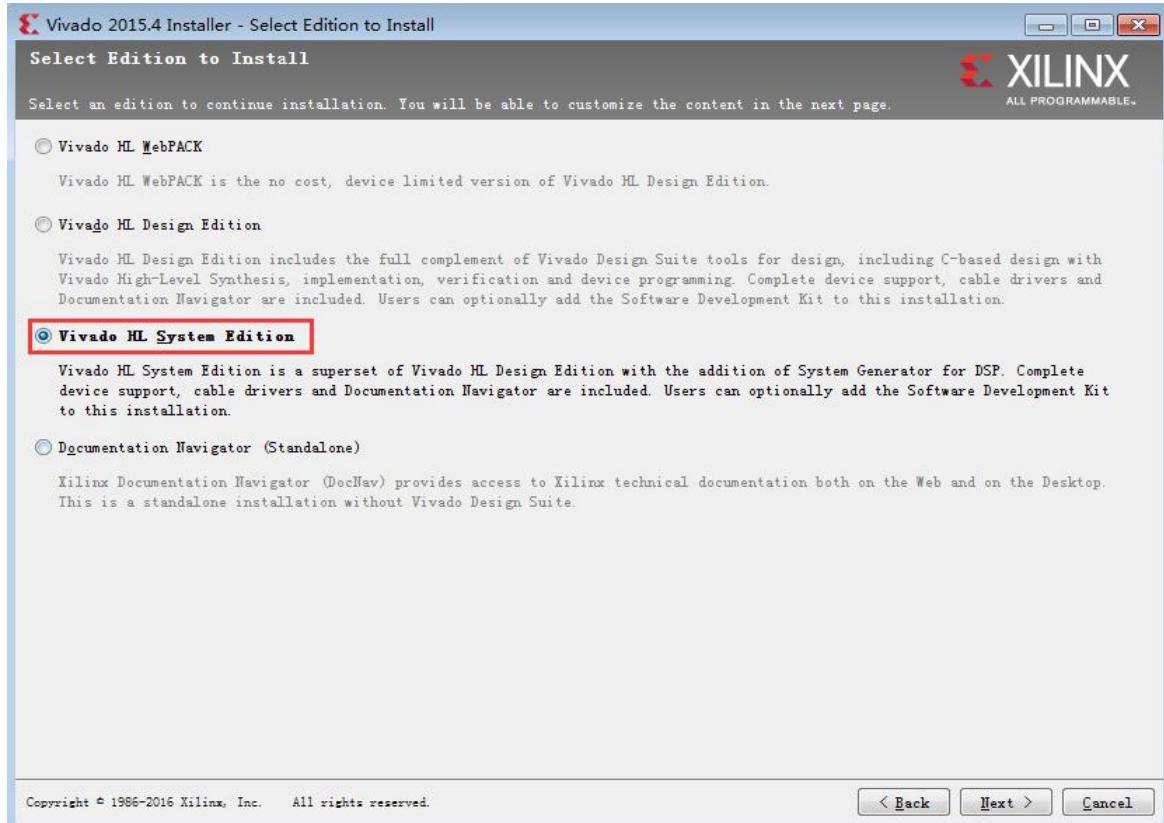
Step2: 驱动欢迎界面后单击 NEXT



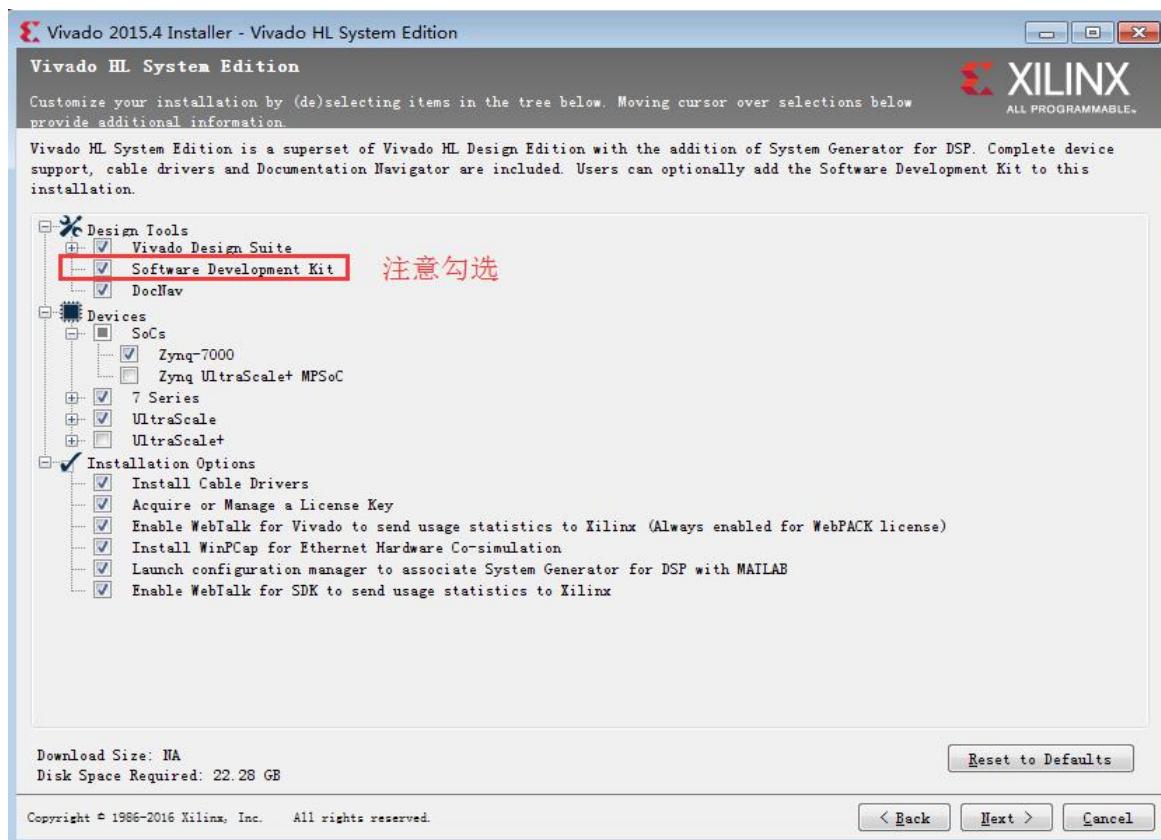
Step3:全勾选上全部，全部同意，单击 NEXT



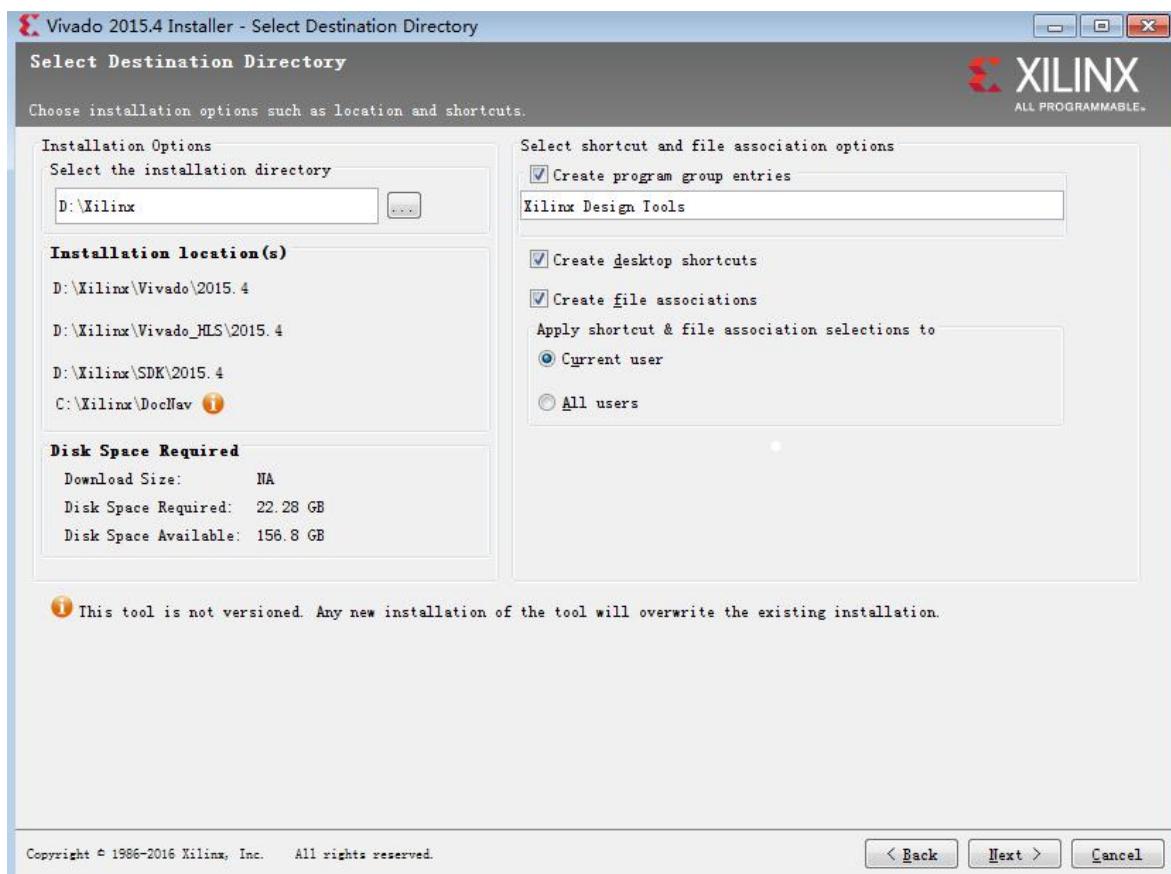
Step4:选择安装 Vivado HL System Edition 单击 NEXT(这个版本功能最全)



Step5:必须勾选上 SDK，其他默认，然后单击 NEXT



Step6:设置安装目录，然后单击 NEXT 进行安装



Step7:安装大概需要半个多小时，

2.3 VIVADO 软件注册

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

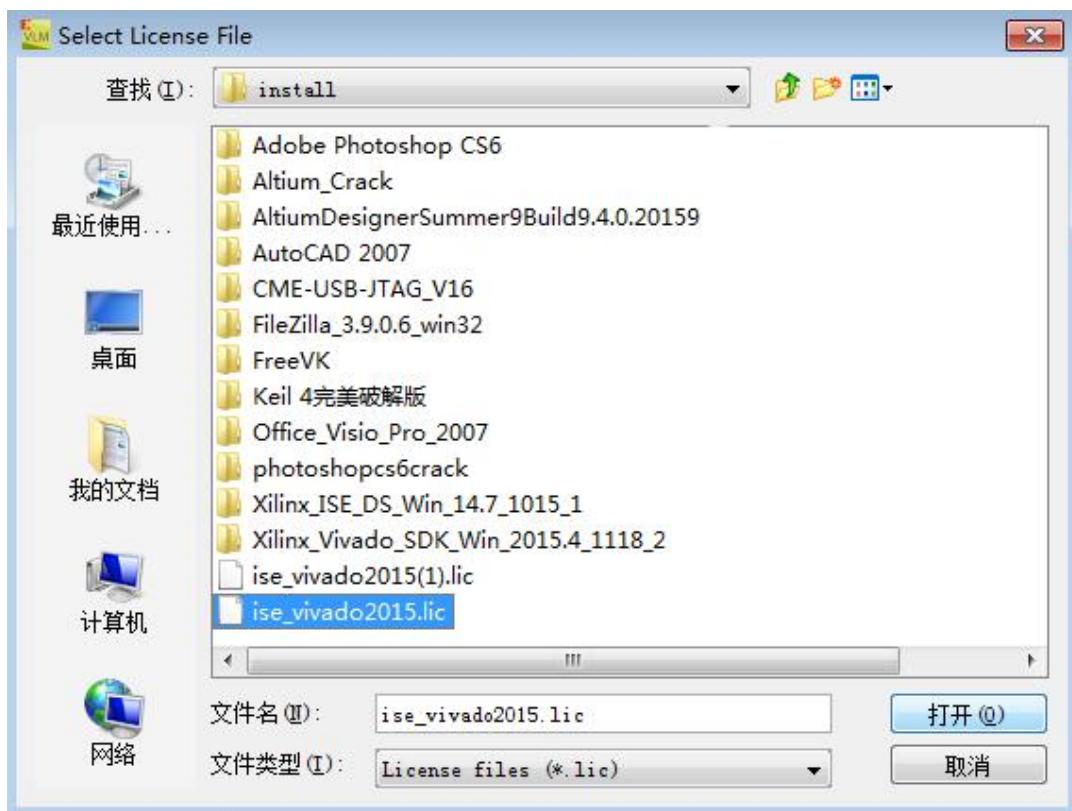


Step2:弹出下图界面后

- 1、单击 load license
- 2、单击 Copy license



Step3: Copy license 选择打开



Step4：弹出添加成功，单击 OK 结束



2.3 本章小结

本章详细描述了如何安装 VIVAOD2015.4 以及如果加载 license,在安装的过程中一定要关闭杀毒软件，以免造成安装失败。

S01_CH03_USB 下载器驱动安装及下载程序

3.1 下载器驱动的安装

默认情况下在上一章节安装 VIVADO 的时候会自动安装好驱动程序。现在发货的都是新版本下载器，插入 USB 接口后，红色的 LED 亮，绿色的灭，当把下载器接口接到开发板，并且给开发板通电后，绿色的 LED 亮。如图所示：



在设备管理器里面可以看到如下设备就是下载器：

- ▼ 通用串行总线控制器
 - Intel(R) USB 3.0 可扩展主机控制器 - 1.0 (Microsoft)
 - USB Composite Device
 - USB Serial Converter**
 - USB 根集线器(xHCI)
 - 通用 USB 集线器
 - 通用 USB 集线器

由于 MIZ702 和 MIZ702N 的串口也会识别成这样，如果读者想知道这个设备是不是 USB JTAG 可以采取先不接通开发板串口，如果设备管理器只有这么一个设备，那就是正常识别了。

大部分时候，安装好 VIVADO 就会自动安装好驱动了，但是也有例外，比如杀毒软件禁止安装驱动了，或者安装 VIVADO 的时候没有勾选安装驱动，或者在使用的过程中驱动损坏了，这个时候可以采取手动安装驱动。

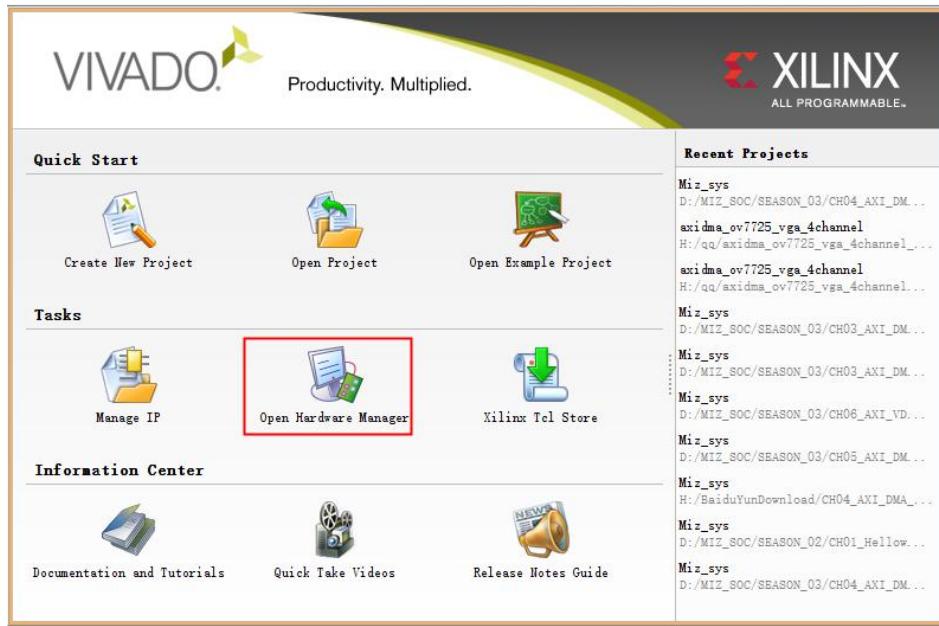
笔者的 VIVADO 是安装在 D 盘下的，驱动安装程序的路径如下图：

软件 (D:) > Xilinx > Vivado > 2015.4 > data > xicom > cable_drivers > nt64 > digilent			
名称	修改日期	类型	大小
install_digilent.exe	2015/11/18 7:10	应用程序	18,745 KB

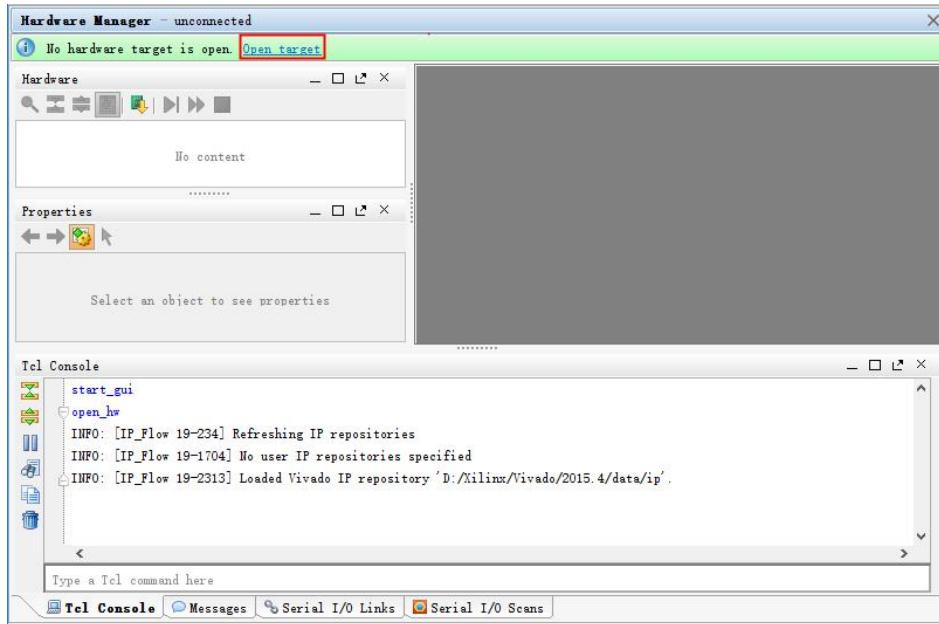
安装驱动的时候需要拔掉下载器，以免安装后出现错误。双击 install_digilent.exe 文件进行安装，一路 NEXT 到结束就按照好了。这个时候插上下载器应该就可以使用了。如果这个时候你还没法使用，就找我们南京米联电子的客服吧。

3.2 下载 runled 工程的 bit 文件验证板子和下载器工作正常

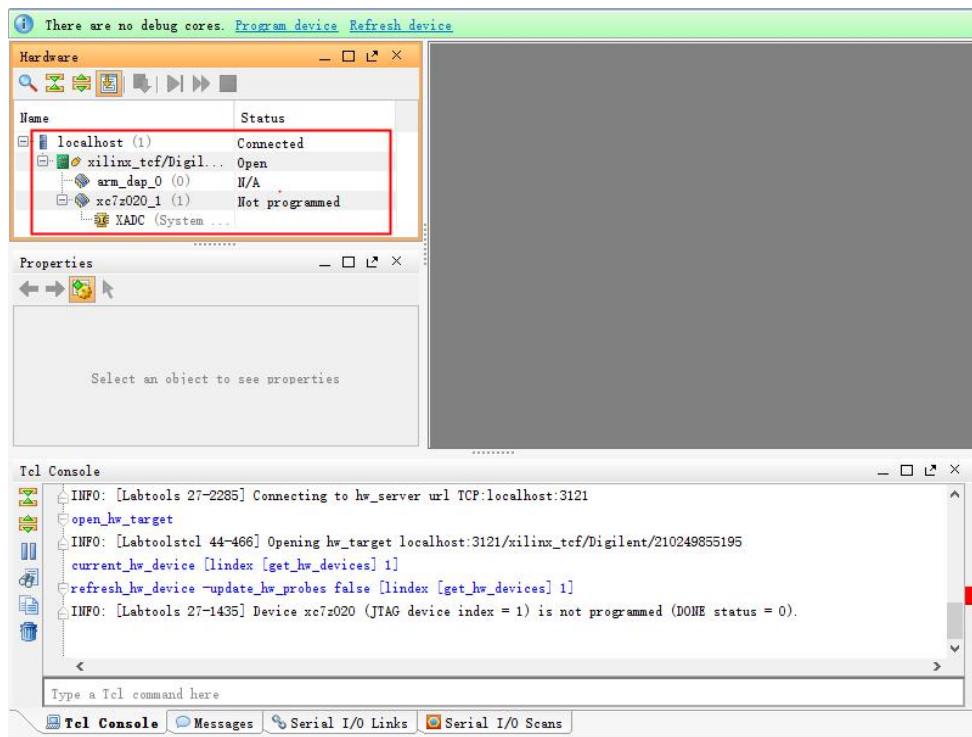
Step1:启动 VIVADO 软件，并且鼠标左击 Open Hardware Manager 图标控件



Step2:鼠标左击 Open_target



Step3:如下图显示，正确检测到了设备



Step4: 下载 CH07 的程序看运行效果

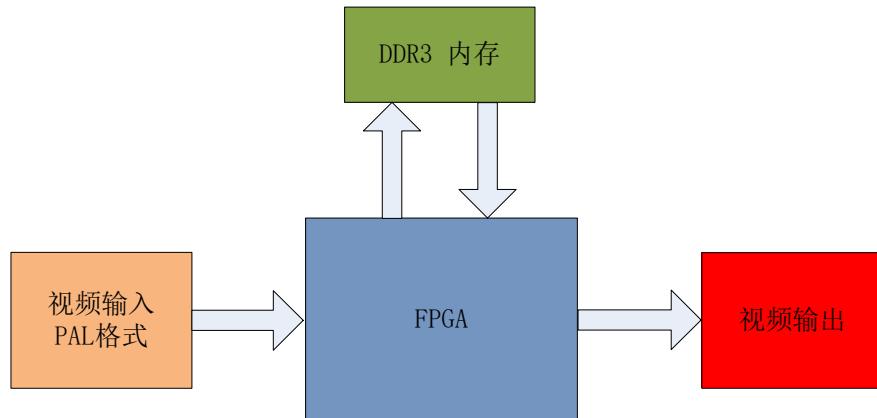
3.3 下载器使用需要注意的问题

- 1、下载器出厂前都经过严格测试了，出问题的几率很低，所以读者在使用的时候特别是第一次使用请对照本课程安装驱动和测试。
- 2、有些故障比如无法识别到芯片，也有可能是转接头的安装松动了，可以手指用力安装紧固。
- 3、当打开多个 VIVADO 工程的时候并且有一个 VIVADO 工程连接了 JTAG，其他工程会连接失败。可以选择关闭已经连接的工程，或者断开连接。
- 4、有时候 VIVADO 工程死机了，关闭工程后，没有释放占用 USB JTAG 的系统资源也会导致，JTAG 无法识别到芯片，可以重启电脑，或者到进程管理器去关闭死机的进程。

1、【第一季】CH04_FPGA 设计 Verilog 基础（一）

4.1 Verilog HDL 代码规范

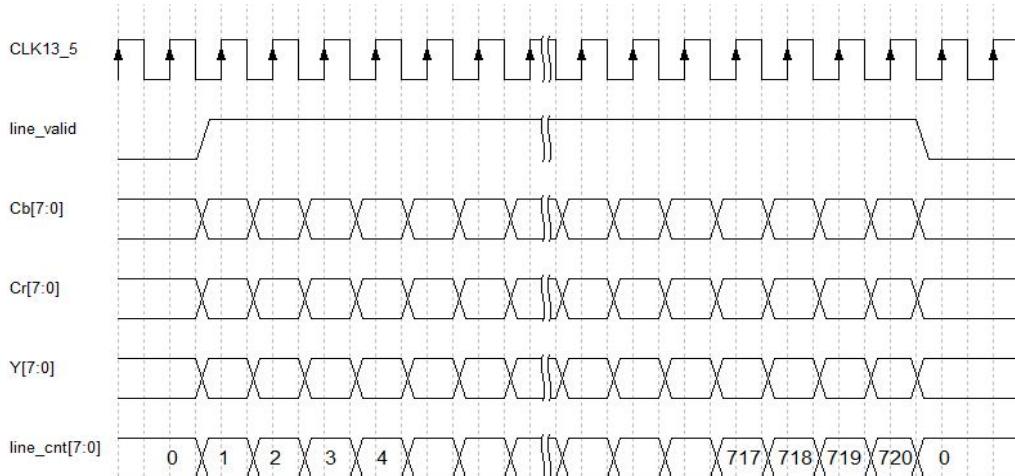
◆ 项目构架设计



项目的构架用于团队的沟通，以及项目设计的全局把控

◆ 接口时序设计规范

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



◆ 命名规则

1、顶层文件

对象+功能+top

比如：video_oneline_top

2、逻辑控制文件

介于顶层和驱动层文件之间

对象+ctr

比如： ddr_ctrl.v

3、驱动程序命名

对象+功能+dri

比如： lcd_dri.v、 uart_rxd_dri.v

4、参数文件命名

对象+para

比如： lcd_para.v

5、模块接口命名：文件名+u

比如 lcd_dir lcd_dir_u(.....)

6、模块接口命名：特征名+文件名+u

比如 mcb_read c3_mcb_read_u

7、程序注释说明

```
////////////////////////////////////////////////////////////////////////
```

```
// Company: milinker corporation
```

```
// Engineer:jinry tang
```

```
// WEB:www.milinker.com
```

```
// BBS:www.osrc.cn
```

```
// Create Date: 07:28:50 07/31/2015
```

```
// Design Name: FPGA STREAM
```

```
// Module Name: FPGA_USB
```

```
// Project Name: FPGA STREAM
```

```
// Target Devices: XC6SLX16-FTG256/XC6SLX25-FTG256 Mis603
```

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

8、端口注释

input Video_vs_i,//输入场同步入

9、信号命名

命名总体规则：对象+功能（+极性）+特性

10、时钟信号

对象+功能+特性

比如：phy_txclk_i、sys_50mhz_i

11、复位信号

对象+功能+极性+特性

比如：phy_RST_N_i、sys_RST_N_i

12、延迟信号

对象+功能+特性 1+特征 2

比如：fram_sync_i_r0、fram_sync_i_r1

13、特定功能计数器

对象+cnt

比如: line_cnt、div_cnt0、div_cnt1

功能+cnt

比如: wr_cnt、rd_cnt

对象+功能+cnt

比如: fifo_wr_cnt、mcb_wr_cnt、mem_wr_cnt

对象+对象+cnt

比如: video_line_cnt、video_fram_cnt

14、一般计数器

cnt+序号

用于不容易混淆的计数

比如: cnt0、cnt1、cnt2

15、时序同步信号

对象+功能+特性

比如: line_sync_i、fram_systc_i

16、使能信号

功能+en

比如: wr_en、rd_en

对象+功能+en

比如: fifo_wr_en、mcb_wr_en

4.2 技术背景

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

高速专用数字逻辑系统或专用数字信号处理器所构成，电路是相当复杂的。因此只有在高速大规模集成电路设计制造技术进步的基础上，才有可能实现真正有意义的实时数字

信号处理系统。对实时数字信号处理系统的要求不断提高，也推动了高速大规模集成电路设计制造技术的进步。现代专用集成电路的设计是借助于电子电路设计自动化(EDA)工具完成的。

学习和掌握硬件描述语言(HDL)是使用电子电路设计自动化(EDA)工具的基础。

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

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

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

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

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

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

■ 顺序控制的第一种办法——状态机设计

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

■ 顺序控制的第二种方法——FPGA 中运行 CPU

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

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

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

两种核可谓各有所长。

■ FPGA 还是 ASIC

根据具体看情况而定，从我们上面的一些介绍，笔者相信你已经有一定的判断能力了。笔者的建议是，低速场合，实时性要求的低的地方用 ASIC，有些功能用 ASIC 方便的用 ASIC，成本低的用 ASIC。排除那些可以不用 FPGA 地方，那么剩余的就要考虑是不是用 FPGA 来实现更加方便。一般来说，FPGA 程序开发相对来说要难度大一些，并且成本要高一些。

讲了这么多的背景知识，我们来看一小段代码：

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

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

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

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

4.3 Verilog 最最基础语法

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

C 语言和 Verilog 的关键词和结构对比：

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

C 语言和 Verilog 运算符对比：

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

真是太振奋人心了，一切都是这么熟悉。学好 FPGA 已经没有心理障碍了。

4.4 关键字

信号部分：

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

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

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

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

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

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

结构部分：

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

代表一个模块，我们的代码写在这个两个关键字中间

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

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

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

这些语句含义上都和高级语言一样：

```
if(...)begin  
.....  
End  
  
if(...)begin  
.....  
end  
else begin  
.....  
End  
  
if(...)begin  
.....  
end  
else if(...)begin  
.....  
end  
  
case(...)  
.....  
endcase
```

begin..... end 作用域范围，类似于 C 的大括号。用法举例：

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

```
.....  
end
```

符号部分：（我们这里 FALSE 为 0， TRUE 为 1）

“;” 分号用于每一句代码的结束，以表示结束，和 C 语言一样。

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

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

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

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

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

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

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

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

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

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

“>>”右移运算符，比如 A>>2 表示把 A 右移 2 位。

“<<”左移运算符，比如 A<<2 表示把 A 左移 2 位。

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

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

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

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

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

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

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

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

参数部分：

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

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

预处理命令

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

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

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

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

他们联合起来使用，确实能让你的程序多样化，就拿 VGA 程序说事吧。

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

假设有个 lcd_para.v 文件，内容如下：

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

```

`define V_FRONT 11'd10
`define V_SYNC 11'd2
`define V_BACK 11'd33
`define V_DISP 11'd480
`define V_TOTAL 11'd525
`endif
// 800 * 600
`ifdef VGA_800_600_72FPS_50MHz
`define H_FRONT 11'd56
`define H_SYNC 11'd120
`define H_BACK 11'd64
`define H_DISP 11'd800
`define H_TOTAL 11'd1040

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

```

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

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

```

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

```

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

其次`include "lcd_para.v" 这个路径也有一点讲究，xxx.v 作为引用 lcd_para.v 的文件它和 lcd_para.v 在同一文件夹下才能怎么写，就是相对路径一说了。也就是以 xxx.v 为当前路径去引索 lcd_para.v 文件的位子。所以如果他们不在一个文件夹那么请写出更详细（正确）的路径。顺便说一句，lcd_para.v 添不添加到工程是无所谓的，只要路径

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

4.5 Verilog 中数值表示的方式

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

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

十进制：8'd180;

16 进制：8'hAA;

4.6 阻塞赋值和非阻塞赋值详解

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

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

首先来说说非阻塞赋值，这个在时序逻辑中随处可见：

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

这段程序里，A 和 B 是同时被赋值的，具体是说在时钟的上升沿来的时刻，A 和 B 同时被置 1。调换 A 和 B 的上下顺序，将得到相同的结果。

接着看另外一段程序：

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

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

接着来看阻塞赋值，它少了一个非，表示会阻塞住，那么体会下这个阻塞：

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

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

回到正题，我们这么写是为了更好的理解阻塞赋值：当时钟上升沿来临的时刻，首先 A 会被置 1，然后 B 寄存器再置 1。区别就是 A 和 B 不再同时置 1。A 要比 B 提前零点

几纳秒。这样就出现了先后顺序。这个过程还是在一个时钟内完成的，但是数据到达 B 寄存器相比上面两段程序晚了那么零点几纳秒！

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

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

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

```
end  
endmodule
```

第一个时钟上升沿来临时，完成 $C \leq A + B$ ；

第二个时钟来临时完成 $result \leq (C \gg 1)$ ；

求出结果，这个过程耗费两个时钟。（不考虑复位消耗的时钟）

再来，用添加阻塞的方式实现：

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

仿真结果：



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

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

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

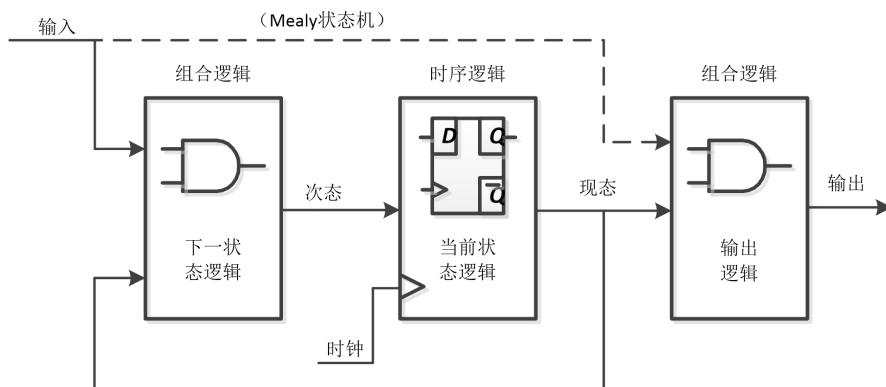
对于理论知识的学习，没必要一开始就研究得那么深刻，只是搞理论学习，对于学习 Verilog 语言，或者 FPGA 开发是不实际的，要联系理论和实践结合。多仿真，多验证，多问题，多学习，多改进。

S01_CH05_FPGA 设计 Verilog 基础（二）

5.1 状态机设计

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

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



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

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

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

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

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

下面根据状态机的三种方法，来比较各种方法的优劣。

5.2 一段式状态机

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

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

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

```
state<= S1;
end
S1 :
begin
    out_r<=1'b1;
    state<= S2;
end
S2 :
begin
    out_r<=1'b0;
    state<= S3;
end
S3 :
begin
    out_r<=1'b1;
end
endcase
end
assign out_o=out_r;
endmodule
```

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

5.3 两段式状态机

```
module detect_2(
    input clk_i,
    input rst_n_i,
    output out_o
);
    reg out_r;
```

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

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

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

```
        end

    S2 :
        begin
            out_r=1'b0;
            Next_state= S3;
        end

    S3 :
        begin
            out_r=1'b1;
            Next_state=Next_state;
        end

    endcase
end

assign out_o=out_r;

endmodule
```

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

5.4 三段式状态机

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

```
parameter [1:0] S2=2'b10;
parameter [1:0] S3=2'b11;

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

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

//输出逻辑：让输出 out， 经过寄存器 out_r 锁存后输出，消除毛刺
```

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

    assign out_o=out_r;
```

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

S01_CH06_FPGA 设计 Verilog 基础（三）

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

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

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

6.1 完成的 Test bench 文件结构

通常，一个完整的测试文件其结构为

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

6.2 时钟激励设计

下面列举出一些常用的封装子程序，这些是常用的写法，在很多应用中都能用到。

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

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

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

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

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

6.3 复位信号设计

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

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

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

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

6.4 特殊信号设计

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

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

```
wire bir_port;
reg bir_port_reg;
reg bi_port_oe;
assign bi_port=bi_port_oe ? bir_port_reg : 1'bz;
/*-----
```

双向信号描述二：强制 force

```
-----*/
```

```
//当双向端口作为输出口时，不需要对其进行初始化，而只需开通三态门
```

```
//当双向端口作为输入时，只需要对其进行初始化并关闭三态门，初始化赋值需
```

```
//使用 wire 型数据，通过 force 命令来对双向端口进行输入赋值
```

```
//assign dinout=(!en) din :16'hz; 完成双向赋值
```

```
initial
```

```
begin
    force dinout=20;
    #200
    force dinout=dinout-1;
end
```

```
/*-----
```

特殊激励信号产生描述三：输入信号产生,一次 SRAM 写信号产生

```
-----*/
```

```
initial
```

```
begin
    cs_n=1;           //片选无效
    wr_n=1;           //写使能无效
    rd_n=1;           //读使能无效
    addr=8'hxx;       //地址无效
    data=8'hzz;       //数据无效
    #100;
    cs_n=0;           //片选有效
    wr_n=0;           //写使能有效
    addr=8'hF1;       //写入地址
end
```

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

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

6.5 仿真控制语句及系统任务描述

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

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

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

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

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

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

6.6 加法器的仿真测试文件编写

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

这里提出以下几点建议供大家参考：

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

下面具体看一段程序：

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

通常，我们模块的调用写法如下：

被调用的模块名字- 自定义的名字- 括号内信号

这里比如括号内的信号，.ina(ina1)

这种写法最常用，信号的顺序可以调换

另外还有一种写法没可以直接这样写

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

这种写法必须确保信号的顺序一致，这种写法几乎没有采用

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

//adder 子模块

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

仿真文件：

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

```

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

```

```

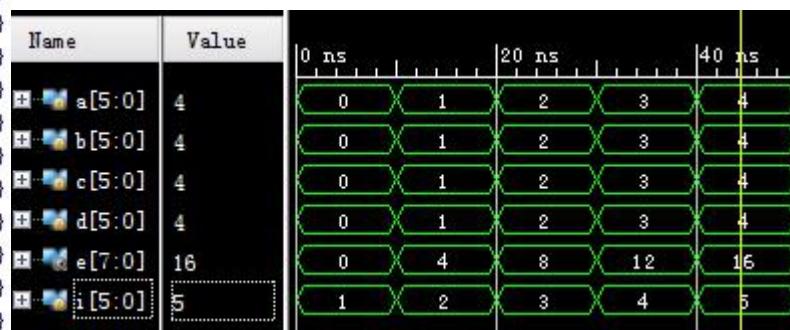
0 0 + 0 + 0 + 0 ={ 0}
10 1 + 1 + 1 + 1 ={ 4}
20 2 + 2 + 2 + 2 ={ 8}
30 3 + 3 + 3 + 3 ={ 12}
40 4 + 4 + 4 + 4 ={ 16}
50 5 + 5 + 5 + 5 ={ 20}
60 6 + 6 + 6 + 6 ={ 24}
70 7 + 7 + 7 + 7 ={ 28}
80 8 + 8 + 8 + 8 ={ 32}
90 9 + 9 + 9 + 9 ={ 36}
100 10 + 10 + 10 + 10 ={ 40}
110 11 + 11 + 11 + 11 ={ 44}
120 12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

```

```

210 21 + 21 + 21 + 21 ={ 84}
220 22 + 22 + 22 + 22 ={ 88}
230 23 + 23 + 23 + 23 ={ 92}
240 24 + 24 + 24 + 24 ={ 96}
250 25 + 25 + 25 + 25 ={100}
260 26 + 26 + 26 + 26 ={104}
270 27 + 27 + 27 + 27 ={108}
280 28 + 28 + 28 + 28 ={112}
290 29 + 29 + 29 + 29 ={116}
300 30 + 30 + 30 + 30 ={120}

```



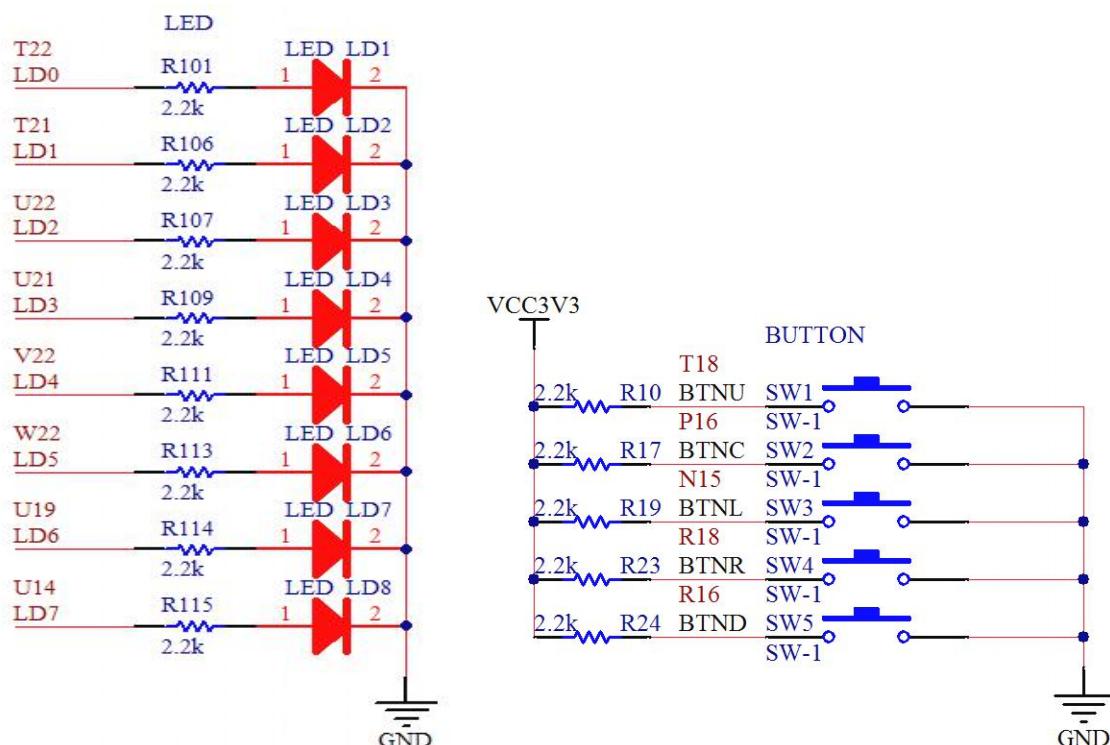
S01_CH07_FPGA_RunLED 创建 VIVADO 工程实验

7.1 硬件图片

先来熟悉一下开发板的硬件：LED 部分及按钮部分



7.2 硬件原理图



PIN 脚定义(讲解以 MIZ702 讲解,MIZ701N 只有 4 个 LED 2 个按钮):

GCLK:Y9(PL 输入时钟) LD0:T22 LD1:T21 LD2:U22 LD3:U21 LD4:V22 LD5:W22 LD6:U19	BTNU:T18 BTNC:P16 BTNL:N15 BTNR:R18 BTND:R16
---	--

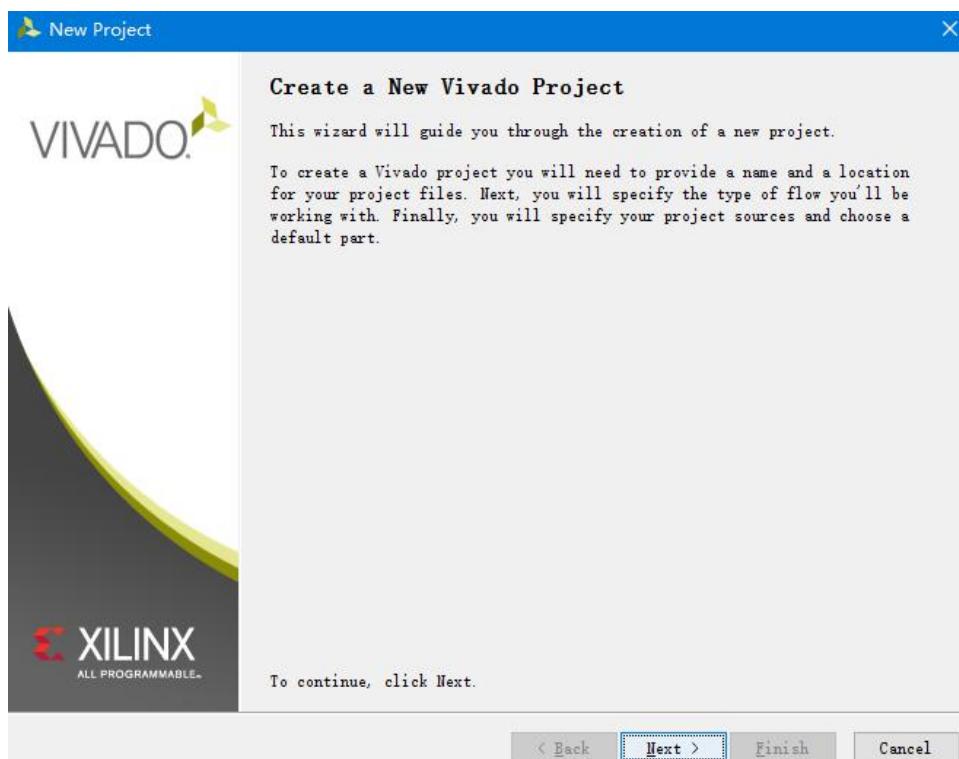
LD7:U14	
---------	--

7.3 新建 VIVADO 工程

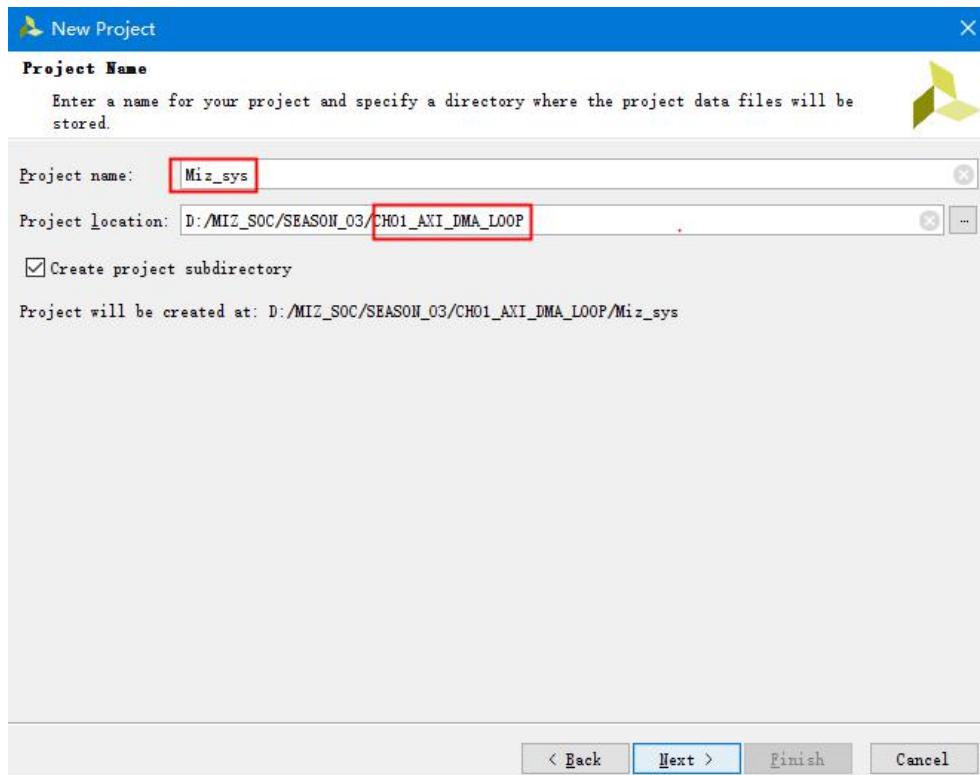
Step1:启动 VIVADO，单击 Create New Project



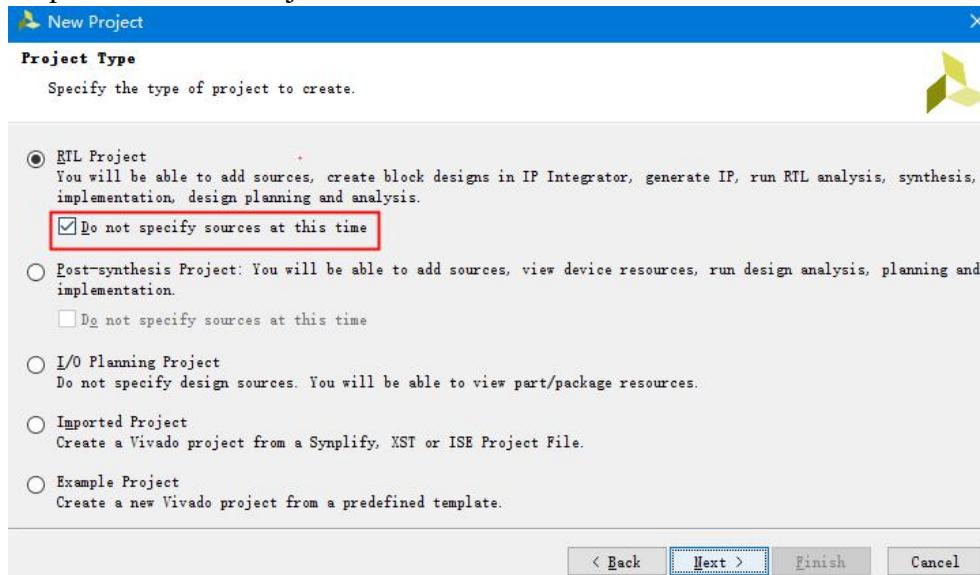
Step2:单击 NEXT



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



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

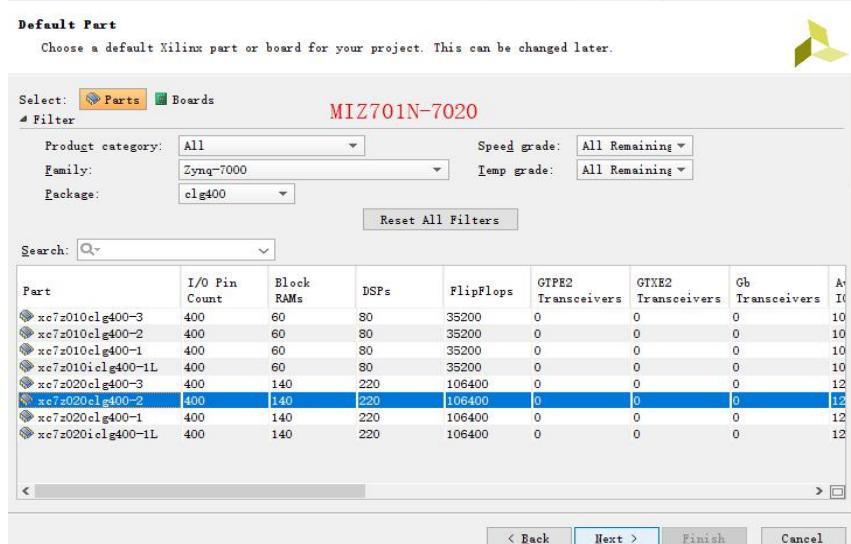
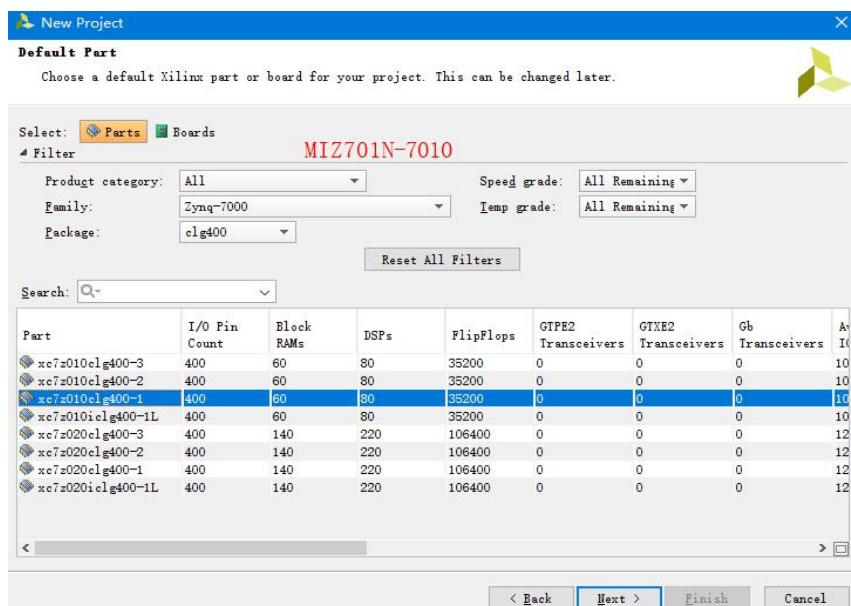
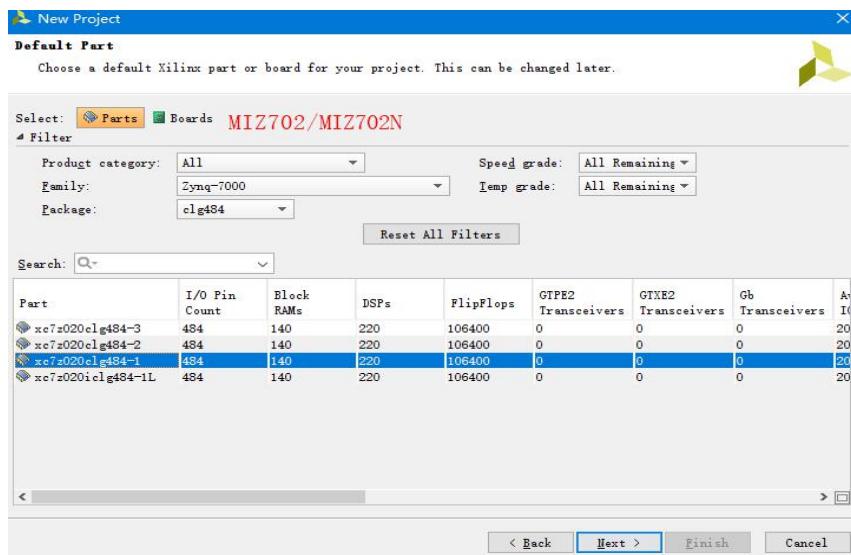


Step5:选择芯片的型号和封装速度等级。

MIZ702/MIZ702N 选择 Zynq-7000-xc7z020clg484-1

MIZ701N-7010 选择 Zynq-7000-xc7z010clg400-1

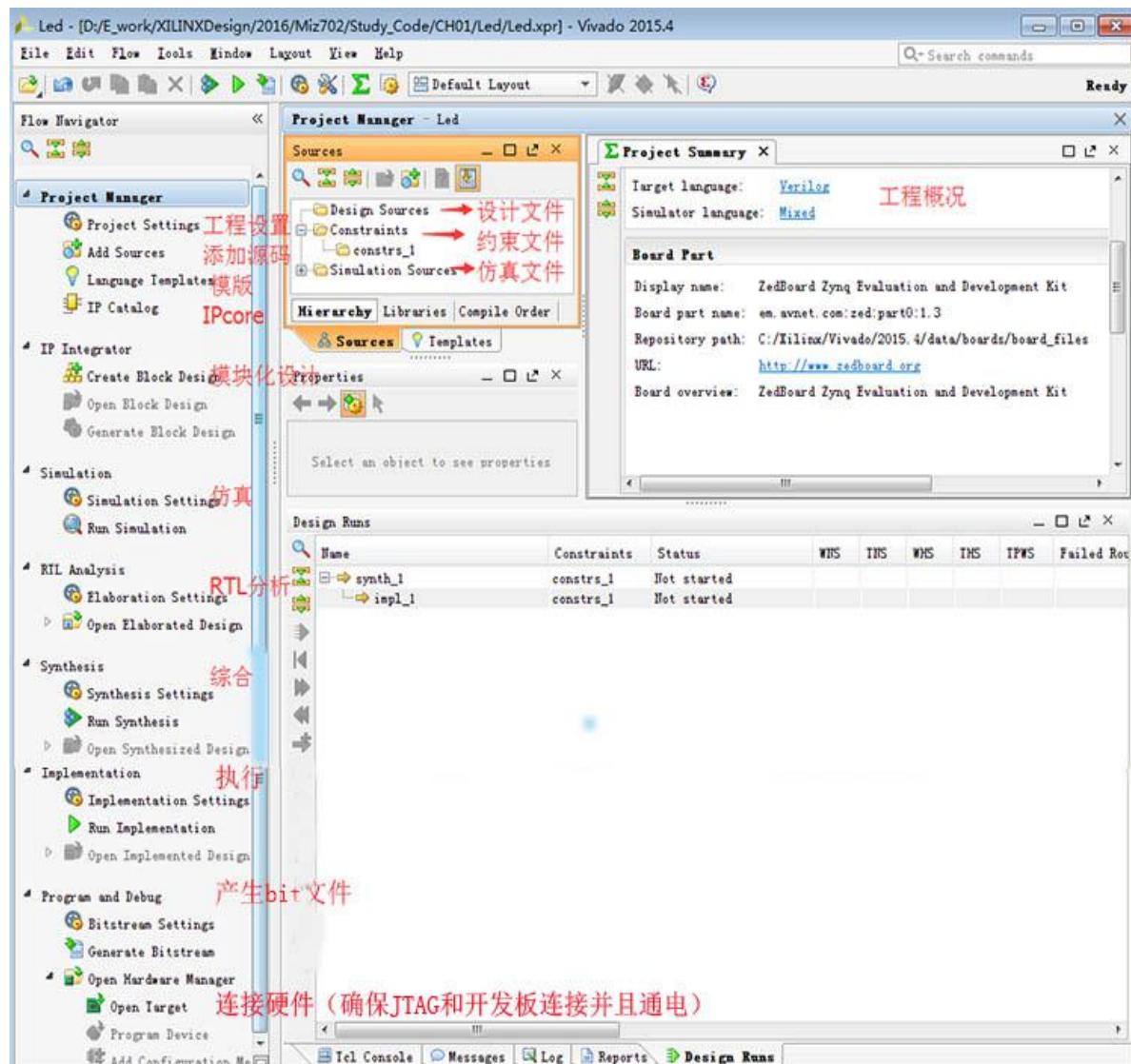
MIZ701N-7020 选择 Zynq-7000-xc7z020clg400-2



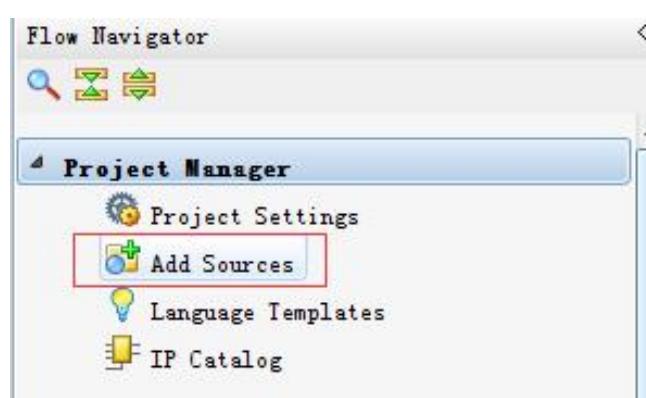
Step:6 单机 Finish 完成工程创建。

7.4 创建工程文件

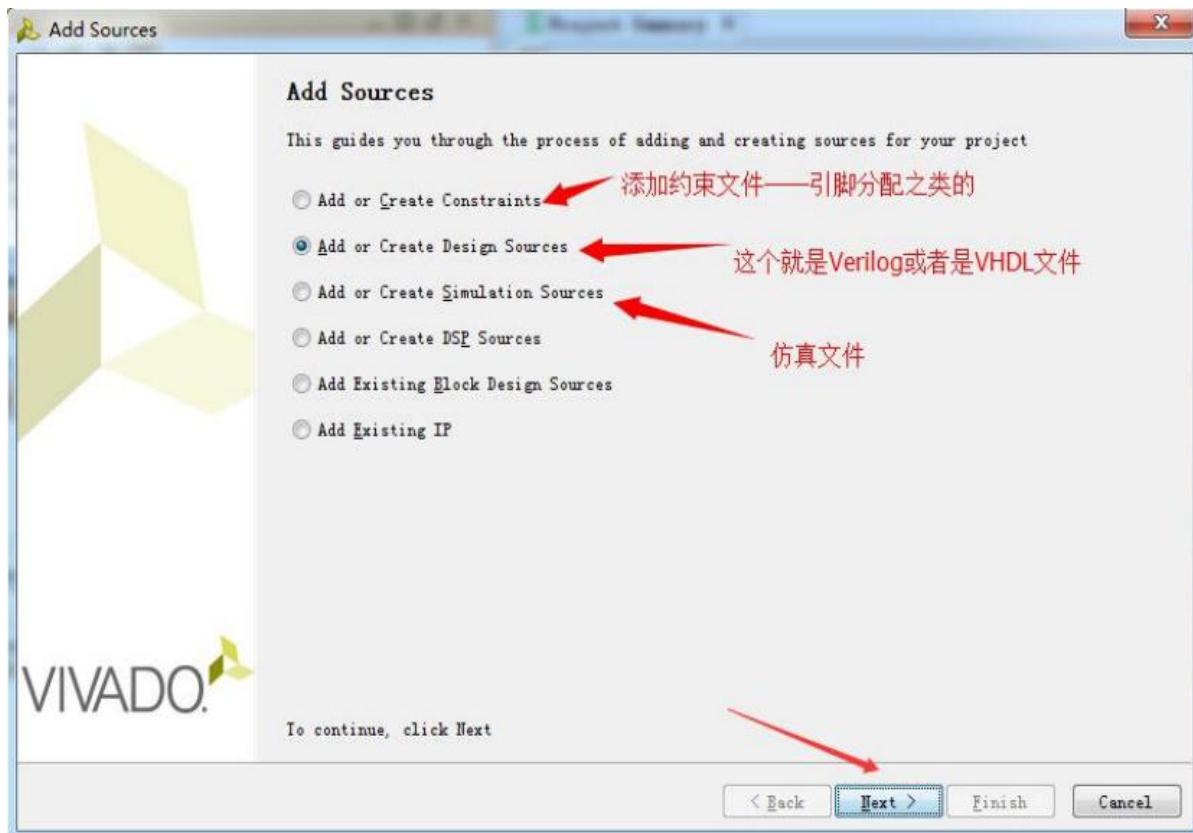
Step1: 打开 VIVADO 软件



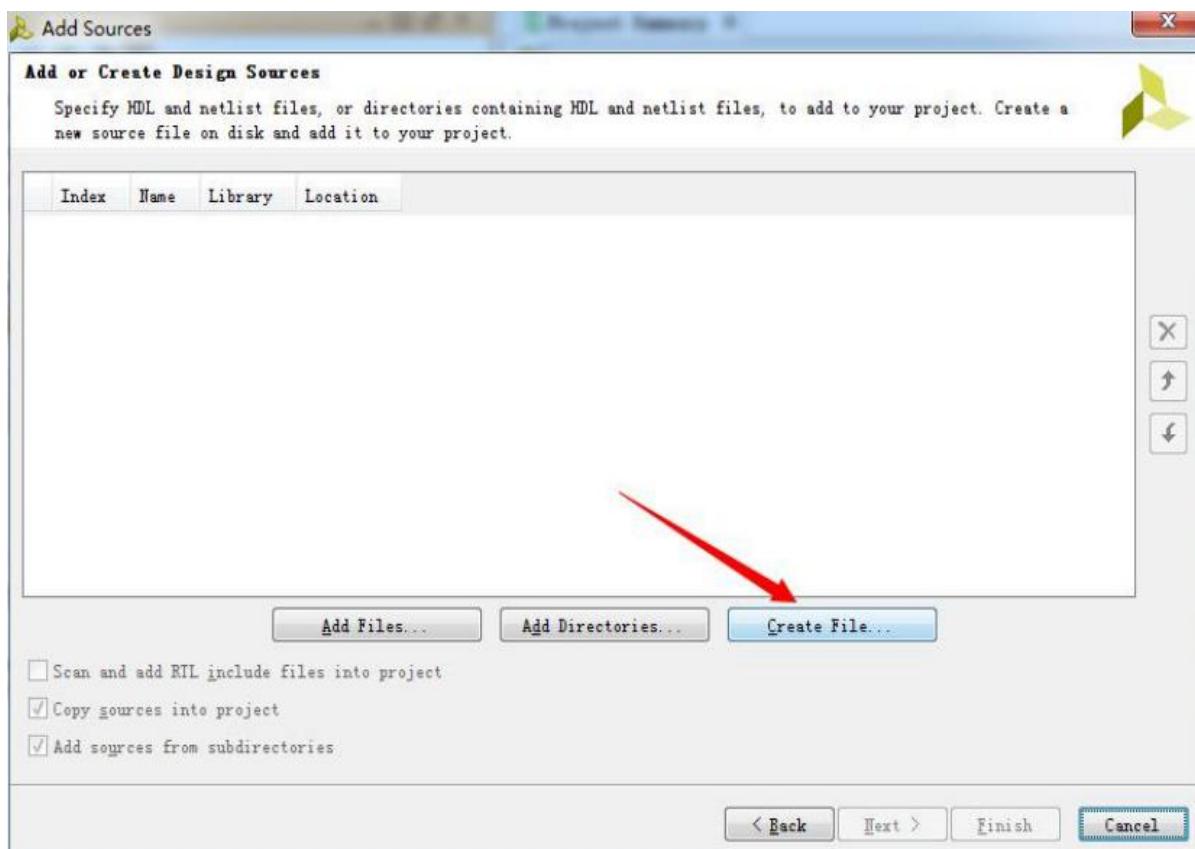
Step2: 单击 Add Sources



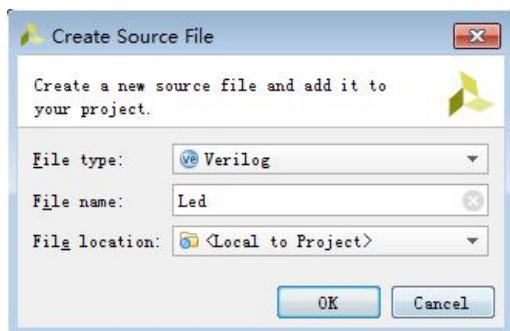
Step4: 选择单击 Add or Create Design Sources 然后单击 NEXT



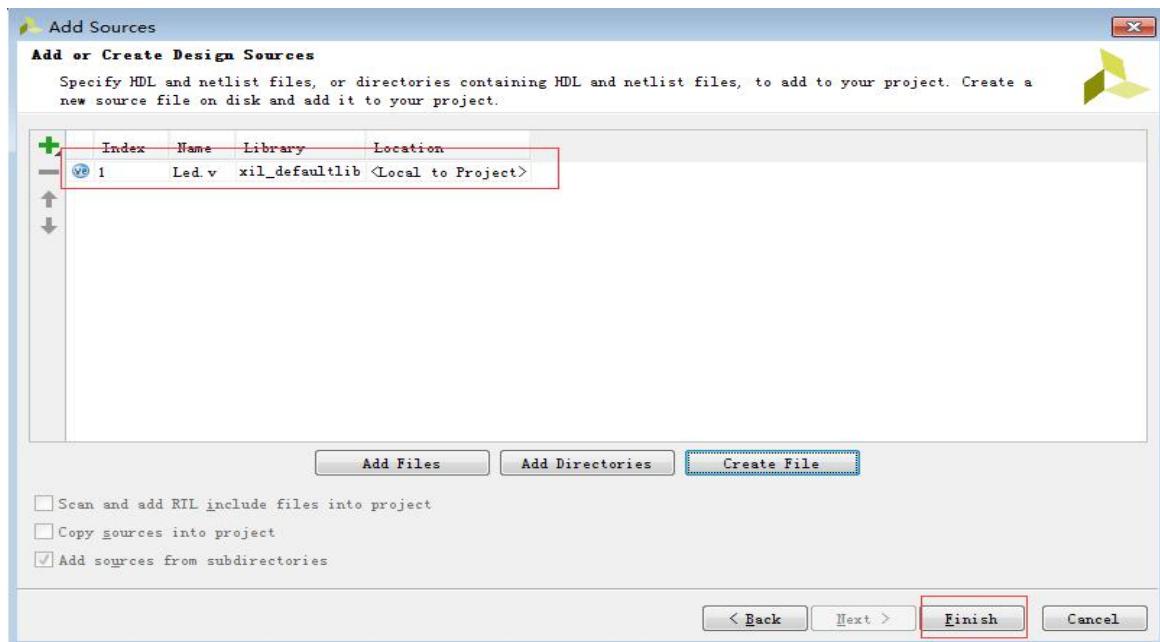
Step5: 单击 Create File 来创建文件



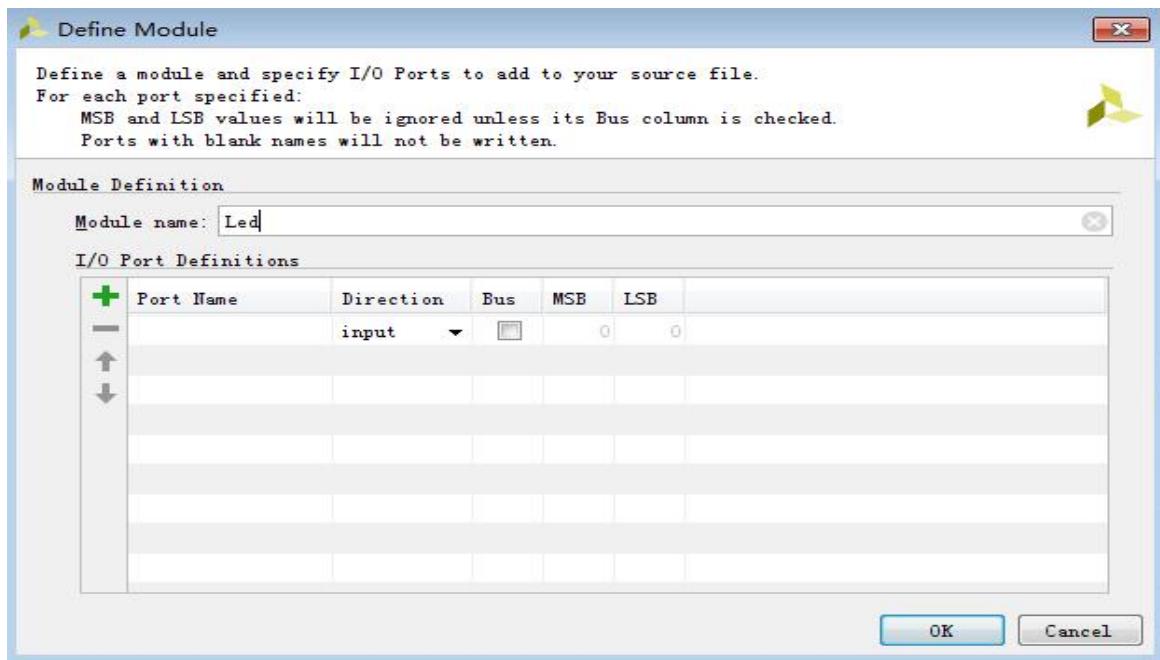
Step6: 创建一个 Led 的文件，并且文件类型选择 Verilog



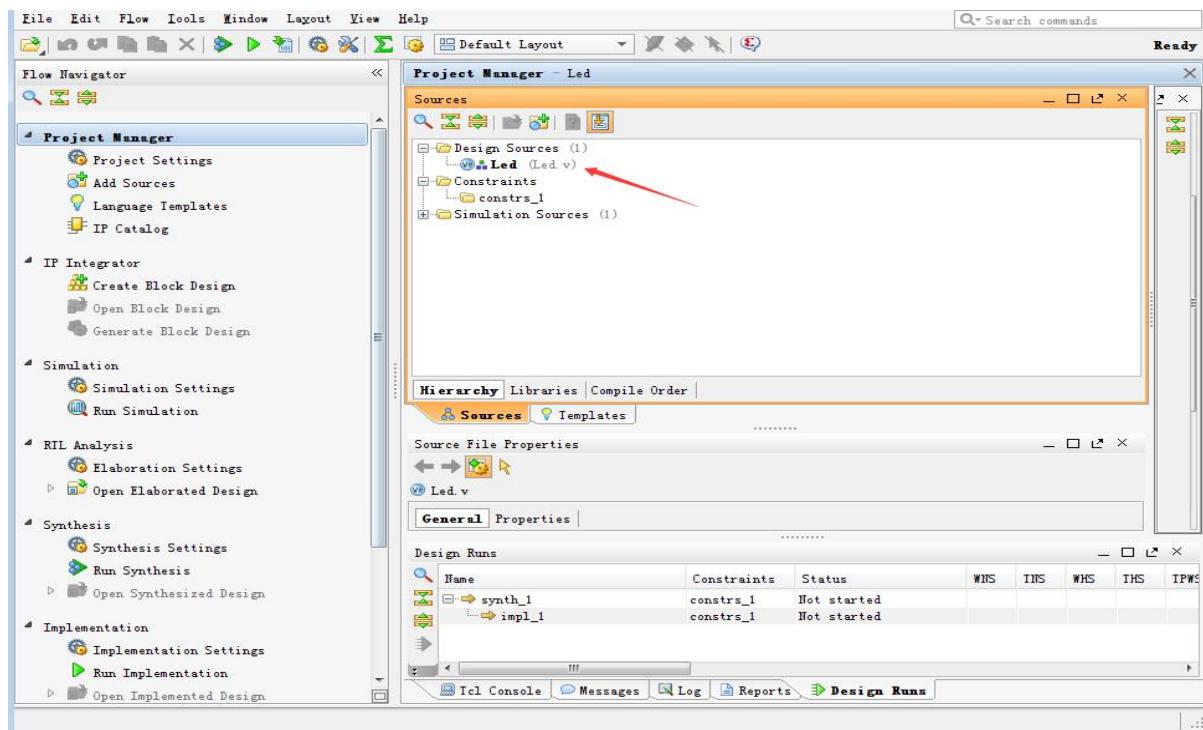
Step7: 添加完成后如下图所示之后单击 finish 完成文件的创建



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



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



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

7.5 Verilog FPGA 流水灯实验

Step1: 双击 Led.v 打开流水程序源码如下

```
'timescale 1ns / 1ps

///////////////////////////////
// Company:
// Engineer:
// Create Date: 2016/03/22 15:05:39
// Design Name:
// Module Name: Led
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
// Dependencies:
// Revision:
```

```
// Revision 0.01 - File Created  
// Additional Comments:  
/////////  
module Led(  
);  
endmodule
```

可以看出这是一个空的工程，我们现在要添加代码同时也要添加工程信息。

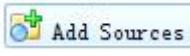
Step2: 编写程序并且添加工程信息

```
// Target Devices: XC7Z020-FGG484  
// Tool versions: VIVADO2015.4  
// Description: water led  
// Revision: V1.1  
// Additional Comments:  
//1) _i PIN input  
//2) _o PIN output  
//3) _n PIN active low  
//4) _dg debug signal  
//5) _r reg delay  
//6) _s state machine  
/////////  
module Led(  
    input CLK_i,  
    input RSTn_i,  
    output reg [7:0]LED_o  
);  
reg [31:0]C0;  
  
always @ (posedge CLK_i)  
if (!RSTn_i)  
begin  
    LED_o <= 8'b0000_0001;  
    C0 <= 32'h0;  
end  
else  
begin  
    if (C0 == 32'd50_000_000)  
    begin  
        C0 <= 32'h0;  
        if (LED_o == 8'b1000_0000)  
            LED_o <= 8'b0000_0001;  
        else LED_o <= LED_o << 1;  
    end
```

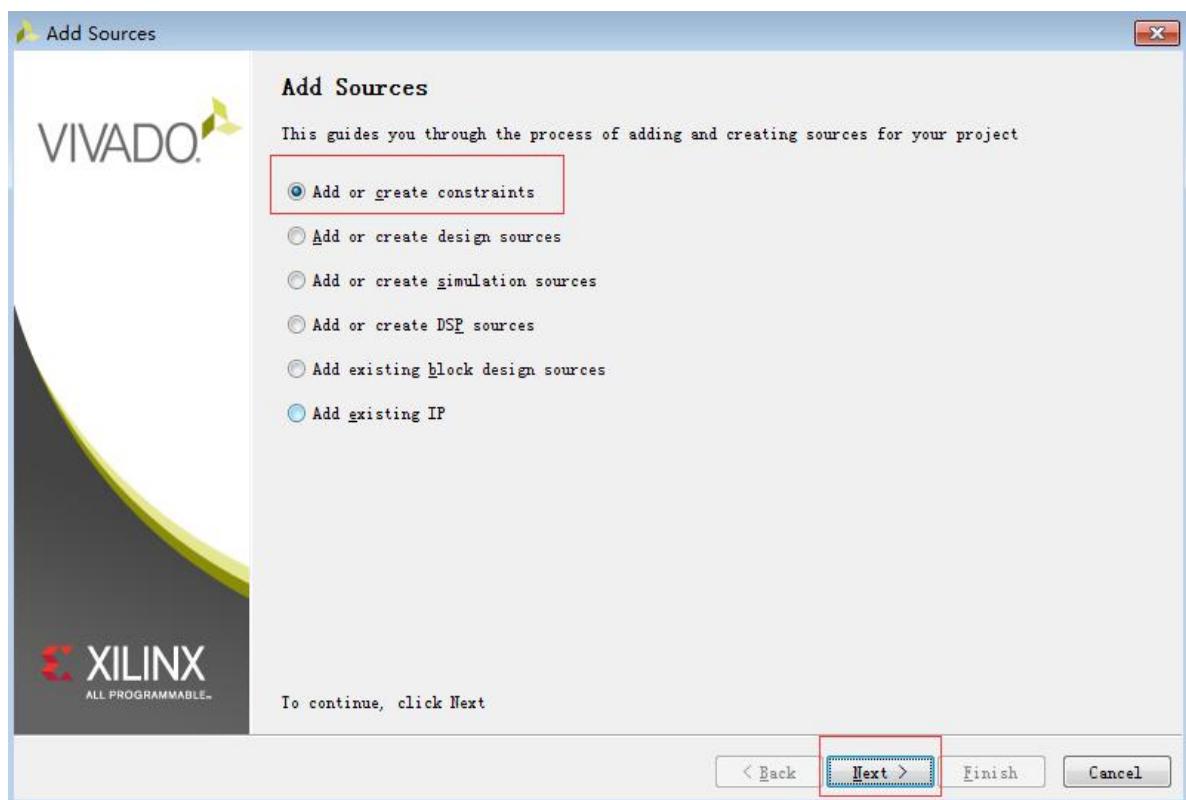
```
end
else begin C0 <= C0 + 1'b1; LED_o <= LED_o; end
end
endmodule
```

这样我们就编写好了代码下面还要添加管脚约束文件。

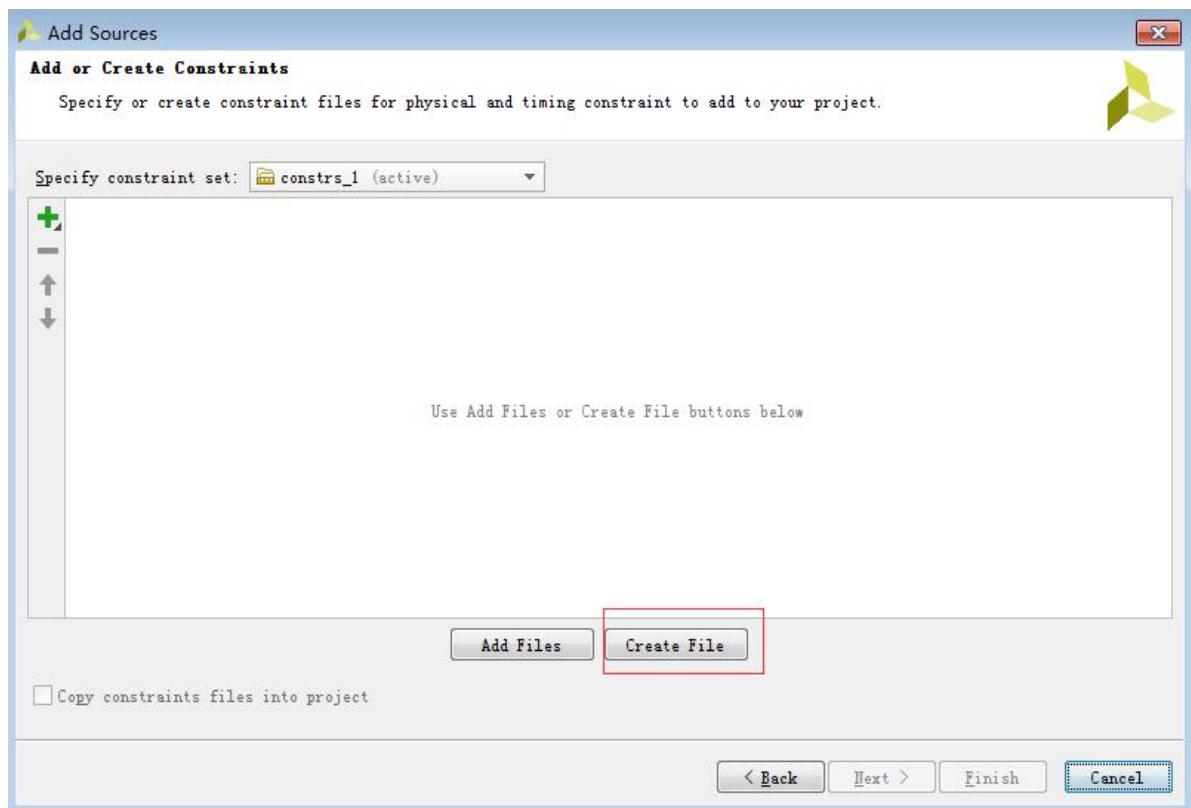
7.6 添加管脚约束文件

Step1:单击  (和添加.v文件一样)

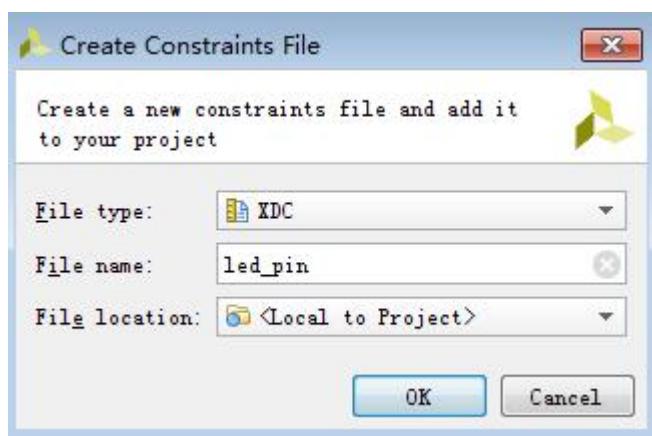
Step2:选择 Add or create constraints 然后单击 NEXT



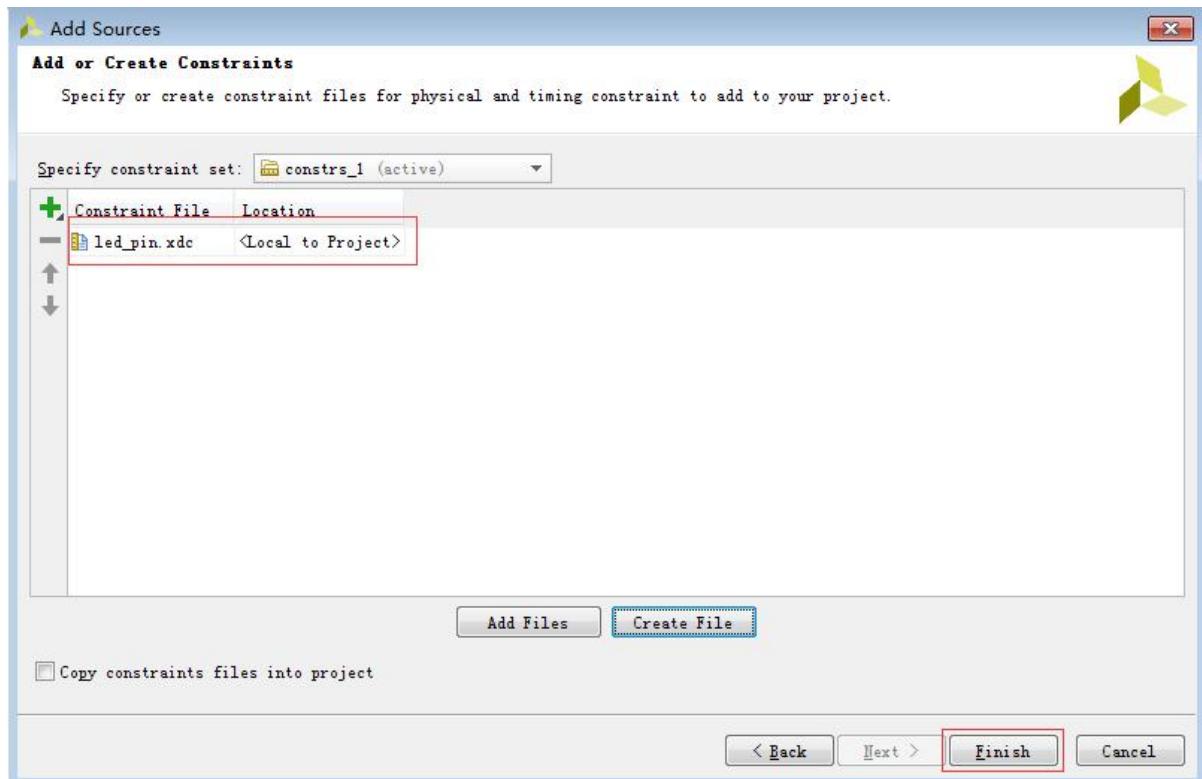
Step3:单击 Create File



Step4:命名为 led_pin 后单击 OK



Step5:可以看到产生了名为 led_pin.adc 的文件然后单击 Finish



Step6: 打开 led_pin.adc 文件添加如下约束

```

create_clock -name clk100MHZ -period 10.0 [get_ports {CLK_i}]

set_property PACKAGE_PIN Y9 [get_ports {CLK_i}]
set_property IOSTANDARD LVCMOS33 [get_ports {CLK_i}]

set_property PACKAGE_PIN N15 [get_ports {RSTn_i}]
set_property IOSTANDARD LVCMOS18 [get_ports {RSTn_i}]

set_property PACKAGE_PIN T22 [get_ports {LED_o[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_o[0]}]

set_property PACKAGE_PIN T21 [get_ports {LED_o[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_o[1]}]

set_property PACKAGE_PIN U22 [get_ports {LED_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_o[2]}]

set_property PACKAGE_PIN U21 [get_ports {LED_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_o[3]}]

set_property PACKAGE_PIN V22 [get_ports {LED_o[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_o[4]}]

```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED_o[5]}]
set_property PACKAGE_PIN W22 [get_ports {LED_o[5]}]

set_property PACKAGE_PIN U19 [get_ports {LED_o[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_o[6]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED_o[7]}]
set_property PACKAGE_PIN U14 [get_ports {LED_o[7]}]
```

7.7 编译并且产生 bit 文件

Step1:单击综合

Step2:单击执行

Step3:单击产生 bit



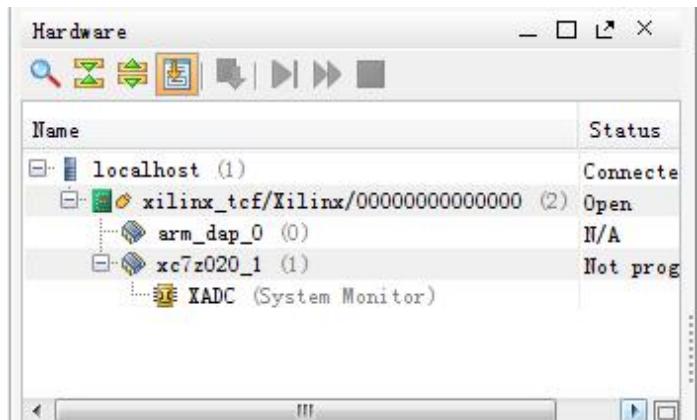
7.8 下载程序

Step1:给开发板通电，并且连接下载器

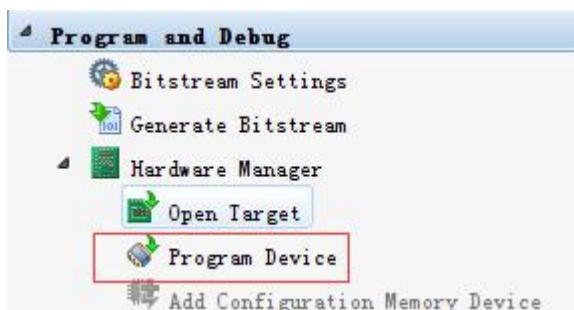
Step2:单击 OpenTarget 然后单击 Auto Connect



Step3:连接成功后



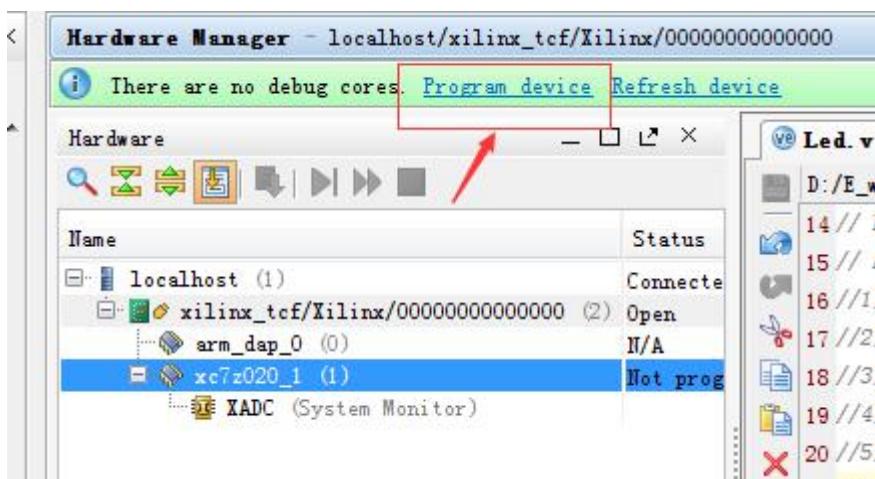
Step4:单击 Program Device



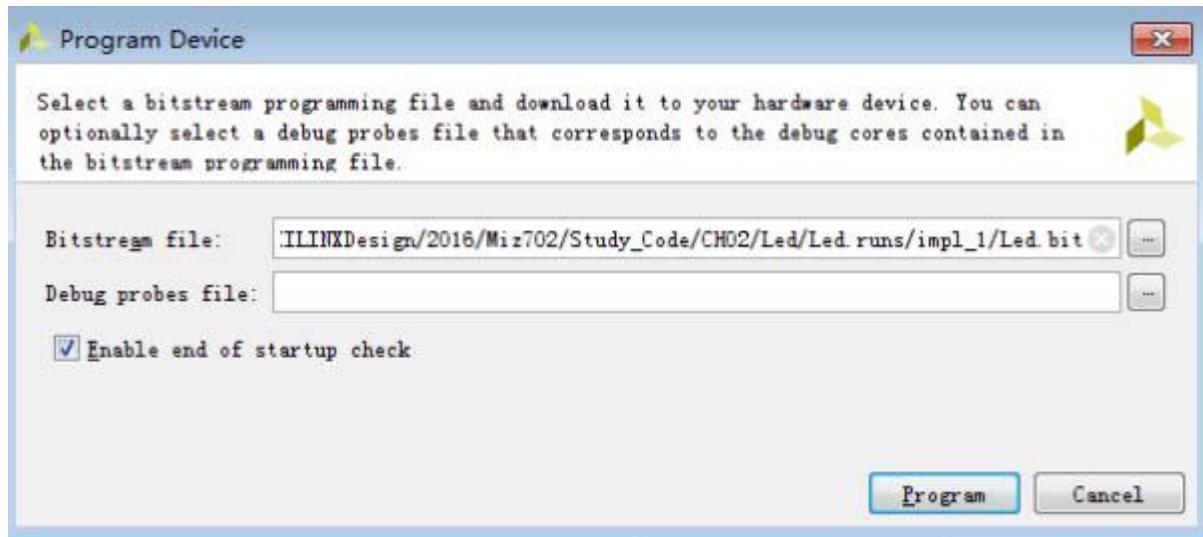
Step5:单击 Program Device 然后选择 XC7Z020_1



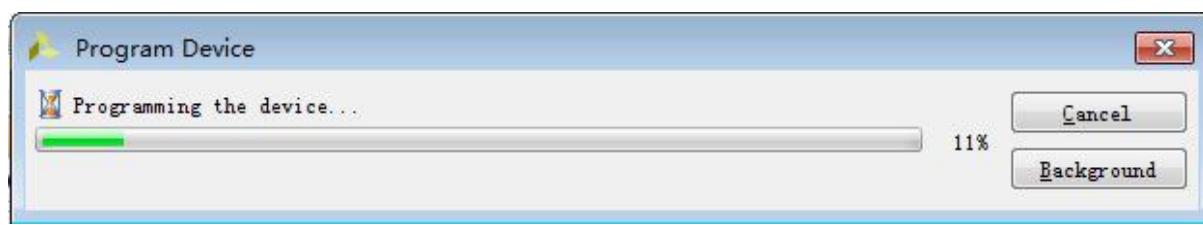
Step6:或者也可以从顶部单击 Program device



Step7: 弹出的对话框中有我们要下载的 Bit 文件

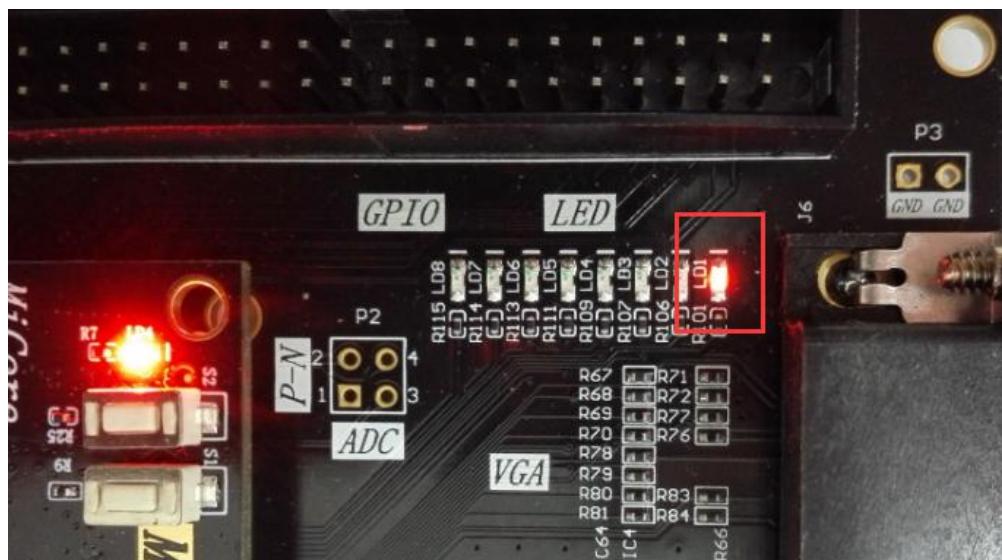


Step7: 下载过程



7.9 实验结果

下载过程下载完成后 LED 流水灯就运行起来了。



7.10 本章小结

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

S01_CH08_FPGA_Button 按钮去抖动实验

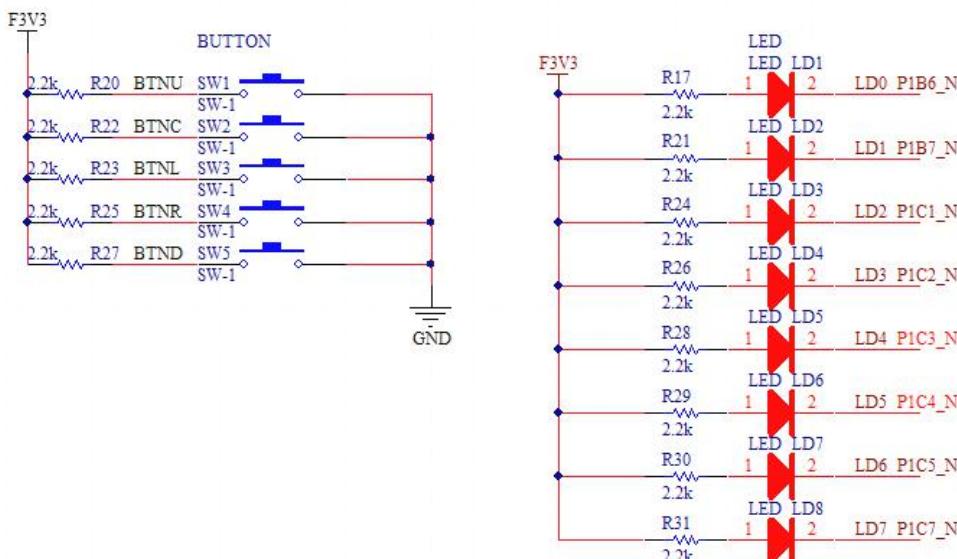
按键的消抖，是指按键在闭合或松开的瞬间伴随着一连串的抖动，这样的抖动将直接影响设计系统的稳定性，降低响应灵敏度。因此，必须对抖动进行处理，即消除抖动的影响。实际工程中，有很多消抖方案，如 RS 触发器消抖，电容充放电消抖，软件消抖。本章利用 FPGA 内部来设计消抖，即采取软件消抖。

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

8.1 硬件介绍

Mis603 底板中配套 5 个独立按键与 FPGA 相连，具体请参见 Misfun 原理图。由于各个按键独立，消抖过程是一样的，故本节就用板子上的一个按键 SW4 来模拟实际环境。按键每按一次，对应的 LED 灯反转一次。即检测按键是否有闭合和断开的过程，如果有，第一次则 LED 灯点亮，第二次，则 LED 灯熄灭。

原理图如下图所示



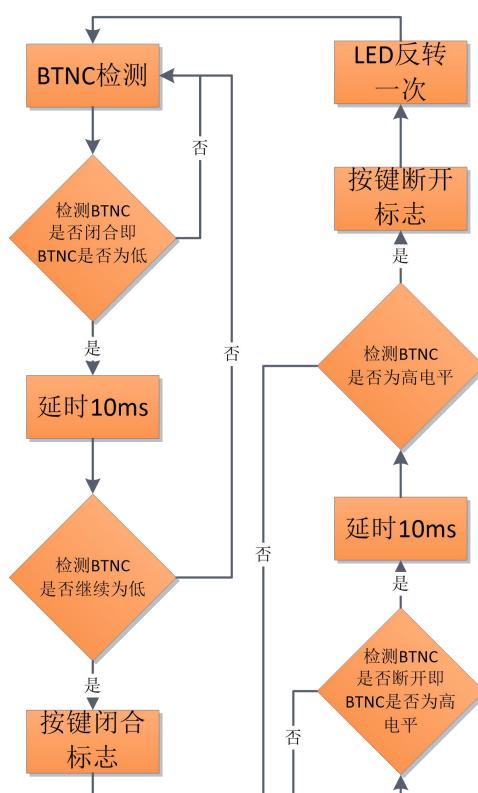
Mis603 开发板底板 5 个按钮和 8 个 LED 与 FPGA 对应的 PIN 定义

BTNU:G16	led[0]:A12
BTNC:G14	led[1]:E11
BTNL:F16	led[2]:D9
BTNR:H15	led[3]:A11
BTND:H16	led[4]:A10
	led[5]:E8
	led[6]:C8

led[7]:A8

8.2 时序设计

由于按键固有的特性，在每次闭合和断开时，经过抖动-稳定-抖动-稳定的过程。因此，检测按键是否有按下的过程，则要进行两次消抖处理。通过检测按键输入的值，当检测到 BTNC 为低电平时，启动计数器，做 10ms 延时，再检测一次，若 BTNC 依然为低，则说明，BTNC 被按下，设置按键按下标志位。再次检测 BTNC，若 BTNC 为高电平，做 10ms 延时，第二次检测，若依然为高电平，则说明 BTNC 已断开，设置 BTNC 断开标志位，通过两个标志位，可以判断，BTNC 已经完成了一次闭合到断开的过程，则 led 灯反转一次。消抖基本流程如下图所示。



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

8.3 程序源码

```

`timescale 1ns / 1ps
//-----
/*
* 文件名字: Key_Jitter.v

```

```
* 程序描述:  
* 作    者:  
* 修改日期:  
* 版本号:  
* 版权所有: 南京米联电子科技有限公司  
*/  
//-----  
module Key_Jitter(  
    input clk_i,  
    input rst_n_i,  
    input key_i,  
    output [3:0] led_o,  
    output [18:0] div_cnt_tb,  
    output [2:0] key_state_tb  
);  
  
localparam DELAY_Param=19'd499_999;//for project  
//localparam DELAY_Param=19'd1500;      //for simulation  
//localparam DELAY_Param=19'd10;       //No filter jitter  
  
reg [3:0] led_o_r;  
(*KEEP = "TRUE" *)reg [18:0] div_cnt;//10ms 去抖时间计数器  
always@(posedge clk_i or negedge rst_n_i)  
begin  
    if(!rst_n_i)  
        div_cnt<=19'd0;  
    else if(div_cnt<DELAY_Param)  
        div_cnt<=div_cnt+1'b1;  
    else  
        div_cnt<=0;  
end  
  
wire delay_10ms=(div_cnt==DELAY_Param) ? 1'b1:1'b0;
```

```
//按键检测状态机

localparam DETECTER1=3'b000;
localparam DETECTER2=3'b001;
localparam DETECTER3=3'b010;
localparam DETECTER4=3'b011;
localparam LED_DIS =3'b100;

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

                DETECTER2 :
                    begin
                        if(key_i!=1'b1)
                            begin
                                low_flag<=1'b1;
```

```
key_state<=DETECTER3;
end
else
begin
    key_state<=DETECTER1;
    low_flag<=low_flag;
end
end

DETECTER3 :
begin
if(key_i==1'b1)
    key_state<=DETECTER4;
else
    key_state<=DETECTER3;
end

DETECTER4 :
begin
if(key_i==1'b1)
begin
    high_flag<=1'b1;
    key_state<=LED_DIS;
end
else
begin
    high_flag<=high_flag;
    key_state<=DETECTER3;
end
end

LED_DIS :
begin
if(high_flag & low_flag)
begin
    key_state<=DETECTER1;
```

```
    led_o_r<=~led_o_r;
    high_flag<=1'b0;
    low_flag<=1'b0;
end
else
begin
    led_o_r<=led_o_r;
    key_state<=key_state;
    high_flag<=high_flag;
    low_flag<=low_flag;
end
default:
begin
    key_state<=DETECTER1;
    led_o_r<=0;
    high_flag<=0;
    low_flag<=0;
end
endcase
end
else
begin
    key_state<=key_state;
    led_o_r<=led_o_r;
    high_flag<=high_flag;
    low_flag<=low_flag;
end
end

assign led_o=led_o_r;
assign div_cnt_tb=div_cnt;
assign key_state_tb=key_state;
endmodule
```

8.4 程序分析

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

8.5 综合布线前仿真时序

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



8.6 Chipscope 在线逻辑分析仪仿真

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

8.7 输出结果

按键 SW4 每按一次，LED 灯很好地熄灭和点亮。LED 灯响应无差错。为清晰的表示消抖的效果，可将延时参数设置很小，可以发现，按键有时候明明已经按下去了，LED 却无响应。

8.8 小结

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

量逐渐增加的基础上，将分模块分文件来实现整个系统的功能。

串行通信接口作为一种最普遍的通信方式，广泛为工业所应用。其优点在于通信线路简单，只需要 2 根线即可实现数据的接收与发送，且通信协议简单、可靠。本节将重点利用 FPGA 来实现串行通信，通过串口调试助手与 Mis603 底板实现通信。

S01_CH09_FPGA 多路分配器设计

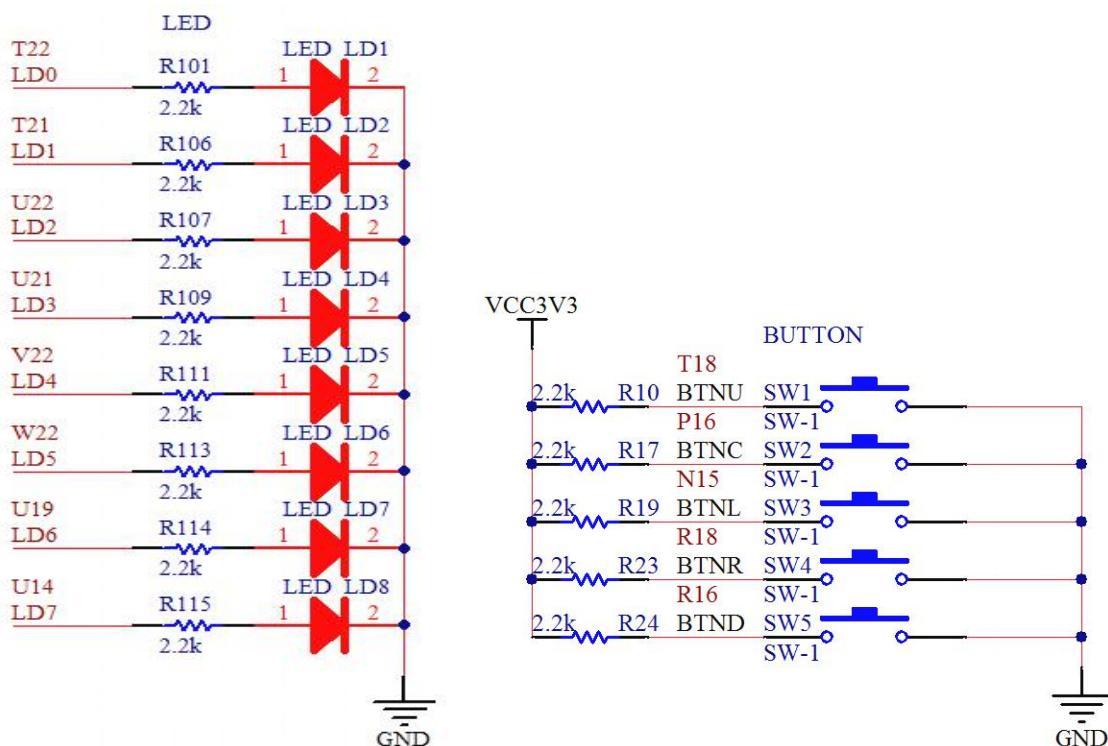
在第二章的学习中，笔者带大家通过一个入门必学的流水灯实验实现，快速掌握了VIVADO 基于 FPGA 开发板的基本流程。考虑到很多初学者并没有掌握好 Vivado 下 FPGA 的开发流程，本章开始笔者讲更加详细地介绍基于 VIVADO FPGA 开发的流程规范，让读者掌全面掌握 FPGA 开发流程包括了如何仿真、综合、执行、下载到开发板测试。

9.1 硬件图片

本章使用到的硬件和前一章一样：LED 部分及按钮部分



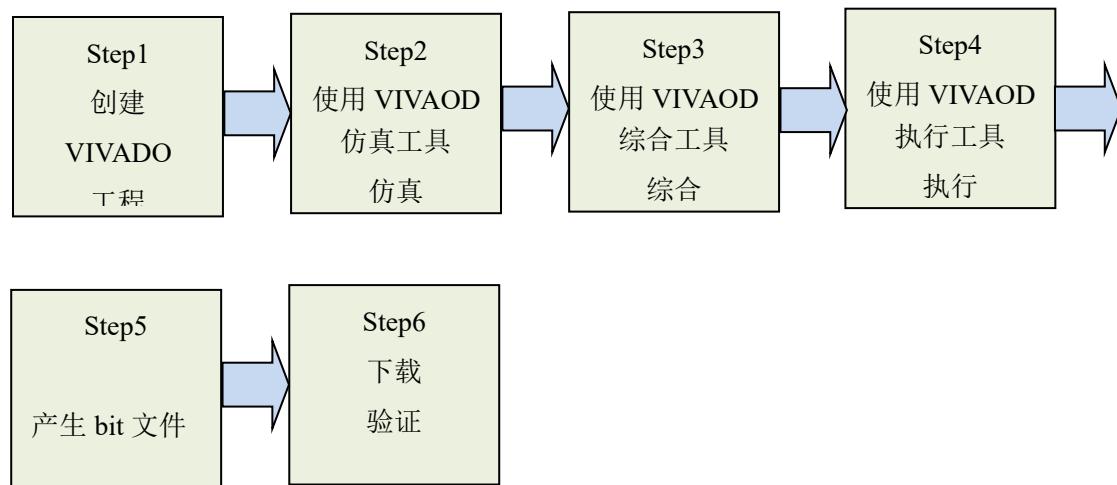
9.2 硬件原理图



PIN 脚定义：

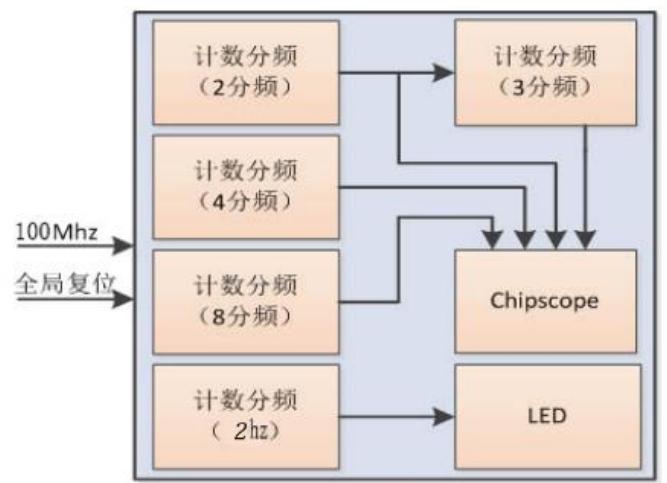
GCLK:Y9(PL 输入时钟) LD0:T22	BTNU:T18
-----------------------------	----------

9.3 介于 VIVADO 的 FPGA 设计流程

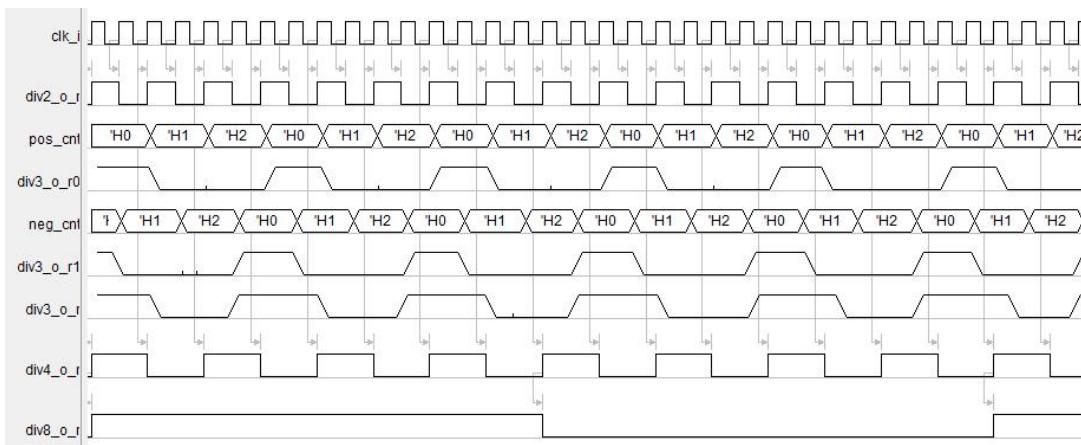


9.4 多路分配器设计思想

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



9.5 时序设计



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

9.6 程序源码

```

`timescale 1ns / 1ps
//-----
// Target Devices: XC7Z020-FGG484
// Tool versions: VIVADO2015.4
// Description: Divider_Multiple
// Revision: V1.1
// Additional Comments:

```

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

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

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

```
div_cnt1<=2'b00;
else
    div_cnt1<=div_cnt1+1'b1;
end

reg div4_o_r;
reg div8_o_r;

always@(posedge clk_i or negedge rst_n_i)
begin
    if(!rst_n_i)
        div4_o_r<=1'b0;
    else if(div_cnt1==2'b00 || div_cnt1==2'b10)
        div4_o_r<=~div4_o_r;
    else
        div4_o_r<=div4_o_r;
end

always@(posedge clk_i or negedge rst_n_i)
begin
    if(!rst_n_i)
        div8_o_r<=1'b0;
    else if((~div_cnt1[0]) && (~div_cnt1[1]))
        div8_o_r<=~div8_o_r;
    else
        div8_o_r<=div8_o_r;
end

reg [1:0] pos_cnt;
reg [1:0] neg_cnt;
always@(posedge div2_o_r or negedge rst_n_i)
```

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

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

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

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

reg div2hz_o_r;
reg [25:0] div2hz_cnt;

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

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

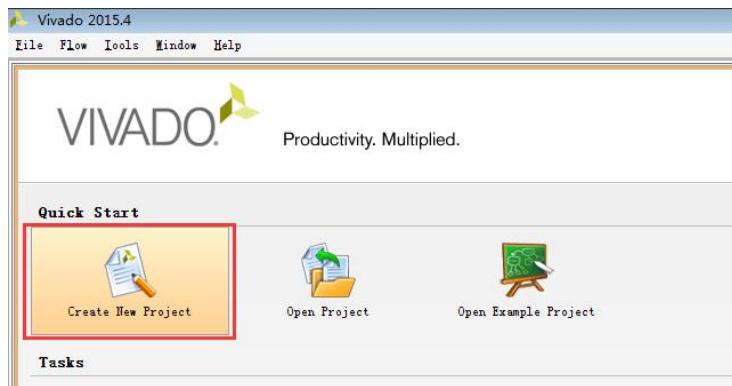
```
assign div2_o=div2_o_r;
assign div3_o=div3_o_r0 | div3_o_r1;
assign div4_o=div4_o_r;
assign div8_o=div8_o_r;
assign div2hz_o=div2hz_o_r;

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

9.7 行为仿真

9.7.1 创建多路分频器工程

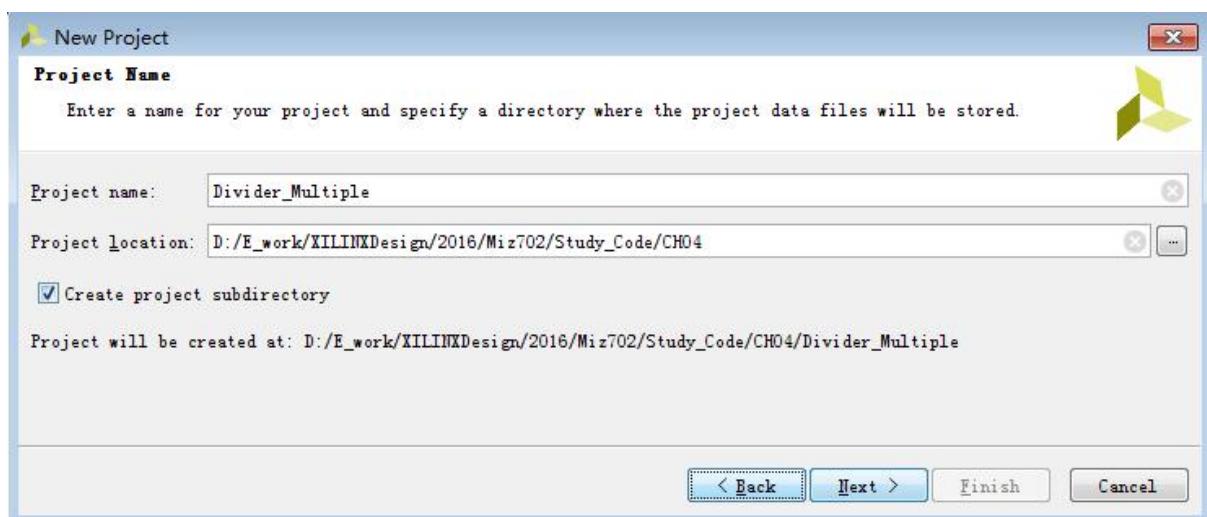
Step1: 创建工程



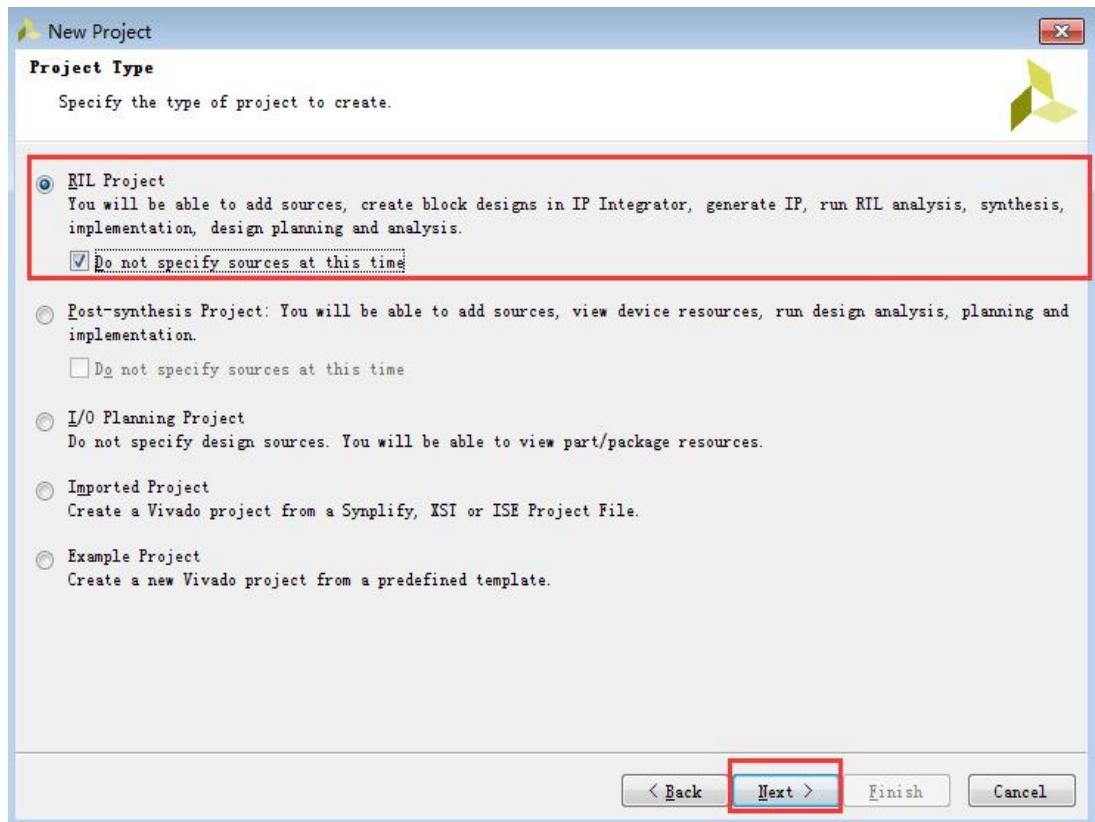
Step2: 欢迎界面直接单击 NEXT



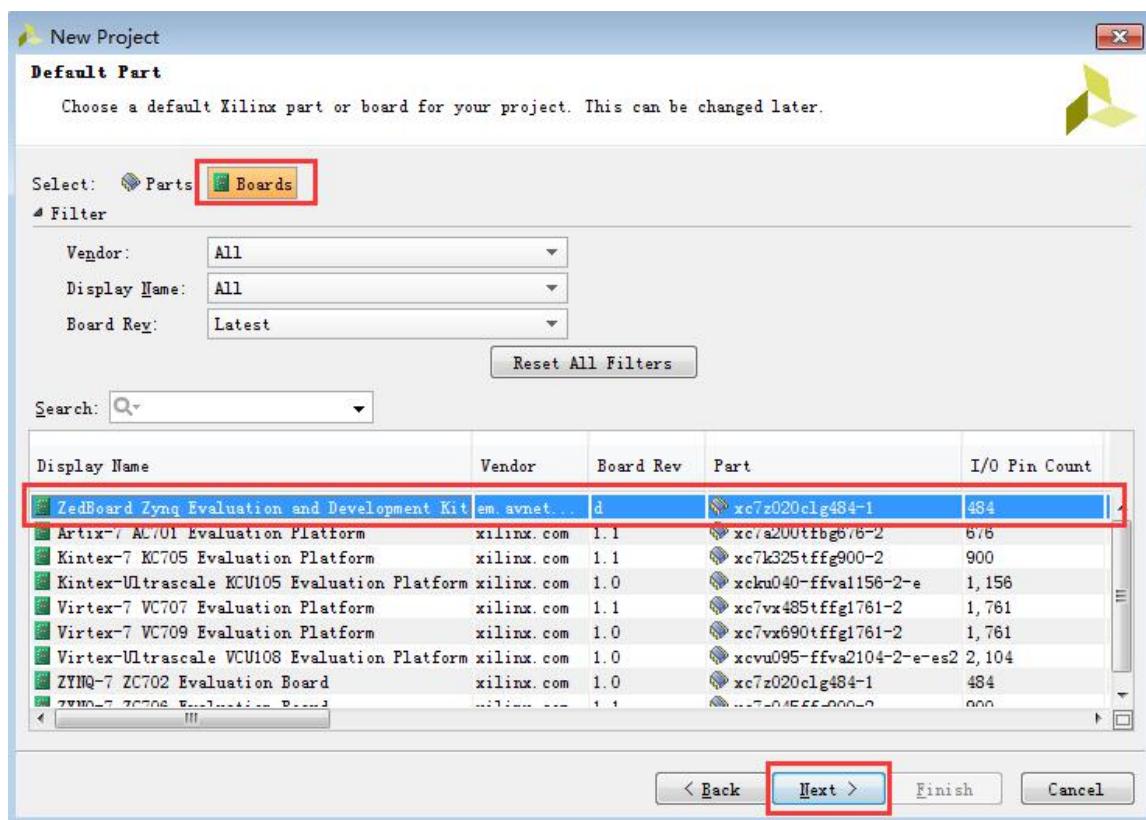
Step3:工程名字命名为 Divider_Multiple，并且设置保存的路径，单击 NEXT



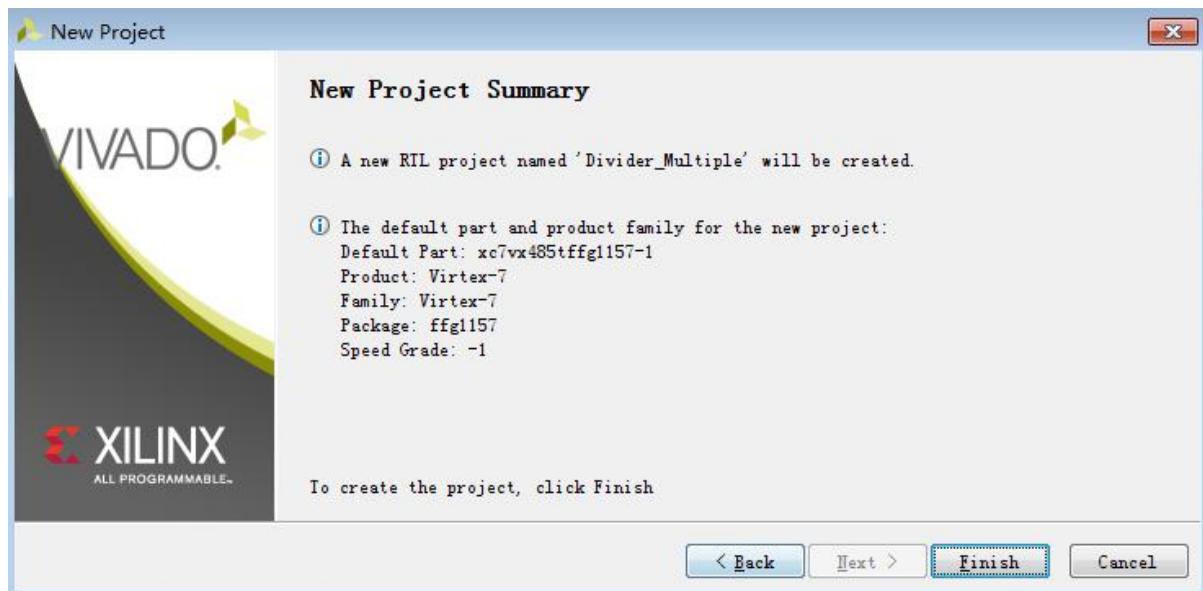
Step4:新建一个 RTL 工程，并且勾选不要添加源文件，单击 NEXT



Step5:由于 MIZ702 和 ZEDBOARD 是兼容的，因此直接选择 ZEDBOARD 硬件开发包作为我们 MIZ702 的开发包。这样可以省去很多麻烦，达到事半功倍的目的。单击 NEXT

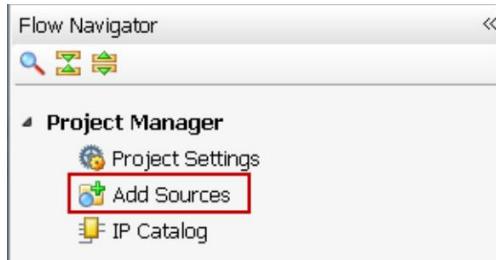


Step6:最后单击 Finish 完成工程的创建

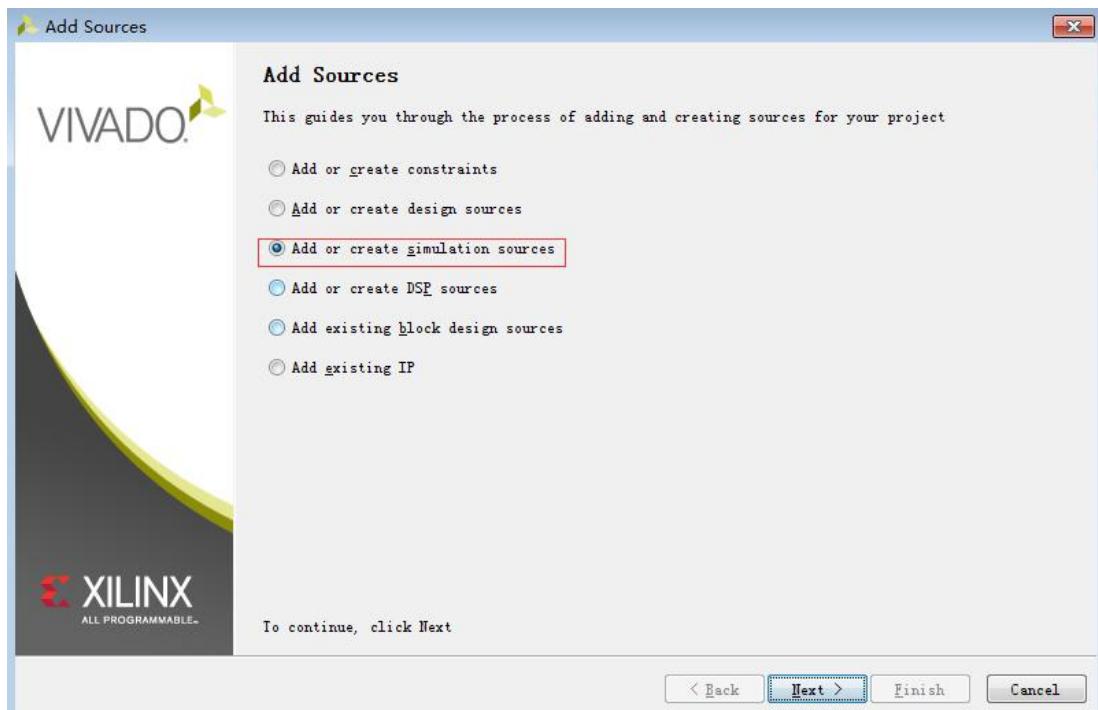


9.7.2 添加仿真文件

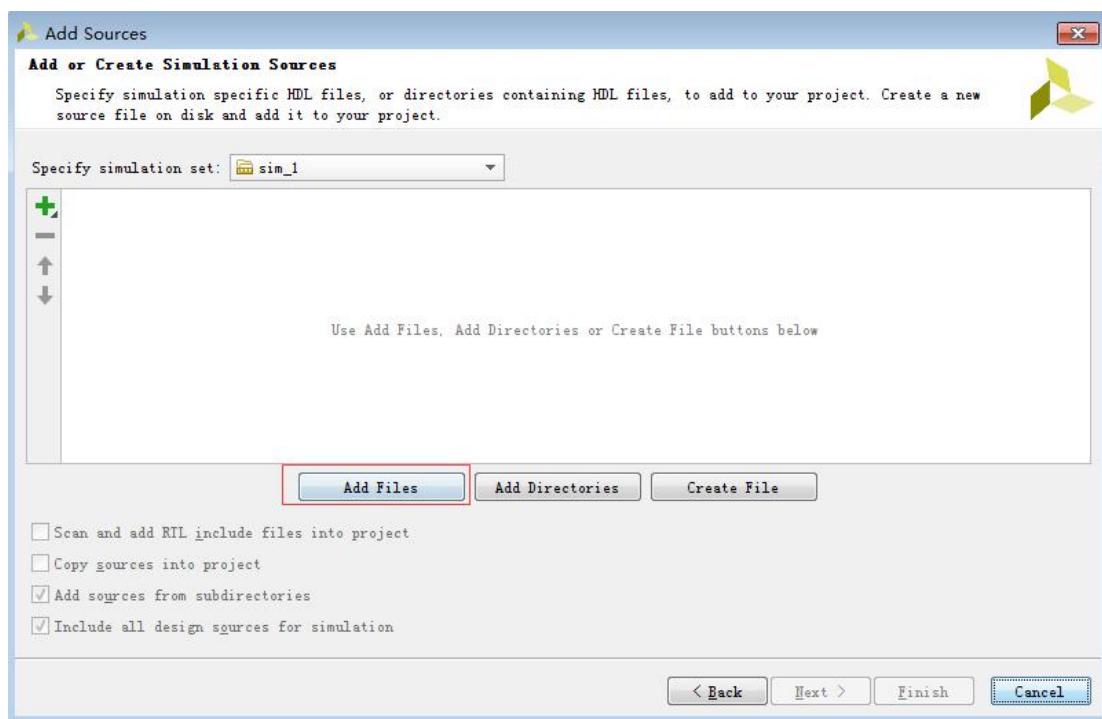
Step1:单击 Add Sources 添加仿真文件



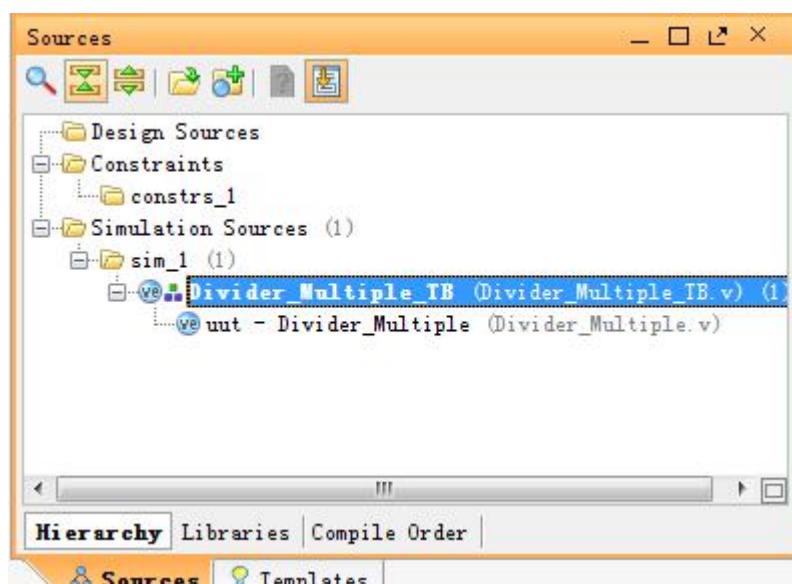
Step2:单击 Add Sources 添加仿真文件



Step3:单击 Add files 把仿真文件添加进来



Step4: 单击 Add files 把仿真文件添加进来



Step5: 仿真文件源码

```
module Divider_Multiple_top_TB;

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

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

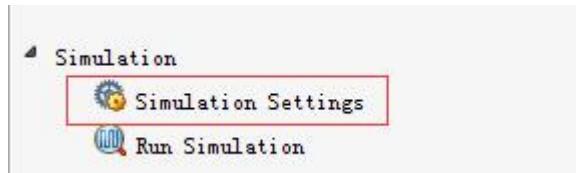
// Instantiate the Unit Under Test (UUT)
Divider_Multiple_top uut (
    .clk_i(clk_i),
    .rst_n_i(rst_n_i),
    .div2_o(div2_o),
    .div3_o(div3_o),
    .div4_o(div4_o),
    .div8_o(div8_o),
    .div2hz_o(div2hz_o)
);

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

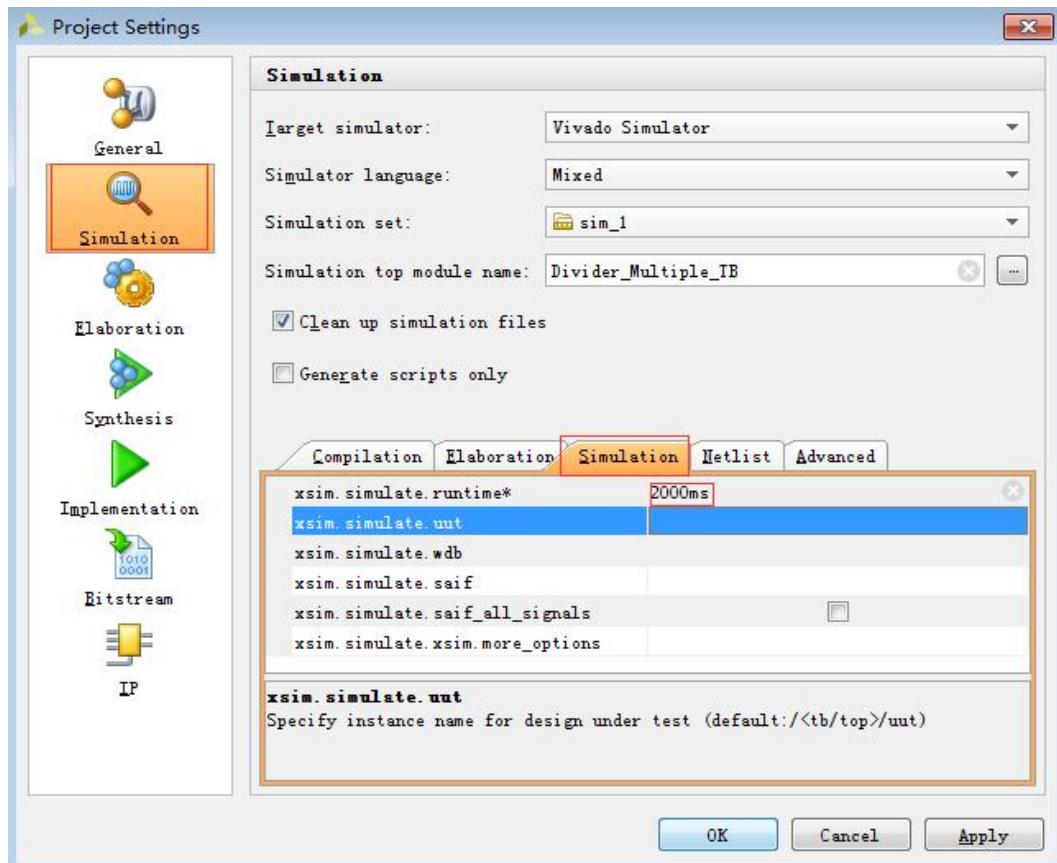
    // Wait 100 ns for global reset to finish
    #96;
    rst_n_i=1;
end
always
begin
    #5 clk_i=~clk_i;
end

endmodule
```

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

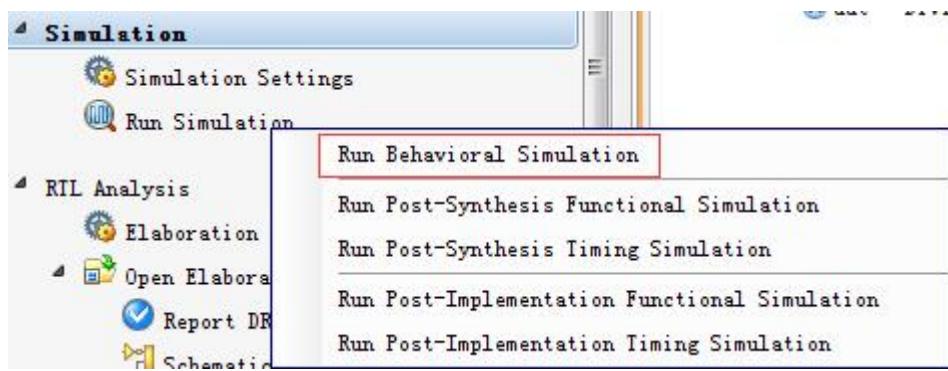


Step7:单击 Simulation 设置仿真时间为 1000ms



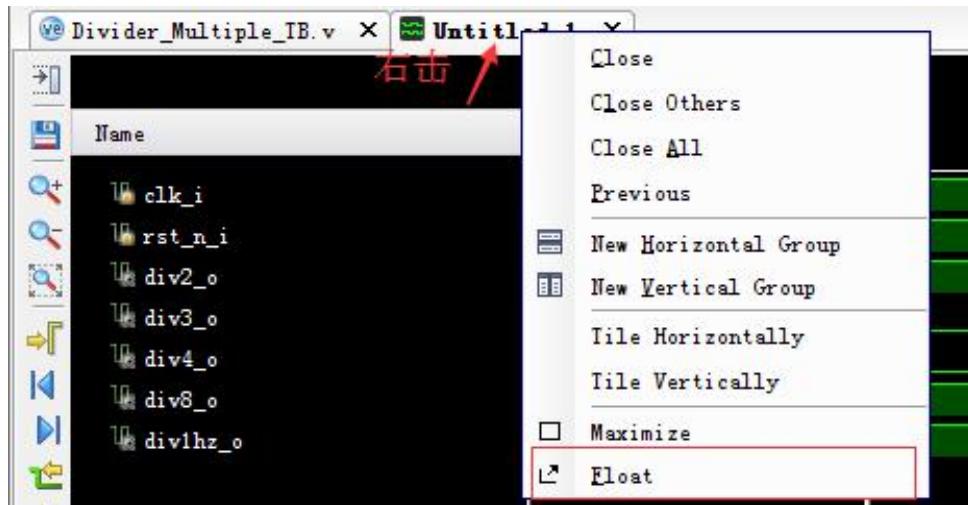
9.7.3 行为级仿真

Step1:单击 Run Simulation

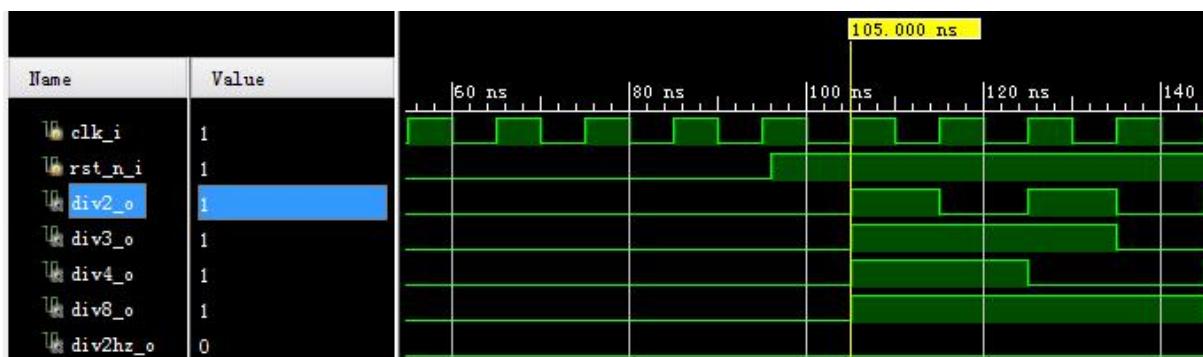


Step2:计算机 CPU 会模拟 FPGA 的运行，1000ms 运行来说通常需要几分钟时间。具体时间和 CPU 的配置有很大关系。

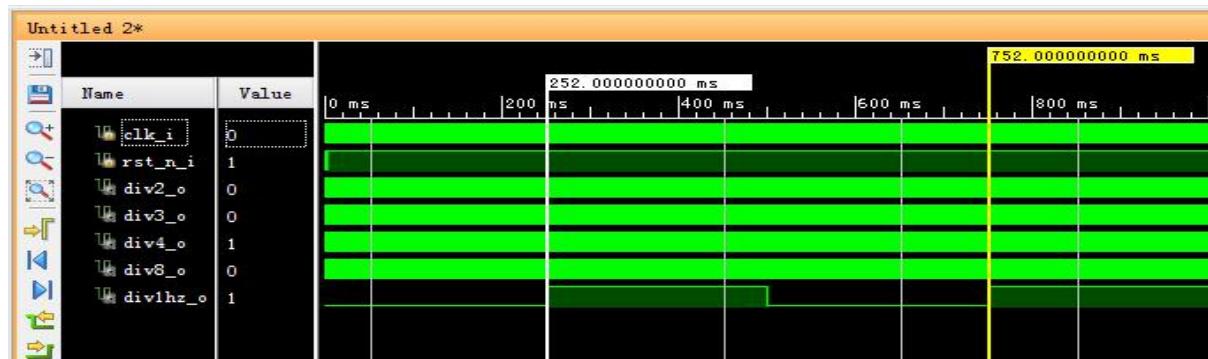
Step3: 仿真结束后查看波形，为了观察方便，右击窗口选择 float



Step4: 使用放大工具放大后观察



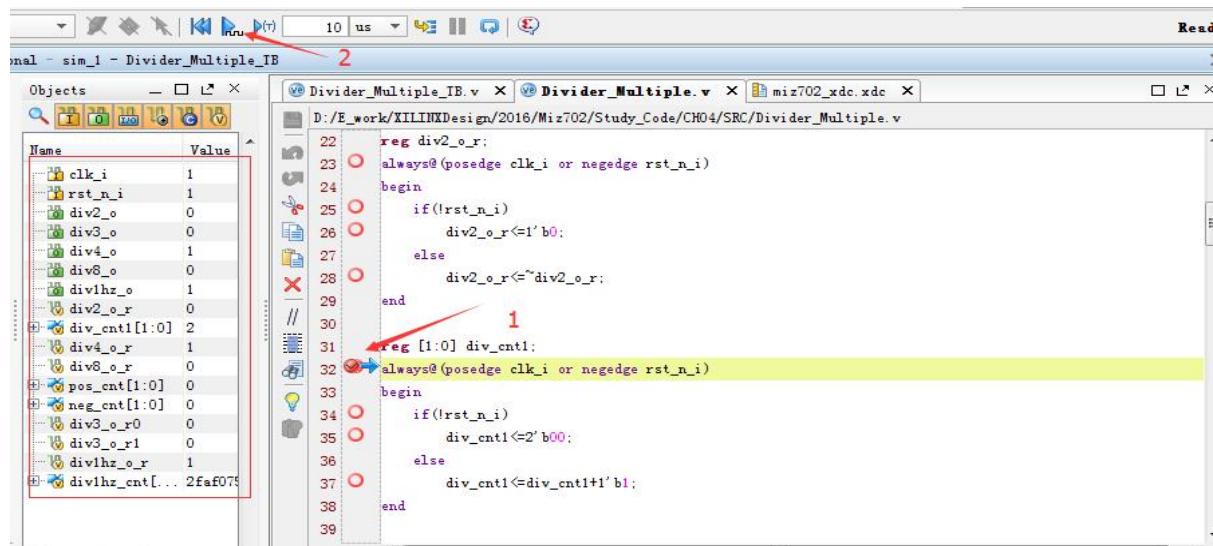
Step5: 使用放大工具放大后观察



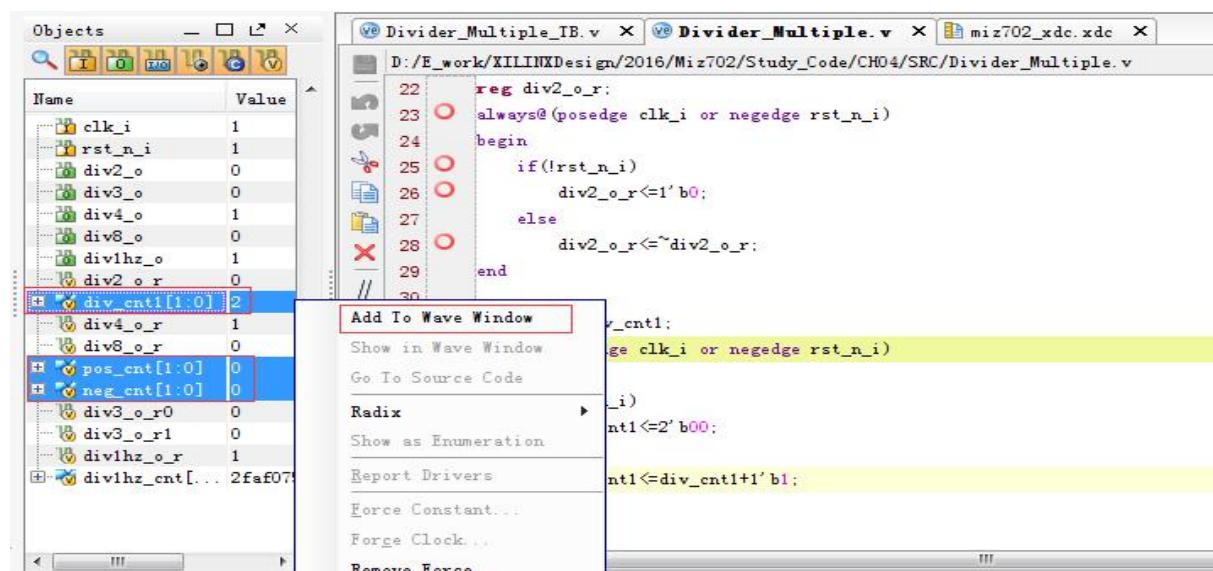
Step6: 断点观察更多信号

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

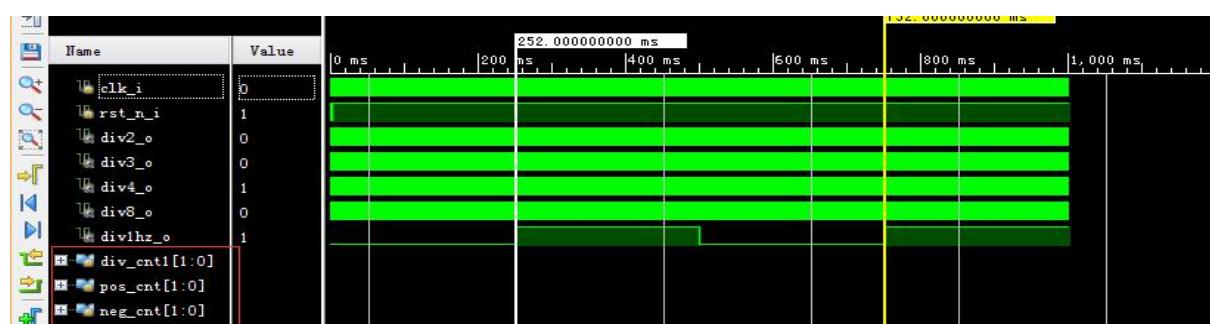
序更加仔细。



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



Step8:可以看到我们添加进来的三个寄存器变量



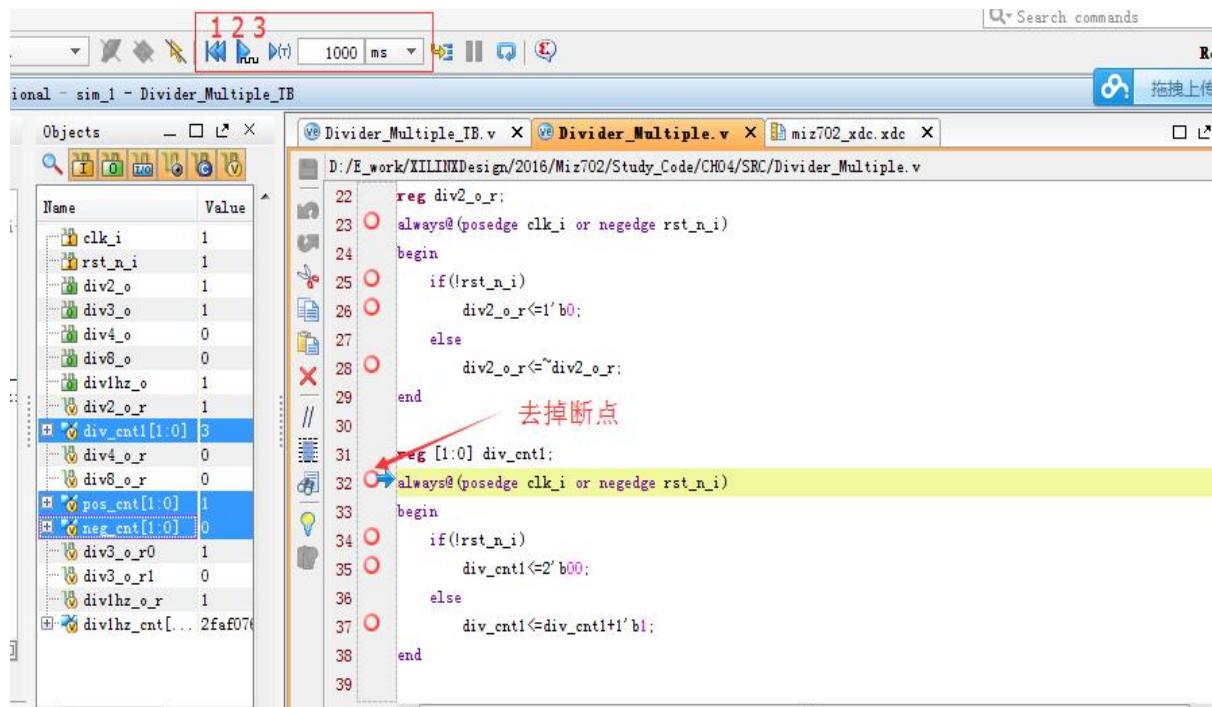
Step9:重新仿真 按钮 1 是初始化仿真 按钮 2 是仿真开始 按钮 3 是仿真到设置时间

1、去掉刚才设置的断点

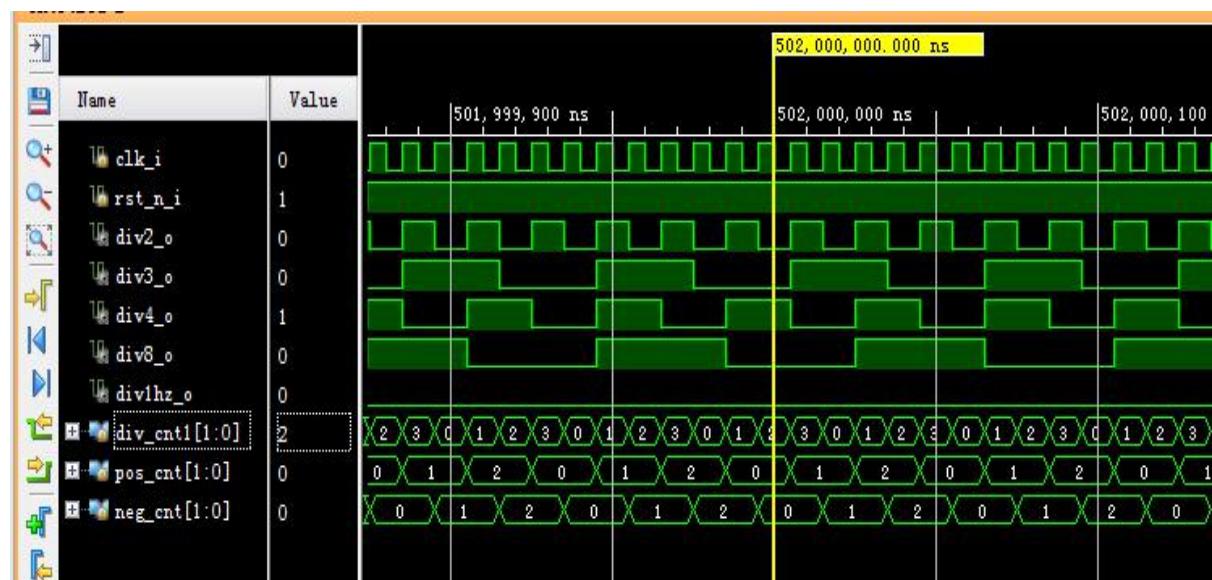
2、单击 1 处按钮重新加载初始化仿真

3、设置仿真时间为 1000ms

4、单击 3 处按钮



Step10: 重新仿真后结果

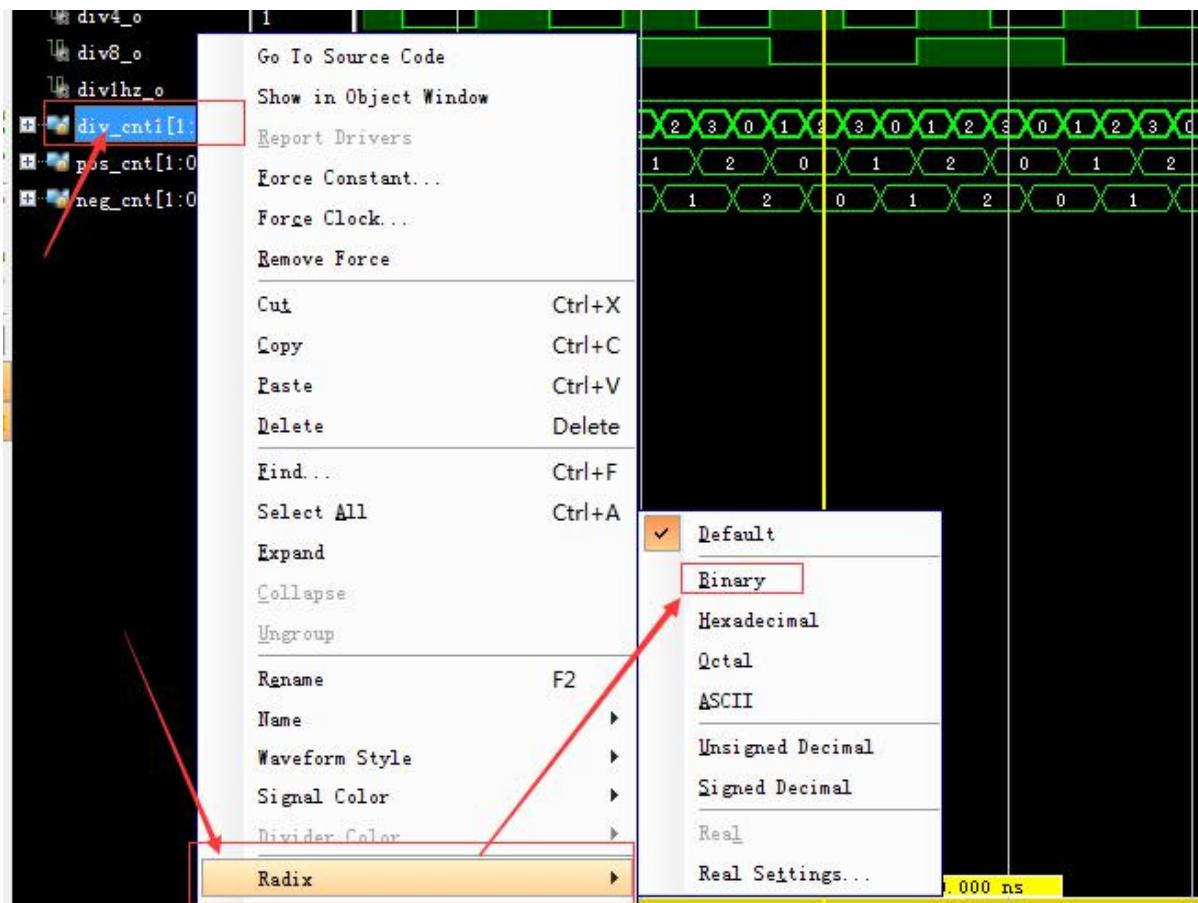


Step11: 设置观察的数据类型

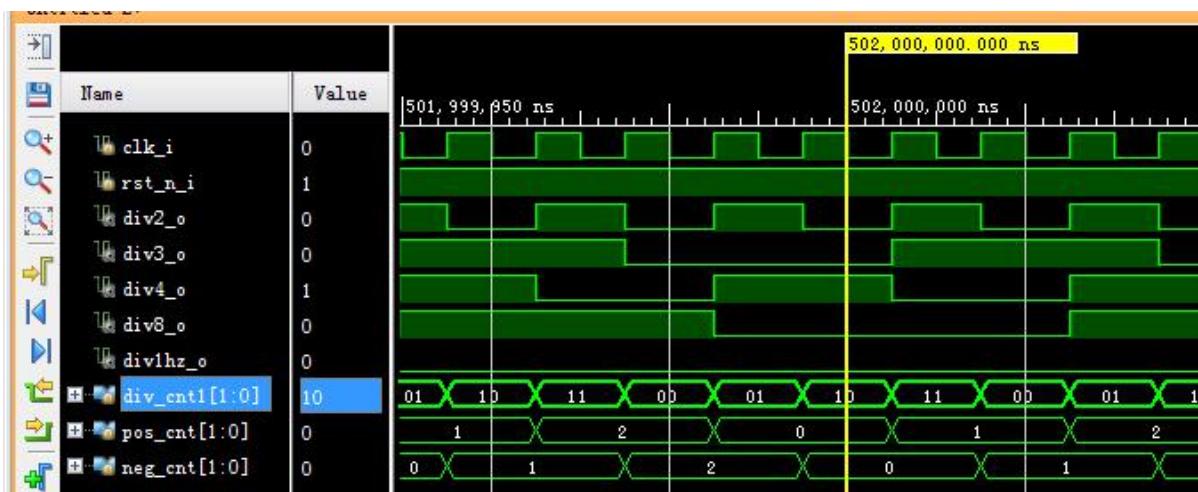
1、首先选择一个要观察的变量

2、右击选择 Radix

3、假设选择 Binary 以二进制形式观察



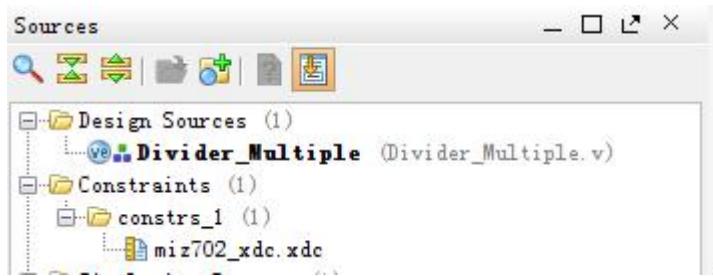
Step12:设置好后的效果



9.8 综合 Synthesis

9.8.1 添加文件

Step1:把 Divider_multiple.v 添加进来，并且创建 xdc 管脚约束文件



XDC 约束文件

```
set_property PACKAGE_PIN Y9 [get_ports {clk_i}]
set_property IOSTANDARD LVCMOS33 [get_ports {clk_i}]

set_property PACKAGE_PIN N15 [get_ports {rst_n_i}]
set_property IOSTANDARD LVCMOS18 [get_ports {rst_n_i}]

set_property PACKAGE_PIN T22 [get_ports {div2hz_o}]
set_property IOSTANDARD LVCMOS33 [get_ports {div2hz_o}]

set_property PACKAGE_PIN T21 [get_ports {div8_o}]
set_property IOSTANDARD LVCMOS33 [get_ports {div8_o}]

set_property PACKAGE_PIN U22 [get_ports {div4_o}]
set_property IOSTANDARD LVCMOS33 [get_ports {div4_o}]

set_property PACKAGE_PIN U21 [get_ports {div3_o}]
set_property IOSTANDARD LVCMOS33 [get_ports {div3_o}]

set_property PACKAGE_PIN V22 [get_ports {div2_o}]
set_property IOSTANDARD LVCMOS33 [get_ports {div2_o}]
```

9.8.2 综合并查看报告

Step1:点击综合按钮



Step2:综合完成后通过查看报告看资源的利用情况

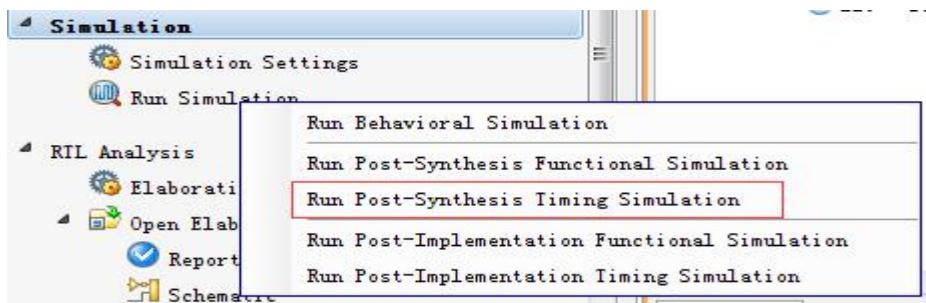


可以看到这个工程只是利用到了很少一部分资源

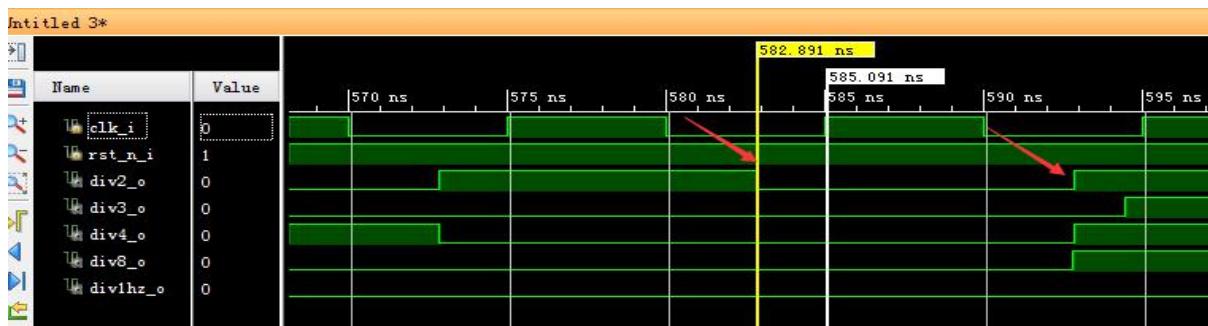


9.8.3 综合时序仿真

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



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



9.9 执行 Implementation

9.9.1 执行并查看报告

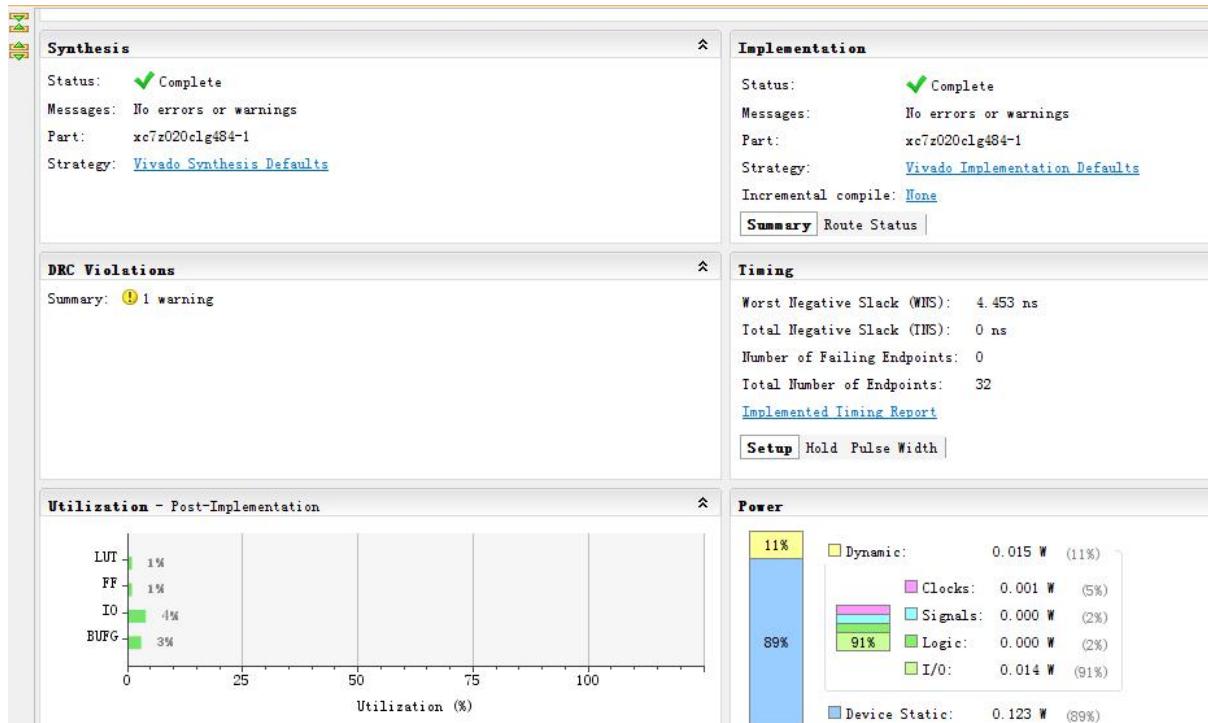
Step1:点击执行按钮



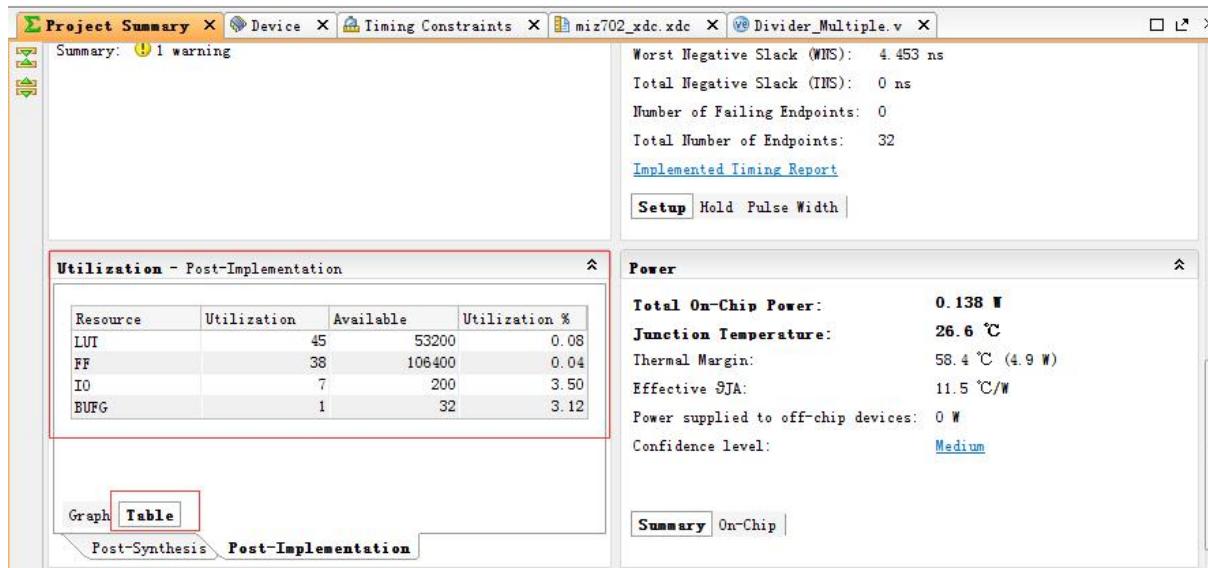
Step2:执行运行完毕后再次单击



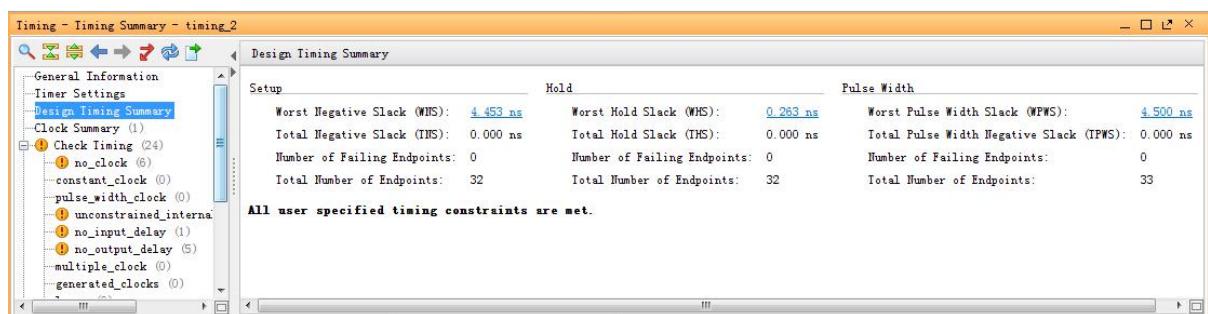
Step3:查看执行运行完毕后的报告,执行完成后的报告比综合后的报告相比,是精确的分析和评估



Step4: 点开 Table 可以看到使用的资料的具体参数



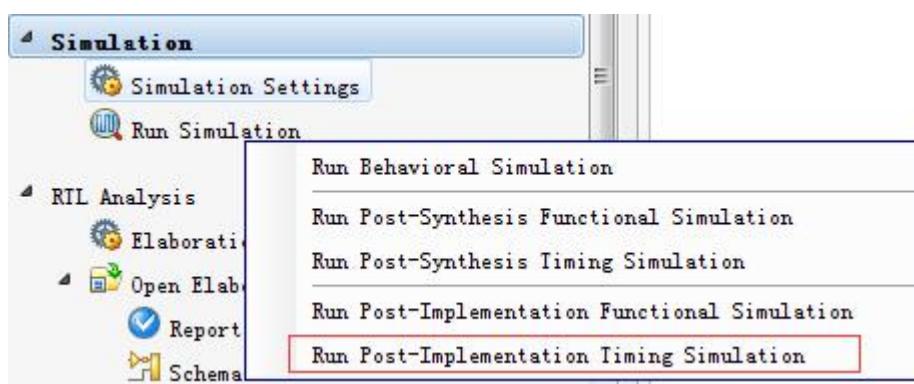
Step5: 查看执行完后的时序约束报告



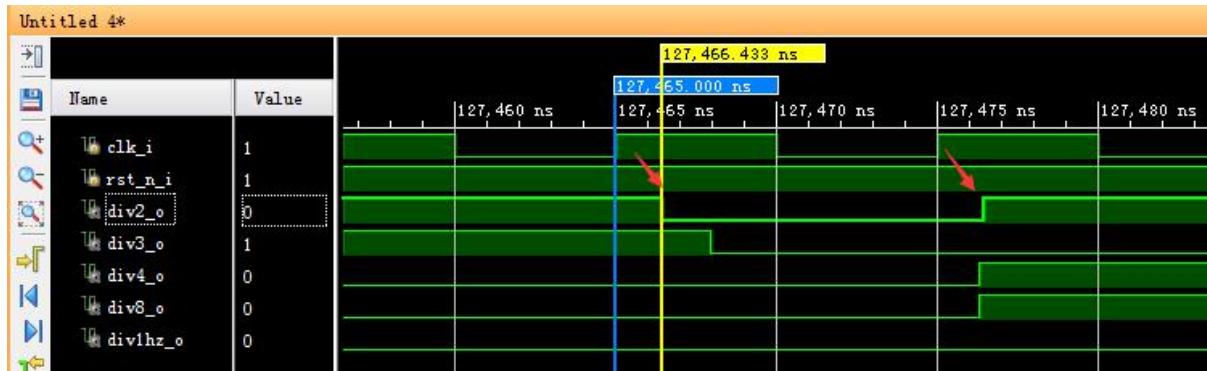
时序约束报告是 FPGA 开发中很重要的一项参数，所以必须看一下是否有违反时序约束的情况。可以看到有一些黄色的 warning。在这里不会影响我们的输出结果，因为我们这输出并没有做时序约束。但是如果要输出很严格的时序就需要加上时序约束。

9.9.2 布局布线后时序仿真

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



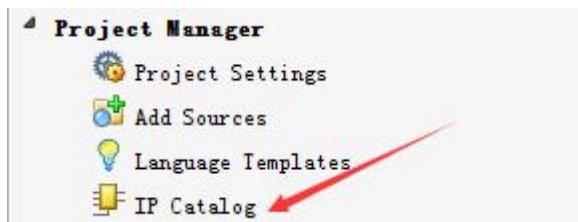
Step2: 观察波形可以清晰看到布局布线后仿真加入了延迟这要比综合后的时序更加接近真实的情况



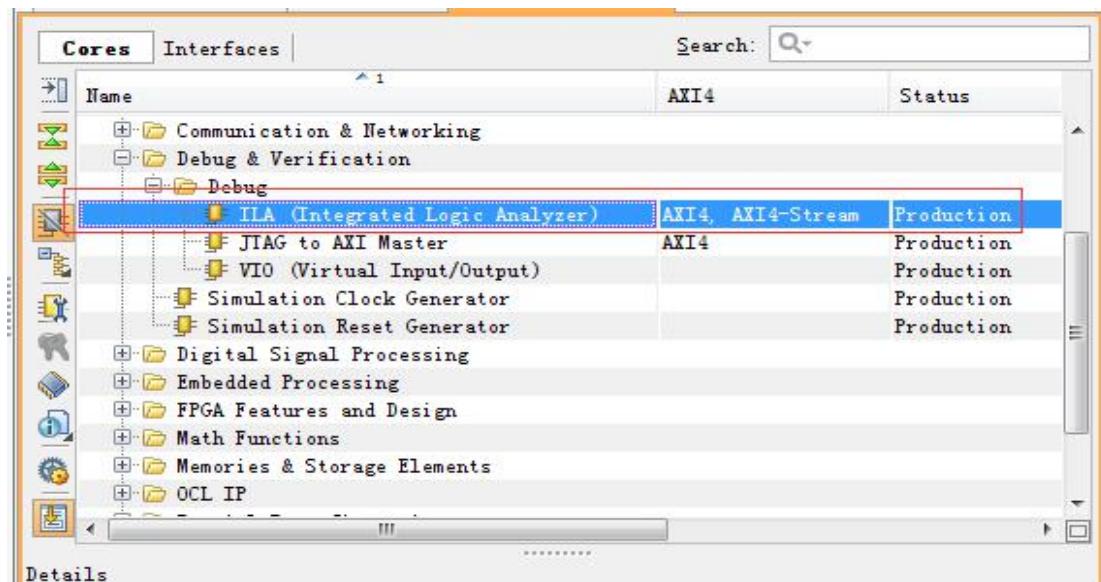
9.10 VIVADO 在线逻辑分析仪使用

9.10.1 IP Catalog 添加 IA ip core

Step1: 单击 IP Catalog



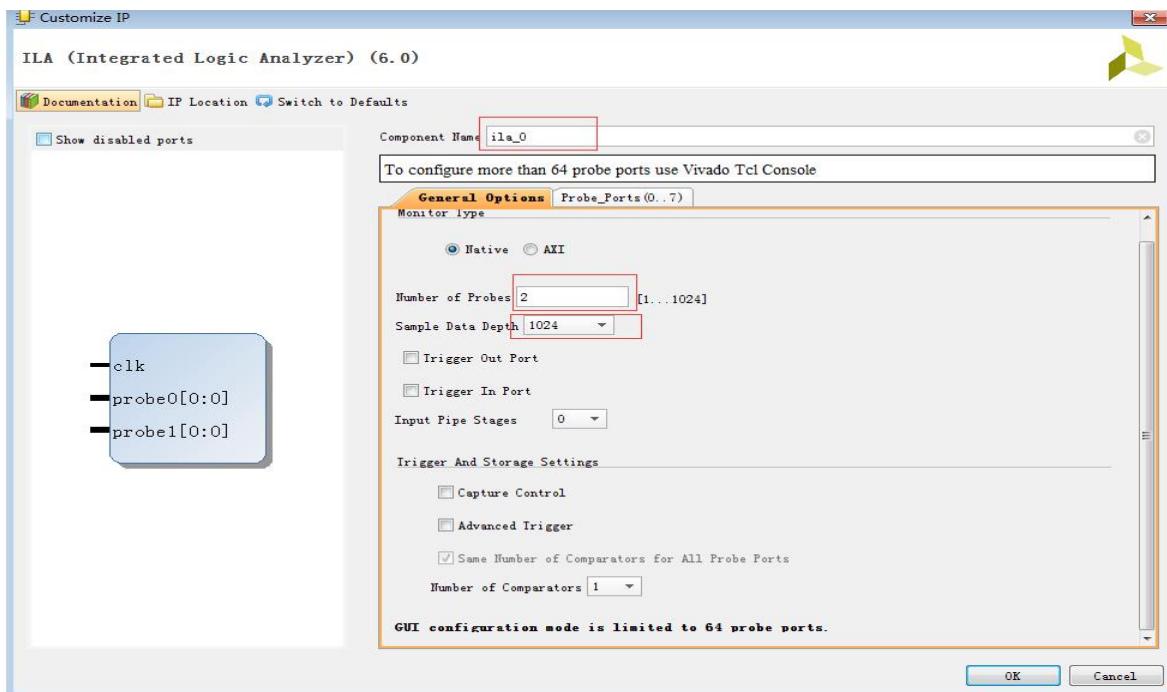
Step2: 打开 Debug & Verification > Debug -> 双击 ILA



Step3: 游标 General Options 设置如下

1、Number of probes 2 为设置需要观察信号的组为2组,因为我们准备1组放触发信号,1组放普通观察的信号

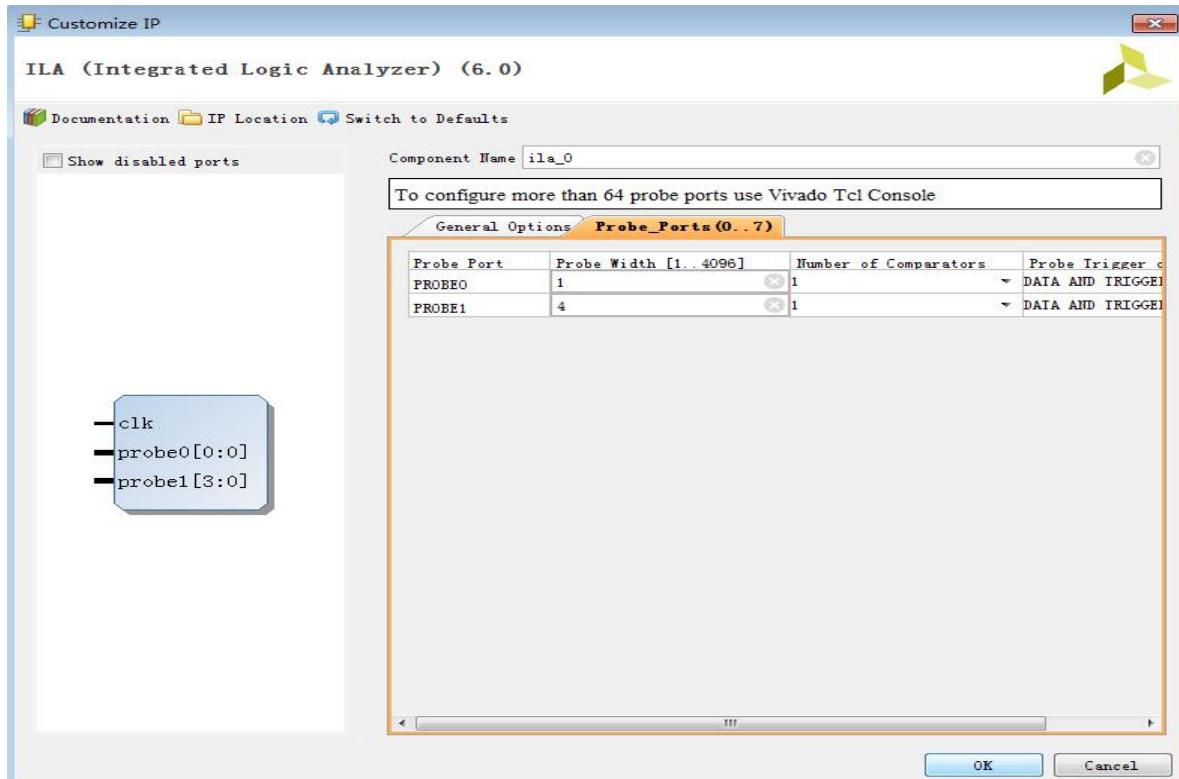
2、Sample Data Depth 1024 设置采样的深度,这是需要消耗FPGA的BRAM的BRAM越大可以设置的采样深度就越大,当然编译速度会降低。



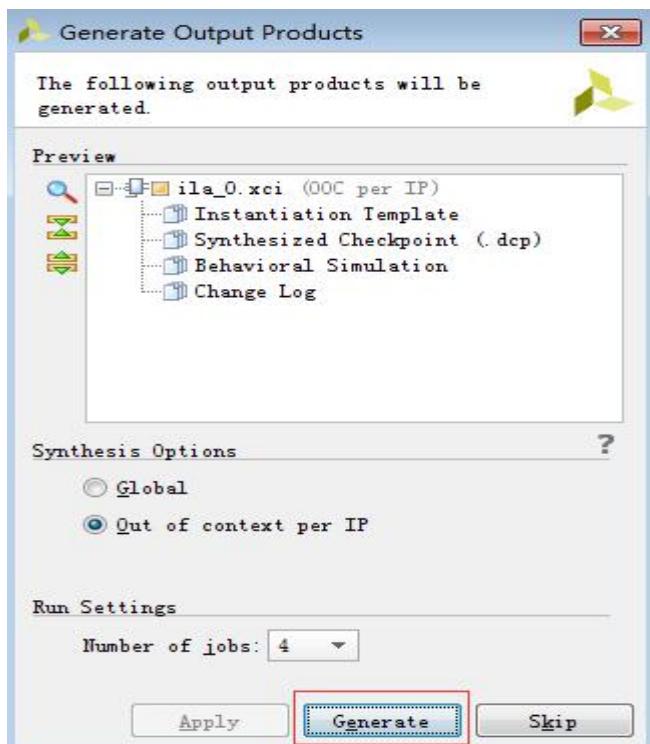
Step4:游标 Probe_Ports 设置如下

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

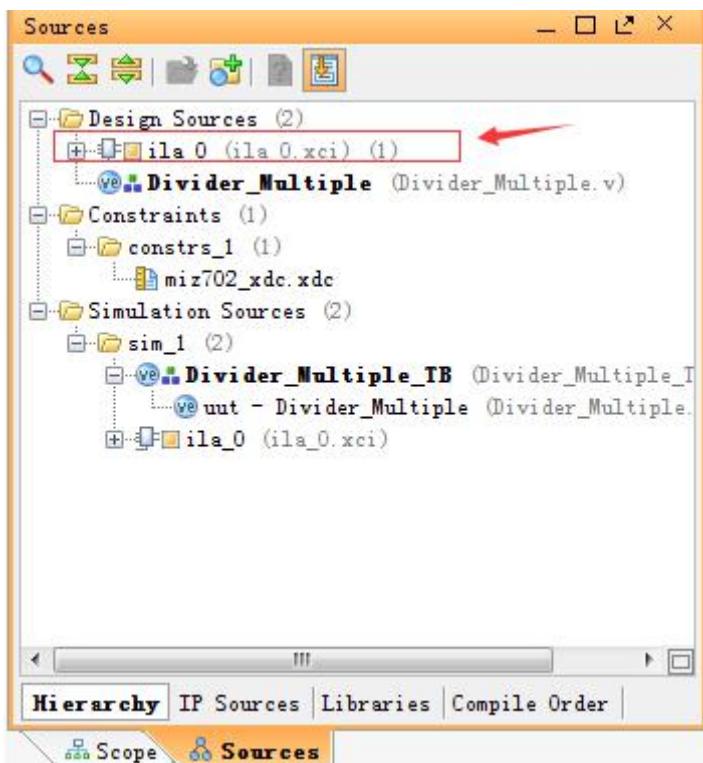
设置好后单击OK关闭窗口



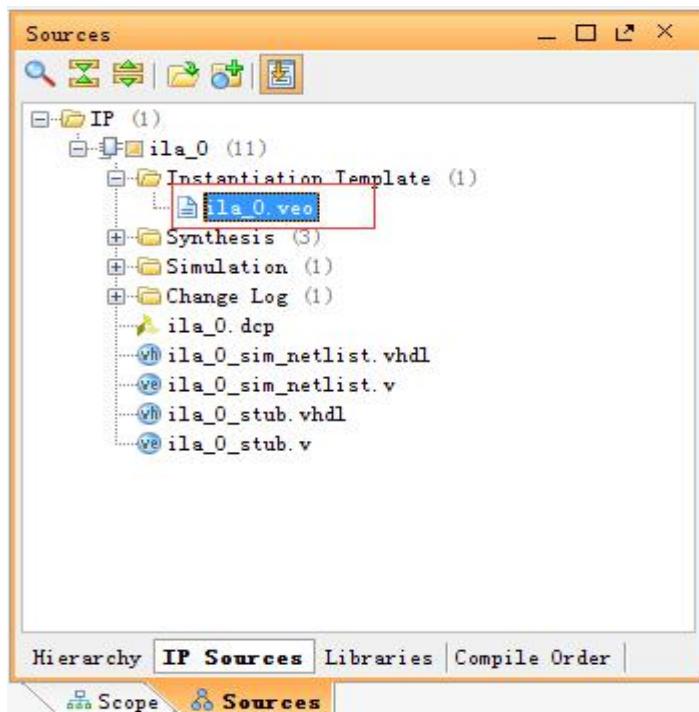
Step5: 直接单击 Generate



Step6: 可以看到 ila 这个逻辑分析仪的 IP 添加进来了



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



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

```

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

```

修改，并且嵌入到顶层文件中

```

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

```

9.10.2 逻辑分析仪抓取的信号

设置好逻辑分析仪，需要抓取的信号为

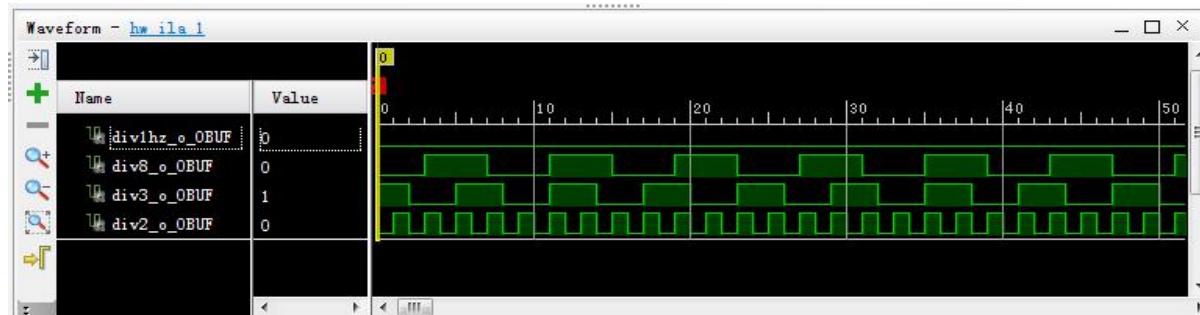
div2_o_r,

```

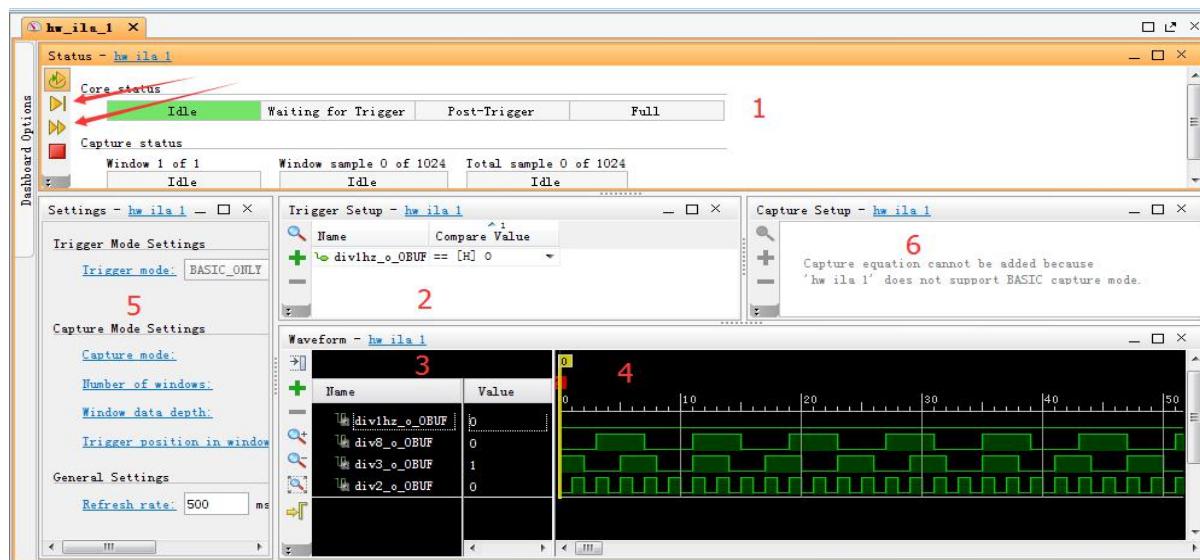
div3_o_r,
div3_o_r0,
div3_o_r1,
div4_o_r,
div8_o_r。

```

逻辑分析仪抓取的信号如下图所示。



9.10.3 逻辑分析仪使用



区域 1: 设置采样的启动停止, 和采样的方式

区域 2: 设置触发信号

区域 3: 被观察的信号名字

区域 4: 被观察的信号波形

区域 5: 触发模式设置

区域 6: 触发设置

那么我们主要使用的有 1、2、3、4 这几个区域。

9.11 小结

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

S01_CH10_VGA 接口测试

VGA 接口是我们在日常生活中经常能见到的一种显示接口，它也叫 D-SUB 接口。与 HDMI 和 DVI 数字接口不同的是，这是一种模拟接口。其时序比较简单，用途广泛，了解了它的时序后，我们能很快的设计出一个 VGA 接口的测试程序。本章就是设计一个 VGA 的测试程序。本章使用的开发板为 MIZ702 和 MIZ702N 开发板，MIZ701N 用户因为不带 VGA 接口，可直接跳过本章。

10.1 硬件介绍

在我们的 MIZ702 开发板上自带了一个 VGA 接口，具体参见 Miz702fun 文件。VGA 接口分为公头和母头，一般显示器都自带一根公头线，MIZ702 开发平台上是母头，VGA 接口一共 15 根线，分为 3 排，标号如下图所示。



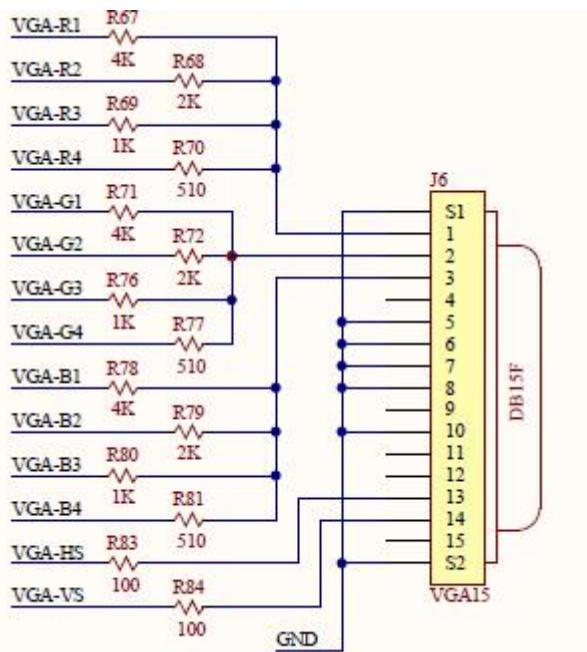
VGA 接口每个引脚的定义如表 11-1 所示。

表 11-1 VGA 引脚定义

标号	名称	描述	标号	名称	描述
1	RED	红色分量视频	9	KEY	保留，每个厂家都不同
2	GREEN	绿色分量视频	10	SGND	同步信号地
3	BLUE	蓝色分量视频	11	ID0	显示器 ID 第 0 位
4	ID2	显示器 ID 第 2 位	12	ID1	显示器 ID 第 1 位
5	GND	地	13	H SYNC	行同步信号
6	RGND	红色分量地	14	V SYNC	场同步信号
7	GGND	绿色分量地	15	ID3	显示器 ID 第 3 位
8	BGND	蓝色分量地			

VGA 的显示效果取决于其三原色分量的位数。最高为 24 位。在我们的 MIZ702 开

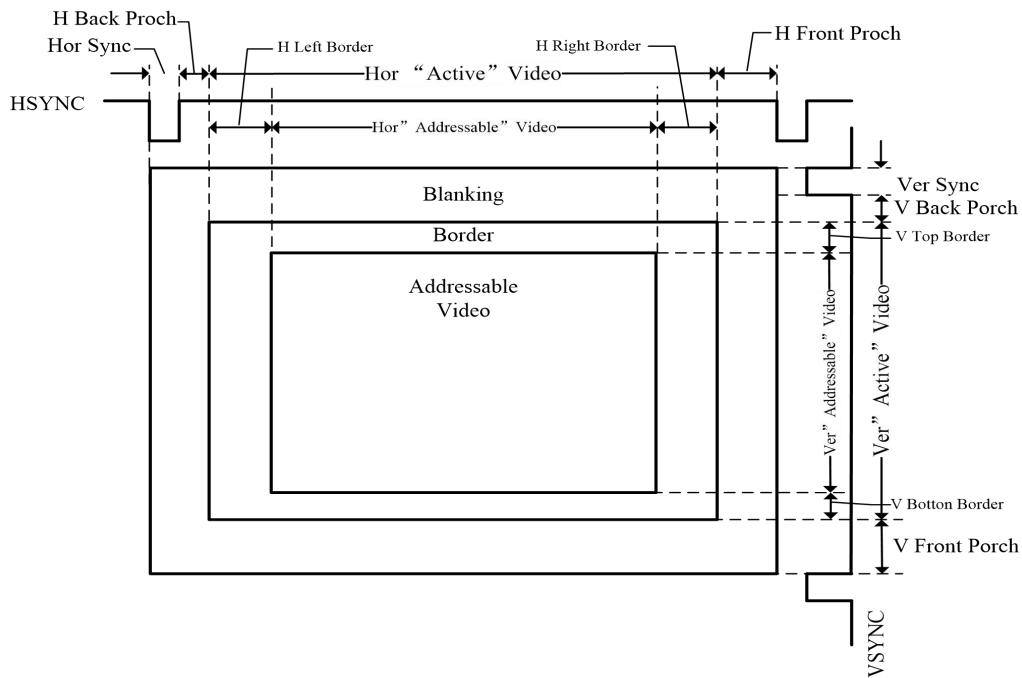
发板上采用了 12 位色彩的设计（MIZ702N 为 16 位色彩），其总共可以显示 2 的 12 次方的色彩数。MIZ702 的 VGA 部分原理图如下图所示：



MIZ702 和 MIZ702N 的 VGA 接口 PIN 脚定义如下：

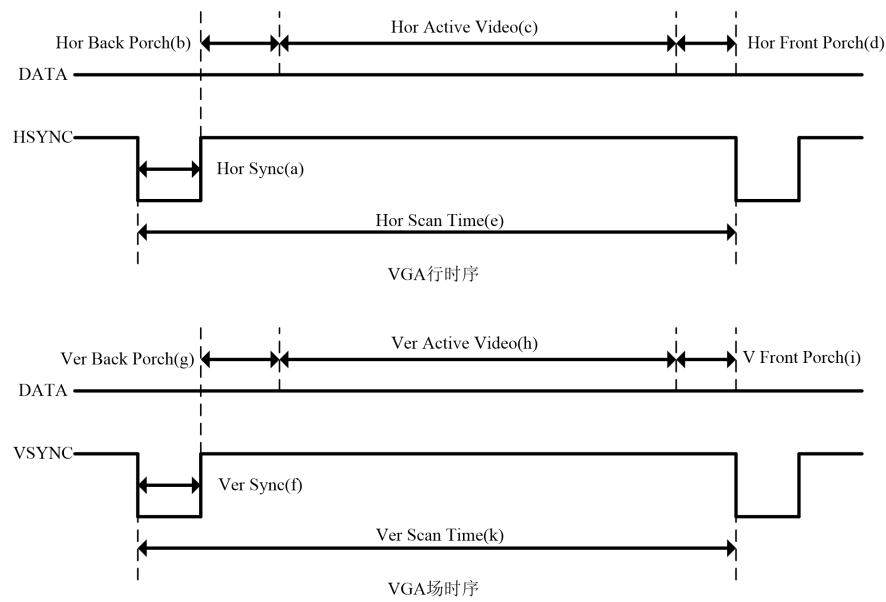
MIZ702		MIZ702N	
VGA_R[0]	V20	VGA_R[0]	T6
VGA_R[1]	U20	VGA_R[1]	R6
VGA_R[2]	V19	VGA_R[2]	U5
VGA_R[3]	V18	VGA_R[3]	U6
VGA_G[0]	AB22	VGA_G[0]	AB11
VGA_G[1]	AA22	VGA_G[1]	AA11
VGA_G[2]	AB21	VGA_G[2]	AB12
VGA_G[3]	AA21	VGA_G[3]	AA12
VGA_B[0]	Y21	VGA_G[4]	T4
VGA_B[1]	Y20	VGA_G[5]	U4
VGA_B[2]	AB20	VGA_B[0]	AA9
VGA_B[3]	AB19	VGA_B[1]	AB7
VGA_HS	AA19	VGA_B[2]	AB10
VGA_VS	Y19	VGA_B[3]	AA8
		VGA_B[4]	AB9
		VGA_HS	Y6
		VGA_VS	W5

10.2 时序分析



VGA 时序如上图所示，编写程序的依据就在这张图中。首先看到有 3 个矩形，第 1 个矩形是 VGA 驱动的最大部分，里面包括所有的信息，在此基础之上有 2 个同步信号，即 HSYNC 和 VSYNC(行同步和场同步)，HSYNC 可以确定一行的开始和结束，VSYNC 可以确定一场的开始和结束，但是同步信号也有时间，因此就引出 Hor Sync 和 Ver Sync (行同步时间和场同步时间)。接下来就是 H Back Proch 和 V Back Porch (行消隐和场消隐)，消隐存在主要是为了兼容电子管屏幕设计的。然后是第 2 个矩形，这是 Hor " Active " Video 和 Ver " Active " Video (行视频有效和场视频有效)，在这个区域中是显示视频的地方。最后就是 H Front Porch 和 V Front Porch (行前肩和场前肩)。到此，一场完整视频就显示完毕了。在这里说一下第 3 个矩形，有 4 个参数 H Left Border、H Right Border、V Top Border 和 V Bottom Border，在不同分辨率中这 4 个参数不同，在 800x600 及其以上的分辨率中这 4 个参数为 0。

说了半天，大家可能晕了？在此给出一个简化的时序图，如下图所示。一行数据包括：Hor Sync (行同步)、Hor Back Porch (行消隐)、Hor Active Video (行视频有效) 和 Hor Front Porch (行前肩)；一场数据包括：Ver Sync (场同步)、Ver Back Porch (场消隐)、Ver Active Video (场视频有效) 和 Ver Front Porch (场前肩)。



VGA 时序主要分为行时序和场时序，行时序是以像素为单位的，场时序是以行为单位的。VGA 行时序对行同步时间、消隐时间、行视频有效时间和行前肩时间有特定要求，列时序也是如此，如果其中一部分时序出现问题就会造成显示出现问题。常用 VGA 分辨率时序参数如下表所示。

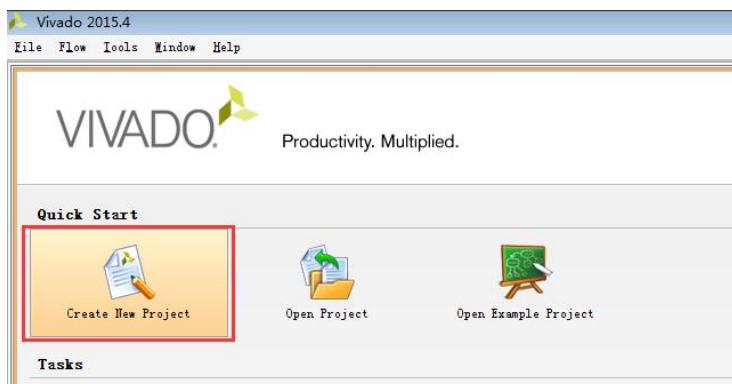
VGA 常用分辨率时序参数

显示模式	时钟 /MHz	行时序参数(单位: 像素)					列时序参数(单位: 行)				
		a	b	c	d	e	f	g	h	i	k
640x480@60Hz	27.175	96	48	640	16	800	2	33	480	10	525
800x600@60Hz	40	128	88	800	40	1056	4	23	600	1	623
1024x768@60Hz	65	136	160	1024	24	1344	6	29	768	3	806
1280x720@60Hz	74.25	40	220	1280	110	1650	5	20	720	5	750
1280x1024@60Hz	108	112	248	1280	48	1688	3	38	1024	1	1066
1920x1080@60Hz	148.5	44	148	1920	88	2200	5	36	1080	4	1125

在这里说一下时钟频率计算，就是上文提到的第一个矩形和场频率有关，思考一下，很简单的。时钟频率=行最大值 x 列最大值 x 扫描频率。

10.3 新建 VIVADO 工程

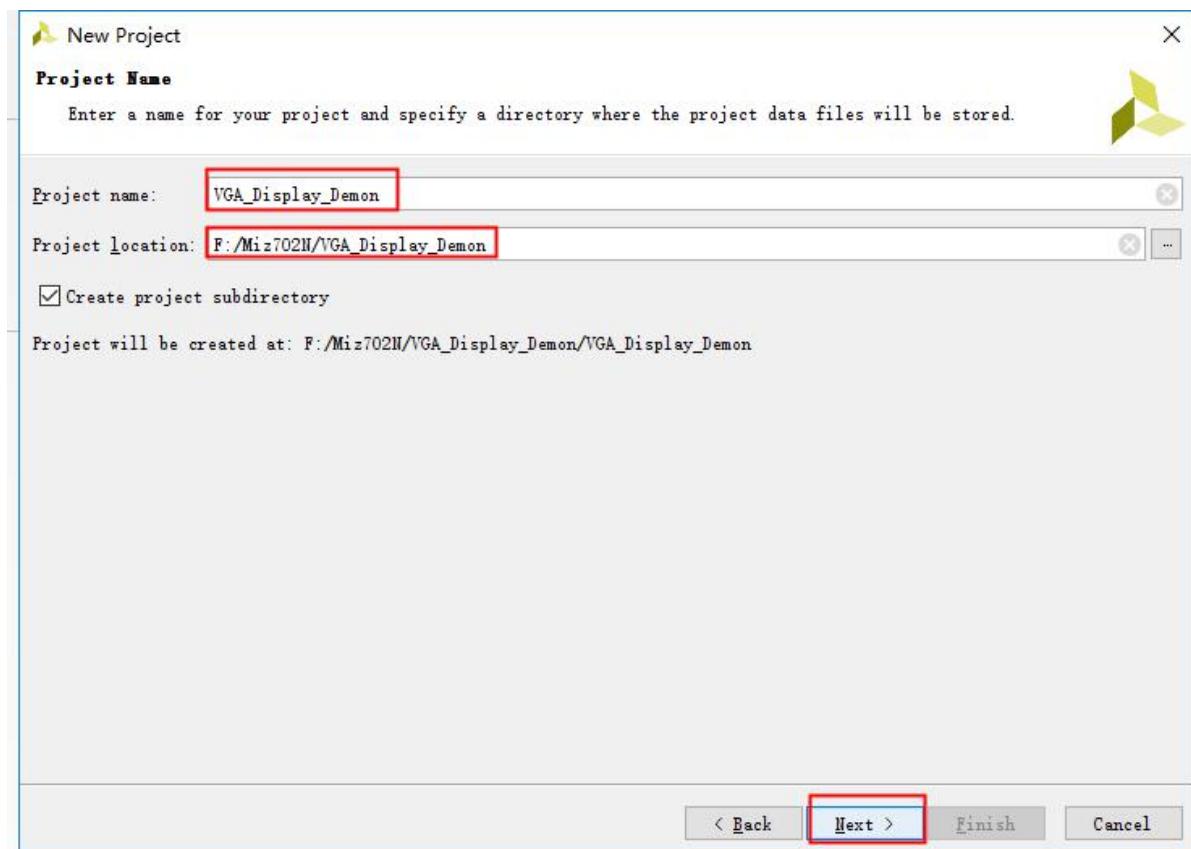
Step1: 创建工程



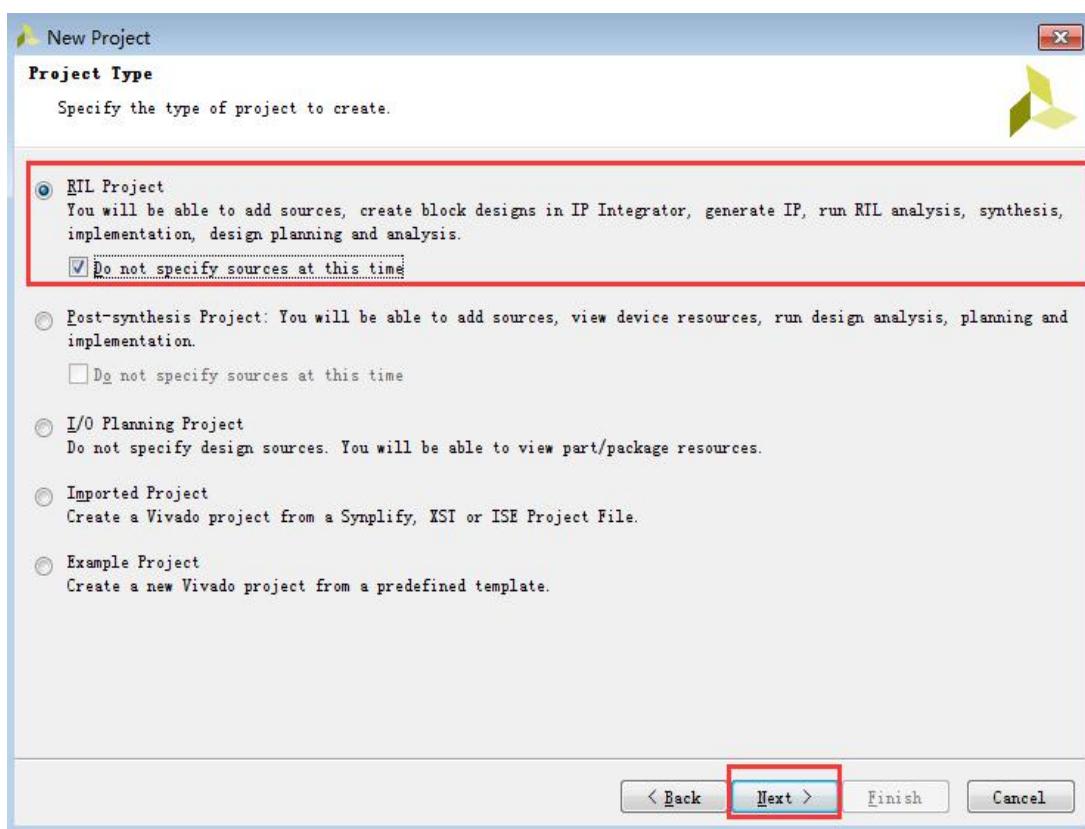
Step2:欢迎界面直接单击 NEXT



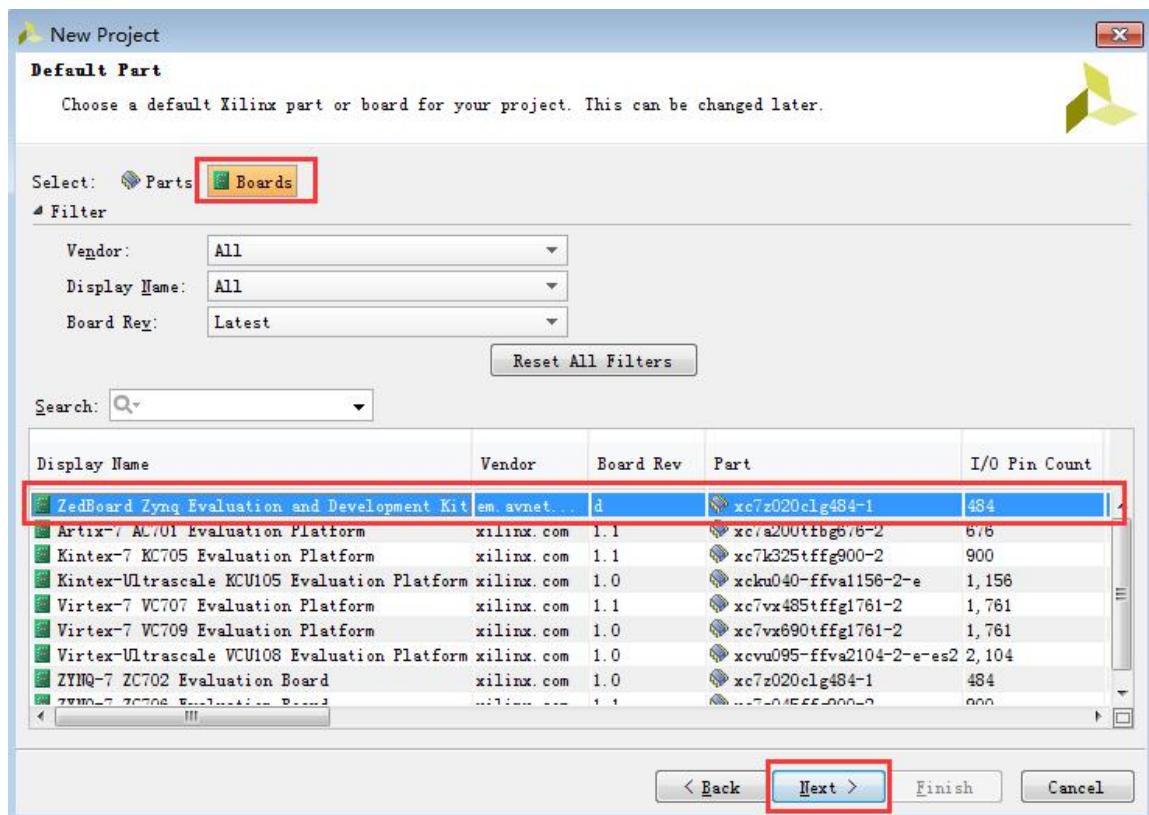
Step3:工程名字命名为 VGA_Display_Demon，并且设置保存的路径，单击 NEXT

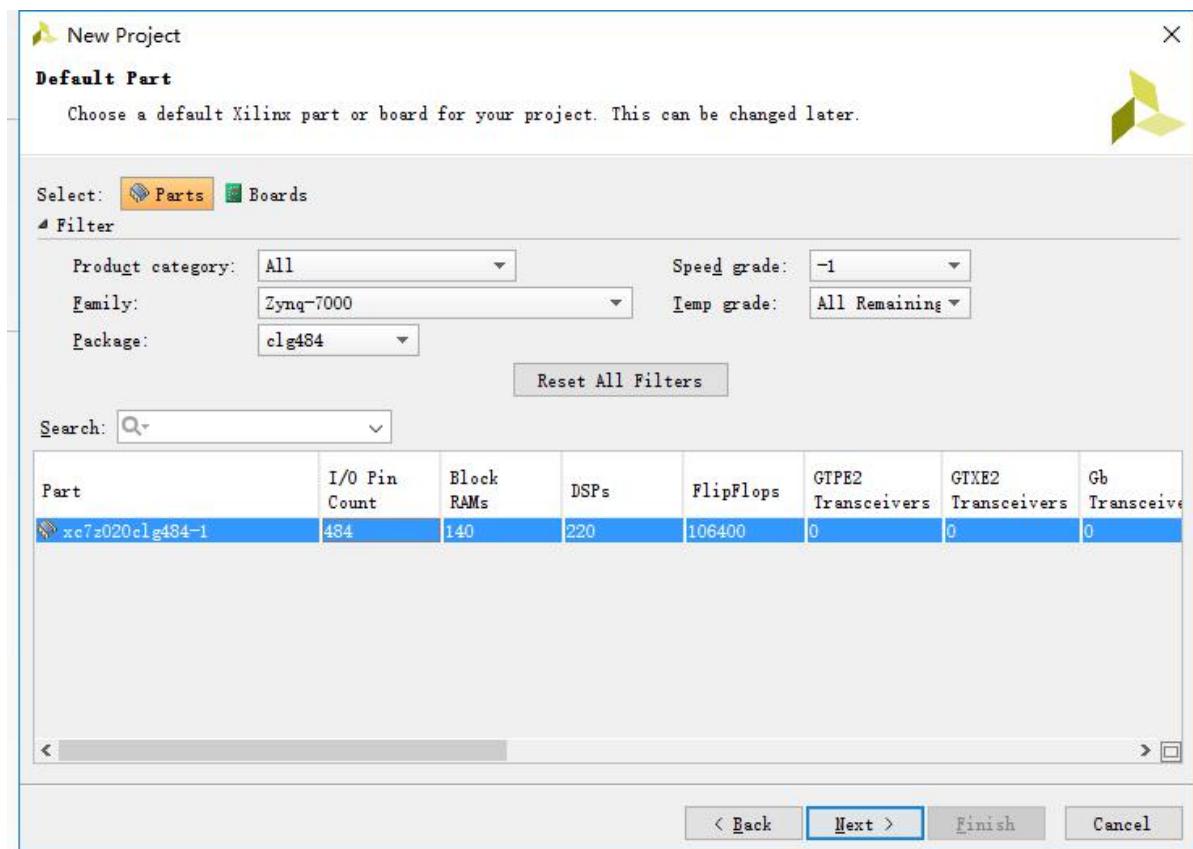


Step4:新建一个 RTL 工程，并且勾选不要添加源文件，单击 NEXT



Step5:由于 MIZ702 和 ZEDBOARD 是兼容的，因此直接选择 ZEDBOARD 硬件开发包作为我们 MIZ702 的开发包（702N 用户请选择 XC7Z020clg484_1 芯片）。这样可以省去很多麻烦，达到事半功倍的目的。单击 NEXT。



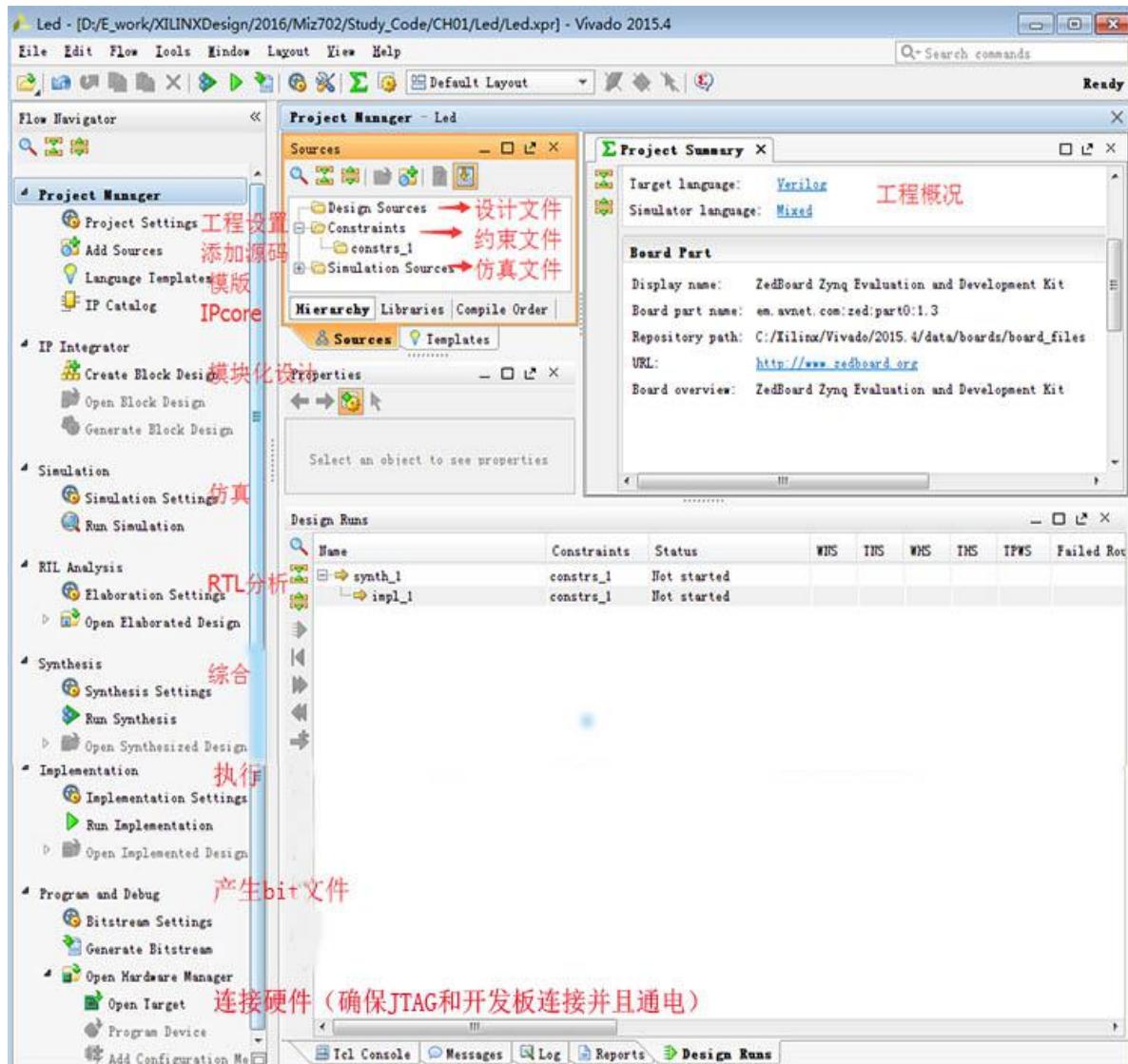


Step6:最后单击 Finish 完成工程的创建

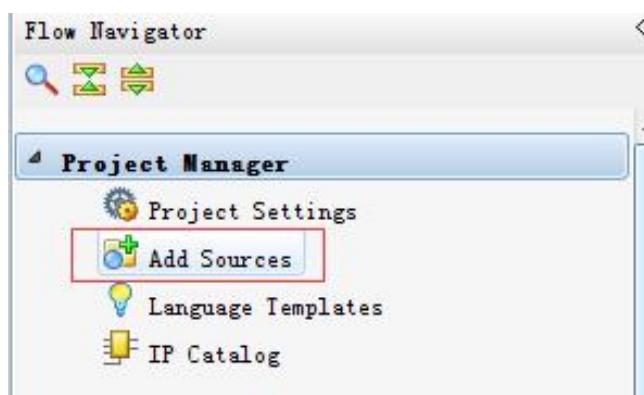


10.4 创建工程文件

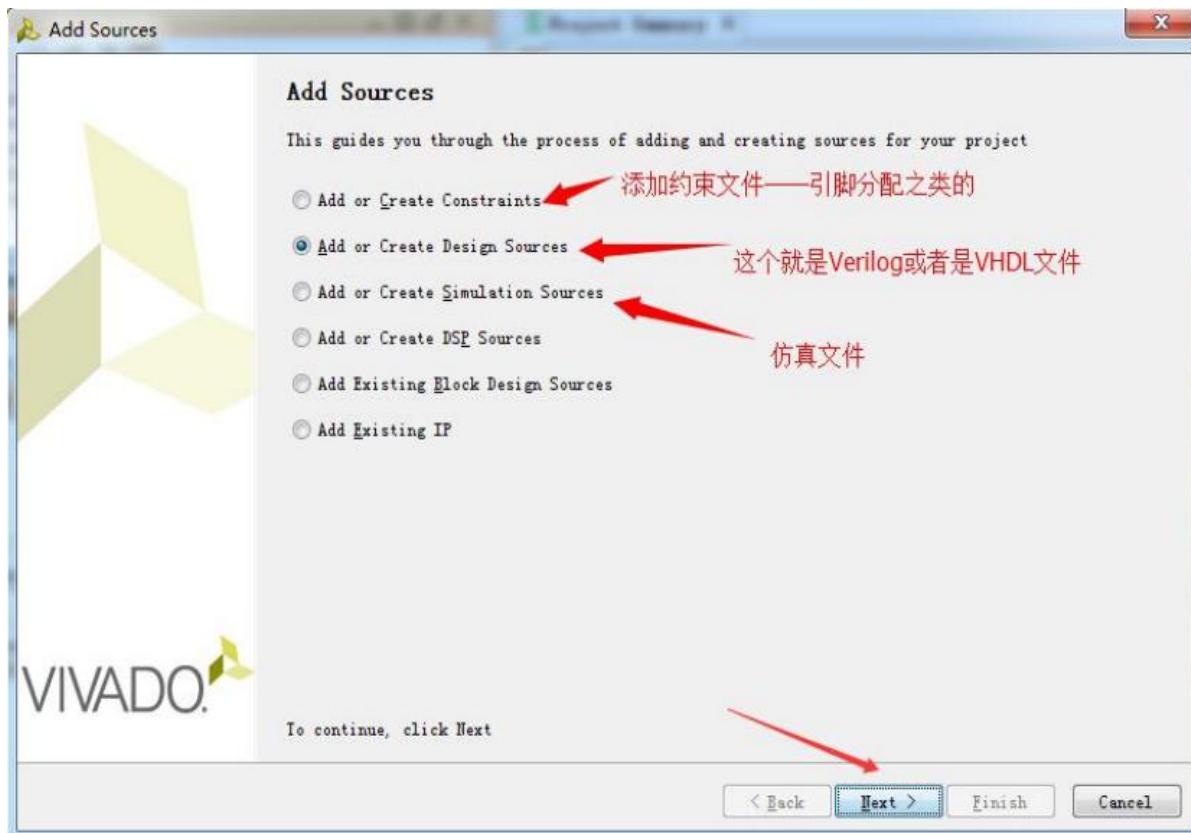
Step1: 打开 VIVADO 软件



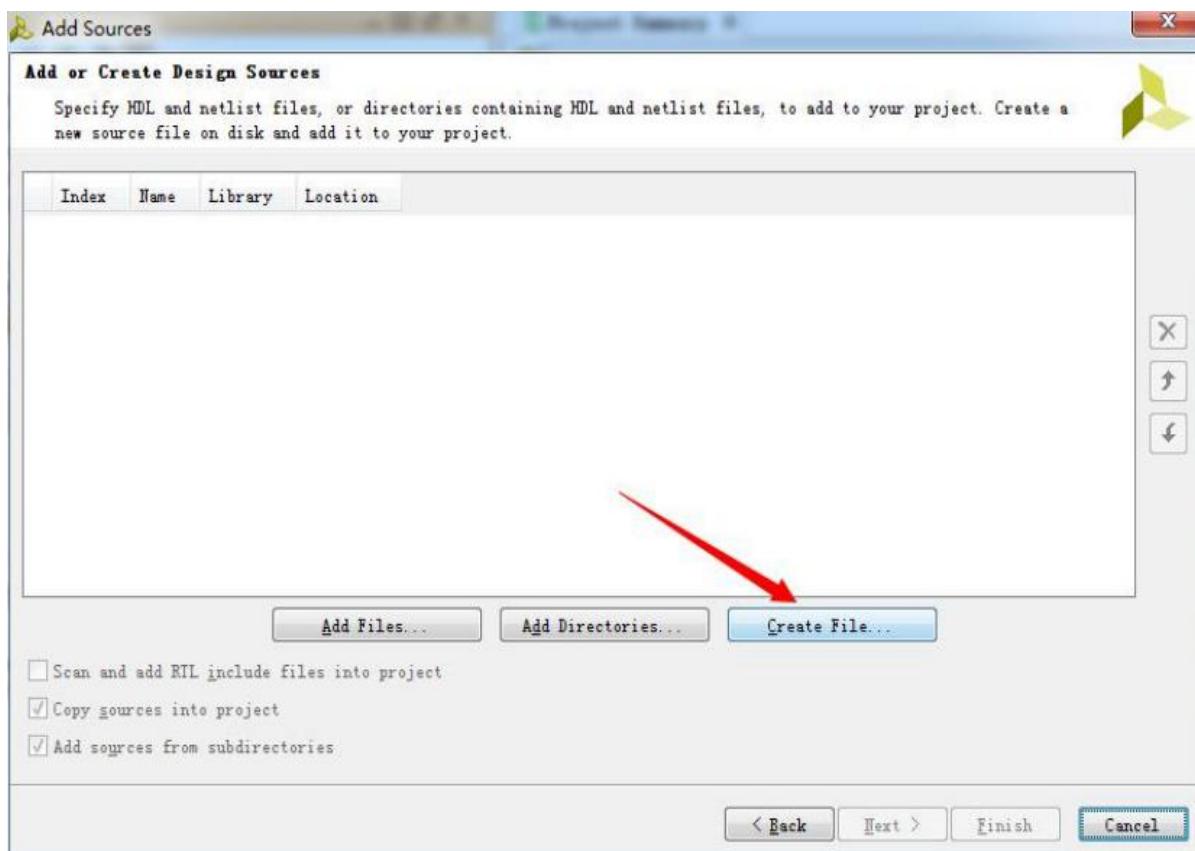
Step2: 单击 Add Sources



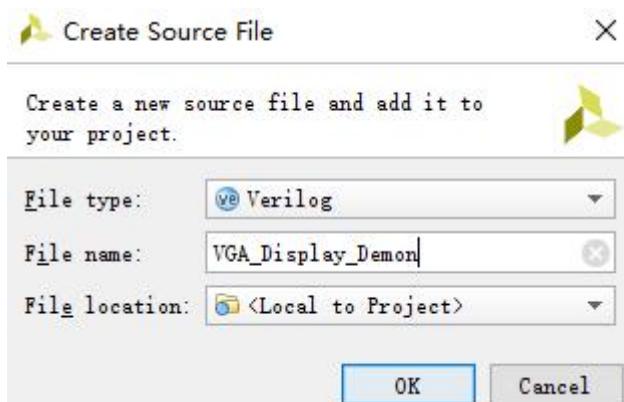
Step4: 选择单击 Add or Create Design Sources 然后单击 NEXT



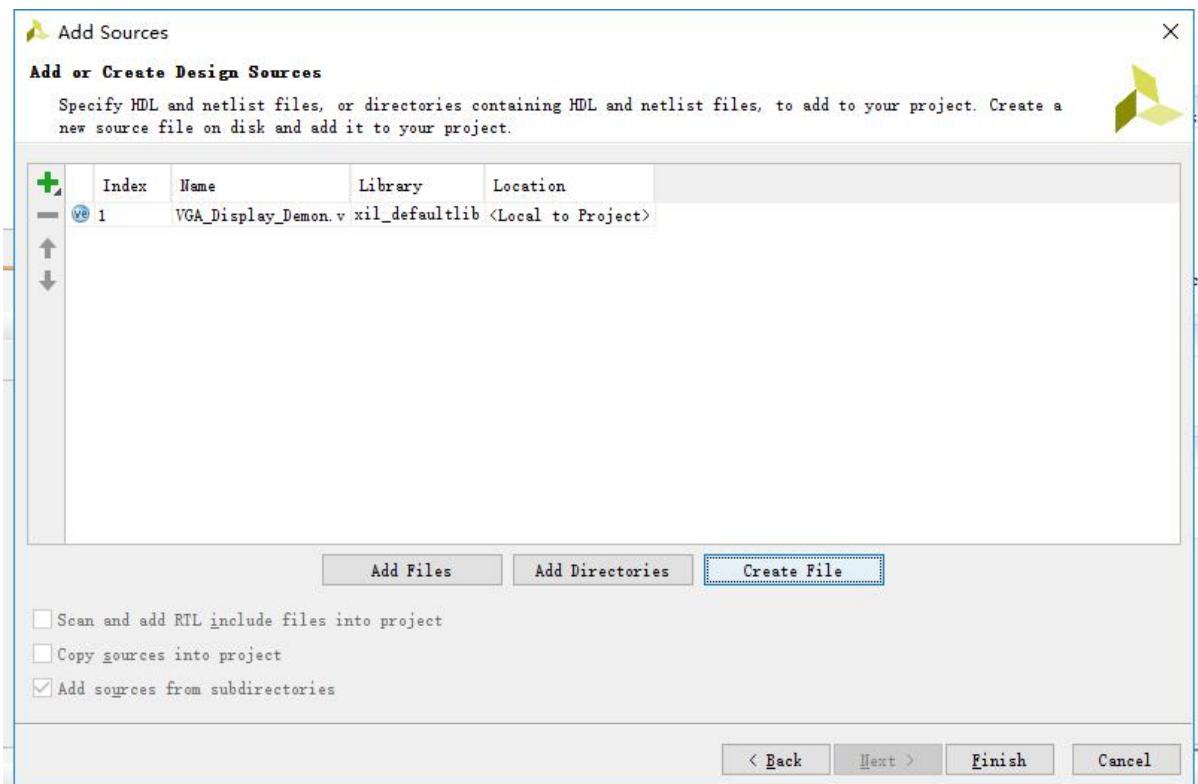
Step5: 单击 Create File 来创建文件



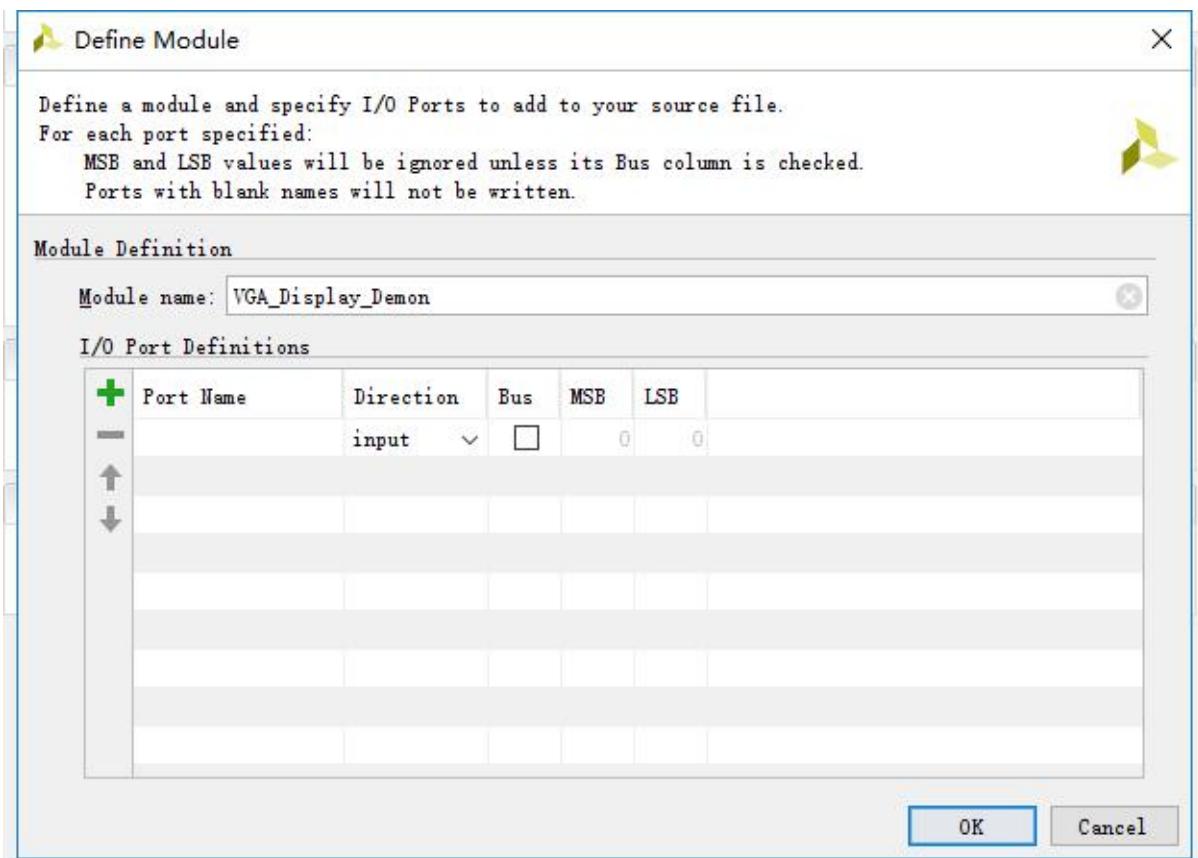
Step6: 创建一个 VGA_Display_Demon 的文件，并且文件类型选择 Verilog



Step7: 添加完成后如下图所示之后单击 finish 完成文件的创建

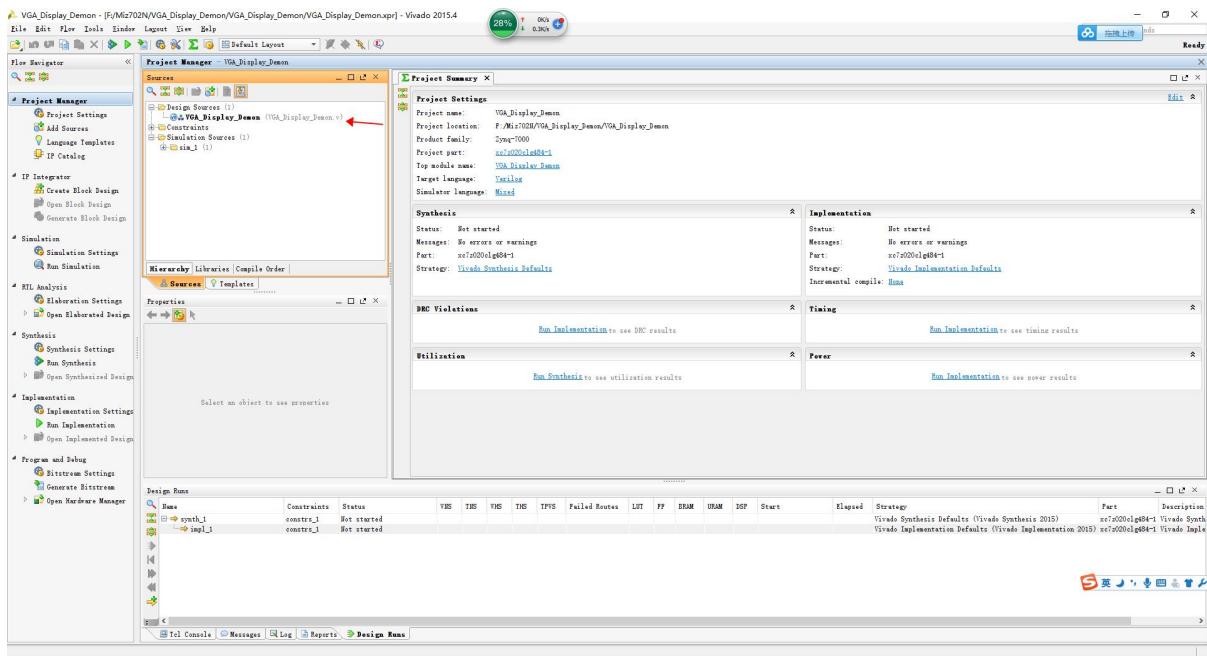


Step8:继续弹出的对话空中，可以设置一些端口，但是我们现在什么都不做。单击OK



Step9:创建完成后可以看到 Design Sources 文件夹中有了 VGA_Display_Demon.v 这

个文件



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

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

```

`timescale 1ns / 1ps

///////////////////////////////
/////
// Company:
// Engineer:
//
// Create Date: 2017/02/22 14:12:56
// Design Name:
// Module Name: VGA_Display_Demon
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:

```

```
//  
// Revision:  
// Revision 0.01 - File Created  
// Additional Comments:  
//  
//////////  
//////  
  
module VGA_Display_Demon(  
);  
endmodule
```

可以看出这是一个空的工程，我们现在要添加代码同时也要添加工程信息。

Step11:编写程序并且添加工程信息

MIZ702 用户添加这段程序

```
module VGA_Display_Demon (  
    input      clk_100M,  
    input      KEY,  
  
    output[3:0] vga_r,  
    output[3:0] vga_g,  
    output[3:0] vga_b,  
    output   vga_hs,  
    output   vga_vs,  
    output [3:0] LED  
);  
  
    wire pixclk;  
    wire[7:0] R, G, B;  
    wire HS, VS, DE;  
    vga_data_gen u_vga_data_gen  
(  
        .pix_clk          (pixclk),  
        .turn_mode        (KEY),  
        .VGA_R            (R),  
        .VGA_G            (G),
```

```

    . VGA_B          (B),
    . VGA_HS         (vga_hs),
    . VGA_VS         (vga_vs),
    . VGA_DE         (DE),
    . mode           (LED)
);
assign vga_r = R[7:4];
assign vga_g = G[7:4];
assign vga_b = B[7:4];

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

```

MIZ702N 用户添加这段程序

```

module VGA_Display_Demon (
    input      clk_100M,
    input      KEY,
    output [4:0] vga_r,
    output [5:0] vga_g,
    output [4:0] vga_b,
    output   vga_hs,
    output   vga_vs,
    output  [3:0] LED
);

wire pixclk;
wire[7:0]  R, G, B;
wire HS, VS, DE;
vga_data_gen u_vga_data_gen
(
    . pix_clk      (pixclk),
    . turn_mode    (KEY),
    . VGA_R        (R),
    . VGA_G        (G),
    . VGA_B        (B),
    . VGA_HS       (vga_hs),
    . VGA_VS       (vga_vs),
    . VGA_DE       (DE),

```

```

    . mode          (LED)
);
assign vga_r = R[7:3];
assign vga_g = G[7:2];
assign vga_b = B[7:3];

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

```

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

```

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

//-----
// 水平扫描参数的设定 1280*720 60HZ
//-----
parameter H_Total      = 1650;
parameter H_Sync        = 40;
parameter H_Back        = 220;
parameter H_Active      = 1280;
parameter H_Front        = 110;
parameter H_Start        = 260;
parameter H_End          = 1540;
//-----
// 垂直扫描参数的设定 1280*720 60HZ
//-----
parameter V_Total      = 750;
parameter V_Sync        = 5;
parameter V_Back        = 20;

```

```
parameter V_Active    =  720;
parameter V_Front     =  5;
parameter V_Start      = 25;
parameter V_End        = 745;
reg[11:0]  x_cnt;
always @(posedge pix_clk)          //水平计数
begin
    if(1'b0)
        x_cnt  <= 1;
    else if(x_cnt==H_Total)
        x_cnt  <= 1;
    else
        x_cnt  <= x_cnt + 1;
end

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

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

reg[11:0]  y_cnt;
always @(posedge pix_clk)
begin
    if(1'b0)
        y_cnt  <= 1;
    else if(y_cnt==V_Total)
        y_cnt  <= 1;
    else if(x_cnt==H_Total)
        y_cnt  <= y_cnt + 1;
end
```

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

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

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

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

reg[23:0] color_bar;
always @(posedge pix_clk)
begin
    if(x_cnt==260)
        color_bar   <= 24'hff0000;
    else if(x_cnt==420)
        color_bar   <= 24'h00ff00;
    else if(x_cnt==580)
        color_bar   <= 24'h0000ff;
```

```
else if(x_cnt==740)
color_bar <= 24'hff00ff;
else if(x_cnt==900)
color_bar <= 24'hffff00;
else if(x_cnt==1060)
color_bar <= 24'h00ffff;
else if(x_cnt==1220)
color_bar <= 24'hffffff;
else if(x_cnt==1380)
color_bar <= 24'h000000;
else
color_bar <= color_bar;
end

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

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

reg[7:0] VGA_R_reg;
reg[7:0] VGA_G_reg;
reg[7:0] VGA_B_reg;
always @(posedge pix_clk)
begin
if(1'b0)
begin
VGA_R_reg<=0;
VGA_G_reg<=0;
VGA_B_reg<=0;
end
```

```
else
    case(dis_mode)
        4' d0:begin
            VGA_R_reg<=0;                                //LCD 显示彩色条
            VGA_G_reg<=0;
            VGA_B_reg<=0;
        end
        4' d1:begin
            VGA_R_reg<=8'b11111111;                      //LCD 显示全白
            VGA_G_reg<=8'b11111111;
            VGA_B_reg<=8'b11111111;
        end
        4' d2:begin
            VGA_R_reg<=8'b11111111;                      //LCD 显示全红
            VGA_G_reg<=0;
            VGA_B_reg<=0;
        end
        4' d3:begin
            VGA_R_reg<=0;                                //LCD 显示全绿
            VGA_G_reg<=8'b11111111;
            VGA_B_reg<=0;
        end
        4' d4:begin
            VGA_R_reg<=0;                                //LCD 显示全蓝
            VGA_G_reg<=0;
            VGA_B_reg<=8'b11111111;
        end
        4' d5:begin
            VGA_R_reg<=grid_data_1;                      // LCD 显示方格 1
            VGA_G_reg<=grid_data_1;
            VGA_B_reg<=grid_data_1;
        end
        4' d6:begin
            VGA_R_reg<=grid_data_2;                      // LCD 显示方格 2
            VGA_G_reg<=grid_data_2;
            VGA_B_reg<=grid_data_2;
        end
        4' d7:begin
            VGA_R_reg<=x_cnt[7:0];                      //LCD 显示水平渐变
            VGA_G_reg<=x_cnt[7:0];
            VGA_B_reg<=x_cnt[7:0];
        end
        4' d8:begin
            VGA_R_reg<=y_cnt[8:1];                      //LCD 显示垂直渐变
        end
    endcase
end
```

色

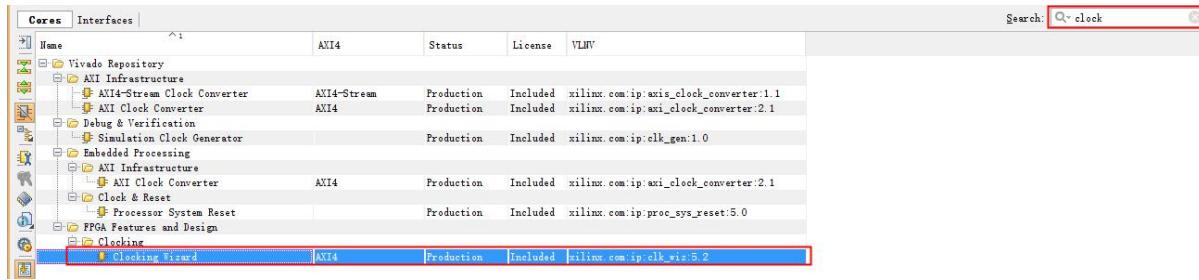
```

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

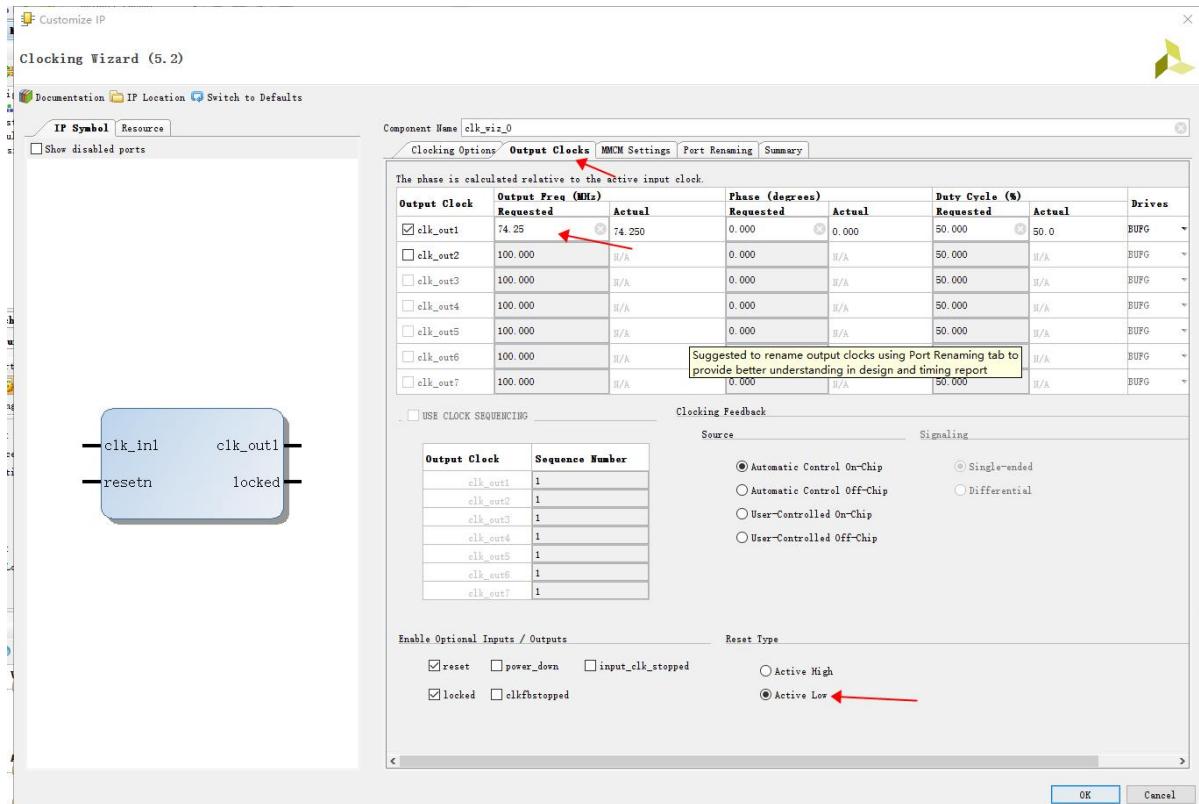
assign VGA_HS = hsync_r;
assign VGA_VS = vsync_r;
assign VGA_DE = hs_de & vs_de;
assign VGA_R = (hs_de & vs_de)?VGA_R_reg:8'h0;
assign VGA_G = (hs_de & vs_de)?VGA_G_reg:8'h0;
assign VGA_B = (hs_de & vs_de)?VGA_B_reg:8'h0;
endmodule

```

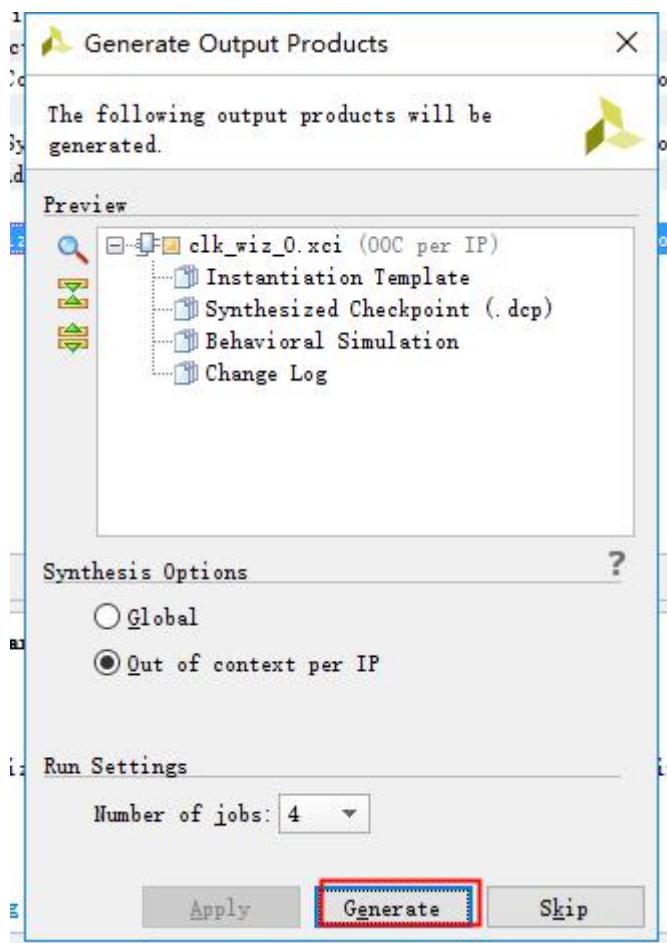
Step14:单击  IP Catalog, 添加一个时钟管理器, 为系统提供时钟。



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



Step16：单击 Generate。

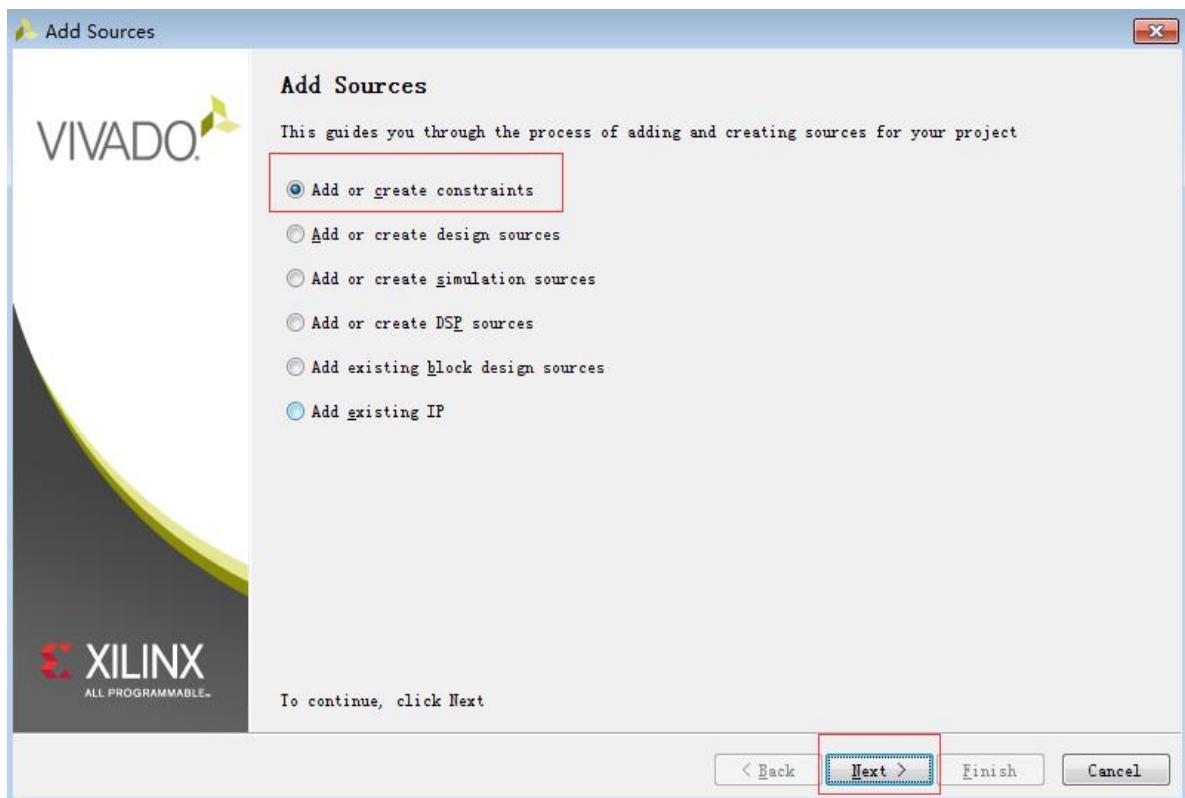


这样我们就编写好了代码下面还要添加管脚约束文件。

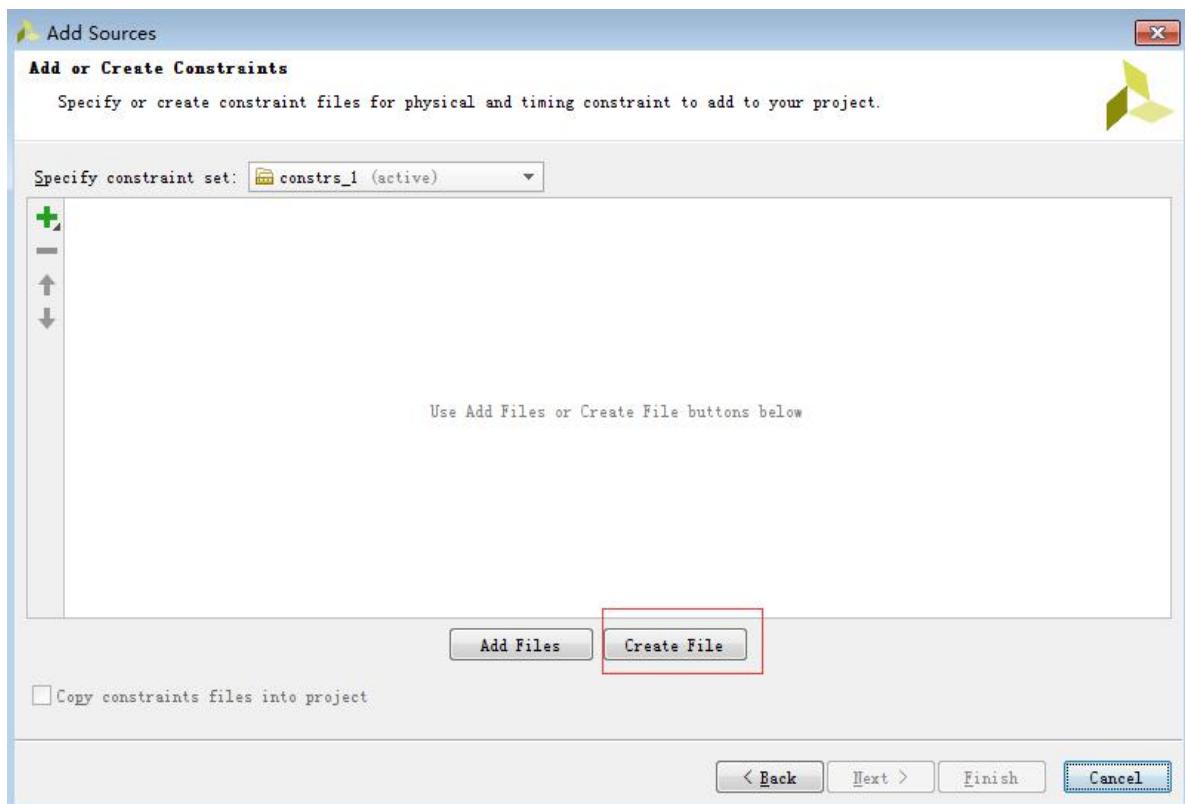
10.5 添加管脚约束文件

Step1: 单击 (和添加.v文件一样)

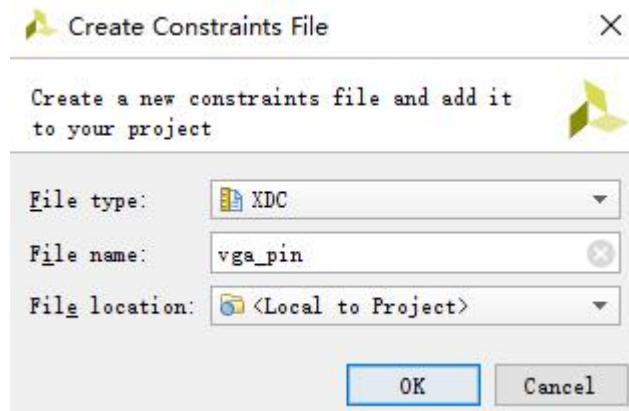
Step2: 选择 Add or create constraints 然后单击 NEXT



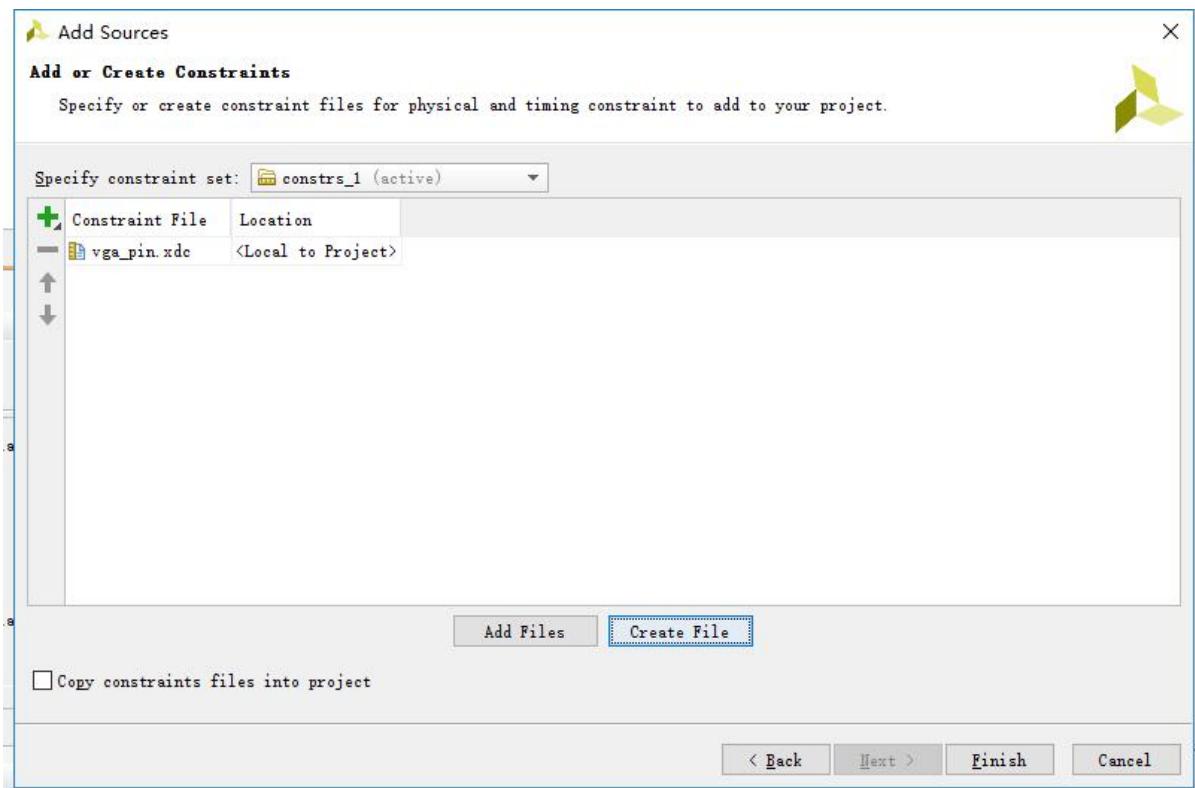
Step3:单击 Create File



Step4:命名为 vga_pin 后单击 OK



Step5:可以看到产生了名为 vga_pin.xdc 的文件然后单击 Finish



Step6:打开 vga_pin.xdc 文件添加如下约束

MIZ702 用户约束:

```
set_property IOSTANDARD LVCMOS33 [get_ports clk_100M]
set_property IOSTANDARD LVCMOS33 [get_ports KEY]
set_property PACKAGE_PIN Y9 [get_ports clk_100M]
set_property PACKAGE_PIN R14 [get_ports KEY]
# "VGA-R1"
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {vga_r[0]}]
set_property PACKAGE_PIN V20 [get_ports {vga_r[0]}]
# "VGA-R2"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_r[1]}]
set_property PACKAGE_PIN U20 [get_ports {vga_r[1]}]
#"VGA-R3"
set_property PACKAGE_PIN V19 [get_ports {vga_r[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_r[2]}]
# "VGA-R4"
set_property PACKAGE_PIN V18 [get_ports {vga_r[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_r[3]}]
# "VGA-G1"
set_property PACKAGE_PIN AB22 [get_ports {vga_g[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[0]}]
# "VGA-G2"
set_property PACKAGE_PIN AA22 [get_ports {vga_g[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[1]}]
# "VGA-G3"
set_property PACKAGE_PIN AB21 [get_ports {vga_g[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[2]}]
# "VGA-G4"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[3]}]
set_property PACKAGE_PIN AA21 [get_ports {vga_g[3]}]
# "VGA-B1"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_b[0]}]
set_property PACKAGE_PIN Y21 [get_ports {vga_b[0]}]
# "VGA-B2"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_b[1]}]
set_property PACKAGE_PIN Y20 [get_ports {vga_b[1]}]
# "VGA-B3"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_b[2]}]
set_property PACKAGE_PIN AB20 [get_ports {vga_b[2]}]
# "VGA-B4"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_b[3]}]
```

```

set_property PACKAGE_PIN AB19 [get_ports {vga_b[3]}]
# "VGA-VS"
set_property IOSTANDARD LVCMOS33 [get_ports vga_vs]
set_property PACKAGE_PIN AA19 [get_ports vga_vs]
# "VGA-HS"
set_property PACKAGE_PIN Y19 [get_ports vga_hs]
set_property IOSTANDARD LVCMOS33 [get_ports vga_hs]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
set_property PACKAGE_PIN T22 [get_ports {LED[0]}]
set_property PACKAGE_PIN T21 [get_ports {LED[1]}]
set_property PACKAGE_PIN U22 [get_ports {LED[2]}]
set_property PACKAGE_PIN U21 [get_ports {LED[3]}]

```

MIZ702N 用户约束:

```

set_property IOSTANDARD LVCMOS33 [get_ports clk_100M]
set_property IOSTANDARD LVCMOS33 [get_ports KEY]
set_property PACKAGE_PIN M19 [get_ports clk_100M]
set_property PACKAGE_PIN AA19 [get_ports KEY]
# "VGA-R1"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_r[0]}]
set_property PACKAGE_PIN T6 [get_ports {vga_r[0]}]
# "VGA-R2"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_r[1]}]
set_property PACKAGE_PIN R6 [get_ports {vga_r[1]}]
#"VGA-R3"
set_property PACKAGE_PIN U5 [get_ports {vga_r[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_r[2]}]
# "VGA-R4"
set_property PACKAGE_PIN U6 [get_ports {vga_r[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_r[3]}]

```

```
# "VGA-R5"
set_property PACKAGE_PIN W6 [get_ports {vga_r[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_r[4]}]
# "VGA-G1"
set_property PACKAGE_PIN AB11 [get_ports {vga_g[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[0]}]
# "VGA-G2"
set_property PACKAGE_PIN AA11 [get_ports {vga_g[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[1]}]
# "VGA-G3"
set_property PACKAGE_PIN AB12 [get_ports {vga_g[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[2]}]
# "VGA-G4"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[3]}]
set_property PACKAGE_PIN AA12 [get_ports {vga_g[3]}]
# "VGA-G5"
set_property PACKAGE_PIN T4 [get_ports {vga_g[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[4]}]
# "VGA-G6"
set_property PACKAGE_PIN U4 [get_ports {vga_g[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {vga_g[5]}]
# "VGA-B1"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_b[0]}]
set_property PACKAGE_PIN AA9 [get_ports {vga_b[0]}]
# "VGA-B2"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_b[1]}]
set_property PACKAGE_PIN AB7 [get_ports {vga_b[1]}]
# "VGA-B3"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_b[2]}]
set_property PACKAGE_PIN AB10 [get_ports {vga_b[2]}]
# "VGA-B4"
set_property IOSTANDARD LVCMOS33 [get_ports {vga_b[3]}]
set_property PACKAGE_PIN AA8 [get_ports {vga_b[3]}]
# "VGA-B5"
```

```

set_property IOSTANDARD LVCMOS33 [get_ports {vga_b[4]}]
set_property PACKAGE_PIN AB9 [get_ports {vga_b[4]}]
# "VGA-VS"

set_property IOSTANDARD LVCMOS33 [get_ports vga_vs]
set_property PACKAGE_PIN W5 [get_ports vga_vs]
# "VGA-HS"

set_property PACKAGE_PIN Y6 [get_ports vga_hs]
set_property IOSTANDARD LVCMOS33 [get_ports vga_hs]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
set_property PACKAGE_PIN AA14 [get_ports {LED[0]}]
set_property PACKAGE_PIN W15 [get_ports {LED[1]}]
set_property PACKAGE_PIN Y15 [get_ports {LED[2]}]
set_property PACKAGE_PIN AB14 [get_ports {LED[3]}]

```

10.6 编译并且产生 bit 文件

Step1:单击综合

Step2:单击执行

Step3:单击产生 bit



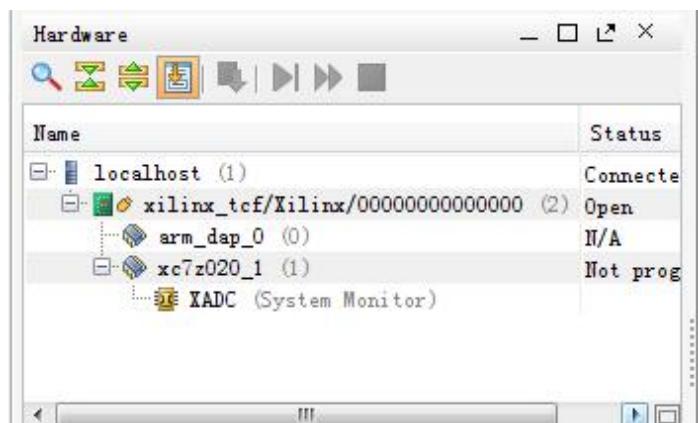
10.7 下载程序

Step1:给开发板通电，并且连接下载器

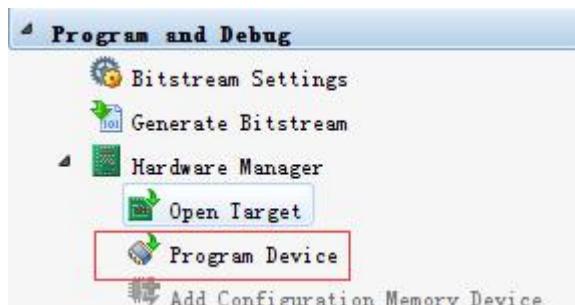
Step2:单击 OpenTarget 然后单击 Auto Connect



Step3:连接成功后



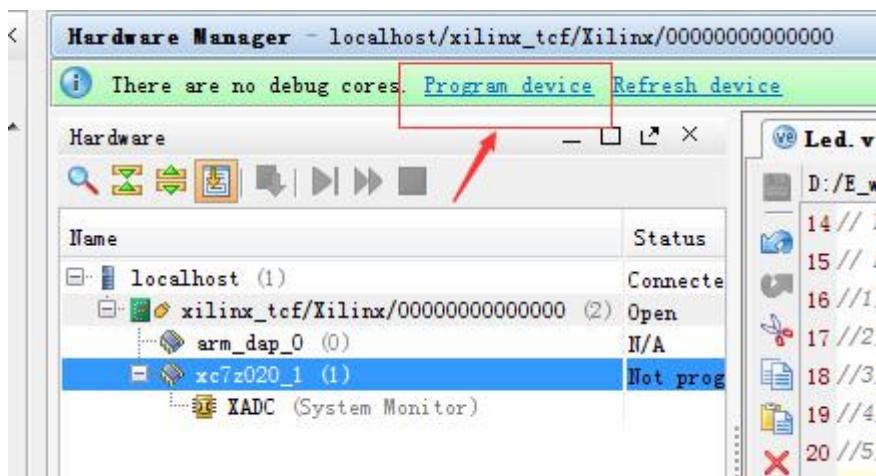
Step4:单击 Program Device



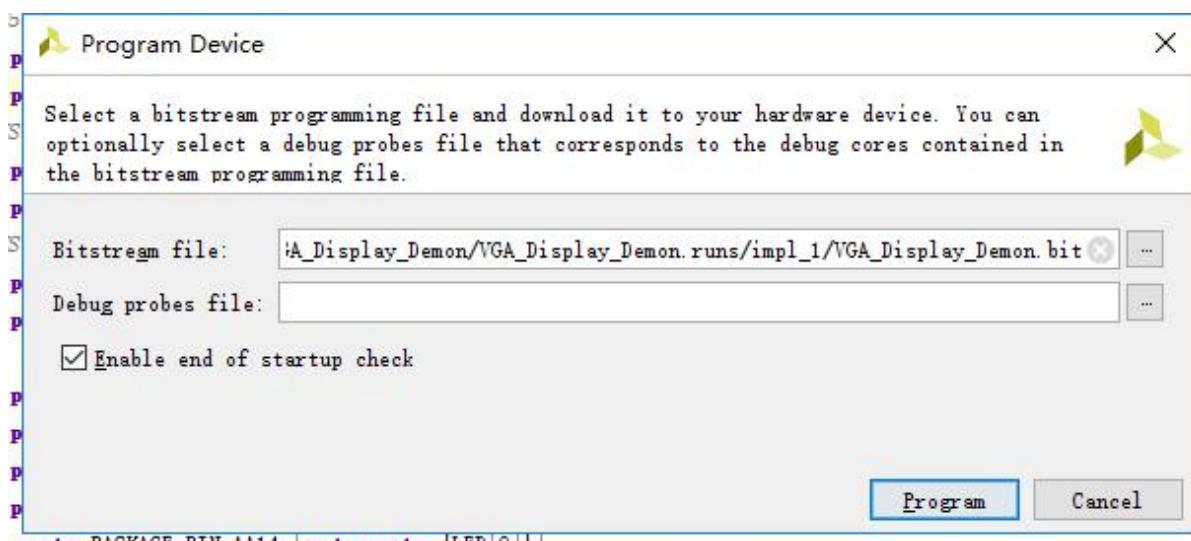
Step5:单击 Program Device 然后选择 XC7Z020_1



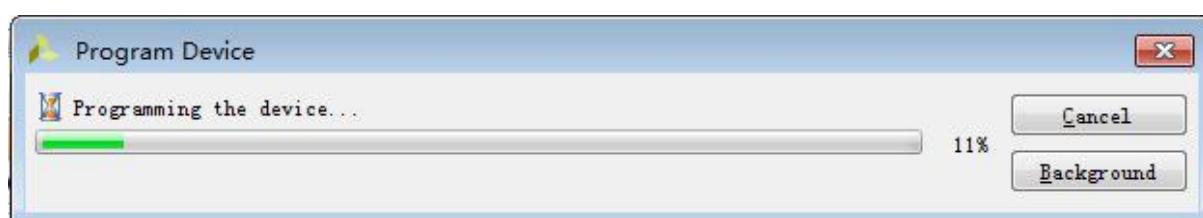
Step6:或者也可以从顶部单击 Program device



Step7:弹出的对话框中有我们要下载的Bit文件

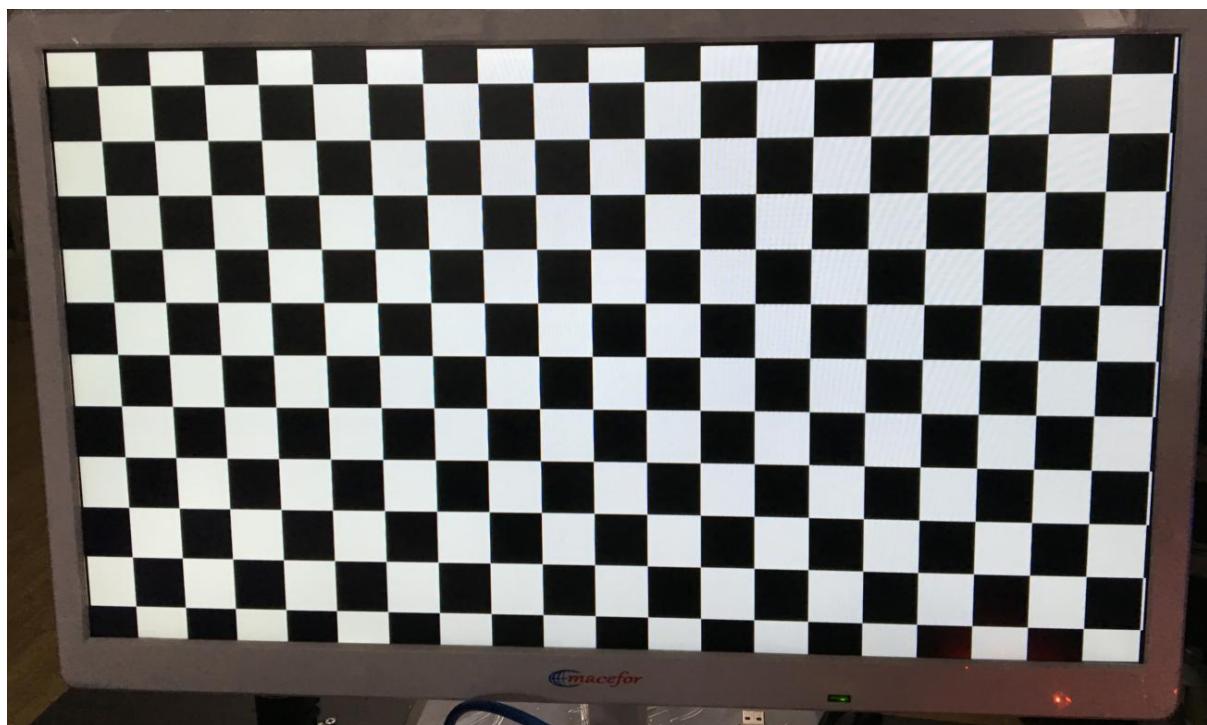


Step7:下载过程



10.8 实验结果

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



10.9 本章小结

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

S01_CH11_ADV7511 HDMI 接口测试

HDMI 既高清晰度多媒体接口 (High Definition Multimedia Interface)，这是一种数字视频/音频接口，相比较前面介绍的 VGA 接口，它传输的信息量大，色彩度高，传输速度快等显著优点。我们的 MIZ702 和 MIZ702 N 上面采用了一颗专用的 HDMI 芯片 ADV7511 做 HDMI 输出使用，这章就将带领大家使用我们封装好的 IP 核来设计一个 HDMI 接口的测试实验。MIZ701N 用户请直接翻到下一章。

11.1 ADV7511 概述

ADV7511 是一款高速高清晰度多媒体接口 (High Definition Multimedia Interface, HDMI) 发送器。能够处理的数据速率高达 165MHz (1080p @60Hz, UXGA @60Hz)，输出数据速率高达 225MHz。像素数据位宽最多为 36 位，为了保证 HDMI 兼容性，在进行硬件设计（原理图和 PCB 布线）时，需要特别小心。

11.1.1 硬件特性

- 支持 HDMI v1.4 特性
 - HEAC (ARC)
 - 3D 视频
 - 高级
 - ◆ Sycc601
 - ◆ Adobe RGB
 - ◆ Adobe YCC601
- 支持深度颜色
- 操作速率高达 225MHz (TMDS 链接频率)
- 集成 CEC，支持 3 消息缓存
- 支持 x.v.Color (Gamut Metadata)
- 内部存储 HDCP 秘钥
- 中断输出针脚
- 支持 I2S, SPDIF, DSD, DS 和 HBR 音频输入格式
- SPDIF 不需要音频主设备时钟 (MCLK)
- 需要 1.8V 和 3.3V 供电
- 片载 EDID 缓存
- 颜色空间转换 (Color Space Converter, CSC)
- 100 LQFP 封装
- 工作温度：0° ~ +70°

11.1.2 视频输入

ADV7511 可以接受最低 8 位 (YCbCr 4:2:2 双倍数据速率 [DDR] 或双倍像素时钟采样

的 YCbCr 4:2:2)，最高 36 位 (RGB 4:4:4 或 YCbCr 4:4:4) 的视频数据。除了数据外，还需要行同步 hsync、场同步 vsync 以及数据有效信号 de。ADV7511 还可以检测到 EIA/CEA-861D 协议定义的 59 种视频格式。

所支持的视频格式有：

- 36, 30 或 24 位 RGB 4:4:4 (独立同步)
- 36, 30 或 24 位 YCbCr 4:4:4 (独立同步)
- 24, 20 或 16 位 YCbCr 4:2:2 (内嵌或独立同步)
- 12, 10 或 8 位 YCbCr 4:2:2 (2 倍像素时钟, 内嵌或独立同步)
- 12, 10 或 8 位 YCbCr 4:2:2 (DDR, 内嵌或独立同步)
- 12 位 RGB 4:4:4 (DDR, 独立同步)
- 12 位 YCbCr 4:4:4 (DDR, 独立同步)

举个简单的例子，方便读者对 ADV7511 手册的理解。

Table 5 Normal RGB or YCbCr 4:4:4 (36, 30, or 24 bits) with Separate Syncs; Input ID = 0

Mode	Format	Input Data D[35:0]																																			
		35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
36 bit	RGB	R[11:0]																G[11:0]									B[11:0]										
	YCrCb	Cr[11:0]																Y[11:0]									Cb[11:0]										
30 bit	RGB	R[9:0]																G[9:0]									B[9:0]										
	YCrCb	Cr[9:0]																Y[9:0]									Cb[9:0]										
24 bit	RGB	R[7:0]																G[7:0]									B[7:0]										
	YCrCb	Cr[7:0]																Y[7:0]									Cb[7:0]										
Pins D[35:0]		35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

An input format of RGB 4:4:4 or YCbCr 4:4:4 can be selected by setting the input ID (R0x15 [3:0]) to 0x0. There is no need to set the Input Style (R0x16[3:2]) or channel alignment (R0x48[4:3]). For timing details see the > ADV7511 Hardware User's Guide.

灰色部分表示未使用的位。

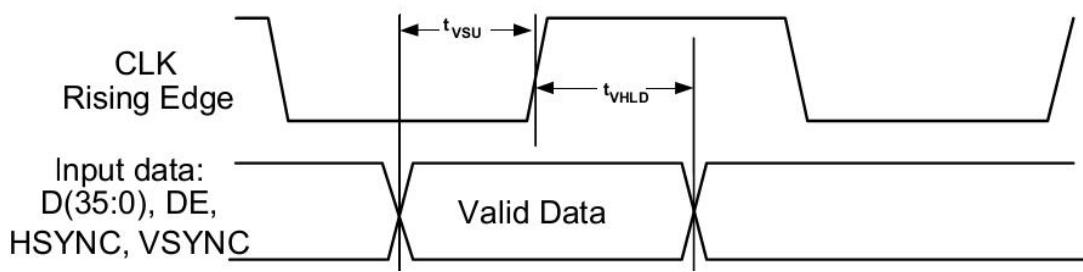
从上图中的文字可以知道，当设置寄存器 R0x15[3:0] 为 0x0 时，指定的视频输入格式为 RGB4:4:4 或 YCbCr4:4:4。没有必要设置输入的样式 (R0x16[3:2]) 或者通道的对齐属性 (R0x48[4:3])。

11.1.3 支持的输出格式

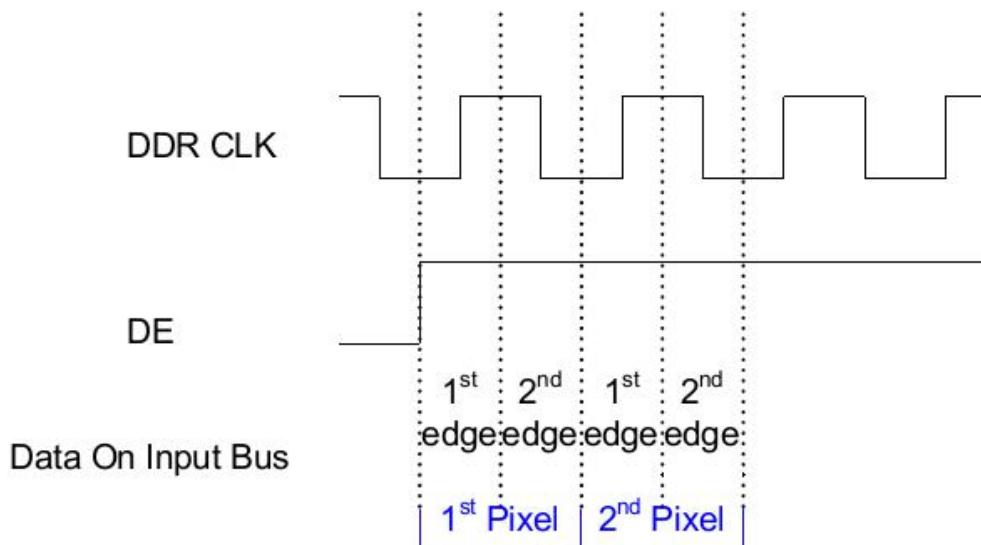
- 36, 30 或 24 位 RGB 4:4:4
- 36, 30 或 24 位 YCbCr 4:4:4
- 24 位 YCbCr 4:2:2

11.1.4 视频接口信号采样

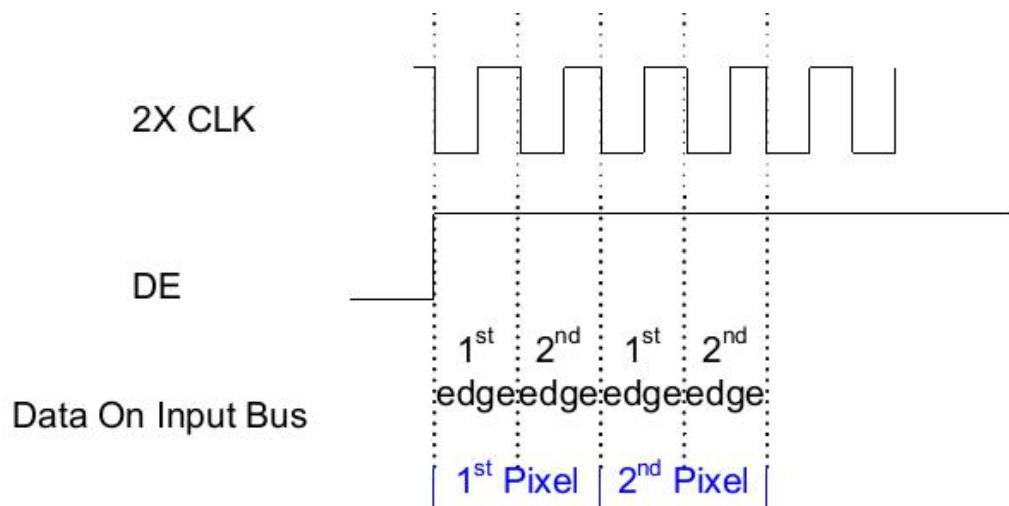
如下图所示，对于单倍速数据而言，在时钟的上升沿对数据和视频控制信号进行采样。



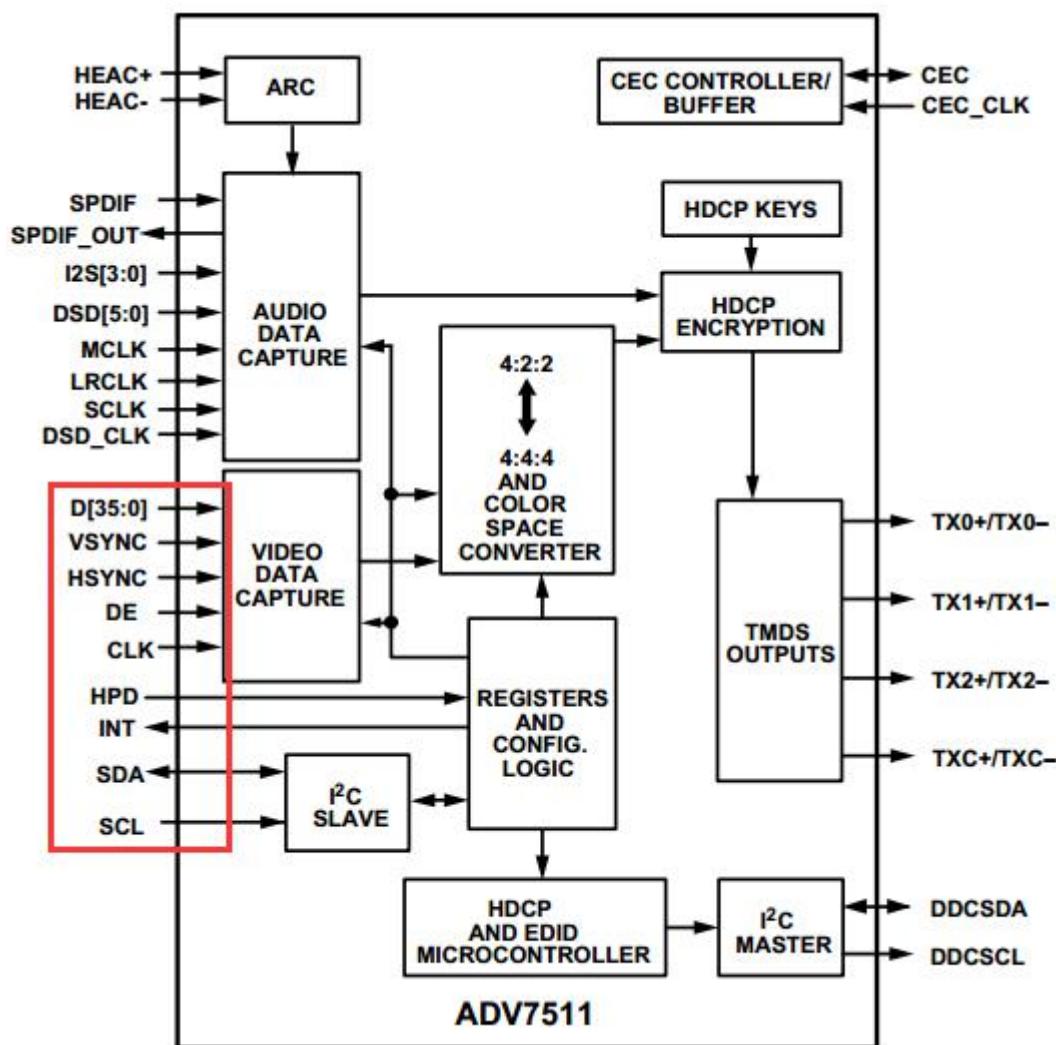
对于双倍速数据[DDR]而言，在时钟的上升沿和下降沿各进行一次采样。如下图所示。



双倍时钟采样方式和 DDR 不尽相同，是在第一个时钟上升沿采集第一个像素的一半数据，第二个时钟上升沿采集第一个像素的另一半数据，两个时钟周期采集到的数据组合起来称为一个像素的数据。



11.1.5 功能框图



ADV7511不仅能够进行视频显示，还具有音频通道，但本章的重点是视频显示，所以不考虑音频通道，这样以来，需要控制的信号数量就大大减少，只有上图红色方框内的信号，而且中断和HPD信号并非必须信号。

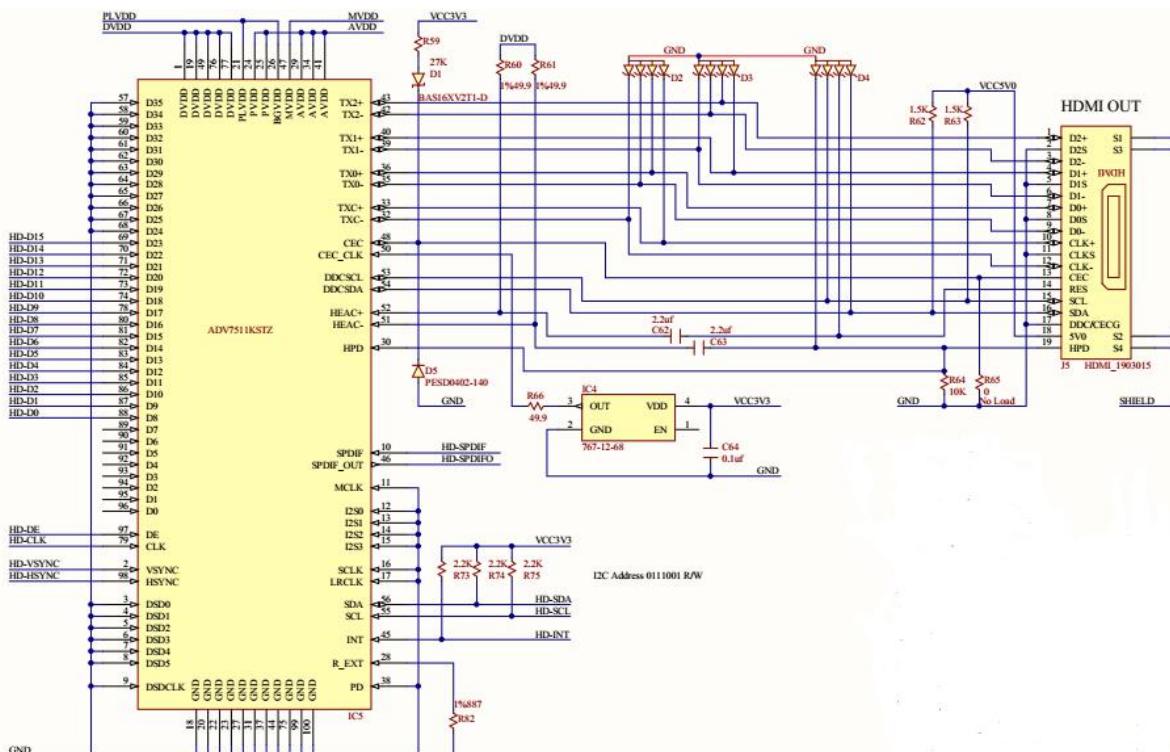
11.1.6 寄存器空间

ADV7511有很多寄存器，本章不能一一详述，仅介绍快速开始向导提及的寄存器，其他寄存器请自行阅读《ADV7511 Programming Guide》。

Tx 上电(HPD 必须拉高)	0x41[6]=0b0
上电时 必须配置的固定寄存器	<ol style="list-style-type: none"> 1. 0x98 = 0x03 2. 0x9A[7:5] = 0b111 3. 0x9C = 0x30 4. 0x9D[1:0] = 0b01 5. 0xA2 = 0xA4 6. 0xA3 = 0xA4

	7. 0xE0[7:0] = 0xD0 8. 0xF9[7:0] = 0x00
设定视频输入格式	1. 0x15[3:0] - 视频格式 ID (默认 4:4:4) 2. 0x16[5:4] - 4:2:2 颜色数据位宽 (默认 12 位) 3. 0x16[3:2] - 视频输入格式 (默认 格式 2) 4. 0x17[1] - 输入视频比例 (0b0=4:3; 0b1=16:9)
设定视频输出格式	1. 0x16[7:6] - 输出格式, 0b0=4:4:4 vs 4:2:2 2. 0x18[7] - CSC 使能, 0b1=YCbCr to RGB 3. 0x18[6:5] - CSC 缩放因子, 0b0=YCbCr to RGB 4. 0xAF[1] - 手动 HDMI/DVI 模式选择, 0b1=HDMI 5. 0x40[7] - 使能 GC(0b1) 6. 0x4C[3:0] - 输出颜色位宽, 通用控制颜色位宽 (GC CD)
HDCP	1. 0xAF[7] = 0xb1 - 使能 HDCP 2. 0x97[6] - BKSV 中断标志 (等待该值置位然后写 0b1)

11.2 硬件电路分析



根据官方手册的建议, 将 ADV7511 未使用到的引脚全部接地。左侧信号为视频信号, 包括 16 位数据, 数据有效, 时钟, 行、场同步信号; 右侧有音频接口信号 HD-SPDIF、HD-SPDIFO, IIC 总线控制接口, 中断信号。结合 ADV7511 支持的视频输入格式可以知道, MIZ702 和 MIZ702N 的 HDMI 接口仅支持 16 位 YcbCr 422 数据, 格式如下表所示。

模式	像素	输入数据																
		35~24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8

格式 1																		
16 位	1 st		Cb[7:0]								Y[7:0]							
	2 nd		Cr[7:0]								Y[7:0]							
格式 2、格式 3																		
16 位	1 st		Y[7:0]								Cb[7:0]							
	2 nd		Y[7:0]								Cr[7:0]							
Pins D[35:0]	35~24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7~0

11.3 创建工程文件

本章的工程与上一章 VGA 的比较相似，在设计中，使用的也是 VGA 的图像生成模块，因此，为了节省开发时间可在上一章的基础上继续完善。

Step1：直接打开上一章的工程。

Step2：双击 VGA_Display_Demon.v，将代码用如下程序替换。

```
module VGA_Display_Demon(
    input      clk_100M,
    input      KEY,
    output     hdmi_clk,
    output     hdmi_hsync,
    output     hdmi_vsync,
    output [15:0] hdmi_d,
    output     hdmi_de,
    output     hdmi_scl,
    inout    hdmi_sda,
    output   [3:0] LED
);

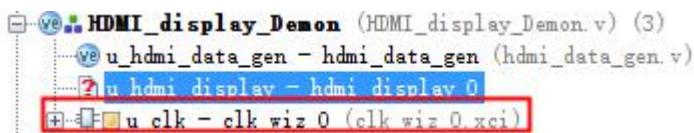
wire pixclk;
wire[7:0] R, G, B;
wire HS, VS, DE;
hdmi_data_gen u_hdmi_data_gen
(
    .pix_clk          (pixclk),
    .turn_mode        (KEY),
    .VGA_R            (R),
    .VGA_G            (G),
    .VGA_B            (B),
    .VGA_HS           (HS),
    .VGA_VS           (VS),
    .VGA_DE          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

```

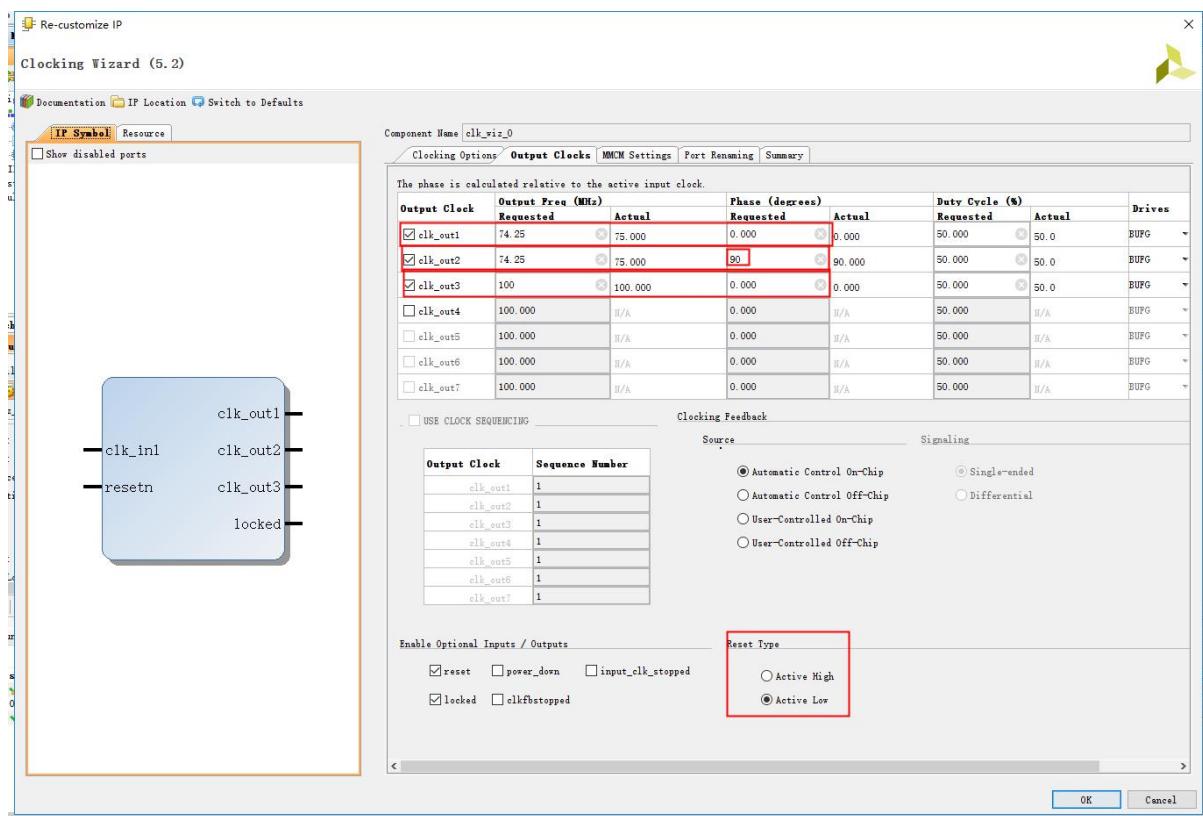
wire pixclk_90;
wire i2c_clk;
wire lock;
wire[23:0] RGB;
assign RGB={R, G, B} ;
hdmi_display_0 u_hdmi_display
(
    .i2c_clk          (i2c_clk),
    .vga_clk          (pixclk),
    .vga_clk_90       (pixclk_90),
    .rgb_in           (RGB),
    .hsync_in         (HS),
    .vsync_in         (VS),
    .de_in            (DE),
    .hdmi_clk         (hdmi_clk),
    .hdmi_hsync       (hdmi_hsync),
    .hdmi_vsync       (hdmi_vsync),
    .hdmi_d           (hdmi_d),
    .hdmi_de          (hdmi_de),
    .hdmi_scl          (hdmi_scl),
    .hdmi_sda          (hdmi_sda)
);
clk_wiz_0 u_clk
(
    .clk_in1          (clk_100M),
    .resetn           (1'b1),
    .clk_out1         (pixclk),
    .clk_out2         (pixclk_90),
    .clk_out3         (i2c_clk),
    .locked           (lock)
);
endmodule

```

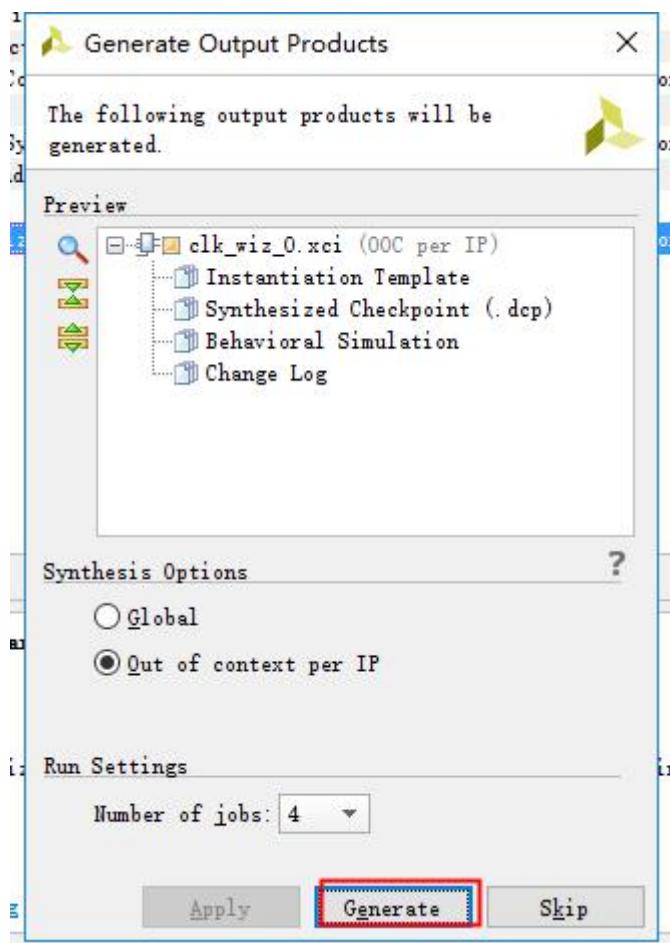
Step3: 双击 u_clk - clk_wiz_0 (clk_wiz_0.xci) , 对其重新配置。



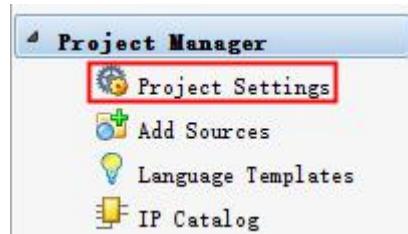
Step4: 在 output clocks 选项卡下如下图设置，其余参数默认配置即可，然后单击 OK。



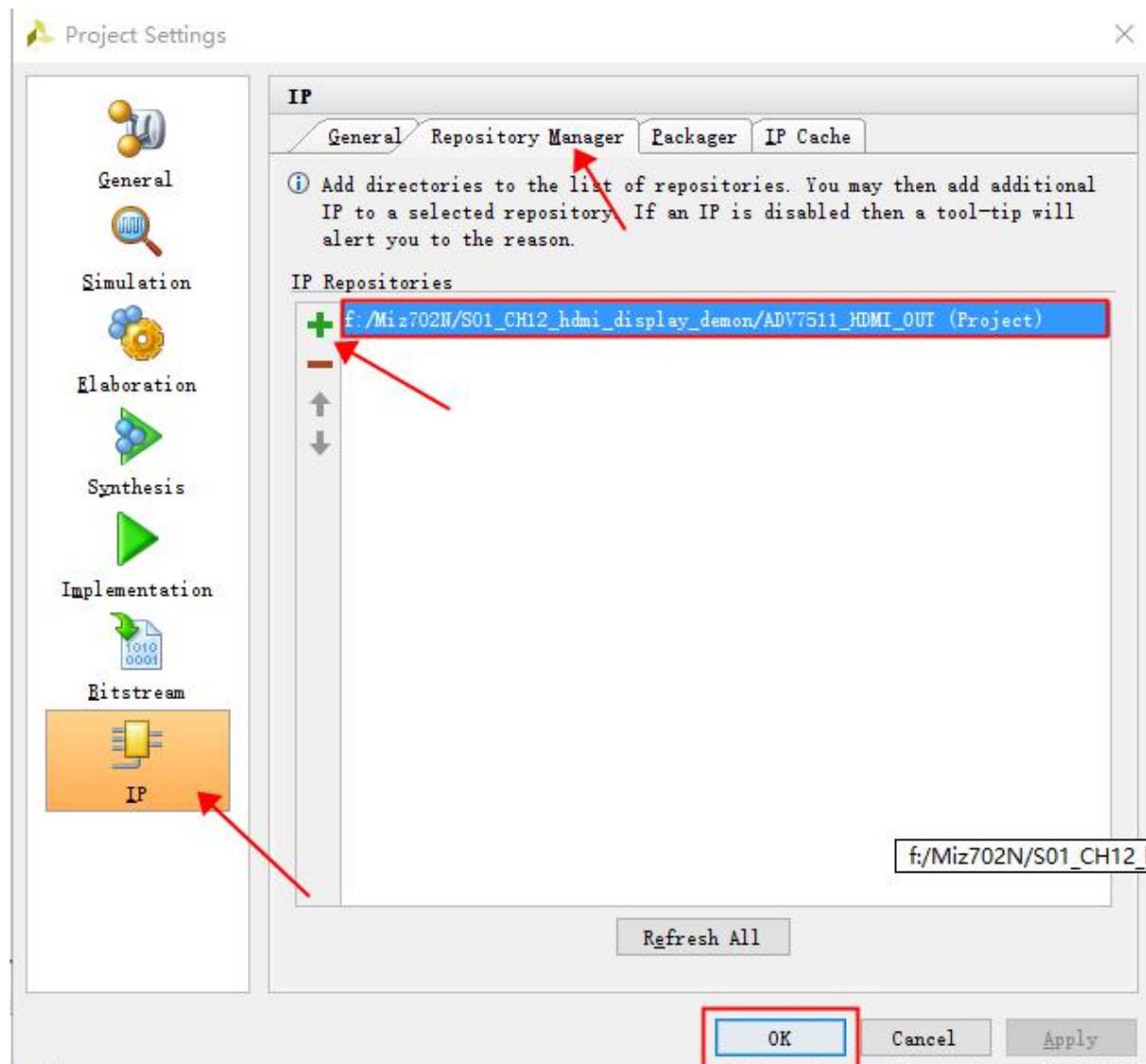
Step5：单击 Generate。



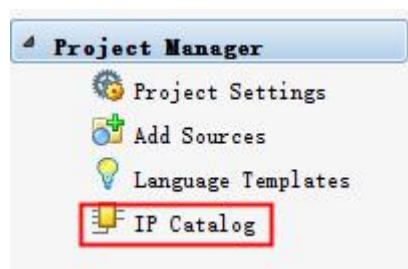
Step6：单击 Project settings,对工程进行设置。



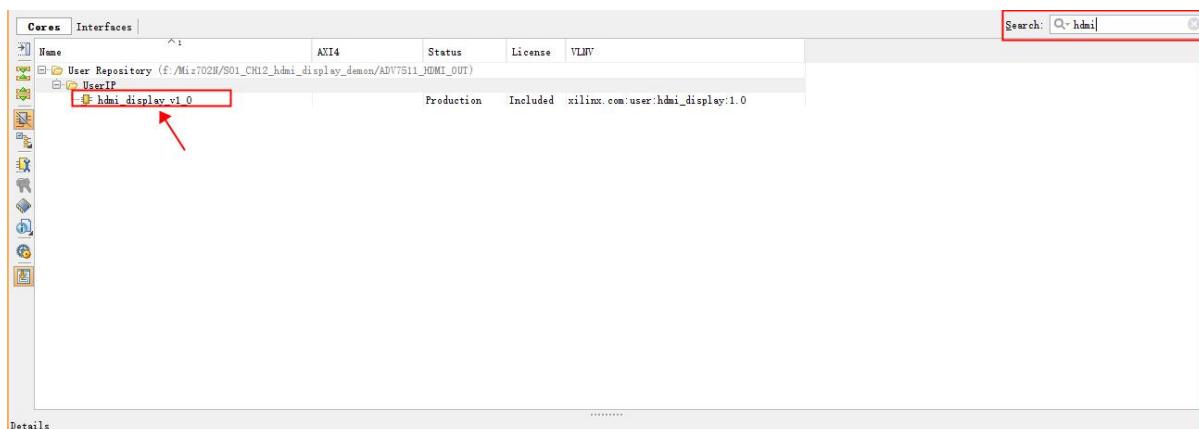
Step7：选择 IP 选项，再选择 Repository Manager，单击里面的+号将我们的 ADV7511 的 IP 核的路径添加进来，最后单击 OK 完成添加（IP 核在我们提供的源代码对应章节的文件包里面的 Miz_ip_lib 文件夹中找到）。



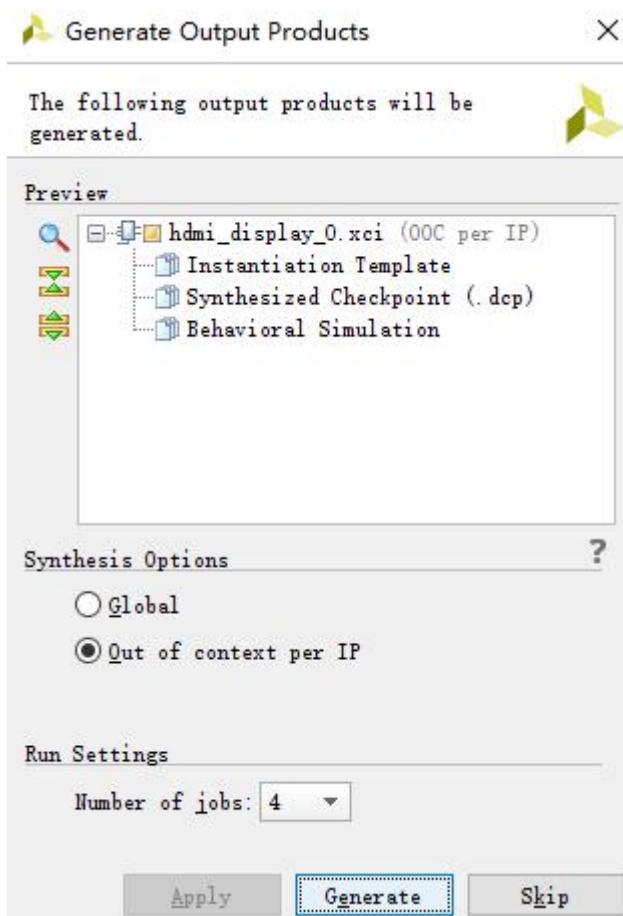
Step8：单击 IP Catalog 添加一个 IP。



Step9：输入 IP 的名字，双击之后单击 OK 不做任何改动，将其添加到工程当中。



Step10：单击 Generate。



这样我们就完成了工程的修改下面还要添加管脚约束文件。

11.4 添加管脚约束文件

Step1: 打开 vga_pin.xdc 文件, 用下面给出的约束文件替换原来的约束文件。

MIZ702 用户约束:

```
set_property -dict {PACKAGE_PIN W18 IOSTANDARD LVCMOS33} [get_ports hdmi_clk]
set_property -dict {PACKAGE_PIN W17 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports hdmi_vsync]
set_property -dict {PACKAGE_PIN V17 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports hdmi_hsync]
set_property -dict {PACKAGE_PIN U16 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports hdmi_de]
set_property -dict {PACKAGE_PIN Y13 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[0]}]
set_property -dict {PACKAGE_PIN AA13 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[1]}]
set_property -dict {PACKAGE_PIN AA14 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[2]}]
set_property -dict {PACKAGE_PIN Y14 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[3]}]
set_property -dict {PACKAGE_PIN AB15 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[4]}]
set_property -dict {PACKAGE_PIN AB16 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[5]}]
set_property -dict {PACKAGE_PIN AA16 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[6]}]
set_property -dict {PACKAGE_PIN AB17 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[7]}]
set_property -dict {PACKAGE_PIN AA17 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[8]}]
set_property -dict {PACKAGE_PIN Y15 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[9]}]
set_property -dict {PACKAGE_PIN W13 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[10]}]
set_property -dict {PACKAGE_PIN W15 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[11]}]
set_property -dict {PACKAGE_PIN V15 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[12]}]
set_property -dict {PACKAGE_PIN U17 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[13]}]
set_property -dict {PACKAGE_PIN V14 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[14]}]
set_property -dict {PACKAGE_PIN V13 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[15]}]

set_property -dict {PACKAGE_PIN AA18 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
```

```
hdmi_scl]
set_property -dict {PACKAGE_PIN Y16 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports hdmi_sda]

set_property PULLUP true [get_ports hdmi_sda]

set_property IOSTANDARD LVCMOS33 [get_ports clk_100M]
set_property PACKAGE_PIN Y9 [get_ports clk_100M]
set_property IOSTANDARD LVCMOS33 [get_ports KEY]
set_property PACKAGE_PIN R14 [get_ports KEY]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
set_property PACKAGE_PIN T22 [get_ports {LED[3]}]
set_property PACKAGE_PIN T21 [get_ports {LED[2]}]
set_property PACKAGE_PIN U22 [get_ports {LED[1]}]
set_property PACKAGE_PIN U21 [get_ports {LED[0]}]
```

MIZ702N 用户约束:

```
set_property -dict {PACKAGE_PIN Y5 IOSTANDARD LVCMOS33} [get_ports hdmi_clk]
set_property -dict {PACKAGE_PIN AA6 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports hdmi_vsync]
set_property -dict {PACKAGE_PIN AA7 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports hdmi_hsync]
set_property -dict {PACKAGE_PIN W7 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports hdmi_de]
set_property -dict {PACKAGE_PIN V8 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[0]}]
set_property -dict {PACKAGE_PIN W8 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[1]}]
set_property -dict {PACKAGE_PIN U10 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[2]}]
set_property -dict {PACKAGE_PIN U9 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[3]}]
set_property -dict {PACKAGE_PIN V10 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports {hdmi_d[4]}]
set_property -dict {PACKAGE_PIN V9 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
```

```
{hdmi_d[5]}

set_property -dict {PACKAGE_PIN Y9 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[6]}]

set_property -dict {PACKAGE_PIN Y8 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[7]}]

set_property -dict {PACKAGE_PIN V12 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[8]}]

set_property -dict {PACKAGE_PIN W12 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[9]}]

set_property -dict {PACKAGE_PIN U12 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[10]}]

set_property -dict {PACKAGE_PIN U11 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[11]}]

set_property -dict {PACKAGE_PIN Y11 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[12]}]

set_property -dict {PACKAGE_PIN Y10 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[13]}]

set_property -dict {PACKAGE_PIN W11 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[14]}]

set_property -dict {PACKAGE_PIN W10 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports
{hdmi_d[15]}]

set_property -dict {PACKAGE_PIN W13 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports hdmi_scl]
set_property -dict {PACKAGE_PIN V13 IOSTANDARD LVCMOS33 IOB TRUE} [get_ports hdmi_sda]

set_property PULLUP true [get_ports hdmi_sda]

set_property IOSTANDARD LVCMOS33 [get_ports clk_100M]
set_property PACKAGE_PIN M19 [get_ports clk_100M]
set_property IOSTANDARD LVCMOS33 [get_ports KEY]
set_property PACKAGE_PIN AA19 [get_ports KEY]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
set_property PACKAGE_PIN AA14 [get_ports {LED[3]}]
set_property PACKAGE_PIN W15 [get_ports {LED[2]}]
set_property PACKAGE_PIN Y15 [get_ports {LED[1]}]
```

```
set_property PACKAGE_PIN AB14 [get_ports {LED[0]}]
```

11.5 编译并且产生 bit 文件

Step1:单击综合

Step2:单击执行

Step3:单击产生 bit



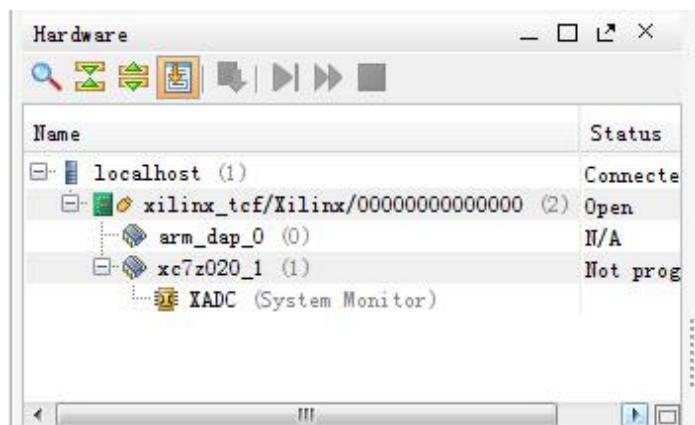
11.6 下载程序

Step1:给开发板通电，并且连接下载器

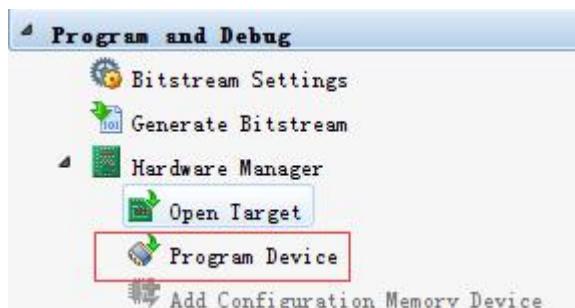
Step2:单击 OpenTarget 然后单击 Auto Connect



Step3:连接成功后



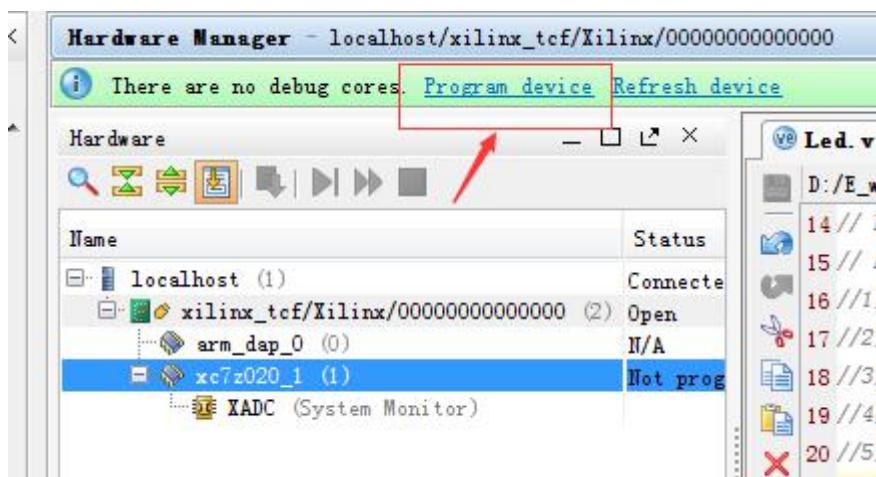
Step4:单击 Program Device



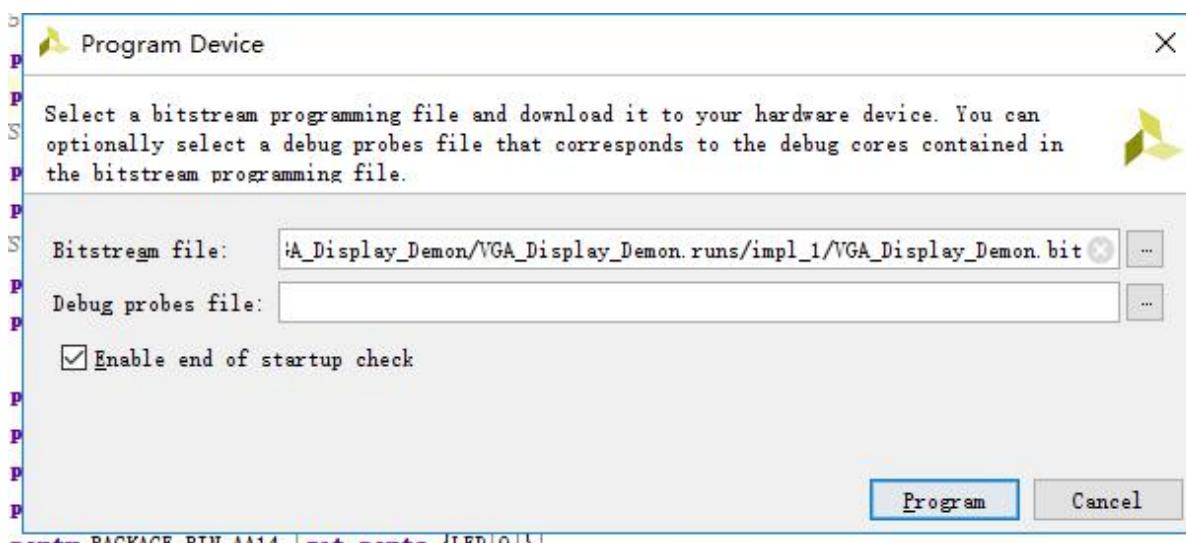
Step5:单击 Program Device 然后选择 XC7Z020_1



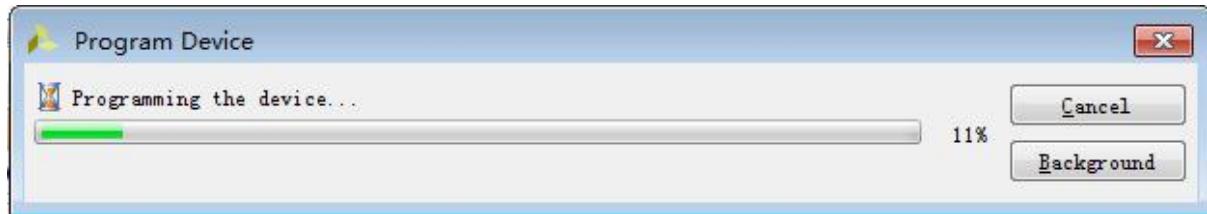
Step6:或者也可以从顶部单击 Program device



Step7:弹出的对话框中有我们要下载的 Bit 文件

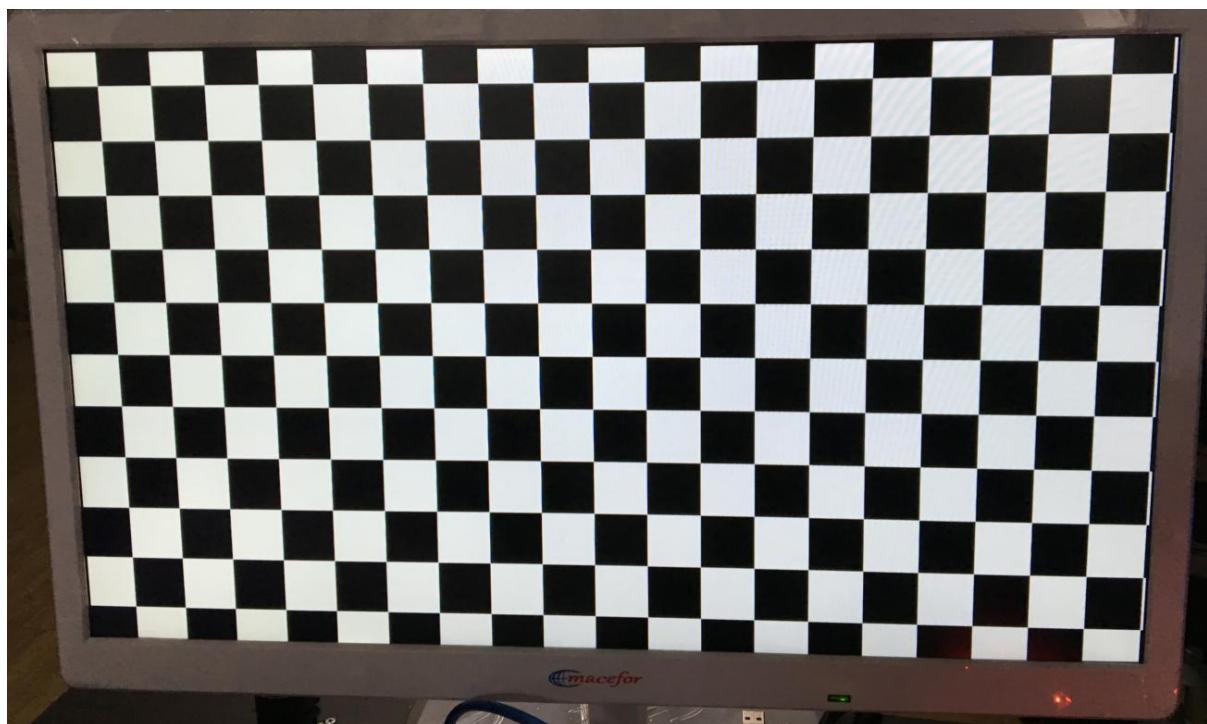


Step7: 下载过程



11.7 实验结果

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



S01_CH12_PL IO 口模拟 HDMI 接口测试

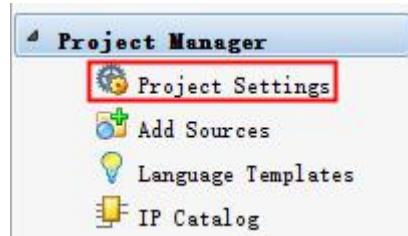
上一章我们介绍了 MIZ702 和 MIZ702N 的 HDMI 接口的测试，这章我们来介绍 MIZ701N 开发板的 HDMI 测试。与 MIZ702 不同的是，MIZ701N 采用的是使用 IO 口模拟 HDMI 时序的方法，来实现的 HDMI 接口的功能，与使用专用芯片实现相比，此方案节省了硬件成本，在芯片资源丰富的情况下，采用这种方法明显更为实用，而且显示效果也比较满意，最大输出分辨率也能达到 1080P。本章也是使用了一个 HDMI 的 IP 来实现功能的，因此可以在上一章的基础上进行修改，得到我们的结果。接下来我们直接步入正题。

12.1 创建工程文件

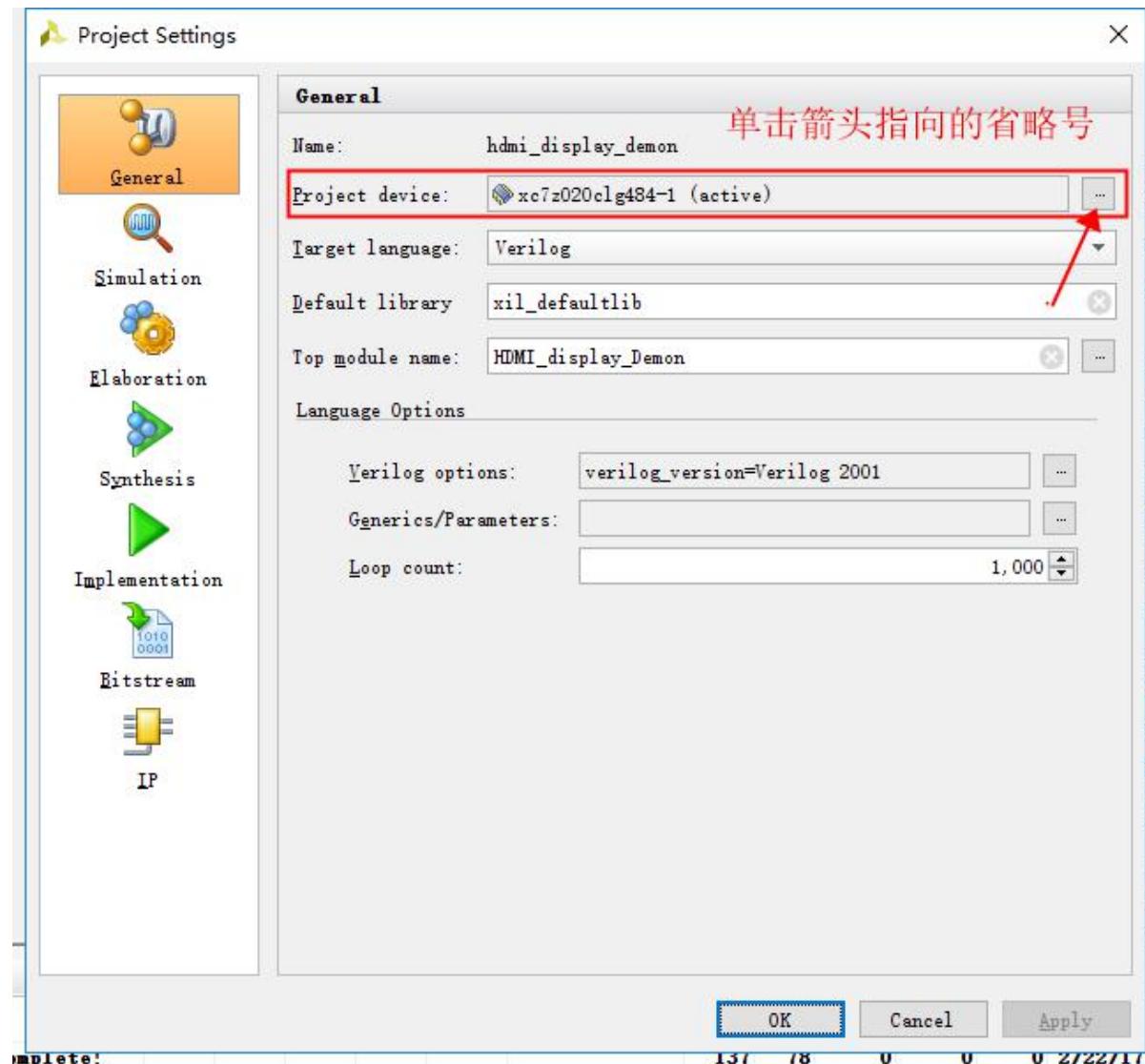
本章的工程与上一章 ADV7511 实现 HDMI 的工程基本相似。因此，为了节省开发的时间，可以直接在上一章的基础上修改。

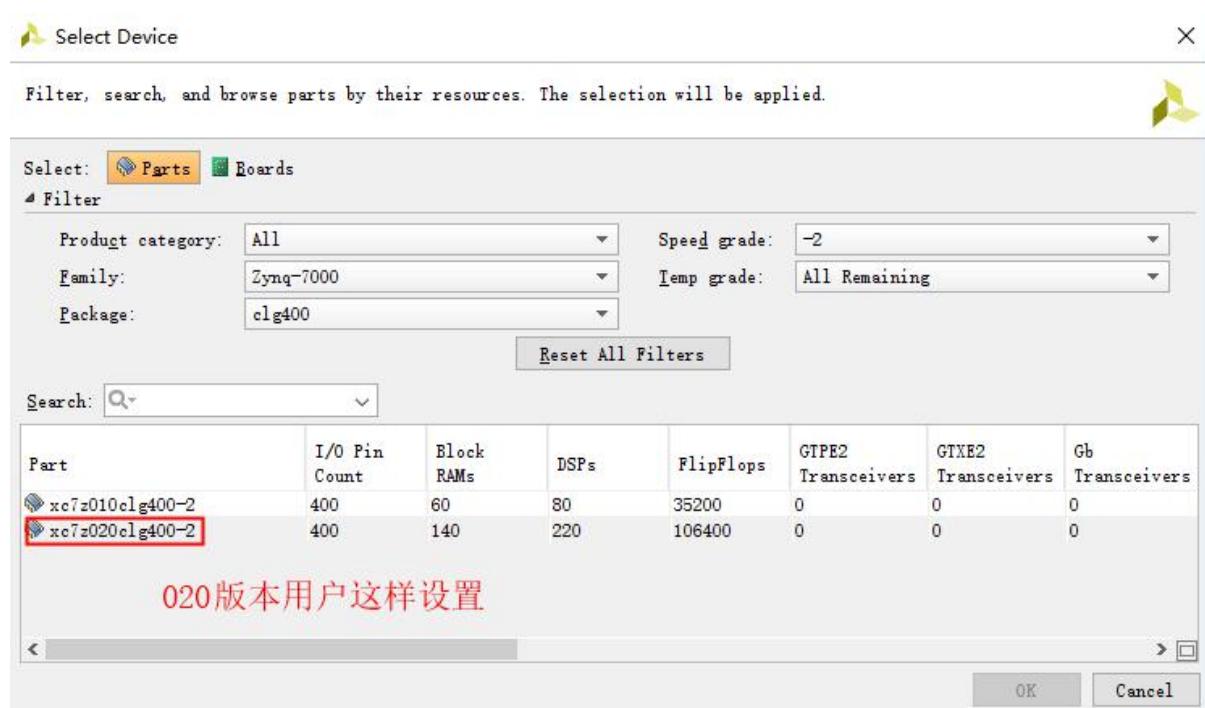
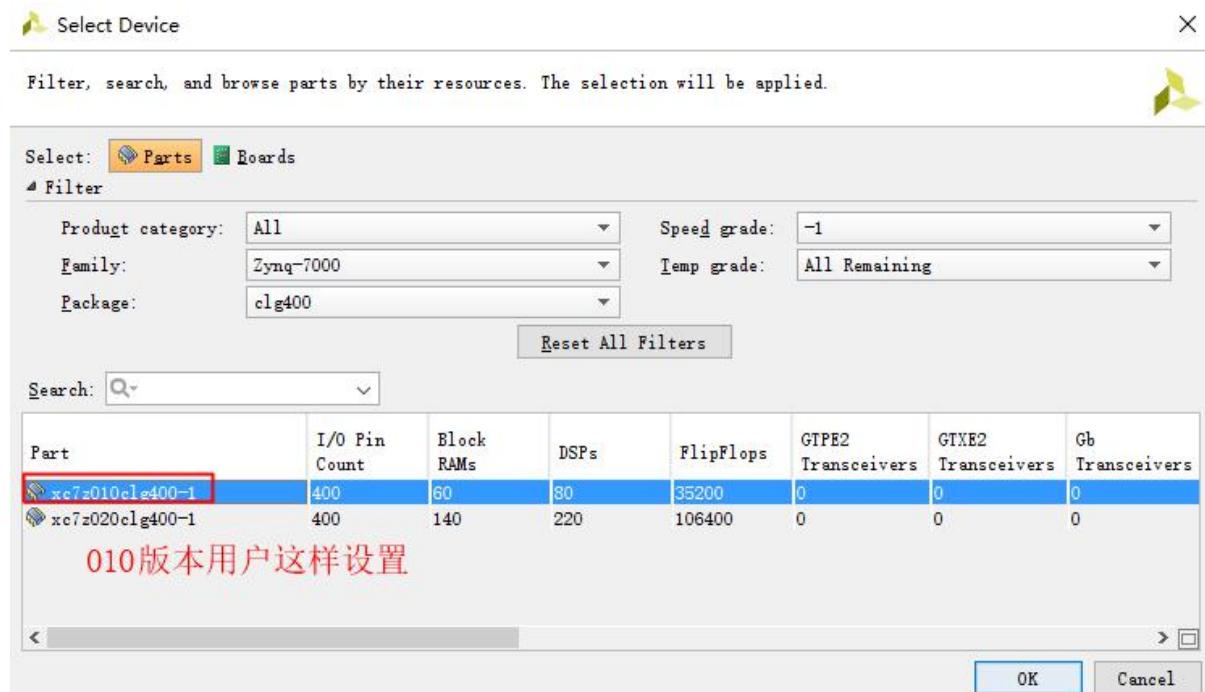
Step1：直接打开上一章的工程。

Step2：单击 Project settings，对工程进行设置。

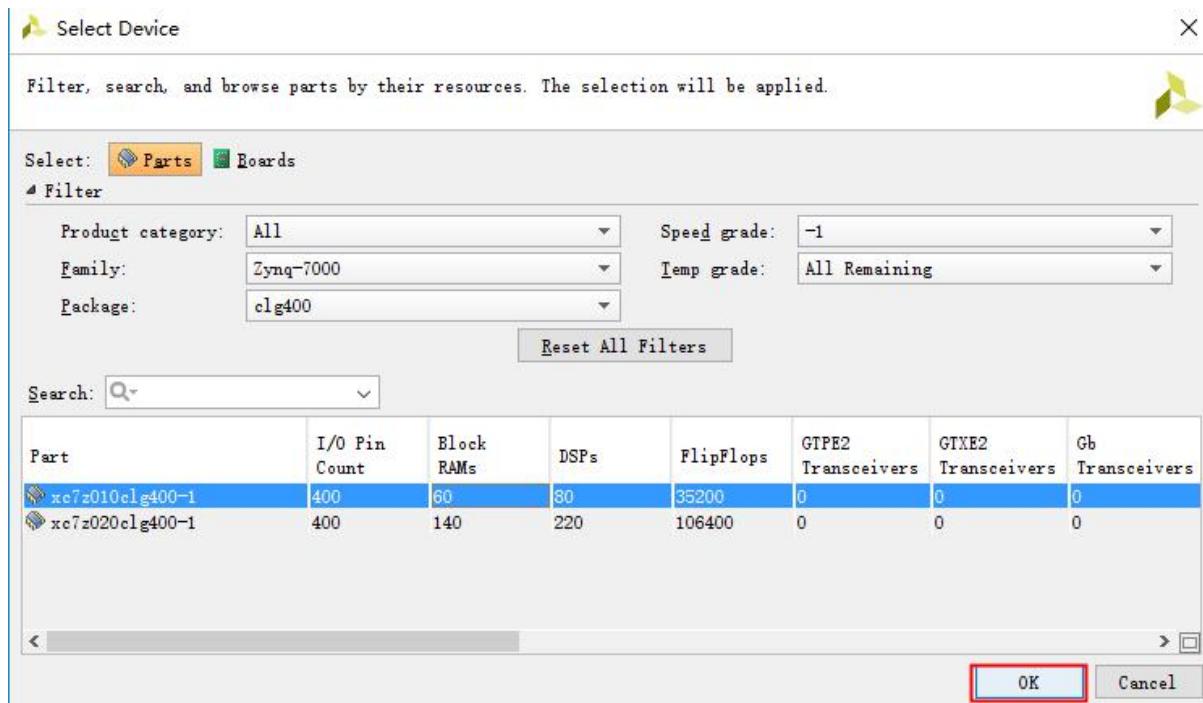


Step3：在 General 选项中，将芯片类型设置为 XC7Z010CLG400-1（020 版本用户选择 XC7Z020CLG400-2）。

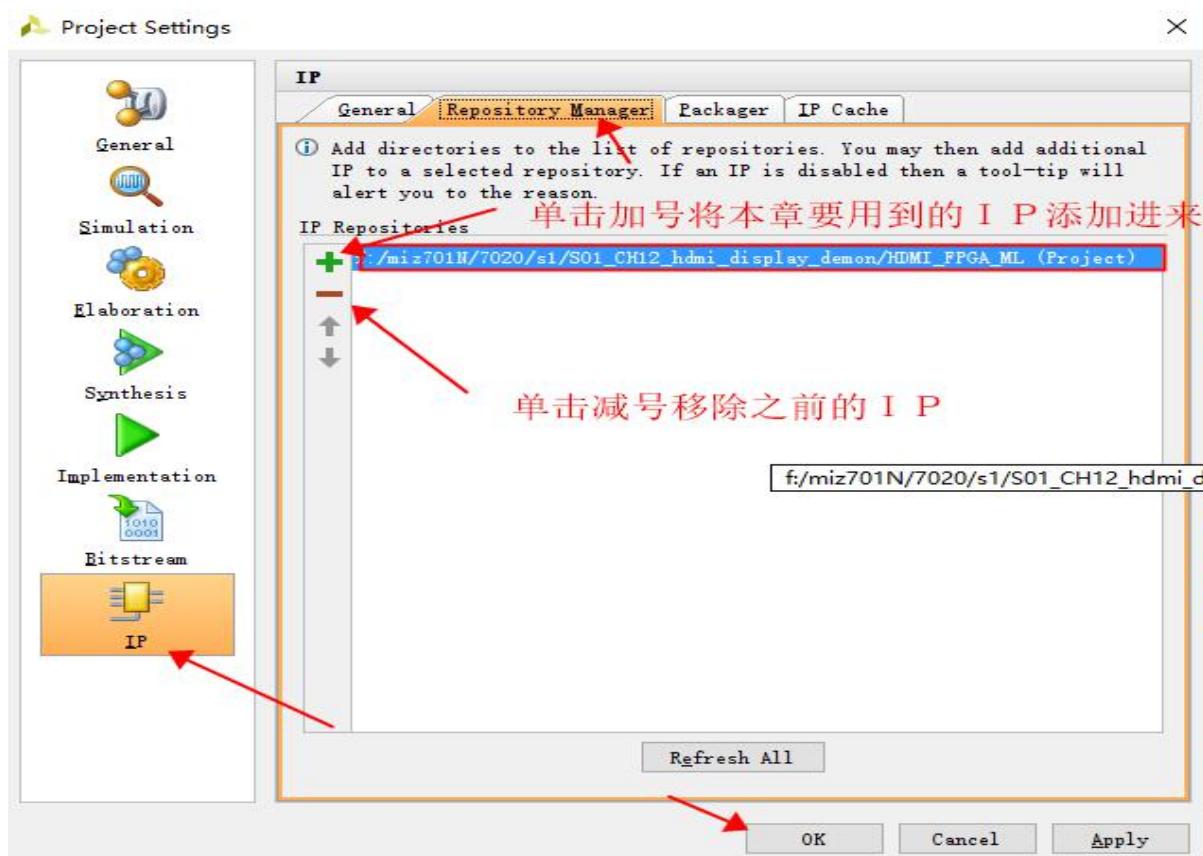




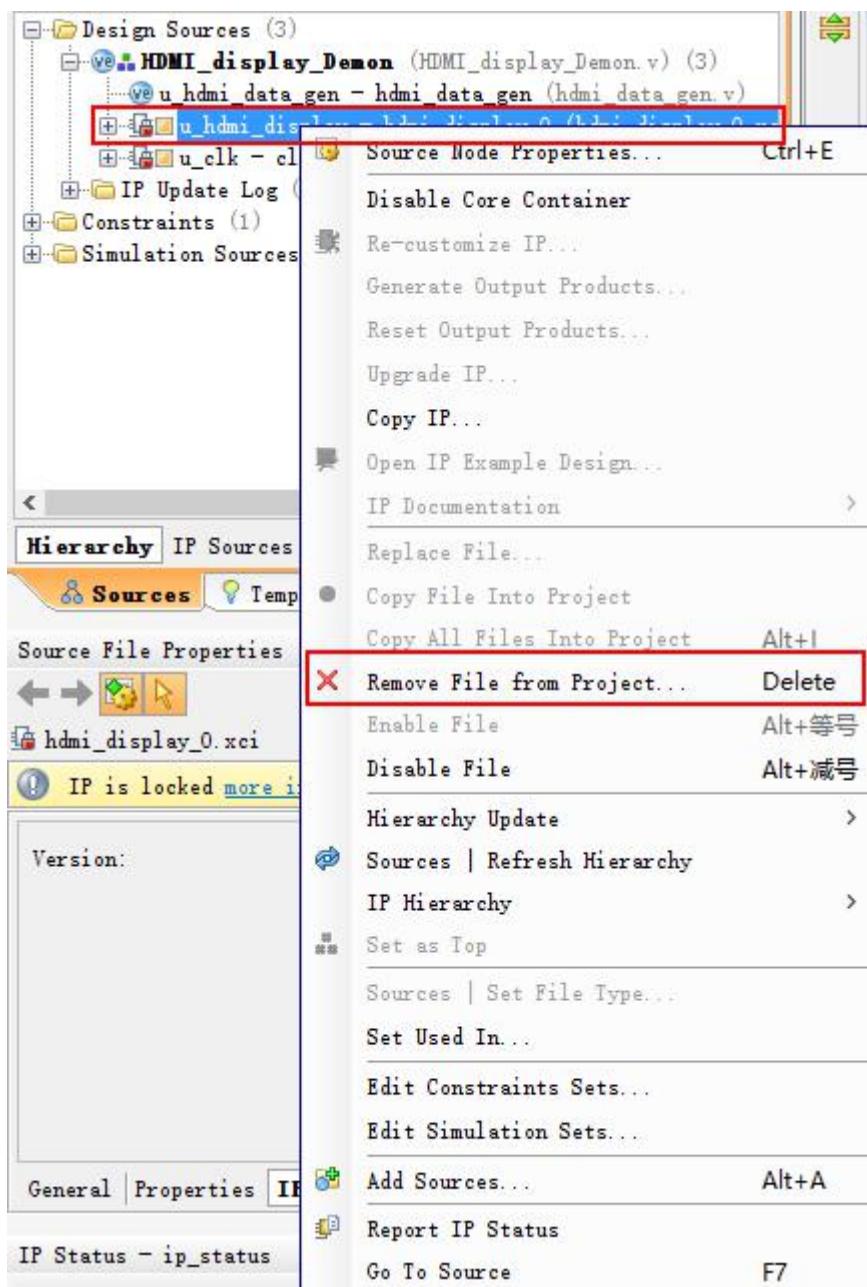
Step4: 单击 OK 后返回到 Project Settings。



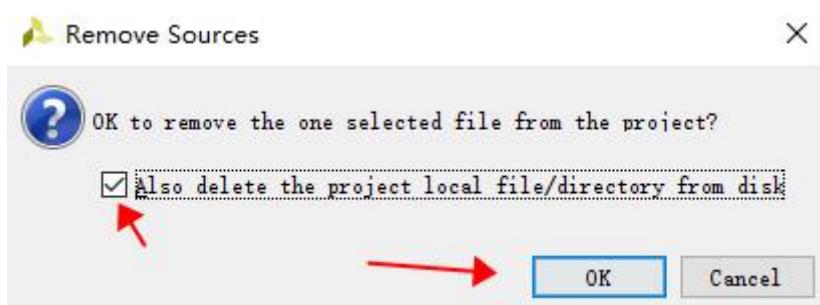
Step5：选择 IP 选项，再选择 Repository Manager，单击里面的-号将之前的 ADV7511 的 IP 核的路径移除，再单击+号将本章要用到的 IP 核添加进来，最后单击 OK，再选择 NO，完成工程的配置（IP 核在我们提供的源代码对应章节的文件包里面的 Miz_ip_lib 文件夹中找到）。



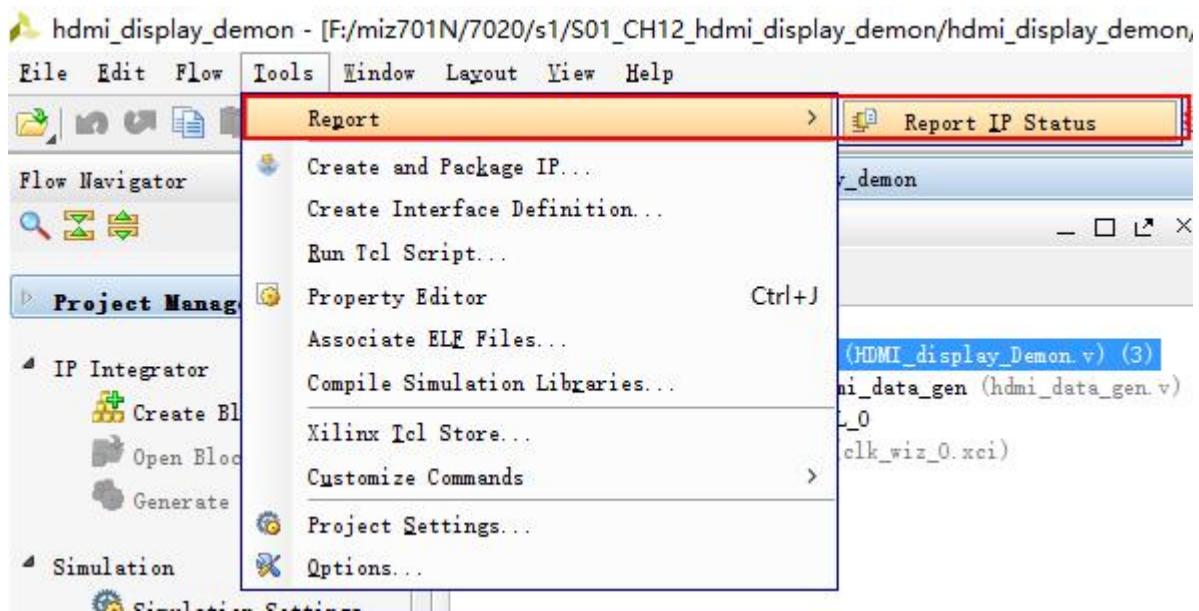
Step6: 选中之前工程的 HDMI 的 IP，右单击选择 Remove file from project。



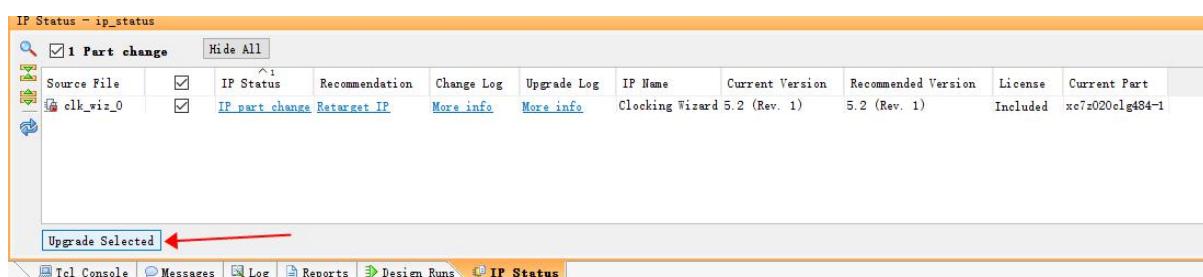
Step7: 勾选复选框，单击 OK。



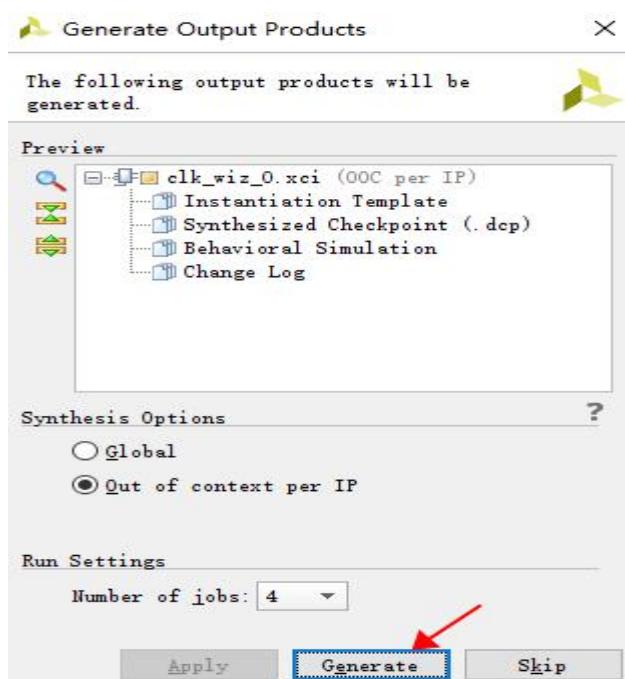
Step8: 单击 Tools 菜单下的 Report-Report IP status 命令。



Step9: 最下面的窗口中单击 Upgrade Selected 选项。



Step10: 单击 Generate, 更新 IP。



Step11: 双击 VGA_Display_Demon.v，将代码用如下程序替换。

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

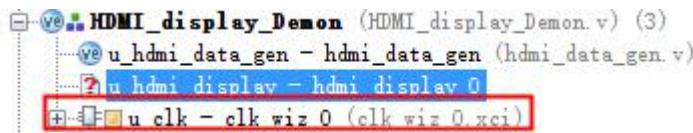
wire pixclk;
wire[7:0] R, G, B;
wire HS, VS, DE;
hdmi_data_gen u_hdmi_data_gen
(
    .pix_clk          (pixclk),
    .turn_mode        (KEY),
    .VGA_R            (R),
    .VGA_G            (G),
    .VGA_B            (B),
    .VGA_HS           (HS),
    .VGA_VS           (VS),
    .VGA_DE          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

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

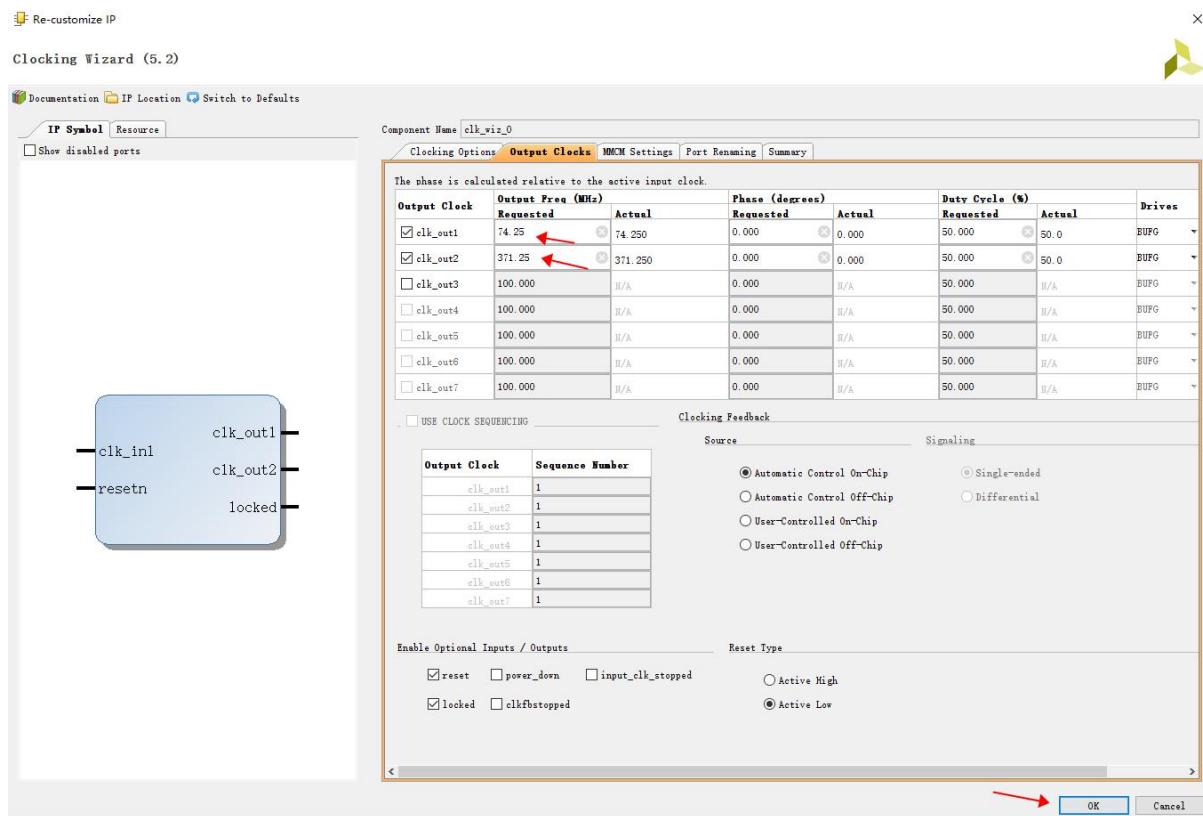
```
. VGA_DE          (DE),
. VGA_RGB         (RGB),
. HDMI_CLK_P     (HDMI_CLK_P),
. HDMI_CLK_N     (HDMI_CLK_N),
. HDMI_D2_P      (HDMI_D2_P),
. HDMI_D2_N      (HDMI_D2_N),
. HDMI_D1_P      (HDMI_D1_P),
. HDMI_D1_N      (HDMI_D1_N),
. HDMI_DO_P      (HDMI_DO_P),
. HDMI_DO_N      (HDMI_DO_N)
);

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

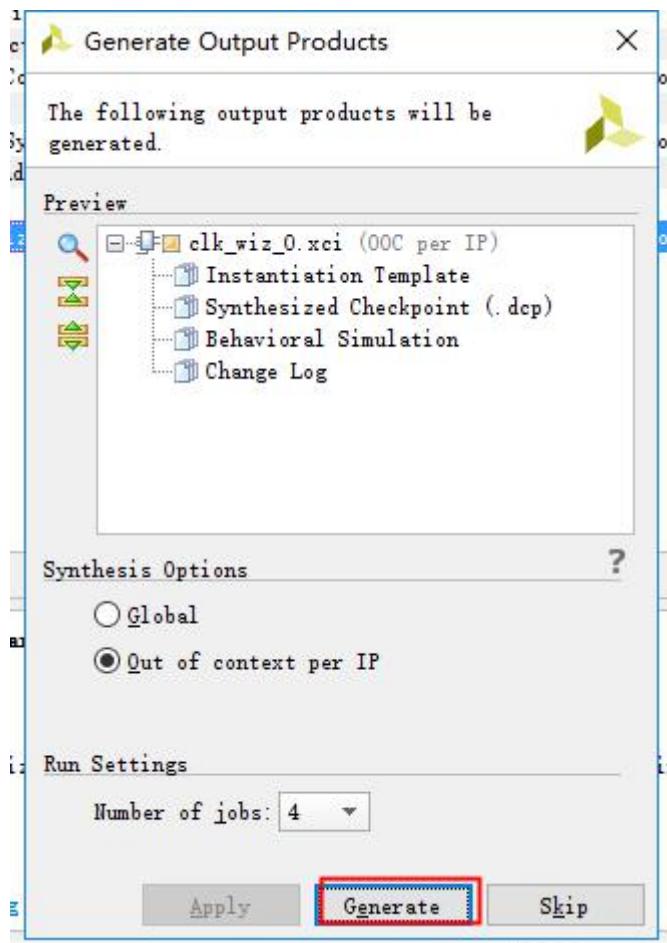
Step3: 双击  `u_clk - clk_wiz_0 (clk_wiz_0.xci)`，对其重新配置。



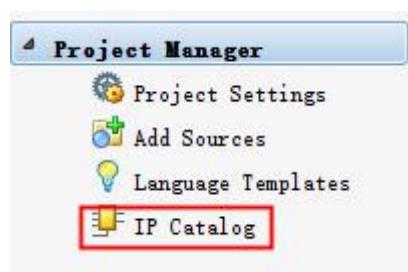
Step4: 在 output clocks 选项卡下如下图设置，其余参数默认配置即可，然后单击 OK。



Step5：单击 Generate。



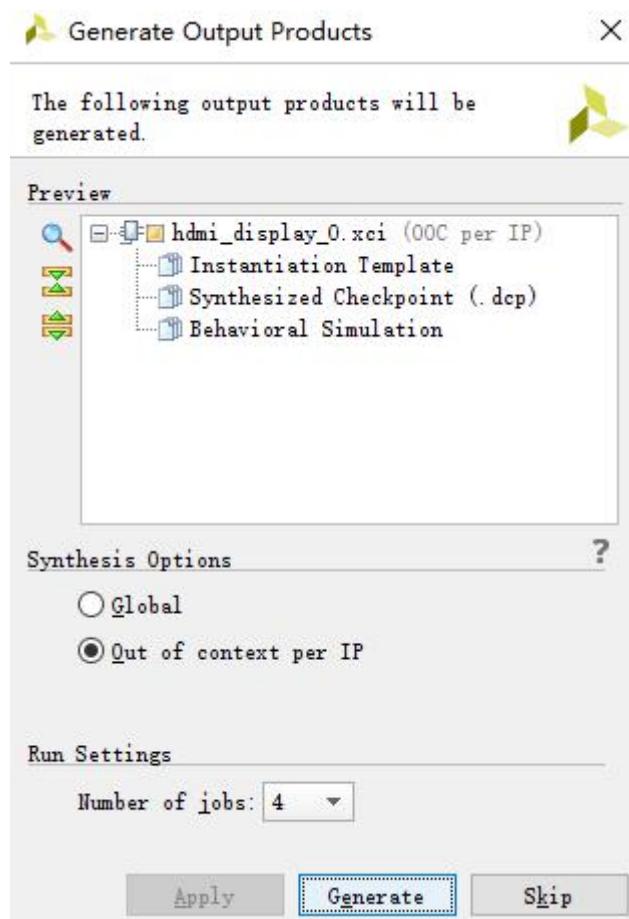
Step8: 单击 IP Catalog 添加一个 IP。



Step9: 输入 IP 的名字, 双击之后单击 OK 不做任何改动, 将其添加到工程当中。



Step10: 单击 Generate。



这样我们就完成了工程的修改下面还要添加管脚约束文件。

12.4 添加管脚约束文件

Step1: 打开 vga_pin.xdc 文件, 用下面给出的约束文件替换原来的约束文件。

```
set_property IOSTANDARD LVCMOS33 [get_ports clk_100M]
set_property IOSTANDARD LVCMOS33 [get_ports KEY]
set_property IOSTANDARD TMDS_33 [get_ports HDMI_CLK_P]
set_property IOSTANDARD TMDS_33 [get_ports HDMI_D0_P]
set_property IOSTANDARD TMDS_33 [get_ports HDMI_D1_P]
set_property IOSTANDARD TMDS_33 [get_ports HDMI_D2_P]
set_property PACKAGE_PIN H16 [get_ports clk_100M]
set_property PACKAGE_PIN M14 [get_ports KEY]
set_property PACKAGE_PIN K17 [get_ports HDMI_CLK_P]
set_property PACKAGE_PIN L19 [get_ports HDMI_D0_P]
set_property PACKAGE_PIN M17 [get_ports HDMI_D1_P]
```

```

set_property PACKAGE_PIN L16 [get_ports HDMI_D2_P]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
set_property PACKAGE_PIN N15 [get_ports {LED[0]}]
set_property PACKAGE_PIN N16 [get_ports {LED[1]}]
set_property PACKAGE_PIN M19 [get_ports {LED[2]}]
set_property PACKAGE_PIN M20 [get_ports {LED[3]}]

```

12.5 编译并且产生 bit 文件

Step1:单击综合

Step2:单击执行

Step3:单击产生 bit



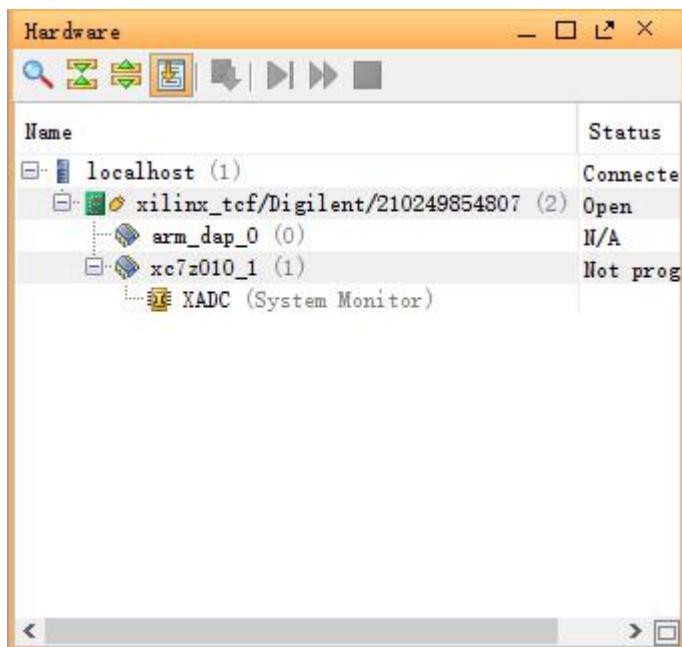
12.6 下载程序

Step1:给开发板通电，并且连接下载器

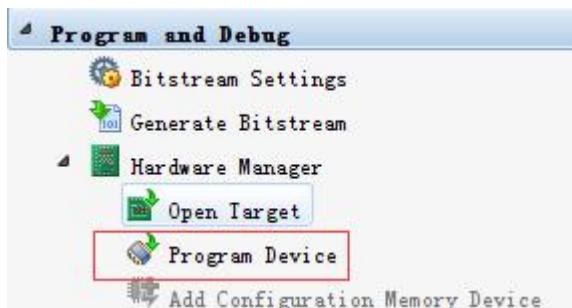
Step2:单击 OpenTarget 然后单击 Auto Connect



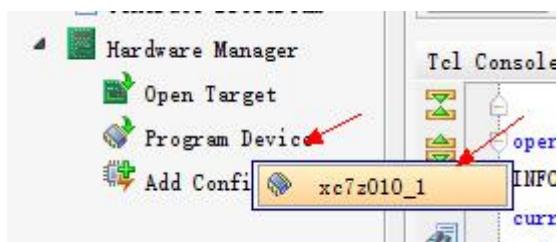
Step3:连接成功后



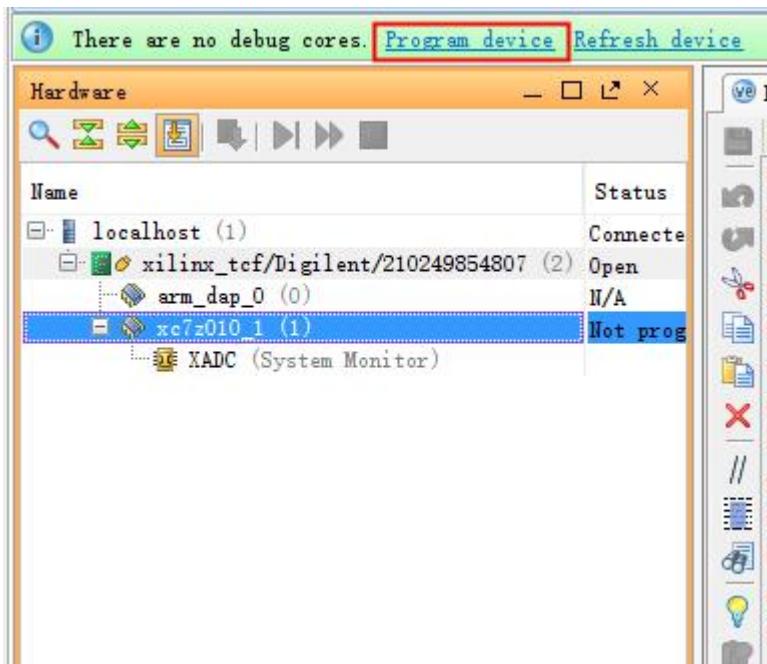
Step4:单击 Program Device



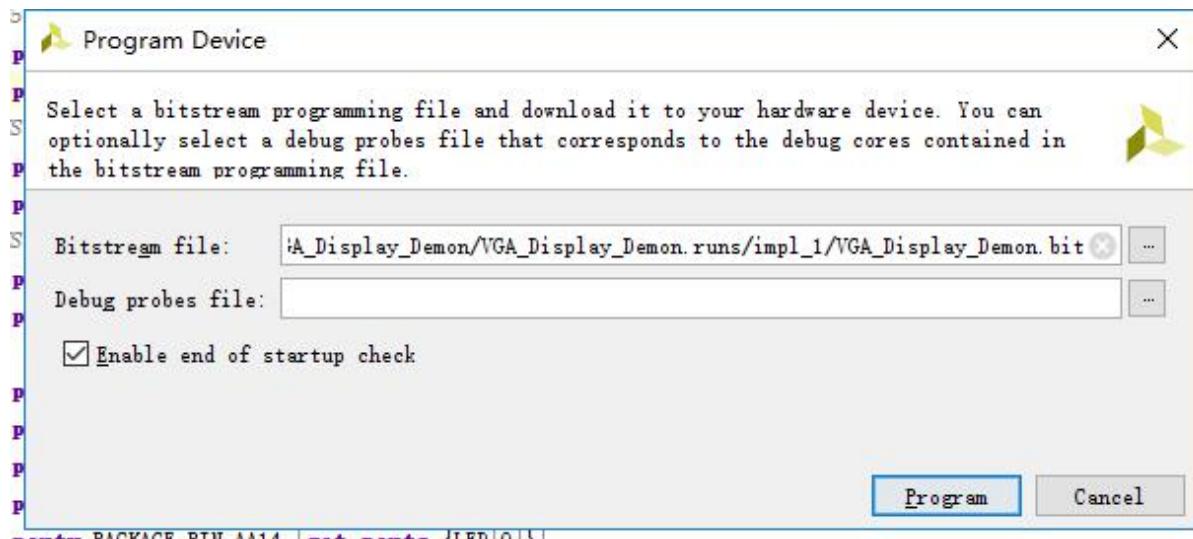
Step5:单击 Program Device 然后选择 XC7Z010_1



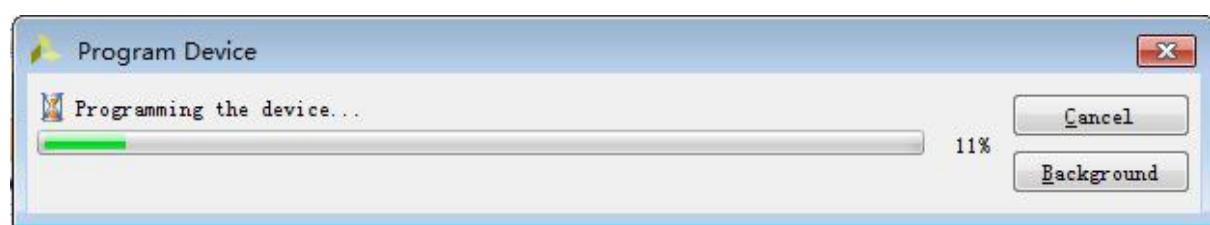
Step6:或者也可以从顶部单击 Program device



Step7: 弹出的对话框中有我们要下载的 Bit 文件

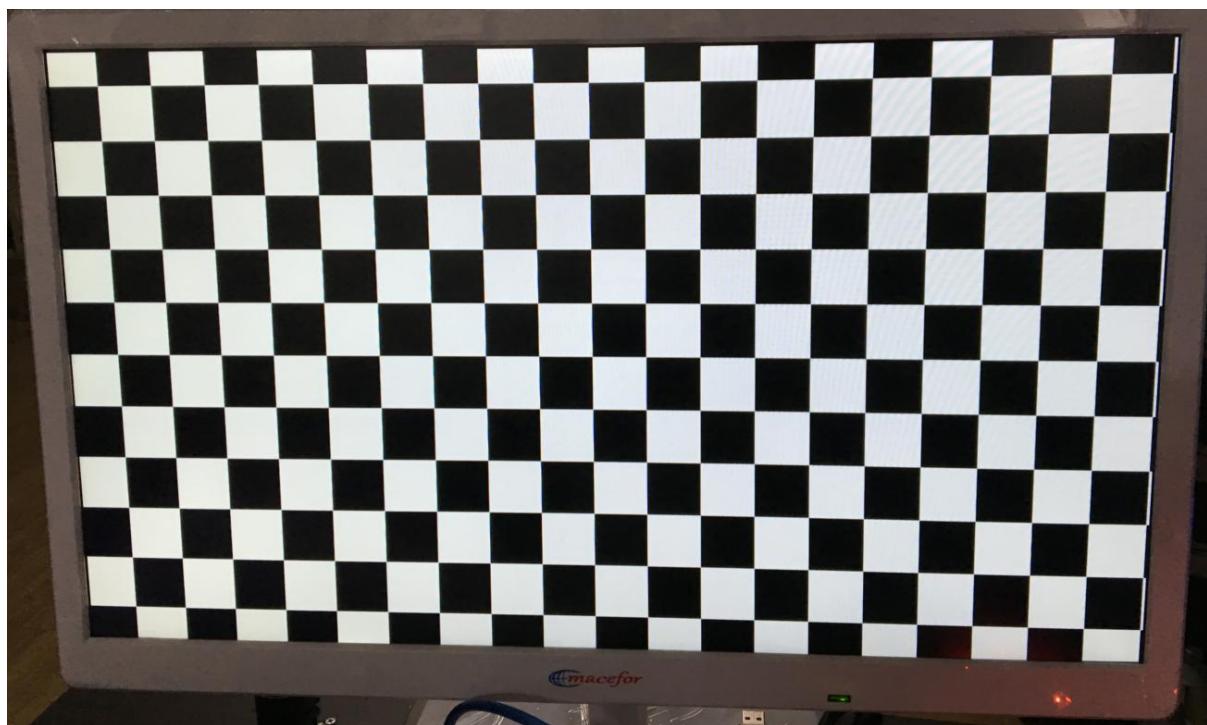


Step7: 下载过程



12.7 实验结果

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



【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。

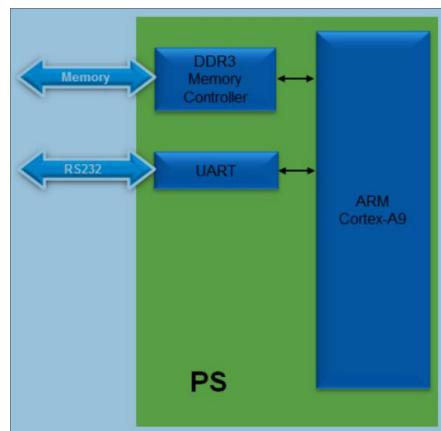
S02_CH01_Hello World 实验

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

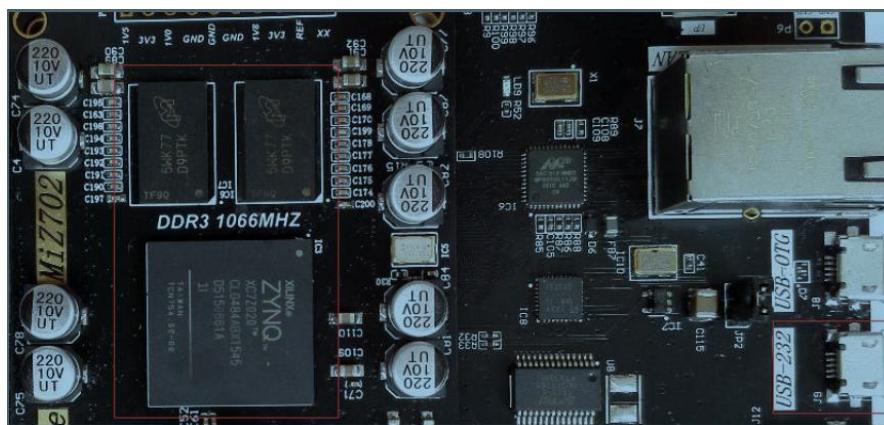
这张图展示了我们需要构建的最小系统。并且下面的嵌入式实验会基于这个最小系统进行添加外设。



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。

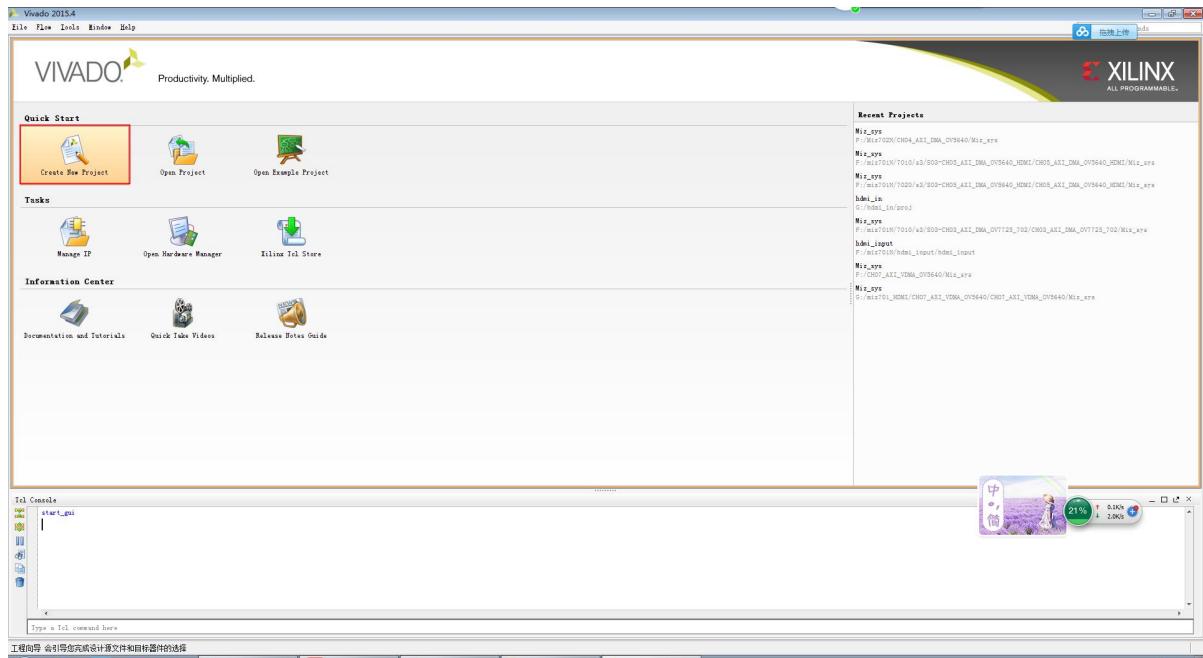
1.2 硬件电路分析

红色线框内就是本次实验需要用到的资源，分别为 CPU XC7Z020、2 片 512MB 内存、一个 Micro 接口的 USB 转 UART。

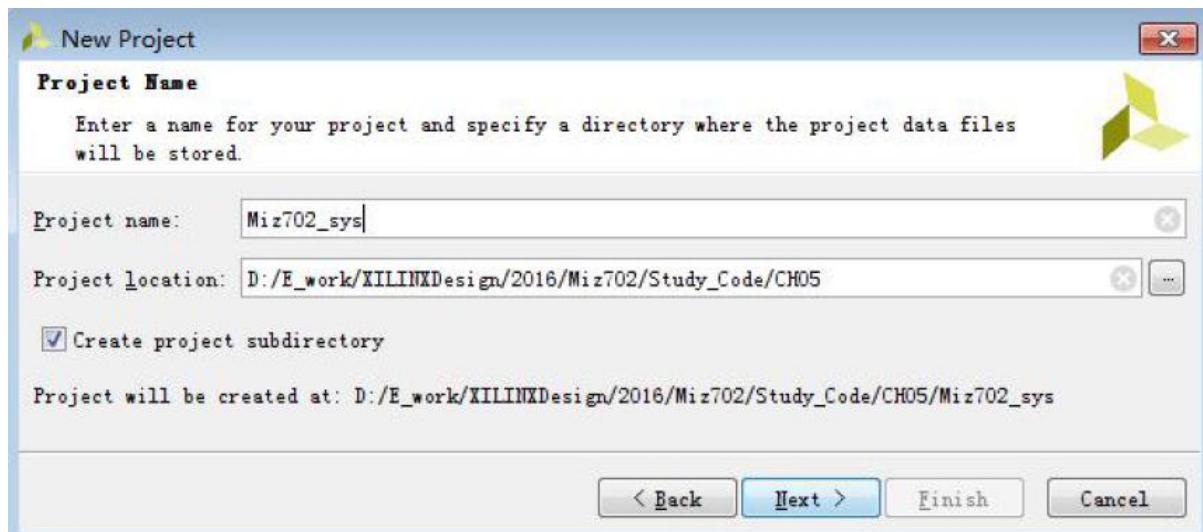


1. 3创建一个VIVADO工程

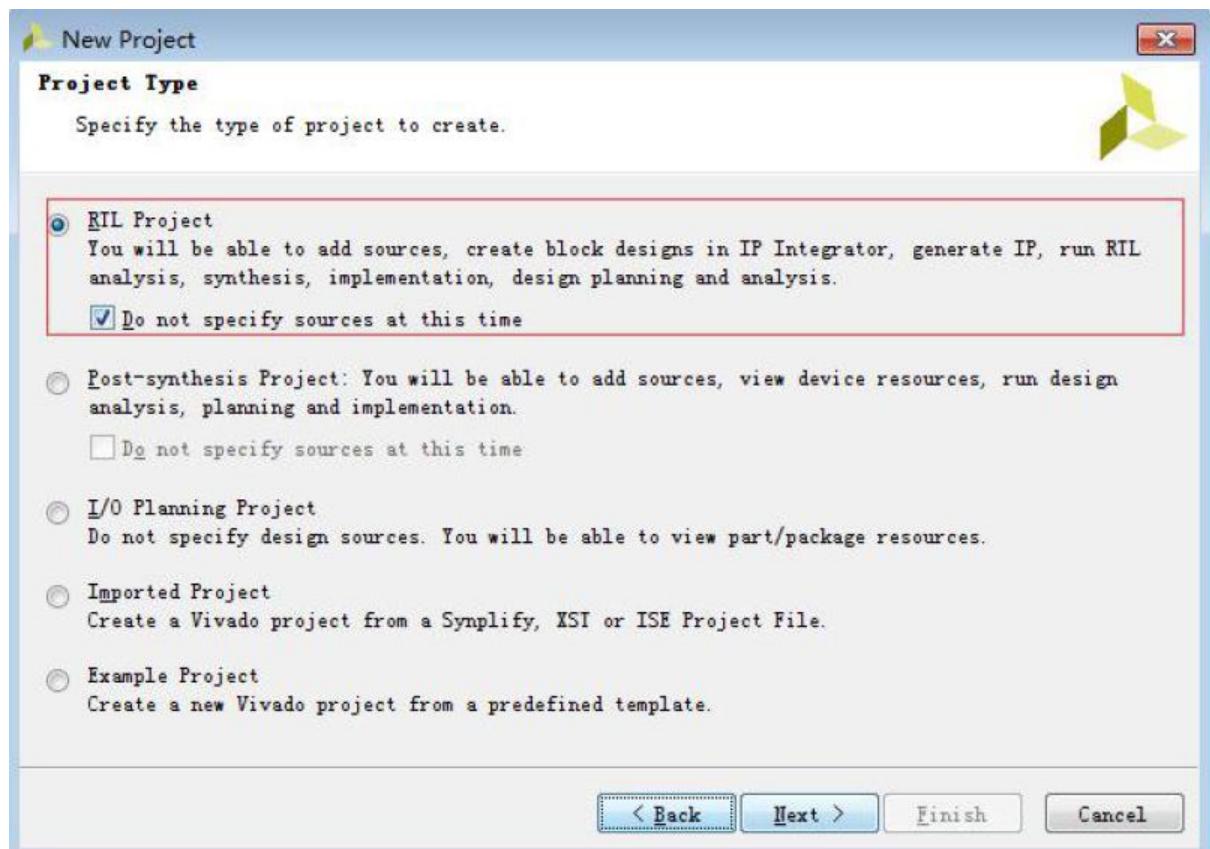
Step1：在打开的VIVADO软件界面，单击Create New Project。



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。

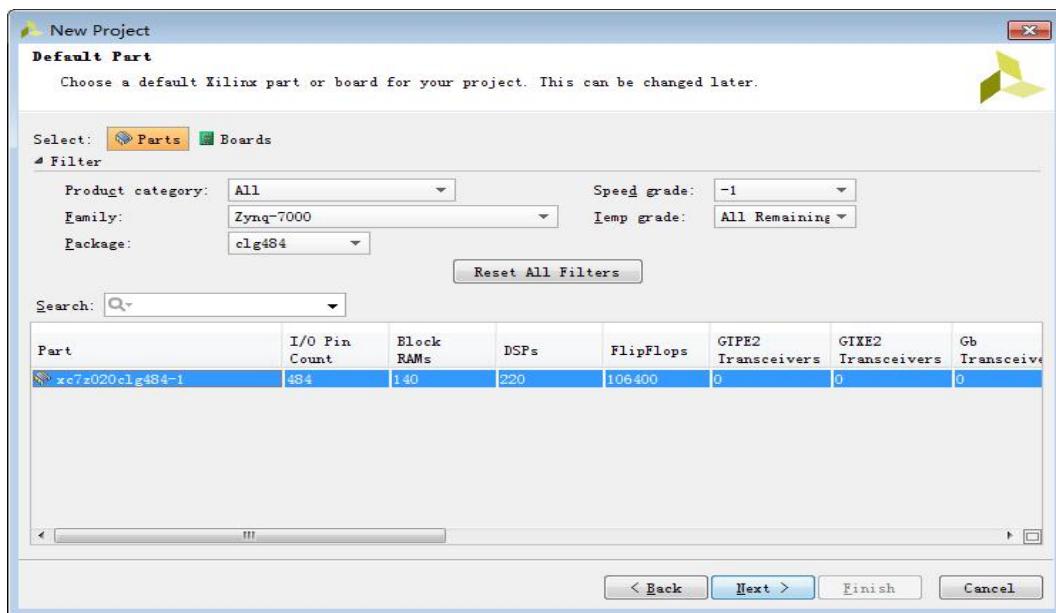


Step3：

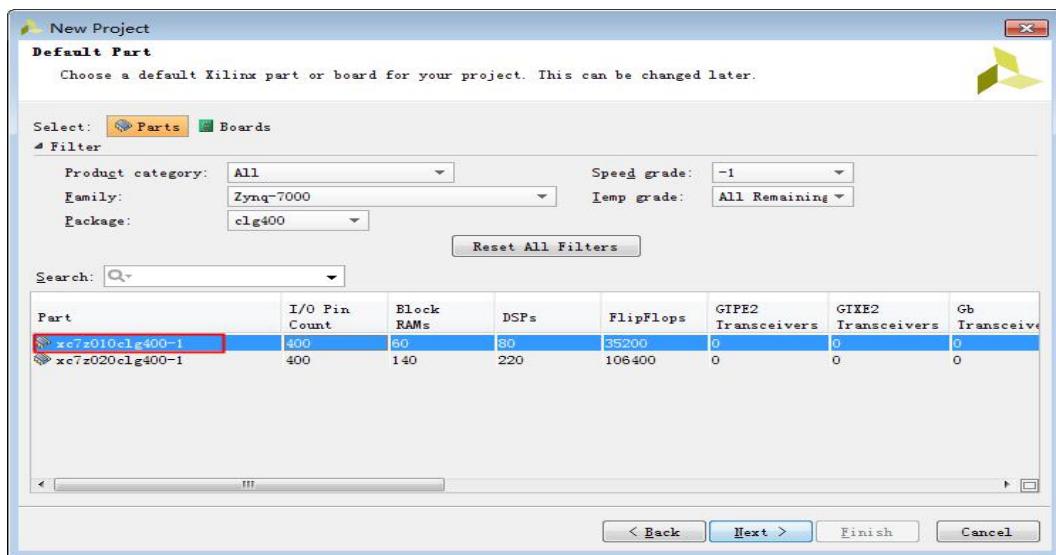


Step4:选择芯片类型。 (请大家根据自己实际的类型选择)

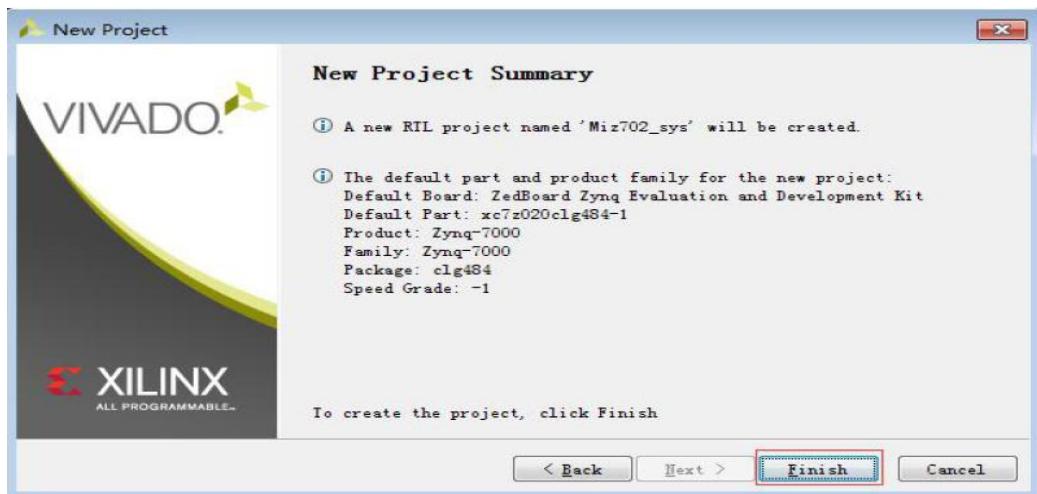
Miz702与Miz702N用户如下设置:



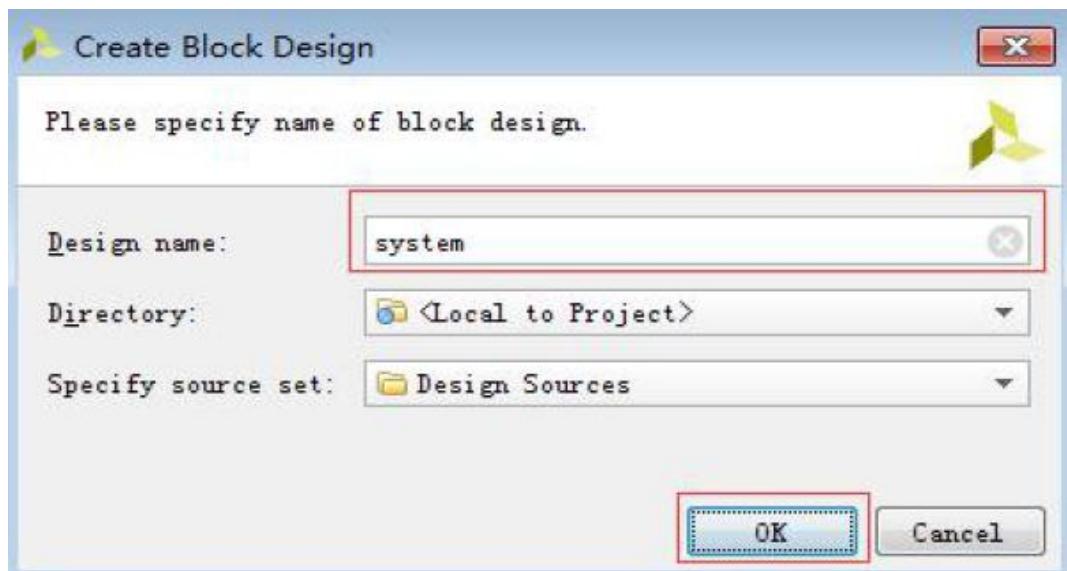
Miz701N与Miz701用户如下设置:



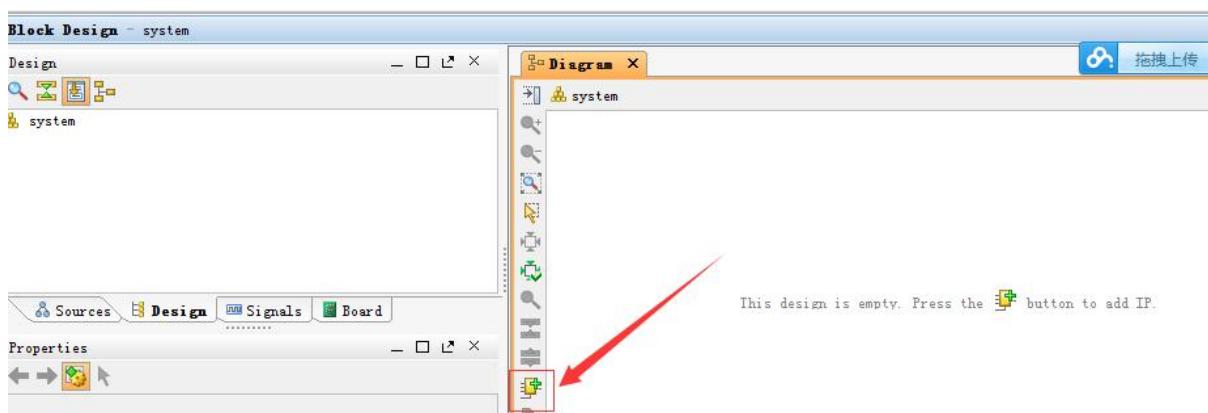
Step5:



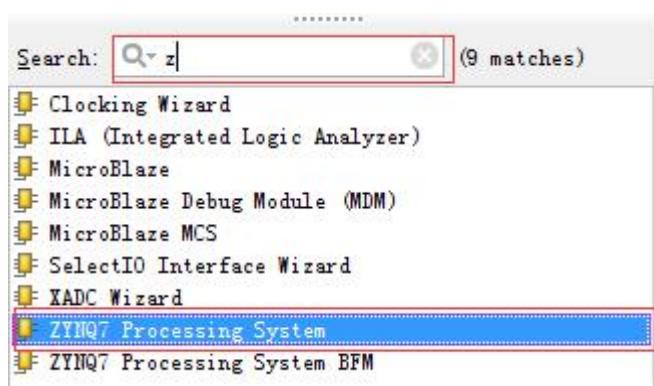
Step6: 单击Create Block Design, 输入System。



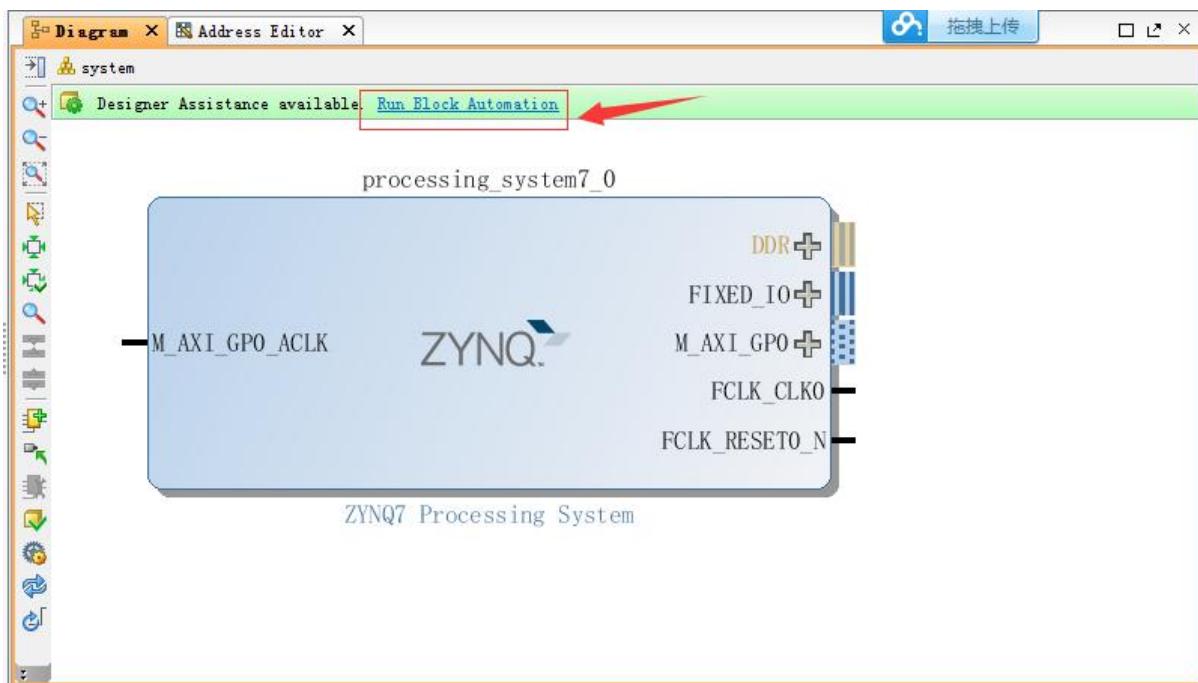
Step7: 单击下图中 添加IP按钮



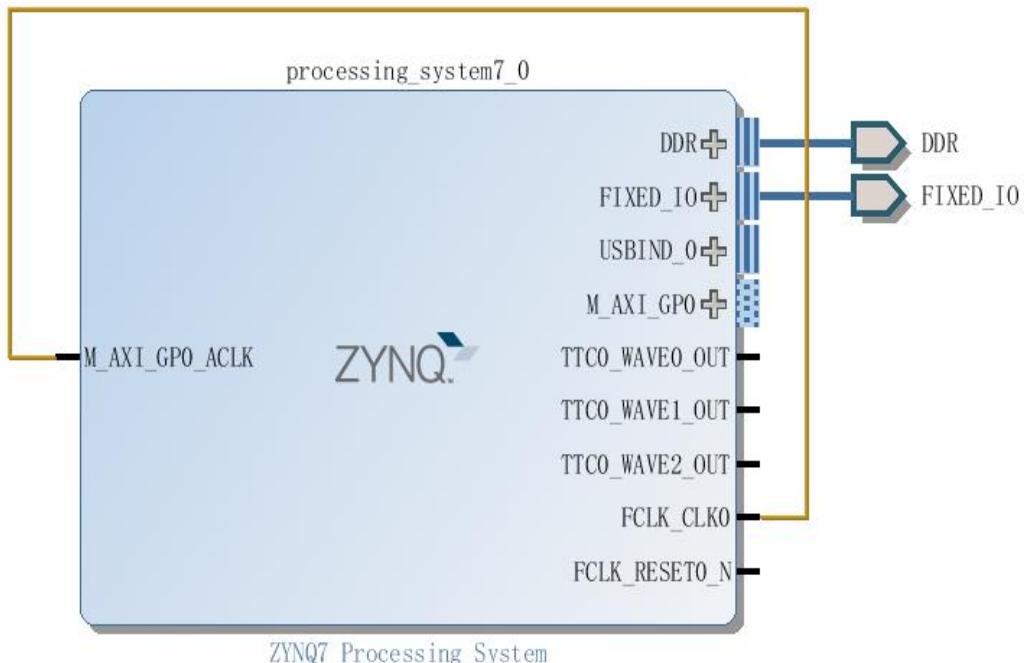
Step8: 搜索单词z选择ZYNQ7 Processing System，然后双击



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

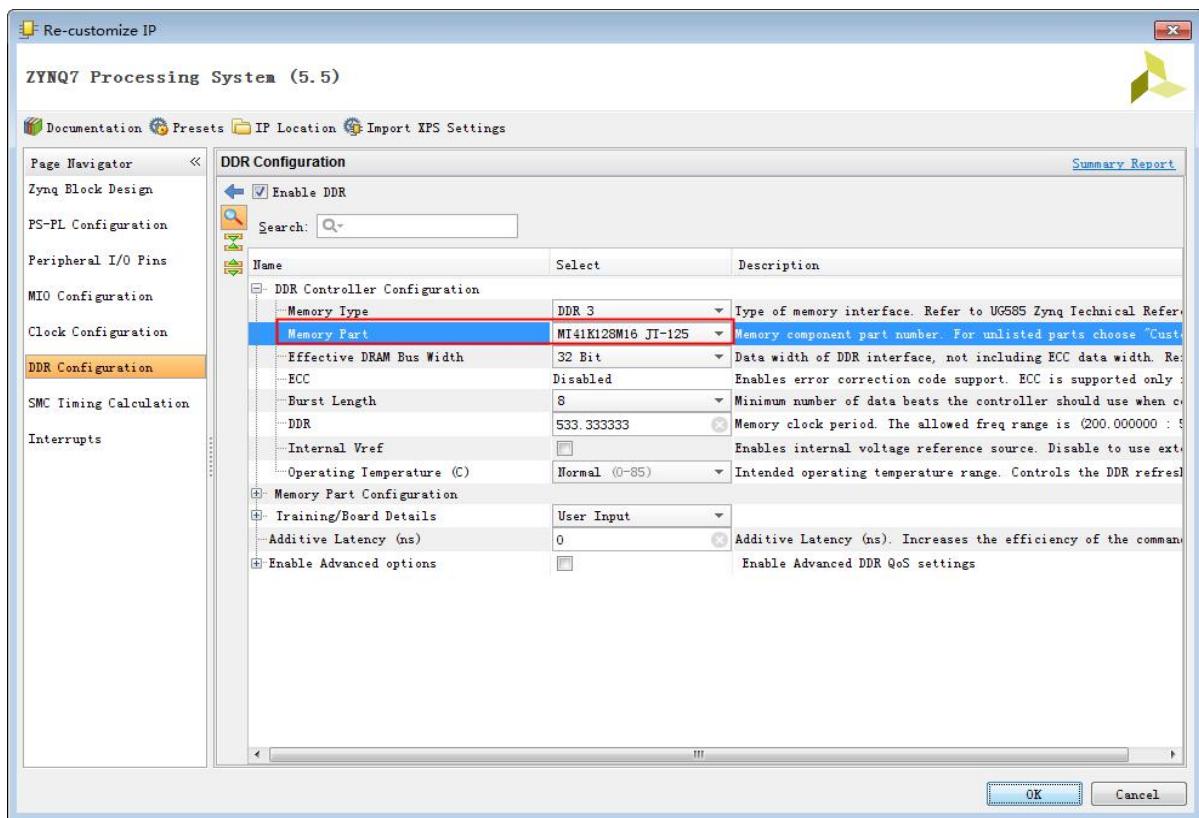
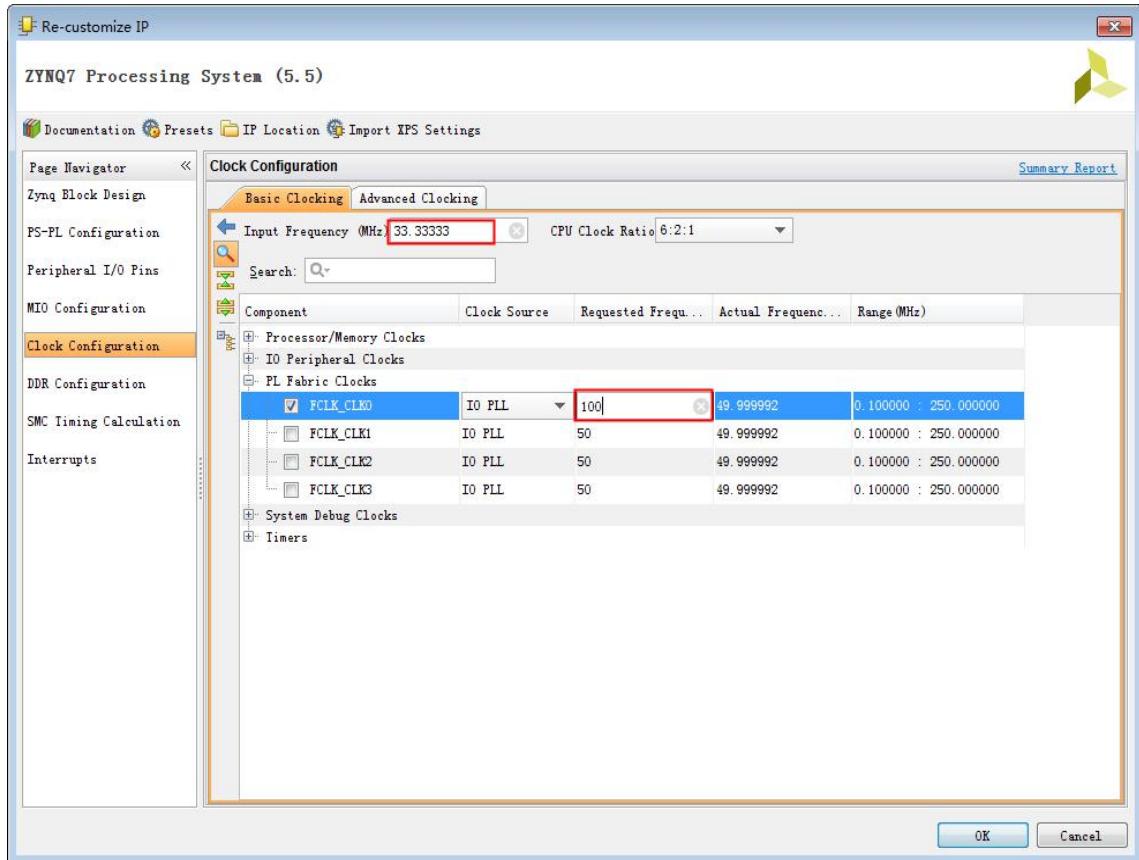


Step10：在Block文件中，我们进行连线，将鼠标放在引脚处，鼠标变成铅笔后逆行拖拽，连线如下图所示。连线的作用就是把PS的时钟可以接入PL部分，当然这里我们暂时用不到PL部分的资源。

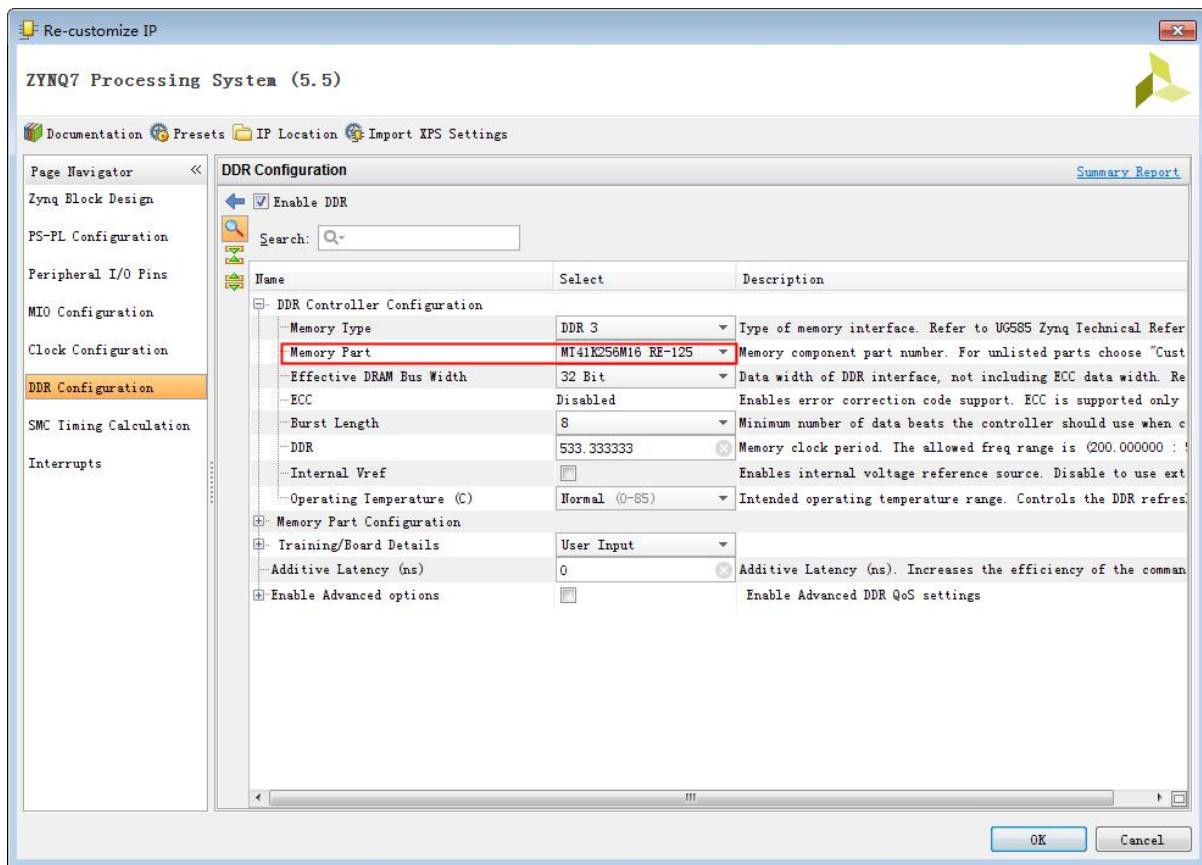
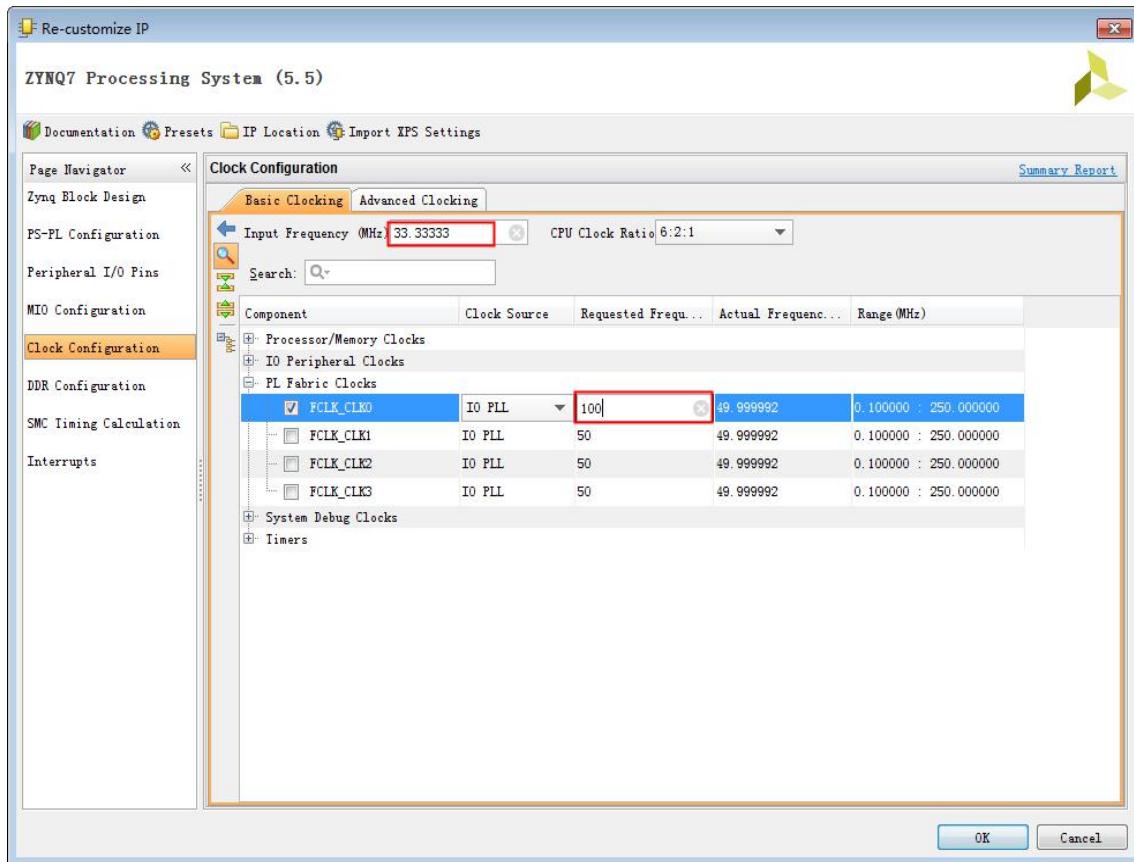


Step11：双击ZYNQ CPU IP，对其进行设置，使其对应我们的硬件设置。在此部分，我们需要做的就是修改时钟频率，内存类型和接口输出。需要注意的是，如果时钟频率与内存类型与我们的硬件不一致时，SDK中的程序会崩溃，运行不过来。这在后续的调试中，是一个小技巧。正确的配置是成功的必要条件。

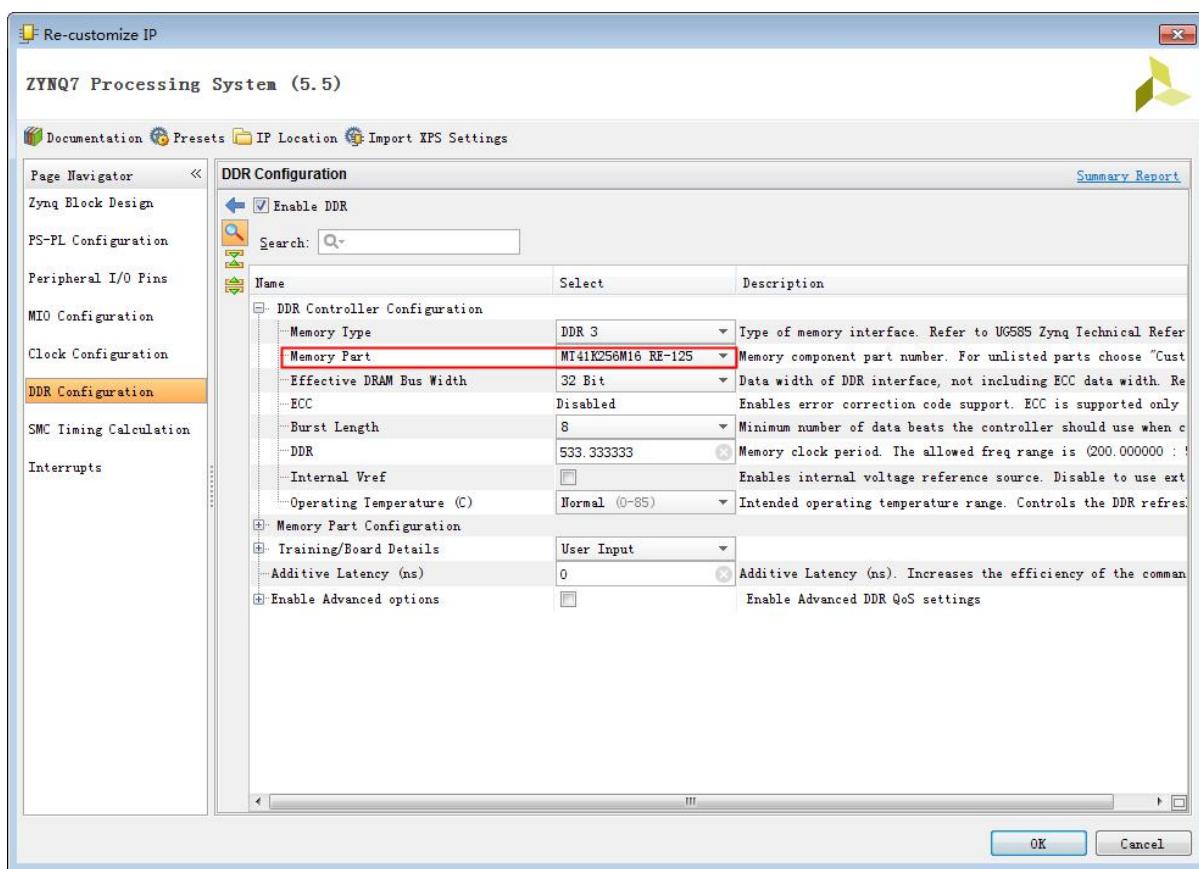
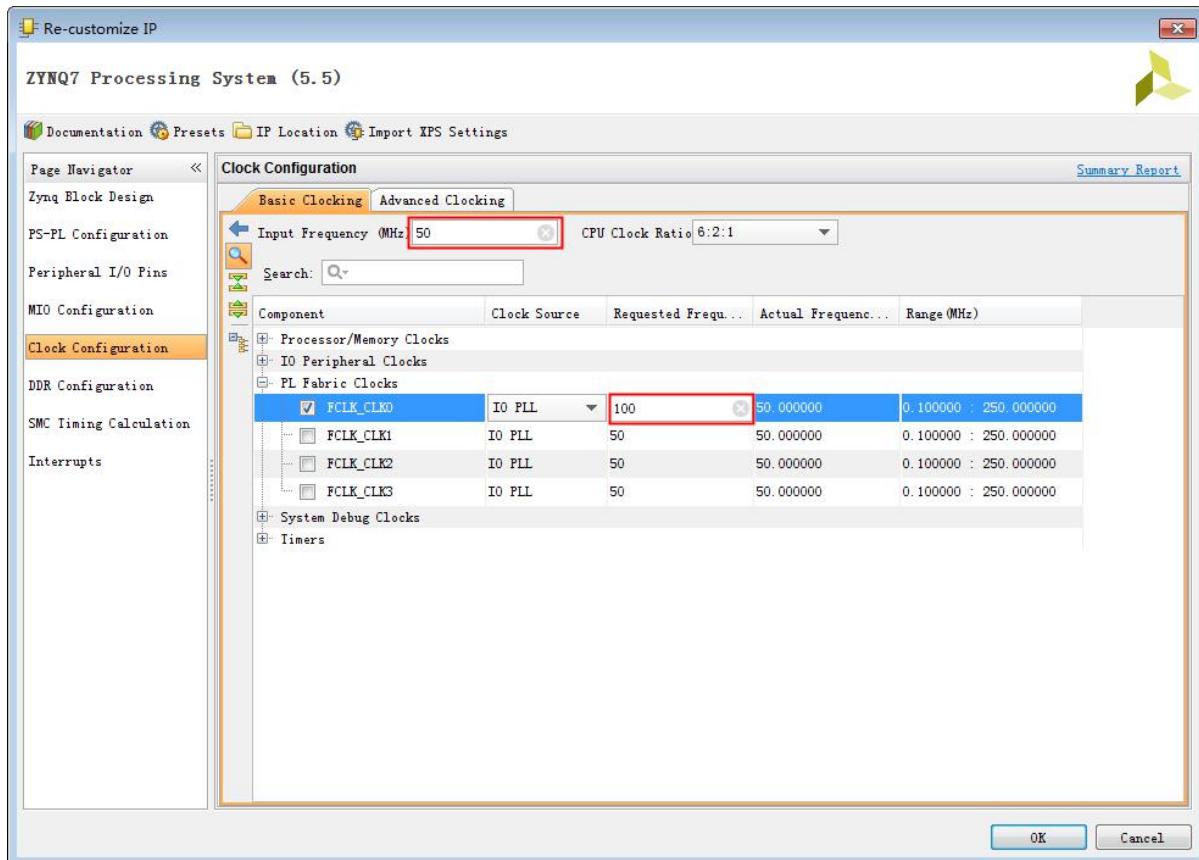
Miz702时钟及内存型号配置如下：



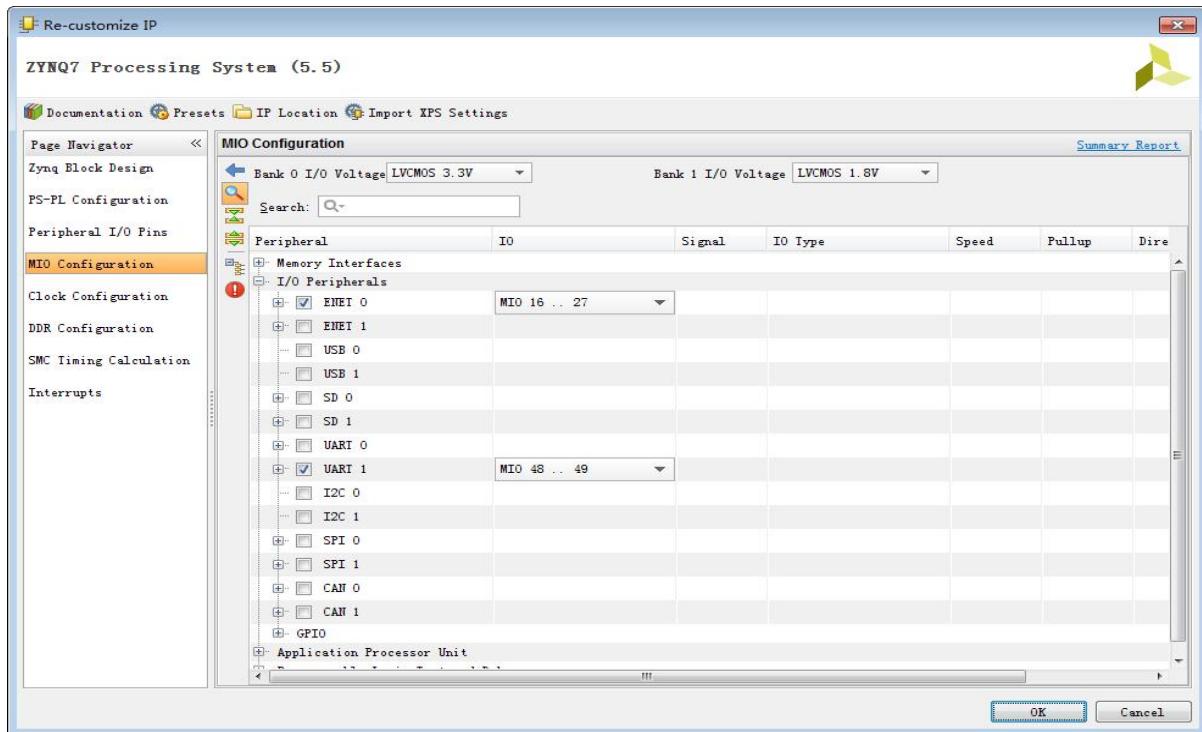
Miz702N时钟及内存型号配置如下：



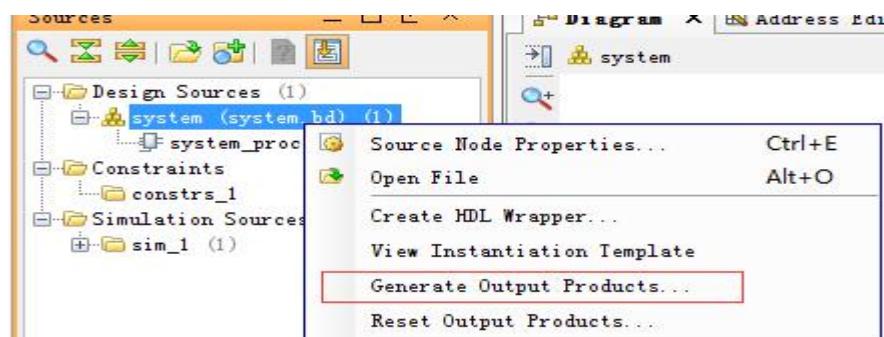
Miz701与Miz701N时钟及内存型号配置如下：



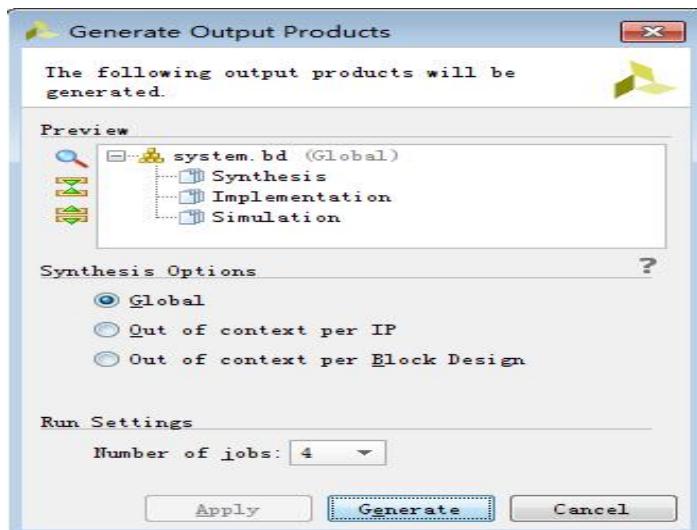
Step12: 设置外扩接口，之后点击OK。



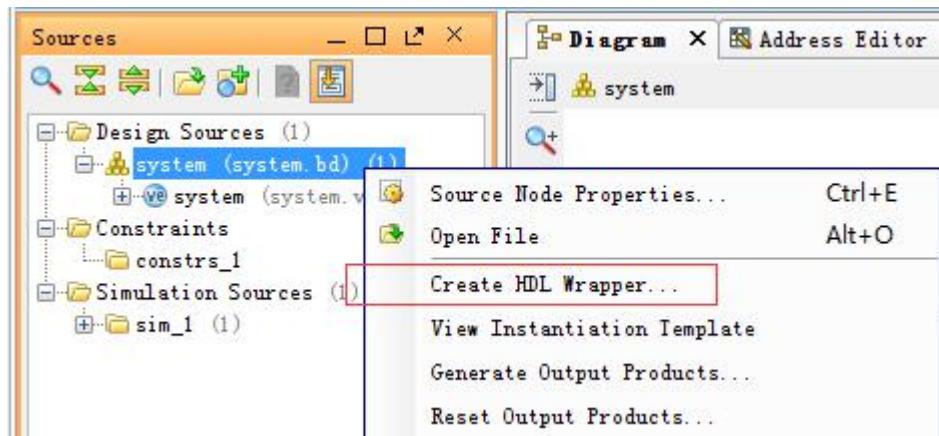
Step13: 右击 system.bd, 单击Generate Output Products。



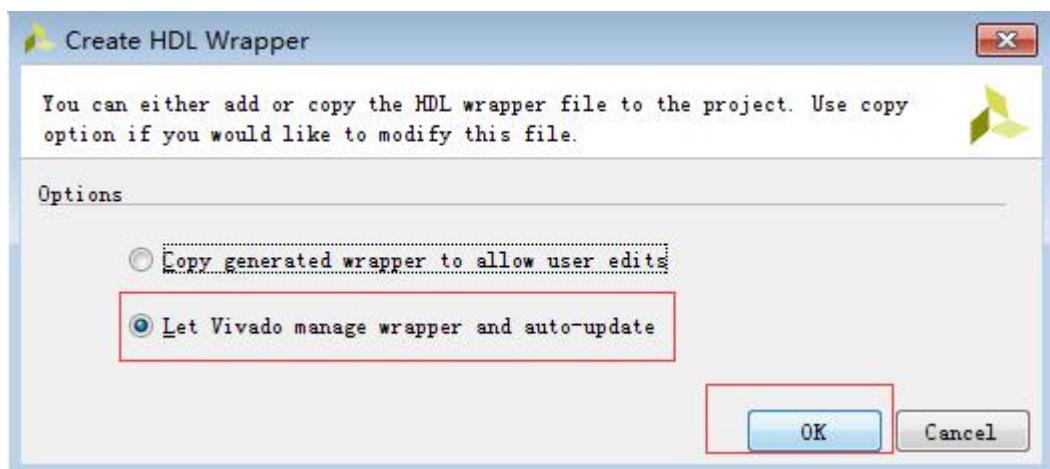
Step14: 支部操作会产生执行、仿真、综合的文件，可以看出来最后的硬件设计步骤还是回到了我们前面的FPGA开发上来了。



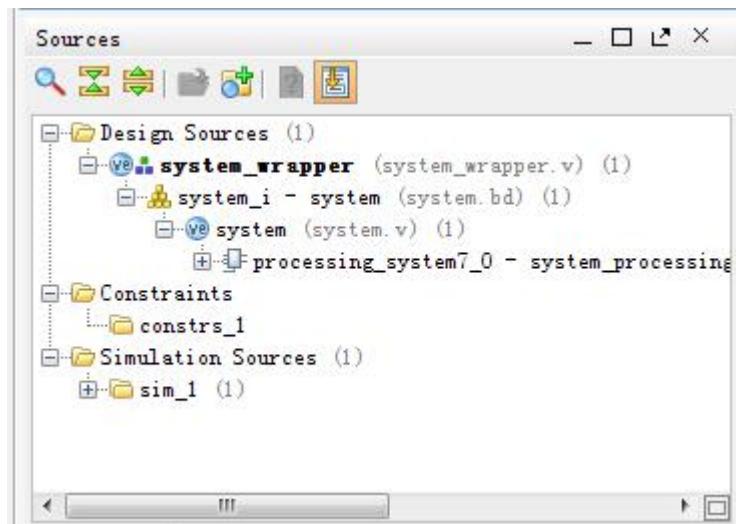
Step15:右击 system.bd 选择 Create HDL Wrapper 这步的作用是产生顶层的 HDL 文件



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

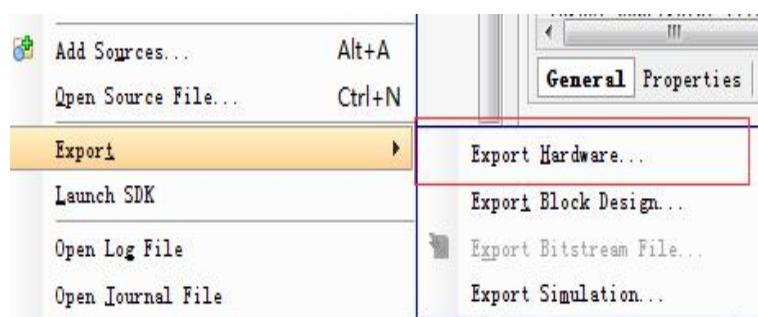
```

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。

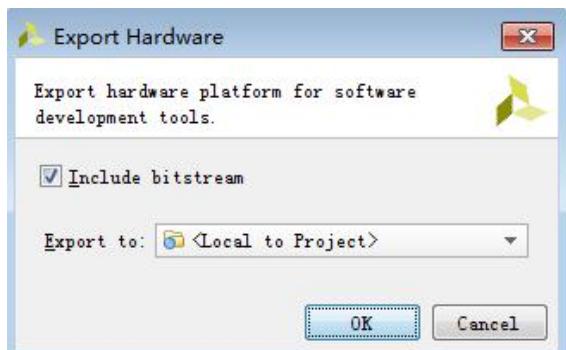
Step19: 执行->产生 bit 文件。 

1.4 导出 SOC 硬件到 SDK

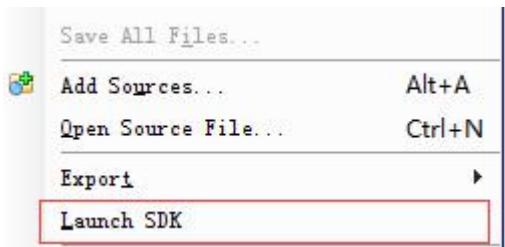
Step1: File->Export->Export Hardware



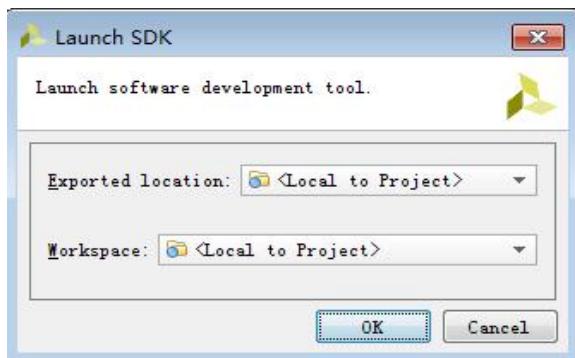
Step2: 勾选 Include bitstream 直接单击 OK



Step3:File->Launch SDK 加载到 SDK



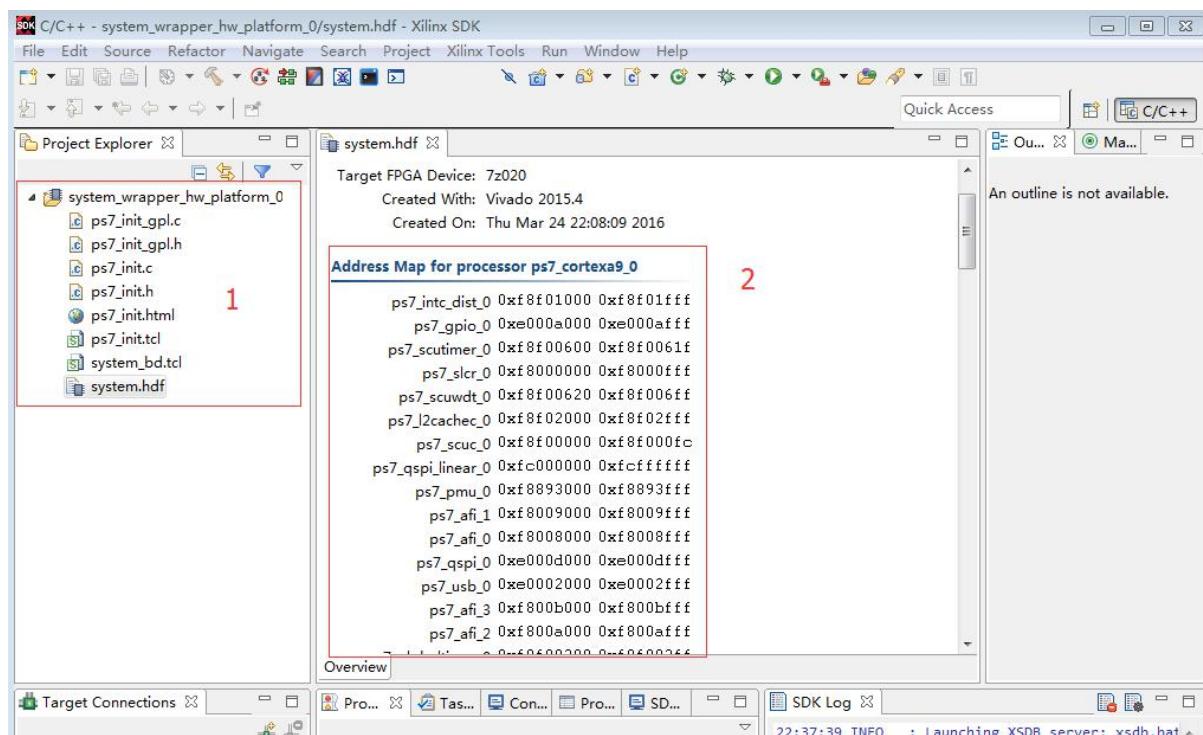
Step4:单击 OK



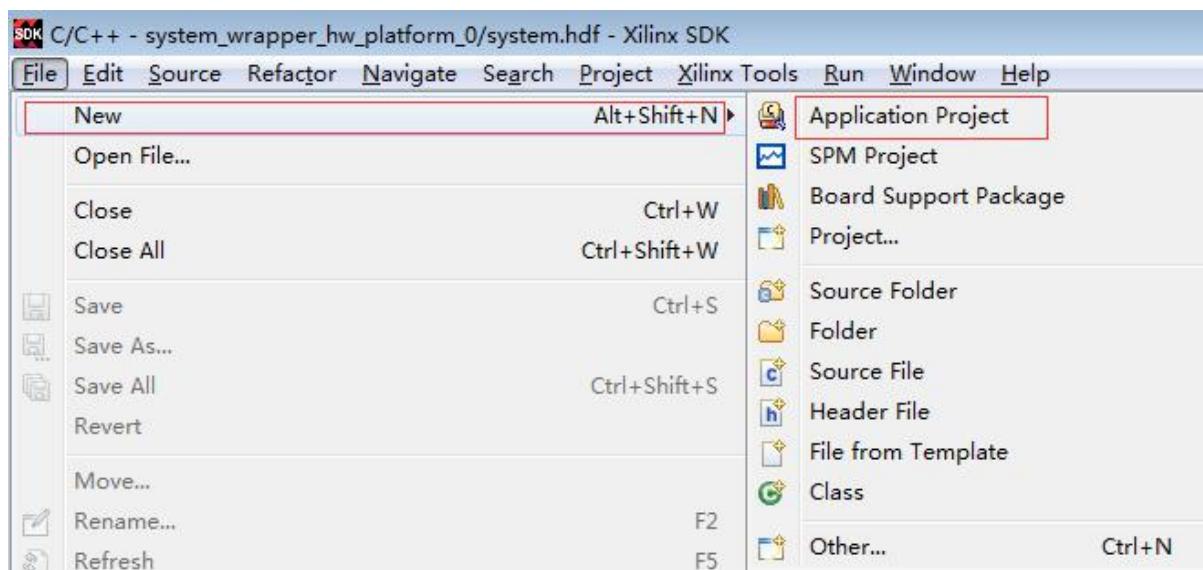
1.5 Hello World实验

Step1:导出完成后如下图

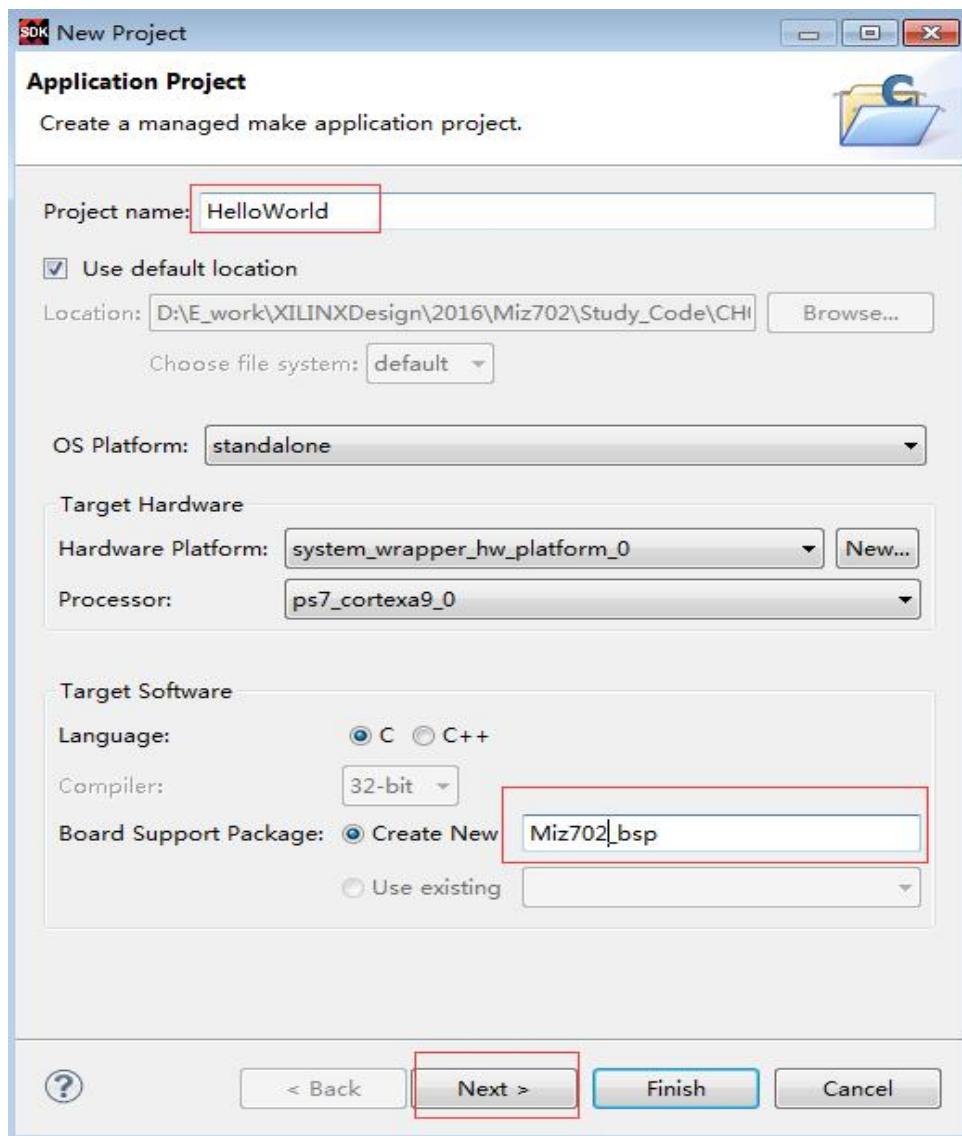
- 1、硬件部分，这部分就是从 VIVADO 定制好的 SOC 硬件
- 2、这部分是硬件的地址空间分配



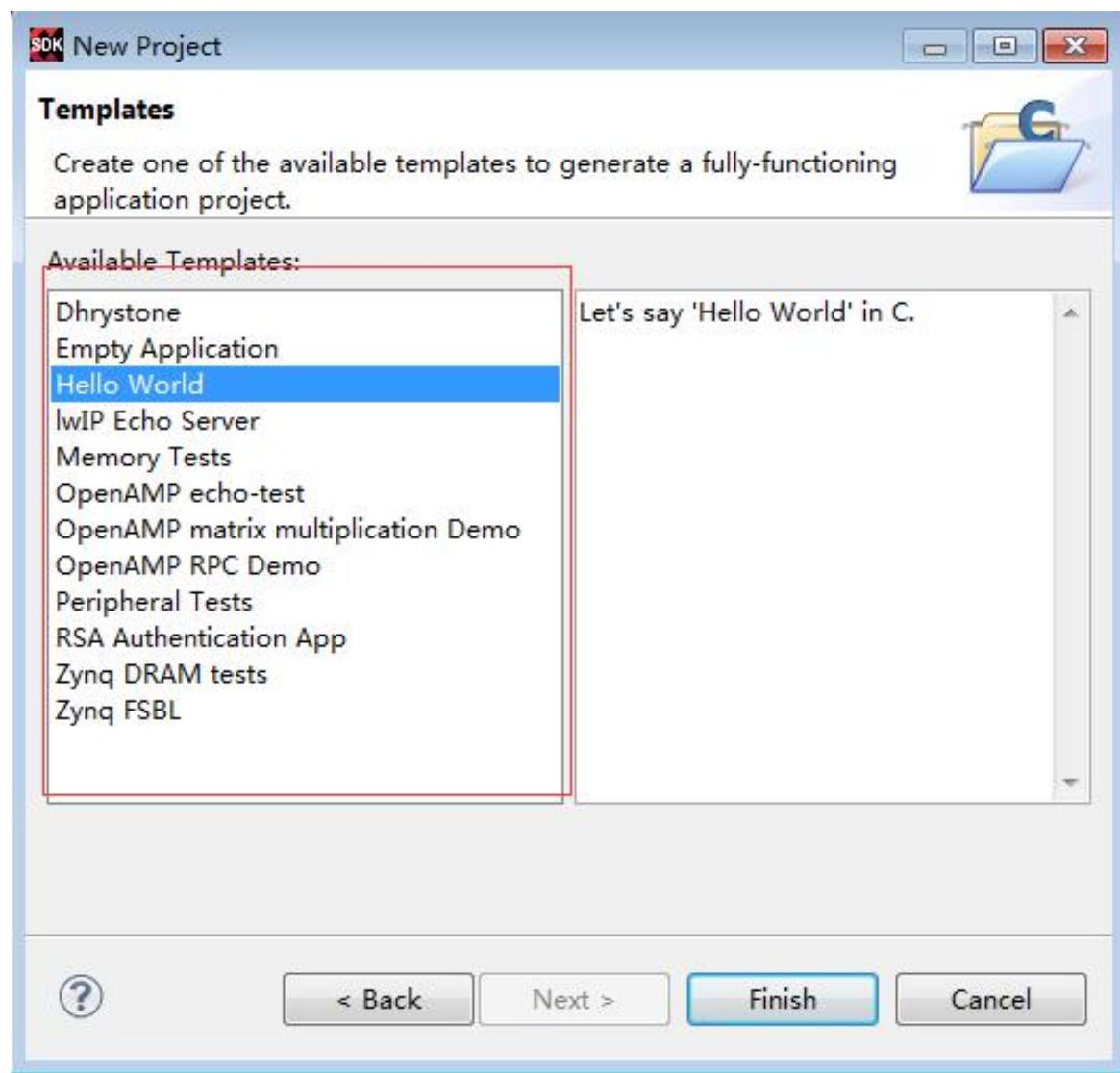
Step2:选择 File->New->Application Project



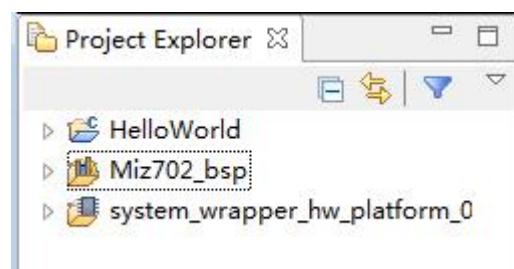
Step3:工程命名为 HelloWorld,创建的 bsp 包取名为 Miz702_bsp,然后单击 NEXT



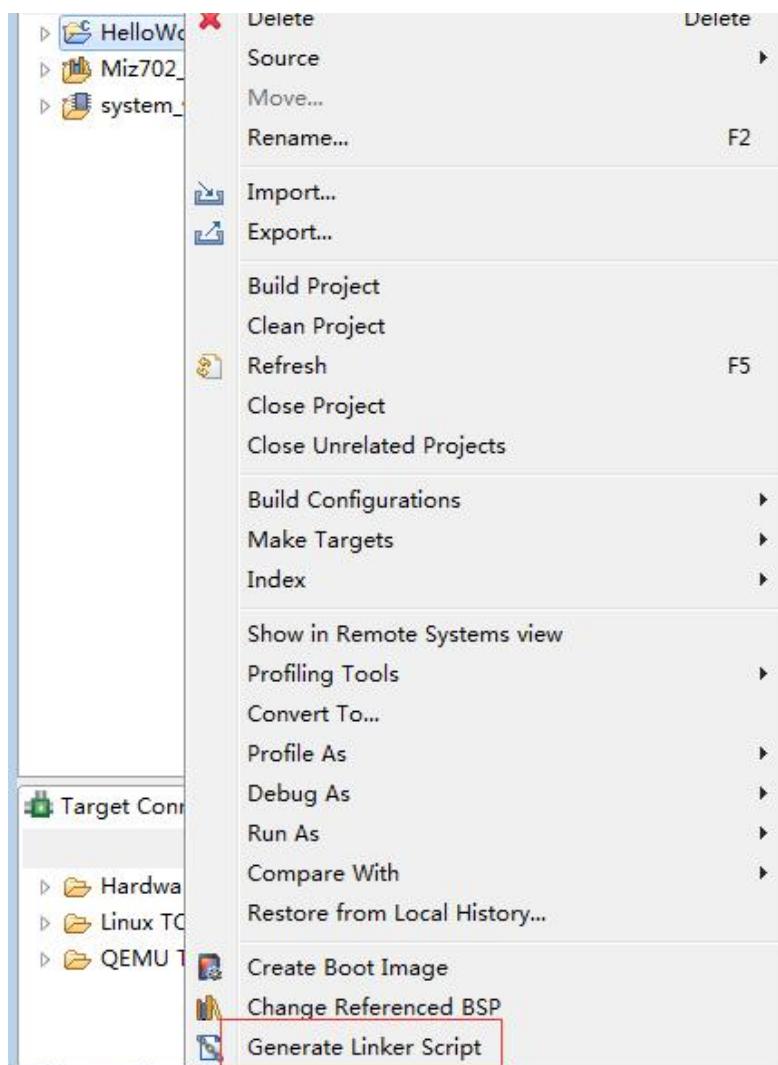
Step4: 系统里面有很多自带的测试程序，本次就用自带的 Helloworld 程序做测试，单击 Finish



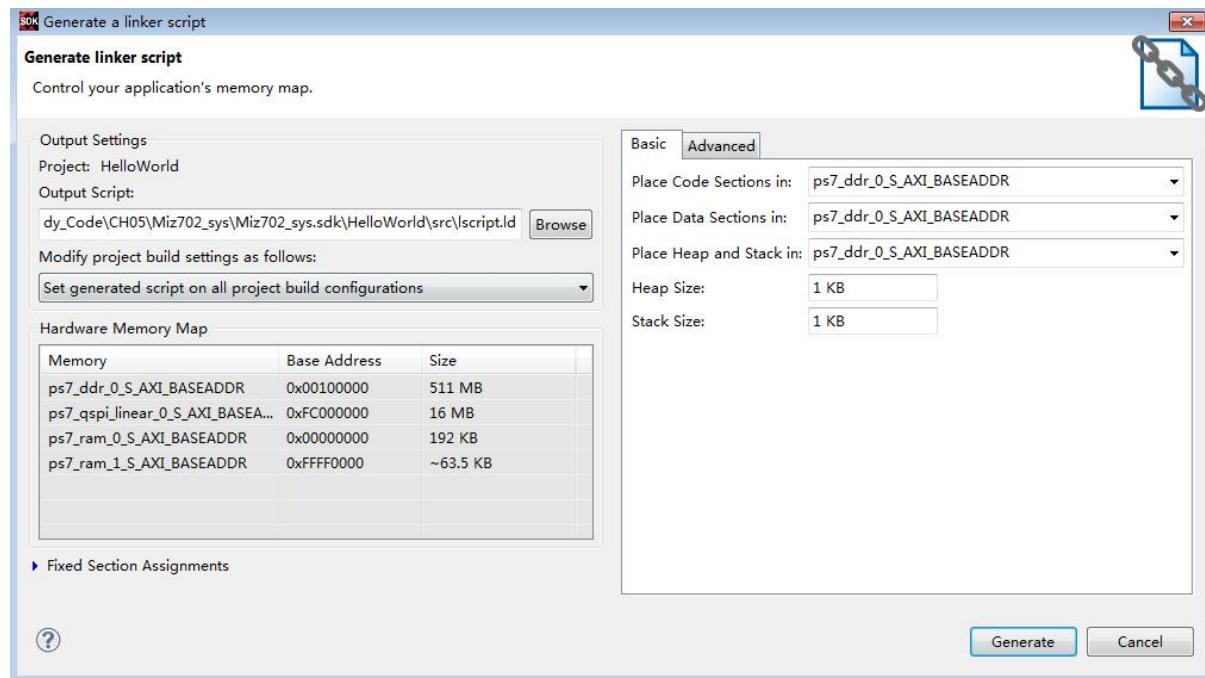
Step5: 完成后



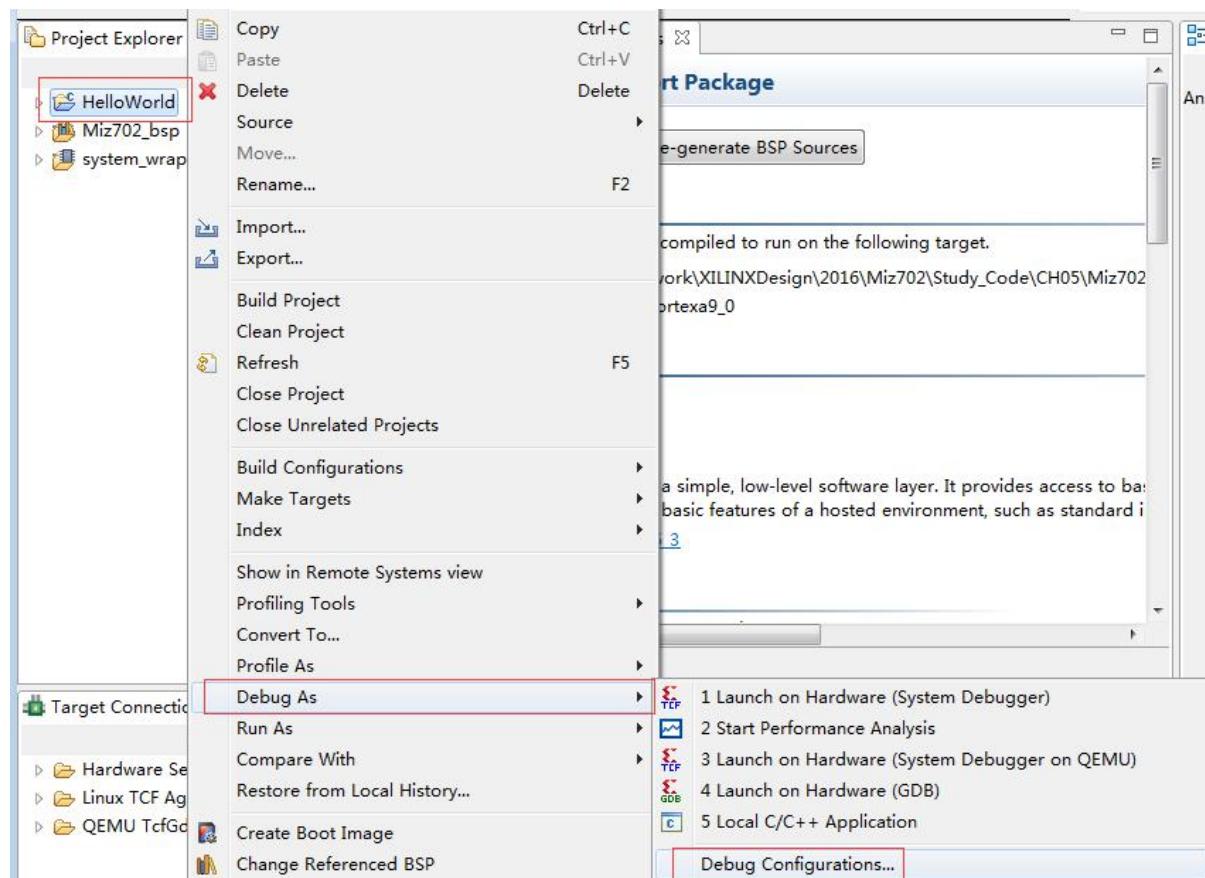
Step6:右击 HelloWorld->Generate linker Script



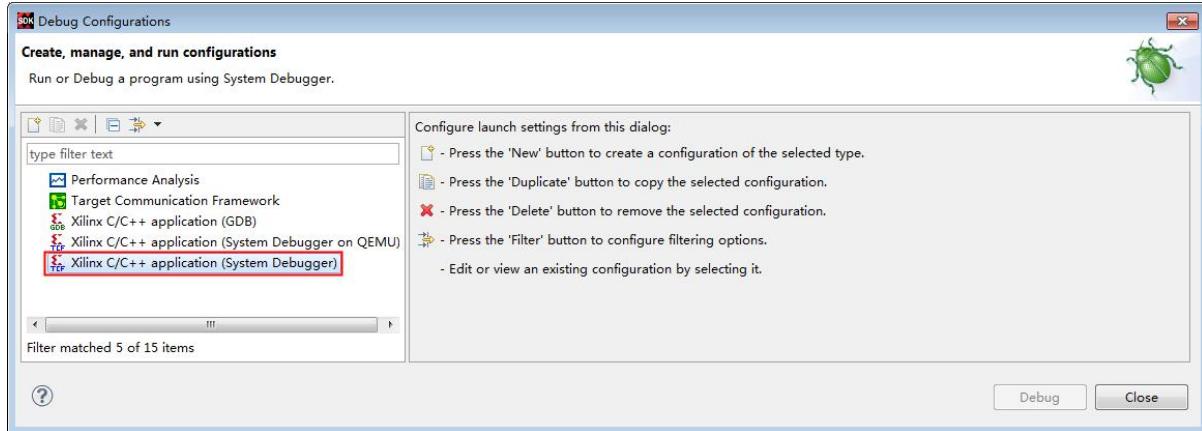
Step7:可以看到所有可用内存的情况，代码、数据、堆栈运行所在内存的情况。不做人为改动，关闭。



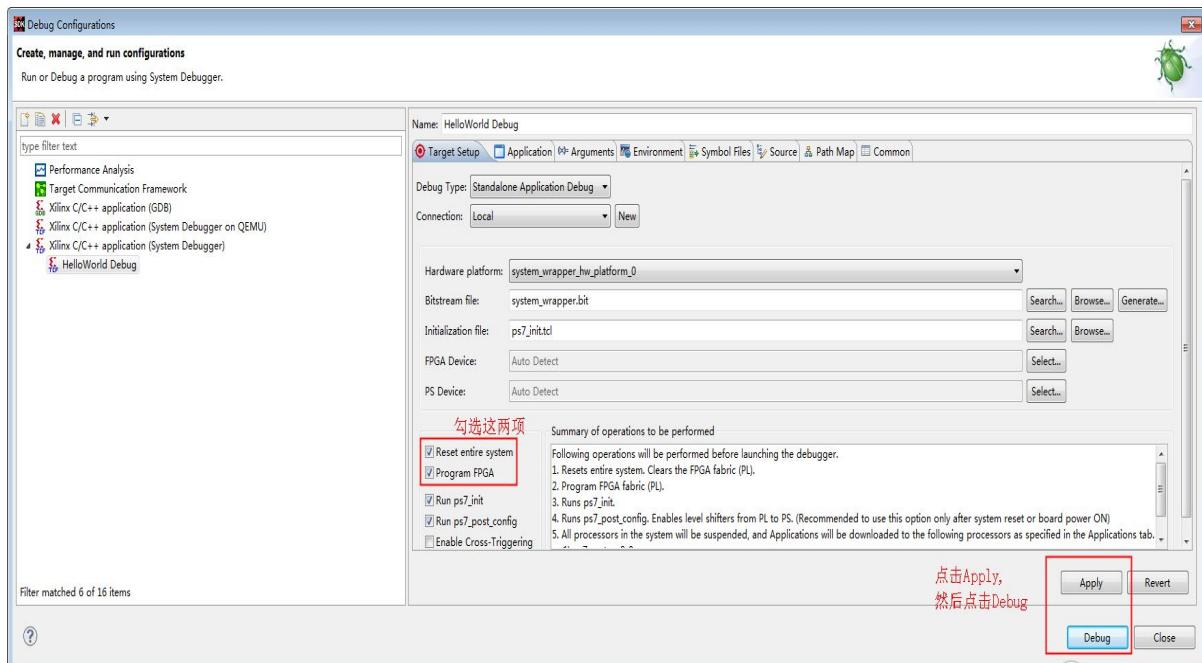
Step8:右击 HelloWorld->



Step9: 双击这个位置新建

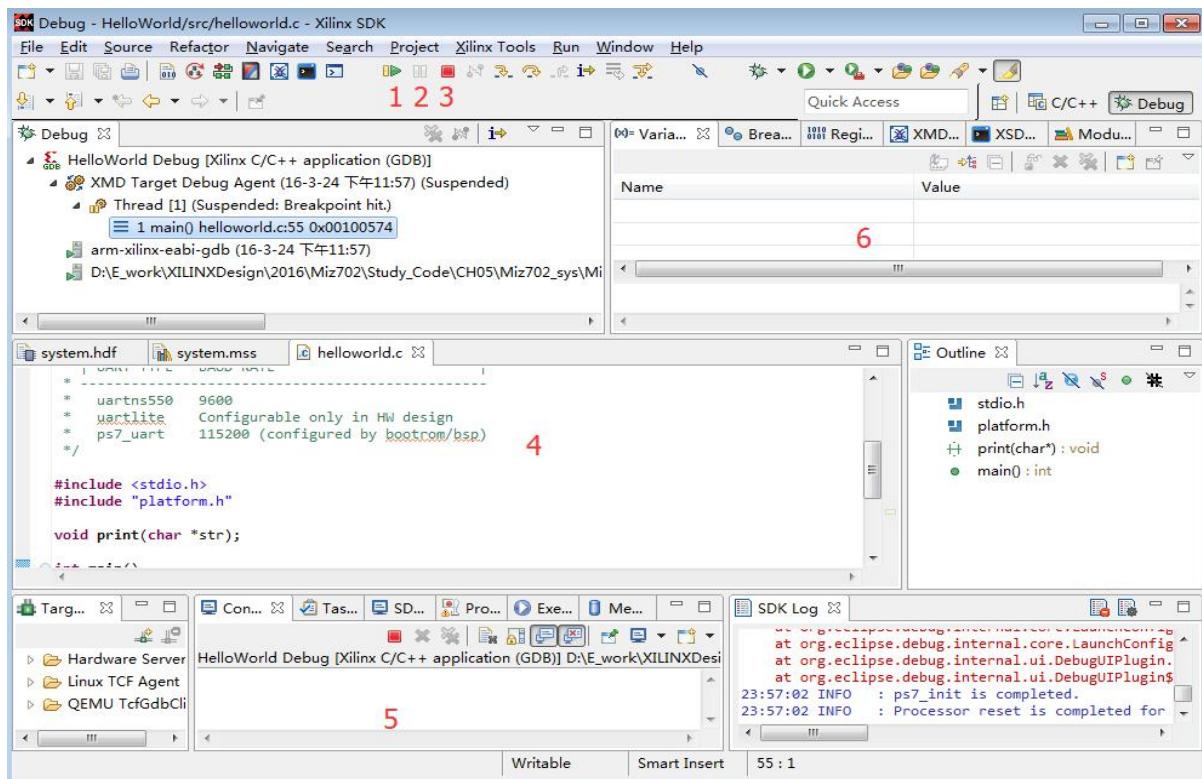


Step10: 然后进行如下设置

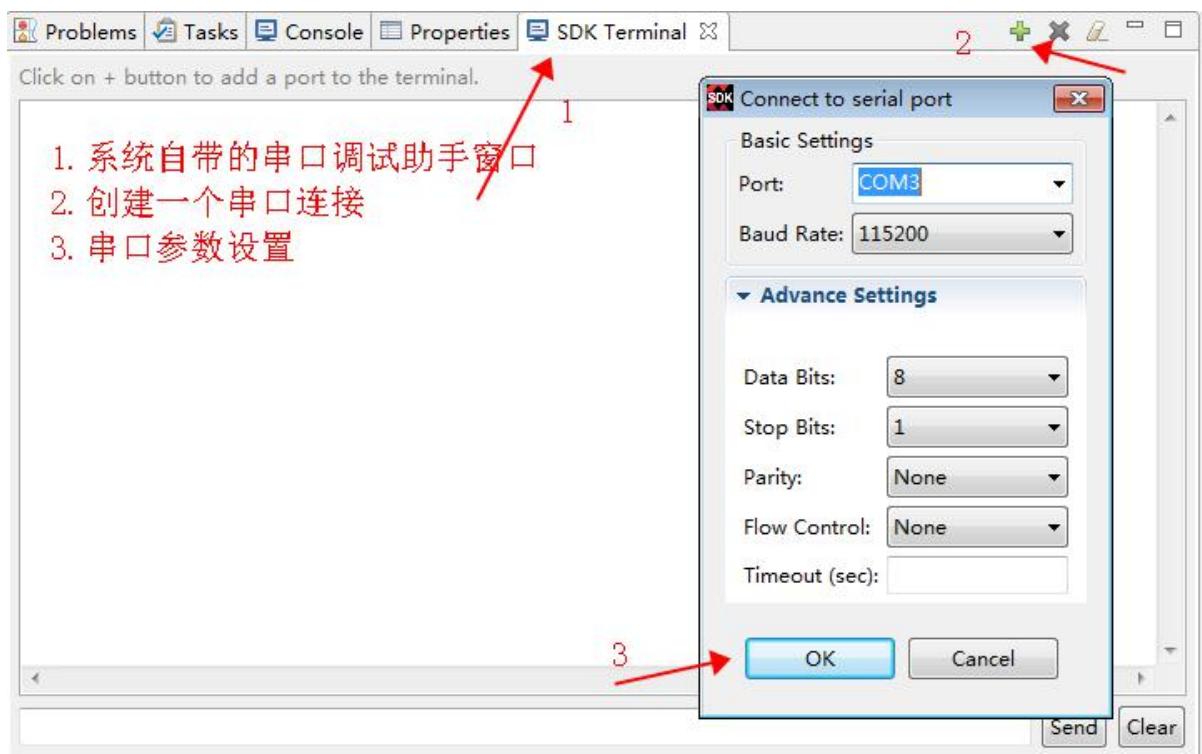


Step11: 进入 SDK 调试界面

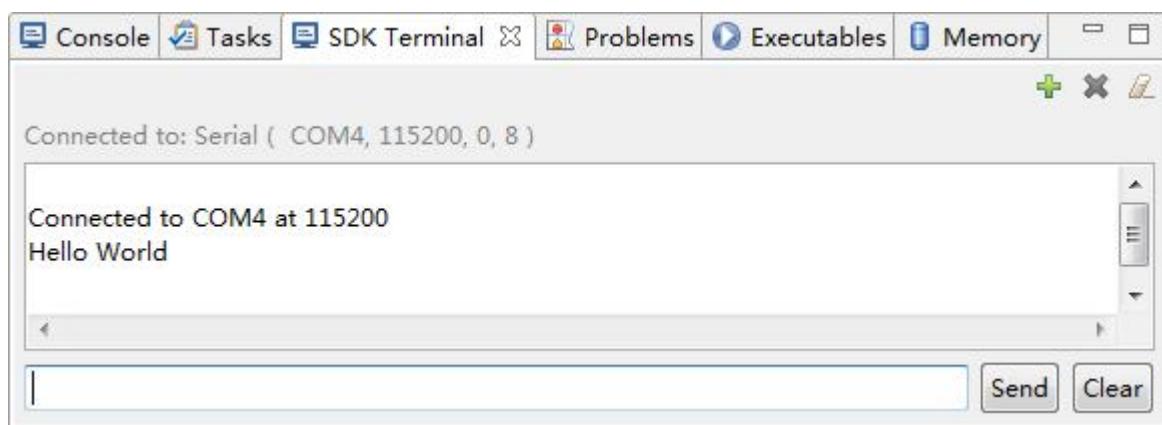
1、启动 2、暂停 3、停止 4、代码 5、信息控制台 6、调试变量



Step12:启用系统自带的串口调试助手，进行相关的设置。

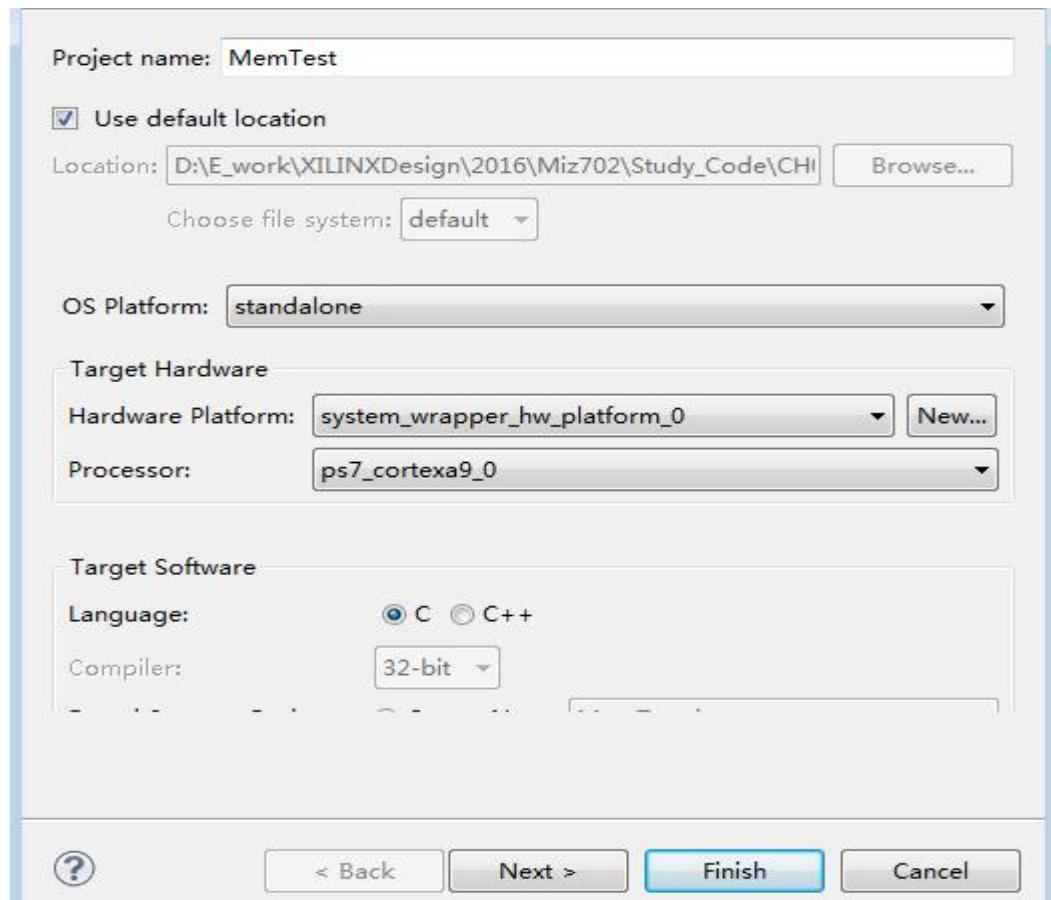


Step13:单击运行输出结果

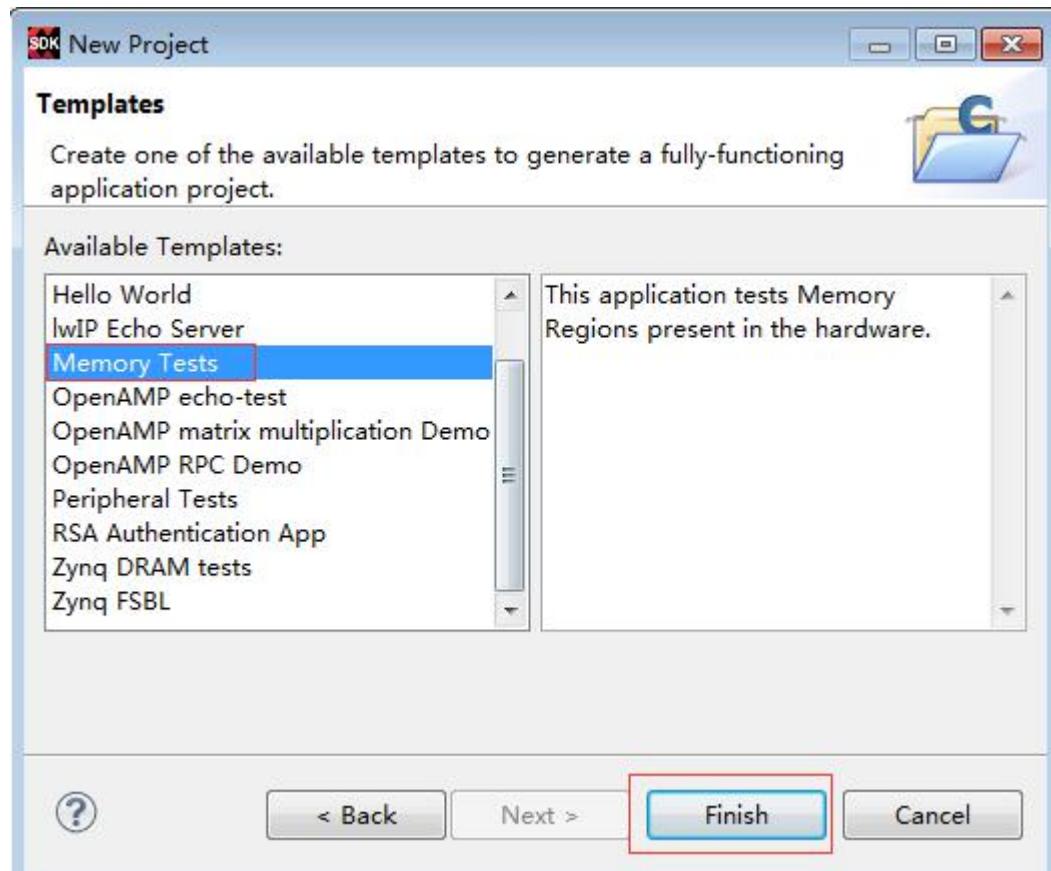


1.6 MemTest 内存测试程序

Step1:新建一个名为 MemTest 的工程



Step2: 任然采用自带的测试函数测试

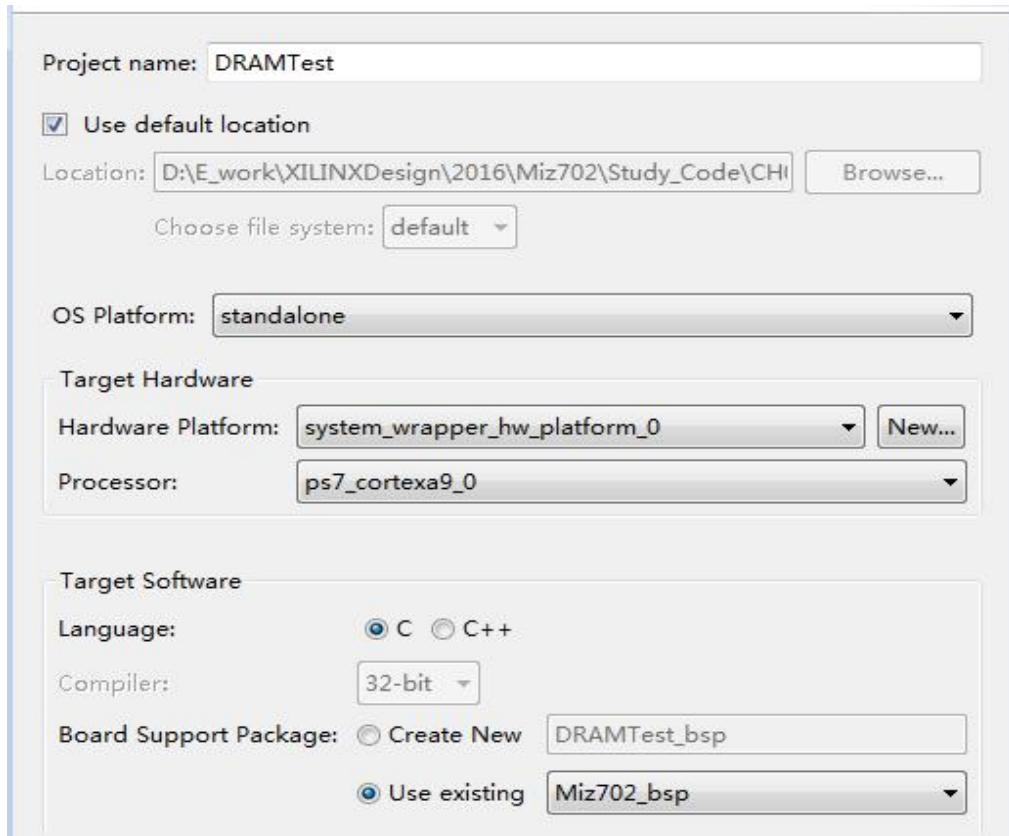


Step3: 测试结果

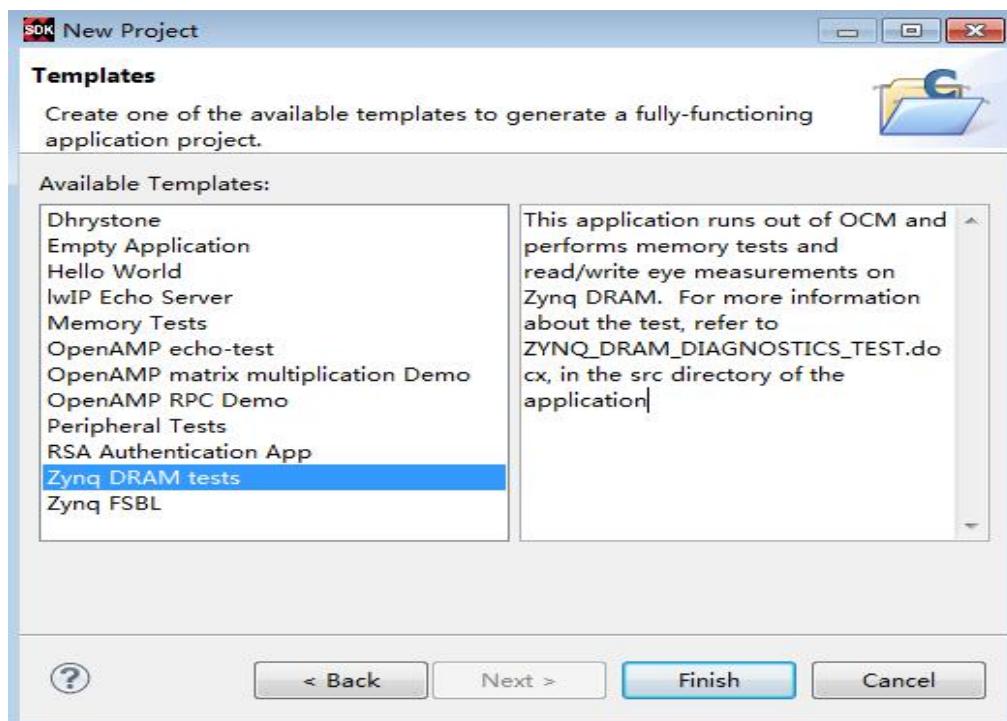
```
--Starting Memory Test Application--  
NOTE: This application runs with D-Cache disabled. As a result, cacheline requests wi  
Testing memory region: ps7_ddr_0  
    Memory Controller: ps7_ddr  
        Base Address: 0x00100000  
        Size: 0x1ff00000 bytes  
        32-bit test: PASSED!  
        16-bit test: PASSED!  
        8-bit test: PASSED!  
Testing memory region: ps7_ram_1  
    Memory Controller: ps7_ram  
        Base Address: 0xfffff0000  
        Size: 0x0000fe00 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



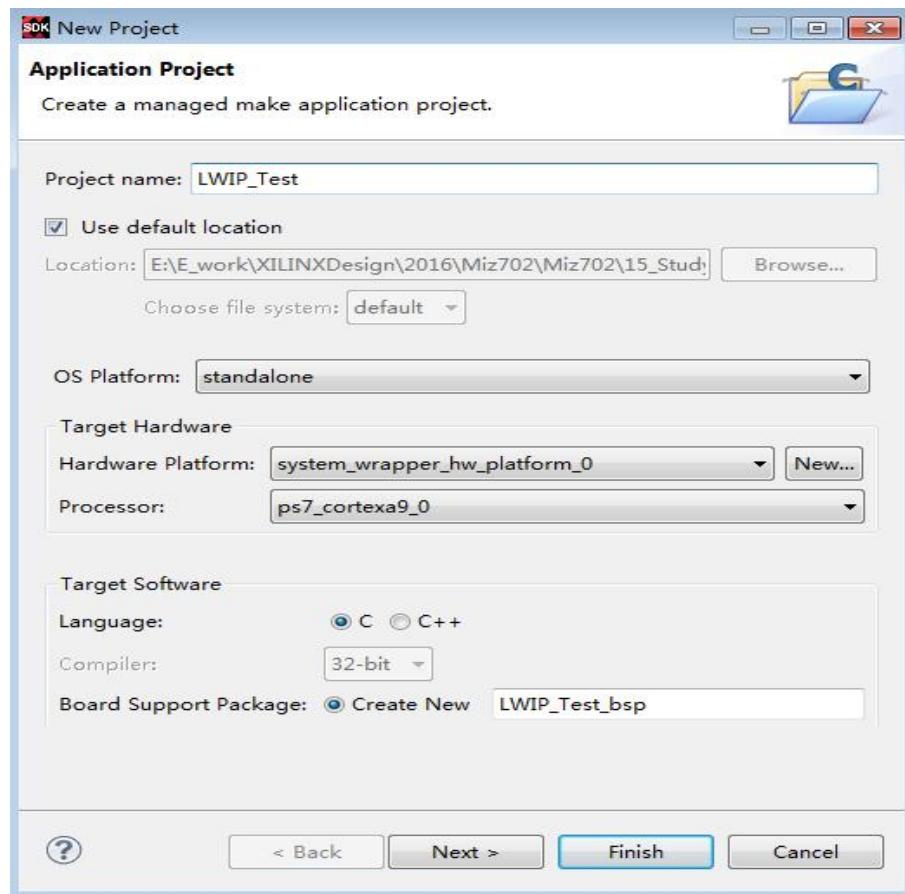
Step4: 测试结果

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）

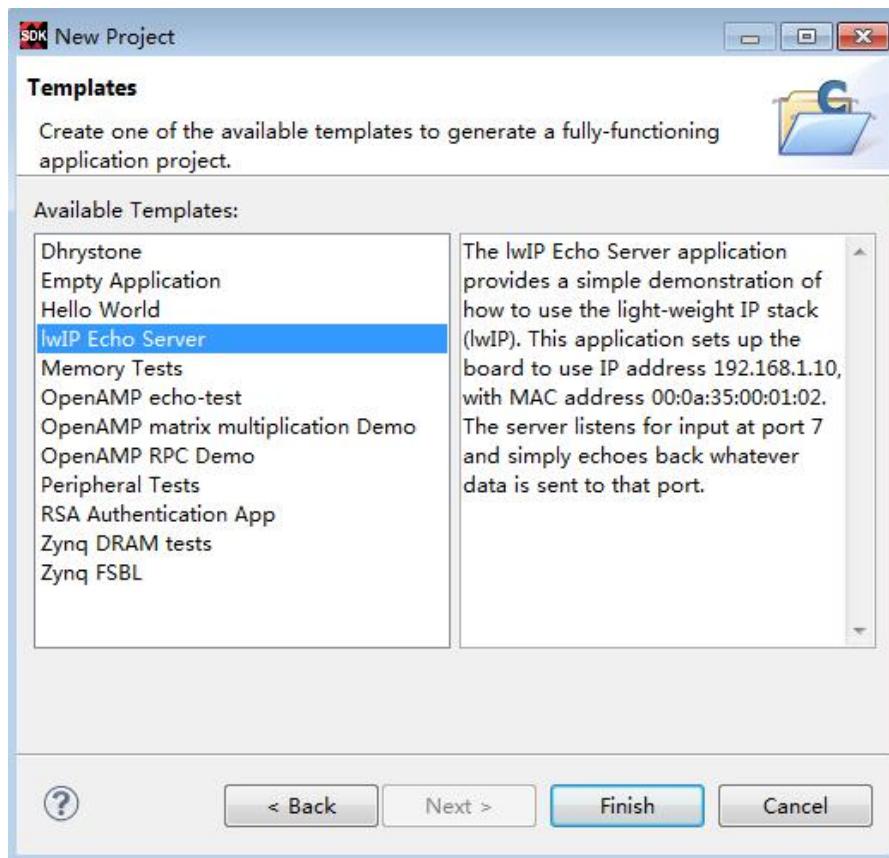
```
----- ZYNQ DRAM DIAGNOSTICS TEST -----
Select one of the options below:
## Memory Test ##
Bus Width = 32, XADC Temperature = 47.0328
's' - Test 1MB length from address 0x100000
'1' - Test 32MB length from address 0x100000
'2' - Test 64MB length from address 0x100000
'3' - Test 128MB length from address 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

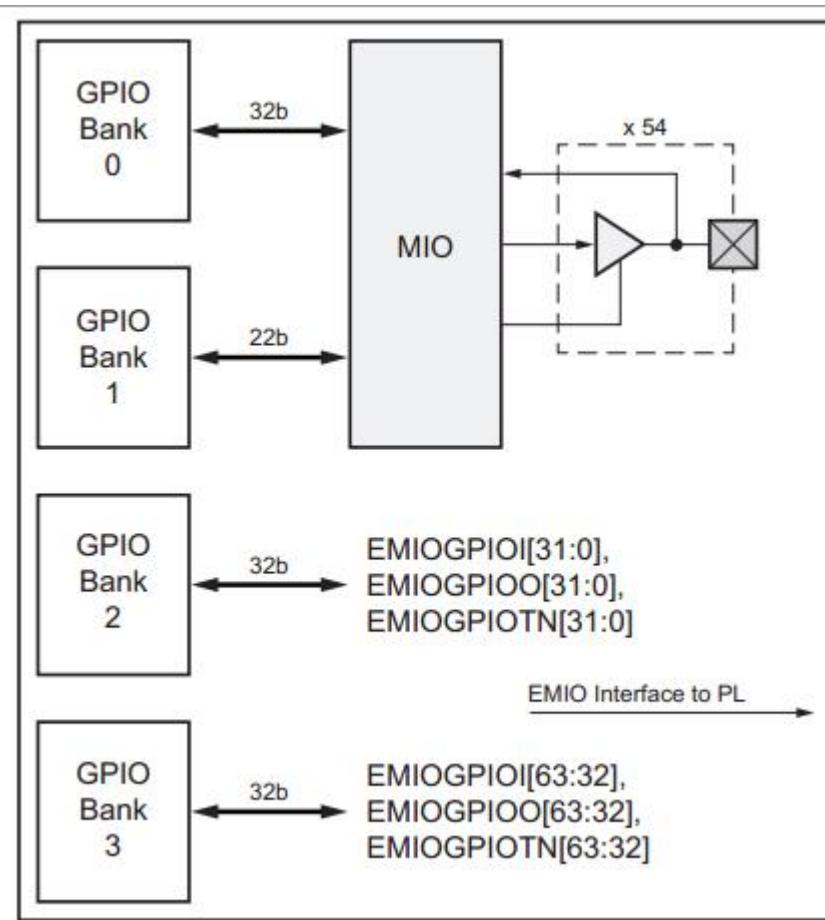
本章详细讲解了定制一个 SOC 最小系统，并且运行了自带的 HelloWorld 工程、MemTest 内存测试工程、DRAMTest 内存测试工程、LWIP 网络协议工程对千兆网口测试。本章让初学者可以搭建一个最小的 SOC 系统，并且教会读者利用软件自动的工程对 SOC 的基本外设进行测试。希望大家多多操作，熟练掌握如何创建 VIVADO 工程，懂得如何根据自己的硬件平台配置 ZYNQ CPU IP，下章我们将不在对这些进行详细的讲解。

S02_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

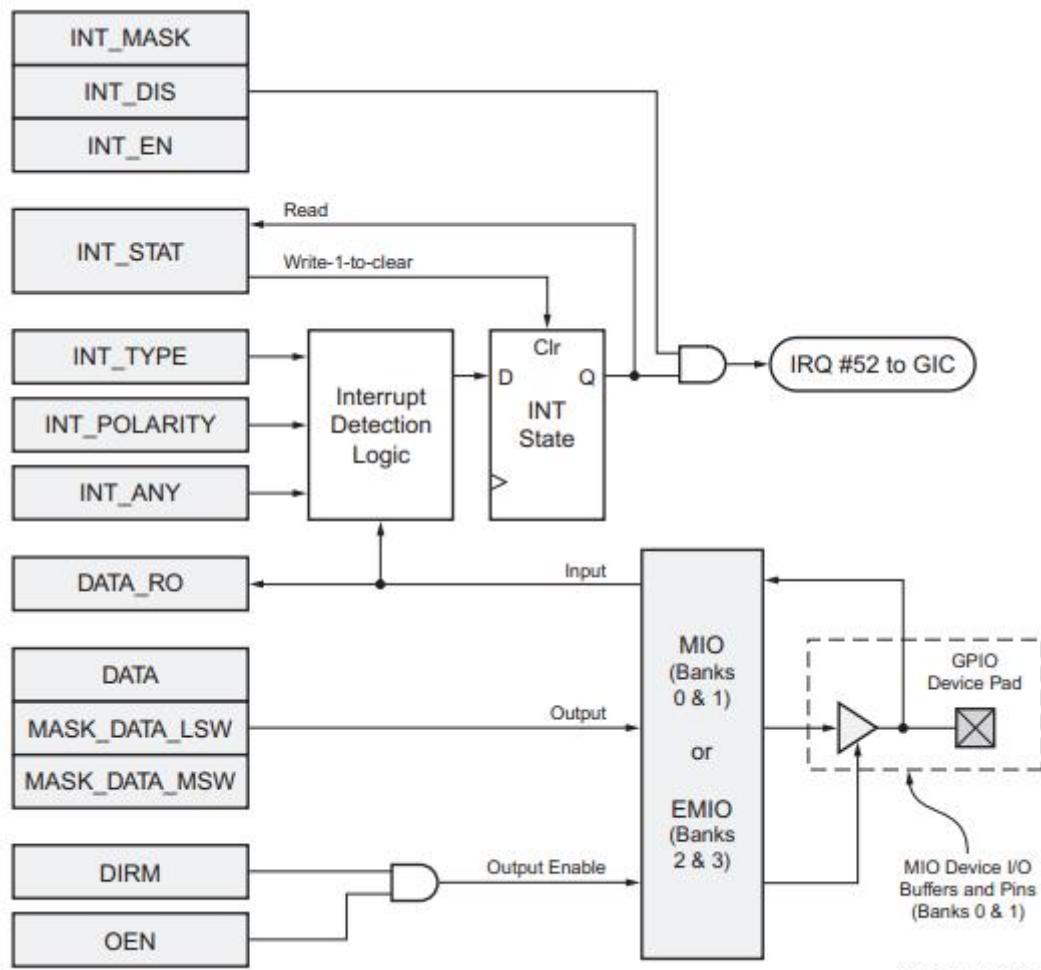
我们在 SDK 下操作的时候底层都是对这些寄存器的操作，具体的相关参数请参考技术手册 ug585-Zynq-7000-TRM.pdf

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

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

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

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。

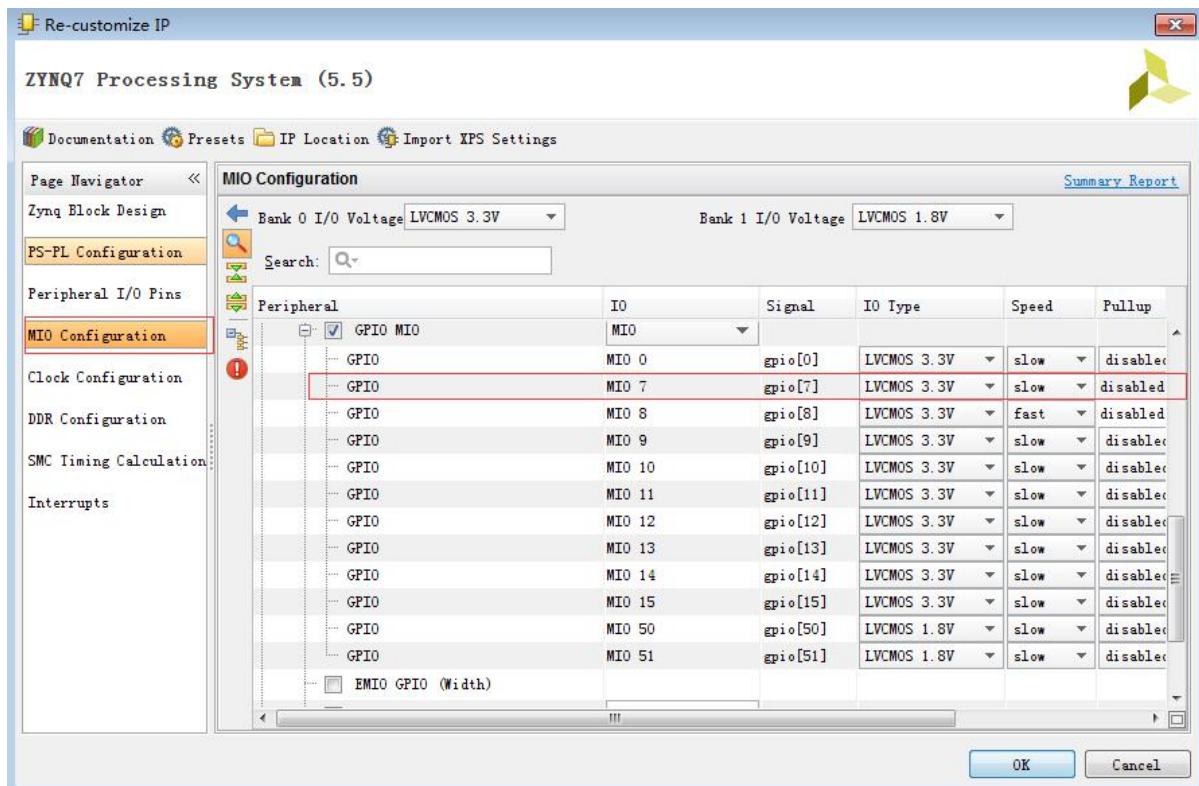
2.3 ZYNQ 核的添加及配置

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。

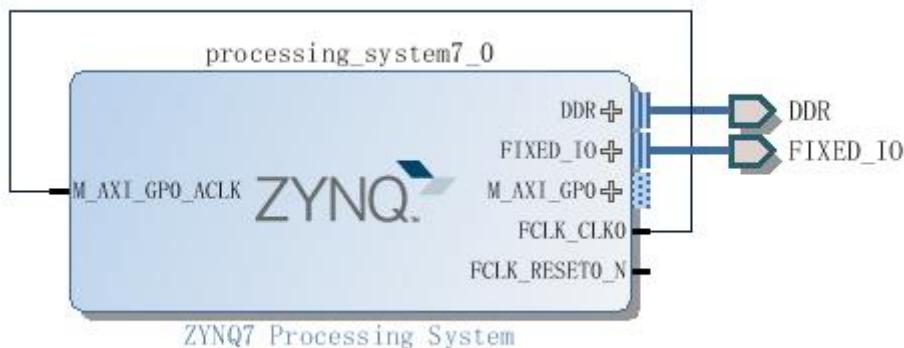
Step2：单击 Create Block Design，创建一个 BD 文件，并命名为 System。

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。

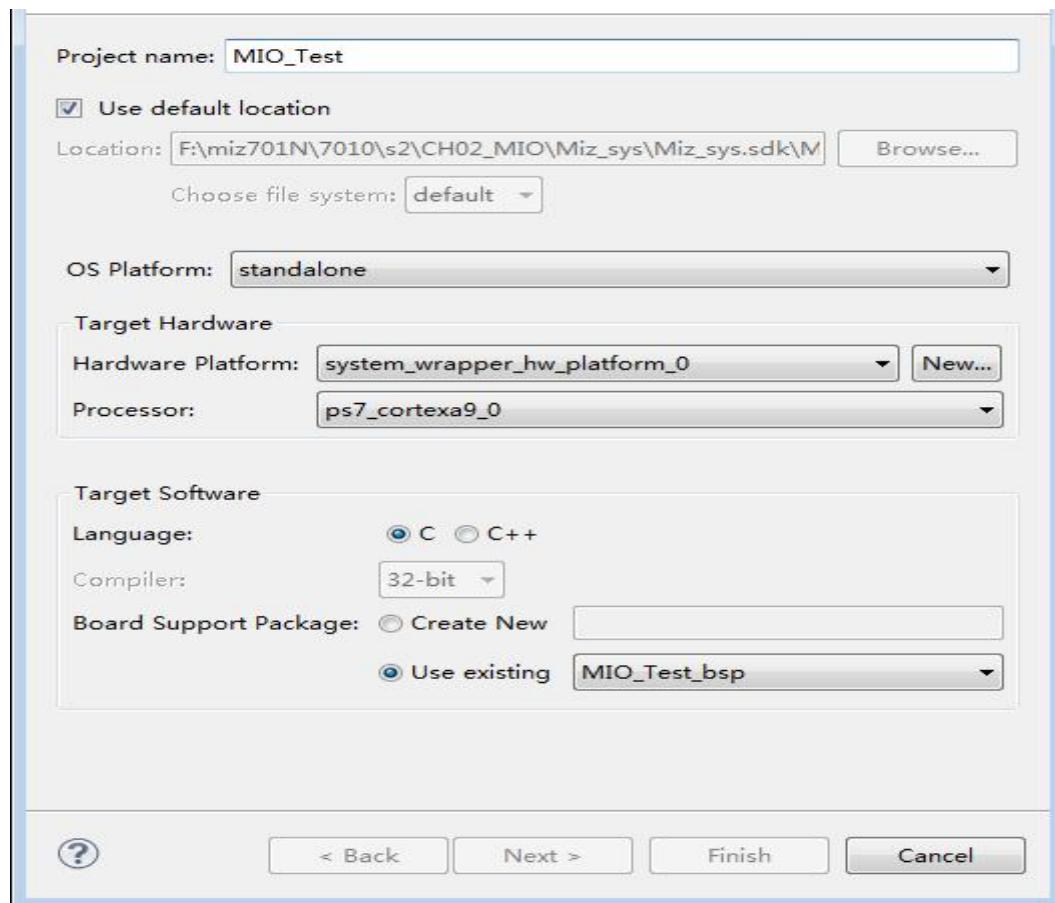
Step8: File->Export->Export Hardware。

Step9: 勾选 Include bitstream 直接单击 OK。

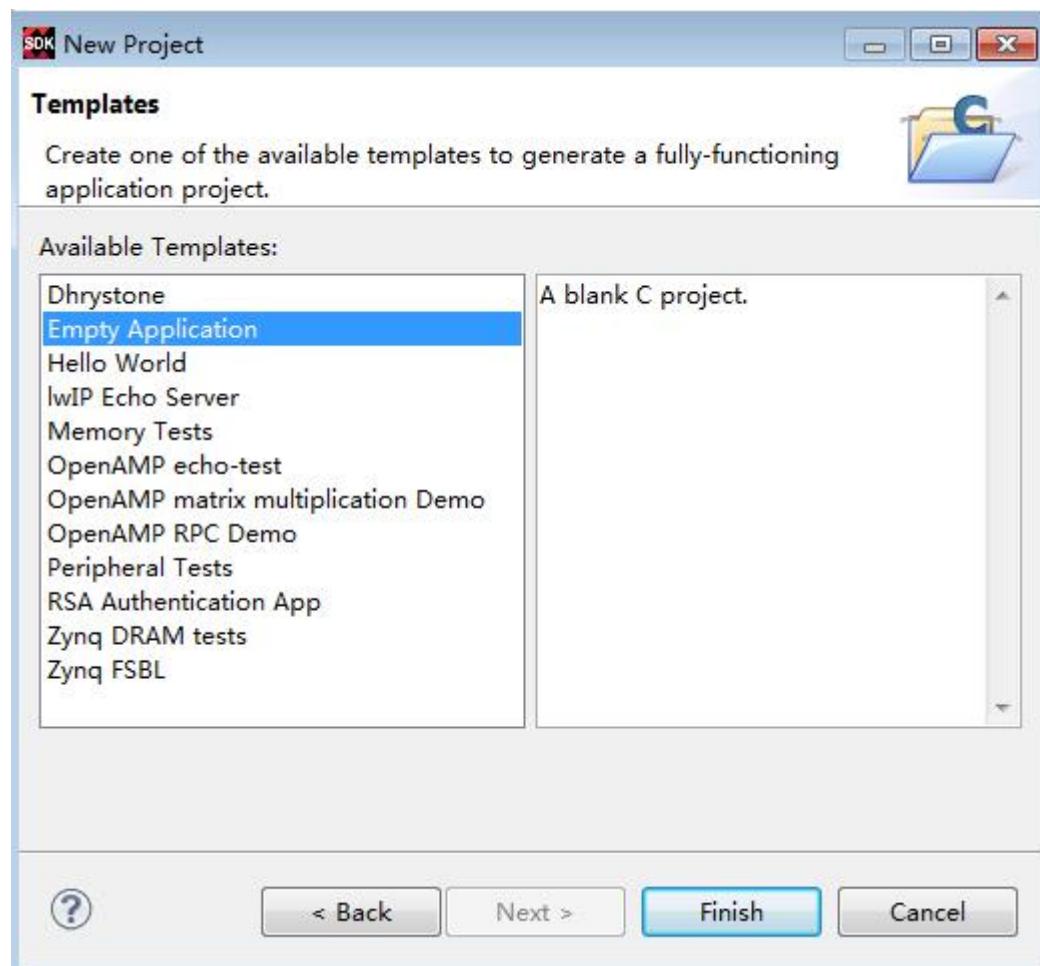
Step10: File->Launch SDK 加载到 SDK, 单击 OK。

2.4 新建 LED_Flash SDK 工程

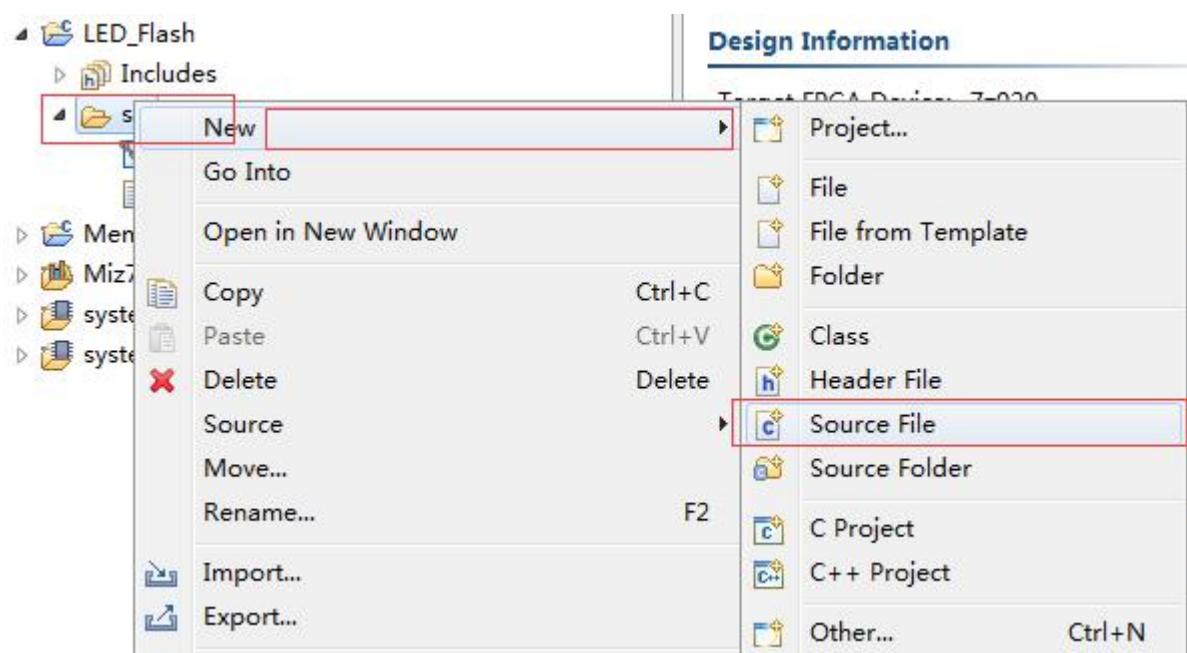
Step1: 在 SDK 界面中, 新建一个名为 MIO_Test 的工程



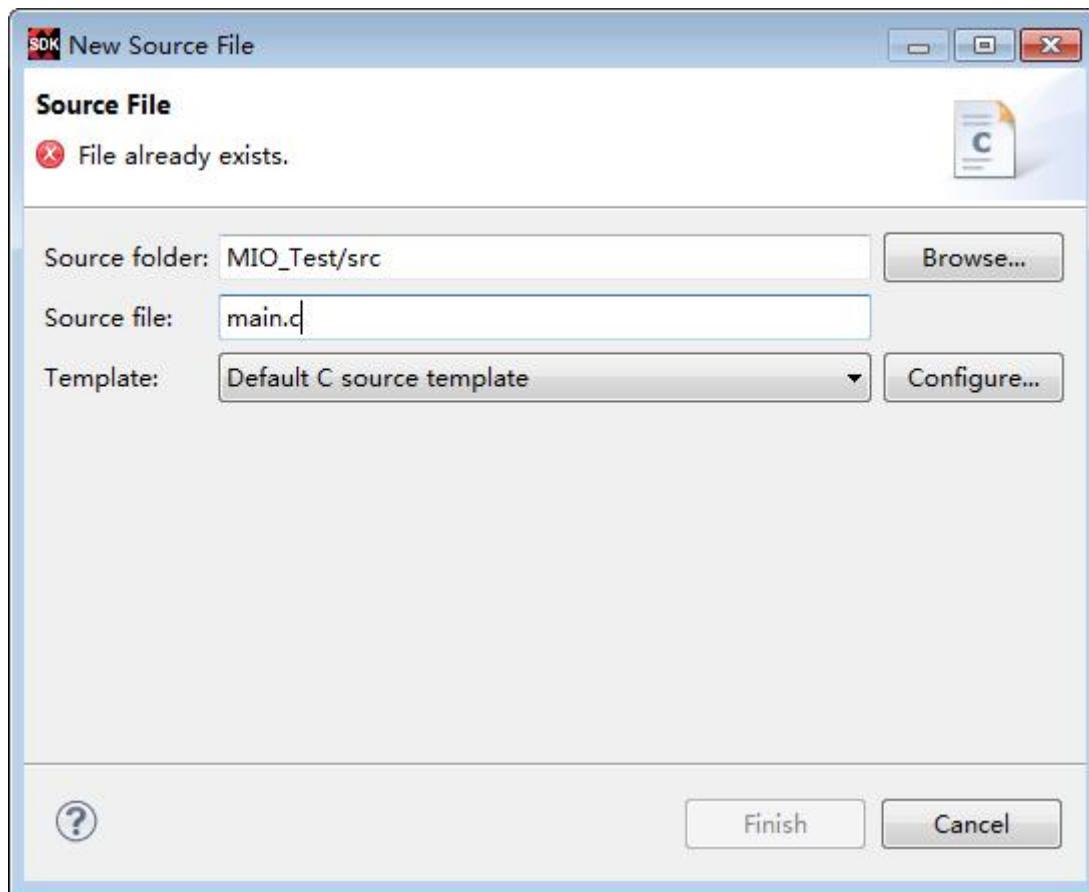
Step2: 建立一个空的工程



Step3: 新建一个 C 的源文件



Step4: 取名为 main.c



接下来就向 main.c 中添加内容了，之前讲过，其中 MI07 接到了 LD9 这个灯上，接下来我们利用程序让他闪起来。

```
#include "xgpiops.h"
#include "sleep.h"
int main()
{
    static XGpioPs psGpioInstancePtr;
    XGpioPs_Config* GpioConfigPtr;
    int iPinNumber= 7;           //LD9连接的是MIO7
    u32 uPinDirection = 0x1;    //1表示输出, 0表示输入
    int xStatus;

    //--MIO的初始化
    GpioConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
    if(GpioConfigPtr == NULL)
        return XST_FAILURE;

    xStatus = XGpioPs_CfgInitialize(&psGpioInstancePtr,GpioConfigPtr,
GpioConfigPtr->BaseAddr);
```

```
if(XST_SUCCESS != xStatus)
    print(" PS GPIO INIT FAILED \n\r");
//--MIO的输入输出操作
XGpioPs_SetDirectionPin(&psGpioInstancePtr, iPinNumber,uPinDirection); //配置
MIO输出方向
XGpioPs_SetOutputEnablePin(&psGpioInstancePtr, iPinNumber,1); //配置MIO的第7位
输出
while(1)
{
    XGpioPs_WritePin(&psGpioInstancePtr, iPinNumber, 1); //点亮MIO的第7位输出1
    usleep(500000); //延时
    XGpioPs_WritePin(&psGpioInstancePtr, iPinNumber, 0); //熄灭MIO的第7位输出0
    usleep(500000); //延时
}
/*********************************************
while(1)
{
    XGpioPs_WriteReg(0xE000A000,0x00000000, 0xFF7FFFFF&0xFFFF0080);
    usleep(500000); //延时
    XGpioPs_WriteReg(0xE000A000,0x00000000, 0xFF7FFFFF&0xFFFF0000);
    usleep(500000); //延时
}
*****************************************/
return 0;
}
```

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。

```

static XGpioPs psGpioInstancePtr;
XGpioPs
int iPi
u32 uPi
*/
int xSt
typedef struct {
    XGpioPs_Config GpioConfig; /* Device configuration */
    u32 IsReady; /* Device is initialized and ready */
    XGpioPs_Handler Handler; /* Status handlers for all banks */
    void *CallBackRef; /* Callback ref for bank handlers */
    u32 Platform; /* Platform data */
    u32 MaxPinNum; /* Max pins in the GPIO device */
    u8 MaxBanks; /* Max banks in a GPIO device */
} XGpioPs;
if(XST_SUCCESS == xStatus)
    if(XST_SUCCESS == xGpioPsInit(&psGpioInstancePtr, BaseAddr))
        pri
//--MIO

```

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 0xFFFFC00000
#define XPAR_PS7_RAM_1_S_AXI_HIGHADDR 0xFFFFFFFF

/* 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 0xF8000000
#define XPAR_PS7_SLCR_0_S_AXI_HIGHADDR 0xF800FFF

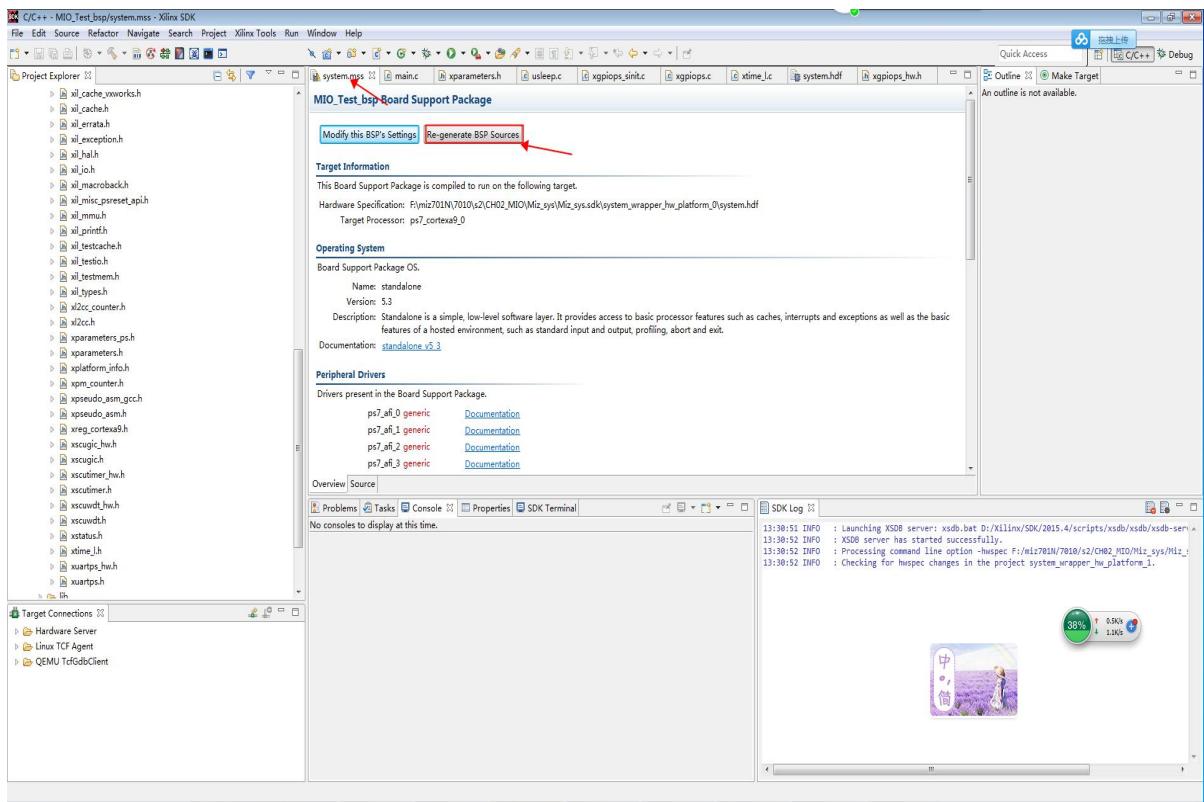
/**************************/
/* Definitions for driver GPIOPS */
#define XPAR_XGPIOPS_NUM_INSTANCES 1

/* 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
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。

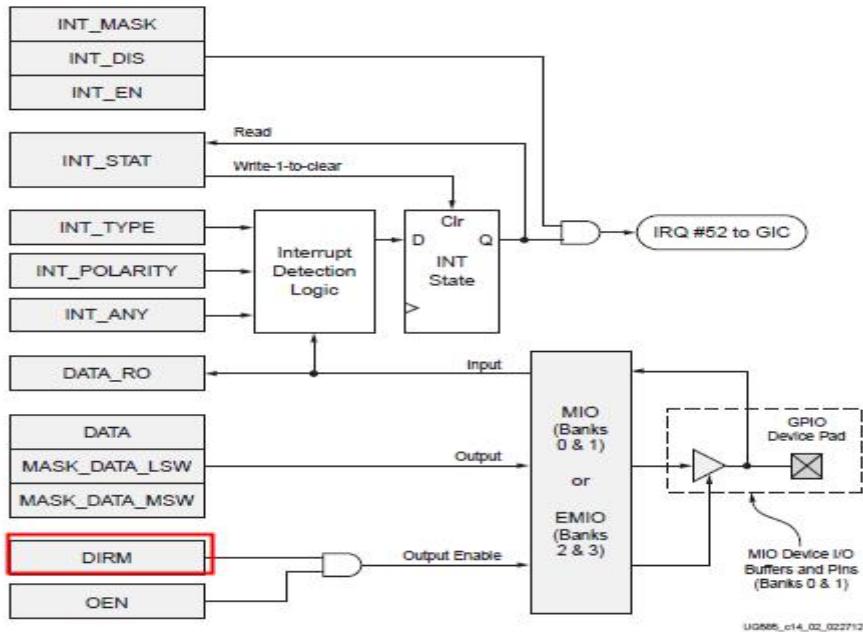


Figure 14-2: GPIO Channel

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，我们就不在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)0x0001;
    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
MASK DATA_0_LSW	0x00000000	32	mixed	x	Maskable Output Data (GPIO Bank0, MIO, Lower 16bits)
MASK DATA_0_MSB	0x00000004	32	mixed	x	Maskable Output Data (GPIO Bank0, MIO, Upper 16bits)
MASK DATA_1_LSW	0x00000008	32	mixed	x	Maskable Output Data (GPIO Bank1, MIO, Lower 16bits)
MASK DATA_1_MSB	0x0000000C	22	mixed	x	Maskable Output Data (GPIO Bank1, MIO, Upper 6bits)
MASK DATA_2_LSW	0x00000010	32	mixed	0x00000000	Maskable Output Data (GPIO Bank2, EMIO, Lower 16bits)
MASK DATA_2_MSB	0x00000014	32	mixed	0x00000000	Maskable Output Data (GPIO Bank2, EMIO, Upper 16bits)
MASK DATA_3_LSW	0x00000018	32	mixed	0x00000000	Maskable Output Data (GPIO Bank3, EMIO, Lower 16bits)
MASK DATA_3_MSB	0x0000001C	32	mixed	0x00000000	Maskable Output Data (GPIO Bank3, EMIO, Upper 16bits)
DATA_0	0x00000040	32	rw	x	Output Data (GPIO Bank0, MIO)
DATA_1	0x00000044	22	rw	x	Output Data (GPIO Bank1, MIO)
DATA_2	0x00000048	32	rw	0x00000000	Output Data (GPIO Bank2, EMIO)
DATA_3	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[~](80000) h，也就是(FFF7FFFF) 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。

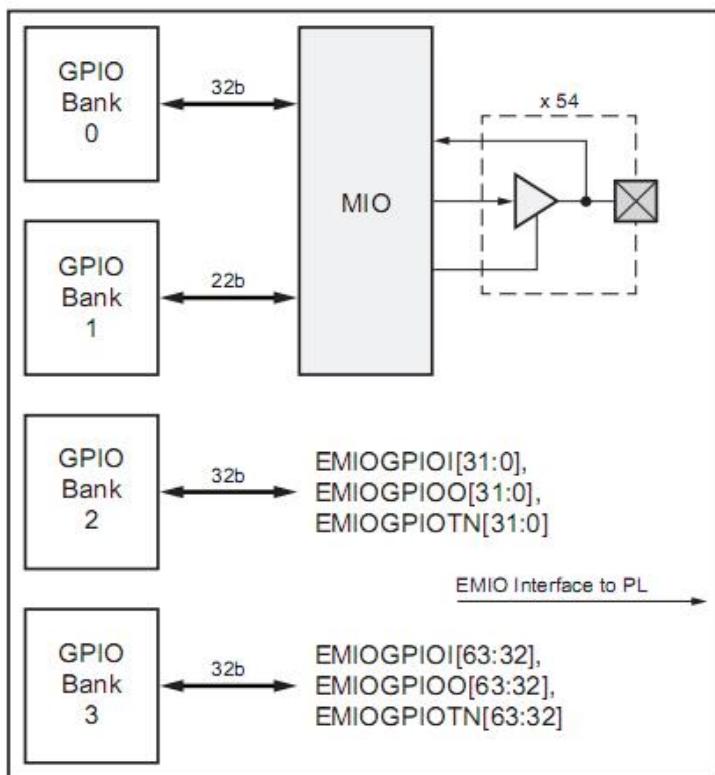
2.6 本章小结

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。

S02_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。

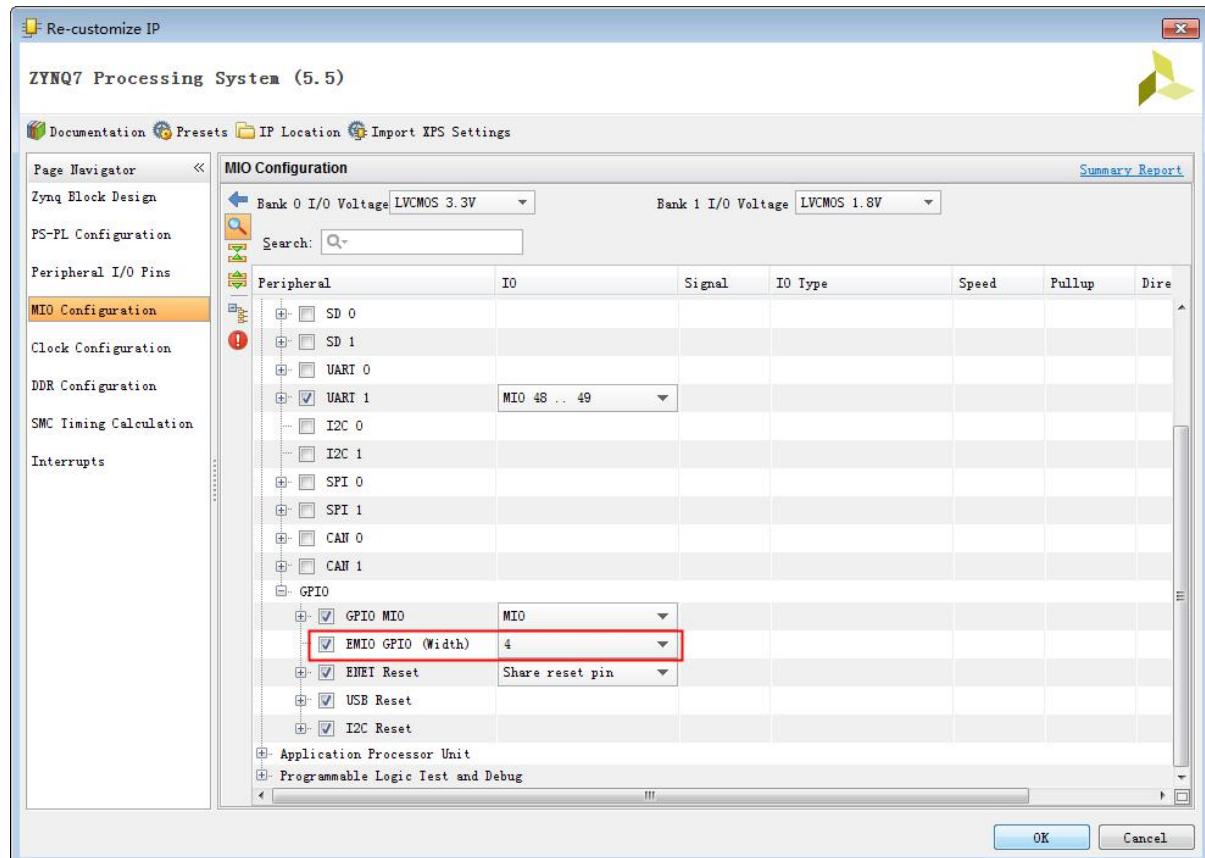
3.3 创建 VIVADO 工程

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。

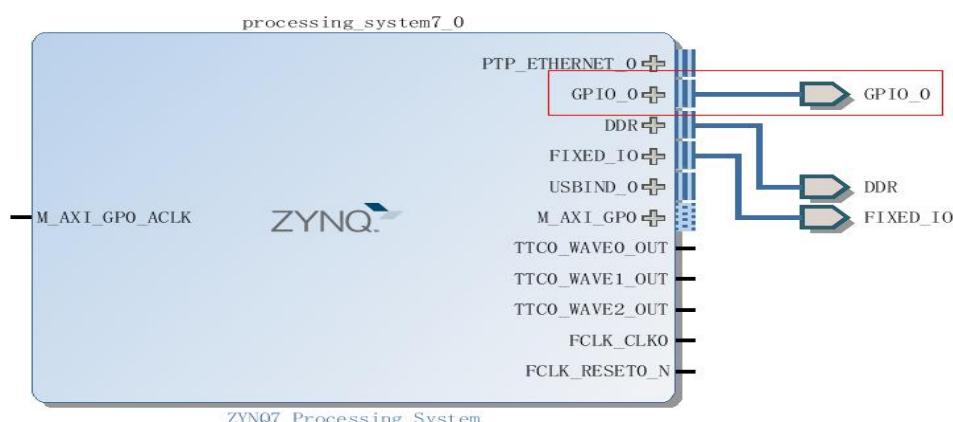
Step2: 创建一个 BD 文件，并命名为 system。

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。

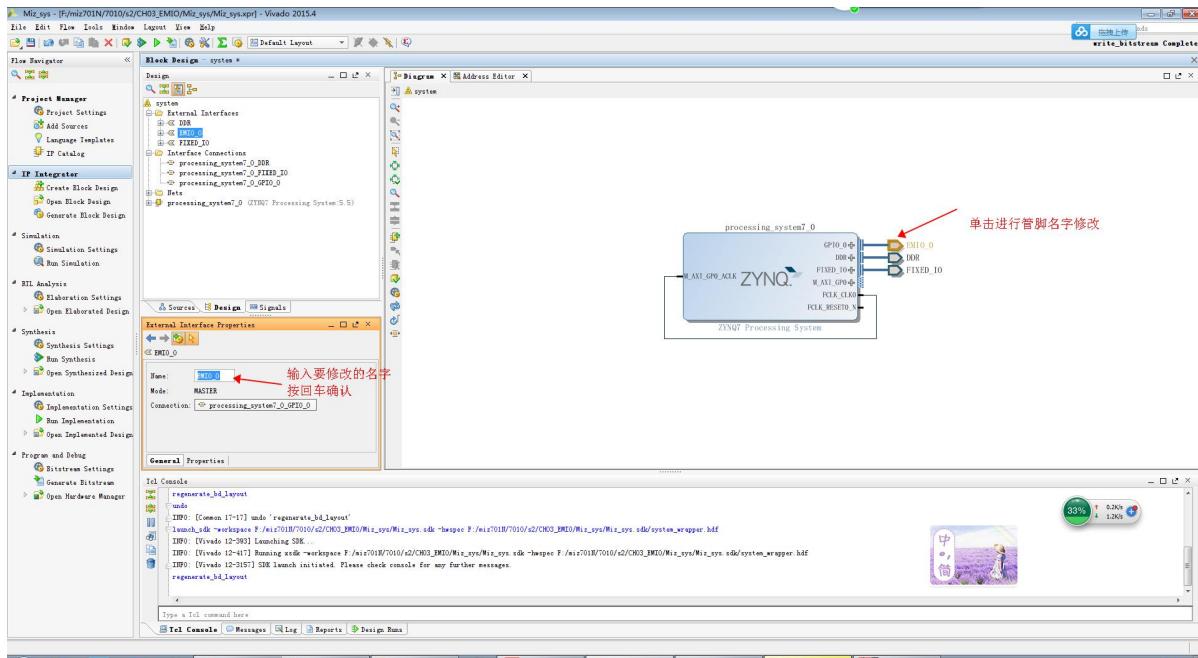
Step4: 在 MIO Configuration 选项卡，再看到 I/O Peripherals 中的 GPIO 一栏，勾选上其中的 EMIO 一栏，并选择 4 位引脚输出（最多可以选择 64 位，但是这个使用只需要 4 位足够了。）



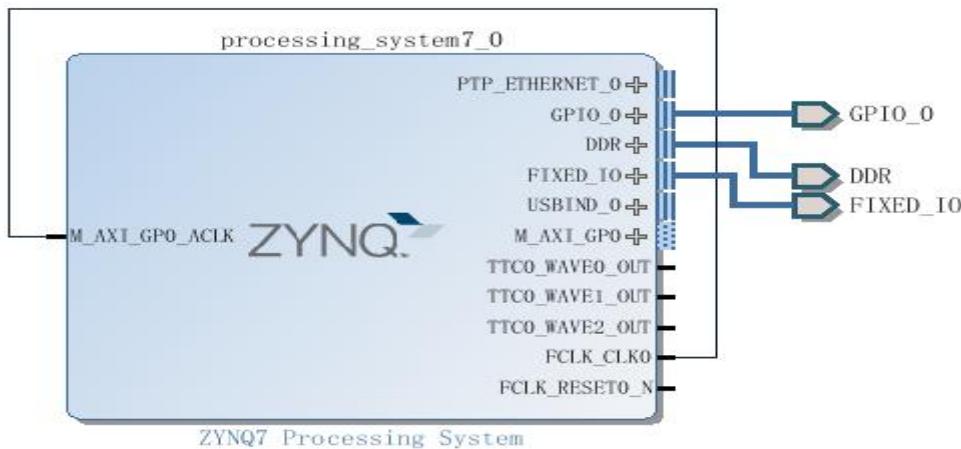
Step5: 单击 OK，仔细观察发现的 zynq 核心多出一组引脚名为 GPIO_0，这个正是我们刚刚设置的一组 EMIO，我们右击该引脚，选择 make external 把 GPIO_0 引脚引出（或者单击该引脚处，按快捷键 Ctrl +t，也可以将引脚引出）。效果如下图所示：



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。

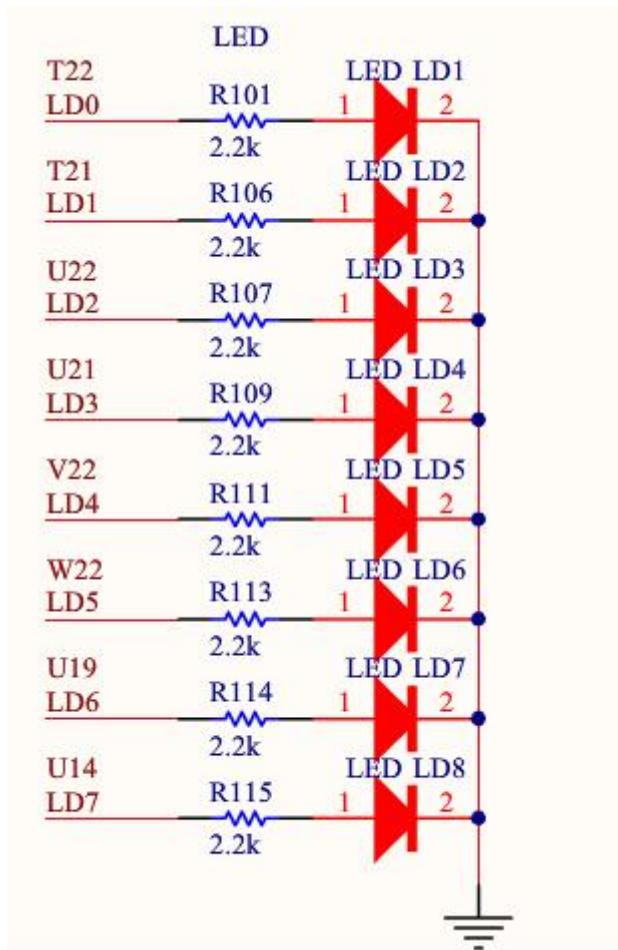


Step8: 右键单击 Block 文件, 文件选择 Generate the Output Products。

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。

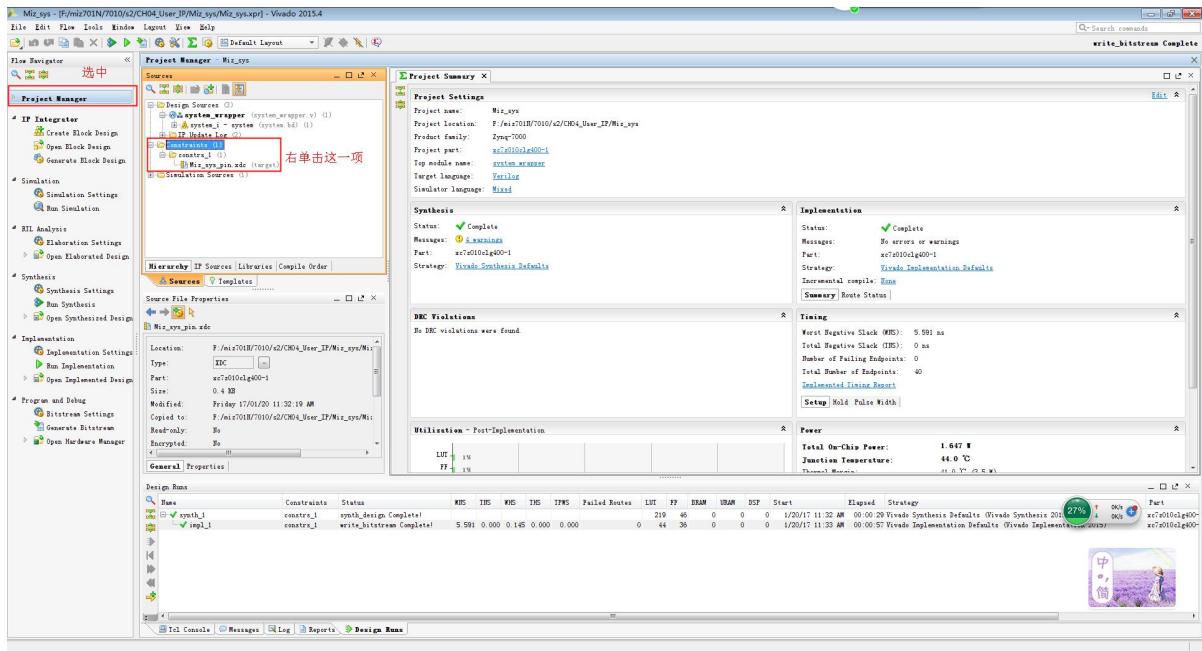
3.4 创建约束文件

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, 此处以 Miz702 开发板为例, Miz702 的 LED 部分原理图如下所示:



此处我们选择 LD1-LD4 分配给 EMIO。

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



Step2:输入文件名，完成创建，将以下约束文件加入约束文件当中。

```

set_property PACKAGE_PIN T22 [get_ports {emio_0_tri_io[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[0]}]

set_property PACKAGE_PIN T21 [get_ports {emio_0_tri_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[1]}]

set_property PACKAGE_PIN U22 [get_ports {emio_0_tri_io[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[2]}]

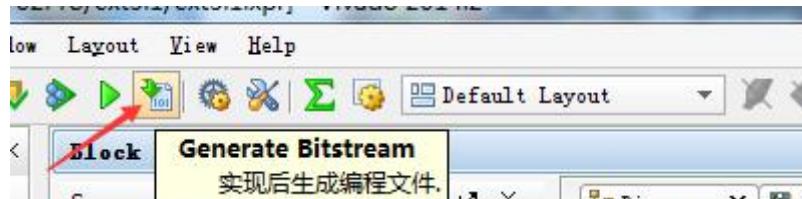
set_property PACKAGE_PIN U21 [get_ports {emio_0_tri_io[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[3]}]

```

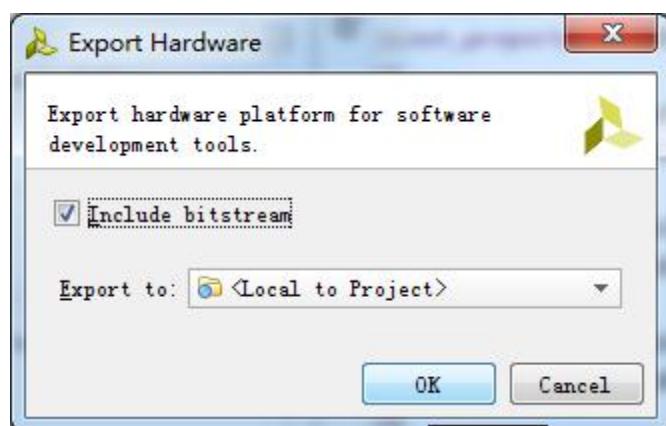
其他型号的用户，可同理查阅原理图或根据型号打开我们提供的源程序的约束文件对系统引脚进行分配。

3.5 产生 bit 文件并导入到 SDK 中

Step1: 生成 bit 文件。

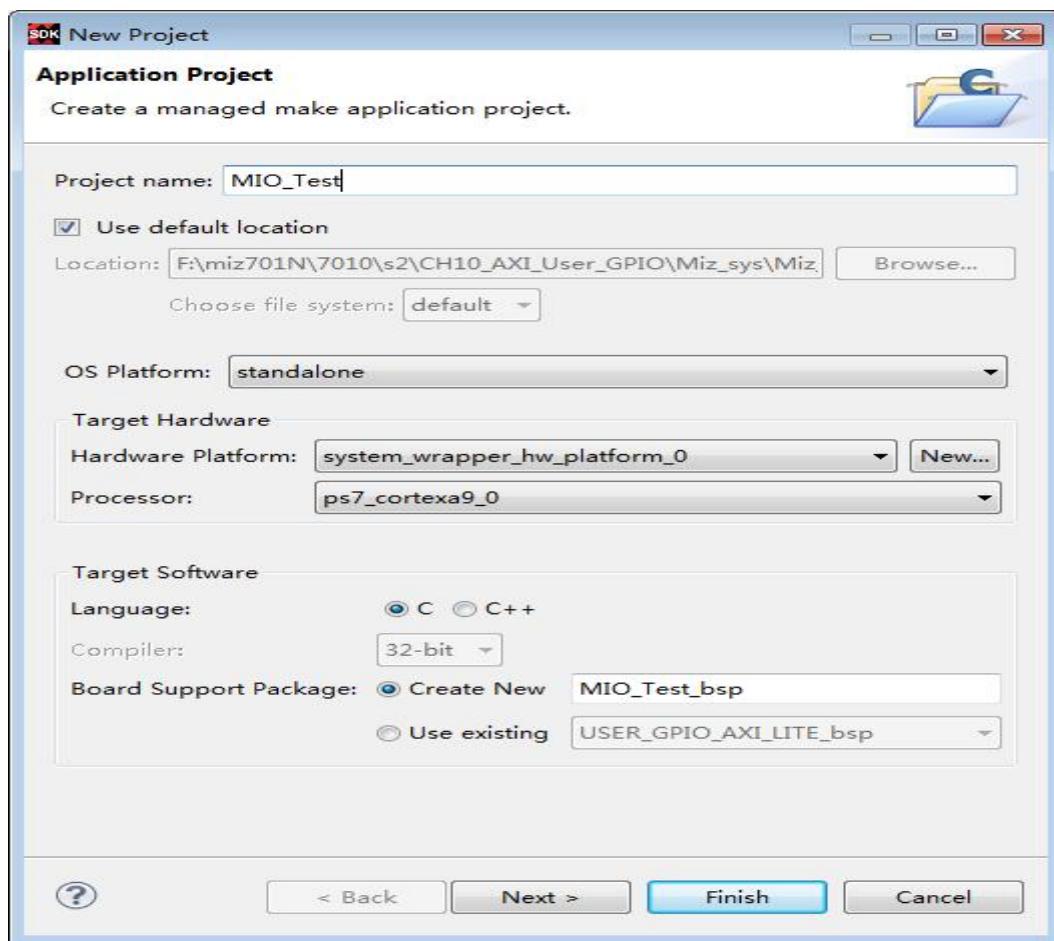


Step2: 导出到硬件。

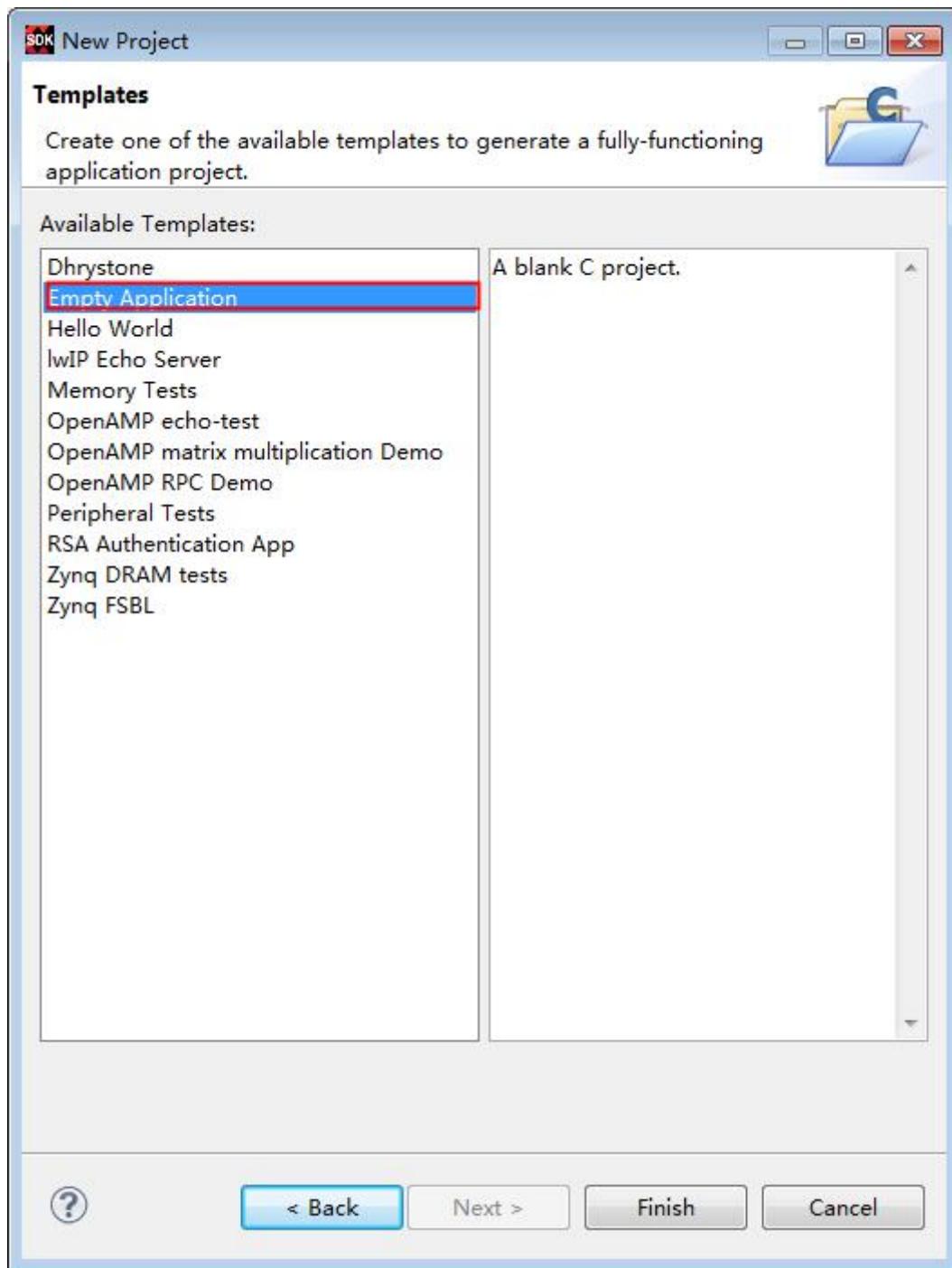


Step3: 打开 SDK，单击 File-New-Application project。

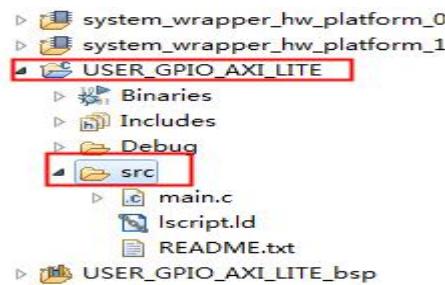
Step4: 输入工程名字，此处命名为 MIO_Test，单击 Next。



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



Step6: 单击工程名字右边的三角形按钮，然后右单击 src，选择 New-source file。



Step7：输入一个文件名，此处命名为 main.c，单击 Finish 按钮完成 C 文件的添加。

Step8：在 main.c 中添加程序如下：

```
#include "xgpiops.h"
#include "sleep.h"

int main()
{
    static XGpioPs psGpioInstancePtr;
    XGpioPs_Config* GpioConfigPtr;
    int xStatus;

    //-- EMIO的初始化
    GpioConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
    if(GpioConfigPtr == NULL)
        return XST_FAILURE;

    xStatus = XGpioPs_CfgInitialize(&psGpioInstancePtr,GpioConfigPtr,
    GpioConfigPtr->BaseAddr);
```

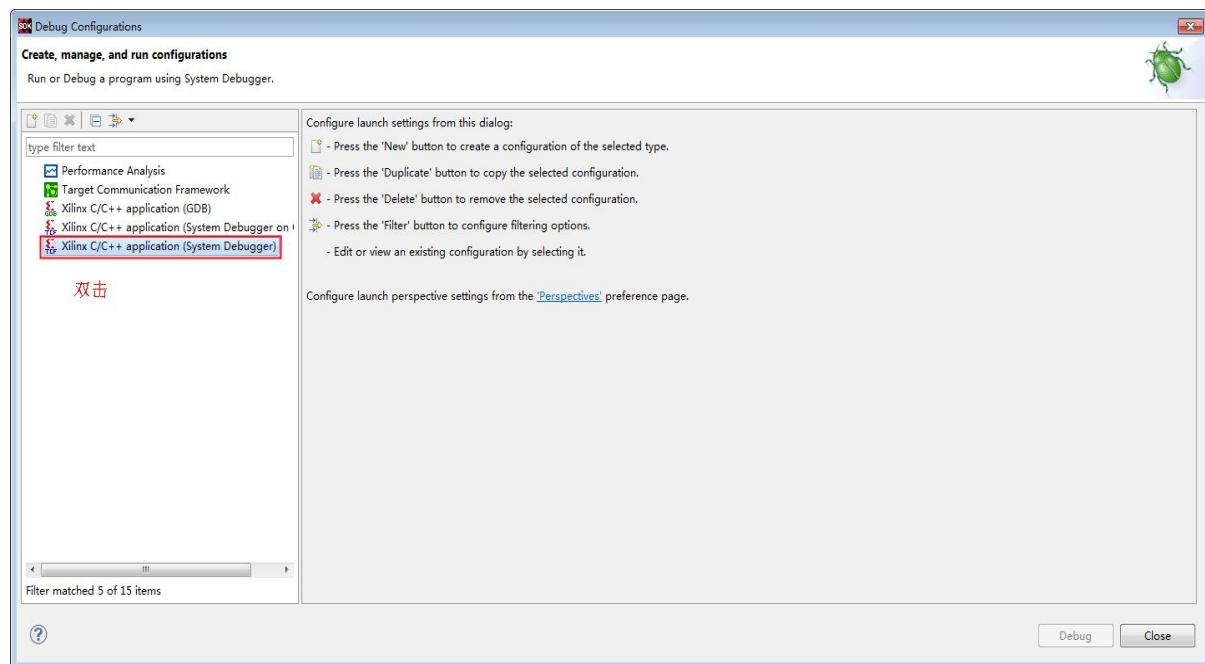
```
if(XST_SUCCESS != xStatus)
    print(" PS GPIO INIT FAILED \n\r");
//--EMIO的输入输出操作
XGpioPs_SetDirectionPin(&psGpioInstancePtr, 54,1);
XGpioPs_SetDirectionPin(&psGpioInstancePtr, 55,1);
XGpioPs_SetDirectionPin(&psGpioInstancePtr, 56,1);
XGpioPs_SetDirectionPin(&psGpioInstancePtr, 57,1);
//使能EMIO输出
XGpioPs_SetOutputEnablePin(&psGpioInstancePtr, 54,1);
XGpioPs_SetOutputEnablePin(&psGpioInstancePtr, 55,1);
XGpioPs_SetOutputEnablePin(&psGpioInstancePtr, 56,1);
XGpioPs_SetOutputEnablePin(&psGpioInstancePtr, 57,1);

while(1)
{
    XGpioPs_WritePin(&psGpioInstancePtr, 54, 1); //EMIO的第0位输出1
    usleep(200000); //延时
    XGpioPs_WritePin(&psGpioInstancePtr, 54, 0); //EMIO的第0位输出0
    usleep(200000); //延时
    XGpioPs_WritePin(&psGpioInstancePtr, 55, 1); //EMIO的第1位输出1
    usleep(200000); //延时
    XGpioPs_WritePin(&psGpioInstancePtr, 55, 0); //EMIO的第1位输出0
    usleep(200000); //延时
    XGpioPs_WritePin(&psGpioInstancePtr, 56, 1); //EMIO的第2位输出1
    usleep(200000); //延时
    XGpioPs_WritePin(&psGpioInstancePtr, 56, 0); //EMIO的第2位输出0
    usleep(200000); //延时
    XGpioPs_WritePin(&psGpioInstancePtr, 57, 1); //EMIO的第3位输出1
    usleep(200000); //延时
    XGpioPs_WritePin(&psGpioInstancePtr, 57, 0); //EMIO的第3位输出0
    usleep(200000); //延时

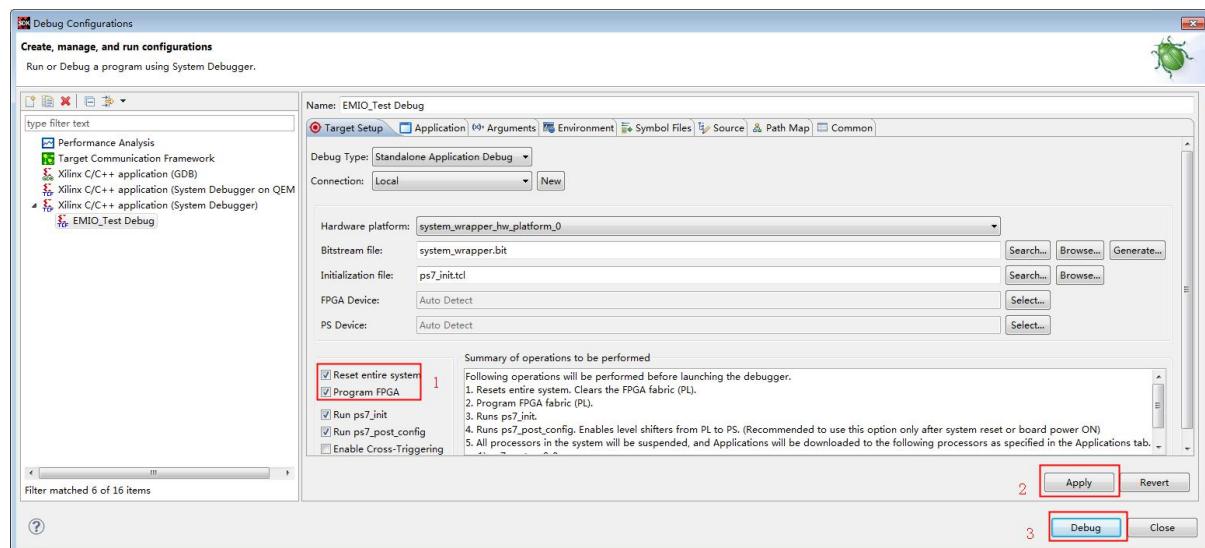
}
return 0;
}
```

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

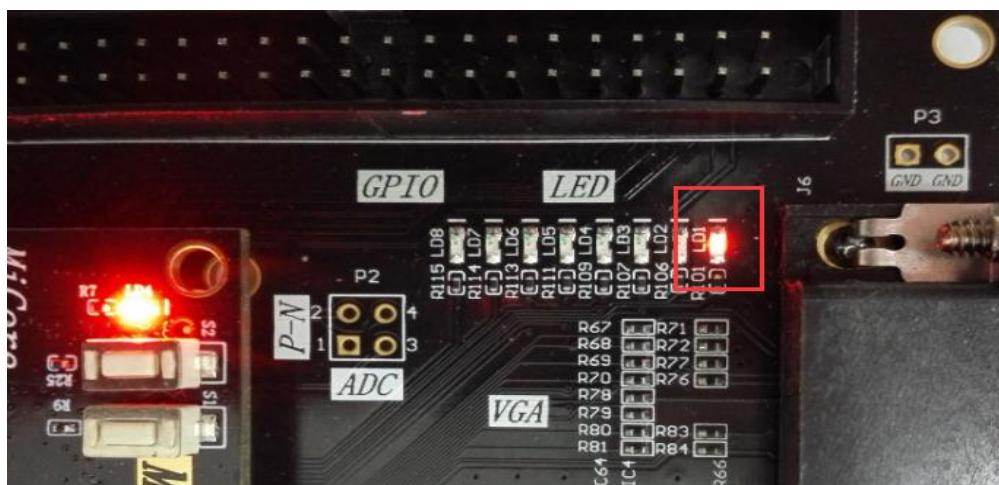
Step10: 选中 system Debugger,双击创建一个系统调试。



Step11: 设置系统调试。

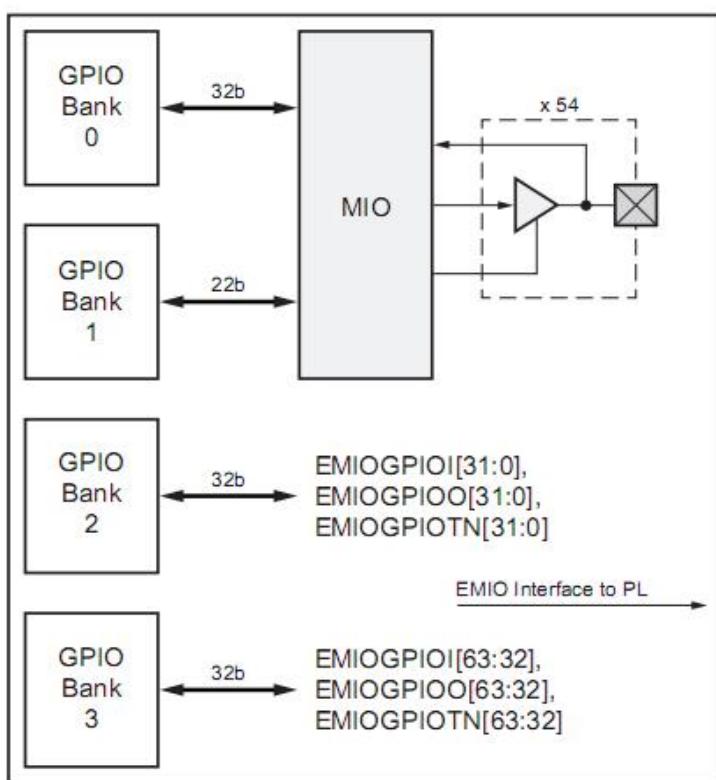


Step12: 单击窗口上的运行按钮，运行程序，可看到 LED 的流水操作。



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。

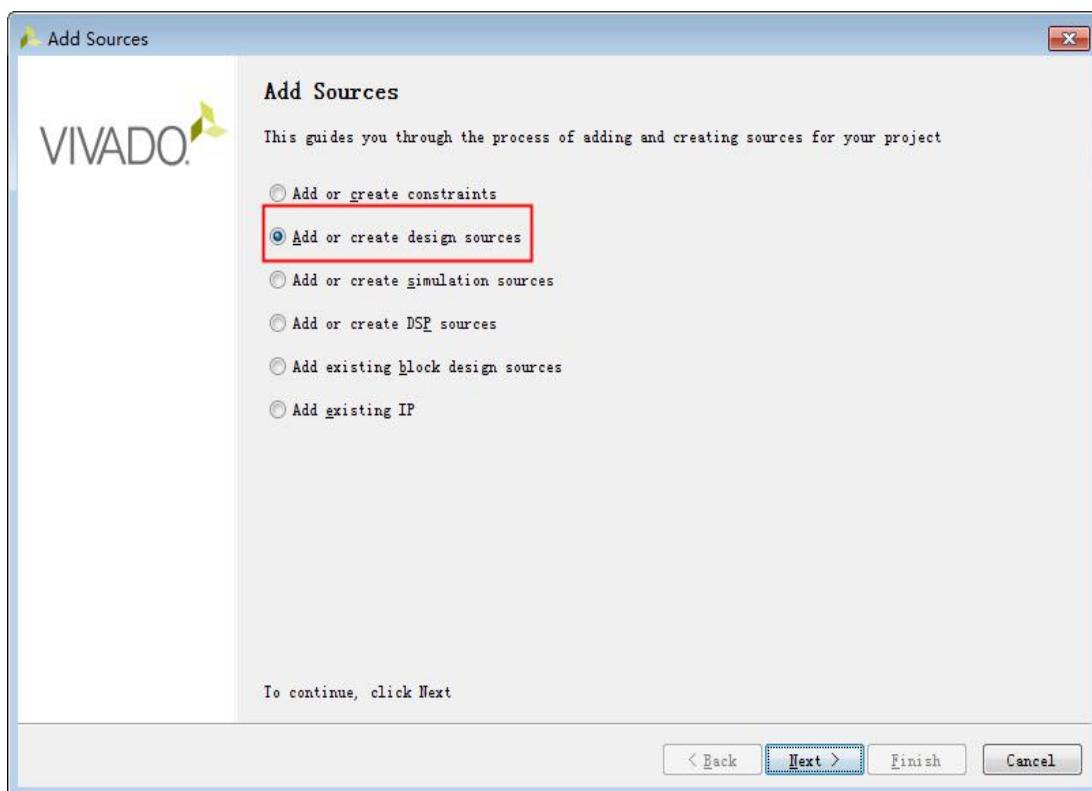
S02_CH04_User_IP 实验

4.1 创建 IP

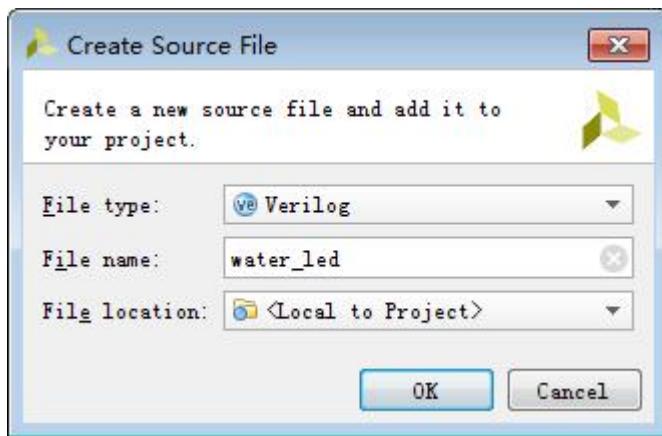
在之前的教程中，我们通过 MIO 与 EMIO 来控制 LED，所使用的也是官方的 IP，实际当中，官方提供的 IP 不可能涵盖到方方面面，用户需要自己编写硬件描述语言，然后将其封装成 IP 来使用，本节就将详细的讲解如何在 VIVADO 中创建用户自定义的 IP。

Step1：打开 VIVADO 软件，新建一个工程。

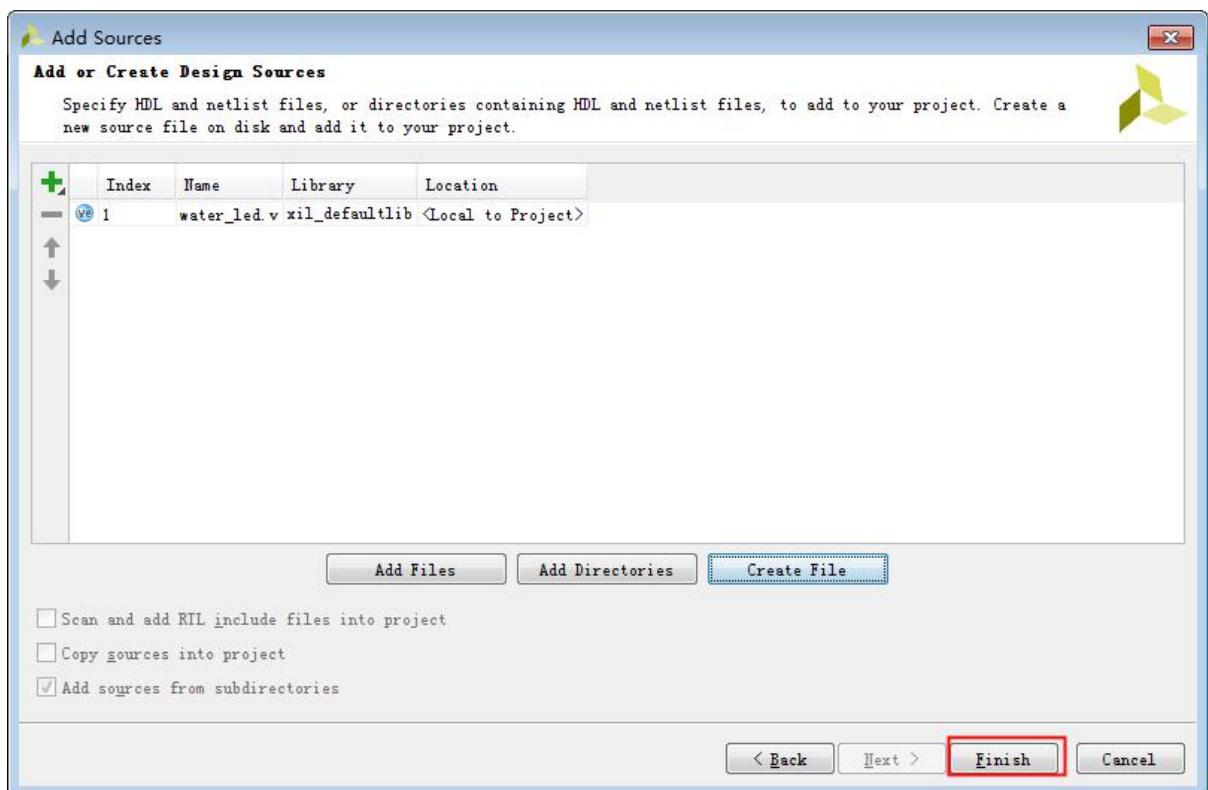
Step2：单击 Add Source，选择 Add or Create design Sources,然后单击 Next。



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。

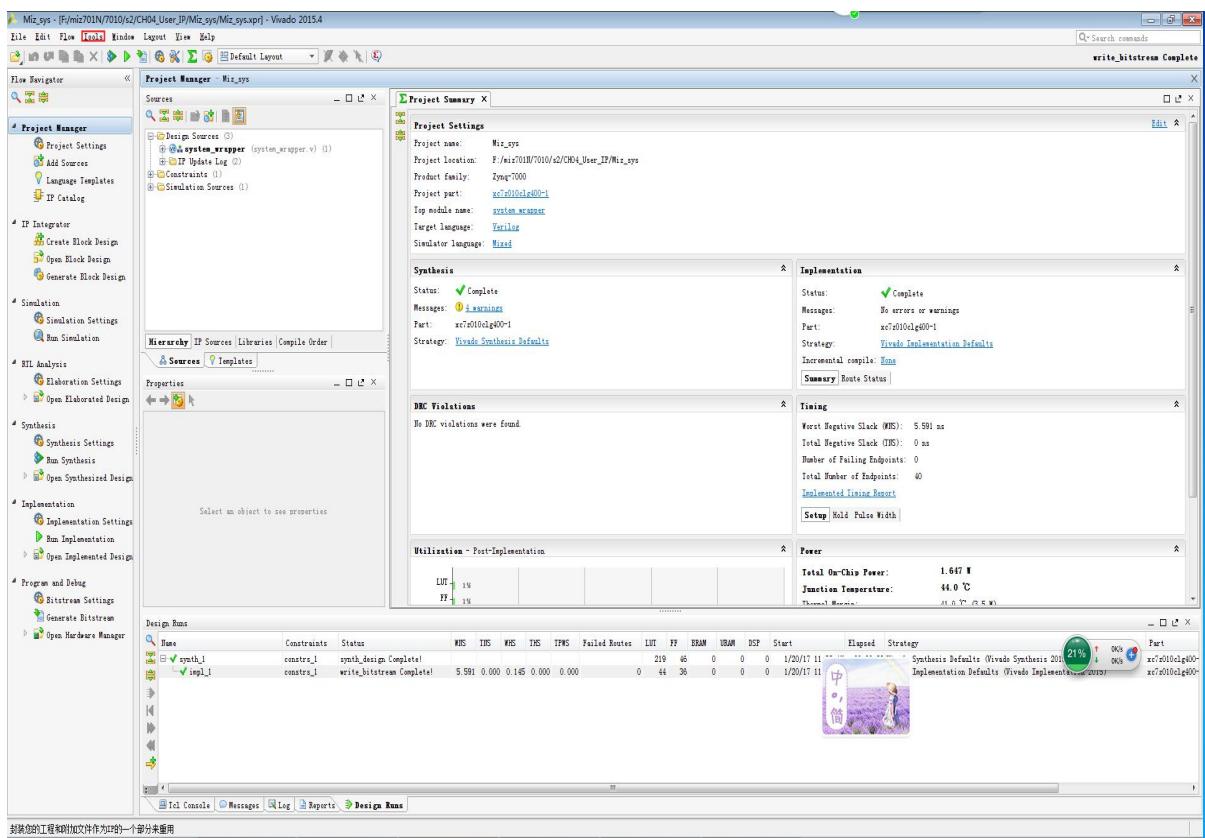


Step5：将以下代码复制入文本编辑区内。

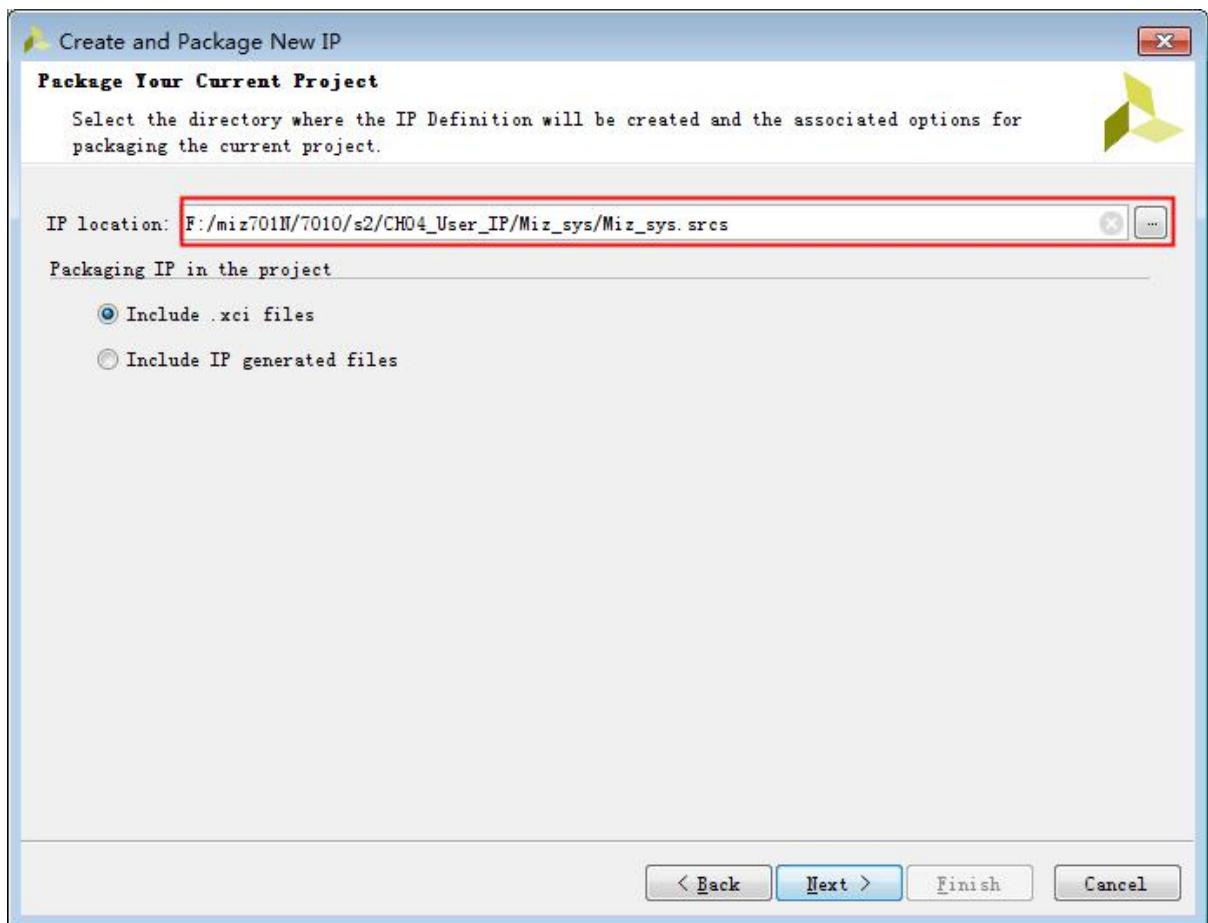
```
module LED_ML(
    input CLK_i,//100MHZ
    input RSTn_i,
    output reg [3:0]LED_o
);
    reg [31:0]C0;
    always @(posedge CLK_i)
```

```
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。

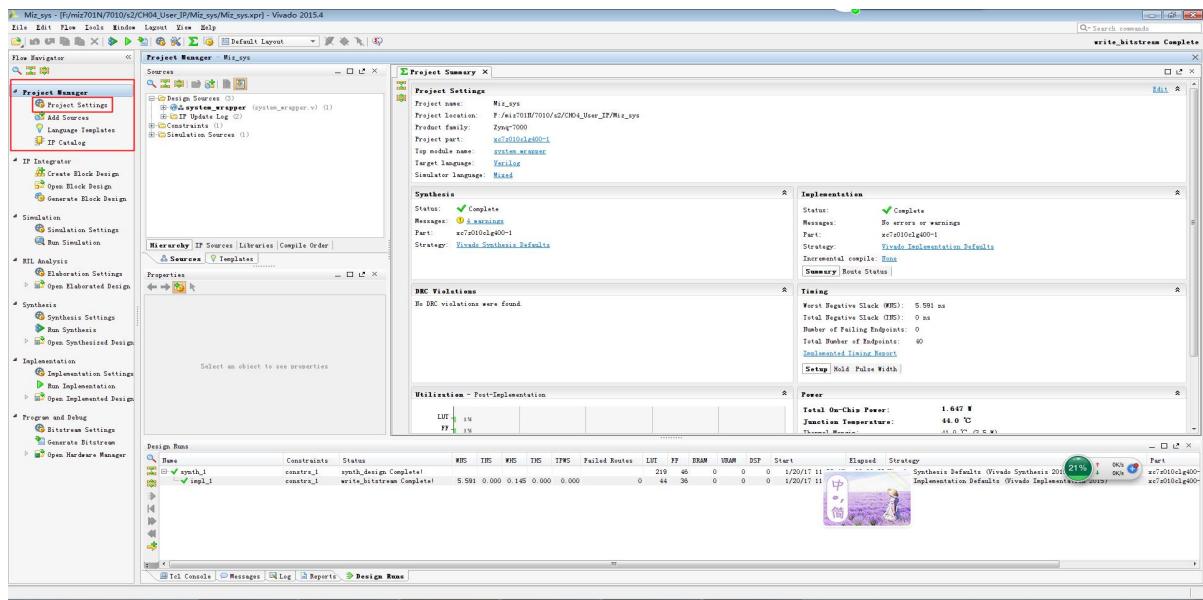


Step8：单击 Finish 完成封装。

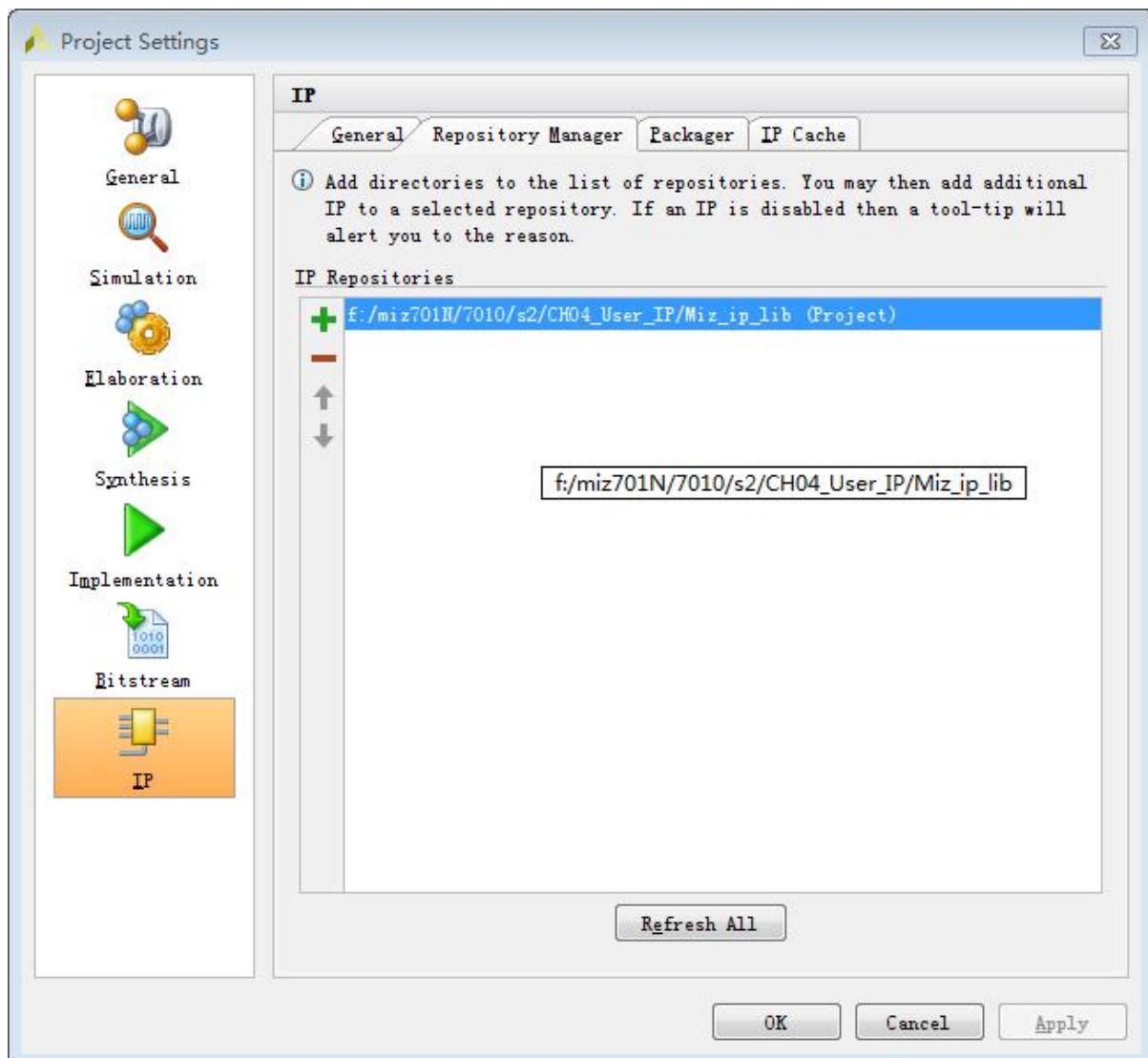
4.2 调用自定义 IP

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, 。

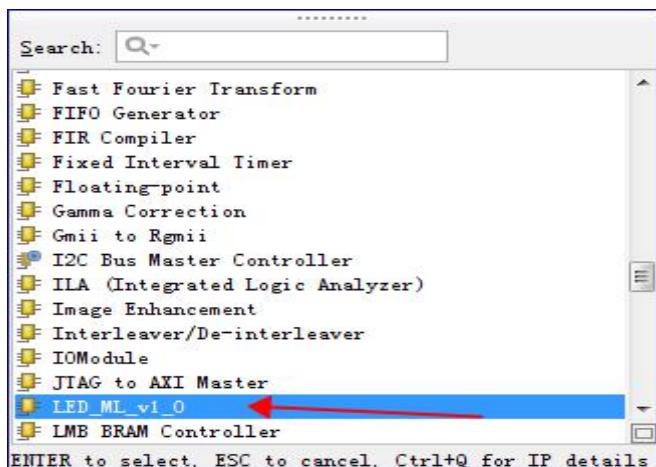


Step4: 单击+号图标, 将上一节封装的 IP 的路劲存放进去, 单击 OK。

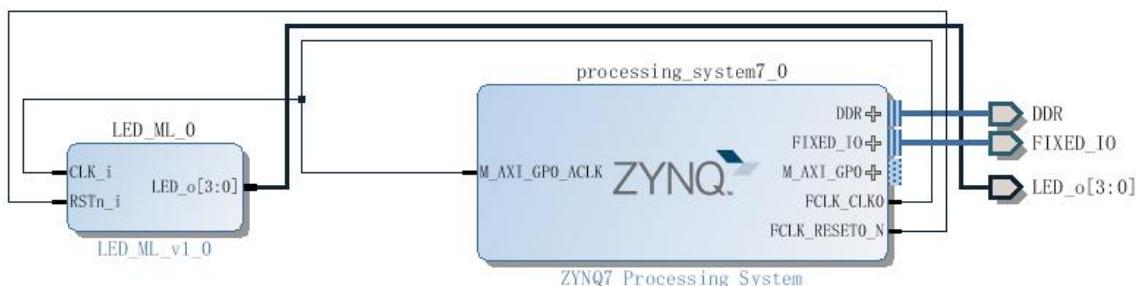
Step5: 新建一个 BD 文件, 输入文件名, 完成创建。

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: 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。

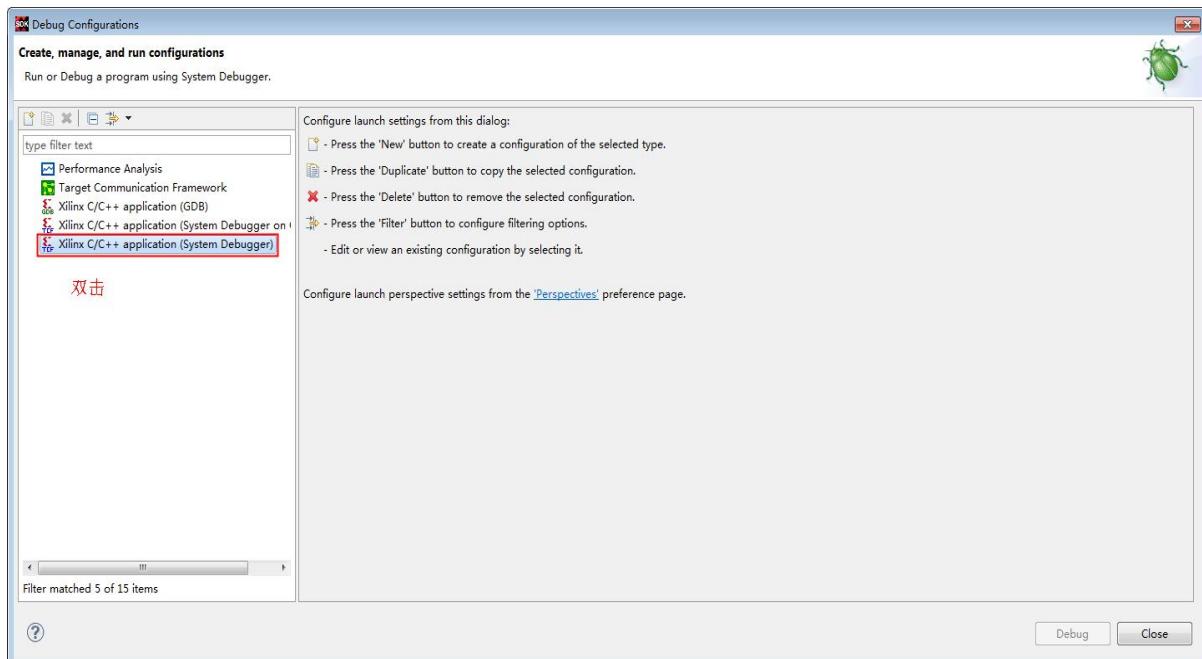
4.3 导入到 S D K

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。

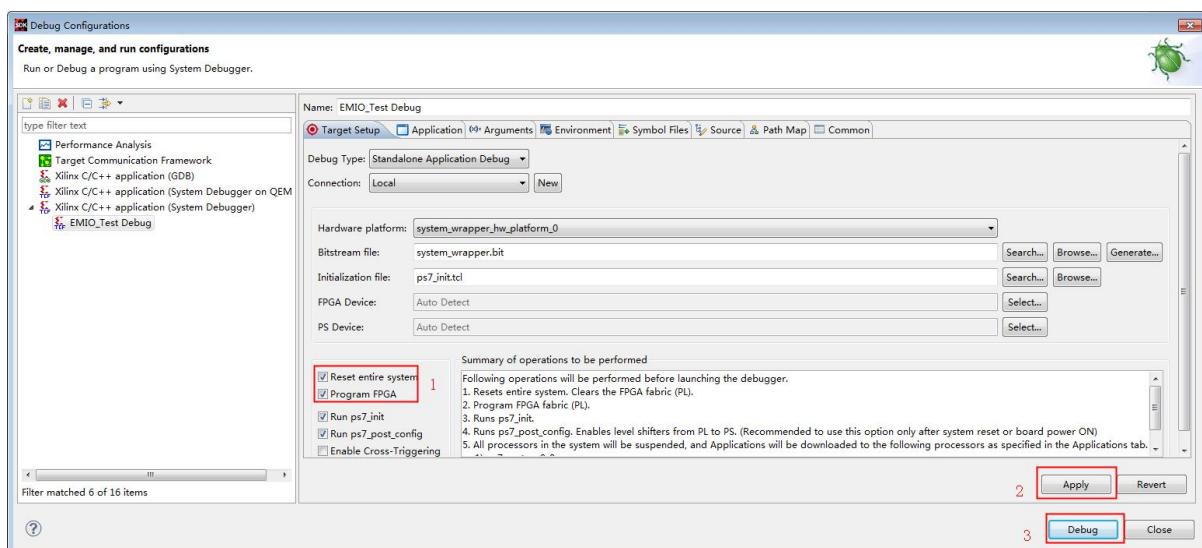
Step1: 创建一个 Hello World 工程。

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。

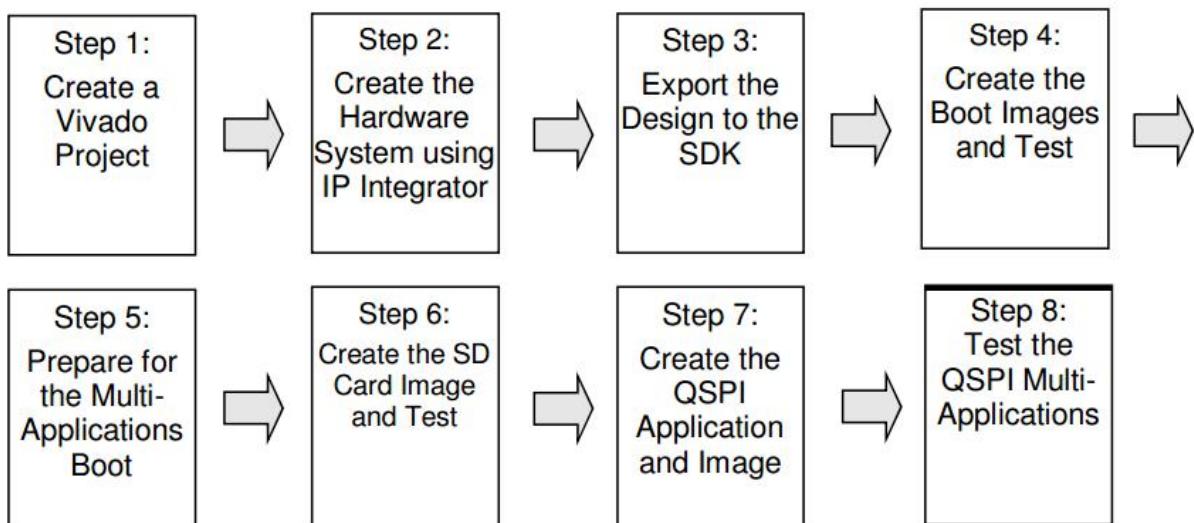
S02_CH05_UBOOT 实验

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。

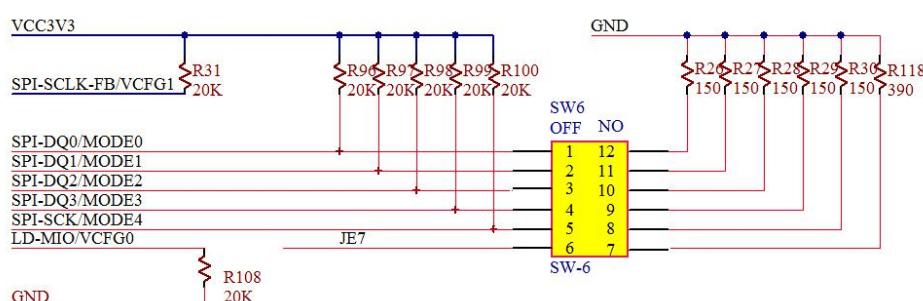
阶段 1：BOOT.bin 加载到 OCM 上就开始执行了，之前说过 BOOT.bin 其实就是由 FSBL.elf+该工程.bit+该工程.elf 构成。而阶段 1 要做的就是：首先配置 PS 部分，PS 完成初始化后，会去配置 PL 部分，最后还可以去加载阶段 2 的代码。

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。

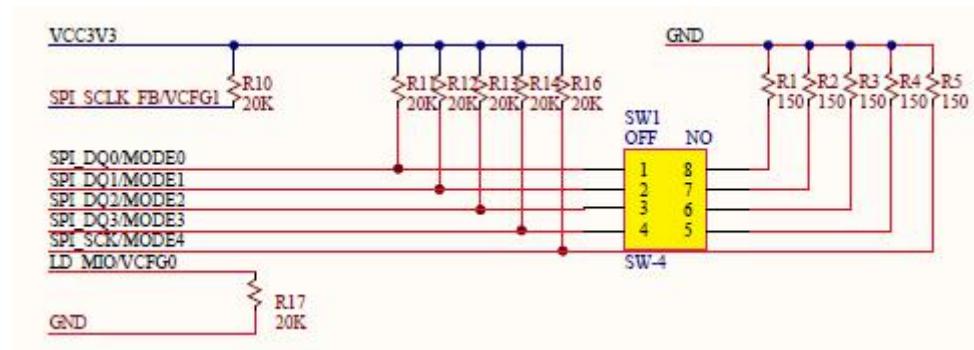
5.5 zynq 启动模式位的选择

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（MiZ702 的拨码开关在按键旁）。

MiZ702 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。



MiZ702 通过拨码开关设置 MIO 的电平状态



Miz701N 和 Miz702N 通过拨码开关设置 MIO 的电平状态

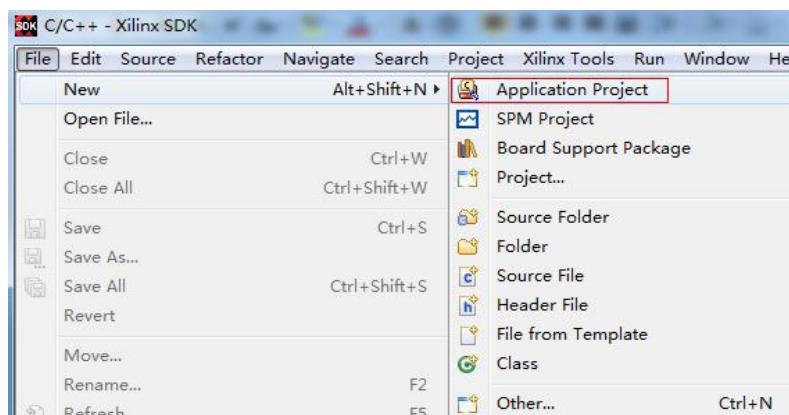
我们的开发板默认拨码的顺序，就是默认的 SD 卡启动，具体模式位应该如何选择如下表所示：

Xilinx TRM →	MIO[6] Boot_Mode[4]	MIO[5] Boot_Mode[3]	MIO[4] Boot_Mode[2]	MIO[3] Boot_Mode[1]	MIO[2] Boot_Mode[0]
JTAG Mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
PLL Mode					
PLL Used	0				
PLL Bypassed	1				
Bank Voltages					
MIO Bank 500				3.3V	
MIO Bank 501				1.8V	

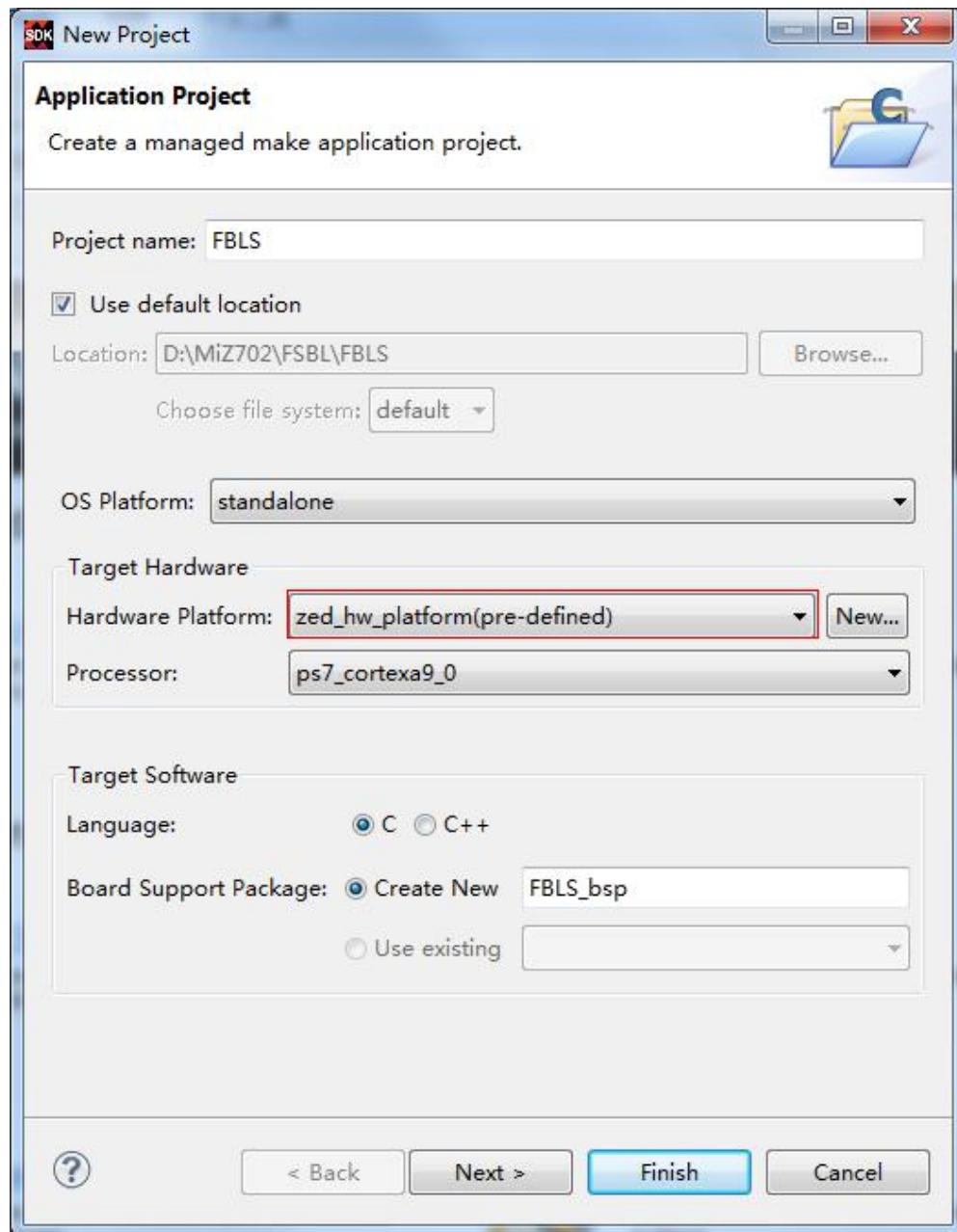
5.6 BOOT.bin 制作过程详解

Step1：打开上一章的工程，并打开 SDK 软件。

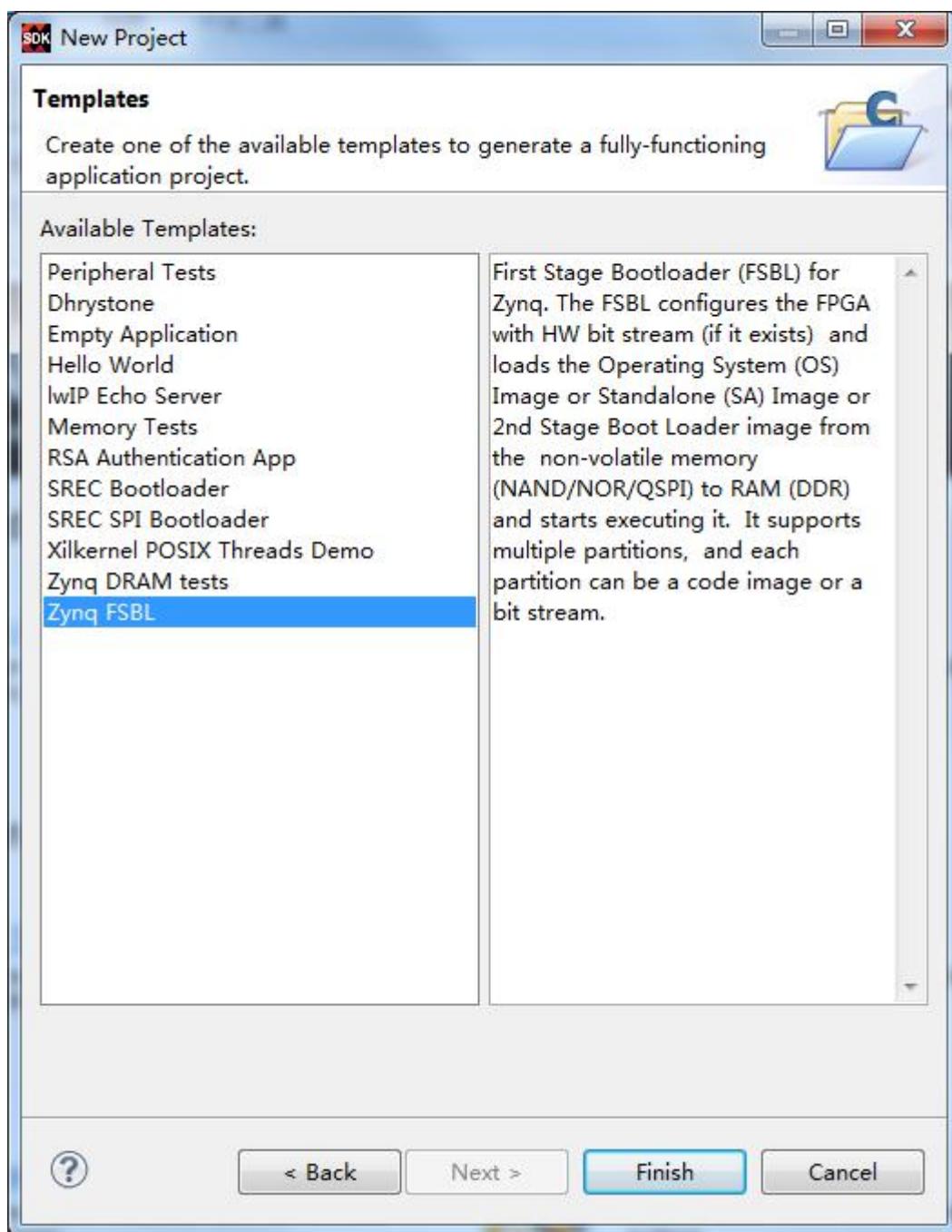
Step2：新建一个应用工程



Step3：填写工程名，点击 Next



Step4: 现在工程类型为 Zynq FSBL



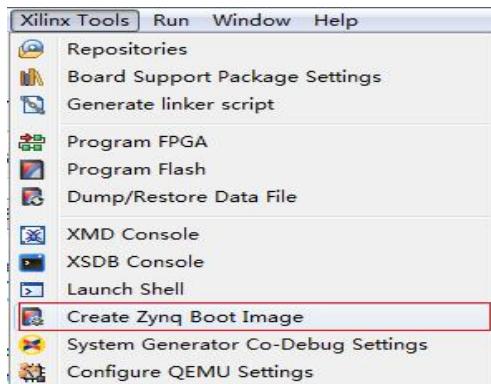
Step5：按下快捷键 Ctrl + B 编译工程就能得到我们梦寐以求的 FSBL.elf。这个文件可以到我们刚刚设置的工作空间的 Debug 目录下找打，我这边具体目录是：
D:\MiZ702\FSBL\FBLS\Debug。

名称	修改日期	类型	大小
src	2016/3/24 22:32	文件夹	
FBLS.elf	2016/3/24 22:32	ELF 文件	409 KB
FBLS.elf.size	2016/3/24 22:32	SIZE 文件	1 KB
makefile	2016/3/24 22:32	文件	2 KB
objects.mk	2016/3/24 22:32	Makefile	1 KB
sources.mk	2016/3/24 22:32	Makefile	1 KB

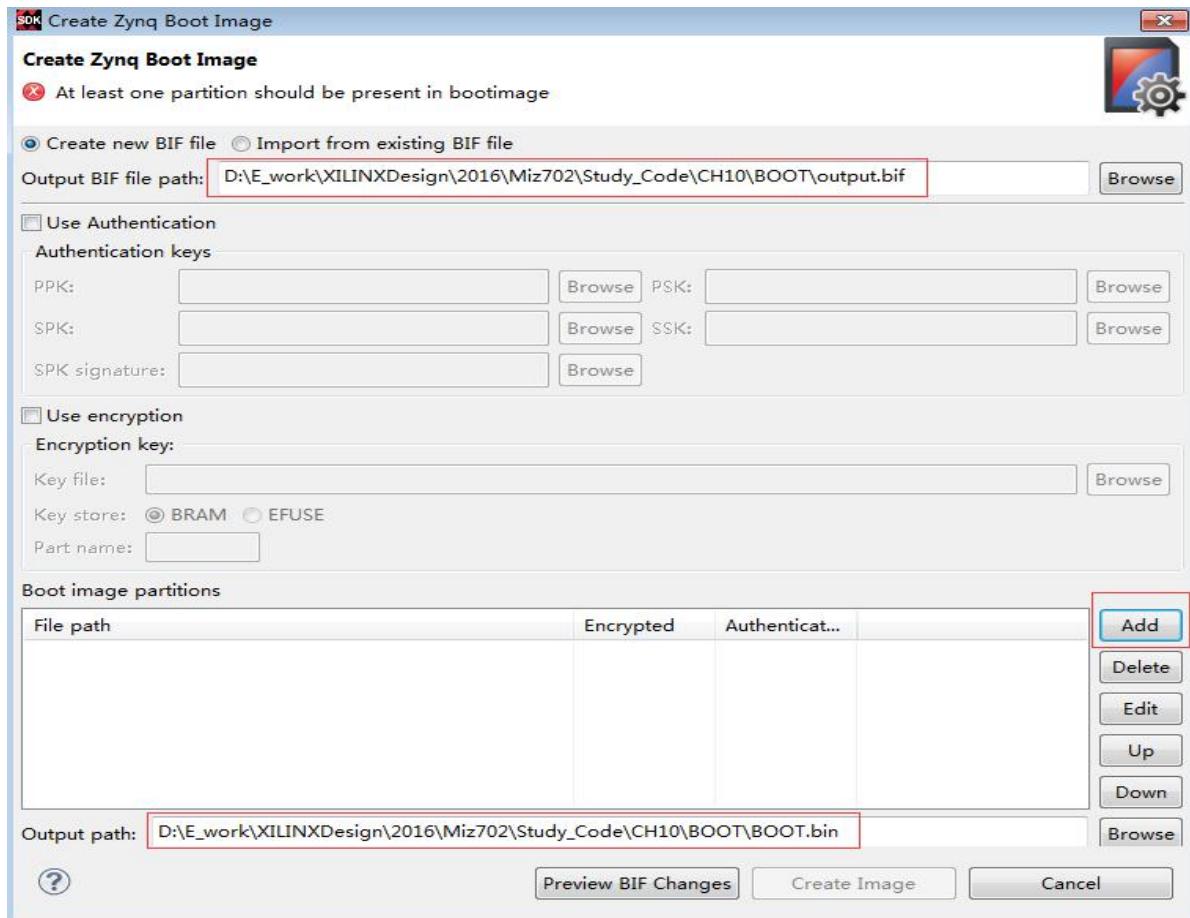
Step6: 我们将 FSBL.elf 连同《CH04_User_IP》工程中的 elf 和 bit 文件拷贝到一个文件夹下备用。

EMIOTest.elf	2016/3/26 21:54	ELF 文件	184 KB
FBLS.elf	2016/3/29 16:06	ELF 文件	417 KB
system_wrapper.bit	2016/3/29 16:04	BIT 文件	3,951 KB

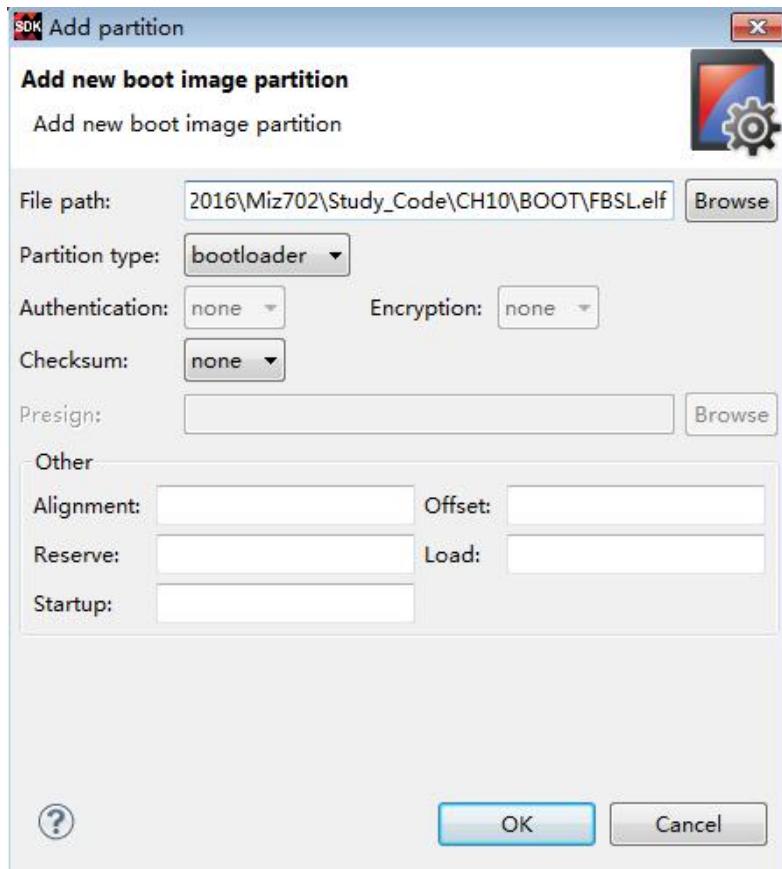
Step7: 单击 SDK 的工具栏处的 Xilinx Tool->Create Zynq Boot Image



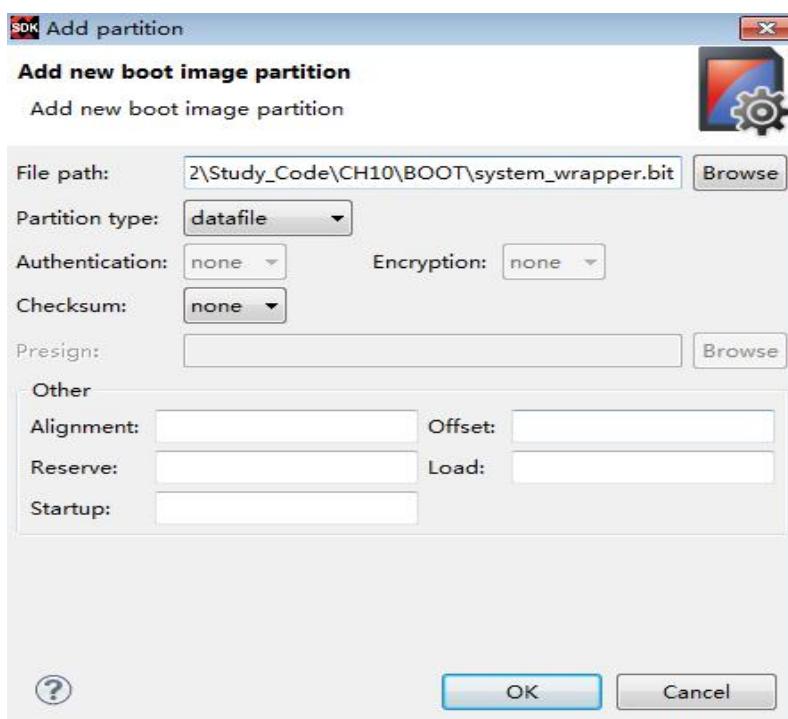
Step8: 依次添加 FBLS.elf, design_1_wrapper.bit, 以及 emio.elf, 请务必按顺序添加。



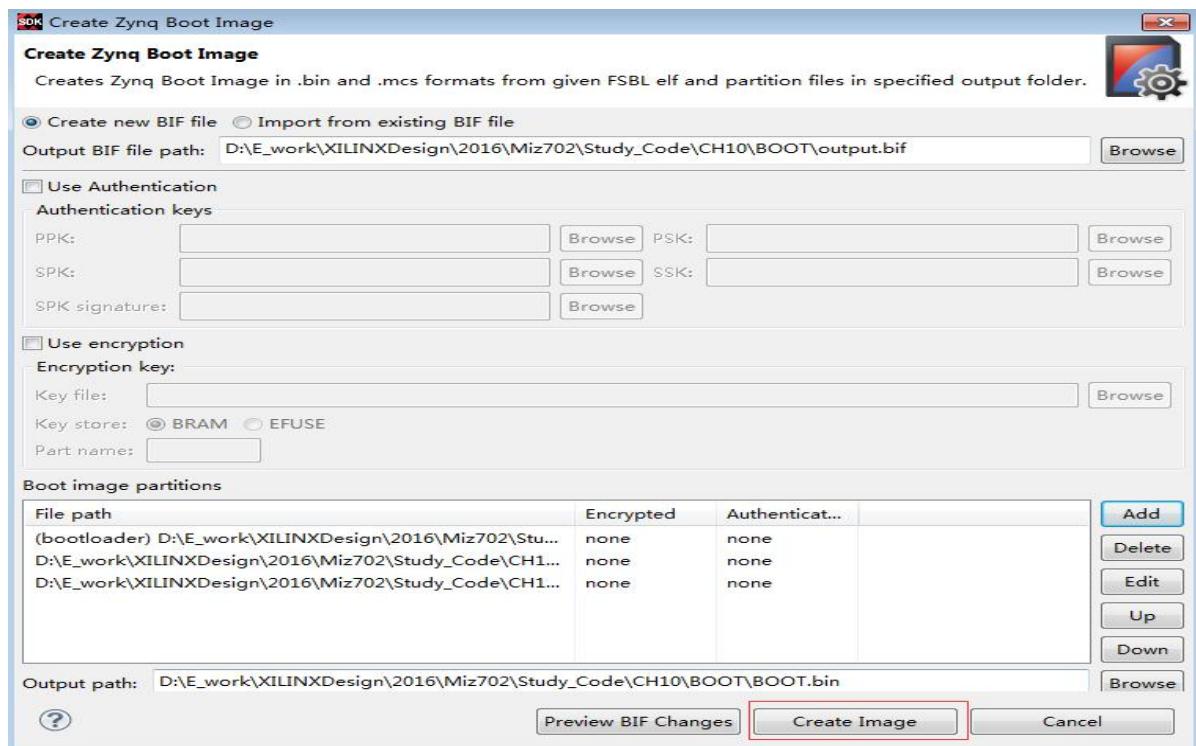
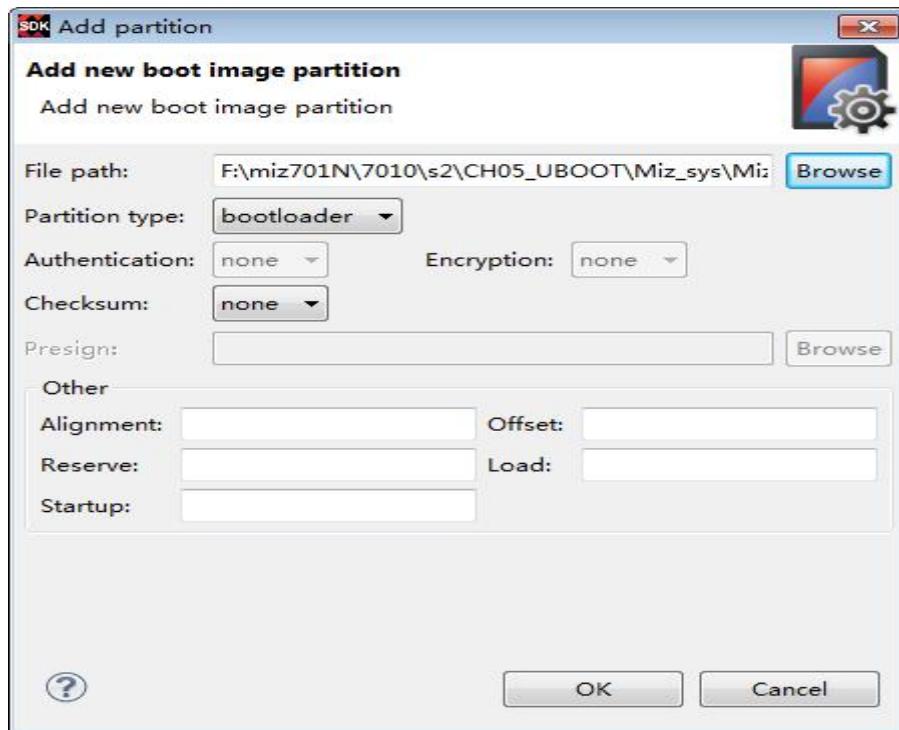
Step9: 点击 add, 添加 FSBL.elf



Step10: 点击 add, 添加 system_wrapper.bit



Step11: 点击 add, 添加 Helloworld.elf 文件



Step12：三个文件添加完毕之后，点击 Create Image 生成 BOOT.bin

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

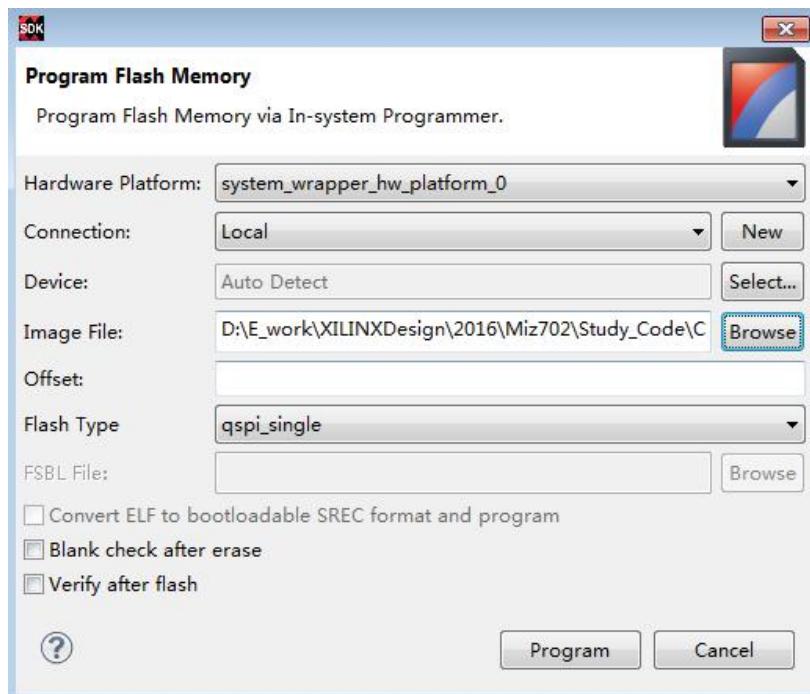
Step1: 设置配置模式

正确设置模块开关跳线，也就是把 MI05 切换到 OFF(上拉为 H) 其他全部切换到 ON(短接到 GND)

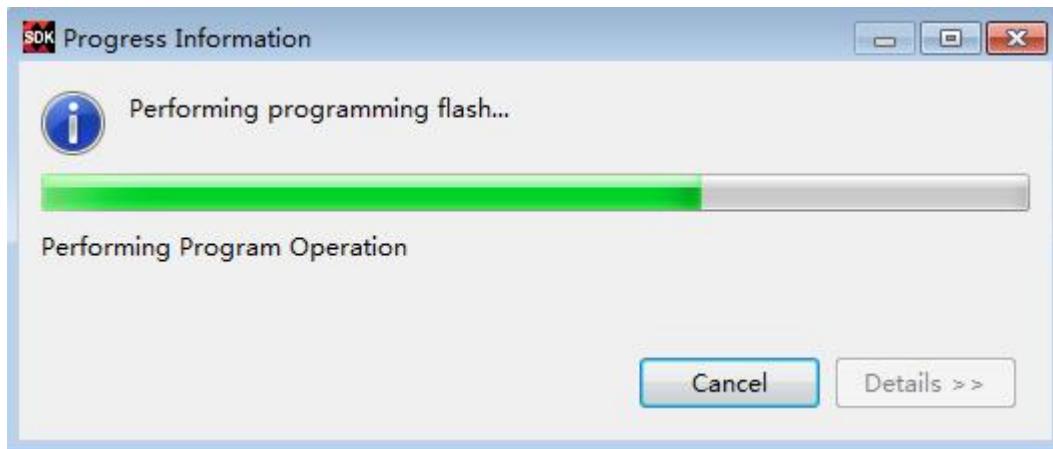
Xilinx TRM→	MIO[6]	MIO[5]	MIO[4]	MIO[3]	MIO[2]
	Boot_Mode[4]	Boot_Mode[0]	Boot_Mode[2]	Boot_Mode[1]	Boot_Mode[3]
JTAG Mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
PLL Mode					
PLL Used	0				
PLL Bypassed	1				
Bank Voltages					
MIO Bank 500				3.3V	
MIO Bank 501				1.8V	

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。

S02_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

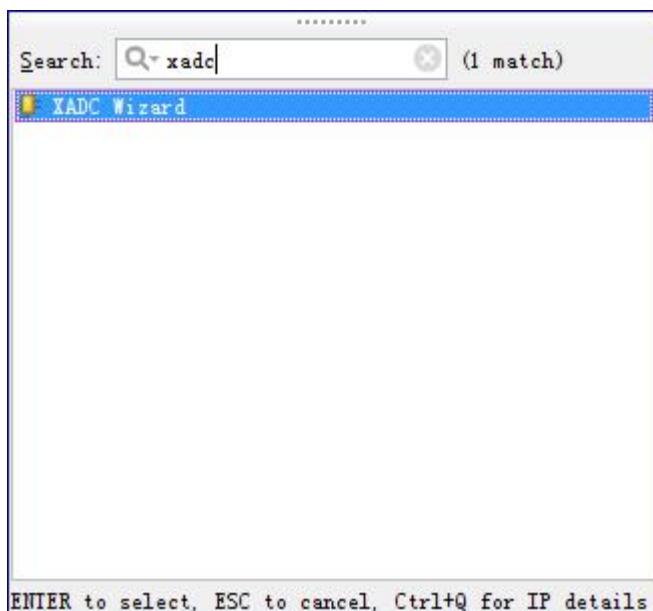
6.2 新建一个 VIVADO 工程

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。

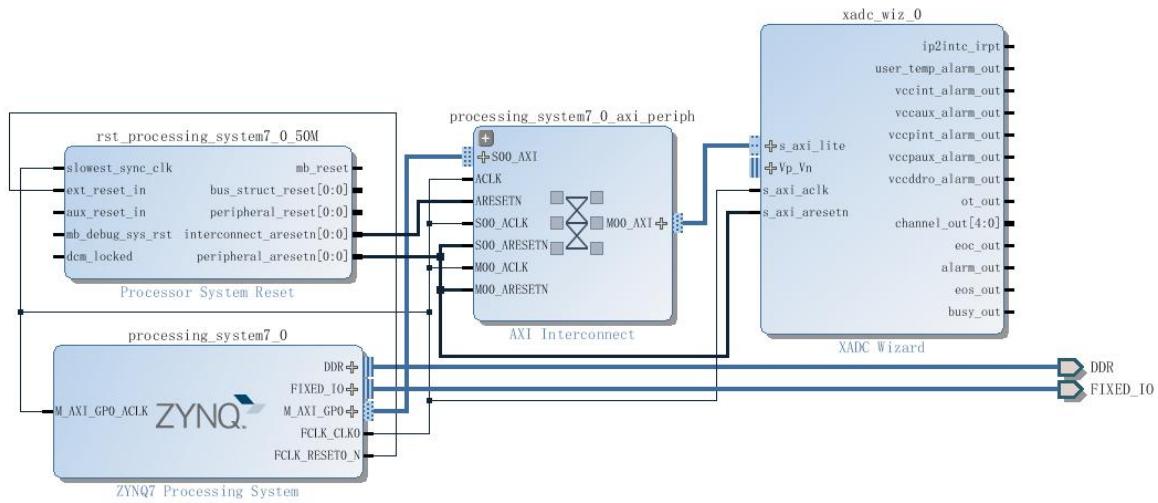
Step3:添加 ZYNQ7 Processing System，根据自己的硬件类型配置好输入时钟频率与内存型号。

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: 新建一个空 SDK 工程，并添加一个 main.c 的文件。

Step3: 在 main.c 文件中添加以下程序，按 Ctrl+S 保存后自动开始编译。

```
/*
 * main.c
 *
 * Created on: 2016 年 6 月 25 日
 * Author: Administrator
 */
#include <stdio.h>
//#include "platform.h"
#include "xadcps.h"
#include "xil_types.h"
#define XPAR_AXI_XADC_0_DEVICE_ID 0

//void print(char *str);
```

```
static XAdcPs XADCMonInst;

int main()
{
    XAdcPs_Config *ConfigPtr;
    XAdcPs *XADCInstPtr = &XADCMonInst;

    //status of initialisation
    int Status_ADC;

    //temperature readings
    u32 TempRawData;
    float TempData;

    //Vcc Int readings
    u32 VccIntRawData;
    float VccIntData;

    //Vcc Aux readings
    u32 VccAuxRawData;
    float VccAuxData;

    //Vbram readings
    u32 VBramRawData;
    float VBramData;

    //VccPInt readings
    u32 VccPIntRawData;
    float VccPIntData;

    //VccPAux readings
    u32 VccPAuxRawData;
    float VccPAuxData;
```

```
//Vddr readings
u32 VDDRRawData;
float VDDRData;

// init_platform();

ConfigPtr = XAdcPs_LookupConfig(XPAR_AXI_XADC_0_DEVICE_ID);
if (ConfigPtr == NULL) {
    return XST_FAILURE;
}

Status_ADC
XAdcPs_CfgInitialize(XADCInstPtr, ConfigPtr, ConfigPtr->BaseAddress);
if (XST_SUCCESS != Status_ADC) {
    print("ADC INIT FAILED\n\r");
    return XST_FAILURE;
}

//self test
Status_ADC = XAdcPs_SelfTest(XADCInstPtr);
if (Status_ADC != XST_SUCCESS) {
    return XST_FAILURE;
}

//stop sequencer
XAdcPs_SetSequencerMode(XADCInstPtr, XADCPS_SEQ_MODE_SINGCHAN);

//disable alarms
XAdcPs_SetAlarmEnables(XADCInstPtr, 0x0);

//configure sequencer to just sample internal on chip parameters
XAdcPs_SetSeqInputMode(XADCInstPtr, XADCPS_SEQ_MODE_SAFE);

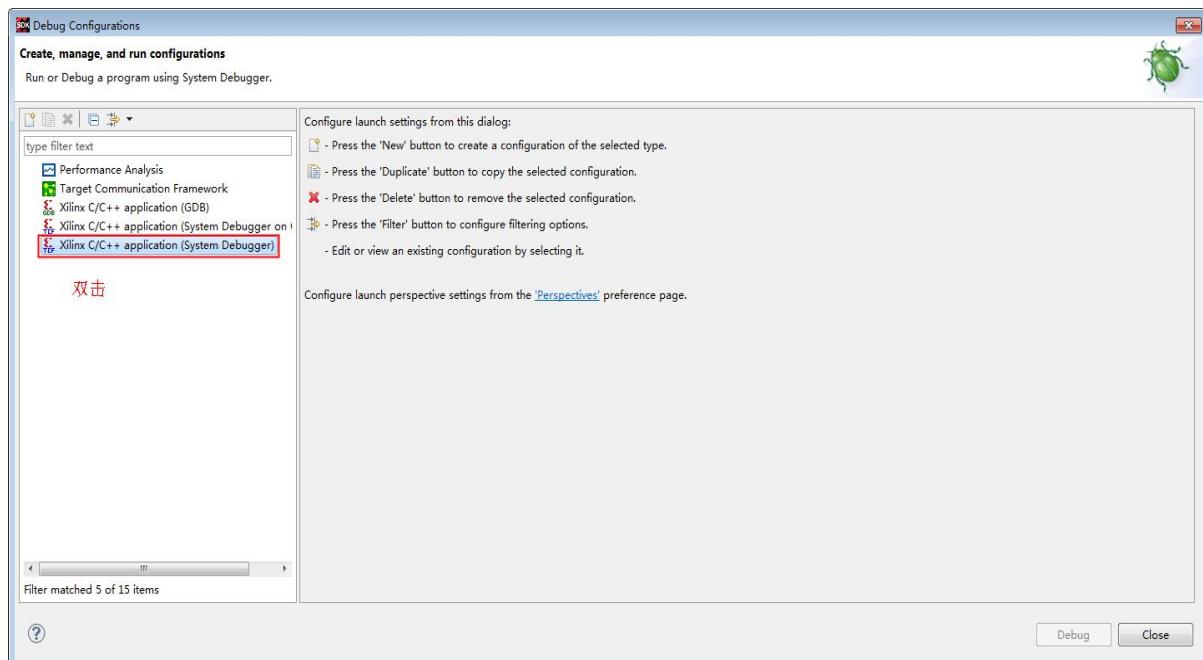
//configure the channel enables we want to monitor
```

```
XAdcPs_SetSeqChEnables(XADCInstPtr, XADCPS_CH_TEMP|XADCPS_CH_VCCINT|XADC  
PS_CH_VCCAUX|XADCPS_CH_VBRAM|XADCPS_CH_VCCPINT|  
XADCPS_CH_VCCPAUX|XADCPS_CH_VCCPDRO) ;  
  
while(1)  
{  
    TempRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_TEMP) ;  
    TempData = XAdcPs_RawToTemperature(TempRawData) ;  
    printf("Raw Temp %lu Real Temp %f \n\r", TempRawData, TempData) ;  
  
    VccIntRawData= XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCINT) ;  
    VccIntData = XAdcPs_RawToVoltage(VccIntRawData) ;  
    printf("Raw      VccInt      %lu      Real      VccInt      %f      \n\r",  
VccIntRawData, VccIntData) ;  
  
    VccAuxRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCAUX) ;  
    VccAuxData = XAdcPs_RawToVoltage(VccAuxRawData) ;  
    printf("Raw      VccAux      %lu      Real      VccAux      %f      \n\r",  
VccAuxRawData, VccAuxData) ;  
  
    VBramRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VBRAM) ;  
    VBramData = XAdcPs_RawToVoltage(VBramRawData) ;  
    printf("Raw  VccBram  %lu  Real  VccBram  %f  \n\r", VBramRawData,  
VBramData) ;  
  
    VccPIntRawData          = XAdcPs_GetAdcData(XADCInstPtr,  
XADCPS_CH_VCCPINT) ;  
    VccPIntData = XAdcPs_RawToVoltage(VccPIntRawData) ;  
    printf("Raw  VccPInt  %lu  Real  VccPInt  %f  \n\r", VccPIntRawData,  
VccPIntData) ;  
  
    VccPAuxRawData          = XAdcPs_GetAdcData(XADCInstPtr,  
XADCPS_CH_VCCPAUX) ;  
    VccPAuxData = XAdcPs_RawToVoltage(VccPAuxRawData) ;  
    printf("Raw  VccPAux  %lu  Real  VccPAux  %f  \n\r", VccPAuxRawData,
```

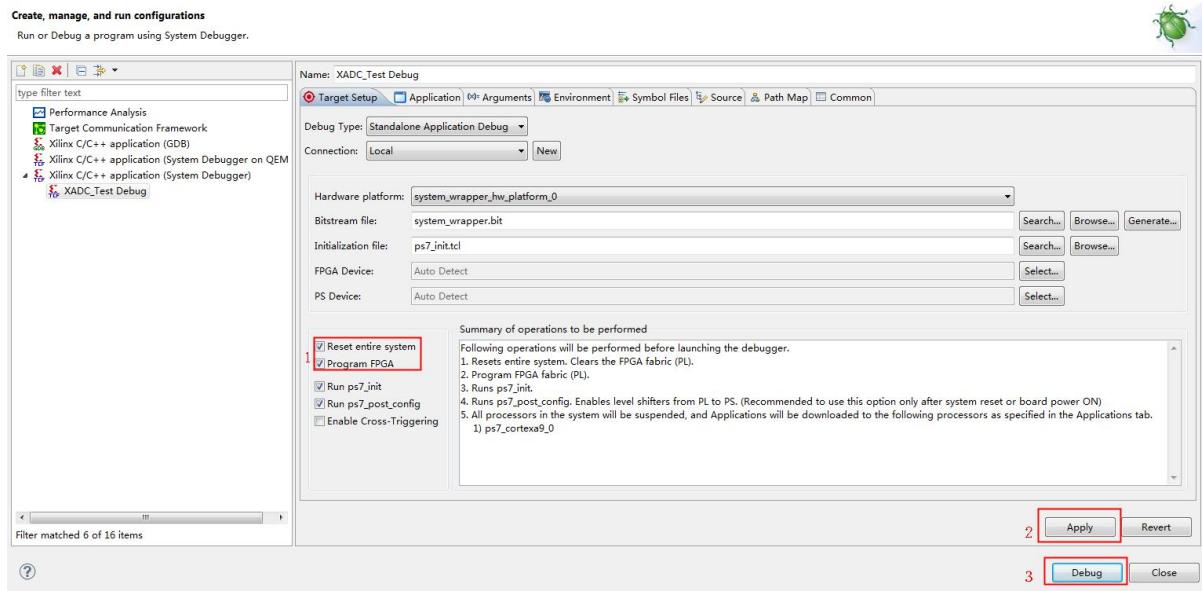
```
VccPAuxData) ;  
  
    VDDRRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCPDR0);  
    VDDRData = XAdcPs_RawToVoltage(VDDRRawData);  
    printf("Raw VccDDR %lu Real VccDDR %f \n\r", VDDRRawData, VDDRData);  
}  
  
return 0;  
}
```

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：

```
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

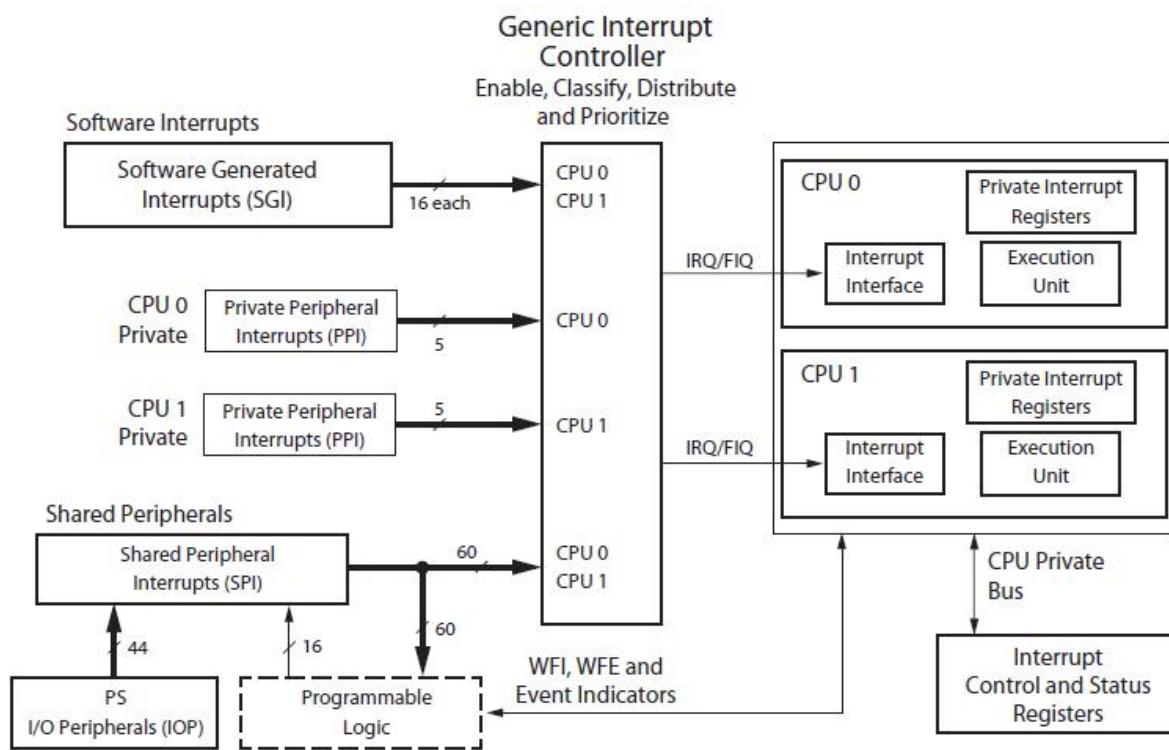
6.5 本章小结

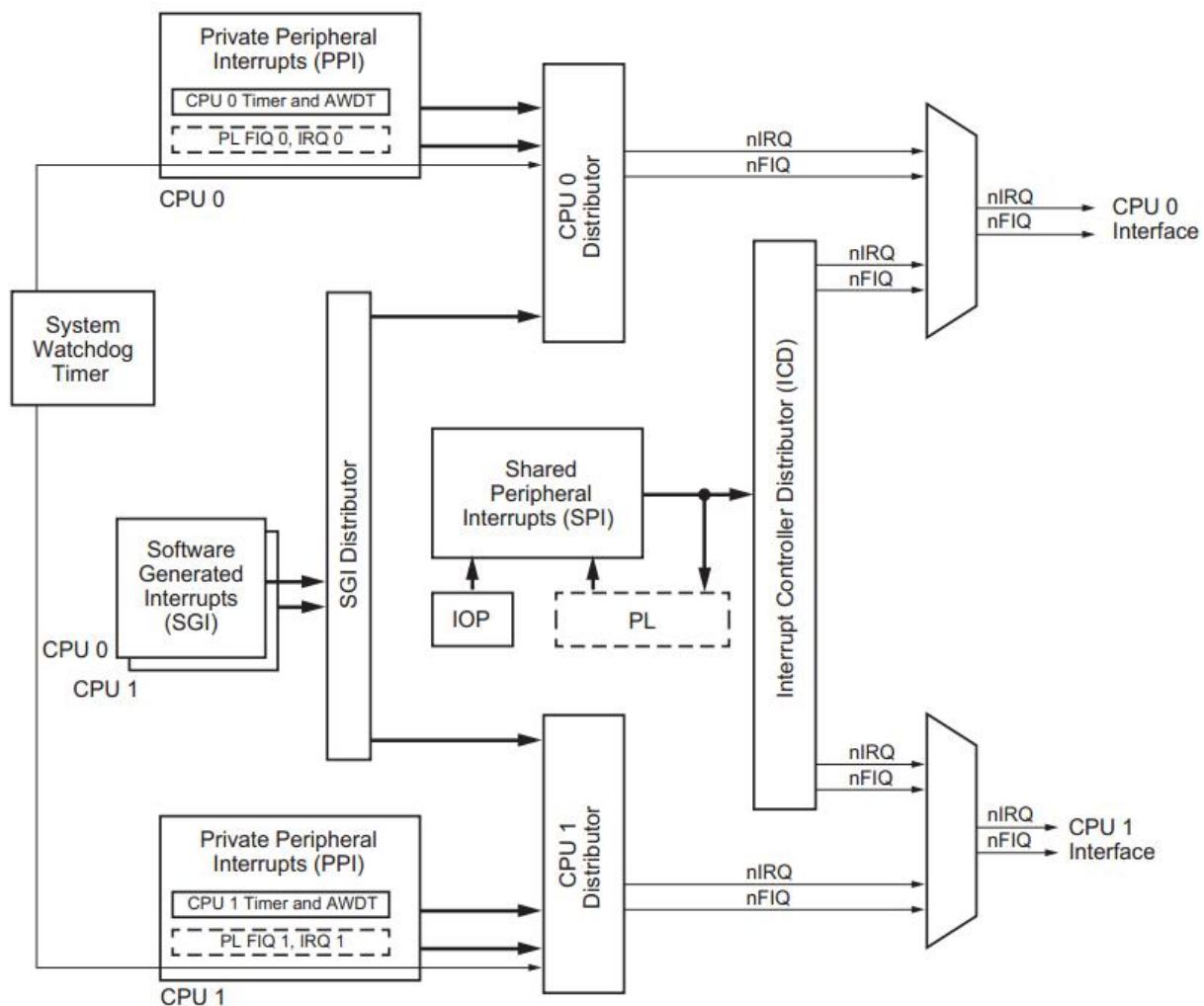
本章讲解了如果采集片上电压以及温度的方法，这个实验在实际工程运用中可以通过测试自生的电压和稳定判断系统是否可以正常工作，是否要做出一些报警之类的行动。

S02_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 私有端口中断

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。

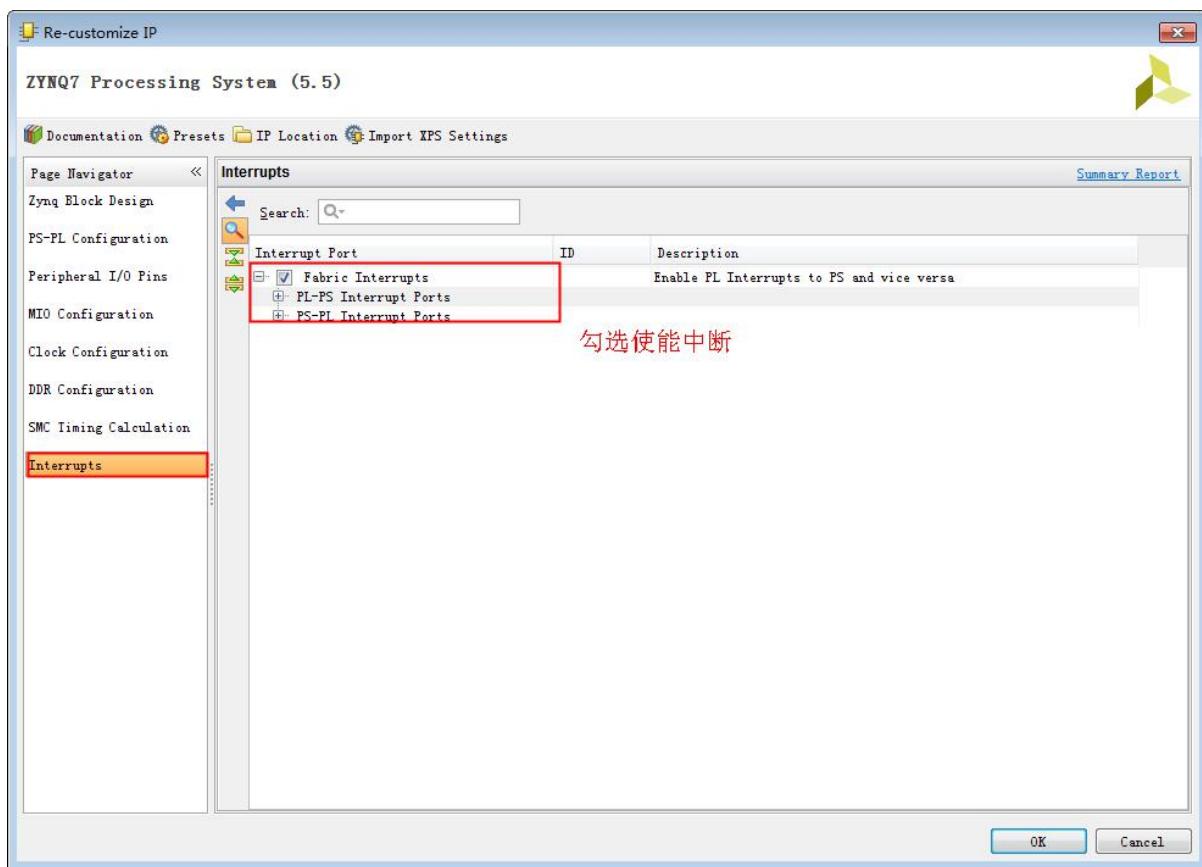
7.2 搭建硬件地址

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。

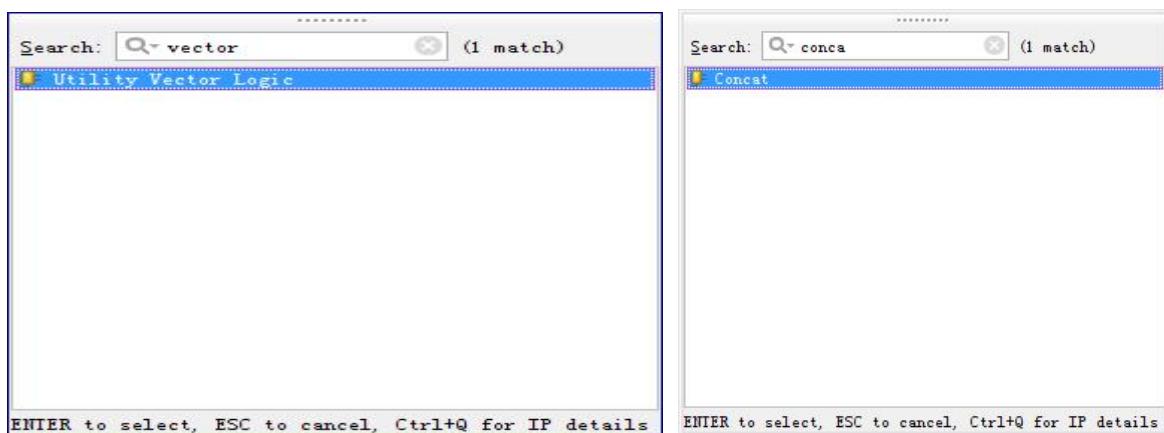
Step2:创建一个 BD 文件，并命名为 system。

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。

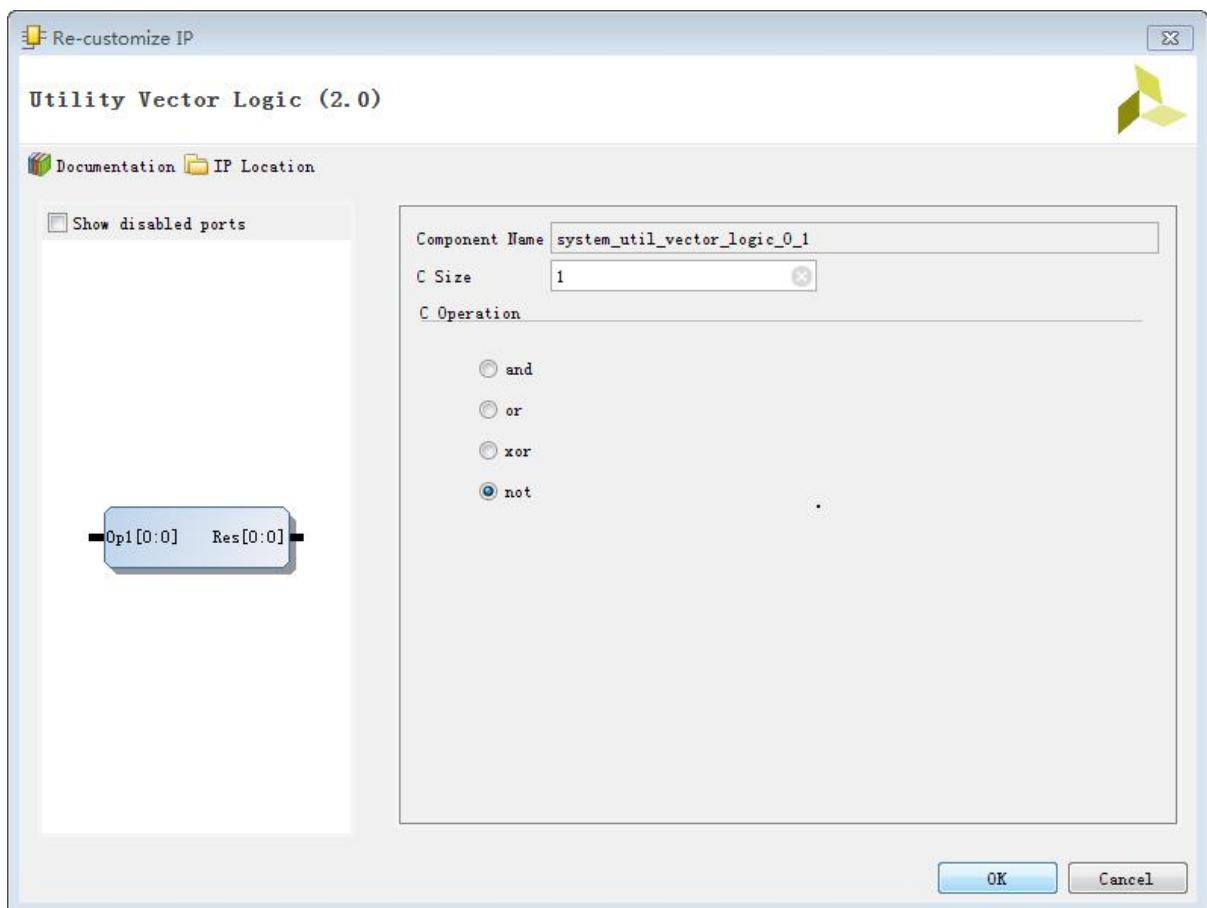
Step4: 在 ZYNQ7 Processing System 配置窗口中，使能中断，单击 OK 完成配置。



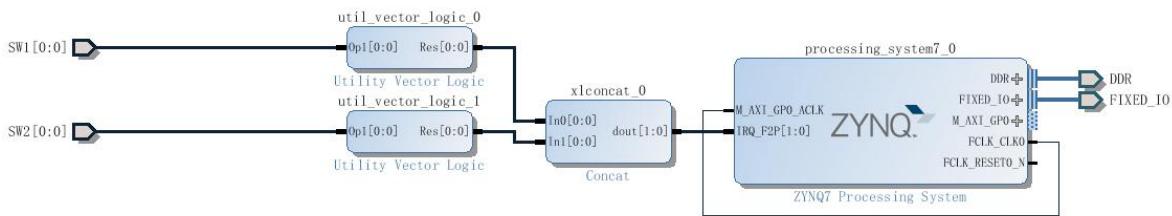
Step5:单击添加 IP 按钮，添加两个逻辑门模块和一个 concat IP。



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: 添加一个约束文件，打开对应自己硬件的原理图，查看按键部分引脚连接情况，完成约束。Miz702 约束文件如下所示：

```

set_property PACKAGE_PIN T18 [get_ports {SW1[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW1[0]}]

set_property PACKAGE_PIN R18 [get_ports {SW2[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW2[0]}]

```

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，并添加一个 main.c 的文件。

Step3: 在 main.c 文件中添加以下程序，按 Ctrl+S 保存后自动开始编译。

```

#include <stdio.h>
#include "xscugic.h"
#include "xil_exception.h"

#define INT_CFG0_OFFSET 0x00000C00

// Parameter definitions
#define SW1_INT_ID 61
#define SW2_INT_ID 62
#define INTC_DEVICE_ID XPAR_PS7_SCUGIC_0_DEVICE_ID

```

```
#define INT_TYPE_RISING_EDGE    0x03
#define INT_TYPE_HIGHLEVEL      0x01
#define INT_TYPE_MASK           0x03

static XScuGic INTCInst;

static void SW_intr_Handler(void *param);
static int IntcInitFunction(u16 DeviceId);

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

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

    intType &= INT_TYPE_MASK;
    mask = XScuGic_DistReadReg(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);
}

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;
}

int main(void)
{
    print("PL int test\n\r");
    IntcInitFunction(INTC_DEVICE_ID);
    while(1);
```

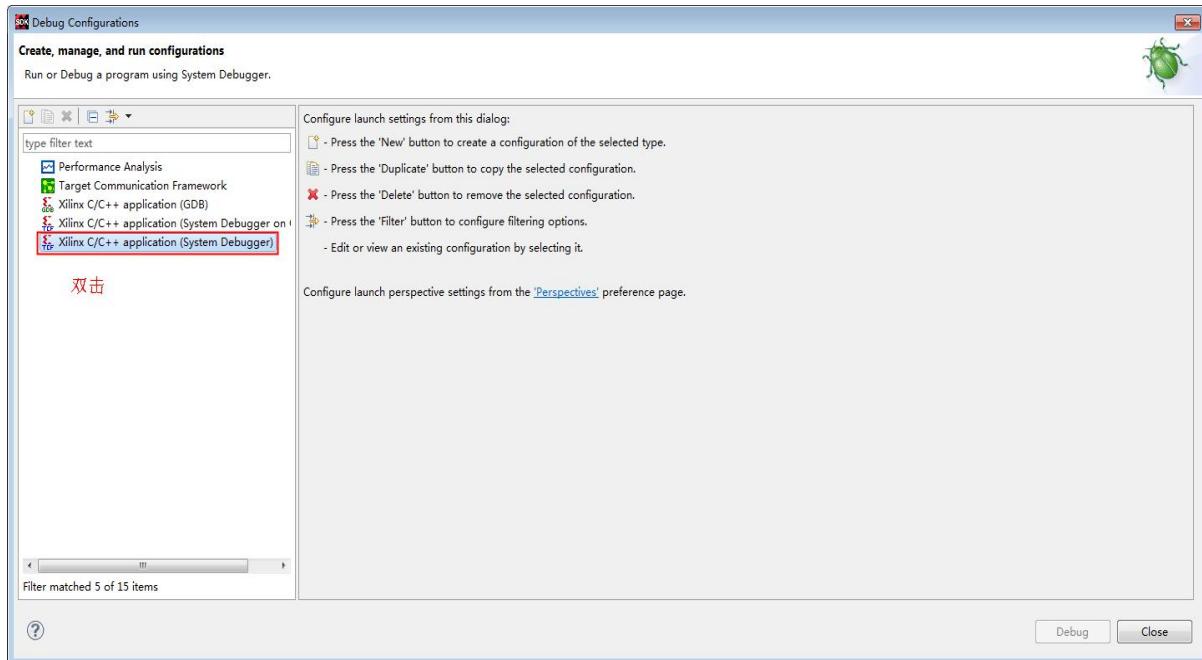
```

    return 0;
}

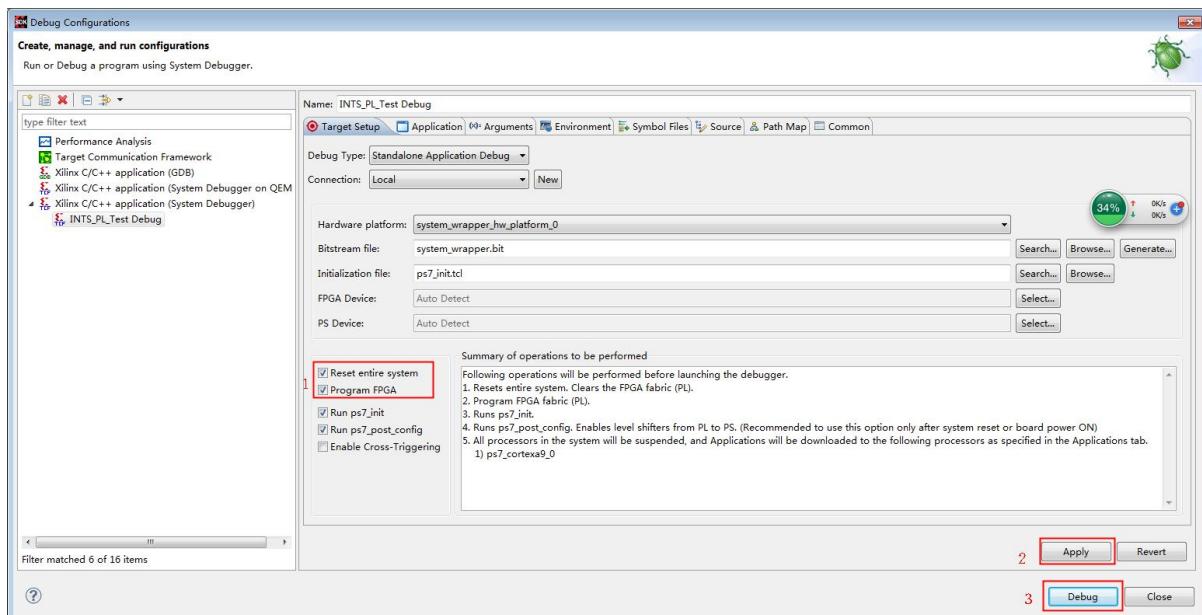
```

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。



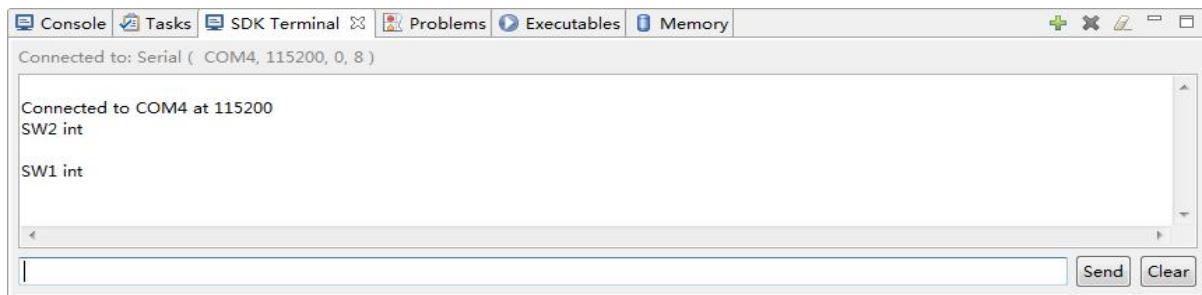
Step6: 设置系统调试。



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：



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。

```
#ifdef __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_DistReadReg(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)
#define INT_CFG0_OFFSET 0x00000C00
int IntcInitFunc()
{
    XScuGic_Config config;
    int status;
    // Interrupt
    IntcConfig =
    status = XScuGic_DistReadReg(InstancePtr, INT_CFG0_OFFSET);
    if(status != 0)
        // Call to interrupt setup
}

```

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 里的一个基址 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)))

***** Function Prototypes *****/

```

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 & (~C0000000) = F0F01C0C`。下一句又是一个运算，这次我们直接计算：`F0F01C0C | 3FFFFFFF = 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。

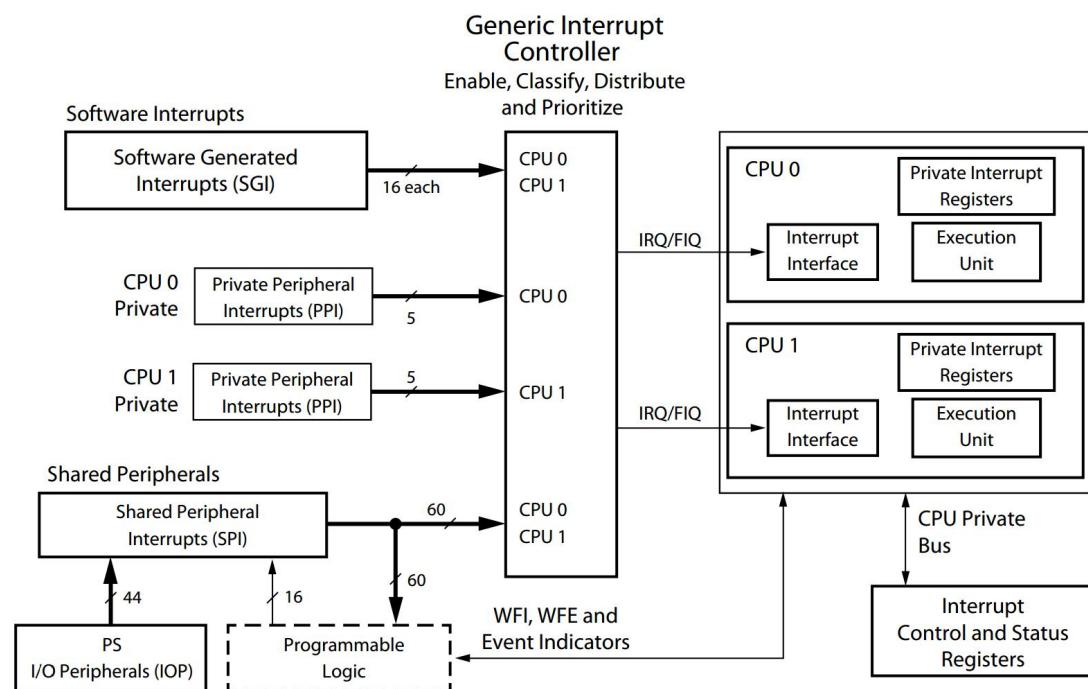
S02_CH08_ZYNQ 定时器中断实验

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。

中断体系结构框图如下：



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 Miz702 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)

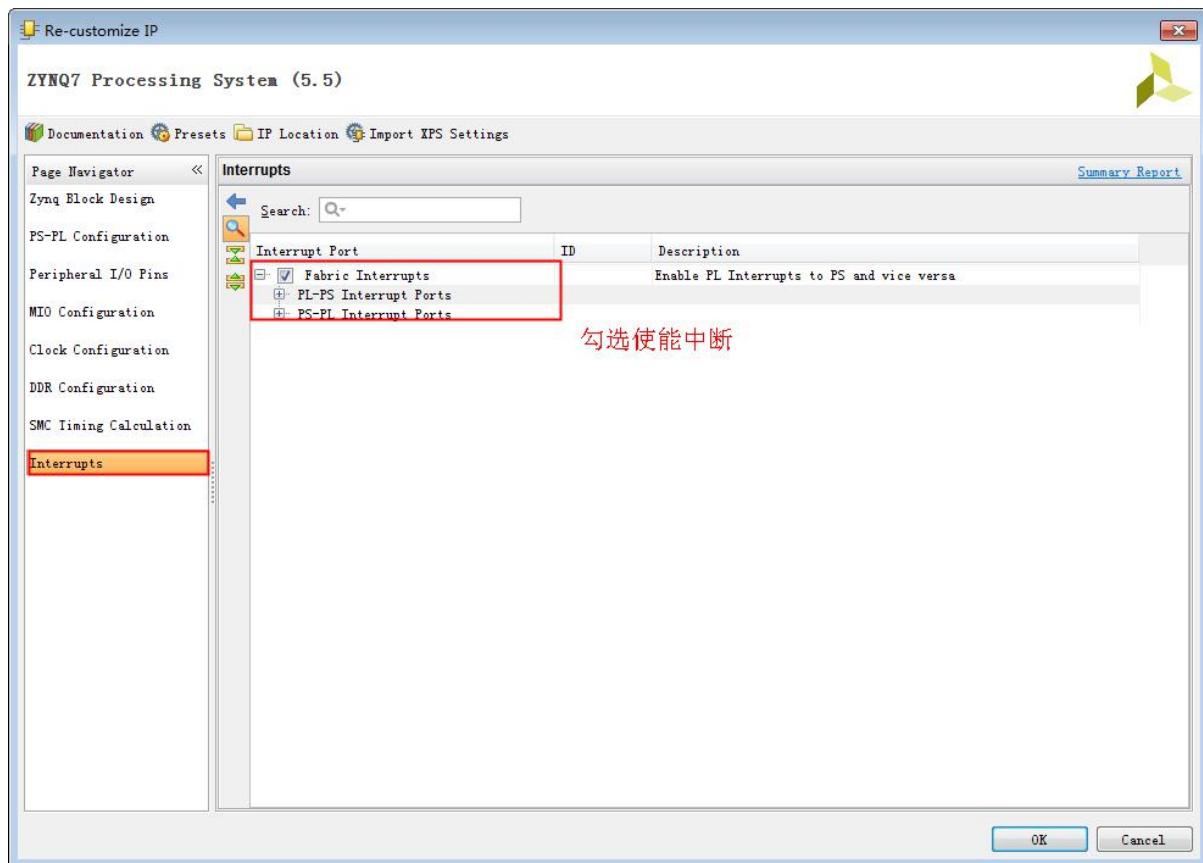
8.2 搭建硬件工程

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。

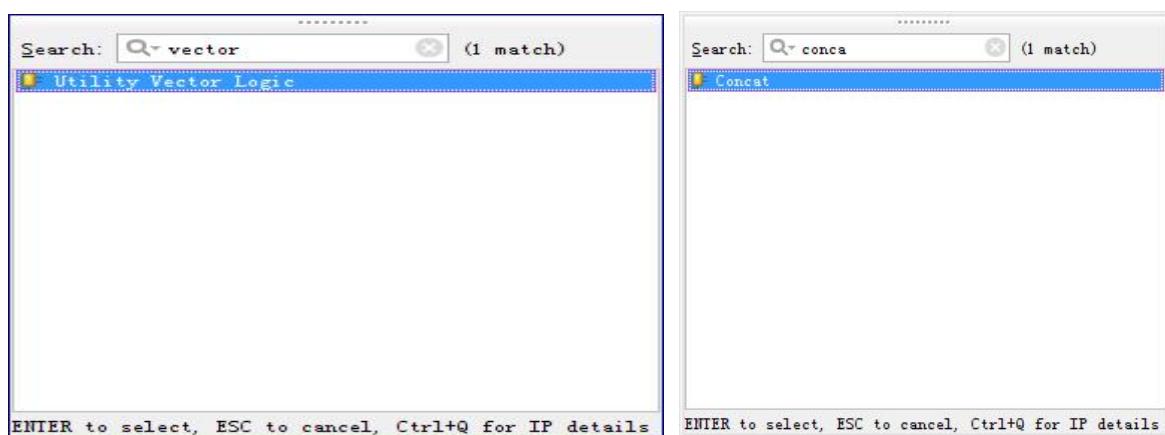
Step2:创建一个 BD 文件，并命名为 system。

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。

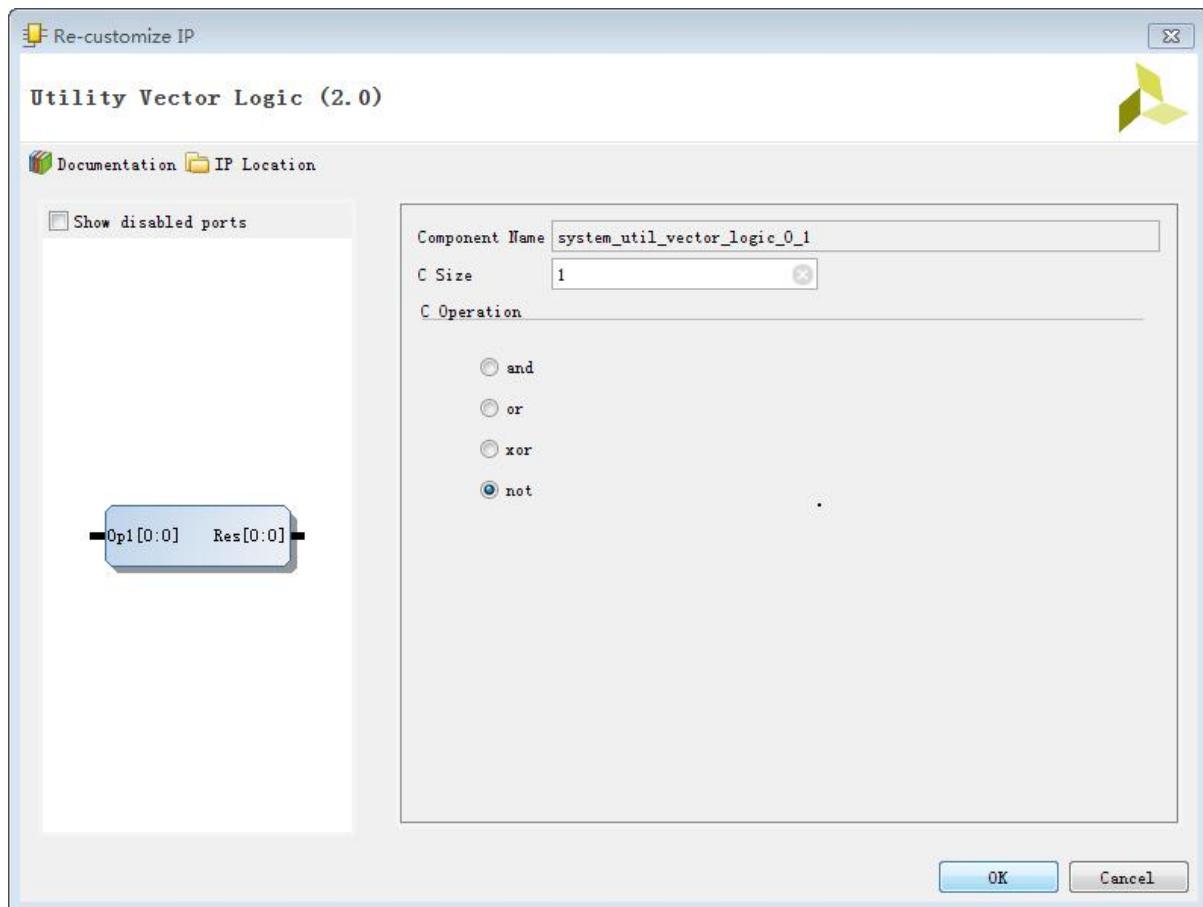
Step4：在 ZYNQ7 Processing System 配置窗口中，使能中断，单击 OK 完成配置。



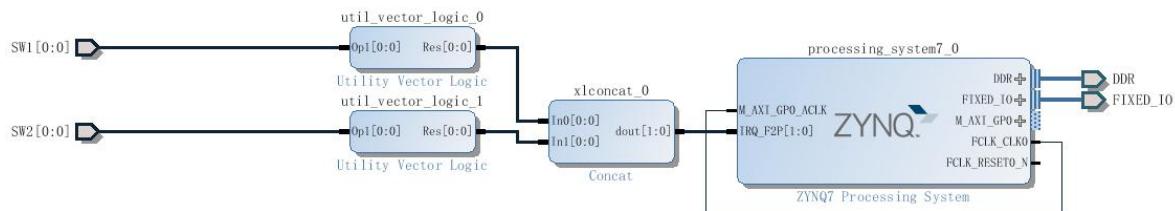
Step5: 单击添加 IP 按钮，添加两个逻辑门模块和一个 concat IP。



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：添加一个约束文件，打开对应自己硬件的原理图，查看按键部分引脚连接情况，完成约束。Miz702 约束文件如下所示：

```
set_property PACKAGE_PIN T18 [get_ports {SW1[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW1[0]}]
```

```
set_property PACKAGE_PIN R18 [get_ports {SW2[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW2[0]}]
```

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，并添加一个 main.c 的文件。

Step3：在 main.c 文件中添加以下程序，按 Ctrl+S 保存后自动开始编译。

```
/*
 * main.c
 *
 * Created on: 2016 年 6 月 26 日
 *      Author: Administrator
 */

#include <stdio.h>
#include "xadcps.h"

#include "xil_types.h"
#include "Xscugic.h"
#include "Xil_exception.h"
#include "xscutimer.h"

//timer info
#define TIMER_DEVICE_ID      XPAR_XSCUTIMER_0_DEVICE_ID
#define INTC_DEVICE_ID        XPAR_SCUGIC_SINGLE_DEVICE_ID
#define TIMER_IRPT_INTR      XPAR_SCUTIMER_INTR

#define TIMER_LOAD_VALUE     0x13D92D3F

static XScuGic Intc; //GIC
static XScuTimer Timer; //timer

static void TimerIntrHandler(void *CallBackRef)
```

```
{  
  
    static int sec = 0; //计数  
    XScuTimer *TimerInstancePtr = (XScuTimer *)CallBackRef;  
    XScuTimer_ClearInterruptStatus(TimerInstancePtr);  
    sec++;  
    printf("%d Second\n\r",sec); //每秒打印输出一次  
}  
  
void SetupInterruptSystem(XScuGic *GicInstancePtr,  
                         XScuTimer *TimerInstancePtr, u16 TimerIntrId)  
{  
  
    XScuGic_Config *IntcConfig; //GIC config  
    Xil_ExceptionInit();  
    //initialise the GIC  
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);  
    XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,  
                          IntcConfig->CpuBaseAddress);  
  
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,  
                               (Xil_ExceptionHandler)XScuGic_InterruptHandler,//connect to the hardware  
                               GicInstancePtr);  
  
    XScuGic_Connect(GicInstancePtr, TimerIntrId,  
                    (Xil_ExceptionHandler)TimerIntrHandler,//set up the timer interrupt  
                    (void *)TimerInstancePtr);  
  
    XScuGic_Enable(GicInstancePtr, TimerIntrId);//enable the interrupt for the Timer at GIC  
    XScuTimer_EnableInterrupt(TimerInstancePtr);//enable interrupt on the timer  
    Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ); //Enable interrupts in the Processor.  
}  
  
int main()  
{
```

```

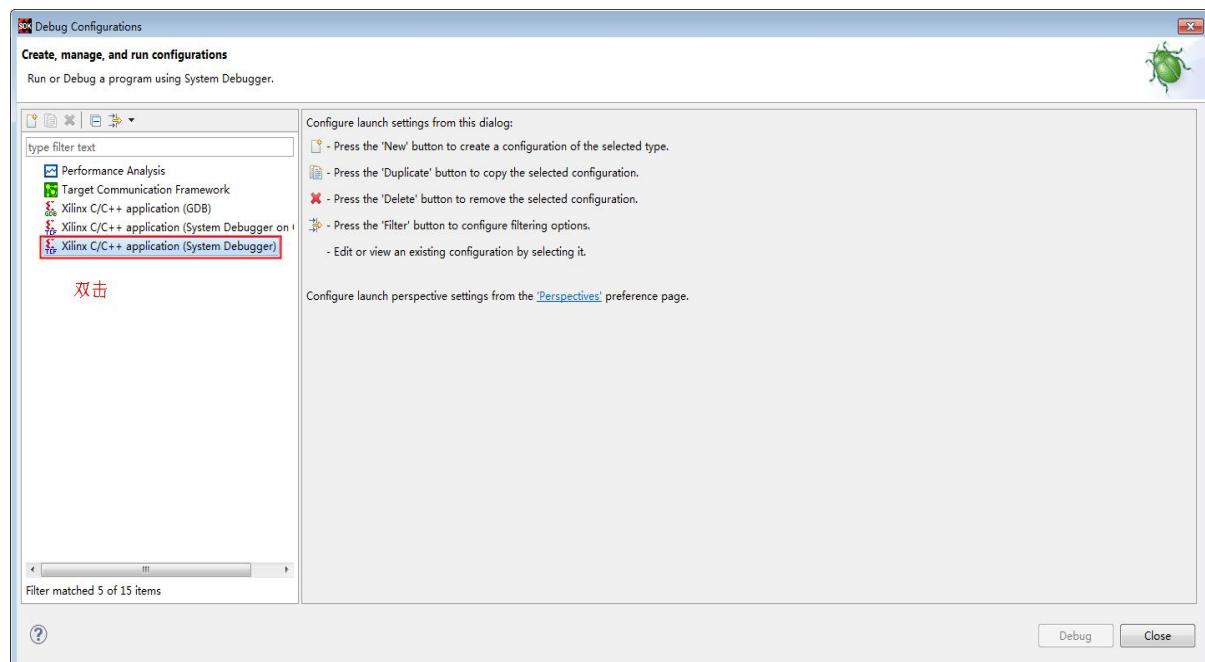
XScuTimer_Config *TMRConfigPtr;      //timer config
printf("-----START-----\n");
//私有定时器初始化
TMRConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
XScuTimer_CfgInitialize(&Timer, TMRConfigPtr,TMRConfigPtr->BaseAddr);
//set up the interrupts
SetupInterruptSystem(&Intc,&Timer,TIMER_IRPT_INTR);
//加载计数周期，私有定时器的时钟为CPU的一般，为333MHZ,如果计数1S,加载值为
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;
}

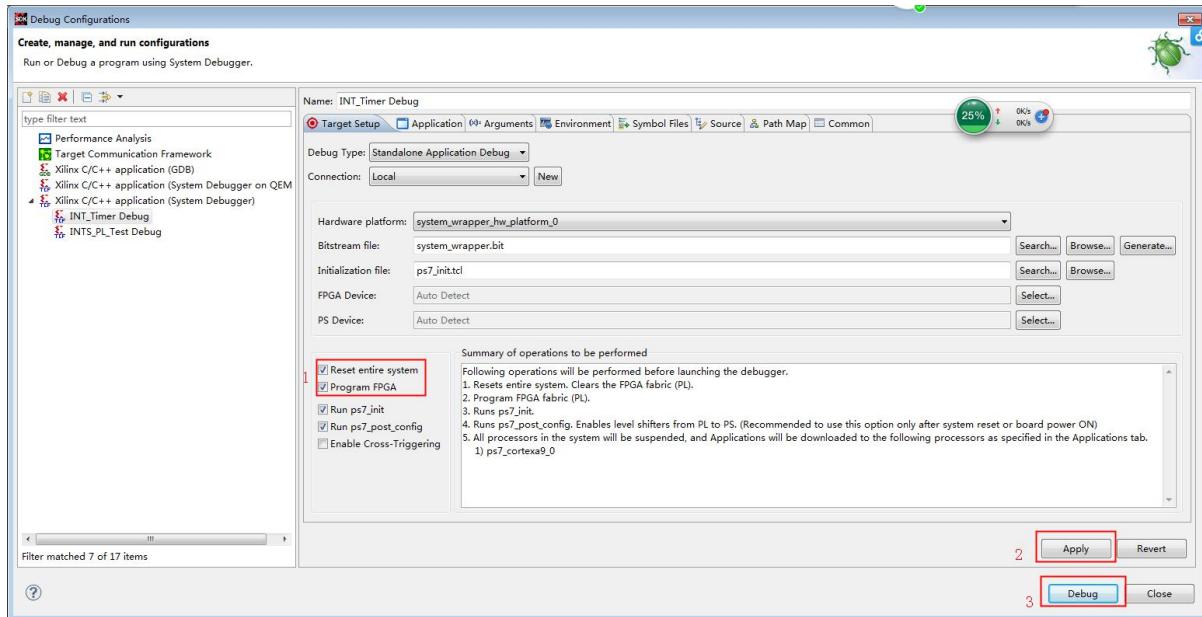
```

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。



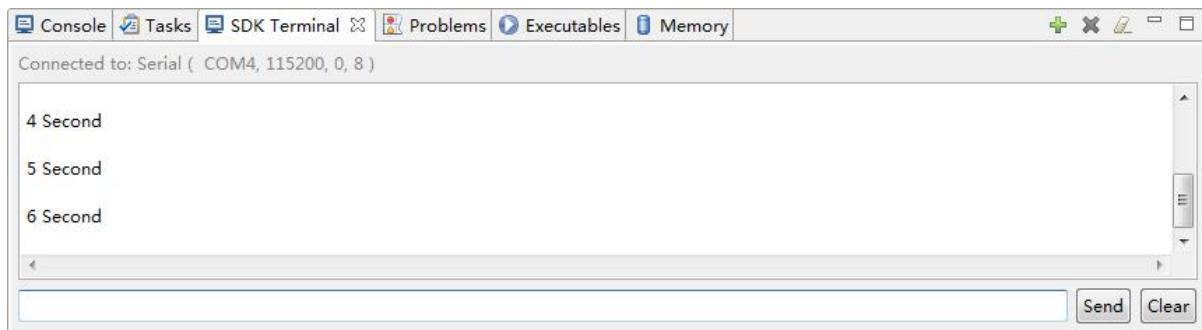
Step6: 设置系统调试。



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：



8.4 程序分析

本章的程序讲解依然从 main 函数处开始。首先我们看看整个程序的结构。

```
int main()
{
    XScuTimer_Config *TMRCConfigPtr;      //timer config
    printf("-----START-----\n");
    //私有定时器初始化
    TMRCConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
    XScuTimer_CfgInitialize(&Timer, TMRCConfigPtr, TMRCConfigPtr->BaseAddr);
    //set up the interrupts
    SetupInterruptSystem(&Intc, &Timer, TIMER_IRPT_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);
    //自动续载
    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地址而已。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.
 *
 * @param InstancePtr is a pointer to the XScuTimer instance.
 * @param ConfigPtr points to the XScuTimer configuration structure.
 * @param EffectiveAddress is the base address for the device. It could be
 * a virtual address if address translation is supported in the
 * system, otherwise it is the physical address.
 *
 * @return
 * - XST_SUCCESS if initialization was successful.
 * - XST_DEVICE_IS_STARTED if the device has already been started.
 *
 * @note None.
 */
s32 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; //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地址。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。

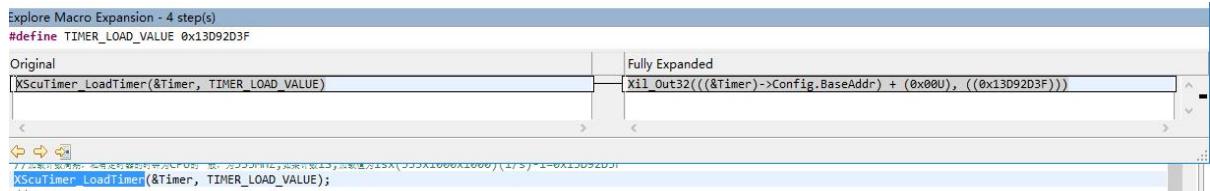
那么，系统是如何做到定时一秒的呢？定时器时间可以通过下式计算：定时时间 = 1/定时器频率 * (预加载值+1)，则可以推算出：预加载值=定时时间*定时器频率-1。定时时间是已知的，如果再知道定时器频率则可以计算出加载的值，查看 xilinx 的编程指导手册 ug585-zynq-7000-TRM 的定时器篇得知：

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 Miz702 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地址，也就是定时器的地址。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。

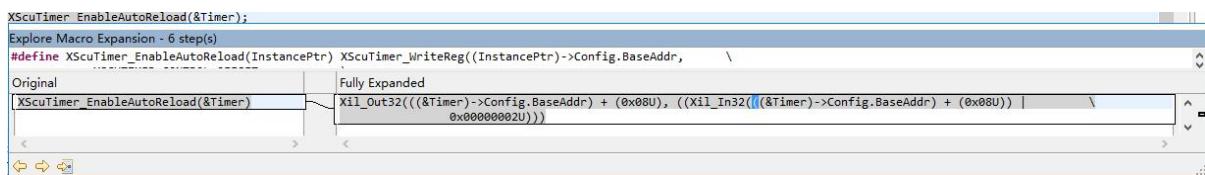
Register ([mpcore](#)) Private_Timer_Load_Register

Name	Private_Timer_Load_Register
Software Name	TIMER_LOAD
Relative Address	0x000000600
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);

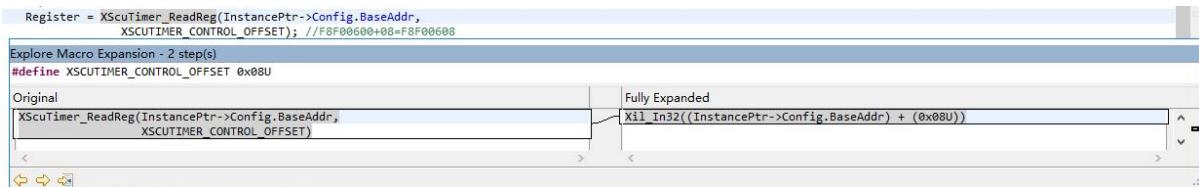
    /*
     * Set the 'timer enable' bit in the register.
     */
    Register |= XSCUTIMER_CONTROL_ENABLE_MASK;

    /*
     * 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，这也是一个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。



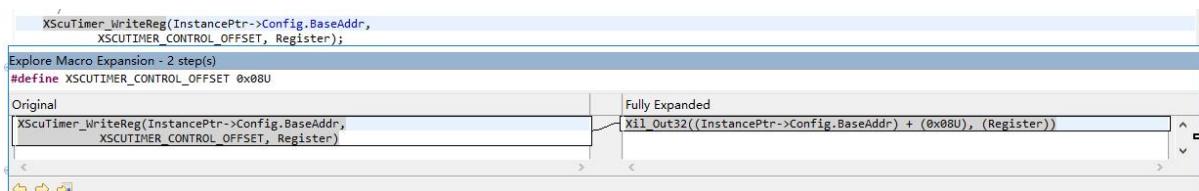
之前的一些初始化程序就跳过不再讲解了，直接看到上图所示的程序，这个程序的分析与我们刚才讲的装载初值的方法是一样的，这里我们可以直接计算此程序读出的寄存器地址：F8F00600+08=F8F00608。

```

Register |= XSCUTIMER_CONTROL_ENABLE_MASK;
/*
 * Update the

```

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。



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。

S02_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，并添加一个 main.c 的文件。

Step4：在 main.c 文件中添加以下程序，按 Ctrl+S 保存后自动开始编译。

```
/*
 * main.c
 *
 * Created on: 2016 年 6 月 26 日
 * Author: Administrator
 */

#include <stdio.h>
#include "xadcps.h"

#include "xil_types.h"
#include "Xscugic.h"
#include "Xil_exception.h"
#include "xuartps.h"

//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

static XScuGic Intc; //GIC
static XUartPs Uart;//uart
```

```
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_RXOVR | (u32)XUARTPS_RXEMPTY |
           (u32)XUARTPS_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);
}

void SetupInterruptSystem(XScuGic *GicInstancePtr,
                         XUartPs *UartInstancePtr, u16 UartIntrId)
{
    XScuGic_Config *IntcConfig; //GIC config
    Xil_ExceptionInit();
    //initialise the GIC
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,
                          IntcConfig->CpuBaseAddress);

    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler, //connect to the hardware
                                GicInstancePtr);
    Xil_ExceptionEnable();
    XScuGic_Connect(GicInstancePtr, UartIntrId,
                     (Xil_InterruptHandler)UartIntrHandler, //set up the timer interrupt
                     (void *)UartInstancePtr);

    XScuGic_Enable(GicInstancePtr, UartIntrId); //enable the interrupt for the Timer at GIC
    XUartPs_SetInterruptMask(UartInstancePtr,           XUARTPS_IXR_RXOVR/*
XUARTPS_IXR_TXEMPTY | XUARTPS_IXR_TNFUL*/ );
    // XUartPs_EnableUart(UartInstancePtr); //enable interrupt on the timer
}
```

```

        Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ); //Enable interrupts in the Processor.

    }

int main()
{
    XUartPs_Config *UartConfigPtr;      //timer config
//    printf("-----START-----\n");
    UartConfigPtr = XUartPs_LookupConfig(UART_DEVICE_ID);

    XUartPs_CfgInitialize(&Uart,UartConfigPtr,UartConfigPtr->BaseAddress);
    //set up the interrupts
    SetupInterruptSystem(&Intc,&Uart,UART_IRPT_INTR);

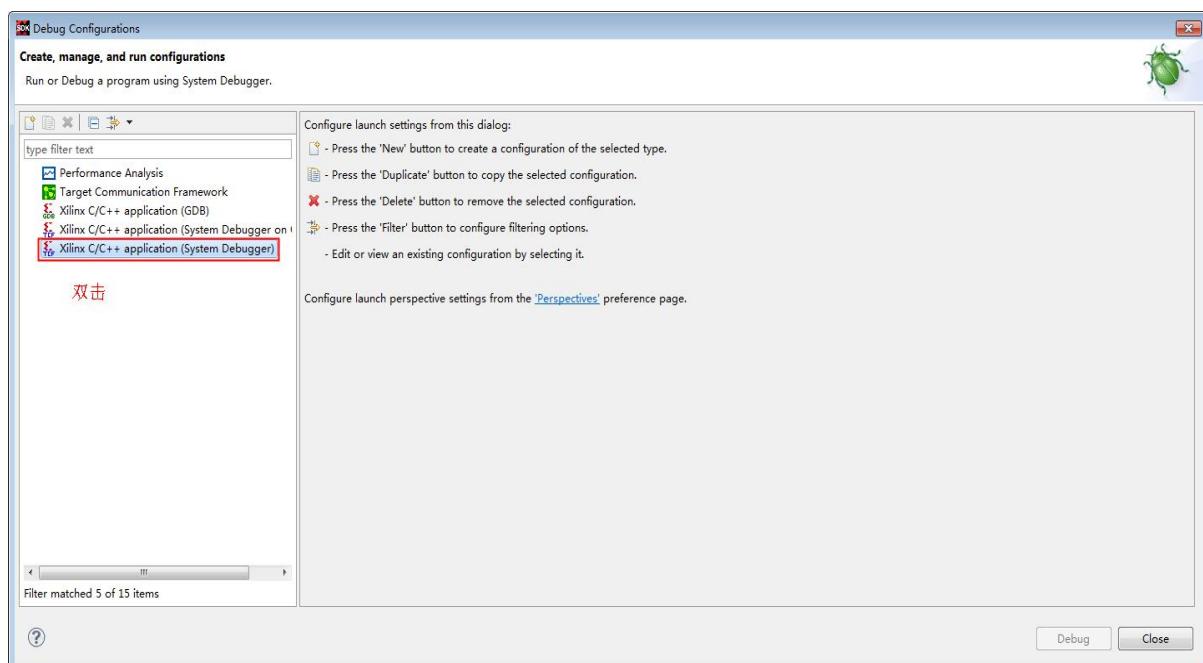
    while(1);

    return 0;
}

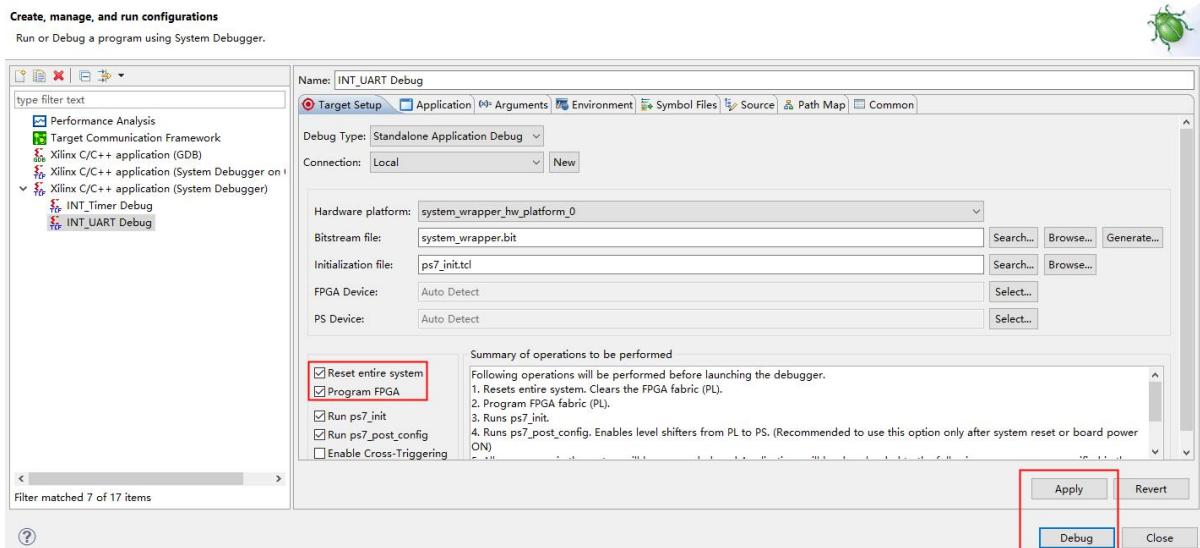
```

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。



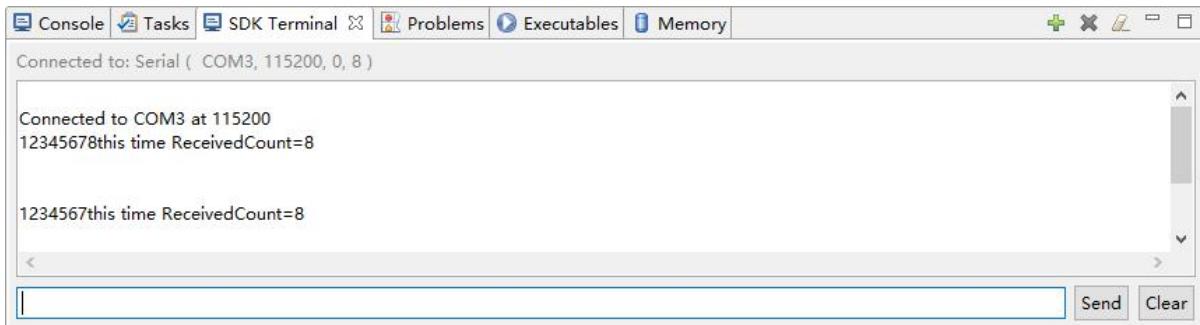
Step6: 设置系统调试。



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_IRQT_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 Handler,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 __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;
```

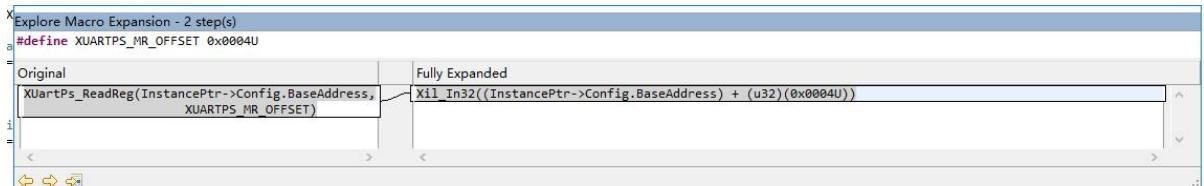
接下来，程序将 UART 的波特率设置为了 115200。

```
BaudRate = (u32)XUARTPS_DFT_BAUDRATE;
Status = XUartPs_SetBaudRate(InstancePtr, BaudRate);
if (Status != (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
    }
};
```

可以知道，此处的基地址为 0xE000100，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-scalar 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));
```

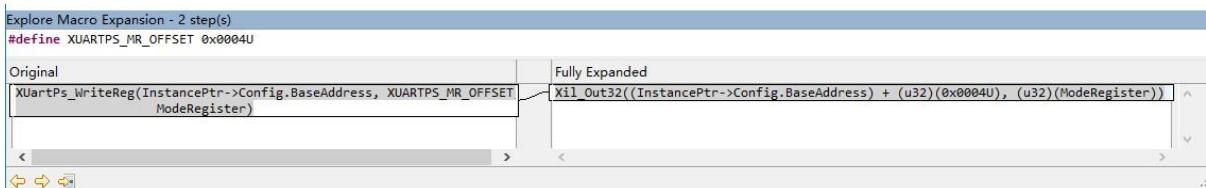
首先得到方框中这三个参数的值。这里我们已经查看程序得知这三个值分别为：6, A0, 38。
然后进行运算：ModeRegister=E0001004 & (~(6|A0|38))=E0001004 & 11 =0。

```
/* 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|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, //每次循环发送接收到的数据
                             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。

S02_CH10_User GPIO 实验

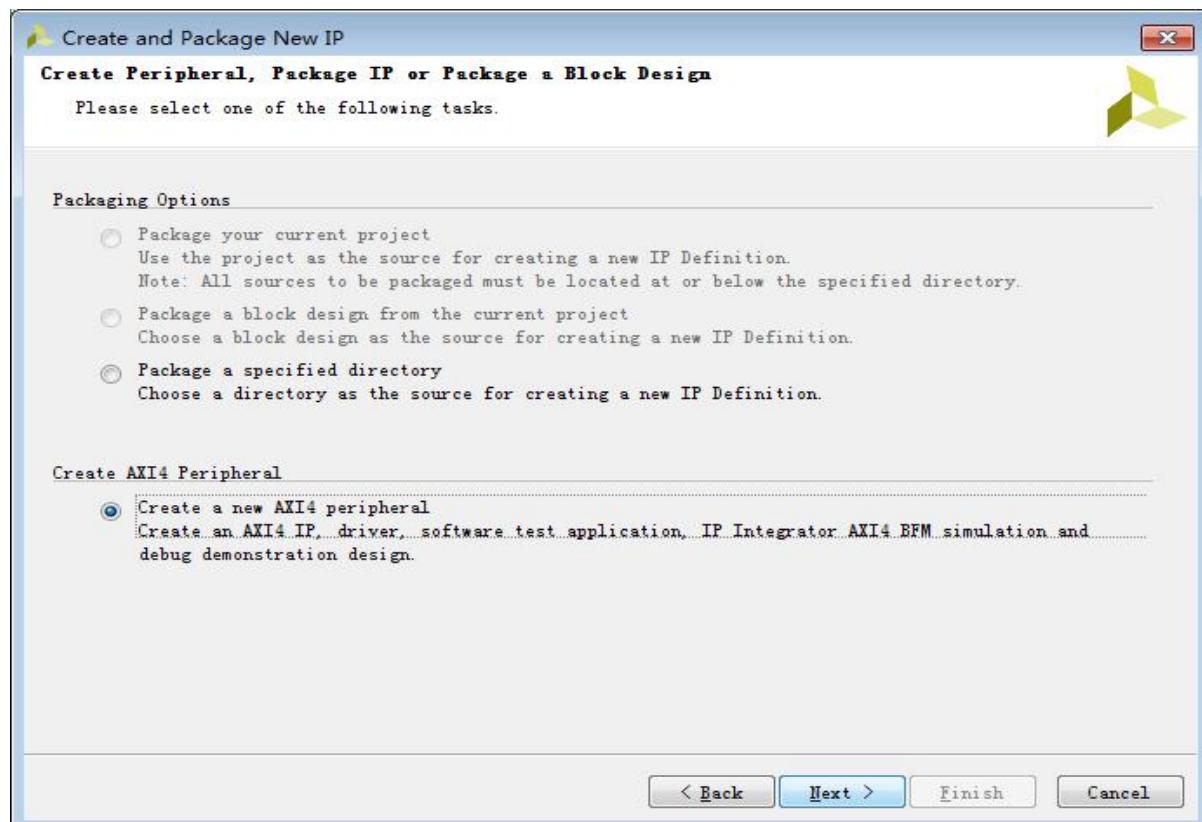
在之前的第四章课程中，我们详细的讲解了如何在 VIVADO 软件下封装一个简单的流水灯程序。在 ZYNQ 开发过程中，有时候我们可能会需要与 ARM 硬核进行通信，在这种情况下，可能就需要用到更高速的接口与 ARM 通信。本章就将讲解如何创建一个基于高速的 AXI 总线的 IP。本章将带领大家创建一个带 AXI 总线接口的自定义 GPIO 模拟的流水灯实验。通过这种方法，我们可以在 GPIO 资源缺乏的情况下，利用 PL 的资源来扩充 GPIO 资源。

10.1 创建 IP

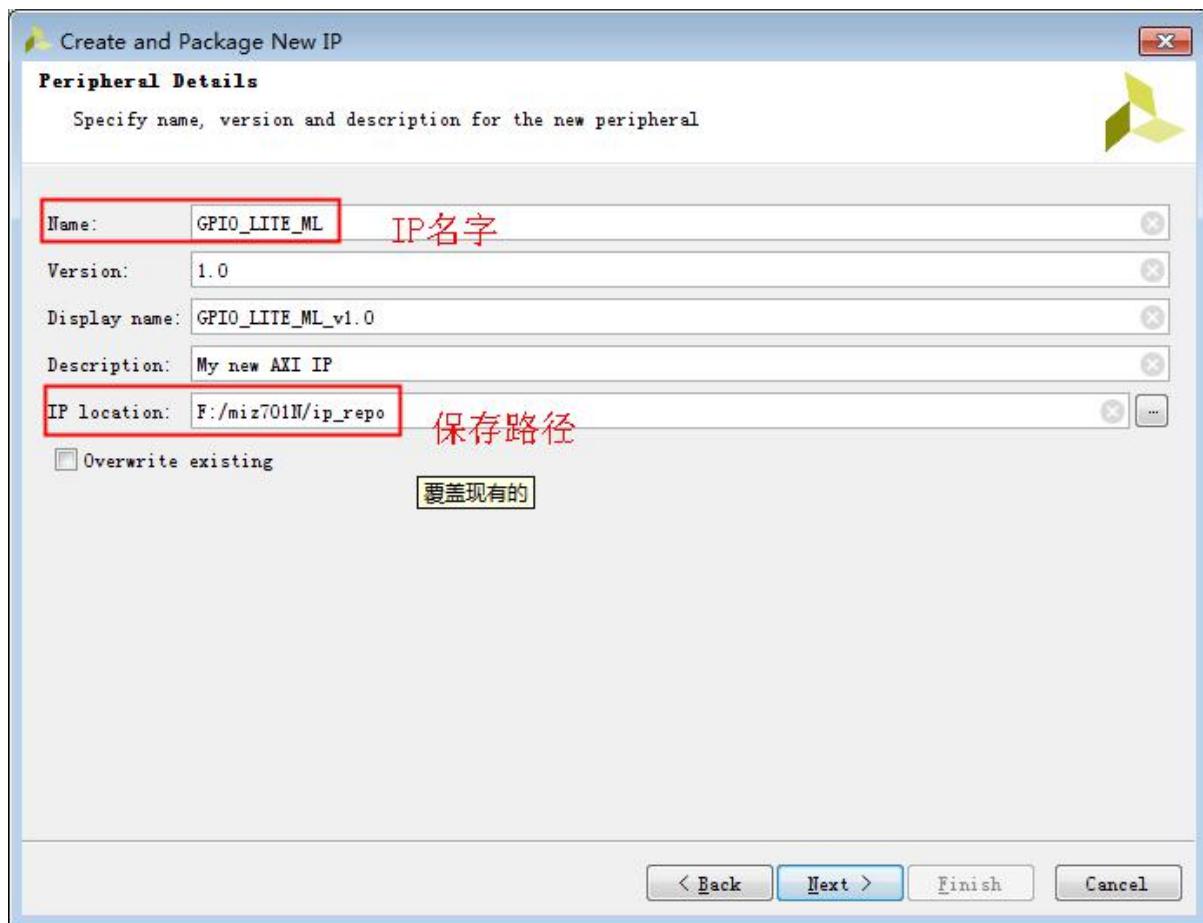
Step1：打开 VIVADO 软件，新建一个工程。

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

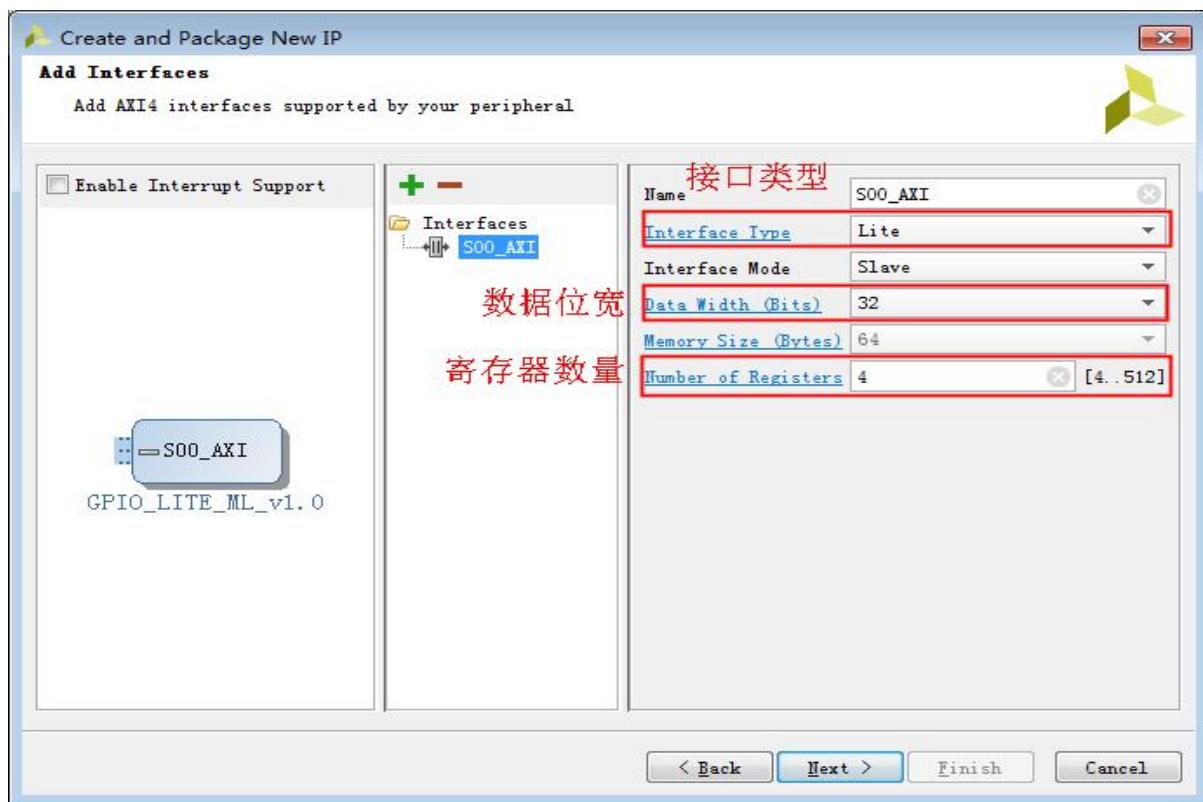
Step3：单击 Next，选择 Create a new AXI4 peripheral，单击 Next。



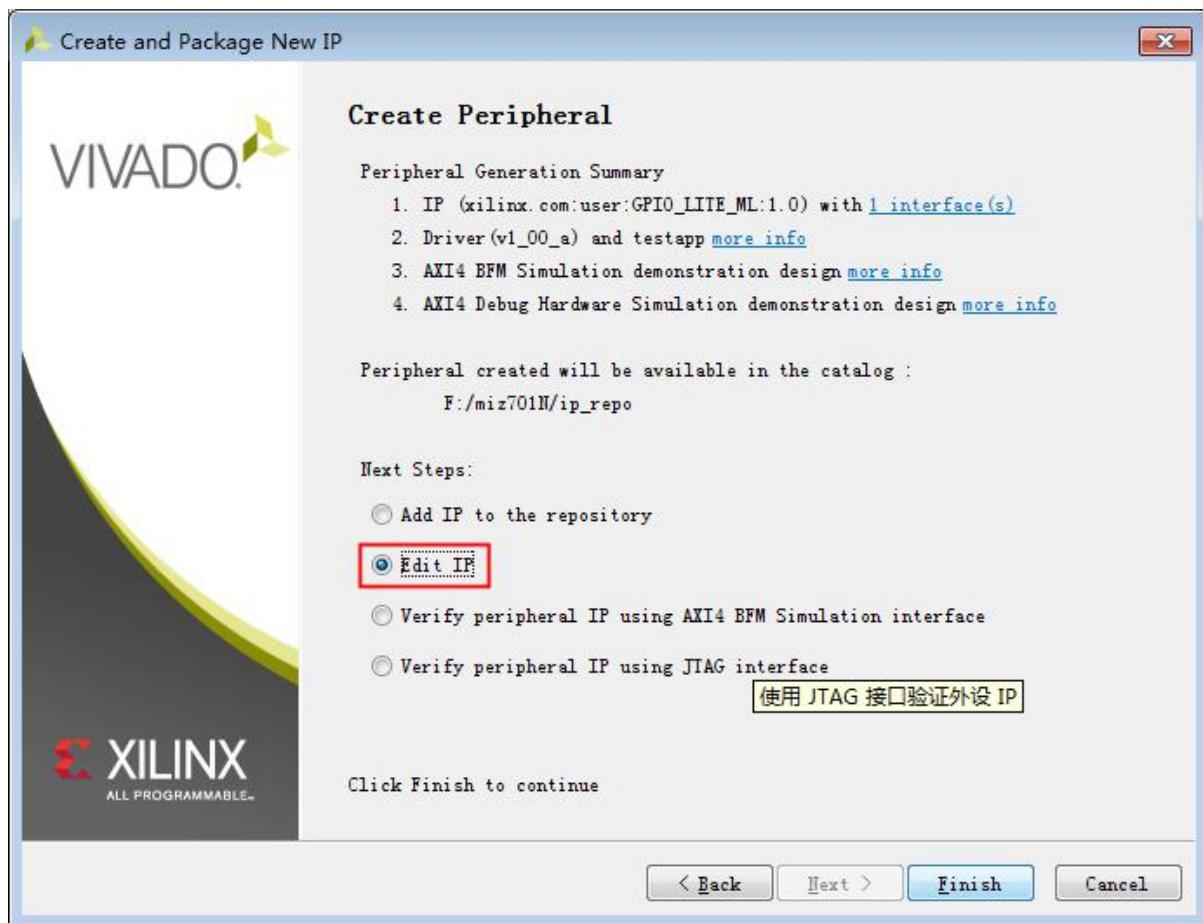
Step4：输入要创建的 IP 名字，此处我们命名为 GPIO_LITE_ML,选择好保存路劲,单击 Next。



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
// 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
    end

```

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

```
        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_already 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_SO0_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_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

```

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

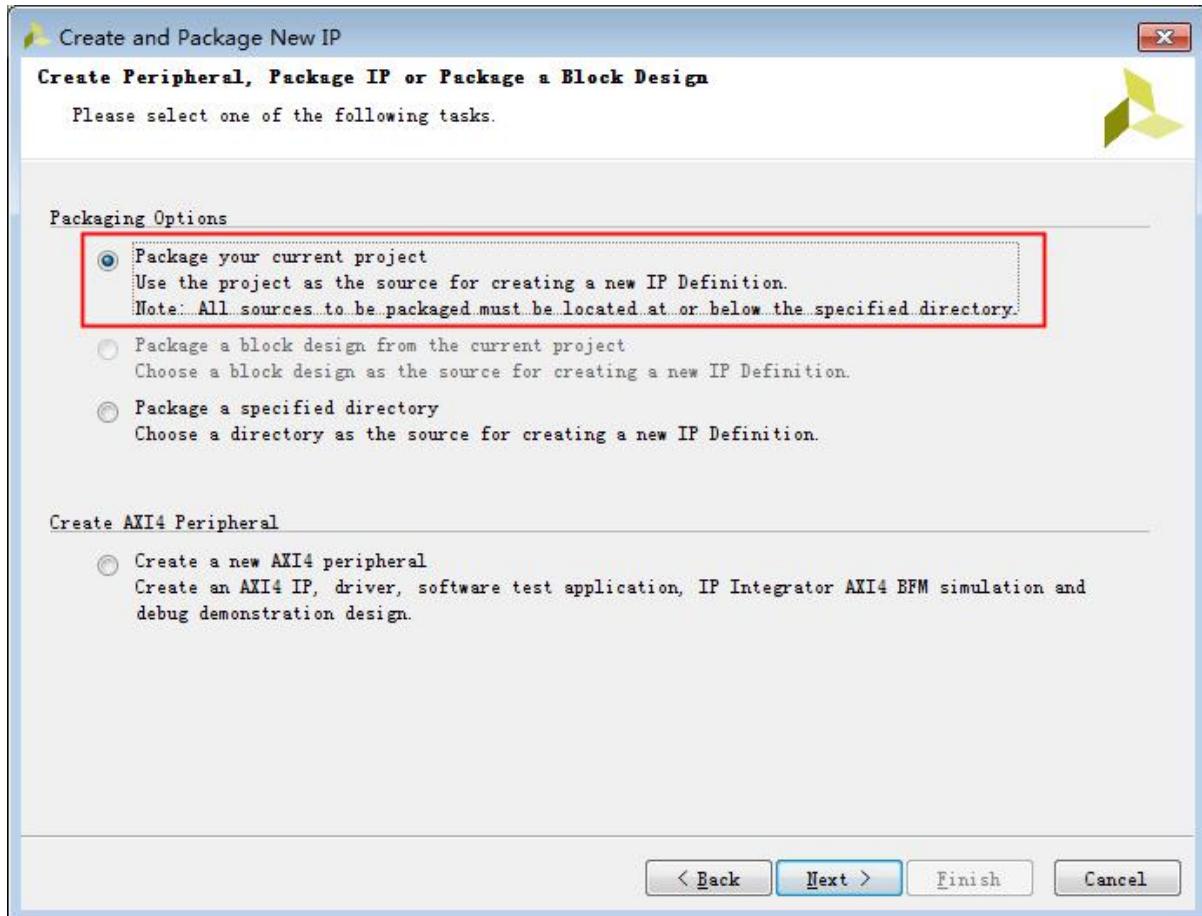
// User logic ends

endmodule
```

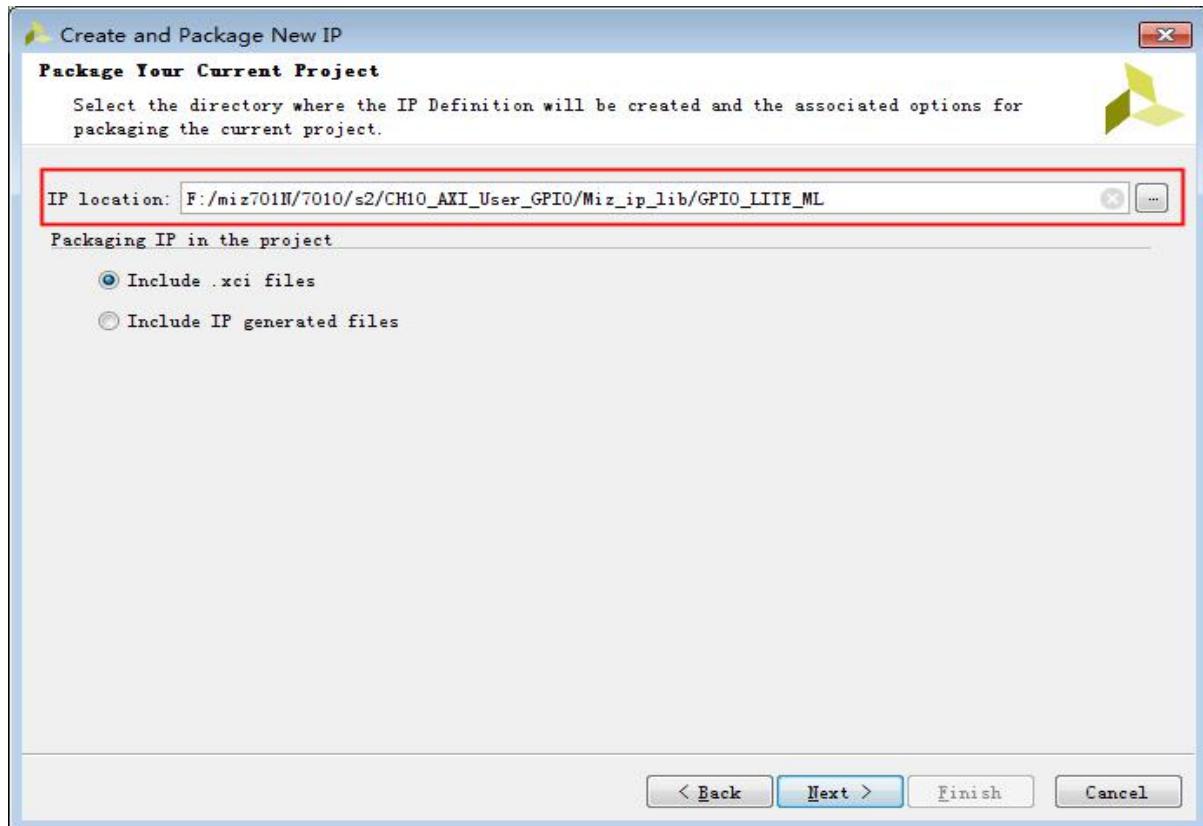
以上的程序也只是在原来的程序的基础上增加了一个用户端口而已，并无什么大的改变。

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。

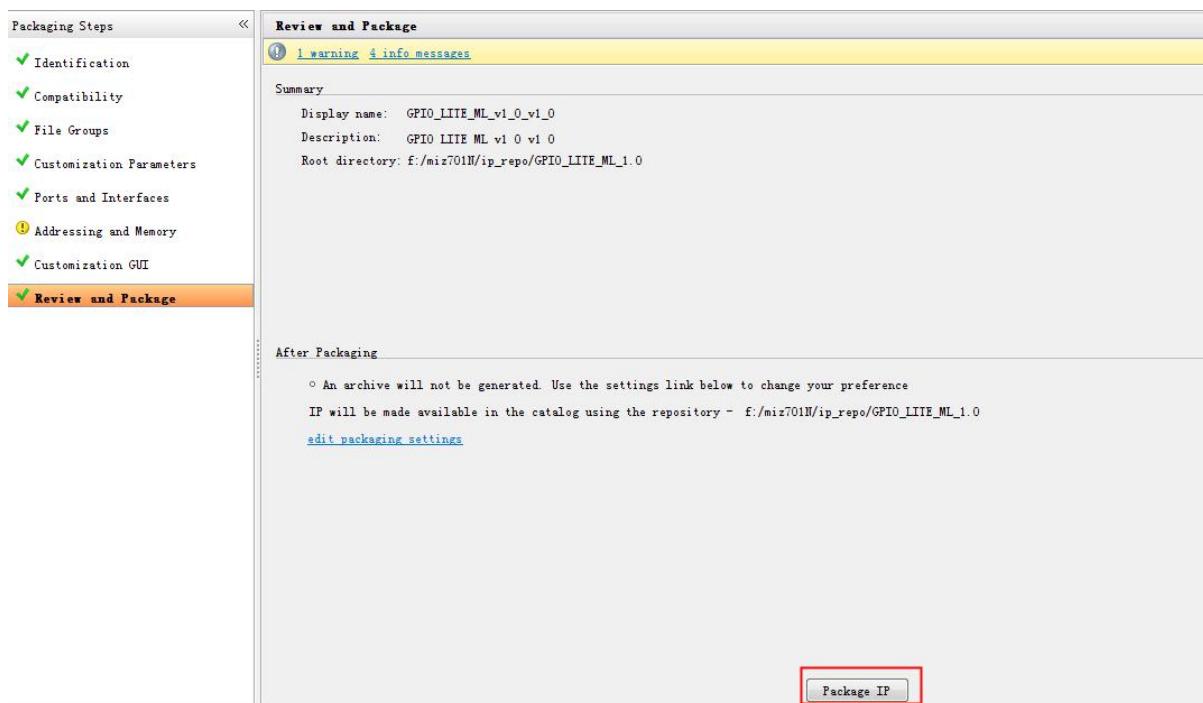


Step11：选择保存的路劲，单击 Next。



Step12：选择 Overwrite，然后单击 Finish。

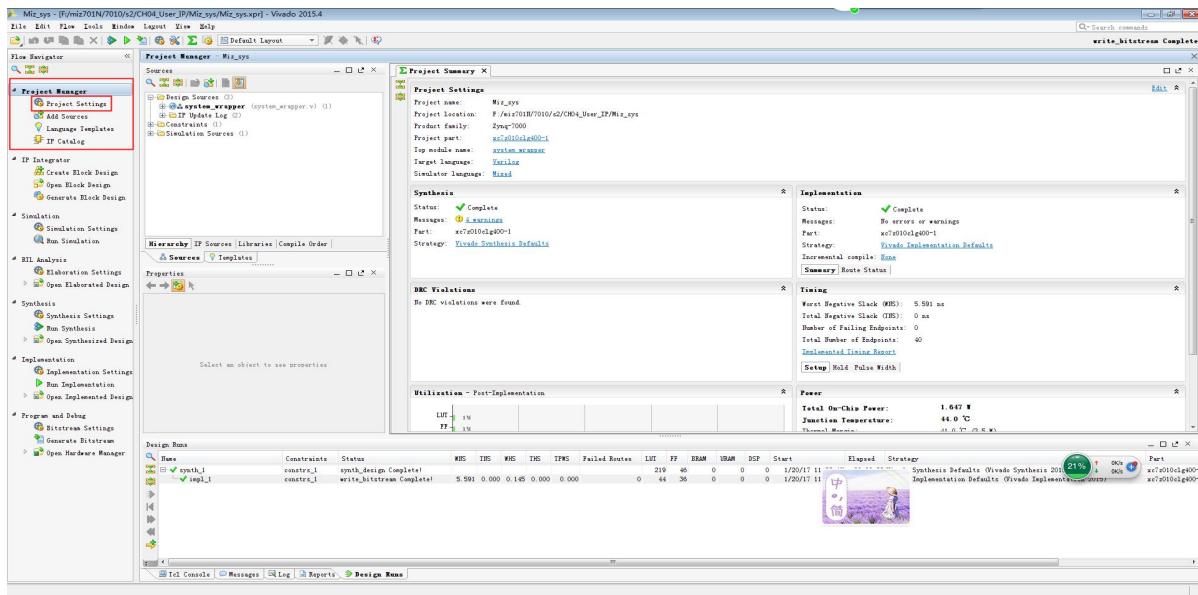
Step13：在新弹出的窗口中，我们注意到有一个警告，直接忽略它，选择 Review and package IP 选项，单击底部的 package IP 按钮完成 IP 的创建。



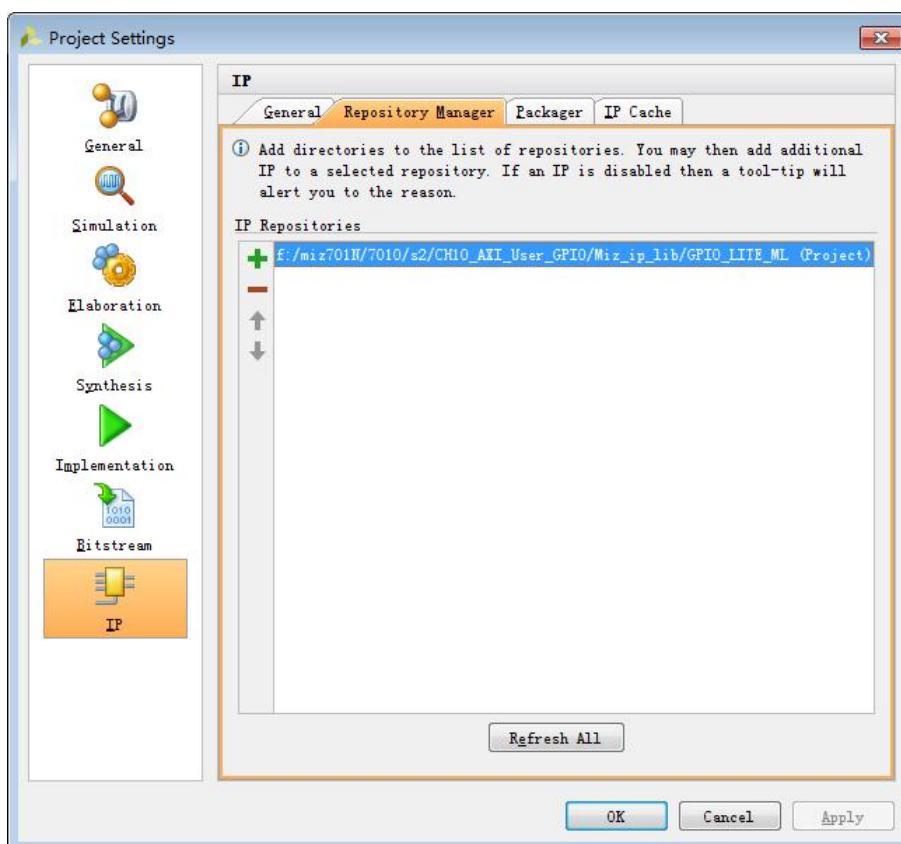
10.2 搭建硬件工程

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。

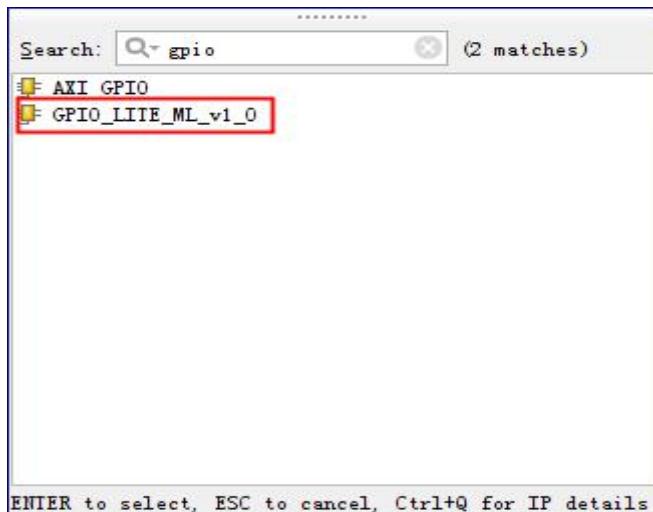


Step4: 单击+号图标，将上一节封装的 IP 的路劲存放进去，单击 OK。

Step5: 新建一个 BD 文件，输入文件名，完成创建。

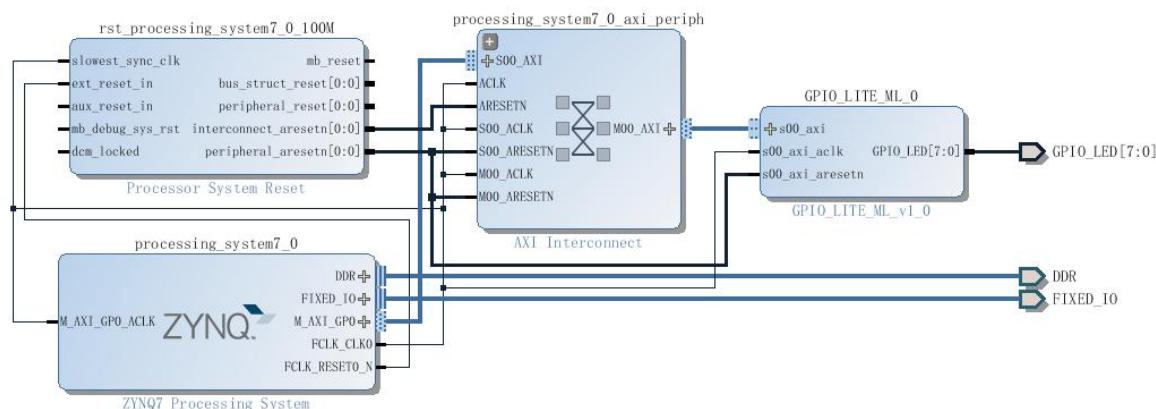
Step6: 向 BD 文件中添加一个 ZYNQ Processing system, 根据自身硬件完成 IP 的配置。

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: 添加一个约束文件，打开对应自己硬件的原理图，查看按键部分引脚连接情况，此次我们只用 4 个 LED 完成实验。Miz702 约束文件如下所示：

```
set_property SEVERITY {Warning} [get_drc_checks NSTD-1]
set_property SEVERITY {Warning} [get_drc_checks UCI0-1]
```

```
set_property PACKAGE_PIN T22 [get_ports {GPIO_LED[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {GPIO_LED[0]}]

set_property PACKAGE_PIN T21 [get_ports {GPIO_LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {GPIO_LED[1]}]

set_property PACKAGE_PIN U22 [get_ports {GPIO_LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {GPIO_LED[2]}]

set_property PACKAGE_PIN U21 [get_ports {GPIO_LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {GPIO_LED[3]}]
```

其他型号开发板参照对应型号的原理图的 LED 部分，修改成对应的引脚即可。

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，并添加一个 main.c 的文件。

Step3：在 main.c 文件中添加以下程序，按 Ctrl+S 保存后自动开始编译。

```
/*
 * main.c
 *
 * Created on: 2016 年 11 月 8 日
 * Author: Administrator
 */
#include <stdio.h>
#include "xparameters.h"
#include "xil_io.h"
#include "sleep.h"
#include "xil_types.h"

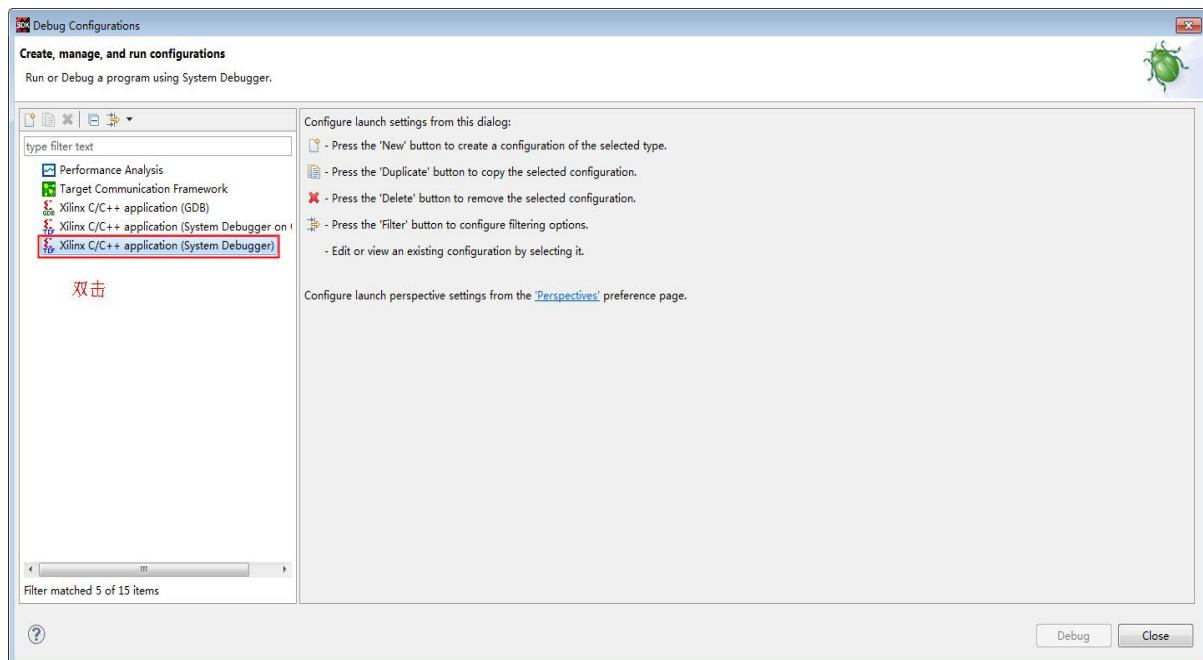
#define XGpio_axi_WriteReg(BaseAddr, RegOffset, Data) \
    Xil_Out32((BaseAddr) + (u32)(RegOffset), (u32)(Data))
```

```
#define XPAR_GPIO_LITE_ML_0 XPAR_GPIO_LITE_ML_0_BASEADDR
#define GPIO_LITE_ML_REG0 0

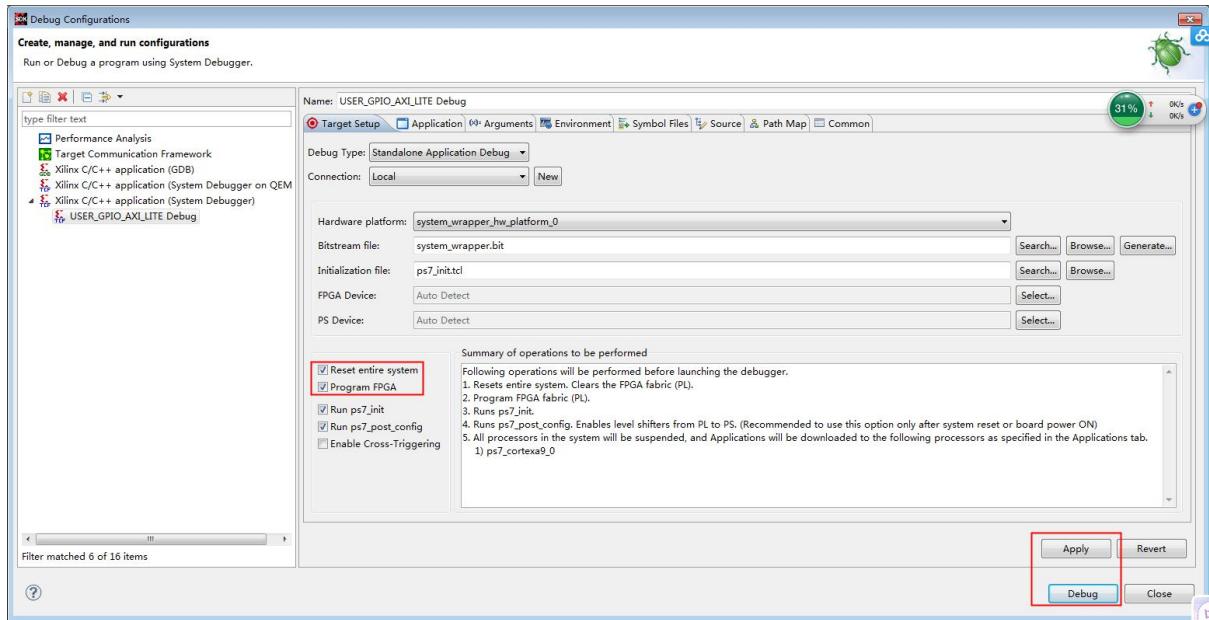
int main()
{
    u8 i=0;
    XGpio_axi_WriteReg(XPAR_GPIO_LITE_ML_0,GPIO_LITE_ML_REG0,0X00);
    while(1)
    {
        for(i=0;i<=3;i++)
        {
            XGpio_axi_WriteReg(XPAR_GPIO_LITE_ML_0,GPIO_LITE_ML_REG0,1<<i);
            usleep(500000);
        }
        i=0;
    }
}
```

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。



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基地址，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。

【第二季】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。

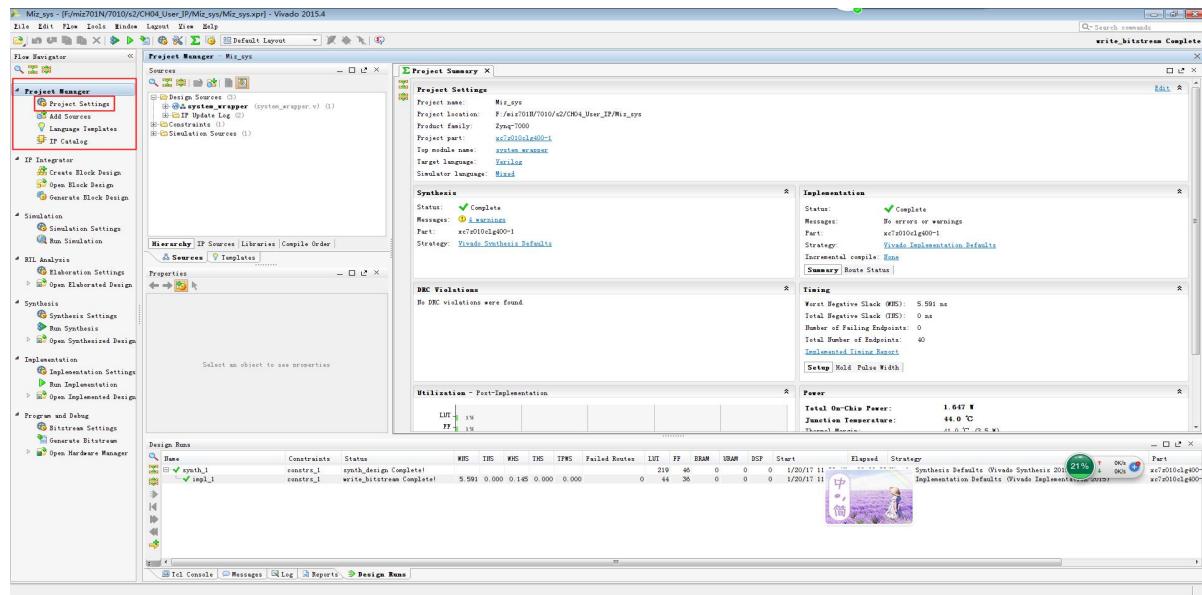
11.1 方案框架

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。

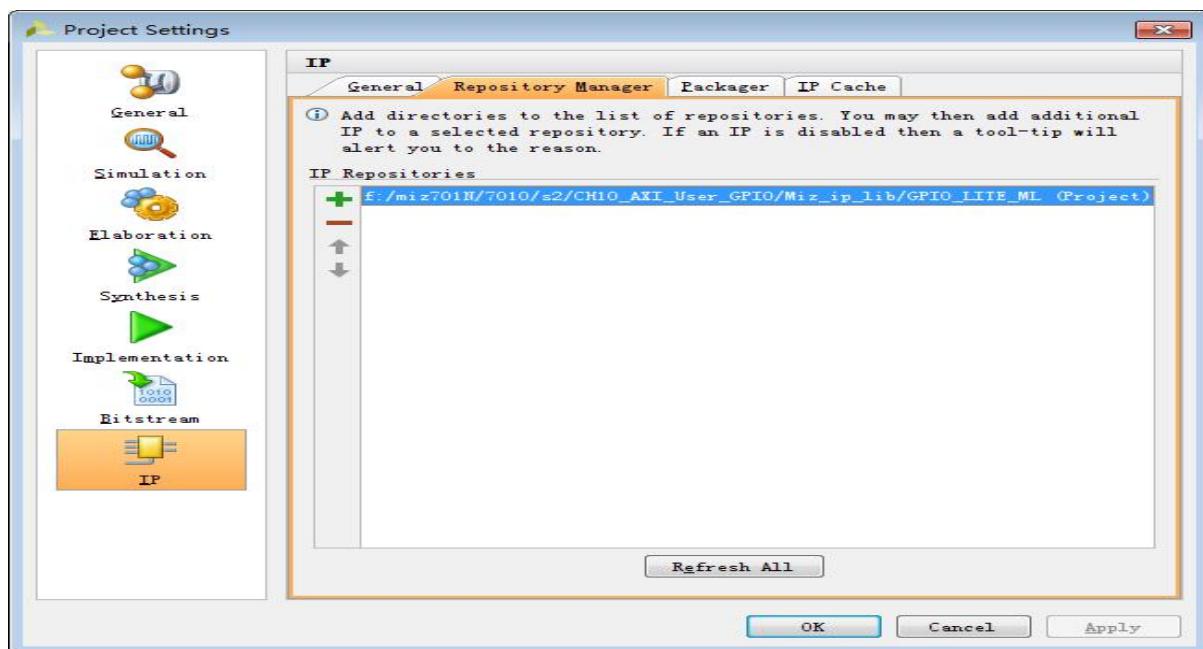
11.2 硬件工程搭建

Step1：做好备份后，直接打开上一章节的硬件工程。

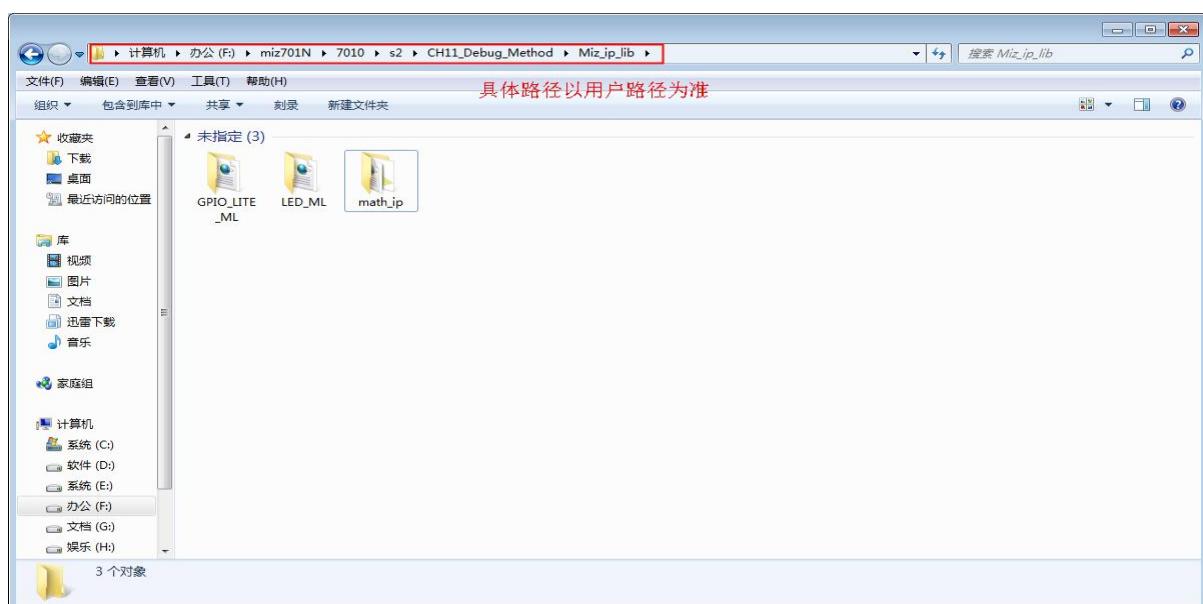
Step2：在 Project manager 区中单击 Project settings。



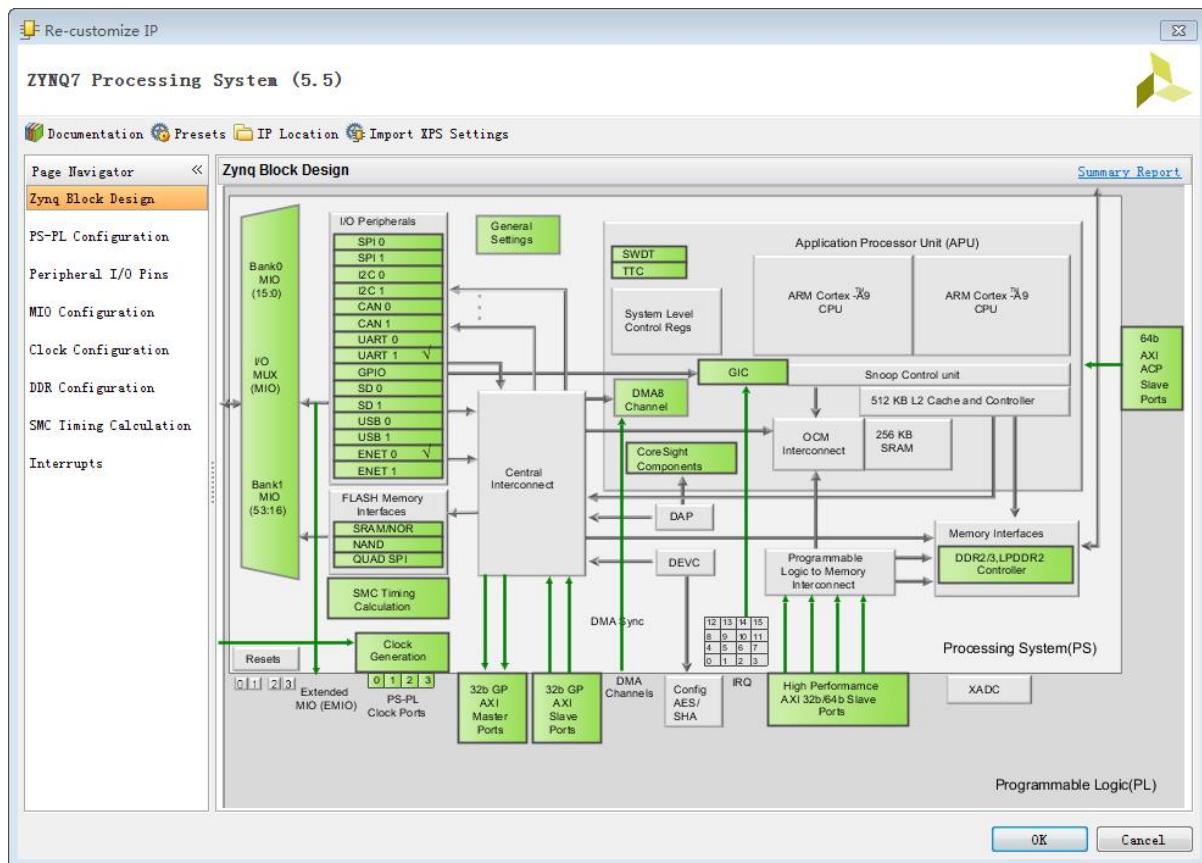
Step3：选择 IP 设置区中的 repository manager。



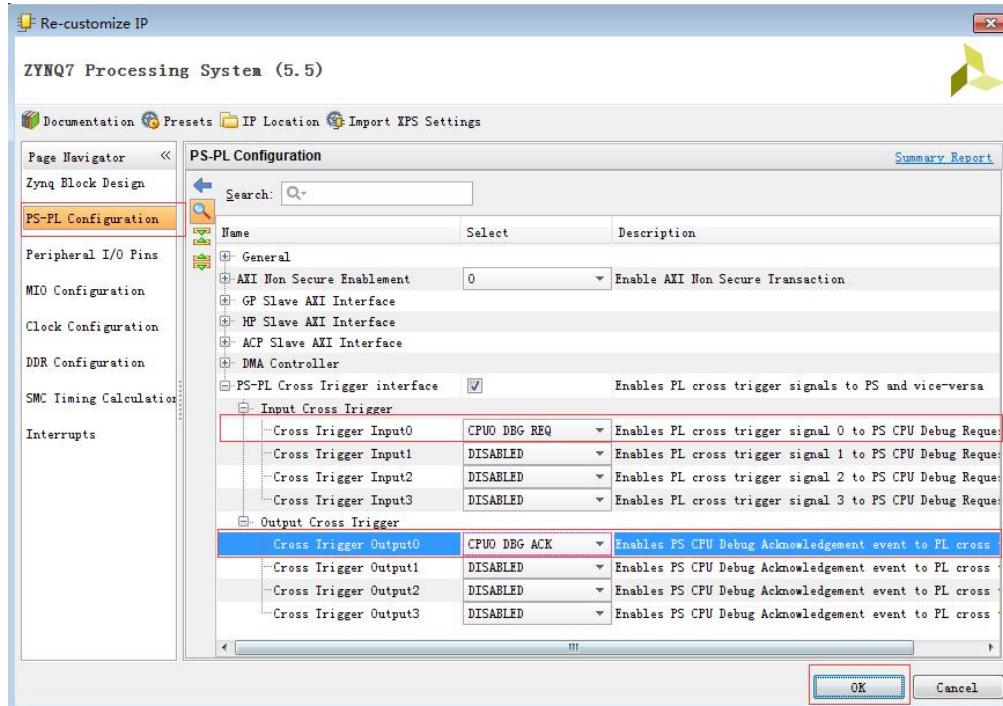
Step:4：单击+号图标，将 math_ip_0 的路径添加进去（math_ip 可在我们附带的第十一章程序文件夹中的 Miz_ip.lib 文件夹中找到），单击 OK。



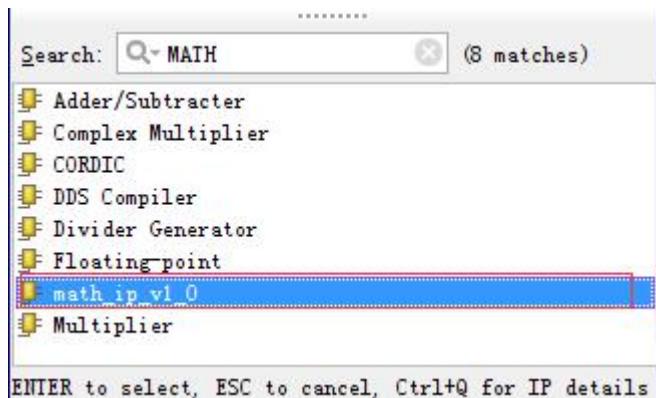
Step5：双击 ZYNQ processing System 图标，配置 IP。



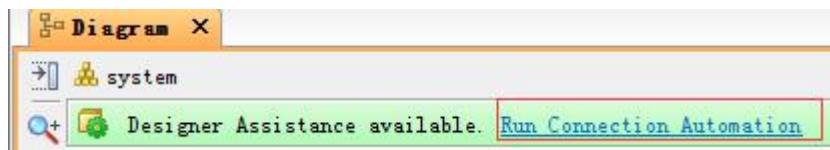
Step6: 展开 PS-PL Cross Trigger interface > Input Cross Trigger, Cross Trigger Input 0 设置为: CPU0 DBG REQ、Output Cross Trigger 设置为 CPU0 DBG ACK, 单击 OK 完成修改。



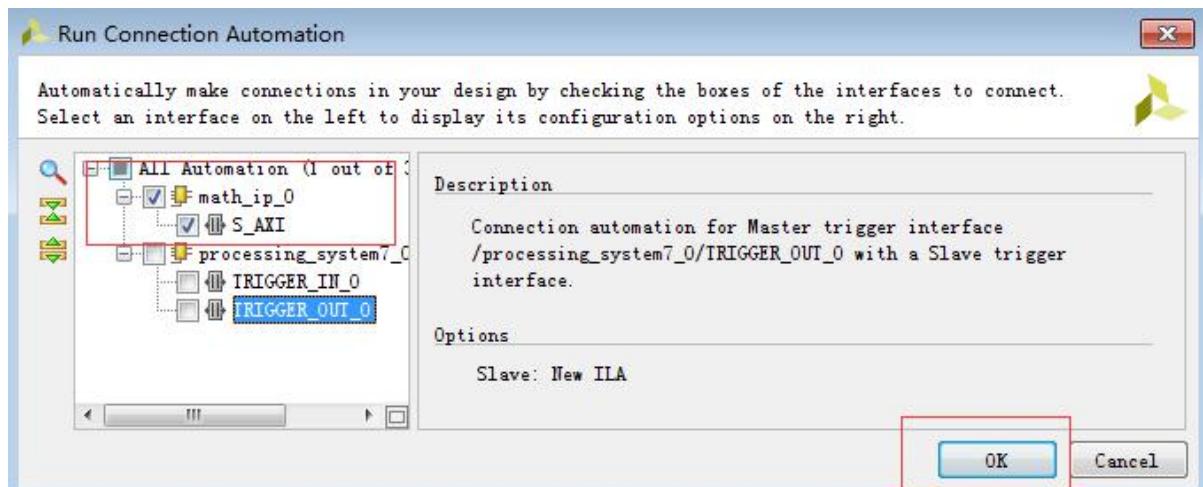
Step7: 单击 IP icon  搜索单词“math”之后双击添加 IPCORE。



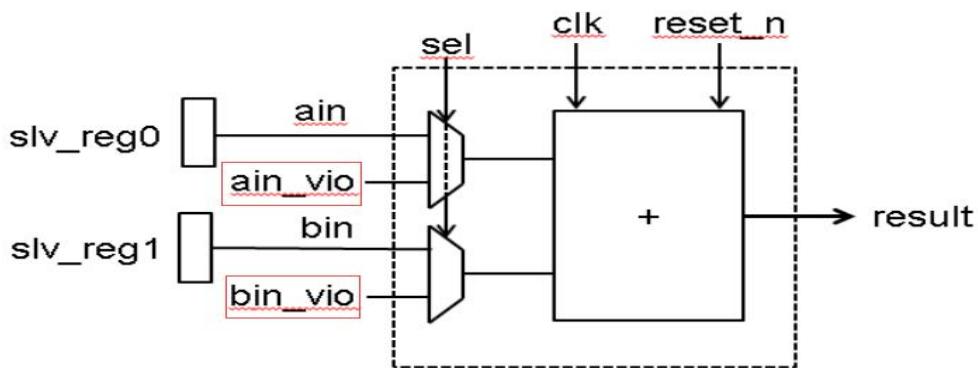
Step8: 单击 Click on Run Connection Automation。



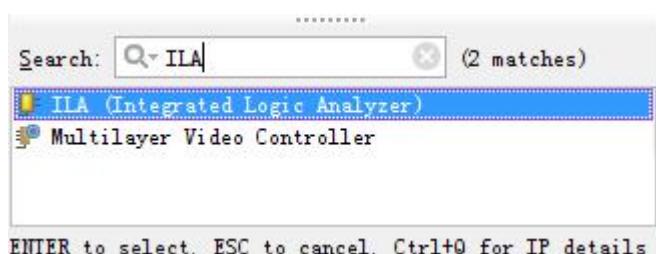
Step9: 勾选 math_ip_0 and S_AXI 之后单击 OK。



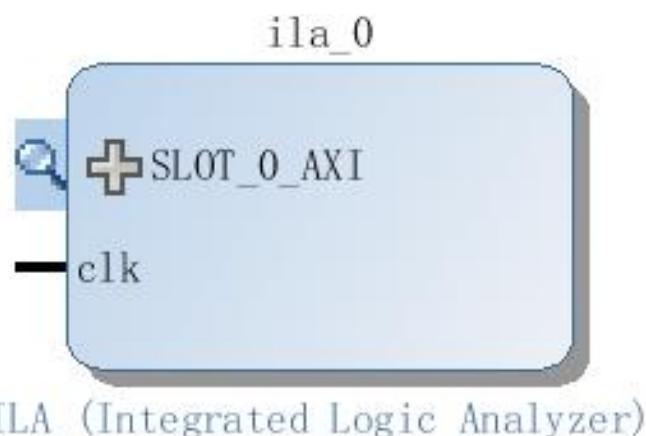
这个 math_ip 实际上是一个简单的硬件加法器。虽然这个简单的加法器在这里没有实用意义,但是如果换成了硬件算法,那么就具备实用价值了。红色的方框内 ain_vio 和 bin_vio 是我们准备通过逻辑分析抓去的观察信号。



Step10: 单击 IP icon 添加 ila CORE

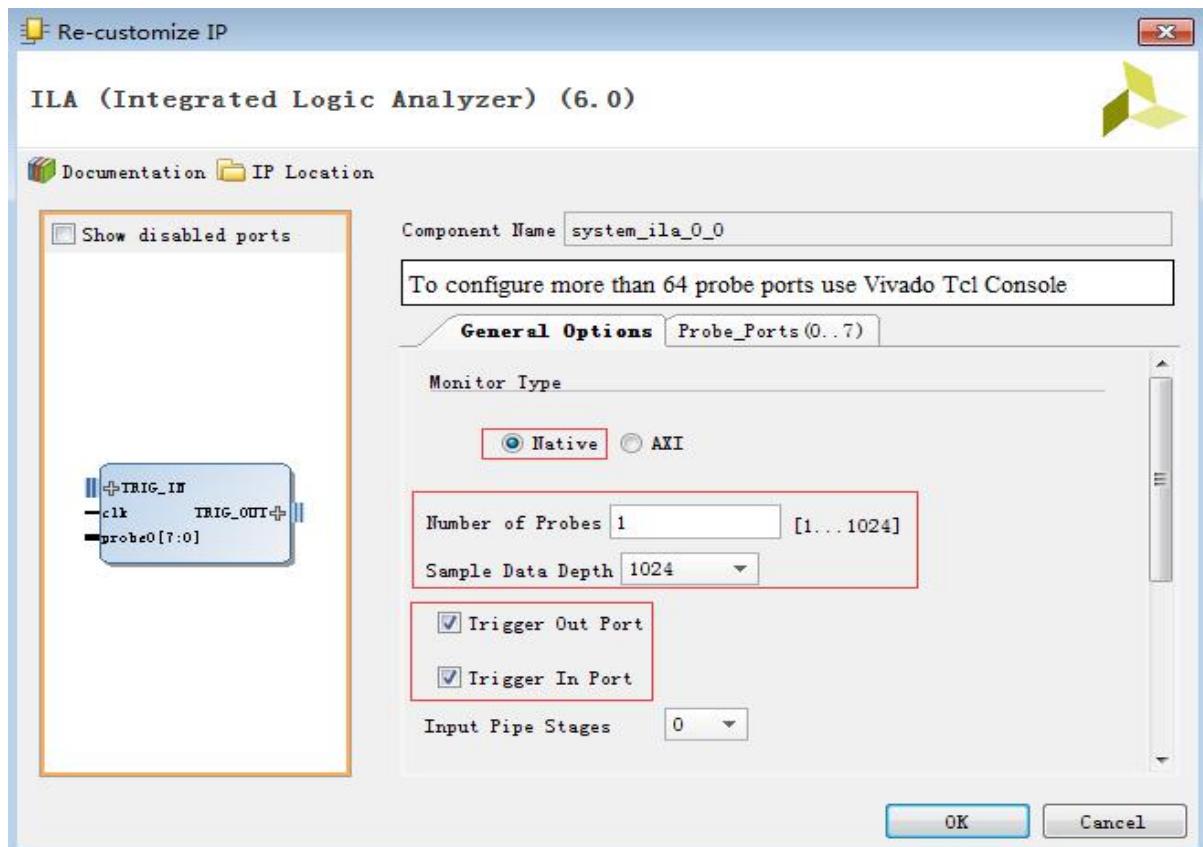


Step11: 双击打开 ILA CORE

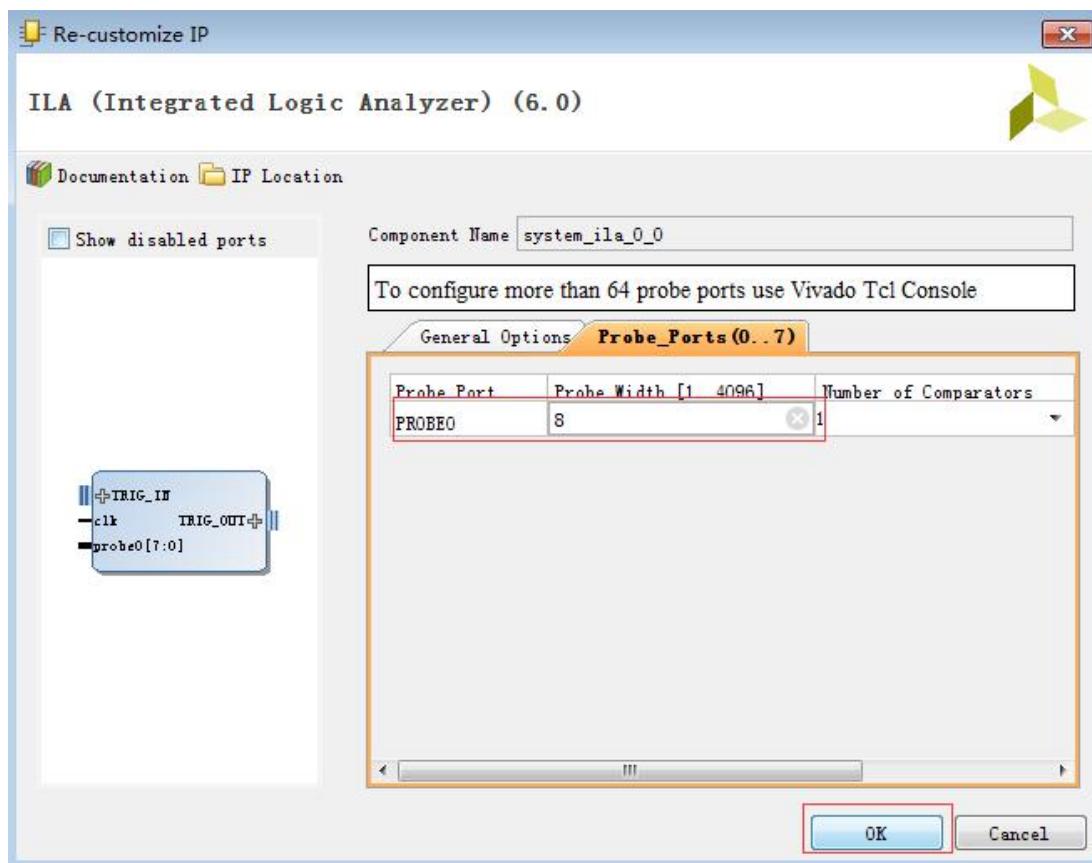


Step12: 双击打开 ILA CORE

General Options 设置如下



Probe Ports 设置如下,之后单击 OK

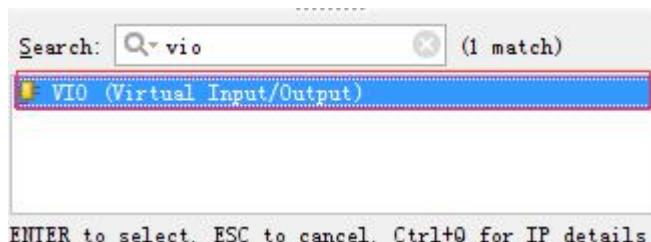


Step13: 连接 Probe0 到 GPIO_LED。

Step14: 连接 CLK 接口到 FCLK_CLK0 接口

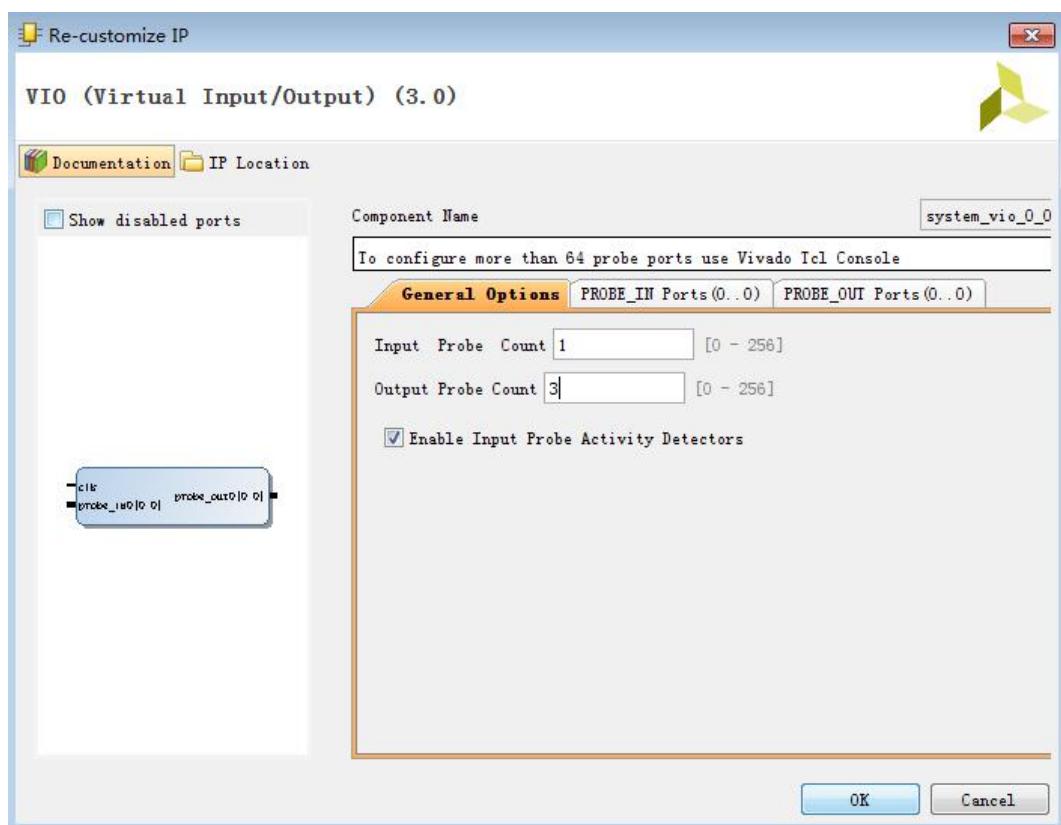
Step15: 连接 TRIGG_IN 和 TRIGGER_OUT_0、TRIG_OUT 和 TRIGGER_IN_0

Step16: 添加 IP icon  添加 vio。

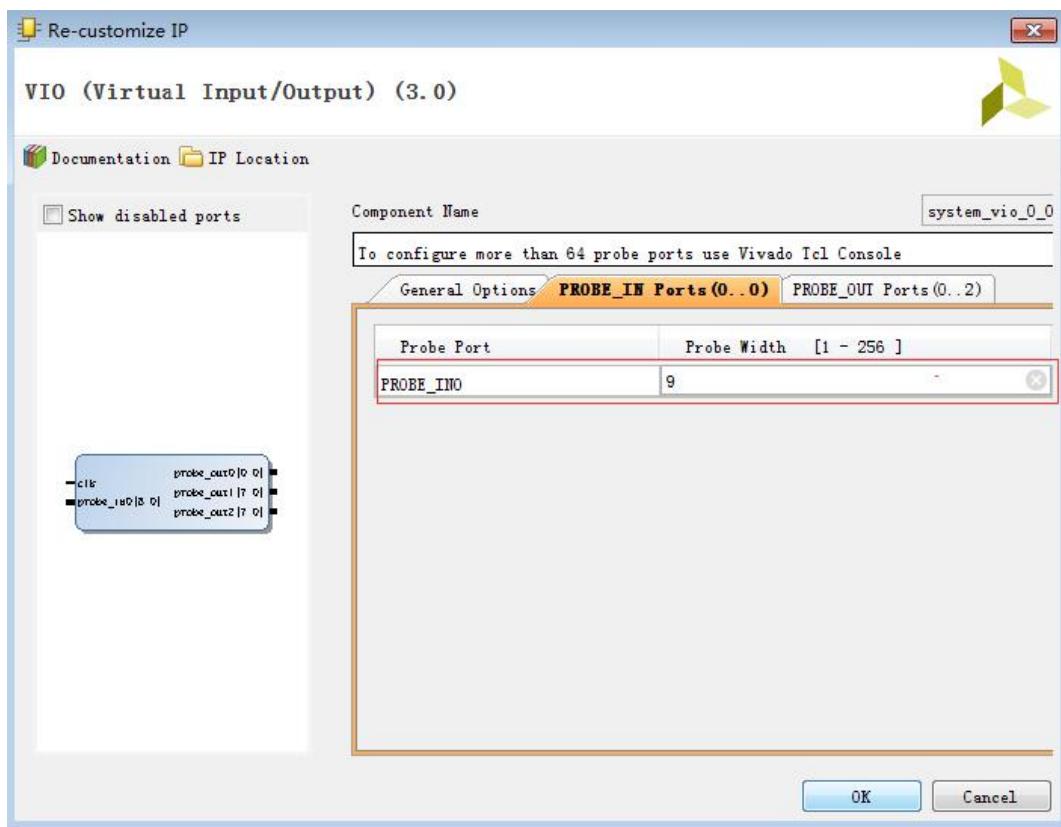


Step17: 双击 VIO core 修改参数

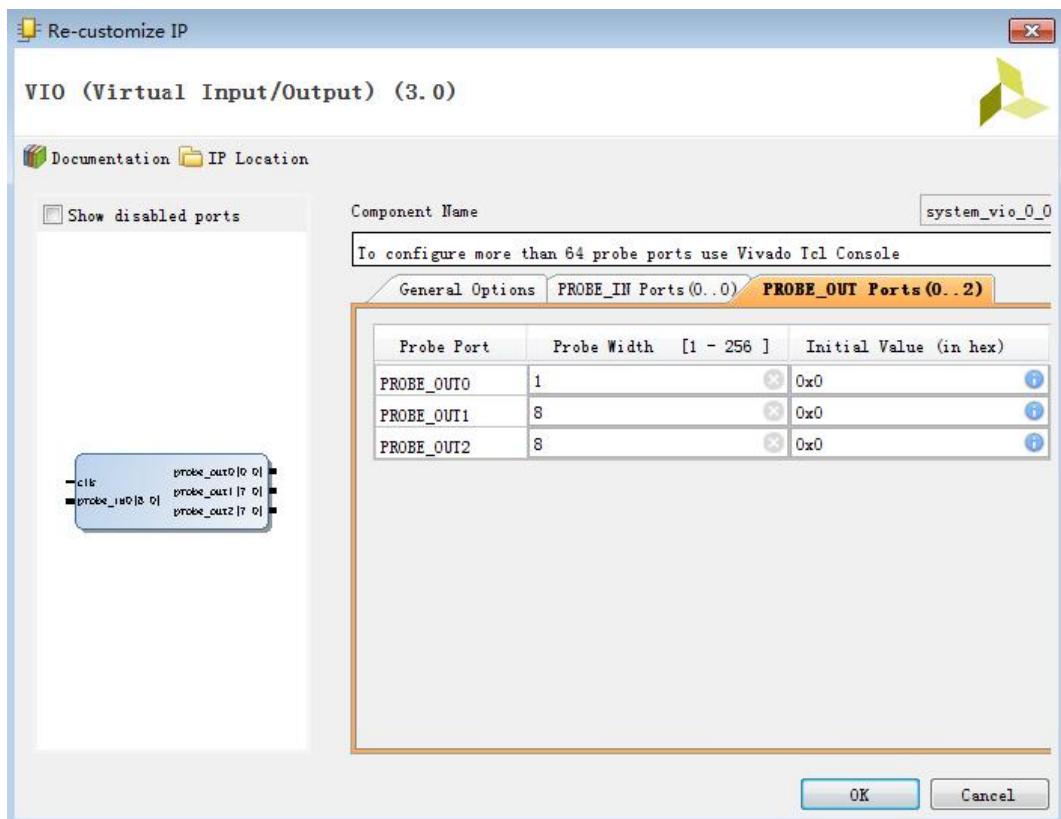
General Options 设置如下，输入 probe 为 1 输出为 3



Probe_in 设置位宽为 9

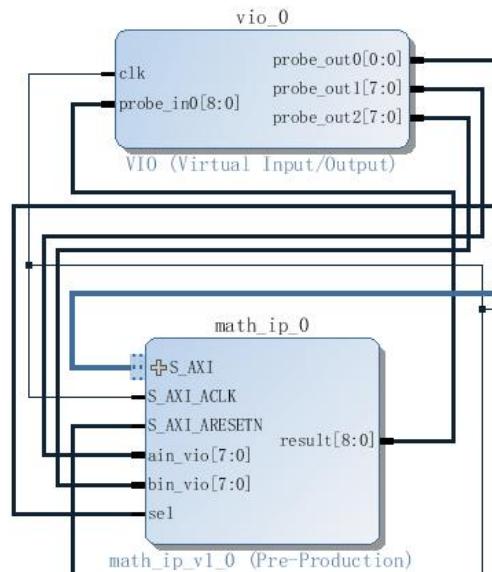


Probe_out0 设置位宽: 1; Probe_out1 设置位宽: 8; Probe_out2 设置位宽: 8;

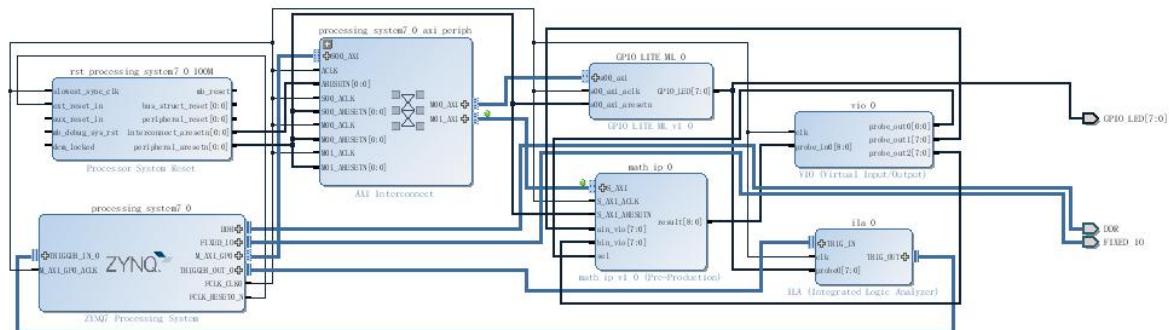


Step18:连接

PROBE_IN -> result
 PROBE_OUT0 -> sel
 PROBE_OUT1 -> ain_vio
 PROBE_OUT2 -> bin_vio
 CLK-> FCLK_CKL0

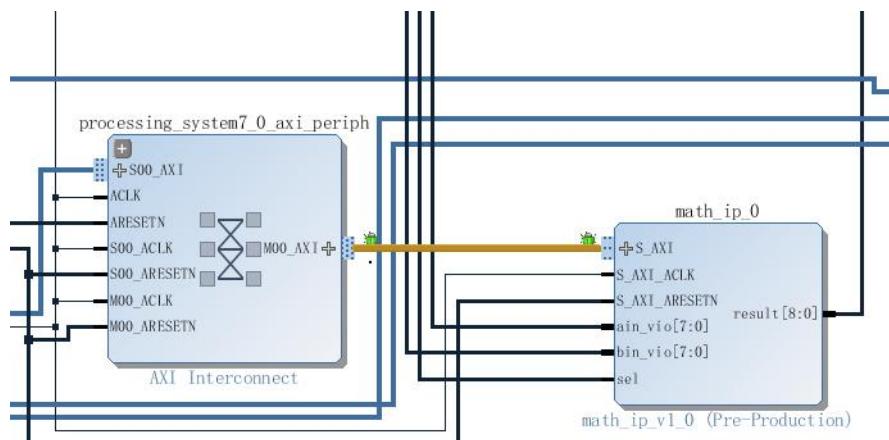


Step19: 连接好的系统整体电路。



Step20: 选中 AXI Interconnect 和 math_0 CORE 之间的 S_AXI 总线

Step21: 右击选择 Mark Debug



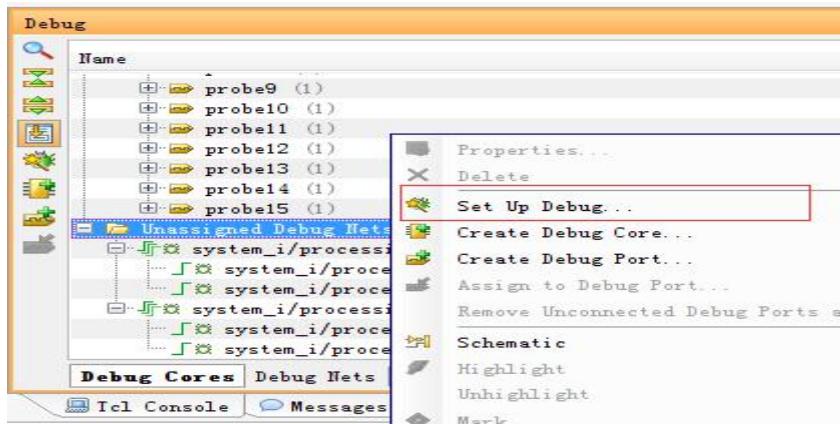
Step22:接下来依然是,右键单击 Block 文件,文件选择 Generate the Output Products。

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。

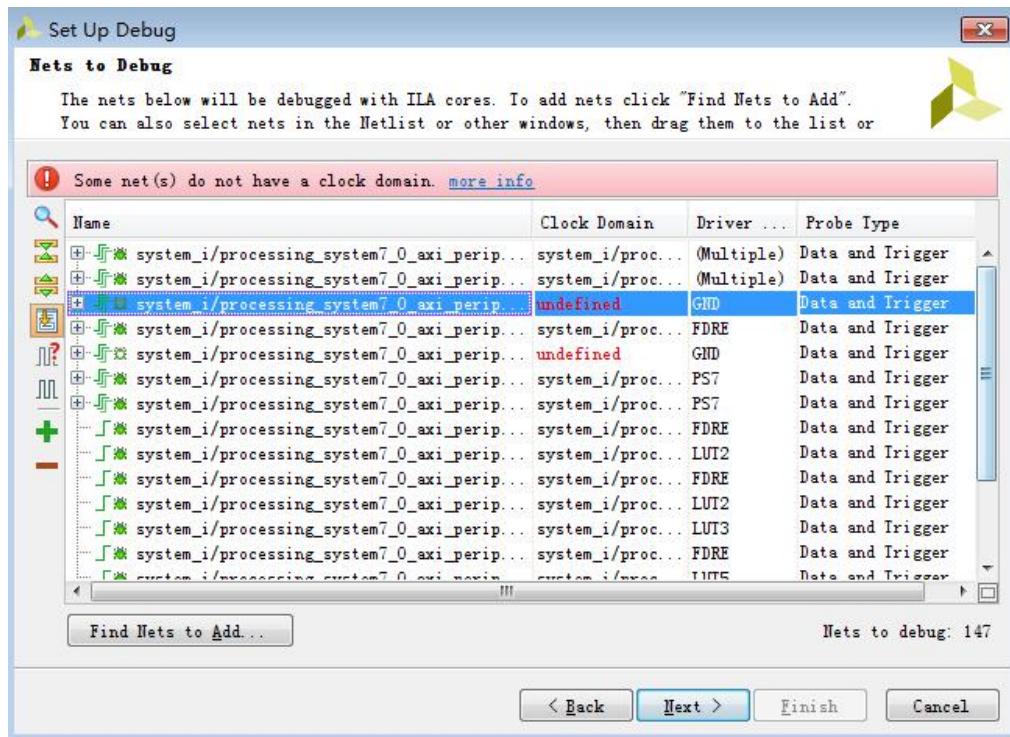
Step25:综合结束后选择 Synthesized Design option 单击 OK。

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，并添加一个 main.c 的文件。

Step3: 在 main.c 文件中添加以下程序，按 Ctrl+S 保存后自动开始编译。

```
/*
 * main.c
 *
 * Created on: 2016 年 11 月 8 日
 * Author: Administrator
 */

#include <stdio.h>
#include "xparameters.h"
#include "xil_io.h"
#include "sleep.h"
#include "xil_types.h"

#define XGpio_axi_WriteReg(BaseAddr, RegOffset, Data) \
```

```
Xil_Out32((BaseAddr) + (u32)(RegOffset), (u32)(Data))

#define XPAR_GPIO_LITE_ML_0 XPAR_GPIO_LITE_ML_0_BASEADDR
#define GPIO_LITE_ML_REG0 0

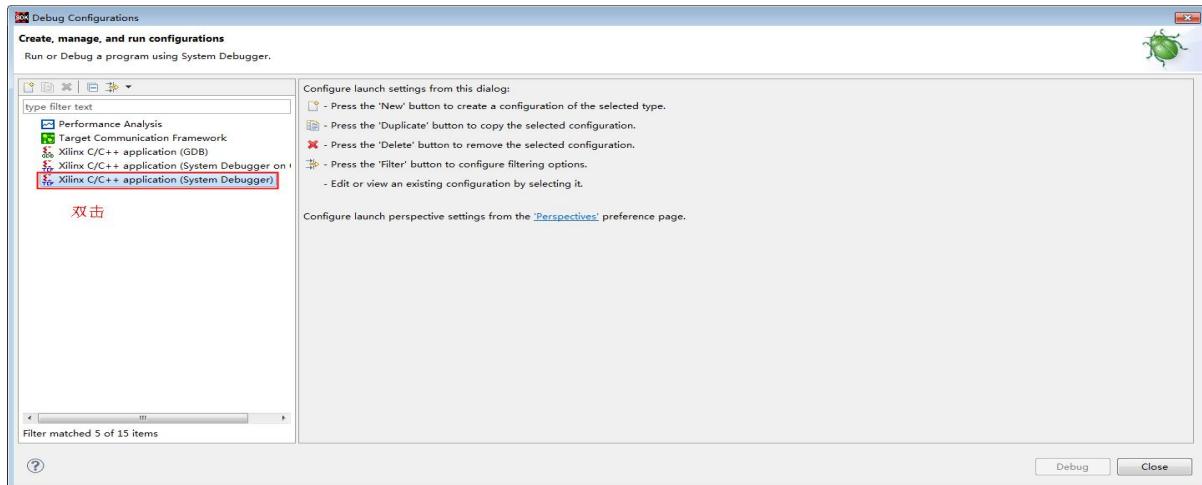
#define MATH_IP_BASE XPAR_MATH_IP_0_BASEADDR
#define MATH_REG0 0
#define MATH_REG1 4
#define MATH_REG2 0

int main()
{
    u8 i=0;
    u8 val=0;
    Xil_Out32(MATH_IP_BASE+MATH_REG0,0X42);
    Xil_Out32(MATH_IP_BASE+MATH_REG1,0X12);
    val = Xil_In32(MATH_IP_BASE+MATH_REG2);
    xil_printf("val=%x",val);

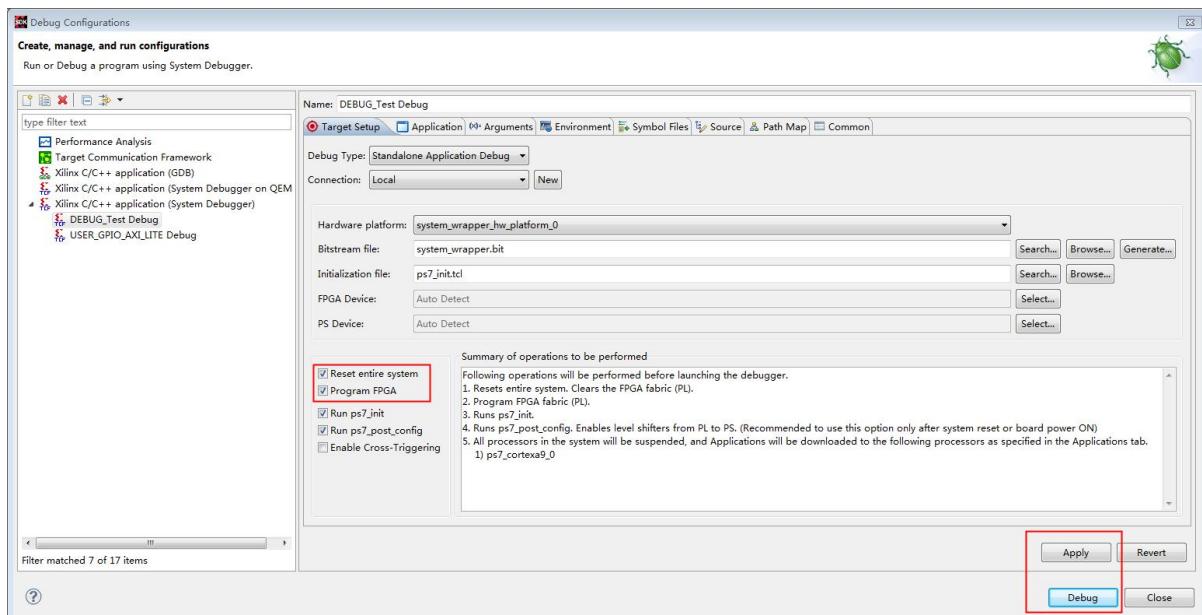
    XGpio_axi_WriteReg(XPAR_GPIO_LITE_ML_0,GPIO_LITE_ML_REG0,0X00);
    while(1)
    {
        for(i=0;i<=3;i++)
        {
            XGpio_axi_WriteReg(XPAR_GPIO_LITE_ML_0,GPIO_LITE_ML_REG0,1<<i);
            usleep(500000);
        }
        i=0;
    }
}
```

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

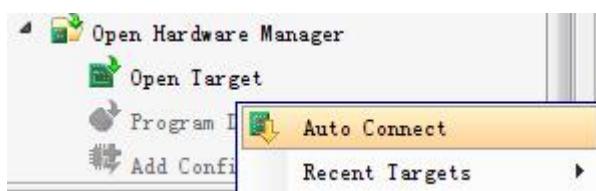
Step5：选中 system Debugger,双击创建一个系统调试。



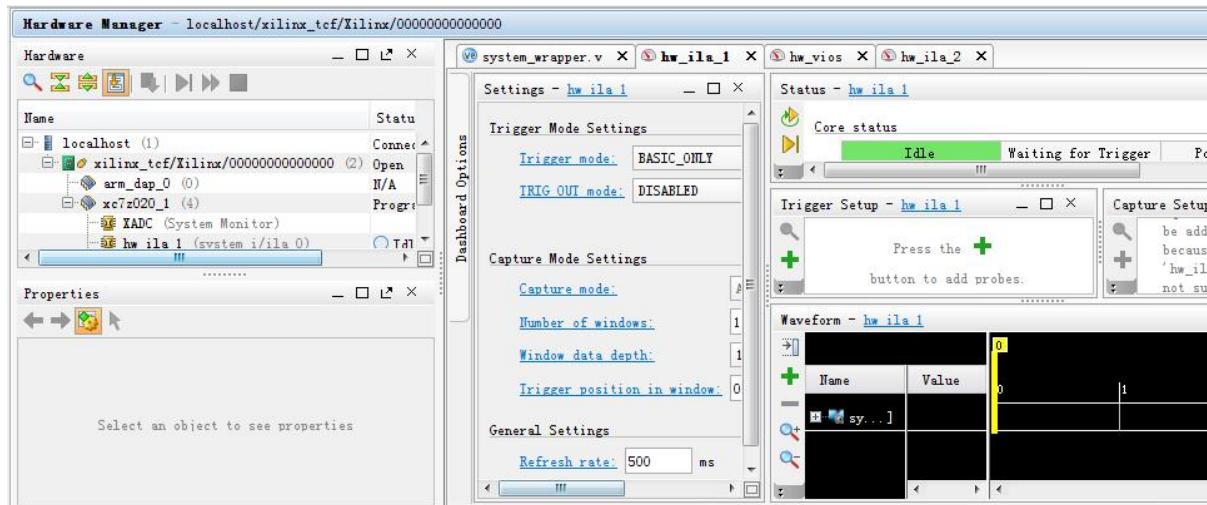
Step6: 设置系统调试。



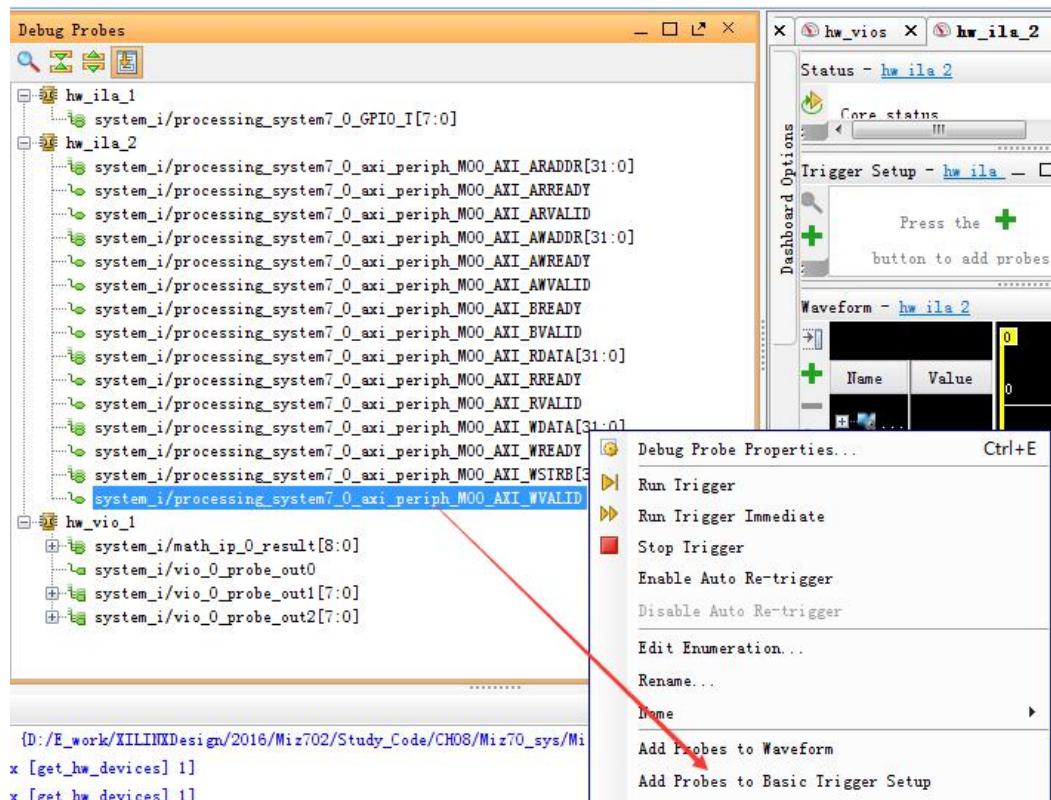
Step7: 回到 VIVADO 单击 Open Target->Auto Connect



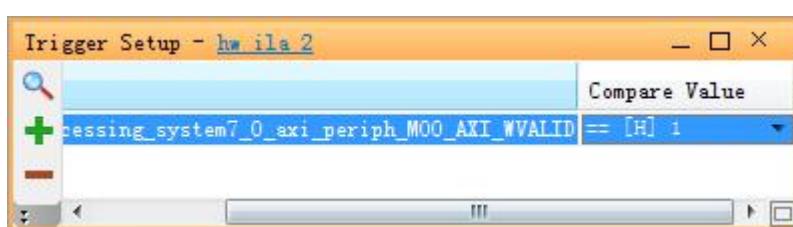
Step8: 加载完成后的界面



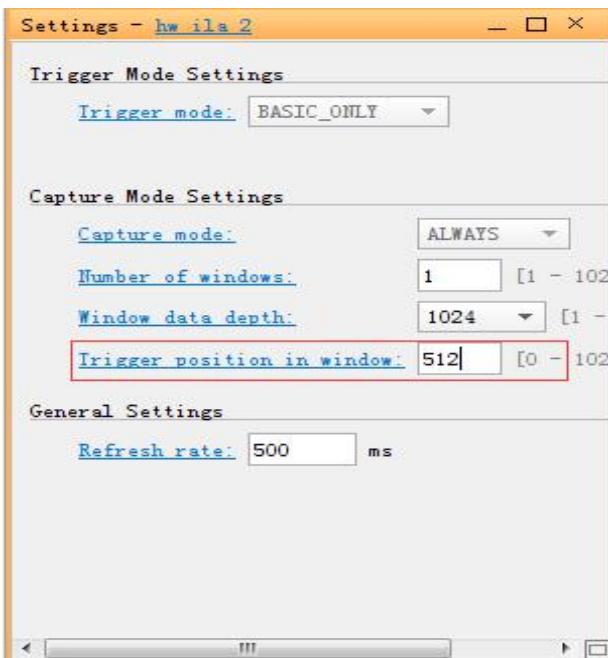
Step9: 选择菜单->window->Debugprobes 选择 AXI_WVALID 做为触发信号



Step10: 设置触发条件为 1



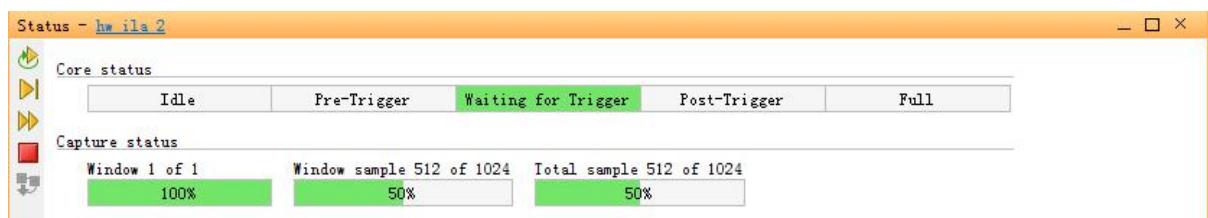
Step11: 设置触发位置为 512



Step12: 单击箭头所指向启动触发



Step13: 进入等待触发状态



Step14: 打开系统自带的串口调试软件。

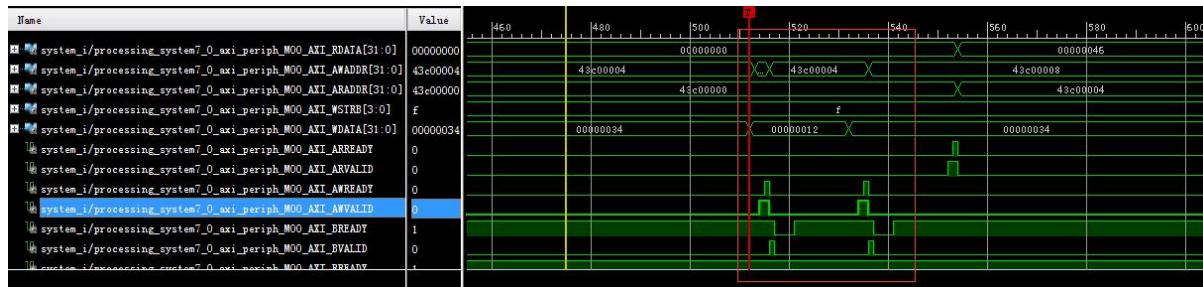
Step15: 在以下位置加入断点（在图中位置双击即可加入断点），方便调试。

```
#define MATH_REG0 0
#define MATH_REG1 4
#define MATH_REG2 8

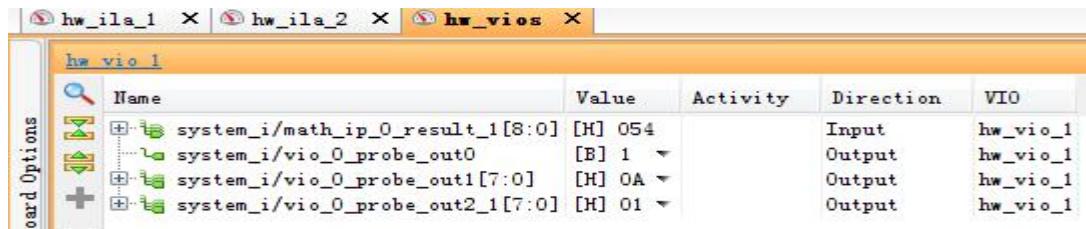
int main()
{
    u8 i=0;
    u8 val=0;
    Xil_Out32(MATH_IP_BASE+MATH_REG0,0x42);
    Xil_Out32(MATH_IP_BASE+MATH_REG1,0x12);
    val = Xil_In32(MATH_IP_BASE+MATH_REG2);
    Xil_Printf("val=%u",val);

    XGpio_axi_WriteReg(XPAR_GPIO_LITE_ML_0,GPIO_LITE_ML_REG0,0x00);
    while(1)
    {
        for(i=0;i<=3;i++)
        {
            XGpio_axi_WriteReg(XPAR_GPIO_LITE_ML_0,GPIO_LITE_ML_REG0,1<<i);
            usleep(500000);
        }
        i=0;
    }
}
```

Step16: 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。

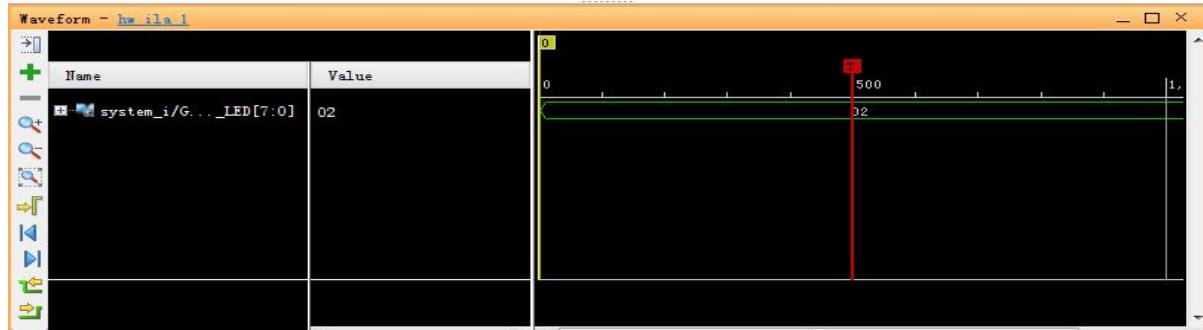


Step17: 同时可以观察到 VIO 核采集到的数据



Step18: 当再次单击 后控制台输出 0X54

Step19: 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。

S02_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

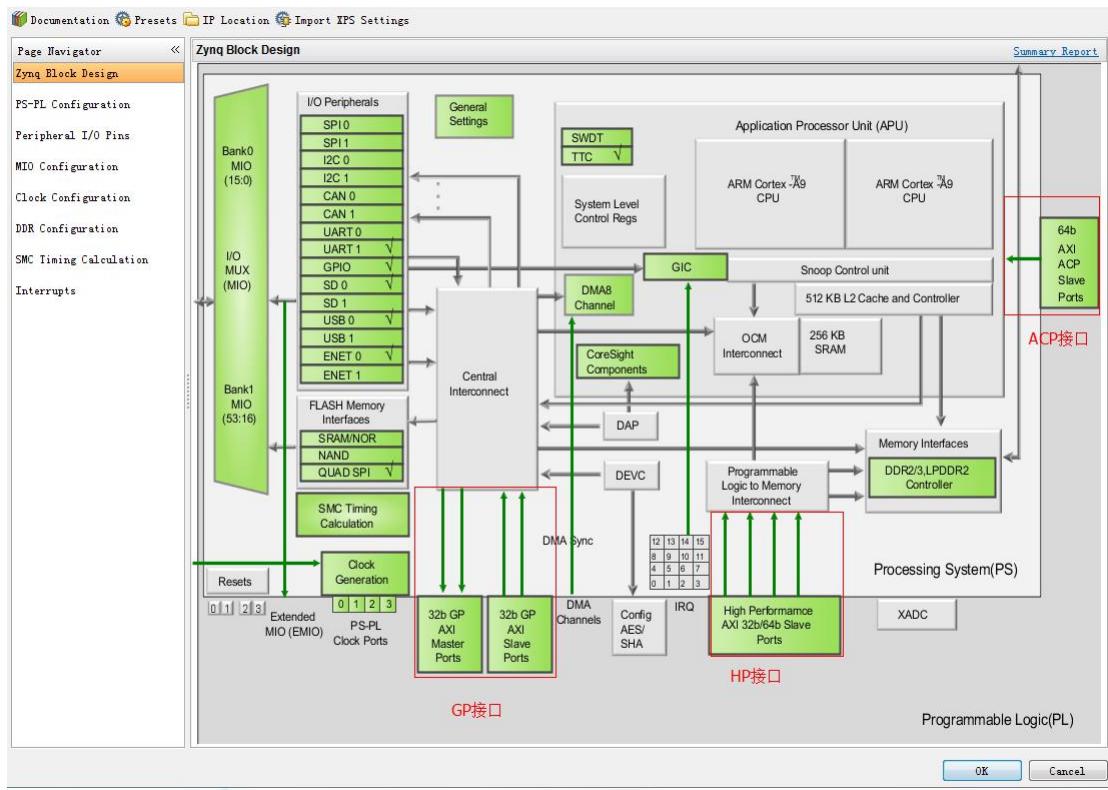
三种AXI接口分别是：

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：

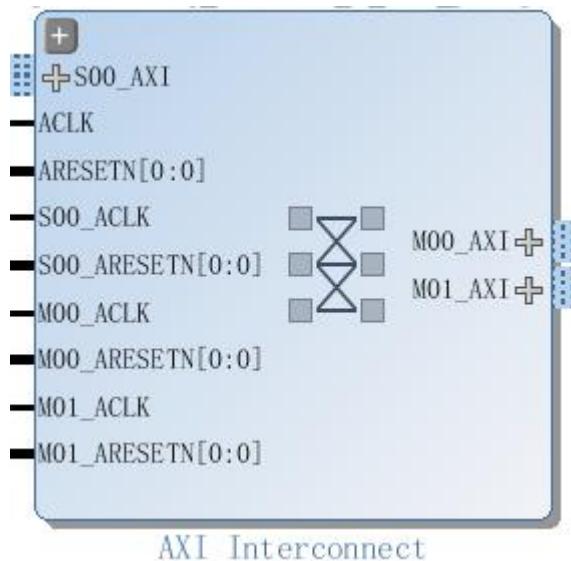


12.3.3 AXI 协议概述

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：

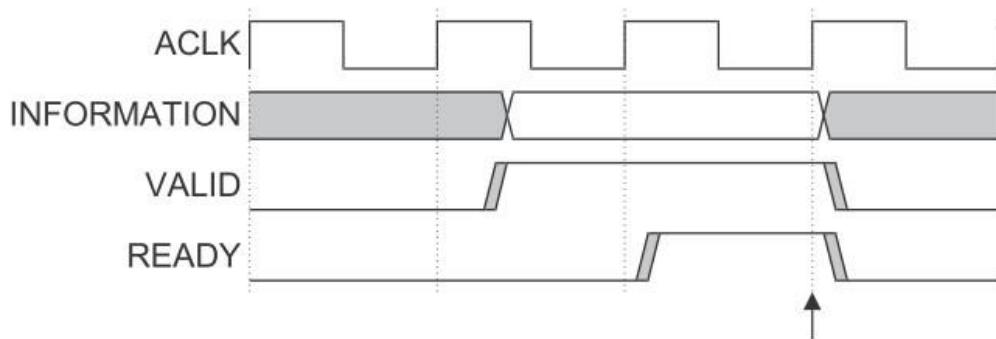
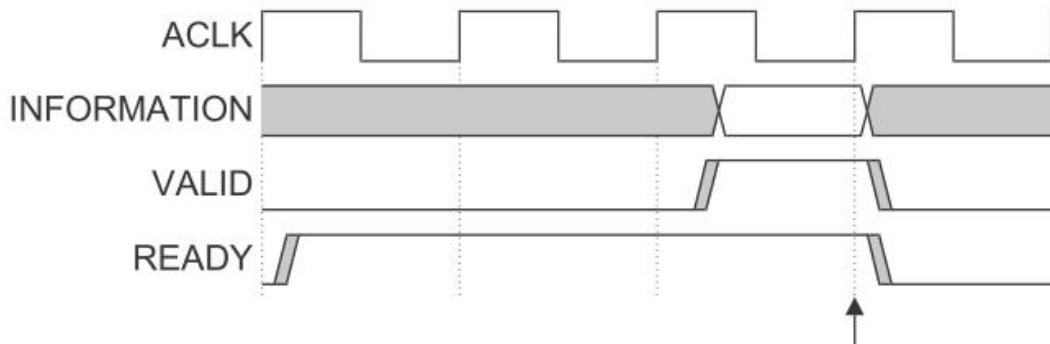


Figure 3-1 VALID before READY handshake

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：

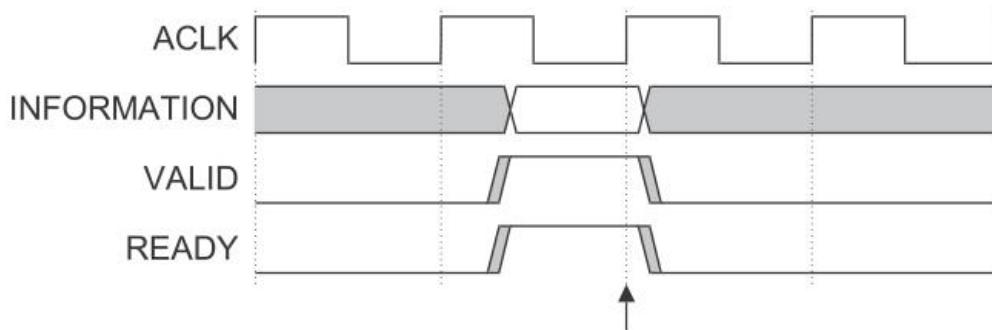


Figure 3-3 VALID with READY handshake

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：

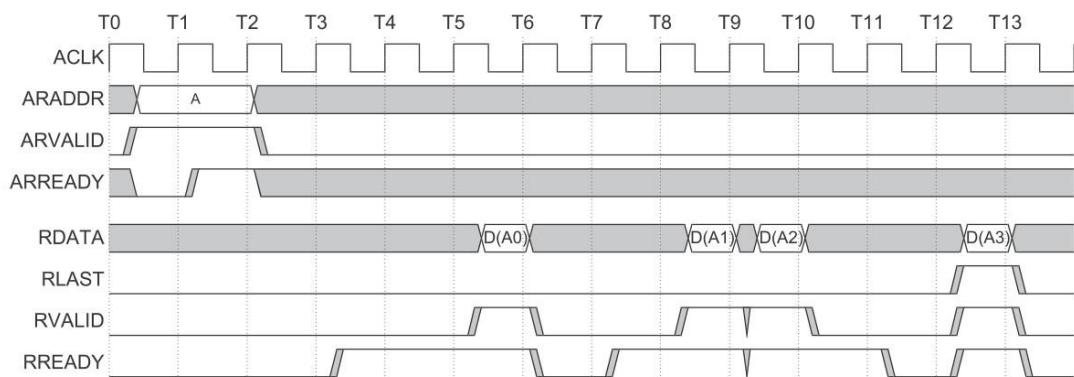


Figure 1-4 Read burst

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：

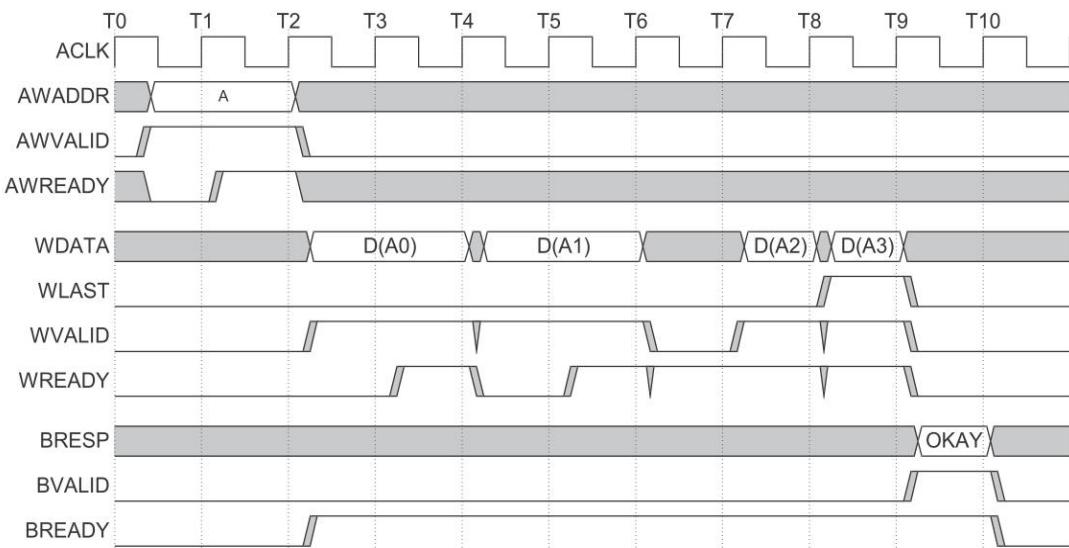


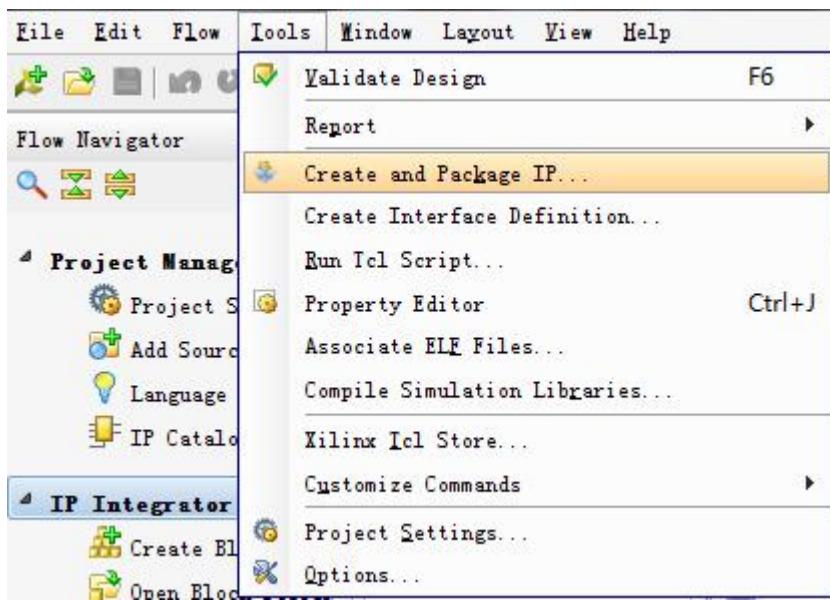
Figure 1-6 Write burst

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。

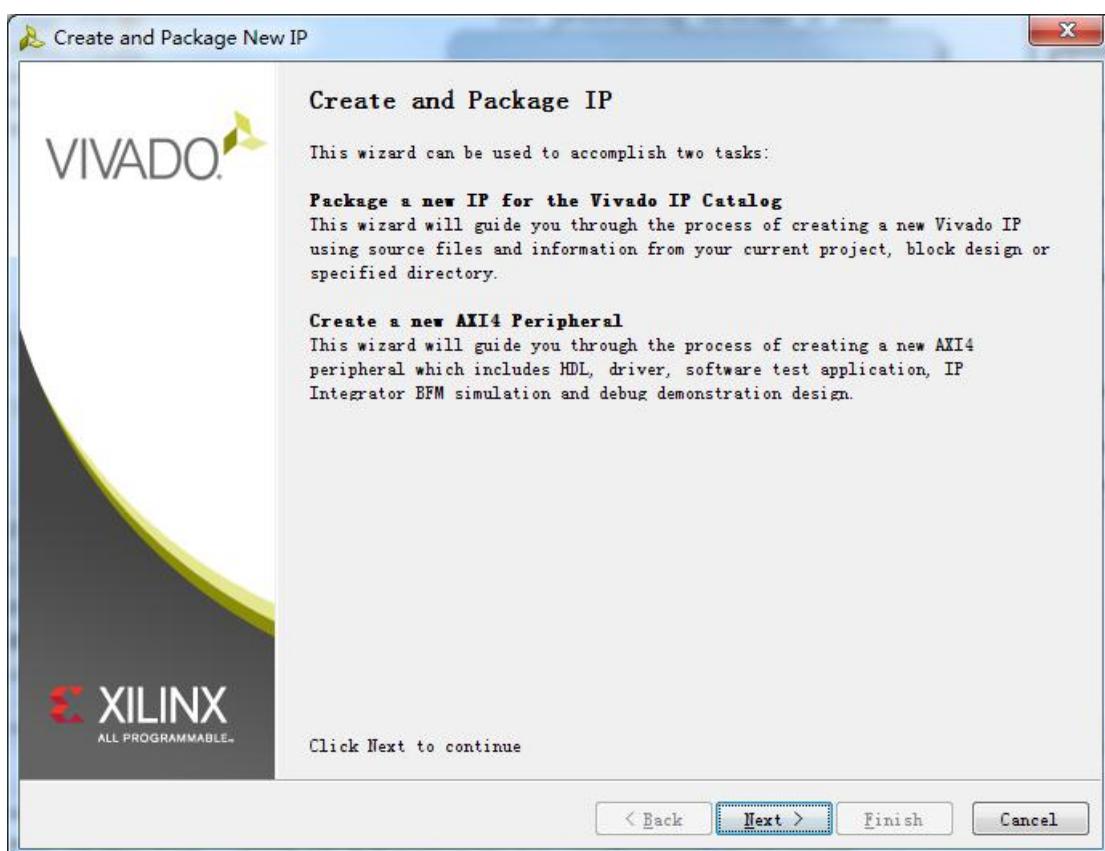
12.4 AXI4-Lite 详解

12.4.1 AXI4-Lite 源码查看

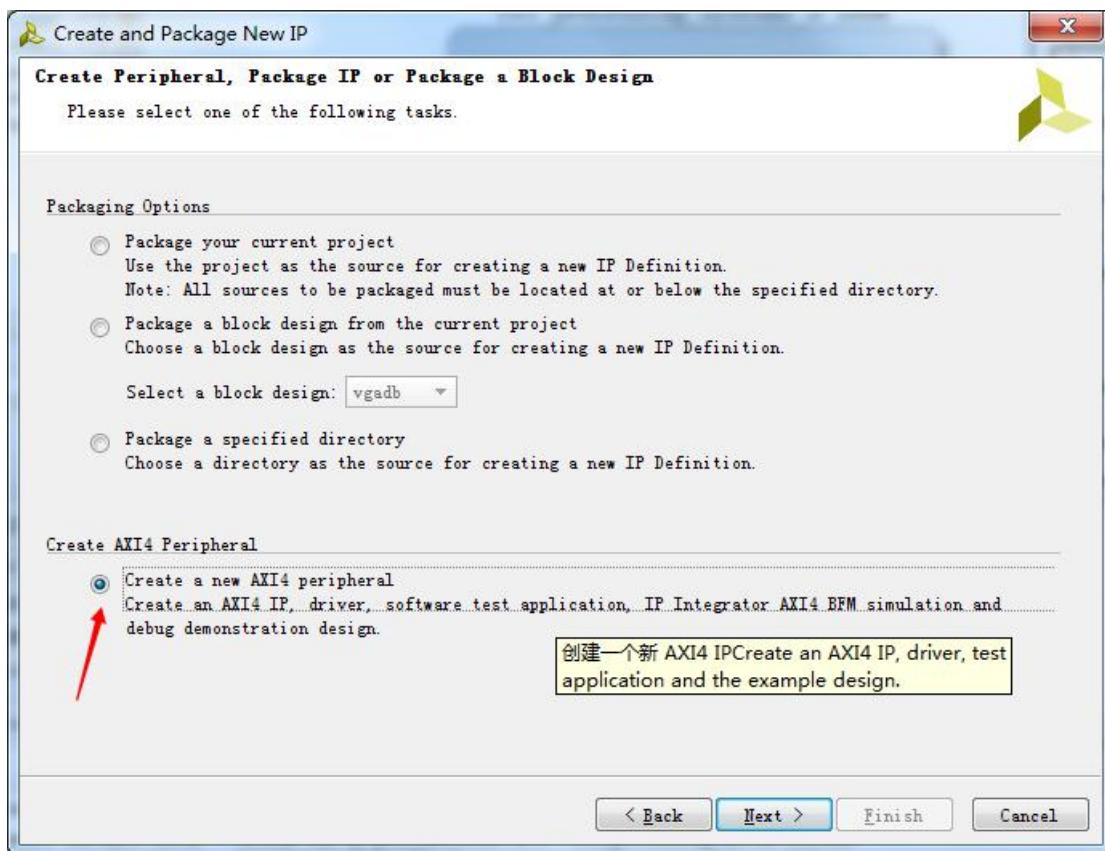
Step1：要看到AXI-Lite的源码，我们先要自定义一个AXI-Lite的IP，新建工程之后，选择，菜单栏->Tools->Create and Package IP：



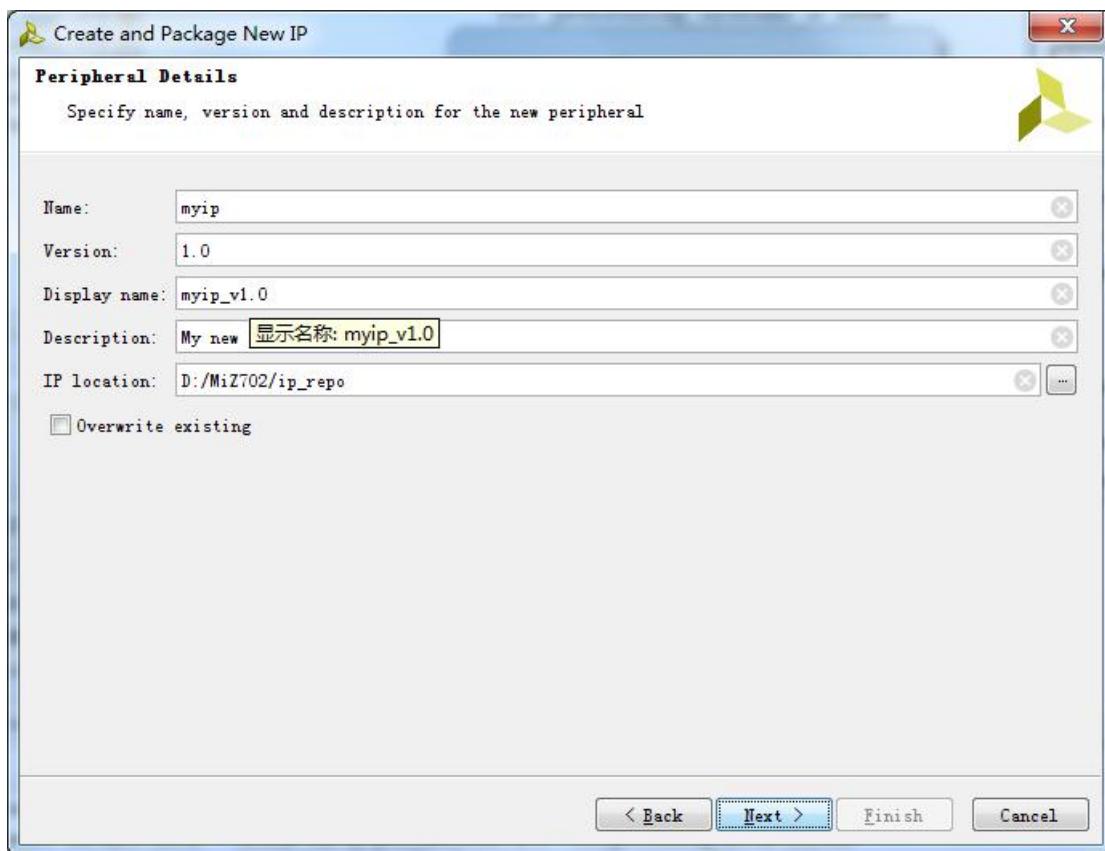
Step2: 选择Next



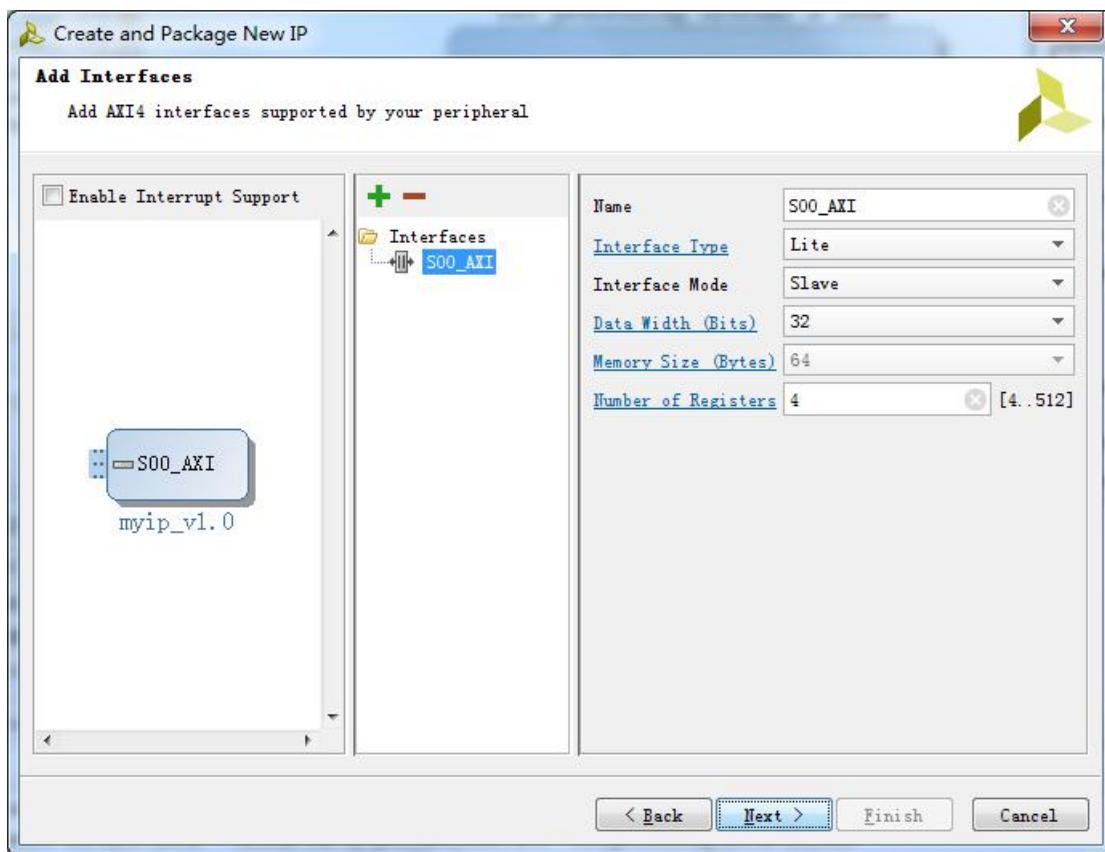
Step3: 选择Create AXI4 Peripheral, 然后Next:



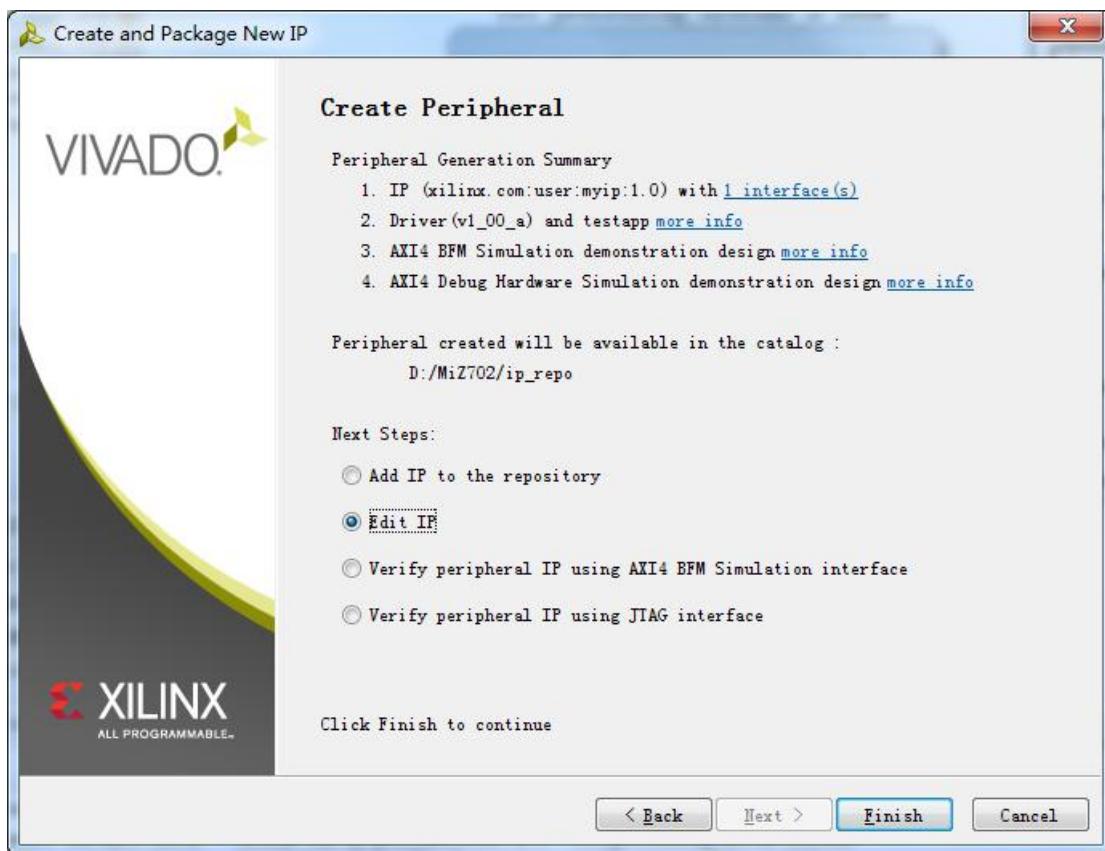
Step4: 默认，选择Next



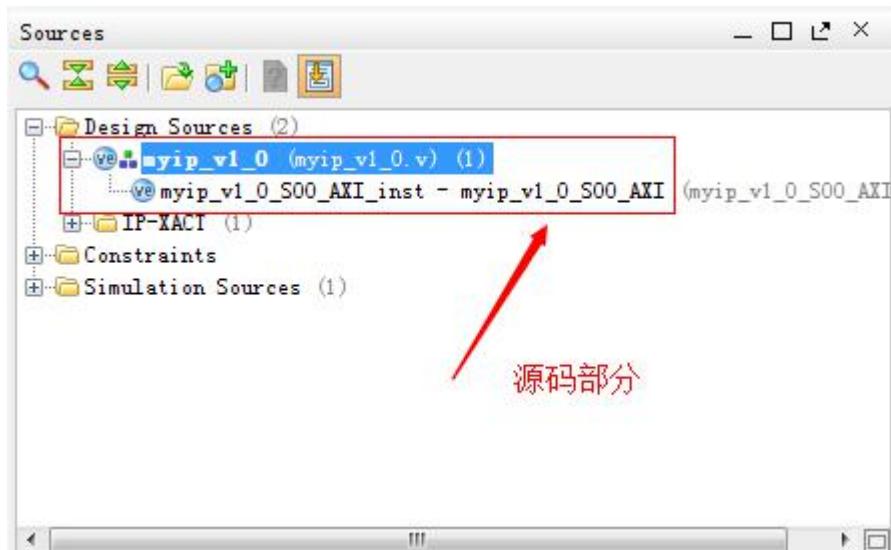
Step5: 注意这里接口类型选择Lite，选择Next:



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
      end
    end
  end
end
```

```

    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底寄存器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_ARVALID == 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_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

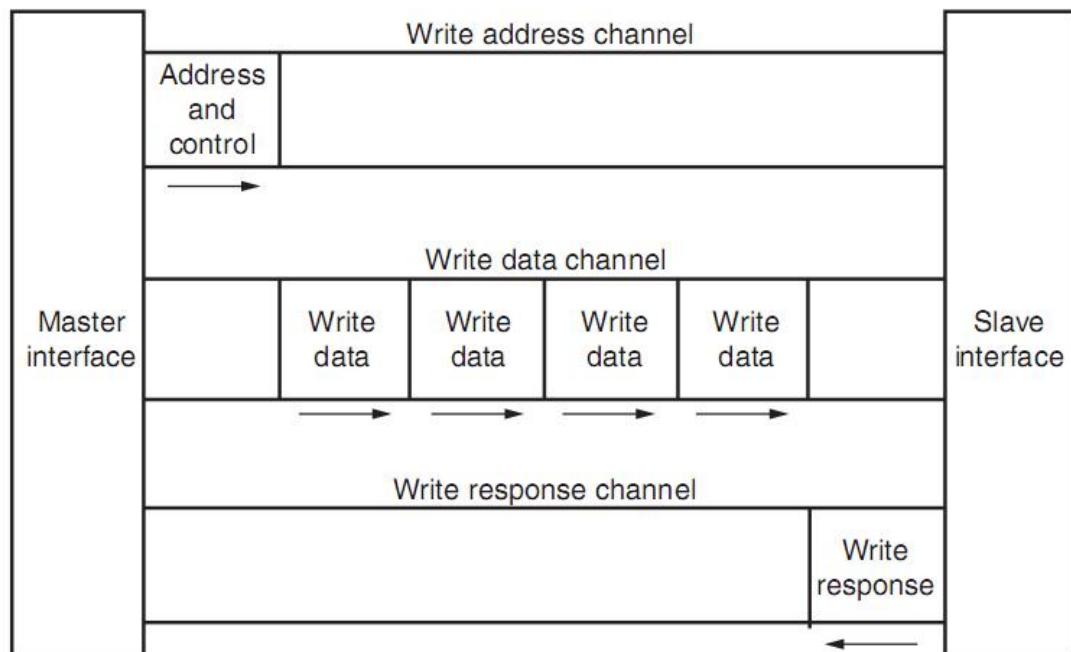
```

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。

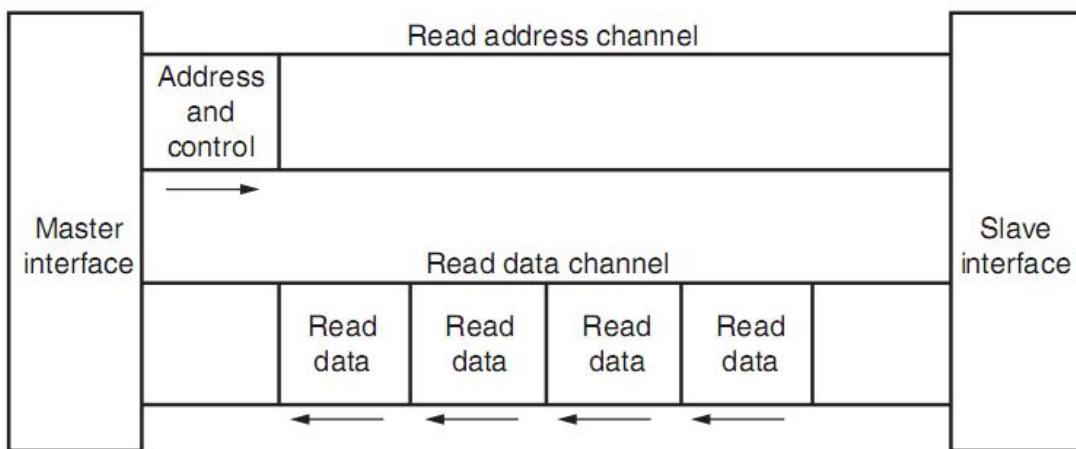
这里可能有的读者会问了，slv_reg0不是总线写过来的数据吗？因为笔者说过这个程序是Vivado为我们提供的例子，它这么做无非是想验证我写出去的值和我读进入的值相等。但是他怎么写确实会对初看代码的人造成困扰。

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;
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;
```

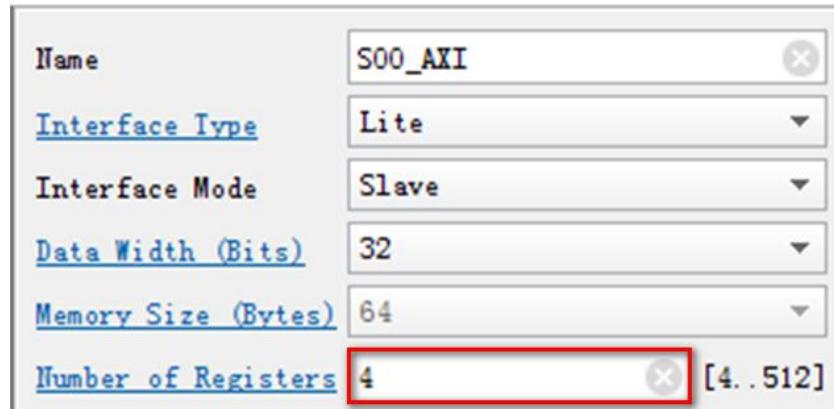
```

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
```

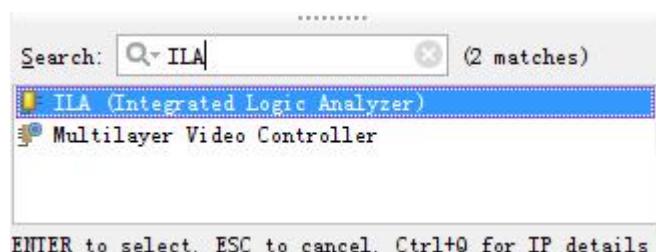
理论上只要基地址 + 偏移量不要超过HIGHADDR即可。

12.5 观察 AXI4-Lite 总线信号

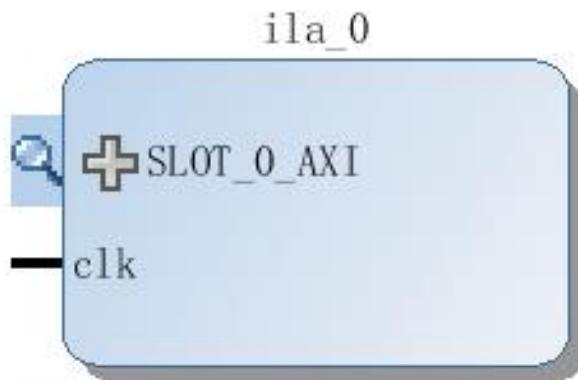
在第十章，我们封装了一个AXI_Lite的GPIO，通过本章的分析，我们在第十章工程的基础上通过添加一个ila核的方式，来具体看看AXI_Lite总线的信号。

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



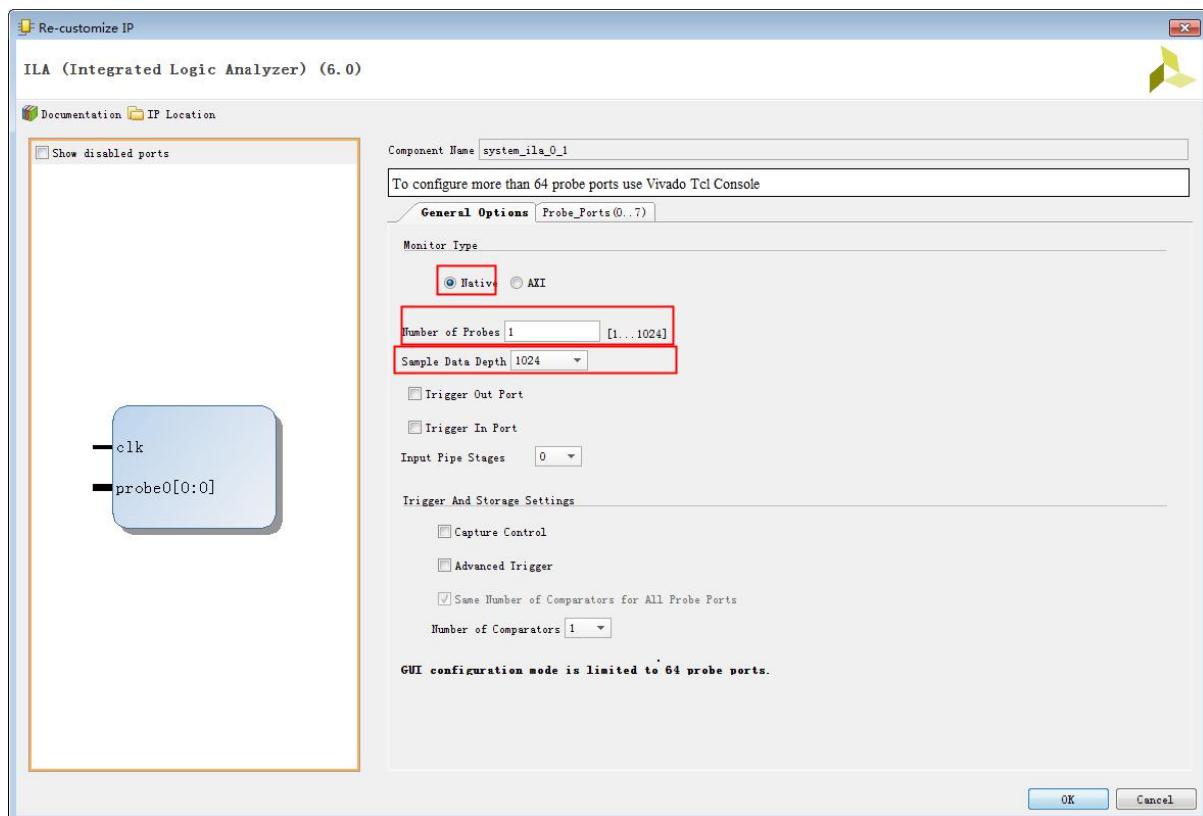
Step3: 双击打开ILA CORE



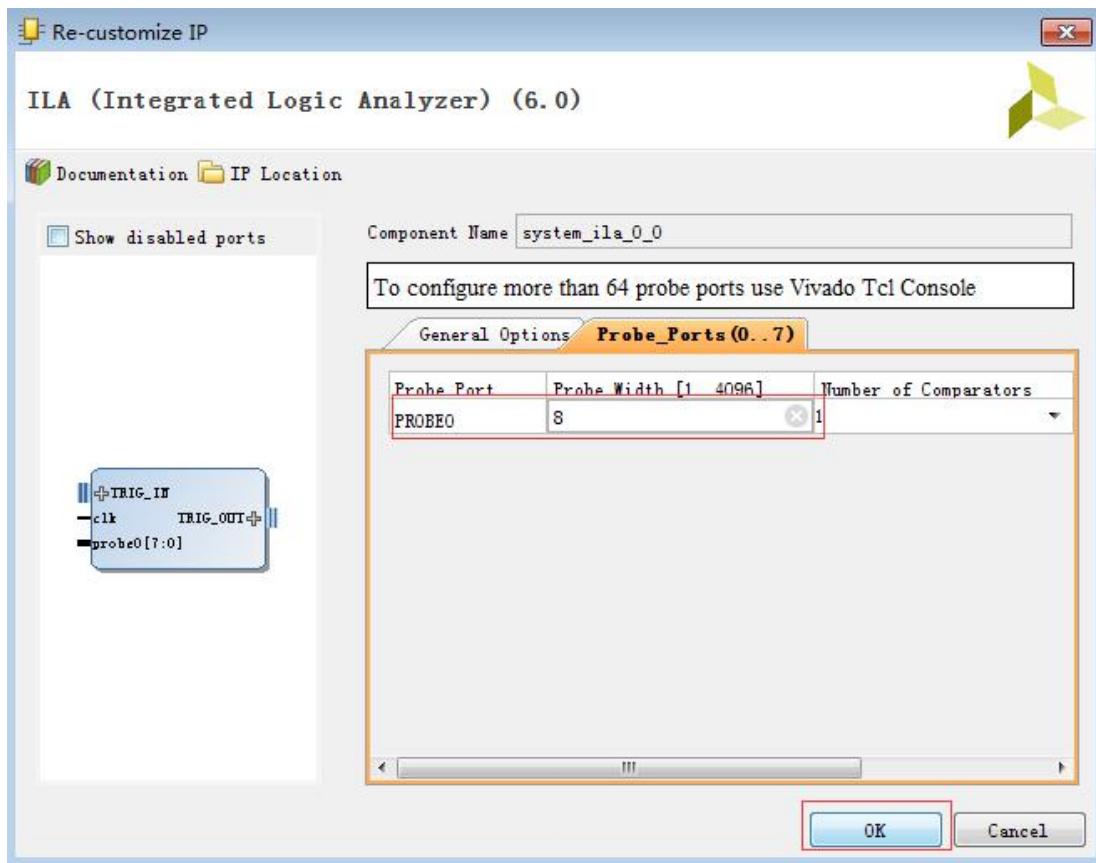
ILA (Integrated Logic Analyzer)

Step4: 双击打开 ILA CORE

General Options 设置如下



Probe_Ports 设置如下,之后单击 OK

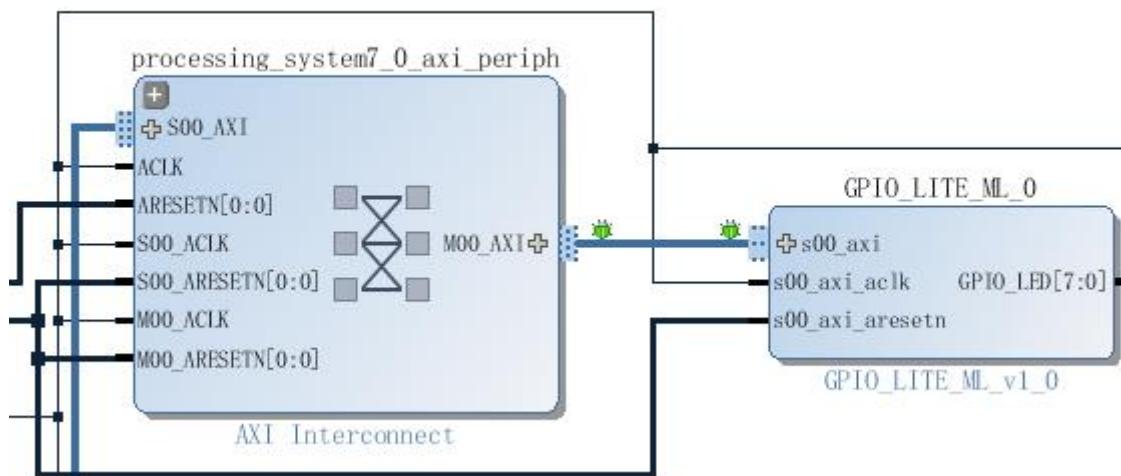


Step5: 连接 Probe0 到 GPIO_LED。

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

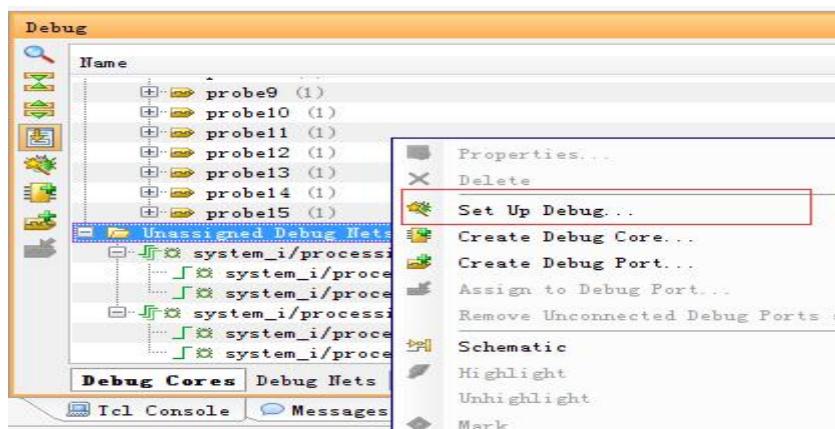
Products。

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。

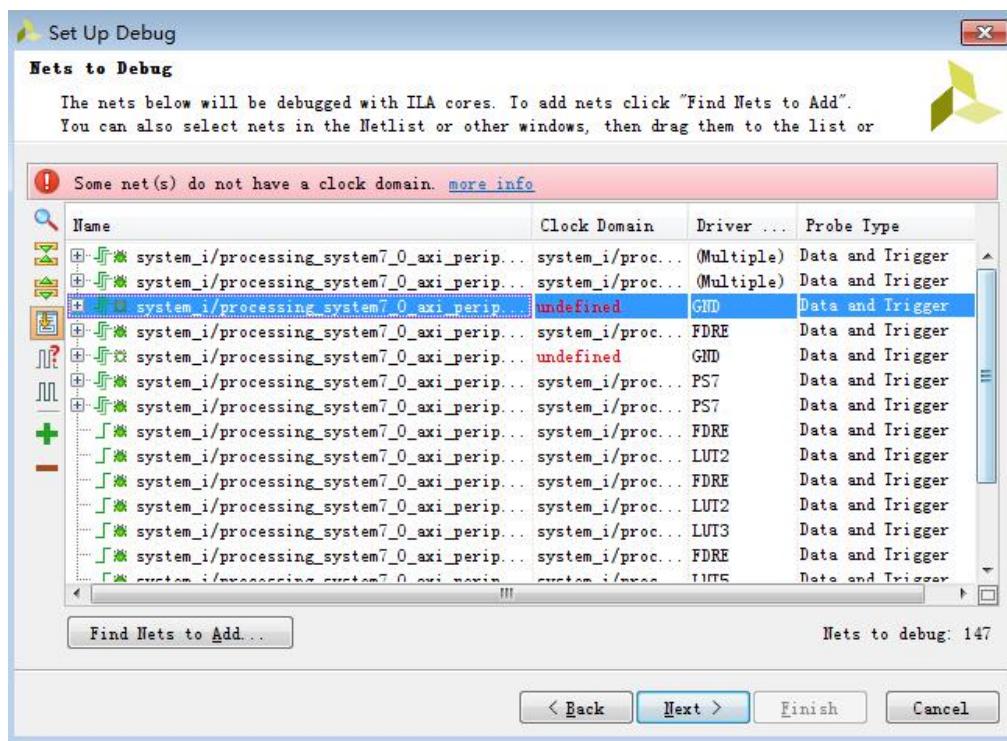
Step12:综合结束后选择 Synthesized Design option 单击 OK。

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



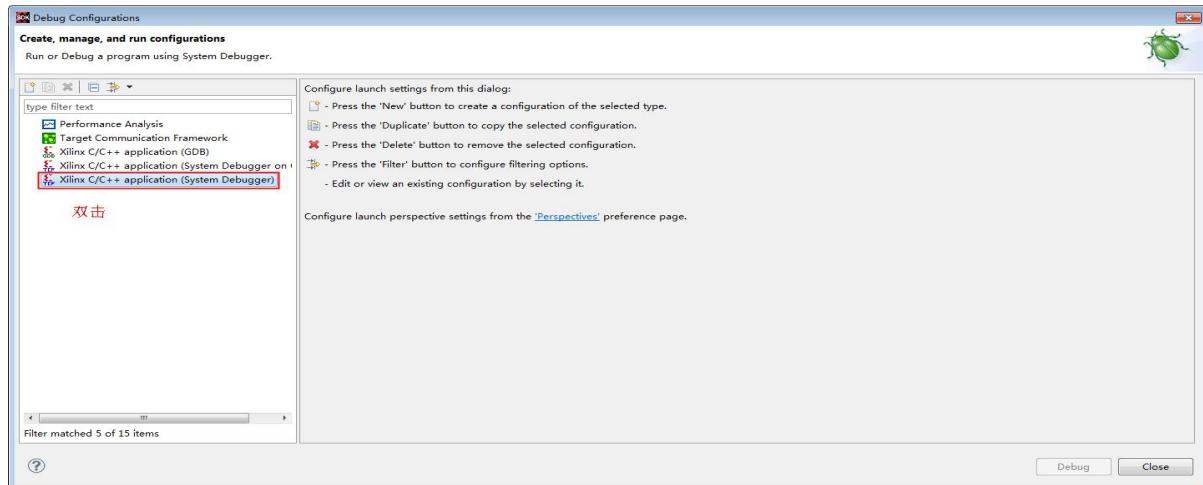
Step16:生成 Bit 文件。

12.6 加载到 SDK

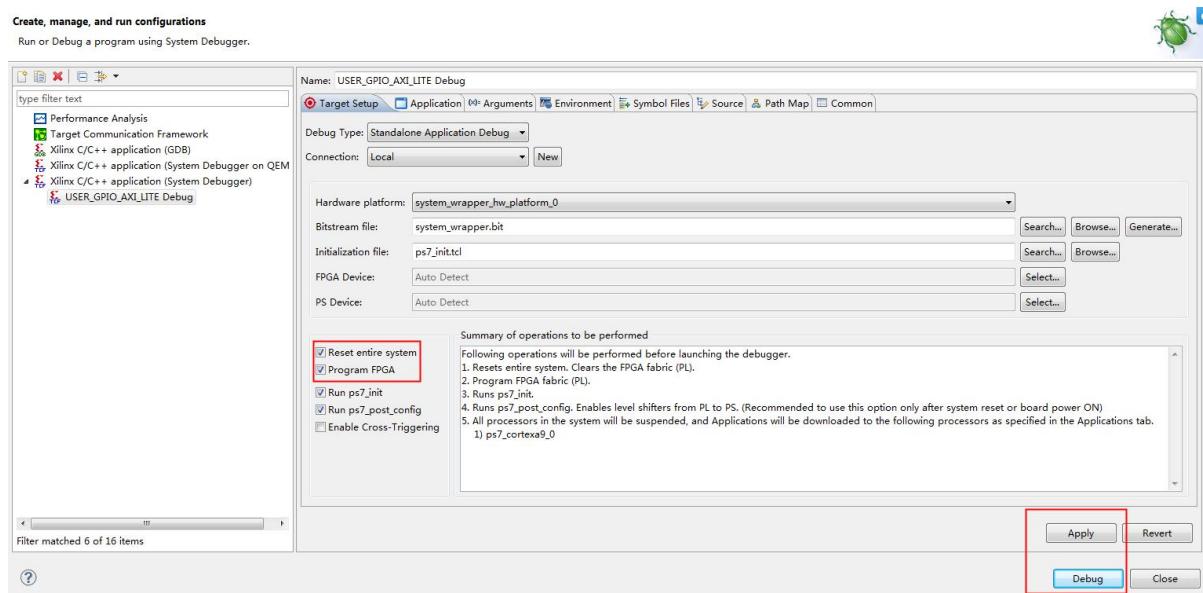
Step1: 导出硬件。

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

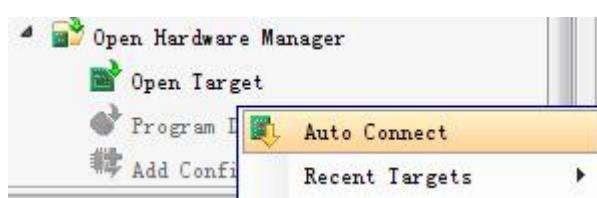
Step3: 选中 system Debugger,双击创建一个系统调试。



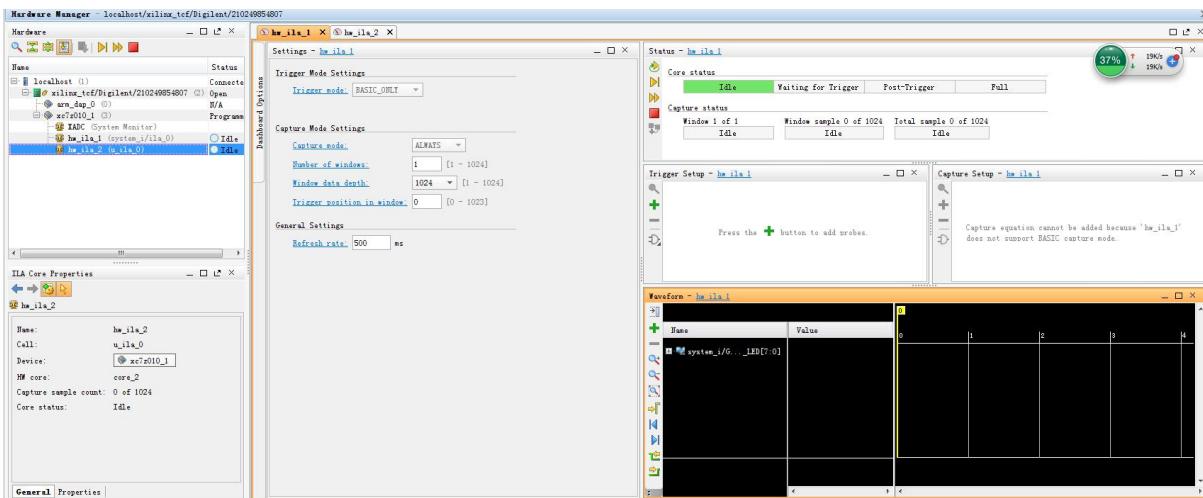
Step4: 设置系统调试。



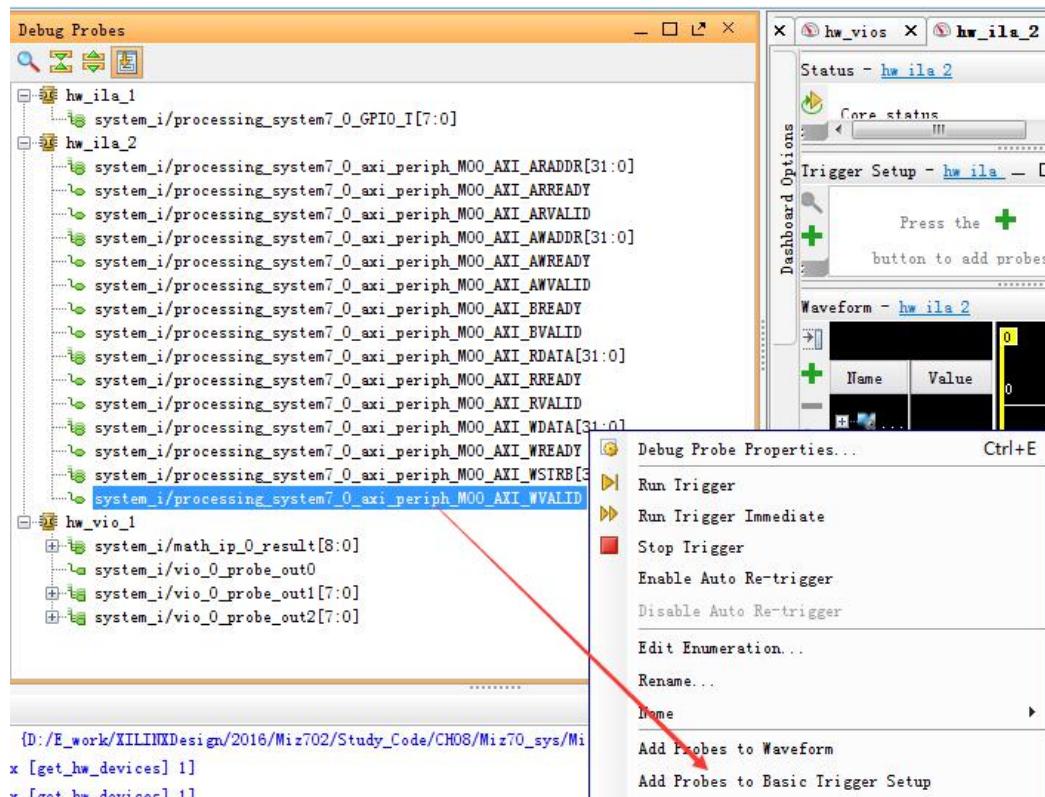
Step5:回到 VIVADO 单击 Open Target->Auto Connect



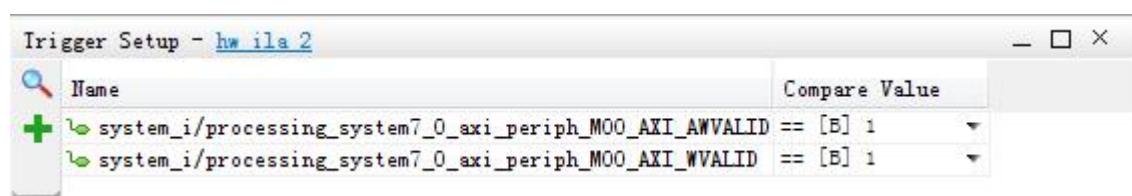
Step6: 加载完成后的界面



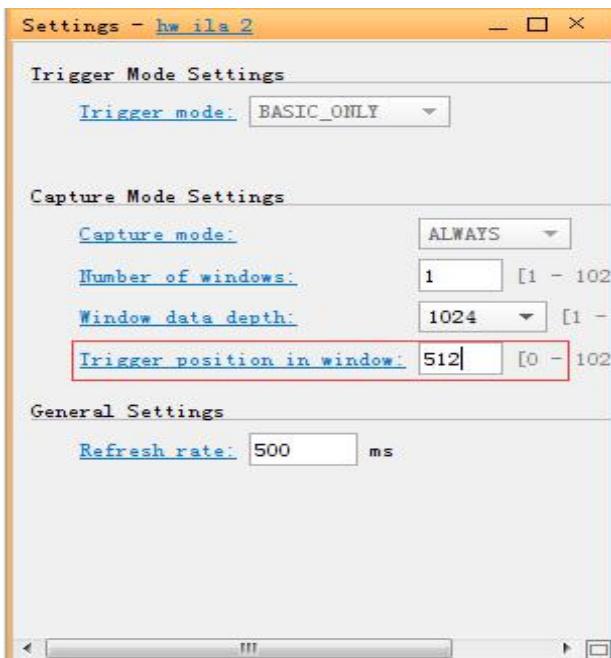
Step7: 选择菜单->window->Debugprobes 选择 AXI_WVALID 和 AXI_AWVALID 做为触发信号



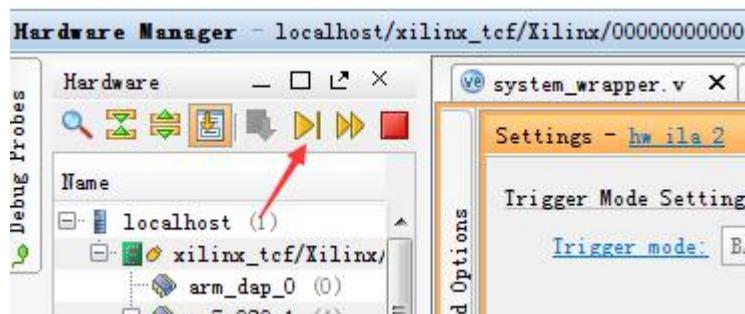
Step8: 设置触发条件为 1



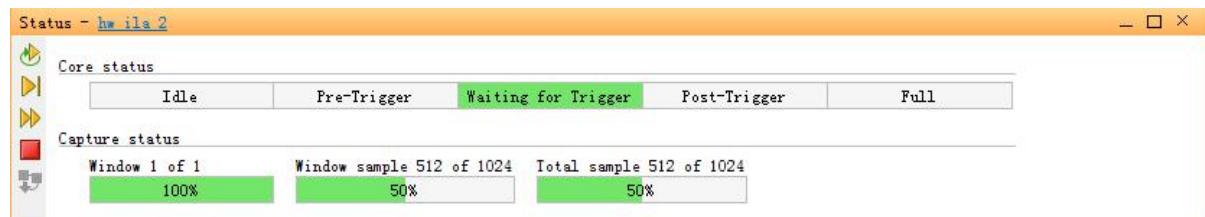
Step9: 设置触发位置为 512



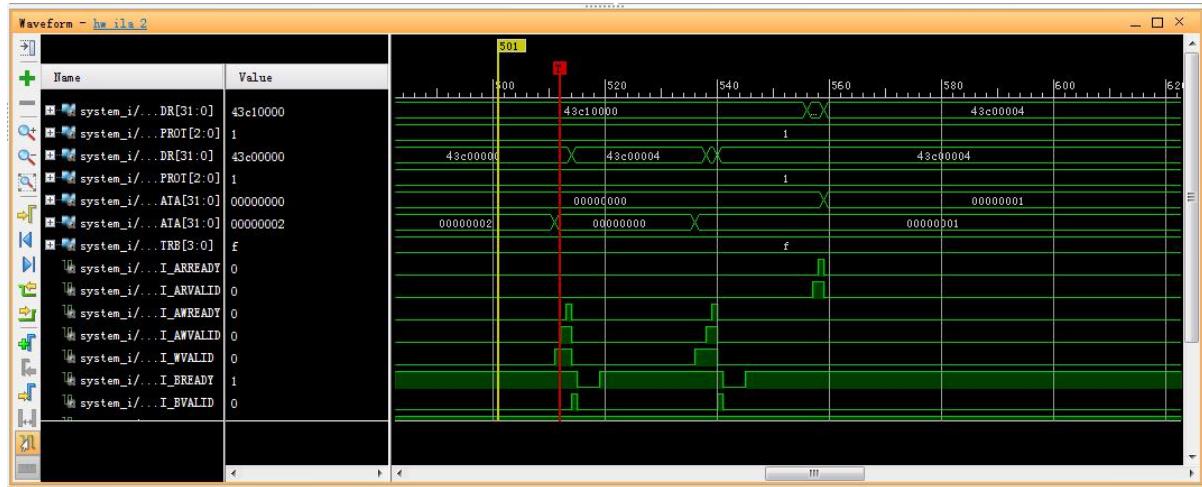
Step10:单击箭头所指向启动触发



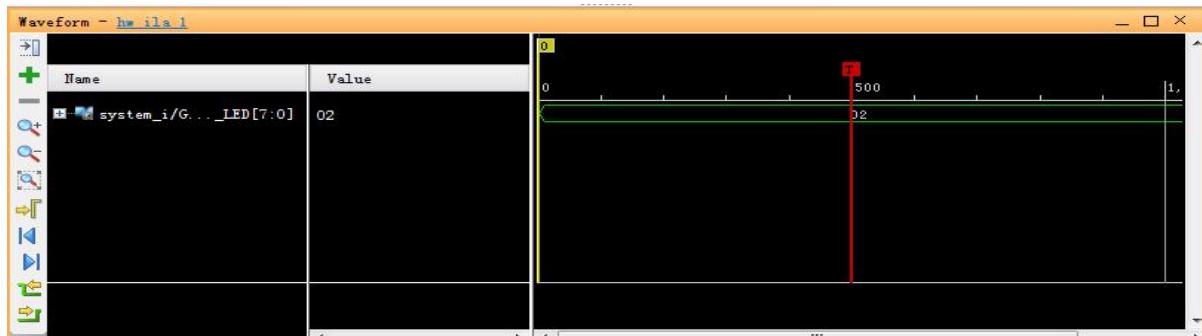
Step11:进入等待触发状态



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。

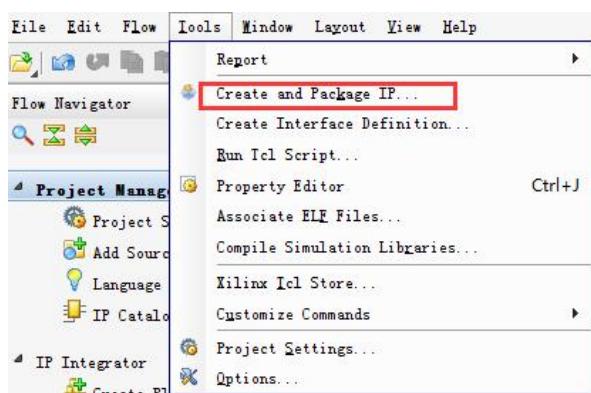
S02_CH13_AXI_PWM 实验

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。

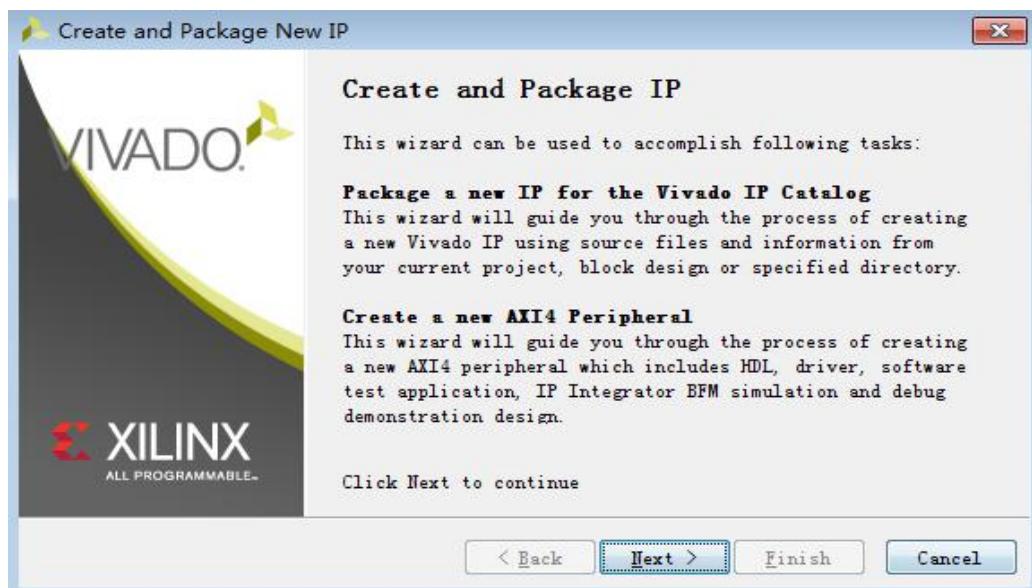
13.1 自定义 IP 的封装

Step1：新建一个名为 Miz_sys 空的工程。

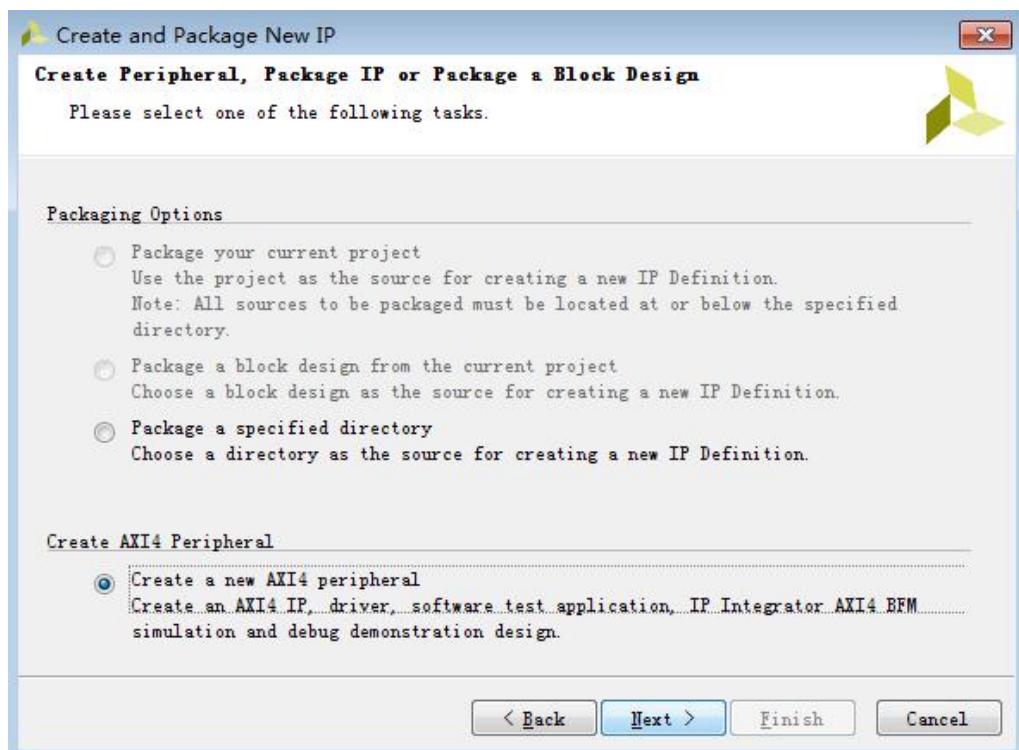
Step2：选择 Tools Create and Package IP 创建 IP



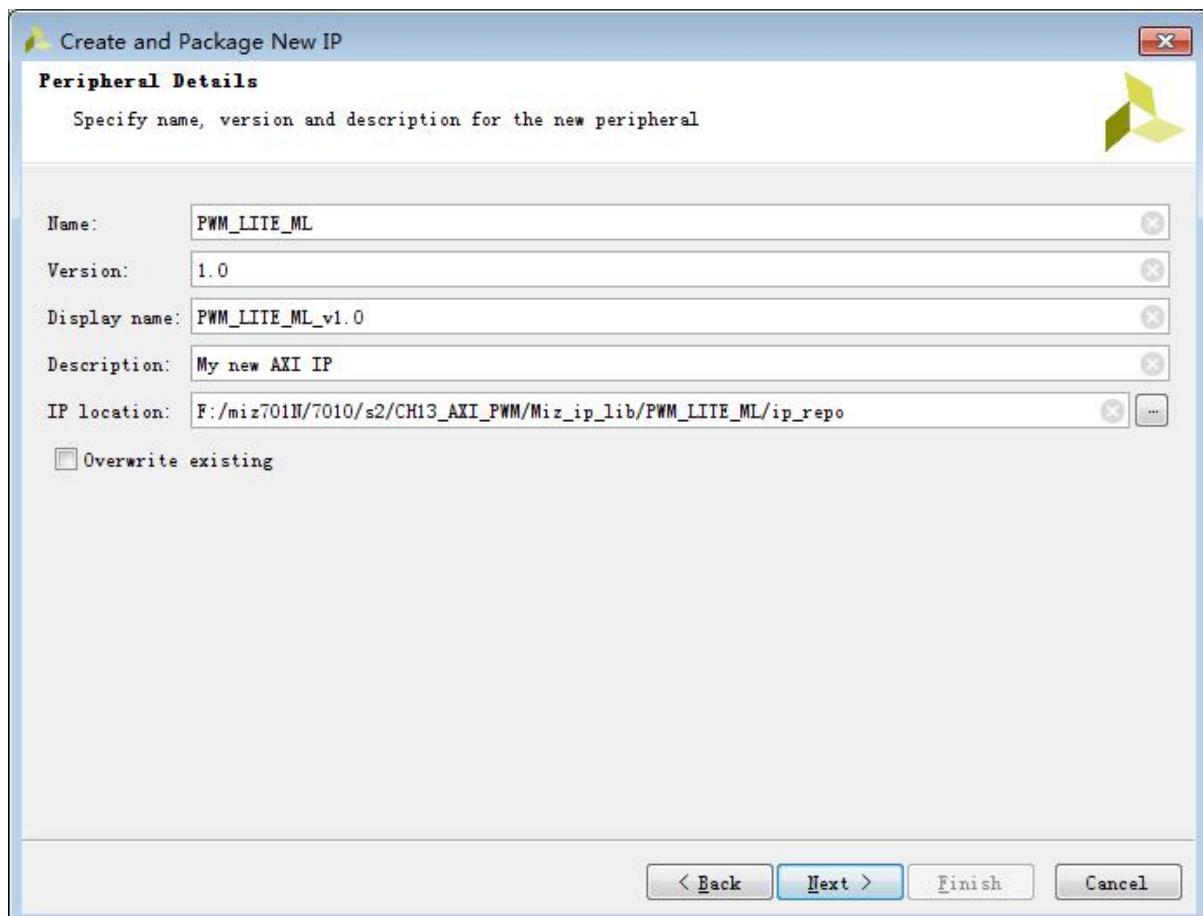
Step3:单击 NEXT



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

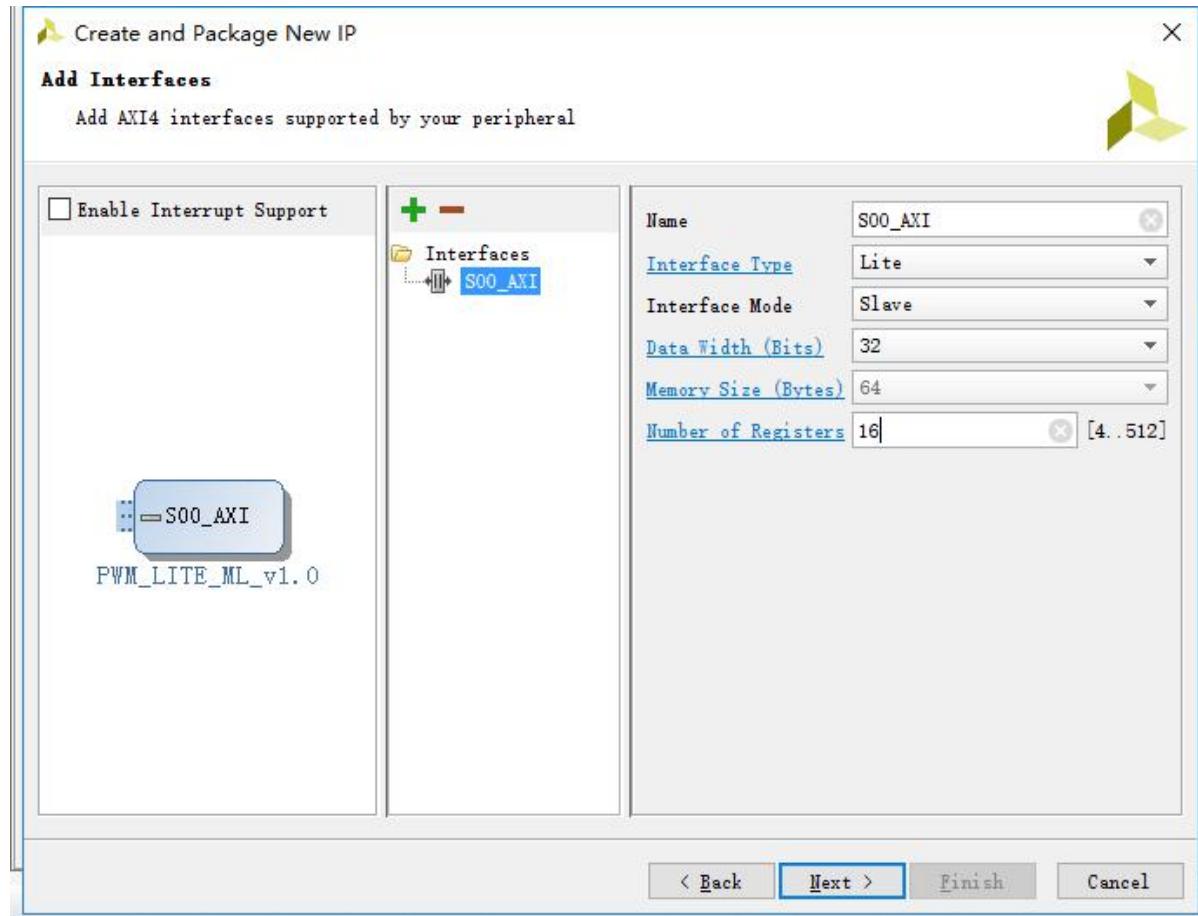


Step5:设置IP的名字为PWM_LITE_ML 版本号默认，并且记住IP的位置

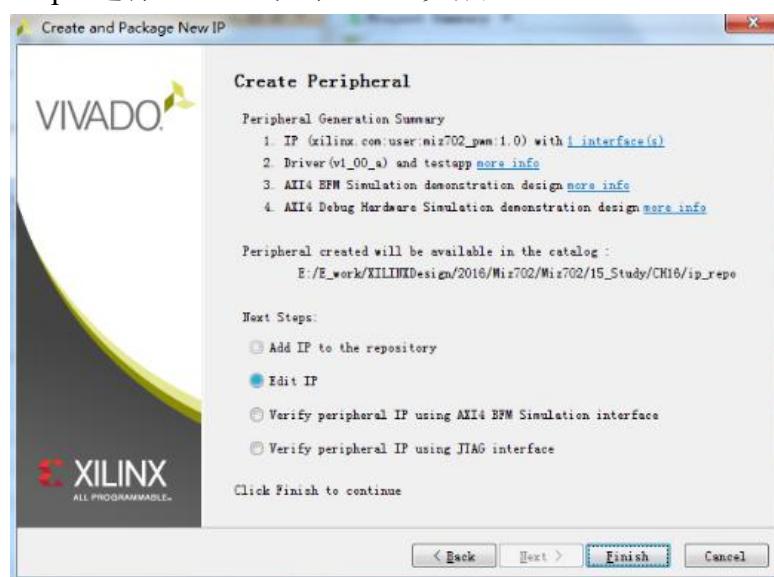


Step6:设置总线形式为 Lite 总线，Lite 总线是简化的 AXI 总线消耗的资源少，当然性能

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。



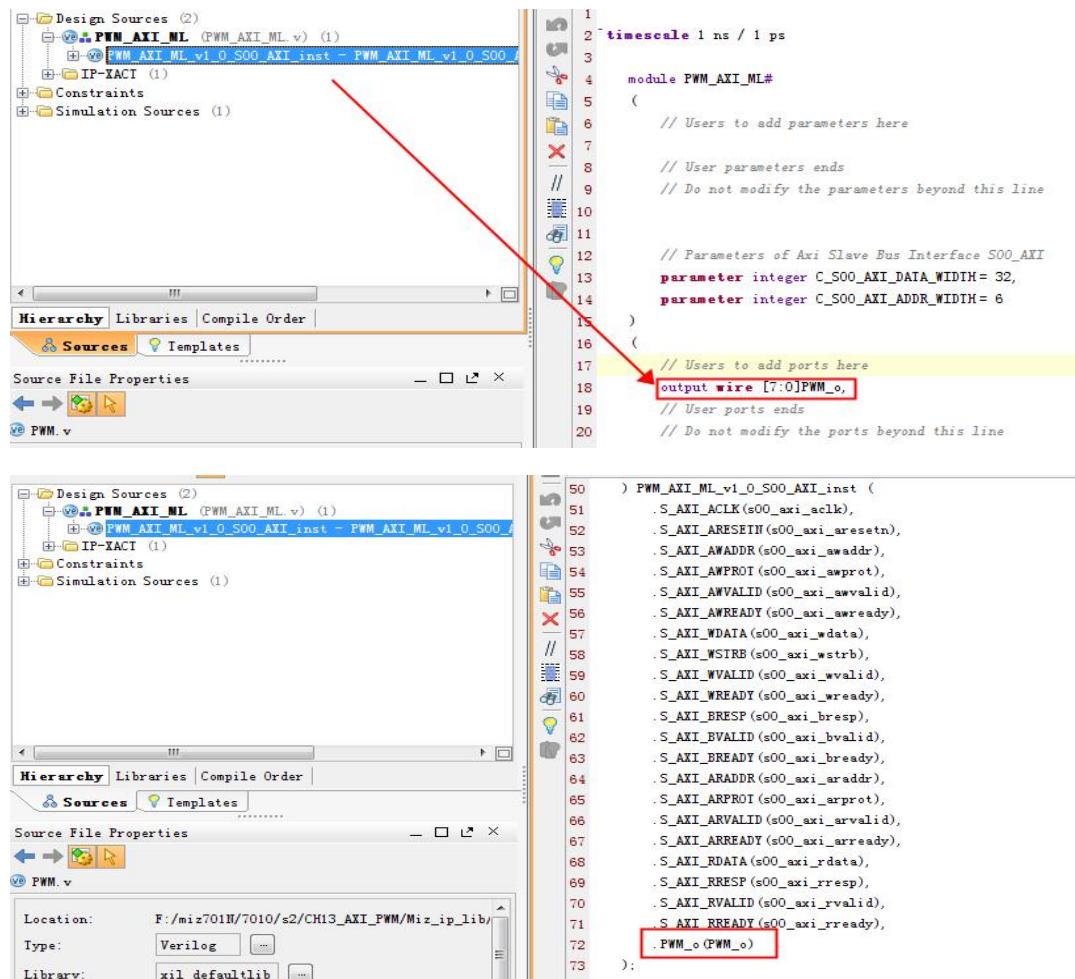
Step7:选择 Edit IP 单击 Finish 完成



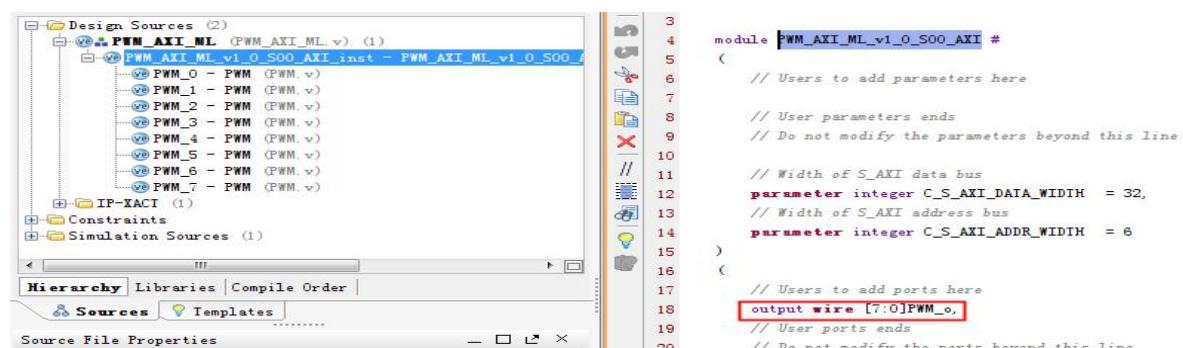
13.2 miz702_pwm 用户 IP 的修改

IP 创建完成后，并不能立马使用，还需要做一些修改。

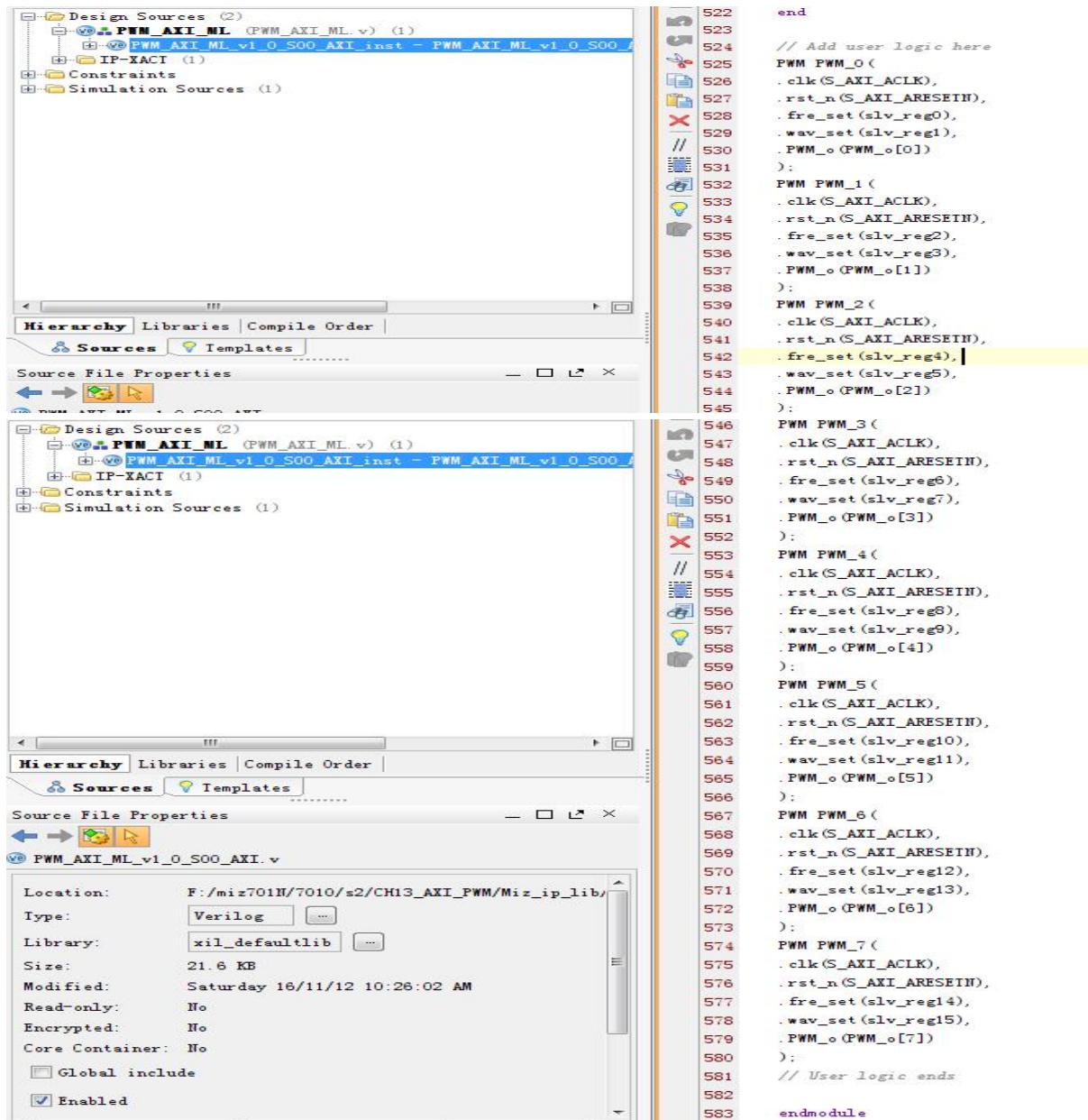
Step1: 打开 PWM_AXI_ML.v 文件在以下位置修改:



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。

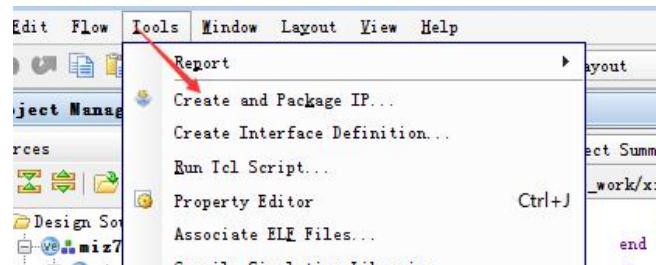
```

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

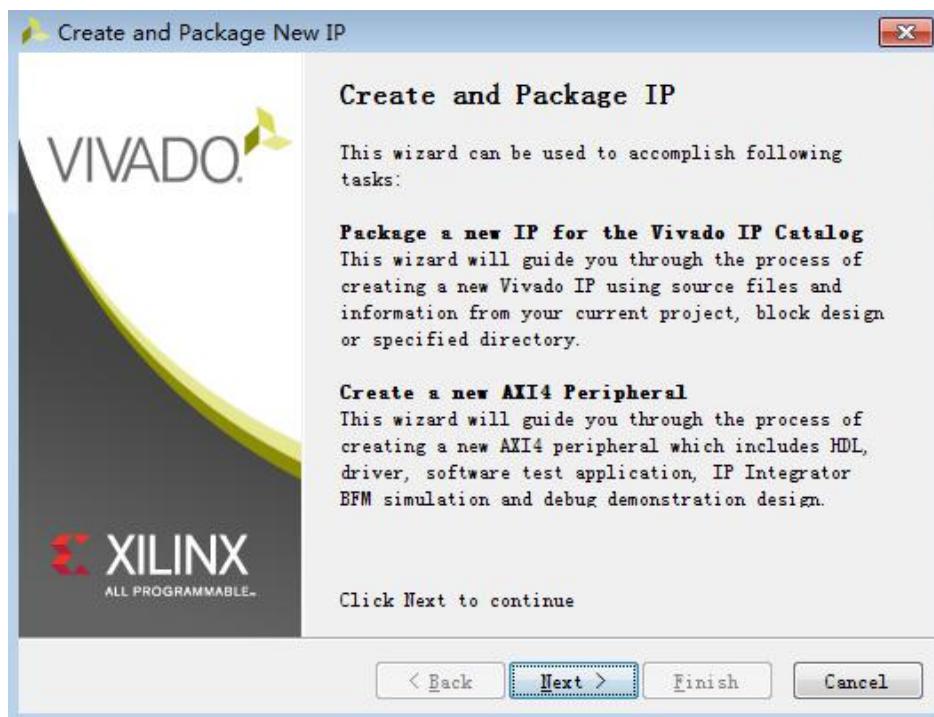
```

```
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
```

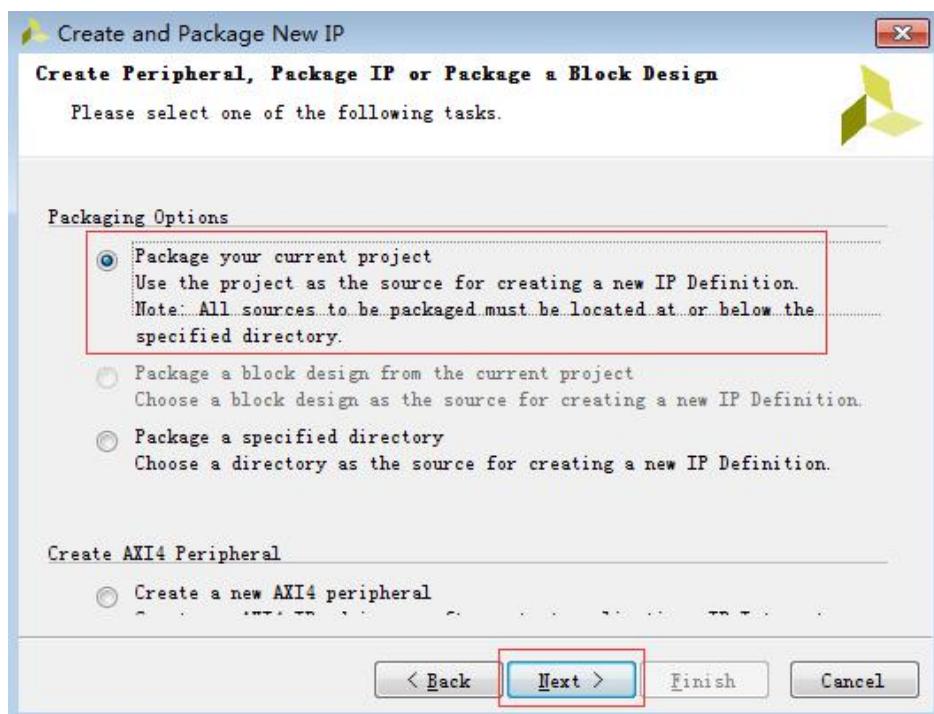
Step5:修改完成后重新封装一次自定义 IP



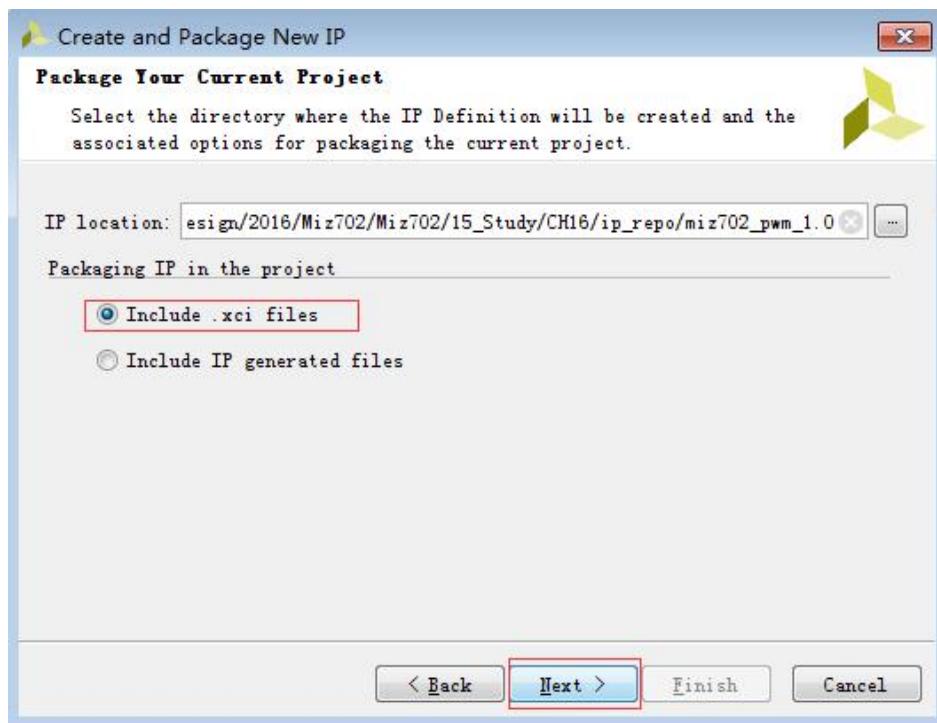
Step6:单击 NEXT



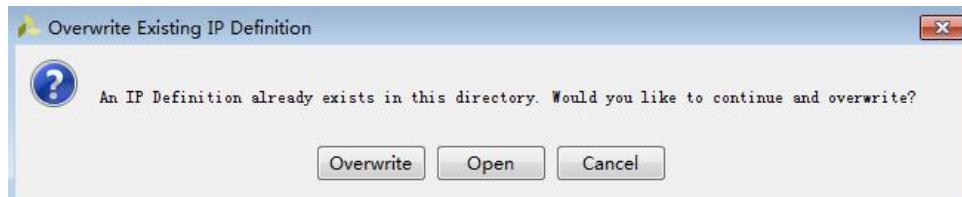
Step7:和第一次不同，这次选择第一个单选框然后单击 NEXT



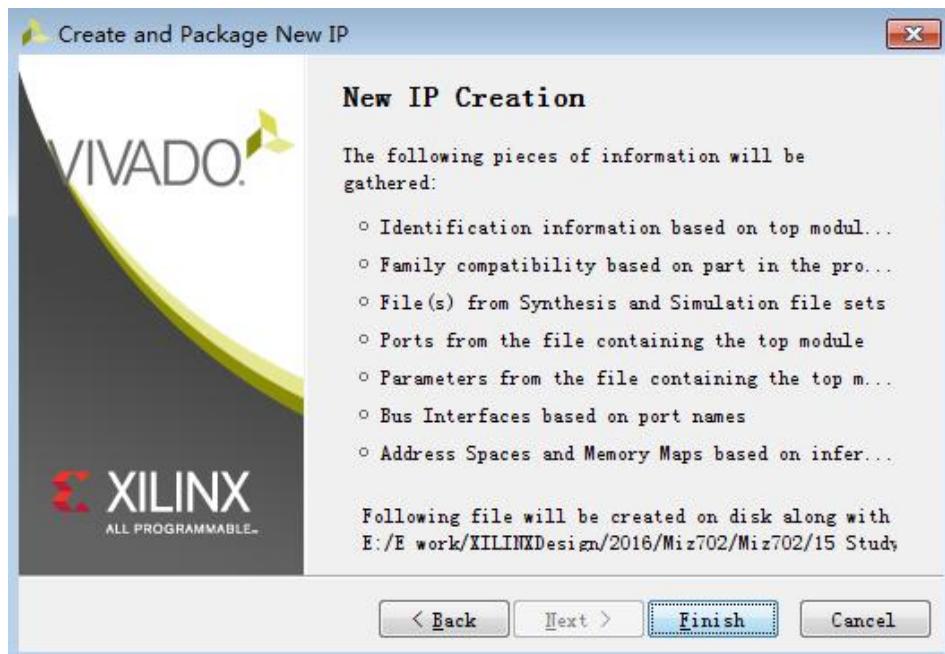
Step8:选择第一个单选框，然后单击 NEXT



Step9:点击 Overwrite



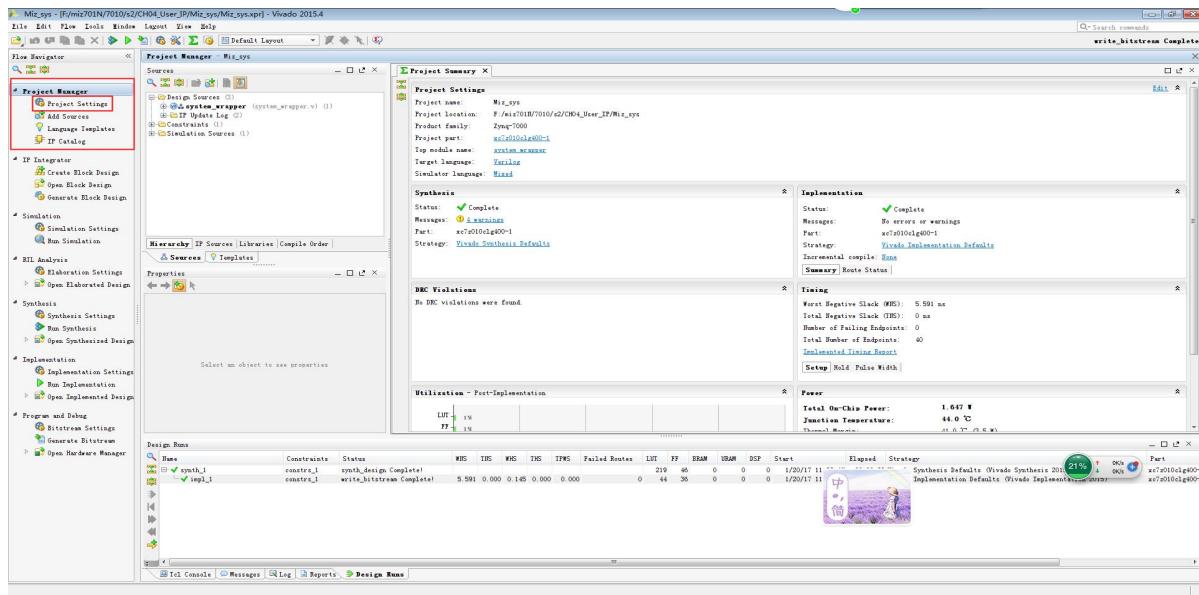
Step10:点击 Finish 到此自定义 IP 结束



13.3 搭建硬件工程

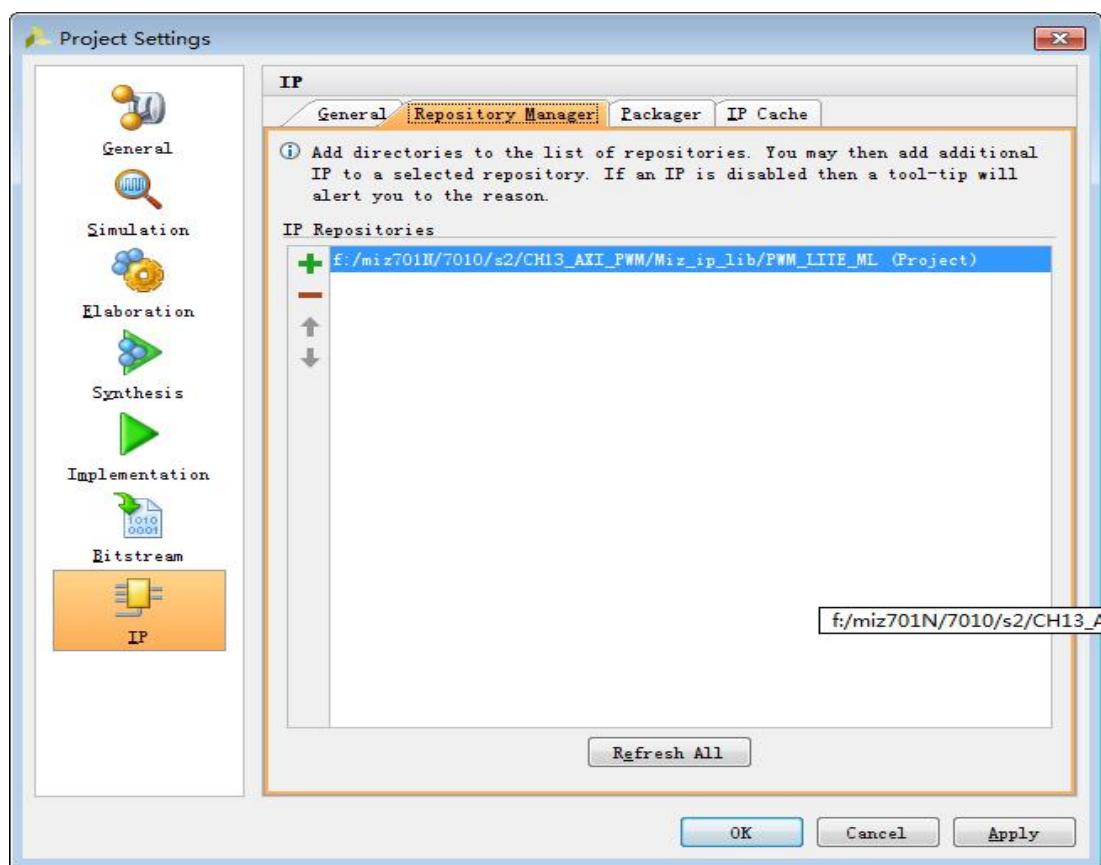
Step1：另外新建一个VIVADO工程，根据自己的开发板正确配置芯片型号。

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。

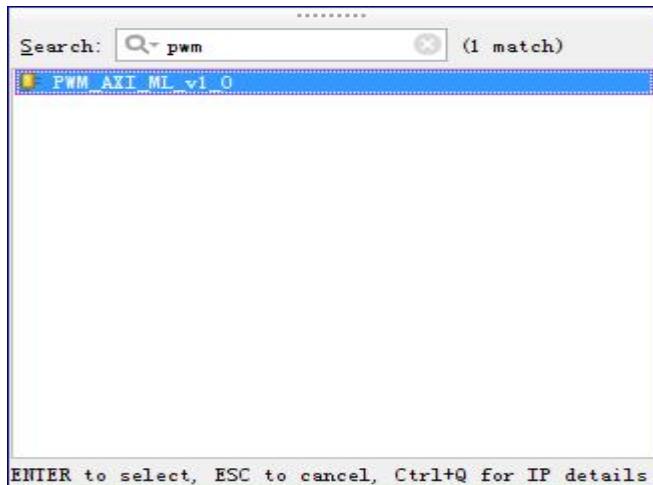
Step4：单击+号图标，将上一节封装的IP的路劲存放进去，单击OK。



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。

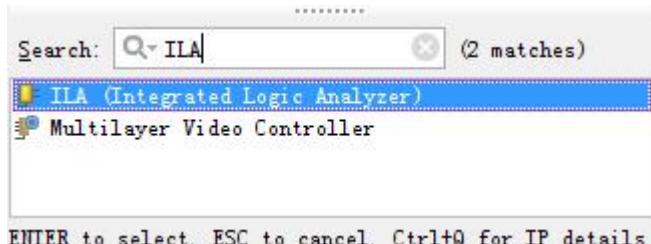


Step8：直接点击 Run connection automation，然后单击 OK。

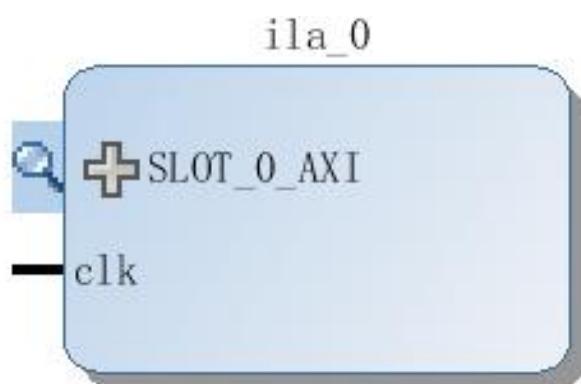
Step9：选中 PWM_o，按 Ctrl+T 组合键引出端口。



Step10：单击 IP icon 添加 ila CORE



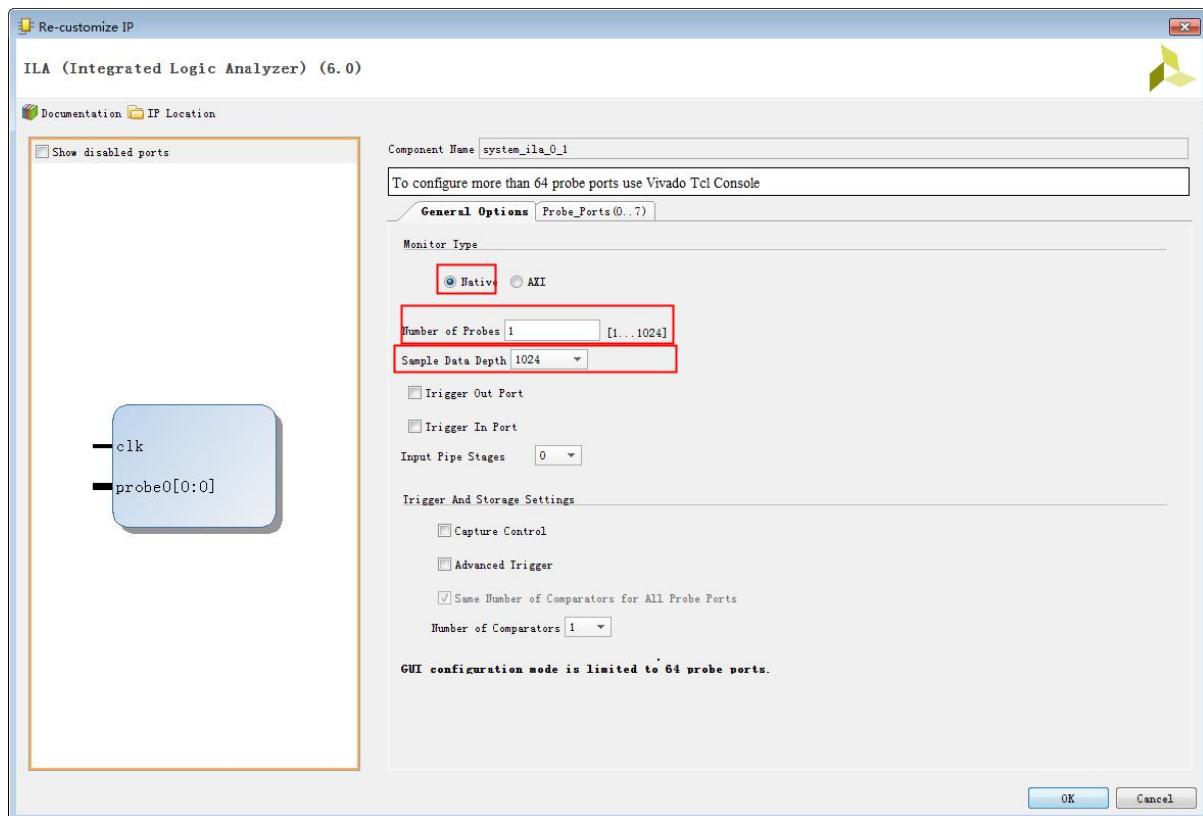
Step11: 双击打开 ILA CORE



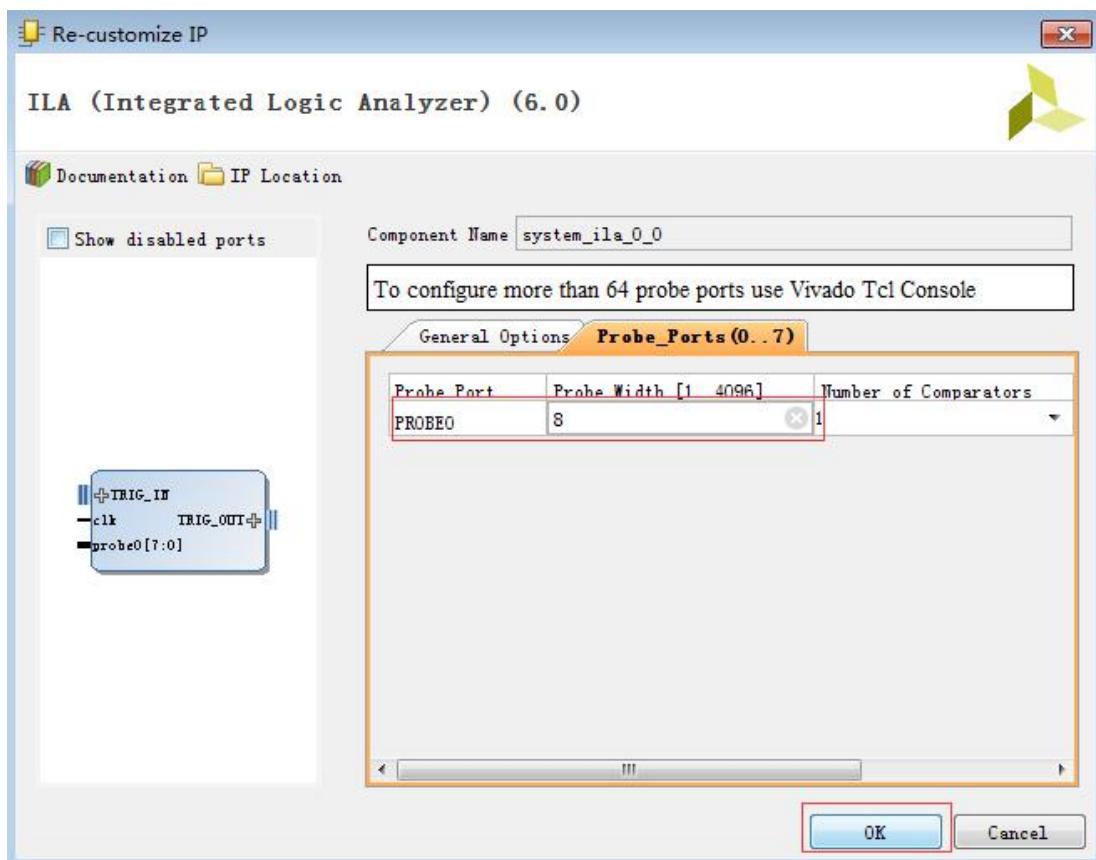
ILA (Integrated Logic Analyzer)

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: 添加一个约束文件，打开对应自己硬件的原理图，查看 LED 部分引脚连接情况，此次我们只用 4 个 LED 完成实验。Miz702 约束文件如下所示：

```
set_property SEVERITY {Warning} [get_drc_checks NSTD-1]
set_property SEVERITY {Warning} [get_drc_checks UCI0-1]

set_property PACKAGE_PIN T22 [get_ports {PWM_o[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {PWM_o[0]}]

set_property PACKAGE_PIN T21 [get_ports {PWM_o[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {PWM_o[1]}]

set_property PACKAGE_PIN U22 [get_ports {PWM_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {PWM_o[2]}]

set_property PACKAGE_PIN U21 [get_ports {PWM_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {PWM_o[3]}]
```

其他型号开发板参照对应型号的原理图的 LED 部分，修改成对应的引脚即可。

Step11: 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，并添加一个 main.c 的文件。

Step3: 在 main.c 文件中添加以下程序，按 Ctrl+S 保存后自动开始编译。

```
#include <stdio.h>
#include "xparameters.h"
#include "xil_io.h"
#include "sleep.h"
#include "xil_types.h"
```

```

int main()
{
    Xil_Out32(XPAR_PWM_AXI_ML_0_BASEADDR,99); //pwm0 fre
    Xil_Out32(XPAR_PWM_AXI_ML_0_BASEADDR+4,10); //pwm0 wav

    Xil_Out32(XPAR_PWM_AXI_ML_0_BASEADDR+8,99); //pwm1 fre
    Xil_Out32(XPAR_PWM_AXI_ML_0_BASEADDR+12,20); //pwm1 wav

    Xil_Out32(XPAR_PWM_AXI_ML_0_BASEADDR+16,99); //pwm2 fre
    Xil_Out32(XPAR_PWM_AXI_ML_0_BASEADDR+20,40); //pwm2 wav

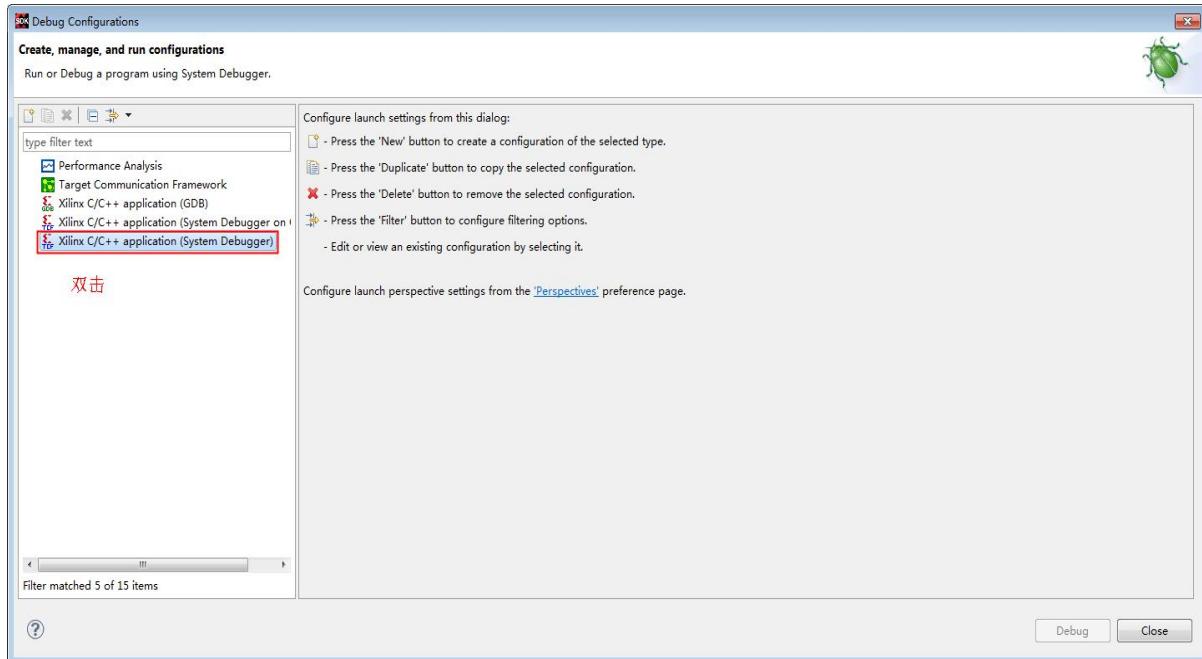
    Xil_Out32(XPAR_PWM_AXI_ML_0_BASEADDR+24,99); //pwm3 fre
    Xil_Out32(XPAR_PWM_AXI_ML_0_BASEADDR+28,80); //pwm3 wav

    return 0;
}

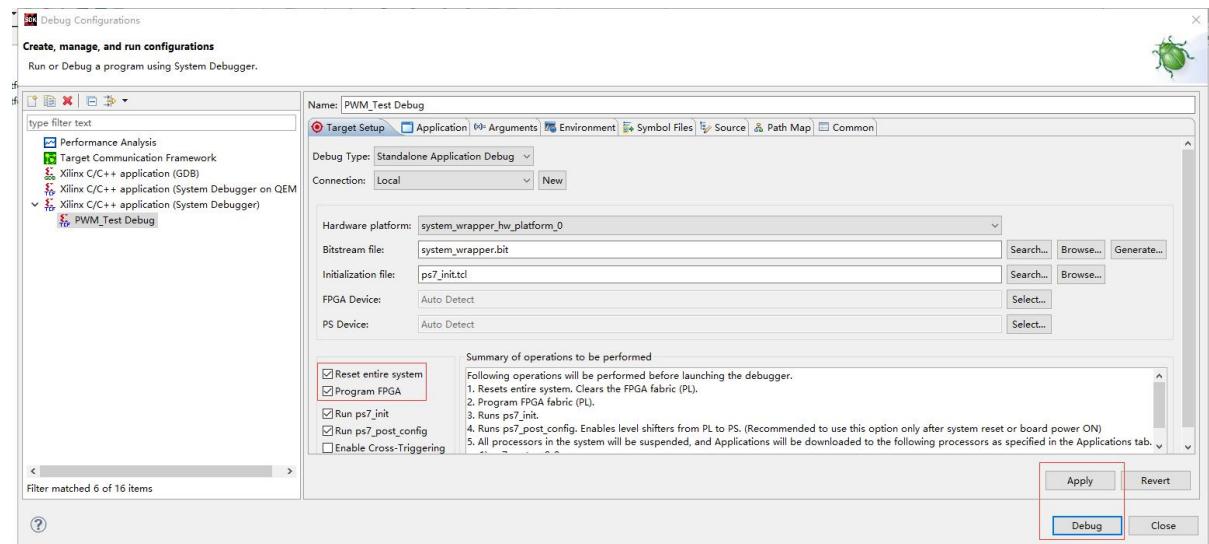
```

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。

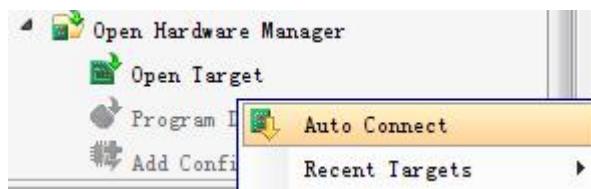


Step6: 设置系统调试。

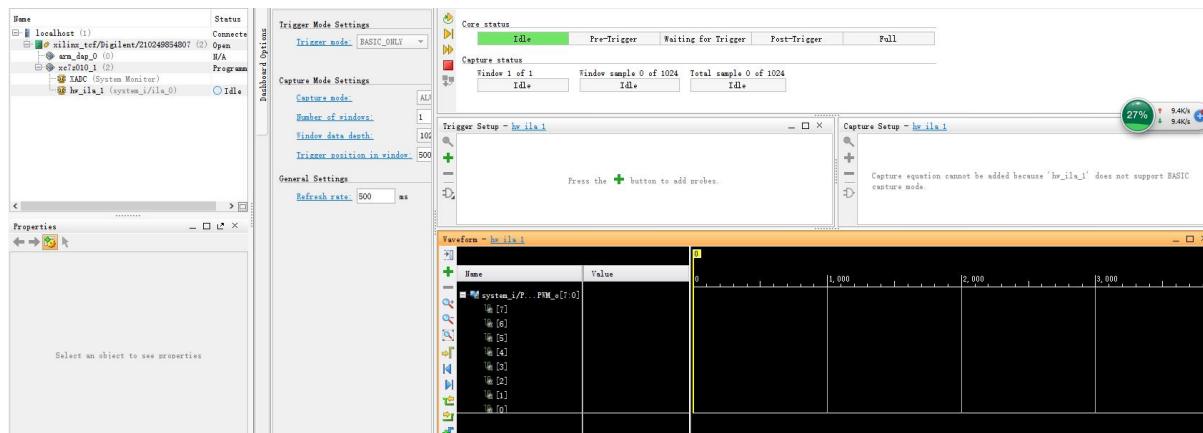


Step7: 单击运行程序按钮 运行程序。

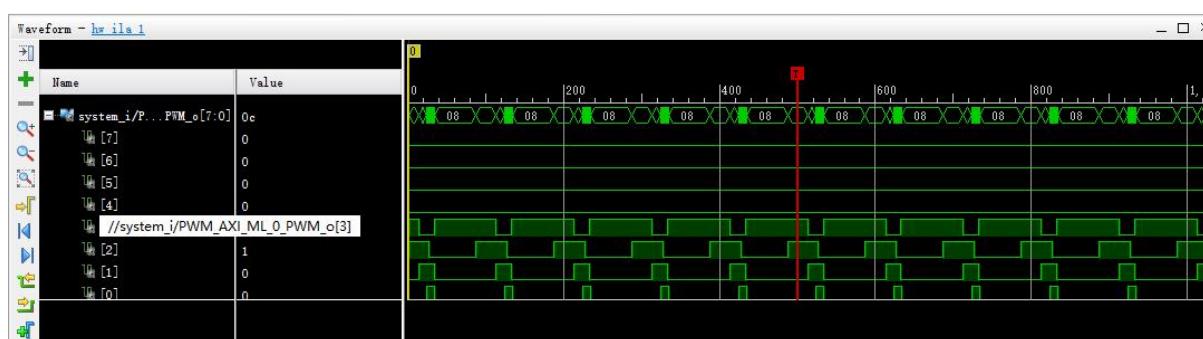
Step8: 回到 VIVADO 单击 Open Target->Auto Connect



Step9: 加载完成后的界面



Step10: 单击箭头所指向启动触发,窗口显示采集到的信号波形。

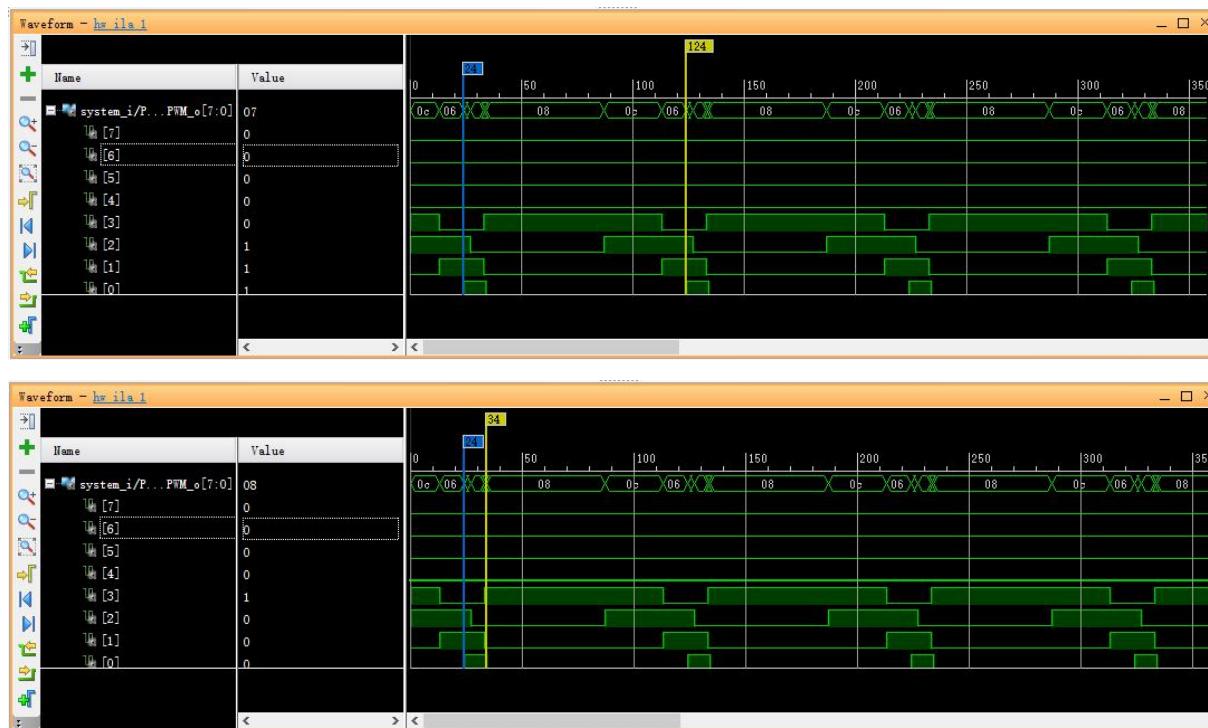


13.5 程序分析

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_0(
    .clk(S_AXI_ACLK),
    .rst_n(S_AXI_ARESETN),
    .fre_set(slv_reg0), ←
    .wav_set(slv_reg1), ←
    .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。

S02_CH14_EMIO_OLED 实验

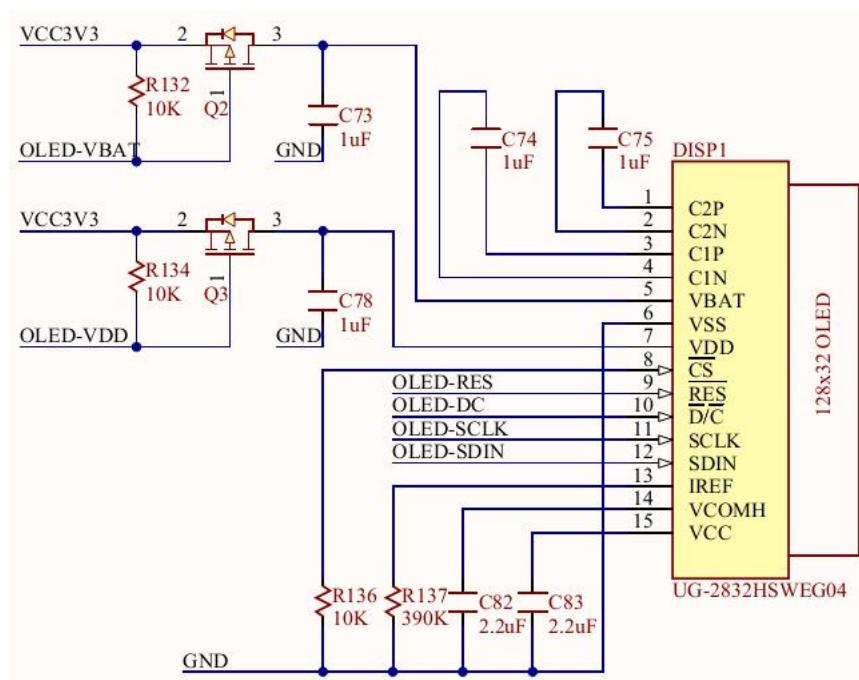
本章将使用 EMIO 模拟 OLED 的时序来驱动 OLED，本方案对米联系列 MIZ702, MIZ702N 和 MIZ701N 全兼容。

14.1 板载 OLED 硬件原理

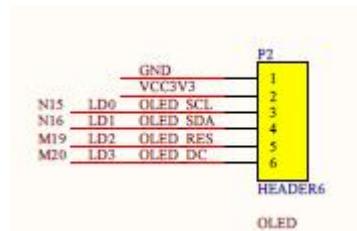
MIZ 系列开发板板载 OLED 的型号是 UG-2832HSWEG04(MIZ701N 为 UG-2864HSWEG04)，分辨率为 128*32 (MIZ701N 为 128*64) ，接口类型为 4 线 SPI，控制芯片为 SSD1306。本小节，首先简要分析开发板 OLED 相关的硬件电路，然后对 SSD1306 控制器进行介绍，为后续的驱动开发做好铺垫。

14.1.1 硬件电路简析

MIZ702 和 MIZ702N OLED 接口电路如下图所示。



MIZ701N 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。高电平表示总线上传输的是数据，低电平表示总线上传输的是命令。
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。

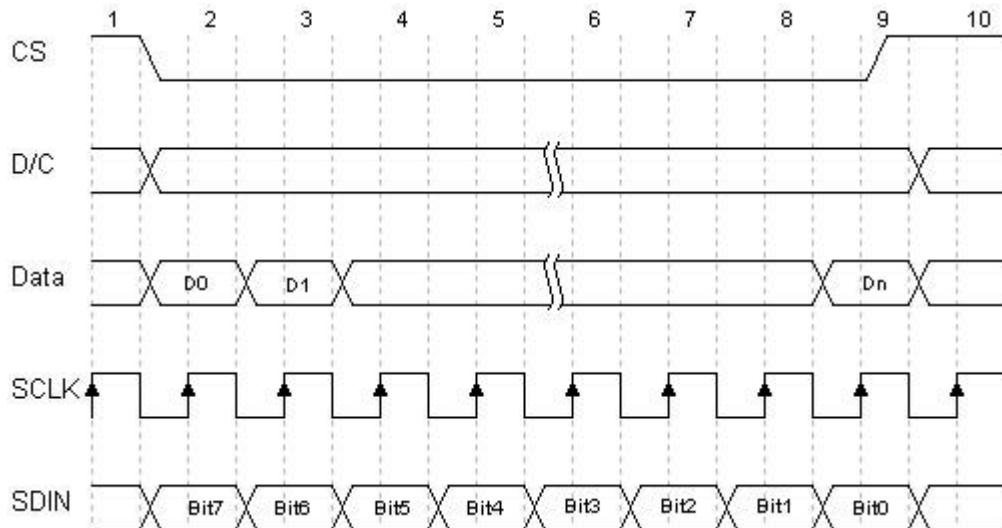
从原理图中可以看出片选信号 CS 通过电阻短接到 GND，因此该信号是一直有效的；OLED-RES、OLED-DC、OLED-SCLK、OLED-SDIN 直接连接到 Zynq GPIO，其中 RES 和 DC 信号低电平有效；PIN7 VDD 和 PIN5 VBAT 是高电平有效的，但是并非直接连接至 Zynq GPIO，而是通过 PMOS 管进行驱动。根据 PMOS 管的导通特性可以知道，当 OLED_VBAT 和 OLED-VDD 为低电平时，3.3V 的电压才会送到 VBAT 和 VDD，换句话说，对于 Zynq 而言，VBAT 和 VDD 是低电平有效。市面上大多是将 VBAT 和 VDD 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：

PAGE0 (COM0-COM7)	Page 0	Row re-mapping PAGE0 (COM 63-COM56)
PAGE1 (COM8-COM15)	Page 1	PAGE1 (COM 55-COM48)
PAGE2 (COM16-COM23)	Page 2	PAGE2 (COM47-COM40)
PAGE3 (COM24-COM31)	Page 3	PAGE3 (COM39-COM32)
PAGE4 (COM32-COM39)	Page 4	PAGE4 (COM31-COM24)
PAGE5 (COM40-COM47)	Page 5	PAGE5 (COM23-COM16)
PAGE6 (COM48-COM55)	Page 6	PAGE6 (COM15-COM8)
PAGE7 (COM56-COM63)	Page 7	PAGE7 (COM 7-COM0)

SEG0 ----- SEG127
Column re-mapping SEG127 ----- SEG0

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

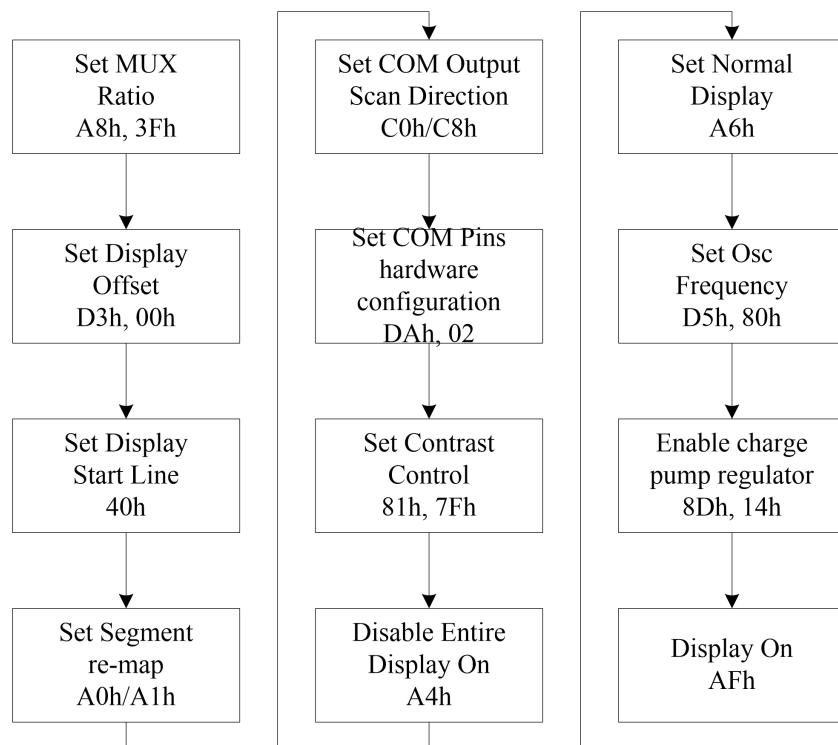
- 写数据

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 ₇	0 A ₆	0 A ₅	0 A ₄	0 A ₃	0 A ₂	0 A ₁	1 A ₀	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 ₀	全部显示开	A4h, X ₀ = 0 : 恢复内存内容显示(默认), 输出内存中的内容 A5h, X ₀ = 1 : 开显示, 输出无视内存的内容
0	A6/A7	1	0	0	0	0	1	1	X ₀	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 ₀	设置显示开 /关	AE: X[0]= 0: 关显示(默认) AF: 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	X ₀	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]	*	*	*	*	*	B ₂	B ₁	B ₀	设置	(水平滚动1列)
0	C[2:0]	*	*	*	*	*	C ₂	C ₁	C ₀		[7:0]: 虚拟字节(设置为00 h)
0	D[2:0]	*	*	*	*	*	D ₂	D ₁	D ₀		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	X ₃	X ₂	X ₁	X ₀	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	X ₃	X ₂	X ₁	X ₀	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 0	20 A[1:0]	0 * 0 * *	0 * *	1 * *	0 * *	0 * *	0 A ₁	0 A ₀	0	设置内存寻址模式	A[1:0]= 00, 水平寻址模式 A[1:0]= 01, 垂直的寻址模式 A[1:0]= 10, 页面寻址模式(重置) A[1:0]= 11, 无效
0 0 0	21 A[6:0] B[6:0]	0 * 0 * A ₆ * B ₆	0 A ₅	1 A ₄	0 A ₃	0 A ₂	0 A ₁	1 A ₀	0 B ₀	设置列地址	设置列开始和结束地址 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 * *	0 * *	1 * *	0 * *	0 A ₂	1 A ₁	0 B ₁	0 B ₀	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 ₂	X ₁	X ₀	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 ₅	X ₄	X ₃	X ₂	X ₁	X ₀	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 ₅ X ₄ X ₃ X ₂ X ₁ X ₀ 。 在复位后起始行地址为0。
0	A0/A1	1	0	1	0	0	0	0	X ₀	设置段重映射	A0, X[0]= 0: 列地址0映射到SEGO(默认值)

										A1 X[0]= 1:列地址127映射到SEGO
0 0	A8 A[5:0]	1 *	0 *	1 A5	0 A4	1 A3	0 A2	0 A1	0 A0	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	X3	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 *	0 A5	1 A4	0 A3	0 A2	1 A1	1 A0	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 *	0 A5	1 A4	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 A2	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，OLED）。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。例如英文字符“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	0x00
	0 0 0 0 0 0 0 0	0x00
	0 0 0 1 0 0 0 0	0x10
	0 0 1 1 1 0 0 0	0x38
	0 1 1 0 1 1 0 0	0x6c
	1 1 0 0 0 1 1 0	0xc6
	1 1 0 0 0 1 1 0	0xc6
	1 1 1 1 1 1 1 0	0xfe
	1 1 0 0 0 1 1 0	0xc6
	1 1 0 0 0 1 1 0	0xc6
	1 1 0 0 0 1 1 0	0xc6
	1 1 0 0 0 1 1 0	0xc6
	0 0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0 0	0x00
	0 0 0 0 0 0 0 0	0x00
	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 0 0 0 0 0 0	0x10, 0xa0
	0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0	0x10, 0x40

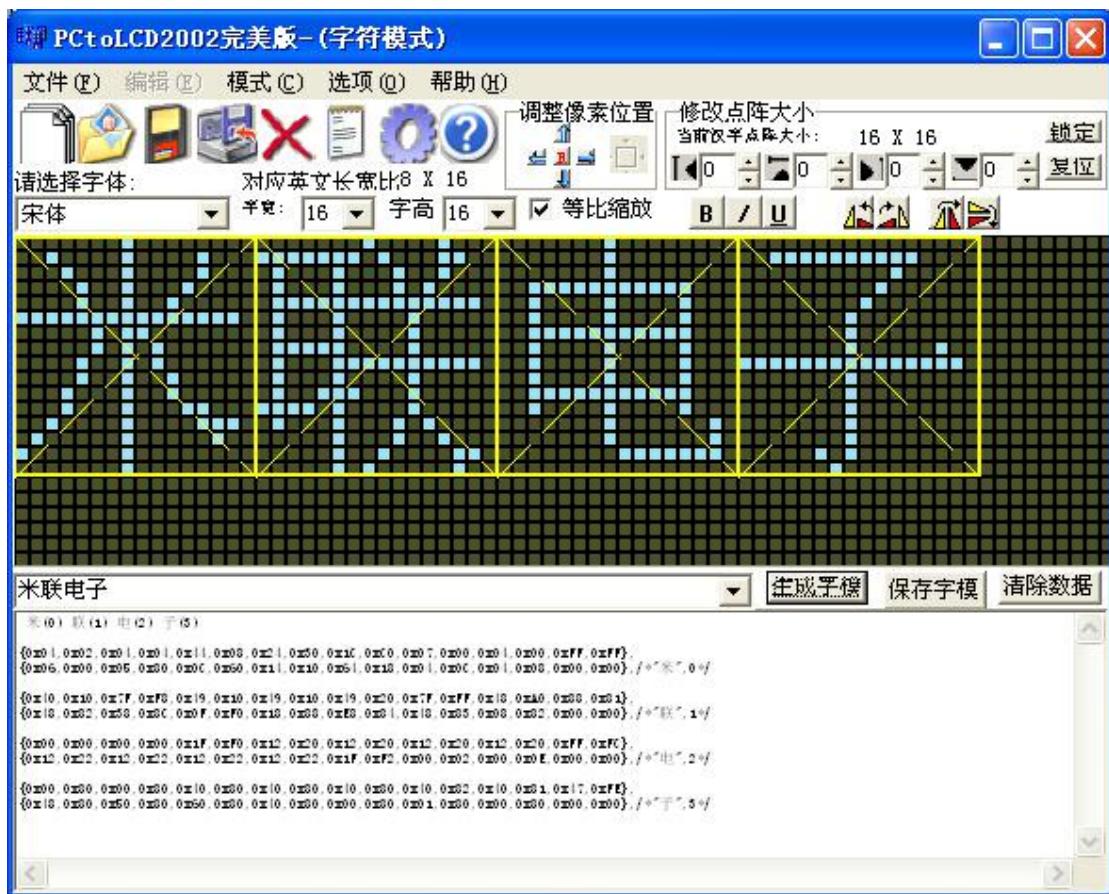
14.4.3 字模的获取

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：

```

font.h
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,0xF8,0x21,0x08,0x01,0x02,0x01,0x02,0x00,0x00,0x00,0x00},/*" 电 ",2*/
{0x01,0x00,0x01,0x00,0x01,0x00,0x3F,0xF8,0x21,0x08,0x21,0x08,0x3F,0xF8,0x21,0x08,
0x21,0x08,0x21,0x08,0x3F,0xF8,0x21,0x08,0x01,0x02,0x01,0x02,0x00,0x00,0x00,0x00},/*" 子 ",3*/
}

```

```
{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(" ");

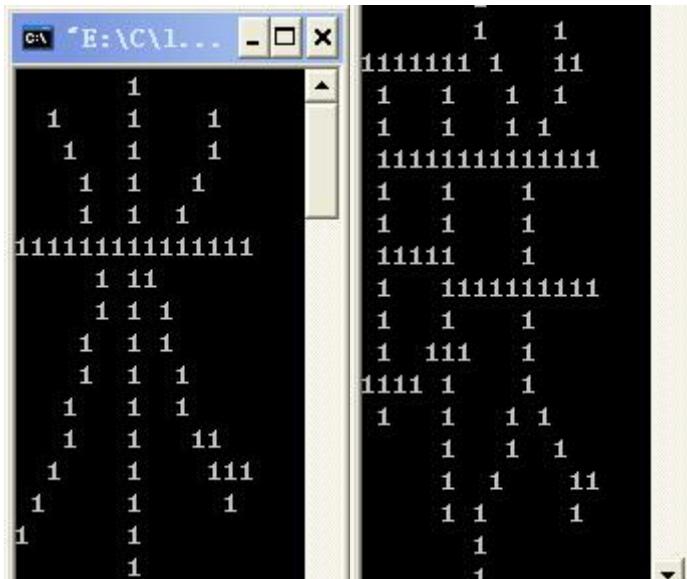
    ch_r = ch_r<<1;
}

// 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。

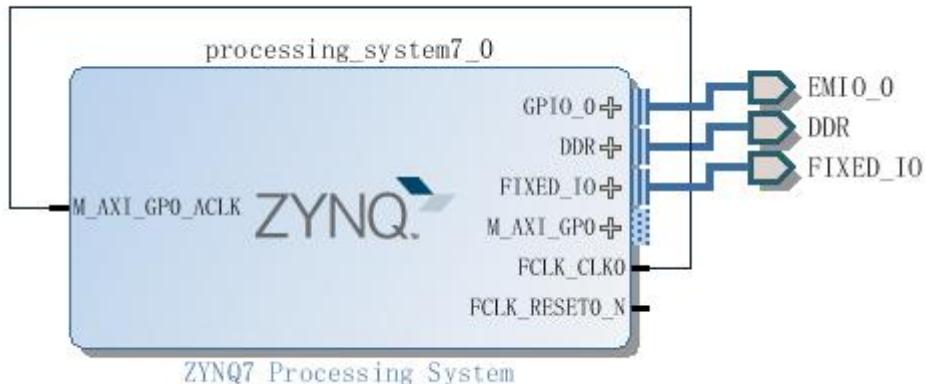
14.5 硬件搭建

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

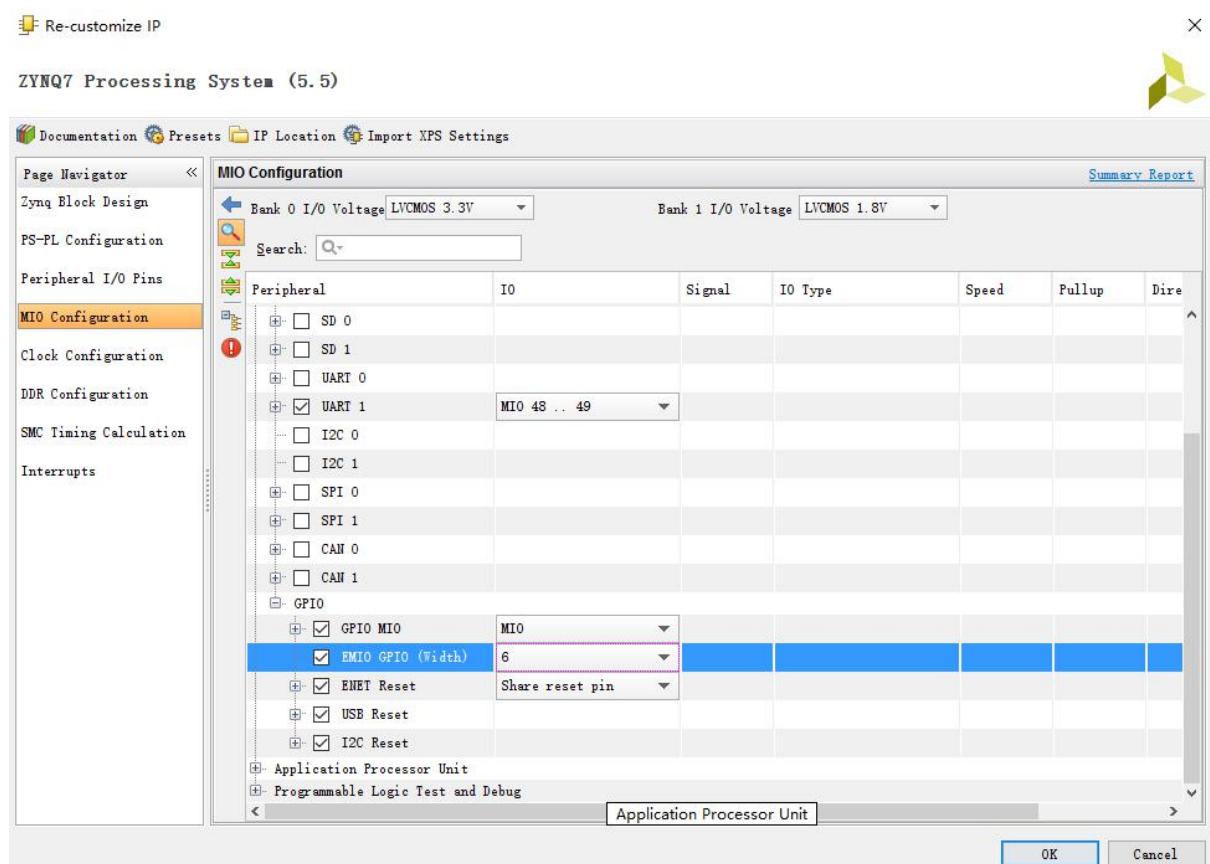
一些细微的修改即可。

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：修改约束文件，打开对应自己硬件的原理图，查看 OLED 部分引脚连接情况，此处以 Miz702 约束为例，其他型号用户对端口稍作修改即可。

```
#DC
set_property PACKAGE_PIN U10 [get_ports {emio_0_tri_io[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[0]}]
#RES
set_property PACKAGE_PIN U9 [get_ports {emio_0_tri_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[1]}]
#SCLK
set_property PACKAGE_PIN AB12 [get_ports {emio_0_tri_io[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[2]}]
#SDIN
set_property PACKAGE_PIN AA12 [get_ports {emio_0_tri_io[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[3]}]
#VBAT
set_property PACKAGE_PIN U11 [get_ports {emio_0_tri_io[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[4]}]
#VDD
set_property PACKAGE_PIN U12 [get_ports {emio_0_tri_io[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_0_tri_io[5]}]
```

Miz701N 用户因为只有四个 OLED 接口，但并不影响，只需要对应约束四个端口就可以了。

Step7:生成 bit 文件。

14.6 导入到 SDK

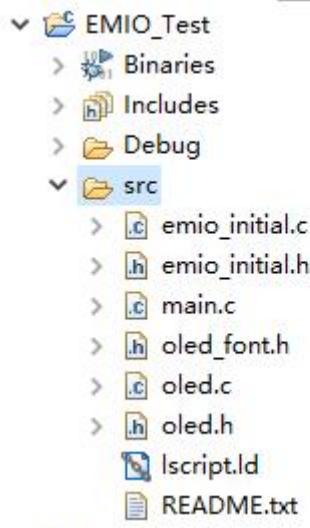
Step1:导出硬件。

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

Step3:打开我们提供的源程序包，在第二季，第 14 章的文件夹中，将 SDK 所有的文件复制过来。



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。

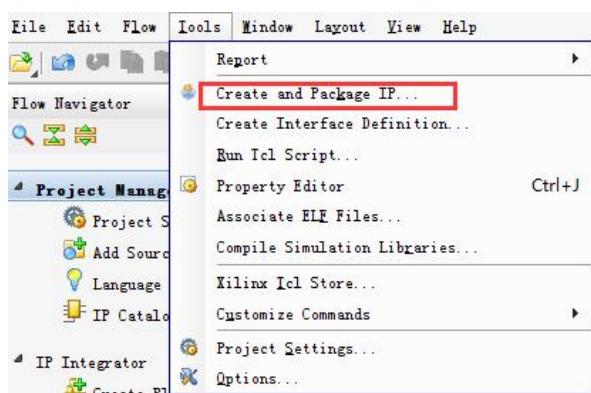
S02_CH15_AXI_OLED 实验

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。

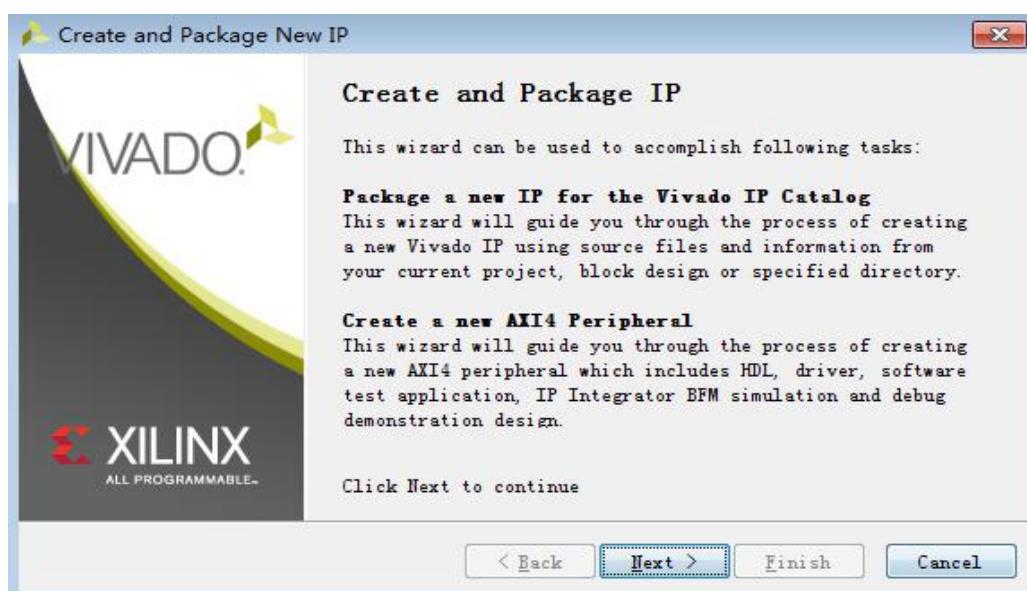
15.1 自定义 IP 的封装

Step1：新建一个名为 Miz_sys 空的工程。

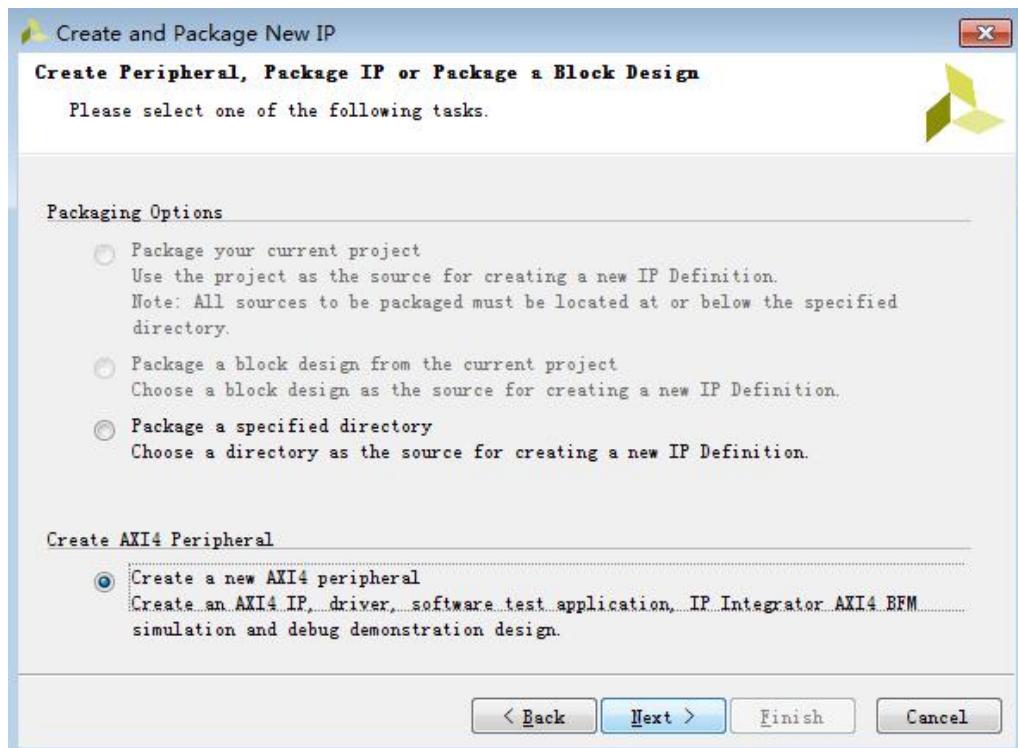
Step2：选择 Tools Create and Package IP 创建 IP



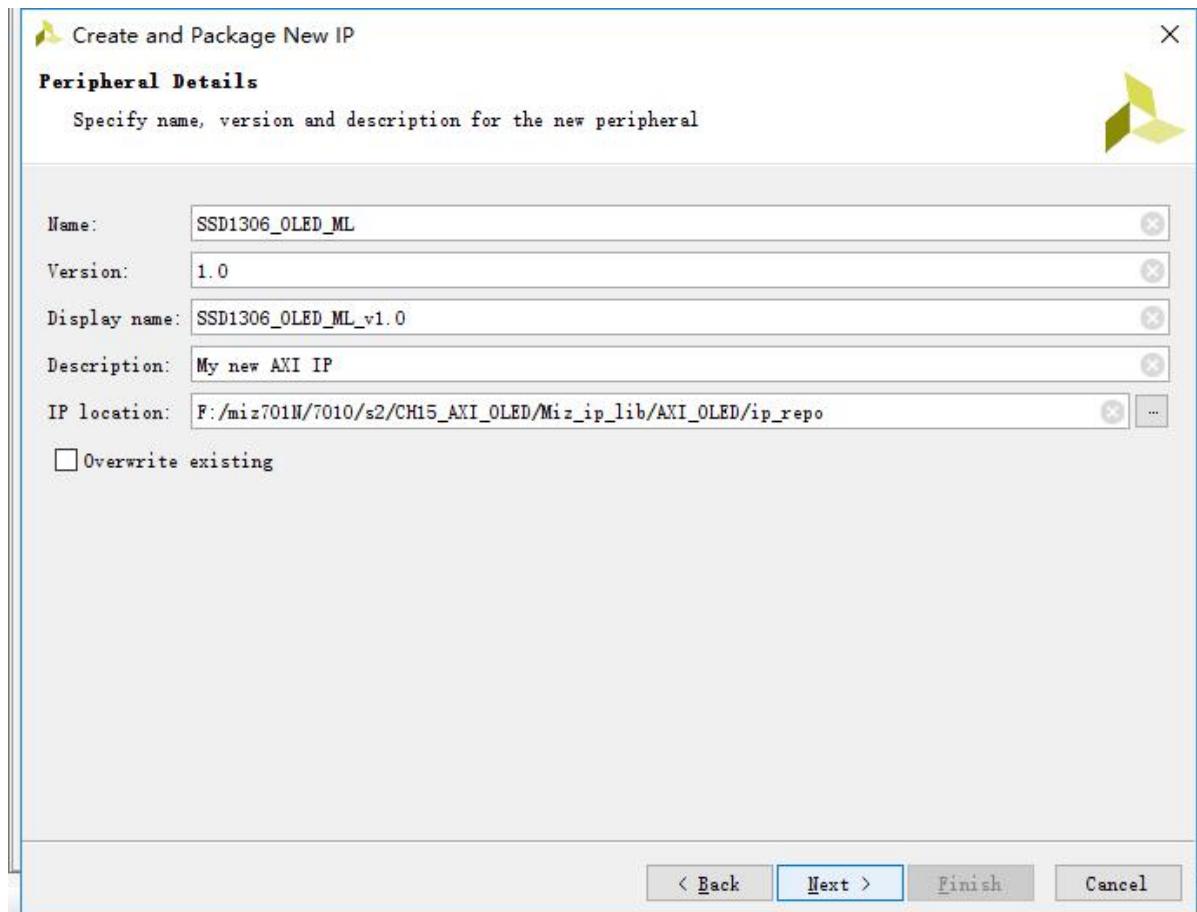
Step3:单击 NEXT



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

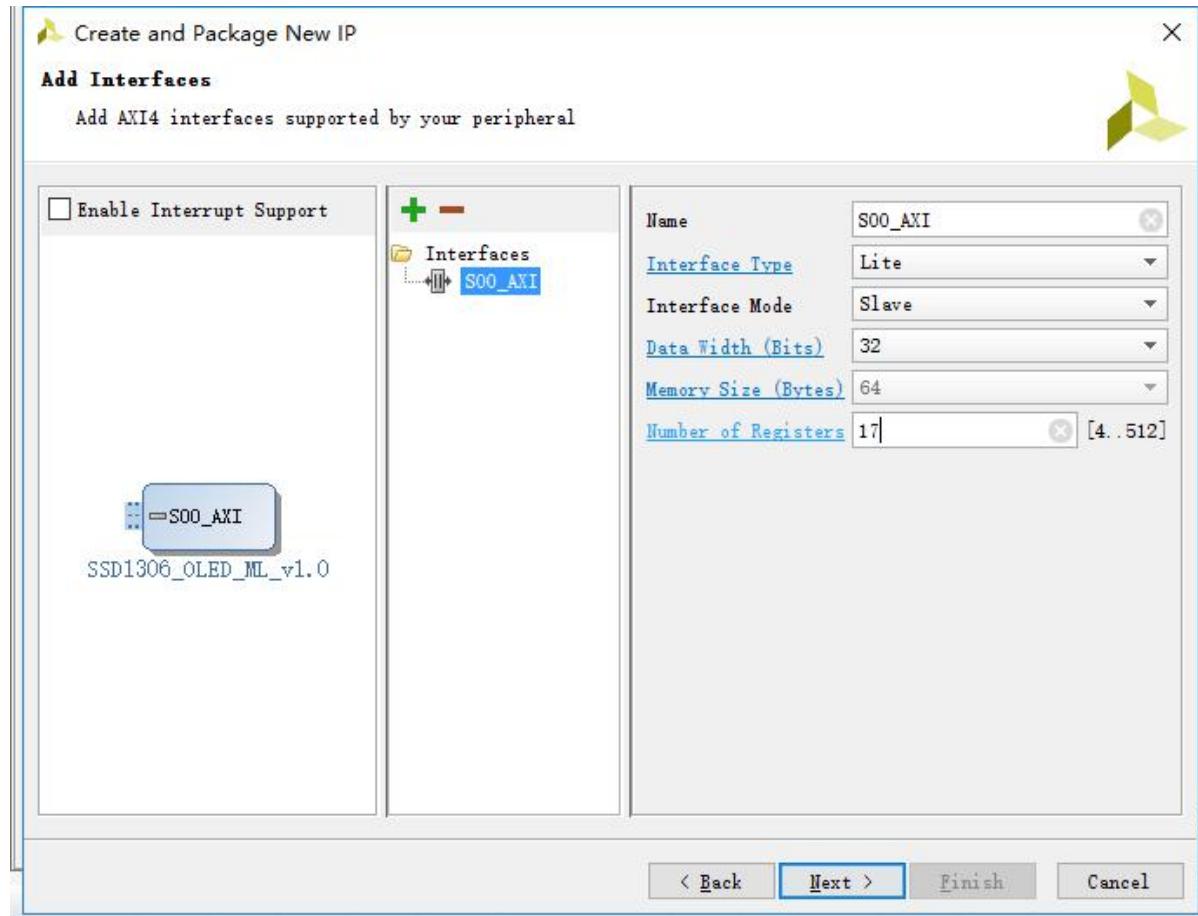


Step5:设置IP的名字为SSD1306_OLED_ML 版本号默认，并且记住IP的位置

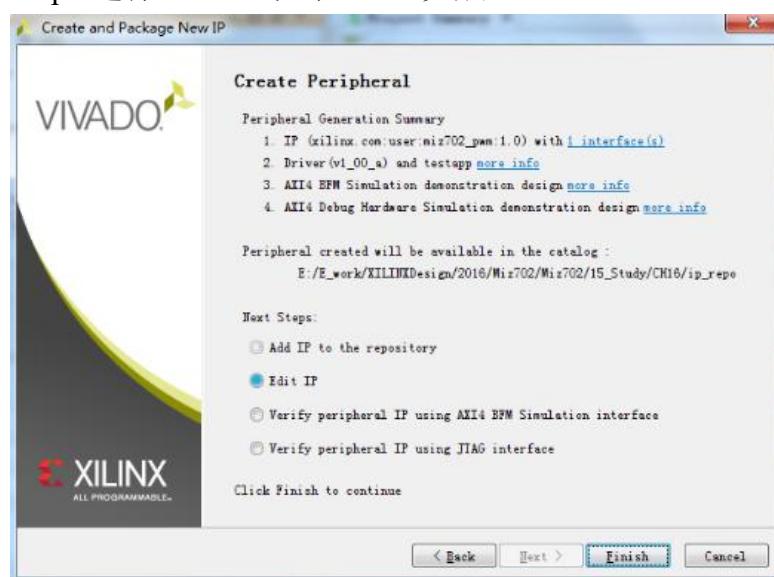


Step6:设置总线形式为 Lite 总线，Lite 总线是简化的 AXI 总线消耗的资源少，当然性能

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。



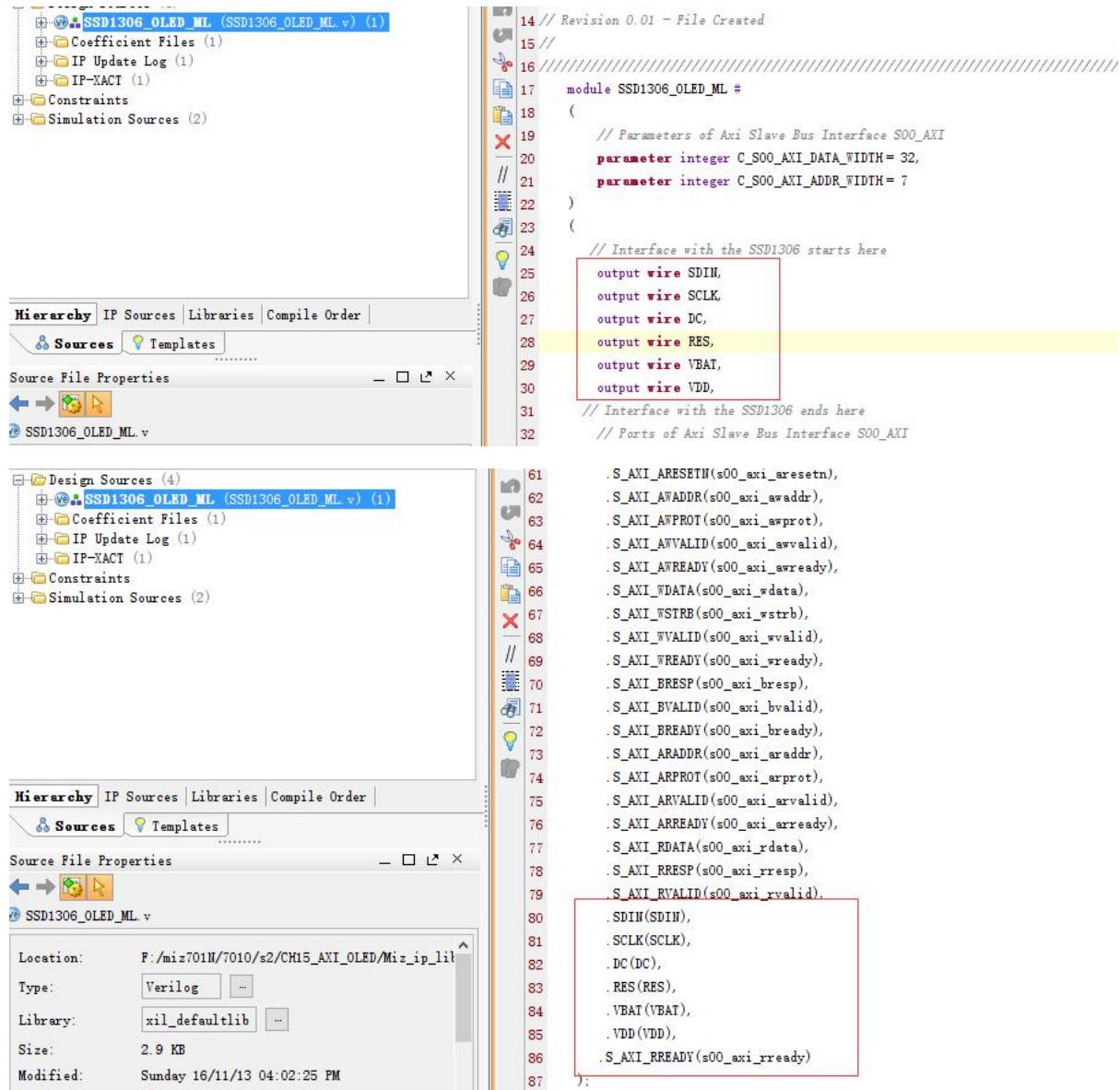
Step7:选择 Edit IP 单击 Finish 完成



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 14.2 (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, accepted 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_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.

//
===== 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
    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;  
    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
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

// Implement axi_already generation
// axi_already is asserted for one S_AXI_ACLK clock cycle when
// S_AXI_ARVALID is asserted. axi_already 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_rvalid 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
    axi_rdata <= reg_data_out;      // register read data
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),  
    .addr(a,temp_addr),  
    .dout(a,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

// 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'b000100010;
    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'b000000000;
    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
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:    ZedboardOLED
// Target Devices:  Zynq
// Tool versions:   Vivado 14.2 (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_BYTE, 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  
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'b000000000000000000; // 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  
        end  
    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

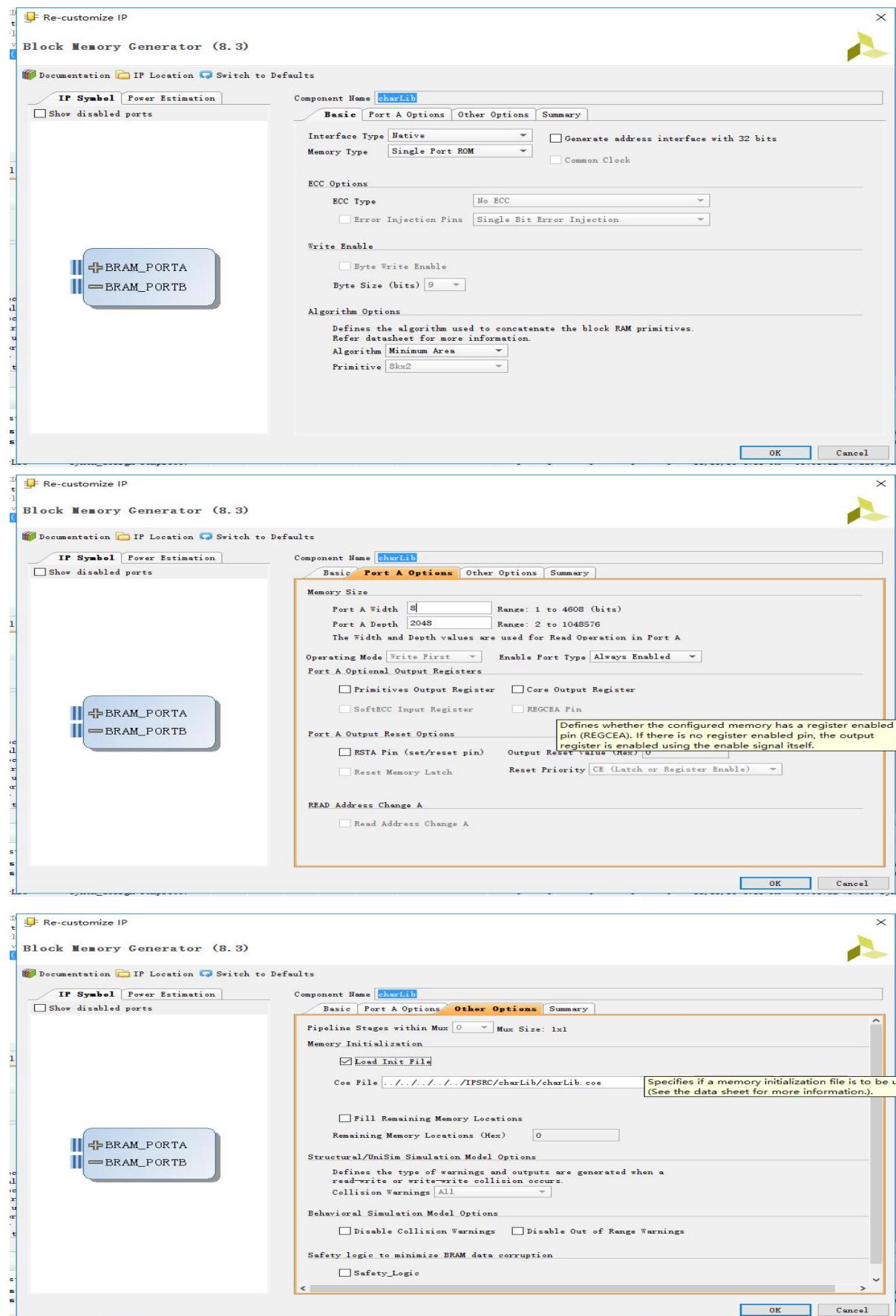
default : current_state <= "Idle";

endcase
end
end

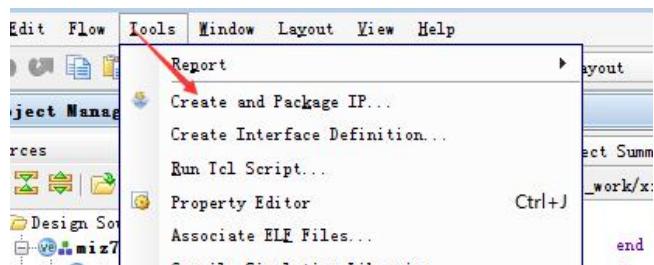
// End State Machine

// 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'b00000000000000000;
            ms_counter <= ms_counter + 1'b1;                  // increments at
1KHz
        end
    else begin
        clk_counter <= clk_counter + 1'b1;
    end
    end
    else begin
        // If not in the hold state reset counters
        clk_counter <= 17'b00000000000000000;
        ms_counter <= 12'h000;
    end
end
endmodule
```

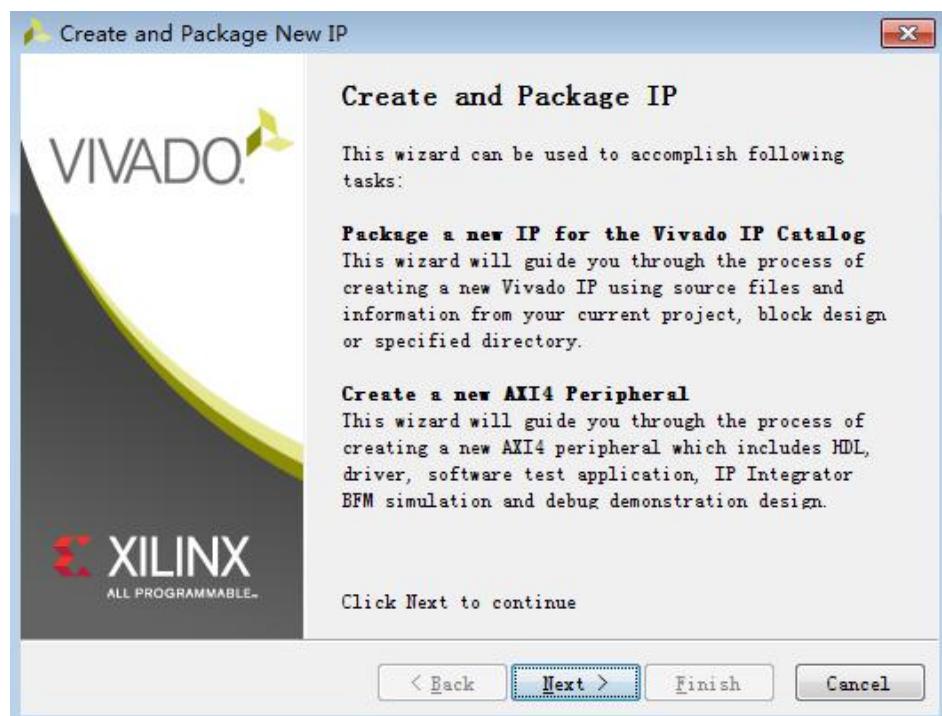
Step5:添加一个 Block ROM IP,按下图进行设置。ROM 的 coe 文件可在我们提供的源代码程序包中获得。



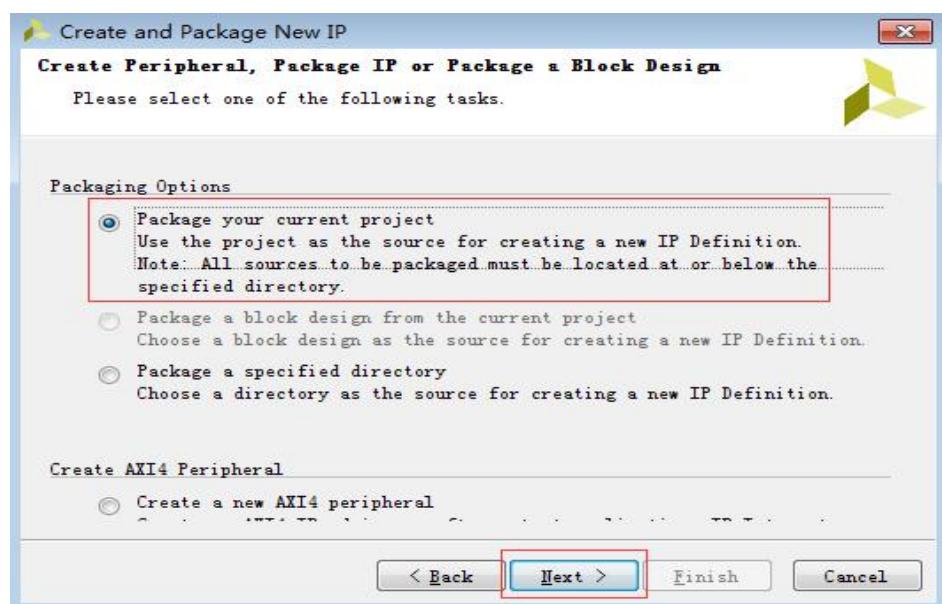
Step6:修改完成后重新封装一次自定义 IP



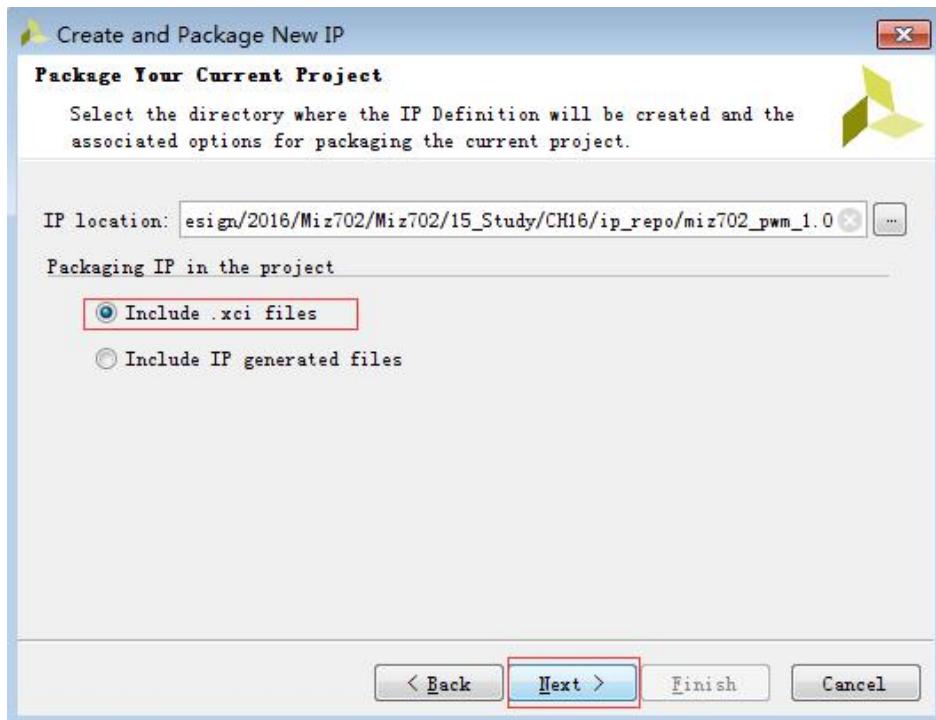
Step7:单击 NEXT



Step8:和第一次不同，这次选择第一个单选框然后单击 NEXT



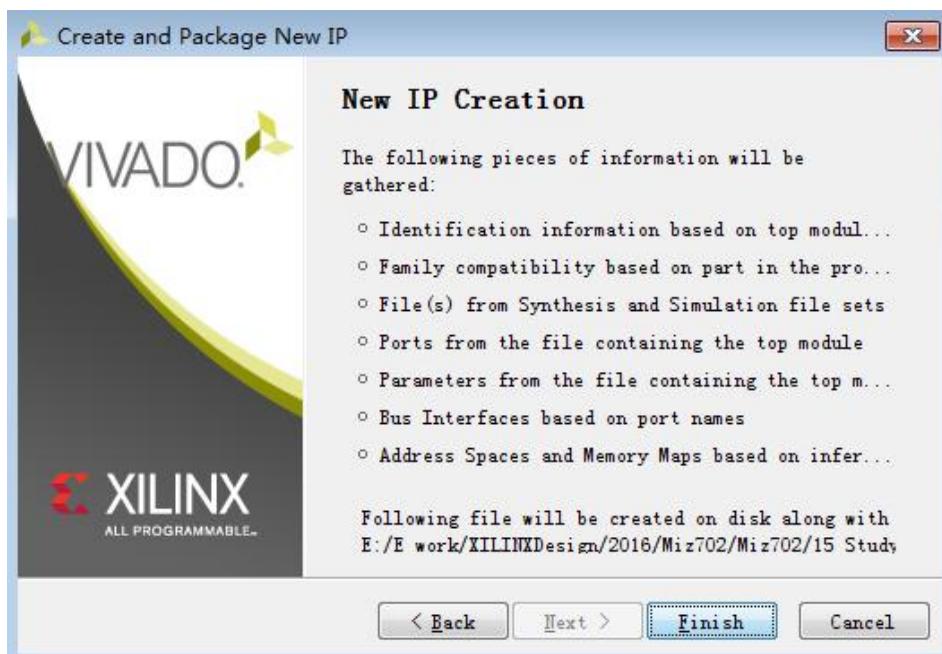
Step9:选择第一个单选框，然后单击 NEXT



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

            // 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; // 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 turning 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

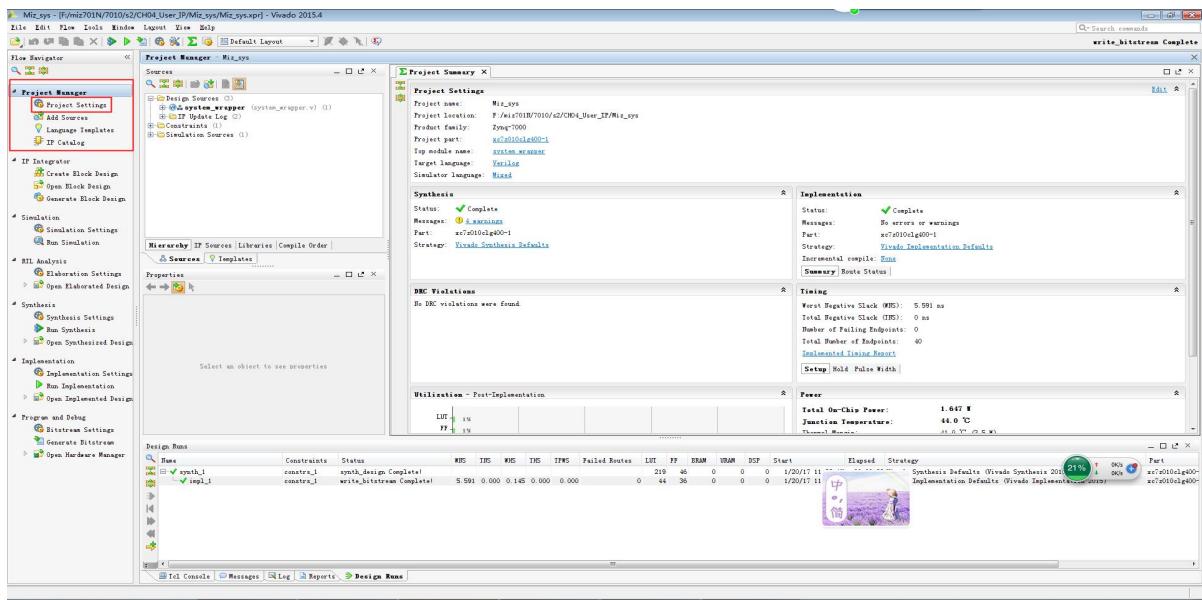
// Internal reset generator
always @(posedge S_AXI_ACLK) begin
if (RST_IN == 1'b1)
    count<=count+1'b1;
if (count == 12'hFFF) begin
    RST_internal <=1'b0;
end
end
```

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

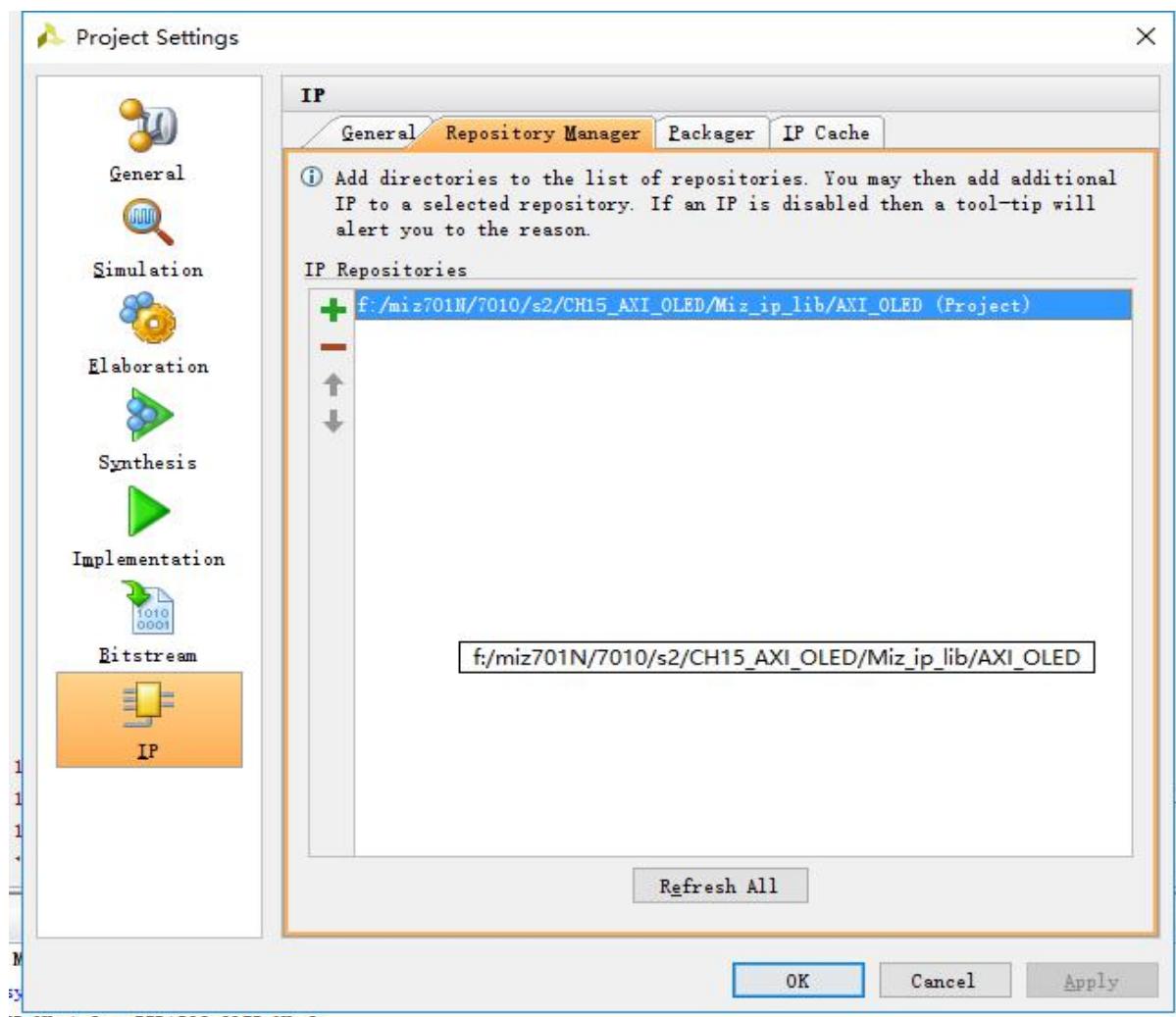
Step1：另外新建一个 VIVADO 工程，根据自己的开发板正确配置芯片型号。

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。

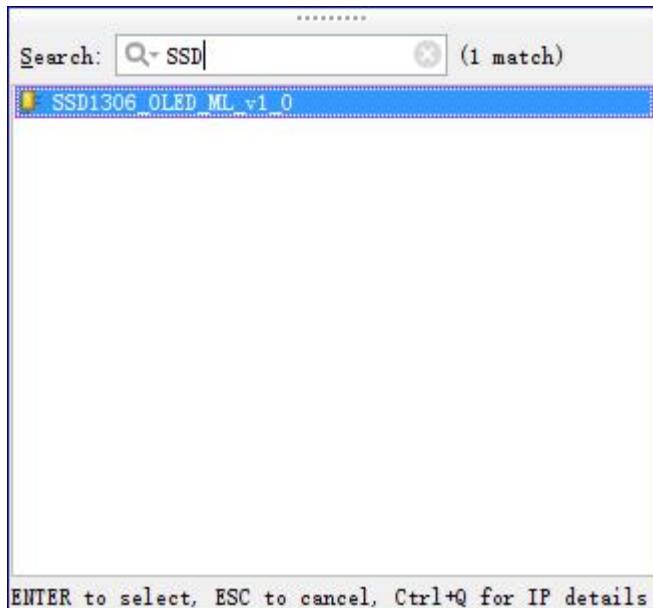
Step4：单击+号图标，将上一节封装的 IP 的路劲存放进去，单击 OK。



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：添加一个约束文件，打开对应自己硬件的原理图，查看 OLED 部分引脚连接情况。Miz702 约束文件如下所示：

```
set_property PACKAGE_PIN U10 [get_ports DC]
set_property PACKAGE_PIN U9 [get_ports RES]
set_property PACKAGE_PIN AB12 [get_ports SCLK]
set_property PACKAGE_PIN AA12 [get_ports SDIN]
set_property PACKAGE_PIN U11 [get_ports VBAT]
set_property PACKAGE_PIN U12 [get_ports VDD]
set_property IOSTANDARD LVCMOS33 [get_ports DC]
set_property IOSTANDARD LVCMOS33 [get_ports RES]
set_property IOSTANDARD LVCMOS33 [get_ports SCLK]
set_property IOSTANDARD LVCMOS33 [get_ports SDIN]
set_property IOSTANDARD LVCMOS33 [get_ports VBAT]
set_property IOSTANDARD LVCMOS33 [get_ports VDD]
```

```
set_property PACKAGE_PIN N17 [get_ports VDD]
```

其他型号开发板参照对应型号的原理图的 OLED 部分，修改成对应的引脚即可。

Step13：生成 bit 文件。

15.5 导入到 SDK

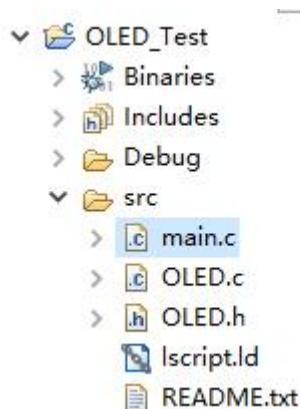
Step1:导出硬件。

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，只能够显示字符；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。

S02_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等精度频率计的原理。

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。

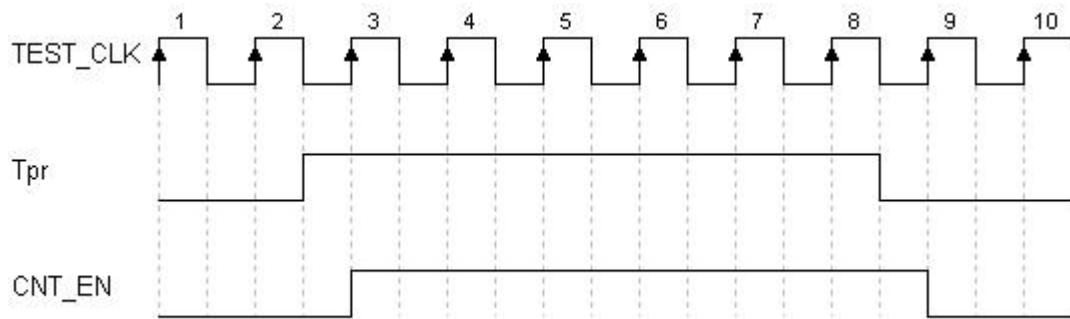


图 1 Tpr 处理后成为 CNT_EN

由于引入了 D 触发器，CNT_EN 不会在 Tpr 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。

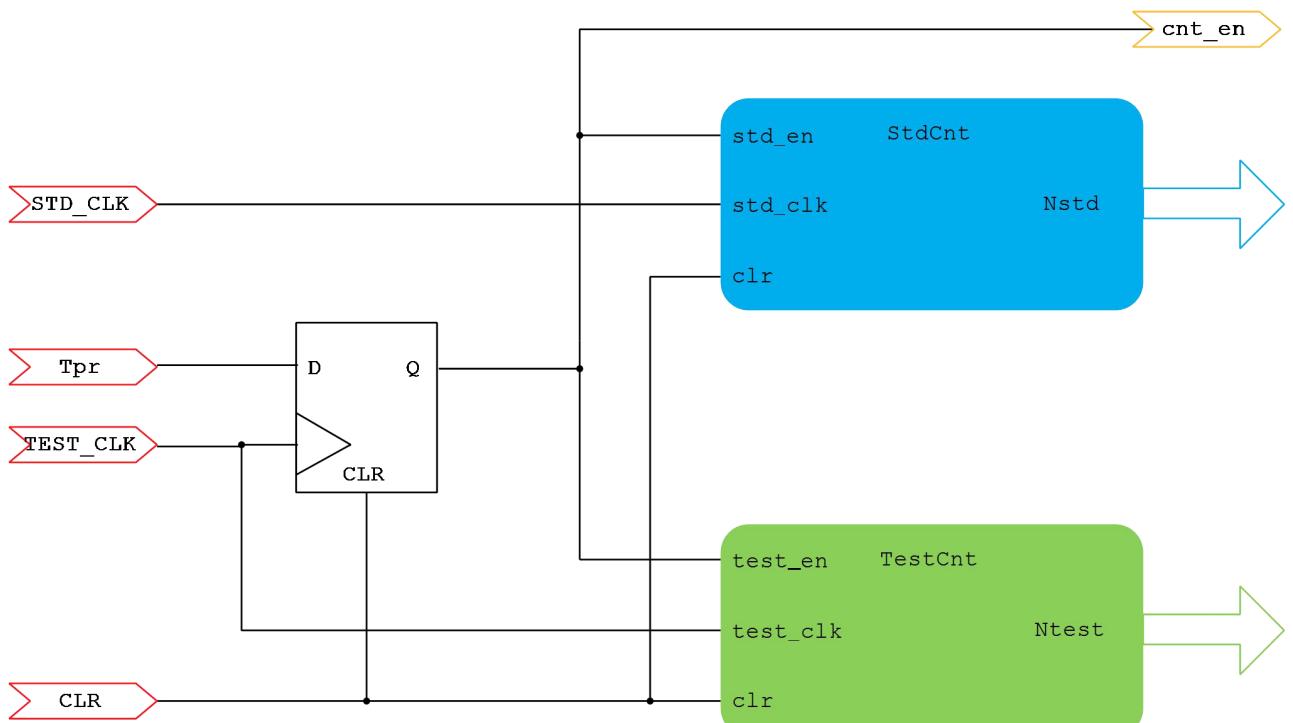


图 2 测频主控模块结构图

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）。

$$N_{std}/F_{std}=N_{test}/F_{test}$$

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 $F_{test}=F_{std} \cdot N_{test}/N_{std}$ 。

由于未对标准时钟进行同步计数，所以测量结果会产生 ± 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 T_{pr} 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 T_{pr} (高时 打开闸门)
Bit0	复位/清零信号 clr (低有效)

reg1：数据寄存器 N_{std} (offset: 0x04)

Bit	功能
Bit31~bit0	标准时钟计数值

reg2：数据寄存器 N_{test} (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 Miz 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^{-8} 。

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

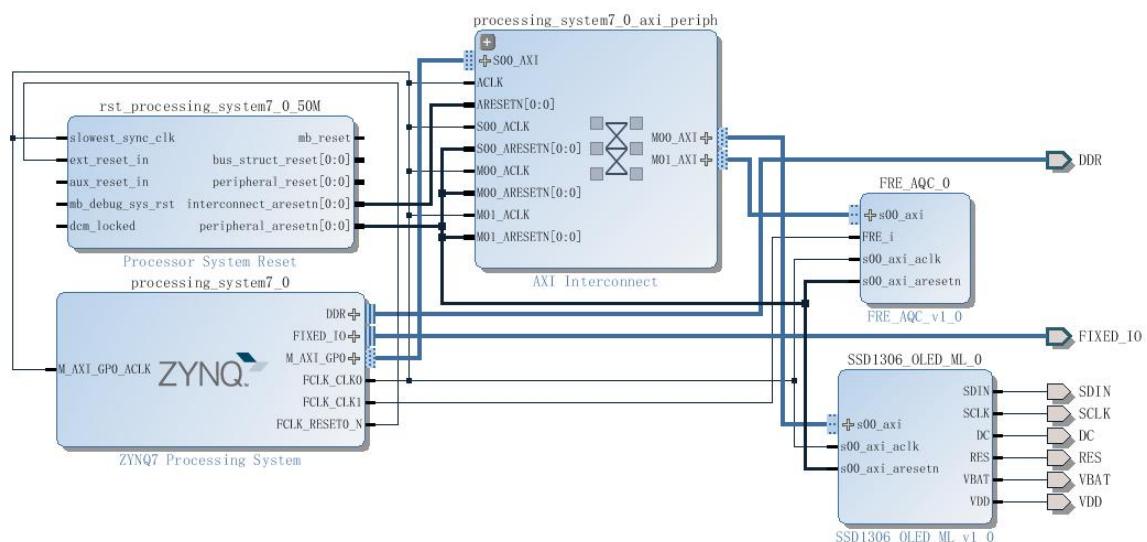
// User logic ends

```

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=%lfMHZ",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=%lfMHZ",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。

【第三季】ZYNQ SOC 裸奔应用方案共 18 课时

第三季攻击 13 课时详细讲解 ZYNQ DMA IP 、 VDMA IP 基于摄像头的采集方案，以及基于以太网的数据通信方案，最后讲解了裸机下双核运行机制。

讲解的内容由浅入深，循序渐进，比如第一课《S03_CH01_AXI_DMA_LOOP 环路测试》只是讲解了 DMA 的 LOOP 传输，《S03_CH02_AXI_DMA PL 发送数据到 PS》在第一课基础上稍微增加难度，实现了 PL 通过 DMA 的方式，高速批量地把数据搬运到 DDR 。《S03_CH03_AXI_DMA_OV7725 摄像头采集系统》就是实现了基于 DMA 的图形显示系统。《S03_CH09_DMA_4_Video_Switch 视频切换系统》和《S03_CH10_DMA_4_Video_Stitch 视频拼接系统》分别利用 DMA 的方式实现了思路视频的切换系统和拼接系统。《S03_CH08_DMA_LWIP 以太网传输》是一个基于 DMA 批量传输，实现了 PL 往 PS 发送数据，再通过以太网接口发送出去的方案。

另外本周还讲解了视频专用 IP VDMA IP 的寄存器分析，时序分析，并且实现了视频采集方案。《S03_CH13_ZYNQ_A9_TCP_UART 双核 AMP 例程》在本季课程最后，解了双核裸机运行的方案。

S03_CH01_AXI_DMA_LOOP 环路测试

1.1 概述

本课程是本季课程里面最简单，也是后面 DMA 课程的基础，读者务必认真先阅读和学习。

本课程的设计原理分析。

本课程是设计一个最基本的 DMA 环路，实现 DMA 的环路测试，在 SDK 里面发送数据到 DMA 然后 DMA 在把数据发回到 DDR 里面，SDK 读取内存地址里面的数据，对比接收的数据是否和发送出去的一致。DMA 的接口部分使用了 data_fifo IP 链接。本课程会详细介绍创建工程的每个步骤，后面的课程将不再详细介绍创建工程的步骤。

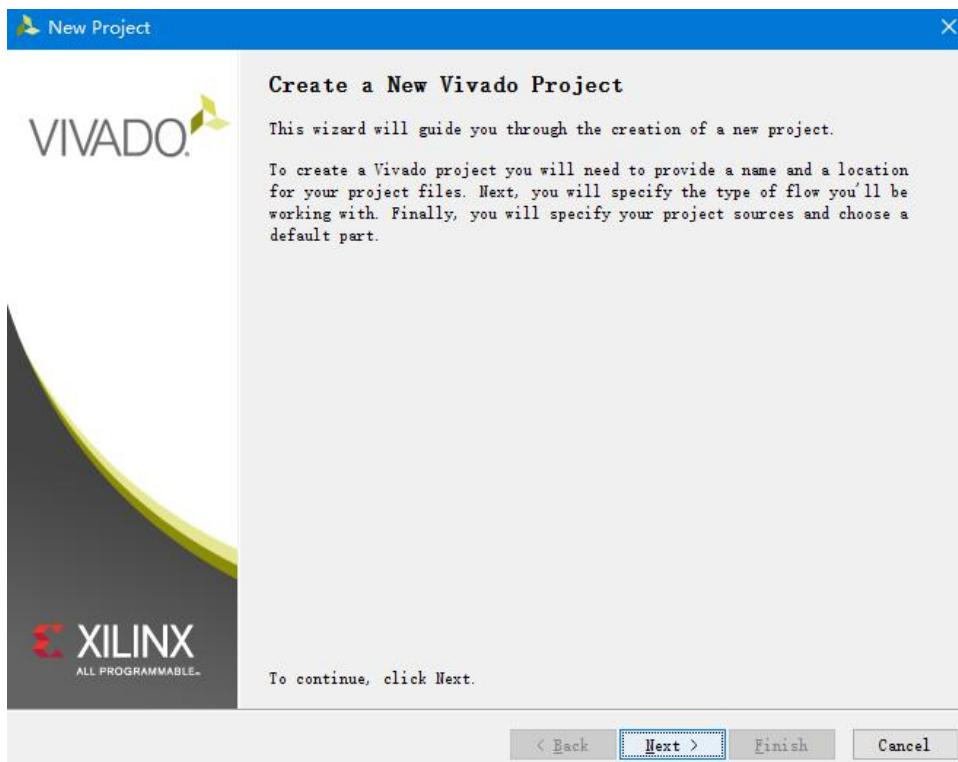
1.2 搭建硬件系统

1.2.1 新建 VIVADO 工程

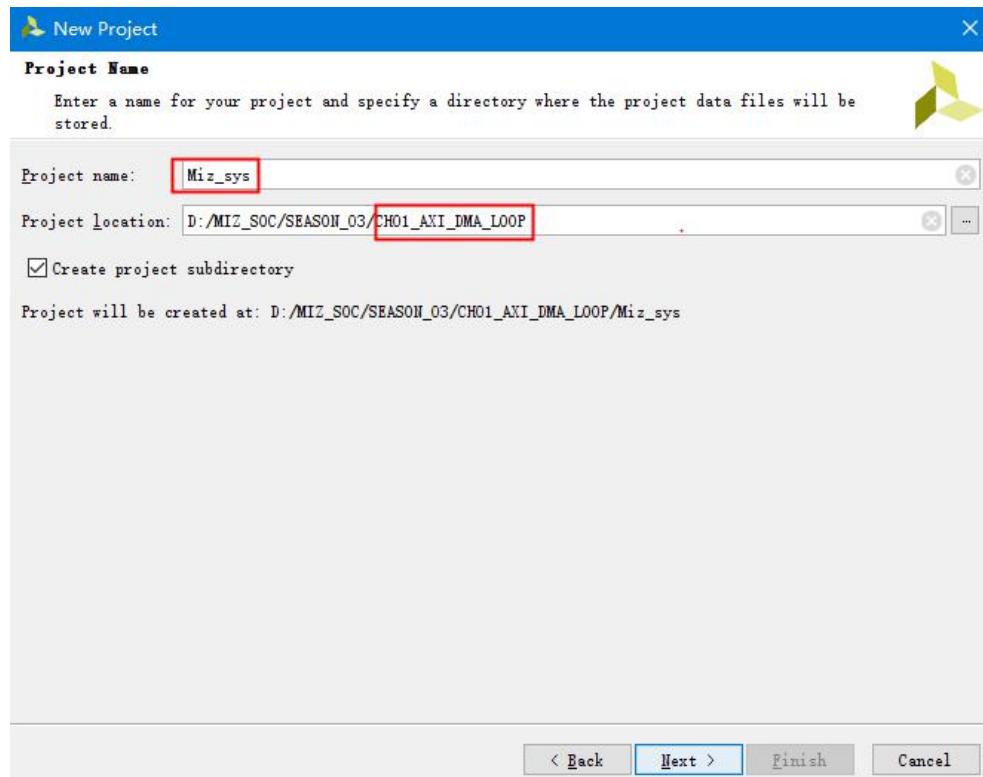
Step1:启动 VIVADO，单击 Create New Project



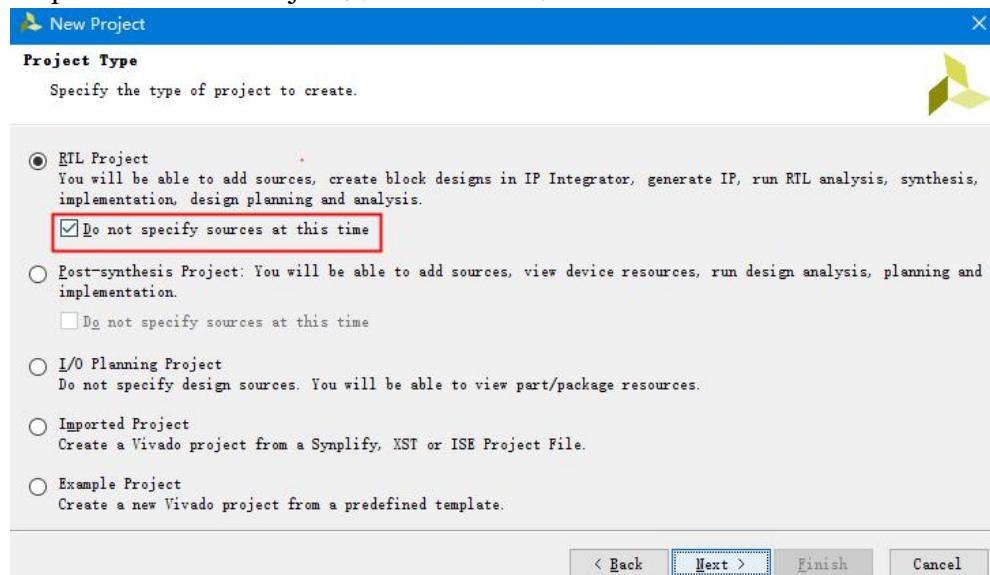
Step2:单击 NEXT



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



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

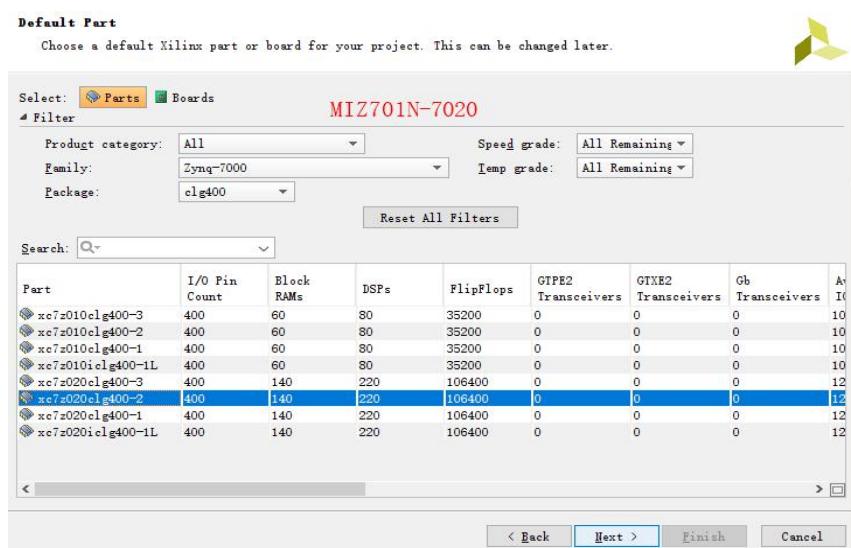
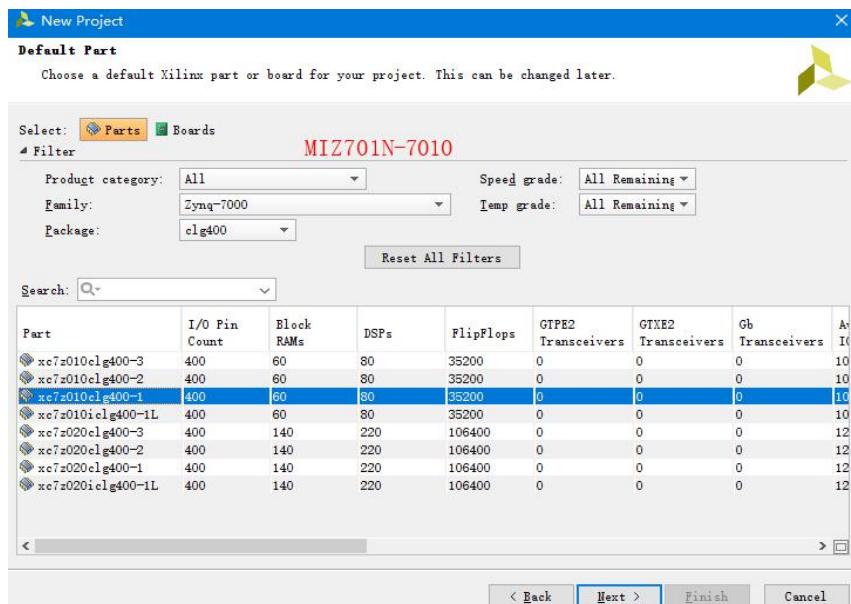
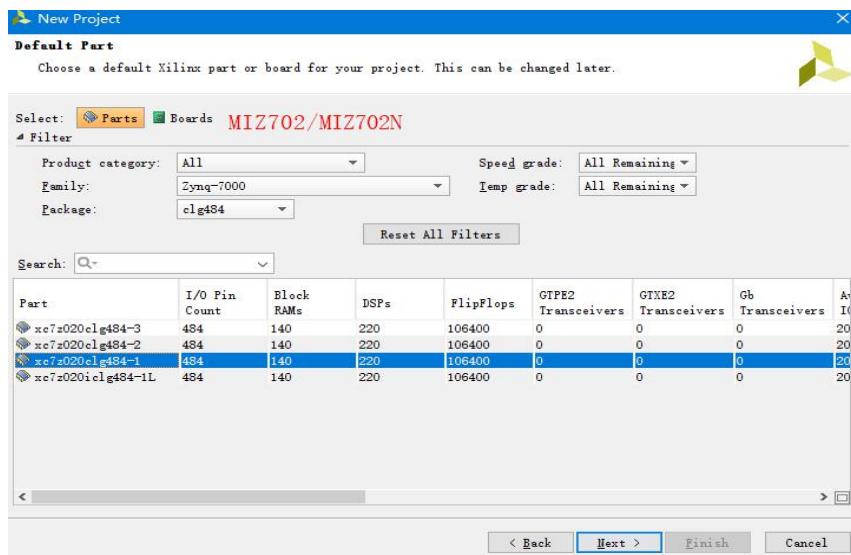


Step5: 选择芯片的型号和封装速度等级。

MIZ702/MIZ702N 选择 Zynq-7000-xc7z020clg484-1

MIZ701N-7010 选择 Zynq-7000-xc7z010clg400-1

MIZ701N-7020 选择 Zynq-7000-xc7z020clg400-2



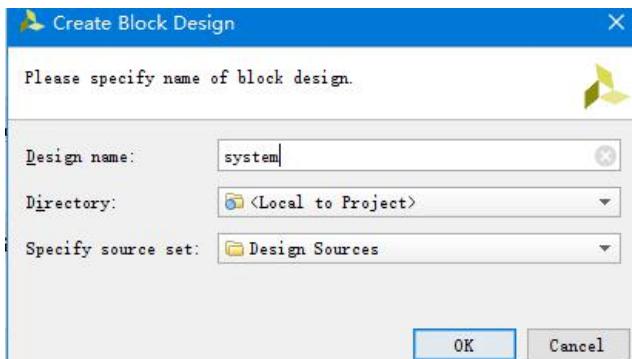
Step:6 单机 Finish 完成工程创建。

1.2.2 创建 VIVADO 硬件构架

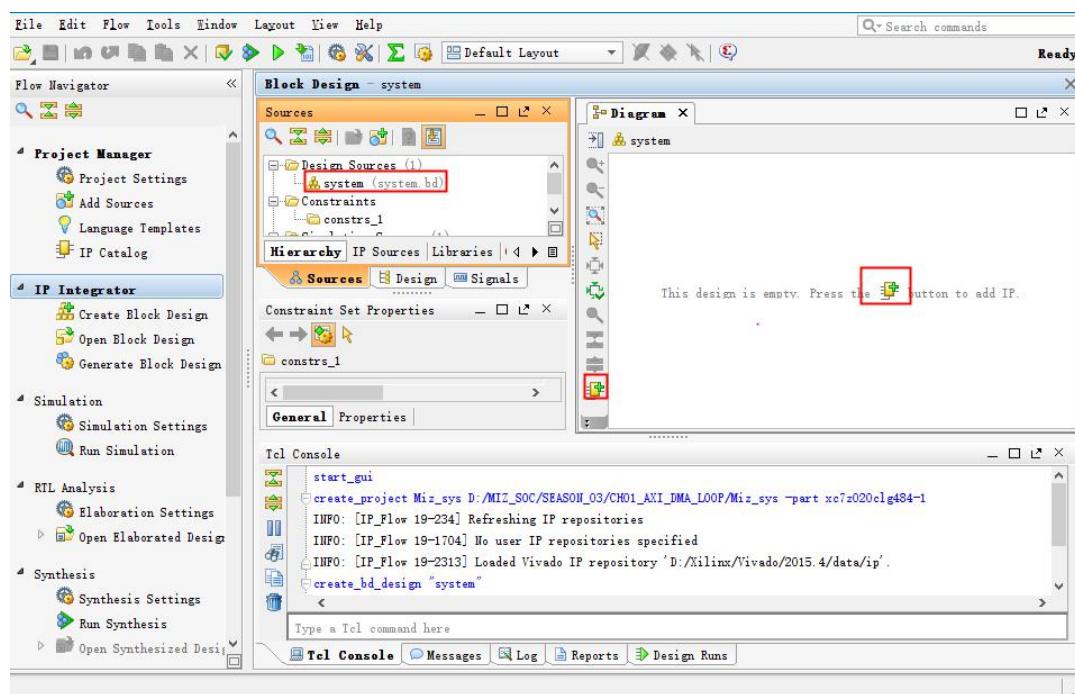
Step1:单击 Create Block Design



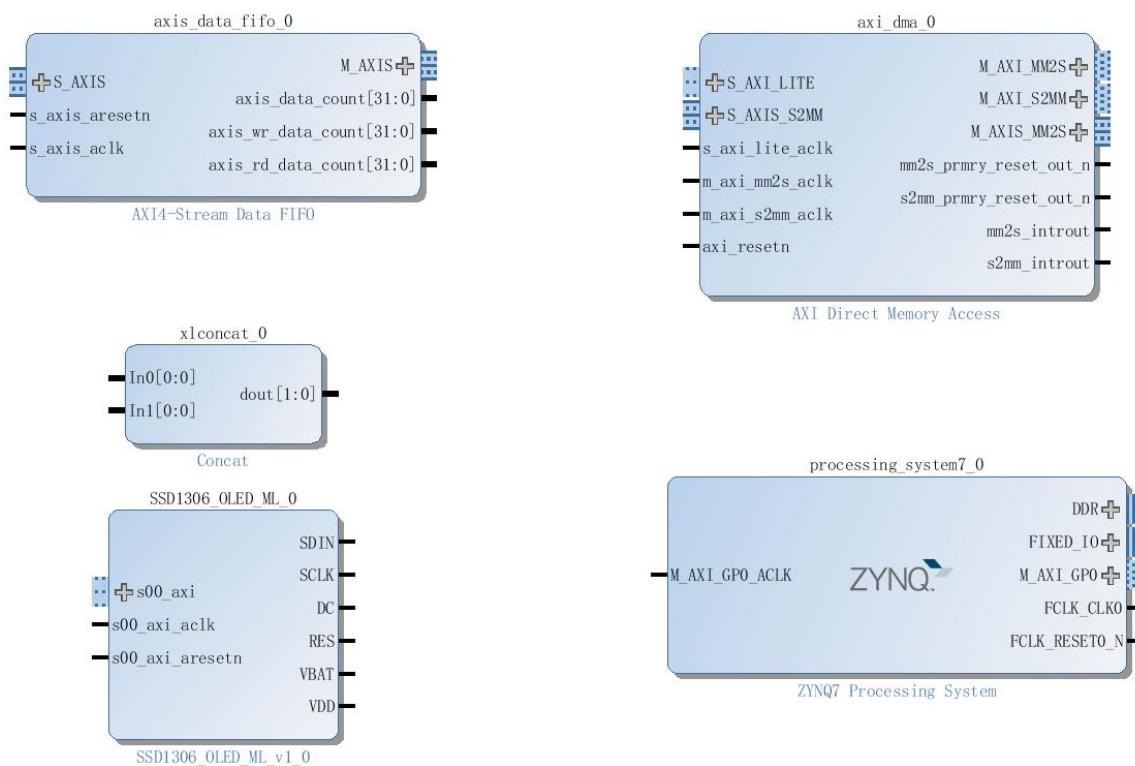
Step2：命名为 system 之后单击 OK



Step3： 创建完成后如下图所示

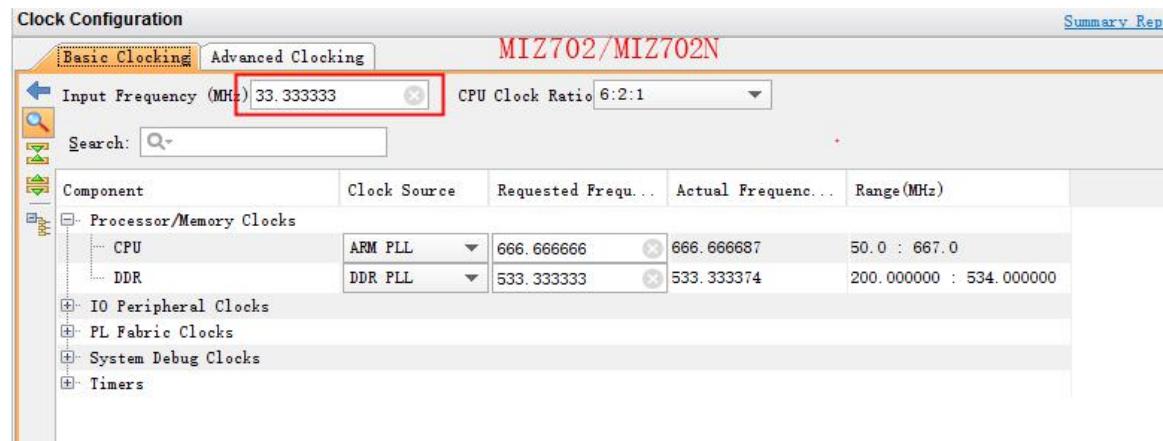


Step3：添加各个模块如图：

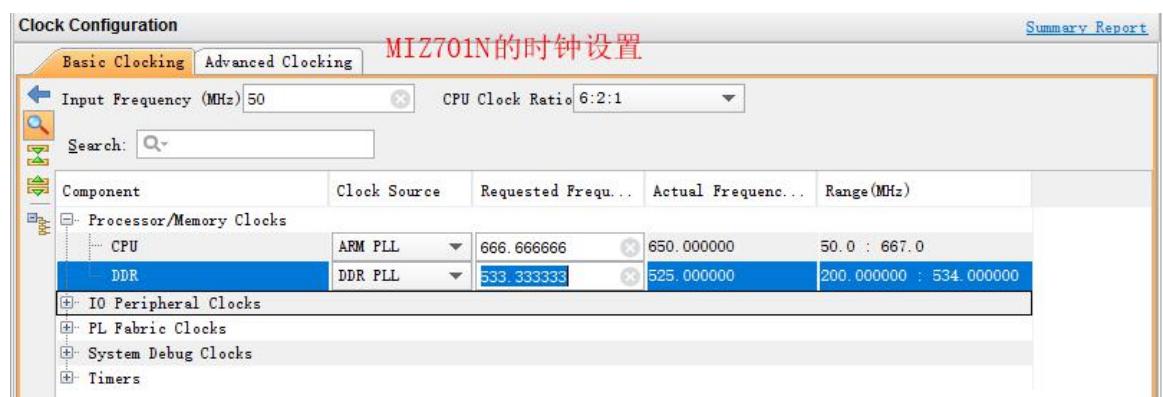


Step4 双击 ZYNQ IP 进行如下步骤配置

Step5: MIZ702 和 MIZ702N 的输入时钟是 333.333333MHZ



Step6: MIZ701N PS 的输入时钟是 50MHZ



Step7: MIZ702 的开发板采用的是单片 256MB 的 MT41K128M16JI-125

Name	Select	Description
Memory Type	DDR 3	Type of memory interface. Refer to UG585 Zynq Technical Ref
Memory Part	MT41K128M16 JI-125	Memory component part number. For unlisted parts choose "Cu
Effective DRAM Bus Width	32 Bit	Data width of DDR interface, not including ECC data width.
ECC	Disabled	Enables error correction code support. ECC is supported onl
Burst Length	8	Minimum number of data beats the controller should use whe
DDR	533.333333	Memory clock period. The allowed freq range is (200.000000
Internal Vref	<input type="checkbox"/>	Enables internal voltage reference source. Disable to use e
Operating Temperature (C)	Normal (0-85)	Intended operating temperature range. Controls the DDR ref

Name	Select	Description
Memory Type	DDR 3	Type of memory interface. Refer to UG585 Zynq Technical Ref
Memory Part	MT41K256M16 RE-125	Memory component part number. For unlisted parts choose "Cu
Effective DRAM Bus Width	32 Bit	Data width of DDR interface, not including ECC data width.
ECC	Disabled	Enables error correction code support. ECC is supported onl
Burst Length	8	Minimum number of data beats the controller should use whe
DDR	533.333333	Memory clock period. The allowed freq range is (200.000000
Internal Vref	<input type="checkbox"/>	Enables internal voltage reference source. Disable to use e
Operating Temperature (C)	Normal (0-85)	Intended operating temperature range. Controls the DDR ref

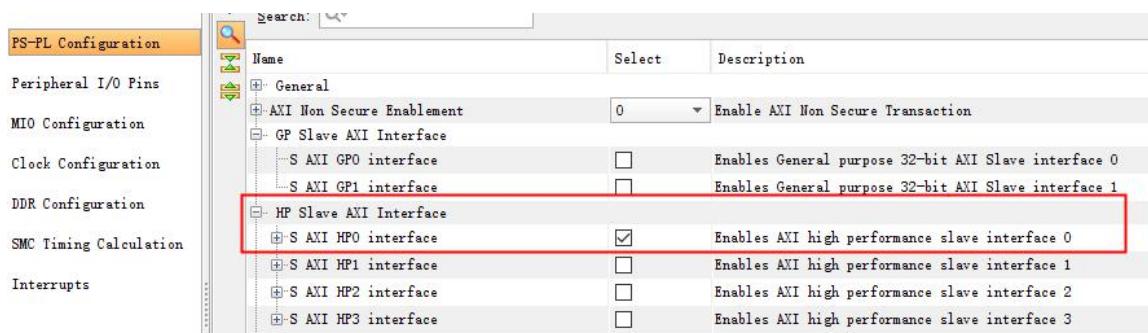
Step8: MIZ701N 和 MIZ702N 的内存型号一样，都是单片 512MB 的 MT41K256M16RE-125

Name	Select	Description
Memory Type	DDR 3	Type of memory interface. Refer to UG585 Zynq Technical Ref
Memory Part	MT41K256M16 RE-125	Memory component part number. For unlisted parts choose "Cu
Effective DRAM Bus Width	32 Bit	Data width of DDR interface, not including ECC data width.
ECC	Disabled	Enables error correction code support. ECC is supported onl
Burst Length	8	Minimum number of data beats the controller should use whe
DDR	533.333333	Memory clock period. The allowed freq range is (200.000000
Internal Vref	<input type="checkbox"/>	Enables internal voltage reference source. Disable to use e
Operating Temperature (C)	Normal (0-85)	Intended operating temperature range. Controls the DDR ref

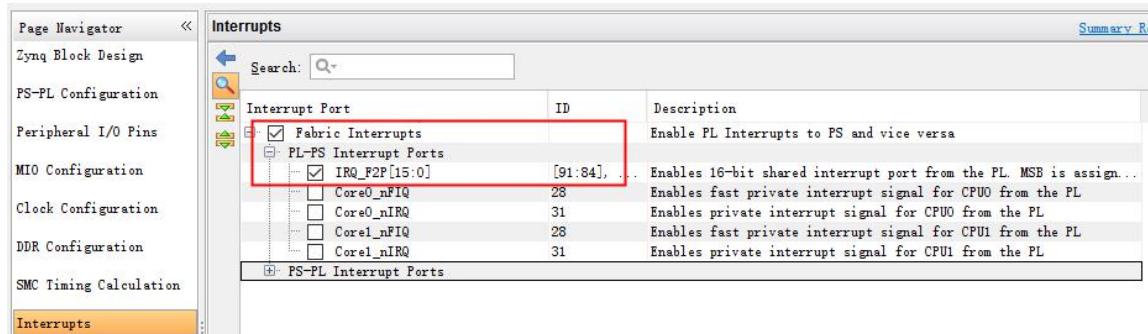
Step9: PS 的 PLL 提供本系统的时钟 100MHz

Component	Clock Source	Requested Freq...	Actual Freqenc...	Range(MHz)
Processor/Memory Clocks				
IO Peripheral Clocks				
PL Fabric Clocks				
FCLK_CLK0	IO PLL	100	100.000000	0.100000 : 250.000000
FCLK_CLK1	IO PLL	50	50.000000	0.100000 : 250.000000
FCLK_CLK2	IO PLL	50	50.000000	0.100000 : 250.000000
FCLK_CLK3	IO PLL	50	50.000000	0.100000 : 250.000000

Step10: 启动 1 路 HP 接口，HP 接口是 ZYNQ 个高速数据接口



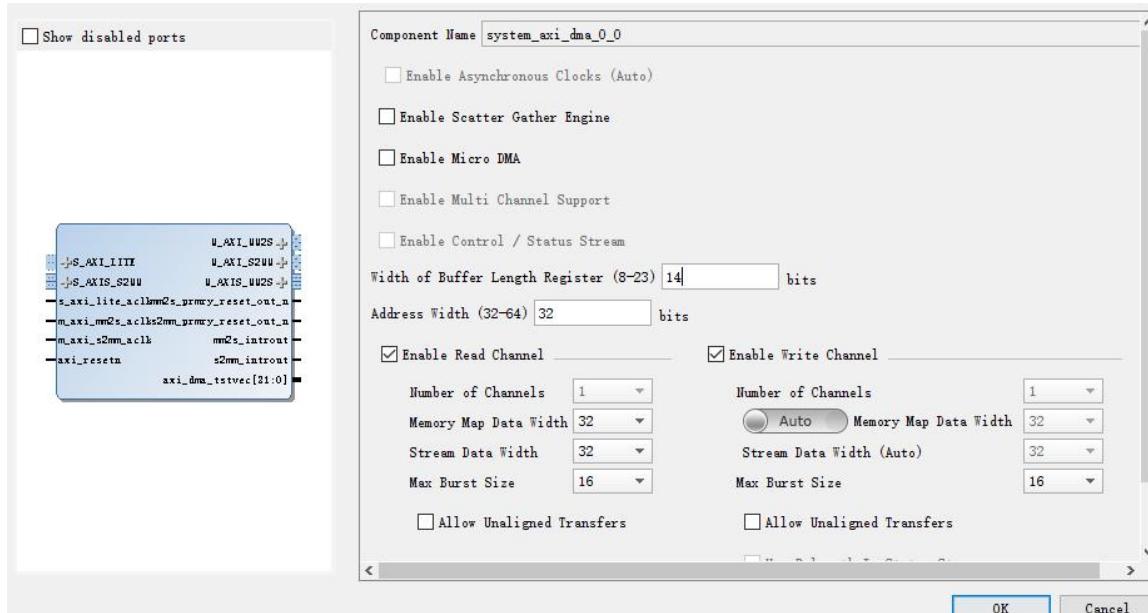
Step11:勾选 PL 到 PS 的中断资源（关于中断，在第二季的课程中有详细讲解，不熟悉的读者可以到第二季课程中温习一下）



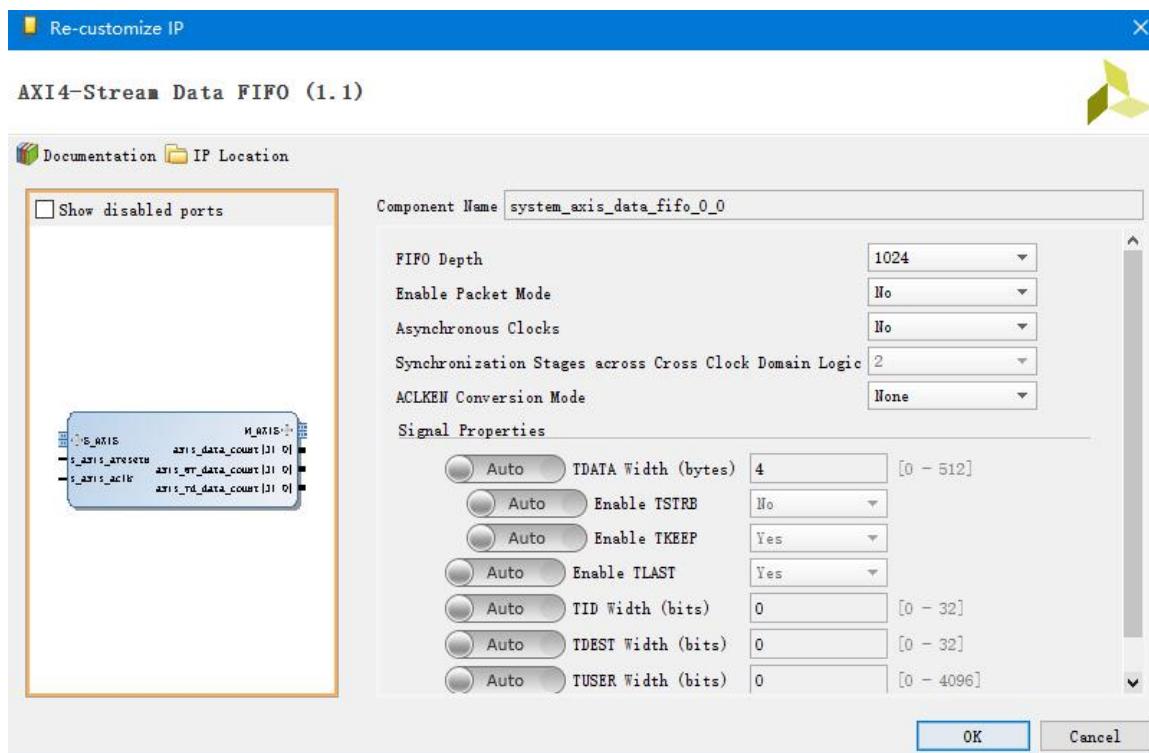
Step12:设置完成后单击 OK

Step13:双击 DMA IP 设置如下：

下图中，同时勾选读通道和写通道，另外设置，Width of buffer length register 为 23bit 这个含义是 2 的 23 次方 8,388,607bytes 8M 大小，这里设置 14bit 就够用了，长度越大需要的资源也就越多。

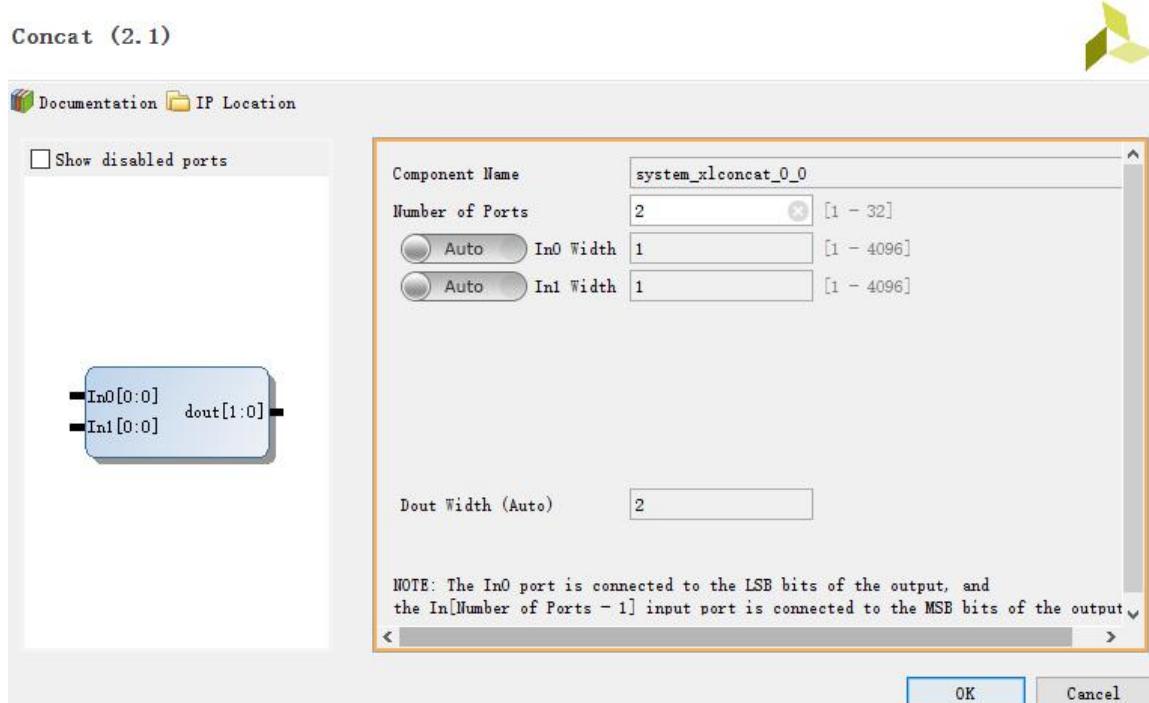


Step14:Data FIFO 设置

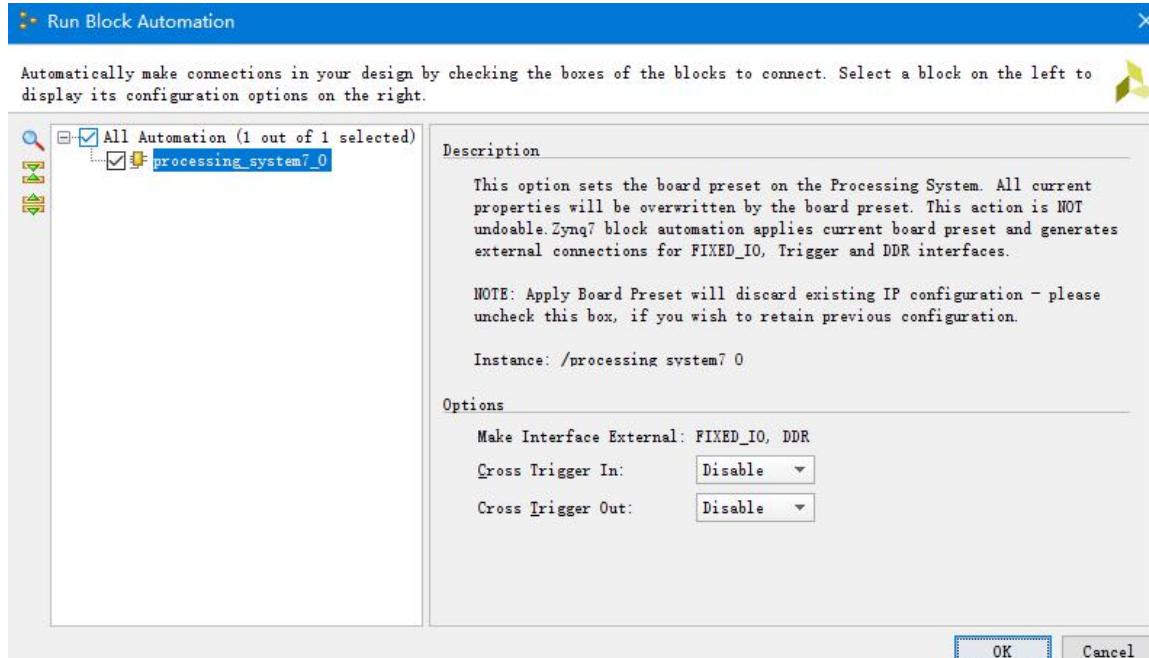


Step15:Concat IP 设置

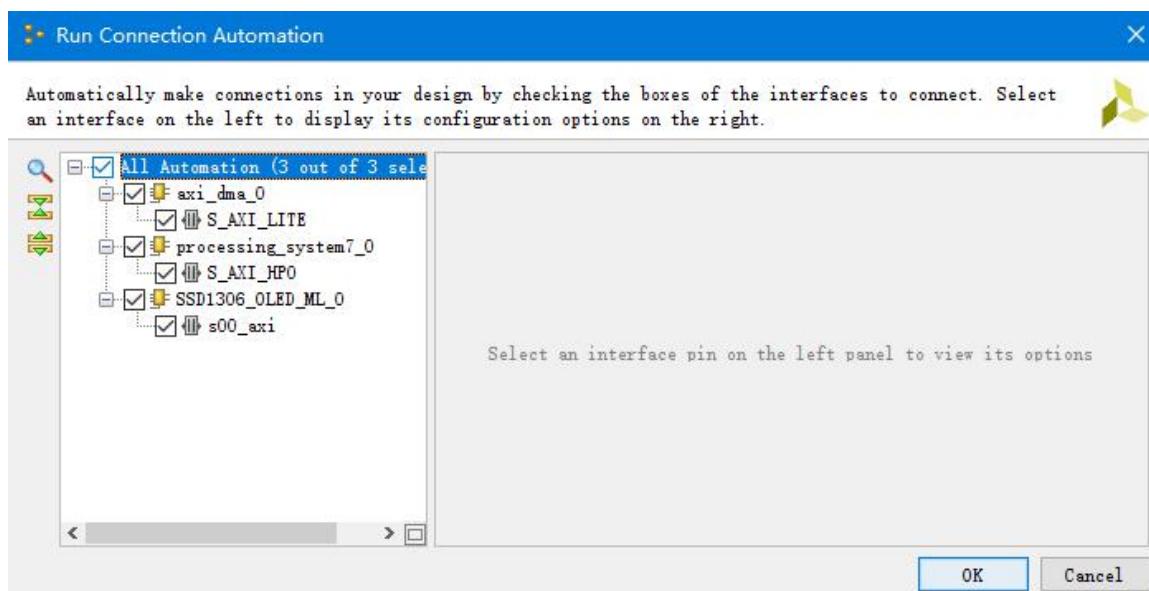
Concat IP 实现了单个分散的信号，整合成总线信号。这里 2 个独立的中断信号，可以合并在一起介入到 ZYNQ IP 的中断信号上。



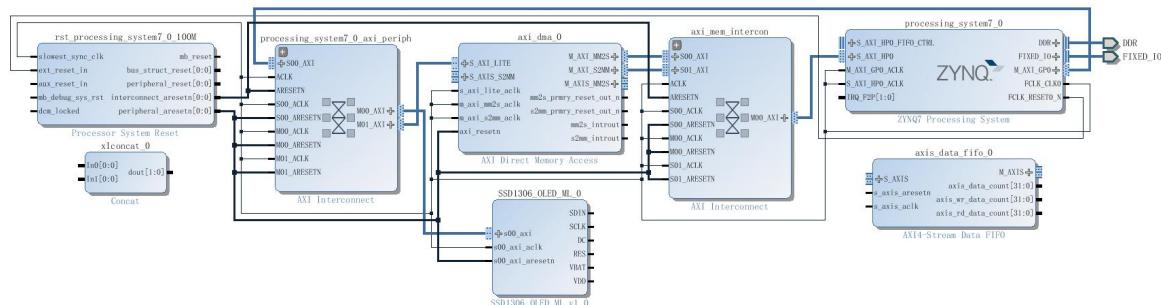
Step16:Run Automation 自动配置 ZYNQ IP 如下图所示



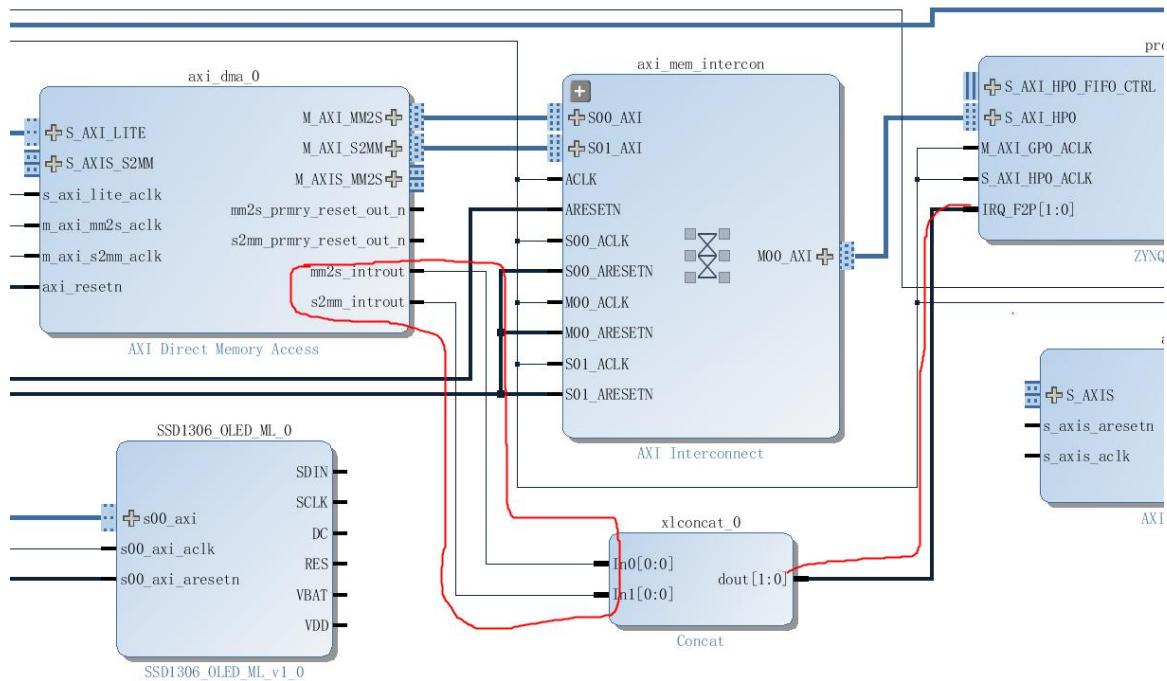
Step17：单击 Run Connection Automation 自动连线，只要软件提示你需要自动连线，一般都需要进行自动连线，除非自己知道如何连线，有特殊需求。



Step18：如果还有提示需要自动连线的继续让软件自动连线，直到出下如下。可以看到，还有未连线的模块。



Step19: 把 DMA 收发中断信号，通过 contact IP 连接到 ZYNQ



Step20:

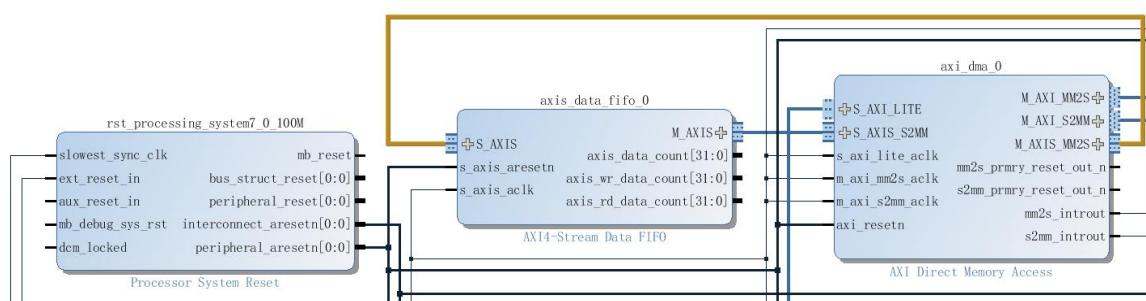
连接 FIFO 的 S_AXIS(写端口)到 DMA 的 M_AXIS(DMA 读端口);

连接 FIFO 的 M_AXIS(读端口)到 DMA 的 S_AXIS(DMA 写端口);

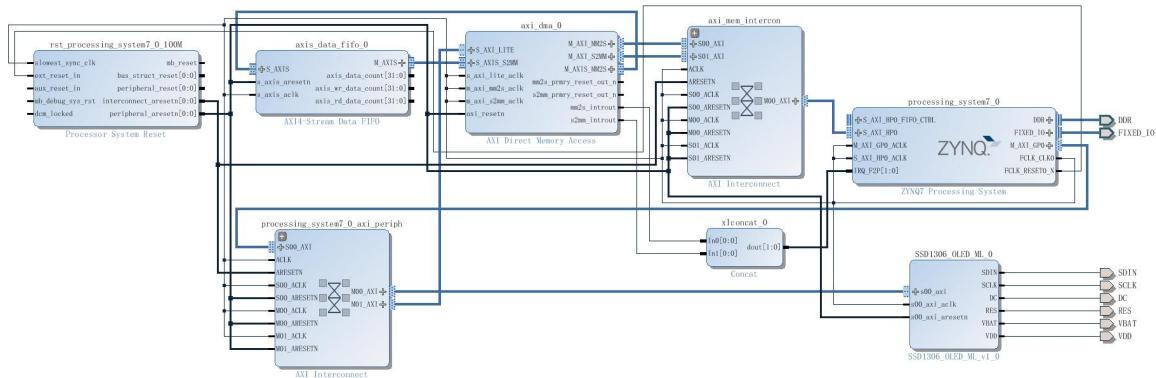
连接 FIFO 的 a_axis_arestn 到 复位 IP 的 peripheral_arestn；

连接 FIFO 的 s_axis_ack 到 ZYNQ IP 的 FCLK0;

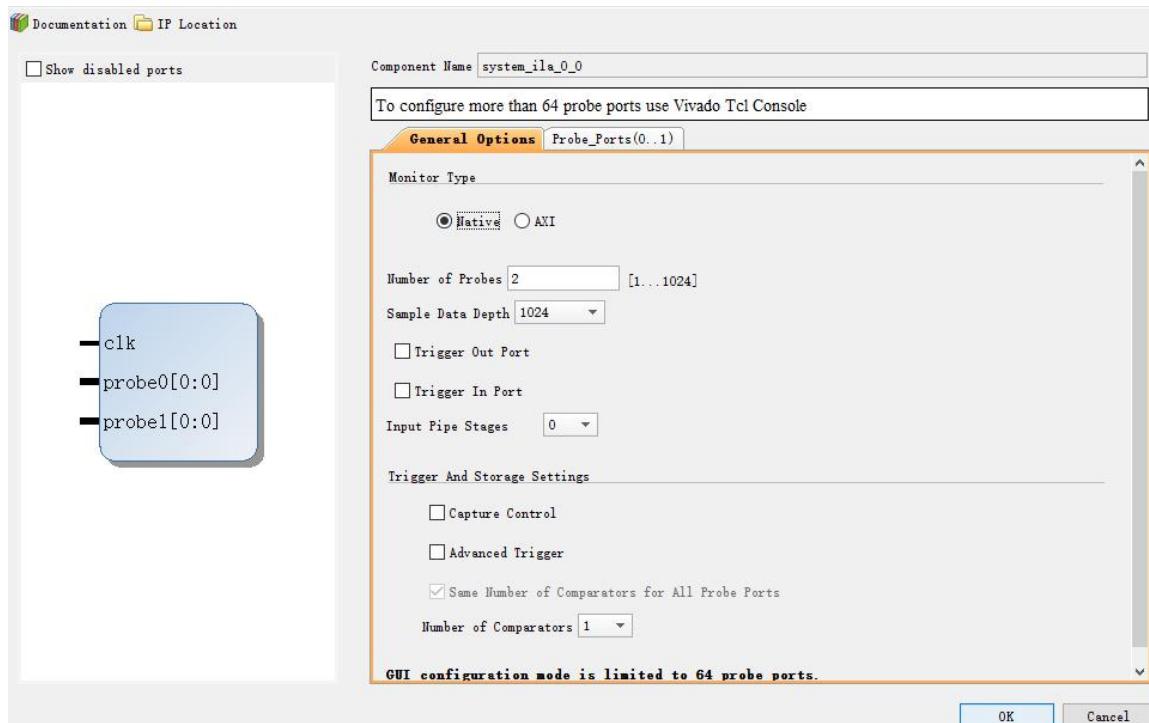
连接完成后如下图

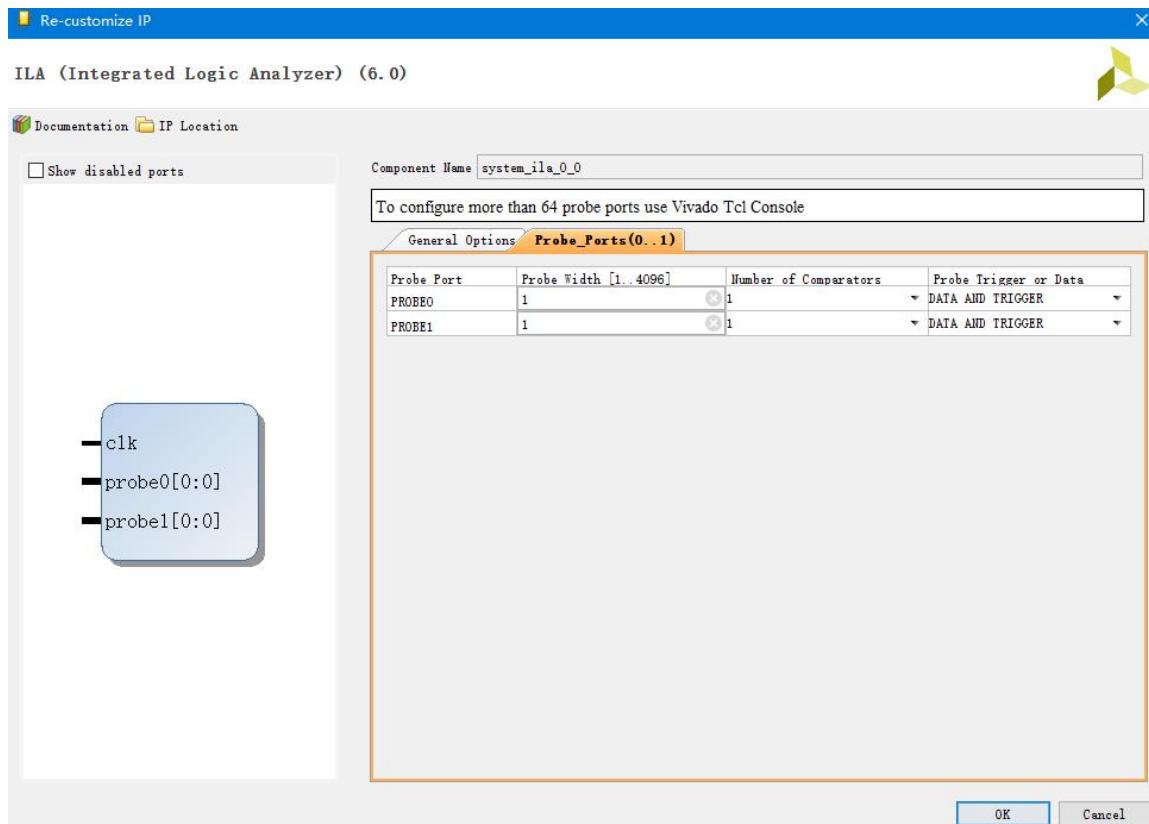


Step21: 把 OLED 模块的 IO 引出来，后面 C 代码部分会用 OLED 显示一些信息（MIZ701N 需要配 OLED 模块）。连接完成后的工程如下图

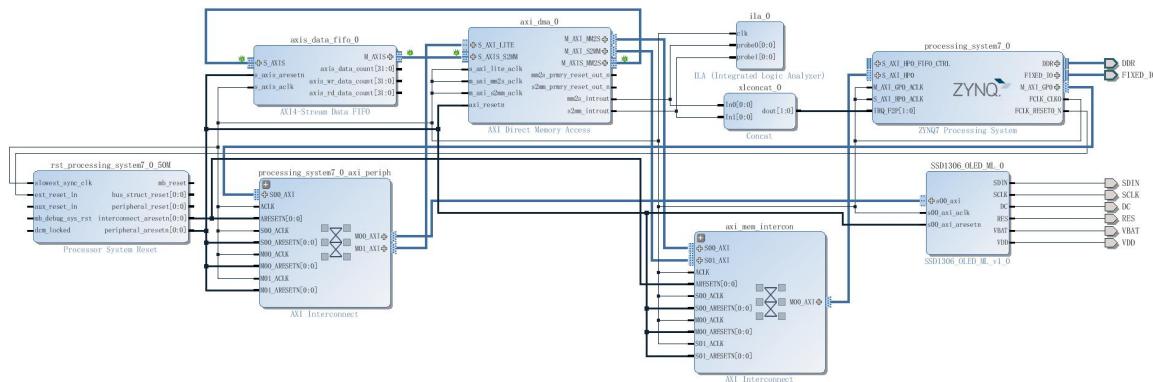


Step22：未来调试的时候可以观察到中断信号的产生，添加 ila 调试 IP 并且进行如下设置





Step23: 把中断信号连接到 ila IP 上，另外，把时钟信号也连接起来。



Step24:

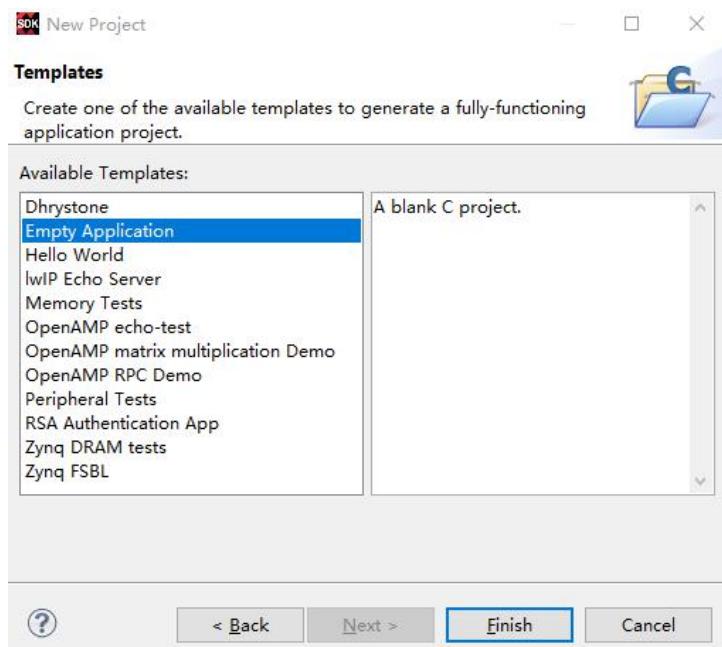
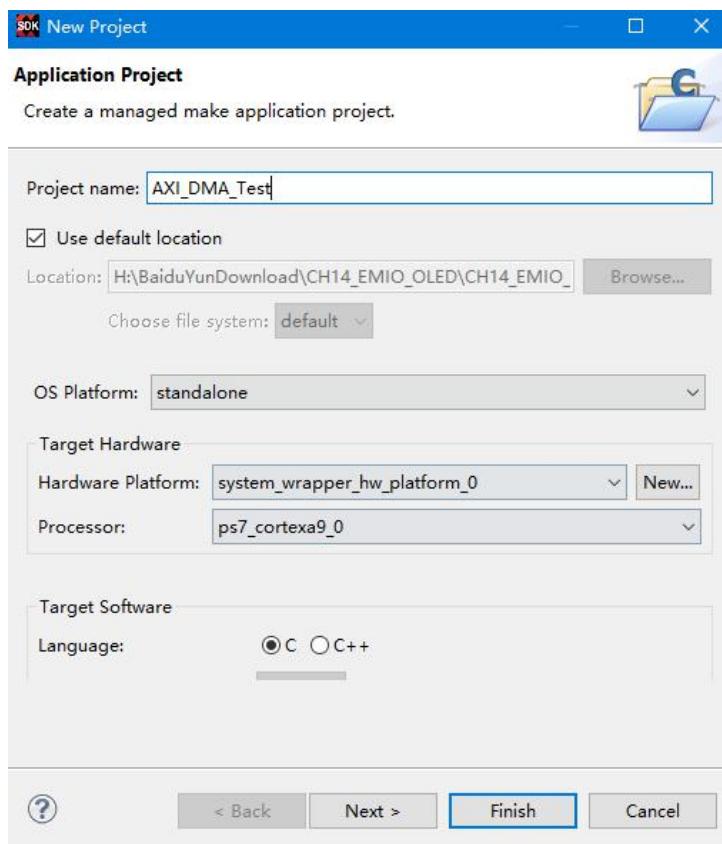
以上就完成独立工程的创建。

之后的过程是 Validate Design->Generate Out products->Create wrappers->Generate Bitstream 产生完成后导入到 SDK 进行软件开发。

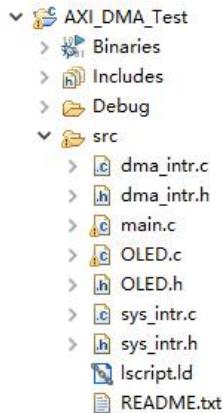
1.3 PS 部分软件分析

1.3.1 新建 SDK 工程

Step1:新建一个名为 AXI_DMA_Test 的空的软件工程



Step2:直接把源码复制过来，软件会自动编译



1.3.2 main.c 源码的分析

`init_intr_sys()`;是对中断资源的初始化，使能中断资源。这个函数里面调用的函数是笔者封装好的初始化函数，使用起来比较方便。一般只要给出中断对象，中断号，就可以对中断进行初始化。

`DMA_Intr_Init(&AxiDma, 0)`;中第一参数是 DMA 的对象，第二参数是硬件 ID
`Init_Intr_System(&Intc)`; 对象是中断对象
`DMA_Setup_Intr_System(&Intc, &AxiDma, TX_INTR_ID, RX_INTR_ID); //注册中断函数，最后 2 个参数是中断号`
`DMA_Intr_Enable(&Intc, &AxiDma)`; 就是启动DMA传输

表 1-3-2-1 init_intr_sys 函数

```

int init_intr_sys(void)
{
    DMA_Intr_Init(&AxiDma,0);//initial interrupt system
    Init_Intr_System(&Intc); // initial DMA interrupt system
    Setup_Intr_Exception(&Intc);
    DMA_Setup_Intr_System(&Intc,&AxiDma,TX_INTR_ID,RX_INTR_ID); //setup dma interrupt
    system
    DMA_Intr_Enable(&Intc,&AxiDma);
}

```

为了发送的数据是已知是确定数据，先对 TxBufferPtr 发送缓冲进行初始化，初始化后用 `Xil_DCacheFlushRange` 函数把数据全部刷到 DDR 中。

`XAxiDma_SimpleTransfer` 函数为启动一次DMA接收传输。
`XAxiDma_SimpleTransfer` 函数为启动一次DMA发送传输
`DMA_CheckData` 函数为对接收的数据进行校验和对比。

表1-3-2-1 DMA_CheckData

```

int axi_dma_test()

```

```
{  
    int Status;  
    TxDone = 0;  
    RxDone = 0;  
    Error = 0;  
  
    xil_printf("\r\n----DMA Test----\r\n");  
    print_message( "----DMA Test----",0); //oled print  
  
    xil_printf("PKT_LEN=%d\r\n",MAX_PKT_LEN);  
  
    sprintf(oled_str,"PKT_LEN=%d",MAX_PKT_LEN);  
    print_message(oled_str,1); //oled print  
  
    //while(1)  
    for(i = 0; i < Tries; i ++)  
    {  
        Value = TEST_START_VALUE + (i & 0xFF);  
        for(Index = 0; Index < MAX_PKT_LEN; Index ++){  
            TxBUFFERPTR[Index] = Value;  
  
            Value = (Value + 1) & 0xFF;  
        }  
  
        /* Flush the SrcBuffer before the DMA transfer, in case the Data Cache  
         * is enabled  
         */  
        Xil_DCacheFlushRange((u32)TxBUFFERPTR, MAX_PKT_LEN);  
  
        Status = XAxiDma_SimpleTransfer(&AxiDma,(u32) RxBufferPtr,  
                                         MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);  
  
        if (Status != XST_SUCCESS) {  
            return XST_FAILURE;  
        }  
  
        Status = XAxiDma_SimpleTransfer(&AxiDma,(u32) TxBUFFERPTR,  
                                         MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);  
  
        if (Status != XST_SUCCESS) {  
            return XST_FAILURE;  
        }  
}
```

```
/*
 * Wait TX done and RX done
 */
while (!TxDone || !RxDone) {
    /* NOP */
}

success++;
TxDone = 0;
RxDone = 0;

if (Error) {
    xil_printf("Failed test transmit%s done, "
               "receive%s done\r\n", TxDone? ":" not",
               RxDone? ":" not");
    goto Done;
}
/*
 * Test finished, check data
 */
Status = DMA_CheckData(MAX_PKT_LEN, (TEST_START_VALUE + (i & 0xFF)));
if (Status != XST_SUCCESS) {
    xil_printf("Data check failed\r\n");
    goto Done;
}

xil_printf("AXI DMA interrupt example test passed\r\n");
xil_printf("success=%d\r\n", success);
sprintf(oled_str,"success=%d",success);
print_message(oled_str,2);
/* Disable TX and RX Ring interrupts and return success */
DMA_DisableIntrSystem(&Intc, TX_INTR_ID, RX_INTR_ID);

Done:
xil_printf("--- Exiting Test --- \r\n");
print_message("--Exiting Test---",3);
return XST_SUCCESS;
}

int init_intr_sys(void)
{
    DMA_Intr_Init(&AxiDma,0); //initial interrupt system
```

```

Init_Intr_System(&Intc); // initial DMA interrupt system
Setup_Intr_Exception(&Intc);
DMA_Setup_Intr_System(&Intc,&AxiDma,TX_INTR_ID,RX_INTR_ID);//setup dma interrupt
system
DMA_Intr_Enable(&Intc,&AxiDma);
}

int main(void)
{
    init_intr_sys();
    oled_fresh_en();// enable oled
    axi_dma_test();

}

```

1.3.3 dma_intr.c 源码分析

`XAxiDma *AxiDmaInst = (XAxiDma *)Callback;` 这句代码是为了获取当前中断的对象。`void *Callback` 是一个无符号的指针，传递进来的阐述可以强制转换成其他任何的对象，这里就是强制转换成 `XAxiDma` 对象了。

`IrqStatus =XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE)` 这个函数获取当前中断号。

`XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE)`；这个函数是响应当前中断，通知CPU 当前中断已经被接收，并且清除中断标志位。

如果中断全部正确，`TxDone` 将被置为1表示发送中断完成。

如果有错误，则复位DMA，并且设置超时参数

表 1-3-3-1 DMA_TxIntrHandler 函数

```

/****************************************************************************
*
* This is the DMA TX Interrupt handler function.
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then sets the TxDone.flag
*
* @param   Callback is a pointer to TX channel of the DMA engine.
*
* @return  None.
*
* @note   None.
*/

```

```
/*
*****
static void DMA_TxIntrHandler(void *Callback)
{

    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE);

    /* Acknowledge pending interrupts */

    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {

        return;
    }

    /*
     * If error interrupt is asserted, raise error flag, reset the
     * hardware to recover from the error, and return with no further
     * processing.
     */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

        Error = 1;

        /*
         * Reset should never fail for transmit channel
         */
        XAxiDma_Reset(AxiDmaInst);

        TimeOut = RESET_TIMEOUT_COUNTER;

        while (TimeOut) {
            if (XAxiDma_ResetIsDone(AxiDmaInst)) {


```

```

        break;
    }

    TimeOut -= 1;
}

return;
}

/*
 * If Completion interrupt is asserted, then set the TxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    TxDone = 1;
}
}

```

接收中断函数的原理和发送一样

`XAxiDma *AxiDmaInst = (XAxiDma *)Callback;` 这句代码是为了获取当前中断的对象。`void *Callback` 是一个无符号的指针，传递进来的阐述可以强制转换成其他任何的对象，这里就是强制转换成 `XAxiDma` 对象了。

`IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);` 这个函数是获取当前中断号。

`XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);` 这个函数是响应当前中断，通知CPU 当前中断已经被接收，并且清除中断标志位。

如果中断全部正确，`RxDone` 将被置为1表示接收中断完成。

如果有错误，则复位DMA，并且设置超时参数

表 1-3-3-2 DMA_RxIntrHandler 函数

```

*****
/*
*
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.
*
* @paramCallback is a pointer to RX channel of the DMA engine.
*
* @return None.
*
* @note      None.

```

```
*  
*****  
static void DMA_RxIntrHandler(void *Callback)  
{  
    u32 IrqStatus;  
    int TimeOut;  
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;  
  
    /* Read pending interrupts */  
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);  
  
    /* Acknowledge pending interrupts */  
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);  
  
    /*  
     * If no interrupt is asserted, we do not do anything  
     */  
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {  
        return;  
    }  
  
    /*  
     * If error interrupt is asserted, raise error flag, reset the  
     * hardware to recover from the error, and return with no further  
     * processing.  
     */  
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {  
  
        Error = 1;  
  
        /* Reset could fail and hang  
         * NEED a way to handle this or do not call it??  
         */  
        XAxiDma_Reset(AxiDmaInst);  
  
        TimeOut = RESET_TIMEOUT_COUNTER;  
  
        while (TimeOut) {  
            if (XAxiDma_ResetIsDone(AxiDmaInst)) {  
                break;  
            }  
  
            TimeOut -= 1;  
        }  
    }  
}
```

```

    }

    return;
}

/*
 * If completion interrupt is asserted, then set RxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    RxDone = 1;
}
}
}

```

表 1-3-3-3 DMA_CheckData 函数

```

*****
/*
*
*
* This function checks data buffer after the DMA transfer is finished.
*
*
* We use the static tx/rx buffers.
*
*
* @param      Length is the length to check
* @param      StartValue is the starting value of the first byte
*
*
* @return
*          - XST_SUCCESS if validation is successful
*          - XST_FAILURE if validation is failure.
*
*
* @note      None.
*
*****
int DMA_CheckData(int Length, u8 StartValue)
{
    u8 *RxPacket;
    int Index = 0;
    u8 Value;

    RxPacket = (u8 *) RX_BUFFER_BASE;
    Value = StartValue;

    /* Invalidate the DestBuffer before receiving the data, in case the
     * Data Cache is enabled
     */
}

```

```

#ifndef __aarch64__
    Xil_DCacheInvalidateRange((u32)RxPacket, Length);
#endif

    for(Index = 0; Index < Length; Index++) {
        if (RxPacket[Index] != Value) {
            xil_printf("Data error %d: %x/%x\r\n",
                       Index, RxPacket[Index], Value);

            return XST_FAILURE;
        }
        Value = (Value + 1) & 0xFF;
    }

    return XST_SUCCESS;
}

```

1.3.4 dam_intr.h 文件分析

一般把 DMA 相关变量、常量、函数的声明或者定义放到头文件中，dam_intr.h 比较关键的参数有
TX_BUFFER_BASE 定义了 DMA 发送缓存的基地址
RX_BUFFER_BASE 定义了 DMA 接收缓存的基地址
MAX_PKT_LEN 表示每一包数据传输的长度
NUMBER_OF_TRANSFERS 用在连续测试的时候的测试次数
TEST_START_VALUE 用于 测试的起始参数

```

int DMA_CheckData(int Length, u8 StartValue); 对数据进行对比
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16
RxIntrId);DMA 中断注册
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr); DMA 中断使能
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);DMA 中断初始化

```

表 1-3-4 dam_intr.h

```

/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
*/
#ifndef DMA_INTR_H
#define DMA_INTR_H
#include "xaxidma.h"

```

```
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
#include "xscugic.h"

/********************* Constant Definitions *****/
/*
 * Device hardware build related constants.
 */
#define DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID

#define MEM_BASE_ADDR      0x01000000

#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR

#define TX_BUFFER_BASE      (MEM_BASE_ADDR + 0x00100000)
#define RX_BUFFER_BASE      (MEM_BASE_ADDR + 0x00300000)
#define RX_BUFFER_HIGH      (MEM_BASE_ADDR + 0x004FFFFF)

/* Timeout loop counter for reset
 */
#define RESET_TIMEOUT_COUNTER    10000
/* test start value
 */
#define TEST_START_VALUE    0xC
/*
 * Buffer and Buffer Descriptor related constant definition
 */
#define MAX_PKT_LEN      256//4MB
/*
 * transfer times
 */
#define NUMBER_OF_TRANSFERS 100000

extern volatile int TxDone;
extern volatile int RxDone;
extern volatile int Error;

int DMA_CheckData(int Length, u8 StartValue);
```

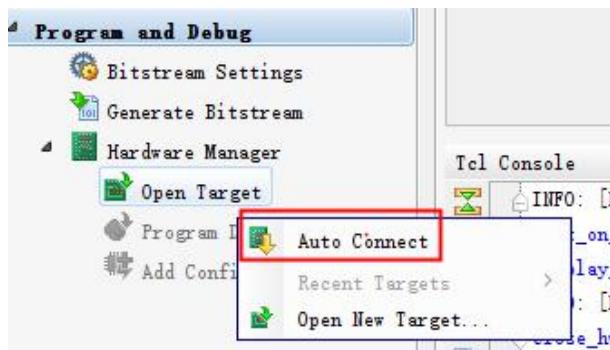
```

int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma * DMA_Ptr);
int DMA_Intr_Init(XAxiDma * DMA_Ptr,u32 DeviceId);
#endif

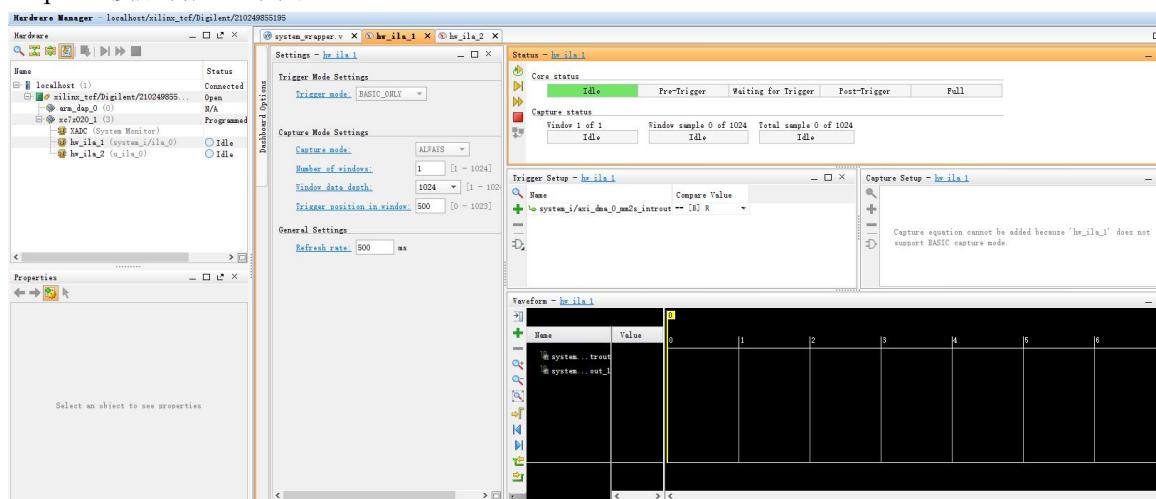
```

1.4 测试结果

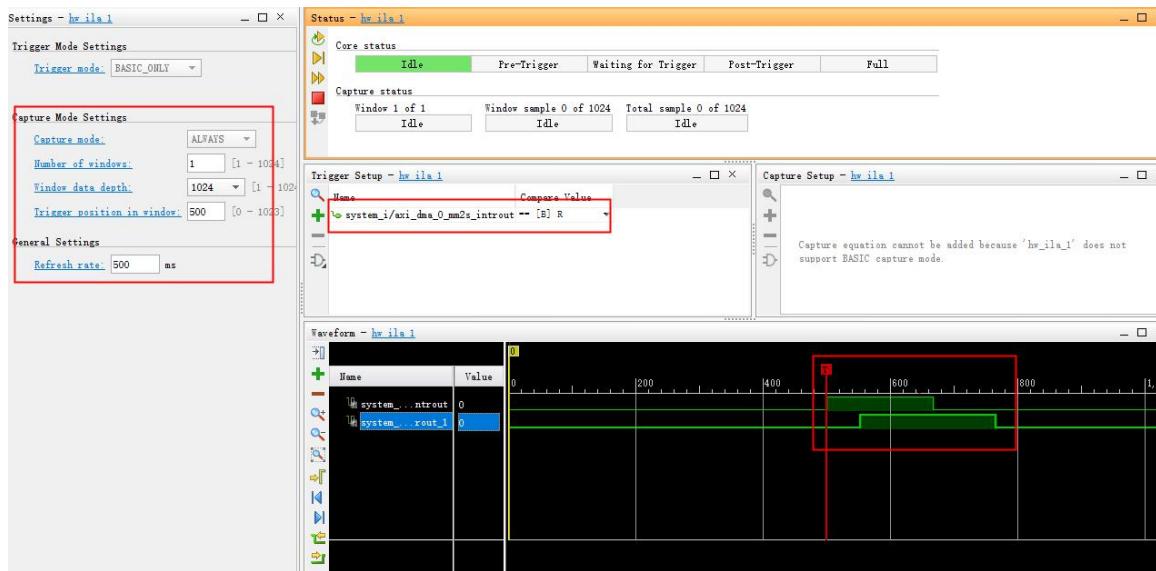
Step1:在 VIVADO 工程中点击 Open Target 然后点击 Auto Connect(前面必须先启动 SDK)



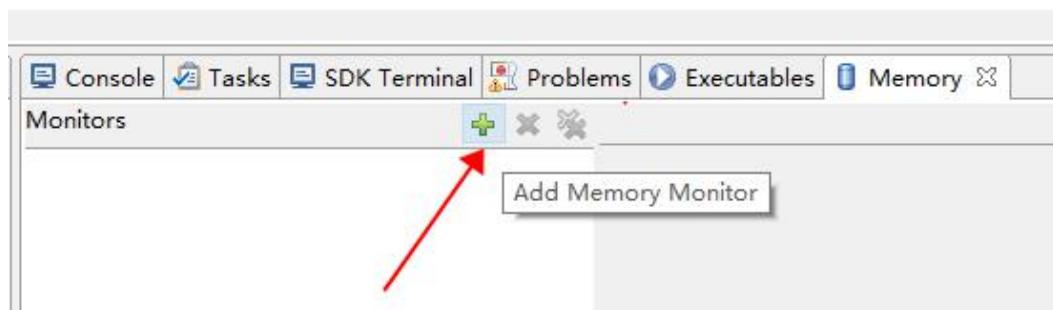
Step2:连接成功后入下图



Step3:设置中断条件，以及观察波形的偏移为 500，当中断触发的时候，如下图所示



点击添加接收内存部分地址用于观察内存中的数据 地址为 0X01300000



为了观察到以下数据，设置断点，之让收发程序先跑一次，可以看到第一个数据是 0X0C 后面是依次加 1

The screenshot shows the Monitors tab displaying memory data at address 0x01300000. A red box highlights the first byte value '0F0E0D0C'.

Address	0 - 3	4 - 7	8 - B	C - F
01300000	0F0E0D0C	13121110	17161514	1B1A1918
01300010	1F1E1D1C	28222120	27262524	2B2A2928
01300020	2F2E2D2C	33323130	37363534	3B3A3938
01300030	3F3E3D3C	43424140	47464544	4B4A4948
01300040	4F4E4D4C	53525150	57565554	5B5A5958
01300050	5F5E5D5C	63626160	67666564	6B6A6968
01300060	6F6E6D6C	73727170	77767574	7B7A7978
01300070	7F7E7D7C	83828180	87868584	8B8A8988
01300080	8F8E8D8C	93929190	97969594	9B9A9998
01300090	9F9E9D9C	A3A2A1A0	A7A6A5A4	ABAA9A8
013000A0	AFAEADAC	B3B2B1B0	B7B6B5B4	BBBAB9B8
013000B0	BFBEBCBC	C3C2C1C0	C7C6C5C4	CBCAC9C8
013000C0	CFCECDCC	D3D2D1D0	D7D6D5D4	DBDAD9D8
013000D0	DFDEDDDC	E3E2E1E0	E7E6E5E4	EBEAE9E8
013000E0	EFEEEDEC	F3F2F1F0	F7F6F5F4	FBFAF9F8
013000F0	FFFEEFD0C	03020100	07060504	0B0A0908
01300100	E040EE78	6B112AAE	46A14489	232368AC
01300110	91522A8A	7C625002	B4A305E0	ACA1A126
01300120	25D33609	160100BB	A4832B0B	BAA5106F

S03_CH02_AXI_DMA PL 发送数据到 PS

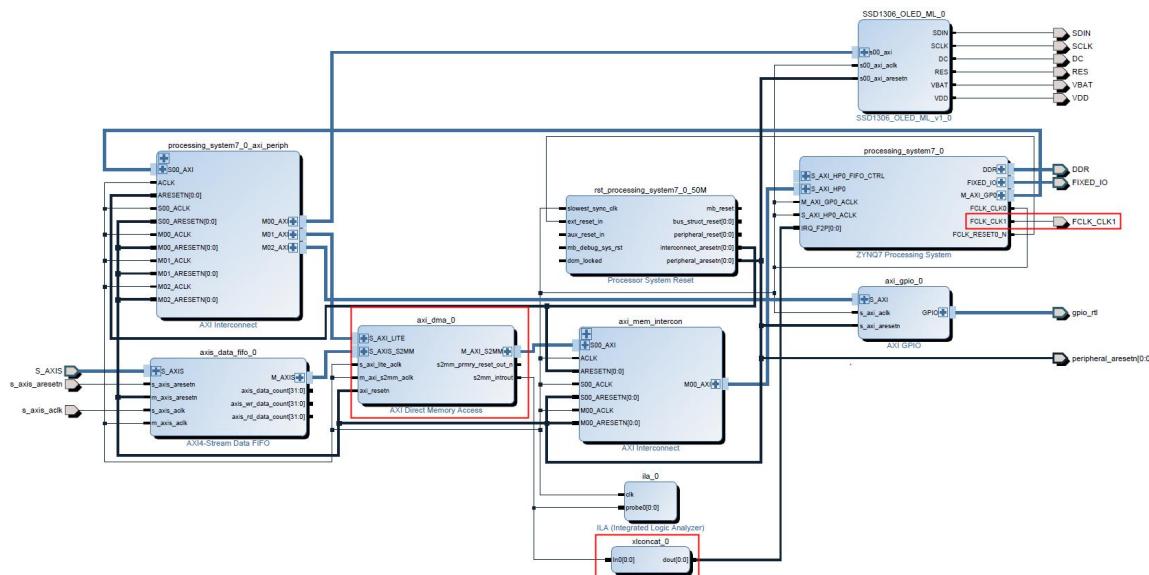
1.1 概述

本课程的设计原理分析。

本课程循序渐进，承接《S03_CH01_AXI_DMA_LOOP 环路测试》这一课程，在 DATA FIFO 端加入 FPGA 代码，通过 verilog 代码对 FIFO 写。其他硬件构架和《S03_CH01_AXI_DMA_LOOP 环路测试》一样。

《S03_CH01_AXI_DMA_LOOP 环路测试》课程中，详解讲解了工程步骤的创建，本章开始，一些简单的操作步骤将会省去。

1.2 系统构架框图

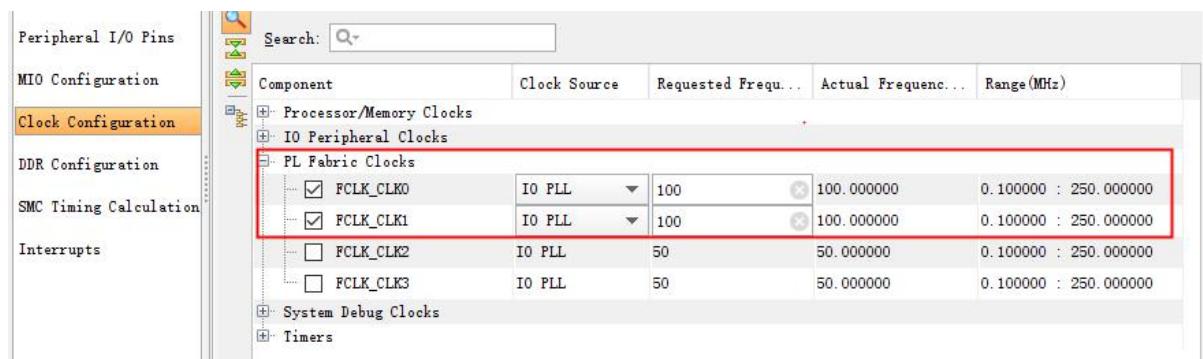


在上图中，红色标记部分是和前面课程有稍微差异的部分。读者需要好好注意下。

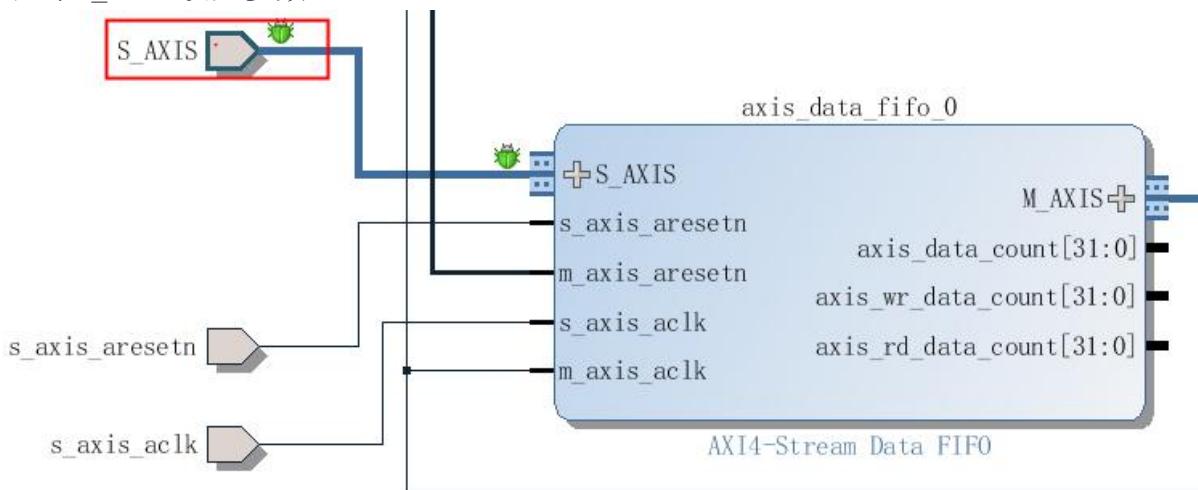
下面看下关键模块的设置

1.2.1 ZYNQ IP 的设置

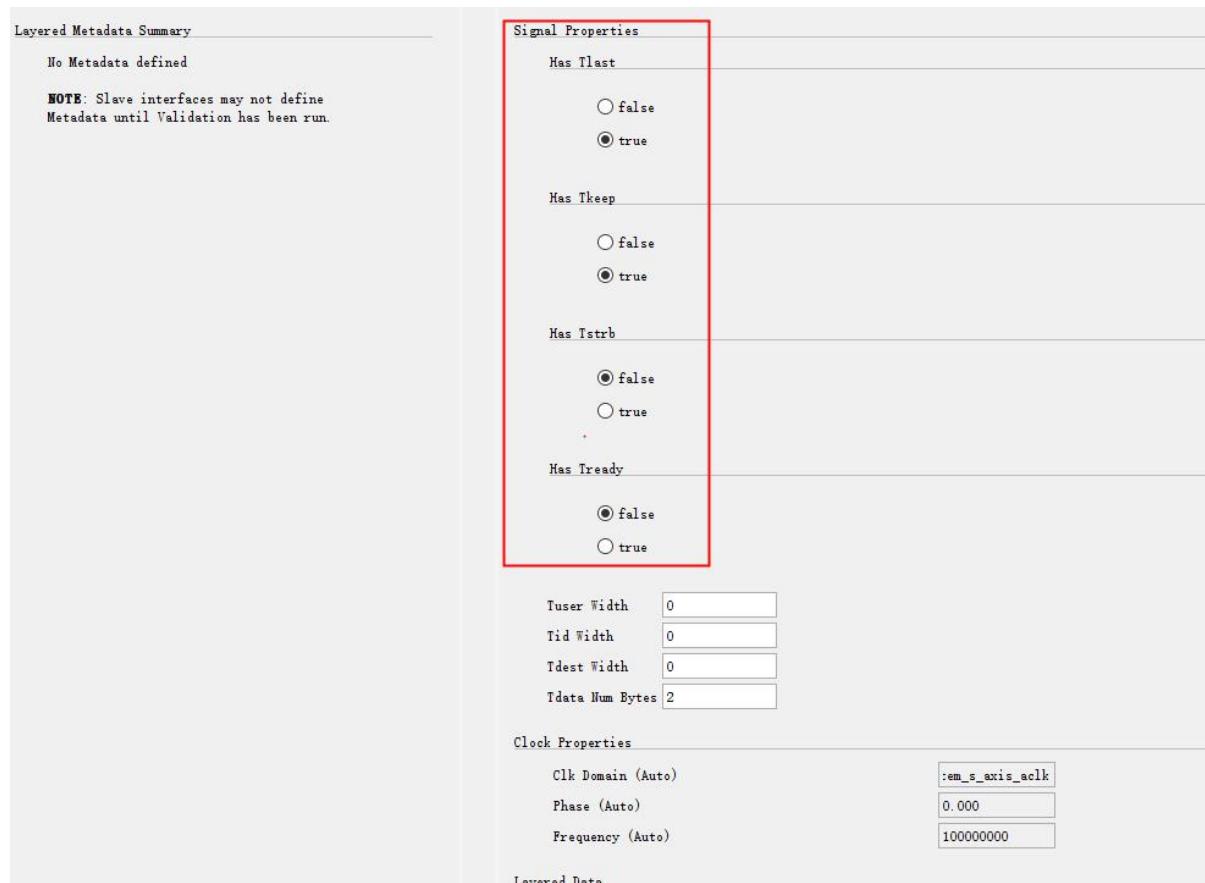
增一路 FCK_CLK1 为 100MHZ（也可以设置其他频率）并且引出到外部提供 verilog 编程时钟。



双击 S_AXIS 设置参数

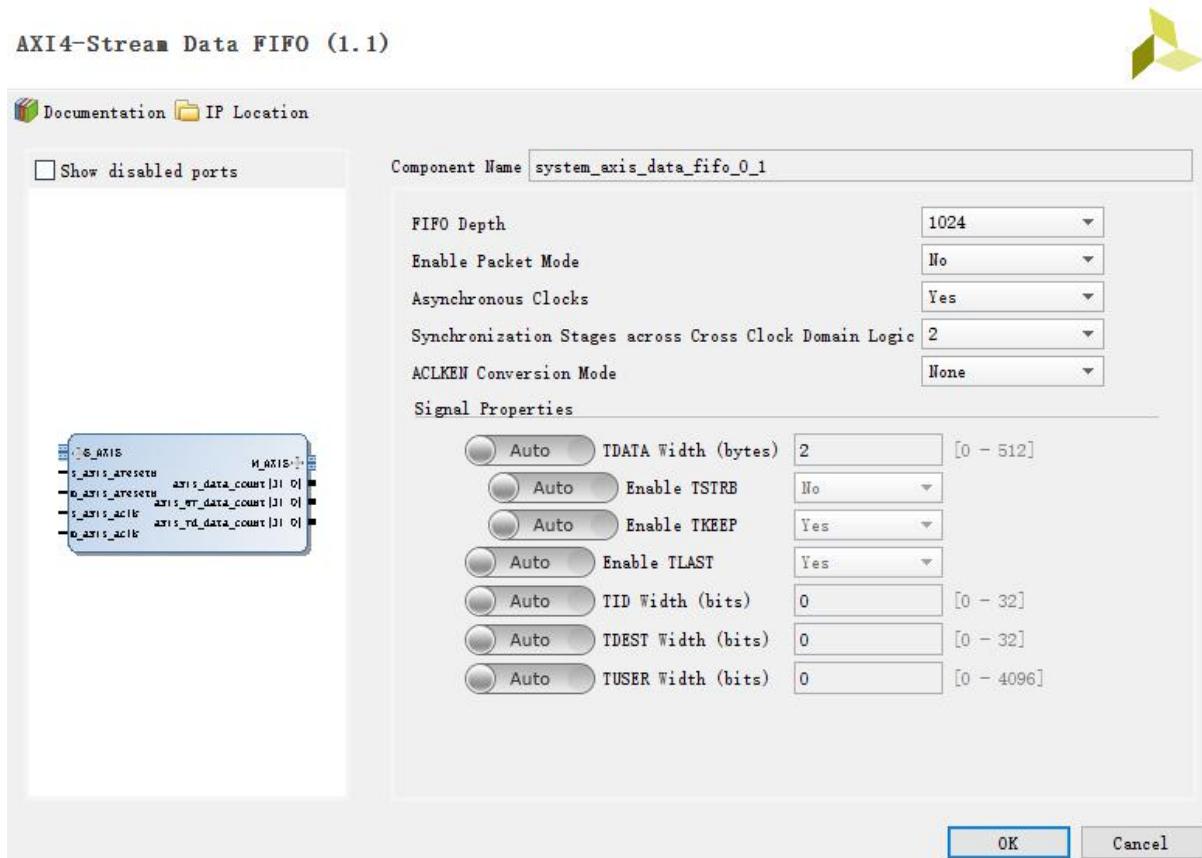


设置如下



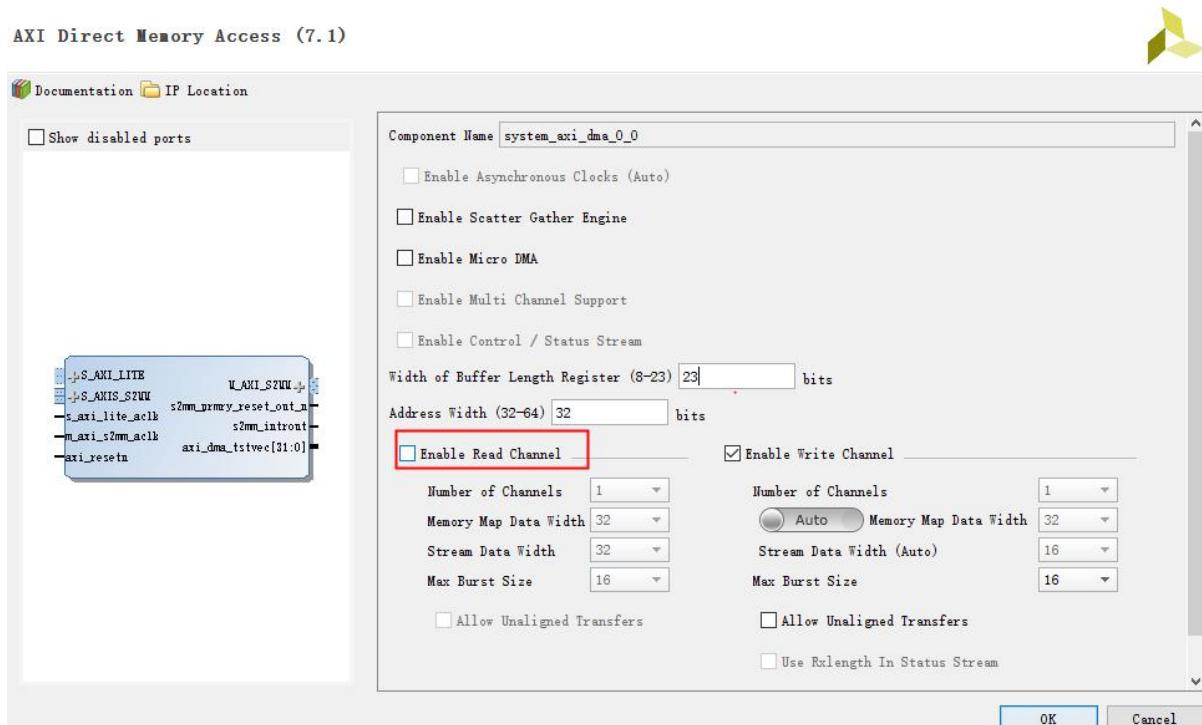
双击 FIFO 进行如下设置

AXI4-Stream Data FIFO (1.1)

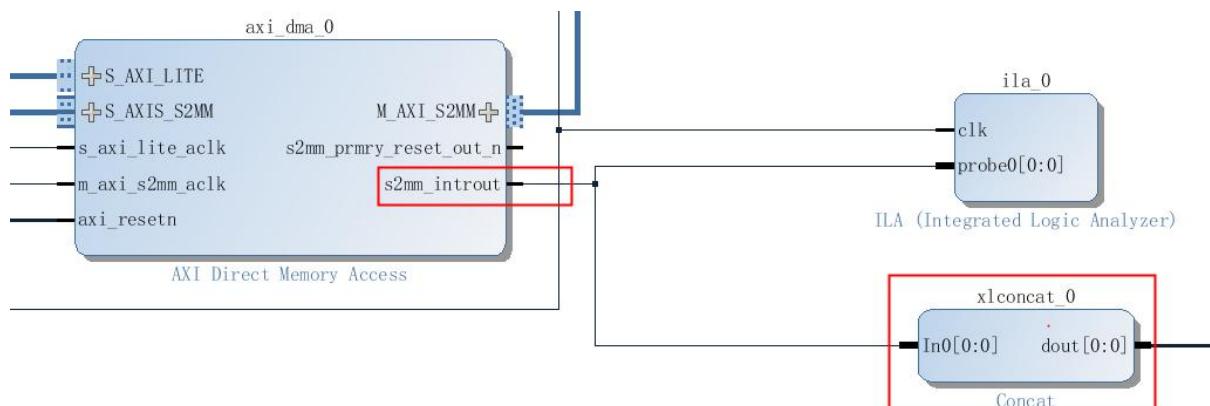


由于只有写 DMA 通道，因此不用勾选读 DMA 通道

AXI Direct Memory Access (7.1)



既然只用到了 DMA 写通道，也就只要使用 1 路中断资源。



1.3 PS 部分

相对于《S03_CH01_AXI_DMA_LOOP 环路测试》中的代码，本章代码只有 DMA 的接收部分。在 main.c 源码中，实现了数据 DMA 的测速，并且通过 OLED 显示出来。为了实现测试，有增加了定时间断，定时器每过 0.5S 中断一次。

中断初始化函数，如下

表 1-3-1 init_intr_sys 函数

```
int init_intr_sys(void)
{
    DMA_Intr_Init(&AxiDma,0); //initial interrupt system
    Timer_init(&Timer,TIMER_LOAD_VALUE,0);
    Init_Intr_System(&Intc); // initial DMA interrupt system
    Setup_Intr_Exception(&Intc);
    DMA_Setup_Intr_System(&Intc,&AxiDma,TX_INTR_ID,RX_INTR_ID); //setup dma interrupt
    system
    Timer_Setup_Intr_System(&Intc,&Timer,TIMER_IRPT_INTR);
    DMA_Intr_Enable(&Intc,&AxiDma);
}
```

DMA 读测速的部分的原理是计数 DMA 读传输的次数，然后每过 2 秒，计算一次速度。通过 OLED 显示测速。

表 1-3-2 测试代码

```
if(RxDone)
{
    RxDone=0;
    RX_ready=1;
    RX_success++;
}

if(TxDone)
{
    TxDone=0;
    TX_ready=1;
```

```
    TX_success++;
}

if(usec==2)
{

    usec=0;
    sprintf(oled_str, "RX_cnt=%d", RX_success);
    xil_printf("%s\r\n", oled_str);
    print_message(oled_str,0);

    speed_rx = MAX_PKT_LEN*RX_success/1024/1024;

    sprintf(oled_str, "RX_sp=%.2fMB/S", speed_rx);
    xil_printf("%s\r\n", oled_str);
    print_message(oled_str,1);

    sprintf(oled_str, "TX_cnt=%d", TX_success);
    xil_printf("%s\r\n", oled_str);
    print_message(oled_str,2);

    speed_tx = (MAX_PKT_LEN)*TX_success/1024/1024;

    sprintf(oled_str, "TX_sp=%.2fMB/S", speed_tx);
    xil_printf("%s\r\n", oled_str);
    print_message(oled_str,3);

    RX_success=0;
    TX_success=0;

}
```

定时器中断在第二季《S02_CH08_ZYNQ 定时器中断实验》已经详细讲解过，至于 DMA 中断《S03_CH01_AXI_DMA_LOOP 环路测试》中也已经详细讲解，不在过多复述。

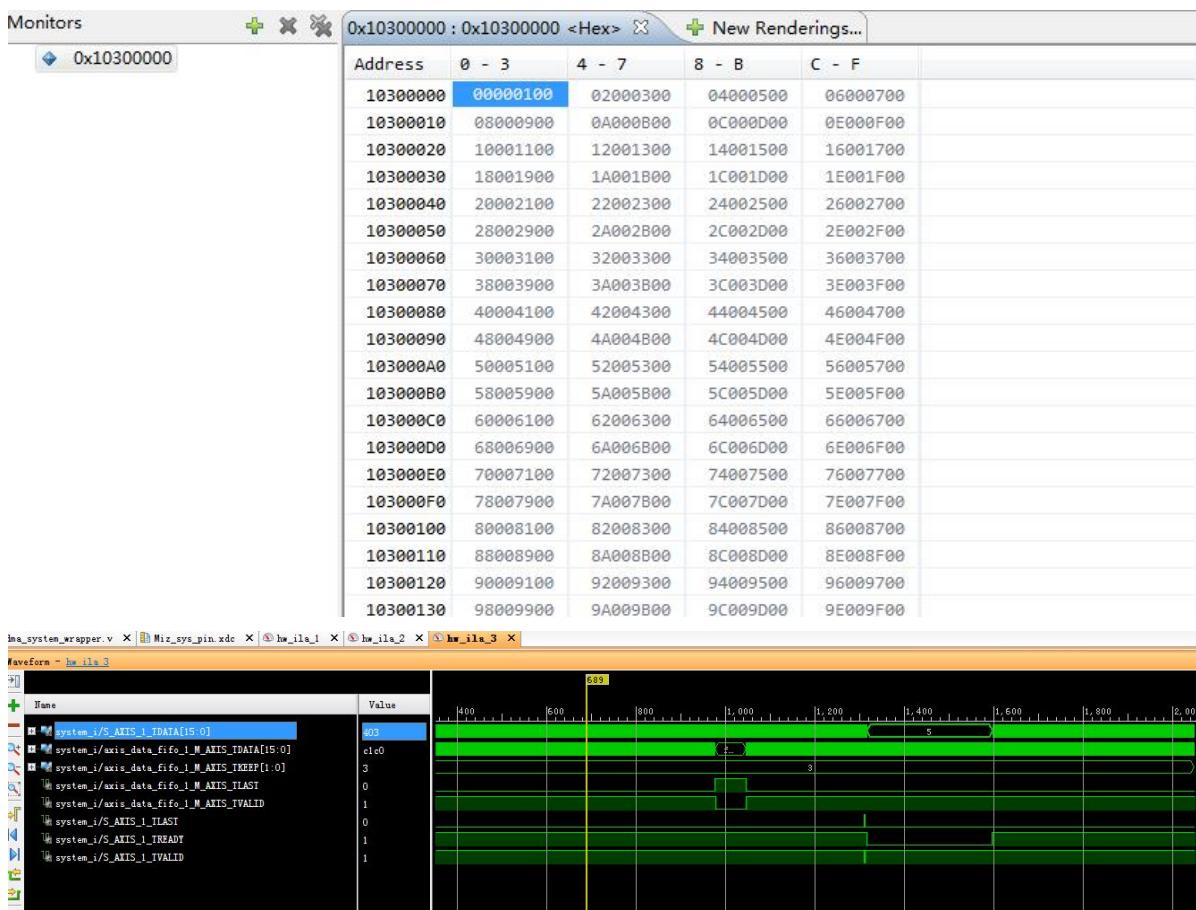
1.4 测试结果

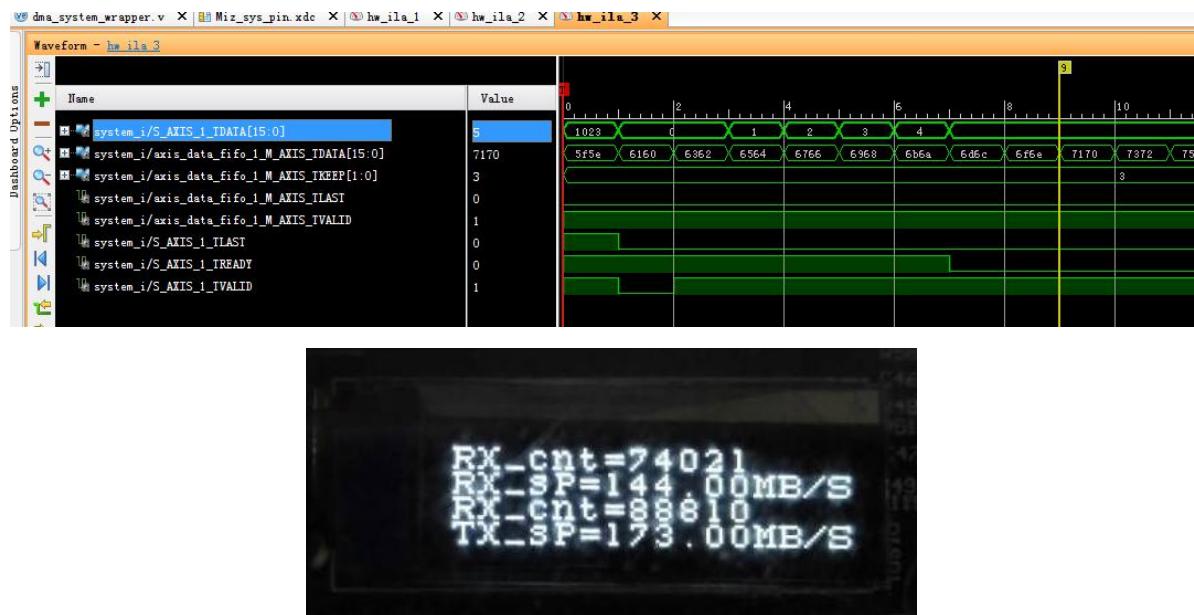
Connected to: Serial (COM69, 115200, 0, 8)

```

Connected to COM69 at 115200
RX_cnt=74020
RX_sp=144.00MB/S
RX_cnt=88810
TX_sp=173.00MB/S
RX_cnt=74020
RX_sp=144.00MB/S
RX_cnt=88809
TX_sp=173.00MB/S
RX_cnt=74022
RX_sp=144.00MB/S
RX_cnt=88800
TX_sp=173.00MB/S
RX_cnt=74020
RX_sp=144.00MB/S
RX_cnt=88812
TX_sp=173.00MB/S

```





S03_CH03_AXI_DMA_OV7725 摄像头采集系统

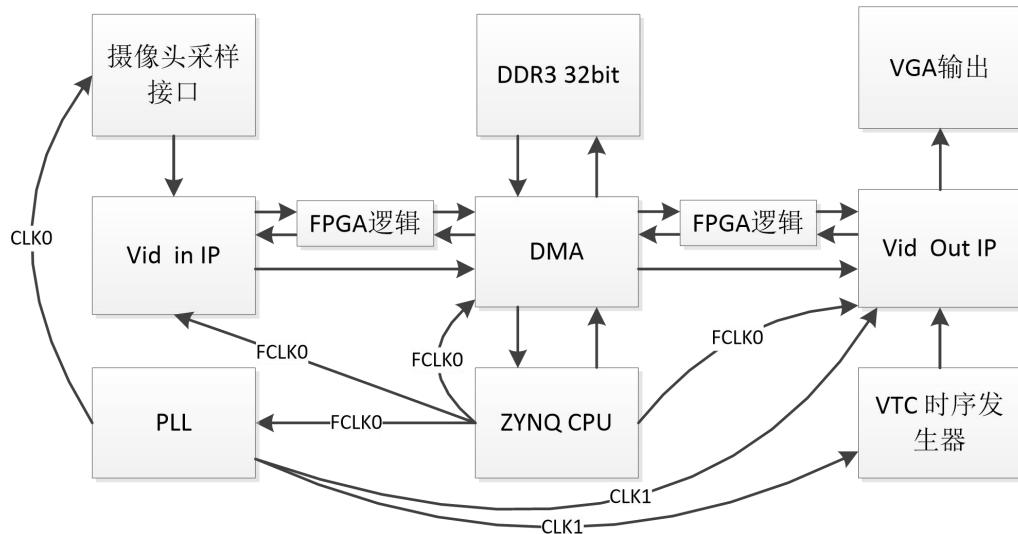
3.1 概述

本课程讲解如何搭建基于 DMA 的图形系统，方案原理如下。

摄像头采样图像数据后通过 DMA 送入到 DDR，在 PS 部分产生 DMA 接收中断，在接收中断里面再把 DDR 里面保持的图形数据 DMA 发送出去。在 FPGA 的接收端口部分产生 VID OUT 时序驱动 VGA 显示器显示图形。MIZ701N 没有 VGA 接口，可以跳过直接看《S03_CH05_AXI_DMA_HDMI 图像输出》或者大家不想看本章的也可以直接跳到《S03_CH05_AXI_DMA_HDMI 图像输出》这两节课的核心教学内容一样。《S03_CH03_AXI_DMA_OV7725 摄像头采集系统》、《S03_CH04_AXI_DMA_OV5640 摄像头采集系统》、《S03_CH05_AXI_DMA_HDMI 图像输出》。读者可以根据自己需求情况而阅读，请知悉。

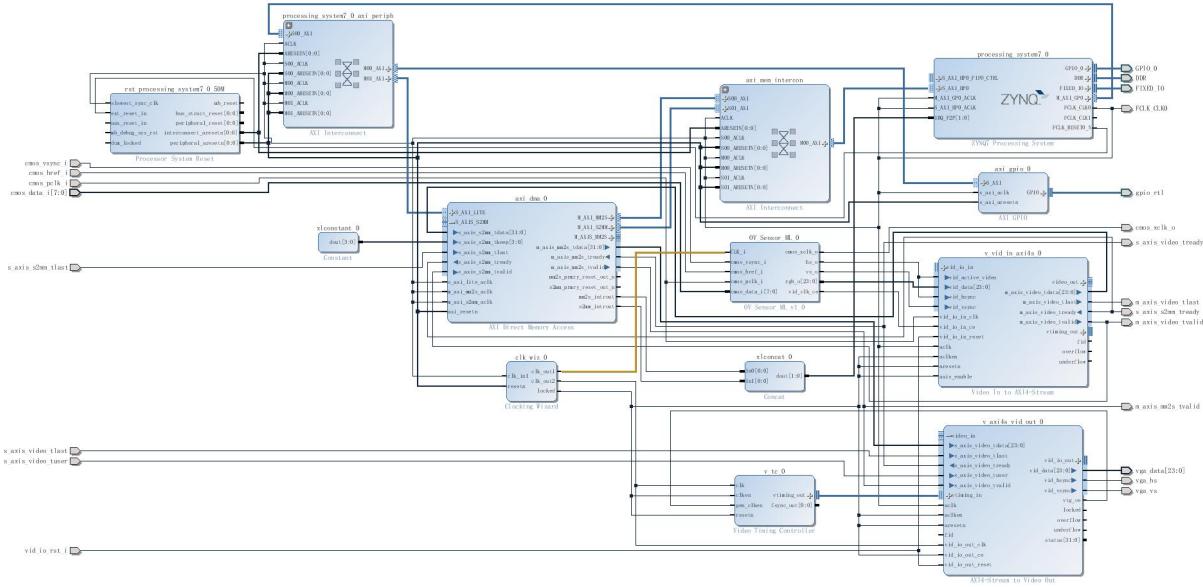
3.2 系统构架

3.2.1 构架方案图



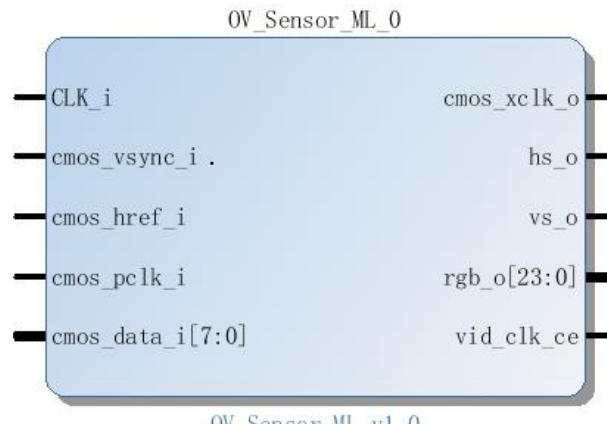
摄像头接口采集的摄像头数据，进过 vid in 视频输入 IP 后，还需要通过用户 FPGA 逻辑编程，和 DMA IP 之间实现握手协议，实现把数据通过 DMA 写入到 DDR。每次写入一副图像的数据后，产生一次接收中断，接收中断函数，会把数据三缓存后，在通过 DMA 发出去，DMA 发送完成后产生中断，在中断中，把缓存好的图像发送出去。DMA 发送的数据需要发送到 vid out 视频输出 IP。同理，DMA 和 vid out IP 之间也许需要增加 FPGA 用户代码实现接口的握手协议。数据进入 vid out 后，会随同 vtc IP 输出符合 VGA 时序的图像信号。vid out 的输出就可以直接定义成 VGA 信号输出。

3.2.2 构造 BLOCK 模块化设计方案图



3.3 vid in IP 介绍

3.3.1 OV Sensor ML 自定义 IP 模块



外部信号接口说明.

CLK_i：为输入时钟，通常接 24MHz 或者 25MHz

Cmos xc1k o:摄像头工作，通常直接把 CLK i 连接到 cmos xc1k o

Cmos vsync i: 摄像头场同步输入 上升沿代表场同步开始

Cmos href i:摄像头行同步输入

Cmos data[7:0]:摄像头数据输入

Hs o: 采集 OV Sensor ML IP 输出的行数据有效

Vid_clk_ce:此信号用于和 vid_in IP 的时钟同步(由于 OV_Sensor_ML IP 是每两个时钟输出一次 rgb[23:0]的图像数据，因此需要通过 Vid_clk_ce 对时钟频率进行同步，有了这个信号，可以解决输入采集 IP 和 vid_in IP 数据接口之间的同步问题).

OV_Sensor_ML IP 包含 3 个源程序文件，分别为 OV_Sensor_ML.v、 cmos_decode_v1.v、 count_reset_v1.v 文件。

OV_Sensor_ML.v 程序中，对 cmos_data_i、 cmos_href_i、 cmos_vsync_i 做了一次寄存器，笔者发现图像效果有所改观。笔者分析，是因为寄存后有利于去除一些毛刺信号，提高了数据的稳定性。

表 3-3-1-1 OV_Sensor_ML 源码 OV_Sensor_ML.v

```
'timescale 1ns / 1ps
///////////////////////////////
// Company: milinker
// Engineer:tangjinyuan
//
// Create Date:    15:54:59 11/21/2015
// Design Name:
// Module Name:    OV7725_IP_ML
// Project Name: OV7725_IP_ML
// Target Devices: ZYNQ
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////
module OV_Sensor_ML(
    input CLK_i,
    //----- CMOS sensor hardware interface -----
    input      cmos_vsync_i,//cmos vsync
    input      cmos_href_i, //cmos hsync refrence
    input      cmos_pclk_i, //cmos pxiel clock
    output     cmos_xclk_o, //cmos externl clock
    input[7:0]cmos_data_i, //cmos data
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    // output de_o,//data enable.
    output [23:0]rgb_o,//data output,
    output vid_clk_ce
);

```

```
//-----视频输出解码模块-----//
wire [15:0]rgb_o_r;
assign rgb_o = {rgb_o_r[4:0] ,3'd0 ,rgb_o_r[10:5] ,2'd0,rgb_o_r[15:11],3'd0};

reg [7:0]cmos_data_r;
reg cmos_href_r;
reg cmos_vsync_r;

always@(posedge cmos_pclk_i)
begin
    cmos_data_r <= cmos_data_i;
    cmos_href_r <= cmos_href_i;
    cmos_vsync_r<= cmos_vsync_i;
end
//assign rgb_o = 24'b11111111_00000000_11111111;
cmos_decode cmos_decode_u0(
    //system signal.
    .cmos_clk_i(CLK_i),//cmos senseor clock.
    .rst_n_i(RESETn_i2c),//system reset.active low.
    //cmos sensor hardware interface.
    .cmos_pclk_i(cmos_pclk_i),//(cmos_pclk),//input pixel clock.
    .cmos_href_i(cmos_href_r),//(cmos_href),//input pixel hs signal.
    .cmos_vsync_i(cmos_vsync_r),//(cmos_vsync),//input pixel vs signal.
    .cmos_data_i(cmos_data_r),//(cmos_data),//data.
    .cmos_xclk_o(cmos_xclk_o),//(cmos_xclk),//output clock to cmos sensor.
    //user interface.
    .hs_o(hs_o),//hs signal.
    .vs_o(vs_o),//vs signal.
    // .de_o(de_o),//data enable.
    .rgb565_o(rgb_o_r),//data output
    .vid_clk_ce(vid_clk_ce)
);

count_reset_v1#(
    .num(20'hffff0)
)(
    .clk_i(CLK_i),
    .rst_o(RESETn_i2c)
);

endmodule
```

1 cmos_decode_v1.v 是本模块的关键部分,实现了RGB565 的解码输出以及 vid_clk_ce 实现了此模块和 vid_in IP 直接时序匹配的关系。

表 3-3-1-2 OV_Sensor_ML 源码 cmos_decode_v1.v

```
'timescale 1ns / 1ps
///////////////////////////////
// Company: milinker corporation
// WEB:www.milinker.com
// BBS:www.osrc.cn
// Engineer:..
// Create Date:    07:28:50 09/04/2015
// Design Name:   cmos_decode_v1
// Module Name:   cmos_decode_v1
// Project Name:  cmos_decode_v1
// Target Devices: XC6SLX25-FTG256 Mis603
// Tool versions: ISE14.7
// Description:   cmos_decode_v1.
// Revision:      V1.0
// Additional Comments:
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low
//4) _dg debug signal
//5) _r reg delay
//6) _s state machine
/////////////////////////////
module cmos_decode(
    //system signal.
    input cmos_clk_i,//cmos senseor clock.
    input rst_n_i,//system reset.active low.
    //cmos sensor hardware interface.
    input cmos_pclk_i,//input pixel clock.
    input cmos_href_i,//input pixel hs signal.
    input cmos_vsync_i,//input pixel vs signal.
    input[7:0]cmos_data_i,//data.
    output cmos_xclk_o,//output clock to cmos sensor.
    //user interface.
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    output reg [15:0]rgb565_o,//data output
    output vid_clk_ce
);
parameter[5:0]CMOS_FRAME_WAITCNT = 4'd15;
```

```
reg[4:0] rst_n_reg = 5'd0;
//reset signal deal with.
always@(posedge cmos_clk_i)
begin
    rst_n_reg <= {rst_n_reg[3:0],rst_n_i};
end

reg[1:0]vsync_d;
reg[1:0]href_d;
wire vsync_start;
wire vsync_end;
//vs signal deal with.
always@(posedge cmos_pclk_i)
begin
    vsync_d <= {vsync_d[0],cmos_vsync_i};
    href_d  <= {href_d[0],cmos_href_i};
end

assign vsync_start =  vsync_d[1]&(!vsync_d[0]);
assign vsync_end   = (!vsync_d[1])&vsync_d[0];

reg[6:0]cmos_fps;
//frame count.
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        begin
            cmos_fps <= 7'd0;
        end
    else if(vsync_start)
        begin
            cmos_fps <= cmos_fps + 7'd1;
        end
    else if(cmos_fps >= CMOS_FRAME_WAITCNT)
        begin
            cmos_fps <= CMOS_FRAME_WAITCNT;
        end
end
//wait frames and output enable.
reg out_en;
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
```

```
begin
    out_en <= 1'b0;
end
else if(cmos_fps >= CMOS_FRAME_WAITCNT)
begin
    out_en <= 1'b1;
end
else
begin
    out_en <= out_en;
end
end

//output data 8bit changed into 16bit in rgb565.
reg [7:0] cmos_data_d0;
reg [15:0] cmos_rgb565_d0;
reg byte_flag;

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        byte_flag <= 0;
    else if(cmos_href_i)
        byte_flag <= ~byte_flag;
    else
        byte_flag <= 0;
end

reg byte_flag_r0;
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        byte_flag_r0 <= 0;
    else
        byte_flag_r0 <= byte_flag;
end

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        cmos_data_d0 <= 8'd0;
    else if(cmos_href_i)
        cmos_data_d0 <= cmos_data_i; //MSB -> LSB
```

```

else if(~cmos_href_i)
    cmos_data_d0 <= 8'd0;
end

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        rgb565_o <= 16'd0;
    else if(cmos_href_i&byte_flag)
        rgb565_o <= {cmos_data_d0,cmos_data_i}; //MSB -> LSB
    else if(~cmos_href_i)
        rgb565_o <= 8'd0;
end

assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
assign vs_o = out_en ? vsync_d[1] : 1'b0;
assign hs_o = out_en ? href_d[1] : 1'b0;

assign cmos_xclk_o = cmos_clk_i;

endmodule

```

count_reset_v1.v 源文件实现了信号的延迟复位。

表 3-3-1-1 count_reset_v1.v

```

`timescale 1ns / 1ps
///////////////////////////////
// Company: milinker corporation
// WEB:www.milinker.com
// BBS:www.osrc.cn
// Engineer:sanliuyaoling.
// Create Date: 07:28:50 12/04/2015
// Design Name: count_reset_v1
// Module Name: count_reset_v1
// Project Name: count_reset_v1
// Target Devices: XC7Z020-CLG484-1I
// Tool versions: vivado2015.4
// Description: count_reset_v1
// Revision: V1.0
// Additional Comments:
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low

```

```
//4) _dg debug signal
//5) _r  reg delay
//6) _s state machine
///////////////////////////////
module count_reset_v1#
(
    parameter[19:0]num = 20'hffff0
)(
    input clk_i,
    output rst_o
);

reg[19:0] cnt = 20'd0;
reg rst_d0;

/*count for clock*/
always@(posedge clk_i)
begin
    cnt <= ( cnt <= num)?( cnt + 20'd1 ):num;
end

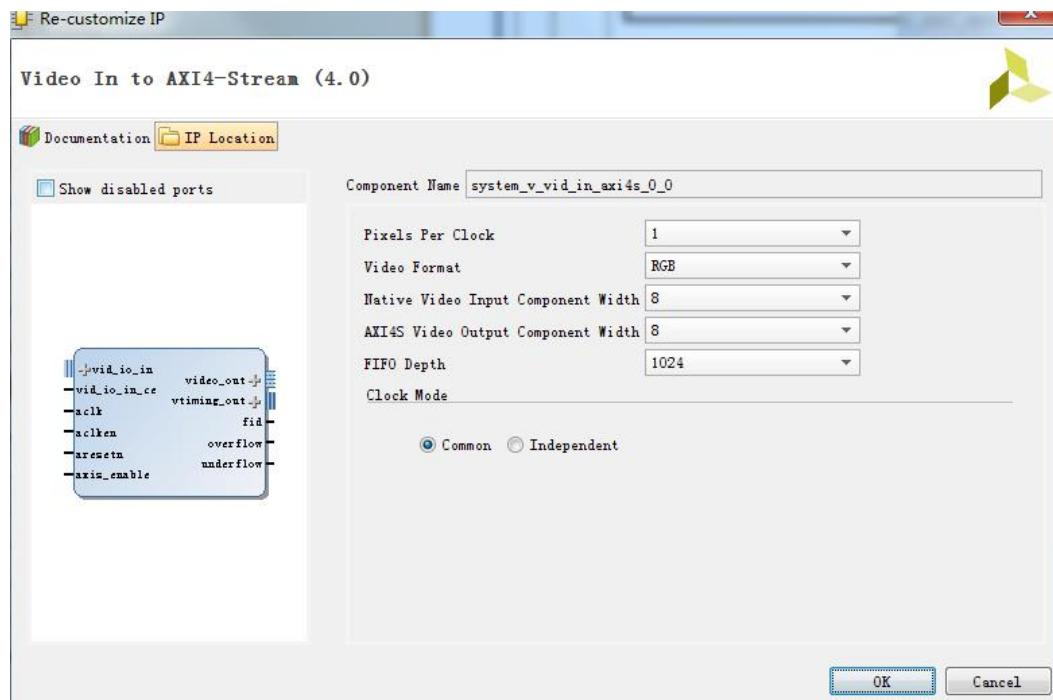
/*generate output signal*/
always@(posedge clk_i)
begin
    rst_d0 <= ( cnt >= num)?1'b1:1'b0;
end

assign rst_o = rst_d0;

endmodule
```

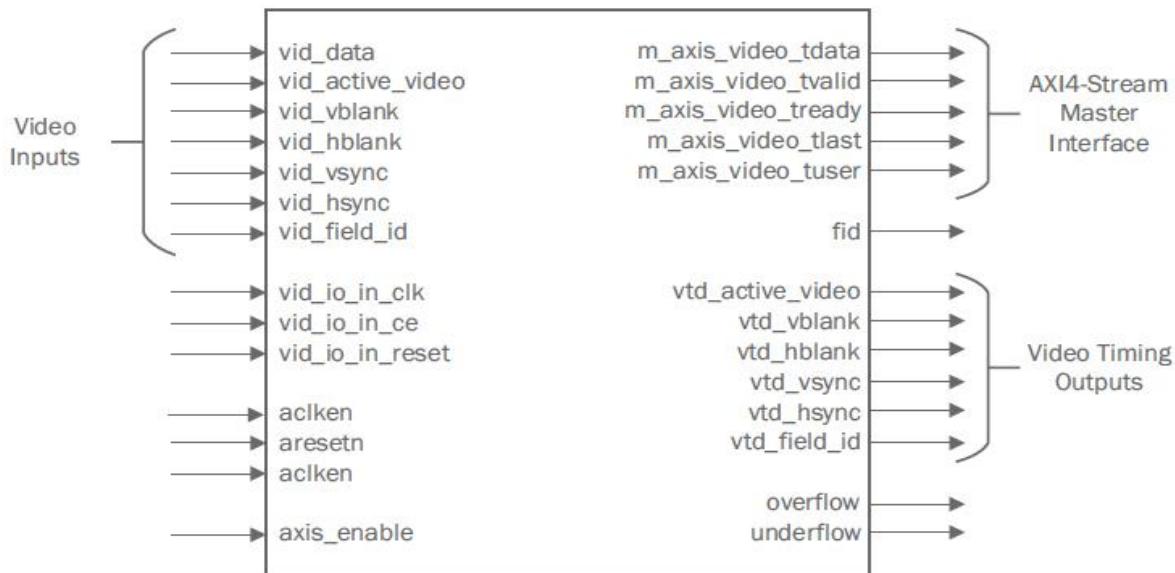
表 3-3-1-2

3.3.2 vid in IP 模块



- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Video Format: 视频格式
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- FIFO Depth: FIFO 深度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟

3.3.2 VID_IN IP 接口信号的定义



Common Interface

Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
axis_enable	Input	1	This input should be connected to the VTC detector locked status and is synchronous to vid_io_in_clk. 1 = Enable writes into FIFO 0 = Disable writes into FIFO
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_in_clk	Input	1	Native video clock. Only available in independent clock mode.
vid_io_in_ce	Input	1	Native video clock enable
vid_io_in_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk. If an overflow occurs, this could indicate that the connected AXI4-Stream Slave is creating excessive back-pressure.
underflow	Output	1	Flag indicating that the FIFO has under-flowed. This should never occur under normal operation. Synchronous to aclk.

Video Timing Interface

Signal Name	Direction	Width	Description
vtd_vsync	Out	1	Vertical sync video timing signal.
vtd_hsync	Out	1	Horizontal sync video timing signal.
vtd_vblank	Out	1	Vertical blank video timing signal.
vtd_hblank	Out	1	Horizontal blank video timing signal.
vtd_active_video	Out	1	Active video flag. 1 = active video, 0 = blanked video
vtd_field_id	Out	1	VTC field ID. 0= even field, 1= odd field.

Video Input Interface

Signal Name	Direction	Width	Description
vid_active_video	In	1	Video data valid. 1 = active video, 0 = blanked video
vid_vsync	In	1	Vertical sync video timing signal. Active High
vid_hsync	In	1	Horizontal sync video timing signal. Active High
vid_vblank	In	1	Vertical blank video timing signal. Active High
vid_hblank	In	1	Horizontal blank video timing signal. Active High
vid_data	In	8-256	Parallel video input data.
vid_field_id	In	1	Video field. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

使用到的信号有：

Vid_in IP 输入端信号：

Vid_data: 视频数据输入

Vid_active_video: 视频数据有效

Vid_hsync: 视频行同步信号 (非常关键信号, 下面重点分析对象)

Vid_vsync: 视频场同步信号

Vid_io_in_ce: 数据输入有效 (非常关键信号, 下面重点分析对象)

vid_io_in_clk: 这是时钟信号和摄像头时钟同步

Vid_io_in_reset: 这个信号, 高电平的时候复位

有很多读者会问笔者, 这些官方的 IP 如何使用, 这么没有详细的技术手册。还别说, 官方就是没有非常详细的技术手册, 有时候笔者也是使出浑身解数分析 vid_in IP 内部信号时序, 掌握 OV_Sensor_ML 自定义 IP 时序接口设计。

打开 v_vid_in_axi4s_v4_0_1_formatter.v 这个文件

下面对其关键的部分进行说明。

表 3-3-2-1 v_vid_in_axi4s_v4_0_1_formatter.v

```
'timescale 1ps/1ps
`default_nettype none
(* DowngradeIPIdentifiedWarnings="yes" *)
```

```
module v_vid_in_axi4s_v4_0_1_formatter #(
    parameter C_NATIVE_DATA_WIDTH = 24
) (
    // System signals
    input wire VID_IN_CLK,           // Native video clock
    input wire VID_RESET,            // Native video reset
    input wire VID_CE,               // Native video clock enable

    // Video input signals
    input wire VID_ACTIVE_VIDEO,     // Native video input data enable
    input wire VID_VBLANK,           // Native video input vertical blank
    input wire VID_HBLANK,           // Native video input horizontal blank
    input wire VID_VSYNC,             // Native video input vertical sync
    input wire VID_HSYNC,             // Native video input horizontal sync
    input wire VID_FIELD_ID,         // Native video input field-id
    input wire [C_NATIVE_DATA_WIDTH-1:0] VID_DATA, // Native video input data

    // Video timing detector signals
    output wire VTD_ACTIVE_VIDEO,    // Native video output data enable
    output wire VTD_VBLANK,           // Native video output vertical blank
    output wire VTD_HBLANK,           // Native video output horizontal blank
    output wire VTD_VSYNC,             // Native video output vertical sync
    output wire VTD_HSYNC,             // Native video output horizontal sync
    output wire VTD_FIELD_ID,         // Native video output field-id
    input wire VTD_LOCKED,            // Native video locked signal from VTD

    // FIFO write signals
    output wire [C_NATIVE_DATA_WIDTH+2:0] FIFO_WR_DATA, // FIFO write data
    output wire FIFO_WR_EN           // FIFO write enable
);

    // Wire and register declarations
    reg de_1 = 0;
    reg vblank_1 = 0;
    reg hblank_1 = 0;
    reg vsync_1 = 0;
    reg hsync_1 = 0;
    reg [C_NATIVE_DATA_WIDTH -1:0] data_1 = 0;
    reg de_2 = 0;
    reg v_blank_sync_2 = 0;
    reg [C_NATIVE_DATA_WIDTH -1:0] data_2 = 0;
    reg de_3 = 0; // DE output register
```

```
reg [C_NATIVE_DATA_WIDTH-1:0] data_3 = 0; // data output register
reg vert_blankning_intvl = 0; // SR, reset by DE rising
reg field_id_1 = 0;
reg field_id_2 = 0;
reg field_id_3 = 0;

wire v_blank_sync_1; // vblank or vsync
wire de_rising;
wire de_falling;
wire vsync_rising;
reg sof;
reg sof_1;
reg eol;
reg vtd_locked;
wire sof_rising;

// Assignments
assign FIFO_WR_DATA      = {field_id_3,sof_1,eol,data_3};
assign FIFO_WR_EN         = de_3 & ~VID_RESET & vtd_locked;
assign VTD_ACTIVE_VIDEO = de_1;
assign VTD_VBLANK        = vblank_1;
assign VTD_HBLANK        = hblank_1;
assign VTD_VSYNC          = vsync_1;
assign VTD_HSYNC          = hsync_1;
assign VTD_FIELD_ID      = field_id_1;

assign v_blank_sync_1 = vblank_1 || vsync_1;
assign de_rising   = de_1 && !de_2;
assign de_falling  = !de_1 && de_2;
assign vsync_rising = v_blank_sync_1 && !v_blank_sync_2;
assign sof_rising  = sof & ~sof_1;

// VTD locked process
always @(posedge VID_IN_CLK) begin
    if(VID_RESET | ~VTD_LOCKED) begin
        vtd_locked <= 1'b0;
    end else if(VID_CE) begin
        vtd_locked <= (sof_rising & VTD_LOCKED) ? 1'b1 : vtd_locked;
    end
end

// input, output, and delay registers
always @ (posedge VID_IN_CLK) begin
```

```

if(VID_RESET) begin
    de_1          <= 1'b0;
    de_2          <= 1'b0;
    de_3          <= 1'b0;
    vblank_1      <= 1'b0;
    hblank_1      <= 1'b0;
    vsync_1       <= 1'b0;
    hsync_1       <= 1'b0;
    field_id_1    <= 1'b0;
    field_id_2    <= 1'b0;
    field_id_3    <= 1'b0;
    data_1         <= {C_NATIVE_DATA_WIDTH{1'b0}};
    data_2         <= {C_NATIVE_DATA_WIDTH{1'b0}};
    data_3         <= {C_NATIVE_DATA_WIDTH{1'b0}};
    v_blank_sync_2 <= 1'b0;
    eol           <= 1'b0;
    sof            <= 1'b0;
    sof_1          <= 1'b0;
end else if(VID_CE) begin
    de_1          <= VID_ACTIVE_VIDEO;
    de_2          <= de_1;
    de_3          <= de_2;
    vblank_1      <= VID_VBLANK;
    hblank_1      <= VID_HBLANK;
    vsync_1       <= VID_VSYNC;
    hsync_1       <= VID_HSYNC;
    field_id_1    <= VID_FIELD_ID;
    field_id_2    <= field_id_1;
    field_id_3    <= field_id_2;
    data_1         <= VID_DATA;
    data_2         <= data_1;
    data_3         <= data_2;
    v_blank_sync_2 <= v_blank_sync_1;
    eol           <= de_falling;
    sof            <= de_rising && vert_blinking_intvl;
    sof_1          <= sof;
end
end

// Vertical back porch SR register
always @ (posedge VID_IN_CLK) begin
    if(VID_CE) begin
        if(vsync_rising) // falling edge of vsync

```

```

    vert_blanking_intvl <= 1;
else if (de_rising)          // rising edge of data enable
    vert_blanking_intvl <= 0;
end
end

endmodule

```

在上面代码中，

```

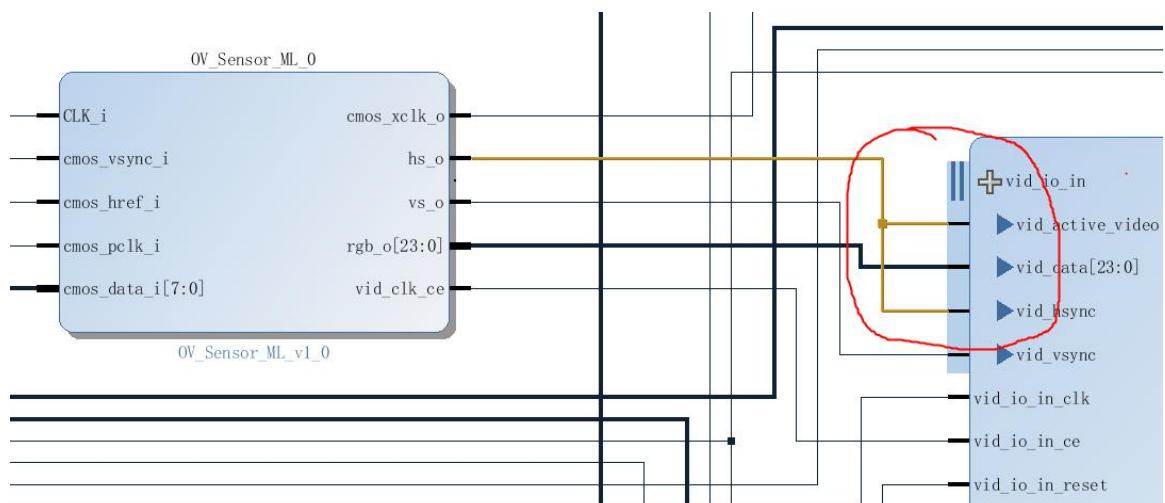
eol      <= de_falling;
sof      <= de_rising && vert_blanking_intvl;

```

eol 实际就是 tlast 信号，而 sof 就是 tuser 信号。tlast 信号代表每行图像数据的最后一个数据，tuser 代表每场数据的第一个数据。

所有非常关键的信号都和 de_falling 和 vert_blanking_intvl 有关系。

hs_o 和 vid_in IP 的连接关系。



上图中，被红色圈起来的 hs_o 信号，同时接到了 vid_in_ip 的 vid_active_video 和 vid_hsync 信号接口。因此，de 信号就是 hs_o 信号，而 vid_hsync 我们发现没有任何作用，也就是说不 hs_o 不连接到 vid_hsync 也不影响这里的程序工作。

VID_CE 这个参数就是前面的 vid_io_in_ce 信号，可以看出这个芯片有效的时候相对应的时序电路才会执行。在本工程中，摄像头每 2 个 pclk 输出 1 个有效的数据，而 vid_in IP 如果 VID_CE 为 1 则数据输入会每个时钟输入 1 个就错了。因此官方的 IP 设计的还是很不错考虑周到，通过 VID_CE 这个条件，控制时钟同步。

```

...
else if(VID_CE) begin
...
end
...

```

现在回到 OV_Sensor_ML 的 cmos_decode_v1.v 文件中有一段红色的代码如下：

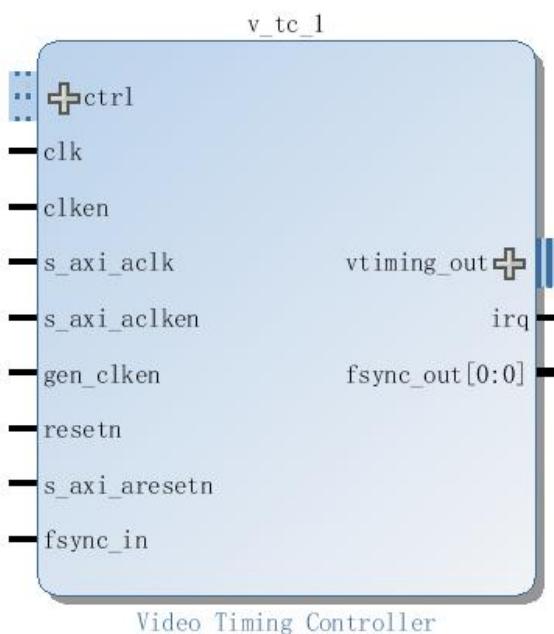
```
assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
```

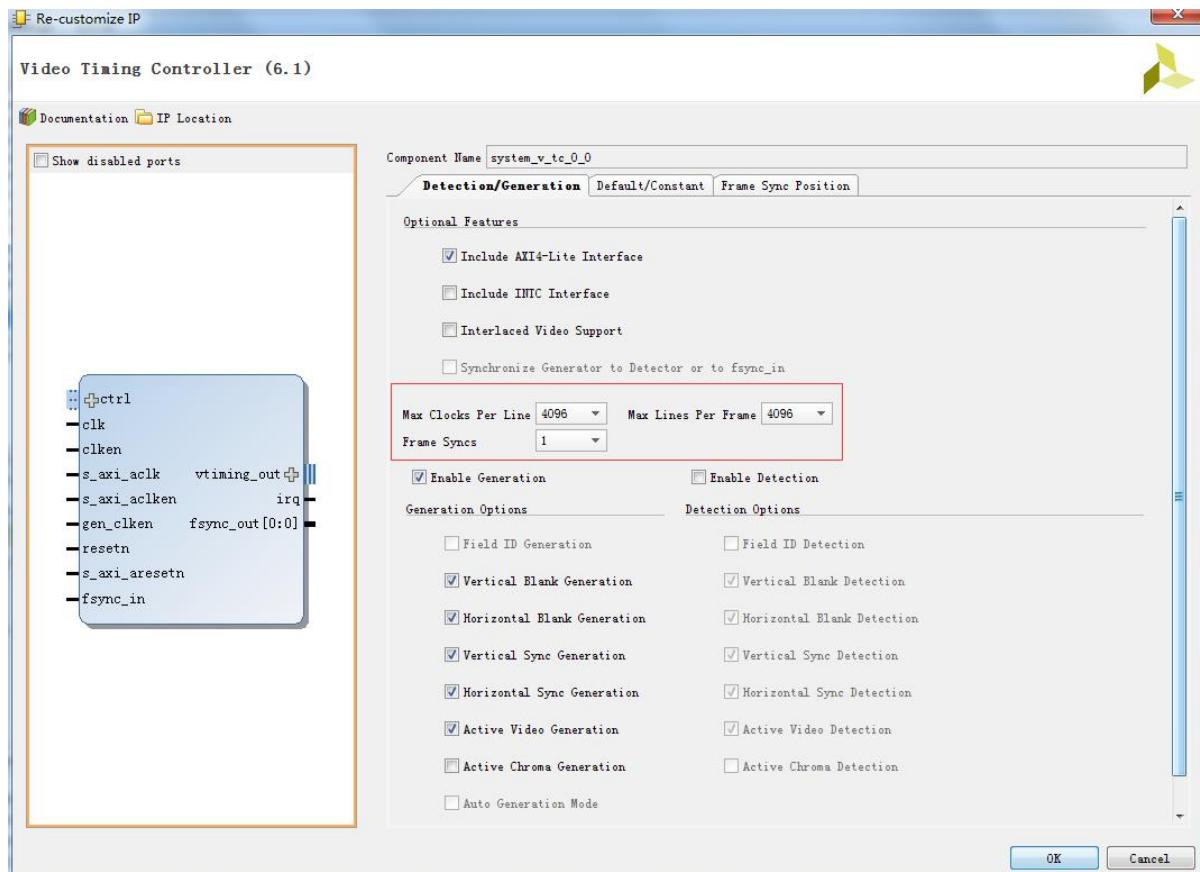
这段代码控制了 vid_clk_ce 的正确输出，关键部分是 `(byte_flag_r0&hs_o)||(!hs_o)`。当 hs_o 有效的时候，vid_in 的 VID_CE 信号就有效，当 hs_o=0 的时候 VID_CE 必须仍然有效，这样才能检测到 vsync_rising 信号了，检测到了 vsync_rising 才能有 assert_blanking_intvl 为 1，才有 tuser 信号。

好了罗嗦了半天，终于解释完了，如果有不清楚的，找我们技术支持吧。

3.4 VTC IP 的分析

3.4.1 VTC IP 的参数介绍





这个 IP 就是一个时序发生器，产生显示器输出所需要的时序信号。

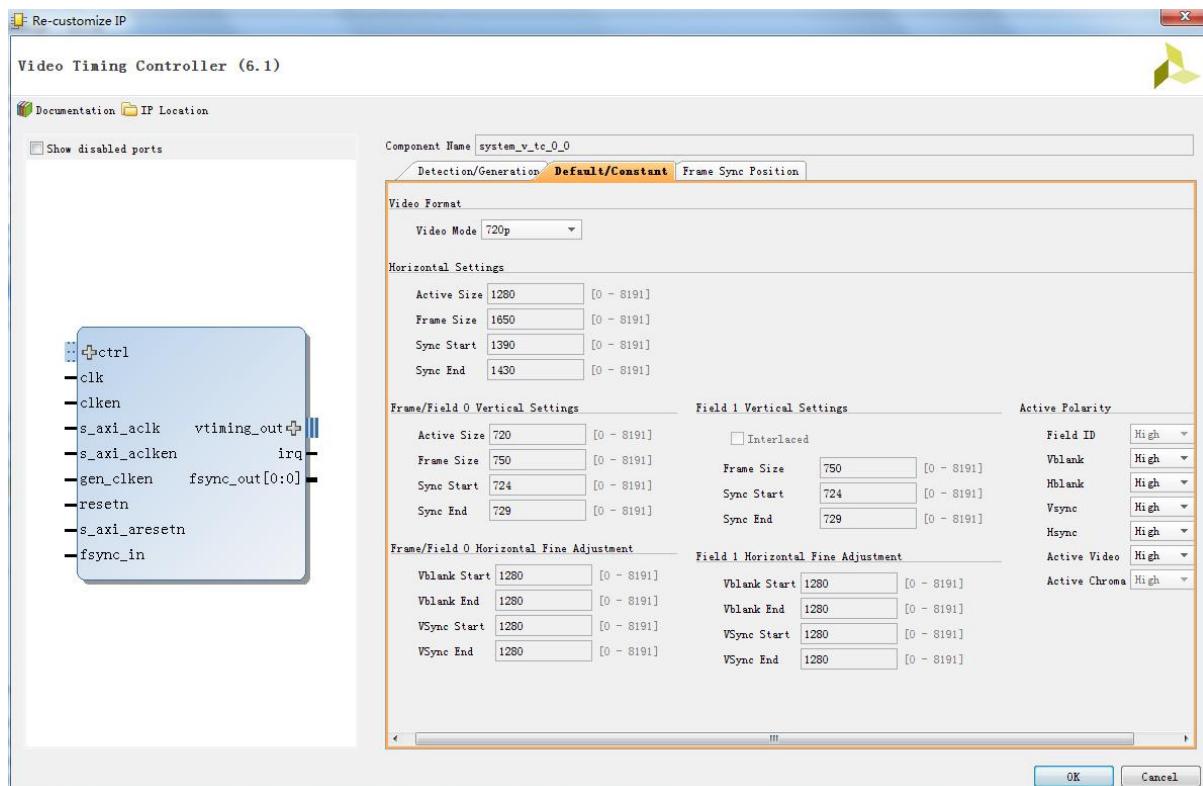
这个页面中，include AXI-lite interface 可以不勾选，不勾选就只能采用默认设置，无法在 C 语言中灵活配置了，所以笔者这里建议大家勾选吧。max clocks per line 和 max_lines per frame 需要设置下，当设置到 4096 的时候可以支持分辨率到最大，当然消耗的资源也更多。笔者这里太奢侈了设置了 4096。实际上设置到 2048 就够用了。本页面的其他信号可以采取默认设置。

Enable Generation:

支持产生时序，这个肯定是必须勾选的。

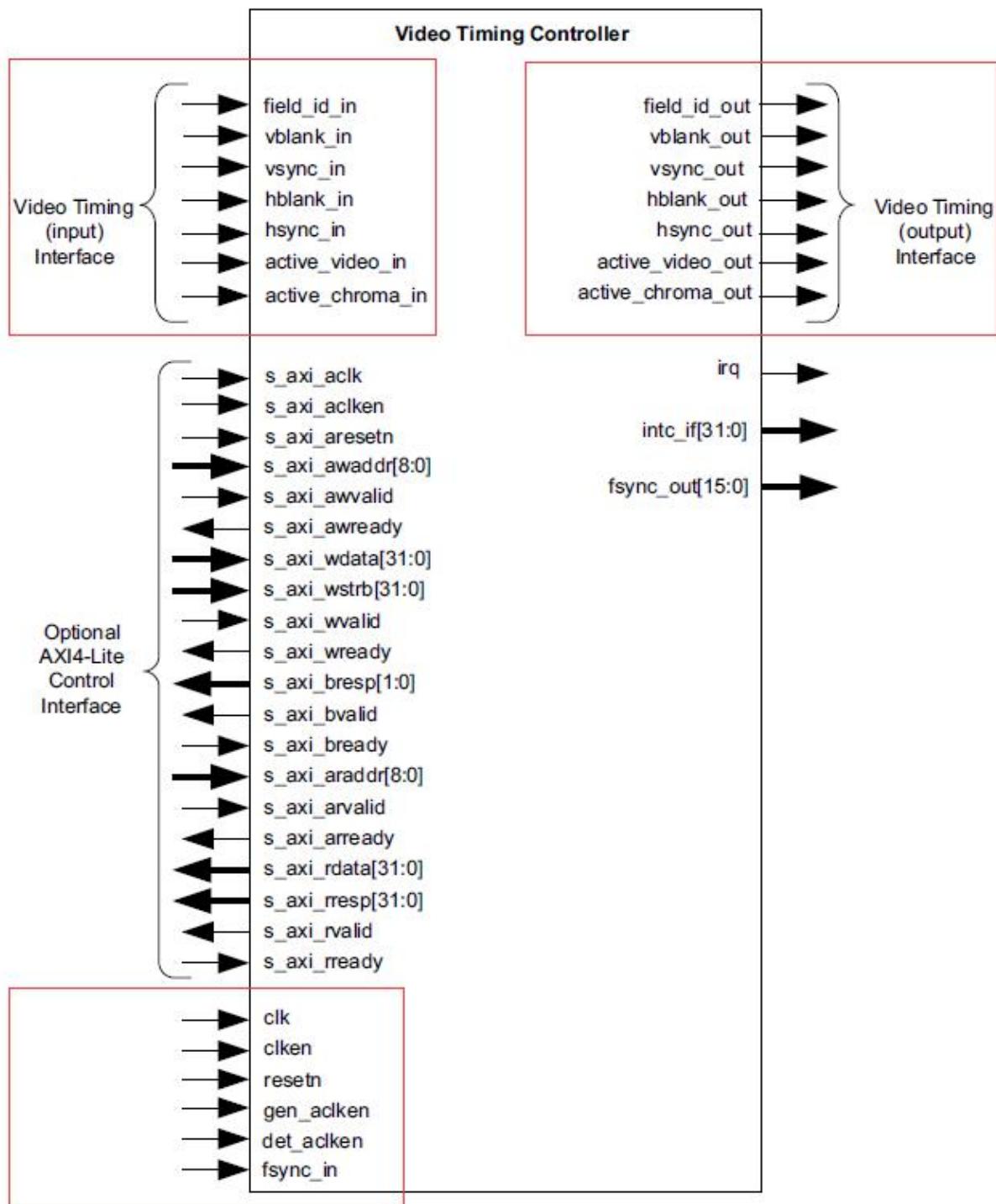
Enable Detection:

支持时序捕捉，这个不是必须的，根据需要而定，如果设置了这个选项，就可以先捕捉输入的时序，然后再设置输出的时序，实现输入和输出一致的效果。



在这个页面中，只要选择需要支持的分辨率就可以了，当然不设置也没关系的，因为我们在 C 代码综合那个会进一步设置的。

3.4.2 VTC IP 接口信号的定义



红色方框内的绝大部分信号需要我们手动联系，所以下面重点是讲解红色方框内的信号作用，至于 AXI4-LITE 接口主要是用来设置参数的。

Common Port Descriptions:

Name	Direction	Width	Description
clk	In	1	Video Core Clock
clken	In	1	Video Core Active High Clock Enable
det_clken	In	1	Video Timing Detection Core Active High Clock Enable
gen_clken	In	1	Video Timing Generator Core Active High Clock Enable
resetn	In	1	Video Core Active Low Synchronous Reset
irq	Output	1	Interrupt request output, active high edge
intc_if	Output	32	OPTIONAL EXTERNAL INTERRUPT CONTROLLER INTERFACE Available when the "Include INTC Interface" or C_HAS_INTC_IF has been selected. Bits [31:8] are the same as the bits [31:8] in the status register (0x0004). Bits [5:0] are the same as bits [21:16] of the error register (0x0008). Bits [7:6] are reserved and are always 0.

Detector Interface (Video Timing Input Interface)			
field_id_in	Input	1	INPUT FIELD ID Used to set the field_id polarity in the Detector Polarity Register (Address Offset 0x002C). Optional. Only valid when interlace support and field id are enabled.
hsync_in	Input	1	INPUT HORIZONTAL SYNCHRONIZATION Used to set the DETECTOR HSYNC register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hsync_in input is not connected, then the "Horizontal Sync Detection" option must be deselected.
hblank_in	Input	1	INPUT HORIZONTAL BLANK Used to set the DETECTOR HSIZE register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hblank_in input is not connected, then the "Horizontal Blank Detection" option must be deselected.
vsync_in	Input	1	INPUT VERTICAL SYNCHRONIZATION Used to set the DETECTOR F0_VSYNC_V and the F0_VSYNC_H registers. Polarity is auto-detected. Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vsync_in input is not connected, then the "Vertical Sync Detection" option must be deselected.

Name	Direction	Width	Description
vblank_in	Input	1	<p>INPUT VERTICAL BLANK Used to set the DETECTOR_VSIZE and the F0_VBLANK_H registers. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vblank_in input is not connected, then the "Vertical Blank Detection" option must be deselected.</p>
active_video_in	Input	1	<p>INPUT ACTIVE VIDEO Used to set the DETECTOR_ACTIVE_SIZE register. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the active_video_in input is not connected, then the "Active Video Detection" option must be deselected.</p>
active_chroma_in	Input	1	<p>INPUT ACTIVE CHROMA Used to set the VIDEO_FORMAT and the CHROMA_PARITY bits in the Detector Encoding Register. Polarity is auto-detected.</p> <p>Optional. If the active_chroma_in input is not connected, then the "Active Chroma Detection" option must be deselected.</p>

Generator Interface (Video Timing Output Interface)			
field_id_out	Output	1	<p>OUTPUT FIELD ID Generated field id signal. Polarity configured by the Generator Polarity Register (Address Offset 0x006C) Optional. Only enabled when interlaced support and field id generation is enabled.</p>
hsync_out	Output	1	<p>OUTPUT HORIZONTAL SYNCHRONIZATION Generated horizontal synchronization signal. Polarity configured by the control register. Asserted active during the cycle set by the HSYNC_START bits and deasserted during the cycle set by the HSYNC_END bits in the GENERATOR HSYNC register.</p>
hblank_out	Output	1	<p>OUTPUT HORIZONTAL BLANK Generated horizontal blank signal. Polarity configured by the control register. Asserted active during the cycle set by ACTIVE_HSIZEx and deasserted during the cycle set by the FRAME_HSIZEx bits in the GENERATOR HSIZEx register.</p>
vsync_out	Output	1	<p>OUTPUT VERTICAL SYNCHRONIZATION Generated vertical synchronization signal. Polarity configured by the control register. Asserted active during the line set by the F#_VSYNC_VSTART bits and deasserted during the line set by the F#_VSYNC_VEND bits in the GENERATOR F#_VSYNC_V registers.</p>

Name	Direction	Width	Description
vblank_out	Output	1	OUTPUT VERTICAL BLANK Generated vertical blank signal. Polarity configured by the control register. Asserted active during the line set by the ACTIVE_VSIZE bits and deasserted during the line set by the GENERATOR VSIZE register.
active_video_out	Output	1	OUTPUT ACTIVE VIDEO Generated active video signal. Polarity configured by the control register. Active for non blanking lines. Asserted active during the first cycle of the field/frame and deasserted during the cycle set by the GENERATOR ACTIVE_SIZE register
active_chroma_out	Output	1	OUTPUT ACTIVE CHROMA Generated active chroma signal. Denotes which lines contain valid chroma samples (used for YUV 4:2:0). Polarity configured by the GENERATOR POLARITY register. Active for non-blanking lines configured by the VIDEO_FORMAT and the CHROMA_PARITY bits in the GENERATOR Encoding Register.
Frame Synchronization Interface			
fsync_out	Output	[Frame Syncs - 1:0]	FRAME SYNCHRONIZATION OUTPUT Each Frame Synchronization bit toggles for only one clock cycle during each frame. The number of bits is configured with the Frame Syncs GUI parameter. Each bit is independently configured for horizontal and vertical clock cycle position with the Frame Sync 0-15 Config registers).
fsync_in	Input	1	FRAME SYNCHRONIZATION INPUT This is a one clock cycle pulse (active high) input. The video timing generator will be synchronized to the input if used.

本例子中没有使用到输入时序的捕捉，因此笔者下面只对用到的信号做一些介绍。

hsync_out:

产生行同步输出

hsync_out:

产生行消影

vsync_out:

产生场同步输出

vblank_out:

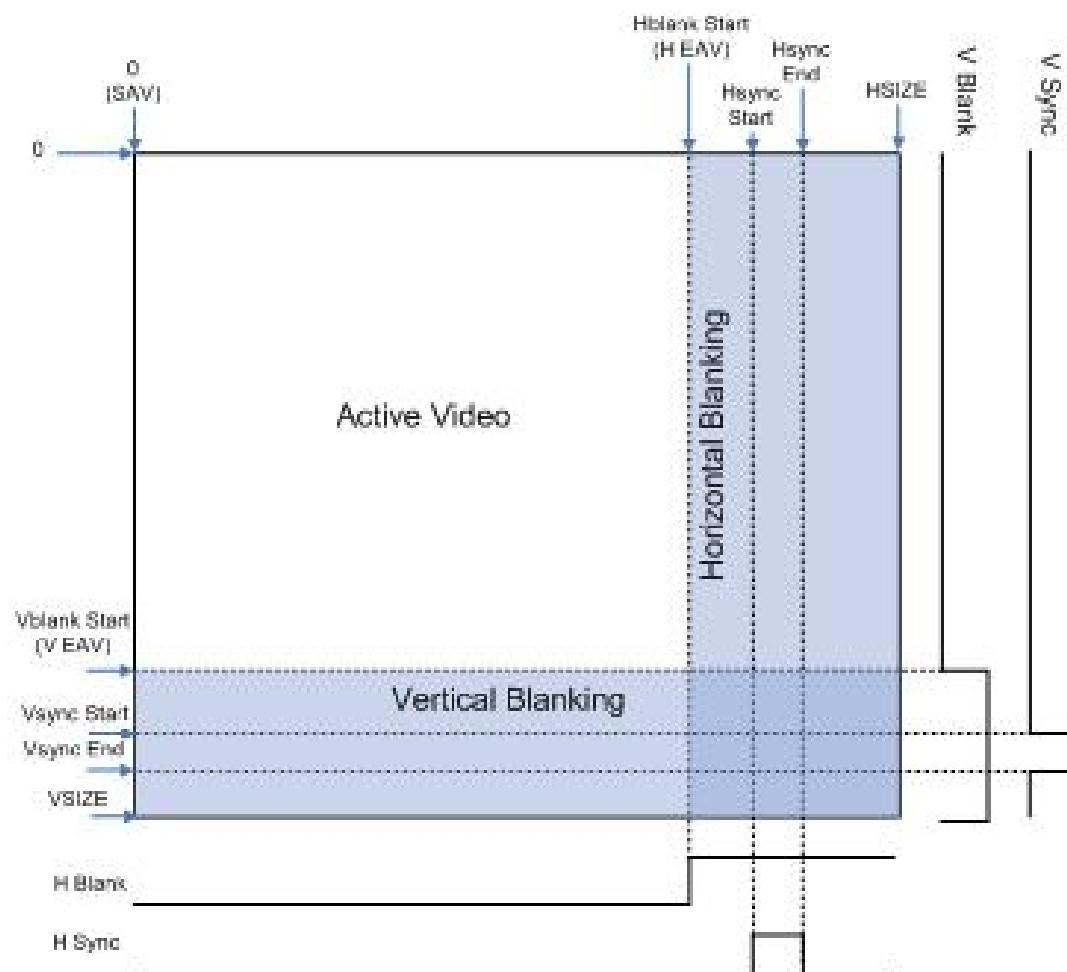
产生场消影

active_video_out:

有效数据输出

3.4.3 VTC IP 配置寄存器

shows the start of the horizontal front porch (Hblank Start), synchronization (Hsync Start), back porch (Hsync End) and active video (SAV). It also shows the start of the vertical front porch (Vblank Start), synchronization (Vsync Start), back porch (Vsync End) and active video (SAV). The total number of horizontal clock cycles is HSIZE and the total number of lines is the VSIZE.



Generator Active Size Register (Address Offset 0x0060)

0x0060	GENERATOR ACTIVE_SIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
ACTIVE_VSIZE	28:16	Generated Vertical Active Frame Size. The height of the frame without blanking in number of lines.
RESERVED	15:13	Reserved
ACTIVE_HSIZE	12:0	Generated Horizontal Active Frame Size. The width of the frame without blanking in number of pixels/clocks.

这是重要的寄存器用来设置有效的行数量和场数量

Generator Timing Status Register (Address Offset 0x0064)

0x0064	GENERATOR TIMING_STATUS	Read
Name	B its	Description
RESERVED	31:3	Reserved
GEN_ACTIVE_VIDEO	2	Generated Active Video Interrupt Status. Set high during the first cycle the output active video is asserted.
GEN_VBLANK	1	Generated Vertical Blank Interrupt Status. Set high during the first cycle the output vertical blank is asserted.
RESERVED	0	Reserved

GEN_ACTIVE_VIDEO:当第一帧图像输出时候置 1

GEN_VBLANK:第一帧有效图像的 blank 信号输出的时候置 1

Generator Encoding Register (Address Offset 0x0068)

0x0068	GENERATOR ENCODING	Read/Write
Name	B its	Description
RESERVED	31:10	Reserved
CHROMA_PARITY	9:8	Generated Chroma Parity 0: Chroma Active during even active-video lines of frame. Active every pixel of active line 1: Chroma Active during odd active-video lines of frame. Active every pixel of active line 2: Chroma Active during even active video lines of frame. Active every even pixel of active line, inactive every odd pixel 3: Chroma Active during odd active video lines of frame. Active every even pixel of active line, inactive every odd pixel
FIELD_ID_PARITY	7	Generated Field ID Parity 0: Field ID input is currently low 1: Field ID input is currently high
INTERLACED	6	Generated Progressive/Interlaced 0: Generated video format is progressive 1: Generated video format is interlaced
RESERVED	5:4	Reserved
VIDEO_FORMAT	3:0	Generated Video Format Denotes when the active_chroma signal is active. 0: YUV 4:2:2 - Active_chroma is active during the same time active_video is active. 1: YUV 4:4:4 - Active_chroma is active during the same time active_video is active. 2: RGB - Active_chroma is active during the same time active_video is active. 3: YUV 4:2:0- Active_chroma is active every other line during the same time active_video is active. See The CHROMA_PARITY bits to control which lines and pixels.

CHROMA_PARITY: 奇偶色度 (读者没明白)

FIELD_ID_PARITY: 奇偶场标志

INTERLACED: 视频格式是渐进式还是各行扫描

VIDEO_FORMAT: 视频格设置, 有 YUV422 YUV444 YUV420 RGB

Generator Polarity Register (Address Offset 0x006C)

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
RESERVED	31:7	Reserved
FIELD_ID_POL	6	Generated Field ID Polarity 0: Low during Field 0 and High during Field 1 1: High during Field 0 and Low during Field 1
ACTIVE_CHROMA_POL	5	Generated Active Chroma Polarity 0: Active Low Polarity 1: Active High Polarity

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
ACTIVE_VIDEO_POL	4	Generated Active Video Polarity 0: Active Low Polarity 1: Active High Polarity
HSYNC_POL	3	Generated Horizontal Sync Polarity 0: Active Low Polarity 1: Active High Polarity
VSYNC_POL	2	Generated Vertical Sync Polarity 0: Active Low Polarity 1: Active High Polarity
HBLANK_POL	1	Generated Horizontal Blank Polarity 0: Active Low Polarity 1: Active High Polarity
VBLANK_POL	0	Generated Vertical Blank Polarity 0: Active Low Polarity 1: Active High Polarity

这个寄存器设置相应的场输出极性和色度输出极性。

Generator Horizontal Frame Size Register (Address Offset 0x0070)

0x0070	GENERATOR HSIZE	Read/Write
Name	B its	Description
RESERVED	31:13	Reserved
FRAME_HSIZE	12:0	Generated Horizontal Frame Size. The width of the frame with blanking in number of pixels/clocks.

一副图像的一行的大小，包括了消隐和有效数据阶段。

Generator Vertical Frame Size Register (Address Offset 0x0074)

0x0074	GENERATOR VSIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
FIELD1_VSIZE	28:16	Generated Vertical Field 1 Size. The height with blanking in number of lines of field 1.
FRAME_VSIZE	12:0	Generated Vertical Frame Size. The height of the frame with blanking in number of lines.

一副图像的一场的大小，包括了消隐和有效数据阶段。

Generator Horizontal Sync Register (Address Offset 0x0078)

0x0078	GENERATOR HSYNC	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
Hsync_End	28:16	Generated Horizontal Sync End End cycle index of horizontal sync. Denotes the first cycle hsync_in is de-asserted.
RESERVED	15:13	Reserved
Hsync_Start	12:0	Generated Horizontal Sync End Start cycle index of horizontal sync. Denotes the first cycle hsync_in is asserted.

设置行的水平同步结束和同步开始

Generator Frame/Field 0 Vertical Blank Cycle Register (Address Offset 0x007C)

0x007C	GENERATOR F0_VBLANK_H	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VBLANK_HEND	28:16	Generated Vertical Blank Horizontal End End Cycle index of vertical blank. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F0_VBLANK_HSTART	12:0	Generated Vertical Blank Horizontal Start Start Cycle index of vertical blank. Denotes the first cycle vblank_in is asserted.

设置 Fram/Field0 的水平消隐结束和开始

Generator Frame/Field 0 Vertical Sync Line Register (Address Offset 0x0080)

0x0080	GENERATOR F0_VSYNC_V	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VSYNC_VEND	28:16	Generated Vertical Sync Vertical End End Line index of vertical sync. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_VSTART	12:0	Generated Vertical Sync Vertical Start Start line index of vertical sync. Denotes the first line vsync_in is asserted.

设置 Fram/Field0 的垂直同步垂直结束和开始

Generator Frame/Field 0 Vertical Sync Cycle Register (Address Offset 0x0084)

0x0084	GENERATOR F0_VSYNC_H	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
F0_VSYNC_HEND	28:16	Generated Vertical Sync Horizontal End End cycle index of vertical sync. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_HSTART	12:0	Generated Vertical Sync Horizontal Start Start cycle index of vertical sync. Denotes the first cycle vsync_in is asserted.

设置 Fram/Field0 的垂直同步水平结束和开始

Generator Field 1 Vertical Blank Cycle Register (Address Offset 0x0088)

0x0088	GENERATOR F1_VBLANK_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VBLANK_HEND	28:16	Generated Field 1 Vertical Blank Horizontal End End Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F1_VBLANK_HSTART	12:0	Generated Field 1 Vertical Blank Horizontal Start Start Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is asserted.

设置 Field1 的水平消隐结束和开始

Generator Field 1 Vertical Sync Line Register (Address Offset 0x008C)

0x008C	GENERATOR F1_VSYNC_V	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_VEND	28:16	Generated Field 1 Vertical Sync Vertical End End Line index of vertical sync for field 1. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_VSTART	12:0	Generated Field 1 Vertical Sync Vertical Start Start line index of vertical sync for field 1. Denotes the first line vsync_in is asserted.

设置 Field1 的垂直同步垂直结束和开始

Generator Field 1 Vertical Sync Cycle Register (Address Offset 0x0090)

0x0090	GENERATOR F1_VSYNC_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_HEND	28:16	Generated Field 1 Vertical Sync Horizontal End End cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_HSTART	12:0	Generated Field 1 Vertical Sync Horizontal Start Start cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is asserted.

设置 Field1 的垂直同步水平结束和开始

Frame Sync 0 - 15 Configuration Registers (Address Offsets 0x0100 - 0x013C)

0x0100	FRAME SYNC 0 CONFIG	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
V_START	28:16	FRAME SYNCHRONIZATION VERTICAL START Vertical line during which the fsync_out[0] output port is asserted active-high. Note: Frame Syncs are not active during the complete line, only in the cycle during which both the V_START and H_START are valid each frame.
RESERVED	15:13	Reserved
H_START	12:0	FRAME SYNCHRONIZATION HORIZONTAL START Horizontal Cycle during which fsync_out[0] output port is asserted active-high

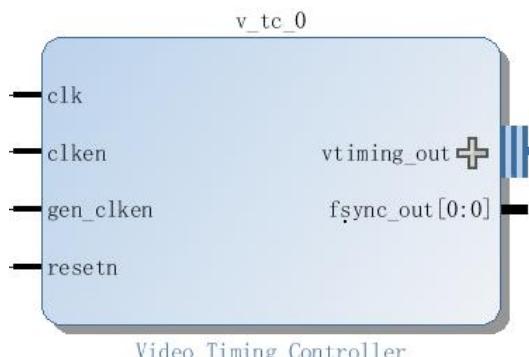
Generator Global Delay Register (Address Offset 0x140)

0x140	Generator Global Delay	Read/Write
Name	Bits	Description
Reserved	31:29	Reserved
V_DELAY	28:16	GENERATOR VERTICAL DELAY Vertical line offset. This is the number of lines that the generated output will be shifted relative to the detector (input timing). The vertical delay is only available when both the detector and generator are enabled. Can be combined with the H_DELAY.

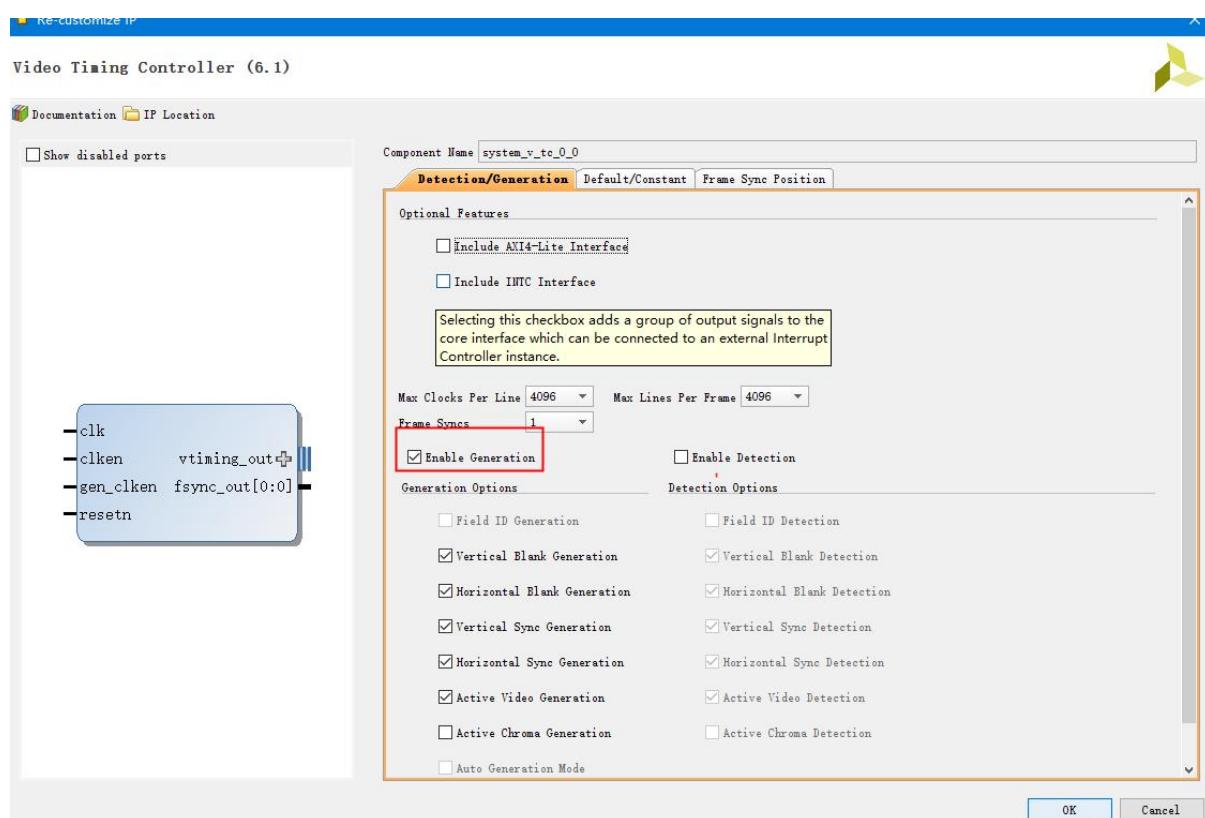
Reserved	15:13	Reserved
H_DELAY	12:0	GENERATOR HORIZONTAL DELAY Horizontal cycle offset. This is the number of clock cycles that the generated output will be shifted relative to the detector (input timing). The horizontal delay is only available when both the detector and generator are enabled. Can be combined with the V_DELAY.

3.4.5 设置 VTC IP

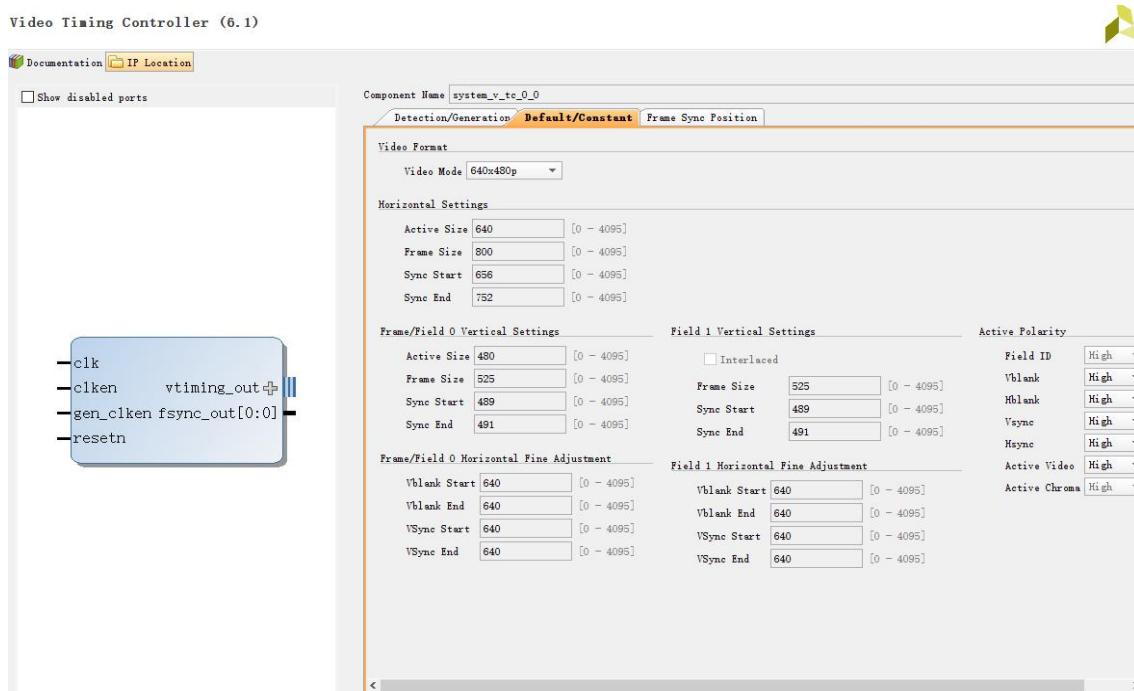
讲了这么多实际上我们用的时候很简单,所以只要这么简单。



由于不使用动态配置，并且只使用了视频时序产生，所以只要勾选如下复选框。

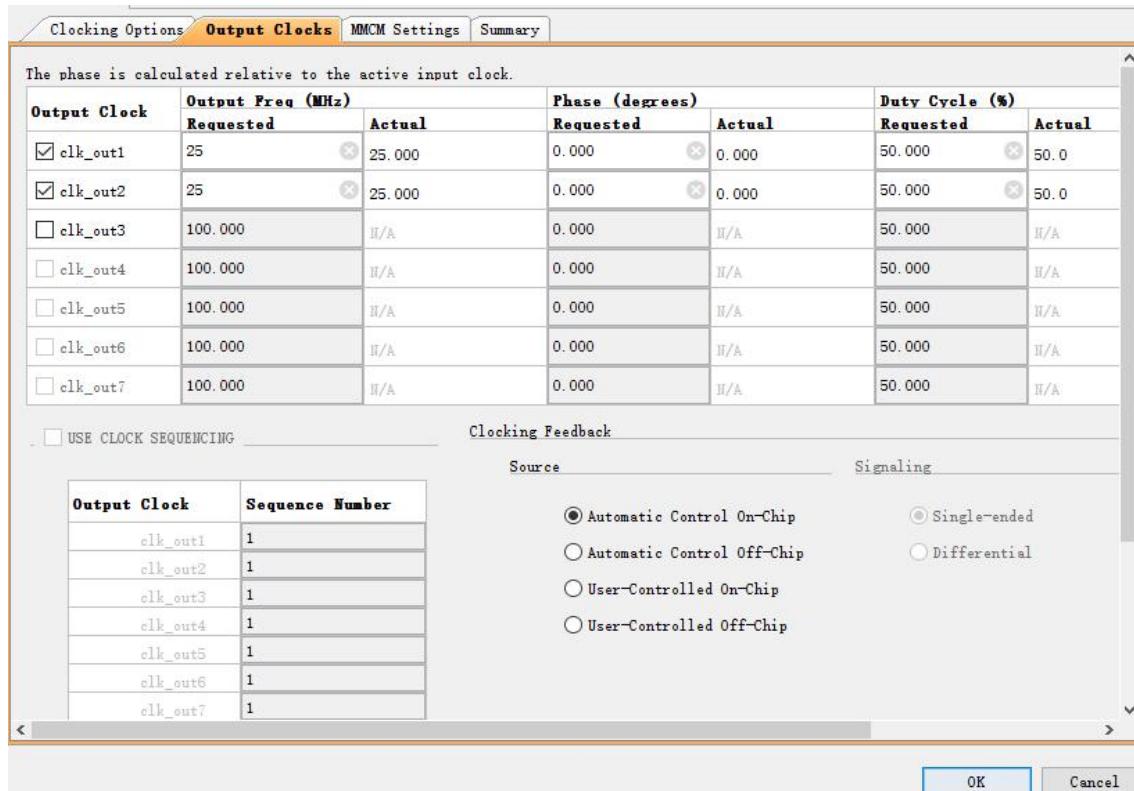


由于 OV7725 分辨率是 640X480 因此直接选择 640PX480 就可以了。



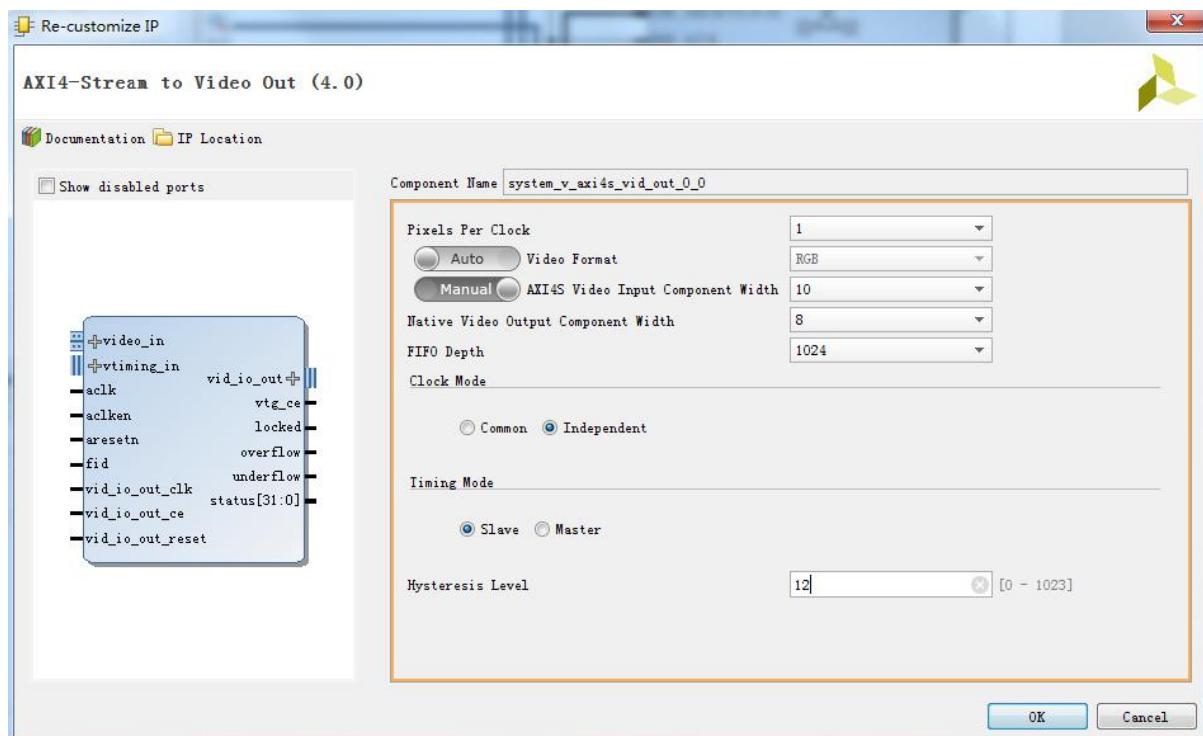
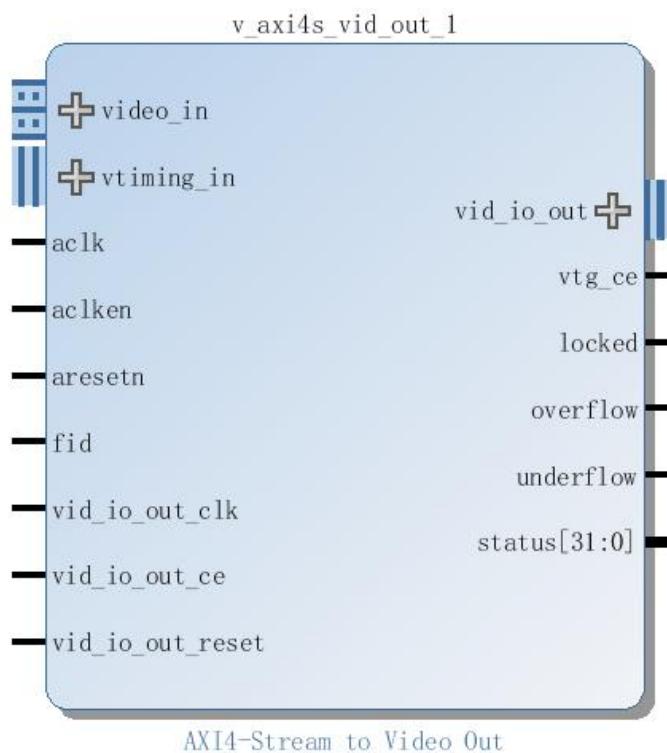
3.6 PLL 时钟设置

由于这里的分辨率是 640X480 因此提供给 VTC IP 和 VID OUTIP 的时钟只要 25M 就可以了



3.7 VID_OUT IP 的分析

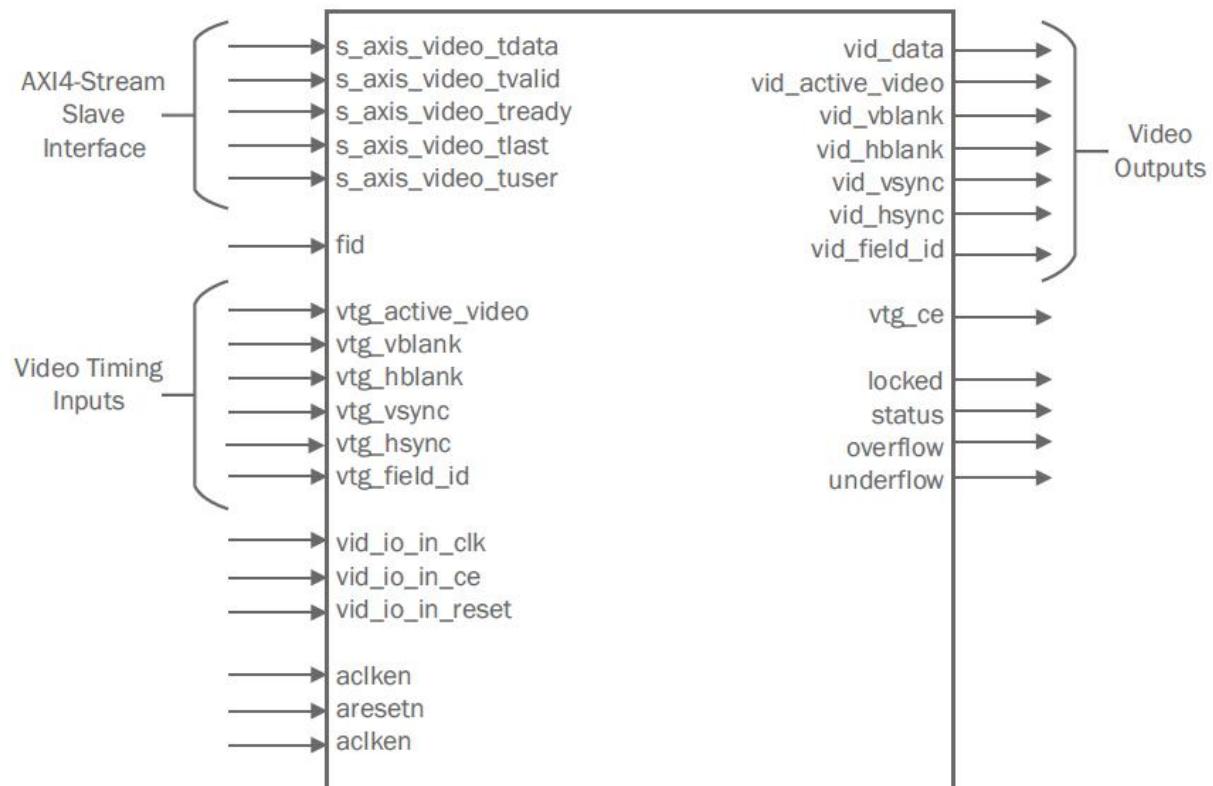
3.7.1 VID_OUT 的参数介绍



这些参数和前面的 V_TPG 参数类似

- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟
- Video Format: 视频格式
- FIFO Depth: FIFO 深度
- Hysteresis Level: 滞后输出

3.7.2 VID_OUT IP 接口信号的定义



Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_out_ce	Input	1	Native video clock enable
vid_io_out_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
vtg_ce	Output	1	VTC clock enable. Used to halt the timing generator for synchronization purposes.
locked	Output	1	Flag indicating whether the VTC is locked to the input timing. 1=locked. Synchronous to vid_io_out_clk.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk.

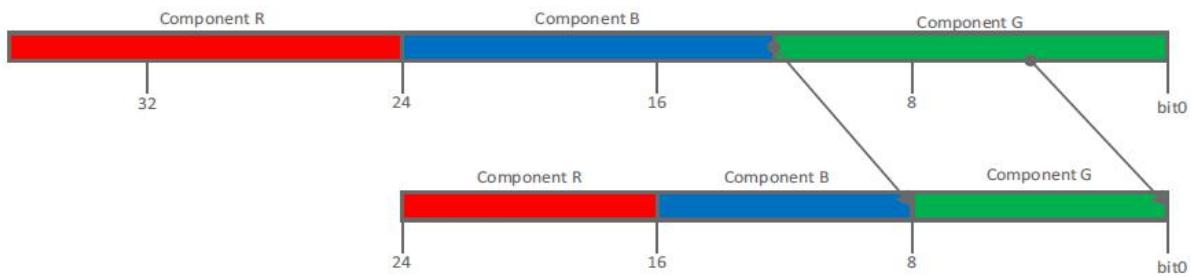
Video Timing Interface

Signal Name	Direction	Width	Description
vtg_vsync	In	1	VTC vertical sync. Active High
vtg_hsync	In	1	VTC horizontal sync. Active High
vtg_vblank	In	1	VTC vertical blank. Active High
vtg_hblank	In	1	VTC horizontal blank. Active High
vtg_act_vid	In	1	VTC active video signal. 1 = active video, 0 = blanked video
vtg_field_id	In	1	VTC field ID. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

AXI4 - Stream Interface

Signal Name	Direction	Width	Description
s_axis_video_tvalid	Input	1	AXI4-Stream TVALID. Active video data enable
s_axis_video_tuser	Input	1	AXI4-Stream TUSER. Start of Frame
s_axis_video_tlast	Input	1	AXI4-Stream TLAST. End of Line
s_axis_video_tready	Output	1	AXI4-Stream TREADY. Inverted FIFO full

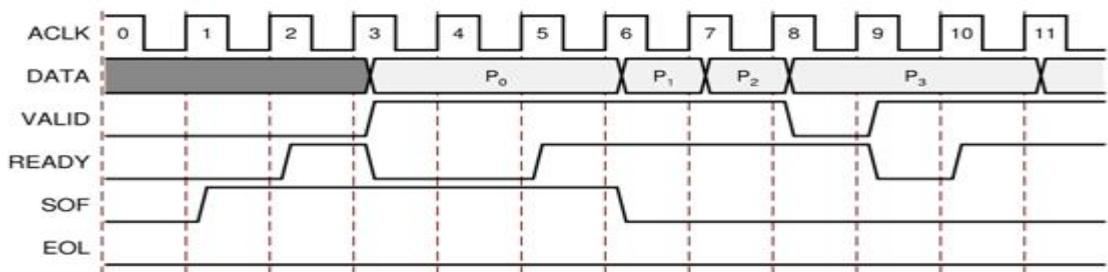
对于 s_axis_video_tdata(TDATA)需要注意一些事情,一般情况下我们的 RGB888 输出,但是,如果 s_axis_video_tdata 是 32bit 那么 VID_OUT IP 会自动截取到 24bit。由于技术手册只给出了 12bit 到 8bit 的截取方式,也就是 RGB 12:12:12 到 RGB 8:8:8 如下图:



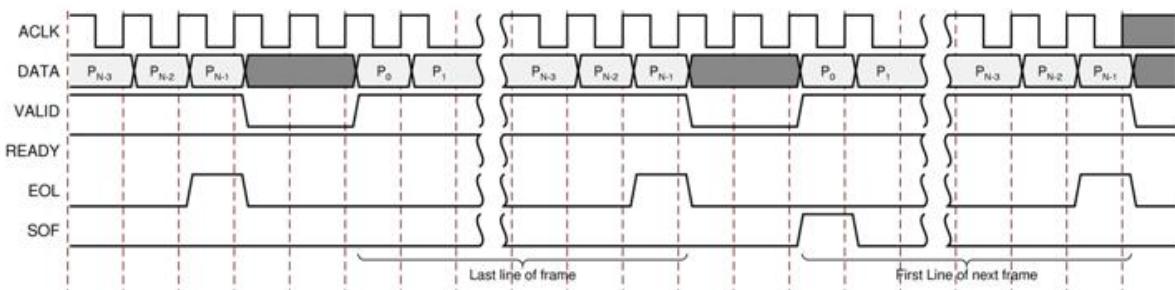
这种截取比较简单，把每个色度的低 4bit 截取就可以了。但是如果是 RGB10:10:10，官方并没有给出截取方式，但是可以通过纯色输出来进行测试。

因此最简单的办法是无需任何截取了，如果 s_axis_video_tdata 是 RGB8:8:8 那就无需任何截取，笔者设计的时候由于 AXI 总线是 32bit 因此数据的低 24bit 为 RGB 8:8:8 只要去掉高 24-31bit 就可以取得 RGB8:8:8，这样最省事。

以下时序图是在 SOF 是一帧图像的开始，当 VALID 和 READY 有效的时候开始传输数据。



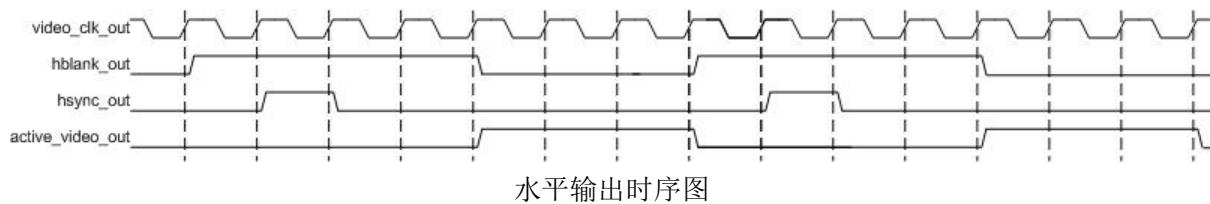
EOL 代表每一行的最后一个数据，SOF 代表前一帧的最后一行的结束，下一帧第一行的开始。SOF 为 1 个 PLUS 有效 (pg044_v_axis_out.pdf 没有描述清楚，而且有错误)。



Example Horizontal Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0003_0003
0x0070	Generator HSize	0x0000_0007
0x0078	Generator HSync	0x0005_0004
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

设置水平输出的相关寄存器

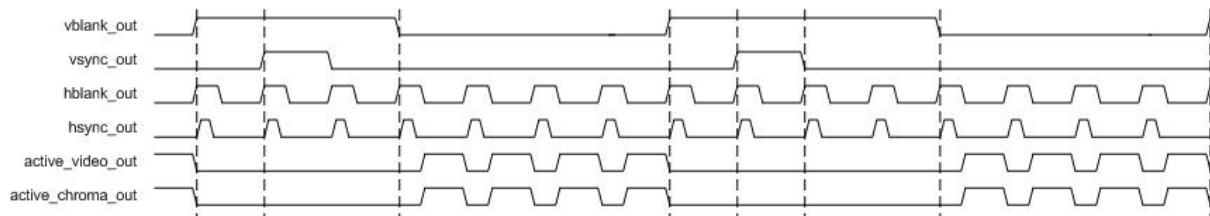


水平输出时序图

Example Vertical Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0004_0003
0x0070	Generator HSize	0x0000_0007
0x0074	Generator VSize	0x0000_0008
0x0078	Generator HSync	0x0005_0004
0x0080	Generator Frame 0 Vsync	0x0006_0005
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

设置垂直输出的相关寄存器



垂直输出时序图

3.8 FPGA 实现的用户逻辑代码

3.8.1 关键信号 1

```
assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready &
m_axis_video_tlast &(vid_in_v_cnt == VID_IN_VS); // dma in last signal
```

m_axis_video_tvalid:此信号是 vid in IP 输出的，代表输出数据有效

s_axis_s2mm_tready:此信号是 DMA IP 输出的，代表 DMA 可以接收数据

m_axis_video_tlast:这是每一行图像数据的最后一个像素的信号标志

vid_in_v_cnt == VID_IN_VS: 表示一副图像的最后一个像素输出。
 s_axis_s2mm_tlast: 所有这些信号有效的时候代表 DMA 的最后一个数据
 s_axis_s2mm_tlast 信号有效。

3.8.2 关键信号 2

```
assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready &
(vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); //vid out user
m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。
s_axis_video_tready: vid out IP 准备好了, 可以接收数据
(vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); 行计数器为 0 场计数器也为 0 说明要么这副图像已经结束, 也可以理解为下一副图像开始前。这样结合
s_axis_video_tready, m_axis_mm2s_tvalid 为 1, 基于 FPGA 时序, 下一个时钟输出
s_axis_video_tuser 为 1 正好是一副图像的第一个像素。
s_axis_video_tuser: 因此 s_axis_video_tuser 代表了每一副图像开始的第一个像素。
```

3.8.3 关键信号 3

```
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt ==
VID_OUT_HS); //vid out last signal
```

m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。
 s_axis_video_tready: vid out IP 准备好了, 可以接收数据
 vid_out_h_cnt == VID_OUT_HS: 图像一行数据的最后一个像素。

3.8.4 部分关键代码

表 3-6-4-1

<pre>reg [10:0] vid_out_v_cnt; reg [10:0] vid_out_h_cnt; reg [10:0] vid_in_v_cnt; parameter VID_OUT_HS = 11'd639;//图像输出行分辨率 parameter VID_OUT_VS = 11'd479;//图像输出场分辨率 parameter VID_IN_VS = 11'd479; always@(posedge FCLK_CLK0) begin if(!gpio_rtl_tri_o_0)</pre>

```

    vid_out_v_cnt <= 11'd0;
else
    if(m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == VID_OUT_HS))
        if(vid_out_v_cnt != VID_OUT_VS)
            vid_out_v_cnt <= vid_out_v_cnt + 1'b1;
        else
            vid_out_v_cnt <= 11'd0;
    else
        vid_out_v_cnt <= vid_out_v_cnt;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_out_h_cnt <= 11'd0;
    else
        if(m_axis_mm2s_tvalid & s_axis_video_tready)
            if(vid_out_h_cnt != VID_OUT_HS)
                vid_out_h_cnt <= vid_out_h_cnt + 1'b1;
            else
                vid_out_h_cnt <= 11'd0;
        else
            vid_out_h_cnt <= vid_out_h_cnt;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_in_v_cnt <= 11'd0;
    else
        if(m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast)
            if(vid_in_v_cnt != VID_IN_VS)
                vid_in_v_cnt <= vid_in_v_cnt + 1'b1;
            else
                vid_in_v_cnt <= 11'd0;
        else
            vid_in_v_cnt <= vid_in_v_cnt;
end

assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == 11'd0) &
(vid_out_v_cnt == 11'd0); //vid out user
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == VID_OUT_HS); //vid out last signal

```

```
assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast
&(vid_in_v_cnt == VID_IN_VS); // dma in last signal
```

3.9 PS 部分

3.9.1 DMA 中断函数部分分析

为了让图像输出高品质效果，PS 部分设计了 3 缓存处理机制。3 缓存处理机制在大量图像缓冲处理方法是最有效的办法之一。

在 DMA_intr.h 文件中，定义 3 段内存空间用于保存三副最新的图像。

```
#define BUFFER0_BASE (MEM_BASE_ADDR)
#define BUFFER1_BASE (MEM_BASE_ADDR + IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define BUFFER2_BASE (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
```

在 DMA_intr.h 文件中，还定义一下 2 个变量 1 个指针数组。tx_buffer_index; 指示了当前的发送缓存序号，rx_buffer_index; 指示了当前的接收缓存序号。*BufferPtr[3] 会被制定到对应的内存地址空间。

```
extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;
extern u32 *BufferPtr[3];
```

在 main 函数里面有这么一段实现了指针数组指向内存地址空间。

```
BufferPtr[0] = (u32 *)BUFFER0_BASE;
BufferPtr[1] = (u32 *)BUFFER1_BASE;
BufferPtr[2] = (u32 *)BUFFER2_BASE;
```

下面给出 dma_intr.h 的完整代码

表 3-7-1-1 dma_intr.h

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 */
#ifndef DMA_INTR_H
#define DMA_INTR_H
#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
```

```
#include "xscugic.h"

/********************* Constant Definitions *****/
/*
 * Device hardware build related constants.
 */

#define DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID

#define MEM_BASE_ADDR      0x10000000

#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR

#define IMAGE_WIDTH      640
#define IMAGE_HEIGHT     480
#define BYTES_PER_PIXEL   4
#define BUFFER_NUM       2

#define MEM_BASE_ADDR      0x10000000

#define BUFFER0_BASE      (MEM_BASE_ADDR )
#define BUFFER1_BASE      (MEM_BASE_ADDR +     IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)
#define BUFFER2_BASE      (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)

/* Timeout loop counter for reset
 */
#define RESET_TIMEOUT_COUNTER    10000
/* test start value
 */
#define TEST_START_VALUE    0xC
/*
 * Buffer and Buffer Descriptor related constant definition
 */
#define MAX_PKT_LEN      (IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
/*
 * transfer times
 */
```

```
/*
#define NUMBER_OF_TRANSFERS 100000

extern volatile int TxDone;
extern volatile int RxDone;
extern volatile int Error;

extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;

extern u32 *BufferPtr[3];

int DMA_CheckData(int Length, u8 StartValue);
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr);
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);
#endif
```

每次一幅图像通过 DMA 进入 DDR 后，会产生 DMA 中断请求，在 DMA 中断请求中，会指定下一次 DMA 接收的 buffer 位置。

表 3-7-1-2 DMA_RxIntrHandler 函数

```
/****************************************************************************
*
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.
*
* @param    Callback is a pointer to RX channel of the DMA engine.
*
* @return   None.
*
* @note    None.
*
****************************************************************************/
static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    u32 Status;
```

```
int TimeOut;
XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

/* Read pending interrupts */
IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);

/* Acknowledge pending interrupts */
XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

/*
 * If no interrupt is asserted, we do not do anything
 */
if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
    return;
}

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
*/
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
    xil_printf("rx error! \r\n");
    return;
}

/*
 * If completion interrupt is asserted, then set RxDone flag
*/
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
    RxDone++;
}

if(rx_buffer_index == 2)
    rx_buffer_index = 0;
else
    rx_buffer_index++;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[rx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma failed! 0 %d\r\n", Status);
```

```
    return;  
}  
  
}
```

发送函数通过 tx_buffer_index 标记需要发送的缓存部分,并且确保发送的是最新保存的一副图像。

表 3-7-3 DMA_TxIntrHandler

```
/**************************************************************************/  
/*  
* This is the DMA TX Interrupt handler function.  
*  
* It gets the interrupt status from the hardware, acknowledges it, and if any  
* error happens, it resets the hardware. Otherwise, if a completion interrupt  
* is present, then sets the TxDone.flag  
*  
* @param    Callback is a pointer to TX channel of the DMA engine.  
*  
* @return   None.  
*  
* @note    None.  
*  
**************************************************************************/  
static void DMA_TxIntrHandler(void *Callback)  
{  
  
    u32 IrqStatus;  
    u32 Status;  
    int TimeOut;  
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;  
  
    /* Read pending interrupts */  
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE);  
  
    /* Acknowledge pending interrupts */  
  
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);  
  
    /*  
     * If no interrupt is asserted, we do not do anything  
     */  
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
```

```
        return;
    }

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
 */
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

    //Error = 1;
    xil_printf("tx error! \r\n");
    return;
}

/*
 * If Completion interrupt is asserted, then set the TxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    TxDone++;
}

if(rx_buffer_index == 0)
    tx_buffer_index = 2;
else
    tx_buffer_index = rx_buffer_index - 1;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[tx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma failed! 0 %d\r\n", Status);
    return;
}
}
```

3.9.2 main.c 文件

这个主程序比较简单，内容比上一个课程的精简很多，这里需要注意的地方是 XGpio_DiscreteWrite(&Gpio, 1, 1); 函数这个函数是这只摄像头和 DMA 之间数据同步的，没有这个同步图像容易错位。另外在主函数里面首先启动 DMA 接收和发送中断各一次，以后就可以在中断里

面继续触发了。

表 3-7-2-1 main.c

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 * axi dma test
 *
 */

#include "dma_intr.h"
#include "sys_intr.h"
#include "xgpio.h"

volatile int TxDone;
volatile int RxDone;
volatile int Error;

volatile u8 tx_buffer_index;
volatile u8 rx_buffer_index;

u32 *BufferPtr[3];

static XScuGic Intc; //GIC
static XAxiDma AxiDma;
static XGpio Gpio;

#define AXI_GPIO_DEV_ID           XPAR_AXI_GPIO_0_DEVICE_ID

int init_intr_sys(void)
{
    DMA_Intr_Init(&AxiDma,0); //initial interrupt system
    Init_Intr_System(&Intc); // initial DMA interrupt system
    Setup_Intr_Exception(&Intc);
    DMA_Setup_Intr_System(&Intc,&AxiDma,TX_INTR_ID,RX_INTR_ID); //setup dma interrupt
    system
    DMA_Intr_Enable(&Intc,&AxiDma);
}

int main(void)
{
```

u32 Status;

```
BufferPtr[0] = (u32 *)BUFFER0_BASE;  
BufferPtr[1] = (u32 *)BUFFER1_BASE;  
BufferPtr[2] = (u32 *)BUFFER2_BASE;
```

```
tx_buffer_index = 0;  
rx_buffer_index = 0;  
TxDone = 0;  
RxDone = 0;  
Error = 0;
```

```
XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);  
XGpio_SetDataDirection(&Gpio, 1, 0);  
init_intr_sys();
```

Miz702_EMIO_init();ov7725_init_rgb();

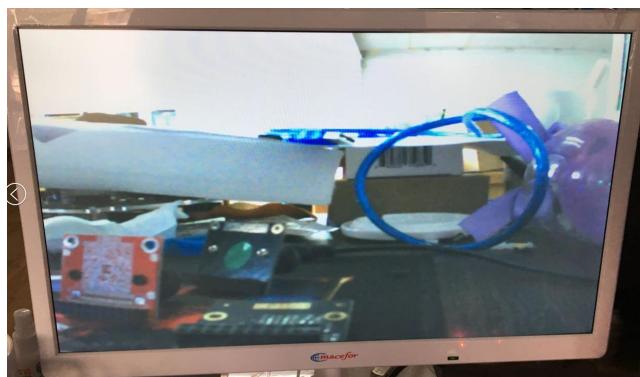
```
XGpio_DiscreteWrite(&Gpio, 1, 1);  
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[rx_buffer_index],  
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
```

```
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[tx_buffer_index],  
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
```

```
while (1);  
    return XST_SUCCESS;
```

{}

3.10 实验效果



S03_CH04_AXI_DMA_OV5640 摄像头采集系统

4.1 概述

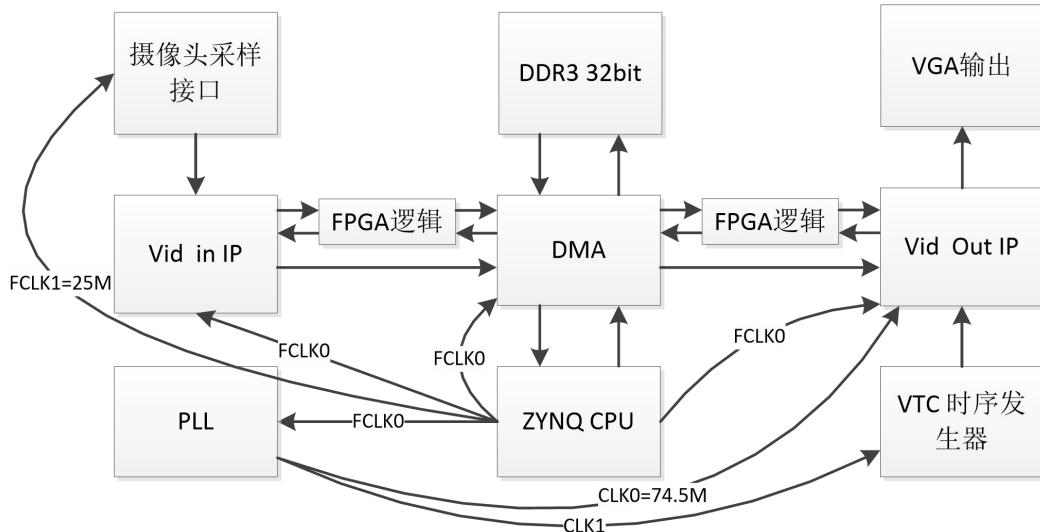
本课程讲解如何搭建基于 DMA 的图形系统，方案原理和搭建 7725 的一样，只是 OV5640 显示的分辨率是 1280X720 如下，只是购买了 OV5640 摄像头的用户可以直接从本章开始学习。

摄像头采样图像数据后通过 DMA 送入到 DDR，在 PS 部分产生 DMA 接收中断，在接收中断里面再把 DDR 里面保持的图形数据 DMA 发送出去。在 FPGA 的接收端口部分产生 VID OUT 时序驱动 VGA 显示器显示图形。MIZ701N 没有 VGA 接口，可以跳过直接看《S03_CH05_AXI_DMA_HDMI 图像输出》或者大家不想看本章的也可以直接跳到《S03_CH05_AXI_DMA_HDMI 图像输出》这两节课的核心教学内容一样。

《S03_CH03_AXI_DMA_OV7725 摄像头采集系统》、《S03_CH04_AXI_DMA_OV5640 摄像头采集系统》、《S03_CH05_AXI_DMA_HDMI 图像输出》。读者可以根据自己需求情况而阅读，请知悉。

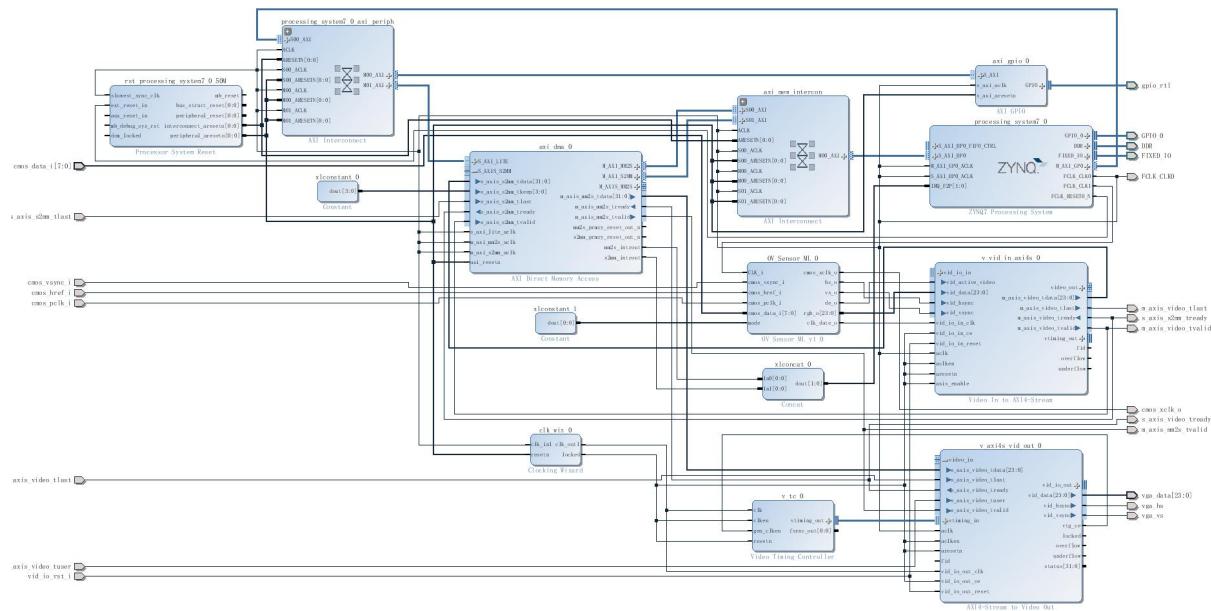
4.2 系统构架

4.2.1 构架方案图



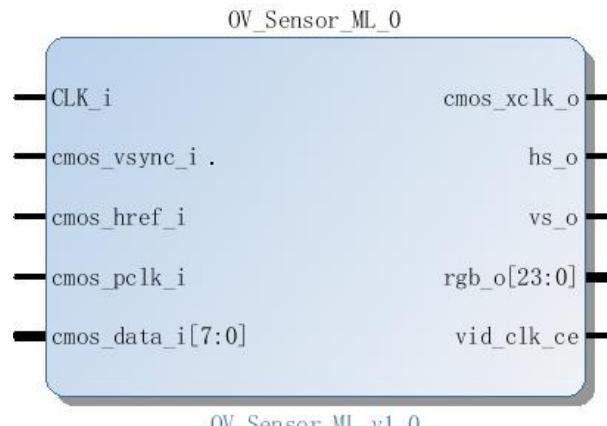
摄像头接口采集的摄像头数据，进过 vid in 视频输入 IP 后，还需要通过用户 FPGA 逻辑编程，和 DMA IP 之间实现握手协议，实现把数据通过 DMA 写入到 DDR。每次写入一副图像的数据后，产生一次接收中断，接收中断函数，会把数据三缓存后，在通过 DMA 发出去，DMA 发送完成后产生中断，在中断中，把缓存好的图像发送出去。DMA 发送的数据需要发送到 vid out 视频输出 IP。同理，DMA 和 vid out IP 之间也许需要增加 FPGA 用户代码实现接口的握手协议。数据进入 vid out 后，会随同 vtc IP 输出符合 VGA 时序的图像信号。vid out 的输出就可以直接定义成 VGA 信号输出。

4.2.2 构 BLOCK 模块化设计方案图



4.3 vid in IP 介绍

4.3.1 OV_Sensor_ML 自定义 IP 模块



外部信号接口说明：

CLK_i : 为输入时钟，通常接 24MHz 或者 25MHz

Cmos_xclk_o:摄像头工作，通常直接把 CLK_i 连接到 cmos_xclk_o

Cmos_vsync_i: 摄像头场同步输入 上升沿代表场同步开始

Cmos_href_i:摄像头行同步输入 高电平代表行数据有效

Cmos_data[7:0]:摄像头数据输入

Hs_o:采集 OV_Sensor_ML IP 输出的行数据有效

Vs_o:采集 OV_Sensor_ML IP 输出的场同步信号

Vid_clk_ce:此信号用于和 vid_in IP 的时钟同步(由于 OV_Sensor_ML IP 是每两个时钟输出一次 rgb[23:0]的图像数据，因此需要通过 Vid_clk_ce 对时钟频率进行同步，有了这个信号，可以解决输入采集 IP 和 vid_in IP 数据接口之间的同步问题).

OV_Sensor_ML IP 包含 3 个源程序文件，分别为 OV_Sensor_ML.v、 cmos_decode_v1.v、 count_reset_v1.v 文件。

OV_Sensor_ML.v 程序中，对 cmos_data_i、 cmos_href_i、 cmos_vsync_i 做了一次寄存器，笔者发现图像效果有所改观。笔者分析，是因为寄存后有利于去除一些毛刺信号，提高了数据的稳定性。

表 3-3-1-1 OV_Sensor_ML 源码 OV_Sensor_ML.v

```
'timescale 1ns / 1ps
///////////////////////////////
// Company: milinker
// Engineer:tangjinyuan
//
// Create Date:    15:54:59 11/21/2015
// Design Name:
// Module Name:    OV7725_IP_ML
// Project Name: OV7725_IP_ML
// Target Devices: ZYNQ
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////
module OV_Sensor_ML(
    input CLK_i,
    //----- CMOS sensor hardware interface -----
    input      cmos_vsync_i,//cmos vsync
    input      cmos_href_i, //cmos hsync refrence
    input      cmos_pclk_i, //cmos pxiel clock
    output     cmos_xclk_o, //cmos externl clock
    input[7:0]cmos_data_i, //cmos data
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    // output de_o,//data enable.
    output [23:0]rgb_o,//data output,
    output vid_clk_ce
);

```

```
//-----视频输出解码模块-----//
wire [15:0]rgb_o_r;
assign rgb_o = {rgb_o_r[4:0] ,3'd0 ,rgb_o_r[10:5] ,2'd0,rgb_o_r[15:11],3'd0};

reg [7:0]cmos_data_r;
reg cmos_href_r;
reg cmos_vsync_r;

always@(posedge cmos_pclk_i)
begin
    cmos_data_r <= cmos_data_i;
    cmos_href_r <= cmos_href_i;
    cmos_vsync_r<= cmos_vsync_i;
end
//assign rgb_o = 24'b11111111_00000000_11111111;
cmos_decode cmos_decode_u0(
    //system signal.
    .cmos_clk_i(CLK_i),//cmos senseor clock.
    .rst_n_i(RESETn_i2c),//system reset.active low.
    //cmos sensor hardware interface.
    .cmos_pclk_i(cmos_pclk_i),//(cmos_pclk),//input pixel clock.
    .cmos_href_i(cmos_href_r),//(cmos_href),//input pixel hs signal.
    .cmos_vsync_i(cmos_vsync_r),//(cmos_vsync),//input pixel vs signal.
    .cmos_data_i(cmos_data_r),//(cmos_data),//data.
    .cmos_xclk_o(cmos_xclk_o),//(cmos_xclk),//output clock to cmos sensor.
    //user interface.
    .hs_o(hs_o),//hs signal.
    .vs_o(vs_o),//vs signal.
    // .de_o(de_o),//data enable.
    .rgb565_o(rgb_o_r),//data output
    .vid_clk_ce(vid_clk_ce)
);

count_reset_v1#(
    .num(20'hffff0)
)(
    .clk_i(CLK_i),
    .rst_o(RESETn_i2c)
);

endmodule
```

1 cmos_decode_v1.v 是本模块的关键部分,实现了RGB565 的解码输出以及 vid_clk_ce 实现了此模块和 vid_in IP 直接时序匹配的关系。

表 3-3-1-2 OV_Sensor_ML 源码 cmos_decode_v1.v

```
'timescale 1ns / 1ps
///////////////////////////////
// Company: milinker corporation
// WEB:www.milinker.com
// BBS:www.osrc.cn
// Engineer:..
// Create Date:    07:28:50 09/04/2015
// Design Name:   cmos_decode_v1
// Module Name:   cmos_decode_v1
// Project Name:  cmos_decode_v1
// Target Devices:
// Tool versions:
// Description:   cmos_decode_v1.
// Revision:      V1.0
// Additional Comments:
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low
//4) _dg debug signal
//5) _r  reg delay
//6) _s state machine
/////////////////////////////
module cmos_decode(
    //system signal.
    input cmos_clk_i,//cmos senseor clock.
    input rst_n_i,//system reset.active low.
    //cmos sensor hardware interface.
    input cmos_pclk_i,//input pixel clock.
    input cmos_href_i,//input pixel hs signal.
    input cmos_vsync_i,//input pixel vs signal.
    input[7:0]cmos_data_i,//data.
    output cmos_xclk_o,//output clock to cmos sensor.
    //user interface.
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    output reg [15:0]rgb565_o,//data output
    output vid_clk_ce
);
parameter[5:0]CMOS_FRAME_WAITCNT = 4'd15;
```

```
reg[4:0] rst_n_reg = 5'd0;
//reset signal deal with.
always@(posedge cmos_clk_i)
begin
    rst_n_reg <= {rst_n_reg[3:0],rst_n_i};
end

reg[1:0]vsync_d;
reg[1:0]href_d;
wire vsync_start;
wire vsync_end;
//vs signal deal with.
always@(posedge cmos_pclk_i)
begin
    vsync_d <= {vsync_d[0],cmos_vsync_i};
    href_d  <= {href_d[0],cmos_href_i};
end

assign vsync_start =  vsync_d[1]&(!vsync_d[0]);
assign vsync_end   = (!vsync_d[1])&vsync_d[0];

reg[6:0]cmos_fps;
//frame count.
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        begin
            cmos_fps <= 7'd0;
        end
    else if(vsync_start)
        begin
            cmos_fps <= cmos_fps + 7'd1;
        end
    else if(cmos_fps >= CMOS_FRAME_WAITCNT)
        begin
            cmos_fps <= CMOS_FRAME_WAITCNT;
        end
end
//wait frames and output enable.
reg out_en;
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
```

```
begin
    out_en <= 1'b0;
end
else if(cmos_fps >= CMOS_FRAME_WAITCNT)
begin
    out_en <= 1'b1;
end
else
begin
    out_en <= out_en;
end
end

//output data 8bit changed into 16bit in rgb565.
reg [7:0] cmos_data_d0;
reg [15:0] cmos_rgb565_d0;
reg byte_flag;

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        byte_flag <= 0;
    else if(cmos_href_i)
        byte_flag <= ~byte_flag;
    else
        byte_flag <= 0;
end

reg byte_flag_r0;
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        byte_flag_r0 <= 0;
    else
        byte_flag_r0 <= byte_flag;
end

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        cmos_data_d0 <= 8'd0;
    else if(cmos_href_i)
        cmos_data_d0 <= cmos_data_i; //MSB -> LSB
```

```

else if(~cmos_href_i)
    cmos_data_d0 <= 8'd0;
end

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        rgb565_o <= 16'd0;
    else if(cmos_href_i&byte_flag)
        rgb565_o <= {cmos_data_d0,cmos_data_i}; //MSB -> LSB
    else if(~cmos_href_i)
        rgb565_o <= 8'd0;
end

assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
assign vs_o = out_en ? vsync_d[1] : 1'b0;
assign hs_o = out_en ? href_d[1] : 1'b0;

assign cmos_xclk_o = cmos_clk_i;

endmodule

```

count_reset_v1.v 源文件实现了信号的延迟复位。

表 3-3-1-1 count_reset_v1.v

```

`timescale 1ns / 1ps
///////////////////////////////
// Company: milinker corporation
// WEB:www.milinker.com
// BBS:www.osrc.cn
// Engineer:sanliuyaoling.
// Create Date: 07:28:50 12/04/2015
// Design Name: count_reset_v1
// Module Name: count_reset_v1
// Project Name: count_reset_v1
// Target Devices: XC7Z020-CLG484-1I
// Tool versions: vivado2015.4
// Description: count_reset_v1
// Revision: V1.0
// Additional Comments:
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low

```

```
//4) _dg debug signal
//5) _r  reg delay
//6) _s state machine
///////////////////////////////
module count_reset_v1#
(
    parameter[19:0]num = 20'hffff0
)(
    input clk_i,
    output rst_o
);

reg[19:0] cnt = 20'd0;
reg rst_d0;

/*count for clock*/
always@(posedge clk_i)
begin
    cnt <= ( cnt <= num)?( cnt + 20'd1 ):num;
end

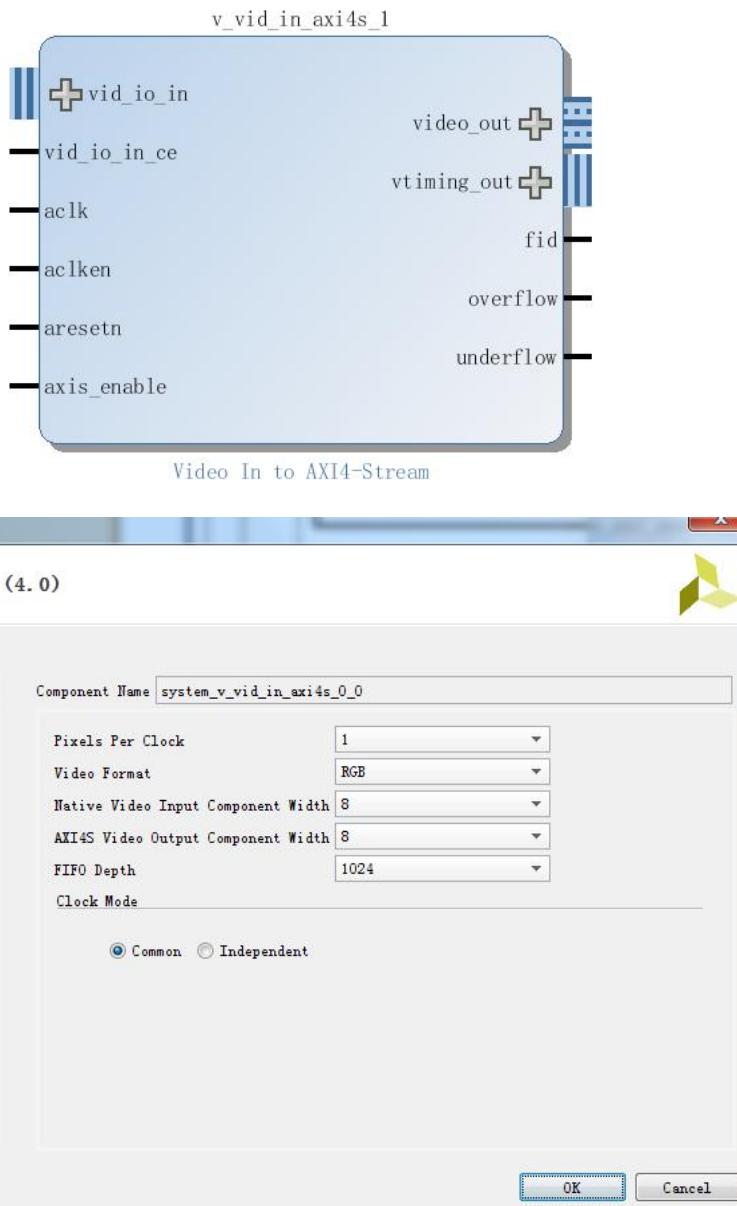
/*generate output signal*/
always@(posedge clk_i)
begin
    rst_d0 <= ( cnt >= num)?1'b1:1'b0;
end

assign rst_o = rst_d0;

endmodule
```

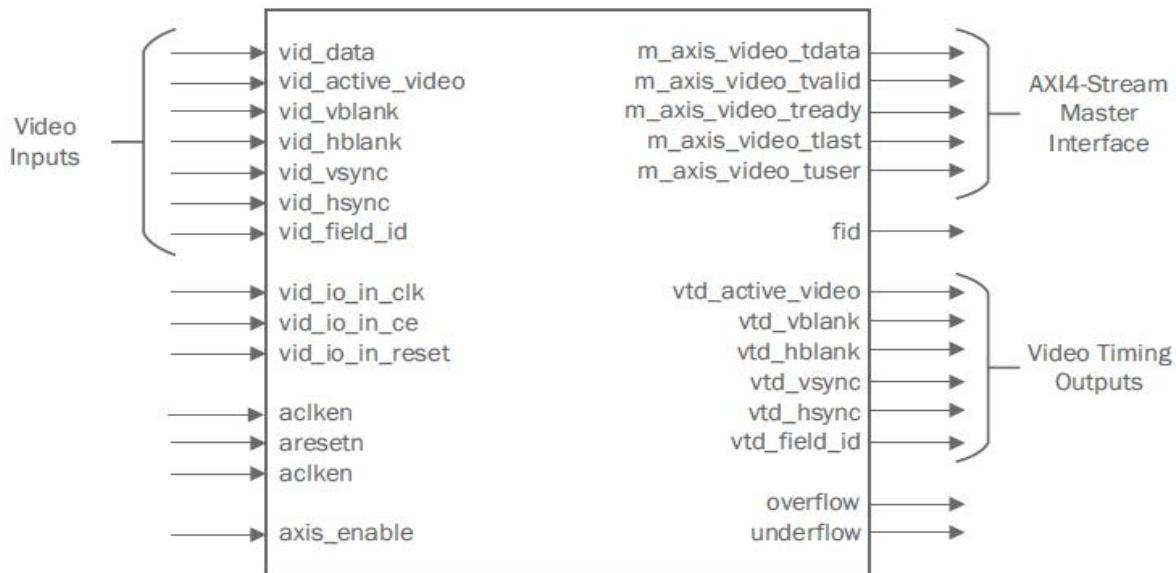
表 3-3-1-2

4.3.2 vid in IP 模块



- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Video Format: 视频格式
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- FIFO Depth: FIFO 深度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟

4.3.2 VID_IN IP 接口信号的定义



Common Interface

Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
axis_enable	Input	1	This input should be connected to the VTC detector locked status and is synchronous to vid_io_in_clk. 1 = Enable writes into FIFO 0 = Disable writes into FIFO
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_in_clk	Input	1	Native video clock. Only available in independent clock mode.
vid_io_in_ce	Input	1	Native video clock enable
vid_io_in_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk. If an overflow occurs, this could indicate that the connected AXI4-Stream Slave is creating excessive back-pressure.
underflow	Output	1	Flag indicating that the FIFO has under-flowed. This should never occur under normal operation. Synchronous to aclk.

Video Timing Interface

Signal Name	Direction	Width	Description
vtd_vsync	Out	1	Vertical sync video timing signal.
vtd_hsync	Out	1	Horizontal sync video timing signal.
vtd_vblank	Out	1	Vertical blank video timing signal.
vtd_hblank	Out	1	Horizontal blank video timing signal.
vtd_active_video	Out	1	Active video flag. 1 = active video, 0 = blanked video
vtd_field_id	Out	1	VTC field ID. 0= even field, 1= odd field.

Video Input Interface

Signal Name	Direction	Width	Description
vid_active_video	In	1	Video data valid. 1 = active video, 0 = blanked video
vid_vsync	In	1	Vertical sync video timing signal. Active High
vid_hsync	In	1	Horizontal sync video timing signal. Active High
vid_vblank	In	1	Vertical blank video timing signal. Active High
vid_hblank	In	1	Horizontal blank video timing signal. Active High
vid_data	In	8-256	Parallel video input data.
vid_field_id	In	1	Video field. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

使用到的信号有：

Vid_in IP 输入端信号：

Vid_data: 视频数据输入

Vid_active_video: 视频数据有效

Vid_hsync: 视频行同步信号 (非常关键信号, 下面重点分析对象)

Vid_vsync: 视频场同步信号

Vid_io_in_ce: 数据输入有效 (非常关键信号, 下面重点分析对象)

vid_io_in_clk: 这是时钟信号和摄像头时钟同步

Vid_io_in_reset: 这个信号, 高电平的时候复位

有很多读者会问笔者, 这些官方的 IP 如何使用, 这么没有详细的技术手册。还别说, 官方就是没有非常详细的技术手册, 有时候笔者也是使出浑身解数分析 vid_in IP 内部信号时序, 掌握 OV_Sensor_ML 自定义 IP 时序接口设计。

打开 v_vid_in_axi4s_v4_0_1_formatter.v 这个文件

下面对其关键的部分进行说明。

表 3-3-2-1 v_vid_in_axi4s_v4_0_1_formatter.v

```
'timescale 1ps/1ps
`default_nettype none
(* DowngradeIPIdentifiedWarnings="yes" *)
```

```
module v_vid_in_axi4s_v4_0_1_formatter #(
    parameter C_NATIVE_DATA_WIDTH = 24
) (
    // System signals
    input wire VID_IN_CLK,           // Native video clock
    input wire VID_RESET,            // Native video reset
    input wire VID_CE,               // Native video clock enable

    // Video input signals
    input wire VID_ACTIVE_VIDEO,     // Native video input data enable
    input wire VID_VBLANK,           // Native video input vertical blank
    input wire VID_HBLANK,           // Native video input horizontal blank
    input wire VID_VSYNC,             // Native video input vertical sync
    input wire VID_HSYNC,             // Native video input horizontal sync
    input wire VID_FIELD_ID,         // Native video input field-id
    input wire [C_NATIVE_DATA_WIDTH-1:0] VID_DATA, // Native video input data

    // Video timing detector signals
    output wire VTD_ACTIVE_VIDEO,    // Native video output data enable
    output wire VTD_VBLANK,           // Native video output vertical blank
    output wire VTD_HBLANK,           // Native video output horizontal blank
    output wire VTD_VSYNC,             // Native video output vertical sync
    output wire VTD_HSYNC,             // Native video output horizontal sync
    output wire VTD_FIELD_ID,         // Native video output field-id
    input wire VTD_LOCKED,            // Native video locked signal from VTD

    // FIFO write signals
    output wire [C_NATIVE_DATA_WIDTH+2:0] FIFO_WR_DATA, // FIFO write data
    output wire FIFO_WR_EN           // FIFO write enable
);

    // Wire and register declarations
    reg de_1 = 0;
    reg vblank_1 = 0;
    reg hblank_1 = 0;
    reg vsync_1 = 0;
    reg hsync_1 = 0;
    reg [C_NATIVE_DATA_WIDTH -1:0] data_1 = 0;
    reg de_2 = 0;
    reg v_blank_sync_2 = 0;
    reg [C_NATIVE_DATA_WIDTH -1:0] data_2 = 0;
    reg de_3 = 0; // DE output register
```

```
reg [C_NATIVE_DATA_WIDTH-1:0] data_3 = 0; // data output register
reg vert_blankning_intvl = 0; // SR, reset by DE rising
reg field_id_1 = 0;
reg field_id_2 = 0;
reg field_id_3 = 0;

wire v_blank_sync_1; // vblank or vsync
wire de_rising;
wire de_falling;
wire vsync_rising;
reg sof;
reg sof_1;
reg eol;
reg vtd_locked;
wire sof_rising;

// Assignments
assign FIFO_WR_DATA      = {field_id_3,sof_1,eol,data_3};
assign FIFO_WR_EN         = de_3 & ~VID_RESET & vtd_locked;
assign VTD_ACTIVE_VIDEO = de_1;
assign VTD_VBLANK        = vblank_1;
assign VTD_HBLANK        = hblank_1;
assign VTD_VSYNC          = vsync_1;
assign VTD_HSYNC          = hsync_1;
assign VTD_FIELD_ID      = field_id_1;

assign v_blank_sync_1 = vblank_1 || vsync_1;
assign de_rising   = de_1 && !de_2;
assign de_falling  = !de_1 && de_2;
assign vsync_rising = v_blank_sync_1 && !v_blank_sync_2;
assign sof_rising  = sof & ~sof_1;

// VTD locked process
always @(posedge VID_IN_CLK) begin
    if(VID_RESET | ~VTD_LOCKED) begin
        vtd_locked <= 1'b0;
    end else if(VID_CE) begin
        vtd_locked <= (sof_rising & VTD_LOCKED) ? 1'b1 : vtd_locked;
    end
end

// input, output, and delay registers
always @ (posedge VID_IN_CLK) begin
```

```

if(VID_RESET) begin
    de_1          <= 1'b0;
    de_2          <= 1'b0;
    de_3          <= 1'b0;
    vblank_1      <= 1'b0;
    hblank_1      <= 1'b0;
    vsync_1       <= 1'b0;
    hsync_1       <= 1'b0;
    field_id_1    <= 1'b0;
    field_id_2    <= 1'b0;
    field_id_3    <= 1'b0;
    data_1         <= {C_NATIVE_DATA_WIDTH{1'b0}};
    data_2         <= {C_NATIVE_DATA_WIDTH{1'b0}};
    data_3         <= {C_NATIVE_DATA_WIDTH{1'b0}};
    v_blank_sync_2 <= 1'b0;
    eol           <= 1'b0;
    sof            <= 1'b0;
    sof_1          <= 1'b0;
end else if(VID_CE) begin
    de_1          <= VID_ACTIVE_VIDEO;
    de_2          <= de_1;
    de_3          <= de_2;
    vblank_1      <= VID_VBLANK;
    hblank_1      <= VID_HBLANK;
    vsync_1       <= VID_VSYNC;
    hsync_1       <= VID_HSYNC;
    field_id_1    <= VID_FIELD_ID;
    field_id_2    <= field_id_1;
    field_id_3    <= field_id_2;
    data_1         <= VID_DATA;
    data_2         <= data_1;
    data_3         <= data_2;
    v_blank_sync_2 <= v_blank_sync_1;
    eol           <= de_falling;
    sof            <= de_rising && vert_blinking_intvl;
    sof_1          <= sof;
end
end

// Vertical back porch SR register
always @ (posedge VID_IN_CLK) begin
    if(VID_CE) begin
        if(vsync_rising) // falling edge of vsync

```

```

    vert_blinking_intvl <= 1;
else if (de_rising)          // rising edge of data enable
    vert_blinking_intvl <= 0;
end
end

endmodule

```

在上面代码中，

```

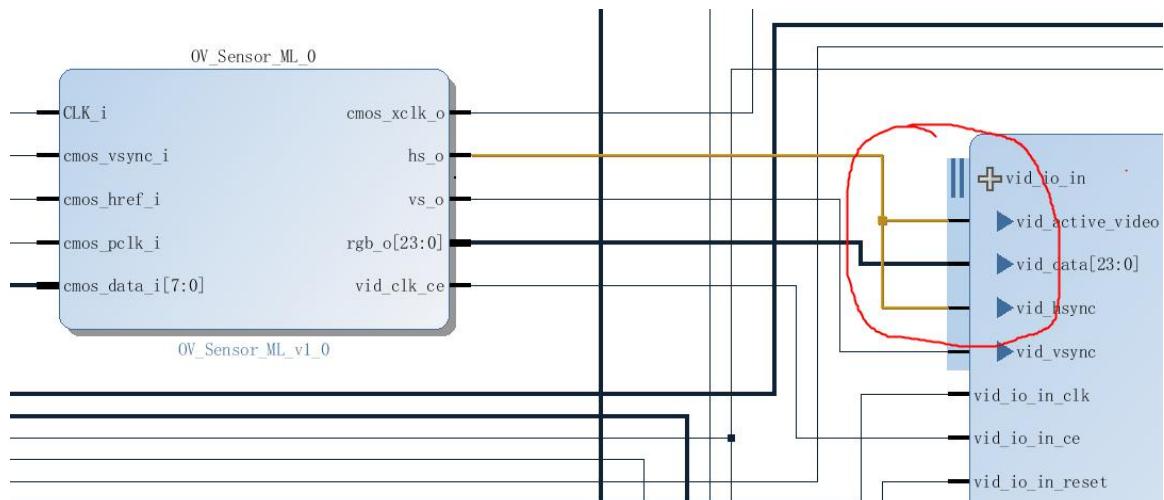
eol      <= de_falling;
sof      <= de_rising && vert_blinking_intvl;

```

eol 实际就是 tlast 信号，而 sof 就是 tuser 信号。tlast 信号代表每行图像数据的最后一个数据，tuser 代表每场数据的第一个数据。

所有非常关键的信号都和 de_falling 和 vert_blinking_intvl 有关系。

hs_o 和 vid_in IP 的连接关系。



上图中，被红色圈起来的 hs_o 信号，同时接到了 vid_in_ip 的 vid_active_video 和 vid_hsync 信号接口。因此，de 信号就是 hs_o 信号，而 vid_hsync 我们发现没有任何作用，也就是说不 hs_o 不连接到 vid_hsync 也不影响这里的程序工作。

VID_CE 这个参数就是前面的 vid_io_in_ce 信号，可以看出这个芯片有效的时候相对应的时序电路才会执行。在本工程中，摄像头每 2 个 pclk 输出 1 个有效的数据，而 vid_in IP 如果 VID_CE 为 1 则数据输入会每个时钟输入 1 个就错了。因此官方的 IP 设计的还是很不错考虑周到，通过 VID_CE 这个条件，控制时钟同步。

```

...
else if(VID_CE) begin
...
end
...

```

现在回到 OV_Sensor_ML 的 cmos_decode_v1.v 文件中有一段红色的代码如下：

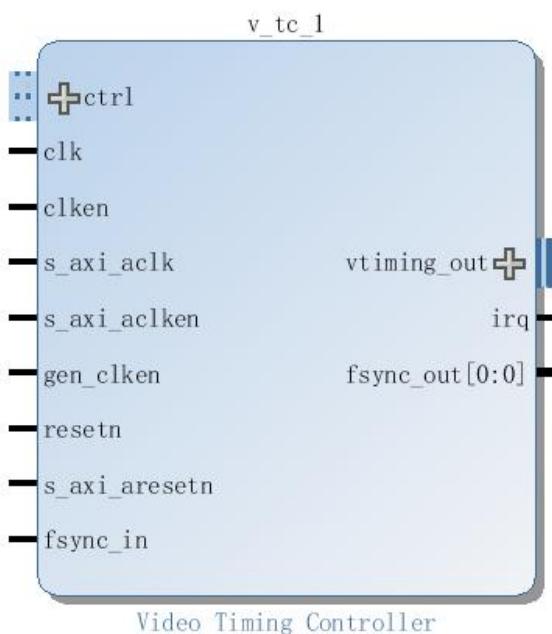
```
assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
```

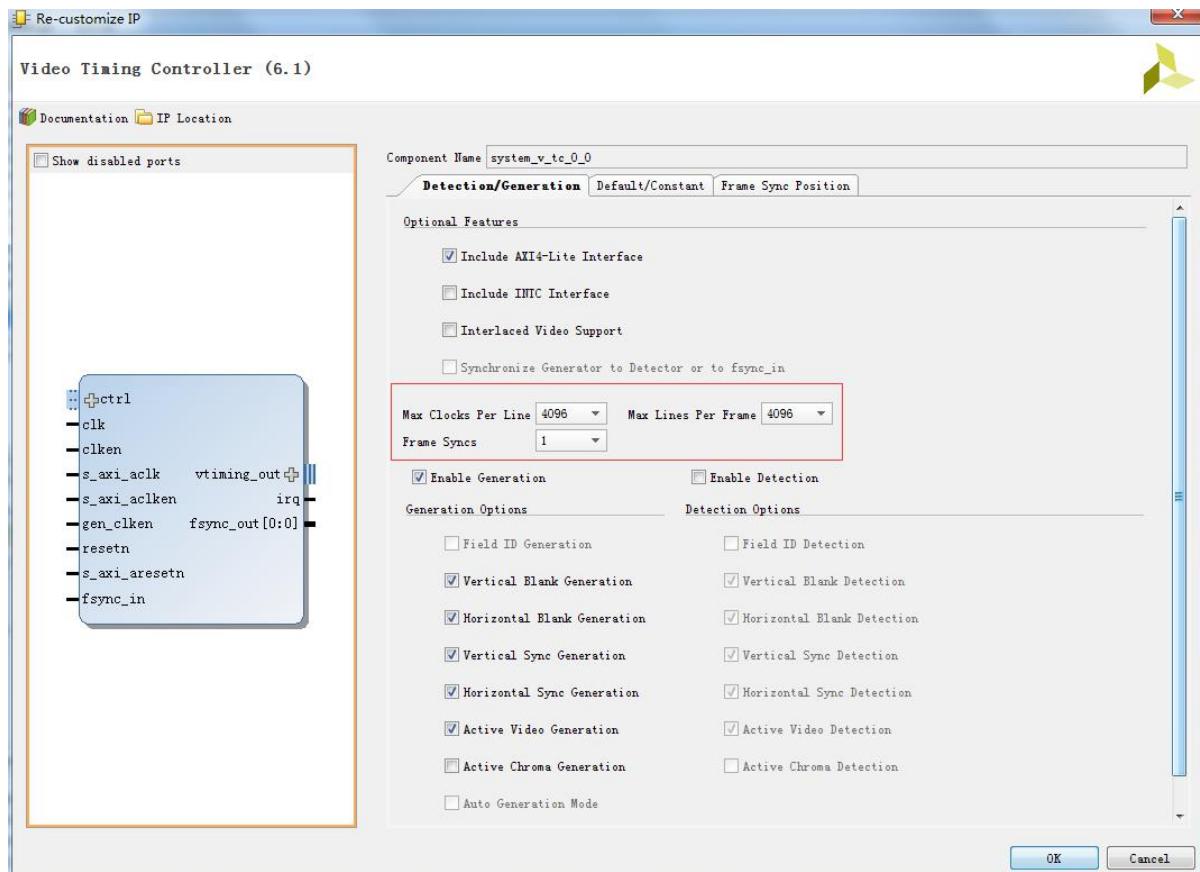
这段代码控制了 vid_clk_ce 的正确输出，关键部分是 `(byte_flag_r0&hs_o)||(!hs_o)`。当 hs_o 有效的时候，vid_in 的 VID_CE 信号就有效，当 hs_o=0 的时候 VID_CE 必须仍然有效，这样才能检测到 vsync_rising 信号了，检测到了 vsync_rising 才能有 `ert_blankning_intvl` 为 1，才有 tuser 信号。

好了罗嗦了半天，终于解释完了，如果有不清楚的，找我们技术支持吧。

4.4 VTC IP 的分析

4.4.1 VTC IP 的参数介绍





这个 IP 就是一个时序发生器，产生显示器输出所需要的时序信号。

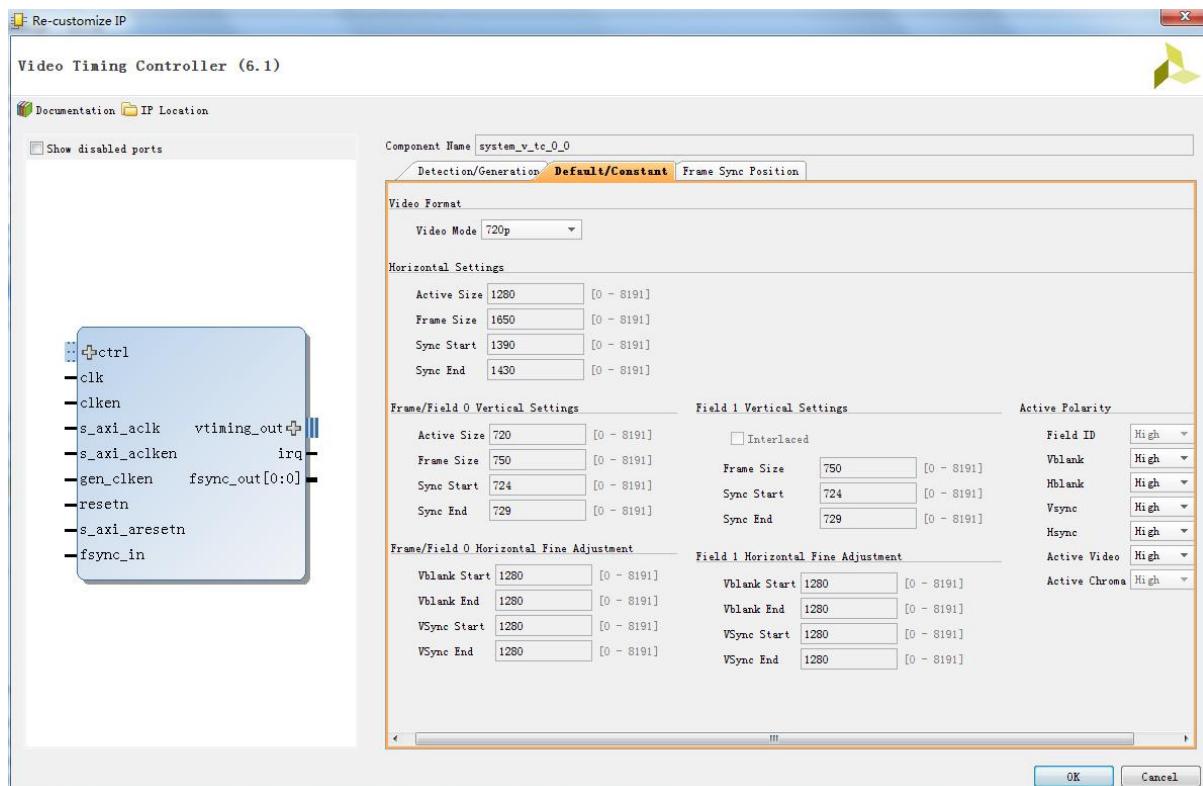
这个页面中，include AXI-lite interface 可以不勾选，不勾选就只能采用默认设置，无法在 C 语言中灵活配置了，所以笔者这里建议大家勾选吧。max clocks per line 和 max_lines per frame 需要设置下，当设置到 4096 的时候可以支持分辨率到最大，当然消耗的资源也更多。笔者这里太奢侈了设置了 4096。实际上设置到 2048 就够用了。本页面的其他信号可以采取默认设置。

Enable Generation:

支持产生时序，这个肯定是必须勾选的。

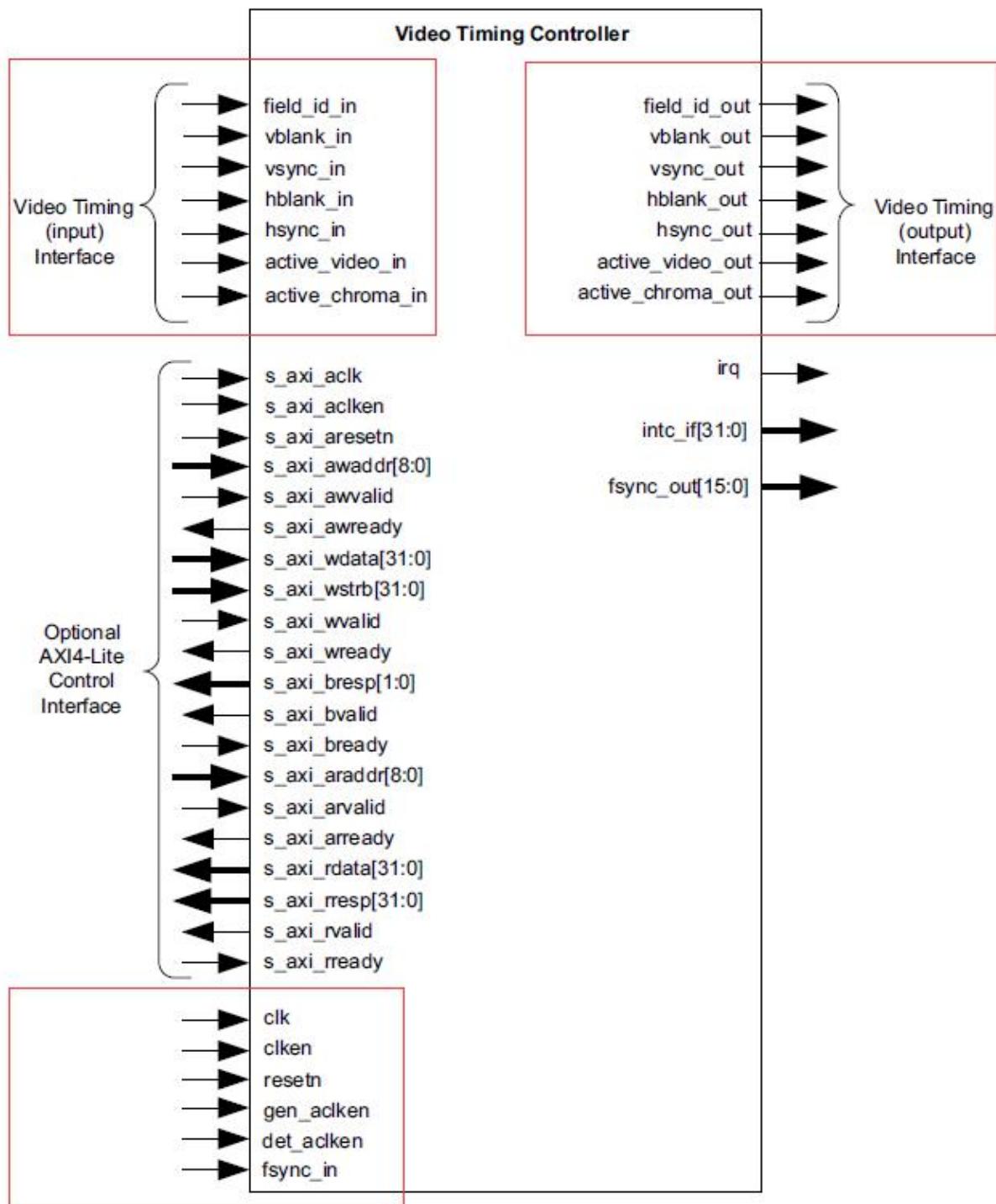
Enable Detection:

支持时序捕捉，这个不是必须的，根据需要而定，如果设置了这个选项，就可以先捕捉输入的时序，然后再设置输出的时序，实现输入和输出一致的效果。



在这个页面中，只要选择需要支持的分辨率就可以了，当然不设置也没关系的，因为我们在 C 代码综合那个会进一步设置的。

4.4.2 VTC IP 接口信号的定义



红色方框内的绝大部分信号需要我们手动联系，所以下面重点是讲解红色方框内的信号作用，至于 AXI4-LITE 接口主要是用来设置参数的。

Common Port Descriptions:

Name	Direction	Width	Description
clk	In	1	Video Core Clock
clken	In	1	Video Core Active High Clock Enable
det_clken	In	1	Video Timing Detection Core Active High Clock Enable
gen_clken	In	1	Video Timing Generator Core Active High Clock Enable
resetn	In	1	Video Core Active Low Synchronous Reset
irq	Output	1	Interrupt request output, active high edge
intc_if	Output	32	OPTIONAL EXTERNAL INTERRUPT CONTROLLER INTERFACE Available when the "Include INTC Interface" or C_HAS_INTC_IF has been selected. Bits [31:8] are the same as the bits [31:8] in the status register (0x0004). Bits [5:0] are the same as bits [21:16] of the error register (0x0008). Bits [7:6] are reserved and are always 0.

Detector Interface (Video Timing Input Interface)			
field_id_in	Input	1	INPUT FIELD ID Used to set the field_id polarity in the Detector Polarity Register (Address Offset 0x002C). Optional. Only valid when interlace support and field id are enabled.
hsync_in	Input	1	INPUT HORIZONTAL SYNCHRONIZATION Used to set the DETECTOR HSYNC register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hsync_in input is not connected, then the "Horizontal Sync Detection" option must be deselected.
hblank_in	Input	1	INPUT HORIZONTAL BLANK Used to set the DETECTOR HSIZE register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hblank_in input is not connected, then the "Horizontal Blank Detection" option must be deselected.
vsync_in	Input	1	INPUT VERTICAL SYNCHRONIZATION Used to set the DETECTOR F0_VSYNC_V and the F0_VSYNC_H registers. Polarity is auto-detected. Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vsync_in input is not connected, then the "Vertical Sync Detection" option must be deselected.

Name	Direction	Width	Description
vblank_in	Input	1	<p>INPUT VERTICAL BLANK Used to set the DETECTOR_VSIZE and the F0_VBLANK_H registers. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vblank_in input is not connected, then the "Vertical Blank Detection" option must be deselected.</p>
active_video_in	Input	1	<p>INPUT ACTIVE VIDEO Used to set the DETECTOR_ACTIVE_SIZE register. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the active_video_in input is not connected, then the "Active Video Detection" option must be deselected.</p>
active_chroma_in	Input	1	<p>INPUT ACTIVE CHROMA Used to set the VIDEO_FORMAT and the CHROMA_PARITY bits in the Detector Encoding Register. Polarity is auto-detected.</p> <p>Optional. If the active_chroma_in input is not connected, then the "Active Chroma Detection" option must be deselected.</p>

Generator Interface (Video Timing Output Interface)			
field_id_out	Output	1	<p>OUTPUT FIELD ID Generated field id signal. Polarity configured by the Generator Polarity Register (Address Offset 0x006C) Optional. Only enabled when interlaced support and field id generation is enabled.</p>
hsync_out	Output	1	<p>OUTPUT HORIZONTAL SYNCHRONIZATION Generated horizontal synchronization signal. Polarity configured by the control register. Asserted active during the cycle set by the HSYNC_START bits and deasserted during the cycle set by the HSYNC_END bits in the GENERATOR HSYNC register.</p>
hblank_out	Output	1	<p>OUTPUT HORIZONTAL BLANK Generated horizontal blank signal. Polarity configured by the control register. Asserted active during the cycle set by ACTIVE_HSIZEx and deasserted during the cycle set by the FRAME_HSIZEx bits in the GENERATOR HSIZEx register.</p>
vsync_out	Output	1	<p>OUTPUT VERTICAL SYNCHRONIZATION Generated vertical synchronization signal. Polarity configured by the control register. Asserted active during the line set by the F#_VSYNC_VSTART bits and deasserted during the line set by the F#_VSYNC_VEND bits in the GENERATOR F#_VSYNC_V registers.</p>

Name	Direction	Width	Description
vblank_out	Output	1	OUTPUT VERTICAL BLANK Generated vertical blank signal. Polarity configured by the control register. Asserted active during the line set by the ACTIVE_VSIZE bits and deasserted during the line set by the GENERATOR VSIZE register.
active_video_out	Output	1	OUTPUT ACTIVE VIDEO Generated active video signal. Polarity configured by the control register. Active for non blanking lines. Asserted active during the first cycle of the field/frame and deasserted during the cycle set by the GENERATOR ACTIVE_SIZE register
active_chroma_out	Output	1	OUTPUT ACTIVE CHROMA Generated active chroma signal. Denotes which lines contain valid chroma samples (used for YUV 4:2:0). Polarity configured by the GENERATOR POLARITY register. Active for non-blanking lines configured by the VIDEO_FORMAT and the CHROMA_PARITY bits in the GENERATOR Encoding Register.
Frame Synchronization Interface			
fsync_out	Output	[Frame Syncs - 1:0]	FRAME SYNCHRONIZATION OUTPUT Each Frame Synchronization bit toggles for only one clock cycle during each frame. The number of bits is configured with the Frame Syncs GUI parameter. Each bit is independently configured for horizontal and vertical clock cycle position with the Frame Sync 0-15 Config registers).
fsync_in	Input	1	FRAME SYNCHRONIZATION INPUT This is a one clock cycle pulse (active high) input. The video timing generator will be synchronized to the input if used.

本例子中没有使用到输入时序的捕捉，因此笔者下面只对用到的信号做一些介绍。

hsync_out:

产生行同步输出

hsync_out:

产生行消影

vsync_out:

产生场同步输出

vblank_out:

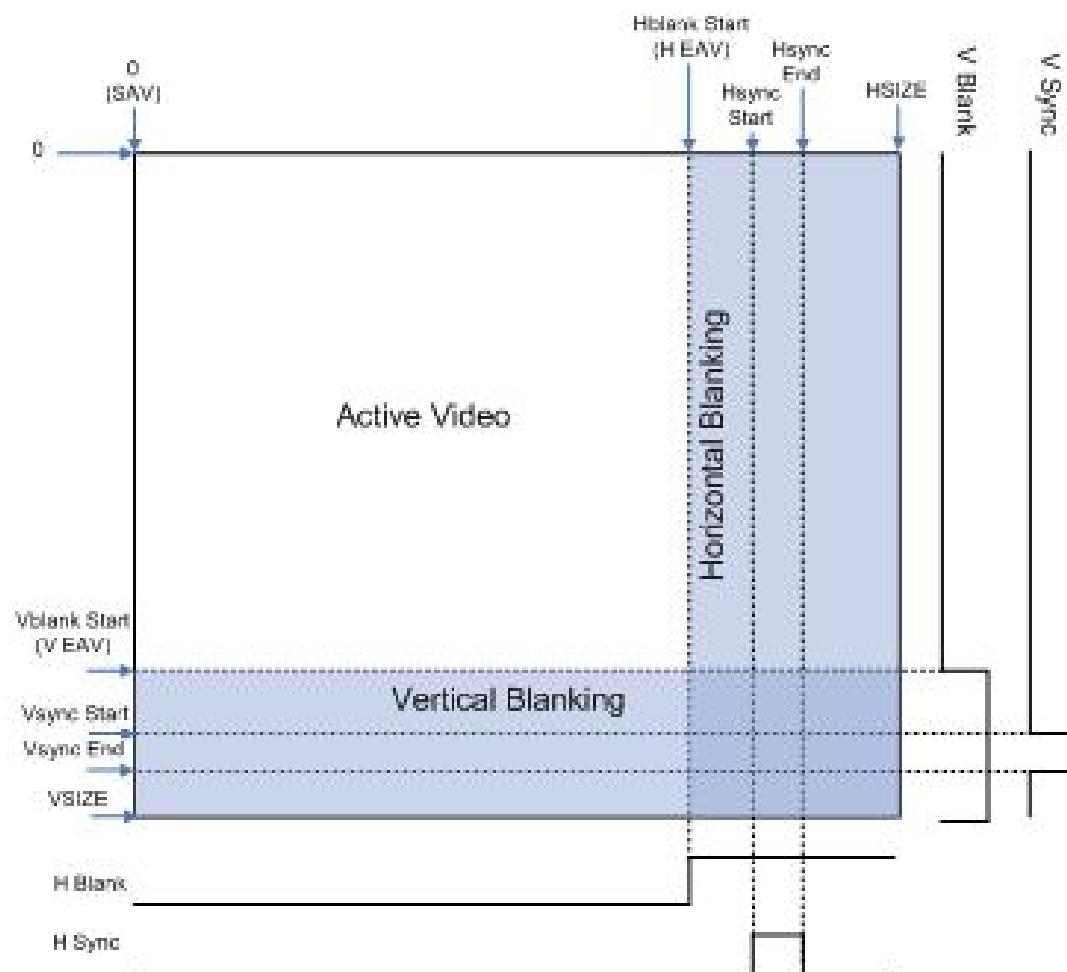
产生场消影

active_video_out:

有效数据输出

4.4.3 VTC IP 配置寄存器

shows the start of the horizontal front porch (Hblank Start), synchronization (Hsync Start), back porch (Hsync End) and active video (SAV). It also shows the start of the vertical front porch (Vblank Start), synchronization (Vsync Start), back porch (Vsync End) and active video (SAV). The total number of horizontal clock cycles is HSIZE and the total number of lines is the VSIZE.



Generator Active Size Register (Address Offset 0x0060)

0x0060	GENERATOR ACTIVE_SIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
ACTIVE_VSIZE	28:16	Generated Vertical Active Frame Size. The height of the frame without blanking in number of lines.
RESERVED	15:13	Reserved
ACTIVE_HSIZE	12:0	Generated Horizontal Active Frame Size. The width of the frame without blanking in number of pixels/clocks.

这是重要的寄存器用来设置有效的行数量和场数量

Generator Timing Status Register (Address Offset 0x0064)

0x0064	GENERATOR TIMING_STATUS	Read
Name	B its	Description
RESERVED	31:3	Reserved
GEN_ACTIVE_VIDEO	2	Generated Active Video Interrupt Status. Set high during the first cycle the output active video is asserted.
GEN_VBLANK	1	Generated Vertical Blank Interrupt Status. Set high during the first cycle the output vertical blank is asserted.
RESERVED	0	Reserved

GEN_ACTIVE_VIDEO:当第一帧图像输出时候置 1

GEN_VBLANK:第一帧有效图像的 blank 信号输出的时候置 1

Generator Encoding Register (Address Offset 0x0068)

0x0068	GENERATOR ENCODING	Read/Write
Name	B its	Description
RESERVED	31:10	Reserved
CHROMA_PARITY	9:8	Generated Chroma Parity 0: Chroma Active during even active-video lines of frame. Active every pixel of active line 1: Chroma Active during odd active-video lines of frame. Active every pixel of active line 2: Chroma Active during even active video lines of frame. Active every even pixel of active line, inactive every odd pixel 3: Chroma Active during odd active video lines of frame. Active every even pixel of active line, inactive every odd pixel
FIELD_ID_PARITY	7	Generated Field ID Parity 0: Field ID input is currently low 1: Field ID input is currently high
INTERLACED	6	Generated Progressive/Interlaced 0: Generated video format is progressive 1: Generated video format is interlaced
RESERVED	5:4	Reserved
VIDEO_FORMAT	3:0	Generated Video Format Denotes when the active_chroma signal is active. 0: YUV 4:2:2 - Active_chroma is active during the same time active_video is active. 1: YUV 4:4:4 - Active_chroma is active during the same time active_video is active. 2: RGB - Active_chroma is active during the same time active_video is active. 3: YUV 4:2:0- Active_chroma is active every other line during the same time active_video is active. See The CHROMA_PARITY bits to control which lines and pixels.

CHROMA_PARITY: 奇偶色度 (读者没明白)

FIELD_ID_PARITY: 奇偶场标志

INTERLACED: 视频格式是渐进式还是各行扫描

VIDEO_FORMAT: 视频格设置, 有 YUV422 YUV444 YUV420 RGB

Generator Polarity Register (Address Offset 0x006C)

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
RESERVED	31:7	Reserved
FIELD_ID_POL	6	Generated Field ID Polarity 0: Low during Field 0 and High during Field 1 1: High during Field 0 and Low during Field 1
ACTIVE_CHROMA_POL	5	Generated Active Chroma Polarity 0: Active Low Polarity 1: Active High Polarity

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
ACTIVE_VIDEO_POL	4	Generated Active Video Polarity 0: Active Low Polarity 1: Active High Polarity
HSYNC_POL	3	Generated Horizontal Sync Polarity 0: Active Low Polarity 1: Active High Polarity
VSYNC_POL	2	Generated Vertical Sync Polarity 0: Active Low Polarity 1: Active High Polarity
HBLANK_POL	1	Generated Horizontal Blank Polarity 0: Active Low Polarity 1: Active High Polarity
VBLANK_POL	0	Generated Vertical Blank Polarity 0: Active Low Polarity 1: Active High Polarity

这个寄存器设置相应的场输出极性和色度输出极性。

Generator Horizontal Frame Size Register (Address Offset 0x0070)

0x0070	GENERATOR HSIZE	Read/Write
Name	B its	Description
RESERVED	31:13	Reserved
FRAME_HSIZE	12:0	Generated Horizontal Frame Size. The width of the frame with blanking in number of pixels/clocks.

一副图像的一行的大小，包括了消隐和有效数据阶段。

Generator Vertical Frame Size Register (Address Offset 0x0074)

0x0074	GENERATOR VSIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
FIELD1_VSIZE	28:16	Generated Vertical Field 1 Size. The height with blanking in number of lines of field 1.
FRAME_VSIZE	12:0	Generated Vertical Frame Size. The height of the frame with blanking in number of lines.

一副图像的一场的大小，包括了消隐和有效数据阶段。

Generator Horizontal Sync Register (Address Offset 0x0078)

0x0078	GENERATOR HSYNC	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
Hsync_End	28:16	Generated Horizontal Sync End End cycle index of horizontal sync. Denotes the first cycle hsync_in is de-asserted.
RESERVED	15:13	Reserved
Hsync_Start	12:0	Generated Horizontal Sync End Start cycle index of horizontal sync. Denotes the first cycle hsync_in is asserted.

设置行的水平同步结束和同步开始

Generator Frame/Field 0 Vertical Blank Cycle Register (Address Offset 0x007C)

0x007C	GENERATOR F0_VBLANK_H	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VBLANK_HEND	28:16	Generated Vertical Blank Horizontal End End Cycle index of vertical blank. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F0_VBLANK_HSTART	12:0	Generated Vertical Blank Horizontal Start Start Cycle index of vertical blank. Denotes the first cycle vblank_in is asserted.

设置 Fram/Field0 的水平消隐结束和开始

Generator Frame/Field 0 Vertical Sync Line Register (Address Offset 0x0080)

0x0080	GENERATOR F0_VSYNC_V	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VSYNC_VEND	28:16	Generated Vertical Sync Vertical End End Line index of vertical sync. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_VSTART	12:0	Generated Vertical Sync Vertical Start Start line index of vertical sync. Denotes the first line vsync_in is asserted.

设置 Fram/Field0 的垂直同步垂直结束和开始

Generator Frame/Field 0 Vertical Sync Cycle Register (Address Offset 0x0084)

0x0084	GENERATOR F0_VSYNC_H	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
F0_VSYNC_HEND	28:16	Generated Vertical Sync Horizontal End End cycle index of vertical sync. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_HSTART	12:0	Generated Vertical Sync Horizontal Start Start cycle index of vertical sync. Denotes the first cycle vsync_in is asserted.

设置 Fram/Field0 的垂直同步水平结束和开始

Generator Field 1 Vertical Blank Cycle Register (Address Offset 0x0088)

0x0088	GENERATOR F1_VBLANK_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VBLANK_HEND	28:16	Generated Field 1 Vertical Blank Horizontal End End Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F1_VBLANK_HSTART	12:0	Generated Field 1 Vertical Blank Horizontal Start Start Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is asserted.

设置 Field1 的水平消隐结束和开始

Generator Field 1 Vertical Sync Line Register (Address Offset 0x008C)

0x008C	GENERATOR F1_VSYNC_V	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_VEND	28:16	Generated Field 1 Vertical Sync Vertical End End Line index of vertical sync for field 1. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_VSTART	12:0	Generated Field 1 Vertical Sync Vertical Start Start line index of vertical sync for field 1. Denotes the first line vsync_in is asserted.

设置 Field1 的垂直同步垂直结束和开始

Generator Field 1 Vertical Sync Cycle Register (Address Offset 0x0090)

0x0090	GENERATOR F1_VSYNC_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_HEND	28:16	Generated Field 1 Vertical Sync Horizontal End End cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_HSTART	12:0	Generated Field 1 Vertical Sync Horizontal Start Start cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is asserted.

设置 Field1 的垂直同步水平结束和开始

Frame Sync 0 - 15 Configuration Registers (Address Offsets 0x0100 - 0x013C)

0x0100	FRAME SYNC 0 CONFIG	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
V_START	28:16	FRAME SYNCHRONIZATION VERTICAL START Vertical line during which the fsync_out[0] output port is asserted active-high. Note: Frame Syncs are not active during the complete line, only in the cycle during which both the V_START and H_START are valid each frame.
RESERVED	15:13	Reserved
H_START	12:0	FRAME SYNCHRONIZATION HORIZONTAL START Horizontal Cycle during which fsync_out[0] output port is asserted active-high

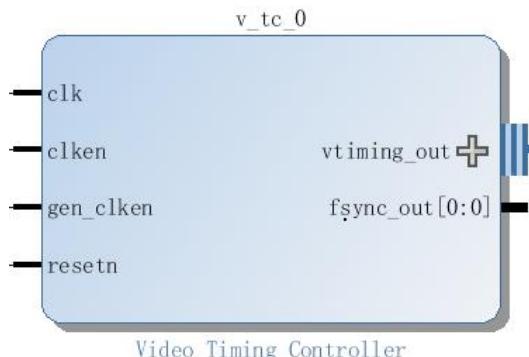
Generator Global Delay Register (Address Offset 0x140)

0x140	Generator Global Delay	Read/Write
Name	Bits	Description
Reserved	31:29	Reserved
V_DELAY	28:16	GENERATOR VERTICAL DELAY Vertical line offset. This is the number of lines that the generated output will be shifted relative to the detector (input timing). The vertical delay is only available when both the detector and generator are enabled. Can be combined with the H_DELAY.

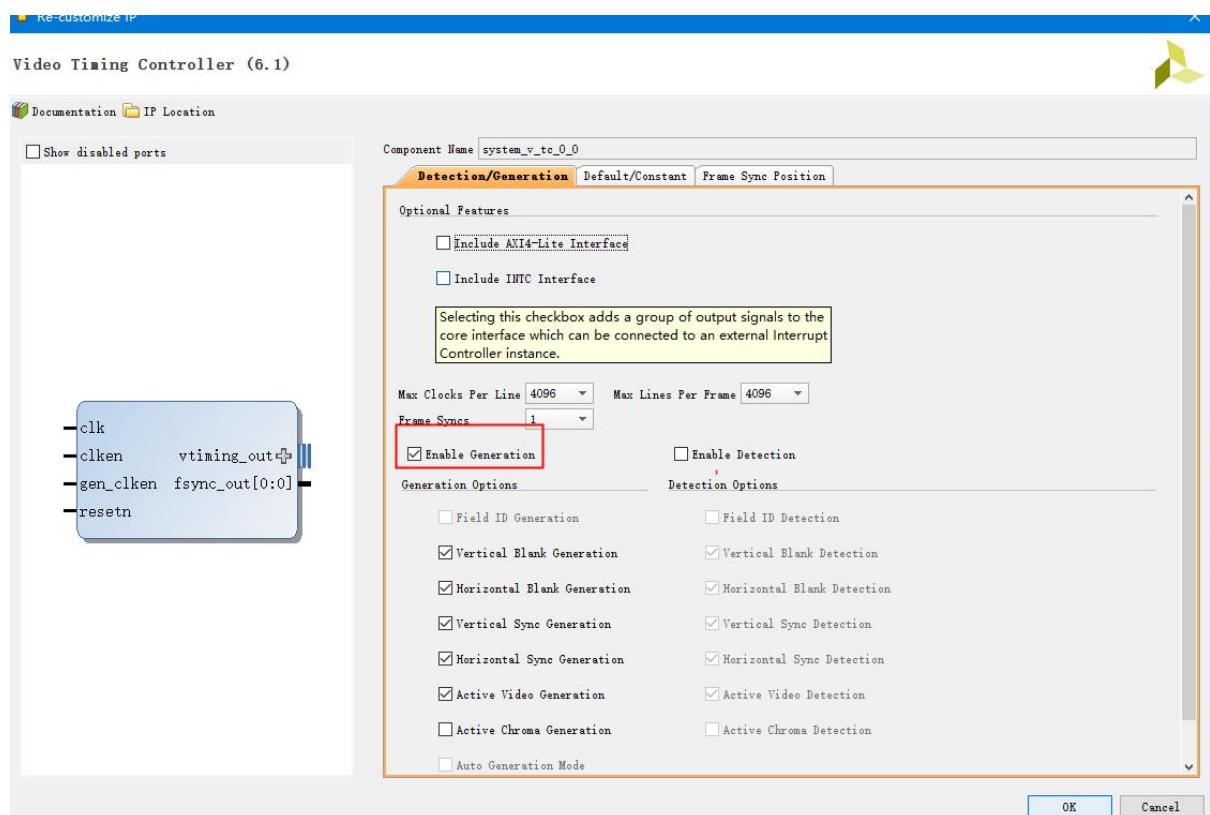
Reserved	15:13	Reserved
H_DELAY	12:0	GENERATOR HORIZONTAL DELAY Horizontal cycle offset. This is the number of clock cycles that the generated output will be shifted relative to the detector (input timing). The horizontal delay is only available when both the detector and generator are enabled. Can be combined with the V_DELAY.

4.4.5 设置 VTC IP

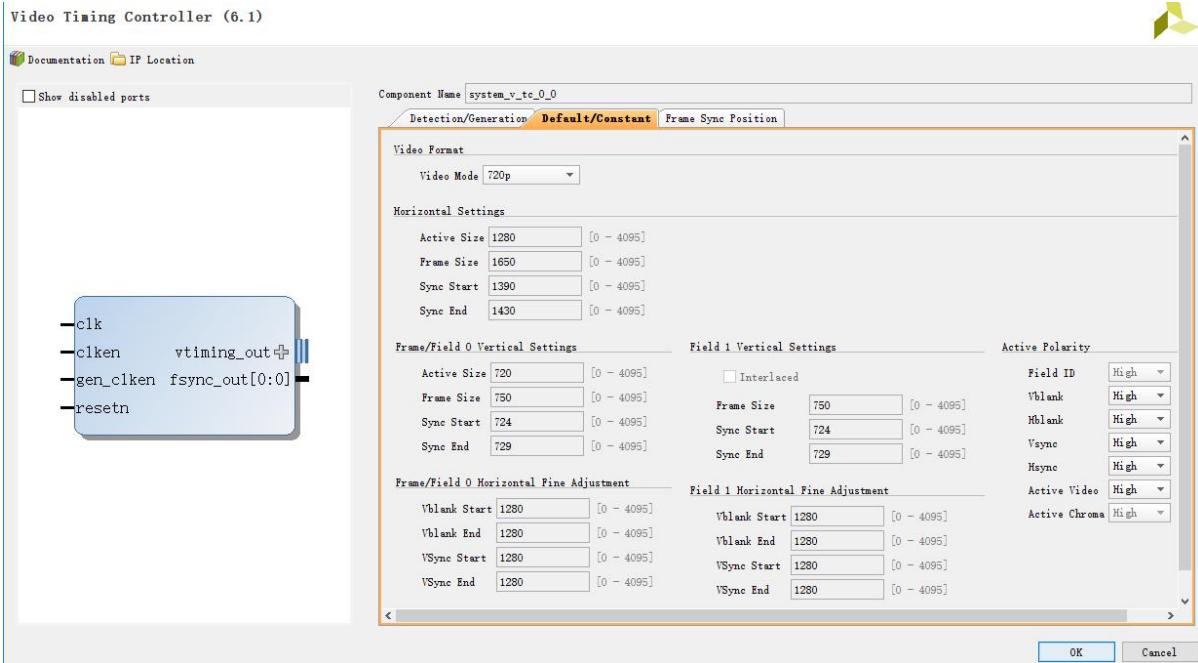
讲了这么多实际上我们用的时候很简单,所以只要这么简单。



由于不使用动态配置，并且只使用了视频时序产生，所以只要勾选如下复选框。

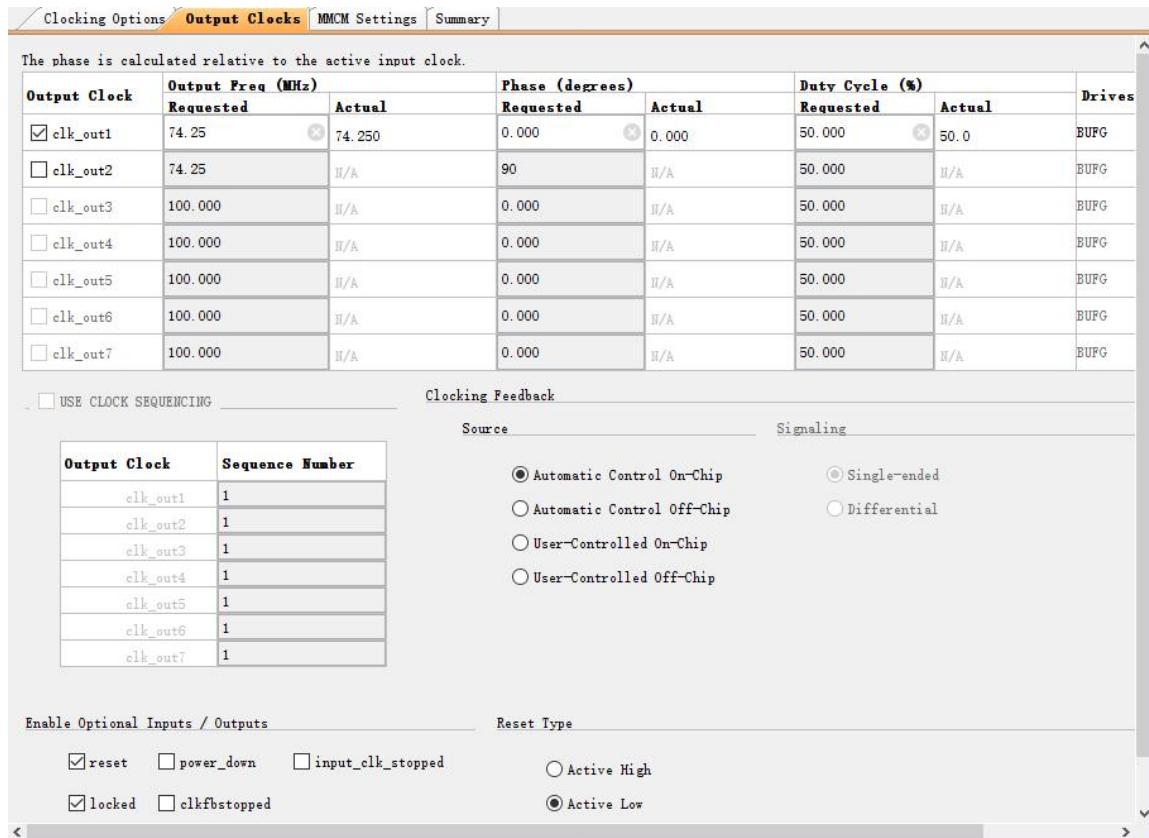


由于 OV5640 分辨率是 1280X720 因此直接选择 720P 即可。



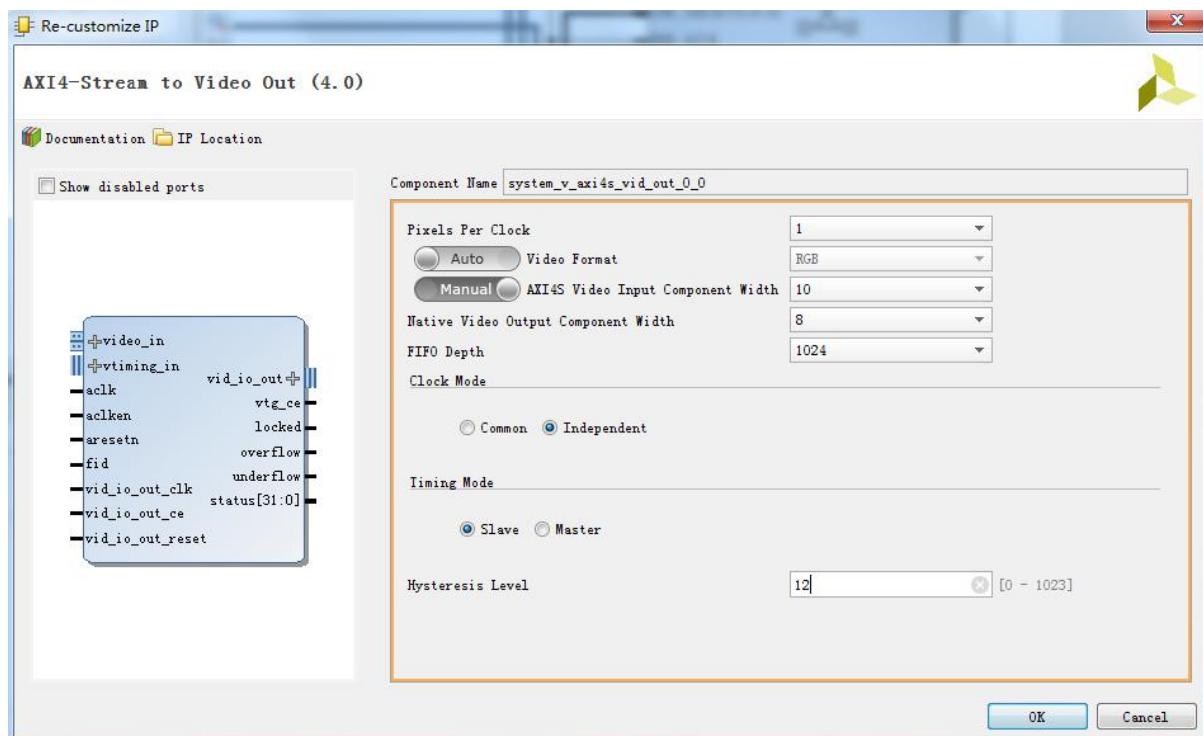
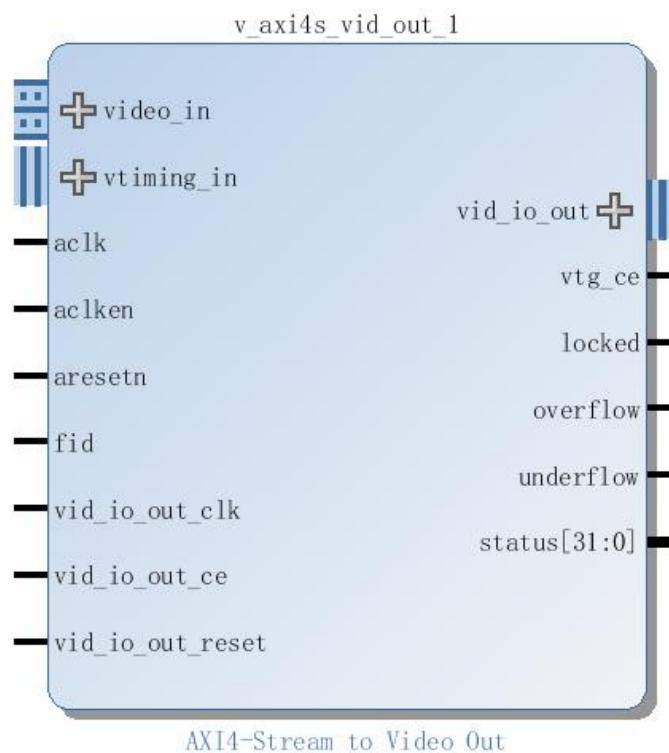
4.6 PLL 时钟设置

由于这里的分辨率是 1280X720 因此提供给 VTC IP 和 VID OUTIP 的时钟 74.5M 就可以了



4.7 VID_OUT IP 的分析

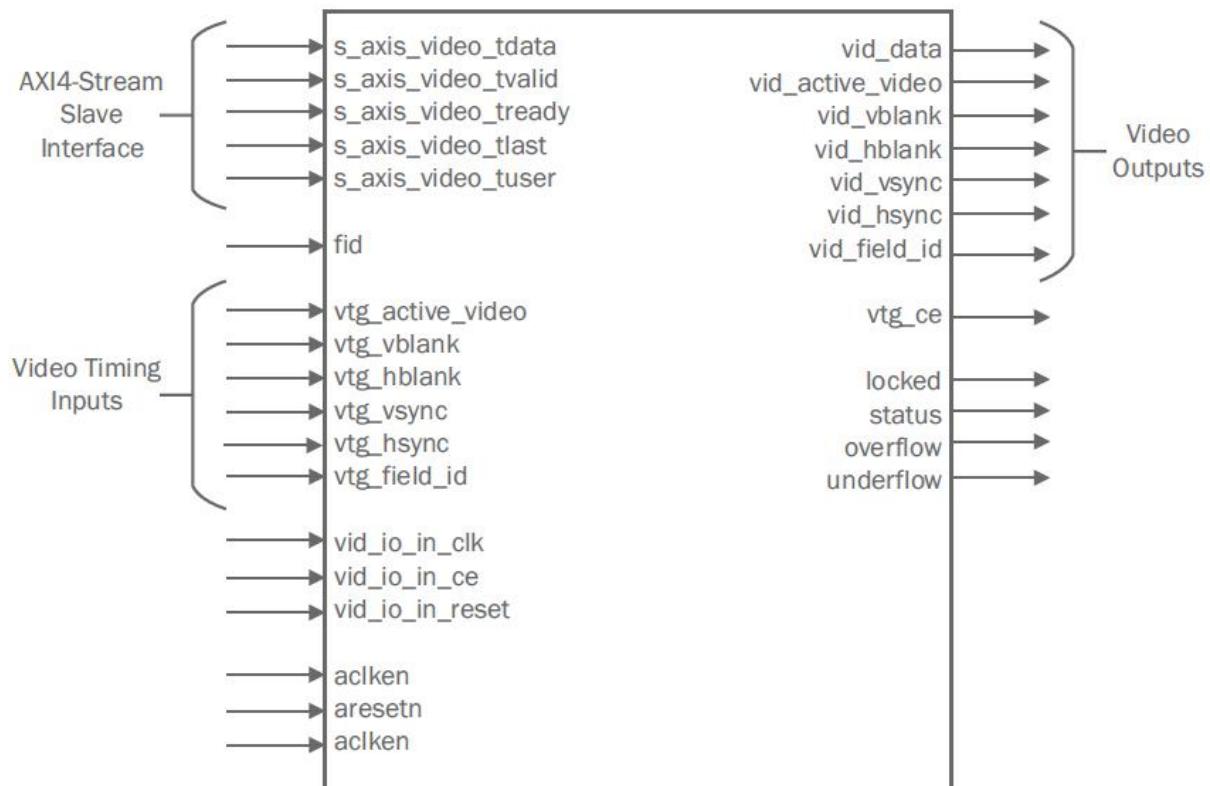
4.7.1 VID_OUT 的参数介绍



这些参数和前面的 V_TPG 参数类似

- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟
- Video Format: 视频格式
- FIFO Depth: FIFO 深度
- Hysteresis Level: 滞后输出

4.7.2 VID_OUT IP 接口信号的定义



Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_out_ce	Input	1	Native video clock enable
vid_io_out_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
vtg_ce	Output	1	VTC clock enable. Used to halt the timing generator for synchronization purposes.
locked	Output	1	Flag indicating whether the VTC is locked to the input timing. 1=locked. Synchronous to vid_io_out_clk.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk.

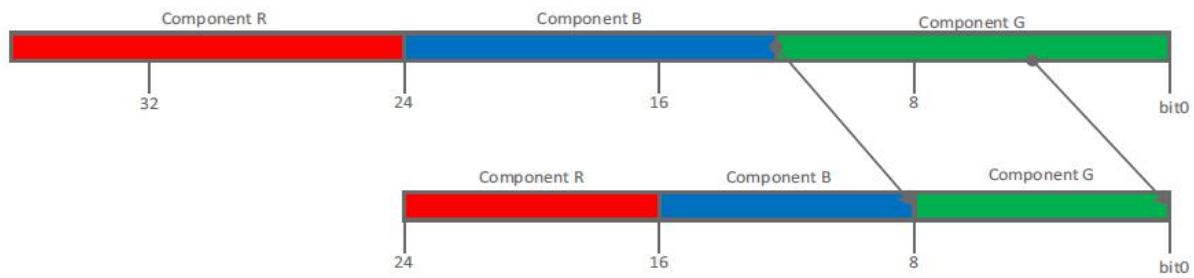
Video Timing Interface

Signal Name	Direction	Width	Description
vtg_vsync	In	1	VTC vertical sync. Active High
vtg_hsync	In	1	VTC horizontal sync. Active High
vtg_vblank	In	1	VTC vertical blank. Active High
vtg_hblank	In	1	VTC horizontal blank. Active High
vtg_act_vid	In	1	VTC active video signal. 1 = active video, 0 = blanked video
vtg_field_id	In	1	VTC field ID. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

AXI4 - Stream Interface

Signal Name	Direction	Width	Description
s_axis_video_tvalid	Input	1	AXI4-Stream TVALID. Active video data enable
s_axis_video_tuser	Input	1	AXI4-Stream TUSER. Start of Frame
s_axis_video_tlast	Input	1	AXI4-Stream TLAST. End of Line
s_axis_video_tready	Output	1	AXI4-Stream TREADY. Inverted FIFO full

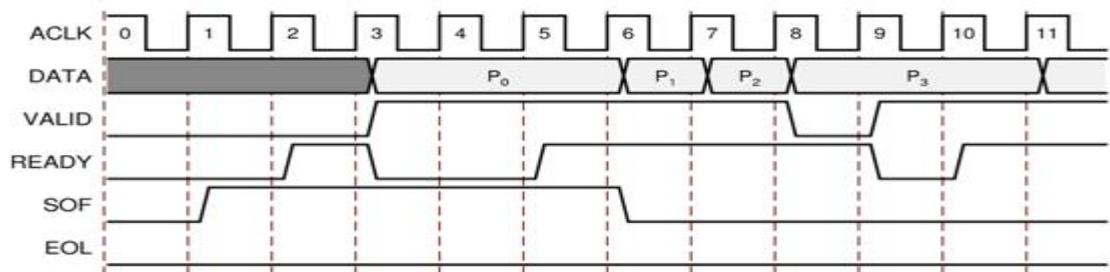
对于 s_axis_video_tdata(TDATA)需要注意一些事情,一般情况下我们的 RGB888 输出,但是,如果 s_axis_video_tdata 是 32bit 那么 VID_OUT IP 会自动截取到 24bit。由于技术手册只给出了 12bit 到 8bit 的截取方式,也就是 RGB 12:12:12 到 RGB 8:8:8 如下图:



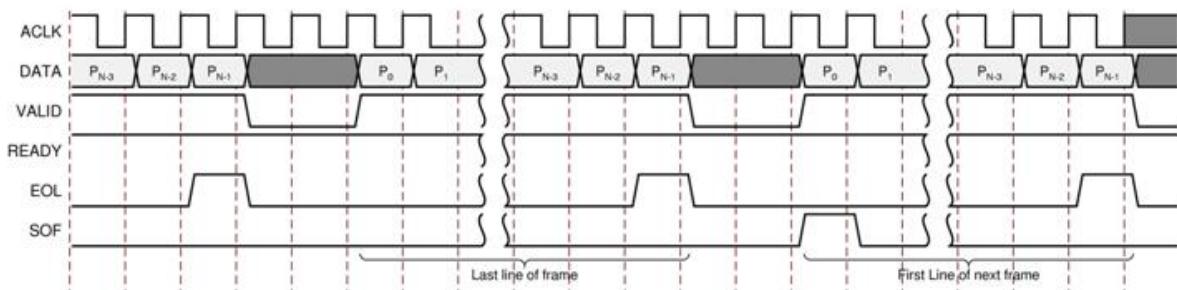
这种截取比较简单，把每个色度的低 4bit 截取就可以了。但是如果是 RGB10:10:10，官方并没有给出截取方式，但是可以通过纯色输出来进行测试。

因此最简单的办法是无需任何截取了，如果 s_axis_video_tdata 是 RGB8:8:8 那就无需任何截取，笔者设计的时候由于 AXI 总线是 32bit 因此数据的低 24bit 为 RGB 8:8:8 只要去掉高 24-31bit 就可以取得 RGB8:8:8，这样最省事。

以下时序图是在 SOF 是一帧图像的开始，当 VALID 和 READY 有效的时候开始传输数据。



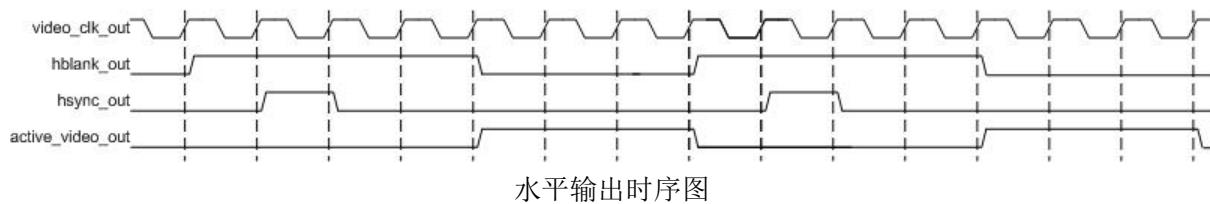
EOL 代表每一行的最后一个数据，SOF 代表前一帧的最后一行的结束，下一帧第一行的开始。SOF 为 1 个 PLUS 有效 (pg044_v_axis_out.pdf 没有描述清楚，而且有错误)。



Example Horizontal Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0003_0003
0x0070	Generator HSize	0x0000_0007
0x0078	Generator HSync	0x0005_0004
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

设置水平输出的相关寄存器

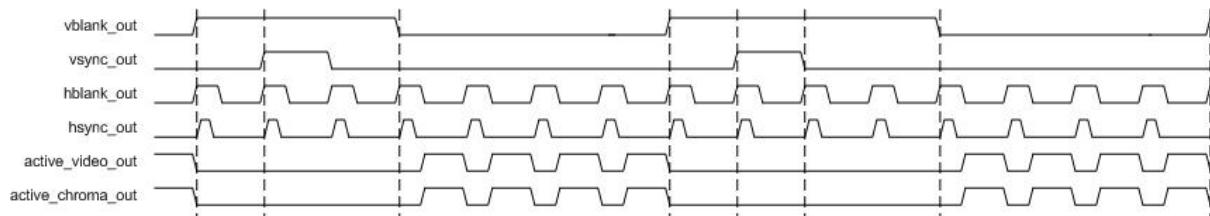


水平输出时序图

Example Vertical Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0004_0003
0x0070	Generator HSize	0x0000_0007
0x0074	Generator VSize	0x0000_0008
0x0078	Generator HSync	0x0005_0004
0x0080	Generator Frame 0 Vsync	0x0006_0005
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

设置垂直输出的相关寄存器



垂直输出时序图

4.8 FPGA 实现的用户逻辑代码

4.8.1 关键信号 1

```
assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready &
m_axis_video_tlast &(vid_in_v_cnt == VID_IN_VS); // dma in last signal
```

m_axis_video_tvalid:此信号是 vid in IP 输出的，代表输出数据有效

s_axis_s2mm_tready:此信号是 DMA IP 输出的，代表 DMA 可以接收数据

m_axis_video_tlast:这是每一行图像数据的最后一个像素的信号标志

vid_in_v_cnt == VID_IN_VS: 表示一副图像的最后一个像素输出。
 s_axis_s2mm_tlast: 所有这些信号有效的时候代表 DMA 的最后一个数据
 s_axis_s2mm_tlast 信号有效。

4.8.2 关键信号 2

```
assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready &
(vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); //vid out user
m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。
s_axis_video_tready: vid out IP 准备好了, 可以接收数据
(vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); 行计数器为 0 场计数器也为 0 说明要么这副图像已经结束, 也可以理解为下一副图像开始前。这样结合
s_axis_video_tready, m_axis_mm2s_tvalid 为 1, 基于 FPGA 时序, 下一个时钟输出
s_axis_video_tuser 为 1 正好是一副图像的第一个像素。
s_axis_video_tuser: 因此 s_axis_video_tuser 代表了每一副图像开始的第一个像素。
```

4.8.3 关键信号 3

```
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt ==
VID_OUT_HS); //vid out last signal
```

m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。
 s_axis_video_tready: vid out IP 准备好了, 可以接收数据
 vid_out_h_cnt == VID_OUT_HS: 图像一行数据的最后一个像素。

4.8.4 部分关键代码

表 3-6-4-1

<pre>reg [10:0] vid_out_v_cnt; reg [10:0] vid_out_h_cnt; reg [10:0] vid_in_v_cnt; parameter VID_OUT_HS = 11'd1279;//图像输出行分辨率 parameter VID_OUT_VS = 11'd719;//图像输出场分辨率 parameter VID_IN_VS = 11'd719; always@(posedge FCLK_CLK0) begin if(!gpio_rtl_tri_o_0)</pre>
--

```

    vid_out_v_cnt <= 11'd0;
else
    if(m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == VID_OUT_HS))
        if(vid_out_v_cnt != VID_OUT_VS)
            vid_out_v_cnt <= vid_out_v_cnt + 1'b1;
        else
            vid_out_v_cnt <= 11'd0;
    else
        vid_out_v_cnt <= vid_out_v_cnt;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_out_h_cnt <= 11'd0;
    else
        if(m_axis_mm2s_tvalid & s_axis_video_tready)
            if(vid_out_h_cnt != VID_OUT_HS)
                vid_out_h_cnt <= vid_out_h_cnt + 1'b1;
            else
                vid_out_h_cnt <= 11'd0;
        else
            vid_out_h_cnt <= vid_out_h_cnt;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_in_v_cnt <= 11'd0;
    else
        if(m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast)
            if(vid_in_v_cnt != VID_IN_VS)
                vid_in_v_cnt <= vid_in_v_cnt + 1'b1;
            else
                vid_in_v_cnt <= 11'd0;
        else
            vid_in_v_cnt <= vid_in_v_cnt;
end

assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == 11'd0) &
(vid_out_v_cnt == 11'd0); //vid out user
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == VID_OUT_HS); //vid out last signal

```

```
assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast
&(vid_in_v_cnt == VID_IN_VS); // dma in last signal
```

4.9 PS 部分

4.9.1 DMA 中断函数部分分析

为了让图像输出高品质效果，PS 部分设计了 3 缓存处理机制。3 缓存处理机制在大量图像缓冲处理方法是最有效的办法之一。

在 DMA_intr.h 文件中，定义 3 段内存空间用于保存三副最新的图像。

```
#define BUFFER0_BASE (MEM_BASE_ADDR)
#define BUFFER1_BASE (MEM_BASE_ADDR + IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define BUFFER2_BASE (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
```

在 DMA_intr.h 文件中，还定义一下 2 个变量 1 个指针数组。tx_buffer_index; 指示了当前的发送缓存序号，rx_buffer_index; 指示了当前的接收缓存序号。*BufferPtr[3] 会被制定到对应的内存地址空间。

```
extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;
extern u32 *BufferPtr[3];
```

在 main 函数里面有这么一段实现了指针数组指向内存地址空间。

```
BufferPtr[0] = (u32 *)BUFFER0_BASE;
BufferPtr[1] = (u32 *)BUFFER1_BASE;
BufferPtr[2] = (u32 *)BUFFER2_BASE;
```

下面给出 dma_intr.h 的完整代码

表 3-7-1-1 dma_intr.h

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 */
#ifndef DMA_INTR_H
#define DMA_INTR_H
#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
```

```
#include "xscugic.h"

/********************* Constant Definitions *****/
/*
 * Device hardware build related constants.
 */

#define DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID

#define MEM_BASE_ADDR      0x10000000

#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR

#define IMAGE_WIDTH      1280
#define IMAGE_HEIGHT     720
#define BYTES_PER_PIXEL   4
#define BUFFER_NUM       2

#define MEM_BASE_ADDR      0x10000000

#define BUFFER0_BASE      (MEM_BASE_ADDR )
#define BUFFER1_BASE      (MEM_BASE_ADDR +     IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)
#define BUFFER2_BASE      (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)

/* Timeout loop counter for reset
 */
#define RESET_TIMEOUT_COUNTER    10000
/* test start value
 */
#define TEST_START_VALUE    0xC
/*
 * Buffer and Buffer Descriptor related constant definition
 */
#define MAX_PKT_LEN      (IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
/*
 * transfer times
 */
```

```
/*
#define NUMBER_OF_TRANSFERS 100000

extern volatile int TxDone;
extern volatile int RxDone;
extern volatile int Error;

extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;

extern u32 *BufferPtr[3];

int DMA_CheckData(int Length, u8 StartValue);
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr);
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);
#endif
```

每次一幅图像通过 DMA 进入 DDR 后，会产生 DMA 中断请求，在 DMA 中断请求中，会指定下一次 DMA 接收的 buffer 位置。

表 3-7-1-2 DMA_RxIntrHandler 函数

```
/****************************************************************************
*
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.
*
* @param    Callback is a pointer to RX channel of the DMA engine.
*
* @return   None.
*
* @note    None.
*
****************************************************************************/
static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    u32 Status;
```

```
int TimeOut;
XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

/* Read pending interrupts */
IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);

/* Acknowledge pending interrupts */
XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

/*
 * If no interrupt is asserted, we do not do anything
 */
if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
    return;
}

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
*/
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
    xil_printf("rx error! \r\n");
    return;
}

/*
 * If completion interrupt is asserted, then set RxDone flag
*/
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
    RxDone++;
}

if(rx_buffer_index == 2)
    rx_buffer_index = 0;
else
    rx_buffer_index++;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[rx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma failed! 0 %d\r\n", Status);
```

```
    return;  
}  
  
}
```

发送函数通过 tx_buffer_index 标记需要发送的缓存部分,并且确保发送的是最新保存的一副图像。

表 3-7-3 DMA_TxIntrHandler

```
/**************************************************************************/  
/*  
* This is the DMA TX Interrupt handler function.  
*  
* It gets the interrupt status from the hardware, acknowledges it, and if any  
* error happens, it resets the hardware. Otherwise, if a completion interrupt  
* is present, then sets the TxDone.flag  
*  
* @param    Callback is a pointer to TX channel of the DMA engine.  
*  
* @return   None.  
*  
* @note    None.  
*  
**************************************************************************/  
static void DMA_TxIntrHandler(void *Callback)  
{  
  
    u32 IrqStatus;  
    u32 Status;  
    int TimeOut;  
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;  
  
    /* Read pending interrupts */  
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE);  
  
    /* Acknowledge pending interrupts */  
  
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);  
  
    /*  
     * If no interrupt is asserted, we do not do anything  
     */  
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
```

```
        return;
    }

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
 */
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

    //Error = 1;
    xil_printf("tx error! \r\n");
    return;
}

/*
 * If Completion interrupt is asserted, then set the TxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    TxDone++;
}

if(rx_buffer_index == 0)
    tx_buffer_index = 2;
else
    tx_buffer_index = rx_buffer_index - 1;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[tx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma failed! 0 %d\r\n", Status);
    return;
}
}
```

4.9.2 main.c 文件

这个主程序比较简单，内容比上一个课程的精简很多，这里需要注意的地方是 XGpio_DiscreteWrite(&Gpio, 1, 1); 函数这个函数是这只摄像头和 DMA 之间数据同步的，没有这个同步图像容易错位。另外在主函数里面首先启动 DMA 接收和发送中断各一次，以后就可以在中断里

面继续触发了。

表 3-7-2-1 main.c

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 * axi dma test
 *
 */

#include "dma_intr.h"
#include "sys_intr.h"
#include "xgpio.h"

volatile int TxDone;
volatile int RxDone;
volatile int Error;

volatile u8 tx_buffer_index;
volatile u8 rx_buffer_index;

u32 *BufferPtr[3];

static XScuGic Intc; //GIC
static XAxiDma AxiDma;
static XGpio Gpio;

#define AXI_GPIO_DEV_ID           XPAR_AXI_GPIO_0_DEVICE_ID

int init_intr_sys(void)
{
    DMA_Intr_Init(&AxiDma,0); //initial interrupt system
    Init_Intr_System(&Intc); // initial DMA interrupt system
    Setup_Intr_Exception(&Intc);
    DMA_Setup_Intr_System(&Intc,&AxiDma,TX_INTR_ID,RX_INTR_ID); //setup dma interrupt
    system
    DMA_Intr_Enable(&Intc,&AxiDma);
}

int main(void)
{
```

u32 Status;

```
BufferPtr[0] = (u32 *)BUFFER0_BASE;  
BufferPtr[1] = (u32 *)BUFFER1_BASE;  
BufferPtr[2] = (u32 *)BUFFER2_BASE;
```

```
tx_buffer_index = 0;  
rx_buffer_index = 0;  
TxDone = 0;  
RxDone = 0;  
Error = 0;
```

```
XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);  
XGpio_SetDataDirection(&Gpio, 1, 0);  
init_intr_sys();
```

Miz702_EMIO_init();Ov5640_init_rgb();

```
XGpio_DiscreteWrite(&Gpio, 1, 1);  
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[rx_buffer_index],  
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
```

```
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[tx_buffer_index],  
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
```

```
while (1);  
    return XST_SUCCESS;
```

{}

4.10 实验效果



S03_CH05_AXI_DMA_HDMI 图像输出

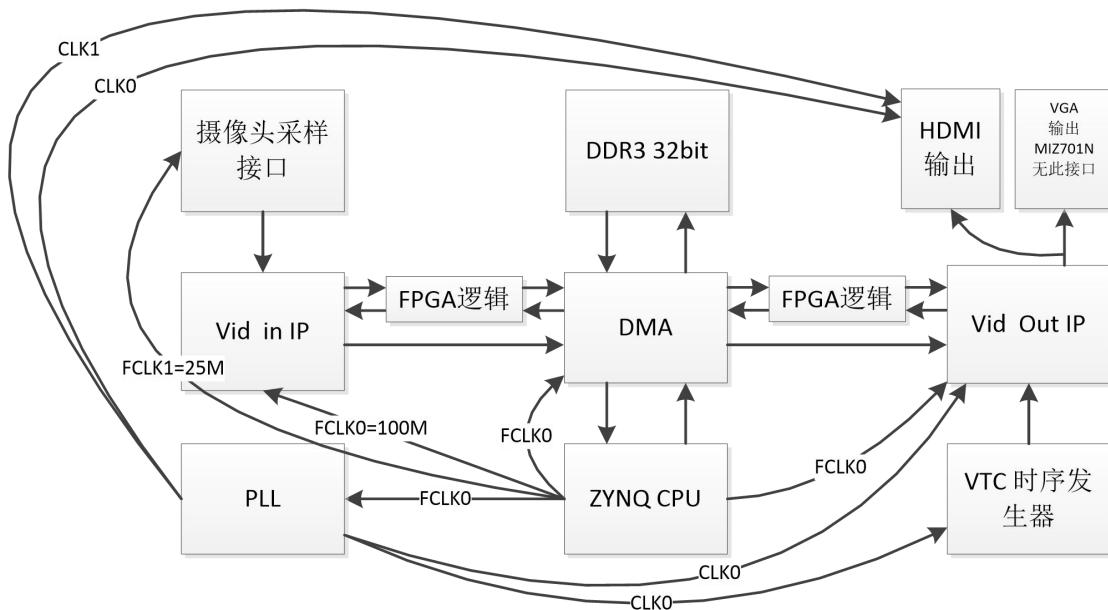
5.1 概述

本课程是在前面课程基础上添加 HDMI IP 实现 HDMI 视频图像的输出。本课程出了多了 HDMI 输出接口，其他内容和《S03_CH03_AXI_DMA_OV7725 摄像头采集系统》。本章课程内容使用的也是 OV7725 摄像头，但是课后代码会给出 OV5640 的配套代码。下面的内容除了涉及到 HDMI 部分的，其他和《S03_CH03_AXI_DMA_OV7725 摄像头采集系统》。

《S03_CH03_AXI_DMA_OV7725 摄像头采集系统》、《S03_CH04_AXI_DMA_OV5640 摄像头采集系统》、《S03_CH05_AXI_DMA_HDMI 图像输出》。读者可以根据自己需求情况而阅读，请知悉。

5.2 系统构架

5.2.1 构架方案图

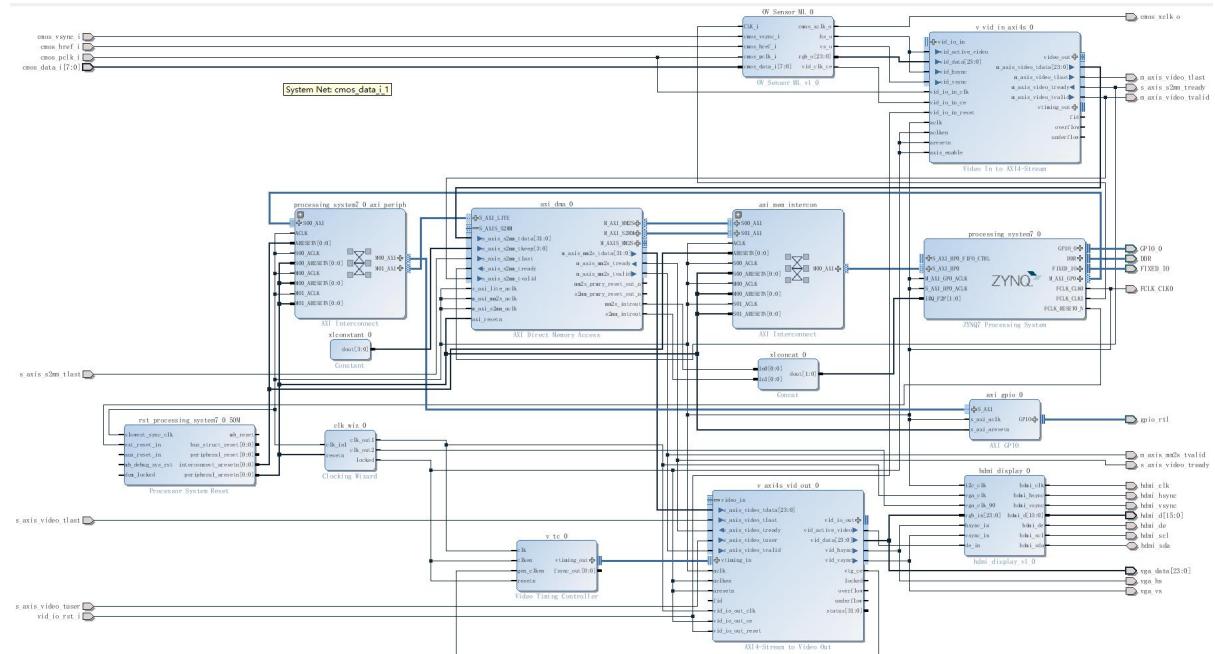


摄像头接口采集的摄像头数据，进过 vid in 视频输入 IP 后，还需要通过用户 FPGA 逻辑编程，和 DMA IP 之间实现握手协议，实现把数据通过 DMA 写入到 DDR。每次写入一副图像的数据后，产生一次接收中断，接收中断函数，会把数据三缓存后，在通过 DMA 发出去，DMA 发送完成后产生中断，在中断中，把缓存好的图像发送出去。DMA 发送的数据需要发送到 vid out 视频输出 IP。同理，DMA 和 vid out IP 之间也许需要增加 FPGA 用户代码实现接口的握手协议。数据进入 vid out 后，会随同 vtc IP 输出

符合 VGA 时序的图像信号。vid_out 的输出就可以直接定义成 VGA 信号输出。

5.2.2 构 BLOCK 模块化设计方案图

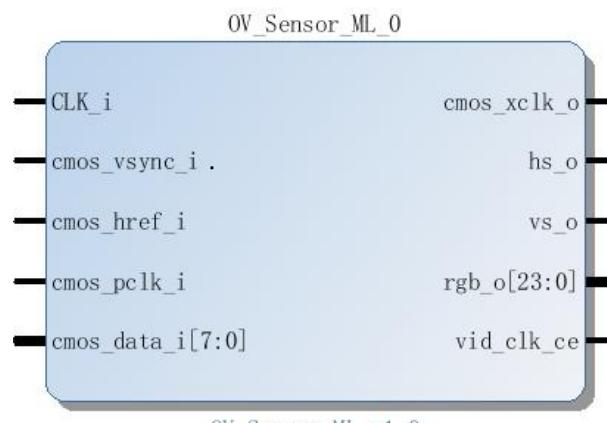
MIZ702/MIZ702N 的 HDMI(ADV7511 HDMI 芯片方案)显示构架图



MIZ701N 的 HDMI(FPGA IO 模拟 HDMI 时序方案)显示构架图

5.3 vid_in IP 介绍

5.3.1 OV_Sensor_ML 自定义 IP 模块



外部信号接口说明：

CLK_i : 为输入时钟，通常接 24MHZ 或者 25MHZ

Cmos_xclk_o:摄像头工作，通常直接把 CLK_i 连接到 cmos_xclk_o

Cmos_vsync_i: 摄像头场同步输入 上升沿代表场同步开始

Cmos_href_i:摄像头行同步输入 高电平代表行数据有效

Cmos_data[7:0]:摄像头数据输入

Hs_o:采集 OV_Sensor_ML IP 输出的行数据有效

Vs_o:采集 OV_Sensor_ML IP 输出的场同步信号

Vid_clk_ce:此信号用于和 vid_in IP 的时钟同步(由于 OV_Sensor_ML IP 是每两个时钟输出一次 rgb[23:0]的图像数据，因此需要通过 Vid_clk_ce 对时钟频率进行同步，有了这个信号，可以解决输入采集 IP 和 vid_in IP 数据接口之间的同步问题).

OV_Sensor_ML IP 包含 3 个源程序文件，分别为 OV_Sensor_ML.v、cmos_decode_v1.v、count_reset_v1.v 文件。

OV_Sensor_ML.v 程序中，对 cmos_data_i、cmos_href_i、cmos_vsync_i 做了一次寄存器，笔者发现图像效果有所改观。笔者分析，是因为寄存后有利于去除一些毛刺信号，提高了数据的稳定性。

表 3-3-1-1 OV_Sensor_ML 源码 OV_Sensor_ML.v

```
'timescale 1ns / 1ps
///////////////////////////////
// Company: milinker
// Engineer:tangjinyuan
//
// Create Date:    15:54:59 11/21/2015
// Design Name:
// Module Name:   OV7725_IP_ML
// Project Name: OV7725_IP_ML
// Target Devices: ZYNQ
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////
module OV_Sensor_ML(
    input CLK_i,
    //----- CMOS sensor hardware interface -----
    input     cmos_vsync_i, //cmos vsync
    input     cmos_href_i, //cmos hsync refrence
    input     cmos_pclk_i, //cmos pxiel clock
    output    cmos_xclk_o, //cmos externl clock
);
```

```
input[7:0]cmos_data_i, //cmos data
output hs_o,//hs signal.
output vs_o,//vs signal.
// output de_o,//data enable.
output [23:0] rgb_o,//data output,
output vid_clk_ce

);
-----视频输出解码模块-----
wire [15:0]rgb_o_r;
assign rgb_o = {rgb_o_r[4:0] ,3'd0 ,rgb_o_r[10:5] ,2'd0,rgb_o_r[15:11],3'd0};

reg [7:0]cmos_data_r;
reg cmos_href_r;
reg cmos_vsync_r;

always@(posedge cmos_pclk_i)
begin
    cmos_data_r <= cmos_data_i;
    cmos_href_r <= cmos_href_i;
    cmos_vsync_r<= cmos_vsync_i;
end
//assign rgb_o = 24'b1111111_00000000_1111111;
cmos_decode cmos_decode_u0(
    //system signal.
    .cmos_clk_i(CLK_i),//cmos senseor clock.
    .rst_n_i(RESETn_i2c),//system reset.active low.
    //cmos sensor hardware interface.
    .cmos_pclk_i(cmos_pclk_i),//(cmos_pclk),//input pixel clock.
    .cmos_href_i(cmos_href_r),//(cmos_href),//input pixel hs signal.
    .cmos_vsync_i(cmos_vsync_r),//(cmos_vsync),//input pixel vs signal.
    .cmos_data_i(cmos_data_r),//(cmos_data),//data.
    .cmos_xclk_o(cmos_xclk_o),//(cmos_xclk),//output clock to cmos sensor.
    //user interface.
    .hs_o(hs_o),//hs signal.
    .vs_o(vs_o),//vs signal.
    // .de_o(de_o),//data enable.
    .rgb565_o(rgb_o_r),//data output
    .vid_clk_ce(vid_clk_ce)
);

count_reset_v1#(
    .num(20'hffff0)
)()
```

```
.clk_i(CLK_i),  
.rst_o(RESETn_i2c)  
);  
  
endmodule
```

1 cmos_decode_v1.v 是本模块的关键部分,实现了RGB565 的解码输出以及 vid_clk_ce 实现了此模块和 vid_in IP 直接时序匹配的关系。

表 3-3-1-2 OV_Sensor_ML 源码 cmos_decode_v1.v

```
'timescale 1ns / 1ps  
//////////////////////////////  
// Company: milinker corporation  
// WEB:www.milinker.com  
// BBS:www.osrc.cn  
// Engineer..  
// Create Date: 07:28:50 09/04/2015  
// Design Name: cmos_decode_v1  
// Module Name: cmos_decode_v1  
// Project Name: cmos_decode_v1  
// Target Devices: XC6SLX25-FTG256 Mis603  
// Tool versions: ISE14.7  
// Description: cmos_decode_v1.  
// Revision: V1.0  
// Additional Comments:  
//1) _i PIN input  
//2) _o PIN output  
//3) _n PIN active low  
//4) _dg debug signal  
//5) _r reg delay  
//6) _s state machine  
/////////////////////////////  
module cmos_decode(  
    //system signal.  
    input cmos_clk_i,//cmos senseor clock.  
    input rst_n_i,//system reset.active low.  
    //cmos sensor hardware interface.  
    input cmos_pclk_i,//input pixel clock.  
    input cmos_href_i,//input pixel hs signal.  
    input cmos_vsync_i,//input pixel vs signal.  
    input[7:0]cmos_data_i,//data.  
    output cmos_xclk_o,//output clock to cmos sensor.  
    //user interface.
```

```
output hs_o,//hs signal.  
output vs_o,//vs signal.  
output reg [15:0] rgb565_o,//data output  
output vid_clk_ce  
);  
parameter[5:0]CMOS_FRAME_WAITCNT = 4'd15;  
  
reg[4:0] rst_n_reg = 5'd0;  
//reset signal deal with.  
always@(posedge cmos_clk_i)  
begin  
    rst_n_reg <= {rst_n_reg[3:0],rst_n_i};  
end  
  
reg[1:0]vsync_d;  
reg[1:0]href_d;  
wire vsync_start;  
wire vsync_end;  
//vs signal deal with.  
always@(posedge cmos_pclk_i)  
begin  
    vsync_d <= {vsync_d[0],cmos_vsync_i};  
    href_d <= {href_d[0],cmos_href_i};  
end  
  
assign vsync_start = vsync_d[1]&(!vsync_d[0]);  
assign vsync_end = (!vsync_d[1])&vsync_d[0];  
  
reg[6:0]cmos_fps;  
//frame count.  
always@(posedge cmos_pclk_i)  
begin  
    if(!rst_n_reg[4])  
        begin  
            cmos_fps <= 7'd0;  
        end  
    else if(vsync_start)  
        begin  
            cmos_fps <= cmos_fps + 7'd1;  
        end  
    else if(cmos_fps >= CMOS_FRAME_WAITCNT)  
        begin  
            cmos_fps <= CMOS_FRAME_WAITCNT;
```

```
        end
    end
//wait frames and output enable.
reg out_en;
always@(posedge cmos_pclk_i)
begin
if(!rst_n_reg[4])
begin
    out_en <= 1'b0;
end
else if(cmos_fps >= CMOS_FRAME_WAITCNT)
begin
    out_en <= 1'b1;
end
else
begin
    out_en <= out_en;
end
end

//output data 8bit changed into 16bit in rgb565.
reg [7:0] cmos_data_d0;
reg [15:0]cmos_rgb565_d0;
reg byte_flag;

always@(posedge cmos_pclk_i)
begin
if(!rst_n_reg[4])
    byte_flag <= 0;
else if(cmos_href_i)
    byte_flag <= ~byte_flag;
else
    byte_flag <= 0;
end

reg byte_flag_r0;
always@(posedge cmos_pclk_i)
begin
if(!rst_n_reg[4])
    byte_flag_r0 <= 0;
else
    byte_flag_r0 <= byte_flag;
end
```

```

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        cmos_data_d0 <= 8'd0;
    else if(cmos_href_i)
        cmos_data_d0 <= cmos_data_i; //MSB -> LSB
    else if(~cmos_href_i)
        cmos_data_d0 <= 8'd0;
end

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        rgb565_o <= 16'd0;
    else if(cmox_href_i&byte_flag)
        rgb565_o <= {cmox_data_d0,cmox_data_i}; //MSB -> LSB
    else if(~cmox_href_i)
        rgb565_o <= 8'd0;
end

assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
assign vs_o = out_en ? vsync_d[1] : 1'b0;
assign hs_o = out_en ? href_d[1] : 1'b0;

assign cmos_xclk_o = cmos_clk_i;

endmodule

```

count_reset_v1.v 源文件实现了信号的延迟复位。

表 3-3-1-1 count_reset_v1.v

```

`timescale 1ns / 1ps
///////////////////////////////
// Company: milinker corporation
// WEB:www.milinker.com
// BBS:www.osrc.cn
// Engineer:sanliuyaoling.
// Create Date: 07:28:50 12/04/2015
// Design Name: count_reset_v1
// Module Name: count_reset_v1
// Project Name: count_reset_v1
// Target Devices: XC7Z020-CLG484-1I

```

```
// Tool versions: vivado2015.4
// Description:      count_reset_v1
// Revision:        V1.0
// Additional Comments:
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low
//4) _dg debug signal
//5) _r  reg delay
//6) _s state machine
///////////////////////////////
module count_reset_v1#
(
    parameter[19:0]num = 20'hffff0
)(
    input clk_i,
    output rst_o
);
reg[19:0] cnt = 20'd0;
reg rst_d0;

/*count for clock*/
always@(posedge clk_i)
begin
    cnt <= ( cnt <= num)?( cnt + 20'd1 ):num;
end

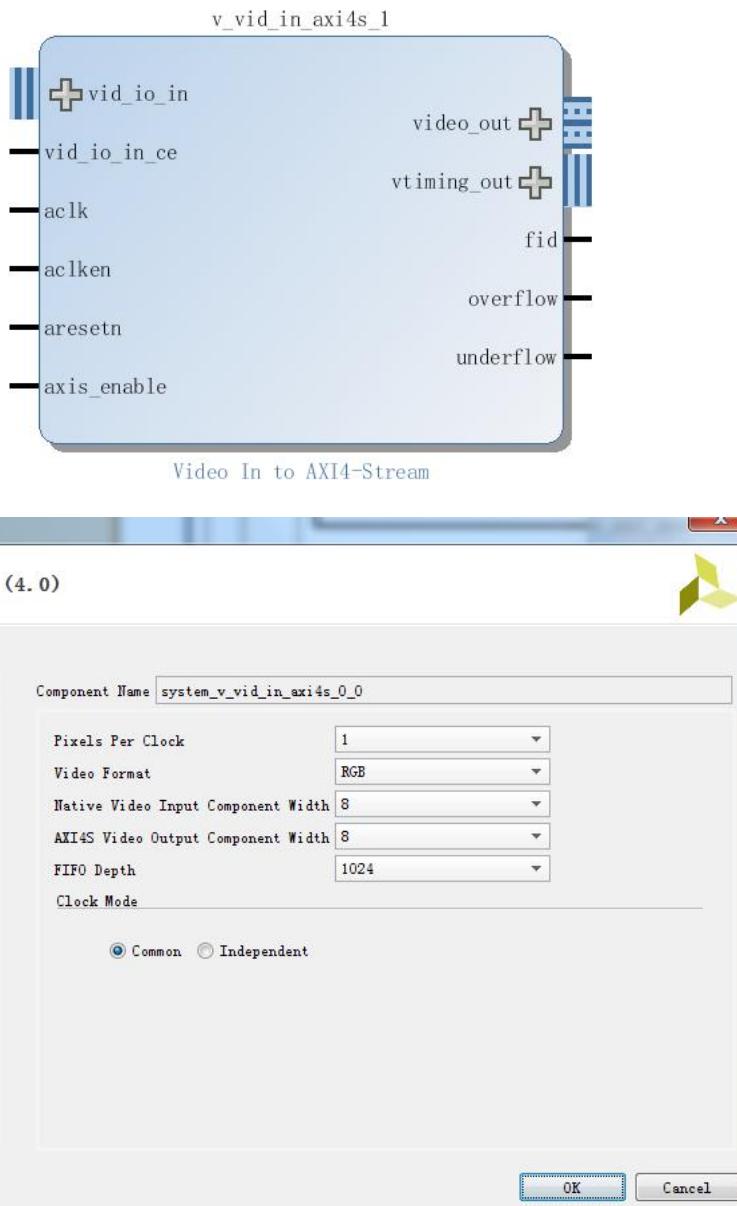
/*generate output signal*/
always@(posedge clk_i)
begin
    rst_d0 <= ( cnt >= num)?1'b1:1'b0;
end

assign rst_o = rst_d0;

endmodule
```

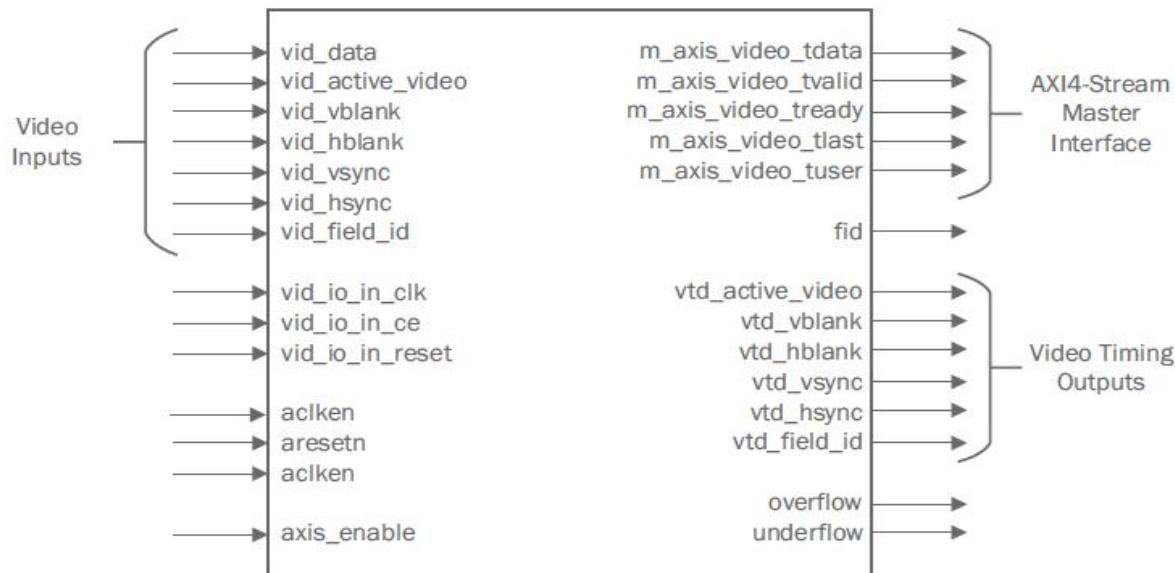
表 3-3-1-2

5.3.2 vid in IP 模块



- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Video Format: 视频格式
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- FIFO Depth: FIFO 深度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟

5.3.2 VID_IN IP 接口信号的定义



Common Interface

Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
axis_enable	Input	1	This input should be connected to the VTC detector locked status and is synchronous to vid_io_in_clk. 1 = Enable writes into FIFO 0 = Disable writes into FIFO
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_in_clk	Input	1	Native video clock. Only available in independent clock mode.
vid_io_in_ce	Input	1	Native video clock enable
vid_io_in_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk. If an overflow occurs, this could indicate that the connected AXI4-Stream Slave is creating excessive back-pressure.
underflow	Output	1	Flag indicating that the FIFO has under-flowed. This should never occur under normal operation. Synchronous to aclk.

Video Timing Interface

Signal Name	Direction	Width	Description
vtd_vsync	Out	1	Vertical sync video timing signal.
vtd_hsync	Out	1	Horizontal sync video timing signal.
vtd_vblank	Out	1	Vertical blank video timing signal.
vtd_hblank	Out	1	Horizontal blank video timing signal.
vtd_active_video	Out	1	Active video flag. 1 = active video, 0 = blanked video
vtd_field_id	Out	1	VTC field ID. 0= even field, 1= odd field.

Video Input Interface

Signal Name	Direction	Width	Description
vid_active_video	In	1	Video data valid. 1 = active video, 0 = blanked video
vid_vsync	In	1	Vertical sync video timing signal. Active High
vid_hsync	In	1	Horizontal sync video timing signal. Active High
vid_vblank	In	1	Vertical blank video timing signal. Active High
vid_hblank	In	1	Horizontal blank video timing signal. Active High
vid_data	In	8-256	Parallel video input data.
vid_field_id	In	1	Video field. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

使用到的信号有：

Vid_in IP 输入端信号：

Vid_data: 视频数据输入

Vid_active_video: 视频数据有效

Vid_hsync: 视频行同步信号 (非常关键信号, 下面重点分析对象)

Vid_vsync: 视频场同步信号

Vid_io_in_ce: 数据输入有效 (非常关键信号, 下面重点分析对象)

vid_io_in_clk: 这是时钟信号和摄像头时钟同步

Vid_io_in_reset: 这个信号, 高电平的时候复位

有很多读者会问笔者, 这些官方的 IP 如何使用, 这么没有详细的技术手册。还别说, 官方就是没有非常详细的技术手册, 有时候笔者也是使出浑身解数分析 vid_in IP 内部信号时序, 掌握 OV_Sensor_ML 自定义 IP 时序接口设计。

打开 v_vid_in_axi4s_v4_0_1_formatter.v 这个文件

下面对其关键的部分进行说明。

表 3-3-2-1 v_vid_in_axi4s_v4_0_1_formatter.v

```
'timescale 1ps/1ps
`default_nettype none
(* DowngradeIPIdentifiedWarnings="yes" *)
```

```
module v_vid_in_axi4s_v4_0_1_formatter #(
    parameter C_NATIVE_DATA_WIDTH = 24
) (
    // System signals
    input wire VID_IN_CLK,           // Native video clock
    input wire VID_RESET,            // Native video reset
    input wire VID_CE,               // Native video clock enable

    // Video input signals
    input wire VID_ACTIVE_VIDEO,     // Native video input data enable
    input wire VID_VBLANK,           // Native video input vertical blank
    input wire VID_HBLANK,           // Native video input horizontal blank
    input wire VID_VSYNC,             // Native video input vertical sync
    input wire VID_HSYNC,             // Native video input horizontal sync
    input wire VID_FIELD_ID,         // Native video input field-id
    input wire [C_NATIVE_DATA_WIDTH-1:0] VID_DATA, // Native video input data

    // Video timing detector signals
    output wire VTD_ACTIVE_VIDEO,    // Native video output data enable
    output wire VTD_VBLANK,           // Native video output vertical blank
    output wire VTD_HBLANK,           // Native video output horizontal blank
    output wire VTD_VSYNC,             // Native video output vertical sync
    output wire VTD_HSYNC,             // Native video output horizontal sync
    output wire VTD_FIELD_ID,         // Native video output field-id
    input wire VTD_LOCKED,            // Native video locked signal from VTD

    // FIFO write signals
    output wire [C_NATIVE_DATA_WIDTH+2:0] FIFO_WR_DATA, // FIFO write data
    output wire FIFO_WR_EN           // FIFO write enable
);

    // Wire and register declarations
    reg de_1 = 0;
    reg vblank_1 = 0;
    reg hblank_1 = 0;
    reg vsync_1 = 0;
    reg hsync_1 = 0;
    reg [C_NATIVE_DATA_WIDTH -1:0] data_1 = 0;
    reg de_2 = 0;
    reg v_blank_sync_2 = 0;
    reg [C_NATIVE_DATA_WIDTH -1:0] data_2 = 0;
    reg de_3 = 0; // DE output register
```

```
reg [C_NATIVE_DATA_WIDTH-1:0] data_3 = 0; // data output register
reg vert_blankning_intvl = 0; // SR, reset by DE rising
reg field_id_1 = 0;
reg field_id_2 = 0;
reg field_id_3 = 0;

wire v_blank_sync_1; // vblank or vsync
wire de_rising;
wire de_falling;
wire vsync_rising;
reg sof;
reg sof_1;
reg eol;
reg vtd_locked;
wire sof_rising;

// Assignments
assign FIFO_WR_DATA      = {field_id_3,sof_1,eol,data_3};
assign FIFO_WR_EN         = de_3 & ~VID_RESET & vtd_locked;
assign VTD_ACTIVE_VIDEO = de_1;
assign VTD_VBLANK        = vblank_1;
assign VTD_HBLANK        = hblank_1;
assign VTD_VSYNC          = vsync_1;
assign VTD_HSYNC          = hsync_1;
assign VTD_FIELD_ID      = field_id_1;

assign v_blank_sync_1 = vblank_1 || vsync_1;
assign de_rising   = de_1 && !de_2;
assign de_falling  = !de_1 && de_2;
assign vsync_rising = v_blank_sync_1 && !v_blank_sync_2;
assign sof_rising  = sof & ~sof_1;

// VTD locked process
always @(posedge VID_IN_CLK) begin
    if(VID_RESET | ~VTD_LOCKED) begin
        vtd_locked <= 1'b0;
    end else if(VID_CE) begin
        vtd_locked <= (sof_rising & VTD_LOCKED) ? 1'b1 : vtd_locked;
    end
end

// input, output, and delay registers
always @ (posedge VID_IN_CLK) begin
```

```

if(VID_RESET) begin
    de_1          <= 1'b0;
    de_2          <= 1'b0;
    de_3          <= 1'b0;
    vblank_1      <= 1'b0;
    hblank_1      <= 1'b0;
    vsync_1       <= 1'b0;
    hsync_1       <= 1'b0;
    field_id_1    <= 1'b0;
    field_id_2    <= 1'b0;
    field_id_3    <= 1'b0;
    data_1         <= {C_NATIVE_DATA_WIDTH{1'b0}};
    data_2         <= {C_NATIVE_DATA_WIDTH{1'b0}};
    data_3         <= {C_NATIVE_DATA_WIDTH{1'b0}};
    v_blank_sync_2 <= 1'b0;
    eol           <= 1'b0;
    sof            <= 1'b0;
    sof_1          <= 1'b0;
end else if(VID_CE) begin
    de_1          <= VID_ACTIVE_VIDEO;
    de_2          <= de_1;
    de_3          <= de_2;
    vblank_1      <= VID_VBLANK;
    hblank_1      <= VID_HBLANK;
    vsync_1       <= VID_VSYNC;
    hsync_1       <= VID_HSYNC;
    field_id_1    <= VID_FIELD_ID;
    field_id_2    <= field_id_1;
    field_id_3    <= field_id_2;
    data_1         <= VID_DATA;
    data_2         <= data_1;
    data_3         <= data_2;
    v_blank_sync_2 <= v_blank_sync_1;
    eol           <= de_falling;
    sof            <= de_rising && vert_blinking_intvl;
    sof_1          <= sof;
end
end

// Vertical back porch SR register
always @ (posedge VID_IN_CLK) begin
    if(VID_CE) begin
        if(vsync_rising) // falling edge of vsync

```

```

    vert_blinking_intvl <= 1;
else if (de_rising)          // rising edge of data enable
    vert_blinking_intvl <= 0;
end
end

endmodule

```

在上面代码中，

```

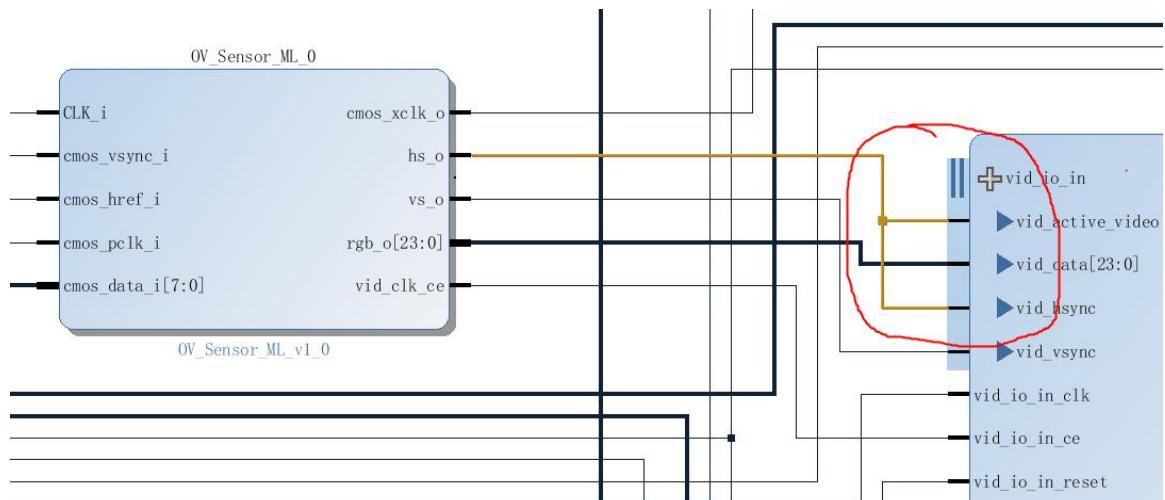
eol      <= de_falling;
sof      <= de_rising && vert_blinking_intvl;

```

eol 实际就是 tlast 信号，而 sof 就是 tuser 信号。tlast 信号代表每行图像数据的最后一个数据，tuser 代表每场数据的第一个数据。

所有非常关键的信号都和 de_falling 和 vert_blinking_intvl 有关系。

hs_o 和 vid_in IP 的连接关系。



上图中，被红色圈起来的 hs_o 信号，同时接到了 vid_in ip 的 vid_active_video 和 vid_hsync 信号接口。因此，de 信号就是 hs_o 信号，而 vid_hsync 我们发现没有任何作用，也就是说不 hs_o 不连接到 vid_hsync 也不影响这里的程序工作。

VID_CE 这个参数就是前面的 vid_io_in_ce 信号，可以看出这个芯片有效的时候相对应的时序电路才会执行。在本工程中，摄像头每 2 个 pclk 输出 1 个有效的数据，而 vid_in IP 如果 VID_CE 为 1 则数据输入会每个时钟输入 1 个就错了。因此官方的 IP 设计的还是很不错考虑周到，通过 VID_CE 这个条件，控制时钟同步。

```

...
else if(VID_CE) begin
...
end
...

```

现在回到 OV_Sensor_ML 的 cmos_decode_v1.v 文件中有一段红色的代码如下：

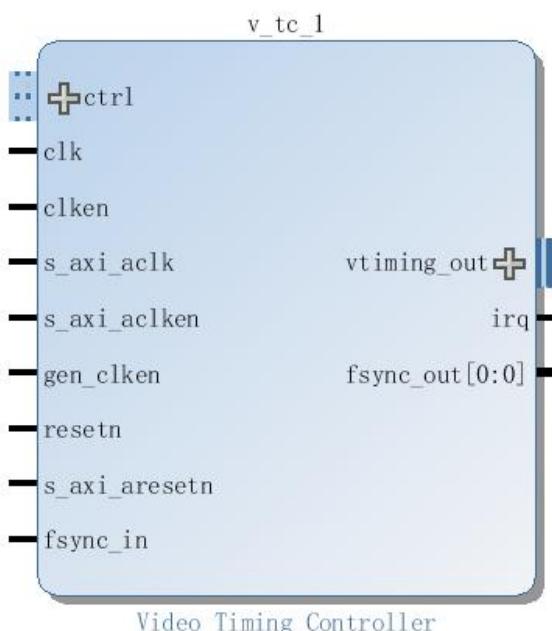
```
assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
```

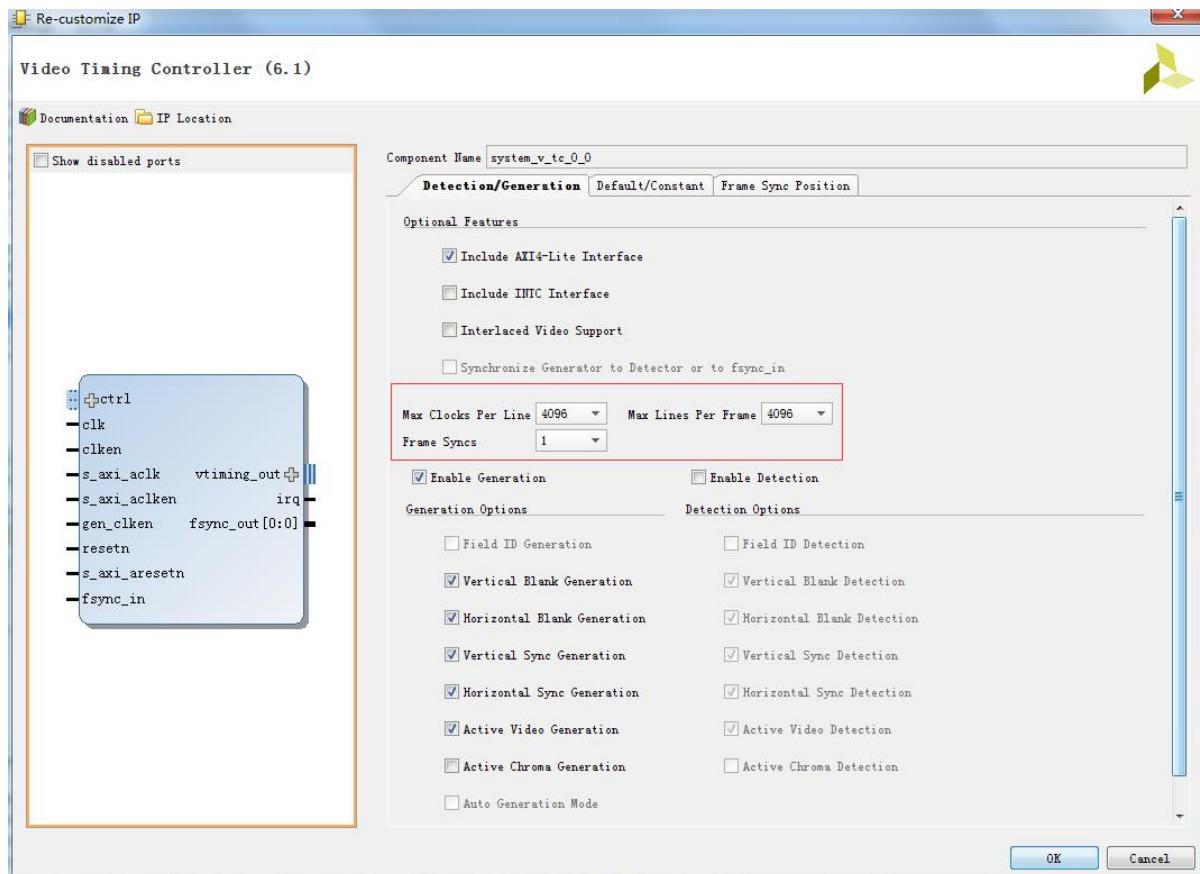
这段代码控制了 vid_clk_ce 的正确输出，关键部分是 `(byte_flag_r0&hs_o)||(!hs_o)`。当 hs_o 有效的时候，vid_in 的 VID_CE 信号就有效，当 hs_o=0 的时候 VID_CE 必须仍然有效，这样才能检测到 vsync_rising 信号了，检测到了 vsync_rising 才能有 `ert_blankning_intvl` 为 1，才有 tuser 信号。

好了罗嗦了半天，终于解释完了，如果有不清楚的，找我们技术支持吧。

5.4 VTC IP 的分析

5.4.1 VTC IP 的参数介绍





这个 IP 就是一个时序发生器，产生显示器输出所需要的时序信号。

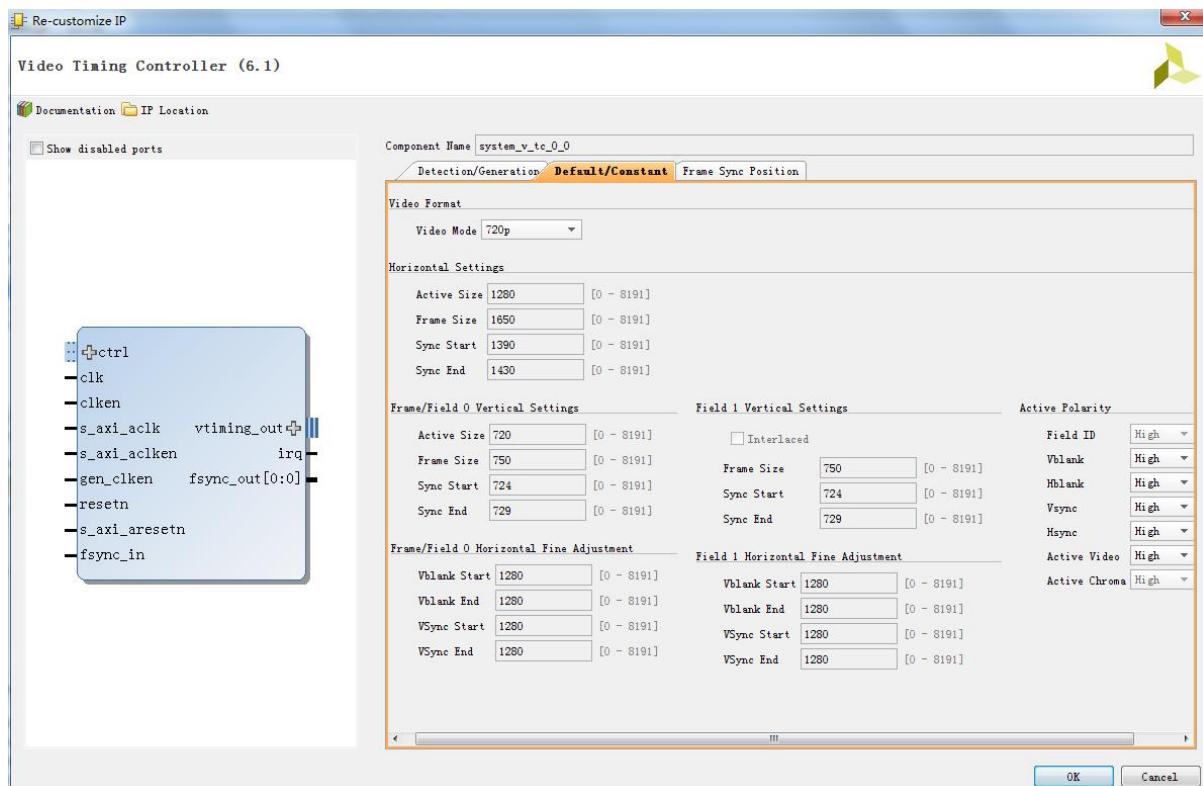
这个页面中，include AXI-lite interface 可以不勾选，不勾选就只能采用默认设置，无法在 C 语言中灵活配置了，所以笔者这里建议大家勾选吧。max clocks per line 和 max_lines per frame 需要设置下，当设置到 4096 的时候可以支持分辨率到最大，当然消耗的资源也更多。笔者这里太奢侈了设置了 4096。实际上设置到 2048 就够用了。本页面的其他信号可以采取默认设置。

Enable Generation:

支持产生时序，这个肯定是必须勾选的。

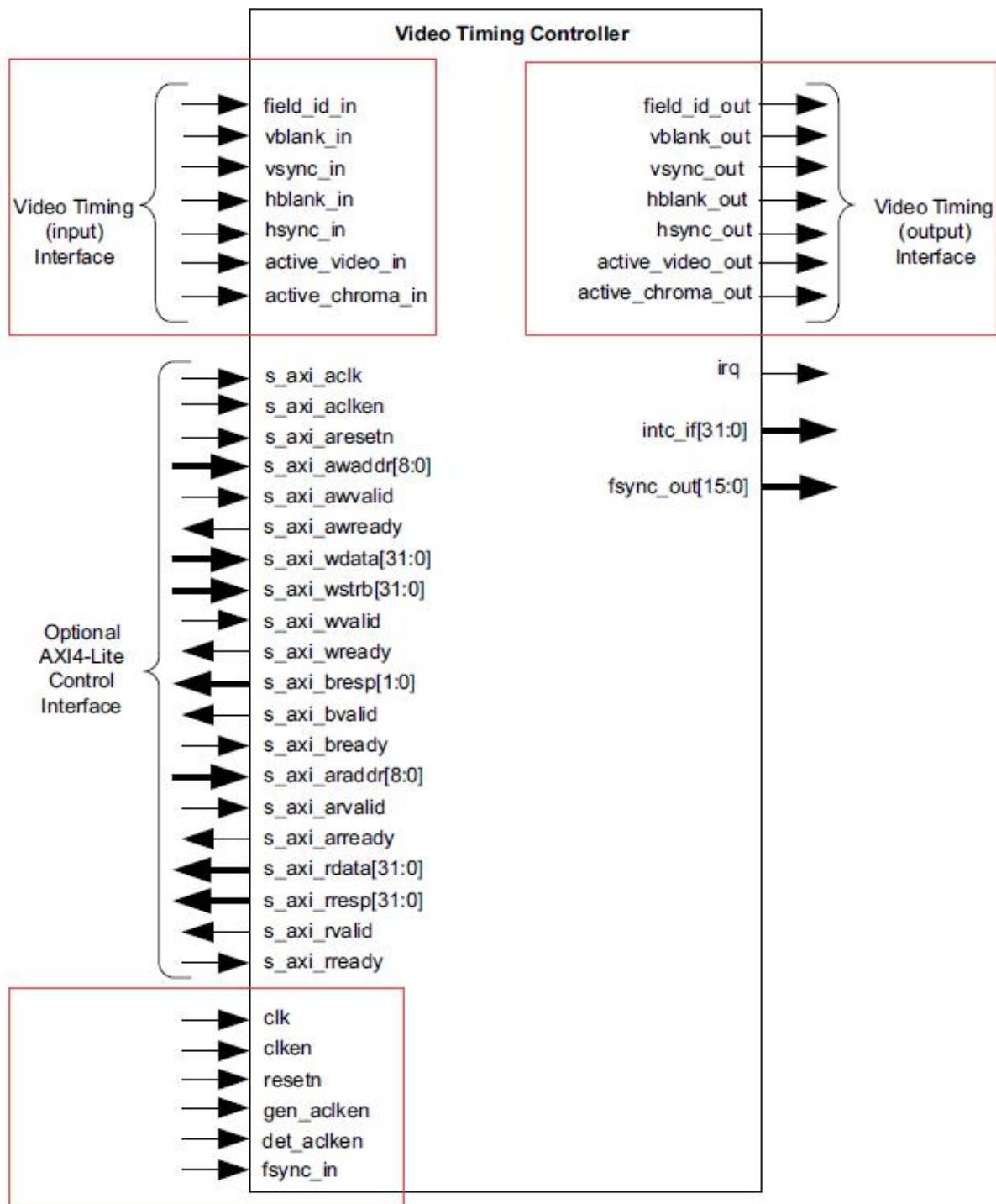
Enable Detection:

支持时序捕捉，这个不是必须的，根据需要而定，如果设置了这个选项，就可以先捕捉输入的时序，然后再设置输出的时序，实现输入和输出一致的效果。



在这个页面中，只要选择需要支持的分辨率就可以了，当然不设置也没关系的，因为我们在 C 代码综合那个会进一步设置的。

5.4.2 VTC IP 接口信号的定义



红色方框内的绝大部分信号需要我们手动联系，所以下面重点是讲解红色方框内的信号作用，至于 AXI4-LITE 接口主要是用来设置参数的。

Common Port Descriptions:

Name	Direction	Width	Description
clk	In	1	Video Core Clock
clken	In	1	Video Core Active High Clock Enable
det_clken	In	1	Video Timing Detection Core Active High Clock Enable
gen_clken	In	1	Video Timing Generator Core Active High Clock Enable
resetn	In	1	Video Core Active Low Synchronous Reset
irq	Output	1	Interrupt request output, active high edge
intc_if	Output	32	OPTIONAL EXTERNAL INTERRUPT CONTROLLER INTERFACE Available when the "Include INTC Interface" or C_HAS_INTC_IF has been selected. Bits [31:8] are the same as the bits [31:8] in the status register (0x0004). Bits [5:0] are the same as bits [21:16] of the error register (0x0008). Bits [7:6] are reserved and are always 0.

Detector Interface (Video Timing Input Interface)			
field_id_in	Input	1	INPUT FIELD ID Used to set the field_id polarity in the Detector Polarity Register (Address Offset 0x002C). Optional. Only valid when interlace support and field id are enabled.
hsync_in	Input	1	INPUT HORIZONTAL SYNCHRONIZATION Used to set the DETECTOR HSYNC register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hsync_in input is not connected, then the "Horizontal Sync Detection" option must be deselected.
hblank_in	Input	1	INPUT HORIZONTAL BLANK Used to set the DETECTOR HSIZE register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hblank_in input is not connected, then the "Horizontal Blank Detection" option must be deselected.
vsync_in	Input	1	INPUT VERTICAL SYNCHRONIZATION Used to set the DETECTOR F0_VSYNC_V and the F0_VSYNC_H registers. Polarity is auto-detected. Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vsync_in input is not connected, then the "Vertical Sync Detection" option must be deselected.

Name	Direction	Width	Description
vblank_in	Input	1	<p>INPUT VERTICAL BLANK Used to set the DETECTOR_VSIZE and the F0_VBLANK_H registers. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vblank_in input is not connected, then the "Vertical Blank Detection" option must be deselected.</p>
active_video_in	Input	1	<p>INPUT ACTIVE VIDEO Used to set the DETECTOR_ACTIVE_SIZE register. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the active_video_in input is not connected, then the "Active Video Detection" option must be deselected.</p>
active_chroma_in	Input	1	<p>INPUT ACTIVE CHROMA Used to set the VIDEO_FORMAT and the CHROMA_PARITY bits in the Detector Encoding Register. Polarity is auto-detected.</p> <p>Optional. If the active_chroma_in input is not connected, then the "Active Chroma Detection" option must be deselected.</p>

Generator Interface (Video Timing Output Interface)			
field_id_out	Output	1	<p>OUTPUT FIELD ID Generated field id signal. Polarity configured by the Generator Polarity Register (Address Offset 0x006C) Optional. Only enabled when interlaced support and field id generation is enabled.</p>
hsync_out	Output	1	<p>OUTPUT HORIZONTAL SYNCHRONIZATION Generated horizontal synchronization signal. Polarity configured by the control register. Asserted active during the cycle set by the HSYNC_START bits and deasserted during the cycle set by the HSYNC_END bits in the GENERATOR HSYNC register.</p>
hblank_out	Output	1	<p>OUTPUT HORIZONTAL BLANK Generated horizontal blank signal. Polarity configured by the control register. Asserted active during the cycle set by ACTIVE_HSIZEx and deasserted during the cycle set by the FRAME_HSIZEx bits in the GENERATOR HSIZEx register.</p>
vsync_out	Output	1	<p>OUTPUT VERTICAL SYNCHRONIZATION Generated vertical synchronization signal. Polarity configured by the control register. Asserted active during the line set by the F#_VSYNC_VSTART bits and deasserted during the line set by the F#_VSYNC_VEND bits in the GENERATOR F#_VSYNC_V registers.</p>

Name	Direction	Width	Description
vblank_out	Output	1	OUTPUT VERTICAL BLANK Generated vertical blank signal. Polarity configured by the control register. Asserted active during the line set by the ACTIVE_VSIZE bits and deasserted during the line set by the GENERATOR VSIZE register.
active_video_out	Output	1	OUTPUT ACTIVE VIDEO Generated active video signal. Polarity configured by the control register. Active for non blanking lines. Asserted active during the first cycle of the field/frame and deasserted during the cycle set by the GENERATOR ACTIVE_SIZE register
active_chroma_out	Output	1	OUTPUT ACTIVE CHROMA Generated active chroma signal. Denotes which lines contain valid chroma samples (used for YUV 4:2:0). Polarity configured by the GENERATOR POLARITY register. Active for non-blanking lines configured by the VIDEO_FORMAT and the CHROMA_PARITY bits in the GENERATOR Encoding Register.
Frame Synchronization Interface			
fsync_out	Output	[Frame Syncs - 1:0]	FRAME SYNCHRONIZATION OUTPUT Each Frame Synchronization bit toggles for only one clock cycle during each frame. The number of bits is configured with the Frame Syncs GUI parameter. Each bit is independently configured for horizontal and vertical clock cycle position with the Frame Sync 0-15 Config registers).
fsync_in	Input	1	FRAME SYNCHRONIZATION INPUT This is a one clock cycle pulse (active high) input. The video timing generator will be synchronized to the input if used.

本例子中没有使用到输入时序的捕捉，因此笔者下面只对用到的信号做一些介绍。

hsync_out:

产生行同步输出

hsync_out:

产生行消影

vsync_out:

产生场同步输出

vblank_out:

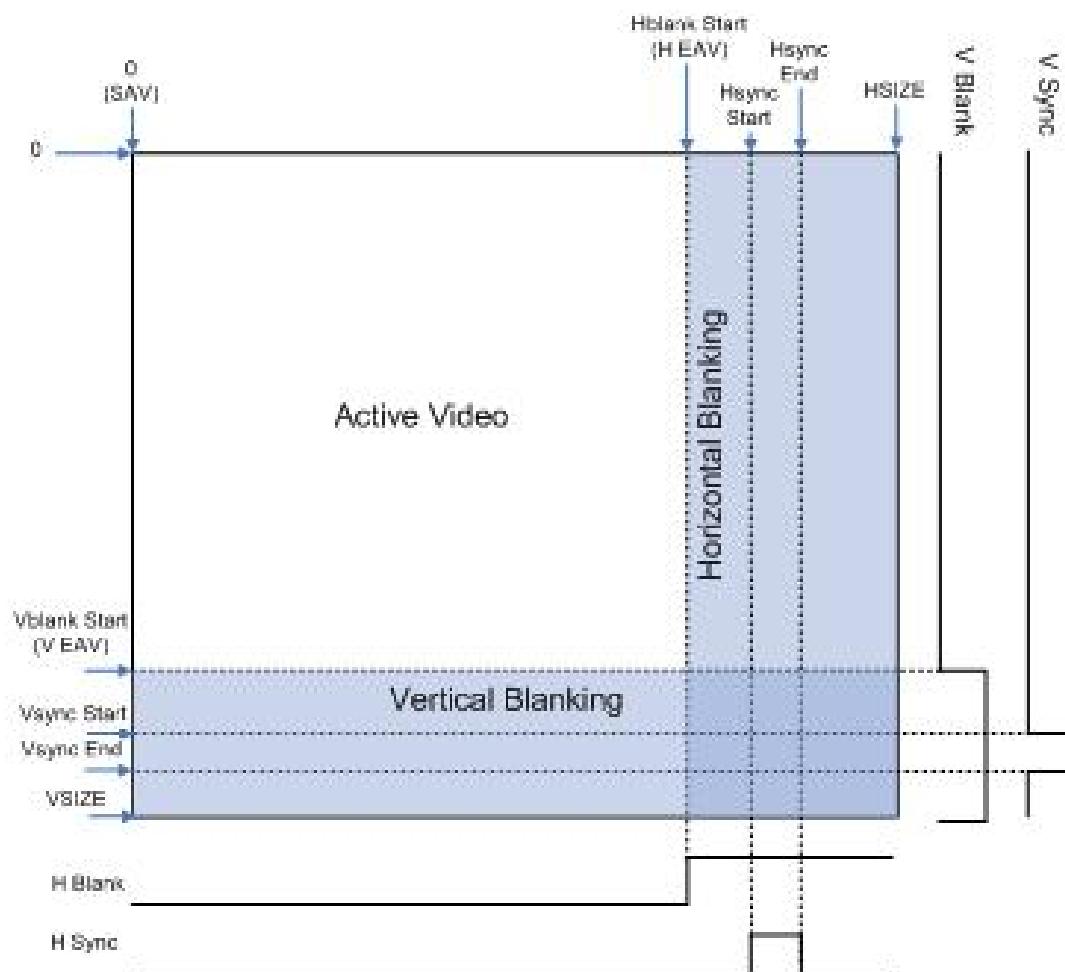
产生场消影

active_video_out:

有效数据输出

5.4.3 VTC IP 配置寄存器

shows the start of the horizontal front porch (Hblank Start), synchronization (Hsync Start), back porch (Hsync End) and active video (SAV). It also shows the start of the vertical front porch (Vblank Start), synchronization (Vsync Start), back porch (Vsync End) and active video (SAV). The total number of horizontal clock cycles is HSIZE and the total number of lines is the VSIZE.



Generator Active Size Register (Address Offset 0x0060)

0x0060	GENERATOR ACTIVE_SIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
ACTIVE_VSIZE	28:16	Generated Vertical Active Frame Size. The height of the frame without blanking in number of lines.
RESERVED	15:13	Reserved
ACTIVE_HSIZE	12:0	Generated Horizontal Active Frame Size. The width of the frame without blanking in number of pixels/clocks.

这是重要的寄存器用来设置有效的行数量和场数量

Generator Timing Status Register (Address Offset 0x0064)

0x0064	GENERATOR TIMING_STATUS	Read
Name	B its	Description
RESERVED	31:3	Reserved
GEN_ACTIVE_VIDEO	2	Generated Active Video Interrupt Status. Set high during the first cycle the output active video is asserted.
GEN_VBLANK	1	Generated Vertical Blank Interrupt Status. Set high during the first cycle the output vertical blank is asserted.
RESERVED	0	Reserved

GEN_ACTIVE_VIDEO:当第一帧图像输出时候置 1

GEN_VBLANK:第一帧有效图像的 blank 信号输出的时候置 1

Generator Encoding Register (Address Offset 0x0068)

0x0068	GENERATOR ENCODING	Read/Write
Name	B its	Description
RESERVED	31:10	Reserved
CHROMA_PARITY	9:8	Generated Chroma Parity 0: Chroma Active during even active-video lines of frame. Active every pixel of active line 1: Chroma Active during odd active-video lines of frame. Active every pixel of active line 2: Chroma Active during even active video lines of frame. Active every even pixel of active line, inactive every odd pixel 3: Chroma Active during odd active video lines of frame. Active every even pixel of active line, inactive every odd pixel
FIELD_ID_PARITY	7	Generated Field ID Parity 0: Field ID input is currently low 1: Field ID input is currently high
INTERLACED	6	Generated Progressive/Interlaced 0: Generated video format is progressive 1: Generated video format is interlaced
RESERVED	5:4	Reserved
VIDEO_FORMAT	3:0	Generated Video Format Denotes when the active_chroma signal is active. 0: YUV 4:2:2 - Active_chroma is active during the same time active_video is active. 1: YUV 4:4:4 - Active_chroma is active during the same time active_video is active. 2: RGB - Active_chroma is active during the same time active_video is active. 3: YUV 4:2:0- Active_chroma is active every other line during the same time active_video is active. See The CHROMA_PARITY bits to control which lines and pixels.

CHROMA_PARITY: 奇偶色度 (读者没明白)

FIELD_ID_PARITY: 奇偶场标志

INTERLACED: 视频格式是渐进式还是各行扫描

VIDEO_FORMAT: 视频格设置, 有 YUV422 YUV444 YUV420 RGB

Generator Polarity Register (Address Offset 0x006C)

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
RESERVED	31:7	Reserved
FIELD_ID_POL	6	Generated Field ID Polarity 0: Low during Field 0 and High during Field 1 1: High during Field 0 and Low during Field 1
ACTIVE_CHROMA_POL	5	Generated Active Chroma Polarity 0: Active Low Polarity 1: Active High Polarity

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
ACTIVE_VIDEO_POL	4	Generated Active Video Polarity 0: Active Low Polarity 1: Active High Polarity
HSYNC_POL	3	Generated Horizontal Sync Polarity 0: Active Low Polarity 1: Active High Polarity
VSYNC_POL	2	Generated Vertical Sync Polarity 0: Active Low Polarity 1: Active High Polarity
HBLANK_POL	1	Generated Horizontal Blank Polarity 0: Active Low Polarity 1: Active High Polarity
VBLANK_POL	0	Generated Vertical Blank Polarity 0: Active Low Polarity 1: Active High Polarity

这个寄存器设置相应的场输出极性和色度输出极性。

Generator Horizontal Frame Size Register (Address Offset 0x0070)

0x0070	GENERATOR HSIZE	Read/Write
Name	B its	Description
RESERVED	31:13	Reserved
FRAME_HSIZE	12:0	Generated Horizontal Frame Size. The width of the frame with blanking in number of pixels/clocks.

一副图像的一行的大小，包括了消隐和有效数据阶段。

Generator Vertical Frame Size Register (Address Offset 0x0074)

0x0074	GENERATOR VSIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
FIELD1_VSIZE	28:16	Generated Vertical Field 1 Size. The height with blanking in number of lines of field 1.
FRAME_VSIZE	12:0	Generated Vertical Frame Size. The height of the frame with blanking in number of lines.

一副图像的一场的大小，包括了消隐和有效数据阶段。

Generator Horizontal Sync Register (Address Offset 0x0078)

0x0078	GENERATOR HSYNC	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
Hsync_End	28:16	Generated Horizontal Sync End End cycle index of horizontal sync. Denotes the first cycle hsync_in is de-asserted.
RESERVED	15:13	Reserved
Hsync_Start	12:0	Generated Horizontal Sync End Start cycle index of horizontal sync. Denotes the first cycle hsync_in is asserted.

设置行的水平同步结束和同步开始

Generator Frame/Field 0 Vertical Blank Cycle Register (Address Offset 0x007C)

0x007C	GENERATOR F0_VBLANK_H	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VBLANK_HEND	28:16	Generated Vertical Blank Horizontal End End Cycle index of vertical blank. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F0_VBLANK_HSTART	12:0	Generated Vertical Blank Horizontal Start Start Cycle index of vertical blank. Denotes the first cycle vblank_in is asserted.

设置 Fram/Field0 的水平消隐结束和开始

Generator Frame/Field 0 Vertical Sync Line Register (Address Offset 0x0080)

0x0080	GENERATOR F0_VSYNC_V	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VSYNC_VEND	28:16	Generated Vertical Sync Vertical End End Line index of vertical sync. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_VSTART	12:0	Generated Vertical Sync Vertical Start Start line index of vertical sync. Denotes the first line vsync_in is asserted.

设置 Fram/Field0 的垂直同步垂直结束和开始

Generator Frame/Field 0 Vertical Sync Cycle Register (Address Offset 0x0084)

0x0084	GENERATOR F0_VSYNC_H	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
F0_VSYNC_HEND	28:16	Generated Vertical Sync Horizontal End End cycle index of vertical sync. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_HSTART	12:0	Generated Vertical Sync Horizontal Start Start cycle index of vertical sync. Denotes the first cycle vsync_in is asserted.

设置 Fram/Field0 的垂直同步水平结束和开始

Generator Field 1 Vertical Blank Cycle Register (Address Offset 0x0088)

0x0088	GENERATOR F1_VBLANK_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VBLANK_HEND	28:16	Generated Field 1 Vertical Blank Horizontal End End Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F1_VBLANK_HSTART	12:0	Generated Field 1 Vertical Blank Horizontal Start Start Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is asserted.

设置 Field1 的水平消隐结束和开始

Generator Field 1 Vertical Sync Line Register (Address Offset 0x008C)

0x008C	GENERATOR F1_VSYNC_V	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_VEND	28:16	Generated Field 1 Vertical Sync Vertical End End Line index of vertical sync for field 1. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_VSTART	12:0	Generated Field 1 Vertical Sync Vertical Start Start line index of vertical sync for field 1. Denotes the first line vsync_in is asserted.

设置 Field1 的垂直同步垂直结束和开始

Generator Field 1 Vertical Sync Cycle Register (Address Offset 0x0090)

0x0090	GENERATOR F1_VSYNC_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_HEND	28:16	Generated Field 1 Vertical Sync Horizontal End End cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_HSTART	12:0	Generated Field 1 Vertical Sync Horizontal Start Start cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is asserted.

设置 Field1 的垂直同步水平结束和开始

Frame Sync 0 - 15 Configuration Registers (Address Offsets 0x0100 - 0x013C)

0x0100	FRAME SYNC 0 CONFIG	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
V_START	28:16	FRAME SYNCHRONIZATION VERTICAL START Vertical line during which the fsync_out[0] output port is asserted active-high. Note: Frame Syncs are not active during the complete line, only in the cycle during which both the V_START and H_START are valid each frame.
RESERVED	15:13	Reserved
H_START	12:0	FRAME SYNCHRONIZATION HORIZONTAL START Horizontal Cycle during which fsync_out[0] output port is asserted active-high

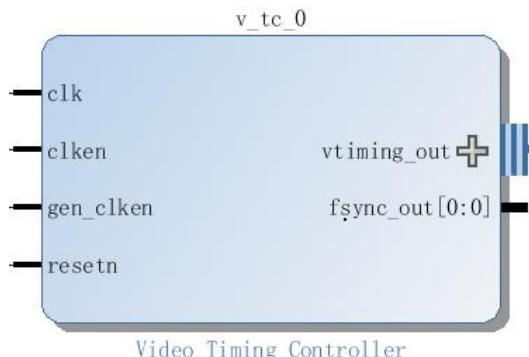
Generator Global Delay Register (Address Offset 0x140)

0x140	Generator Global Delay	Read/Write
Name	Bits	Description
Reserved	31:29	Reserved
V_DELAY	28:16	GENERATOR VERTICAL DELAY Vertical line offset. This is the number of lines that the generated output will be shifted relative to the detector (input timing). The vertical delay is only available when both the detector and generator are enabled. Can be combined with the H_DELAY.

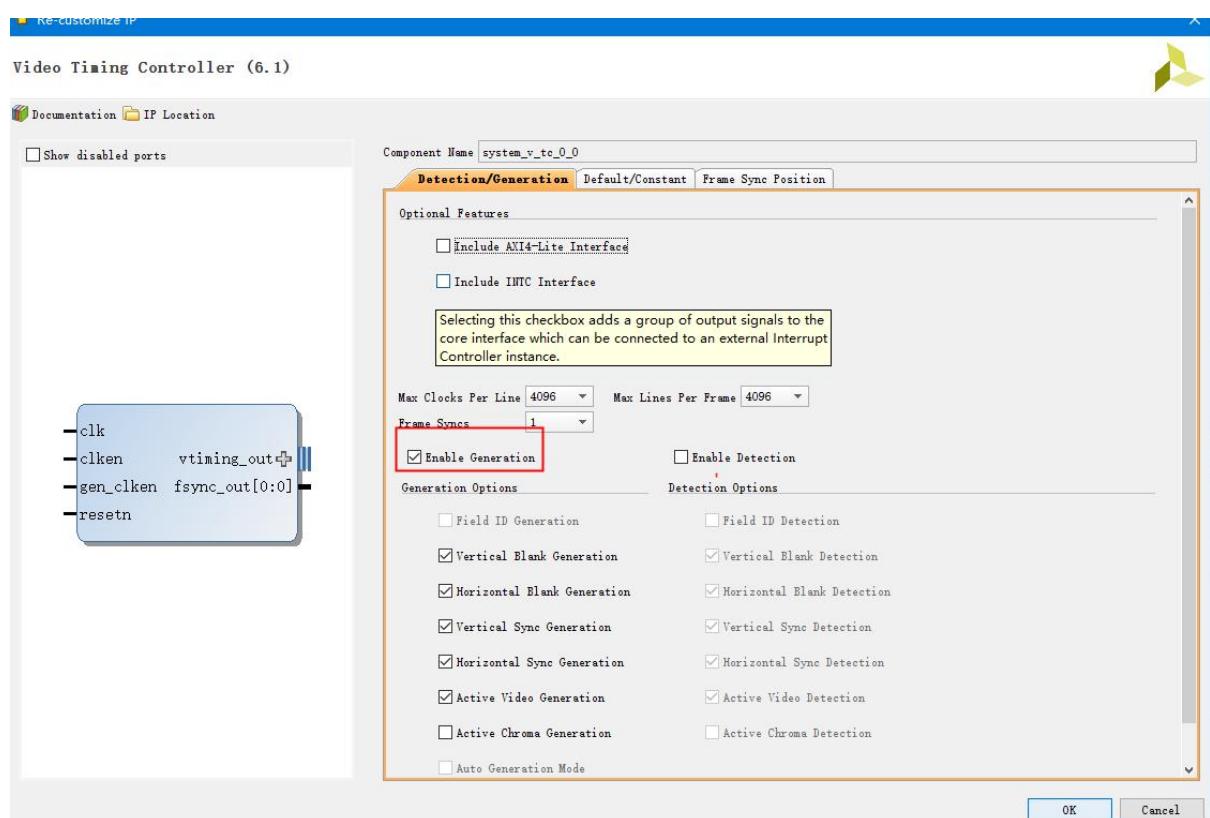
Reserved	15:13	Reserved
H_DELAY	12:0	GENERATOR HORIZONTAL DELAY Horizontal cycle offset. This is the number of clock cycles that the generated output will be shifted relative to the detector (input timing). The horizontal delay is only available when both the detector and generator are enabled. Can be combined with the V_DELAY.

5.4.5 设置 VTC IP

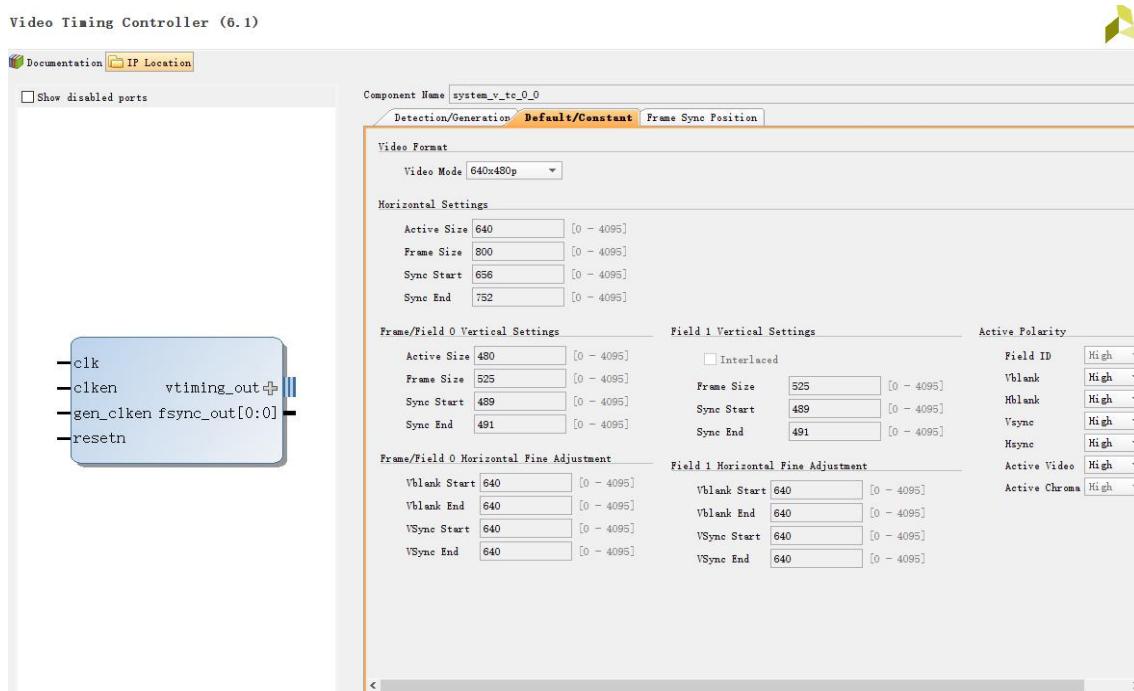
讲了这么多实际上我们用的时候很简单,所以只要这么简单。



由于不使用动态配置，并且只使用了视频时序产生，所以只要勾选如下复选框。



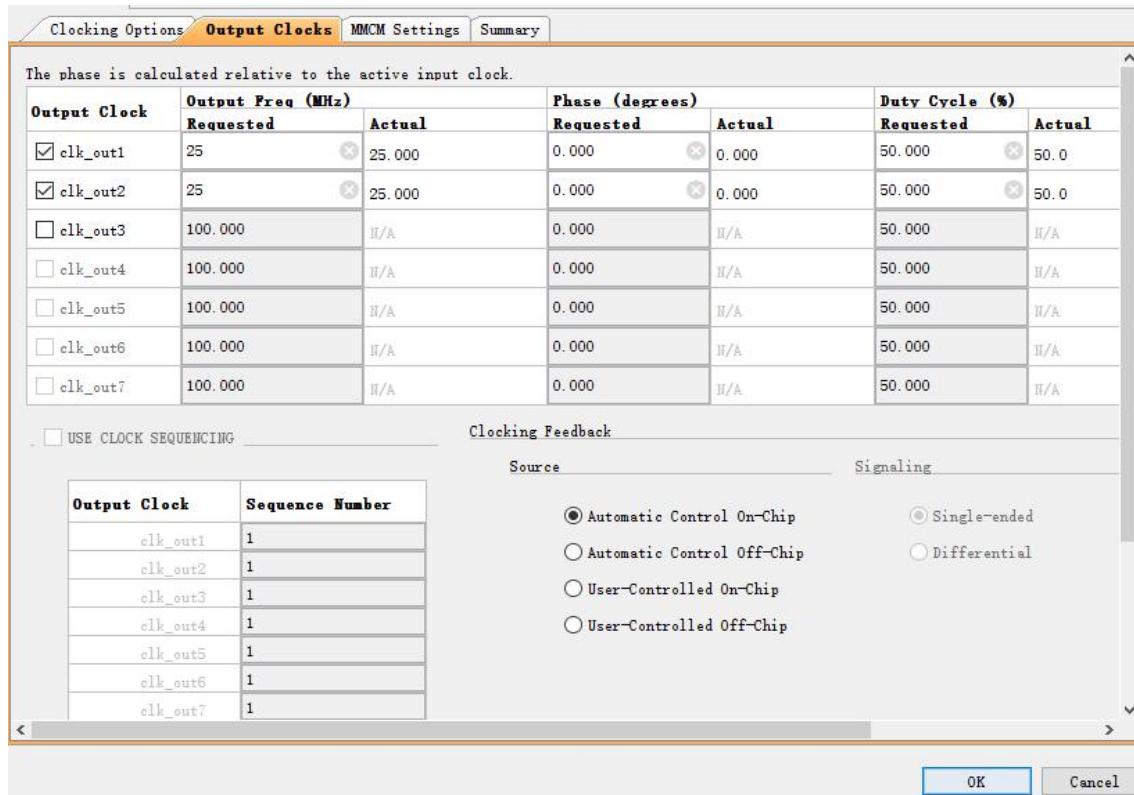
由于 OV7725 分辨率是 640X480 因此直接选择 640PX480 就可以了。



5.6 PLL 时钟设置

由于这里的分辨率是 640X480 因此提供给 VTC IP 和 VID OUTIP 的时钟只要 25M 就可以了

MIZ702/MIZ702N 时钟设置



MIZ701N 时钟设置

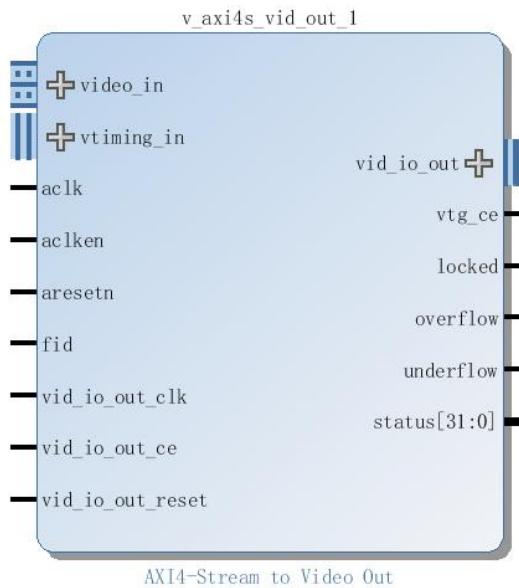
The screenshot shows the 'Output Clocks' tab of the MIZ701N clock configuration interface. It displays the following table:

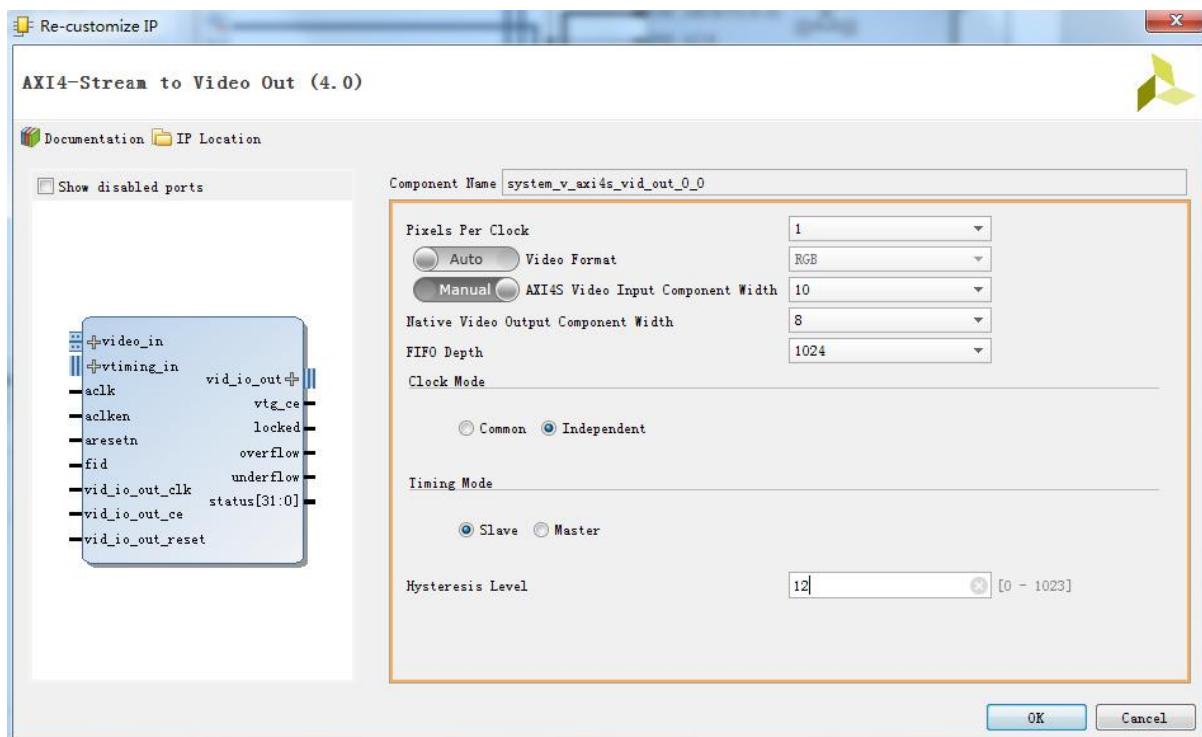
Output Clock	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)	
	Requested	Actual	Requested	Actual	Requested	Actual
<input checked="" type="checkbox"/> clk_out1	25	25.000	0.000	0.000	50.000	50.0
<input checked="" type="checkbox"/> clk_out2	125	125.000	0	0.000	50.000	50.0
<input type="checkbox"/> clk_out3	100.000	N/A	0.000	N/A	50.000	N/A
<input type="checkbox"/> clk_out4	100.000	N/A	0.000	N/A	50.000	N/A
<input type="checkbox"/> clk_out5	100.000	N/A	0.000	N/A	50.000	N/A
<input type="checkbox"/> clk_out6	100.000	N/A	0.000	N/A	50.000	N/A
<input type="checkbox"/> clk_out7	100.000	N/A	0.000	N/A	50.000	N/A

Below the table, there is a checkbox labeled 'USE CLOCK SEQUENCING' and a section titled 'Clocking Feedback' with 'Source' and 'Signaling' tabs.

5.7 VID_OUT IP 的分析

5.7.1 VID_OUT 的参数介绍

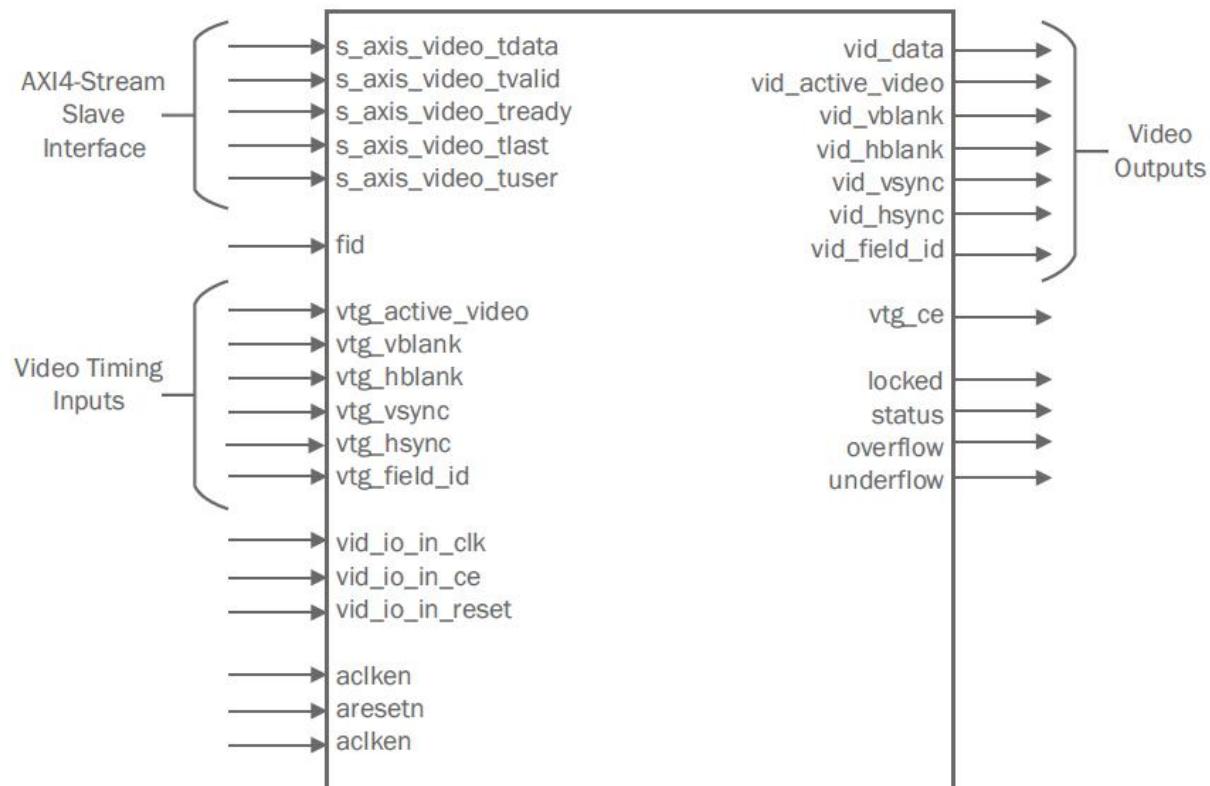




这些参数和前面的 V_TPG 参数类似

- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟
- Video Format: 视频格式
- FIFO Depth: FIFO 深度
- Hysteresis Level: 滞后输出

5.7.2 VID_OUT IP 接口信号的定义



Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_out_ce	Input	1	Native video clock enable
vid_io_out_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
vtg_ce	Output	1	VTC clock enable. Used to halt the timing generator for synchronization purposes.
locked	Output	1	Flag indicating whether the VTC is locked to the input timing. 1=locked. Synchronous to vid_io_out_clk.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk.

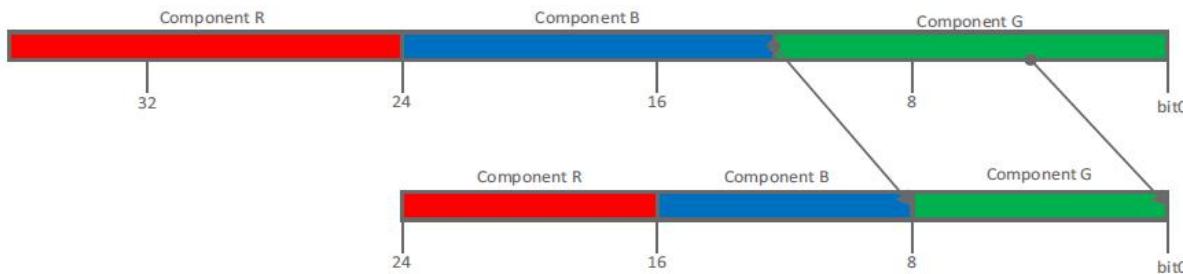
Video Timing Interface

Signal Name	Direction	Width	Description
vtg_vsync	In	1	VTC vertical sync. Active High
vtg_hsync	In	1	VTC horizontal sync. Active High
vtg_vblank	In	1	VTC vertical blank. Active High
vtg_hblank	In	1	VTC horizontal blank. Active High
vtg_act_vid	In	1	VTC active video signal. 1 = active video, 0 = blanked video
vtg_field_id	In	1	VTC field ID. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

AXI4 - Stream Interface

Signal Name	Direction	Width	Description
s_axis_video_tvalid	Input	1	AXI4-Stream TVALID. Active video data enable
s_axis_video_tuser	Input	1	AXI4-Stream TUSER. Start of Frame
s_axis_video_tlast	Input	1	AXI4-Stream TLAST. End of Line
s_axis_video_tready	Output	1	AXI4-Stream TREADY. Inverted FIFO full

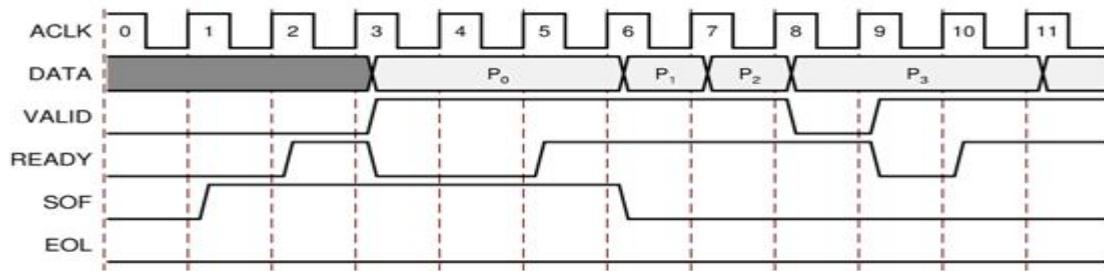
对于 s_axis_video_tdata(TDATA)需要注意一些事情,一般情况下我们的 RGB888 输出,但是,如果 s_axis_video_tdata 是 32bit 那么 VID_OUT IP 会自动截取到 24bit。由于技术手册只给出了 12bit 到 8bit 的截取方式,也就是 RGB 12:12:12 到 RGB 8:8:8 如下图:



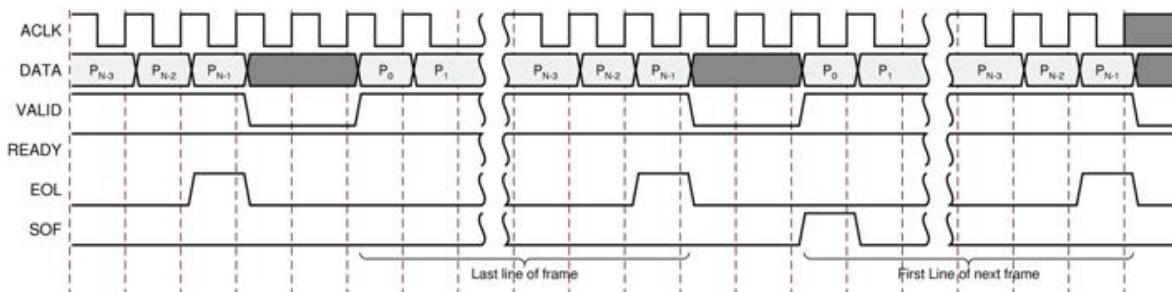
这种截取比较简单,把每个色度的低 4bit 截取就可以了。但是如果是 RGB10:10:10 ,官方并没有给出截取方式,但是可以通过纯色输出来进行测试。

因此最简单的办法是无需任何截取了,如果 s_axis_video_tdata 是 RGB8:8:8 那就无需任何截取,笔者设计的时候由于 AXI 总线是 32bit 因此数据的低 24bit 为 RGB 8:8:8 只要去掉高 24-31bit 就可以取得 RGB8:8:8,这样最省事。

以下时序图是在 SOF 是一帧图像的开始,当 VALID 和 READY 有效的时候开始传输数据。



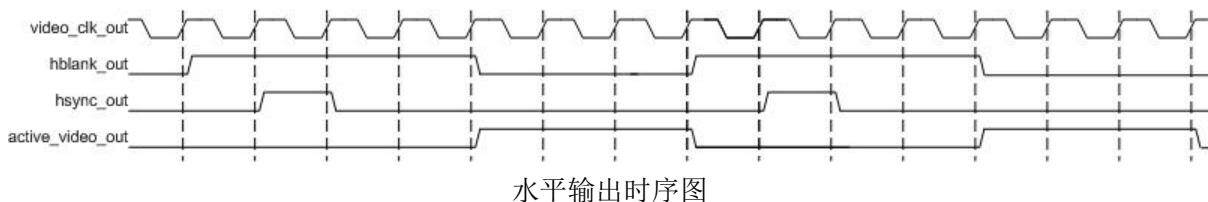
EOL 代表每一行的最后一个数据, SOF 代表前一帧的最后一行的结束, 下一帧第一行的开始。SOF 为 1 个 PLUS 有效 (pg044_v_axis_out.pdf 没有描述清楚, 而且有错误)。



Example Horizontal Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0003_0003
0x0070	Generator HSize	0x0000_0007
0x0078	Generator HSync	0x0005_0004
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

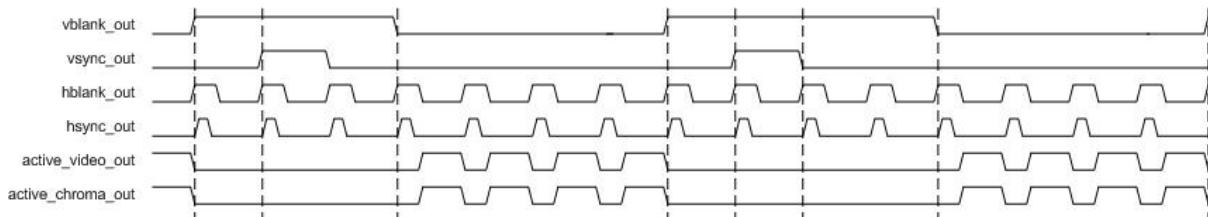
设置水平输出的相关寄存器



Example Vertical Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0004_0003
0x0070	Generator HSize	0x0000_0007
0x0074	Generator VSize	0x0000_0008
0x0078	Generator HSync	0x0005_0004
0x0080	Generator Frame 0 Vsync	0x0006_0005
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

设置垂直输出的相关寄存器



垂直输出时序图

5.8 FPGA 实现的用户逻辑代码

5.8.1 关键信号 1

```
assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready &
m_axis_video_tlast &(vid_in_v_cnt == VID_IN_VS); // dma in last signal
```

m_axis_video_tvalid:此信号是 vid in IP 输出的，代表输出数据有效

s_axis_s2mm_tready:此信号是 DMA IP 输出的，代表 DMA 可以接收数据

m_axis_video_tlast:这是每一行图像数据的最后一个像素的信号标志

vid_in_v_cnt == VID_IN_VS:表示一副图像的最后一个像素输出。

s_axis_s2mm_tlast:所有这些信号有效的时候代表 DMA 的最后一个数据
s_axis_s2mm_tlast 信号有效。

5.8.2 关键信号 2

```
assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready &
```

(vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); //vid out user
 m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。
 s_axis_video_tready: vid out IP 准备好了, 可以接收数据
 (vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); 行计数器为 0 场计数器也为 0 说明要么这副图像已经结束, 也可以理解为下一副图像开始前。这样结合 s_axis_video_tready, m_axis_mm2s_tvalid 为 1, 基于 FPGA 时序, 下一个时钟输出 s_axis_video_tuser 为 1 正好是一副图像的第一个像素。
 s_axis_video_tuser: 因此 s_axis_video_tuser 代表了每一副图像开始的第一个像素。

5.8.3 关键信号 3

```
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == VID_OUT_HS); //vid out last signal
```

m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。
 s_axis_video_tready: vid out IP 准备好了, 可以接收数据
 vid_out_h_cnt == VID_OUT_HS: 图像一行数据的最后一个像素。

5.8.4 部分关键代码

表 3-6-4-1

<pre>reg [10:0] vid_out_v_cnt; reg [10:0] vid_out_h_cnt; reg [10:0] vid_in_v_cnt; parameter VID_OUT_HS = 11'd639;//图像输出行分辨率 parameter VID_OUT_VS = 11'd479;//图像输出场分辨率 parameter VID_IN_VS = 11'd479; always@(posedge FCLK_CLK0) begin if(!gpio_rtl_tri_o_0) vid_out_v_cnt <= 11'd0; else if(m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == VID_OUT_HS)) if(vid_out_v_cnt != VID_OUT_VS) vid_out_v_cnt <= vid_out_v_cnt + 1'b1; else vid_out_v_cnt <= 11'd0; end end</pre>
--

```
        vid_out_v_cnt <= vid_out_v_cnt;
    end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_out_h_cnt <= 11'd0;
    else
        if(m_axis_mm2s_tvalid & s_axis_video_tready)
            if(vid_out_h_cnt != VID_OUT_HS)
                vid_out_h_cnt <= vid_out_h_cnt + 1'b1;
            else
                vid_out_h_cnt <= 11'd0;
        else
            vid_out_h_cnt <= vid_out_h_cnt;
    end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_in_v_cnt <= 11'd0;
    else
        if(m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast)
            if(vid_in_v_cnt != VID_IN_VS)
                vid_in_v_cnt <= vid_in_v_cnt + 1'b1;
            else
                vid_in_v_cnt <= 11'd0;
        else
            vid_in_v_cnt <= vid_in_v_cnt;
    end

assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == 11'd0) &
(vid_out_v_cnt == 11'd0); //vid out user
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt ==
VID_OUT_HS); //vid out last signal

assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast
&(vid_in_v_cnt == VID_IN_VS); //dma in last signal
```

5.9 PS 部分

5.9.1 DMA 中断函数部分分析

为了让图像输出高品质效果，PS 部分设计了 3 缓存处理机制。3 缓存处理机制在大量图像缓冲处理方法是最有效的办法之一。

在 DMA_intr.h 文件中，定义 3 段内存空间用于保存三副最新的图像。

```
#define BUFFER0_BASE (MEM_BASE_ADDR)
#define BUFFER1_BASE (MEM_BASE_ADDR + IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define BUFFER2_BASE (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
```

在 DMA_intr.h 文件中，还定义一下 2 个变量 1 个指针数组。tx_buffer_index; 指示了当前的发送缓存序号，rx_buffer_index; 指示了当前的接收缓存序号。*BufferPtr[3] 会被制定到对应的内存地址空间。

```
extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;
extern u32 *BufferPtr[3];
```

在 main 函数里面有这么一段实现了指针数组指向内存地址空间。

```
BufferPtr[0] = (u32 *)BUFFER0_BASE;
BufferPtr[1] = (u32 *)BUFFER1_BASE;
BufferPtr[2] = (u32 *)BUFFER2_BASE;
```

下面给出 dma_intr.h 的完整代码

表 3-7-1-1 dma_intr.h

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
*/
#ifndef DMA_INTR_H
#define DMA_INTR_H
#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
#include "xscugic.h"

/**************** Constant Definitions *****/
/*
```

```
* Device hardware build related constants.  
*/  
  
#define DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID  
  
#define MEM_BASE_ADDR      0x10000000  
  
#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR  
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR  
  
#define IMAGE_WIDTH      640  
#define IMAGE_HEIGHT     480  
#define BYTES_PER_PIXEL   4  
#define BUFFER_NUM       2  
  
#define MEM_BASE_ADDR      0x10000000  
  
#define BUFFER0_BASE      (MEM_BASE_ADDR )  
#define BUFFER1_BASE      (MEM_BASE_ADDR +     IMAGE_WIDTH * IMAGE_HEIGHT *  
BYTES_PER_PIXEL)  
#define BUFFER2_BASE      (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *  
BYTES_PER_PIXEL)  
  
/* Timeout loop counter for reset  
 */  
#define RESET_TIMEOUT_COUNTER    10000  
/* test start value  
 */  
#define TEST_START_VALUE    0xC  
/*  
 * Buffer and Buffer Descriptor related constant definition  
 */  
#define MAX_PKT_LEN      (IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)  
/*  
 * transfer times  
 */  
#define NUMBER_OF_TRANSFERS 100000  
  
extern volatile int TxDone;
```

```

extern volatile int RxDone;
extern volatile int Error;

extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;

extern u32 *BufferPtr[3];

int DMA_CheckData(int Length, u8 StartValue);
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16
RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr);
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);
#endif

```

每次一幅图像通过 DMA 进入 DDR 后，会产生 DMA 中断请求，在 DMA 中断请求中，会指定下一次 DMA 接收的 buffer 位置。

表 3-7-1-2 DMA_RxIntrHandler 函数

```

/****************************************************************************
*/
/*
*
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.
*
* @param    Callback is a pointer to RX channel of the DMA engine.
*
* @return   None.
*
* @note    None.
*
*/
static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    u32 Status;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */

```

```
IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);

/* Acknowledge pending interrupts */
XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

/*
 * If no interrupt is asserted, we do not do anything
 */
if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
    return;
}

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
 */
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
    xil_printf("rx error! \r\n");
    return;
}

/*
 * If completion interrupt is asserted, then set RxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
    RxDone++;
}

if(rx_buffer_index == 2)
    rx_buffer_index = 0;
else
    rx_buffer_index++;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[rx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma failed! 0 %d\r\n", Status);
    return;
}

}
```

发送函数通过 tx_buffer_index 标记需要发送的缓存部分,并且确保发送的是最新保存的一副图像。

表 3-7-3 DMA_TxIntrHandler

```
/*************************************************************************/
/*
 *
 * This is the DMA TX Interrupt handler function.
 *
 * It gets the interrupt status from the hardware, acknowledges it, and if any
 * error happens, it resets the hardware. Otherwise, if a completion interrupt
 * is present, then sets the TxDone.flag
 *
 * @param    Callback is a pointer to TX channel of the DMA engine.
 *
 * @return   None.
 *
 * @note    None.
 *
 */
static void DMA_TxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    u32 Status;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE);

    /* Acknowledge pending interrupts */

    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {

        return;
    }

    /*

```

```

    * If error interrupt is asserted, raise error flag, reset the
    * hardware to recover from the error, and return with no further
    * processing.
    */
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

    //Error = 1;
    xil_printf("tx error! \r\n");
    return;
}

/*
 * If Completion interrupt is asserted, then set the TxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    TxDone++;
}

if(rx_buffer_index == 0)
    tx_buffer_index = 2;
else
    tx_buffer_index = rx_buffer_index - 1;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[tx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma failed! 0 %d\r\n", Status);
    return;
}
}

```

5.9.2 main.c 文件

这个主程序比较简单，内容比上一个课程的精简很多，这里需要注意的地方是 XGpio_DiscreteWrite(&Gpio, 1, 1); 函数这个函数是这只摄像头和 DMA 之间数据同步的，没有这个同步图像容易错位。另外在主函数里面首先启动 DMA 接收和发送中断各一次，以后就可以在中断里面继续触发了。

表 3-7-2-1 main.c

```

/*
 *

```

```
* www.osrc.cn
* www.milinker.com
* copyright by nan jin mi lian dian zi www.osrc.cn
* axi dma test
*
*/
#include "dma_intr.h"
#include "sys_intr.h"
#include "xgpio.h"

volatile int TxDone;
volatile int RxDone;
volatile int Error;

volatile u8 tx_buffer_index;
volatile u8 rx_buffer_index;

u32 *BufferPtr[3];

static XScuGic Intc; //GIC
static XAxiDma AxiDma;
static XGpio Gpio;

#define AXI_GPIO_DEV_ID           XPAR_AXI_GPIO_0_DEVICE_ID

int init_intr_sys(void)
{
    DMA_Intr_Init(&AxiDma,0); //initial interrupt system
    Init_Intr_System(&Intc); // initial DMA interrupt system
    Setup_Intr_Exception(&Intc);
    DMA_Setup_Intr_System(&Intc,&AxiDma,TX_INTR_ID,RX_INTR_ID); //setup dma interrupt
    system
    DMA_Intr_Enable(&Intc,&AxiDma);
}

int main(void)
{
    u32 Status;
    BufferPtr[0] = (u32 *)BUFFER0_BASE;
    BufferPtr[1] = (u32 *)BUFFER1_BASE;
```

```
BufferPtr[2] = (u32 *)BUFFER2_BASE;

tx_buffer_index = 0;
rx_buffer_index = 0;
TxDone = 0;
RxDone = 0;
Error = 0;

XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);
XGpio_SetDataDirection(&Gpio, 1, 0);
init_intr_sys();

Miz702_EMIO_init();
ov7725_init_rgb();

XGpio_DiscreteWrite(&Gpio, 1, 1);
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[rx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[tx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

while (1);
    return XST_SUCCESS;
}
```

5.10 实验效果



左图 OV7725 右图 OV5640

S03_CH06_AXI_VDMA_OV7725 摄像头采集系统

本课程将对 Xilinx 提供的一款 IP 核——AXI VDMA (Video Direct Memory Access) 进行详细讲解，为后续的学习和开发做好准备。内容安排如下：首先分析为什么要使用 VDMA、VDMA 的作用；然后详细介绍 VDMA 的特点、寄存器空间；最后阐述如何使用 VDMA，包括 IP 核的配置方法、代码编写流程等。

本章主要是理论学习，学习完本章，会对 VDMA 有全面的认识，有利于学习后续的图像生成、视频采集处理系统。由于 VDMA 主要用于视频流数据的存取，单独测试的意义不大，所以在接下来的章节会提供一些样例设计，进一步学习如何使用 VDMA。

6.1 为什么要用 VDMA

在讲解 VDMA 之前，先来探讨一下为什么要学习和使用 VDMA，以明确学习目的。由于使用 VDMA 可以方便地实现双缓冲和多缓冲机制，所以本小节引入了帧缓存和缓冲机制的概念。另外，VDMA 可以很好地契合 Zynq 内部架构，缩短开发周期。再加上 VDMA 本身能够高效地实现数据存取，所以在基于 Zynq（也包括其他 Xilinx FPGA）图像、视频处理系统中，VDMA 可谓是必不可少的。

6.1.1 什么是帧缓存

帧缓冲存储器(Frame Buffer)：简称帧缓存或显存，它是屏幕所显示画面的一个直接映象，又称为位映射图(Bit Map)或光栅。帧缓存的每一存储单元对应屏幕上的一个像素，整个帧缓存对应一帧图像。

在开发者看来，FrameBuffer 是一块显示缓存，往显示缓存中写入特定格式的数据就意味着向屏幕输出内容。所以说 FrameBuffer 就是一块画布，系统在画布上绘制好画面之后，就可以通知显示设备读取 Frame Buffer 进行显示了。

注意，笔者这里所说的 Frame Buffer 和 Linux 的 Frame Buffer 不是同一个概念，这里仅指显示缓存（画布）本身，并不是 Linux 下的一个设备。

6.1.2 双缓冲机制

最早解释多缓冲区如何工作的方式，是通过一个现实生活中的实例来解释的。在一个阳光明媚的日子，你想将水池里的水打满，而又找不到水管的时候，就只能用手边的木桶来灌满水池。水桶满了之后，关掉水龙头，将水提到水池旁边，倒进去，然后走回到水龙头。重复上述工作，如此往复直到将水池灌满。这就类似单缓冲工作过程，当你想将木桶里的水倒出的时候，你必须关掉水龙头。

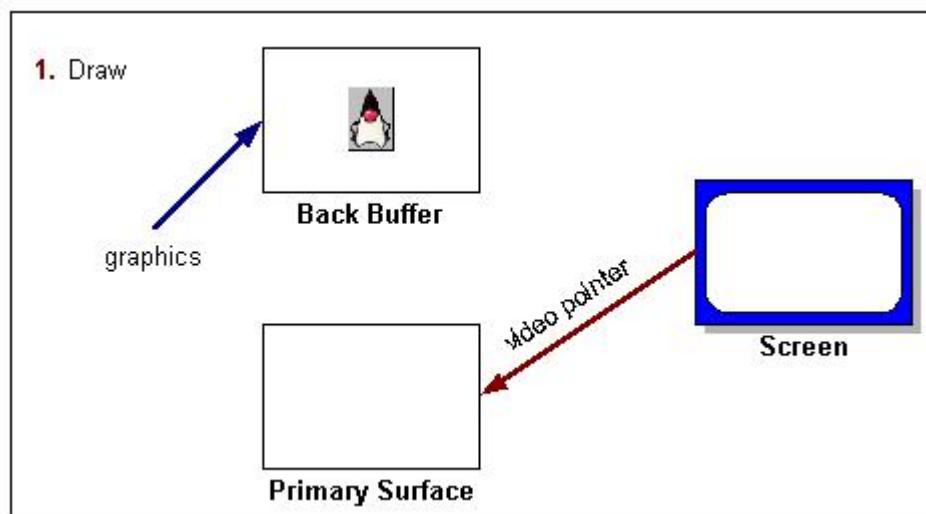
现在假设你用两个木桶来做上面的工作。你会注满第一个木桶然后将第二个木桶换到水龙头下面，这样，在第二个木桶注满的时间内，你就可以将第一个木桶里面的水倒进水池里面，当你回来的时候，你只需要再将第一个木桶换下第二个注满水木桶，当第一个木桶开始注水的时候你就将第二个木桶里面的水倒进水池里面。重复这个过程直到水池被注满。很容易看得到用这种技术注满水池将会更快，同时也节省了很多等待木桶被注满的时间，而这段时间里你什么也做不了，而水龙头也就不用等待从木桶被注满到你回来的这段时间了。

当你雇佣另外一个人来搬运一个被注满的木桶时，这就有点类似于三个缓冲区的工作原理。如果将搬运木桶的时间很长，你可以用更多的木桶，雇佣更多的人，这样水龙头就会一直开着注满木桶了。

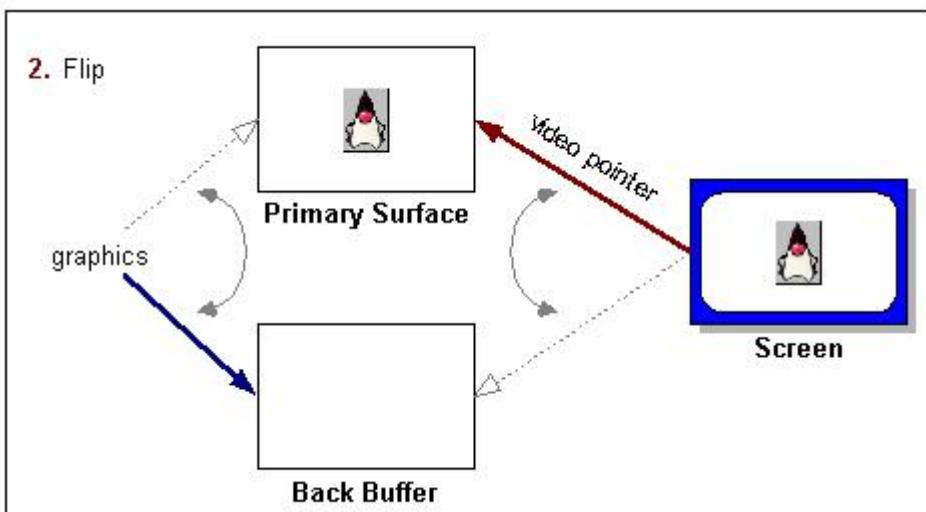
在计算机图形学中，双缓冲是一种画图技术，使用这种技术可以使得画图没有（至少是减少）闪烁、撕裂等不良效果，并减少等待时间。

双缓冲机制的原理大概是：所有画图操作将它们画图的结果保存在一块系统内存区域中，这块区域通常被称作“后缓冲区（back buffer）”，当所有的绘图操作结束之后，将整块区域复制到显示内存中，这个复制操作通常要跟显示器的光栈束同步，以避免撕裂。双缓冲机制必须要求有比单缓冲更多的显示内存和CPU消耗时间，因为“后缓冲区”需要显示内存，而复制操作和等待同步需要CPU时间。

基于双缓冲机制可以实现页交换，页交换初始状态如下图所示：



如上图所示，此时由于处于初始状态，画图操作的结果都在后缓冲区中，而屏幕上显示的则是前缓冲区中的内容。此时画图操作尚未完成，画图操作完成之后，页转换操作开始执行，示意图如下图所示：



如上图所示，画图操作结束，下一个画图操作的结果保存对象指向前缓冲区，屏幕的显示对象指向后缓冲区，此时前缓冲区变成实际意义上的后缓冲区，后缓冲区变成实

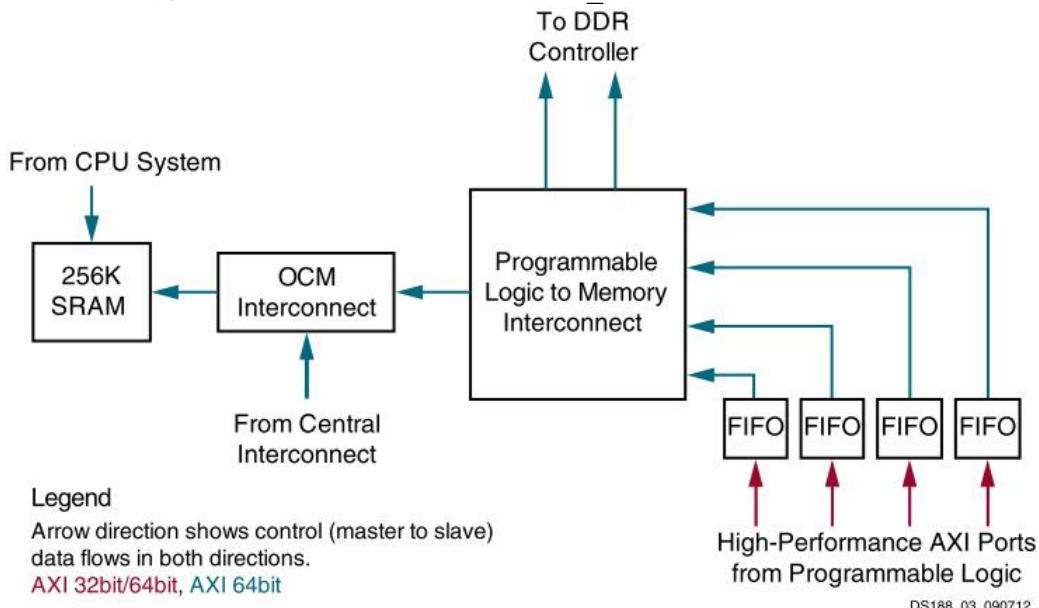
际意义上的前缓冲去，即实现“页交换”操作。

有时候也在页交换链中设置多个“后缓冲区”，这是就需要多缓冲区机制的支持。

6.1.3 Zynq 硬件架构

在 Zynq 芯片内部，PS 和 PL 是共享 DDR 控制器的。PS 访问 DDR 十分简单，只要操作 DDR 映射的虚拟地址即可。对于 PL 而言，要接入 DDR，必须通过 AXI_HP 端口。

Zynq 共有四个 AXI_HP 通道，通道数据宽度可以配置为 32 位或 64 位，这些接口通过 FIFO 控制器连接 PL 到存储接口上，其中有两条连接到 DDR 存储控制器上，还有一条是连接到双端口的 OCM 上的，下图是 AXI_HP 访问 DDR 和 OCM 的连接图。



由上图可以看出，AXI_HP 接口也是遵循 AXI 协议的，因此利用 VDMA 可以直接连接 HP 端口。除了使用 VDMA，当然也可以自己开发出符合 AXI 协议的 IP，但是综合考虑设计成本，没太有必要自己实现。此外，自己实现的 IP 功能也不见得比 VDMA 强大。

6.1.4 VDMA 的作用

VDMA 数据接口可以分为读、写通道，用户可以通过写通道将 AXI-Stream 类型的数据流写入 DDR3，通过读通道可以从 DDR3 读取数据，并以 AXI-Stream 类型的格式输出。由此可知，VDMA 本质上是一个数据搬运 IP，为数据进、出 DDR3 提供了一种便捷的方案。

将数据存入 DDR 之后，CPU 就可以进行一些处理（缩放、裁剪等），然后再送至显示设备，达到期望的应用目的。当然，也可能是简单地对捕获的视频进行解析，将数据存入帧缓存，以供显示。

VDMA 可以控制多达 32 个帧存，并可以自由地进行帧存切换，所以就能够轻松地实现双缓冲和多缓冲操作。这也是一个很重要的特性，在后续进行系统设计的时候，通常是采用多缓冲的方式实现显示。

由以上分析可以发现，在基于 Zynq 的图像、视频处理系统中使用 VDMA 是十分有必要的。

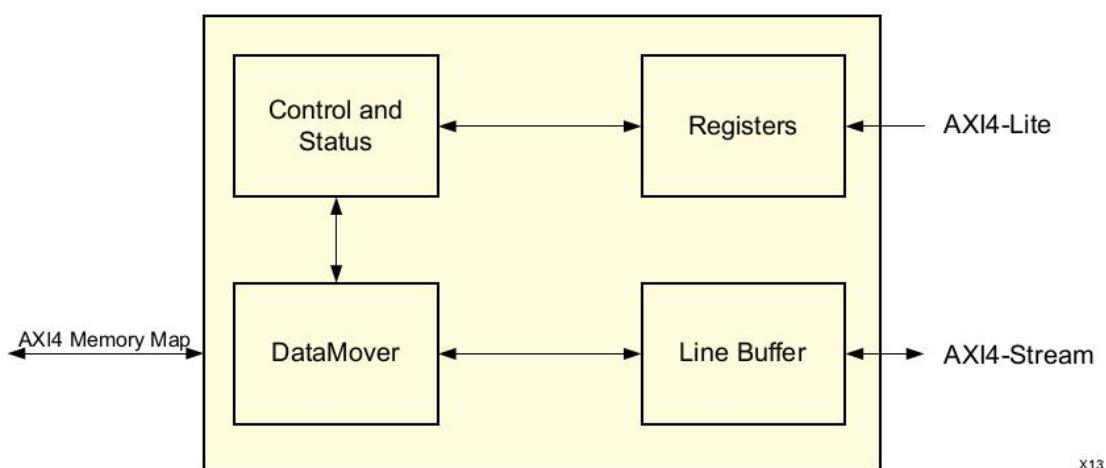
6.2 VDMA 概述

AXI VDMA 是 Xilinx 提供的软核 IP，用于将 AXI Stream 格式的数据流转换为 Memory Map 格式或将 Memory Map 格式的数据转换为 AXI Stream 数据流，从而实现与 DDR3 进行通信。

许多视频类应用都需要帧缓存来处理帧率变化或者进行图像的缩放、裁剪等尺寸变换操作。AXI VDMA 设计的初衷就是用于高效地实现 AXI4-Stream 视频流接口和 AXI4 接口之间的数据传输。

VDMA 的关键特性&优势有以下几点：

- 使视频流能够高带宽直接接入内存
 - 高效的二维 DMA 操作
 - 独立的异步读写通道操作
 - Gen-Lock 帧存同步机制
 - 最多支持 32 个帧存
 - 支持视频格式动态切换
 - 猝发长度和行缓存深度可调节
 - 处理器可以控制 IP 的初始化、状态、中断和管理寄存器
 - 基础 AXI 流数据位宽为 8 的整数倍，如 8,16,24, 32 等，最大可达 1024 个位
- AXI VDMA 框图如下所示。



主要有以下几种接口类型：

- AXI-lite: PS 通过该接口来配置 VDMA
- AXI Memory Map write: 映射到存储器写
- AXI Memory Map read: 映射到存储器读
- AXI Stream Write(S2MM): AXI Stream 视频流写入图像
- AXI Stream Read(MM2S): AXI Stream 视频流读出图像

从框图中可以看出，VDMA 主要由控制和状态寄存器、数据搬运模块、行缓冲这几部分构成。数据进出 DDR 要经过行缓冲进行缓存，然后由数据搬运模块写入或者读出数据。数据搬运模块具体如何工作，由相关寄存器负责控制。VDMA 的工作状态可以通过读取状态寄存器进行获取。

6.3 VDMA 详细介绍

6.3.1 接口

6.3.1.1 时钟和复位

各种总线都有自己的时钟信号，不用特别说明，需要指出的是，这些时钟是异步的，并不需要用同一个时钟。但在设计过程中，如无特别需求，可以使用相同的时钟，以降低设计难度。

同步复位信号 axi_resetn，同步时钟为 s_axi_lite_aclk，低电平有效（至少要保持 16 个时钟周期的低电平，才能够生效），有效时复位整个 IP 核。

6.3.1.2 AXI 总线相关信号

- AXI4-Lite 接口 (S_AXI_LITE)
- AXI4 读接口 (M_AXI_MM2S)
- AXI4 写接口 (M_AXI_S2MM)
- AXI4-Stream 主接口 (M_AXI_MM2S)
- AXI4-Stream 从接口 (S_AXI_S2MM)

前缀 S_、M_ 分别表示 Slave 和 Master；后缀 MM2S、S2MM 说明数据流向是从 memory map 到 stream 还是从 stream 到 memory map。具体每个接口所包含的信号，在基础篇第 20 章已有介绍，此处不再重复。

6.3.1.3 视频同步接口信号

信号名称	方向	详细描述
mm2s_fsync	Frame	MM2S 帧同步输入。使能该信号后，VDMA 操作开始于 fsync 每个下降沿。该信号至少要持续一个 m_axis_mm2s_aclk 时钟周期
	Sync	
s2mm_fsync	Frame	S2MM 帧同步输入。使能该信号后，VDMA 操作开始于 fsync 每个下降沿。该信号至少要持续一个 s_axis_s2mm_aclk 时钟周期
	Sync	

6.3.1.4 GenLock 相关信号

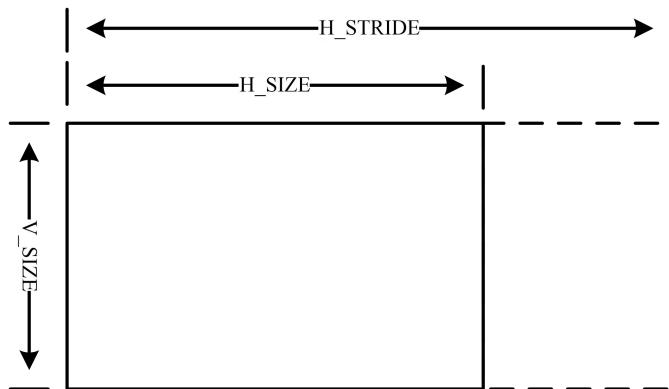
在下一节将详细介绍这些信号的作用和应用场合。

信号名称	方向	详细描述
mm2s_frame_ptr_in(5:0)	输入	输入的帧编号
mm2s_frame_ptr_out(5:0)	输出	输出当前帧的编号

s2mm_frame_ptr_in(5:0)	输入	输入的帧编号
s2mm_frame_ptr_out(5:0)	输出	输出当前帧的编号

6.3.2 VDMA 帧存格式

在讲述寄存器时，需要设定和显示（帧存）相关的参数，为了方便读者的理解，这里先介绍 VDMA 数据存放框架，如下图所示，黑色实线内的区域为实际存储画面的帧存。



图中 **H_STRIDE** 代表水平方向上的跨度，**H_SIZE** 表示水平方向数据总量，**V_SIZE** 表示竖直方向总共有多少行。

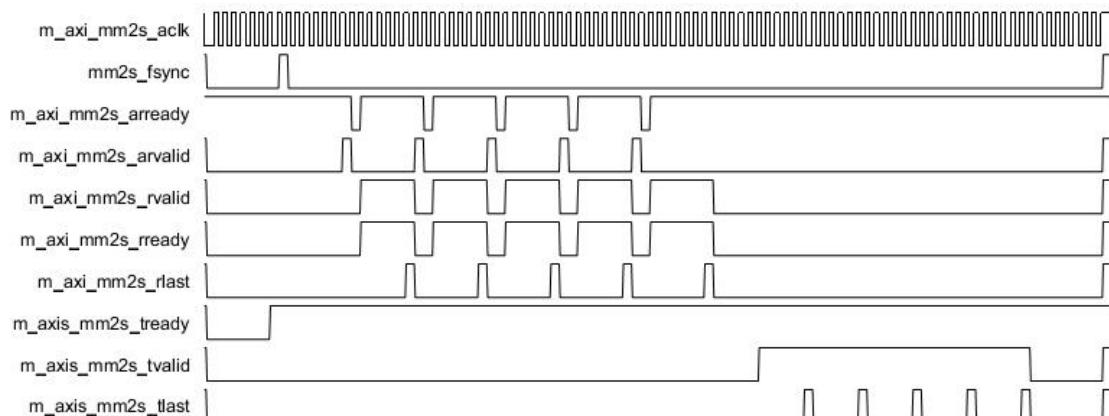
至于帧存内部数据如何组织，就取决于软件代码和硬件逻辑如何匹配了，通常来讲，数据存放格式为 RGB+Alpha 或者 Alpha+RGB。

22.3.3 读写通道工作时序

清晰地理解 VDMA 读写通道的工作时序，对以后的设计有很大的帮助，很多设计都是根据本小节所示的样例时序设计出来的。在下一章，读者就能够有所体会。

6.3.3.1 读通道（MM2S）时序

下图描述了读通道的时序，5 行，每行 16 字节，跨度为 32 字节。

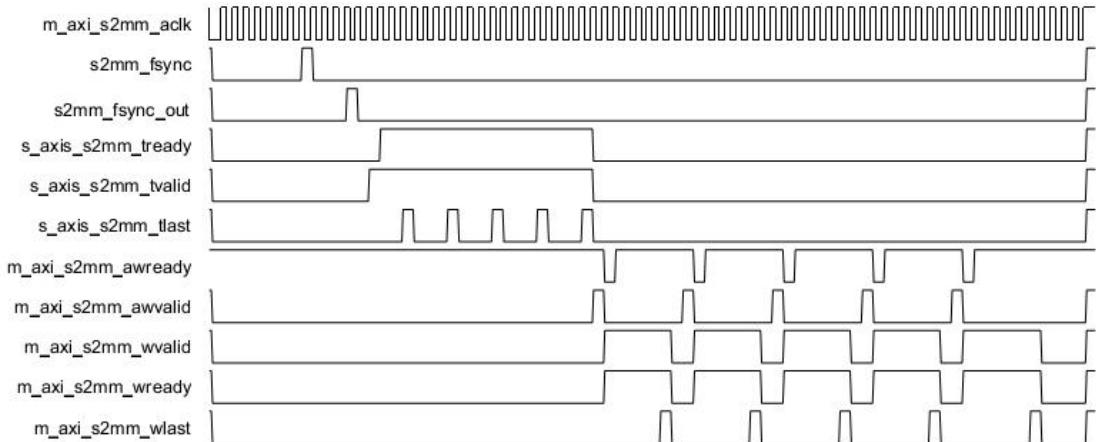


从图中可以看出：在收到 **mm2s_fsync** 信号后，VDMA 在 **m_axi_mm2s_araddr** 的起始地址处发出 **m_axi_mm2s_arvalid** 信号。**M_axi_mm2s_arvalid** 总共有效 5 次，分别获取

一帧的 5 行数据。从 MM 读取的数据存储在行缓存里，当收到来自 axi-stream 端的 m_axis_mm2s_tvalid 信号后，将数据发送到 axi-stream 端。每一行的结束，axi-stream 端会使 m_axis_mm2s_tlast 有效。

6.3.3.2 写通道(S2MM)时序

下图描述了写通道的时序，5 行，每行 16 字节，跨度为 32 字节。



从图中可以看出：在收到 s2mm_fsync 信号后，VDMA 发出 s2mm_fsync_out 和 s_axis_s2mm_tready 表明已经准备好接收来自 axi-stream 端的数据。读取到的数据存储在行缓存里，m_axi_s2mm_awvalid 有效后，紧接着有效 m_axi_s2mm_wvalid 信号，同时将数据放至 m_axi_s2mm_wdata。

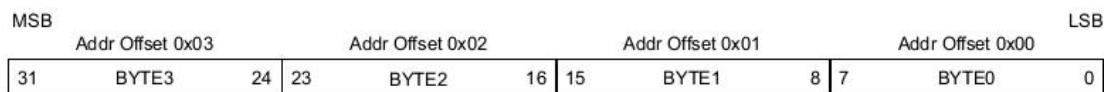
6.3.4 寄存器

VDMA 的寄存器如下表所示。所有寄存器都被映射到非缓存内存空间。该内存空间必须按照 AXI 字(32 位)进行对齐，换句话说，寄存器偏移地址至少间隔 4 个字节。

寄存器名称	偏移地址	详细描述
MM2S_VDMACR	00h	MM2S VDMA 控制寄存器
MM2S_VDMASR	04h	MM2S VDMA 状态寄存器
保留	08h~10h	N/A
MM2S_REG_INDEX	14h	MM2S 寄存器索引
保留	18h~24h	N/A
PARK_PRT_REG	28h	MM2S 和 S2MM Park 指针寄存器
VDMA_VERSION	2Ch	VDMA 版本寄存器
S2MM_VDMACR	30h	S2MM VDMA 控制寄存器
S2MM_VDMASR	34h	S2MM VDMA 状态寄存器
保留	38h	N/A
S2MM_VDMA_IRQ_MASK	3Ch	S2MM 错误中断掩码寄存器
保留	40h	N/A
S2MM_REG_INDEX	44h	S2MM 寄存器索引
保留	48h~4Ch	N/A

MM2S_VSIZE	50h	MM2S 垂直方向显示大小寄存器
MM2S_HSIZE	54h	MM2S 水平方向显示大小寄存器
MM2S_FRMDLY_STRIDE	58h	MM2S 帧延迟和跨度寄存器
MM2S_START_ADDRESS(1~16)	5Ch~98h	MM2S 帧存起始地址 (1~16)
保留	9Ch	N/A
S2MM_VSIZE	A0h	S2MM 垂直方向显示大小寄存器
S2MM_HSIZE	A4h	S2MM 水平方向显示大小寄存器
S2MM_FRMDLY_STRIDE	A8h	S2MM 帧延迟和跨度寄存器
S2MM_START_ADDRESS(1~16)	ACh~E8h	S2MM 帧存起始地址 (1~16)

所有寄存器字节序都是小端格式，如下图所示。

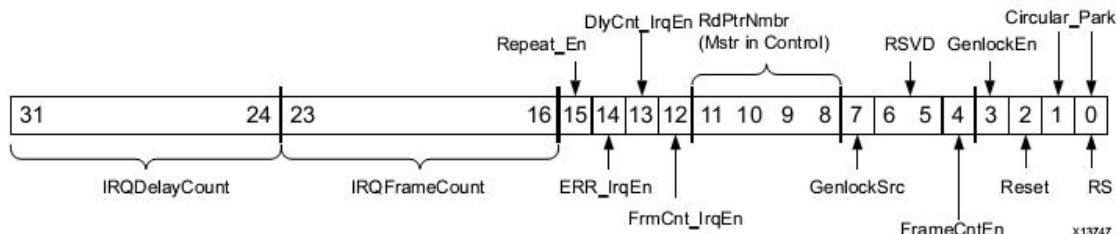


各个寄存器的名称和大致作用从上表就可以看出，接下来，笔者会详细介绍重要寄存器的具体 bit 的作用。明白了每个 bit 的作用之后，自然就知道写入什么值能够达到自己的控制目的。

从上表可以看出，寄存器可以分为两组，分别对应 MM2S 通道和 S2MM 通道，两组寄存器的功能是相似的，区别仅在于偏移地址和所服务的对象。因此，在学习完 MM2S 通道的所有寄存器之后，只要大致浏览一下 S2MM 通道对应的寄存器的关键位即可（个别位不相同），在使用高级功能时，再仔细查阅 VDMA 用户手册。

6.3.2.1 MM2S VDMA 控制寄存器 (00h)

顾名思义，该寄存器用于控制 VDMA，具体可以实现复位、使能锁相同步、设定帧存切换模式、启动 VDMA 读写通道等操作。每一位作用如下图所示，低 4 位是最重要的，接下来会详细介绍。

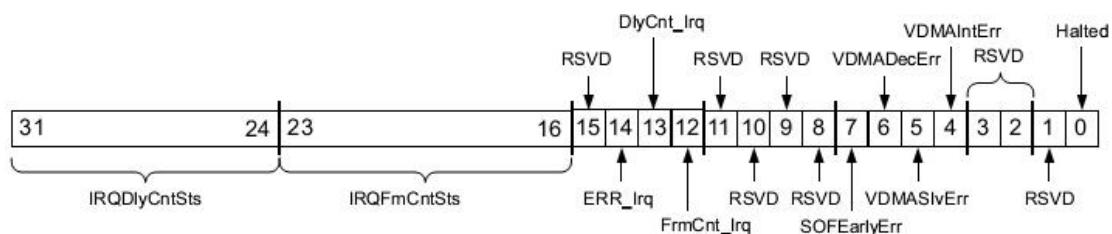


位	名称	默认值	接入类型	描述
31~4				非常用位，请参考 VDMA 使用手册自学
3	GenlockEn	0h	可读可写	使能锁相同步或者动态锁相同步模式。 0：关闭 Genlock 或动态 Genlock 同步 1：开启 Genlock 或动态 Genlock 同步 注：该位仅在通道被配置成锁相同步从接口或者动态锁相主、从接口时才起作用。配置成锁相同步主接口时，该位为保留位，值恒为 0。

2	Reset	0h	可读可写	0: 正常操作; 1: 复位 MM2S 通道
1	Circular_Park	1h	可读可写	指定帧存为循环模式还是停留模式 0: 停留模式 - 显示用缓存页将停留在 PARK_PTR_REG.RdFrmPntrRef 指定的帧存; 1: 循环模式 - 循环切换显示用缓存页
0	RS	0h	可读可写	运行/停止, 控制 VDMA 通道的运行和停止。 开始任何 VDMA 操作前, 该位必须置 1. 0: 停止; 1: 运行。

6.3.2.2 MM2S VDMA 状态寄存器 (04h)

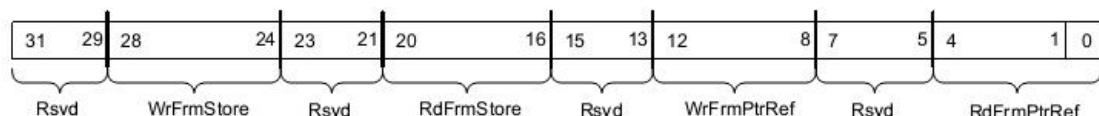
该寄存器用于获取 VDMA 工作状态。每一位作用如下图所示，低 4 位是最重要的，接下来会详细介绍。



位	名称	默认值	接入类型	描述
31~1				非常用位, 请参考 VDMA 使用手册自学
0	Halted	1h	只读	指示 VDMA 运行是否停止。 0: 运行; 1: 停止。

6.3.2.3 PARK_PTR_REG 停留指针寄存器 (28h)

该寄存器用于管理读、写通道的数据传输。



位	名称	默认值	接入类型	描述
31~29	保留	0h	只读	
28~24	WrFrmStore	0h	只读	用于存储写通道正在操作的帧的编号。指示 S2MM 通道正在操作的帧。
23~21	保留	0h	只读	
20~16	RdFrmStore	0h	只读	用于存储读通道正在操作的帧的编号。指示 MM2S 通道正在操作的帧。
15~13	保留	0h	只读	

12~8	WrFrmPtrRef	0h	可读可写	通过帧编号指定写通道操作的帧。当工作在停留模式，S2MM 通道操作对象停留在 WrFrmPtrRef 指定的帧。
7~5	保留	0h	只读	
4~0	RdFrmPtrRef	0h	可读可写	通过帧编号指定读通道操作的帧。当工作在停留模式，MM2S 通道操作对象停留在 RdFrmPtrRef 指定的帧。

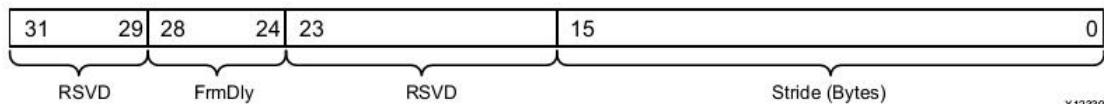
学习了这个寄存器之后，就可以发现：当 VDMA 工作在 Parked 模式下，通过操作该寄存器，就能够实现帧缓存的切换，建立自己想要的缓存切换机制。

6.3.2.4 MM2S 帧存起始地址 (0x5C~0x98)

有最多 32 个寄存器用于存放帧存起始地址，其分别存在于两个寄存器 bank 上：bank0 和 bank1，每个 bank 上有 16 个寄存器。这两个 bank 上有相同的起始偏移地址（0x5C），选择这两个 bank 可以通过 MM2S_REG_INDEX 的值进行选择。假如想访问第 1 个寄存器，则给 MM2S_REG_INDEX 赋值为 0，并设定偏移地址为 0x5C；如果想访问第 17 个寄存器，需要将 MM2S_REG_INDEX 设为 1，并设定初始偏移地址为 0x5C。

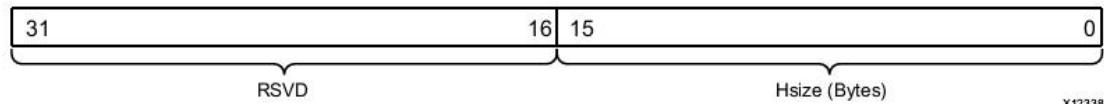
6.3.2.5 MM2S_FRMDLY_STRIDE MM2S 帧延迟和跨度 (58h)

该寄存器有两个作用，第一是 bit24~bit28 指定帧延迟，仅用于 Genlock 从模式，指定从接口比主接口至少要延迟多少个帧；第二是低 16 位指定水平方向的跨度，同样以字节为单位。所谓跨度是指每两行第一个像素之间间隔的数据个数，具体请参考 22.3.2 小节，VDMA 帧存格式。



6.3.2.6 MM2S_HSIZE MM2S 水平方向尺寸 (54h)

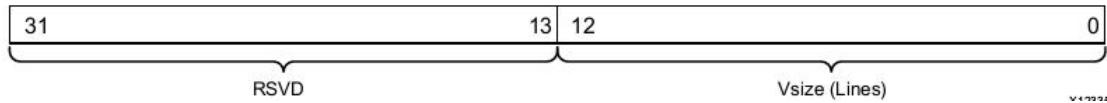
该寄存器的低 16 位用于指定每一行有多少字节的数据需要传输。例如显示分辨率为 640*480，每个像素 4 个字节（RGB+Alpha），该值应该设定为 640*4。



6.3.2.7 MM2S_VSIZE MM2S 垂直方向尺寸 (50h)

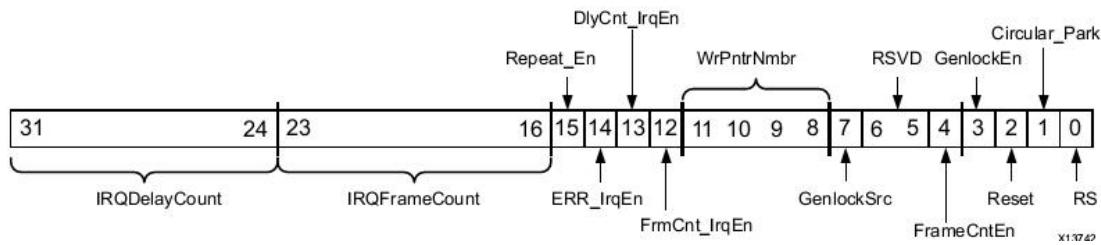
该寄存器有两个作用，第一是用低 13 位指定总共有多少行；第二是启动 MM2S 的传

输。当 MM2S_VDMACR.RS=1，对该寄存器的写操作会将所有设定参数传递给 VDMA 内部寄存器模块，用于 VDMA 控制。**对某个通道进行配置时，必须在最后一步设置该寄存器。**



6.3.2.8 S2MM VDMA 控制寄存器 (30h)

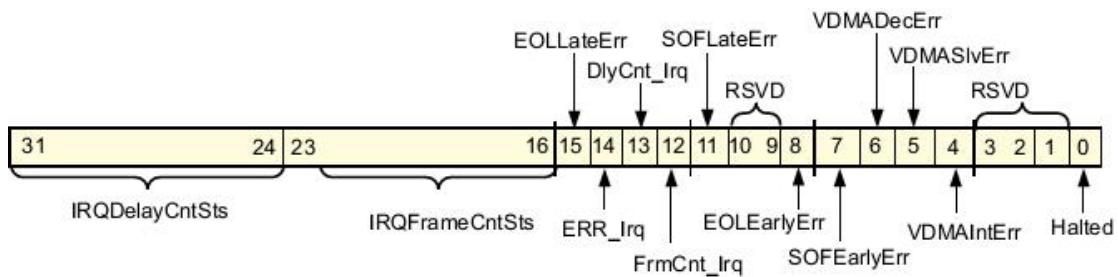
顾名思义，该寄存器用于控制 VDMA S2MM 通道，具体可以实现复位、使能锁相同步、设定帧存切换模式、启动 VDMA 读写通道等操作。每一位作用如下图所示，低 4 位是最重要的，接下来会详细介绍。



位	名称	默认值	接入类型	描述
31~4				非常用位，请参考 VDMA 使用手册自学
3	GenlockEn	0h	可读可写	使能锁相同步或者动态锁相同步模式。 0：关闭 Genlock 或动态 Genlock 同步 1：开启 Genlock 或动态 Genlock 同步 注：该位仅在通道被配置成锁相同步从接口或者动态锁相主、从接口时才起作用。配置成锁相同步主接口时，该位为保留位，值恒为 0。
2	Reset	0h	可读可写	0：正常操作；1：复位 S2MM 通道
1	Circular_Park	1h	可读可写	指定帧存为循环模式还是停留模式 0：停留模式 - 显示用缓存页将停留在 PARK_PTR_REG.RdFrmPntrRef 指定的帧存； 1：循环模式 - 循环切换显示用缓存页
0	RS	0h	可读可写	运行/停止，控制 VDMA 通道的运行和停止。 开始任何 VDMA 操作前，该位必须置 1。 0：停止；1：运行。

6.3.2.9 S2MM VDMA 状态寄存器 (34h)

该寄存器用于获取 S2MM 工作状态。每一位作用如下图所示，低 4 位是最重要的，接下来会详细介绍。



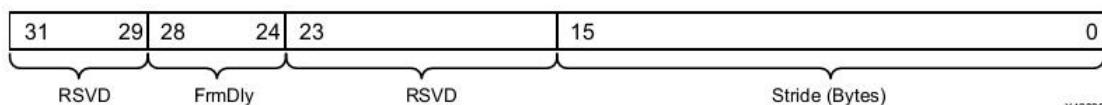
位	名称	默认值	接入类型	描述
31~1				非常用位, 请参考 VDMA 使用手册自学
0	Halted	1h	只读	指示 VDMA 运行是否停止。 0: 运行; 1: 停止。

6.3.2.4 S2MM 帧存起始地址 (0xAC~0xE8)

有最多 32 个寄存器用于存放帧存起始地址，其分别存在于两个寄存器 bank 上：bank0 和 bank1，每个 bank 上有 16 个寄存器。这两个 bank 上有相同的起始偏移地址（0x5C），选择这两个 bank 可以通过 S2MM_REG_INDEX 的值进行选择。假如想访问第 1 个寄存器，则给 S2MM_REG_INDEX 赋值为 0，并设定偏移地址为 0x5C；如果想访问第 17 个寄存器，需要将 MM2S_REG_INDEX 设为 1，并设定初始偏移地址为 0x5C。

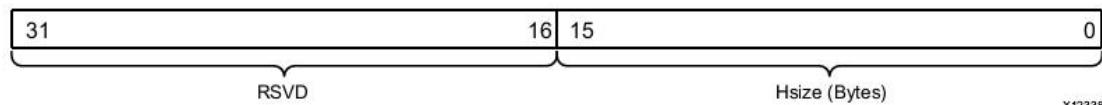
6.3.2.5 S2MM_FRMDLY_STRIDE S2MM 帧延迟和跨度 (A8h)

该寄存器有两个作用，第一是 bit24~bit28 指定帧延迟，仅用于 Genlock 从模式，指定从接口比主接口至少要延迟多少个帧；第二是低 16 位指定水平方向的跨度，同样以字节为单位。所谓跨度是指每两行第一个像素之间间隔的数据个数，具体请参考 22.3.2 小节，VDMA 帧存格式。



6.3.2.6 S2MM_HSIZE S2MM 水平方向尺寸 (A4h)

该寄存器的低 16 位用于指定每一行有多少字节的数据需要传输。例如显示分辨率为 640*480，每个像素 4 个字节（RGB+Alpha），该值应该设定为 640*4。



6.3.2.7 S2MM_VSIZE S2MM 垂直方向尺寸 (A0h)

该寄存器有两个作用，第一是用低 13 位指定总共有多少行；第二是启动 S2MM 的传输。当 S2MM_VDMACR_RS=1，对该寄存器的写操作会将所有设定参数传递给 VDMA 内部寄存器模块，用于 VDMA 控制。**对某个通道进行配置时，必须在最后一步设置该寄存器。**

31	13	12	0
RSVD		Vsize (Lines)	

6.3.5 帧同步选项

VDMA 支持以下三种帧同步源：

- 基于 AXI4-Stream 的帧同步（使用 tuser(0) 信号）
 - 读通道使用 m_axis_mm2s_tuser(0) 作为帧起始信号
 - 写通道使用 s_axis_s2mm_tuser(0) 作为帧起始信号
- S2MM 帧同步(s2mm_fsync)
- MM2S 帧同步(mm2s_fsync)

6.3.6 Genlock 同步机制

6.3.6.1 什么是 Genlock？

Genlock，同步锁相，可以使一套或多套系统与同一同步源实现同步。能够使视频的刷新和外部视频源保持一致。当提供了一个适当的信号后，系统就会把它的显示刷新率和这个信号进行锁定。

在许多视频应用中，输入端产生数据的速率往往不同于输出端数据速率，为了避免由速率不一致导致的潜在错误，帧缓冲的使用是很有必要的。帧缓冲机制开辟多个缓冲页，用于保存数据，输入和输出端分别操作不同的帧存，从而避免了冲突。

VDMA 的锁相同步特性正是用于阻止读、写通道同时操作同一个帧存。VDMA 的每个通道都可以选择自己的操作类型（同步锁相主/从或者动态同步锁相主/从），利用该特性，禁止主从接口同时访问同一缓存，从而保持同步。

VDMA 支持四种模式的锁相同步，分别为：

- Genlock Master (锁相同步主端)
- Genlock Slave (锁相同步从端)
- Dynamic Genlock Master (动态锁相同步主端)
- Dynamic Genlock Slave (动态锁相同步从端)

6.3.6.2 Genlock Master

读通道(MM2S)：当配置为 Genlock Master 时，该通道不会跳过或者重复任一帧数据，并把当前帧的编号输出在 mm2s_frame_ptr_out 端口。通道不会检测

`mm2s_frame_ptr_in` 端口提供的帧编号。Genlock Slave 通道应跟随 Genlock Master 通道变化，但有一定的延迟。延迟大小预定义在寄存器中 (`*frmdly_stride[28:24]`)。

写通道 (S2MM)：当配置为 Genlock Master 时，该通道不会跳过或者重复任一帧数据，并把当前帧的编号输出到 `s2mm_frame_ptr_out` 端口。通道不会检测 `s2mm_frame_ptr_in` 端口提供的帧编号。Genlock Slave 通道应跟随 Genlock Master 通道变化，但有一定的延迟。延迟大小预定义在寄存器中 (`*frmdly_stride[28:24]`)。

6.3.6.3 Genlock Slave

读通道 (MM2S)：当配置为 Genlock Slave 时，该通道会通过跳过或者重复一些帧的方式，尝试与 Genlock Master 同步。通道会对 `mm2s_frame_ptr_in` 端口进行采样，获取 Genlock Master 的帧编号。为了实现状态反馈，通道会把当前帧的编号输出到 `mm2s_frame_ptr_out` 端口。

指定通道工作在 Genlock Slave 模式，必须进行如下操作。

- 将 GenlockEn 置 1 (`MM2S_VDMACR[3]=1`)，使能主、从通道之间的 Genlock 同步。
- 将 GenlockSrc 置 1 (`MM2S_VDMACR[7]=1`)，使能内部 Genlock 模式。如果在 Vivado IDE 中同时使能读、写通道，该位默认置位。当 GenlockSRC=1 时，VDMA 默认支持内部同步锁相总线。这样一来就没有必要在外部对帧指针端口 (`*frame_ptr_out` 和 `*frame_ptr_in`) 进行连接了。
- 根据主从通道的帧率，使用 `mm2s_frmdly_stride[28:24]` 设定合适的延迟时间。

写通道 (S2MM)：当配置为 Genlock Slave 时，该通道会通过跳过或者重复一些帧的方式，尝试与 Genlock Master 同步。通道会对 `s2mm_frame_ptr_in` 端口进行采样，获取 Genlock Master 的帧编号。为了实现状态反馈，通道会把当前帧的编号输出到 `s2mm_frame_ptr_out` 端口。

指定通道工作在 Genlock Slave 模式，必须进行如下操作。

- 将 GenlockEn 置 1 (`S2MM_VDMACR[3]=1`)，使能主、从通道之间的 Genlock 同步。
- 将 GenlockSrc 置 1 (`S2MM_VDMACR[7]=1`)，使能内部 Genlock 模式。如果在 Vivado IDE 中同时使能读、写通道，该位默认置位。当 GenlockSRC=1 时，VDMA 默认支持内部同步锁相总线。这样一来就没有必要在外部对帧指针端口 (`*frame_ptr_out` 和 `*frame_ptr_in`) 进行连接了。
- 根据主从通道的帧率，使用 `mm2s_frmdly_stride[28:24]` 设定合适的延迟时间。

6.3.6.4 Dynamic Genlock Master

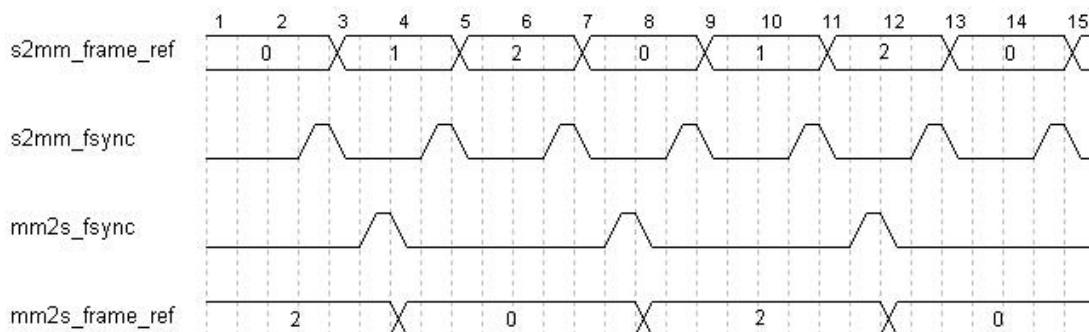
动态 Genlock Master 与 Genlock Master 的区别在于，主通道会跳过从通道正在操作的帧。举例而言，对于三帧存而言，动态 Genlock Master 会按照 0, 1, 2, 0, 1, 2 的顺

序循环使用帧存，一旦检测到 Master 即将操作 Slave 正在操作的帧，就会跳过该帧继续循环。因此，如果 Slave 通道一直在操作帧存 1，那么 Master 通道就会在帧 0 和帧 2 之间来回切换。

6.3.6.5 Dynamic Genlock Slave

Dynamic Genlock Slave 通道会操作 Dynamic Genlock Master 通道上一周期操作的帧。

下图描述了一种简单的 Genlock 操作时序。在这个示例中，S2MM 通道是 Genlock Master，MM2S 通道是 Genlock Slave，并且写通道帧率高于读通道帧率。

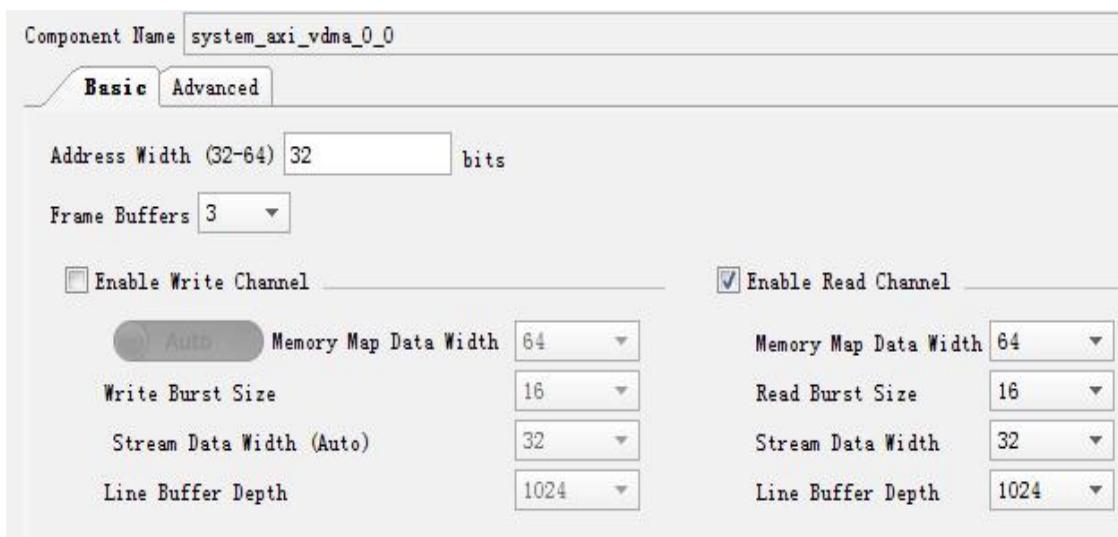


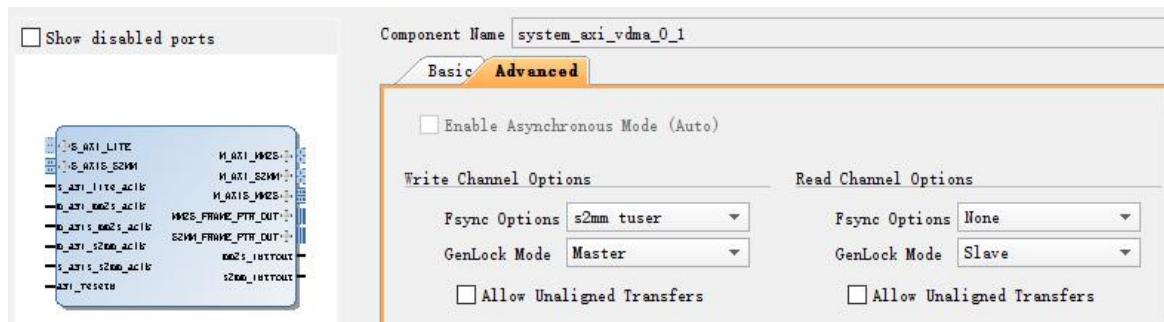
由于读通道帧率慢于写通道，所以读通道仅处理帧 2 和帧 0，跳过帧 1 不做处理。

6.4 使用 VDMA

6.4.1 IP 核配置

Xilinx 集成开发环境升级到 Vivado 之后，VDMA 的配置项比以前少了不少，一定程度上降低了使用难度。主要配置页面如下面两幅图所示。





具体配置项参见下表。

基本配置	高级配置
地址线宽度	是否使能异步模式（自动）
帧存数量	写通道帧同步
是否使能读写通道	写通道 GenLock 模式选择
数据线宽度	写通道是否允许非对齐传输
触发长度	读通道帧同步
AXI-Stream 流数据位宽	读通道 GenLock 模式选择
Line Buffer 深度	读通道是否允许非对齐传输

关于地址线和数据线宽度，需要根据设计的实际情况配置。

Line buffer 深度不能太小。

GenLock 和帧同步前文已经讲解，根据需求自行配置即可。

6.4.2 软件控制流程

以下步骤是最简单的 VDMA 控制初始化操作。

- 写 VDMACR 寄存器，将 VDMACR.RS 设为 1，启动 VDMA 通道。
- 设定有效的帧缓存起始地址。
- 设定帧延迟（仅针对 Genlock 从模式）以及跨度到 FRMDLY_STRIDE 寄存器。
- 设定水平方向字节数到 HSIZE 寄存器。
- 设定竖直方向行数到 VSIZE 寄存器。启动通道的数据传输。

在 VDMA 运行过程中，可以动态的进行显示参数配置，但是需要注意的是，想要使参数生效，必须在设置的最后一步，对 VSIZE 寄存器进行写操作。

最后，给出一段通过 VDMA 对 DDR 读写传输的进行初始化的示例代码：

```
//VDMA configurateAXI VDMA0
/*****************从 DDR 读数据设置*****/
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x0, 0x4); //reset
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x0, 0x8); //gen-lock

XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x5C, 0x08000000);
// AXI4 Data Width 为 32 位，是 4 个字节数
// 0xA000000 0x0015F900
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x5C+4, 0xA0000000);
```

```

// 0x09000000 0x002BF200
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x5C+8, 0x09000000);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x54, 640); // 640
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x58, 0x01000280);
// 第 0 位： 运行 - 启动 VDMA 操作,在运行 VDMA 时，其状态寄存器中的停止位
赋值为 0 第一位：循环模式 -通过连续循环帧缓冲
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x0, 0x03);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x50, 480); //480

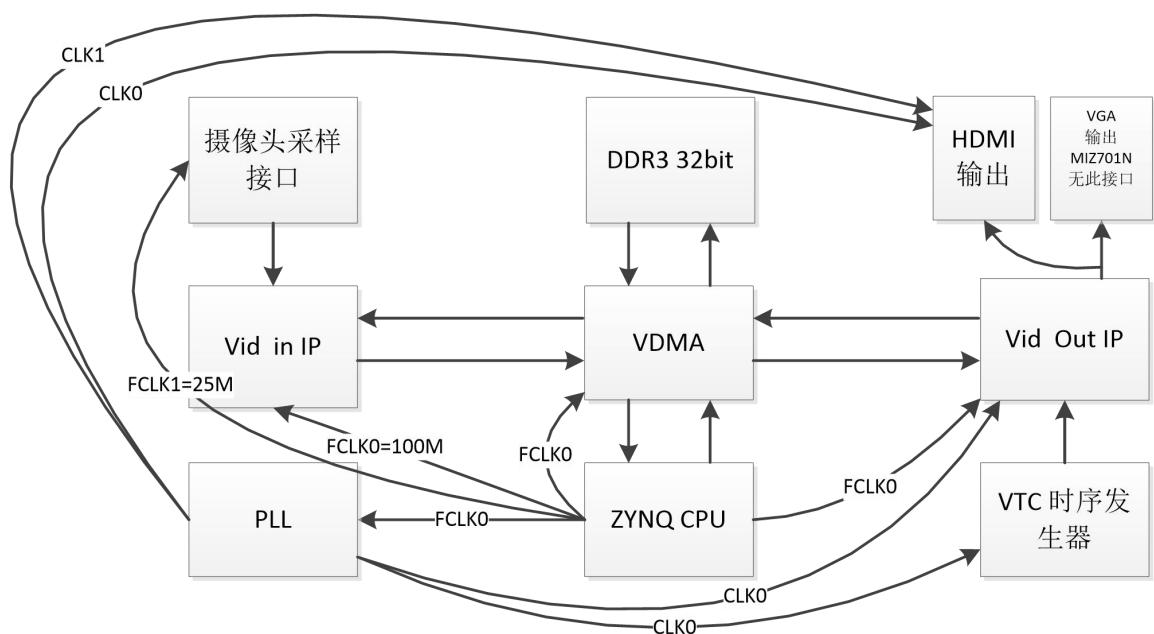
***** 写入 DDR 设置*****
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x30, 0x4); //reset
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x30, 0x8); //genlock

XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xAC, 0x08000000);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xAC+4, 0x0A000000);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xAC+8, 0x09000000);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xA4, 640);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xA8, 0x01000280);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x30, 0x03);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xA0, 480);

```

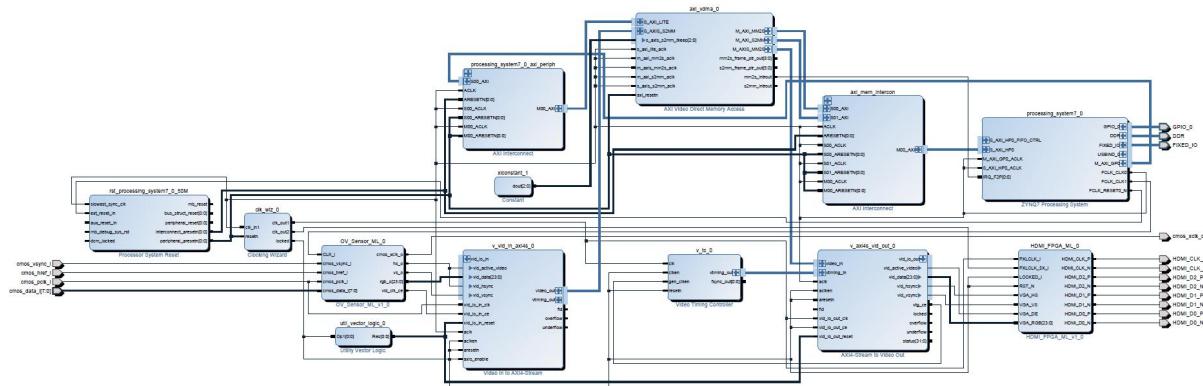
6.5 搭建 VDMA 图像系统

6.5.1 构架方案图



可以看到 VDMA 的图像系统和前面介绍的 DMA 系统相比非常类似。实际上他们都是属于 DMA 系统，只是 VDMA 在配置完成后，可以无需依赖 CPU 可以独立运行，有点类似显卡的功能了。

6.5.2 构 BLOCK 模块化设计方案图



6.6 PS 部分

6.6.1 main 函数

本课程提供了二种方式启动 VDMA，第一种是通过库函数版本，第二种是通过寄存器版本。寄存器版本主要是验证我们对 VDMA 的寄存器掌握情况。库函数具备更强的功能，和可维护性。

表 6-6-1

```
#include "sys_intr.h"
#include "xaxivdma.h"
#include "xaxivdma_i.h"

#define VTC_BASEADDR XPAR_MIZ702_VTG_VGA_0_BASEADDR
#define DDR_BASEADDR      0x00000000
//#define UART_BASEADDR    0xe0001000
#define VDMA_BASEADDR     XPAR_AXI_VDMA_0_BASEADDR
#define H_STRIDE          640
#define H_ACTIVE          640
#define V_ACTIVE          480
#define pi                3.14159265358
#define COUNTS_PER_SECOND (XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ)/64
```

```

#define VIDEO_LENGTH (H_STRIDE*V_ACTIVE)
#define VIDEO_BASEADDR0 DDR_BASEADDR + 0x2000000
#define VIDEO_BASEADDR1 DDR_BASEADDR + 0x3000000
#define VIDEO_BASEADDR2 DDR_BASEADDR + 0x4000000

u32 *BufferPtr[3];

unsigned int srcBuffer = (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x1000000);
int run_triple_frame_buffer(XAxiVdma* InstancePtr, int DeviceId, int hsize,
    int vsize, int buf_base_addr, int number_frame_count,
    int enable_frm_cnt_intr);

int main(void)
{
    u32 Status;

    Miz702_EMIO_init();
    ov7725_init_rgb();

    XAxiVdma InstancePtr;

    xil_printf("Starting the first VDMA \r\n");

    Status = run_triple_frame_buffer(&InstancePtr, 0, 640, 480,
        srcBuffer, 2, 0);
    if (Status != XST_SUCCESS) {
        xil_printf("Transfer of frames failed with error = %d\r\n", Status);
        return XST_FAILURE;
    } else {
        xil_printf("Transfer of frames started \r\n");
    }
    print("TEST PASS\r\n");

//VDMA configureAXI VDMA0
/****************往 DDR 写数据设置*******/
/*Xil_Out32((VDMA_BASEADDR + 0x030), 0x00000003); // enable circular mode
Xil_Out32((VDMA_BASEADDR + 0x0AC), VIDEO_BASEADDR0); // start address
Xil_Out32((VDMA_BASEADDR + 0x0B0), VIDEO_BASEADDR1); // start address
Xil_Out32((VDMA_BASEADDR + 0x0B4), VIDEO_BASEADDR2); // start address
Xil_Out32((VDMA_BASEADDR + 0xA8), (H_STRIDE*4)); // h offset (640 * 4) bytes

```

```

Xil_Out32((VDMA_BASEADDR + 0x0A4), (H_ACTIVE*4)); // h size (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR + 0x0A0), V_ACTIVE);/* // v size (480)
*****从 DDR 读数据设置*****
/*Xil_Out32((VDMA_BASEADDR + 0x000), 0x00000003); // enable circular mode
Xil_Out32((VDMA_BASEADDR + 0x05c), VIDEO_BASEADDR0); // start address
Xil_Out32((VDMA_BASEADDR + 0x060), VIDEO_BASEADDR1); // start address
Xil_Out32((VDMA_BASEADDR + 0x064), VIDEO_BASEADDR2); // start address
Xil_Out32((VDMA_BASEADDR + 0x058), (H_STRIDE*4)); // h offset (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR + 0x054), (H_ACTIVE*4)); // h size (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR + 0x050), V_ACTIVE); // v size (480)
*/
while (1);
return XST_SUCCESS;
}

```

6.6.2 vdma_api.c 函数

XAxivdma_LookupConfig 函数是 XILINX 库函数的标准调用方式，可以获取到硬件的默认配置参数。默认的配置参数保存在 参数表 XAxivdma_ConfigTable 中。

表 6-6-2-1 XAxivdma_LookupConfig

```

/*******************************/
/** 
 * Look up the hardware configuration for a device instance
 *
 * @param DeviceId is the unique device ID of the device to lookup for
 *
 * @return
 * The configuration structure for the device. If the device ID is not found,
 * a NULL pointer is returned.
 *
 */
XAxivdma_Config *XAxivdma_LookupConfig(u16 DeviceId)
{
    extern XAxivdma_Config XAxivdma_ConfigTable[];
    XAxivdma_Config *CfgPtr = NULL;
    int i;

    for (i = 0; i < XPAR_XAXIVDMA_NUM_INSTANCES; i++) {
        if (XAxivdma_ConfigTable[i].DeviceId == DeviceId) {
            CfgPtr = &XAxivdma_ConfigTable[i];
        }
    }
}

```

```
        break;  
    }  
}  
  
return CfgPtr;  
}
```

表 6-6-2-2 XAxiVdma_ConfigTable 参数表

```
XAxiVdma_Config XAxiVdma_ConfigTable[] =  
{  
    {  
        XPAR_AXI_VDMA_0_DEVICE_ID,  
        XPAR_AXI_VDMA_0_BASEADDR,  
        XPAR_AXI_VDMA_0_NUM_FSTORES,  
        XPAR_AXI_VDMA_0_INCLUDE_MM2S,  
        XPAR_AXI_VDMA_0_INCLUDE_MM2S_DRE,  
        XPAR_AXI_VDMA_0_M_AXI_MM2S_DATA_WIDTH,  
        XPAR_AXI_VDMA_0_INCLUDE_S2MM,  
        XPAR_AXI_VDMA_0_INCLUDE_S2MM_DRE,  
        XPAR_AXI_VDMA_0_M_AXI_S2MM_DATA_WIDTH,  
        XPAR_AXI_VDMA_0_INCLUDE_SG,  
        XPAR_AXI_VDMA_0_ENABLE_VIDPRMTR_READS,  
        XPAR_AXI_VDMA_0_USE_FSYNC,  
        XPAR_AXI_VDMA_0_FLUSH_ON_FSYNC,  
        XPAR_AXI_VDMA_0_MM2S_LINEBUFFER_DEPTH,  
        XPAR_AXI_VDMA_0_S2MM_LINEBUFFER_DEPTH,  
        XPAR_AXI_VDMA_0_MM2S_GENLOCK_MODE,  
        XPAR_AXI_VDMA_0_S2MM_GENLOCK_MODE,  
        XPAR_AXI_VDMA_0_INCLUDE_INTERNAL_GENLOCK,  
        XPAR_AXI_VDMA_0_S2MM_SOF_ENABLE,  
        XPAR_AXI_VDMA_0_M_AXIS_MM2S_TDATA_WIDTH,  
        XPAR_AXI_VDMA_0_S_AXIS_S2MM_TDATA_WIDTH,  
        XPAR_AXI_VDMA_0_ENABLE_DEBUG_INFO_1,  
        XPAR_AXI_VDMA_0_ENABLE_DEBUG_INFO_5,  
        XPAR_AXI_VDMA_0_ENABLE_DEBUG_INFO_6,  
        XPAR_AXI_VDMA_0_ENABLE_DEBUG_INFO_7,  
        XPAR_AXI_VDMA_0_ENABLE_DEBUG_INFO_9,  
        XPAR_AXI_VDMA_0_ENABLE_DEBUG_INFO_13,  
        XPAR_AXI_VDMA_0_ENABLE_DEBUG_INFO_14,  
        XPAR_AXI_VDMA_0_ENABLE_DEBUG_INFO_15,  
        XPAR_AXI_VDMA_0_ENABLE_DEBUG_ALL,  
        XPAR_AXI_VDMA_0_ADDR_WIDTH  
    }  
}
```

```
};
```

WriteSetup VDMA 写通道设置，主要设置分辨率，延迟参数，开启 CircularBuf 模式，使能 Gen-Lock。更底层的分析读者可以顺藤摸瓜下去。

表 6-6-2-3 WriteSetup

```
*****  
***/  
/**  
  
* This function sets up the write channel  
  
*  
* @param  dma_context is the context pointer to the VDMA engine..  
*  
* @return XST_SUCCESS if the setup is successful, XST_FAILURE otherwise.  
*  
* @note    None.  
  
*****  
**/  
static int WriteSetup(vdma_handle *vdma_context)  
{  
    int Index;  
    u32 Addr;  
    int Status;  
  
    vdma_context->WriteCfg.VertSizeInput = vdma_context->vsize;  
    vdma_context->WriteCfg.HoriSizeInput = vdma_context->hsize;  
  
    vdma_context->WriteCfg.Stride = vdma_context->hsize;  
    vdma_context->WriteCfg.FrameDelay = 0; /* This example does not test frame delay  
*/  
  
    vdma_context->WriteCfg.EnableCircularBuf = 1;  
    vdma_context->WriteCfg.EnableSync = 1; /* Gen-Lock */  
  
    vdma_context->WriteCfg.PointNum = 0;  
    vdma_context->WriteCfg.EnableFrameCounter = 0; /* Endless transfers */  
  
    vdma_context->WriteCfg.FixedFrameStoreAddr = 0; /* We are not doing parking */  
    /* Configure the VDMA is per fixed configuration, This configuration  
     * is being used by majority of customers. Expert users can play around  
     * with this if they have different configurations  
     */
```

```
Status = XAxiVdma_DmaConfig(vdma_context->InstancePtr, XAXIVDMA_WRITE,
&vdma_context->WriteCfg);
if (Status != XST_SUCCESS) {
    xil_printf(
        "Write channel config failed %d\r\n", Status);

    return Status;
}

/* Initialize buffer addresses
 *
 * Use physical addresses
 */
Addr = vdma_context->buffer_address;
/* If Debug mode is enabled write frame is shifted 3 Frames
 * store ahead to compare read and write frames
 */
#if DEBUG_MODE
    Addr = Addr + vdma_context->InstancePtr->MaxNumFrames * \
            (vdma_context->WriteCfg.Stride * vdma_context->vsize);
#endif

for(Index = 0; Index < vdma_context->InstancePtr->MaxNumFrames; Index++) {
    vdma_context->WriteCfg.FrameStoreStartAddr[Index] = Addr;
#if DEBUG_MODE
    xil_printf("Write Buffer %d address: 0x%0x \r\n",Index,Addr);
#endif

    Addr += (vdma_context->hsize * vdma_context->vsize);
}

/* Set the buffer addresses for transfer in the DMA engine */
Status = XAxiVdma_DmaSetBufferAddr(vdma_context->InstancePtr,
XAXIVDMA_WRITE,
vdma_context->WriteCfg.FrameStoreStartAddr);
if (Status != XST_SUCCESS) {
    xil_printf("Write channel set buffer address failed %d\r\n",
Status);
    return XST_FAILURE;
}

/* Clear data buffer
*/
```

```

#if DEBUG_MODE
    memset((void *)vdma_context->buffer_address, 0,
           vdma_context->ReadCfg.Stride * vdma_context->ReadCfg.VertSizeInput *
vdma_context->InstancePtr->MaxNumFrames);
#endif
    return XST_SUCCESS;
}

```

ReadSetup VDMA 读通道设置，主要设置分辨率，这里的延迟参数 1，否则图像会有卡顿，开启 CircularBuf 模式，使能 Gen-Lock。更底层的分析读者可以顺藤摸瓜下去。

表 6-6-2-4 ReadSetup

```

*****
*/
/**
*
*
* This function sets up the read channel
*
* @param vdma_context is the context pointer to the VDMA engine.
*
* @return XST_SUCCESS if the setup is successful, XST_FAILURE otherwise.
*
* @note None.
*
*****
*/
static int ReadSetup(vdma_handle *vdma_context)
{
    int Index;
    u32 Addr;
    int Status;

    vdma_context->ReadCfg.VertSizeInput = vdma_context->vsize;
    vdma_context->ReadCfg.HoriSizeInput = vdma_context->hsize;

    vdma_context->ReadCfg.Stride = vdma_context->hsize;
    vdma_context->ReadCfg.FrameDelay = 0; /* This example does not test frame delay
*/
    vdma_context->ReadCfg.EnableCircularBuf = 1;
    vdma_context->ReadCfg.EnableSync = 1; /* Gen-Lock */

    vdma_context->ReadCfg.PointNum = 0;
}

```

```
vdma_context->ReadCfg.EnableFrameCounter = 0; /* Endless transfers */

vdma_context->ReadCfg.FixedFrameStoreAddr = 0; /* We are not doing parking */
/* Configure the VDMA is per fixed configuration, This configuration is being used by
majority
 * of customer. Expert users can play around with this if they have different
configurations */

Status = XAxiVdma_DmaConfig(vdma_context->InstancePtr, XAXIVDMA_READ,
&vdma_context->ReadCfg);
if (Status != XST_SUCCESS) {
    xil_printf("Read channel config failed %d\r\n", Status);
    return XST_FAILURE;
}

/* Initialize buffer addresses
 *
 * These addresses are physical addresses
 */
Addr = vdma_context->buffer_address;

for(Index = 0; Index < vdma_context->InstancePtr->MaxNumFrames; Index++) {
    vdma_context->ReadCfg.FrameStoreStartAddr[Index] = Addr;

    /* Initializing the buffer in case of Debug mode */

#if DEBUG_MODE
{
    u32 i;
    u8 *src;
    u32 total_pixel = vdma_context->ReadCfg.Stride * vdma_context->vsize;
    src = (unsigned char *)Addr;
    xil_printf("Read Buffer %d address: 0x%lx \r\n",Index,Addr);
    for(i=0;i<total_pixel;i++)
    {
        src[i] = i & 0xFF;
    }
}
#endif
    Addr +=  vdma_context->hsize * vdma_context->vsize;
}

/* Set the buffer addresses for transfer in the DMA engine
```

```

* The buffer addresses are physical addresses
*/
Status = XAxiVdma_DmaSetBufferAddr(vdma_context->InstancePtr,
XAXIVDMA_READ,
    vdma_context->ReadCfg.FrameStoreStartAddr);
if (Status != XST_SUCCESS) {
    xil_printf(
        "Read channel set buffer address failed %d\r\n", Status);

    return XST_FAILURE;
}

return XST_SUCCESS;
}

```

StartTransfer 启动 VDMA 读写通道

表 6-6-2-5 StartTransfer

```

*****
*/
/**
*
*
* This function starts the DMA transfers. Since the DMA engine is operating
* in circular buffer mode, video frames will be transferred continuously.
*
* @param InstancePtr points to the DMA engine instance
*
* @return
*      - XST_SUCCESS if both read and write start successfully
*      - XST_FAILURE if one or both directions cannot be started
*
* @note None.
*
*****
*/
static int StartTransfer(XAxiVdma *InstancePtr)
{
    int Status;
    /* Start the write channel of VDMA */
    Status = XAxiVdma_DmaStart(InstancePtr, XAXIVDMA_WRITE);
    if (Status != XST_SUCCESS) {
        xil_printf("Start Write transfer failed %d\r\n", Status);

        return XST_FAILURE;
    }
}

```

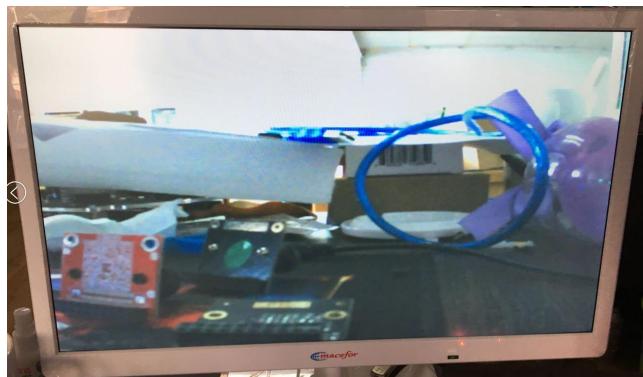
```
}

/* Start the Read channel of VDMA */
Status = XAxiVdma_DmaStart(InstancePtr, XAXIVDMA_READ);
if (Status != XST_SUCCESS) {
    xil_printf("Start read transfer failed %d\r\n", Status);

    return XST_FAILURE;
}

return XST_SUCCESS;
}
```

6.7 测试结果



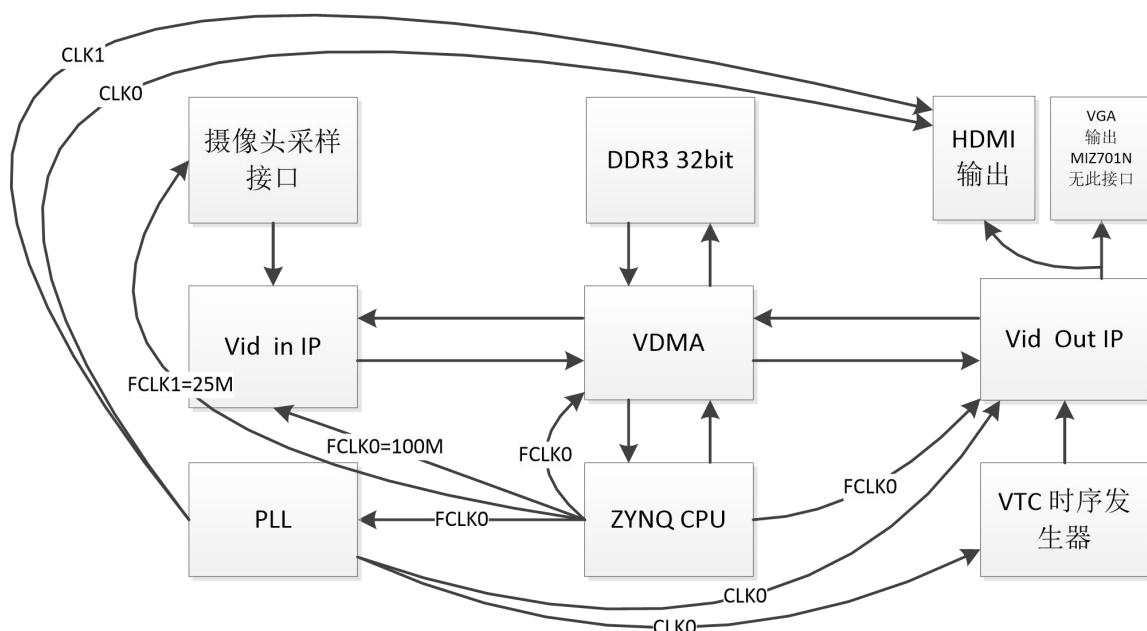
S03_CH07_AXI_VDMA_OV5640 摄像头采集系统

7.1 概述

本章内容和《S03_CH06_AXI_VDMA_OV7725 摄像头采集系统》只是摄像头采用的分辨率不同，其他原理都一样，由于在《S03_CH06_AXI_VDMA_OV7725 摄像头采集系统》中详细介绍了 VDMA 的原理，如果读者只是购买了 OV5640，可以回到《S03_CH06_AXI_VDMA_OV7725 摄像头采集系统》仔细阅读 VDMA 的基础知识。

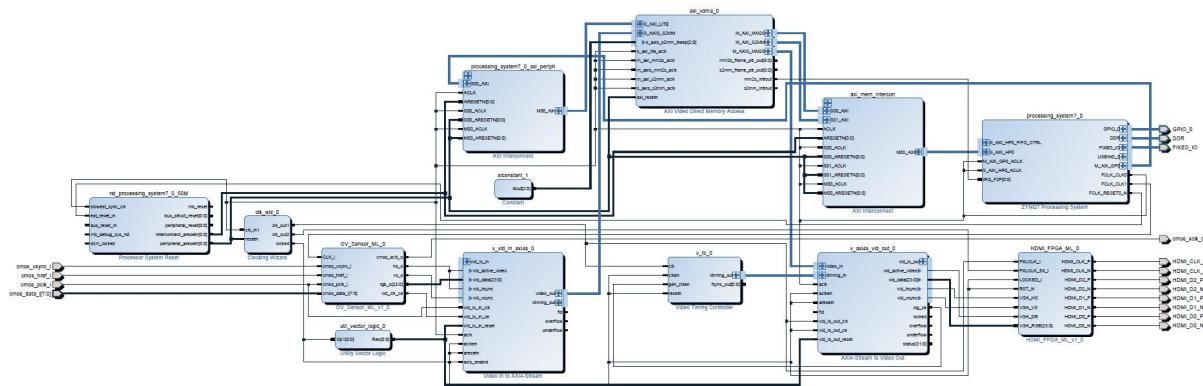
7.2 搭建 VDMA 图像系统

7.2.1 构架方案图



可以看到 VDMA 的图像系统和前面介绍的 DMA 系统相比非常类似。实际上他们都是属于 DMA 系统，只是 VDMA 在配置完成后，可以无需依赖 CPU 可以独立运行，有点类似显卡的功能了。

7.2.2 构 BLOCK 模块化设计方案图



7.3 PS 部分

本课程提供了二种方式启动 VDMA，第一种是通过库函数版本，第二种是通过寄存器版本。寄存器版本主要是验证我们对 VDMA 的寄存器掌握情况。库函数具备更强的功能，和 OV7725 相比，这里的分辨率设置为 1280X720

表 6-6-1

```
#include "sys_intr.h"
#include "xaxivdma.h"
#include "xaxivdma_i.h"

#define VTC_BASEADDR XPAR_MIZ702_VTG_VGA_0_BASEADDR
#define DDR_BASEADDR 0x00000000
#define UART_BASEADDR 0xe0001000
#define VDMA_BASEADDR XPAR_AXI_VDMA_0_BASEADDR
#define H_STRIDE 1280
#define H_ACTIVE 1280
#define V_ACTIVE 720
#define COUNTS_PER_SECOND (XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ)/64

#define VIDEO_LENGTH (H_STRIDE*V_ACTIVE)
#define VIDEO_BASEADDR0 DDR_BASEADDR + 0x2000000
#define VIDEO_BASEADDR1 DDR_BASEADDR + 0x3000000
#define VIDEO_BASEADDR2 DDR_BASEADDR + 0x4000000

u32 *BufferPtr[3];

unsigned int srcBuffer = (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x1000000);
```

```

int run_triple_frame_buffer(XAxiVdma* InstancePtr, int DeviceId, int hsize,
    int vsize, int buf_base_addr, int number_frame_count,
    int enable_frm_cnt_intr);

int main(void)
{
    u32 Status;

    Miz702_EMIO_init();
    ov7725_init_rgb();

    XAxiVdma InstancePtr;

    xil_printf("Starting the first VDMA \n\r");

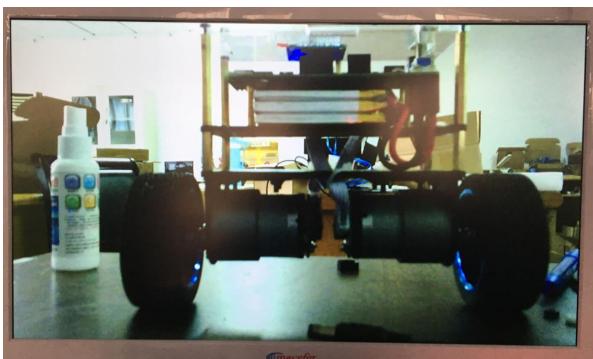
    Status = run_triple_frame_buffer(&InstancePtr, 0, 1280, 720,
        srcBuffer, 2, 0);
    if (Status != XST_SUCCESS) {
        xil_printf("Transfer of frames failed with error = %d\n",Status);
        return XST_FAILURE;
    } else {
        xil_printf("Transfer of frames started \r\n");
    }
    print("TEST PASS\r\n");

//VDMA configureAXI VDMA0
/****************往 DDR 写数据设置*******/
/*Xil_Out32((VDMA_BASEADDR + 0x030), 0x00000003);// enable circular mode
Xil_Out32((VDMA_BASEADDR + 0x0AC), VIDEO_BASEADDR0); // start address
Xil_Out32((VDMA_BASEADDR + 0x0B0), VIDEO_BASEADDR1); // start address
Xil_Out32((VDMA_BASEADDR + 0x0B4), VIDEO_BASEADDR2); // start address
Xil_Out32((VDMA_BASEADDR + 0x0A8), (H_STRIDE*4)); // h offset (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR + 0x0A4), (H_ACTIVE*4)); // h size (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR + 0x0A0), V_ACTIVE);*/ // v size (480)
/****************从 DDR 读数据设置*******/
/*Xil_Out32((VDMA_BASEADDR + 0x000), 0x00000003); // enable circular mode
Xil_Out32((VDMA_BASEADDR + 0x05c), VIDEO_BASEADDR0); // start address
Xil_Out32((VDMA_BASEADDR + 0x060), VIDEO_BASEADDR1); // start address
Xil_Out32((VDMA_BASEADDR + 0x064), VIDEO_BASEADDR2); // start address
Xil_Out32((VDMA_BASEADDR + 0x058), (H_STRIDE*4)); // h offset (640 * 4) bytes

```

```
Xil_Out32((VDMA_BASEADDR + 0x054), (H_ACTIVE*4));      // h size (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR + 0x050), V_ACTIVE);           // v size (480)
*/
while (1);
return XST_SUCCESS;
}
```

7.4 测试结果



S03_CH08_DMA_LWIP 以太网传输

8.1 概述

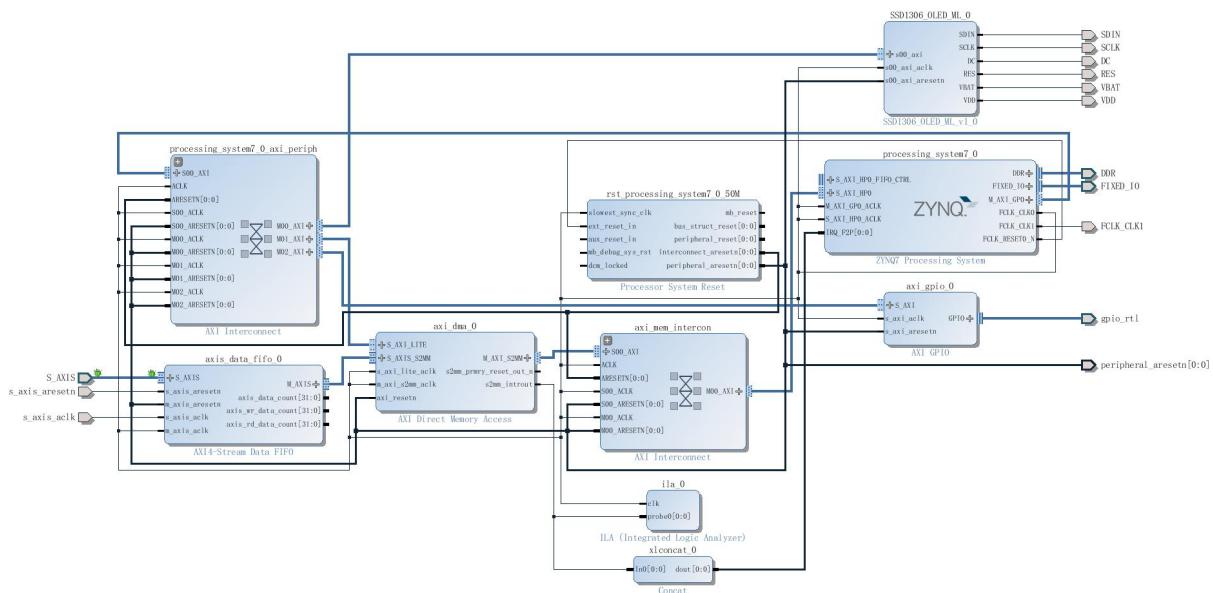
本例程详细创建过程和本季课程第一课《S03_CH01_AXI_DMA_LOOP 环路测试》非常类似，因此如果读者不清楚如何创建工作，请仔细阅读本季第一课时。

本例程的基本原理如下。

PS 通过 AXI GPIO IP 核启动 PL 不间断循环构造 16bit 位宽的 0~1023 的数据，利用 AXI DMA IP 核，通过 PS 的 Slave AXI GP 接口传输至 PS DDR3 的乒乓缓存中。PL 每发完一次 0~1023，AXI DMA IP 核便会产生一个中断信号，PS 得到中断信号后将 DDR3 缓存的数据通过乒乓操作的方式由 TCP 协议发送至 PC 机。

8.2 搭建硬件系统

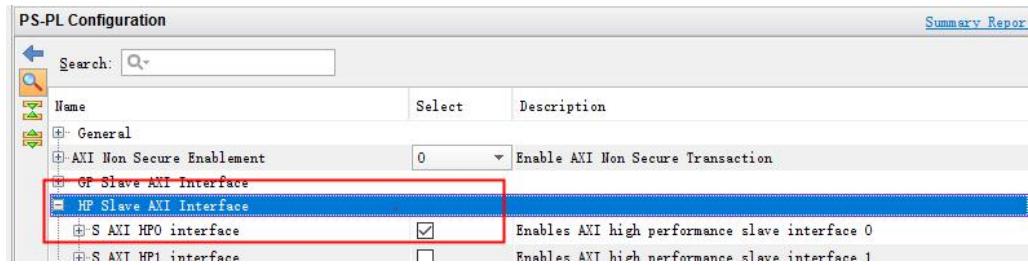
8.2.1 系统构架



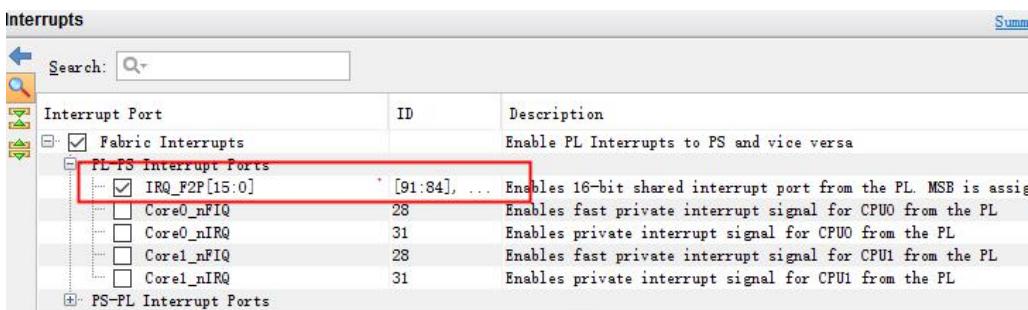
这个系统实际上就是在前面章节 DMA 的基础上去掉 FPGA 读 DMA 通道，只有 FPGA 往 DMA 写输入数据，当 DMA 接收中断产生后，在通过 LWIP 协议，把数据通过网口发送出去。网口是接在 PS 的 ARM 端口的因此，ARM 部分也是要把网口配置好。如下图所示。

8.2.1 启用 HP 接口

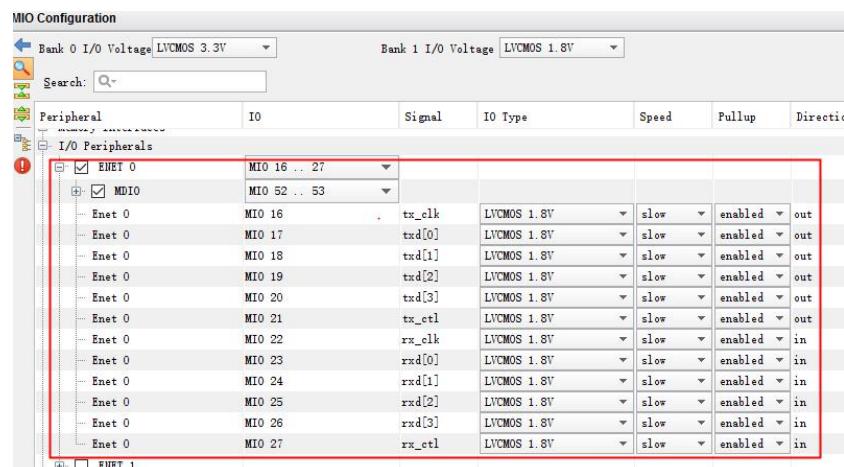
双击 ZYNQ 的 IP 之后启动 HP 接口，这里只要用到 1 个 HP 接口通道



8.2.2 启用 PL 到 PS 的中断资源



8.2.3 启动 PS 部分的以太网接口



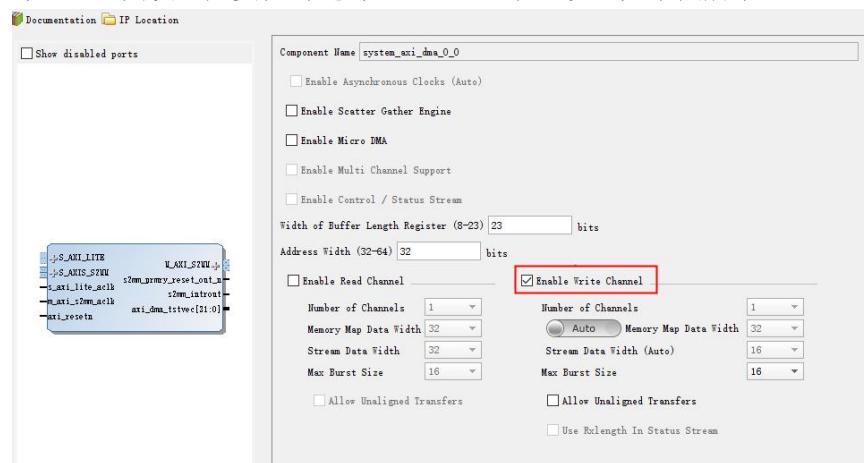
8.2.4 时钟的设置

将 FCLK_CLK0 和 FCLK_CLK1 均设为 100Mhz, 其中 CLK1 为 PL 构造数据部分逻辑的时钟源, 在实际应用中可自由调节时钟频率, 故与 CLK0 分开使用。设置如下图所示。

Component	Clock Source	Requested Frequenc...	Actual Frequenc...	Range (MHz)
Processor/Memory Clocks				
IO Peripheral Clocks				
PL Fabric Clocks				
FCLK_CLK0	IO PLL	100.000000	100.000000	0.100000 : 250.000000
FCLK_CLK1	IO PLL	100.000000	100.000000	0.100000 : 250.000000
FCLK_CLK2	IO PLL	50.000000	50.000000	0.100000 : 250.000000
FCLK_CLK3	IO PLL	50	50.000000	0.100000 : 250.000000
System Debug Clocks				

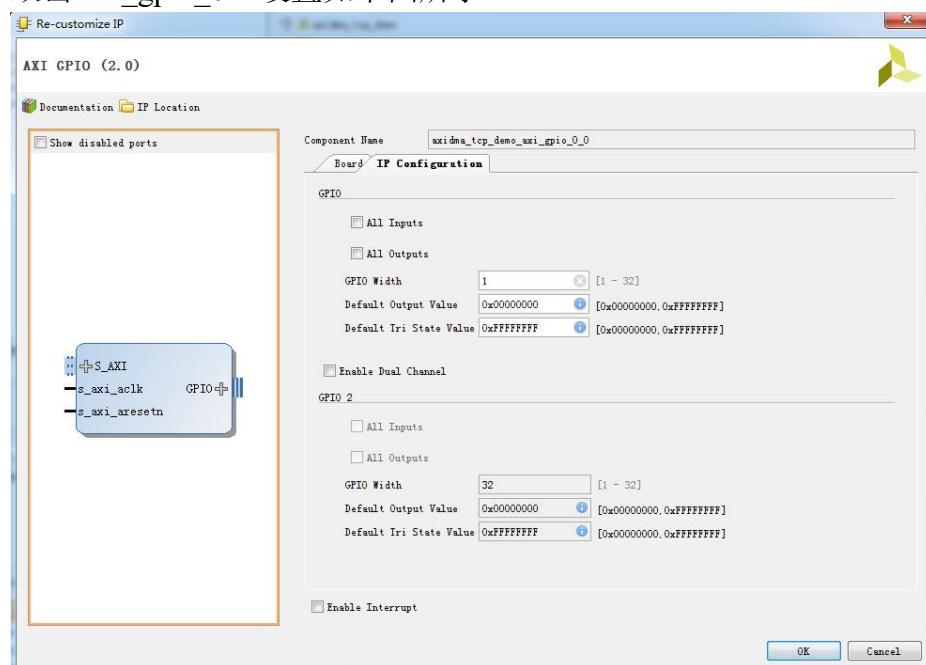
8.2.5 DMA IP 配置

由于只用到了写通道，因次，只要把勾选写 DMA 通道既可以，如下图所示：



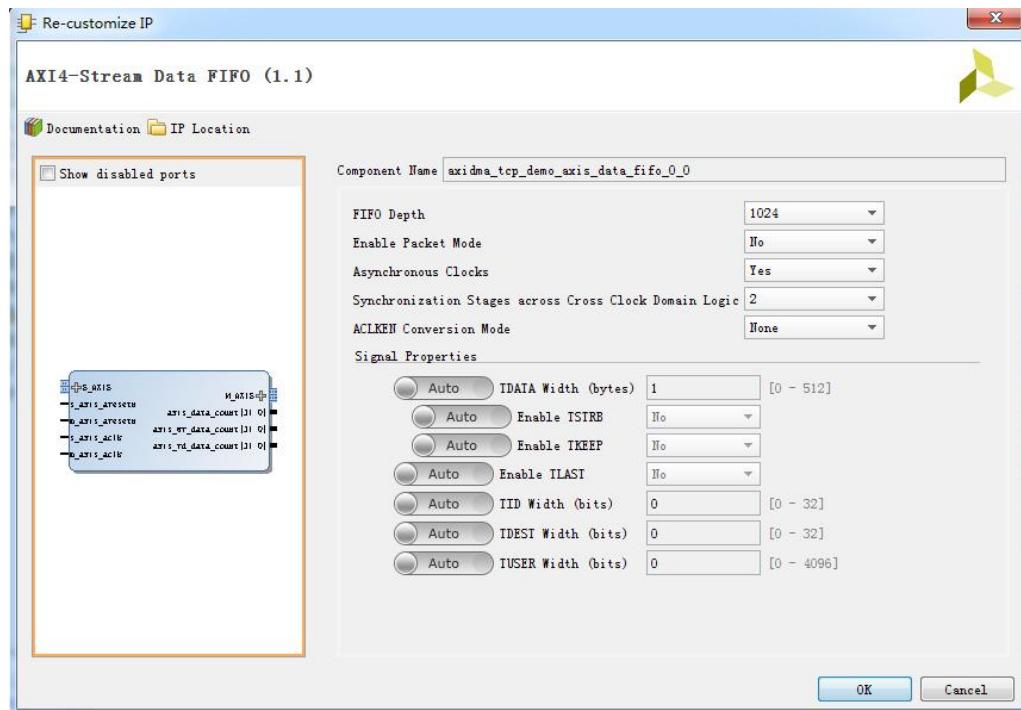
8.2.6 GPIO 的配置

双击 axi_gpio_0。设置如下图所示

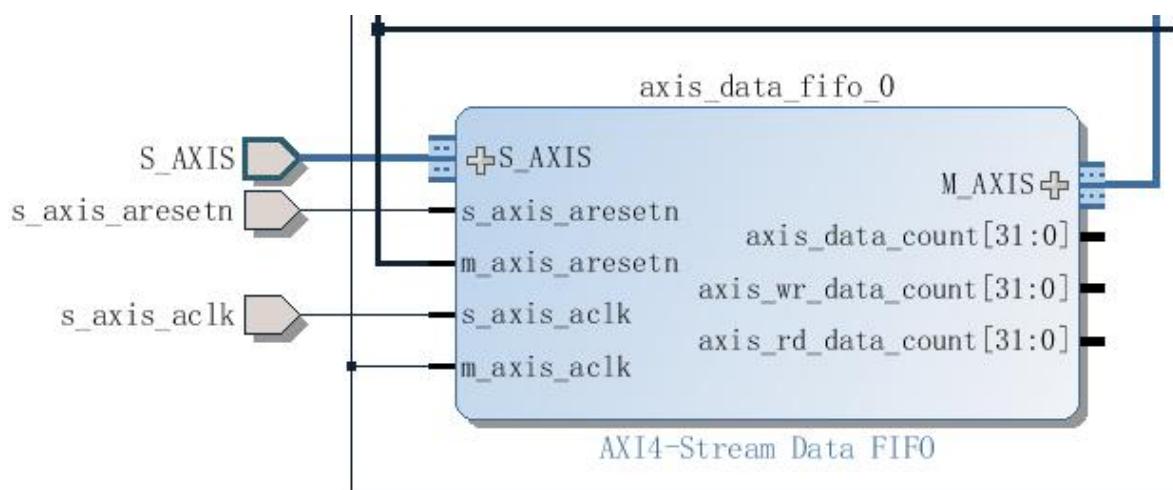


8.2.7 配置 axi_data_fifo_0

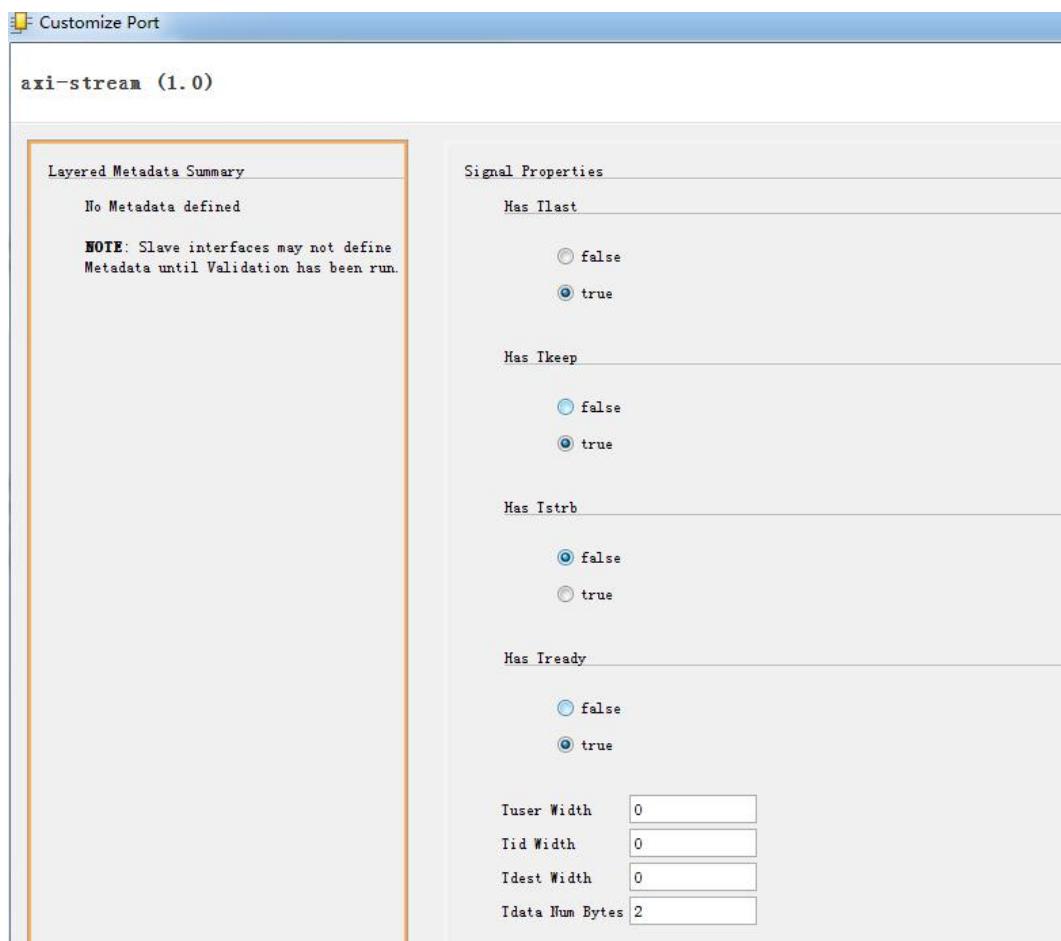
双击 axis_data_fifo_0，设置如下图所示。fifo 在本例程中作为 axi_dma_0 的 S_AXIS_S2MM 接口所在的 FCLK0 时钟域与外部数据生成逻辑 stream 接口所在的 FCLK1 时钟域之间的转换媒介。



8.2.8 设置 S_AXIS 接口

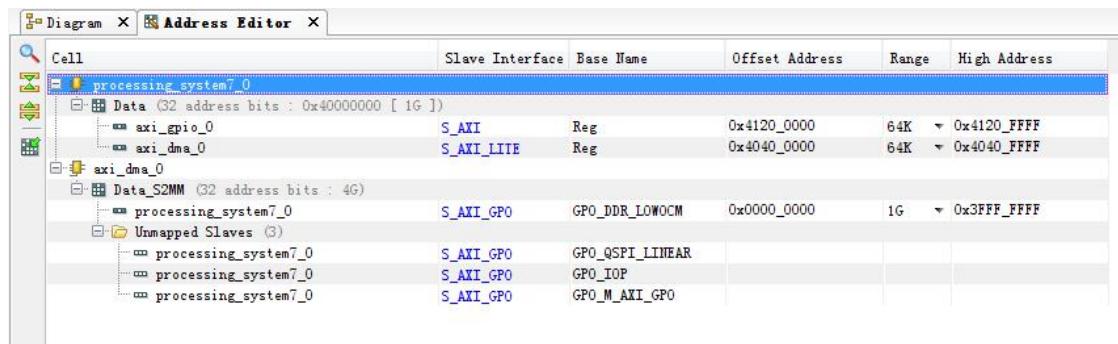


双击 S_AXIS 端口，进行如下图设置



8.2.9 地址空间映射

打开 Address Editor，设置如下图所示



8.3 FPGA 的发送代码

表 8-3-1

always@(posedge FCLK_CLK1)

```
begin
    if(!peripheral_aresetn) begin //系统复位
        S_AXIS_tvalid <= 1'b0;
        S_AXIS_tdata <= 32'd0;
        S_AXIS_tlast <= 1'b0;
        state <=0;
    end
    else begin
        case(state) //状态机
            0: begin
                if(gpio_tri_o_0&& S_AXIS_tready) begin //当 FIFO 非满的时候
                    S_AXIS_tvalid <= 1'b1;//设置写 FIFO 数据有效标志为 1
                    state <= 1;//转入状态 1
                end
                else begin
                    S_AXIS_tvalid <= 1'b0;
                    state <= 0;
                end
            end
            1:begin
                if(S_AXIS_tready) begin //当 FIFO 非满
                    S_AXIS_tdata <= S_AXIS_tdata + 1'b1;
                    if(S_AXIS_tdata == 16'd1022) begin //从 0-1022 一共写入 1023 个字节
                        S_AXIS_tlast <= 1'b1;//发送最后一个数据
                        state <= 2;
                    end
                    else begin
                        S_AXIS_tlast <= 1'b0;
                        state <= 1;
                    end
                end
                else begin
                    S_AXIS_tdata <= S_AXIS_tdata;
                    state <= 1;
                end
            end
            2:begin
                if(!S_AXIS_tready) begin //如果 FIFO 满, 等待
                    S_AXIS_tvalid <= 1'b1;
                    S_AXIS_tlast <= 1'b1;
                    S_AXIS_tdata <= S_AXIS_tdata;
                    state <= 2;
                end
            end
        endcase
    end
end
```

```

else begin //写入结束
    S_AXIS_tvalid <= 1'b0;
    S_AXIS_tlast <= 1'b0;
    S_AXIS_tdata <= 16'd0;
    state <= 0;
end
end
default: state <=0;
endcase
end
end

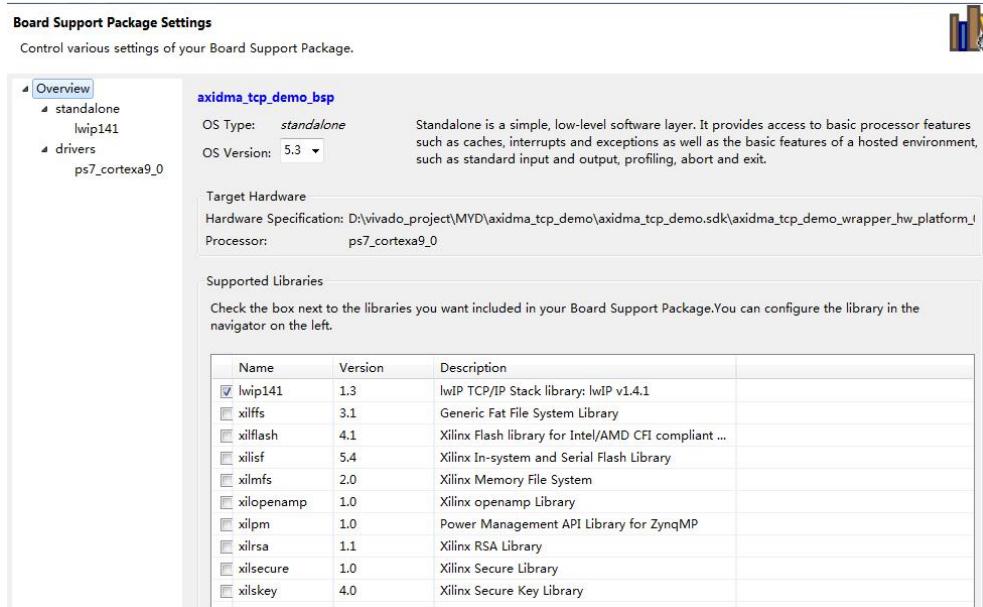
```

发送代码和本季第一课程的代码一样，每次发送 1024X16bit 的数据到通过 DMA 到 DDR 内存。代码比较简单，如果 verilog 语法基础不好的，无法对发送时序精确把我的 FPGA 初学者可以多思考思考，如果还是无法理解可以找我们 FPGA 技术支持。

8.4 PS 部分 BSP 设置

8.4.1 SDK 工程 BSP 设置

进入 sdk 后，新建空工程，命名为 AXI_DMA_PL_PS_Test，同时创建相应的 bsp。修改 AXI_DMA_PL_PS_Test_bsp 的设置，使能 lwip 1.4.1 函数库。如下图所示。



8.4.2 lwip 函数库设置

本例程使用 RAW API，即函数调用不依赖操作系统。传输效率也比 SOCKET API 高，(具体可参考 xapp1026)。将 use_axieth_on_zynq 和 use_emaclite_on_zynq 设为 0。如下图所示。

Configuration for library: lwip141

Name	Value	Default	Type	Description
api_mode	RAW_API	RAW_API	enum	Mode of operation for lwIP (I
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axi
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures em

修改 lwip_memory_options 设置，将 mem_size , memp_n_pbuf , memp_n_tcp_pcb , memp_n_tcp_seg 这 4 个参数值设大，这样会提高 TCP 传输效率。如下图所示。

lwip_memory_options				
				Options controlling lwIP mem
mem_size	524288	131072	integer	Size of the heap memory (byt
memp_n_pbuf	2048	16	integer	Number of memp struct pbuf
memp_n_sys_timeout	8	8	integer	Number of simultaneously ac
memp_n_tcp_pcb	1024	32	integer	Number of active TCP PCBs. (
memp_n_tcp_pcb_listen	8	8	integer	Number of listening TCP con
memp_n_tcp_seg	1024	256	integer	Number of simultaneously qu
memp_n_udp_pcb	4	4	integer	Number of active UDP PCBs.
memp_num_api_msg	16	16	integer	Number of api msg structure
memp_num_netbuf	8	8	integer	Number of struct netbufs (so
memp_num_netconn	16	16	integer	Number of struct netconns (s
memp_num_tcip_msg	64	64	integer	Number of tcip msg structu

修改 pbuf_options 设置，将 pbuf_pool_size 设大，增加可用的 pbuf 数量，这样同样会提高 TCP 传输效率。如下图所示。

pbuf_options				
	true	true	boolean	Pbuf Options
pbuf_link_hlen	16	16	integer	Number of bytes that should
pbuf_pool_bufsize	1700	1700	integer	Size of each pbuf in pbuf po
pbuf_pool_size	4096	256	integer	Number of buffers in pbuf po

修改 tcp_options 设置，将 tcp_snd_buf, tcp_wnd 参数设大，这样同样会提高 TCP 传输效率。如下图所示。

tcp_options				
	true	true	boolean	Is TCP required ?
lwip_tcp	true	true	boolean	Is TCP required ?
tcp_maxrtx	12	12	integer	TCP Maximum retransmissio
tcp_mss	1460	1460	integer	TCP Maximum segment size (
tcp_queue_ooseq	1	1	integer	Should TCP queue segments
tcp_snd_buf	65535	8192	integer	TCP sender buffer space (byt
tcp_synmaxrtx	4	4	integer	TCP Maximum SYN retransmi
tcp_ttl	255	255	integer	TCP TTL value
tcp_wnd	65535	2048	integer	TCP Window (bytes)

修改 temac_adapter_options 设置，将 n_rx_descriptors 和 n_tx_descriptors 参数设大。这样可以提高 zynq 内部 emac dma 的数据迁移效率，同样能提高 TCP 传输效率。如下图所示。

temac_adapter_options				
	true	true	boolean	Settings for xps_ll-temac/Ax
emac_number	0	0	integer	Zynq Ethernet Interface num
n_rx_coalesce	1	1	integer	Setting for RX Interrupt coa
n_rx_descriptors	256	64	integer	Number of RX Buffer Descr
n_tx_coalesce	1	1	integer	Setting for TX Interrupt coa
n_tx_descriptors	256	64	integer	Number of TX Buffer Descr
phy_link_speed	CONFIG_LINKSPEED_A...	CONFIG_LINKSPEED_A...	enum	link speed as negotiated by
tcp_ip_rx_checksum_offload	false	false	boolean	Offload TCP and IP Receive
tcp_ip_tx_checksum_offload	false	false	boolean	Offload TCP and IP Transm
tcp_rx_checksum_offload	false	false	boolean	Offload TCP Receive checks
tcp_tx_checksum_offload	false	false	boolean	Offload TCP Transmit check
temac_use_jumbo_frames	false	false	boolean	use jumbo frames

其余选项的参数默认即可，不用修改。点击 OK，重建 bsp。

8.5 PS 部分程序分析

8.5.1 main.c 分析

main 函数的主要流程为：

- 1): 初始化并配置 PL 侧的 AXI GPIO
- 2): 初始化并配置 PL 侧的 AXI DMA
- 3): 初始化并配置 PS 的中断控制器
- 4): 初始化 lwip 协议栈和 PS 的以太网控制器
- 5): 配置 TCP 传输所需的相关参数，并与服务器建立 TCP 连接
- 6): 通过 AXI GPIO 启动 PL 进行数据生成和传输
- 7): 通过 AXI DMA 接收 PL 传输的数据，通过 TCP 发送至 PC 机，并不断循环该过程。

表 8-5-1

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 * axi dma test
 *
 */

#include "dma_intr.h"
#include "timer_intr.h"
#include "sys_intr.h"
#include "xgpio.h"
#include "OLED.h"

#include "lwip/err.h"
#include "lwip/tcp.h"
#include "lwipopts.h"
#include "netif/xadapter.h"
#include "lwipopts.h"

static XScuGic Intc; //GIC
static XScuTimer Timer; //timer
XAxiDma AxiDma;
u16 *RxBufferPtr[2]; /* ping pong buffers*/
```

```
volatile u32 RX_success;
volatile u32 TX_success;

volatile u32 RX_ready=1;
volatile u32 TX_ready=1;

#define TIMER_LOAD_VALUE      XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ / 8 //0.25S

extern void send_received_data(void);
extern unsigned tcp_client_connected;

char oled_str[17]="";
static XGpio Gpio;

#define AXI_GPIO_DEV_ID      XPAR_AXI_GPIO_0_DEVICE_ID

int init_intr_sys(void)
{
    DMA_Intr_Init(&AxiDma,0);//initial interrupt system
    Timer_init(&Timer,TIMER_LOAD_VALUE,0);
    Init_Intr_System(&Intc); // initial DMA interrupt system
    Setup_Intr_Exception(&Intc);
    DMA_Setup_Intr_System(&Intc,&AxiDma,0,RX_INTR_ID);//setup dma interrupt system
    Timer_Setup_Intr_System(&Intc,&Timer,TIMER_IRPT_INTR);
    DMA_Intr_Enable(&Intc,&AxiDma);

}

int main(void)
{
    int Status;
    struct netif *netif, server_netif;
    struct ip_addr ipaddr, netmask, gw;

    /* the mac address of the board. this should be unique per board */
    unsigned char mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };

    /* Initialize the ping pong buffers for the data received from axidma */
    RxBufferPtr[0] = (u16 *)RX_BUFFER0_BASE;
    RxBufferPtr[1] = (u16 *)RX_BUFFER1_BASE;
```

```
XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);
XGpio_SetDataDirection(&Gpio, 1, 0);
init_intr_sys();
TcpTmrFlag = 0;

netif = &server_netif;

IP4_ADDR(&ipaddr, 192, 168, 1, 10);
IP4_ADDR(&netmask, 255, 255, 255, 0);
IP4_ADDR(&gw, 192, 168, 1, 1);

/*lwip library init*/
lwip_init();
/* Add network interface to the netif_list, and set it as default */
if (!xemac_add(netif, &ipaddr, &netmask, &gw, mac_ethernet_address,
XPAR_XEMACPS_0_BASEADDR)) {
    xil_printf("Error adding N/W interface\r\n");
    return -1;
}
netif_set_default(netif);

/* specify that the network if is up */
netif_set_up(netif);

/* initialize tcp pcb */
tcp_send_init();

XGpio_DiscreteWrite(&Gpio, 1, 1);
oled_fresh_en(); // enable oled
Timer_start(&Timer);

while (1)
{
    /* call tcp timer every 250ms */
    if(TcpTmrFlag)
    {
        tcp_tmr();
        TcpTmrFlag = 0;
    }
    /*receive input packet from emac*/
    xemacif_input(netif); // 将 MAC 队列里的 packets 传输到你的 LwIP/IP stack 里
    /* if connected to the server, start receive data from PL through axidma, then transmit the data to
```

```

the PC software by TCP*/
    if(tcp_client_connected)
        send_received_data();
    }
    return 0;
}

```

8.5.2 AXI DMA 数据传输过程

例程中 axi dma 采用了 simple transfer 方式，通过 XAxiDma_SimpleTransfer 函数完成。每次 dma 传输都需要 PS 主动发起，PS 通过 AXI 总线配置 PL 侧 axi dma 内部寄存器，发起一次 dma 传输。dma 传输发起后，axi dma 开始通过 S_AXIS_S2MM 接口接收数据，当其中的 tlast 信号被拉高，则代表当次传输所需要的数据发送完毕，当该次 dma 传输结束，axi dma 通过 s2mm_introut 产生中断信号，触发 PS 中断控制器产生中断，PS 通过中断服务函数 Dma_RxIsr 清除 axi dma 的中断状态，在 DM 中断函数中，拉高 dma 完成指示信号 packet_trans_done，一次完整的 simple transfer 的 dma 传输结束。下表为 dma 中断接收函数，接收来自 PL 的中断信号，并且设置 packet_trans_done。

表：8-5-2-1 DMA_RxIntrHandler DMA 中断接收函数

```

/****************************************************************************
*
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.
*
* @param    Callback is a pointer to RX channel of the DMA engine.
*
* @return   None.
*
* @note    None.
*
****************************************************************************/
static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;
}

```

```
/* Read pending interrupts */
IrqStatus = XAxiDma_IntrGetIRQ(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);

/* Acknowledge pending interrupts */
XAxiDma_IntrAckIRQ(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

/*
 * If no interrupt is asserted, we do not do anything
 */
if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
    return;
}

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
 */
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
    xil_printf("rx error! \r\n");
    return;
}

/*
 * If completion interrupt is asserted, then set RxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
    if(packet_trans_done)
        xil_printf("last transmission has not finished!\r\n");
    else
        /*set the axidma done flag*/
        packet_trans_done = 1;
}
}
```

PS 的 dma 数据接收采用了乒乓操作的模式，两个缓冲区交替进行数据接收。

需要注意的是，XAxiDma_SimpleTransfer 函数中 Length，以字节为单位，每次发起 dma 时，所设置的 Length 的值必须大于或等于 PL 实际传输的数据长度，否则会出现错误。本例程中设置的长度为 2048 字节。

first_trans_start 是为了进行第一次先进行一次 DMA 中断传输，这样完成后设置 first_trans_start 为 0。以后每次完成网络传输后，再启动 DMA 接受。

TCP 数据包的发送主要依赖于 tcp_write 和 tcp_output 两个函数，tcp_write 将所需要发送的数据写入 tcp 发送缓冲区等待发送，tcp_output 函数则将缓存区内数据包

发送出去。在发送 TCP 数据包时，这两个函数往往要同时配合使用。

收发送函数的具体源码如下表。

8-5-2: send_received_data() 发送函数源码

```
void send_received_data()
{
#if __arm__
    int copy = 3;
#else
    int copy = 0;
#endif
    err_t err;
    int Status;
    struct tcp_pcb *tpcb = connected_pcb;

    /*initial the first axdma transmission, only excuse once*/
    if(!first_trans_start)
    {
        Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)RxBufferPtr[0],
                                         (u32)(PAKET_LENGTH), XAXIDMA_DEVICE_TO_DMA);
        if (Status != XST_SUCCESS)
        {
            xil_printf("axi dma failed! 0 %d\r\n", Status);
            return;
        }
        /*set the flag, so this part of code will not excuse again*/
        first_trans_start = 1;
    }

    /*if the last axidma transmission is done, the interrupt triggered, then start TCP transmission*/
    if(packet_trans_done)
    {
        if (!connected_pcb)
            return;

        /* if tcp send buffer has enough space to hold the data we want to transmit from PL, then start
        tcp transmission*/
        if (tcp_sndbuf(tpcb) > SEND_SIZE)
        {
            /*transmit received data through TCP*/
```

```

err = tcp_write(tpcb, RxBufferPtr[packet_index & 1], SEND_SIZE, copy);
if (err != ERR_OK) {
    xil_printf("txperf: Error on tcp_write: %d\r\n", err);
    connected_pcb = NULL;
    return;
}

err = tcp_output(tpcb);
if (err != ERR_OK) {
    xil_printf("txperf: Error on tcp_output: %d\r\n", err);
    return;
}

}

packet_index++;
/*clear the axidma done flag*/
packet_trans_done = 0;

/*initial the other axidma transmission when the current transmission is done*/
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)RxBufferPtr[(packet_index + 1)&1],
                                (u32)(PAKET_LENGTH), XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS)
{
    xil_printf("axi dma %d failed! %d \r\n", (packet_index + 1), Status);
    return;
}

}

}

```

3.3 TCP 发送流程

3.3.1 TCP 连接建立

在本例程中，zynq 作为客户端，PC 作为服务器。由 zynq 向 PC 主动发起 TCP 连接请求，通过 `tcp_connect` 函数便可以完成这个过程。该函数的参数包含了一个回调函数指针 `tcp_connected_fn`，该回调函数将在 TCP 连接请求三次握手完成后被自动调用。该回调函数被调用时代表客户端和服务器之间的 TCP 连接建立完成。在本例程中，该回调函数被定义为 `tcp_connected_callback`，在该函数中，拉高连接建立完成信号 `tcp_client_connected`，并通过 `tcp_sent` 函数配置另一个 TCP 发送完成的回调函数。该回调函数在每个 TCP 包发送完成后会被自动调用，代表 TCP 包发送完成。该回调函数在本例程中被定义为 `tcp_sent_callback`，仅作发送完成数据包的计数。

表 tcp connected callback 回调函数

```
static err_t  
tcp_connected_callback(void *arg, struct tcp_pcb *tpcb, err_t err)
```

```
{
    xil_printf("txperf: Connected to iperf server\r\n");

    /* store state */
    connected_pcb = tpcb;

    /* set callback values & functions */
    tcp_arg(tpcb, NULL);
    tcp_sent(tpcb, tcp_sent_callback);

    tcp_client_connected = 1;

    /* initiate data transfer */
    return ERR_OK;
}
```

表 tcp_sent_callback 回调函数

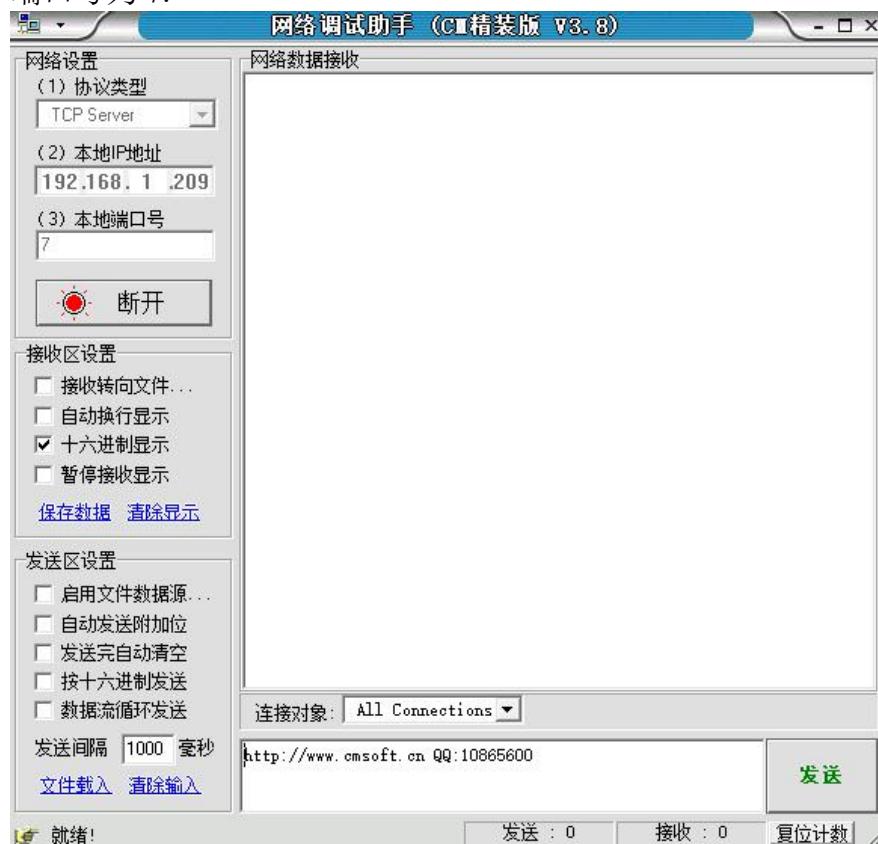
```
/*this fuction just used to count the tcp transmission times*/
static err_t
tcp_sent_callback(void *arg, struct tcp_pcb *tpcb, u16_t len)
{
    tcp_trans_done++;
    return ERR_OK;
}
```

8.6 连接测试

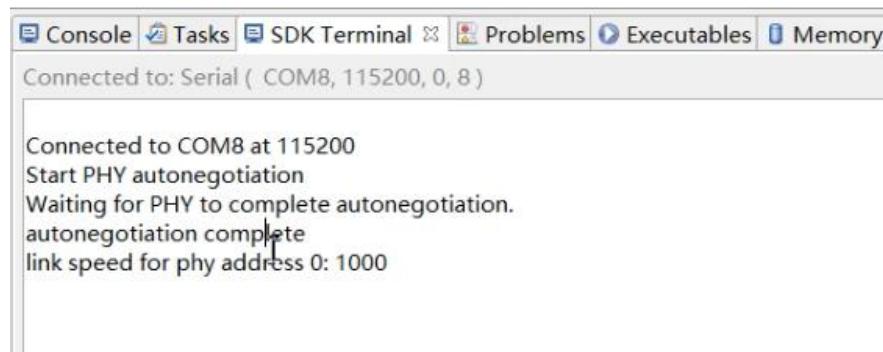
把开发板网卡通过网线接到 PC 网口上，修改 IP 地址如下图



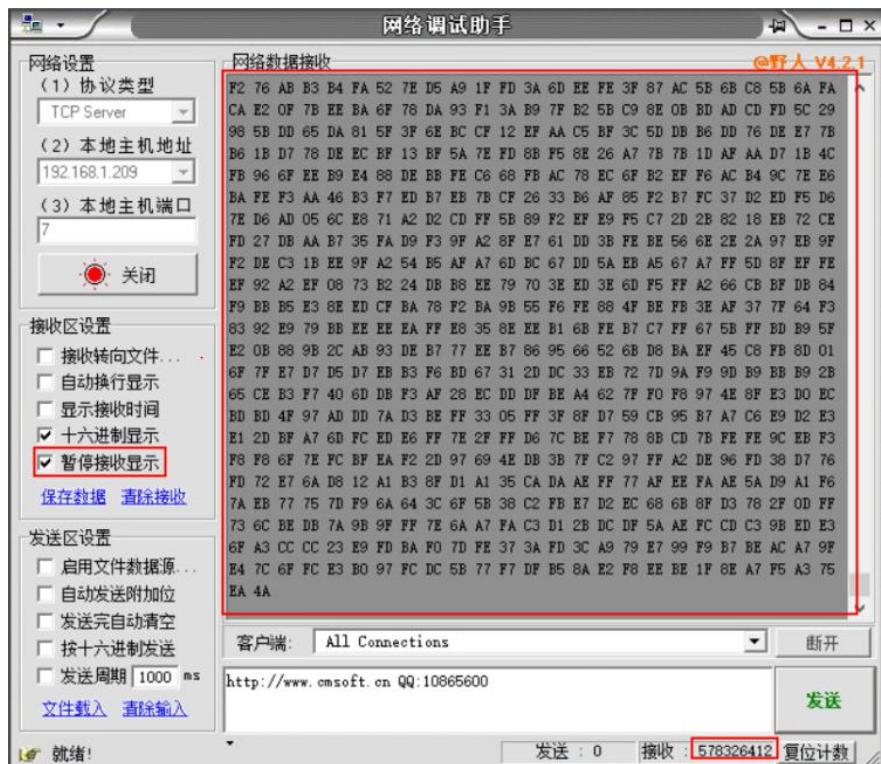
打开网络调试助手，第一次用的时候 windows 会提示你是否允许访问网络一定要选择是，否则你就无法通信了。设置电脑为 TCP Server 本机 IP 为刚才设置的 192.168.1.209 端口号为 7。



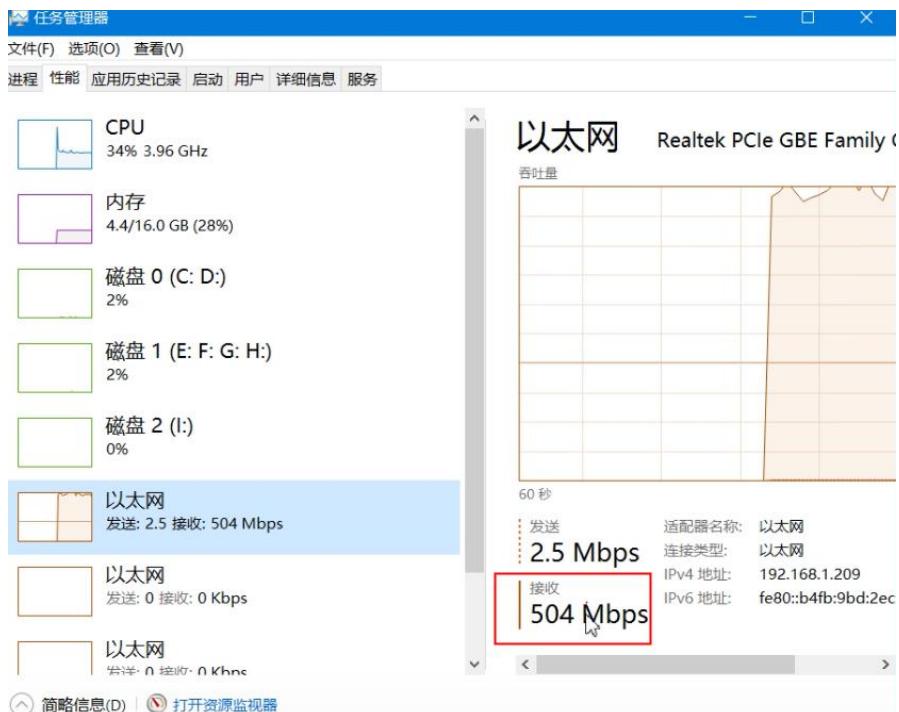
在 SDK 里面打开串口，并且启动 SDK 调试(调试方法前面已经讲过)，板子是 client 可以看到成功连接到了 PCB 上。



这个时候可以看到网络调试助手接收到数据了，由于数据量大，刷新数据显示，会导致电脑变慢，这里把勾选暂停显示。右下角是接收数据的计数器，可以看到计数器在飞奔中。



现在检测下网速，可以看到时间速度在 500Mbps/s 左右查看网速，大概是在 62MB/S-70MB/S 的速度，当然我们也可以通过优化实现更高速度的传输。



在 SDK 里面设定内存空间的查看地址，查看内存中的数据，可以看到正是 PL 发出的数据。

The screenshot shows the Xilinx Vivado IDE interface with the "Memory" tab selected. The left sidebar displays a tree view under "Monitors" with two entries: "0x10320000" and "0x1030000". The main window shows a memory dump for the range 0x10320000 to 0x10320000. The data is presented in a grid with columns labeled Address, 0 - 3, 4 - 7, 8 - B, and C - F. The first row of data is highlighted in blue, showing the address 10320000 and the value 00010000.

Address	0 - 3	4 - 7	8 - B	C - F
10320000	00010000	00030002	00050004	00070006
10320010	00090008	000B000A	000D000C	000F000E
10320020	00110010	00130012	00150014	00170016
10320030	00190018	001B001A	001D001C	001F001E
10320040	00210020	00230022	00250024	00270026
10320050	00290028	002B002A	002D002C	002F002E
10320060	00310030	00330032	00350034	00370036
10320070	00390038	003B003A	003D003C	003F003E

S03_CH09_DMA_4_Video_Switch 视频切换系统

9.1 概述

本例程详细创建过程和本季课程第一课《S03_CH01_AXI_DMA_LOOP 环路测试》非常类似，因此如果读者不清楚如何创建工作，请仔细阅读本季第一课时。本例程的基本原理如下。

PL 通过 OV7725 OV7725 实时 采集 1 路 640×480 视频，分别将原始彩色、视频，分别将原始彩色、R 分量、G 分量、B 分量作为常量输出，实现背景为红、绿、蓝 视频共 4 路视频通过 4 个独立的 AXI DMA IP 核传输至 PS 的 DDR 中进行缓存，然后再通过 AXIDMA AXIDMAAXIDMA 将 4 路视频同时 路视频同时 从 DDR 读出，通过 PL 在 VGA 显示器上 显示器上 切换显示 其中任意 1 路。视频切换通过 。视频切换通过 miz702 miz702miz702 底板的 SW2SW2SW2 按键（5 个按键的中心位置） 个按键的中心位置） 实现，每按 下一次 切换一路视频。

9.2 修改 OV_Sensor_ML 摄像头采集 IP

由于 MIZ702 开发板只有 1 路摄像头视频输入接口，MIZ701N 和 MIZ702N 只有 2 路视频输入接口，无法满足演示 4 路视频输入的接口需求，因此修改 OV_Sensor_ML ip 使之输出 4 路数据通路。修改的代码如下：

表 9-2-1:

```
module OV_Sensor_ML(
    input CLK_i,
    //----- CMOS sensor hardware interface -----
    input cmos_vsync_i, //cmos vsync
    input cmos_href_i, //cmos hsync refrence
    input cmos_pclk_i, //cmos pxiel clock
    output cmos_xclk_o, //cmos externl clock
    input[7:0] cmos_data_i, //cmos data
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    // output de_o,//data enable.
    output [23:0] rgb_o1,//data output,
    output [23:0] rgb_o2,//data output,
    output [23:0] rgb_o3,//data output,
    output [23:0] rgb_o4,//data output,
    output vid_clk_ce
);
//----- 视频输出解码模块 -----
wire [15:0]rgb_o_r;
assign rgb_o1 = {rgb_o_r[4:0],3'd0,rgb_o_r[10:5],2'd0,rgb_o_r[15:11],3'd0};
```

```

assign rgb_o2 = {5'b11111      ,3'd0 ,rgb_o_r[10:5],2'd0,rgb_o_r[15:11],3'd0};
assign rgb_o3 = {rgb_o_r[4:0],3'd0 ,rgb_o_r[10:5],2'd0,5'b11111      ,3'd0};
assign rgb_o4 = {rgb_o_r[4:0],3'd0 ,6'b111111      ,2'd0,rgb_o_r[15:11],3'd0};

reg [7:0]cmos_data_r;
reg cmos_href_r;
reg cmos_vsync_r;

always@(posedge cmos_pclk_i)
begin
    cmos_data_r <= cmos_data_i;
    cmos_href_r <= cmos_href_i;
    cmos_vsync_r <= cmos_vsync_i;
end
//assign rgb_o = 24'b11111111_00000000_11111111;
cmos_decode cmos_decode_u0(
    //system signal.
    .cmos_clk_i(CLK_i),//cmos senseor clock.
    .rst_n_i(RESETn_i2c),//system reset.active low.
    //cmos sensor hardware interface.
    .cmos_pclk_i(cmos_pclk_i),//(cmos_pclk),//input pixel clock.
    .cmos_href_i(cmos_href_r),//(cmos_href),//input pixel hs signal.
    .cmos_vsync_i(cmos_vsync_r),//(cmos_vsync),//input pixel vs signal.
    .cmos_data_i(cmos_data_r),//(cmos_data),//data.
    .cmos_xclk_o(cmos_xclk_o),//(cmos_xclk),//output clock to cmos sensor.
    //user interface.
    .hs_o(hs_o),//hs signal.
    .vs_o(vs_o),//vs signal.
    // .de_o(de_o),//data enable.
    .rgb565_o(rgb_o_r),//data output
    .vid_clk_ce(vid_clk_ce)
);

count_reset_v1#(
    .num(20'hffff0)
)(
    .clk_i(CLK_i),
    .rst_o(RESETn_i2c)
);

```

修改后代码为的为 OV_Sensor_ML.v 上表中红色字体部分，可以看出，代码之作了简单修改，增加了 3 路数据输出，为了让数据颜色有对比，第一路保持原始图像数据颜色，剩余三路增加了颜色背景。这样在做切换测试的时候就可以看到不同的通路变化了。

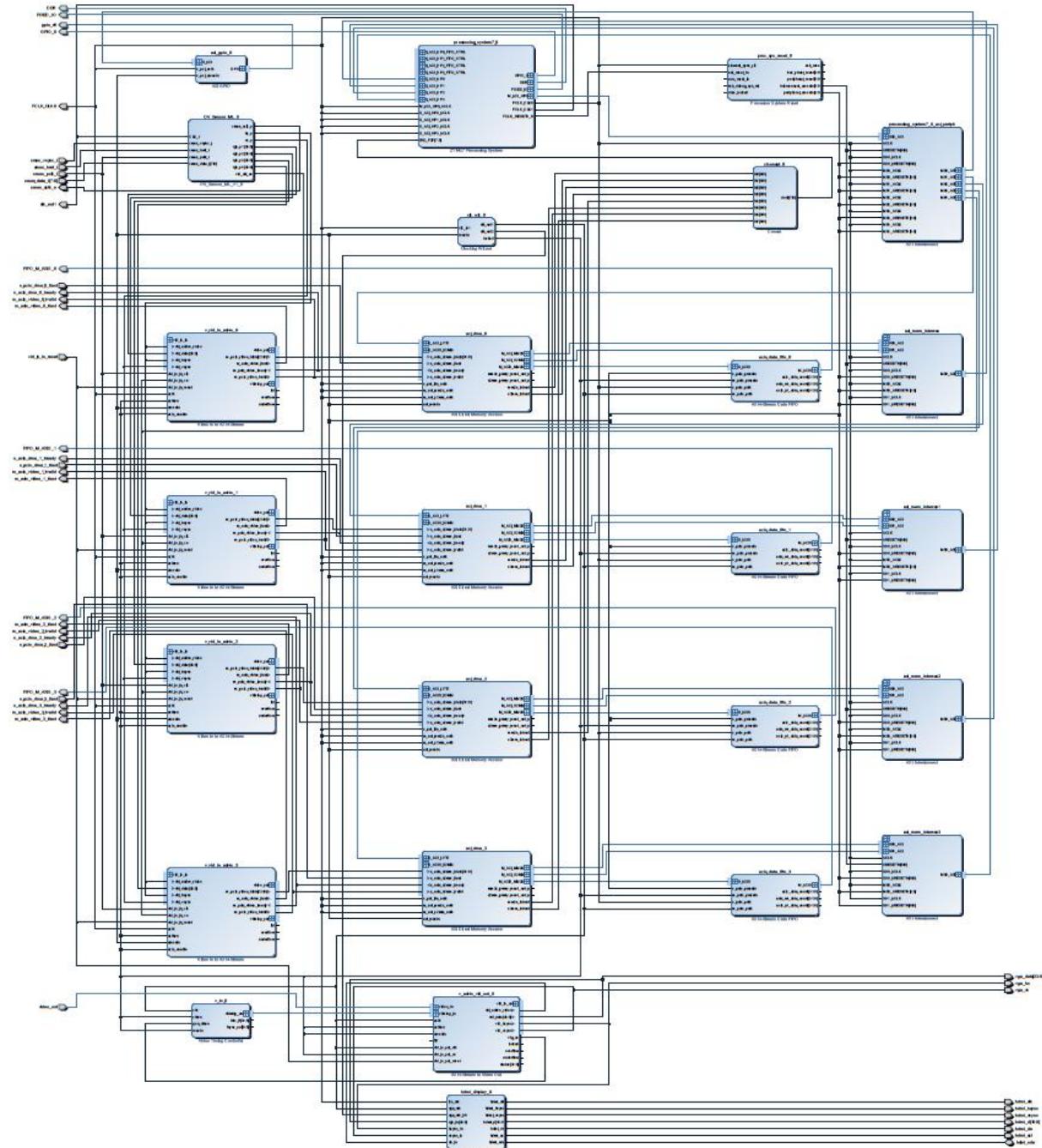
当然对于 MIZ701N 和 MIZ702N 可以使用 2 个 OV_Sensor_ML 接 2 个摄像头，每个模

块出来 2 个数据通道就可以了。如果要接 4 个摄像头，可以购买我们的外扩摄像头模块再扩展 2 路摄像头就可以实现 4 路摄像头了。

9.3 搭建硬件系统

9.3.1 系统图

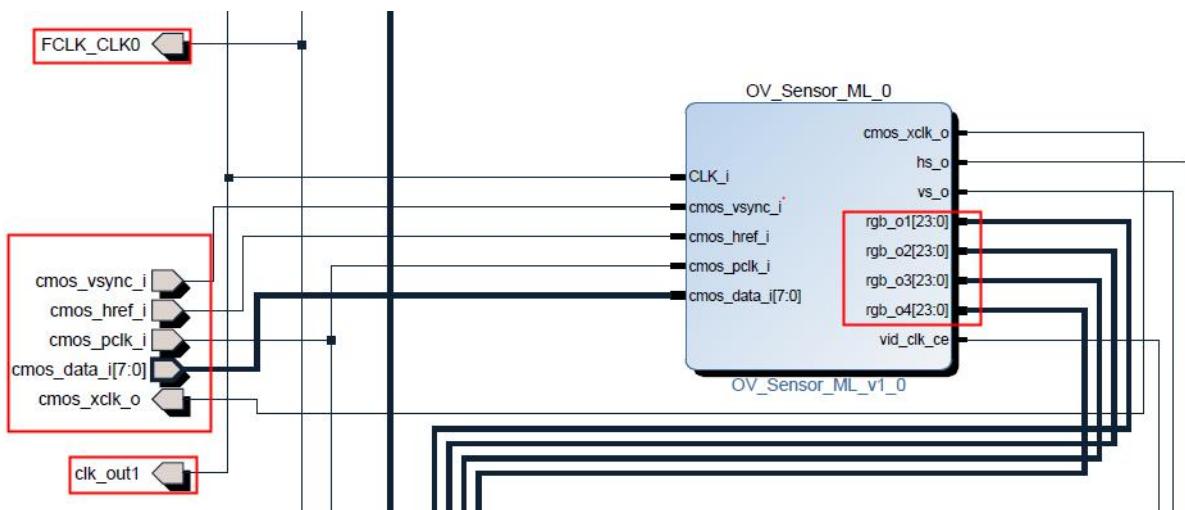
完成了 IP 的修改后，下面就可以搭建硬件系统了，由于 VIVADO 采用了图形化设计，带来了很大便捷。下面分别把 MIZ702 的系统构架图贴出来。



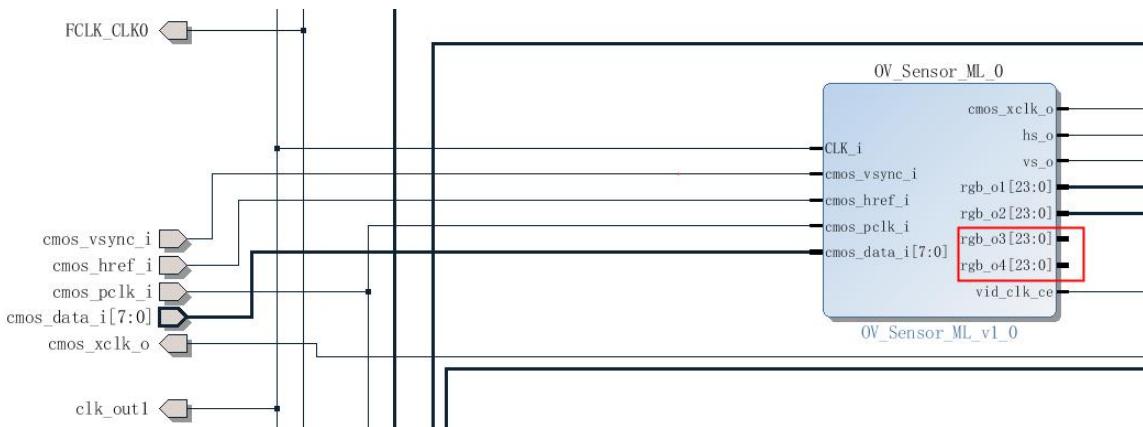
由于图片太大，只能看到大概的框架，大家学习的时候可以打开工程放大后去阅读，这里为了分析的时候方便把局部视图放大截图。

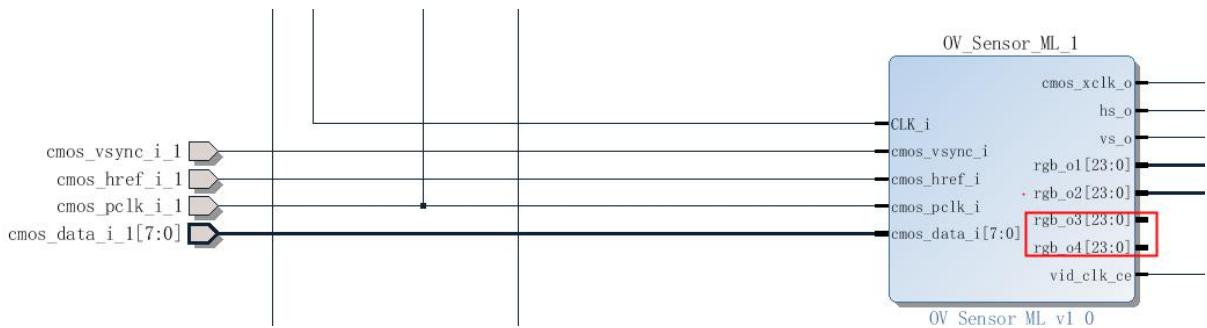
9.3.2 OV_Sensor_ML IP 接线图

下图中主要是前面我们自定义 OV_Sensor_ML 采集 IP 修改后的图形界面。可以看到多出了 rgb_o1、rgb_o2、rgb_o3、rgb_o4 接口这样我们就虚拟了 4 路摄像头数据输入接口啦。OV_Sensor_ML 前天的信号还是不变。下图中，还有 2 个信号分别是 FCLK_CLK0 和 clk_out1 他们分别是 VID_IN IP 和 VID_OUT IP 相关的时钟，把它们引出去到顶层模块中，后面需要使用到。



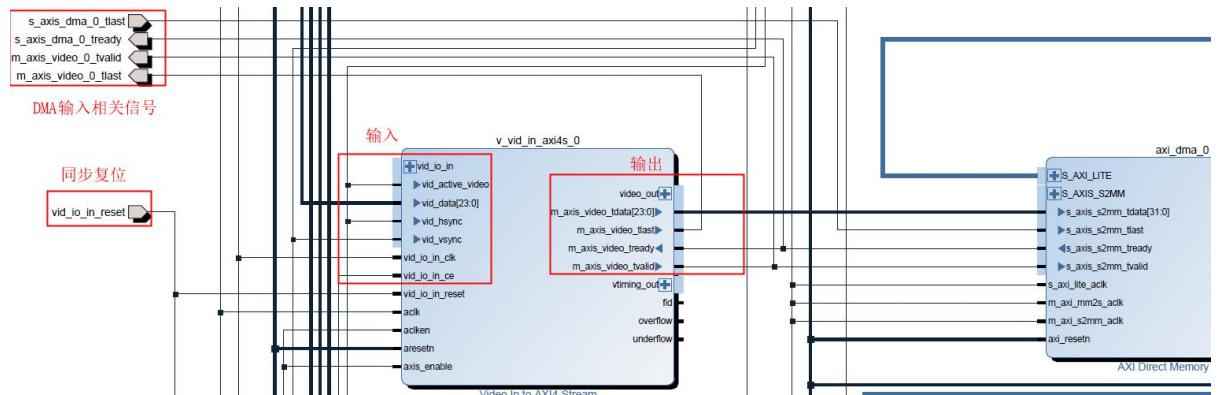
当然对于 MIZ701N 和 MIZ702N 可接 2 路摄像头的，那么这个模块只要接出 2 个通道就可以了，并且使用 2 个这样的模块。如下图是 MIZ702N 和 MIZ701N 连接 2 路摄像头。这里是链接了 rgb_o1 和 rgb_o2。





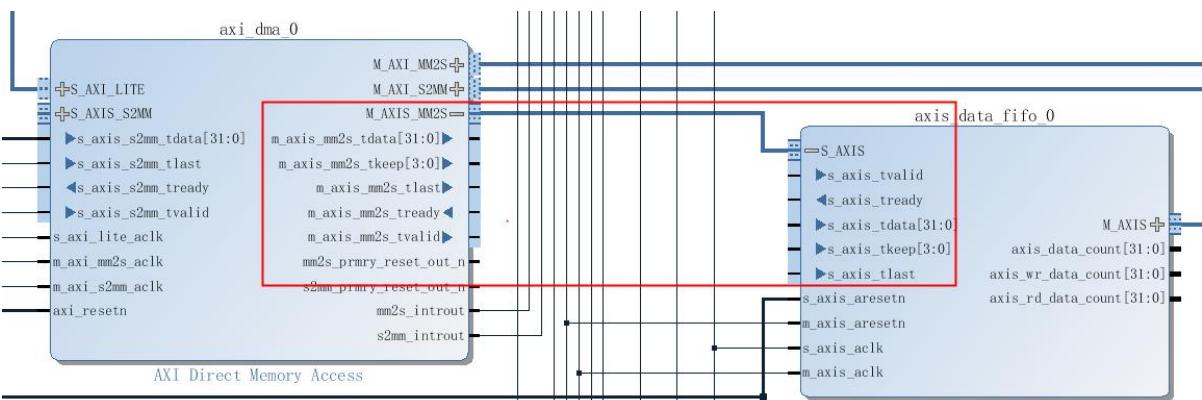
9.3.3 vid_in IP 的接线图

下图大家可以放大后观看，vid_in IP 的输入接口是连接到摄像头采样输出 IP 的。vid_in IP 的输出接口是和 DMA 链接了。DMA 输入相关的信号被引出到外部，用来添加 FPGA 代码实现写 DMA 时序。还有一个 vid_io_reset 信号，是用来控制所有 vid_in 和 vid_out IP 的同步，也是连接到外部，用 FPGA 代码控制。

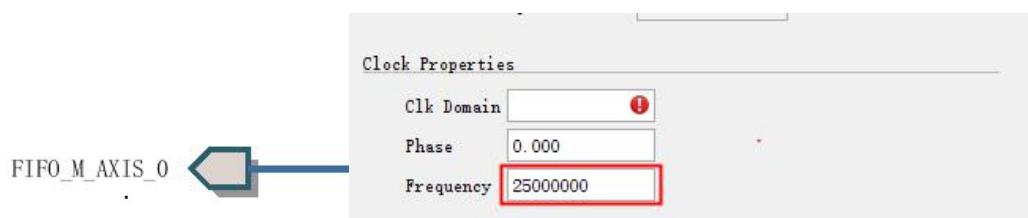


9.3.4 DMA 和 FIFO 通路

下图是 DMA 和 data fifo 的链接通路。

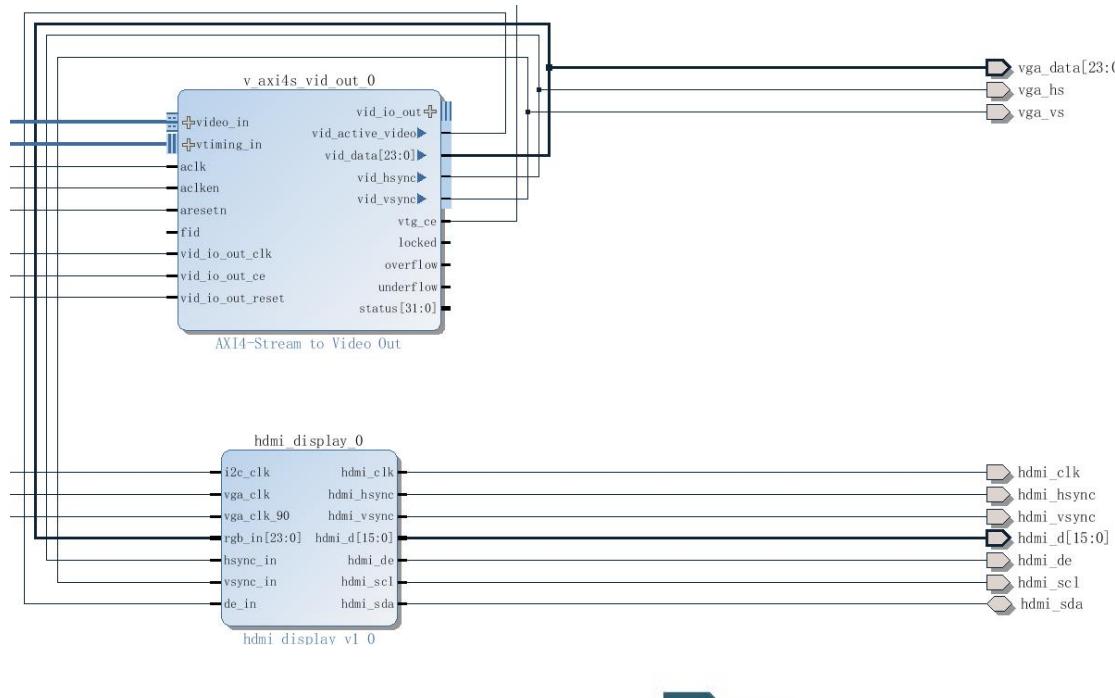


上图中 data fifo 的 M_AXIS 将被引出到外部受 FPGA 代码控制。如下图，FIFO_M_AXIS_0 就是连接到 axis_data_fifo_0 的 M_AXIS 接口的。双击此接口需要设置时钟，这里的数据速度时钟是 25MHZ。不同的分辨率应当设置对应的分辨率时钟。



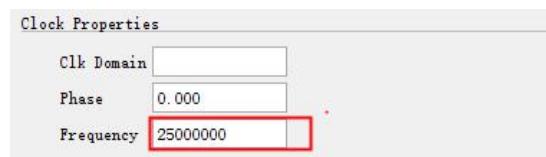
9.3.5 vid_out IP 的通路

输出部分可以看到 vid_out 输出的是 VGA 时序信号，在 VGA 时序信号上，我们还挂载了一个 VGA 转 HDMI 的 IP 实现了 HDMI 和 VGA 同时输出（MIZ701N 没有 VGA 所以无需把 VGA 信号，引出去）



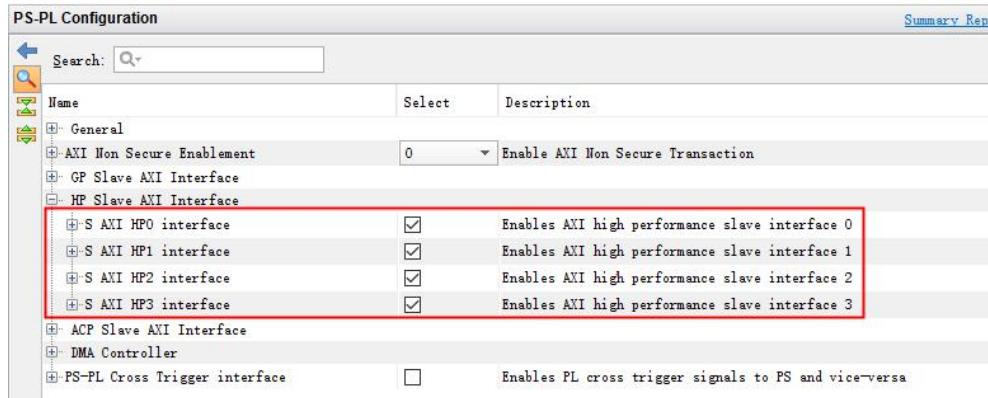
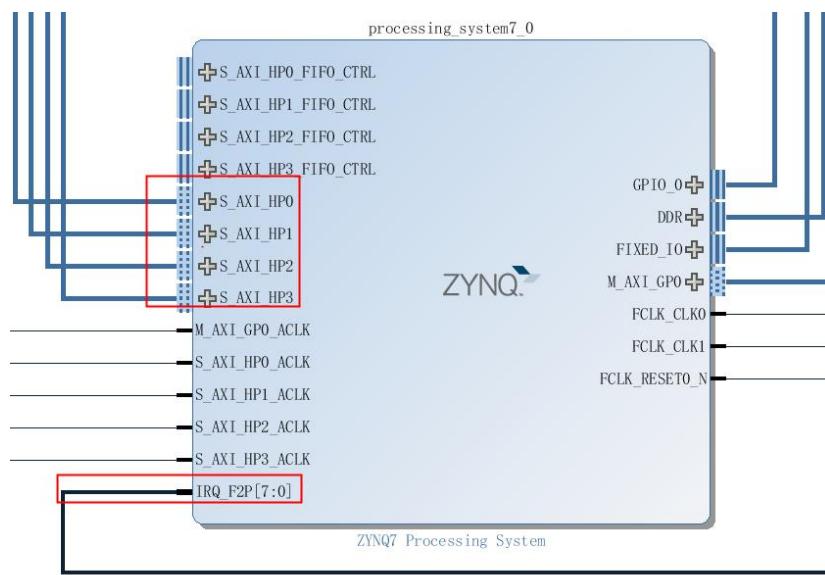
上图中的 vid_out IP 数据输入通道如图所示

双击这个接口也要设置时钟频率，由于输出像素为 640X480 因此为 25000000HZ



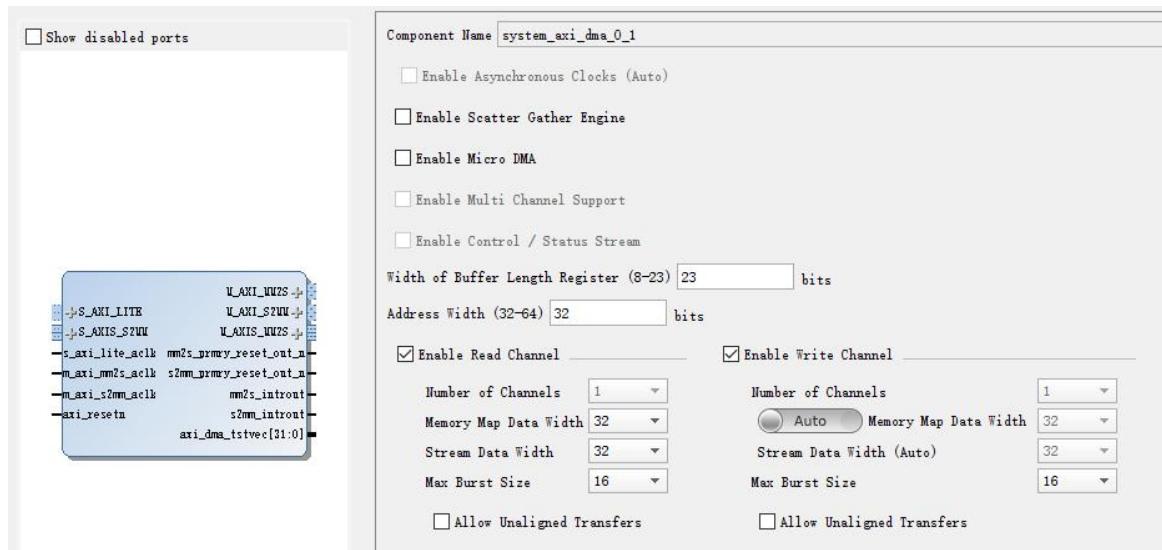
9.3.6 AXI HP 通道和 DMA 中断

由于是四路视频输入，外接了 4 个 DMA 模块因此使用了 4 个 HP 和 8 个 DMA 中断如图



9.3.7 DMA IP 的设置

下图中，同时勾选读通道和写通道，另外设置，Wideh of buffer length register 为 23bit 这个含义是 2 的 23 次方 8,388,607bytes 8M 大小。一副 1080P 的图像大小为 $1920 \times 1080 / 1024 / 1024 * 4 = 7.9M$ 因此一次 DMA 就可以传输一副 1080P 的图像。



9.3.8 时钟管理模块

时钟管理模块前面已经讲过了，640X480 的分辨率是设置 25MHZ，不在具体累述。

9.3.9 VTC 图像时序发生模块

VTC 图像时序发生模块的使用只要配置对应的分辨率，这里是设置 640X480 的分辨率，前面章节已经讲过不再累述。

9.4 FPGA 四路输入以及图像切换源码分析

9.4.1 按钮输入去抖代码

表 9-4-1

```
always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0) begin
        button_reg0 <= 1'b0;
        button_reg1 <= 1'b0;
        button_reg2 <= 1'b0;
        button_reg3 <= 1'b0;
    end
    else begin
        button_reg0 <= button;
        button_reg1 <= button_reg0;
```

```

        button_reg2 <= button_reg1;
        button_reg3 <= button_reg2;
    end
end

assign button_en = button_reg0 & button_reg1 & ~button_reg2 & ~button_reg3;

```

好简洁的去抖动代码，信号延迟 4 个时钟，连续监测到 4 次，就是认为按下了。关键稳定吗，还真不够稳定，你试试就知道了。有时候按下去会跳过几幅图像。所以按下去的时候要果断点，手指不要抖。哈哈，关键是代码简洁，满足了实验要求。读者如果要做自己的演示系统，还是把去抖动代码写好一些。

9.4.2 DMA 4 路视频输入的 FPGA 代码

表 9-4-2-1

```

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_0 <= 11'd0;
    else
        if(m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast)
            if(v_cnt_0 != 11'd479)
                v_cnt_0 <= v_cnt_0 + 1'b1;
            else
                v_cnt_0 <= 11'd0;
        else
            v_cnt_0 <= v_cnt_0;
end

```

上表可以看到 `gpio_rtl_tri_o_0` 就是可编程的复位信号，可以用 C 代码控制同步时序。上表的代码实现的是视频通路 0 的 vs 行计数器。可以看出来计数器在 `m_axis_video_0_tvalid` (vid in 输出数据有效)、`s_axis_dma_0_tready` (DMA 通道准备好)、`m_axis_video_0_tlast` (vid_in 行结束信号)都有效的时候累加 1。这里的分辨率是 640X480 因此累计一共 480 行数据。由于使用了 4 个输入输入通道，因此 vs 行计数器的完成代码如下表。

表 9-4-2-2

```

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_0 <= 11'd0;
    else
        if(m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast)
            if(v_cnt_0 != 11'd479)

```

```
v_cnt_0 <= v_cnt_0 + 1'b1;
else
    v_cnt_0 <= 11'd0;
else
    v_cnt_0 <= v_cnt_0;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_1 <= 11'd0;
    else
        if(m_axis_video_1_tvalid & s_axis_dma_1_tready & m_axis_video_1_tlast)
            if(v_cnt_1 != 11'd479)
                v_cnt_1 <= v_cnt_1 + 1'b1;
            else
                v_cnt_1 <= 11'd0;
        else
            v_cnt_1 <= v_cnt_1;
    end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_2 <= 11'd0;
    else
        if(m_axis_video_2_tvalid & s_axis_dma_2_tready & m_axis_video_2_tlast)
            if(v_cnt_2 != 11'd479)
                v_cnt_2 <= v_cnt_2 + 1'b1;
            else
                v_cnt_2 <= 11'd0;
        else
            v_cnt_2 <= v_cnt_2;
    end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_3 <= 11'd0;
    else
        if(m_axis_video_3_tvalid & s_axis_dma_3_tready & m_axis_video_3_tlast)
            if(v_cnt_3 != 11'd479)
                v_cnt_3 <= v_cnt_3 + 1'b1;
```

```

    else
        v_cnt_3 <= 11'd0;
    else
        v_cnt_3 <= v_cnt_3;
end

```

下表是 s_axis_dma_0_tlast、s_axis_dma_1_tlast、s_axis_dma_2_tlast、s_axis_dma_3_tlast 代表每个通道一副图像传输完成后的 last 信号。这个信号为高电平 1 个周期，提交一次 DMA 数据到 DDR，并且会产生一次对应端口的中断信号。

表 9-4-2-4

```

assign s_axis_dma_0_tlast = m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast &(v_cnt_0 ==
11'd479);
assign s_axis_dma_1_tlast = m_axis_video_1_tvalid & s_axis_dma_1_tready & m_axis_video_1_tlast &(v_cnt_1 ==
11'd479);
assign s_axis_dma_2_tlast = m_axis_video_2_tvalid & s_axis_dma_2_tready & m_axis_video_2_tlast &(v_cnt_2 ==
11'd479);
assign s_axis_dma_3_tlast = m_axis_video_3_tvalid & s_axis_dma_3_tready & m_axis_video_3_tlast &(v_cnt_3 ==
11'd479);

```

9.4.3 DMA 输出通道

```

always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0)
        h_cnt <= 11'd0;
    else
        if(video_out_tvalid & video_out_tready)
            if(h_cnt != 11'd639)
                h_cnt <= h_cnt + 1'b1;
            else
                h_cnt <= 11'd0;
        else
            h_cnt <= h_cnt;
end

```

表 9-4-3-1

上表是 vid out ip 输入数据部分的列计数器，一共有 640 列。当 video_out_tvalid(FIFO 输出数据有效信号)和 video_out_tready(vid out IP 准备好接收数据信号)都为 1 的时候开始计数。

表 9-4-3-2

```

always@(posedge clk_out1)
begin

```

```

if(!gpio_rtl_tri_o_0)
    v_cnt <= 11'd0;
else
    if(video_out_tvalid & video_out_tready & (h_cnt == 11'd639))
        if(v_cnt != 11'd479)
            v_cnt <= v_cnt + 1'b1;
        else
            v_cnt <= 11'd0;
    else
        v_cnt <= v_cnt;
end

```

上表是 vid out IP 输入数据的行计数器，当 video_out_tvalid (FIFO 数据输出有效) video_out_tready (vid out 准备好接收数据信号)和 h_cnt == 11'd639(代表一行数据结束)行计数器 v_cnt 加 1。

表 9-4-3-3

```

assign video_out_tdata = (channel_switch == 2'b00) ? FIFO_M_AXIS_0_tdata[23:0] :
                           ((channel_switch == 2'b01) ? FIFO_M_AXIS_1_tdata[23:0] :
                            ((channel_switch == 2'b10) ? FIFO_M_AXIS_2_tdata[23:0] :
                             FIFO_M_AXIS_3_tdata[23:0]));

assign video_out_tvalid = video_en & ((channel_switch == 2'b00) ? FIFO_M_AXIS_0_tvalid :
                                         ((channel_switch == 2'b01) ? FIFO_M_AXIS_1_tvalid :
                                          ((channel_switch == 2'b10) ? FIFO_M_AXIS_2_tvalid :
                                           FIFO_M_AXIS_3_tvalid)));

assign video_out_tuser = video_out_tvalid & video_out_tready & (h_cnt == 11'd0) & (v_cnt == 11'd0);

assign video_out_tlast = video_out_tvalid & video_out_tready & (h_cnt == 11'd639);

assign FIFO_M_AXIS_0_tready = video_en & video_out_tready;
assign FIFO_M_AXIS_1_tready = video_en & video_out_tready;
assign FIFO_M_AXIS_2_tready = video_en & video_out_tready;
assign FIFO_M_AXIS_3_tready = video_en & video_out_tready;

```

上表中， video_out_tvalid 是代表了 FIFO 输出的有效数据的信号，通过 channel_switch 切换到当前选定的 FIFO valid 信号上。

上表中， video_out_tdata 是代表了 FIFO 输出的数据通道，通过 channel_switch 切换到当前选定的 FIFO 数据通道。

上表中， video_out_tuser 是代表了 vid out 一帧图像开始信号。每行从 0 开始第一个数据。当 video_out_tvalid(FIFO 输出数据有效)、 video_out_tready(vid out 可以接收数据信号)、 h_cnt==11'd0(第一行第一个数据)、 v_cnt == 11'd0(一帧图像的第 0 行)都满足条件 video_out_tuser 输出 1, 告知 vid_out IP 一帧图像开始。

上表中， video_out_tlast 代表了 vid out 输入图像数据的一行结束，每一行结束都要输出

video_out_tlast 为 1.

9.5 4 路视频切换 DMA C 处理源码分析

9.5.1 main.c 源码

表 9-5-1 main.c

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 * axi dma test
 *
 */

#include "dma_intr.h"
#include "sys_intr.h"
#include "xgpio.h"

volatile int TxDone0;
volatile int TxDone1;
volatile int TxDone2;
volatile int TxDone3;
volatile int RxDone0;
volatile int RxDone1;
volatile int RxDone2;
volatile int RxDone3;
volatile u8 tx0_buffer_index;
volatile u8 rx0_buffer_index;
volatile u8 tx1_buffer_index;
volatile u8 rx1_buffer_index;
volatile u8 tx2_buffer_index;
volatile u8 rx2_buffer_index;
volatile u8 tx3_buffer_index;
volatile u8 rx3_buffer_index;
volatile int Error;

u32 *BufferPtr0[3];
u32 *BufferPtr1[3];
u32 *BufferPtr2[3];
```

```
u32 *BufferPtr3[3];\n\nXAxiDma AxiDma0;\nXAxiDma AxiDma1;\nXAxiDma AxiDma2;\nXAxiDma AxiDma3;\n\n/********************* Variable Definitions *****/\nstatic XScuGic Intc; //GIC\nstatic XGpio Gpio;\n\n#define AXI_GPIO_DEV_ID          XPAR_AXI_GPIO_0_DEVICE_ID\n\nint init_intr_sys(void)\n{\n    // initial DMA interrupt handle\n    DMA_Intr_Init(&AxiDma0,XPAR_AXIDMA_0_DEVICE_ID);\n    DMA_Intr_Init(&AxiDma1,XPAR_AXIDMA_1_DEVICE_ID);\n    DMA_Intr_Init(&AxiDma2,XPAR_AXIDMA_2_DEVICE_ID);\n    DMA_Intr_Init(&AxiDma3,XPAR_AXIDMA_3_DEVICE_ID);\n\n    Init_Intr_System(&Intc); // initial DMA interrupt system\n    Setup_Intr_Exception(&Intc);\n\n    DMA_Setup_Intr_System(&Intc,&AxiDma0,TX0_INTR_ID,RX0_INTR_ID);//setup dma interrupt system\n    DMA_Setup_Intr_System(&Intc,&AxiDma1,TX1_INTR_ID,RX1_INTR_ID);//setup dma interrupt system\n    DMA_Setup_Intr_System(&Intc,&AxiDma2,TX2_INTR_ID,RX2_INTR_ID);//setup dma interrupt system\n    DMA_Setup_Intr_System(&Intc,&AxiDma3,TX3_INTR_ID,RX3_INTR_ID);//setup dma interrupt system\n\n    DMA_Intr_Enable(&Intc,&AxiDma0);\n    DMA_Intr_Enable(&Intc,&AxiDma1);\n    DMA_Intr_Enable(&Intc,&AxiDma2);\n    DMA_Intr_Enable(&Intc,&AxiDma3);\n}\n\nint main(void)\n{\n    int Status;\n\n    XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);\n    XGpio_SetDataDirection(&Gpio, 1, 0);
```

```
BufferPtr0[0] = (u32 *)CH0_BUFFER0_BASE;
BufferPtr0[1] = (u32 *)CH0_BUFFER1_BASE;
BufferPtr0[2] = (u32 *)CH0_BUFFER2_BASE;

BufferPtr1[0] = (u32 *)CH1_BUFFER0_BASE;
BufferPtr1[1] = (u32 *)CH1_BUFFER1_BASE;
BufferPtr1[2] = (u32 *)CH1_BUFFER2_BASE;

BufferPtr2[0] = (u32 *)CH2_BUFFER0_BASE;
BufferPtr2[1] = (u32 *)CH2_BUFFER1_BASE;
BufferPtr2[2] = (u32 *)CH2_BUFFER2_BASE;

BufferPtr3[0] = (u32 *)CH3_BUFFER0_BASE;
BufferPtr3[1] = (u32 *)CH3_BUFFER1_BASE;
BufferPtr3[2] = (u32 *)CH3_BUFFER2_BASE;

/* Initialize flags before start transfer test */
TxDone0 = 0;
TxDone1 = 0;
TxDone2 = 0;
TxDone3 = 0;
RxDone0 = 0;
TxDone1 = 0;
TxDone2 = 0;
TxDone3 = 0;
tx0_buffer_index = 0;
rx0_buffer_index = 0;
tx1_buffer_index = 0;
rx1_buffer_index = 0;
tx2_buffer_index = 0;
rx2_buffer_index = 0;
tx3_buffer_index = 0;
rx3_buffer_index = 0;
Error = 0;

init_intr_sys();
Miz702_EMIO_init();
ov7725_init_rgb();

XGpio_DiscreteWrite(&Gpio, 1, 1);

Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[tx0_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
```

```
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma0 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[tx1_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma1 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[tx2_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma2 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[tx3_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma3 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[rx0_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma0 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[rx1_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma1 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[rx2_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
```

```

    xil_printf("tx axi dma2 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[rx3_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma3 failed! %d\r\n", Status);
    return XST_FAILURE;
}

while (1);
return XST_SUCCESS;
}

```

上表中的代码我们很熟悉了，这里是注册了 4 个 DMA 通道，8 个中断(接收和发送 4 路)。

9.5.2 dma_intr.h 源码

表 9-5-2 dma_intr.h

```

/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 */

#ifndef DMA_INTR_H
#define DMA_INTR_H
#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
#include "xscugic.h"

/********************* Constant Definitions ********************/
/*
 * Device hardware build related constants.
 */

#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR

```

```
#define IMAGE_WIDTH      640
#define IMAGE_HEIGHT     480
#define BYTES_PER_PIXEL   4
#define MAX_BUFFER_NUM    8

#define MEM_BASE_ADDR     0x10000000

#define DMA0_DEV_ID       XPAR_AXIDMA_0_DEVICE_ID
#define DMA1_DEV_ID       XPAR_AXIDMA_1_DEVICE_ID
#define DMA2_DEV_ID       XPAR_AXIDMA_2_DEVICE_ID
#define DMA3_DEV_ID       XPAR_AXIDMA_3_DEVICE_ID

#define RX0_INTR_ID        XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX0_INTR_ID        XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR
#define RX1_INTR_ID        XPAR_FABRIC_AXI_DMA_1_S2MM_INTROUT_INTR
#define TX1_INTR_ID        XPAR_FABRIC_AXI_DMA_1_MM2S_INTROUT_INTR
#define RX2_INTR_ID        XPAR_FABRIC_AXI_DMA_2_S2MM_INTROUT_INTR
#define TX2_INTR_ID        XPAR_FABRIC_AXI_DMA_2_MM2S_INTROUT_INTR
#define RX3_INTR_ID        XPAR_FABRIC_AXI_DMA_3_S2MM_INTROUT_INTR
#define TX3_INTR_ID        XPAR_FABRIC_AXI_DMA_3_MM2S_INTROUT_INTR

#define CH0_BUFFER0_BASE    (MEM_BASE_ADDR)
#define CH0_BUFFER1_BASE    (CH0_BUFFER0_BASE +      IMAGE_WIDTH * IMAGE_HEIGHT *
                           BYTES_PER_PIXEL)
#define CH0_BUFFER2_BASE    (CH0_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
                           BYTES_PER_PIXEL)

#define CH1_BUFFER0_BASE    (CH0_BUFFER0_BASE + MAX_BUFFER_NUM * IMAGE_WIDTH *
                           IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define CH1_BUFFER1_BASE    (CH1_BUFFER0_BASE +      IMAGE_WIDTH * IMAGE_HEIGHT *
                           BYTES_PER_PIXEL)
#define CH1_BUFFER2_BASE    (CH1_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
                           BYTES_PER_PIXEL)

#define CH2_BUFFER0_BASE    (CH1_BUFFER0_BASE + MAX_BUFFER_NUM * IMAGE_WIDTH *
                           IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define CH2_BUFFER1_BASE    (CH2_BUFFER0_BASE +      IMAGE_WIDTH * IMAGE_HEIGHT *
                           BYTES_PER_PIXEL)
#define CH2_BUFFER2_BASE    (CH2_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
                           BYTES_PER_PIXEL)

#define CH3_BUFFER0_BASE    (CH2_BUFFER0_BASE + MAX_BUFFER_NUM * IMAGE_WIDTH *
```

```
IMAGE_HEIGHT * BYTES_PER_PIXEL)

#define CH3_BUFFER1_BASE      (CH3_BUFFER0_BASE +          IMAGE_WIDTH * IMAGE_HEIGHT * 
BYTES_PER_PIXEL)
#define CH3_BUFFER2_BASE      (CH3_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT * 
BYTES_PER_PIXEL)

/* Timeout loop counter for reset
*/
#define RESET_TIMEOUT_COUNTER    10000
/* test start value
*/
#define TEST_START_VALUE    0xC
/*
 * Buffer and Buffer Descriptor related constant definition
*/
#define MAX_PKT_LEN        (IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
/*
 * transfer times
*/
#define NUMBER_OF_TRANSFERS 100000

extern volatile int TxDone0;
extern volatile int TxDone1;
extern volatile int TxDone2;
extern volatile int TxDone3;
extern volatile int RxDone0;
extern volatile int RxDone1;
extern volatile int RxDone2;
extern volatile int RxDone3;
extern volatile u8 tx0_buffer_index;
extern volatile u8 rx0_buffer_index;
extern volatile u8 tx1_buffer_index;
extern volatile u8 rx1_buffer_index;
extern volatile u8 tx2_buffer_index;
extern volatile u8 rx2_buffer_index;
extern volatile u8 tx3_buffer_index;
extern volatile u8 rx3_buffer_index;
extern volatile int Error;

extern u32 *BufferPtr0[3];
extern u32 *BufferPtr1[3];
extern u32 *BufferPtr2[3];
```

```

extern u32 *BufferPtr3[3];

extern XAxiDma AxiDma0;
extern XAxiDma AxiDma1;
extern XAxiDma AxiDma2;
extern XAxiDma AxiDma3;

int DMA_CheckData(int Length, u8 StartValue);
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr);
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);

#endif

```

上表中主要定义 DMA 用到的变量，每个 DMA 通道的地址分配，DMA 通道对象的定义，以及 DMA 中断函数、DMA 中断使能函数。

9.5.3 dma_intr.c 中断接收源码

表 9-5-3 DMA_RxIntrHandler 源码

```

/******************************************************************************/
/*
*
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.
*
* @param Callback is a pointer to RX channel of the DMA engine.
*
* @return None.
*
* @note      None.
*
*/
static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int Status;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);
}

```

```
/* Acknowledge pending interrupts */
XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

/*
 * If no interrupt is asserted, we do not do anything
 */
if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
    xil_printf("no interrupt! \r\n");
    return;
}

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
*/
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

//    Error = 1;

    xil_printf("rx error! \r\n");

    /* Reset could fail and hang
     * NEED a way to handle this or do not call it??
     */
//    XAxiDma_Reset(AxiDmaInst);

//    TimeOut = RESET_TIMEOUT_COUNTER;

//    while (TimeOut) {
//        if(XAxiDma_ResetIsDone(AxiDmaInst)) {
//            break;
//        }

//        TimeOut -= 1;
//    }

    return;
}

/*
 * If completion interrupt is asserted, then set RxDone flag
*/
```

```
/*
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    if(AxiDmaInst == &AxiDma0)
    {
        RxDone0++;
        if(rx0_buffer_index == 2)
            rx0_buffer_index = 0;
        else
            rx0_buffer_index++;

        Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[rx0_buffer_index],
                                         MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

        if (Status != XST_SUCCESS) {
            xil_printf("rx axi dma0 failed! 0 %d\r\n", Status);
            return;
        }
    }

    else if(AxiDmaInst == &AxiDma1)
    {
        RxDone1++;
        if(rx1_buffer_index == 2)
            rx1_buffer_index = 0;
        else
            rx1_buffer_index++;

        Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[rx1_buffer_index],
                                         MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

        if (Status != XST_SUCCESS) {
            xil_printf("rx axi dma1 failed! 0 %d\r\n", Status);
            return;
        }
    }

    else if(AxiDmaInst == &AxiDma2)
    {
        RxDone2++;
        if(rx2_buffer_index == 2)
            rx2_buffer_index = 0;
        else
            rx2_buffer_index++;

        Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[rx2_buffer_index],
                                         MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

        if (Status != XST_SUCCESS) {
            xil_printf("rx axi dma2 failed! 0 %d\r\n", Status);
            return;
        }
    }
}
```

```

Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[rx2_buffer_index],
    MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma2 failed! 0 %d\r\n", Status);
    return;
}

else if(AxiDmaInst == &AxiDma3)
{
    RxDone3++;
    if(rx3_buffer_index == 2)
        rx3_buffer_index = 0;
    else
        rx3_buffer_index++;

    Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[rx3_buffer_index],
        MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        xil_printf("rx axi dma3 failed! 0 %d\r\n", Status);
        return;
    }
    else
        xil_printf("error!\r\n");
}
}

```

上表中和单独 DMA 视频的却别就是通过 AxiDmaInst 判断当前 DMA 输入的通路，来确定当前输入当道下一次接收的数据需要保存到的 BUFFER 地址。

表 9-5-4-3 dma_intr.c 源码

9.5.4 dma_intr.c 中断发送源码

表 9-5-4-1 DMA_TxIntrHandler 函数源码

```

/****************************************************************************
*
* This is the DMA TX Interrupt handler function.
*
****************************************************************************/

```

```
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then sets the TxDone.flag
*
* @param Callback is a pointer to TX channel of the DMA engine.
*
* @return None.
*
* @note      None.
*
*****
static void DMA_TxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int Status;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE);

    /* Acknowledge pending interrupts */

    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
        xil_printf("no interrupt! \r\n");
        return;
    }

    /*
     * If error interrupt is asserted, raise error flag, reset the
     * hardware to recover from the error, and return with no further
     * processing.
     */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

        //Error = 1;
        xil_printf("tx error! \r\n");
    }
}
```

```
//      /*
//       * Reset should never fail for transmit channel
//       */
//      XAxiDma_Reset(AxiDmaInst);
//
//      TimeOut = RESET_TIMEOUT_COUNTER;
//
//      while (TimeOut) {
//          if (XAxiDma_ResetIsDone(AxiDmaInst)) {
//              break;
//          }
//
//          TimeOut -= 1;
//      }
//
//      return;
//  }

/*
 * If Completion interrupt is asserted, then set the TxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    if(AxiDmaInst == &AxiDma0)
    {
        TxDone0++;
        if(rx0_buffer_index == 0)
            tx0_buffer_index = 2;
        else
            tx0_buffer_index = rx0_buffer_index - 1;

        Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[tx0_buffer_index],
                                         MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

        if (Status != XST_SUCCESS) {
            xil_printf("tx axi dma0 failed! 0 %d\r\n", Status);
            return;
        }
    }

    else if(AxiDmaInst == &AxiDma1)
    {
        TxDone1++;
    }
}
```

```
if(rx1_buffer_index == 0)
    tx1_buffer_index = 2;
else
    tx1_buffer_index = rx1_buffer_index - 1;

Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[tx1_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma1 failed! 0 %d\r\n", Status);
    return;
}
}
else if(AxiDmaInst == &AxiDma2)
{
TxDone2++;
if(rx2_buffer_index == 0)
    tx2_buffer_index = 2;
else
    tx2_buffer_index = rx2_buffer_index - 1;

Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[tx2_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma2 failed! 0 %d\r\n", Status);
    return;
}
}
else if(AxiDmaInst == &AxiDma3)
{
TxDone3++;
if(rx3_buffer_index == 0)
    tx3_buffer_index = 2;
else
    tx3_buffer_index = rx3_buffer_index - 1;

Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[tx3_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma3 failed! 0 %d\r\n", Status);
    return;
}
```

```
    }
}
else
    xil_printf("error!\r\n");
}
}
```

上表的发送中断函数，和接收中断函数处理机制一致，也是通过 AxiDmaInst 判断当前 DMA 的通道，并且为当前 DMA 通道发送数据，指定对应的 BUFFER。

9.6 本章小结

本章给出的是一个实用化的 DMA 应用方案，设计了 4 路视频通过 DMA 输入到 DDR。在 C 代码中实现 3 缓存输出。输出的时候，提供切换 FIFO 的通道，实现把其中一路输出到显示器。本方案的应用场景可以用于电视广播系统、视频会议等。

S03_CH10_DMA_4_Video_Stitch 视频拼接系统

10.1 概述

注意：本课程和上一课程《S03_CH09_DMA_4_Video_Switch 视频切换系统》基本相同，不同部分用紫色字体或者框图注明。

PL 通过 OV7725 实时 采集 1 路 640×480 视频，分别将原始彩色、视频，分别将原始彩色、R 分量、G 分量、B 分量作为常量输出，实现背景为红、绿、蓝 视频共 4 路视频通过 4 个独立的 AXI DMA IP 核传输至 PS 的 DDR 中进行缓存，然后再通过 AXIDMA 将 4 路视频同时从 DDR 读出，通过 PL 在 VGA 显示器上以 1080P 分辨率同时显示 4 路原始分辨率拼接而成的视频。

10.2 修改 OV_Sensor_ML 摄像头采集 IP

由于 MIZ702 开发板只有 1 路摄像头视频输入接口，MIZ701N 和 MIZ702N 只有 2 路视频输入接口，无法满足演示 4 路视频输入的接口需求，因此修改 OV_Sensor_ML ip 使之输出 4 路数据通路。修改的代码如下：

表 10-2-1:

```
module OV_Sensor_ML(
    input CLK_i,
    //----- CMOS sensor hardware interface -----
    input cmos_vsync_i, //cmos vsync
    input cmos_href_i, //cmos hsync refrence
    input cmos_pclk_i, //cmos pxiel clock
    output cmos_xclk_o, //cmos externl clock
    input[7:0] cmos_data_i, //cmos data
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    // output de_o,//data enable.
    output [23:0] rgb_o1,//data output,
    output [23:0] rgb_o2,//data output,
    output [23:0] rgb_o3,//data output,
    output [23:0] rgb_o4,//data output,
    output vid_clk_ce
);
//-----视频输出解码模块-----
wire [15:0]rgb_o_r;
assign rgb_o1 = {rgb_o_r[4:0],3'd0 ,rgb_o_r[10:5],2'd0,rgb_o_r[15:11],3'd0};
assign rgb_o2 = {5'b11111 ,3'd0 ,rgb_o_r[10:5],2'd0,rgb_o_r[15:11],3'd0};
assign rgb_o3 = {rgb_o_r[4:0],3'd0 ,rgb_o_r[10:5],2'd0,5'b11111 ,3'd0};
assign rgb_o4 = {rgb_o_r[4:0],3'd0 ,6'b111111 ,2'd0,rgb_o_r[15:11],3'd0};
```

```

reg [7:0]cmos_data_r;
reg cmos_href_r;
reg cmos_vsync_r;

always@(posedge cmos_pclk_i)
begin
    cmos_data_r <= cmos_data_i;
    cmos_href_r <= cmos_href_i;
    cmos_vsync_r <= cmos_vsync_i;
end
//assign rgb_o = 24'b11111111_00000000_11111111;
cmos_decode cmos_decode_u0(
    //system signal.
    .cmos_clk_i(CLK_i),//cmos senseor clock.
    .rst_n_i(RESETn_i2c),//system reset.active low.
    //cmos sensor hardware interface.
    .cmos_pclk_i(cmos_pclk_i),//(cmos_pclk),//input pixel clock.
    .cmos_href_i(cmos_href_r),//(cmos_href),//input pixel hs signal.
    .cmos_vsync_i(cmos_vsync_r),//(cmos_vsync),//input pixel vs signal.
    .cmos_data_i(cmos_data_r),//(cmos_data),//data.
    .cmos_xclk_o(cmos_xclk_o),//(cmos_xclk),//output clock to cmos sensor.
    //user interface.
    .hs_o(hs_o),//hs signal.
    .vs_o(vs_o),//vs signal.
    //.
    .de_o(de_o),//data enable.
    .rgb565_o(rgb_o_r),//data output
    .vid_clk_ce(vid_clk_ce)
);
count_reset_v1#(
    .num(20'hffff0)
)(
    .clk_i(CLK_i),
    .rst_o(RESETn_i2c)
);

```

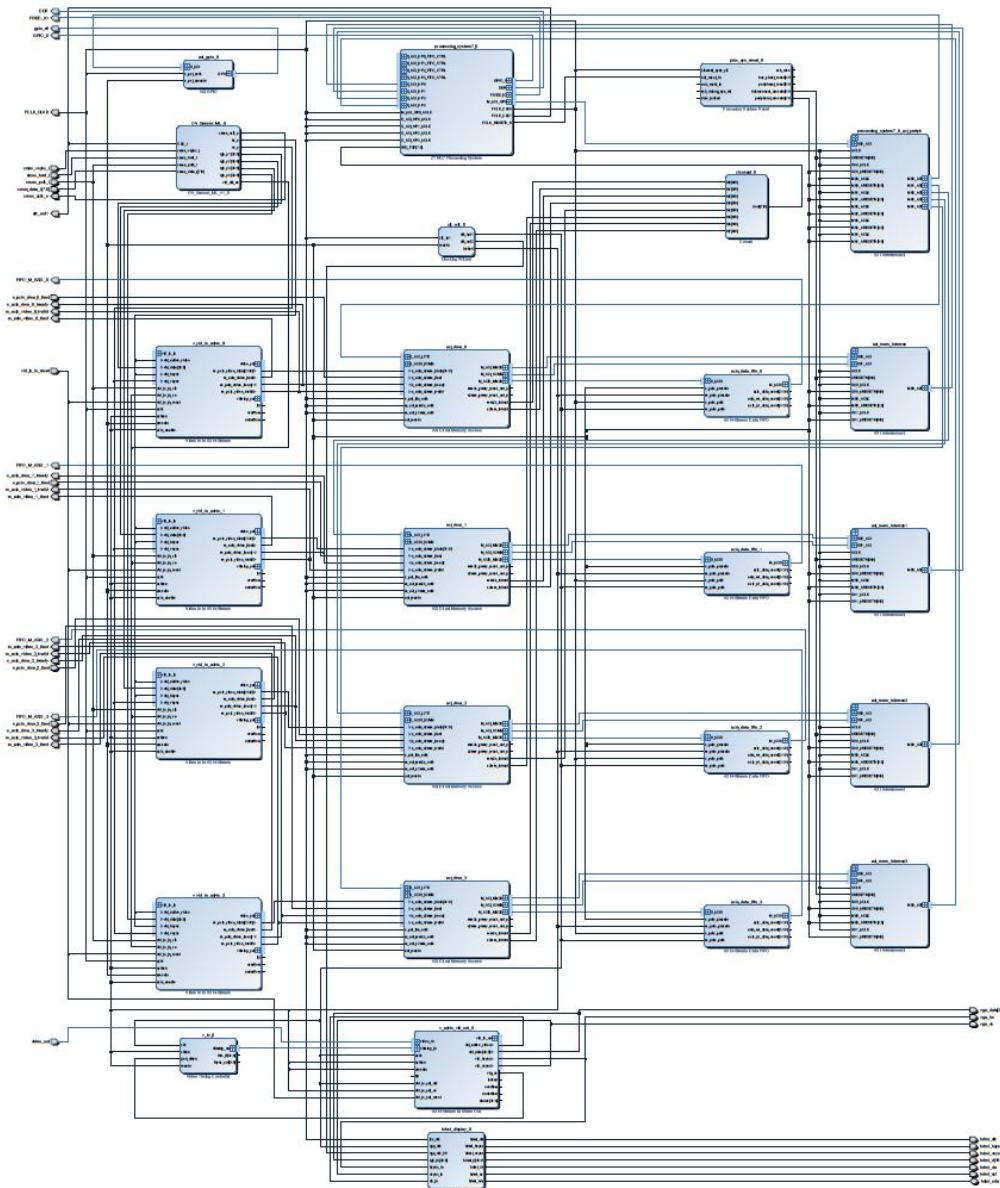
修改后代码为的为 OV_Sensor_ML.v 上表中红色字体部分，可以看出，代码之作了简单修改，增加了 3 路数据输出，为了让数据颜色有对比，第一路保持原始图像数据颜色，剩余三路增加了颜色背景。这样在做切换测试的时候就可以看到不同的通路变化了。

当然对于 MIZ701N 和 MIZ702N 可以使用 2 个 OV_Sensor_ML 接 2 个摄像头，每个模块出来 2 个数据通道就可以了。如果要接 4 个摄像头，可以购买我们的外扩摄像头模块再扩展 2 路摄像头就可以实现 4 路摄像头了。

10.3 搭建硬件系统

10.3.1 系统图

完成了 IP 的修改后，下面就可以搭建硬件系统了，由于 VIVADO 采用了图形化设计，带来了很大便捷。下面分别把 MIZ702 的系统构架图贴出来。

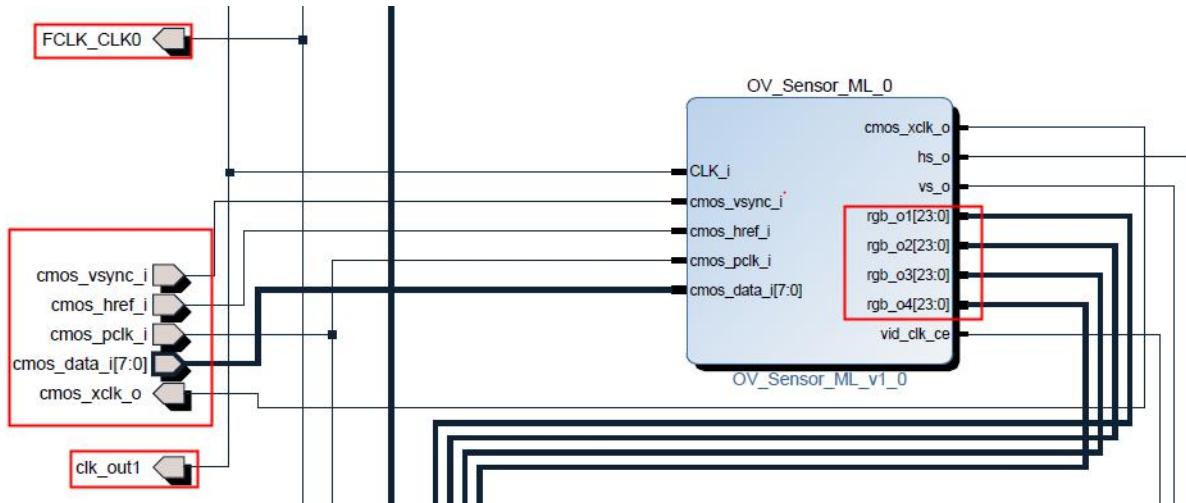


由于图片太大，只能看到大概的框架，大家学习的时候可以打开工程放大后去阅读，这里为了分析的时候方便把局部视图放大截图。

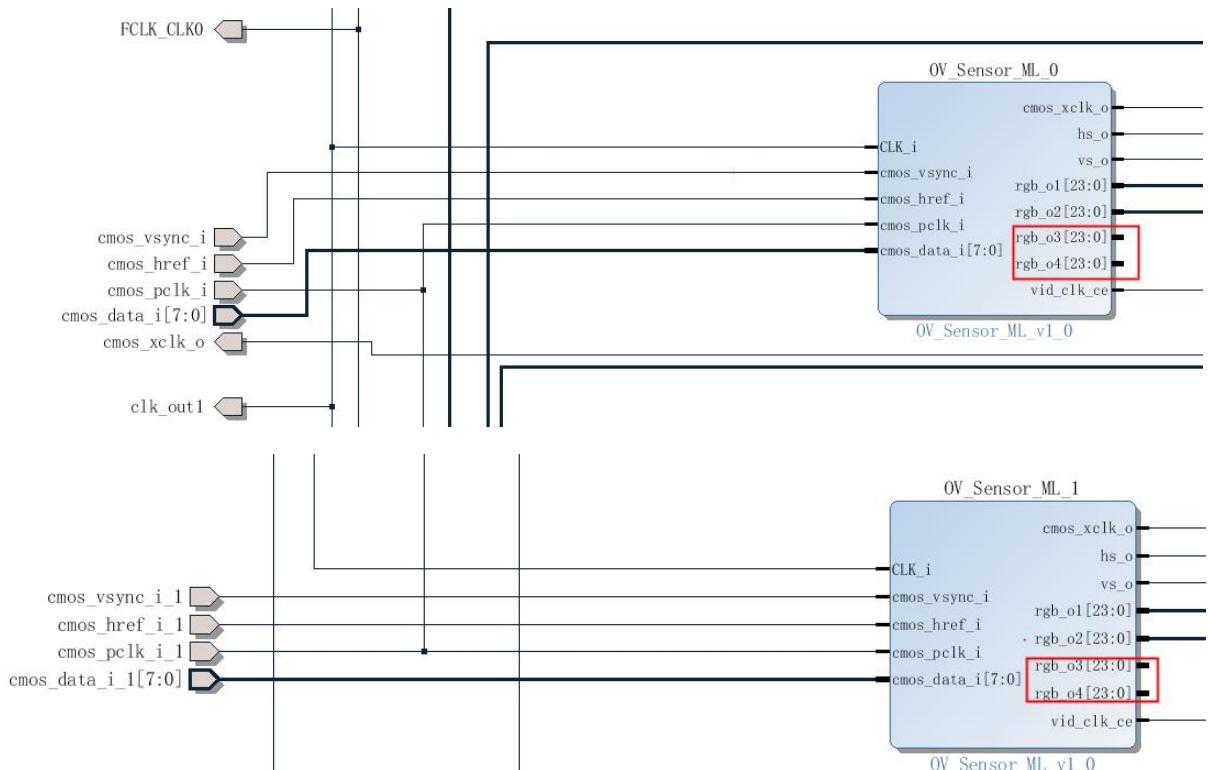
10.3.2 OV_Sensor_ML IP 接线图

下图中主要是前面我们自定义 OV_Sensor_ML 采集 IP 修改后的图形界面。可以看到

多出了 rgb_o1、rgb_o2、rgb_o3、rgb_o4 接口这样我们就虚拟了 4 路摄像头数据输入接口啦。OV_Sensor_ML 前天的信号还是不变。下图中，还有 2 个信号分别是 FCLK_CLK0 和 clk_out1 他们分别是 VID_IN IP 和 VID_OUT IP 相关的时钟，把它们引出去到顶层模块中，后面需要使用到。

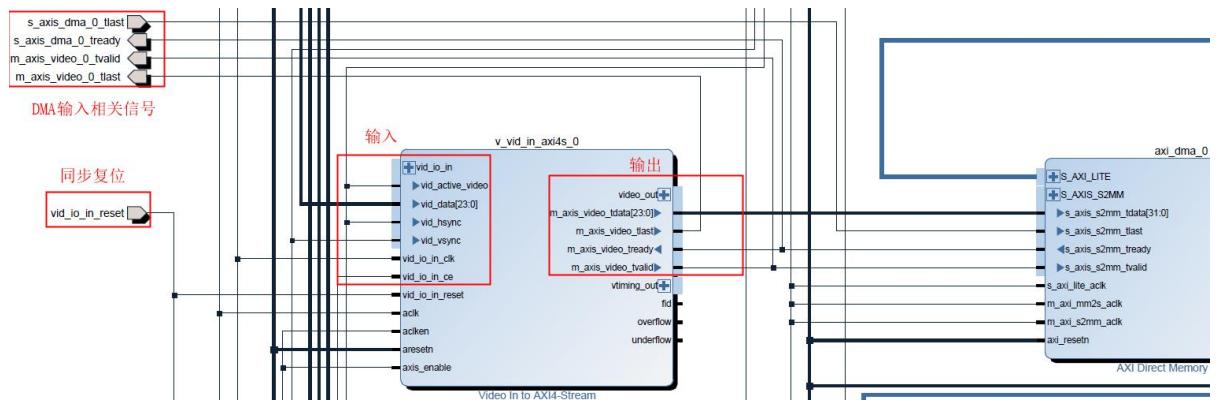


当然对于 MIZ701N 和 MIZ702N 可接 2 路摄像头的，那么这个模块只要接出 2 个通道就可以了，并且使用 2 个这样的模块。如下图是 MIZ702N 和 MIZ701N 连接 2 路摄像头。这里是链接了 rbg_o1 和 rbg_o2。



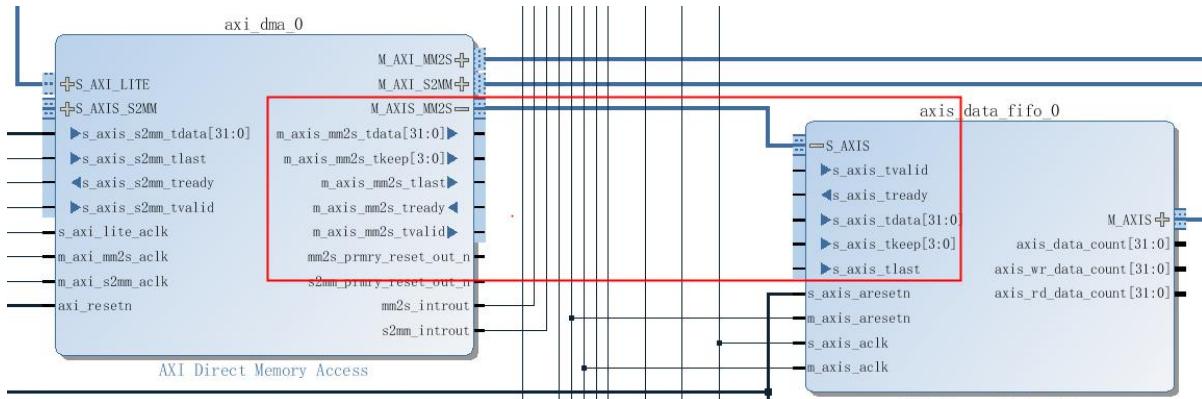
10.3.3 vid_in IP 的接线图

下图大家可以放大后观看，vid_in IP 的输入接口是连接到摄像头采样输出 IP 的。vid_in IP 的输出接口是和 DMA 链接了。DMA 输入相关的信号被引出到外部，用来添加 FPGA 代码实现写 DMA 时序。还有一个 vid_io_reset 信号，是用来控制所有 vid_in 和 vid_out IP 的同步，也是连接到外部，用 FPGA 代码控制。

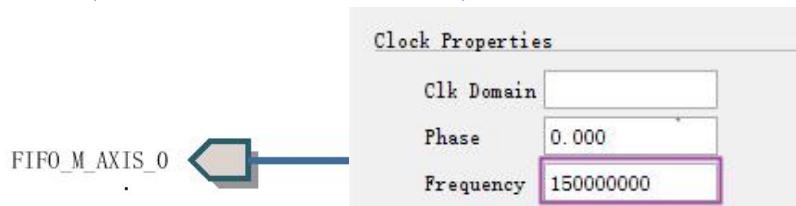


10.3.4 DMA 和 FIFO 通路

下图是 DMA 和 data fifo 的链接通路。

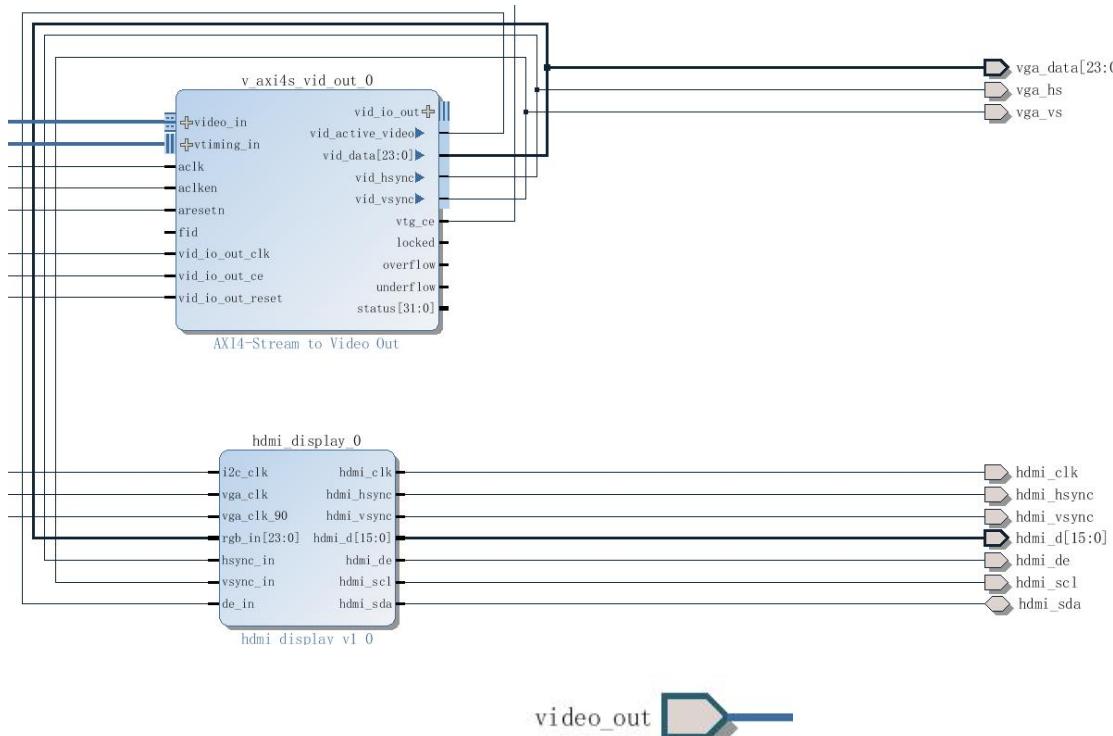


上图中 data fifo 的 M_AXIS 将被引出到外部受 FPGA 代码控制。如下图，FIFO_M_AXIS_0 就是连接到 axis_data_fifo_0 的 M_AXIS 接口的。双击此接口需要设置时钟，这里的数据速度时钟是 150MHZ(MIZ701N 是标准的 148.5MHZ)。不同的分辨率应当设置对应的分辨率时钟。



10.3.5 vid_out IP 的通路

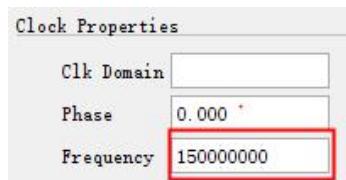
输出部分可以看到 vid_out 输出的是 VGA 时序信号，在 VGA 时序信号上，我们还挂载了一个 VGA 转 HDMI 的 IP 实现了 HDMI 和 VGA 同时输出（MIZ701N 没有 VGA 所以无需把 VGA 信号，引出去）



上图中的 vid out IP 数据输入通道如图所示

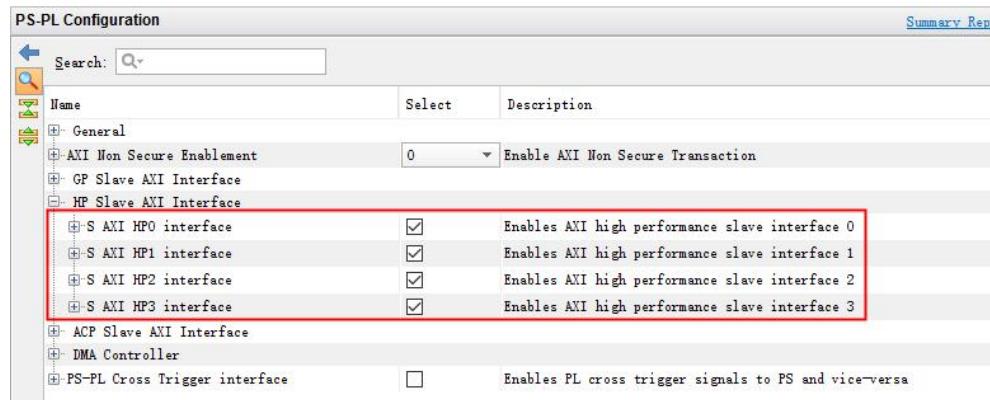
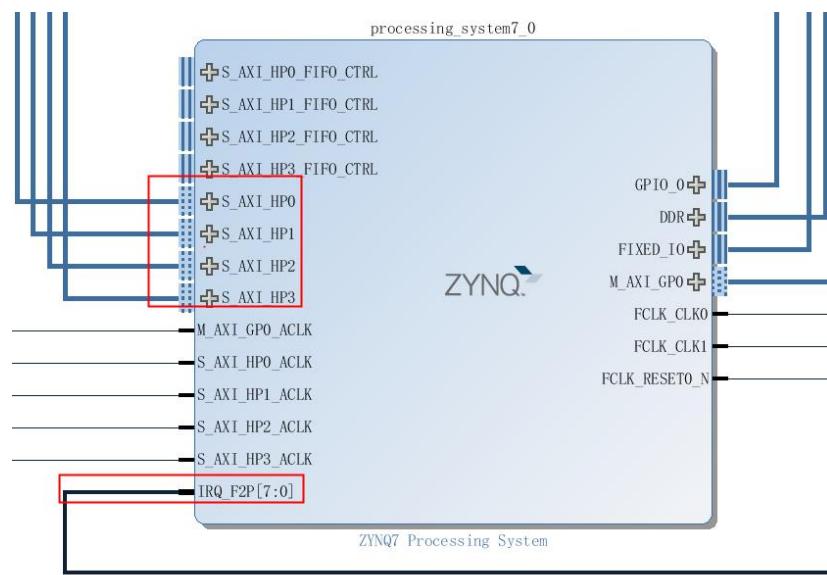


双击这个接口也要设置时钟频率，由于输出像素为 1920X108060HZ 因此为 15000000HZ(MIZ701N 是标准的 148500000)



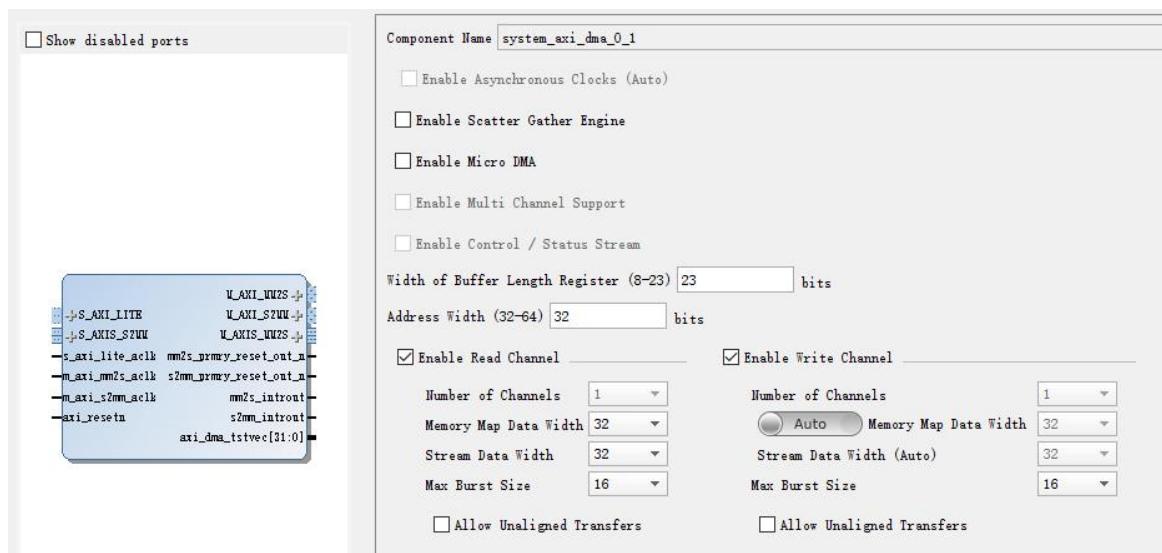
10.3.6 AXI HP 通道和 DMA 中断

由于是四路视频输入，外接了 4 个 DMA 模块因此使用了 4 个 HP 和 8 个 DMA 中断如图



10.3.7 DMA IP 的设置

下图中，同时勾选读通道和写通道，另外设置，Wideh of buffer length register 为 23bit 这个含义是 2 的 23 次方 8,388,607bytes 8M 大小。一副 1080P 的图像大小为 $1920 \times 1080 / 1024 / 1024 * 4 = 7.9M$ 因此一次 DMA 就可以传输一副 1080P 的图像。



10.3.7 时钟管理模块

时钟管理模块前面已经讲过了，1920X1080 的分辨率是设置 150MHZ(MIZ701N 是 148.5MHZ)，不在具体累述。

10.3.8 VTC 图像时序发生模块

VTC 图像时序发生模块的使用只要配置对应的分辨率，这里是设置 640X480 的分辨率，前面章节已经讲过不再累述。

10.4 FPGA 四路输入以及图像拼接源码分析

10.4.1 图像常量参数

表 10-4-1

localparam	MONITOR_HEIGHT = 11'd1080;//设置输出行分辨率
localparam	MONITOR_WIDTH = 11'd1920;//设置输出列分辨率
localparam	VIDEO_HEIGHT = 11'd480;//设置输入视频行分辨率
localparam	VIDEO_WIDTH = 11'd640;//设置输入视频列分辨率
localparam	GAP_HEIGHT = 11'd100;//设置四副图形中间的空白尺寸高度
localparam	GAP_WIDTH = 11'd100;//设置四副图形中间的空白尺寸宽度
localparam	BLACK_HEIGHT = (MONITOR_HEIGHT - 2 * VIDEO_HEIGHT - GAP_HEIGHT) >> 1;//上下边框高度 BLACK_HEIGHT = 10
localparam	BLACK_WIDTH = (MONITOR_WIDTH - 2 * VIDEO_WIDTH - GAP_WIDTH) >> 1;//左右边框宽度 BLACK_WIDTH = 270

```

localparam CH01_V_START = BLACK_HEIGHT;//第一路图像垂直开始 CH01_V_START = 10
localparam CH01_V_END      = BLACK_HEIGHT + VIDEO_HEIGHT;// 第一路图像垂直结束
CH01_V_END=490
localparam CH23_V_START = BLACK_HEIGHT + VIDEO_HEIGHT + GAP_HEIGHT;//第二路图像垂直开始
CH23_V_START=590
localparam CH23_V_END      = BLACK_HEIGHT + VIDEO_HEIGHT + GAP_HEIGHT + VIDEO_HEIGHT;//第二路图像垂直结束 CH23_V_END= 1070
localparam CH02_H_START = BLACK_WIDTH;//第一路图像水平开始 CH02_H_START=270
localparam CH02_H_END      = BLACK_WIDTH + VIDEO_WIDTH;//第一路图像水平结束 CH02_H_END=910
localparam CH13_H_START = BLACK_WIDTH + VIDEO_WIDTH + GAP_WIDTH;//第二路图像水平开始
1010
localparam CH13_H_END      = BLACK_WIDTH + VIDEO_WIDTH + GAP_WIDTH + VIDEO_WIDTH;第二路
图像水平结束//1550

```

以上代码是对图像的输出分辨率，被拼接图像的分辨率、空白、边框、背景进行设置。

10.4.2 DMA 4 路视频输入的 FPGA 代码

```

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_0 <= 11'd0;
    else
        if(m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast)
            if(v_cnt_0 != 11'd479)
                v_cnt_0 <= v_cnt_0 + 1'b1;
            else
                v_cnt_0 <= 11'd0;
        else
            v_cnt_0 <= v_cnt_0;
    end

```

表 10-4-2-1

上表可以看到 `gpio_rtl_tri_o_0` 就是可编程的复位信号，可以用 C 代码控制同步时序。上表的代码实现的是视频通路 0 的 vs 行计数器。可以看出来计数器在 `m_axis_video_0_tvalid` (vid in 输出数据有效)、`s_axis_dma_0_tready`(DMA 通道准备好)、`m_axis_video_0_tlast` (vid_in 行结束信号)都有效的时候累加 1。这里的分辨率是 640X480 因此累计一共 480 行数据。由于使用了 4 个输入输入通道，因此 vs 行计数器的完成代码如下表。

表 10-4-2-2

```
always@(posedge FCLK_CLK0)
```

```
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_0 <= 11'd0;
    else
        if(m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast)
            if(v_cnt_0 != 11'd479)
                v_cnt_0 <= v_cnt_0 + 1'b1;
            else
                v_cnt_0 <= 11'd0;
        else
            v_cnt_0 <= v_cnt_0;
    end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_1 <= 11'd0;
    else
        if(m_axis_video_1_tvalid & s_axis_dma_1_tready & m_axis_video_1_tlast)
            if(v_cnt_1 != 11'd479)
                v_cnt_1 <= v_cnt_1 + 1'b1;
            else
                v_cnt_1 <= 11'd0;
        else
            v_cnt_1 <= v_cnt_1;
    end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_2 <= 11'd0;
    else
        if(m_axis_video_2_tvalid & s_axis_dma_2_tready & m_axis_video_2_tlast)
            if(v_cnt_2 != 11'd479)
                v_cnt_2 <= v_cnt_2 + 1'b1;
            else
                v_cnt_2 <= 11'd0;
        else
            v_cnt_2 <= v_cnt_2;
    end

always@(posedge FCLK_CLK0)
begin
```

```

if(!gpio_rtl_tri_o_0)
    v_cnt_3 <= 11'd0;
else
    if(m_axis_video_3_tvalid & s_axis_dma_3_tready & m_axis_video_3_tlast)
        if(v_cnt_3 != 11'd479)
            v_cnt_3 <= v_cnt_3 + 1'b1;
        else
            v_cnt_3 <= 11'd0;
    else
        v_cnt_3 <= v_cnt_3;
end

```

下表是 s_axis_dma_0_tlast、s_axis_dma_1_tlast、s_axis_dma_2_tlast、s_axis_dma_3_tlast 代表每个通道一副图像传输完成后的 last 信号。这个信号为高电平 1 个周期，提交一次 DMA 数据到 DDR，并且会产生一次对应端口的中断信号。

表 10-4-2-3

```

assign s_axis_dma_0_tlast = m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast &(v_cnt_0 == 11'd479);
assign s_axis_dma_1_tlast = m_axis_video_1_tvalid & s_axis_dma_1_tready & m_axis_video_1_tlast &(v_cnt_1 == 11'd479);
assign s_axis_dma_2_tlast = m_axis_video_2_tvalid & s_axis_dma_2_tready & m_axis_video_2_tlast &(v_cnt_2 == 11'd479);
assign s_axis_dma_3_tlast = m_axis_video_3_tvalid & s_axis_dma_3_tready & m_axis_video_3_tlast &(v_cnt_3 == 11'd479);

```

10.4.3 DMA 输出通道

表 10-4-3-1

```

always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0)
        h_cnt <= 11'd0;
    else
        if(video_out_tvalid & video_out_tready)
            if(h_cnt != (MONITOR_WIDTH - 1'b1))
                h_cnt <= h_cnt + 1'b1;
            else
                h_cnt <= 11'd0;
        else
            h_cnt <= h_cnt;
end

```

上表是 vid out ip 输入数据部分的列计数器，一共有 1920 列。当 video_out_tvalid(FIFO 输出数据有

效信号)和 video_out_tready(vid out IP 准备好接收数据信号)都为 1 的时候开始计数。

表 10-4-3-2

```
always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt <= 11'd0;
    else
        if(video_out_tvalid & video_out_tready & (h_cnt == (MONITOR_WIDTH - 1'b1)))
            if(v_cnt != (MONITOR_HEIGHT - 1'b1))
                v_cnt <= v_cnt + 1'b1;
            else
                v_cnt <= 11'd0;
        else
            v_cnt <= v_cnt;
end
```

上表是 vid out IP 输入数据的行计数器，当 video_out_tvalid (FIFO 数据输出有效) video_out_tready (vid out 准备好接收数据信号)和 h_cnt == 11'd1919 共计 1920 点(代表一行数据结束) 行计数器 v_cnt 加 1。

```
always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0)
        channel_switch <= 3'd4;
    else
        if((v_cnt >= CH01_V_START) && (v_cnt < CH01_V_END))
            if((h_cnt >= CH02_H_START) && (h_cnt < CH02_H_END))
                channel_switch <= 3'd0;
            else if((h_cnt >= CH13_H_START) && (h_cnt < CH13_H_END))
                channel_switch <= 3'd1;
            else
                channel_switch <= 3'd4;
        else if((v_cnt >= CH23_V_START) && (v_cnt < CH23_V_END))
            if((h_cnt >= CH02_H_START) && (h_cnt < CH02_H_END))
                channel_switch <= 3'd2;
            else if((h_cnt >= CH13_H_START) && (h_cnt < CH13_H_END))
                channel_switch <= 3'd3;
            else
                channel_switch <= 3'd4;
        else
            channel_switch <= 3'd4;
end
```

表 10-4-3-3

上表代码实现了视频在显示器上的拼接输出，有点类似前面的四路切换方案，区别是这次是把所有视频全部输出到 1080P 的显示器上了。

表 10-4-3

```

assign video_out_tdata = (channel_switch == 3'd0) ? FIFO_M_AXIS_0_tdata[23:0] :
    ((channel_switch == 3'd1) ? FIFO_M_AXIS_1_tdata[23:0] :
    ((channel_switch == 3'd2) ? FIFO_M_AXIS_2_tdata[23:0] :
    ((channel_switch == 3'd3) ? FIFO_M_AXIS_3_tdata[23:0] : 24'd0));

assign video_out_tvalid = (channel_switch == 3'd0) ? FIFO_M_AXIS_0_tvalid :
    ((channel_switch == 3'd1) ? FIFO_M_AXIS_1_tvalid :
    ((channel_switch == 3'd2) ? FIFO_M_AXIS_2_tvalid :
    ((channel_switch == 3'd3) ? FIFO_M_AXIS_3_tvalid : 1'b1)));

assign video_out_tuser = video_out_tvalid & video_out_tready & (h_cnt == 11'd0) & (v_cnt == 11'd0);

assign video_out_tlast = (h_cnt == (MONITOR_WIDTH - 1'b1)) ? 1'b1 : 1'b0;

assign FIFO_M_AXIS_0_tready = (channel_switch == 3'd0) ? video_out_tready : 1'b0;
assign FIFO_M_AXIS_1_tready = (channel_switch == 3'd1) ? video_out_tready : 1'b0;
assign FIFO_M_AXIS_2_tready = (channel_switch == 3'd2) ? video_out_tready : 1'b0;
assign FIFO_M_AXIS_3_tready = (channel_switch == 3'd3) ? video_out_tready : 1'b0;

```

上表中，`video_out_tvalid` 是代表了 FIFO 输出的有效数据的信号，通过 `channel_switch` 切换到当前选定的 FIFO valid 信号上。

上表中，`video_out_tdata` 是代表了 FIFO 输出的数据通道，通过 `channel_switch` 切换到当前选定的 FIFO 数据通道。

上表中，`video_out_tuser` 是代表了 vid out 一帧图像开始信号。每行从 0 开始第一个数据。当 `video_out_tvalid`(FIFO 输出数据有效)、`video_out_tready`(vid out 可以接收数据信号)、`h_cnt==11'd0`(第一行第一个数据)、`v_cnt==11'd0`(一帧图像的第 0 行)都满足条件 `video_out_tuser` 输出 1，告知 vid_out IP 一帧图像开始。

上表中，`video_out_tlast` 代表了 vid out 输入图像数据的最后一行最后一个数据，这里是 1920X1080 的图像，因此到 1919，每一行最后一个数据都要输出 `video_out_tlast` 为 1.

10.5 4 路视频切换 DMA C 处理源码分析

10.5.4.1 main.c 源码

表 10-5-1 main.c

```

/*
 *

```

```
* www.osrc.cn
* www.milinker.com
* copyright by nan jin mi lian dian zi www.osrc.cn` 
* axi dma test
*
*/
#include "dma_intr.h"
#include "sys_intr.h"
#include "xgpio.h"

volatile int TxDone0;
volatile int TxDone1;
volatile int TxDone2;
volatile int TxDone3;
volatile int RxDone0;
volatile int RxDone1;
volatile int RxDone2;
volatile int RxDone3;
volatile u8 tx0_buffer_index;
volatile u8 rx0_buffer_index;
volatile u8 tx1_buffer_index;
volatile u8 rx1_buffer_index;
volatile u8 tx2_buffer_index;
volatile u8 rx2_buffer_index;
volatile u8 tx3_buffer_index;
volatile u8 rx3_buffer_index;
volatile int Error;

u32 *BufferPtr0[3];
u32 *BufferPtr1[3];
u32 *BufferPtr2[3];
u32 *BufferPtr3[3];

XAxiDma AxiDma0;
XAxiDma AxiDma1;
XAxiDma AxiDma2;
XAxiDma AxiDma3;

***** Variable Definitions *****/
static XScuGic Intc; //GIC
static XGpio Gpio;
```

```
#define AXI_GPIO_DEV_ID XPAR_AXI_GPIO_0_DEVICE_ID

int init_intr_sys(void)
{
    // initial DMA interrupt handle
    DMA_Intr_Init(&AxiDma0,XPAR_AXIDMA_0_DEVICE_ID);
    DMA_Intr_Init(&AxiDma1,XPAR_AXIDMA_1_DEVICE_ID);
    DMA_Intr_Init(&AxiDma2,XPAR_AXIDMA_2_DEVICE_ID);
    DMA_Intr_Init(&AxiDma3,XPAR_AXIDMA_3_DEVICE_ID);

    Init_Intr_System(&Intc); // initial DMA interrupt system
    Setup_Intr_Exception(&Intc);

    DMA_Setup_Intr_System(&Intc,&AxiDma0,TX0_INTR_ID,RX0_INTR_ID);//setup dma interrupt system
    DMA_Setup_Intr_System(&Intc,&AxiDma1,TX1_INTR_ID,RX1_INTR_ID);//setup dma interrupt system
    DMA_Setup_Intr_System(&Intc,&AxiDma2,TX2_INTR_ID,RX2_INTR_ID);//setup dma interrupt system
    DMA_Setup_Intr_System(&Intc,&AxiDma3,TX3_INTR_ID,RX3_INTR_ID);//setup dma interrupt system

    DMA_Intr_Enable(&Intc,&AxiDma0);
    DMA_Intr_Enable(&Intc,&AxiDma1);
    DMA_Intr_Enable(&Intc,&AxiDma2);
    DMA_Intr_Enable(&Intc,&AxiDma3);
}

int main(void)
{
    int Status;

    XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);
    XGpio_SetDataDirection(&Gpio, 1, 0);

    BufferPtr0[0] = (u32 *)CH0_BUFFER0_BASE;
    BufferPtr0[1] = (u32 *)CH0_BUFFER1_BASE;
    BufferPtr0[2] = (u32 *)CH0_BUFFER2_BASE;

    BufferPtr1[0] = (u32 *)CH1_BUFFER0_BASE;
    BufferPtr1[1] = (u32 *)CH1_BUFFER1_BASE;
    BufferPtr1[2] = (u32 *)CH1_BUFFER2_BASE;

    BufferPtr2[0] = (u32 *)CH2_BUFFER0_BASE;
    BufferPtr2[1] = (u32 *)CH2_BUFFER1_BASE;
```

```
BufferPtr2[2] = (u32 *)CH2_BUFFER2_BASE;

BufferPtr3[0] = (u32 *)CH3_BUFFER0_BASE;
BufferPtr3[1] = (u32 *)CH3_BUFFER1_BASE;
BufferPtr3[2] = (u32 *)CH3_BUFFER2_BASE;

/* Initialize flags before start transfer test */
TxDone0 = 0;
TxDone1 = 0;
TxDone2 = 0;
TxDone3 = 0;
RxDone0 = 0;
TxDone1 = 0;
TxDone2 = 0;
TxDone3 = 0;
tx0_buffer_index = 0;
rx0_buffer_index = 0;
tx1_buffer_index = 0;
rx1_buffer_index = 0;
tx2_buffer_index = 0;
rx2_buffer_index = 0;
tx3_buffer_index = 0;
rx3_buffer_index = 0;
Error = 0;

init_intr_sys();
Miz702_EMIO_init();
ov7725_init_rgb();

XGpio_DiscreteWrite(&Gpio, 1, 1);

Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[tx0_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma0 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[tx1_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma1 failed! %d\r\n", Status);
    return XST_FAILURE;
```

```
}

Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[tx2_buffer_index],
                                 MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma2 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[tx3_buffer_index],
                                 MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma3 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[rx0_buffer_index],
                                 MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma0 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[rx1_buffer_index],
                                 MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma1 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[rx2_buffer_index],
                                 MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma2 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[rx3_buffer_index],
                                 MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma3 failed! %d\r\n", Status);
    return XST_FAILURE;
}
```

```
    while (1);  
    return XST_SUCCESS;  
}
```

上表中的代码我们很熟悉了，这里是注册了4个DMA通道，8个中断(接收和发送4路)。

10.5.4.2 dma_intr.h 源码

表 10-5-2 dma_intr.h

```
/*  
*  
* www.osrc.cn  
* www.milinker.com  
* copyright by nan jin mi lian dian zi www.osrc.cn  
*/  
  
#ifndef DMA_INTR_H  
#define DMA_INTR_H  
  
#include "xaxidma.h"  
#include "xparameters.h"  
#include "xil_exception.h"  
#include "xdebug.h"  
#include "xscugic.h"  
  
***** Constant Definitions *****/  
/*  
 * Device hardware build related constants.  
 */  
  
#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR  
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR  
  
  
#define IMAGE_WIDTH      640  
#define IMAGE_HEIGHT     480  
#define BYTES_PER_PIXEL  4  
#define MAX_BUFFER_NUM   8  
  
#define MEM_BASE_ADDR    0x10000000  
  
#define DMA0_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID  
#define DMA1_DEV_ID      XPAR_AXIDMA_1_DEVICE_ID
```

```
#define DMA2_DEV_ID      XPAR_AXIDMA_2_DEVICE_ID
#define DMA3_DEV_ID      XPAR_AXIDMA_3_DEVICE_ID

#define RX0_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX0_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR
#define RX1_INTR_ID      XPAR_FABRIC_AXI_DMA_1_S2MM_INTROUT_INTR
#define TX1_INTR_ID      XPAR_FABRIC_AXI_DMA_1_MM2S_INTROUT_INTR
#define RX2_INTR_ID      XPAR_FABRIC_AXI_DMA_2_S2MM_INTROUT_INTR
#define TX2_INTR_ID      XPAR_FABRIC_AXI_DMA_2_MM2S_INTROUT_INTR
#define RX3_INTR_ID      XPAR_FABRIC_AXI_DMA_3_S2MM_INTROUT_INTR
#define TX3_INTR_ID      XPAR_FABRIC_AXI_DMA_3_MM2S_INTROUT_INTR

#define CH0_BUFFER0_BASE      (MEM_BASE_ADDR)
#define CH0_BUFFER1_BASE      (CH0_BUFFER0_BASE + IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)
#define CH0_BUFFER2_BASE      (CH0_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)

#define CH1_BUFFER0_BASE      (CH0_BUFFER0_BASE + MAX_BUFFER_NUM * IMAGE_WIDTH *
IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define CH1_BUFFER1_BASE      (CH1_BUFFER0_BASE + IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)
#define CH1_BUFFER2_BASE      (CH1_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)

#define CH2_BUFFER0_BASE      (CH1_BUFFER0_BASE + MAX_BUFFER_NUM * IMAGE_WIDTH *
IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define CH2_BUFFER1_BASE      (CH2_BUFFER0_BASE + IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)
#define CH2_BUFFER2_BASE      (CH2_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)

#define CH3_BUFFER0_BASE      (CH2_BUFFER0_BASE + MAX_BUFFER_NUM * IMAGE_WIDTH *
IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define CH3_BUFFER1_BASE      (CH3_BUFFER0_BASE + IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)
#define CH3_BUFFER2_BASE      (CH3_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)

/* Timeout loop counter for reset
*/
#define RESET_TIMEOUT_COUNTER    10000
```

```
/* test start value
*/
#define TEST_START_VALUE 0xC
/*
 * Buffer and Buffer Descriptor related constant definition
 */
#define MAX_PKT_LEN      (IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
/*
 * transfer times
 */
#define NUMBER_OF_TRANSFERS 100000

extern volatile int TxDone0;
extern volatile int TxDone1;
extern volatile int TxDone2;
extern volatile int TxDone3;
extern volatile int RxDone0;
extern volatile int RxDone1;
extern volatile int RxDone2;
extern volatile int RxDone3;
extern volatile u8 tx0_buffer_index;
extern volatile u8 rx0_buffer_index;
extern volatile u8 tx1_buffer_index;
extern volatile u8 rx1_buffer_index;
extern volatile u8 tx2_buffer_index;
extern volatile u8 rx2_buffer_index;
extern volatile u8 tx3_buffer_index;
extern volatile u8 rx3_buffer_index;
extern volatile int Error;

extern u32 *BufferPtr0[3];
extern u32 *BufferPtr1[3];
extern u32 *BufferPtr2[3];
extern u32 *BufferPtr3[3];

extern XAxiDma AxiDma0;
extern XAxiDma AxiDma1;
extern XAxiDma AxiDma2;
extern XAxiDma AxiDma3;
int DMA_CheckData(int Length, u8 StartValue);
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr);
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);
```

```
#endif
```

上表中主要定义 DMA 用到的变量，每个 DMA 通道的地址分配，DMA 通道对象的定义，以及 DMA 中断函数、DMA 中断使能函数。

10.5.4.3 dma_intr.c 中断接收源码

表 10-5-4-3

```
*****  
/*  
*  
* This is the DMA RX interrupt handler function  
*  
* It gets the interrupt status from the hardware, acknowledges it, and if any  
* error happens, it resets the hardware. Otherwise, if a completion interrupt  
* is present, then it sets the RxDone flag.  
*  
* @param Callback is a pointer to RX channel of the DMA engine.  
*  
* @return None.  
*  
* @note None.  
*  
*****  
static void DMA_RxIntrHandler(void *Callback)  
{  
    u32 IrqStatus;  
    int Status;  
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;  
  
    /* Read pending interrupts */  
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);  
  
    /* Acknowledge pending interrupts */  
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);  
  
    /*  
     * If no interrupt is asserted, we do not do anything  
     */  
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {  
        xil_printf("no interrupt! \r\n");  
        return;  
    }
```

```
}

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
 */
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

    // Error = 1;

    xil_printf("rx error! \r\n");

    /* Reset could fail and hang
     * NEED a way to handle this or do not call it??
     */
    // XAxiDma_Reset(AxiDmaInst);

    // TimeOut = RESET_TIMEOUT_COUNTER;

    // while (TimeOut) {
    //     if(XAxiDma_ResetIsDone(AxiDmaInst)) {
    //         break;
    //     }

    //     TimeOut -= 1;
    // }

    return;
}

/*
 * If completion interrupt is asserted, then set RxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    if(AxiDmaInst == &AxiDma0)
    {

        RxDone0++;
        if(rx0_buffer_index == 2)
            rx0_buffer_index = 0;
        else
            rx0_buffer_index++;
    }
}
```

```
Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[rx0_buffer_index],  
    MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);  
  
if (Status != XST_SUCCESS) {  
    xil_printf("rx axi dma0 failed! 0 %d\r\n", Status);  
    return;  
}  
}  
  
else if(AxiDmaInst == &AxiDma1)  
{  
    RxDone1++;  
    if(rx1_buffer_index == 2)  
        rx1_buffer_index = 0;  
    else  
        rx1_buffer_index++;  
  
    Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[rx1_buffer_index],  
        MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);  
  
    if (Status != XST_SUCCESS) {  
        xil_printf("rx axi dma1 failed! 0 %d\r\n", Status);  
        return;  
    }  
}  
else if(AxiDmaInst == &AxiDma2)  
{  
    RxDone2++;  
    if(rx2_buffer_index == 2)  
        rx2_buffer_index = 0;  
    else  
        rx2_buffer_index++;  
  
    Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[rx2_buffer_index],  
        MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);  
  
    if (Status != XST_SUCCESS) {  
        xil_printf("rx axi dma2 failed! 0 %d\r\n", Status);  
        return;  
    }  
}  
else if(AxiDmaInst == &AxiDma3)
```

```

{
    RxDone3++;

    if(rx3_buffer_index == 2)
        rx3_buffer_index = 0;
    else
        rx3_buffer_index++;

    Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[rx3_buffer_index],
                                    MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        xil_printf("rx axi dma3 failed! 0 %d\r\n", Status);
        return;
    }
}
else
    xil_printf("error!\r\n");
}

}

```

上表中和单独 DMA 视频的却别就是通过 AxiDmaInst 判断当前 DMA 输入的通路，来确定当前输入当道下一次接收的数据需要保存到的 BUFFER 地址。

表 9-5-4-3 dma_intr.c 源码

10.5.4.4 dma_intr.c 中断发送源码

表 10-5-4-4

```

*****
/*
*
* This is the DMA TX Interrupt handler function.
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then sets the TxDone.flag
*
* @param Callback is a pointer to TX channel of the DMA engine.
*
* @return None.
*
* @note      None.
*

```

```
*****
static void DMA_TxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int Status;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE);

    /* Acknowledge pending interrupts */

    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
        xil_printf("no interrupt! \r\n");
        return;
    }

    /*
     * If error interrupt is asserted, raise error flag, reset the
     * hardware to recover from the error, and return with no further
     * processing.
     */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

        //Error = 1;
        xil_printf("tx error! \r\n");

        //
        /*
         * Reset should never fail for transmit channel
         */
        //
        XAxiDma_Reset(AxiDmaInst);
        //
        TimeOut = RESET_TIMEOUT_COUNTER;
        //
        while (TimeOut) {
            if (XAxiDma_ResetIsDone(AxiDmaInst)) {
                break;
            }
        }
    }
}
```

```
//      }

//      TimeOut -= 1;

//      }

      return;
}

/*
 * If Completion interrupt is asserted, then set the TxDone flag
 */

if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {

    if(AxiDmaInst == &AxiDma0)
    {
        TxDone0++;
        if(rx0_buffer_index == 0)
            tx0_buffer_index = 2;
        else
            tx0_buffer_index = rx0_buffer_index - 1;

        Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[tx0_buffer_index],
                                         MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

        if (Status != XST_SUCCESS) {
            xil_printf("tx axi dma0 failed! 0 %d\r\n", Status);
            return;
        }
    }

    else if(AxiDmaInst == &AxiDma1)
    {
        TxDone1++;
        if(rx1_buffer_index == 0)
            tx1_buffer_index = 2;
        else
            tx1_buffer_index = rx1_buffer_index - 1;

        Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[tx1_buffer_index],
                                         MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

        if (Status != XST_SUCCESS) {
            xil_printf("tx axi dma1 failed! 0 %d\r\n", Status);
        }
    }
}
```

```
        return;
    }
}
else if(AxiDmaInst == &AxiDma2)
{
    TxDone2++;
    if(rx2_buffer_index == 0)
        tx2_buffer_index = 2;
    else
        tx2_buffer_index = rx2_buffer_index - 1;

    Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[tx2_buffer_index],
                                    MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

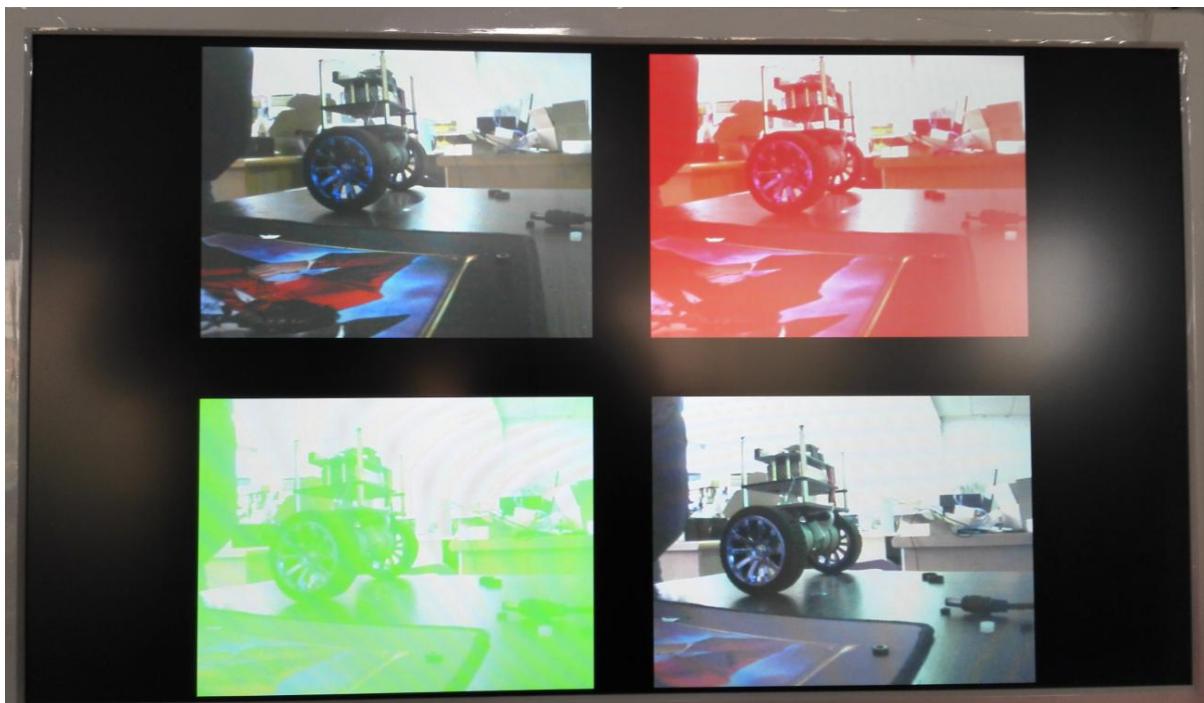
    if (Status != XST_SUCCESS) {
        xil_printf("tx axi dma2 failed! 0 %d\r\n", Status);
        return;
    }
}
else if(AxiDmaInst == &AxiDma3)
{
    TxDone3++;
    if(rx3_buffer_index == 0)
        tx3_buffer_index = 2;
    else
        tx3_buffer_index = rx3_buffer_index - 1;

    Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[tx3_buffer_index],
                                    MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

    if (Status != XST_SUCCESS) {
        xil_printf("tx axi dma3 failed! 0 %d\r\n", Status);
        return;
    }
}
else
    xil_printf("error!\r\n");
}
```

上表的发送中断函数，和接收中断函数处理机制一致，也是通过 AxiDmaInst 判断当前 DMA 的通道，并且为当前 DMA 通道发送数据，指定对应的 BUFFER。

10.6 测试结果



S03_CH11_基于 TCP 的 QSPI Flash bin 文件网络烧写

11.1 概述

针对 ZYNQ 中使用 QSPI BOOT 的应用，将 BOOT.bin 文件烧写至 QSPI Flash 基本都是通过 USB Cable 连接 PC，由 JTAG 口连接板卡后，在 SDK 软件中使用“Program Flash”功能进行现场在线烧写。然而，这种常规方法存在两个缺点。

速度慢。Flash 的擦除（Erase）、写入（Program）、校验（Verify）3 个过程所费的时间总和通常都需要若干分钟。

无法脱离 JTAG 口。对于某些产品而言，当产品量产上市后进入维护升级阶段，若需修改、更新 Flash 中的 bin 文件，则需对产品进行拆解才可进行。

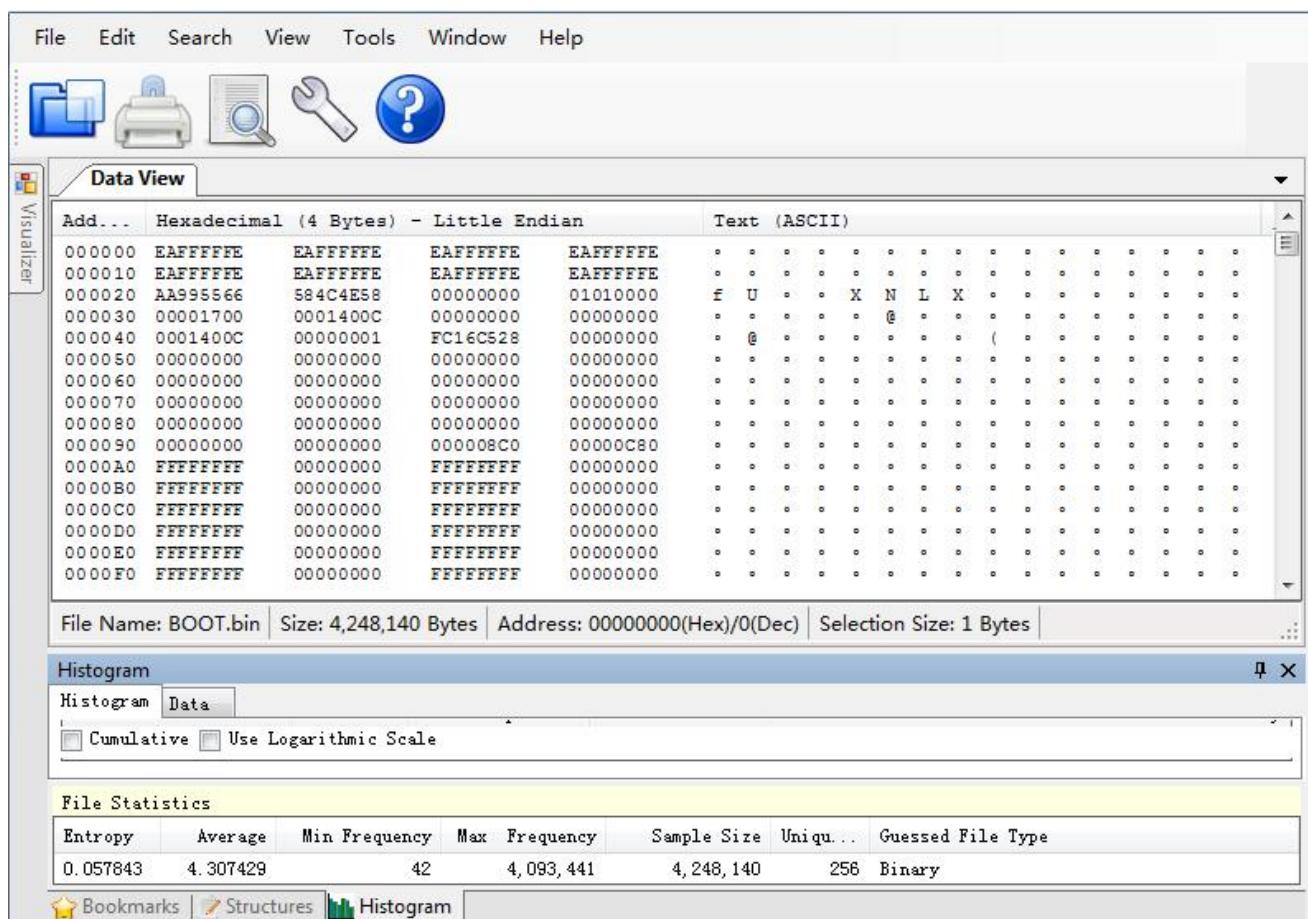
本例程实现了一种基于 TCP 协议的 Flash bin 文件更新方法。一方面，在较大程度上提高了 bin 文件的烧写速度（4MB 大小 bin 文件可缩短至 20 秒左右）；另一方面，提供了一种远程更新 Flash 的方法，可脱离 JTAG 接口，同时避免固件升级时对产品进行拆解。

11.2 基本原理

首先，在 ZYNQ 的 ARM 中基于 LWIP 库建立一个 TCP Server，板卡通过网线与电脑连接。在电脑中通过网络调试助手以 TCP Client 模式与 ZYNQ 中的 TCP Server 建立 TCP 连接。然后，通过网络调试助手将 BOOT.bin 文件以二进制形式发送至 TCP Server，并存储在 ZYNQ 所连接的 DDR 中。最后，当 TCP Server 接收完 bin 文件所有的数据后，网络调试助手发送烧写启动命令，将 bin 文件的数据按顺序一一连续写入 QSPI Flash 中，随后再全部读出与所接收的 bin 文件进行一一比对检验。断电重启板卡，便可验证 bin 文件的更新。

11.3 Bin 文件

在 SDK 中所生成的 BOOT.bin 文件为普通二进制文件，通过 UltraEdit 或 Binary Viewer 软件可打开并查看 bin 文件的内容。将 bin 文件中所有的数据按顺序一一连续写入 QSPI Flash 中，即可完成 bin 文件的烧写，也就是说 Flash 中的数据与 bin 文件中的数据完全是一一对应的。由此可见，烧写 bin 文件的原理并不复杂，不存在类似编码、解码的过程，且与 SDK 中的“Program Flash”的所完成的功能相同。使用 Binary Viewer 软件打开查看 bin 文件的效果如下图所示。



11.4 QSPI Flash

Flash 在写入数据之前必须先进行擦除（Erase），擦除过程以扇区（Sector）为单位。然后以页（Page）为单位，依次将数据写入 Flash 中连续的各页。

以米联 ZYNQ 开发板所使用的 QSPI Flash: S25FL256S 为例。Sector 的大小为 64KB，Page 的大小为 256B。另外，还存在两个重要参数：单位 Sector 擦除时间和单位 Page 写入时间，这决定了 Flash 的烧写速度。S25FL256S 所对应的单位 Sector 擦除时间为 130ms，单位 Page 写入时间 250 μ s。如下图最右侧的两栏所示。具体可参考芯片 datasheet。

Sector Erase Time (typ.)	30 ms (4 kB), 150 ms (64 kB)	500 ms (64 kB)	130 ms (64 kB), 520 ms (256 kB)
Page Programming Time (typ.)	700 μs (256B)	1500 μs (256B)	250 μs (256B), 340 μs (512B)

以 4MB 大小的 BOOT.bin 文件为例，写入之前需要擦除 $4096/64 = 64$ 个 Sector，最短需耗时 $64 \times 130\text{ms} = 8.32\text{s}$ 。接着，需要写入 $4096 \times 1024/256 = 16384$ 个 Page，最短需耗时 $16384 \times 250 \mu\text{s} = 4.096\text{s}$ 。加上 ARM 中应用程序的软件开销，QSPI Flash 的擦除、写入时间总和不超过 15s。若需读出进行校验，则再额外增加读出时间、比对时间。QSPI Flash 的读出速度很快，读出整个 bin 文件耗时小于 1s，ARM 中应用程序的比对时间通常也很短，1 至 2s 即可完成。因此，对于 4MB 大小的 bin 文件，QSPI Flash 的擦除（Erase）、写入（Program）、校验（Verify）3 个过程所耗费的时间总和可以

控制在 20s 以内。

11.5 驱动程序

所有的驱动程序文件均包含在 c_driver 文件夹中的 tcp 文件夹。

main 函数的完成的功能如下：

- 关闭 Data Cache，避免 bin 文件在 DDR 拷贝过程中维护 Cache 一致性造成的麻烦
- 配置 QSPI 接口及 QSPI Flash
- 配置 Timer 及其中断
- 初始化中断控制器及系统中断
- 初始化 LWIP 协议栈
- 建立 TCP Server，启动 Timer
- 持续从 LWIP 协议栈接收数据

11.5.1 建立 TCP Server

基于 LWIP 库在 ARM 中建立一个 TCP Server，IP 地址为 192.168.1.10，端口号为 5010。

11.5.2 lwip 库设置

本例程使用 RAW API，即函数调用不依赖操作系统。传输效率也比 SOCKET API 高，(具体可参考 xapp1026)。将 use_axieth_on_zynq 和 use_emaclite_on_zynq 设为 0。如下图所示。

Configuration for library: lwip141				
Name	Value	Default	Type	Description
api_mode	RAW_API	RAW_API	enum	Mode of operation for lwIP (1=SOCKET, 2=RAW)
socket_mode_thread_prio	2	2	integer	Priority of threads in socket receiver
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axieth interface is used
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures emaclite interface is used

修改 lwip_memory_options 设置，将 mem_size, memp_n_pbuf, mem_n_tcp_pcb, memp_n_tcp_seg 这 4 个参数值设大，这样会提高 TCP 传输效率。如下图所示。

lwip_memory_options					Options controlling lwIP memory usage
mem_size	10485760	131072	integer		Size of the heap memory
memp_n_pbuf	2048	16	integer		Number of memp struct pointers
memp_n_sys_timeout	8	8	integer		Number of simultaneously active sys timeouts
memp_n_tcp_pcb	1024	32	integer		Number of active TCP PCBs
memp_n_tcp_pcb_listen	8	8	integer		Number of listening TCP connections
memp_n_tcp_seg	1024	256	integer		Number of simultaneously allocated TCP segments
memp_n_udp_pcb	4	4	integer		Number of active UDP PCBs
memp_num_api_msg	16	16	integer		Number of api msg structures
memp_num_netbuf	8	8	integer		Number of struct netbufs
memp_num_netconn	16	16	integer		Number of struct netconn
memp_num_tcip_msg	64	64	integer		Number of tcip msg structures

修改 pbuf_options 设置，将 pbuf_pool_size 设大，增加可用的 pbuf 数量，这样同样会提高 TCP 传输效率。如下图所示。

pbuf_options					Pbuf Options
pbuf_link_hlen	true	16	integer		Number of bytes that should be aligned to
pbuf_pool_bufsize	1700	1700	integer		Size of each pbuf in pbuf pool
pbuf_pool_size	4096	256	integer		Number of buffers in pbuf pool

修改 tcp_options 设置，将 tcp_snd_buf, tcp_wnd 参数设大，这样同样会提高 TCP 传输效率。如下图所示。

tcp_options					Is TCP required ?
lwip_tcp	true	true	boolean		Is TCP required ?
tcp_maxrtx	12	12	integer		TCP Maximum retransmission count
tcp_mss	1460	1460	integer		TCP Maximum segment size (bytes)
tcp_queue_ooseq	1	1	integer		Should TCP queue segments out of order
tcp_snd_buf	65535	8192	integer		TCP sender buffer space (bytes)
tcp_synmaxrtx	4	4	integer		TCP Maximum SYN retransmission count
tcp_ttl	255	255	integer		TCP TTL value
tcp_wnd	65535	2048	integer		TCP Window (bytes)

修改 temac_adapter_options 设置，将 n_rx_descriptors 和 n_tx_descriptors 参数设大。这样可以提高 zynq 内部 emac dma 的数据搬移效率，同样能提高 TCP 传输效率。如下图所示。

temac_adapter_options					Settings for xps-ll-temac/Adreno
emac_number	0	0	integer		Zynq Ethernet Interface number
n_rx_coalesce	1	1	integer		Setting for RX Interrupt coalescing
n_rx_descriptors	256	64	integer		Number of RX Buffer Descriptors
n_tx_coalesce	1	1	integer		Setting for TX Interrupt coalescing
n_tx_descriptors	256	64	integer		Number of TX Buffer Descriptors
phy_link_speed	CONFIG_LINKSPEED_AUTO	CONFIG_LINKSPEED_AUTO	enum		link speed as negotiated by PHY
tcp_ip_rx_checksum_offload	false	false	boolean		Offload TCP and IP Receive checksums
tcp_ip_tx_checksum_offload	false	false	boolean		Offload TCP and IP Transmit checksums
tcp_rx_checksum_offload	false	false	boolean		Offload TCP Receive checksums
tcp_tx_checksum_offload	false	false	boolean		Offload TCP Transmit checksums
temac_use_jumbo_frames	false	false	boolean		use jumbo frames

其余选项的参数默认即可，不用修改。

11.5.3 程序解析

TCP Server 建立由 `tcp_transmission.c` 文件中的 `tcp_recv_init` 和 `connect_accept_callback` 函数完成。

- tcp_recv_init 函数：
 - 调用 tcp_new 函数建立 1 个建立 TCP 连接所需的结构体。
 - 调用 tcp_bind 函数绑定 TCP Server 的本地 IP 地址和 TCP 端口号。
 - 调用 tcp_listen 函数启动监听外部任何 TCP Client 向 TCP Server 发送的 TCP 连接请求。
 - 调用 tcp_accept 函数指定当 TCP Server 与外部 TCP Client 建立 TCP 连接后的回调函数为 connect_accept_callback。
- connect_accept_callback 函数：
 - 当 ARM 中建立的 TCP Server 与外部 TCP Client 建立连接后，connect_accept_callback 函数将会被调用。
 - 调用 tcp_recv 函数指定当 TCP Server 接收来自 TCP Client 所发送数据包的回调函数为 tcp_recv_callback。
 - 调用 tcp_sent 函数指定当 TCP Server 向 TCP Client 发送数据包时的回调函数为 tcp_sent_callback，在例程 tcp_sent_callback 为空函数，无实际作用。

11.5.4 接收保存 BOOT.bin 文件

接收 BOOT.bin 文件通过 tcp_transmission.c 中的 tcp_recv_callback 函数完成，该函数为 TCP Server 接收数据包的回调函数，每当接收到 TCP Client 的数据包时该函数都会被调用。该函数将 TCP Server 所接收到的 bin 文件的数据包依次拷贝至 DDR 中首地址为 FILE_BASE_ADDR 的区域中，FILE_BASE_ADDR 为宏定义。

```
#define FILE_BASE_ADDR 0x10000000
```

可根据具体要求定义其地址，注意要与 lscript.ld 中的 DDR 区域分开，不能重合。

11.5.5 烧写 QSPI Flash

烧写 QSPI Flash 由 qspi_g128_flash.c 文件中的 update_flash、FlashErase、FlashWrite、FlashRead 等函数完成，可支持 Micron、Spansion 等多个厂商，128Mb 以上多种容量的 QSPI Flash。qspi_g128_flash.c 是根据 SDK 中 QSPI 接口的 example:xqspips_g128_flash_example.c 修改而成。

当接收完整个 bin 文件后，在网络调试助手中输入“start update”，含空格一共 12 个字符。tcp_recv_callback 函数接收到该命令之后便调用 update_flash 函数启动 bin 文件至 QSPI Flash 的烧写。该函数需要 1 个读缓存和 1 个写缓存，分别存放从 flash 中读出的 bin 文件数据和需要写入 flash 中的 bin 文件数据。读缓存和写缓存的地址分别由 READ_BASE_ADDR 和 WRITE_BASE_ADDR 宏定义指定。可根据具体要求定义其地址，同样需要注意要与 lscript.ld 中的 DDR 区域分开，不能重合。

```
#define READ_BASE_ADDR 0x11000000  
#define WRITE_BASE_ADDR 0x12000000
```

update_flash 函数：

- 将 TCP Server 接收的 bin 文件数据拷贝至写缓存中起始地址为 WRITE_BASE_ADDR + 4 的区域中，增加 4 字节的地址偏移是由于在 FlashWrite 函数中，写缓存的前 4 字节将被用于填充命令和写入地址字段。

- 调用 FlashErase 函数将 bin 文件数据所对应数量的扇区擦除。
- 调用 FlashWrite 将 bin 文件数据以页为单位依次全部写入 Flash 中。
- 调用 FlashRead 函数将刚才写入 Flash 的 bin 文件全部读出存入读缓存中。
- 将读出的 bin 文件数据与 TCP Server 接收的原始 bin 文件数据进行一一比对验证烧写的正确性。

FlashErase、FlashWrite、FlashRead 函数均源自于 SDK 中 QSPI 接口的 example code。可参考相应的 example 具体分析函数功能。

11.5.6 TCP 调试信息输出

例程中设计了一个 `tcp_printf` 函数，用于向网络调试助手输出字符串调试信息。该函数暂时只支持字符串以 “\n” 结尾。

在本例程中使用该函数存在一个问题，即在 Flash 烧写开始直至结束前所有通过 `tcp_printf` 输出的调试信息，将无法及时发送，在烧写结束后应用程序回到 `main` 函数中包含 `xemacif_input(netif)` 函数的 `while` 循环中时，同时全部通过 TCP 发送至网络调试助手。

笔者尝试过几种方法均未能解决这个问题，例如通过 `tcp_nagle_disable()` 函数关闭 nagle 算法功能。笔者认为，一方面，可能跟 LWIP 库 TCP 部分函数的设计原理有关，另一方面，由于 flash 烧写部分应用程序将长时间占用 ARM，使得 `xemacif_input(netif)` 函数长时间无法被调用，而 ZYNQ 中的 LWIP 协议栈所有的数据接收都需要依靠应用程序调用 `xemacif_input()` 函数而实现。因此在这段时间中，可能由于长时间无法调用该函数而对 TCP 部分函数的运行造成某些影响，使数据发送产生阻塞或者延迟。由于 LWIP 中 TCP 部分库函数结构较复杂，笔者对于 TCP 协议研究也不多，限于时间和精力未作深究。

11.6 网络调试助手操作方法

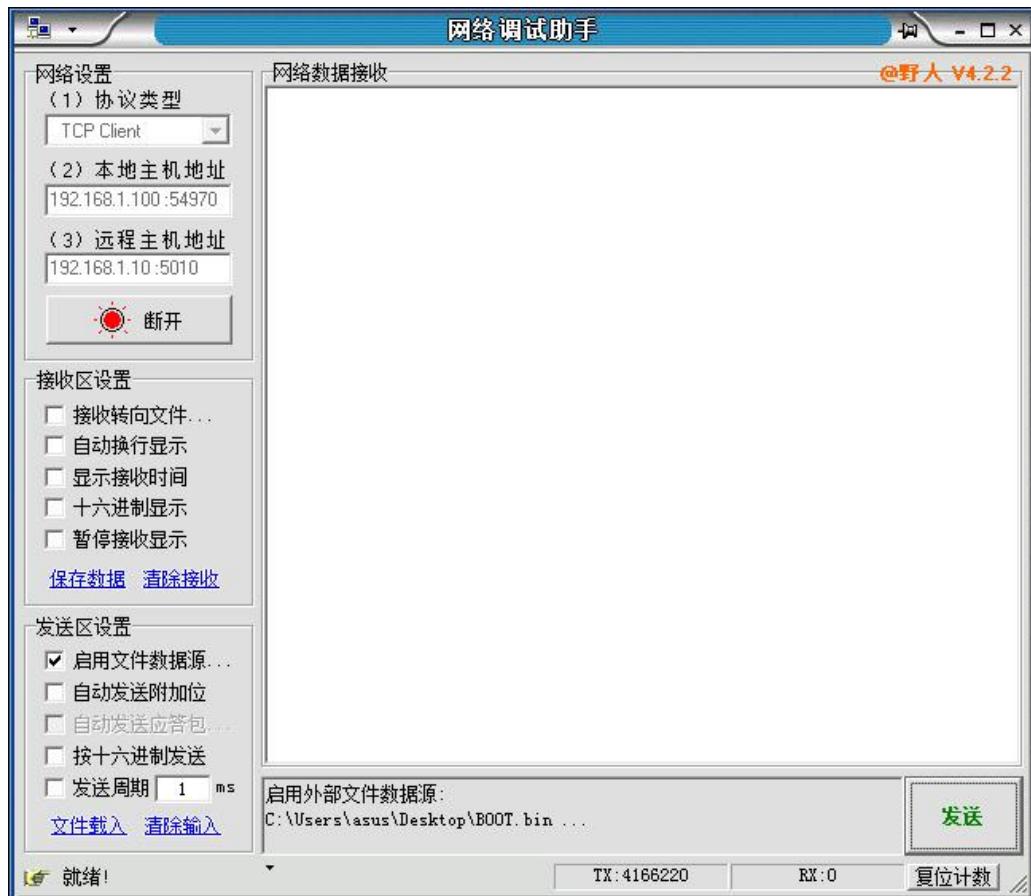
11.6.1 发送 bin 文件

在 SDK 中下载程序至 ZYNQ 中。打开网络调试助手，选择 TCP Client 方式，输入 ARM 中定义的 TCP Server 的 IP 地址和端口号，然后点击连接按键，建立 TCP 连接。SDK 串口终端打印信息如下图所示。

The screenshot shows a terminal window titled "Connected to: Serial (COM10, 115200, 0, 8)". The text output is as follows:

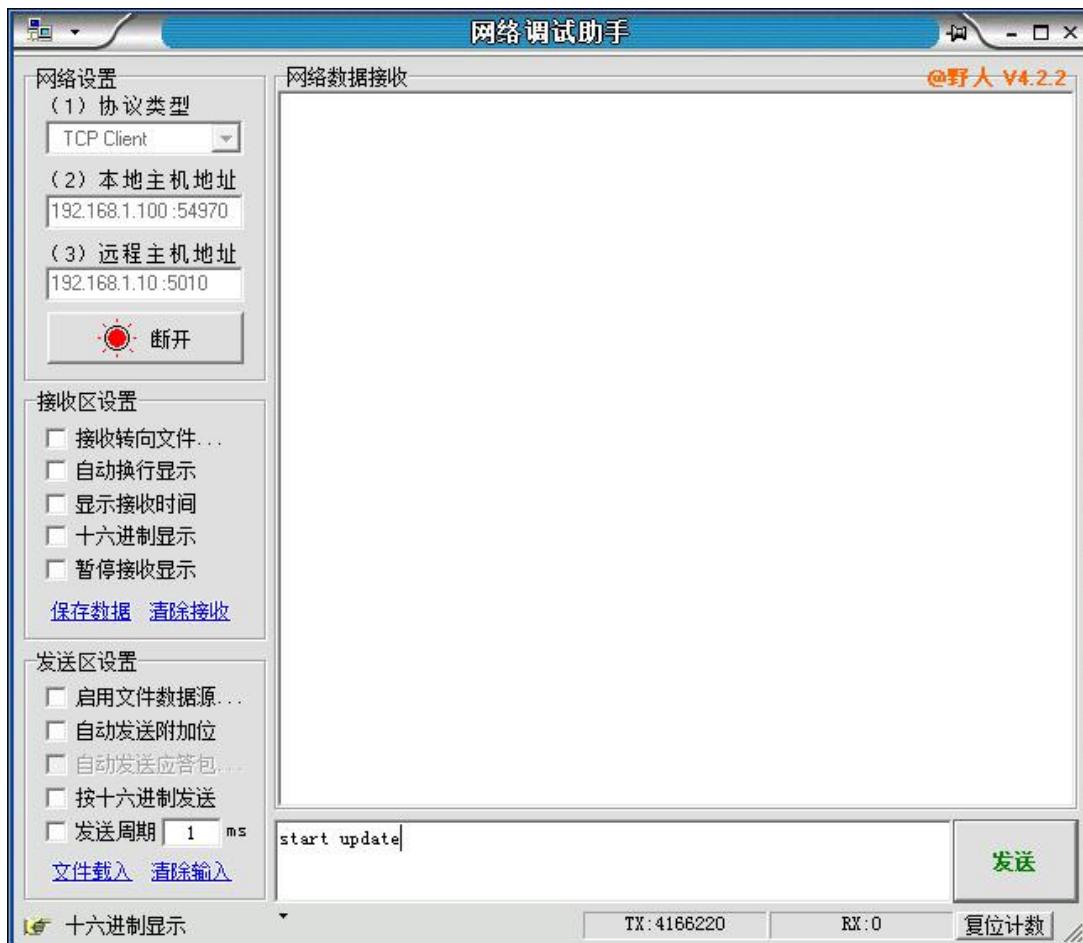
```
FlashID=0x1 0x2 0x19
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
tcp_server: Connection Accepted
```

在网络调试助手发送区设置里选择“启用文件数据源”，选择需要发送的 BOOT.bin 文件，然后点击发送。如下图所示。



11.6.7 发送启动 Flash 烧写命令

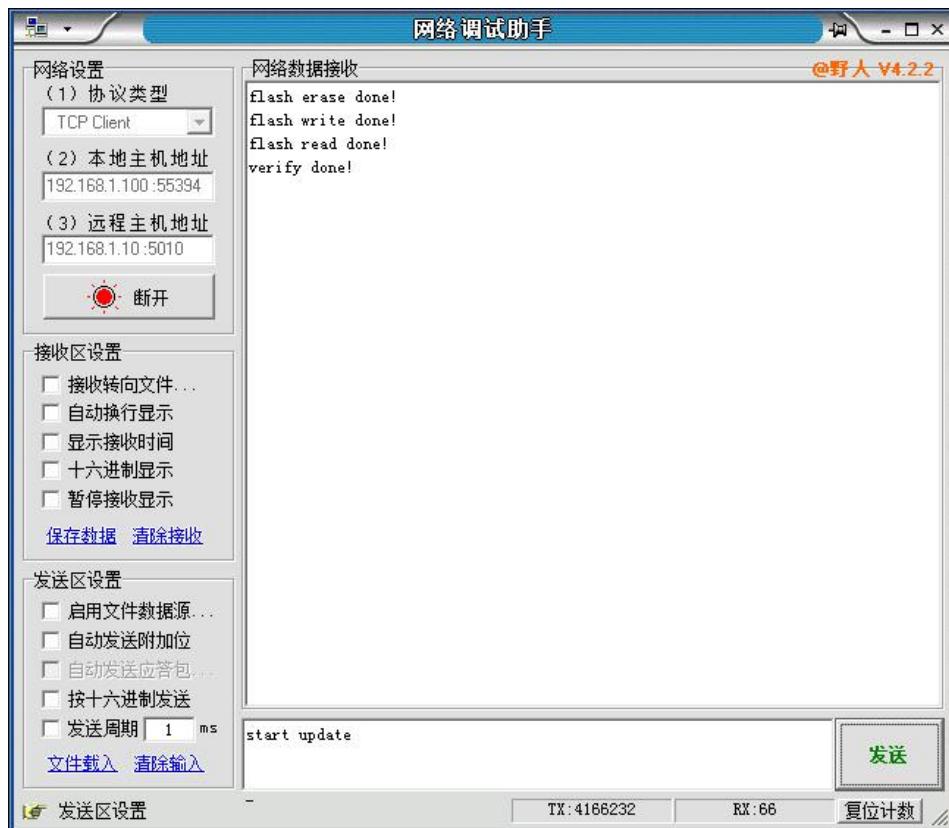
然后输入烧写启动命令“start update”，不要选择“按十六进制发送”，本例程中需要以 ASCII 码形式发送，含空格一共 12 个字符（不要在末尾加回车），千万不要输错，否则需要全部重新再来一遍。如下图所示。



启动烧写后，SDK 串口终端打印信息如下图所示。当提示“verify done!”表示整个烧写过程成功完成。

```
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
tcp_server: Connection Accepted
flash update start!
file length of BOOT.bin is 4166220 Bytes
flash erase done!
flash write done!
flash read done!
verify done!
```

网络调试助手接收 `tcp_printf` 函数的输出的信息如下图所示。



11.7 Bin 文件更新验证

烧写完成后，此时可断电重启验证更新的 BOOT.bin 文件。本例程作为演示，烧入的 bin 文件为 hello world 工程。重启开发板，SDK 串口终端打印信息如下图所示，代表 bin 文件更新成功。



11.8 待改进之处

- `tcp_printf` 函数在 flash 烧写过程中无法及时发送数据的问题有待解决。
- 无校验差错重新写入功能。当将 bin 文件写入 Flash 之后再读出校验时，若出现错误，则需将错误的 Page 重新写入。
- 由于例程中使用网络调试助手作为传输 BOOT.bin 文件的工具，仅使用了 TCP 协议，无法在 TCP 协议之上设计自定义协议来规范并完善 bin 文件的传输过程，用户可控性较差。例如，启动命令“`start update`”一旦输入错误并发送至 ZYNQ，则板卡将需断电重启，整个过程需从头重来一遍。理想情况下，用户应根据实际需求在 TCP 之上设计相应的 bin 文件传输协议（例如给每一个 bin 文件数据包加上含有协议信息的包头、包尾等）来提高传输的可靠性和可控性。另外，用户还需设计与之相对应的上位机软件进行 bin 文件的发送，以及与 ZYNQ 的通信。

S03_CH12_基于 UDP 的 QSPI Flash bin 文件网络烧写

12.1 概述

为了满足不同的需求，本例程在“基于 TCP 的 QSPI Flash bin 文件网络烧写”上进行修改，将 bin 文件的传输协议替换为 UDP。与采用 TCP 协议的例程相比，本例程无需使用 ZYNQ 内部的定时器，无定时器中断，LWIP 中 UDP 部分的 API 函数结果也更为简洁，易于使用，简化了 ARM 中的 C 程序设计，但使用 UDP 协议后文件传输的可靠性无法保证，因此需要更具实际应用进行权衡。

本例程基于 Vivado 2015.4 版本开发。

12.2 基本原理

与“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程相同，并将 TCP 协议替换为 UDP 协议。

12.2.1 Bin 文件

见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

12.2.2 QSPI Flash

见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

12.3 驱动程序

所有的驱动程序文件均包含在 c_driver 文件夹中的 udp 文件夹。

12.3.1 main 函数

main 函数的完成的功能如下：

- 关闭 Data Cache，避免 bin 文件在 DDR 拷贝过程中维护 Cache 一致性造成的麻烦
- 配置 QSPI 接口及 QSPI Flash
- 初始化中断控制器及系统中断
- 初始化 LWIP 协议栈
- 建立 UDP 连接
- 持续从 LWIP 协议栈接收数据

12.3.2 建立 UDP 连接

基于 LWIP 库在 ARM 中建立一个 UDP 连接，ZYNQ 的 IP 地址为 192.168.1.10，端口号为 5010，远程 IP 地址为 192.168.1.100，端口号为 8080。

12.3.3 lwip 库设置

本例程使用 RAW API，即函数调用不依赖操作系统。传输效率也比 SOCKET API 高，(具体可参考 xapp1026)。将 use_axieth_on_zynq 和 use_emaclite_on_zynq 设为 0。如下图所示。

Configuration for library: lwip141				
Name	Value	Default	Type	Description
api_mode	RAW_API	RAW_API	enum	Mode of operation for lwIP (I)
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axi
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures em

修改 lwip_memory_options 设置，将 mem_size，memp_n_pbuf 这 2 个参数值设大，这样会提高 UDP 传输效率。如下图所示。

lwip_memory_options					Options control
mem_size	10485760	131072	integer	Size of the heap	Options control
memp_n_pbuf	2048	16	integer	Number of men	Options control
memp_n_sys_timeout	8	8	integer	Number of simu	Options control
memp_n_tcp_pcb	32	32	integer	Number of activ	Options control
memp_n_tcp_pcb_listen	8	8	integer	Number of liste	Options control
memp_n_tcp_seg	256	256	integer	Number of simu	Options control
memp_n_udp_pcb	4	4	integer	Number of activ	Options control
memp_num_api_msg	16	16	integer	Number of api	Options control
memp_num_netbuf	8	8	integer	Number of struc	Options control
memp_num_netconn	16	16	integer	Number of struc	Options control
memp_num_tcip_msg	64	64	integer	Number of tcipi	Options control

修改 pbuf_options 设置，将 pbuf_pool_size 设大，增加可用的 pbuf 数量，这样同样会提高 UDP 传输效率。如下图所示。

pbuf_options					Pbuf Options
pbuf_link_hlen	16	16	integer	Number of bytes that should	Pbuf Options
pbuf_pool_bufsize	1700	1700	integer	Size of each pbuf in pbuf po	Pbuf Options
pbuf_pool_size	4096	256	integer	Number of buffers in pbuf po	Pbuf Options

由于无需使用 TCP 协议，修改 tcp_options 设置，将 lwip_tcp 设置为 false，tcp_queue_ooseq 设为 0，关闭 tcp 功能，如下图所示。

tcp_options					Is TCP required ?
lwip_tcp	false	true	boolean	Is TCP required ?	Is TCP required ?
tcp_maxrtx	12	12	integer	TCP Maximum retran	Is TCP required ?
tcp_mss	1460	1460	integer	TCP Maximum segme	Is TCP required ?
tcp_queue_ooseq	0	1	integer	Should TCP queue se	Is TCP required ?
tcp_snd_buf	8192	8192	integer	TCP sender buffer sp	Is TCP required ?
tcp_synmaxrtx	4	4	integer	TCP Maximum SYN re	Is TCP required ?
tcp_ttl	255	255	integer	TCP TTL value	Is TCP required ?
tcp_wnd	2048	2048	integer	TCP Window (bytes)	Is TCP required ?

修改 temac_adapter_options 设置，将 n_rx_descriptors 和 n_tx_descriptors 参数设大。

这样可以提高 zynq 内部 emac dma 的数据搬移效率，同样能提高 UDP 传输效率。如下图所示。

temac_adapter_options	true	true	boolean	Settings for xps-ll-temac/Ad
emac_number	0	0	integer	Zynq Ethernet Interface num
n_rx_coalesce	1	1	integer	Setting for RX Interrupt coa
n_rx_descriptors	256	64	integer	Number of RX Buffer Descr
n_tx_coalesce	1	1	integer	Setting for TX Interrupt coa
n_tx_descriptors	256	64	integer	Number of TX Buffer Descr
phy_link_speed	CONFIG_LINKSPEED_A...	CONFIG_LINKSPEED_A...	enum	link speed as negotiated by
tcp_ip_rx_checksum_offload	false	false	boolean	Offload TCP and IP Receive
tcp_ip_tx_checksum_offload	false	false	boolean	Offload TCP and IP Transm
tcp_rx_checksum_offload	false	false	boolean	Offload TCP Receive checks
tcp_tx_checksum_offload	false	false	boolean	Offload TCP Transmit check
temac_use_jumbo_frames	false	false	boolean	use jumbo frames

其余选项的参数默认即可，不用修改。

12.3.4 程序解析

UDP 连接建立由 `udp_transmission.c` 文件中的 `udp_recv_init` 函数完成。

`udp_recv_init` 函数：

调用 `udp_new` 函数建立 1 个建立 UDP 连接所需的结构体。

调用 `udp_bind` 函数绑定本地 IP 地址和 UDP 端口号。

调用 `udp_connect` 函数建立 UDP 连接，并绑定远程 IP 地址和 UDP 端口号。

调用 `udp_recv` 函数指定用于接收 UDP 数据包的回调函数为 `udp_recv_callback`。

12.3.5 接收保存 BOOT.bin 文件

接收 `BOOT.bin` 文件通过 `udp_transmission.c` 中的 `udp_recv_callback` 函数完成，该函数为 UDP 接收数据包的回调函数，每当接收到 UDP 的数据包时该函数都会被调用。保存位置与“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程相同。

12.3.6 烧写 QSPI Flash

见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

12.3.7 UDP 调试信息输出

例程中设计了一个 `udp_printf` 函数，用于向网络调试助手输出字符串调试信息。该函数暂时只支持字符串以“\n”结尾。

该函数实现思路与“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程中 `tcp_printf` 函数基本相同。不同的是，任何时刻通过 `udp_printf` 发送的数据都会及时通过网络发出，并被网络调试助手接收，而不像 `tcp_printf` 函数的发送会在 flash 烧写过程中被阻塞。

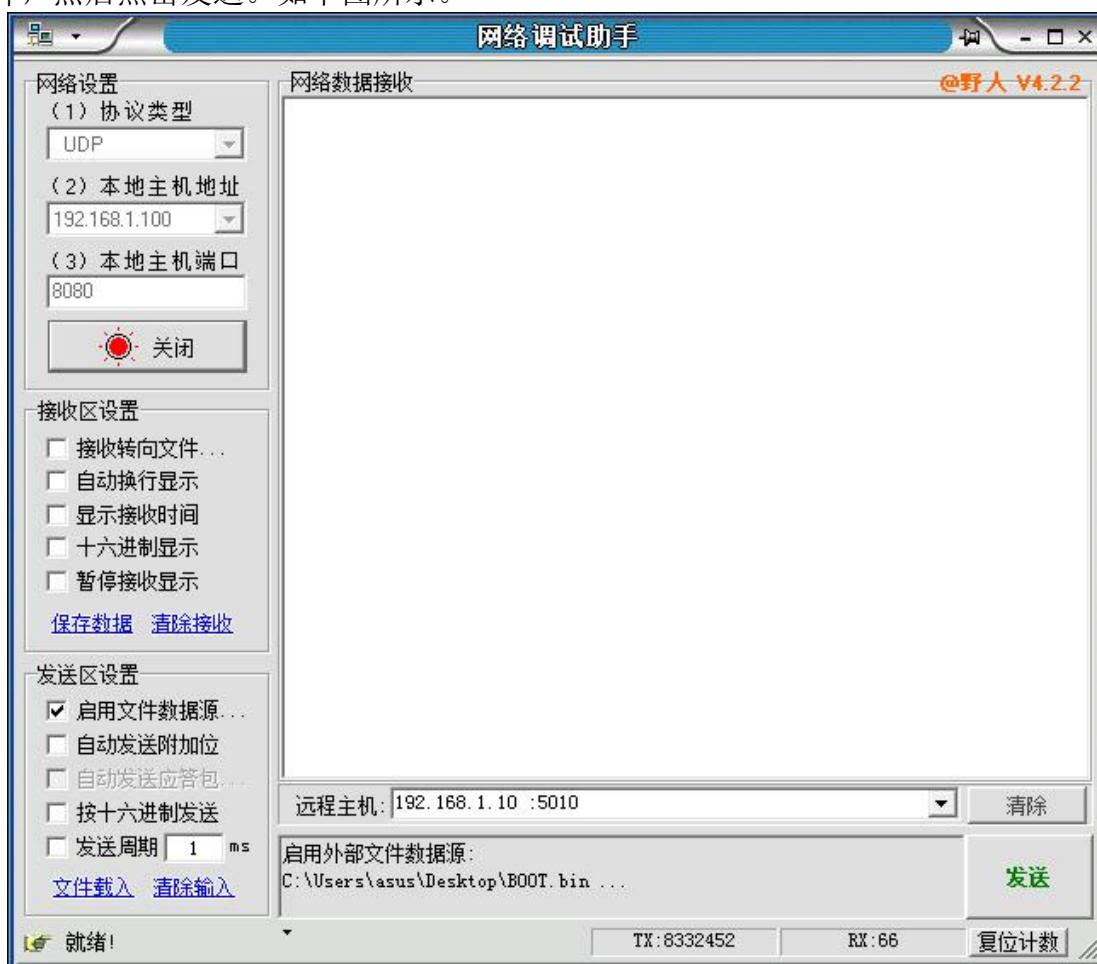
这是由于 UDP 为不可靠传输，不像 TCP 在传输过程中需进行持续握手。因此，UDP

接收和发送过程相互独立，具体在 LWIP 协议栈中反应为 udp_send 函数不依赖 xemacif_input 函数。

12.4 网络调试助手操作方法

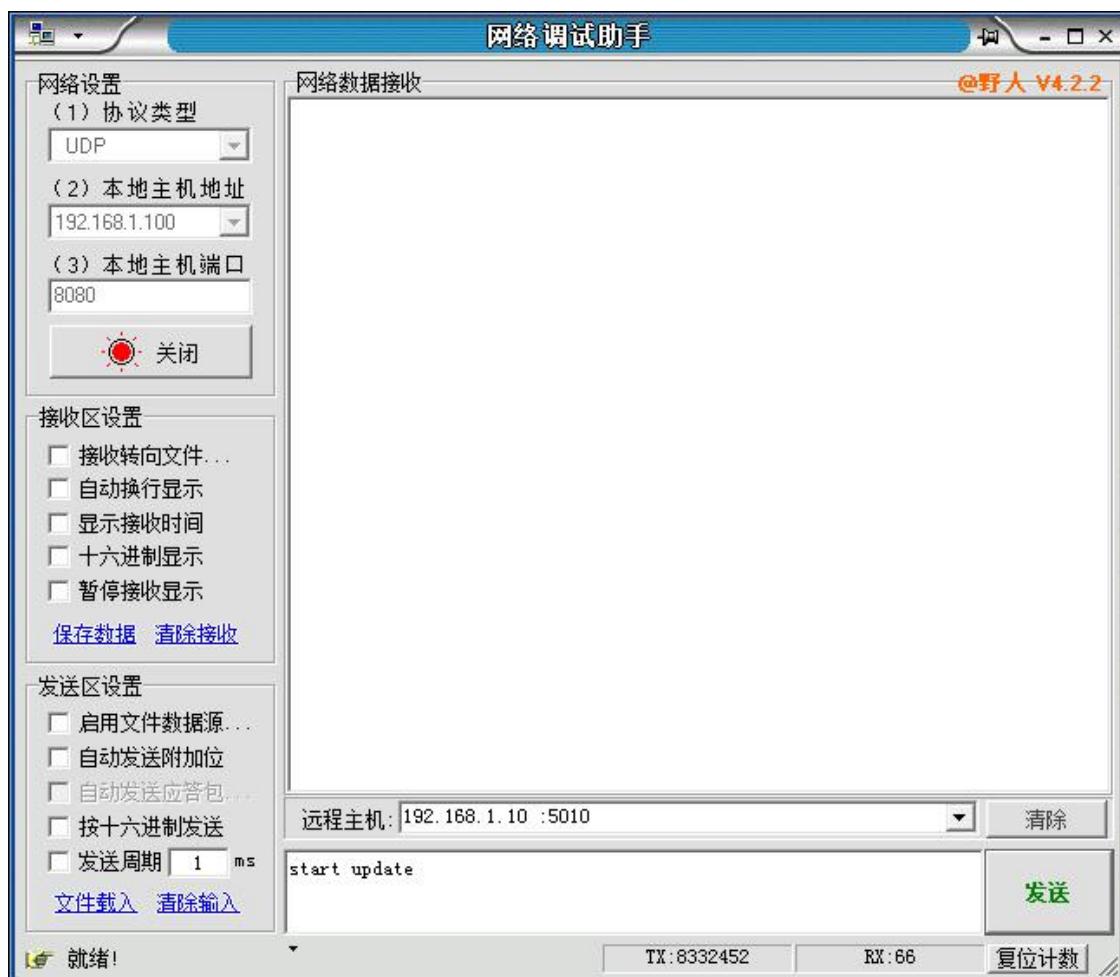
12.4.1 发送 bin 文件

在 SDK 中下载程序至 ZYNQ 中。打开网络调试助手，选择 UDP 方式，输入电脑的 IP 地址和 UDP 端口号，然后点击打开按键，在远程主机中填入 ZYNQ 的 IP 地址和 UDP 端口号。在网络调试助手发送区设置里选择“启用文件数据源”，选择需要发送的 BOOT.bin 文件，然后点击发送。如下图所示。



12.4.2 发送启动 Flash 烧写命令

然后输入烧写启动命令“start update”，不要选择“按十六进制发送”，本例程中需要以 ASCII 码形式发送，含空格一共 12 个字符（不要在末尾加回车），千万不要输错，否则需要全部重新再来一遍。如下图所示。

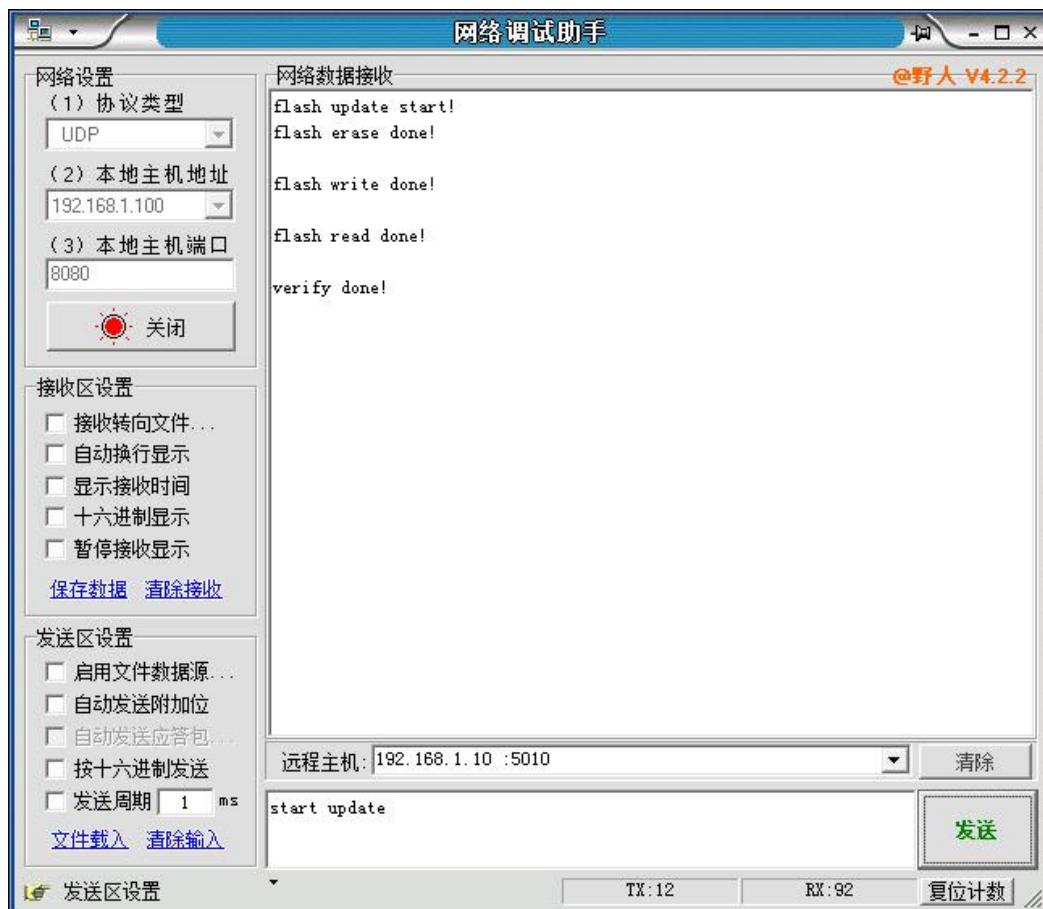


启动烧写后，SDK 串口终端打印信息如下图所示。当提示“verify done!”表示整个烧写过程成功完成。

```
Connected to: Serial ( COM10, 115200, 0, 8 )
FlashID=0x1 0x2 0x19

Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
flash update start!
file length of BOOT.bin is 4166220 Bytes
flash erase done!
flash write done!
flash read done!
verify done!
```

网络调试助手接收 `udp_printf` 函数实时输出的信息如下图所示。



12.5 Bin 文件更新验证

见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

12.6 待改进之处

- UDP 为不可靠传输，为了提高使用 UDP 协议传输 bin 文件的可靠性，可在其之上设计额外的传输协议，并加上类似传输握手的功能。
- 其余见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

S03_CH13_ZYNQ A9 TCP UART 双核 AMP 例程

13.1 概述

ZYNQ 中存在两个独立的 ARM 核，在很多应用场景中往往只需使用其中的 1 个核心即可。然而，对于复杂的设计，例如多任务，并行控制、处理等，单个核心将难以胜任。因此，为了尽可能发挥 ZYNQ 中双 ARM 核的优势和性能，进行双核应用的开发显得尤为重要。同时，也进一步为 Xilinx 下一代 MPSOC 多核异构处理器的使用打下基础。

在 ZYNQ 中实现双 ARM 核 AMP 应用可以参考 Xilinx 官方的 XAPP1078 和 XAPP1079。在 SDK 中也有用于双核应用开发的 openamp 库可以使用。

本例程未使用 openamp 库，通过自行设计的代码实现了一个简单的双核应用，旨在说明：

- 双核通过软件中断进行核间通信的原理及方法。
- 双核通过共享内存进行数据交互的基本原理和设计方法。
- 双核协同工作的基本模式。
- 双核 BOOT 的方法。

本例程基于 vivado 2015.4 开发。

13.2 基本原理

本例程在 ARM 核的 CORE0 建立一个 TCP Server，CORE0 通过 TCP Server 从外部 TCP Client 接收数据。当 CORE0 接收到 1 个 TCP 包后，将数据复制到 DDR3 中的缓存区域内，然后将数据信息（长度、首地址）放入 CORE0 和 CORE1 在 DDR3 的共享内存中。

接着，CORE0 通过触发软件中断通知 CORE1 数据信息已存入共享内存中，CORE1 接到中断后从共享内存读出数据信息，并将对应长度的数据复制到缓存区中，然后通过 UART 将数据输出。当 CORE1 通过串口完成数据输出后，同样通过触发软件中断通知 CORE0 数据发送已完成，CORE0 接到中断后通过串口打印完成信息，然后开始接收下一个 TCP 包，重复上述过程。

最后，通过 FSBL 实现双核的 QSPI BOOT。

13.2.1 软件中断

在 UG585 的 Interrupts 部分 7.2.1 章节可以找到关于软件中断（SGI）的说明。简而言之，软件中断就是 CPU 自己产生的中断，可用于触发自身和其他 CPU。ZYNQ 中共有 16 个软件中断可以使用，对应的中断号为 0~15，如下图所示。通过软件中断可以实现 CPU 之间的相互通信。本例程使用了编号 1 和 2 的软件中断。

Table 7-2: Software Generated Interrupts (SGI)

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	

13.2.2 共享内存通信

所谓共享内存就是，CORE0 和 CORE1 在 DDR3 内存中约定一块地址及长度已知的内存区域。然后，两者之间便可通过这片区域进行数据的传递。

两个核心各自拥有独立的 L1 DCache，并且共享同一个 L2 DCache，在 ZYNQ 中存在一个 Snoop Control Unit (SCU) 用于维护 CORE0 和 CORE1 的 L1 DCache 与 L2 DCache 之间的一致性，无需用户干预。因此，虽然 CORE0 和 CORE1 的共享内存区域位于 DDR 中，两者之间的数据传递并不需要考虑 DCache 一致性的维护。但是，为了更好阐明多级存储器结构的特性以及 DCache 一致性维护的问题，本例程在两核间通过 DDR3 共享内存进行数据交互时加入了 DCache 一致性操作，最终达到的效果与不使用 DCache 一致性操作时相同。

DCache 一致性维护的原理为：在多级存储器结构中，CPU 通过 1 级或多级 Cache 与 DDR 产生连接，CPU 本身不直接访问 DDR，而是通过 Cache 访问 DDR。Cache 中始终会暂存一小部分（通常是 KB~几 MB 量级）CPU 最近访问的 DDR 某些地址区域中的数据。因此，在应用程序中对 DDR 进行读或写操作，实际上都是 CPU 对 Cache 进行读或写操作。当 DDR 中某个地址范围内的数据突然被除 CPU 以外的 Master（如 DMA）改变时，若此时 Cache 中保存了这些区域的数据，且这些数据在 Cache 中状态为有效时，当 CPU 需要再次读取 DDR 这片区域的数据时，就不会让 Cache 去读取 DDR 中此区域内最新的数据来更新 Cache，再从 Cache 里读取最新的数据，而是直接从 Cache 中读取原来的旧数据，显然这不是我们所期望的结果。这时，便引入了所谓的 Cache 一致性问题。

ZYNQ 中存在 ICache 和 DCache，ICache 用于缓存可执行程序，DCache 用于缓存数据。一般情况下，用于保存可执行程序的 DDR 地址范围不会被除 CPU 以外的对象访问。因此，一般不存在 ICache 的一致性问题。而 DCache 在很多应用中却经常会被除 CPU 以外的对象访问，所以存在一致性问题。

ZYNQ 中维护 DCache 一致性的方法为：写入方将数据写入 DDR 对应地址区域后，需将残留在 DCache 中相应地址范围内的数据全部刷入 DDR3 中。读取方在从 DDR 相应地址读取数据之前，需将 DCache 中 DDR 相应地址范围内的数据全部设置为 invalid，然后 CPU 会再次通过 DCache 从 DDR3 中读取该地址范围内最新的数据。

13.2.3 双核 BOOT

裸机双核 BOOT 的方法参考自 Xilinx 的 XAPP1079。首先，通过 FSBL 实现 CORE0

的 BOOT。当 CORE0 启动进入 main 函数后，配合 FSBL 再实现 CORE1 的启动。
具体原理参考 XAPP1079，此处不作赘述。

13.3 驱动程序

CORE0 工程的驱动程序文件位于 c_driver 文件夹中的 core0 文件夹中，CORE1 工程的驱动程序文件位于 c_driver 文件夹中的 core1 文件夹中。需要说明的是，core0 和 core1 的工程在 DDR 所占用的地址区域进行如下划分：

- CORE0: 0x00100000~0x01FFFFFF，见下图 core0 的 lscript.ld 文件设置。

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x100000	0x1F00000
ps7_qspi_linear_0_S_AXI_BASEADDR	0xFC000000	0x1000000
ps7_ram_0_S_AXI_BASEADDR	0x0	0x30000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0xFE00

- CORE1: 0x02000000~0x02FFFFFF，见下图 core1 的 lscript.ld 文件设置。

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x2000000	0x1000000
ps7_qspi_linear_0_S_AXI_BASEADDR	0xFC000000	0x1000000
ps7_ram_0_S_AXI_BASEADDR	0x0	0x30000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0xFE00

设计双核应用，两个工程的内存分配是一个关键的前提，程序所占用的 DDR 空间不能发生重合，应完全分隔开。

13.3.3 CORE0 工程

13.3.3.1 main 函数

main 函数的完成的功能如下：

- 配置 Timer 及其中断
- 初始化中断控制器
- 初始化软件中断
- 初始化 LWIP 协议栈
- 建立 TCP Server，启动 Timer
- 配合 FSBL 启动 CORE1
- 持续从 LWIP 协议栈接收数据，若接收到 CORE1 触发的软件中断，则作出响应。

13.3.3.2 建立 TCP Server

基于 LWIP 库在 ARM 中建立一个 TCP Server，IP 地址为 192.168.1.10，端口号为 5010。

- lwip 库设置
见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。
- 程序解析
见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

13.3.3.3 初始化软件中断

通过 software_intr.c 中的 Init_Software_Intr() 函数初始化软件中断，对于 CORE0，该函数完成如下功能。

- 初始化 CORE1 到 CORE0 的软件中断，中断号为 2，设置中断优先级和触发方式。
- 绑定该中断的中断服务函数为 Cpu0_Intr_Handler。
- 将该中断映射至 CORE0，并使能。

13.3.3.4 启动 CORE1

通过 main.c 中的 Start_cpu1() 函数配合 FSBL 完成 CORE1 的启动。Start_cpu1() 原理如下：

- 将 FSBL 工程位于 OCM 中的 stack 区域和 vector table 区域的 cache 禁用。
- 将 CORE1 工程代码的位于 DDR 中的起始地址 0x02000000（见 core1 的 lscript.1d）写入 FSBL 的 vector table 中定义的 cpu1_catch 地址内。CORE1 工程代码的起始地址由如下宏所定义。若 core1 的 lscript.1d 中代码位于 DDR 的起始地址发生更改，则该宏定义也必须进行相应更改。

```
#define APP_CPU1_ADDR 0x02000000
```

- 复位 CORE1 及其时钟
- 使能 CORE1 及其时钟

13.3.3.5 数据写入共享内存

CORE0 通过 TCP 协议接收外部 TCP Client 发送的数据包，并将数据信息写入 CORE0 和 CORE1 共享内存中。共享内存首地址定义为：

```
#define SHARED_BASE_ADDR 0x08000000
```

为共享内存区域在 shared_mem.h 中定义结构体 shared_region，其中包含了两核间所需交互的数据长度及数据指针。

```
typedef struct
{
    u32 data_length;
    u8* dataload;
} shared_region;
```

CORE0 通过 shared_mem.c 中的 put_data_to_region() 函数将所需传递的数据长度及

指针存入 shared_region 结构体中。在 put_data_to_region 函数中调用 Xil_DCacheFlushRange 函数将 DCache 中该数据指针所指向内存区域的数据刷入 DDR 中，进行 DCache 一致性维护。

CORE0 将此结构体放置于共享内存首地址 SHARED_BASE_ADDR，CORE1 便可以从该地址读取 CORE0 所需传递的数据信息，从而进一步获取数据。

13.3.3.6 触发软件中断

CORE0 通过调用 shared_mem.c 中的 Gen_Software_Intr 函数触发 CORE0 到 CORE1 的软件中断，中断号为 1，中断目标 CPU 设置为 CORE1。

13.3.3.7 响应软件中断

当 CORE1 向 CORE0 触发中断号为 2 的软件中断时，CORE0 调用 Cpu0_Intr_Handler 函数响应此中断。然后，CORE0 通过串口打印相应信息。

13.4 CORE1 工程

13.4.1 main 函数

main 函数的完成的功能如下：

- 配置 Timer 及其中断
- 初始化中断控制器
- 初始化软件中断
- 等待 CORE0 触发的软件中断，将 CORE0 通过共享内存传递的数据由串口输出
- 串口数据输出完成后触发 CORE1 到 CORE0 的软件中断

13.4.2 初始化软件中断

通过 software_intr.c 中的 Init_Software_Intr() 函数初始化软件中断，对于 CORE1，该函数完成如下功能。

- 初始化 CORE0 到 CORE1 的软件中断，中断号为 1，设置中断优先级和触发方式。
- 绑定该中断的中断服务函数为 Cpl_Intr_Handler。
- 将该中断映射至 CORE1，并使能。

13.4.3 响应软件中断

当 CORE0 向 CORE1 触发中断号为 1 的软件中断时，CORE1 调用 Cpu1_Intr_Handler 函数响应此中断。然后，CORE1 开始从共享内存中读取 CORE0 所

传递的数据。

13.4.4 共享内存数据读出

CORE1 从共享内存区域 SHARED_BASE_ADD 地址中获取 CORE0 传递的数据信息，通过 shared_mem.c 中的 get_data_from_region() 函数将 CORE0 传递的数据长度及指针读出，并复制到本地缓存中。

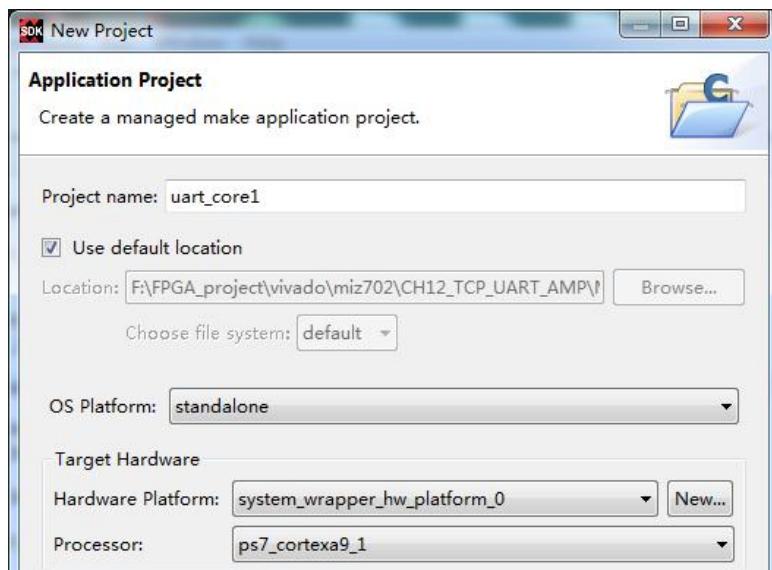
在 get_data_from_region 函数中，在将 CORE0 传递的数据复制到本地缓存区域之前，调用 Xil_DCacheInvalidateRange 函数将 DCache 中该数据指针所指向内存区域的数据设置为 invalid，进行 DCache 一致性维护。

13.4.5 触发软件中断

当 CORE1 将从共享内存中读取的数据通过串口输出完毕后，CORE1 通过调用 shared_mem.c 中的 Gen_Software_Intr 函数触发 CORE1 到 CORE0 的软件中断，中断号为 2，中断目标 CPU 设置为 CORE0。以此通知 CORE0，CORE1 串口输出数据完成。

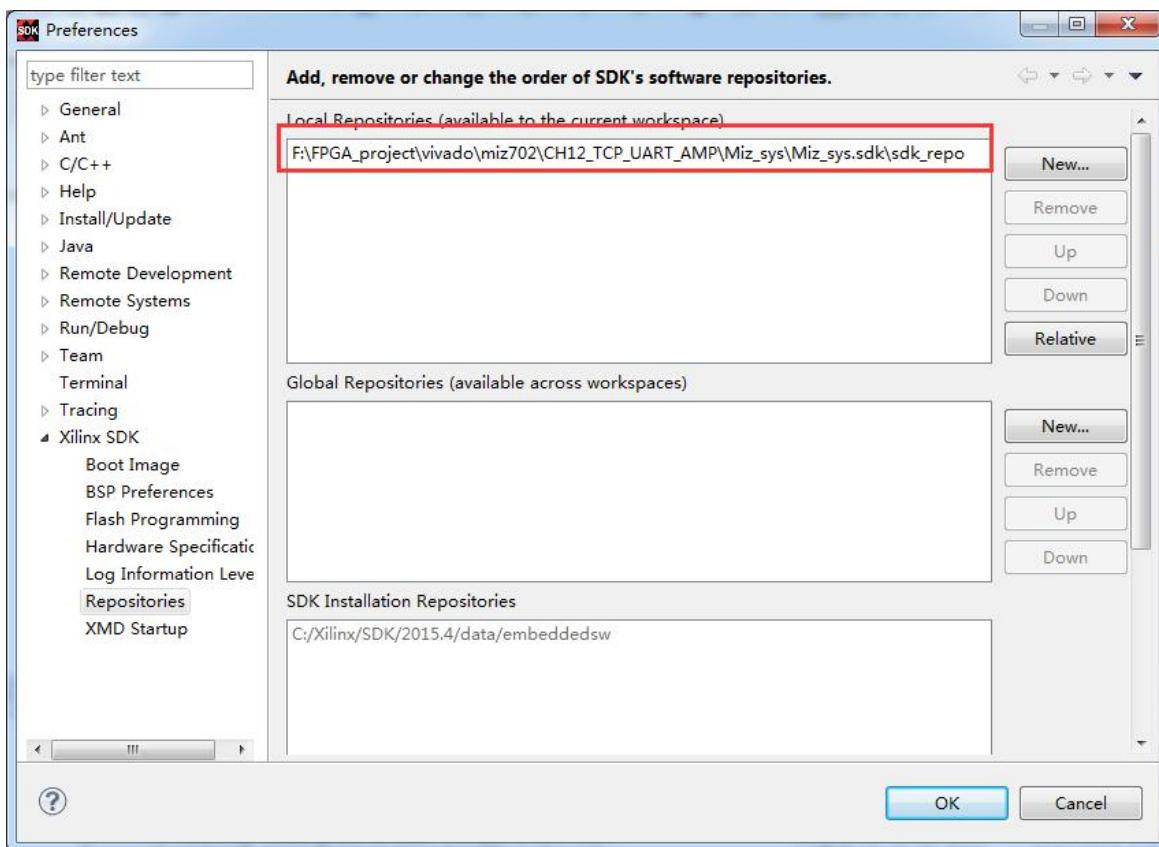
13.5 工程创建及设置关键步骤

- CORE1 工程创建时 Processor 要选择 ps_cortexa7_1，不要选成 ps_cortexa7_0，如下图所示。

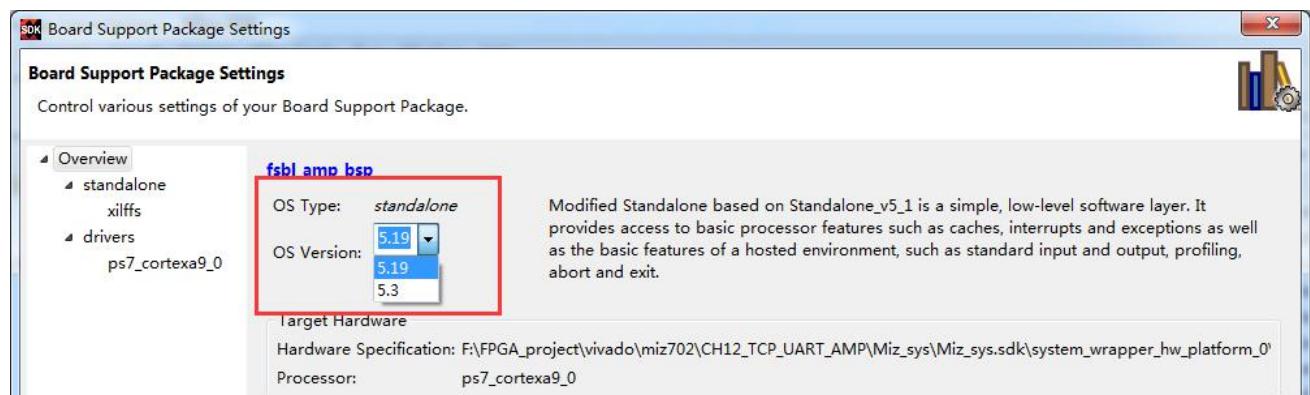


- 创建 FSBL 工程后，需替换其 bsp 的 standalone 库。

首先，设置工程目录下 sdk_repo 文件夹的路径，sdk_repo 文件夹中包含了需要使用的 standalone 库，如下图所示。用户需要根据自己所建工程的实际路径进行修改，使用原工程的路径设置将产生错误。

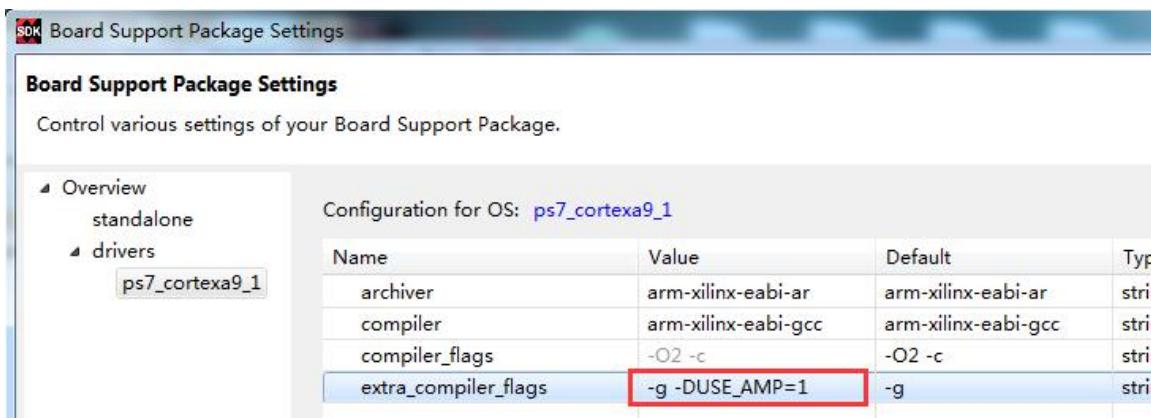


其次，更改 FSBL 工程的 bsp 中 standalone 的版本，将其更换为 sdk_repo 文件夹中的 5.19 版本，如下图所示。该版本 standalone 库来自 xapp1079，只有使用该版本的 standalone 库才可实现双核 BOOT，自带的 standalone 库无法实现。



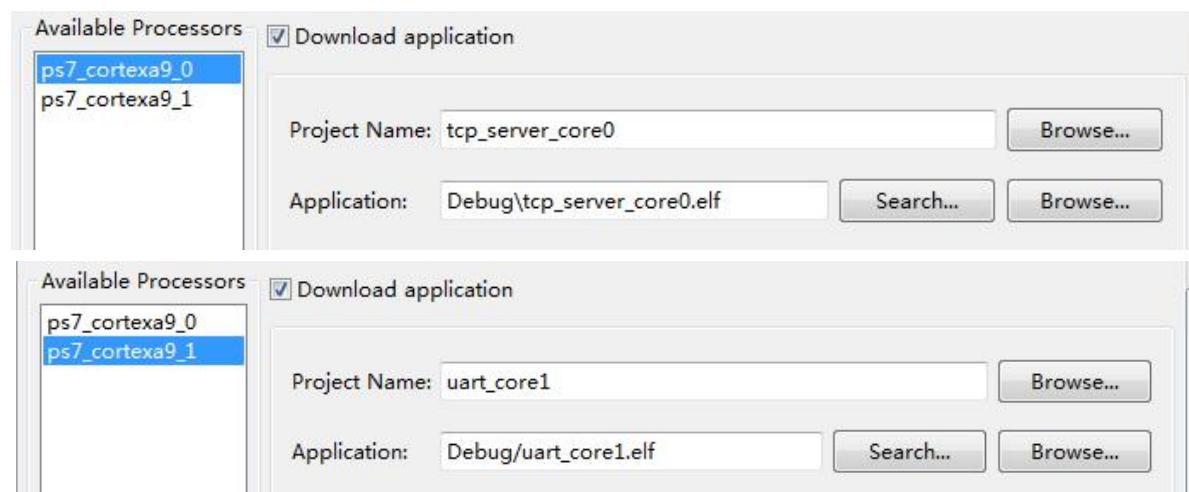
● 设置 CORE1 工程的编译选项

在 CORE1 工程的 bsp 中要增加编译选项 “-DUSE_AMP=1” ，如下图所示。该编译选项将影响到 CORE1 工程代码里中断控制器 SCUGIC 的初始化函数以及 Cache 操作函数的编译，若不增加该选项，可能会出现 CORE0 和 CORE1 中断异常和 Cache 一致性维护异常。



13.6 工程调试关键步骤

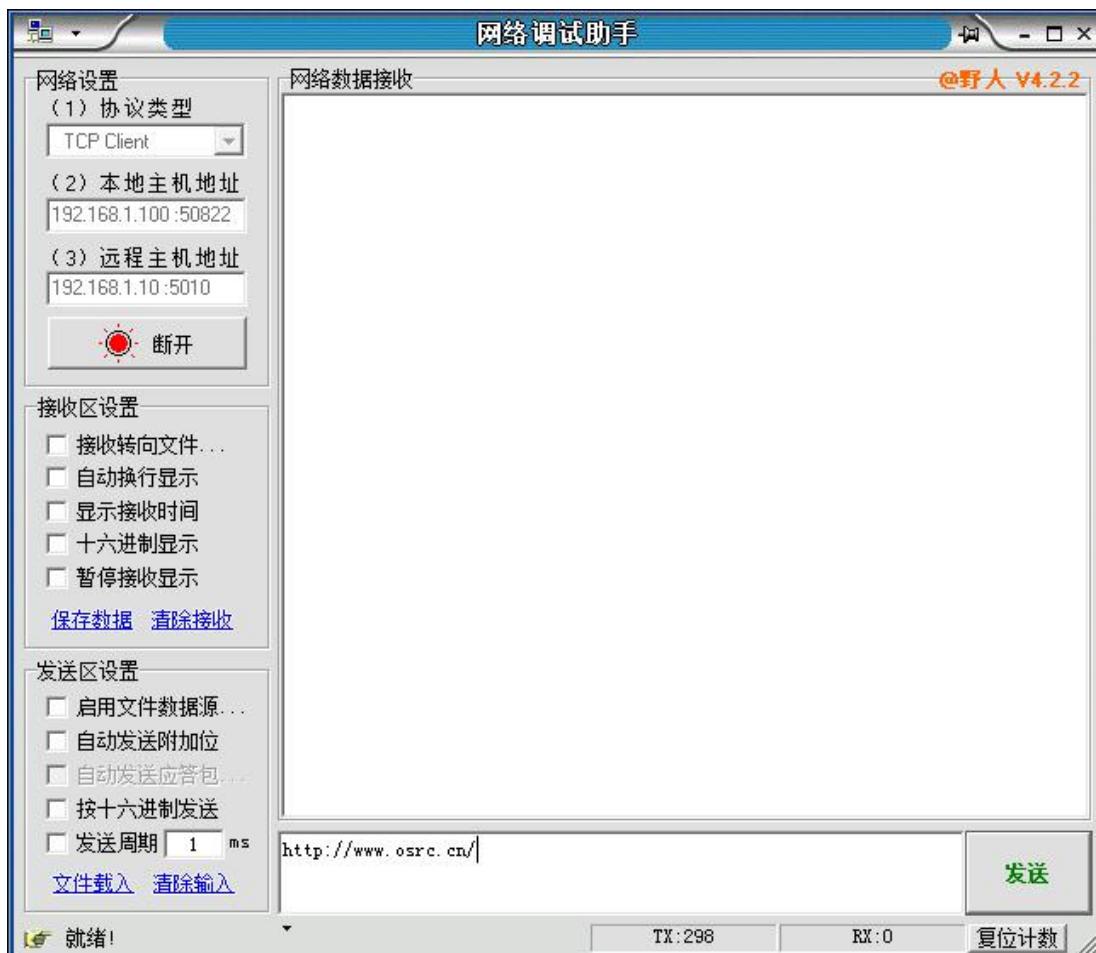
- system debugger 里同时添加 core0 和 core1 工程文件，如下图所示。Debug 时先让 CORE0 运行，再让 CORE1 运行。



- Debug 时一定要注释 CORE0 工程中 main 函数里的 Start_cpu1 函数，否则在 debug 时，CORE1 将无法正常工作。只有当需要生成 BOOT.bin 文件时，才需要使用该函数。

13.7 网络调试助手操作方法

在 SDK 中下载程序至 ZYNQ 中。打开网络调试助手，选择 TCP Client 方式，输入 ARM 中定义的 TCP Server 的 IP 地址和端口号，然后点击连接按键，建立 TCP 连接。输入任意文字信息发送。如图所示。



SDK 终端串口输出的信息如下图所示。

```
Connected to: Serial ( COM10, 115200, 0, 8 )
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
http://www.osrc.cn/
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
```

继续通过网络调试助手发送信息，串口输出如下图所示。

```
Connected to: Serial ( COM10, 115200, 0, 8 )
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
http://www.osrc.cn/
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
南京米联电子
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
```



```
Connected to: Serial ( COM10, 115200, 0, 8 )
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
南京米联电子
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
南京米联电子ZYNQ开发板
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
```

13.8 生成 BOOT.bin

切记在生成 BOOT.bin 文件时不要注释 CORE0 工程中 main 函数里的 Start_cpl1 函数，否则将无法 BOOT CORE1。生成 BOOT.bin 文件时依次添加 FSBL 工程的 elf, bit 文件，CORE0 和 CORE1 的 elf 文件，如下图所示。

Boot image partitions	
File path	
(bootloader) F:\FPGA_project\vivado\miz702\CH12_TCP_UART_AMP\Miz_sys\Miz_sys.sdk\fsbl_amp\Debug\fsbl_amp.elf	
F:\FPGA_project\vivado\miz702\CH12_TCP_UART_AMP\Miz_sys\Miz_sys.sdk\system_wrapper_hw_platform_0\system_wrapper.bit	
F:\FPGA_project\vivado\miz702\CH12_TCP_UART_AMP\Miz_sys\Miz_sys.sdk\tcp_server_core0\Debug\tcp_server_core0.elf	
F:\FPGA_project\vivado\miz702\CH12_TCP_UART_AMP\Miz_sys\Miz_sys.sdk\uart_core1\Debug\uart_core1.elf	

13.9 双核 BOOT 验证

将 BOOT.bin 文件烧入 QSPI flash 中，重启开发板。SDK 串口终端输出信息，如下图所示。当提示“core1:application start”，代表 CORE0 和 CORE1 都已成功启动。

```
Connected to: Serial ( COM10, 115200, 0, 8 )
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
core0: application start
core1: application start
```

使用网络调试助手进行 TCP 连接并发送数据，串口输出如下图所示。此时，验证了双核 BOOT 后，CORE0 和 CORE1 均运行正常。

```
Connected to: Serial ( COM10, 115200, 0, 8 )
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
core0: application start
core1: application start
core0: tcp_server: Connection Accepted
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
南京米联电子ZYNQ开发板
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
```

S03_CH14_通过 BRAM 进行 PS 和 PL 间的数据交互

14.1 概述

在之前的教程中，已经介绍了一种通过使用 AXI DMA 来进行 PS 和 PL 间的高速数据传输的方法。这种方案的特点在于，传输速度快，数据以批量的形式进行传输，无需占用 PS 端的 ARM。

然而，当我们需要在 PS 和 PL 之间传输少量，地址不连续，且长度不规则的数据，比如，配置参数，变量，控制信息等，此时 AXI DMA 便不再适用了。为此，本例程针对这种应用场合，介绍了一种基于 PL 端 BRAM 的方式，来进行 PS 和 PL 间的数据交互。

本例程基于 Vivado 2016.4 版本开发。

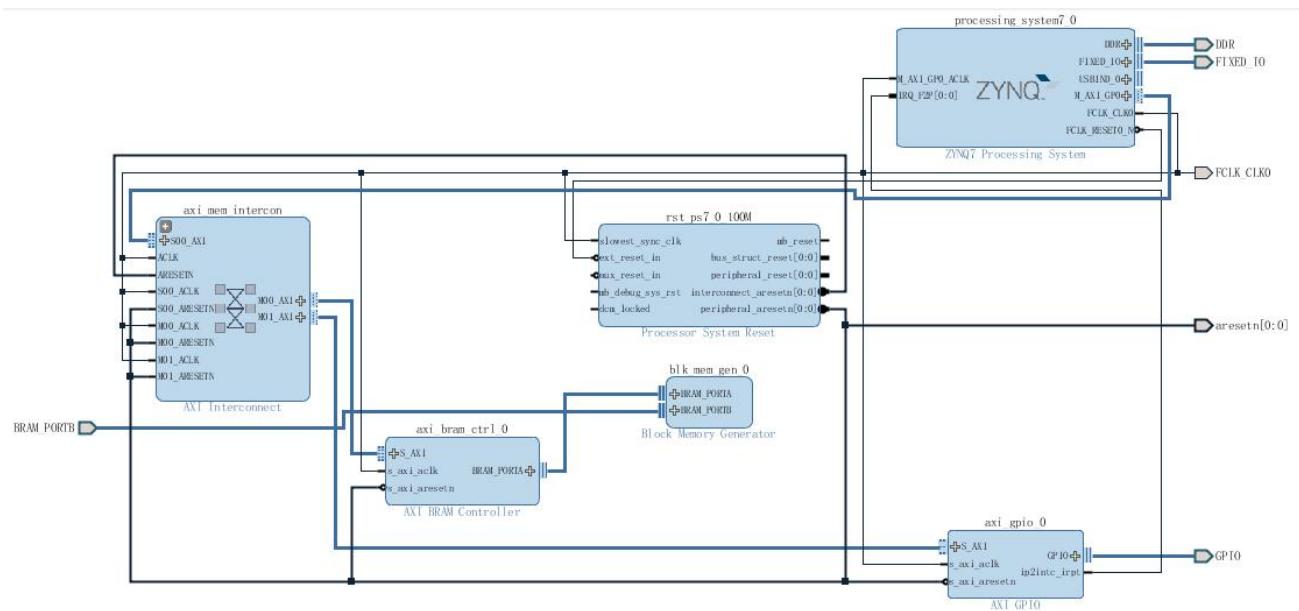
14.2 基本原理

在本例程中，在 PL 端设计了 1 个 4KB（位宽 32，深度 1024）的 BRAM。首先，PS 通过 M_AXI_GP 口向 BRAM 中 1024 个地址依次存入 1024 个 32 位的数据。PS 每向 BRAM 完成写入 1 个 32 位数据后通过 AXI GPIO 输出 1 个上升沿信号，PL 捕获上升沿后立即将 PS 写入的 32 位数据读出，然后加 2，再存入原地址中。存储完成后，PL 通过 AXI GPIO 向 PS 输入 1 个翻转信号，每翻转 1 次，AXI GPIO 便向 PS 触发 1 次中断。PS 触发中断后，再从 BRAM 中读出该数据，判断是否被加了 2，若不一致，则报错。

以上过程重复 1024 次，便将 BRAM 的所有地址都进行了遍历。之后，则不断重复这个过程。

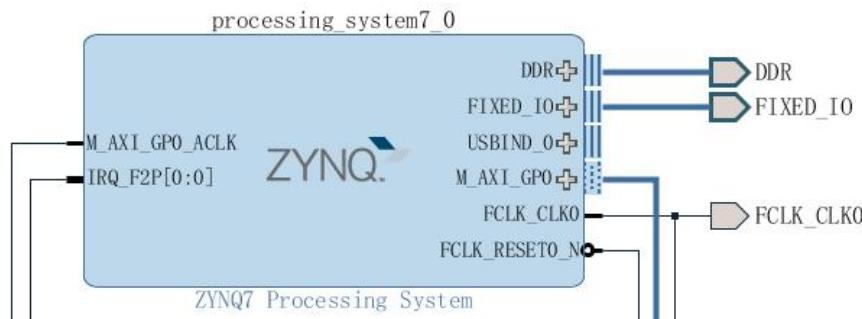
14.3 PL 部分设计

14.3.1 IP 连线图



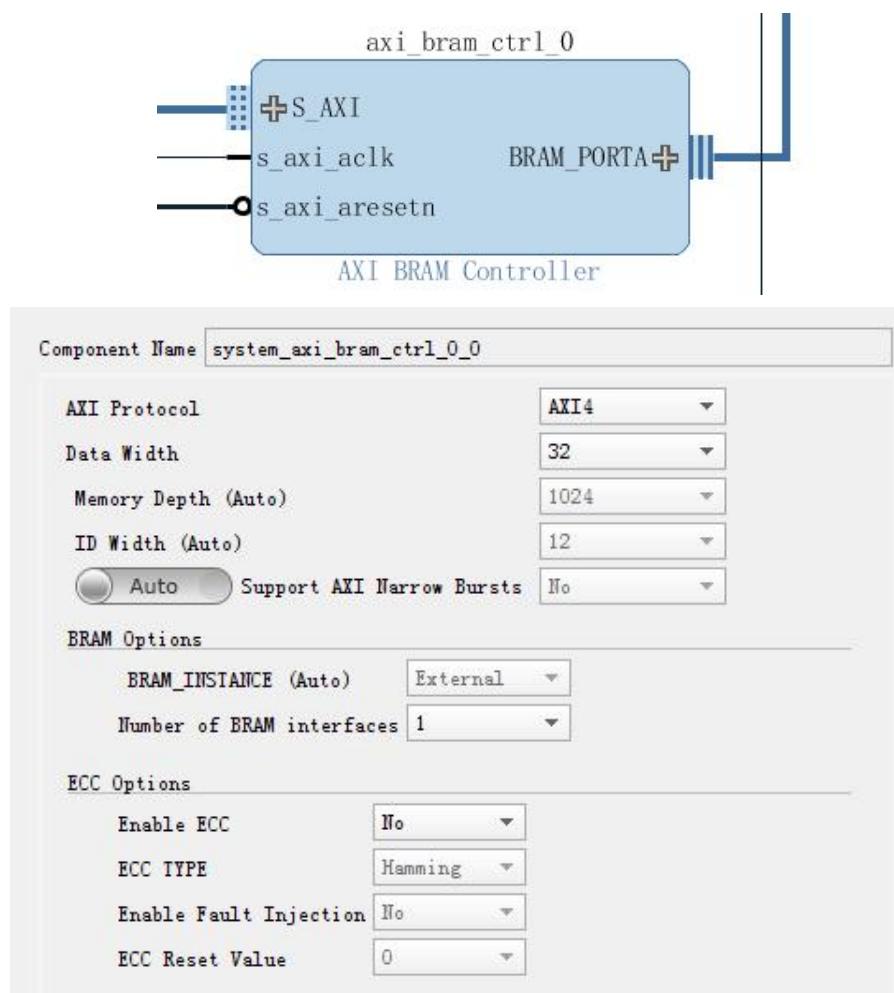
14.3.2 PS 配置

PS 的配置如下图所示。使能 M_AXI_GP0 口，将 FCLK_CLK0 设为 100MHz，使能 PL 至 PS 的中断。



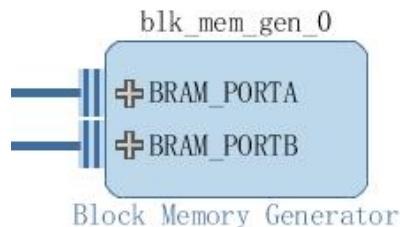
14.3.3 AXI BRAM Controller

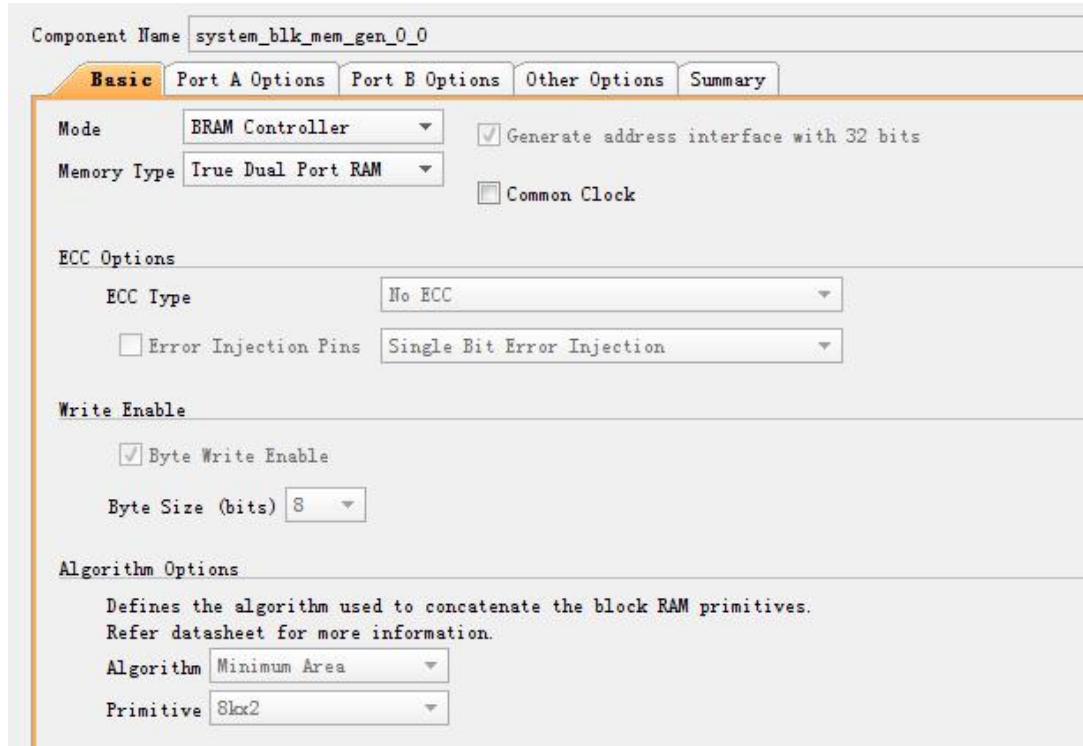
AXI BRAM Controller 是本例程中的关键，该 IP 核连接 PS 的 M_AXI_GP0 口和 BRAM，完成 AXI 接口至 BRAM 接口的转换。设置如下图所示。



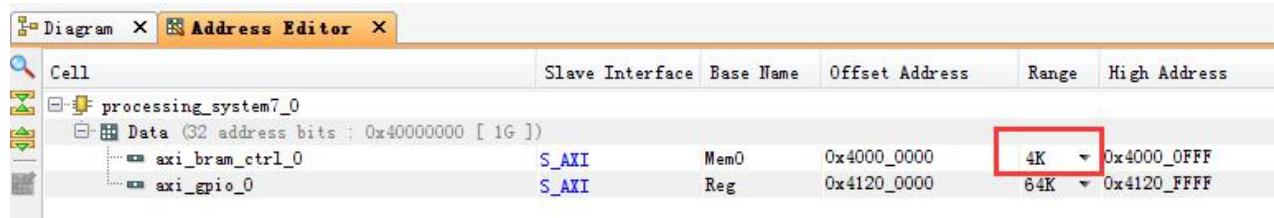
14.3.4 Block Memory Generator

添加 BRAM，将 BRAM 设置为双口 RAM，将 PORTA 与 AXI BRAM Controller 连接，PS 通过 PORTA 读写 BRAM，另外，将 PORTB 引出至外部模块，PL 通过 PORTB 读写 BRAM。如下图所示。

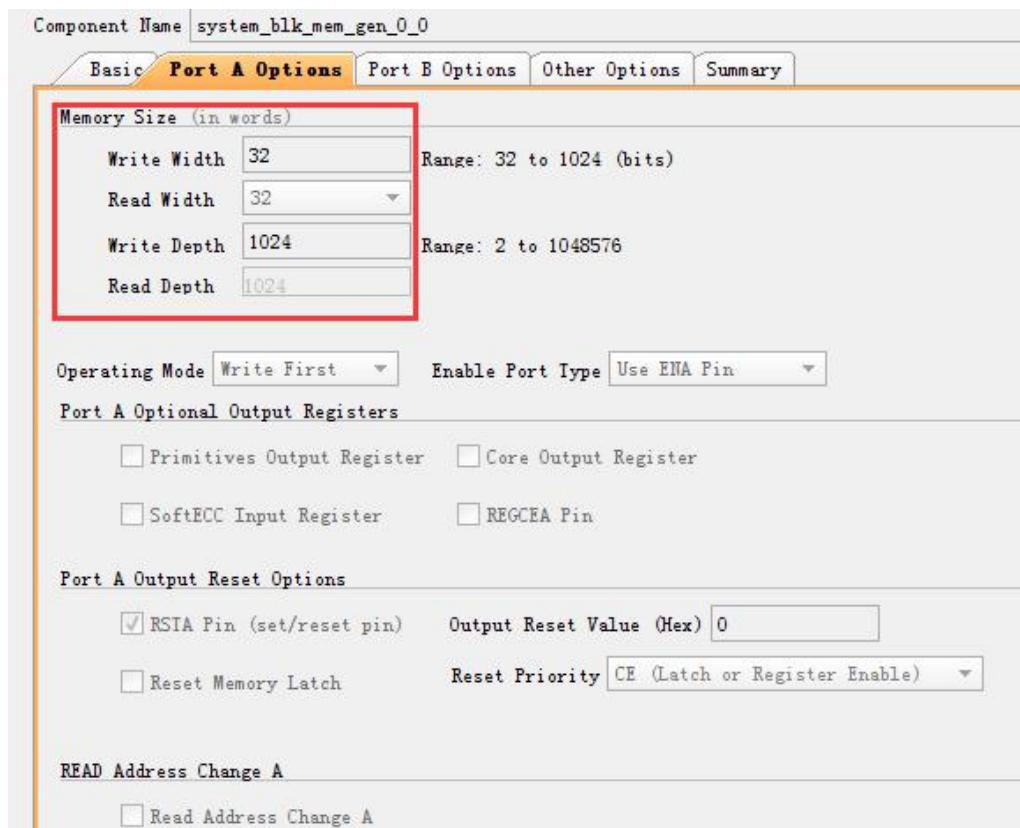




由于要与 AXI BRAM Controller 进行连接，BRAM 接口的位宽固定为 32 位。BRAM 的深度无法在该 IP 中进行设置，需要在所有模块连接完成后，在 Address Editor 里对 AXI BRAM Controller 的地址范围进行设置，本例程中设置为 4KB。该地址范围即对应 BRAM 的深度，如下图所示。

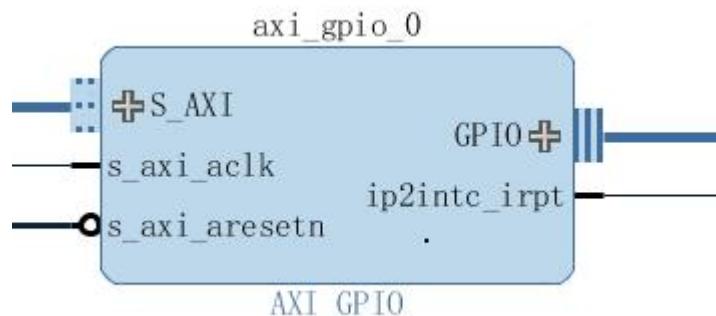


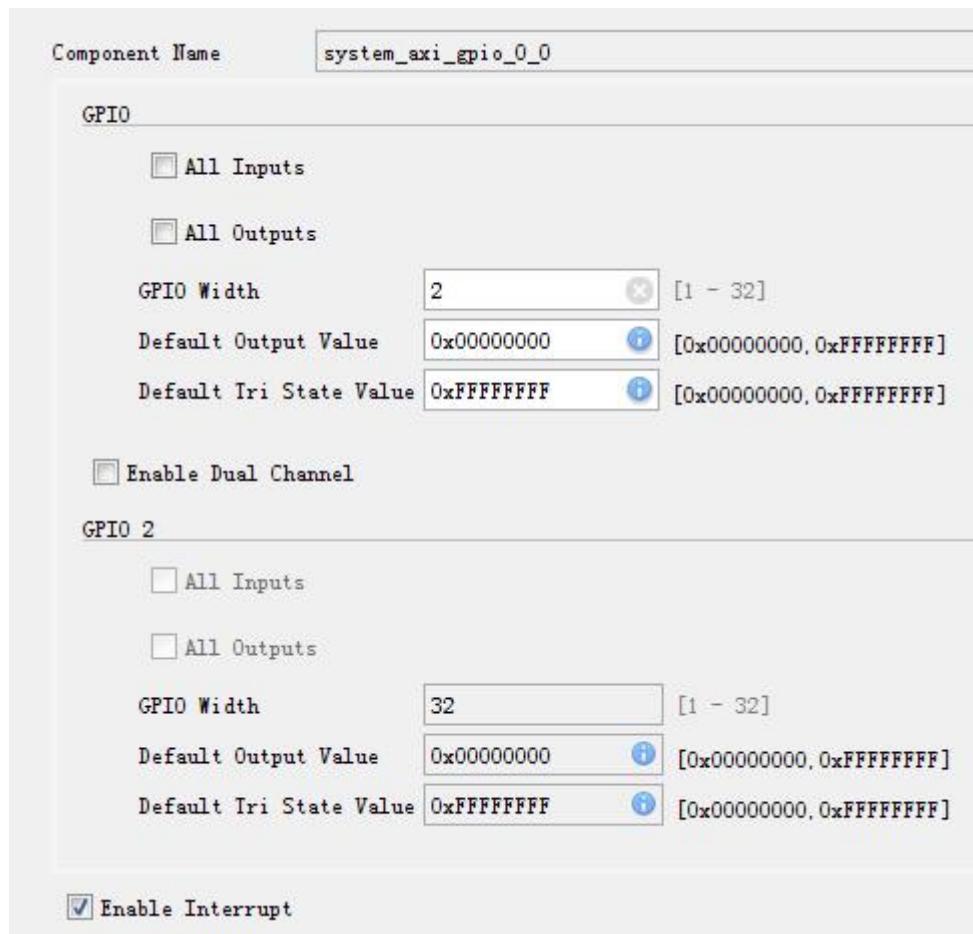
例如， $4\text{KB} = 32\text{bit} \times 1024$ ，因此 BRAM 的深度为 1024。如下图所示。



14.3.5 AXI GPIO

添加 AXI GPIO，位宽设为 2，使能中断，将中断输出与 PS 的中断接口连接，设置如下图所示。其中 GPIO0 作为 PS 向 PL 输出的 PS BRAM 写入完成信号，对于 PL 而言，上升沿有效。GPIO1 作为 PL 向 PS 输入的 BRAM 写入完成信号，该信号为翻转信号，每次翻转向 PS 产生 1 次中断。





14.4 逻辑设计

PL 部分逻辑设计主要包括以下几个过程：

检测 AXI GPIO 输出的 GPIO0 的上升沿；

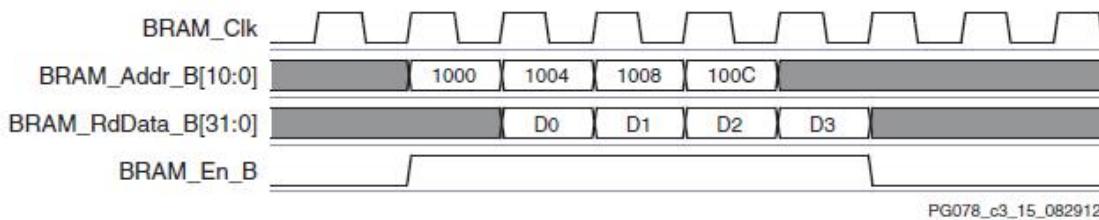
若检测到 GPIO0 的上升沿，则从 BRAM 的某 1 个地址中读出 1 个 PS 写入 32 位数据，然后加 2，存入原地址中；

1 个 32 数据存储完毕后，将 AXI GPIO 的输入信号 GPIO1 进行翻转，告知 PS 一次数据读写完成。

不断循环上述过程，依次遍历 BRAM 中 0~4092 的地址范围。

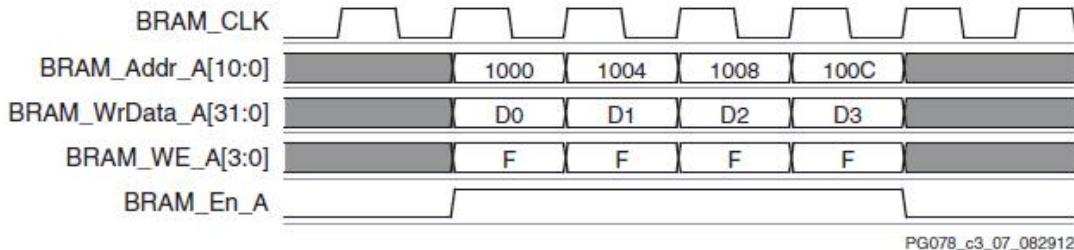
14.4.1 BRAM 读时序

BRAM 读时序如下图所示。



14.4.2 BRAM 写时序

BRAM 写时序如下图所示。



14.5 PS 程序设计

所有的驱动程序文件均包含在 `c_driver` 文件夹中。

14.5.1 main 函数

`main` 函数的完成的功能如下：

配置 AXI GPIO 及其中断；

初始化中断控制器及系统中断；

依次向 BRAM 所对应的地址写入 1024 个 32 位整型数据，每写入 1 个数据等待 PL 的读写完成中断来临后继续写入下一个；

依次从 BRAM 所对应的地址读出 1024 个 32 位整型数据，并判断是否被加了 2，若比对不一致则报错。

14.5.2 GPIO 输入输出

在 `main` 函数调用 `Gpiopl_init()` 函数初始化 AXI GPIO，设置 2 个 GPIO 的方向，其中 `GPIO[0]` 为输出，`GPIO[1]` 为输入。并将 `GPIO[0]` 置为 0。每个 GPIO 的宏定义在 `gpiopl_intr.h` 中，如下所示。

```
#define PS_BRAM_MASK          0x00000001
#define PL_INTR_MASK           0x00000002
```

通过 `Gpiopl_Setup_Intr_System()` 初始化并使能 AXI GPIO 的输入中断，`GPIO[1]` 的输入发生 1 次改变将触发 1 次中断，`GpioplIntrHandler()` 为 GPIO 的中断函数。

GpioplIntrHandler()函数将 PL 读写 BRAM 完成中断标志位 pl_bram_flag 置 1，该信号将在 PS 写入 BRAM 时使用。

14.5.3 BRAM 数据写入

每一个循环 PS 向 BRAM 写入 1024 个 32bit 整型数据，每次循环后将下一次写入的 1024 个数据都加 1。因此，第 1 次写入 BRAM 的数据为 0~1023，第 2 次写入的为 1~1024，第 3 次为 2~1025，以此类推。BRAM 写入通过如下函数完成：

```
Xil_Out32(XPAR_BRAM_0_BASEADDR + 4*i, write_data);
```

由于在 ZYNQ 中最小可寻址单元为字节，因此 1 个 32 位数据需占用 4 个地址，每次写入的地址都需要加 4。

每次写入完成后，拉高 GPIO0，通过如下代码完成：

```
XGpio_DiscreteWrite(&Gpio, 1, PS_BRAM_MASK);
```

然后，PS 等待 PL 完成 BRAM 中该地址 32bit 数据的读写，PL 翻转 GPIO1 使 AXI GPIO 产生中断，代码如下所示：

```
while(!pl_bram_flag);
pl_bram_flag = 0;
```

PL 完成 BRAM 读写后，PS 拉低 GPIO0，通过如下代码完成：

```
XGpio_DiscreteWrite(&Gpio, 1, ~PS_BRAM_MASK);
```

循环 1024 次，完成 1024 个数据的写入。

14.5.4 BRAM 数据读出

当 PS 完成 1024 个数据的写入，此时 PL 也完成了 1024 个数据的读出、加 2、写入工作。因此，PS 需要依次将 1024 个数据读出进行比对，验证 PL 给每个数据加 2 的正确性，因此，第 1 次读出的 1024 个数据应该为 2~1025，第 2 次为 3~1026，第 3 次为 4~1027，以此类推。若比对出现不一致，则通过串口进行报错。代码如下：

```
for(i      = 0;      4*i      <      (XPAR_BRAM_0_HIGHADDR -  
XPAR_BRAM_0_BASEADDR); i++)  
{  
    read_data = Xil_In32(XPAR_BRAM_0_BASEADDR + 4*i);  
    //xil_printf("data at address %d is %d\r\n", 4*i, read_data);  
    /*compare data read from bram if they are add by 2*/  
    if(read_data != (i + j + 2))  
        xil_printf("error: data at address %d should be %d, but is %d\r\n", 4*i,  
(i + j + 2), read_data);  
}
```

其中，可通过 xil_printf("data at address %d is %d\r\n", 4*i, read_data); 查看每一次从 BRAM 读出的数据的具体值，若无需使用则可注释掉。

14.6 程序测试

程序测试串口打印信息如下图所示。

第 1 次读出的 1024 个数据。

Connected to: Serial (COM1)

```
data at address 0 is 2
data at address 4 is 3
data at address 8 is 4
data at address 12 is 5
data at address 16 is 6
data at address 20 is 7
data at address 24 is 8
data at address 28 is 9
data at address 32 is 10
data at address 36 is 11

data at address 4064 is 1018
data at address 4068 is 1019
data at address 4072 is 1020
data at address 4076 is 1021
data at address 4080 is 1022
data at address 4084 is 1023
data at address 4088 is 1024
data at address 4092 is 1025
data at address 0 is 3
data at address 4 is 4
data at address 8 is 5
data at address 12 is 6
data at address 16 is 7
data at address 20 is 8
data at address 24 is 9
data at address 28 is 10
data at address 32 is 11
data at address 36 is 12
data at address 40 is 13
data at address 44 is 14
data at address 48 is 15
data at address 52 is 16
data at address 56 is 17
data at address 60 is 18
```

第 2 次读出的 1024 个数据。

```
data at address 4060 is 1018
data at address 4064 is 1019
data at address 4068 is 1020
data at address 4072 is 1021
data at address 4076 is 1022
data at address 4080 is 1023
data at address 4084 is 1024
data at address 4088 is 1025
data at address 4092 is 1026
data at address 0 is 4
data at address 4 is 5
data at address 8 is 6
data at address 12 is 7
data at address 16 is 8
data at address 20 is 9
data at address 24 is 10
data at address 28 is 11
data at address 32 is 12
data at address 36 is 13
data at address 40 is 14
data at address 44 is 15
data at address 48 is 16
data at address 52 is 17
data at address 56 is 18
```

第3次读出的1024个数据。

```
data at address 4052 is 1017
data at address 4056 is 1018
data at address 4060 is 1019
data at address 4064 is 1020
data at address 4068 is 1021
data at address 4072 is 1022
data at address 4076 is 1023
data at address 4080 is 1024
data at address 4084 is 1025
data at address 4088 is 1026
data at address 4092 is 1027
data at address 0 is 5
data at address 4 is 6
data at address 8 is 7
data at address 12 is 8
data at address 16 is 9
data at address 20 is 10
data at address 24 is 11
data at address 28 is 12
data at address 32 is 13
data at address 36 is 14
data at address 40 is 15
data at address 44 is 16
data at address 48 is 17
```

后面不作一一列举。

14.7 课后习题

本课读写的速度相对较慢，读者可以尝试修改程序，实现批量数据的读写。

S03_CH15_EMIO 光电通信-FEP 子卡的使用

15.1 概述

目前，在 ZYNQ 中进行以太网开发的方案，大部分都是基于通过 PS 的 MIO 以 RGMII 接口连接外部 PHY 芯片的方式。但是，由于使用 PS 的 MIO 只能以 RGMII 接口连接外部 PHY 芯片，这就限制了支持其他接口 PHY 芯片的使用，如 GMII、SGMII、MII 等等。因此，若将 PS 内部的以太网控制器 ENET0/ENET1 通过 EMIO 的方式扩展至 PL，便可通过 PL 连接更多种类的接口，从而增加了设计的灵活性。

本例程基于米联电子设计的光电双口扩展子卡 FEP_ETH_SFP_CARD，设计了一种通过 EMIO 实现 PL 连接外部 RGMII 接口的 PHY 芯片，实现 PS LWIP 网络通信的方案。扩展子卡使用了 88E1512 芯片，可接光口或电口，且为自适应。

本例程基于 Vivado 2016.4 版本开发，参考了 fpgadeveloper 工程师的应用工程：

<https://github.com/fpgadeveloper/ethernet-fmc-zynq-gem>。

15.2 基本原理

本例程将 PS 的 ETH1 通过 EMIO 方式引出，通过 EMIO 引出的 ETH1 为 GMII 接口，将其与 GMII to RGMII IP 核连接后转换成 RGMII 接口，然后与外部子卡中的 88E1512 芯片连接。在 PS 端通过 SDK 自带的 lwip echo server 例程通过子卡，分别以 RJ45 电口和 SFP 电口两种方式与 PC 机实现 TCP 网络通信。

15.15.1 88E1512

子卡所使用的 88E1512 芯片支持电口和光口，当其上电复位后，自动进入 RGMII to Copper 以及 RGMII to 1000BASEX 自适应状态，这是由其如下图寄存器值决定。当需要使用电口时，将网线与子卡的 RJ45 连接，当需要使用光口时，在子卡的 SFP 屏蔽笼中插入光模块即可。

Fiber/Copper Auto-Selection

The device has a patented feature to automatically detect and switch between fiber and copper cable connections. The auto-selection operates in one of three modes: Copper /1000BASE-X, Copper/100BASE-FX, and Copper/SGMII Media Interface.

Register 20_18.2:0 and register 20_18.6 select the Fiber/Copper auto media modes of operation. See [Table 33](#) for details.

Table 33: Fiber/Copper Auto-media Modes of Operation

Reg 20_18.2:0	Reg 20_18.6	Auto-media Modes of Operation
110	X	Copper/SGMII media
111	0	Copper/1000BASE-X
011	1	Copper/100BASE-FX

The device monitors the signals of the S_INP/N and the MDIP/N[3:0] lines. If a fiber optic cable is plugged in, the device will adjust itself to be in fiber mode. If an RJ-45 cable is plugged in, the device will adjust itself to be in copper mode. If both cables are connected then the first media to establish link, or the preferred media will be enabled. The media which is not enabled will be automatically turned off to save power. If the link on the first media is lost, then the inactive media will be powered up, and both media will once again start searching for link.

Table 220: General Control Register 1
Page 18, Register 20

Bits	Field	Mode	HW Rst	SW Rst	Description
15	Reset	R/W, SC	0x0	SC	Mode Software Reset. Affects page 6 and 18 Writing a 1 to this bit causes the main PHY state machines to be reset. When the reset operation is done, this bit is cleared to 0 automatically. The reset occurs immediately. 1 = PHY reset 0 = Normal operation
14:13	Reserved	R/W	0x0	Retain	Set to 0s.
12:10	Reserved	R/W	0x0	Retain	Reserved for future use.
9:7	Reserved	R/W	0x4	Retain	Set to 100
6	Auto-Media Detect (AMD) 100BASE-FX/ 1000BASE-X	R/W	0x0	Retain	This bit selects the fiber auto-media modes as follows: 1 = mode 011 is AMD between copper and 100BASE-FX 0 = mode 111 is AMD between copper and 1000BASE-X
5:4	Auto Media Detect Preferred Media	R/W	0x0	Retain	00 = Link on first media 01 = Copper Preferred 10 = Fiber Preferred 11 = Reserved
3	Reserved	R/W	0x0	Update	Set to 0
2:0	MODE[2:0]	R/W	See Descr.	Update	Changes to this bit are disruptive to the normal operation; therefore, any changes to these registers must be followed by a software reset to take effect. 1512 device comes up in MODE[2:0] =0x7 on hardware reset. 000 = RGMII (System mode) to Copper 001 = SGMII (System mode) to Copper 010 = RGMII (System mode) to 1000BASE-X 011 = RGMII (System mode) to 100BASE-FX 100 = RGMII (System mode) to SGMII (Media mode) 101 = Reserved 110 = RGMII (System mode) to Auto Media Detect Copper/SGMII (Media mode) 111 = RGMII (System mode) to Auto Media Detect Copper/1000BASE-X/100BASE-FX (see 20_18.6)

15.15.2 88E1512 RGMII 接口时序

88E1512 芯片 RGMII 接口的时序存在延迟和非延迟 2 种模式，由其内部的一个寄存器值来决定，如下图所示。

Table 121: MAC Specific Control Register 2
Page 2, Register 21

Bits	Field	Mode	HW Rst	SW Rst	Description
12:7	Reserved		0x20	0x20	Reserved.
6	Default MAC interface speed (MSB)	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. Also, used for setting speed of MAC interface during MAC side loopback. Requires that customer set both these bits and force speed using register 0 to the same speed. MAC Interface Speed during Link down. Bits 6, 13 00 = 10 Mbps 01 = 100 Mbps 10 = 1000 Mbps
5	RGMII Receive Timing Control	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. 1 = Receive clock transition when data stable 0 = Receive clock transition when data transitions
4	RGMII Transmit Timing Control	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. 1 = Transmit clock internally delayed 0 = Transmit clock not internally delayed

芯片上电通过RESET引脚复位后，寄存器中决定RGMII输入输出时序关系的bit位的值都是1。也就是说，上电复位后芯片的RGMII接口均为延迟时序模式。延迟模式是指88E1512芯片在其内部给输出的接收时钟信号RX_CLK和输入的发送的时钟信号TX_CLK增加了2ns左右的延时。这是为了避免通过PCB绕线的方式增加时钟信号的延时，使RX_CLK和TX_CLK都能与相应RXD和TXD数据窗的中心对齐，从而使得RGMII接口数据的建立和保持时间达到余量最大的状态。

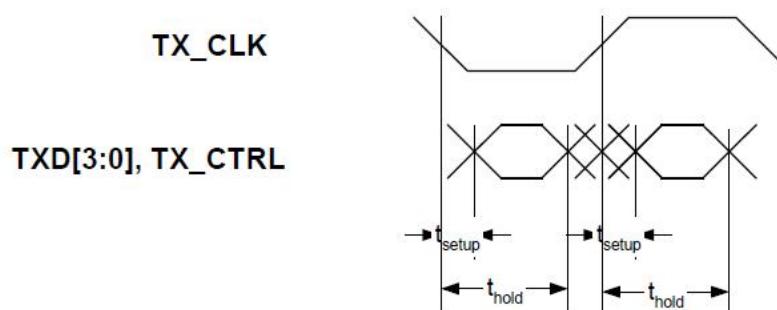
因此，此时RGMII接口发送部分信号的时序关系如下图所示。

4.10.2.2 PHY Input - TX_CLK Delay when Register 21_2.4 = 1

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.4 = 1 (add delay)	-0.9			ns
t_{hold}		2.7			ns

Figure 37: TX_CLK Delay Timing - Register 21_2.4 = 1 (add delay)



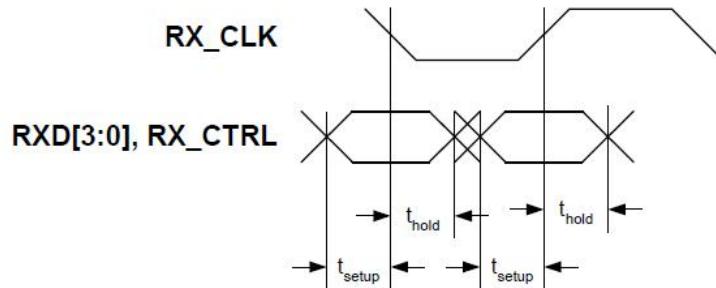
RGMII接口接收部分信号的时序关系如下图所示。

4.10.2.4 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

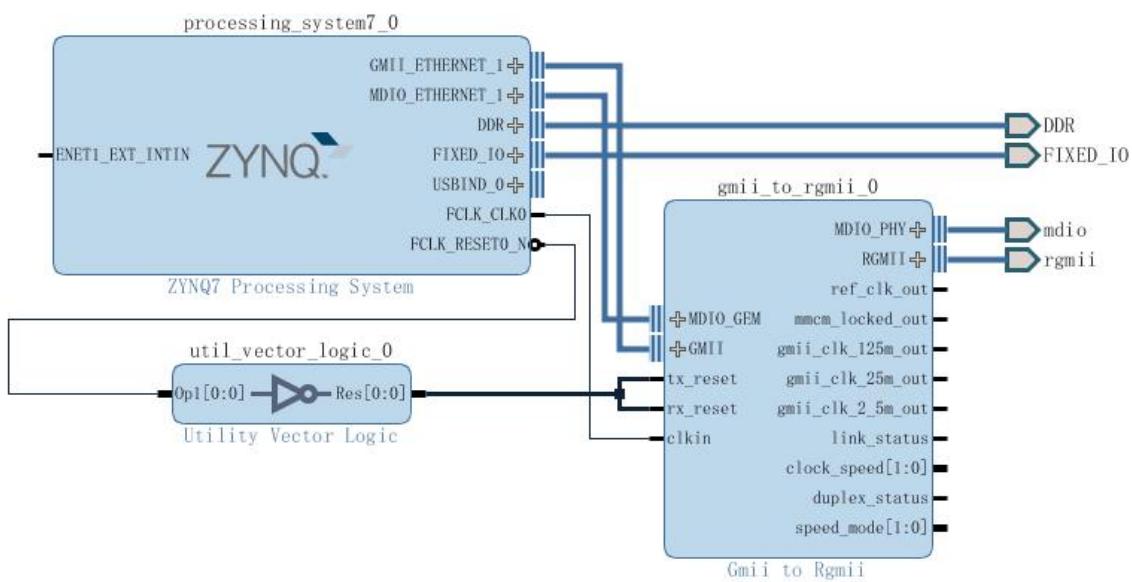
Symbol	Parameter	Min	Typ	Max	Units
tsetup	Register 21_2.5 = 1 (add delay)	1.2			ns
t _{hold}		1.2			ns

Figure 39: RGMII RX_CLK Delay Timing - Register 21_2.5 = 1 (add delay)



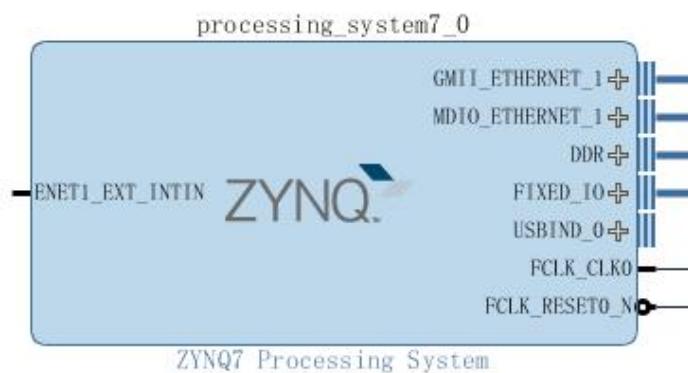
15.3 PL 部分设计

15.3.1 IP 连线图



15.3.2 ZYNQ PS 设置

将 ETH1 及其 DMIO 通过 EMIO 引出，将 FCLK_CLK0 设置为 200M，作为 GMII to RGMII IP 核内部 IDELAYCTRL 的参考时钟，FCLK_RESET0_N 通过 Utility Vector Logic 生成的非门后作为 GMII to RGMII IP 核的复位信号。如下图所示。



15.3.3 GMII to RGMII

15.3.3.1 PS ETH EMIO

PS 的 ETH 通过 EMIO 引出后为标准的 GMII 接口，如下图所示。

Zynq-7000 AP SoC Device

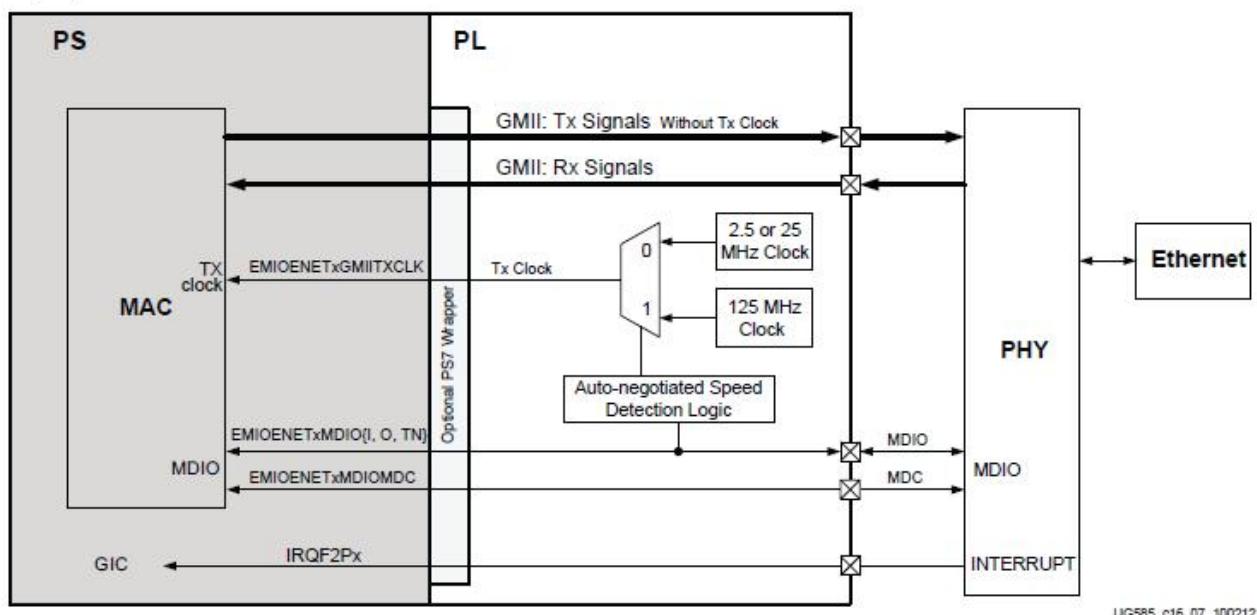


Figure 16-8: GMII Interface via EMIO Connections

需要通过 IP 核 GMII to RGMII 将 GMII 接口转换为 RGMII 接口，才能与 PHY 芯片 88E1512 连接。连接原理如下图所示。

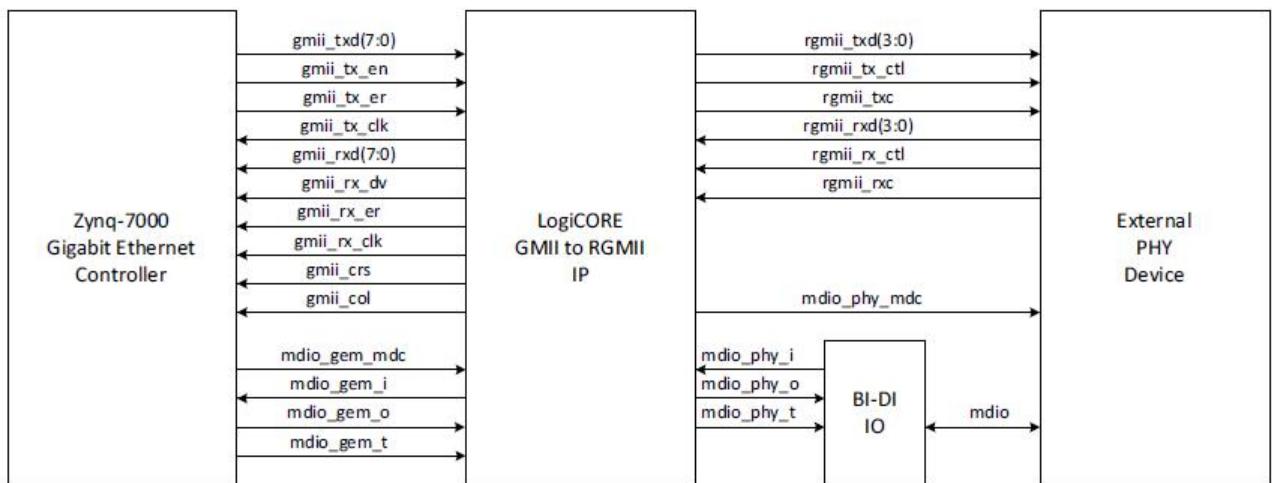
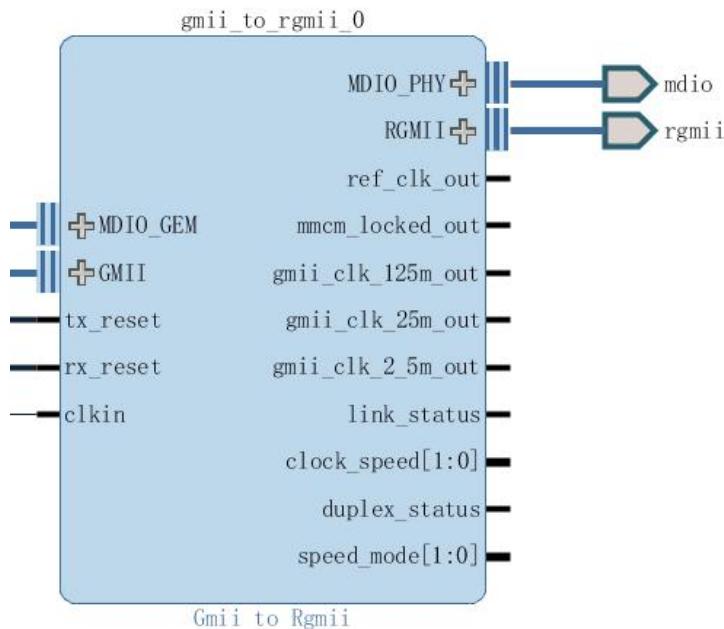


Figure 2-1: GMII to RGMII Core Ports and Interfaces

15.3.3.2 IP 核设置

GMII to RGMII IP 核的设置如下图所示。



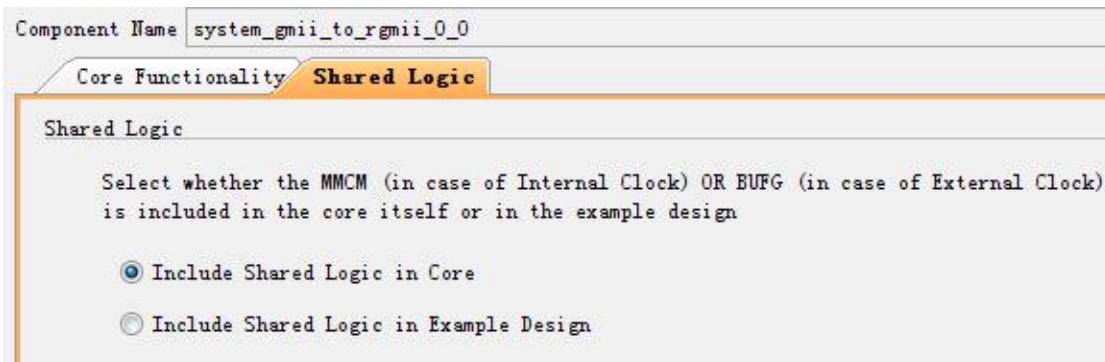
ZYNQ7010/7020 只有 HR BANK，在 HR BANK 中，IP 核中 RGMII 接口的接收数据信号和控制信号需要通过 IDELAYE2 来调整信号输入延时，使其时序满足建立和保持时间约束。因此需要在 IP 核包含与 IDELAYE2 相关的 IDELAYCTRL，用来校准 IDELAYE2 每个延时 tap 的延时值。

将本 IP 核的 PHY address 设置为 8（该值可任意设置，但不能与 88E1512 的 PHY address 相同，否则将产生冲突使 IP 核工作异常，子卡中 88E1512 的 PHY address 为 0）。

由于 88E1512 发送信号延时由芯片内部提供，因此，选择 2ns 的延时 skew 由 PHY 增加。



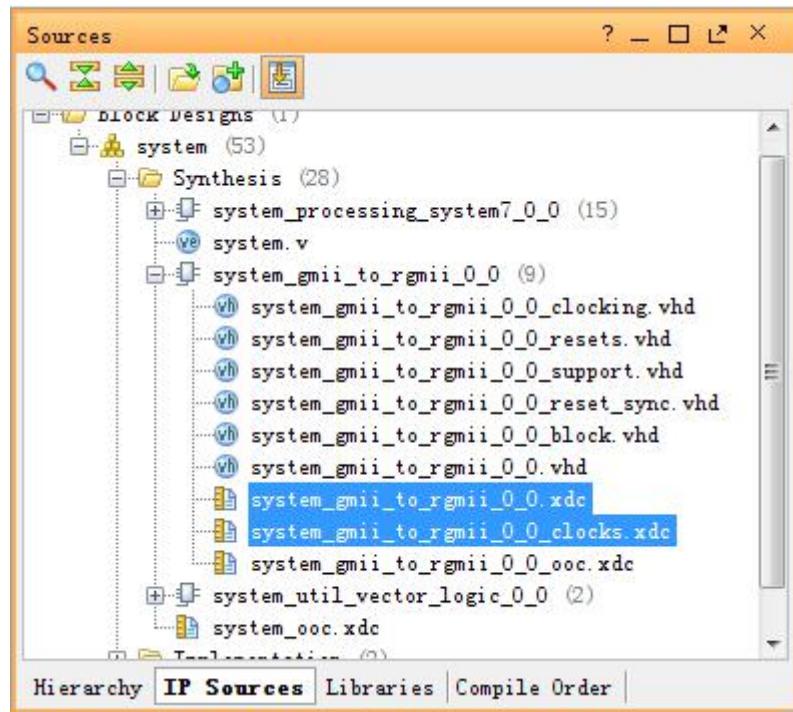
选择 shared logic 包含在 IP 核内部。



15.3.4 时序约束

本例程中，时序约束主要是针对 RGMII 接口进行 input delay 和 output delay 的约束，以及 IDELAYE2 延时 tap 数的设置，使 RGMII 接口的满足建立和保持时间的要求，从而达到时序收敛。

对于 input delay 和 output delay 的约束，GMII to RGMII IP 核所自带的 system_gmii_to_rgmii_0_0_clocks.xdc 文件中已经包含了默认设置。对于 IDELAYE2 延时 tap 数的设置，在 system_gmii_to_rgmii_0_0.xdc 中包含了默认模板。这两个 xdc 文件位置如下图所示。



15.3.4.1 input delay 约束

system_gmii_to_rgmii_0_0_clocks.xdc 文件中，关于 input delay 的约束如下所示。约束范围为：-1.5~15.8ns。

```

# define a virtual clock to simplify the timing constraints
create_clock -name system_gmii_to_rgmii_0_0_rgmii_rx_clk -period 8

# Identify RGMII Rx Pads only.
# This prevents setup/hold analysis being performed on false inputs,
# eg, the configuration_vector inputs.

set_input_delay -clock [get_clocks system_gmii_to_rgmii_0_0_rgmii_rx_clk] -max -1.5 [get_ports {rgmii_rx[0] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks system_gmii_to_rgmii_0_0_rgmii_rx_clk] -min -2.8 [get_ports {rgmii_rx[1] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks system_gmii_to_rgmii_0_0_rgmii_rx_clk] -clock_fall -max -1.5 -add_delay [get_ports {rgmii_rx[0] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks system_gmii_to_rgmii_0_0_rgmii_rx_clk] -clock_fall -min -2.8 -add_delay [get_ports {rgmii_rx[1] rgmii_rx_ctl}]

```

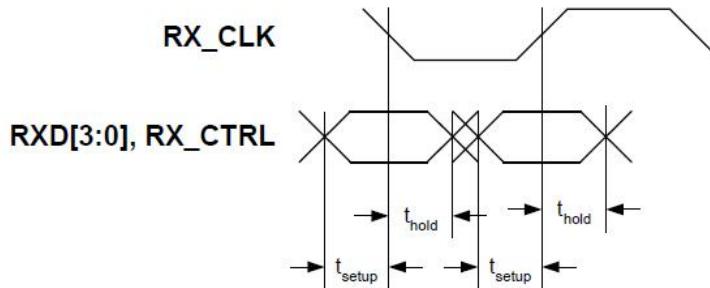
在 15.2 节中已经对 88e1512 芯片的 RGMII 接口时序进行了介绍，对于 RX 接口，如下图。

4.10.2.4 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.5 = 1 (add delay)	1.2			ns
t_{hold}		1.2			ns

Figure 39: RGMII RX_CLK Delay Timing - Register 21_2.5 = 1 (add delay)



按照上图中的 setup time 和 hold time, input delay 的-min 应该为- (4-1.2) =-15.8ns, -max 应该为-1.2ns。显然, IP 核自带 input delay 的约束范围为-1.5ns~15.8ns, 比我们计算的结果-1.2ns~15.8ns 略为宽松, 用于 setup time 计算所需的-max 时间小了 0.3ns, 为了保证时序严格收敛, 需要将 system_gmii_to_rgmii_0_0_clocks.xdc 文件中 set_input_delay -max 的-1.5 全部改为-1.2。需要注意的是, 这些 xdc 文件由 IP 核自动生成, 每次重新配置 IP 后, vivado 都会重新生成 IP 的 output product, 因此会将被修改的 xdc 文件覆盖还原, 需要手动重新再次修改 xdc 文件才行。

15.3.4.2 output delay 约束

system_gmii_to_rgmii_0_0_clocks.xdc 文件中, 关于 output delay 的约束如下所示。约束范围为: -1.0~15.6ns。

```
create_generated_clock -add -name rgmii_tx_clk -divide_by 1 -source [get_pins -of [get_cells -hier -filter {name =~ *rgmii_txc_out}]

set_output_delay -clock [get_clocks rgmii_tx_clk] -max -1.0 [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
set_output_delay -clock [get_clocks rgmii_tx_clk] -min -2.6 [get_ports {rgmii_txd[*] rgmii_tx_ctl}] -add_delay
set_output_delay -clock [get_clocks rgmii_tx_clk] -clock_fall -max -1.0 [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
set_output_delay -clock [get_clocks rgmii_tx_clk] -clock_fall -min -2.6 [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
```

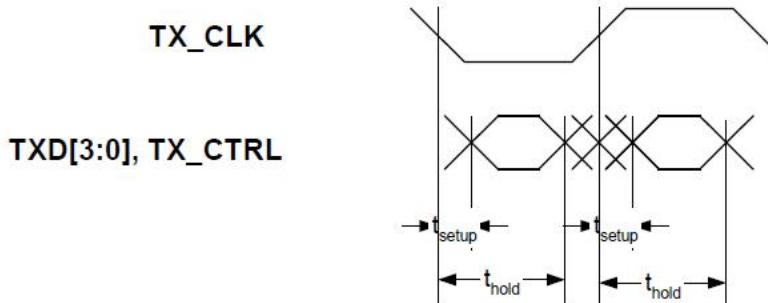
对于 88e1512 RGMII 的 TX 接口, 时序关系如下图。

4.10.2.2 PHY Input - TX_CLK Delay when Register 21_2.4 = 1

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.4 = 1 (add delay)	-0.9			ns
t_{hold}		2.7			ns

Figure 37: TX_CLK Delay Timing - Register 21_2.4 = 1 (add delay)



照上图中的 setup time 和 hold time, output delay 的-min 应该为-0.9ns, -max 应该为-15.7ns。显然, IP 核自带 output delay 的约束范围为-1.0ns~15.6ns, 比我们计算的结果-0.9ns~15.7ns 略为宽松, 用于 setup time 计算所需的-max 时间小了 0.1ns, 用于 hold time 计算所需的-min 时间大了 0.1ns。为了保证时序严格收敛, 需要将 system_gmii_to rgmii_0_0_clocks.xdc 文件中 set_output_delay -max 的 -1.0 全部改为-0.9, -min 的-15.6 全部改为-15.7。

15.3.4.3 IDELAYE2 延时设置

在 GMII to RGMII 的 IP 核内部为 rgmii_rx_ctl 和 rgmii_rxd[3:0]端口都连接了 IDELAYE2 模块, 用来为这些信号进入 PL 内部前引入额外的延时。IDELEYE2 延时值的大小与 xdc 中的 input delay 约束是否能收敛密切相关。一般情况下, 当 input delay 约束的 setup time 出现违例, 应该将 IDELAYE2 的 tap 数减小, 降低延时值; 当 input delay 约束的 hold time 出现违例, 应该将 IDELAYE2 的 tap 数增大, 增加延时值。

system_gmii_to rgmii_0_0.xdc 中包含了设置 IDELAYE2 的 tap 数的模板, 如下图所示。

```
#-----
# To Adjust GMII Rx Input Setup/Hold Timing
# These IDELAY Tap values are given for illustration
# purpose. Please modify as per design requirements
#-----
#set_property IDELAY_VALUE "16" [get_cells -hier -filter {name == *system_gmii_to rgmii_0_0_core/*delay_rgmii_rx_ctl}]
#set_property IDELAY_VALUE "16" [get_cells -hier -filter {name == *system_gmii_to rgmii_0_0_core/*delay_rgmii_rxd*}]
#set_property IODELAY_GROUP "gpr1" [get_cells -hier -filter {name == *system_gmii_to rgmii_0_0_core/*delay_rgmii_rx_ctl}]
#set_property IODELAY_GROUP "gpr1" [get_cells -hier -filter {name == *system_gmii_to rgmii_0_0_core/*delay_rgmii_rxd*}]
#set_property IODELAY_GROUP "gpr1" [get_cells -hier -filter {name == *i_system_gmii_to rgmii_0_0_idelayctrl}]
```

该段约束默认是被注释的, 默认的延时 tap 数为 16, 我们可以将这段约束复制到工程自己新建的 xdc 文件中, 编译工程后根据时序报告查看 input delay 是否时序收敛, 若存在时序违例, 则需要根据实际情况调整 tap 的值。在本例程中, 针对 miz701n-7010 开发板, 经过尝试后, 最佳的 tap 数为 14。

15.3.4 IO 口

在 block design 自动生成的 system_wrapper.v 文件中, 手动添加 2 个输出端口, 如下所示。

```

    output sfp_tx_disable;
    output sfp_rate_sel;

```

这两个端口连接到 SFP，将 `sfp_tx_disable` 置 0，`sfp_rate_sel` 置 1。如下所示。

```

assign sfp_tx_disable = 1'b0;
assign sfp_rate_sel = 1'b1;

```

15.4 PS 程序设计

所有的驱动程序文件均包含在 `c_driver` 文件夹中。

15.4.1 LWIP 库修改

在 SDK 2016.4 中所使用的 LWIP 1.4.1 库的版本为 1.7。本例程中，由于 ETH1 通过 EMIO 连接了 GMII to RGMII 这个 IP 核，若直接使用原版 LWIP 库会存在数据收发异常的现象。

原因在于，该 IP 核内部存在 1 个寄存器，如下图所示。

Control Register

This register is 16-bits wide. Its address is 0x10. The composition of this register is similar to the IEEE standard 802.3 MDIO control register 0x0, which is shown in [Table 2-4](#).

Table 2-4: Core-Specific Control Register (Address 0x10)

Bit	Name	Description	R/W
15	Reset	1 = Resets the core and this register 0 = Normal operation	R/W Self-clearing
14	Reserved	Write as 0, ignore on read	R/W
13	Speed Selection (LSB)	6 13 1 1 = Reserved 1 0 = 1,000 Mb/s 0 1 = 100 Mb/s 0 0 = 10 Mb/s	R/W
12:9	Reserved	Write as 0, ignore on read	R/W
8:7	Reserved	Write as 0, ignore on read	R/W
6	Speed Selection (MSB)	6 13 1 1 = Reserved 1 0 = 1,000 Mb/s 0 1 = 100 Mb/s 0 0 = 10 Mb/s	R/W
5:0	Reserved	Write as 0, ignore on read	R/W

本例程中，PS 的 ETH1 的 MDIO 连接到了 GMII to RGMII，PS 需要通过 MDIO 正确配置该寄存器的值，来选择当前 PHY 芯片的工作速率。而 GMII to RGMII IP 核则根据该寄存器的值来切换其于 PHY 芯片连接的 RGMII 接口的时钟频率（125M、25M、15.5M）和数据位宽。因此，若 PS 无法正确配置该寄存器，则 IP 核将无法正常工作。

打开 SDK 自带的 LWIP 库中 `xemacpsif_physpeed.c` 文件，我们可以发现其中 `phy_setup()` 函数里其实已经包含了当 PS 的 ETH 通过 EMIO 连接 GMII to RGMII 时，通过 MDIO 配置该寄存器的代码，如下图所示。

```

#define XPAR_GMII2RGMIICON_0N_ETH0_ADDR
    convphyaddr = XPAR_GMII2RGMIICON_0N_ETH0_ADDR;
    conv_present = 1;
#endif|
#define XPAR_GMII2RGMIICON_0N_ETH1_ADDR
    convphyaddr = XPAR_GMII2RGMIICON_0N_ETH1_ADDR;
    conv_present = 1;
#endif

#ifndef CONFIG_LINKSPEED_AUTODETECT
    link_speed = get_IEEE_phy_speed(xemacpsp, phy_addr);
    if (link_speed == 1000) {
        SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,1000);
        convspeeddupsetting = XEMACPS_GMII2RGMII_SPEED1000_FD;
    } else if (link_speed == 100) {
        SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,100);
        convspeeddupsetting = XEMACPS_GMII2RGMII_SPEED100_FD;
    } else if (link_speed != XST_FAILURE){
        SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,10);
        convspeeddupsetting = XEMACPS_GMII2RGMII_SPEED10_FD;
    } else {
        xil_printf("Phy setup error \r\n");
        return XST_FAILURE;
    }
#elif defined(CONFIG_LINKSPEED1000)
    SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,1000);
    link_speed = 1000;
    configure_IEEE_phy_speed(xemacpsp, phy_addr, link_speed);
    convspeeddupsetting = XEMACPS_GMII2RGMII_SPEED1000_FD;
    sleep(1);
#elif defined(CONFIG_LINKSPEED100)
    SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,100);
    link_speed = 100;
    configure_IEEE_phy_speed(xemacpsp, phy_addr, link_speed);
    convspeeddupsetting = XEMACPS_GMII2RGMII_SPEED100_FD;
    sleep(1);
#elif defined(CONFIG_LINKSPEED10)
    SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,10);
    link_speed = 10;
    configure_IEEE_phy_speed(xemacpsp, phy_addr, link_speed);
    convspeeddupsetting = XEMACPS_GMII2RGMII_SPEED10_FD;
    sleep(1);
#endif
    if (conv_present) {
        XEmacPs_PhphyWrite(xemacpsp, convphyaddr,
        XEMACPS_GMII2RGMII_REG_NUM, convspeeddupsetting);
    }
}

```

然而，由于相关头文件中缺少了两个宏定义：XPAR_GMII2RGMIICON_0N_ETH0_ADDR 和 XPAR_GMII2RGMIICON_0N_ETH1_ADDR，使得通过 MDIO 配置 GMII to RGMII 寄存器的代码不会被编译执行。因此，直接使用自带 SDK 库将无法完成该寄存器的配置，IP 核也将无法正常工作。按理，vivado 中已经设置了 GMII to RGMII 的 IP 核及其地址，导入 SDK 后应该自动生成这两个宏定义，这可能是软件设计时的疏忽之处。为此，我们需要自己修改 LWIP 库来增加这两个缺失的宏定义。

首先，找到 SDK 安装目录下的 LWIP 库的路径，例如：

C:\Xilinx\SDK\2016.4\data\embeddedsw\ThirdParty\sw_service

将 lwip141_v1_7 文件夹复制一份到工程目录的 sdk_repo\bsp 文件夹下，将其重新命名为 lwip141_v1_73。第一步，修改 lwip141_v1_73\data\lwip141.mld 文件（可用 Notepad++ 等编辑器打开），将其中的版本编号

OPTION VERSION = 1.7;

修改为

OPTION VERSION = 1.73;

然后，在其中增加如下字段：

BEGIN CATEGORY emio_options

PARAM name = emio_options, desc = "Settings for ETH using EMIO in PL";

PARAM name = use_gmii2rgmii_core_on_eth0, desc = "Settings for ETH0 using GMII to RGMII ip core in PL", type = bool, default = false;

PARAM name = use_gmii2rgmii_core_on_eth1, desc = "Settings for ETH1 using GMII to RGMII ip core in PL", type = bool, default = false;

PARAM name = gmii2rgmii_core_address_on_eth0, desc = "Settings for ETH0's PHY address of GMII to RGMII ip core in PL", type = int, default = 0;

PARAM name = gmii2rgmii_core_address_on_eth1, desc = "Settings for ETH1's PHY address of GMII to RGMII ip core in PL", type = int, default = 0;

PARAM name = use_1000basex_on_88e1512, desc = "Settings for operation mode of 88e1512, 1000/100/10 baset or 1000basex", type = bool, default = false;

END CATEGORY

增加这段代码的目的是为了在 SDK 中 BSP 设置里，lwip 参数设置对话框里增加一栏选项 emio_options，其中包含了 5 个子选项，如下图所示。

Configuration for library: [lwip141](#)

Name	Value	Def...	Type	Description
api_mode	RAW ...	RA...	enum	Mode of operation for lwIP (RAW API/Sockets API)
socket_mode_thread_prio	2	2	integer	Priority of threads in socket mode
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axiethernet adapter being use...
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures emaclite adapter being used in...
▷ arp_options	true	true	boolean	ARP Options
▷ debug_options	true	true	boolean	Turn on lwIP Debug?
▷ dhcp_options	true	true	boolean	Is DHCP required?
▷ emio_options				Settings for ETH using EMIO in PL
gmii2rgmii_core_address_on_eth0	0	0	integer	Settings for ETH0's PHY address of GMII to RGMII ip cor...
gmii2rgmii_core_address_on_eth1	8	0	integer	Settings for ETH1's PHY address of GMII to RGMII ip cor...
use_1000basex_on_88e1512	false	false	boolean	Settings for operation mode of 88e1512, 1000/100/10 b...
use_gmii2rgmii_core_on_eth0	false	false	boolean	Settings for ETH0 using GMII to RGMII ip core in PL
use_gmii2rgmii_core_on_eth1	true	false	boolean	Settings for ETH1 using GMII to RGMII ip core in PL
▷ icmp_options	true	true	boolean	ICMP Options
igmp_options	false	false	boolean	IGMP Options
lwip_ip_options	true	true	boolean	IP Options
lwip_memory_options				Options controlling lwIP memory usage
pbuf_options	true	true	boolean	Pbuf Options
stats_options	true	true	boolean	Turn on lwIP statistics?
tcp_options	true	true	boolean	Is TCP required ?
temac_adapter_options	true	true	boolean	Settings for xps-ll-temac/Axi-Ethernet/Gem lwIP adapter
udp_options	true	true	boolean	Is UDP required ?

其中，use_gmii2rgmii_core_on_eth0 和 use_gmii2rgmii_core_on_eth1 分别表示 ETH0 和 ETH1 是否通过 EMIO 连接了 GMII to RGMII IP 核。gmii2rgmii_core_address_on_eth0 和 gmii2rgmii_core_address_on_eth1 分别表示与 ETH0 和 ETH1 所连接的 GMII to RGMII IP 核中所设置

的 PHY address，此处的设置需要和 vivado 中的 IP 核设置完全一致。use_1000basex_on_88e1512 表示子卡中的 88E1512 芯片是否处于 1000BASEX 工作模式，即是否使用使用子卡中的 SFP 接口。

接着，打开 lwip141_v1_73\data\lwip141.tcl 文件，在 proc generate_lwip_opts {libhandle} 所在的大括号内其中增加如下字段：

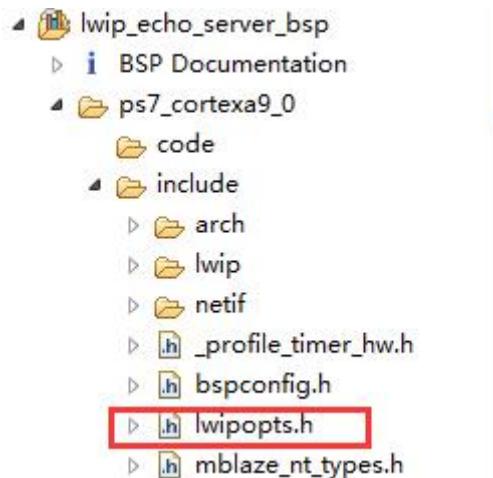
```
# EMIO options
set use_gmii2rgmii_core_on_eth0 [common::get_property CONFIG.use_gmii2rgmii_core_on_eth0 $libhandle]
set use_gmii2rgmii_core_on_eth1 [common::get_property CONFIG.use_gmii2rgmii_core_on_eth1 $libhandle]
set gmii2rgmii_core_address_on_eth0 [common::get_property CONFIG.gmii2rgmii_core_address_on_eth0 $libhandle]
set gmii2rgmii_core_address_on_eth1 [common::get_property CONFIG.gmii2rgmii_core_address_on_eth1 $libhandle]
set use_1000basex_on_88e1512 [common::get_property CONFIG.use_1000basex_on_88e1512 $libhandle]

if { $use_gmii2rgmii_core_on_eth0 == true } {
    puts $lwipopts_fd "#define XPAR_GMII2RGMIIICON_ON_ETH1_ADDR $gmii2rgmii_core_address_on_eth0"
}

if { $use_gmii2rgmii_core_on_eth1 == true } {
    puts $lwipopts_fd "#define XPAR_GMII2RGMIIICON_ON_ETH1_ADDR $gmii2rgmii_core_address_on_eth1"
}

if { $use_1000basex_on_88e1512 == true } {
    puts $lwipopts_fd "#define USE_1000BASEX"
}
puts $lwipopts_fd ""
```

增加这段代码的目的是为了在工程所对应的 bsp 中的 lwipopts.h 头文件里，如下所示。



是否增加

```
#define XPAR_GMII2RGMIIICON_ON_ETH0_ADDR 8 或
#define XPAR_GMII2RGMIIICON_ON_ETH1_ADDR 8 或
#define USE_1000BASEX
```

的宏定义。

由于 88E1512 芯片可以连接 RJ45 电口或者 SFP 光口，且完全自适应，当芯片工作于两种不同模式时，其通过 MDIO 接口所需访问或配置的寄存器便存在较大差异。在 lwip141_v1_73\src\contrib\ports\xilinx\netif\xemacpsif_physpeed.c 源文件中的 get_Marvell_phy_speed() 函数仅包含了当芯片工作于电口模式时的寄存器读写操作。

因此，需要添加当其工作于光口 1000BASEX 模式时的寄存器读写操作。上面提到的宏定义 #define USE_1000BASEX 就是为了用来设置 get_Marvell_phy_speed() 函数对于 88E1512 芯片的寄存器是使用电口配置还是光口配置。

添加的代码如下，增加条件编译#ifndef USE_1000BASEX，如下图所示。

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

XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
XEmacPs_PhysRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
control |= IEEE_RGMII_TXRX_CLOCK_DELAYED_MASK;
XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, control);

#ifndef USE_1000BASEX

    XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

    XEmacPs_PhysRead(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
    control |= IEEE_ASYMMETRIC_PAUSE_MASK;
    control |= IEEE_PAUSE_MASK;
    control |= ADVERTISE_100;
    control |= ADVERTISE_10;
    XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

#endif
```

添加其工作于 1000BASEX 模式时的寄存器读写配置代码，并设置条件编译，如下所示。

```

#else

    XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 1);

    XEmacPs_PhysRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
    control |= IEEE_STAT_AUTONEGOTIATE_RESTART;
    //control &= IEEE_CTRL_ISOLATE_DISABLE;
    control &= 0xFBFF;
    XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

    XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    XEmacPs_PhysRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    control |= IEEE_CTRL_RESET_MASK;
    XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

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

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

    XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 1);
    XEmacPs_PhysRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);
    while ( !(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE) ) {
        sleep(1);
        XEmacPs_PhysRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET,
                          &status);
    }
    xil_printf("autonegotiation complete \r\n");

//XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 1);
XEmacPs_PhysRead(xemacpsp, phy_addr, IEEE_PARTNER_ABILITIES_1_REG_OFFSET, &temp);
if ((temp & 0x0020) == 0x0020) {
    XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    return 1000;
}
else {
    XEmacPs_PhysWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    xil_printf("Link error, temp = %x\r\n", temp);
    return 0;
}

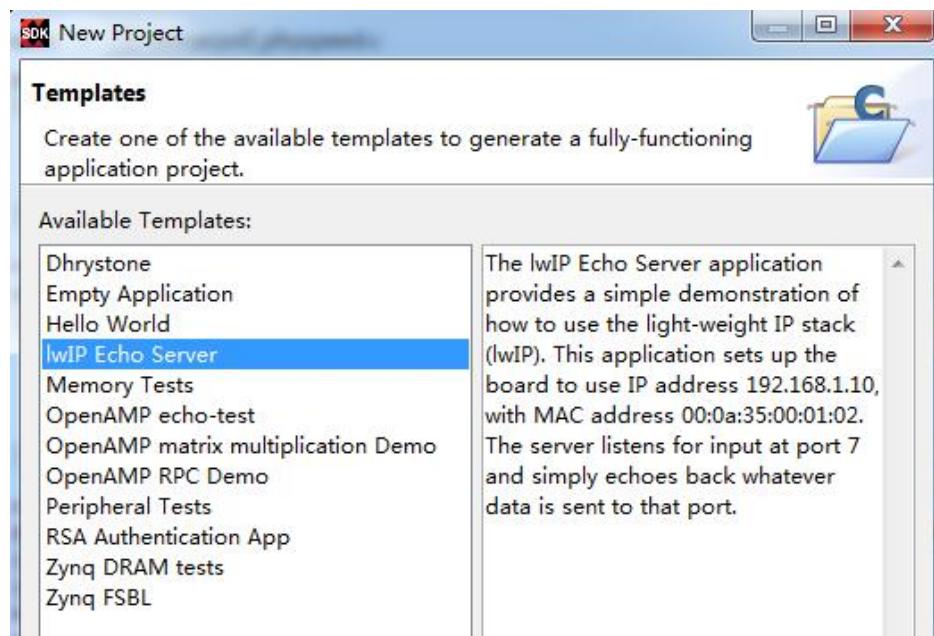
#endif

```

到此，LWIP 库的修改完成。

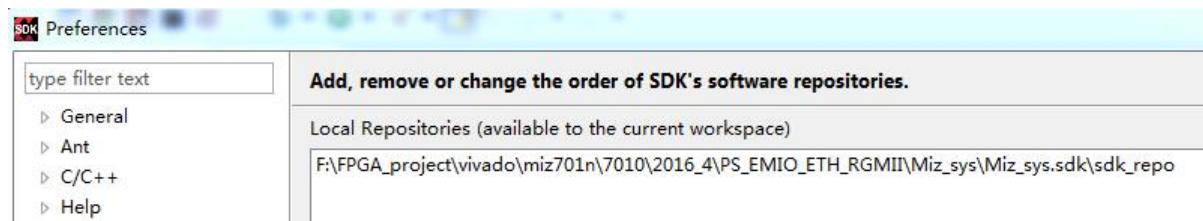
15.4.2 创建工程

本例程使用了 SDK 自带的 lwip echo server 例程来验证子卡电口和光口的功能，因此在创建工程时选择 LwIP Echo Server 模板，如下图所示。该例程基于 LWIP 库在 ARM 中建立一个 TCP Echo Server，IP 地址为 1915.168.1.10，端口号为 7。



15.4.15.1 lwip 库设置

在 SDK 软件中选择修改后的 lwip 库的路径（需根据实际情况更改此路径，若不更改将产生错误），如下图所示。



修改完成后，打开 BSP 设置，此时 lwip echo server 例程使用的是 SDK 自带的 1.7 版本 LWIP 库，为了替换成我们所修改过的 1.73 版本，首先需要取消 lwip141 1.7 的勾选，如下图所示。



然后关闭 SDK，然后重启 SDK 打开该工程目录。然后打开 BSP 设置，此时 lwip 库便变成了所修改过的 1.73 版本，如下图所示。勾选 lwip141 v1.73 版本。

Supported Libraries

Check the box next to the libraries you want included in your Board Support Package. You can use the search bar or scroll through the list in the navigator on the left.

Name	Version	Description
libmetal	1.1	Libmetal Library
lwip141	1.73	LwIP TCP/IP Stack library: LwIP v1.4.1
openamp	1.2	OpenAmp Library
xilffs	3.5	Generic Fat File System Library
xilflash	4.2	Xilinx Flash library for Intel/AMD CFI compliant ...
xilisf	5.7	Xilinx In-system and Serial Flash Library
xilmfs	2.2	Xilinx Memory File System
xilpm	2.0	Power Management API Library for ZynqMP
xilrsa	1.2	Xilinx RSA Library
xilskey	6.1	Xilinx Secure Key Library

将 use_axieth_on_zynq 和 use_emaclite_on_zynq 设为 0。如下图所示。

Configuration for library: lwip141				
Name	Value	Default	Type	Description
api_mode	RAW_API	RAW_API	enum	Mode of operation for LwIP (I
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axi
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures em

在 lwip 库增加的选项中，由于子卡通过 GMII to RGMII IP 核与 ETH1 连接，因此，将 use_gmii2rgmii_core_on_eth1 设置为 true，GMII to RGMII IP 核对应的 PHY 地址为 8，即 gmii2rgmii_core_address_on_eth1 设为 8。ETH0 不使用 EMIO，因此 use_gmii2rgmii_core_on_eth0 为 false，gmii2rgmii_core_address_on_eth0 无需进行设置。

当子卡的电口工作时，use_1000basex_on_88e1512 设置为 false；当工作于光口模式时，将其设为 true 即可。如下图所示。

emio_options				
gmii2rgmii_core_address_on_eth0	0	0	integer	
gmii2rgmii_core_address_on_eth1	8	0	integer	
use_1000basex_on_88e1512	false	false	boolean	
use_gmii2rgmii_core_on_eth0	false	false	boolean	
use_gmii2rgmii_core_on_eth1	true	false	boolean	

其余选项的参数默认即可，不用修改。

15.4.15.2 example 代码修改

将 platform_config.h 的关于 EMAC 的宏定义改为 ETH1，如下图所示。

```
#define PLATFORM_EMAC_BASEADDR XPAR_XEMACPS_1_BASEADDR
```

其余部分无需修改。

15.5 程序测试

15.5.1 电口测试

15.5.1.1 lwip 库设置

电口模式下 lwip 的设置如下图所示。

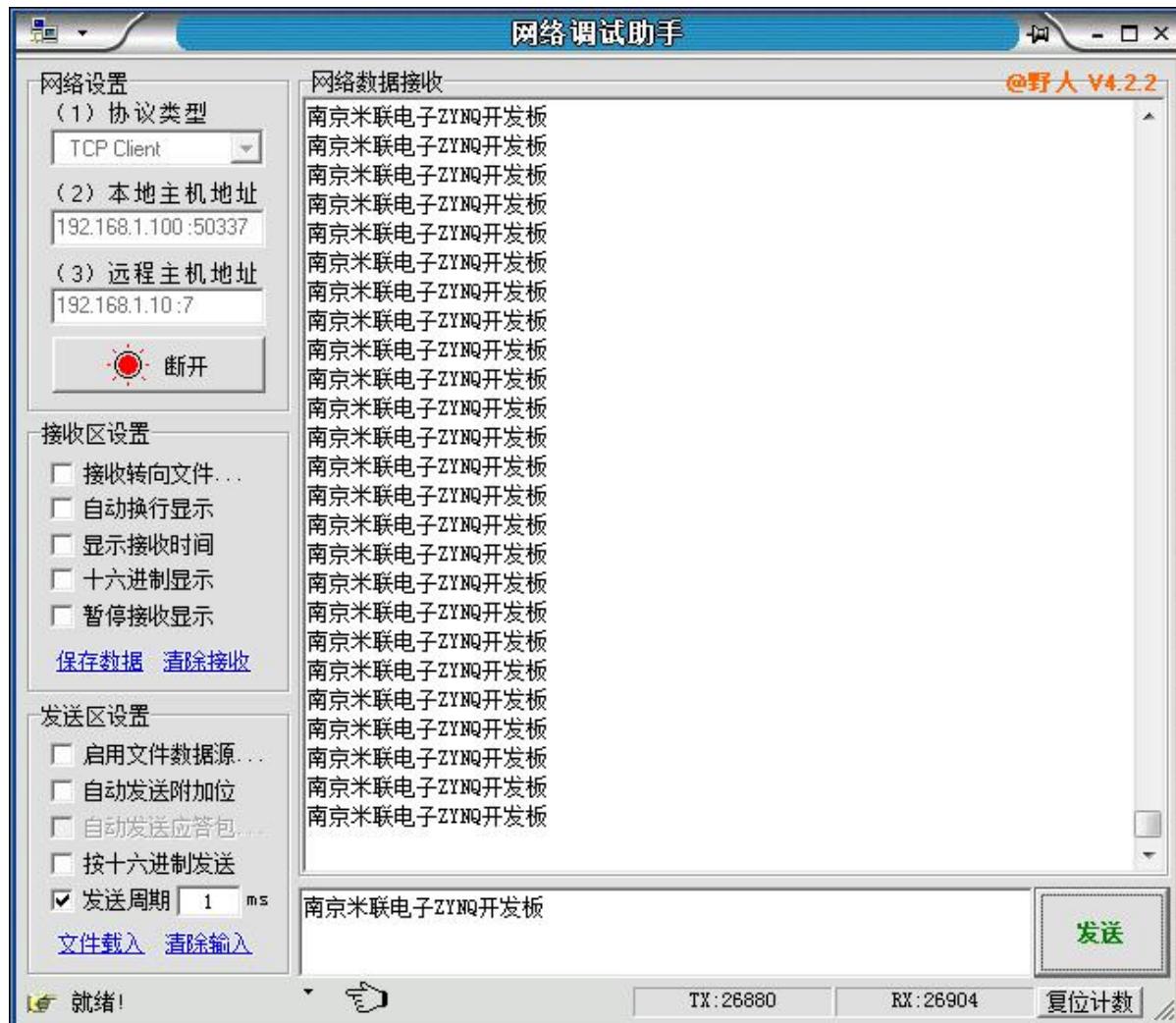
emio_options			
gmii2rgmii_core_address_on_eth0	0	0	integer
gmii2rgmii_core_address_on_eth1	8	0	integer
use_1000basex_on_88e1512	false	false	boolean
use_gmii2rgmii_core_on_eth0	false	false	boolean
use_gmii2rgmii_core_on_eth1	true	false	boolean

15.5.1.2 网络测试

将千兆网线插入子卡的 RJ45 座中，与电脑连接，将电脑的 ip 地址设为 1915.168.1.100，子网掩码为 255.255.255.0。然后给开发板上电，通过 SDK 将程序下载入开发板后，观察 SDK 串口打印信息，如下图示所示。表示千兆网络连接正常。

```
Connected to: Serial ( COM12, 115200, 0, 8 )
-----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
Board IP: 192.168.1.10
Netmask : 255.255.255.0
Gateway : 192.168.1.1
TCP echo server started @ port 7
```

打开网络调试助手，以 TCP Client 模式连接开发板的 IP 地址 1915.168.1.10，端口号 7，输入文字发送，网络调试助手便可收到开发板返回相同的文字，如下图所示。



15.5.2 光口测试

15.5.15.1 lwip 库设置

光口模式下 lwip 的设置如下图所示，要将 use_1000basex_on_88e1512 设为 true。

emio_options			
gmii2rgmii_core_address_on_eth0	0	0	integer
gmii2rgmii_core_address_on_eth1	8	0	integer
use_1000basex_on_88e1512	true	false	boolean
use_gmii2rgmii_core_on_eth0	false	false	boolean
use_gmii2rgmii_core_on_eth1	true	false	boolean

15.5.15.2 网络测试

将 SFP 电口模块插入子卡的 SFP 屏蔽笼中，将千兆网线插入 SFP 电口模块，与电脑连接，将

电脑的 ip 地址设为 1915.168.1.100，子网掩码为 255.255.255.0。然后给开发板上电，通过 SDK 将程序下载入开发板后，观察 SDK 串口打印信息，如下图所示。

```
Connected to: Serial ( COM12, 115200, 0, 8 )

-----lwIP TCP echo server -----

TCP packets sent to port 6001 will be echoed back

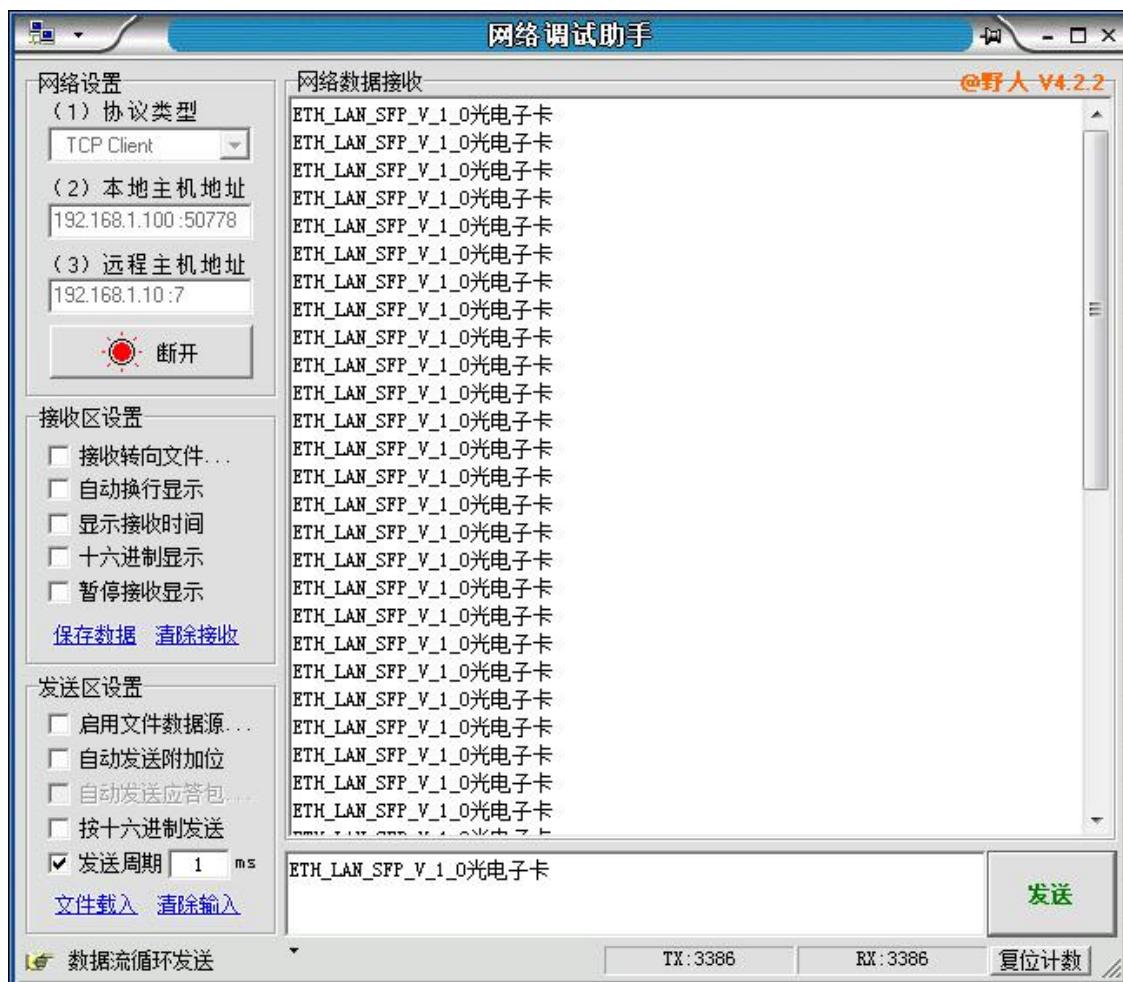
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
Board IP: 192.168.1.10

Netmask : 255.255.255.0

Gateway : 192.168.1.1

TCP echo server started @ port 7
```

同理，打开网络调试助手，以 TCP Client 模式连接开发板的 IP 地址 1915.168.1.10，端口号 7，输入文字发送，网络调试助手便可收到开发板返回相同的文字，如下图所示。



S04_CH16 PL AXI ETH 光电网络通信

16.1 概述

关于在 ZYNQ 中进行以太网应用的开发，目前在米联的教程中已经讲解了 2 种实现方案。如下：

- 将 PS 的 ENET0/ENET1 通过 MIO 以 RGMII 接口连接外部 PHY 芯片。
- 将 PS 的 ENET0/ENET1 通过 EMIO 的方式扩展至 PL，在 PL 中再通过 RGMII 接口连接外部 PHY 芯片。

本例程将基于米联电子设计的光电双口扩展子卡 FEP_ETH_SFP_CARD 介绍第 3 种应用方案。本例程完全脱离 PS 内部的以太网外设资源，在 PL 中分别通过 AXI 1G/2.5G Ethernet Subsystem 和 AXI Direct Memory Access 这两个 IP 核来实现 PS 中以太网外设 GEMAC 和 DMA 的功能，PS 通过 AXI 总线控制这两个 IP 核便可由子卡实现 LWIP 网络通信。

本例程基于 Vivado 2016.4 版本开发，参考 xapp1082 和 fpgadeveloper 工程师的应用工程：

<https://github.com/fpgadeveloper/ethernet-fmc-axi-eth>。

16.2 基本原理

本例程在 PL 中搭建了 1 个 AXI 1G/2.5G Ethernet Subsystem 以及 1 个 AXI Direct Memory Access IP 核。这两个 IP 核均通过 AXI 总线经 S_AXI_HP0 口与 PS 连接，PS 通过 AXI 总线对其进行配置和控制。其中，AXI 1G/2.5G Ethernet Subsystem IP 核通过 RGMII 接口与外部子卡中的 88E1512 芯片连接。在 PS 端通过 SDK 自带的 lwip echo server 例程，通过子卡分别以 RJ45 电口和 SFP 电口两种方式与 PC 机实现 TCP 网络通信。

16.2.1 88E1512

子卡所使用的 88E1512 芯片支持电口和光口，当其上电复位后，自动进入 RGMII to Copper 以及 RGMII to 1000BASEX 自适应状态，这是由其如下图寄存器值决定。当需要使用电口时，将网线与子卡的 RJ45 连接，当需要使用光口时，在子卡的 SFP 屏蔽笼中插入光模块即可。

Fiber/Copper Auto-Selection

The device has a patented feature to automatically detect and switch between fiber and copper cable connections. The auto-selection operates in one of three modes: Copper /1000BASE-X, Copper/100BASE-FX, and Copper/SGMII Media Interface.

Register 20_18.2:0 and register 20_18.6 select the Fiber/Copper auto media modes of operation. See [Table 33](#) for details.

Table 33: Fiber/Copper Auto-media Modes of Operation

Reg 20_18.2:0	Reg 20_18.6	Auto-media Modes of Operation
110	X	Copper/SGMII media
111	0	Copper/1000BASE-X
011	1	Copper/100BASE-FX

The device monitors the signals of the S_INP/N and the MDIP/N[3:0] lines. If a fiber optic cable is plugged in, the device will adjust itself to be in fiber mode. If an RJ-45 cable is plugged in, the device will adjust itself to be in copper mode. If both cables are connected then the first media to establish link, or the preferred media will be enabled. The media which is not enabled will be automatically turned off to save power. If the link on the first media is lost, then the inactive media will be powered up, and both media will once again start searching for link.

Table 220: General Control Register 1
Page 18, Register 20

Bits	Field	Mode	HW Rst	SW Rst	Description
15	Reset	R/W, SC	0x0	SC	Mode Software Reset. Affects page 6 and 18 Writing a 1 to this bit causes the main PHY state machines to be reset. When the reset operation is done, this bit is cleared to 0 automatically. The reset occurs immediately. 1 = PHY reset 0 = Normal operation
14:13	Reserved	R/W	0x0	Retain	Set to 0s.
12:10	Reserved	R/W	0x0	Retain	Reserved for future use.
9:7	Reserved	R/W	0x4	Retain	Set to 100
6	Auto-Media Detect (AMD) 100BASE-FX/ 1000BASE-X	R/W	0x0	Retain	This bit selects the fiber auto-media modes as follows: 1 = mode 011 is AMD between copper and 100BASE-FX 0 = mode 111 is AMD between copper and 1000BASE-X
5:4	Auto Media Detect Preferred Media	R/W	0x0	Retain	00 = Link on first media 01 = Copper Preferred 10 = Fiber Preferred 11 = Reserved
3	Reserved	R/W	0x0	Update	Set to 0
2:0	MODE[2:0]	R/W	See Descr.	Update	Changes to this bit are disruptive to the normal operation; therefore, any changes to these registers must be followed by a software reset to take effect. 1512 device comes up in MODE[2:0] =0x7 on hardware reset. 000 = RGMII (System mode) to Copper 001 = SGMII (System mode) to Copper 010 = RGMII (System mode) to 1000BASE-X 011 = RGMII (System mode) to 100BASE-FX 100 = RGMII (System mode) to SGMII (Media mode) 101 = Reserved 110 = RGMII (System mode) to Auto Media Detect Copper/SGMII (Media mode) 111 = RGMII (System mode) to Auto Media Detect Copper/1000BASE-X/100BASE-FX (see 20_18.6)

16.2.2 88E1512 RGMII 接口时序

88E1512 芯片 RGMII 接口的时序存在延迟和非延迟 2 种模式，由其内部的一个寄存器值来决定，如下图所示。

Table 121: MAC Specific Control Register 2
Page 2, Register 21

Bits	Field	Mode	HW Rst	SW Rst	Description
12:7	Reserved		0x20	0x20	Reserved.
6	Default MAC interface speed (MSB)	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. Also, used for setting speed of MAC interface during MAC side loopback. Requires that customer set both these bits and force speed using register 0 to the same speed. MAC Interface Speed during Link down. Bits 6, 13 00 = 10 Mbps 01 = 100 Mbps 10 = 1000 Mbps
5	RGMII Receive Timing Control	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. 1 = Receive clock transition when data stable 0 = Receive clock transition when data transitions
4	RGMII Transmit Timing Control	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. 1 = Transmit clock internally delayed 0 = Transmit clock not internally delayed

芯片上电通过 RESET 引脚复位后，寄存器中决定 RGMII 输入输出时序关系的 bit 位的值都是 1。也就是说，上电复位后芯片的 RGMII 接口均为延迟时序模式。延迟模式是指 88E1512 芯片在其内部给输出的接收时钟信号 RX_CLK 和输入的发送的时钟信号 TX_CLK 增加了 2ns 左右的延时。这是为了避免通过 PCB 绕线的方式增加时钟信号的延时，使 RX_CLK 和 TX_CLK 都能与相应 RXD 和 TXD 数据窗的中心对齐，从而使得 RGMII 接口数据的建立和保持时间达到余量最大的状态。

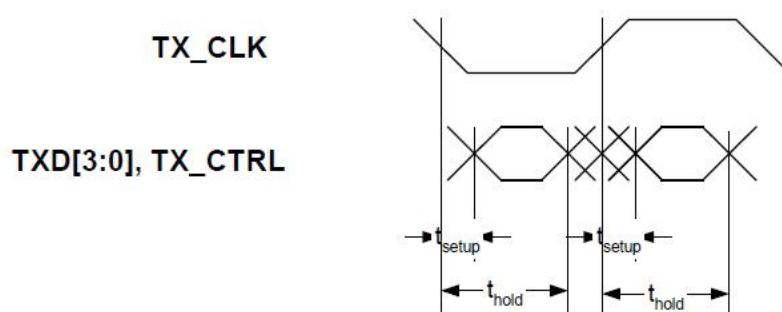
因此，此时 RGMII 接口发送部分信号的时序关系如下图所示。

4.10.2.2 PHY Input - TX_CLK Delay when Register 21_2.4 = 1

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.4 = 1 (add delay)	-0.9			ns
t_{hold}		2.7			ns

Figure 37: TX_CLK Delay Timing - Register 21_2.4 = 1 (add delay)



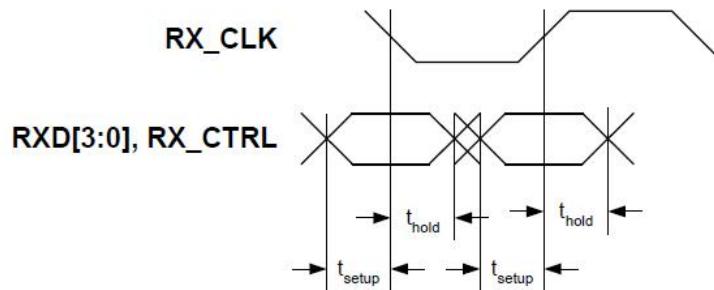
RGMII 接口接收部分信号的时序关系如下图所示。

4.10.2.4 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t _{setup}	Register 21_2.5 = 1 (add delay)	1.2			ns
t _{hold}		1.2			ns

Figure 39: RGMII RX_CLK Delay Timing - Register 21_2.5 = 1 (add delay)



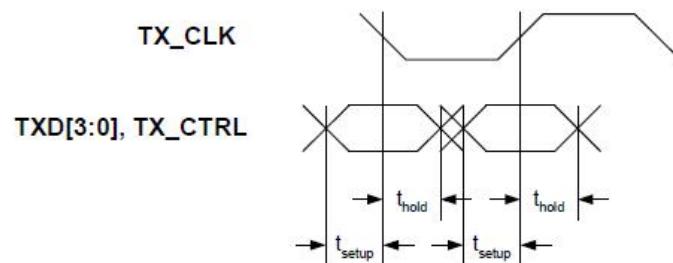
另外，非延迟模式下 RGMII 接口发送部分信号的时序关系如下图所示。

4.10.2.1 PHY Input - TX_CLK Delay when Register 21_2.4 = 0

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t _{setup}	Register 21_2.4 = 0	1.0			ns
t _{hold}		0.8			ns

Figure 36: TX_CLK Delay Timing - Register 21_2.4 = 0



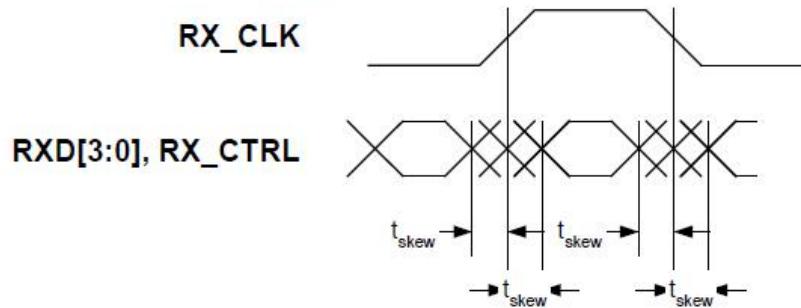
非延迟模式下 RGMII 接口接收部分信号的时序关系如下图所示。

4.10.2.3 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

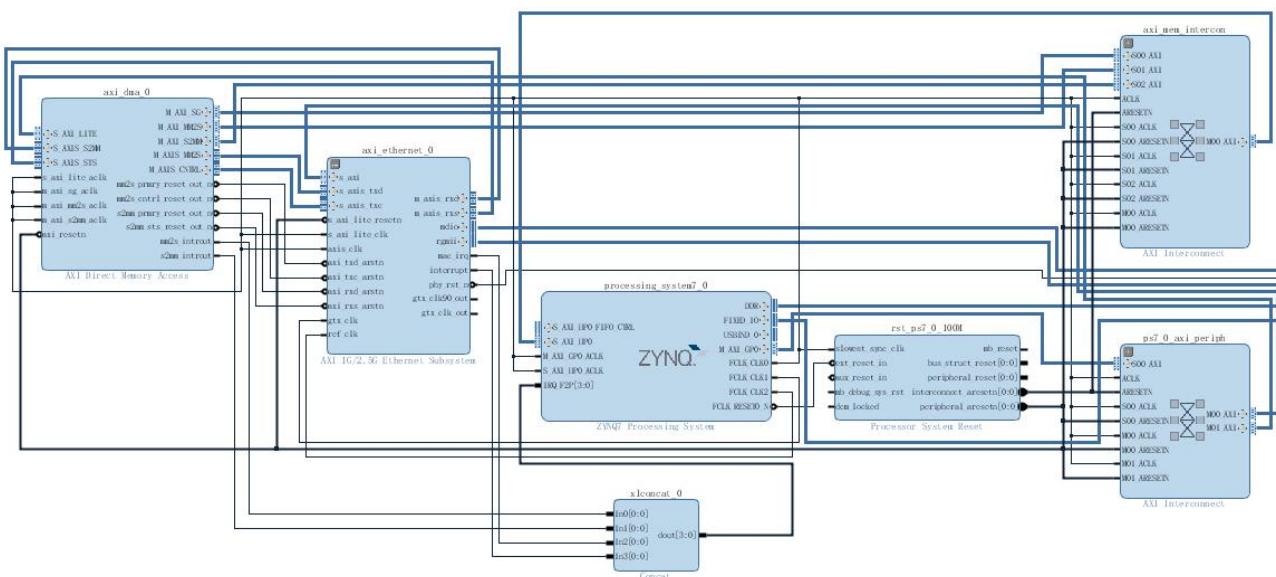
Symbol	Parameter	Min	Typ	Max	Units
t_{skew}	Register 21_2.5 = 0	- 0.5		0.5	ns

Figure 38: RGMII RX_CLK Delay Timing - Register 21_2.5 = 0



16.3 PL 部分设计

16.3.1 IP 连线图



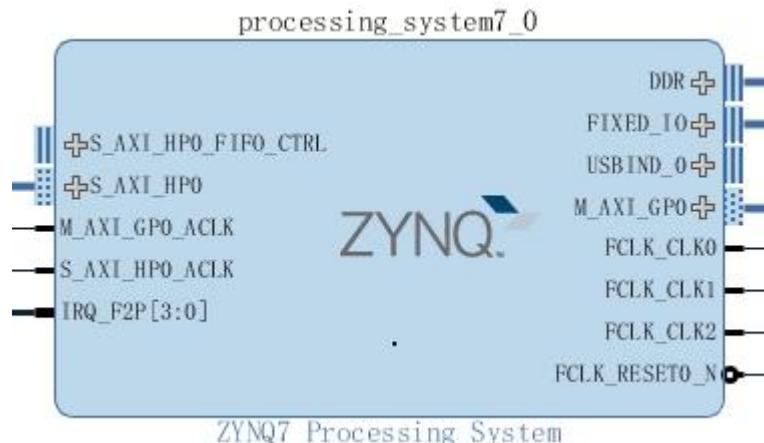
16.3.2 ZYNQ PS 设置

ZYNQ PS 需要作如下配置：

- 使能 M_AXI_GP0 接口
- 使能 S_AXI_HP0 接口
- 使能 PL 至 PS 的中断接口 IRQ_F2P
- 使能 FCLK_CLK0，设为 100M，作为 PL 部分所有 AXI 接口的时钟

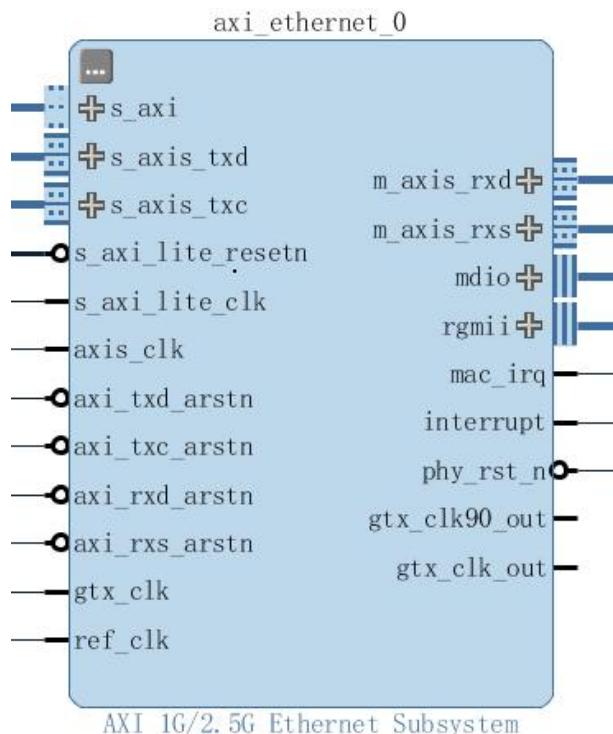
- 使能 FCLK_CLK1，设为 125M，作为 AXI 1G/2.5G Ethernet Subsystem IP 核 gtx_clk 的输入时钟。
- 使能 FCLK_CLK2，设为 200M，作为 AXI 1G/2.5G Ethernet Subsystem IP 核 ref_clk 的输入时钟，作为其内部 IDELAYCTRL 的参考时钟。

如下图所示。

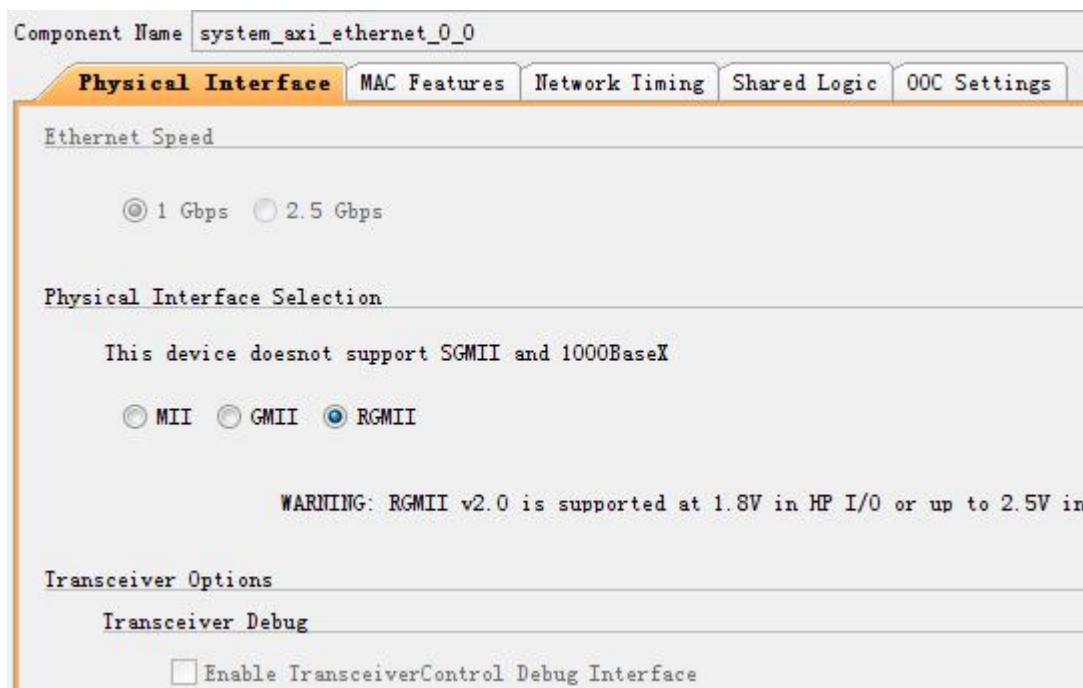


16.3.3 AXI 1G/2.5G Ethernet Subsystem

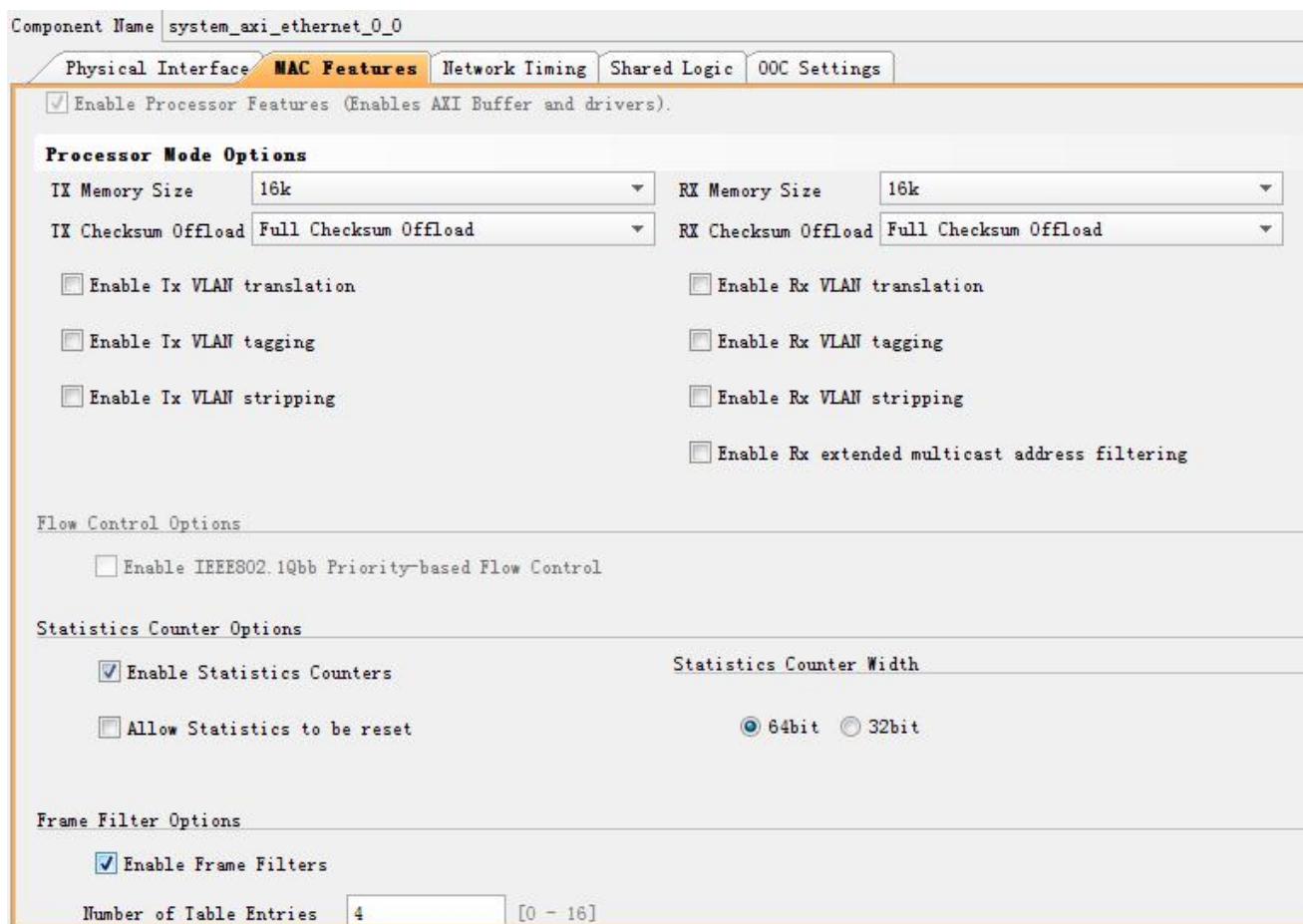
AXI 1G/2.5G Ethernet Subsystem IP 核的设置如下图所示。



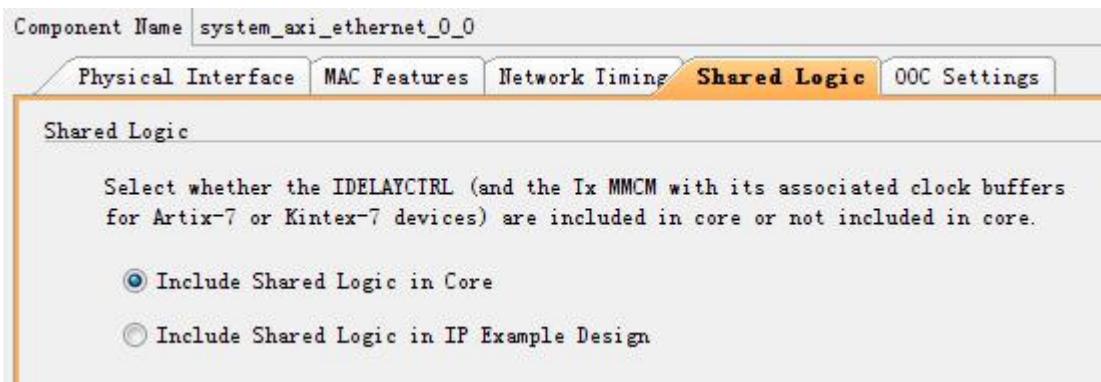
由于子卡中的 88E1512 芯片为 RGMII 接口，因此将 IP 设为 RGMII 接口，如下图所示。



TX Memory 和 RX Memory 对应 IP 核内部的接收和发送缓存，缓存越大将有利于提高接收和发送的速率，不妨将 TX Memory 和 RX Memory 都设为 16k。另外，将 TX 和 RX Checksum Offload 全部设置为 Full Checksum Offload，这将使 IP 核完成所有收发数据包中 TCP UDP IP 协议首部校验和的计算功能，使 PS 无需再进行这些校验和的计算，从而提高 PS 中网络协议栈的运行效率。（PS 内部的 GEMAC 也具有该功能）如下图所示。



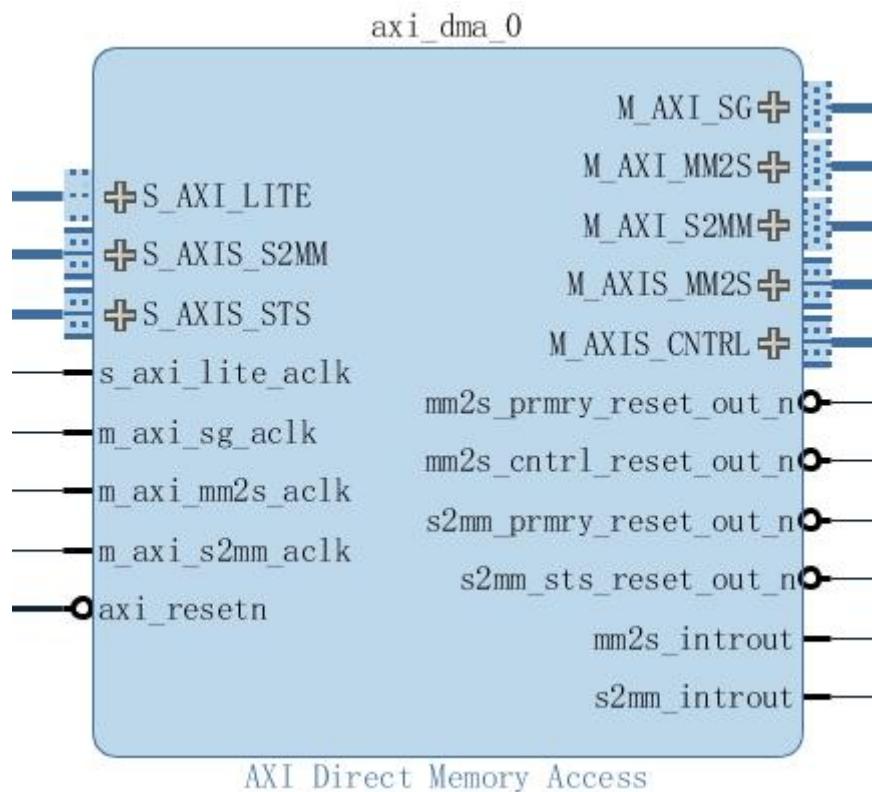
在独立使用单个 IP 核时，应选择将 shared logic 包含在 IP 核内部，如下图所示。



其余选项保持默认即可，无需更改。

16.3.4 AXI Direct Memory Access

AXI Direct Memory Access IP 核的设置如下图所示。



需要说明的是：

- 将 AXI DMA 配置为链式 DMA 工作模式
- 使能 TX Control 和 RX Status 接口，这两个接口是 AXI DMA 与 AXI 1G/2.5G Ethernet Subsystem 专用的连接接口。分别用于发送控制，和接收状态传递。
- 将 Length Register 的位宽设为 16，对应 1 次链式 DMA 传输的最大总数据量为 64KB。若想进一步增大数据传输效率，可将位宽设大，但不应该小于 AXI 1G/2.5G Ethernet Subsystem 中所设置的 TX 和 RX Memory 的值，本例程中为 16KB，对应的最小位宽应为 14。

所有的参数设置如下图所示。

Component Name `system_axi_dma_0_0`

Enable Asynchronous Clocks (Auto)

Enable Scatter Gather Engine

Enable Micro DMA

Enable Multi Channel Support

Enable Control / Status Stream

Width of Buffer Length Register (8-23) `16` bits

Address Width (32-64) `32` bits

Enable Read Channel Enable Write Channel

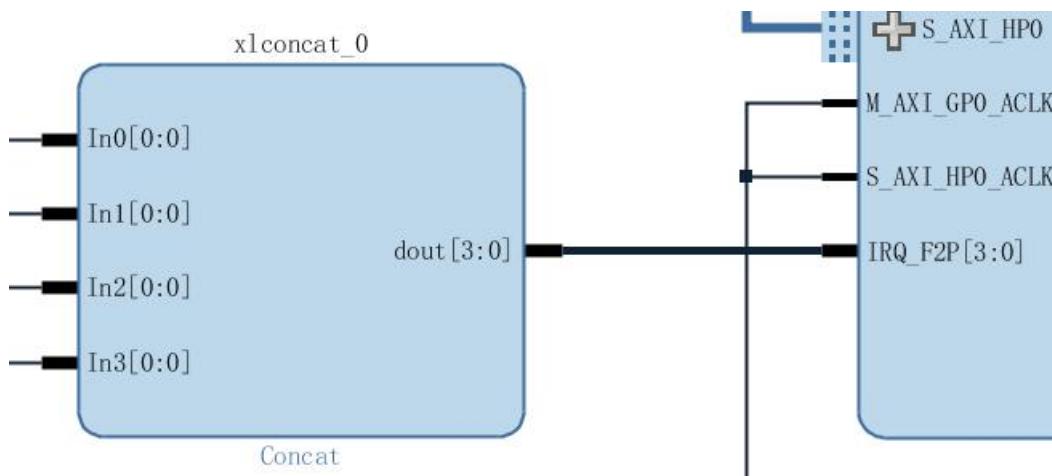
Number of Channels	<code>1</code>	Number of Channels	<code>1</code>
Memory Map Data Width	<code>32</code>	Memory Map Data Width	<code>32</code>
Stream Data Width	<code>32</code>	Stream Data Width (Auto)	<code>32</code>
Max Burst Size	<code>16</code>	Max Burst Size	<code>16</code>

Allow Unaligned Transfers Allow Unaligned Transfers

Use Rxlength In Status Stream

16.3.5 PL 至 PS 的中断

AXI 1G/2.5G Ethernet Subsystem 和 AXI DMA 分别有 2 个输出中断信号，通过 Concat 将这 4 个中断信号与 PS 的 IRQ_F2P 接口连接，如下图所示。



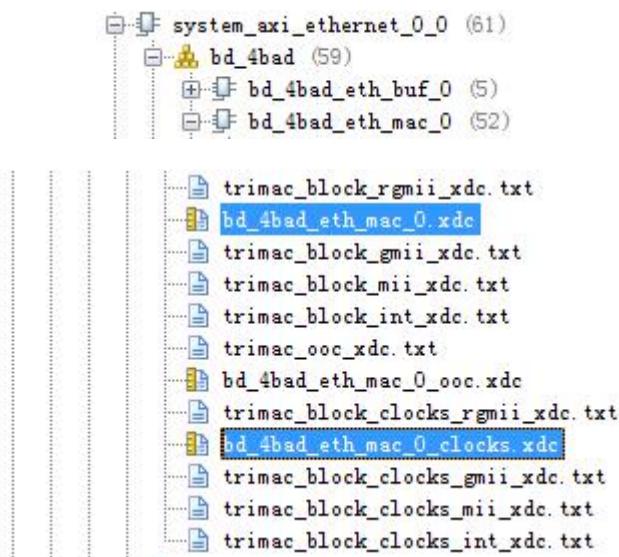
其中，AXI DMA 的 `mm2s_intout`、`s2mm_intout` 两个中断信号分别对应 TX 和 RX 两个方向的数据搬移完成、传输错误等中断。AXI 1G/2.5G Ethernet Subsystem 的 `interrupt` 为其内部各种状态的中断信号，例如自协商完成，接收错误，发送完成等；`mac_irq` 为 mdio 接口中断信号。实际在 PS

端的 LWIP 库中的驱动程序里并不使用 AXI 1G/2.5G Ethernet Subsystem 的这 2 个中断。用户可根据需求自行设计。

16.3.6 时序约束

本例程中，时序约束主要是针对 RGMII 接口进行 input delay 和 output delay 的约束，以及 IDELAYE2 延时 tap 数的设置，使 RGMII 接口的满足建立和保持时间的要求，从而达到时序收敛。

对于 input delay 和 output delay 的约束，AXI 1G/2.5G Ethernet Subsystem IP 核所自带的 bd_****_eth_mac_0_clock.xdc 文件中已经包含了默认设置。对于 IDELAYE2 延时 tap 数的设置，在 bd_****_eth_mac_0.xdc 中包含了默认模板。这两个 xdc 文件位置如下图所示。



16.3.6.1 input delay 约束

bd_****_eth_mac_0_clock.xdc 文件中，关于 input delay 的约束如下所示。约束范围为：-1.5~-2.8ns。

```
# define a virtual clock to simplify the timing constraints
create_clock -name [current_instance .]_rgmii_rx_clk -period 8
set rgmii_rx_clk [current_instance .]_rgmii_rx_clk

# Identify RGMII Rx Pads only.
# This prevents setup/hold analysis being performed on false inputs,
# eg, the configuration_vector inputs.

set_input_delay -clock [get_clocks $rgmii_rx_clk] -max -1.5 [get_ports {rgmii_rxd[*] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks $rgmii_rx_clk] -min -2.8 [get_ports {rgmii_rxd[*] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks $rgmii_rx_clk] -clock_fall -max -1.5 -add_delay [get_ports {rgmii_rxd[*] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks $rgmii_rx_clk] -clock_fall -min -2.8 -add_delay [get_ports {rgmii_rxd[*] rgmii_rx_ctl}]
```

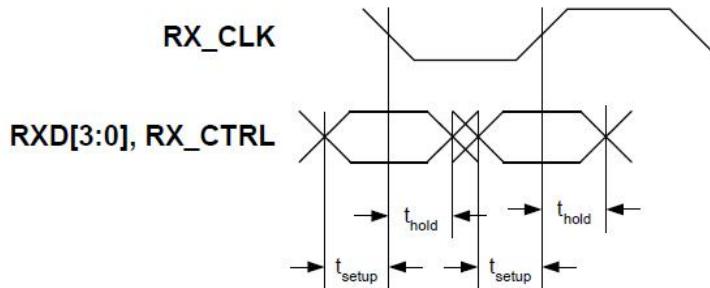
在 2.2 节中已经对 88e1512 芯片的 RGMII 接口时序进行了介绍，对于 RX 接口，如下图。

4.10.2.4 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.5 = 1 (add delay)	1.2			ns
t_{hold}		1.2			ns

Figure 39: RGMII RX_CLK Delay Timing - Register 21_2.5 = 1 (add delay)



按照上图中的 setup time 和 hold time, input delay 的-min 应该为 $-(4-1.2) = -2.8\text{ns}$, -max 应该为 -1.2ns 。显然, IP 核自带 input delay 的约束范围为 $-1.5\text{ns} \sim -2.8\text{ns}$, 比我们计算的结果 $-1.2\text{ns} \sim -2.8\text{ns}$ 略为宽松, 用于 setup time 计算所需的 -max 时间小了 0.3ns , 为了保证时序严格收敛, 需要将 bd_****_eth_mac_0_clock.xdc 文件中 set_input_delay -max 的 -1.5 全部改为 -1.2 。需要注意的是, 这些 xdc 文件由 IP 核自动生成, 每次重新配置 IP 后, vivado 都会重新生成 IP 的 output product, 因此会将被修改的 xdc 文件覆盖还原, 需要手动重新再次修改 xdc 文件才行。

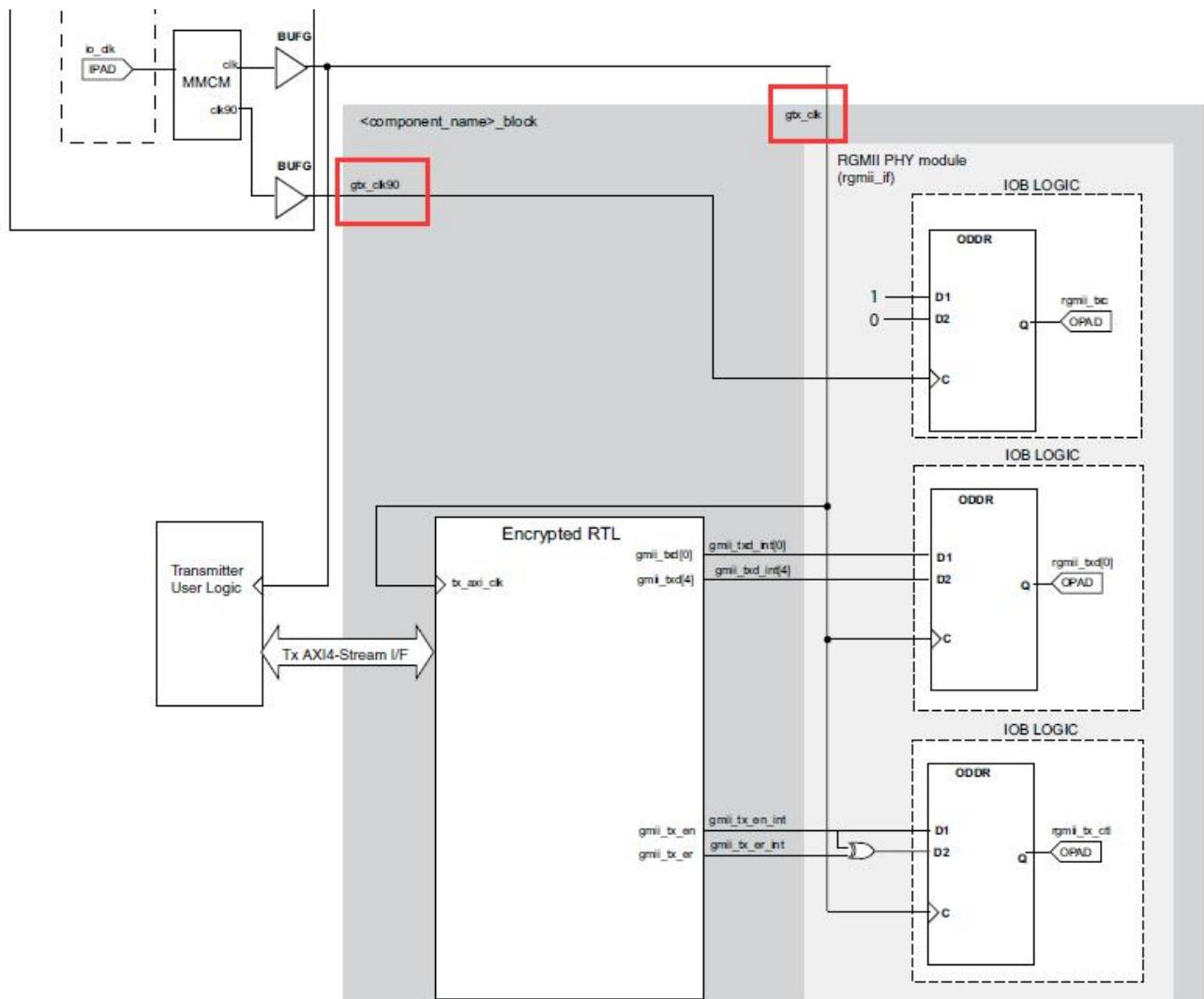
16.3.6.2 output delay 约束

bd_****_eth_mac_0_clock.xdc 文件中, 关于 output delay 的约束如下所示。约束范围为: $-0.75 \sim 0.75\text{ns}$ 。

```
create_generated_clock -name [current_instance .]_rgmii_tx_clk -divide_by 1 -source [get_pins {tri_mode_ethernet_mac_i/rgmii_interface/rgmii_txc_ddr/C}]
set rgmii_tx_clk [current_instance .]_rgmii_tx_clk

set_output_delay 0.75 -max -clock [get_clocks $rgmii_tx_clk] [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
set_output_delay -0.75 -min -clock [get_clocks $rgmii_tx_clk] [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
set_output_delay 0.75 -max -clock [get_clocks $rgmii_tx_clk] [get_ports {rgmii_txd[*] rgmii_tx_ctl}] -clock_fall -add_delay
set_output_delay -0.75 -min -clock [get_clocks $rgmii_tx_clk] [get_ports {rgmii_txd[*] rgmii_tx_ctl}] -clock_fall -add_delay
```

在 AXI 1G/2.5G Ethernet Subsystem 中, 对于 HR BANK 所采用的 RGMII 发送接口方案如下图所示。



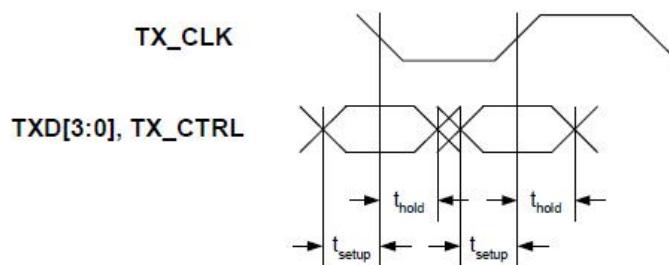
rgmii_txc 的发送时钟 gtx_clk90，与 rgmii_td、rgmii_tx_ctl 的发送时钟 gtx_clk 存在 90° 即 2ns 的相位差。因此，此时 88E1512 芯片 RGMII 的 TX 接口时序不能再使用默认的内部延迟模式，而需要使用外部延迟模式，其时序关系如下图。

4.10.2.1 PHY Input - TX_CLK Delay when Register 21_2.4 = 0

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t _{setup}	Register 21_2.4 = 0	1.0			ns
t _{hold}		0.8			ns

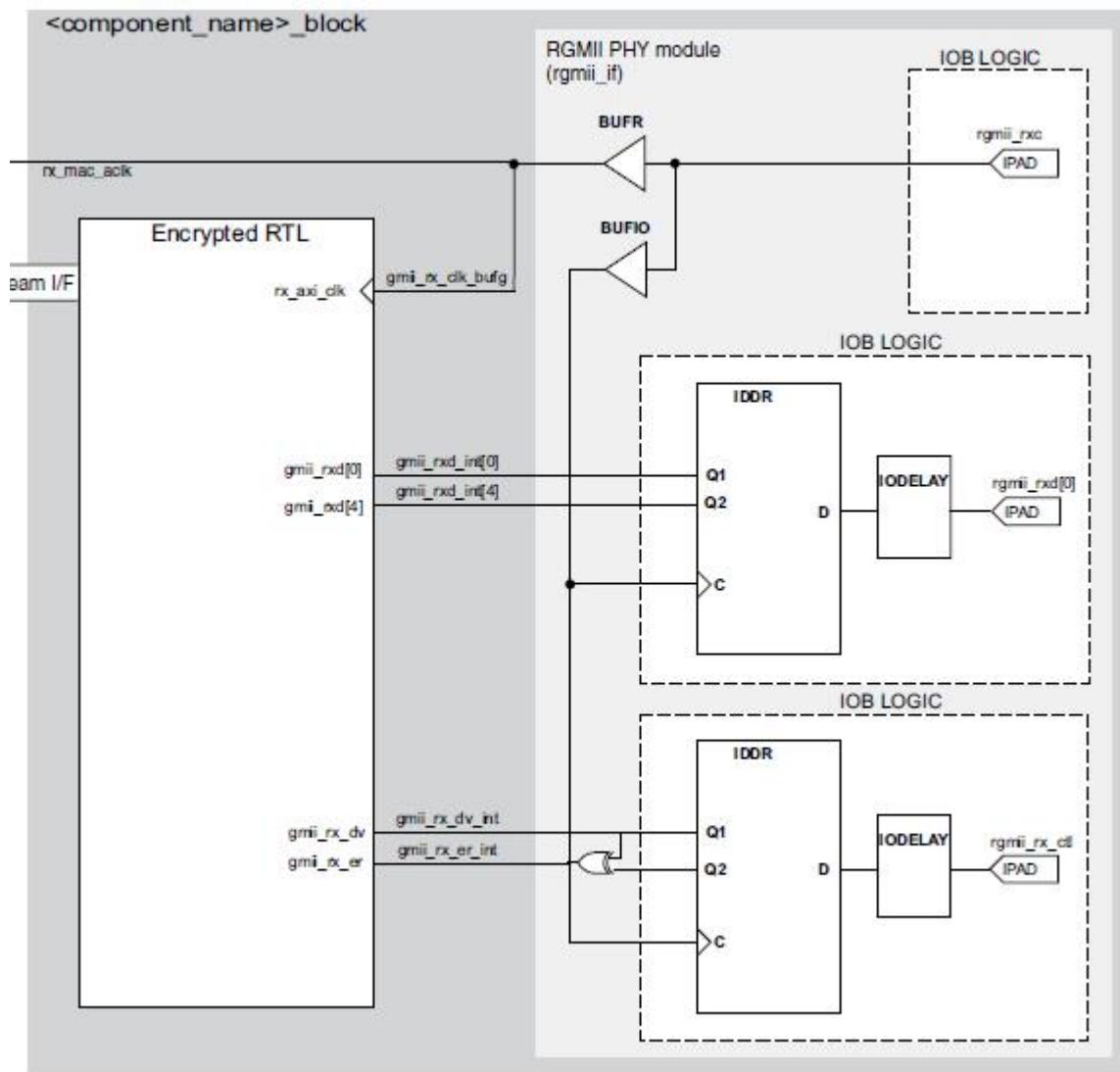
Figure 36: TX_CLK Delay Timing - Register 21_2.4 = 0



照上图中的 setup time 和 hold time, output delay 的 -min 应该为 -0.8ns, -max 应该为 1.0ns。显然, IP 核自带 output delay 的约束范围为 -0.75ns~0.75ns, 比我们计算的结果 -0.8ns~1.0ns 略为宽松, 用于 setup time 计算所需的 -max 时间小了 0.25ns, 用于 hold time 计算所需的 -min 时间大了 0.05ns。为了保证时序严格收敛, 需要将 bd_****_eth_mac_0_clock.xdc 文件中 set_output_delay -max 的 0.75 改为 1.0, -min 的 -0.75 全部改为 -0.8。

16.3.6.3 IDELAYE2 延时设置

在 AXI 1G/2.5G Ethernet Subsystem IP 核内部为 rgmii_rx_ctl 和 rgmii_rxd[3:0] 端口都连接了 IDELAYE2 模块, 如下图所示。



IDELAYE2 用来为这些信号进入 PL 内部前引入额外的延时。IDELAYE2 延时值的大小与 xdc 中的 input delay 约束是否能收敛密切相关。一般情况下, 当 input delay 约束的 setup time 出现违例, 应该将 IDELAYE2 的 tap 数减小, 降低延时值; 当 input delay 约束的 hold time 出现违例, 应该将 IDELAYE2 的 tap 数增大, 增加延时值。

bd_****_eth_mac_0.xdc 中包含了 IDELAYE2 的 tap 数的设置, 如下图所示。

```
# Apply the same IDELAY_VALUE to all RGMII RX inputs.
# This is to provide a similar Clock Path and Data Path delay.
set_property IDELAY_VALUE 15 [get_cells {tri_mode_ethernet_mac_i/rgmii_interface/delay_rgmii_rx* tri_mode_ethernet_mac_i}

# Group IODELAY components
set_property IODELAY_GROUP tri_mode_ethernet_mac_iodelay_grp [get_cells {tri_mode_ethernet_mac_i/rgmii_interface/delay_r
set_property IODELAY_GROUP tri_mode_ethernet_mac_iodelay_grp [get_cells {tri_mode_ethernet_mac_idelayctrl_common_i}]}
```

该段约束默认是被注释的，默认的延时 tap 数为 15，编译工程后根据时序报告查看 input delay 是否时序收敛，若存在时序违例，则需要根据实际情况调整 tap 的值。经过尝试后，tap 数为 15 时可以满足时序收敛的要求，因此无需修改。

16.3.7 IO 口

AXI 1G/2.5G Ethernet Subsystem IP 核的 rgmii、mdio、phy_reset_n 接口需要作为 IO 口引出与外部 PHY 芯片连接。其中，PHY 芯片的复位信号 phy_reset_n 的低电平有效。

在 block design 自动生成的 system_wrapper.v 文件中，手动添加 2 个输出端口，如下所示。

```
output sfp_tx_disable;
output sfp_rate_sel;
```

这两个端口连接到 SFP，将 sfp_tx_disable 置 0，sfp_rate_sel 置 1。如下所示。

```
assign sfp_tx_disable = 1'b0;
assign sfp_rate_sel = 1'b1;
```

16.4 PS 程序设计

所有的驱动程序文件均包含在 c_driver 文件夹中。

16.4.1 LWIP 库修改

在 SDK 2016.4 中所使用的 LWIP 1.4.1 库的版本为 1.7。其中缺少当 PS 连接 AXI 1G/2.5G Ethernet Subsystem IP 核时对于 88E1512 芯片的配置驱动程序，若直接使用原版 LWIP 库将使 88E1512 芯片无法正常工作，从而无法进行数据传输。因此，需要手动修改库文件添加驱动程序。

首先，找到 SDK 安装目录下的 LWIP 库的路径，例如：

C:\Xilinx\SDK\2016.4\data\embeddedsw\ThirdParty\sw_service

将 lwip141_v1_7 文件夹复制一份到工程目录的 sdk_repo\bsp 文件夹下，将其重新命名为 lwip141_v1_74。第一步，修改 lwip141_v1_74\data\lwip141.mld 文件（可用 Notepad++ 等编辑器打开），将其中的版本编号

OPTION VERSION = 1.7;

修改为

OPTION VERSION = 1.74;

然后，在其中增加如下字段：

```
PARAM name = use_1000basex_on_88e1512, desc = "Settings for operation mode of 88e1512, 1000/100/10 baset
or 1000basex", type = bool, default = false;
```

增加这段代码的目的是为了在 SDK 中 BSP 设置里，lwip 参数设置对话框里增加一个选项 use_1000basex_on_88e1512，如下图所示。

Configuration for library: lwip141

Name	Value	Default	Type	Description
api_mode	RAW API (RAW_API)	RAW_API	enum	Mode of operation for
socket mode thread prio	2	2	integer	Priority of threads in so
use_1000basex_on_88e1512	true	false	boolean	Settings for operation r
use_axieth_on_zynq	1	1	integer	Option if set to 1 ensur
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensur
▷ arp_options	true	true	boolean	ARP Options
▷ debug_options	true	true	boolean	Turn on lwIP Debug?
▷ dhcp_options	true	true	boolean	Is DHCP required?
▷ icmp_options	true	true	boolean	ICMP Options
igmp_options	false	false	boolean	IGMP Options
▷ lwip_ip_options	true	true	boolean	IP Options
▷ lwip_memory_options				Options controlling lwIP
▷ pbuf_options	true	true	boolean	Pbuf Options
▷ stats_options	true	true	boolean	Turn on lwIP statistics?
▷ tcp_options	true	true	boolean	Is TCP required ?
▷ temac_adapter_options	true	true	boolean	Settings for xps-ll-tema
▷ udp_options	true	true	boolean	Is UDP required ?

use_1000basex_on_88e1512 表示子卡中的 88E1512 芯片是否处于 1000BASEX 工作模式，即是否使用使用子卡中的 SFP 接口。

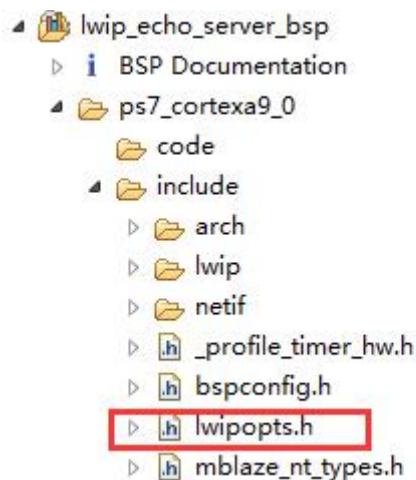
接着，打开 lwip141_v1_74\data\lwip141.tcl 文件，在 proc generate_lwip_opts {libhandle} 所在的大括号内其中增加如下字段：

```
set use_1000basex_on_88e1512 [common::get_property CONFIG.use_1000basex_on_88e1512 $libhandle]

if { $use_1000basex_on_88e1512 == true }{
    puts $lwipopts_fd "#define USE_1000BASEX"
}

puts $lwipopts_fd ""
```

增加这段代码的目的是为了在工程所对应的 bsp 中的 lwipopts.h 头文件里，如下所示。



是否增加

```
#define USE_1000BASEX
```

的宏定义。

打开 lwip141_v1_74\src\contrib\ports\xilinx\netif\xaxiemacif_physpeed.c 源文件，在其中添加函数 get_phy_speed_88E1512() 函数，对 88E1512 芯片进行配置，函数源代码不在此进行列举，详细参考工程目录文件。

需要说明的是，其中对于 88E1512 芯片 RGMII 接口时序的设置部分代码如下：

```
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
control &= ~IEEE_RGMII_TX_CLOCK_DELAYED_MASK;
control |= IEEE_RGMII_RX_CLOCK_DELAYED_MASK;
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, control);
```

对于 TX 接口不使用内部延迟模式，对于 RX 接口使用延迟模式。

上面提到的宏定义 #define USE_1000BASEX 就是作为条件编译选项，用来设置 get_phy_speed_88E1512() 函数对于 88E1512 芯片的寄存器是使用电口配置还是光口配置。

另外，添加宏定义

```
#define MARVEL_PHY_88E1512_MODEL 0x01D0
```

该宏定义为 88E1512 芯片的 ID。

然后，找到如下代码：

```
if (phy_identifier == MARVEL_PHY_IDENTIFIER) {
    phy_model = phy_model & MARVEL_PHY_MODEL_NUM_MASK;

    if (phy_model == MARVEL_PHY_88E1116R_MODEL) {
        return get_phy_speed_88E1116R(xaxiemacp, phy_addr);
    } else if (phy_model == MARVEL_PHY_88E1111_MODEL) {
        return get_phy_speed_88E1111(xaxiemacp, phy_addr);
    }
}
```

改为：

```
if (phy_identifier == MARVEL_PHY_IDENTIFIER) {
    phy_model = phy_model & MARVEL_PHY_MODEL_NUM_MASK;

    if (phy_model == MARVEL_PHY_88E1116R_MODEL) {
        return get_phy_speed_88E1116R(xaxiemacp, phy_addr);
    } else if (phy_model == MARVEL_PHY_88E1111_MODEL) {
        return get_phy_speed_88E1111(xaxiemacp, phy_addr);
    } else if (phy_model == MARVEL_PHY_88E1512_MODEL) {
        return get_phy_speed_88E1512(xaxiemacp, phy_addr);
    }
}
```

打开 lwip141_v1_74\src\contrib\ports\xilinx\netif\xaxiemacif_dma.c 源文件，在 init_axi_dma() 函数中找到如下代码：

```
dmaconfig = XAxiDma_LookupConfig(XPAR_AXIDMA_0_DEVICE_ID);
```

改为：

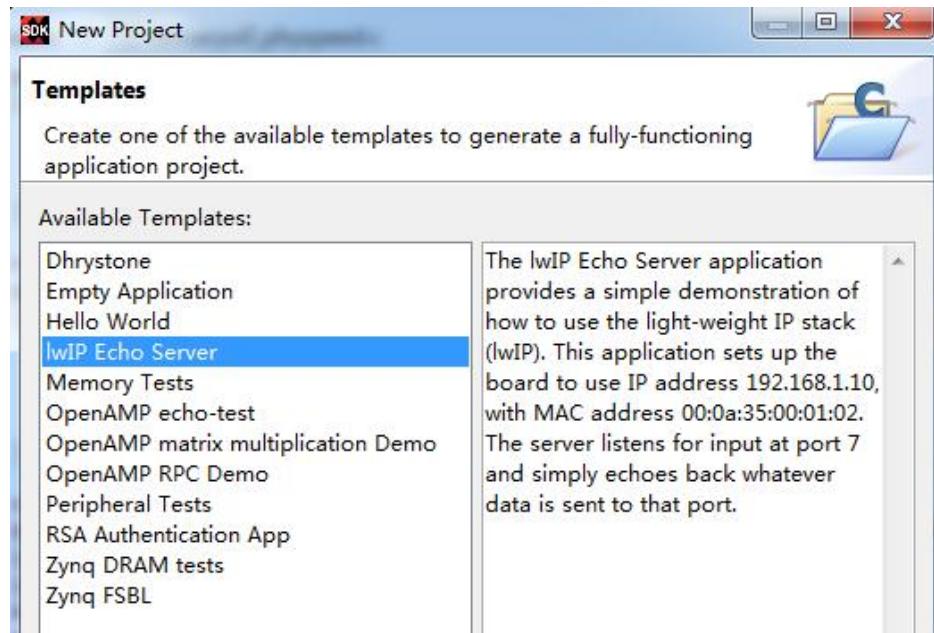
```
dmaconfig = XAxiDma_LookupConfig(xemac->topology_index);
```

这样是为了便于在设计中使用多个 AXI 1G/2.5G Ethernet Subsystem 和 AXI DMA 实现多个网口。

到此，LWIP 库的修改完成。

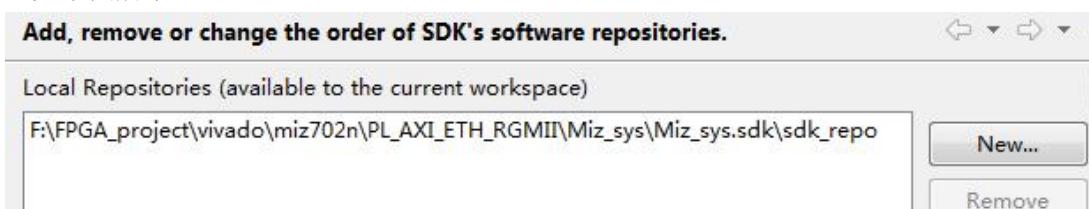
16.4.2 创建工程

本例程使用了 SDK 自带的 lwip echo server 例程来验证子卡电口和光口的功能，因此在创建工程时选择 LwIP Echo Server 模板，如下图所示。该例程基于 LWIP 库在 ARM 中建立一个 TCP Echo Server，IP 地址为 192.168.1.10，端口号为 7。



16.4.2.1 lwip 库设置

在 SDK 软件中选择修改后的 lwip 库的路径（需根据实际情况更改此路径，若不更改将产生错误），如下图所示。



修改完成后，打开 BSP 设置，此时 lwip echo server 例程使用的是 SDK 自带的 1.7 版本 LWIP 库，为了替换成我们所修改过的 1.74 版本，首先需要取消 lwip141 1.7 的勾选，如下图所示。



然后关闭 SDK，然后重启 SDK 打开该工程目录。然后打开 BSP 设置，此时 lwip 库便变成了所修改过的 1.74 版本，如下图所示。勾选 lwip141 v1.74 版本。

Supported Libraries		
Check the box next to the libraries you want included in your Board Support Package. You can use the search bar or the tree navigator on the left.		
Name	Version	Description
libmetal	1.1	Libmetal Library
<input checked="" type="checkbox"/> lwip141	1.74	lwIP TCP/IP Stack library: lwIP v1.4.1
openamp	1.2	OpenAmp Library
xilffs	3.5	Generic Fat File System Library
xilflash	4.2	Xilinx Flash library for Intel/AMD CFI compliant ...
xilifsl	5.7	Xilinx In-system and Serial Flash Library
xilmfs	2.2	Xilinx Memory File System
xilpm	2.0	Power Management API Library for ZynqMP
xilrsa	1.2	Xilinx RSA Library
xilskey	6.1	Xilinx Secure Key Library

将 use_axieth_on_zynq 设置为 1，表示此时不使用 PS 内部的 GEMAC 和 DMA，而是使用 PL 部分的 AXI 1G/2.5G Ethernet Subsystem 和 AXI DMA 来实现。将 use_emaclite_on_zynq 设为 0。如下图所示。

use_axieth_on_zynq	1	1	integer
use_emaclite_on_zynq	0	1	integer

在 lwip 库增加的选项中，当子卡的电口工作时，use_1000basex_on_88e1512 设置为 false；当工作于光口模式时，将其设为 true 即可。如下图所示。

use_1000basex_on_88e1512	true	false	boolean	Settings for operation mode

在 teamc_adapter_options 中，将 tcp_ip_tx_checksum_offload 和 tcp_ip_rx_checksum_offload 设为 true，这是因为在 AXI 1G/2.5G Ethernet Subsystem 的配置中启用了该功能。

n_rx_coalesce 和 n_tx_coalesce 表示 AXI DMA 在工作于链式 DMA 模式时，触发 1 次中断所需要完成传输的链表中 descriptor 的个数。将该值设大将会减少 AXI DMA 中断的触发次数，提高链式 DMA 的工作效率和 PS 的运行效率，可提高网口传输速率。

n_rx_descriptors 和 n_tx_descriptors 表示 AXI DMA 工作于链式 DMA 模式时，接收和发送链表中 descriptor 的个数，descriptor 的个数越大，链式 DMA 的工作效率将会提高，网口的传输速度也将会提高。最大可设为 256。

设置如下图所示。

temac_adapter_options	true	true	boolean
emac_number	0	0	integer
n_rx_coalesce	8	1	integer
n_rx_descriptors	64	64	integer
n_tx_coalesce	8	1	integer
n_tx_descriptors	64	64	integer
phy_link_speed	Autodetect (CONFIG_LINKSPEED_AUTODETECT)	CONFIG_LINKSPEED_AUTODETECT	enum
tcp_ip_rx_checksum_offload	true	false	boolean
tcp_ip_tx_checksum_offload	true	false	boolean
tcp_rx_checksum_offload	false	false	boolean
tcp_tx_checksum_offload	false	false	boolean
temac_use_jumbo_frames	false	false	boolean

其余选项的参数默认即可，不用修改。

16.4.2.2 example 代码修改

工程代码无需作任何修改。

16.5 程序测试

16.5.1 电口测试

16.5.1.1 lwip 库设置

电口模式下 lwip 的设置如下图所示。将 use_1000basex_on_88e1512 设为 false。

use_1000basex_on_88e1512	false	false	boolean
--------------------------	-------	-------	---------

16.5.1.2 网络测试

将千兆网线插入子卡的 RJ45 座中，与电脑连接，将电脑的 ip 地址设为 192.168.1.100，子网掩码为 255.255.255.0。然后给开发板上电，通过 SDK 将程序下载入开发板后，观察 SDK 串口打印信息，如下图所示。表示千兆网络连接正常。

Connected to: Serial (COM14, 115200, 0, 8)

-----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

auto-negotiated link speed: 1000

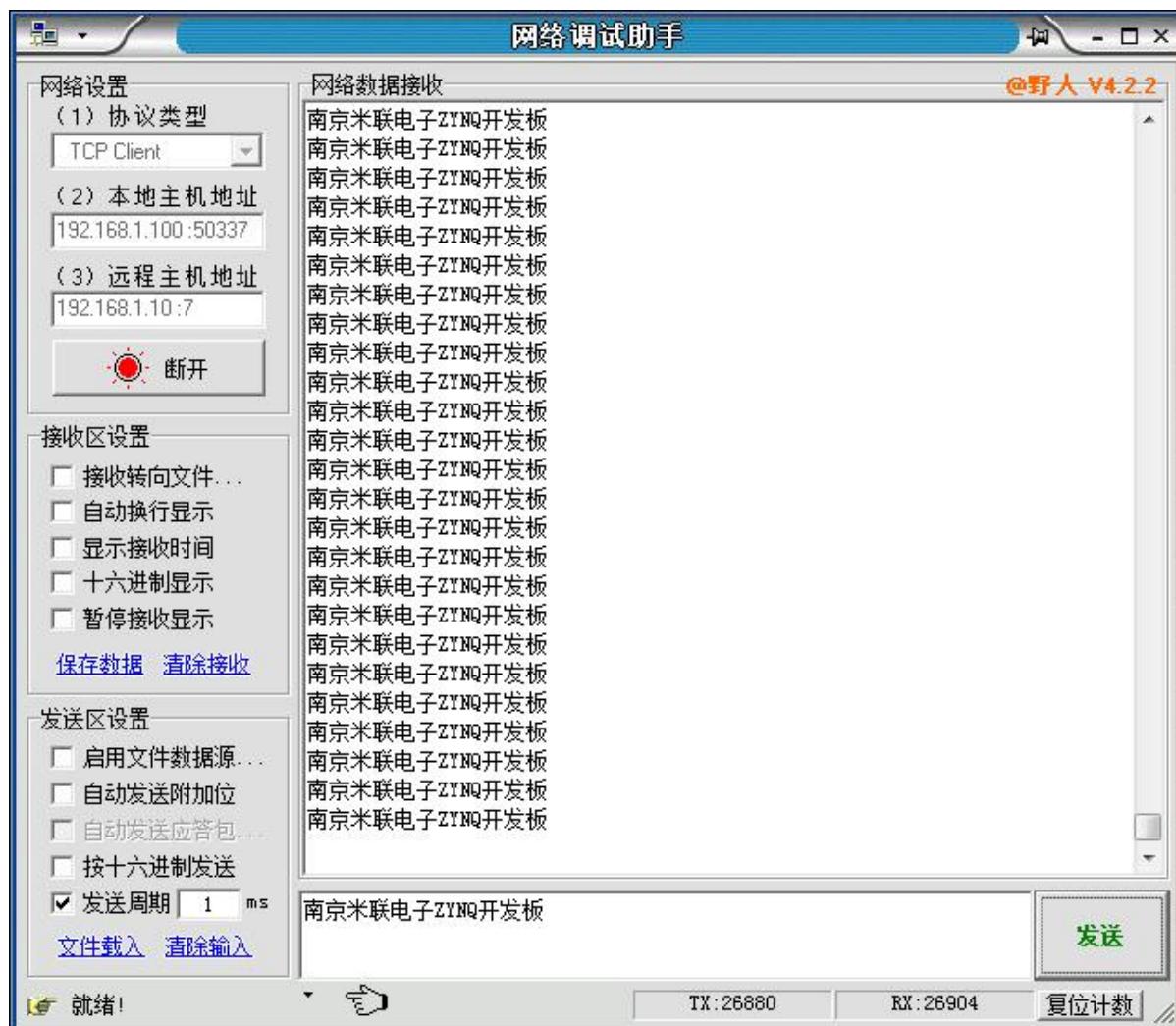
Board IP: 192.168.1.10

Netmask : 255.255.255.0

Gateway : 192.168.1.1

TCP echo server started @ port 7

打开网络调试助手，以 TCP Client 模式连接开发板的 IP 地址 192.168.1.10，端口号 7，输入文字发送，网络调试助手便可收到开发板返回相同的文字，如下图所示。



16.5.2 光口测试

16.5.2.1 lwip 库设置

光口模式下 lwip 的设置如下图所示，要将 use_1000basex_on_88e1512 设为 true。

use_1000basex_on_88e1512	true	false	boolean	Settings for operation mode
--------------------------	------	-------	---------	-----------------------------

16.5.2.2 网络测试

将 SFP 电口模块插入子卡的 SFP 屏蔽笼中，将千兆网线插入 SFP 电口模块，与电脑连接，将电脑的 ip 地址设为 192.168.1.100，子网掩码为 255.255.255.0。然后给开发板上电，通过 SDK 将程序下载入开发板后，观察 SDK 串口打印信息，如下图所示。

Connected to: Serial (COM14, 115200, 0, 8)

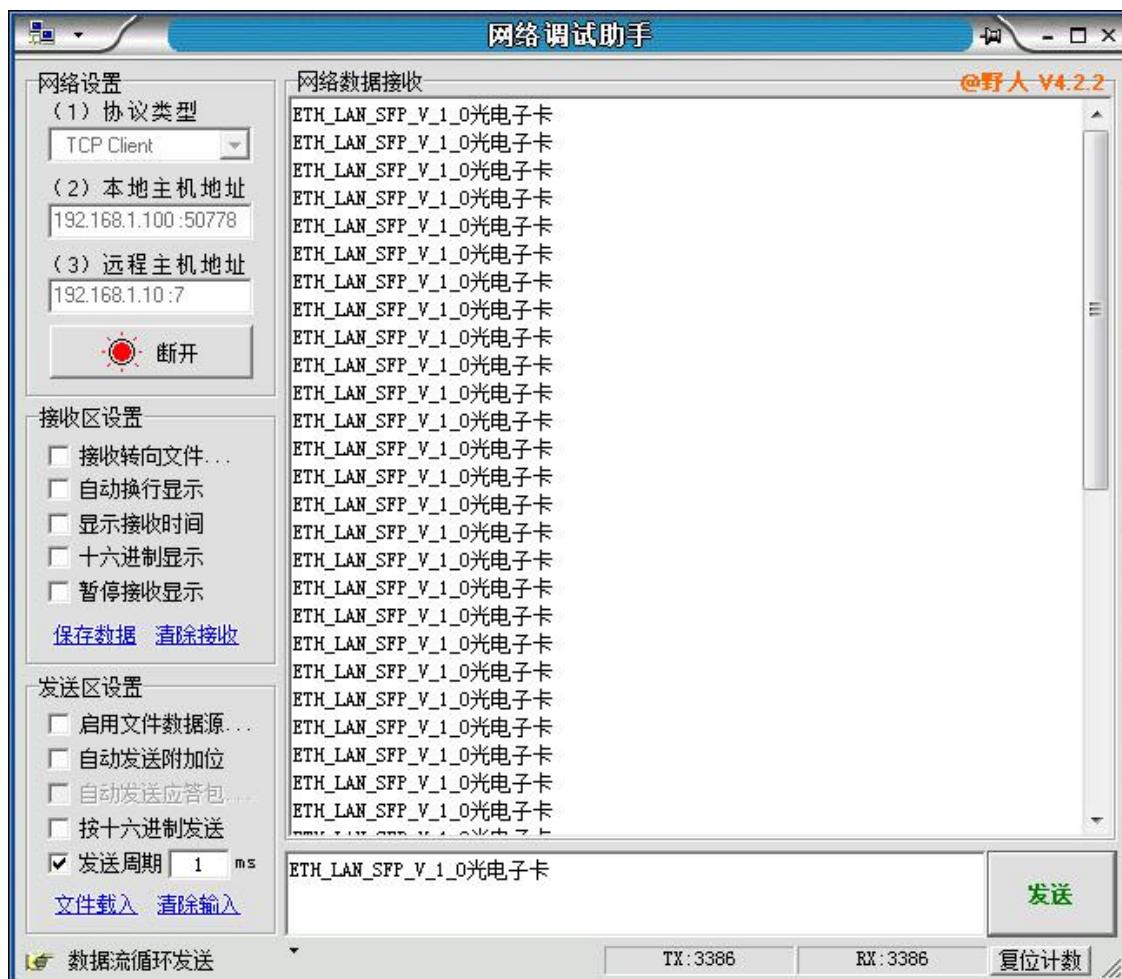
```
-----lwIP TCP echo server -----
TCP packets sent to port 6001 will be echoed back
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
auto-negotiated link speed: 1000
Board IP: 192.168.1.10

Netmask : 255.255.255.0

Gateway : 192.168.1.1

TCP echo server started @ port 7
```

同理，打开网络调试助手，以 TCP Client 模式连接开发板的 IP 地址 192.168.1.10，端口号 7，输入文字发送，网络调试助手便可收到开发板返回相同的文字，如下图所示。



S03_CH17 基于μGUI 的触摸屏 GUI 界面设计

17.1 概述

很多工程师都在 ZYNQ 上做过 LINUX 相关的应用和开发，在 ZYNQ 运行 LINUX 操作系统，可以实现非常细致的 GUI 图形界面。用户可以通过鼠标、键盘等外部设备与操作系统通过 GUI 界面进行人机交互，与平时使用电脑的体验相当。

但同时，也有很用户只涉及 ZYNQ 的裸机开发，那么在无操作系统支持的情况下，是否可以在裸机环境中构建一个 GUI 图形界面呢？

答案是肯定的。本例程通过利用开源 GUI 库 μ GUI，在 ZYNQ 裸机环境下创建一组简单的 GUI 界面，并通过外部液晶触摸屏实现与 ZYNQ 的人机交互。本例程所涉及的应用知识点如下：

- LCD 触摸屏的使用原理
- 移植开源μGUI 库，利用 API 函数搭建一组 GUI 图形界面
- 通过 VDMA、AXI-S Video Out、VTC 等 IP 实现 GUI 图形界面的显示
- 通过定时器中断实现 GUI 界面的周期性刷新
- 通过 PS 实现动态配置 MMCM/PLL
- 通过 PS 动态配置 AXI PWM 调节液晶屏亮度
- 通过 AXI GPIO 检查触摸屏的中断信号
- 通过 I2C 读取触摸屏的触摸信息
- 通过 PS 设置 Video Timing Controller 显示分辨率
- 设计 GUI 界面的动态变化机制，将触摸信息反馈至 GUI 界面后，GUI 产生动态变化，形成人机交互。

本例程基于 Vivado 2017.4 版本开发。

17.2 基本原理

本例程通过开源的 μ GUI v0.3 库设计了 1 个 GUI 界面，为该 GUI 界面设计了 5 个窗口，为每个窗口设计了标题、按键、文本、图片 logo 等元素，并给每个按键设定了相应功能，如窗口间切换、LED 灯控制、屏幕亮度调节等，并在 1 个窗口中设计了绘图功能。然后，将设计的 GUI 窗口通过触摸屏显示。用户通过按下触摸屏中所显示的窗口对应的按键位置，便可以实现该按键所设定的相应功能，从而实现人机交互。

17.3 LCD 触摸屏

LCD 触摸屏由 LCD 液晶屏和触摸屏两部分组成，其中 LCD 液晶屏用于画面显示，触摸屏用于实现触摸控制。本例程中所使用的是微雪公司的 7 寸 LCD 电容触摸屏，如下图所示。



其中，LCD 液晶屏采用了 24 位 RGB888 接口。电容触摸屏采用了 I2C 接口。LCD 触摸屏接口定义如下图所示。其中，1~35 为液晶屏接口，37~40 为触摸屏接口。

RGB接口定义

引脚号	标识	描述	类型	功能
1	BL_VDD	电源正	电源型	背光电源，一般接5V
2	BL_VDD			GND
3	GND			一般接3.3V
4	VDD			
5	R0	数据线	输入	红色调色板数据线
6	R1			
7	R2			
8	R3			
9	R4			
10	R5			
11	R6			
12	R7			
13	G0			
14	G1			
15	G2			
16	G3			
17	G4			
18	G5			
19	G6			
20	G7			

21	B0			
22	B1			
23	B2			
24	B3	数据线	输入	蓝色调色板数据线
25	B4			
26	B5			
27	B6			
28	B7			
29	GND	电源地	电源型	GND
30	DCLK	LCD时钟	输入	LCD时钟信号源
31	DISP	背光控制使能	输入	控制使能背光控制，一般接VDD
32	H SYNC	行同步	输入	水平同步信号输入
33	V SYNC	帧同步	输入	垂直同步信号输入
34	DE	控制模式选择	输入	DE=0:SYNC模式 DE=1:DE模式
35	PWM	背光明暗调节	输入	PWM调节背光信号线
36	GND	电源地	电源型	GND
37	I2C_SDA	I2C数据脚	输出/输入	I2C数据传输脚，读写数据
38	I2C_SCL	I2C时钟脚	输入	I2C时钟控制脚，控制时钟
39	CAP_WAKE	WAKEUP引脚	输出	用于唤醒外部TPU控制器
40	CAP_INT	中断引脚	输出	外部触摸中断控制引脚

17.3.1 液晶屏

液晶屏为 1024×600 分辨率，这个分辨率不常用，在网上很难找到统一的标准。在该液晶屏 HJ070NA-13A 的手册中可以找到此液晶屏在该分辨率下的时序要求，如下图所示。

Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Clock Frequency	fclk	40.8	51.2	67.2	MHz	Frame rate =60Hz
Horizontal display area	thd	1024			DCLK	
HS period time	th	1114	1344	1400	DCLK	
HS Blanking	thb	90	320	376	DCLK	
Vertical display area	tvd	600			H	
VS period time	tv	610	635	800	H	
VS Blanking	thb	10	35	200	H	

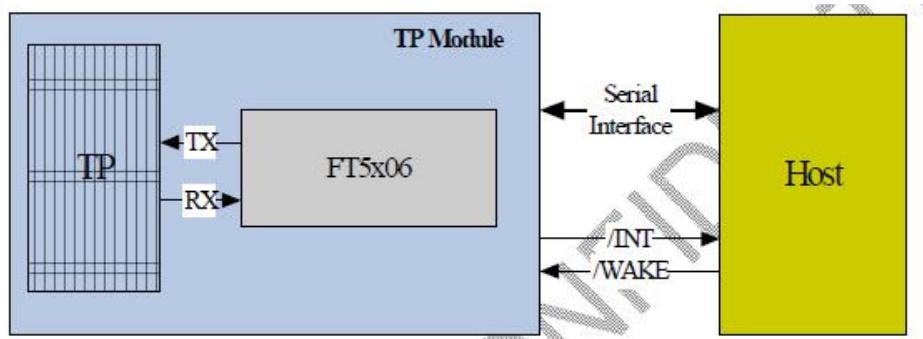
本例程采用了 51.2MHz 像素时钟所对应的参数设置。液晶屏的驱动方法与显示器类似，通过时钟，行、场同步信号，数据有效信号来完成，此处不作赘述。

要使液晶屏正常显示，背光源使能信号 DISP 要拉高，通过调节 PWM 的占空比可以改变背光源的亮度。该液晶屏的 PWM 为负极性，即低电平占空比越高，背光源越亮，

PWM 信号的频率范围为 100Hz~200KHz。

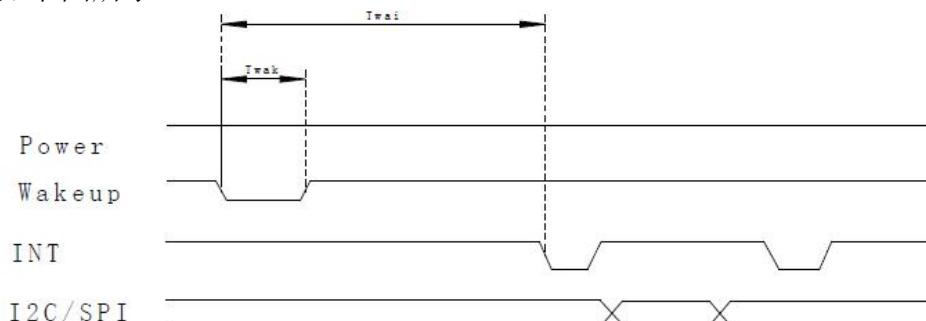
17.3.2 触摸屏

触摸屏为电容屏，采用了 FT5206 作为主控芯片，支持 5 点触控。触摸屏与 ZYNQ 的接口如下图所示。



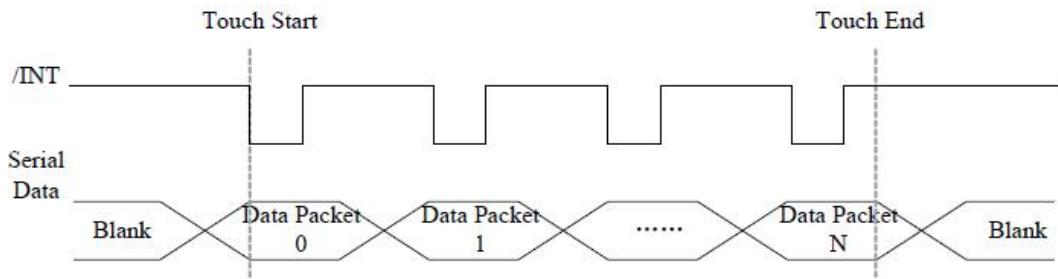
17.3.3 触摸屏唤醒

在触摸屏与 ZYNQ 的接口中，WAKE 信号为低电平有效，ZYNQ 通过拉低 WAKE 信号若干毫秒，再将其拉高来唤醒 FT5026 芯片，当触摸屏正常工作时 WAKE 信号应恒为高电平。如下图所示。



17.3.4 触摸中断

INT 信号也为低电平有效。当手指触摸电容屏时，INT 信号会以固定频率（默认为 60Hz）脉冲信号的形式输出，当手指离开电容屏，INT 信号重新恢复高电平，如下图所示。



17.3.5 触摸信息获取

每当 INT 变为低电平时, ZYNQ 就立即通过 I2C 接口读出 FT5206 芯片内部记录的触摸坐标信息, 完成一次触摸响应。

FT5206 芯片 I2C 接口作为 slave 从设备, 最高速率为 400KHz, I2C 地址为 0x38。这里补充说明一点, 关于 I2C 地址芯片 datasheet 中作了如下图所示的描述, 笔者在搜集到的关于 FT5206 的资料中均未能找到关于 I2CCON register 的说明。后来, 通过网络搜索发现使用过该系列芯片的记录中, 所提到的 I2C 地址均为 0x38, 笔者通过尝试得到了验证。

A[6:0]	Slave address
	A[6:4]: 3'b011
	A[3:0]: data bits are identical to those of I2CCON[7:4] register.

FT5206 芯片通过一系列寄存器记录与触摸相关的信息。相关寄存器如下图所示。

Address	Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Host Access
Op.00h	DEVIDE_MODE				Device Mode[2:0]					RW
Op.01h	GEST_ID				Gesture ID[7:0]					R
Op.02h	TD_STATUS					Number of touch points[3:0]				R
Op.03h	TOUCH1_XH		1 st Event Flag			1 st Touch X Position[11:8]				R
Op.04h	TOUCH1_XL				1 st Touch X Position[7:0]					R
Op.05h	TOUCH1_YH			1 st Touch ID[3:0]		1 st Touch Y Position[11:8]				R
Op.06h	TOUCH1_YL				1 st Touch Y Position[7:0]					R
Op.07h										
Op.08h										
Op.09h	TOUCH2_XH		2 nd Event			2 nd -Touch				R

		Flag		X Position[11:8]	
Op,0Ah	TOUCH2_XL	2 nd touch X Position[7:0]		R	
Op,0Bh	TOUCH2_YH	2 nd Touch ID[3:0]	2 nd Touch Y Position[11:8]		R
Op,0Ch	TOUCH2_YL	2 nd Touch Y Position[7:0]		R	
Op,0Dh				R	
Op,0Eh				R	
Op,0Fh	TOUCH3_XH	3 rd Event Flag		3 rd Touch X Position[11:8]	R
Op,10h	TOUCH3_XL	3 rd Touch X Position[7:0]		R	
Op,11h	TOUCH3_YH	3 rd Touch ID[3:0]	3 rd Touch Y Position[11:8]		R
Op,12h	TOUCH3_YL	3 rd Touch Y Position[7:0]		R	
Op,13h				R	
Op,14h				R	
Op,15h	TOUCH4_XH	4 th Event Flag		4 th Touch X Position[11:8]	R
Op,16h	TOUCH4_XL	4 th Touch X Position[7:0]		R	
Op,17h	TOUCH4_YH	4 th Touch ID[3:0]	4 th Touch Y Position[11:8]		R
Op,18h	TOUCH4_YL	4 th Touch Y Position[7:0]		R	
Op,19h				R	
Op,1Ah				R	
Op,1Bh	TOUCH5_XH	5 th Event Flag		5 th Touch X Position[11:8]	R
Op,1Ch	TOUCH5_XL	5 th Touch X Position[7:0]		R	
Op,1Dh	TOUCH5_YH	5 th Touch ID[3:0]	5 th Touch Y Position[11:8]		R
Op,1Eh	TOUCH5_YL	5 th Touch Y Position[7:0]		R	

其中，TD_STATUS 寄存器记录了触摸点数，如下图所示。

2.1.3 TD_STATUS

This register is the Touch Data status register.

Address	Bit Address	Register Name	Description
Op,02h	3:0	Number of touch points[3:0]	How many points detected. 1-5 is valid.
	7:4		

记录触摸坐标、触摸动作的寄存器如下图所示。

2.1.4 TOUCHn_XH (n:1-5)

This register describes MSB of the X coordinate of the nth touch point and the corresponding event flag.

Address	Bit Address	Register Name	Description
Op,03h ~ Op,39h	7:6	Event Flag	00b: Put Down 01b: Put Up 10b: Contact 11b: Reserved
	5:4		Reserved
	3:0	Touch X Position [11:8]	MSB of Touch X Position in pixels

2.1.5 TOUCHn_XL (n:1-5)

This register describes LSB of the X coordinate of the nth touch point.

Address	Bit Address	Register Name	Description
Op,04h ~ Op,3Ah	7:0	Touch X Position [7:0]	LSB of the Touch X Position in pixels

2.1.6 TOUCHn_YH (n:1-5)

This register describes MSB of the Y coordinate of the nth touch point and corresponding touch ID.

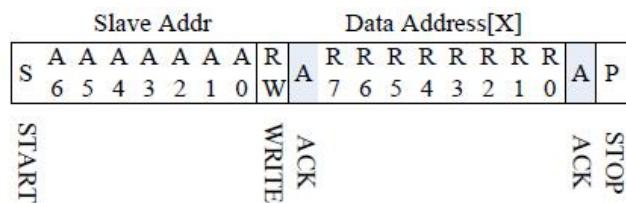
Address	Bit Address	Register Name	Description
Op.05h	7:4	Touch ID[3:0]	Touch ID of Touch Point
~ Op.3Bh	3:0	Touch X Position [11:8]	MSB of Touch Y Position in pixels

2.1.7 TOUCHn YL (n:1-5)

This register describes LSB of the Y coordinate of the nth touch point.

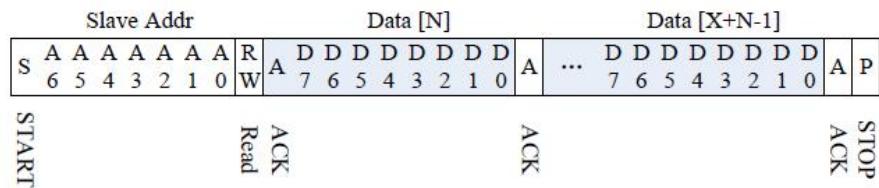
Address	Bit Address	Register Name	Description
Op.06h ~ Op.3Ch	7:0	Touch X Position [7:0]	LSB of The Touch Y Position in pixels

ZYNQ 通过 I2C 读取这些寄存器值分为两个步骤。首先，发送需要连续读取的起始寄存器地址，如下图所示。



然后，连续读取若干个地址连续的寄存器值，如下图所示。

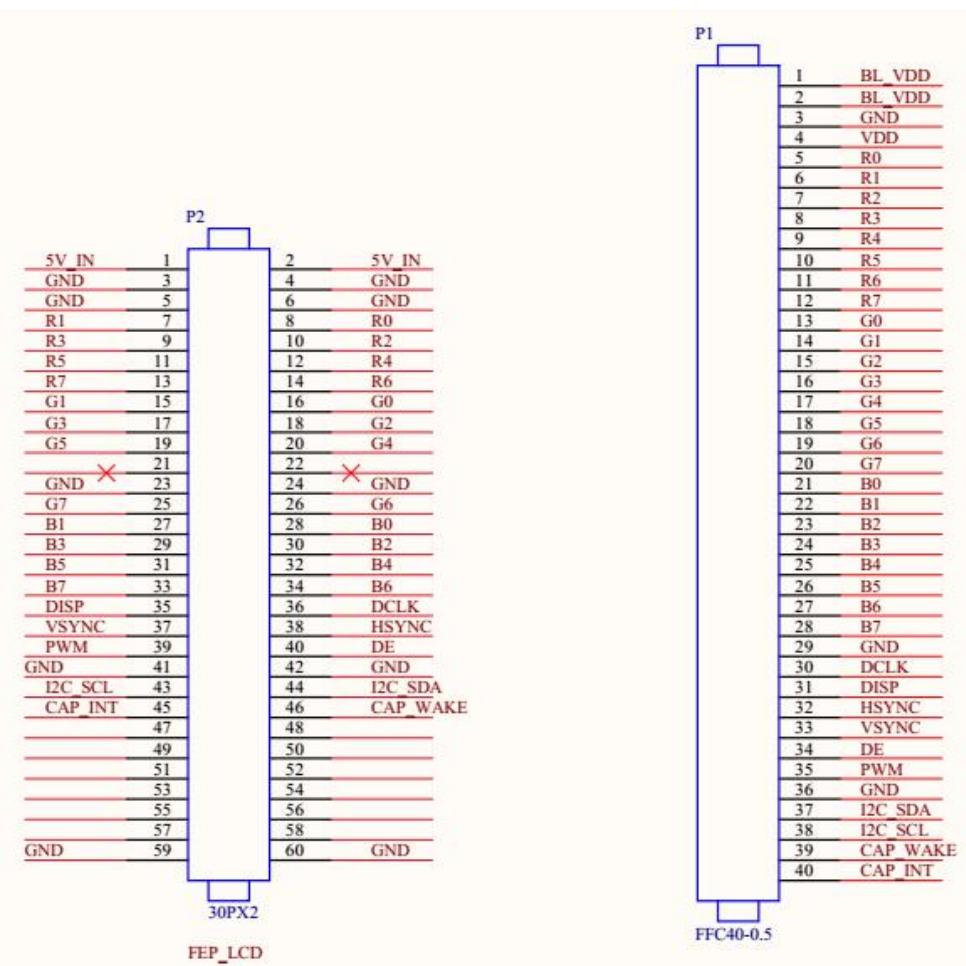
Read X bytes from I²C Slave



17.3.6 触摸屏与 Miz ZYNQ 开发板的接口

由于本例程所使用的 LCD 触摸屏接口为 FPC，而开发板并不具有 FPC 接口。因此，需要设计一个转接板将触摸屏与开发板的 40pin NEP 接口连接。需要注意的是，在该触摸屏内部，I²C 信号线均未接上拉电阻。因此，在转接板中需要将两个 I²C 信号各通过 1 个几 KΩ 的电阻上拉接至 3.3V，否则 I²C 接口将无法正常使用。

NEP 接口定义如下图所示。



17.4 μ GUI 概述

μ GUI 是一个开源的 GUI 图形界面设计库，包含了窗口（window），按键（button），文本框（textbox），图片（image）等 4 种元素。用户可以根据它们所对应的 API 函数自由发挥，对这些元素的数量、颜色、大小、位置、功能等参数进行定义，实现简易的 GUI 界面设计。

μ GUI 具有一个特色，提供了专门的 API 函数来支持外部的二维坐标（X, Y）输入设备，例如，触摸屏。用户所创建的 GUI 界面可以通过这些二维坐标获取外界物体输入的交互信息（比如触摸具有某个特定功能的按键），以此为基础实现人机交互。

该库一共包含 ugui.c 和 ugui.h 两个文件。关于 μ GUI 库更为详细的介绍以及相应 API 函数的使用可参考其使用手册。读者可以访问其所在网站 www.embeddedlightning.com，可以下载源文件、使用手册以及 example project。

17.4.1 μ GUI 库移植

由于 μ GUI 是一个跨平台（DSP、ARM、MCU、单片机等）的 C 语言库，在将 μ GUI 的 ugui.c 和 ugui.h 文件移植到 ZYNQ 的 ARM 中，并通过 SDK 进行程序设计和编译之前，需要完成 2 个重要工作：

17.4.1.1 添加单像素点像素值设置函数

1 个 GUI 界面其实就是 1 幅图像，1 幅图像由很多个像素点组成。要设计 1 个 GUI 界面，就需要设置其中每个像素点的像素值。简单的说，就是要给 GUI 界面对应图像所在存储区域的各个地址赋值。但针对不同的平台，不同的应用，GUI 界面的存储方式会存在不同，因此像素值设置的方式会有区别。出于跨平台的目的， μ GUI 给出了这个函数的原型声明，提供了开放接口，由用户自行设计这个函数来完成 GUI 界面各像素点值的设置， μ GUI 中所有与界面设计相关的 API 函数都将调用这个函数。本例程在 gui_window.c 中所设计的函数如下：

```
voidPixelSet(UG_S16 x, UG_S16 y, UG_COLOR c)
{
    u32iPixelAddr;
    iPixelAddr = y * GUI_WIDTH + x;
    BufferPtr[0][iPixelAddr] = c;
}
```

其中 x, y 对应像素点的二维坐标（X,Y），c 为该像素点所需设置的像素值，位宽 32bit。由于本例程中 GUI 界面位于 DDR 中连续的内存区域，通过 GUI 界面的指针 BufferPtr [0] 以及 x, y 便可计算出像素点的地址。

17.4.1.2 调整各种变量类型定义

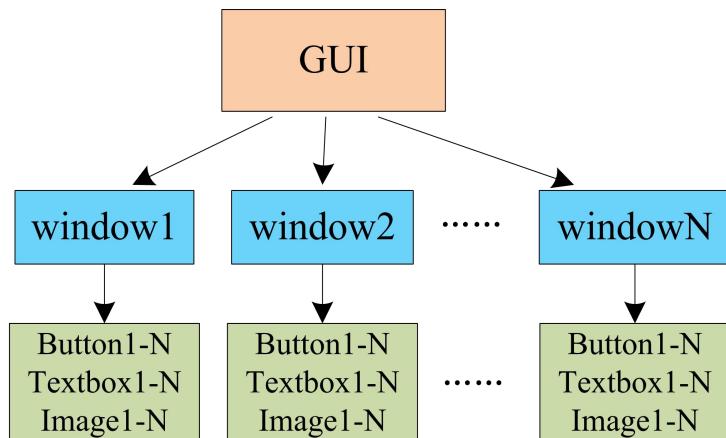
在不同的嵌入式平台中，对于各种变量类型的定义可能会存在区别，例如，int型变量在不同的平台中可能是64位、32位或者16位的。因此，需要在ugui.h中调整各变量类型的定义。ugui.h中默认的变量定义如下：

```
typedef uint8_t U8;
typedef int8_t S8;
typedef uint16_t U16;
typedef int16_t S16;
typedef uint32_t U32;
typedef int32_t S32;
```

上述变量定义与ZYNQ平台编译器一致，因此无需修改，只需在ugui.h文件中将#include "system.h"改为#include "stdint.h"即可。

17.4.1.3 GUI 窗口

窗口是μGUI中GUI界面必需的基本组成，不可缺少，没有窗口就无法创建GUI界面。而文本框、按键、图片等对象则是可选项，在GUI界面中可有可无。μGUI中的窗口包含了文本框、按键、图片3种基本对象，每个窗口都可以拥有若干个多种对象。简而言之，1个窗口可以包含若干个对象，而1个GUI界面可以包含若干个窗口，它们之间的关系如下图所示。



17.4.1.4 GUI 界面刷新

作为GUI界面，应该具有动态变化的特性。例如，不同窗口间切换、文本框内容变化、按键位置调整等。μGUI提供了一个函数UG_Update()来实现整个GUI界面的刷新。当用户通过μGUI库中某些API函数改变当前窗口中某些对象的特性后（例如，文字、颜色、大小），就必须尽快调用UG_Update()来确保这些更改被刷新到GUI界面上。若不调用UG_Update()，则GUI界面将永远不会发生任何改变。

因此，在使用μGUI库进行GUI界面设计时，为了保证其动态变化的特性，用户必须要周期性的调用UG_Update()函数，调用频率的高低可根据实际要求来确定。

17.4.1.5 人机交互

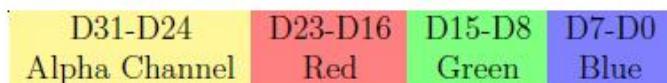
GUI 界面的动态变化特性，其中很大一部分都源自于人机交互，上面提到了 μ GUI 库支持外部的二维坐标 (X, Y) 输入设备，由外部输入的这些二维坐标就是 GUI 界面所需要的人机交互信息。例如，当设备与触摸屏连接，用户触碰了触摸屏的某个区域，触摸屏将这个区域的中心二维坐标反馈给设备，设备再将坐标输入 GUI 界面。

那如何将人机交互与 GUI 界面的动态特性相关联呢？ μ GUI 库提供了 UG_TouchUpdate() 函数，用于向 GUI 界面输入当前窗口中外部输入的二维坐标信息。通过该函数，GUI 界面可判断出该坐标对应当前窗口中具体哪个对象，例如，某个按键。那当知道了某个对象被触摸后，是否应该作出，或者具体如何作出动态变化来响应这个输入坐标呢？这完全由用户自己来进行定义。 μ GUI 库为每个窗口提供了窗口回调函数来完成这个任务，并给出了回调函数的原型声明（详情参考 μ GUI 使用手册），函数的具体内容完全由用户自己设计。

窗口回调函数在 UG_Update() 中被调用，回调函数获取输入二维坐标对应的对象信息，用户可以据此自由发挥，设计出相应的动态变化效果。因此，GUI 的动态变化最终将由 GUI 界面刷新函数 UG_Update() 来实现。这也说明了为何必须周期性的调用 UG_Update() 来确保 GUI 界面的动态变化特性。

17.4.2 颜色空间

在 μ GUI 库中，GUI 界面里每个像素点的像素值都是 32 位的无符号整型变量，采用 ARGB888 颜色空间，如下图所示。然而，最新的版本 μ GUI v0.3 只使用了其中的 RGB888 部分。因此，对于用户来说，GUI 界面就是一幅 24 位的 RGB888 彩色图像，高 8 位完全忽略。



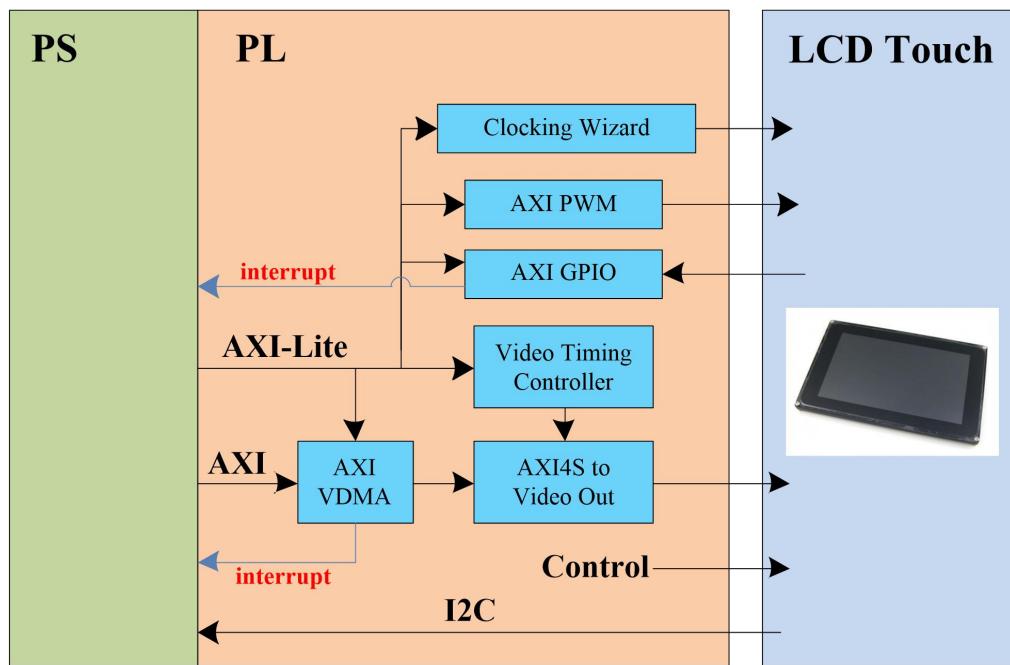
在 μ GUI 中，定义了上百种 RGB888 的颜色种类，用户可根据自己的喜好为 GUI 界面中的窗口、按键、文本等对象选择相应的颜色进行设计。

17.4.3 字体大小

μ GUI 库中定义了十几种大小不同的字体，用于 GUI 界面中窗口、文本框、按键等对象的文本显示。

17.5 PL 逻辑框架

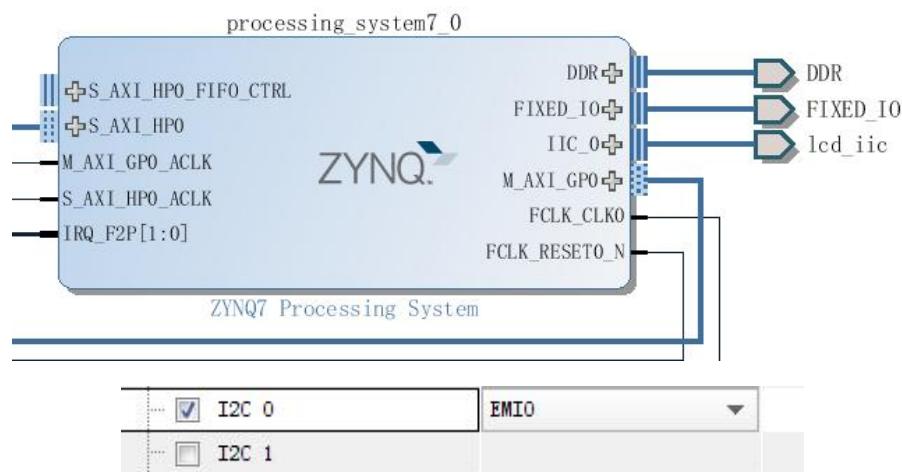
本例程的 PL 部分设计原理如下图所示。



17.5.1 PS 设置

PS 部分的设置下图所示。

- 使能 M_GP0、S_HPO，PL 到 PS 的中断接口。
- 使能 I2C0 接口，并将其通过 EMIO 方式引出。I2C0 接口与触摸屏连接，用于读取触摸屏的触摸坐标信息。
- 使能 FCLK_CLK0，时钟频率设置为 100MHz。



17.5.2 GUI 界面显示

本例程中 GUI 界面的显示通过 AXI VDMA、AXI4-S to Video Out、Video Timing

Controller 三个 IP 核完成。显示分辨率为 1024*600@60Hz。

17.5.2.1 AXI VDMA 设置

● AXI VDMA 与 AXI DMA 使用对比

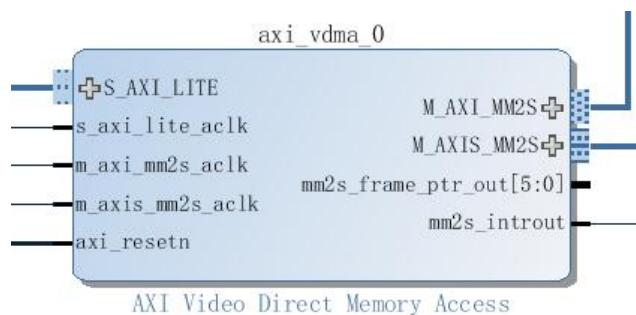
在米联第三季教程中有关于通过 AXI DMA 实现摄像头图像显示的例程。然而在本例程中，AXI DMA 并不适用。原因如下：

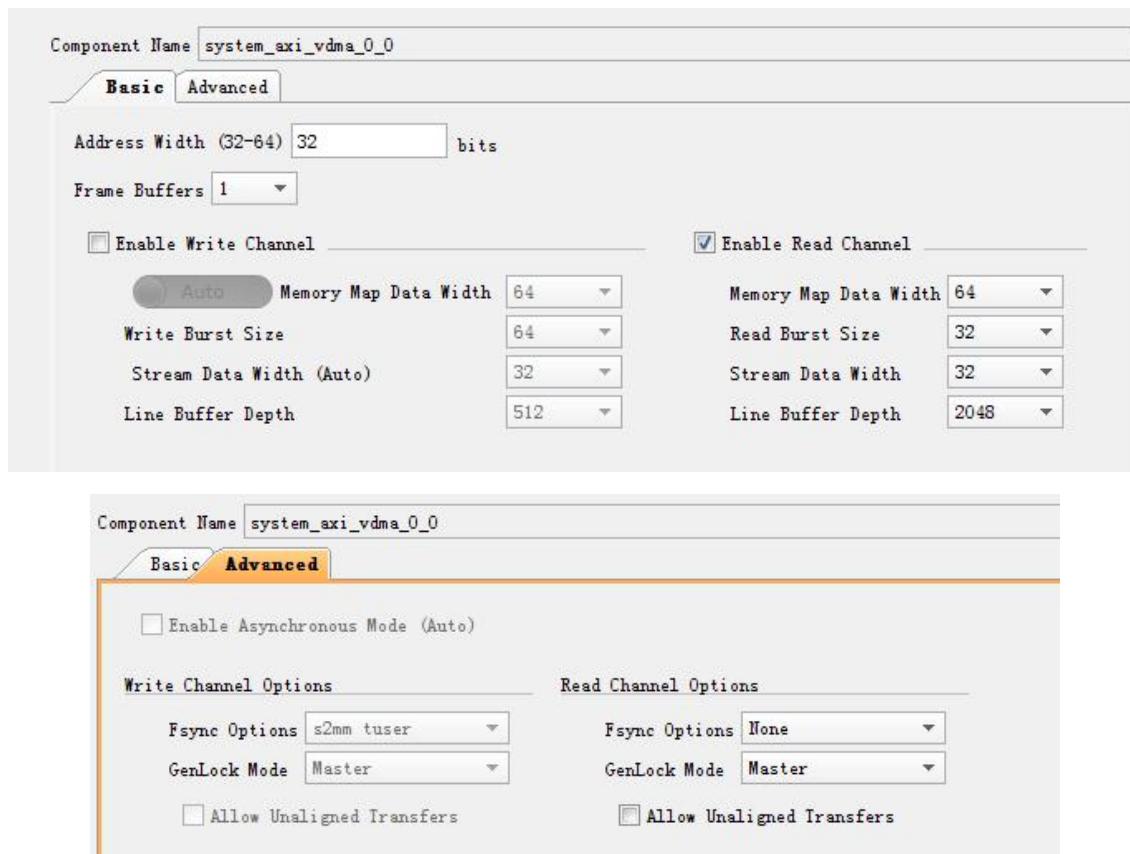
当使用 AXI DMA 时，PS 每次向 PL 发送 GUI 界面图像都需要通过调用函数配置 PL 的 AXI DMA 才能完成。在米联第三季中关于 AXI DMA 图像显示的例程中，都是通过在 AXI DMA 发送完成中断函数中发起下一次图像传输的方式来实现连续的图像发送。同时，在本例程中，GUI 界面的刷新过程是在定时器中断函数中完成，当触摸屏以 60Hz 显示时，1 幅图像的显示时间为 17.67ms。若 GUI 界面刷新的时间超过触摸屏中 1 幅图像显示的时间时，AXI DMA 图像发送完成中断将不能被立即响应执行，会导致图像显示发生中断。当 GUI 界面刷新完成 CPU 退出定时器中断后，AXI DMA 图像发送才能被继续执行，重新恢复 GUI 界面的显示，这样在每次 GUI 界面刷新的时候就会产生“跳屏”的现象。总而言之，定时器中断函数的执行时间会影响 AXIDMA 发送中断函数的执行。笔者已使用 AXI DMA 验证了这个现象。

而 VDMA 具有 free run 的特点，GUI 界面图像的发送可由 PL 的 VDMA 独自完成，无需 PS 参与，因此，定时器中断中 GUI 界面刷新与图像显示不会产生任何冲突，可以保证 GUI 界面显示的连续性。因此，在本例中使用 VDMA 更合适。

● AXI VDMA 设置

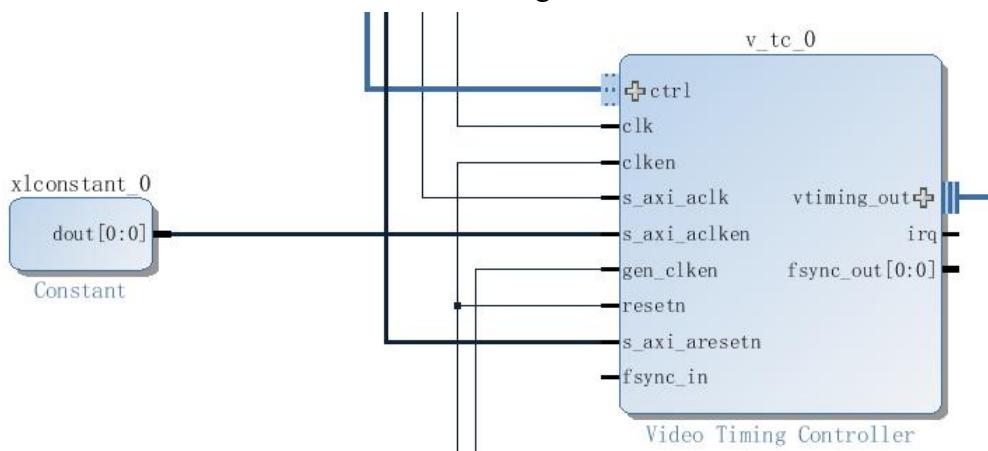
在本例程中，只存在 PS 的 DDR 向 PL 的图像传输，即 Memory Map to Stream (MM2S) 方向。因此，只需使能读通道，可关闭写通道来节省逻辑资源。由于 GUI 界面只对应 DDR 中的同一幅图像，只需 1 个图像缓存，所以将 frame buffer 设置为 1 即可。另外，将中断输出与 PS 连接。AXI VDMA 的设置如下图所示。

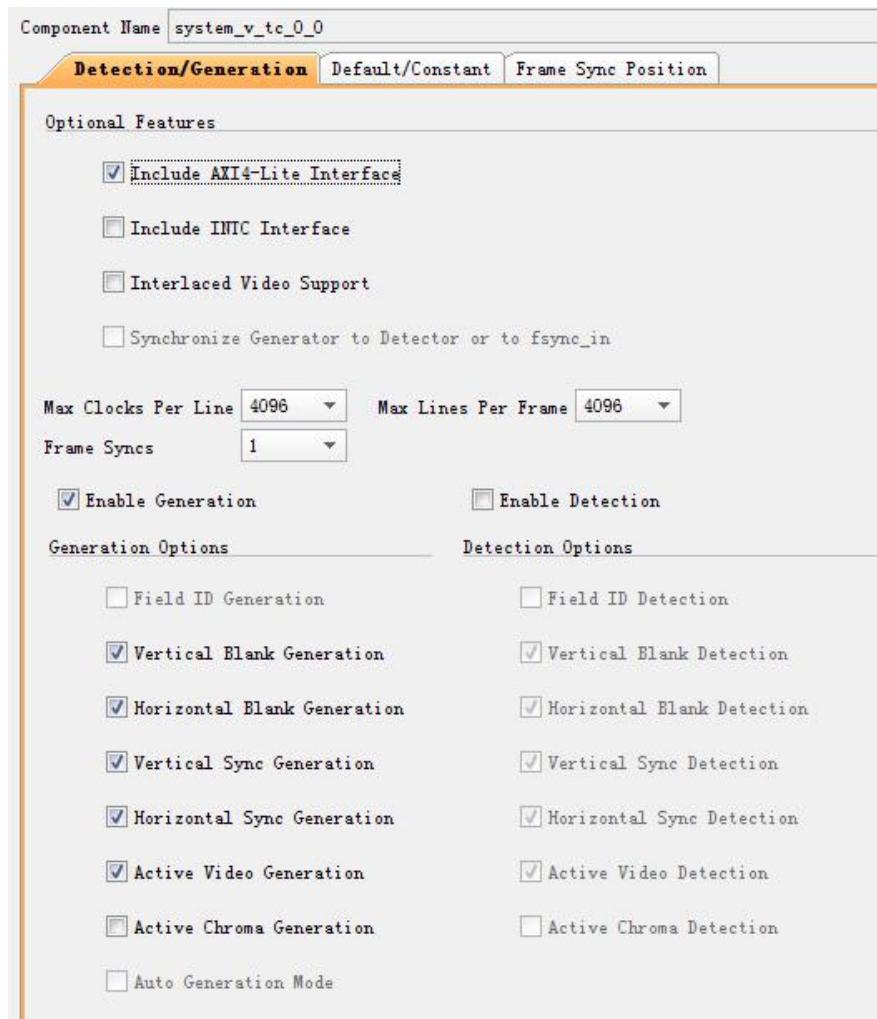




17.5.2.2 Video Timing Controller

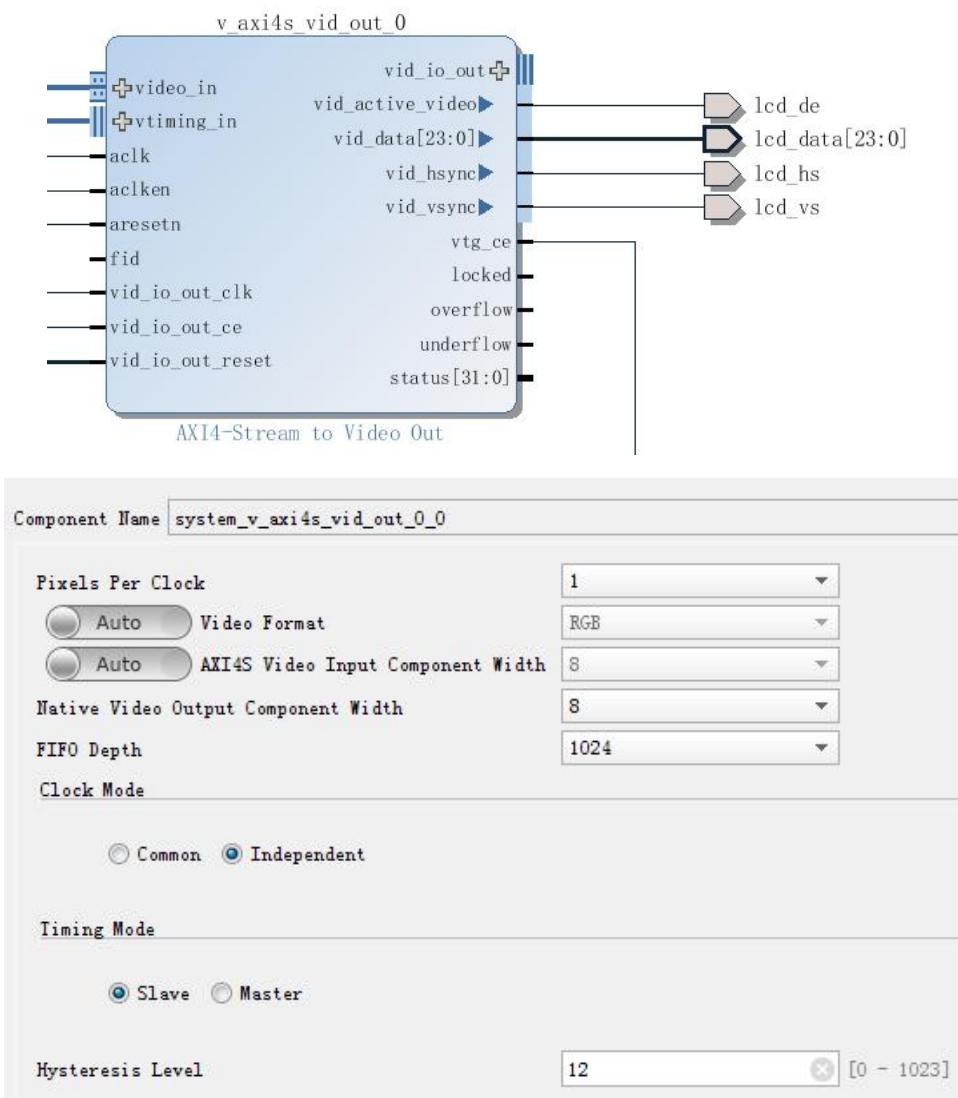
为了方便 PS 动态配置 Video Timing Controller 的显示分辨率，使能 AXI-lite 接口，其默认的分辨率可不用进行设置。Video Timing Controller 的设置如下图所示。





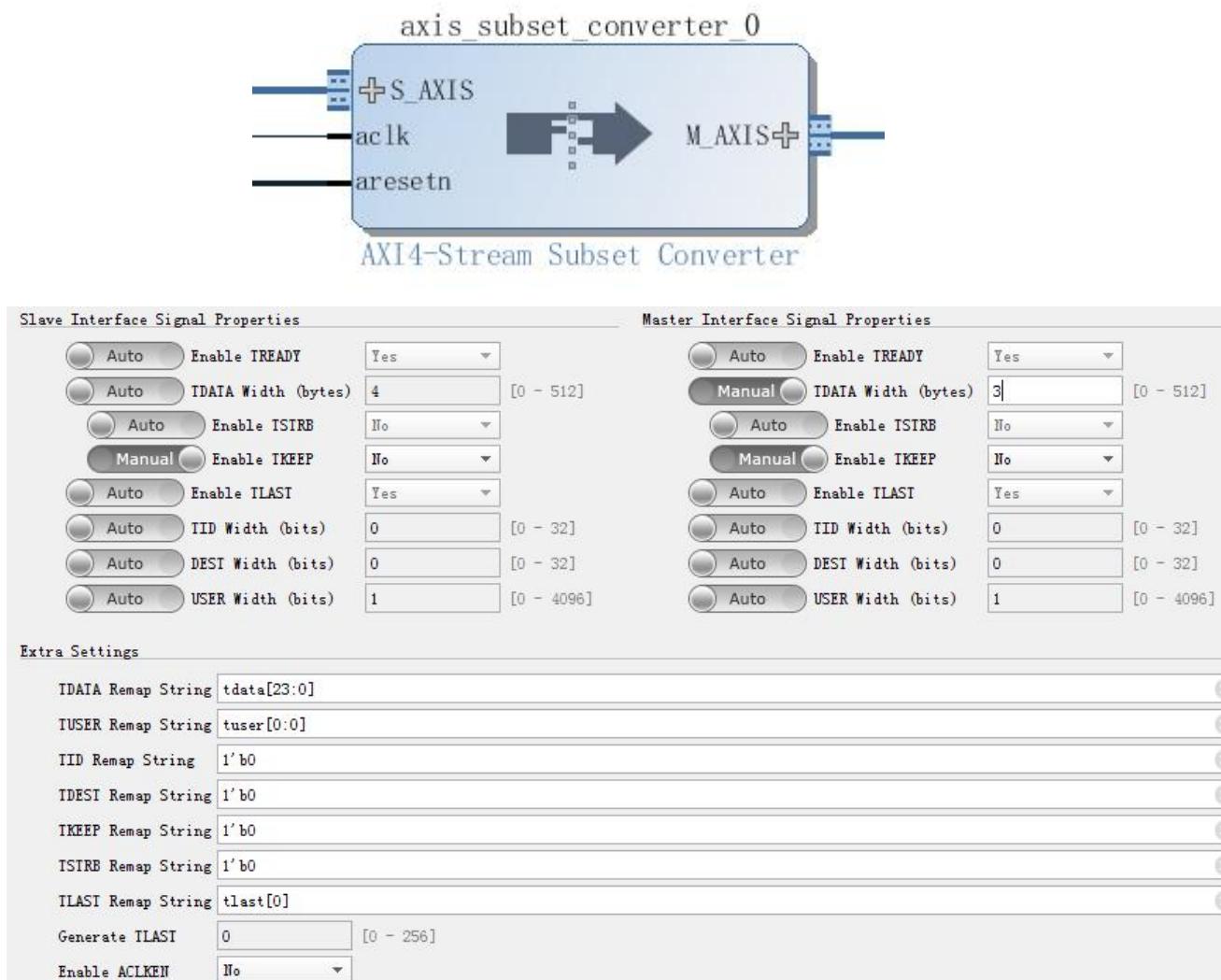
17.5.2.3 AXI4-Sream to Video Out

本例程中，输出至触摸屏的 GUI 图像为 RGB888 格式，AXI4- Streamto Video Out 设置如下图所示。需要引出 RGB 数据 data、行同步信号 hs、场同步信号 vs 以及数据有效信号 de 与 LCD 触摸屏连接。



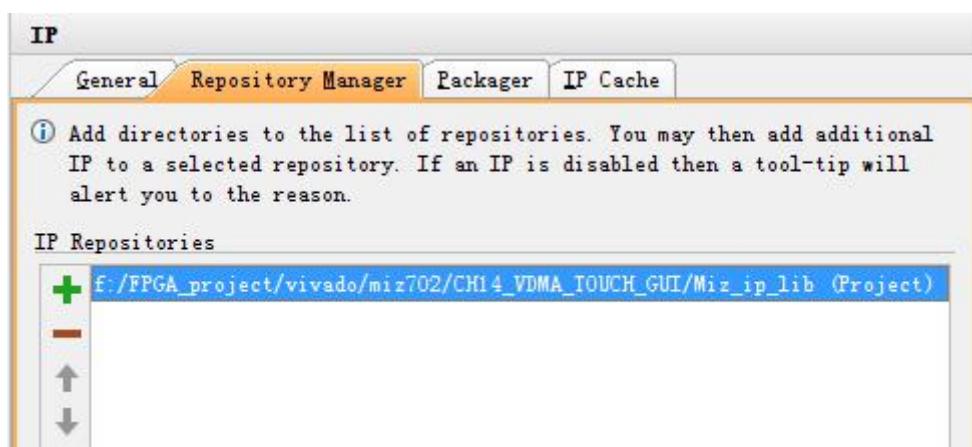
17.5.2.4 AXI-Stream Subset Converter

AXI VDMA 与 AXI4-Stream to Video Out 之间通过 AXI Stream 接口连接, 由于 AXI VDMA 的 M_AXIS_MM2S 接口的数据为 32 位, 且含有 4 位的 tkeep 信号。而 AXI4-Stream to Video Out 的 `video_in` 接口的数据为 24 位, 且无 tkeep 信号。为了更好的将两个信号之间存在差异的 Stream 接口连接, 通过 AXI-Stream Subset Converter 完成差异信号的重映射。设置如下图所示。

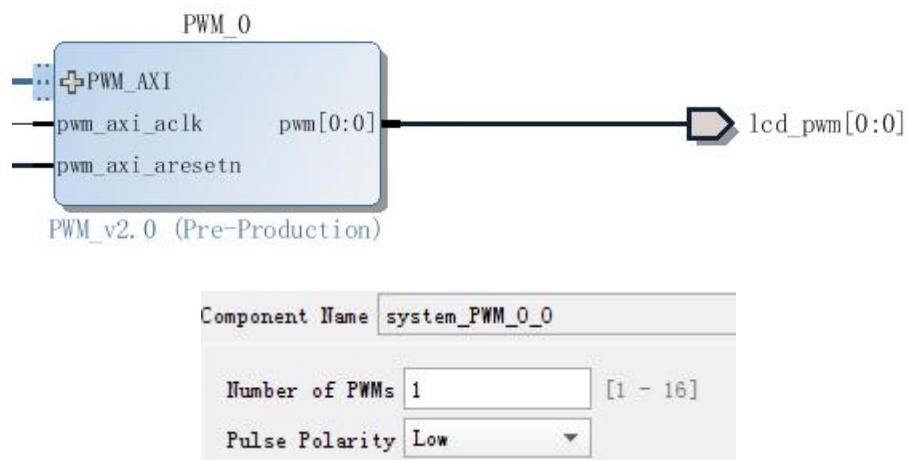


17.5.3 AXI PWM

本例程中，PL 部分通过使用 digilent 公司 AXI PWM 的 IP 核来实现 PWM 信号的输出，用于驱动触摸屏的背光源。该 ip 位于 Miz_ip_lib 文件夹下，IP 的路径设置如下图所示。



很多触摸屏的背光源都是由 PWM 信号驱动的，通过改变 PWM 信号的占空比可以调节触摸屏的亮度。AXI PWM 通过 AXI 总线与 PS 连接，PS 通过 AXI4-Lite 对其进行配置和控制，并可动态调节输出 PWM 信号的占空比。AXI PWM 的设置如下图所示，只需输出 1 路 PWM 信号，由于使用的触摸屏所需的 PWM 信号为负极性，即低电平占空比越大，屏幕越亮，因此将极性设为负。输出的 PWM 信号与触摸屏连接。

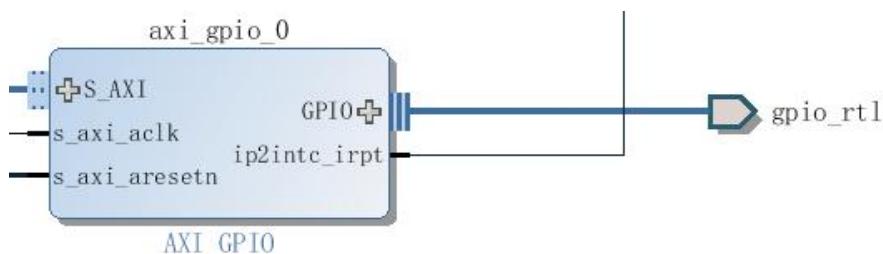


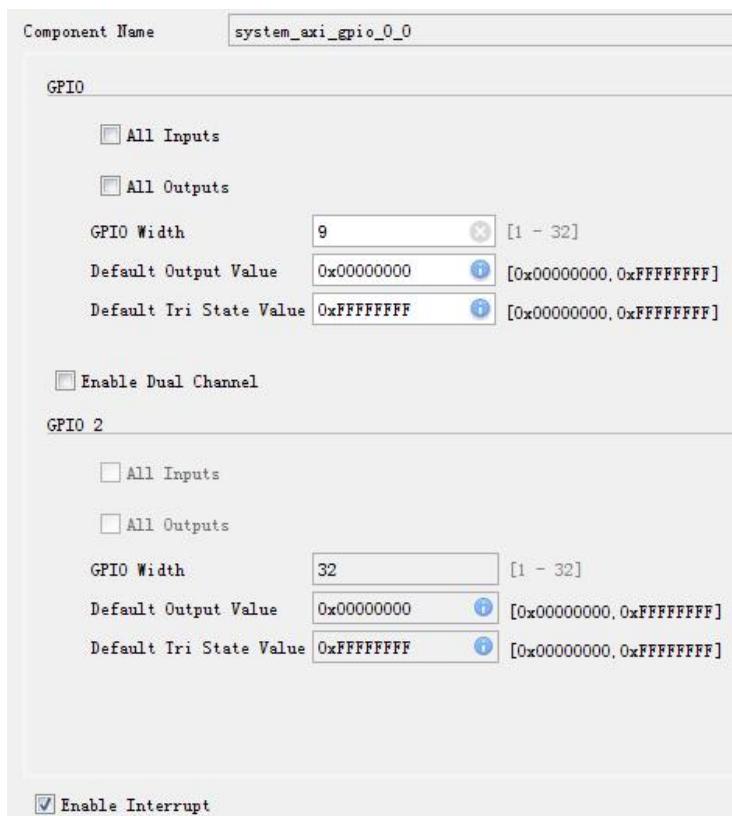
17.5.4 AXI GPIO

本例程中，PL 部分通过 AXI GPIO 实现 1 个触摸屏触摸中断信号输入，2 个按键信号输入，5 个 LED 控制信号输出，1 个液晶屏背光源使能信号输出，共计 9 个 GPIO 端口，定义如下。

- GPIO[0]，触摸屏中断信号输入
- GPIO[1]，按键信号输入，mi702 对应 SW3，miz701n 对应 SW0
- GPIO[2]，按键信号输入，mi702 对应 SW4，miz701n 对应 SW1
- GPIO[3]~GPIO[7]，LED1~LED5 控制信号输出，miz702 对应 LD1~LD5。由于 miz701n 的 LD5 与 PS 的 MIO51 脚连接，需要由 PS 控制，为了确保不同开发板 C 程序的一致性，只使用 miz701n 的 LD1~LD4，LD5 不作控制。
- GPIO[8]，液晶屏背光源控制信号输出

每当输入的触摸中断信号或按键信号发生一次改变，AXI GPIO 则会向 PS 触发一次中断。AXI GPIO 的设置如下图所示。使能中断接口，将其与 PS 连接。

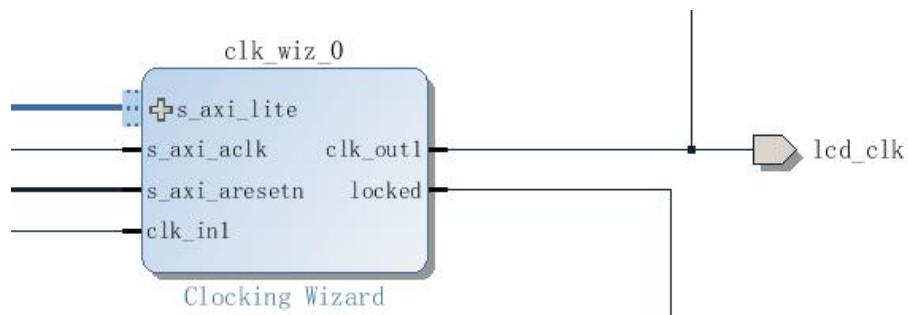


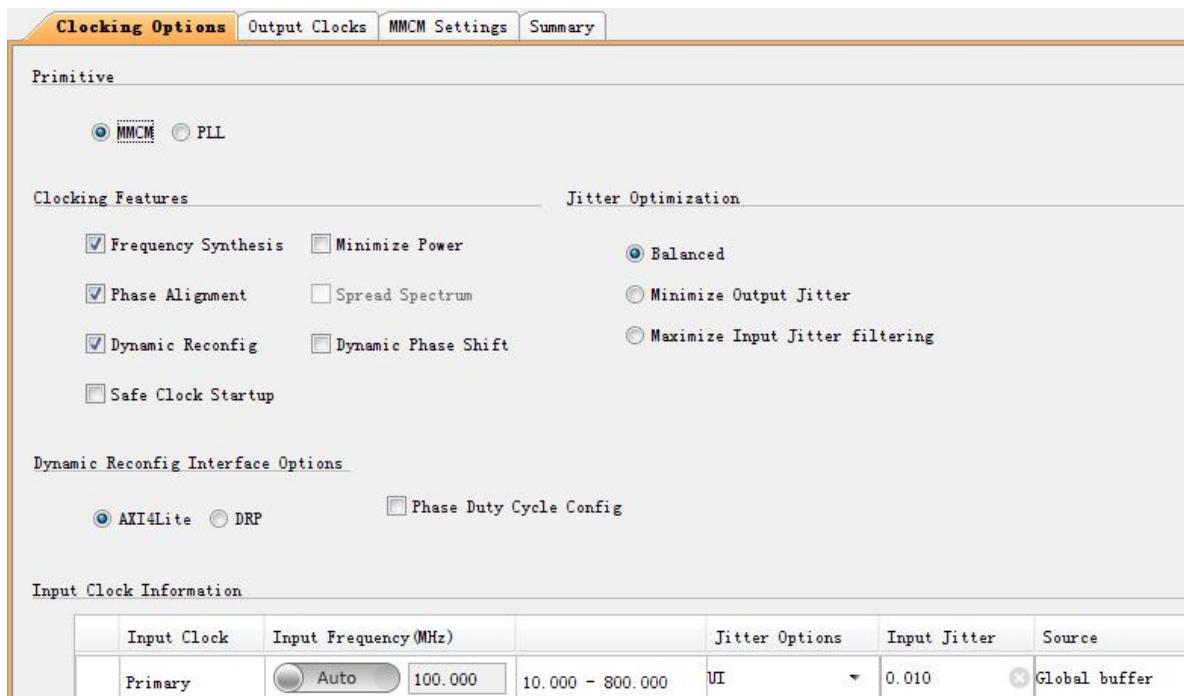


17.5.5 Clocking Wizard

本例程中，Clocking Wizard 用于提供 LCD 触摸屏进行 GUI 显示的参考时钟，该时钟被 Video Timing Controller 以及 AXI4-Sreamto Video Out 两个 IP 核使用，同时引出至顶层模块。将 Clocking Wizard 设置为 MMCM(因为 MMCM 内部倍频系数和 CLKOUT0 分频系数可以为小数，比 PLL 更易获得所需的频率)，并使能 AXI-lite 动态重配置接口，将动态重配置接口通过 AXI 总线与 PS 连接，方便 PS 随时对 MMCM 或 PLL 输出的时钟频率进行重配置，从而避免 PL 部分的重新编译。

Clocking Wizard 设置如下图所示。注意将 input clock 的时钟源设置为 Global buffer，因为输入时钟来自 PS 的 FCLK_CLK0，该时钟也被其他 IP 所使用，在进入 MMCM 之前已经位于全局时钟网络 BUFG。





输出时钟频率可设置也可不作设置，为了避免误解，在这里设置为所 LCD 显示屏需要的 51.2MHz。若用户使用其他触摸屏，需要更高的分辨率，建议将 clk_out1 设置一个大于等于最高分辨率所对应时钟频率的值，这样可以确保时序约束的可靠性。

The phase is calculated relative to the active input clock.					
Output Clock	Output Freq (MHz)		Phase	Request	Response
	Requested	Actual			
clk_out1	51.2	51.200	0.0		

17.5.6 IO 口

17.5.6.1 PL IO 信号定义

本例程除了 PS 部分的固定 IO 口之外，PL 部分引出的 IO 口如下图所示。

```

input [1:0] button_i;
input lcd_int_i;
output lcd_clk_o;
output [7:0]lcd_r_o;
output [7:0]lcd_g_o;
output [7:0]lcd_b_o;
output lcd_de_o;
output lcd_hs_o;
output lcd_vs_o;
inout lcd_iic_scl_io;
inout lcd_iic_sda_io;
output [0:0]lcd_pwm_o;
output lcd_wake_n_o;
output lcd_bl_en_o;
output [4:0]led_o;

```

信号定义如下表所示。

PL 部分 IO 口定义表

端口名称	方向	说明
button_i[1:0]	input	按键信号, m702 对应 SW3, SW4; miz701n 对应 SW1, SW2
lcd_int_i	input	触摸屏中断信号
lcd_clk_o	output	液晶屏数据时钟信号
lcd_r_o[7:0]	output	液晶屏像素点 R 分量
lcd_g_o[7:0]	output	液晶屏像素点 G 分量
lcd_b_o[7:0]	output	液晶屏像素点 B 分量
lcd_de_o	output	液晶屏数据有效信号
lcd_hs_o	output	液晶屏行同步信号
lcd_vs_o	output	液晶屏场同步信号
lcd_I2C_scl_io	inout	触摸屏 I2C 时钟信号
lcd_I2C_sda_io	inout	触摸屏 I2C 数据信号
lcd_pwm_o	output	液晶屏背光源 PWM 信号
lcd_wake_n_o	output	触摸屏唤醒信号
lcd_bl_en_o	output	液晶屏背光源使能信号
led_o[4:0]	output	LED 控制信号, m702 对应 LD1~LD5, miz701n 只使用 LD1~LD4, led_o[5]脚并非与 LD5 连 接, 而是连接到 40Pin 连接器 P1 的 T10 脚

17.5.6.2 LCD 时钟信号输出

Clocking Wizard 引出的时钟信号 lcd_clk, 需要经过 ODDR 产生 1 个反相的时钟 lcd_clk_o 与 LCD 触摸屏连接。目的在于使输出时钟 lcd_clk_o 的边沿位于 LCD 其他控制、数据信号窗口的中心位置, 使建立和保持时间最大化。

```

ODDR #((
    .DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"
    .INIT(1'b0),           // Initial value of Q: 1'b0 or 1'b1
    .SRTYPE("SYNC")        // Set/Reset type: "SYNC" or "ASYNC"
) lcd_clk_addr (
    .Q(lcd_clk_o),      // 1-bit DDR output
    .C(lcd_clk),         // 1-bit clock input
    .CE(1'b1),           // 1-bit clock enable input
    .D1(1'b0),           // 1-bit data input (positive edge)
    .D2(1'b1),           // 1-bit data input (negative edge)
    .R(1'b0),             // 1-bit reset
    .S(1'b0)              // 1-bit set
);

```

17.5.6.3 LCD RGB 信号映射

AXI4- Streamto Video Out 输出的 24 位像素点数据 lcd_data 与 LCD 触摸屏 RGB 信号的映射关系如下所示。

```

assign lcd_r_o = lcd_data[23:16];
assign lcd_g_o = lcd_data[15:8];
assign lcd_b_o = lcd_data[7:0];

```

17.5.6.4 LCD 控制信号

LCD 液晶屏的背光源使能信号与 AXI GPIO 的最高位连接，使 PS 可以控制背光源的开关。触摸屏唤醒信号置为 1。

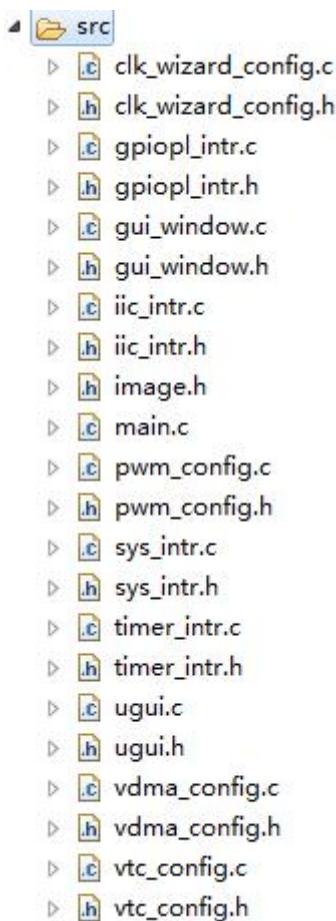
```

assign lcd_wake_n_o = 1'b1;
assign lcd_bl_en_o = gpio_rtl_tri_o[8];

```

17.6 PS 程序设计

PS 部分程序所有的源文件如下图所示。



17.6.1 main 函数

main 函数主要完成如下功能：

- 初始化 Clocking Wizard，并重新设置其输出时钟
- 初始化中断控制器及系统中断
- 初始化并配置定时器及其中断
- 初始化并配置 AXI GPIO 及其中断
- 初始化并配置 I2C 接口
- 初始化并配置 Video Timing Controller，设置显示分辨率
- 初始化并配置 AXI VDMA 及其中断
- 初始化并配置 AXI PWM，设置输出 PWM 信号频率及占空比
- 启动定时器工作
- 创建 GUI 界面
- 启动 AXI VDMA 工作
- 打开背光源，启动 AXI PWM 输出 PWM 信号点亮触摸屏
- 进入空循环，等待触摸屏中断，根据触摸坐标信息进行相应响应，GUI 界面产生相应变化

17.6.2 时钟重配置

在 PL 中，Clocking Wizard 使能了动态重配置功能，因此 PS 可以通过其 AXI-lite 接口动态重配置 MMCM 或者 PLL，来改变其时钟的输出频率。

17.6.2.1 SDK 库移植

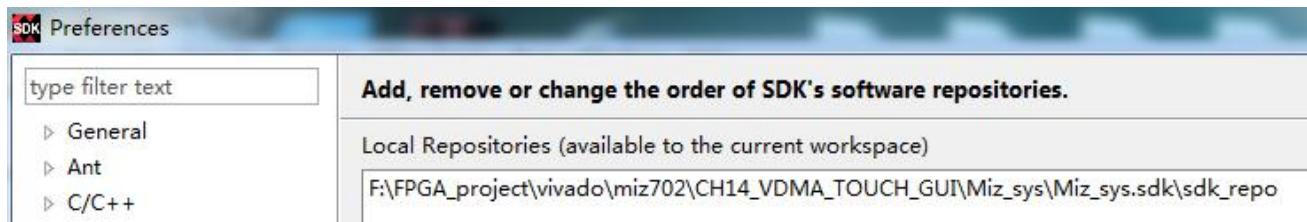
由于在 vivado 2017.4 的 SDK 中还没有 Clocking Wizard 的驱动函数库，缺少可以利用的 API 函数。而 vivado 2017.4 的 SDK 中包含了 Clocking Wizard 的驱动库。因此，为了便于开发，在 2017.4 中借用 2017.4 的驱动库进行设计。

首先，在 SDK 2017.4 的安装目录（如：

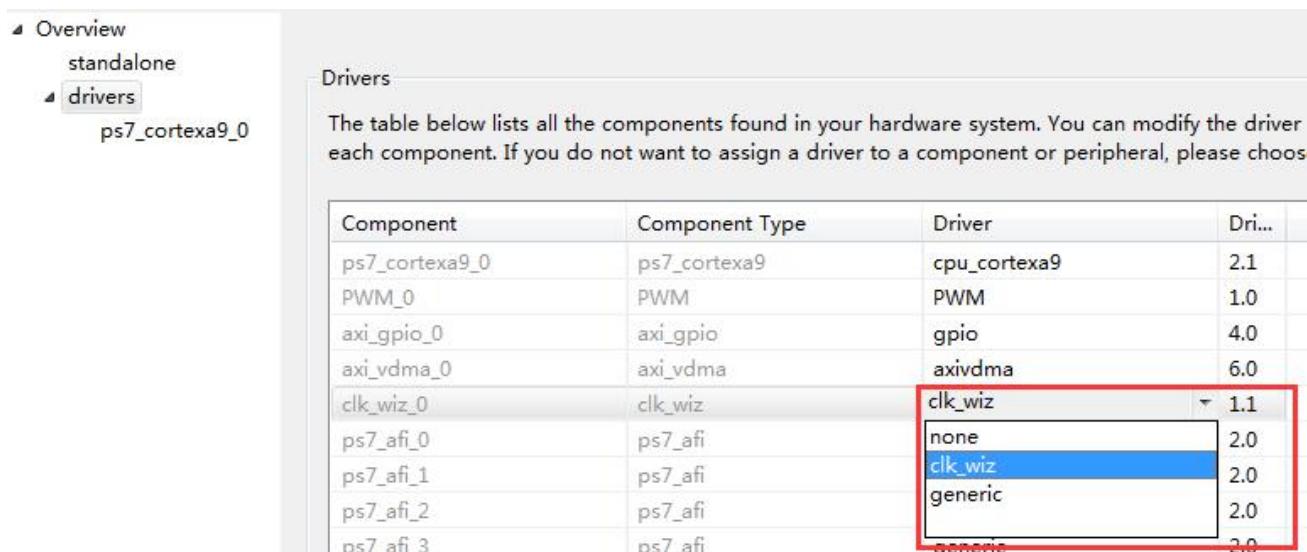
C:\Xilinx\SDK\2017.4\data\embeddedsw\XilinxProcessorIPLib\drivers）下找到 Clocking Wizard 的驱动库文件夹 clk_wiz_v1_1，如下图所示。



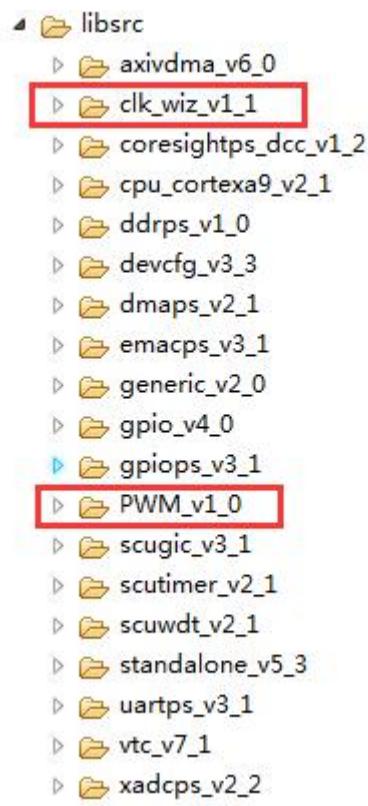
将文件夹 clk_wiz_v1_1 复制到工程目录下的 sdk_repo 的 bsp 文件夹中。然后在 sdk 中设置 sdk_repo 文件夹的路径，如下图所示。



然后更改工程 bsp 中的驱动函数设置，将 clk_wiz_0 的驱动改为 clk_wiz 1.1 版本，如下图所示。



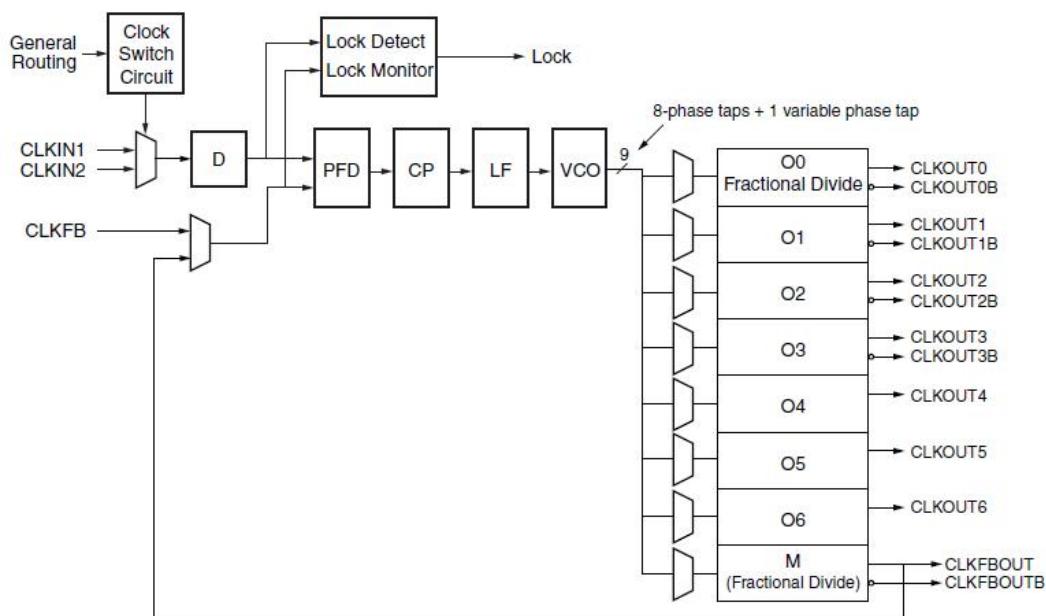
重新生成 bsp 后，在 bsp 的 libsrc 下就会出现 clk_wiz_v1_1 的驱动，如下图所示。



17.6.2.2 驱动程序

Clocking Wizard 的驱动程序由 `clk_wizard_config.c` 和 `clk_wizard_config.h` 组成。

本例程中，使用了 MMCM，MMCM 的重配置通过 `Clk_Wiz_Reconfig()` 函数完成，该函数参考自 SDK 中 `clk_wiz_v1_1` 的 example: `xclk_wiz_intr_example.c`。由于 SDK 关于 Clocking Wizard 的驱动还未完善，且 MMCM/PLL 内部的寄存器信息未有官方文档可参考，所以 `Clk_Wiz_Reconfig()` 不在此做具体分析。MMCM 的基本结构如下图所示。



MMCM 内部的压控振荡器 VCO 频率与 MMCM 的输出时钟频率的公式如下：

$$F_{VCO} = F_{CLKIN} \times \frac{M}{D}$$

$$F_{OUT} = F_{CLKIN} \times \frac{M}{D \times O}$$

ZYNQ 7010/7015/7020 内部 MMCM 的 VCO 频率范围为如下图所示。

Symbol	Description	Speed Grade				Units
		-3	-2	-1C/-1I/-1LI	-1Q	
MMCM_F_VCOMIN	Minimum MMCM VCO frequency	600.00	600.00	600.00	600.00	MHz
MMCM_F_VCOMAX	Maximum MMCM VCO frequency	1600.00	1440.00	1200.00	1200.00	MHz

动态配置 MMCM 其实就是通过 PS 改变其内部倍频系数 M，输入分频系数 D，以及输出分频系数 O 的值来实现。在 clk_wizard_config.h 有 4 个重要的宏定义，如下：

```
#define VCO_FREQ          600
#define DYNAMIC_INPUT_FREQ 100
#define DYNAMIC_OUTPUT_FREQ 51.2
#define CLK_FRAC_EN         1
```

其中 VCO_FREQ 表示 MMCM 工作时的 VCO 频率为 600MHz，DYNAMIC_INPUT_FREQ 表示 MMCM 输入的时钟频率为 100MHz，DYNAMIC_OUTPUT_FREQ 表示 MMCM 输出的时钟频率为 51.2MHz，CLK_FRAC_EN 表示允许 CLKOUT0 的分频系数为 1/8 精度的小数。

Clk_Wiz_Reconfig() 函数根据这 4 个宏定义的值对 MMCM 的 M、D、O 的值进行配置，并通过 Wait_For_Lock() 判断各环节的输出时钟是否 LOCK。用户也可以根据实际需求在允许的范围内调节 VCO 的频率。

17.6.3 PWM 信号输出

17.6.3.1 SDK 库设置

digilent AXI PWM IP 核对应的 PS 驱动库 PWM_v1_0 也位于工程目录 sdk_repo 的 bsp 文件夹中。在 bsp 的 libsrc 下可以看到 PWM_v1_0 驱动库，如 6.2.1 节图所示。

17.6.3.2 驱动程序

AXI PWM 的驱动程序由 pwm_config.c 和 pwm_config.h 组成。

本例程所使用的液晶屏建议输入 PWM 信号频率为 100Hz~200K Hz。在 main 函数中通过 PWM_Init()函数设置初始输出 PWM 信号的周期和占空比。

PWM_Init 函数：

- 通过 PWM_Set_Period 函数设置 PWM 信号周期
- 通过 PWM_Set_Duty 函数设置 PWM 信号低电平的占空比

上述函数都是以时钟周期数来设置。在 pwm_config.h 包含了 PWM 信号周期和占空比的时钟周期数宏定义。如下：

```
#define PERIOD_CLOCK_NUM      409600  
#define DUTY_CLOCK_NUM        204800
```

由于本例程中，PL 部分 AXI PWM 的参考时钟频率为 100MHz，一个时钟周期就是 10ns。因此，PWM_Init 所设置的 PWM 信号的周期为 4.096ms，占空比为 50%，频率约为 250Hz。

为了对 PWM 信号的占空比进行动态调节达到改变液晶屏背光源亮度的目的，又设计了 PWM_increase_duty()和 PWM_decrease_duty()函数用于增大和减小 PWM 信号中低电平的占空比。

最后，在 main 函数中调用 PWM_Enable()使能 PWM 信号输出。

17.6.4 GPIO 输入输出

AXI GPIO 的驱动程序由 gpiopl_intr.c 和 gpiopl_intr.h 组成。

在 main 函数调用 Gpiopl_init()函数初始化 AXI GPIO，设置 9 个 GPIO 的方向，其中 GPIO[0]~GPIO[2]为输入，GPIO[3]~GPIO[8]为输出。并将 GPIO[3]~GPIO[8]的输出都置为 0。每个 GPIO 的宏定义在 gpiopl_intr.h 中，如下所示。

```
#define TOUCH_INTR_MASK      0x00000001  
#define BUTTON0_INTR_MASK    0x00000002  
#define BUTTON1_INTR_MASK    0x00000004  
#define LED1_MASK            0x00000008  
#define LED2_MASK            0x00000010  
#define LED3_MASK            0x00000020  
#define LED4_MASK            0x00000040
```

```
#define LED5_MASK          0x00000080
#define LCD_BL_EN_MASK       0x00000100
```

通过 Gpiopl_Setup_Intr_System() 初始化并使能 AXI GPIO 的输入中断，GPIO[0]~GPIO[2] 的任意 1 个信号的输入发生 1 次改变将触发 1 次中断，GpioplIntrHandler()为 GPIO 的中断函数。

GpioplIntrHandler()函数：

- 判断 GPIO[0]输入的触摸屏中断信号是否为 0，若为 0，则调用 I2C_write()和 I2C_read()函数通过 I2C 接口读出触摸屏的触摸信息，并触摸屏中断标志位 touch_flag 置 1，该信号将在定时器中断中使用。
- 判断 GPIO[1]输入的 SW3 按键信号是否为 0，若为 0，将 GPIO[8]信号拉低，打开液晶屏背光源。
- 判断 GPIO[2]输入的 SW4 按键信号是否为 0，若为 0，将 GPIO[8]信号拉高，关闭液晶屏背光源。

最后，在 main 函数中将 GPIO[8]置为 1，开启液晶屏的背光源。

GPIO[3]~GPIO[7]信号的输出在 gui_window.c 中进行控制，用于在 GUI 界面中控制开发包中的 LED 灯。

17.6.5 I2C 读取触摸信息

PS 的 I2C 接口的驱动程序由 iic_intr.c 和 iic_intr.h 组成。

在本例程中，PS 的 I2C 接口不能工作于中断模式，其原因在于，通过 I2C 接口读写触摸屏是在 GPIO 的中断函数 GpioplIntrHandler()中执行，此时若 I2C 接口产生中断，该中断将不能在 GPIO 中断中被嵌套响应，无法完成读写操作。因此，I2C 接口只能工作于轮询 Poll 模式。

触摸屏中 FT5206 控制芯片的 I2C 地址和 I2C 时钟频率设置如 iic_intr.h 宏定义所示。I2C 地址为 0x38，I2C 时钟频率设为 400KHz，为 FT5206 的最高值。

```
#define IIC_SLAVE_ADDR    0x38
#define IIC_SCLK_RATE      400000
```

在 main 函数中调用 Iic_init()函数初始化 PS 的 I2C 接口，设置 I2C 时钟频率。当触摸屏中断信号拉低触发 GPIO 产生中断后，在 GPIO 中断函数 GpioplIntrHandler()中首先调用 I2C_write()向 FT5206 芯片发送需要读取的寄存器首地址。在本例程中，需要从地址为 0x02 的寄存器开始读，因此 I2C_write()发送的首地址为 0x02。然后，调用 I2C_read()函数从 FT5206 连续读取 29 个寄存器的值，每个寄存器的值为 1 个字节，一共读取 29 个字节的触摸信息。

I2C_write()和 I2C_read()函数均以轮询模式控制 I2C 接口。

17.6.6 GUI 界面显示

GUI 界面显示在 PL 部分由 AXI VDMA，Video Timing Controller，AXI4- Sreamto Video Out 三个 IP 协同完成。其中 AXI VDMA 和 Video Timing Controller 需要 PS 进行设置，AXI VDMA 完成 PS 端 DDR3 中 GUI 界面的读取，Video Timing Controller 产生通过 AXI4- Sreamto Video Out 进行 GUI 界面显示的控制时序。

17.6.6.1 AXI VDMA

AXI VDMA 的驱动程序由 vdma_config.c 和 vdma_config.h 组成。

在本例程中，AXI VDMA 仅有 MM2S 方向（PS DDR 到 PL）的读通道工作。另外，需要让 AXI VDMA 工作于 free run 模式，因此，只使能错误中断。

在 main 函数中调用 Vdma_Init() 函数对 AXI VDMA 进行初始化，在 Vdma_Init() 中调用 ReadSetup() 对 MM2S 读通道的参数进行设置。VDMA 读通道的相关设置由 vdma_config.h 中的宏定义所决定，如下所示。

```
#define IMAGE_WIDTH      GUI_WIDTH
#define IMAGE_HEIGHT     GUI_HEIGHT
#define BYTES_PER_PIXEL 4
#define NUMBER_OF_READ_FRAMES 1
#define MEM_BASE_ADDR    0x10000000
#define BUFFER0_BASE     (MEM_BASE_ADDR)
```

- IMAGE_WIDTH，图像宽度，1024
- IMAGE_HEIGHT，图像高度，600
- BYTES_PER_PIXEL，每个像素点的字节数为 4
- NUMBER_OF_READ_FRAMES，图像缓存数量，实际只需使用 1 个缓存
- BUFFER0_BASE，图像缓存的首地址，0x10000000

然后，在 main 函数中调用 Vdma_Setup_Intr_System() 函数，配置使能 VDMA 读通道的错误中断，ReadErrorCallBack() 为中断函数。

最后，调用 Vdma_Start() 函数，启动 AXI VDMA 的读通道开始工作。

17.6.6.2 显示时序设置

Video Timing Controller 的驱动程序由 vtc_config.c 和 vtc_config.h 组成。

在本例程中，液晶屏使用了 $1024 \times 600 @ 60Hz$ 显示分辨率，使用的时序参数如下：

<ul style="list-style-type: none"> ● 行总像素数：1344 ● 行有效像素数：1024 ● 行同步前肩像素数：24 ● 行同步信号像素数：136 ● 行同步后肩像素数：160 	<ul style="list-style-type: none"> ● 场总行数：635 ● 场有效行数：600 ● 场同步前肩行数：8 ● 场同步信号行数：4 ● 场同步后肩行数：23
--	---

在 main 函数中调用 Vtc_init() 函数对 Video Timing Controller 进行初始化，将上述的显示时序参数写入其中，然后使能其产生控制时序信号。

17.6.7 定时器

PS 定时器的驱动程序由 timer_intr.c 和 timer_intr.h 组成。

在本例程中，PS 定时器用于周期性产生中断来实现 GUI 界面的刷新，以固定的频率的调用 UG_Update() 函数实现 GUI 的动态特性。

首先，在 main 函数中调用 Timer_init() 函数对定时器进行初始化，其中断周期由以下宏定义决定，周期为 20ms，即 GUI 的刷新频率为 50Hz，刷新频率越高，GUI 的动

态变化越快，用户可以根据需求进行调整。

```
#define TIMER_LOAD_VALUE XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ / 100
```

然后，在 main 函数中调用 Timer_Setup_Intr_System() 函数使能定时器的中断，将中断函数设置为 TimerIntrHandler()。

TimerIntrHandler() 函数的过程如下：

- 关闭定时器中断，因为定时器中断函数的执行时间可能会超过定时器设置的中断周期，如 20ms。
- 若触摸屏中断标志位 touch_flag 为 1，则对 GPIO 中断函数中读出的 29 字节触摸信息进行判断，若第一个触摸点状态为接触(contact)，则读出并计算出第一个触摸点的 x,y 坐标信息，通过 UG_TouchUpdate() 函数将该触摸坐标反馈至 GUI 界面，表示 GUI 上的该点被触摸。若第一个触摸点状态为抬起(put up)，则将无效坐标(-1, -1)通过 UG_TouchUpdate() 反馈至 GUI 界面，表示 GUI 上的无任何点被触摸。目前 μGUI 库的按键触摸功能只支持单点触摸，因此无法同时设计出两个按键同时被触摸的效果。
- 若当前的窗口为 window 4，则为 5 点绘图窗口。此时，读出 5 个触摸点的坐标信息，以每个触摸点为圆心，沿每个手指移动轨迹不断画圆。5 个触摸点的圆形图案颜色按顺序依次为：红、黄、蓝、绿、橙。
- 调用 UG_Update() 刷新 GUI 界面，让触摸使 GUI 产生的动态变化更新至对应的图像中。
- 将变化后最新的 GUI 界面对应的图像通过 Xil_DCacheFlushRange() 函数刷进 DDR 中，从而保证 VDMA 将最新的 GUI 界面读出，并显示在触摸屏上。
- 重新使能定时器中断。

最后，在 main 函数里调用 Timer_start() 函数启动定时器工作。

17.6.8 GUI 界面设计

GUI 界面的驱动程序由 gui_window.c、gui_window.h、image.h、ugui.c 和 ugui.h 组成，其中 ugui.c 和 ugui.h 来自 μGUI 库。

在本例程中，一共设计了 5 个窗口，对应 window1~window5。同时，也为每个窗口设计了对应的回调函数 window_1_callback()~window_5_callback()。

在 main 函数中，调用 gui_create() 函数对 GUI 进行初始化，并创建所有的窗口及窗口所包含的所有对象。

gui_create() 函数的流程如下：

- 调用 UG_Init() 函数初始化 GUI
- 调用 UG_FillScreen() 函数设置所有 GUI 窗口的背景颜色
- 调用 create_window1()~create_window5() 函数依次创建 5 个窗口
- 调用 UG_WindowShow() 函数设置 GUI 显示的第一个窗口为 window1
- 调用 UG_WaitForUpdate() 函数等待定时器中断到来，刷新并显示 GUI 界面

GUI 界面设计以窗口为基础进行，每个界面对应一个窗口，但所有的窗口都对应同一片内存区域。μGUI 库中有很多 API 函数，这里只介绍本例程所涉及的 API 函数，其余的 API 可参考 μGUI 使用手册。5 个窗口的设计在 create_window1()~create_window5() 函数中完成。

17.6.8.1 GUI 初始化

通过 UG_Init()函数设置 GUI 界面的长宽分辨率，并绑定在 4.1.1 节所述用户自定义的像素值设置函数 PixelSet()。通过 UG_FillScreen()函数将所有 GUI 窗口的背景设置为浅灰色。

17.6.8.2 窗口 1 设计

窗口 1 为 GUI 的欢迎界面。如下图所示。



窗口 1 的设计在 create_window1()函数中完成，流程如下：

1) 窗口创建

- 调用 UG_WindowCreate()函数绑定窗口 1 的回调函数为 window_1_callback(), 设置窗口可包含对象的最大个数为 15。
- 调用 UG_WindowSetTitleText()函数设置窗口 1 的标题内容。
- 调用 UG_WindowSetTitleTextFont()函数设置标题字体的大小。

2) 按键创建

- 调用 UG_ButtonCreate()函数创建按键 0，并设置按键 0 的坐标范围
- 调用 UG_ButtonSetStyle()函数设置按键 0 的模式为 3D，且按下后背景和字体的颜色会跳变
- 调用 UG_ButtonSetAlternateForeColor 函数设置按键 0 按下后改变的字体颜色
- 调用 UG_ButtonSetAlternateBackColor 函数设置按键 0 按下后改变的按键背景颜色
- 调用 UG_ButtonSetFont 函数设置按键 0 的字体大小
- 调用 UG_ButtonSetText 函数设置按键 0 显示的内容

3) 文本框创建

- 文本框 0
 - 调用 UG_TextboxCreate() 函数创建文本框 0，并设置文本框 0 的坐标范围。
 - 调用 UG_TextboxSetFont() 函数设置文本框 0 内容的字体大小。
 - 调用 UG_TextboxSetText() 函数设置文本框 0 的内容。
 - 调用 UG_TextboxSetAlignment() 函数设置文本框 0 中的内容在文本框中的对齐方式。
- 文本框 1
 - 调用 UG_TextboxSetForeColor() 函数设置文本框 1 内容的字体颜色
 - 调用 UG_TextboxSetHSpace() 函数设置文本框 1 中每个字符之间的横向距离
 - 其余函数调用与文本框 0 同理

4) 图片创建

目前，最新版的 μ GUI v0.3 仅支持在 GUI 界面中添加 RGB565 格式的 BMP 图像。由于本设计中所需添加的两个米联 logo 均为 RGB888 格式，为了让 μ GUI 能支持 24 位 RGB888 的图片，需要对 ugui.c 文件中的 UG_DrawBMP() 函数进行修改，如下所示。其中通过阴影部分为添加的代码。

```
void UG_DrawBMP(UG_S16xp, UG_S16yp, UG_BMP* bmp )
{
    UG_S16x,y,xs;
    UG_U8r,g,b;
    UG_U16* p;
    /*add by osrc 2017.2.18*/
    UG_U32* p1;
    UG_U16tmp;
    /*add by osrc 2017.2.18*/
    UG_U32 tmp1;
    UG_COLOR c;

    if ( bmp->p == NULL ) return;

    /* Only support 16 bpp so far */
    if ( bmp->bpp == BMP_BPP_16 )
    {
        p = (UG_U16*)bmp->p;
    }
    /*add support for 32 bpp, by osrc 2017.2.18*/
    elseif(bmp->bpp == BMP_BPP_32)
    {
        p1 = (UG_U32*)bmp->p;
    }
}
```

```

else
{
    return;
}

xs = xp;
for(y=0;y<bmp->height;y++)
{
    xp = xs;
    for(x=0;x<bmp->width;x++)
    {
        /*add support for RGB888, by osrc 2017.2.18*/
        if(bmp->colors == BMP_RGB565)
        {
            tmp = *p++;
/* Convert RGB565 to RGB888 */
            r = (tmp>>11)&0x1F;
            r<<=3;
            g = (tmp>>5)&0x3F;
            g<<=2;
            b = (tmp)&0x1F;
            b<<=3;
            c = ((UG_COLOR)r<<16) | ((UG_COLOR)g<<8) | (UG_COLOR)b;
        }
        else
        {
            tmp1 = *p1++;
            c = tmp1;
        }
        UG_DrawPixel(xp++ , yp , c );
    }
    yp++;
}
}
➤ 图片 0
添加的图片 0 如下所示，为米联的 logo。图片各像素点的值包含在 image.h 的 logo1_bmp 数组中。

```



定义该图片在 GUI 中的结构体，如下所示。图片的指针 logo1_bmp，为图片的长、宽均为 65 个像素，每个像素点占 32bit，图片格式为 RGB888。

```
constUG_BMP logo1 =
{
    (void*)logo1_bmp,
    65,
    65,
    BMP_BPP_32,
    BMP_RGB888
};
```

- 调用 UG_ImageCreate 函数在窗口 1 中创建该图片对象，并设置图片的坐标位置。
- 调用 UG_ImageSetBMP 函数将上述的 logo 图像与创建的图片对象绑定。

➤ 图片 1

添加的图片 1 如下所示，为米联客的 logo。图片各像素点的值包含在 image.h 的 logo2_bmp 数组中。



定义该图片在 GUI 中的结构体，如下所示。图片的指针 logo2_bmp，为图片的长位 259 像素，宽为 105 个像素，每个像素点占 32bit，图片格式为 RGB888。

```
constUG_BMP logo2 =
{
    (void*)logo2_bmp,
    259,
    105,
    BMP_BPP_32,
    BMP_RGB888
};
```

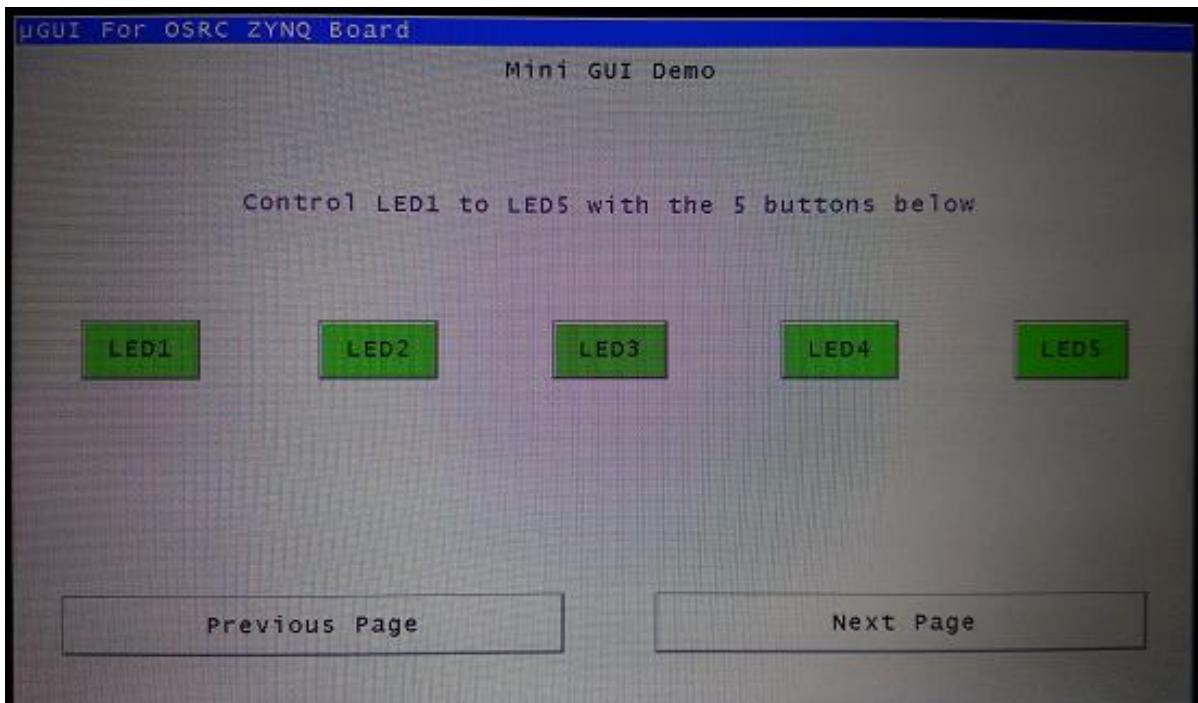
图片对象创建添加方式与图片 0 相同。

5) 回调函数

1 个窗口的回调函数当该窗口中的按键的触摸状态发生改变时，会在 UG_Update() 函数中被调用。在窗口 1 的回调函数 window_1_callback() 中设置了按键 0 的功能，当在触摸屏中按下 start application now 按键便会使 window_1_callback() 被调用，通过 UG_WindowShow 函数，让 GUI 界面刷新后切换到窗口 2。在按下按键的同时，可以观察到按键 0 字体和背景颜色产生的变化。

17.6.8.3 窗口 2 设计

窗口 2 为 LED 灯控制界面，如下图所示。在该窗口中，用户可以通过触摸 LED1~LED5 按键来控制开发板上的标识为 LD1~LD5 的 LED 灯。



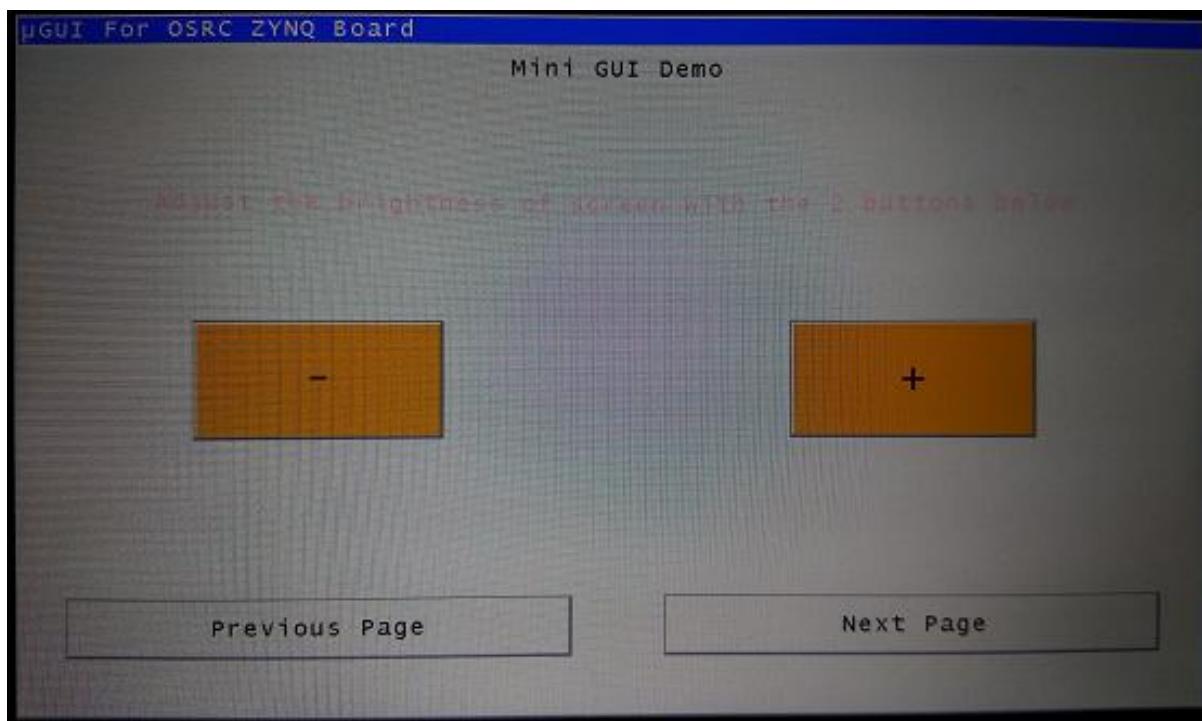
窗口 2 的设计在 `create_window2()` 函数中完成，窗口 2 中的标题、按键、文本框的创建流程与 `create_window1()` 原理相同，详细可参阅工程代码，此处不作赘述。

1) 回调函数

窗口 2 的回调函数 `window_2_callback()` 中设置了 7 个按键 0~6 的功能，其中 LED1~LED5 对应 5 个 LED 灯的控制功能，通过控制 GPIO[3]~GPIO[7] 的输出来实现 LED 灯的开关。当 LED 灯亮时，对应按键的背景颜色为红色，当 LED 熄灭时，对应按键的背景颜色还原为绿色。（注意 miz701n 只有 LED1~LED4 受控制，按键 LED5 实际无控制 LD5 的作用）另外，按下 Previous Page 按键让 GUI 界面切换至窗口 1，按下 Next Page 按键让 GUI 界面切换至窗口 3。在按下按键的同时，可以观察到各按键字体和背景颜色产生的变化。

17.6.8.4 窗口 3 设计

窗口 3 为液晶屏亮度调节界面，如下图所示。在该窗口中，用户可以通过触摸加、减按键来调节液晶屏背光源的亮度。



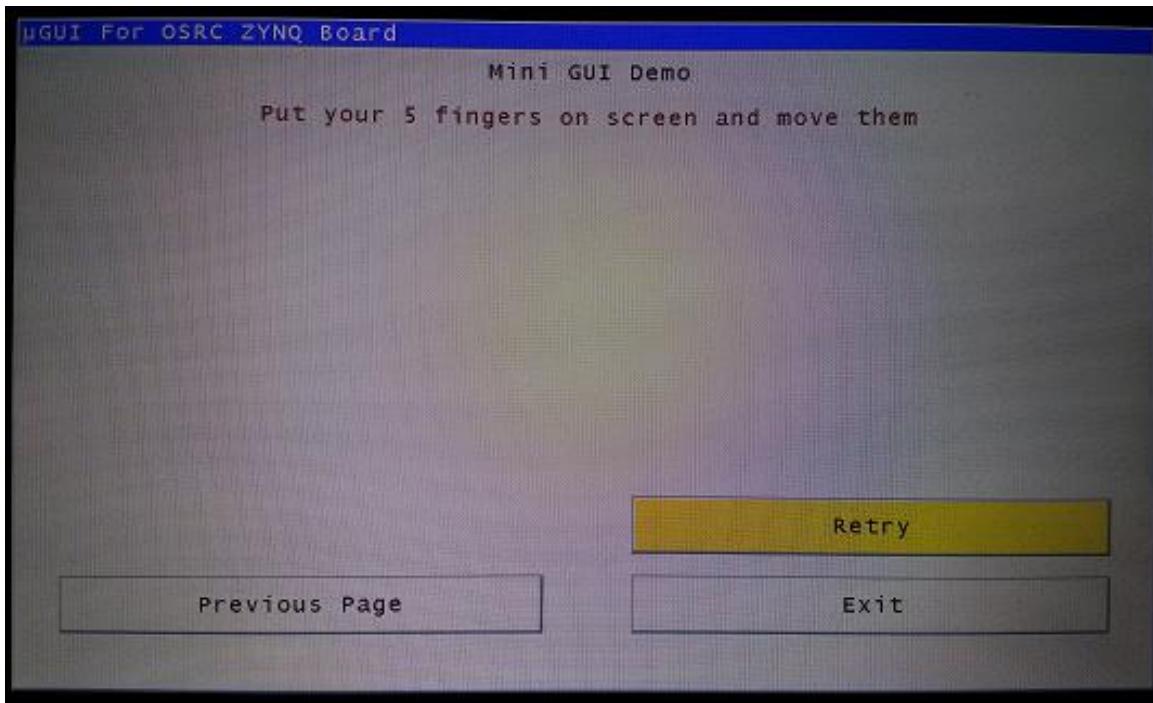
窗口 3 的设计在 `create_window3()` 函数中完成，窗口 3 中的标题、按键、文本框的创建流程与 `create_window1()` 原理相同，详细可参阅工程代码，此处不作赘述。

1) 回调函数

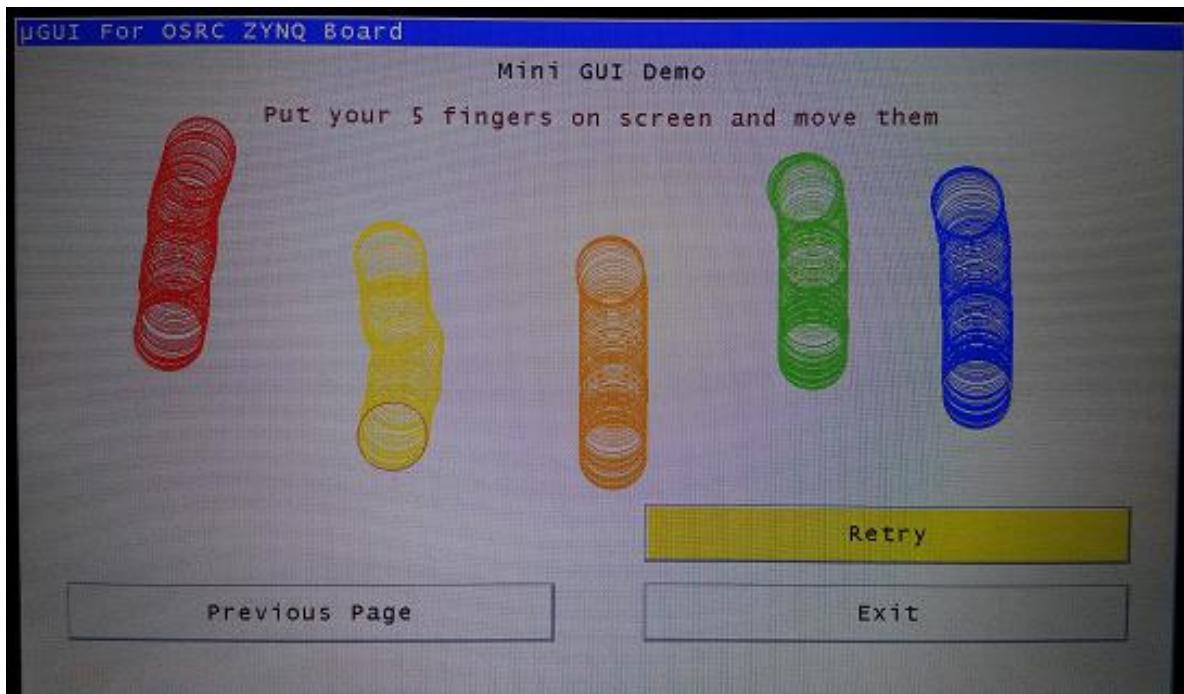
窗口 3 的回调函数 `window_3_callback()` 中设置了 4 个按键 0~3 的功能，其中“-”按键控制 PWM 信号低电平的占空比减小，从而降低液晶屏亮度；“+”按键控制 PWM 信号低电平的占空比增大，从而提高液晶屏亮度。另外，按下“Previous Page”按键让 GUI 界面切换至窗口 2，按下“Next Page”按键让 GUI 界面切换至窗口 4。在按下按键的同时，可以观察到各按键字体和背景颜色产生的变化。

17.6.8.5 窗口 4 设计

窗口 4 为绘图界面，如下图所示。在该窗口中，可实现 5 点触控，用户可以通过 5 个手指同时触摸平面来进行简单画图。



绘图时，以每个手指触摸点为圆心，沿每个手指移动轨迹不断画圆。5个触摸点的圆形图案颜色按顺序依次为：红、黄、蓝、绿、橙。由于 GUI 界面存在一定刷新的时间间隔，因此绘图轨迹中间会出现间断的现象，如下图所示。这里用户自行改进来实现真实的绘图效果。



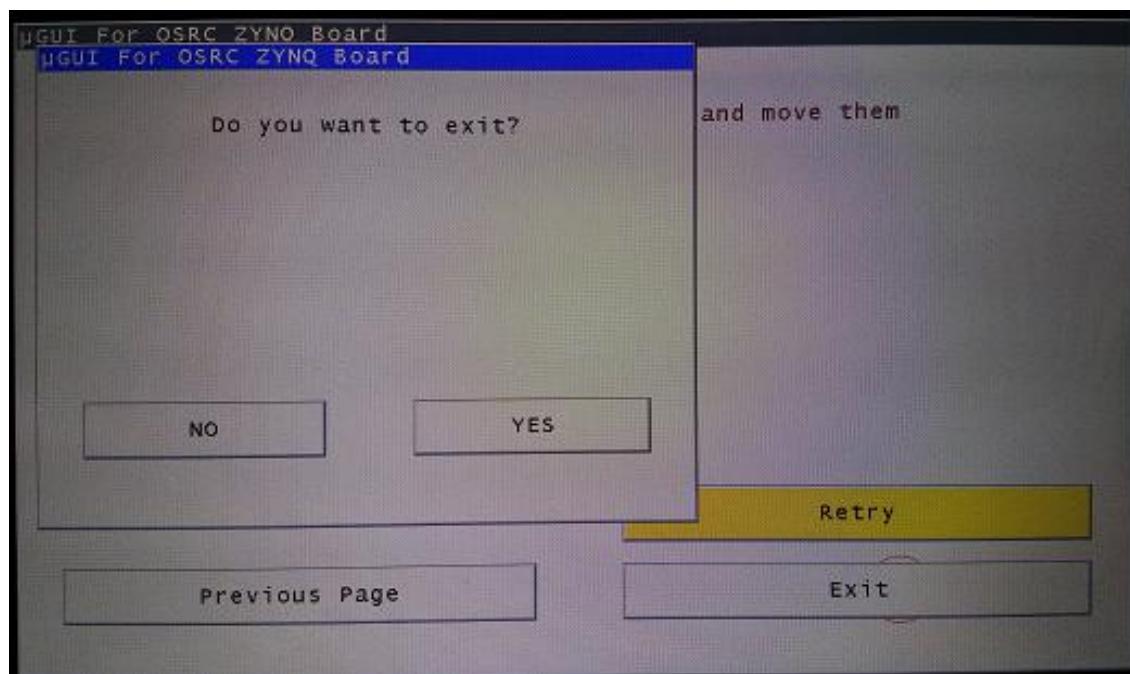
窗口 4 的设计在 `create_window4()` 函数中完成，窗口 4 中的标题、按键、文本框的创建流程与 `create_window1()` 原理相同，详细可参阅工程代码，此处不作赘述。

1) 回调函数

窗口 4 的回调函数 `window_4_callback()` 中设置了 3 个按键 0~2 的功能，其中“Retry”按键用于清除当前窗口中的绘图结果，使用户可以重新进行绘图。另外，按下“Previous Page”按键让 GUI 界面切换至窗口 3，按下“Next Page”按键让 GUI 界面切换至窗口 5。在按下按键的同时，可以观察到各按键字体和背景颜色产生的变化。

17.6.8.6 窗口 5 设计

窗口 5 为退出界面，如下图所示。在该窗口中，用户可以通过触摸“YES”、“NO”按键来选择是否退出回到窗口 1 的欢迎界面。由于窗口 5 的尺寸小于其他窗口，因此当 GUI 界面从窗口 4 切换到窗口 5 时，可以看出两个窗口叠加的效果，并且当窗口 5 出现时，窗口 4 的标题栏会由蓝变灰。



窗口 5 的设计在 `create_window5()` 函数中完成，由于窗口 5 的尺寸小于 1024×600 ，因此需要调用 `UG_WindowResize()` 函数重新设置窗口 5 的尺寸大小及位置。窗口 5 中的标题、按键、文本框的创建流程与 `create_window1()` 原理相同，详细可参阅工程代码，此处不作赘述。

2) 回调函数

窗口 5 的回调函数 `window_5_callback()` 中设置了 2 个按键 0~1 的功能，按下“YES”按键让 GUI 界面切换至窗口 1，按下“NO”按键让 GUI 界面切换至窗口 4。在按下按键的同时，可以观察到各按键字体和背景颜色产生的变化。

17.7 注意事项

17.7.1 更改 GUI 分辨率

若用户要接其他分辨率的触摸屏，需要更改触摸屏显示分辨率时，需要更改 3 个地方。

- main 函数中 Vtc_init() 函数中的分辨率参数
- 更改 gui_window.h 中的 2 个宏定义

```
#define GUI_HEIGHT 600
#define GUI_WIDTH 1024
```
- 更改 clk_wizard_config.h 中的 1 个宏定义，将其改为所需要的时钟频率，单位为 MHz

```
#define DYNAMIC_OUTPUT_FREQ 51.2
```

17.7.2 SDK 路径设置

注意 6.2.1 节关于 clk_wiz_v1_1 与 PWM_v1_0 库路径设置的描述，用户将原版程序在自己的电脑上运行时，务必要重新设置该路径，否则 SDK 将提示错误。

17.7.3 miz701n 的 LD5 不受 GUI 控制

由于 miz701n 的 LD5 与 PS 的 MIO51 脚连接，需要由 PS 控制，为了确保 miz702 和 miz701n 程序的一致性，避免引入误解，只使用 miz701n 的 LD1~LD4，LD5 不受 GUI 窗口 2 中 LED5 按键的控制。

第四季 LINUX 系统开发共计 16 课时

第四季课程共计 16 课时，主要讲解 LINUX 开发环境搭建，LINUX 如何移植，如何修改设备树，批处理文件的使用和理解。EMMC 测试、编译 QSPI FLASH UBOOT.BIN 文件，烧写 UBOOT.BIN 到 QSPI FLASH。通过 U 盘烧写 UBOOT.BIN。

然后会讲解 LINUX 驱动入门，QT 变成入门、OPENCV 移植入门等教程。

由于目前本季课程还在更新，请大家耐心等待，并且关注我们 VIP QQ 群的最新发布。

S04_CH01_搭建工程移植 LINUX/测试 EMMC/VGA

1.1 概述：

本章内容是在已经提供安装了VIVADO2015.4的ubuntu系统下，进行。大家以下周我们已经提供的虚拟机镜像，我们提供的虚拟机镜像是安装了VIVADO的ubuntu系统，系统版本是ubuntu14.04。

主要完成的内容如下：

- 1)、利用 VIVADO 搭建 VDMA Framebuffer 工程 修改 VTG IP 模块 支持 1024X600 分辨率（主要考虑支持 7 寸 HDMI 液晶显示器）
- 2)、产生 FSBL 文件
- 3)、环境变量的批量设置
- 4)、基于 linux kernel 部分 zynq_zed.dts (miz702n 需要使用到)/zynq_zybo(miz701n 需要使用到) 修改设备树
- 5)、修改 kernel 其他文件
- 6)、通过 menuconfig 向导配置 framebuffer 驱动
- 7)、编译 kernel、编译 uboot
- 8)、测试显示器输出和串口打印信息
- 9)、测试读 EMMC 内存、写 EMMC 内存、读写 EMMC 内存
- 10)、测试 framebuffer 应用程序

1.2 LINUX 开发环境搭建

1.2.1 虚拟机环境配置（提供下载虚拟机已经完成）

Step1:

本例程的工作环境（包括 FPGA 及嵌入式 Linux 的开发）是在 ubuntu14.04 操作系统下完成，对于其它 Linux 操作系统可能需要解决相关包的依赖问题。

Step2:

例子放在 /mnt/workspace/linux 目录下，读者可以在该目录下正确编译、运行。而 /mnt/workspace/linux 目录是为读者实验准备的。

Step3:

单击桌面上的控制台或者 (ctrl+alt+T) 即可打开命令行，然后输入 su，根据提示输入 root 密码即可切换到 root 用户。

Step4:

对于新安装的 ubuntu 操作系统需要命令行下运行一下 scripts 目录下的 fix_xilinx_deps.sh 脚本，该脚本主要是解决编译 u-boot、kernel 源码等所需要的包依赖。而提供的虚拟机已经解决了这些问题。

/mnt/workspace/linux/scripts/fix_xilinx_deps.sh

注意：开机的时候可能系统会提示正在检查更新（如），此时可以打开图示的有个

勾的那个图标，然后点击 Install Updates，待其完成更新后再运行该脚本（如下图所示）。当然，在平时的开发过程中，可以在打开虚拟机之前，禁止网络功能，提供的虚拟机已经禁止了。

Step5：

对于 Vivado 开发工具的安装，将下载的压缩包解压后，从命令行进入该目录，执行 xsetup 即可像在 Windows 一样安装。注意：为减少虚拟机所占用硬盘空间，虚拟机里提供的开发套件是直接将本人 PC 中安装好的 Vivado 和 SDK 等复制到 /mnt/workspace/toolchains 目录中的，这里不提供全新安装 Vivado 的步骤，若需要帮助的话，可以通过邮件联系我。

Step6：

本开发使用的是 Vivado 开发套件里提供的交叉编译器，无须再安装其它交叉编译器。

Step7：

整个开发过程主要使用脚本进行操作，故在每次开发前，需要执行如下图所示操作来设置好环境变量。如果需要支持新的开发板时，只需要建立新的目录（如 miz702 等），然后把 scripts 复制到新的目录中，修改相关设置（如 uboot 版本等），十分方便。

1.2.2 下载资源

使用 get_xilinx_sources.sh 脚本下载 uboot、kernel、device tree 等源码及 ramdisk 到 packages 目录中，并解压 uboot 和 kernel 等源码。若需要更改源码的版本，则打开 get_xilinx_sources.sh 文件后，修改相应的设置即可。当网络不是很好时，可以直接压缩包目录中的包复制到 packages 目录下即可。

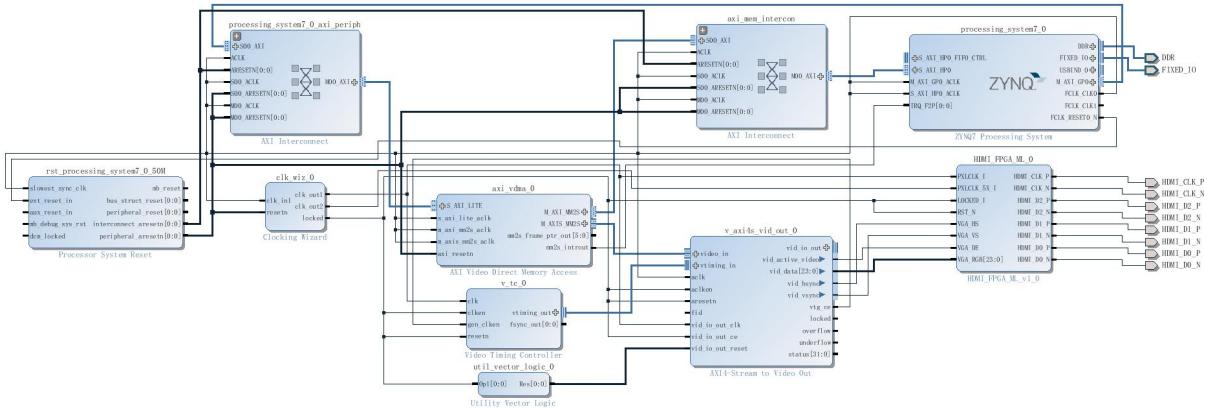
1.3 VIVADO 工程的搭建

计 FPGA 这部分相信读者已经相当熟悉了，这里只是对工程里的一些关键地方进行说明。在命令行下切换到 /mnt/workspace/MIZ702N (工程目录读者可以自己指定) 目录下，通过输入 vivado 即可打开 vivado 工具。

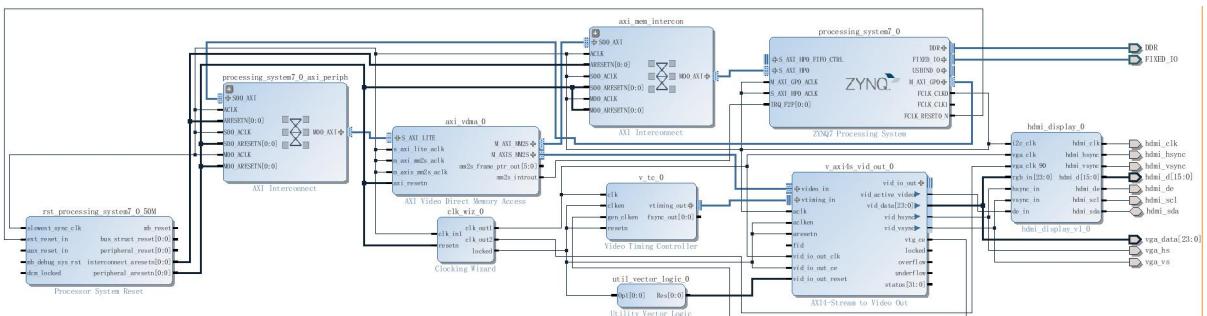
此工程可以在 WINDOWS 或者 LINUX 下使用。建议初学者 WINDOWS 下使用。

1.3.1 VIVADO 硬件工程构架

MIZ701N

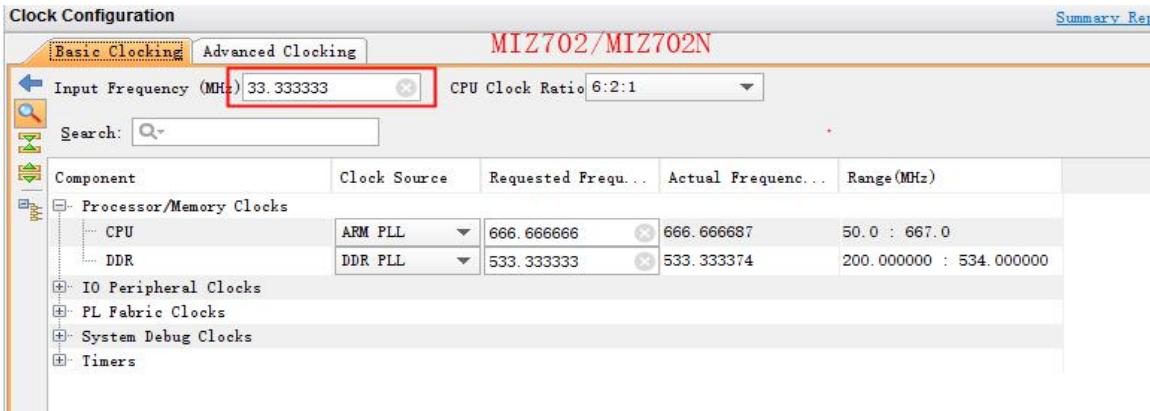


MIZ702/MIZ702N

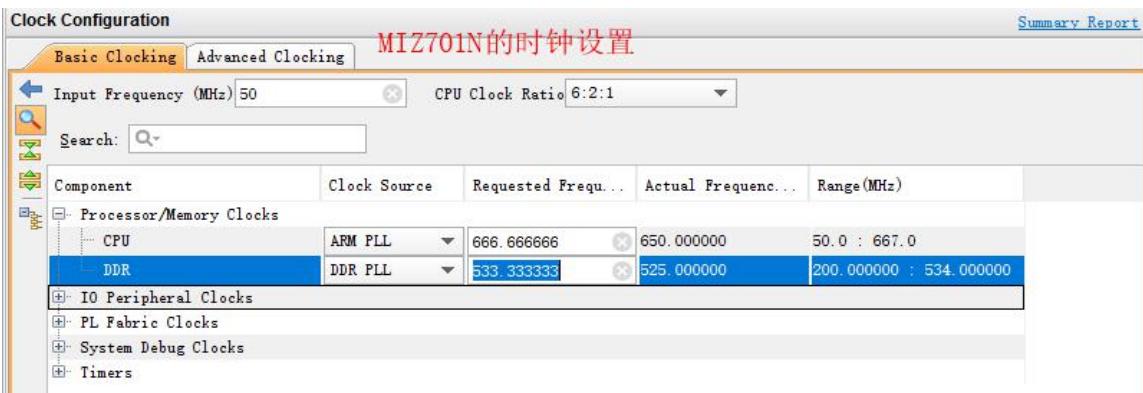


1.3.2 时钟设置

Step1： 双击 ZYNQ CPU IP 进行如下步骤设置： MIZ702 和 MIZ702N 的输入时钟是 333.333333MHZ



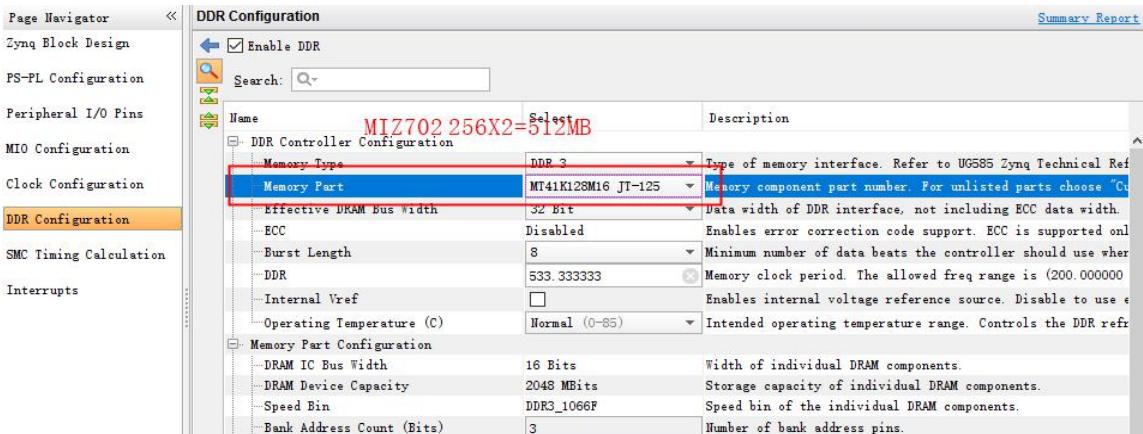
Step2： MIZ701N PS 的输入时钟是 50MHz



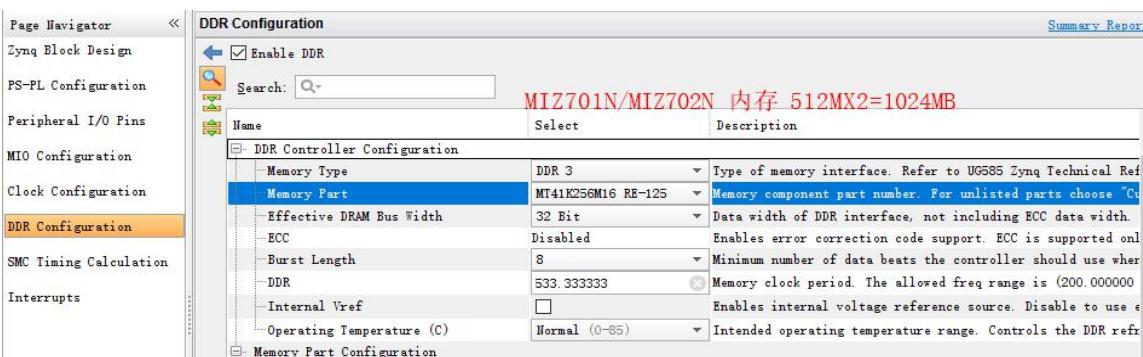
Step3: PS 的 PLL 提供本系统的时钟 100MHZ

Component	Clock Source	Requested Freq...	Actual Freq...	Range (MHz)
Processor/Memory Clocks				
CPU	ARM PLL	666.666666	650.000000	50.0 : 667.0
DDR	DDR PLL	533.333333	525.000000	200.000000 : 534.000000
IO Peripheral Clocks				
PL Fabric Clocks				
System Debug Clocks				
Timers				

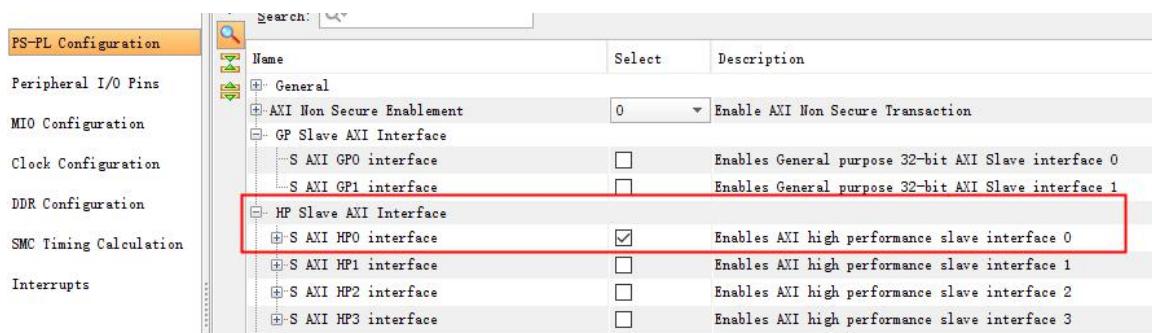
Step4: MIZ702 的开发板采用的是单片 256MB 的 MT41K128M16JI-125



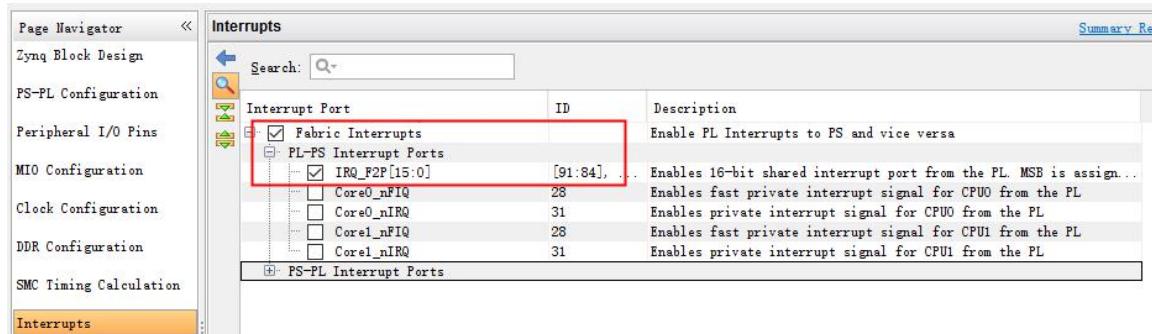
Step5: MIZ701N 和 MIZ702N 的内存型号一样，都是单片 512MB 的 MT41K256M16RE-125



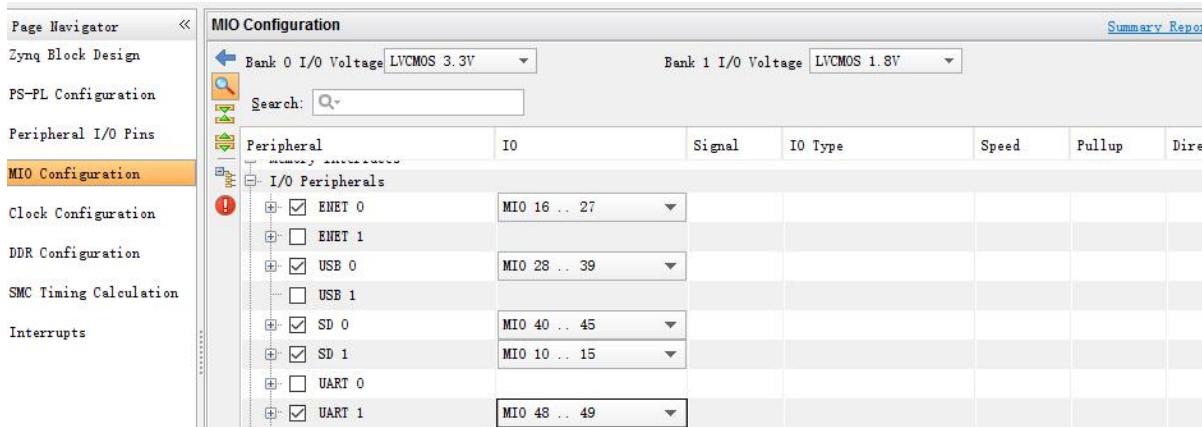
Step6: 启动 1 路 HP 接口，HP 接口是 ZYNQ 个高速数据接口



Step7:勾选 PL 到 PS 的中断资源（关于中断，在第二季的课程中有详细讲解，不熟悉的读者可以到第二季课程中温习一下）

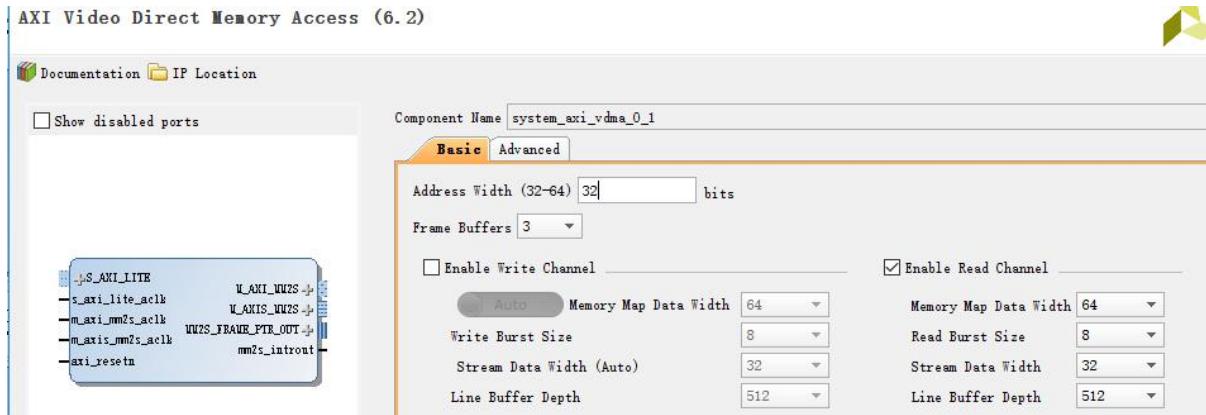


Step8:增加必要的外设，包括 ENET0 以太网接口、USB_0 USB 接口、SD0 TF 卡接口、SD1 EMMC 接口、UART1 串口。

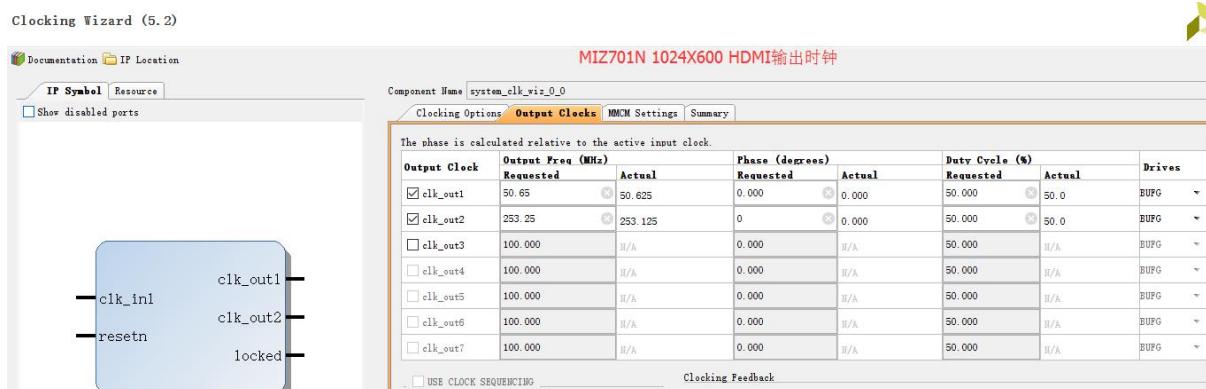


Step9:设置完成后单击 OK

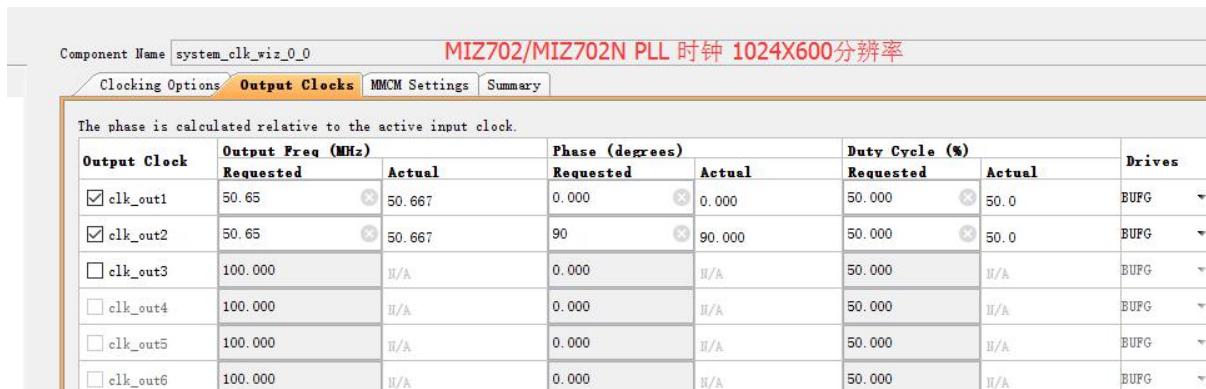
Step10:双击 VDMA IP 由于只使用了 VDMA 读通道设置如下：



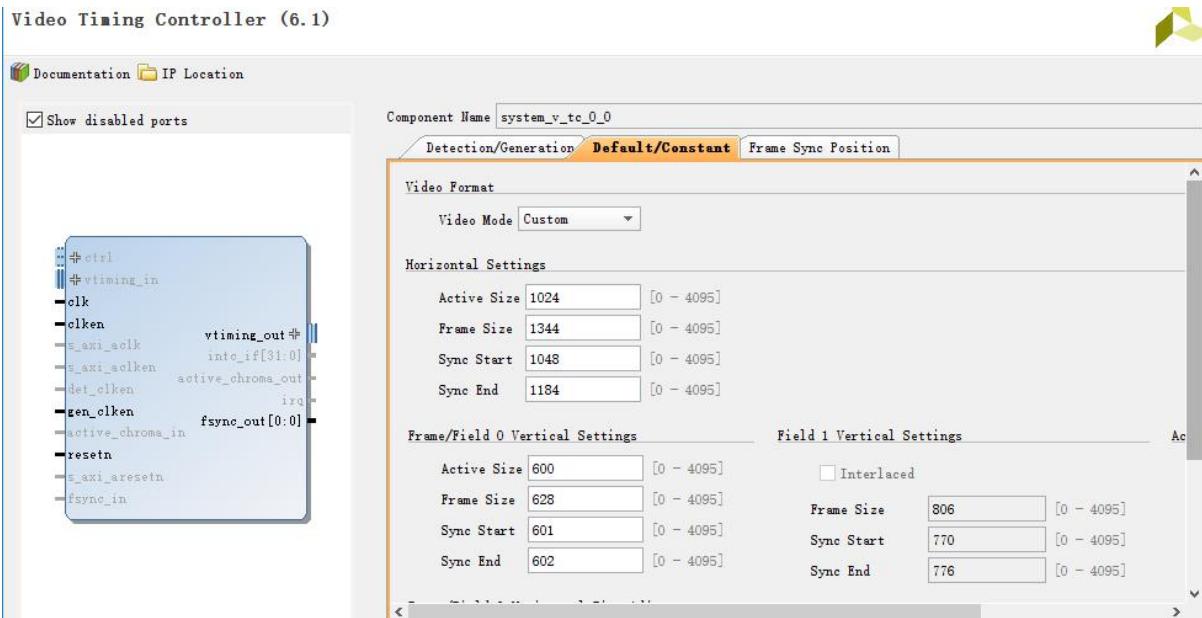
Step11: 双击 PLL 时钟 IP MIZ701N PLL 时钟设置



MIZ702/MIZ702N PLL 时钟设置



Step13: 修改 VTC 显示时序发生 IP 参数符合 1024X600 分辨率



1.4 PS 设置

1.4.1 PS SDK 测试显示器输出

新建 Display_VMDA_Test 空的工程，为了测试在裸机下图形系统显示正确，编写 SDK 测试代码 main.c 函数以及其他必要函数。

```
/*
 *南京米联电子科技有限公司
 *www.milinker.com
 *www.osrc.cn
 *test display
 Copyright (c) 2009-2012 Xilinx, Inc. All rights reserved.
 */

#include "xaxivdma.h"
#include "xaxivdma_i.h"
#include "sleep.h"

#define DDR_BASEADDR      0x00000000
#define VDMA_BASEADDR    XPAR_AXI_VDMA_0_BASEADDR
#define H_STRIDE          1024
#define H_ACTIVE           1024
#define V_ACTIVE            600
#define pi                 3.14159265358
#define COUNTS_PER_SECOND  (XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ)/64
```

```
#define VIDEO_LENGTH (H_STRIDE*V_ACTIVE)
#define VIDEO_BASEADDR0 DDR_BASEADDR + 0x2000000
#define VIDEO_BASEADDR1 DDR_BASEADDR + 0x3000000
#define VIDEO_BASEADDR2 DDR_BASEADDR + 0x4000000

u32 *BufferPtr[3];

unsigned int srcBuffer = (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x1000000);
int run_triple_frame_buffer(XAxiVdma* InstancePtr, int DeviceId, int hsize,
    int vsize, int buf_base_addr, int number_frame_count,
    int enable_frm_cnt_intr);

//函数声明
void Xil_DCacheFlush(void);
// 所有数据格式 为 RGBA， 低位的透明度暂不起作用
extern const unsigned char gImage_beauty[1729536];
extern const unsigned char gImage_mm[1228800];
extern const unsigned char gImage_miz702[600000];
extern const unsigned char gImage_miz702_rgba[600000];

void show_img(u32 x, u32 y, u32 disp_base_addr, const unsigned char * addr, u32 size_x, u32 size_y)
{
    //计算图片 左上角坐标
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    u32 start_addr=disp_base_addr;
    start_addr = disp_base_addr + 4*x + y*4*H_STRIDE;
    for(j=0;j<size_y;j++)
    {
        for(i=0;i<size_x;i++)
        {
            b = *(addr+(i+j*size_x)*4+1);
            g = *(addr+(i+j*size_x)*4+2);
            r = *(addr+(i+j*size_x)*4+3);
            Xil_Out32((start_addr+(i+j*H_STRIDE)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
        }
    }
    Xil_DCacheFlush();
}

int main(void)
{
```

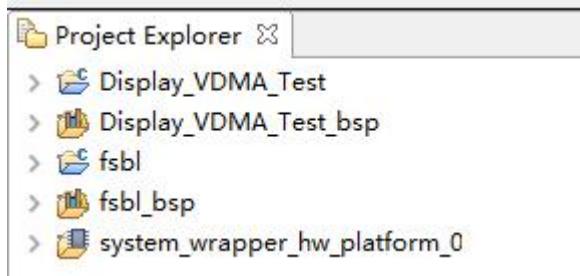
```
u32 i;
//Xil_DCacheFlush();
xil_printf("Starting the first VDMA \n\r");
//VDMA configurateAXI VDMA0
/****************往 DDR 写数据设置*******/
//Xil_Out32((VDMA_BASEADDR + 0x030), 0x00000003); // enable circular mode
//Xil_Out32((VDMA_BASEADDR + 0x0AC), VIDEO_BASEADDR0); // start address
//Xil_Out32((VDMA_BASEADDR + 0x0B0), VIDEO_BASEADDR1); // start address
//Xil_Out32((VDMA_BASEADDR + 0x0B4), VIDEO_BASEADDR2); // start address
//Xil_Out32((VDMA_BASEADDR + 0x0A8), (H_STRIDE*4)); // h offset (640 * 4) bytes
//Xil_Out32((VDMA_BASEADDR + 0x0A4), (H_ACTIVE*4)); // h size (640 * 4) bytes
//Xil_Out32((VDMA_BASEADDR + 0x0A0), V_ACTIVE); // v size (480)
/****************从 DDR 读数据设置*******/
Xil_Out32((VDMA_BASEADDR + 0x000), 0x00000003); // enable circular mode
Xil_Out32((VDMA_BASEADDR + 0x05c), VIDEO_BASEADDR0); // start address
Xil_Out32((VDMA_BASEADDR + 0x060), VIDEO_BASEADDR0); // start address
Xil_Out32((VDMA_BASEADDR + 0x064), VIDEO_BASEADDR0); // start address
Xil_Out32((VDMA_BASEADDR + 0x058), (H_STRIDE*4)); // h offset (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR + 0x054), (H_ACTIVE*4)); // h size (640 * 4) bytes
Xil_Out32((VDMA_BASEADDR + 0x050), V_ACTIVE); // v size (480)
for(i=0;i<614400;i++)
{
    Xil_Out32(VIDEO_BASEADDR0+i,0);
}
while(1)
{
    show_img(0,0,VIDEO_BASEADDR0,&gImage_beauty[0],563,600);
    sleep(5);
    show_img(0,0,VIDEO_BASEADDR0,&gImage_miz702_rgba[0],375,400);
    sleep(5);
}

return 0;
}
```

1.4.2 测试效果 缺图

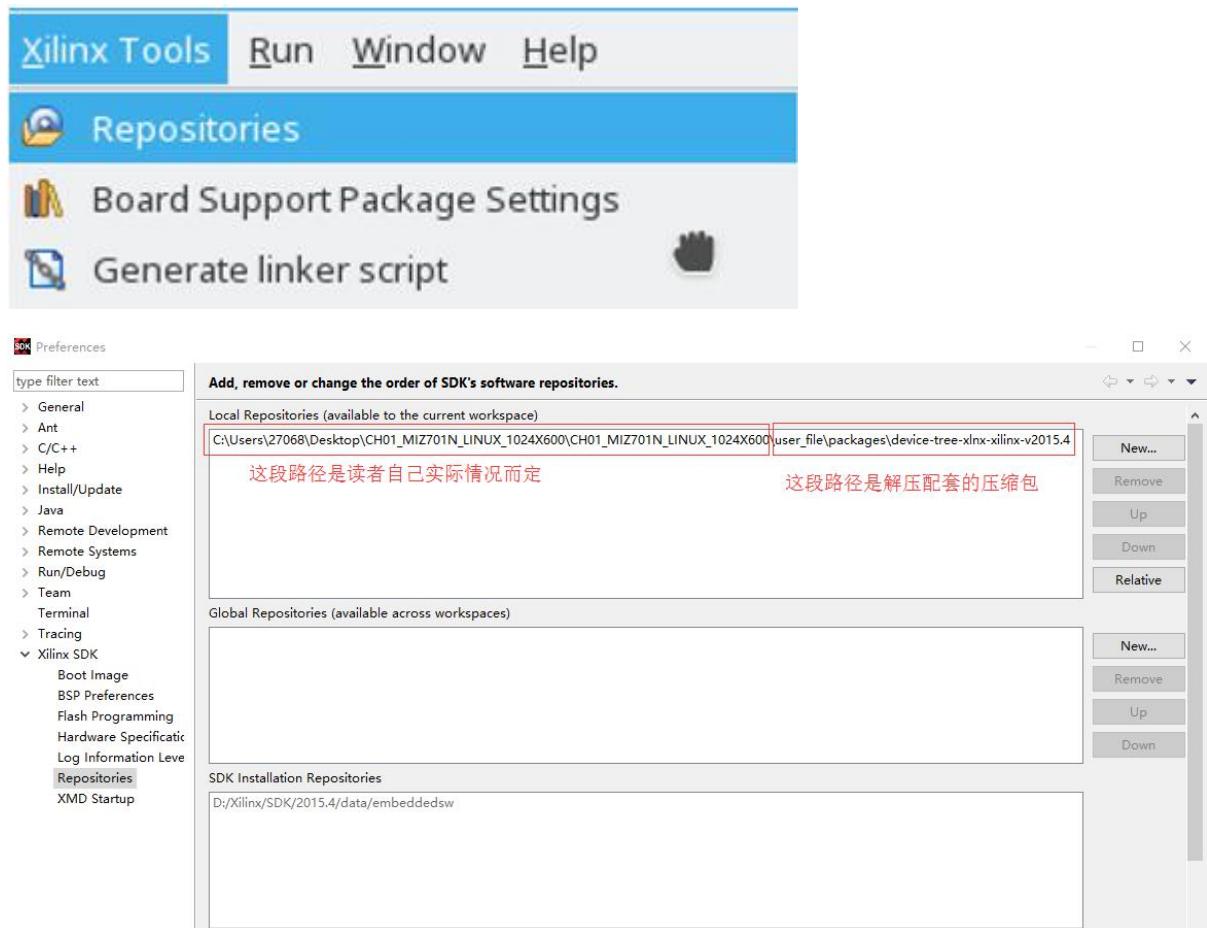
1.4.3 新建 FSBL 工程

产生的 fsbl.elf 后面将用于参数 BOOT.BIN 文件

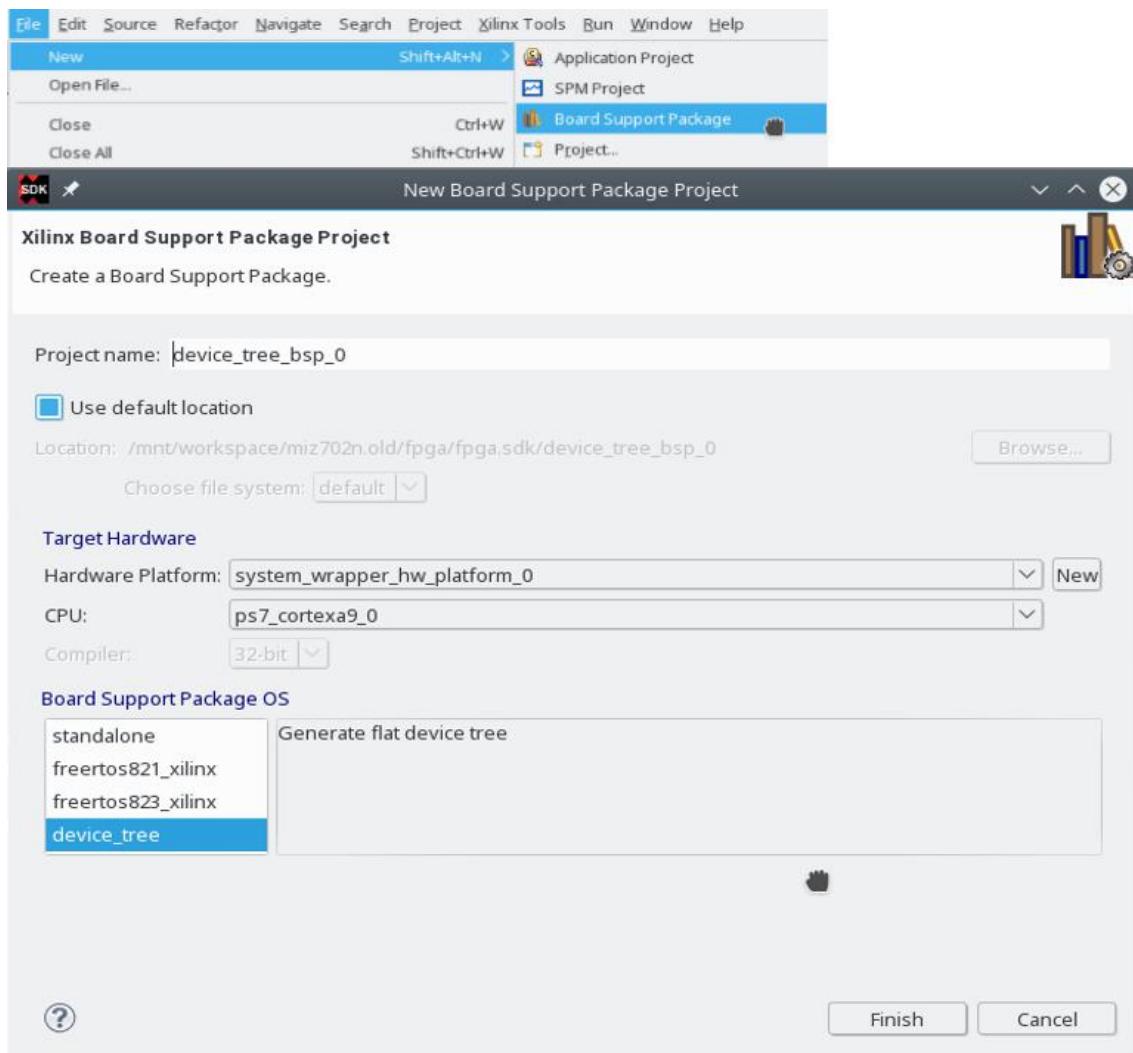


1.4.4 产生设备树

Step1: 首先解压 device-tree-xlnx-xilinx-v2015.4.tar.gz , 然后打开 Xilinx Tools → Repositories 将刚才解压的目录包含进来。



Step2: 打开 File→New→Board Support Package 创建,其它弹出窗口按默认设置即可



Step3: 在该工程中，zynq-7000.dtsi 文件是 Xilinx 提供给所有 zynq-7000 开发板使用的，只所有的 status 都 设置为” disabled”，而 system.dts 里根据板子上的具体实现，修改设备树使其可以正常工作。pl.dtsi 文件 是 FPGA 里使用的 IP 对应的设备树。

1.5 编译 u-boot、kernel、设备树和文件系统

1.5.1 批处理文件

这里使用 xilinx 在 github 上提供的 u-boot 和 kernel 源码，在 Wiki 上提供的文件系统（当然，也可以直接使用 buildroot 自己匹配根文件系统，该方法简单快捷，不用再去解决什么依赖问题，但好像国内很少有人这么干，他们都把 busybox 等模块一个个搭建起来）。1、配置 uboot 和编译 uboot 自带的工具，为编译 kernel 提供 mkimage 工具的支持。注意，由于编译新版本的 uboot 需要 openssl 和 dtc 的支持，这里呢，在系统里已经安装好了 openssl，而 dtc 则利用 kernel 里自带的，所以这一步做完，我们将直接编译内核，等编译完内核后再来编译 uboot。注意：cfg_bootloader.sh 可以在任意目录下使用，而 make tools 只能在 bootloader 目录

下使用，bootloader 就是我们存放 uboot 源码的地方。（注意：这部分仅在恢复配置文件至 Xilinx 提供的配置时用，在自己修改完配置之后，除了需要恢复配置文件外，请谨慎使用这两条命令）。

Step1:运行 setup_env.sh 批处理文件(管理员模式下的密码为 root)

```
osrc@osrc-virtual-machine:~$ sudo su
[sudo] password for osrc:
root@osrc-virtual-machine:/home/osrc# cd /mnt/workspace/linux/scripts
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# ls
cfg_bootloader.sh  get_xilinx_sources.sh  mk_bootloader.sh  setup_env.sh
cfg_kernel.sh      install_linaro_image.sh  mk_kernel.sh    vivado.jou
echo_color.sh      install_sd_image.sh    mk_sd_image.sh  vivado.log
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# source setup_env.sh
```

Step2:运行 cfg_bootloader.sh 批处理文件

```
root@osrc-virtual-machine:/mnt/workspace/linux# cd ./bootloader
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader# cfg_bootloader.sh
Configure U-Boot on the /mnt/workspace/linux/bootloader
make: Entering directory '/mnt/workspace/linux/bootloader'
#
# configuration written to .config
#
make: Leaving directory '/mnt/workspace/linux/bootloader'
U-Boot - configure Done.
```

Step2:运行 make tools 处理命令（执行的是清理工作，重置配置时候使用）

```
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader# make tools
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK  include/config.h
  GEN  include/autoconf.mk
  GEN  include/autoconf.mk.dep
  GEN  spl/include/autoconf.mk
  CHK  include/config/uboot.release
  CHK  include/generated/version autogenerated.h
  CHK  include/generated/timestamp autogenerated.h
  UPD  include/generated/timestamp autogenerated.h
  CHK  include/generated/generic-asm-offsets.h
  CHK  include/generated/asm-offsets.h
  HOSTCC tools/mkenvimage.o
  HOSTLD tools/mkenvimage
  HOSTCC tools/fit_image.o
  HOSTCC tools/image-host.o
  HOSTCC tools/dumpimage.o
  HOSTLD tools/dumpimage
  HOSTCC tools/mkimage.o
  HOSTLD tools/mkimage
  HOSTLD tools/fit_info
  HOSTLD tools/fit_check_sign
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader#
```

Step3:配置内核(注意：这部分仅在恢复配置文件至 Xilinx 提供的配置时用，在自己修改完配置之后，除了需要恢复配置文件外，请谨慎使用这条命令)

```
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader# cfg_kernel.sh
Configure Linux kernel on the /mnt/workspace/linux/kernel
make: Entering directory '/mnt/workspace/linux/kernel'
#
# configuration written to .config
#
make: Leaving directory '/mnt/workspace/linux/kernel'
Linux Kernel - configure Done.
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader#
```

1.5.2 修改设备树

Step1:由于各种开发板可以参考 zedboard 的配置设备参数，因此我们可以修改已经存在的相关文件，来满足我们的自定义需求 打开 /mnt/workspace/linux/kernel/arch/arm/boot/dts/zynq-zed.dts

Step2:修改 bootargs 信息 启动参数 bootargs 是传递给 kernel 的参数。Console 是一个输出系统管理信息的文本输出设备，这些信息来自于内核，系统启动和系统用户，其中 console=ttyPS0, 115200 用来设置串口作为输出终端设备，是这些信息可以通过串口在远程的终端上显示。而 console=tty0 则是设置显示器作为输出终端。

如果想屏幕永不休眠，则在启动参数 bootargs 中增加 consoleblank=0，网络上大部分是讲修改内核源码，但我觉得这种方法才是最方便，且符合一个内核适合多种产品的思想。其它参数读者应该也知道是什么意思，这里也就不多讲了。

```
chosen {
    /* bootargs = ""; */
    bootargs = "console=ttyPS0,115200 console=tty0 consoleblank=0 root=/dev/ram rw earlyprintk";
    stdout-path = "serial0:115200n8";
};
```

Step3:添加 vdma 和 framebuffer 设备树节点，注意：这里是添加到根节点。笔者觉得，arm linux 引进设备树，比以前版本的做法好很多，只是驱动里需要设备树提供哪些参数，在 Linux 文档里没有很好的说明，或者是驱动里所需要的参数改变了，而文档又没有及时更新，这部分是 linux 的不足之处。所以，开发 Linux 过程中，最好还是以源码为中心。设备树这部分是参考之前在 SDK 里产生的设备树和内核里相关文档，在阅读 vdma 等驱动源码后，修改而来的，并不是说 SDK 里产生的设备树直接搬来就可以正常使用。

```

/ {
    amba_pl: amba_pl {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";
        ranges ;
        /* 液晶显示屏 */
        /* framebuffer架构驱动 */
        axi_vdma_0: dma@43000000 {
            compatible = "xlnx,axi-vdma-1.00.a";
            #dma-cells = <1>;
            reg = <0x43000000 0x10000>;
            /* dma-ranges = <0x00000000 0x00000000 0x40000000>; */
            interrupt-parent = <&intc>;
            /* interrupts = <0 33 4>; */
            xlnx,num-fstores = <0x1>;
            xlnx,flush-fsync = <0x2>;
            xlnx,addrwidth = <0x20>;
            clocks = <&clkc 0>;
            clock-names = "s_axi_lite_aclk";
            dma-channel@43000000 {
                compatible = "xlnx,axi-vdma-mm2s-channel";
                interrupts = <0 33 4>;
                xlnx,datawidth = <0x20>;
                xlnx,device-id = <0x0>;
            } ;
        } ;
        axi_vdma_1cd {
            compatible = "topic,vdma-fb";
            dmas = <&axi_vdma_0 0>;
            dma-names = "axivdma";
        } ;
    } ;
}

```

Step4:修改 SD 卡和 emmc 设备树节点

这部分直接将之前在 SDK 里产生的设备树复制过来就可以正常使用。总的来说，对于 PS 的外设，其 硬件是固定的，驱动也基本不会有太大的变化，故基本上是可以直接搬来就可以用，但也有例外，如果 USB，SDK 里并不知道你要当 host 或者 otg 来使用，所以需要做些小修改。而对于 PL 中使用的 IP，其 硬件和驱动经常有变化，SDK 里产生的设备树不能直接拿来使用。

```

&sdhci0 {
    status = "okay";
    xlnx,has-cd = <0x1>;
    xlnx,has-power = <0x0>;
    xlnx,has-wp = <0x1>;
};

&sdhci1 {
    status = "okay";
    xlnx,has-cd = <0x1>;
    xlnx,has-power = <0x0>;
    xlnx,has-wp = <0x1>;
};

```

1.5.3 添加 framebuffer 驱动

Step1: 把 vdmafb.c 文件复制到 /mnt/workspace/linux/kernel/drivers/video/fbdev 目录下, 对于驱动这部分, 主要还是要看源码, 注意, 该驱动仅支持 640x480 分辨率, 若需要支持其它分辨率, 需要修改和调试相关源码。

Step2: 打开 /mnt/workspace/linux/kernel/drivers/video/fbdev 目录下的 Makefile 文件, 当然也可以用其它文本编辑器。

找到 CONFIG_FB_XILINX, 在其下方添加 obj-\$(CONFIG_FB_VDMA) += vdmafb.o。读者可以直接复制笔者提供的开发包中的修改好的文件。

```
obj-$(CONFIG_FB_XILINX)          += xilinxfb.o
obj-$(CONFIG_FB_VDMA)           += vdmafb.o
```

Step3: 打开 /mnt/workspace/linux/kernel/drivers/video/fbdev 目录下的 Kconfig 文件:

找到 FB_XILINX, 在其下方添加以下信息。读者可以直接复制笔者提供的开发包中的修改好的文件。

```
config FB_XILINX
    tristate "Xilinx frame buffer support"
    depends on FB && (XILINX_VIRTEX || MICROBLAZE || ARCH_ZYNQ || ARCH_ZYNQMP)
    select FB_CFB_FILLRECT
    select FB_CFB_COPYAREA
    select FB_CFB_IMAGEBLIT
    ---help---
        Include support for the Xilinx ML300/ML403 reference design
        framebuffer. ML300 carries a 640*480 LCD display on the board,
        ML403 uses a standard DB15 VGA connector.

config FB_VDMA
    tristate "VDMA frame buffer support"
    depends on FB && ARCH_ZYNQ
    select FB_CFB_FILLRECT
    select FB_CFB_COPYAREA
    select FB_CFB_IMAGEBLIT
    select XILINX_AXIVDMA
    ---help---
        Include support for the VDMA LCD reference design framebuffer.
```

Step4: 打开 /mnt/workspace/linux/kernel/drivers 目录下的 Makefile 文件, 找到 obj-y += video/ 读者可以直接复制笔者提供的开发包中的修改好的文件。

```
# GPIO must come after pinctrl as gpios may need to mux pins etc
obj-$(CONFIG_PINCTRL)          += pinctrl/
obj-y                         += gpio/
obj-y                         += pwm/
obj-$(CONFIG_PCI)              += pci/
obj-$(CONFIG_PARISC)            += parisc/
obj-$(CONFIG_RAPIDIO)           += rapidio/
obj-y                         += video/
obj-y                         += idle/
```

把它剪切并粘贴到

```
# iommu/ comes before gpu as gpu are using iommu controllers
obj-$(CONFIG_IOMMU_SUPPORT)      += iommu/
# gpu/ comes after char for AGP vs DRM startup and after iommu
obj-y                           += gpu/
obj-$(CONFIG_CONNECTOR)          += connector/
obj-y                           += video/
```

Step5:回到 kernel 目录下，执行 make menuconfig，配置 kernel。

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# make menuconfig
scripts/kconfig/mconf Kconfig
```

按向下方向键找到 Device Drivers，按回车键进入

```
-** Patch physical to virtual translations at runtime
  General setup --->
  [*] Enable loadable module support --->
  [*] Enable the block layer --->
    System Type --->
    Bus support --->
    Kernel Features --->
    Boot options --->
    CPU Power Management --->
    Floating point emulation --->
    Userspace binary formats --->
    Power management options --->
  [*] Networking support --->
  [ ] Device Drivers --->
    Firmware Drivers --->
    File systems --->
    Kernel hacking --->
    Security options --->
  -** Cryptographic API --->
    Library routines --->
  -** Virtualization --->
```

按向下方向键找到 Graphics support，按回车键进入

```
-** Board level reset or power off --->
  [ ] Adaptive Voltage Scaling class support ----
  <*> Hardware Monitoring support --->
  <*> Generic Thermal sysfs driver --->
  [*] Watchdog Timer Support --->
    Sonics Silicon Backplane --->
    Broadcom specific AMBA --->
    Multifunction device drivers --->
  [*] Voltage and Current Regulator Support --->
  <*> Multimedia support --->
  [ ] Graphics support --->
    <*> Sound card support --->
    HID support --->
  [*] USB support --->
  < > Ultra Wideband devices ----
```

按向下方向键找到 Bootup logo，按空格键选择

```

Display interface Bridges
< > DRM Support for STMicroelectronics SoC stiH41x Series
<*> Xilinx DRM
--> Xilinx DRM Display Port Driver
--> Xilinx DRM Display Port Subsystem Driver
< > Xylon DRM
  Frame buffer Devices --->
[ ] Backlight & LCD device support ----
  Console display driver support --->
[*] Bootup logo --->

```

按向上方向键找到 Frame buffer Devices，按回车键选择

```

--> Xilinx DRM Display Port Driver
--> Xilinx DRM Display Port Subsystem Driver
< > Xylon DRM
[*] Frame buffer Devices --->
[ ] Backlight & LCD device support ----
  Console display driver support --->
[*] Bootup logo --->

```

至此已经完成 linux kernel 的配置，按向左方向键至<Exit>，再按回车键返回上一级菜单



直到看到以下窗口，按<Yes>保存。



1.5.4 执行 mk_kernel.sh 编译内核

```

root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_kernel.sh
Build kernel and drivers on the /mnt/workspace/linux/kernel
make: Entering directory '/mnt/workspace/linux/kernel'
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK    include/config/kernel.release
  CHK    include/generated/uapi/linux/version.h
  CHK    include/generated/utsrelease.h
make[1]: 'include/generated/mach-types.h' is up to date.
  CHK    include/generated/timeconst.h
  CHK    include/generated/bounds.h
  CHK    include/generated/asm-offsets.h
  CALL   scripts/checksyscalls.sh
  CHK    include/generated/compile.h
  GZIP   kernel/config_data.gz
  CHK    kernel/config_data.h
  UPPD   kernel/config_data.h

```

1.5.5 执行 mk_bootloader.sh 编译 uboot

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_bootloader.sh
Build U-Boot on the /mnt/workspace/linux/bootloader
make: Entering directory '/mnt/workspace/linux/bootloader'
CHK      include/config/uboot.release
CHK      include/generated/timestamp autogenerated.h
UPD      include/generated/timestamp autogenerated.h
CHK      include/generated/version autogenerated.h
CHK      include/generated/asm-offsets.h
CHK      include/generated/generic-asm-offsets.h
HOSTCC  tools/mkenvimage.o
HOSTCC  tools/image-host.o
HOSTCC  tools/fit_image.o
HOSTCC  tools/dumpimage.o
HOSTCC  tools/mkimage.o
HOSTLD  tools/mkenvimage
HOSTLD  tools/fit_info
```

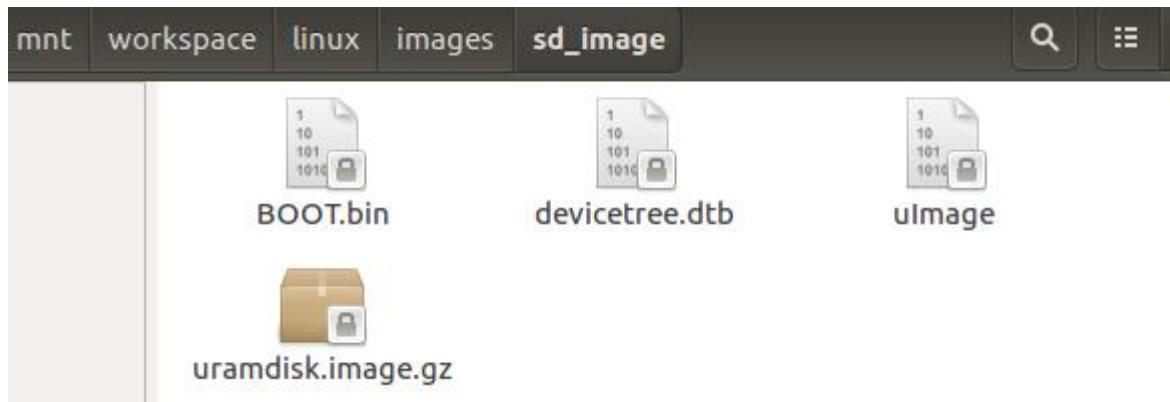
1.5.6 制作 UBOOT.BIN

Step1: 把 system_wrapper.bit 和 fsbl.elf 复制到 output/target/ 目录下。

Step2: 执行 mk_sd_image.sh 打包从 SD 启动所需要的文件

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_sd_image.sh
Remove older files...
```

Step3: 至此系统部分已经完成, 复制 linux/image/sd_images 文件夹下文件到 TF 卡测试移植好的 LINUX 系统。



1.6 EMMC 8GB 内存测试(MIZ702 不支持)

Step1: 从系统的启动信息可以看到, 系统已经发现 mmc0 (即 SD 卡) 和 mmc1 (即 emmc), 而且列出了 SD 卡为 7.41GB, 而 emmc 为 7.20GB

```

sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhci-pltfm: SDHCI platform and OF driver helper
mmc0: SDHCI controller on e0100000.sdhci [e0100000.sdhci] using DMA
mmc1: SDHCI controller on e0101000.sdhci [e0101000.sdhci] using DMA
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
mmc0: new high speed SDHC card at address 59b4
NET: Registered protocol family 10
mmcblk0: mmc0:59b4 USD00 7.41 GiB
sit: IPv6 over IPv4 tunneling driver
  mmcblk0: p1
NET: Registered protocol family 17
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
Registering SWP/SWPB emulation handler
hctosys: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
RAMDISK: gzip image found at block 0
mmc1: new high speed MMC card at address 0001
mmcblk1: mmc1:0001 P1XXXX 7.20 GiB
mmcblk1boot0: mmc1:0001 P1XXXX partition 1 2.00 MiB
mmcblk1boot1: mmc1:0001 P1XXXX partition 2 2.00 MiB
mmcblk1rpmb: mmc1:0001 P1XXXX partition 3 128 KiB

```

Step2:给 eMMC 分区 当然，如果只分为一个分区的话，其它也可以不分区。在命令行下运行 fdisk /dev/mmcblk1 来对 emmc 进行分区。这里需要注意，确认有没有 SD 卡插入，也就是说确认当前 eMMC 是/dev/mmcblk1 还是 /dev/mmcblk0，还有对于有多个分区的，可能存在/dev/mmcblk0p1、/dev/mmcblk0p2 等等。

```

zynq> fdisk /dev/mmcblk1
The number of cylinders for this disk is set to 236032.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
  (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): m
Command Action
a   toggle a bootable flag
b   edit bsd disklabel
c   toggle the dos compatibility flag
d   delete a partition
l   list known partition types
n   add a new partition
o   create a new empty DOS partition table
p   print the partition table
q   quit without saving changes
s   create a new empty Sun disklabel
t   change a partition's system id
u   change display/entry units
v   verify the partition table
w   write table to disk and exit
x   extra functionality (experts only)

Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-236032, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-236032, default 236032): Using default value 236032

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table
mmcblk1: p1

```

Step3: 将分区格式化为 ext2 格式 能格式化为什么格式, 如 ext2、ext3、nfts、fat32, 这些都跟系统的定制有关

```
zynq> mkfs.ext2 /dev/mmcblk1p1
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
472352 inodes, 1888254 blocks
94412 blocks (5%) reserved for the super user
First data block=0
Maximum filesystem blocks=4194304
58 block groups
32768 blocks per group, 32768 fragments per group
3144 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632
random: nonblocking pool is initialized
zynq> ls /mnt/
zynq> mount /dev/mmcblk1p1 /mnt/
EXT4-fs (mmcblk1p1): couldn't mount as ext3 due to feature incompatibilities
EXT4-fs (mmcblk1p1): filesystem too large to mount safely on this system
zynq> ls /mnt/
lost+found
zynq>
```

Step4: 测试 emmc 的性能

在 Linux 下, 既可以使用 dd 命令来低格 U 盘等, 也可以用来复制文件 (类似于 windows 下的 ghost 功能), 当然也可以用来测试硬盘等的性能, 虽然没有专业软件的测试得准确, 但用来对比性能已经足够了。对于/dev/zero 和/dev/null 两个设备的说明, 可以百度一下

Step4.1 写性能

下面使用 dd 命令将从/dev/zero 设备中产生一个 1GB 的文件写入到 emmc:

```
zynq> dd if=/dev/zero of=/mnt/test.img bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.0GB) copied, 75.173231 seconds, 13.6MB/s
```

Step4.2 读性能

下面使用 dd 命令将 1GB 的文件写入到/dev/null 设备中:

```
zynq> dd if=/mnt/test.img of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.0GB) copied, 52.177487 seconds, 19.6MB/s
```

Step4.3 读写性能

下面使用 dd 命令将 emmc 中的一个 1GB 的文件写入到 emmc 的另外一个文件

```
zynq> dd if=/mnt/test.img of=/mnt/test.copy.img
2097152+0 records in
2097152+0 records out
1073741824 bytes (1.0GB) copied, 130.697851 seconds, 7.8MB/s
```

1.7 测试 framebuffer

Step1: 将虚拟机/mnt/workspace/linux/framebuffer 目录下的测试程序(源代码也

在该目录里) 复制到 SD 卡上, 然后给开发板上电。在 Windows 下打开串口终端 putty 软件。

Step2: 切换到 sd 卡目录下 (比如 mount /dev/mmcblk0p1 /mnt, cd /mnt), 然后运行 ./framebuffer, 在串口终端 会显示 以下信息。

```
zyng> ./framebuffer
Fixed screen info:
  id: vdma-fb
  smem_start: 0x1f100000
  smem_len: 1228800
  type: 0
  type_aux: 0
  visual: 2
  xpanstep: 0
  ypanstep: 0
  ywrapstep: 0
  line_length: 2560
  mmio_start: 0x0
  mmio_len: 0
  accel: 0

Variable screen info:
  xres: 640
  yres: 480
  xres_virtual: 640
  yres_virtual: 480
  yoffset: 0
  xoffset: 0
  bits_per_pixel: 32
  grayscale: 0
  red: offset: 16, length: 8, msb_right: 0
  green: offset: 8, length: 8, msb_right: 0
  blue: offset: 0, length: 8, msb_right: 0
  transp: offset: 24, length: 8, msb_right: 0
  nonstd: 0
  activate: 0
  height: 0
  width: 0
  accel_flags: 0x0
  pixclock: 39721
  left_margin: 0
  right_margin: 0
  upper_margin: 0
  lower_margin: 0
  hsync_len: 0
  vsync_len: 0
  sync: 0
  vmode: 0

Frame Buffer Performance test...
Average: 2186 usecs
Bandwidth: 536.082 MByte/Sec
Max. FPS: 457.457 fps

Will draw 3 rectangles on the screen,
they should be colored red, green and blue (in that order).
Done.
```

Step3: 在屏幕上会显示以下信息

S04_CH02_工程移植 ubuntu 并一键制作启动盘

2.1 概述

2.2 搭建硬件系统

本章硬件工程还是使用《S04_CH01_搭建工程移植 LINUX/测试 EMMC/VGA》所搭建的 VIVADO 工程。由于使用批处理命令，本章操作起来十分简单，但是批处理命令的源码大家可以好好分析。笔者会在后期的课程中专门把所有用到的常用批处理命令总结分析一下。

2.3 一键制作

Step1: 输入 su 切换到 root 用户

```
osrc@osrc-virtual-machine:~$ sudo su
[sudo] password for osrc:
root@osrc-virtual-machine:/home/osrc#
```

Step2: 输入 cd /mnt/workspace/linux/scripts 切换到 /mnt/workspace/linux/scripts 目录下，执行 source setup_env.sh（注意：source 和 “.” 是相同的效 果）初始化环境变量

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# source setup_env.sh
```

Step3: 执行 mk_kernel.sh 编译内核

Step4: 执行 mk_sd_image.sh 生成基于 ramdisk 文件系统的 SD 镜像，当然，这里主要使用 linaro 文件系统，只会使用 到其中的部分文件而已

Step5: 插入 SD

Step6: 执行 install_linaro_image.sh 即可删除 SD 卡的所有分区，然后重新分区、格式化，并烧录 linaro 系统，执行完毕 后，拔下 SD 卡至 开发板上，设置启动模式为从 SD 启动即可。

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# install_linaro_image.sh
硬盘分区信息
Disk /dev/sda: 20 GiB, 21474836480 bytes, 41943040 sectors
Disk /dev/sdb: 40 GiB, 42949672960 bytes, 83886080 sectors
请输入需要分区的磁盘{hdX|sdX}或者输入q退出此脚本: sdc
即将格式化并分区磁盘sdc，请输入(y/n)确认?y
df: /mnt/hgfs: Protocol error
1+0 records in
1+0 records out
512 bytes copied, 0.00555828 s, 92.1 kB/s
已经清除磁盘sdc上的所有分区
磁盘sdc分区成功
df: /mnt/hgfs: Protocol error
正在格式化启动分区...
正在格式化根文件系统...
完成磁盘sdc的分区操作
挂载/dev/sdc1至/mnt/workspace/linux/output/linaro/boot目录下
挂载/dev/sdc2至/mnt/workspace/linux/output/linaro/rootfs目录下
正在制作Eboot...
正在制作文件系统...
已经烧录Linaro系统至SD卡...
root@osrc-virtual-machine:/mnt/workspace/linux/scripts#
```

2.4 运行结果

S04_CH03_QSPI 烧写 LINUX 系统

3.1 概述

3.2 搭建硬件系统

本章硬件工程还是使用《S04_CH01_搭建工程移植 LINUX/测试 EMMC/VGA》所搭建的 VIVADO 工程。

3.3 修改内核文件

Step1:切换到管理员模式

```
osrc@osrc-virtual-machine:~$ sudo su  
[sudo] password for osrc:  
root@osrc-virtual-machine:/home/osrc# █
```

Step2:切换到 scripts 目录下，执行 source setup_env.sh（注意 source 和 “.” 是一致的），并将 scripts.tar.gz 中的两个脚本放到 scripts 目录下，通过这两个脚本可以打包 QSPI 镜像和将 QSPI 镜像烧录至 QSPI 中。本项目是基于上一个项目工程。

```
osrc@osrc-virtual-machine:~$ sudo su  
[sudo] password for osrc:  
root@osrc-virtual-machine:/home/osrc# cd /mnt/workspace/linux/scripts  
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# . setup_env.sh  
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# █
```

Step4:切换到 bootloader 源码目录，打开 include/configs/zynq-common.h 文件

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# cd ..  
root@osrc-virtual-machine:/mnt/workspace/linux# cd bootloader  
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader# cd include/configs  
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader/include/configs# vi zynq-common.h  
root@osrc-virtual-machine:/mnt/workspace/linux/bootloader/include/configs#
```

Step5:修改以下内核、设备树及内存文件系统在 QSPI 的起始位置如 Step6 所示：

```

/* Default environment */
#ifndef CONFIG_EXTRA_ENV_SETTINGS
#define CONFIG_EXTRA_ENV_SETTINGS
"ethaddr=00:0a:35:00:01:22\0" \
"kernel_image=uImage\0" \
"kernel_load_address=0x2080000\0" \
"ramdisk_image=uramdisk.image.gz\0" \
"ramdisk_load_address=0x4000000\0" \
"devicetree_image=devicetree.dtb\0" \
"devicetree_load_address=0x2000000\0" \
"bitstream_image=system.bit.bin\0" \
"boot_image=BOOT.bin\0" \
"loadbit_addr=0x1000000\0" \
"loadbootenv_addr=0x2000000\0" \
"kernel_size=0x500000\0" \
"devicetree_size=0x20000\0" \
"ramdisk_size=0x5E0000\0" \
"boot_size=0xF00000\0" \
"fdt_high=0x2000000\0" \
"initrd_high=0x2000000\0" \
"bootenv=uEnv.txt\0" \
"loadbootenv=load mmc 0 ${loadbootenv_addr} ${bootenv}\0" \
"importbootenv=echo Importing environment from SD ...; " \
"    env import -t ${loadbootenv_addr} ${filesize}\0" \
"sd_uEnvtxt_existence_test=test -e mmc 0 /uEnv.txt\0" \
"preboot;if test $modeboot = sdboot && env run sd_uEnvtxt_existence_test; " \
"    then if env run loadbootenv; " \
"        then env run importbootenv; " \
"            fi; " \
"        fi; \0" \
"mmc_loadbit=echo Loading bitstream from SD/MMC/eMMC to RAM.. && " \
"    mmcinfo && " \
"    load mmc 0 ${loadbit_addr} ${bitstream_image} && " \
"    fpga load 0 ${loadbit_addr} ${filesize}\0" \
"norboot=echo Copying Linux from NOR flash to RAM... && " \
"    cp.b 0xE2100000 ${kernel_load_address} ${kernel_size} && " \
"    cp.b 0xE2600000 ${devicetree_load_address} ${devicetree_size} && " \
"    echo Copying ramdisk... && " \
"    cp.b 0xE2620000 ${ramdisk_load_address} ${ramdisk_size} && " \
"    bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}\0" \
"qspiboot=echo Copying Linux from QSPI flash to RAM... && " \
"    sf probe 0 0 0 && " \
"    sf read ${kernel_load_address} 0x100000 ${kernel_size} && " \
"    sf read ${devicetree_load_address} 0x600000 ${devicetree_size} && " \
"    echo Copying ramdisk... && " \
"    sf read ${ramdisk_load_address} 0x620000 ${ramdisk_size} && " \
"    bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}\0" \
"uenvboot=" \
"if run loadbootenv; then "

```

Step6:QSPI 的起始位置放置 FSBL、FPGA 比特流和 uboot，文件大小大概是 5MB，所以内核的存放位置从 0x500000 开始，而这里同样给内核留 5MB，当然，如果你的内核增加更多配置或减少配置，可以适当修改，这样的话，设备树就需要从 0xA00000 位置开始存放，然后留给设备树的空间大概是 2K，即从 0xA20000 开始存储文件系统，这样 ramdisk 的大小就应该是 0x15E0000，当然，内核默认支持 8MB

```

/* Default environment */
#ifndef CONFIG_EXTRA_ENV_SETTINGS
#define CONFIG_EXTRA_ENV_SETTINGS      \
    "ethaddr=00:0a:35:00:01:22\0"     \
    "kernel_image=uImage\0"           \
    "kernel_load_address=0x2080000\0"  \
    "ramdisk_image=uramdisk.image.gz\0" \
    "ramdisk_load_address=0x4000000\0" \
    "devicetree_image=devicetree.dtb\0" \
    "devicetree_load_address=0x2000000\0" \
    "bitstream_image=system.bit.bin\0" \
    "boot_image=BOOT.bin\0"           \
    "loadbit_addr=0x100000\0"          \
    "loadbootenv_addr=0x2000000\0"     \
    "kernel_size=0x500000\0"           \
    "devicetree_size=0x20000\0"        \
    "ramdisk_size=0x15E0000\0"         \
    "boot_size=0xF000000\0"            \
    "fdt_high=0x20000000\0"           \
    "initrd_high=0x20000000\0"         \
    "bootenv=uEnv.txt\0"              \
    "loadbootenv=load mmc 0 ${loadbootenv_addr} ${bootenv}\0" \
    "importbootenv=echo Importing environment from SD ...; " \
        "env import -t ${loadbootenv_addr} ${filesize}\0" \
    "sd_uEnvtxt_existence_test=test -e mmc 0 /uEnv.txt\0" \
    "preboot;if test $modeboot = sdboot && env run sd_uEnvtxt_existence_test; " \
        "then if env run loadbootenv; " \
            "then env run importbootenv; " \
                "fi; " \
        "fi; \0" \
    "mmc_loadbit=echo Loading bitstream from SD/MMC/eMMC to RAM.. && " \
        "mmcinfo && " \
        "load mmc 0 ${loadbit_addr} ${bitstream_image} && " \
        "fpga load 0 ${loadbit_addr} ${filesize}\0" \
    "norboot=echo Copying Linux from NOR Flash to RAM... && " \
        "cp.b 0xE210000 ${kernel_load_address} ${kernel_size} && \n" \
        "cp.b 0xE260000 ${devicetree_load_address} ${devicetree_size} && " \
        "echo Copying ramdisk... && " \
        "cp.b 0xE262000 ${ramdisk_load_address} ${ramdisk_size} && " \
        "bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}\0" \
    "qspiboot=echo Copying Linux from QSPI flash to RAM... && " \
        "sf probe 0 0 0 && " \
        "sf read ${kernel_load_address} 0x500000 ${kernel_size} && " \
        "sf read ${devicetree_load_address} 0xA00000 ${devicetree_size} && " \
        "echo Copying ramdisk... && " \
        "sf read ${ramdisk_load_address} 0xA20000 ${ramdisk_size} && " \
        "bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}\0" \
    "uenvboot=" \
        "if run loadbootenv; then " \

```

Step7: 打开 zynq-zed.dts，找到 qspi 节点，这里把原有的分区删掉，当然，你也可以根据刚才对 QSPI 的分区，做对应的修改。可以出从 SD 启动或从 QSPI 启动 Linux，然后在系统里更新 QSPI 镜像。

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# vi arch/arm/boot/dts/zynq-zed.dts
```

```

&qspi {
    status = "okay";
    is-dual = <0>;
    num-cs = <1>;
    flash@0 {
        compatible = "n25q128a11";
        reg = <0x0>;
        spi-tx-bus-width = <1>;
        spi-rx-bus-width = <4>;
        spi-max-frequency = <50000000>;
        #address-cells = <1>;
        #size-cells = <1>;
        partition@qspi-fsbl-uboot {
            label = "qspi-fsbl-uboot";
            reg = <0x0 0x100000>;
        };
        partition@qspi-linux {
            label = "qspi-linux";
            reg = <0x100000 0x500000>;
        };
        partition@qspi-device-tree {
            label = "qspi-device-tree";
            reg = <0x600000 0x20000>;
        };
        partition@qspi-rootfs {
            label = "qspi-rootfs";
            reg = <0x620000 0x5E0000>;
        };
        partition@qspi-bitstream {
            label = "qspi-bitstream";
            reg = <0xC00000 0x400000>;
        };
    };
};

&qspi {
    status = "okay";
    is-dual = <0>;
    num-cs = <1>;
    flash@0 {
        compatible = "n25q128a11";
        reg = <0x0>;
        spi-tx-bus-width = <1>;
        spi-rx-bus-width = <4>;
        spi-max-frequency = <50000000>;
        #address-cells = <1>;
        #size-cells = <1>;
        partition@qspi-fsbl-uboot {
            label = "qspi-fsbl-uboot";
            reg = <0x0 0x100000>;
        };
        partition@qspi-linux {
            label = "qspi-linux";
            reg = <0x100000 0x500000>;
        };
        partition@qspi-device-tree {
            label = "qspi-device-tree";
            reg = <0x600000 0x20000>;
        };
        partition@qspi-rootfs {
            label = "qspi-rootfs";
            reg = <0x620000 0x5E0000>;
        };
        partition@qspi-bitstream {
            label = "qspi-bitstream";
            reg = <0xC00000 0x400000>;
        };
    };
};

```

3.3 编译内核及 uboot

Step1: 执行 mk_bootloader.sh 编译 bootloader 源码

```

root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_bootloader.sh
Build U-Boot on the /mnt/workspace/linux/bootloader
make: Entering directory '/mnt/workspace/linux/bootloader'
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK      include/config.h
  UPD      include/config.h
  GEN      include/autoconf.mk.dep
  GEN      include/autoconf.mk
  GEN      spl/include/autoconf.mk

```

Step2: 执行 mk_kernel.sh 编译 kernel 源码

```

root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_kernel.sh
Build kernel and drivers on the /mnt/workspace/linux/kernel
make: Entering directory '/mnt/workspace/linux/kernel'
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK      include/config/kernel.release
  CHK      include/generated/uapi/linux/version.h
  CHK      include/generated/utsrelease.h
  HOSTCC  scripts/genksyms/lex.lex.o
  HOSTCC  scripts/dtc/dtc.o

```

3.4 制作 qspi 镜像

Step3: 制作 qspi 镜像，制作完毕后，可以在 images/qspi_image 目录下看到 qspi_image.bin 镜像

```

root@osrc-virtual-machine:/mnt/workspace/linux/kernel# mk_qspi_image.sh
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# 

```

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# cd /mnt/workspace/linux/images/qspi_image/
root@osrc-virtual-machine:/mnt/workspace/linux/images/qspi_image# ls
qspi_image.bin
root@osrc-virtual-machine:/mnt/workspace/linux/images/qspi_image#
```

3.5 安装 screen

Step1:安装 screen，可以直接在 Ubuntu 系统下查看串口调试信息，当然，你也可以在 Windows 下使用 putty 之类的

```
root@osrc-virtual-machine:~# sudo apt-get install screen
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  linux-headers-4.4.0-59 linux-headers-4.4.0-59-generic linux-headers-4.4.0-62
  linux-headers-4.4.0-62-generic linux-image-4.4.0-59-generic linux-image-4.4.0-62-generic
  linux-image-extra-4.4.0-59-generic linux-image-extra-4.4.0-62-generic
Use 'sudo apt autoremove' to remove them.
Suggested packages:
  iselect | screenie | byobu ncurses-term
The following NEW packages will be installed:
  screen
0 upgraded, 1 newly installed, 0 to remove and 334 not upgraded.
Need to get 560 kB of archives.
After this operation, 972 kB of additional disk space will be used.
Get:1 http://cn.archive.ubuntu.com/ubuntu xenial/main amd64 screen amd64 4.3.1-2build1 [560 kB]
Fetched 560 kB in 0s (918 kB/s)
Selecting previously unselected package screen.
(Reading database ... 274503 files and directories currently installed.)
Preparing to unpack .../screen_4.3.1-2build1_amd64.deb ...
Unpacking screen (4.3.1-2build1) ...
Processing triggers for systemd (229-4ubuntu10) ...
Processing triggers for ureadahead (0.100.0-19) ...
Processing triggers for install-info (6.1.0.dfsg.1-5) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up screen (4.3.1-2build1) ...
Processing triggers for systemd (229-4ubuntu10) ...
Processing triggers for ureadahead (0.100.0-19) ...
root@osrc-virtual-machine:~#
```

Step2:输入 /dev/ttyUSB0 正是串口

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# ls /dev/
agpgart      loop0      ram2      tty10     tty4      ttyS1    uhid
autofs       loop1      ram3      tty11     tty40     ttyS10   uinput
block        loop2      ram4      tty12     tty41     ttyS11   urandom
bsg         loop3      ram5      tty13     tty42     ttyS12   userio
btrfs-control loop4      ram6      tty14     tty43     ttyS13   vcs
bus          loop5      ram7      tty15     tty44     ttyS14   vcs1
cdrom        loop6      ram8      tty16     tty45     ttyS15   vcs2
cdrw        loop7      ram9      tty17     tty46     ttyS16   vcs3
char        loop-control random   rtkill   tty18     tty47     ttyS17   vcs4
console      mapper   mmclog   rtc      tty19     tty48     ttyS18   vcs5
core         mem       mem      rtc0     tty20     tty49     ttyS19   vcs6
cpu          memory_bandwidth sda      tty21     tty50     ttyS20   vcsa
cpu_dma_latency suse     midi    sg0      tty22     tty51     ttyS21   vcsa1
disk         mqueue   net      sda1     tty23     tty52     ttyS22   vcsa2
dm-midi      net      sda2     sg1      tty24     tty53     ttyS23   vcsa3
dri          network_latency sdb      sg2      tty25     tty54     ttyS24   vcsa4
dvd          network_throughput sdb1    serial   tty26     tty55     ttyS25   vcsa5
ecryptfs     null     port    sg0      tty27     tty56     ttyS26   vcsa6
fb0          port     ppp     sg1      tty28     tty57     ttyS27   vcsa7
fd           ppp      psaux   sg2      tty29     tty58     ttyS28   vfio
full         psaux   ptmx    sg3      tty30     tty59     ttyS29   vga_arbiter
fuse         ptmx   snapshot sram0   tty31     tty60     ttyS30   vhost-net
hidraw0     pts      snd     sram1   tty32     tty61     ttyS31   vmci
hpet         ram0     sr0     ram1    tty33     tty62     ttyS32   vsock
hugepages   ram2     stderr  ram10   tty34     tty63     ttyS33   zero
hwrng        ram3     stdio   ram11   tty35     tty7      ttyS34
initctl     ram4     stdout  ram12   tty36     tty8      ttyS35
input        ram5     tty    ram13   tty37     tty9      ttyS36
kmsg         ram6     tty0   ram14   tty38     ttyprintk  ttyS37
lightnvm   ram7     tty1   ram15   tty39     tty50     ttyS38
log          ram8
```

3.6 一件烧写 QSPI FLASH 1

接下 JTAG 并打开开发板电源，执行 `program_qspi_flash.sh` 将 QSPI 镜像烧录到 QSPI 芯片上，或许会出现以下错误，没关系，此时因为驱动还没有加载，重新执行一次。

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# program_qspi_flash.sh
*****
* Xilinx Program Flash
***** Program Flash v2015.4 (64-bit)
**** SW Build 1412921 on Wed Nov 18 09:44:32 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Connecting to hw_server @ TCP:127.0.0.1:3121

WARNING: Failed to connect to hw_server at TCP:127.0.0.1:3121
Attempting to launch hw_server at TCP:127.0.0.1:3121

Connected to hw_server @ TCP:127.0.0.1:3121
Available targets and devices:
Target 0 : jsn-JTAG-HS1-210249855195
    Device 0: jsn-JTAG-HS1-210249855195-4ba00477-0

JTAG chain configuration
-----
Device   ID Code      IR Length  Part Name
1        4ba00477      4         arm_dap
2        13722093       6         xc7z010

-----
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".
-----
```

```
CortexA9 Processor Configuration
-----
Version.....0x00000003
User ID.....0x00000000
No of PC Breakpoints.....6
No of Addr/Data Watchpoints.....4
Waiting for Bootrom to re-enable DAP after reset
Processor Reset .... DONE
SF: Detected S25FL256S_64K with page size 256 Bytes, erase size 64 KiB, total 32 MiB
Performing Erase Operation...
Erase Operation successful.
INFO: [Xicom 50-44] Elapsed time = 29 sec.
Performing Program Operation...
0%.....10%.....
.....20%.....
.....30%.....
.....40%.....
.....50%.....
.....60%.....
.....70%.....
.....80%.....
.....90%.....
%.....100%.....
.....Program Operation successful.
INFO: [Xicom 50-44] Elapsed time = 234 sec.

Flash Operation Successful
root@osrc-virtual-machine:/mnt/workspace/linux/kernel#
```

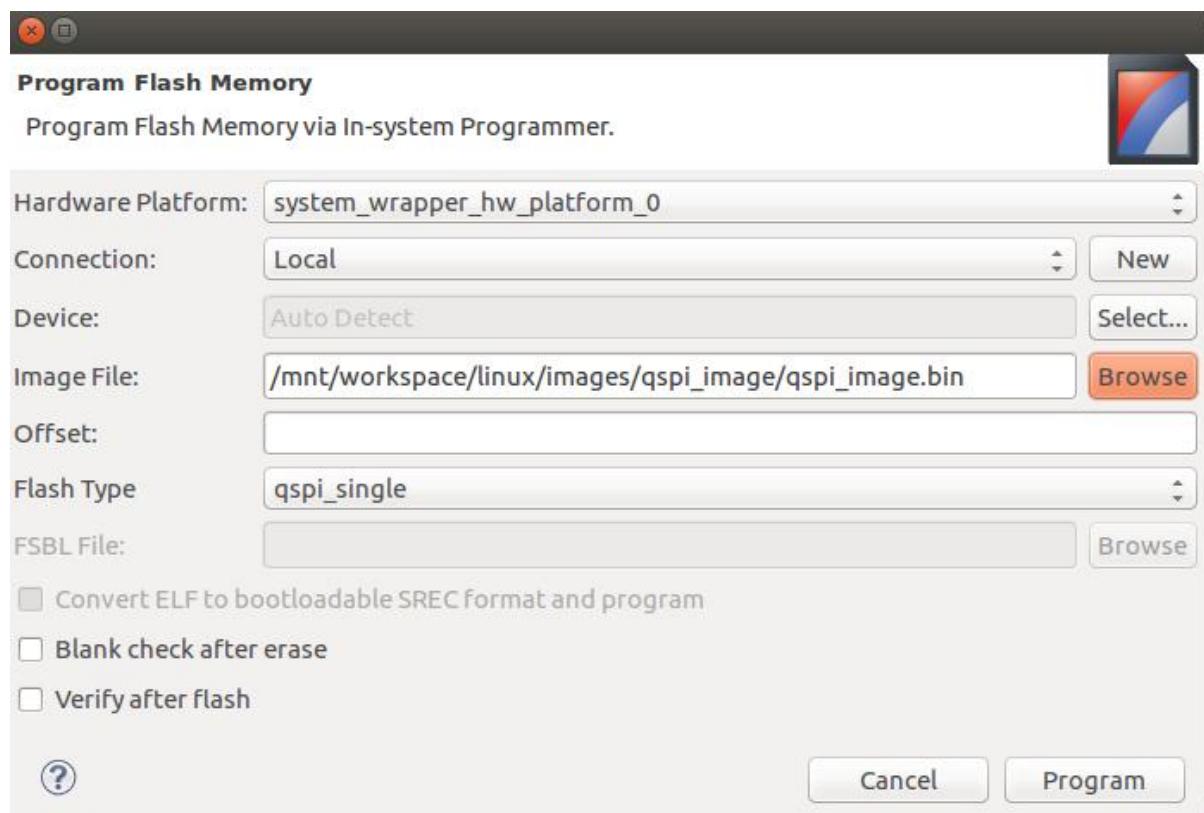
3.7 烧写 QSPI FLASH 2

如果确实不行，那么打开 SDK 来烧录吧

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel# cd /mnt/workspace/miz701n  
root@osrc-virtual-machine:/mnt/workspace/miz701n# cd Miz_sys  
root@osrc-virtual-machine:/mnt/workspace/miz701n/Miz_sys# vivado Miz_sys.xpr
```

如果是切换到 su 的话，可能会提示没有找到 license，那么把/home/osrc/.Xilinx 目录复制到/root/目录下

```
root@osrc-virtual-machine:/home/osrc# cp .Xilinx/ /root/ -r  
root@osrc-virtual-machine:/home/osrc#
```



烧录完毕后，重启开发板，当然，请先确保调整启动模式为从 QSPI 启动，在 Ubuntu 下打开串口终端

```
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# screen /dev/ttyUSB0 115200
```

即可以看到以下信息

```
ULPI transceiver vendor/product ID 0x0151,0x1507
Found TI TUSB1210 ULPI transceiver.
ULPI integrity check: passed.
ci_hdrc ci_hdrc.0: EHCI Host Controller
ci_hdrc ci_hdrc.0: new USB bus registered, assigned bus number 1
ci_hdrc ci_hdrc.0: USB 2.0 started, EHCI 1.00
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
mousedev: PS/2 mouse device common for all mice
i2c /dev entries driver
EDAC MC: ECC not enabled
Xilinx Zynq CpuIdle Driver started
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhci-pltfm: SDHCI platform and OF driver helper
mmc0: SDHCI controller on e0100000.sdhci [e0100000.sdhci] using DMA
mmc1: SDHCI controller on e0101000.sdhci [e0101000.sdhci] using DMA
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
NET: Registered protocol family 10
mmc0: new high speed SDHC card at address 0001
sit: IPv6 over IPv4 tunneling driver
NET: Registered protocol family 17
mmcblk0: mmc0:0001 00000 7.32 GiB
  mmcblk0: p1 p2
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
Registering SWP/SWPB emulation handler
hctosys: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
RAMDISK: gzip image found at block 0
usb 1-1: new full-speed USB device number 2 using ci_hdrc
mmc1: new high speed MMC card at address 0001
  mmcblk1: mmc1:0001 P1XXXX 7.20 GiB
  mmcblk1boot0: mmc1:0001 P1XXXX partition 1 2.00 MiB
  mmcblk1boot1: mmc1:0001 P1XXXX partition 2 2.00 MiB
  mmcblk1rpmb: mmc1:0001 P1XXXX partition 3 128 KiB
    mmcblk1: p1
EXT4-fs (ram0): couldn't mount as ext3 due to feature incompatibilities
EXT4-fs (ram0): mounted filesystem without journal. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 1024K (c0800000 - c0900000)
random: sshd urandom read with 5 bits of entropy available
zyinq> █
```

在 HDMI 显示器上也可以看到相关信息。



S04_CH04_自动挂载 8GB EMMC 板载内存

4.1 概述

EMMC 内存是直接焊接在 PCB 板上的，因此不能像 TF 卡那样方便的通过插入 USB 接口进行分区格式化。在第一次使用的时候，首先要分区，然后格式化，最后挂载到系统。本课程承接上一课程，请读者学习的时候注意学习顺序。

4.2 执行 source setup_env.sh

不管如何第一条指令我们要执行 source setup_env.sh 设置环境变量

4.3 修改 zynq-7000.dtsi 文件

Step1:进入如下目录并且用 vim 打开 zynq-7000.dtsi 文件

```
root@osrc-virtual-machine:/mnt/workspace/linux/kernel/arch/arm/boot/dts# vim zynq-7000.dtsi
```

Step2:通过 vim 打开文件后入下图

```
root@osrc-virtual-machine: /mnt/workspace/linux/kernel/arch/arm/boot/dts
/*
 * Copyright (c) 2011 - 2014 Xilinx
 *
 * This software is licensed under the terms of the GNU General Public
 * License version 2, as published by the Free Software Foundation, and
 * may be copied, distributed, and modified under those terms.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 */
/include/ "skeleton.dtsi"

{
    compatible = "xlnx,zynq-7000";

    cpus {
        #address-cells = <1>;
        #size-cells = <0>

        cpu0: cpu@0 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <0>;
            clocks = <&clkc 3>;
            clock-latency = <1000>;
            cpu0-supply = <&regulator_vccpint>;
            operating-points = <
                /* kHz      uV */
            >
        }
    }
}
```

Step3:输入/sdhci0 (vim 搜索命令)(在左下角可以看到输入的命令)按 回车

```

/ {
    compatible = "xlnx,zynq-7000";

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;

        cpu0: cpu@0 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <0>;
            clocks = <&clkc 3>;
            clock-latency = <1000>;
            cpu0-supply = <&regulator_vccpint>;
            operating-points = <
                /* kHz      uV */
        };
    };
}

```

Step4:找到如下位置(完成 vim 搜索 可以看到 vim 搜索是效率非常高的)(修改前一定要备份下, 因为第七课开始需要继续用到未修改前的)

```

sdhci0: sdhci@e0100000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 21>, <&clkc 32>;
    interrupt-parent = <&intc>;
    interrupts = <0 24 4>;
    reg = <0xe0100000 0x1000>;
    broken-adma2;
};

sdhci1: sdhci@e0101000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 22>, <&clkc 33>;
    interrupt-parent = <&intc>;
    interrupts = <0 47 4>;
    reg = <0xe0101000 0x1000>;
    broken-adma2;
};

```

Step5:现在需要把 sdhci0 和 sdhci1 掉个顺序, 这里也是使用 vim 命令操作。输入 v 后可以看到左下角变成了大写的 VISUAL

```

sdhci0: sdhci@e0100000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 21>, <&clkc 32>;
    interrupt-parent = <&intc>;
    interrupts = <0 24 4>;
    reg = <0xe0100000 0x1000>;
    broken-adma2;
};

-- VISUAL --

```

Step6:按键盘 j(vim 命令 每按一次鼠标下移一行)选中 sdhci0 整个段落

```
sdhci0: sdhci@e0100000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 21>, <&clkc 32>;
    interrupt-parent = <&intc>;
    interrupts = <0 24 4>;
    reg = <0xe0100000 0x1000>;
    broken-adma2;
};
```

Step7:按键盘 d 实现剪切, 然后, 把光标移动到 sdhci0 段落最后一行再输入 p 就完成剪切了

```
sdhci1: sdhci@e0101000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 22>, <&clkc 33>;
    interrupt-parent = <&intc>;
    interrupts = <0 47 4>;
    reg = <0xe0101000 0x1000>;
    broken-adma2;
};

sdhci0: sdhci@e0100000 {
    compatible = "arasan,sdhci-8.9a";
    status = "disabled";
    clock-names = "clk_xin", "clk_ahb";
    clocks = <&clkc 21>, <&clkc 32>;
    interrupt-parent = <&intc>;
    interrupts = <0 24 4>;
    reg = <0xe0100000 0x1000>;
    broken-adma2;
};
```

Step8:输入:wq!保存并关闭 zynq-7000.dtsi 文件

4.4 设置 mount_emmc.sh 批处理命令的开机启动

Step1:首先看下已经编写好的 mount_emmc.sh 文件。可以看到执行 mount_emmc.sh 批处理命令, 首先是在根目录下创建 emmc 文件夹路径。之后再把 ext2 格式的 EMMC 挂载到/mnt/emmc..。如果开机发现没有分区, 就会分区后格式化, 再挂载。

```
mkdir -p /mnt/emmc
mount -t ext2 /dev/mmcblk0p1 /mnt/emmc
ret=$?
```

```

echo $ret
if [ $ret -ne 0 ]; then
    echo -e "n \n p \n 1 \n \n \n w \n" | fdisk /dev/mmcblk0
    mkfs.ext2 /dev/mmcblk0p1
    mount -t ext2 /dev/mmcblk0p1 /mnt/emmc
fi

```

Step2:挂载 mount_emmc.sh 到根文件系统，并且拷贝 mount_emmc.sh 到 rootfs 路径

```

root@osrc-virtual-machine:/mnt/workspace/linux/output# mount_rootfs.sh
根文件系统已经挂载在/mnt/workspace/linux/output/rootfs。
root@osrc-virtual-machine:/mnt/workspace/linux/output# cd rootfs
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# cp mount_emmc.sh ../../scripts/mount_emmc.sh
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs#

```

Step3:打开根文件系统下的 etc/init.d/rcS 文件

```

root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# vi etc/init.d/rcS

```

Step4:增加如下代码并且输入:wq!保存退出

```

if [ -f /mnt/init.sh ]
then
    echo "++ Running user script init.sh from SD Card"
    source /mnt/init.sh
fi

if [ -f /mount_emmc.sh ]
then
    echo "++Running user script mount_emmc.sh"
    source /mount_emmc.sh
fi

echo "rcS Complete"
~_

```

可以看到红色框线内的代码就是在开机后可以执行 mount_emmc.sh 文件，当然 LINUX 下面开机自动运行的方式很多，这里不一一列举。

Step5:切换到 rootfs 路径之外的任何路径最简单地可以输入 cd 命令

```

root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# cd
root@osrc-virtual-machine:~#

```

Step6:执行 umount_rootfs.sh

```

root@osrc-virtual-machine:~# umount_rootfs.sh
Unmount the initrd image...
Compress the initrd image...
Wrapping the image with a U-Boot header...
Image Name:
Created:      Sun Apr  9 22:21:30 2017
Image Type:   ARM Linux RAMDisk Image (gzip compressed)
Data Size:    5307435 Bytes = 5183.04 kB = 5.06 MB
Load Address: 00000000
Entry Point:  00000000
root@osrc-virtual-machine:~#

```

4.5 烧写程序到 QSPI FLASH

Step1：硬件设置拨码开关到 ON ON ON OFF 这样为 QSPI flash 模式,连接下载器和串口。

Step2:分别执行 mk_qspi_image.sh 和 program_qspi_flash.sh 打包并烧录 QSPI 镜像

```
root@osrc-virtual-machine:~# mk_qspi_image.sh
root@osrc-virtual-machine:~# program_qspi_flash.sh

***** Xilinx Program Flash
***** Program Flash v2015.4 (64-bit)
**** SW Build 1412921 on Wed Nov 18 09:44:32 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Connecting to hw_server @ TCP:127.0.0.1:3121

WARNING: Failed to connect to hw_server at TCP:127.0.0.1:3121
Attempting to launch hw_server at TCP:127.0.0.1:3121

Connected to hw_server @ TCP:127.0.0.1:3121
Available targets and devices:
Target 0 : jsn-JTAG-HS1-210249855627
Device 0: jsn-JTAG-HS1-210249855627-4ba00477-0

JTAG chain configuration
-----
Device ID Code      IR Length Part Name
1   4ba00477        4          arm_dap
2   23727093         6          xc7z020

-----
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".
```

4.6 验证测试

Step1:烧写完成后，输入 ls /dev 查看串口是否连接好，如下图红框所示设备

```
root@osrc-virtual-machine:~# ls /dev
agpgart      hwrng      ppp      tty0    tty32    tty56    ttyS20   vcs2
autofs       initctl    psaux    tty1    tty33    tty57    ttyS21   vcs3
block        input      ptmx     tty10   tty34    tty58    ttyS22   vcs4
bsg         kmsg      pts      tty11   tty35    tty59    ttyS23   vcs5
btrfs-control lightnvm random   tty12   tty36    tty6    ttyS24   vcs6
bus          log       rfckill  tty13   tty37    tty60    ttyS25   vcs7
cdrom        loop0     rtc      tty14   tty38    tty61    ttyS26   vcsa
cdrw         loop1     sda      tty15   tty39    tty62    ttyS27   vcsa1
char         loop2     sda1     tty16   tty4    tty63    ttyS28   vcsa2
console      loop3     sda2     tty17   tty40    tty7    ttyS29   vcsa3
core         loop4     sda5     tty18   tty41    tty8    ttyS3    vcsa4
cpu          loop5     sdb      tty19   tty42    tty9    ttyS30   vcsa5
cpu_dma_latency loop6     sdb1     tty2    tty43    ttyprintk  ttyS31   vcsa6
cuse         loop7     serial   tty20   tty44    tty50    ttyS4    vcsa7
disk         loop-control  sg0     tty21   tty45    tty51    ttyS5    vfio
dmmidi       mapper    sg1     tty23   tty47    tty511   ttyS6    vga_arbiter
dri          mcelog    sg2     tty24   tty48    tty512   ttyS7    vhci
dvd          mem       shm     tty25   tty49    tty513   ttyS8    vhost-net
ecryptfs     memory_bandwidth  midi   snapshot  tty26   tty5    ttyS9    vmci
fb0          mqueue   snd      tty27   tty50    tty515   ttyUSB0  vsock
fd           net       sr0     tty28   tty51    tty516   uhid    zero
full         network_latency  stderr  stdin    stdout   tty29   tty52    urandom
fuse         network_throughput  null   tty3    tty53    tty517   userio
hidraw0      null     stderr  tty30   tty54    tty518   vcs
hpet         port      tty     tty31   tty55    tty519   vcs1
hugepages    port      tty     tty32   tty56    tty520   vcs2

root@osrc-virtual-machine:~# s
```

Step2:用 screen 打开/dev/ttyUSB0

```
fb0      midi      snapshot  tty26  tty5   ttyS14  ttyUSB0  vsock
fd       mqueue    snd        tty27  tty50  ttyS15  uhid     zero
full    net        sr0        tty28  tty51  ttyS16  uinput   urandom
fuse    network_latency  stderr  tty29  tty52  ttyS17  userio
hidraw0 network_throughput  stdin   tty3   tty53  ttyS18  vcs
hpet    null       stdout    tty30  tty54  ttyS19  vcs1
hugepages port      tty       tty31  tty55  ttyS2
root@osrc-virtual-machine:~# screen /dev/ttyUSB0 115200
```

Step3:断电后重启开发板

Step4:用“ls /mnt/emmc/”可以查看到 emmc 已经挂载好了，然后我们往里面写入一个 hello.txt 文件

```
zynq> ls /mnt/emmc/
lost+found
zynq> vi /mnt/emmc/hello.txt
```

Step5:输入 hello emmc 几个单词(先按 i 再输入单词，输入完成后 :wq!保存并退出)

```
hello emmc
~
~
```

Step6:后卸载 emmc，再重新挂载 emmc，确认挂载的是否是 emmc 中还有 hello.txt 文件

```
zynq> ls /mnt/emmc/
hello.txt  lost+found
zynq> umount /mnt/emmc/
zynq> ls /mnt/emmc/
zynq> mount -t ext2 /dev/mmcblk0p1 /mnt/emmc/
zynq> ls /mnt/emmc/
hello.txt  lost+found
zynq>
```

Step7:打开 hello.txt 查看文件内容是否是我们之前输入的

```
zynq> vi /mnt/emmc/hello.txt
hello emmc
~
```

Step8: 请读者断电重启，查看 hello.txt 是否还存在(结果是存在)。

```
devtmpfs: Mounted
Freeing unused kernel memory: 1024K (c0800000 - c0900000)
EXT4-fs (mmcblk0p1): couldn't mount as ext3 due to feature incompatibilities
EXT4-fs (mmcblk0p1): filesystem too large to mount safely on this system
EXT2-fs (mmcblk0p1): warning: mounting unchecked fs, running e2fsck is recommended
random: sshd urandom read with 4 bits of entropy available
zynq> ls /mnt/emmc/
emmc      hello.txt  lost+found
```

4.7 思考为什么

因为你的 emmc 是挂在 SD1 上是吧，这样的话，当系统启动后，它扫描了设备树，就把 SD0 放在了第一个设备，SD1 放在了第二个设备，如果你没有插 SD 卡的话，那 emmc 就是第一个设备了，这样，你的脚本就比较难判断到底哪个才是 emmc

S04_CH05_在线升级 QSPI 镜像(U 盘方式)

5.1 概述

在线升级 QSPI 镜像，是指从 QSPI 模式或 SD 模式下启动 Linux，然后通过 U 盘、网络等下载 QSPI 镜像，然后在 Linux 下升级 QSPI 镜像，而不是通过 JTAG 的方式烧录 QSPI 镜像，有点类似于当前安卓手机的 OTA 升级方式。

5.2 执行 source setup_env.sh

不管如何第一条指令我们要执行 source setup_env.sh 设置环境变量

5.3 烧写程序到 QSPI FLASH

Step1：硬件设置拨码开关到 ON ON ON OFF 这样为 QSPI flash 模式,连接下载器和串口。

Step2:分别执行 mk_qspi_image.sh 和 program_qspi_flash.sh 打包并烧录 QSPI 镜像

```
root@osrc-virtual-machine:~# mk_qspi_image.sh
root@osrc-virtual-machine:~# program_qspi_flash.sh

***** Xilinx Program Flash
***** Program Flash v2015.4 (64-bit)
**** SW Build 1412921 on Wed Nov 18 09:44:32 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Connecting to hw_server @ TCP:127.0.0.1:3121

WARNING: Failed to connect to hw_server at TCP:127.0.0.1:3121
Attempting to launch hw_server at TCP:127.0.0.1:3121

Connected to hw_server @ TCP:127.0.0.1:3121
Available targets and devices:
Target 0 : jsn-JTAG-HS1-210249855627
    Device 0: jsn-JTAG-HS1-210249855627-4ba00477-0

JTAG chain configuration
-----
Device   ID Code      IR Length  Part Name
 1       4ba00477      4        arm_dap
 2       23727093      6        xc7z020

-----
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".
```

5.4 查看系统根目录

Step1:烧写完成后，输入 ls /dev 查看串口是否连接好，如下图红框所示设备

```
root@osrc-virtual-machine:~# ls /dev
appgarb      hwrng      ppp      tty0      tty32      tty56      ttyS20      vcs2
autofs       initctl    psaux    tty1      tty33      tty57      ttyS21      vcs3
block        input      ptmx     tty10     tty34      tty58      ttyS22      vcs4
bsg         kmsg      pts      tty11     tty35      tty59      ttyS23      vcs5
btrfs-control lightnvm   random   tty12     tty36      tty6      ttyS24      vcs6
bus          log       rfkill   tty13     tty37      tty60      ttyS25      vcs7
cdrom       loop0     rtc      tty14     tty38      tty61      ttyS26      vcsa
cdrw        loop1     rtc0     tty15     tty39      tty62      ttyS27      vcsa1
char        loop2     sda      tty16     tty4      tty63      ttyS28      vcsa2
console     loop3     sda1     tty17     tty40      tty7      ttyS29      vcsa3
core        loop4     sda2     tty18     tty41      tty8      ttyS3      vcsa4
cpu         loop5     sda5     tty19     tty42      tty9      ttyS30      vcsa5
cpu_dma_latency loop6     sdb      tty2      tty43      ttyprintk  ttyS31      vcsa6
cuse        loop7     sdb1     tty20     tty44      tty50      ttyS4      vcsa7
disk        loop-control serial   tty21     tty45      tty51      ttyS5      vfio
dmmdidi    mapper    sg0      tty22     tty46      tty510     ttyS6      vga_arbiter
dri         mcelog    sg1      tty23     tty47      tty511     ttyS7      vhci
dvd          mem      sg2      tty24     tty48      tty512     ttyS8      vhost-net
ecryptfs   memory_bandwidth shm      tty25     tty49      tty513     ttyS9      vmci
fb0         midi      snapshot  tty26     tty5      tty514     ttyUSBO    vsock
fd          mqueue   snd      tty27     tty50      tty515     uhid      zero
full       net       sr0      tty28     tty51      tty516     uinput
fuse        network_latency stderr   tty29     tty52      tty517     urandom
hidraw0    network_throughput stdin   tty3      tty53      tty518     userio
hpet        null      stdout   tty30     tty54      tty519     vcs
hugepages   port      tty     tty31     tty55      tty52      vcs1
root@osrc-virtual-machine:~# s
```

Step2: 用 screen 打开 /dev/ttyUSB0

```
fb0          midi      snapshot  tty26      tty5      ttyS14     ttyUSBO    vsock
fd          mqueue   snd      tty27      tty50     ttyS15     uhid      zero
full       net       sr0      tty28      tty51     ttyS16     uinput
fuse        network_latency stderr   tty29     tty52      ttyS17     urandom
hidraw0    network_throughput stdin   tty3      tty53      tty518     userio
hpet        null      stdout   tty30     tty54      tty519     vcs
hugepages   port      tty     tty31     tty55      tty52      vcs1
root@osrc-virtual-machine:~# screen /dev/ttyUSB0 115200
```

Step3: 开发板根目录

设置开发板从 QSPI 模式启动，启动后，我们可以看到根目录下有 update_qspi.sh 脚本，这个脚本是 xilinx 提供的 ramdisk 就自带的，不过我们这里不是使用这种方法来升级 QSPI 镜像。

```
Freeing unused kernel memory: 1024K (c0800000 - c0900000)
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
random: sshd urandom read with 4 bits of entropy available
zynq> ls /
README      home      lost+found      root      update_qspi.sh
bin         lib       mnt      sbin      usr
dev         licenses  opt      sys      var
etc         linuxrc  proc      tmp
zynq>
```

5.5 基于 U 盘在线升级

Step1: 为了展示升级的 QSPI 镜像与升级前的 QSPI 镜像不一样，或者是说证明我们在线升级成功，我们将刚才看的到“update_qspi.sh”文件删掉，首先在 ubuntu 开发环境下执行 mount_rootfs.sh 脚本挂载根文件系统。

```
root@osrc-virtual-machine:~# cd /mnt/workspace/linux/scripts
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# mount_rootfs.sh
根文件系统已经挂载在 /mnt/workspace/linux/output/rootfs.
```

Step2: 然后切换到 output/rootfs 目录下，执行 rm update_qspi.sh 命令将“update_qspi.sh”脚本删掉，最后执行 umount_rootfs.sh 脚本卸载、处理根文件系统。

```

root@osrc-virtual-machine:/mnt/workspace/linux/scripts# cd .. /output/rootfs/
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# ls
bin etc lib linuxrc mnt opt README sbin tmp usr
dev home licenses lost+found mount_emmc.sh proc root sys update_qspi.sh var
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# rm update_qspi.sh
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# ls
bin etc lib linuxrc mnt opt README sbin tmp var
dev home licenses lost+found mount_emmc.sh proc root sys usr
root@osrc-virtual-machine:/mnt/workspace/linux/output/rootfs# cd ..
root@osrc-virtual-machine:/mnt/workspace/linux/output# cd ..
root@osrc-virtual-machine:/mnt/workspace/linux# cd scripts
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# umount_rootfs.sh
Unmount the initrd image...
Compress the initrd image...
Wrapping the image with a U-Boot header...
Image Name:
Created:      Sun Apr  9 23:15:47 2017
Image Type:   ARM Linux RAMDisk Image (gzip compressed)
Data Size:    5307433 Bytes = 5183.04 kB = 5.06 MB
Load Address: 00000000
Entry Point:  00000000
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# █

```

Step3: 执行 `mk_qspi_image.sh` 重新打包 QSPI 镜像，并将 `images/qspi_image` 目录的 QSPI 镜像拷贝到 U 盘中(可以用我们配套的 TF 卡+读卡器)。注意：0403-0201 这是系统自动命名的，大家要注意，不要变了名字就不认识了。

```

root@osrc-virtual-machine:/mnt/workspace/linux/scripts# cp .. /images/qspi_image/ /media/osrc/0403-0201/ -r
root@osrc-virtual-machine:/mnt/workspace/linux/scripts# █

```

Step4: 在正在运行的开发板子上插入 U 盘，在 screen 打开的终端下输入 `ls /dev/` 命令查看 U 盘对应的设备，我这边对应的是 `/dev/sda` 设备，通过 `mount /dev/sda1 /mnt` 将 U 盘挂载到 `/mnt` 目录下，然后切换到 U 盘里存放 QSPI 镜像的目录下

```

zyng> mount /dev/sda1 /mnt/
FAT-fs (sda1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
zyng> cd /mnt/qspi_image/
zyng> ls
qspi_image.bin
zyng> █

```

Step5: 通过 `dd` 命令烧录 QSPI 镜像，可以看到，在线升级会比通过 JTAG 方式快。

```

zyng> ls /mnt/qspi_image/qspi_image.bin
/mnt/qspi_image/qspi_image.bin
zyng> dd if=/mnt/qspi_image/qspi_image.bin of=/dev/mtdblock0
random: nonblocking pool is initialized
31106+1 records in
31106+1 records out
15926776 bytes (15.2MB) copied, 66.021045 seconds, 235.6KB/s
zyng> █

```

Step6: 执行 `reboot` 重启开发板

```

zyng> reboot
zyng> █

```

Step7: 现在可以看到，已经没有 `update_qspi.sh` 文件了，说明升级成功了。

```
zyng> ls /
README          home        lost+found    proc        tmp
bin             lib         mnt          root        usr
dev             licenses   mount_emmc.sh sbin        var
etc             linuxrc   opt
zyng>
```

S04_CH06_hello_linux

6.1 概述

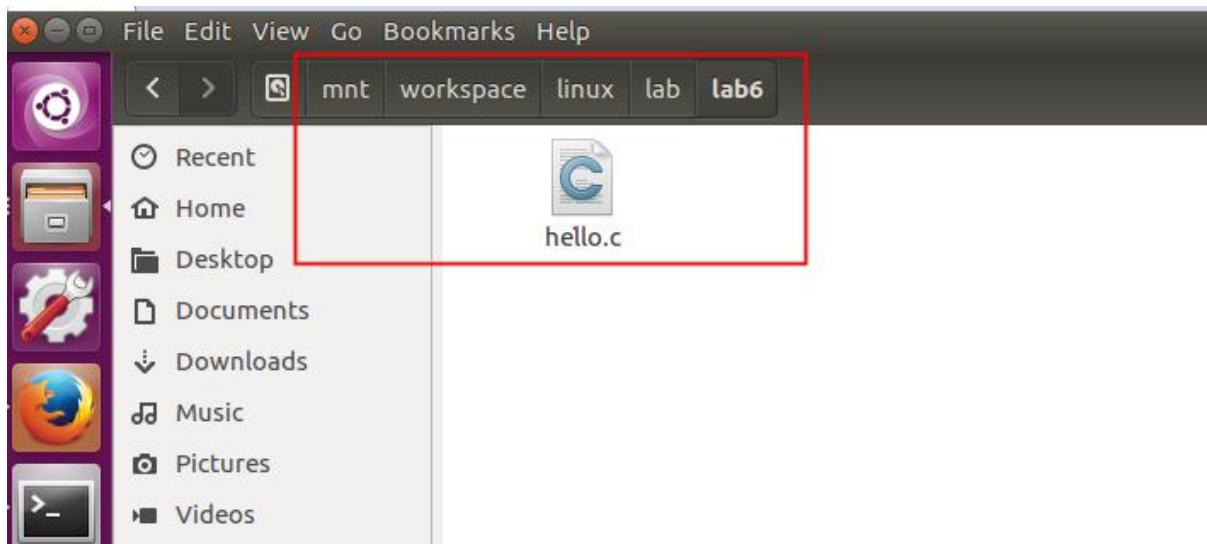
hello 是一个经典的程序，是学习入门必学的一个简单程序，能跑 hello linux 意味着编程者已经跨入编程的大门了。笔者这里的 hellolinux 程序需要在开发板的 linux 系统上执行 SD 卡运行和 EMMC 运行。

6.2 执行 source setup_env.sh

不管如何第一条指令我们要执行 source setup_env.sh 设置环境变量。

6.3 SD 卡手动运行 hello 程序

Step1:在新建 lab/lab6 文件夹，并且拷贝已经准备好的 hello.c 程序



Step2:进入 lab6 文件夹

```
root@osrc-virtual-machine:/mnt/workspace/linux/lab/lab6#
```

Step3:执行 arm-xilinx-linux-gnueabi-gcc hello.c



Step3:拷贝 lab6 文件夹和 sd_image 文件中的文件到 TF 卡



Step4: 卸载 TF 卡, 然后插入到开发板中, 连接串口到系统, 并且打开串口

```
root@osrc-virtual-machine:/mnt/workspace/linux/lab/lab6# ls /dev
agpgart      hwrng      ppp      tty0      tty32      tty56      ttyS20      vcs2
autofs       initctl    psaux    tty1      tty33      tty57      ttyS21      vcs3
block        input      ptmx     tty10     tty34      tty58      ttyS22      vcs4
bsg          kmsg      random   tty11     tty35      tty59      ttyS23      vcs5
btrfs-control lightnvm  rfkill   tty12     tty36      tty6      ttyS24      vcs6
bus          log       rtc      tty13     tty37      tty60      ttyS25      vcs7
cdrom        loop0     rtc0     tty14     tty38      tty61      ttyS26      vcsa
cdrw         loop1     rtc0     tty15     tty39      tty62      ttyS27      vcsa1
char         loop2     sda      tty16     tty4      tty63      ttyS28      vcsa2
console      loop3     sda1     tty17     tty40      tty7      ttyS29      vcsa3
core         loop4     sda2     tty18     tty41      tty8      ttyS3      vcsa4
cpu          loop5     sda5     tty19     tty42      tty9      ttyS30      vcsa5
cpu_dma_latency loop6     sdb      tty2      tty43      ttyprintk  ttyS31      vcsa6
cuse         loop7     sdb1     tty20     tty44      tty50      ttyS4      vcsa7
disk         loop-control serial   sg0      tty22     tty46      tty51      ttyS5      vfio
dmmidi       mapper    sg1      tty23     tty47      tty51      ttyS6      vga_arbiter
dri          mcelog    sg2      tty24     tty48      tty52      ttyS7      vhci
dvd          mem       sg3      tty25     tty49      tty53      ttyS8      vhost-net
ecryptfs     memory_bandwidth snapshot  snd      tty26     tty5      tty515     ttyS9      vmci
fb0          midi      snapshot  sr0      tty27     tty50      tty515     ttyUSBO    vsock
fd          mqueue    stderr   stdin   tty28     tty51      tty516     uhid      zero
full         net       stderr   stdio   tty29     tty52      tty517     uinput
fuse         network_latency  stdin   tty3      tty53      tty518     urandom
hidraw0     network_throughput stdio   tty30     tty54      tty519     userio
hpét        null      stderr   tty31     tty55      tty520     vcs
hugepages   port      stderr   tty32     tty56      tty521     vcs1
root@osrc-virtual-machine:/mnt/workspace/linux/lab/lab6# screen /dev/ttyUSB0 115200
```

Step5: 通电启动开发板, 启动完成后, 输入 ls /dev

```
zynq> ls /dev
bus          port      tty15      tty47
console      psaux    tty16      tty48
cpu_dma_latency ptmx     tty17      tty49
fb0          pts      tty18      tty5
full         ram0     tty19      tty50
gpiochip0    ram1     tty2      tty51
i2c          ram10    tty20      tty52
iio:device0   ram11    tty21      tty53
input         ram12    tty22      tty54
kmsg         ram13    tty23      tty55
loop-control ram14    tty24      tty56
loop0        ram15    tty25      tty57
loop1        ram2      tty26      tty58
loop2        ram3      tty27      tty59
loop3        ram4      tty28      tty6
loop4        ram5      tty29      tty60
loop5        ram6      tty3      tty61
loop6        ram7      tty30      tty62
loop7        ram8      tty31      tty63
mem          ram9      tty32      tty7
memory_bandwidth random   tty33      tty8
mice         root     tty34      tty9
mmcblk0      sda      tty35      ttyPS0
mmcblk0boot0 sda1     tty36      urandom
mmcblk0boot1 sg0      tty37      vcs
mmcblk0p1    EMMC    snd      tty38      vcs1
mmcblk0rpmb  timer   tty39      vcsa
mmcblk1      tty      tty4      vcsa1
mmcblk1p1    TF卡    tty0      vga_arbiter
mtd0        tty1      tty1      watchdog
mtd0ro      tty10     tty42     watchdog
mtdblock0   tty11     tty43     xdevcfg
network_latency  tty12     tty44     zero
network_throughput  tty13     tty45
null        tty14     tty46
```

由于本教程是基于做完第五课教材来做的，所以可以看到，mmcblk0p1 是 EMMC,而 mmcblk1p1 是 TF 卡（第四课里面调换了顺序 zynq-7000.dtsi）

Step6:

```
ls tmp (创建 tmp 文件夹)
mount /dev/mmcblk1p1 /tmp (挂载 SD 到 tmp)
cd /tmp (查看 tmo 下的目录)
ls
./a.out
最后输出: Hello Linux!
```

```
zynq> ls tmp
zynq> mount /dev/mmcblk1p1 /tmp
FAT-fs (mmcblk1p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
zynq> cd /tmp
zynq> ls
BOOT.bin      devicetree.dtb      uramdisk.image.gz
a.out          uImage
zynq> ./a.out
Hello Linux!
```

6.4 EMMC 卡手动运行 hello 程序

Step1: 复制 a.out 可执行程序到 emmc

```
zynq> cd
zynq> ls
README      home      lost+found      proc      tmp
bin         lib       mnt           root      update_qspi.sh
dev         licenses  mount_emmc.sh  sbin      usr
etc         linuxrc   opt            sys       var
zynq> cp /tmp/a.out /mnt/emmc/ -r
zynq> cd /mnt/emmc/
zynq> ls
a.out      emmc      hello.txt    lost+found
```

Step2: 执行 ./a.out

```
zynq> ./a.out
Hello Linux!
zynq>
```

Step3: 断电重启，查看 EMMC 是否还有 a.out 程序

```
zynq> ls /mnt/emmc
a.out      emmc      hello.txt    lost+found
zynq>
```

S04_CH07_Hello_Qt 在开发板上的运行

7.1 概述

本课程讲解 Qt 交叉编译环境的搭建，Qte 的安装，以及实现在开发板上运行 Qt 程序，并且实现开机启动。本教程使用的 QT 交叉编译环境版本是 qt-everywhere-opensource-src-5.7.0。Linux PC 端安装的环境是 qt-opensource-linux-x64-5.8.0。

注意：本节课使用新 ramdisk，老的 ramdisk 不能正常运行。另外本节课开始使用的

另外要注意：SD 位于/dev/mmcblk0p1，鼠标位于/dev/event0 而第四节课

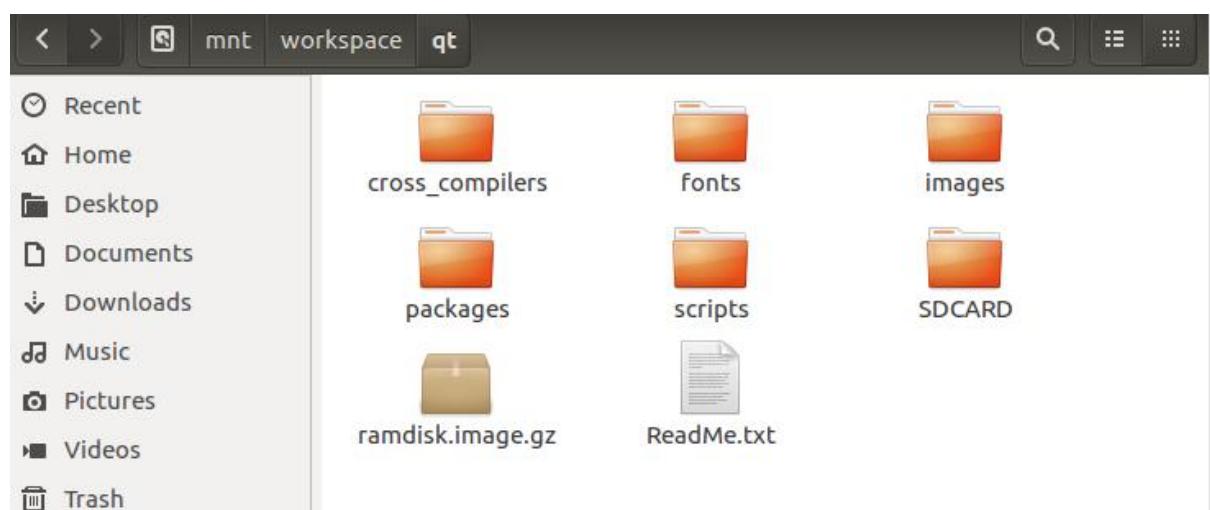
7.2 搭建交叉编译环境

7.2.1 使用批处理命令搭建交叉编译环境

Step1:首先把 qt.tar.gz 复制到 mnt/workspace 路径然后解压



查看 qt 文件夹



cross_compilers 文件夹：里面是交叉编译环境需要的文件

fonts 文件夹：里面放了 qt 需要用到的字库

images 文件夹：编译好的镜像文件和 init.sh 开机 packages 文件夹

packages 文件夹：里面有基于 xilinx 公司的 c/c++ 交叉编译文件、嵌入式交叉编译环境

qt-everywhere-opensource-src-5.7.0.tar.gz 、 LINUX PC 端 Qt 安装环境

qt-opensource-linux-x64-5.8.0.run

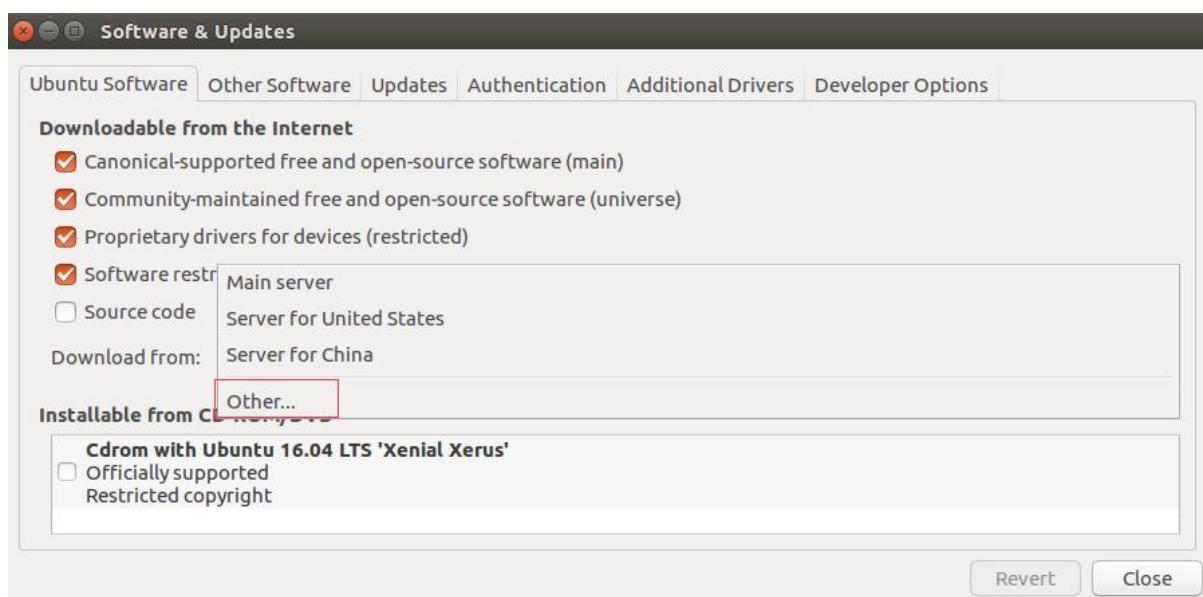
scripts 文件夹：下有三个批处理文件。

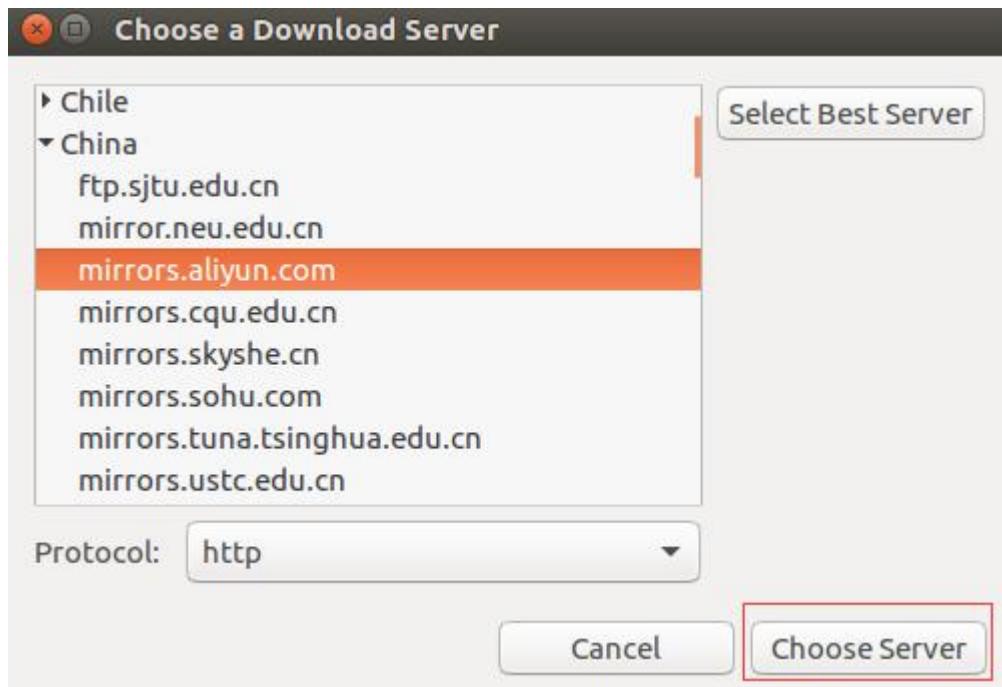
Setup_env.sh 设置或者创建了交叉编译器的路径、相关文件路径。

get_qt_sources.sh 获取需要使用到的资源

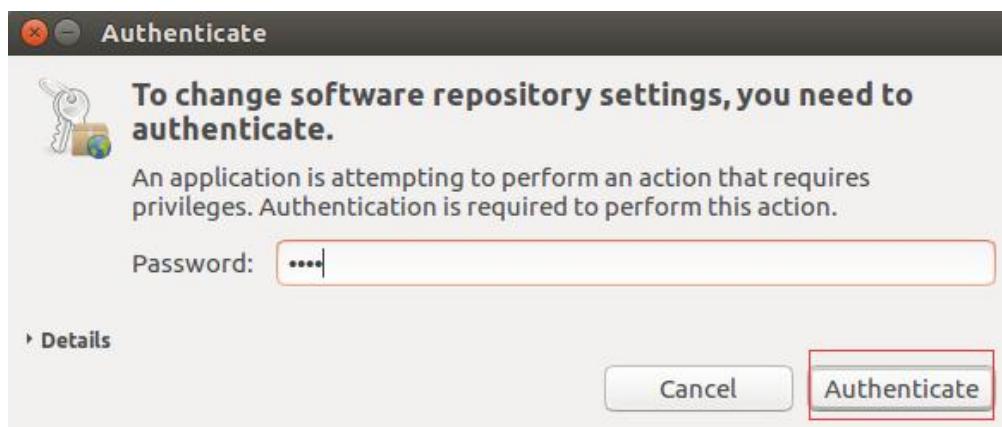
mk_qt_img.sh 包括了配置交叉编译环境、编译交叉编译环境、安装交叉编译器、编译自带的程序、制作 image 镜像。

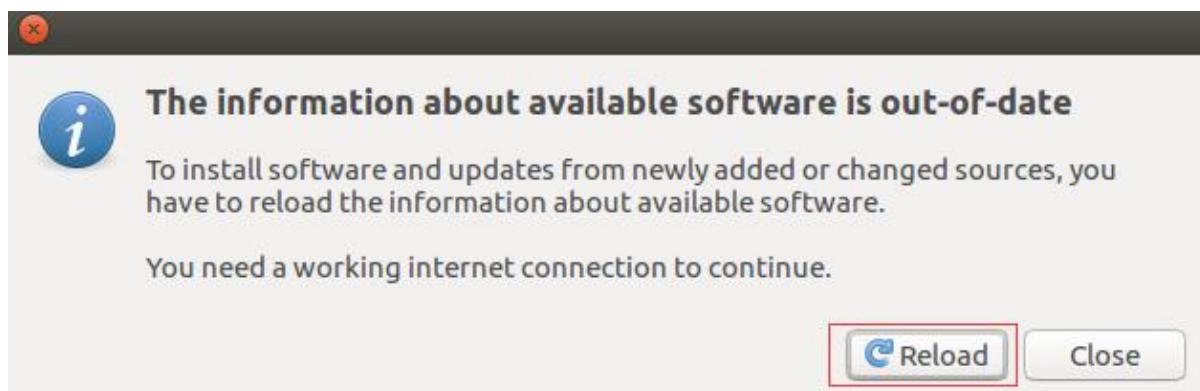
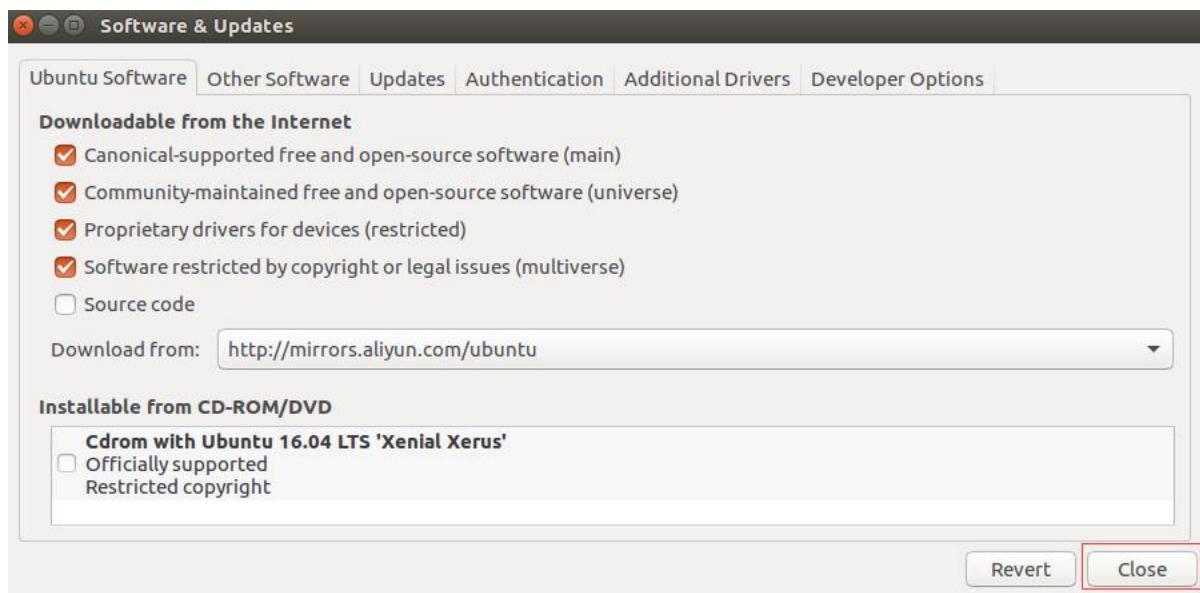
Step2: 设置更新的源





输入密码:root





Step3: 安装一些必要的文件

sudo apt-get update

sudo apt-get install libgtk2.0-0:i386 libxtst6:i386 gtk2-engines-murrine:i386 \lib32stdc++6
libxt6:i386 libdbus-glib-1-2:i386 libasound2:i386

sudo apt-get install build-essential

Step3: 切换到 scripts 目录下, 执行 source setup_env.sh 配置所需要的 QT 开发环境

```
root@osrc-virtual-machine:/mnt/workspace/qt/scripts# source setup_env.sh
root@osrc-virtual-machine:/mnt/workspace/qt/scripts#
```

Step3: 执行 get_qt_sources.sh, 获取所需源码

```
.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/main.cpp
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/BasicCamera.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/HousePlant.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/SortedForwardRen-
derer.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/PlaneEntity.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/materials.qrc
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/src/
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/src/material-
s.qdoc
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/images/
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/doc/images/mater-
ials.png
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/Barrel.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/main.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/materials.pro
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/TrefoilKnot.qml
qt-everywhere-opensource-src-5.7.0/qt3d/examples/qt3d/materials/Chest.qml
qt-everywhere-opensource-src-5.7.0/qt3d/LICENSE.LGPL3
qt-everywhere-opensource-src-5.7.0/qt3d/LICENSE.GPL
qt-everywhere-opensource-src-5.7.0/qt3d/LICENSE.GPL2
root@osrc-virtual-machine:/mnt/workspace/qt/scripts#
```

Step4: 执行 mk_qt_img.sh 脚本。

如果是一开始构建的话，可以在第一步“gmake confclean”选择 n，其它选择 y，这样就完成了 QT 库的编译。当然，也可以是一路 y。而如果你只是想执行

其中的某一步，如最后一步，可以其它 n，最后一步才是 y，总之，可以根据实际情况进行

选择。另外，请根据实际情况，修改脚本中的要相关配置。

```
root@osrc-virtual-machine:/mnt/workspace/qt/scripts# mk_qt_img.sh
Run 'gmake confclean'? (y/n)
```

```
XCB ..... no
Session management .... yes
SQL drivers:
  DB2 ..... no
  InterBase ..... no
  MySQL ..... no
  OCI ..... no
  ODBC ..... no
  PostgreSQL ..... no
  SQLite 2 ..... no
  SQLite ..... yes (plugin, using bundled copy)
  TDS ..... no
  tslib ..... no
  udev ..... no
  xkbcommon-x11..... no
  xkbcommon-evdev..... no
  zlib ..... yes (bundled copy)

NOTE: Qt is using double for qreal on this system. This is binary incompatible a
gainst Qt 5.1.
Configure with '-qreal float' to create a build that is binary compatible with 5
.1.
Info: creating stash file /mnt/workspace/qt/qt-src-5.7.0/.qmake.stash
Info: creating super cache file /mnt/workspace/qt/qt-src-5.7.0/.qmake.super

Qt is now configured for building. Just run 'make'.
Once everything is built, you must run 'make install'.
Qt will be installed into /mnt/workspace/qt/qt-arm-5.7.0

Prior to reconfiguration, make sure you remove any leftovers from
the previous build.

Run 'gmake'? (y/n) y
```

编译完成后进行安装

```
/mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/lrelease_wrapper.sh q
txmlpatterns_ru.ts -qm qtxmlpatterns_ru.qm
Updating 'qtxmlpatterns_ru.qm'...
Generated 455 translation(s) (455 finished and 0 unfinished)
Ignored 25 untranslated source text(s)
/mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/lrelease_wrapper.sh q
txmlpatterns_sk.ts -qm qtxmlpatterns_sk.qm
Updating 'qtxmlpatterns_sk.qm'...
Generated 151 translation(s) (151 finished and 0 unfinished)
Ignored 308 untranslated source text(s)
/mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/lrelease_wrapper.sh q
txmlpatterns_uk.ts -qm qtxmlpatterns_uk.qm
Updating 'qtxmlpatterns_uk.qm'...
Generated 49 translation(s) (49 finished and 0 unfinished)
Ignored 431 untranslated source text(s)
make[2]: Leaving directory '/mnt/workspace/qt/qt-src-5.7.0/qttranslations/transl
ations'
make[1]: Leaving directory '/mnt/workspace/qt/qt-src-5.7.0/qttranslations'
Run 'gmake install'? (y/n) y
```

接下来编译自带的 qttranslations 程序

```
install -m 644 -p /mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/qtx
mlpatterns_pl.qm /mnt/workspace/qt/qt-arm-5.7.0/translations/
install -m 644 -p /mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/qtx
mlpatterns_ru.qm /mnt/workspace/qt/qt-arm-5.7.0/translations/
install -m 644 -p /mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/qtx
mlpatterns_sk.qm /mnt/workspace/qt/qt-arm-5.7.0/translations/
install -m 644 -p /mnt/workspace/qt/qt-src-5.7.0/qttranslations/translations/qtx
mlpatterns_uk.qm /mnt/workspace/qt/qt-arm-5.7.0/translations/
make[2]: Leaving directory '/mnt/workspace/qt/qt-src-5.7.0/qttranslations/transl
ations'
make[1]: Leaving directory '/mnt/workspace/qt/qt-src-5.7.0/qttranslations'
是否编译例子? (y/n) y
```

```
mkdir: cannot create directory '/mnt/workspace/qt/images': File exists
make: *** No targets specified and no makefile found. Stop.
cp: omitting directory '/mnt/workspace/qt/qt-src-5.7.0/qtbase/examples/widgets/a
nimation/animatedtiles'
To create a image? (y/n) y
```

```
80+0 records in
80+0 records out
83886080 bytes (84 MB, 80 MiB) copied, 0.09341 s, 898 MB/s
mke2fs 1.42.13 (17-May-2015)
Discarding device blocks: done
Creating filesystem with 81920 1k blocks and 20480 inodes
Filesystem UUID: fe4b529f-088a-4e3b-afc5-6cf9311837e7
Superblock backups stored on blocks:
      8193, 24577, 40961, 57345, 73729

Allocating group tables: done
Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done
```

```
root@osrc-virtual-machine:/mnt/workspace/qt/scripts#
```

Step5: 编译好后, 将 images 目录下的 animatedtiles、QT5.7.0.image、init.sh 拷贝到 SD 卡中。

Step6: 确保你的 SD 位于 /dev/mmcblk0p1, 鼠标位于 /dev/event0, 这样, 开机将自动启动 QT。

7.2.2 setup_env.sh 批处理文件源码

```
#!/bin/bash
#####
# Qt is a C++ framework for GUI application development. With the release of Qt 5.0,
# Qt no longer contains its own window system implementation. The Qt Platform
```

```
# Abstraction (QPA) provides multiple platform plugins that are potentially usable
# on Embedded Linux systems: EGLFS, LinuxFB, DirectFB, Wayland. In this tutorial we
# configure Qt to use the LinuxFB plugin which is well suited for embedded devices
# without GPU running Linux.
#####
#
# 提示使用 source 命令
#
# source 命令也称为“点命令”，也就是一个点符号（.）；通常用于重新执行刚修改的初始
# 化文件，使之立即生效，而不必注销并重新登录。
if [ $BASH_SOURCE == $0 ];then
    echo -e "\nThis script should be sourced, like \"source ${0##*/}\".\n"
    exit 1
fi

# 脚本所在目录
export SCRIPTSDIR="$(dirname $(readlink -f $0))"
# 项目根目录
export SRCROOT="$(dirname $SCRIPTSDIR)"
# 方便在各个地方执行 scripts 里的脚本
export PATH=${SCRIPTSDIR}:$PATH

# QT 版本号
export QT_MAJOR_VERSION=5
export QT_MINOR_VERSION=7
export QT_PATCH_VERSION=0
export
QT_VERSION=${QT_MAJOR_VERSION}.${QT_MINOR_VERSION}.${QT_PATCH_VERSION}

# 交叉编译器
export PATH=${SRCROOT}/cross_compilers/bin:$PATH
# for Zynq-7000 (Linaro - hard float)
# if which arm-linux-gnueabihf-gcc >/dev/null; then
#     export CROSS_COMPILE=arm-linux-gnueabihf-
# for Zynq-7000 (CodeSourcery - soft float)
# elif which arm-xilinx-linux-gnueabi-gcc >/dev/null; then
#     export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
# else
#     echo "无法找到交叉编译器！！！"
#     return 1
# fi

# 资源包
```

```
export PACKAGES=${SRCROOT}/packages

# 镜像
export IMAGES=${SRCROOT}/images

# QT 源码目录
export QTDIR=${SRCROOT}/qt-arm-${QT_VERSION}
export PATH=${QTDIR}/bin:$PATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${QTDIR}/lib

# APP
export QTAPP=${SRCROOT}/app

# Invoke a second make in the output directory, passing relevant variables
# check that the output directory actually exists
mkdir -p ${PACKAGES} ${IMAGES} ${QTDIR}
```

7.2.3 get_qt_sources.sh 批处理文件源码

```
#!/bin/bash
#####
# 文件名: get_qt_sources.sh
# 功 能: 获取 QT 源码
# 版本号: V 0.2
#####
source echo_color.sh
#
# 提示使用 source 命令
#
# source 命令也称为“点命令”，也就是一个点符号(.)；通常用于重新执行刚修改的初始化文件，使之立即生效，而不必注销并重新登录。
if [ ! "${CROSS_COMPILE}" ];then
    echo_red "请切换到 script 目录下，执行 'source setup_env.sh' 命令来配置环境变量。"
    exit 1
fi

# 准备开发环境
# Download the Qt sources and extract the archive to your Qt build area.
```

```
tar zxvf ${PACKAGES}/qt-everywhere-opensource-src-${QT_VERSION}.tar.gz  
mv ./qt-everywhere-opensource-src-${QT_VERSION} ${QTSRC}  
  
# 2. Prepare a mkspec  
#  
# Before we can do the configuration for the target system, we will need a set  
# of mkspecs that tells qmake which tool chain it should reference when it creates  
# the Makefiles. In this example we will provide an mkspec to go along with the  
# ARM GNU tools.  
cp ${PACKAGES}/xlnx-gnueabi-g++ ${QTSRC}/qtbase/mkspecs/ -R  
cp ${PACKAGES}/xlnx-gnueabihf-g++ ${QTSRC}/qtbase/mkspecs/ -R
```

7.2.4 mk_qt_img.sh 批处理文件源码

```
#!/bin/bash  
#  
# sudo apt-get install build-essential perl python git  
#  
# http://doc.qt.io/  
# http://doc.qt.io/qt-5/embedded-linux.html  
# http://www.it165.net/embed/html/201606/3507.html  
#  
#####  
# Qt is a C++ framework for GUI application development. With the release of Qt 5.0,  
# Qt no longer contains its own window system implementation. The Qt Platform  
# Abstraction (QPA) provides multiple platform plugins that are potentially usable  
# on Embedded Linux systems: EGLFS, LinuxFB, DirectFB, Wayland. In this tutorial we  
# configure Qt to use the LinuxFB plugin which is well suited for embedded devices  
# without GPU running Linux.  
#####  
source echo_color.sh  
#  
# 提示使用 source 命令  
#  
# source 命令也称为“点命令”，也就是一个点符号（.）；通常用于重新执行刚修改的初始  
# 化文件，使之立即生效，而不必注销并重新登录。  
if [ ! "${CROSS_COMPILE}" ];then  
    echo_red "请切换到 script 目录下，执行 'source setup_env.sh' 命令来配置环境变量。"  
    exit 1  
fi  
# 检查源码是否存在  
if [ ! -d "${QTSRC}" ]; then
```

```
echo_red "请执行 'get_qt_sources.sh' 下载所需源码。"
exit 1
else
    cd ${QTDIR}
fi

# 1. To reconfigure, run 'gmake confclean' and 'configure'.
# 注意：仅在重新配置里需要执行此步骤
read -p "Run 'gmake confclean'? (y/n) " REPLY
if [ "$REPLY" == "y" ]; then
    make distclean
    rm ${QTDIR}/* -r
fi

# 2. Configure the Target Build
read -p "Configure the Target Build (y/n) " REPLY
if [ "$REPLY" == "y" ]; then
    # 请根据实际需要修改些配置
    ./configure -opensource -confirm-license -verbose -release \
        -xplatform xlnx-gnueabi-g++ -prefix ${QTDIR} \
        -nomake examples \
        -no-tslib \
        -no-directfb \
        -no-opengl \
        -no-eglfs \
        -no-xcb \
        -no-iconv \
        -qt-libpng \
        -qt-libjpeg \
        -qt-freetype \
        -skip virtualkeyboard \
        -skip qtdoc
fi

# 3. Run make to build the cross-compiled target version of Qt.
read -p "Run 'gmake'? (y/n) " REPLY
if [ "$REPLY" == "y" ]; then
    make
fi

# 4. Once the build has completed it is time to install Qt. You may need to su
# to root to do this part depending upon what prefix you configured the build with.
read -p "Run 'gmake install'? (y/n) " REPLY
```

```
if [ "$REPLY" == "y" ]; then
    sudo make install
    # 如果根文件系统中缺少 c++库, 需要补全该库
    cp -P ${SRCROOT}/cross_compilers/arm-xilinx-linux-gnueabi/libc/usr/lib/libstdc++.so* $QTDIR/lib

fi

# 5. 编译 QT 源码中自带的例子
read -p "是否编译例子? (y/n) " REPLY
if [ "$REPLY" == "y" ]; then
    mkdir ${IMAGES}
    cd ${QTSRC}/qtbase/examples/widgets/animation/animatedtiles
    make
    cp ${QTSRC}/qtbase/examples/widgets/animation/animatedtiles ${IMAGES}
fi
#####
# 编译完 Qt 后, 只需将生成的 lib 和 plugins 文件夹拷贝到开发板, 另外, 当在嵌入式 Linux
# 平台上运行应用程序前, 应根据自己平台的实际情况提前设置好下面几个环境变量:
#
# export QT_QPA_PLATFORM_PLUGIN_PATH=/opt/Qt-5.3.2/armv7-a/plugins/platforms
# export QT_QPA_PLATFORM=linuxfb:tty=/dev/fb0
# export QT_QPA_FONTDIR=/opt/Qt-5.3.2/armv7-a/lib/fonts
# export QT_QPA_GENERIC_PLUGINS=tslib:/dev/touchscreen-1wire #使用 tslib 插件
#
# 然后就可以运行 Qt 程序了
#####

# 6. Create a File System Image with Pre-Compiled Qt/Qwt Libraries
# http://www.wiki.xilinx.com/Zynq+Qt+and+Qwt+Base+Libraries-Build+Instructions
read -p "To create a image? (y/n) " REPLY
if [ "$REPLY" == "y" ]; then

    TARGET=${IMAGES}/QT${QT_VERSION}.image
    TEMP=${SRCROOT}/temp
    mkdir -p ${TEMP}

    # 制作一个 80M 的镜像文件, 请根据实际情况修改
    dd if=/dev/zero of=${TARGET} bs=1M count=80

    mkfs.ext4 -F ${TARGET} -L "QT"
    chmod a+rwx ${TARGET}
    sudo mount -o loop ${TARGET} ${TEMP}

    cp -r ${QTDIR}/lib ${TEMP}
```

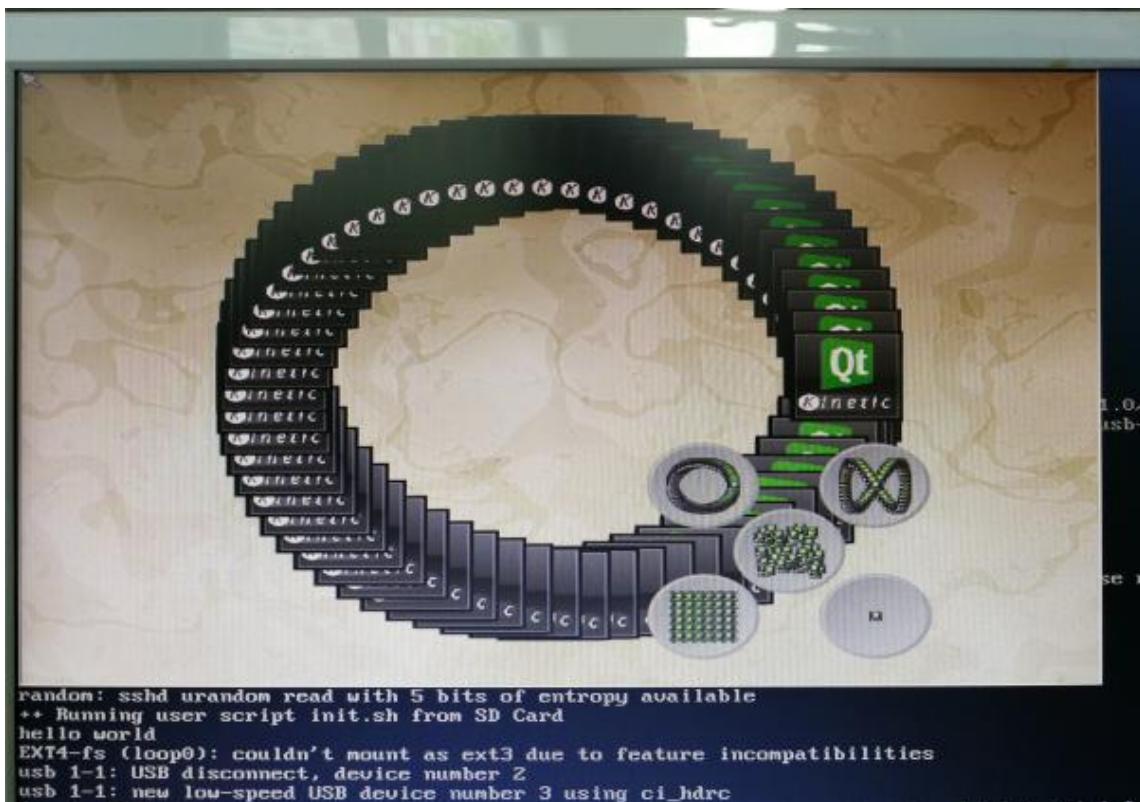
```
cp -r ${QTDIR}/plugins ${TEMP}  
  
umount ${TEMP}  
  
fi
```

7.2.4 init.sh 文件

```
#!/bin/sh  
  
echo "hello world"  
  
export QTDIR=/opt/QT5.7  
mkdir -p ${QTDIR}  
  
mount -o loop /mnt/QT5.7.0.image ${QTDIR}  
  
export QT_QPA_FONTDIR=${QTDIR}/lib/fonts  
export QT_QPA_PLATFORM_PLUGIN_PATH=${QTDIR}/plugins/  
export LD_LIBRARY_PATH=${QTDIR}/lib:$LD_LIBRARY_PATH  
  
export QT_QPA_PLATFORM=linuxfb:fb=/dev/fb0  
  
#http://www.360doc.com/content/14/1215/10/18578054_433033534.shtml  
export QT_QPA_EVDEV_MOUSE_PARAMETERS=evdevmouse:/dev/event0  
  
/mnt/animatedtiles &
```

7.2.5 测试结果

编译好后，将 images 目录下的 animatedtiles、QT5.7.0.image、init.sh 拷贝到 SD 卡中。确保你的 SD 位于/dev/mmcblk0p1，鼠标位于/dev/event0，这样，开机将自动启动 QT。



```
random: sshd urandom read with 5 bits of entropy available
++ Running user script init.sh from SD Card
hello world
EXT4-fs (loop0): couldn't mount as ext3 due to feature incompatibilities
usb 1-1: USB disconnect, device number 2
usb 1-1: new low-speed USB device number 3 using cl_hdrc
```

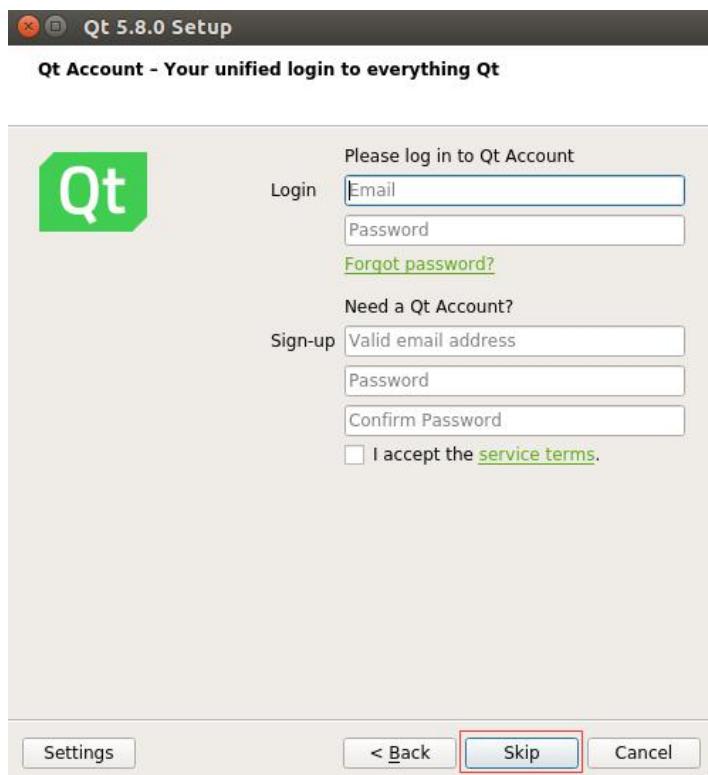
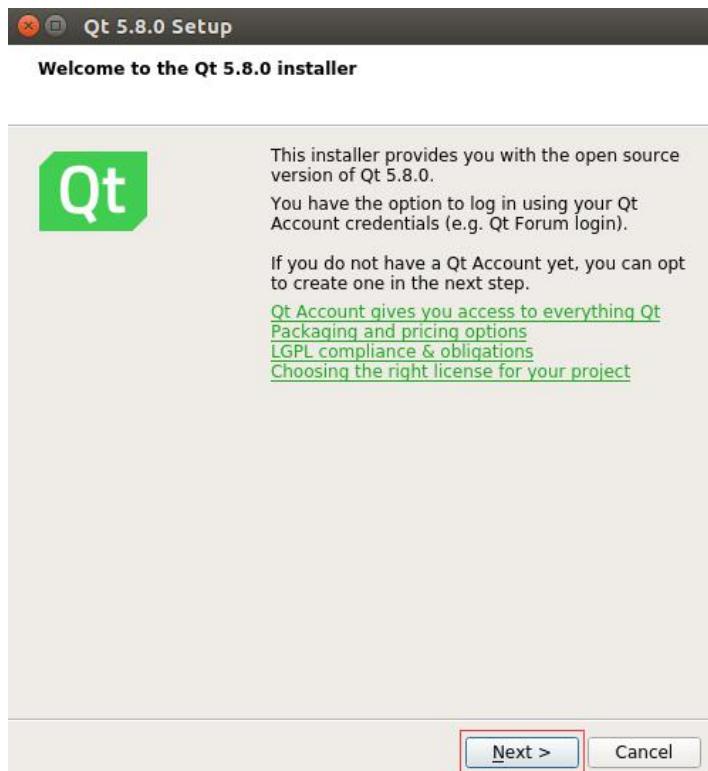
7.3 在 PC 端 LINUX 安装 qt5.8.0

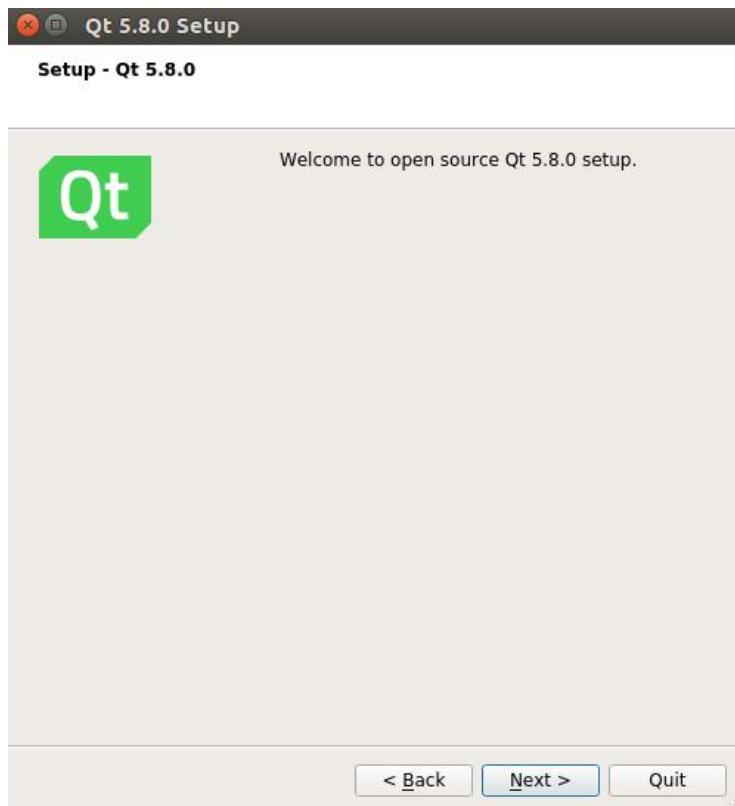
Step1:进入 mnt/workspace/packages 文件夹下

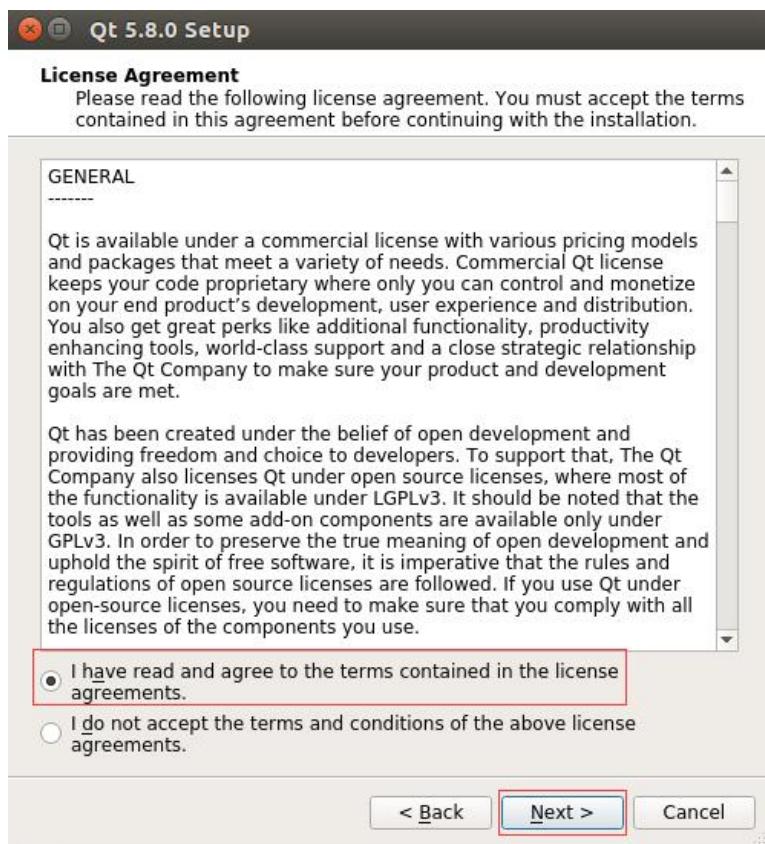
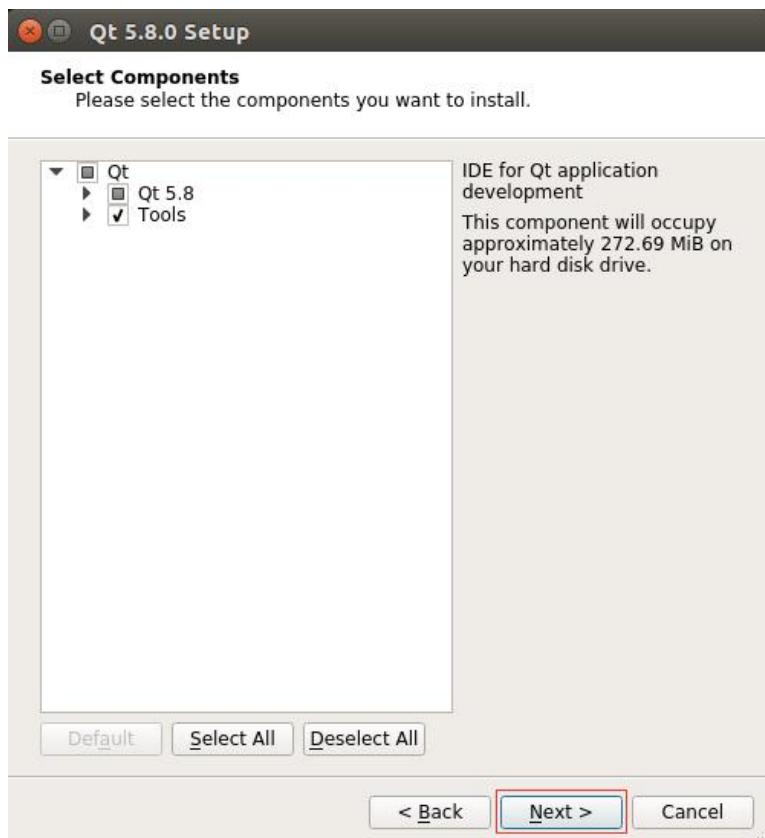
Step2: 执行 chmod +x qt-opensource-linux-x64-5.8.0.run

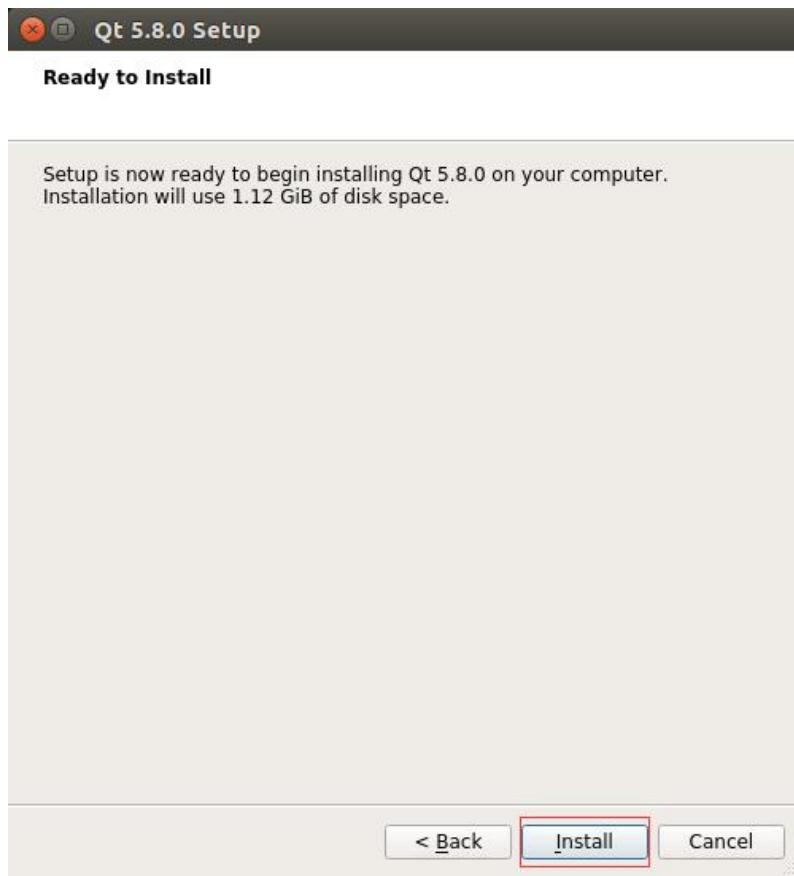
Step3: 执行./qt-opensource-linux-x64-5.8.0.run

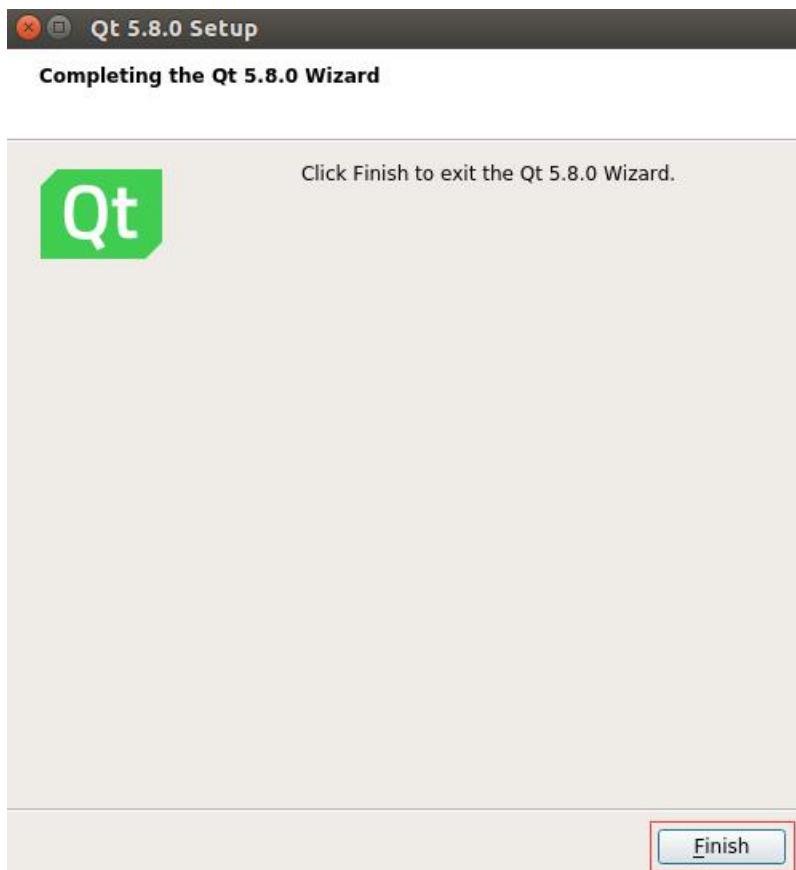
```
root@osrc-virtual-machine:/mnt/workspace/qt/scripts# cd ..../packages
root@osrc-virtual-machine:/mnt/workspace/qt/packages# chmod +x qt-opensource-lin
ux-x64-5.8.0.run
root@osrc-virtual-machine:/mnt/workspace/qt/packages# ./qt-opensource-linux-x64-
5.8.0.run
```









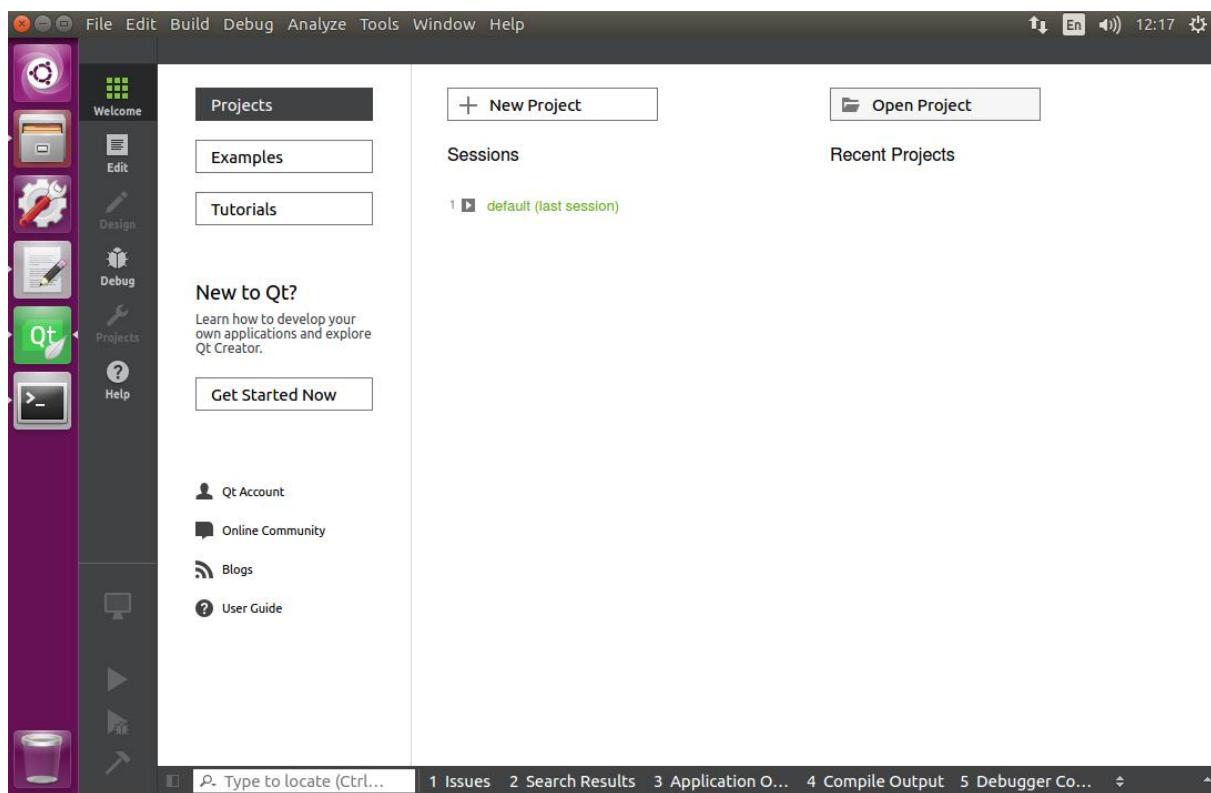


安装完成

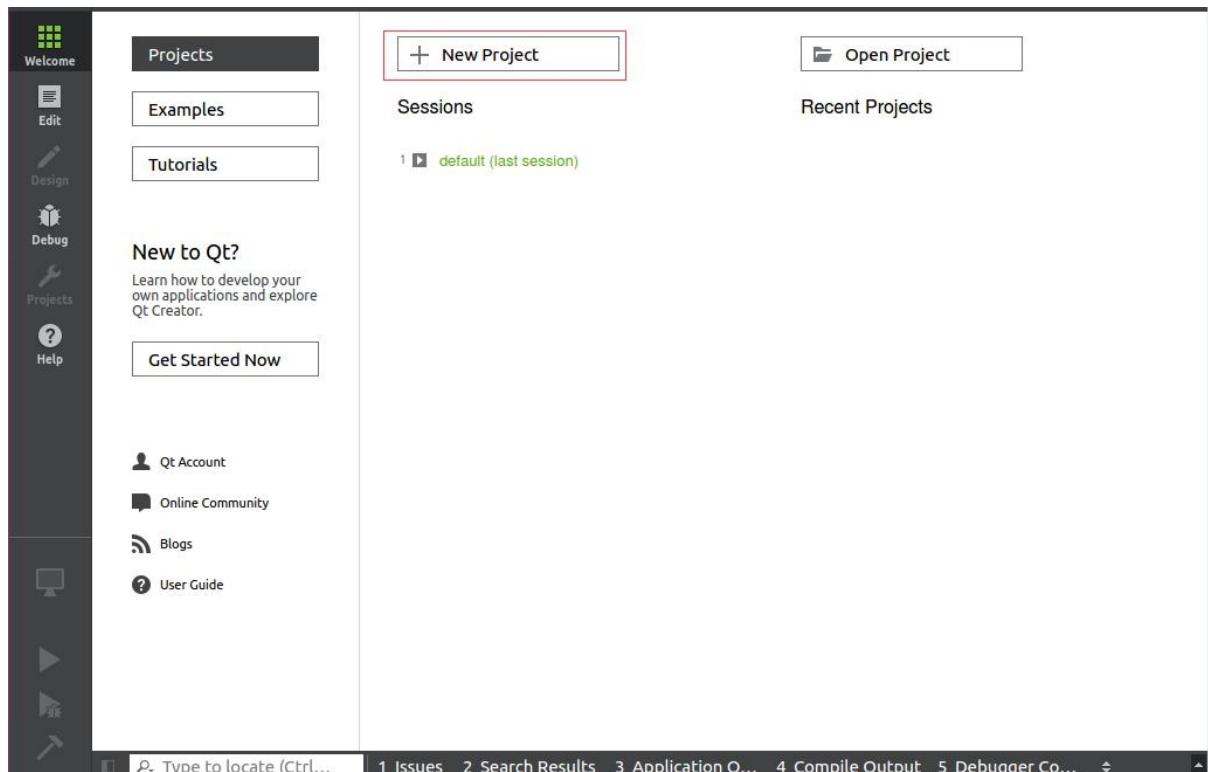
7.4 QtE LINUX PC 端创建工程

Step1: 双击 qtcreator 启动 qte

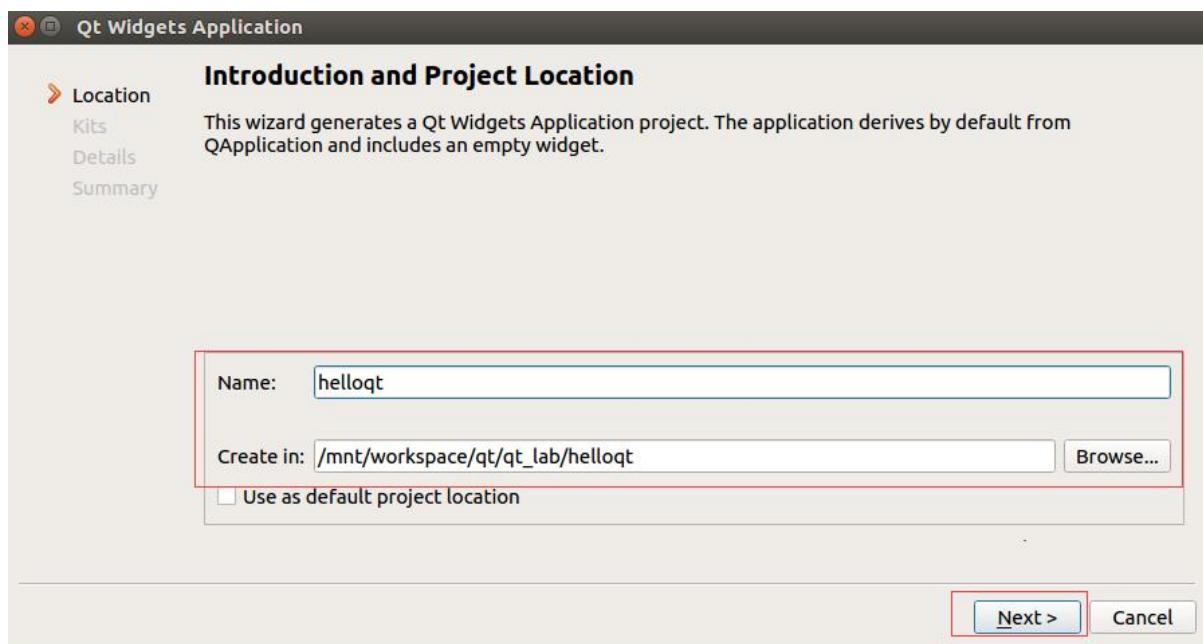
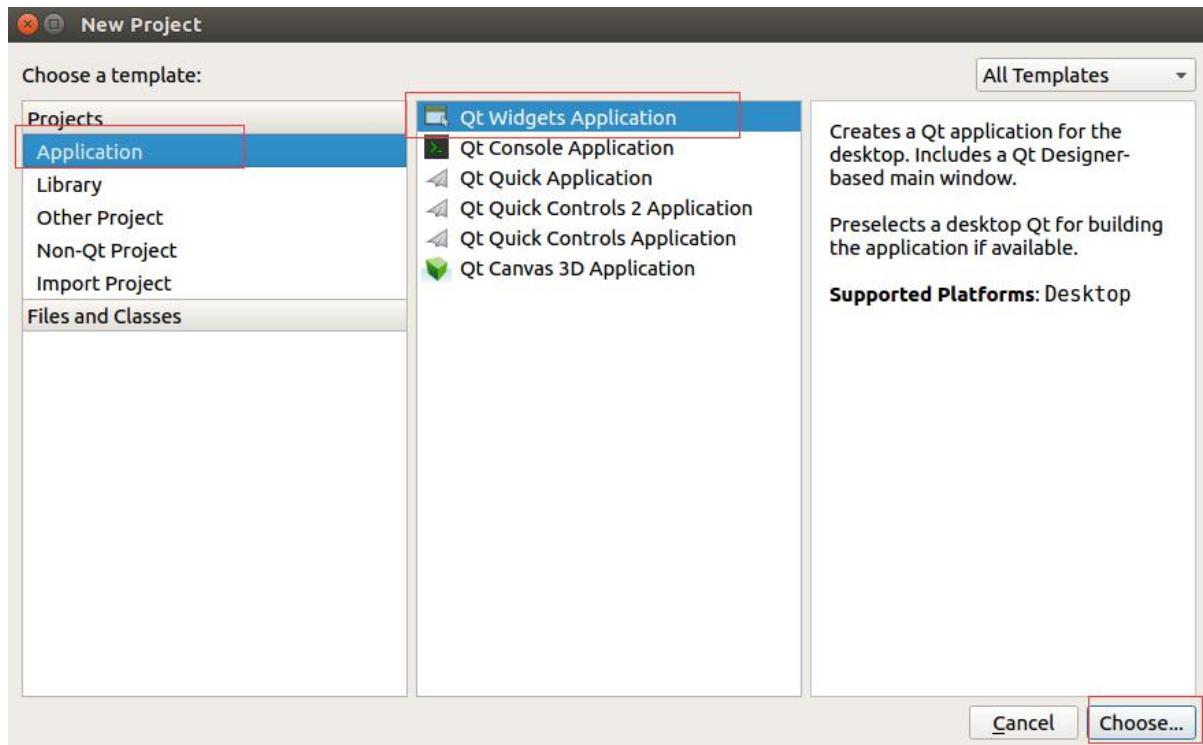




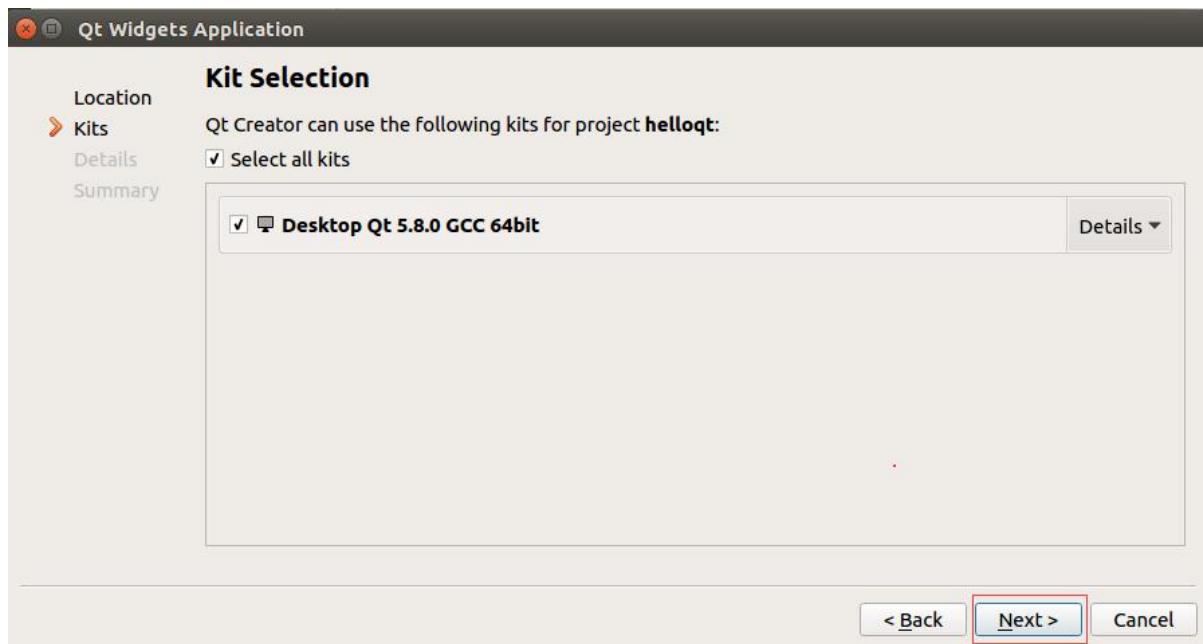
Step2:单击 New Project 新建一个 qt 工程



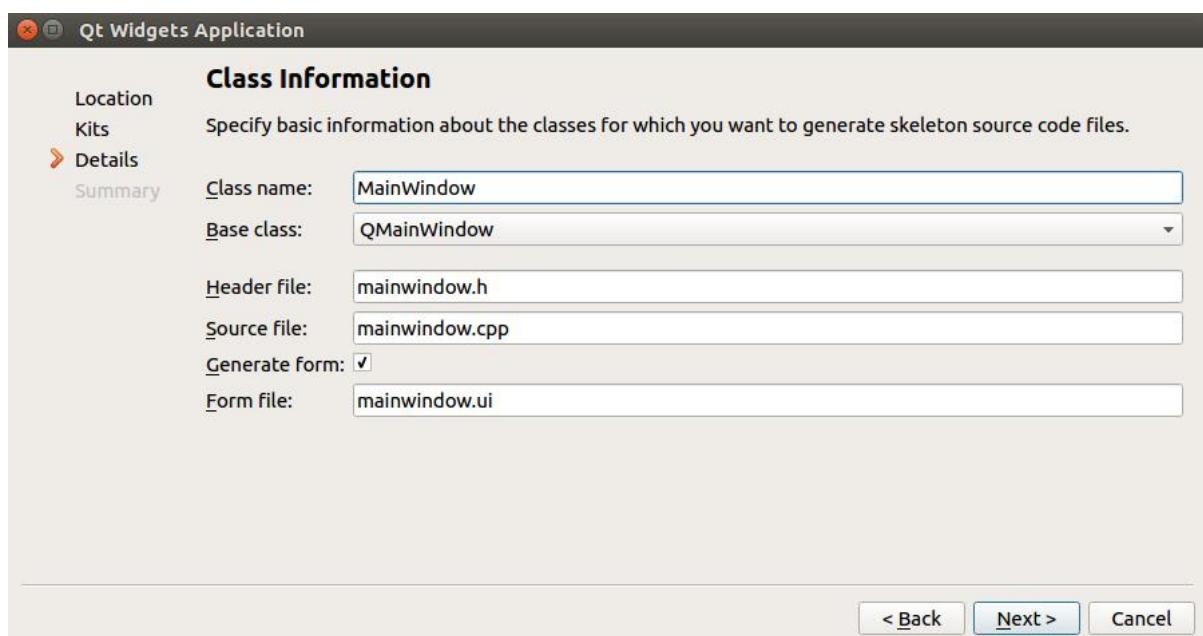
Step3:设置路径和工程名字



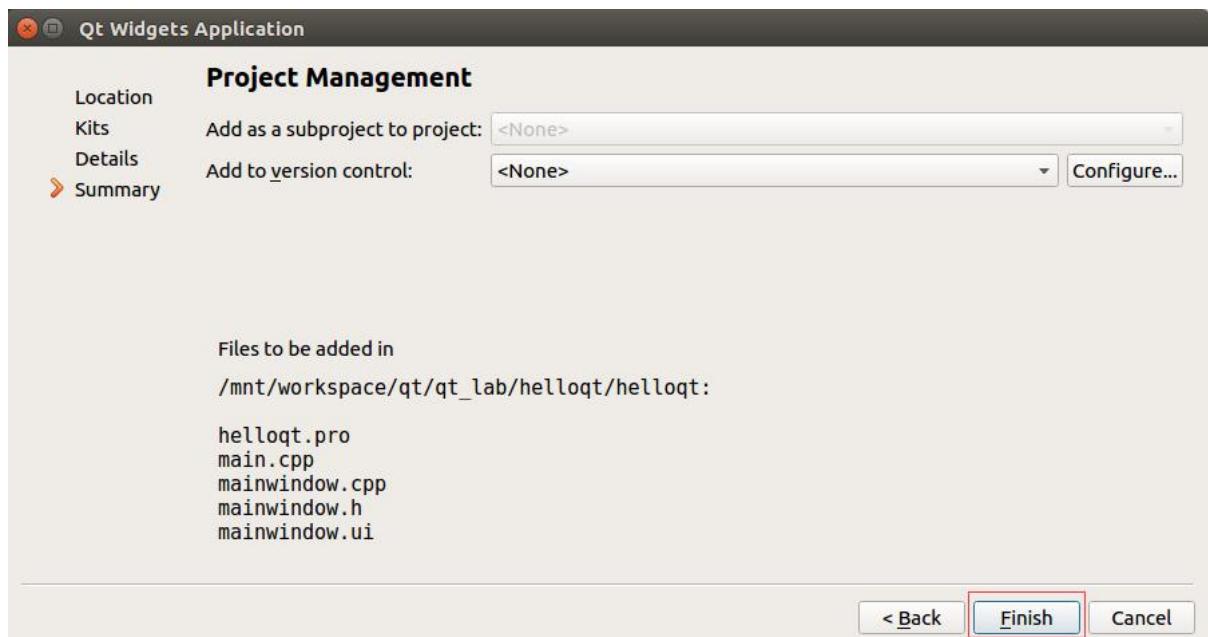
Step4:单击 NEXT



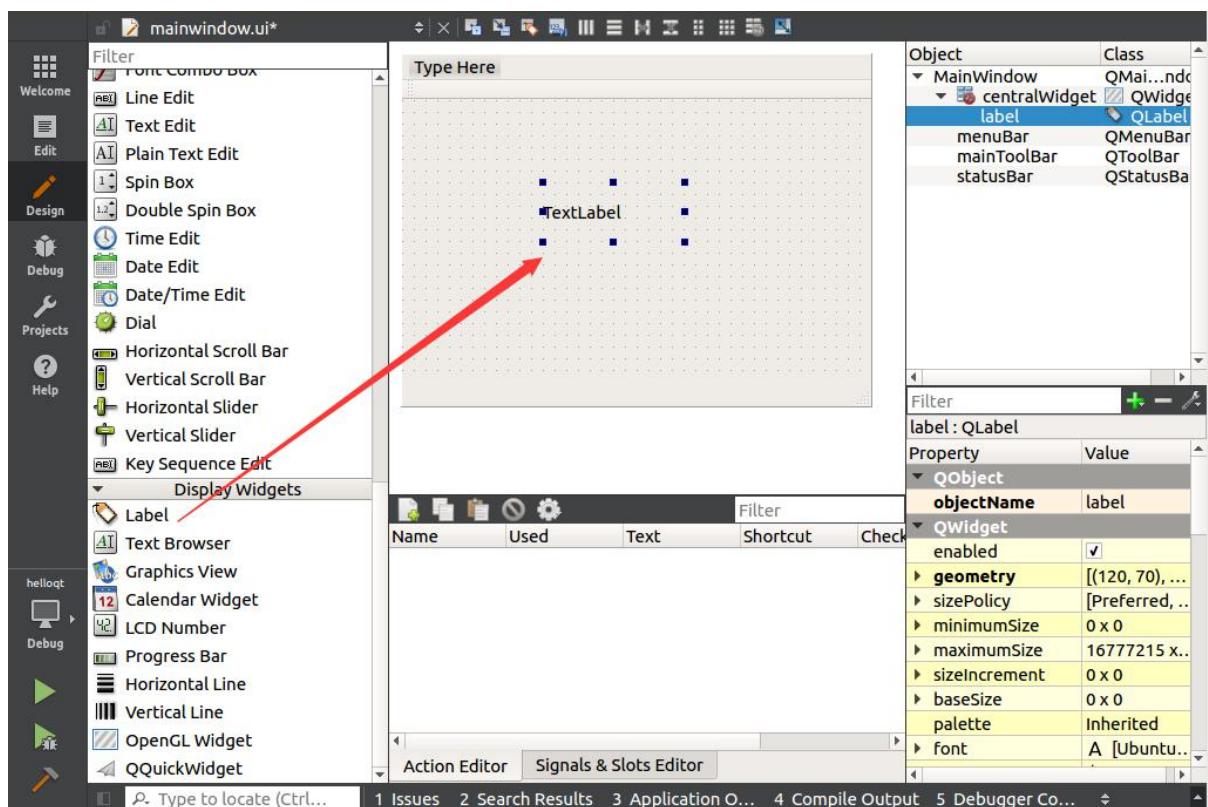
Step5:单击 NEXT



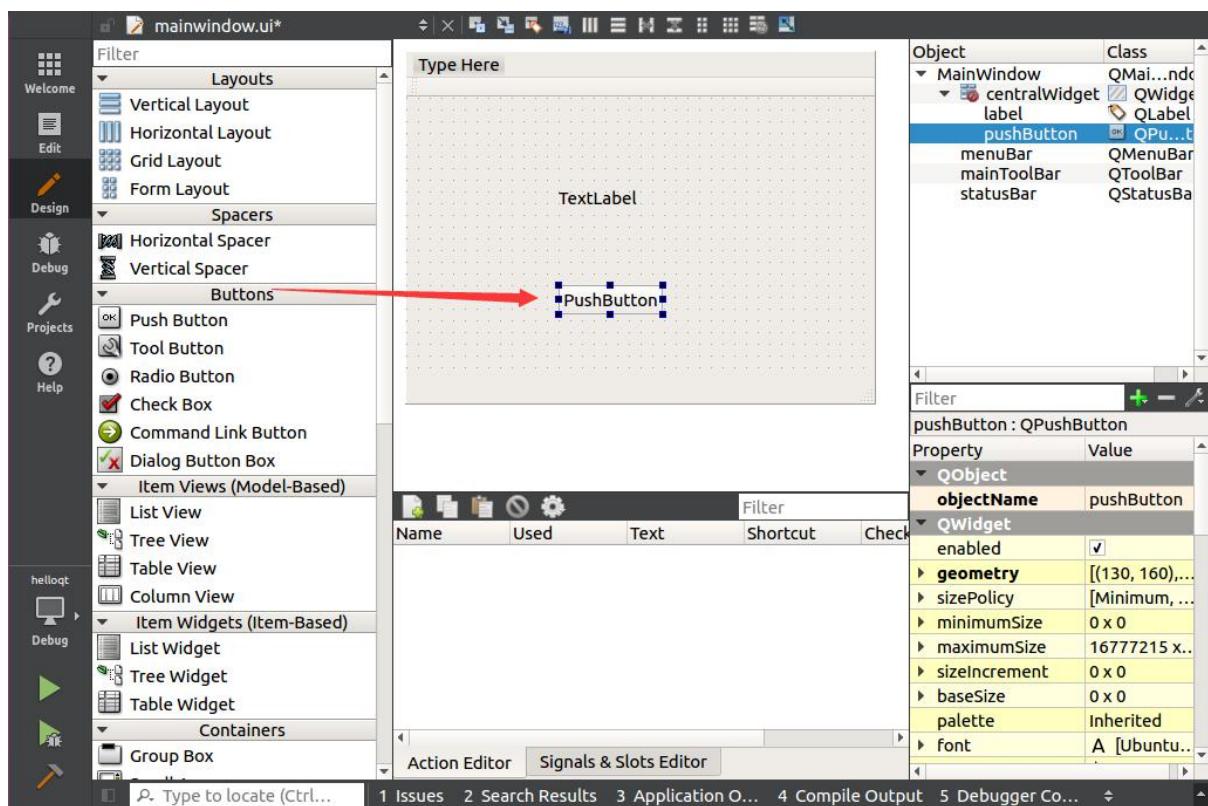
Step6:单击 Finish



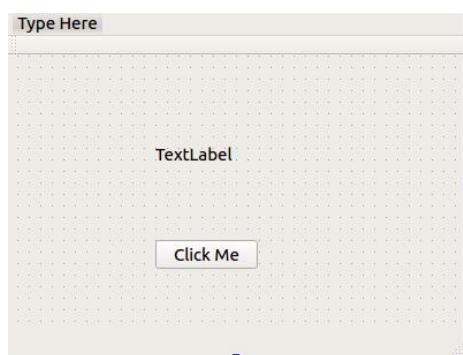
Step7: 拖动 Label 标签控件到窗口



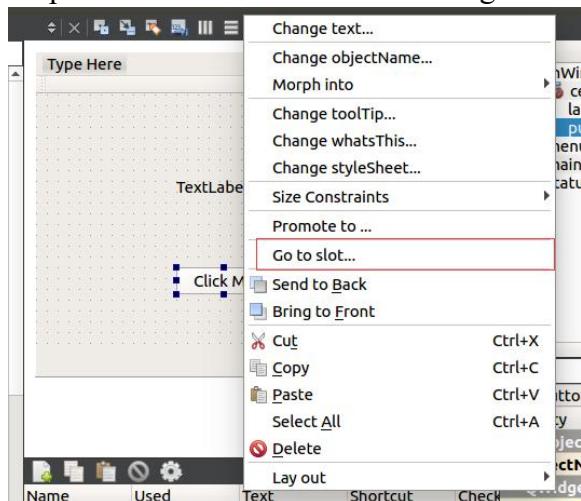
Step8: 拖动 Push Button 控件到窗口



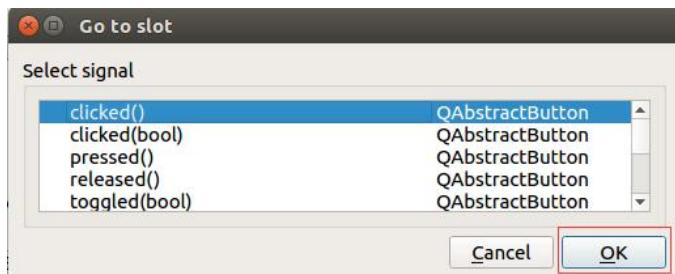
Step9:右击 PushButton 控件修改控件显示文本为



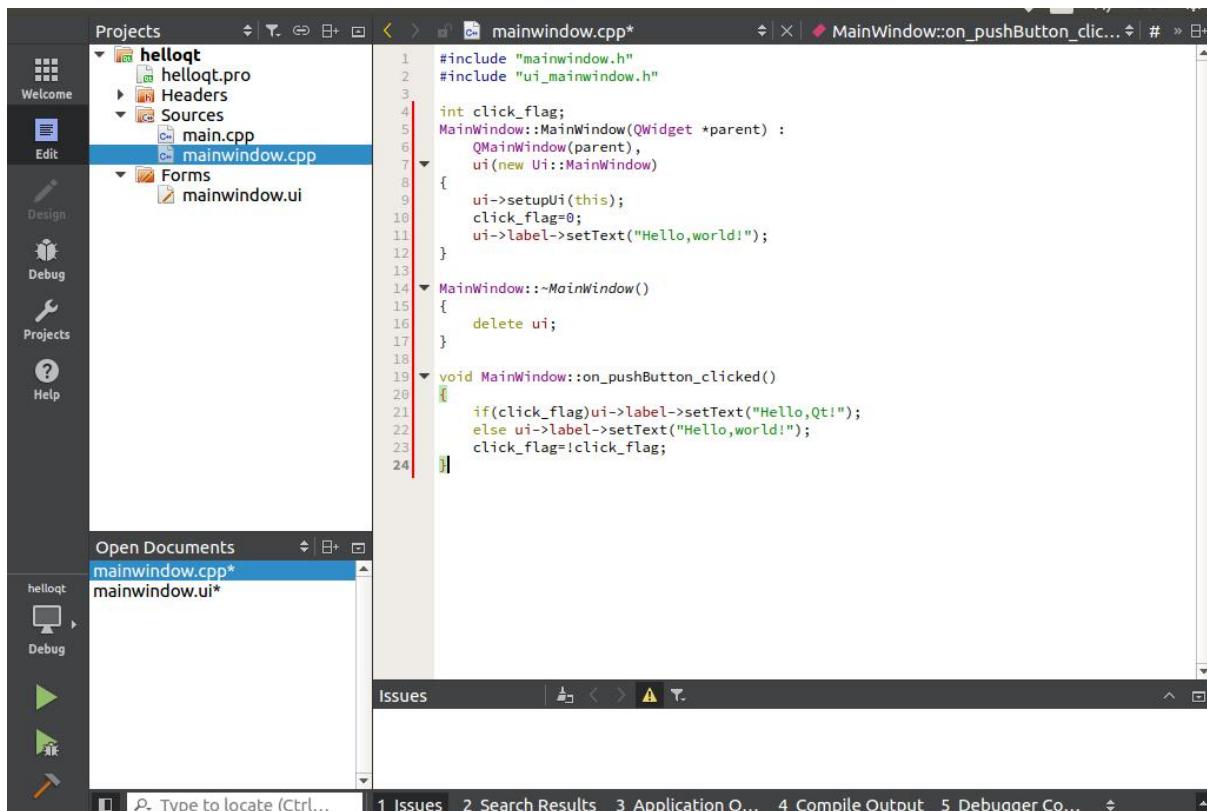
Step10:继续右击 PushButton 选择 goto slot



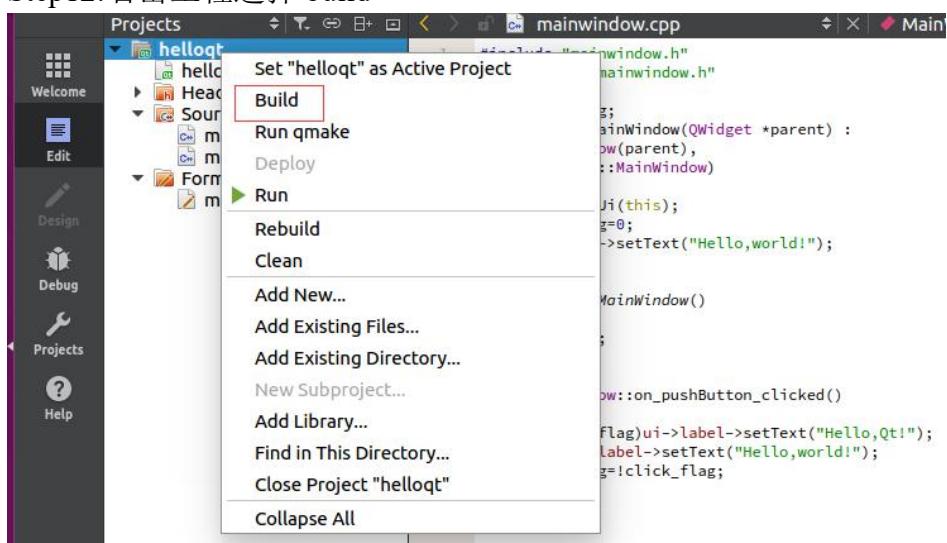
Step11:选择相应单击事件



Step11:为控件添加如下代码



Step12:右击工程选择 build



Step13: 提示错误信息



```

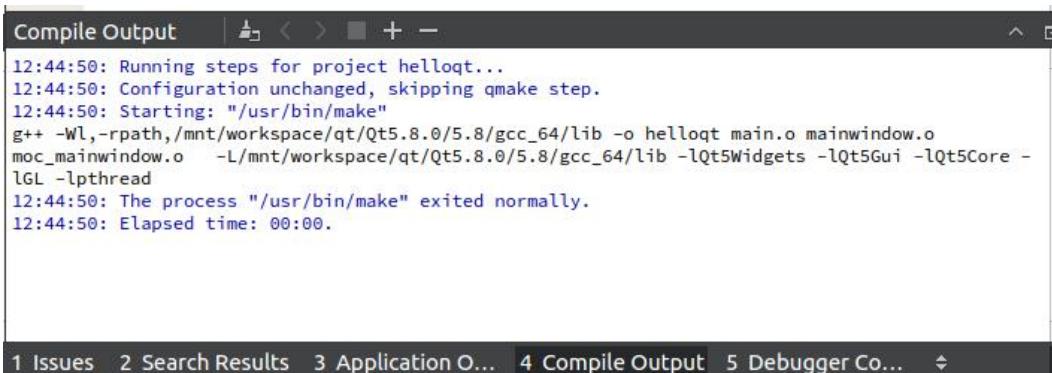
Compile Output | ↻ ⟲ ⟳ ⟷ + -
12:41:56: Starting: "/usr/bin/make"
g++ -Wl,-rpath,/mnt/workspace/qt/Qt5.8.0/5.8/gcc_64/lib -o helloqt main.o mainwindow.o
moc_mainwindow.o -L/mnt/workspace/qt/Qt5.8.0/5.8/gcc_64/lib -lQt5Widgets -lQt5Gui -lQt5Core -
lGL -lpthread
/usr/bin/ld: cannot find -lGL
Makefile:236: recipe for target 'helloqt' failed
collect2: error: ld returned 1 exit status
make: *** [helloqt] Error 1
12:41:56: The process "/usr/bin/make" exited with code 2.
Error while building/deploying project helloqt (kit: Desktop Qt 5.8.0 GCC 64bit)
When executing step "Make"
12:41:56: Elapsed time: 00:00.

```

这是由于缺少必要的库导致，在控制台输入：

`sudo apt-get install libgl1-mesa-dev`

再次编译就可以通过

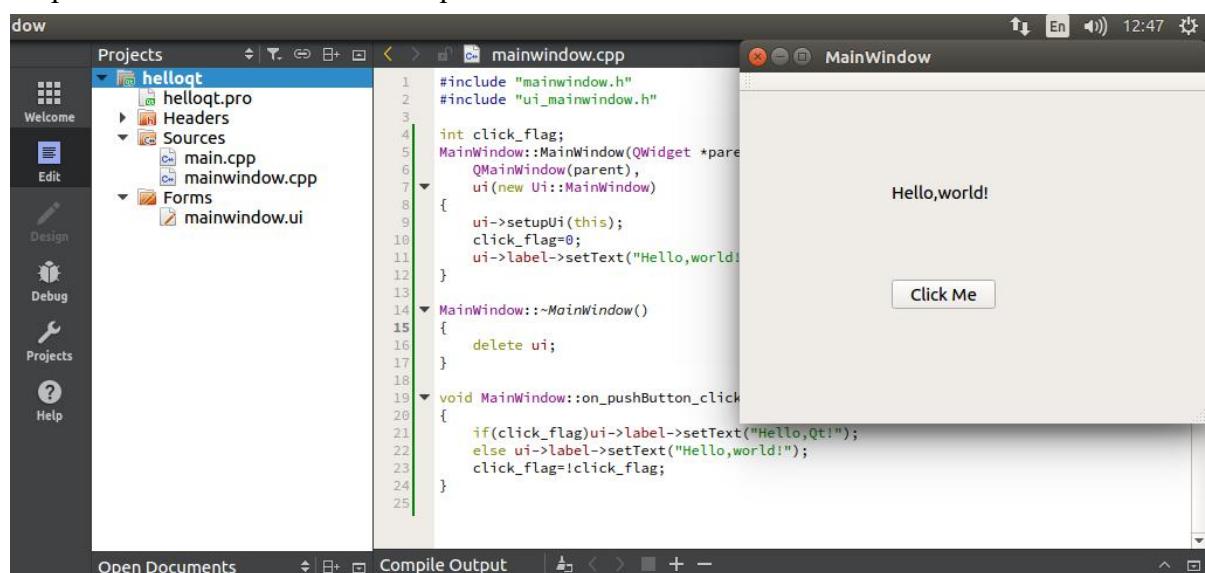


```

Compile Output | ↻ ⟲ ⟳ ⟷ + -
12:44:50: Running steps for project helloqt...
12:44:50: Configuration unchanged, skipping qmake step.
12:44:50: Starting: "/usr/bin/make"
g++ -Wl,-rpath,/mnt/workspace/qt/Qt5.8.0/5.8/gcc_64/lib -o helloqt main.o mainwindow.o
moc_mainwindow.o -L/mnt/workspace/qt/Qt5.8.0/5.8/gcc_64/lib -lQt5Widgets -lQt5Gui -lQt5Core -
lGL -lpthread
12:44:50: The process "/usr/bin/make" exited normally.
12:44:50: Elapsed time: 00:00.

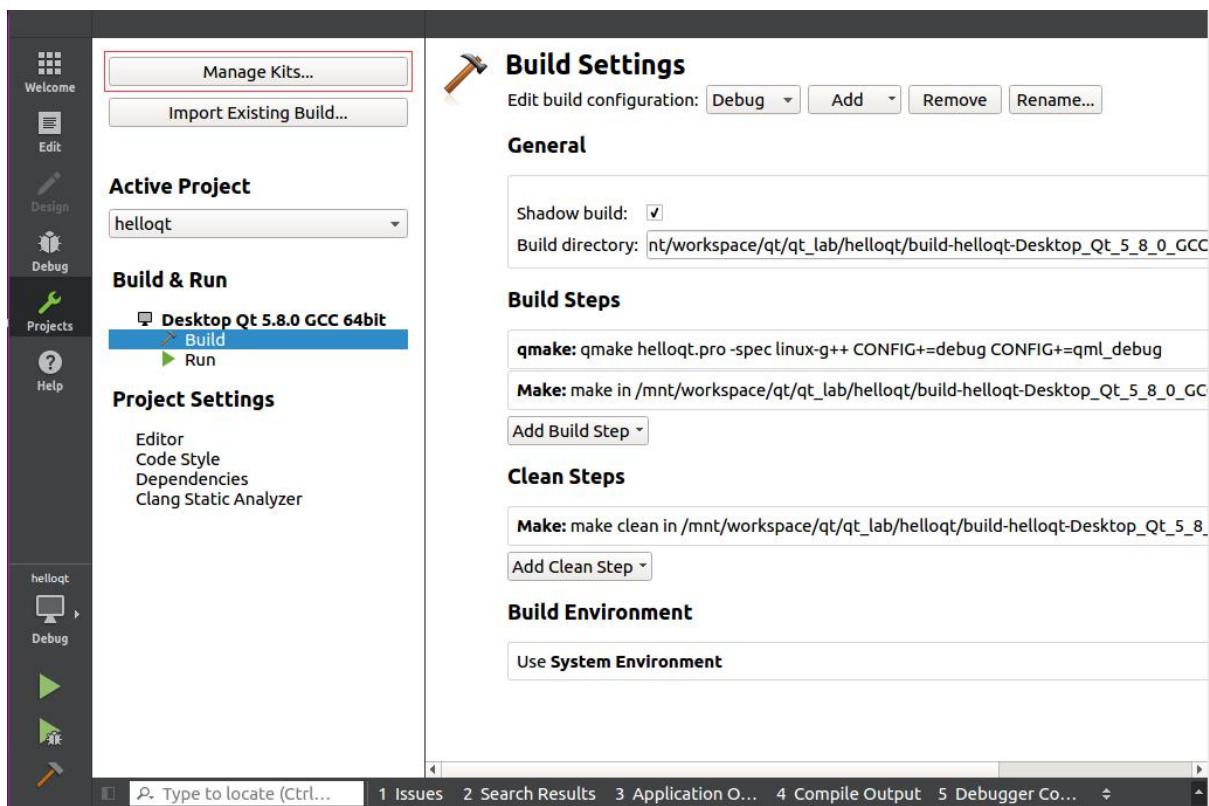
```

Step14: 右击工程选择 run 运行 qt 界面

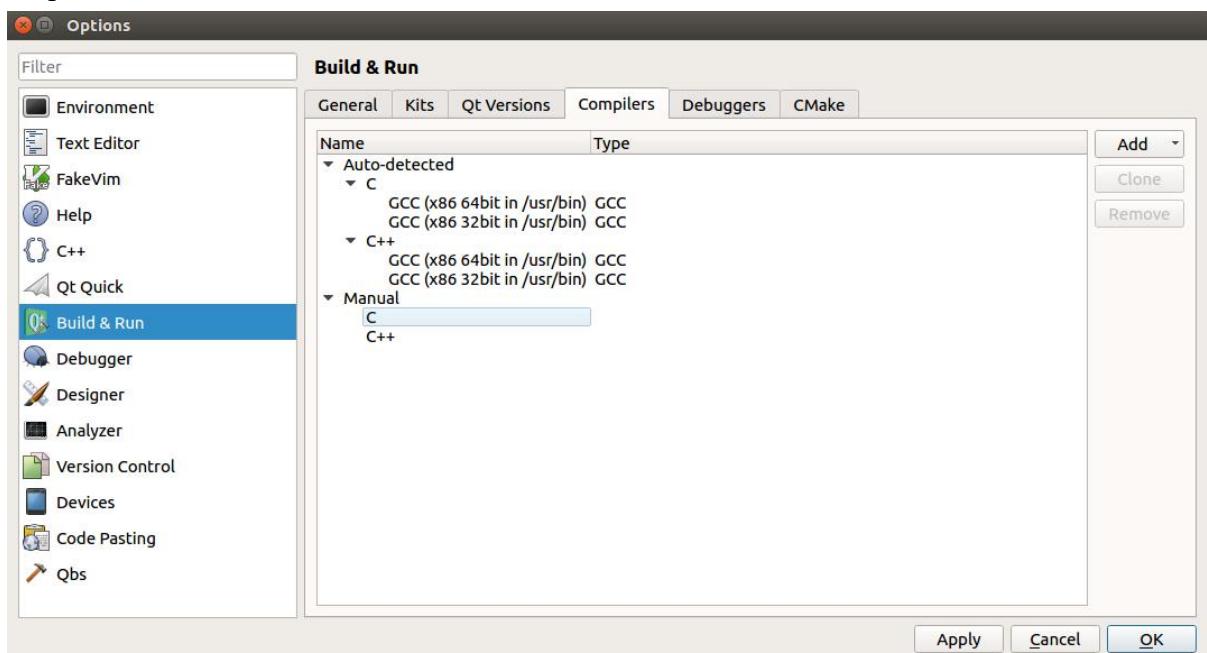


7.5 对 QtE 设置交叉编译

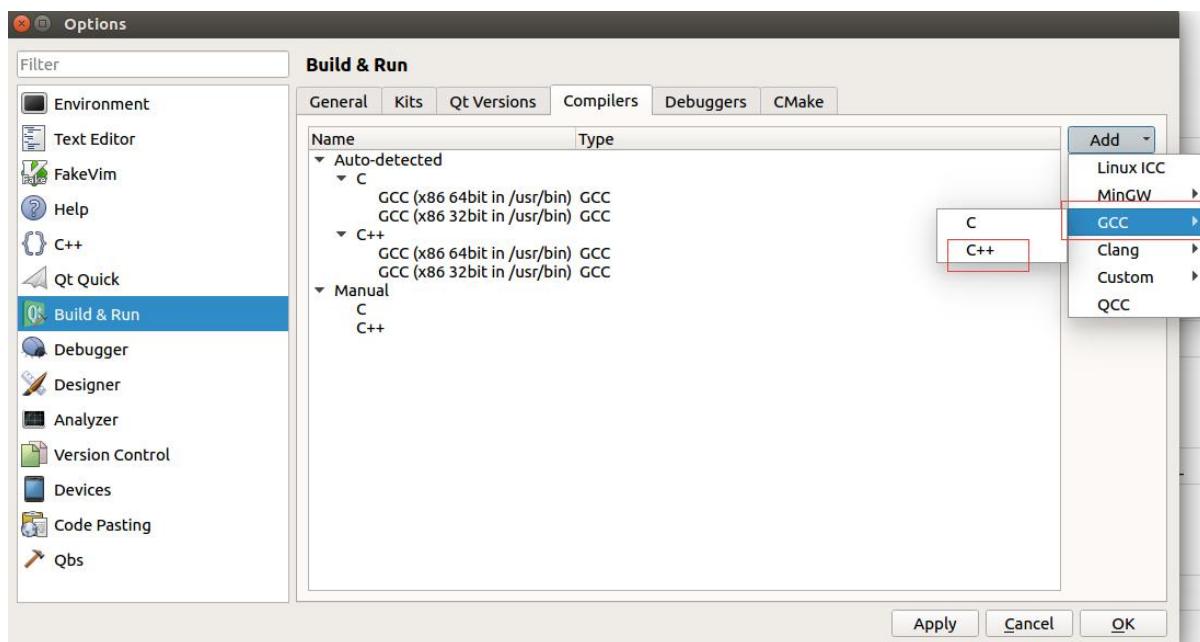
Step1: 选择 Manage Kits



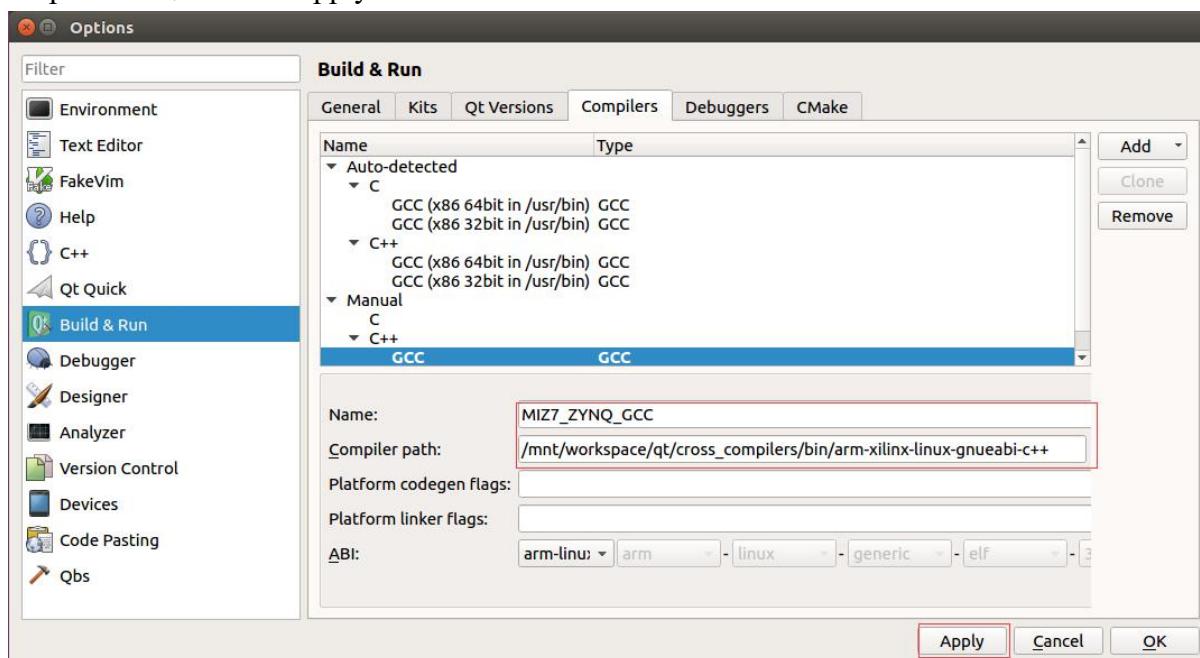
Step2:设置交叉编译器



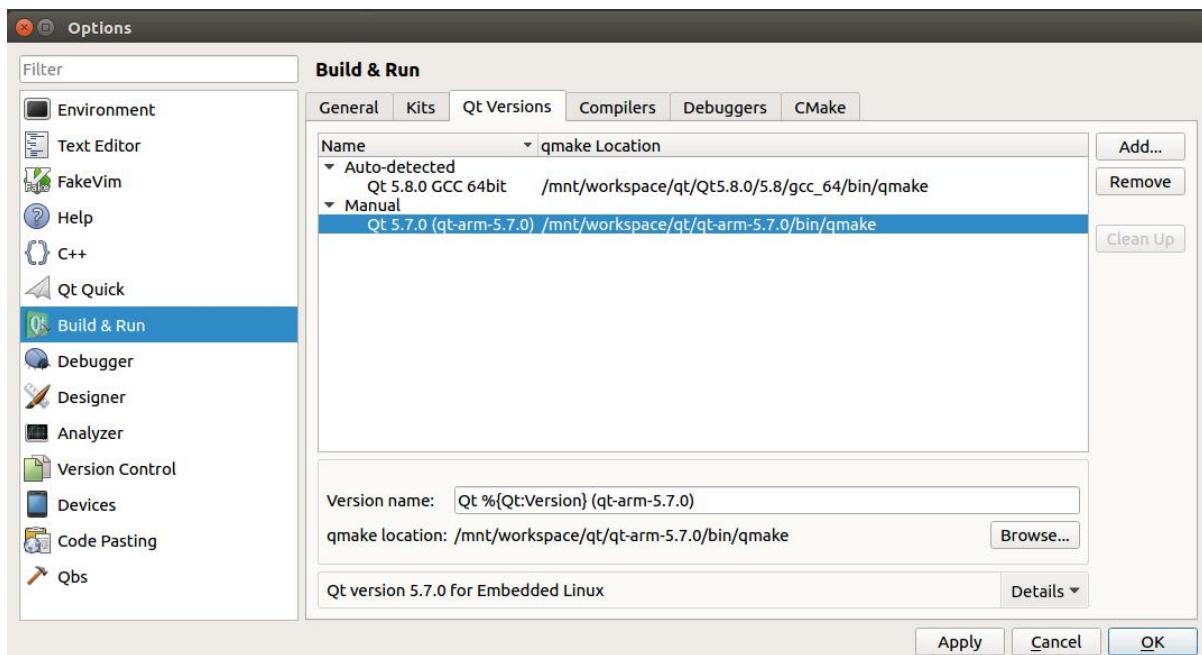
Step3:设置 C++编译器



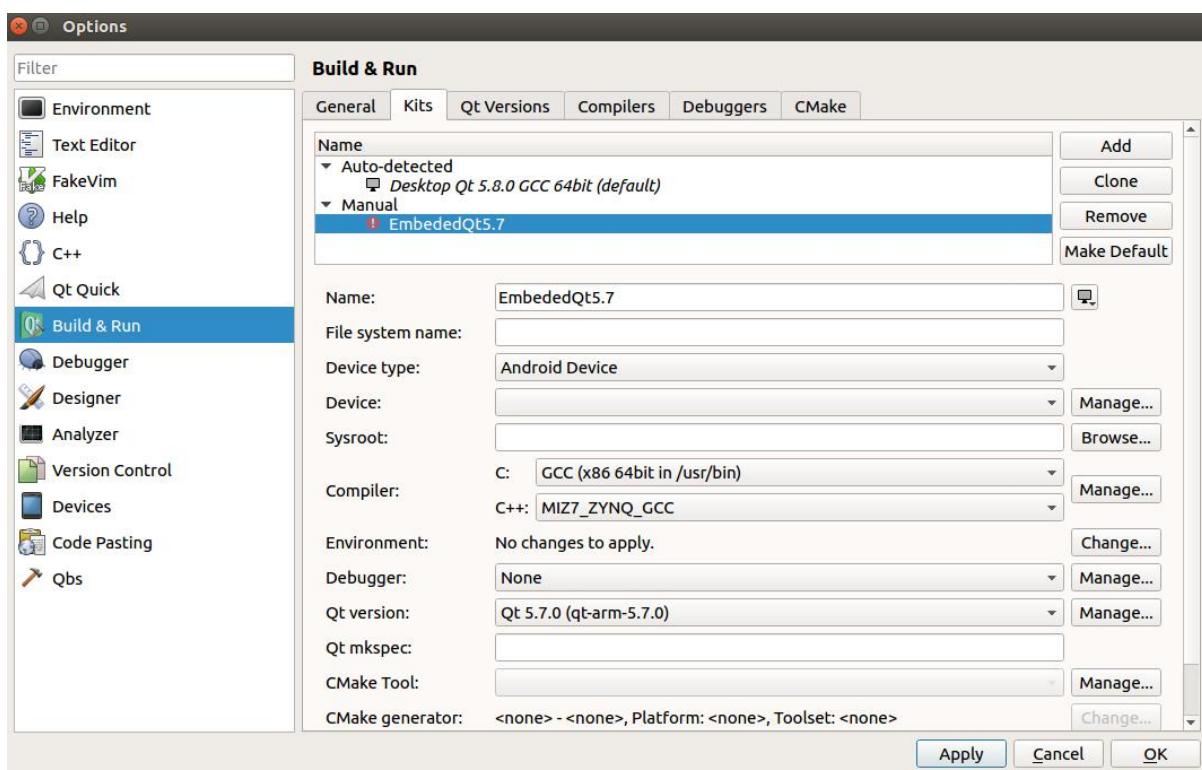
Step4:设置好后单击 Apply



Step5:设置 Qt Versions

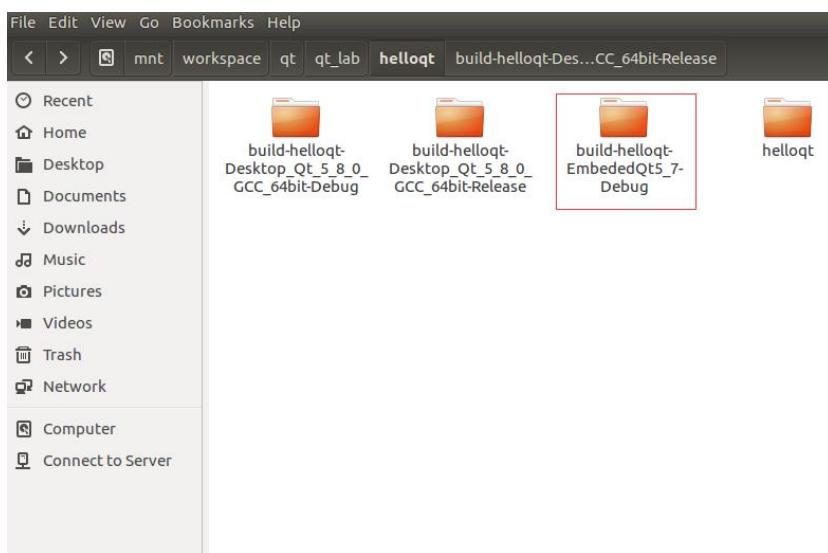


Step6:设置 Qt Versions



Step7:单击 OK

Step8:右击工程再次编译后可以看到如下文件夹下增加了基于嵌入式 LINUX 系统的可执行文件



Step8: 复制 helloqt 到 SD 卡并且修改 init.sh 文件的开机自动启动为 helloqt
Init.sh 文件修改如下

```
#!/bin/sh

echo "hello world"

export QTDIR=/opt/QT5.7
mkdir -p ${QTDIR}

mount -o loop /mnt/QT5.7.0.image ${QTDIR}

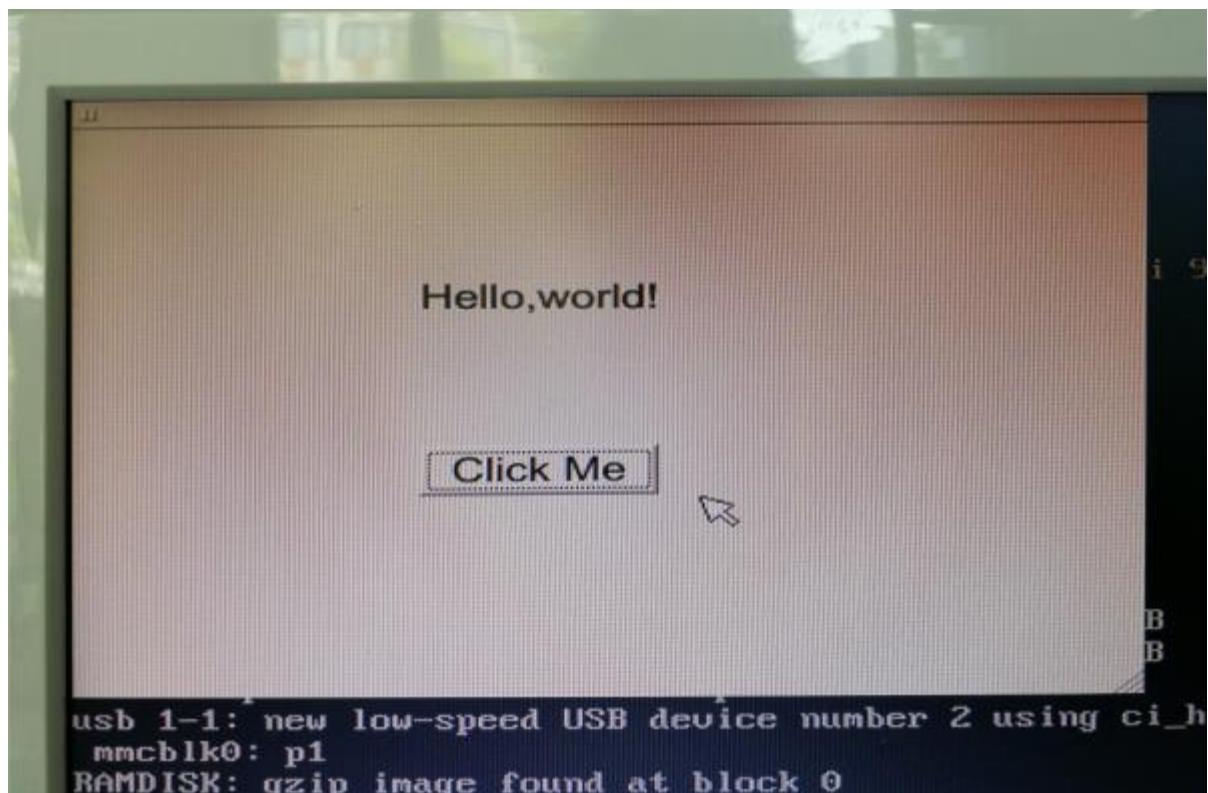
export QT_QPA_FONTDIR=${QTDIR}/lib/fonts
export QT_QPA_PLATFORM_PLUGIN_PATH=${QTDIR}/plugins/
export LD_LIBRARY_PATH=${QTDIR}/lib:$LD_LIBRARY_PATH

export QT_QPA_PLATFORM=linuxfb:fb=/dev/fb0

#http://www.360doc.com/content/14/1215/10/18578054_433033534.shtml
```

```
export QT_QPA_EVDEV_MOUSE_PARAMETERS=evdevmouse:/dev/event0  
/mnt/helloqt&
```

7.6 测试结果



第五季 HSL 算法基础入门 12 课时

HLS 是高层次综合的简称，“综合”即“Synthesis”，在 ug627《XST User Guide》中解释综合是将程序代码翻译为称为 NGC 的特殊网表文件中，这样才能够对其进行实现。

至于“层次”，或许可以这样理解。书中一般把 FPGA 设计分为以下几个级别（对于这个分级实际上没有一个特定的说法，可以参考第 13 章抽象级别的描述）：

- 系统级
- 算法级
- RTL 级
- 门级、开关级

一般认为 RTL 级及以下设计是可用的，“层次”即从什么角度去描述想要实现的功能。譬如， $a \text{ xor } b$ 采用门级描述就是 a, b 是一个异或门的输入；而采用高一点层次描述就是 $a+b$ 。显然，越低层次的描述越困难，后文例子中也能发现这一点。

HLS 就是从高层次描述，之后综合成可用的网表文件的技术。这里的“高”指采用 C、C++ 等编写程序，而不是传统的 HDL 语言。然而，实际上 Vivado 套件中是预先采用 Vivado HLS 这个软件将 C 程序转换成为 Verilog HDL 或者 VHDL 代码，之后进行下一步操作的，并不是直接综合 C 代码。

本章围绕 HLS 的应用提出 12 课程带来大家如何 HLS 硬件算法的高级编程。

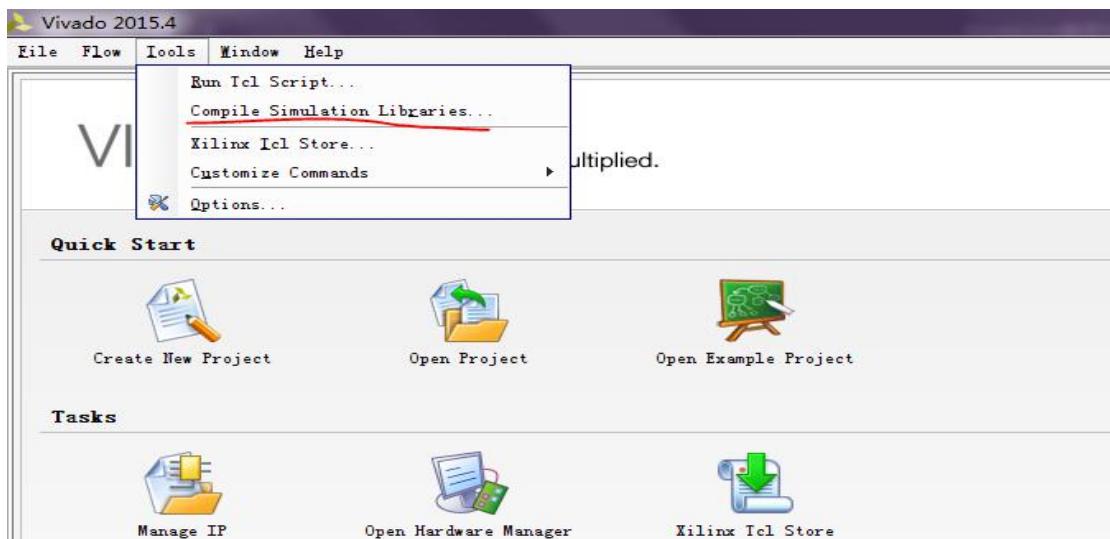
S05_CH01_搭建 Modelsim 和 Vivado 联合调试环境

1.1 概述

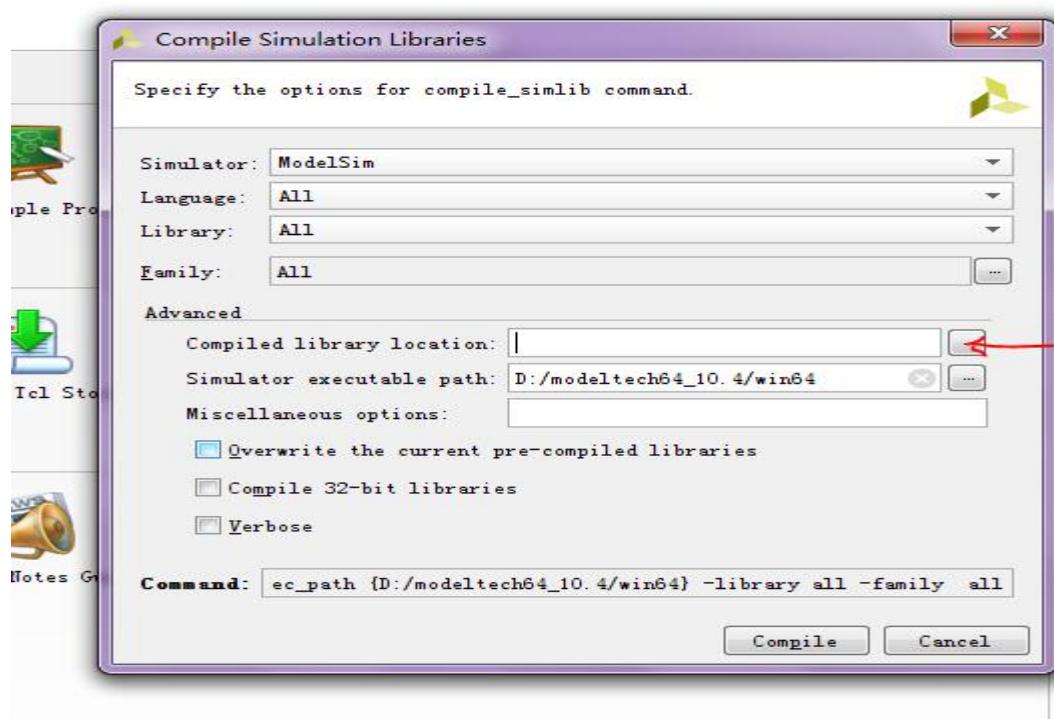
在进行 Vivado HLS 的学习之前，我们先把相应的准备工作做好，所谓工欲善其事，必先利其器，那么开发工具的安装必然是首要的，这里就不在赘述，这一节我们来讲讲如何搭建 Modelsim 与 Vivado 联合仿真环境。

1.2 使用 GUI 编译仿真库

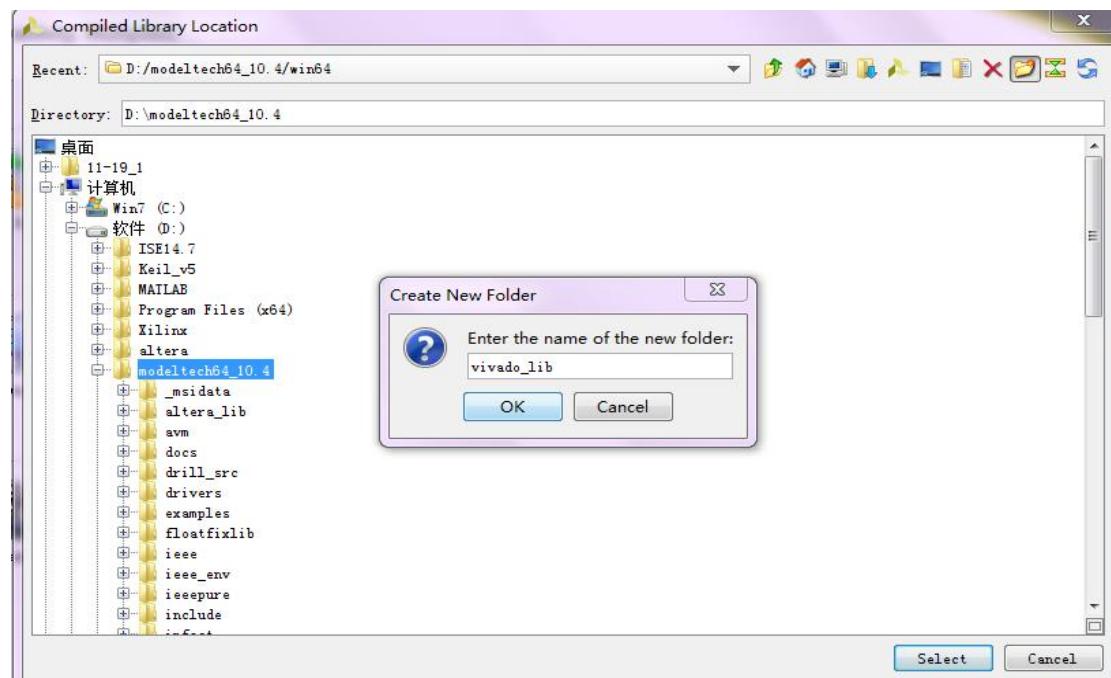
Step1:启动 Vivado，在菜单栏中选择 Tools → Compile Simulation Libraries，



Step2:启动编译界面如下，在红色箭头指向的位置选择默认的仿真库的编译路径，其他的默认



Step3: 新建一个文件夹存放编译库的文件，这里我命名为 vivado_lib，单击 OK 完成操作。



Step4: 点击 Compile 静静等待仿真库文件的生成。

```

Tcl Console
-----
* secureip      | verilog | secureip      | 0   | 0   *
*-----*
* axi_bfm       | verilog | secureip      | 0   | 0   *
*-----*
* unisim        | verilog | unisims_ver   | 0   | 0   *
*-----*
* unimacro      | verilog | unimacro_ver  | 0   | 0   *
*-----*
* unifast        | verilog | unifast_ver   | 0   | 0   *
*-----*
* simprim        | verilog | simprims_ver  | 0   | 0   *
*-----*
compile_simlib: Time (s): cpu = 00:00:03 ; elapsed = 00:02:00 . Memory (MB): peak = 599.078 : gain = 10.488

Type a Tcl command here
  
```

1.3 使用命令行编译仿真库

使用 TCL 脚本：compile_simlib 编译仿真库，下面的命令就可以实现（更多内容参考 ug835）

```
compile_simlib -directory <library_output_directory>-simulator <agr>
simulator_exec_path<sim_install_location> 例 如 : a ) 仿 真 库 编 译 到
D:/modeltech64_10.4/vivado_lib ; 仿 真 工 具 使 用 Modelsim ; ModelSim 安 装 在
D:/modeltech64_10.4/win64; 那 么 完 整 的 tcl 命 令 就 是 :
```

```
compile_simlib      -directory      D:/modeltech64_10.4/vivado_lib      -simulator
modelsim      -simulator_exec_path  D:/modeltech64_10.4/win64
```

```

***** Vivado v2015.4 (64-bit)
***** SW Build 1412921 on Wed Nov 18 09:43:45 MST 2015
***** IP Build 1412160 on Tue Nov 17 13:47:24 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Vivado% compile_simlib -directory D:/modeltech64_10.4/vivado_lib -simulator modelsim -simulator_exec_path D:/modeltech64_10.4/win64
Compiling libraries for 'modelsim' simulator in 'D:/modeltech64_10.4/vivado_lib'

Creating modelsim.ini file...
Model Technology ModelSim SE-64 vmap 10.4 Lib Mapping Utility 2014.12 Dec 3 201
4
vmap -
Copying D:/modeltech64_10.4/win64../modelsim.ini to modelsim.ini
--> Compiling 'verilog.secureip' library...
  > Source Library = 'D:\Xilinx\Vivado\2015.4\data\secureip'
  > Compiled Path = 'D:\modeltech64_10.4\vivado_lib\secureip'
  > Log File     = 'D:\modeltech64_10.4\vivado_lib\secureip/.cxl.verilog.sec
ureip.secureip.nt64.log'
** Warning: <vlib-34> Library already exists at "D:/modeltech64_10.4/vivado_lib/
secureip".
--> Compiling 'verilog.secureip:verilog.axi_bfm' library...
  > Source Library = 'D:\Xilinx\Vivado\2015.4\data\secureip\axi_bfm'
  > Compiled Path = 'D:\modeltech64_10.4\vivado_lib\secureip'
```

当等待一段时间出现如下编译界面时，表明编译无误：

```

ca. Vivado 2015.4 Tcl Shell - D:\Xilinx\Vivado\2015.4\bin\vivado.bat -mode tcl

* unifast          : vhdl      : unifast           : 0      :
@      *                                         :
*                                         :
* unisim           : verilog    : unisims_ver       : 0      :
@      *                                         :
*                                         :
* unimacro         : verilog    : unimacro_ver     : 0      :
@      *                                         :
*                                         :
* unifast          : verilog    : unifast_ver      : 0      :
@      *                                         :
*                                         :
* simprim          : verilog    : simprims_ver     : 0      :
@      *                                         :
*                                         :

compile_simlib: Time <s>: cpu = 00:00:01 ; elapsed = 00:04:02 . Memory <MB>: peak = 191.508 ; gain = 2.648
Vivado%

```

1.4 HLS 简单介绍

Vivado HLS 是 Xilinx 推出的高层次综合工具，采用 C/C++语言进行 FPGA 设计，HLS 提供了一些 example 样例方便大家熟悉基本的开发流程，另外关于 HLS 的使用介绍，Xilinx 官方有两个非常重要的开发文档，ug871-vivado-high-level-synthesis-tutorial.pdf 和 ug902-vivado-high-level-synthesis.pdf，里面详细介绍了包括怎样建立 HLS 工程，怎么编写 testbench，怎么进行优化等问题。关于优化，上面提到的两篇 PDF 文档里介绍的比较详细，在 HLS 软件界面，点击程序所在的文件，在右侧边栏有个 Directive，里面列出了程序中所有用到的变量、函数和循环结构，点击右键可以给其配置。对循环结构，一般选择 unroll（即展开循环），可以自己设定展开因子 factor。为提高程序的并行化处理，可以给函数选择 PIPELINE。对应数组，可以设置为 ARRAY_PARTITION，数组维数可以自己设定。HLS 软件其实很智能的，简单的结构，一般软件自己会优化好。每一个优化方案都保存在一个 Solution 里，HLS 可以创建多个 Solution，用于比较不用的优化效果。

从工业检测系统到自动驾驶系统，计算机视觉拥有广泛的应用领域。而 OpenCV 包含 2500 多个优化的视频函数的函数库，并且专门针对台式机处理器和图形处理器（Graphic Processing Unit, GPU）进行优化。利用逻辑硬件来加速 OpenCV 的性能在 HLS 上得以实现。

Xilinx 提供的 Vivado HLS 高层次综合工具能够通过 C/C++编写的代码直接创建 RTL 硬件，显著提高设计效率；同时，Xilinx ZynqSOC 系列器件嵌入双核 ARM Cortex-A9 处理器将软件可编程能

力与 FPGA 的硬件可编程能力实现完美结合，以低功耗和低成本等系统优势实现单芯片无以伦比的系统性能、灵活性、可扩展性，加速图形处理产品设计上市时间。

1.4.1 OpenCV 和 HLS 视频库

如下图所示，OpenCV 在视频处理系统中可以有四种不同的应用方式。第一种方式中，算法的设计和实现完全依赖于 OpenCV 的函数调用，利用文件的访问功能进行图片的输入、输出和处理；第二种方式中，算法可以在嵌入式系统（例如 Zynq Base TRD）中实现，利用特定平台的函数调用访问输入输出图像，但是，视频处理的实现依赖于嵌入式系统中处理器（例如 Cortex™-A9）对 OpenCV 功能函数的调用；第三种方式中，处理算法的 OpenCV 功能函数被 Xilinx Vivado HLS 视频库函数替换，而 OpenCV 函数则用于访问输入和输出图像，提供视频处理算法实现的设计原型。Vivado HLS 提供的视频库函数可以被综合，在对这些函数综合后，可以将处理程序模块整合到诸如 Zynq 的可编程逻辑中。这样，这些程序逻辑块就可以处理由处理器生产的视频流、从文件中读取的数据、外部输入的实时视频流。

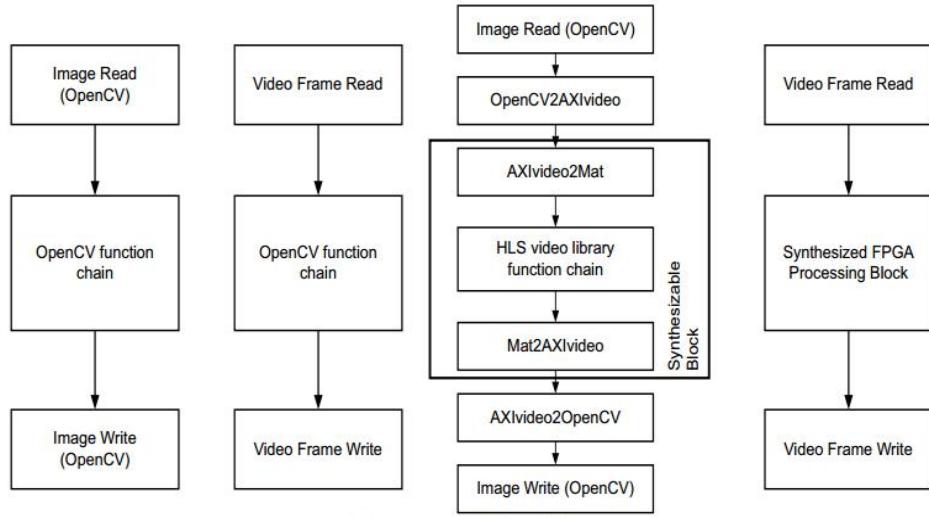


Figure 1: Design Flow

Vivado HLS 包含大量的视频库函数，方便于构建各种各样的视频处理程序。通过可综合的 C++ 代码，实现这些视频库函数。在视频处理功能和数据结构方面，这些综合后的代码与 OpenCV 基本对应。许多视频概念与抽象和 OpenCV 非常相似，很多图像处理模块函数和 OpenCV 库函数一致。

例如，OpenCV 中用于代表图片的很重要的一个类便是 cv::Mat 类，cv::Mat 对象定义如下：

```
cv::Mat image(1080, 1920, CV_8UC3);
```

该行代码声明了一个 1080×1920 像素，每一个像素都是由 3 个 8 位无符号数表示的变量 `image`。对应的 HLS 视频库模板类 `hls::Mat<>` 声明如下：

```
hls::Mat<2047, 2047, HLS_8UC3> image(1080, 1920);
```

这两行代码的参数形式、图像尺寸最大值、语法规则不同，生成的对象是相似的。如果图像规定的最大尺寸和图像的实际尺寸相同的话，也可以用下面的代码替代：

```
hls::Mat<1080, 1920, HLS_8UC3> image();
```

1.4.2 AXI4 流和视频接口

通过 AXI4 流协议，Xilinx 提供的视频处理模块实现像素数据通信。尽管底层 AXI4 流媒体协议不需要限制图片尺寸，但是，若图片尺寸相同，则将会大大简化大部分的复杂视频处理计算。对于遵循 AXI4 流协议的输入接口，可以保证每一帧都包含 `ROWS * COLS` 的像素。在保证前面视频帧保持完整性和矩形性的情况下，后续模块实现对视频帧有效地处理。

如图所示，Vivado HLS 包含 2 个可综合的视频接口转换函数。

Table 2: Vivado HLS Synthesizable Video Functions

Video Library Function	Description
<code>hls::AXIvideo2Mat</code>	Converts data from an AXI4 video stream representation to <code>hls::Mat</code> format.
<code>hls::Mat2AXIvideo</code>	Converts data stored as <code>hls::Mat</code> format to an AXI4 video stream.

视频库还提供了其他不可综合的视频接口函数，这些函数用于在基于 OpenCV 测试平台与综合后的函数结合。下图给出了不可综合的接口函数。

Table 3: Vivado HLS Non-Synthesizable Video Functions

Video Library Functions	
<code>hls::cvMat2AXIvideo</code>	<code>hls::AXIvideo2cvMat</code>
<code>hls::IplImage2AXIvideo</code>	<code>hls::AXIvideo2IplImage</code>
<code>hls::CvMat2AXIvideo</code>	<code>hls::AXIvideo2CvMat</code>

VivadoHLS 视频处理函数库使用 `hls::Mat<>` 数据类型，这种类型用于模型化视频像素流处理，实质等同于 `hls::steam<>` 流的类型，而不是 OpenCV 中在外部 memory 中存储的 matrix 矩阵类型。因此，在用 Vivado HLS 实现 OpenCV 的设计中，需要将输入和输出 HLS 可综合的视频设计接口，修

改为 Video stream 接口，也就是采用 HLS 提供的 video 接口可综合函数，实现 AXI4 video stream 到 VivadoHLS 中 hls::Mat<>类型的转换。

1.4.3 OpenCV 到 RTL 代码转换的流程

OpenCV 图像处理是基于存储器帧缓存而构建的，它总是假设视频帧数据存放在外部 DDR 存储器中。由于处理器的小容量高速缓存性能的限制，因此，OpenCV 访问局部图像性能较差。并且，从性能的角度来说，基于 OpenCV 设计的架构比较复杂，功耗更高。在对分辨率或帧速率要求低，或者在更大的图像中对需要的特征或区域进行处理时，OpenCV 似乎足以满足很多应用的要求；但是，对于高分辨率和高帧率实时处理的应用中，OpenCV 很难满足高性能和低功耗的需求。

基于视频流的架构能提供高性能和低功耗，链条化的图像处理函数减少了外部存储器访问。针对视频优化的行缓存和窗口缓存比处理器高速缓存更简单高效，更易于使用 Xilinx 提供的 Vivado HLS 在 FPGA 器件中采用数据流优化来实现。

Vivado HLS 对 OpenCV 的支持，不是指可以将 OpenCV 的函数库直接综合成 RTL 代码，而是需要将代码转换为可综合的代码。这些可综合的视频库称为 Vivado HLS 视频库，它们由 Vivado HLS 工具提供。

由于 OpenCV 函数一般都包含动态的内存分配、浮点以及假设图像在外部存储器中存放或者修改，所以不能直接通过 Vivado HLS 对 OpenCV 函数进行综合。

Vivado HLS 视频库用于替换很多基本的 OpenCV 函数，它与 OpenCV 具有相似的接口和算法，它主要针对在 FPGA 架构中实现的图像处理函数，其中包含了专门面向 FPGA 的优化，比如定点运算而非浮点运算（不必精确到比特位），片上的行缓存（line buffer）和窗口缓存（window buffer）。

1.5 本章小结

本章是 HLS 部分的第一章，为大家简单的介绍了一些关于 HLS 的基础知识，为下面正式开始学习 HLS 打下基础。HLS 的优势在于，它可以把 C 语言转化为我们的硬件描述语言，这意味着，因此，它可以让更多不懂硬件描述语言的开发者投入到 FPGA 的开发中来。一些优秀的 C 算法，我们也可以使用 HLS 将其转换为硬件描述语言，而且其质量也要比人工设计的更好，节省了人力和物力。

S05_CH02_shift_led 实验

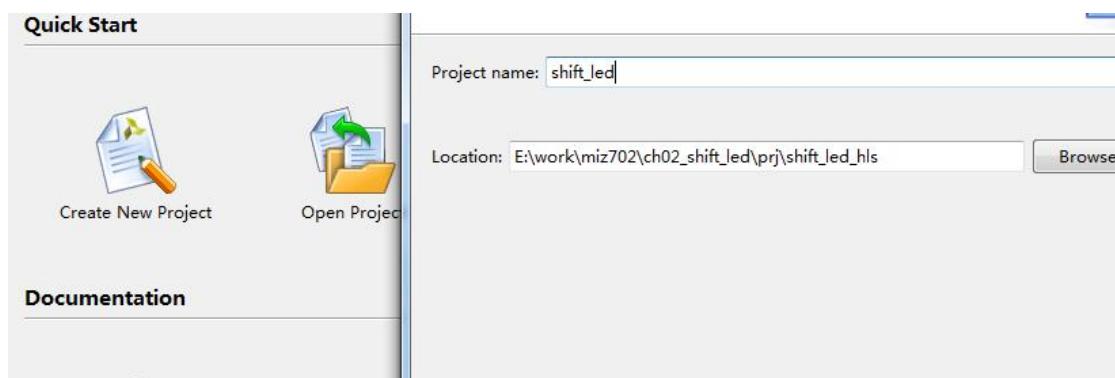
2.1 概述

如同软件开发都是从“Hello World!”进入编程的大门一样，这一章我们就通过 HLS 封装一个移位流水灯的程序熟悉 HLS 的开发流程，包括工程的创建，仿真，综合，封装，以及在硬件平台上的实现。

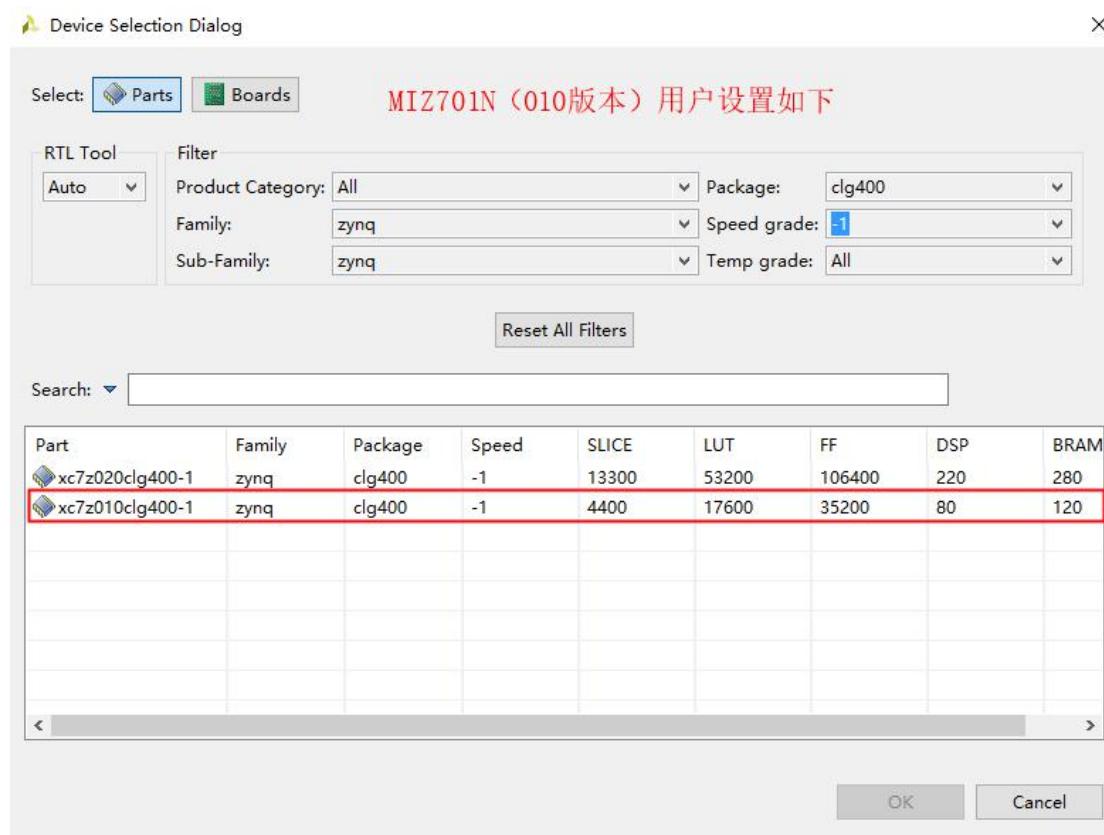
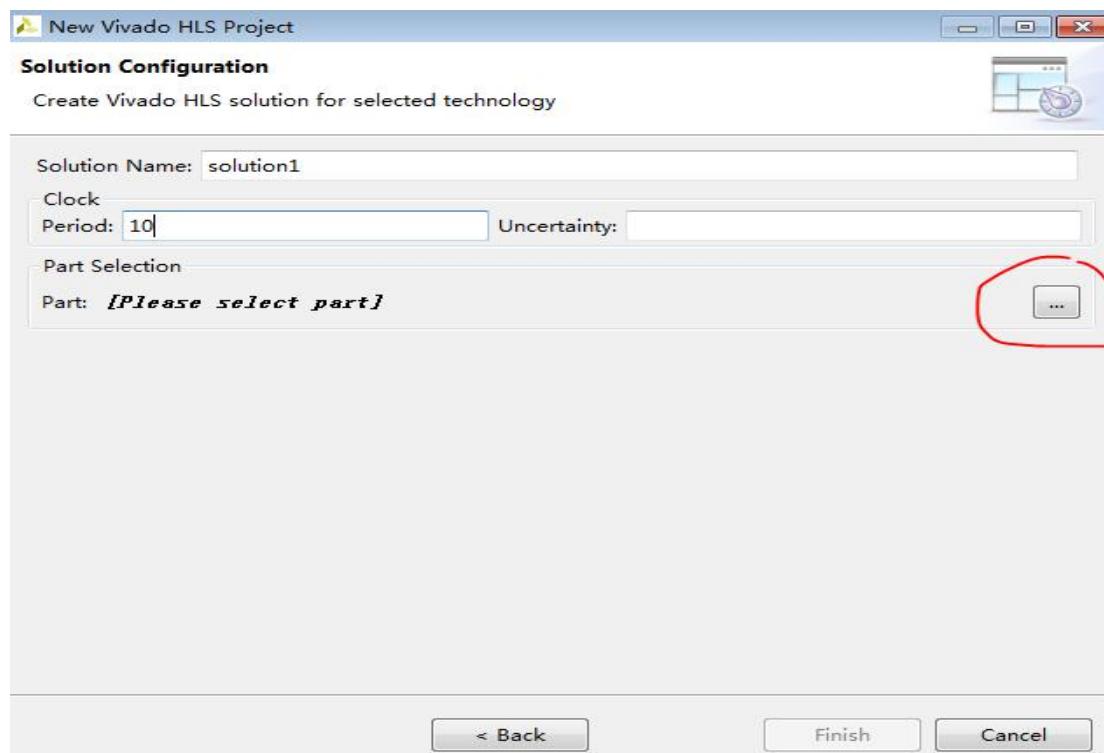
2.2 工程创建、仿真及优化

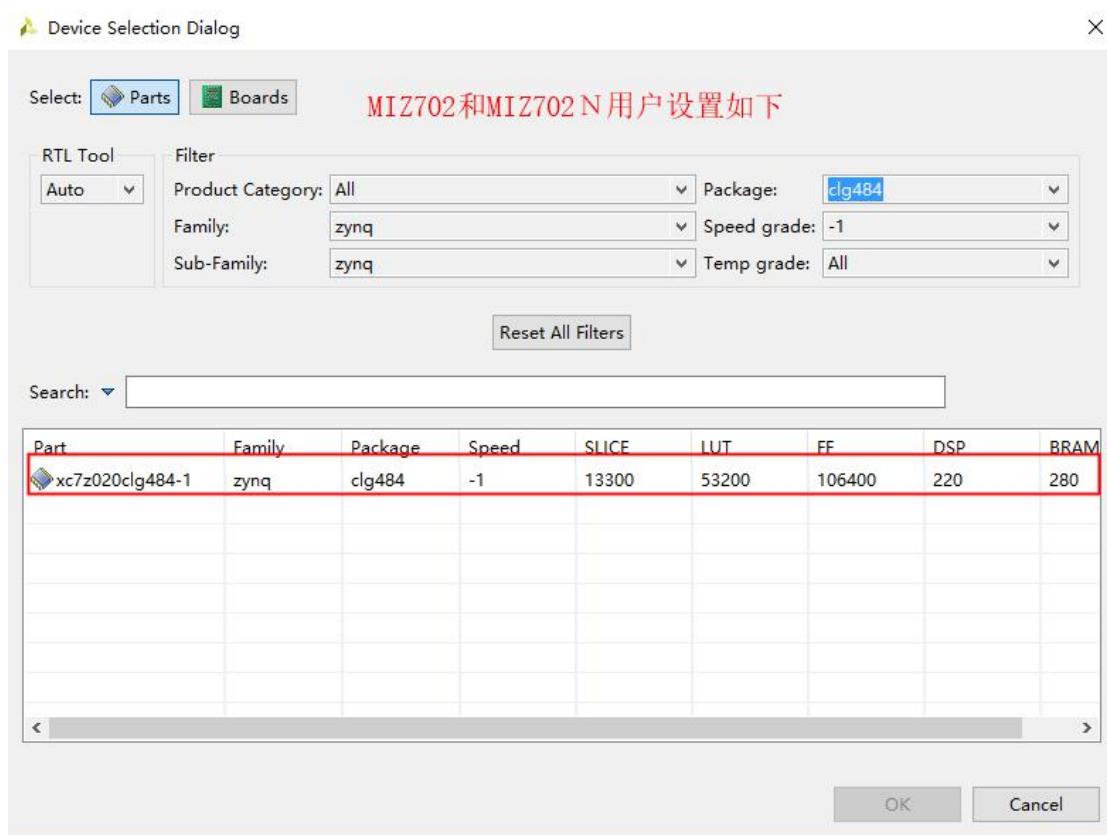
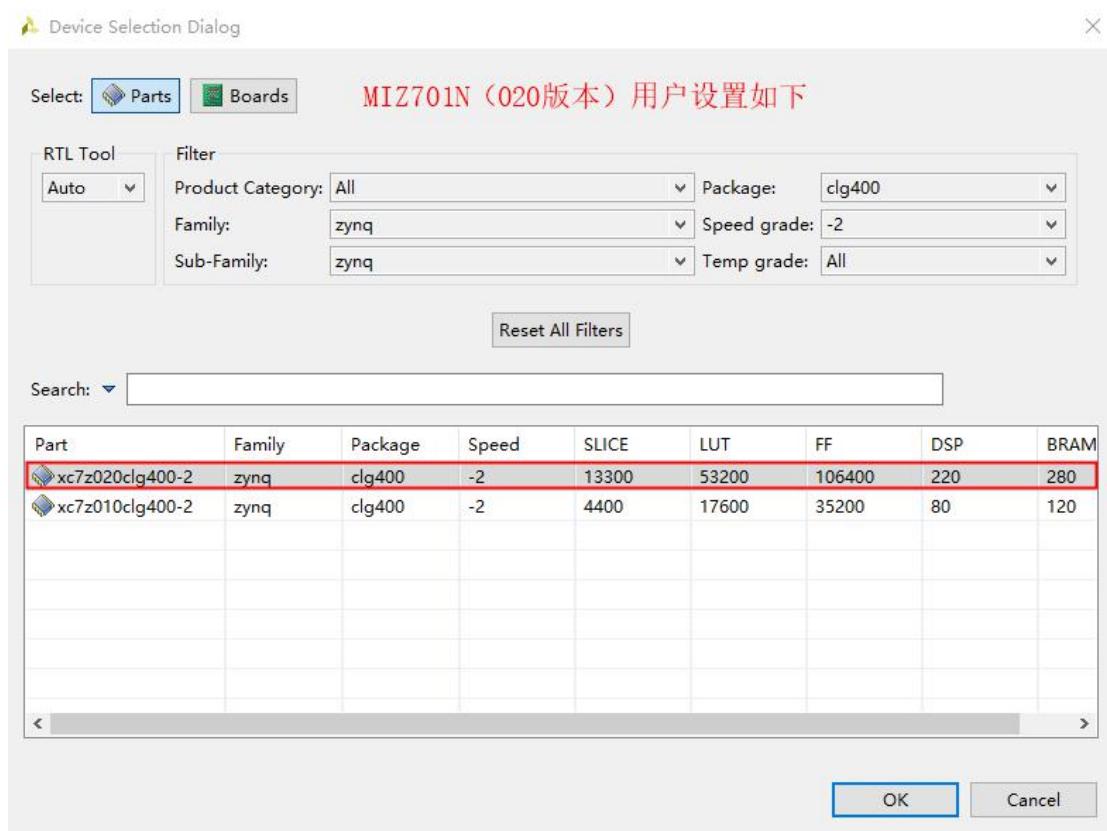
2.2.1 工程创建

Step1: 打开 Vivado HLS 开发工具，单击 Create New Project 创建一个新工程，设置好工程路径和工程名，一直点击 Next 按照默认设置，

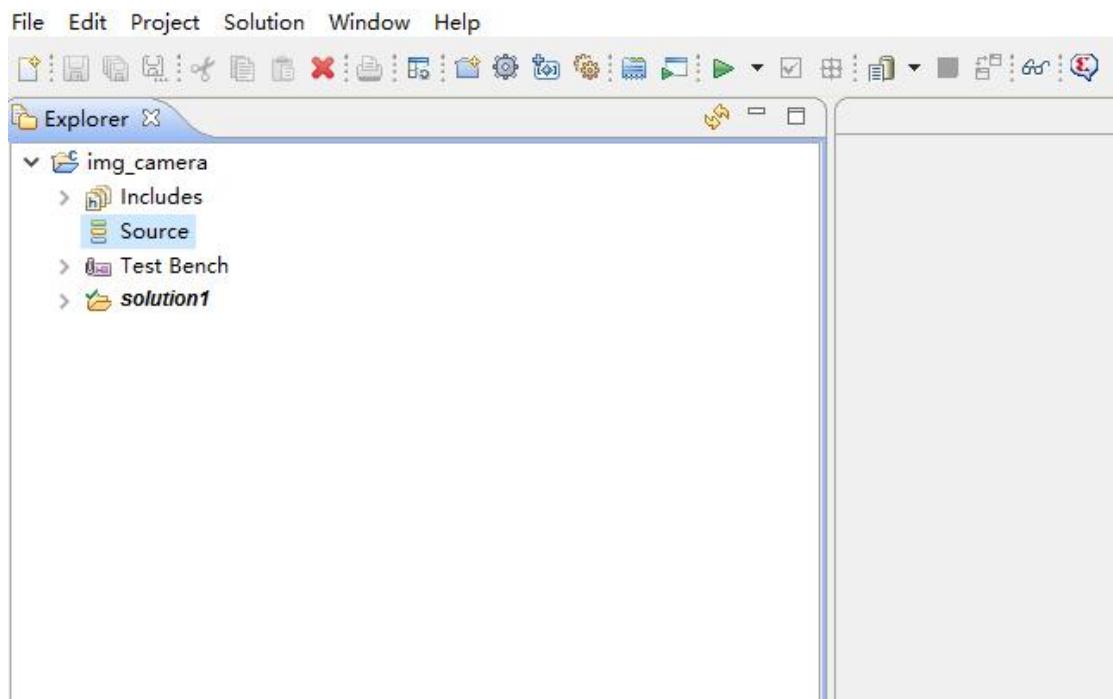


Step2: 出现如下图所示界面，时钟周期 Clock Period 按照默认 10ns，Uncertainty 和 Solution Name 均按照默认设置，点击红色圆圈部分选择芯片类型，然后点击 OK。

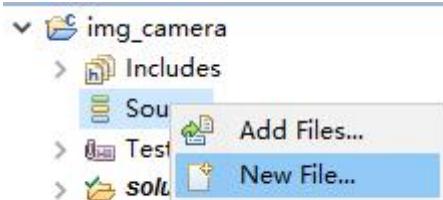




Step3: 点击 Finish, 出现如下界面:



Step4: 右单击 Source, 选择 New file, 添加一个设计源文件。



Step5: 输入文件名和选择保存的路径, 然后单击保存。



Step6: 双击刚才添加的文件, 添加如下程序。

```

#include "shift_led.h"

void shift_led(led_t *led_o, led_t led_i)
{
#pragma HLS INTERFACE ap_ovld port=led_o
#pragma HLS INTERFACE ap_ovld port=led_o
#pragma HLS INTERFACE ap_vld port=led_i

    led_t tmp_led;
    cnt32_t i;//for循环的延时变量
    tmp_led = led_i;
    for(i = 0; i < MAX_CNT; i++)
    {
        if(i==SHIFT_FLAG)
        {
            tmp_led = ((tmp_led>>7)&0x01) + ((tmp_led<<1)&0xFE); //左移
            *led_o = tmp_led;
        }
    }
}

```

Step7：按照同样的方法添加一个 shift_led.h 文件，添加下面的程序。

```

#ifndef _SHIFT_LED_H_
#define _SHIFT_LED_H_

//加入设置 int 自定义位宽的头文件
#include "ap_int.h"

//设置灯半秒动一次，开发板时钟频率是 100M
//#define MAX_CNT 1000/2 //仅用于仿真，不然时间较长
#define MAX_CNT 100000000/2

#define SHIFT_FLAG MAX_CNT-2

//typedef int led_t;
//typedef int cnt32_t;//计数器
typedef ap_fixed<4, 4> led_t; //第一个 4 代表总位宽，第二个 4 代表整数部分的位宽

```

是 8，则小数部分位宽=4-4=0

```
typedef ap_fixed<32, 32> cnt32_t;

void shift_led(led_t *led_o, led_t led_i);

#endif
```

Step8：单击 Test Bench，添加一个名为 Test_shift_led.cpp 的测试文件，并添加如下程序。

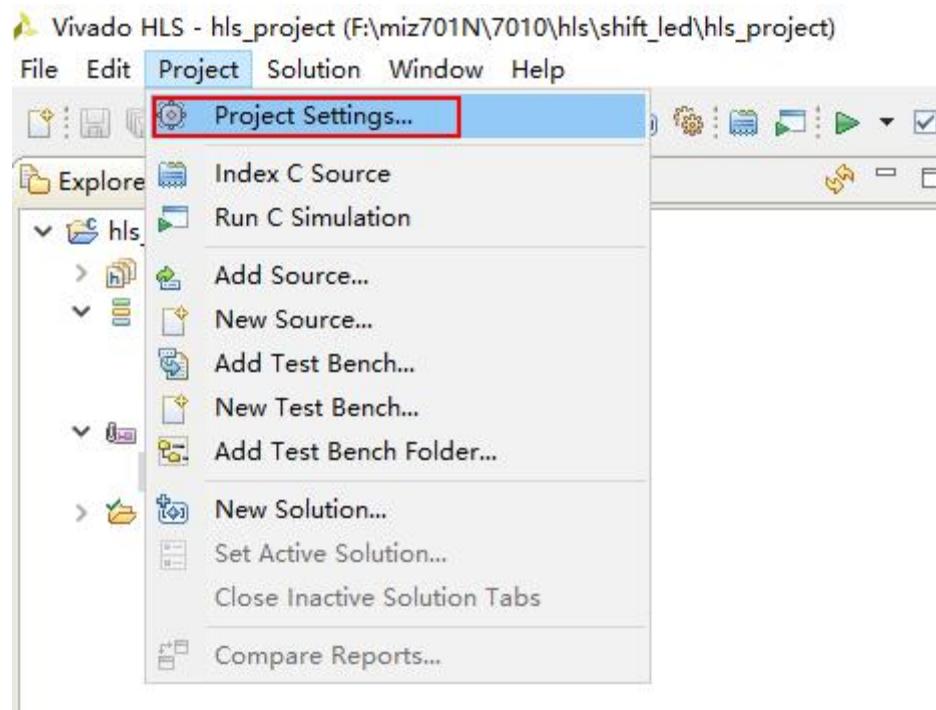
```
#include "shift_led.h"
#include <stdio.h>

using namespace std;

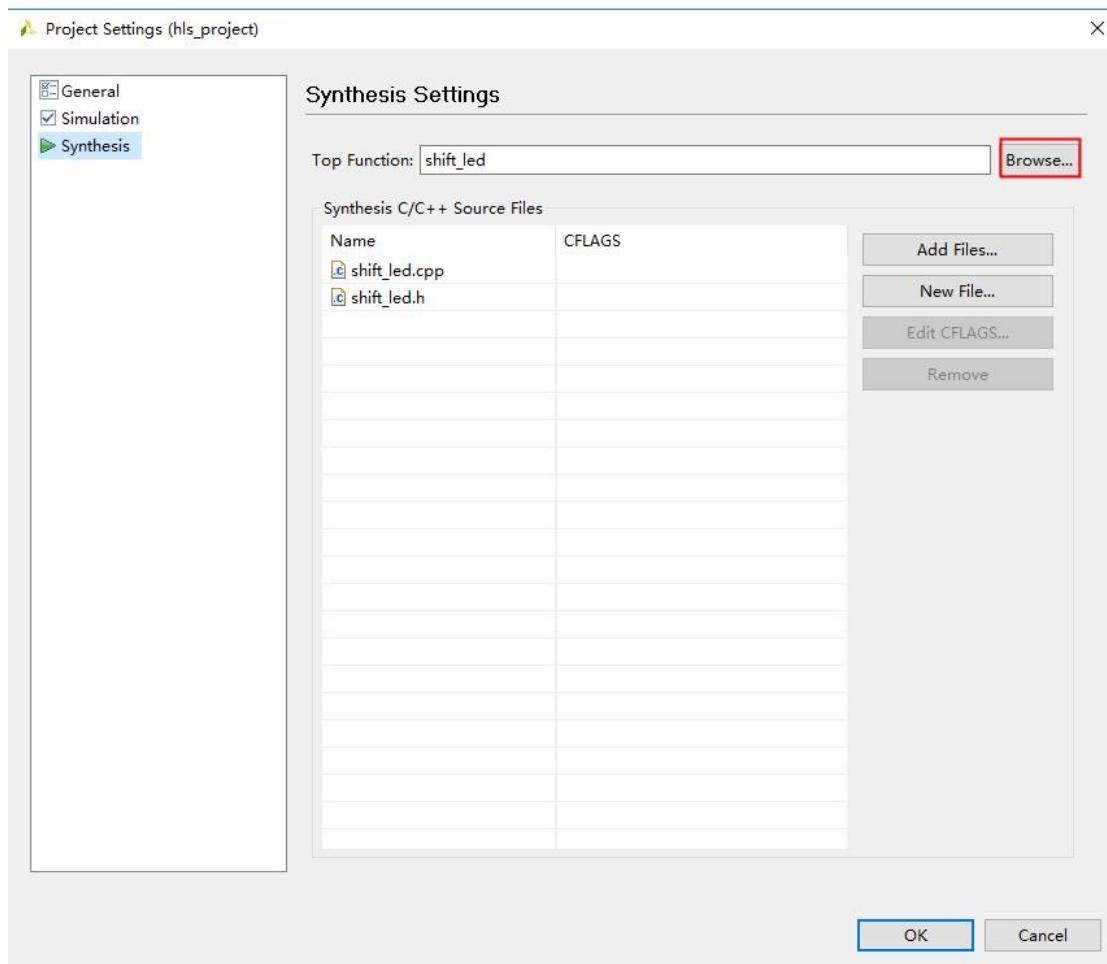
int main()
{
    led_t led_o;
    led_t led_i = 0xFE; //1111_1110
    const int SHIFT_TIME = 8;
    int i;
    for(i = 0;i < SHIFT_TIME;i++)
    {
        shift_led(&led_o,led_i);
        led_i = led_o;
        char string[25];
        itoa((unsigned int)led_o&0xFF,string,2); //&0xFF是为了取led_o的8位，转化为二进制数出
        if(i == 6)
            fprintf(stdout,"shift_out= 0%s\n",string); //数据对齐，高位补零
        else
            fprintf(stdout,"shift_out= %s\n",string);
    }
}
```

2.2.2 代码综合

Step1:点击 Project → Project Settings 出现下图，在 Synthesis 界面下选择综合的顶层函数名。



Step2：单击 Browse 指定工程的顶层文件，最后单击 OK 完成修改。



Step3: 因为当前工程中只存在一个 Solution (解决方案)，我们选择 Solution -> Run C Synthesis -> Active Solutions 或者直接点击 进行综合，等待一段时间，在未经优化的情况下综合报告如图所示，我们可以看到 FF 和 LUT 分别使用了 93 和 186。

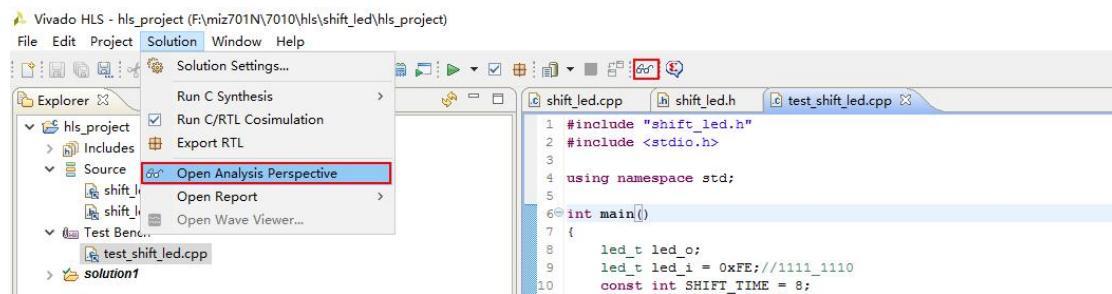
Performance Estimates

- Timing (ns)**
 - Summary**

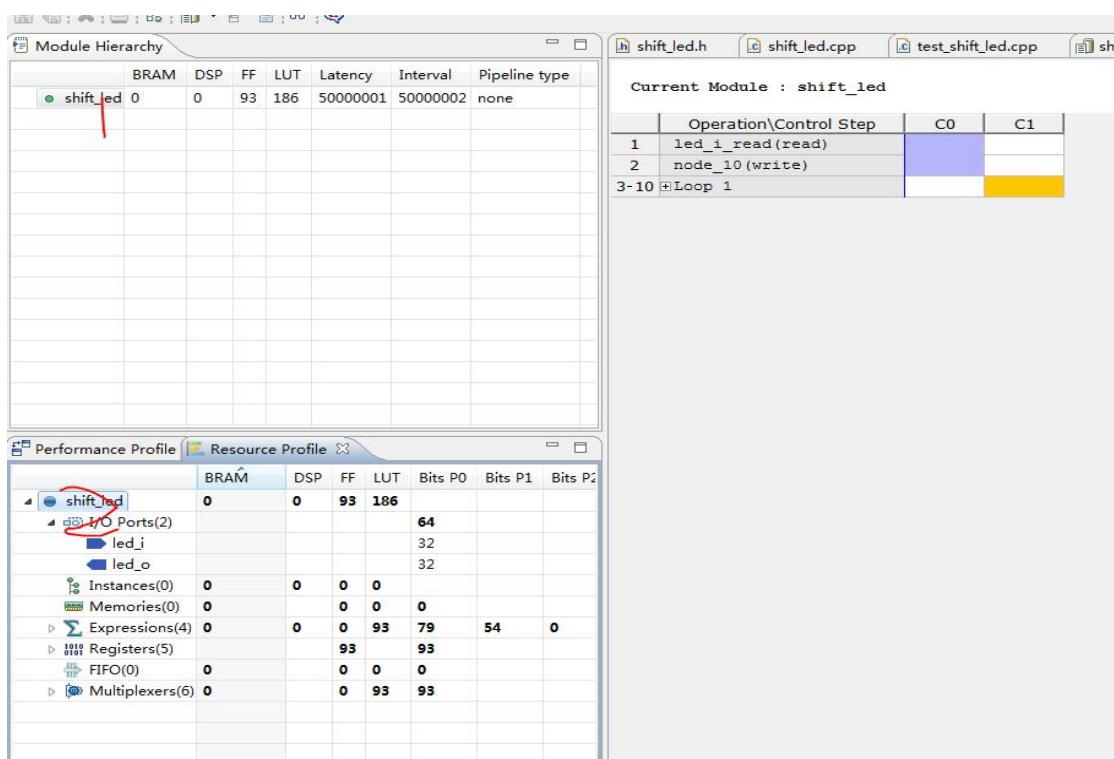
Clock	Target	Estimated	Uncertainty
default	10.00	2.43	1.25
 - Latency (clock cycles)**
- Utilization Estimates**
 - Summary**

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	93
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	93
Register	-	-	93	-
Total	0	0	93	186
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0
 - Detail**
 - Instance**

Step4: 我们单击方框选中的地方点击选择打开分析报告，

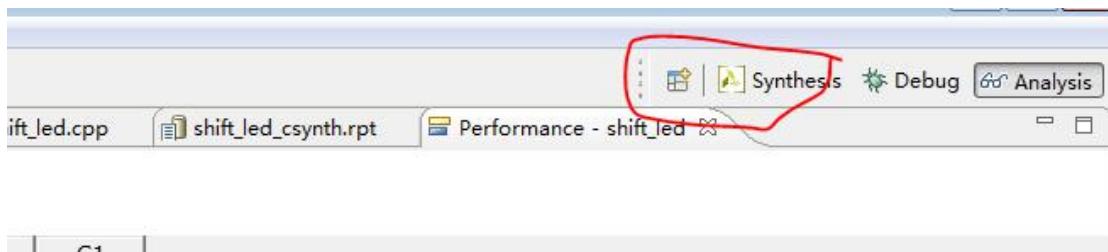


Step5: 在下图所示 1 的地方点击出现 2 所示的 shift_led，点击展开后可以看到 LED 输入和输出位宽均为 32 位，我们板载是 8 个 LED (Miz701N 是 4 个)，那么该怎么去进行优化得到我们想要的结果呢？



2.2.3 代码优化

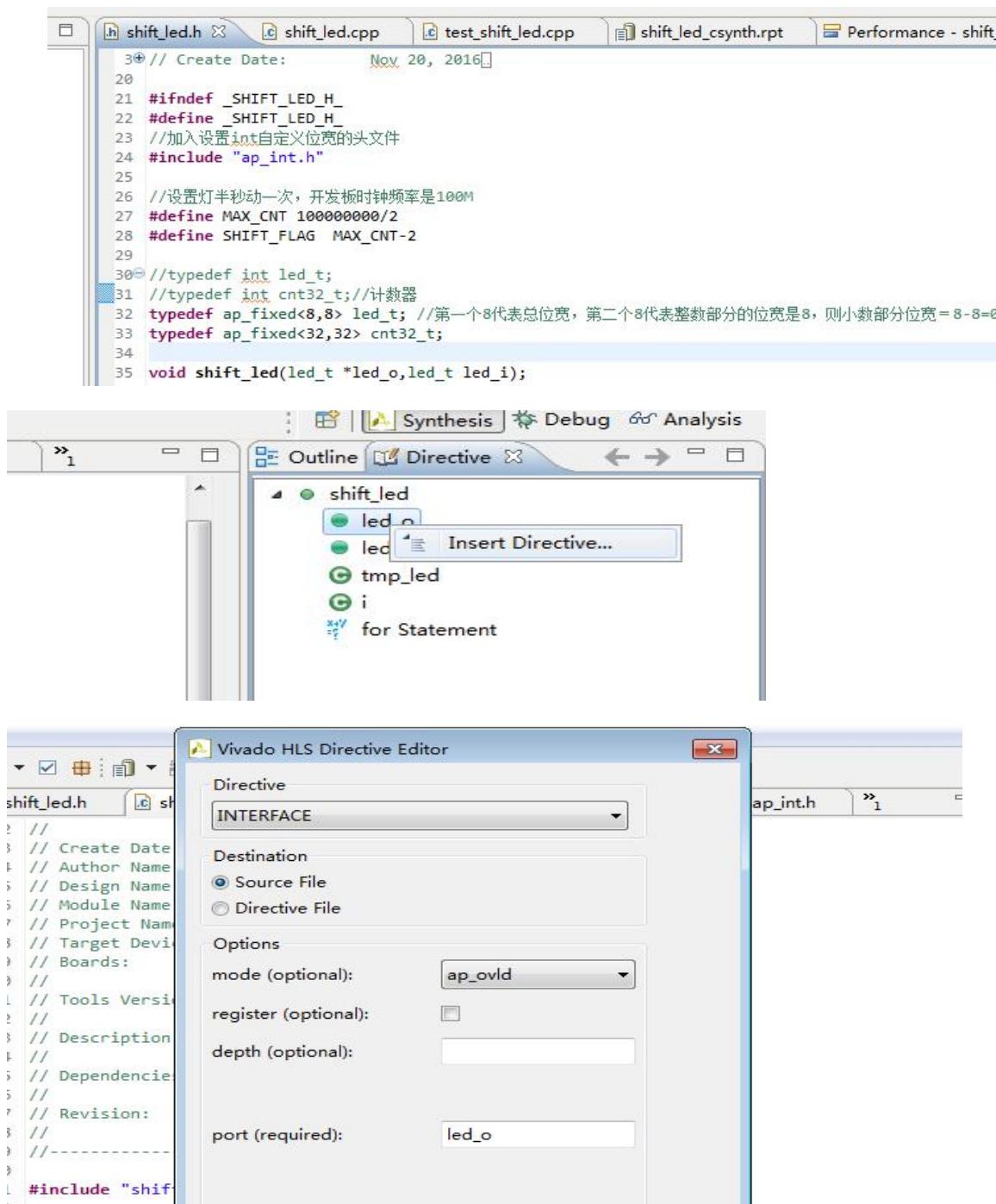
点击 Synthesis 切换到工作空间主界面。



在 shift_led.h 文件中我们包含一个设置 int 自定义位宽的头文件"ap_int.h"，我们使用 ap_fixed() 函数自定义 int 型的 bit 数，其函数原型为：

`class ap_fixed: public ap_fixed_base<_AP_W, _AP_I, true, _AP_Q, _AP_O, _AP_N>,` 那么这个函数怎么使用呢？它的定义如下：

ap_fixed<M, N>, 第一个 M 代表数据总位宽, N 代表数据整数部分的位宽, 那么小数部分的位宽即 M - N; 在下面的代码的第 30 和 31 行我们将它替换为 32 和 33 行所示的代码, 我们即可实现对端口数据位宽的约束, 另外我们再进行端口约束, 约束方法如下, 在需综合的 shift_led.cpp 文件中的 Directive 目录下的 led_o 上右键选择 Insert Directive



因为 led_o 是接口，所以 Directive 我们选择为 INTERFACE, Destination 选择为 Source File, 那么有的会问了，这两个有什么区别吗？区别就是 Source File 是针对所有的 Solution 采用同一个优化手段，而 Directive File 是对当前的 Solution 有效，mode(optional) 我们选为 ap_ovld，即输出使能。对 led_i 进行同样的约束，最终效果如下，我们再次综合，对比下进行约束以后的资源利用情况。

```

15 // dependencies:
16 //
17 // Revision: Nov 20, 2016: 1.00 Initial version
18 //
19 //-----
20
21 #include "shift_led.h"
22
23 void shift_led(led_t *led_o, led_t led_i){
24 #pragma HLS INTERFACE ap_ovld port=led_o
25 #pragma HLS INTERFACE ap_vld port=led_i
26

```

Performance Estimates																																																																																																																	
Timing (ns) <table border="1"> <thead> <tr> <th colspan="4">Summary</th> </tr> <tr> <th>Clock</th> <th>Target</th> <th>Estimated</th> <th>Uncertainty</th> </tr> </thead> <tbody> <tr> <td>default</td> <td>10.00</td> <td>2.43</td> <td>1.25</td> </tr> </tbody> </table> Latency (clock cycles) <table border="1"> <thead> <tr> <th colspan="4">Summary</th> </tr> <tr> <th>Clock</th> <th>Target</th> <th>Estimated</th> <th>Uncertainty</th> </tr> </thead> <tbody> <tr> <td>default</td> <td>10.00</td> <td>2.43</td> <td>1.25</td> </tr> </tbody> </table>				Summary				Clock	Target	Estimated	Uncertainty	default	10.00	2.43	1.25	Summary				Clock	Target	Estimated	Uncertainty	default	10.00	2.43	1.25																																																																																						
Summary																																																																																																																	
Clock	Target	Estimated	Uncertainty																																																																																																														
default	10.00	2.43	1.25																																																																																																														
Summary																																																																																																																	
Clock	Target	Estimated	Uncertainty																																																																																																														
default	10.00	2.43	1.25																																																																																																														
Utilization Estimates <table border="1"> <thead> <tr> <th colspan="5">Summary</th> </tr> <tr> <th>Name</th> <th>BRAM_18K</th> <th>DSP48E</th> <th>FF</th> <th>LUT</th> </tr> </thead> <tbody> <tr> <td>Expression</td> <td>-</td> <td>-</td> <td>0</td> <td>93</td> </tr> <tr> <td>FIFO</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <td>Instance</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <td>Memory</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <td>Multiplexer</td> <td>-</td> <td>-</td> <td>-</td> <td>93</td> </tr> <tr> <td>Register</td> <td>-</td> <td>-</td> <td>93</td> <td>-</td> </tr> <tr> <td>Total</td> <td>0</td> <td>0</td> <td>93</td> <td>186</td> </tr> <tr> <td>Available</td> <td>280</td> <td>220</td> <td>106400</td> <td>53200</td> </tr> <tr> <td>Utilization (%)</td> <td>0</td> <td>0</td> <td>~0</td> <td>~0</td> </tr> </tbody> </table> Detail <table border="1"> <thead> <tr> <th colspan="5">Summary</th> </tr> <tr> <th>Name</th> <th>BRAM_18K</th> <th>DSP48E</th> <th>FF</th> <th>LUT</th> </tr> </thead> <tbody> <tr> <td>Expression</td> <td>-</td> <td>-</td> <td>0</td> <td>93</td> </tr> <tr> <td>FIFO</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <td>Instance</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <td>Memory</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <td>Multiplexer</td> <td>-</td> <td>-</td> <td>-</td> <td>45</td> </tr> <tr> <td>Register</td> <td>-</td> <td>-</td> <td>45</td> <td>-</td> </tr> <tr> <td>Total</td> <td>0</td> <td>0</td> <td>45</td> <td>138</td> </tr> <tr> <td>Available</td> <td>280</td> <td>220</td> <td>106400</td> <td>53200</td> </tr> <tr> <td>Utilization (%)</td> <td>0</td> <td>0</td> <td>~0</td> <td>~0</td> </tr> </tbody> </table>				Summary					Name	BRAM_18K	DSP48E	FF	LUT	Expression	-	-	0	93	FIFO	-	-	-	-	Instance	-	-	-	-	Memory	-	-	-	-	Multiplexer	-	-	-	93	Register	-	-	93	-	Total	0	0	93	186	Available	280	220	106400	53200	Utilization (%)	0	0	~0	~0	Summary					Name	BRAM_18K	DSP48E	FF	LUT	Expression	-	-	0	93	FIFO	-	-	-	-	Instance	-	-	-	-	Memory	-	-	-	-	Multiplexer	-	-	-	45	Register	-	-	45	-	Total	0	0	45	138	Available	280	220	106400	53200	Utilization (%)	0	0	~0	~0
Summary																																																																																																																	
Name	BRAM_18K	DSP48E	FF	LUT																																																																																																													
Expression	-	-	0	93																																																																																																													
FIFO	-	-	-	-																																																																																																													
Instance	-	-	-	-																																																																																																													
Memory	-	-	-	-																																																																																																													
Multiplexer	-	-	-	93																																																																																																													
Register	-	-	93	-																																																																																																													
Total	0	0	93	186																																																																																																													
Available	280	220	106400	53200																																																																																																													
Utilization (%)	0	0	~0	~0																																																																																																													
Summary																																																																																																																	
Name	BRAM_18K	DSP48E	FF	LUT																																																																																																													
Expression	-	-	0	93																																																																																																													
FIFO	-	-	-	-																																																																																																													
Instance	-	-	-	-																																																																																																													
Memory	-	-	-	-																																																																																																													
Multiplexer	-	-	-	45																																																																																																													
Register	-	-	45	-																																																																																																													
Total	0	0	45	138																																																																																																													
Available	280	220	106400	53200																																																																																																													
Utilization (%)	0	0	~0	~0																																																																																																													

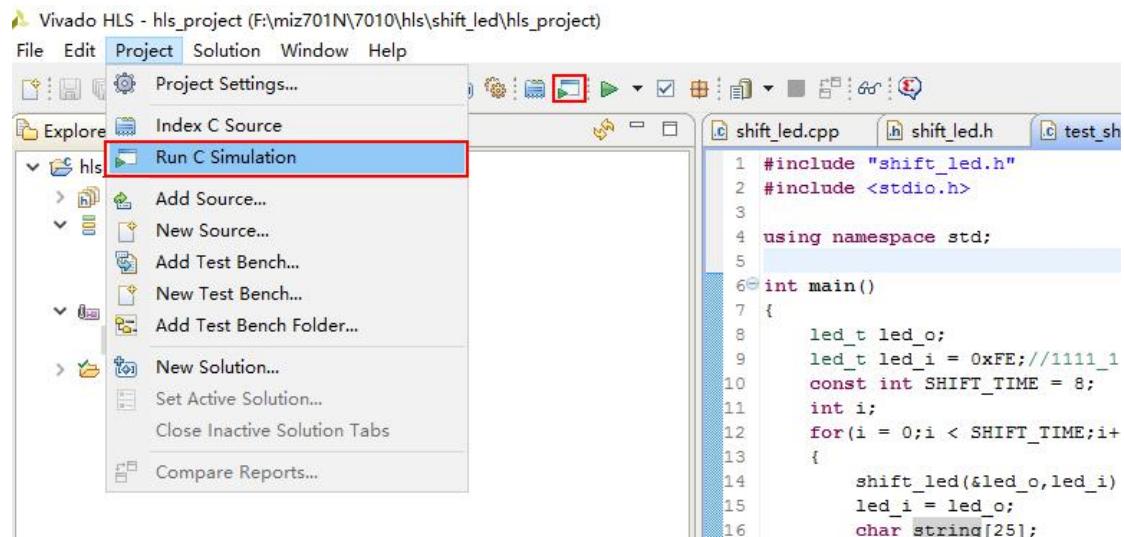
图左：优化前

图右：优化后

对比发现，经过优化以后，资源利用率显著降低，这为以后我们对资源进行优化提供了一个参考思路。

2.2.4 仿真实现

Step1:单击 Project 下的 Run C Simulation 或直接单击  开始 C 仿真。



Step2：等待一段时间，仿真结果如下，我们可以看到数据循环左移了一位，达到了我们想要的实验效果。

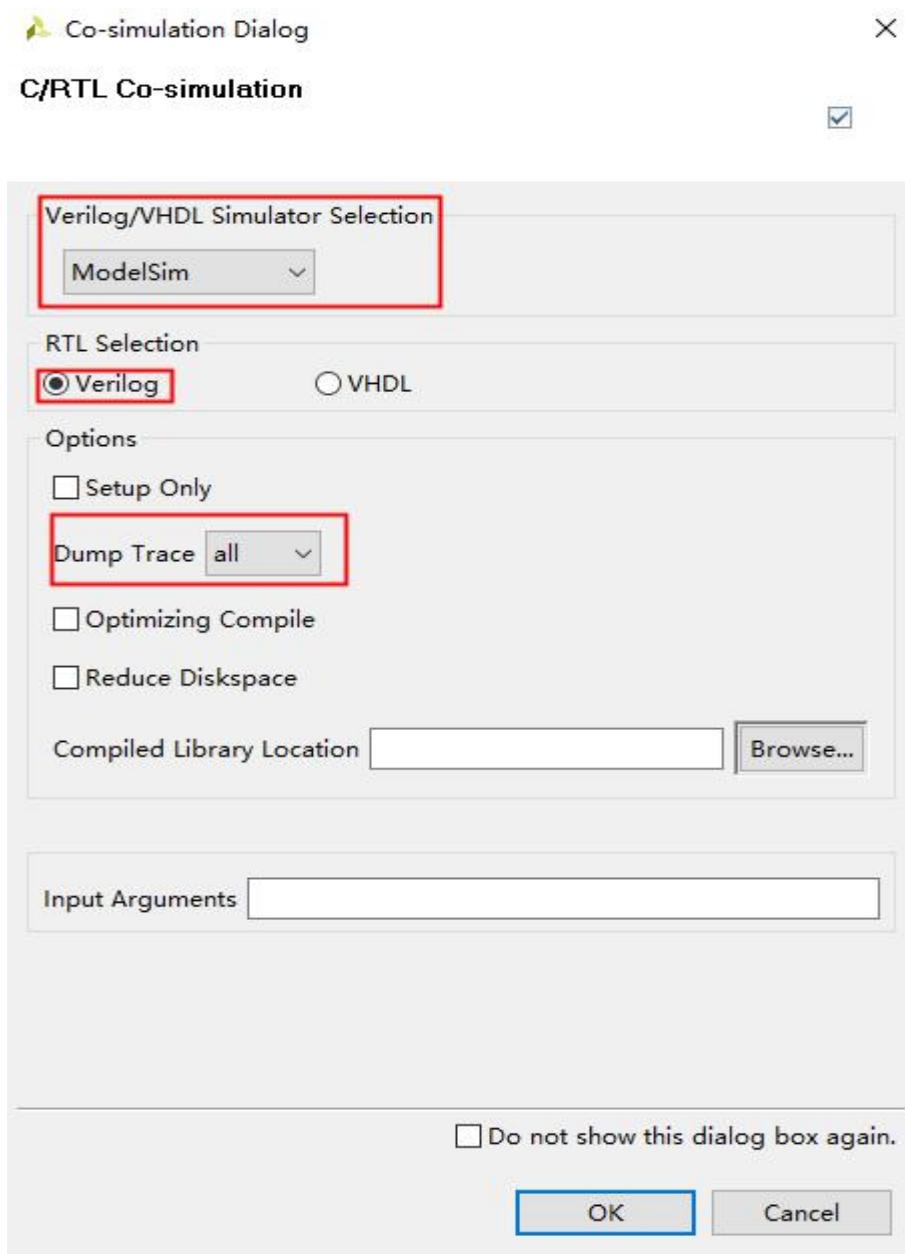
The screenshot shows the Vivado HLS Console output. It displays the command to set the target device and compile the source files, followed by the generated CSIM executable and its execution. The console output shows the shift register's state changing from 0xFE to 0x01 after 8 iterations of shifting.

```

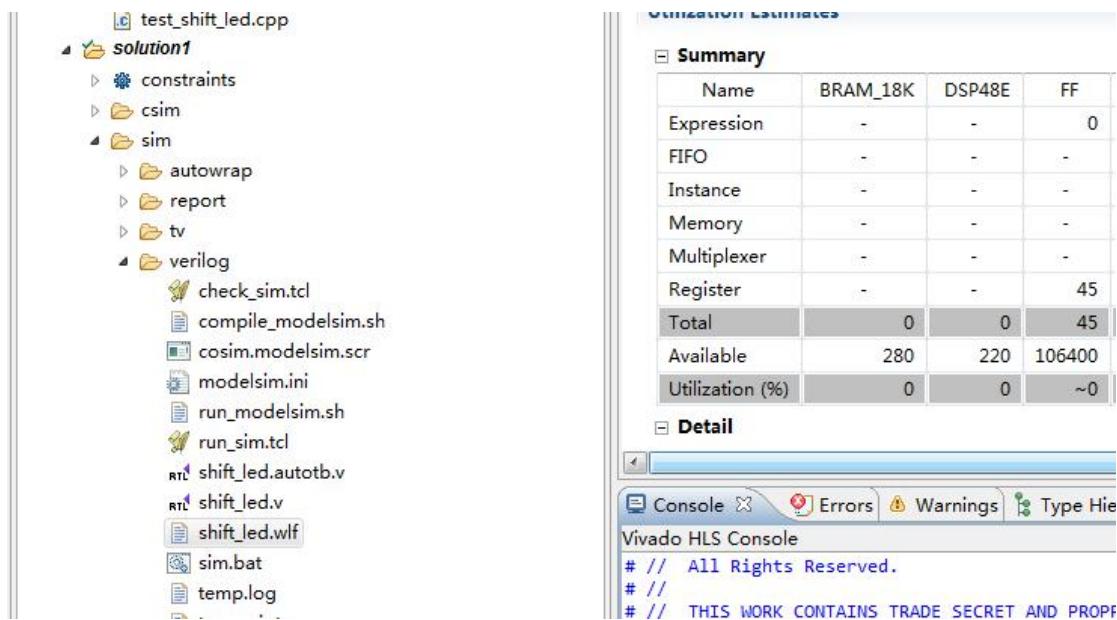
Vivado HLS Console
@I [HLS-101] Setting target device to xc7z020clg484-1
Compiling ../../../../../../src/test_shift_led.cpp in debug mode
Compiling ../../../../../../src/shift_led.cpp in debug mode
Generating csim.exe
shift_out= 11111101
shift_out= 11111011
shift_out= 11110111
shift_out= 11101111
shift_out= 11011111
shift_out= 10111111
shift_out= 01111111
shift_out= 11111110
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [VIVADO_HLS]

```

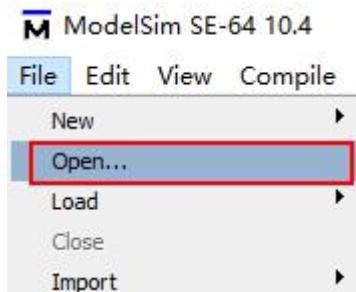
Step3：单击 Solution 下的 Run C/RTL cosimulation 运行协同仿真。



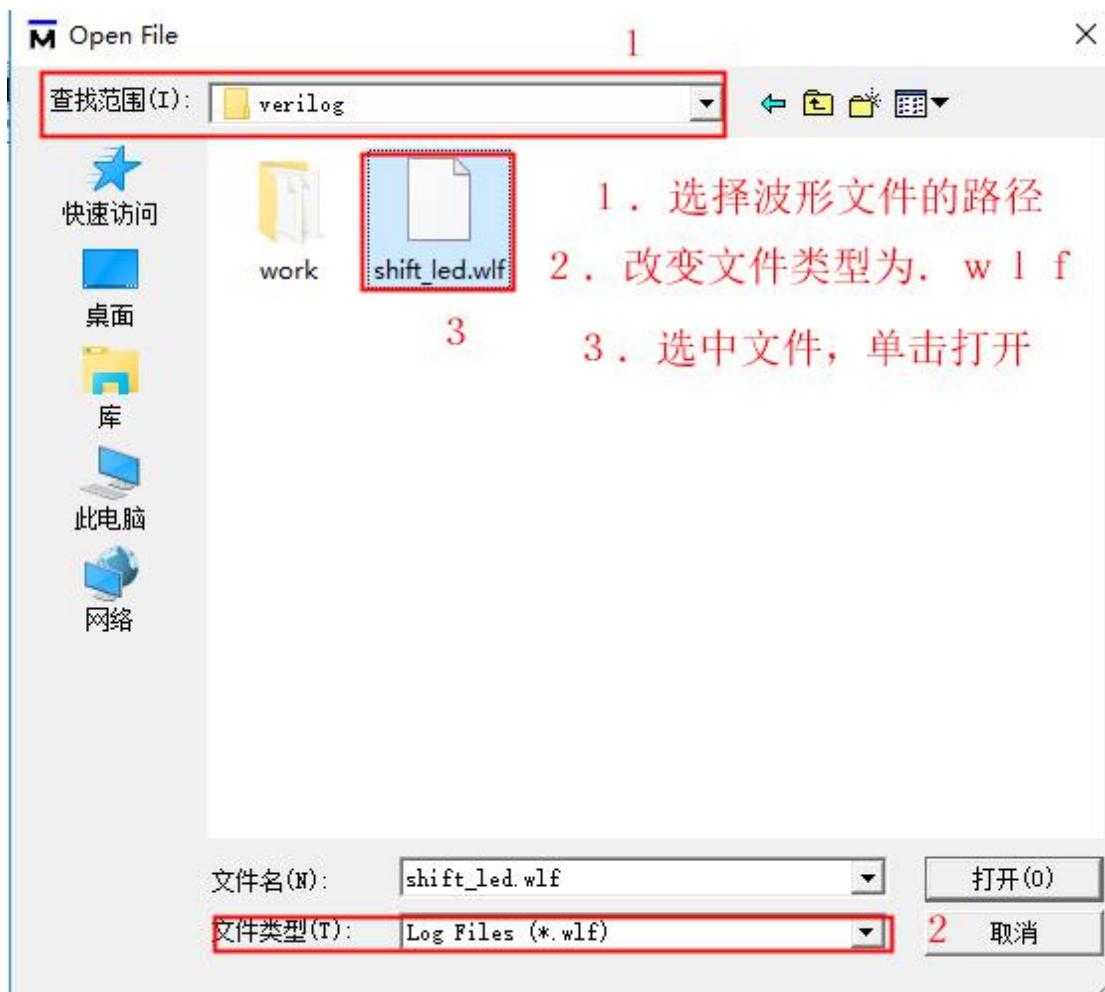
Step4：运行协同仿真一段时间后，我们可以发现在 solution1 目录下多了一个 sim 文件夹，在其 verilog 文件夹下我们可以看到生成的波形文件 shift_led.wlf。



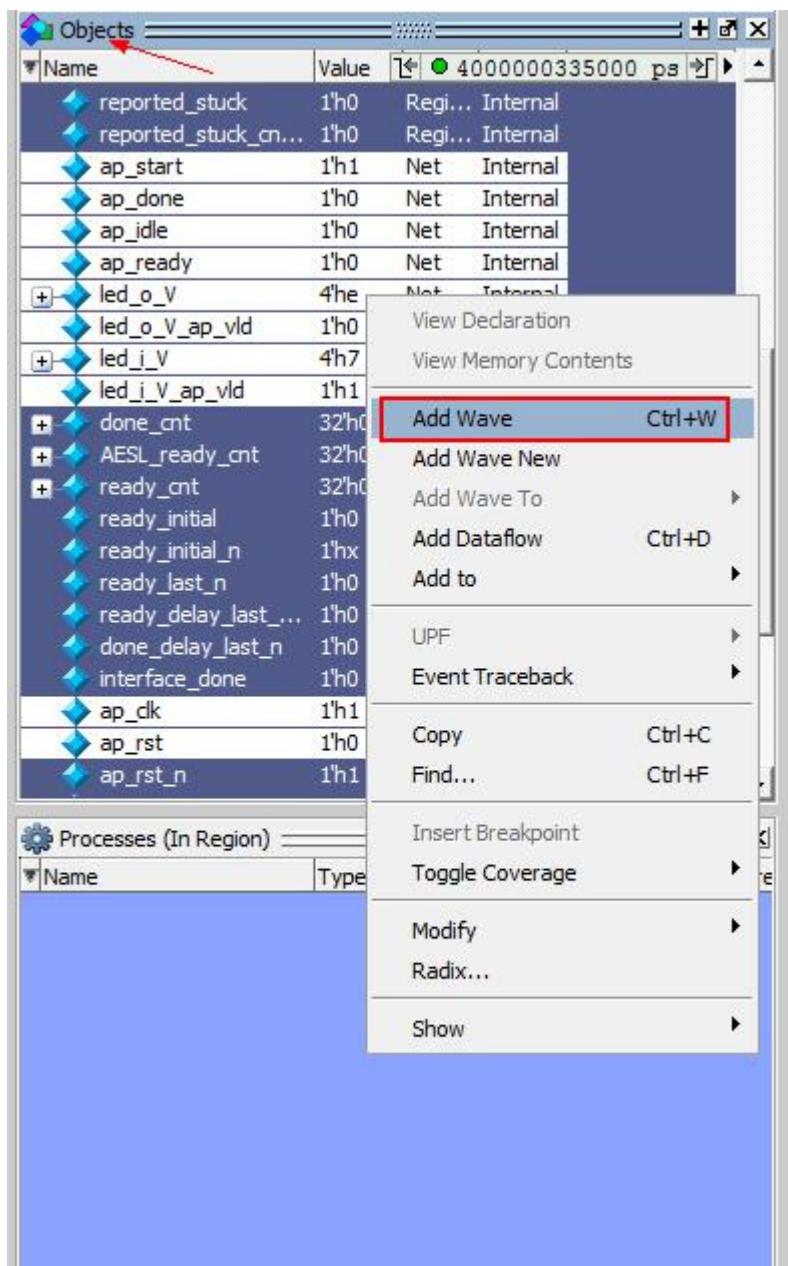
Step5: 通过 modelsim 打开该文件，我们看一下关键接口的时序。首先打开 modelsim，然后单击 File 菜单下的 open 命令。



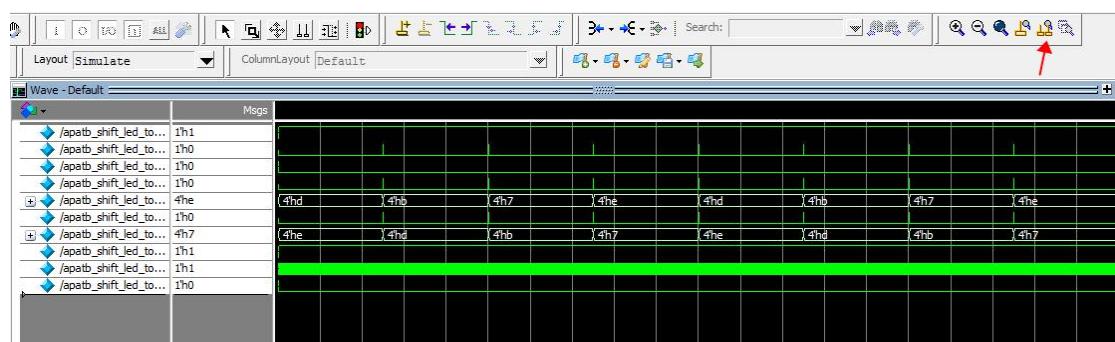
Step6: 打开刚才生成的波形文件 shift_led.wlf 的路径，将文件类型改为*.wlf，然后选中 shift_led.wlf，单击打开按钮。



Step7: 在 objects 设置区，按住 ctrl 键选中要查看波形的信号，然后右单击选择 Add wave 命令。



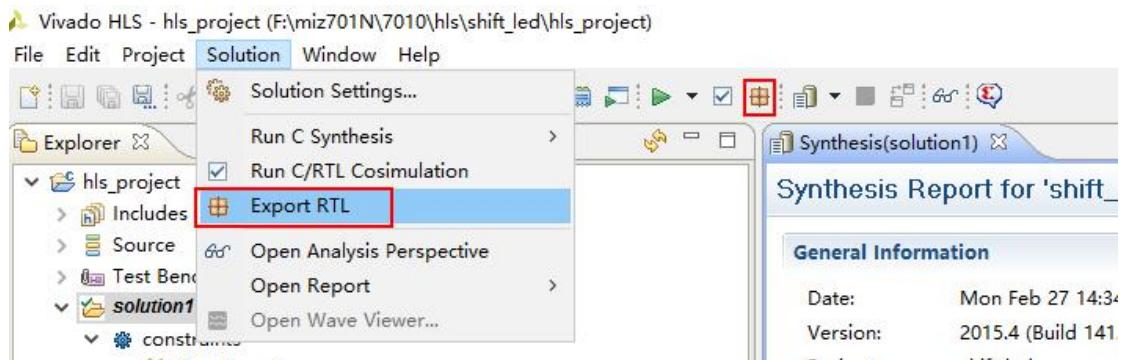
Step8：选中波形显示区，然后单击 调整波形，方便查看功能是否正确。



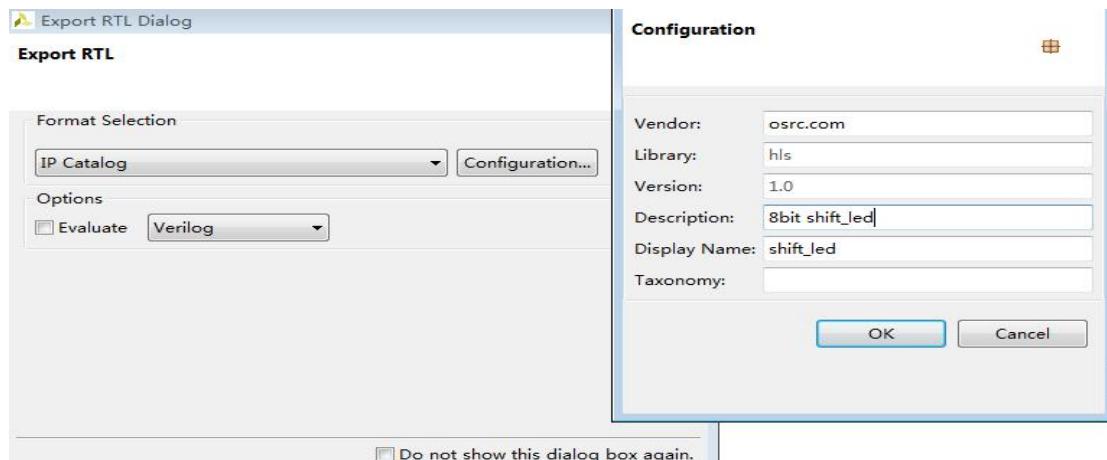
2.3 HLS 代码封装

通过前面的学习，我们了解了 HLS 基本的工程创建，仿真及优化技巧，但是 HLS 只是把你的算法实现从 C 到 RTL 的转化，而不能在硬件平台上进行测试，这一节我们就讲解一下如何把 HLS 工程打包成一个 IP 以便于 Vivado 进行调用。

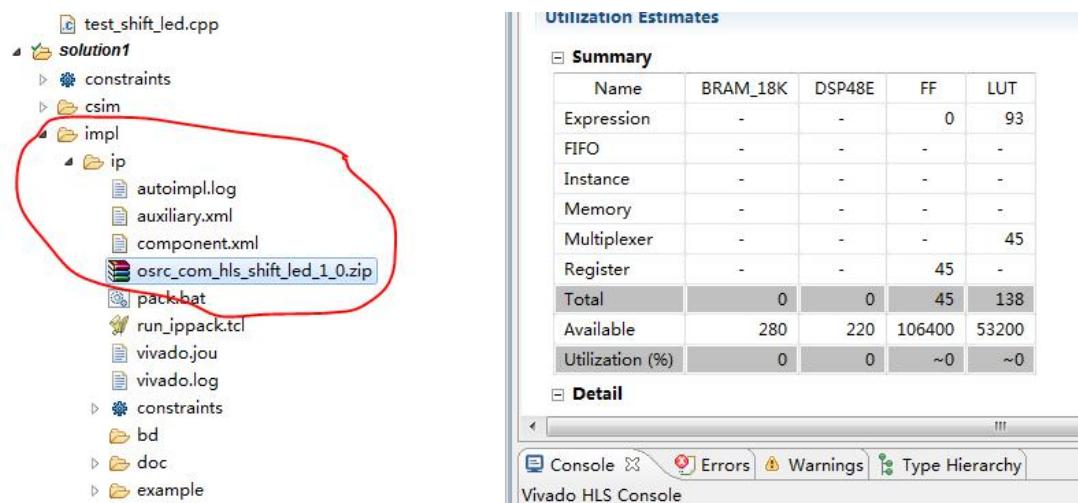
Step1：单击 Solution 菜单下的 Export RTL 或直接单击  导出 RTL 级。



Step2：在弹出的窗口中，点击 configuration 对一些参数进行补充，单击两次 OK。



Step3：等待一段时间后在 solution1 目录下多了一个 impl 文件夹，并且在 ip 文件夹生成了一个压缩包，这就是打包好的 IP，后续我们可以在 Vivado 中进行使用。

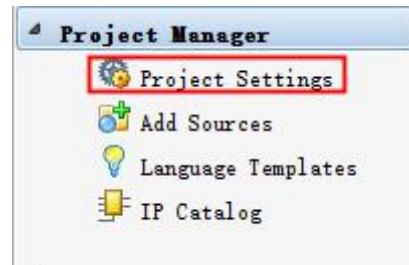


2.4 硬件平台实现

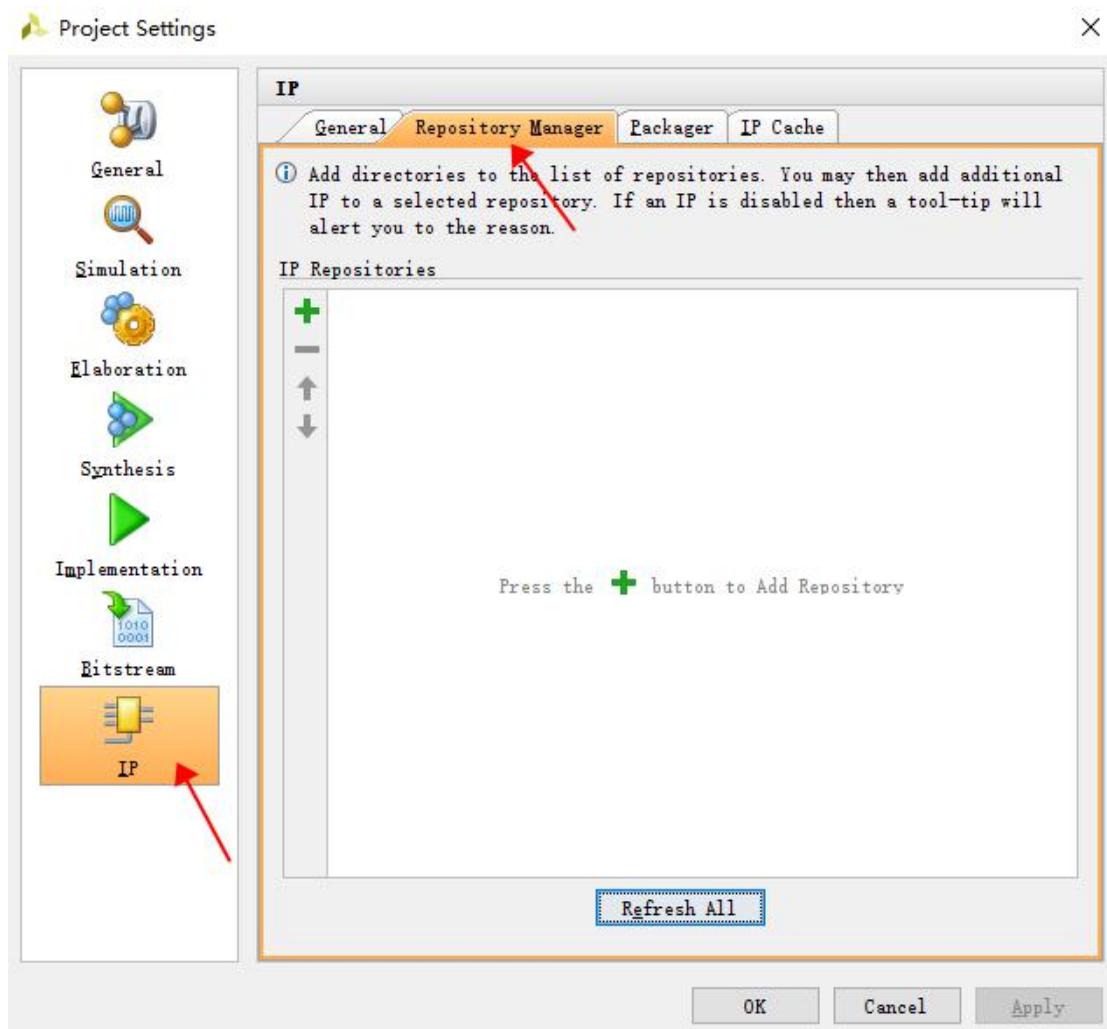
HLS 生成了 IP 之后，接下来的任务就是验证了。将刚才生成的 IP 解压，然后创建一个新的文件夹用来存放 VIVADO 工程。

Step1：创建一个新的 VIVADO 工程。

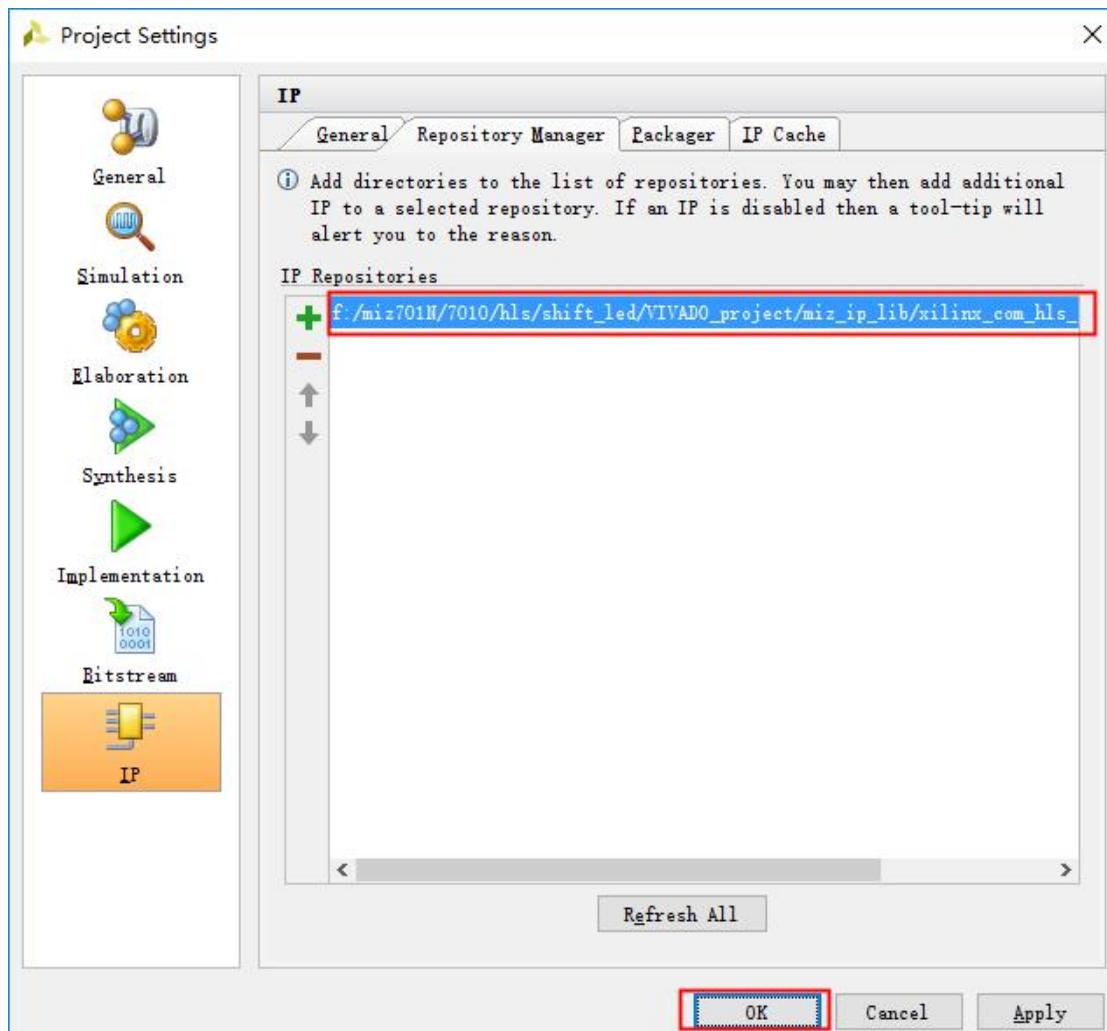
Step2：在 project Manager 中单击 Project Setting。



Step3：选择 IP 设置区的 Repository Manager 设置区。



Step4: 单击 将 IP 的路径添加进去，最后单击 OK 完成设置。



Step5：创建一个名为 shift_led 的 Verilog 文件，并添加如下程序（Miz701N 因为只有 4 个 LED，所以程序中的 DATA_WIDTH 应该改为 4）。

Miz701N 用户添加如下程序

```
module shift_led
#(
    parameter DATA_WIDTH = 4 // 鎏版嵁缩夸綵淪?
)
(
    input          i_clk,
    input          i_rst_n,
    output reg [DATA_WIDTH-1:0] led
);

reg      [1:0] cnt      ;
reg  [DATA_WIDTH-1:0] led_i_V  ;
wire      ap_start  ;
wire      led_i_vld;
wire  [DATA_WIDTH-1:0] led_o_V  ;
```

```
always@(posedge i_clk or negedge i_rst_n)begin
    if(i_rst_n == 1'b0)
        cnt <= 2'd0;
    else if(cnt[1]==1'b0)
        cnt <= cnt + 1'b1;
end

always@(posedge i_clk or negedge i_rst_n)begin
    if(i_rst_n == 1'b0)
        led_i_V <= 2'd0;
    else if(cnt[0]==1'b1)
        led_i_V <= 4'h1;//LED 錄潰 鑷?
    else if(led_o_vld == 1'b1)
        led_i_V <= led_o_V;
end

always@(posedge i_clk or negedge i_rst_n)begin
    if(i_rst_n == 1'b0)
        led <= 1'b0;
    else if(led_o_vld == 1'b1)
        led <= led_o_V;
end

assign ap_start  = cnt[1];
assign led_i_vld = cnt[1];

shift_led_0 u_shift_led_0(
    .led_o_V_ap_vld      (led_o_vld),//  output  wire
    led_o_vld
    .led_i_V_ap_vld     (led_i_vld),// input wire led_i_vld
    .ap_clk              (i_clk      ),// input wire ap_clk
    .ap_rst              (~i_rst_n ),// input wire ap_rst
    .ap_start             (ap_start   ),// input wire ap_start
    .ap_done              (          ),// output wire ap_done
    .ap_idle              (          ),// output wire ap_idle
    .ap_ready             (          ),// output wire ap_ready
    .led_i_V              (led_i_V   ),// output wire [7 : 0]
    led_o_V
    .led_o_V              (led_o_V   ) // input wire [7 : 0]
    led_i_V
);

```

```
endmodule
```

MIZ702 和 MIZ702N 添加如下程序

```
module shift_led
#(
    parameter DATA_WIDTH = 8 // 鑄版堪綰夸綴淪?
)
(
    input          i_clk,
    input          i_rst_n,
    output reg [DATA_WIDTH-1:0] led
);

reg [1:0] cnt      ;
reg [DATA_WIDTH-1:0] led_i_V ;
wire          ap_start ;
wire          led_i_vld;
wire [DATA_WIDTH-1:0] led_o_V ;

always@(posedge i_clk or negedge i_rst_n)begin
    if(i_rst_n == 1'b0)
        cnt <= 2'd0;
    else if(cnt[1]==1'b0)
        cnt <= cnt + 1'b1;
end

always@(posedge i_clk or negedge i_rst_n)begin
    if(i_rst_n == 1'b0)
        led_i_V <= 2'd0;
    else if(cnt[0]==1'b1)
        led_i_V <= 8'h1;
    else if(led_o_vld == 1'b1)
        led_i_V <= led_o_V;
end

always@(posedge i_clk or negedge i_rst_n)begin
    if(i_rst_n == 1'b0)
        led <= 1'b0;
    else if(led_o_vld == 1'b1)
        led <= led_o_V;
end

assign ap_start = cnt[1];
assign led_i_vld = cnt[1];
```

```

shift_led_0 u_shift_led_0(
    .led_o_V_ap_vld  (led_o_vld), // output wire led_o_vld
    .led_i_V_ap_vld  (led_i_vld), // input wire led_i_vld
    .ap_clk          (i_clk      ), // input wire ap_clk
    .ap_rst          (~i_rst_n ), // input wire ap_rst
    .ap_start         (ap_start   ), // input wire ap_start
    .ap_done          (          ), // output wire ap_done
    .ap_idle          (          ), // output wire ap_idle
    .ap_ready         (          ), // output wire ap_ready
    .led_i_V          (led_i_V   ), // output wire [7 : 0]
    led_o_V
        .led_o_V          (led_o_V   ) // input wire [7 : 0]
    led_i_V
);
endmodule

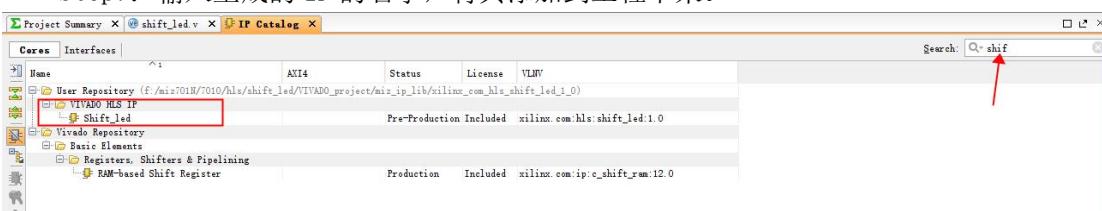
```

这里例化的 shift_led_0 就是我们刚刚用 HLS 创建的 IP，这个程序的编写也是依据其时序来编写的。

Step6: 单击 Project Manager 中的 IP catalog。



Step7: 输入生成的 IP 的名字，将其添加到工程中来。



Step8: 创建一个名为 zynq_pin 的约束文件，并根据自己的硬件平台建立约束。此处以 Miz702 开发板为例。Miz702 开发板约束文件如下所示：

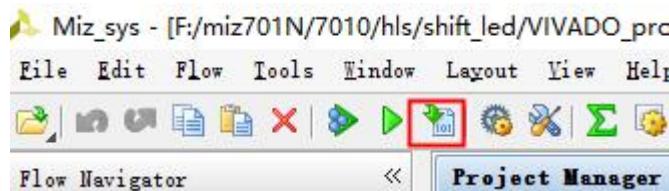
```

set_property IOSTANDARD LVCMOS33 [get_ports i_clk]
set_property IOSTANDARD LVCMOS33 [get_ports i_rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports {led[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[1]}]

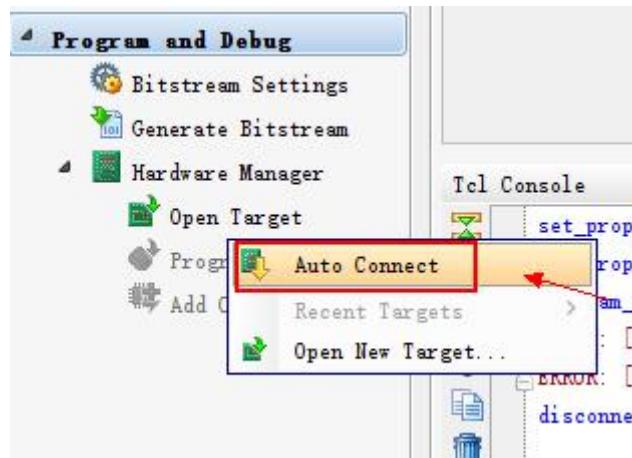
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {led[0]}]
set_property PACKAGE_PIN T22 [get_ports {led[7]}]
set_property PACKAGE_PIN T21 [get_ports {led[6]}]
set_property PACKAGE_PIN U22 [get_ports {led[5]}]
set_property PACKAGE_PIN U21 [get_ports {led[4]}]
set_property PACKAGE_PIN V22 [get_ports {led[3]}]
set_property PACKAGE_PIN W22 [get_ports {led[2]}]
set_property PACKAGE_PIN U19 [get_ports {led[1]}]
set_property PACKAGE_PIN U14 [get_ports {led[0]}]
set_property PACKAGE_PIN P16 [get_ports i_rst_n]
set_property PACKAGE_PIN Y9 [get_ports i_clk]
```

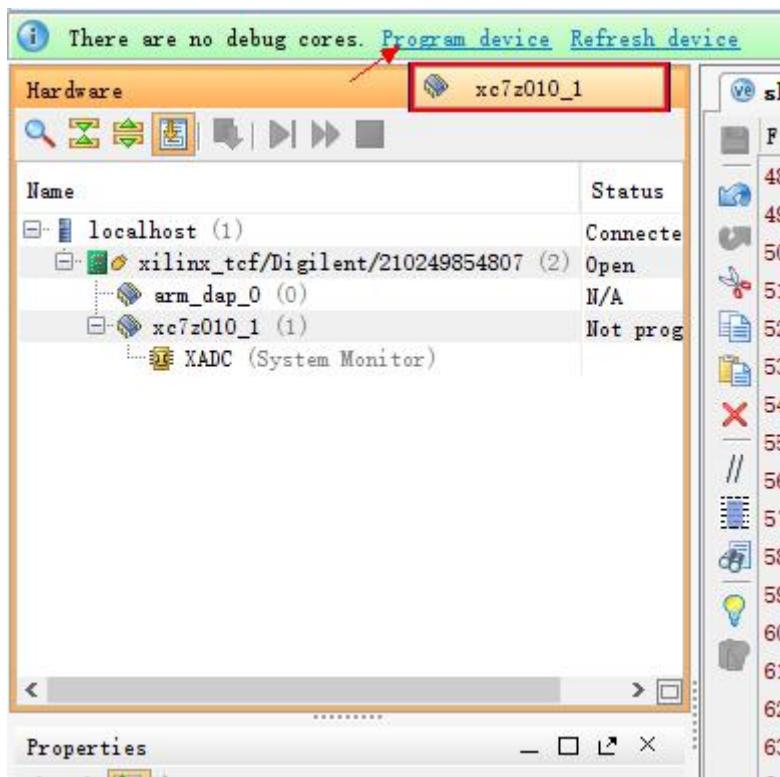
Step9: 单击 生成 bit 文件。



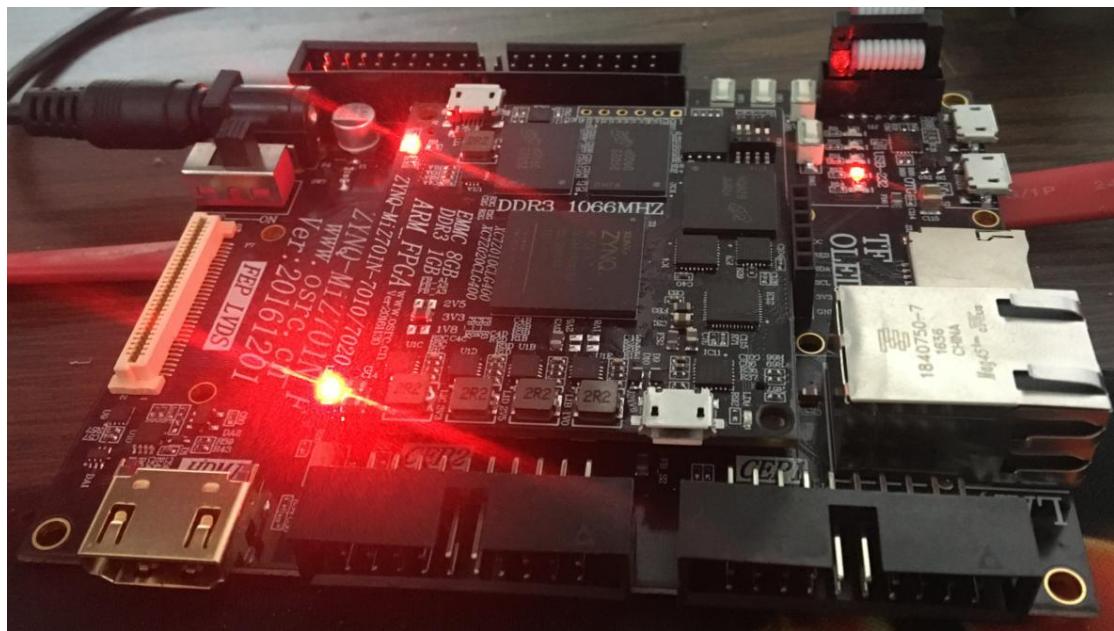
Step10: bit 文件生成完后，打开开发板电源，在 Program and Debug 设置区中单击 Hardware Manager 左边的三角形，选择 Open Target。



Step11: 下载程序到开发板中。



将程序进行编译下载后，我们可以看到板载 LED 每隔半秒向左移一位循环往复，结果如下图：



2.5 本章小结

本章通过一个简单的流水灯程序，向大家介绍了 HLS 的工程创建过程，以及代码的仿真方法，代码的简单优化等等。通过本章，大家需要掌握 HLS 的基本开发流程，为下面的课程打下基础。

S05_CH03_ImageLoad 实验

3.1 概述

通过前面的学习，我们已经基本熟悉了 HLS 的开发流程，那么在 HLS 上进行算法开发，算法实现综合后我们需要对算法的正确性进行验证，那么我们如何得到我们想要的视频图像数据流，对算法模块进行测试呢？在这一章我们就这些问题讨论一下如何搭建验证平台，主要包括图像、视频数据流以及外部摄像头的调用，方便对算法模块进行测试。

3.2 图片数据的获取

当我们进行前期算法验证的时候，需要读取图片进行仿真，那么关键的一步就是如何加载图片进行测试，在 HLS 中比较基础的两种加载图片的方法如下：

通过 cvLoadImage 函数加载图片，其格式如下：

```
IplImage* src = cvLoadImage(INPUT_IMAGE);
```

```
cvShowImage("src", src);
```

函数原型： IplImage* cvLoadImage(const char*filename, int iscolor

CV_DEFAULT(CV_LOAD_IMAGE_COLOR));

filename : 要被读入的文件的文件名(包括后缀)；

iscolor: 指定读入图像的颜色和深度；

指定的颜色可以将输入的图片转为 3 信道(CV_LOAD_IMAGE_COLOR)，单信道

(CV_LOAD_IMAGE_GRAYSCALE)，或者保持不变(CV_LOAD_IMAGE_ANYCOLOR)。

cvLoadImage 函数使用方法有下面几种：

```
cvLoadImage( filename, -1 ); 默认读取图像的原通道数
```

```
cvLoadImage( filename, 0 ); 强制转化读取图像为灰度图
```

```
cvLoadImage( filename, 1 ); 读取彩色图
```

我们在这里演示一个通过 cvLoadImage 函数读取图片显示的例子，源码及结果如下图：

```
IplImage* src = cvLoadImage(INPUT_IMAGE);
```

```
IplImage* dst = cvCreateImage(cvGetSize(src), src->depth, src->nChannels); // 获取原始图像大小
```

```
AXI_STREAM src_axi, dst_axi;
```

```
IplImage2AXIVideo(src, src_axi);
AXIVideo2IplImage(src_axi, dst);
cvSaveImage(OUTPUT_IMAGE, dst);
cvShowImage("result_1080p", dst);
```



通过 `imread` 函数读取图片，格式如下：

```
Mat src_rgb = imread(INPUT_IMAGE);

IplImage src = src_rgb;

cvShowImage("src", &src);
```

首先，我们看 `imread` 函数，可以在官方文档中查到其原型如下：

```
CV_EXPORTS_W Mat imread( const string& filename, int flags=1 );
```

第一个参数，`const string&`类型的 `filename`，填我们需要载入的图片路径名。

第二个参数，`int`类型的 `flags`，为载入标识，它指定一个加载图像的颜色类型。可以看到它自带缺省值 1. 所以有时候这个参数在调用时我们可以忽略，在看了下面的讲解之后，我们就会发现，如果在调用时忽略这个参数，就表示载入三通道的彩色图像。通过转到定义，我们可以在 `highgui_c.h` (192–204 行) 中发现这个枚举的定义是这样的：

```
enum
{
/* 8bit, color or not */
CV_LOAD_IMAGE_UNCHANGED = -1,
/* 8bit, gray */
CV_LOAD_IMAGE_GRAYSCALE = 0,
/* ?, color */
CV_LOAD_IMAGE_COLOR = 1,
```

```
/* any depth, ? */
CV_LOAD_IMAGE_ANYDEPTH =2,
/* ?, any color */
CV_LOAD_IMAGEANYCOLOR =4
};
```

我们通过 imread 函数读取一副图片并且灰度显示的代码如下：

```
Mat src_rgb = imread(INPUT_IMAGE, CV_LOAD_IMAGE_GRAYSCALE); //加载图片并灰度显示
IplImage src = src_rgb;
cvSaveImage(OUTPUT_IMAGE, &src);
cvShowImage("src", &src);
```



3.3 视频流文件的载入

cvCaptureFromAVI 函数进行视频文件的载入，

格式： cvCaptureFromAVI ("AVI 文件名称")；

功能： 用来播放 AVI 文件视频；我们在 highgui_c.h 文件可以看到有如下定义：

```
#define cvCaptureFromFile cvCreateFileCapture
```

```
#define cvCaptureFromAVI cvCaptureFromFile
```

说明： 所以用 cvCaptureFromAVI () 跟 cvCaptureFromFile (), cvCreateFileCapture () 都是一样的作用；文件的类型不一定必须是 AVI 格式，只要文件符合 OpenCV 支持的格式就能播放。

格式： int cvGrabFrame (CvCapture 结构体)；

功能： 将 capture 抓下来的相片放在 OpenCV 中；其与 cvQueryFrame () 是相同的步骤；

cvGrabFrame() 返回值为 0 或 1； 0 是失败，1 是成功。

格式： cvRetrieveFrame(CvCapture 结构)；

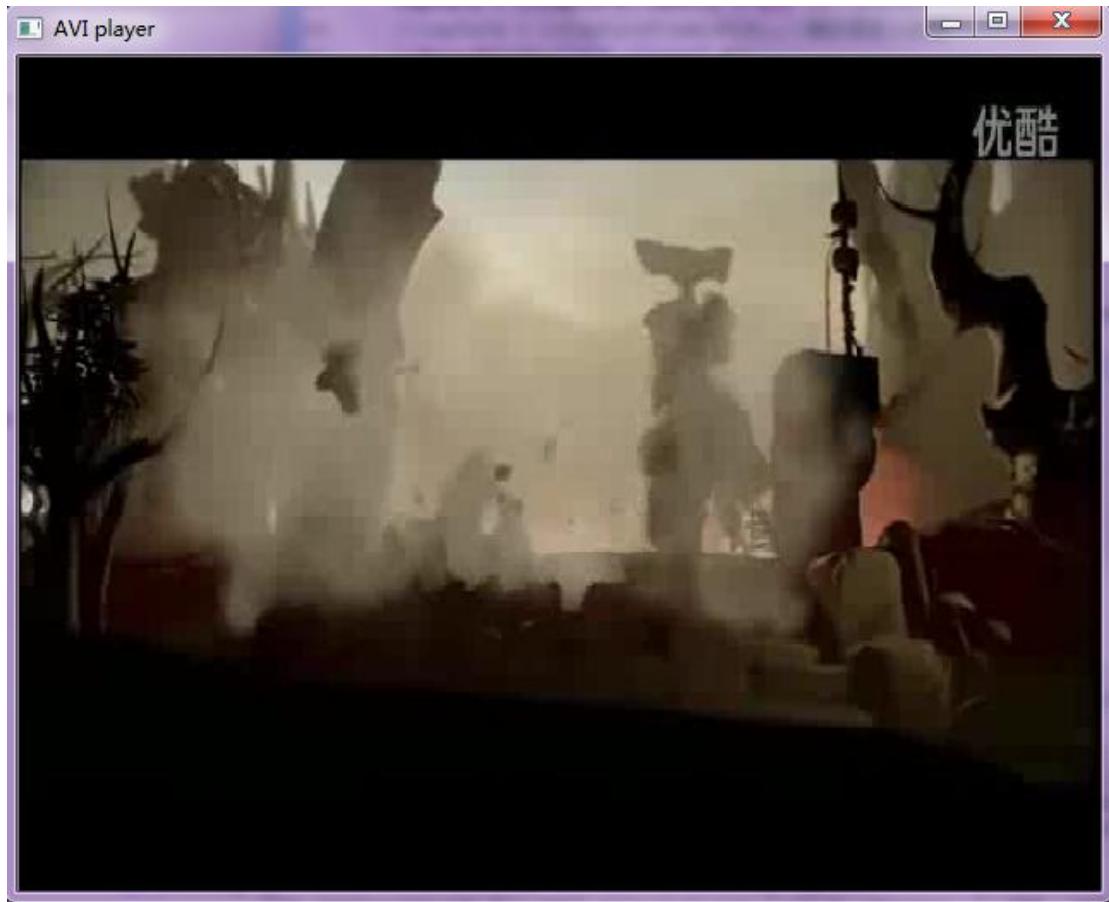
功能：从 OpenCV 快取中得到 Frame，并配置给 IplImage 结构体；其中：

cvQueryFrame()=cvGrabFrame() +cvRetrieveFrame().

我们通过一个实例让大家理解这几个函数的使用

```
IplImage *frame;  
  
CvCapture *capture = cvCaptureFromAVI("1.avi");//获取视频数据  
  
cvNamedWindow("AVI player", 0);  
  
while(true)  
{  
    if(cvGrabFrame(capture))  
    {  
        frame = cvRetrieveFrame(capture);  
  
        cvShowImage("AVI player", frame);  
  
        if(cvWaitKey(10)>=0) break;  
    }  
}
```

通过运行仿真，我们截取的视频画面如下：



3.4 外部摄像头的调用

```
CvCapture*cvCaptureFromCAM( int index );
```

参数: index

要使用的摄像头索引。释放这个结构, 使用函数 cvReleaseCapture。

要将视频写入文件中, 使用 cvWriteFrame 写入一帧到一个视频文件中

```
int cvWriteFrame( CvVideoWriter* writer, const IplImage* image );
```

通过摄像头捕获视频数据的关键代码如下, 并且通过调用外部 USB 摄像头成功采集到视频数据, 为后期算法验证提供了测试依据。

```
IplImage *frame;
```

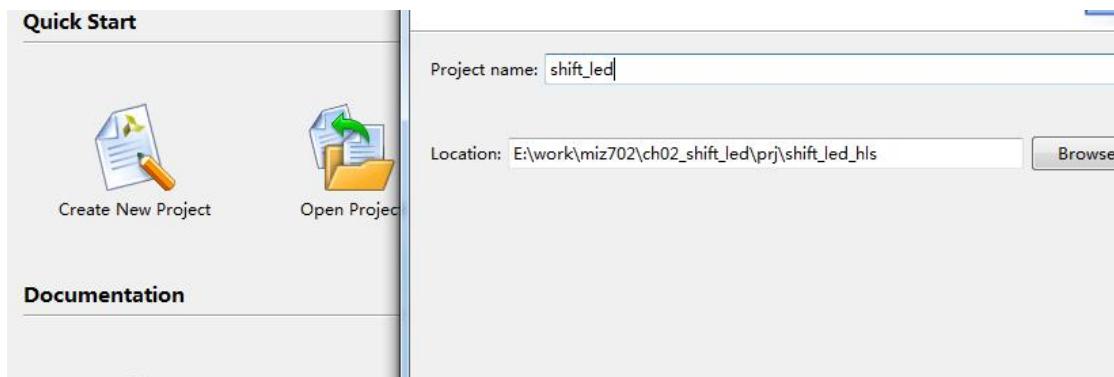
```
CvCapture *capture = cvCaptureFromCAM(1); //捕获摄像头数据0--笔记本自带摄像头 1--外部摄像头
```



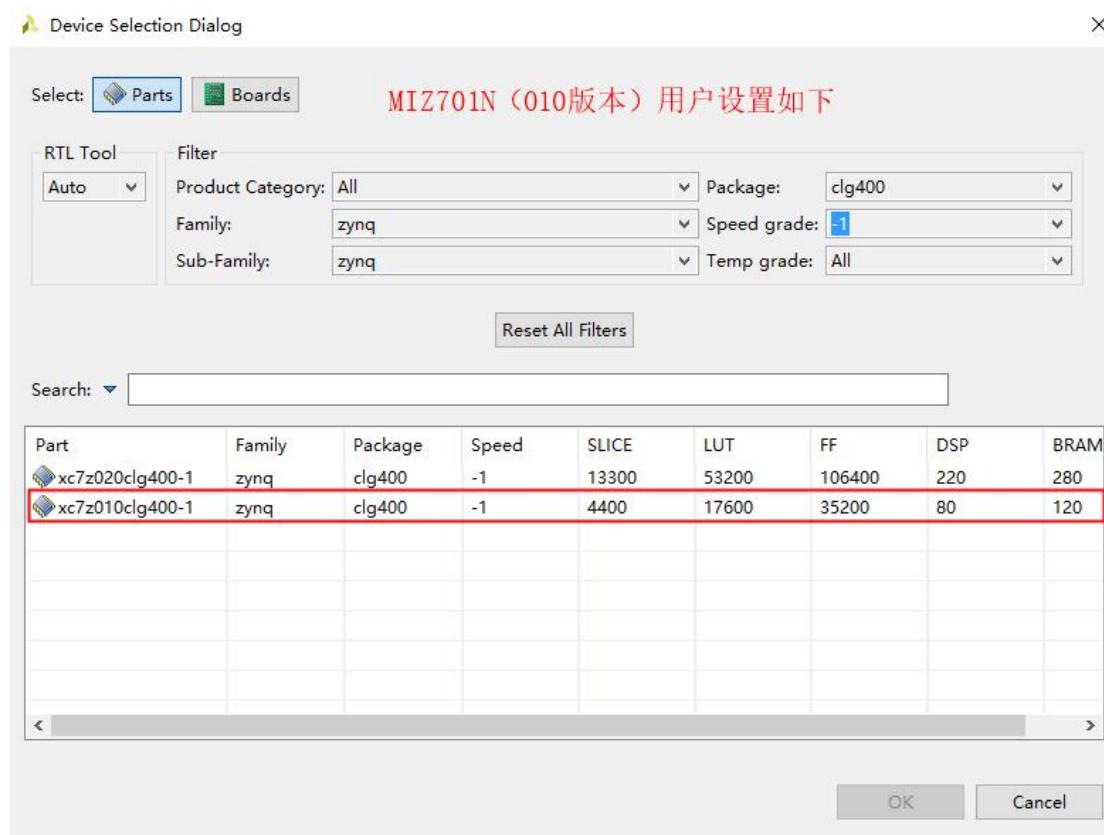
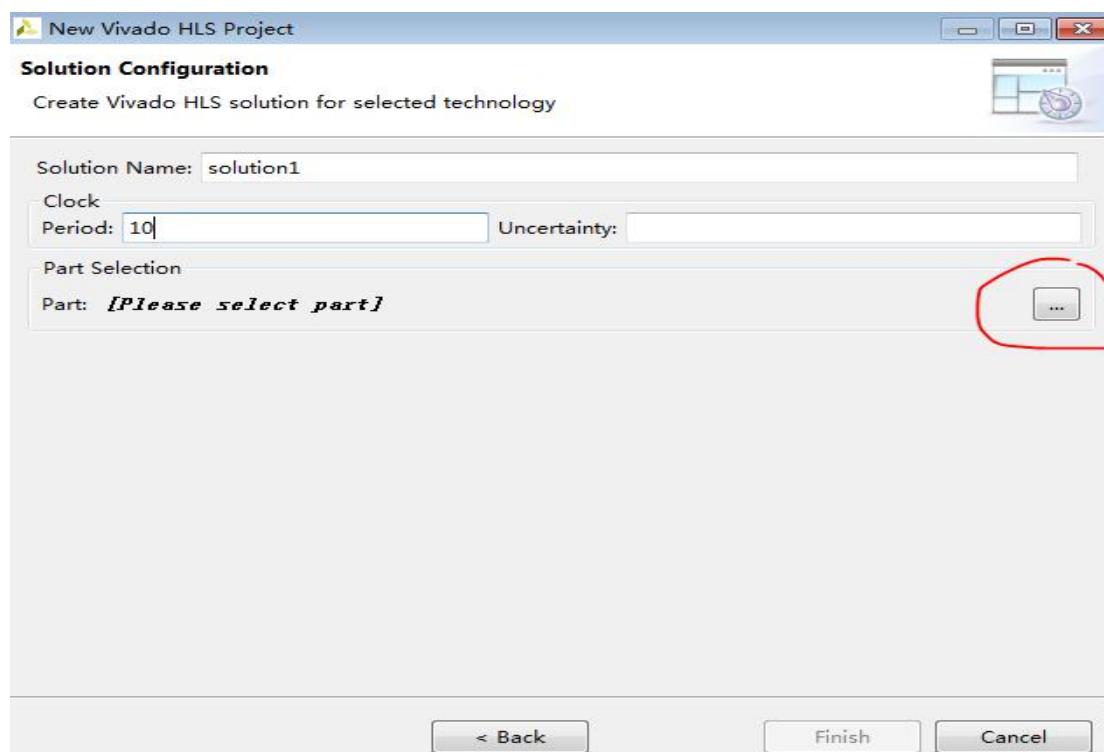
3.5 工程创建与验证

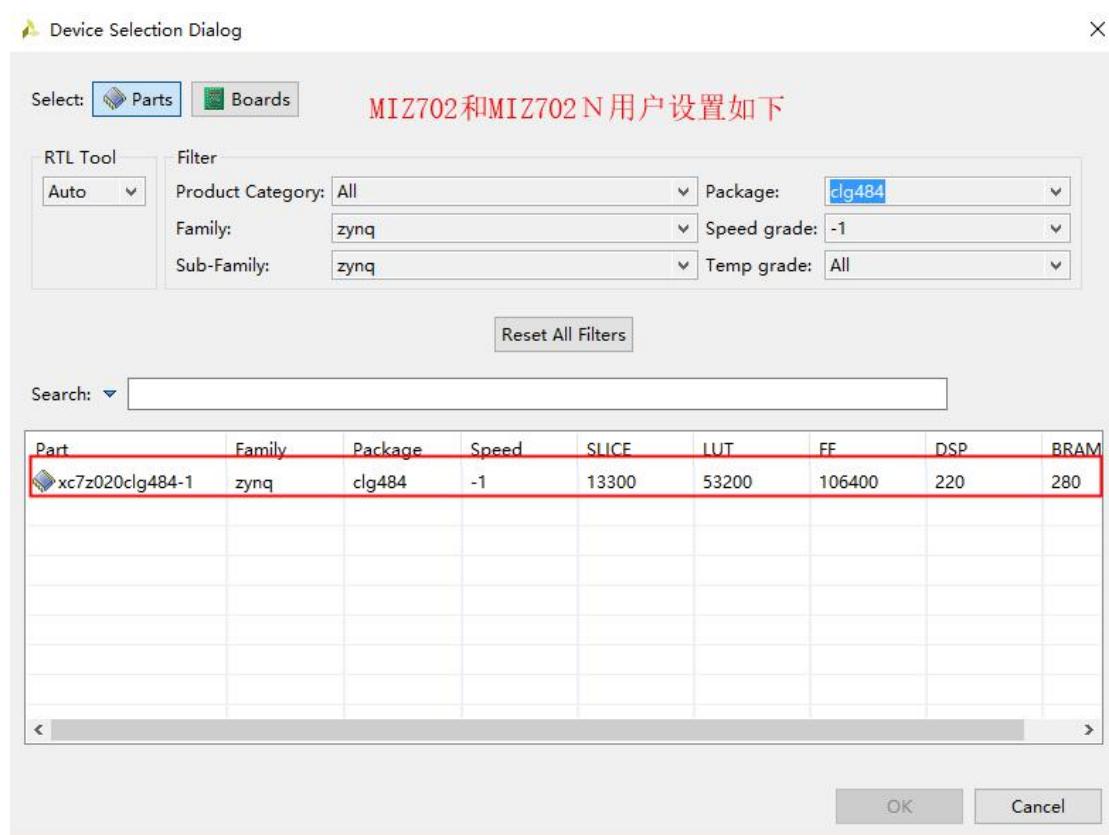
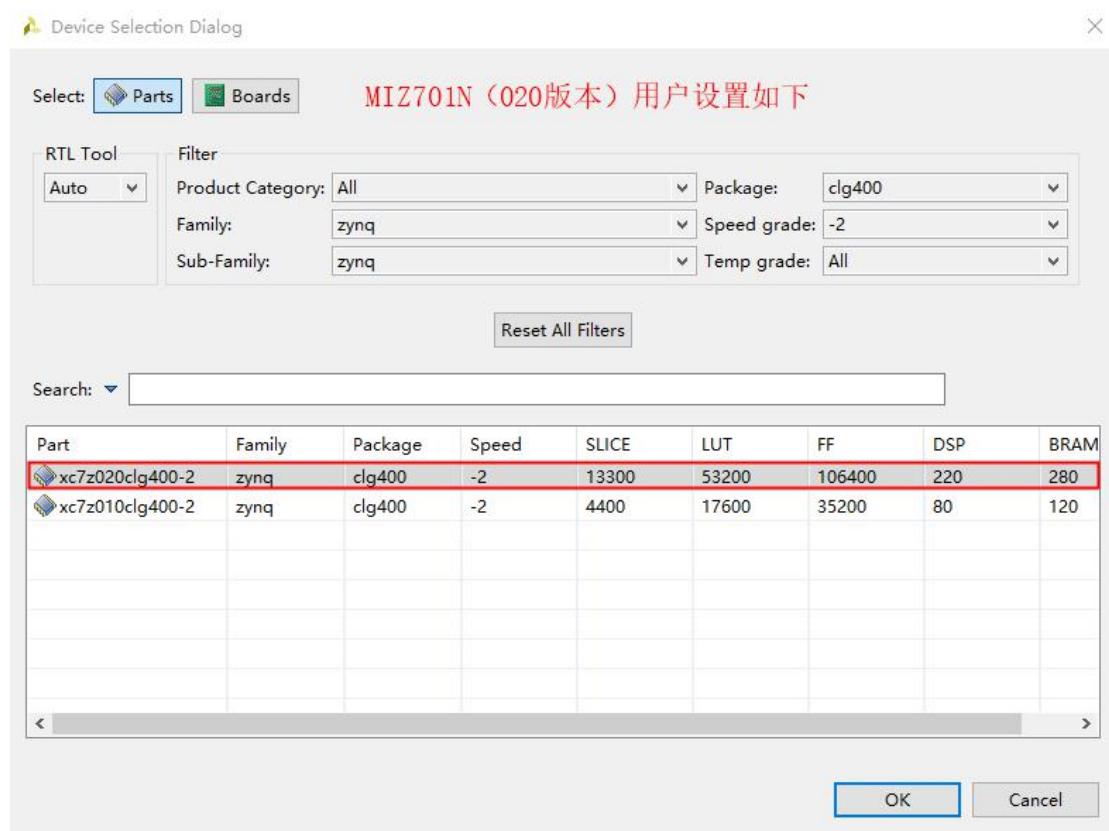
上面介绍完了仿真数据流的获取方法之后，接下来我们创建一个工程对其进行验证。

Step1: 打开 Vivado HLS 开发工具，单击 Create New Project 创建一个新工程，设置好工程路径和工程名，一直点击 Next 按照默认设置。

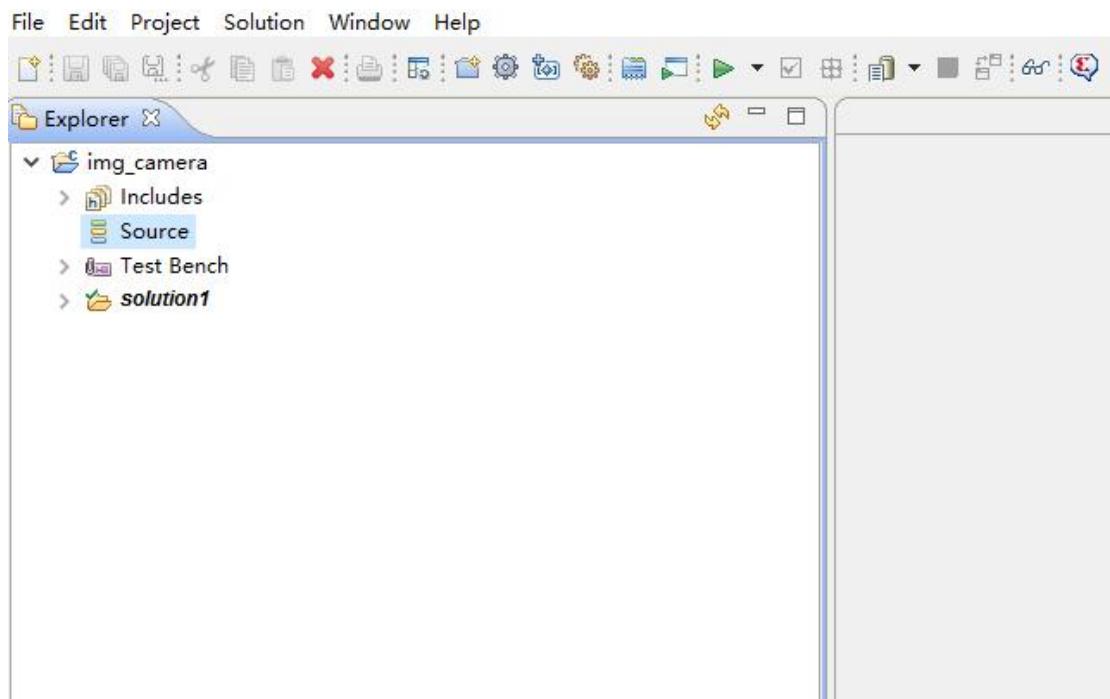


Step2: 出现如下图所示界面，时钟周期 Clock Period 按照默认 10ns，Uncertainty 和 Solution Name 均按照默认设置，点击红色圆圈部分选择芯片类型，然后点击 OK。





Step3: 点击 Finish, 出现如下界面:



Step4：与上一章不同的是，本章只要进行仿真，因此直接单击 Test Bench，添加一个名为 Test.cpp 的测试文件，并添加如下程序。

```
#include "hls_opencv.h"

using namespace cv;

#define INPUT_IMAGE           "test_1080p.bmp"
#define OUTPUT_IMAGE          "result_1080p.bmp"

int main (int argc, char** argv) {

    //方法1 cvLoadImage函数加载图片
    IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels); //获取原始图像大小

    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIvideo(src, src_axi);
    //image_filter(src_axi, dst_axi, src->height, src->width);
    AXIvideo2IplImage(src_axi, dst);

    cvSaveImage(OUTPUT_IMAGE, dst);
    cvShowImage( "result_1080p",dst);
```

```
cvReleaseImage (&src);

cvWaitKey ();

/*
//方法2 cvLoadImage函数加载图片
Mat src_rgb = imread(INPUT_IMAGE,CV_LOAD_IMAGE_GRAYSCALE); //加载
图片并灰度显示

IplImage src = src_rgb;
cvSaveImage(OUTPUT_IMAGE, &src);
cvShowImage("src",&src);
waitKey(0);
return 0;
*/

/*
//读取视频文件
IplImage *frame;
CvCapture *capture = cvCaptureFromAVI("1.avi"); //获取视频数据
cvNamedWindow("AVI player",0);
while(true)
{
    if (cvGrabFrame(capture))
    {
        frame = cvRetrieveFrame(capture);
        cvShowImage("AVI player",frame);
        if (cvWaitKey(10)>=0) break;
    }
    else
    {
        break;
    }
}
cvReleaseCapture(&capture);
cvDestroyWindow("AVI player");

return 0;
*/

/*
//摄像头操作
```

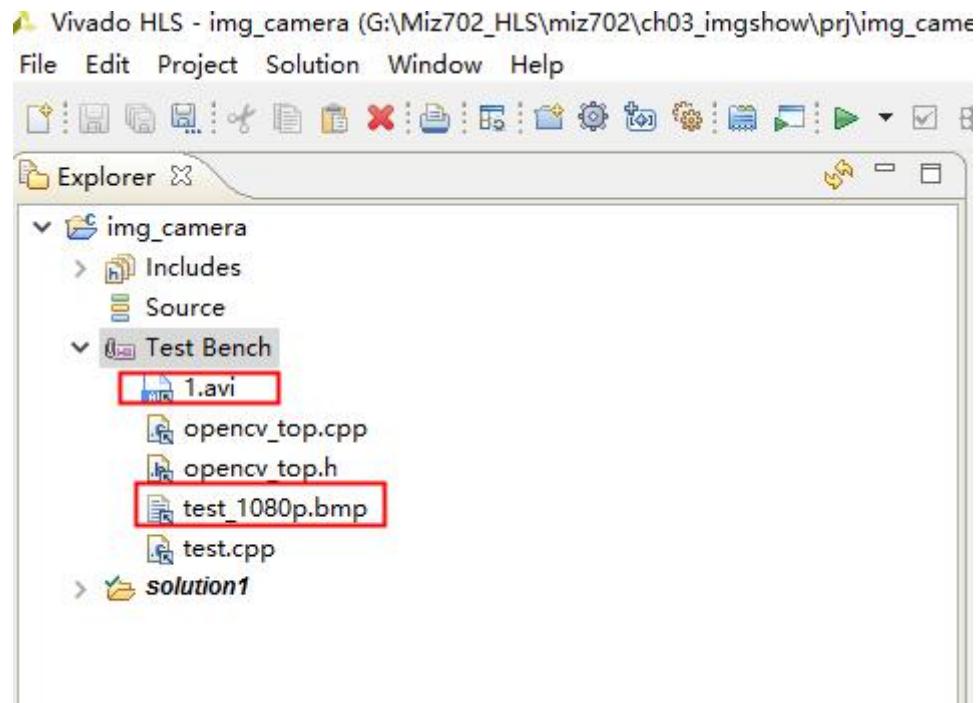
```
IplImage *frame;
CvCapture *capture = cvCaptureFromCAM(1); //捕获摄像头数据0--笔记本
自带摄像头 1--外部摄像头
cvNamedWindow("AVI player", 0);
while(true)
{
    if(cvGrabFrame(capture))
    {
        frame = cvRetrieveFrame(capture);

        cvShowImage("AVI player", frame);
        if(cvWaitKey(10)>=0) break;
    }
    else
    {
        break; //没有采集到视频数据退出
    }
}
cvReleaseCapture(&capture);
cvDestroyWindow("AVI player");

return 0;
*/
}
```

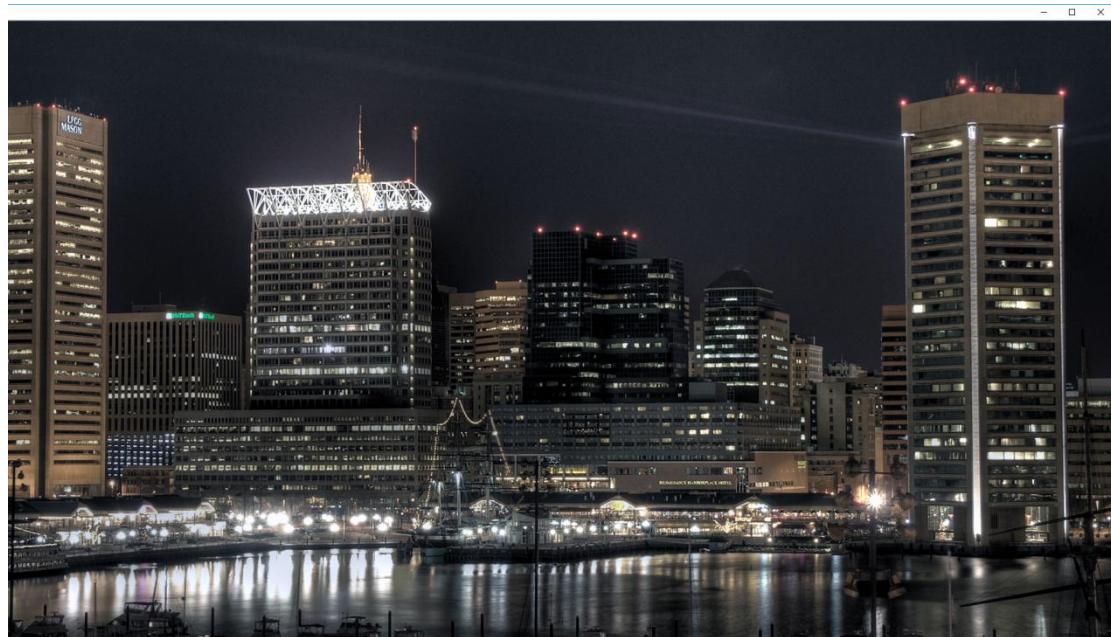
在这个程序中，我们把前 3 节的程序都写到了同一个函数当中，在程序中已经给出了注释，在仿真过程中可以将不必要的代码通过注释的方法屏蔽，分别对这三个部分进行测试。

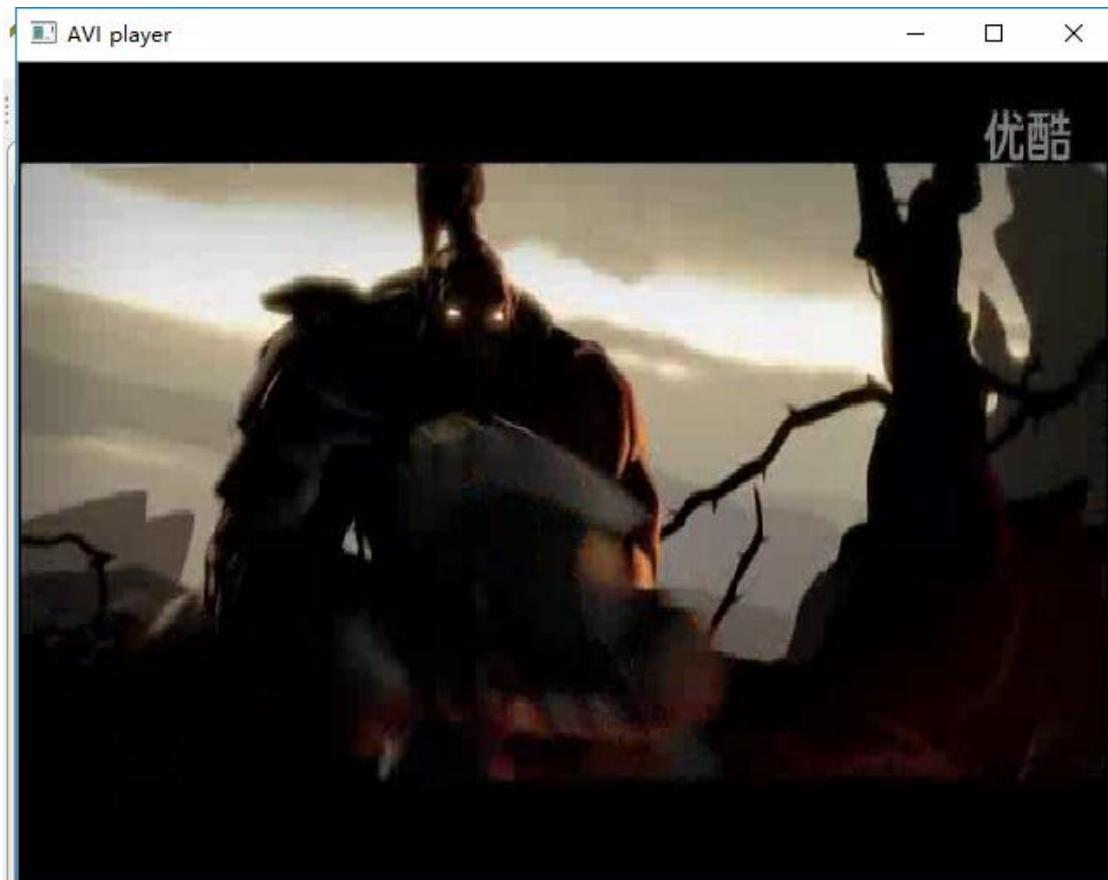
Step5：将要测试的文件添加进 Test Bench 中（文件在我们提供的源程序文件夹中的 image 文件夹中可以找到），添加方法是选中 Test Bench 右单击然后选择 Add File 命令。

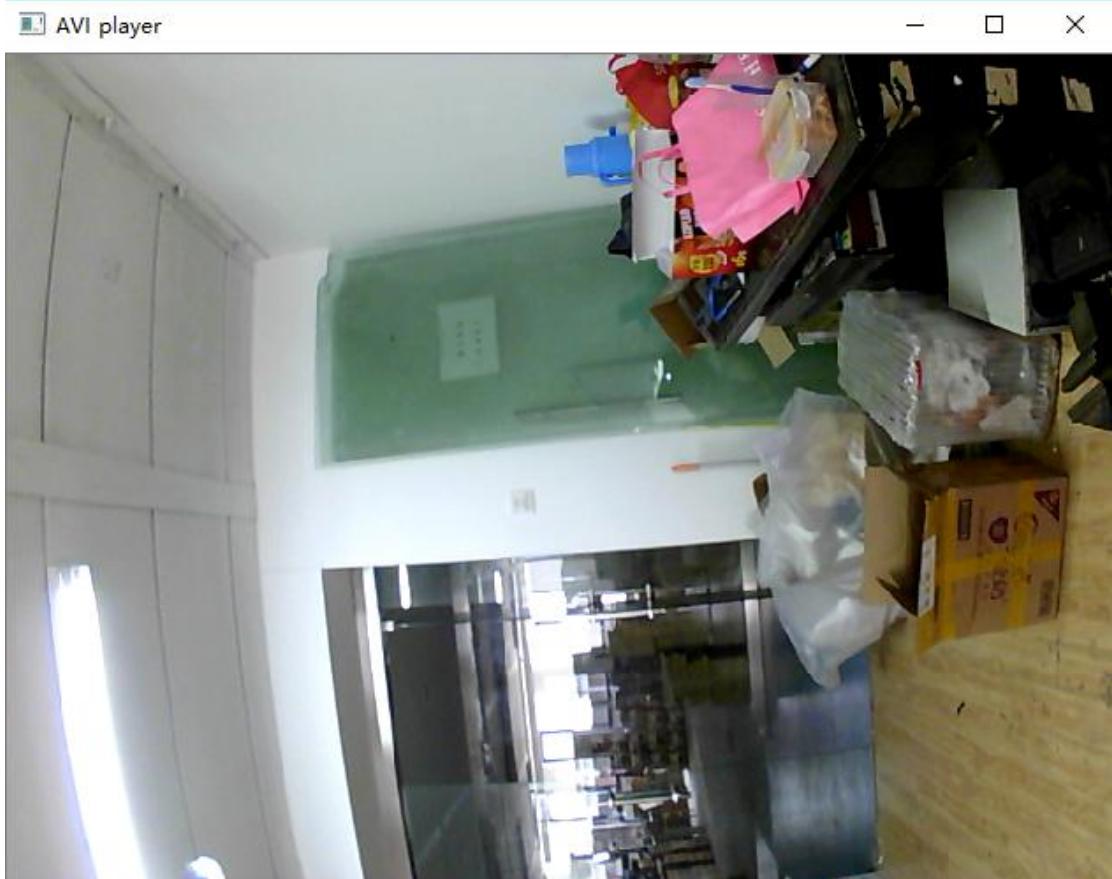


Step6: 单击 开始仿真。

Step7: 运行一段时间后, 运行结果如下所示:







3.6 本章小结

本章介绍了 3 种 HLS 仿真中最常用的获取仿真数据流的方法，并对其进行了验证。在接下来的算法设计中，我们就即将用到本章介绍的这 3 种方法，对设计的功能进行仿真，掌握好这一章的内容以便完成接下来的算法设计。

S05_CH04_Skin_Dection 实验

4.1 肤色检测原理及应用

肤色作为人的体表显著特征之一,尽管人的肤色因为人种的不同有差异,呈现出不同的颜色,但是在排除了亮度和视觉环境等对肤色的影响后,皮肤的色调基本一致,这就为利用颜色信息来做肤色分割提供了理论的依据。

在肤色识别中,常用的颜色空间为 YCbCr 颜色空间。在 YCbCr 颜色空间中, Y 代表亮度,Cb 和 Cr 分别代表蓝色分量和红色分量,两者合称为色彩分量。YCbCr 颜色空间具有将色度与亮度分离的特点,在 YCbCr 色彩空间中,肤色的聚类特性比较好,而且是二维独立分布,能够比较好地限制肤色的分布区域,并且受人种的影响不大。对比 RGB 颜色空间和 YCbCr 颜色空间,当光强发生变化时,RGB 颜色空间中 (R, G, B) 会同时发生变化,而 YCbCr 颜色空间中受光强相对独立,色彩分量受光强度影响不大,因此 YCbCr 颜色空间更适合用于肤色识别。

由于肤色在 YCbCr 空间受亮度信息的影响较小,本算法直接考虑 YCbCr 空间的 CbCr 分量,映射为二维独立分布的 CbCr 空间。在 CbCr 空间下,肤色类聚性好,利用人工阈值法将肤色与非肤色区域分开,形成二值图像。

RGB 转 YCbCr 的公式为:

$$Y = 0.257*R + 0.564*G + 0.098*B + 16$$

$$Cb = -0.148*R - 0.291*G + 0.439*B + 128$$

$$Cr = 0.439*R - 0.368*G - 0.071*B + 128$$

对肤色进行判定的条件常使用如下判定条件:

$$Cb > 77 \quad \&& \quad Cb < 127$$

$$Cr > 133 \quad \&& \quad Cr < 173$$

4.2 检测算法实现

4.2.1 工程创建

Step1: 打开 HLS, 按照之前介绍的方法, 创建一个新的工程, 命名为 Skin_Dection。

Step2: 右单击 Source 选项, 选择 New File, 创建一个名为 Top.cpp 的文件。



Step3: 在打开的编辑区中，把下面的程序拷贝进去：

```
#include "top.h"
#include <string.h>

void hls::hls_skin_dection(RGB_IMAGE& src, RGB_IMAGE& dst, int rows,
int cols,
    int y_lower, int y_upper, int cb_lower, int cb_upper, int
cr_lower, int cr_upper)
{

    LOOP_ROWS:for(int row = 0; row < rows ; row++)
    {

        LOOP_COLS:for(int col = 0; col < cols; col++)
        {

            //变量定义
            RGB_PIXEL src_data;
            RGB_PIXEL pix;
            RGB_PIXEL dst_data;
            bool skin_region;

            if(row < rows && col < cols) {
                src >> src_data;
            }

            //获取RGB通道数据
            uchar B = src_data.val[0];
            uchar G = src_data.val[1];
            uchar R = src_data.val[2];

            //RGB-->YCbCr颜色空间转换
            uchar y = (76 * R + 150 * G + 29 * B) >> 8;
            uchar cb = ((128*B -43*R - 85*G)>>8) + 128 ;
            uchar cr = ((128*R -107*G - 21 * B)>>8)+ 128 ;
        }
    }
}
```

```
//肤色区域判定
    if (y > y_lower && y < y_upper && cb > cb_lower && cb < cb_upper
&& cr > cr_lower && cr < cr_upper)
        skin_region = 1;
    else
        skin_region = 0;

    uchar temp0= (skin_region == 1)? (uchar)255: B;
    uchar temp1= (skin_region == 1)? (uchar)255: G;
    uchar temp2= (skin_region == 1)? (uchar)255: R;

    dst_data.val[0] = temp0;
    dst_data.val[1] = temp1;
    dst_data.val[2] = temp2;

    dst << dst_data;
}
}

void ImgProcess_Top(AXI_STREAM& input, AXI_STREAM& output, int rows, int cols,
                     int y_lower, int y_upper, int cb_lower, int cb_upper, int
                     cr_lower, int cr_upper)
{
    RGB_IMAGE img_0(rows, cols);
    RGB_IMAGE img_1(rows, cols);

    #pragma HLS dataflow
    hls::AXIvideo2Mat(input, img_0);
    hls::hls_skin_dection(img_0, img_1, rows, cols, y_lower, y_upper, cb_lower, cb_upper,
                          cr_lower, cr_upper);
    hls::Mat2AXIvideo(img_1, output);
}
```

Step4：在 Test Bench 中，用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码：

```
#include "top.h"
#include "hls_opencv.h"
#include "iostream"
#include <time.h>

using namespace std;
using namespace cv;

int main (int argc, char** argv)
{
    //IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* src = cvLoadImage("test.jpg");
    //IplImage* src = cvLoadImage("test_img1.jpg");
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth, src->nChannels);

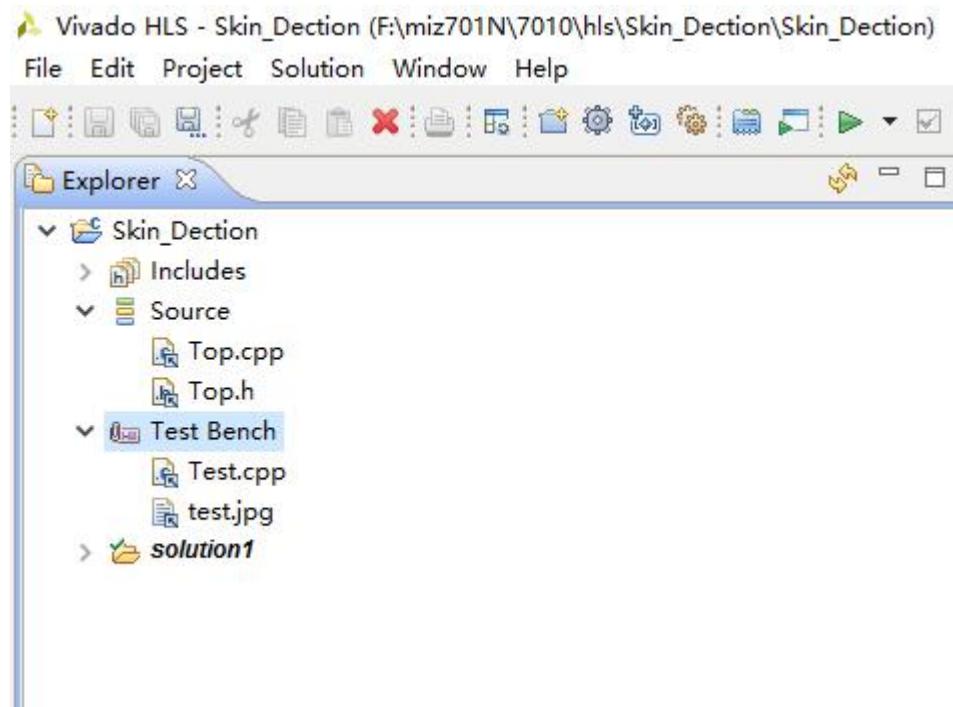
    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIvideo(src, src_axi);

    ImgProcess_Top(src_axi, dst_axi, src->height,
src->width, 0, 255, 75, 125, 131, 185);
    AXIvideo2IplImage(dst_axi, dst);

    cvShowImage("src", src);
    cvShowImage("dst_hls", dst);
    waitKey(0);

    return 0;
}
```

Step6：在 Test Bench 中添加一张名为 test.jpg 的测试图片，图片可以在我们提供的源程序中的 Image 文件夹中找到。完整的工程如下图所示：



4.2.2 代码综合

了解了肤色检测的原理以后我们进行算法实现可以发现，算法的关键就是进行 RGB 到 YCbCr 颜色空间的转换，其关键算法如下，大家可以发现其实算法只需要几十行就可以很容易的实现检测算法，不过需要大家注意的是因为我们使用 C++ 进行开发，那么为了程序的通用性以及方便团队合作开发，建议大家使用 namespace 命名空间，这样可以很好的解决重名问题，而且可以增加程序的可读性。

```
//声明hls命名空间
namespace hls
{
    void hls_skin_dectection(RGB_IMAGE& src, RGB_IMAGE& dst,int rows, int cols,
                                int y_lower,int y_upper,int cb_lower,int cb_upper,int cr_lower,int cr_upper);
}
```

```
void hls::hls_skin_dection(RGB_IMAGE& src, RGB_IMAGE& dst, int rows, int cols,
    int y_lower, int y_upper, int cb_lower, int cb_upper, int cr_lower, int cr_upper)
{
    LOOP_ROWS:for(int row = 0; row < rows ; row++)
    {
        #pragma HLS loop_flatten off
        LOOP_COLS:for(int col = 0; col < cols; col++)
        {
            #pragma HLS pipeline II=1
            //变量定义
            RGB_PIXEL src_data;
            RGB_PIXEL pix;
            RGB_PIXEL dst_data;
            bool skin_region;
            if(row < rows && col < cols) {
                src >> src_data;
            }

            //获取RGB通道数据
            uchar B = src_data.val[0];
            uchar G = src_data.val[1];
            uchar R = src_data.val[2];

            //RGB-->YCbCr颜色空间转换
            uchar Y = (76 * R + 150 * G + 29 * B) >> 8;
            uchar cb = ((128*B - 43*R - 85*G)>>8) + 128 ;
            uchar cr = ((128*R - 107*G - 21 * B)>>8)+ 128 ;

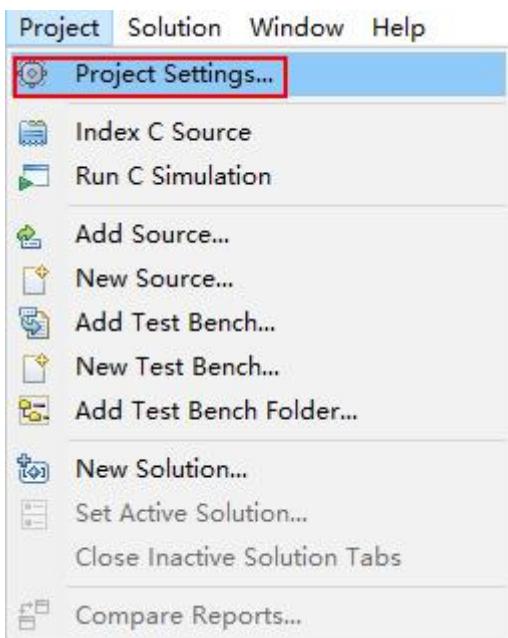
            //肤色区域判定
            if (y > y_lower && y < y_upper && cb > cb_lower && cb < cb_upper && cr > cr_lower && cr < cr_upper)
                skin_region = 1;
            else
                skin_region = 0;

            uchar temp0= (skin_region == 1)? (uchar)255: B;
            uchar temp1= (skin_region == 1)? (uchar)255: G;
            uchar temp2= (skin_region == 1)? (uchar)255: R;

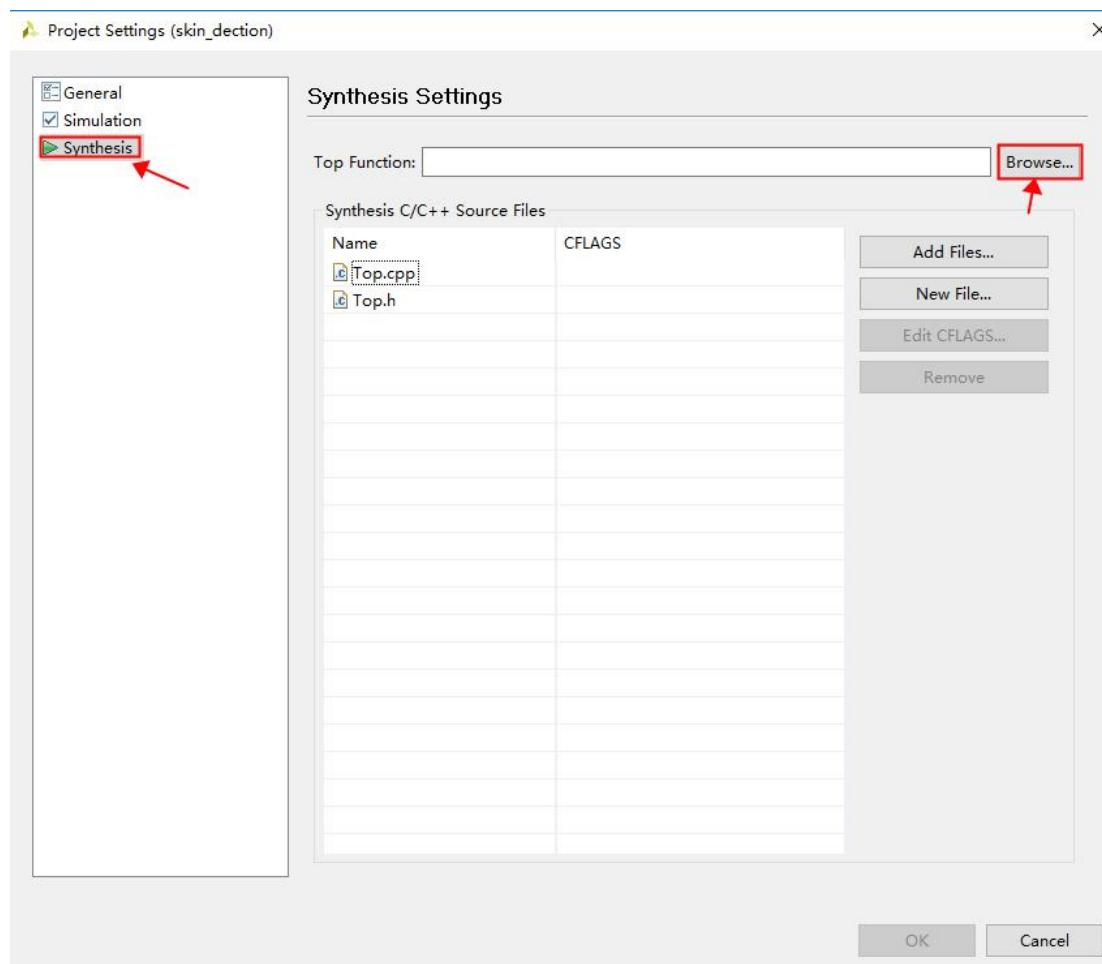
            dst_data.val[0] = temp0;
            dst_data.val[1] = temp1;
            dst_data.val[2] = temp2;
            dst << dst_data;
        }
    }
}
```

接下来，我们对项目进行代码综合。

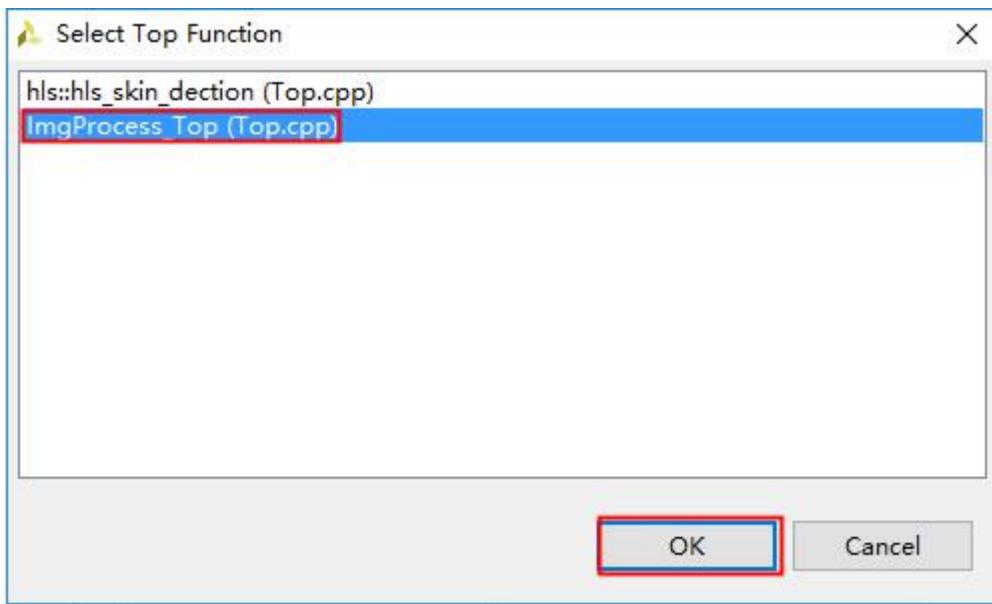
Step1：单击 Project—project settings 命令。



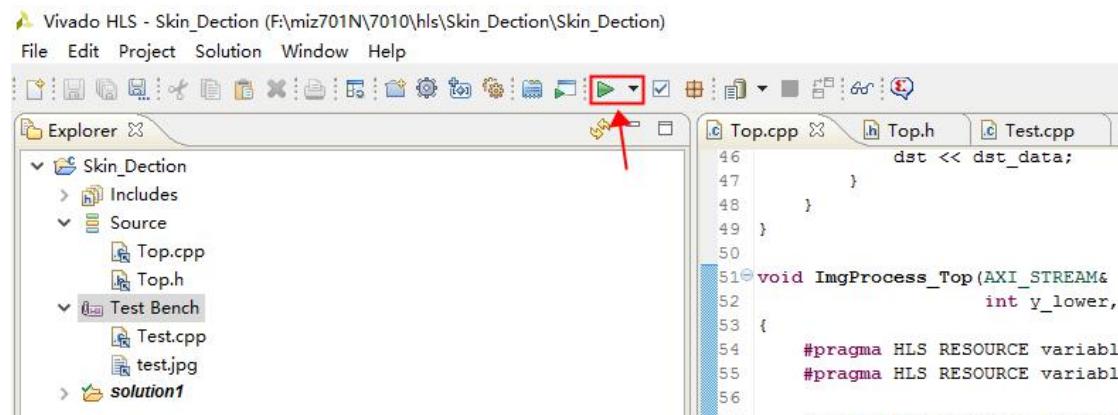
Step2: 在弹出的窗口中选择综合选项，然后在顶层函数设置区中单击右边的 Browse...按钮。



Step3: 选择 ImgProcess_Top 为顶层函数，然后单击两次 OK，完成工程的设置。



Step4：单击综合快捷按钮开始代码综合。



等待一段时间后，在未经代码优化及约束的条件下生成的综合报告如图：

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	0	-	90	568
Instance	-	7	687	552
Memory	-	-	-	-
Multiplexer	-	-	-	6
Register	-	-	11	-
Total	0	7	788	1127
Available	280	220	106400	53200
Utilization (%)	0	3	~0	2

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
ImgProcess_Top_AXIvideo2Mat_U0	ImgProcess_Top_AXIvideo2Mat	0	0	198	220
ImgProcess_Top_Block_Mat_exit45_proc36_U0	ImgProcess_Top_Block_Mat_exit45_proc36	0	0	26	25
ImgProcess_Top_Mat2AXIvideo_U0	ImgProcess_Top_Mat2AXIvideo	0	0	79	75
ImgProcess_Top_hls_skin_dection_U0	ImgProcess_Top_hls_skin_dection	0	7	384	232
Total		4	0	7	687

DSP48

Memory

FIFO

4.2.3 代码优化

根据官方手册 how_to_accelerate_opencv_applications_using_vivado_hls.pdf 中的描述：

➤ **Assign ‘input’ to be an AXI4 stream named “INPUT_STREAM”**

```
#pragma HLS RESOURCE variable=input core=AXIS metadata="-bus_bundle INPUT_STREAM"
```

➤ **Assign control interface to an AXI4-Lite interface**

```
#pragma HLS RESOURCE variable=return core=AXI_SLAVE metadata="-bus_bundle CONTROL_BUS"
```

➤ **Assign ‘rows’ to be accessible through the AXI4-Lite interface**

```
#pragma HLS RESOURCE variable=rows core=AXI_SLAVE metadata="-bus_bundle CONTROL_BUS"
```

➤ **Declare that ‘rows’ will not be changed during the execution of the function**

```
#pragma HLS INTERFACE ap_stable port=rows
```

➤ **Enable streaming dataflow optimizations**

```
#pragma HLS dataflow
```

我们对视频接口的约束如下：将 src 和 dst 指定为以“INPUT_STREAM”命名的 AXI4 Stream，将控制接口分配到 AXI4 Lite 接口，指定“rows”可通过 AXI4-Lite 接口进行访问并且声明在函数执行过程中“rows”不会改变，其实这几句优化语句中最关键的一条指令为启用数据流优化：

#pragma HLS dataflow , 它使得任务之间以流水线的方式执行，即
hls::AXIvideo2Mat(src, img_0);hls::skin_detect(img_0, img_1, rows, cols, cb_lower, cb_upper,
cr_lower, cr_upper);hls::Mat2AXIvideo(img_1, dst) ,这三个函数之间为流水线方式。

```
#pragma HLS RESOURCE variable=input core=AXIS metadata="-bus_bundle  
INPUT_STREAM"  
    #pragma HLS RESOURCE variable=output core=AXIS  
metadata="-bus_bundle OUTPUT_STREAM"  
  
    #pragma HLS RESOURCE core=AXI_SLAVE variable=rows  
metadata="-bus_bundle CONTROL_BUS"  
    #pragma HLS RESOURCE core=AXI_SLAVE variable=cols  
metadata="-bus_bundle CONTROL_BUS"  
    #pragma HLS RESOURCE core=AXI_SLAVE variable=y_lower  
metadata="-bus_bundle CONTROL_BUS"  
    #pragma HLS RESOURCE core=AXI_SLAVE variable=y_upper  
metadata="-bus_bundle CONTROL_BUS"  
    #pragma HLS RESOURCE core=AXI_SLAVE variable=cb_lower  
metadata="-bus_bundle CONTROL_BUS"  
    #pragma HLS RESOURCE core=AXI_SLAVE variable=cb_upper  
metadata="-bus_bundle CONTROL_BUS"  
    #pragma HLS RESOURCE core=AXI_SLAVE variable=cr_lower  
metadata="-bus_bundle CONTROL_BUS"  
    #pragma HLS RESOURCE core=AXI_SLAVE variable=cr_upper  
metadata="-bus_bundle CONTROL_BUS"  
    #pragma HLS RESOURCE core=AXI_SLAVE variable=return  
metadata="-bus_bundle CONTROL_BUS"  
  
    #pragma HLS INTERFACE ap_stable port=rows  
    #pragma HLS INTERFACE ap_stable port=cols  
    #pragma HLS INTERFACE ap_stable port=y_lower  
    #pragma HLS INTERFACE ap_stable port=y_upper  
    #pragma HLS INTERFACE ap_stable port=cb_lower  
    #pragma HLS INTERFACE ap_stable port=cb_upper  
    #pragma HLS INTERFACE ap_stable port=cr_lower  
    #pragma HLS INTERFACE ap_stable port=cr_upper  
  
RGB_IMAGE img_0(rows, cols);  
RGB_IMAGE img_1(rows, cols);  
  
#pragma HLS dataflow
```

```

55 void ImgProcess_Top(AXI_STREAM& input, AXI_STREAM& output,int rows, int cols,
56                         int y_lower,int y_upper,int cb_lower,int cb_upper,int cr_lower,int cr_upper)
57 {
58
59     #pragma HLS RESOURCE variable=input core=AXI metadata="-bus_bundle INPUT_STREAM"
60     #pragma HLS RESOURCE variable=output core=AXI metadata="-bus_bundle OUTPUT_STREAM"
61
62     #pragma HLS RESOURCE core=AXI_SLAVE variable=rows metadata="-bus_bundle CONTROL_BUS"
63     #pragma HLS RESOURCE core=AXI_SLAVE variable=cols metadata="-bus_bundle CONTROL_BUS"
64     #pragma HLS RESOURCE core=AXI_SLAVE variable=y_lower metadata="-bus_bundle CONTROL_BUS"
65     #pragma HLS RESOURCE core=AXI_SLAVE variable=y_upper metadata="-bus_bundle CONTROL_BUS"
66     #pragma HLS RESOURCE core=AXI_SLAVE variable=cb_lower metadata="-bus_bundle CONTROL_BUS"
67     #pragma HLS RESOURCE core=AXI_SLAVE variable=cb_upper metadata="-bus_bundle CONTROL_BUS"
68     #pragma HLS RESOURCE core=AXI_SLAVE variable=cr_lower metadata="-bus_bundle CONTROL_BUS"
69     #pragma HLS RESOURCE core=AXI_SLAVE variable=cr_upper metadata="-bus_bundle CONTROL_BUS"
70     #pragma HLS RESOURCE core=AXI_SLAVE variable=return metadata="-bus_bundle CONTROL_BUS"
71
72     #pragma HLS INTERFACE ap_stable port=rows
73     #pragma HLS INTERFACE ap_stable port=cols
74     #pragma HLS INTERFACE ap_stable port=y_lower
75     #pragma HLS INTERFACE ap_stable port=y_upper
76     #pragma HLS INTERFACE ap_stable port=cb_lower
77     #pragma HLS INTERFACE ap_stable port=cb_upper
78     #pragma HLS INTERFACE ap_stable port=cr_lower
79     #pragma HLS INTERFACE ap_stable port=cr_upper
80
81     RGB_IMAGE img_0(rows, cols);
82     RGB_IMAGE img_1(rows, cols);
83
84     #pragma HLS dataflow
85     hls::AXIVideo2Mat(input,img_0);
86     hls::hls_skin_dection(img_0,img_1,rows,cols,y_lower,y_upper,cb_lower,cb_upper,cr_lower,cr_upper);
87     hls::Mat2AXIVideo(img_1, output);
88 }
89

```

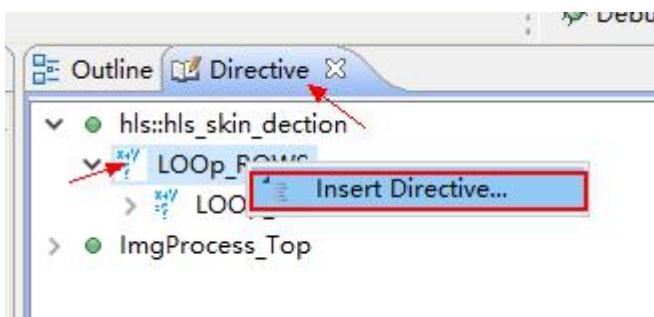
并且在肤色检测函数中添加如下的流水线：

```

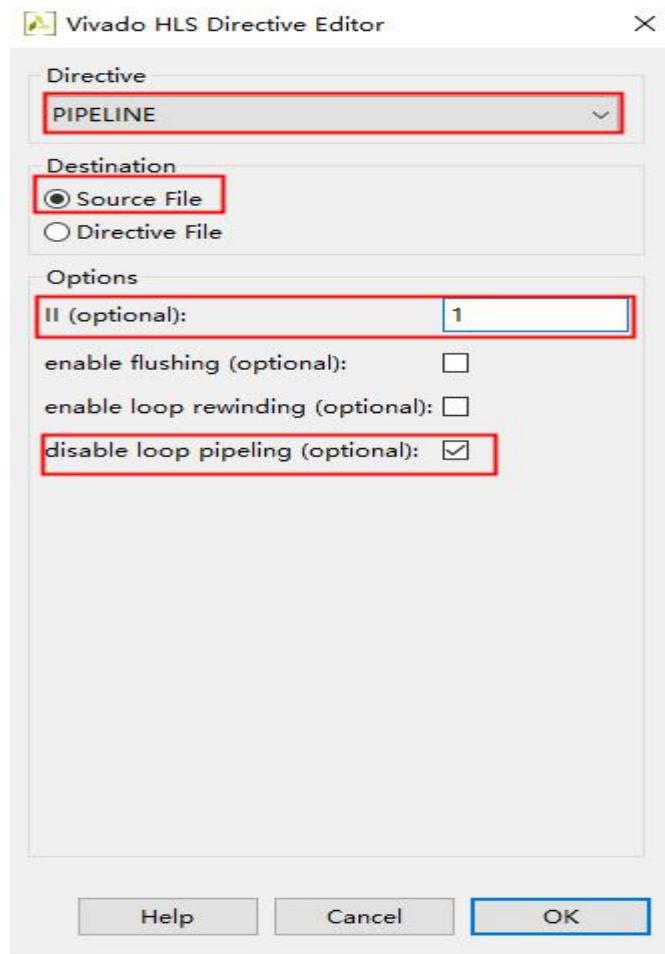
4 void hls::hls_skin_dection(RGB_IMAGE& src, RGB_IMAGE& dst, int rows, int cols,
5                           int y_lower,int y_upper,int cb_lower,int cb_upper,int cr_lower,int cr_upper)
6 {
7
8     LOOp_ROWS:for(int row = 0; row < rows ; row++)
9     {
10    #pragma HLS PIPELINE II=1 off
11
12        LOOp_COLS:for(int col = 0; col < cols; col++)
13        {
14
15            //变量定义
16            RGB_PIXEL src_data;
17            RGB_PIXEL pix;
18            RGB_PIXEL dst_data;
19            bool skin_region;
20
21            if(row < rows && col < cols) {
22                src >> src_data;
23            }
24

```

那么这条指令是如何添加的呢？在下图位置右键点击 Insert Directive...



配置如下即可生成#pragma HLS PIPELINE II=1 off 的优化指令，



进行优化以后我们再次进行综合，综合的结果如下，我们对比发现

□ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	0	-	30	120
Instance	-	7	360	466
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	4	-
Total	0	7	394	587
Available	280	220	106400	53200
Utilization (%)	0	3	~0	1

□ Detail

□ Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
ImgProcess_Top_AXIvideo2Mat_U0	ImgProcess_Top_AXIvideo2Mat	0	0	147	163
ImgProcess_Top_Mat2AXIvideo_U0	ImgProcess_Top_Mat2AXIvideo	0	0	43	71
ImgProcess_Top_hls_skin_dection_U0	ImgProcess_Top_hls_skin_dection	0	7	170	232
Total		3	0	7	360

+ DSP48

+ Memory

□ FIFO

Name	BRAM_18K	FF	LUT	Depth	Bits	Size:D*B
img_0_data_stream_0_V_U	0	5	20	1	8	8
img_0_data_stream_1_V_U	0	5	20	1	8	8
img_0_data_stream_2_V_U	0	5	20	1	8	8
img_1_data_stream_0_V_U	0	5	20	1	8	8
img_1_data_stream_1_V_U	0	5	20	1	8	8
img_1_data_stream_2_V_U	0	5	20	1	8	8
Total	0	30	120	6	48	48

我们可以发现使用了 7 个 DSP48E, 以及 394 个触发器(FF)和 587 个查找表(LUT), 并且在 Instance 里面可以看到例化了四个模块, 其实对应的就是下面红框内的三个函数,

```

104     RGB_IMAGE img_0(rows, cols);
105     RGB_IMAGE img_1(rows, cols);
106
107     hls::AXIvideo2Mat(src, img_0);
108     hls::skin_detect(img_0,img_1,rows,cols,cb_lower,cb_upper,cr_lower,cr_upper);
109     hls::Mat2AXIvideo(img_1, dst);
110 }
111

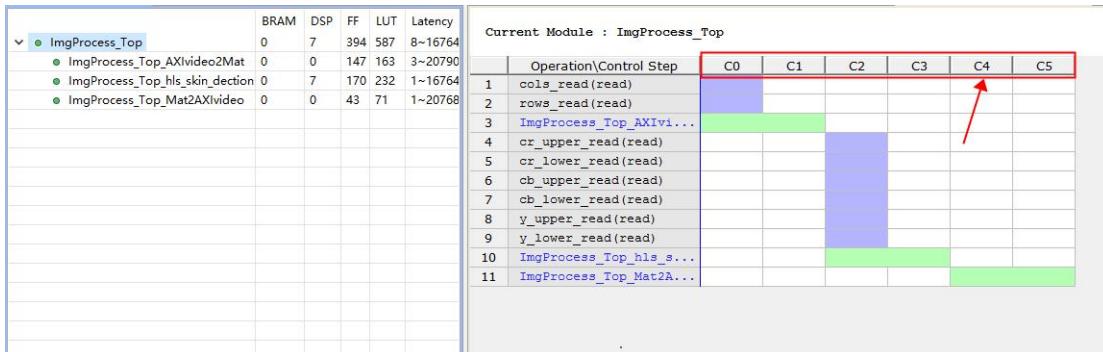
```

并且我们可以看到在 FIFO 中例化了 6 个不同名称的 FIFO, 那么对比名称我们能发现什么呢? 对了, 它就是声明中定义的 img_0 和 img_1 的数据的存储区, 那么为什么是 3 个 8bit 的呢? 因为对于 RGB_IMAGE 我们的定义如下, 其中 HLS_8UC3 即 3 通道的 8bit unsigned char 型数据, 因此, 综合生成了 3 个 8bit 位宽的 FIFO。

```
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> RGB_IMAGE;
```

单击 打开综合分析窗口, 我们再看一下我们的肤色检测的核心函数所消耗的时间,

将 LOOP 展开，我们可以发现函数消耗 6 个时钟周期才处理完毕。



查看代码我们可以发现我们使用了 for 循环 (LOOP)，下面对循环 LOOP 控制的优化指令进行说明：

Unrolling：展开循环，用于创建多个独立的操作，而不是对单个操作进行整合。

Merging：合并连续的循环，降低总延迟，提高共享和优化。

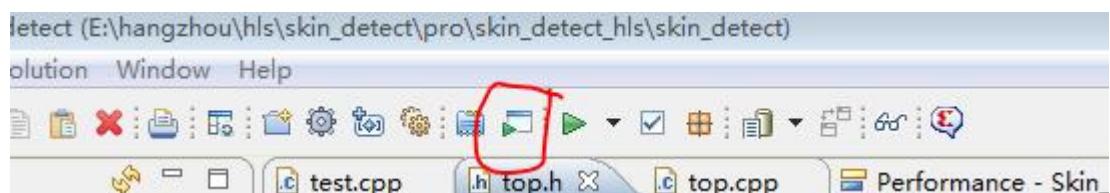
Flatting：允许将带有改善延迟和逻辑优化的嵌套循环，整理成一个单个的循环。

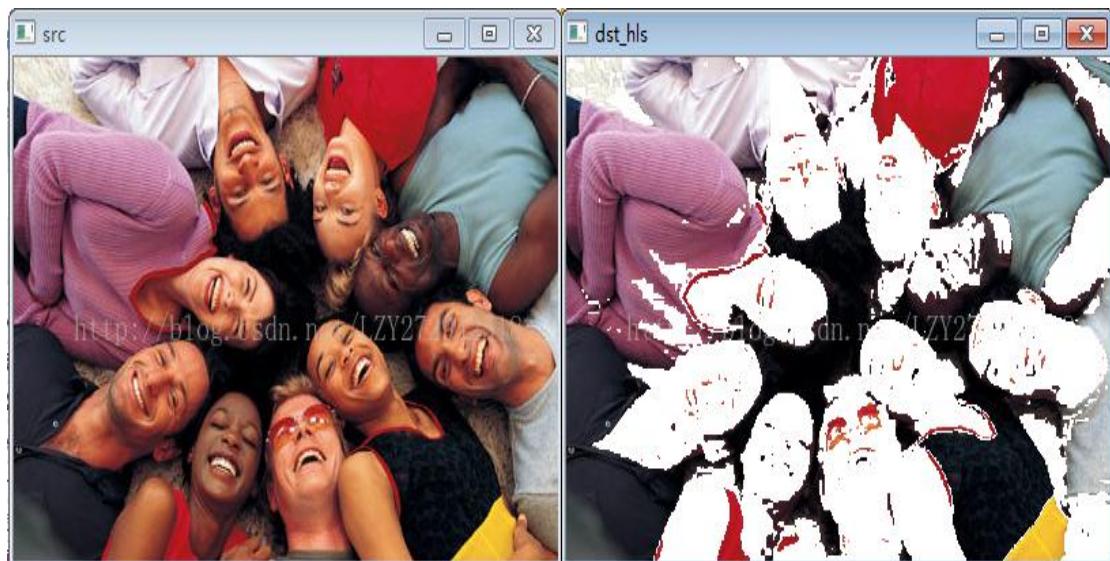
Dataflow：允许顺序循环，并发的操作。

Pipelining：通过执行并发的操作，提高吞吐量。

4.3 仿真测试

点击如下图方框圈出的快捷仿真按钮，我们对 Test.jpg 图片进行肤色检测，检测结果如下图所示。





通过对比发现基于颜色空间转换加阈值进行判断能在一定程度上进行检测，但是存在的检测误差较大的，在实际的项目应用中，需要对算法本身进行优化，那么在这里就不在进行介绍。

4.4 本章小结

本章节通过设计一个肤色检测的算法对前面几章的内容进行了巩固和验证。最后通过一组仿真验证了整个算法的有效性，虽然最后发现基于颜色空间转换加阈值进行肤色检测的结果又一点误差，但我们完全可以对算法进行优化，减小误差，本章重点介绍的是如何设计算法，因此对算法的优化就不再介绍，感兴趣的可以尝试优化一下算法。

S05_CH05_Sobel 算子硬件实现(一)_HLS 实现

5.1 Sobel 原理介绍

索贝尔算子 (Sobel operator) 主要用作边缘检测，在技术上，它是一离散性差分算子，用来运算图像亮度函数的灰度之近似值。在图像的任何一点使用此算子，将会产生对应的灰度矢量或是其法矢量

Sobel 卷积因子为：

-1	0	+1
-2	0	+2
-1	0	+1

G_x

+1	+2	+1
0	0	0
-1	-2	-1

G_y

该算子包含两组 3x3 的矩阵，分别为横向及纵向，将之与图像作平面卷积，即可分别得出横向及纵向的亮度差分近似值。如果以 A 代表原始图像， G_x 及 G_y 分别代表经横向及纵向边缘检测的图像灰度值，其公式如下：

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

具体计算如下：

$$\begin{aligned} G_x &= (-1)*f(x-1, y-1) + 0*f(x, y-1) + 1*f(x+1, y-1) \\ &\quad + (-2)*f(x-1, y) + 0*f(x, y) + 2*f(x+1, y) \\ &\quad + (-1)*f(x-1, y+1) + 0*f(x, y+1) + 1*f(x+1, y+1) \\ &= [f(x+1, y-1) + 2*f(x+1, y) + f(x+1, y+1)] - [f(x-1, y-1) + 2*f(x-1, y) + f(x-1, y+1)] \\ G_y &= 1*f(x-1, y-1) + 2*f(x, y-1) + 1*f(x+1, y-1) \\ &\quad + 0*f(x-1, y) + 0*f(x, y) + 0*f(x+1, y) \\ &\quad + (-1)*f(x-1, y+1) + (-2)*f(x, y+1) + (-1)*f(x+1, y+1) \end{aligned}$$

$$= [f(x-1, y-1) + 2f(x, y-1) + f(x+1, y-1)] - [f(x-1, y+1) + 2*f(x, y+1) + f(x+1, y+1)]$$

其中 $f(a, b)$, 表示图像 (a, b) 点的灰度值;

图像的每一个像素的横向及纵向灰度值通过以下公式结合, 来计算该点灰度的大小:

$$G = \sqrt{G_x^2 + G_y^2}$$

通常, 为了提高效率 使用不开平方的近似值:

$$|G| = |G_x| + |G_y|$$

如果梯度 G 大于某一阀值 则认为该点 (x, y) 为边缘点。

然后可用以下公式计算梯度方向:

$$\Theta = \arctan\left(\frac{G_y}{G_x}\right)$$

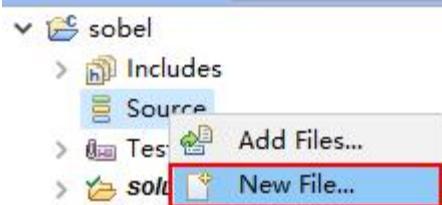
Sobel 算子根据像素点上下、左右邻点灰度加权差, 在边缘处达到极值这一现象检测边缘。对噪声具有平滑作用, 提供较为精确的边缘方向信息, 边缘定位精度不够高。当对精度要求不是很高时, 是一种较为常用的边缘检测方法。

5.2 Sobel 算子在 HLS 上的实现

5.2.1 工程创建

Step1: 打开 HLS, 按照之前介绍的方法, 创建一个新的工程, 命名为 sobel。

Step2: 右单击 Source 选项, 选择 New File, 创建一个名为 Top.cpp 的文件。



Step3: 在打开的编辑区中, 把下面的程序拷贝进去:

```
#include "top.h"

void hls_sobel(AXI_STREAM& INPUT_STREAM, AXI_STREAM&
OUTPUT_STREAM, int rows, int cols)
{
    //Create AXI streaming interfaces for the core
    #pragma HLS INTERFACE axis port=INPUT_STREAM
```

```

#pragma HLS INTERFACE axis port=OUTPUT_STREAM

#pragma HLS RESOURCE core=AXI_SLAVE variable=rows
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cols
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=return
metadata="-bus_bundle CONTROL_BUS"

#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols

RGB_IMAGE img_0(rows, cols);
RGB_IMAGE img_1(rows, cols);
RGB_IMAGE img_2(rows, cols);
RGB_IMAGE img_3(rows, cols);
RGB_IMAGE img_4(rows, cols);
RGB_IMAGE img_5(rows, cols);
RGB_PIXEL pix(50, 50, 50);

#pragma HLS dataflow
hls::AXIVideo2Mat(INPUT_STREAM, img_0);
hls::Sobel<1,0,3>(img_0, img_1);
hls::SubS(img_1, pix, img_2);
hls::Scale(img_2, img_3, 2, 0);
hls::Erode(img_3, img_4);
hls::Dilate(img_4, img_5);
hls::Mat2AXIVideo(img_5, OUTPUT_STREAM);
}

```

Step4: 再在 Source 中添加一个名为 Top.h 的库函数，并添加如下程序：

```

#ifndef _TOP_H_
#define _TOP_H_

#include "hls_video.h"

// maximum image size
#define MAX_WIDTH 1920
#define MAX_HEIGHT 1080

// I/O Image Settings
#define INPUT_IMAGE      "test_1080p.bmp"
#define OUTPUT_IMAGE     "result_1080p.bmp"

```

```

#define OUTPUT_IMAGE_GOLDEN "result_1080p_golden.bmp"

// typedef video library core structures
typedef hls::stream<ap_axiu<32,1,1,1> > AXI_STREAM;
typedef hls::Scalar<3, unsigned char> RGB_PIXEL;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> RGB_IMAGE;

// top level function for HW synthesis
void hls_sobel(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int rows,
int cols);

#endif

```

Step5: 在 Test Bench 中, 用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码:

```

#include "top.h"
#include "opencv_top.h"

using namespace std;
using namespace cv;

int main (int argc, char** argv)
{
    //获取图像数据
    IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels);

    //使用HLS库进行处理
    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIVideo(src, src_axi);
    hls_sobel(src_axi, dst_axi, src->height, src->width);
    AXIVideo2IplImage(dst_axi, dst);
    cvSaveImage(OUTPUT_IMAGE,dst);
    cvShowImage("hls_dst", dst);

    //使用OPENCV库进行处理
    opencv_image_filter(src, dst);
    cvShowImage("cv_dst", dst);
    cvSaveImage(OUTPUT_IMAGE_GOLDEN,dst);
    waitKey(0);

    //释放内存
    cvReleaseImage(&src);
}

```

```

    cvReleaseImage(&dst);
}

```

Step6：用同样的方法，再在 Test Bench 中创建一个 opencv_top.cpp 和 Opencv_top.h 文件，添加如下程序：

Opencv_top.cpp 代码如下：

```

#include "opencv_top.h"
#include "top.h"

void opencv_image_filter(IplImage* src, IplImage* dst)
{
    IplImage* tmp = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels);
    cvCopy(src, tmp);
    cv::Sobel((cv::Mat)tmp, (cv::Mat)dst, -1, 1, 0);
    cvSubS(dst, cvScalar(50,50,50), tmp);
    cvScale(tmp, dst, 2, 0);
    cvErode(dst, tmp);
    cvDilate(tmp, dst);
    cvReleaseImage(&tmp);
}

void sw_image_filter(IplImage* src, IplImage* dst)
{
    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIvideo(src, src_axi);
    hls_sobel(src_axi, dst_axi, src->height, src->width);
    AXIvideo2IplImage(dst_axi, dst);
}

```

Opencv_top.h 代码如下：

```

#ifndef __OPENCV_TOP_H__
#define __OPENCV_TOP_H__

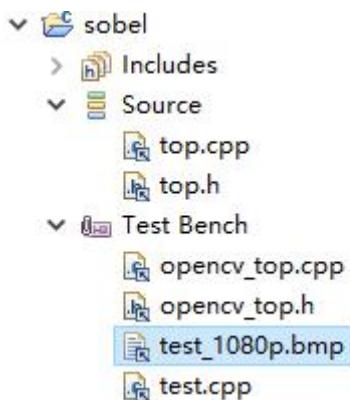
#include "hls_opencv.h"

void opencv_image_filter(IplImage* src, IplImage* dst);
void sw_image_filter(IplImage* src, IplImage* dst);

#endif

```

Step7：在 Test Bench 中添加一张名为 test_1080p.bmp 的测试图片，图片可以在我们提供的源程序中的 Image 文件夹中找到。完整的工程如下图所示：



5.2.2 代码优化及仿真

在前面几个章节中我们已经讲过代码优化的方法和策略，这里在工程源文件中提供已经优化好的代码文件，本章节的代码优化部分如下所示：

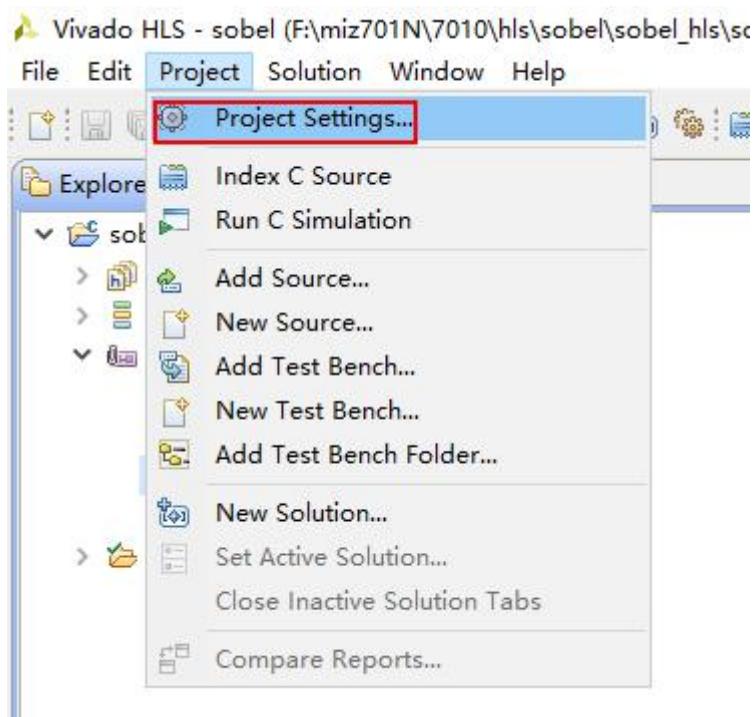
```
#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM

#pragma      HLS      RESOURCE      core=AXI_SLAVE      variable=rows
metadata="-bus_bundle CONTROL_BUS"
#pragma      HLS      RESOURCE      core=AXI_SLAVE      variable=cols
metadata="-bus_bundle CONTROL_BUS"
#pragma      HLS      RESOURCE      core=AXI_SLAVE      variable=return
metadata="-bus_bundle CONTROL_BUS"

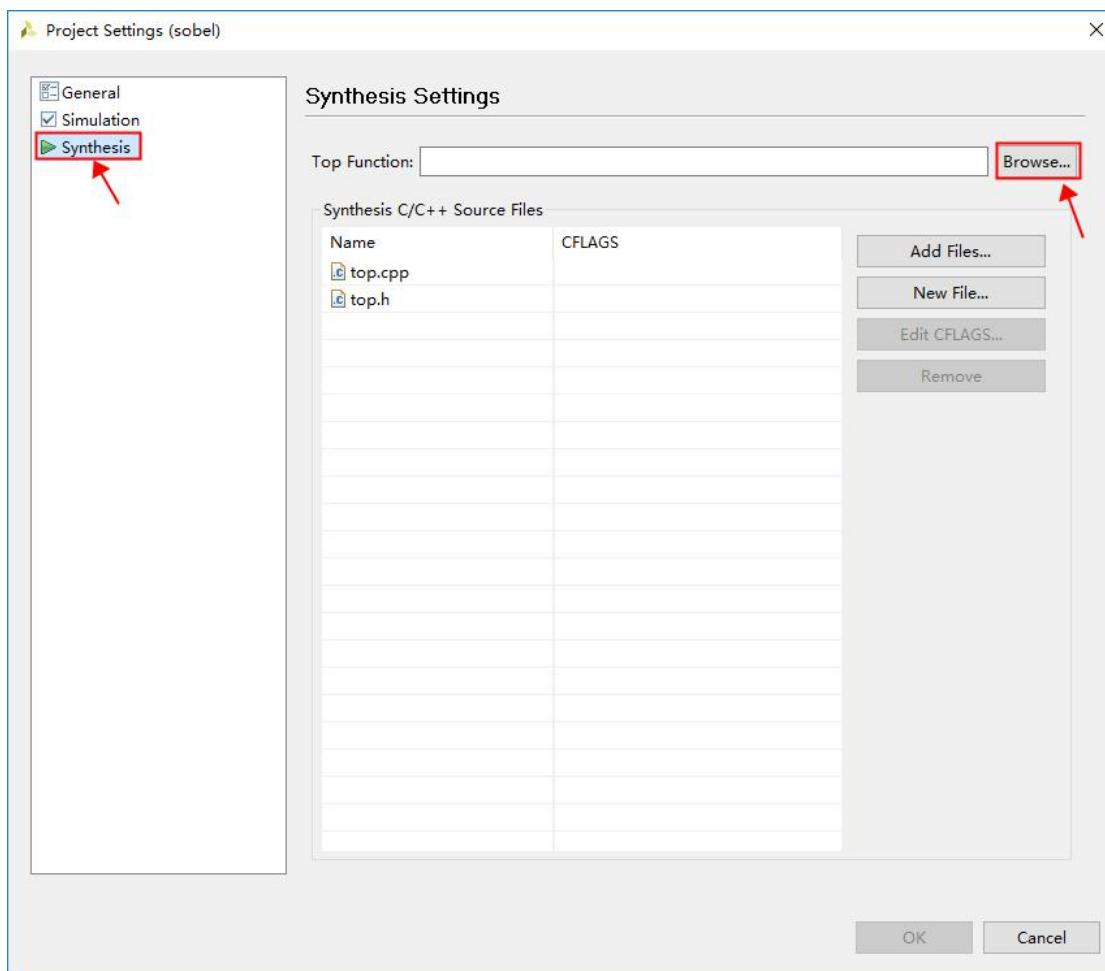
#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols
```

接下来对工程进行编译和仿真。

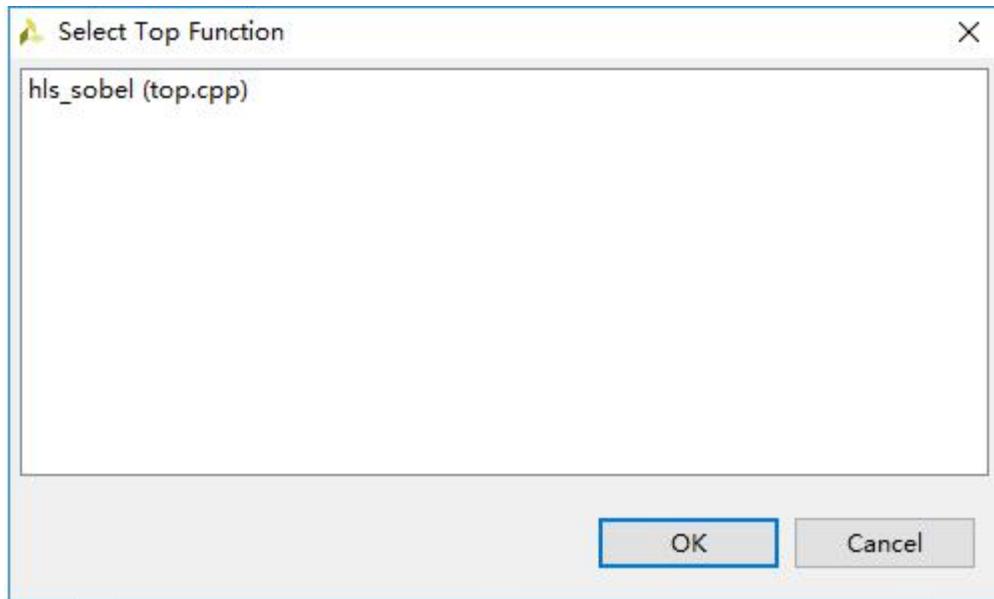
Step1：单击 Project→Project settings。



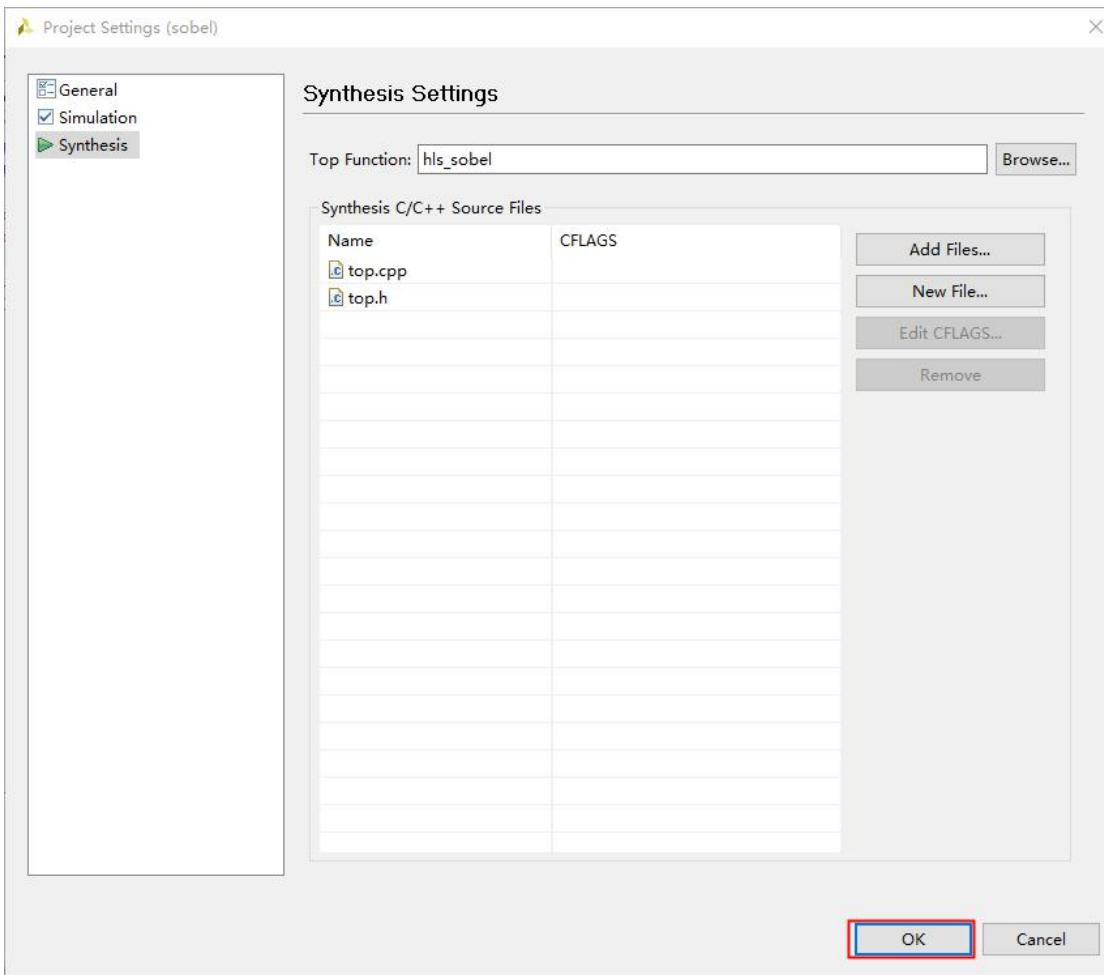
Step2: 选择 Synthesis 选项，然后点击 Browse.. 指定一个顶层函数。



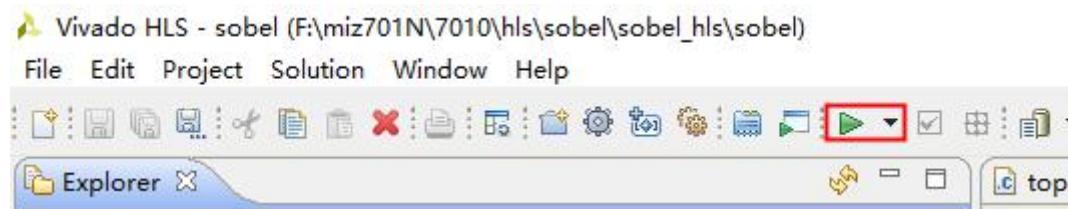
Step3: 选定 hls_sobel 为顶层函数，然后点击 O K。



Step4: 再次单击 OK，完成工程的修改。



Step5：单击  开始综合。



综合报告如下：

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.73	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
197	2088277	185	2088265	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

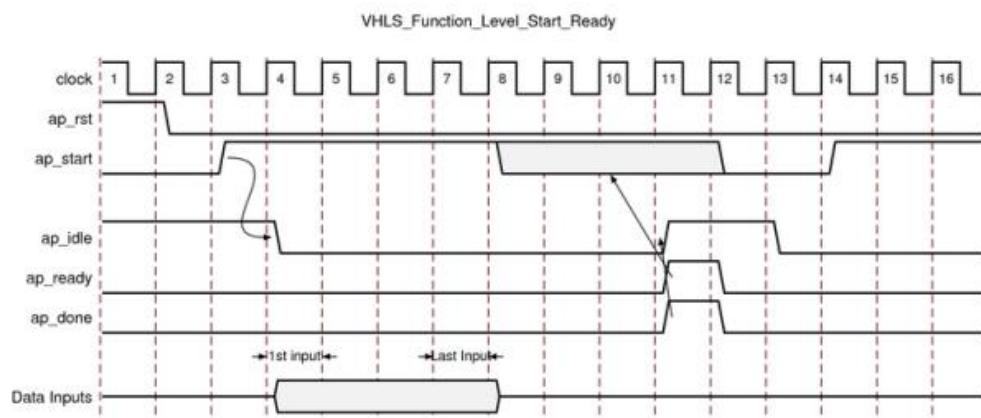
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	0	-	90	360
Instance	27	-	2099	3332
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	8	-
Total	27	0	2197	3693
Available	120	80	35200	17600
Utilization (%)	22	0	6	20

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
INPUT_STREAM_TDATA	in	32	axis	INPUT_STREAM_V_data_V	pointer
INPUT_STREAM_TKEEP	in	4	axis	INPUT_STREAM_V_keep_V	pointer
INPUT_STREAM_TSTRB	in	4	axis	INPUT_STREAM_V_strb_V	pointer
INPUT_STREAM_TUSER	in	1	axis	INPUT_STREAM_V_user_V	pointer
INPUT_STREAM_TLAST	in	1	axis	INPUT_STREAM_V_last_V	pointer
INPUT_STREAM_TID	in	1	axis	INPUT_STREAM_V_id_V	pointer
INPUT_STREAM_TDEST	in	1	axis	INPUT_STREAM_V_dest_V	pointer
INPUT_STREAM_TVALID	in	1	axis	INPUT_STREAM_V_dest_V	pointer
INPUT_STREAM_TREADY	out	1	axis	INPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TDATA	out	32	axis	OUTPUT_STREAM_V_data_V	pointer
OUTPUT_STREAM_TKEEP	out	4	axis	OUTPUT_STREAM_V_keep_V	pointer
OUTPUT_STREAM_TSTRB	out	4	axis	OUTPUT_STREAM_V_strb_V	pointer
OUTPUT_STREAM_TUSER	out	1	axis	OUTPUT_STREAM_V_user_V	pointer
OUTPUT_STREAM_TLAST	out	1	axis	OUTPUT_STREAM_V_last_V	pointer
OUTPUT_STREAM_TID	out	1	axis	OUTPUT_STREAM_V_id_V	pointer
OUTPUT_STREAM_TDEST	out	1	axis	OUTPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TVALID	out	1	axis	OUTPUT_STREAM_V_dest_V	pointer
OUTPUT_STREAM_TREADY	in	1	axis	OUTPUT_STREAM_V_dest_V	pointer
rows	in	32	ap_stable	rows	scalar
cols	in	32	ap_stable	cols	scalar
ap_clk	in	1	ap_ctrl_hs	hls_sobel	return value
ap_rst_n	in	1	ap_ctrl_hs	hls_sobel	return value
ap_start	in	1	ap_ctrl_hs	hls_sobel	return value
ap_done	out	1	ap_ctrl_hs	hls_sobel	return value
ap_idle	out	1	ap_ctrl_hs	hls_sobel	return value
ap_ready	out	1	ap_ctrl_hs	hls_sobel	return value

在协议类型里面我们可以看到我们主要使用了三种协议，分别是 axis、ap_stable 和 ap_ctrl_hs 三种，这些协议的详细解释我们均可以在官方手册 ug902 中找到，其中 ap_ctrl_hs 的时序操作如下图所示，说简单点就是复位完成等待 ap_start 信号开始进行操作。

ap_ctrl_hs

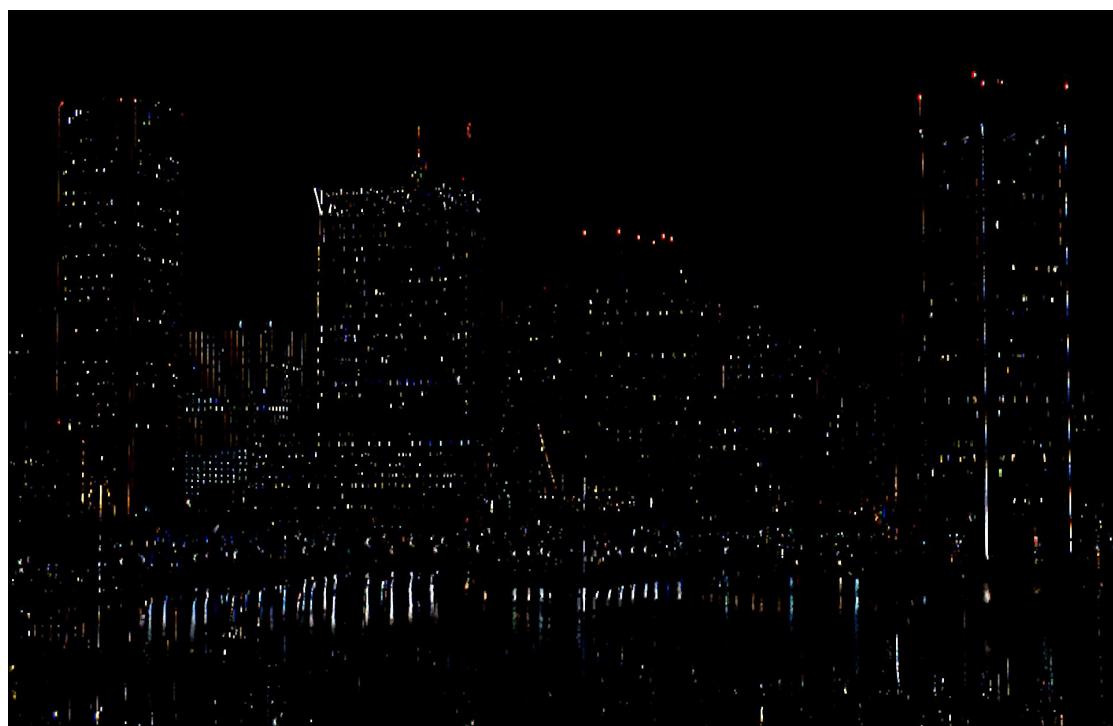
The behavior of the block level handshake signals created by interface mode ap_ctrl_hs are shown in the following figure and summarized below.



Step6：综合完毕，我们对代码进行仿真测试，单击 开始仿真。

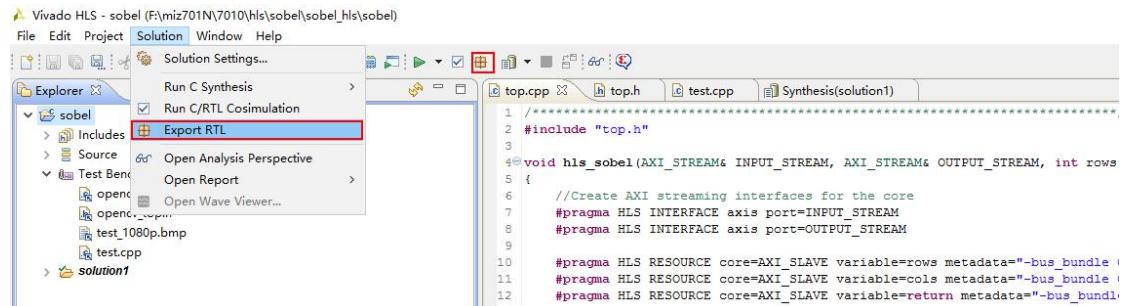


仿真结果如下，与通过 OPENCV 实现的 Sobel 检测结果基本一致。

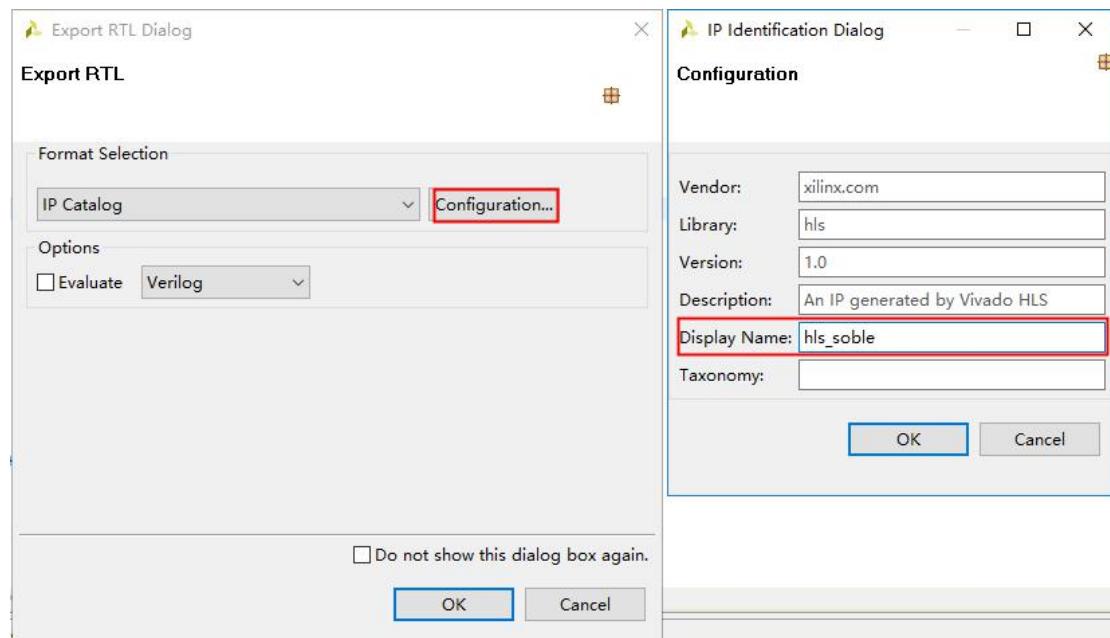


5.2.3 工程封装

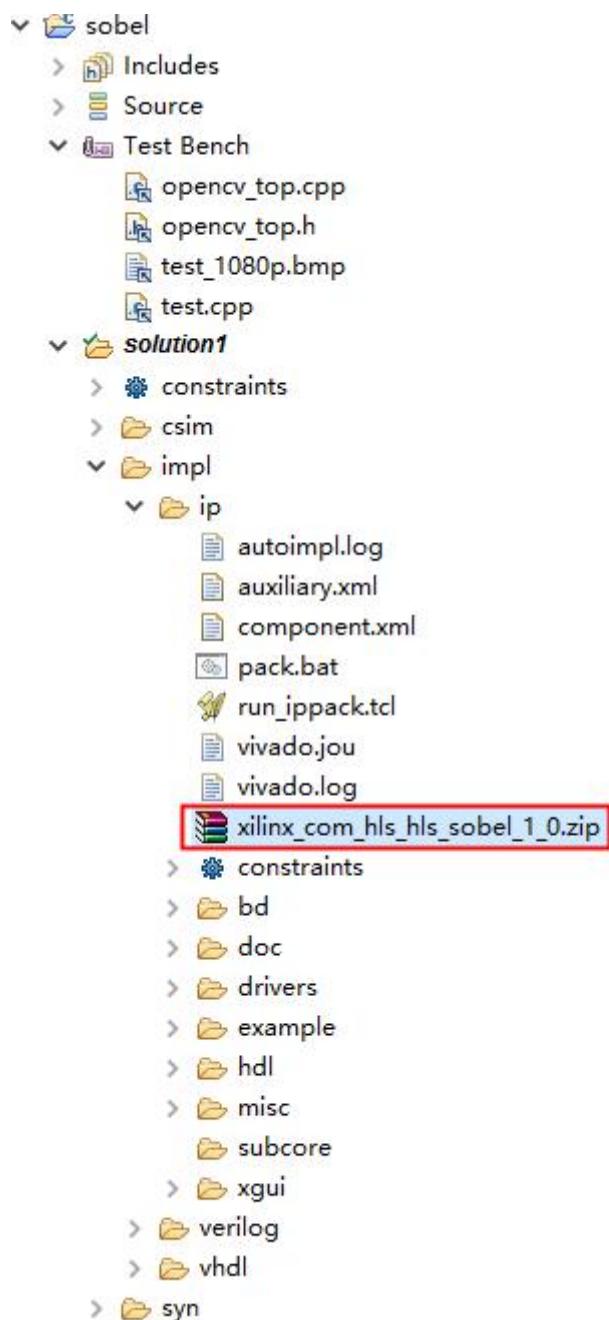
Step1：工程验证无误后，我们就可以开始进行代码封装了。单击 Solution-Export RTL 或直接单击 。



Step2：在弹出来的窗口中，点击 configuration，然后在弹出的窗口中做如下设置。



Step3：单击两次 OK，开始代码封装，最终在 Solution 里面生成了一个 impl 的文件夹，并且在 ip 里面多出了一个压缩包，这就是我们最终生成的 ip。



5.3 代码详解

在这一章我们通过调用 HLS 封装的视频处理库函数进行 Sobel 检测算子的实现，了解自带的处理函数的使用方法。在 HLS 中实现 Sobel 边缘检测的核心函数就下面几行，我们就对这几行代码进行介绍，使大家熟悉这几个函数的使用，为后续开发做好铺垫。

```

RGB_IMAGE img_0(rows, cols);
RGB_IMAGE img_1(rows, cols);
RGB_IMAGE img_2(rows, cols);
RGB_IMAGE img_3(rows, cols);
RGB_IMAGE img_4(rows, cols);
RGB_IMAGE img_5(rows, cols);
RGB_PIXEL pix(50, 50, 50);
```

```

#pragma HLS dataflow
hls::AXIvideo2Mat(INPUT_STREAM, img_0);
hls::Sobel<1,0,3>(img_0, img_1);
hls::SubS(img_1, pix, img_2);
hls::Scale(img_2, img_3, 2, 0);
hls::Erode(img_3, img_4);
hls::Dilate(img_4, img_5);
hls::Mat2AXIvideo(img_5, OUTPUT_STREAM);
```

我们首先用 hls 自带的视频处理库函数 hls::Sobel 进行 X 方向的边缘检测，为什么是 X 方向的？在官方给出的 ug902 技术手册中找到这个函数的介绍，如下图所示：

hls::Sobel

Synopsis

```

template<int XORDER, int YORDER, int SIZE, typename BORDERMODE, int SRC_T, int DST_T,
int ROWS,int COLS,int DROWS,int DCOLS>
void Sobel (
    Mat<ROWS, COLS, SRC_T>& _src,
    Mat<DROWS, DCOLS, DST_T>& _dst)
```



```

template<int XORDER, int YORDER, int SIZE, int SRC_T, int DST_T, int ROWS,int
COLS,int DROWS,int DCOLS>
void Sobel (
    Mat<ROWS, COLS, SRC_T>& _src,
    Mat<DROWS, DCOLS, DST_T>& _dst)
```

Parameters

Table 4-56: Parameters

Parameter	Description
<code>src</code>	Input image
<code>dst</code>	Output image

Description

- Computes a horizontal or vertical Sobel filter, returning an estimate of the horizontal or vertical derivative, using a filter such as:

[-1, 0, 1]

[-2, 0, 2]

[-1, 0, 1]

- SIZE=3, 5, or 7 is supported. This is the same as the equivalent OpenCV function.

- Only XORDER=1 and YORDER=0 (corresponding to horizontal derivative) or XORDER=0 and YORDER=1 (corresponding to a vertical derivative) are supported.

从上图可以得知，此函数的原函数使用了一个模板函数，在我们的程序中只使用了模板函数的前3个参数。Hls::sobel()函数中的两个参数src和dst分别是输入的图像和处理后的图像。

至于为什么是X方向的边缘检测，在图中方框圈出的地方给出了解释。方框中给出的意思为XORDER和YORDER控制是X方向还是Y方向检测。Hls::sobel()函数中这两个参数为1和0，是水平检测，也就是X方向的边缘检测。

模板函数中的第三个参数决定了窗口矩阵的大小，我们可以从图中看到可以是3种窗口，分别是3*3、5*5和7*7。所以整句函数的意思就是在3X3的窗口进行X方向的Sobel检测。当然在很多函数中我们能看到template<typename _T>的身影，这就是C++中模板的声明，建议大家在以后的代码编写中多使用模板，这样便于优化程序，关于这方面的知识自己查阅一下对应的资料，了解这块的开发过程。

进行边缘检测后我们使用hls::Subs进行数组的减法运算，函数原型如下：

```
template<int ROWS, int COLS, int SRC_T, typename _T, int DST_T>
void hls::SubRS (
    hls::Mat<ROWS, COLS, SRC_T>& src,
    hls::Scalar<HLS_MAT_CN(SRC_T), _T>& scl,
    hls::Mat<ROWS, COLS, DST_T>& dst);
```

Parameters

Table 4-59: Parameters

Parameter	Description
src	Input image
scl	Input scalar
dst	Output image
mask	Operation mask, an 8-bit single channel image that specifies elements of the dst image to be computed
dst_ref	Reference image that stores the elements for output image when mask(I) = 0

Description

- Computes the differences between elements of image src and scalar value scl.
- Saves the result in dst.

If computed with mask:

$$\text{dst}(I) = \begin{cases} \text{src}(I)-\text{scl} & \text{if } \text{mask}(I) \neq 0 \\ \text{dst_ref}(I) & \text{if } \text{mask}(I) = 0 \end{cases}$$

- Image data must be stored in src.
- If computed with mask, mask and dst_ref must have data stored.
- The image data of dst must be empty before invoking.
- Invoking this function consumes the data in src and fills the image data of dst.
- If computed with mask, the data of mask and dst_ref are also consumed.
- src and scl must have the same number of channels.
- dst and dst_ref must have the same size and number of channels as src

上图是官方的 hls 指导手册 ug902 对这个函数的描述，从圈出部分可以得知，这个函数是将输入的图像数据 src 与标称值 scl 进行比较，然后将得出的结果给 dst。也就是第二个方框中圈出的公式。在这个公式中可以注意到有一个 mask，这个值一般都不为 0。

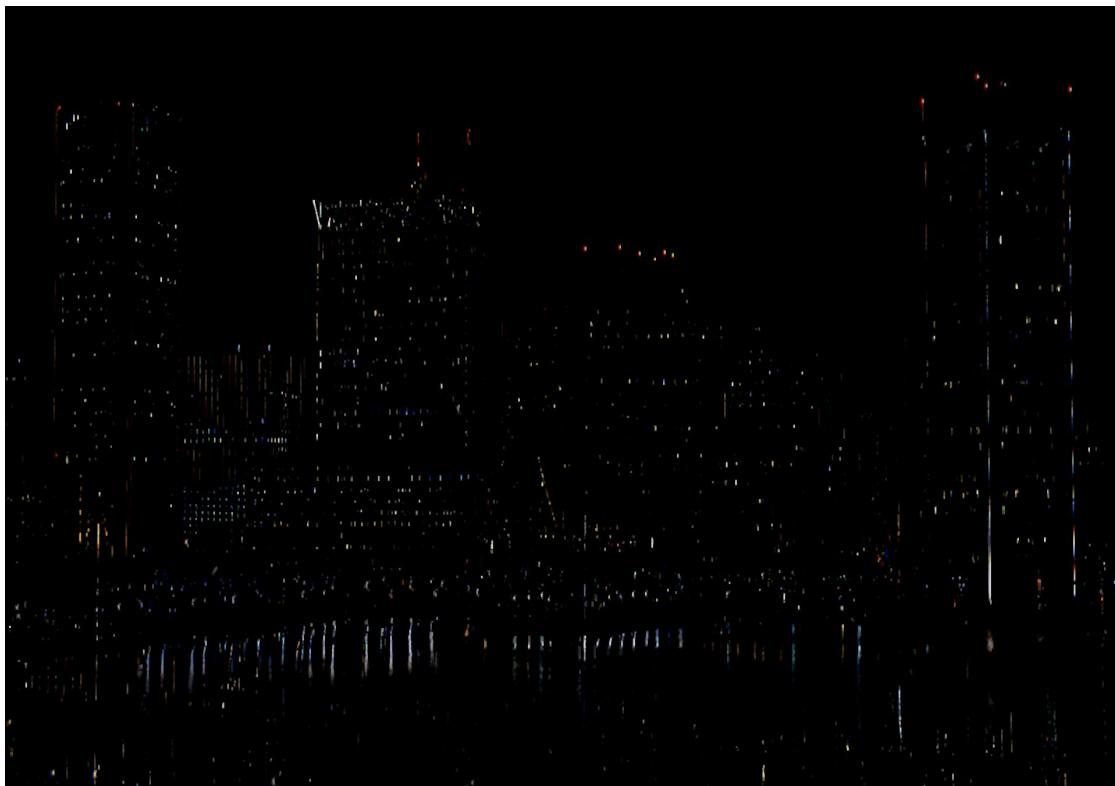
因为后面我们需要对图像进行腐蚀膨胀操作，为了避免图像失真，我们先通过 hls::Scalar() 这个函数对图像做线性变换，如果不转换的话，可以发现显示的图像显示偏色。

Description

- Converts an input image src with optional linear transformation.
- Saves the result as image dst.

$$\text{dst}(I) = \text{src}(I) * \text{scale} + \text{shift}$$

- Image data must be stored in src.
- The image data of dst must be empty before invoking.
- Invoking this function consumes the data in src and fills the image data of dst.
- src and dst must have the same size and number of channels. scale and shift must have the same data types.



失真图像

最后我们通过 `hls::Erode()` 和 `hls::Dilate()` 函数分别进行腐蚀和膨胀操作，腐蚀和膨胀是一对相反的操作，腐蚀是对局部求最小值的操作，那么膨胀就是求局部最大值的操作，在数学上解释就是对图像或部分区域对核进行卷积的一个运算。官方对这两个函数的描述及原型如下：

```
template<int ROWS, int COLS, int SRC_T, int DST_T>
void hls::Erode (
    hls::Mat<ROWS, COLS, SRC_T>& src,
    hls::Mat<ROWS, COLS, DST_T>& dst);
```

- Erodes the image `src` using the specified structuring element constructed within kernel.
- Saves the result in `dst`.
- The erosion determines the shape of a pixel neighborhood over which the maximum is taken, each channel of image `src` is processed independently:

$$\text{dst}(x, y) = \min_{(x', y') : \text{element}(x', y') \neq 0} \text{src}(x + x', y + y')$$

- Image data must be stored in `src`.
- The image data of `dst` must be empty before invoking.
- Invoking this function consumes the data in `src` and fills the image data of `dst`.
- `src` and `dst` must have the same size and number of channels.

```
template<int ROWS, int COLS, int SRC_T, int DST_T>
void hls::Dilate (
    hls::Mat<ROWS, COLS, SRC_T>& src,
    hls::Mat<ROWS, COLS, DST_T>& dst);
```

- Each channel of image `src` is processed independently.

$$\text{dst}(x, y) = \max_{(x', y') : \text{element}(x', y') \neq 0} \text{src}(x + x', y + y')$$

- Image data must be stored in `src`.
- The image data of `dst` must be empty before invoking.
- Invoking this function consumes the data in `src` and fills the image data of `dst`.
- `src` and `dst` must have the same size and number of channels.

5.3 本章小结

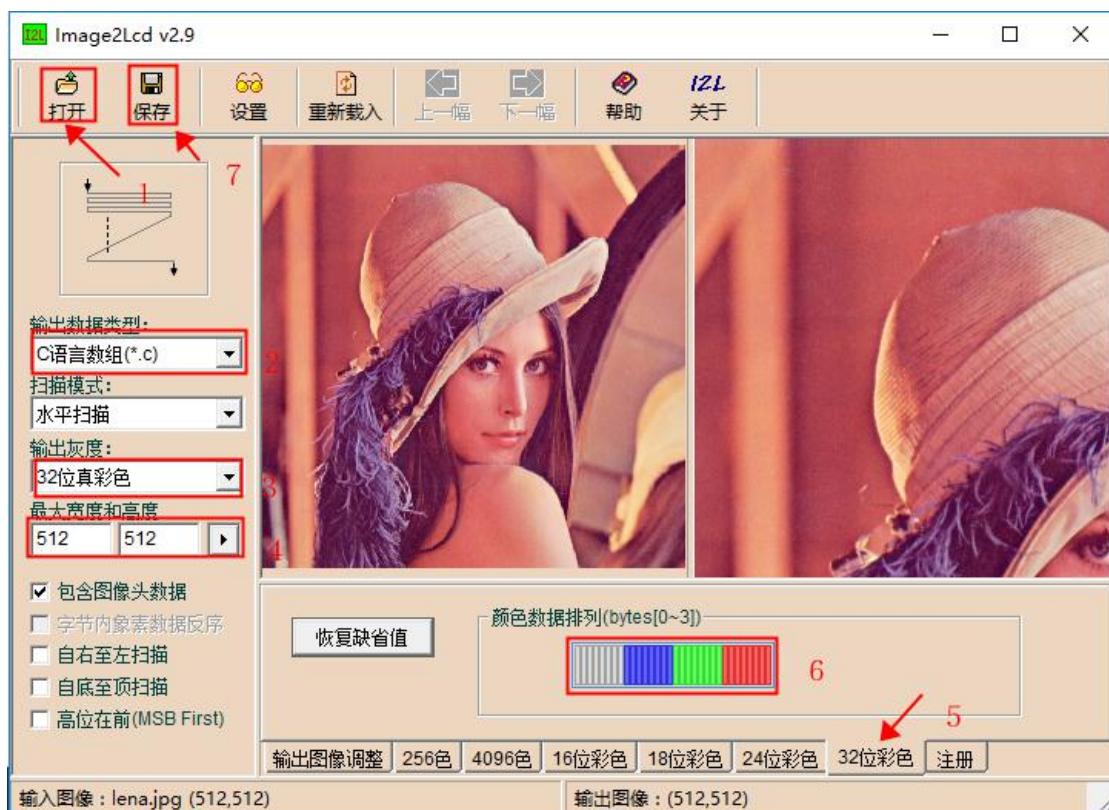
本章介绍了一种使用 HLS 实现 Sobel 检测的方法，最后通过软件封装得到了一个可以在 VIVADO 上使用的硬件 IP。下一章我们将使用这个 IP，对此 IP 进行功能的验证。

S05_CH06_Sobel 算子硬件实现(二)_硬件验证

6.1 系统硬件设计

这一章我们通过调用 HLS 生成的 Sobel 算子的 IP 搭建一个 Sobel 边缘检测的硬件实现平台，解决大家一直疑惑和关心的如何使用 HLS 生成的 IP。

在搭建硬件系统之前，我们先用 Image2LCD 对一幅图像进行取模，后期我们通过 SDK 来对取模的图像送 SOBEL 处理，然后将处理后的结果和源图像一起显示。首先，我们打开 Image2LCD 软件，如下图所示对 lena.jpg 进行取模（lena.jpg 可在上一章工程文件夹中的 image 文件夹中找到）：



- 1、打开一幅图像
- 2、选择 C 语言数组
- 3、选择 32 位真彩色
- 4、输入宽度和高度（输入完成后，记得一定要点击旁边的三角形按钮，再看底部的提示信息是否正确，如果点击之后仍然底部的输出图像不对，则超过了软件的取模范围，请裁切图片再次取模）。
- 5、选择 32 位彩色
- 6、拖动颜色块，将其如上图所示排列

7、保存取模输出的 C 语言数组

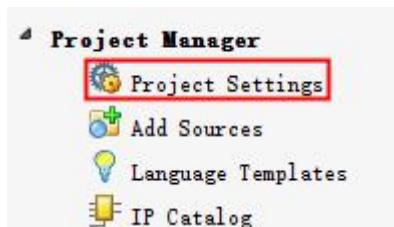
6.2 硬件工程创建

本章节的硬件工程可直接用第四季第一章 Linux 移植的图像显示系统进行修改，节省开发时间。

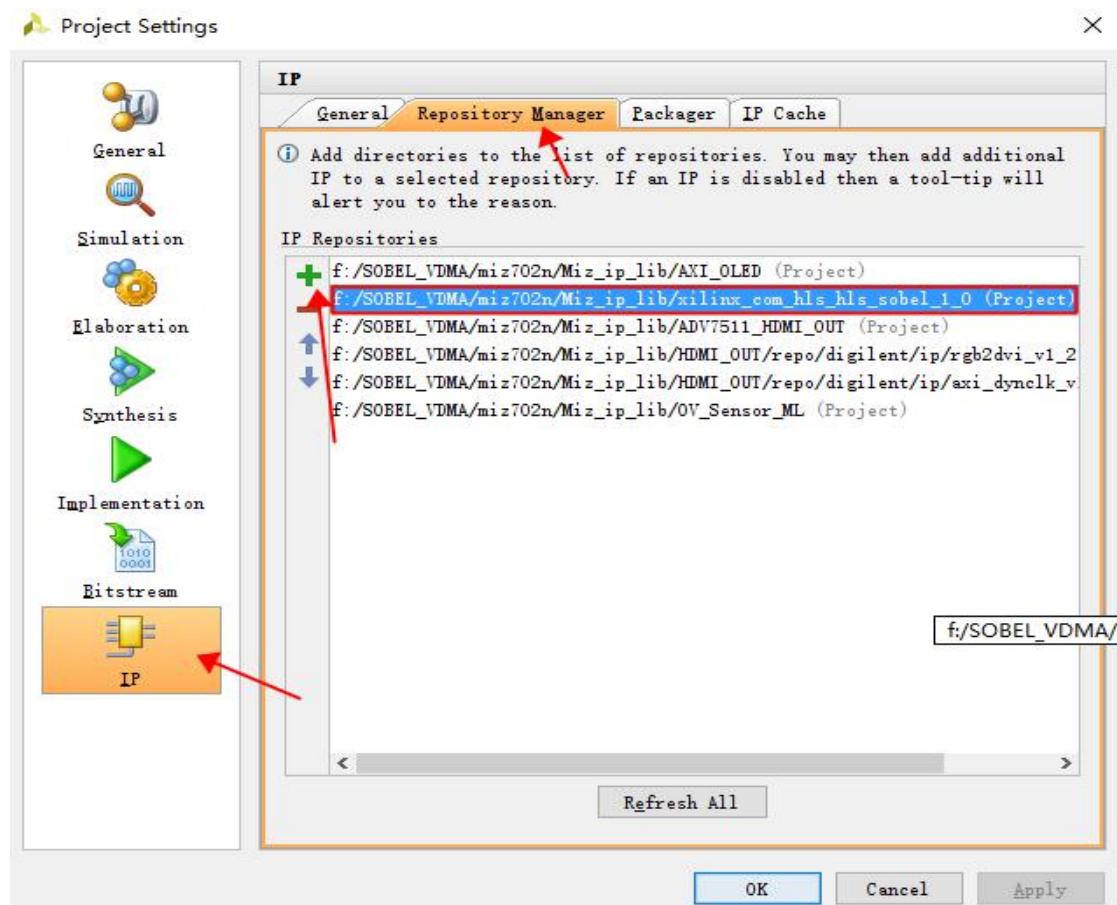
Step1：打开第四季第一章的工程（选择对应自己硬件的工程）。

Step2：双击 BD 文件对其进行一些修改。

Step3：在 Project Manager 设置区单击 Project settings。

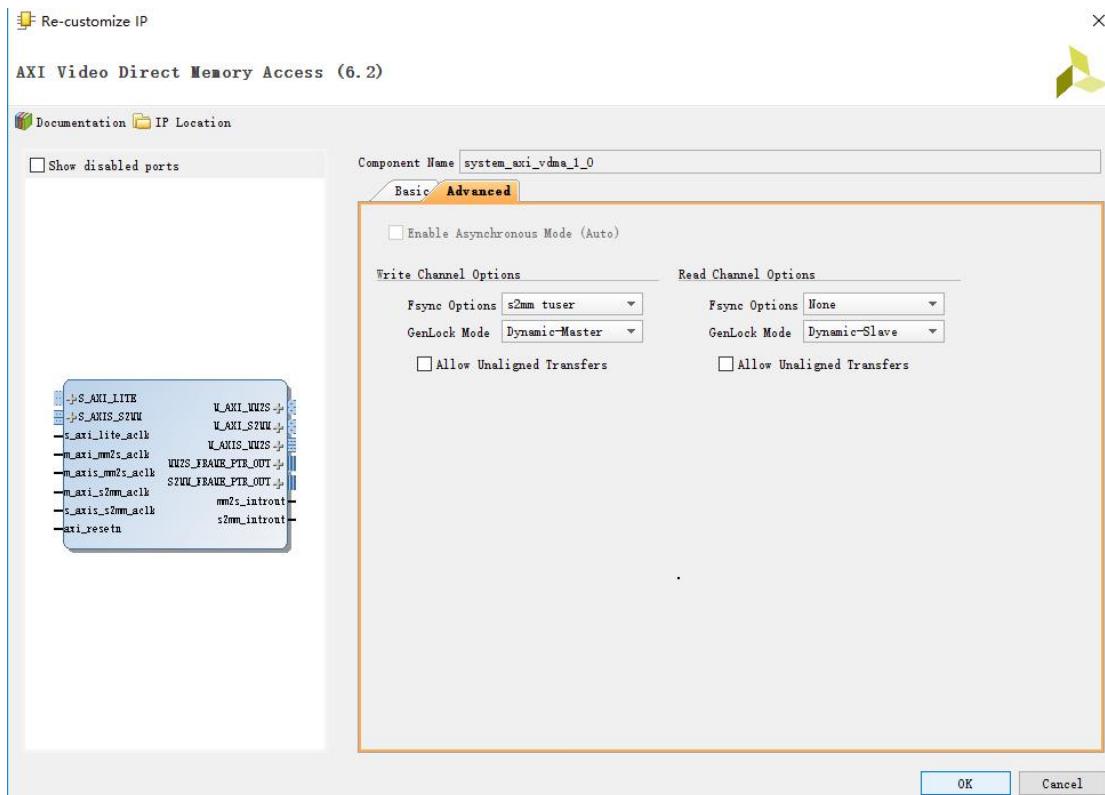
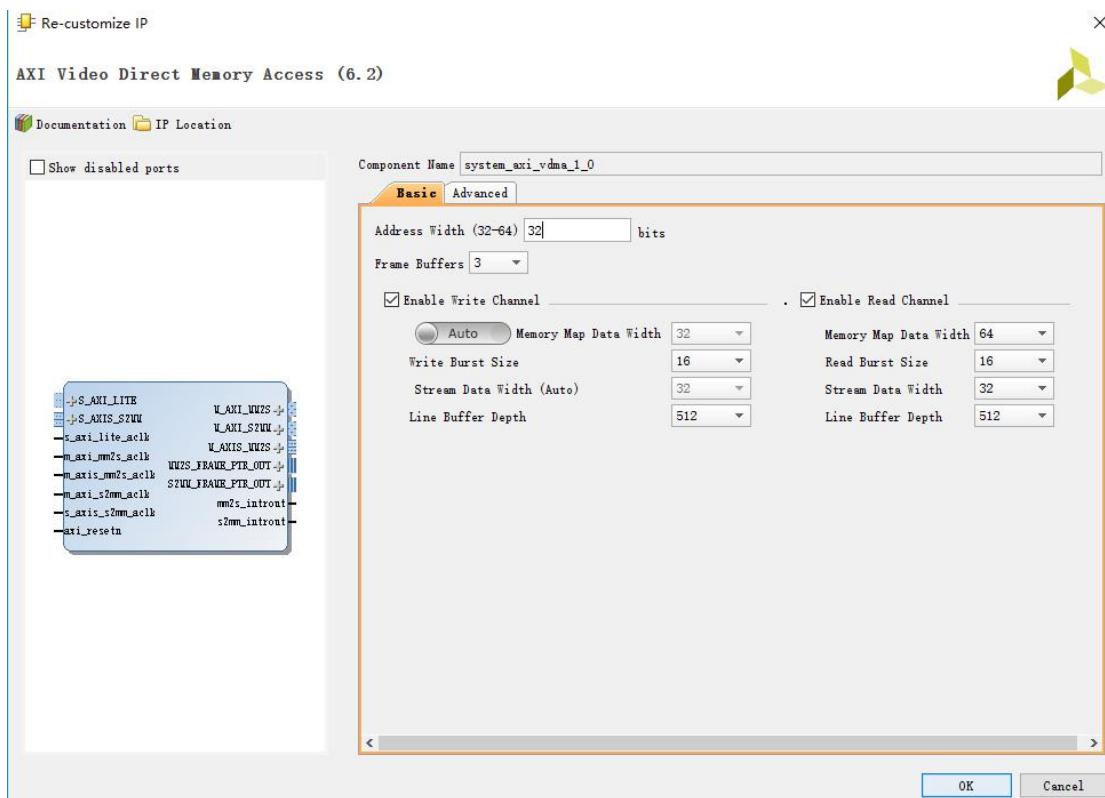


Step4：在弹出的窗口中，如下图设置：



单击图中的加号图标将我们上一章的 SOBEL 的 IP 添加到工程当中。

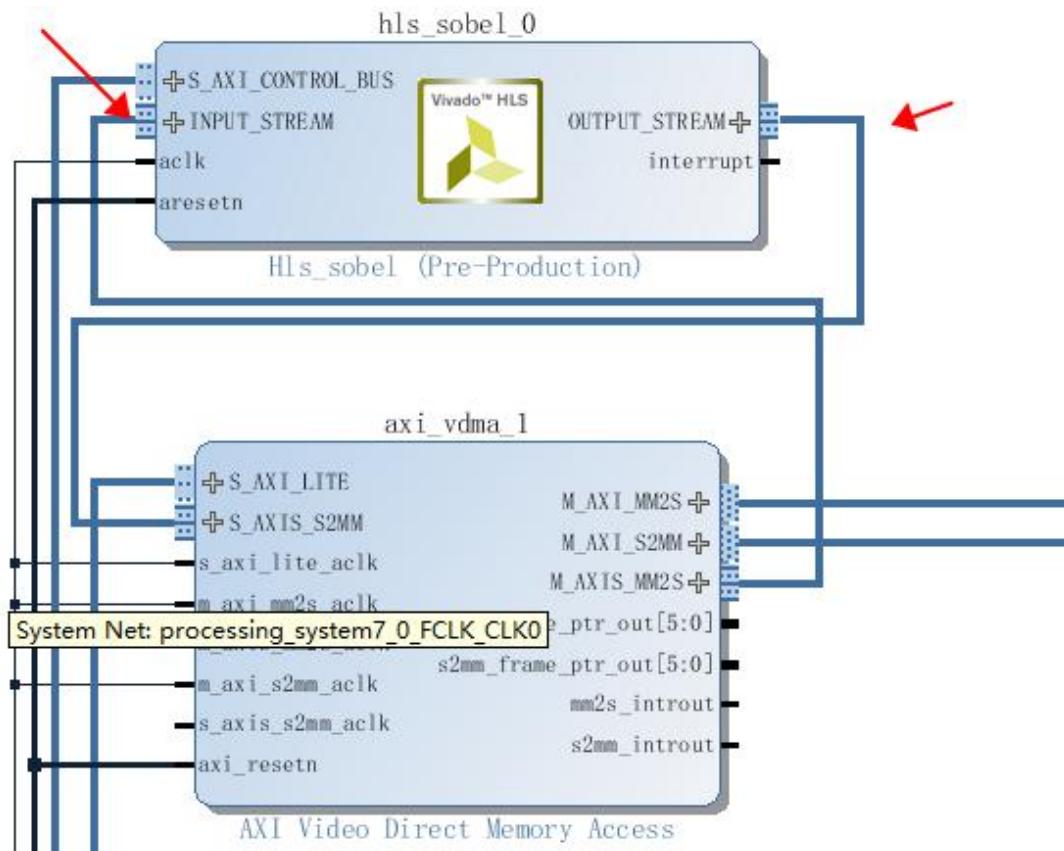
Step5：添加一个 VDMA，并如下图所示配置。



Step6：单击 Run connection Automation 进行自动布线。

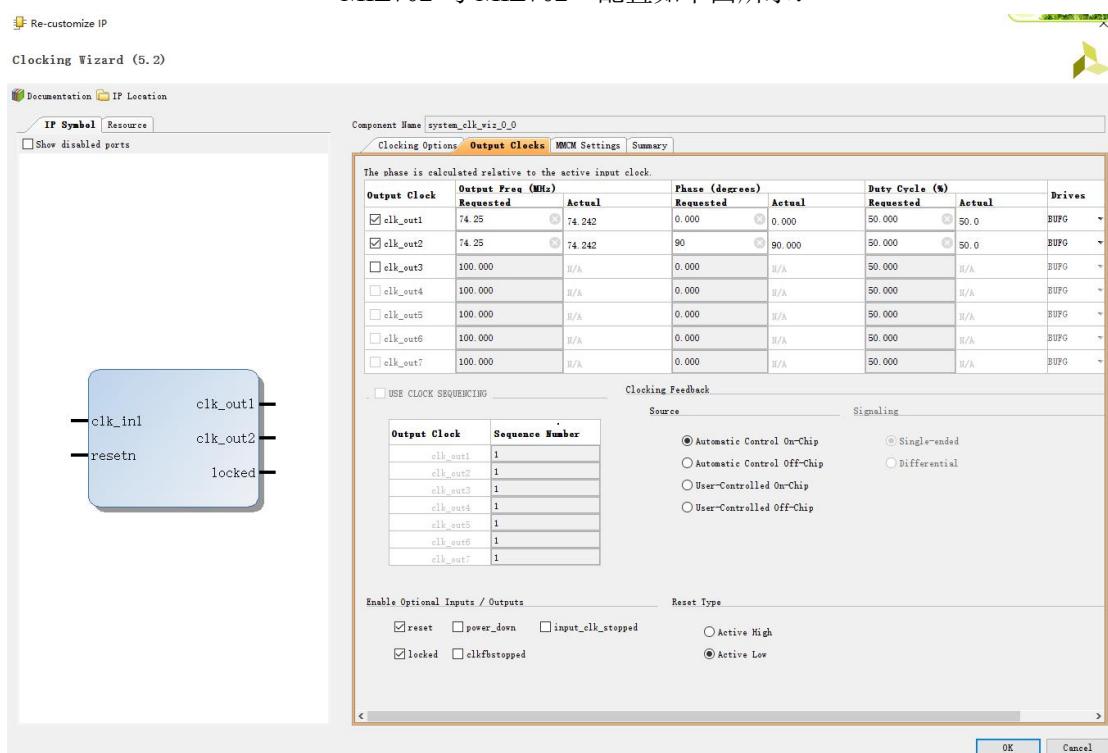
Step7：添加 SOBEL IP，然后单击 Run connection Automation。

Step8：按下图所示，完善连线。

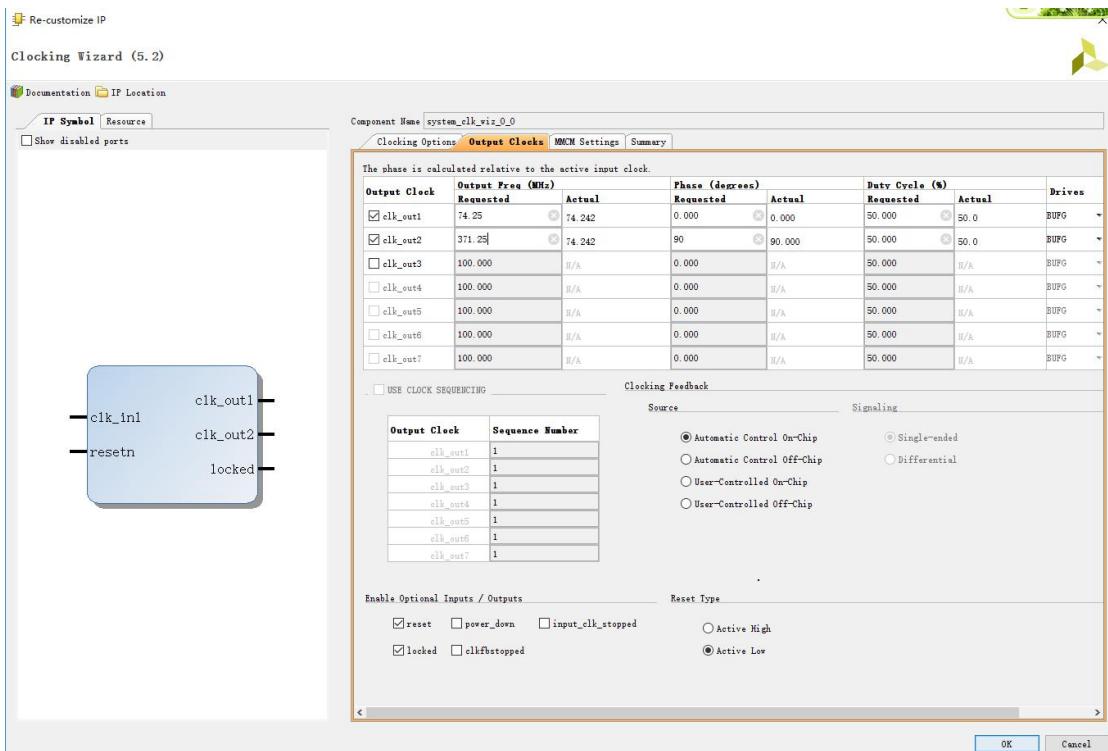


Step9: 双击 Clock wizard, 如下图配置 (注意由于 MIZ701N 是使用的 IO 模拟的 HDMI, 配置方式不一样)。

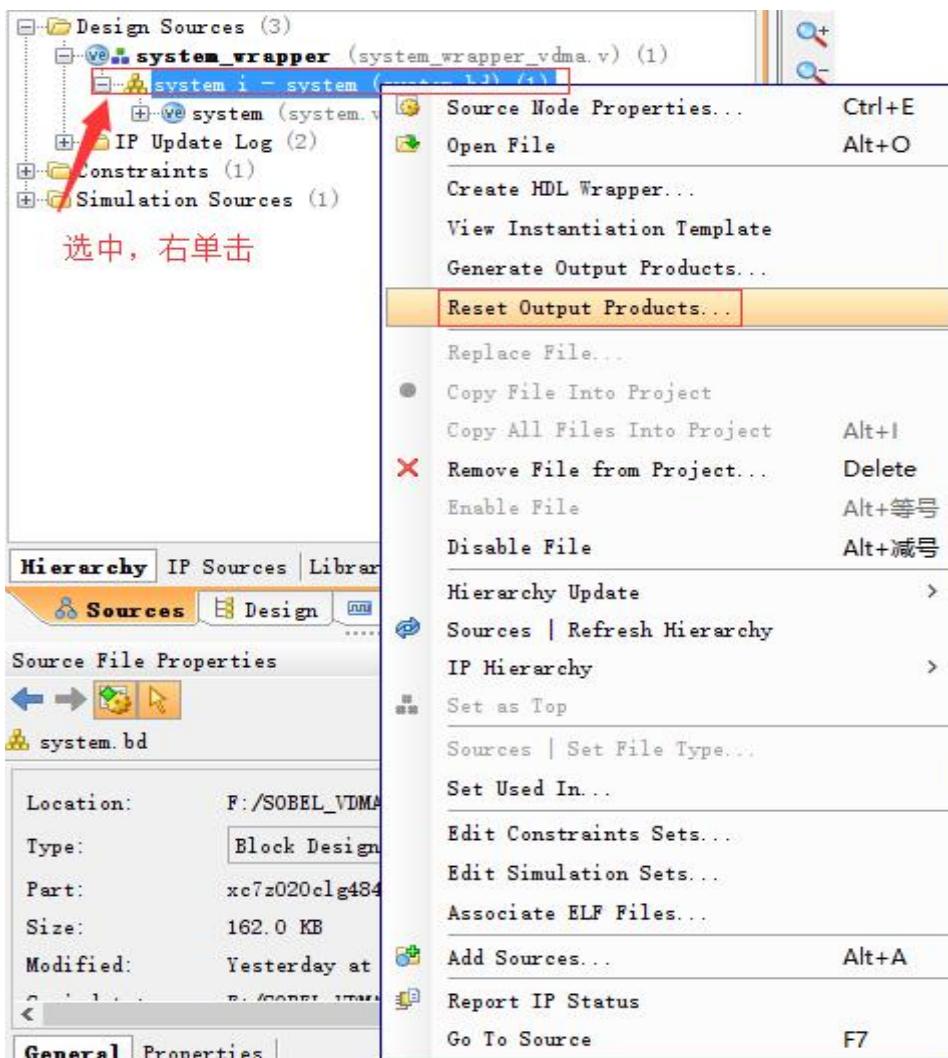
MIZ702 与 MIZ702 N 配置如下图所示:



MIZ701N 如下图所示:



Step10: 选中 BD 文件，然后右单击，选择 Reset Output Products...。



Step11: 再次选中 BD 文件，选择 Generate Output Products..。

Step12: 选中 BD 文件，然后选择 Create HDL Wrapper。

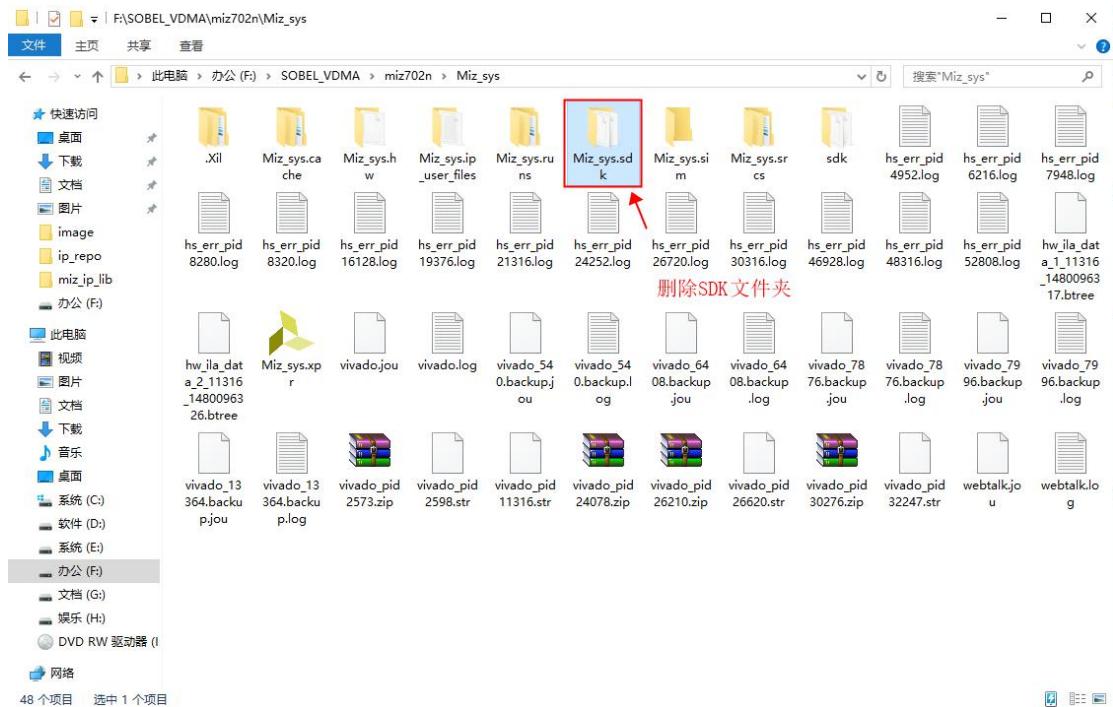
Step13: 单击 生成 Bit 文件。



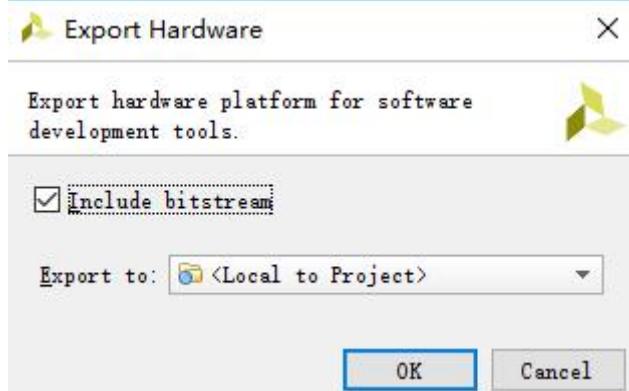
6.3 导入到 SDK

Bit 文件之后，接下来就是修改 SDK，添加相关的 SOBEL 的驱动。

Step1: 在工程文件夹中删除原来的工程的文件夹，这是因为如果不删除原来的 SDK 工程的话，系统就不会自动生成相关 HLS 生成的 IP 自带的驱动，笔者就曾经遇到过这种问题。



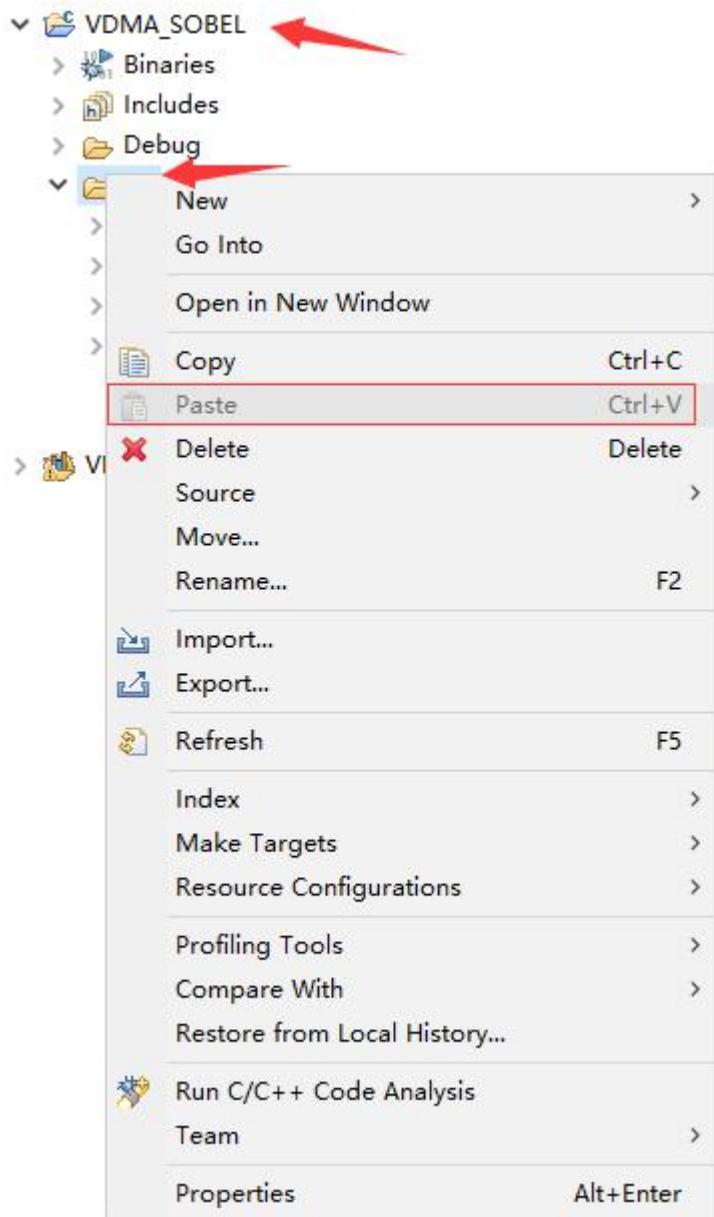
Step2: 单击 File-Export-Export Hardware..。



Step2: 单击 File-Launch SDK。

Step3: 新建一个名为 VDMA_SOBEL 的空的工程。

Step4: 复制第一节中生成的图片 C 数组, 然后在 SDK 中, 选中工程, 在 Src 下直接按 Ctrl +V 将其复制到工程中来。



Step5: 双击 main.c，用如下程序进行替换。

```
#include "xaxivdma.h"
#include "xaxivdma_i.h"
#include "xhls_sobel.h"
#include "sleep.h"

#define DDR_BASEADDR      0x00000000
#define DISPLAY_VDMA      XPAR_AXI_VDMA_0_BASEADDR + 0
#define SOBEL_VDMA        XPAR_AXI_VDMA_1_BASEADDR + 0
#define DIS_X              1280
#define DIS_Y              720
#define SOBEL_ROW          512
#define SOBEL_COL          512
```

```
#define pi          3.14159265358
#define COUNTS_PER_SECOND (XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ)/64

#define SOBEL_S2MM    0x08000000
#define SOBEL_MM2S    0x0A000000
#define DISPLAY_MM2S  0x0C000000

u32 *BufferPtr[3];
static XHls_sobel sobel;

//函数声明
void Xil_DCacheFlush(void);
// 所有数据格式 为 RGBA, 低位的透明度暂不起作用
extern const unsigned char gImage_lena[1048584];

void SOBEL_VDMA_setting(unsigned int width,unsigned int height,unsigned
int s2mm_addr,unsigned int mm2s_addr)
{
    //S2MM
    Xil_Out32(SOBEL_VDMA + 0x30, 0x4); //reset S2MM VDMA Control
Register
    usleep(10);
    Xil_Out32(SOBEL_VDMA + 0x30, 0x0); //genlock
    Xil_Out32(SOBEL_VDMA + 0xAC, s2mm_addr); //S2MM Start Addresses
    Xil_Out32(SOBEL_VDMA + 0xAC+4, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xAC+8, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xA4, width*4); //S2MM Horizontal Size
    Xil_Out32(SOBEL_VDMA + 0xA8, width*4); //S2MM Frame Delay and
Stride
    Xil_Out32(SOBEL_VDMA + 0x30, 0x3); //S2MM VDMA Control Register
    Xil_Out32(SOBEL_VDMA + 0xA0, height); //S2MM Vertical Size start
an S2M
    // Xil_DCacheFlush();

    //MM2S
    Xil_Out32(SOBEL_VDMA + 0x00,0x00000003); // enable circular mode
    Xil_Out32(SOBEL_VDMA + 0x5c,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x60,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x64,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x58,(width*4)); // h offset
    Xil_Out32(SOBEL_VDMA + 0x54,(width*4)); // h size
    Xil_Out32(SOBEL_VDMA + 0x50,height); // v size
    //Xil_DCacheFlush();
```

```
}

void DISPLAY_VDMA_setting(unsigned int width,unsigned height,unsigned int mm2s_addr)
{
    Xil_Out32((DISPLAY_VDMA + 0x000), 0x00000003); // enable
    circular mode
    Xil_Out32((DISPLAY_VDMA + 0x05c), mm2s_addr); // start address
    Xil_Out32((DISPLAY_VDMA + 0x060), mm2s_addr); // start address
    Xil_Out32((DISPLAY_VDMA + 0x064), mm2s_addr); // start address
    Xil_Out32((DISPLAY_VDMA + 0x058), (width*4)); // h offset
    (640 * 4) bytes
    Xil_Out32((DISPLAY_VDMA + 0x054), (width*4)); // h size
    (640 * 4) bytes
    Xil_Out32((DISPLAY_VDMA + 0x050), height); // v size
    (480)
}

void SOBEL_DDRWR(unsigned int addr,unsigned int cols,unsigned int lows)
{
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    for(i=0;i<cols;i++)
    {
        for(j=0;j<lows;j++)
        {
            b= gImage_lena[(j+i*cols)*4+1];
            g= gImage_lena[(j+i*cols)*4+2];
            r= gImage_lena[(j+i*cols)*4+3];

            Xil_Out32((addr+(j+i*cols)*4),((r<<24)|(g<<16)|(b<<8)|0x0));
        }
    }
    Xil_DCacheFlush();
}

void SOBEL_Setup()
{
    //const int cols = 512;
    //const int rows = 512;
    XHls_sobel_SetRows(&sobel, SOBEL_COL);
    XHls_sobel_SetCols(&sobel, SOBEL_ROW);
```

```
XHls_sobel_DisableAutoRestart(&sobel);
XHls_sobel_InterruptGlobalDisable(&sobel);
SOBEL_VDMA_setting(SOBEL_ROW, SOBEL_COL, SOBEL_S2MM, SOBEL_MM2S);
SOBEL_DDRWR(SOBEL_MM2S, SOBEL_ROW, SOBEL_COL);
XHls_sobel_Start(&sobel);
}

void show_img(u32 x, u32 y, u32 disp_base_addr, const unsigned char * addr,
u32 size_x, u32 size_y, u32 type)
{
    //计算图片 左上角坐标
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    u32 start_addr=disp_base_addr;
    start_addr = disp_base_addr + 4*x + y*4*DIS_X;
    for(j=0;j<size_y;j++)
    {
        for(i=0;i<size_x;i++)
        {
            if(type==0)
            {
                b = *(addr+(i+j*size_x)*4+2); //08
                g = *(addr+(i+j*size_x)*4+1); //60
                r = *(addr+(i+j*size_x)*4); //01
            }
            else
            {
                b = *(addr+(i+j*size_x)*4+1); //08
                g = *(addr+(i+j*size_x)*4+2); //60
                r = *(addr+(i+j*size_x)*4+3); //01
            }

            Xil_Out32((start_addr+(i+j*DIS_X)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
        }
    }
    Xil_DCacheFlush();
}

int main(void)
{
```

```

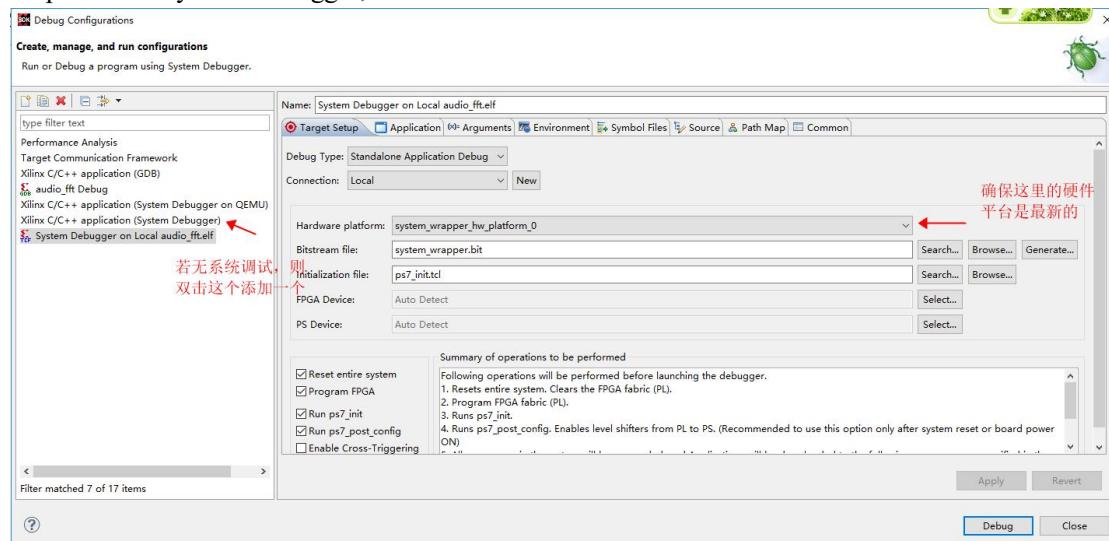
//Xil_DCacheFlush();
xil_printf("Starting the first VDMA \n\r");
int status = XHls_sobel_Initialize(&sobel,
XPAR_HLS_SOBEL_0_S_AXI_CONTROL_BUS_BASEADDR);
if(0 != status)
{
    xil_printf("XHls_Sobel_Initialize failed \n");
}
SOBEL_Setup();
DISPLAY_VDMA_setting(DIS_X,DIS_Y,DISPLAY_MM2S);

while(1)
{
    show_img(0,0,DISPLAY_MM2S,(void*)SOBEL_S2MM,512,512,0);
    show_img(522,0,DISPLAY_MM2S,(void*)SOBEL_MM2S,512,512,1);
}
return 0;
}

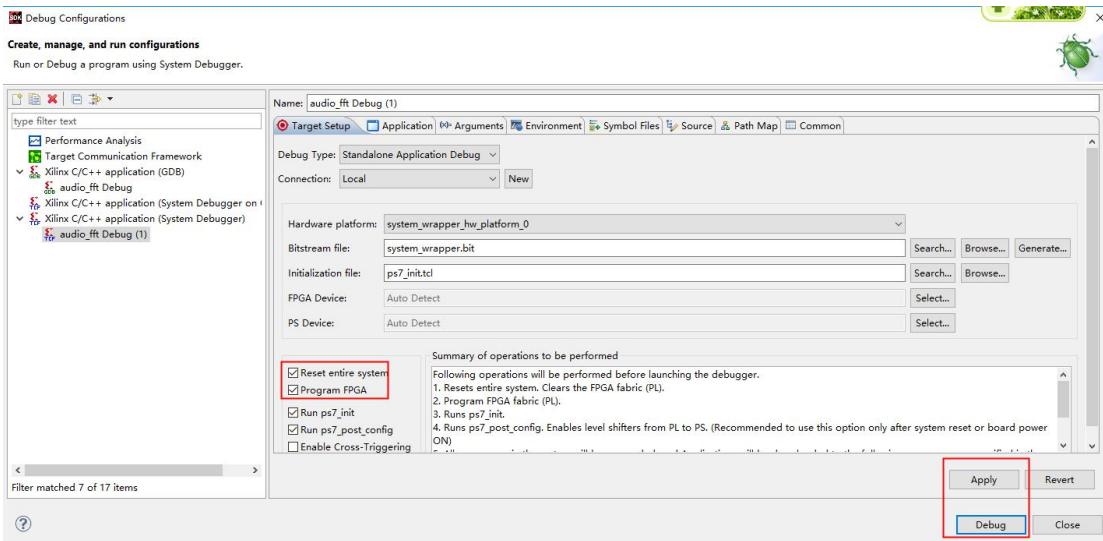
```

Step6: 右击工程，选择 Debug as ->Debug configuration。

Step7: 选中 system Debugger,双击创建一个系统调试。



Step8: 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。



程序运行之后，运行结果如下图所示：



在图中，左边部分是 SOBEL 处理之后的结果，右边是未经处理的源图像，在上图中我们看到经过 SOBEL 之后，图像的边缘被检测出来了。

6.4 程序分析

本章的程序比较简单，HLS 的驱动使用的都是官方的驱动，大家在使用的时候照葫芦画瓢即可。这里重点介绍一下本章的 VDMA 配置。在本章的硬件工程中，使用了两个 VDMA，其中 VDMA 只有一个

写通道，负责完成显示的缓存功能，VDMA1 有两个通道，负责将我们取模的数组送入内存和 SOBEL 处理后的数据缓存的作用。因此，在本章中一共使用了三个 VDMA 的通道，因此在本章中我们设置了三块内存，分别存放这三个通道的数据，如下图所示：

```
#define SOBEL_S2MM      0x08000000
#define SOBEL_MM2S      0x0A000000
#define DISPLAY_MM2S    0x0C000000
```

接下来就是对这两个 VDMA 进行配置，这里就比如说配置 VDMA，其配置如下：

```
@void SOBEL_VDMA_setting(unsigned int width,unsigned int height,unsigned int s2mm_addr,unsigned int mm2s_addr)
{
    //S2MM
    Xil_Out32(SOBEL_VDMA + 0x30, 0x4); //reset S2MM VDMA Control Register
    usleep(10);
    Xil_Out32(SOBEL_VDMA + 0x30, 0x0); //genlock
    Xil_Out32(SOBEL_VDMA + 0xAC, s2mm_addr); //S2MM Start Addresses
    Xil_Out32(SOBEL_VDMA + 0xAC+4, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xAC+8, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xA4, width*4); //S2MM Horizontal Size
    Xil_Out32(SOBEL_VDMA + 0xA8, width*4); //S2MM Frame Delay and Stride
    Xil_Out32(SOBEL_VDMA + 0x30, 0x3); //S2MM VDMA Control Register
    Xil_Out32(SOBEL_VDMA + 0xA0, height); //S2MM Vertical Size start an S2M
    // Xil_DCacheFlush();

    //MM2S
    Xil_Out32(SOBEL_VDMA + 0x00, 0x00000003); // enable circular mode
    Xil_Out32(SOBEL_VDMA + 0x5c, mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x60, mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x64, mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x58, (width*4)); // h offset
    Xil_Out32(SOBEL_VDMA + 0x54, (width*4)); // h size
    Xil_Out32(SOBEL_VDMA + 0x50, height); // v size
    //Xil_DCacheFlush();
}
```

这里的配置比较简单，就是往 VDMA 的寄存器中写入相应的值即可，在程序中都给出了对应的注释，分析的方法也在前面的教程中进行了介绍，这里不再反复的去讲解。直接看到以下程序部分：

```
show_img(0,0,DISPLAY_MM2S,(void*)SOBEL_S2MM,512,512);
show img(522,0,DISPLAY_MM2S,(void*)SOBEL_MM2S,512,512);
```

这个程序就是我们本章程序的描点程序，我们看看具体的函数定义：

```
@void show_img(u32 x, u32 y, u32 disp_base_addr, const unsigned char * addr, u32 size_x, u32 size_y)
{
    //计算图片左上角坐标
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    u32 start_addr=disp_base_addr;
    start_addr = disp_base_addr + 4*x + y*4*DIS_X;
    for(j=0;j<size_y;j++)
    {
        for(i=0;i<size_x;i++)
        {
            //if(type==0)
            //{
                //b = *(addr+(i+j*size_x)*4+2); //08
                //g = *(addr+(i+j*size_x)*4+1); //60
                //r = *(addr+(i+j*size_x)*4); //01
            //}
            //else
            //{
                b = *(addr+(i+j*size_x)*4); //08
                g = *(addr+(i+j*size_x)*4+1); //60
                r = *(addr+(i+j*size_x)*4+2); //01
            //}
            Xil_Out32((start_addr+(i+j*DIS_X)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
        }
    }
    Xil_DCacheFlush();
}
```

在这里，前两个参数是描点的起始位置，`disp_base_addr` 是要写入的位置，`addr` 是写入的数据的来源，之后两个是图像的尺寸大小，这个程序比较简单，就是从 `addr` 中提取数据，然后抽取出三原色数据，再重新排列，最后写入到对应的内存当中。`Xil_DcacheFlush()` 函数是一个刷内存函数，

负责将 SDK 中写入的数据刷入内存当中。本章中还有一个与之类似的程序，如下图所示：

```
void SOBEL_DDRWR(unsigned int addr,unsigned int cols,unsigned int lows)
{
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    for(i=0;i<cols;i++)
    {
        for(j=0;j<lows;j++)
        {
            b= gImage_lena[(j+i*cols)*4+1];
            g= gImage_lena[(j+i*cols)*4+2];
            r= gImage_lena[(j+i*cols)*4+3];
            Xil_Out32((addr+(j+i*cols)*4),((r<<24)|(g<<16)|(b<<8)|0x0));
        }
    }
    Xil_DCacheFlush();
}
```

这个函数顾名思义，也是一个 SOBEL 往 DDR 中写入寄存器的函数，写入的数据就是取模的图片的数据。回到描点函数的分析中，接下来我们来看看描点函数是怎么被调用的。

```
show_img(0,0,DISPLAY_MM2S,(void*)SOBEL_S2MM,512,512);
show_img(522,0,DISPLAY_MM2S,(void*)SOBEL_MM2S,512,512);
```

结合上面的讲解，可以得知，这里写的数据来源来自 SOBEL_S2MM（也就是 VDMA1 的读通道），要写入的地址是 DISPLAY_MM2S（也就是 VDMA0 的写通道），整个框架下来就是 VDMA1 将取模的数组刷入内存，通过 VDMA1 的写通道送给 SOBEL 处理，然后将处理后的结果放在 VDMA1 的读通道中，然后再操作 VDMA0 的写通道将 VDMA1 S2MM 里 SOBEL 的结果显示在屏幕上，思想比较简单易懂，这里的划分内存操作的方式是大家在 VDMA 使用中一种很常用的方法，大家可以深入的了解一下。

6.5 本章小结

本章介绍了上一章生成的 SOBEL IP 如何进行调用，如何正确的驱动 HLS 生成的 IP，另外本章还介绍了一种全新的 VDMA 的操作方式，通过划分内存的方式对图片进行显示，这种方法可以延伸到以后的算法处理当中，是一种很方便的操作方式。

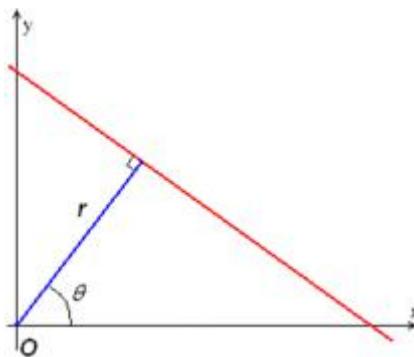
S05_CH07_基于 Hough 变换的圆检测

7.1 Hough 变换原理介绍

霍夫变换 (Hough Transform) 是图像处理中的一种特征提取技术，该过程在一个参数空间中通过计算累计结果的局部最大值得到一个符合该特定形状的集合作为霍夫变换结果。霍夫变换于 1962 年由 Paul Hough 首次提出，后于 1972 年由 Richard Duda 和 Peter Hart 推广使用，经典霍夫变换用来检测图像中的直线，后来霍夫变换扩展到任意形状物体的识别，多为圆和椭圆。霍夫变换运用两个坐标空间之间的变换将在一个空间中具有相同形状的曲线或直线映射到另一个坐标空间的一个点上形成峰值，从而把检测任意形状的问题转化为统计峰值问题。

7.1.1 Hough 变换直线检测

我们知道，一条直线在直角坐标系下可以用 $y=kx+b$ 表示，霍夫变换的主要思想是将该方程的参数和变量交换，即用 x, y 作为已知量 k, b 作为变量坐标，所以直角坐标系下的直线 $y=kx+b$ 在参数空间表示为点 (k, b) ，而一个点 (x_1, y_1) 在直角坐标系下表示为一条直线 $y_1=x_1 \cdot k+b$ ，其中 (k, b) 是该直线上的任意点。为了计算方便，我们将参数空间的坐标表示为极坐标下的 γ 和 θ 。因为同一条直线上的点对应的 (γ, θ) 是相同的，因此可以先将图片进行边缘检测，然后对图像上每一个非零像素点，在参数坐标下变换为一条直线，那么在直角坐标下属于同一条直线的点便在参数空间形成多条直线并内交于一点。因此可用该原理进行直线检测。



参数空间变换结果

如图所示，对于原图内任一点 (x, y) 都可以在参数空间形成一条直线，以图中一条直线为例有参数 $(\gamma, \theta)=(69.641, 30^\circ)$ ，所有属于同一条直线上的点会在参数空间交于一点，该点即为对应直线的参数。

7.1.2 Hough 变换圆检测

继使用 hough 变换检测出直线之后，顺着坐标变换的思路，提出了一种检测圆的方法。

1 如何表示一个圆？

与使用 (r, θ) 来表示一条直线相似，使用 (a, b, r) 来确定一个圆心为 (a, b) 半径为 r 的圆。

2 如何表示过某个点的所有圆？

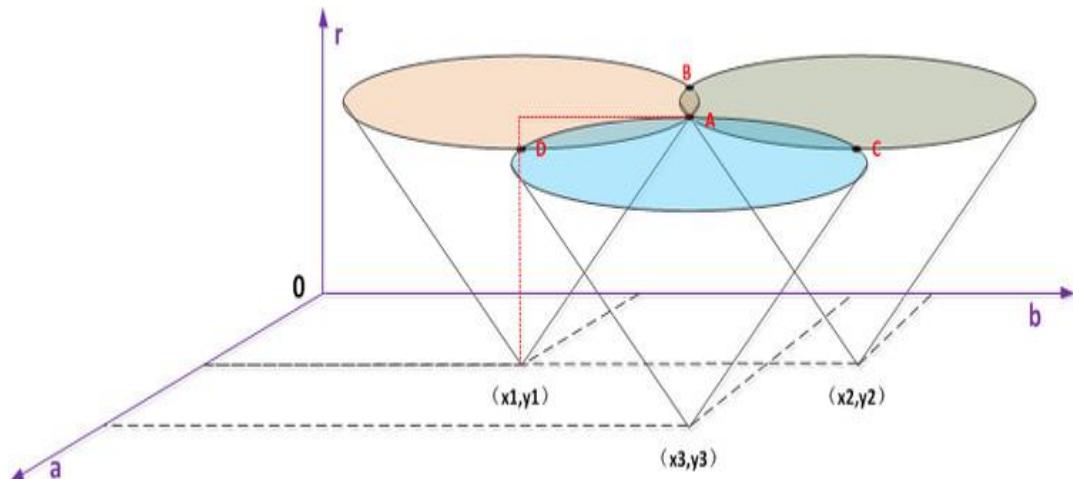
某个圆过点 (x_1, y_1) ，则有： $(x_1-a)^2 + (y_1-b)^2 = r^2$ 。

那么过点 (x_1, y_1) 的所有圆可以表示为 $(a_1(i), b_1(i), r_1(i))$ ，其中 $r_1 \in (0, \infty)$ ，每一个 i 值都对应一个不同的圆， $(a_1(i), b_1(i), r_1(i))$ 表示了无穷多个过点 (x_1, y_1) 的圆。

3 如何确定多个点在同一个圆上？

如(2)中说明，过点 (x_1, y_1) 的所有圆可以表示为 $(a_1(i), b_1(i), r_1(i))$ ，过点 (x_2, y_2) 的所有圆可以表示为 $(a_2(i), b_2(i), r_2(i))$ ，过点 (x_3, y_3) 的所有圆可以表示为 $(a_3(i), b_3(i), r_3(i))$ ，如果这三个点在同一个圆上，那么存在一个值 (a_0, b_0, r_0) ，使得 $a_0 = a_1(k) = a_2(k) = a_3(k)$ 且 $b_0 = b_1(k) = b_2(k) = b_3(k)$ 且 $r_0 = r_1(k) = r_2(k) = r_3(k)$ ，即这三个点同时在圆 (a_0, b_0, r_0) 上。

从下图可以形象的看出：



首先，分析过点 (x_1, y_1) 的所有圆 $(a_1(i), b_1(i), r_1(i))$ ，当确定 $r_1(i)$ 时， $(a_1(i), b_1(i))$ 的轨迹是一个以 $(x_1, y_1, r_1(i))$ 为圆心半径为 $r_1(i)$ 的圆。那么，所有圆 $(a_1(i), b_1(i), r_1(i))$ 的组成了一个以 $(x_1, y_1, 0)$ 为顶点，锥角为 90 度的圆锥面。

三个圆锥面的交点 A 既是同时过这三个点的圆。

7.1.3 Hough 变换圆检测算法实现流程

Hough 变换时一种利用图像的全局特征将特定形状边缘链接起来。它通过点线的对偶性，将源图像上的点影射到用于累加的参数空间，把原始图像中给定曲线的检测问题转化为寻找参数空间中的峰值问题。由于利用全局特征，所以受噪声和边界间断的影响较小，比较鲁棒。

Hough 变换思想为：在原始图像坐标系下的一个点对应了参数坐标系中的一条直线，同样参数坐标系的一条直线对应了原始坐标系下的一个点，然后，原始坐标系下呈现直线的所有点，它们的斜率和截距是相同的，所以它们在参数坐标系下对应于同一个点。这样在将原始坐标系下的各个点投影到参数坐标系下之后，看参数坐标系下有没有聚集点，这样的聚集点就对应了原始坐标系下的直线。

因此采用 hough 变换主要有以下几个步骤：

1) Detect the edge

检测得到图像的边缘

2) Create accumulator

采用二维向量描述图像上每一条直线区域，将图像上的直线区域计数器映射到参数空间中的存储单元， ρ 为直线区域到原点的距离，所以对于对角线长度为 n 的图像， ρ 的取值范围为 $(0, n)$ ， θ 值得取值范围为 $(0, 360)$ ，定义为二维数组 $\text{HoughBuf}[n][360]$ 为存储单元。

对所有像素点 (x, y) 在所有 θ 角的时候，求出 ρ 。从而累加 ρ 值出现的次数。高于某个阈值的 ρ 就是一个直线。这个过程就类似于横坐标是 θ 角， ρ 就是到直线的最短距离。横坐标 θ 不断变换，根据直线方程公司， $\rho = x\cos\theta + y\sin\theta$ 对于所有的不为 0 的像素点，计算出 ρ ，找到 ρ 在坐标 (θ, ρ) 的位置累加 1。

3) Detect the peaks, maximal in the accumulator

通过统计特性，假如图像平面上有两条直线，那么最终会出现 2 个峰值，累加得到最高的数组的值为所求直线参数。

7.2 Hough 在 HLS 上的实现

我们看下面一个实际问题：我们要从一副图像中检测出半径以知的圆形来。我们可以取和图像平面一样的参数平面，以图像上每一个前景点为圆心，以已知的半径在参数平面上画圆，并把结果进行累加。最后找出参数平面上的峰值点，这个位置就对应了图像上的圆心。在这个问题里，图像

平面上的每一点对应到参数平面上的一个圆。

把上面的问题改一下，假如我们不知道半径的值，而要找出图像上的圆来。这样，一个办法是把参数平面扩大称为三维空间。就是说，参数空间变为 $x - y - R$ 三维，对应圆的圆心和半径。图像平面上的每一点就对应于参数空间中每个半径下的一个圆，这实际上是一个圆锥。最后当然还是找参数空间中的峰值点。不过，这个方法显然需要大量的存储空间，运行速度也会是很大问题。

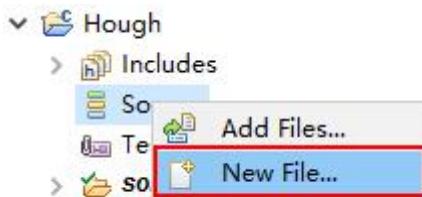
那么有什么比较好的解决方法么？我们前面假定的图像都是黑白图像（二值图像），实际上这些二值图像多是彩色或灰度图像通过边缘提取来的。我们前面提到过，图像边缘除了位置信息，还有方向信息也很重要，这里就用上了。根据圆的性质，圆的半径一定在垂直于圆的切线的直线上，也就是说，在圆上任意一点的法线上。这样，解决上面的问题，我们仍采用 2 维的参数空间，对于图像上的每一前景点，加上它的方向信息，都可以确定出一条直线，圆的圆心就在这条直线上。这样一来，问题就会简单了许多。

接下来我们来设计利用 Hough 变换来进行圆检测。

7.2.1 工程创建

Step1：打开 HLS，按照之前介绍的方法，创建一个新的工程，命名为 Hough。

Step2：右单击 Source 选项，选择 New File，创建一个名为 Top.cpp 的文件。



Step3：在打开的编辑区中，把下面的程序拷贝进去：

```
#include "top.h"
#include <stdio.h>
#include <iostream>

using namespace std;

void hls::hls_hough_line(GRAY_IMAGE &src, GRAY_IMAGE &dst, int
rows, int cols)
{
    GRAY_PIXEL result;
    int row ,col,k;
```

```
//参数空间的参数圆心O (a,b) 半径radius
int a = 0,b = 0, radius = 0;

//累加器
int A0 = rows;
int B0 = cols;

//注意HLS不支持变长数组，所以这里直接指定数据长度
const int Size = 1089900;//Size = rows*cols*(120-110);

#ifndef __SYNTHESIS__
    int _count[Size];
    int *count = &_count[0];
#else
    int *count =(int *) malloc(Size * sizeof(int));
#endif

//偏移
int index ;

//为累加器赋值0
for (row = 0;row < Size;row++)
{
    count[row] = 0;
}

GRAY_PIXEL src_data;
uchar temp0;
for (row = 0; row< rows;row++)
{
    for (col = 0; col< cols;col++)
    {
        src >> src_data;
        uchar temp = src_data.val[0];
        //检测黑线
        if (temp == 0)
        {
            //遍历a ,b 为;累加器赋值
            for (a = 0;a < A0;a++)
            {
                for (b = 0;b < B0;b++)
                {
                    radius = (int)(sqrt(pow((double)(row-a),2) +
```

```

pow((double)(col - b), 2));
    if(radius > 110 && radius < 120)
    {
        index = A0 * B0 *(radius-110) + A0*b + a;
        count[index]++;
    }
}
{
}
}

//遍历累加器数组，找出所有的圆
for (a = 0 ; a < A0 ; a++)
{
    for (b = 0 ; b < B0; b++)
    {
        for (radius = 110 ; radius < 120; radius++)
        {
            index = A0 * B0 *(radius-110) + A0*b + a;
            if (count[index] > 210)
            {
                //在image2中绘制该圆
                for(k = 0; k < rows;k++)
                {
                    for (col = 0 ; col< cols;col++)
                    {
                        //x有两个值，根据圆公式(x-a)^2+(y-b)^2=r^2得
                        int temp =
(int)(sqrt(pow((double)radius,2)- pow((double)(col-b),2)));
                        int x1 = a + temp;
                        int x2 = a - temp;
                        if ( (k == x1) || (k == x2) ){
                            result.val[0] = (uchar)255;
                        }
                        else{
                            result.val[0] = (uchar)0;
                        }
                        dst << result;
                    }
                }
            }
        }
    }
}

```

到

```

        }
    }

}

void hls_hough(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int rows,
int cols)
{
    GRAY_IMAGE img_src(rows, cols);
    GRAY_IMAGE img_dst(rows, cols);

    hls::AXIVideo2Mat(src_axi, img_src);
    hls::hls_hough_line(img_src, img_dst, rows, cols);
    hls::Mat2AXIVideo(img_dst, dst_axi);
}

```

代码中有一段非常需要注意的如下几行代码，这段代码中使用了动态分配内存的函数malloc，另外HLS中不支持C++中使用new关键字分配内存的方法。该函数只能用于在综合文件中进行仿真，主要是解决大数组所造成的的编译出问题，详细介绍可参考ug902关于Array的章节。

```

#ifdef __SYNTHESIS__
int _count[Size];
int *count = &_count[0];
else
int *count =(int *) malloc(Size * sizeof(int));
#endif

```

另外需要注意的是malloc 函数返回的是 void * 类型。对于C++，如果你写成：

int *count = **malloc**(Size * **sizeof(int)**);则程序无法通过编译，报错：“不能将 void* 赋值给 int * 类型变量”。所以必须通过 (**int** *) 来将强制转换。而对于C，没有这个要求，但为了使C程序更方便的移植到C++中来，建议养成强制转换的习惯。

Step4：再在 Source 中添加一个名为 Top.h 的库函数，并添加如下程序：

```

#ifndef _TOP_H_
#define _TOP_H_

//#include "iostream"
#include "hls_video.h"
#include "math.h"

//maximum image size
#define MAX_WIDTH 1920
#define MAX_HEIGHT 1080

```

```

//I/O Image Settings
#define INPUT_IMAGE           "circle2.bmp"
#define OUTPUT_IMAGE          "result.jpg"
#define OUTPUT_IMAGE_GOLDEN   "result_golden.jpg"

//typedef video library core structures
typedef hls::stream<ap_axiu<32,1,1,1> >           AXI_STREAM;
typedef hls::Scalar<3, unsigned char>                  RGB_PIXEL;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3>    RGB_IMAGE;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1>    GRAY_IMAGE;
typedef hls::Scalar<1, unsigned char>                  GRAY_PIXEL;
typedef unsigned char uchar;

//定义点的结构体
template<typename T>
struct hls_Point
{
    T x;
    T y;
};

//定义命名空间hls并进行函数声明
namespace hls
{
    double hls_round(double x, int n);
    void hls_hough_line(GRAY_IMAGE &src,GRAY_IMAGE &dst,int
rows,int cols);
}

//top level function for HW synthesis
void hls_hough(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int rows,
int cols);

#endif

```

Step5: 在 Test Bench 中, 用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码:

```

#include <iostream>
#include "hls_opencv.h"
#include "top.h"
#include "opencv_top.h"

using namespace cv;

```

```
using namespace std;

int main()
{
    //获取图像数据
    IplImage* src =
cvLoadImage(INPUT_IMAGE, CV_LOAD_IMAGE_GRAYSCALE); //获取仿真图片并
直接转为灰度图像
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels);
    cvShowImage("src", src);

    //使用HLS库进行处理
    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIvideo(src, src_axi);
    hls_hough(src_axi, dst_axi, src->height, src->width);
    AXIvideo2IplImage(dst_axi, dst);
    cvShowImage("hls_dst", dst);

    //使用OPENCV库进行处理
    opencv_hough(src, dst);
    cvShowImage("cv_dst", dst);

    waitKey(0);

    //释放内存
    cvReleaseImage(&dst);
    return 0;
}
```

Step6：用同样的方法，再在 Test Bench 中创建一个 opencv_top.cpp 和 Opencv_top.h 文件，添加如下程序：

Opencv_top.cpp 代码如下：

```
#include "opencv_top.h"
#include "top.h"

using namespace cv;
using namespace std;

void opencv_hough(IplImage* src, IplImage* dst)
{
    int i, j;
```

```
unsigned char *ptr, *dst_data;
//参数空间的参数 圆心O (a,b) 半径radius
int a = 0,b = 0,radius = 0;
//累加器
int A0 = src->height;
int B0 = src->width;
int R0 = (src->width > src->height)? 2*src->width :
2*src->height;//R0取长宽的最大值的2倍

int countLength = A0*B0*R0;
int *count = new int[countLength];
//偏移
int index = A0 * B0 *radius + A0*b + a;
//为累加器赋值0
for (i= 0;i<countLength;i++)
{
    count[i]=0;
}

for (i = 0 ; i< src->height; i++)
{
    for (j = 0 ; j< src->width ; j++)
    {
        ptr = (unsigned char *)src->imageData + i*src->widthStep
+ j;
        //检测黑线
        if (*ptr == 0 )
        {
            //遍历a ,b 为;累加器赋值
            for (a = 0 ; a< A0;a++)
            {
                for (b = 0; b< B0;b++)
                {
                    radius = (int)(sqrt(pow((double)(i-a),2) +
pow((double)(j - b),2)));
                    index = A0 * B0 *radius + A0*b + a;
                    count[index]++;
                }
            }
        }
    }
}
//image2全部赋值为0
```

```
for (i= 0; i<dst->height; i++)
{
    for (j = 0 ; j< dst->width;j++)
    {
        dst_data = (unsigned char *)dst->imageData +
i*dst->widthStep + j;
        *dst_data = 0;
    }
}

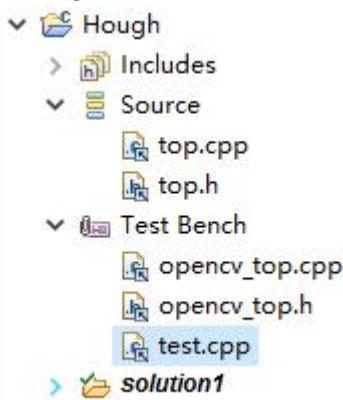
//遍历累加器数组，找出所有的圆
for (a = 0 ; a < A0 ; a++)
{
    for (b = 0 ; b< B0; b++)
    {
        for (radius = 0 ; radius < R0; radius++)
        {
            index = A0 * B0 *radius + A0*b + a;
            if (count[index] > 200)//值的不同会导致识别的效果差异
            {
                //在image2中绘制该圆
                for (j = 0 ; j< src->width;j++)
                {
                    i = (int)(sqrt(pow((double)radius,2)-
pow((double)(j-b),2)) + a);
                    if ((i< dst->height) && (i >= 0))
                    {
                        dst_data = (unsigned char*)(dst->imageData
+ i * dst->widthStep + j);
                        *dst_data = 255;
                    }
                    //i有两个值
                    i = a -(int)(sqrt(pow((double)radius,2)-
pow((double)(j-b),2)));
                    if ((i< dst->height) && (i >= 0))
                    {
                        dst_data = (unsigned char*)(dst->imageData
+ i * dst->widthStep + j);
                        *dst_data = 255;
                    }
                }
            }
        }
    }
}
```

```
    }  
}  
}
```

OpenCV_top.h 代码如下：

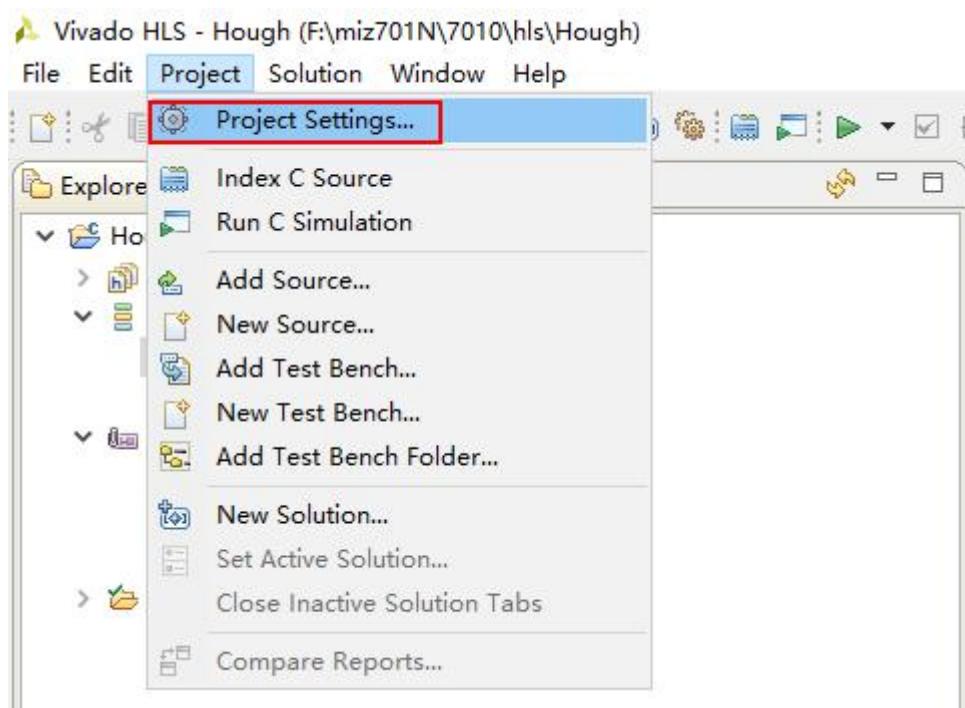
```
#ifndef __OPENCV_TOP_H__  
#define __OPENCV_TOP_H__  
  
#include "hls_opencv.h"  
  
void opencv_hough(IplImage* src, IplImage* dst);  
  
#endif
```

Step7：在 Test Bench 中添加一张名为 circle2.bmp 的测试图片，图片可以在我们提供的源程序中的 Image 文件夹中找到。完整的工程如下图所示：

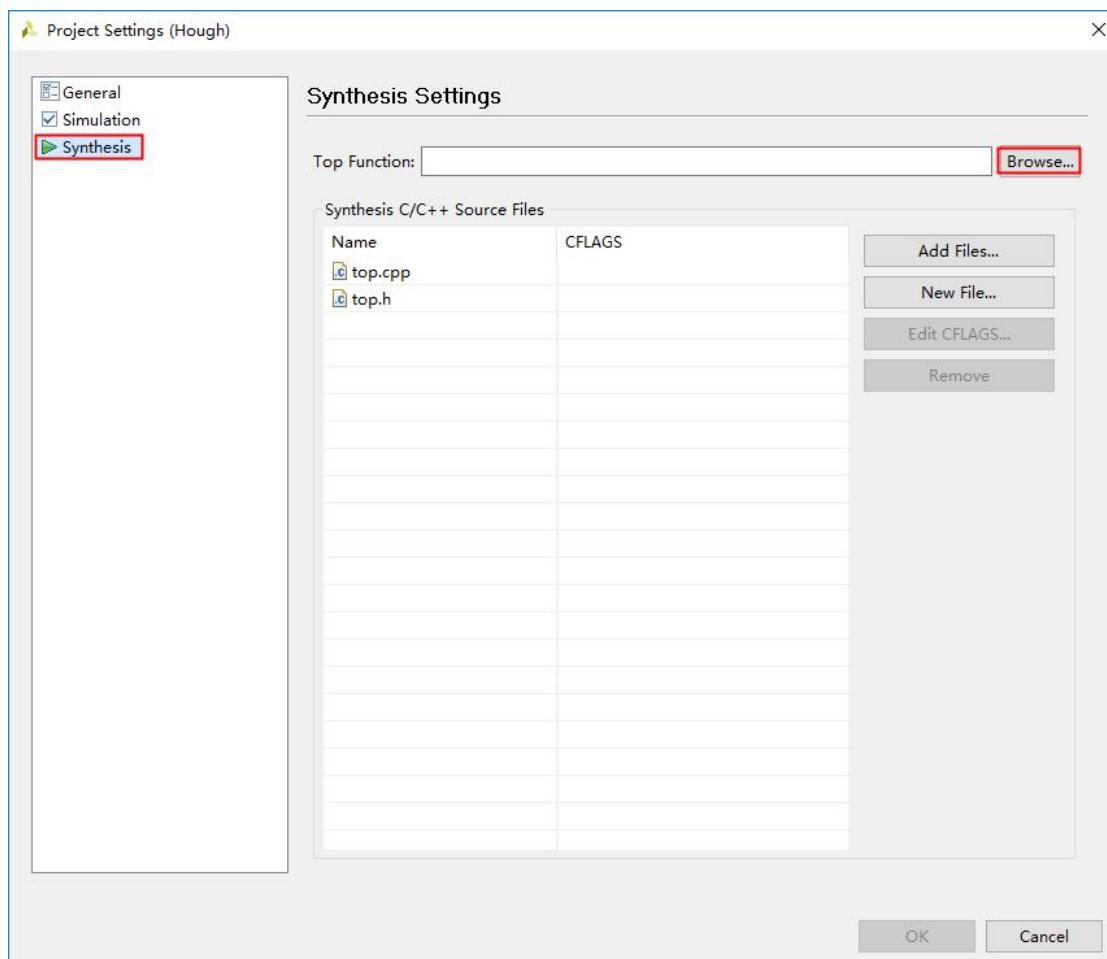


7.2.2 仿真及优化

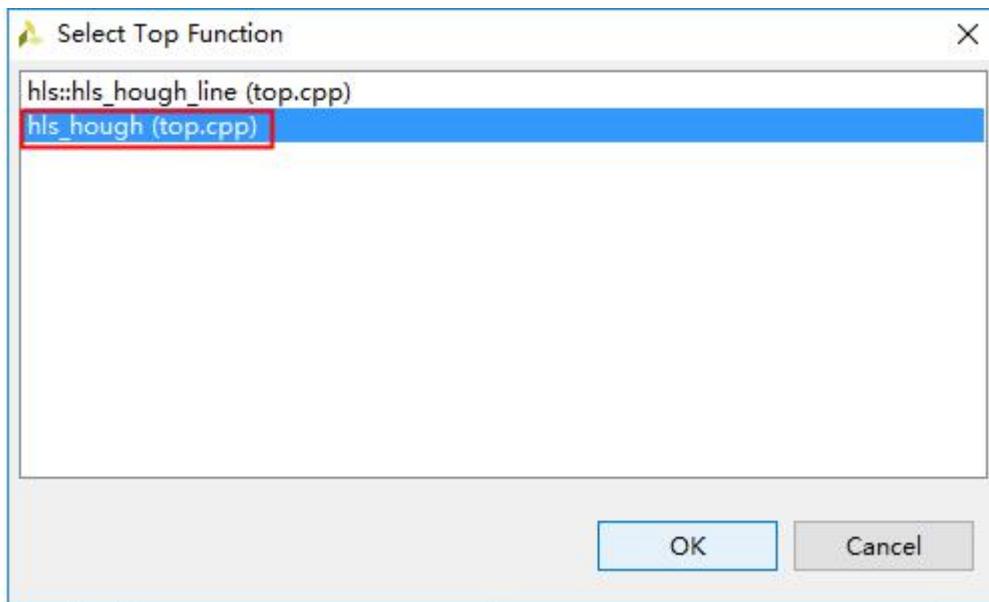
Step1：单击 Project→Project settings。



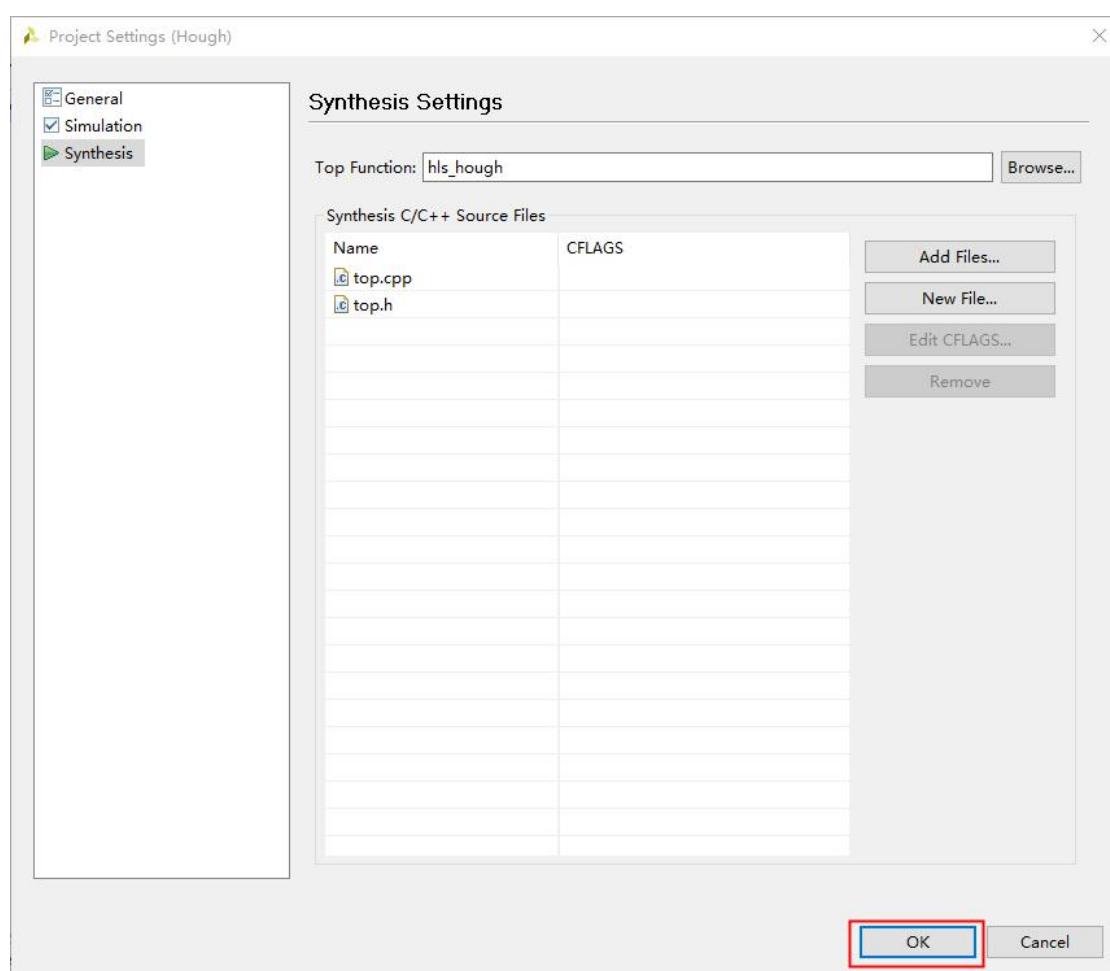
Step2: 选择 Synthesis 选项，然后点击 Browse.. 指定一个顶层函数。



Step3: 选定 hls_hough 为顶层函数，然后点击 O K。



Step4: 再次单击 OK，完成工程的修改。



Step5: 单击 开始综合。



综合报告如下：

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2179898	2481068676906370	2179899	2481068676906371	none

Detail

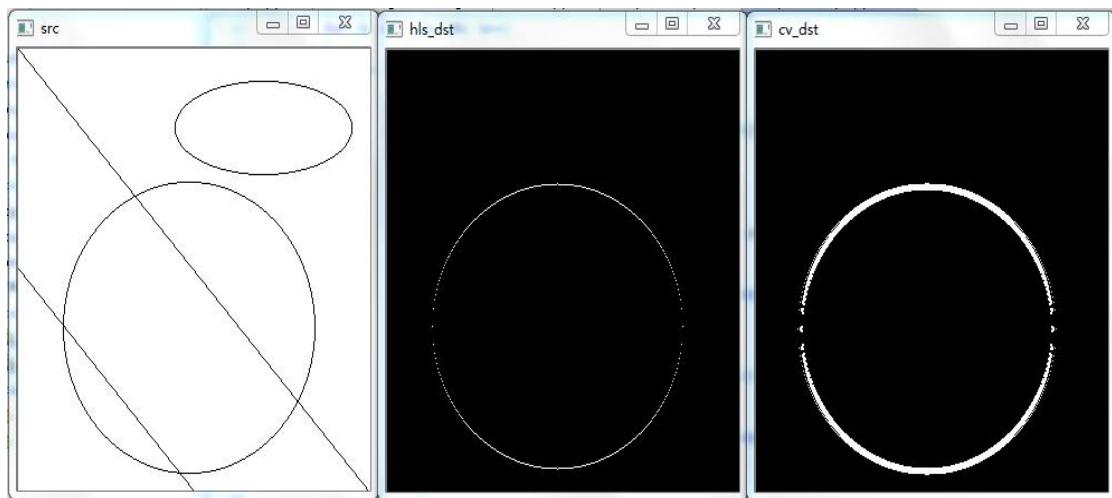
- Instance**
- Loop**

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	49
FIFO	0	-	10	40
Instance	4096	18	4301	7287
Memory	-	-	-	-
Multiplexer	-	-	-	34
Register	-	-	104	-
Total	4096	18	4415	7410
Available	120	80	35200	17600
Utilization (%)	3413	22	12	42

Step6：直接单击 开始进行仿真。系统运行结束后，处理结果如下所示：



在这里我们给出的是 hough 变换的圆的检测，各位也可以根据 HLS 提供的视频库进行 hough 变换最基本的直线检测，这里给出函数的简单示例：

```

hls::Scalar<3,unsigned char> color(50,50,50);
const unsigned int linesMax = 3;
hls::Polar<float,float> Polar_line[linesMax];
hls::HoughLines2<1,2>(src,Polar_line,150);
//依次在图中绘制出每条线段
for( int i = 0; i < linesMax; i++ )
{
    float rho = Polar_line[i].rho, theta = Polar_line[i].angle;
    hls_Point<int> pt1, pt2;
    ...
}

```

其中 hls_Point 的结构体原型为：

```

//定义点的结构体
template<typename T>
struct hls_Point
{
    T x;
    T y;
};

```

Hough 变换还可以解决许多类似的问题，如检测出椭圆，正方形，长方形，圆弧等等。这些方法大都类似，关键就是需要熟悉这些几何形状的数学性质。霍夫变换由于不受图像旋转的影响，所以很容易的可以用来进行定位。

霍夫变换有许多改进方法，一个比较重要的概念是广义霍夫变换，它是针对所有曲线的，用处也很大。就是针对直线的霍夫变换也有很多改进算法，比如前面的方法我们没有考虑图像上的这一直线上的点是否连续的问题，这些都要随着应用的不同而有优化的方法。

简单点概括就是：Hough 变换的基本原理在于利用点与线的对偶性，将原始图像空间的给定的曲线通过曲线表达形式变为参数空间的一个点。这样就把原始图像中给定曲线的检测问题转化为寻找参数空间中的峰值问题。也即把检测整体特性转化为检测局部特性。比如直线、椭圆、圆、弧线等。算法如下：

- (1) 初始化一个变换域空间的数组， r 方向上的量化数目图像对角线方向像素， 0 方向上的量化数目为 180。

(2) 顺序搜索图像的所有黑点。对每一个黑点，在变换域的对应点加一。

(3) 求出变换域最大值点并记录。最大值点就是直线的所在了。

或者说：hough 变换利用图像空间和 hough 参数空间的点一线对偶性，把图像空间中的检测问题转换到参数空间。通过在参数空间里进行简单的累加统计，然后在 hough 参数空间寻找累加器峰值的方法检测直线。

7.3 程序分析

本章节的程序比较简单，主要的圆检测功能函数如下图所示：

```

28 void hls::hls_hough_line(GRAY_IMAGE &src,GRAY_IMAGE &dst,int rows,int cols)
29 {
30     GRAY_PIXEL result;
31     int row,col,k;
32     //参数空间的参数圆心O (a,b) 半径radius
33     int a = 0,b = 0,radius = 0;
34
35     //累加?
36     int A0 = rows;
37     int B0 = cols;
38
39     //注意HLS不支持变长数组，以这里直接指定数据长?
40     const int Size = 1089900;//Size = rows*cols*(120-110);
41
42 #ifdef __SYNTHESIS__
43     int _count[Size];
44     int *count = &_count[0];
45 #else
46     int *count =(int *) malloc(Size * sizeof(int));
47 #endif
48
49     //偏移
50     int index ;
51
52     //为累加器赋?
53     for (row = 0;row < Size;row++)
54     {
55         count[row] = 0;
56     }
57
58     GRAY_PIXEL src_data;
59     uchar temp0;
60     for (row = 0; row< rows;row++)
61     {
62         for (col = 0; col< cols;col++)
63         {
64             src >> src_data;
65             uchar temp = src_data.val[0];
66             //测黑?
67             if (temp == 0)
68             {
69                 //遍历a ,b ?,累加器赋?
70                 for (a = 0;a < A0;a++)
71                 {
72                     for (b = 0;b < B0;b++)
73                     {
74                         radius = (int)(sqrt(pow((double)(row-a),2) + pow((double)(col - b),2)));
75                         if(radius > 110 && radius < 120)
76                         {
77                             index = A0 * B0 *(radius-110) + A0*b + a;
78                             count[index]++;
79                         }
80                     }
81                 }
82             }
83         }
84     }
85 }
```

```

86 //遍历累加器数组，找出◆?有的◆?
87 for (a = 0; a < A0; a++)
88 {
89     for (b = 0; b < B0; b++)
90     {
91         for (radius = 110; radius < 120; radius++)
92         {
93             index = A0 * B0 *(radius-110) + A0*b + a;
94             if (count[index] > 210)
95             {
96                 //在image2中绘制该◆?
97                 for(k = 0; k < rows;k++)
98                 {
99                     for (col = 0; col< cols;col++)
100                     {
101                         //x有两个◆?，根据圆公◆?(x-a)^2+(y-b)^2=r^2得到
102                         int temp = (int)(sqrt(pow((double)radius,2)- pow((double)(col-b),2)));
103                         int x1 = a + temp;
104                         int x2 = a - temp;
105                         if ( (k == x1)|| (k == x2) )
106                             result.val[0] = (uchar)255;
107                         else{
108                             result.val[0] = (uchar)0;
109                         }
110                         dst << result;
111                     }
112                 }
113             }
114         }
115     }
116 }
117 }
118 }

```

在程序中有一些必要的注释，大家可以看一看。之前的一些变量定义在此都给出了释义，在此就直接跳过，先看到此函数中的这一部分，如下图：

```

52 //为累加器赋◆?0
53 for (row = 0;row < Size;row++)
54 {
55     count[row] = 0;
56 }
57
58 GRAY_PIXEL src_data;
59 uchar temp0;
60 for (row = 0; row< rows;row++)
61 {
62     for (col = 0; col< cols;col++)
63     {
64         src >> src_data;
65         uchar temp = src_data.val[0];
66         //◆?测黑◆?
67         if (temp == 0)
68         {
69             //遍历a ,b◆?:累加器赋◆?
70             for (a = 0;a < A0;a++)
71             {
72                 for (b = 0;b < B0;b++)
73                 {
74                     radius = (int)(sqrt(pow((double)(row-a),2) + pow((double)(col - b),2)));
75                     if(radius > 110 && radius < 120)
76                     {
77                         index = A0 * B0 *(radius-110) + A0*b + a;
78                         count[index]++;
79                     }
80                 }
81             }
82         }
83     }
84 }

```

一开始，程序中用了一个 for 循环，把 count 数组中的内容清零清理，这里需要提出的是 Size 的容量问题，在其定义时也给出了相关的注释，如下图所示：

```

//注意HLS不支持变长数组，◆?以这里直接指定数据长◆?
const int Size = 108900;//Size = rows*cols*(120-110);

```

这里方便仿真，我们直接限定了其范围，将数组的范围定义成了一个固定的数值，但是需要注意的是如果在实际的项目当中，就应该对内存进行操作。在 size 公式中的 (120-110) 是表示圆的半径在

110-120 个摄像点之间。回到函数的分析当中，接下来，函数定义了两个变量，我们注意到第一个变量的类型为一个名为 GRAY_PIXEL 的关键字，我们将鼠标停放在这个关键字上查看其定义，如下图所示：

```

39 typedef hls::Scalar<3, unsigned char>           RGB_PIXEL;
40 typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3>   RGB_IMAGE;
41 typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1>   GRAY_IMAGE;
42 typedef hls::Scalar<1, unsigned char>           GRAY_PIXEL;
43 typedef unsigned char uchar;
44

```

从上图中我们注意到有一个与其非常相似的定义，通过二者的比较，我们可以猜测这必定是定义了一种色彩空间，其类型为无符号整型数据，因此这里的 src_data 应该就是一个灰度值数据。接下来看到这一句：

```

for (row = 0; row < rows; row++)
{
    for (col = 0; col < cols; col++)
    {
        src >> src_data;
        uchar temp = src_data.val[0];
        //测黑?
        if (temp == 0)
    }
}

```

一开始，是个双重 for 语句，将行列的数据逐个的赋值给了 src_data，然后将 temp 指向了 src_data 的开始位，也就是说 temp 指向的是图像的起始处。然后是个 if 判断，这个 if 判断在这就是起了个检测图像的边框的黑线的作用，简单的来说就是检测图像的起始地址。然后看到接下来的这一部分：

```

//遍历a,b累加器赋值?
for (a = 0; a < A0; a++)
{
    for (b = 0; b < B0; b++)
    {
        radius = (int) (sqrt(pow((double) (row - a), 2) + pow((double) (col - b), 2)));
        if (radius > 110 && radius < 120)
        {
            index = A0 * B0 * (radius - 110) + A0 * b + a;
            count[index]++;
        }
    }
}

```

这部分就是遍历 a,b 为累加器赋值！这里的 radius 就是求出圆的半径，也就是我们第一节中讲到的

检测过圆的点的公式：
$$(x_1 - a_1)^2 + (y_1 - b_1)^2 = r_1^2$$
。之后求出的

半径在于实际的圆的半径对比，如果符合条件，就记下这点的偏移量，然后将这个偏移量存放在 count 数组对应的偏移地址中。经过这一个过程，所有过圆的点就将全部都会记录在 count 数组当中。最后分析一下这段代码：

```
//遍历累加器数组，找出是否有
for (a = 0 ; a < A0 ; a++)
{
    for (b = 0 ; b < B0; b++)
    {
        for (radius = 110 ; radius < 120; radius++)
        {
            index = A0 * B0 *(radius-110) + A0*b + a;
            if (count[index] > 210)
            {
                //在image2中绘制该点
                for(k = 0; k < rows;k++)
                {
                    for (col = 0 ; col< cols;col++)
                    {
                        //x有两个点，根据圆公式(x-a)^2+(y-b)^2=r^2得到
                        int temp = (int)(sqrt(pow((double)radius,2) - pow((double)(col-b),2)));
                        int x1 = a + temp;
                        int x2 = a - temp;
                        if ( (k == x1) || (k == x2) ){
                            result.val[0] = (uchar)255;
                        }
                        else{
                            result.val[0] = (uchar)0;
                        }
                        dst << result;
                    }
                }
            }
        }
    }
}
```

在这部分程序当中也是先遍历 a, b, 然后在指定的半径中, 先求出了一个偏移量 index, 然后判断在 count 数组的这个偏移量位置中的数据是否大于 210, 这里的这个 210 是我们设置的一个阀值, 实际上也可以取其他的数值, 之后在指定范围 (rows,cols) 内, 通过特定的公式判断这点是否是圆上的一点, 若是则用 255 代替原来的数值, 否则用 0 代替原来的数值, 最后将结果赋值给了目标图像。

7.4 本章小结

本章介绍了如何使用 HLS 设计一个基于 Hough 变换的圆检测算法, 通过这种方法, 还可以对其进行拓展, 设计基于 Hough 变换的直线检测算法, 感兴趣的可以尝试一下, 在一些特征物体具有特定的形状时, 此算法可方便对目标进行识别, 方便定位, 具有一定得实用价值。

S05_CH08_傅里叶变换的 HLS 实现

8.1 FFT 原理介绍

FFT 是离散傅立叶变换的快速算法，可以将一个信号变换到频域。有些信号在时域上是很难看出什么特征的，但是如果变换到频域之后，就很容易看出特征了。这就是很多信号分析采用 FFT 变换的原因。另外，FFT 可以将一个信号的频谱提取出来，这在频谱分析方面也是经常用的。在笔者参与的几个项目中就有几个使用到了 FFT，因此在这里准备在 HLS 上实现这一算法。另外在后面的几个实验中我们都用到了这一算法，因此前面先对它进行讲解，方便大家在接下来的实验中进行使用。

FFT 结果的物理意义网上有一大神圈圈对此做了详细的描述，我们在这里摘录如下方便大家理解 FFT。

一个模拟信号，经过 ADC 采样之后，就变成了数字信号。根据采样定理，采样频率要大于信号频率的两倍。采样得到的数字信号，就可以做 FFT 变换了。N 个采样点，经过 FFT 之后，就可以得到 N 个点的 FFT 结果。为了方便进行 FFT 运算，通常 N 取 2 的整数次方。

假设采样频率为 F_s ，信号频率 F ，采样点数为 N 。那么 FFT 之后结果就是一个为 N 点的复数。每一个点就对应着一个频率点。这个点的模值，就是该频率值下的幅度特性。具体跟原始信号的幅度有什么关系呢？假设原始信号的峰值为 A ，那么 FFT 的结果的每个点（除了第一个点直流分量之外）的模值就是 A 的 $N/2$ 倍。而第一个点就是直流分量，它的模值就是直流分量的 N 倍。而每个点的相位就是在该频率下的信号的相位。第一个点表示直流分量（即 0Hz），而最后一个点 N 的再下一个点（实际上这个点是不存在的，这里是假设的第 $N+1$ 个点，也可以看做是将第一个点分做两半分，另一半移到最后）则表示采样频率 F_s ，中间被 $N-1$ 个点平均分成 N 等份，每个点的频率依次增加。例如某点 n 所表示的频率为： $F_n = (n-1)*F_s/N$ 。由上面的公式可以看出， F_n 所能分辨到频率为 F_s/N ，如果采样频率 F_s 为 1024Hz，采样点数为 1024 点，则可以分辨到 1Hz。1024Hz 的采样率采样 1024 点，刚好是 1 秒，也就是说，采样 1 秒时间的信号并做 FFT，则结果可以分析到 1Hz，如果采样 2 秒时间的信号并做 FFT，则结果可以分析到 0.5Hz。如果要提高频率分辨率，则必须增加采样点数，也即采样时间。频率分辨率和采样时间是倒数关系。

假设 FFT 之后某点 n 用复数 $a+bi$ 表示，那么这个复数的模就是 $A_n = \sqrt{a^2+b^2}$ ，相位就是 $P_n = \text{atan2}(b, a)$ 。根据以上的结果，就可以计算出 n 点 ($n \neq 1$, 且 $n < N/2$) 对应的信号的表达式为：

$A_n / (N/2) * \cos(2\pi F_n t + P_n)$, 即 $2 * A_n / N * \cos(2\pi F_n t + P_n)$ 。

对于 $n=1$ 点的信号, 是直流分量, 幅度即为 A_1 / N 。

由于 FFT 结果的对称性, 通常我们只使用前半部分的结果, 即小于采样频率一半的结果。

假设我们有一个信号, 它含有 2V 的直流分量, 频率为 50Hz、相位为 -30 度、幅度为 3V 的交流信号, 以及一个频率为 75Hz、相位为 90 度、幅度为 1.5V 的交流信号。用数学表达式就是如下:

$$S = 2 + 3 * \cos(2\pi * 50 * t - \pi * 30 / 180) + 1.5 * \cos(2\pi * 75 * t + \pi * 90 / 180)$$

式中 cos 参数为弧度, 所以 -30 度和 90 度要分别换算成弧度。我们以 256Hz 的采样率对这个信号进行采样, 总共采样 256 点。按照我们上面的分析, $F_n = (n-1) * F_s / N$, 我们可以知道, 每两个点之间的间距就是 1Hz, 第 n 个点的频率就是 $n-1$ 。我们的信号有 3 个频率: 0Hz、50Hz、75Hz, 应该分别在第 1 个点、第 51 个点、第 76 个点上出现峰值, 其它各点应该接近 0。实际情况如何呢?

我们来看看 FFT 的结果的模值如图所示。

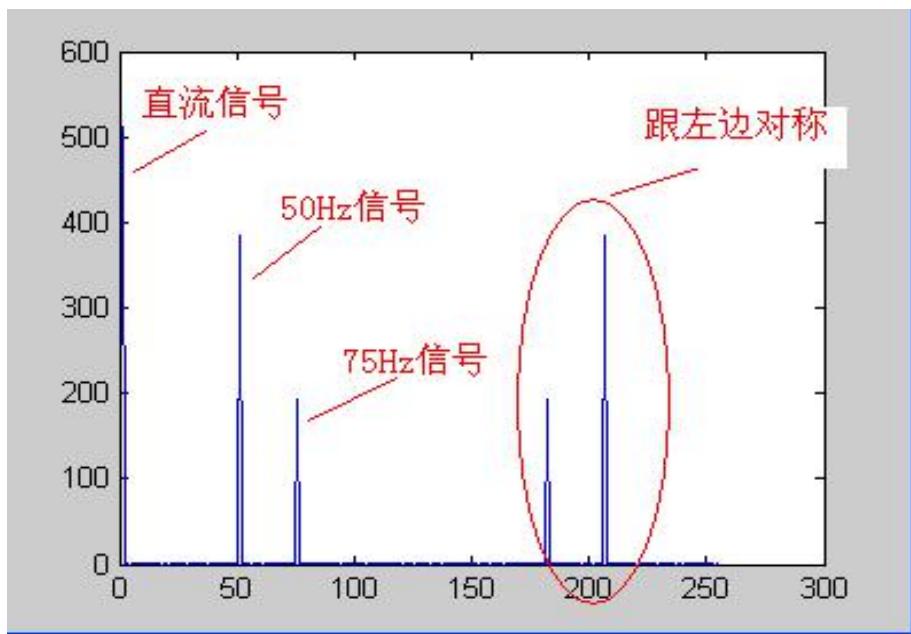


图 1 FFT 结果

从图中我们可以看到, 在第 1 点、第 51 点、和第 76 点附近有比较大的值。我们分别将这三个点附近的数据拿上来细看:

1 点: $512+0i$

2 点: $-2.6195E-14 - 1.4162E-13i$

3 点: $-2.8586E-14 - 1.1898E-13i$

50 点: $-6.2076E-13 - 2.1713E-12i$

51 点: 332.55 - 192i

52 点: -1.6707E-12 - 1.5241E-12i

75 点: -2.2199E-13 -1.0076E-12i

76 点: 3.4315E-12 + 192i

77 点: -3.0263E-14 +7.5609E-13i

很明显，1点、51点、76点的值都比较大，它附近的点值都很小，可以认为是0，即在那些频率点上的信号幅度为0。接着，我们来计算各点的幅度值。分别计算这三个点的模值，结果如下：

1 点: 512

51 点: 384

76 点: 192

按照公式，可以计算出直流分量为： $512/N=512/256=2$ ；50Hz 信号的幅度为： $384/(N/2)=384/(256/2)=3$ ；75Hz 信号的幅度为 $192/(N/2)=192/(256/2)=1.5$ 。可见，从频谱分析出来的幅度是正确的。

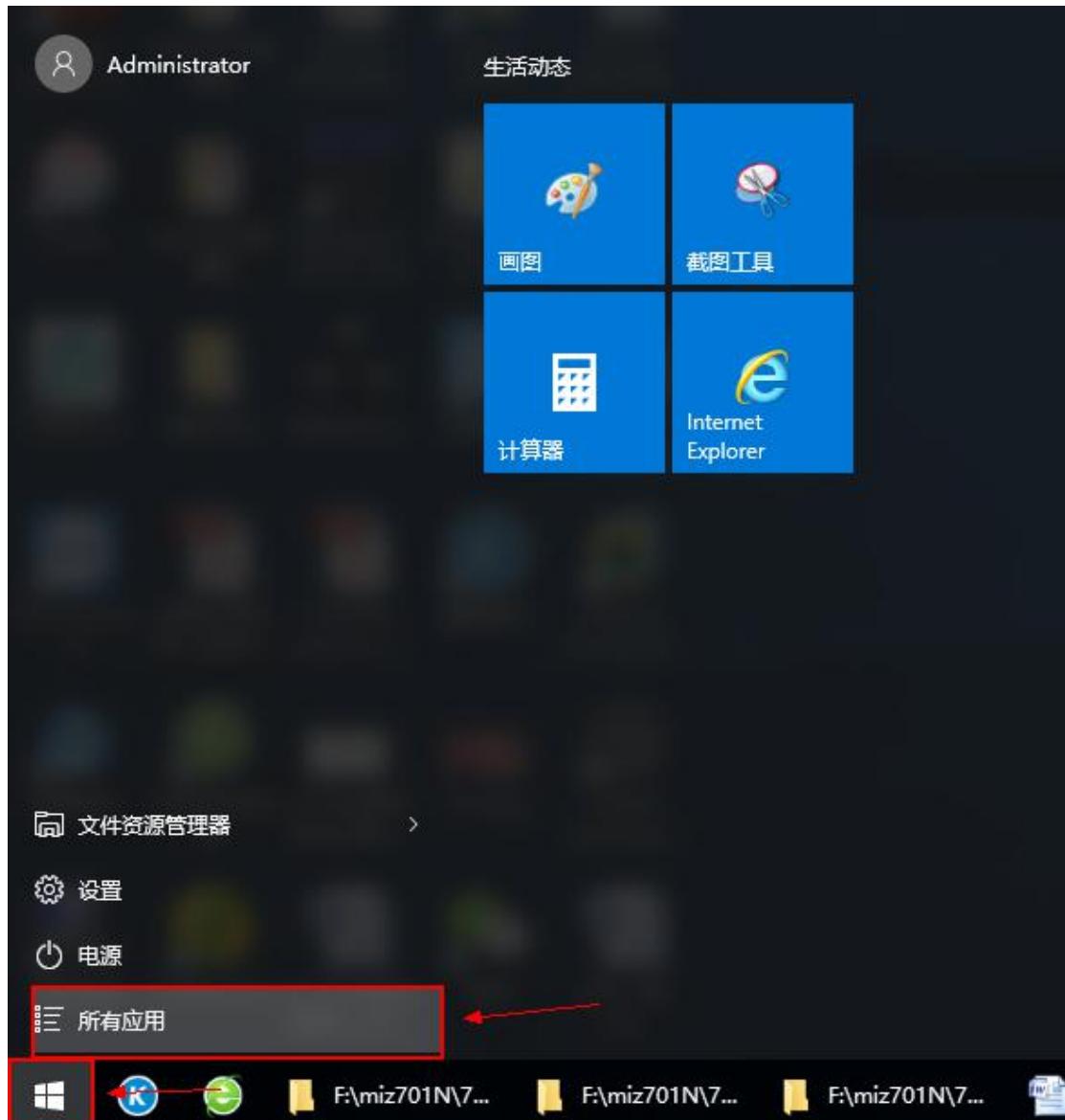
然后再来计算相位信息。直流信号没有相位可言，不用管它。先计算 50Hz 信号的相位， $\text{atan2}(-192, 332.55)=-0.5236$ ，结果是弧度，换算为角度就是 $180*(-0.5236)/\pi=-30.0001$ 。再计算 75Hz 信号的相位， $\text{atan2}(192, 3.4315E-12)=1.5708$ 弧度，换算成角度就是 $180*1.5708/\pi=90.0002$ 。可见，相位也是对的。根据 FFT 结果以及上面的分析计算，我们就可以写出信号的表达式了，它就是我们开始提供的信号。

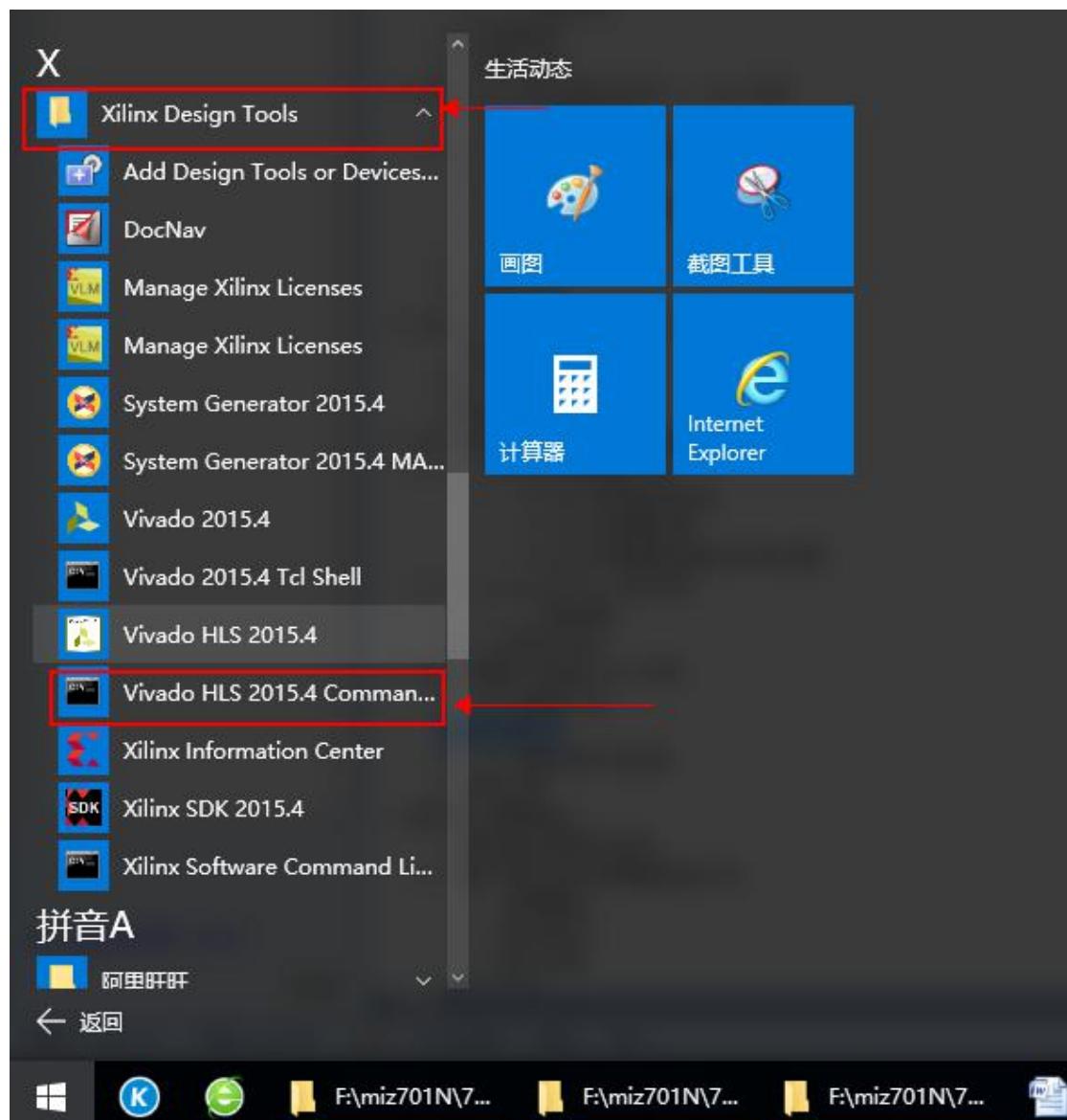
总结：假设采样频率为 F_s ，采样点数为 N ，做 FFT 之后，某一点 n （ n 从 1 开始）表示的频率为： $F_n=(n-1)*F_s/N$ ；该点的模值除以 $N/2$ 就是对应该频率下的信号的幅度（对于直流信号是除以 N ）；该点的相位即是对应该频率下的信号的相位。相位的计算可用函数 $\text{atan2}(b, a)$ 计算。 $\text{atan2}(b, a)$ 是求坐标为 (a, b) 点的角度值，范围从 $-\pi$ 到 π 。要精确到 x Hz，则需要采样长度为 $1/x$ 秒的信号，并做 FFT。要提高频率分辨率，就需要增加采样点数，这在一些实际的应用中是不现实的，需要在较短的时间内完成分析。解决这个问题的方法有频率细分法，比较简单的方法是采样比较短时间的信号，然后在后面补充一定数量的 0，使其长度达到需要的点数，再做 FFT，这在一定程度上能够提高频率分辨率。具体的频率细分法可参考相关文献。

8.2 HLS 实现

在官方的参考手册 ug871-vivado-high-level-synthesis-tutorial.pdf 中对 FFT 有比较详细的介绍，并且官方提供的 Example 中也有对应的工程方便大家学习，这一节我们主要说一下如何使用 TCL 语言创建工程，并且进行对应的操作，源码直接使用官方提供的。

Step1: 在开始菜单中找到并打开 vivado hls 2015.4 command (此处以 win10 操作系统为例, win7 操作系统区别不是很大)。





Step2:一般通过 cd 路径即可进入你想进入的文件路径，比如我们想进入我们的工作路径，输入 cd E:\work\miz702，我们看红色标注的地方会发现路径没有切换，那么这个问题我们该怎么解决呢？输入 cd/?查看系统给出的解决方案，那么进入盘符我们通过使用 cd /d 工作文件夹路径的命令进行路径切换，

```
ex Vivado HLS 2015.4 Command Prompt
=====
== Vivado HLS Command Prompt
== Available commands:
== vivado_hls, apcc, gcc, g++, make
=====
Microsoft Windows [版本 10.0.10586]
(c) 2015 Microsoft Corporation。保留所有权利。
d:\Xilinx\Vivado_HLS\2015.4>cd f:\miz701N\7010\hls\hls_fft\tcl
d:\Xilinx\Vivado_HLS\2015.4>
```

搜狗拼音输入法 全 :

```
选择Vivado HLS 2015.4 Command Prompt
d:\Xilinx\Vivado_HLS\2015.4>cd f:\miz701N\7010\hls\hls_fft\tcl
d:\Xilinx\Vivado_HLS\2015.4>cd /?
显示当前目录名或改变当前目录。
CHDIR [/D] [drive:][path]
CHDIR [..]
CD [/D] [drive:][path]
CD [..]
... 指定要改成父目录。
键入 CD drive: 显示指定驱动器中的当前目录。
不带参数只键入 CD, 则显示当前驱动器和目录。
使用 /D 开关, 除了改变驱动器的当前目录之外,
还可改变当前驱动器。
如果命令扩展被启用, CHDIR 会如下改变:
当前的目录字符串会被转换成使用磁盘名上的大小写。所以,
如果磁盘上的大小写如此, CD C:\TEMP 会将当前目录设为
C:\Temp。
CHDIR 命令不把空格当作分隔符, 因此有可能将目录名改为一个
带有空格但不带有引号的子目录名。例如:
cd \winnt\profiles\username\programs\start menu
与下列相同:
请按任意键继续. . .
搜狗拼音输入法 全 :
```

```
d:\Xilinx\Vivado_HLS\2015.4>cd /d f:\miz701N\7010\hls\hls_fft\tcl
f:\miz701N\7010\hls\hls_fft\tcl>
```

Step3:路径切换以后我们通过 vivado_hls -f run_hls.tcl 命令对 tcl 文件进行编译，注意路径及命令格式(所有的文件在我们提供的源程序包中的 src 文件夹中可以找到)：

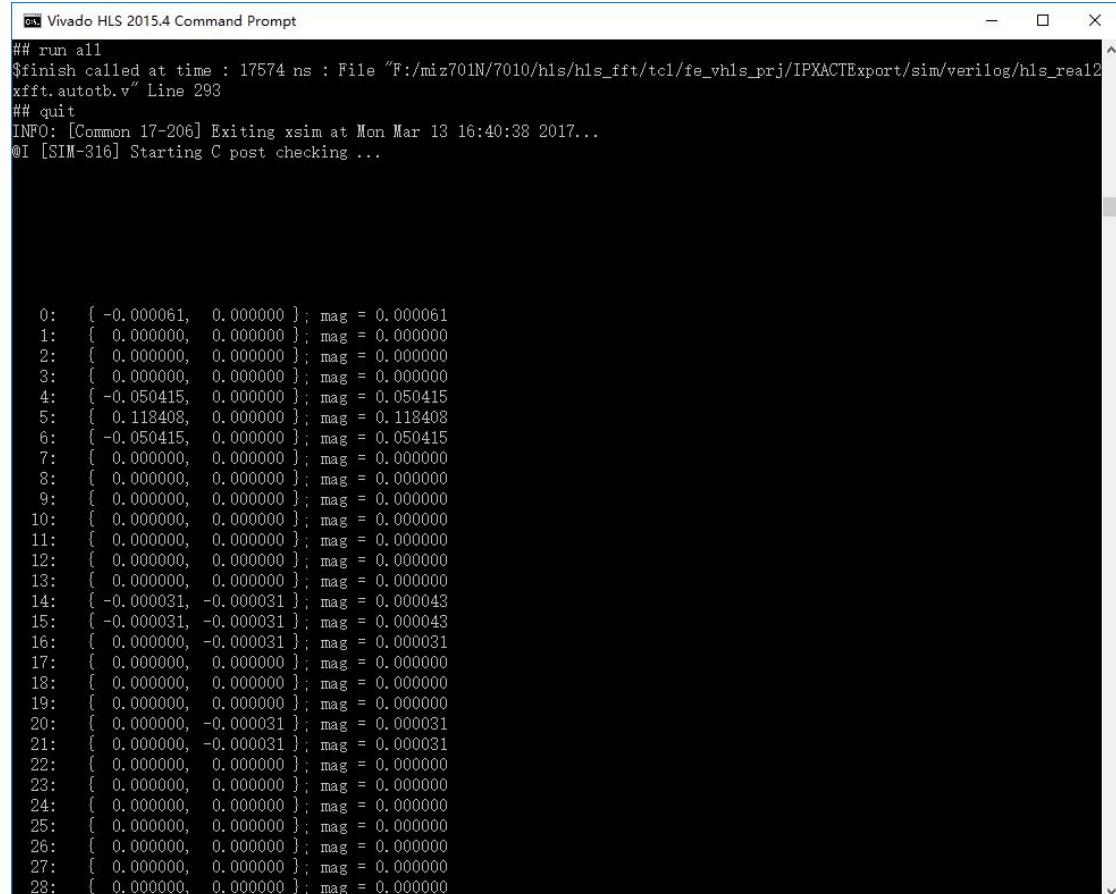
```
cd "\winnt\profiles\username\programs\start menu"
```

在扩展停用的情况下，你必须键入以上命令。

```
d:\Xilinx\Vivado_HLS\2015.4>cd /d f:\miz701N\7010\hls\hls_fft\tcl
```

```
f:\miz701N\7010\hls\hls_fft\tcl>vivado_hls -f run_hls.tcl
```

Step4:运行命令后，系统就将自动的完成 tcl 脚本中的设定，运行信息和结果如下所示：



```
## run all
$finish called at time : 17574 ns : File "T:/miz701N/7010/hls/hls_fft/tcl/fe vhls_prj/IPXACTExport/sim/verilog/hls_real2xfft.autotb.v" Line 293
## quit
INFO: [Common 17-206] Exiting xsim at Mon Mar 13 16:40:38 2017...
@I [SIM-316] Starting C post checking ...

0: { -0.000061, 0.000000 }; mag = 0.000061
1: { 0.000000, 0.000000 }; mag = 0.000000
2: { 0.000000, 0.000000 }; mag = 0.000000
3: { 0.000000, 0.000000 }; mag = 0.000000
4: { -0.050415, 0.000000 }; mag = 0.050415
5: { 0.118408, 0.000000 }; mag = 0.118408
6: { -0.050415, 0.000000 }; mag = 0.050415
7: { 0.000000, 0.000000 }; mag = 0.000000
8: { 0.000000, 0.000000 }; mag = 0.000000
9: { 0.000000, 0.000000 }; mag = 0.000000
10: { 0.000000, 0.000000 }; mag = 0.000000
11: { 0.000000, 0.000000 }; mag = 0.000000
12: { 0.000000, 0.000000 }; mag = 0.000000
13: { 0.000000, 0.000000 }; mag = 0.000000
14: { -0.000031, -0.000031 }; mag = 0.000043
15: { -0.000031, -0.000031 }; mag = 0.000043
16: { 0.000000, -0.000031 }; mag = 0.000031
17: { 0.000000, 0.000000 }; mag = 0.000000
18: { 0.000000, 0.000000 }; mag = 0.000000
19: { 0.000000, 0.000000 }; mag = 0.000000
20: { 0.000000, -0.000031 }; mag = 0.000031
21: { 0.000000, -0.000031 }; mag = 0.000031
22: { 0.000000, 0.000000 }; mag = 0.000000
23: { 0.000000, 0.000000 }; mag = 0.000000
24: { 0.000000, 0.000000 }; mag = 0.000000
25: { 0.000000, 0.000000 }; mag = 0.000000
26: { 0.000000, 0.000000 }; mag = 0.000000
27: { 0.000000, 0.000000 }; mag = 0.000000
28: { 0.000000, 0.000000 }; mag = 0.000000
```

```

选择Vivado HLS 2015.4 Command Prompt

@I [RTGEN-100] Finished creating RTL model for 'hls_xfft2real_Loop_realfft_be_buffer_proc'.
@I [HLS-111] Elapsed time: 0.164 seconds; current memory usage: 796 MB.
@I [HLS-10] -----
@I [HLS-10] -- Generating RTL for module 'hls_xfft2real_Loop_realfft_be_descramble_pro'
@I [HLS-10] -----
@I [RTGEN-100] Generating core module 'hls_xfft2real_mac_muladd_16s_16s_31s_31_3' : 1 instance(s).
@I [RTGEN-100] Generating core module 'hls_xfft2real_mac_mulsub_16s_16s_31s_31_3' : 1 instance(s).
@I [RTGEN-100] Generating core module 'hls_xfft2real_mul_mul_16s_16s_31_3' : 2 instance(s).
@I [RTGEN-100] Finished creating RTL model for 'hls_xfft2real_Loop_realfft_be_descramble_pro'.
@I [HLS-111] Elapsed time: 0.227 seconds; current memory usage: 796 MB.
@I [HLS-10] -----
@I [HLS-10] -- Generating RTL for module 'hls_xfft2real'
@I [HLS-10] -----
@I [RTGEN-500] Setting interface mode on port 'hls_xfft2real/din_V_data' to 'axis'.
@I [RTGEN-500] Setting interface mode on port 'hls_xfft2real/din_V_last_V' to 'axis'.
@I [RTGEN-500] Setting interface mode on port 'hls_xfft2real/dout_V' to 'axis'.
@I [RTGEN-500] Setting interface mode on function 'hls_xfft2real' to 'ap_ctrl_hs'.
@I [RTGEN-100] Finished creating RTL model for 'hls_xfft2real'.
@I [HLS-111] Elapsed time: 0.274 seconds; current memory usage: 796 MB.
@I [RTMG-279] Implementing memory 'hls_xfft2real_Loop_realfft_be_descramble_pro_twid_rom_0_rom' using auto ROMs.
@I [RTMG-279] Implementing memory 'hls_xfft2real_Loop_realfft_be_descramble_pro_twid_rom_1_rom' using auto ROMs.
@I [RTMG-278] Implementing memory 'hls_xfft2real_descramble_buf_0_M_real_V_memcore_ram' using block RAMs.
@I [HLS-10] Finished generating all RTL models.
@I [WSYSC-301] Generating RTL SystemC for 'hls_xfft2real'.
@I [VWHDL-304] Generating RTL VHDL for 'hls_xfft2real'.
@I [VLOG-307] Generating RTL Verilog for 'hls_xfft2real'.
@I [IMPL-8] Exporting RTL as an IP in IP-XACT.

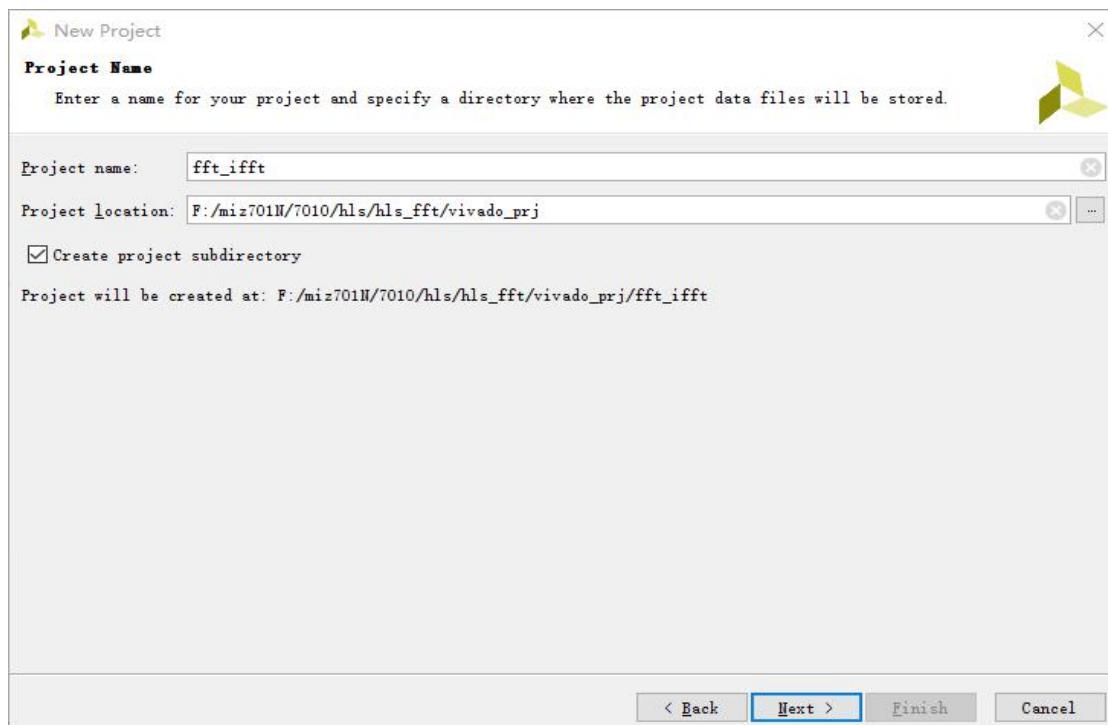
***** Vivado v2015.4 (64-bit)
***** SW Build 1412921 on Wed Nov 18 09:43:45 MST 2015
***** IP Build 1412160 on Tue Nov 17 13:47:24 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

source run_ipack.tcl -notrace
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository 'D:/Xilinx/Vivado/2015.4/data/ip'.
INFO: [Common 17-206] Exiting Vivado at Mon Mar 13 16:42:18 2017...
@I [HLS-112] Total elapsed time: 233.481 seconds; peak memory usage: 796 MB.
F:\miz701N\7010\hls\hls_fft\tcl>

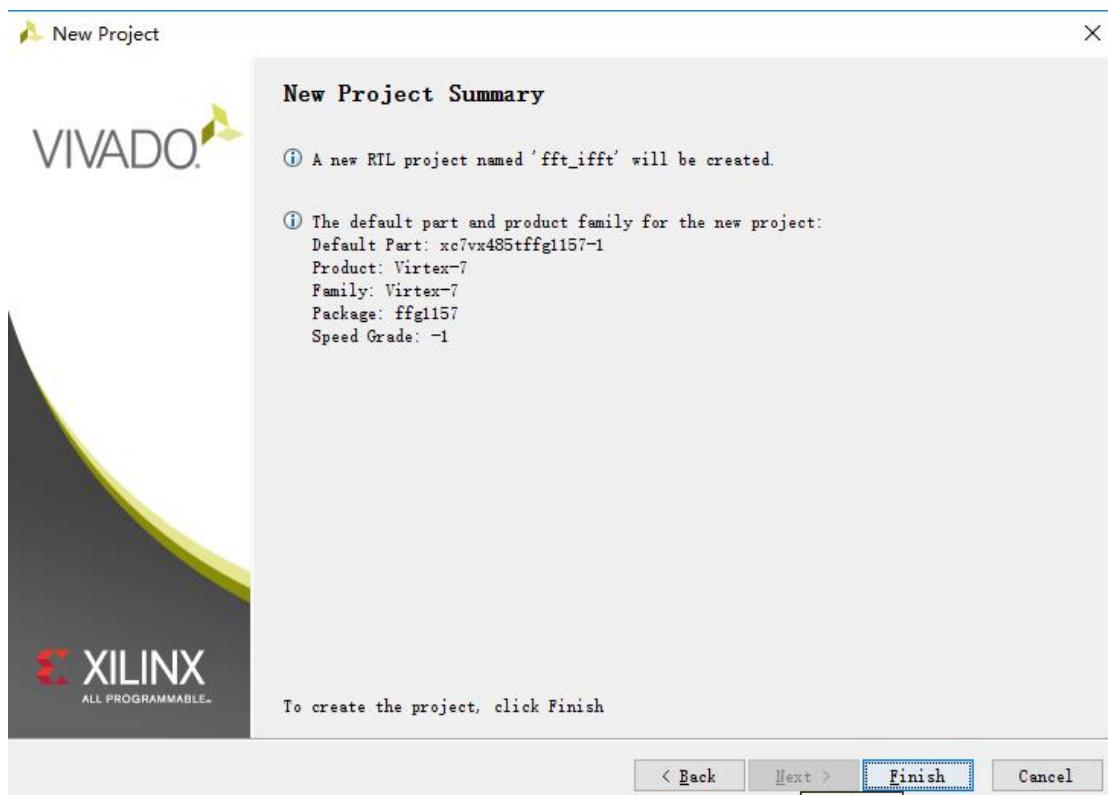
```

8.3 Vivado 模块例化及 IP 封包

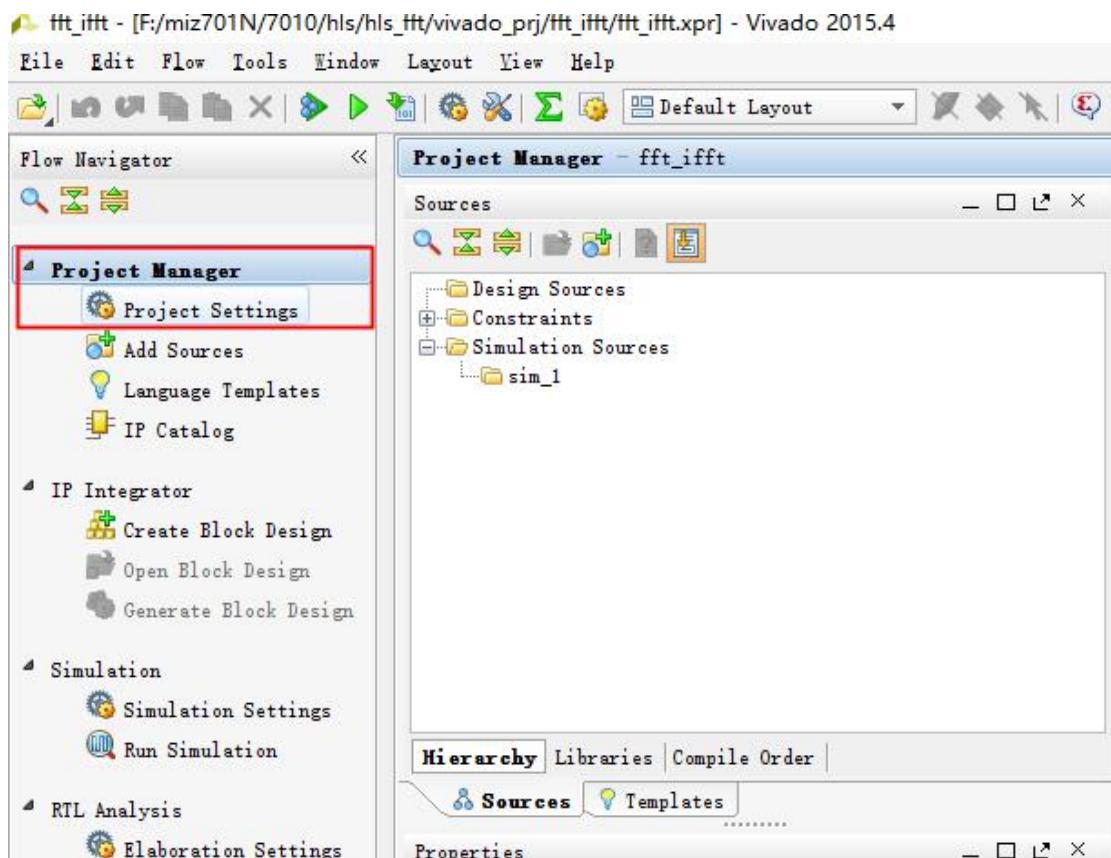
Step1:HLS 生成了 IP 之后, 此时还不能直接拿来使用, 还需要对其进行一些必要的配置, 然后将上一节生成的两个 IP 封装成一个 IP 使用。首先我们创建一个名为 fft_ifft 的工程。



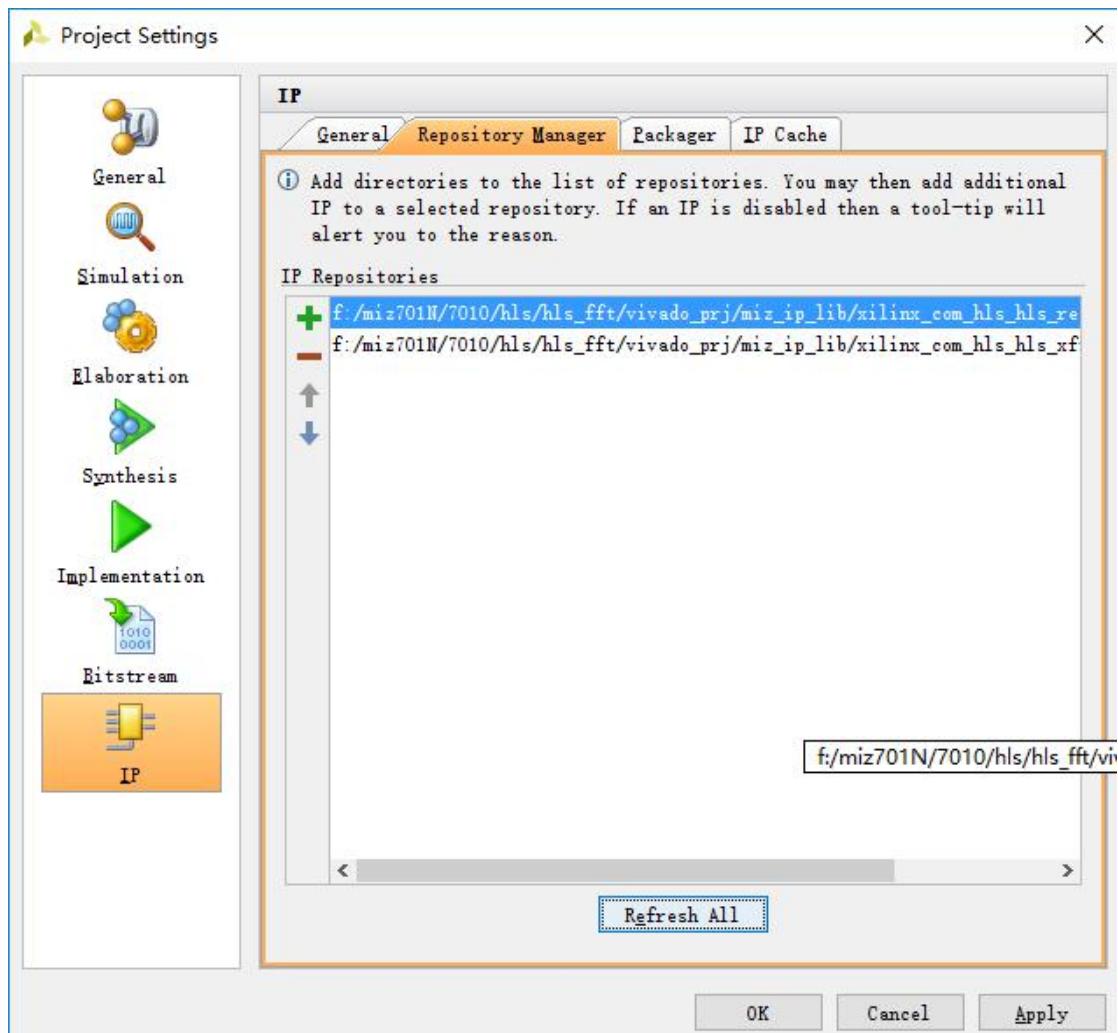
Step2: 一直 next 按照默认设置配置下去，在芯片设置步骤时根据自身芯片类型选择，最后点击 Finish 完成工程的创建。



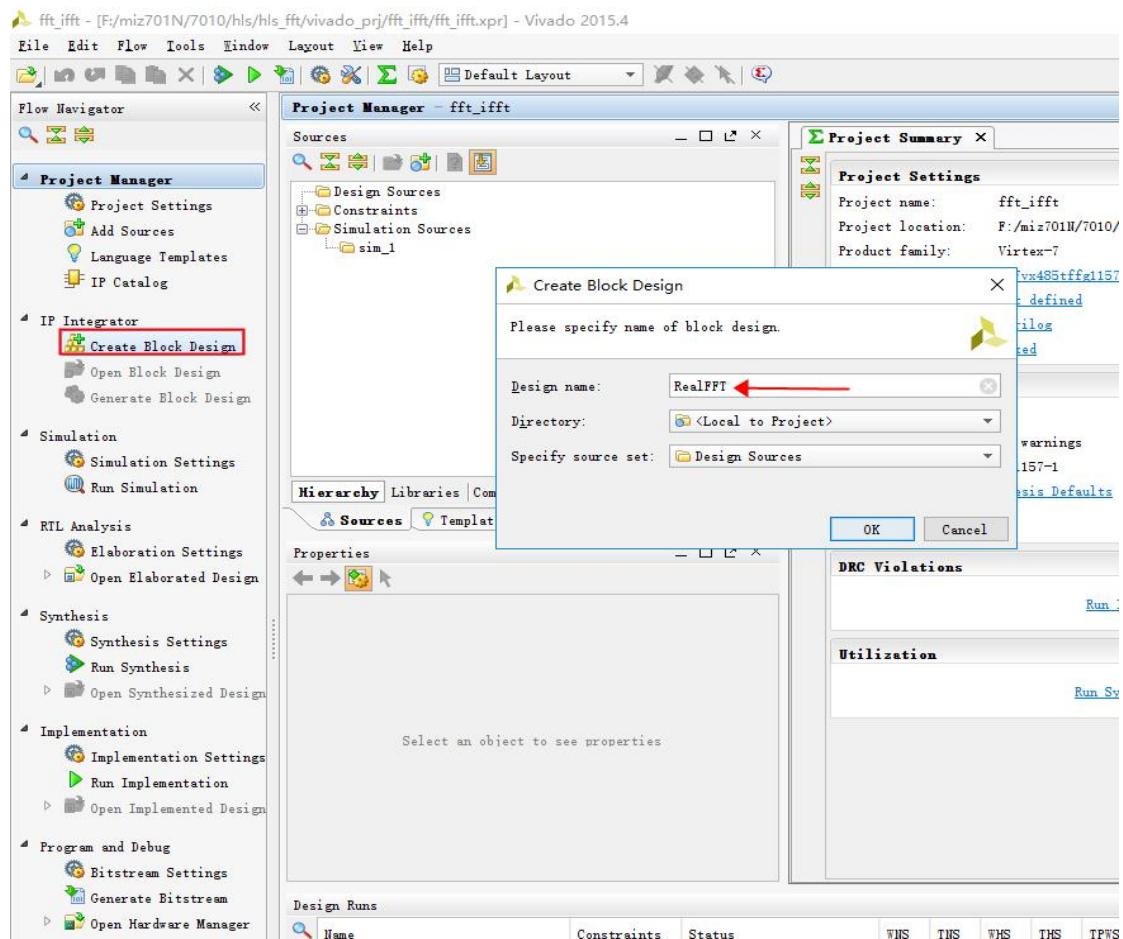
Step3: 工程创建好以后，我们需要把生成的 HLS IP 添加进当前的工程项目当中，单击 project manager 中的 project settings。



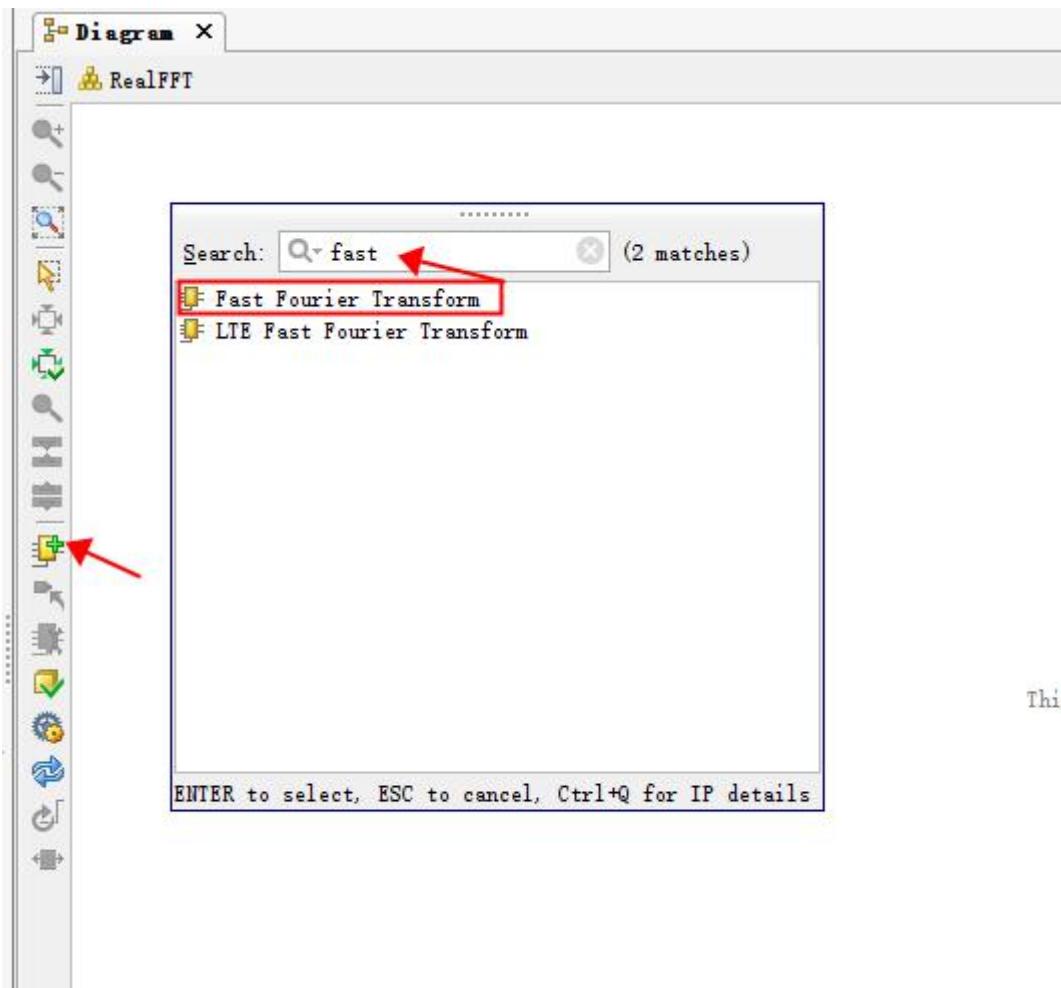
Step4：选择 IP 选项，再选择 Repository Manager 选项，单击加号图标将上一节生成的 IP 添加到工程当中，最后单击 OK。



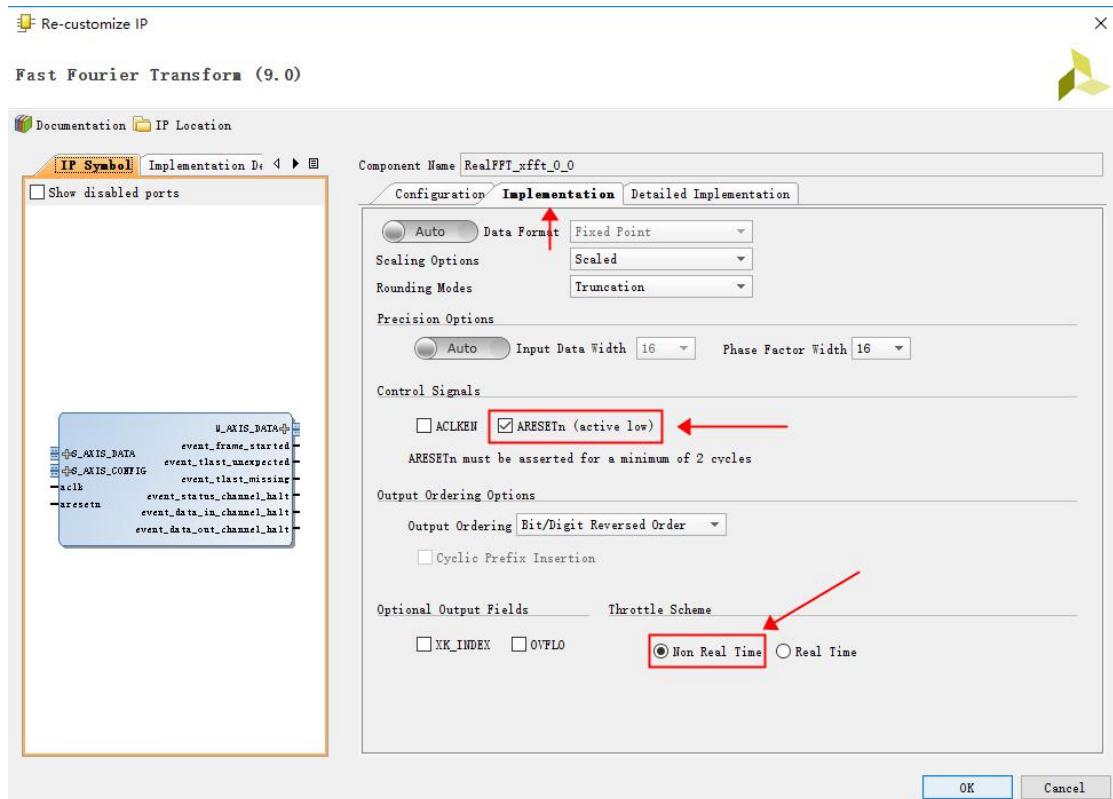
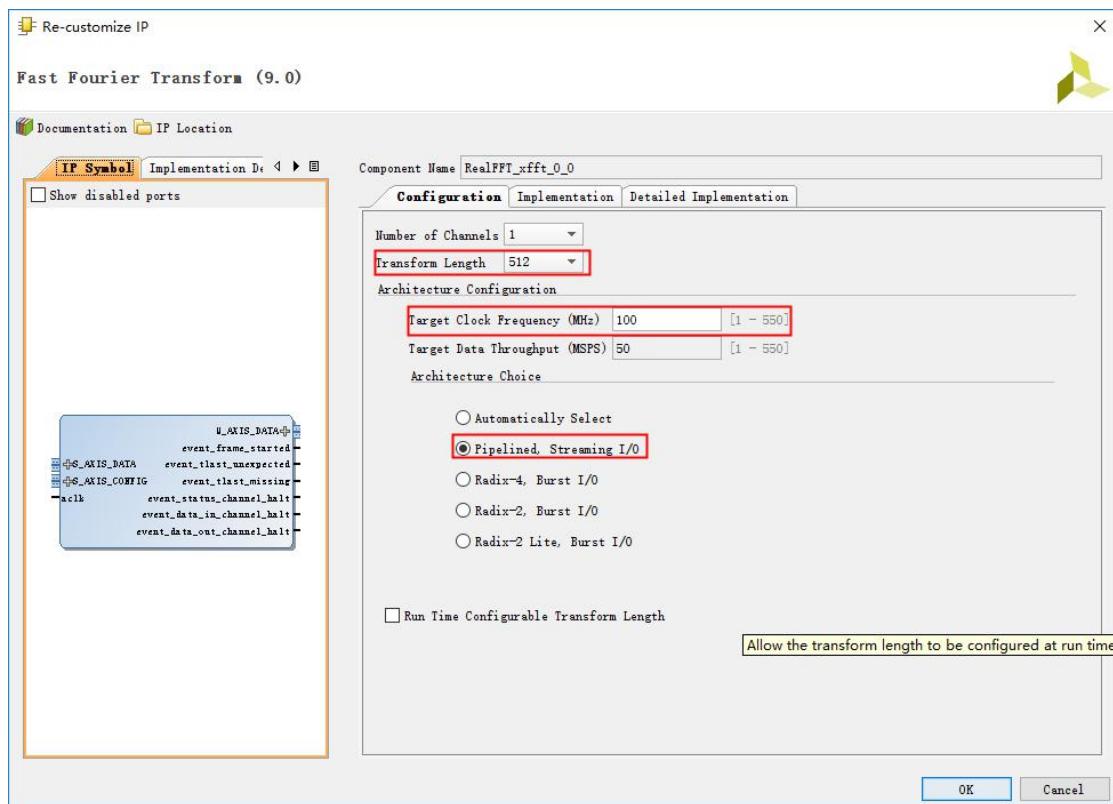
Step5: 单击 Create Block design, 创建一个名为 RealFFT 的 BD 文件。



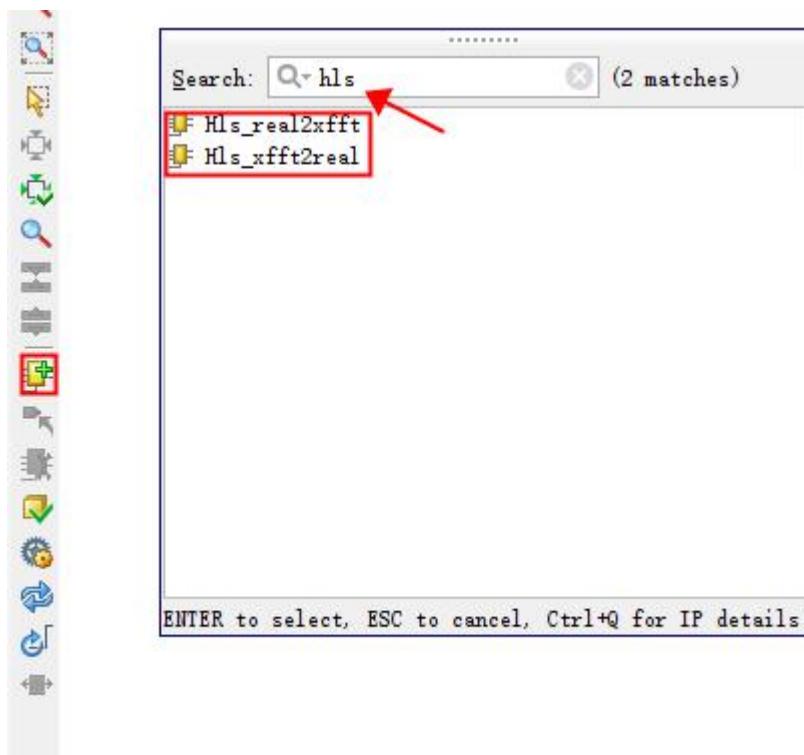
Step6: 单击 添加一个名为 Fast Fourier Transfer 的 IP。



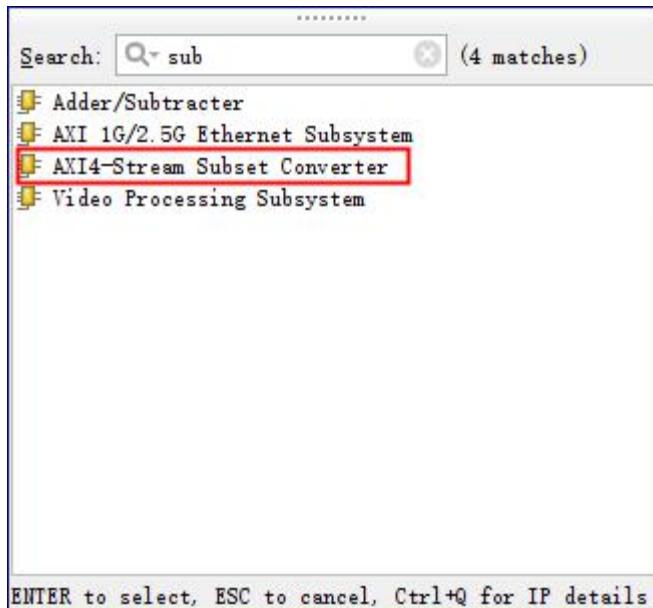
Step7: 双击这个IP，如下图所示配置此IP，然后单击OK。



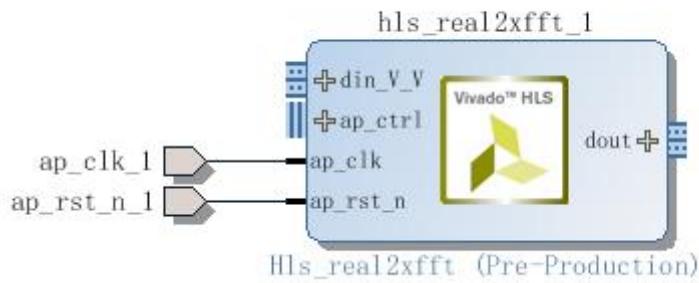
Step8: 单击 将上一节生成的两个 IP 添加到 BD 文件中来。



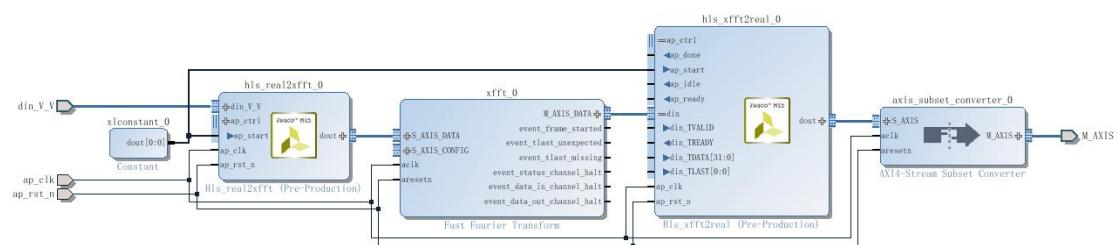
Step9: 添加一个 AXI4-Stream Subset Converter。



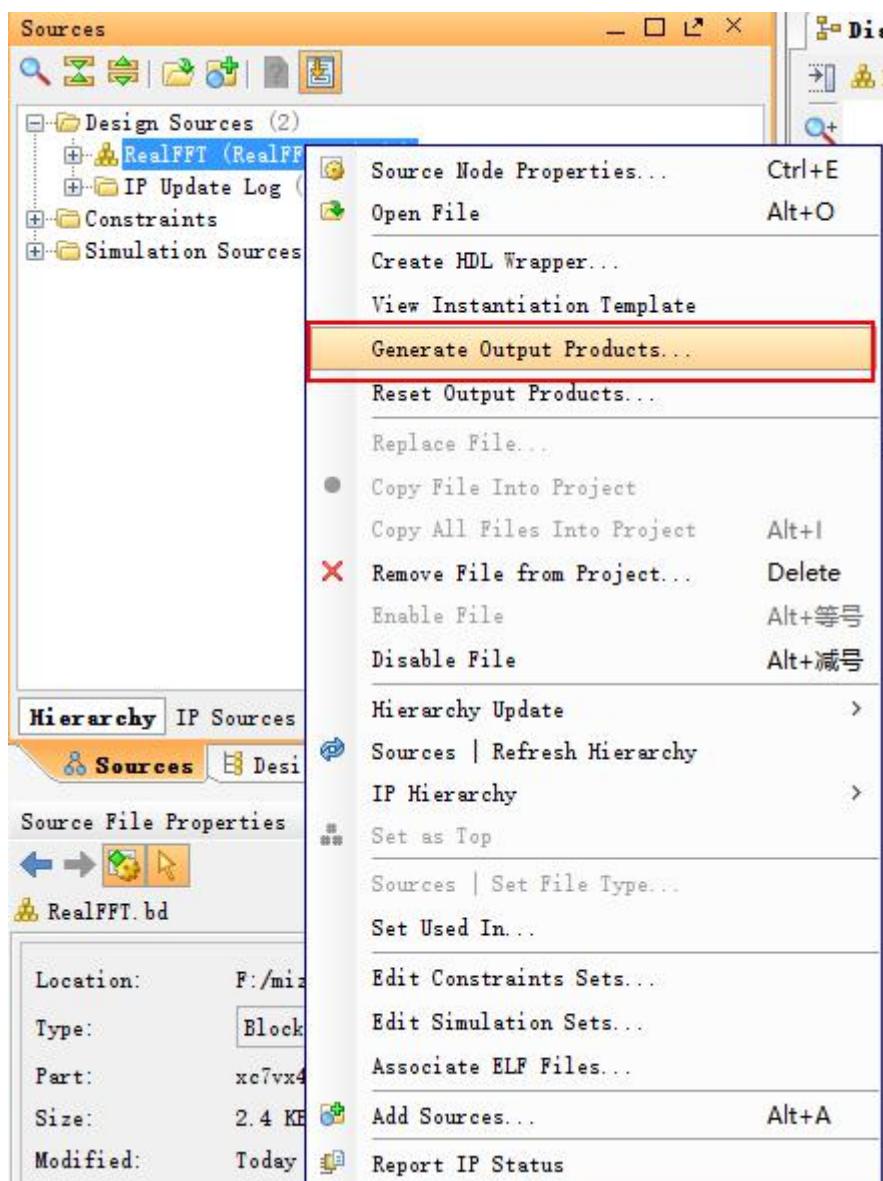
Step10: 为 ap_clk, ap_rst_n 引出一个端口, 如下图所示。



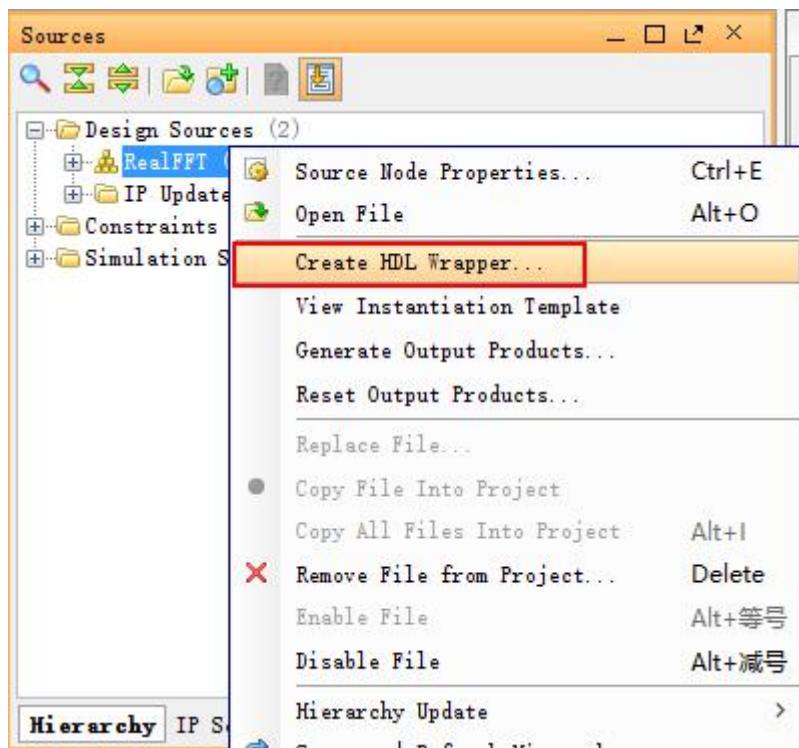
Step11: 单击 再添加一个 constant IP, 然后按下图完善硬件电路。



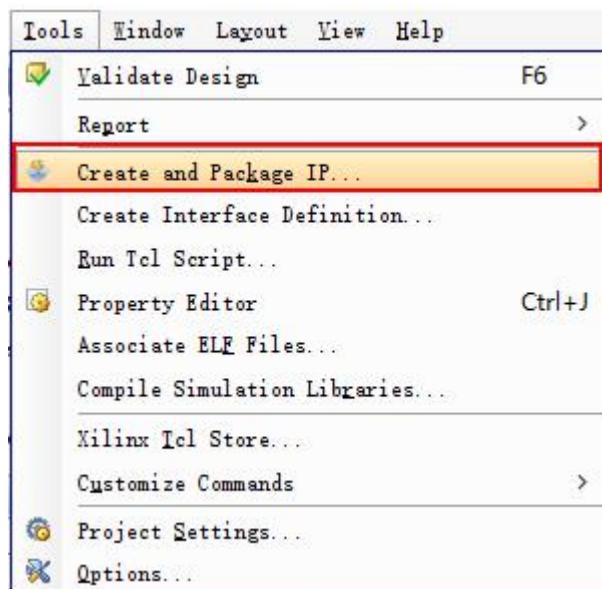
Step12: 选中 BD 文件, 右单击选择 Generate Output Products。



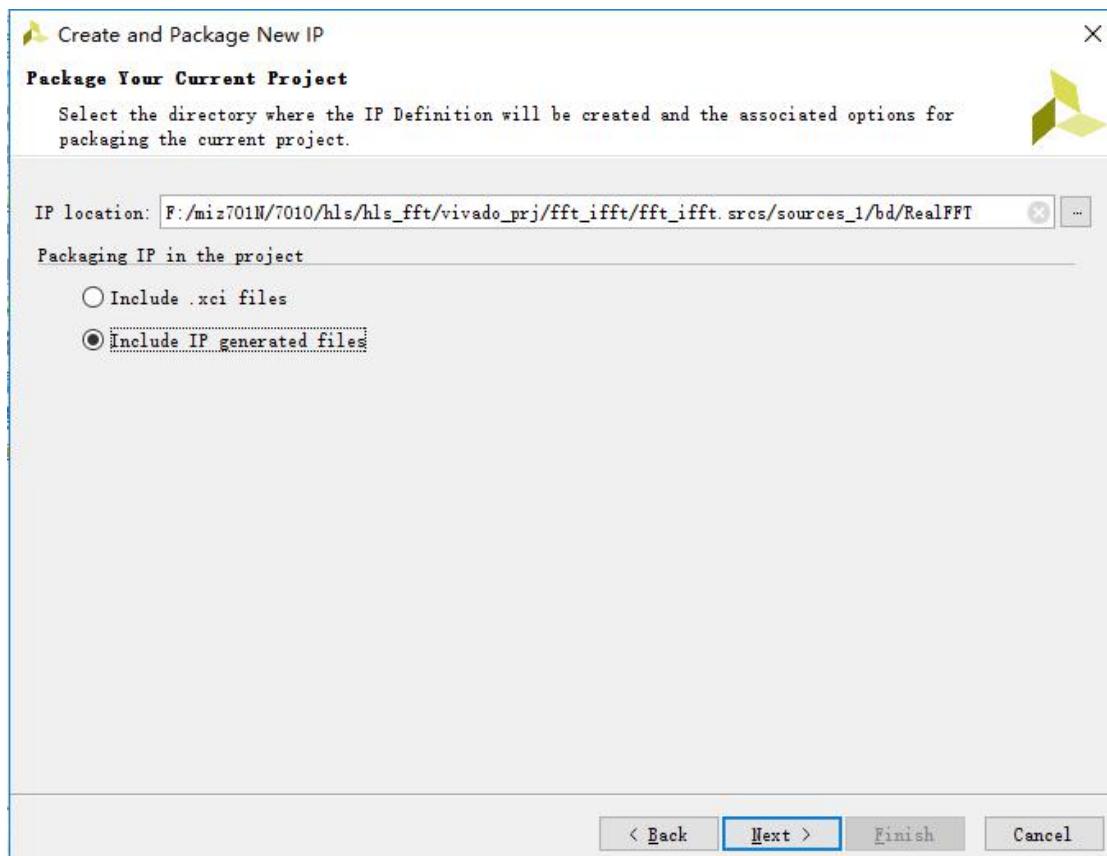
Step13: 单击 Create HDL Wrapper 生成顶层文件。



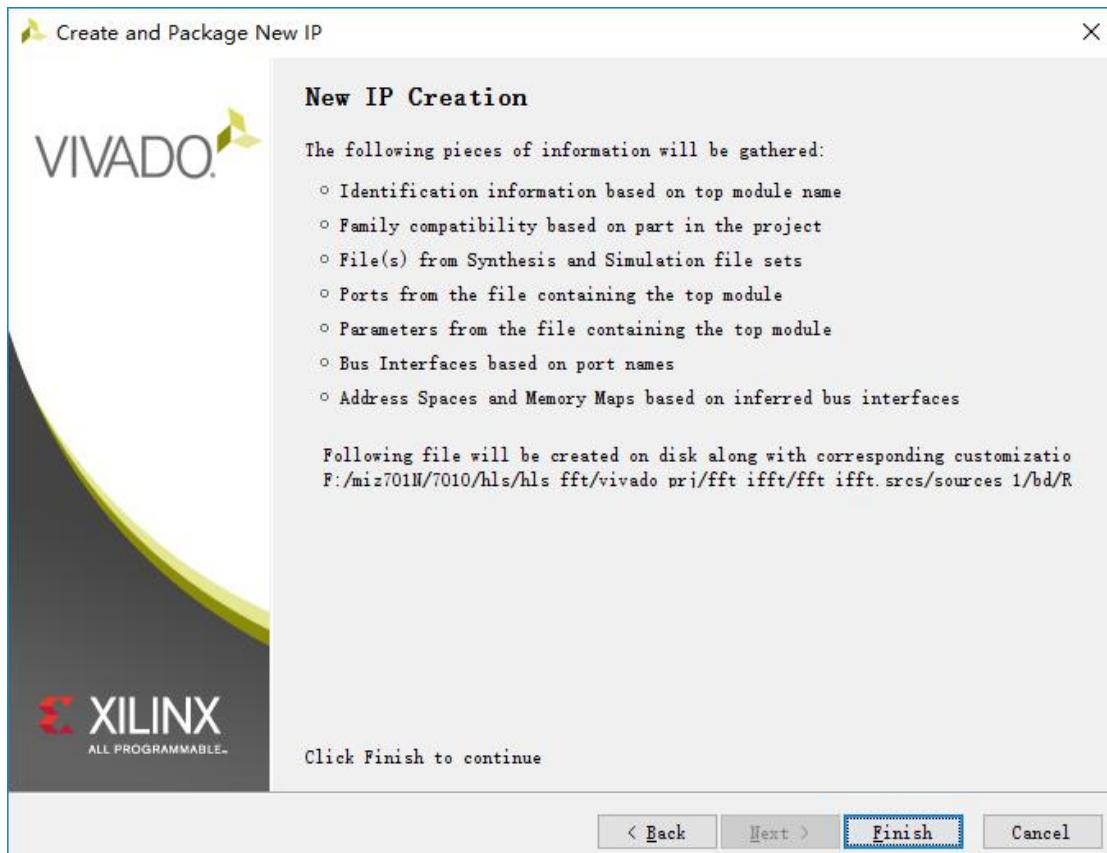
Step14: 接下来将其打包成一个 IP，单击 Tools 菜单下的 Create and package IP.. 命令。



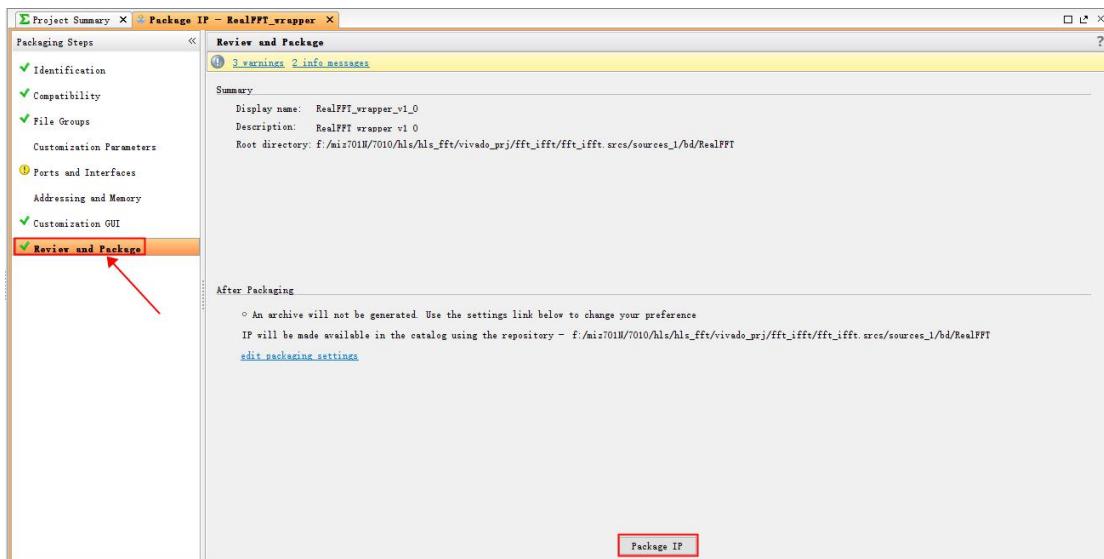
Step15: 单击 Next, 直到出现下图界面，按图中的设置方法设置。



Step16：单击 next，最后单击 Finish。



Step17：在跳出来的页面当中，直接选择 Review and Package，然后点击 Package IP 既完成了 IP 的封装。



8.4 本章小结

本章向大家介绍了 FFT 的原理，然后使用官方给出的代码使用 HLS 生成了两个 FFT 的 IP，之后在 VIVADO 中将其封装成了一个可供之后章节使用的 IP。本章需要重点掌握的是 FFT 的原理和熟练的掌握 HLS 的开发流程。本章参考了以下文献：

<http://www.xuebuyuan.com/539160.html>

<http://blog.csdn.net/sshcx/article/details/1651616>

S05_CH09_OTSU 自适应二值化

9.1 OTSU 自适应二值化原理简介

最大类间方差法是由日本学者大津于 1979 年提出的, 是一种自适应的阈值确定的方法, 又叫大津法, 简称 OTSU。它是按图像的灰度特性, 将图像分成背景和目标 2 部分。背景和目标之间的类间方差 σ^2 越大, 说明构成图像的 2 部分的差别越大, 当部分目标错分为背景或部分背景错分为目标都会导致 2 部分差别变小。因此, 使类间方差最大的分割意味着错分概率最小。对于图像 $I(x, y)$, 前景(即目标)和背景的分割阈值记作 T , 属于前景的像素点数占整幅图像的比例记为 ω_0 , 其平均灰度 μ_0 ; 背景像素点数占整幅图像的比例为 ω_1 , 其平均灰度为 μ_1 。图像的总平均灰度记为 μ , 类间方差记为 σ^2 。假设图像的背景较暗, 并且图像的大小为 $M \times N$, 图像中像素的灰度值小于阈值 T 的像素个数记作 N_0 , 像素灰度大于阈值 T 的像素个数记作 N_1 , 则有:

$$\omega_0 = N_0 / M \times N \quad (1)$$

$$\omega_1 = N_1 / M \times N \quad (2)$$

$$N_0 + N_1 = M \times N \quad (3)$$

$$\omega_0 + \omega_1 = 1 \quad (4)$$

$$\mu = \omega_0 * \mu_0 + \omega_1 * \mu_1 \quad (5)$$

$$\sigma^2 = \omega_0 (\mu_0 - \mu)^2 + \omega_1 (\mu_1 - \mu)^2 \quad (6)$$

将式(5)代入式(6), 得到等价公式: $\sigma^2 = \omega_0 \omega_1 (\mu_0 - \mu_1)^2 \quad (7)$

采用遍历的方法得到使类间方差最大的阈值 T , 即为所求。

Otsu 算法步骤如下:

设图象包含 L 个灰度级 ($0, 1 \dots, L-1$), 灰度值为 i 的象素点数为 N_i , 图象总的象素点数为 $N = N_0 + N_1 + \dots + N_{L-1}$ 。灰度值为 i 的点的概率为:

$$P(i) = N(i) / N.$$

门限 t 将整幅图象分为暗区 c_1 和亮区 c_2 两类, 则类间方差 σ^2 是 t 的函数:

$$\sigma^2 = a_1 * a_2 * (u_1 - u_2)^2 \quad (2)$$

式中, a_j 为类 c_j 的面积与图象总面积之比, $a_1 = \sum(P(i)) \quad i \rightarrow t$, $a_2 = 1 - a_1$; u_j 为类 c_j 的均值, $u_1 = \sum(i * P(i)) / a_1 \quad 0 \rightarrow t$,

$$u_2 = \sum(i * P(i)) / a_2, \quad t+1 \rightarrow L-1$$

该法选择最佳门限 t^* 使类间方差最大, 即: 令 $\Delta u = u_1 - u_2$, $\sigma_b = \max\{a_1(t) * a_2(t) \Delta u^2\}$

9.2 HLS 实现

9.2.1 工程创建

Step1: 打开 HLS, 按照之前介绍的方法, 创建一个新的工程, 命名为 otsu_threshold。

Step2: 右单击 Source 选项, 选择 New File, 创建一个名为 Top.cpp 的文件。

Step3:在打开的编辑区中，把下面的程序拷贝进去：

```
#include "top.h"

//http://blog.csdn.net/taoyanbian1022/article/details/9030825
//参考网址

void hls::adaptive_threshold(GRAY_IMAGE &src_img,GRAY_IMAGE
&dst_img,int12_t rows, int12_t cols)
{
    //统计不同灰度值个数寄存器
    int pixelCount[256];
    float pixelPro[256];
    LOOP_CLEAR:for(int i = 0;i < 256; i++)
    {
        pixelCount[i] = 0;
        pixelPro[i] = 0;
    }

    //统计灰度级中每个像素在整幅图像中的个数
    LOOP_ROWS:for(int12_t row = 0; row < rows; row++)
    {
        LOOP_COLS:for(int12_t col = 0; col < cols; col++)
        {
            GRAY_PIXEL src_data;
            GRAY_PIXEL dst_data;
            src_img >> src_data;
            pixelCount[(uchar) (src_data.val[0])]++; //轮询数据存到对应的寄存器
            dst_data.val[0] = (uchar) (src_data.val[0]);
            src_img << dst_data;
        }
    }

    int pixelSum = rows * cols;
    uchar threshold = 0;
    for(int i = 0; i < 256; i++)
    {
        pixelPro[i] = (float) (pixelCount[i]) / (float) (pixelSum);
    }

    //经典ostu算法,得到前景和背景的分割
    //遍历灰度级[0,255],计算出方差最大的灰度值,为最佳阈值
```

```
float w0, w1, u0tmp, u1tmp, u0, u1, u, deltaTmp, deltaMax = 0;
for(int i = 0; i < 256; i++)
{
#pragma HLS PIPELINE II=1 off

w0 = w1 = u0tmp = u1tmp = u0 = u1 = u = deltaTmp = 0;

for(int j = 0; j < 256; j++)
{
    if(j <= i) //背景部分,以i为阈值分类,第一类总的概率
    {
        w0 += pixelPro[j];
        u0tmp += j * pixelPro[j];
    }
    else//前景部分,以i为阈值分类,第二类总的概率
    {
        w1 += pixelPro[j];
        u1tmp += j * pixelPro[j];
    }
}

u0 = u0tmp / w0;           //第一类的平均灰度
u1 = u1tmp / w1;           //第二类的平均灰度
u = u0tmp + u1tmp;         //整幅图像的平均灰度
//计算类间方差
deltaTmp = w0 * (u0 - u)*(u0 - u) + w1 * (u1 - u)*(u1 - u);
//找出最大类间方差以及对应的阈值
if(deltaTmp > deltaMax)
{
    deltaMax = deltaTmp;
    threshold = i;
}
}

hls::Threshold(src_img,dst_img,threshold,255,HLS_THRESH_BINARY);
//使用HLS视频库进行阈值变换
}

void hls_counter_color(AXI_STREAM_IN& INPUT_STREAM,
AXI_STREAM_OUT& OUTPUT_STREAM, int12_t rows, int12_t cols)
{
//Create AXI streaming interfaces for the core
```

```

#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM

#pragma HLS RESOURCE core=AXI_SLAVE variable=rows
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cols
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=return
metadata="-bus_bundle CONTROL_BUS"

#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols

RGB_IMAGE img_0(rows, cols);
GRAY_IMAGE img_1(rows, cols);
GRAY_IMAGE img_2(rows, cols);

// #pragma HLS dataflow
    hls:::AXIVideo2Mat(INPUT_STREAM, img_0);
    hls:::CvtColor<HLS_RGB2GRAY>(img_0, img_1); // 将RGB图像转换为灰度
    图像
    hls:::adaptive_threshold(img_1, img_2, rows, cols); // 自适应阈值变换
    hls:::Mat2AXIVideo(img_2, OUTPUT_STREAM);
}

```

Step4: 再在 Source 中添加一个名为 Top.h 的库函数，并添加如下程序：

```

#ifndef _TOP_H_
#define _TOP_H_

#include "hls_video.h"
#include "ap_int.h"

// maximum image size
#define MAX_WIDTH 800
#define MAX_HEIGHT 600

// I/O Image Settings
#define INPUT_IMAGE      "car.bmp"
#define OUTPUT_IMAGE     "result_1080p.bmp"
#define OUTPUT_IMAGE_GOLDEN "result_1080p_golden.bmp"

// typedef video library core structures
typedef hls:::stream<ap_axiu<24,1,1,1> >

```

```

AXI_STREAM_IN;
typedef hls::stream<ap_axiu< 8,1,1,1> >
AXI_STREAM_OUT;
typedef hls::Scalar<3, unsigned char> RGB_PIXEL;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> RGB_IMAGE;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1> GRAY_IMAGE;
typedef hls::Scalar<1, unsigned char> GRAY_PIXEL;

typedef unsigned char uchar;
typedef ap_uint<12> int12_t;//自定义12位无符号整型

//顶层综合函数
void hls_counter_color(AXI_STREAM_IN& src_axi, AXI_STREAM_OUT&
dst_axi, int12_t rows, int12_t cols);

//自定义处理函数
namespace hls
{
    void adaptive_threshold(GRAY_IMAGE &src_img,GRAY_IMAGE
&dst_img,int12_t rows, int12_t cols);
    void reduce_distraction(GRAY_IMAGE &src_img,GRAY_IMAGE
&dst_img,int12_t rows, int12_t cols);
    void median(GRAY_IMAGE &src_img,GRAY_IMAGE &dst_img,int12_t
rows, int12_t cols);
}

#endif

```

Step5: 在 Test Bench 中, 用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码:

```

#include "top.h"
#include "opencv_top.h"

using namespace std;
using namespace cv;

int main (int argc, char** argv)
{
    //获取图像数据
    IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
1);//src->nChannels

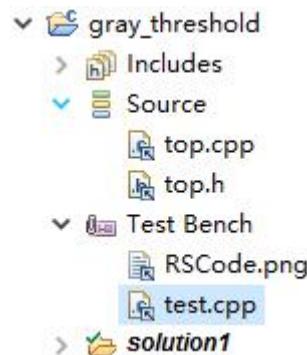
    //使用HLS库进行处理

```

```
AXI_STREAM_IN src_axi;
AXI_STREAM_OUT dst_axi;
IplImage2AXIVideo(src, src_axi);
hls_counter_color(src_axi, dst_axi, src->height, src->width);
AXIVideo2IplImage(dst_axi, dst);
cvSaveImage(OUTPUT_IMAGE, dst);
cvShowImage("hls_dst", dst);
waitKey(0);

//释放内存
cvReleaseImage(&src);
cvReleaseImage(&dst);
}
```

Step6: 在 Test Bench 中添加一张名为 car.bmp 的测试图片, 图片可以在我们提供的源程序中的 Image 文件夹中找到。完整的工程如下图所示:



9.2.2 仿真及优化

9.3 硬件工程实现 (待更新)

9.4 程序分析

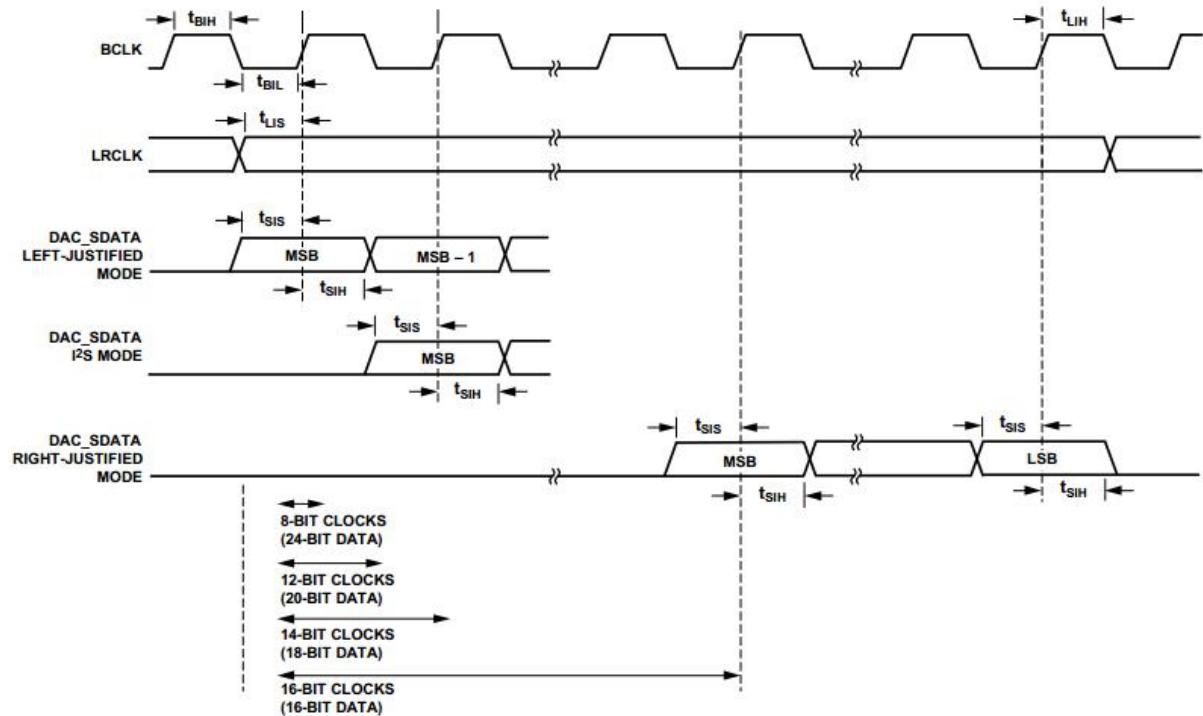
S05_CH10_音频滤波

10.1 ADAU1761 简介

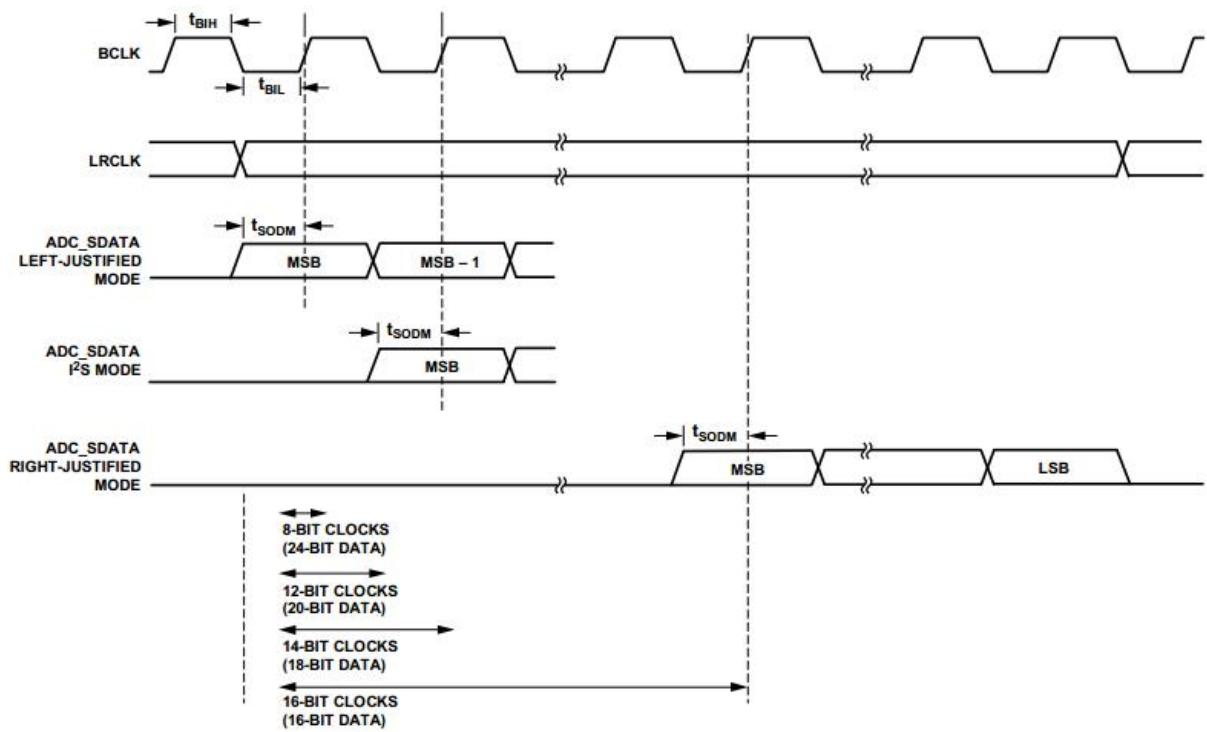
ADI 公司的 ADAU1761 是低功耗集成了数字音频处理的立体声 CODEC, 支持立体声 48kHz 录音, 1.8V 播放时的功耗为 14mW. 立体声 ADC 和 DAC 支持取样速率从 8kHz 到 96kHz 以及数字音量控制. 音频处理 SigmaDSP 核具有 28 位处理能力, 能使设计者补偿麦克风, 扬声器, 放大器和听觉环境中现实世界的限制, 通过均衡, 多频带压缩, 限幅和第三方算法来极大地改善音频质量. 主要应用在智能手机/多媒体手机, 数码相机/数码摄像机, 手提媒体播放器/手提音频播放器等.

10.1.1 ADAU1761 收发时序

音频输入采样时序:



音频输出时序:



10.1.2 ADAU1761 时钟和采样率

ADAU1761 内部有两个时钟控制寄存器 R0 和 R1 来控制其时钟和采样率的大小。ADAU1761 时钟树原理图如下图所示：

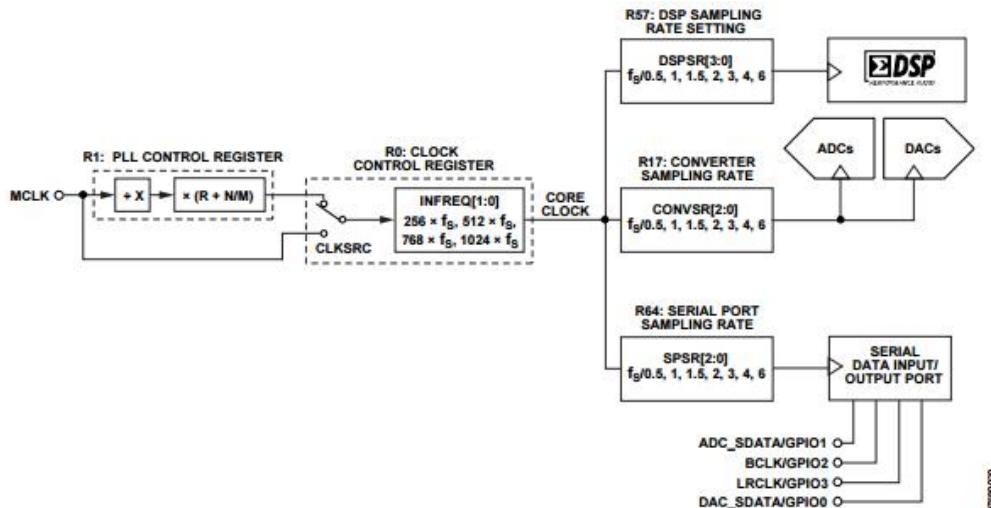


Figure 30. Clock Tree Diagram

在上图中 core clock 就是我们端口，转换器和 DSP 的时钟源。通过控制 R0 和 R1 可以选择 core clock 是通过 p11 生成还是直接通过 MCLK 得到。R0 和 R1 的寄存器如下图所示：

Table 12. Clock Control Register (Register R0, Address 0x4000)

Bits	Bit Name	Settings
3	CLKSRC	0: Direct from MCLK pin (default) 1: PLL clock
[2:1]	INFREQ[1:0]	00: $256 \times f_s$ (default) 01: $512 \times f_s$ 10: $768 \times f_s$ 11: $1024 \times f_s$
0	COREN	0: Core clock disabled (default) 1: Core clock enabled

Table 15. PLL Control Register (Register R1, Address 0x4002)

Bits	Bit Name	Description
[47:32]	M[15:0]	Denominator of the fractional PLL: 16-bit binary number 0x00FD: M = 253 (default)
[31:16]	N[15:0]	Numerator of the fractional PLL: 16-bit binary number 0x000C: N = 12 (default)
[14:11]	R[3:0]	Integer part of PLL: four bits, only values 2 to 8 are valid 0010: R = 2 (default) 0011: R = 3 0100: R = 4 0101: R = 5 0110: R = 6 0111: R = 7 1000: R = 8

Rev. C | Page 27 of 92

ADAU1761

Bits	Bit Name	Description
[10:9]	X[1:0]	PLL input clock divider 00: X = 1 (default) 01: X = 2 10: X = 3 11: X = 4
8	Type	PLL operation mode 0: Integer (default) 1: Fractional
1	Lock	PLL lock (read-only bit) 0: PLL unlocked (default) 1: PLL locked
0	PLLEN	PLL enable 0: PLL disabled (default) 1: PLL enabled

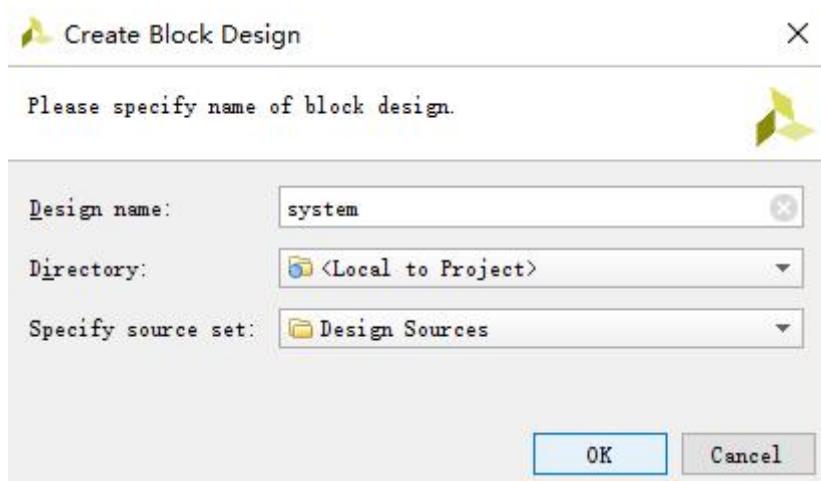
通过这两个寄存器的设置，我们就可以知道 ADAU1761 的采样率和时钟情况，具体的分析我们放在程序部分讲解。

10.2 硬件平台的搭建

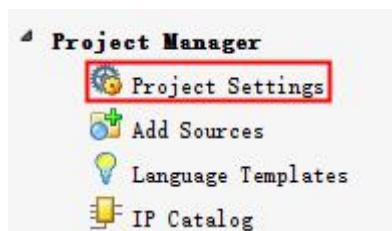
这一章可以算是对前面的实验的一个综合，涉及到 Audio 控制、OLED 显示控制以及 HLS 生成的 IP 核的使用，操作步骤如下：

Step1: 打开 VIVADO 软件, 创建一个名为 miz_sys 的工程, 根据自身的硬件平台正确配置芯片型号。

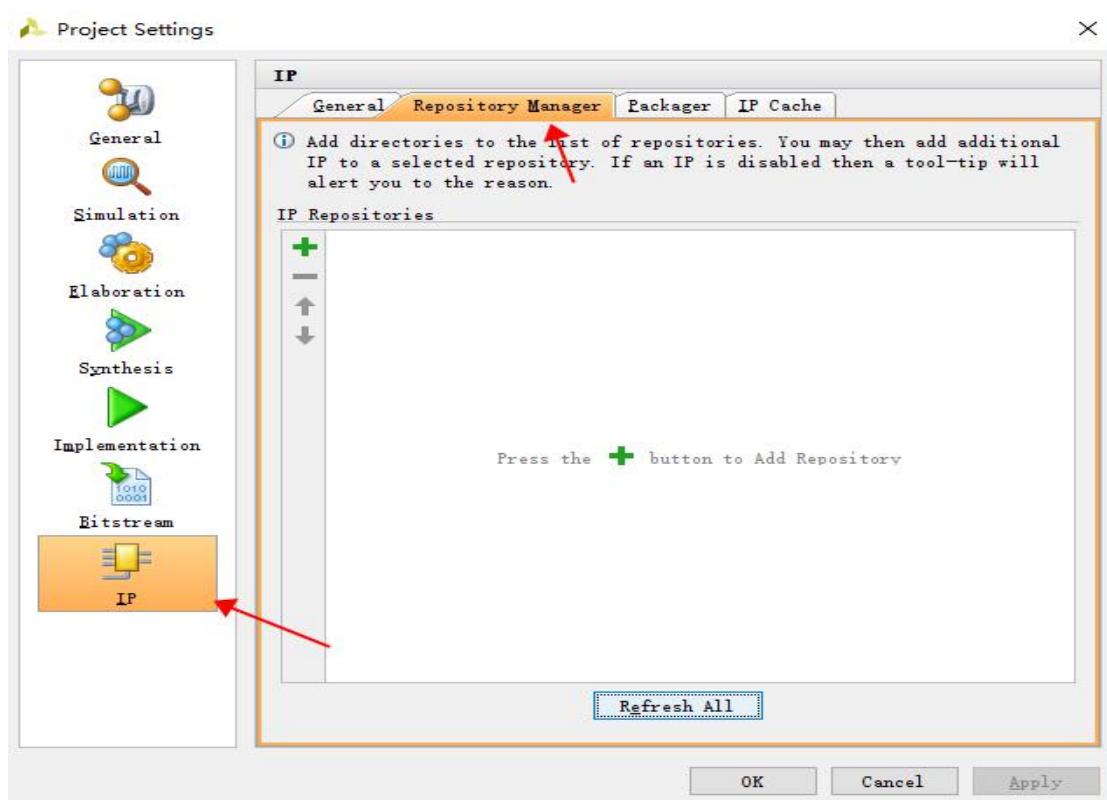
Step2: 双击 Create Block Design, 创建一个名为 system 的 BD 文件。



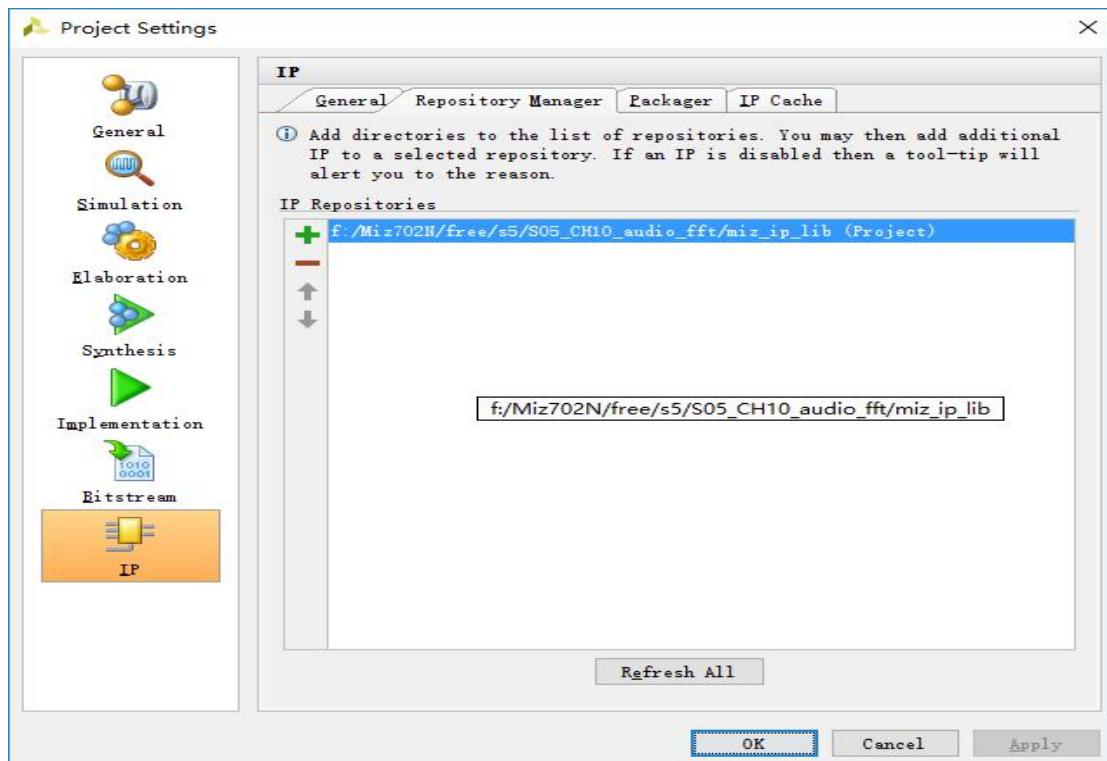
Step3: 在 Project Manager 设置区单击 Project settings。



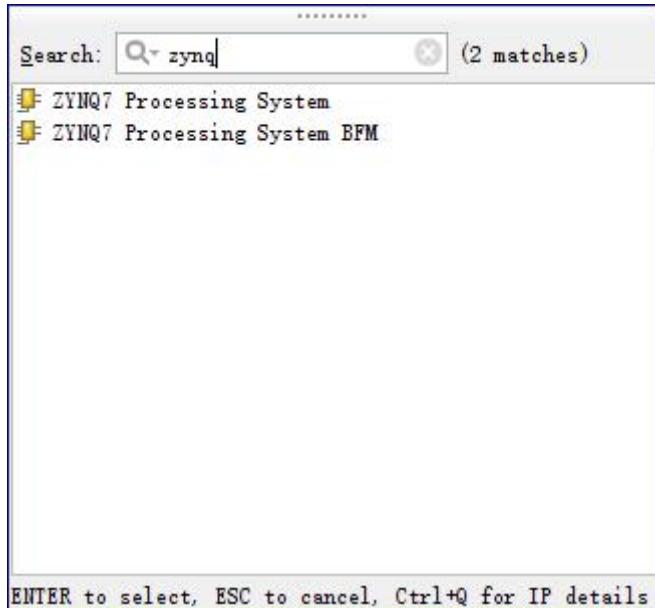
Step4: 在弹出的窗口中, 如下图设置:



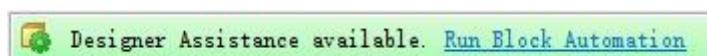
Step5：在我们提供的源程序对应章节的文件夹中找到 miz_ip_lib 文件夹，此文件夹存放了我们要用到的 IP，这其中就包括了我们第八章封装好了的 FFT IP 和我们之前用到过的 OLED IP，单击+图标，将 miz_ip_lib 文件夹添加到 IP 库当中，系统会自动扫描其中的 IP，之后单击 OK 完成修改。



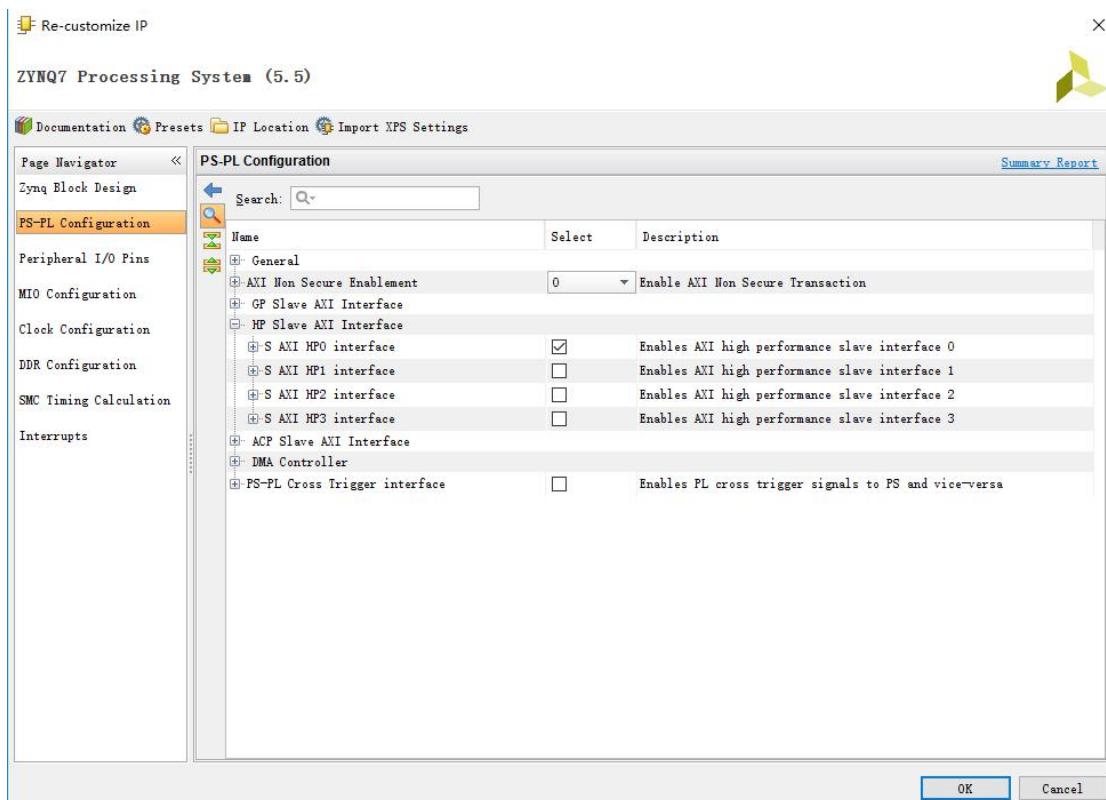
Step6: 单击  图标添加一个 ZYNQ Processing System IP。

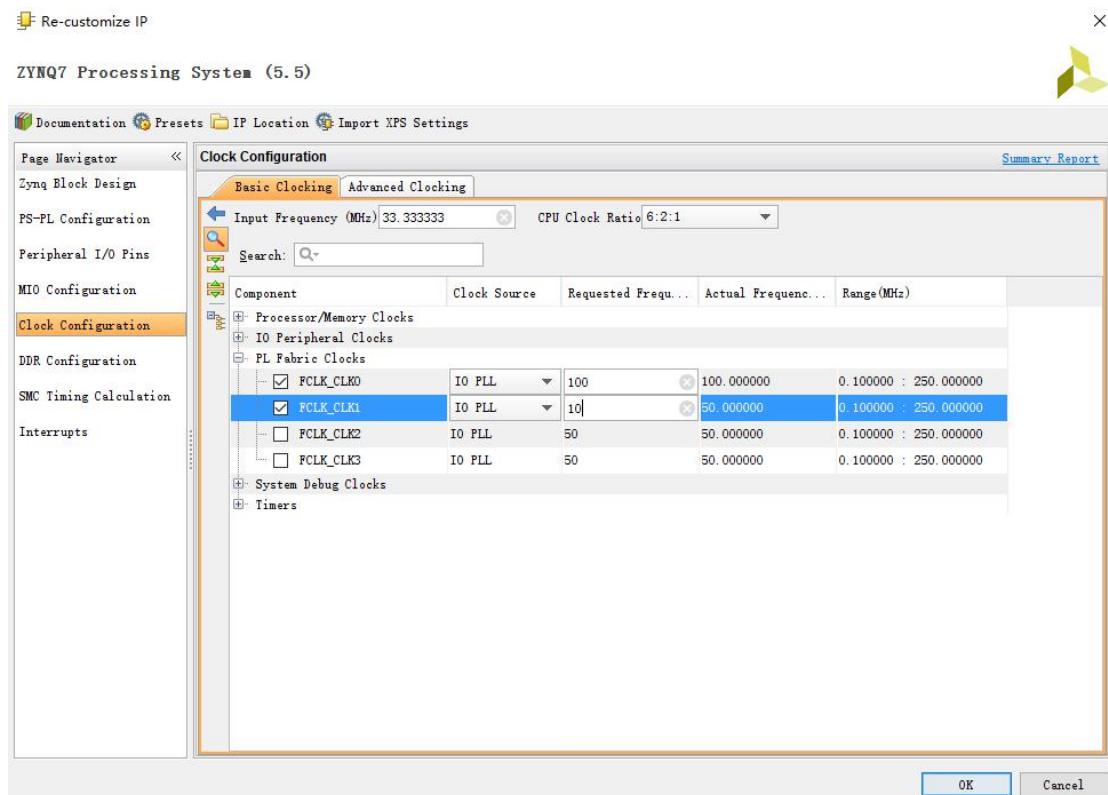
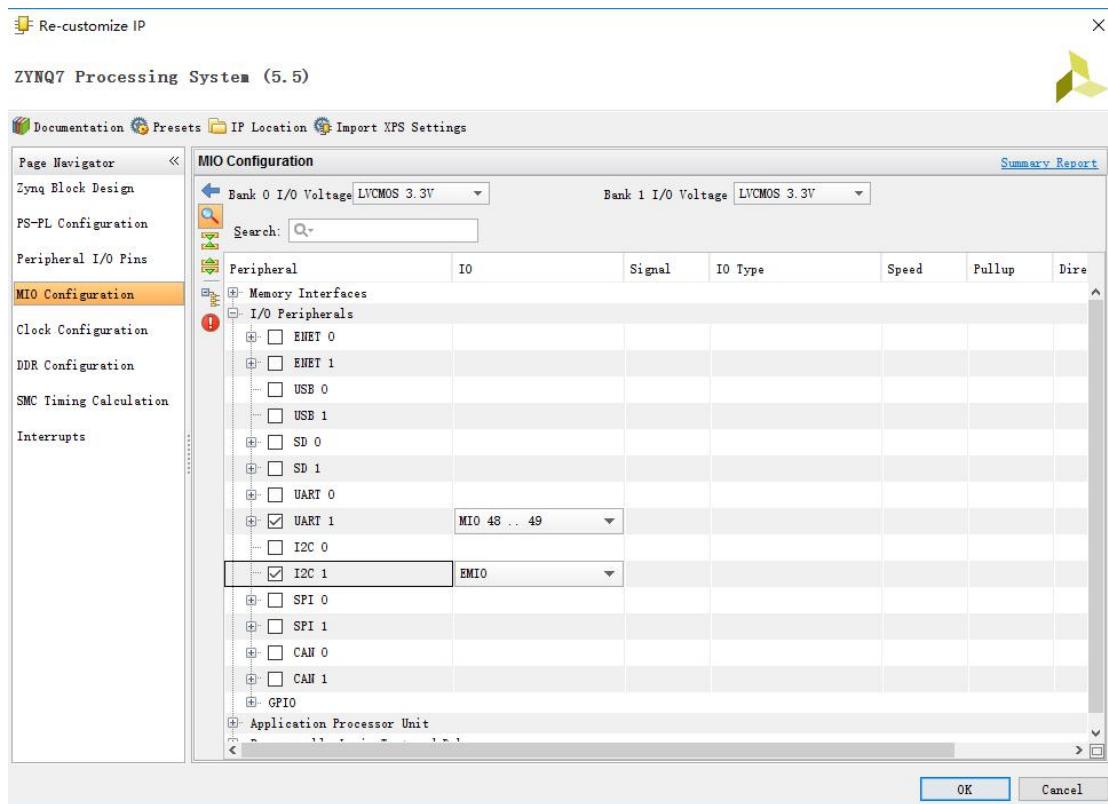


Step7: 单击 Run Block Automation, 在弹出来的窗口中直接单击 OK。

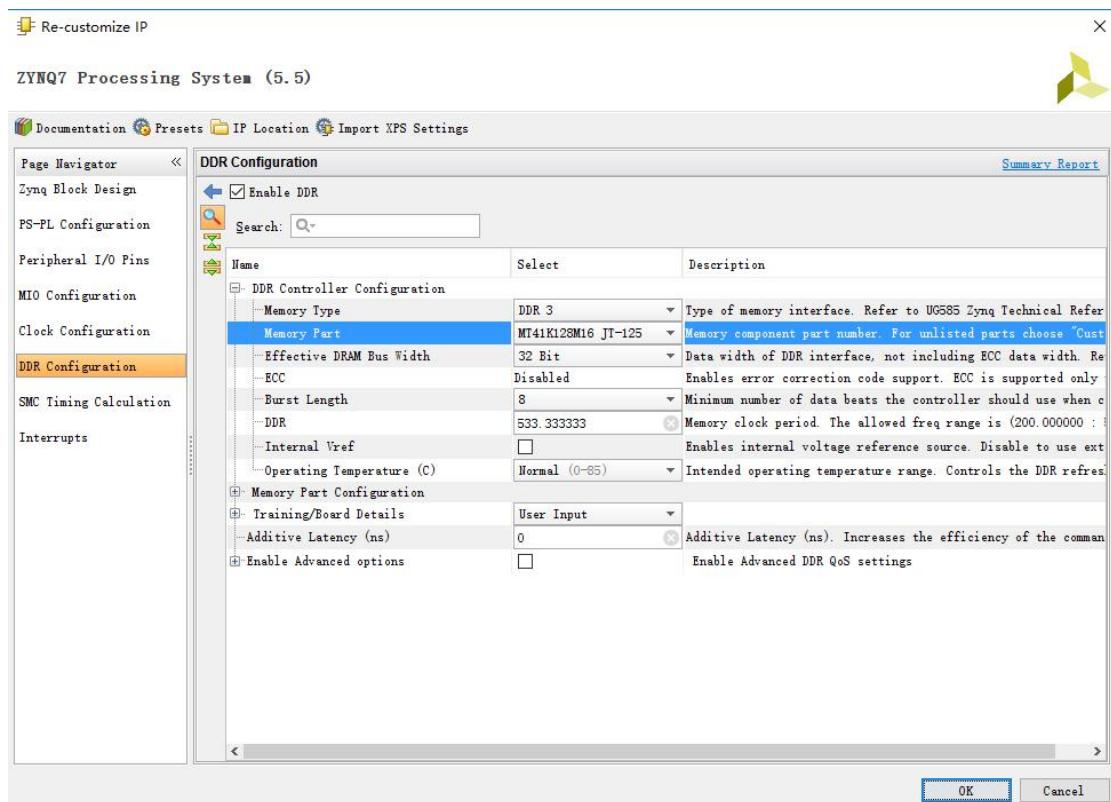


Step8: 根据下图设置 ZYNQ Processing system。

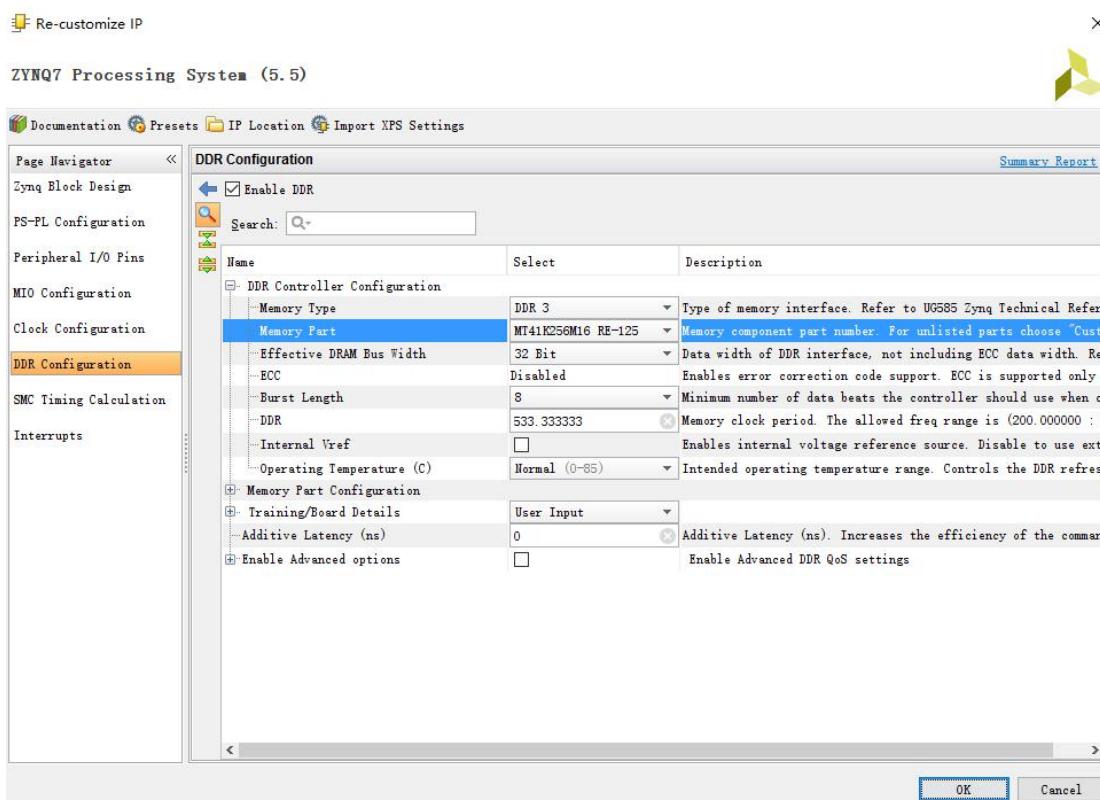




Miz702 内存型号如下设置：

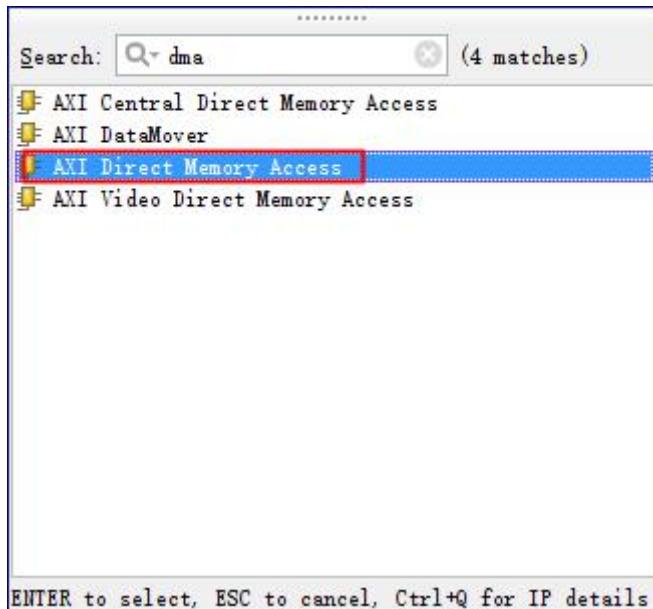


Miz702N 内存型号如下设置：

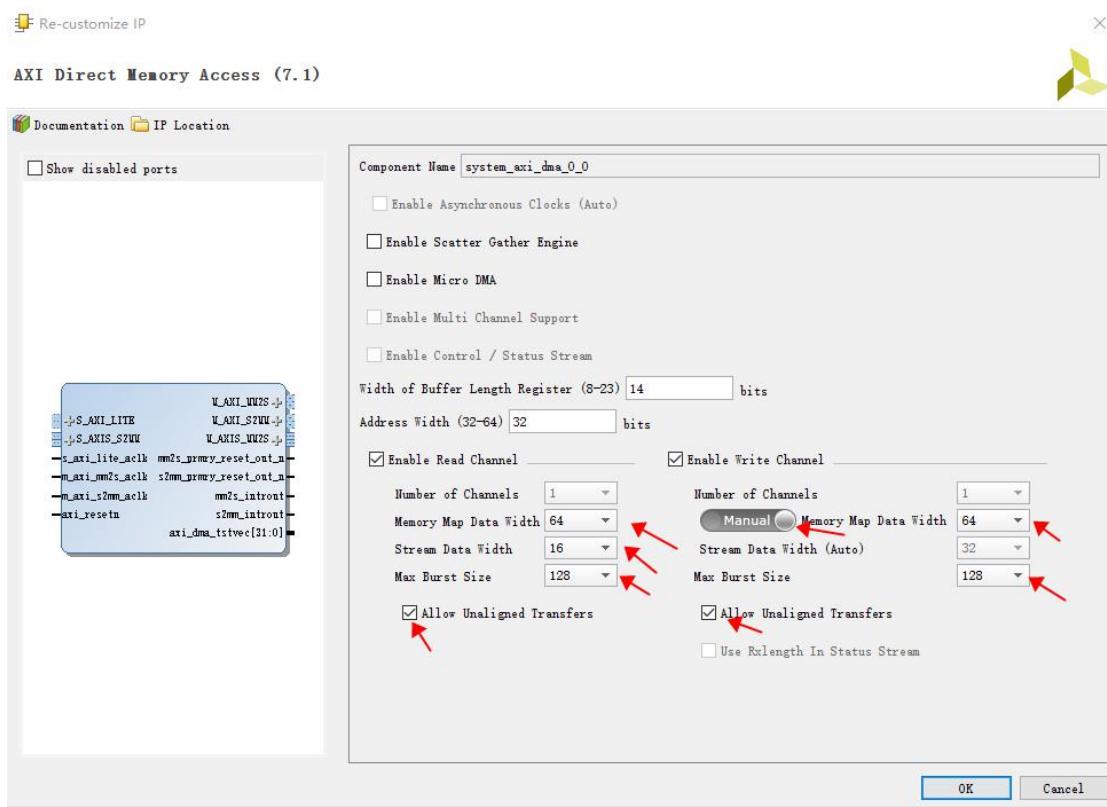


全部设置好了之后单击 OK 完成修改。

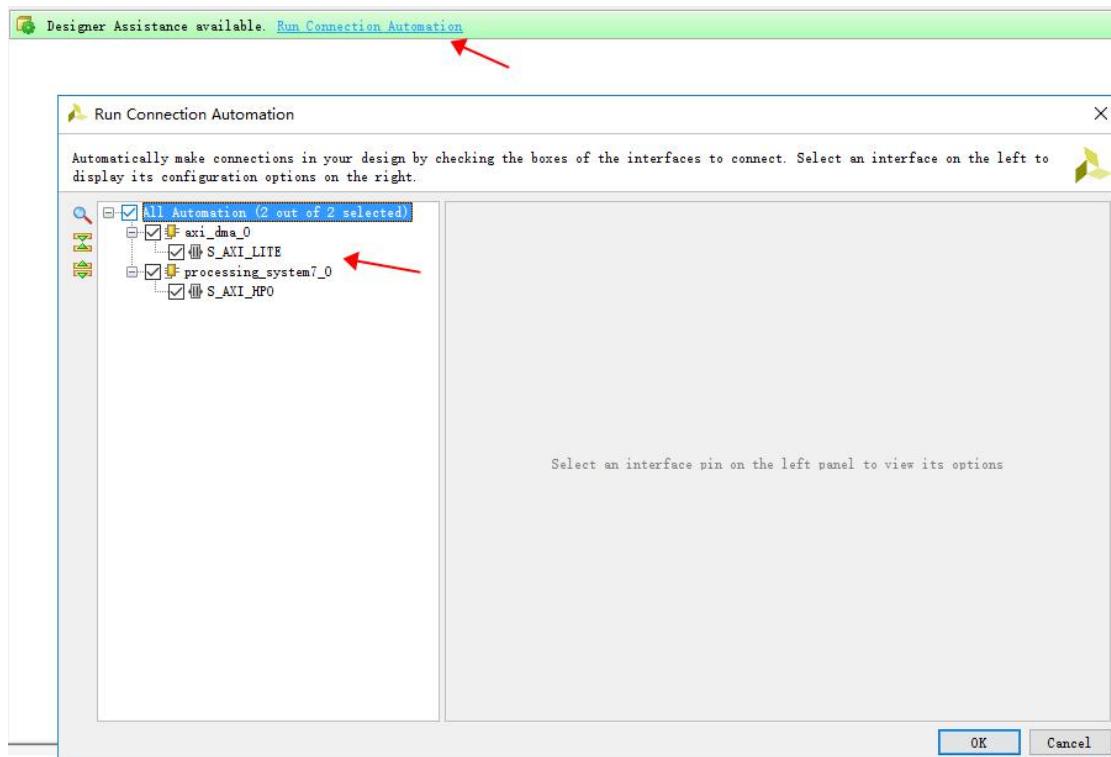
Step9: 根据上面讲过的方法，添加一个 DMA 的 I P。



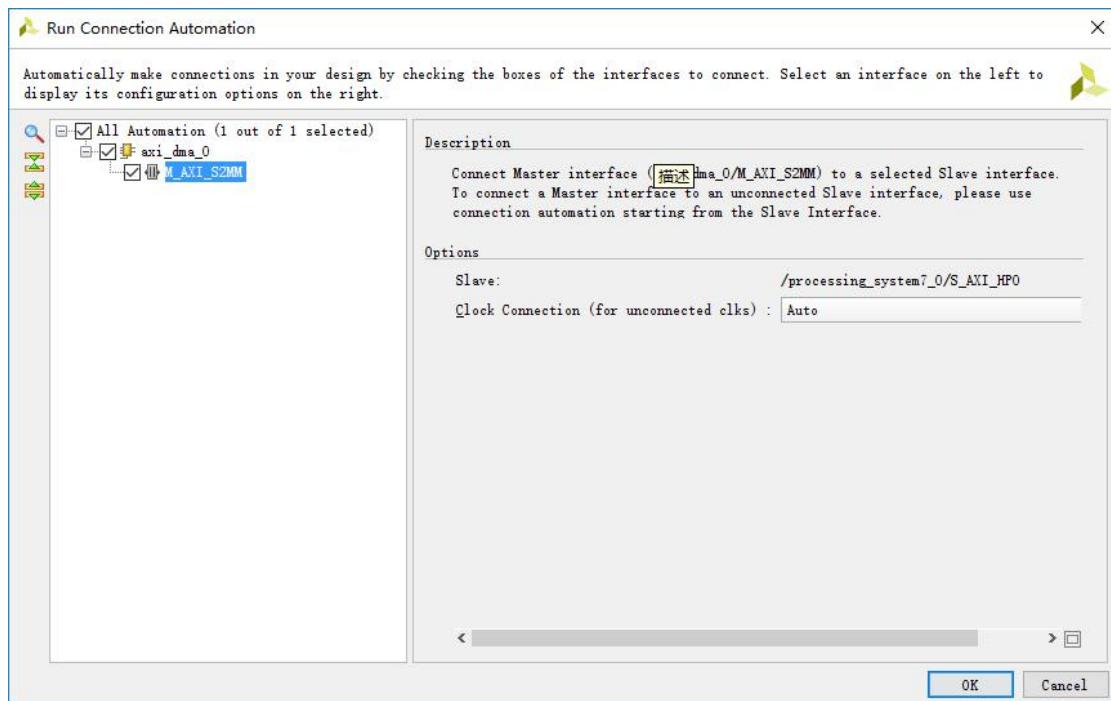
Step10: 如下图所示配置 DMA，然后单击 OK。



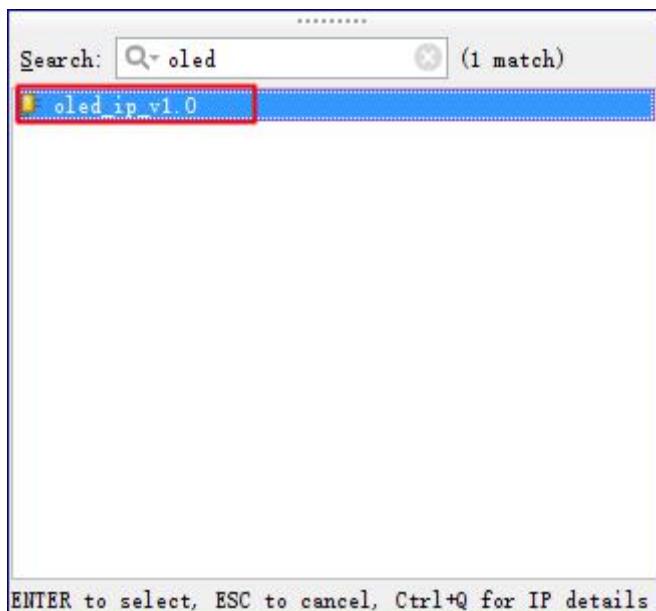
Step11: 单击 Run connection Automation。



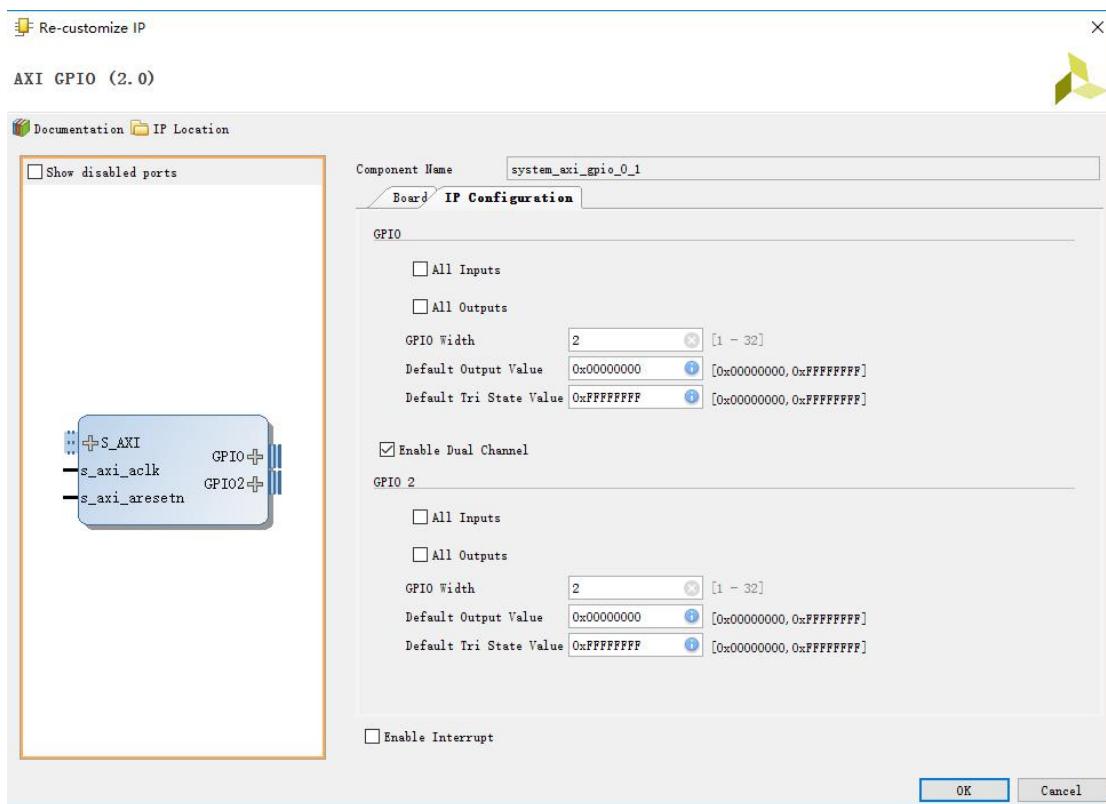
Step12: 再次单击 Run connection Automation。



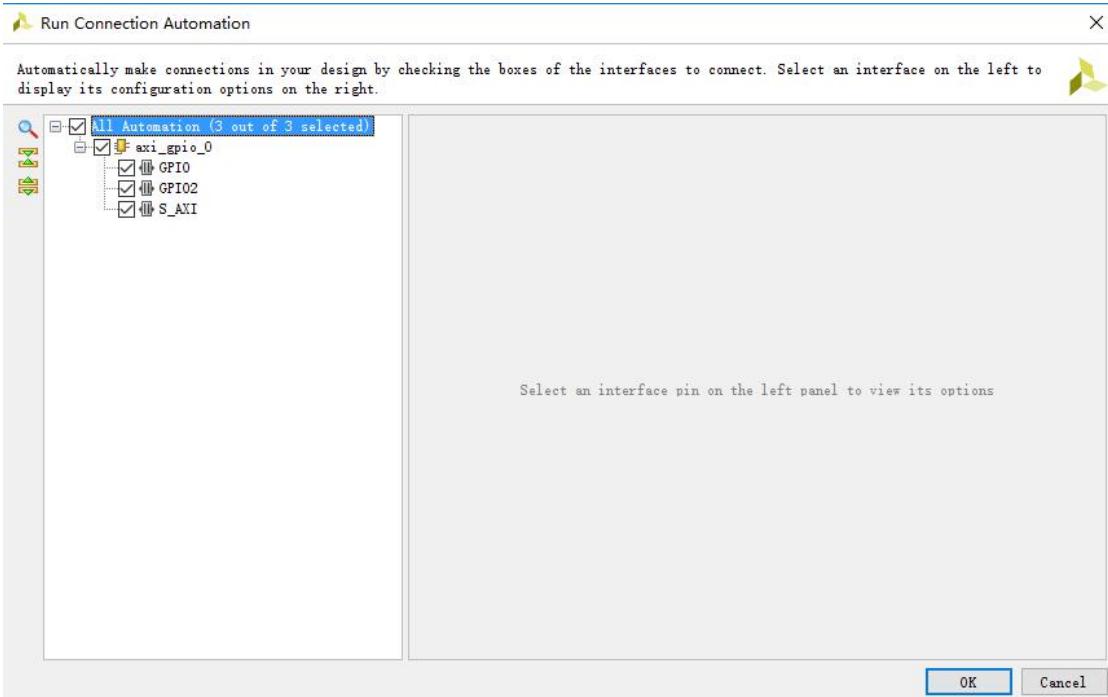
Step13: 添加一个oled IP, 然后单击 run connection Automation。



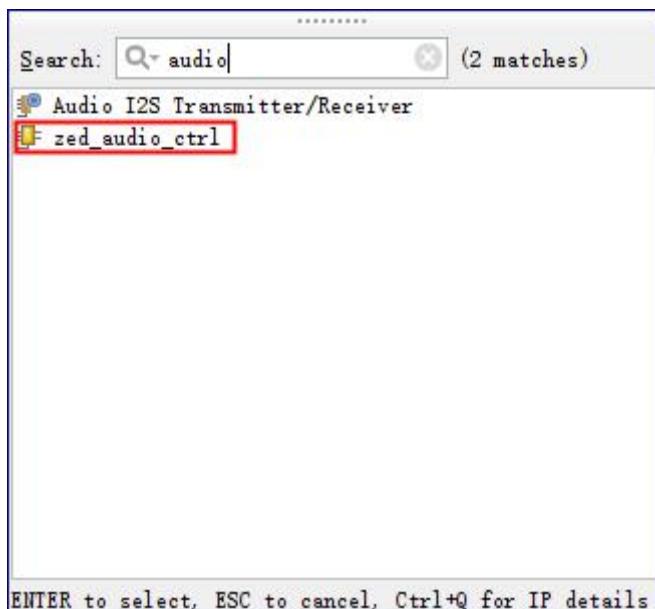
Step14: 添加一个 AXI GPIO IP，并对其进行如下配置。



Step15: 单击 Run connection Automation，然后按下图设置，最后单击 OK。

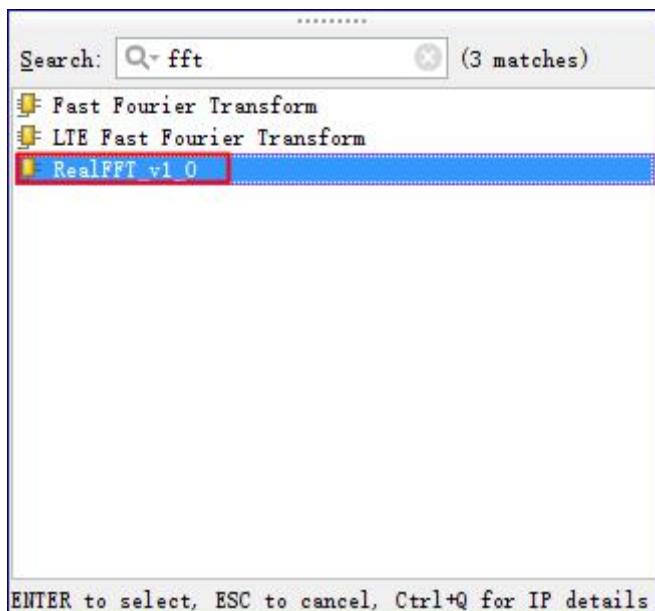


Step16: 添加一个 zed_audio_ctrl IP，然后单击 Run connection Automation。

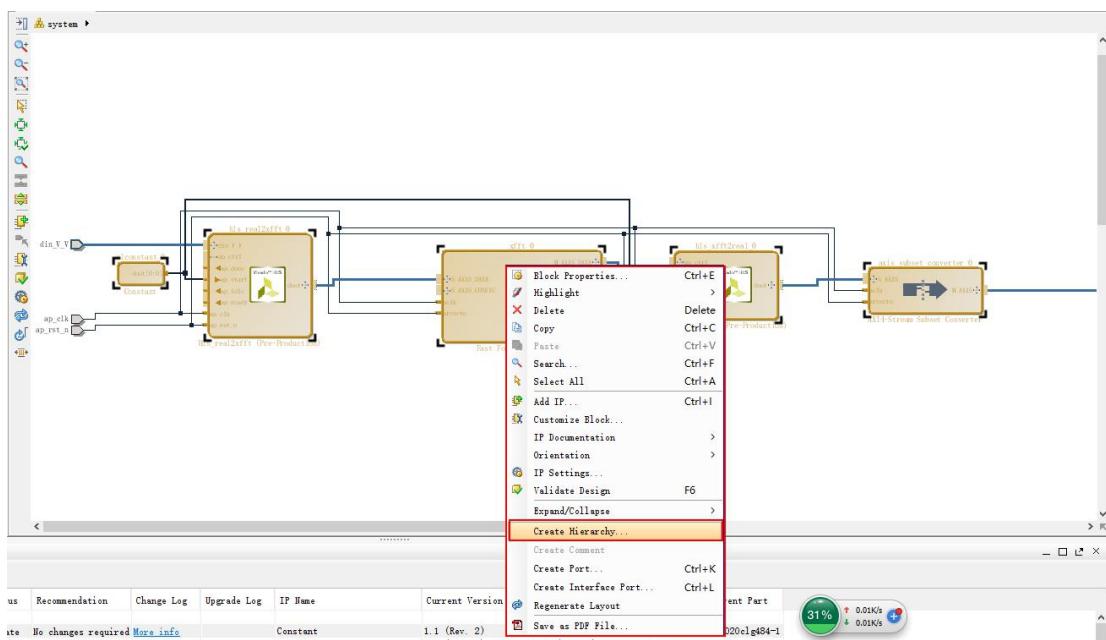


Step17: 依次为 SDATA_I, BCLK, LRCLK, SDATA_O, OLED[5:0], IIC1 引出端口，方法是选中这些信号，然后按 Ctrl+T。

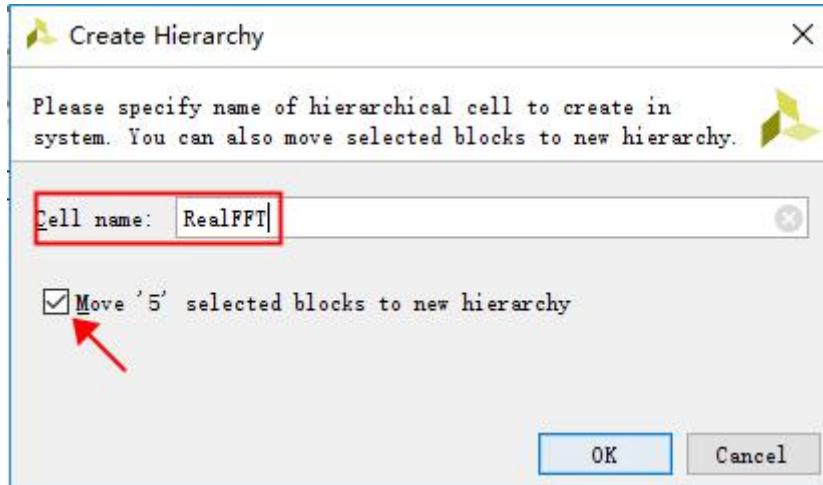
Step18: 将第八章中封装好的 IP 添加到 BD 文件当中。



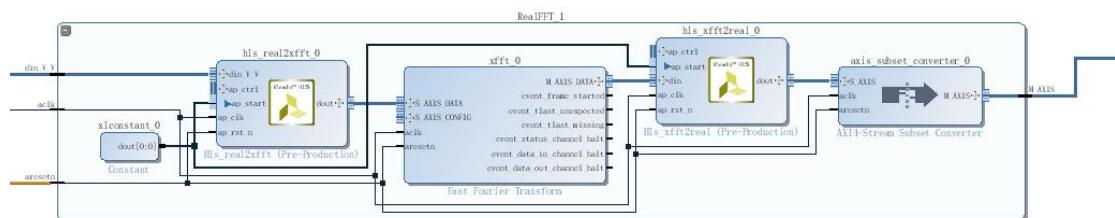
此处另外介绍一种方便多 IP 管理的方法：先按照第八章封装 IP 的硬件电路把要用到的 IP 添加到 BD 文件当中，所有配置与连线都与之前一样，完成之后选中这些 IP，然后右单击，选择 Create Hierarchy，如下图所示：



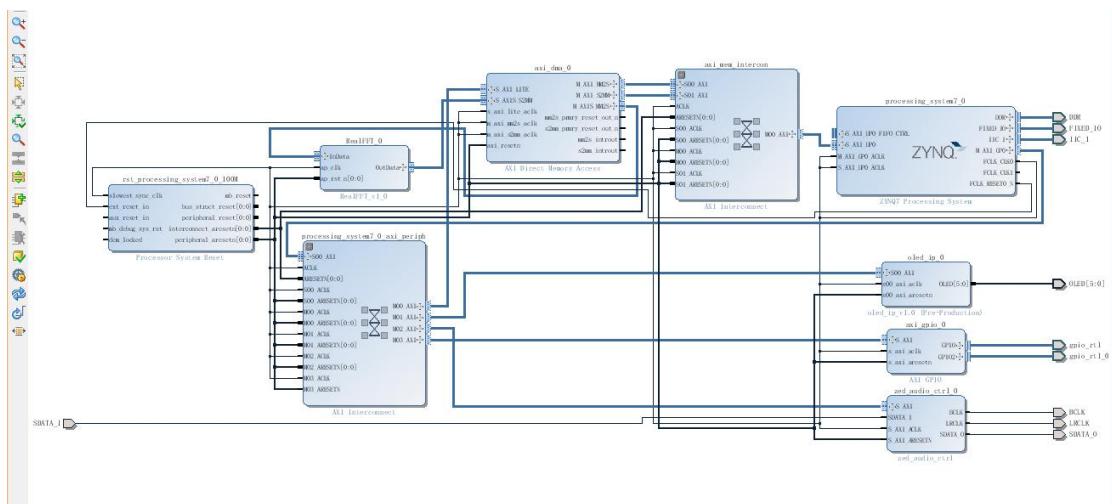
在跳出来的窗口中如下图设置，最后单击 OK。

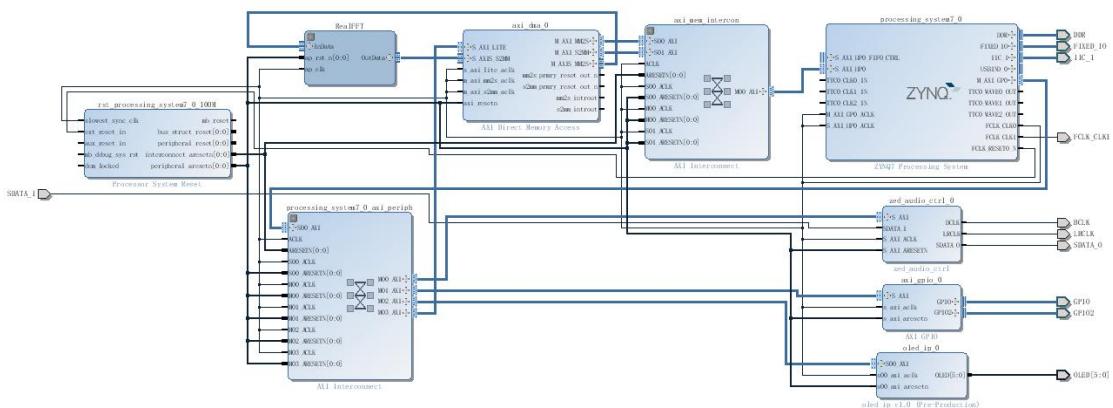


之后系统就会将这几个 IP 整合到一起，生成了一个新的 GUI，在这个 GUI 上有一个加号图标，将鼠标放到这个加号图标上，鼠标的图形会变成斜的展开的式样，这时单击即可查看各个 IP，这样就方便了 IP 的管理，如下图所示：

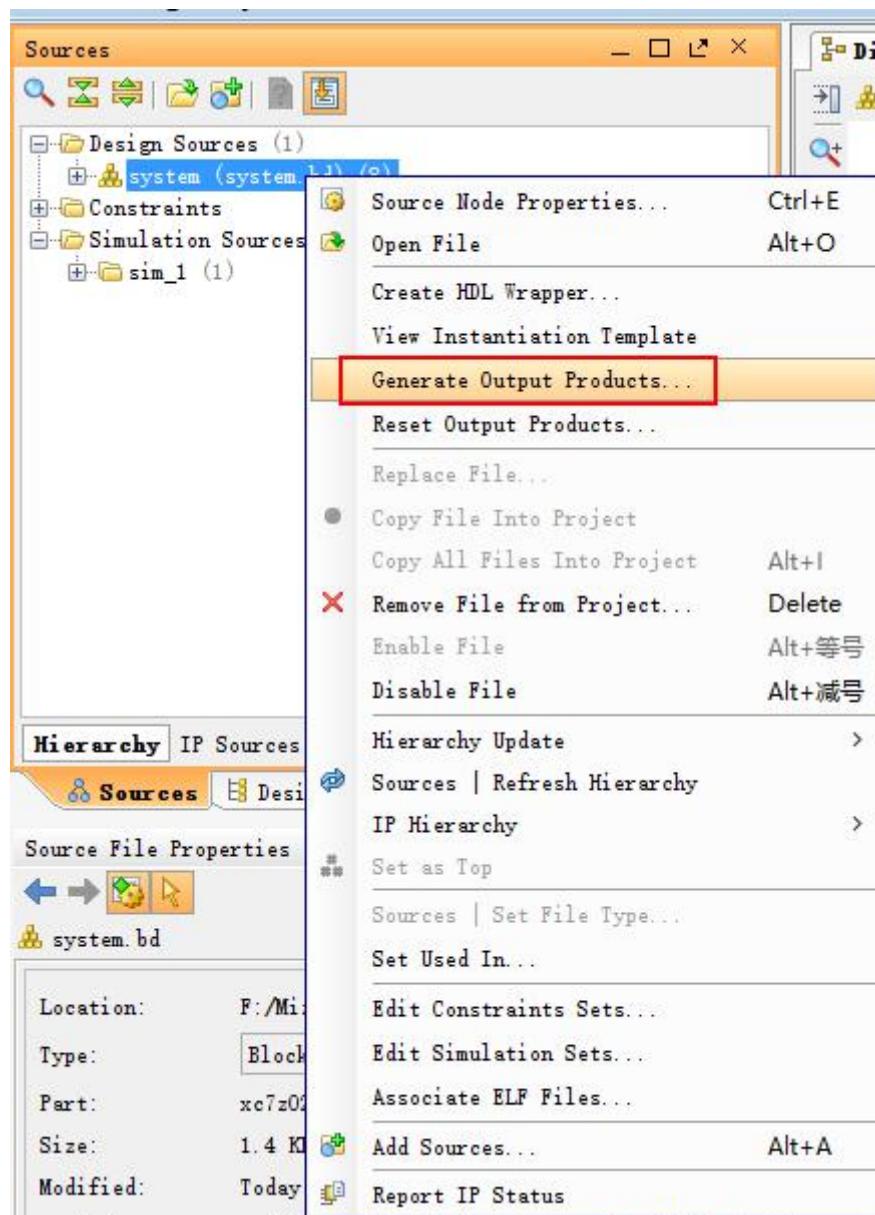


Step19：将以下线路连接起来： InData->M_AXIS_MM2S， OutData->S_AXIS_S2MM， ap_clk->s_axi_lite_aclk, ap_rst_n->axi_resetn (Create Hierarchy之后要将其引出的端口删除再进行连接)。完成后的整体电路如下图所示：

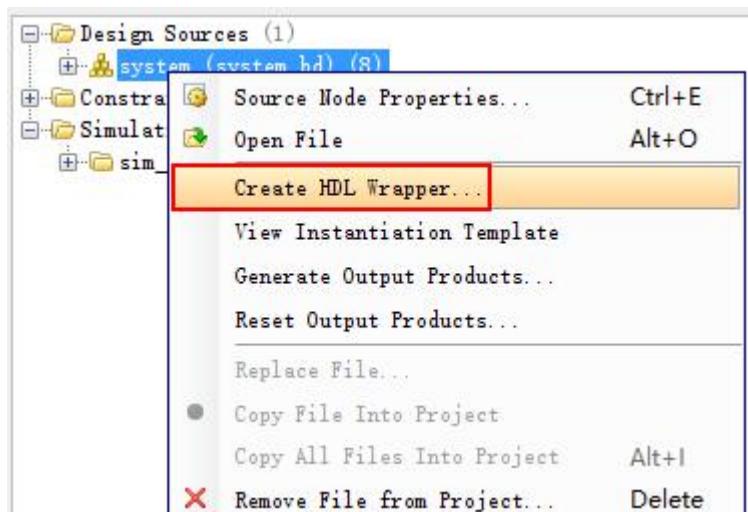




Step20: 选中 BD 文件，右单击选择 Generator Output Products...。



Step21: 右单击 BD 文件，选择 Create HDL Wrapper。



Step22: 添加一个名为 zynq_pin 的约束文件，并添加如下约束：

Miz702 用户添加如下约束

```
set_property PACKAGE_PIN AA6 [get_ports BCLK]
set_property IOSTANDARD LVCMOS33 [get_ports BCLK]

set_property PACKAGE_PIN Y6 [get_ports LRCLK]
set_property IOSTANDARD LVCMOS33 [get_ports LRCLK]

set_property PACKAGE_PIN AA7 [get_ports SDATA_I]
set_property IOSTANDARD LVCMOS33 [get_ports SDATA_I]

set_property PACKAGE_PIN Y8 [get_ports SDATA_O]
set_property IOSTANDARD LVCMOS33 [get_ports SDATA_O]

#MCLK
set_property PACKAGE_PIN AB2 [get_ports FCLK_CLK1]
set_property IOSTANDARD LVCMOS33 [get_ports FCLK_CLK1]

set_property PACKAGE_PIN AB4 [get_ports iic_1_scl_io]
set_property IOSTANDARD LVCMOS33 [get_ports iic_1_scl_io]

set_property PACKAGE_PIN AB5 [get_ports iic_1_sda_io]
set_property IOSTANDARD LVCMOS33 [get_ports iic_1_sda_io]

set_property PACKAGE_PIN AB1 [get_ports { gpio_rtl_tri_io[0] }]
set_property IOSTANDARD LVCMOS33 [get_ports { gpio_rtl_tri_io[0] }]
```

```

set_property PACKAGE_PIN Y5 [get_ports { gpio_rtl_tri_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports { gpio_rtl_tri_io[1]}]

# GPIO2[0] for bypass a sample or send it through the filter
set_property PACKAGE_PIN F22 [get_ports { gpio_rtl_0_tri_io[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports { gpio_rtl_0_tri_io[0]}]

# GPIO2[1] for bypass a sample or send it through the filter
set_property PACKAGE_PIN G22 [get_ports { gpio_rtl_0_tri_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports { gpio_rtl_0_tri_io[1]}]
# DC
set_property PACKAGE_PIN U10 [get_ports {OLED[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[0]}]

# RES
set_property PACKAGE_PIN U9 [get_ports {OLED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[1]}]

# SCLK
set_property PACKAGE_PIN AB12 [get_ports {OLED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[2]}]

# SDIN
set_property PACKAGE_PIN AA12 [get_ports {OLED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[3]}]

# "BAT"
set_property PACKAGE_PIN U11 [get_ports {OLED[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[4]}]

# "VDD"
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[5]}]
set_property PACKAGE_PIN U12 [get_ports {OLED[5]}]

```

Miz702N 用户添加如下约束

```

# DC
set_property PACKAGE_PIN K15 [get_ports {OLED[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[0]}]

# RES
set_property PACKAGE_PIN J16 [get_ports {OLED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[1]}]

```

```
# SCLK
set_property PACKAGE_PIN J18 [get_ports {OLED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[2]}]

# SDIN
set_property PACKAGE_PIN K18 [get_ports {OLED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[3]}]

# "BAT"
set_property PACKAGE_PIN J15 [get_ports {OLED[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[4]}]

# "VDD"
set_property IOSTANDARD LVCMOS33 [get_ports {OLED[5]}]
set_property PACKAGE_PIN J17 [get_ports {OLED[5]}]

set_property PACKAGE_PIN V4 [get_ports BCLK]
set_property IOSTANDARD LVCMOS33 [get_ports BCLK]

set_property PACKAGE_PIN Y4 [get_ports LRCLK]
set_property IOSTANDARD LVCMOS33 [get_ports LRCLK]

set_property PACKAGE_PIN AB6 [get_ports SDATA_I]
set_property IOSTANDARD LVCMOS33 [get_ports SDATA_I]

set_property PACKAGE_PIN V5 [get_ports SDATA_O]
set_property IOSTANDARD LVCMOS33 [get_ports SDATA_O]

#MCLK
set_property PACKAGE_PIN AB2 [get_ports FCLK_CLK1]
set_property IOSTANDARD LVCMOS33 [get_ports FCLK_CLK1]

set_property PACKAGE_PIN AB4 [get_ports iic_1_scl_io]
set_property IOSTANDARD LVCMOS33 [get_ports iic_1_scl_io]

set_property PACKAGE_PIN AB5 [get_ports iic_1_sda_io]
set_property IOSTANDARD LVCMOS33 [get_ports iic_1_sda_io]

set_property PACKAGE_PIN AB1 [get_ports {gpio_rtl_tri_io[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_io[0]}]

set_property PACKAGE_PIN AA4 [get_ports {gpio_rtl_tri_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_tri_io[1]}]
```

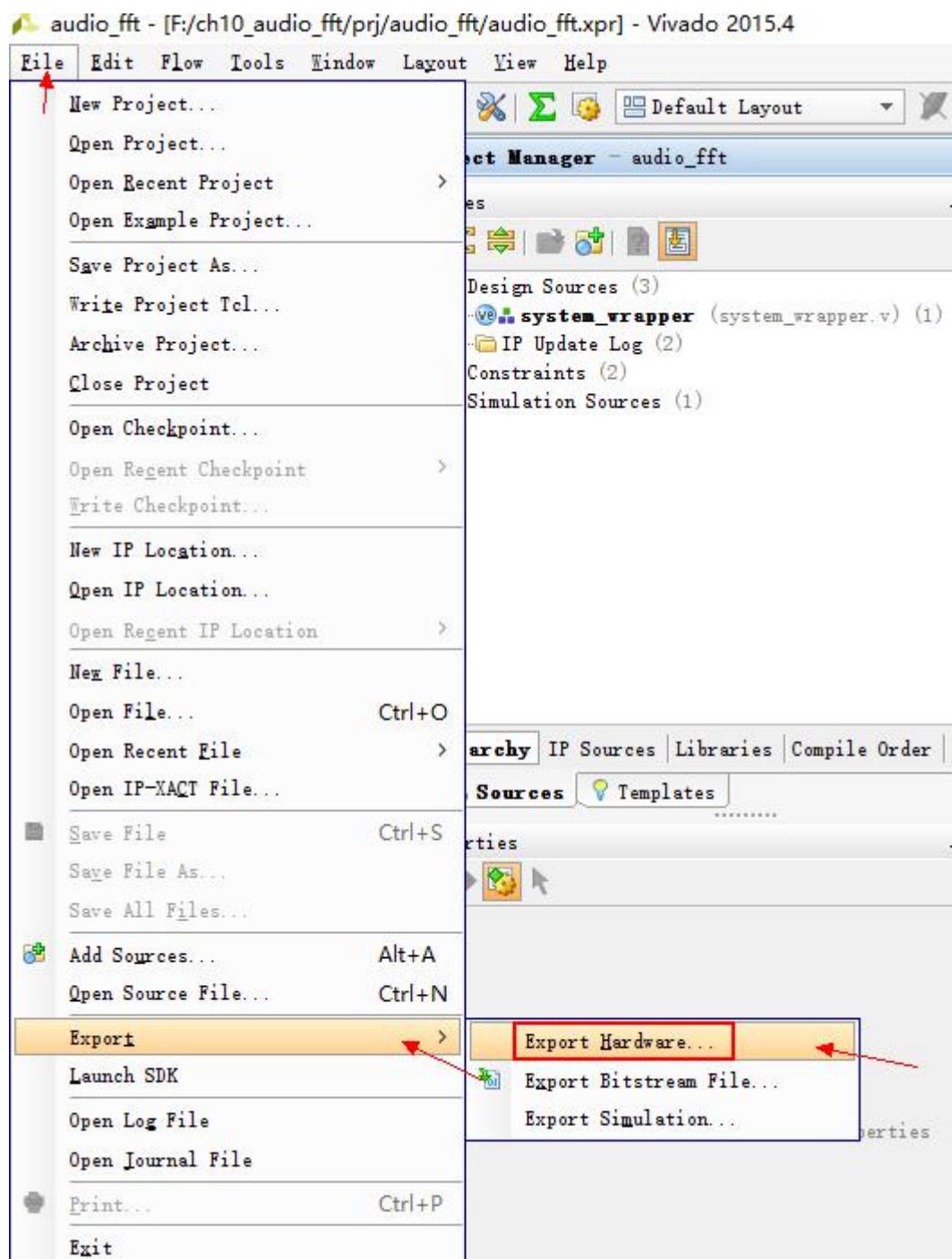
```
# GPIO2[0] for bypass a sample or send it through the filter  
set_property PACKAGE_PIN F22 [get_ports {gpio_rtl_0_tri_io[0]}]  
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_0_tri_io[0]}]  
  
# GPIO2[1] for bypass a sample or send it through the filter  
set_property PACKAGE_PIN G22 [get_ports {gpio_rtl_0_tri_io[1]}]  
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_rtl_0_tri_io[1]}]
```

Step23: 单击  生成 bit 文件。

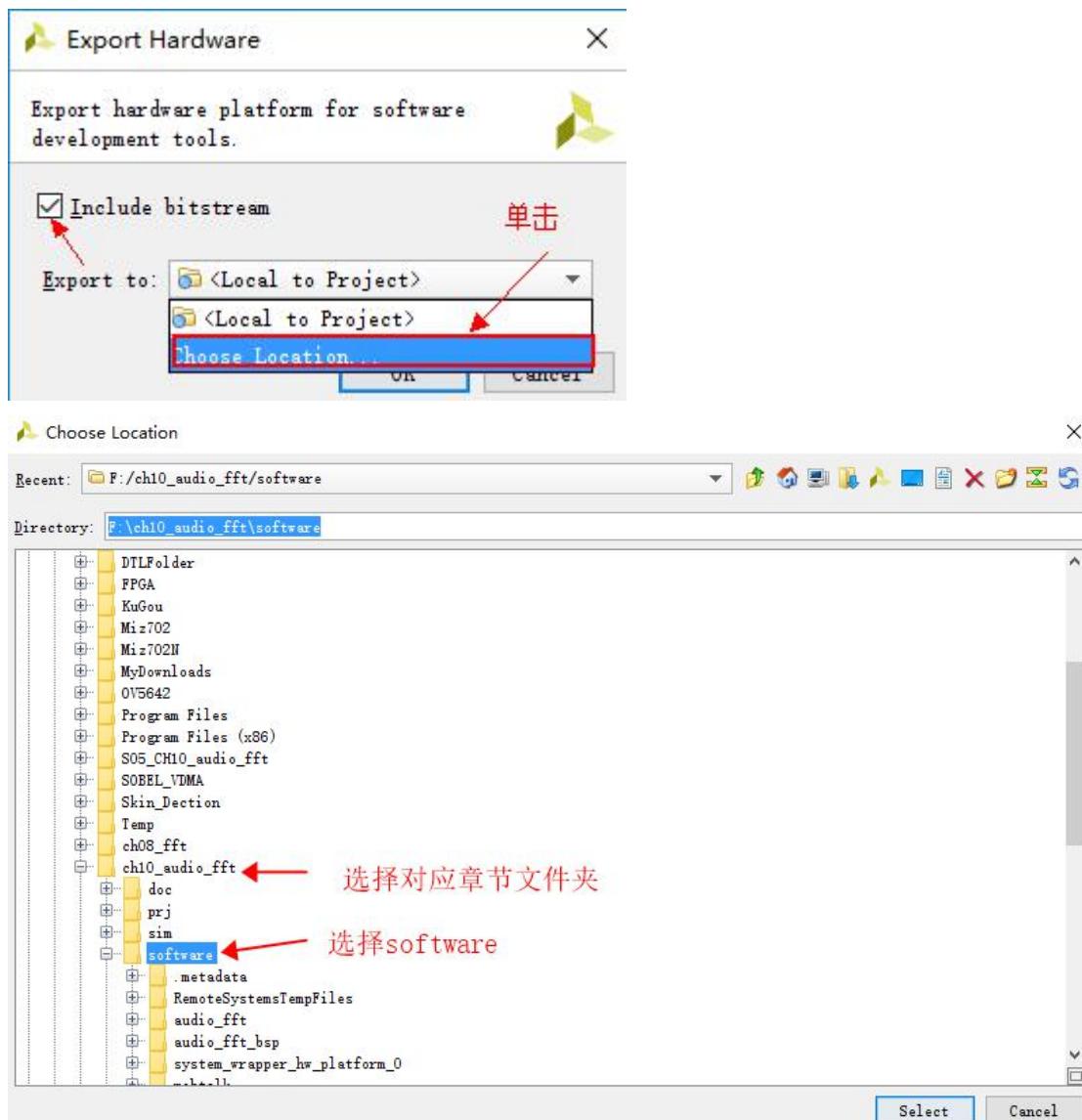
10.3 导入到 SDK

生成 Bit 文件之后，接下来就是 SDK 驱动的编写。在我们提供的源码中，已经提供了供 SDK 使用的工程，为了节省时间，可以直接使用我们提供的工程。

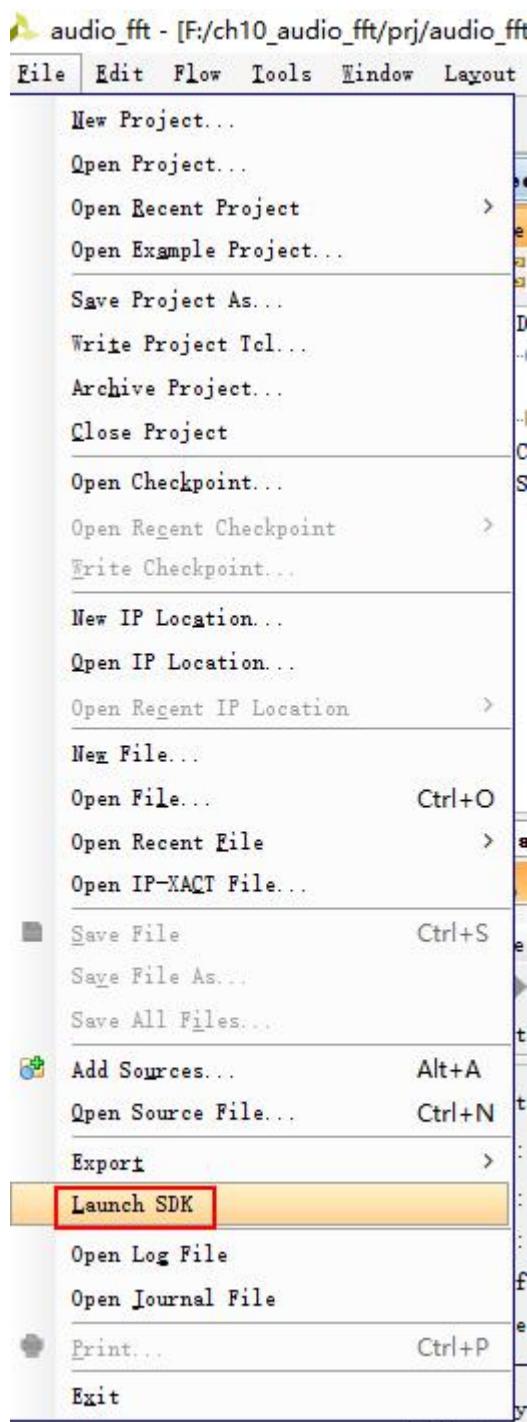
Step1: 单击 File-Export-Export Hardware..。



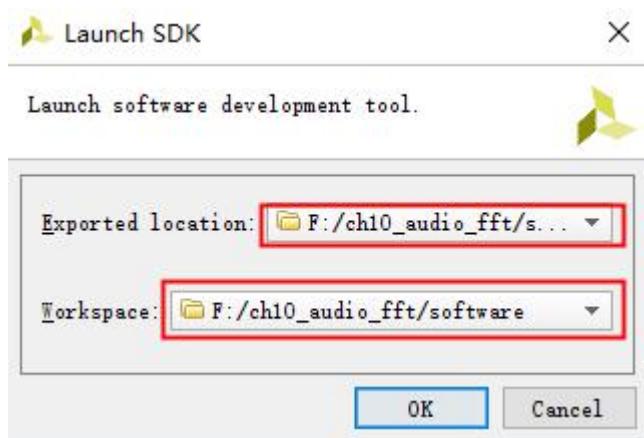
Step2：选择导出的路径，这里将其指向我们提供的SDK驱动工程（在我们提供的源代码对应章节文件夹下的software文件夹），如下图所示。



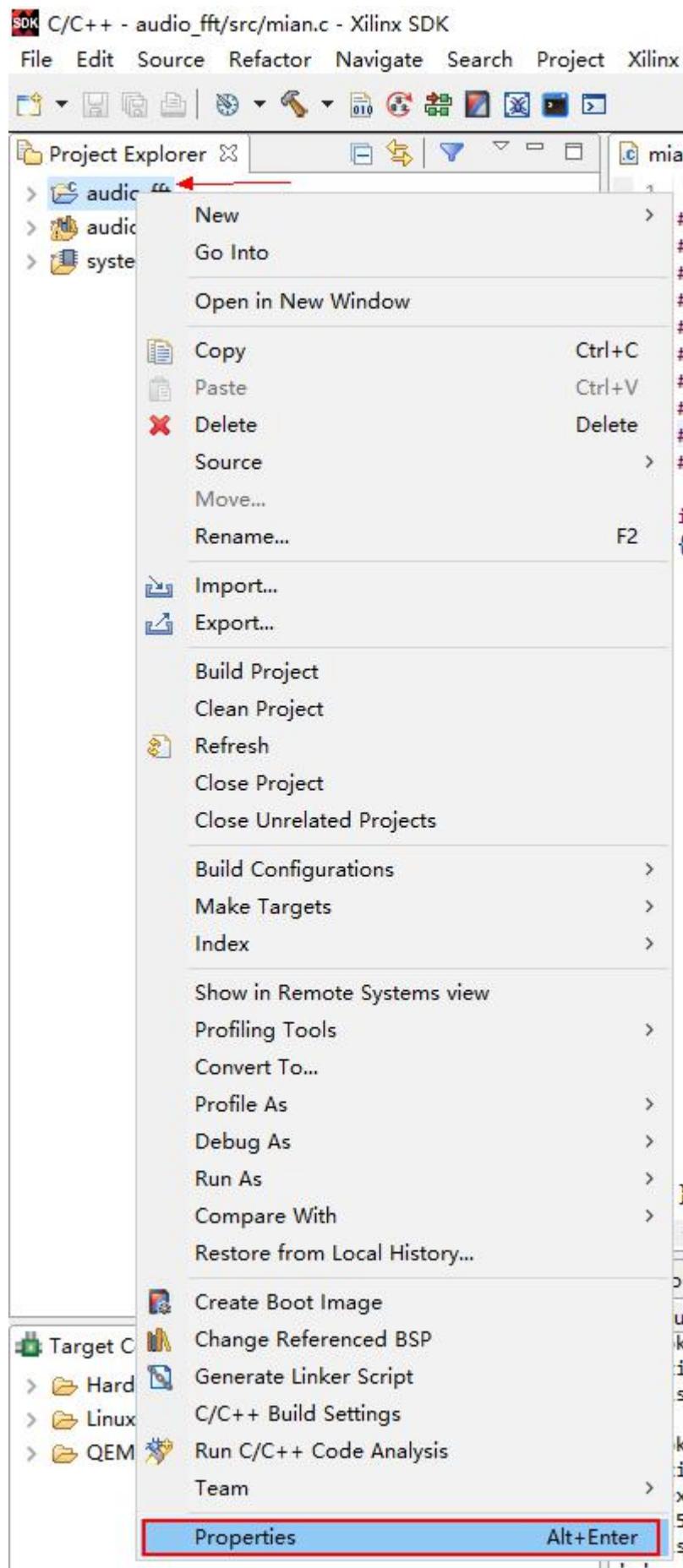
Step3: 单击 File-Launch SDK。



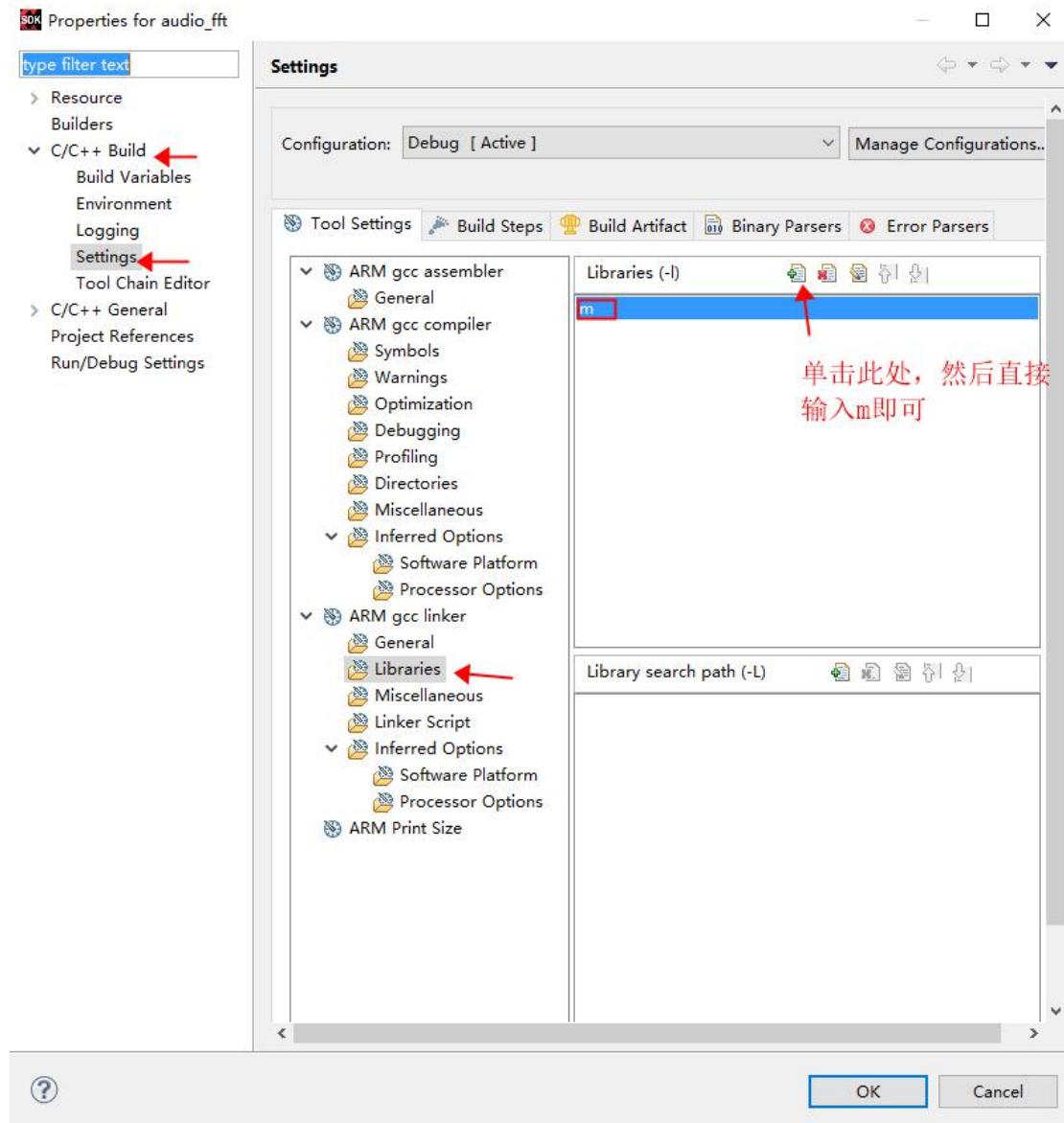
Step4: 在弹出的窗口中，将下图方框中的路径都选择为 software 文件夹。



Step5: 右单击 audio fft,选择 Properties。

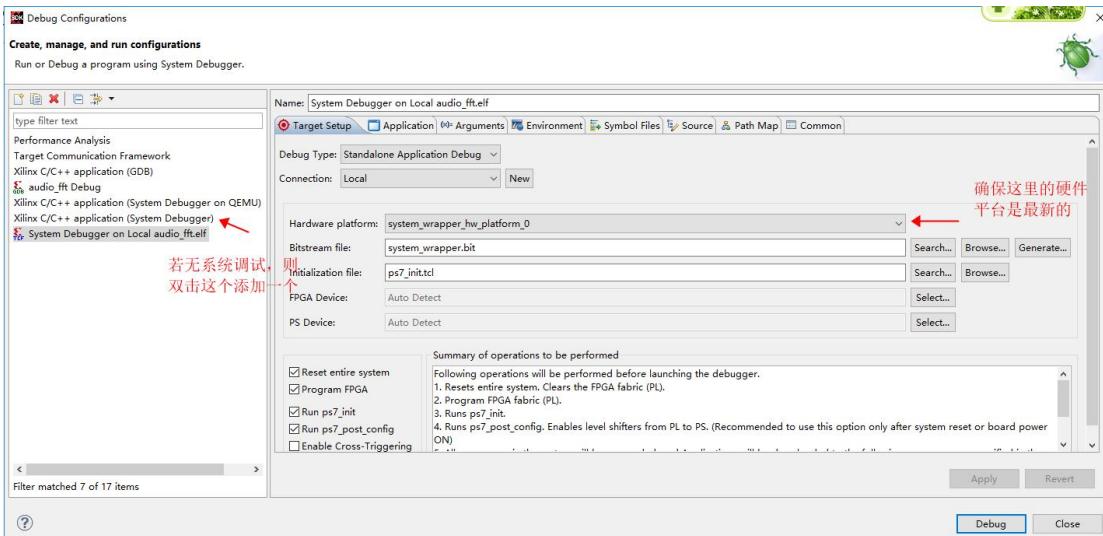


Step5：在下图所示界面将数学函数库关联到 SDK 当中来。

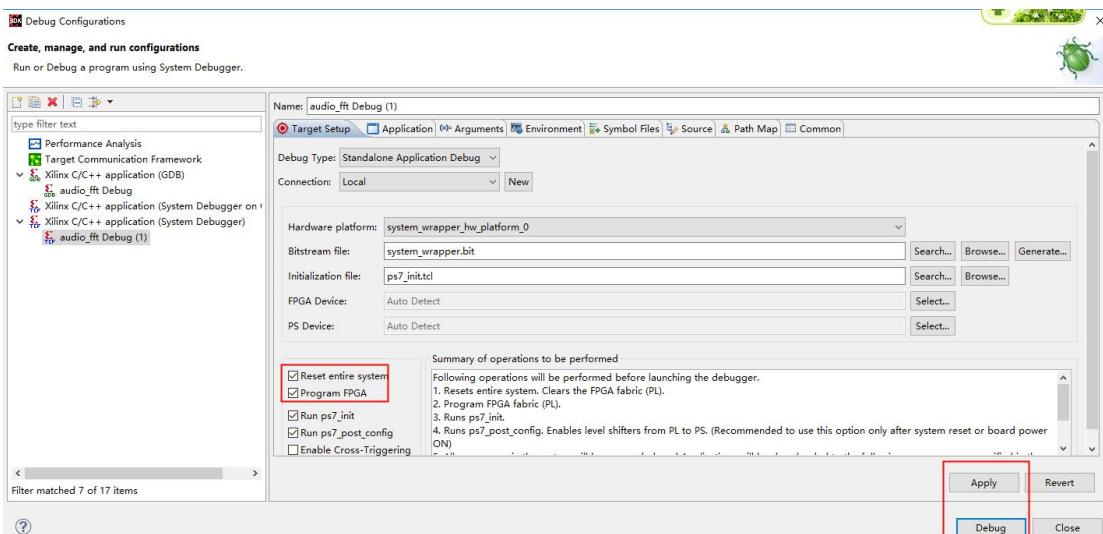


Step6：右击工程，选择 Debug as ->Debug configuration。

Step7：选中 system Debugger, 双击创建一个系统调试。



Step8：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。



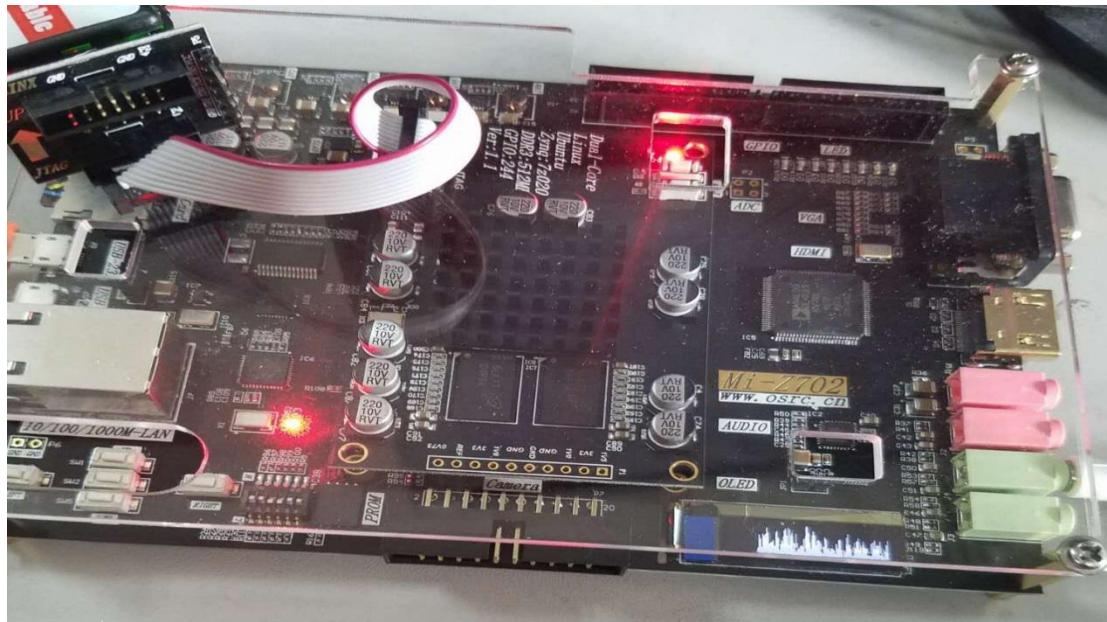
系统运行结果如下图所示：

```

maxfre init
maxfre mag done
maxfre compare done
Cur n=1;maxf= 1.000000 Hz;max mag=2599.963867;
oled_show begin
*-----End of test-----
*-----start of test-----
*-----End of frame-----

```

在音频输入接入开发板 的 J4 口，可看到 oled 实时显示音频的频谱。



10.4 程序分析

本章的程序主要可以分为四大部分：DMA 驱动、OLED 驱动、FFT 配置、ADAU1761 驱动。其中前两个都在我们之前的教程中有过详细的讲解，这里着重讲解后两部分。先看到 ADAU1761 的驱动，ADAU1761.c 就是 ADAU1761 的驱动。AUAD1761.c 中有五个函数，其中 IicConfig() 与 AudioWriteToReg() 比较简单，其中的大部分函数在之前都有过讲解，在用户自己设计相关项目时，可以直接拿来稍微修改就可以使用。本节重点讲解另外三个函数，先看到下面这个函数：

```

void AudioPllConfig() {
    unsigned char u8TxData[8], u8RxData[6];
    int Status;

    Status = IicConfig(XPAR_XIICPS_0_DEVICE_ID); //, FMC_IIC_ID, HDMI_IIC_ID);
    if(Status != XST_SUCCESS) {
        xil_printf("\nError initializing IIC");
        //return XST_FAILURE;
    }

    AudioWriteToReg(R0_CLOCK_CONTROL, 0x0E);

    // Write 6 bytes to R1
    u8TxData[0] = 0x40;
    u8TxData[1] = 0x02;
    u8TxData[2] = 0x02; // byte 1
    u8TxData[3] = 0x71; // byte 2
    u8TxData[4] = 0x02; // byte 3
    u8TxData[5] = 0x3C; // byte 4
    u8TxData[6] = 0x21; // byte 5
    u8TxData[7] = 0x01; // byte 6

    XIicPs_MasterSendPolled(&Iic, u8TxData, 8, (IIC_SLAVE_ADDR >> 1));
    while(XIicPs_BusIsBusy(&Iic));

    // Poll PLL Lock bit
    u8TxData[0] = 0x40;
    u8TxData[1] = 0x02;

    do {
        XIicPs_MasterSendPolled(&Iic, u8TxData, 2, (IIC_SLAVE_ADDR >> 1));
        while(XIicPs_BusIsBusy(&Iic));
        XIicPs_MasterRecvPolled(&Iic, u8RxData, 6, (IIC_SLAVE_ADDR >> 1));
        while(XIicPs_BusIsBusy(&Iic));
    }
    while((u8RxData[5] & 0x02) == 0);

    AudioWriteToReg(R0_CLOCK_CONTROL, 0x0F); //COREN
}

```

从上图的函数命令和一些注释我们可以得知这是一个 pll 的配置，一开始是之前老套路的初始化程序，初始化之后，我们注意到其向 R0 写入了一个值 0E，我们可以通过前面的讲解来看看这是什么意思。

Table 12. Clock Control Register (Register R0, Address 0x4000)

Bits	Bit Name	Settings
3	CLKSRC	0: Direct from MCLK pin (default) 1: PLL clock
[2:1]	INFREQ[1:0]	00: 256 × f _s (default) 01: 512 × f _s 10: 768 × f _s 11: 1024 × f _s E=1110
0	COREN	0: Core clock disabled (default) 1: Core clock enabled

因此这里的意思就很明显了禁止 core clock。接下来看到这一段程序。

```

// Write 6 bytes to R1
u8TxData[0] = 0x40;
u8TxData[1] = 0x02;
u8TxData[2] = 0x00; // byte 1 //625
u8TxData[3] = 0x7D; // byte 2
u8TxData[4] = 0x00; // byte 3 //572
u8TxData[5] = 0x12; // byte 4
u8TxData[6] = 0x31; // byte 5
u8TxData[7] = 0x01; // byte 6

XIicPs_MasterSendPolled(&Iic, u8TxData, 8, (IIC_SLAVE_ADDR >> 1));
while(XIicPs_BusIsBusy(&Iic));

```

从上图的注释可以知道这部分就是一个向 R1 寄存器的写入操作，此时两个时钟控制寄存器都已配置好，根据 10.1.2 的时钟树原理图可以求出 core clock：

$$\text{Core clock} = \text{CLKSRC} * \text{INFREQ}[1:0]$$

其中 $\text{CLKSRC} = \text{MCLK} * (\text{R} + \text{N}/\text{M})/\text{X}$, 对照 R1 的寄存器定义, 我们得知 M 是写入的 byte1 和 byte2(也就是 (0x007D=125)), N 是写入的 byte3 和 byte4 (0x0012=18), R 和 X 分别是 byte5 的 Bit[14:11] 和 Bit[10:9]。此时算得 $\text{CLKSRC} = 8 * (6 + 18/125) = 49.152$ 。

求得了 CLKSRC 之后，就可以求出采样率了。具体怎么求，数据中已经给出了方法，如下图所示：

CORE CLOCK

Clocks for the converters, the serial ports, and the DSP are derived from the core clock. The core clock can be derived directly from MCLK or it can be generated by the PLL. The CLKSRC bit (Bit 3 in Register R0, Address 0x4000) determines the clock source.

The INFREQ[1:0] bits should be set according to the expected input clock rate selected by CLKSRC; this value also determines the core clock rate and the base sampling frequency, f_s .

For example, if the input to CLKSRC = 49.152 MHz (from PLL), then

$$\text{INFREQ}[1:0] = 1024 \times f_s$$

$$f_s = 49.152 \text{ MHz} / 1024 = 48 \text{ kHz}$$

The PLL output clock rate is always $1024 \times f_s$, and the clock control register automatically sets the INFREQ[1:0] bits to $1024 \times f_s$ when using the PLL. When using a direct clock, the INFREQ[1:0] frequency should be set according to the MCLK pin clock rate and the desired base sampling frequency.

在本节程序中， INFREQ=11 (E=1110)， CLKSRC 为 49.152，刚好和图中数据一致，因此和上图中的采样率 f_s 一致，都为 48KHZ。

接下来的两个函数是对 ADAU1761 的输入和输出进行配置，如下图所示：

To utilize the maximum amount of DSP instructions, the core clock should run at a rate of $1024 \times f_s$.

Table 12. Clock Control Register (Register R0, Address 0x4000)

Bits	Bit Name	Settings
3	CLKSRC	0: Direct from MCLK pin (default) 1: PLL clock
[2:1]	INFREQ[1:0]	00: $256 \times f_s$ (default) 01: $512 \times f_s$ 10: $768 \times f_s$ 11: $1024 \times f_s$
0	COREN	0: Core clock disabled (default) 1: Core clock enabled

```

112 void AudioConfigureJacks()
113 {
114     AudioWriteToReg(R4_RECORD_MIXER_LEFT_CONTROL_0, 0x01); //enable mixer 1
115     AudioWriteToReg(R5_RECORD_MIXER_LEFT_CONTROL_1, 0x07); //unmute Left channel of line in into mxc_1 and set gain to 6 dB
116     AudioWriteToReg(R6_RECORD_MIXER_RIGHT_CONTROL_0, 0x01); //enable mixer 2
117     AudioWriteToReg(R7_RECORD_MIXER_RIGHT_CONTROL_1, 0x07); //unmute Right channel of line in into mxc_2 and set gain to 6 dB
118     AudioWriteToReg(R19_ADC_CONTROL, 0x13); //enable ADCs
119
120     AudioWriteToReg(R22_PLAYBACK_MIXER_LEFT_CONTROL_0, 0x21); //unmute Left DAC into Mxr_3; enable mxc_3
121     AudioWriteToReg(R24_PLAYBACK_MIXER_RIGHT_CONTROL_0, 0x41); //unmute Right DAC into Mxr_4; enable mxc_4
122     AudioWriteToReg(R26_PLAYBACK_LR_MIXER_LEFT_LINE_OUTPUT_CONTROL, 0x05); //unmute Mxr3 into Mxr5 and set gain to 6dB; enable mxc_5
123     AudioWriteToReg(R27_PLAYBACK_LR_MIXER_RIGHT_LINE_OUTPUT_CONTROL, 0x11); //unmute Mxr4 into Mxr6 and set gain to 6dB; enable mxc_6
124     AudioWriteToReg(R29_PLAYBACK_HEADPHONE_LEFT_VOLUME_CONTROL, 0x00); //Mute Left channel of HP port (LHP)
125     AudioWriteToReg(R30_PLAYBACK_HEADPHONE_RIGHT_VOLUME_CONTROL, 0x00); //Mute Right channel of HP port (RHP)
126     AudioWriteToReg(R31_PLAYBACK_LINE_OUTPUT_LEFT_VOLUME_CONTROL, 0xE6); //set LOUT volume (0db); unmute left channel of Line out port; set Line out port to l
127     AudioWriteToReg(R32_PLAYBACK_LINE_OUTPUT_RIGHT_VOLUME_CONTROL, 0xE6); //set ROUT volume (0db); unmute right channel of Line out port; set Line out port to r
128     AudioWriteToReg(R35_PLAYBACK_POWER_MANAGEMENT, 0x03); //enable left and right channel playback (not sure exactly what this does...)
129     AudioWriteToReg(R36_DAC_CONTROL_0, 0x03); //enable both DACs
130
131     AudioWriteToReg(R58_SERIAL_INPUT_ROUTE_CONTROL, 0x01); //Connect I2S serial port output (SDATA_O) to DACs
132     AudioWriteToReg(R59_SERIAL_OUTPUT_ROUTE_CONTROL, 0x01); //connect I2S serial port input (SDATA_I) to ADCs
133
134     AudioWriteToReg(R65_CLOCK_ENABLE_0, 0x7F); //Enable clocks
135     AudioWriteToReg(R66_CLOCK_ENABLE_1, 0x03); //Enable rest of clocks
136 }

145 void LineinLineoutConfig() {
146
147     AudioWriteToReg(R17_CONVERTER_CONTROL_0, 0x06); //96 kHz
148     AudioWriteToReg(R64_SERIAL_PORT_SAMPLING_RATE, 0x06); //96 kHz
149     AudioWriteToReg(R19_ADC_CONTROL, 0x13);
150     AudioWriteToReg(R36_DAC_CONTROL_0, 0x03);
151     AudioWriteToReg(R35_PLAYBACK_POWER_MANAGEMENT, 0x03);
152     AudioWriteToReg(R58_SERIAL_INPUT_ROUTE_CONTROL, 0x01);
153     AudioWriteToReg(R59_SERIAL_OUTPUT_ROUTE_CONTROL, 0x01);
154     AudioWriteToReg(R65_CLOCK_ENABLE_0, 0x7F);
155     AudioWriteToReg(R66_CLOCK_ENABLE_1, 0x03);
156
157     AudioWriteToReg(R4_RECORD_MIXER_LEFT_CONTROL_0, 0x01);
158     AudioWriteToReg(R5_RECORD_MIXER_LEFT_CONTROL_1, 0x05); //0 dB gain
159     AudioWriteToReg(R6_RECORD_MIXER_RIGHT_CONTROL_0, 0x01);
160     AudioWriteToReg(R7_RECORD_MIXER_RIGHT_CONTROL_1, 0x05); //0 dB gain
161
162     AudioWriteToReg(R22_PLAYBACK_MIXER_LEFT_CONTROL_0, 0x21);
163     AudioWriteToReg(R24_PLAYBACK_MIXER_RIGHT_CONTROL_0, 0x41);
164     AudioWriteToReg(R26_PLAYBACK_LR_MIXER_LEFT_LINE_OUTPUT_CONTROL, 0x03); //0 dB
165     AudioWriteToReg(R27_PLAYBACK_LR_MIXER_RIGHT_LINE_OUTPUT_CONTROL, 0x09); //0 dB
166     AudioWriteToReg(R29_PLAYBACK_HEADPHONE_LEFT_VOLUME_CONTROL, 0xE7); //0 dB
167     AudioWriteToReg(R30_PLAYBACK_HEADPHONE_RIGHT_VOLUME_CONTROL, 0xE7); //0 dB
168     AudioWriteToReg(R31_PLAYBACK_LINE_OUTPUT_LEFT_VOLUME_CONTROL, 0xE6); //0 dB
169     AudioWriteToReg(R32_PLAYBACK_LINE_OUTPUT_RIGHT_VOLUME_CONTROL, 0xE6); //0 dB
170 }

```

程序的意思已经在程序中进行了必要的注释，具体的寄存器意义还请读者自行查阅 ADAU1761 的官方数据手册。

接下来看到 FFT 的配置部分，FFT 的配置位于 fft_audio.c 当中。首先看到这一段函数：

```
5 void audio_fre()
6 {
7     int i,j;
8     short dataIn[FFT_SIZE*4];
9     compx out[FFT_SIZE/2];
10    float mag[FFT_SIZE/2];
11
12    //===== DMA =====
13    XAxiDma axiDma;
14    int status;
15    status = init_dma(&axiDma);
16    if (status != XST_SUCCESS)
17    {
18        exit(-1);
19    }
20
21    printf("*-----start of test-----*\n\r");
22
23    //-----audio value-----//
24    get_audioData(dataIn);
25    for (i = 1; i < 4; i++)
26    {
27        for(j=0;j<FFT_SIZE;j++)
28        {
29            dataIn[j + i * FFT_SIZE]=dataIn[j];
30        }
31    }
32
33    //-----fft-----//
34    RealFFT_DMA(&axiDma, dataIn, out);
35    maxfre(out,mag);
36    oled_show(mag);
37
38    //----- end -----//
39    printf("*-----End of test-----*\n\r");
40 }
```

这一段函数相当于 FFT 配置函数的 main 函数功能。首先程序进行了一些初始化，之后对 DMA 进行了初始化。之后在看到 get_audioData 函数。其函数定义如下图所示：

```

103 void get_audioData(short* audio_data)
104 {
105     unsigned long u32Temp;
106     Xint32 i;
107     short in_data;
108     i = 0;
109     while (i<FFT_SIZE)           //128
110     {
111         do //wait for RX data to become available
112         {
113             u32Temp = Xil_In32(I2S_STATUS_REG);
114         } while (u32Temp == 0);
115
116         in_data = Xil_In32(I2S_DATA_RX_L_REG);
117         Xil_Out32(I2S_DATA_TX_L_REG, in_data);
118
119         *audio_data= in_data;//((in_data<<8)>>16);
120
121         Xil_Out32(I2S_STATUS_REG, 0x00000001); //Clear data ready bit
122         audio_data++;
123         i++;
124     }
125 }
126 }
```

这一段程序完成的是从 ADAU1761 中获取音频数据，这里需要注意的是获取到的数据要在 FFT_SIZE 这个范围内，这段程序实际上就是一个反复的写寄存器，然后读寄存器中的数据的操作。这里的 I2S_DATA_RX_L_REG 和 I2S_DATA_TX_L_REG 就是 ADAU1761 存放音频数据的寄存器，我们任取一个将鼠标停放在它们上面，然后按 F3 可查看其定义，如下图所示：

```

enum i2s_regs {
    I2S_DATA_RX_L_REG          = 0x00 + AUDIO_BASE,
    I2S_DATA_RX_R_REG          = 0x04 + AUDIO_BASE,
    I2S_DATA_TX_L_REG          = 0x08 + AUDIO_BASE,
    I2S_DATA_TX_R_REG          = 0x0c + AUDIO_BASE,
    I2S_STATUS_REG              = 0x10 + AUDIO_BASE,
};
```

在获取了音频的数据之后，通过一个 for 循环，将其数据量放在了一个四倍于其容量的数组当中，为接下来的处理做准备。然后看到 RealFFT() 函数，其定义如下：

```

201 void RealFFT_DMA(XAxiDma *InstancePtr, short *dataIn, compx * dataOut)
202 {
203     int i=0;
204     int status;
205
206     Xil_DCacheFlushRange((unsigned)dataIn, 4 * FFT_SIZE * sizeof(short)); //Refresh cache
207     status = XAxidma_SimpleTransfer(InstancePtr, (u32)dataIn, 4 * FFT_SIZE * sizeof(short), XAXIDMA_DMA_TO_DEVICE); //send enough data to DMA
208     for (i = 0; i < 2; i++)
209     {
210         // Setup DMA from PL to PS memory using AXI DMA's 'simple' transfer mode
211         status = XAxidma_SimpleTransfer(InstancePtr, (u32)dataOut,
212                                         FFT_SIZE / 2 * sizeof(compx), XAXIDMA_DEVICE_TO_DMA); //send result to PS
213         // Poll the AXI DMA core
214         do
215         {
216             status = XAxidma_Busy(InstancePtr, XAXIDMA_DEVICE_TO_DMA); //wait until transfer done
217         } while(status);
218
219         // Data cache must be invalidated for 'realspectrum' buffer after DMA
220         Xil_DCacheInvalidateRange((unsigned)dataOut, FFT_SIZE / 2 * sizeof(compx));
221
222         // DMA another frame of data to PL
223         if (!XAxidma_Busy(InstancePtr, XAXIDMA_DMA_TO_DEVICE)) //Continue sending data
224             status = XAxidma_SimpleTransfer(InstancePtr, (u32)dataIn, FFT_SIZE * sizeof(short), XAXIDMA_DMA_TO_DEVICE);
225
226         printf("-----End of frame-----\n\r");
227     }
228
229     fflush(stdout);
230 }
```

程序一开始，通过刷内存函数 Xil_DCacheFlushRange 将数据刷入了内存。紧接着启动了 DMA 传输，将一定量的数据传输给了 DMA。

```
// Setup DMA from PL to PS memory using AXI DMA's 'simple' transfer mode
status = XAxiDma_SimpleTransfer(InstancePtr, (u32)dataOut,
    FFT_SIZE / 2 * sizeof(compx), XAXIDMA_DEVICE_TO_DMA);           //send result to PS
// Poll the AXI DMA core
do
{
    status = XAxiDma_Busy(InstancePtr, XAXIDMA_DEVICE_TO_DMA);        //wait until transfer done
}
while(status);
```

接下来的这一句用了一个 do while 语句，其目的是为了确保 DMA 传输完成，这一段完成了一个向 PS 发送数据的功能，发送的数据也就是 dataOut，也就是 FFT 处理之后的结果。之后再看到这一句：

```
// Data cache must be invalidated for 'realspectrum' buffer after DMA
Xil_DCacheInvalidateRange((unsigned)dataOut,FFT_SIZE / 2 * sizeof(compx));
```

Xil_DcacheInvalidateRange() 函数将一段指定长度的高速数据缓冲器无效，这里即为在整段处理的结果传送完成之后，再释放这一段缓冲区，为下一次数据传输做准备。接下来的这一句函数就是启动传输下一段数据传输，如下图所示：

```
// DMA another frame of data to PL
if (!XAxiDma_Busy(InstancePtr, XAXIDMA_DMA_TO_DEVICE))//Continue sending data
status = XAxiDma_SimpleTransfer(InstancePtr, (u32)dataIn,FFT_SIZE * sizeof(short), XAXIDMA_DMA_TO_DEVICE);
```

这个函数整体下来就是一个数据传输过来，然后再将结果传到 PS，然后准备下一次传输的过程。回到 audio_fre() 函数的分析当中，接下来看到 maxfre 函数，其函数定义如下所示：

```
128 void maxfre(compx *x, float *mag)
129 {
130     printf("maxfre in \r\n");
131     int i, n=1;
132     float tmp, maxf;
133     printf("maxfre init \r\n");
134     for (i=0; i<FFT_SIZE/2; i++)          //get the magnitude
135     {
136         float real =(float) x[i].real;
137         float imag =(float) x[i].imag;
138         mag[i]=sqrtf(real * real + imag * imag);
139     }
140     printf("maxfre mag done \r\n");
141     tmp=mag[1];                         //first is DC component
142     for(i=1; i<FFT_SIZE/2; i++)          //get the max mag & its freq
143     {
144         if(tmp<mag[i])
145         {
146             tmp=mag[i];
147             n=i;
148         }
149     }
150     }
151     printf("maxfre compare done \r\n");
152     maxf=1.0*n*FFT_SIZE/FFT_SIZE;
153     printf("Cur n=%d;maxf= %f Hz;max mag=%f;\r\n",n,maxf,tmp);
154 }
```

这个程序完成的计算 FFT 频谱的频率的功能。这里面的计算是一种特定的计算方法，mag 是 FFT 的幅度值，它的值为 FFT 的实部和虚部的平方和。这里的第一个 for 循环也就是求出输入的所有数据的幅度，然后由第二个 for 循环找出其中最大的幅度值，并由这个幅度值求出其频率。

10.5 本章小结

本章向大家介绍了利用第八章生成的 FFT IP 设计了一个实时音乐频谱显示的例子，FFT 在我们的设计过程中非常常见，尤其是在无线通信领域应用非常广泛，传统的 FFT 实现方式十分的繁琐，而利用 HLS 设计则非常的节省时间而且不是十分复杂，其实用性不言而喻。

S05_CH11_快速角点检测的 HLS 实现

11.1 角点定义

角点检测(Corner Detection)是计算机视觉系统中用来获得图像特征的一种方法，广泛应用于运动检测、图像匹配、视频跟踪、三维建模和目标识别等领域中。也称为特征点检测。

角点通常被定义为两条边的交点，更严格的说，角点的局部邻域应该具有两个不同区域的不同方向的边界。而实际应用中，大多数所谓的角点检测方法检测的是拥有特定特征的图像点，而不仅仅是“角点”。这些特征点在图像中有具体的坐标，并具有某些数学特征，如局部最大或最小灰度、某些梯度特征等。

现有的角点检测算法并不是都十分的鲁棒。很多方法都要求有大量的训练集和冗余数据来防止或减少错误特征的出现。角点检测方法的一个很重要的评价标准是其对多幅图像中相同或相似特征的检测能力，并且能够应对光照变化、图像旋转等图像变化。

在我们解决问题时，往往希望找到特征点，“特征”顾名思义，指能描述物体本质的东西，还有一种解释就是这个特征微小的变化都会对物体的某一属性产生重大的影响。而角点就是这样的特征。

观察日常生活中的“角落”就会发现，“角落”可以视为所有平面的交汇处，或者说是所有表面的发起处。假设我们要改变一个墙角的位置，那么由它而出发的平面势必都要有很大的变化。所以，这就引出了图像角点的定义，“如果某一点在任意方向的一个微小变动都会引起灰度很大的变化，那么我们就把它称之为角点”

特征检测与匹配是 Computer Vision 应用中重要的一部分，这需要寻找图像之间的特征建立对应关系。点，也就是图像中的特殊位置，是很常用的一类特征，点的局部特征也可以叫做“关键特征点”(keypoint feature)，或“兴趣点”(interest point)，或“角点”(corner)。

关于角点的具体描述可以有几种：

1. 一阶导数(即灰度的梯度)的局部最大所对应的像素点；
2. 两条及两条以上边缘的交点；
3. 图像中梯度值和梯度方向的变化速率都很高的点；
4. 角点处的一阶导数最大，二阶导数为零，指示物体边缘变化不连续的方向。

11.2 角点检测算法

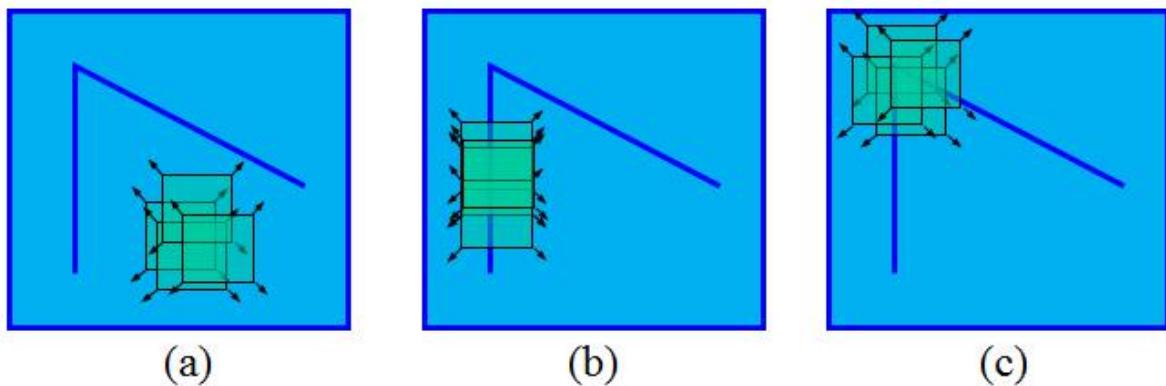
11.2.1 Moravec 角点检测算法

Moravec 角点检测算法是最早的角度检测算法之一。该算法将角点定义为具有低“自相关性”的点。算法会检测图像的每一个像素，将像素周边的一个邻域作为一个 patch，并检测这个 patch 和周围其他 patch 的相关性。这种相关性通过两个 patch 间的平方差之和 (SSD) 来衡量，SSD 值越小则相似性越高。

如果像素位于平滑图像区域内，周围的 patch 都会非常相似。如果像素在边缘上，则周围的 patch 在与边缘正交的方向上会有很大差异，在与边缘平行的方向上则较为相似。而如果像素是各个方向上都有变化的特征点，则周围所有的 patch 都不会很相似。Moravec 会计算每个像素 patch 和周围 patch 的 SSD 最小值作为强度值，取局部强度最大的点作为特征点。

11.2.2 Harris 角点检测

当一个窗口在图像上移动，在平滑区域如图(a)，窗口在各个方向上没有变化。在边缘上如图(b)，窗口在边缘的方向上没有变化。在角点处如图(c)，窗口在各个方向上具有变化。Harris 角点检测正是利用了这个直观的物理现象，通过窗口在各个方向上的变化程度，决定是否为角点。



将图像窗口平移 $[u, v]$ 产生灰度变化 $E(u, v)$

$$E(u, v) = \sum_{x,y} w(x, y) [I(x+u, y+v) - I(x, y)]^2$$

由： $I(x+u, y+v) = I(x, y) + I_x u + I_y v + O(u^2, v^2)$ ， 得到：

$$E(u, v) = [u, v] \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

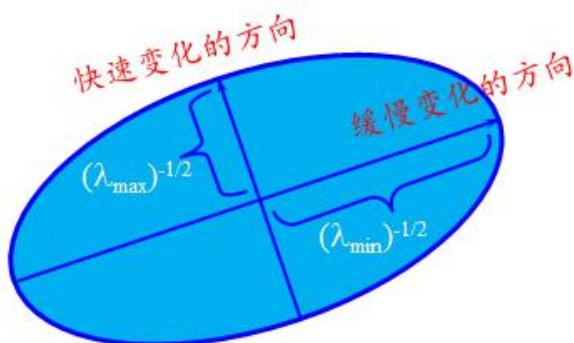
对于局部微小的移动量 $[u, v]$ ，近似表达为：

$$E(u, v) \approx [u, v] M \begin{bmatrix} u \\ v \end{bmatrix}$$

其中 M 是 2×2 矩阵，可由图像的导数求得：

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

$E(u, v)$ 的椭圆形式如下图：



定义角点响应函数 R 为：

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

Harris 角点检测算法就是对角点响应函数 R 进行阈值处理： $R > \text{threshold}$ ，即提取 R 的局部极大值。

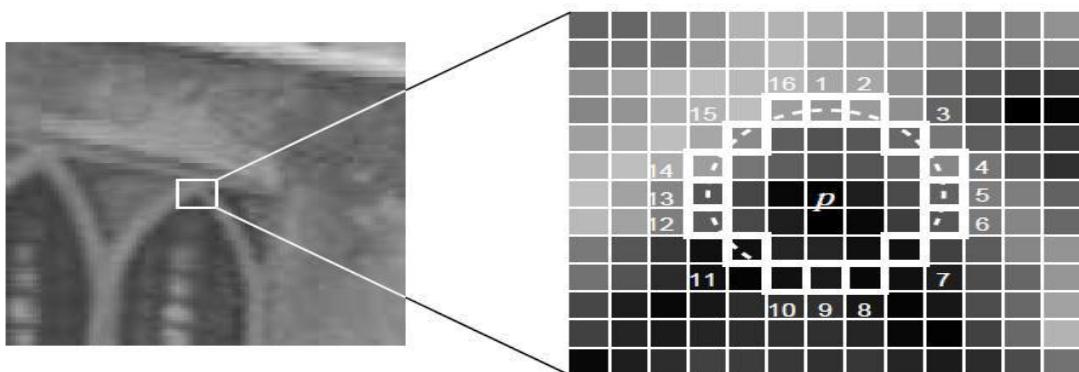
11.2.3 FAST 角点检测算法

Smith 和 Brady 在 1997 年提出了一种完全不同的角点提取方法，即“SUSAN (Smallest Univalued Segment Assimilating Nucleus)” 提取算子。SUSAN 提取算子的基本原理是，与每一图像点相关的局部区域具有相同的亮度。如果某一窗口区域内的每一像元亮度值与该窗口中心的像元亮度值相同或相似，这一窗口区域将被称之为“USAN”。计算图像每一像元的“USAN”，为我们提供了是否有边缘的方法。位于边缘上的像元的“USAN”较小，位于角点上的像元的“USAN”更小。因此，我们仅需寻找最小的“USAN”，就可确定角点。该方法由于不需要计算图像灰度差，因此，具

有很强的抗噪声的能力。

Edward Rosten and Tom Drummond 在 2006 年提出了一种简单快速的角度探测算法，该算法检测的角度定义为在像素点的周围邻域内有足够的像素点与该点处于不同的区域。应用到灰度图像中，即有足够的像素点的灰度值大于该点的灰度值或者小于该点的灰度值。

考虑下图中 p 点附近半径为 3 的圆环上的 16 个点，一个思路是若其中有连续的 12 个点的灰度值与 p 点的灰度值差别超过某一阈值，则可以认为 p 点为角点。



这一思路可以使用机器学习的方法进行加速。对同一类图像，例如同一场景的图像，可以在 16 个方向上进行训练，得到一棵决策树，从而在判定某一像素点是否为角点时，不再需要对所有方向进行检测，而只需要按照决策树指定的方向进行 2-3 次判定即可确定该点是否为角点。

在本章节的 HLS 实现中我们主要介绍 FAST 角点检测，很多传统的算法都很耗时，而且特征点检测算法只是很多复杂图像处理里中的第一步，得不偿失。FAST 特征点检测是公认的比较快速的特征点检测方法，只利用周围像素比较的信息就可以得到特征点，简单，有效。

FAST 特征检测算法来源于 corner 的定义，这个定义基于特征点周围的图像灰度值，检测候选特征点周围一圈的像素值，如果候选点周围领域内有足够的像素点与该候选点的灰度值差别够大，则认为该候选点为一个特征点。

$$N = \sum_{x \in \text{circle}(p)} |I(x) - I(p)| > \epsilon_d \quad (1)$$

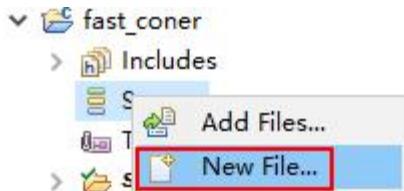
其中 $I(x)$ 为圆周上任意一点的灰度， $I(p)$ 为圆心的灰度， ϵ_d 为灰度值差得阈值，如果 N 大于给定阈值，一般为周围圆圈点的四分之三，则认为 p 是一个特征点。

11.3 HLS 实现

11.3.1 工程创建

Step1：打开 HLS，按照之前介绍的方法，创建一个新的工程，命名为 fast_corner。

Step2：右单击 Source 选项，选择 New File，创建一个名为 Top.cpp 的文件。



Step3: 在打开的编辑区中，把下面的程序拷贝进去：

```
#include "top.h"

void hls_fast_corner(AXI_STREAM& INPUT_STREAM, AXI_STREAM&
OUTPUT_STREAM, int rows, int cols, int threshold)
{
    //Create AXI streaming interfaces for the core
#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM

#pragma HLS RESOURCE core=AXI_SLAVE variable=rows
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cols
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=threshold
metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=return
metadata="-bus_bundle CONTROL_BUS"

#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols
#pragma HLS INTERFACE ap_stable port=threshold

RGB IMAGE      _src(rows,cols);
RGB IMAGE      _dst(rows,cols);
RGB IMAGE      src0(rows,cols);
RGB IMAGE      src1(rows,cols);
GRAY IMAGE     mask(rows,cols);
```

```

GRAY IMAGE      dmask(rows,cols);
GRAY IMAGE      gray(rows,cols);

#pragma HLS dataflow
#pragma HLS stream depth=20000 variable=src1.data_stream
hls::AXIvideo2Mat(INPUT_STREAM, _src);
hls::Scalar<3,unsigned char> color(255,0,0);
hls::Duplicate(_src,src0,src1);
hls::CvtColor<HLS_BGR2GRAY>(src0,gray);
hls::FASTX(gray,mask,threshold,true);
hls::Dilate(mask,dmask);
hls::PaintMask(src1,dmask,_dst,color);
hls::Mat2AXIvideo(_dst, OUTPUT_STREAM);
}

```

Step4: 再在 Source 中添加一个名为 Top.h 的库函数，并添加如下程序：

```

#ifndef _TOP_H_
#define _TOP_H_

#include "hls_video.h"

// maximum image size
#define MAX_WIDTH 1920
#define MAX_HEIGHT 1080

// I/O Image Settings
#define INPUT_IMAGE          "test_1080p.bmp"
#define OUTPUT_IMAGE          "result.bmp"
#define OUTPUT_IMAGE_GOLDEN   "result_golden.bmp"

// typedef video library core structures
typedef hls::stream<ap_axiu<32,1,1,1> >           AXI_STREAM;
typedef hls::Scalar<3, unsigned char>                  RGB_PIXEL;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3>     RGB_IMAGE;
typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1>     GRAY_IMAGE;

// top level function for HW synthesis
void hls_fast_corner(AXI_STREAM& src_axi, AXI_STREAM& dst_axi, int
rows, int cols, int threshold);

#endif

```

Step5: 在 Test Bench 中，用同样的方法添加一个名为 Test.cpp 的测试程序。添加如下代码：

```
#include "hls_opencv.h"
#include "top.h"

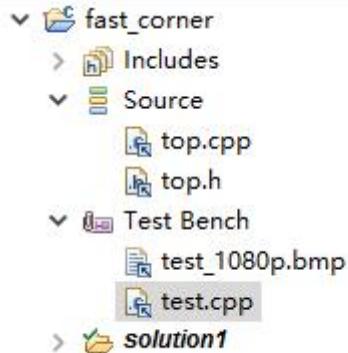
using namespace cv;

int main (int argc, char** argv) {
    IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth,
src->nChannels);
    cvShowImage("hls_src", src);

    //HLS视频库处理
    AXI_STREAM src_axi, dst_axi;
    IplImage2AXIVideo(src, src_axi);
    hls_fast_corner(src_axi, dst_axi, src->height, src->width, 20);
    AXIVideo2IplImage(dst_axi, dst);
    cvShowImage("hls_dst", dst);
    cvSaveImage(OUTPUT_IMAGE,dst);
    waitKey(0);

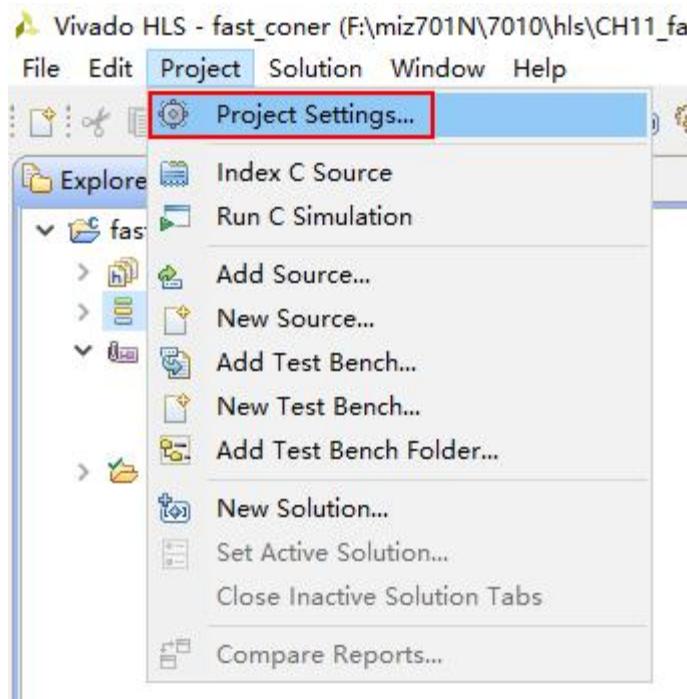
    return 0;
}
```

Step6: 在 Test Bench 中添加一张名为 test_1080P.bmp 的测试图片，图片可以在我们提供的源程序中的 Image 文件夹中找到。完整的工程如下图所示：

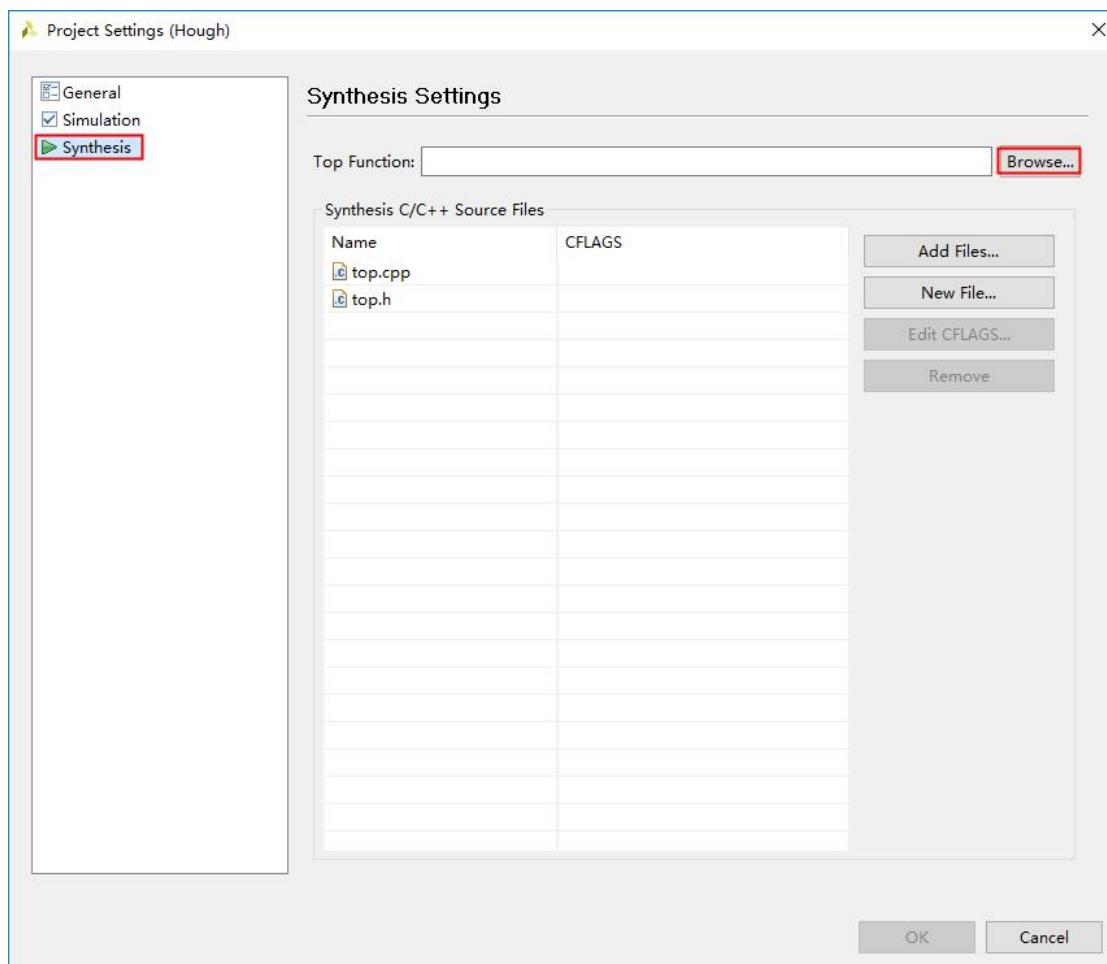


11.3.2 仿真及优化

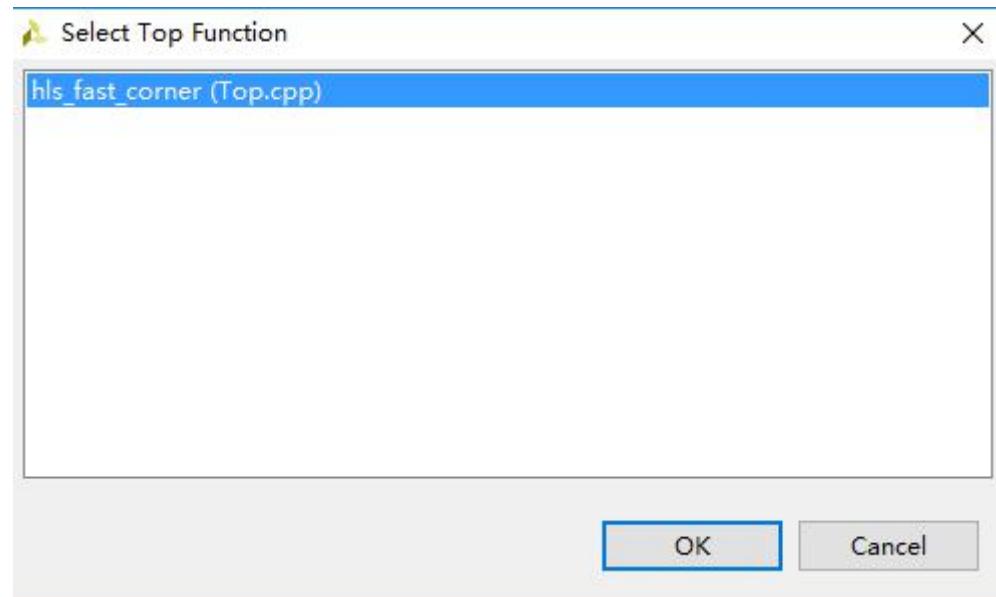
Step1：单击 Project→Project settings。



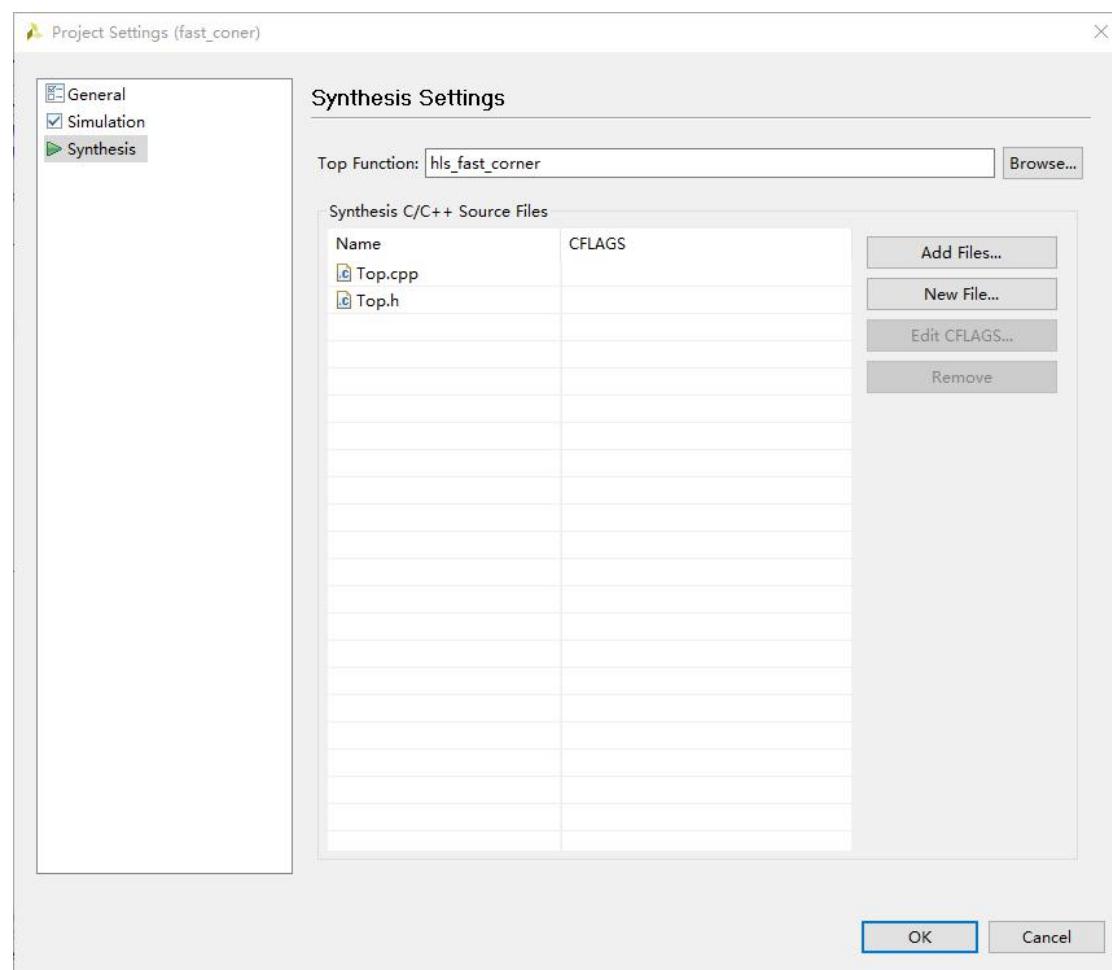
Step2: 选择 Synthesis 选项，然后点击 Browse.. 指定一个顶层函数。



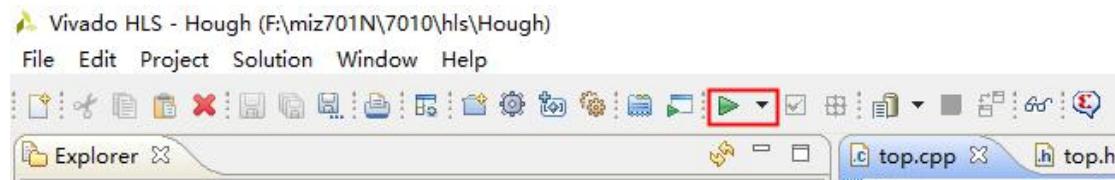
Step3：选定 hls_fast_corner 为顶层函数，然后点击 O K。



Step4：再次单击 OK，完成工程的修改。



Step5: 单击 开始综合。



综合报告如下：

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	11.12	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
210	2111641	183	2111634	dataflow

Detail

Instance

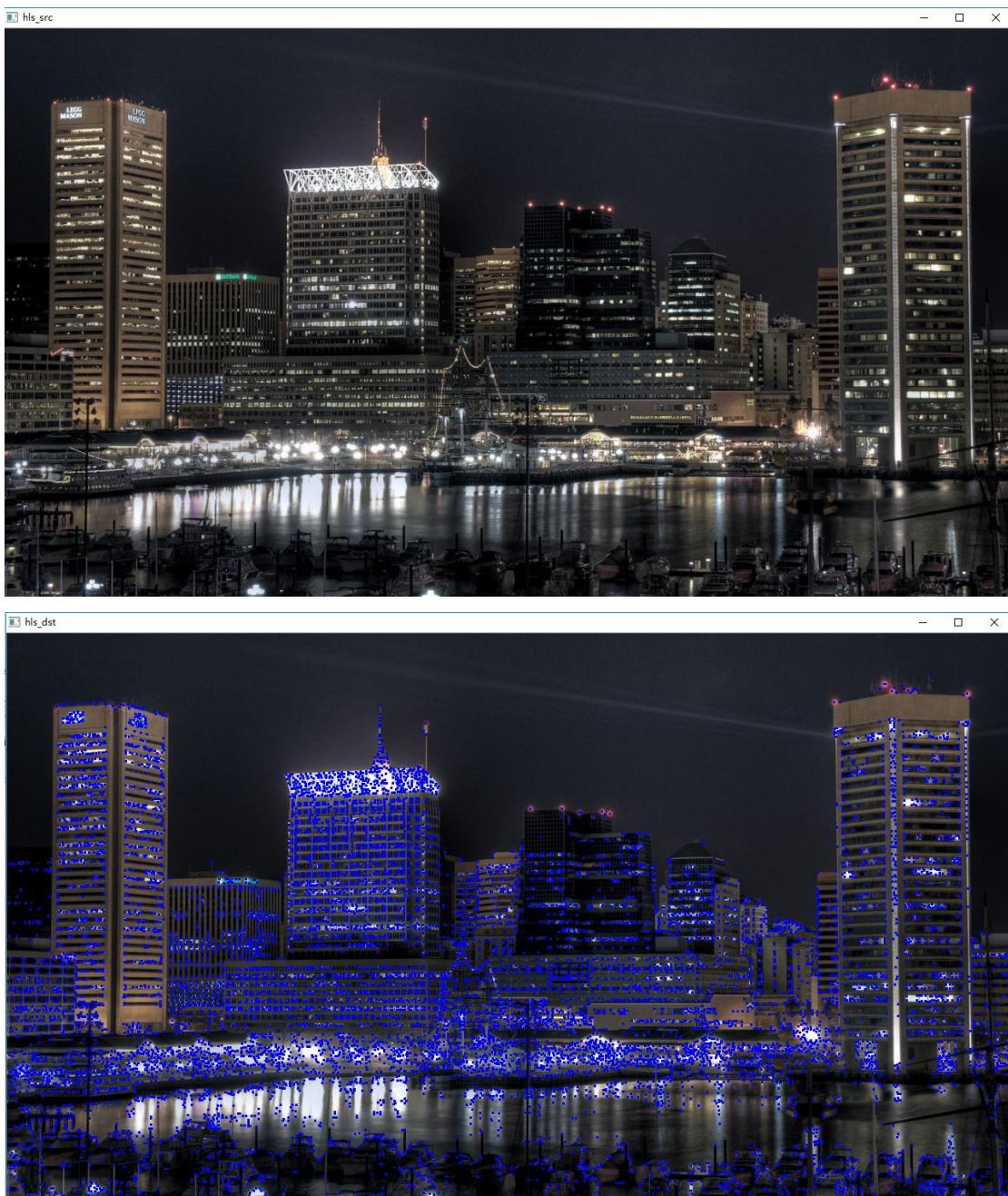
Loop

Utilization Estimates

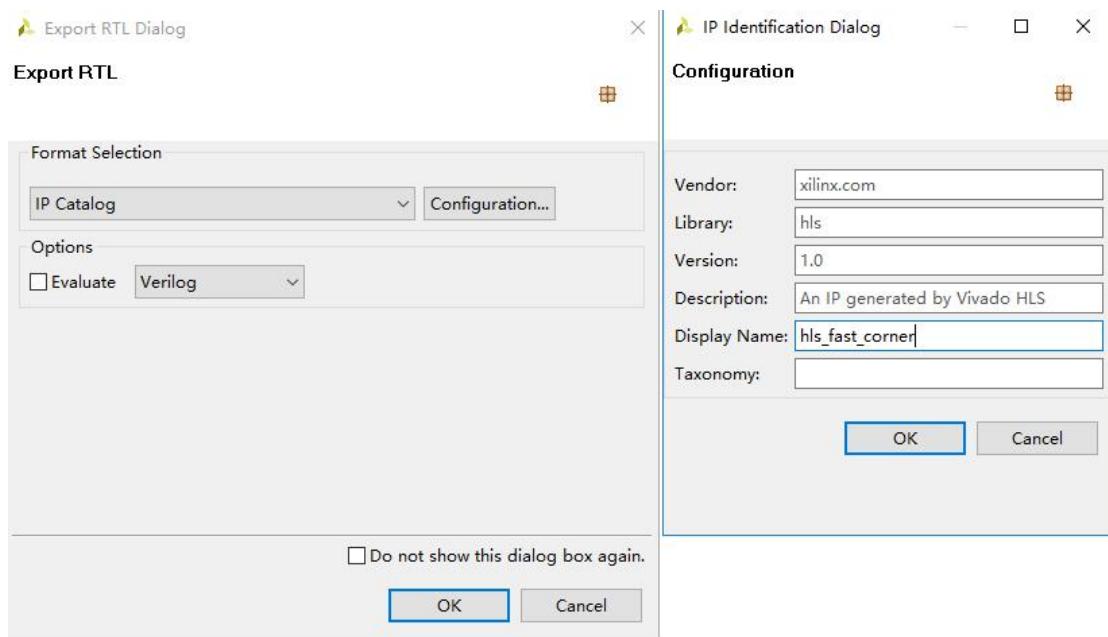
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1
FIFO	48	-	708	2334
Instance	11	3	5208	5867
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	8	-
Total	59	3	5924	8202
Available	120	80	35200	17600
Utilization (%)	49	3	16	46

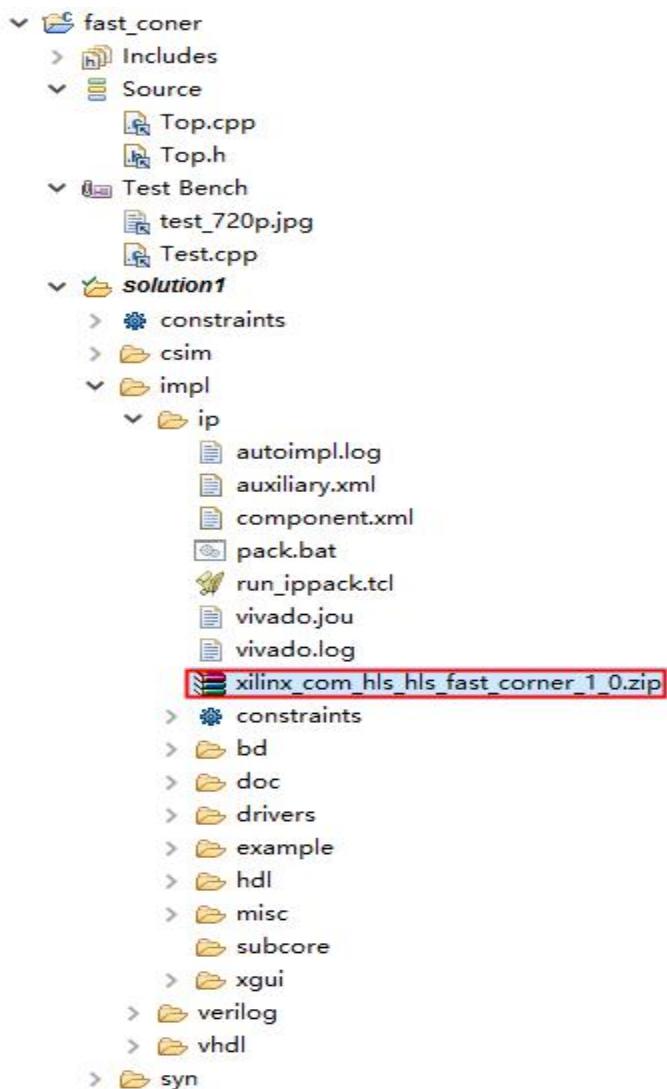
Step6: 直接单击 开始进行仿真。系统运行结束后，处理结果如下所示：



Step7:单击 导出供 VIVADO 使用的 IP，然后在弹出来的新窗口中按下图设置。



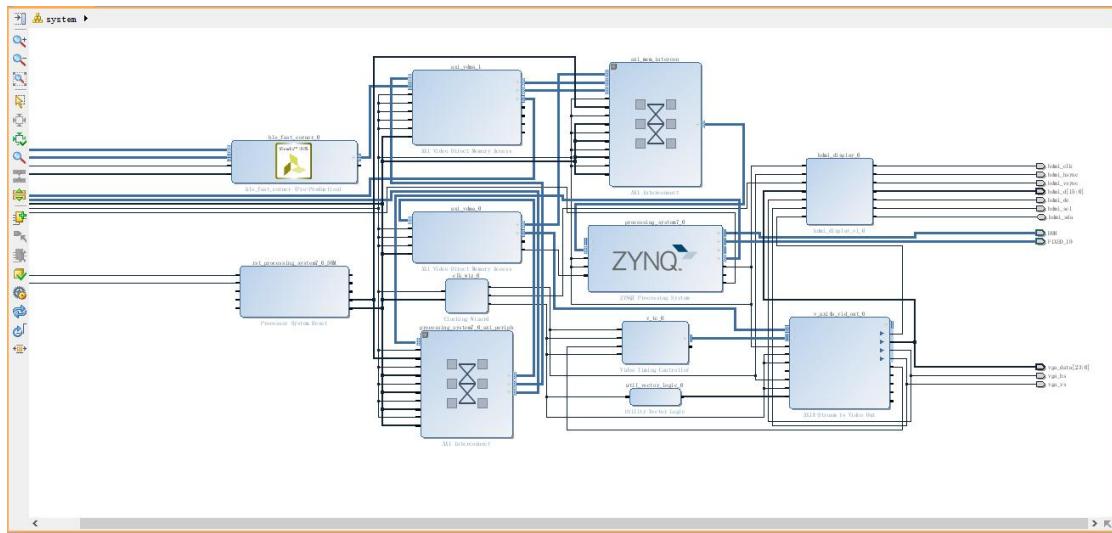
综合完成之后可以看到系统生成的 IP。



11.4 硬件工程创建

11.4.1 硬件平台搭建

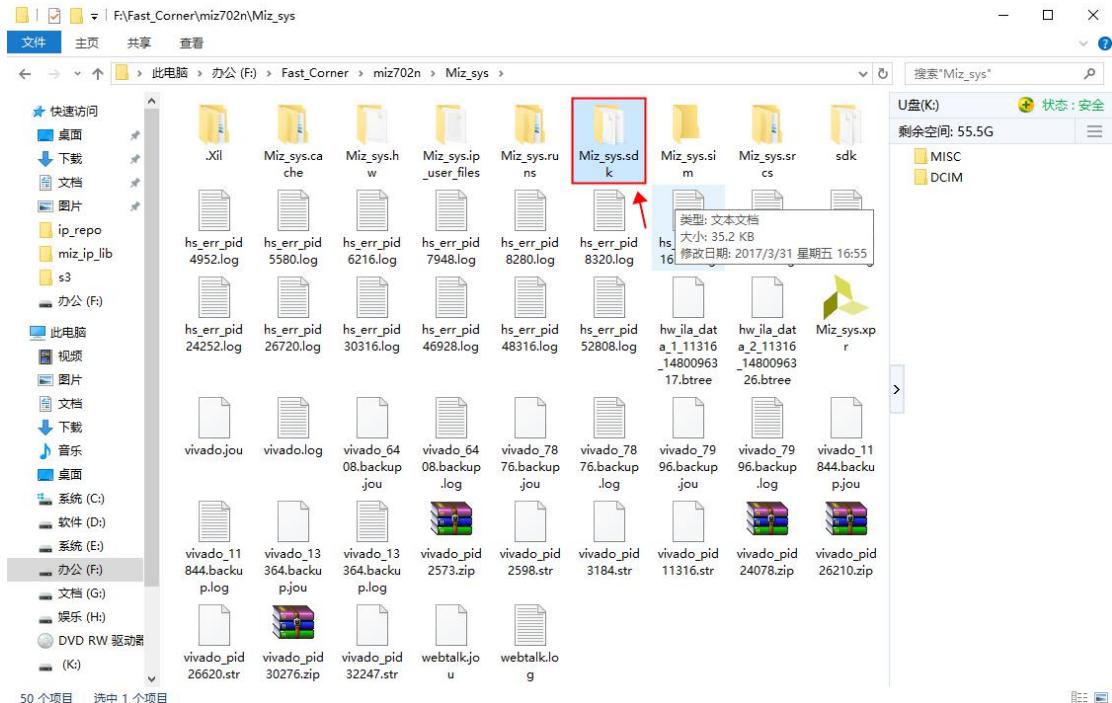
本章的硬件平台与 sobel 算子的实现几乎是一样的，只是在这里需要把 sobel 实现的 HLS IP 替换成角点检测的 IP 。系统整体硬件电路如下图所示：



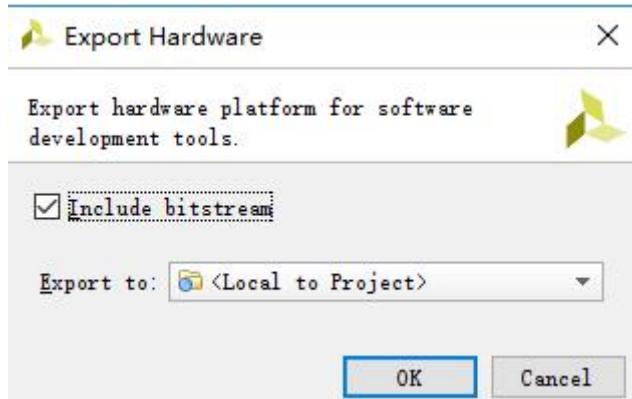
11.4.2 导入到 SDK

硬件平台搭建完成之后，让工程重新生成 Bit 文件，之后记得删除之前的 SDK 工程，以便 HLS IP 的驱动文件能正确的产生。

Step1: 在工程文件夹中删除原来的工程的文件夹。



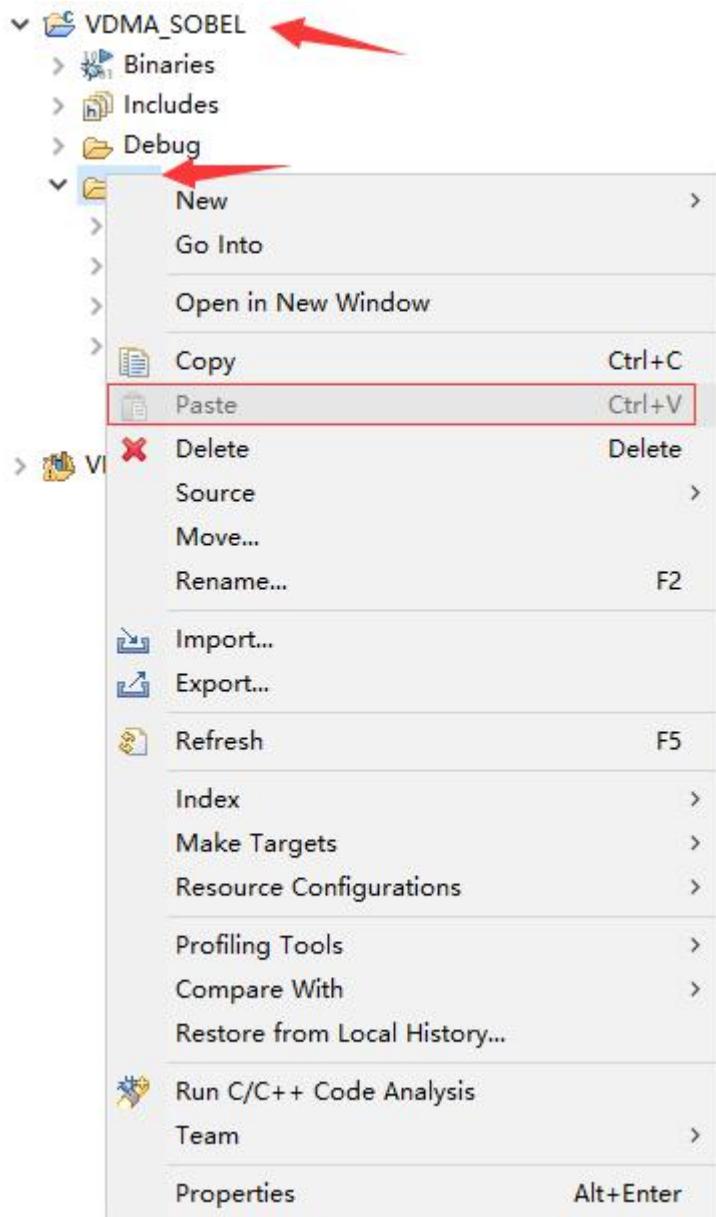
Step2: 单击 File-Export-Export Hardware...。



Step3: 单击 File-Launch SDK。

Step4: 新建一个名为 Fast_Corner 的空白工程。

Step5: 复制 S05_CH06 的 SDK 驱动文件, 然后在 SDK 中, 选中工程, 在 Src 下直接按 Ctrl +V 将其复制到工程中来。



Step6: 双击 main.c，用如下程序进行替换。

```
#include "xaxivdma.h"
#include "xaxivdma_i.h"
#include "xhls_fast_corner.h"
#include "sleep.h"

#define DDR_BASEADDR      0x00000000
#define DISPLAY_VDMA      XPAR_AXI_VDMA_0_BASEADDR + 0
#define SOBEL_VDMA        XPAR_AXI_VDMA_1_BASEADDR + 0
#define DIS_X              1280
#define DIS_Y              720
#define SOBEL_ROW          512
#define SOBEL_COL          512
```

```
#define pi 3.14159265358
#define COUNTS_PER_SECOND (XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ)/64

#define SOBEL_S2MM 0x08000000
#define SOBEL_MM2S 0x0A000000
#define DISPLAY_MM2S 0x0C000000

u32 *BufferPtr[3];
u32 threshold=20;
static XHls_fast_corner fast;

//函数声明
void Xil_DCacheFlush(void);
// 所有数据格式 为 RGBA, 低位的透明度暂不起作用
extern const unsigned char gImage_lena[1048584];
extern const unsigned char gImage_logo[284008];

void SOBEL_VDMA_setting(unsigned int width,unsigned int height,unsigned int s2mm_addr,unsigned int mm2s_addr)
{
    //S2MM
    Xil_Out32(SOBEL_VDMA + 0x30, 0x4); //reset S2MM VDMA Control Register
    usleep(10);
    Xil_Out32(SOBEL_VDMA + 0x30, 0x0); //genlock
    Xil_Out32(SOBEL_VDMA + 0xAC, s2mm_addr); //S2MM Start Addresses
    Xil_Out32(SOBEL_VDMA + 0xAC+4, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xAC+8, s2mm_addr);
    Xil_Out32(SOBEL_VDMA + 0xA4, width*4); //S2MM Horizontal Size
    Xil_Out32(SOBEL_VDMA + 0xA8, width*4); //S2MM Frame Delay and Stride
    Xil_Out32(SOBEL_VDMA + 0x30, 0x3); //S2MM VDMA Control Register
    Xil_Out32(SOBEL_VDMA + 0xA0, height); //S2MM Vertical Size start an S2M
    // Xil_DCacheFlush();

    //MM2S
    Xil_Out32(SOBEL_VDMA + 0x00,0x00000003); // enable circular mode
    Xil_Out32(SOBEL_VDMA + 0x5c,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x60,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x64,mm2s_addr); // start address
    Xil_Out32(SOBEL_VDMA + 0x58,(width*4)); // h offset
    Xil_Out32(SOBEL_VDMA + 0x54,(width*4)); // h size
```

```
Xil_Out32(SOBEL_VDMA + 0x50,height); // v size
//Xil_DCacheFlush();
}

void DISPLAY_VDMA_setting(unsigned int width,unsigned height,unsigned int mm2s_addr)
{
    Xil_Out32((DISPLAY_VDMA + 0x000), 0x00000003);      // enable
circular mode
    Xil_Out32((DISPLAY_VDMA + 0x05c), mm2s_addr); // start address
    Xil_Out32((DISPLAY_VDMA + 0x060), mm2s_addr); // start address
    Xil_Out32((DISPLAY_VDMA + 0x064), mm2s_addr); // start address
    Xil_Out32((DISPLAY_VDMA + 0x058), (width*4)); // h offset
(640 * 4) bytes
    Xil_Out32((DISPLAY_VDMA + 0x054), (width*4)); // h size
(640 * 4) bytes
    Xil_Out32((DISPLAY_VDMA + 0x050), height); // v size
(480)
}

void SOBEL_DDRWR(unsigned int addr,unsigned int cols,unsigned int lows)
{
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    for(i=0;i<cols;i++)
    {
        for(j=0;j<lows;j++)
        {
            b= gImage_lena[(j+i*cols)*4+2];
            g= gImage_lena[(j+i*cols)*4+1];
            r= gImage_lena[(j+i*cols)*4];
            Xil_Out32((addr+(j+i*cols)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
        }
    }
    Xil_DCacheFlush();
}

void SOBEL_Setup()
{
    //const int cols = 512;
    //const int rows = 512;
    XHls_fast_corner_SetRows(&fast, SOBEL_COL);
```

```
XHls_fast_corner_SetCols(&fast, SOBEL_ROW);
XHls_fast_corner_SetThrehold(&fast, threshold);
XHls_fast_corner_DisableAutoRestart(&fast);
XHls_fast_corner_InterruptGlobalDisable(&fast);
SOBEL_VDMA_setting(SOBEL_ROW, SOBEL_COL, SOBEL_S2MM, SOBEL_MM2S);
SOBEL_DDRWR(SOBEL_MM2S, SOBEL_ROW, SOBEL_COL);
//init_hls_sobel_dma(cols,rows, VIDEO_BASEADDR,
HLS_VDMA_MM2S_ADDR);
//DDRVideoWr(HLS_VDMA_MM2S_ADDR, cols,rows);
XHls_fast_corner_Start(&fast);
}

void Set_background(u32 size_x,u32 size_y,u32 disp_addr)
{
    u32 i=0;
    u32 j=0;
    //u32 r,g,b;
    for(j=0;j<size_y;j++)
    {
        for(i=0;i<size_x;i++)
        {
            Xil_Out32((disp_addr+(i+j*size_x)*4),0);
        }
    }
    Xil_DCacheFlush();
}

void show_img(u32 x, u32 y, u32 disp_base_addr, const unsigned char * addr,
u32 size_x, u32 size_y)
{
    //计算图片 左上角坐标
    u32 i=0;
    u32 j=0;
    u32 r,g,b;
    u32 start_addr=disp_base_addr;
    start_addr = disp_base_addr + 4*x + y*4*DIS_X;
    for(j=0;j<size_y;j++)
    {
        for(i=0;i<size_x;i++)
        {
            //if(type==0)
            //{
                //b = *(addr+(i+j*size_x)*4+2); //08
            }
        }
    }
}
```

```
//g = *(addr+(i+j*size_x)*4+1); //60
//r = *(addr+(i+j*size_x)*4); //01
//}
//else
//{
    b = *(addr+(i+j*size_x)*4); //08
    g = *(addr+(i+j*size_x)*4+1); //60
    r = *(addr+(i+j*size_x)*4+2); //01
}

Xil_Out32((start_addr+(i+j*DIS_X)*4),((r<<16)|(g<<8)|(b<<0)|0x0));
}
}

int main(void)
{
    //Xil_DCacheFlush();
    xil_printf("Starting the first VDMA \n\n");
    int status = XHls_fast_corner_Initialize(&fast,
XPAR_HLS_FAST_CORNER_0_S_AXI_CONTROL_BUS_BASEADDR);
    if(0 != status)
    {
        xil_printf("XHls_Sobel_Initialize failed \n");
    }
    SOBEL_Setup();
    DISPLAY_VDMA_setting(DIS_X,DIS_Y,DISPLAY_MM2S);
    Set_blackground(1280,720,DISPLAY_MM2S);
    ****
    for(i=0;i<614400;i++)
    {
        Xil_Out32(VIDEO_BASEADDR0+i,0);
    }
    ****
    while(1)
    {
        //show_img(0,0,VIDEO_BASEADDR0,&gImage_beauty[0],563,600);
        //sleep(5);

        //show_img(0,0,VIDEO_BASEADDR0,&gImage_miz702_rgba[0],375,400);
        //sleep(5);
    }
}
```

```

show_img(0,0,DISPLAY_MM2S,(void*)SOBEL_S2MM,512,512);
show_img(522,0,DISPLAY_MM2S,(void*)SOBEL_MM2S,512,512);
//show_img(0,513,DISPLAY_MM2S,&gImage_logo[0],355,200);

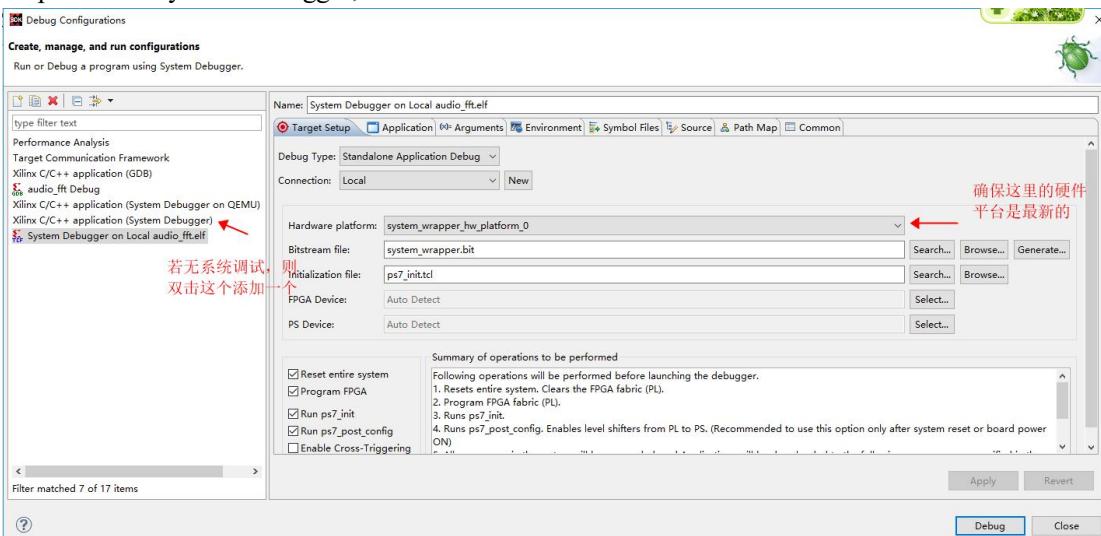
}

return 0;
}

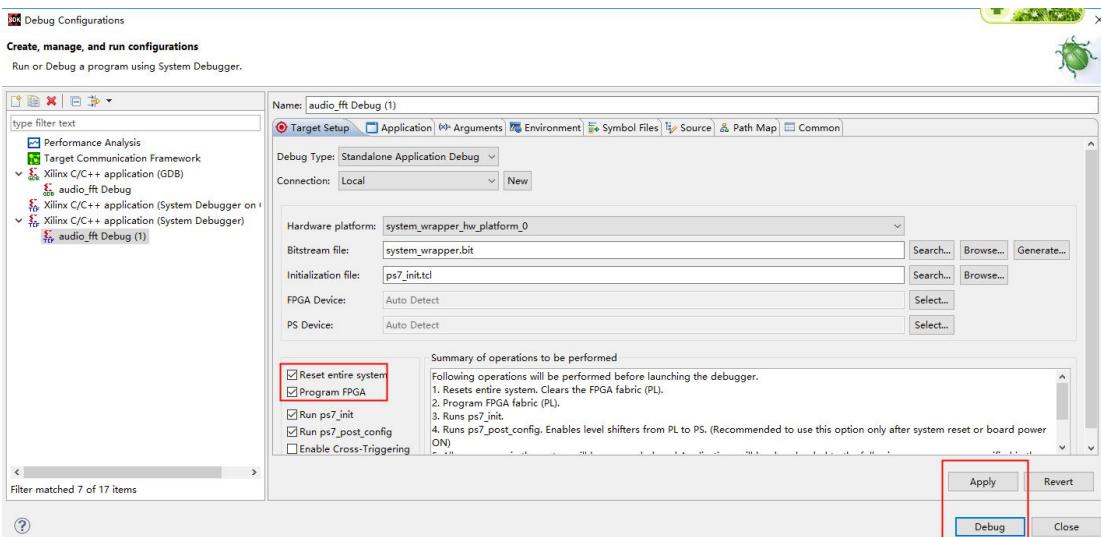
```

Step8: 右击工程, 选择 Debug as ->Debug configuration。

Step9: 选中 system Debugger, 双击创建一个系统调试。



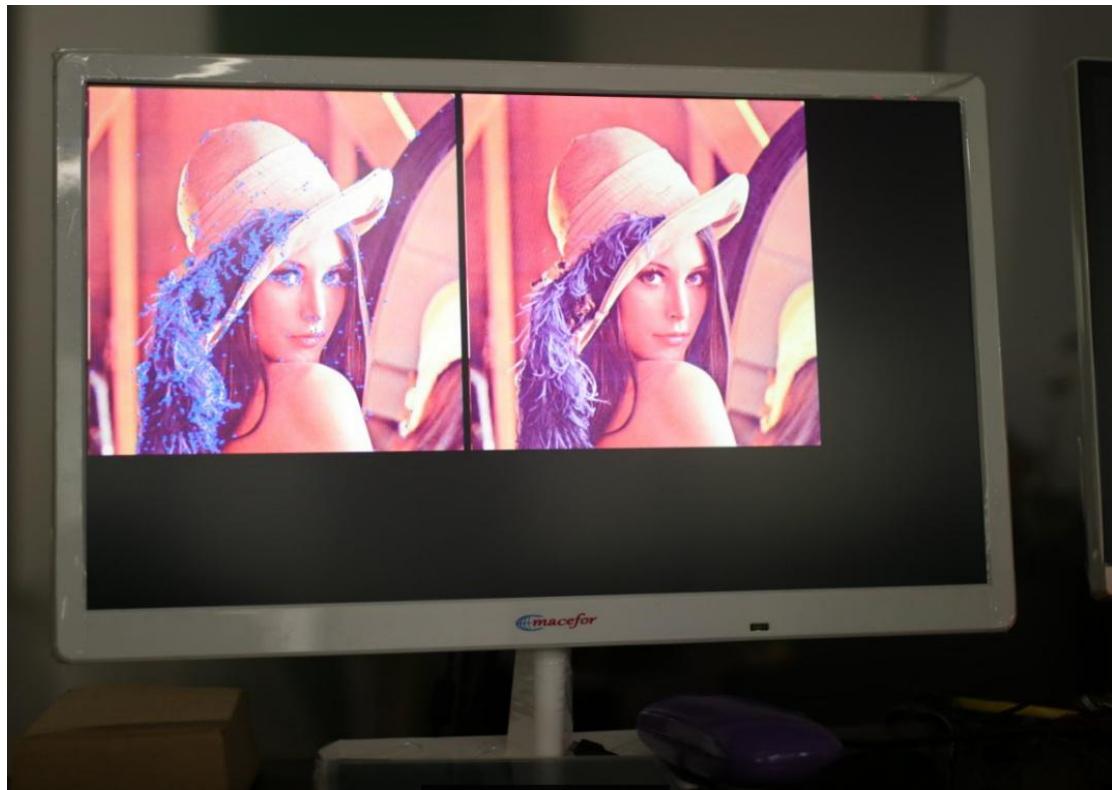
Step10: 设置系统调试。



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。



程序运行之后, 运行结果如下图所示:



11.5 程序分析

本章节的程序与第六章 SOBEL 的程序基本上是一致的，这是因为 HLS 产生的 IP 大多驱动方式是一样的，本章程序中快速角点检测 IP 的驱动函数如下图所示：

```
//const int cols = 512;
//const int rows = 512;
XHls_fast_corner_SetRows(&fast, SOBEL_COL);
XHls_fast_corner_SetCols(&fast, SOBEL_ROW);
XHls_fast_corner_SetThreshold(&fast, threshold);
XHls_fast_corner_DisableAutoRestart(&fast);
XHls_fast_corner_InterruptGlobalDisable(&fast);
SOBEL_VDMA_setting(SOBEL_ROW, SOBEL_COL, SOBEL_S2MM, SOBEL_MM2S);
SOBEL_DDRWR(SOBEL_MM2S, SOBEL_ROW, SOBEL_COL);
//init_hls_sobel_dma(cols,rows, VIDEO_BASEADDR, HLS_VDMA_MM2S_ADDR);
//DDRVideoWr(HLS_VDMA_MM2S_ADDR, cols,rows);
XHls_fast_corner_Start(&fast);
```

可以看出，这与第六章的程序相比较只是函数名改了个名字而已，驱动方式是一样的，大家在以后的设计中就可以仿照这里对 HLS 生成的 IP 进行驱动

11.6 本章小结

本章向大家介绍了如何利用 HLS 来实现快速角点检测算法，快速角点检测可以用于特征提取，因此具有很大的实用性，最后利用我们手头上的 MIZ 系列开发板对其进行了验证，通过本章，大家应该掌握利用 HLS 来进行基本的算法开发，这也大大节省我们的时间。

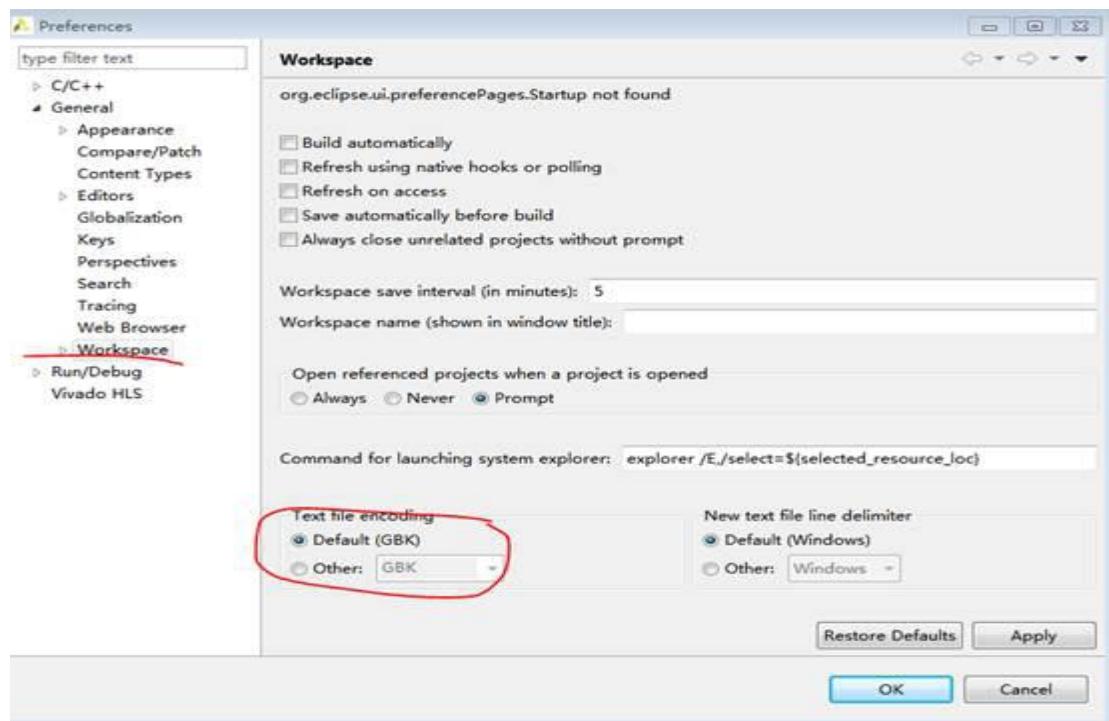
S05_CH12_HLS 车牌识别（待更新）

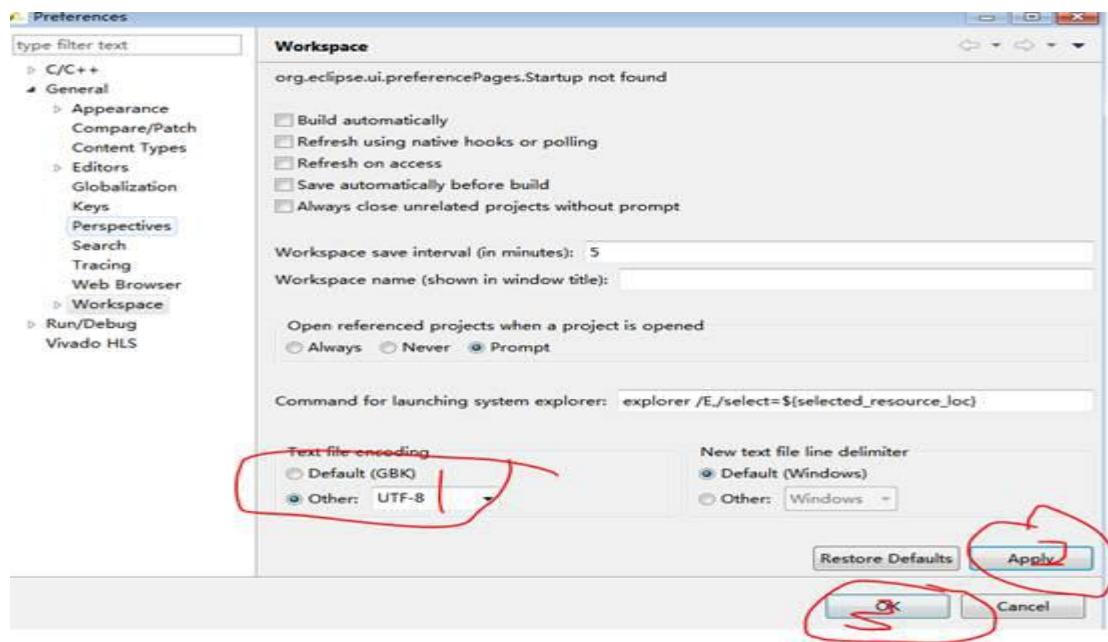
附录(常见问题汇总)

一、工具篇

1.1 HLS 中文注释乱码问题解决方案

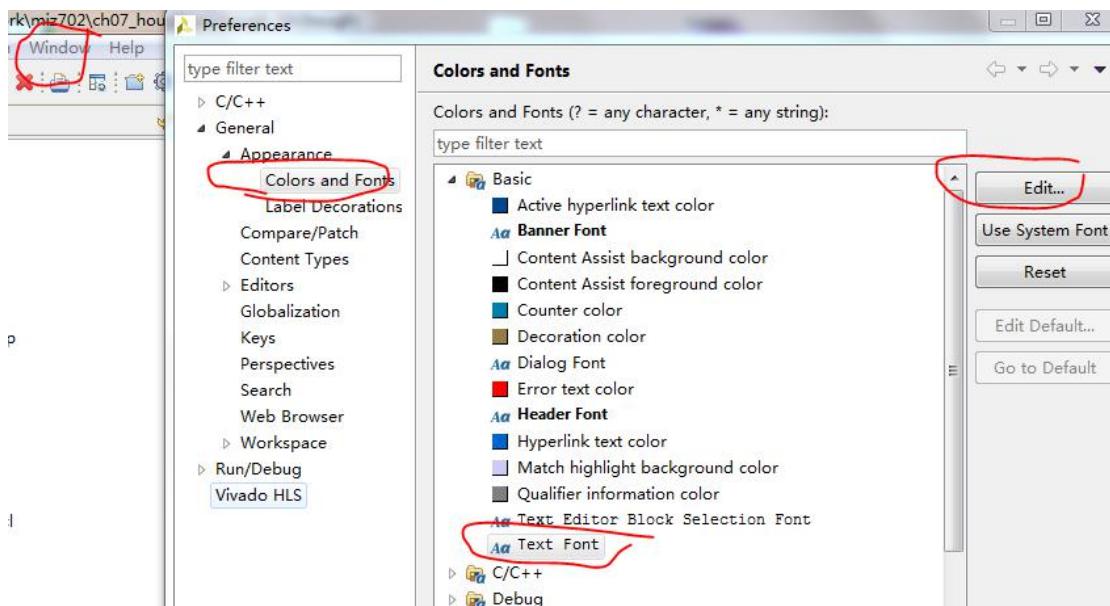
菜单栏选择 Window ->Preferences -> Workspace, 将红色标注的默认格式更改为 UTF-8

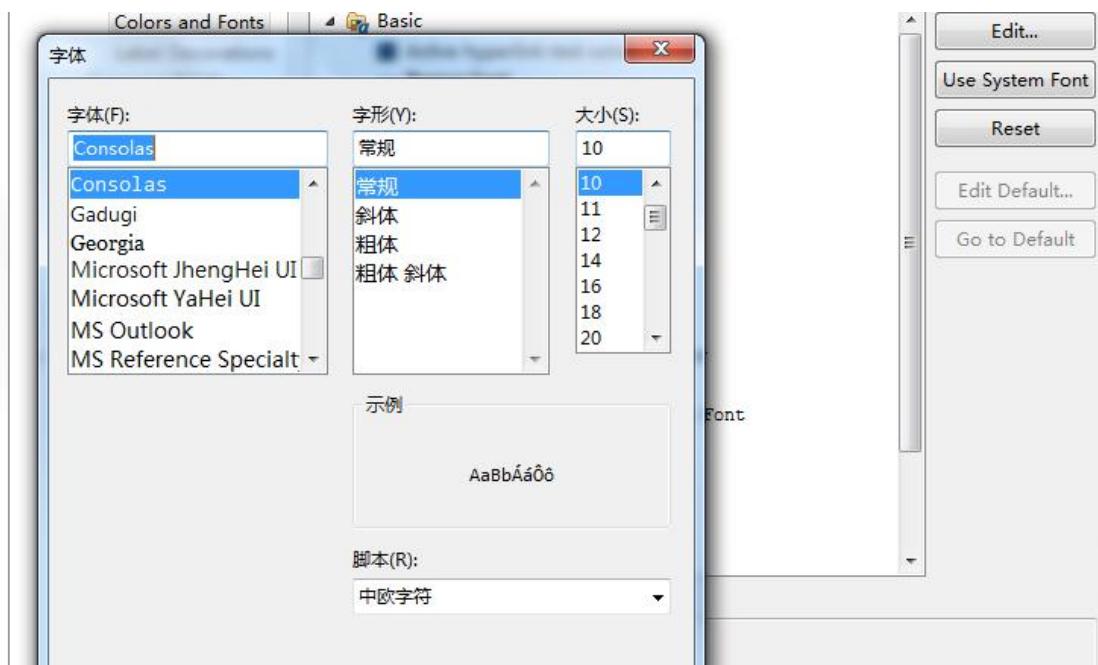




1.2 代码字体大小修改

在 HLS 代码编辑窗口中文注释字体代码偏小，解决办法如下，在字体里的脚本设置为中欧字符。





二、设计篇

2.1 hls::stream 仿真警告

```

6 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
7 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
8 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
9 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
10 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
11 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
12 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
13 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
14 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
15 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
16 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
17 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
18 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
19 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
20 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
21 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han
22 WARNING: Hls::stream 'hls::stream<unsigned char>.8' is read while empty, which may result in RTL simulation han

```

检查设计文件，查看是否是定义了 stream 类型的结构但是没有使用或者综合优化掉了。

2.2 仿真时使用 cvShowImage() 函数但是没有任何错误提示仿真界面直接关闭

- 1、检查测试代码中是否添加 waitKey(0) 或类似语句，笔者调试的时候就因为不小心把这句语句屏蔽掉了，仿真界面直接关闭，导致查代码查了好久。
- 2、检查代码的正确性，因为 HLS 存在一些 Bug，可以尝试新建一个 Solution 再次仿真

一个完美的结束
意味着一个新的开始！

www.osrc.cn

米联客
技术论坛
秀出你的风采！