Z7-Lite 系列教程
之 SDK 篇

Xilinx ZYNQ-7000 SoC

Rev. 1.0

版本记录

版本	时间	描述
Rev. 1.0	2020.6.1	第一版发布

版权声明：

XME/IME 系列核心模块、评估板及其相关知识产权由微相科技有限公司所有。

本文档由微相科技有限公司版权所有，并保留一切权利。在未经微相公司书面许可的情况下，不得以任何方式或形式来修改、分发或复制本文档的任何部分。

免责声明：

产品中所提供的程序源代码、软件、资料文档等，微相科技不提供任何类型的担保；不论是明确的，还是隐含的，包括但不限于合适特定用途的保证，全部的风险，由使用者来承担。

目录

版本记录.....2
版权声明：.....3
免责声明：.....3
目录.....4
开发环境及硬件等相关注意事项.....11
第 1 章 Hello World 实验.....12
1.1 开发流程及系统介绍.....13
1.2 硬件设计.....14
1.3 软件设计.....34
1.4 下载验证.....39
1.5 章节小结.....44
第 2 章 按键控制 LED 实验.....45
2.1 基础知识介绍.....46
2.2 硬件设计.....48
2.3 软件设计.....51
2.4 下载验证.....55
2.5 章节小结.....57
第 3 章 外部中断实验.....58
3.1 实验简介.....59
3.2 硬件设计.....59
3.3 软件设计.....65
3.4 下载验证.....72
3.5 章节小结.....73
第 4 章 EMIO 使用实验.....74
4.1 实验简介.....75

4.2 硬件设计.....75
4.3 软件设计.....85
4.4 下载验证.....91
4.5 章节小结.....92
第 5 章 AXI GPIO 中断实验.....93
5.1 实验简介.....94
5.2 硬件设计.....95
5.3 软件设计.....113
5.4 下载验证.....119
5.5 章节小结.....120
第 6 章 定时器中断实验.....121
6.1 实验简介.....122
6.2 硬件设计.....123
6.3 软件设计.....123
6.4 下载验证.....127
6.5 章节小结.....127
第 7 章 自定义 IP 核实验.....128
7.1 实验简介.....129
7.2 硬件设计.....129
7.3 软件设计.....161
7.4 下载验证.....163
7.5 章节小结.....164
第 8 章 UART 读写实验.....165
8.1 实验简介.....166
8.2 硬件设计.....170
8.3 软件设计.....170

8.4 下载验证.....177
8.5 章节小结.....179
第 9 章 XADC 实验.....180
9.1 实验简介.....181
9.2 Vivado 读取 XADC 数据.....182
9.3 软件读取 XADC 数据—软件设计.....185
9.4 下载验证.....186
9.5 章节小结.....187
第 10 章 PS 端以太网应用实验.....188
10.1 实验简介.....189
10.2 硬件设计.....191
10.3 软件设计.....199
10.4 下载验证.....201
10.5 章节小结.....221
第 11 章 双核 AMP 实验.....222
11.1 实验简介.....223
11.2 硬件设计.....224
11.3 软件设计.....225
11.4 下载验证.....232
11.5 本章小结.....238
第 12 章 FLASH 数据读写实验.....239
12.1 实验简介.....240
12.2 硬件设计.....240
12.3 软件设计.....241
12.4 下载验证.....247
12.5 章节小结.....248

第 13 章 PL 读写 PS 端 DDR 数据.....249
13.1 实验简介.....250
13.2 硬件设计.....254
13.3 软件设计.....269
13.4 下载验证.....271
13.5 章节小结.....273
第 14 章 Micro SD 卡读写 TXT 文本实验.....274
14.1 实验简介.....275
14.2 硬件设计.....278
14.3 软件设计.....282
14.4 下载验证.....288
14.5 章节小结.....290
第 15 章 BRAM 应用实验.....290
15.1 实验简介.....291
15.2 硬件设计.....292
15.3 软件设计.....298
15.4 下载验证.....298
15.5 章节小结.....301
第 16 章 DMA 环路实验.....302
16.1 实验简介.....303
16.2 硬件设计.....303
16.3 软件设计.....314
16.4 下载验证.....317
16.5 章节小结.....324
第 17 章 HDMI 显示实验.....324
17.1 实验简介.....325

17.2 硬件设计.....329
17.3 软件设计.....372
17.4 下载验证.....378
17.5 章节小结.....384
第 18 章 读取 SD 卡内 BMP 图片 HDMI 显示实验.....384
18.1 实验简介.....385
18.2 硬件设计.....388
18.3 软件设计.....390
18.4 下载验证.....395
18.5 章节小结.....397
第 19 章 读取 SD 卡内 BMP 图片 LCD 显示实验.....398
19.1 实验简介.....399
19.2 硬件设计.....399
19.3 软件设计.....417
19.4 下载验证.....419
19.5 章节小结.....422
第 20 章 OV5640 摄像头 HDMI 显示实验.....423
20.1 实验简介.....424
20.2 硬件设计.....425
20.3 软件设计.....447
20.4 下载验证.....449
20.5 章节小结.....450
第 21 章 双目摄像头 HDMI 显示实验.....451
21.1 实验简介.....452
21.2 硬件设计.....452
21.3 软件设计.....465

21.4 下载验证.....468
21.5 章节小结.....469
第 22 章 DAC 波形发生器实验.....470
22.1 实验简介.....471
22.2 硬件设计.....473
22.3 软件设计.....487
22.4 下载验证.....493
22.5 章节小结.....498
第 23 章 ADC 示波器实验.....499
23.1 实验简介.....500
23.2 硬件设计.....501
23.3 软件设计.....514
23.4 下载验证.....519
23.5 章节小结.....522
第 24 章 基于 TCP 协议的远程更新 QSPI Flash 实验.....523
24.1 实验简介.....524
24.2 硬件设计.....524
24.3 软件设计.....526
24.4 下载验证.....531
24.5 章节小结.....541
第 25 章 基于 UDP 协议的远程更新 QSPI Flash 实验.....542
25.1 实验简介.....543
25.2 硬件设计.....543
25.3 软件设计.....543
25.4 下载验证.....546
25.5 章节小结.....553

第 26 章 PL 端网口应用实验.....错误！未定义书签。
26.1 实验简介.....错误！未定义书签。
26.2 硬件设计.....错误！未定义书签。
26.3 软件设计.....错误！未定义书签。
26.4 下载验证.....错误！未定义书签。
26.5 章节小结.....错误！未定义书签。

开发环境及硬件等相关注意事项

开发环境

硬件开发环境：Vivado 2018.3

软件开发环境：SDK 2018.3

硬件平台

开发板型号	芯片型号
Mizar-7010 版	XC7Z010-1CLG400
Mizar-7020 版	XC7Z020-2CLG400
XME0715-PE100	XC7Z015-2CLG485
XME0720-PE200	XC7Z020-2CLG484
Z7-Lite 7010 版	XC7Z010-1CLG400
Z7-Lite 7020 版	XC7Z020-2CLG400

注意事项：

1、 本教程示例工程基于上述开发环境，如用户使用其他版本的开发环境，请查阅相关资料，了解不同版本之间的差异。

2、 本教程为通用教程，适合不同型号开发板用户学习，仅需修改芯片型号和硬件接口约束等少量配置，差异部分在教程相关位置会有说明，请仔细阅读。

第 1 章　　Hello World 实验

“Hello World!”是各类编程教程中最经典的实验，无数软件工程师都是从“Hello World”开始自己的编程之路。我们也沿袭这一传统，从串口打印“Hello World”开始我们 ZYNQ 的学习之路。通过本次实验我们将了解到 ZYNQ 的 PS 部分的开发流程，以及熟悉相关软件的操作。

本章包括以下几个部分：

1.1 开发流程及系统介绍

1.2 硬件设计

1.3 软件设计

1.4 下载验证

1.5 章节小结

1.1 开发流程及系统介绍

首先我们来了解一下一个完整的 ZYNQ 嵌入式系统的组成部分：1、系统内核：ARM Cortex-A9；2、存储器：DDR3 Memory Controller+存储芯片；3、其他外设；4、外部电路组件；5、应用程序。

由此可见我们开发一个 ZYNQ 嵌入式系统需要完成硬件设计和软件设计两大部分的内容。针对本次串口输出实验，我们仅需要 UART 串口这一个外设即可实现需要的功能。

结合我们要完成的 ZYNQ 嵌入式系统开发，根据上面的介绍可以发现一个嵌入式系统的开发流程为：第一步：创建 Vivado 工程；第二部：使用 IP 核创建处理器系统；第三步：生成顶层文件并导出硬件描述文件到 SDK;第四步：在 SDK 中创建应用程序；第五步：在硬件电路板上验证功能。

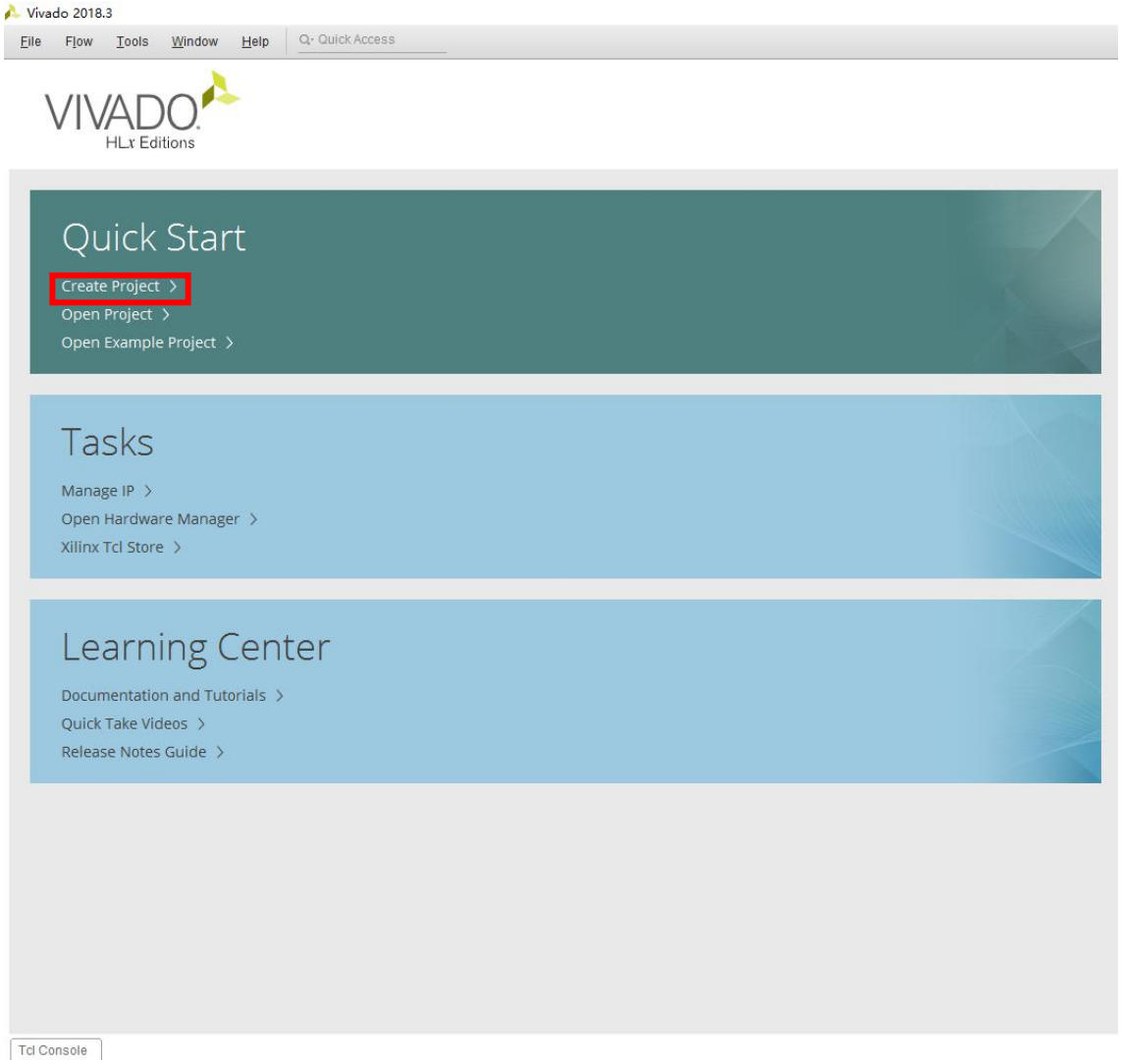
在开发的过程中还要不断的进行 debug，排除硬件设计和软件设计上的错误，使整个系统运行的更加的高效稳定。

下面，我们将按照上面介绍的内容，一步步搭建实现项目目标的嵌入式系统。

1.2 硬件设计

第一步：创建 Vivado 工程

双击打开 Vivado，在 Vivado 主界面点击“Quick Start”栏的“Creat Project”，



之后弹出工程向导界面，点击“Next”。如下图所示：

图 1.2.1 点击创建工程

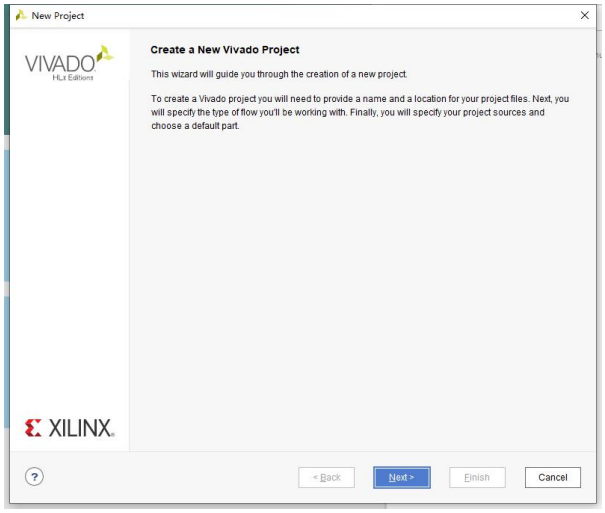


图 1.2.2 创建工程向导

进入工程命名界面。在“Project name”栏输入工程名“hello_world”，工程路径可使用任意路径，本章我们将该工程放到 D：/ZYNQ_Mizar/SDK_Demo 文件夹下。读者可自行创建一个单独文件夹保存实验工程。**此处注意！工程名和保存路径只能由字母、数字、下划线组成，不能包含中文、空格以及其他特殊字符！**

勾选“Creat project subdirectiry”,在文件夹下创建工程子目录，点击“Next”，如下图所示：

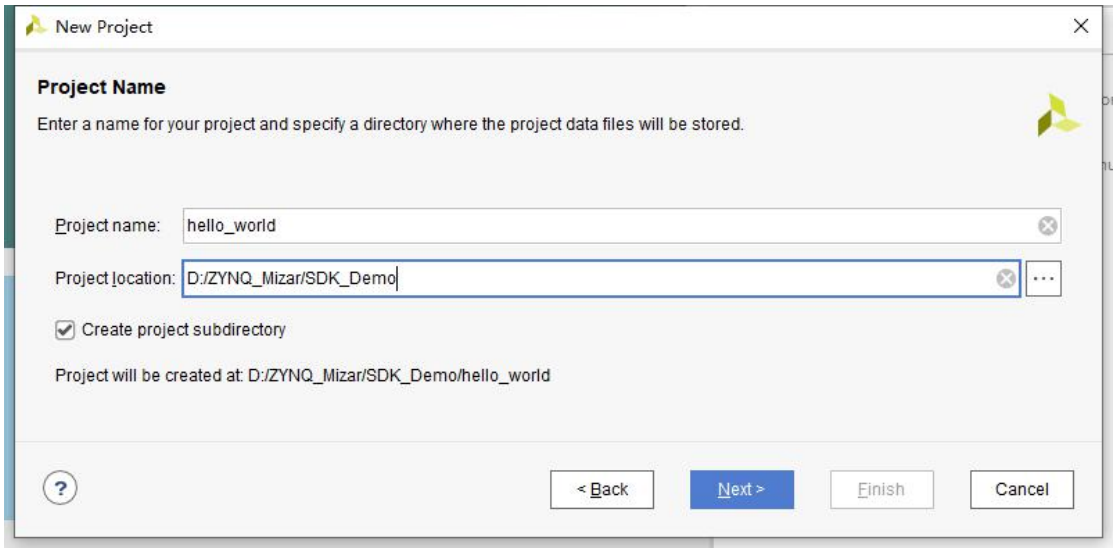


图 1.2.3 设置工程名及保存路径

进入图 1.2.4 所示界面，在此界面设置工程类型。我们选择"RTL Project"。对于不需要添加源文件和约束文件的项目，可以勾选“Don not specify source at this time”。跳过添加源文件和约束文件的步骤，点击“Next”跳到器件选型的界面。

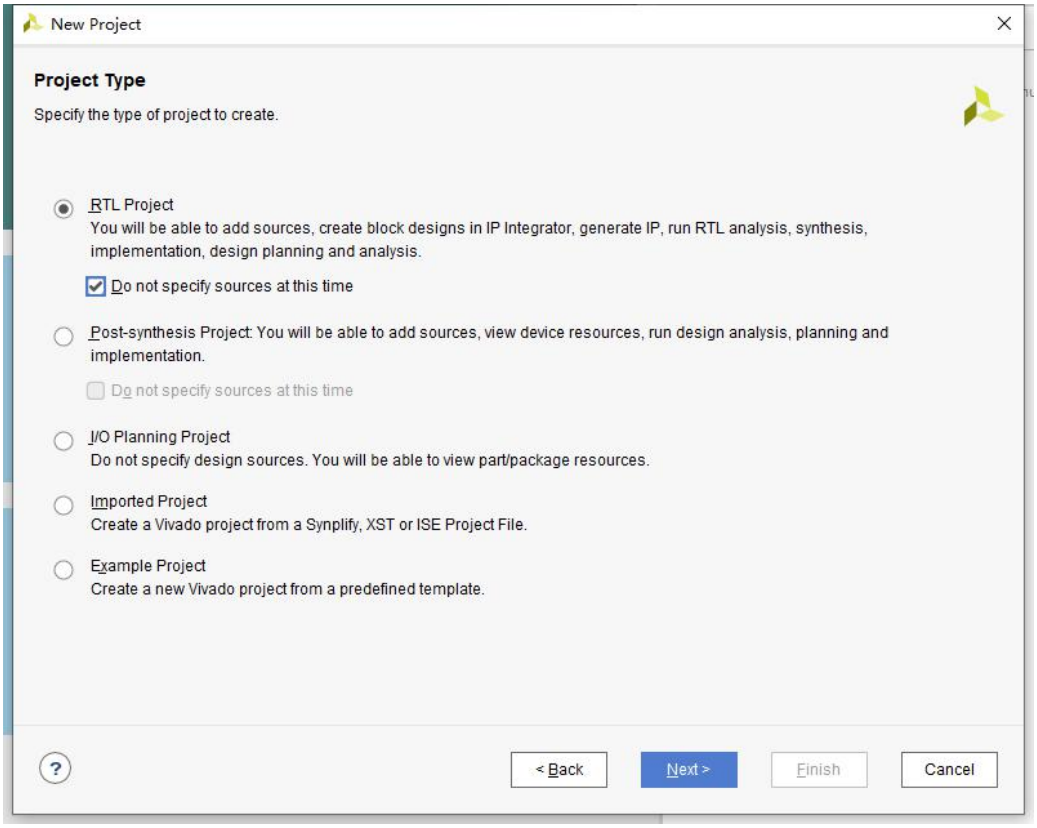


图 1.2.4 设置工程类型

在器件选型界面选择 ZYNQ 芯片的型号，该型号一定要和开发板上芯片的型号一致。Mizar 开发板上 ZYNQ 芯片的型号有两种，分别为 XC7Z010 和 XC7Z020。可以通过查看芯片丝印或者查阅购买信息确定芯片型号。本例程以 XC7Z020 为例。（注意：对于 XME0715-PE100 开发板，选择 XC7Z015-2CLG485；对于 XME0720-PE200 开发板，选择 XC7Z020-2CLG484）

芯片有许多的型号，为了快速查找需要的型号，我们可以通过过滤来缩小查找的范围，在选型界面 Family 栏里选择“Zynq_7000”系列，Speed 栏选择“-2”，

在 Package 栏选择“clg400”。然后根据 ZYNQ 芯片的型号，在下面的器件列表里面选择“xc7z020clg400-2”,如下图所示。若是 XC7Z010 芯片，则在 Speed 栏选择“-1”，在器件列表里面选择“xc7z010clg400-1”。选中之后，点击“Next”。

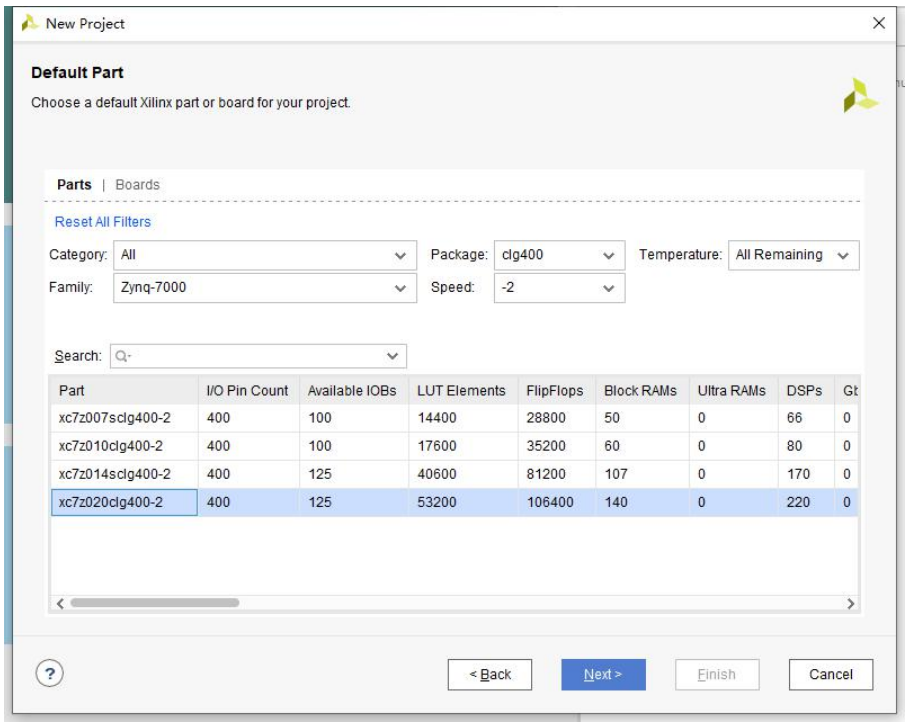


图 1.2.5 选择芯片型号

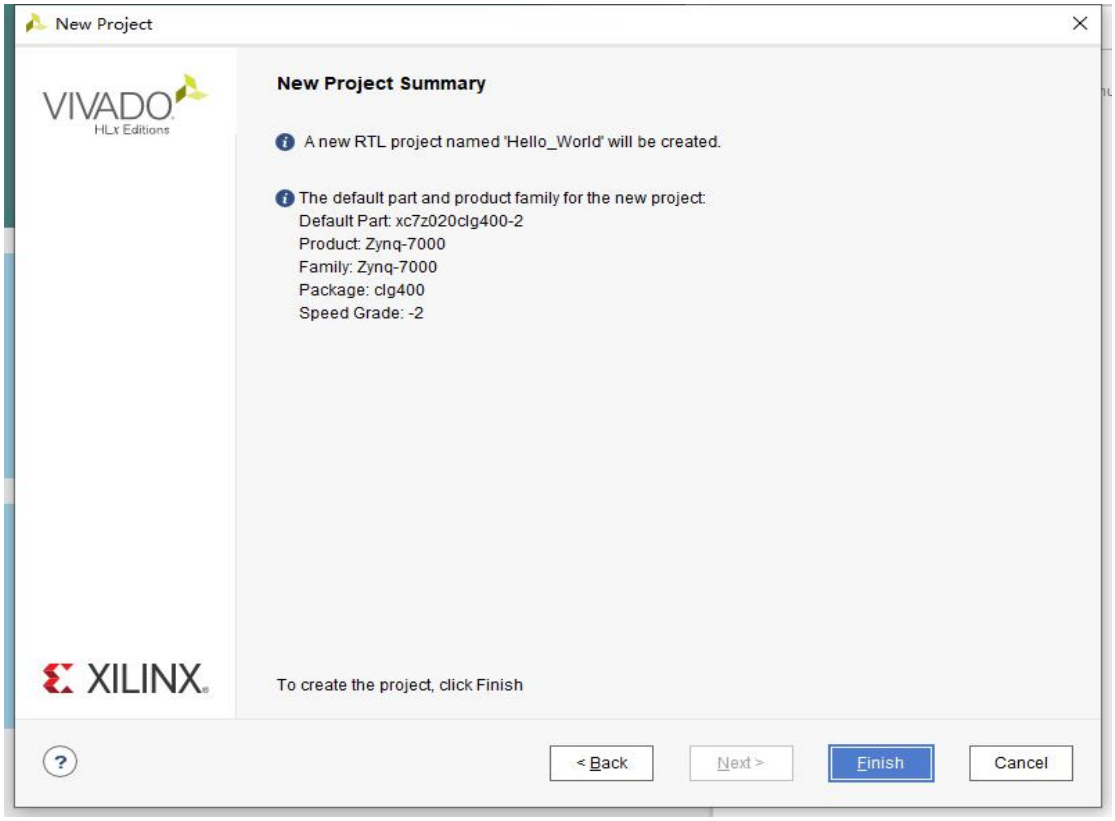
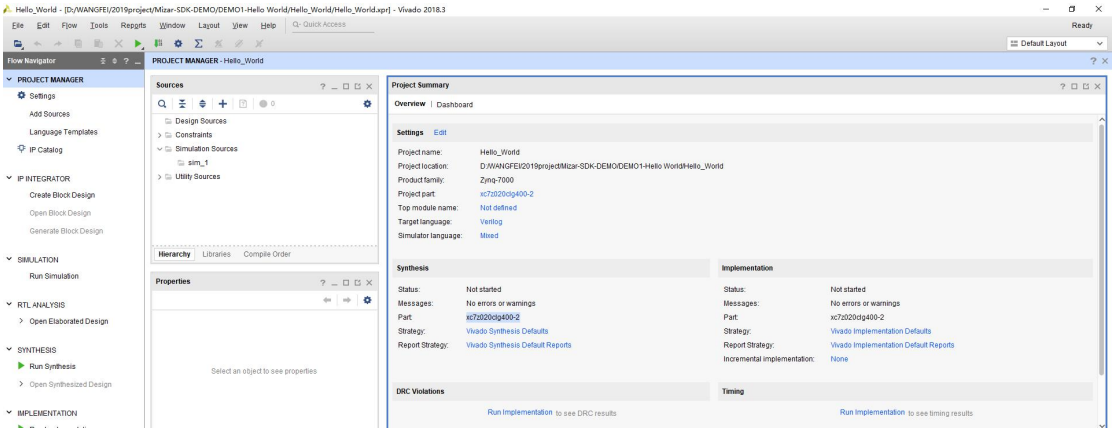


图 1.2.6 工程信息界面

之后进入工程信息界面，显示该工程的一些信息，如图 1.2.6 所示。在此界面可以查看之前设置的工程名、器件型号等信息，检查是否有设置错误的地方。如果设置有误，可以点击“Back”返回之前的设置界面重新设置。检查无误后点



击“Finish”，完成工程的创建。

图 1.2.7 创建完成后的界面

创建完成后的界面如图 1.2.7 所示。

工程创建完成后我们就开始系统硬件部分的设计，Vivado 提供了一个面向对象的设计工具——IP 集成器（IP intergator），在 IP 集成器中可以非常简单的添加各种官方提供或自己编写的各种功能模块（IP）,并配置相关参数。而且它支持大部分官方 IP 接口的智能自动连接、一键式 IP 子系统生成和实时 DRC 等功能，极大的提高了开发效率，降低出错的风险，即使面对复杂庞大的系统也能完成快速搭建。

接下来我们来讲解本系统的搭建流程。

如图 1.2.8 所示，点击左侧导航栏（Flow Navigator）中的 IP Integrator 选项下的 Creat Block Design。然后在弹出的对话框中输入所创建的 Block Design 的名称，在 Design name 栏中输入“hello_world”,点击“OK”。

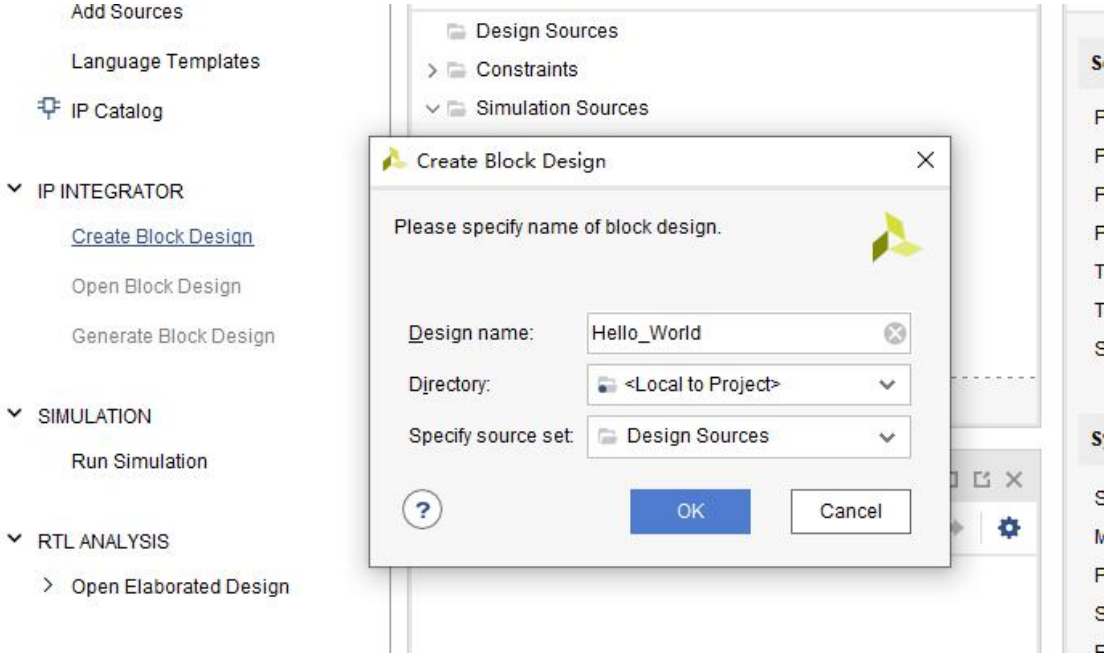


图 1.2.8 创建 BlockDesign

之后弹出 Diagram 窗口，如图 1.2.9 所示，我们将在此窗口下完成完成本实验硬件方面的设计。点击界面中间的“+”或者按快捷键 Ctrl+ I，或者在空白区域点击鼠标右键，选择“ADD IP”。　　之后会弹出可添加的 IP 目录，在搜索栏中输入“ZYNQ”,在搜索结果中找到“ZYNQ Processing System”并双击，将该处理器系统 IP 添加到设计中。

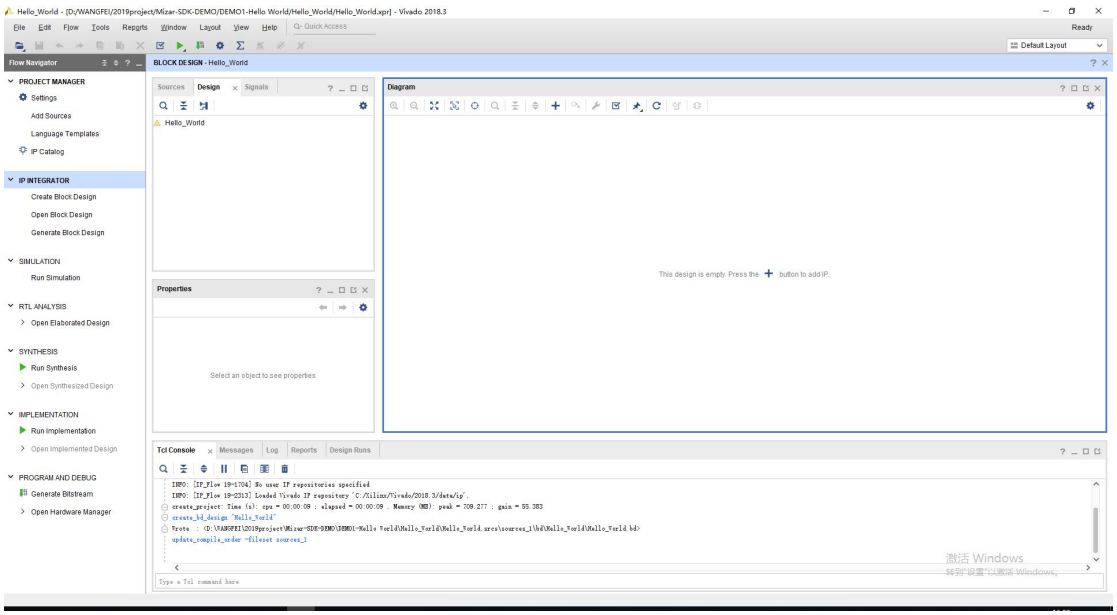


图 1.2.9 Block Design 界面

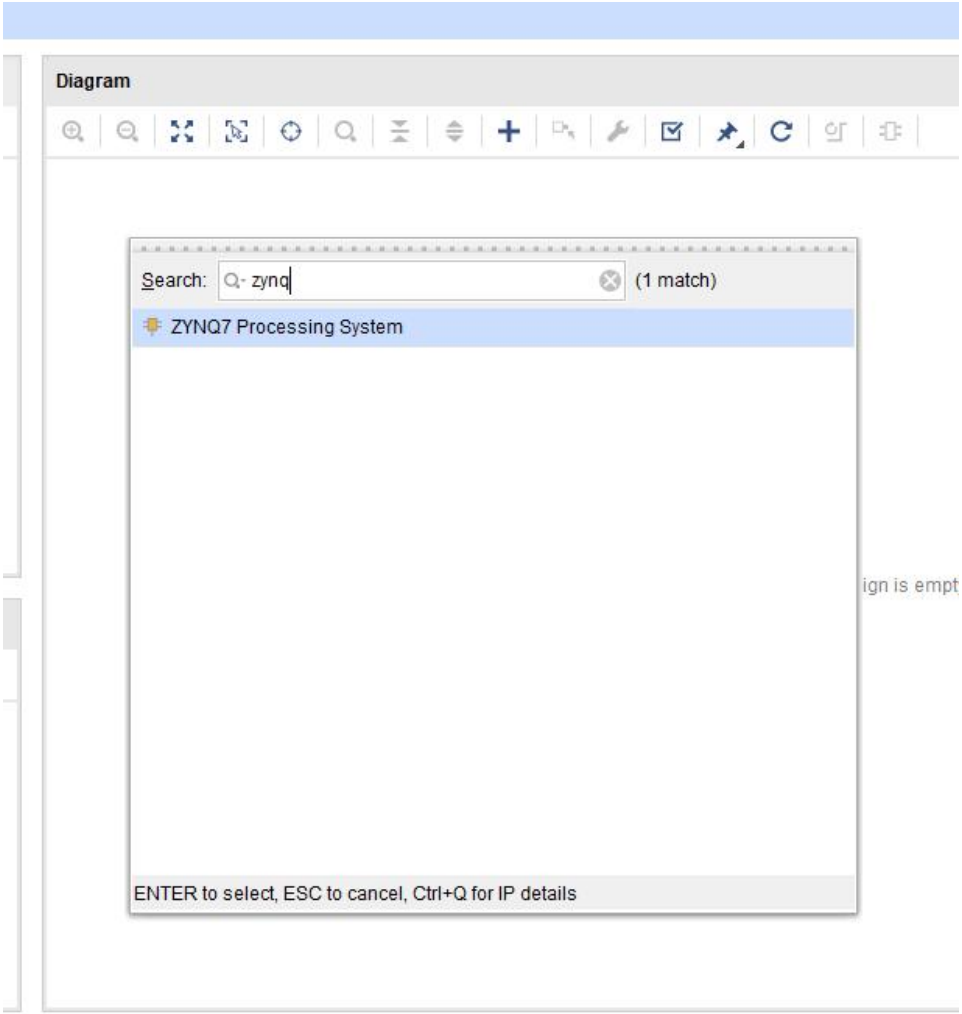
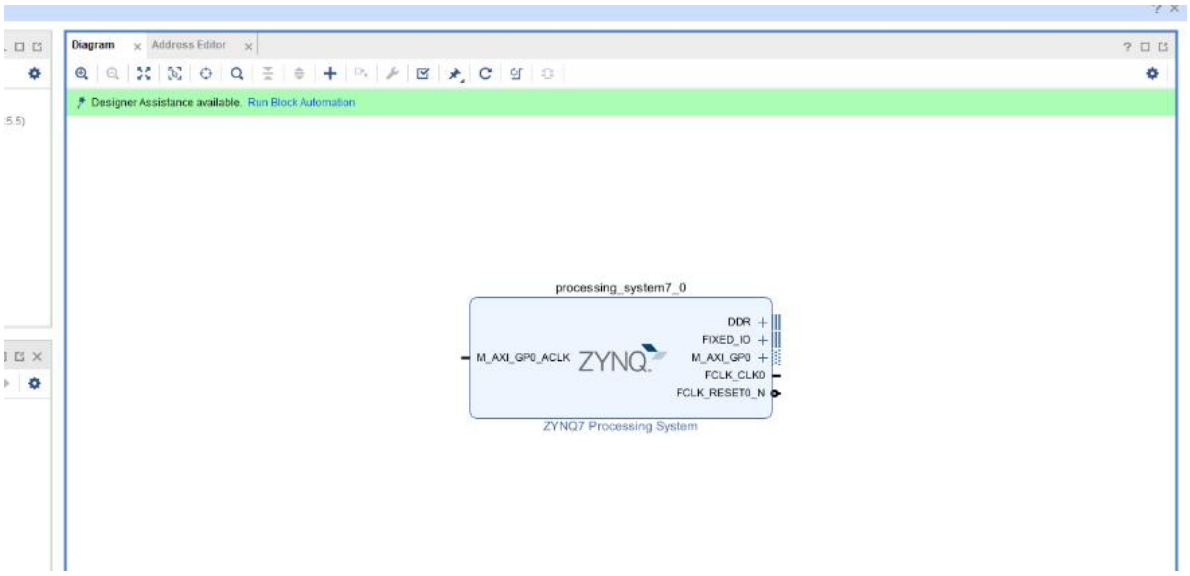


图 1.2.10 添加 ZYNQ 处理系统 IP



添加完成后，模块 IP 出现在 Diagram 中，如下图所示：

图 1.2.11 ZYNQ7 Processing System IP

双击添加的 ZYNQ7 Processing System 模块，进入 ZYNQ 处理系统配置界面。页面左侧显示导航面板，右侧为配置信息面板。如下图所示：

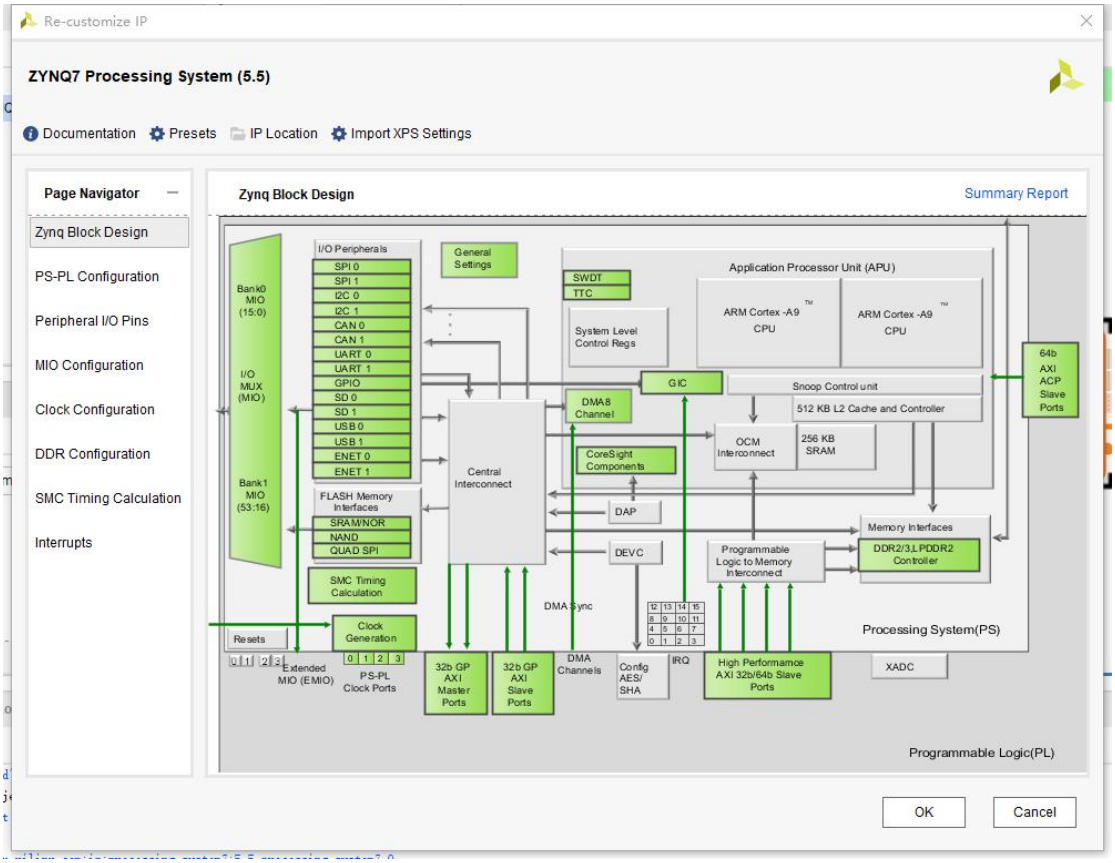


图 1.2.12 ZYNQ7 Processing System 配置界面

在该页面点击“Peripheral I/O Pins”，右侧显示出引脚配置界面，在这里我们可以看到哪些引脚可以被配置成那些相应的功能。本次实验我们需要用到串口的功能，但是图上有那么多引脚可以被配置为串口，我们应该怎么做呢？这里我们要通过查看原理图来确定哪些引脚应该被配置为串口的功能。如图 1.2.13 所示，在开发板上 MIO14,MIO15 连接到串口芯片上实现串口的功能，所以我们要在 Vivado 软件上将 MIO14,MIO15 配置为串口。在配置界面点击 MIO14,MIO15 下面的 UART0,方块的颜色会变成绿色，同时两个 MIO 也变成绿色，表示这两个 I/O 已经和芯片内部的串口电路相连接，可以实现串口的功能，如图 1.2.14 所示。对于其他型号开发板，串口使用的 MIO 引脚可能不同，请自行查阅原理图进行设

置，在后续的实验中不再进行特别提醒。

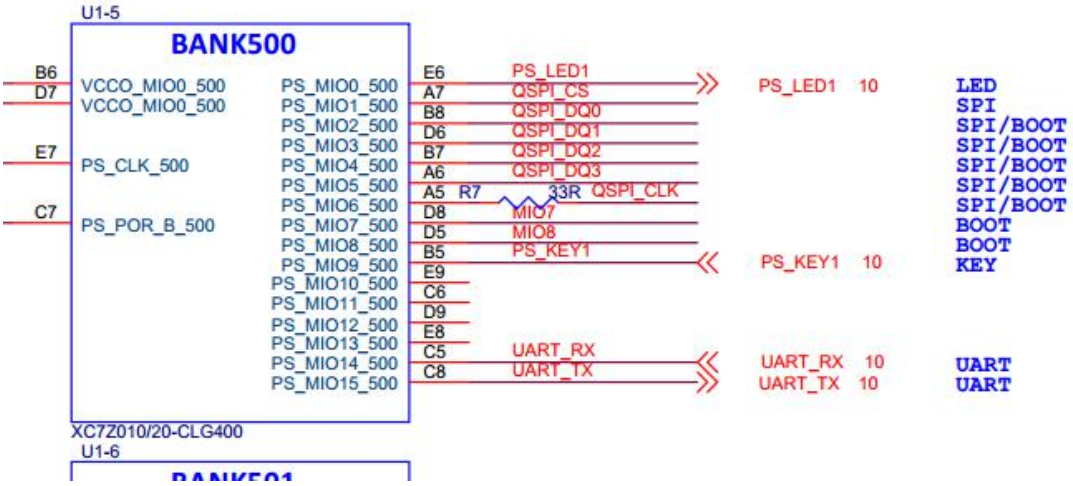


图 1.2.13 串口相关原理图

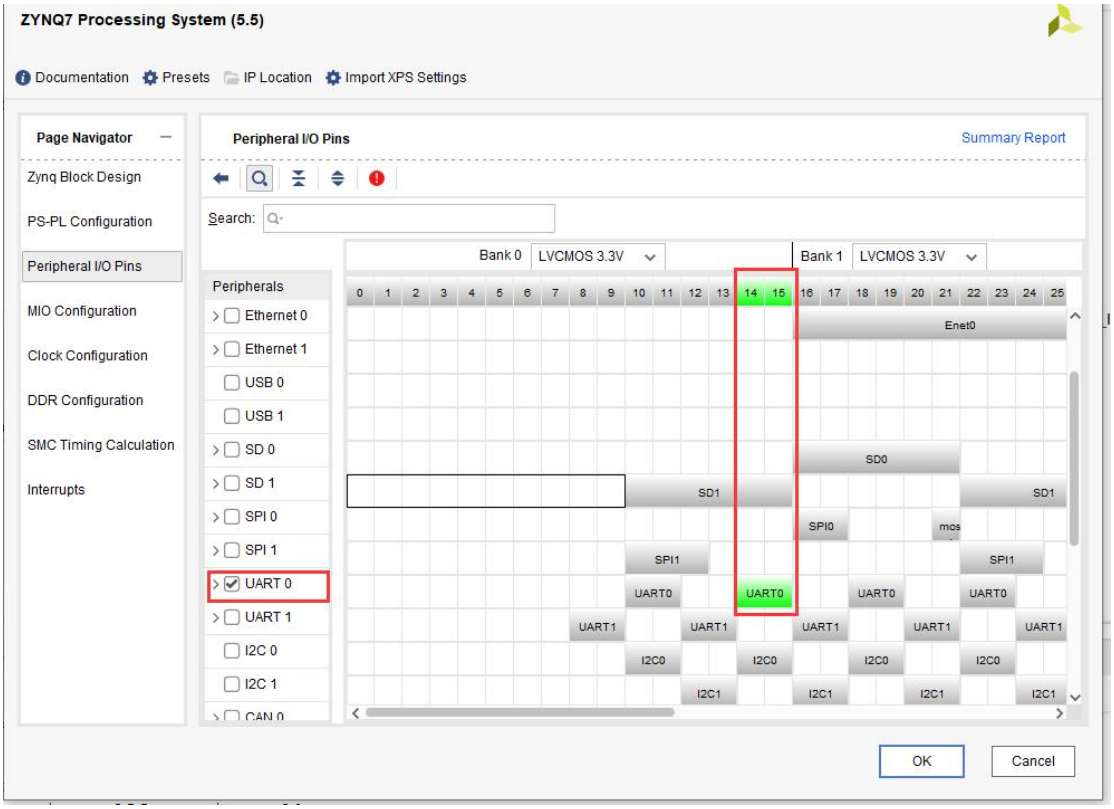
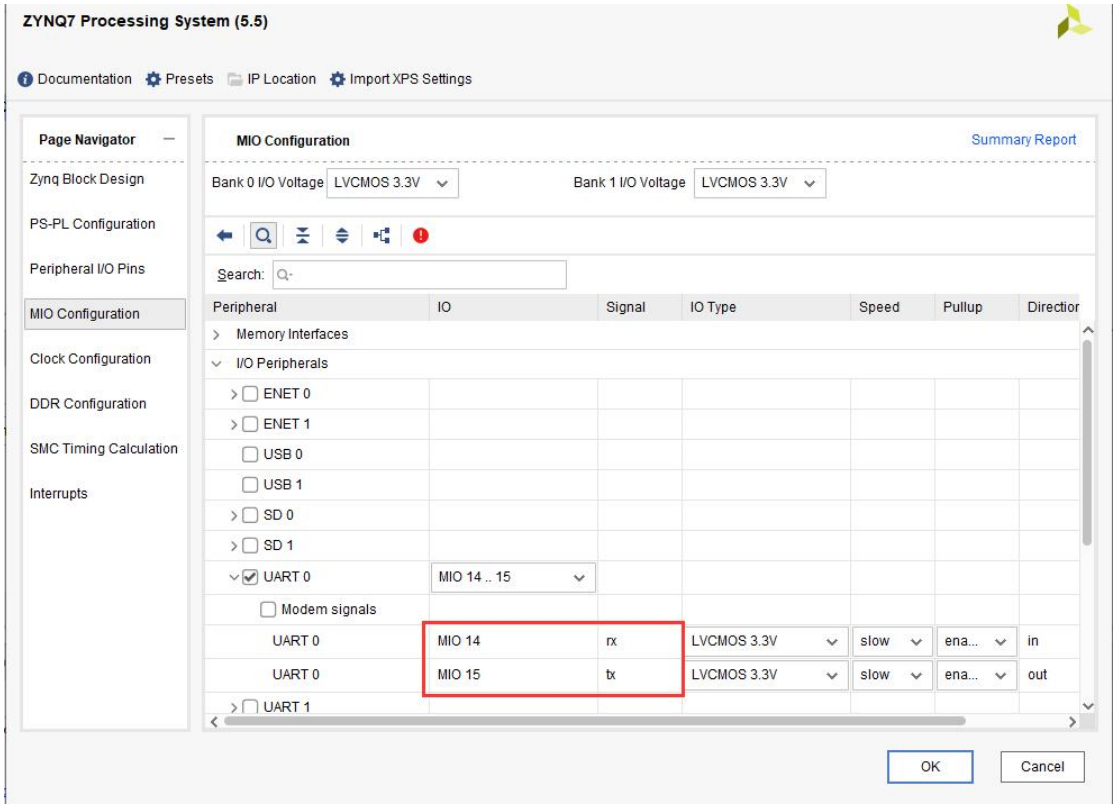


图 1.2.14 配置 MIO14 MIO15 为 UART0

对于 XME0715-PE100 开发板，串口为 UART0,对应 MIO14,MIO15。涉及硬件设计的差异之处以开发板原理图为准，设置前请仔细查阅开发板原理图。

点击左侧的“MIO Configuration”，在右侧展开 I/O Peripherals->UART1，因为我们使用的引脚为 MIO14 和 MIO15，属于 Bank0。在此处可以看到具体的引脚配置信息。其中 MIO15 为 TX 引脚方向为输出，MIO14 为 RX 引脚方向为输入，



引脚电压为 3.3V，速度为 slow，使能上拉，如下图所示：

图 1.2.15 UART1 串口配置信息

除此之外，我们还要配置串口波特率等相关配置，点击左侧“PS-PL Configuration”，我们可以在这里设置波特率等设置，点开“General”可以看到默认的串口波特率为 115200，可以通过下拉按钮选择其他波特率，这里我们保持默认设置。

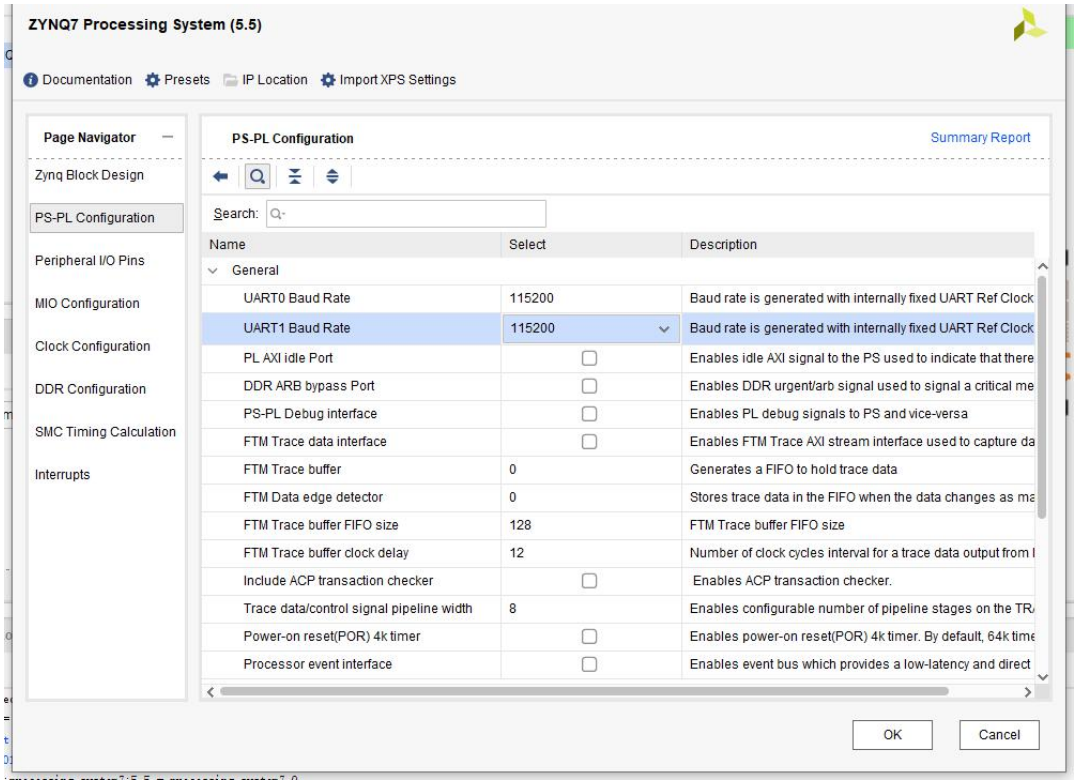
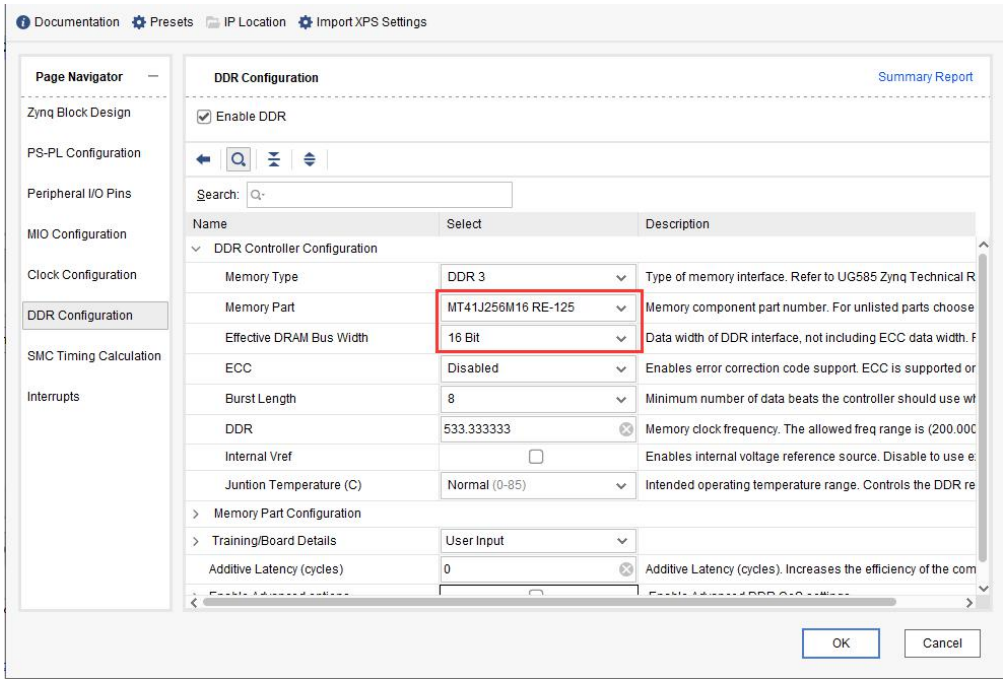


图 1.2.16 UART0 波特率设置

点击左侧“DDR Cfinguration”，选择 DDR 器件的型号，点开 DDR Controller Configuration 下拉按钮，在 Memory Part 一栏选择“MT41J256M16 RE-125”，总



线宽度选择 16bit。

图 1.2.17 配置 DDR3 控制器

点击左侧“Clock Configuration”，显示 ZYNQ PS 端的时钟频率配置页面，此处的各个时钟频率保持默认设置，因为本实验不需要用到 PL 部分，所以点开“PL Fabric Clock”，取消勾选“FCLK_CLK0”，如图 1.2.18 所示。

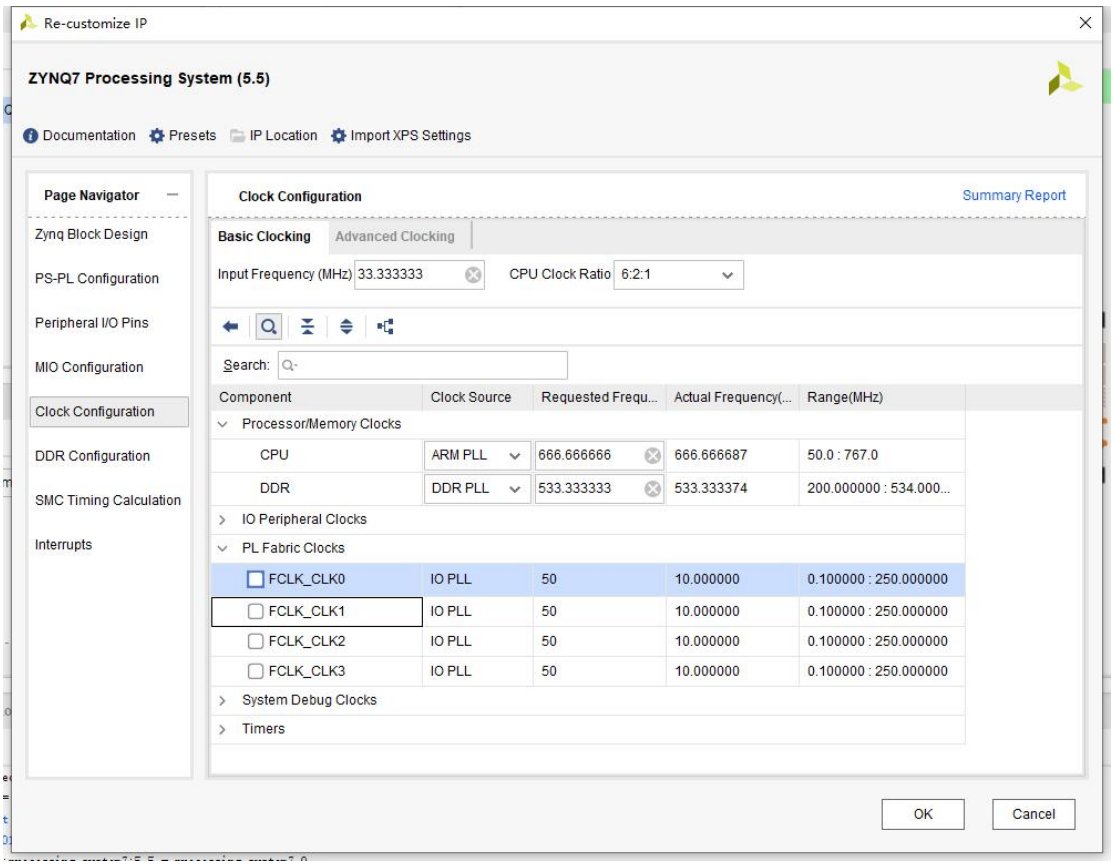


图 1.2.18 配置相关时钟

除此之外，我们还要取消一些与 PL 相关设置的勾选，点击“PS-PL Configuration”，在右侧展开 General 下的 Enable Clock Resets， 取消勾选其中的 FCLK_RESET0_N。

另外在当前界面中展开 AXI Non Secure Enablement 下的 GP Master AXI Interface，取消勾选其中的 MAXI GP0 interface。如下图所示：

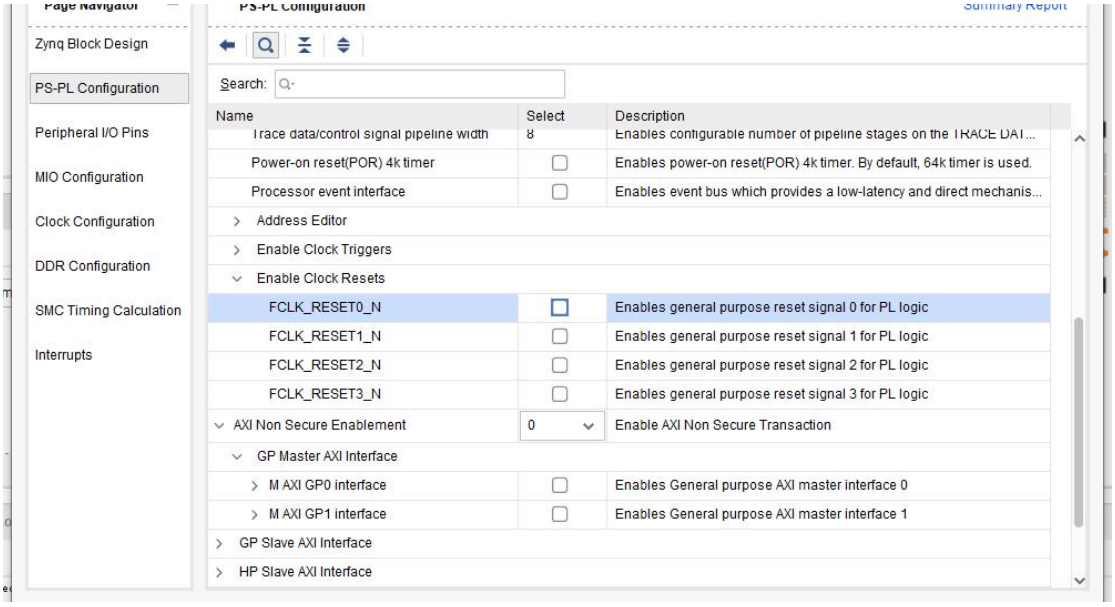


图 1.2.19 取消勾选 PL 相关设置

这样我们的配置就完成了，点击“OK”，保存配置并返回。

回到主界面我们会看到 ZYNQ7 模块发生了变化，如图 1.2.20，对比之前发现少了部分接口，这是因为我们取消了关于 PL 的部分设置。

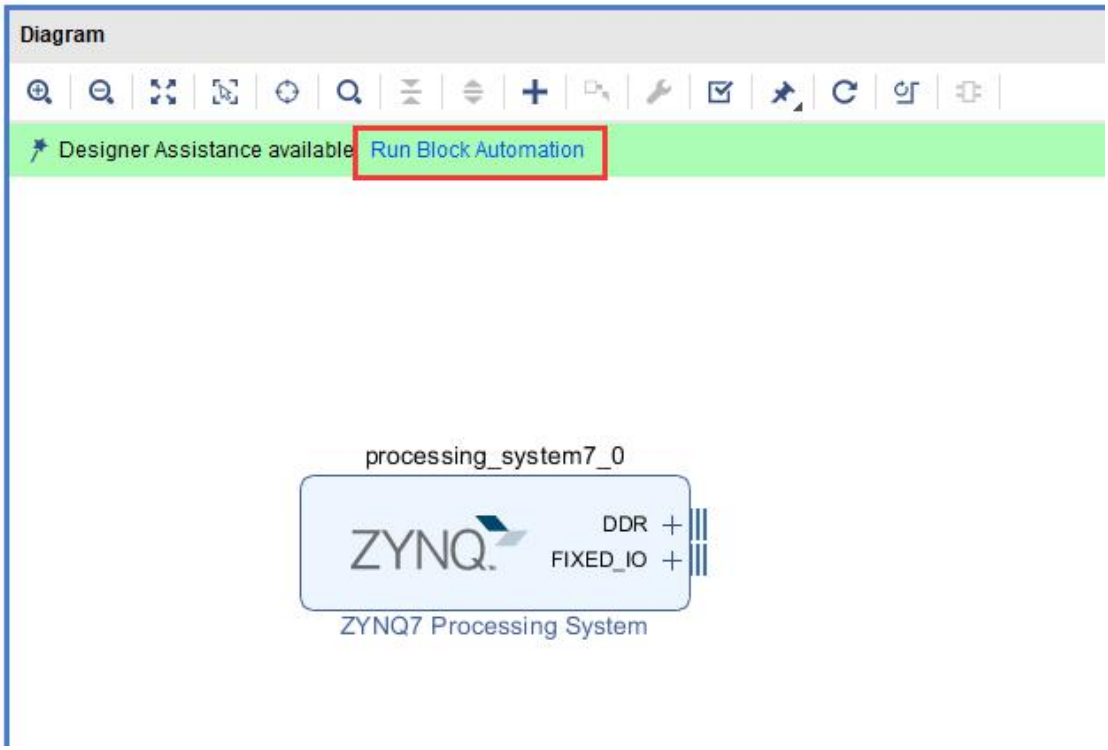
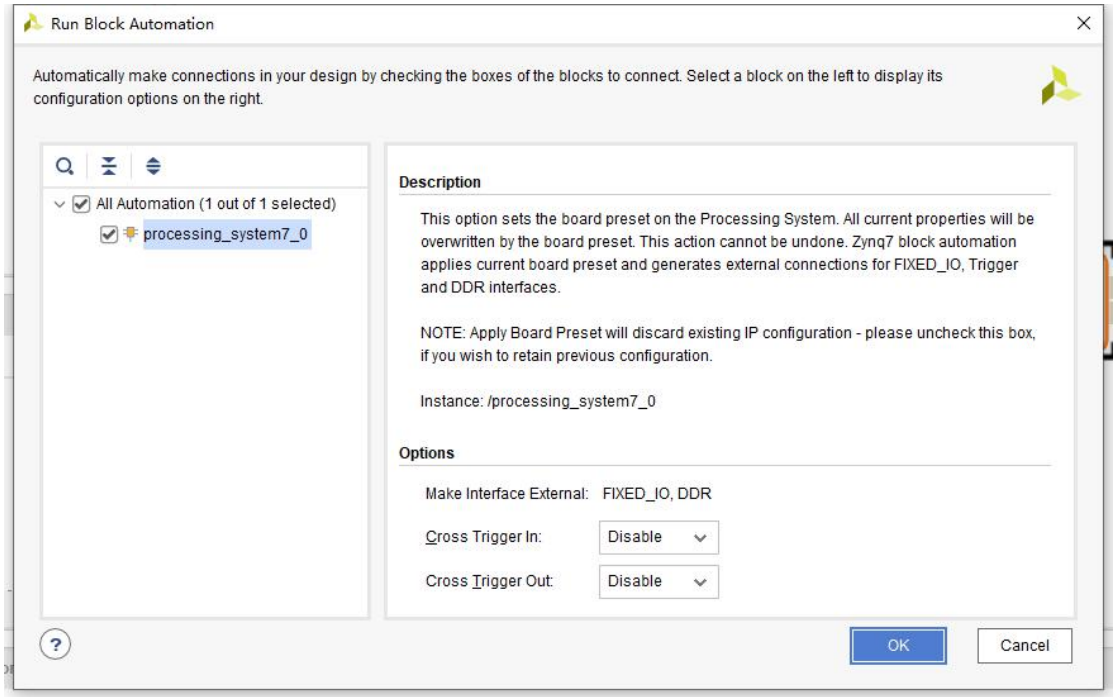


图 1.2.20 配置完成



之后点击上图中的“Run Block Automation”，会弹出如下对话框：

图 1.2.21 自动配置端口及 IP 间的互联

全选，点击“OK”，软件会自动为 IP 添加输入输出端口以及各个 IP 之间的连接，因为本实验只用到一个 IP，所以系统只引出了两组信号。

配置完成的 IP 如下图所示，点击接口旁边的“+”可以看到展开两组接口，查看其中具体的信号有哪些。

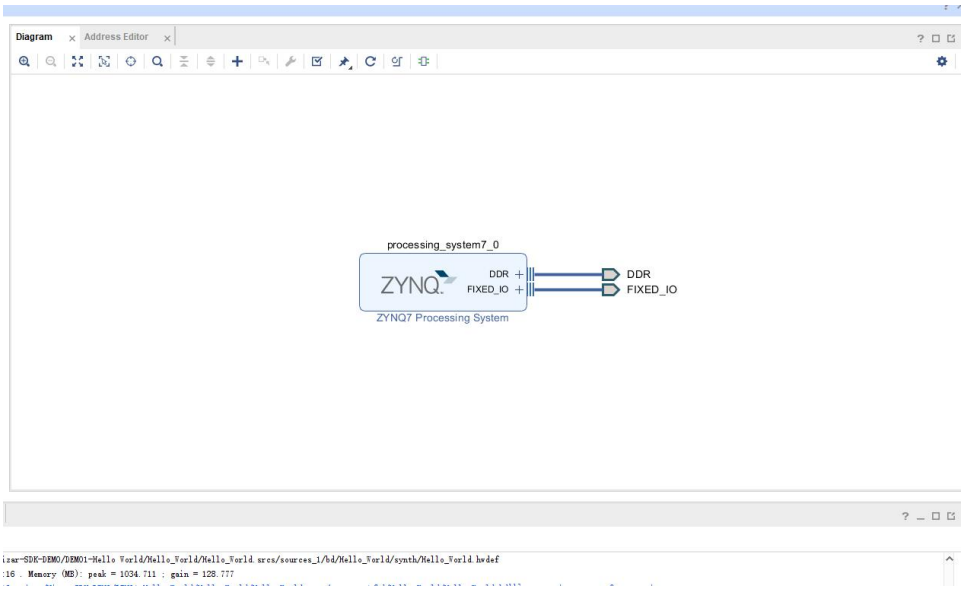
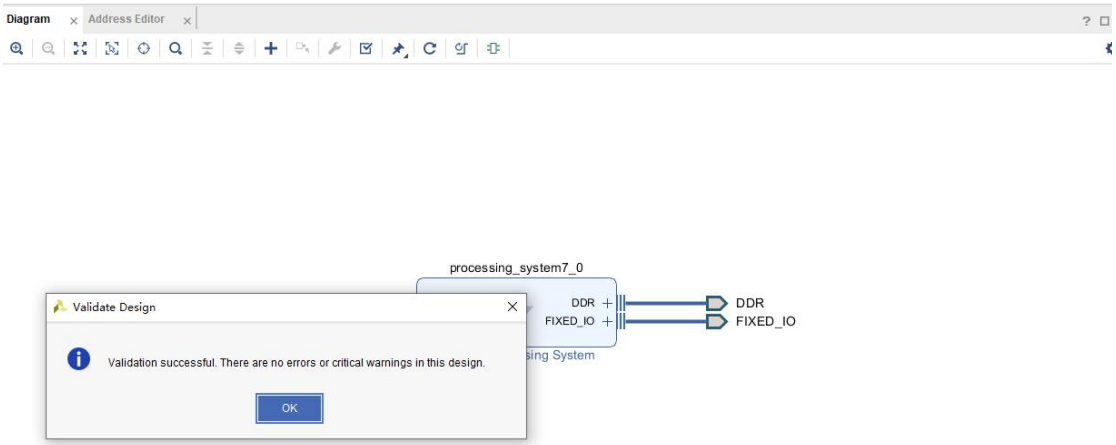


图 1.2.22 配置完成的 IP

当前实验不需要添加其他 IP，快捷键 Ctrl + S 保存当前设计，点击工具栏的 Validate Design 按钮验证当前的设计。验证完成后弹出对话框提示没有错误或者



关键警告，点击“OK”，如下图所示：

图 1.2.23 验证设计

如果出现错误或严重警告说明之前的配置有问题，需要返回重新检查配置是否有遗漏或错误。

设计完成后需要生成顶层 HDL 模块，在 Sources 窗口，选中 Design Source 下的 hello_world.bd，右键点击 hello_world.bd，在弹出的菜单中选择“Generate Output Products”，如下图所示：

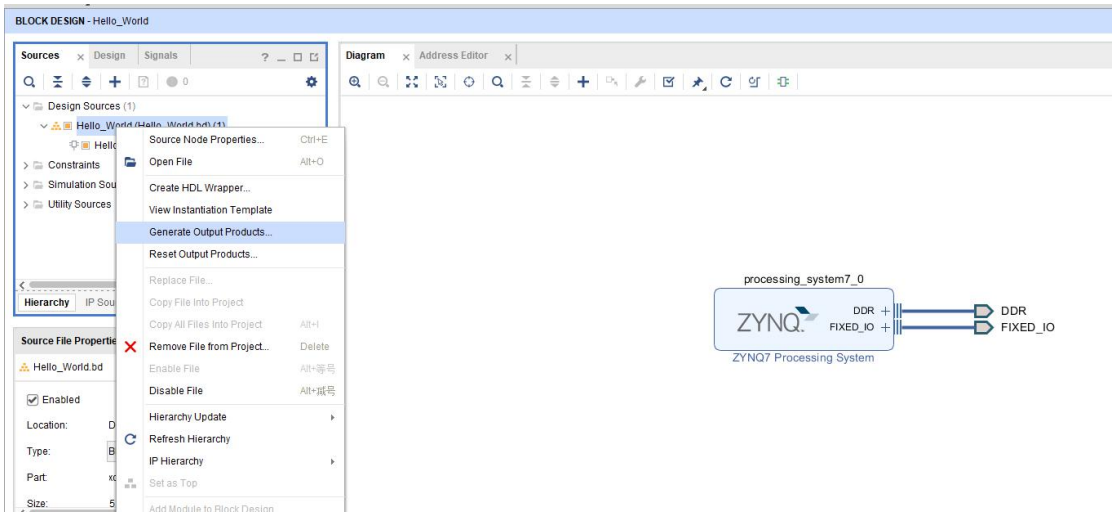
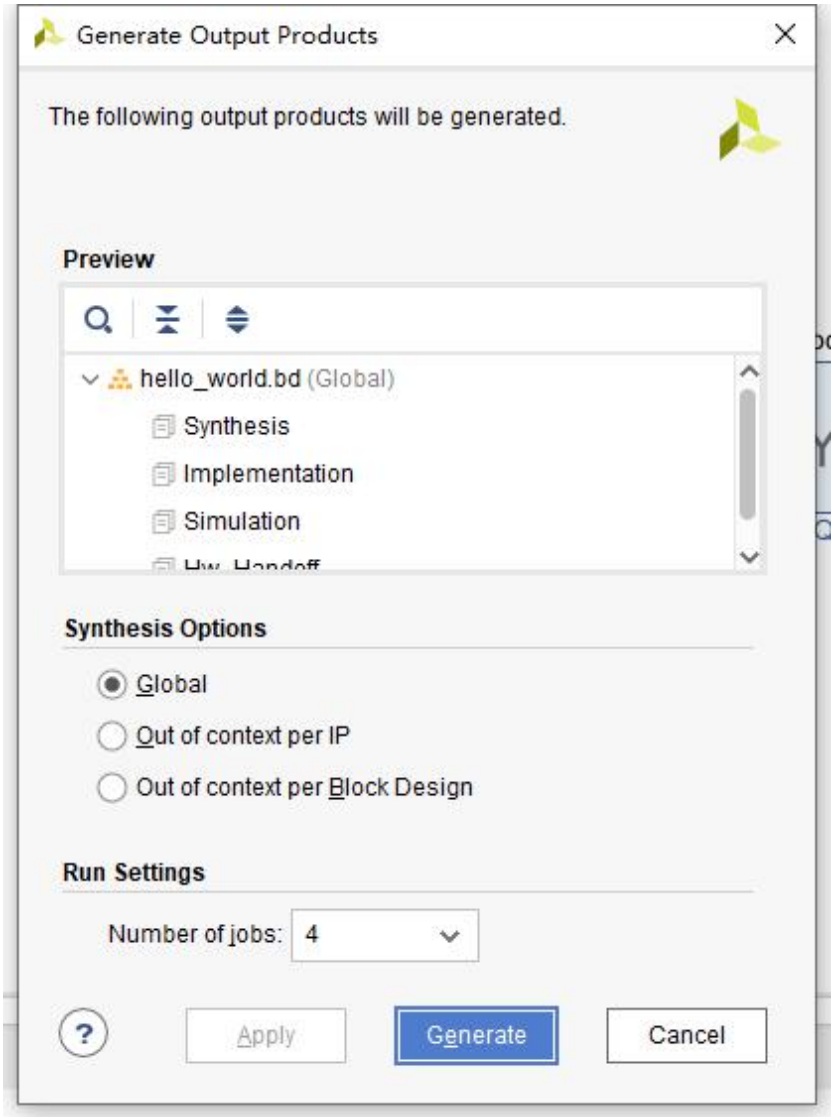


图 1.2.24 选择 Generate Output Products



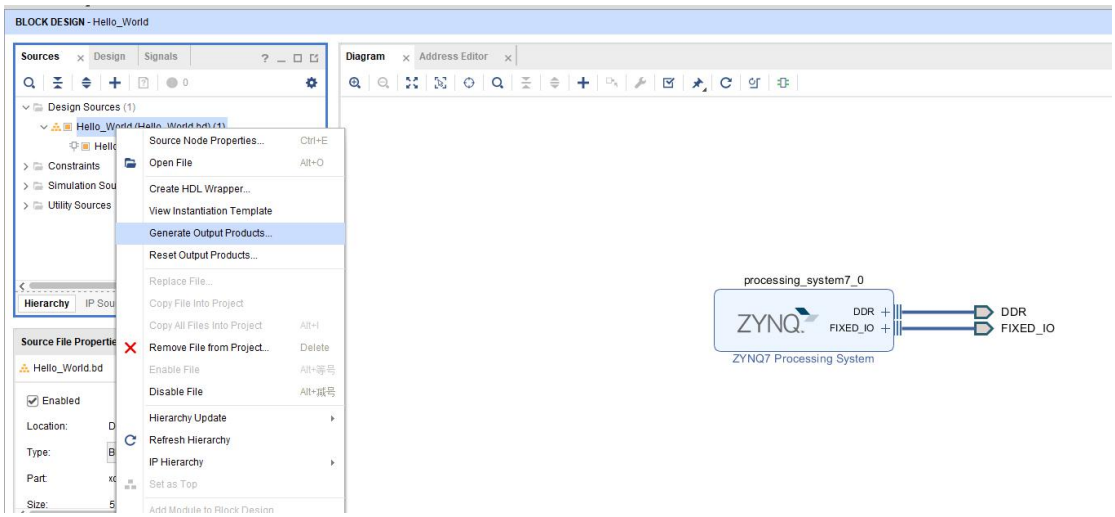
弹出“Generate Output Products”对话框，如下图所示：

图 1.2.25 设置 Generate 选项

在对话框中 Synthesis Options 选择 Global； Run Setings 用于设置生成过程中要使用的处理器的线程数，进行多线程处理，保持默认或设置为个人电脑最大可使用线程数的一半。然后点击“Generate”来生成设计的综合、实现和仿真文件。

Generate 完成后，在弹出的对话框中点击“OK”。

在 Sources 窗口，选中 Design Source 下的 hello_world.bd，右键点击



hello_world.bd，在弹出的菜单中选择“Create HDL Wrapper”，如下图所示：

图 1.2.26 选择 Create HDL Wrapper

在弹出的对话框中确认勾选“Let Vivado manage wrapper and auto-update”，然后点击“OK”。

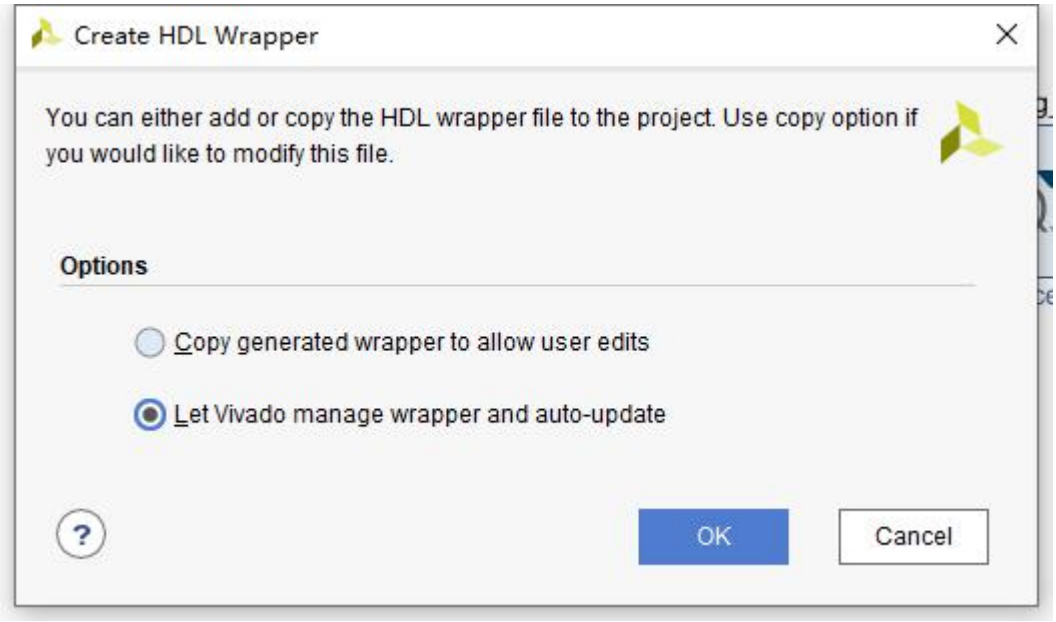
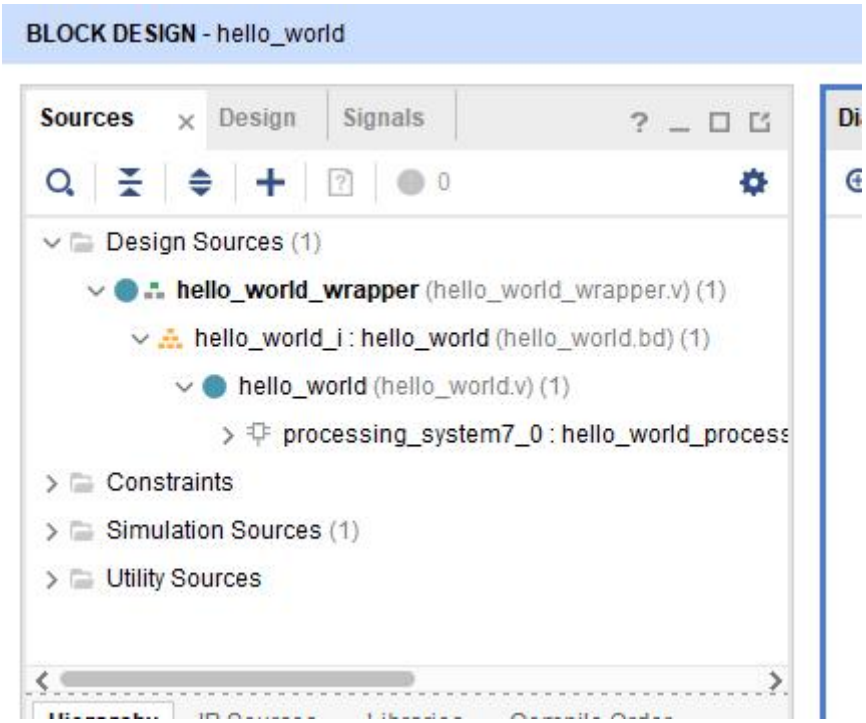


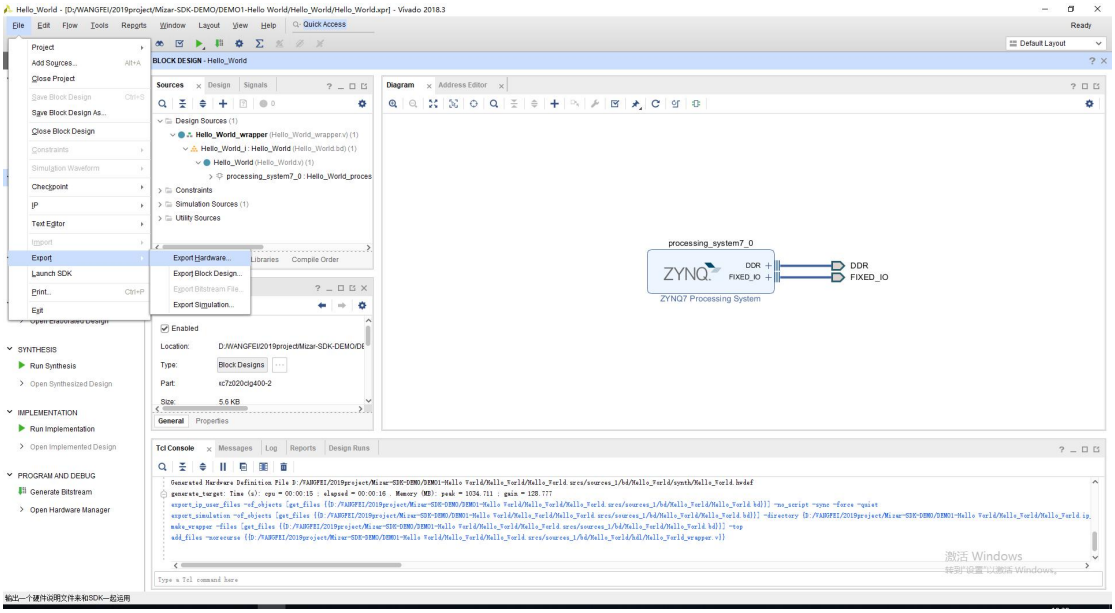
图 1.2.27 建立顶层 HDL



这样我们的硬件设计就创建完成了，完成的 Design Source 结构如下图所示：

图 1.2.28 工程结构

对于用到 PL 部分资源的项目需要添加引脚约束并进行综合、实现并生成 Bitstream 文件。由于本实验未用到 PL 部分，所以无需上述步骤，可直接导出硬件到 SDK 即可。



在菜单栏选择 File > Export > Export hardware。

图 1.2.29 导出硬件到 SDK

在弹出的对话框中可以选则是否包含 Bitstream 文件，因为我们没有生成 Bitstream 文件，所以不勾选，点击“OK”。

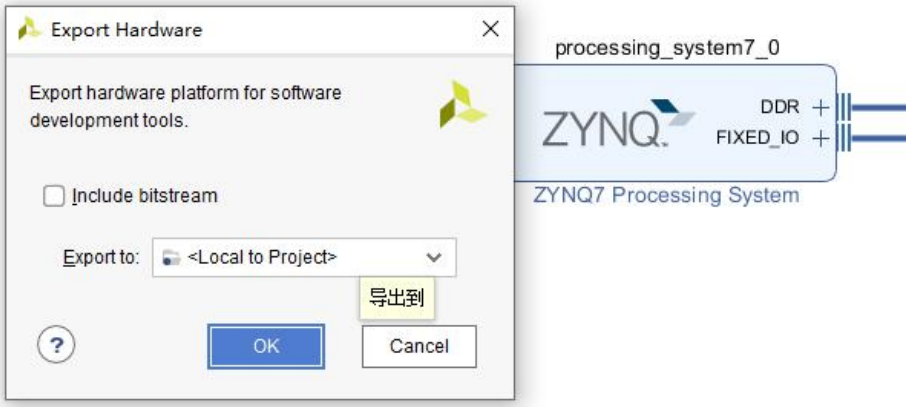


图 1.2.30 导出选项

硬件导出完成后，在菜单栏中选择 File > Launch SDK，启动 SDK 开发环境。如下图所示：

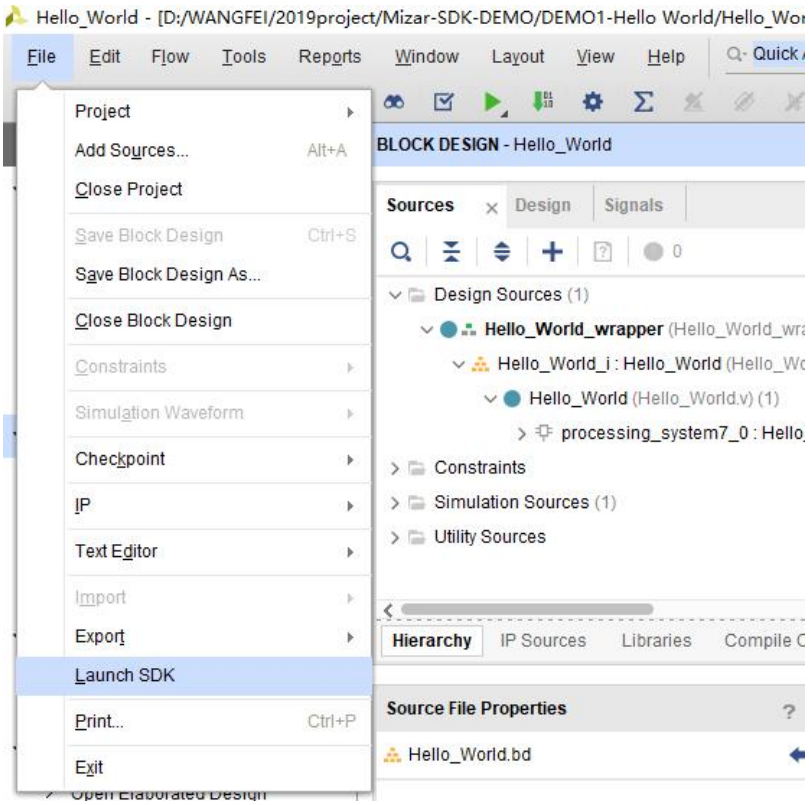
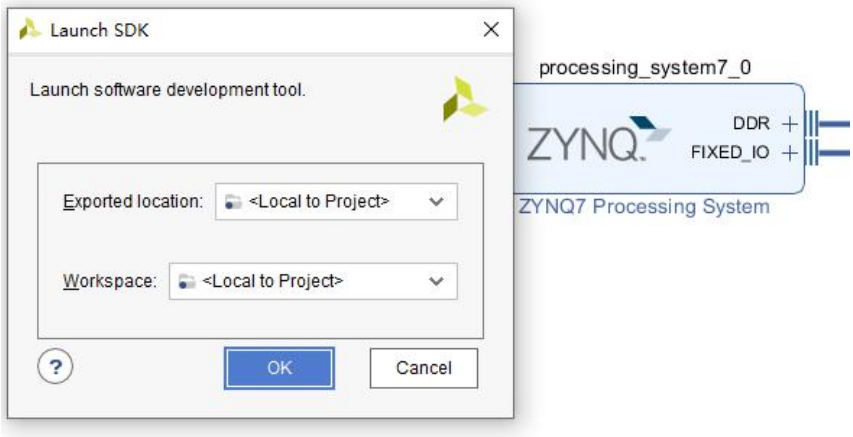


图 1.2.31 启动 SDK 开发环境



在弹出的对话框中，直接点击“OK”，如下图所示：

图 1.2.32 选择硬件路径及工作空间

到这里，我们就完成了系统硬件部分的设计，接下来需要到 SDK 软件中进行应用程序开发，也就是软件设计部分。

1.3 软件设计

完成硬件设计的时候我们将硬件设计导出并打开了 SDK 软件，软件界面如下图所示：

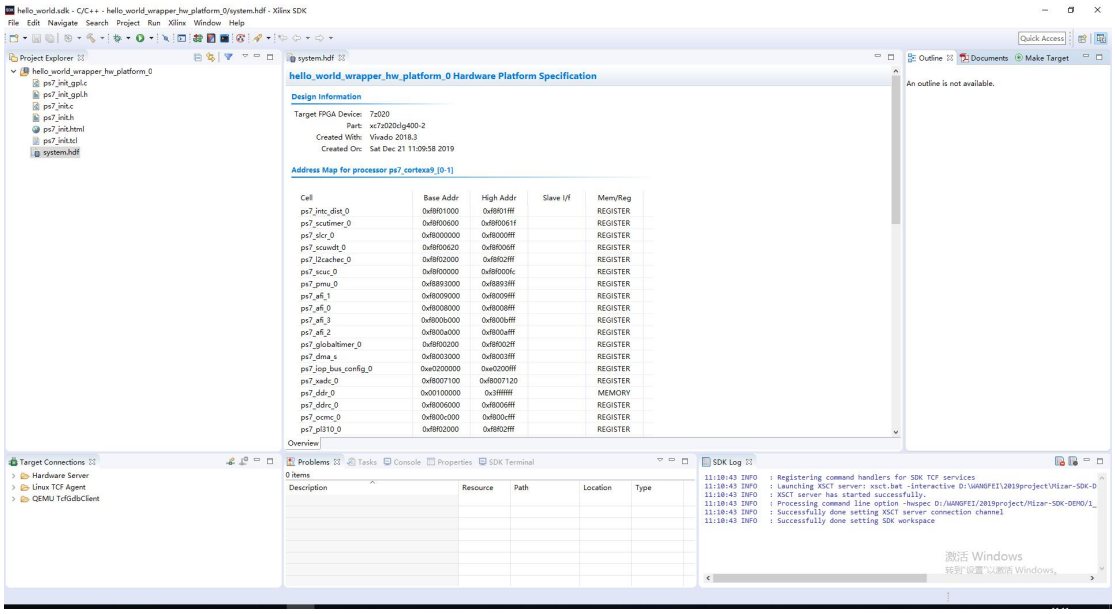
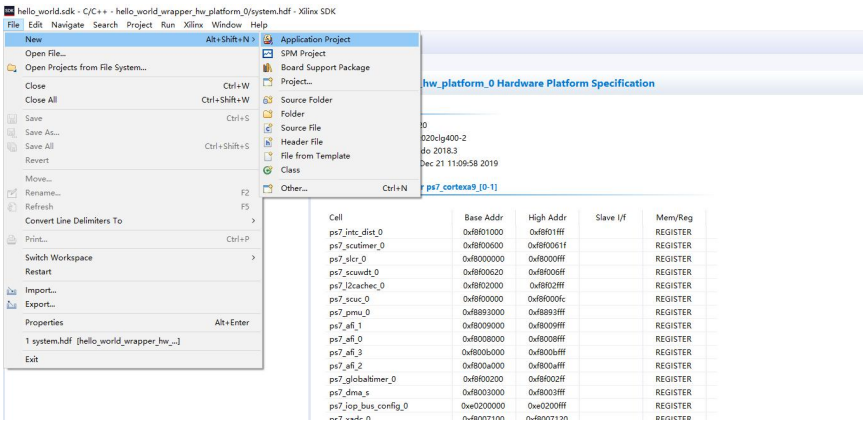


图 1.3.1 SDK 软件界面

在编辑框中可以看到打开的 system.hdf 文件，这就是我们导出的硬件描述文件，上面显示了整个系统的地址映射信息。

现在我们开始软件部分的设计。首先在菜单栏选择 File > New > Application Project，新建一个 SDK 应用工程。在弹出的对话框中，输入工程名“hello_world”，



其他选项保持默认，点击“Next”。

图 1.3.2 建立 SDK 工程

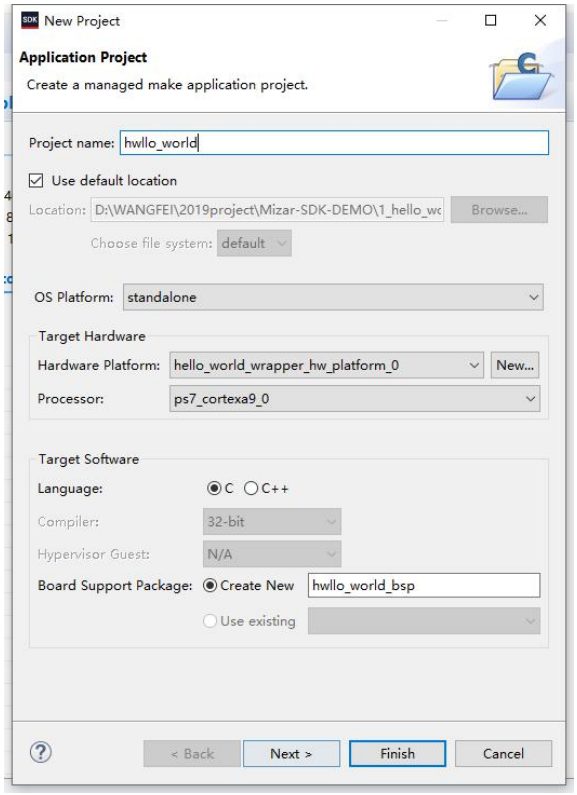
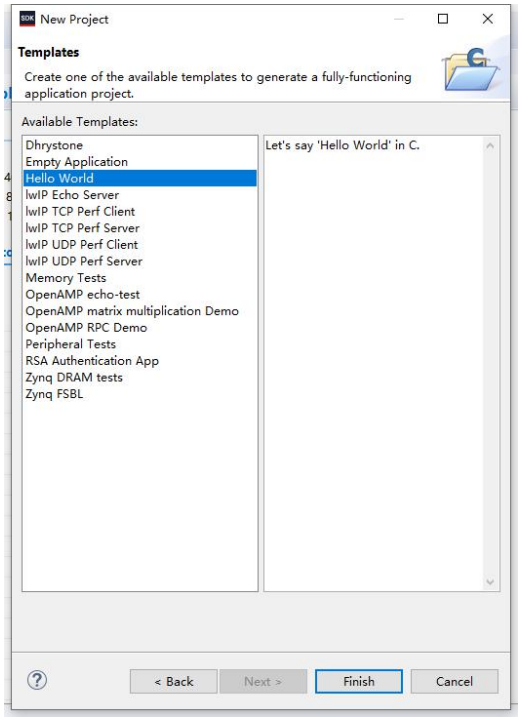


图 1.3.3 配置工程名等设置

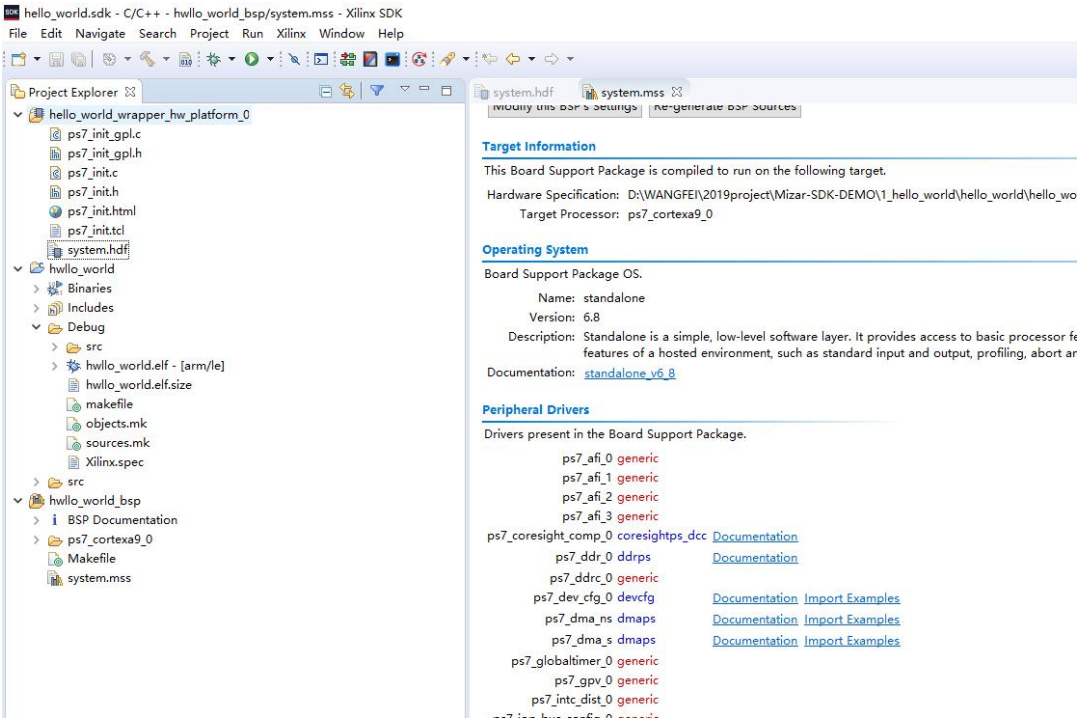
在弹出的对话框中选择希望使用的工程模板，这里我们选择“Hello World”，



然后点击“Finish”。

图 1.3.4 选择工程模板

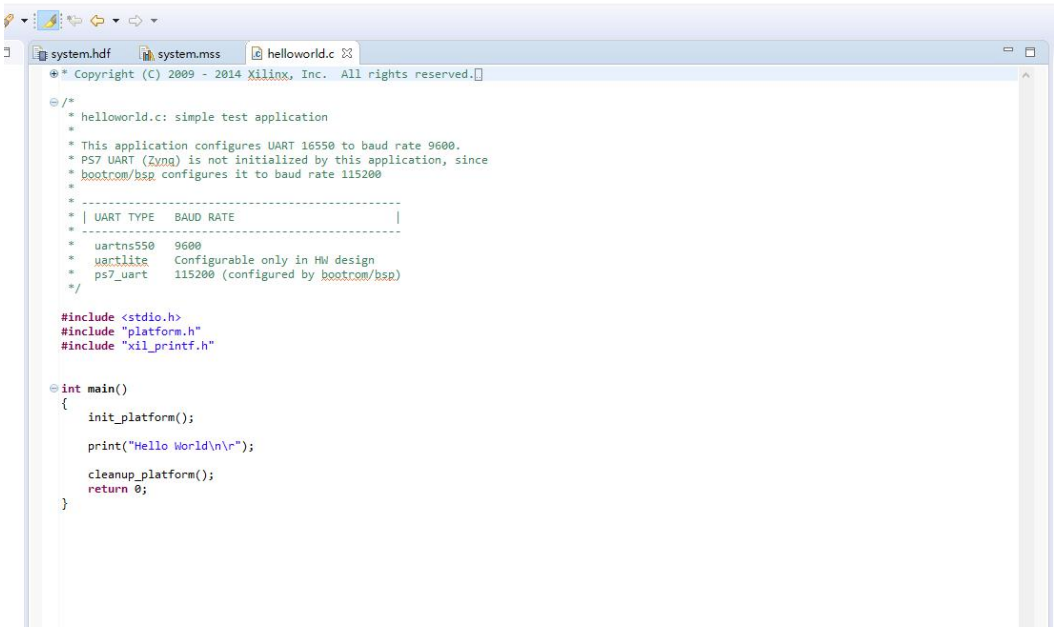
之后 SDK 创建了一个 hello_world 应用工程和 hello_world_bsp 板级支持包（BSP）工程，同时 SDK 会自动对工程进行编译，并生成可执行 ELF 文件



“hello_world.elf”， 如下图所示：

图 1.3.5 创建完成的工程界面

双击打开 hello_world/src 工程目录下 helloworld.c 文件，可以看到源代码



如下：

图 1.3.6 主函数程序

在这里我们可以看到主函数调用了三个子函数，将鼠标箭头停在函数上会显示函数的声明，按住 Ctrl 然后点击函数名就可以跳转到函数定义的位置，我们

```
init_uart()
{
#ifdef STDOUT_IS_16550
    XUartNs550_SetBaud(STDOUT_BASEADDR, XPAR_XUARTNS550_CLOCK_HZ, UART_BAUD);
    XUartNs550_SetLineControlReg(STDOUT_BASEADDR, XUN_LCR_8_DATA_BITS);
#endif
    /* Bootrom/BSP configures PS7/PSU UART to 115200 bps */
}

void
init_platform()
{
    /*
     * If you want to run this example outside of SDK,
     * uncomment one of the following two lines and also #include "ps7_init.h"
     * or #include "ps7_init.h" at the top, depending on the target.
     * Make sure that the ps7/psu_init.c and ps7/psu_init.h files are included
     * along with this example source files for compilation.
     */
    /* ps7_init();*/
    /* psu_init();*/
    enable_caches();
    init_uart();
}

void
cleanup_platform()
{
    disable_caches();
}
```

可以看到 init_platform()函数定义如下：

图 1.3.7 init_platform()函数定义

在这个函数中使能了 caches 和初始化 uart，实际上本实验只需要初始化 uart 就可以了，另一个函数是工程模板为了兼容其他平台增加的，在本实验中没有实际作用，可以删除。为了保持工程模板的完整，我们这里保留这个函数。

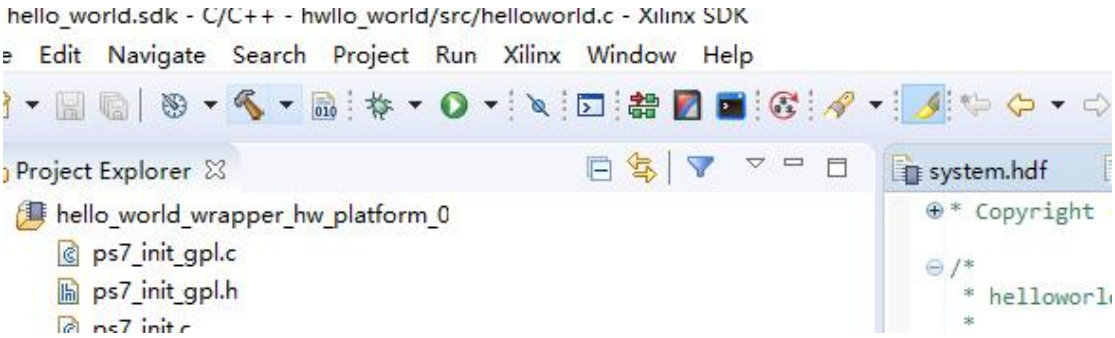
在 SDK 中修改并保存源文件后，软件会自动对文件进行重新编译。可以在下面的控制台面板中查看编译进度，编译完成后会显示“Finished building”。如下

```
Problems  Tasks  Console  Properties  SDK Terminal
CDT Build Console [hwllo_world]
' '
'Invoking: ARM v7 Print Size'
arm-none-eabi-size hwllo_world.elf  |tee "hwllo_world.elf.size"
   text    data     bss     dec     hex filename
  19352    1144   22568   43064    a838 hwllo_world.elf
'Finished building: hwllo_world.elf.size'
' '

11:13:45 Build Finished (took 2s.93ms)
```

图所示：

图 1.3.8 编译完成

另外也可以点击工具栏中的“Build All”或者快捷键 Ctrl + B 来编译工程，



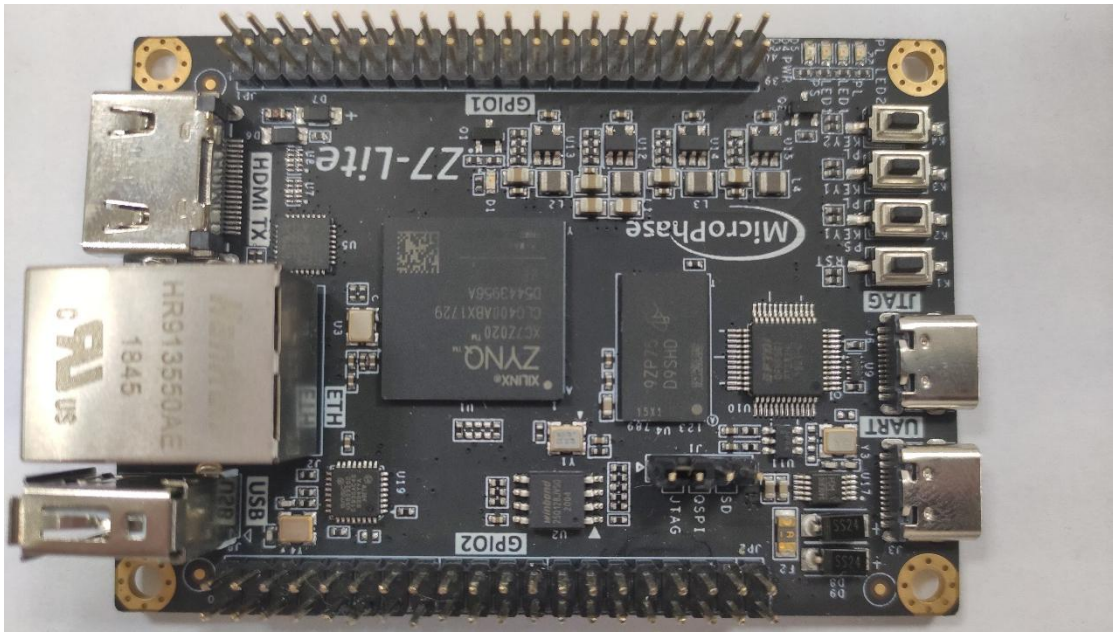
如下图所示：

图 1.3.9 编译工程

到此我们就完成了软件设计的工作，接下来验证功能能否实现。

1.4 下载验证

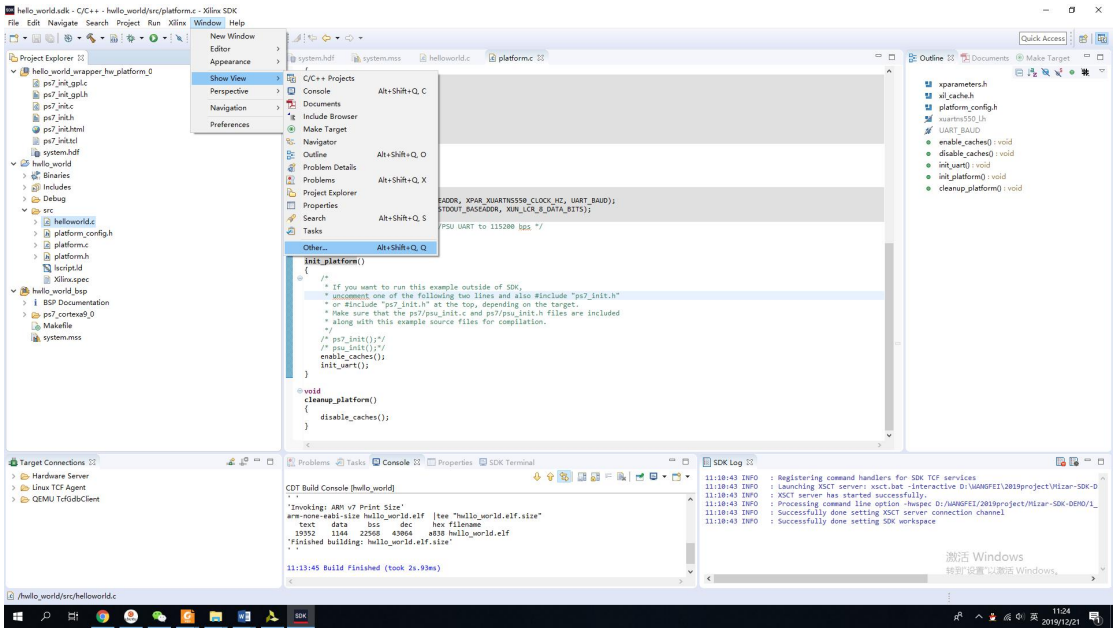
本次实验我们使用 JTAG 的下载方式烧录程序，烧录之前需要将开发板设置为 JTAG 启动方式，按照下图所示调整跳线帽位置。本型号开发板使用跳线帽来调整启动方式，短接下方两根排针为 JTAG 启动，短接中间两根为 QSPI 启动，短



接上面两根为 SD 卡启动。

图 1.4.1 调整启动方式为 JTAG

菜单栏中选择 Window > Show View > Other，在 弹出的 Show View 窗口中



搜索 Terminal，选中，点击“OK”。

图 1.4.2 窗口菜单

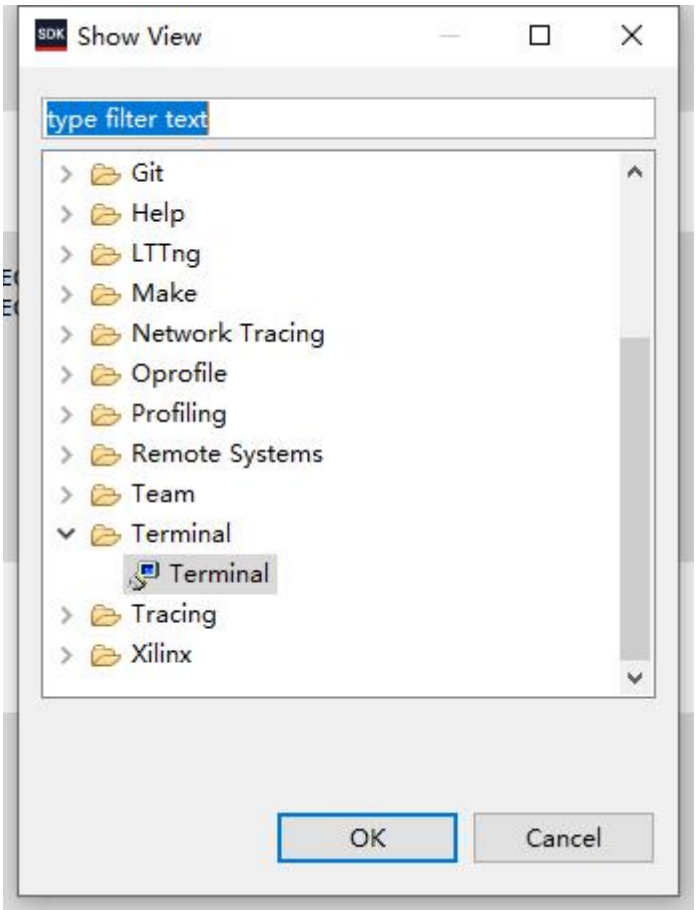


图 1.4.3 添加串口工具

本型号开发板不带有电源接口，直接使用 JTAG 接口供电，最大负载电流为 500mA。将 JTAG 接口和 UART 接口通过 type-c 线连接到电脑，如下图所示：

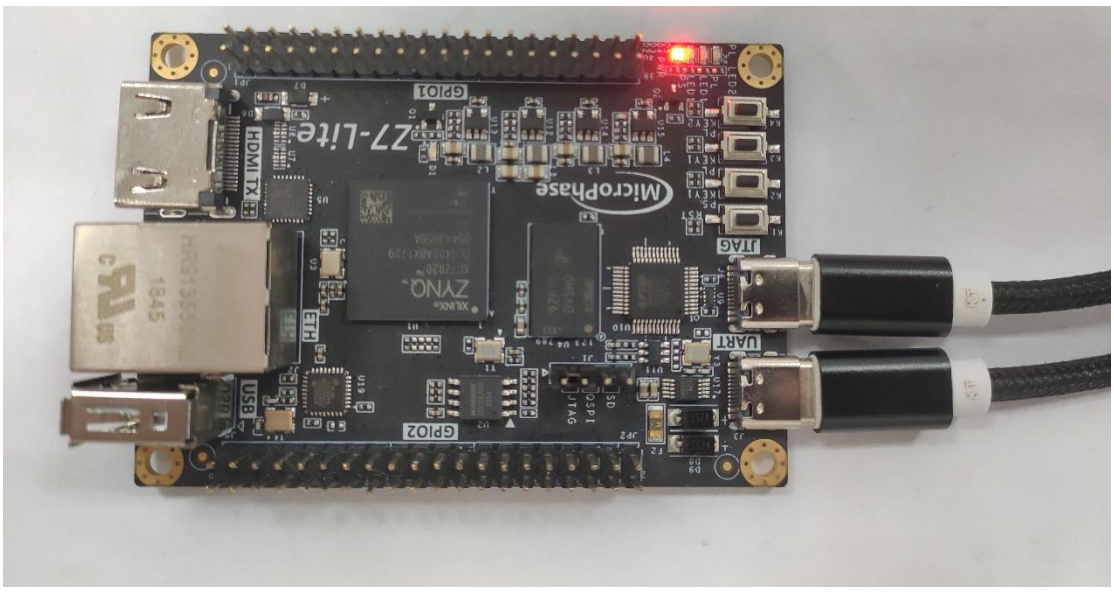


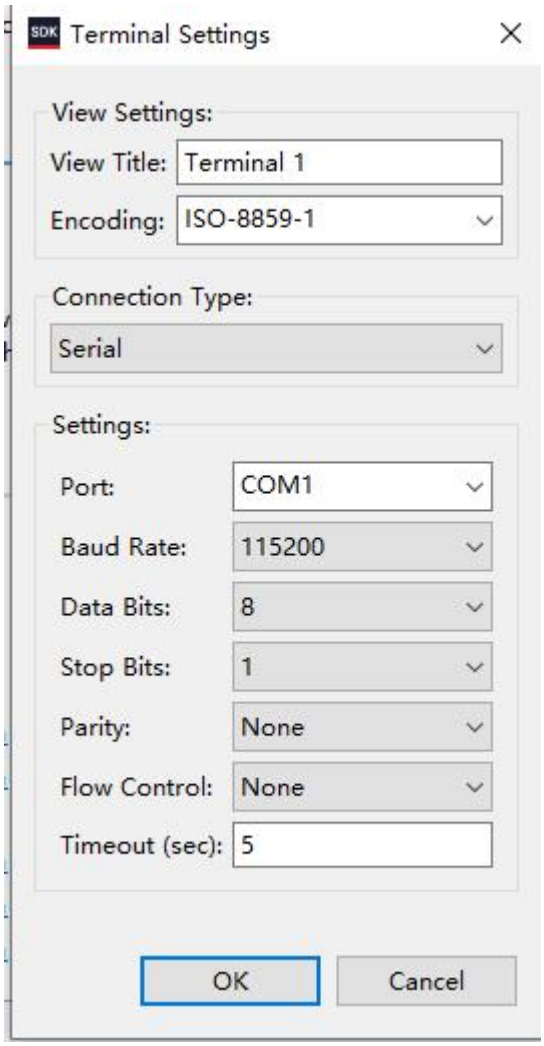
图 1.4.4 开发板供电及接口连接

打开电脑的设备管理器界面查看串口的端口号，我们使用的串口芯片是 CH340，所以端口号为 COM8，如下图所示：

图 1.4.5 查看串口端口号

在左下角 Terminal 窗口，点击连接按钮，如图 1.4.6，

图 1.4.6 Terminal 窗口



在弹出的窗口中设置要连接的串口参数，具体如下图所示：

图 1.4.7 串口参数设置

连接成功后，下载程序。右键点击 hello_world 工程，在弹出的菜单栏中选择 Run as > 1 Launch on Hardware(sysntem Debugger)，如下图所示：

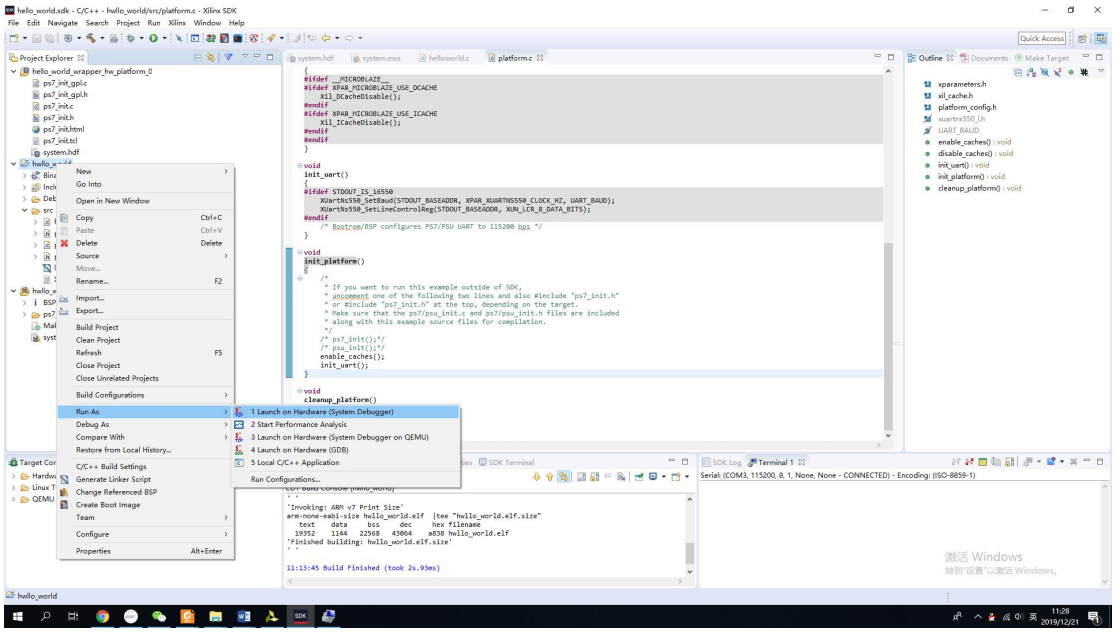


图 1.4.8 烧录程序

烧录完成后程序会自动运行，可以在 Terminal 窗口看到芯片通过串口发送

的字符“Hello World”，如下图所示：

图 1.4.9 运行结果

窗口成功打印出了“Hello World”字符串，说明我们的硬件和软件设计都是

没有问题的。

1.5 章节小结

通过一个简单的工程，我们初步了解了一些 ZYNQ SDK 项目的开发流程和一些软件的操作方法，在此基础上，可以通过多次重复本实验熟悉各个步骤的操作流程和各选项的功能，熟悉开发环境，为以后的学习工作打下坚实的基础。

基本实验工程的建立，硬件描述文件的导出操作在所有项目中都是相同的，后续的教程不再介绍这些基本操作。

第 2 章　　按键控制 LED 实验

ZYNQ 芯片的 GPIO 分为三种，分别是 MIO、EMIO 和 AXI_GPIO，其中 MIO 是 PS 的专用 I/O，作为普通输入输出或者外设驱动 I/O 来使用，EMIO 是扩展的 MIO，当 MIO 不够时，可以使用部分 PL 的 I/O 做为 MIO 来使用，而 AXI_GPIO 是一个可以实现 GPIO 功能的 IP 核，PS 端可以通过 AXI 总线对其进行操作。在需要众多 I/O 口的应用场合，AXI_GPIO 可以将所有可用引脚作为 I/O 口使用。本实验我们就来了解一下 MIO 的一些相关知识，以及 MIO 最基础功能的使用方法。

本章包括以下几部分内容：

2.1 基础知识介绍

2.2 硬件设计

2.3 软件设计

2.4 下载验证

2.5 章节小结

2.1 基础知识介绍

在芯片的数据手册中我们可以看到芯片 GPIO 的组成，如图 2.1.1 所示，GPIO 分为 4 个 Bank，其中 Bank0 有 32bit，Bank1 有 22bit 共 54bit 属于 MIO，Bank2 和 Bank3 各有 32bit 属于可扩展的 EMIO，所以对于用户而言有最多 64 个 I/O 可以使用。另外由于 AXI_GPIO 的使用方法有所不同，所以不在这里列出，在后续的实验中会进行介绍。

MIO 是沟通 PS 内核和外设的桥梁，PS 所有的外设都可以通过 MIO 访问，也可以将外设与 MIO 连接，在使用上具有极大的灵活性，为项目硬件设计提供了许多便利，降低了电路板布局布线的难度。但是即使如此，在给不同外设分配引脚时也不是随心所欲的，必须遵循一定的规范。

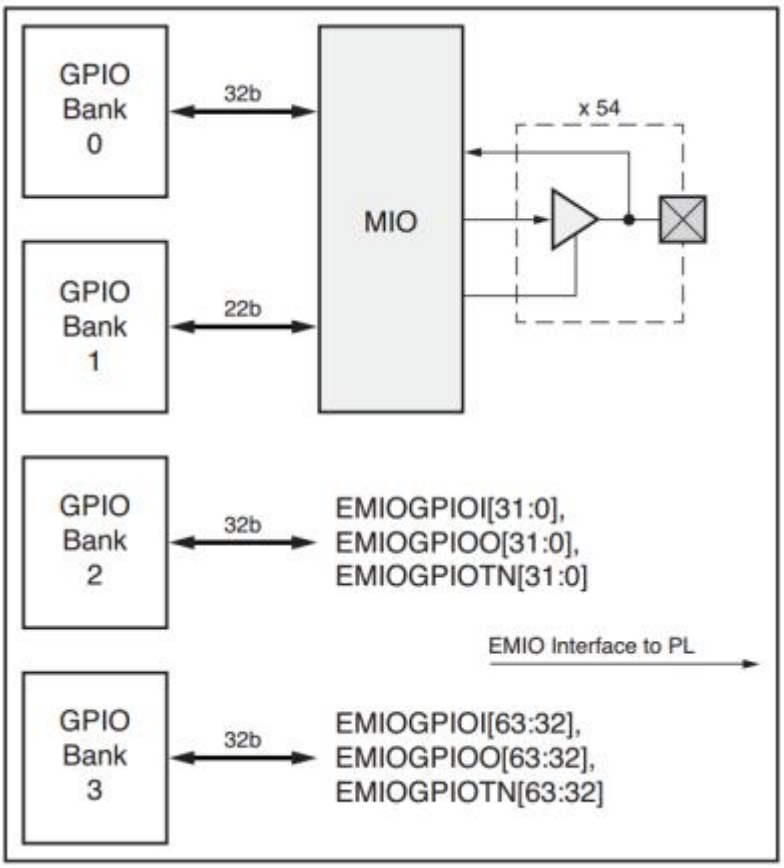


图 2.1.1 芯片 GPIO 框图

如图 2.1.2 所示，可以看到许多外设能够分配的 I/O 是有限制的，当你在创建 Vivado 工程进行硬件设计时，使能了某个外设时也就分配了该外设连接的引脚，不需要后续添加引脚约束。像 QSPI、Ethernet0、Ethernet1 只能分配在固定引脚。所以在进行硬件设计时必须要考虑好引脚分配的问题，合理分配引脚，才能降低后续电路板绘制的难度。

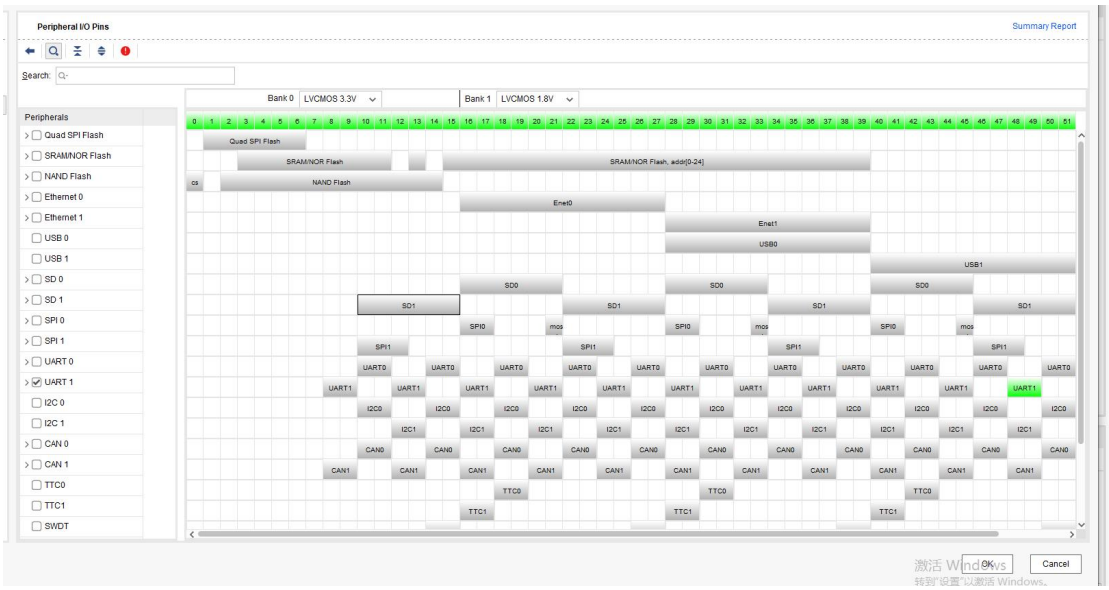


图 2.1.2 MIO 可分配功能

2.2 硬件设计

按照之前实验的步骤建立 Vivado 工程，命名为“mio_key_led”，如下图所示。注意在建立工程时 ZYNQ 芯片型号的选择，根据开发板选择相应的芯片。

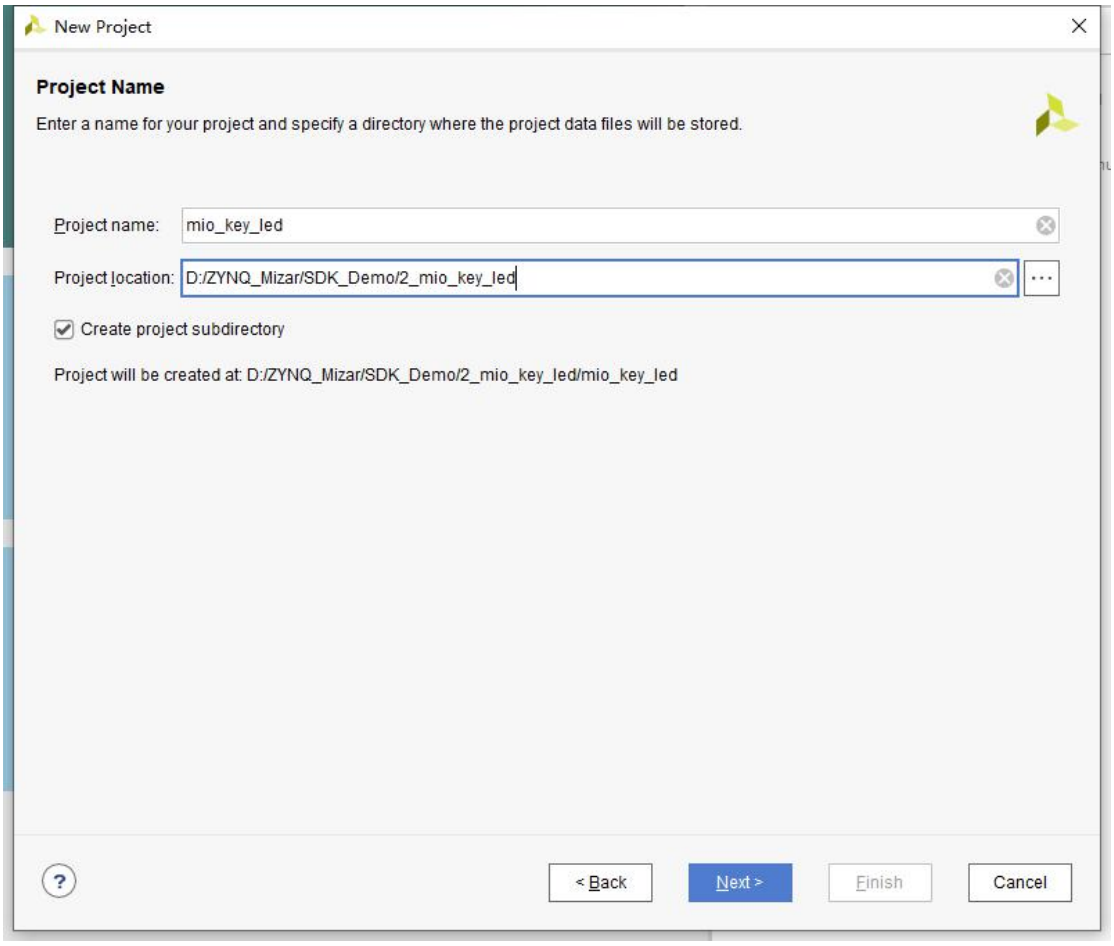


图 2.1.1 建立工程

工程创建完成后，按照之前的步骤创建 Block Design，添加 ZYNQ7 Processing System IP 核，双击 IP 核进入配置页面基本配置与之前相同，在此不再赘述，重点讲解本次实验需要配置的项目。在本次实验中，我们需要用到 MIO，所以在配置中使能 GPIO MIO，如下图所示：

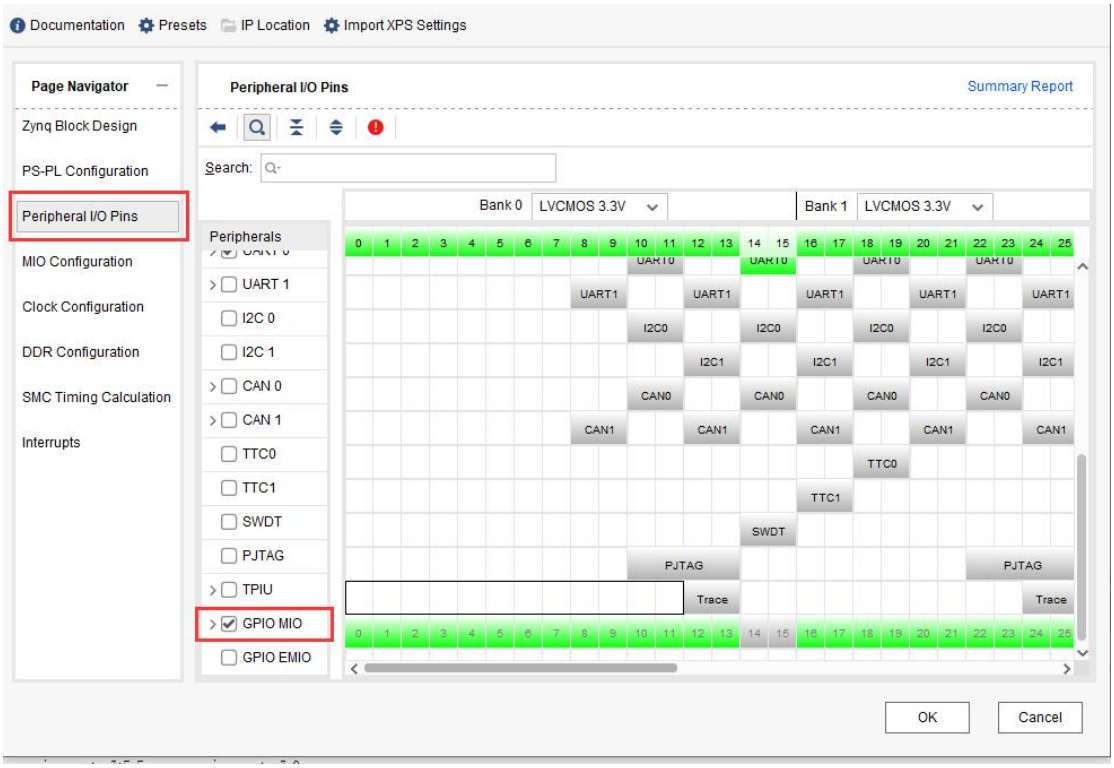


图 1.2.2 使能 MIO

配置完成后点击“OK”保存并返回。接下来参照之前实验的操作在 sources 面板中，右键选择 Design Sources 下的 mio_key_led_wrapper 中的 mio_key_led_i: mio_key_led，在弹出的菜单中选择 Generate Output Products...生成输出文件，然后在相同位置右键选择 Create HDL Wrapper...生成顶层 HDL 文件。由于本次实验不需要用到 PL 部分，所以不需要生成 Bitstream 文件，直接导出即可。

这样就完成了硬件方面的设计。

2.3 软件设计

完成硬件设计并导出到 SDK 后，我们就可以开始进行软件方面的设计工作，打开的 SDK 页面如下图所示：

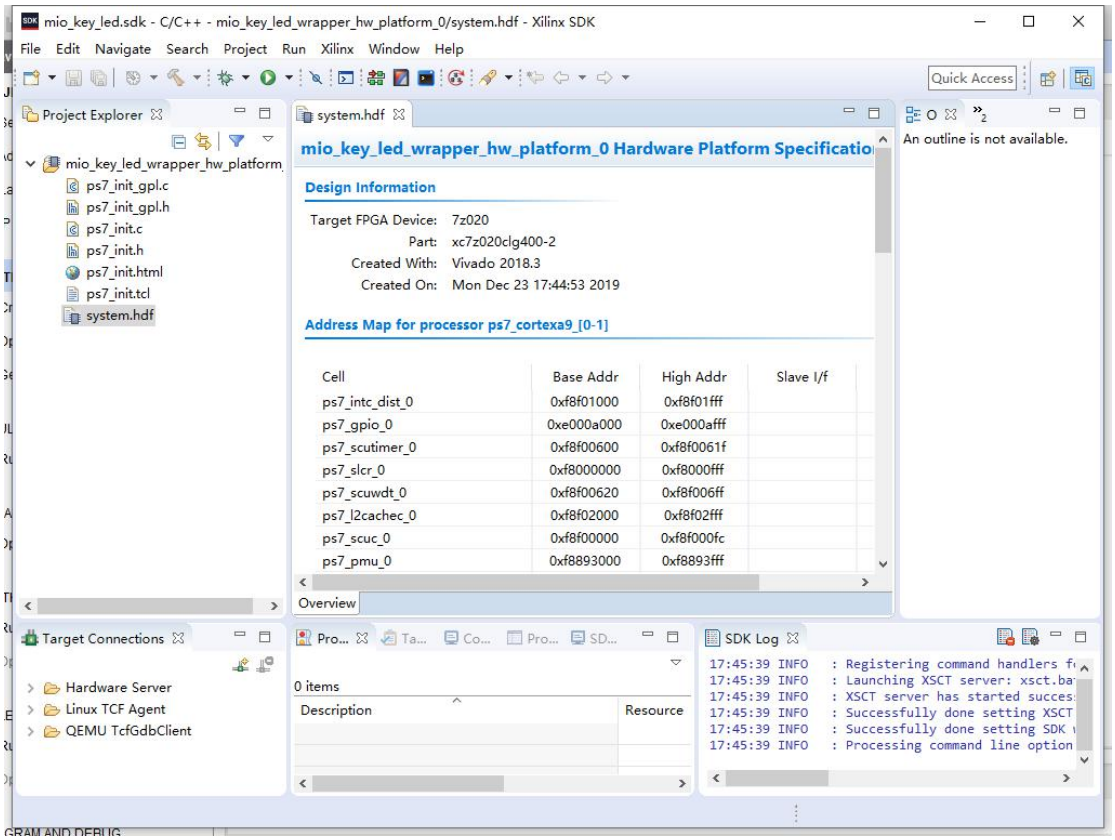
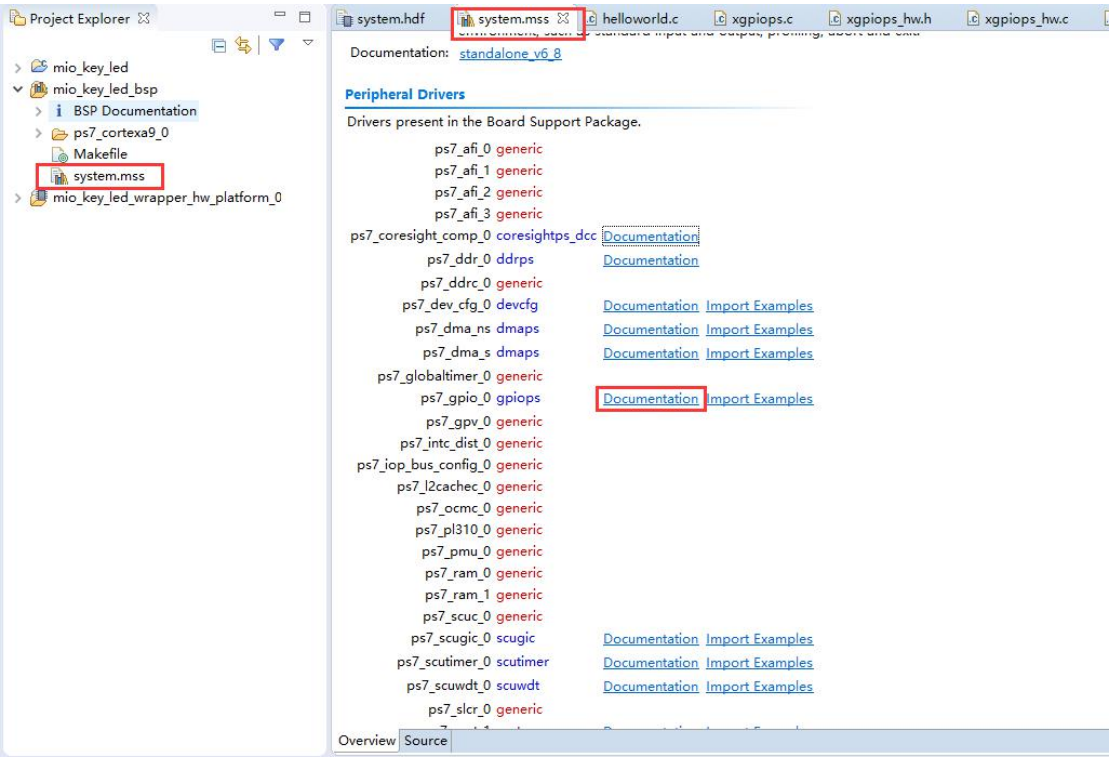


图 2.3.1 SDK 界面

我们按照之前的步骤创建一个新的“hello world”工程，命名为“mio_key_led”创建好之后打开板级支持包 mio_key_led_bsp 下面的 system.mss 文件如图 2.3.2 所示，在文件中可以看到本实验所用到的芯片上的所有资源，在这个文件下点击外设后面的“Docunmentation”可以打开关于该外设的数据手册，里面会详细描述该外设的功能，使用方法，库函数等信息，能有效帮助我们学会使用该外设，在项目开发过程中多阅读这些文档能加深我们我们的理解。除此之外，Xilinx 官方还提供了一些示例程序供使用者参考，点击外设后面的“Import Example”可以导入这些例程，这些例程都是该外设的基本应用，对于初学者是非常好的参考。



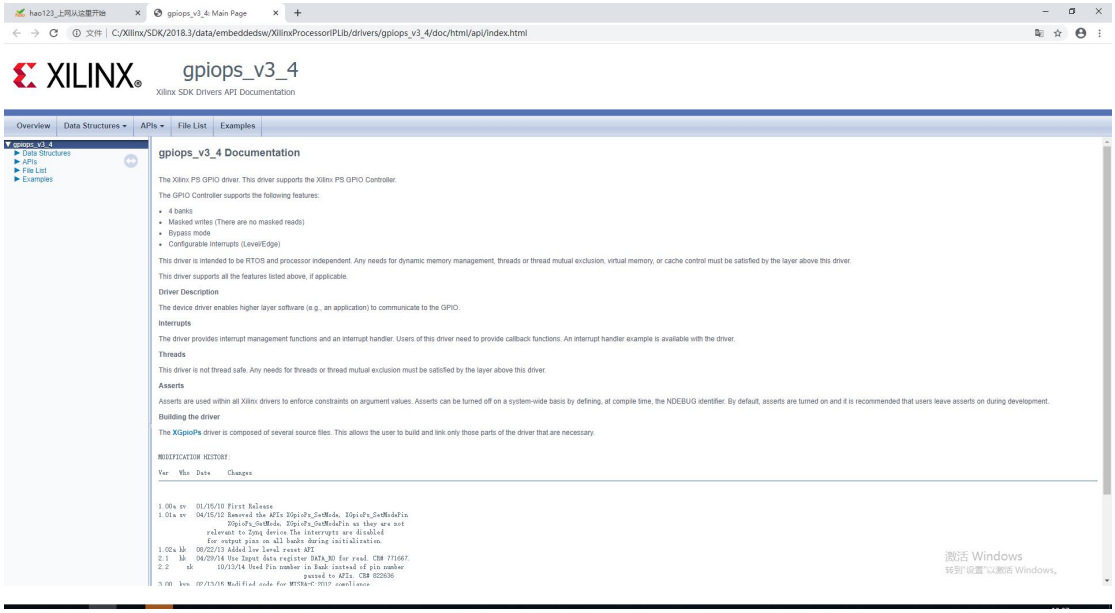


图 2.3.2 system.mss 文件

图 2.3.3 gpio 数据手册

在本实验中我们没有使用官方例程，读者可自行查阅，交叉对比我们的程序，强化学习效果。接下来我们讲解一下关于库函数的相关知识。

Xilinx 官方提供了功能详尽库函数，免去了用户在使用外设时查阅手册配置

寄存器的烦恼，极大的提高了开发效率，降低了出错的概率。相关库函数的文件保存在 mio_key_led_bsp 板级支持包下面 ps7_cortrxa9_0> libsrc> gpios_v3_4 >src 里面。如下图所示：

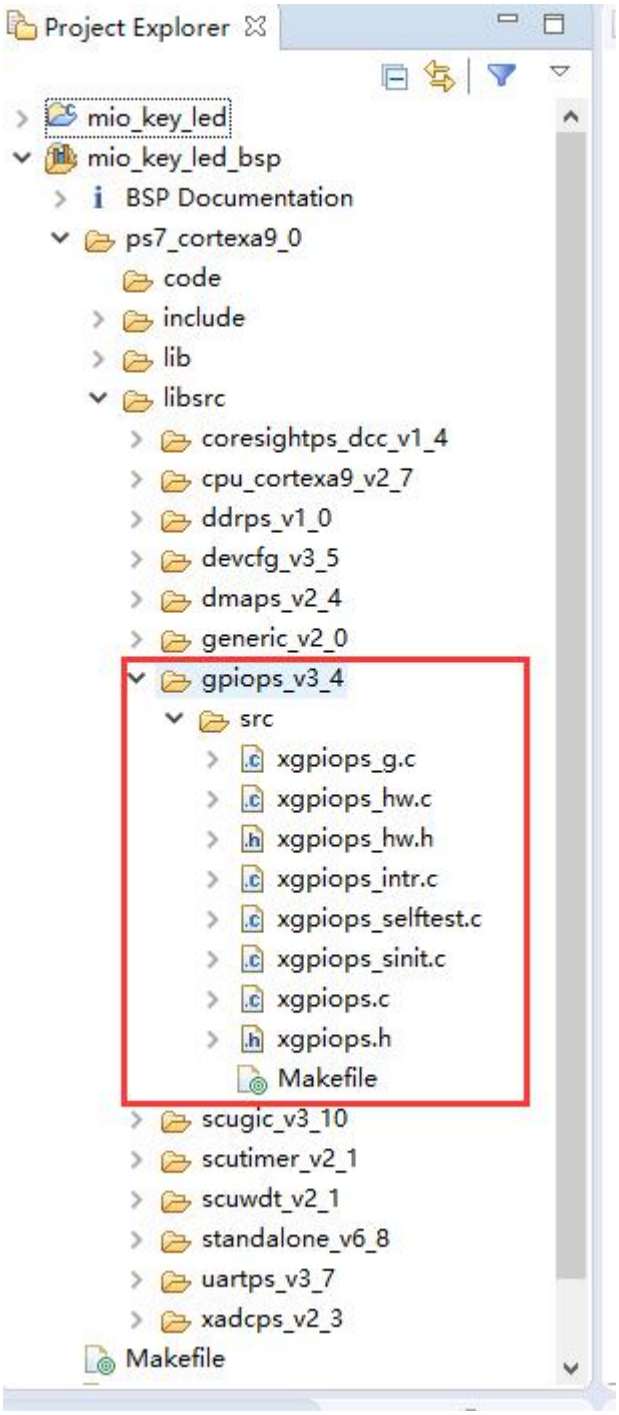


图 2.3.4 GPIO 库函数文件

在库函数的使用中，我们不关心函数到底是怎样实现功能的，只需要知道

某个函数的功能以及其参数的含义即可。对于该实验，我们的目的是利用 I/O 口的输入输出功能来实现按键控制 LED 的亮灭。在程序中我们需要做的事情就是 1、初始化 GPIO；2、配置 LED 接口为输出，KEY 接口为输入；3、LED 接口输出使能；4、读取 KEY 接口状态，根据 KEY 接口状态改变 LED 接口的高低电平输出。

在明确了需要完成的任务之后，要做的事情就是查找相关的库函数，并确定要配置的参数等相关设置。本实验的代码如下：

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xgpiops_hw.h"



#define MIOLED1 0 //LED1连接到MIO0

#define MIOKEY1 9  //KEY1连接到MIO0

#define input  0
#define output 1

XGpioPs Gpios;

int main()
{
    init_platform();

    int Status;
     XGpioPs_Config *ConfigPtr;

     //初始化GPIO
     print("MIO Test! \n\r");
     ConfigPtr = XGpioPs_LookupConfig(XPAR_XGPIOPS_0_DEVICE_ID);
     Status = XGpioPs_CfgInitialize(&Gpios, ConfigPtr,
     ConfigPtr->BaseAddr);
     if (Status != XST_SUCCESS){
     return XST_FAILURE;
     }
```

```
79      //设置LED接口方向为输出
80     XGpioPs_SetDirectionPin(&Gpios,MIOLED1 ,output);
81     //设置KEY接口方向为输入
82     XGpioPs_SetDirectionPin(&Gpios,MIOKEY1 ,input);
83     //使能LED接口输出
84     XGpioPs_SetOutputEnablePin(&Gpios,MIOLED1,1);
85     while(1)
86     {
87         //读取KEY状态，改变LED输出电平
88         if(XGpioPs_ReadPin(&Gpios,MIOKEY1))
89             {XGpioPs_WritePin(&Gpios,MIOLED1,1);}
90         else
91             XGpioPs_WritePin(&Gpios,MIOLED1,0);
92
93     }
94
95     cleanup_platform();
96     return XST_SUCCESS;
97 }
```

该程序实现了按键控制 LED 的功能，按键按下 LED 点亮，按键松开 LED 熄灭。在上述代码中我们添加了一些注释描述了各函数的功能，所以不再进行详细讲解。针对其他型号的开发板只需修改宏定义中对应的 MIO 号即可。

程序修改完成后点击保存，软件自动编译生成可执行文件。至此软件设计完成。

2.4 下载验证

完成硬件和软件的设计之后，我们就可以将程序烧录到开发板上进行验证。首先按照之前的方式连接串口，可以通过串口打印判断程序是否在运行。

这次我们讲解另一种下载方式。点击菜单栏 Run>Run Configuration，如图 2.4.1 所示，在弹出的界面中双击 Xilinx C/C++ application（System Debugger），之后点击 Run 即可将程序下载到开发板上。

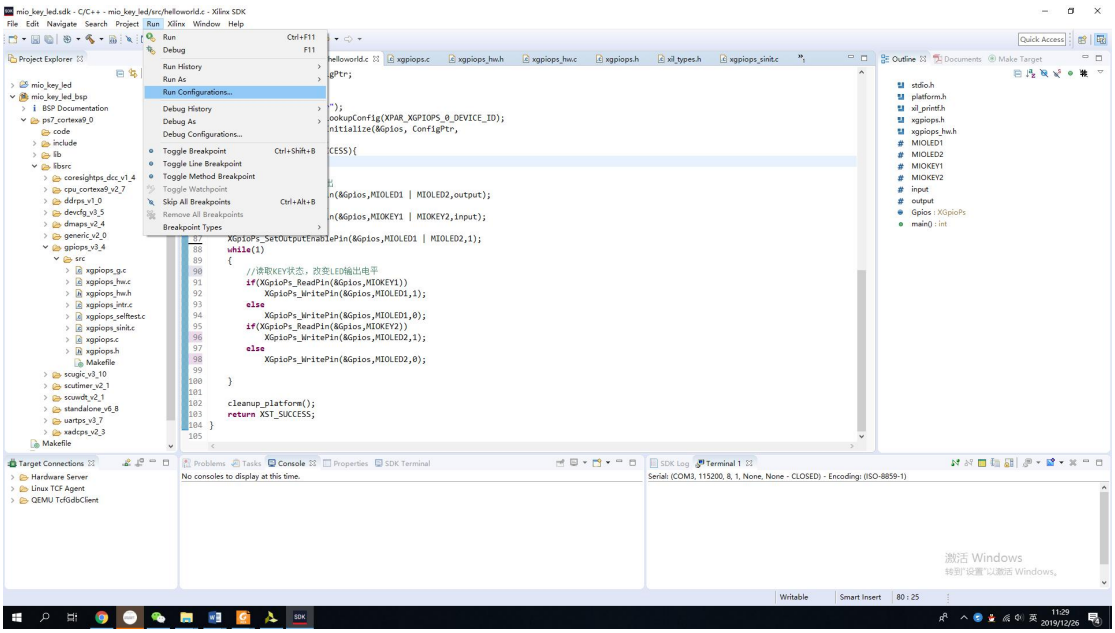


图 2.4.1 进入 Run Configuration

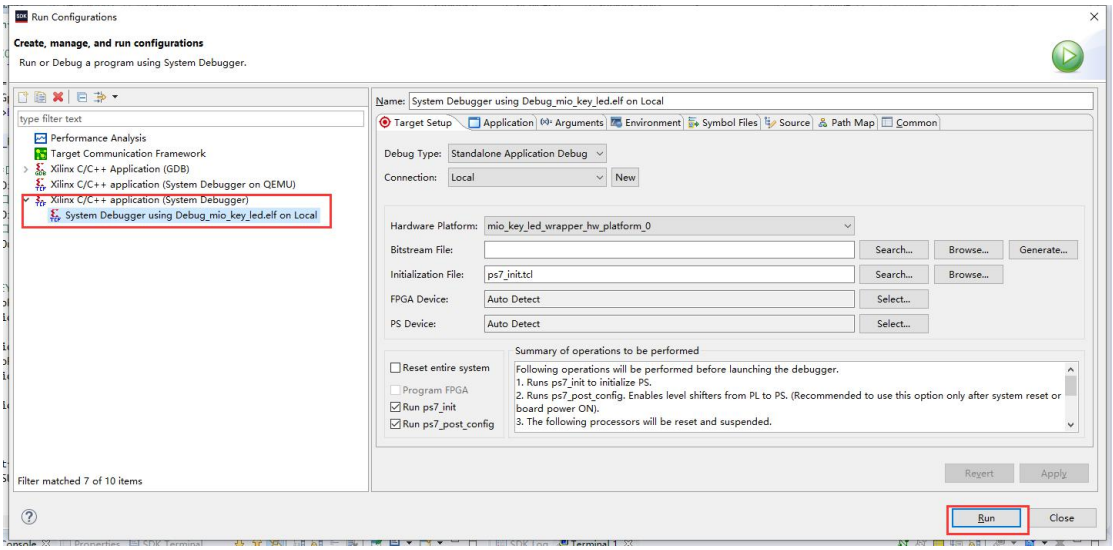


图 2.4.2 烧录程序

烧录完成后可以开发板上可以看到按下 PS_KEY1，PS_LED1 点亮，说明硬件设计和软件设计都是正确的。

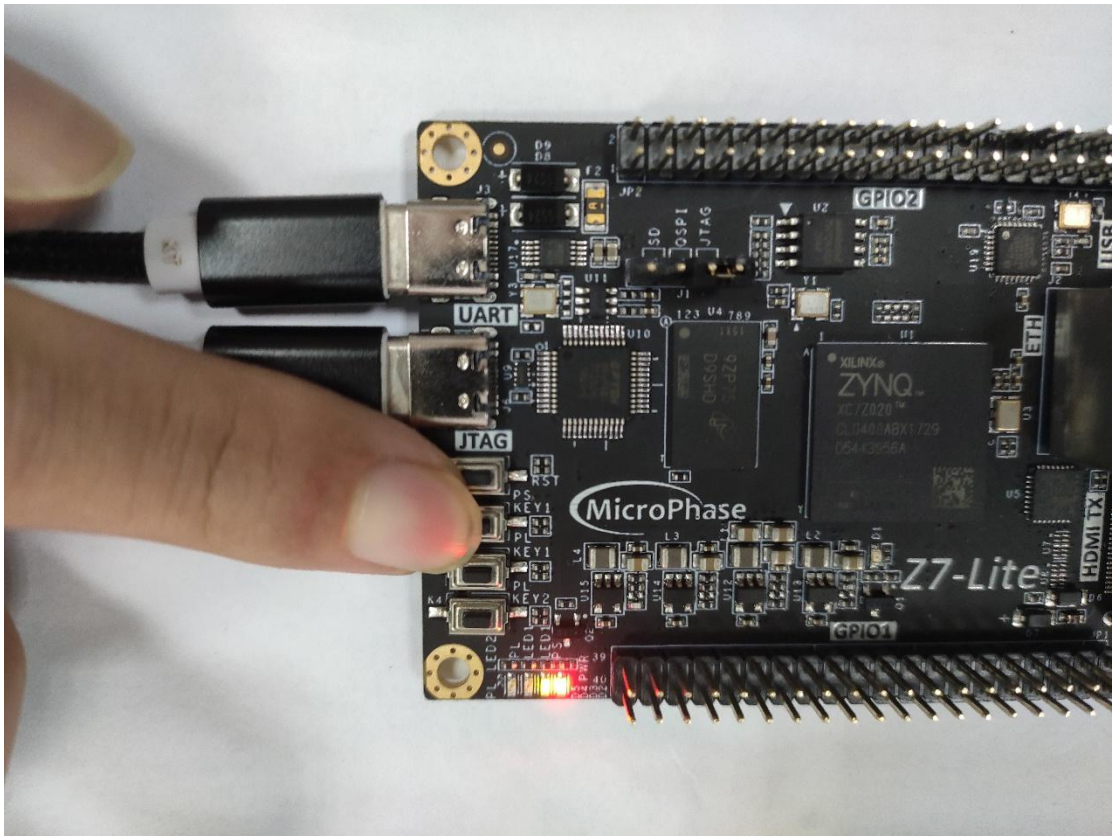


图 2.4.3 实验现象

2.5 章节小结

通过本次实验，我们了解了 ZYNQ 操作 MIO 的基本方法，在此过程中了解了官方数据手册及例程的使用方法。对于 ZYNQ，Xilinx 官方出了很多详细的说明文档和例程，用户要学会多研究这些资源，能够有效提升学习的效果。

第 3 章　　外部中断实验

在之前的实验中，我们用轮询的方式，不停的扫描按键 I/O 口的状态来控制 LED 的亮灭，这种方式对于简单小型的项目还可以接受，但对于一些比较复杂的项目而言不仅效率低下而且浪费了大量的系统资源，严重影响系统的性能。这就需要一种更加高效的方式来处理一些突发事件。中断作为系统最重要的组成部分之一就承担了这样的工作，它在节省了大量的系统资源的同时监控一些突发事件的产生，及时向系统申请暂停主程序转而处理突发事件，大大提高了系统运行的效率。中断通常由硬件处理单元、外部设备或者软件本身产生。

本章我们将通过 GPIO 接口中断的实验来初步认识 ZYNQ 的中断系统。本章包括以下几个部分：

3.1 实验简介

3.2 硬件设计

3.3 软件设计

3.4 下载验证

3.5 章节小结

3.1 实验简介

本实验我们通过实现按键中断控制 LED 亮灭来了解 ZYNQ 的中断系统，学习如何使用外部输入中断，了解中断配置的流程、相关函数的作用以及各参数的含义，更进一步认识 ZYNQ 芯片的功能和系统组成。

本实验将按键按下作为中断触发信号，检测到中断之后改变 LED 的状态作为检测到中断信号的指示。

3.2 硬件设计

打开开发板的原理图可以看到按键部分的电路设计如下图所示，按键未按下时 I/O 口为高电平，按下后变为低电平，过程中会产生一个下降沿信号。芯片的外部中断可以分为边沿触发和电平触发，我们本实验选取按键的下降沿信号作为触发信号。

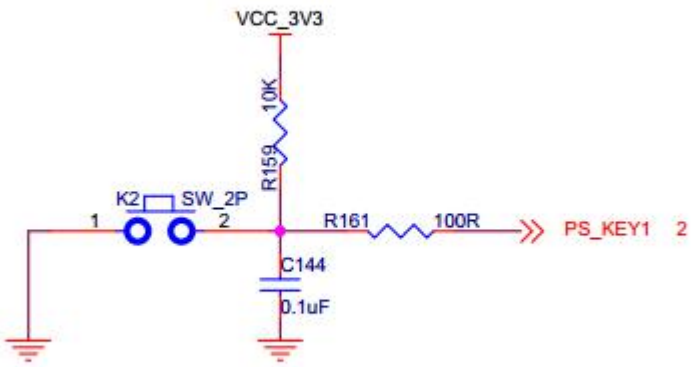
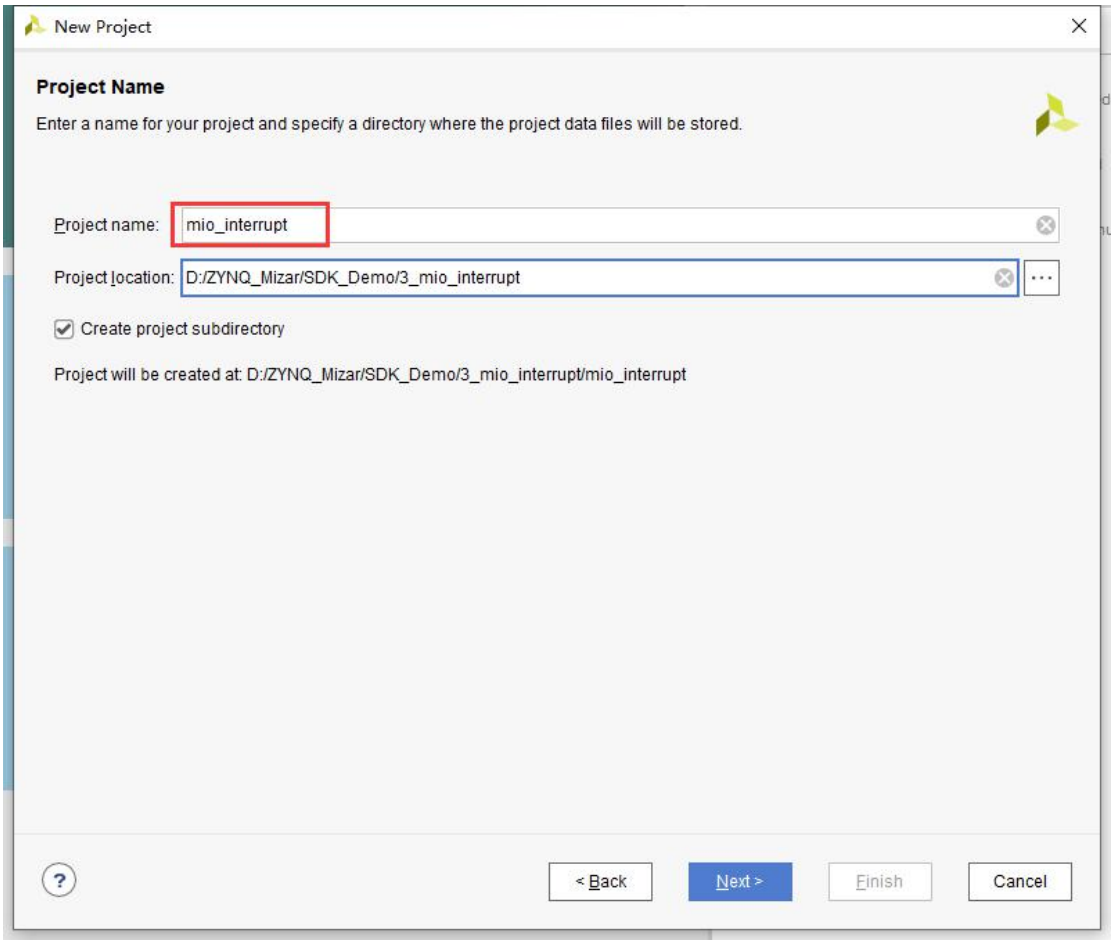


图 3.2.1 按键电路

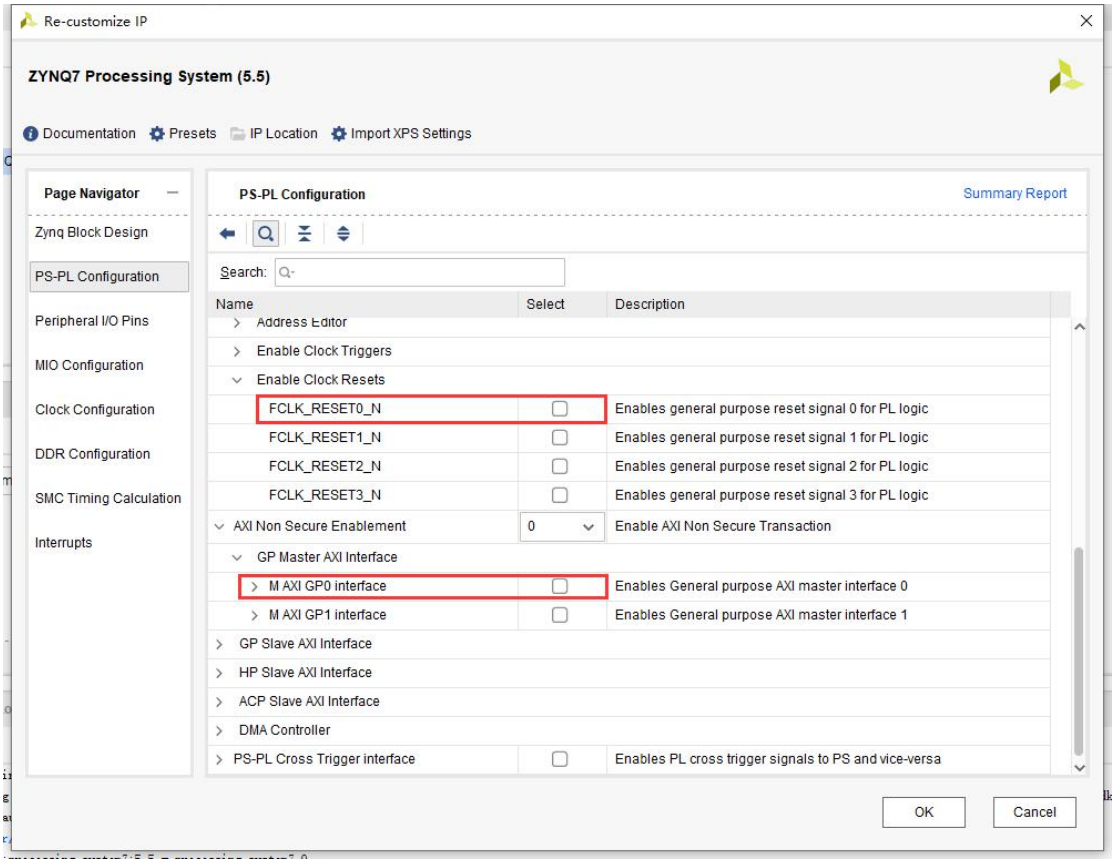
我们按照之前的步骤建立一个新的 Vivado 工程，工程名为“mio_interrupt”，

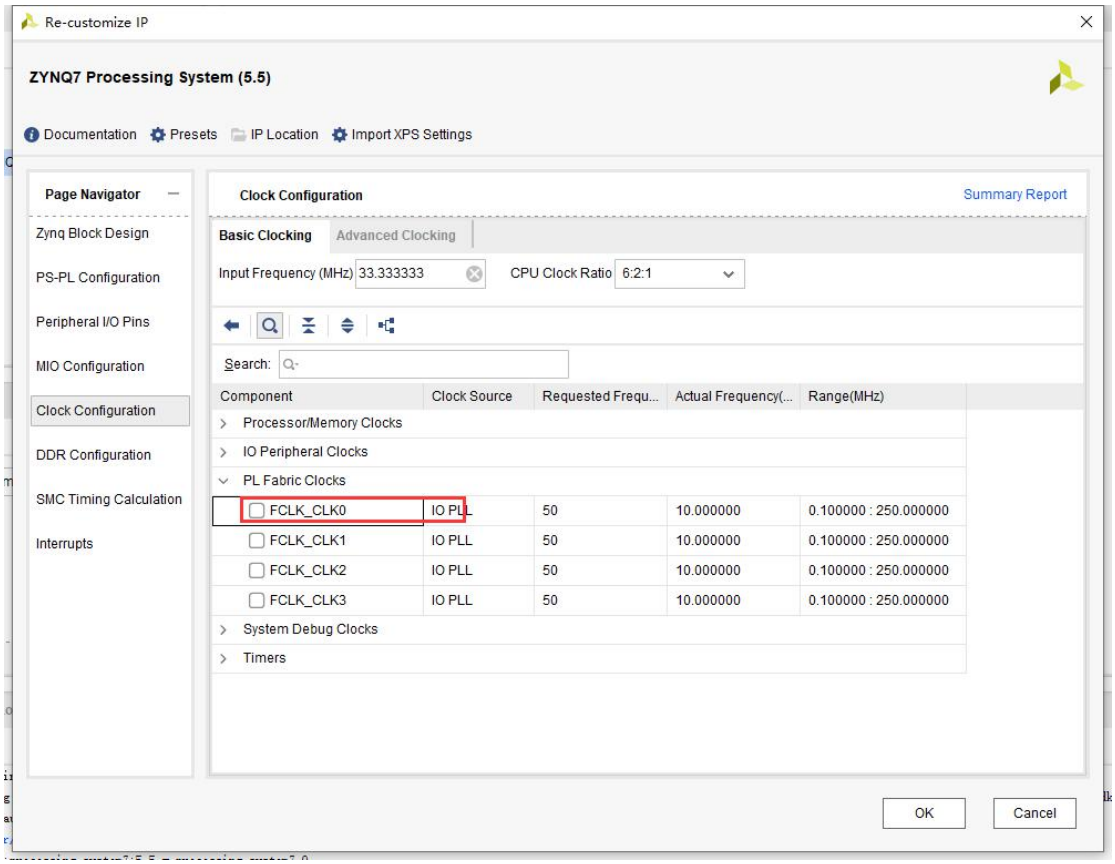


如下图所示：

图 3.2.2 建立“mio_interrupt”工程

建立完成后创建“Block Dseign”，命名为“mio_interrupt”，在其中添加 ZYNQ7 Processing System IP 核，IP 核的配置与之前的实验完全相同，在此不再赘述，请参照之前的实验进行配置，具体配置如下图所示：


Re-customize IP
ZYNQ7 Processing System (5.5)
Documentation
Presets
IP Location
Import XPS Settings
Page Navigator
Zynq Block Design
PS-PL Configuration
Peripheral I/O Pins
MIO Configuration
Clock Configuration
DDR Configuration
SMC Timing Calculation
Interrupts
PS-PL Configuration
Summary Report
Search:
Name
Select
Description
Address Editor
Enable Clock Triggers
Enable Clock Resets
FCLK_RESET0_N
Enables general purpose reset signal 0 for PL logic
FCLK_RESET1_N
Enables general purpose reset signal 1 for PL logic
FCLK_RESET2_N
Enables general purpose reset signal 2 for PL logic
FCLK_RESET3_N
Enables general purpose reset signal 3 for PL logic
AXI Non Secure Enablement
0
Enable AXI Non Secure Transaction
GP Master AXI Interface
M AXI GP0 interface
Enables General purpose AXI master interface 0
M AXI GP1 interface
Enables General purpose AXI master interface 1
GP Slave AXI Interface
HP Slave AXI Interface
ACP Slave AXI Interface
DMA Controller
PS-PL Cross Trigger interface
Enables PL cross trigger signals to PS and vice-versa
OK
Cancel


Re-customize IP
ZYNQ7 Processing System (5.5)
Documentation
Presets
IP Location
Import XPS Settings
Page Navigator
Zynq Block Design
PS-PL Configuration
Peripheral I/O Pins
MIO Configuration
Clock Configuration
DDR Configuration
SMC Timing Calculation
Interrupts
Clock Configuration
Summary Report
Basic Clocking
Advanced Clocking
Input Frequency (MHz)
33.333333
CPU Clock Ratio
6:2:1
Search:
Component
Clock Source
Requested Frequ...
Actual Frequency(...
Range(MHz)
Processor/Memory Clocks
IO Peripheral Clocks
PL Fabric Clocks
FCLK_CLK0
IO PLL
50
10.000000
0.100000 : 250.000000
FCLK_CLK1
IO PLL
50
10.000000
0.100000 : 250.000000
FCLK_CLK2
IO PLL
50
10.000000
0.100000 : 250.000000
FCLK_CLK3
IO PLL
50
10.000000
0.100000 : 250.000000
System Debug Clocks
Timers
OK
Cancel

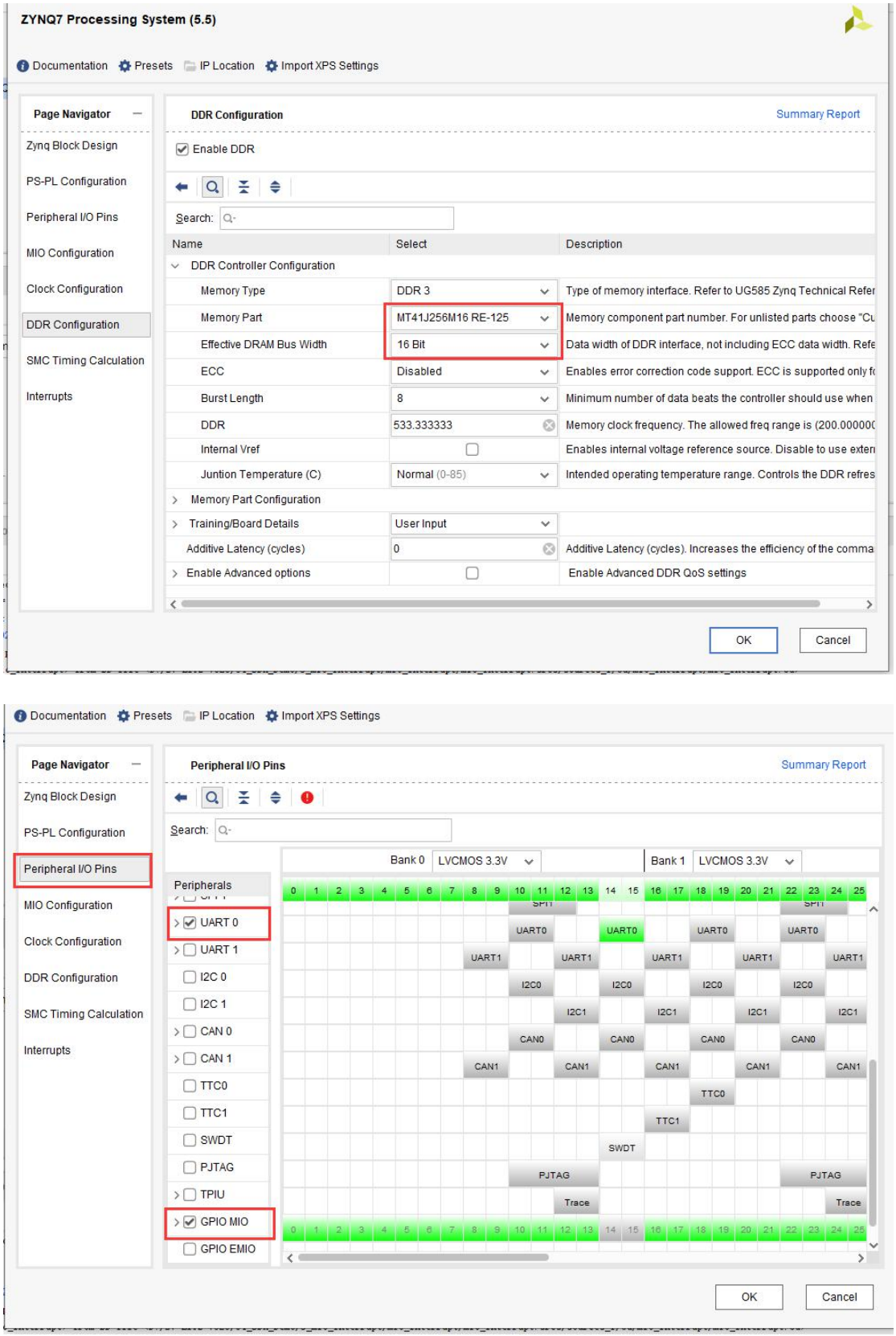
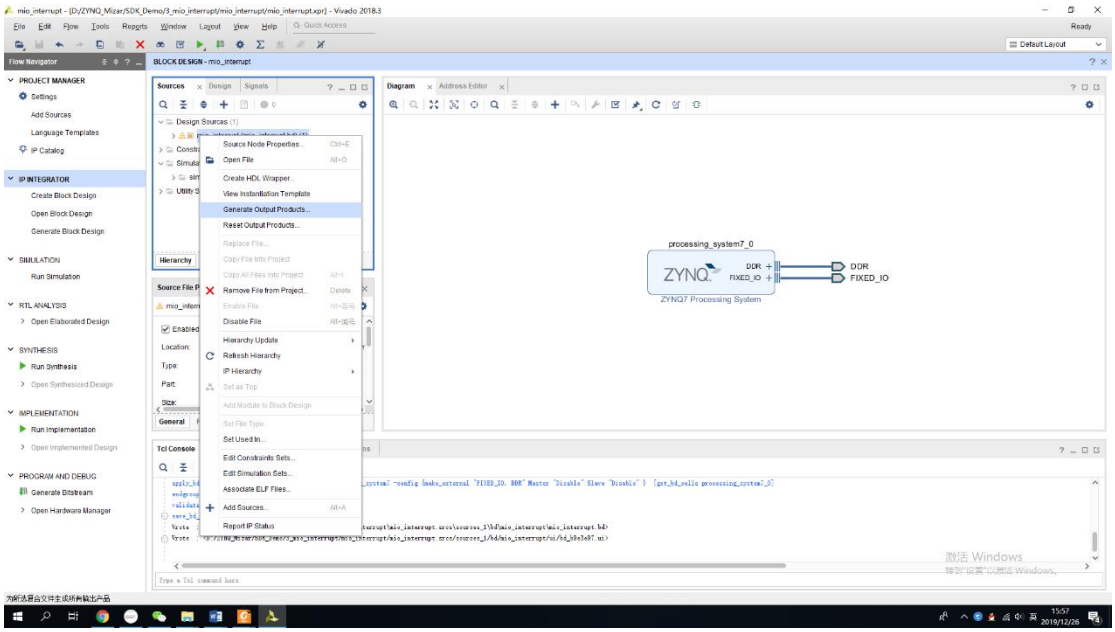
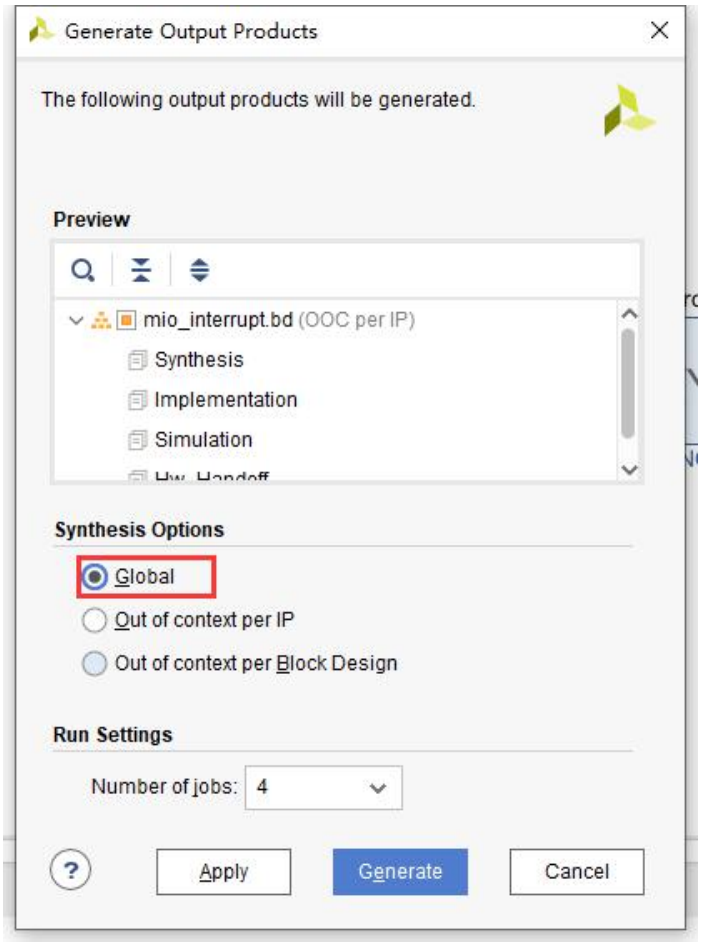


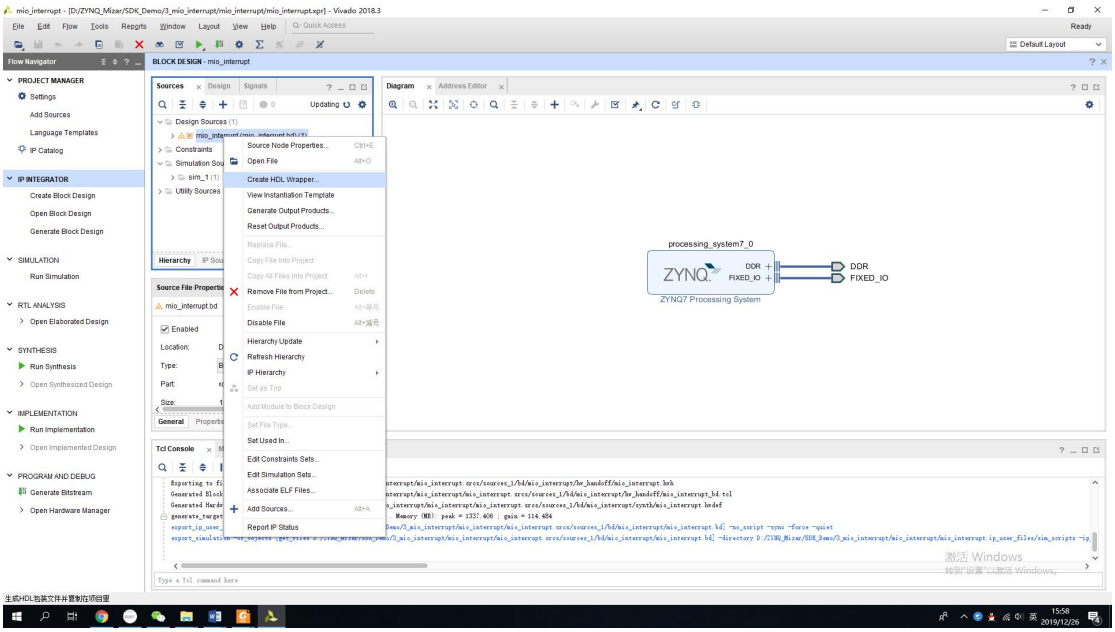
图 3.2.3 IP 核具体配置

配置完成后保存并验证设计，没有错误就可以生成顶层 HDL，导出硬件描述





文件到 SDK，准备进行软件的设计。生成顶层 HDL 的操作如下图所示：



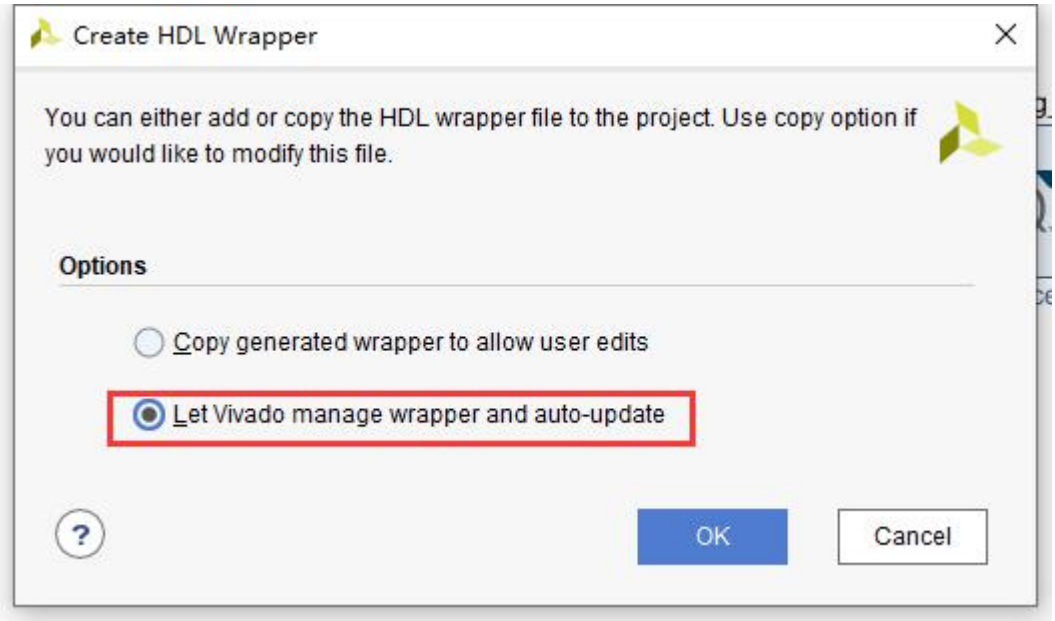
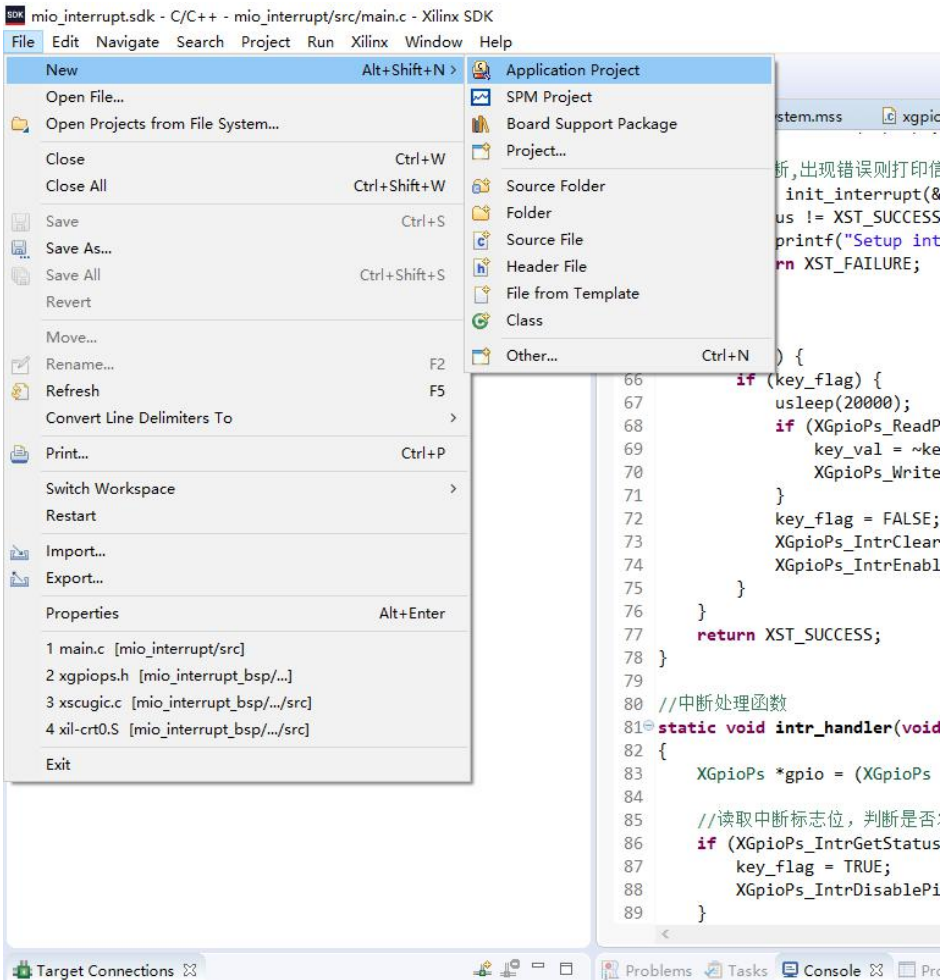


图 3.2.4 生成顶层 HDL 操作

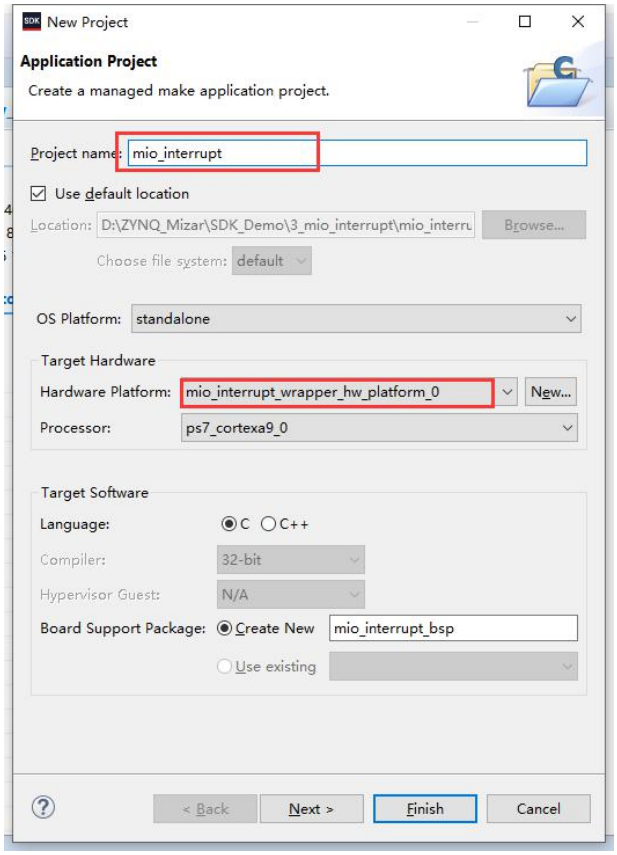
3.3 软件设计

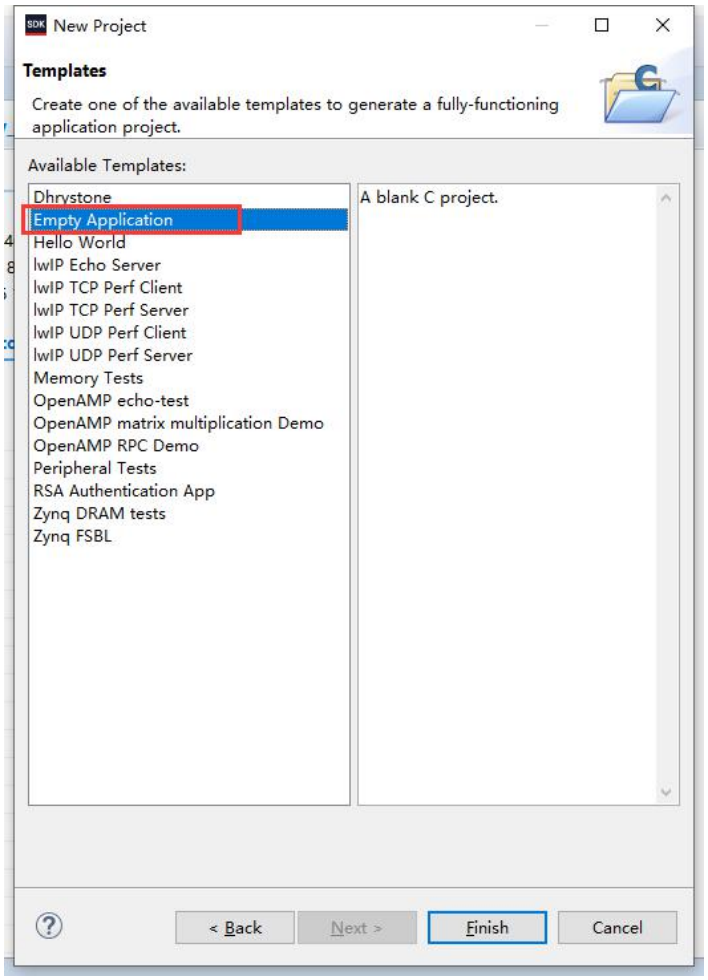
进行软件设计之前我们来讲解一下 ZYNQ 系统中断的配置流程。类似于之前配置 GPIO 的操作，使用该功能之前需要先通过查找 ID 号加载该功能的驱动器并初始化该驱动器，之后配置相关设置并使能。在本实验中我们用到的有 GPIO 功能，通用中断控制功能，所以对于本实验，我们程序的设计流程如下：1、通过查找 GPIO 驱动器 ID 加载驱动器并初始化 GPIO；2、设置 KEY 所连接 MIO 的方向为输入，LED 所连接的 MIO 的方向为输出并使能输出；3、查找并初始化中断控制器；4、设置并使能中断；5、设置中断处理函数；6、设置中断类型为 GPIO 中断，触发方式为下降沿中断，使能中断源；7、处理中断事件，反转 LED 电平。

在了解了设计流程之后，我们来进行软件设计的工作。首先在打开的 SDK

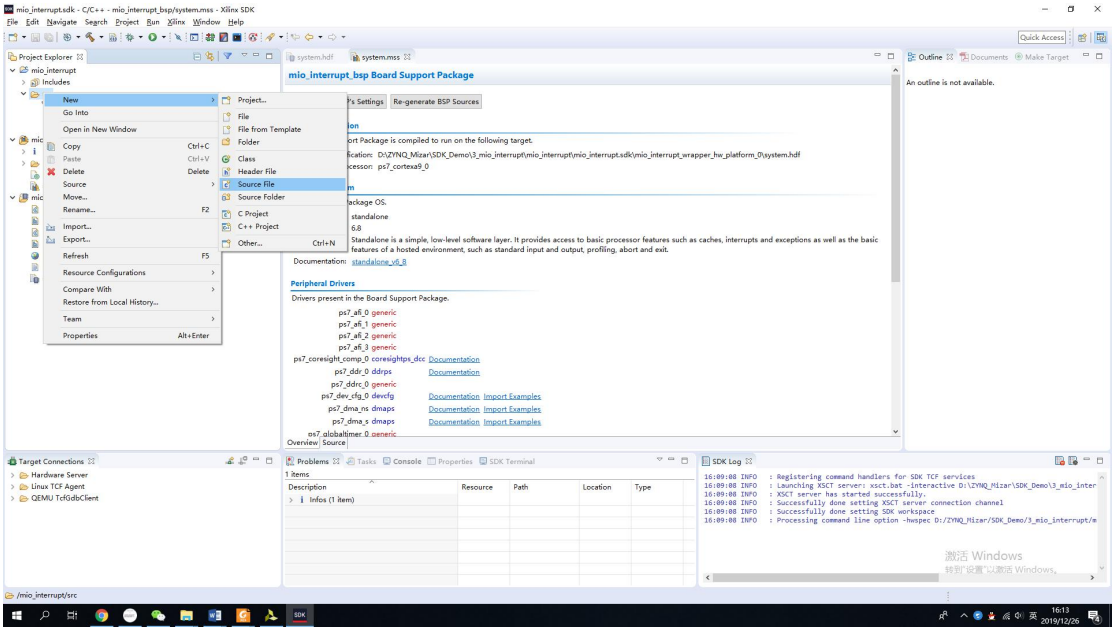


软件内新建一个空白工程命名为“mio_interrupt”，操作步骤如下图所示。


New Project
Application Project
Create a managed make application project.
Project name: mio_interrupt
Use default location
Location: D:\ZYNQ_Mizar\SDK_Demo\3_mio_interrupt\mio_interru
Browse...
Choose file system: default
OS Platform: standalone
Target Hardware
Hardware Platform: mio_interrupt_wrapper_hw_platform_0
New...
Processor: ps7_cortexa9_0
Target Software
Language: C C++
Compiler: 32-bit
Hypervisor Guest: N/A
Board Support Package: Create New mio_interrupt_bsp
Use existing
< Back
Next >
Finish
Cancel

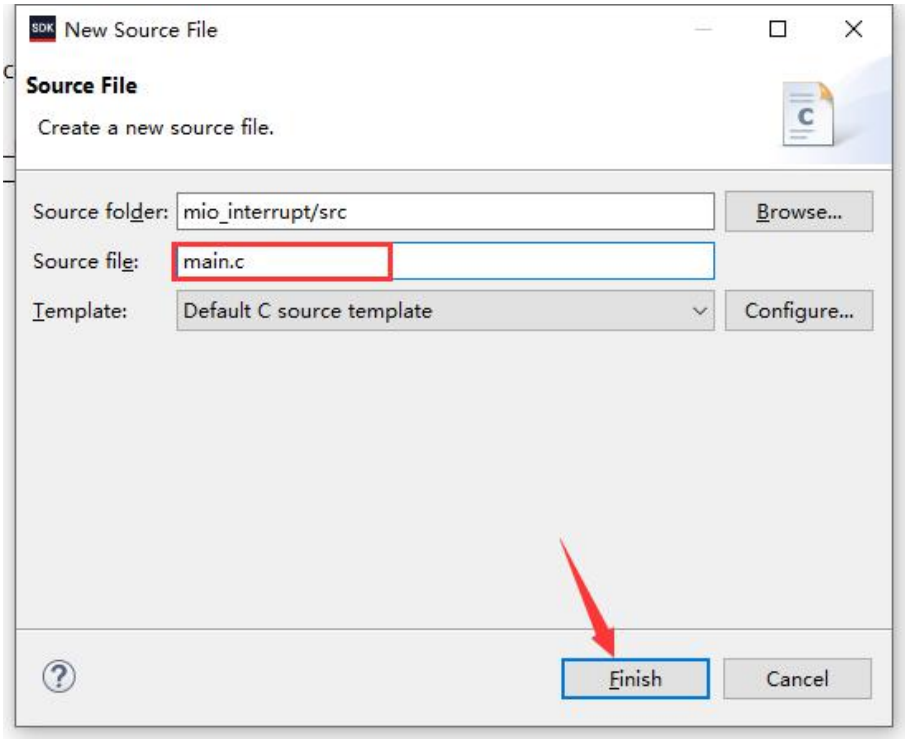

New Project
Templates
Create one of the available templates to generate a fully-functioning application project.
Available Templates:
Dhrystone
Empty Application
Hello World
lwIP Echo Server
lwIP TCP Perf Client
lwIP TCP Perf Server
lwIP UDP Perf Client
lwIP UDP Perf Server
Memory Tests
OpenAMP echo-test
OpenAMP matrix multiplication Demo
OpenAMP RPC Demo
Peripheral Tests
RSA Authentication App
Zynq DRAM tests
Zynq FSBL
A blank C project.
< Back
Next >
Finish
Cancel

因为我们创建的是一个空白的工程，所以工程中只包含板级支持包、库函数核驱动文件等内容而没有主程序，所以我们要手动在程序中添加主程序文件。右键点击 mio_interrupt 工程下面的 src 文件夹，在弹出的选项栏中选择



New->Source File，如下图所示：

图 3.3.1 添加主程序源文件



在弹出的对话框中输入 main.c，然后点击 finish。

图 3.3.2 添加 main.c 文件

```c
#include "xparameters.h"
#include "xgpiops.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xplatform_info.h"
#include <xil_printf.h>
#include "sleep.h"



#define GPIO_DEVICE_ID      XPAR_XGPIOPS_0_DEVICE_ID      //PS端GPIO器件ID
#define INTC_DEVICE_ID      XPAR_SCUGIC_SINGLE_DEVICE_ID  //通用中断控制器ID
#define GPIO_INTERRUPT_ID   XPAR_XGPIOPS_0_INTR           //PS端GPIO中断ID


#define KEY1  9             //KEY1连接到MIO0
#define LED1  0            //LED1 连接到MIO50



static void intr_handler(void *callback_ref);
int init_interrupt(XScuGic *gic_ins_ptr, XGpioPs *gpio, u16 GpioIntrId);



XGpioPs Gpiops;
XScuGic intc;
u32 key_flag;  //按键中断标志
u32 key_val;    //控制LED亮灭
```

```c
int main(void)
{
    int status;
    XGpioPs_Config *ConfigPtr;

    xil_printf("mio interrupt \r\n");


    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    if (ConfigPtr == NULL) {
        return XST_FAILURE;
    }
    //初始化GPIO
    XGpioPs_CfgInitialize(&Gpiops, ConfigPtr, ConfigPtr->BaseAddr);

    //设置KEY所连接的MIO引脚的方向为输入
    XGpioPs_SetDirectionPin(&Gpiops, KEY1, 0);

    //设置LED所连接的MIO引脚的方向为输出并使能输出
    XGpioPs_SetDirectionPin(&Gpiops, LED1, 1);
    XGpioPs_SetOutputEnablePin(&Gpiops, LED1, 1);
    XGpioPs_WritePin(&Gpiops, LED1, 1);

    //建立中断,出现错误则打印信息并退出
    status = init_interrupt(&intc, &Gpiops, GPIO_INTERRUPT_ID);
    if (status != XST_SUCCESS) {
        xil_printf("Setup interrupt system failed\r\n");
        return XST_FAILURE;
    }
```

```c


    while (1) {
        if (key_flag) {                                  //触发中断
            usleep(20000);                                  //延时消抖
            if (XGpioPs_ReadPin(&Gpiops, KEY1) == 0) {      //检测按键按下
                key_val = ~key_val;
                XGpioPs_WritePin(&Gpiops, LED1, key_val);   //反转LED电平
            }
            key_flag = FALSE;
            XGpioPs_IntrClearPin(&Gpiops, KEY1);       //清除中断标志
            XGpioPs_IntrEnablePin(&Gpiops, KEY1);      //使能中断
        }
    }
    return XST_SUCCESS;
}
```

添加完成后回自动打开该文件的文本编辑框，我们 main.c 中输入以下代码：

```c
//中断处理函数
static void intr_handler(void *callback_ref)
{
    XGpioPs *gpio = (XGpioPs *) callback_ref;

    //读取中断标志位，判断是否发生中断
    if (XGpioPs_IntrGetStatusPin(gpio, KEY1)){
        key_flag = TRUE;
        XGpioPs_IntrDisablePin(gpio, KEY1);          //屏蔽按键KEY中断
    }
}

//初始化中断系统，使能KEY1按键的下降沿中断
int init_interrupt(XScuGic *gic_ins_ptr, XGpioPs *gpio, u16 GpioIntrId)
{
    int status;
    XScuGic_Config *IntcConfig;

    //初始化中断控制器
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (NULL == IntcConfig) {
        return XST_FAILURE;
    }

    status = XScuGic_CfgInitialize(gic_ins_ptr, IntcConfig,
            IntcConfig->CpuBaseAddress);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    //设置并使能中断
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler) XScuGic_InterruptHandler, gic_ins_ptr);
    Xil_ExceptionEnable();

    //设置中断处理函数
    status = XScuGic_Connect(gic_ins_ptr, GpioIntrId,
            (Xil_ExceptionHandler) intr_handler, (void *) gpio);
    if (status != XST_SUCCESS) {
        return status;

    }

    //使能中断类型为来自GPIO的中断
    XScuGic_Enable(gic_ins_ptr, GpioIntrId);

    //设置中断触发方式为下降沿中断
    XGpioPs_SetIntrTypePin(gpio, KEY1, XGPIOPS_IRQ_TYPE_EDGE_FALLING);

    //使能中断源
    XGpioPs_IntrEnablePin(gpio, KEY1);

    return XST_SUCCESS;
}
```

输入完成后按快捷键 Ctrl + S 保存 main.c 文件，软件会自动进行编译，编译过程可以编辑框下面的控制台查看，同时如果软件有错误的话也会在里面显示出来。对于部分语法错误，双击该错误即可跳转到错误的位置 ，具体情况如图所示：

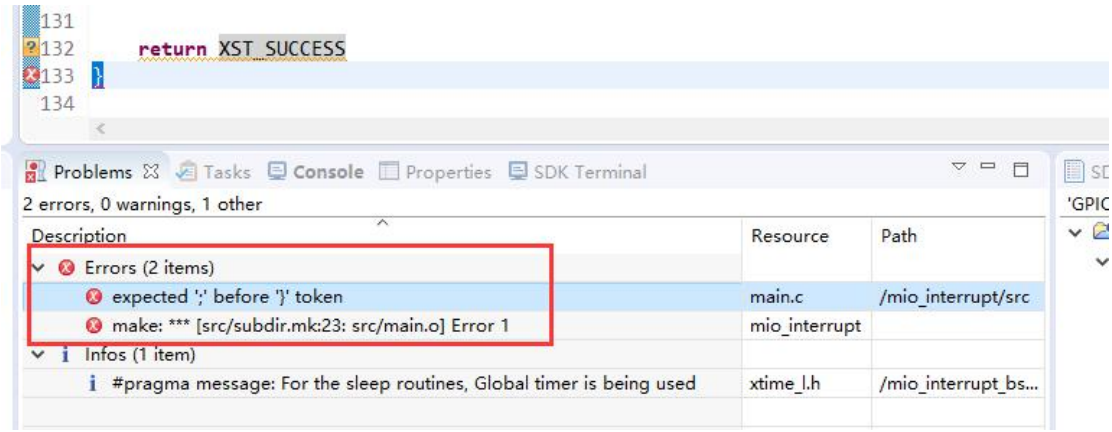


图 3.3.3 错误提示

3.4 下载验证

编译完成后我们就可以将程序烧录到开发板上进行验证，按照之前的方式连接好 JTAG 下载接口和串口，在 Terminal 窗口连接串口，在应用工程 gpio_mio_interrupt 上右击，选择“Run As”，然后选择第一项“1 Launch on Hardware (System Debugger)”烧录程序。烧录完成后可以在 Terminal 窗口看到打印的信息“mio interrupt”，说明程序已经运行。

图 3.4.1 串口打印信息

之后我们反复按下 KEY1 按键即可观察到 LED1 亮灭的变化，说明硬件设计和软件设计都是正确的。对于其他型号的开发板，想要实现相同的功能，只需要在创建 Vivado 工程时选择相应型号的芯片以及在编写程序时在宏定义处修改 KEY 和 LED 连接的 MIO 端口号即可。

3.5 章节小结

通过本实验我们初步了解了 ZYNQ 中断系统时使用方法，但外部输入中断只是整个中断系统的冰山一角，本实验仅起到一个抛砖引玉的作用。整个中断是一个非常庞大且复杂的系统，想要学好中断的使用必须多加练习，仔细阅读官方文档，加深对于中断的理解。

第 4 章　　EMIO 使用实验

MIO 是最常用也是使用最多的应用，但是 ZYNQ 最多只有 54 个 MIO 可以使用，如果 MIO 不够怎么办？这个时候就需要用到 EMIO，EMIO 是对于 MIO 的扩展 I/O，它是将 PL 端的部分 I/O 作为 MIO 供 PS 使用。本实验我们就将学习 EMIO 的使用方法。

本章包括以下几个部分：

4.1 实验简介

4.2 硬件设计

4.3 软件设计

4.4 下载验证

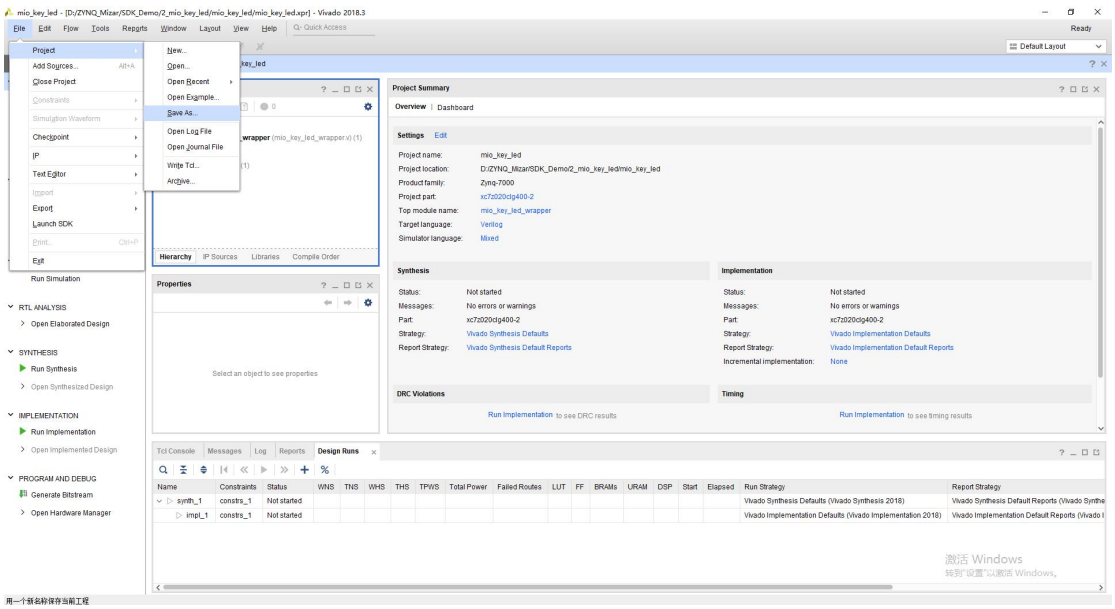
4.5 章节小结

4.1 实验简介

在实际应用中，由于 I/O 数量、电路设计规则等各种限制，我们需要将部分外部设备连接到 PL 端的 I/O 接口上然后使用 PS 来控制这些 I/O，这就是 EMIO 的作用，它不仅扩展了可用 I/O 口的数量，更提高了硬件设计的灵活性，为硬件工程师在设计硬件时提供了更多的选择。同时，使用 PS 来控制 PL 端 I/O 口的输入输出也是 PS 与 PL 端交互最常见的方式。本实验我们就通过使用连接在 PL 端的按键控制 PL 端的 LED 来学习 EMIO 的的的使用方法。

4.2 硬件设计

不同于之前新建工程的方式，我们这次学习一种新的更快捷的方式来进行硬件设计，充分利用之前的工程。我们打开上一个按键控制 LED 实验的 Vivado 工程，点击菜单栏 File->Project->Save As，在弹出的对话框中输入工程名“emio_key_led”并更改工程保存的路径。如下图所示：



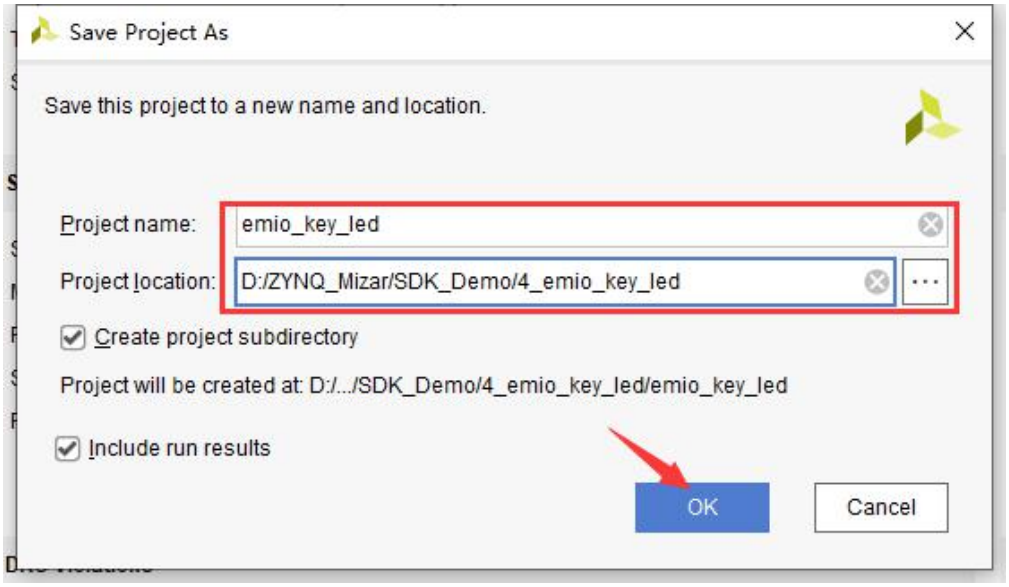


图 4.2.1 另存 mio_key_led 的工程为 emio_key_led

之后 Vivado 会自动关闭当前工程并打开另存的“emio_key_led”工程。我们接下来配置相关设置，点击左侧导航栏 IP INTEGRATOR->Open Block Design 打开 Diagram 界面，双击 ZYNQ7 Processing System 进入配置界面，点击左侧 MIO Configuration，展开右侧 I/O Peripherals->GPIO 的下拉菜单，在其中勾选 EMIO GPIO，并修改位宽为 2，点击 OK 保存设置。如下图所示：

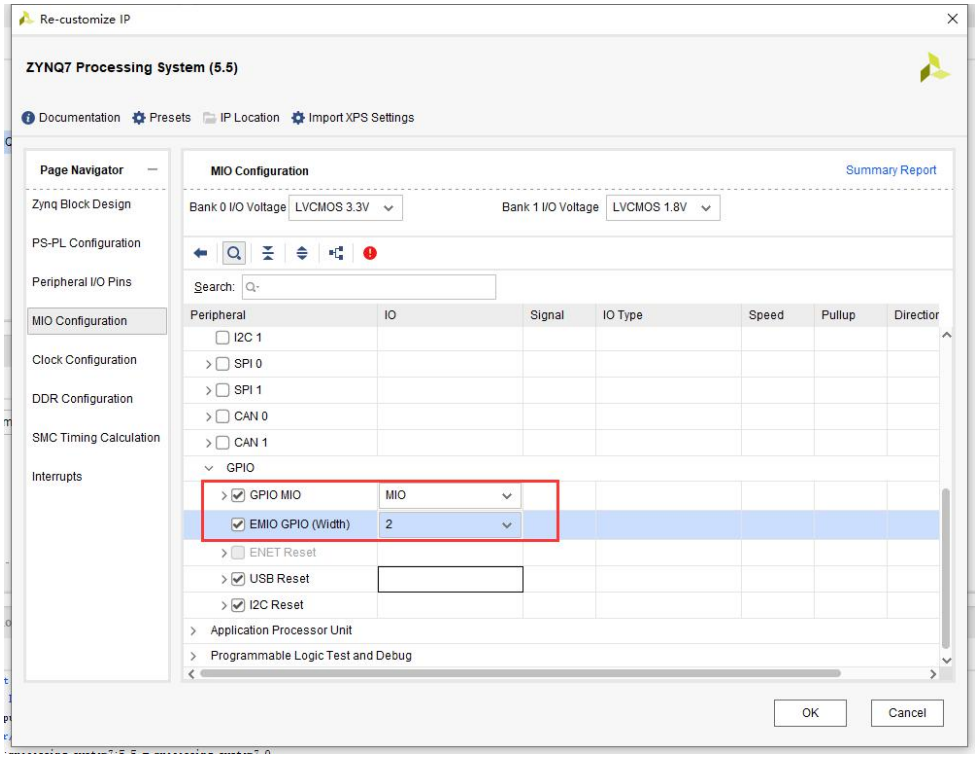
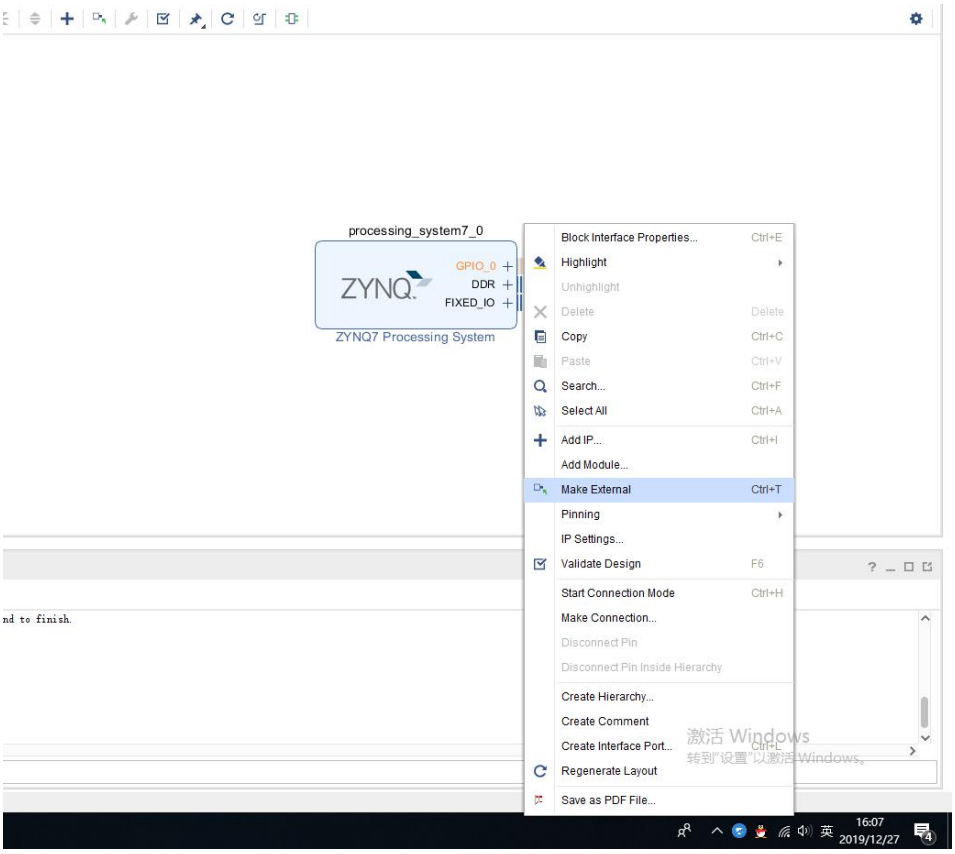


图 4.2.2 使能 EMIO

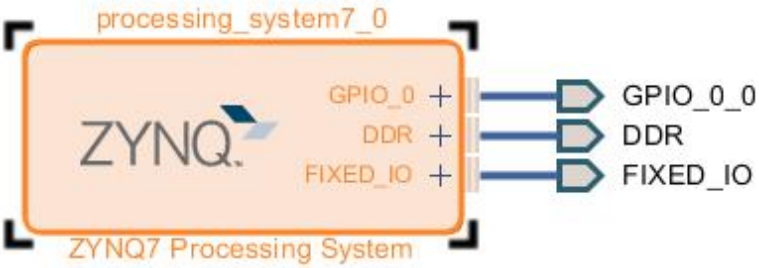
之后返回界面可以看到 IP 核上增加了一组接口，接下来将鼠标箭头移动到接口上，当看到箭头变为铅笔形状是点击鼠标右键，在弹出的菜单中选择 Make



External 将接口修改为外部接口，如下图所示：

图 4.2.3 修改 GPIO 为外部接口

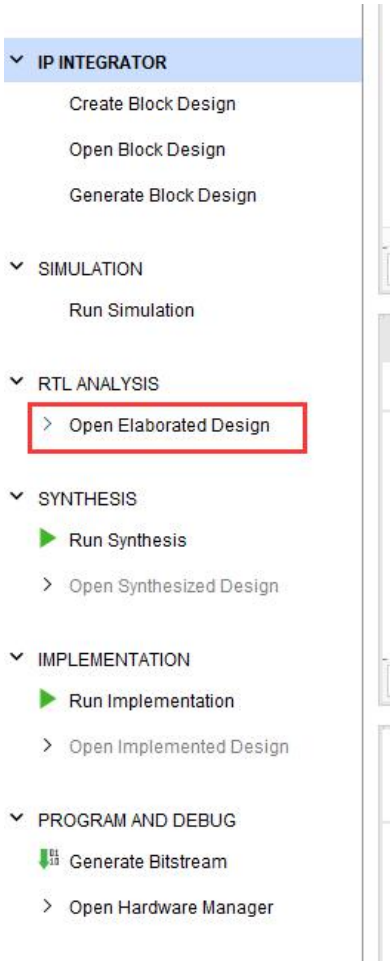
之后可以看到输出接口变为三组，点击接口旁边的“+”可以看到具体由哪

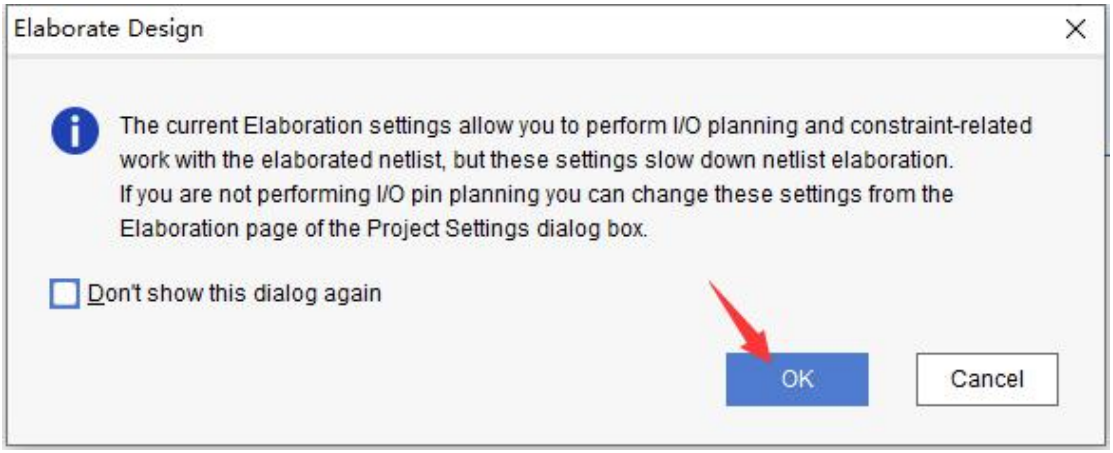


些信号组成，如下图所示：

图 4.2.4 外部信号

之后使用快捷键 Ctrl + S 保存文件，之后按照之前的操作方式生成顶层 HDL 文件。与之前实验在生成 HDL 之后就可以导出到 SDK 不同，本实验中因为我们使用到了 PL 端的资源引脚，所以我们为添加的两个 EMIO 添加引脚约束，确定实际引脚的连接关系。点击左侧导航栏 RTL ANALYSIS->Open Elaborated Design，





在弹出的对话框中点击“OK”打开电气设计窗口。如下图所示：

图 4.2.5 打开电气设计窗口

在打开的窗口内点击箭头所指的 I/O Ports，会在下面打开配置引脚约束的窗口如图 4.2.6 所示。其中 FIXED 和 DDR 接口系统会自动分配，无需手动配置，我们只需配置新添加的两个 I/O，查看原理图可知 PL_KEY1 连接的引脚为 P16，PL_LED1 连接的引脚是 P15,并且所连接端口皆为 3.3V 供电，接口电平配置为 3.3V。具体配置如下图所示：

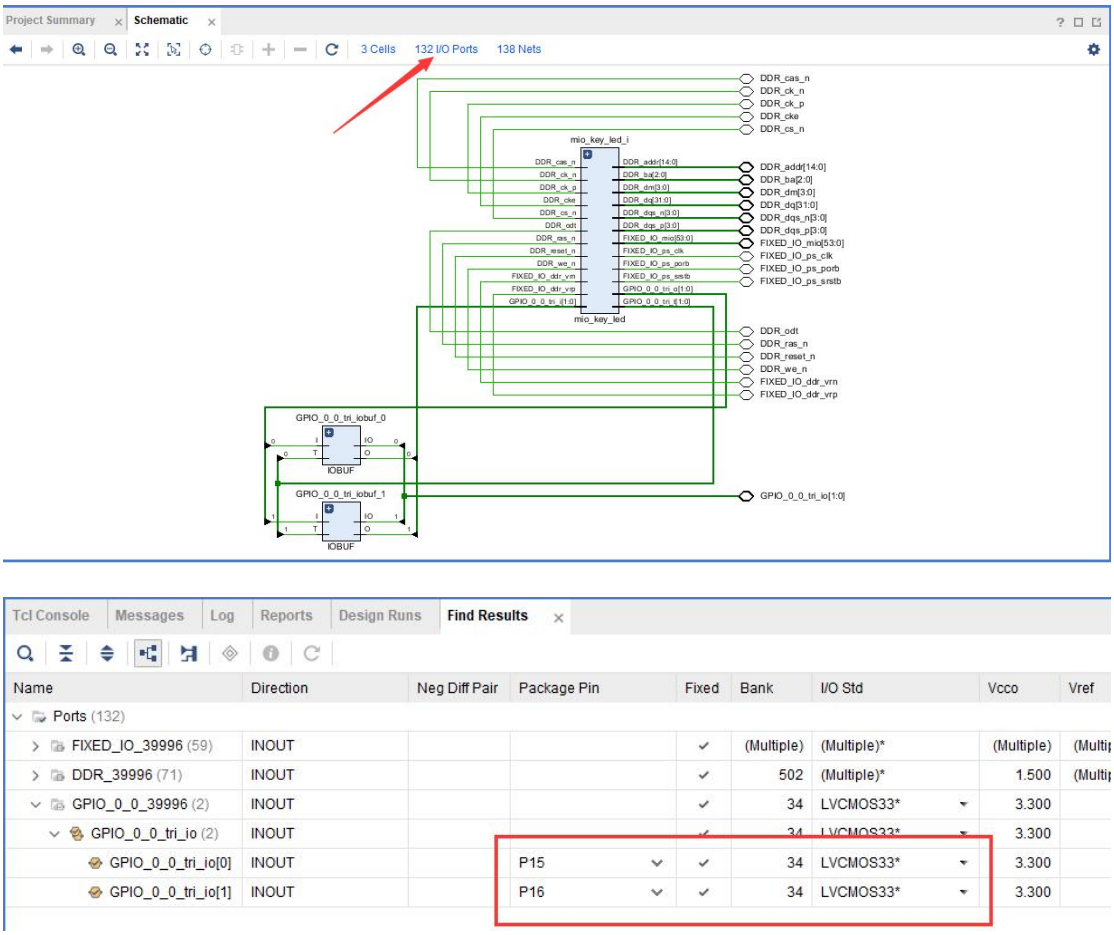
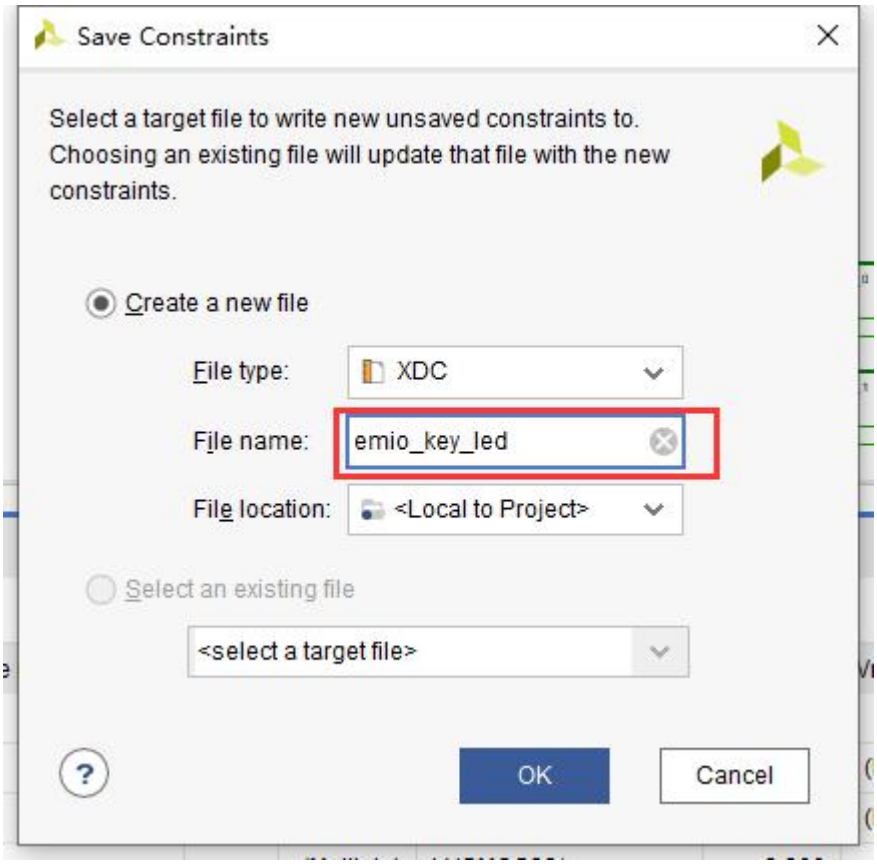


图 4.2.6 I/O 接口配置

其他型号的开发板请查看原理图确定 PL_KEY1 和 PL_LED1 连接的引脚编号。

配置完成后使用快捷键 Ctrl + S 保存配置，在弹出的对话框输入引脚约束文件.xdc 的文件名“emio_key_led”然后点击“OK”。之后在 Source 栏下面

Constraints->constrs->constrs_1 下可以看到刚保存的.xdc 文件，如下图所示：



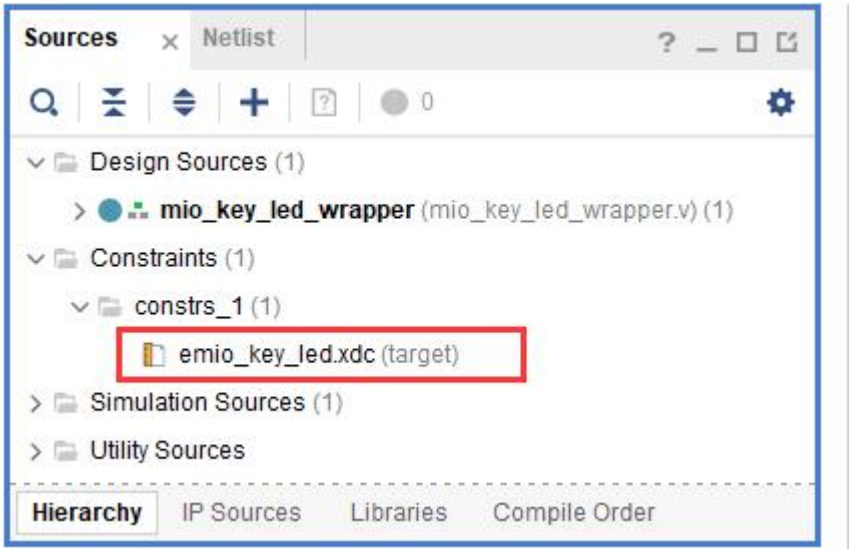


图 4.2.7 保存引脚约束文件.xdc

双击打开.xdc 文件可以看到如下内容，在配置引脚约束时也可以直接在.xdc 文件内通过编写约束规则来实现。

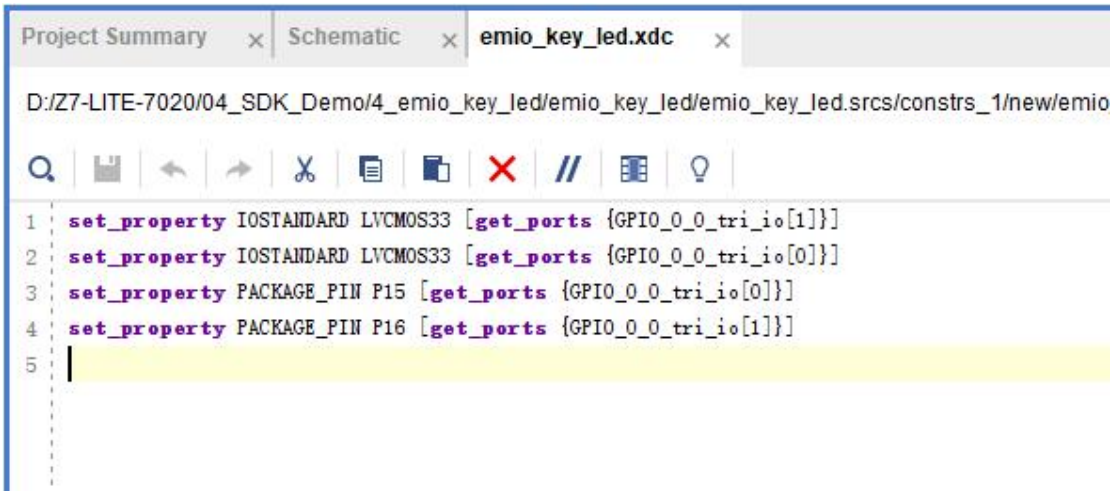
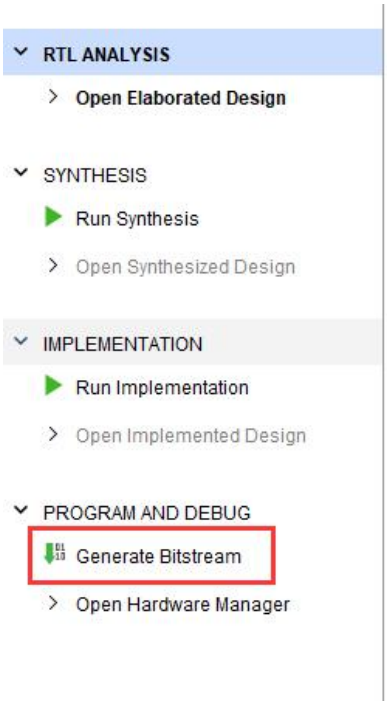
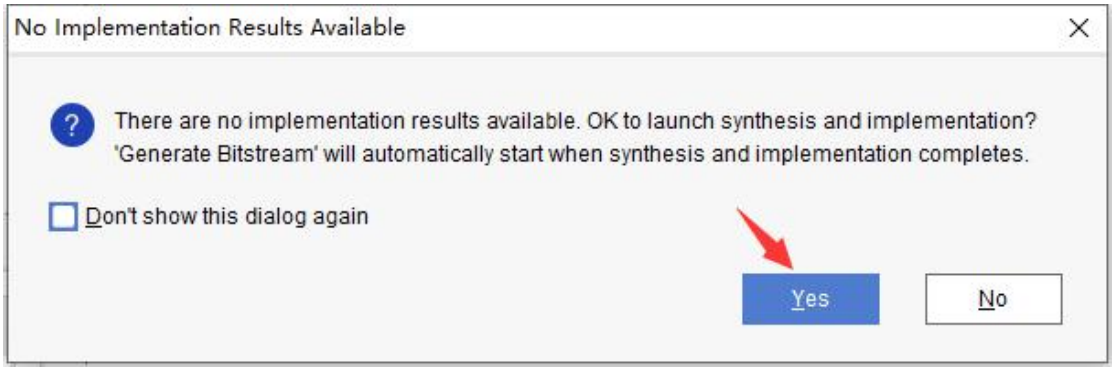


图 4.2.8 .xdc 文件内容

当前我们已经完成了硬件设计工作，接下来就要生成相关的文件为软件设计做准备。对于添加了引脚约束的项目，我们需要生成 Bitstream 文件。点击左侧导航栏内 PROGRAM AND DEBUG->Generate Bitstream，在之后弹出的对话框连续点击 yes，即可开始生成 Bitstream 文件，生成过程需要较长时间，在下方 kongzhi



台可以看到生成的过程。



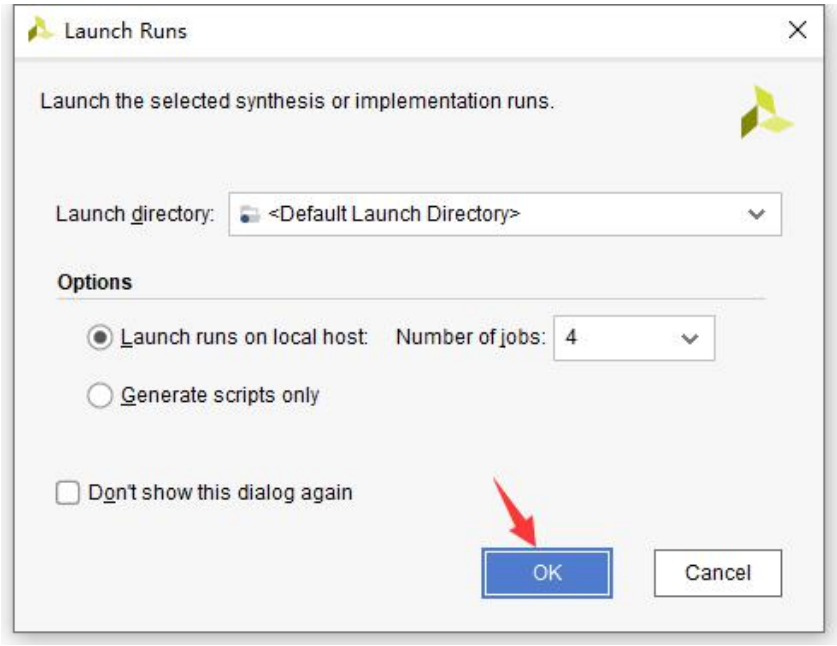
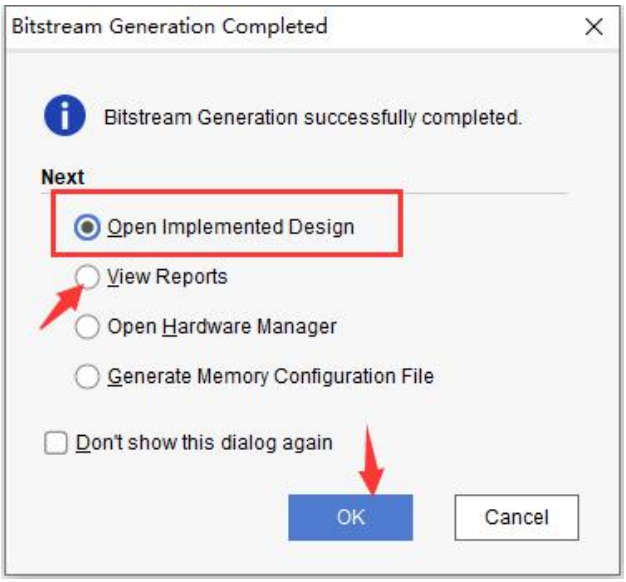


图 4.2.9 生成 Bitstream 文件

生成完成弹出如下对话框，选择 Open Implemented Design，点击 yes。在



弹出的提示框点击 yes。

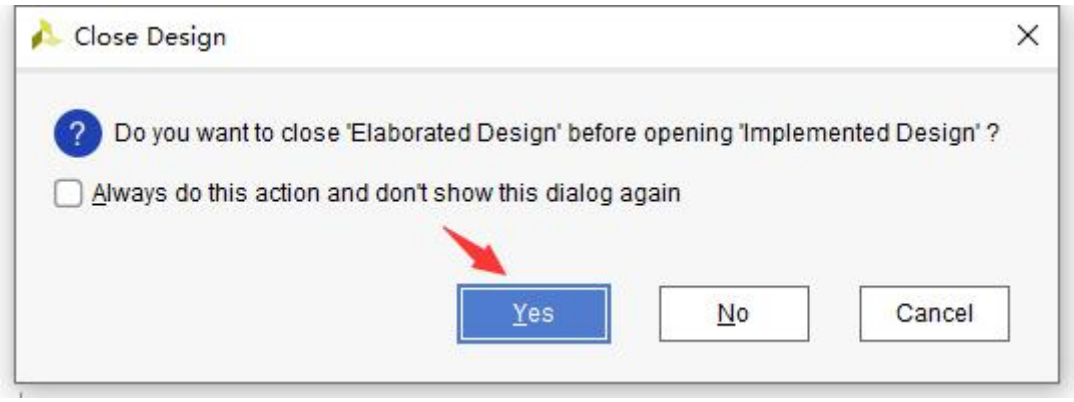


图 4.2.10 打开综合结果

即可打开下图所示页面，在图中即为本次设计实际的版图，高亮部分为用到的硬件电路，在下方的控制台看到 There are no user specified timing constraints 说明没有错误。同时在控制还可以看到本次设计所消耗的系统资源和功耗情况。如下图所示：

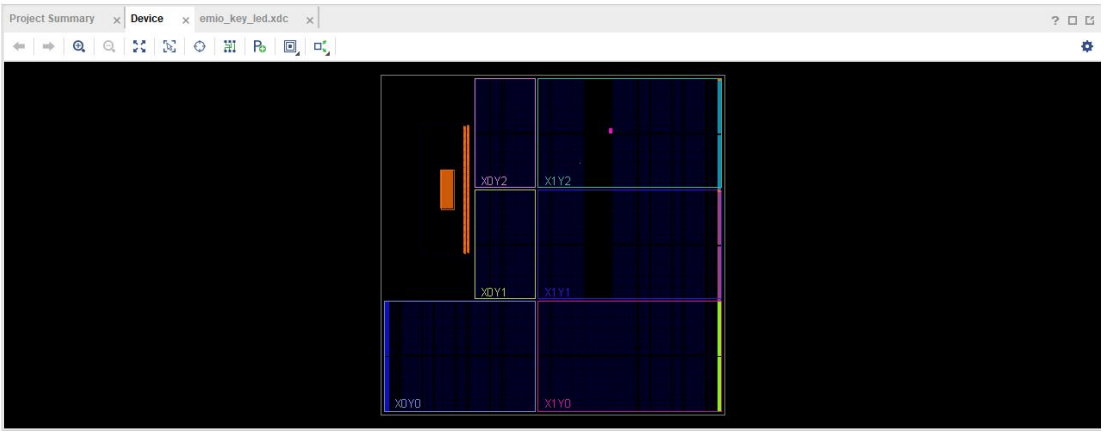
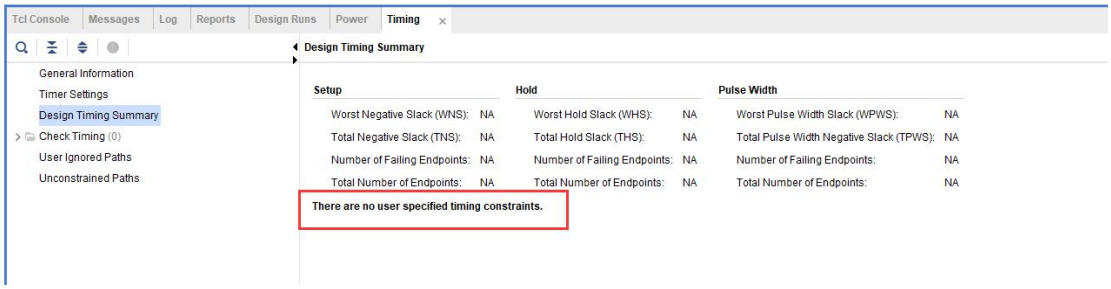


图 4.2.11 设计版图



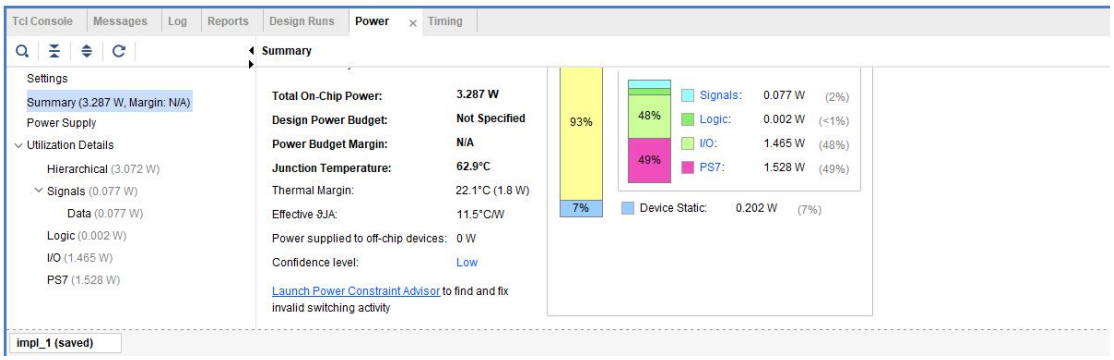
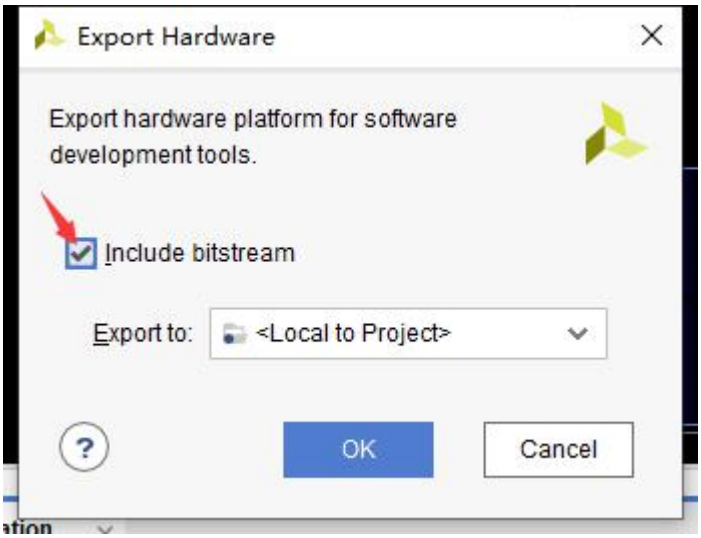


图 4.2.12 资源使用及功耗情况

在综合完成及生成比特流文件（Bitstream）之后，即可导出到 SDK 进行软件设计的工作。在导出之前打开工程文件夹，shanchu 其中的.sdk 文件夹，因为这次实验是通过另存为实验 2 之前实验建立的，所以.sdk 文件夹下是之前的程序，为防止其干扰本次软件的设计所以先删除掉。之后点击菜单栏 File->Export->Export Hardware，因为本次设计生成了 Bitstream 文件，所以要在弹出的对话框



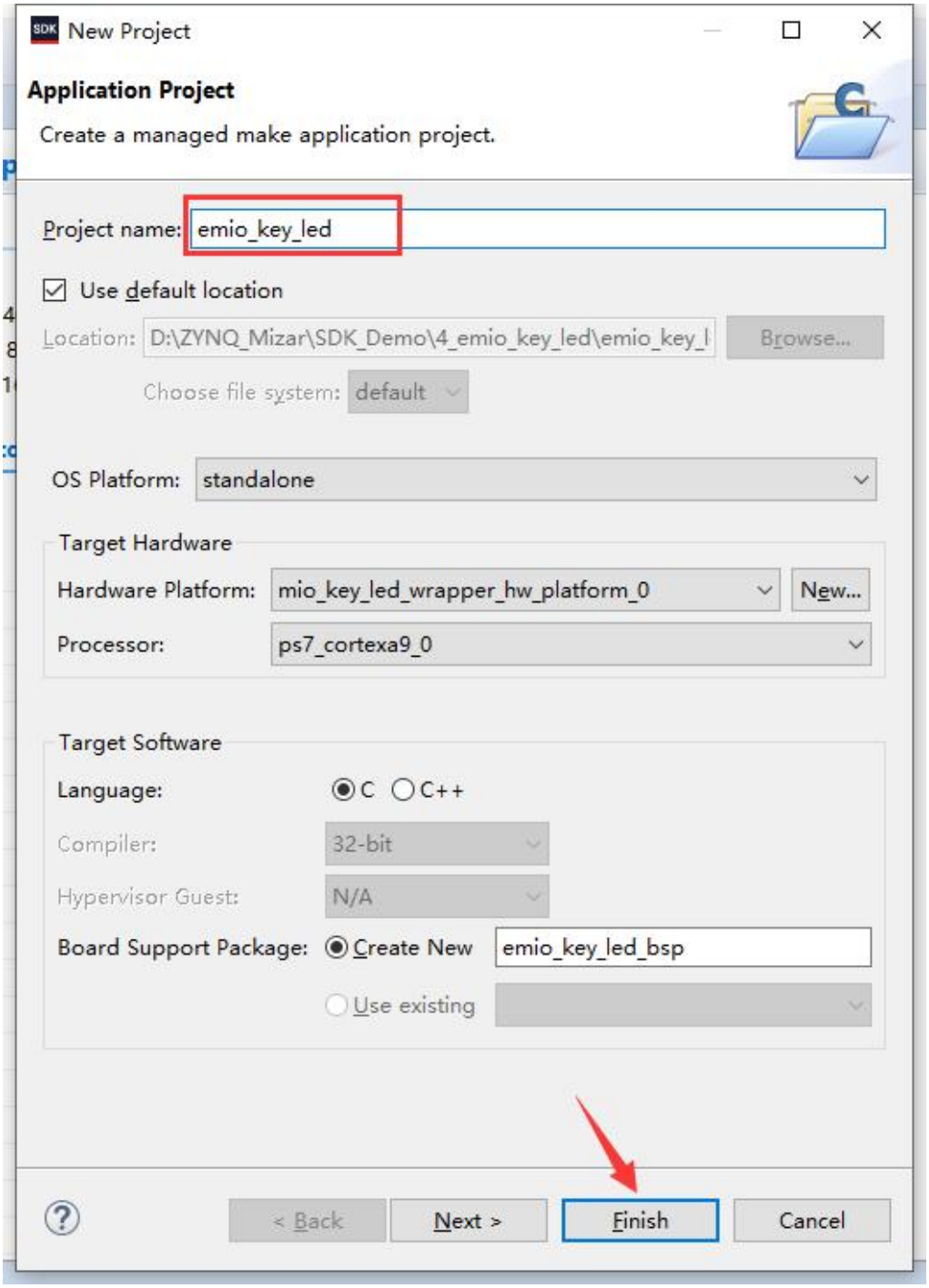
中勾选 Include bitstream，在之后点击 File->Launch SDK 打开 SDK.

图 4.2.13 勾选 Include bitstream

到此为止我们就完成了硬件设计的全部工作。

4.3 软件设计

在打开的 SDK 中我们可以看到导入的硬件平台等文件，之后按照流程建立一个空白的工程，并命名为“emio_key_led”，如下图所示：


New Project
Application Project
Create a managed make application project.
Project name: emio_key_led
Use default location
Location: D:\ZYNQ_Mizar\SDK_Demo\4_emio_key_led\emio_key_l
Browse...
Choose file system: default
OS Platform: standalone
Target Hardware
Hardware Platform: mio_key_led_wrapper_hw_platform_0
New...
Processor: ps7_cortexa9_0
Target Software
Language: C C++
Compiler: 32-bit
Hypervisor Guest: N/A
Board Support Package: Create New emio_key_led_bsp
Use existing
< Back
Next >
Finish
Cancel

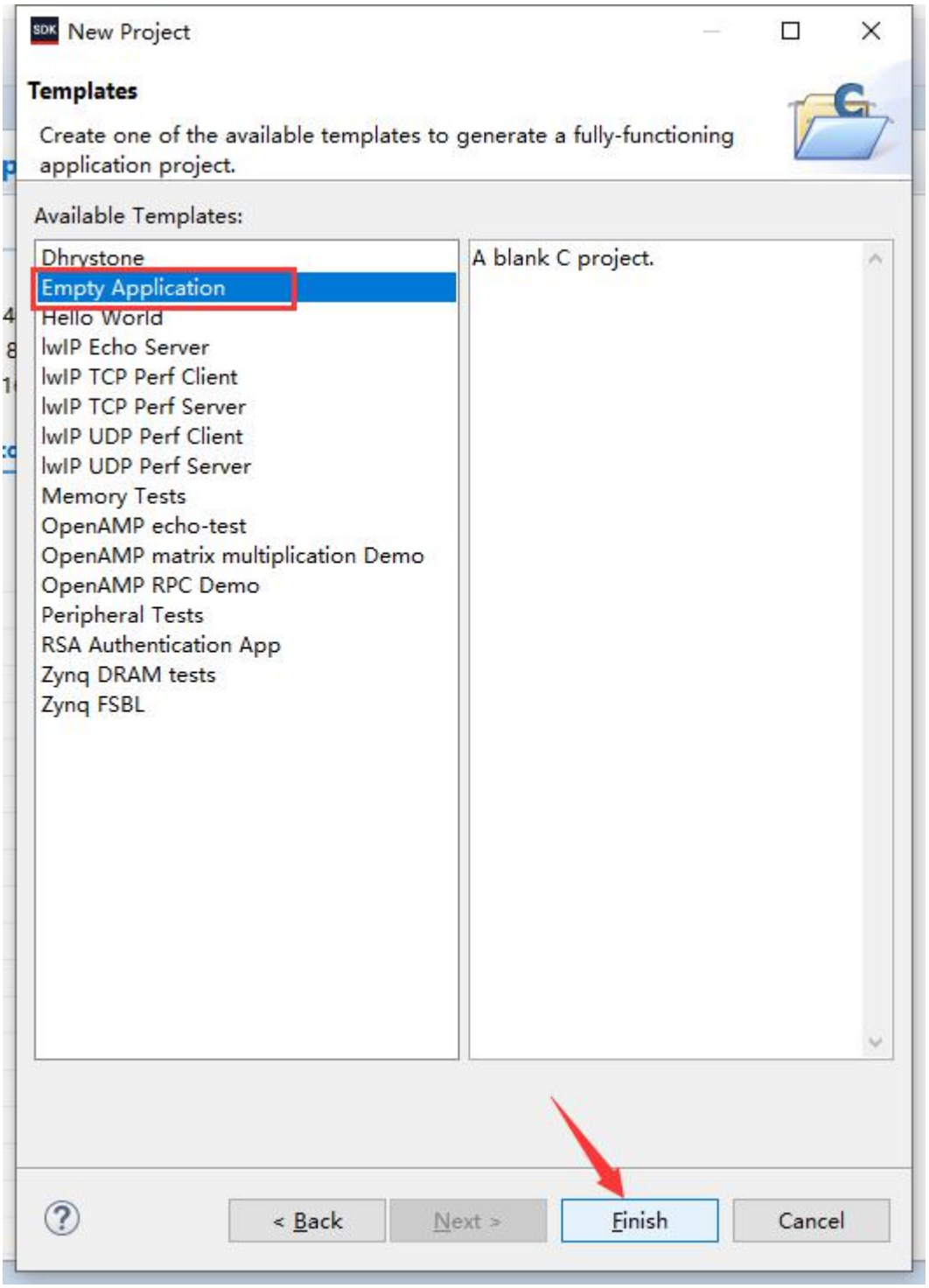
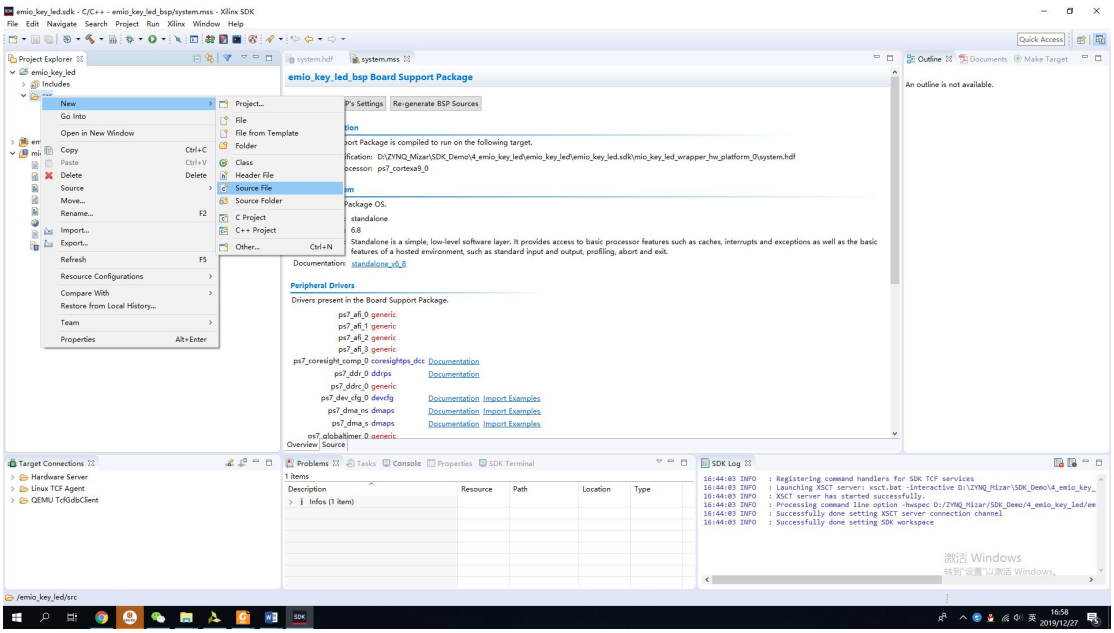


图 4.3.1 建立空白工程

工程建立完成后右键点击 emio_key_led->src，在弹出的选项中点击 New->Source file，在弹出的对话框中输入 main.c，为程序添加主函数源文件。



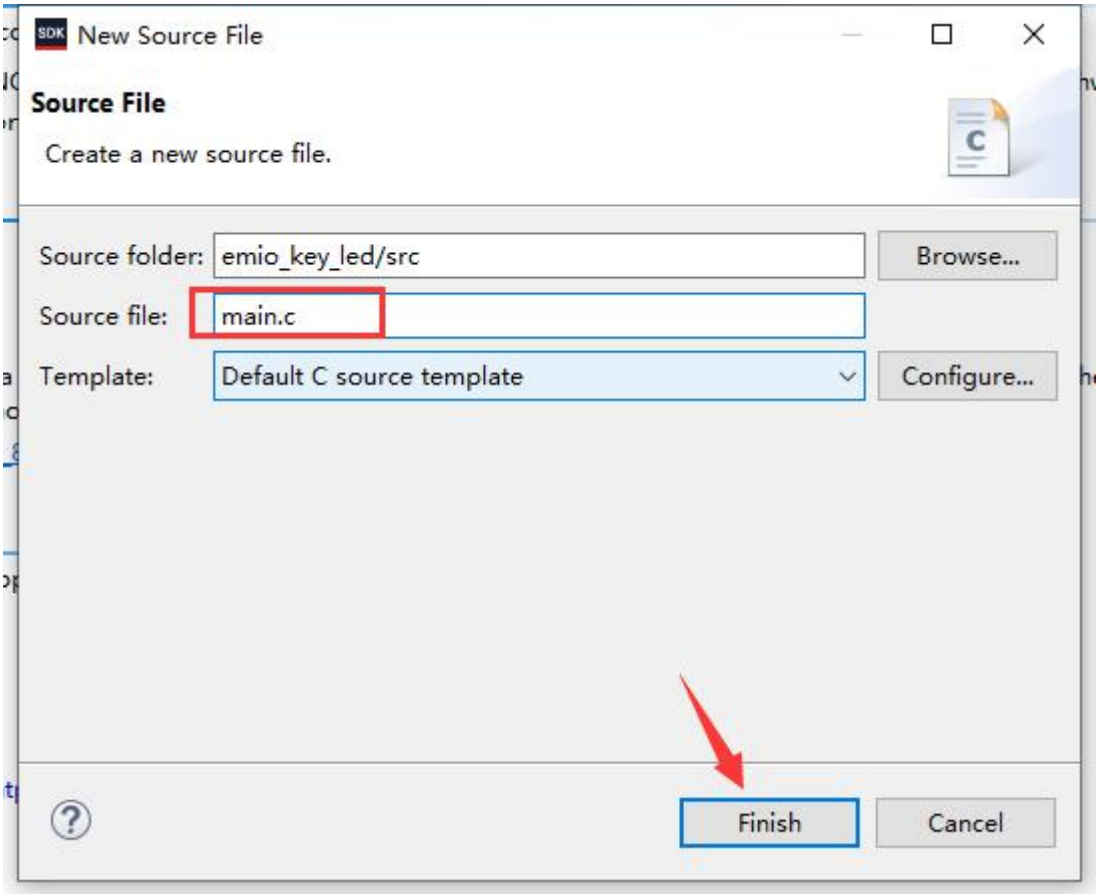


图 4.3.2 添加并命名主函数源文件

添加完成后会自动打开文本编辑窗口，在其中输入以下程序。

```
#include <stdio.h>
#include "xil_printf.h"
#include "xgpiops.h"
#include "xgpiops_hw.h"


#define EMIOLED1 54 //PL_LED1

#define EMIOKEY1 55  //PL_KEY1

#define input  0
#define output 1

XGpioPs Gpios;
```

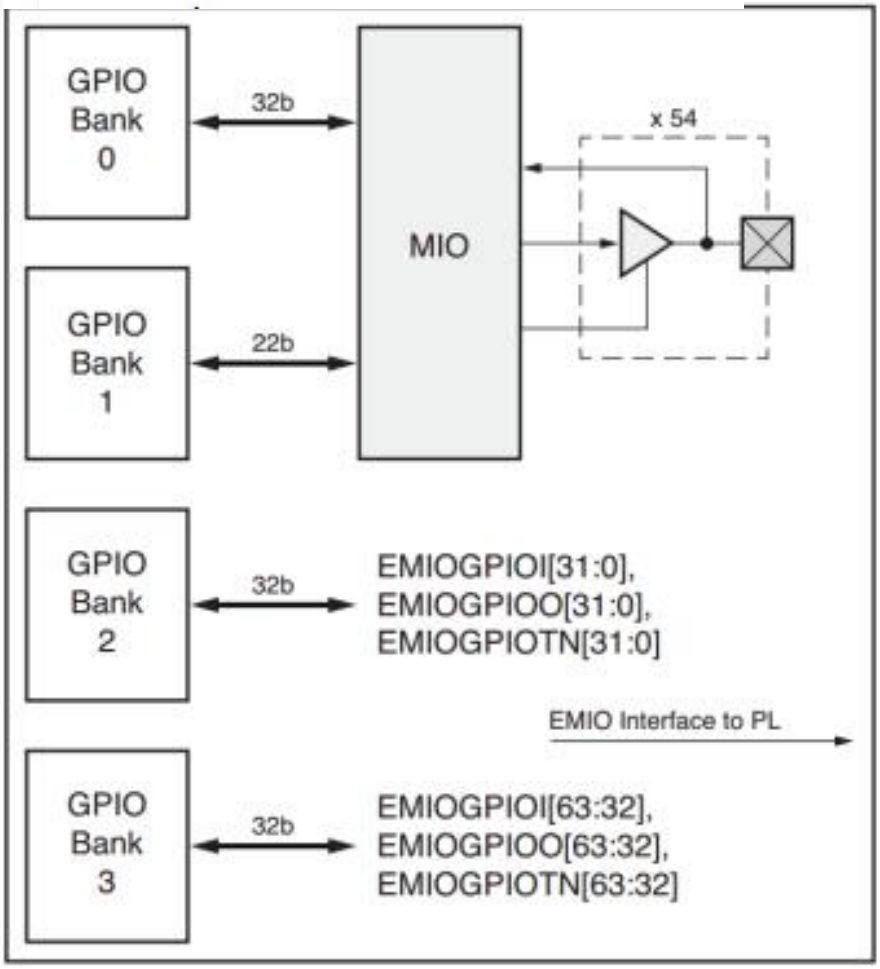
```
int main()
{
    int Status;
     XGpioPs_Config *ConfigPtr;

     //初始化GPIO
     print("EMIO Test! \n\r");
     ConfigPtr = XGpioPs_LookupConfig(XPAR_XGPIOPS_0_DEVICE_ID);
     Status = XGpioPs_CfgInitialize(&Gpios, ConfigPtr,
     ConfigPtr->BaseAddr);
     if (Status != XST_SUCCESS){
     return XST_FAILURE;
     }

     //设置LED接口方向为输出
    XGpioPs_SetDirectionPin(&Gpios,EMIOLED1,output);
    //设置KEY接口方向为输入
    XGpioPs_SetDirectionPin(&Gpios,EMIOKEY1,input);
    //使能LED接口输出
    XGpioPs_SetOutputEnablePin(&Gpios,EMIOLED1,1);
    while(1)
    {
        //读取KEY状态，改变LED输出电平
        if(XGpioPs_ReadPin(&Gpios,EMIOKEY1))
            {XGpioPs_WritePin(&Gpios,EMIOLED1,1);}
        else
            XGpioPs_WritePin(&Gpios,EMIOLED1,0);

    }

    return XST_SUCCESS;
}
```

图 4.3.3 主程序

从程序中可以看到 EMIO 的配置流程和 MIO 的使用是一样的，重点是在如

下的宏定义为什么端口号是 54 55，怎么确定 LED 和 KEY 接的是哪个？

```
#define EMIOLED1 54 //PL_LED1

#define EMIOKEY1 55  //PL_KEY1
```




```
*/
#define XGPIOPS_DEVICE_MAX_PIN_NUM  (u32)118 /**< Max pins in the Zynq GPIO device
                                      * 0 - 31,  Bank 0
                                      * 32 - 53, Bank 1
                                      * 54 - 85, Bank 2
                                      * 86 - 117, Bank 3
                                      */
```

通过之前的讲解我们了解到 ZYNQ 共有 118 个可用 I/O,其中 PS 端有 54 个，PL 端有 64 个，所以 PS 端的编号为 0 - 53，54 - 117 为 PL 端 I/O 编号。在 xgpiops.h 文件内也可以看到每个 Bank 对应的 I/O 口编号。

在本次实验中我们使用了两个 PL 端的 I/O 口，那么对应的端口号就是 54 和 55。那么怎么确定 54 对应的是 LED 还是 KEY 呢？这个取决于在硬件设计引脚

约束时的设置，打开 Vivado 中的 emio_key_led.xdc 文件可以看到如下图所示的引脚约束，我们可以看到 P15 分配了 GPIO_0，P16 分配了 GPIO_1。所以 EMIO54

```
set_property PACKAGE_PIN R19 [get_ports {GPIO_0_0_tri_io[0]}]
set_property PACKAGE_PIN G14 [get_ports {GPIO_0_0_tri_io[1]}]
```

就是 P15,EMIO55 就是 P16。而 P16 与 PL_KEY1 相连，P15 与 PL_LED1 相连，所以 PL_KEY1 对应的端口号为 55，PL_LED1 对应的端口号为 54。

程序编辑完成后使用快捷键 Ctrl + S 保存并编译程序。我们软件设计的工作就完成了。

4.4 下载验证

连接开发板的 JTAG 下载接口和串口，然后按照之前实验的步骤添加 Terminal 窗口并连接串口，右键点击 emio_key_led_bsp，点击 Run As->Run Configurations...，在弹出的窗口双击 xilinx C/C++ application （System Debugger），在右侧勾选 Reset entire system 和 Program FPGA，之后点击 Run 即可烧录程序。

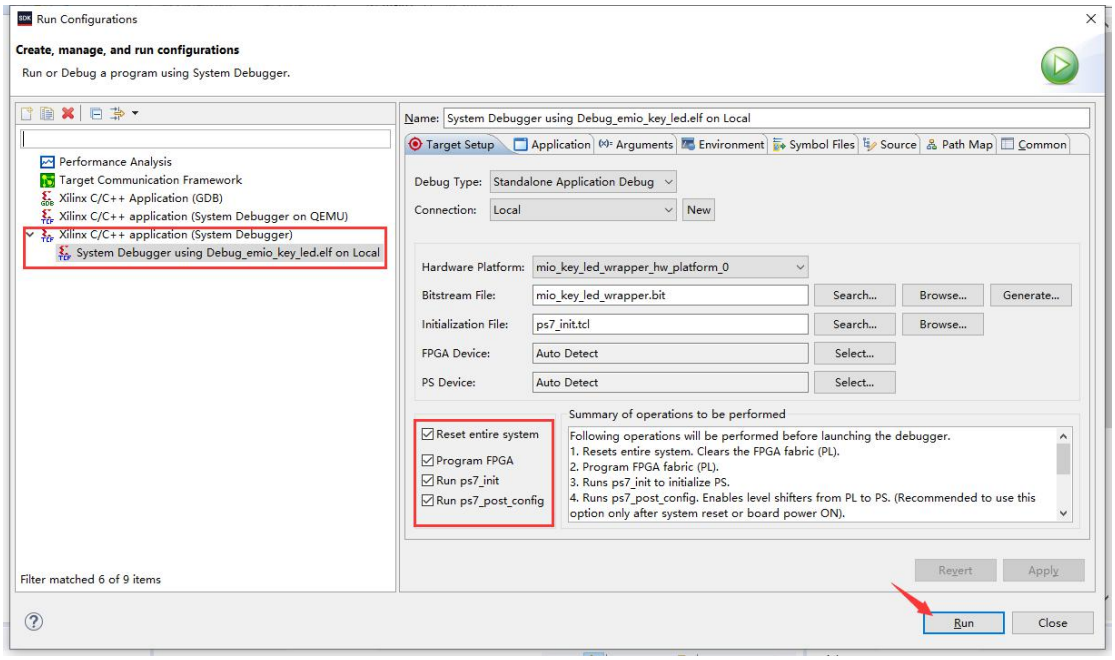


图 4.4.1 烧录程序

烧录完成后再 Terminal 窗口看到打印的“EMIO TEST !”信息，说明程序已经执行，之后按下 PL_KEY1 可以看到 PL_LED 点亮，松开熄灭，说明实验成功。

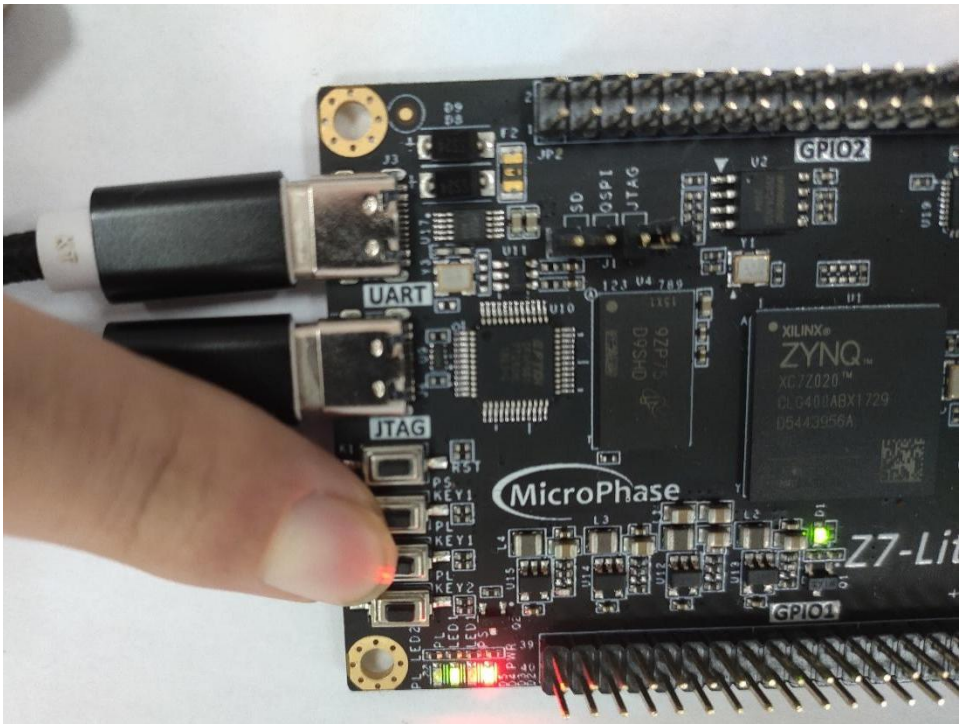


图 4.4.2 实验现象

对于其他型号的开发板在实现本实验时只需修改对应的芯片型号以及查阅原理图分配正确的引脚约束即可，程序方面不做修改，在此不再详细讲解。

4.5 章节小结

在本实验中，我们学习了 ZYNQ EMIO 的使用方法，首次接触到 PS 和 PL 的交互，可以看到在软件设计上和 MIO 基本相同，重点是在硬件设计时使能 EMIO 功能和配置引脚约束，并记住每次修改硬件设计都需要重新生成新的 Bitstream 文件。

第 5 章　　AXI GPIO 中断实验

在上一个实验中，我们首次使用到了 PL 端的资源，但是在实际的应用中 PS 与 PL 交互最主要的方式是通过 AXI 互联接口来实现的，PS 和 PL 分别相当于一个系统中独立的两个部分，AXI 接口负责协调两部分的工作，使他们结合成一个有机的整体，共同构成一个完整的系统。

本实验我们将调用 AXI GPIO IP 核，然后 PS 端通过 AXI4-Lite 接口控制 PL 端的 AXI GPIO 读取按键中断的状态来控制 PS 端的 LED 的亮灭。本章包括以下几个部分：

5.1 实验简介

5.2 硬件设计

5.3 软件设计

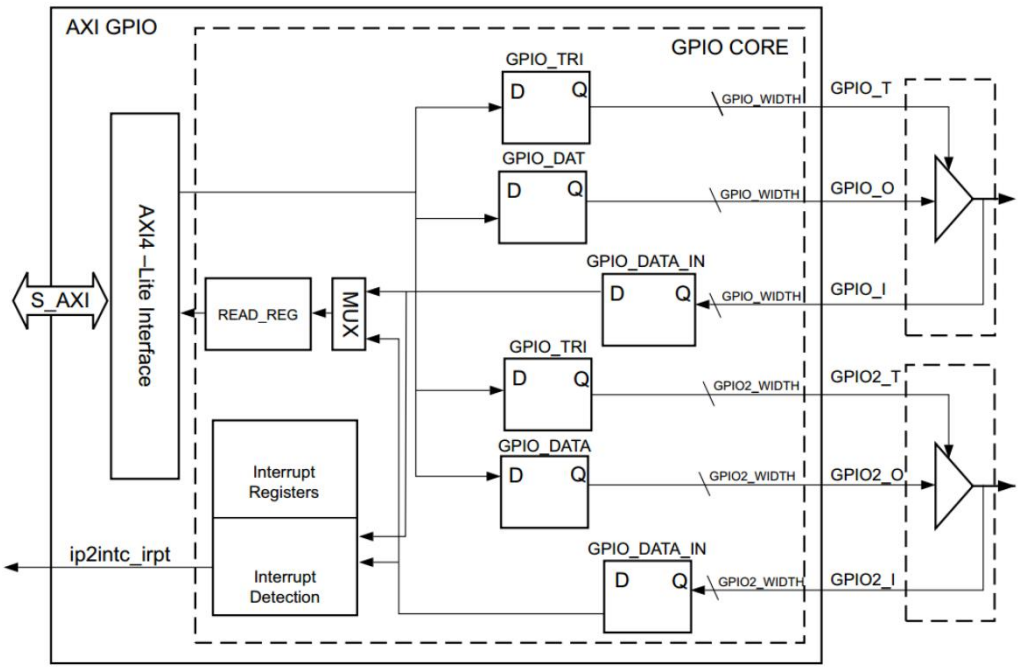
5.4 下载验证

5.5 章节小结

5.1 实验简介

在本实验中，我们将在硬件设计时添加一个 AXI GPIO 的 IP 核。IP 的定义为“用于 ASIC 或 FPGA 中的预先设计好的电路功能模块”。IP 主要分为软 IP、固 IP 和硬 IP。软 IP 是用 Verilog/VHDL 等硬件描述语言描述的功能块，但是并不涉及用什么具体电路元件实现这些功能。固 IP 是完成了综合的功能块。硬 IP 提供设计的最终阶段产品——掩膜。AXI GPIO 就是一个软 IP，它通过使用 PL 端的逻辑资源来实现我们需要的功能，实际上在芯片中并不存在这样一个电路。这与 PS 端的 GPIO 不同，PS 端的 GPIO 是一个实际的电路。

在官方资料中我们可以看到 AXI GPIO 的结构框图如下图所示，从结构框图中我们可以看到模块通过一个 AXI4-Lite 来实现主机对模块的访问，来读取和配置 AXI GPIO 内的各个寄存器，同时，模块还可以产生中断信号来通知主机中断事件的发生。另外可以看到 AXI GPIO 有两个通道，可以配置为单通道或双通道模式。各个 I/O 既可以在软件上配置也可以在硬件设计时被动的配置为输入或输



出。

图 5.1.1 AXI GPIO 结构框图

5.2 硬件设计

双击打开 Vivado 2018.3，按照之前的步骤创建一个新工程并命名为“AXI_GPIO”，在芯片型号选择页面根据自己的开发板型号选择相应的型号，本教程以 xc7z020clg400-2 为例，具体情况如下图所示：

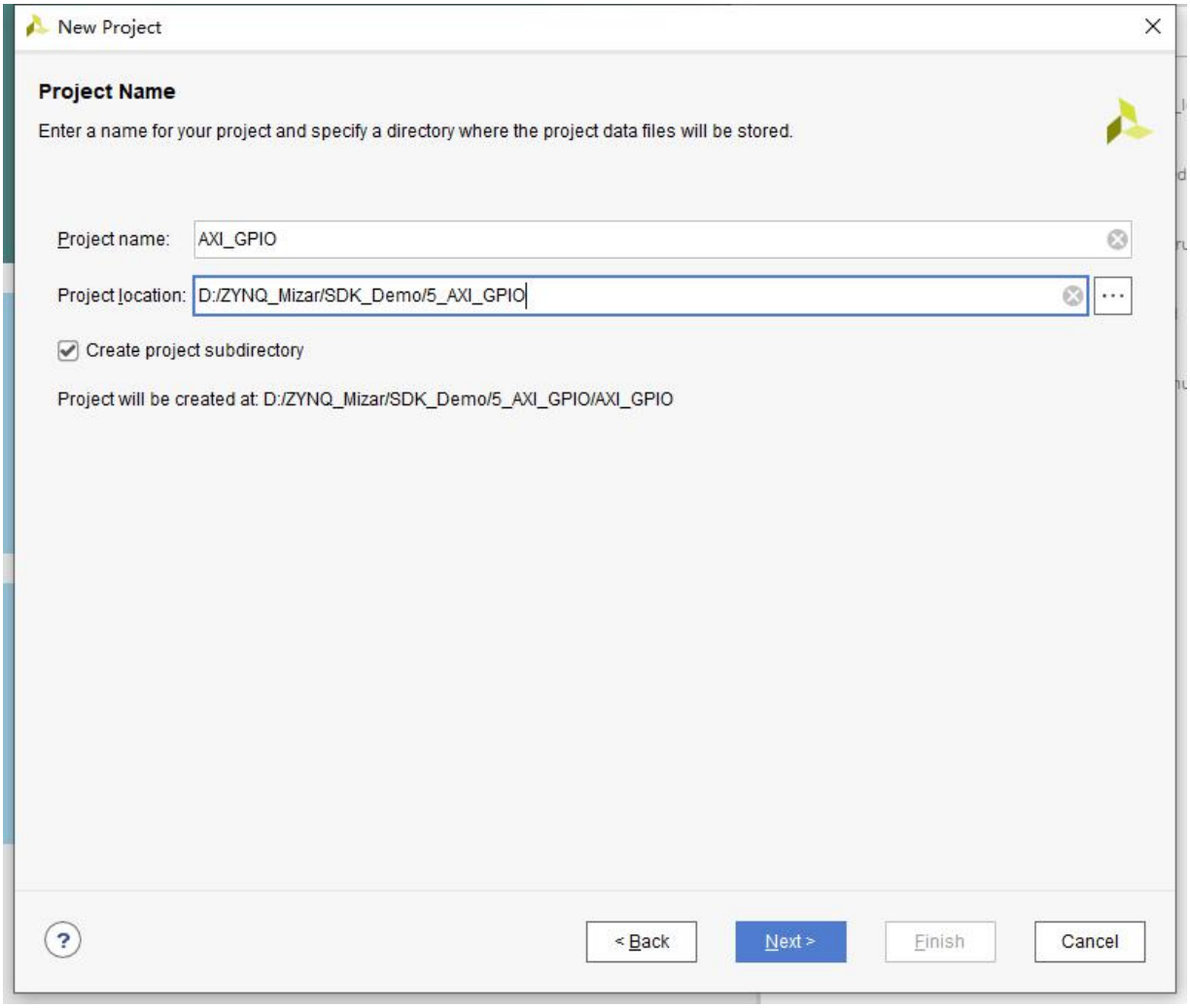


图 5.2.1 创建“AXI_GPIO”工程

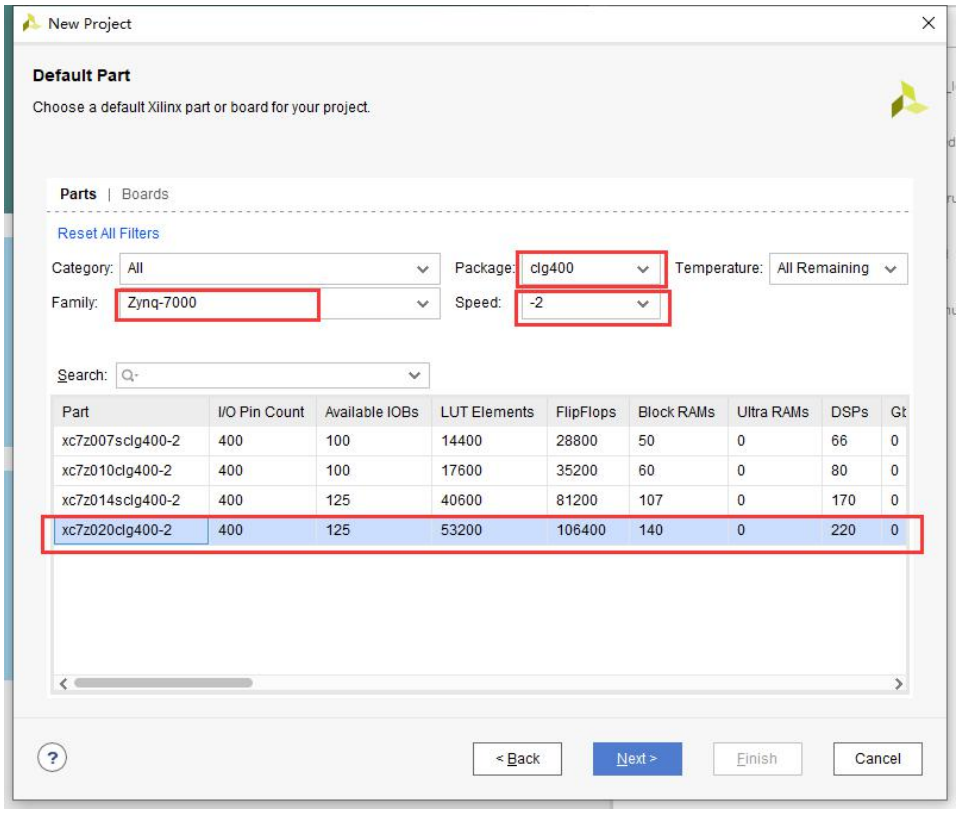


图 5.2.2 选择芯片型号

工程创建完成后点击 Create Block Design 建立新的名为“AXI_GPIO”的设计页面，如下图所示，并添加 ZYNQ7 Processing System IP，双击 IP 进入配置页面。

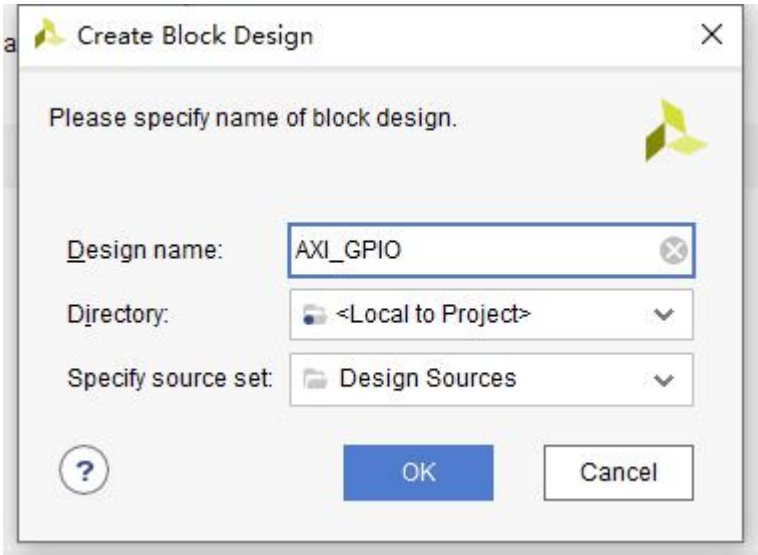
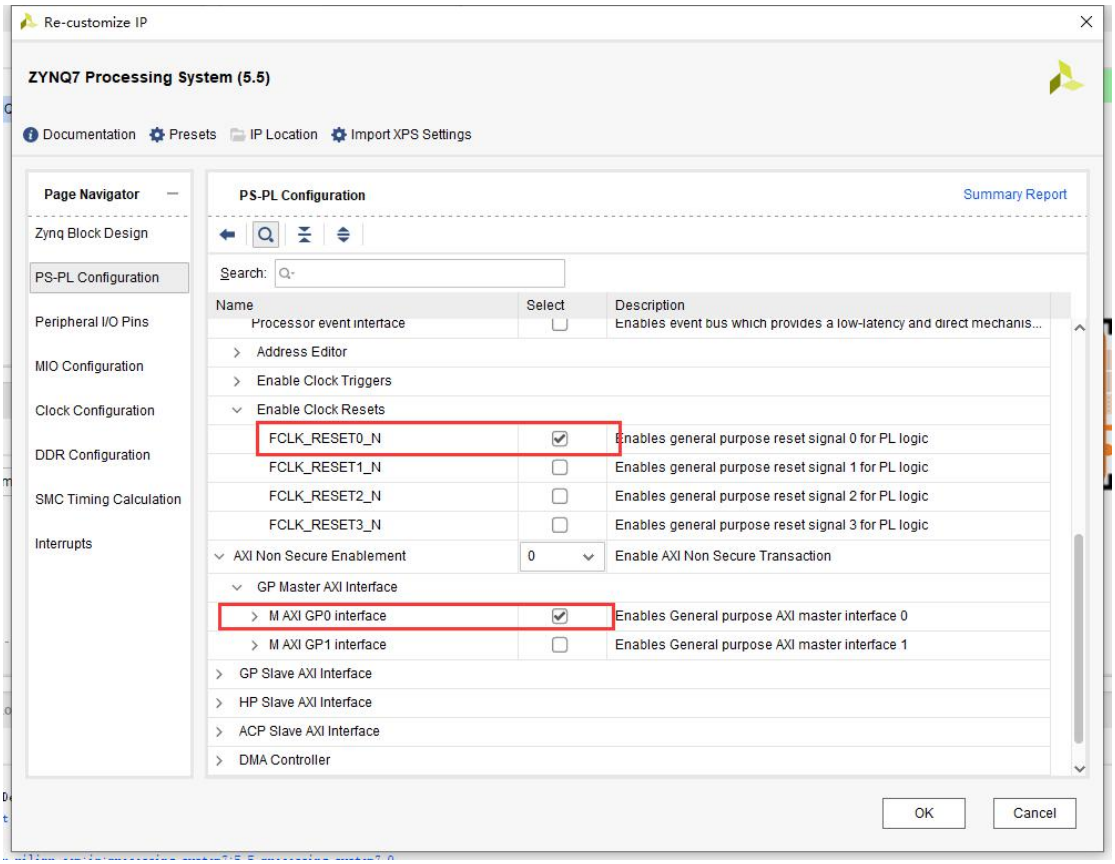


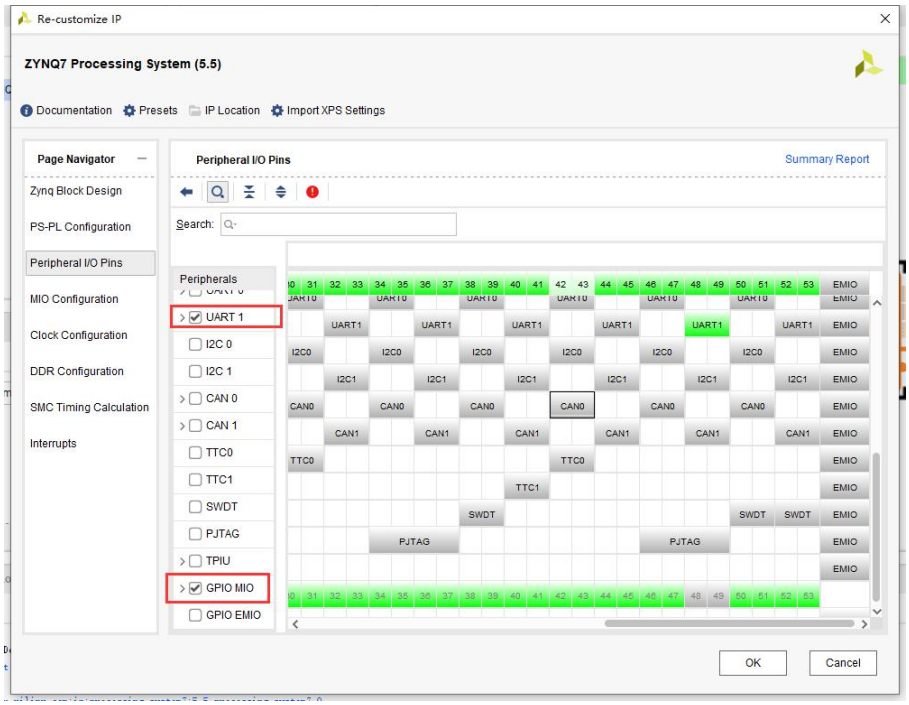
图 5.2.3 新建 Block Design

在之前的实验中，我们取消了许多用于 PS 和 PL 交互的设置的勾选，这次我们要



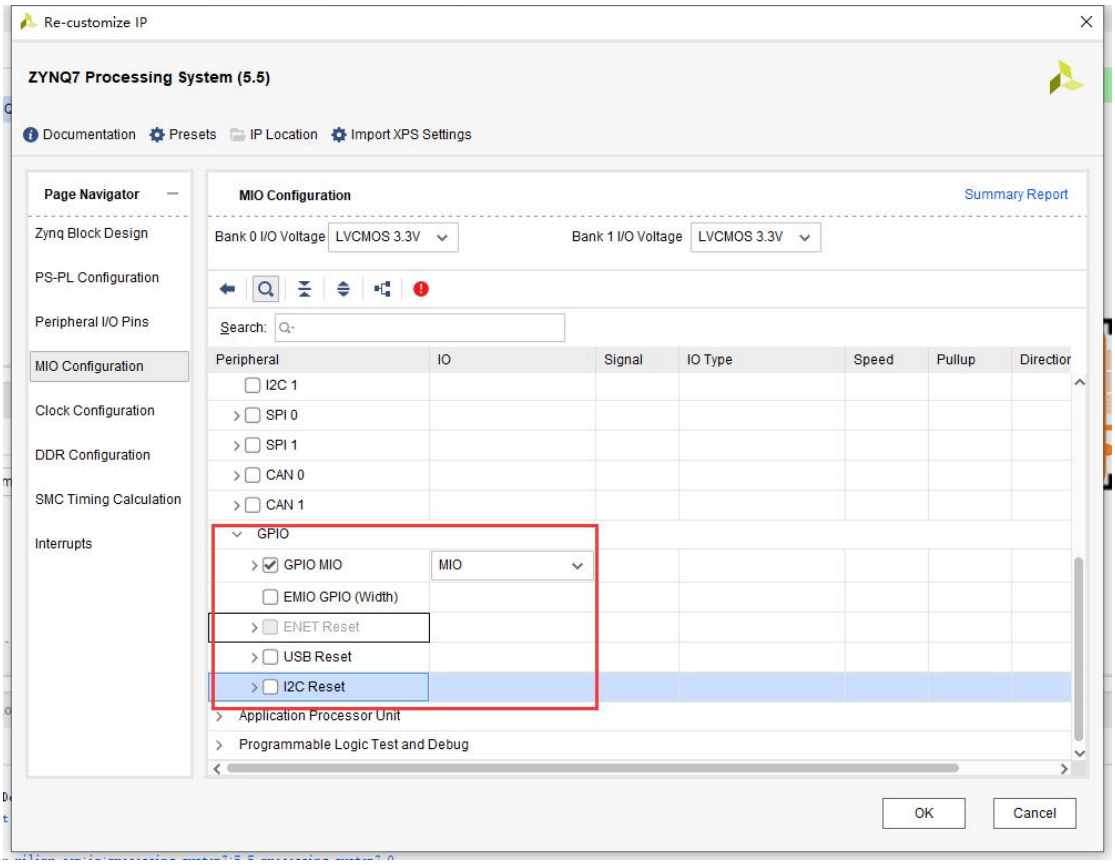
予以保留，如下图所示，保留相关项目的勾选。

图 5.2.4 保留 PS 和 PL 交互接口



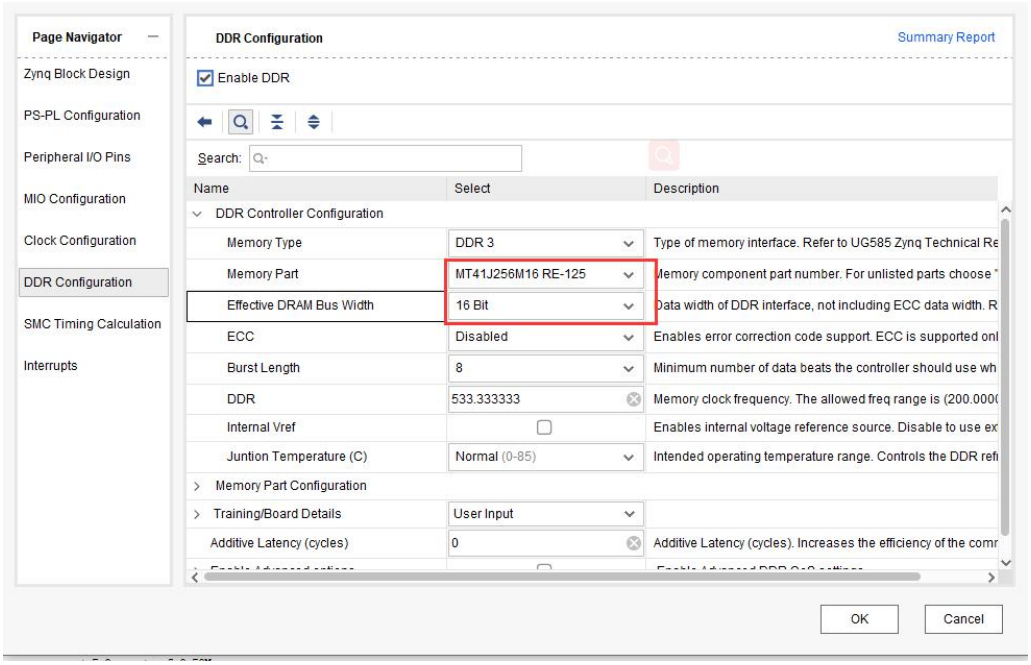
配置 UART、MIO 使能，不同型号开发板 UART 端口不同，请查阅原理图。

图 5.2.5 配置 UART、MIO 使能



在 MIO Configuration 界面取消除 GPIO MIO 以外其他信号的勾选。

图 5.2.6 取消 GPIO MIO 以外信号的勾选



在 DDR Configuration 界面配置 DDR3 控制器。

图 5.2.7 配置 DDR 控制器

另外，本实验还要用到 PL 端的中断，所以在配置页面左侧点击 Interrupt，然后在右侧的界面里面分别勾选“Fabric Intrerrupt”和 PL-PS Interrupt Port 下拉菜单里面的“IRQ_F2P[15:0]”.

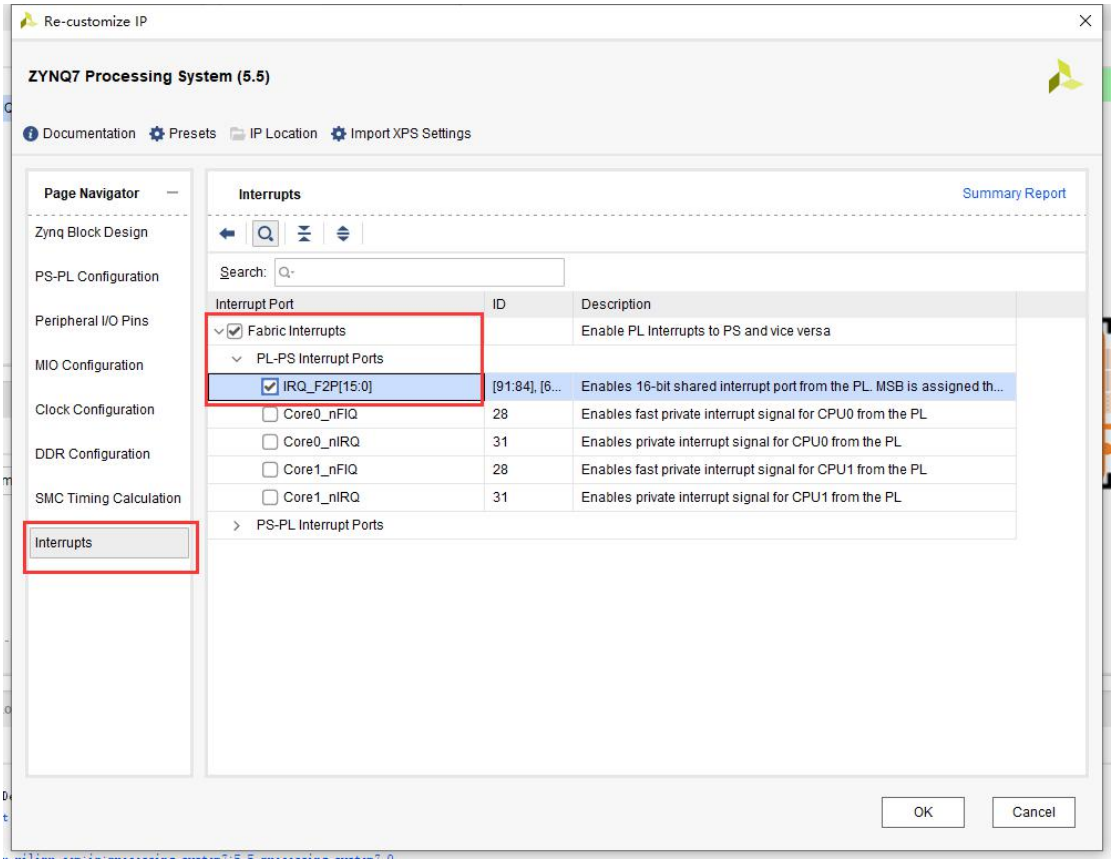


图 5.2.8 配置 PL 端中断

配置完成后点击“OK”完成并返回设计界面。ZYNQ 处理系统的配置就完成了。配置完成后 ZYNQ7 的接口如下图所示：

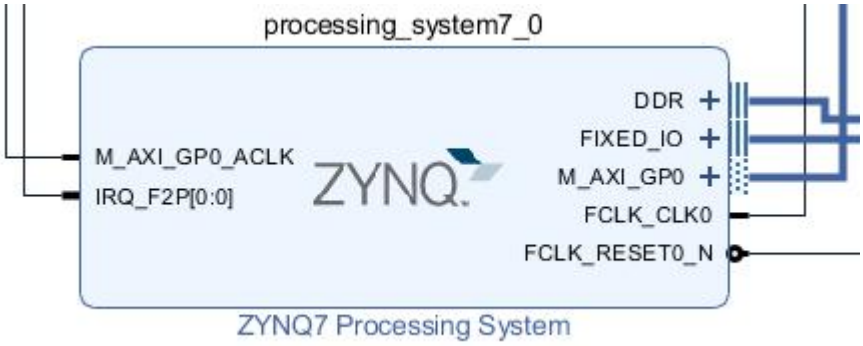


图 5.2.9 ZYNQ7 模块接口

接下来向 Block Design 中添加 AXI GPIO IP 核，点击上方添加 IP 核的选项，在弹出

的 IP 核目录里面键入“AXI GPIO”搜索 IP 核，然后双击搜索到的“AXI GPIO”将其添加到设计中，操作如图所示：

图 5.2.10 搜索添加 AXI GPIO IP 核

添加完成后 Diagram 窗口如下图所示：

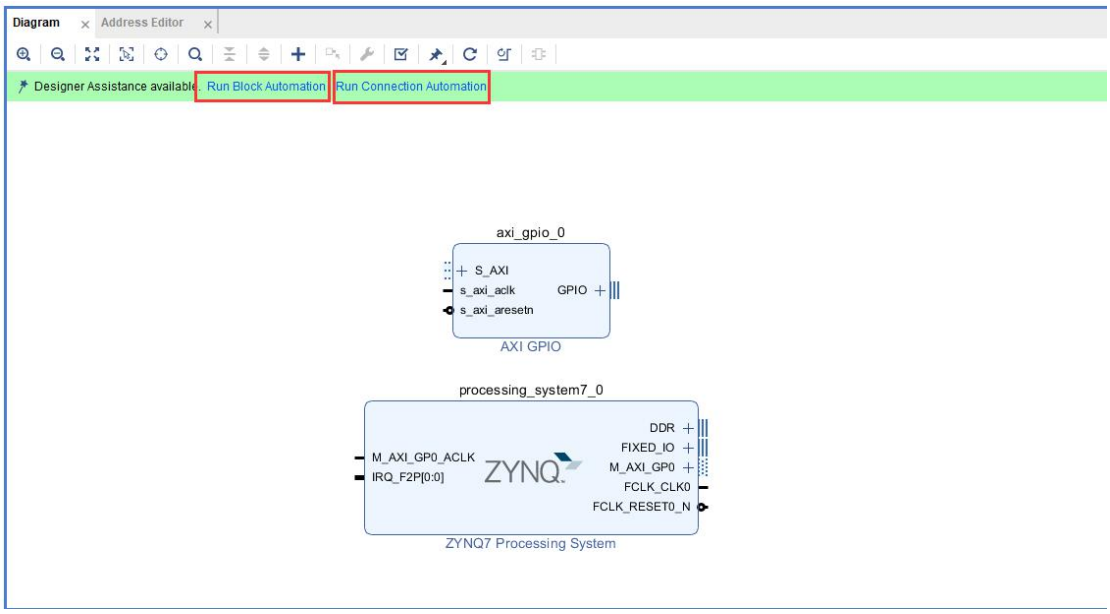


图 5.2.11 添加 AXI GPIO 完成

之后点击图中红色框圈出的“Run Block Automation”，在弹出的对话框中勾选 processing_system7_0，然后点击“OK”，引出 DDR 和 FIXED_IO 端口。如下图所示：

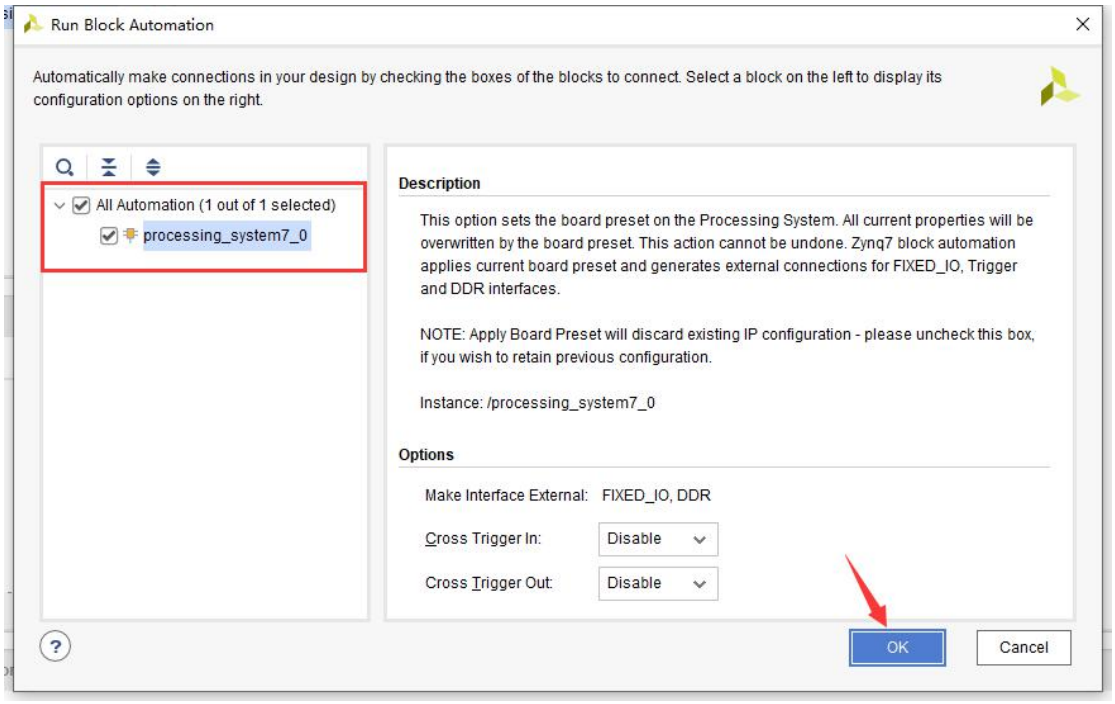


图 5.2.12 引出 DDR 和 FIXED_IO 接口

之后再点击“Run Connection Automation”，在弹出的对话框中全选所有端口，点击“OK”。

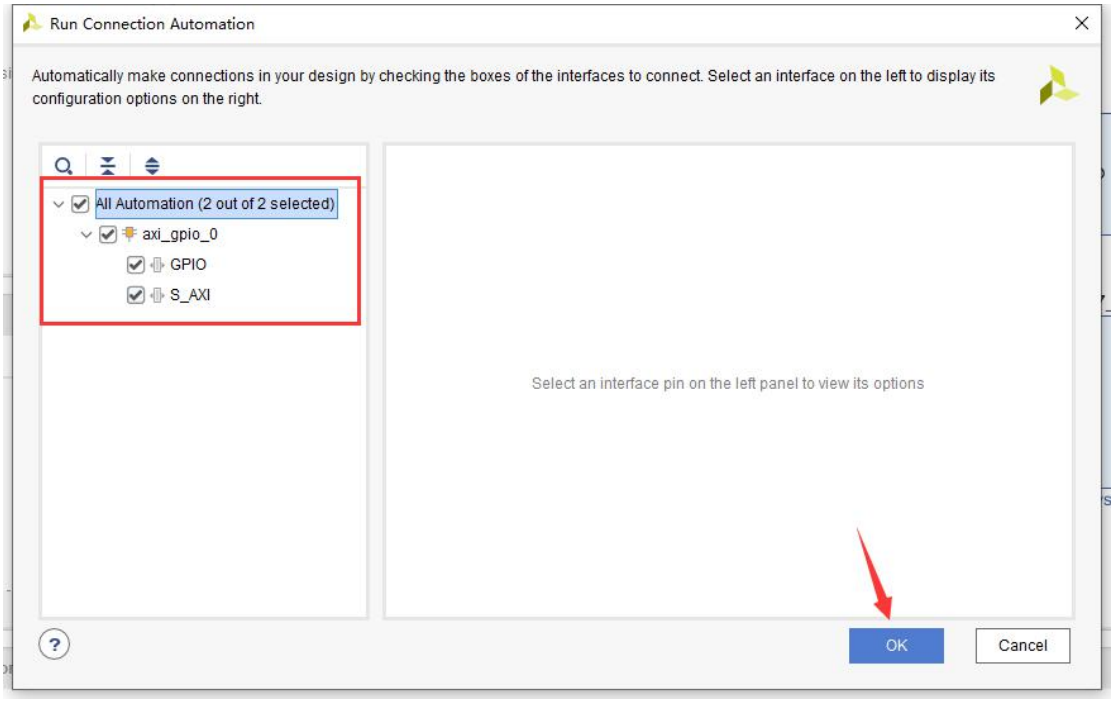


图 5.2.13 自动连接各模块接口

连接完成后如下图所示，在图中我们可以看到除了我们手动添加的两个 IP 之外，软件自动添加了两个 IP 核，分别是 AXI Interconnect 和 Processor System Reset。这是因为仅有我们添加的两个 IP 核系统时无法正常工作的，软件会自动添加部分必要的 IP 使系统正常工作。这两个 IP 核的作用读者可以自行在网上搜索。

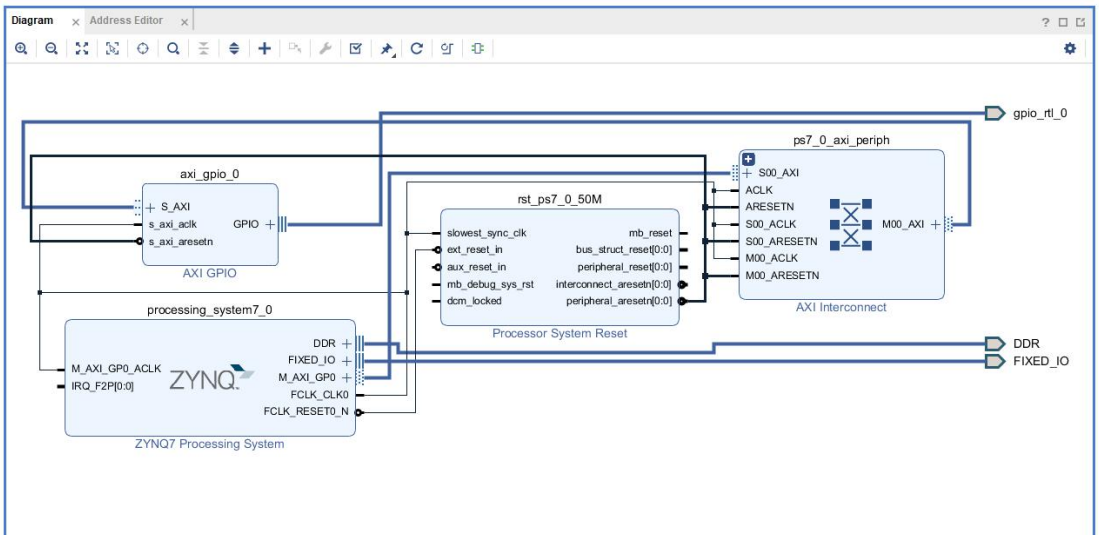
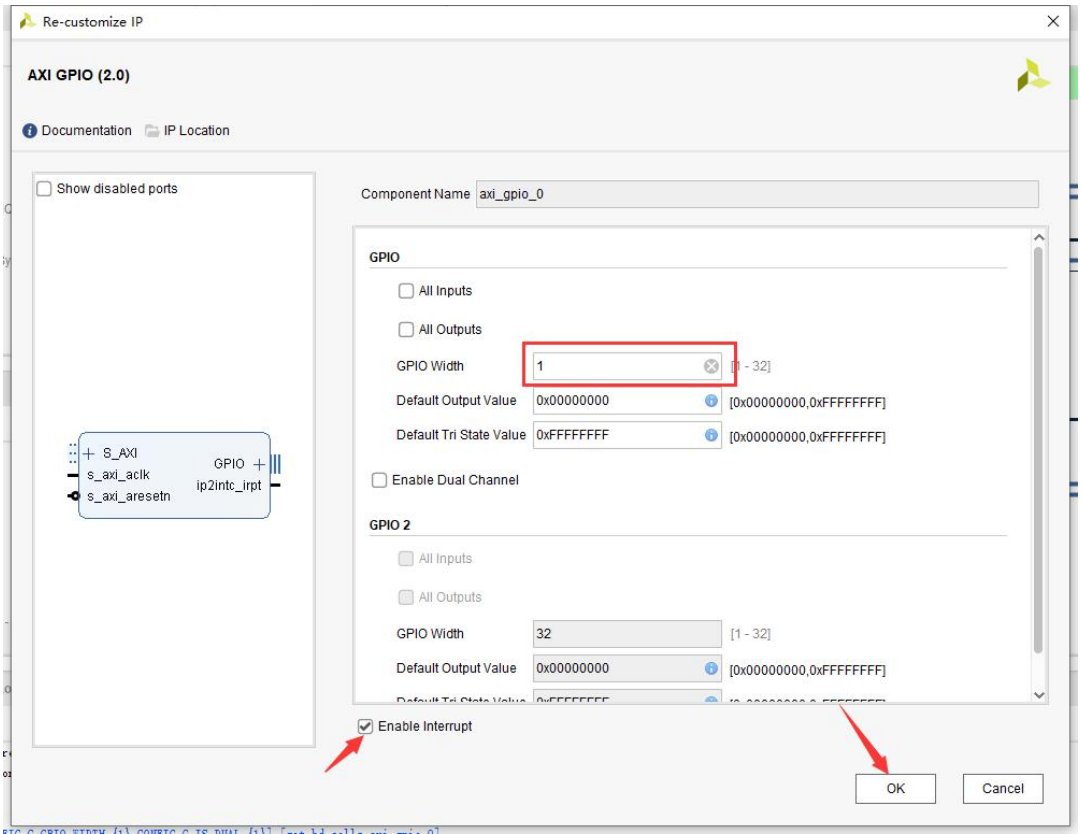


图 5.2.14 连接布局完成后的设计界面

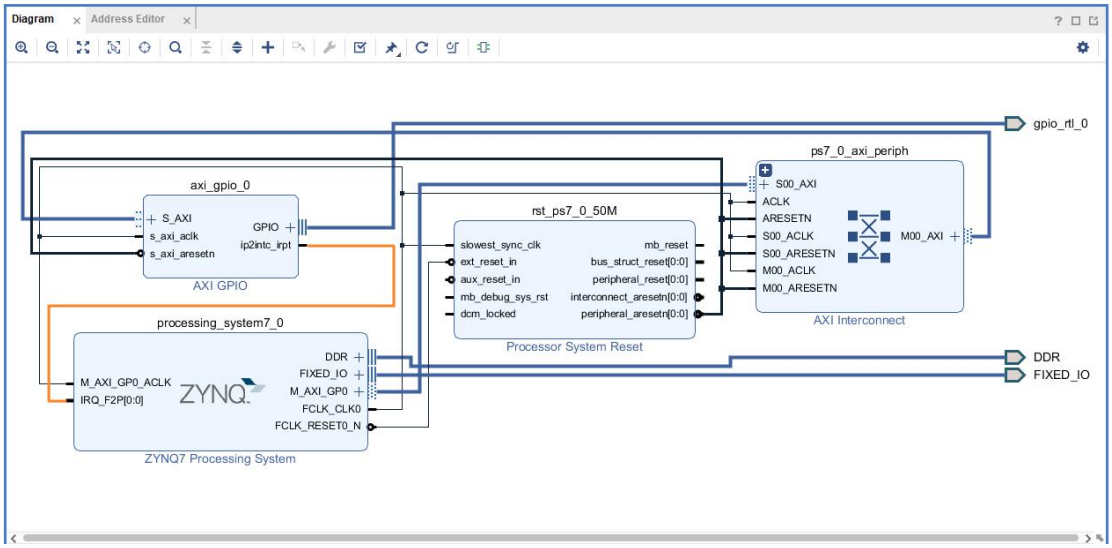
图 5.2.15 AXI GPIO 相关设置

布局完成后我们要配置 AXI GPIO IP 核的相关参数，双击该 IP 核，按照图 5.2.15



配置相关设置。

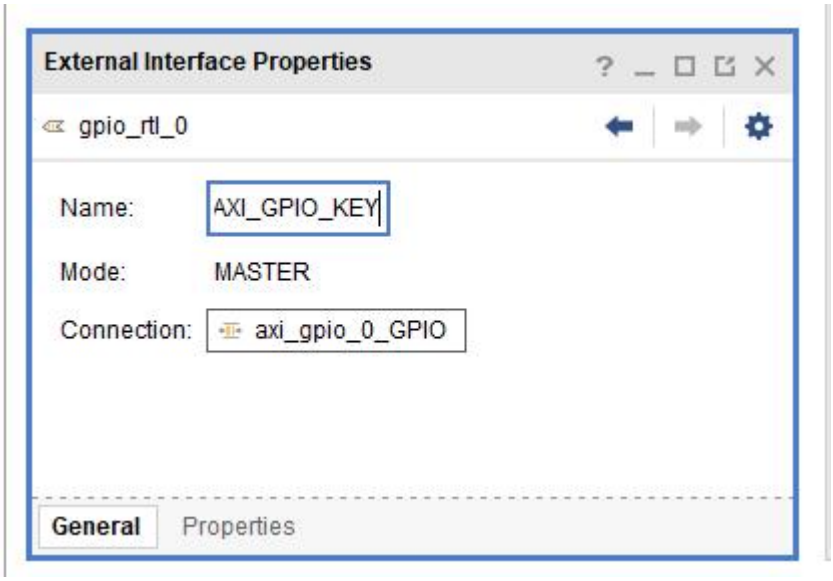
之后将 AXI GPIO 输出的中断信号连接到 ZYNQ7 Processing System 的中断输入接口 IRQ_F2P[0:0]上。将鼠标放到 ip2intc_irpt 上，变为铅笔形状后拖动到



IRQ_F2P[0:0]上即可。

图 5.2.16 连接中断信号

为了方便区分不同的信号，点击信号接口 gpio_trl_0，在左侧的窗口内将信号名称修改为 AXI_GPIO_KEY，如下图所示，即可看到信号名称被修改。



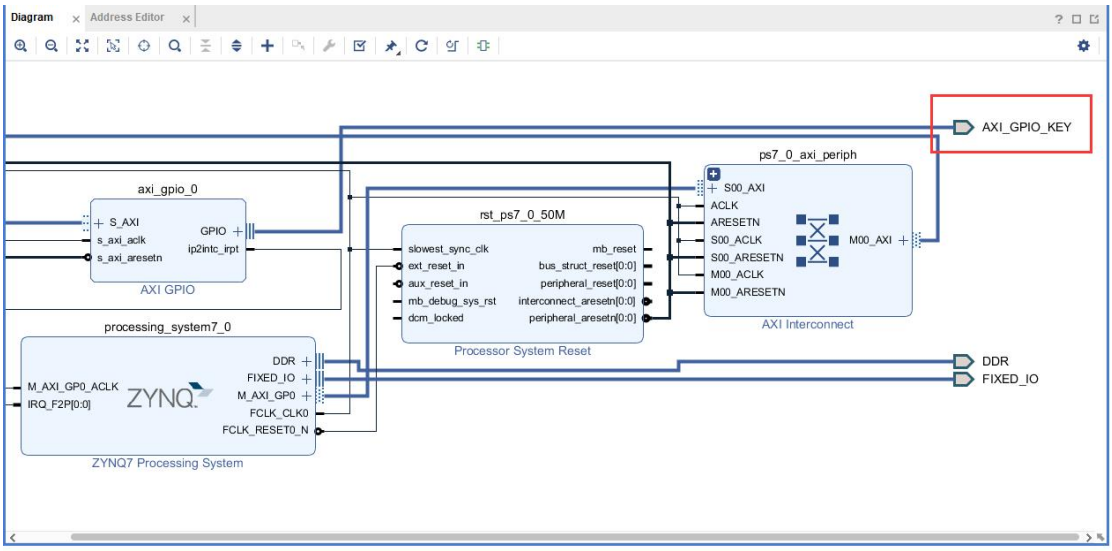


图 5.2.17 修改信号名称

修改完成后点击下图箭头所指选项验证设置，验证没有错误之后快捷键 Ctrl + S 保存设计。

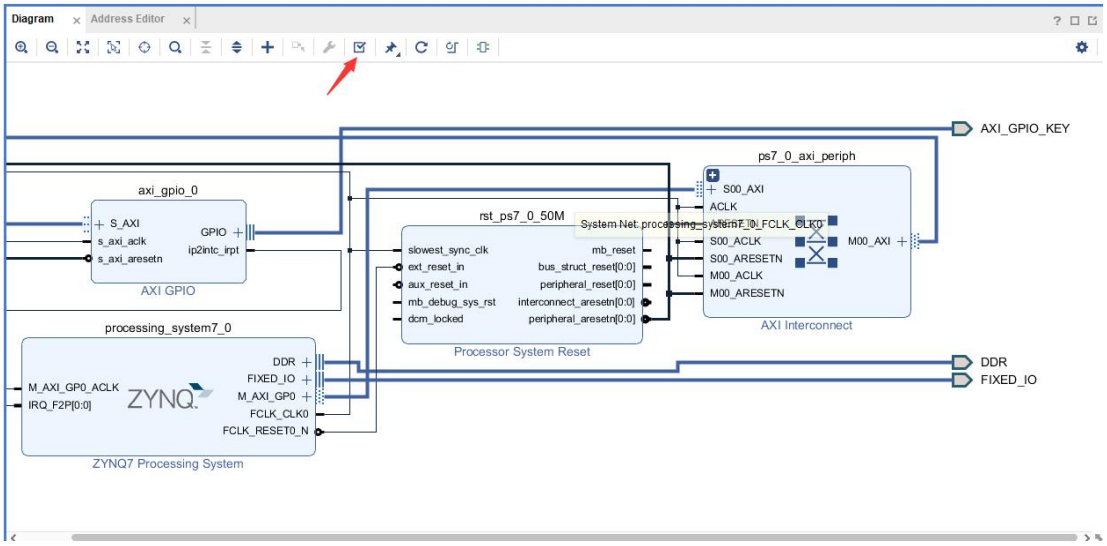
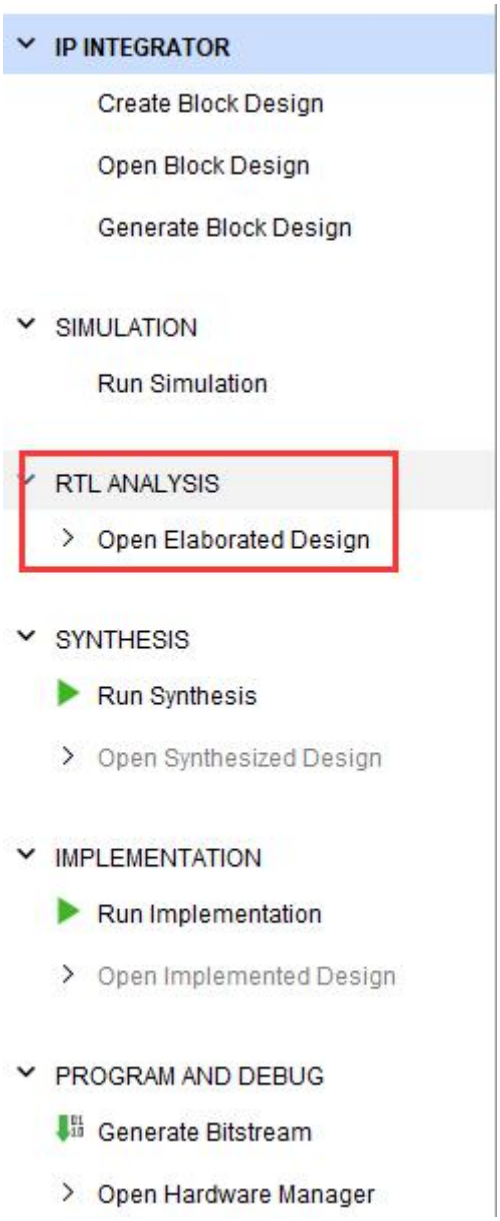


图 5.2.18 验证设计

设计完成之后按照之前的步骤生成顶层 HDL 文件。然后点击“Open Elaborated Design”，在弹出的对话框中点击“OK”打开引脚约束窗口。


IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager

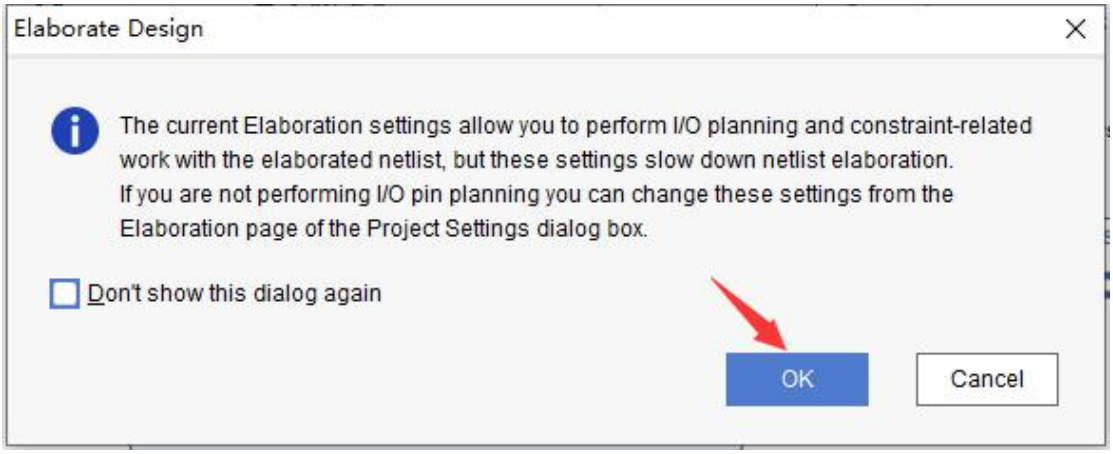
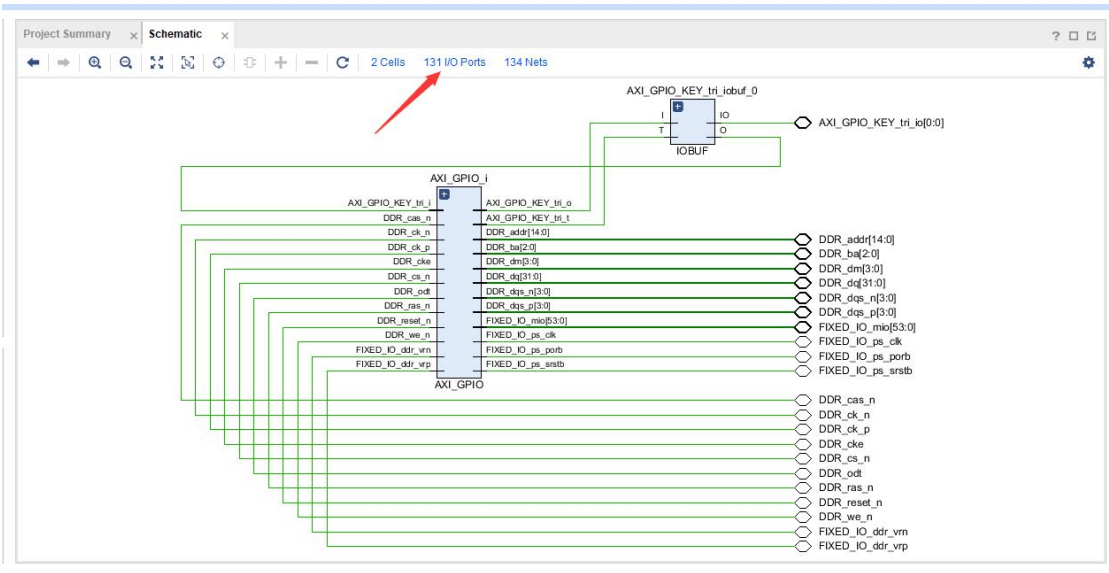


图 5.2.19 打开引脚约束窗口

在打开的窗口中点击 I/O Port，在下方的窗口中找到 AXI_GPIO_KEY，按照下图所示进行配置。对于不同型号的开发板请参照原理图修改为对应的引脚。配置完成后快捷键 Ctrl + S 保存。



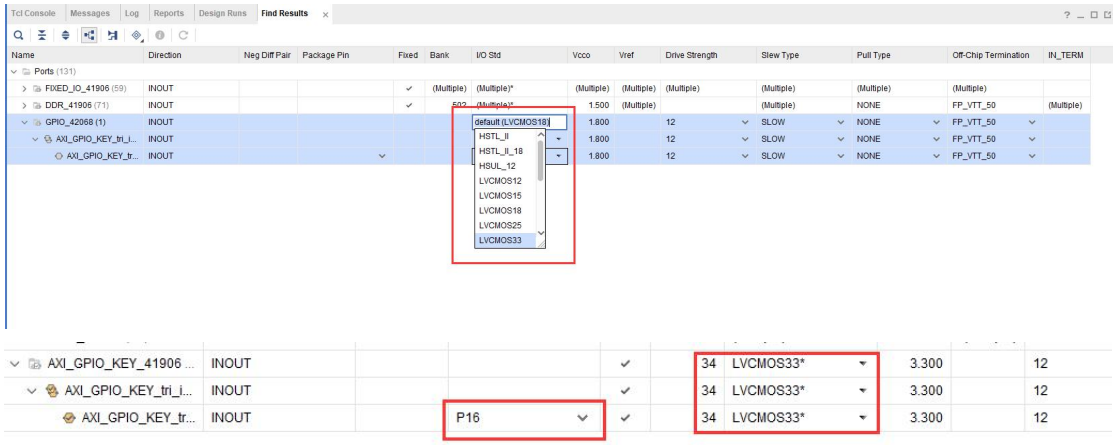
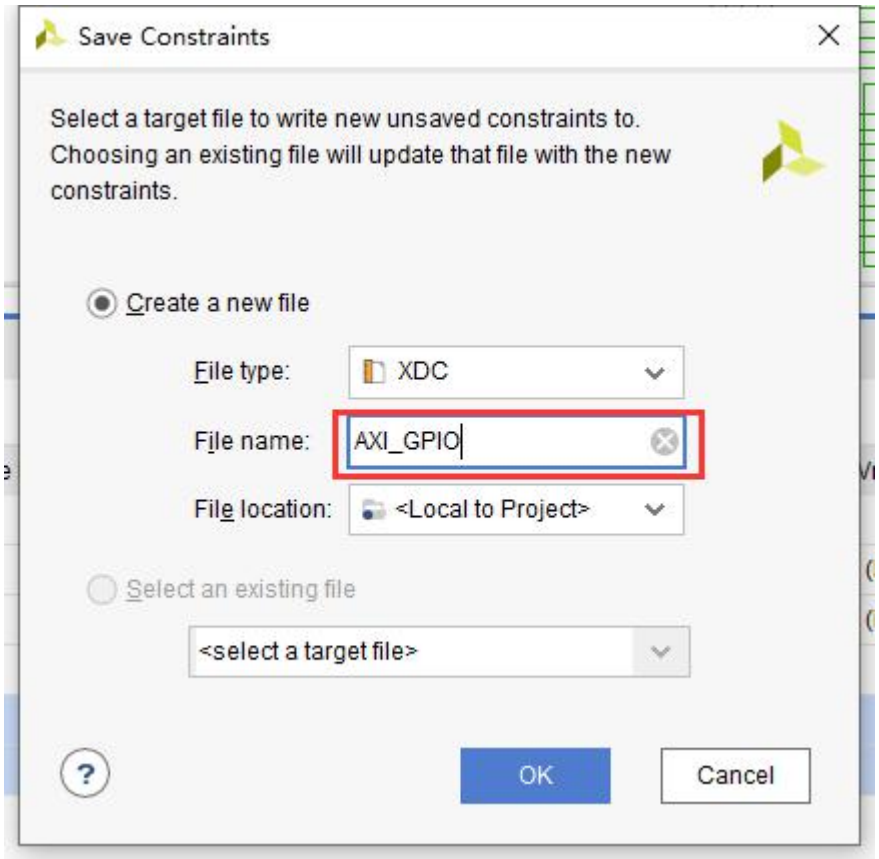
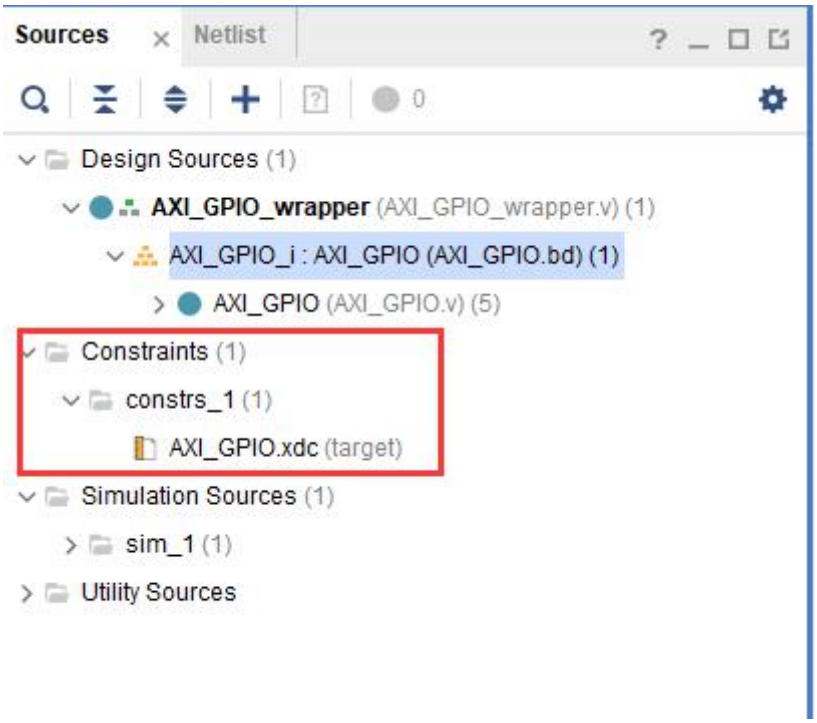


图 5.2.20 配置引脚约束



在弹出的窗口中输入“AXI_GPIO”，点击“OK”保存.xdcwen 件。

图 5.2.21 保存.XDC 文件



之后在 source 窗口下 Constraints->constrs 可以看到保存的文件。

双击打开文件即可看到约束条件，也可直接编写约束语句实现引脚约束。

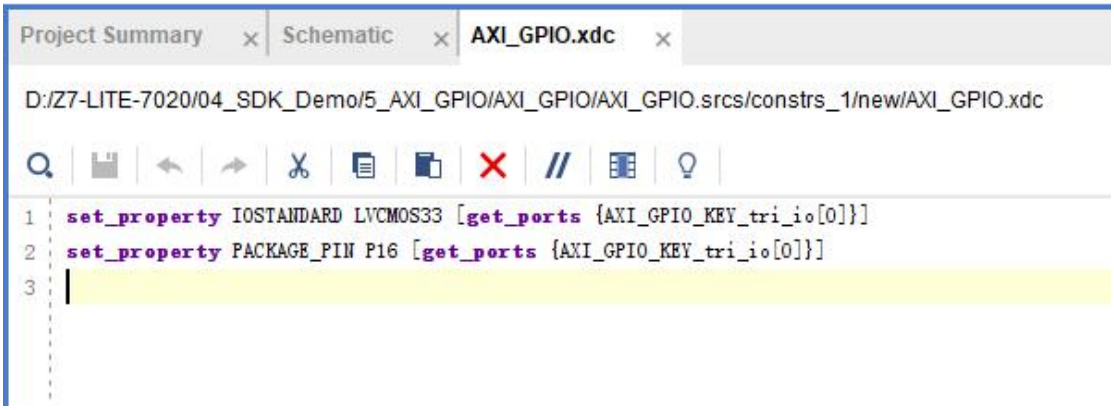
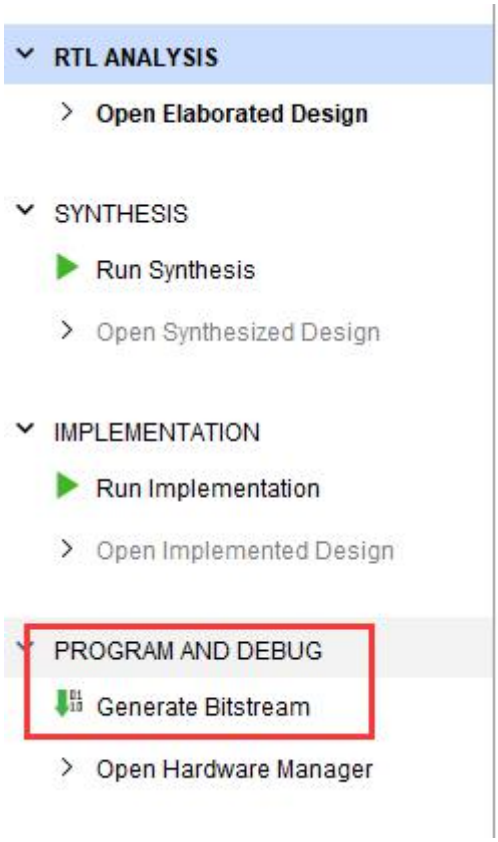
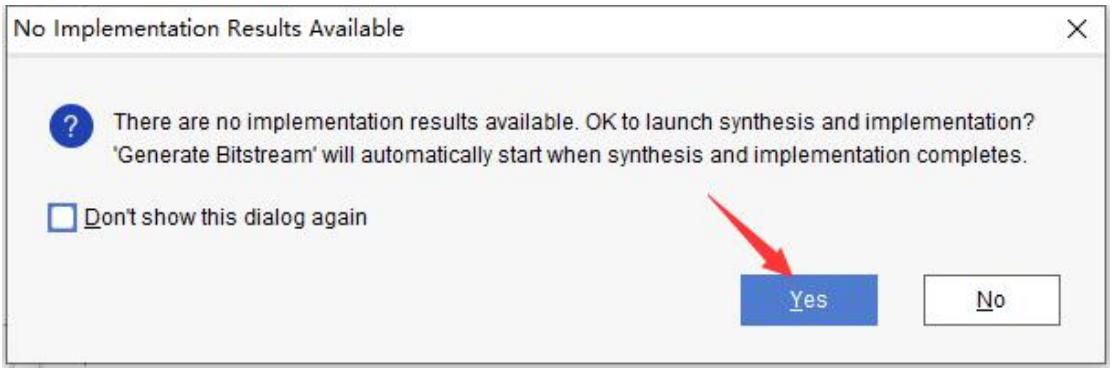


图 5.2.22 引脚约束文件

然后点击左侧导航栏中 Generate Bitstream，在弹出的对话框点击 yes，然后点击 OK 生成比特流文件





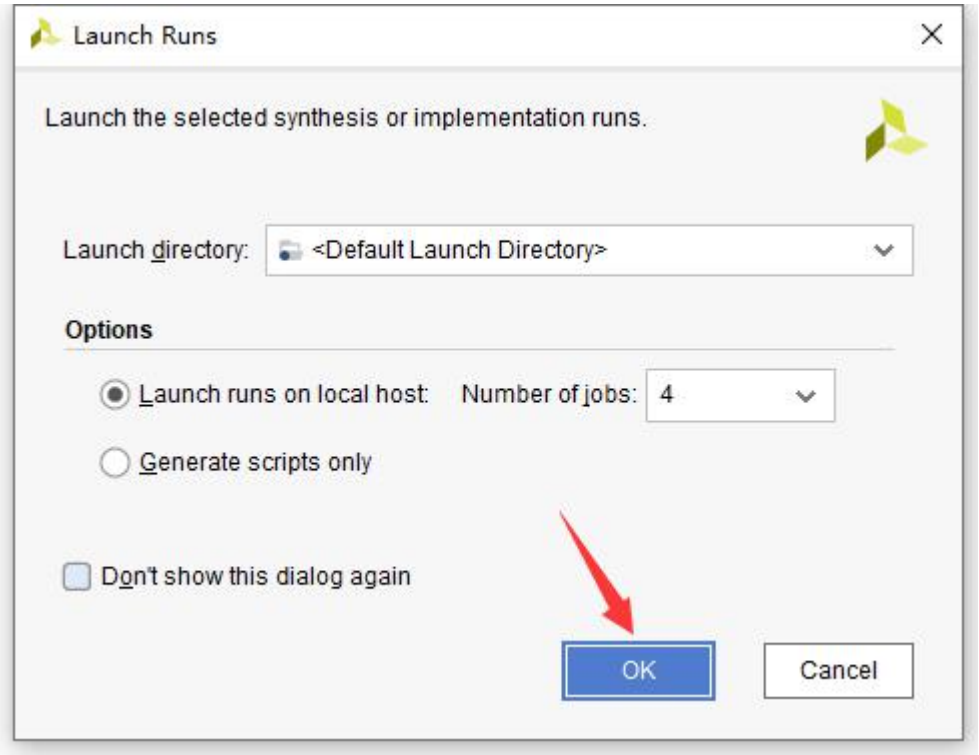


图 5.2.23 生成 Bitstream 文件

Bitstream 文件生成后可以查看综合结果，没有错误说明硬件设计完成，可以导出到 SDK。点击导出硬件描述文件，切记勾选“Include bitstream”，然后点击“OK”。导出完成后点击“launch SDK”打开 SDK.

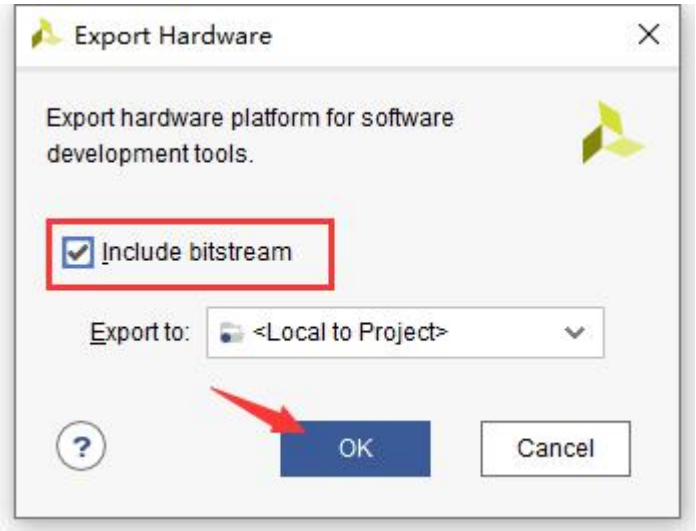
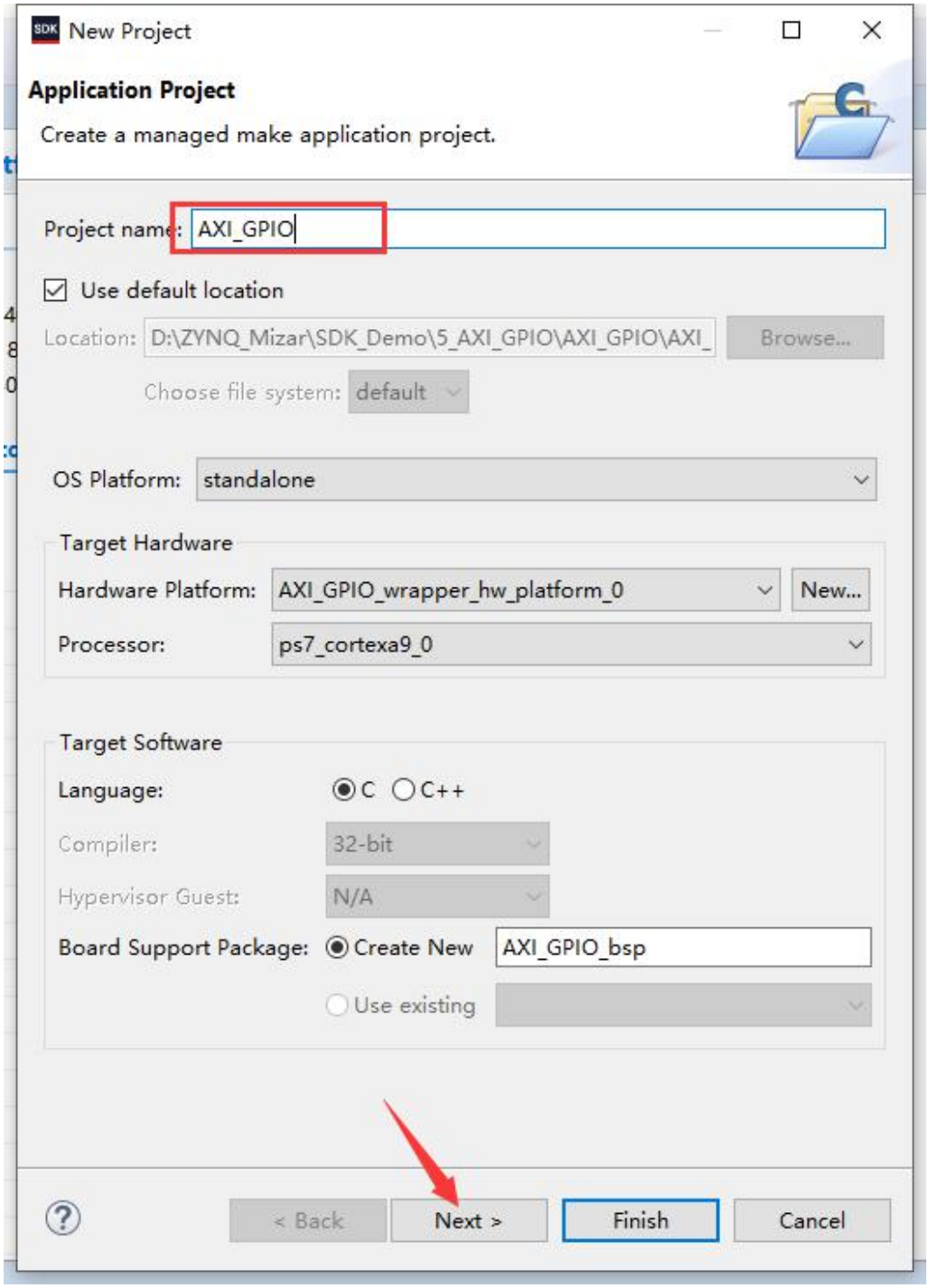


图 5.2.24 导出硬件描述文件

5.3 软件设计

在打开的 SDK 软件中创建一个名为“AXI_GPIO”的新工程，工程模板选择



为空白模板，具体信息如下图所示：

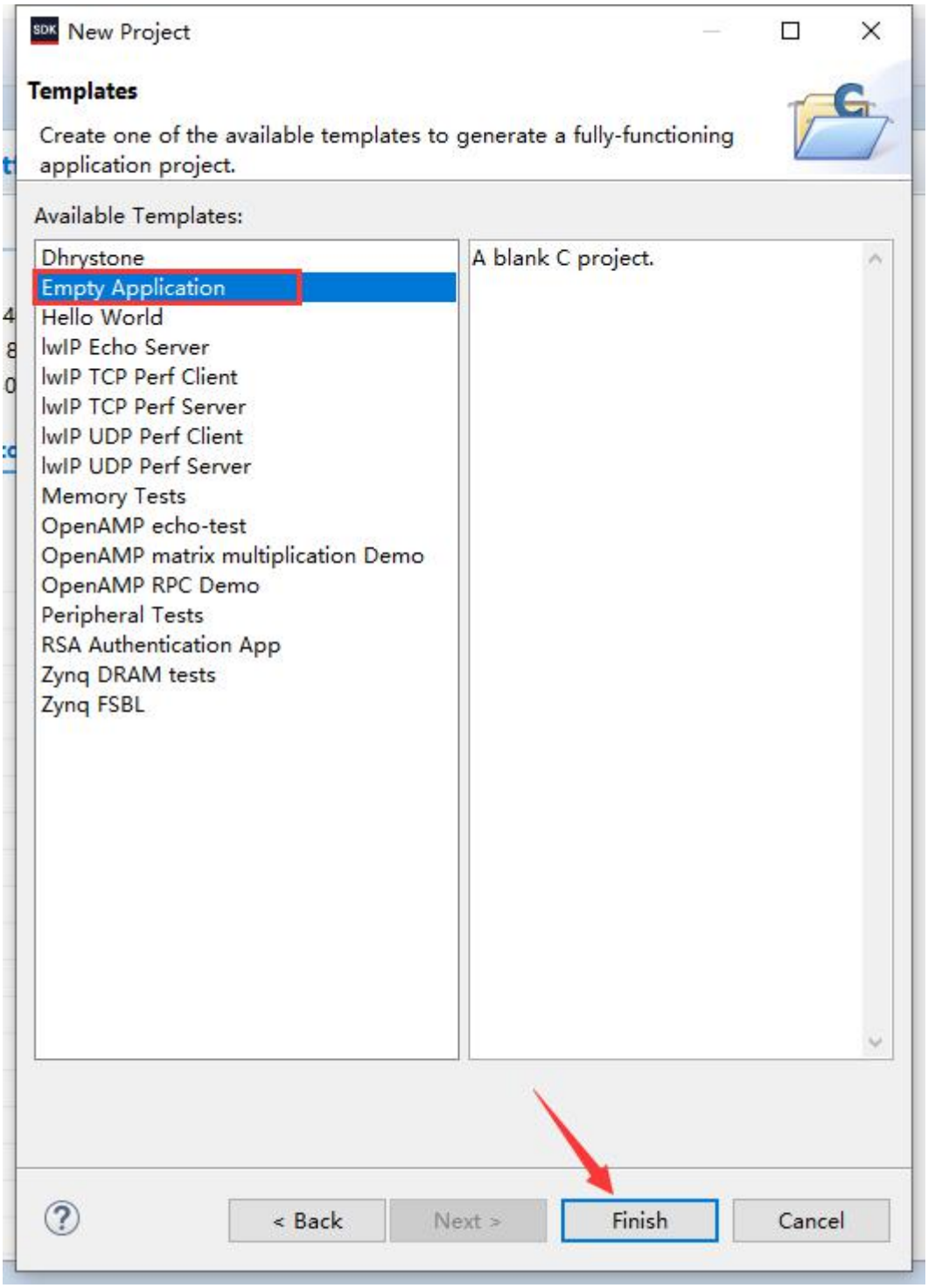
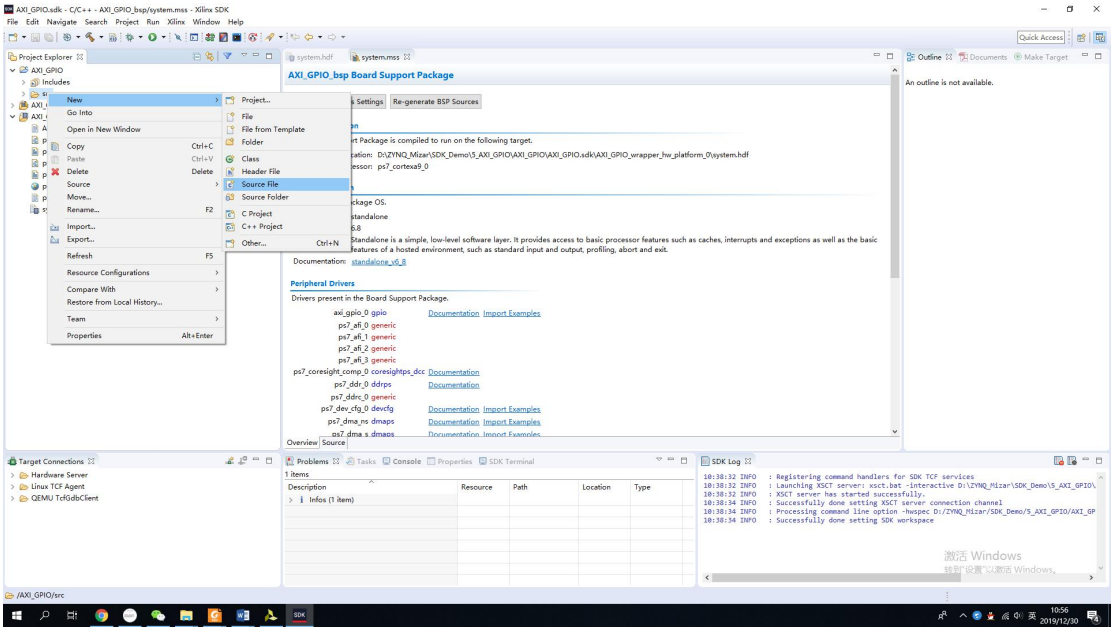


图 5.3.1 创建空白工程“AXI_GPIO”

创建完成后，向工程中添加主程序源文件 main.c，操作如下图所示：



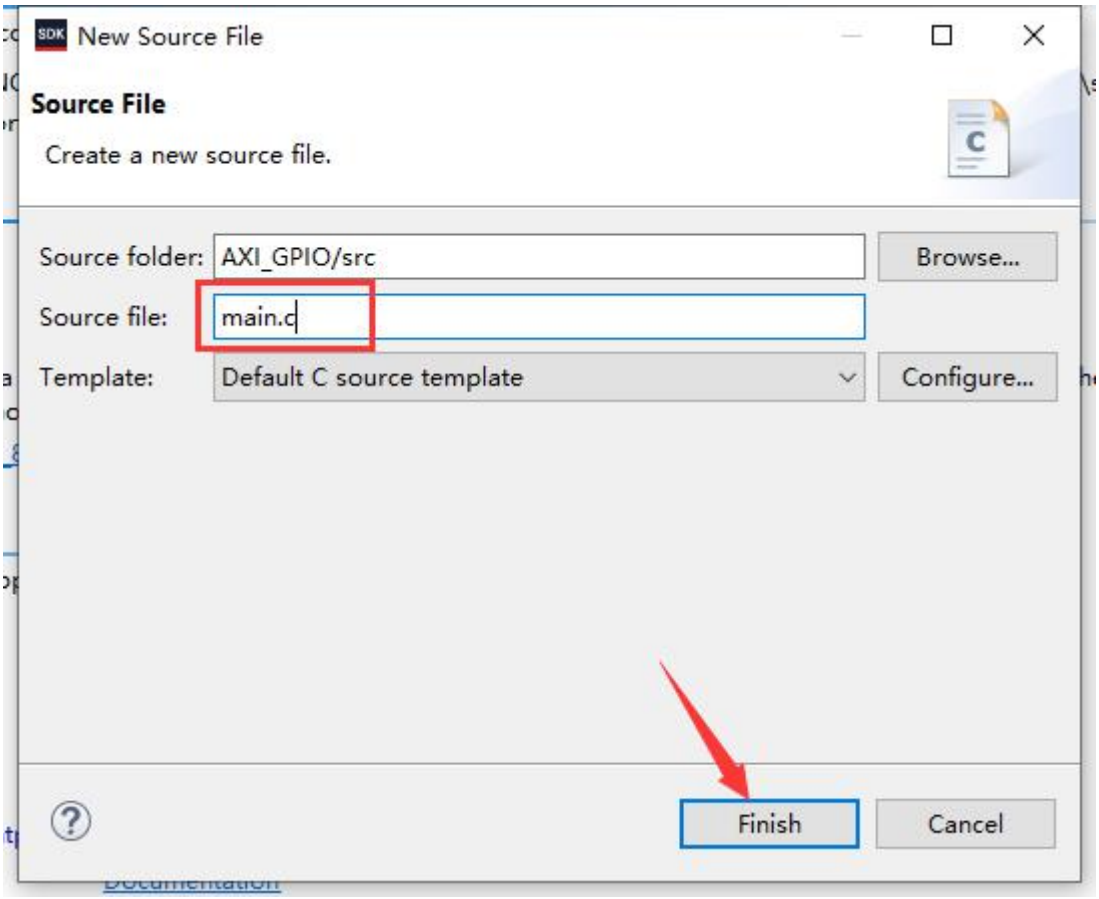


图 5.3.2 添加主程序源文件

在本次实验中，要实现功能为 AXI GPIO 读取按键中断并通知 PS 端，之后 PS 端改变 LED 的亮灭。所以在程序中需要完成的工作有：1、初始化 PS 端 GPIO 驱动，初始化 PL 端 AXI GPIO 驱动，初始化中断控制器驱动；2、配置 PS GPIO 端口方向、使能输出等参数；3、配置 PL 端 AXI GPIO 通道 1 为输入，使能通道 1 中断，使能 AXI GPIO 全局中断（AXI GPIO 读取引脚中断）；4、设置中断相关参数，设置中断优先级和中断触发类型，关联中断源和中断处理函数，使能 AXI GPIO 中断（PS 端读取来自 AXI GPIO 的中断）；5、设置并打开中断异常处理功能；6、编写中断处理函数；7、等待中断触发并做相应处理。

在明确了程序的结构组成之后我们就可以进行程序的编写工作了。在

```
42 #include "stdio.h"
43 #include "xparameters.h"
44 #include "xgpiops.h"
45 #include "xgpio.h"
46 #include "xscugic.h"
47 #include "xil_exception.h"
48 #include "xil_printf.h"
49 #include "sleep.h"
50
51 //器件ID宏定义
52 #define SCUGIC_ID    XPAR_SCUGIC_0_DEVICE_ID      //中断控制器
53 #define GPIOPS_ID    XPAR_XGPIOPS_0_DEVICE_ID     //PS端 GPIO器件
54 #define AXI_GPIO_ID XPAR_AXI_GPIO_0_DEVICE_ID    //PL端 AXI GPIO器件
55 #define GPIO_INT_ID XPAR_FABRIC_GPIO_0_VEC_ID    //PL端 AXI GPIO中断
56
57 //连接端口宏定义
58 #define MIO_LED1      0                          //PS LED1连接到MIO50
59 #define KEY_CHANNEL   1                          //PL_KEY1使用AXI GPIO通道1
60 #define KEY_MASK      XGPIO_IR_CH1_MASK          //通道1的位定义
61
62 //函数声明
63 void axi_gpio_intr_handler(void *CallbackRef);      //中断服务函数
64
65 //PS端GPIO
66 XGpioPs          gpiops;                    //PS端GPIO 驱动实例
67 XGpioPs_Config * gpiops_cfg;                //PS端GPIO 配置信息
68
69 //PL端AXI_GPIO
70 XGpio            axi_gpio;                  //PL端AXI GPIO 驱动实例
71 //中断控制
72 XScuGic          scugic;                    //中断控制器 驱动实例
73 XScuGic_Config * scugic_cfg;                //中断控制器 配置信息
```

```
35 int led_val;                          //LED显示状态
36 int key_flag=0;                       //中断标志
37
38 int main()
39 {
40     printf("AXI_GPIO Test!\n");
41
42     //初始化PS端 GPIO驱动
43     gpiops_cfg = XGpioPs_LookupConfig(GPIOPS_ID);
44     XGpioPs_CfgInitialize(&gpiops, gpiops_cfg, gpiops_cfg->BaseAddr);
```

main.c 中输入以下程序：

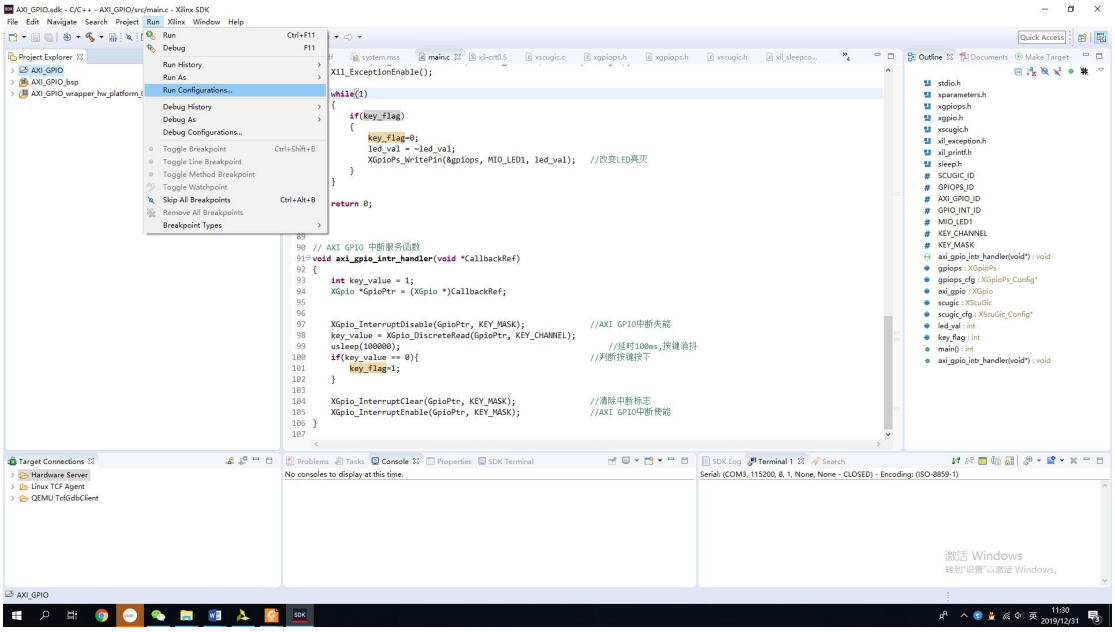
```
    //初始化中断控制器驱动
    scugic_cfg = XScuGic_LookupConfig(SCUGIC_ID);
    XScuGic_CfgInitialize(&scugic, scugic_cfg, scugic_cfg->CpuBaseAddress);

    //配置PS GPIO
    XGpioPs_SetDirectionPin(&gpiops, MIO_LED1, 1);      //设置 PS GPIO 为输出
    XGpioPs_SetOutputEnablePin(&gpiops, MIO_LED1, 1);   //使能LED输出
    XGpioPs_WritePin(&gpiops, MIO_LED1, led_val);       //点亮LED

    //配置PL AXI GPIO
    XGpio_SetDataDirection(&axi_gpio, KEY_CHANNEL, 1); //设置PL AXI GPIO通道1为输入
    XGpio_InterruptEnable(&axi_gpio, KEY_MASK);     //使能通道1中断
    XGpio_InterruptGlobalEnable(&axi_gpio);         //使能AXI GPIO全局中断

    //设置中断优先级和触发类型(高电平触发)
    XScuGic_SetPriorityTriggerType(&scugic, GPIO_INT_ID, 0xA0, 0x1);
    //关联中断源AXI_GPIO和中断处理函数axi_gpio_intr_handler
    XScuGic_Connect(&scugic, GPIO_INT_ID, axi_gpio_intr_handler, &axi_gpio);
    //使能AXI GPIO中断
    XScuGic_Enable(&scugic, GPIO_INT_ID);

    //设置并打开中断异常处理功能
    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler)XScuGic_InterruptHandler, &scugic);
    Xil_ExceptionEnable();
```

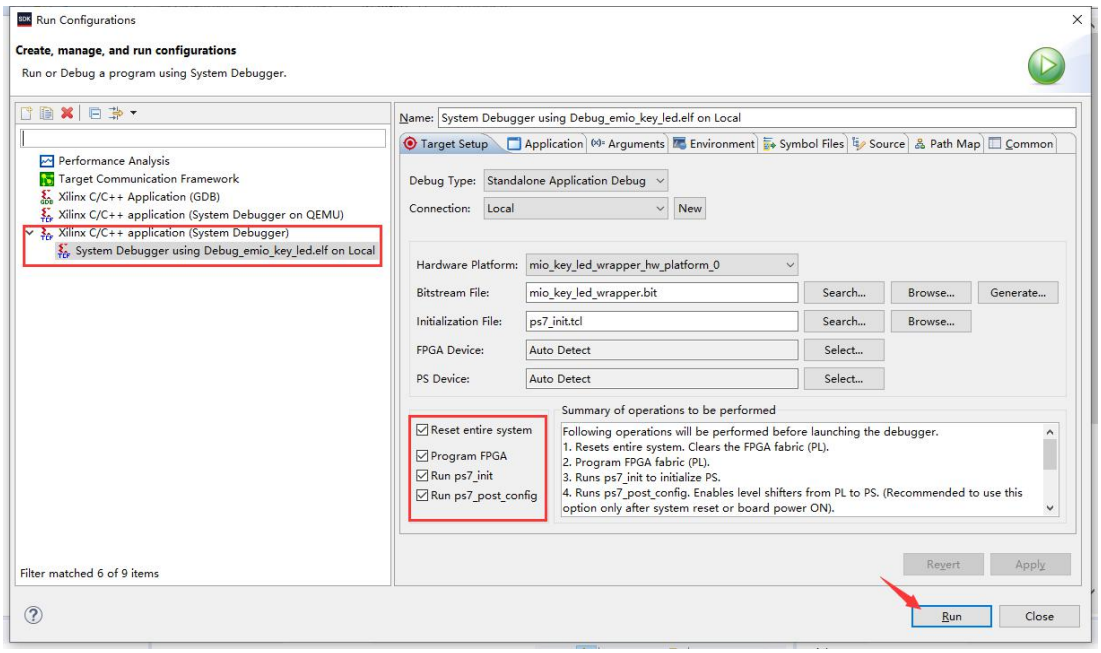
```
    while(1)
    {
        if(key_flag)
        {
            key_flag=0;
            led_val = ~led_val;
            XGpioPs_WritePin(&gpiops, MIO_LED1, led_val);   //改变LED亮灭
        }
    }

    return 0;
}


// AXI GPIO 中断服务函数
void axi_gpio_intr_handler(void *CallbackRef)
{
    int key_value = 1;
    XGpio *GpioPtr = (XGpio *)CallbackRef;


    XGpio_InterruptDisable(GpioPtr, KEY_MASK);              //AXI GPIO中断失能
    key_value = XGpio_DiscreteRead(GpioPtr, KEY_CHANNEL);
    usleep(100000);                                             //延时100ms,按键消抖
    if(key_value == 0){                                     //判断按键按下
        key_flag=1;
    }

    XGpio_InterruptClear(GpioPtr, KEY_MASK);                //清除中断标志
    XGpio_InterruptEnable(GpioPtr, KEY_MASK);               //AXI GPIO中断使能
}
```

输入完成后快捷键 Ctrl + S 保存并编译程序。软件会自动生成可执行 elf 文件。

5.4 下载验证

首先连接开发板的电源，然后使用两根 TYPE-C 线一端连接开发板的 JTAG 下载接口和串口，另一端连接电脑的 USB 接口。在 SDK 软件上添加 Terminal 窗口并连接串口。然后点击菜单栏 Run->Run Configurations，然后在弹出的窗口双击 xilinx C/C++ application （System Debugger），在右侧勾选 Reset entire system 和



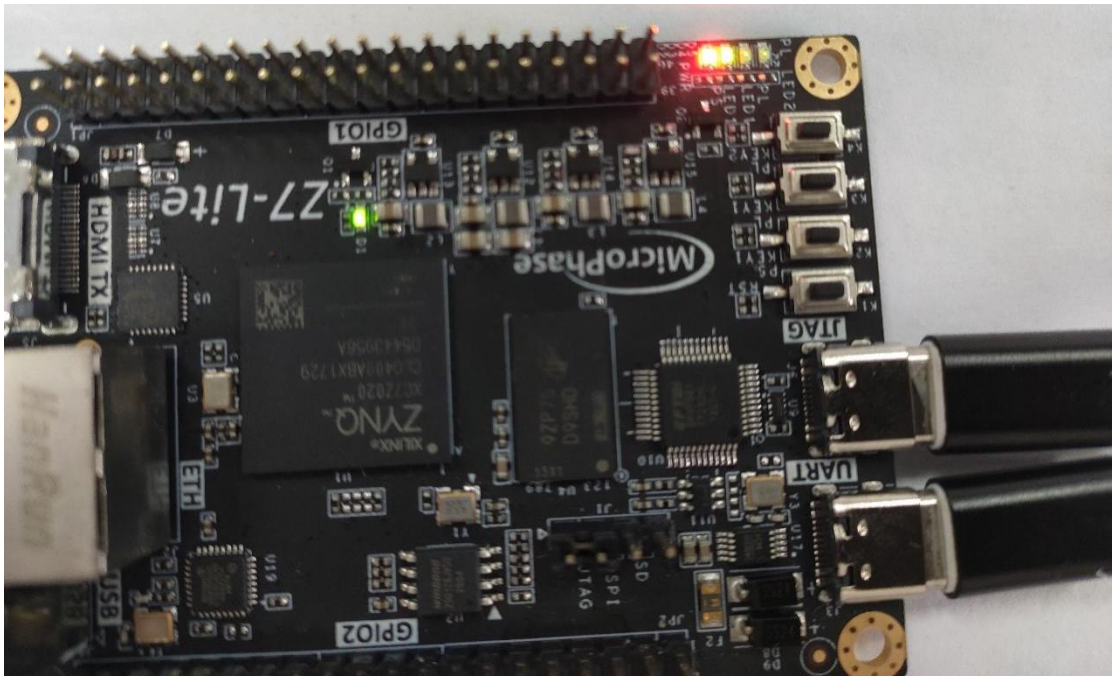


Program FPGA，之后点击 Run 即可烧录程序。

图 5.4.1 烧录程序

烧录完成后可以看到在 Terminal 窗口打印出“AXI GPIO Test！”说明程序已经运行，然后多次按下 PL_KEY1 可以看到 PS_LED1 的亮灭不断变化。说明程序设计正确。

实验结果如下图所示：



对于其他型号的开发板实现本实验，在硬件设计时在芯片选择界面选择对应型号，在引脚约束时查看开发板原理图配置引脚。在软件设计时，在宏定义处修改对应的 MIO 端口号即可。

5.5 章节小结

在本实验中，通过添加 IP 核，实现了 PS 端和 PL 端的交互。这也是 ZYNQ 最大的优势，那就是将单片机和 FPGA 相结合，利用单片机的逻辑处理优势，结合 FPGA 的高速数据运算和可编程逻辑电路的灵活性。在后续的实验中，我们还

将深入的学习这种组合的使用方式。

第 6 章　　定时器中断实验

定时器是嵌入式系统重要的组成部分之一，大多数嵌入式系统需要定时器的参与来实现计时、定时、控制输出脉冲宽度、计算输入脉冲宽度、监控系统是否正常运行（看门狗定时器）、中断产生等功能。本章我们将完成定时器中断实验，了解 PS 端私有定时器的使用方法。

本章包括以下内容：

6.1 实验简介

6.2 硬件设计

6.3 软件设计

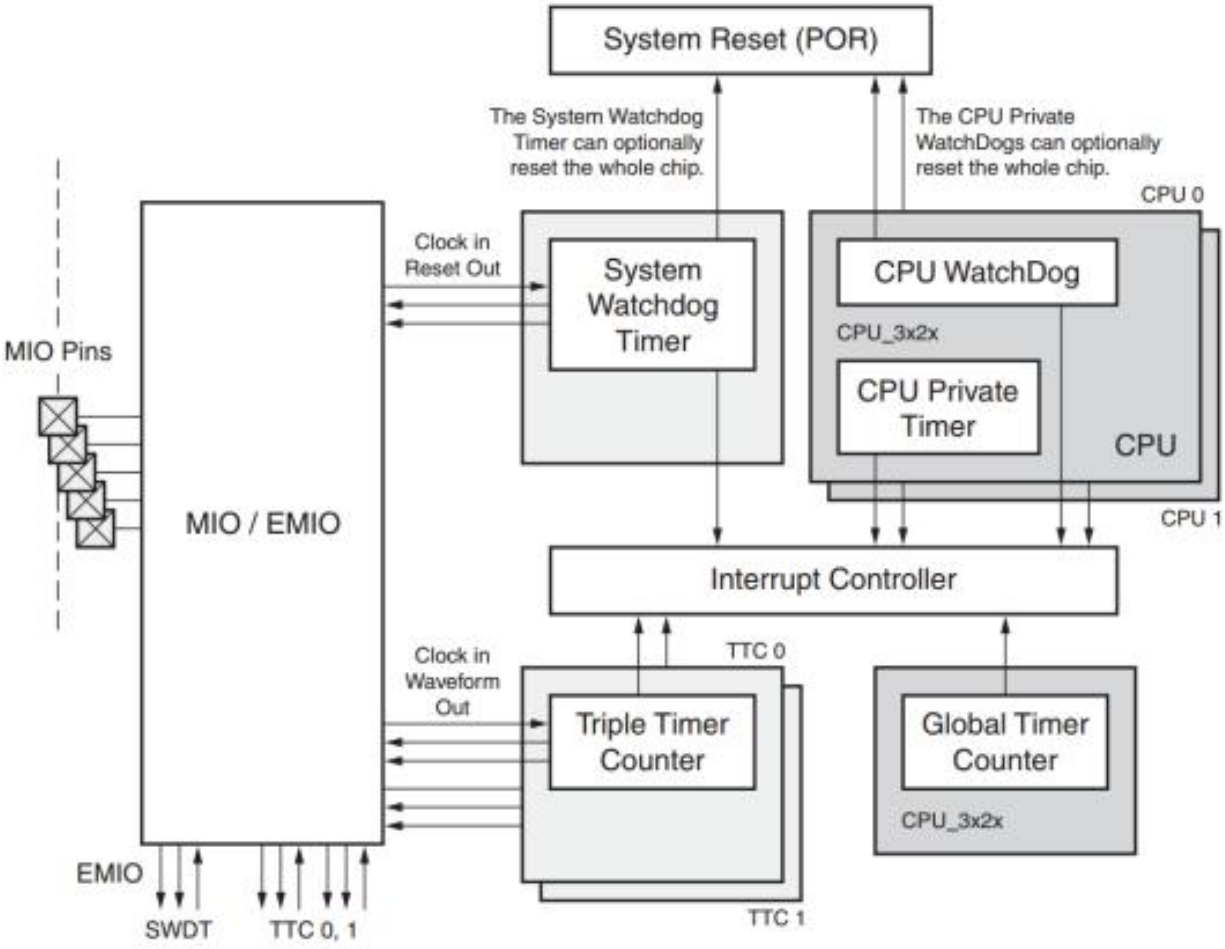
6.4 下载验证

6.5 本章小结

6.1 实验简介

打开官方资料中关于定时器的描述可以看到定时器的结构框图如下图所示。从图中可以看到系统所包含的定时器资源有：系统看门狗定时器（Syetem Watchdog Timer）、每个 CPU 处理器包含一个 32 位看门狗和 32 位的私有定时器、两个 Triple Timer Counter 定时器、一个 64 位共享定时器。所有定时器都与中断控制器相连，可以产生中断。

在本实验中，我们使用 PS 端的私有定时器每秒产生一个中断信号来实现一个 30 秒的倒计时。通过此实验来学习私有定时器的配置和使用方法，了解 ZYNQ



系统定时器的特点。

图 6.1.1 定时器结构框图

6.2 硬件设计

本次实验需要用到的资源有 PS 端的 UART 和 DDR 控制器，与实验 1 用到的资源相同，所以我们不再重新建立新工程而是利用实验 1 的工程。在 Vivado 中打开实验 1 的工程，另存为一个新的工程，命名为“timer_interrupt”，然后删除新工程内的.sdk 文件夹。之后导出硬件描述文件并打开 SDK。就完成了硬件设计。

6.3 软件设计

硬件导出到 SDK 之后就可以创建新的工程开始进行软件设计工作。建立一个新的 SDK 工程并命名为“timer_interrupt”，创建的步骤与之前相同，在此不再赘述。在 timer_interrupt->src 下添加主函数源文件并命名为“main.c”。

在进行程序的编写之前，我们首先了解定时器的一些相关知识。私有定时器的时钟频率为 CPU 时钟频率的一半，如 ARM 的工作时钟频率为 666.666Mhz，则私有定时器的时钟频率为 333.333Mhz。私有定时器有一个向下计数的 32 位计数器，计数到 0 时产生中断信号；8 位预分频器，可控制定时器频率；可配置单次计时或自动重载模式。

在本次实验中我们将利用私有定时器每一秒产生一个中断，利用这个中断实现一次 30 秒的倒计时，并通过串口打印出来。那么程序实现的流程为：1、初始化定时器，设置重装载计数器值为 CPU 频率的一半，设置为自动装载模式；2、初始化中断控制器，打开中断异常处理功能；3、关联中断源与中断服务函数，使能中断控制器中的定时器中断，使能私有定时器中断产生；4、设置中断服务

函数（每次中断产生倒计时秒数减一并打印）；5、启动定时器，判断是否倒计时结束，若结束则关闭定时器中断。

根据上面描述的程序流程，我们在 main.c 中输入以下程序：

```c
#include "xparameters.h"
#include "xscutimer.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_printf.h"

/************************** Constant Definitions *****************************/

 * The following constants map to the XPAR parameters created in the
#define TIMER_DEVICE_ID     XPAR_XSCUTIMER_0_DEVICE_ID   //定时器ID
#define INTC_DEVICE_ID      XPAR_SCUGIC_SINGLE_DEVICE_ID //中断ID
#define TIMER_IRPT_INTR     XPAR_SCUTIMER_INTR           //定时器中断ID


#define TIMER_LOAD_VALUE    XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ/2                    //定时器装载值

/************************** Variable Definitions *****************************/
XScuGic IntcInstance;               //中断控制器驱动程序实例
XScuTimer TimerInstance;            //定时器驱动程序实例

/************************** Function Prototypes ******************************/
int Timer_init(XScuTimer *timer_ptr);
static void TimerIntrHandler(void *CallBackRef);
void TimerIntrInit(XScuGic *intc_ptr,XScuTimer *timer_ptr);

int second=30;
```

```c
//main函数
int main()
{
    int status;
    xil_printf("SCU Timer Interrupt Example Test \r\n");

    status = Timer_init(&TimerInstance);     //定时器初始化
    if (status != XST_SUCCESS) {
        xil_printf("SCU Timer Interrupt Example Test Failed\r\n");
        return XST_FAILURE;
    }
    TimerIntrInit(&IntcInstance,&TimerInstance);   //定时器中断初始化
    XScuTimer_Start(&TimerInstance);         //启动定时器

    while(1)
    {

        if(second==0)
            break;

    }
    XScuTimer_DisableInterrupt(&TimerInstance);
    xil_printf("SCU Timer Interrupt Test Finish\r\n");
    return 0;
}
```

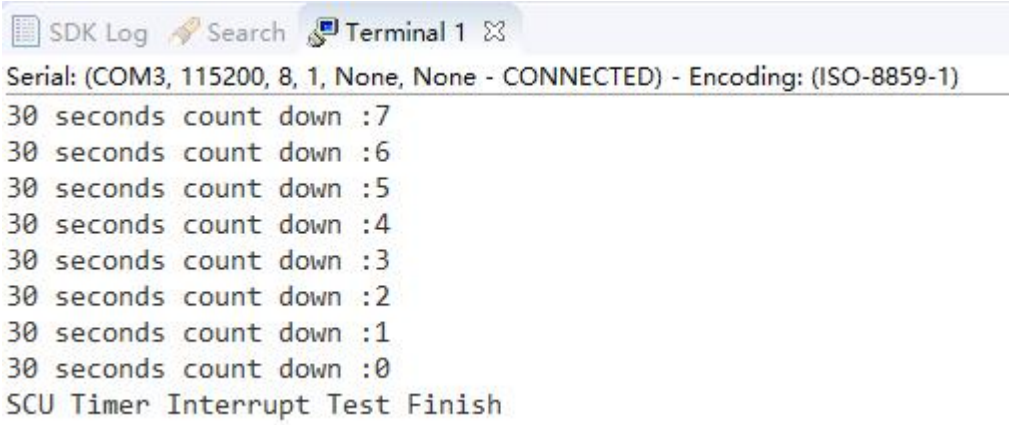
```c
//定时器初始化
int Timer_init(XScuTimer *timer_ptr)
{
    int status;

    XScuTimer_Config *timer_cfg_ptr;
    //私有定时器初始化
    timer_cfg_ptr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
    if (NULL == timer_cfg_ptr)
        return XST_FAILURE;
    status = XScuTimer_CfgInitialize(timer_ptr, timer_cfg_ptr,
            timer_cfg_ptr->BaseAddr);
    if (status != XST_SUCCESS)
        return XST_FAILURE;

    XScuTimer_LoadTimer(timer_ptr, TIMER_LOAD_VALUE); // 设置重装载计数器值
    XScuTimer_EnableAutoReload(timer_ptr);            // 设置为自动装载模式

    return XST_SUCCESS;
}
```

```c
//中断服务函数
void TimerIntrHandler(void *CallBackRef)
{
    XScuTimer *timer_ptr = (XScuTimer *) CallBackRef;

    //倒计时
    second--;
    xil_printf("30 seconds count down : %d \r\n",second);
    //清除定时器中断标志
    XScuTimer_ClearInterruptStatus(timer_ptr);

}

//定时器中断初始化
void TimerIntrInit(XScuGic *intc_ptr,XScuTimer *timer_ptr)
{
    //初始化中断控制器
    XScuGic_Config *intc_cfg_ptr;
    intc_cfg_ptr = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(intc_ptr, intc_cfg_ptr,
            intc_cfg_ptr->CpuBaseAddress);
    //设置并打开中断异常处理功能
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler)XScuGic_InterruptHandler, intc_ptr);
    Xil_ExceptionEnable();

    //关联中断源与中断服务函数
    XScuGic_Connect(intc_ptr, TIMER_IRPT_INTR,
          (Xil_ExceptionHandler)TimerIntrHandler, (void *)timer_ptr);

    XScuGic_Enable(intc_ptr, TIMER_IRPT_INTR); //使能GIC中的定时器中断
    XScuTimer_EnableInterrupt(timer_ptr);      //使能定时器中断
}
```

程序输入完成后按快捷键 Ctrl + S 保存并编译程序。

6.4 下载验证

按照之前的实验连接开发板电源，下载线及串口 USB 线。之后在 SDK 添加 Terminal 窗口并连接串口。将编译好的程序烧录到芯片中。即可在 Terminal 窗口看到打印的 30 秒倒计时的信息，倒计时结束后停止打印。实验现象如下图所示：



6.5 章节小结

在本实验中我们了解了一些关于 ZYNQ 私有定时器的相关知识及定时器中断的用法，除此之外，定时器还有许多其他的功能。本实验的目的是抛砖引玉，让读者初步认识定时器，若想熟练掌握定时器的各项功能还需读者多多查阅资料，编写程序练习使用其他功能。

第 7 章　　自定义 IP 核实验

在之前的实验中，我们使用过 Vivado 软件提供的 IP 核，这些 IP 核种类众多，功能丰富，它能帮助我们很快的实现需要的功能。但即使如此，在某些项目中也无法找到合适的 IP 核，这就需要我们来自定义 IP 核。Vivado 软件提供了创建核封装 IP 向导的方式来自定义 IP 核，可以帮助我们实现定制化系统设计，这些设计还可以在其他系统上实现重用，极大缩短研发周期。在本实验中，我们将讲解自定义 IP 核的使用方法。

本章包括以下内容：

7.1 实验简介

7.2 硬件设计

7.3 软件设计

7.4 下载验证

7.5 章节小结

7.1 实验简介

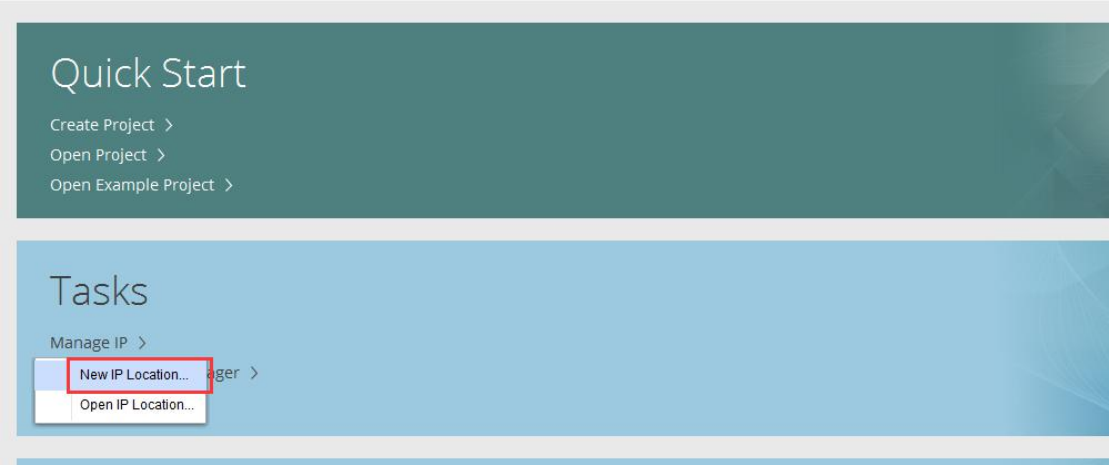
在 zynq 系统的设计中，我们将许多实现特定功能的硬件设计模块封装起来，称之为 IP 核。这种方式极大的提高了设计效率，降低了设计门槛，让设计人员不需要在每次需要某个功能时多次重复相同的工作，或者需要某个功能时去自行研究实现方法，只需要添加实现该功能的 IP 核即可。在添加 IP 核的列表即可看出 xilinx 官方提供了非常丰富的 IP 资源供我们使用。但在实际设计中总会遇到一些特殊的需求，这种时候就需要我们自行设计并封装 IP 核，将它运用到各个系统中。

在本次实验中，我们将创建一个能产生 PWM 波形的 IP 核，在开发板上实现呼吸灯的功能。该 IP 核带有 AXI 总线，可以通过 PS 端访问 IP 核内部寄存器，实现 PS 端对于 IP 核的控制。

7.2 硬件设计

本实验的硬件设计分为两部分，分别是 IP 核的创建和封装以及整个系统的建立。

首先讲解 IP 核的创建与封装。打开 Vivado，在主界面中点击“Task”栏下



的“Manage IP”。在弹出的选项中选择“New IP Location”，如下图所示：

图 7.2.1 新建 IP

在弹出的窗口中点击“Next”，之后配置 IP 保存的路径，其他配置保持默认，之后点击“Finish”完成创建。

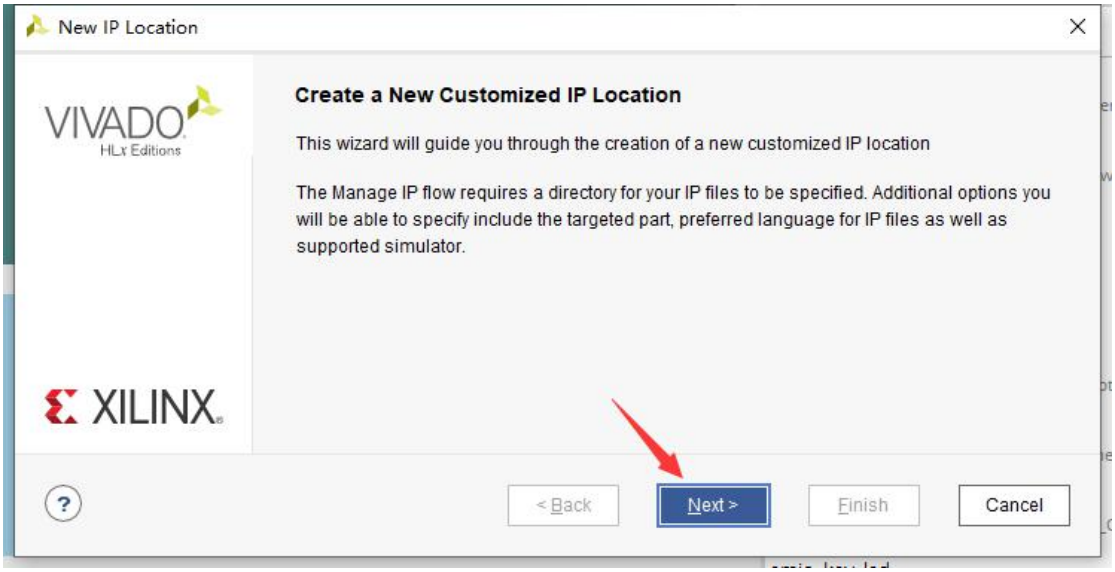


图 7.2.2 创建向导

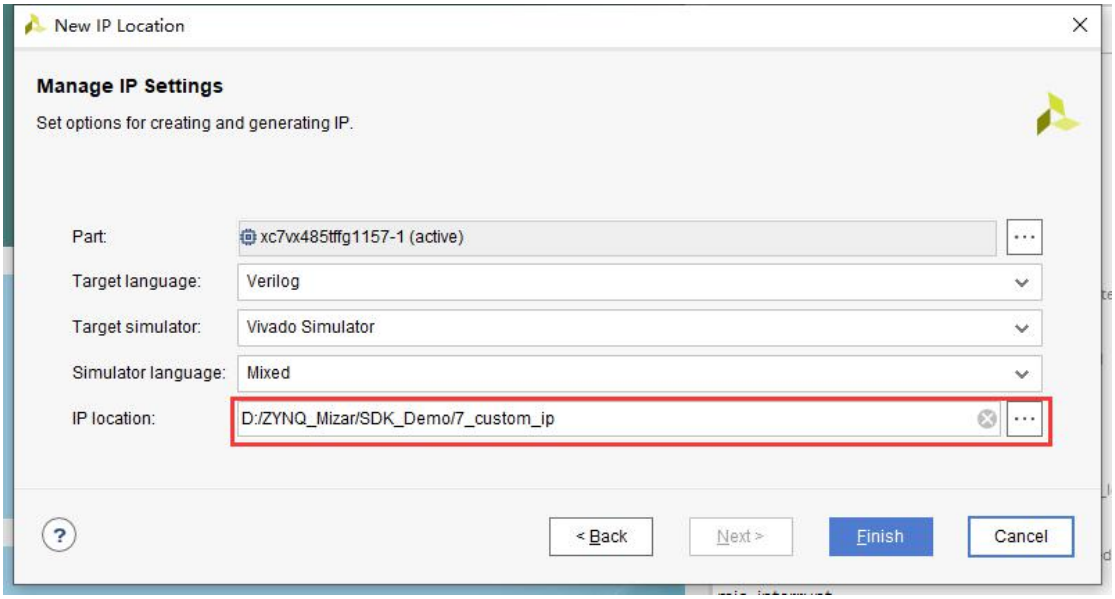


图 7.2.3 相关设置

创建完成后点击菜单栏 Tools->Creat and Package New IP，之后在弹出的对话框中点击“Next”。

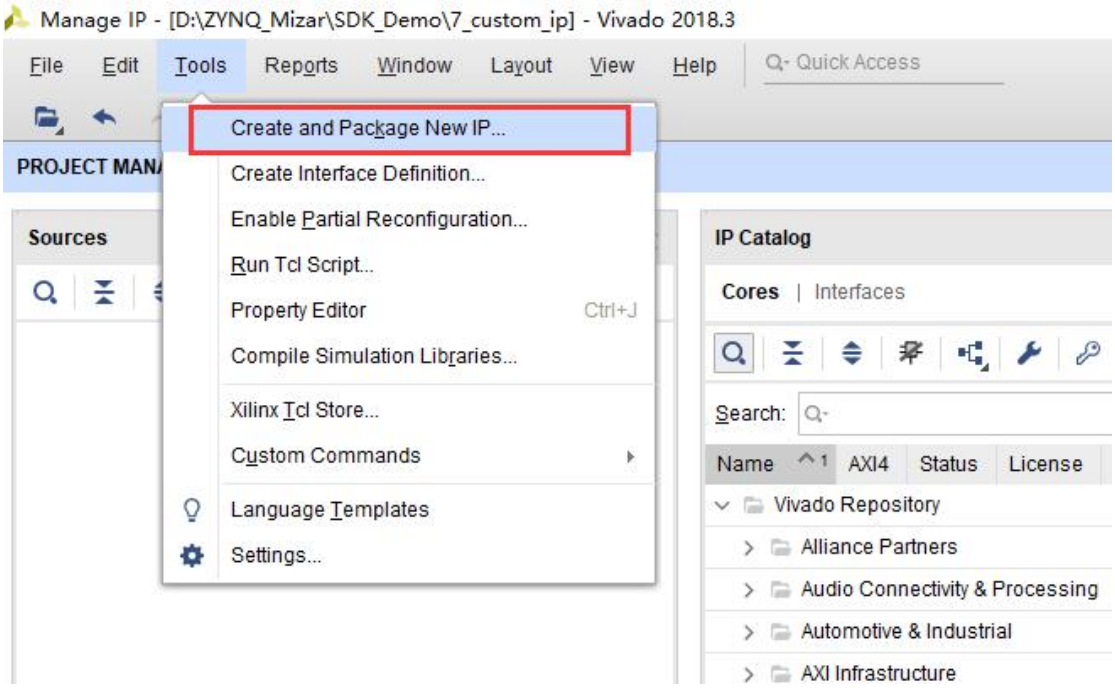


图 7.2.4 创建和封装 IP

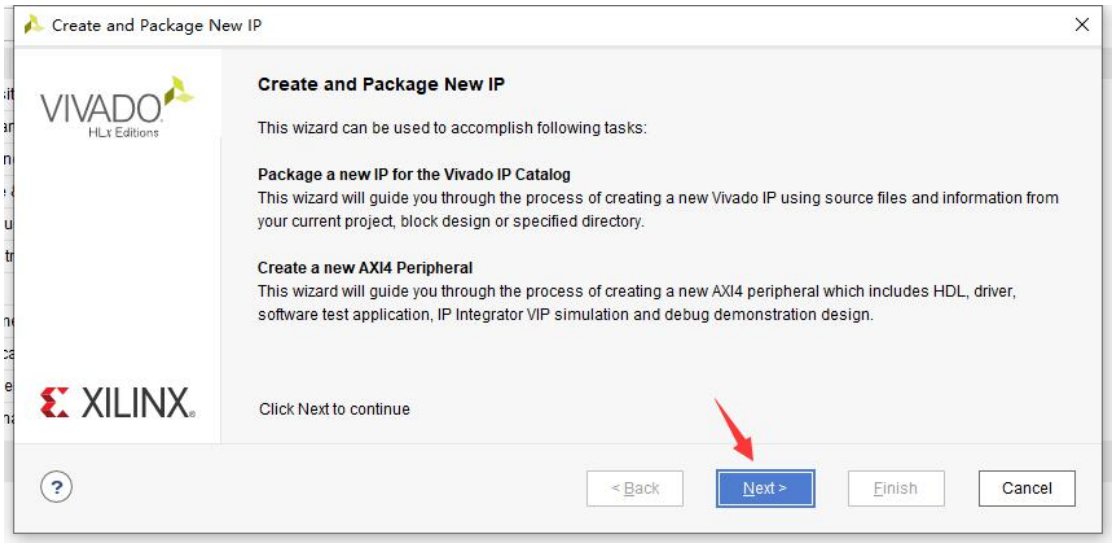
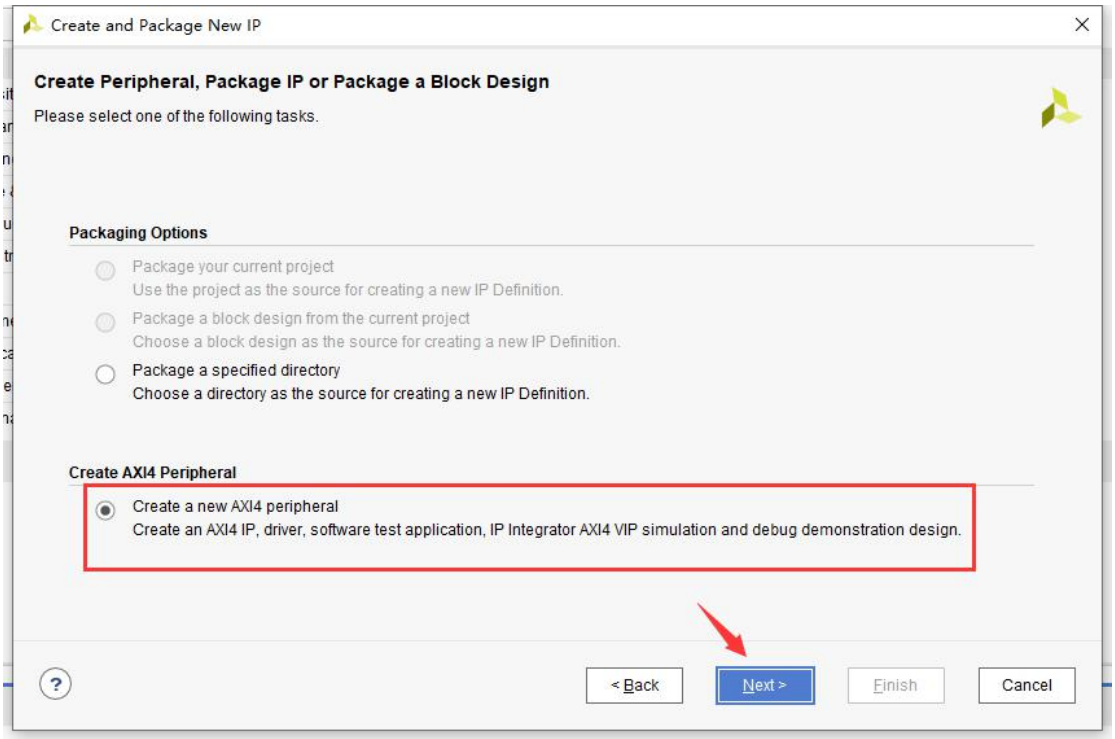


图 7.2.5 创建和封装 IP 向导

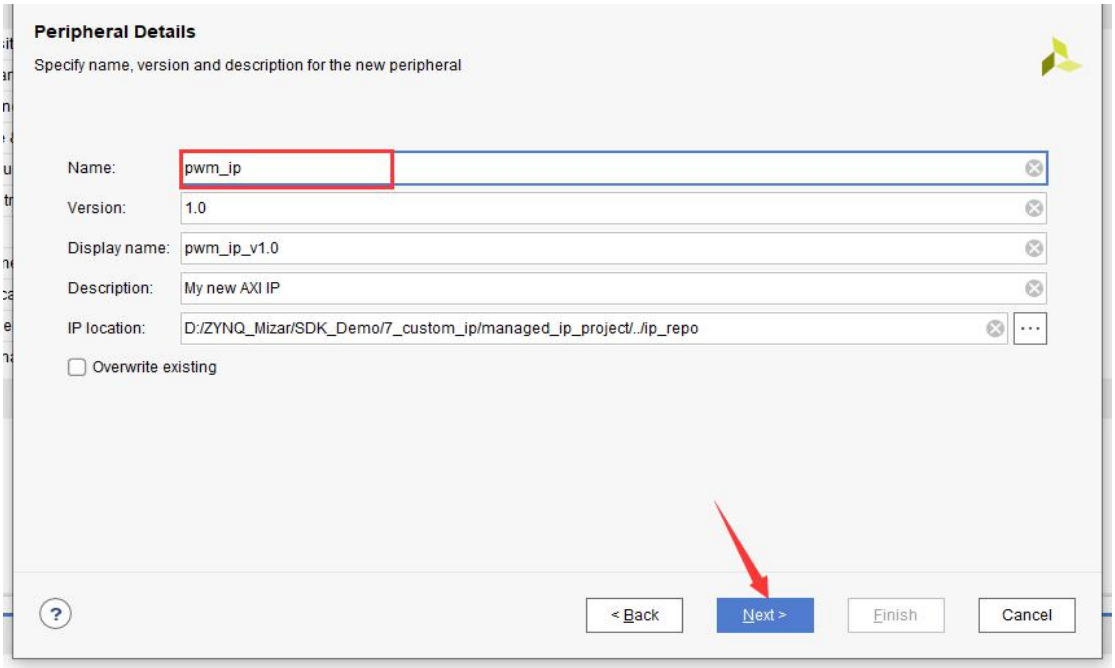
接下来选择封装 IP 或者创建一个带 AXI4 接口的 IP 核，此处我们选择创建



一个带有 AXI4 接口的 IP 核，选中“Create a new AXI4 peripheral”，然后点击“Next”。

图 7.2.6 选中创建 IP 类型

接下来弹出设置页面，设置 IP 核名称、版本号、显示名、描述、保存路径。



此处我们设置 IP 核名称为“pwm_ip”，其他保持默认，然后点击“Next”。

图 7.2.7 设置 IP 参数

接下来弹出 AXI 接口设置，此处保持默认，点击“Next”。

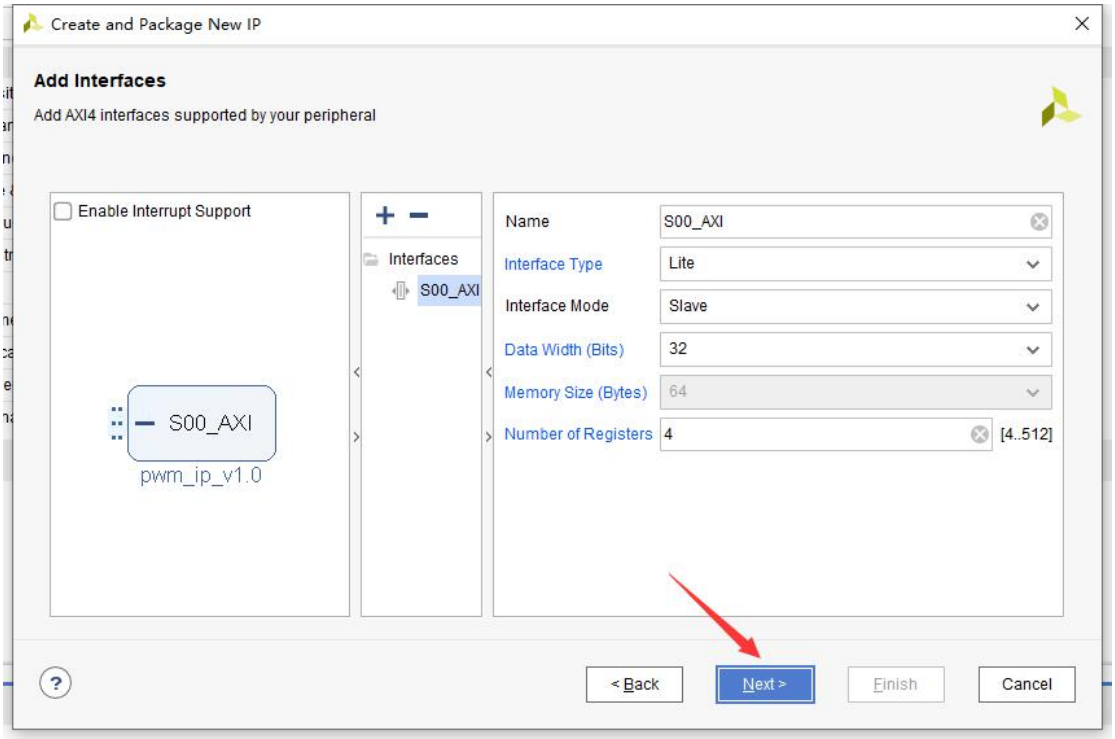
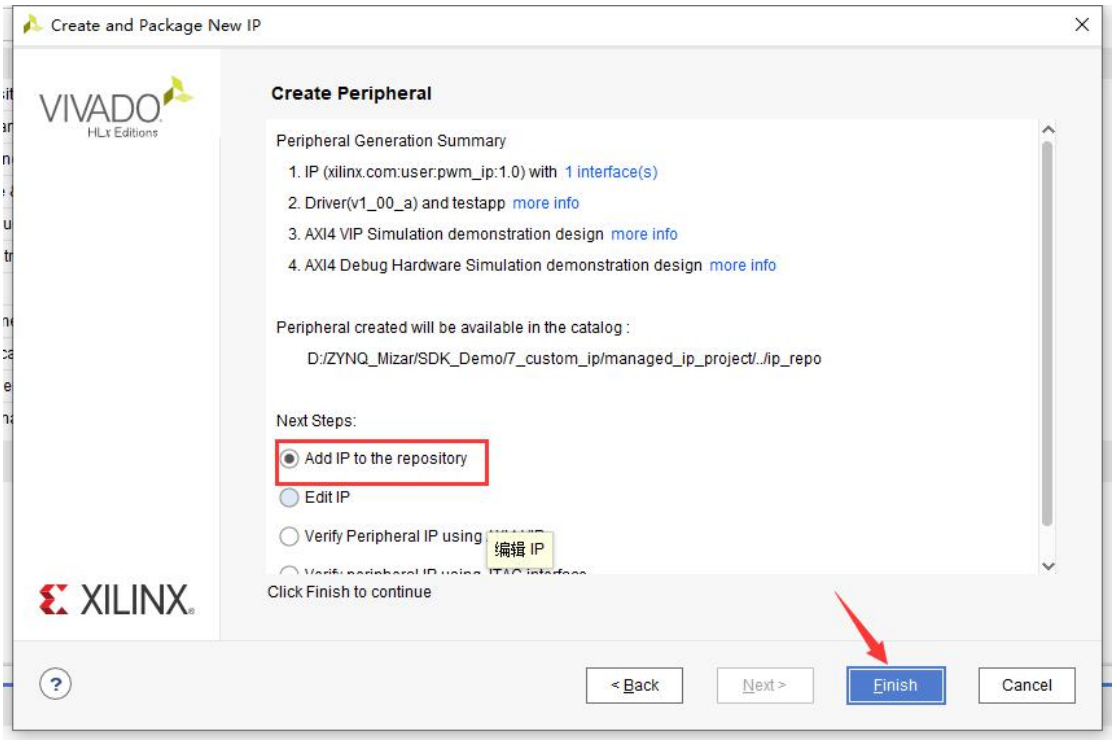


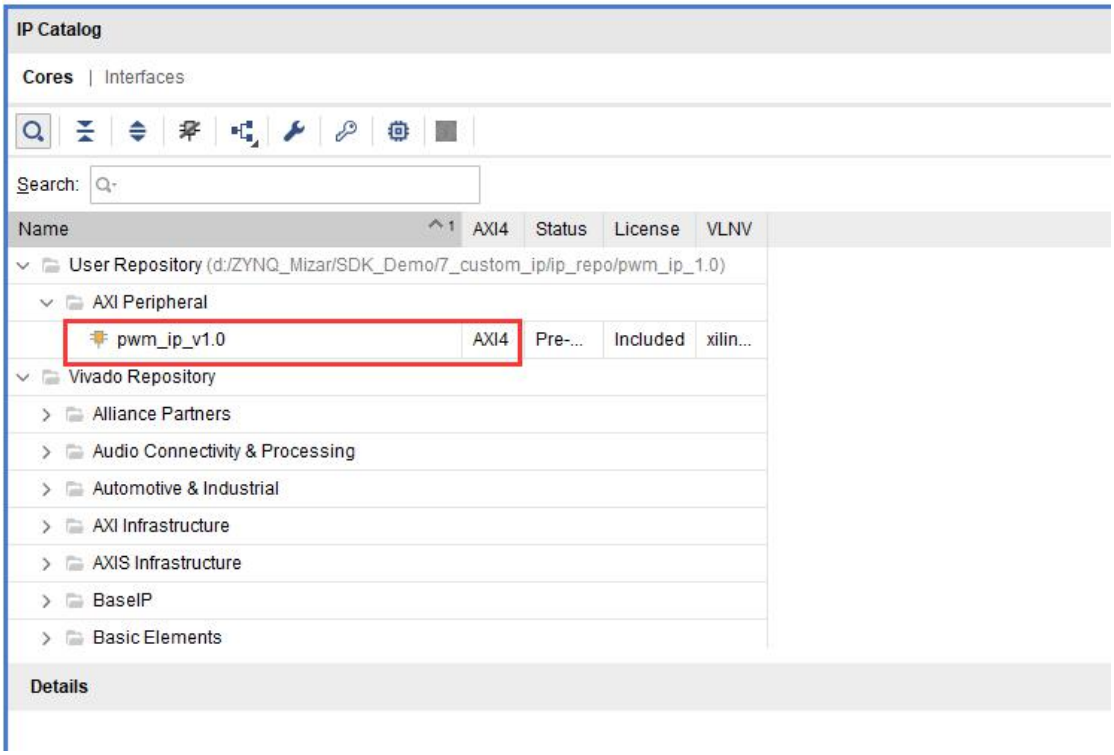
图 7.2.8 AXI 参数设置

接下来弹出封装接口的总结描述和下一步操作选项的界面。这里保持默认，



点击“Finish”按钮完成创建及封装。

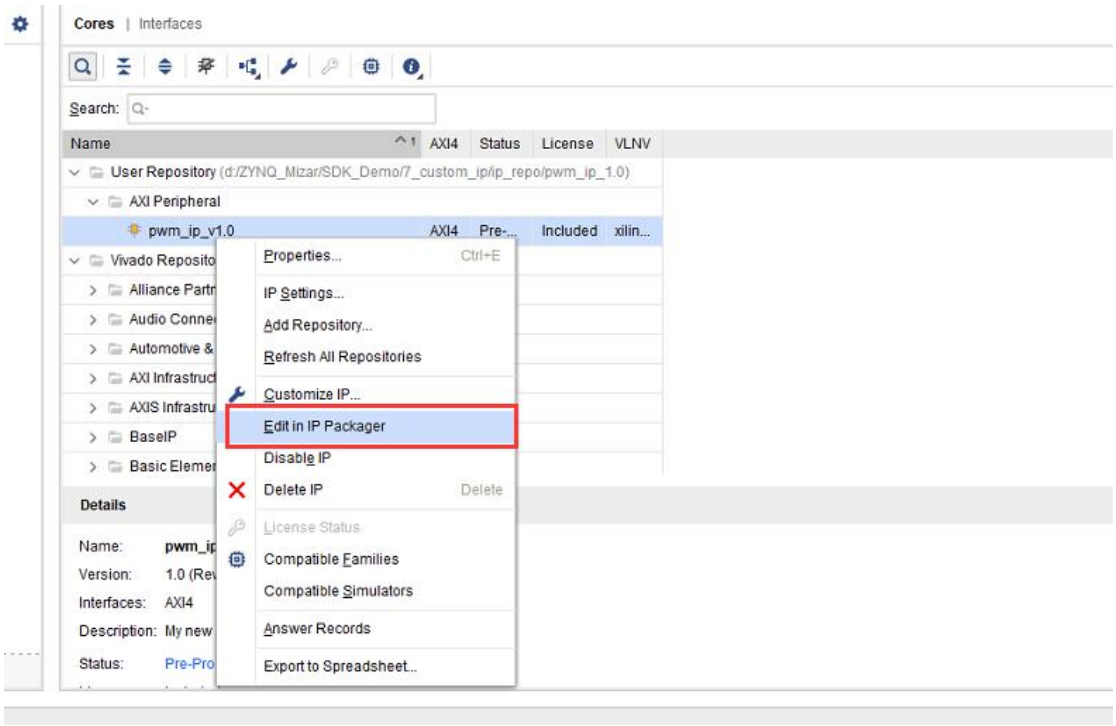
图 7.2.9 创建和封装 IP 核总结界面



在 IP Catalog 界面可以看到创建完成的 IP 核如下图所示：

图 7.2.10 创建完成的 IP 核

之后右键点击该 IP 核，选择“Edit in IP Packager”进入该 IP 核的编辑页面，在弹



出的界面点击“OK”。

图 7.2.11 打开编辑 IP 核界面

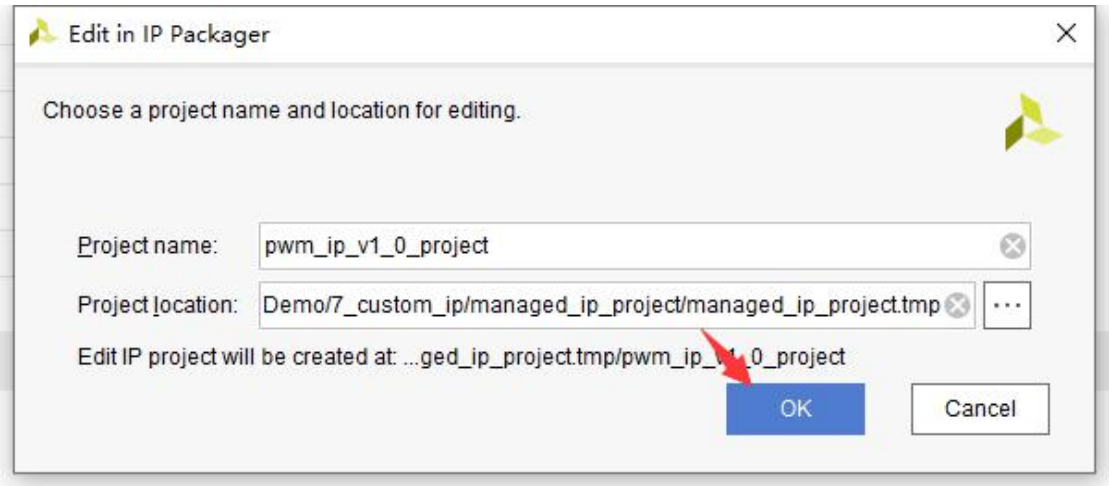
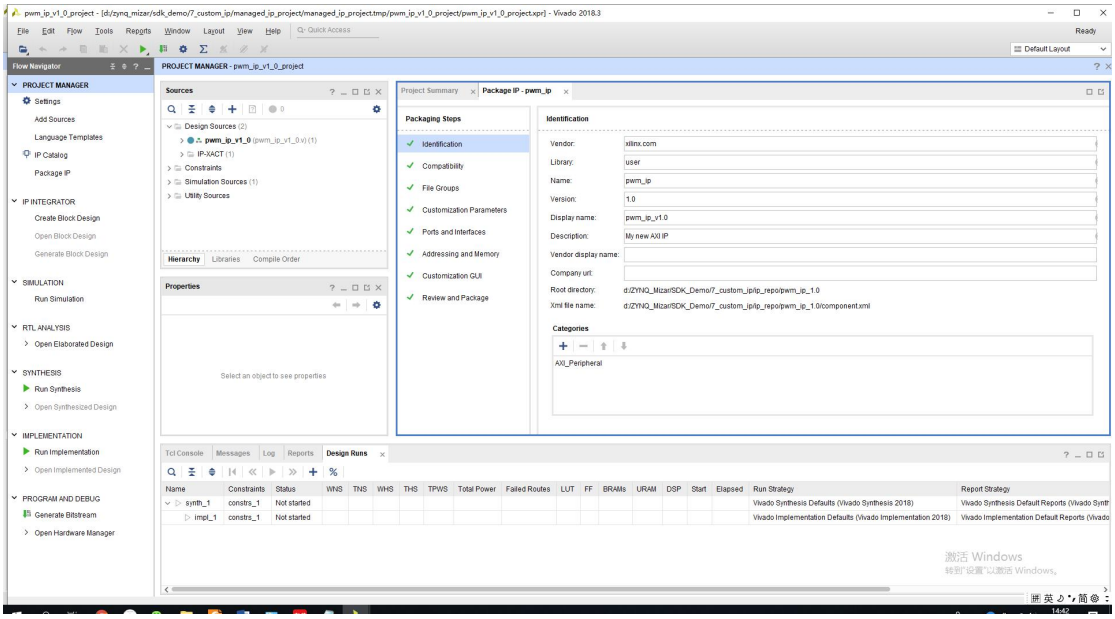
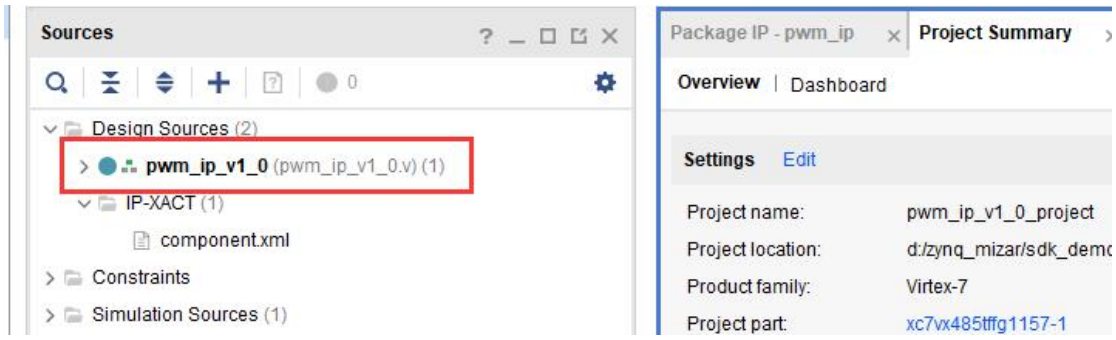


图 7.2.12 设置工程名及路径



点击“OK”后会自动打开一个新的工程，如下图所示：

图 7.2.13 编辑 IP 核工程界面



双击“pwm_ip_v1_0”，如下图所示，进入编辑页面。

图 7.2.14 双击“pwm_ip_v1_0”

```
    output pwm,
```

在第 18 行添加如下代码：

```
1
2   `timescale 1 ns / 1 ps
3
4       module pwm_ip_v1_0 #
5       (
6           // Users to add parameters here
7
8           // User parameters ends
9           // Do not modify the parameters beyond this line
10
11
12          // Parameters of Axi Slave Bus Interface S00_AXI
13          parameter integer C_S00_AXI_DATA_WIDTH    = 32,
14          parameter integer C_S00_AXI_ADDR_WIDTH    = 4
15      )
16      (
17          // Users to add ports here
18          output pwm,
19          // User ports ends
20          // Do not modify the ports beyond this line
21
```

添加完成的代码如下图所示：

图 7.2.15 添加相关代码

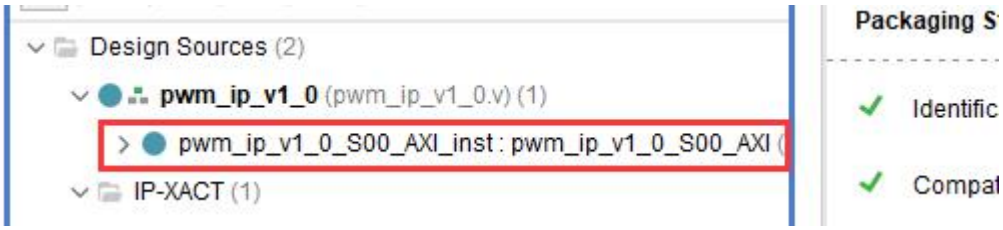
```
.pwm(pwm),
```

在第 51 行添加如下代码：

```
46  // Instantiation of Axi Bus Interface S00_AXI
47      pwm_ip_v1_0_S00_AXI # (
48          .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
49          .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
50      ) pwm_ip_v1_0_S00_AXI_inst (
51          .pwm(pwm),
52          .S_AXI_ACLK(s00_axi_aclk),
53          .S_AXI_ARESETN(s00_axi_aresetn),
54          .S_AXI_AWADDR(s00_axi_awaddr),
55          .S_AXI_AWPROT(s00_axi_awprot),
56          .S_AXI_AWVALID(s00_axi_awvalid),
57          .S_AXI_AWREADY(s00_axi_awready),
```

代码添加完成后，快捷键 Ctrl + S 保存。如下图所示：

图 7.2.16 添加端口例化



双击 pwm_ip_v1_0 下的 pwm_ip_v1_0_S00_AXI_inst，如下图所示：

图 7.2.17 双击 pwm_ip_v1_0_S00_AXI_inst

在第 8 行添加如下代码：

```
18      output pwm,
```

添加完成后的代码如图所示：

d:/ZYNQ_Mizar/SDK_Demo/7_custom_ip/ip_repo/pwm_ip_1.0/hdl/pwm_ip_v1_0_S00_AXI.v

```
1
2   `timescale 1 ns / 1 ps
3
4       module pwm_ip_v1_0_S00_AXI #
5       (
6           // Users to add parameters here
7
8           // User parameters ends
9           // Do not modify the parameters beyond this line
10
11          // Width of S_AXI data bus
12          parameter integer C_S_AXI_DATA_WIDTH = 32,
13          // Width of S_AXI address bus
14          parameter integer C_S_AXI_ADDR_WIDTH = 4
15      )
16      (
17          // Users to add ports here
18          output pwm,
19          // User ports ends
20          // Do not modify the ports beyond this line
```

图 7.2.18 添加代码

```
401     pwm_ip u_pwm_ip(
402         .sys_clk (S_AXI_ACLK),
403         .sys_rst_n (S_AXI_ARESETN),
404         .set_freq_step (slv_reg1[9:0]),
405         .pwm (pwm)
406     );
```

之后在代码的第 401 行例化 pwm_Ip.v 文件，添加如下代码：

```
396            end
397        end
398    end
399
400    // Add user logic here
401    pwm_ip u_pwm_ip(
402        .sys_clk (S_AXI_ACLK),
403        .sys_rst_n (S_AXI_ARESETN),
404        .set_freq_step (slv_reg1[9:0]),
405        .pwm (pwm)
406    );
407    // User logic ends
408
409    endmodule
410
```

添加完成后按下快捷键 Ctrl + S 保存，添加完成的代码如下图所示：

图 7.2.19 例化 pwm_ip.v

之后添加 pwm_ip.v 文件，右键点击“Design Source”，选择“Add Source”，在弹出的界面选择“Add or Create design source”，点击“Next”。

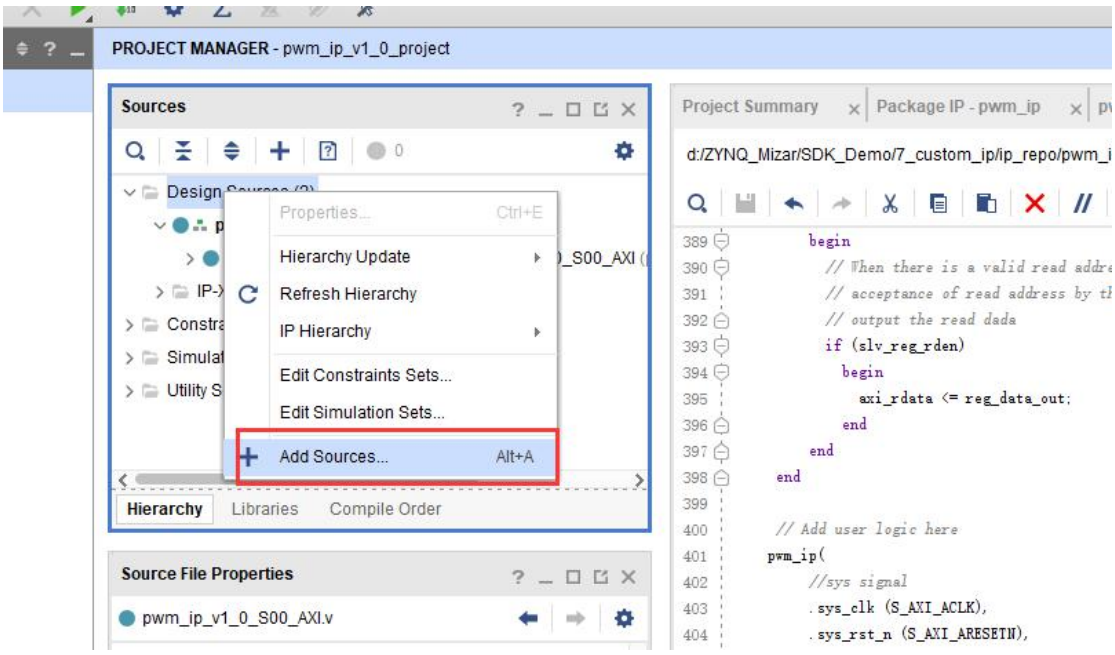


图 7.2.20 添加 pwm_ip.v

图 7.2.21 选择添加设计源文件

点击“Create File”，创建新文件，在弹出的界面输入文件名“pwm_ip”，设置保存路径，如图 7.2.23 所示。

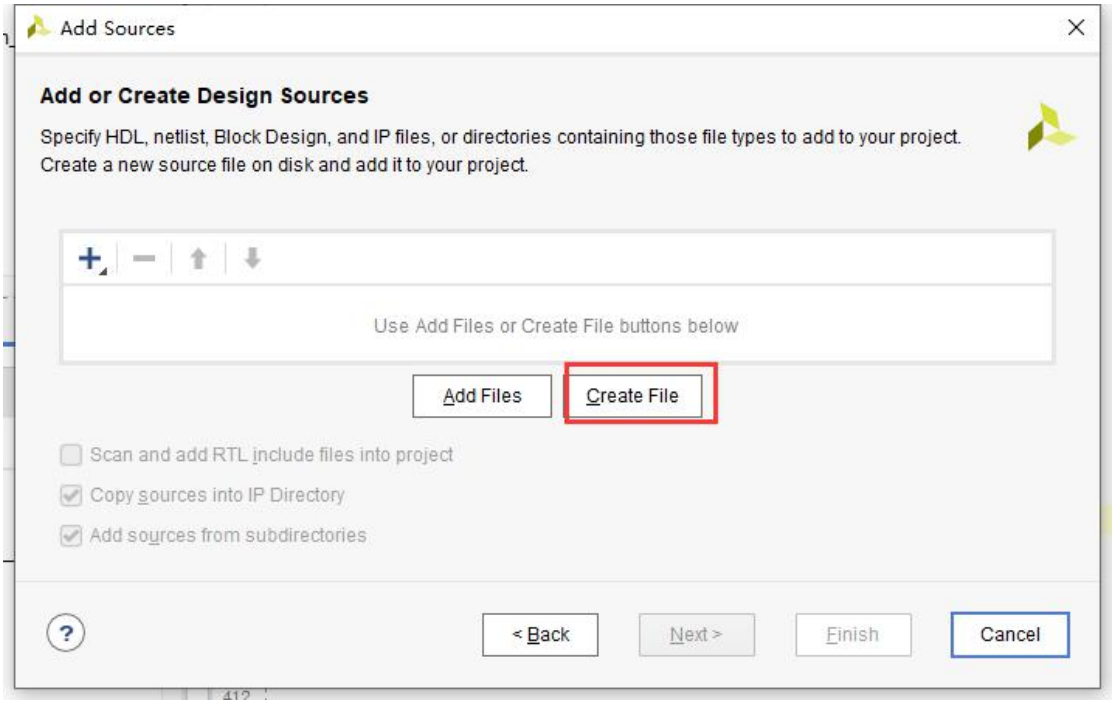


图 7.2.22 创建新文件

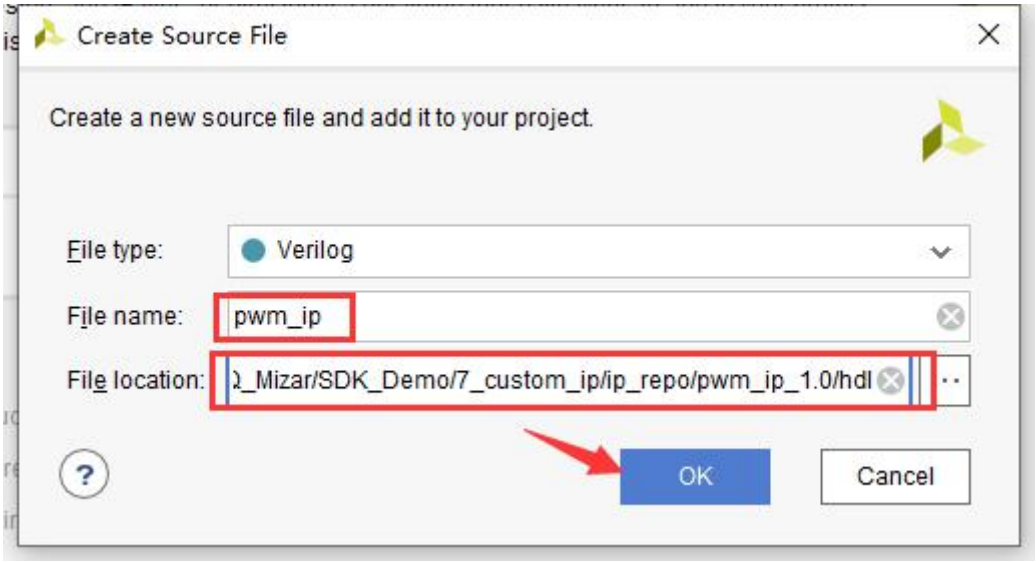


图 7.2.23 设置文件名及保存路径

创建完成的界面如下图所示，点击“Finish”完成创建。

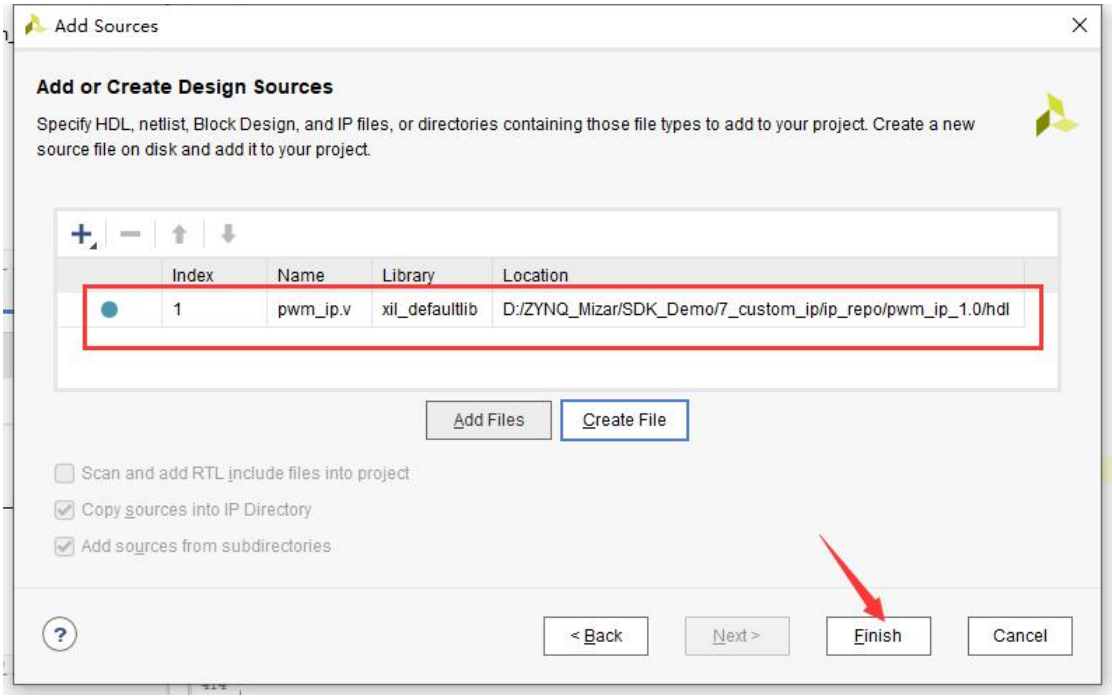
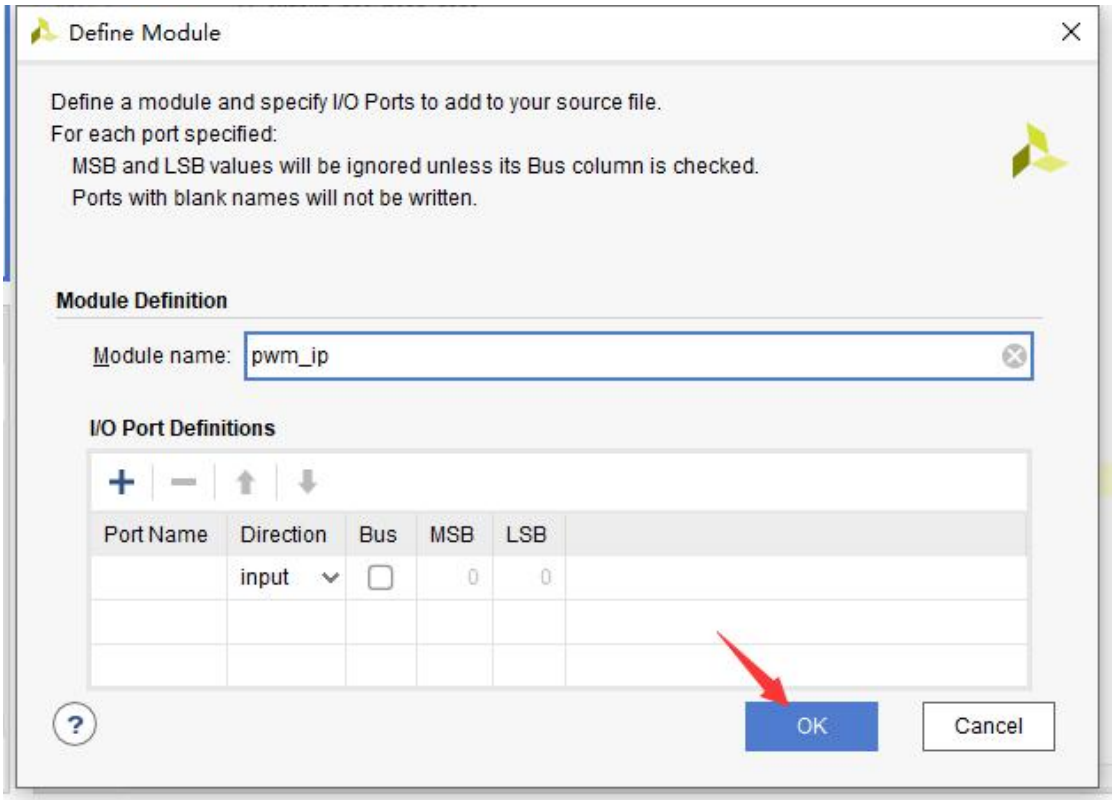


图 7.2.24 完成设计文件创建

在接下来弹出的模块定义界面点击“OK”，在弹出的确认对话框中点击“yes”。



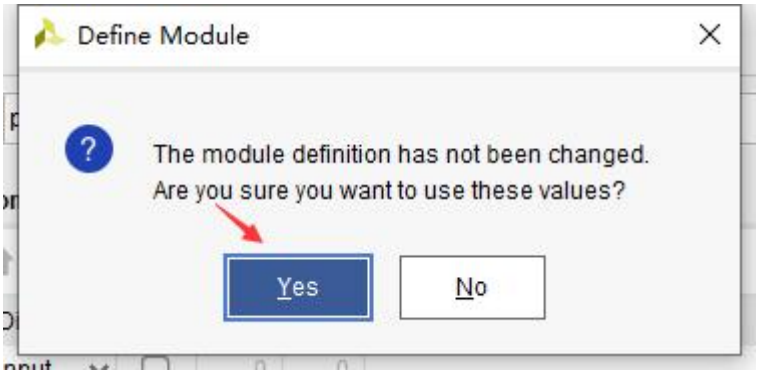
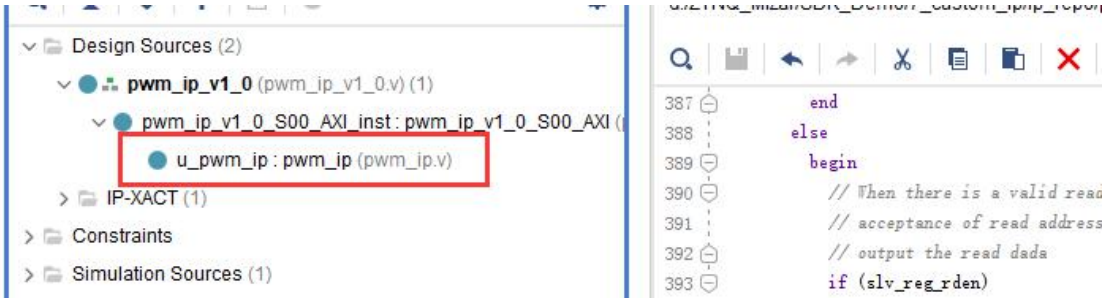


图 7.2.25 模块定义界面

图 7.2.26 确认对话框

接下来编辑相关代码。双击 u_pwm_ip:pwm_ip(pwm_ip.v)，打开编辑窗口，



如下图所示：

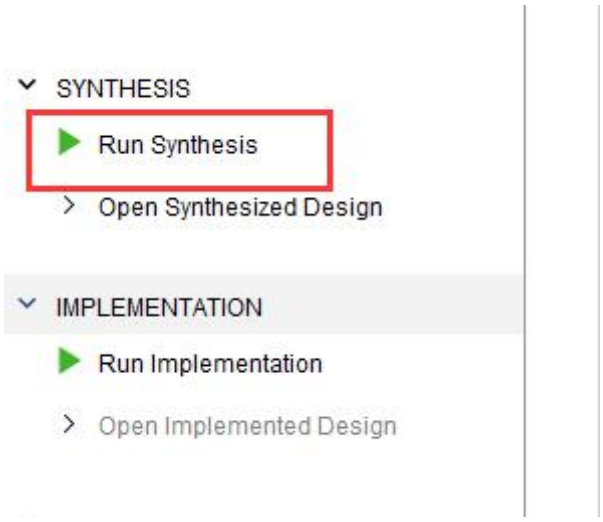
图 7.2.27 双击打开编辑窗口

在编辑框中输入以下代码：

```
22
23  module pwm_ip(
24      input sys_clk ,
25      input sys_rst_n ,
26      input [9:0] set_freq_step ,
27      output pwm
28  );
29   //*****************************************************
30   //** main code
31   //*****************************************************
32   //reg define
33  reg [15:0] period_cnt ;
34  reg [15:0] duty_cycle ;
35  reg inc_flag ;
36  //wire define
37  wire led_t ;
38
39  assign led_t = ( period_cnt <= duty_cycle ) ? 1'b1 : 1'b0 ;
40  assign pwm = led_t;
41  always @ (posedge sys_clk) begin
42      if (!sys_rst_n)
43          period_cnt <= 16'd0;
44      else if( period_cnt == 16'd50_0000 )
45          period_cnt <= 16'd0;
46      else
47          period_cnt <= period_cnt + 16'd1;
```

```
48      end
49  always @(posedge sys_clk) begin
50      if (sys_rst_n == 1'b0) begin
51          duty_cycle <= 16'd0;
52          inc_flag <= 1'b0;
53      end
54      else if( period_cnt == 16'd50_0000 ) begin
55          if( inc_flag ) begin
56              if( duty_cycle == 16'd0 )
57                  inc_flag <= 1'b0;
58              else if(duty_cycle < 10'd100)
59                  duty_cycle <= 16'd0;
60          else
61              duty_cycle <= duty_cycle - set_freq_step;
62          end
63      else begin
64          if( duty_cycle >= 16'd50_0000 )
65              inc_flag <= 1'b1;
66          else
67              duty_cycle <= duty_cycle + set_freq_step;
68          end
69      end
70      else
71          duty_cycle <= duty_cycle ;
72      end
73  endmodule
```

编辑完成后快捷键 Ctrl + S 保存，之后点击点击导航栏 SYNTHESIS->Run



Synthesis，在弹出的对话框中点击“OK”，开始编译代码。

图 7.2.28 编译代码

编译完成后双击 IP-XACT 下的 component.xml 打开封装 IP 页面。

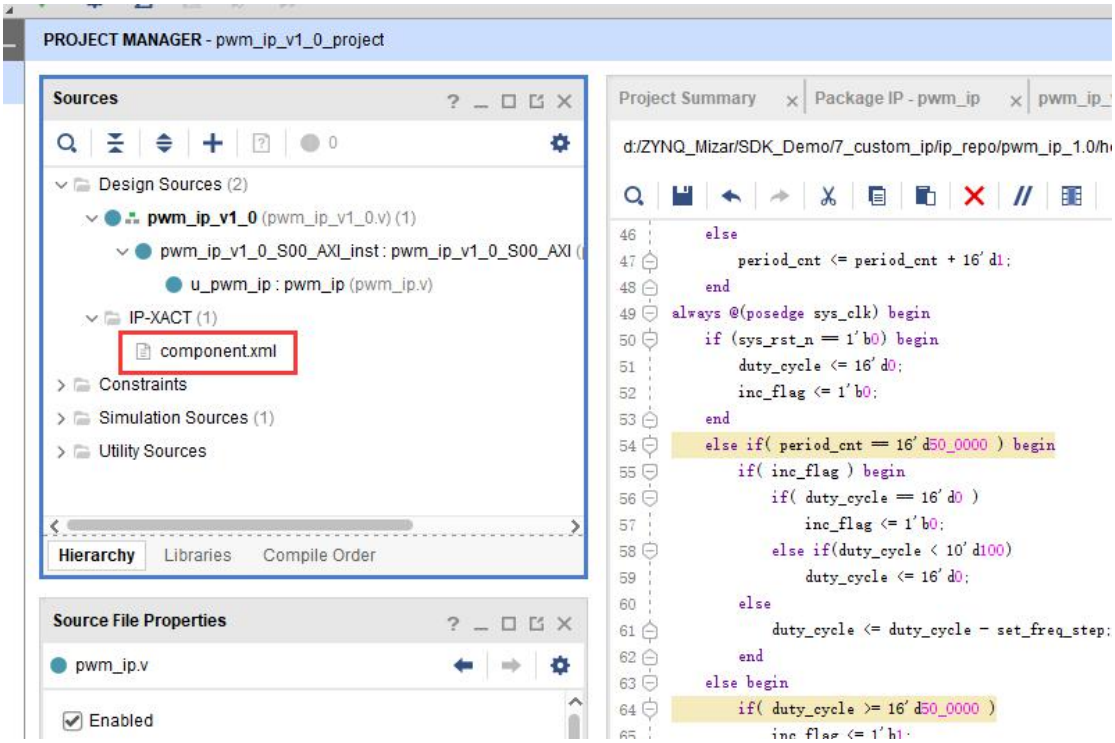


图 7.2.29 打开 IP 封装页面

Identification 这一栏设置 IP 分类，此处保持默认，需要修改的化点击箭头所指“+”即可。

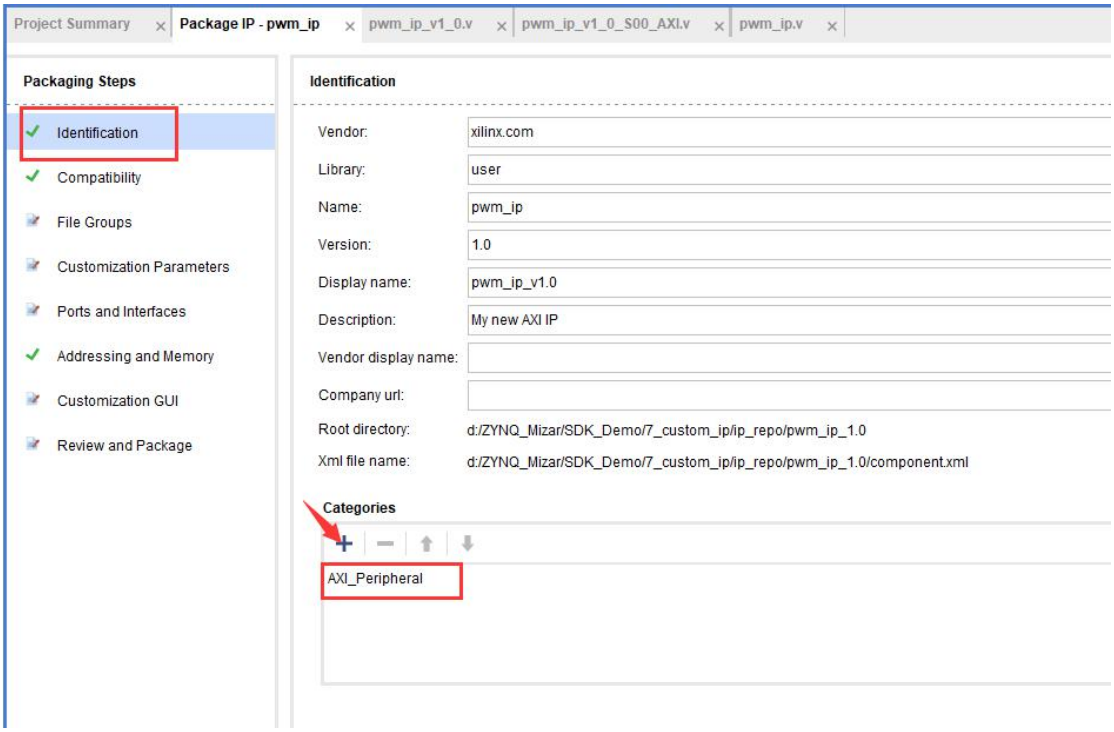
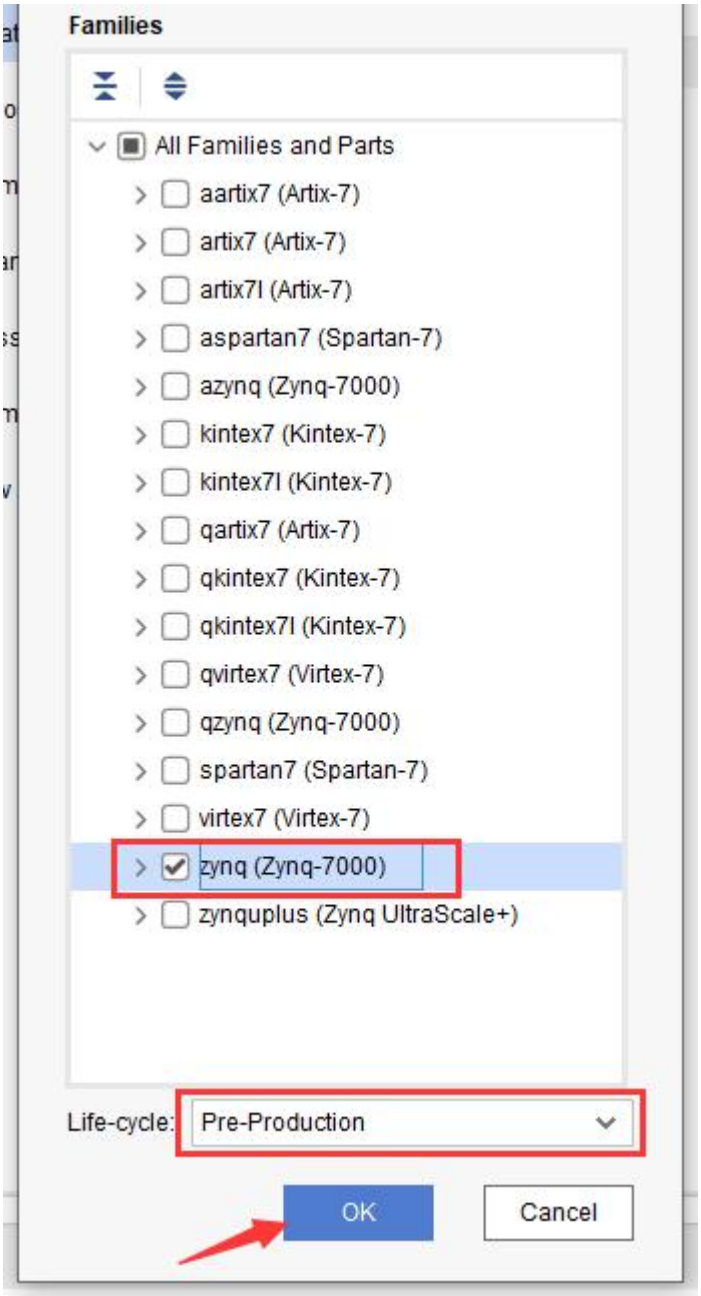


图 7.2.30 修改 IP 核分类

Compatibility 一栏修改支持器件类型。点击箭头所指“+”，选择“Add Family Explicitly…”，如下图所示：

图 7.2.31 Compatibility 栏修改支持器件类型

此处勾选“zynq(zynq-7000)”，表示该 IP 核支持 zynq 器件。而 Life-cycle 表



明该 IP 核的生命周期，这里选择“Pre-Production”。

图 7.2.32 设置支持器件

点击 File Groups，然后点击界面上的“Merge Changes from Gile Groups

Wizard”，如下图所示：

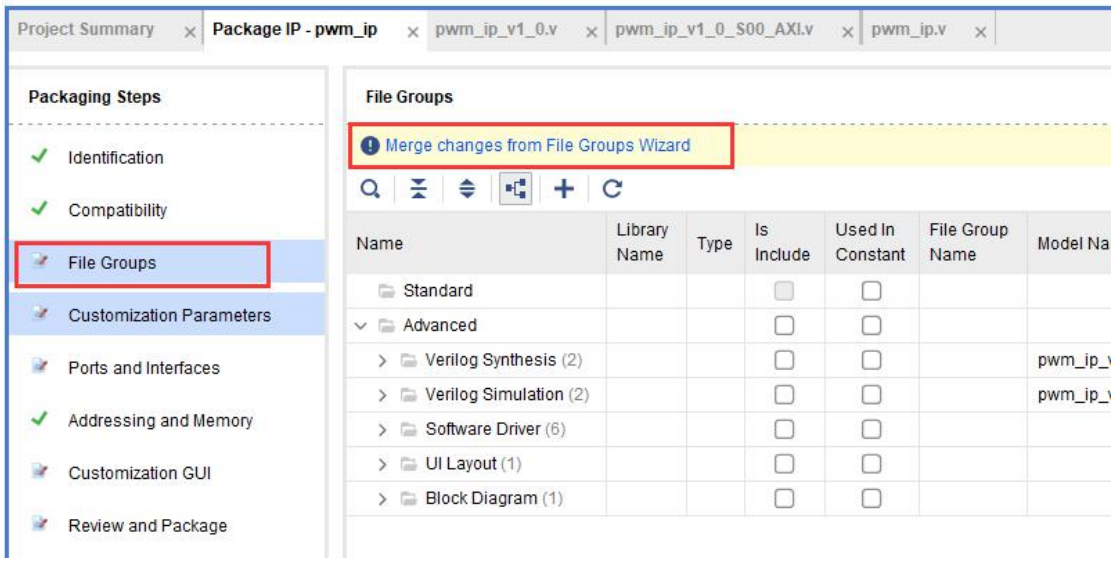


图 7.2.33 File Groups 界面

点击 Customization Parameters，点击界面上的“Merge Changes from Customization Parameters Wizard”，如下图所示：

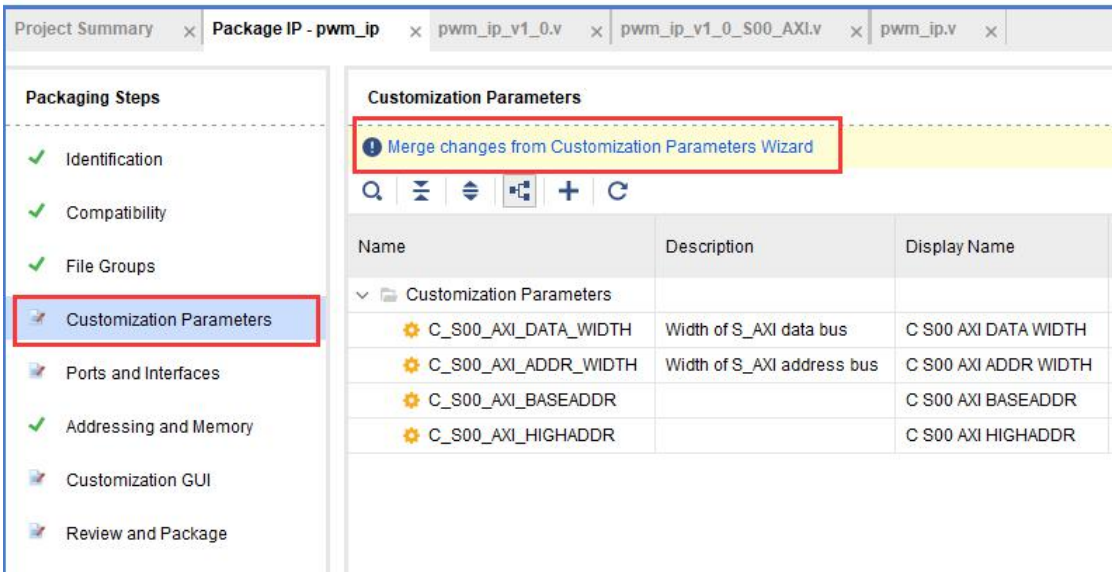


图 7.2.34 Customization Parameters 界面

点击“Review and Package”，然后点击“IP has been modified”更新总结界面，最后点击“Re-Package IP”，如下图所示：

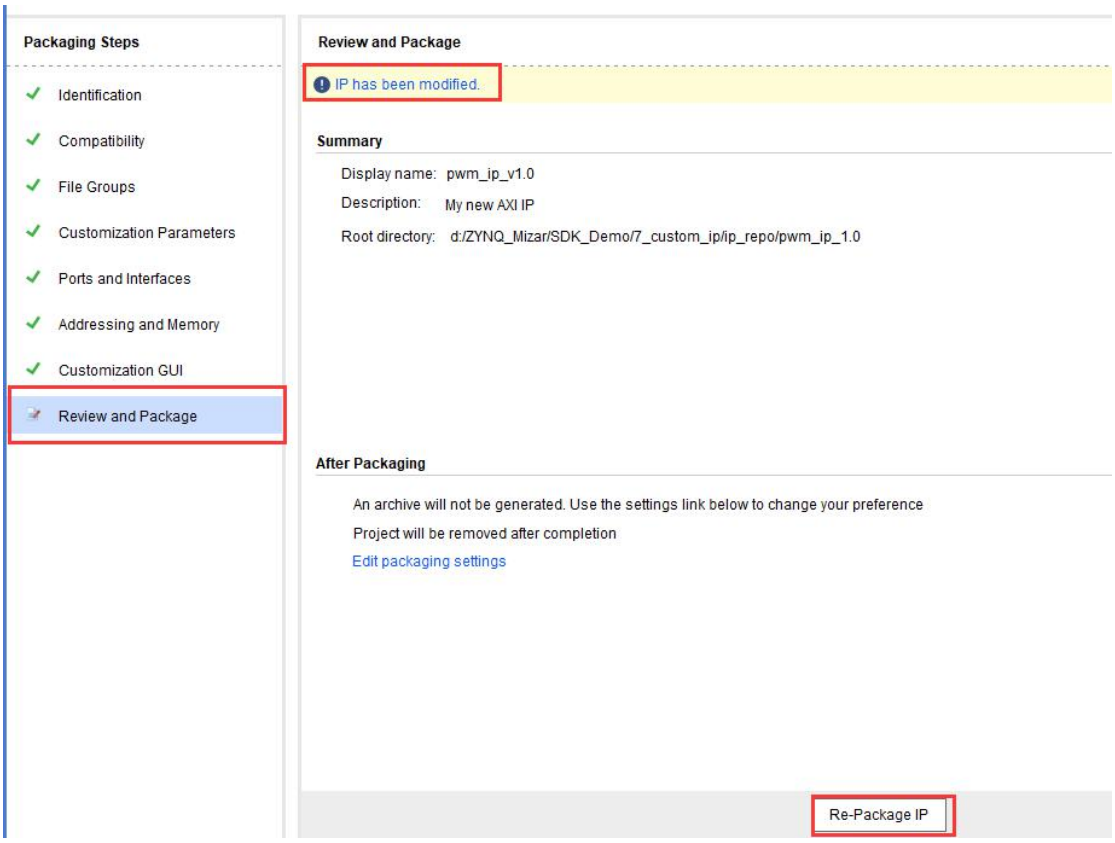


图 7.2.35 Review and Package 界面

封装完成后弹出对话框询问是否关闭工程，点击“yes”关闭工程，如下图所示：

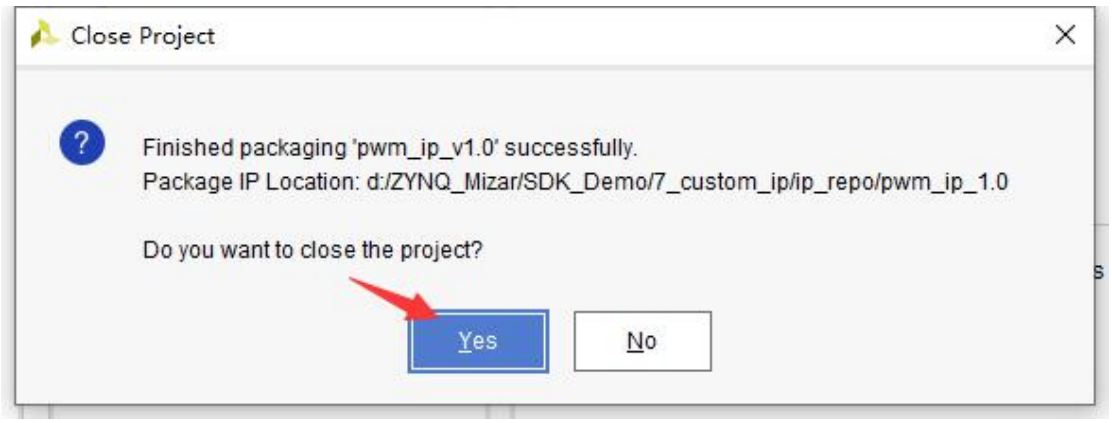


图 7.2.36 询问是否关闭工程

到此我们就完成了该 IP 核的封装，接下来开始系统硬件的设计。关闭之前设计 IP 的工程，在 Vivado 新建一个工程命名为“pwm_led”。创建完成后点击 Create Block Design 创建新的设计。之后将自定义的 IP 添加到工程的 IP 库中，点击菜单

栏“Tools”，选择“Settings”，如下图所示：

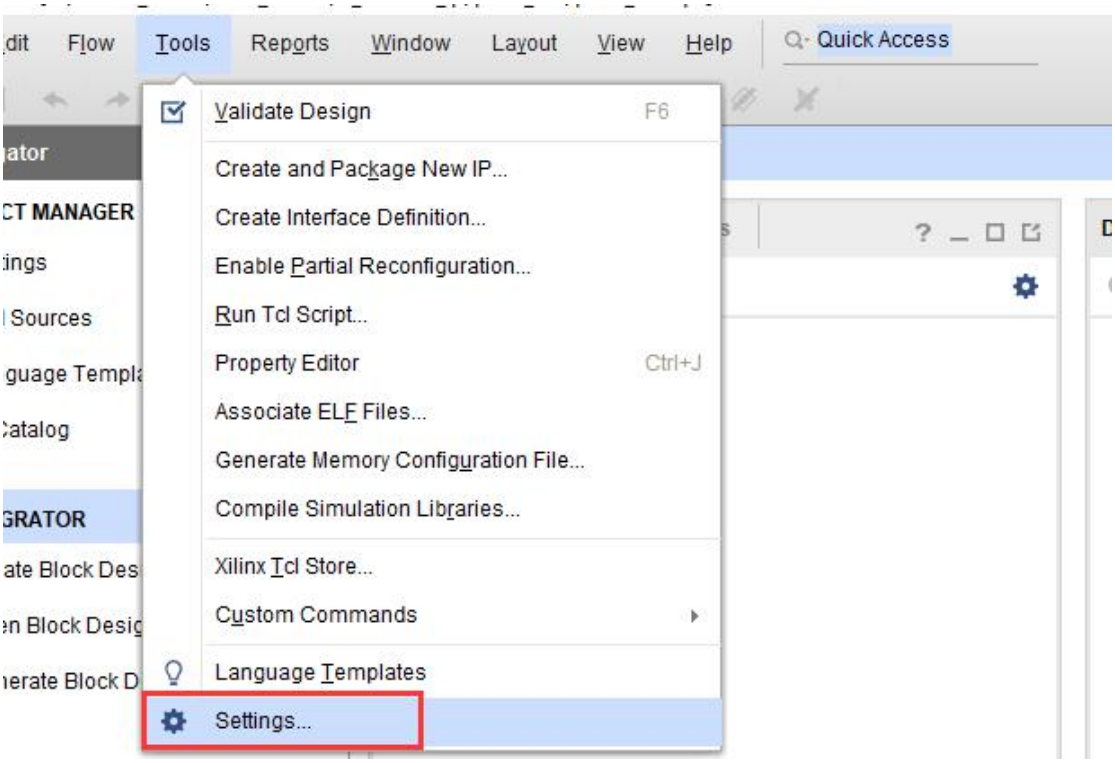


图 7.2.37 打开设置界面。

点击“IP”一栏下的“Repository”，然后点击“+”来添加自定义的 IP 核，如下图所示：

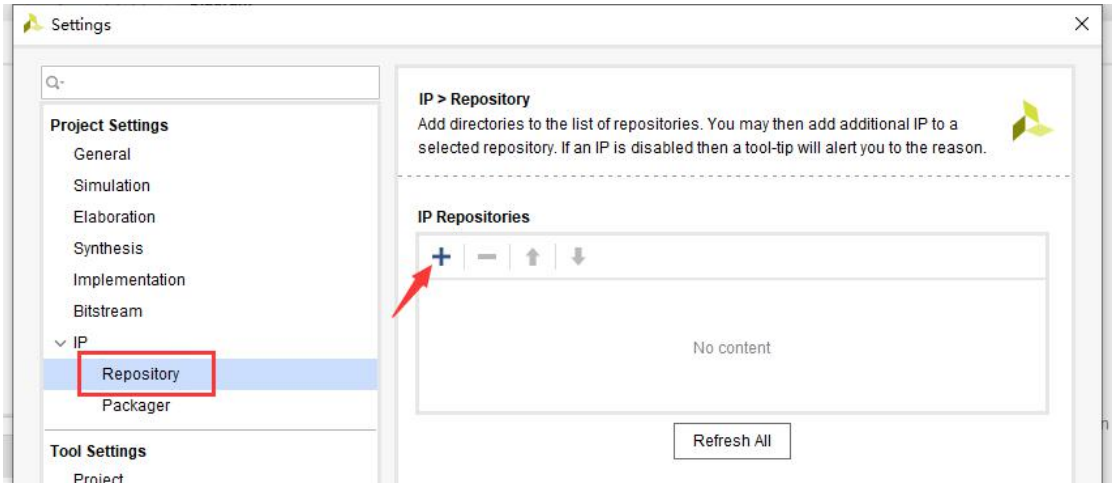
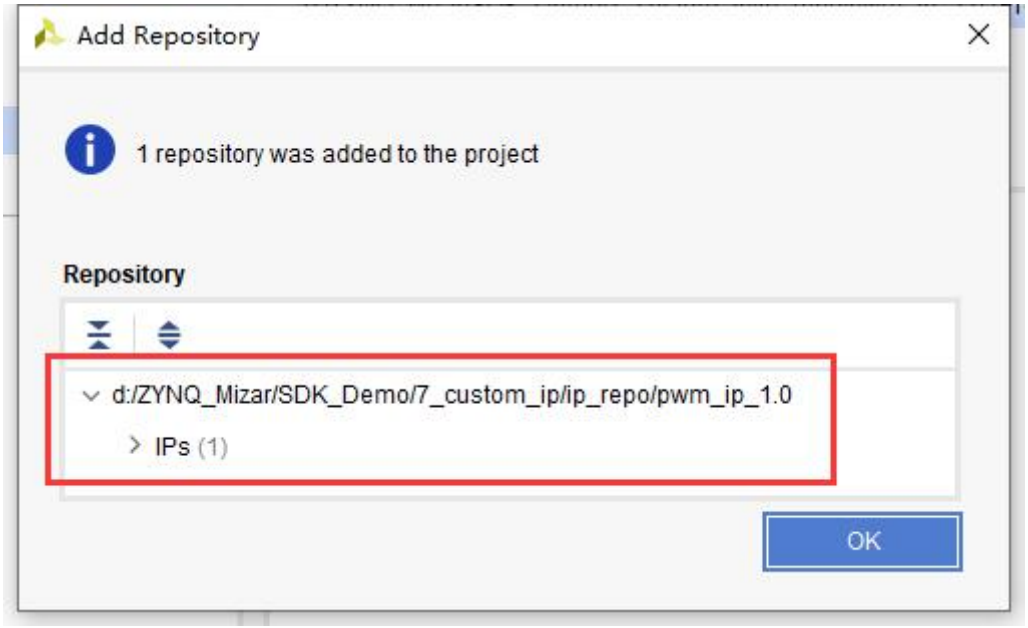


图 7.2.38 IP 库添加

在弹出的窗口中添加自定义 IP 核的路径，选择../7_custom_ip/ip_repo/pwm_ip_1.0，点击“Select”，在弹出的界面可以看到识别的 IP 核，点击“OK”，



添加此 IP 核，如下图所示：

图 7.2.39 添加 IP 核

之后点击“OK”，完成 IP 核的添加，如下图所示：

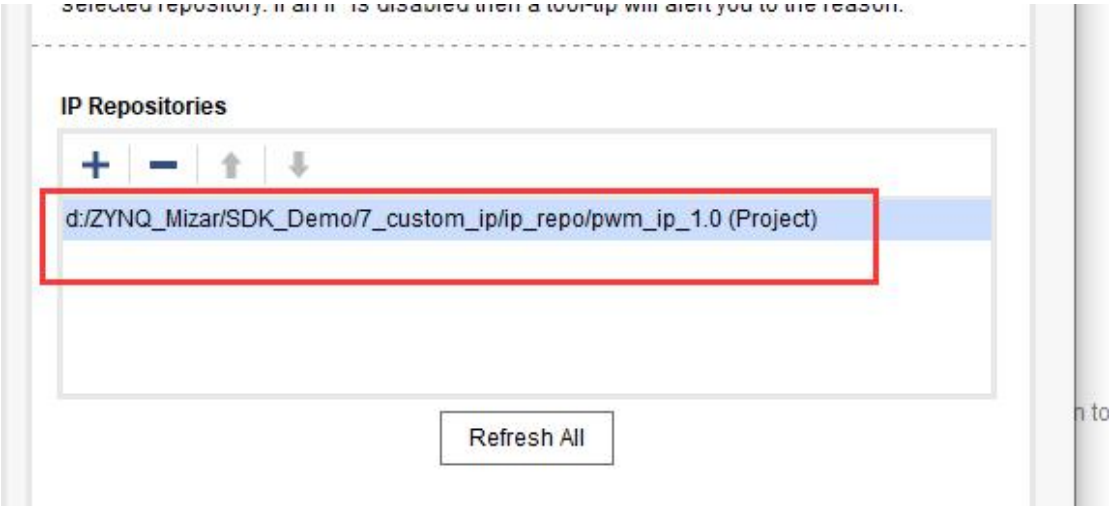


图 7.2.40 添加 IP 核完成

之后在 Diagram 界面添加 ZYNQ7 Processing System 处理系统 IP，双击 IP 进

入配置页面。如之前实验配置 UART 和 DDR3 控制器，其他保持默认，点击“OK”完成配置。配置完成后，点击“Run Block Automation”，如下图所示：

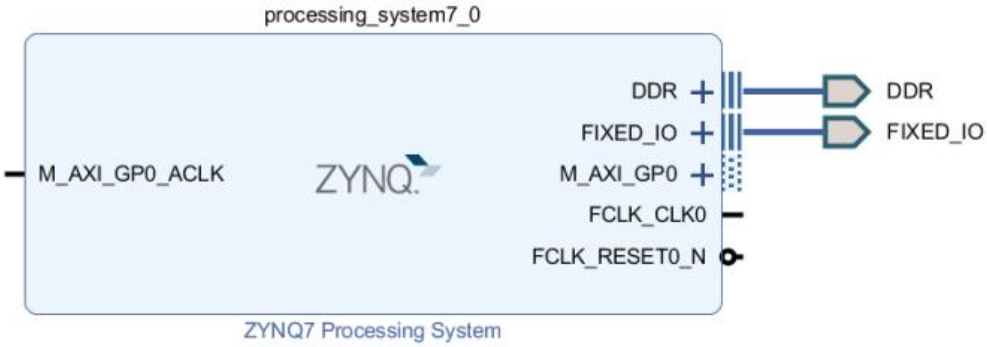
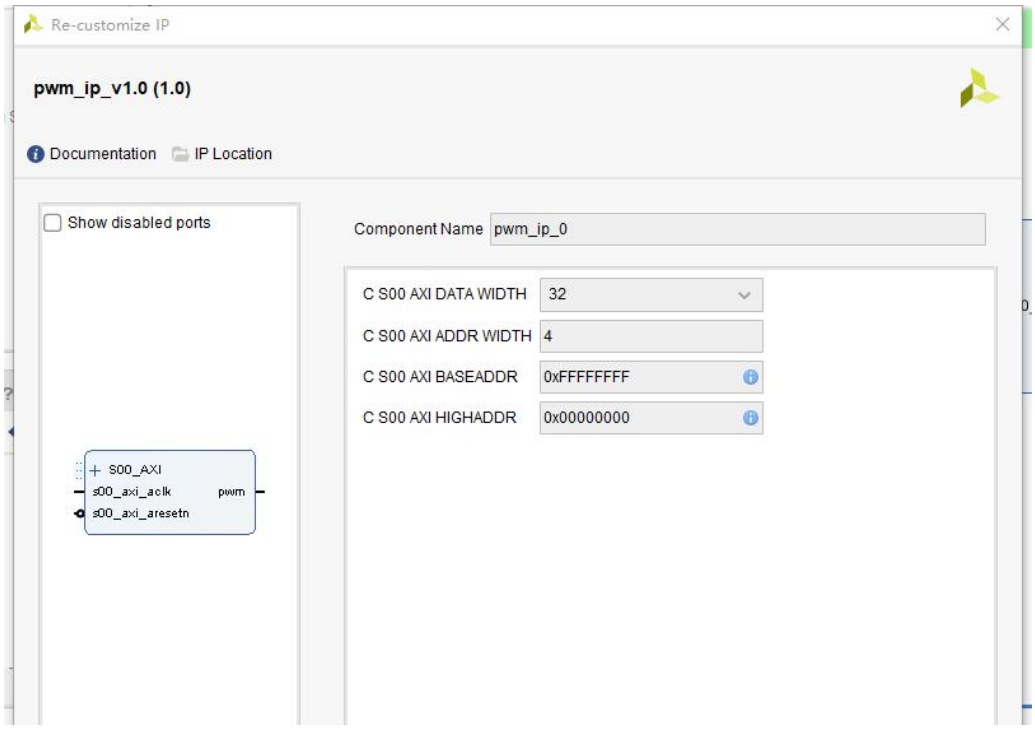


图 7.2.41 zynq 配置完成

接下来添加 PWM IP 核，点击“+”图标，在搜索框中输入“pwm”，如下图所示：

图 7.2.42 添加 PWM IP 核

双击添加此 IP 核到设计中。添加完成后双击此 IP 核可以配置参数，在此实



验中参数无需配置，保持默认即可。

图 7.2.43 配置 IP 核参数

接下来点击“Run Connection Automation”来自动连线，左侧全选所有连接，点击“OK”，连线完成后如图 7.2.45 所示：

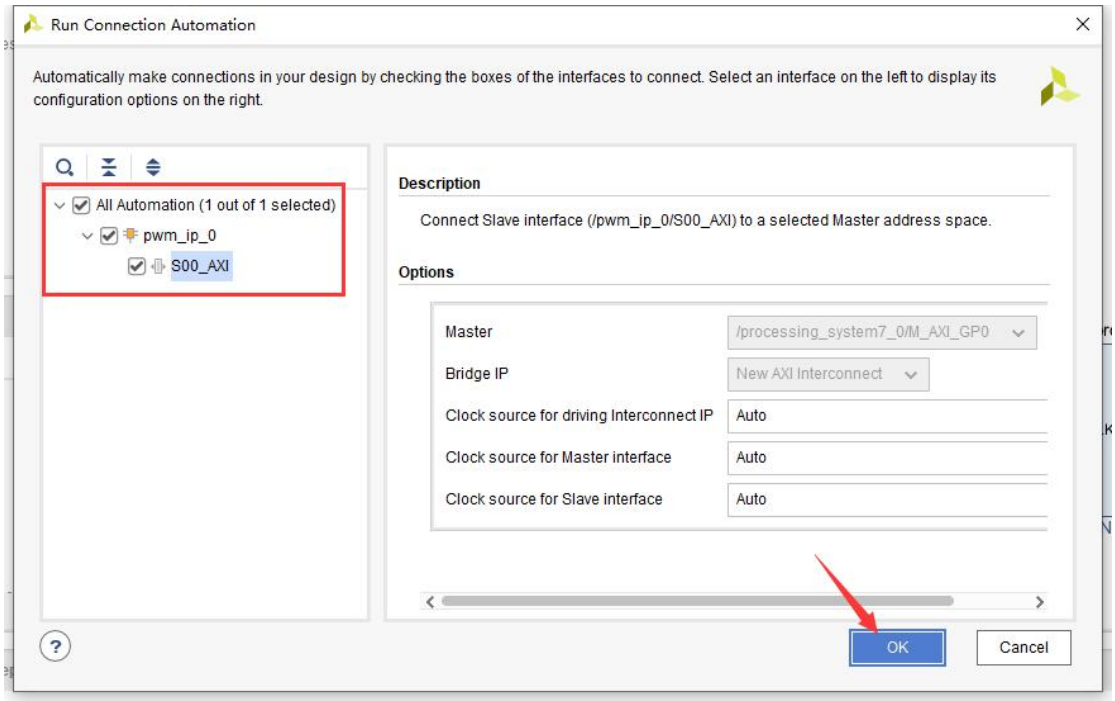


图 7.2.44 设置连接项目

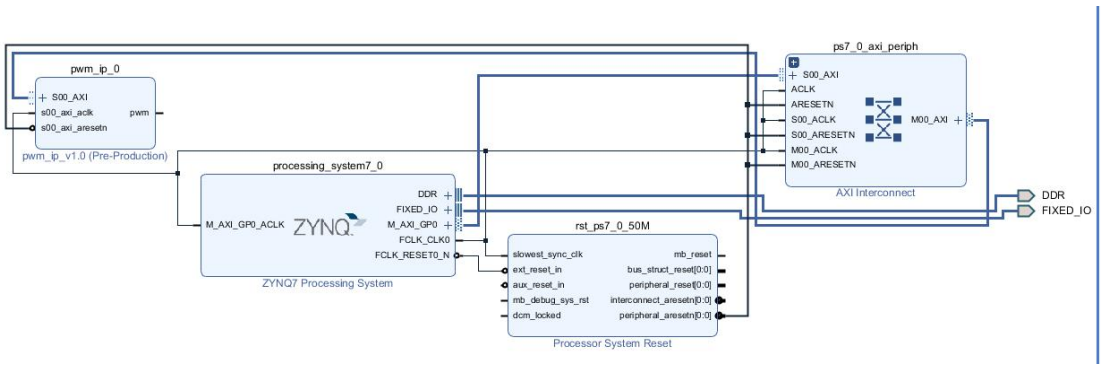


图 7.2.45 完成连线

连线完成后将 pwm 输出的引脚引出，移动鼠标箭头到 pwm 引脚上，变为

铅笔形状后点击右键，选择“Make External”，如下图所示：

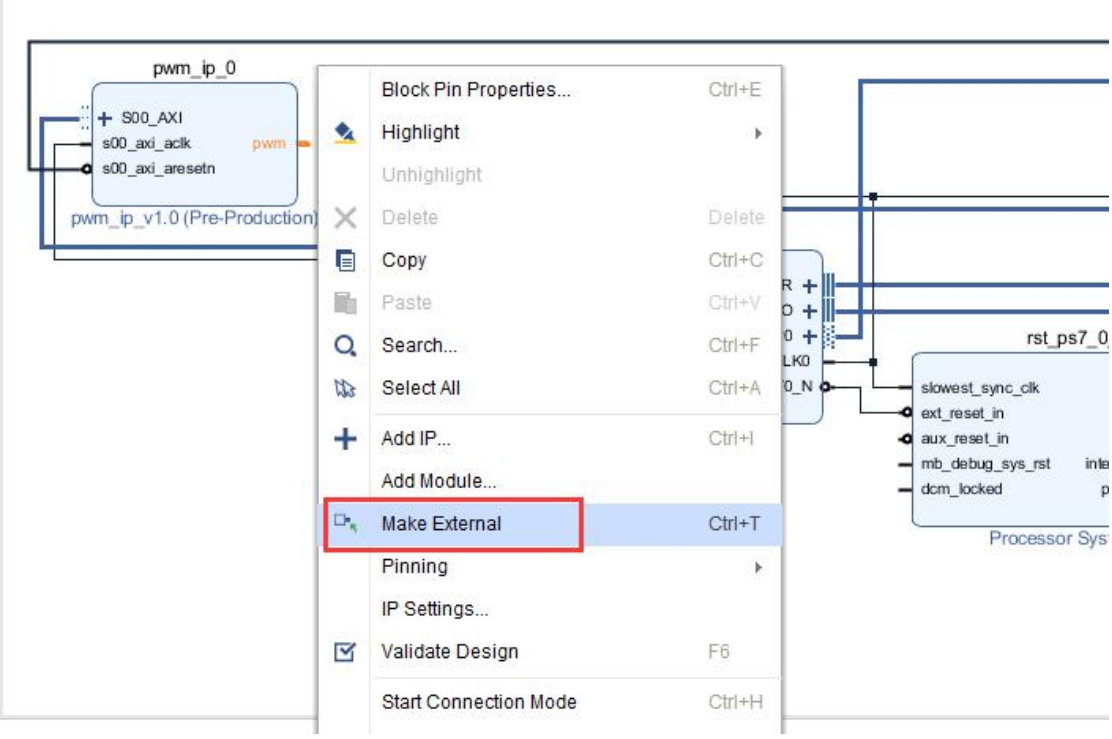


图 7.2.46 引出 pwm 引脚

点击引出的 pwm 引脚，在左侧的窗口修改引脚名称为“pwm”，如下图所示：

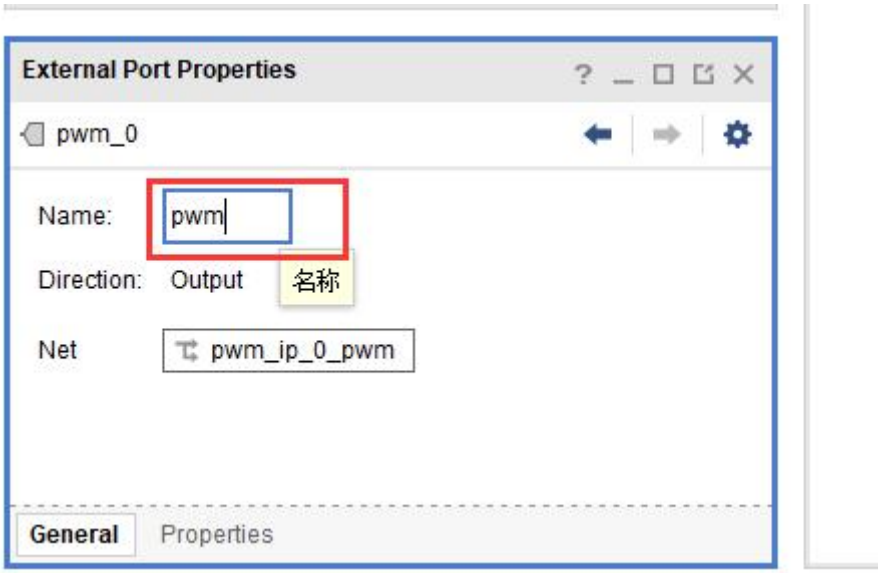


图 7.2.47 修改引脚名称

点击“Regenerate Layout”图标，最终原理图界面如图 7.2.48 所示：

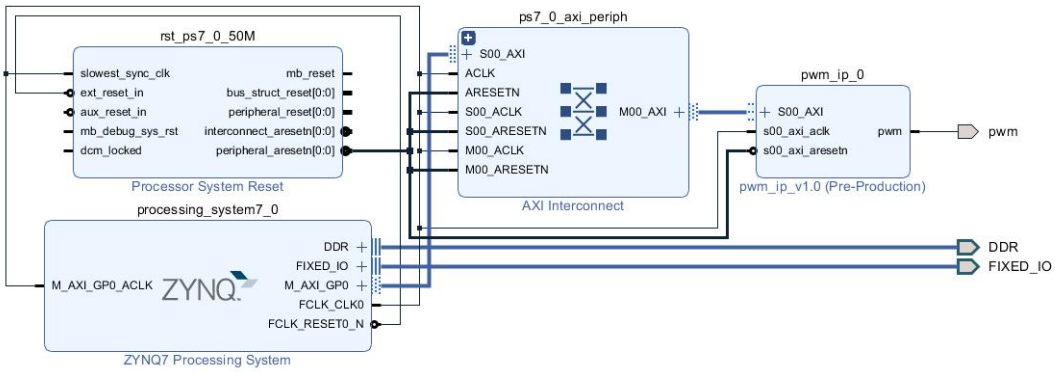


图 7.2.48 最终原理图

之后验证设计并生成顶层 HDL。在 Sources 窗口中，选中 Design Sources 下的 pwm_led.bd，这就是我们刚刚完成的 Block Design 设计。右键点击 pwm_led.bd，在弹出的菜单栏中选择“Generate Output Products”，等待 Generate 完成。

在 Sources 窗口中，右键点击 pwm_led.bd，在弹出的菜单栏中选择“Create HDL Wrapper”。

在左侧 Flow Navigator 导航栏中找到 RTL ANALYSIS，点击该选项中的“Open Elaborated Design”。在弹出的对话框中点击“OK”。 如下图所示：

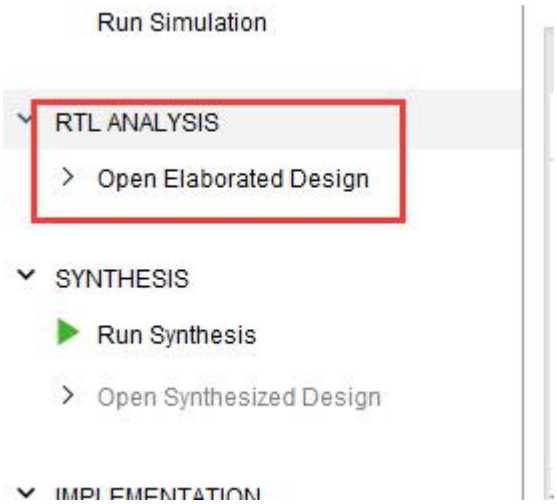


图 7.2.49 打开引脚约束

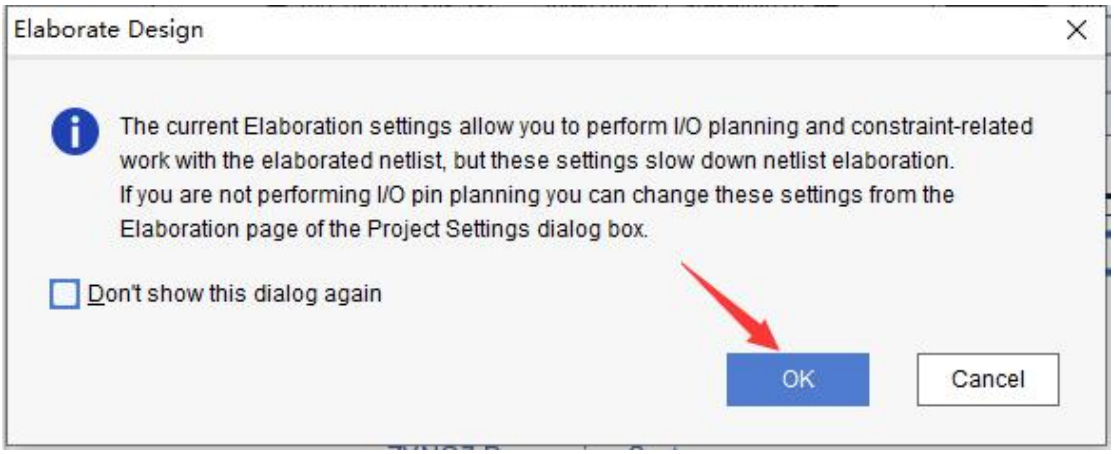


图 7.2.50 点击“OK”

在打开的界面点击“I/O port”，之后在下方配置引脚约束，以及引脚电平。本实验以 Mizar 开发板为例，选择 pl_led1，其他型号开发板请参考原理图配置。

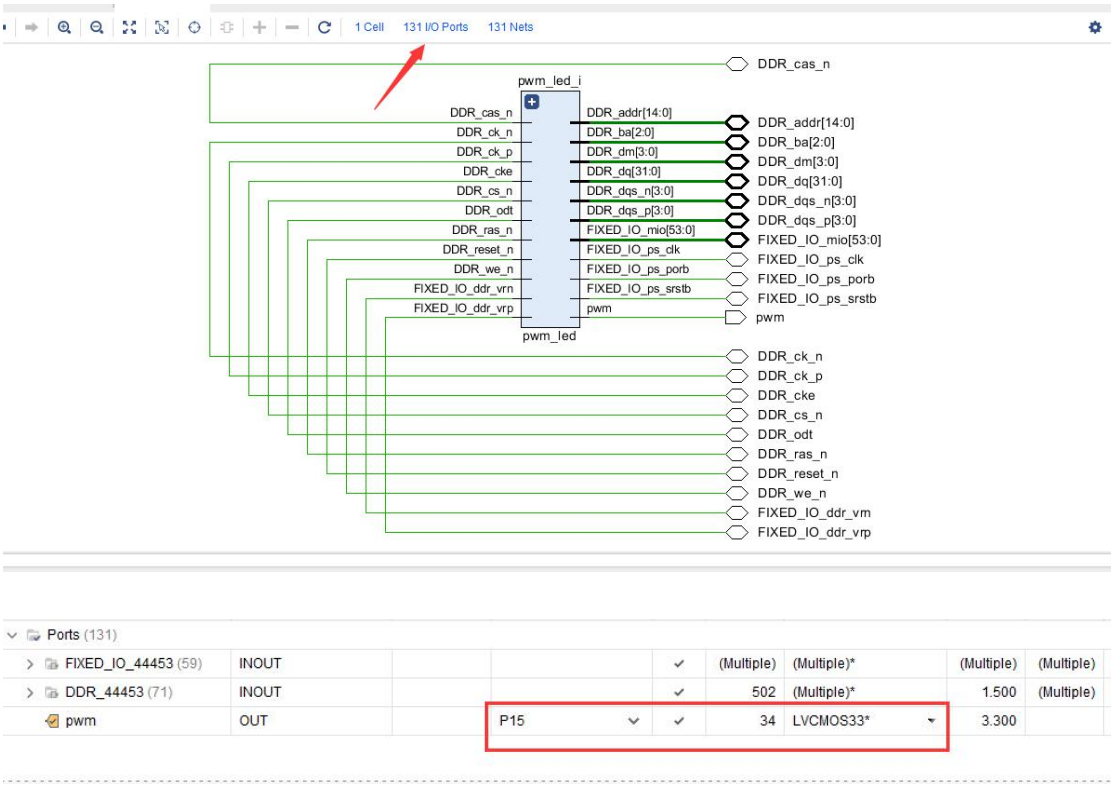


图 7.2.51 配置引脚约束

配置完成后快捷键 Ctrl + S 保存设置，在对话框中输入文件名“pwm_led”，点击“OK”。

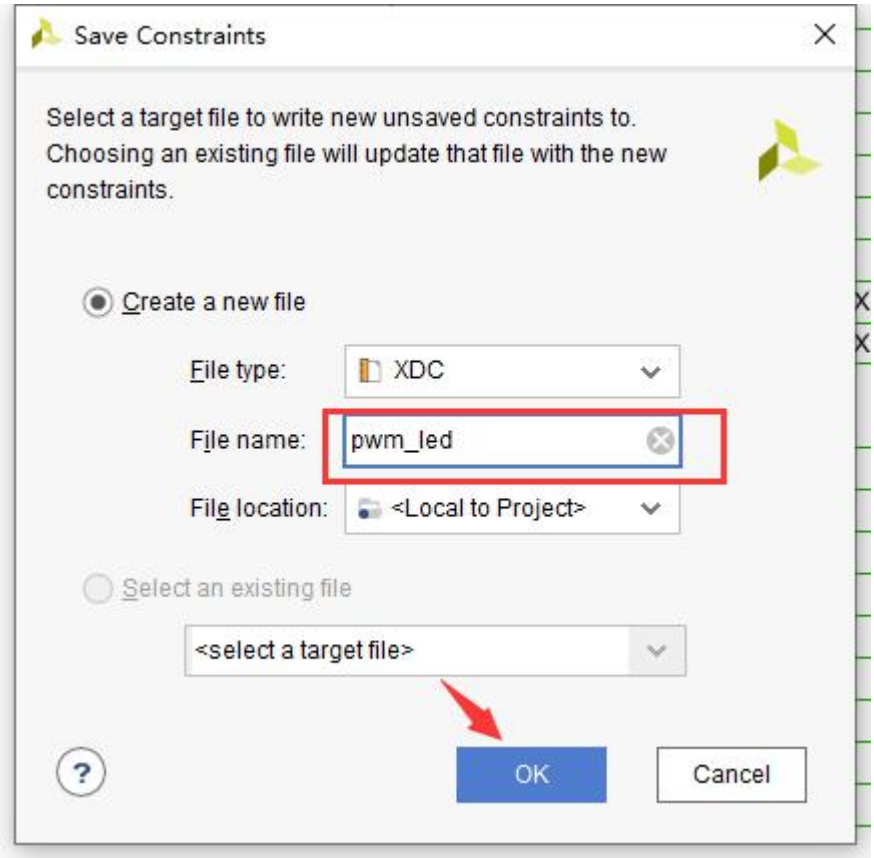
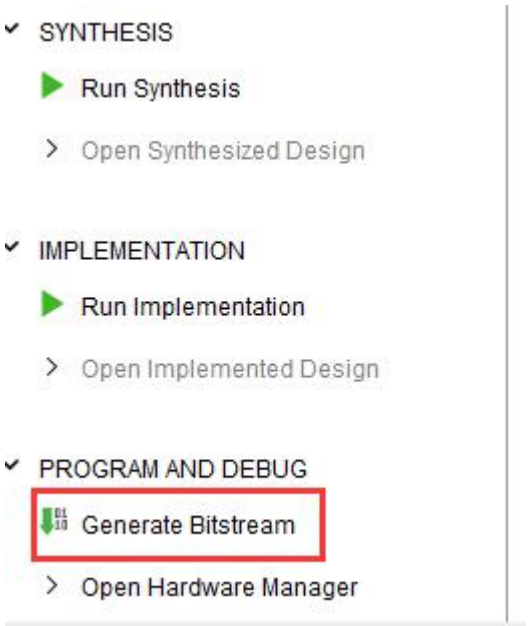


图 7.2.52 保存为.xdc 文件

在左侧 Flow Navigator 导航栏中找到 PROGRAM AND DEBUG，点击该选项中的“Generate Bitstream”。在连续弹出的对话框中依次点击“YES”、 “OK”。然



后 Vivado 工具开始依次对设计进行综合、实现、并生成 Bitstream 文件。

图 7.2.53 生成 Bitstream 文件

生成完成导出到 SDK，在菜单栏中选择 File > Export > Export hardware。在弹出的对话框中，勾选“Include bitstream”，然后点击“OK”按钮。

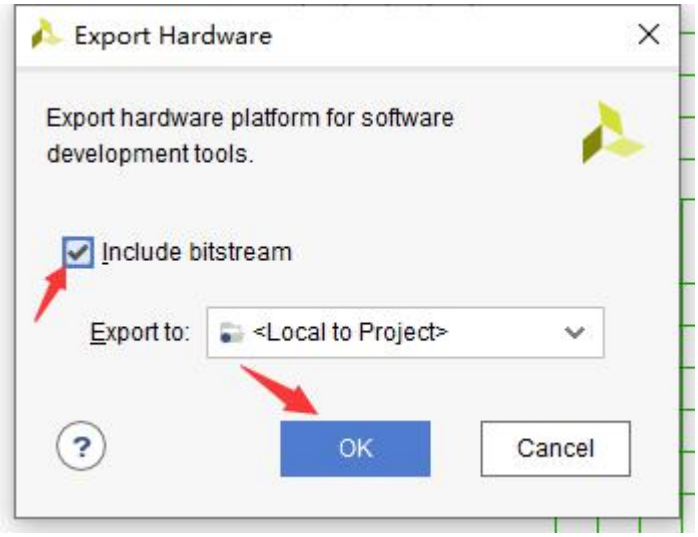


图 7.2.54 导出 Bitstream 文件

硬件导出完成后，选择菜单 File->Launch SDK。在弹出的界面中，点击“OK”。硬件的设计工作就完成了。

7.3 软件设计

在打开的 SDK 软件中点击菜单栏 File->New->Application Project，新建一个工程并命名为“pwm_led”，工程模板选择“Empty Application”。

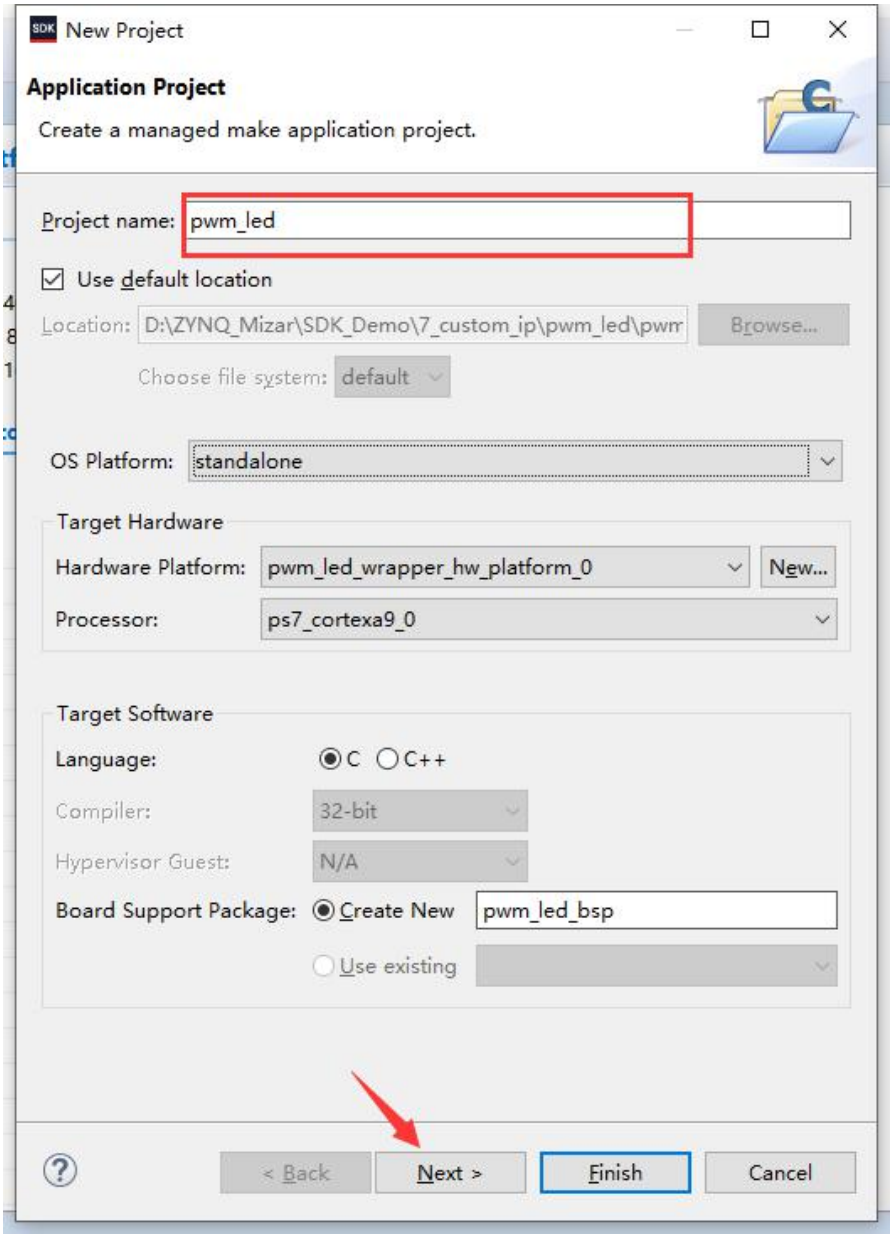


图 7.3.1 新建工程

工程建立后在新工程中，新建源文件。在 pwm_led/src 目录上右键，选择 New->Source File 添加主程序源文件，并命名为“main.c” 如图 7.3.2 所示：

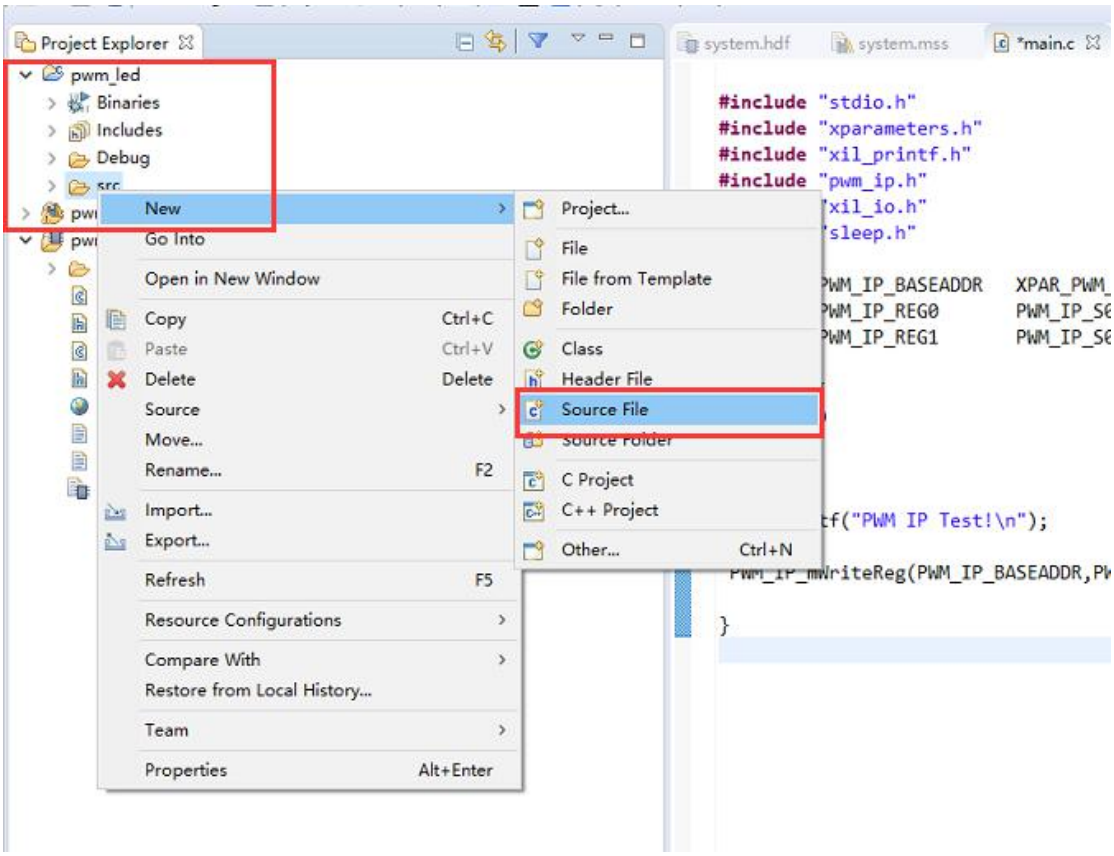


图 7.3.2 添加主程序源文件

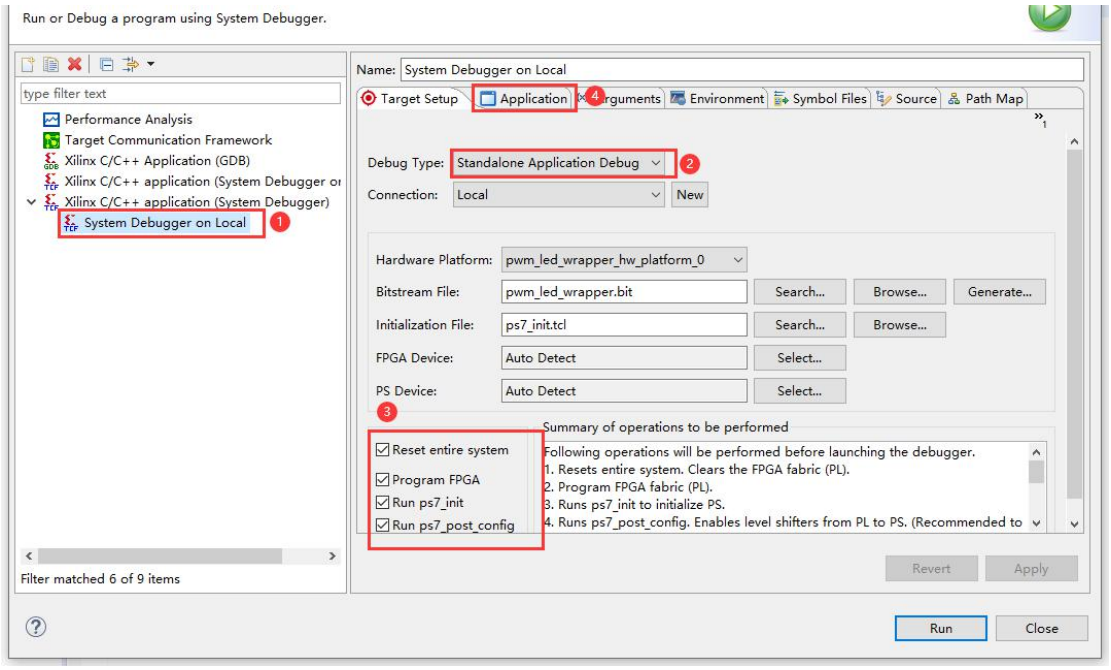
添加完成后自动打开文本编辑窗口，在其中输入以下程序：

输入完成后快捷键 Ctrl + S 保存并编译程序。编译完成没有错误软件设计工

作就完成了。

7.4 下载验证

设计完成之后要将程序下载到开发板上进行验证，点击菜单栏“Run”，选择“Run Configurations…”，在弹出的页面双击“Xilinx C/C++ application(System



Debugger)”，之后按照如图所示顺序依次配置：

图 7.4.1 程序烧录配置

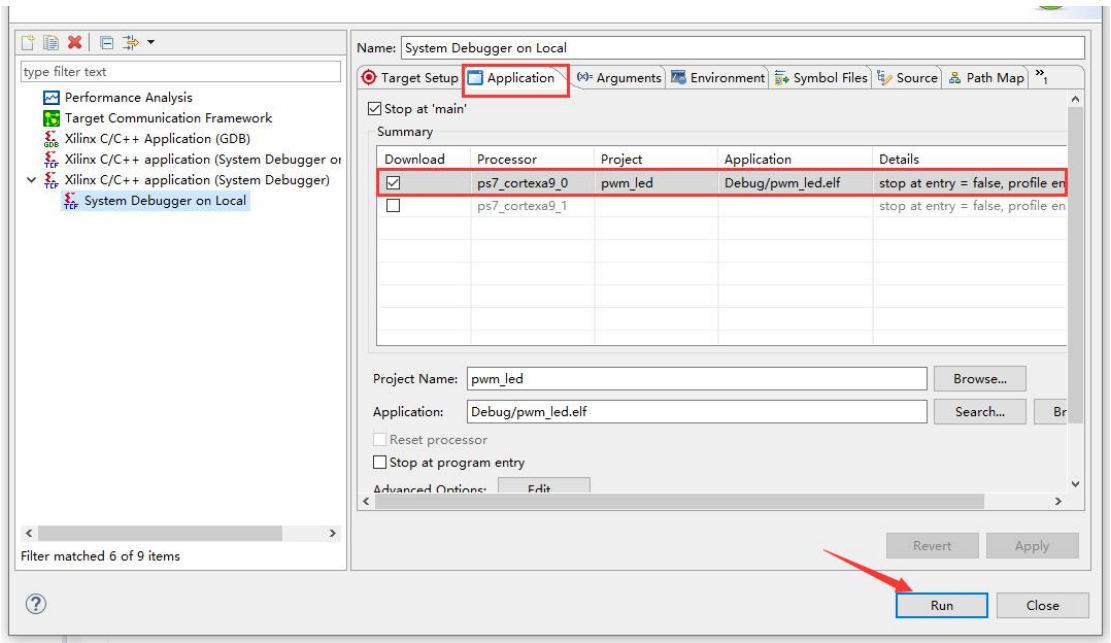


图 7.4.2 选择烧录内核

配置完成后点击“Run”开始烧录。烧录完成后可以看到开发板上 pl_led1 出现呼吸效果，如下图所示：

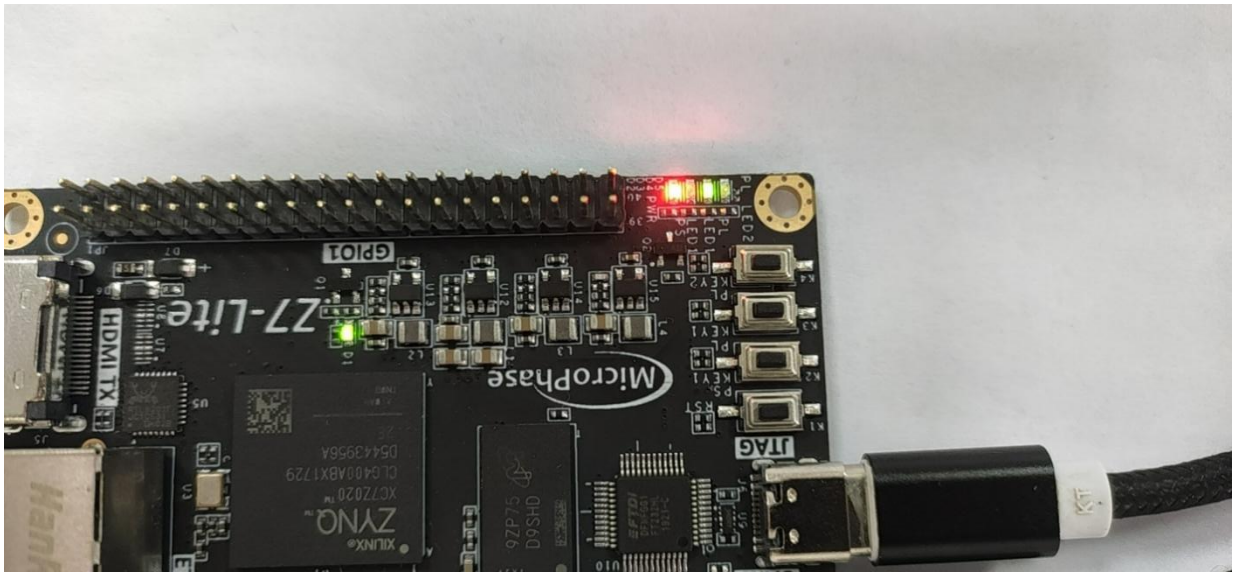


图 7.4.3 实验效果

7.5 章节小结

在本实验中，我们学习了自定义 IP 核的使用方法，这种方法在实际开发过程中经常用到，这也是软件工程师核 FPGA 工程师合作的主要方式，那就是 FPGA 工程师负责设计 IP 核，软件工程师调用这些 IP 核来实现具体功能。

第 8 章　　UART 读写实验

在之前的实验中，我们多次利用串口打印信息来监控程序的运行，同时在调试程序、查看数据时使用串口也是非常方便简单的。除此之外，许多模块的使用控制都是利用串口和各种通讯协议来实现的，这就需要使用者不仅知道怎么利用串口打印信息，还要学会串口中断等串口的其他功能。本章我们就来学习串口中断的使用方法。

本章包括以下几个部分：

8.1 实验简介

8.2 硬件设计

5.3 软件设计

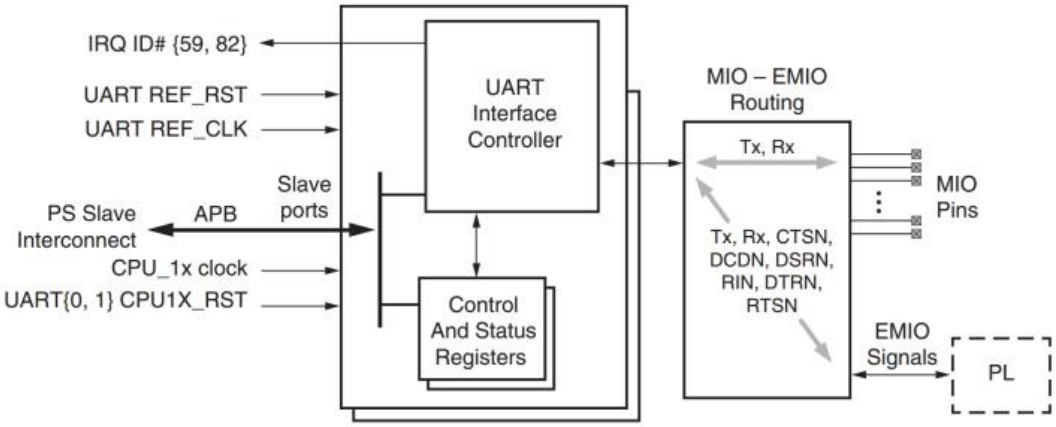
5.4 下载验证

5.5 章节小结

8.1 实验简介

UART 控制器

Zynq 中的 UART 控制器是一个全双工异步收发器，支持各种可编程的波特率和 I/O 信号格式，能够自动生成奇偶校验，支持多种检测模式。控制器的结构



框图如图所示：

UART 操作由配置和模式寄存器控制。采用独立的 Rx 和 Tx 数据路径，每个路径包含一个 64 字节的 FIFO。控制器对 Tx 和 Rx FIFO 中的数据进行串并转换操作，还有一个模式开关，支持 RxD 和 TxD 信号的各种回环配置。FIFO 的中断标志支持轮询处理或中断驱动处理两种方式。程序中我们通过 Rx 和 Tx 的数据端口寄存器来读写数据字节。

Zynq 内部有两个 UART 控制器，UART0 和 UART1，都支持如下特性：

. 可编程波特率发生器；

. 64 字节的接收和发送 FIFO；

. 可编程的串口协议，6、7、8 位的数据位；1、1.5、2 位的停止位；奇校验、偶校验、无校验等；

. 奇偶校验检测、帧错误检测、溢出错误检测；

. 生成换行符；

. 生成中断；

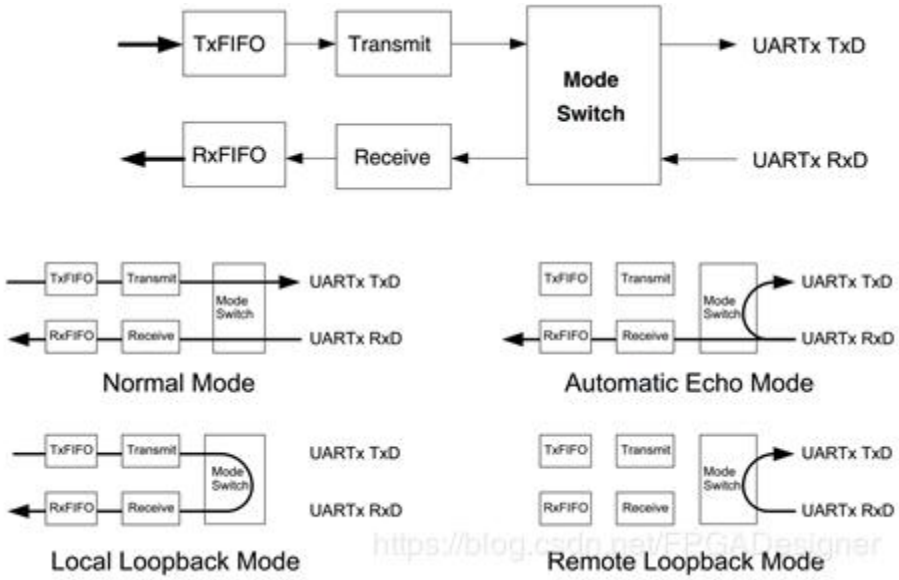
. RxD 和 TxD 模式，使用模式开关选择 Normal、echo 或诊断回环；

. 将 UART0 和 UART1 回环；

. 调制解调控制信号 CTS、RTS、DSR、DTR、RI 和 DCD，仅在 EMIO 接口上可用。

I/O 模式选择

模式开关控制着 RxD 和 TxD 信号在控制器内部的路由。有 4 种工作模式，如下图所示：



下面简单介绍下各模式的功能：

1、正常模式：这是标准的 UART 工作模式，TxD 和 RxD 有独立的路径。

2、自动 echo 模式：Echo 模式仍然会接收 RxD 上的数据，但模式开关将数据路由到接收器和 TxD 引脚，控制器不能用发射机来发射数据。

3、本地回环模式：这个模式内部信号不与 RxD 和 TxD 连接，发射的数据被回环到接收器中。

4、远程回环模式：此模式将 RxD 与 TxD 连接，控制器无法在 TxD 上发送任何内容，也不能在 RxD 上接收任何内容。

1. UART 启动

UART 的启动顺序如下：

1、复位控制器；

2、配置 I/O 信号路由。Rx/Tx 可以路由到 MIO 或 EMIO，只有 EMIO 可以使用调制解调控制信号；

3、配置 UART 参考时钟 UART_Ref_Clk；

4、配置控制器功能；

5、配置中断，我们可通过中断来管理 RxFIFO 和 TxFIFO；

6、配置调制解调控制（可选）；

7、管理发送和接收的数据，可以采用轮询或中断驱动处理两种方式。

2. 配置控制器函数

控制器功能主要配置字符帧、波特率、FIFO 触发器等级、Rx 超时机制，并启用控制器。重置控制器后必须要配置所有这些参数。步骤如下：

配置 UART 字符帧，数据位长度、校验位、停止位、I/O 模式等；

配置波特率；

设置 RxFIFO 触发器等级，可以选择启用或禁用该功能；

使能控制器；

配置接收器的超时机制，可以选择启用或禁用该功能。

3. 发送数据

程序中我们可以使用轮询和中断两种方式控制到 TxFIFO 和 RxFIFO 的数据流。这两个 FIFO 大小为 64 个字节，因此当 TxFIFO 的空标志有效时，我们可以直接向其写入 64 个字节，无需检查 TxFIFO 的状态。实际上当发送器处于活跃状态时，可写入的字节数要超过 64 个字节，因为控制器同时也在移出数据，将其串行化转移到 TxD 信号上。

使用轮询方法的数据发送流程如下：

1、检查 TxFIFO 是否为空；

2、向 TxFIFO 写入数据，可写入 64 个字节；

3、写入更多数据。我们可以等 TxFIFO 空了后再写 64 个字节，即执行第 2 步；也可以检测 TxFIFO 是否有空间，即不停的读取 4、4、TFUL 标志和写单个字节的数据。

使用中断方法的数据发送流程如下：

1、禁用 TxFIFO 空中断；

2、向 TxFIFO 写 64 个字节的数据；

3、检测 TxFIFO 是否有空间，不停的读取 TFUL 标志和写单个字节的数据；

4、重复步骤 2-3，直到 TxFIFO 已满；

5、使能 TxFIFO 空中断；

6、等待，直到 TxFIFO 为空，然后从步骤 1 重新开始；

4. 接收数据

使用轮询方法的数据接收流程如下：

1、等待，直到 RxFIFO 中的数据数量达到触发等级；

2、从 RxFIFO 中读取数据；

3、重复步骤 2 直到 FIFO 空；

4、发生 Rx 超时中断时将其重置。

使用中断方法的数据接收流程如下：

1、使能中断；

2、等待，直到 RxFIFO 中的数据数量达到触发等级或者发生超时；

3、从 RxFIFO 中读取数据；

4、重复步骤 2-3 直到 FIFO 空；

5、清除中断标志。

上面介绍了一些 UART 控制器的相关知识，由于篇幅限制介绍的不够完整，读者可在网上自行查阅更加详细的资料。在了解相关知识以后，我们接下来进行硬件设计的工作。

8.2 硬件设计

在本次实验中，主要学习串口中断的使用方法，不需要其他的功能外设，在硬件设计上与实验 1 相同，所以我们打开实验 1 的 Vivado 工程，点击 File->Project->Save As 另存为新的工程，命名为“uart”。然后点击导出硬件描述文件到 SDK 即可。

8.3 软件设计

本实验的目标是利用串口的中断功能读取电脑通过串口工具发送给芯片的数据，然后再通过 TX 发送回电脑。在编写程序之前，我们先来了解一下本实验流程。程序主要包括四部分：主程序、串口初始化程序、串口中断配置程序、中断服务程序。各程序流程如下：

主程序流程：

UART 初始化->UART 中断配置->检查是否收到数据->数据如果收到，发送收到的数据；

UART 初始化程序流程：

初始化 UART 控制器->UART 设备自检->设置工作模式为普通模式->设置波特率，符号位，停止位，校验位->设置中断触发等级

中断初始化流程：

中断控制器初始化->设置并打开中断异常处理功能->关联中断源与中断服务函数->设置中断触发方式为，FIFO 接收到一位数据及触发中断，FIFO 空时触发中断->使能串口中断

中断服务程序流程：

判断状态寄存器是接收中断还是 FIFO 空中断->清除相应中断标志->接收中断则读取 RxFIFO 数据，FIFO 空中断则将接收标志 ReceivedFlag 置 1 表明数据已读取完成。

在了解了程序流程之后，我们来完成程序的编写。在打开的 SDK 中新建一个空白的工程命名为“uart”，然后在 UART->src 下添加一个源文件命名为“main.c”。

```
#include "xparameters.h"
#include "xuartps.h"
#include "xil_printf.h"
#include "xscugic.h"
#include "stdio.h"

#define UART_DEVICE_ID      XPAR_XUARTPS_0_DEVICE_ID     //串口设备ID
#define INTC_DEVICE_ID      XPAR_SCUGIC_SINGLE_DEVICE_ID //中断ID
#define UART_INT_IRQ_ID     XPAR_XUARTPS_1_INTR          //串口中断ID

XScuGic Intc;               //中断控制器驱动程序实例
XUartPs Uart_Ps;            //串口驱动程序实例

#define MAX_LEN    2000//数据缓冲区大小
u8 ReceivedBuffer[MAX_LEN] ;/* 接收数组 */
u8 *ReceivedBufferPtr ;/* 指针*/
volatile u32 ReceivedByteNum ;/* 数据计数 */
volatile u32 ReceivedFlag  ;//接收完成标志
```

在 main.c 的编辑页面输入以下程序：

```
XUartPsFormat UartFormat =
{
        115200,
        XUARTPS_FORMAT_8_BITS,
        XUARTPS_FORMAT_NO_PARITY,
        XUARTPS_FORMAT_1_STOP_BIT
};
/*
 * Function declaration
 */
int UartPsSend(XUartPs *InstancePtr, u8 *BufferPtr, u32 NumBytes) ;
int UartPsRev (XUartPs *InstancePtr, u8 *BufferPtr, u32 NumBytes) ;
int UartInit(XUartPs* uart_ps);
void Handler(void *call_back_ref);
int UartIntrInit(XScuGic *intc, XUartPs *uart_ps);
int main(void)
{
    int status;
    u32 SendByteNum ;
    u8 *SendBufferPtr ;

    ReceivedBufferPtr = ReceivedBuffer ;

    ReceivedFlag = 0 ;
    ReceivedByteNum = 0 ;
    status = UartInit(&Uart_Ps);    //串口初始化
    if (status == XST_FAILURE) {
        xil_printf("Uart Initial Failed\r\n");
        return XST_FAILURE;
    }

```

```

//UART初始化函数
int UartInit(XUartPs* uart_ps)
{
    int status;
    XUartPs_Config *uart_cfg;

    uart_cfg = XUartPs_LookupConfig(UART_DEVICE_ID);
    if (NULL == uart_cfg)
        return XST_FAILURE;
    status = XUartPs_CfgInitialize(uart_ps, uart_cfg, uart_cfg->BaseAddress);
    if (status != XST_SUCCESS)
        return XST_FAILURE;
```

```
    UartIntrInit(&Intc, &Uart_Ps); //串口中断初始化
    xil_printf("Uart Initial Successful\r\n");
    while (1)
    {
        if (ReceivedFlag)
            {
                ReceivedBufferPtr = ReceivedBuffer ;
                SendBufferPtr = ReceivedBuffer ;
                SendByteNum = ReceivedByteNum ;
                ReceivedFlag = 0 ;
```

```c
    //UART设备自检
    status = XUartPs_SelfTest(uart_ps);
    if (status != XST_SUCCESS)
        return XST_FAILURE;

    //设置工作模式:普通模式
    XUartPs_SetOperMode(uart_ps, XUARTPS_OPER_MODE_NORMAL);
    /* 波特率 115200, 八位符号位，无奇偶校验位，一位停止位 */
    XUartPs_SetDataFormat(uart_ps, &UartFormat) ;
    //设置RxFIFO的中断触发等级
    XUartPs_SetFifoThreshold(uart_ps, 1);

    return XST_SUCCESS;
}
```

```c
//UART中断服务函数
void Handler(void *call_back_ref)
{
    XUartPs *uart_instance_ptr = (XUartPs *) call_back_ref;
    u32 ReceivedCount = 0 ;
        u32 UartSrValue ;

        UartSrValue = XUartPs_ReadReg(uart_instance_ptr->Config.BaseAddress, XUARTPS_SR_OFFSET) &
                (XUARTPS_IXR_RXOVR|XUARTPS_IXR_RXEMPTY);
        ReceivedFlag = 0 ;

        if (UartSrValue & XUARTPS_IXR_RXOVR)   /* 检测中断触发 */
        {
            ReceivedCount = UartPsRev(&Uart_Ps, ReceivedBufferPtr, MAX_LEN) ;
            ReceivedByteNum += ReceivedCount ;
            ReceivedBufferPtr += ReceivedCount ;
            /* 清除中断标志 */
            XUartPs_WriteReg(uart_instance_ptr->Config.BaseAddress, XUARTPS_ISR_OFFSET, XUARTPS_IXR_RXOVR) ;
        }
        else if (UartSrValue & XUARTPS_IXR_RXEMPTY)
            {

                XUartPs_WriteReg(uart_instance_ptr->Config.BaseAddress, XUARTPS_ISR_OFFSET, XUARTPS_IXR_RXEMPTY) ;
                ReceivedFlag = 1 ;
            }

}
```

```c
//串口中断初始化
int UartIntrInit(XScuGic *intc, XUartPs *uart_ps)
{
    int status;
    //初始化中断控制器
    XScuGic_Config *intc_cfg;
    intc_cfg = XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (NULL == intc_cfg)
        return XST_FAILURE;
    status = XScuGic_CfgInitialize(intc, intc_cfg,
            intc_cfg->CpuBaseAddress);
    if (status != XST_SUCCESS)
        return XST_FAILURE;

    //设置并打开中断异常处理功能
    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler)XScuGic_InterruptHandler,
            (void *)intc);
    Xil_ExceptionEnable();
```

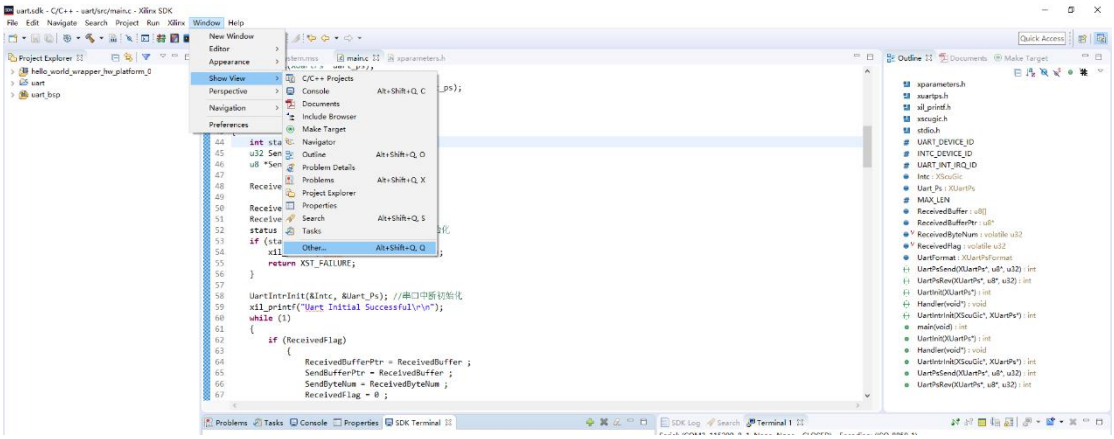
```c
    //为中断设置中断处理函数
    XScuGic_Connect(intc, UART_INT_IRQ_ID,
            (Xil_ExceptionHandler) Handler,(void *) uart_ps);
    //设置UART的中断触发方式
    XUartPs_SetInterruptMask(uart_ps, XUARTPS_IXR_RXOVR|XUARTPS_IXR_RXEMPTY);
    //使能GIC中的串口中断
    XScuGic_Enable(intc, UART_INT_IRQ_ID);
    return XST_SUCCESS;
}

//发送数据
int UartPsSend(XUartPs *InstancePtr, u8 *BufferPtr, u32 NumBytes)
{
    u32 SentCount = 0U;

    /* 设置数据缓冲区 */
    InstancePtr->SendBuffer.RequestedBytes = NumBytes;
    InstancePtr->SendBuffer.RemainingBytes = NumBytes;
    InstancePtr->SendBuffer.NextBytePtr = BufferPtr;


    while (InstancePtr->SendBuffer.RemainingBytes > SentCount)
    {
        /* 将缓冲区的数据填充到FIFO中 */
        if (!XUartPs_IsTransmitFull(InstancePtr->Config.BaseAddress))
        {
            XUartPs_WriteReg(InstancePtr->Config.BaseAddress,
                    XUARTPS_FIFO_OFFSET,
                    ((u32)InstancePtr->SendBuffer.
                            NextBytePtr[SentCount]));
            SentCount++;
        }
    }

    /* 更新缓冲区 */
    InstancePtr->SendBuffer.NextBytePtr += SentCount;
    InstancePtr->SendBuffer.RemainingBytes -= SentCount;

    return SentCount;
}

//接收数据
int UartPsRev(XUartPs *InstancePtr, u8 *BufferPtr, u32 NumBytes)
{
    u32 ReceivedCount = 0;
    u32 CsrRegister;

    /* 设置缓冲区 */
    InstancePtr->ReceiveBuffer.RequestedBytes = NumBytes;
    InstancePtr->ReceiveBuffer.RemainingBytes = NumBytes;
    InstancePtr->ReceiveBuffer.NextBytePtr = BufferPtr;

    /*读取通道状态确定是否收到数据 */
    CsrRegister = XUartPs_ReadReg(InstancePtr->Config.BaseAddress,
            XUARTPS_SR_OFFSET);
```

```
    /*读取数据并将它们存储到缓冲区    */
    while((ReceivedCount < InstancePtr->ReceiveBuffer.RemainingBytes)&&
            (((CsrRegister & XUARTPS_SR_RXEMPTY) == (u32)0)))
    {
        InstancePtr->ReceiveBuffer.NextBytePtr[ReceivedCount] =
                XUartPs_ReadReg(InstancePtr->Config.BaseAddress,XUARTPS_FIFO_OFFSET);
        ReceivedCount++;
        CsrRegister = XUartPs_ReadReg(InstancePtr->Config.BaseAddress,
                XUARTPS_SR_OFFSET);
    }
    InstancePtr->is_rxbs_error = 0;


    if(InstancePtr->ReceiveBuffer.NextBytePtr != NULL){
        InstancePtr->ReceiveBuffer.NextBytePtr += ReceivedCount;
    }
    InstancePtr->ReceiveBuffer.RemainingBytes -= ReceivedCount;

    return ReceivedCount;
}
```

程序输入完毕后快捷键 Ctrl + S 保存并编译，编程完成后查看是否有错误，如有错误检查并修改直至没有错误为止。

8.4 下载验证

在之前的实验中，我们通过在 SDK 中的 teiminal 窗口来查看串口的数据，但是 Termianl 只能查看而不能向串口发送数据所以无法满足我们的实验需要。本实验我使用 SDK 自带的另一个串口工具 SDK Terminal。点击菜单栏 Window->Show View->Others，在弹出的窗口中选择 Xilinx->SDK Terminal，或者使用快捷键 Shift +



Alt+Q，然后再按 Q 即可打开。

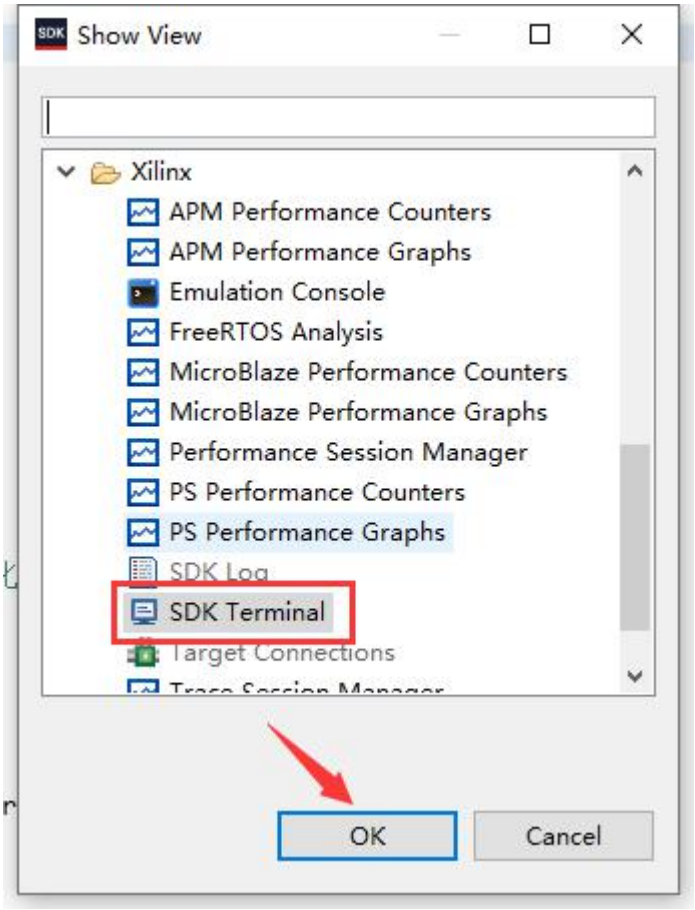


图 8.4.1 打开 SDK Terminal 窗口

打开的 SDK Terminal 窗口显示在下方的控制台，点击如图 8.4.2 所示箭头所指的“+”号可以连接串口。连接成功后下载程序到开发板中可以看到打印出“Uart Initial Successful”的信息。

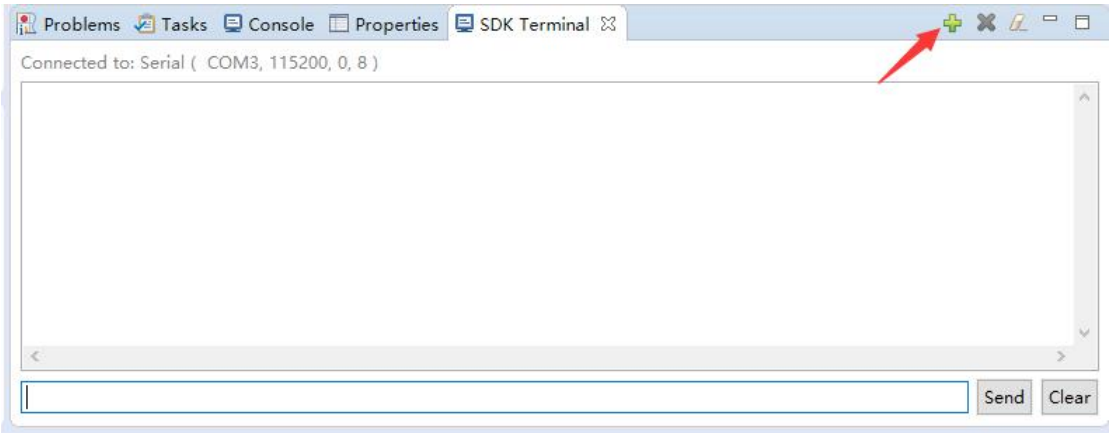
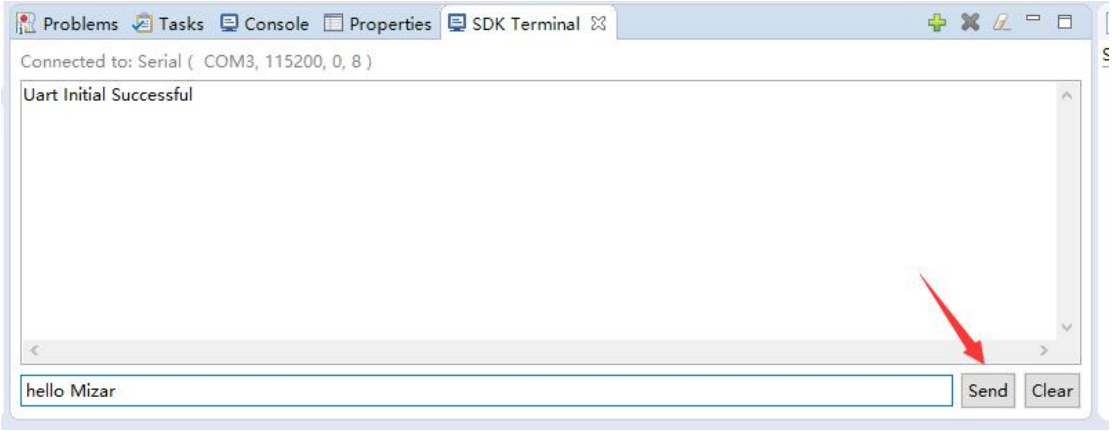


图 8.4.2 连接串口

在下方的输入框内输入“hello Mizar”，点击 Send 即可看到开发板返回我们

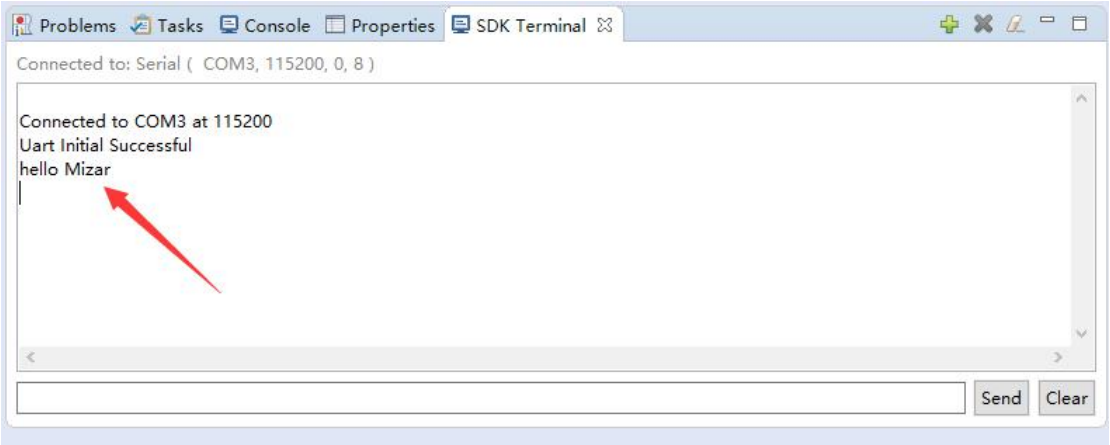


发送的数据。说明我们的实验成功。

图 8.4.3 实验结果

8.5 章节小结

在本实验中我们学习了 UART 中断的相关使用方法，在实际应用中许多使用串口通讯协议控制的模块都会通过串口中断来实现数据传输，这样能够最大程



度节约芯片资源，提高控制效率。希望读者在学会串口中断的基础上，学习串口通信协议的使用。

第 9 章　　XADC 实验

在众多的嵌入式系统中都包含模数转换的功能，在参数采集，设备运行状况监控方面发挥着重要的作用，但是在许多芯片中并不集成模数转换的功能，需要在外部添加专用芯片来实现。而在 ZYNQ 上集成了一个数模混合模块—XADC。XADC 功能强大，它不仅可以读取芯片温度和供电电压等信息来监控芯片的工作状态，还能测量最多 17 路外部模拟信号 ，且支持单极、双极和差分等信号类型。本实验我们来学习 XADC 的使用方法。

本章包括以下几部分内容：

9.1 实验简介

9.2 Vivado 读取 XADC 数

9.3 软件读取 XADC 数据—软件设计

9.4 下载验证

9.5 章节小结

9.1 实验简介

XADC 结构框图如图 9.1.1 所示。从图中可以看到在 XADC 模块中，包含两个精度为 12bit，最大采样率为 1MSPS 的模数转换器，一个模拟多路复用器，片上温度传感器和片上电压传感器。

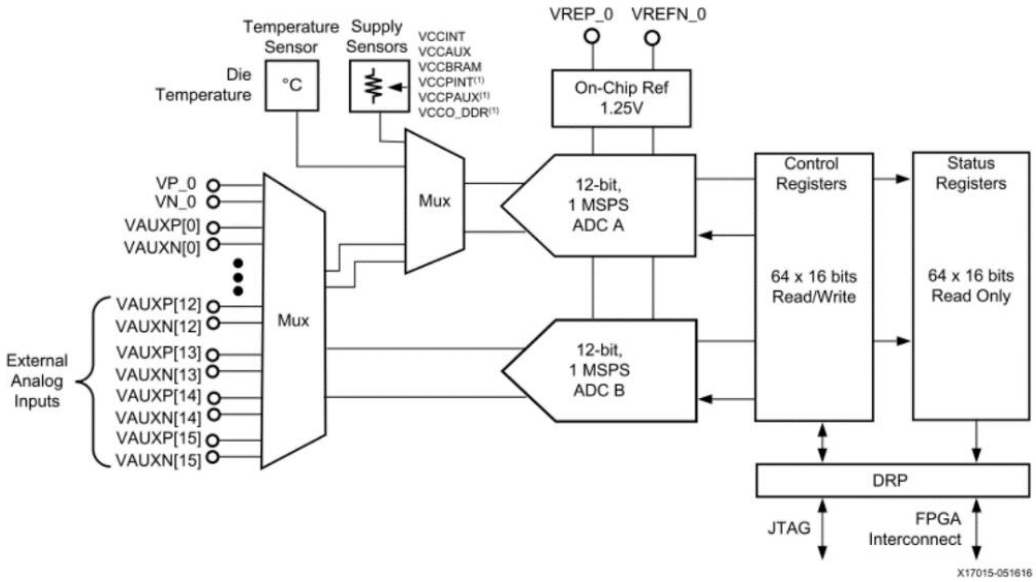


Figure 1-1: **XADC Block Diagram**

图 9.1.1 XADC 结构框图

XADC 的系统框图如下图所示，从图中可以看到我们可以通过三种方式访问 XADC 模块，分别为：1、JTAG 接口，利用 Vivado 软件通过 JTAG 接口直接访问 XADC 内部状态和控制寄存器；2、PS 端通过 PS-XADC 访问 XADC 模块；3、添加 AXI XADC IP 核，通过 M_AXI_GP 总线和 DRP（dynamic reconfiguration port）接口。在本次实验中，我们学习前两种访问方法。

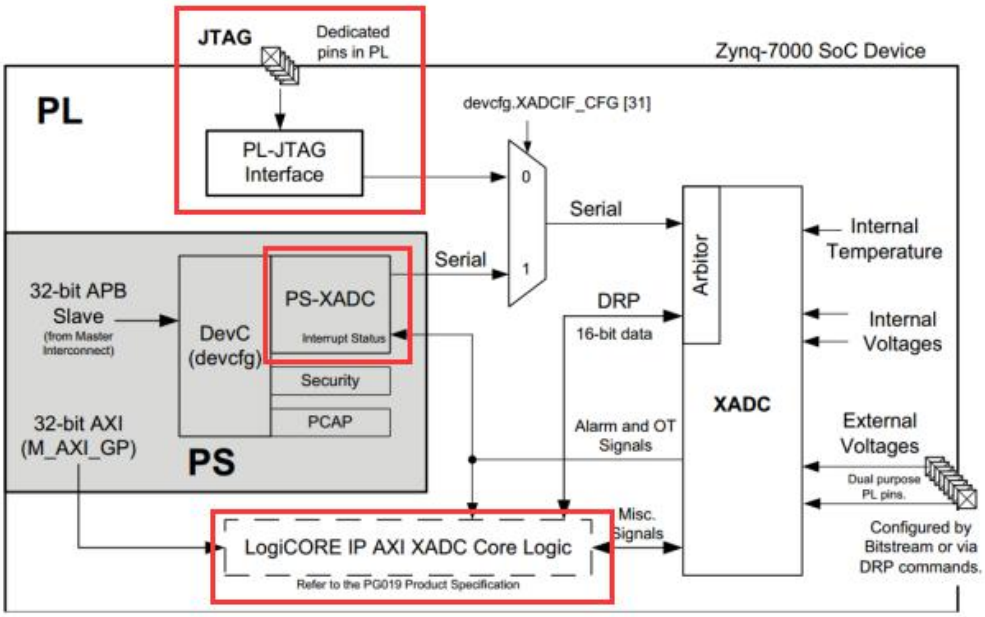
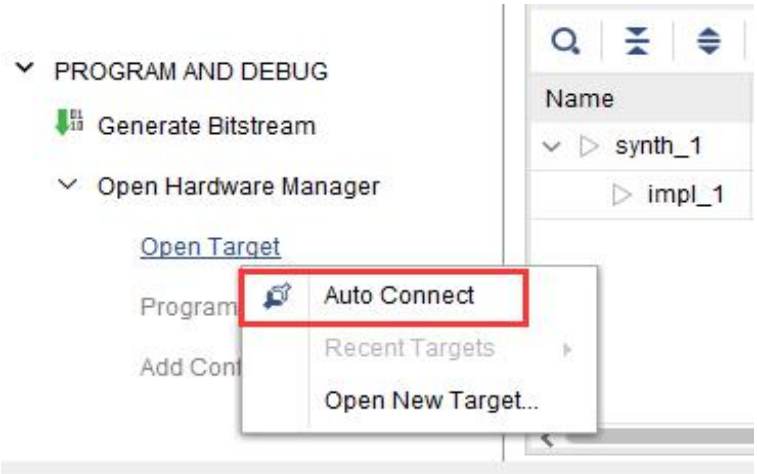


图 9.1.2 XADC 模块系统框图

9.2 Vivado 读取 XADC 数据

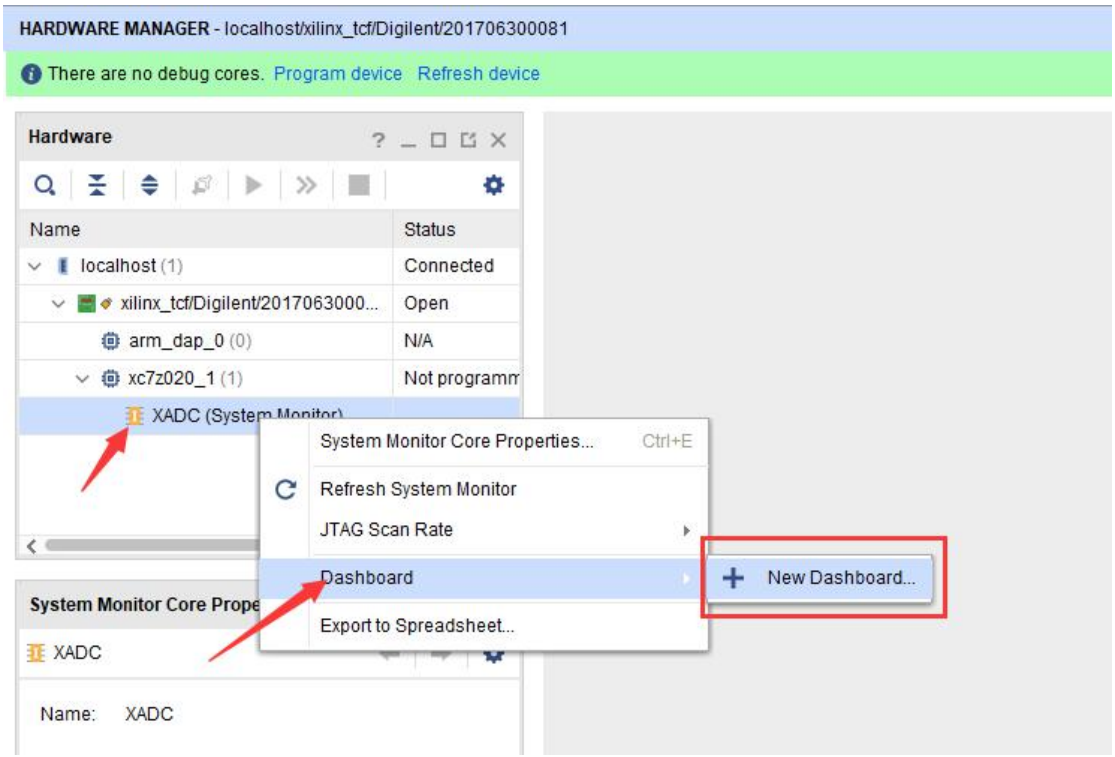
在本次实验中，由于 XADC 是固化到芯片中的，无需配置，只需要打开串口打印数据即可，所以不需要重新设计硬件，只需打开实验 1 的工程，另存为一个新的工程，命名为“XADC”即可。

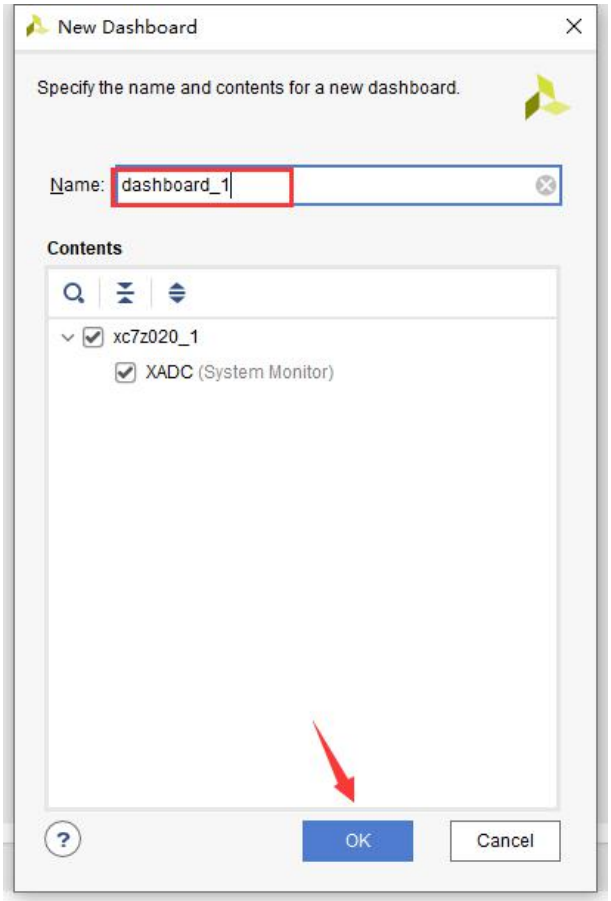
打开“XADC”工程，连接好开发板的电源，将 JTAG 下载接口通过 type-c 线与电脑连接，设置开发板为 JTAG 模式。然后如下图所示点击 Open Hardware



Manager->Open Target->Auto Connect。

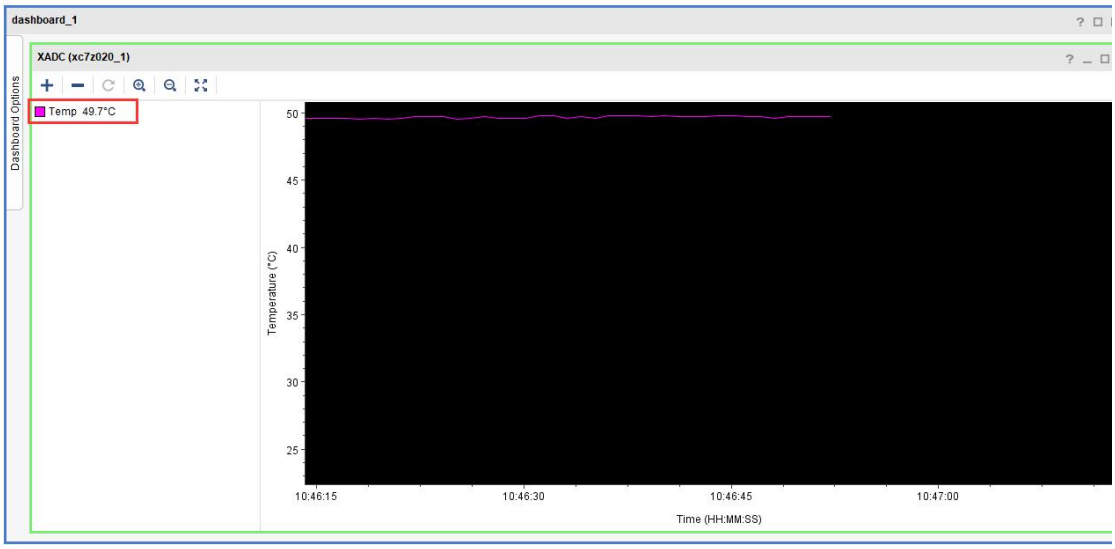
在弹出的界面中右键点击 XADC，之后依次点击 Dashboard->New Dashboard，





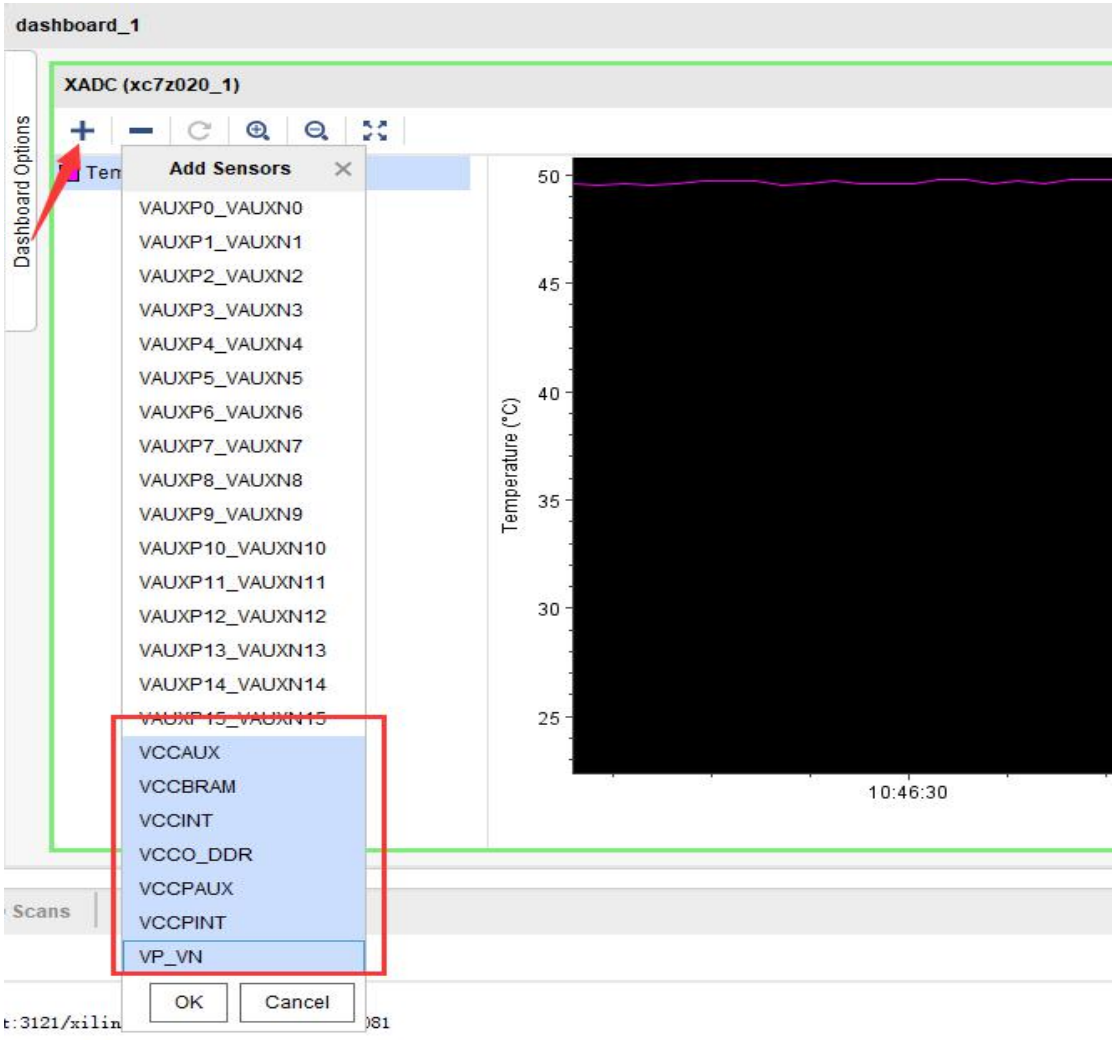
在弹出的对话框中输入名称，然后点击“OK”，就打开了数据监控的窗口。

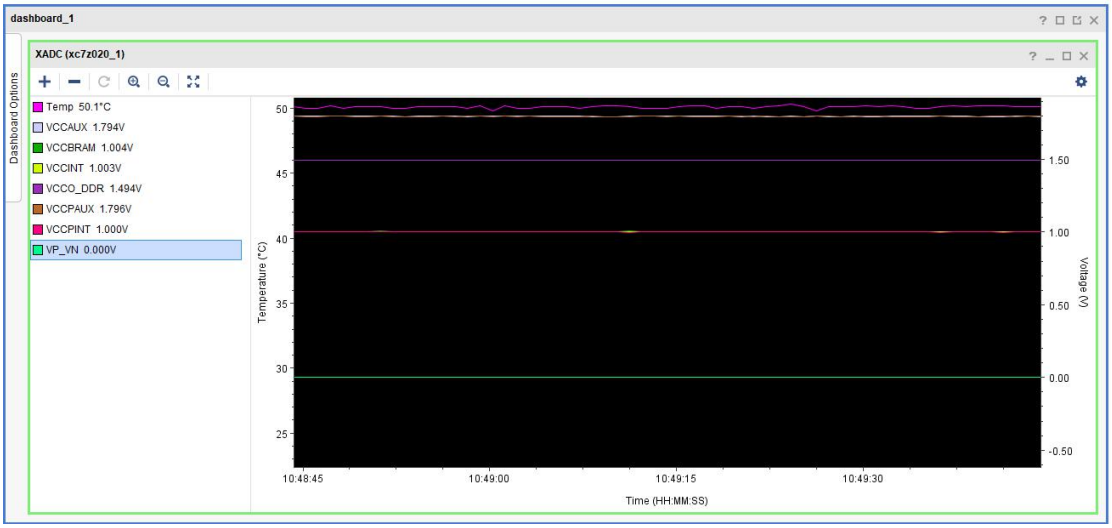
在数据监控窗口，默认只添加了温度数据，可以看到芯片温度波动的曲线，和实



时的温度数据，如下图所示：

点击温度数据上方的“+”可以添加电压的数据。





添加完成后电压数据的显示如图所示：

在窗口中，我们可以通过波形的变化直观的感受到芯片的温度和电压变化，而且这种方法操作简单，但缺点也显而易见，首先需要 Vivado 软件连接，不能脱机使用，其次无法保存相关数据。所以我们接下来介绍另一种方法。

9.3 软件读取 XADC 数据—软件设计

在之前 Vivado 读取 XADC 数据时我们已经建立了本实验的硬件工程，所以不再新建，只需要导出硬件描述文件并打开 SDK 即可。导出前删除工程文件夹下的.sdk 文件夹。在打开的 SDK 软件中新建一个空白工程，命名为“XADC”，在新建的工程内添加主程序源文件 main.c。然后在编辑框内输入以下程序：

```
/***************************** Include Files *******************************/

#include "xparameters.h"
#include "xadcps.h"
#include "xstatus.h"
#include "stdio.h"
#include "xil_printf.h"
#include "sleep.h"
/************************** Constant Definitions ***************************/
#define XADC_DEVICE_ID      XPAR_XADCPS_0_DEVICE_ID
/************************** Variable Definitions ***************************/

static XAdcPs XAdcInst;      /* XADC driver instance */

```

```c
int main(void)
{
    int Status;
        XAdcPs_Config *ConfigPtr;
        u32 TempRawData;
        u32 VccPintRawData;
        u32 VccPauxRawData;
        u32 VccPdroRawData;
        float TempData;
        float VccPintData;
        float VccPauxData;
        float MaxData;
        float MinData;
        XAdcPs *XAdcInstPtr = &XAdcInst;

        printf("Entering the XAdc Polled Example. \r\n");

        /* Initialize the XAdc driver.       */
        ConfigPtr = XAdcPs_LookupConfig(XADC_DEVICE_ID);
        if (ConfigPtr == NULL) {
            return XST_FAILURE;
        }
        XAdcPs_CfgInitialize(XAdcInstPtr, ConfigPtr,
                    ConfigPtr->BaseAddress);

        /* Self Test the XADC/ADC device         */
        Status = XAdcPs_SelfTest(XAdcInstPtr);
        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }

        /* Disable the Channel Sequencer before configuring the Sequence
         * registers. */
        XAdcPs_SetSequencerMode(XAdcInstPtr, XADCPS_SEQ_MODE_SAFE);
        while(1)
        {
        /* Read the on-chip Temperature Data from the ADC data registers.    */
        TempRawData = XAdcPs_GetAdcData(XAdcInstPtr, XADCPS_CH_TEMP);
        TempData = XAdcPs_RawToTemperature(TempRawData);
        printf("\r\nThe Current Temperature is %.04f Centigrades.\r\n",
                    TempData);
        /* Read the VccPint Votage Data  from the ADC data registers.        */
        VccPintRawData = XAdcPs_GetAdcData(XAdcInstPtr, XADCPS_CH_VCCPINT);
        VccPintData = XAdcPs_RawToVoltage(VccPintRawData);
        printf("The Current VCCPINT is %.04f Volts. \r\n",
                VccPintData);
        /* Read the VccPaux Votage Data from the ADC data registers.         */
        VccPauxRawData = XAdcPs_GetAdcData(XAdcInstPtr, XADCPS_CH_VCCPAUX);
        VccPauxData = XAdcPs_RawToVoltage(VccPauxRawData);
        printf("The Current VCCPAUX is %.04f Volts. \r\n",
                 VccPauxData );
        /* Read the VccPdro Votage Data  from the ADC data registers. */
        VccPdroRawData = XAdcPs_GetAdcData(XAdcInstPtr, XADCPS_CH_VCCPDRO);
        VccPintData = XAdcPs_RawToVoltage(VccPdroRawData);
        printf("The Current VCCPDDRO is %.04f Volts. \r\n",
                 VccPintData );
        sleep(3);
        }


        return XST_SUCCESS;
}
```

程序输入完成后快捷键 Ctrl + S 保存并编译。

9.4 下载验证

程序设计完成之后就可下载到开发板中进行验证。按照之前的方法连接好电源，下载线和串口线，在 SDK 软件中添加 Terminal 窗口并连接串口。之后将程序烧录到开发板上，即可在 Terminal 窗口中看到开发板每隔 3S 打印一次芯片的

温度和电压等信息。

图 9.4.1 实验结果

9.5 章节小结

在本实验中，我们学习了两种读取 XADC 数据的方法，我们仅打印了部分数据作为示例，读者可以查阅 xadcps.h 头文件尝试打印更多其他的数据，深入理解 XADC 模块。

第 10 章　　PS 端以太网应用实验

随着 5G 的兴起，万物互联成为将来发展的一个方向，更多的设备将具备联网的功能，将数据上传到云端。得益于 TCP/IP 协议的优越性，该协议已成为当前应用的主流网络协议。在嵌入式网络设备中由于受到硬件资源的限制，实现完整的 TCP/IP 协议十分困难，这就需要一种特殊的实现方式，LWIP 作为一种轻量级的 TCP/IP 协议实现方式充分满足了这一要求。本章我们就来初步认识、了解 LWIP 的使用。

本章包括以下几个部分：

10.1 实验简介

10.2 硬件设计

10.3 软件设计

10.4 下载验证

10.5 章节小结

10.1 实验简介

（1）TCP/IP 协议简介

TCP/IP（Transmission Control Protocol/Internet Protocol，传输控制协议/网际协议）体系结构是指能够在多个不同网络间实现的协议簇。该协议簇是在美国国防高级研究计划局（Defense Advanced Research Projects Agency，DARPA）所资助的实验性 ARPARNET 分组交换网络、无线电分组网络和卫星分组网络上研究开发成功的。网络部分瘫痪时仍保持较强的工作能力和灵活性。这种应用环境导致了一系列协议的出现，从而使不同类型的终端和网络间能够进行有效通信。实际上，Internet 已经成为全球计算机互联的主要体系结构，而 TCP/IP 协议是 Internet 的代名词，是将异种网络、不同设备互联起来，进行正常数据通信的格式和大家遵守的约定。

TCP/IP 协议包括两部分：传输控制协议和网际协议。TCP/IP 的通信任务组织成 5 个相对独立的层次：应用层、传输层、互联网层（对应 OSI 的网络层）、网络接口层和物理层，其中网络接口层和物理层常称为物理网层。各层功能如下:

(1)应用层：应用程序通过这一层访问网络，常见 FTP、HTTP、DNS 和 TELNET 协议；

(2)传输层：TCP 协议和 UDP 协议；

(3)网络层：IP 协议，ARP、RARP 协议，ICMP 协议等，网络层负责相邻计算机之间的通信

(4)网络接口层：是 TCP/IP 协议的基层（最低层），负责数据帧的发送和接收。

（2）LWIP 简介

lwip 是瑞典计算机科学院(SICS)的 Adam Dunkels 开发的一个小型开源的 TCP/IP 协议栈。LwIP 是 Light Weight (轻型)IP 协议，有无操作系统的支持都可以运行。LwIP 实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用，它只需十几 KB 的 RAM 和 40K 左右的 ROM 就可以运行，这使 LwIP 协议栈适合在低端的嵌入式系统中使用。

lwIP 协议栈主要关注的是怎么样减少内存的使用和代码的大小，这样就可以让 lwIP 适用于资源有限的小型平台例如嵌入式系统。（来自百度百科）

其主要特性如下：

(1)支持多网络接口下的 IP 转发；

(2)支持 ICMP 协议；

(3)包括实验性扩展的 UDP(用户数据报协议)；

(4)包括阻塞控制、RTT 估算、快速恢复和快速转发的 TCP(传输控制协议)；

(5)提供专门的内部回调接口(Raw API)，用于提高应用程序性能；

(6)可选择的 Berkeley 接口 API (在多线程情况下使用) ；

(7)在最新的版本中支持 ppp；

(8) 新版本中增加了的 IP fragment 的支持；

(9) 支持 DHCP 协议,动态分配 ip 地址。

我们简单介绍了一些关于 TCP/IP 和 LWIP 的相关知识，让读者有一个初步的了解，实际上关于 TCP/IP 的知识体系十分庞大，需要读者自行去学习，有一个更加深入的了解。在本次实验中，我们将利用 SDK 自带的工程模板来实现 LWIP 的通讯，初步学习 LWIP 的应用。

在本型号开发板上的网络 phy 芯片并未直接连接到芯片 PS 端网络控制器的 MIO 之上，在本次实验中，我们将利用 EMIO 将 PS 端的网络控制器引出到 PL 端

使用。

10.2 硬件设计

首先按照之前的步骤新建一个 Vivado 工程，并命名为“ps_lwip”。创建好的工程如下图所示：

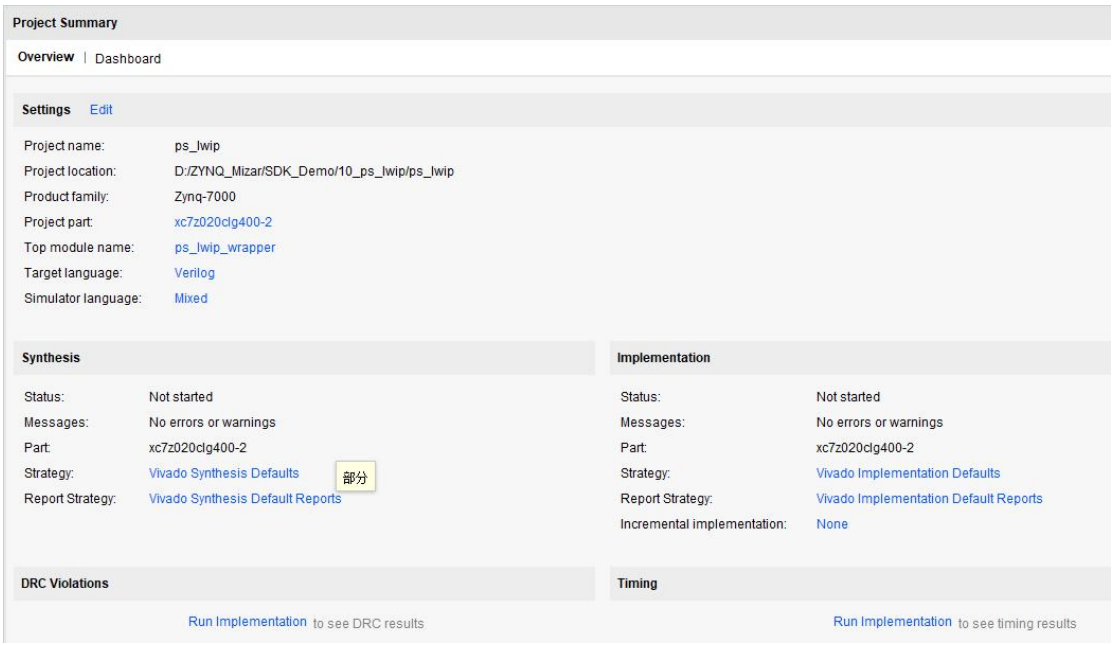


图 10.2.1 创建工程

新建 Block Design 并命名为“ps_lwip”。

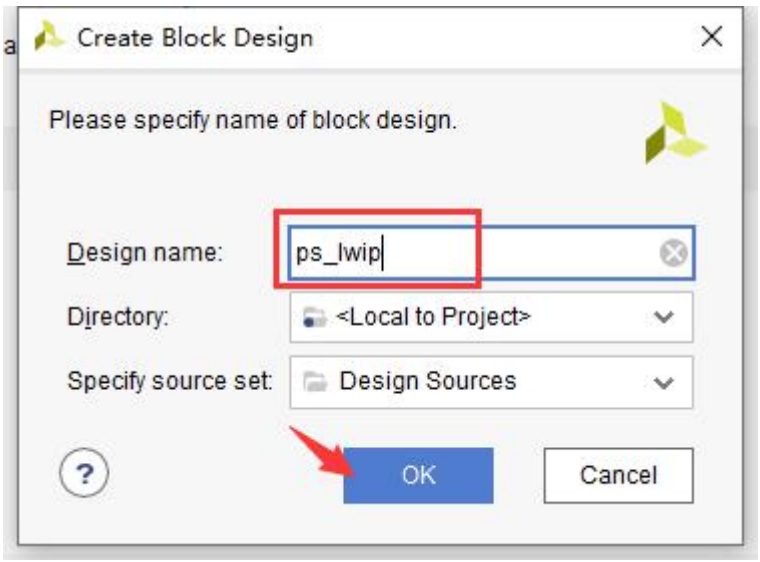
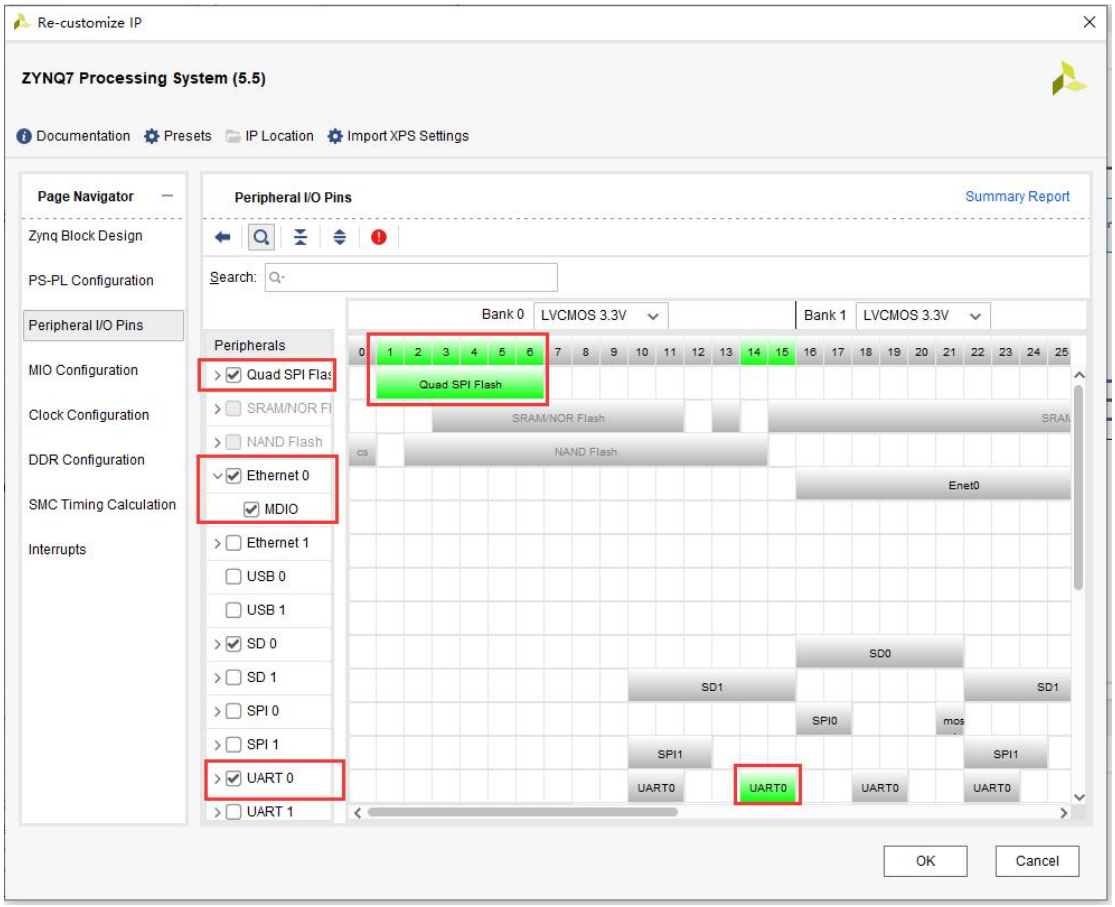
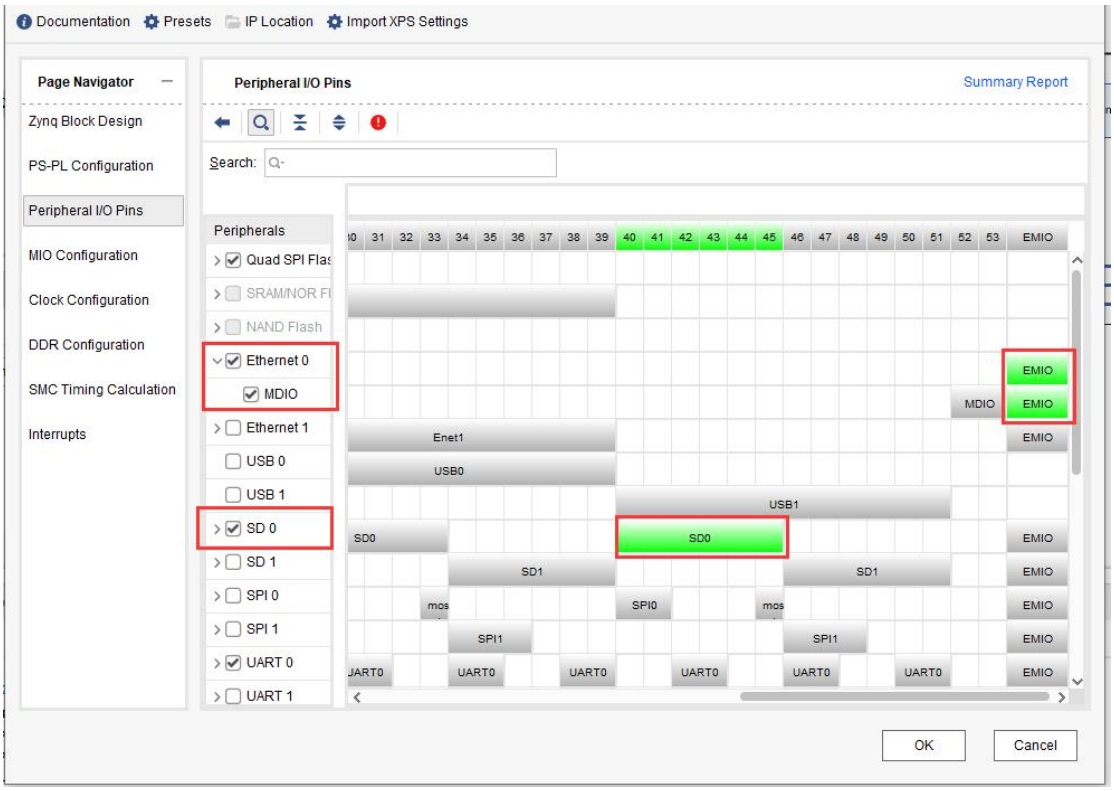
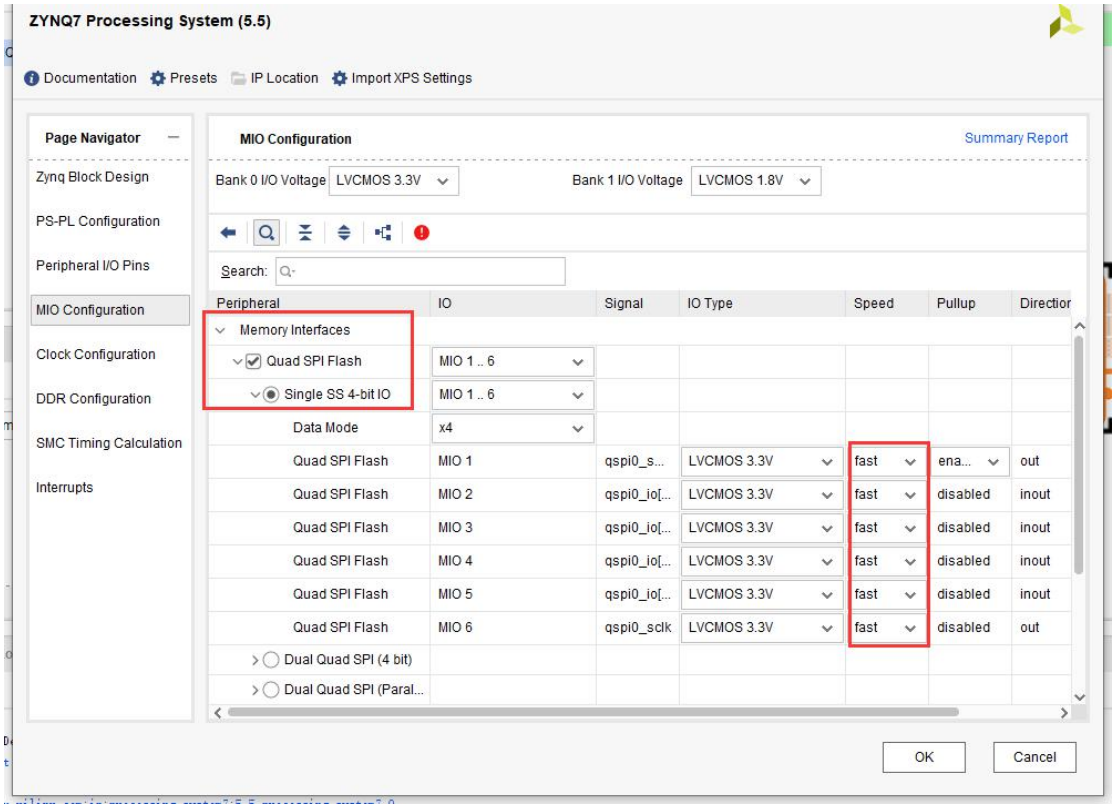


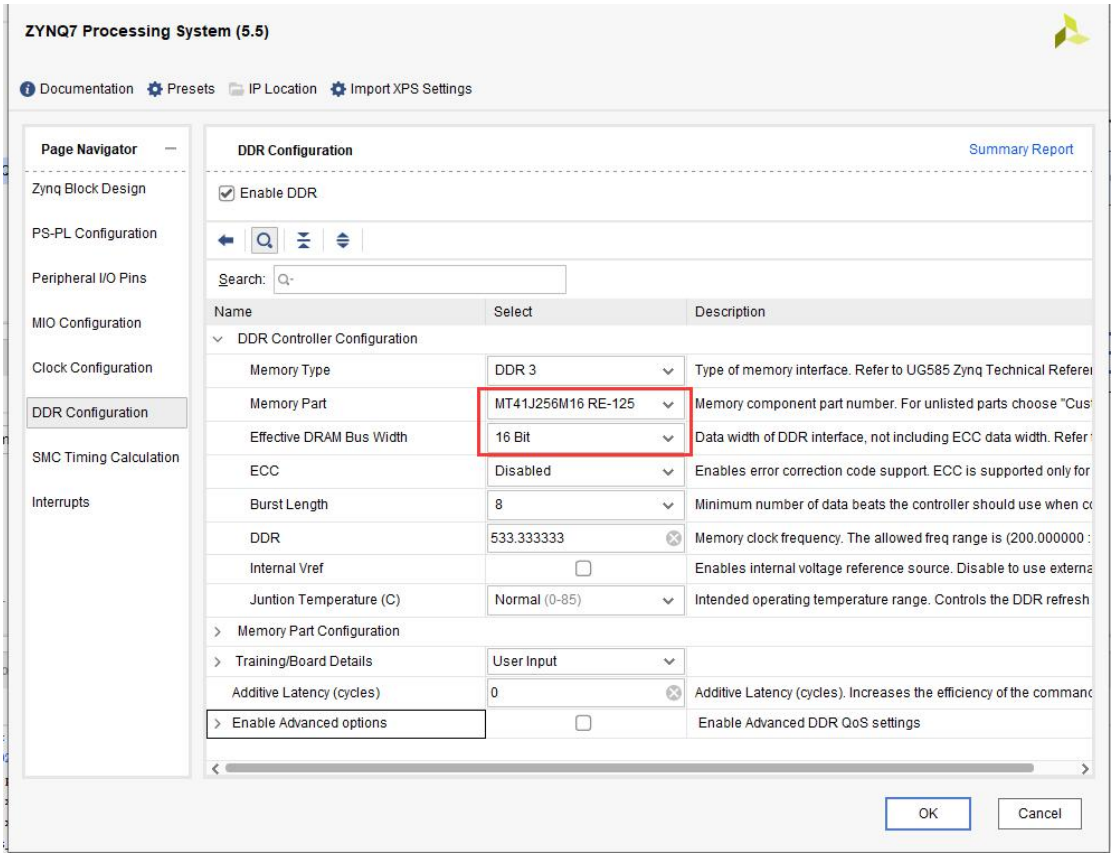
图 10.2.2 创建 Block Design

如下图所示配置 ZYNQ7 Processing System 相关设置。




Documentation
Presets
IP Location
Import XPS Settings
Page Navigator
Zynq Block Design
PS-PL Configuration
Peripheral I/O Pins
MIO Configuration
Clock Configuration
DDR Configuration
SMC Timing Calculation
Interrupts
Peripheral I/O Pins
Summary Report
Search:
Peripherals
Quad SPI Flas
SRAM/NOR Fl
NAND Flash
Ethernet 0
MDIO
Ethernet 1
USB 0
USB 1
SD 0
SD 1
SPI 0
SPI 1
UART 0
UART 1
EMIO
MDIO
Enet1
USB0
USB1
SD0
SD1
SPI0
SPI1
UART0
OK
Cancel

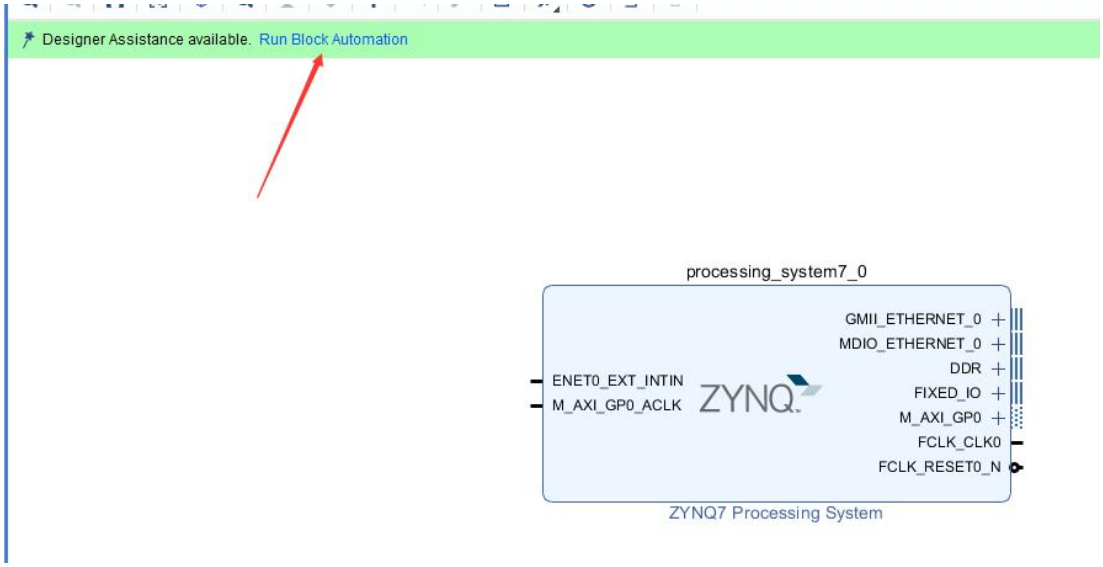


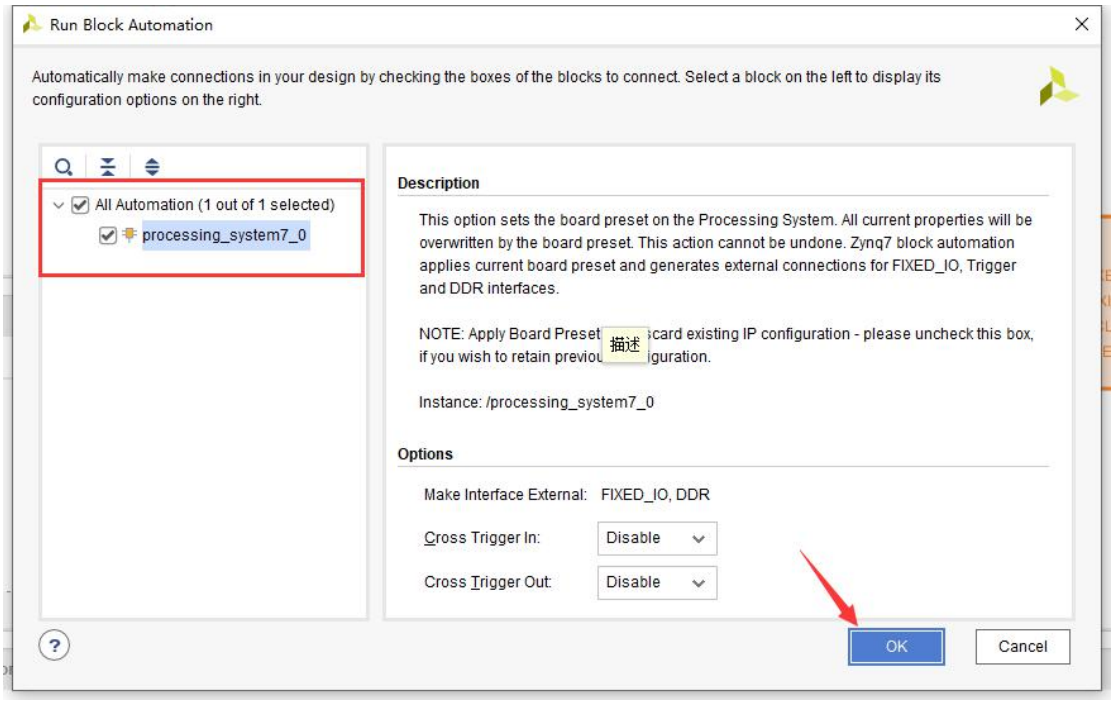


将 QSPI 的引脚速度全部改为 fast。

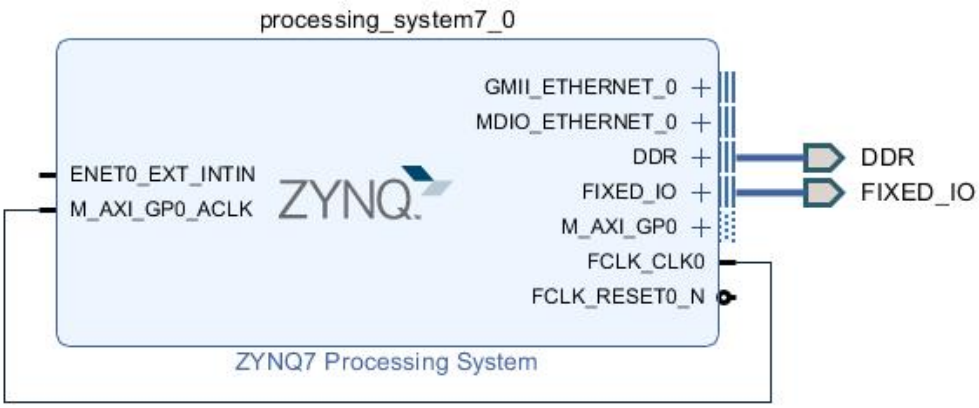
配置 DDR3 控制器。配置完成点击“OK”。

在 Diagram 界面点击 Run Block Automation，在弹出的对话框中选择全部接口，然后点击“OK”。

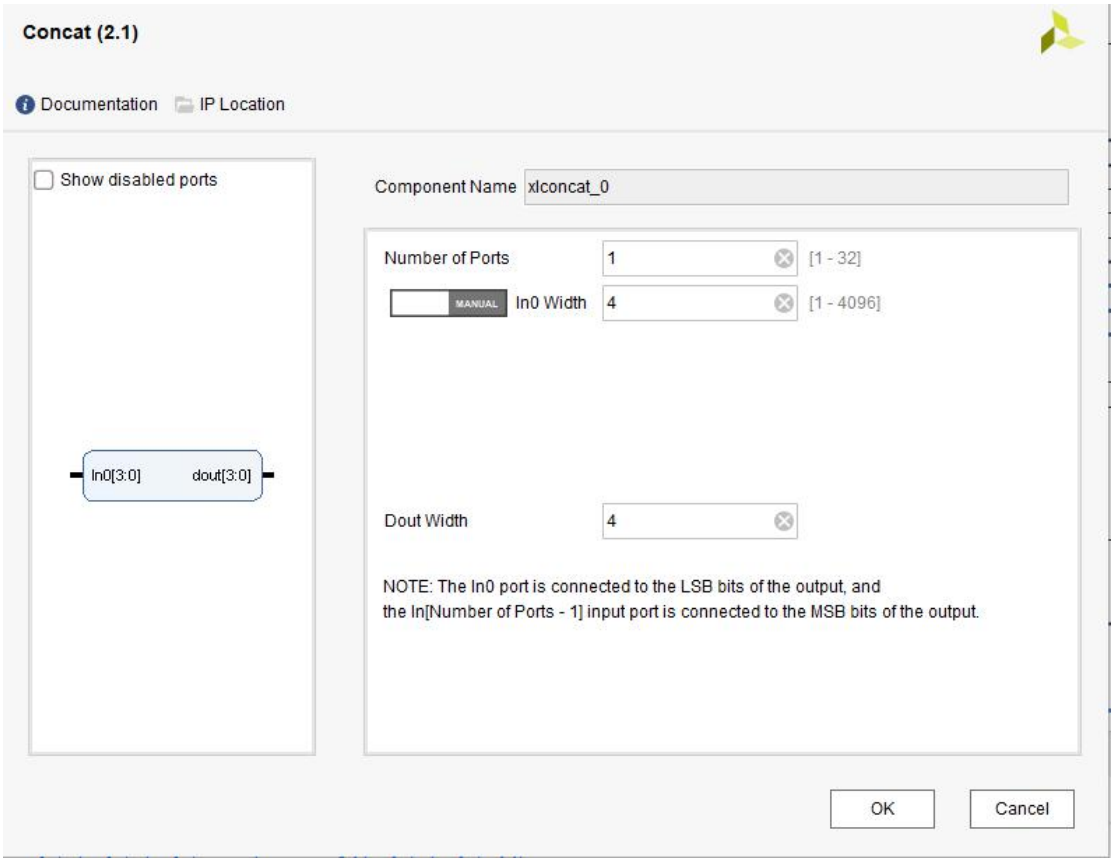


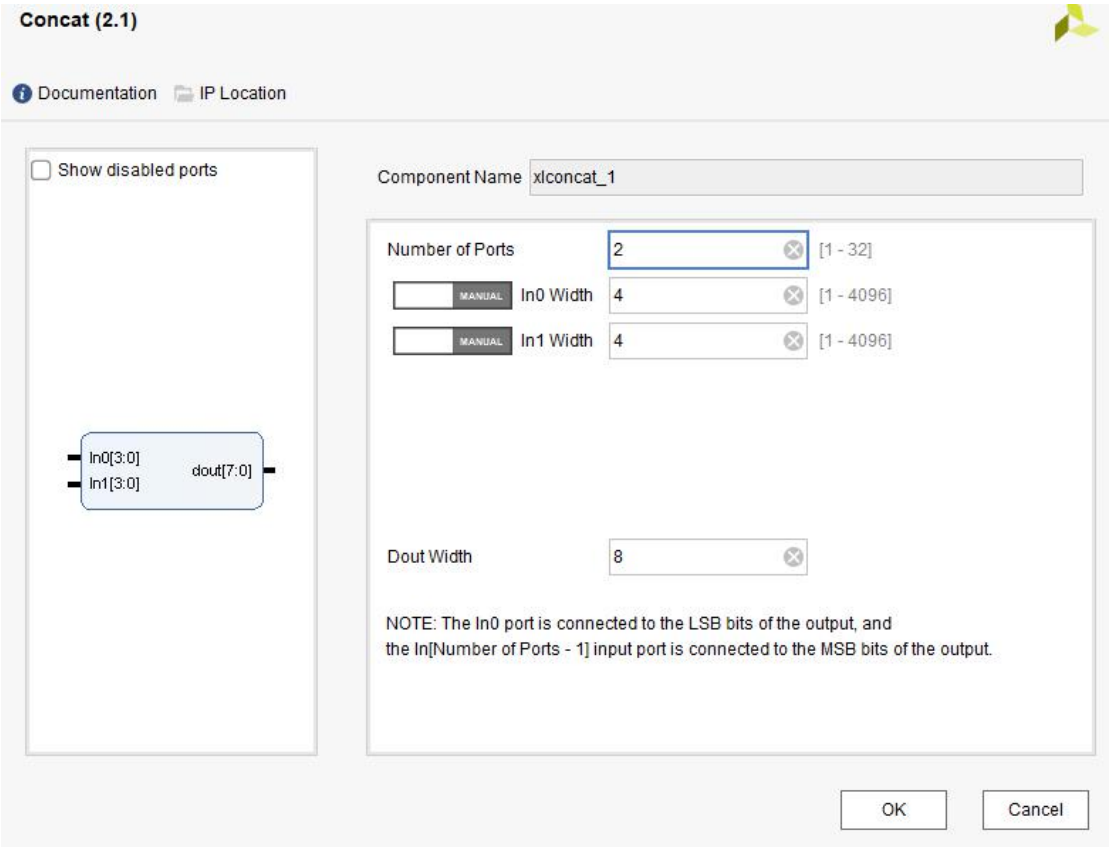


连接完成后手动连接 M_AXI_GP0_ACLK 和 FCLK_CLK0。

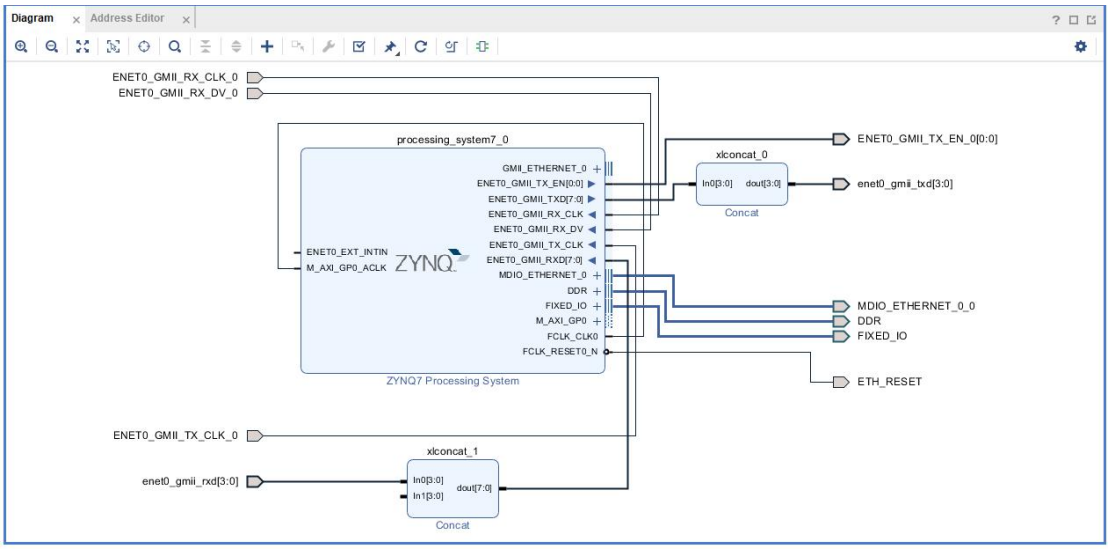


添加两个 concat IP 核，因为在开发板上使用的 phy 芯片的接口类型为 mii 接口，数据接口 为 4 位。而默认的网络接口为 8 位，需要进行调整。添加完成后 IP 设置如下图所示：





之后将相应的信号引出并连接相关 IP 的端口，信号连接如下图所示。



连接完成后验证并保存设计。之后生成输出文件核顶层文件。

接下来进行引脚约束，具体约束如下图所示：

```
27 set_property PACKAGE_PIN J15 [get_ports MDIO_ETHERNET_0_0_mdio_io]
28 set_property PACKAGE_PIN G14 [get_ports MDIO_ETHERNET_0_0_mdc]
29
30 set_property IOSTANDARD LVCMOS33 [get_ports ETH_RESET]
31 set_property PACKAGE_PIN H20 [get_ports ETH_RESET]
32
33 set_property SLEW FAST [get_ports {enet0_gmii_txd[3]}]
34 set_property SLEW FAST [get_ports {enet0_gmii_txd[2]}]
35 set_property SLEW FAST [get_ports {enet0_gmii_txd[1]}]
36 set_property SLEW FAST [get_ports {enet0_gmii_txd[0]}]
37 set_property SLEW FAST [get_ports {ENET0_GMII_TX_EN_0[0]}]
38 set_property SLEW FAST [get_ports MDIO_ETHERNET_0_0_mdio_io]
39 set_property SLEW FAST [get_ports MDIO_ETHERNET_0_0_mdc]
```

```
1  set_property IOSTANDARD LVCMOS33 [get_ports {enet0_gmii_txd[3]}]
2  set_property IOSTANDARD LVCMOS33 [get_ports {enet0_gmii_txd[2]}]
3  set_property IOSTANDARD LVCMOS33 [get_ports {enet0_gmii_txd[1]}]
4  set_property IOSTANDARD LVCMOS33 [get_ports {enet0_gmii_txd[0]}]
5  set_property IOSTANDARD LVCMOS33 [get_ports {enet0_gmii_rxd[3]}]
6  set_property IOSTANDARD LVCMOS33 [get_ports {enet0_gmii_rxd[2]}]
7  set_property IOSTANDARD LVCMOS33 [get_ports {enet0_gmii_rxd[1]}]
8  set_property IOSTANDARD LVCMOS33 [get_ports {enet0_gmii_rxd[0]}]
9  set_property IOSTANDARD LVCMOS33 [get_ports ENET0_GMII_RX_DV_0]
10 set_property IOSTANDARD LVCMOS33 [get_ports {ENET0_GMII_TX_EN_0[0]}]
11 set_property IOSTANDARD LVCMOS33 [get_ports MDIO_ETHERNET_0_0_mdio_io]
12 set_property IOSTANDARD LVCMOS33 [get_ports MDIO_ETHERNET_0_0_mdc]
13 set_property IOSTANDARD LVCMOS33 [get_ports ENET0_GMII_RX_CLK_0]
14 set_property IOSTANDARD LVCMOS33 [get_ports ENET0_GMII_TX_CLK_0]
15 set_property PACKAGE_PIN K17 [get_ports ENET0_GMII_RX_CLK_0]
```

引脚约束完成之后保存并生成 bitstream。最后导出到 SDK 中即可。

10.3 软件设计

导出硬件描述文件后，在打开的 SDK 软件中，新建一个工程并命名为“ps_lwip”。如下图所示。

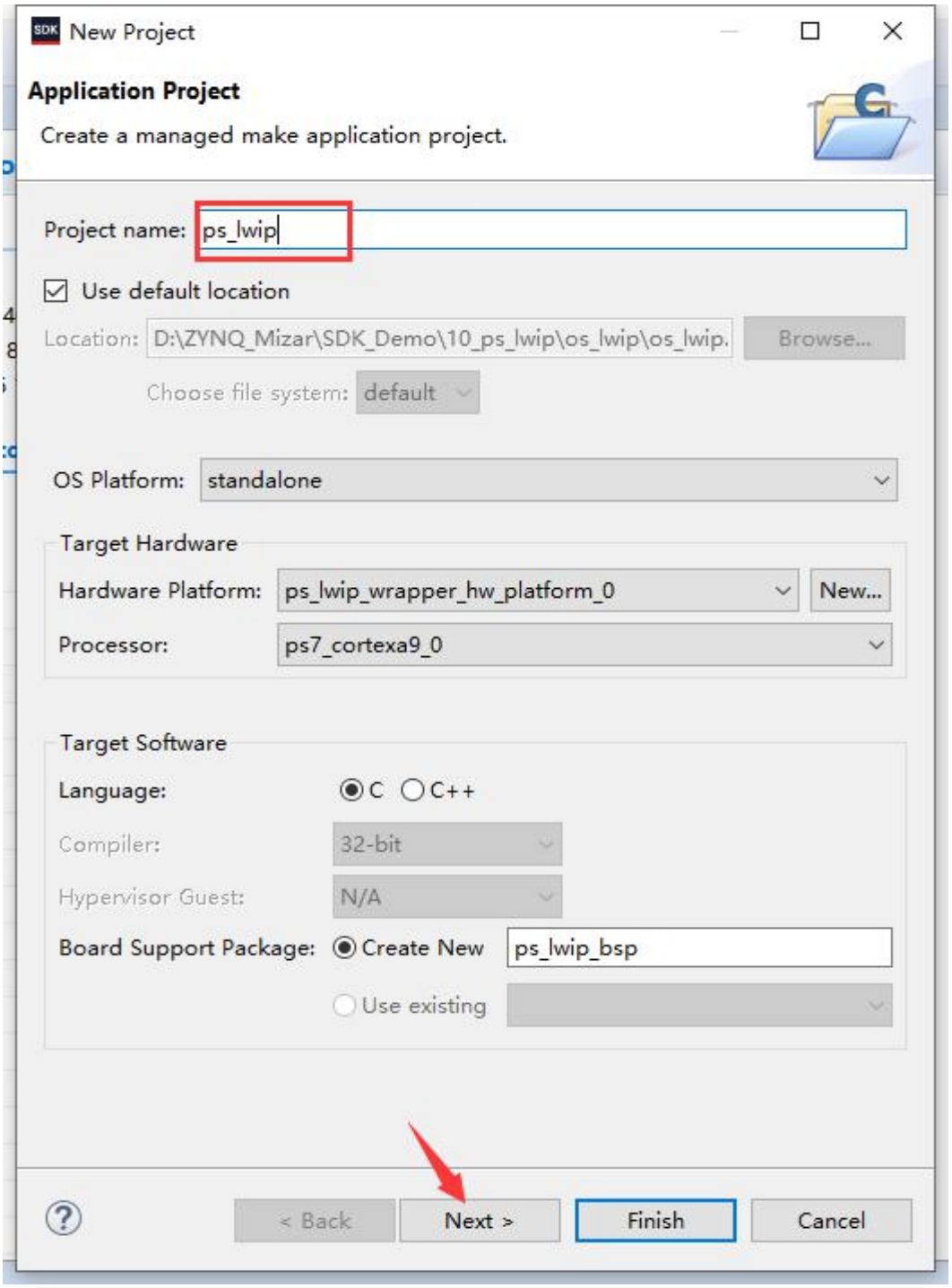
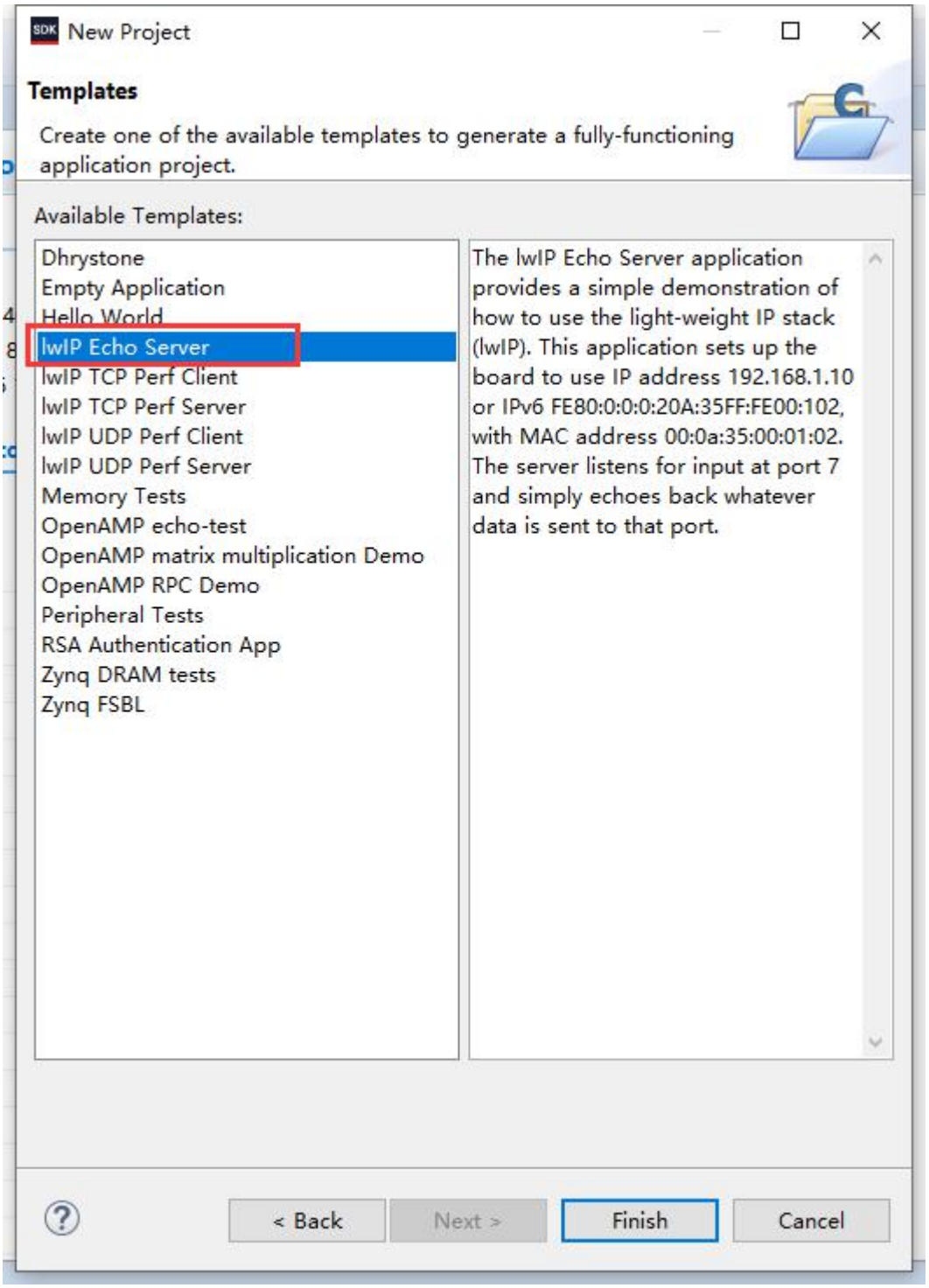


图 10.3.1 新建工程

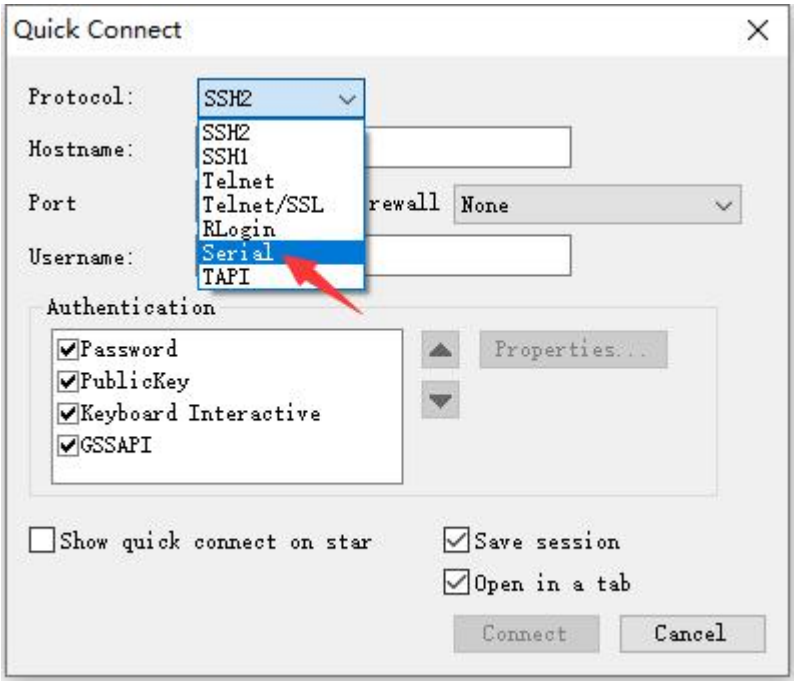
不同于之前的实验，我们这次选用 lwIP Echo Server 的工程模板。该模板将以太网设置为回环模式，将收到的所有数据全部发送回去。

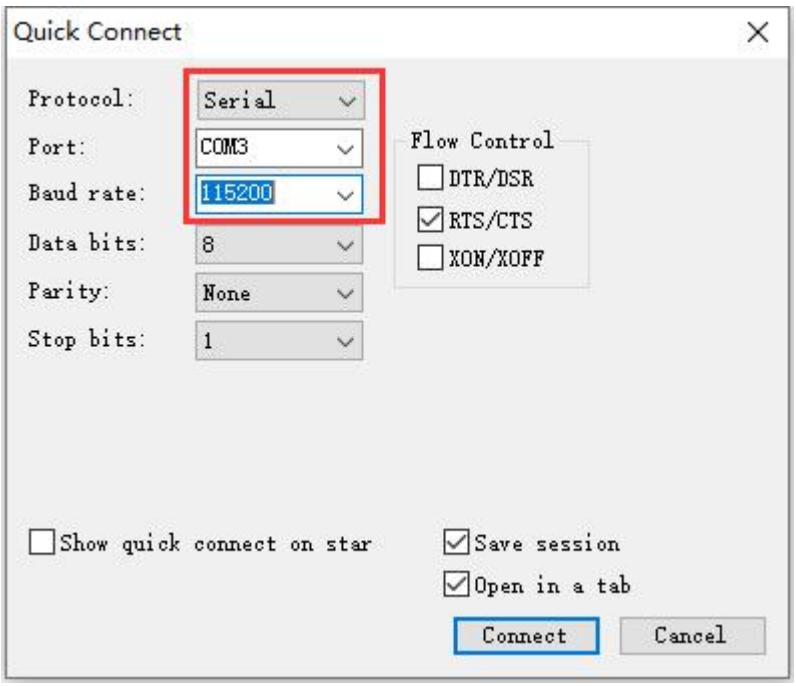


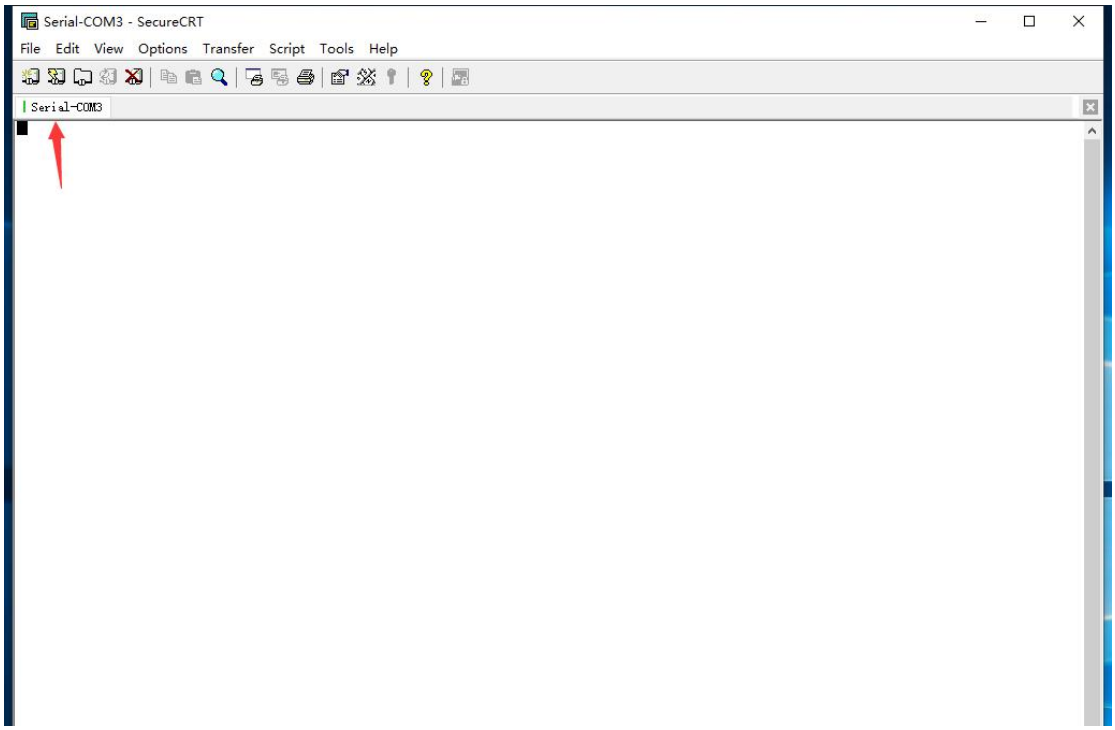
因为使用了标准工程模板，所以我们不需要对程序进行修改，直接编译下载即可。

10.4 下载验证

在烧录程序之前，首先连接开发板的串口，因为这次实验的需要，我们使用 SecureCRT5.5 这个串口程序，程序保存在资料里面 Tools_and_Driver 文件夹。打开程序会自动弹出快速连接的配置窗口，按照下图所示配置即可连接开发板串口。

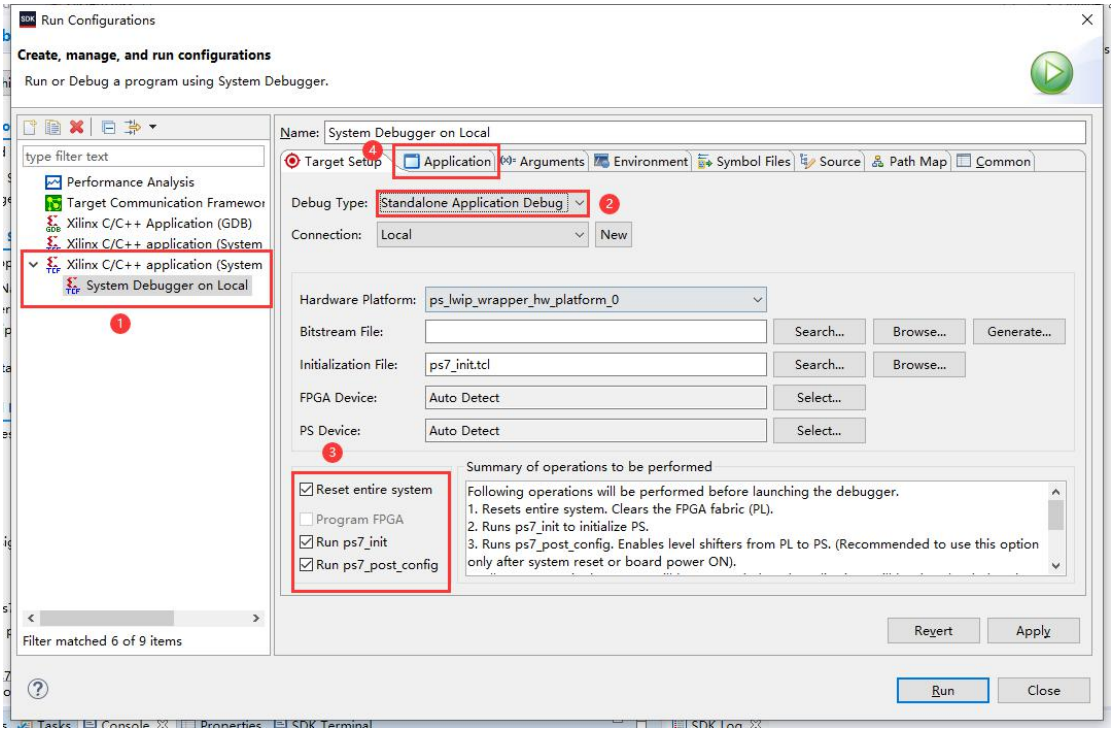


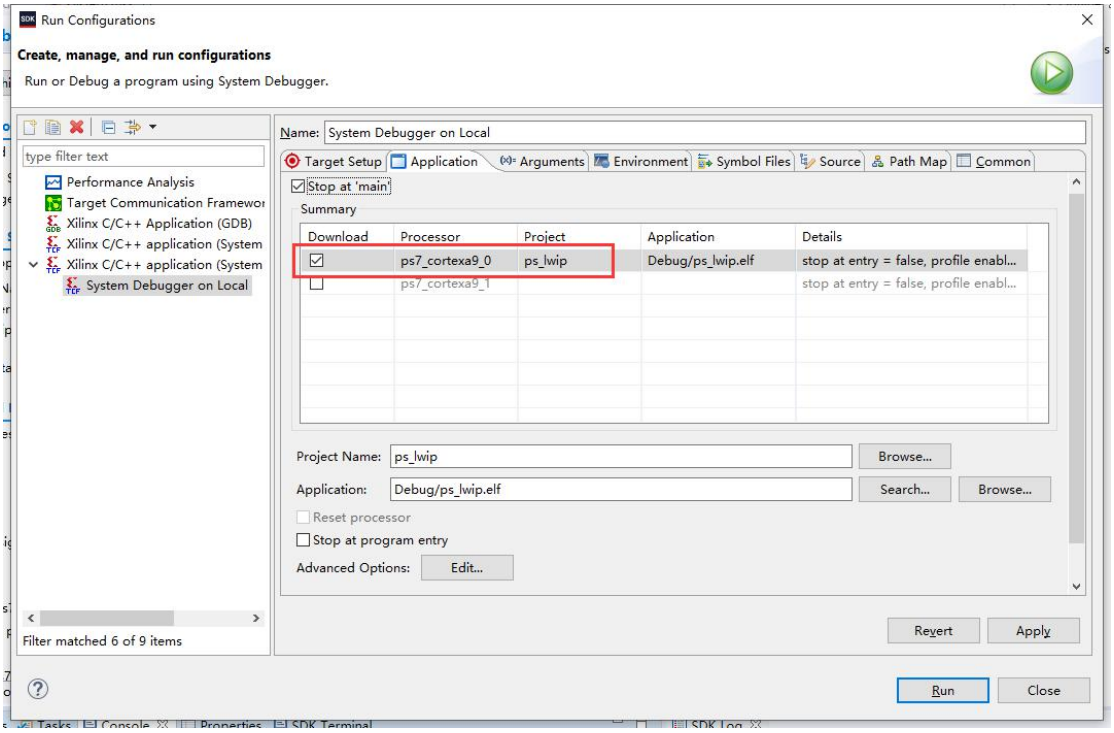




连接成功后在标签页前端会显示绿色，如下图所示：

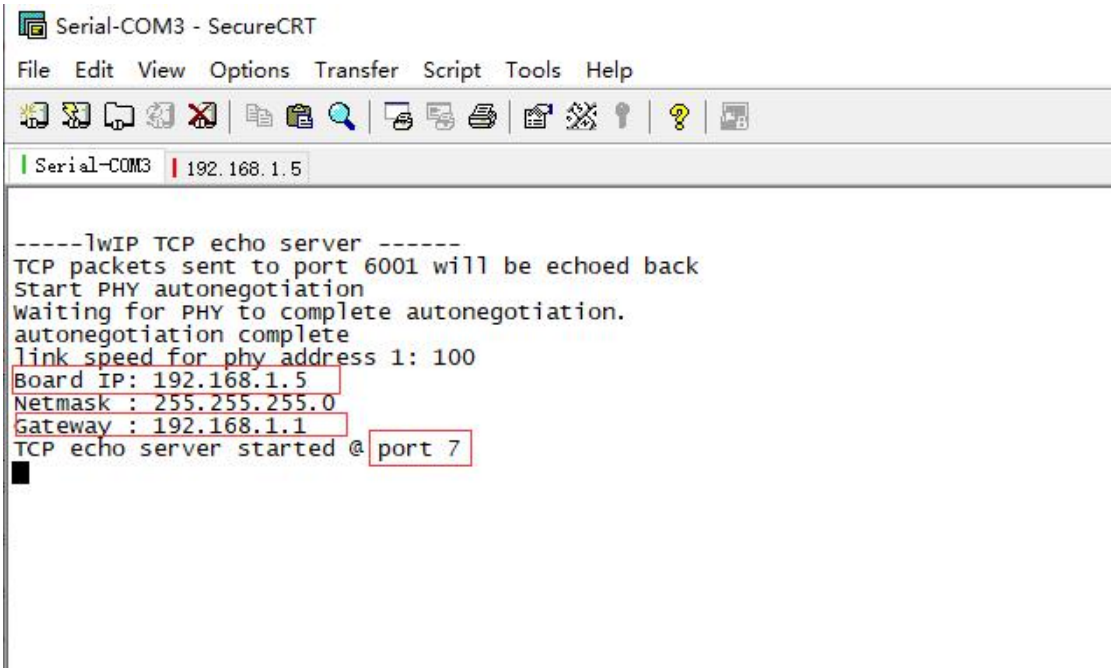
连接后即可烧录程序。菜单栏点击 Run->Run Configurations，在 tanchu 的界面双击 Xilinx C/C++ application（System Debugger），然后按照下图所示，按照图上所示顺序依次设置。



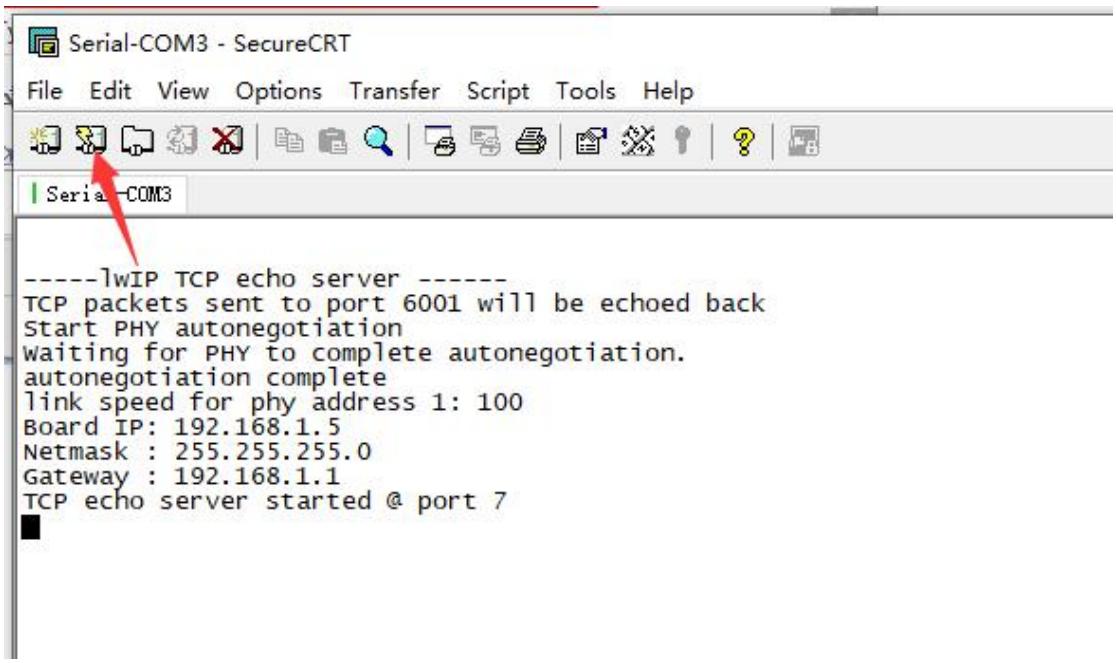


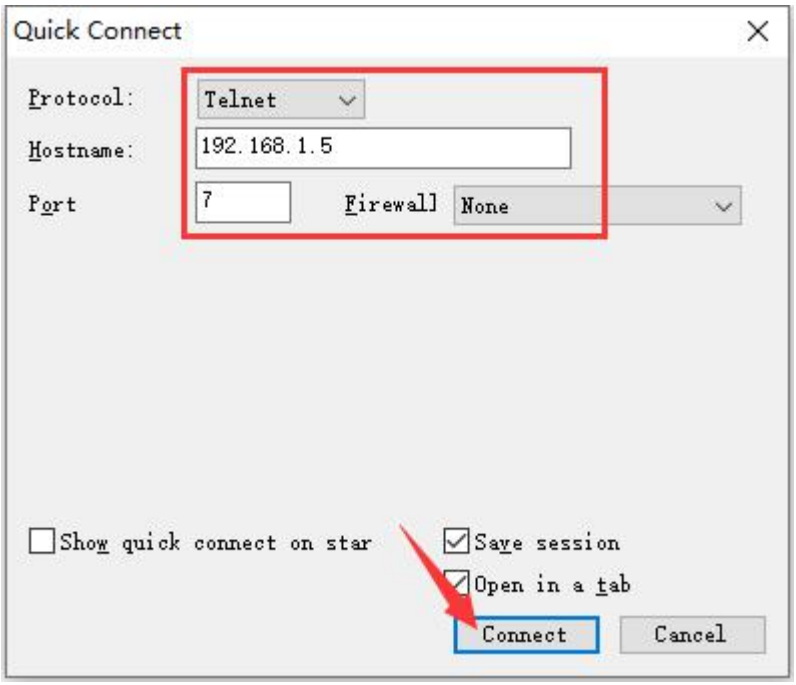
配置完成后点击“Run”即可烧录程序。烧录之前按照之前的操作连接号电源等连接线，然后用一根网线连接开发板网线接口和路由器接口。烧录完成后即可看到串口打印的信息如下图所示，从图中可以看到由路由器分配的开发板 IP 地址为 192.168.1.5，路由器 IP 为 192.168.1.1，端口号为 7。如果路由器未分配

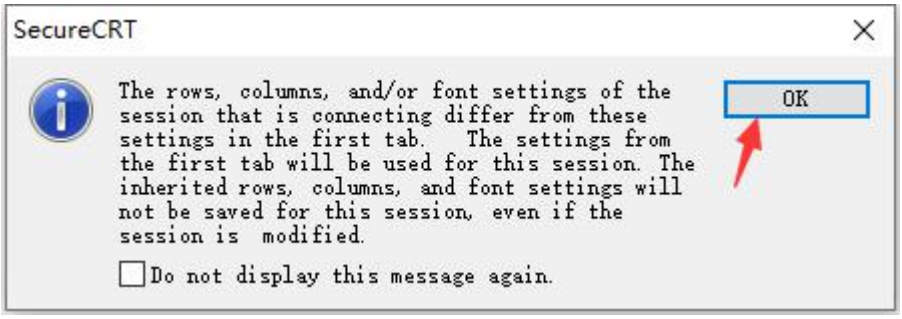
IP 地址则配置开发板 IP 为 192.168.1.10。



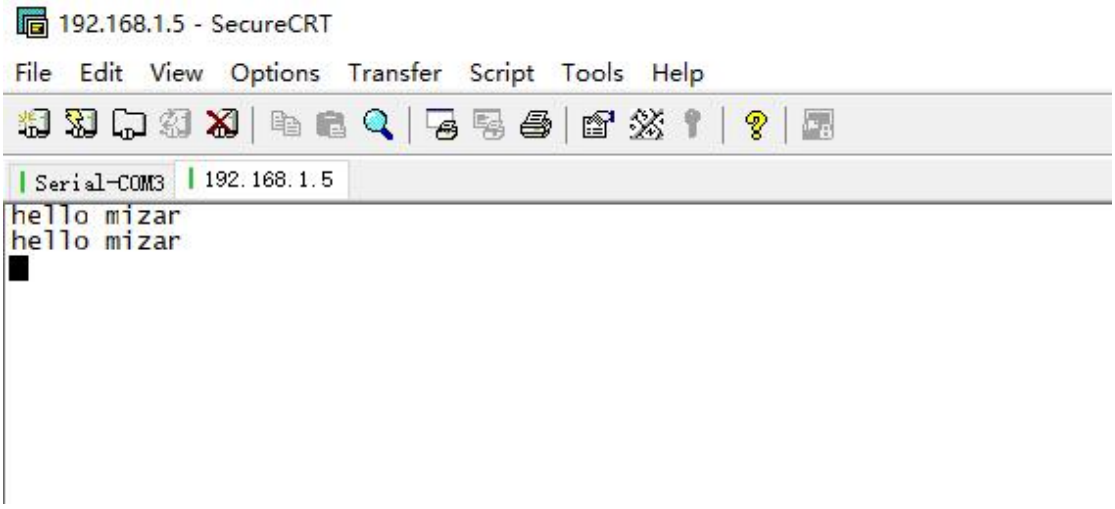
如果电脑和开发板连接同一台路由器，还可以测试电脑与开发板通过路由器通信。如下图所示点击打开快速连接窗口，配置 IP 地址等相关信息，点击“Connect”。弹出的对话框点击“OK”。





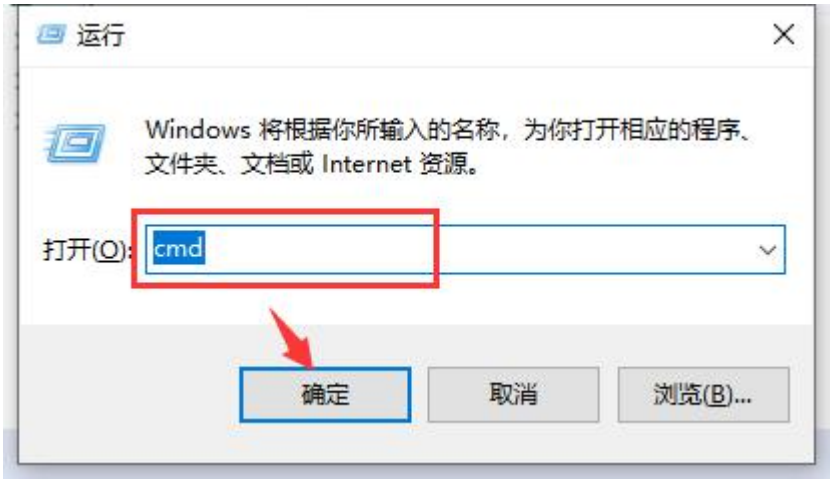


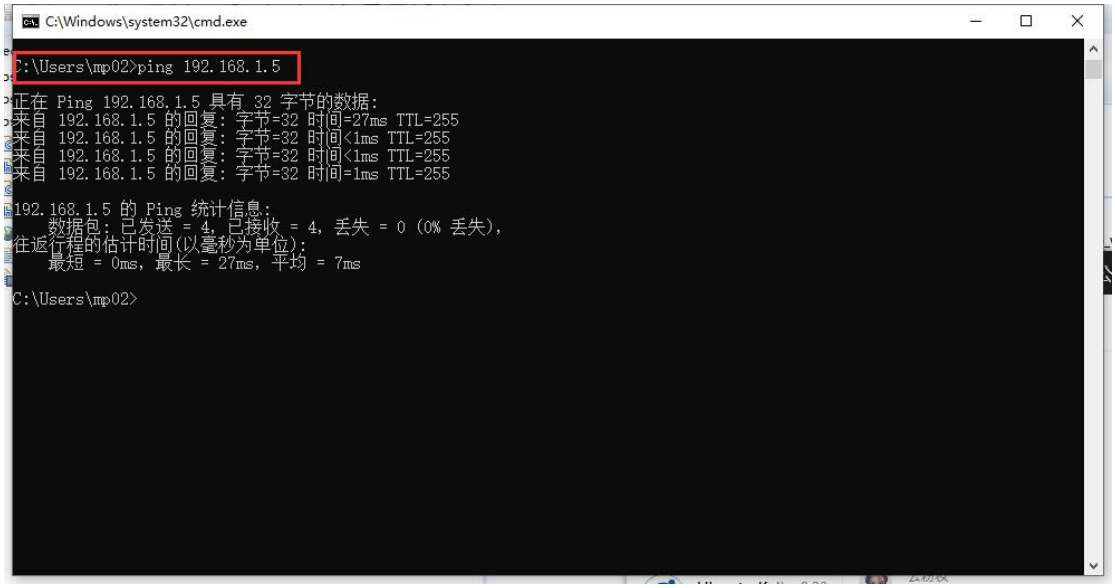
可以看到电脑和开发板已通过路由器连接，在对话框中输入“hello mizar”，



点击回车，可以看到开发板返回我们刚发送的数据，说明通信正常。

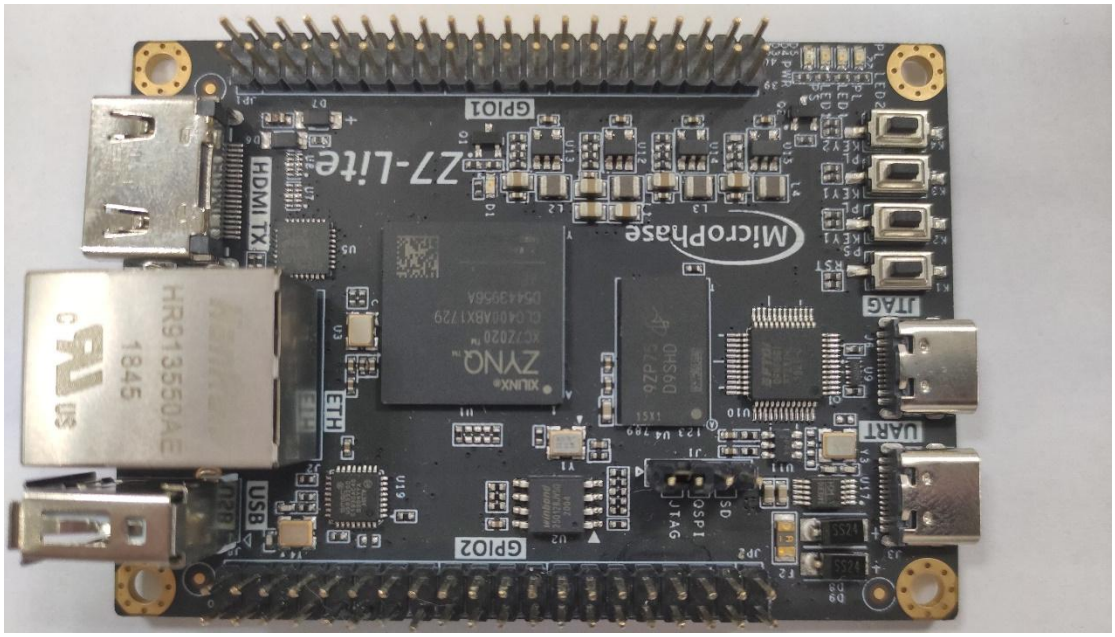
也可以通过命令行测试连接，快捷键 Windows + R 调出运行，输入 cmd 点击确定打开命令操作窗口。输入 ping 192.168.1.5 按下回车，可以看到如下信息，说明连接正常。



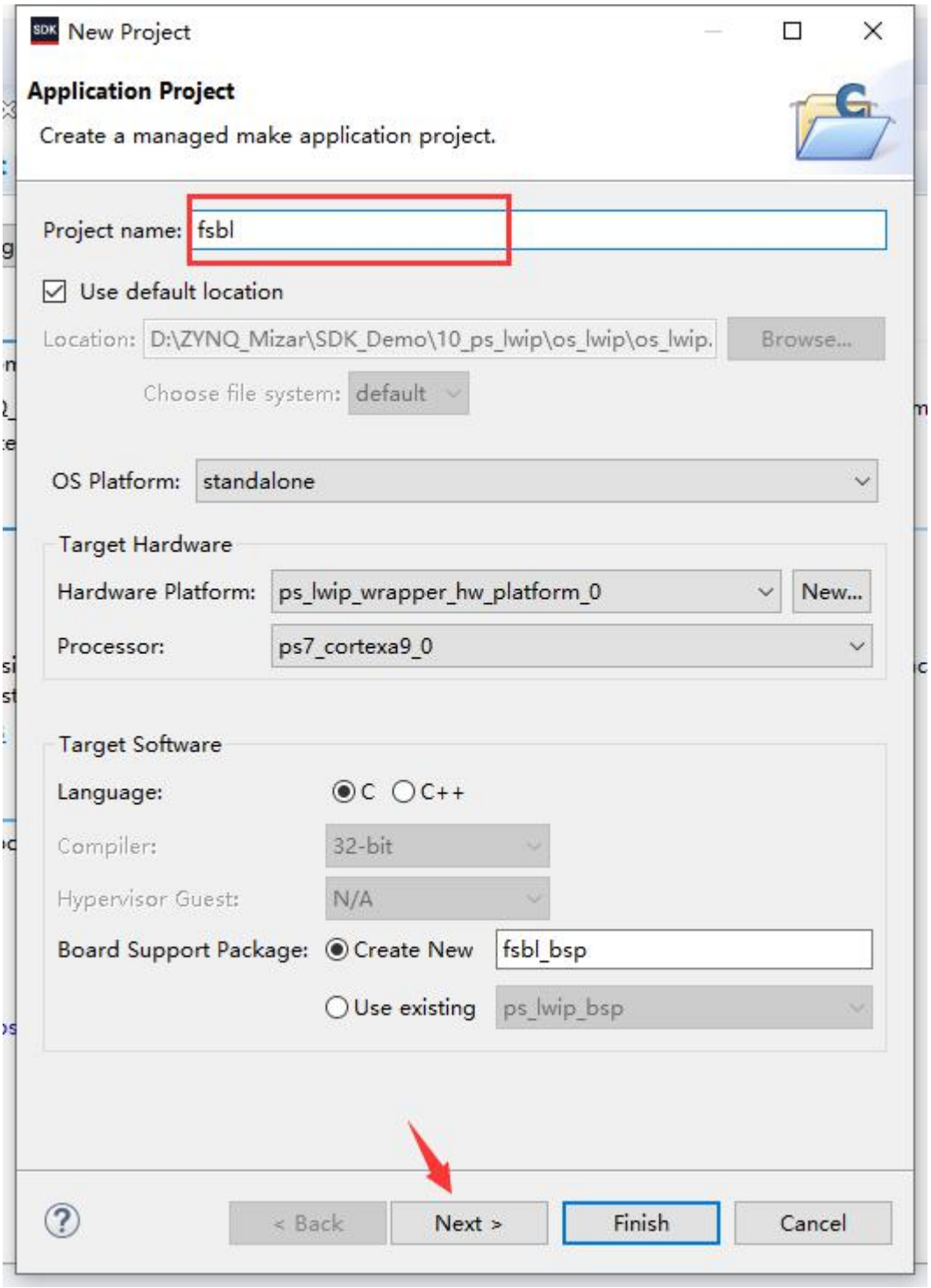


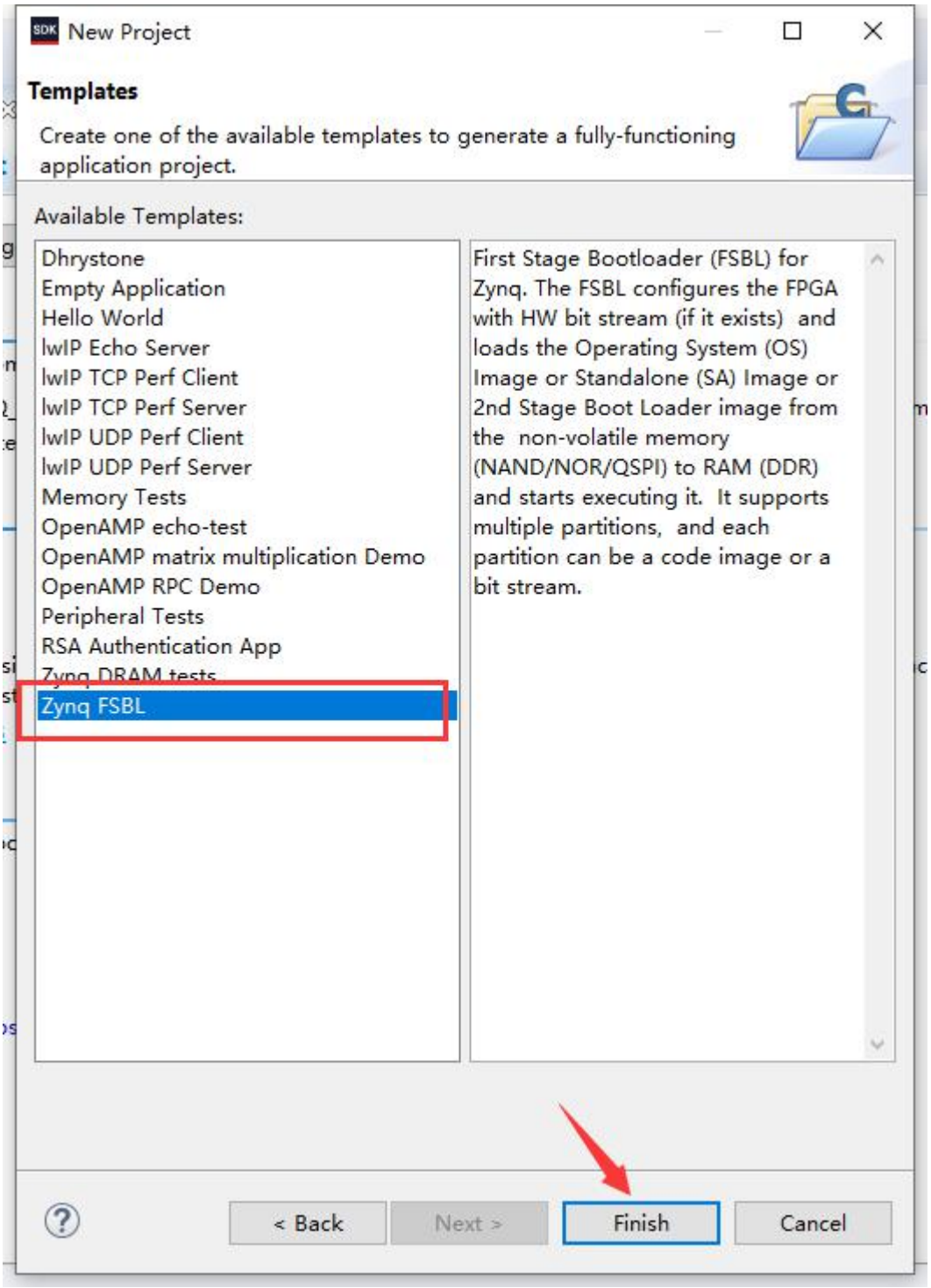
关于 lwip 实验相关的讲解到此为止，接下来我们讲解一下其他运行程序的方式。在之前的实验中，我们使用的都是 JTAG 的启动方式，程序在复位或掉电时丢失，每次重新运行都需要重新烧录，这种方式对于量产产品显然是不合适，这就需要其他不会丢失程序的启动方式。在开发板上有设置启动方式的排针，可以看到有三种启动方式，分别为：1、JTAG 启动；2、QSPI 启动；3、SD 卡启动。接下来我们讲解其他两种启动方式的操作方法。这两种启动方式都需要硬件支持，

我们在硬件设计时已经使能了 QSPI 和 SD0。

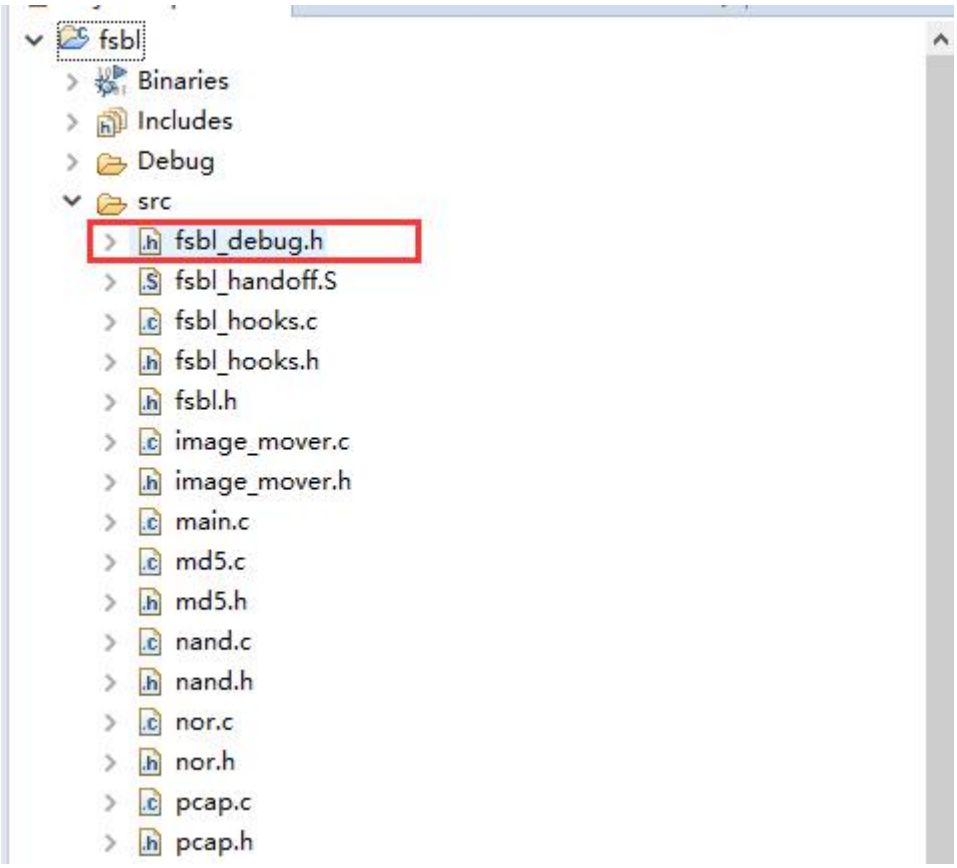


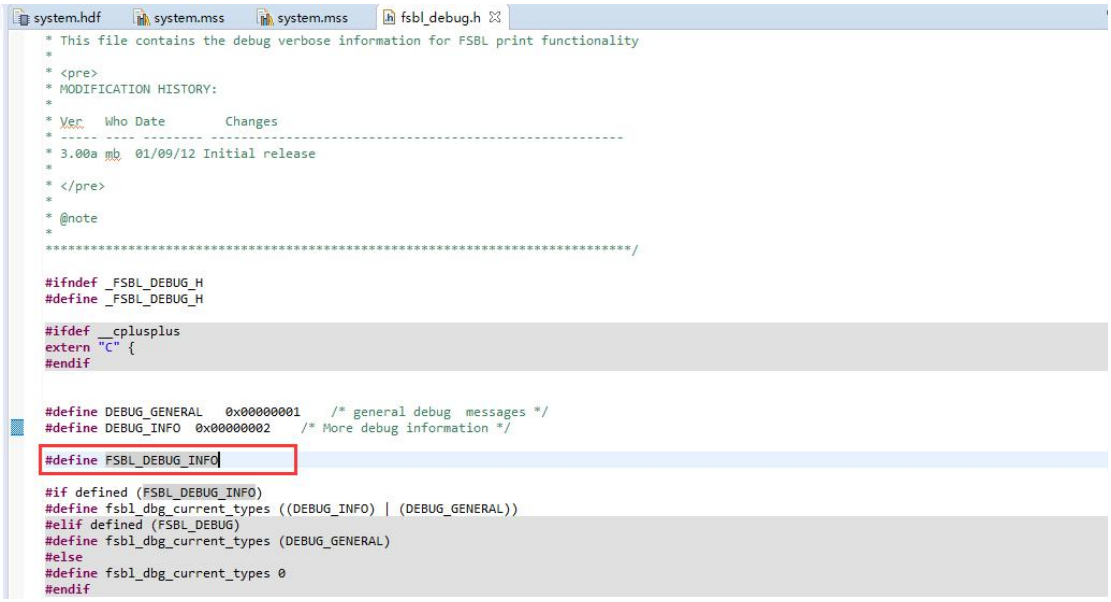
首先需要生成烧录文件，之后通过不同的方法烧录到 flash 芯片，SD 卡等介质上。创建一个新的工程命名为“fsbl”，工程模板选择 Zynq FSBL。


New Project
Application Project
Create a managed make application project.
Project name: fsbl
Use default location
Location: D:\ZYNQ_Mizar\SDK_Demo\10_ps_lwip\os_lwip\os_lwip.
Browse...
Choose file system: default
OS Platform: standalone
Target Hardware
Hardware Platform: ps_lwip_wrapper_hw_platform_0
New...
Processor: ps7_cortexa9_0
Target Software
Language: C C++
Compiler: 32-bit
Hypervisor Guest: N/A
Board Support Package: Create New fsbl_bsp
Use existing ps_lwip_bsp
< Back
Next >
Finish
Cancel

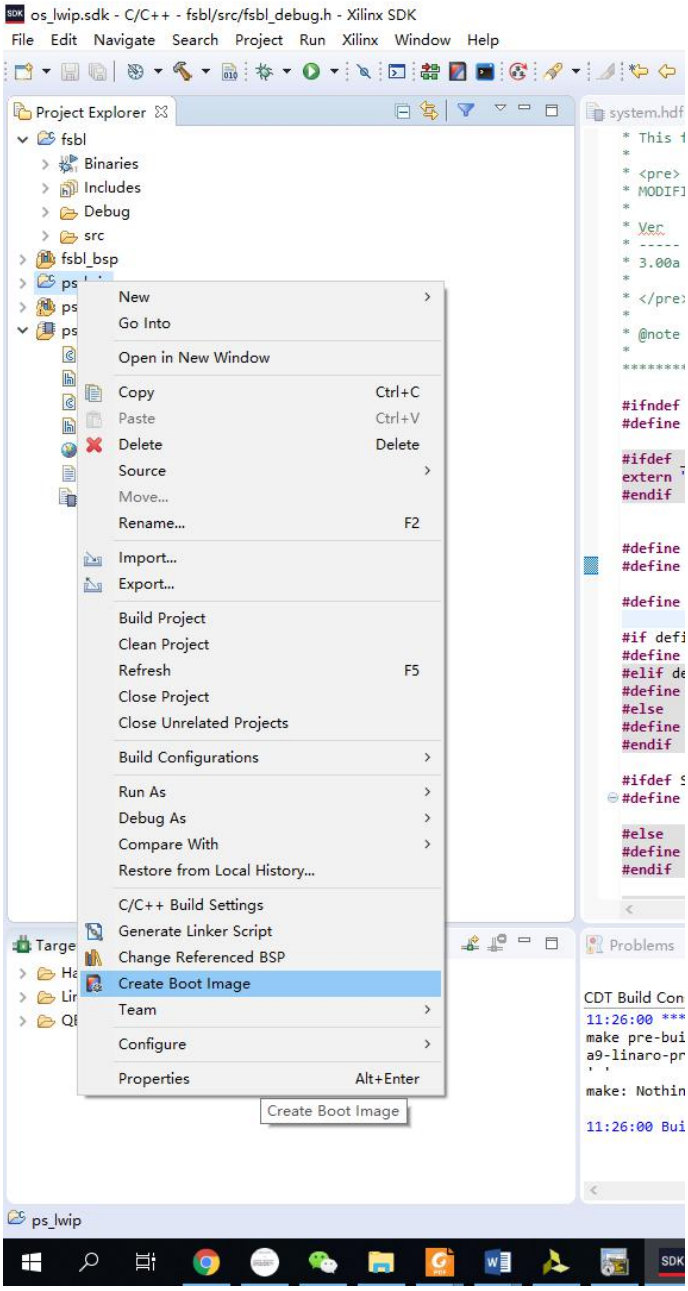

New Project
Templates
Create one of the available templates to generate a fully-functioning application project.
Available Templates:
Dhrystone
Empty Application
Hello World
lwIP Echo Server
lwIP TCP Perf Client
lwIP TCP Perf Server
lwIP UDP Perf Client
lwIP UDP Perf Server
Memory Tests
OpenAMP echo-test
OpenAMP matrix multiplication Demo
OpenAMP RPC Demo
Peripheral Tests
RSA Authentication App
Zynq DRAM tests
Zynq FSBL
First Stage Bootloader (FSBL) for Zynq. The FSBL configures the FPGA with HW bit stream (if it exists) and loads the Operating System (OS) Image or Standalone (SA) Image or 2nd Stage Boot Loader image from the non-volatile memory (NAND/NOR/QSPI) to RAM (DDR) and starts executing it. It supports multiple partitions, and each partition can be a code image or a bit stream.
< Back
Next >
Finish
Cancel

为了得到更多的调试信息，我们打开 fsbl_debug.h 文件，增加如图所示的宏定义。

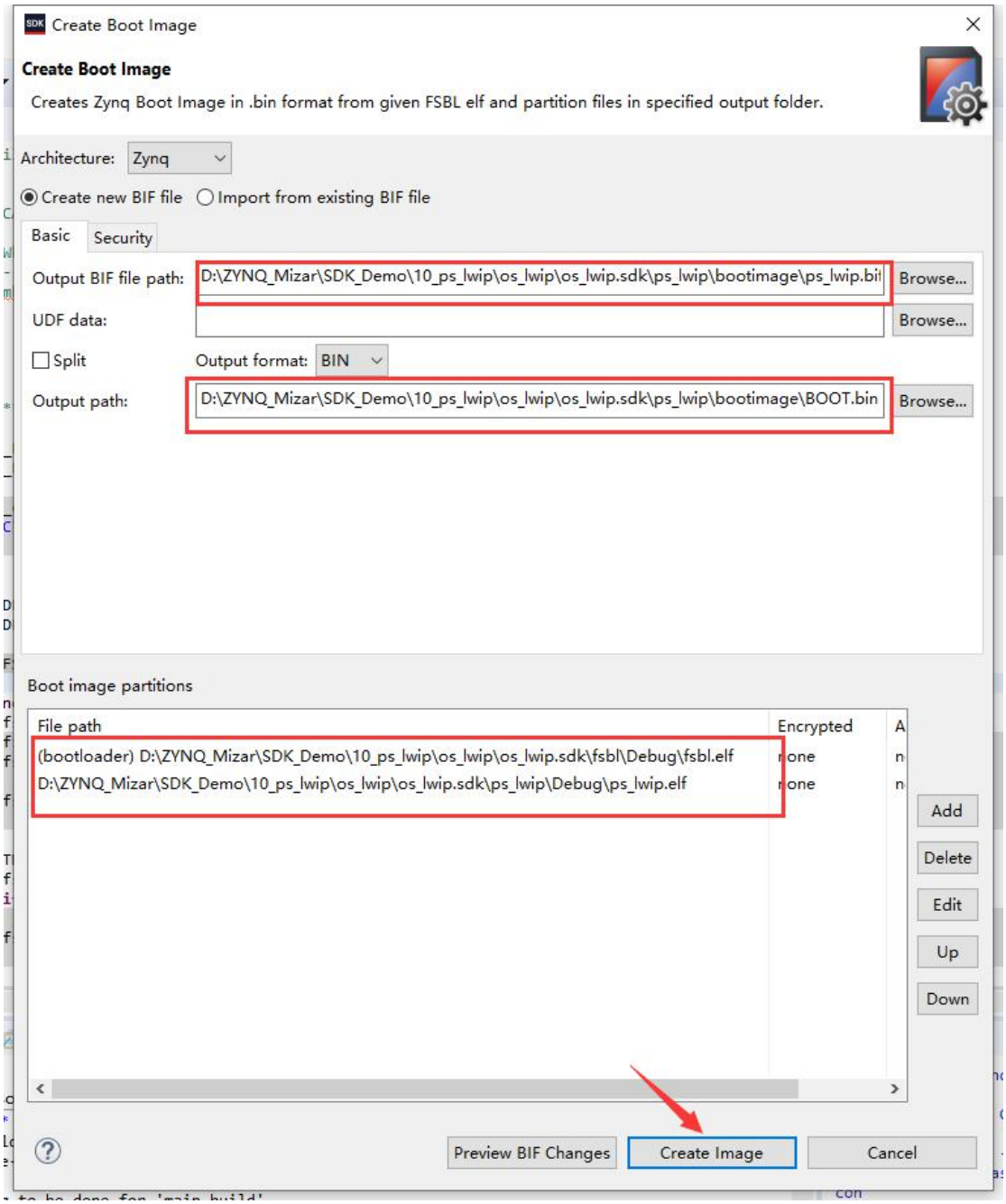




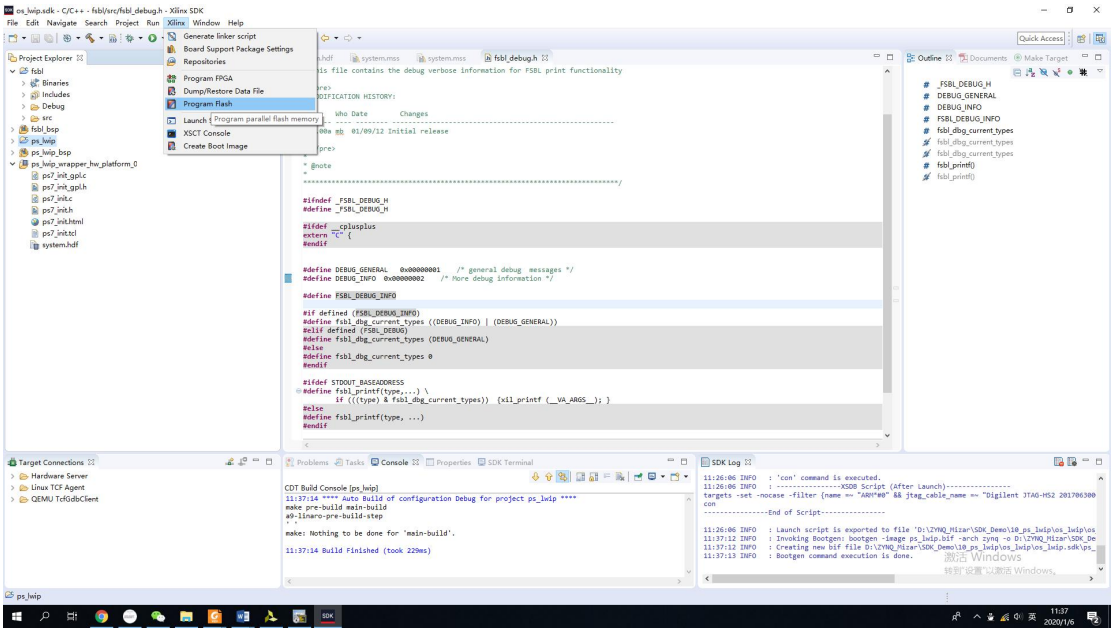
之后右键点击 ps_lwip，选择 Create Boot Image。



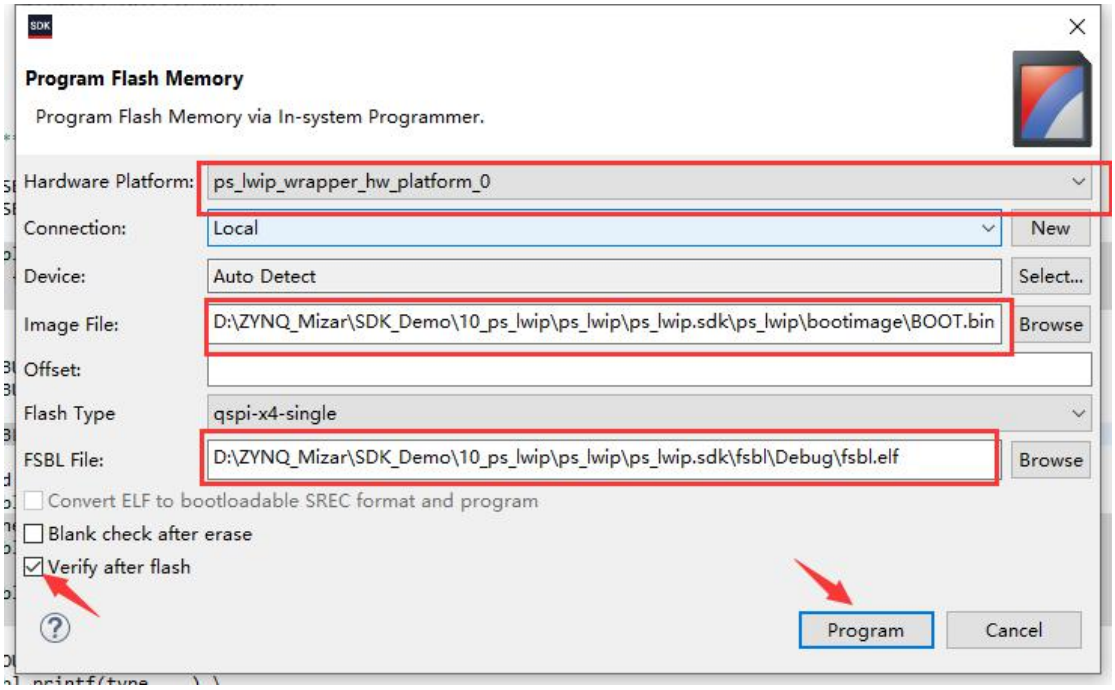
在弹出的对话框可以看到输出文件的保存路径和生成 BOOT 文件需要的原文件。点击“Create Image”。



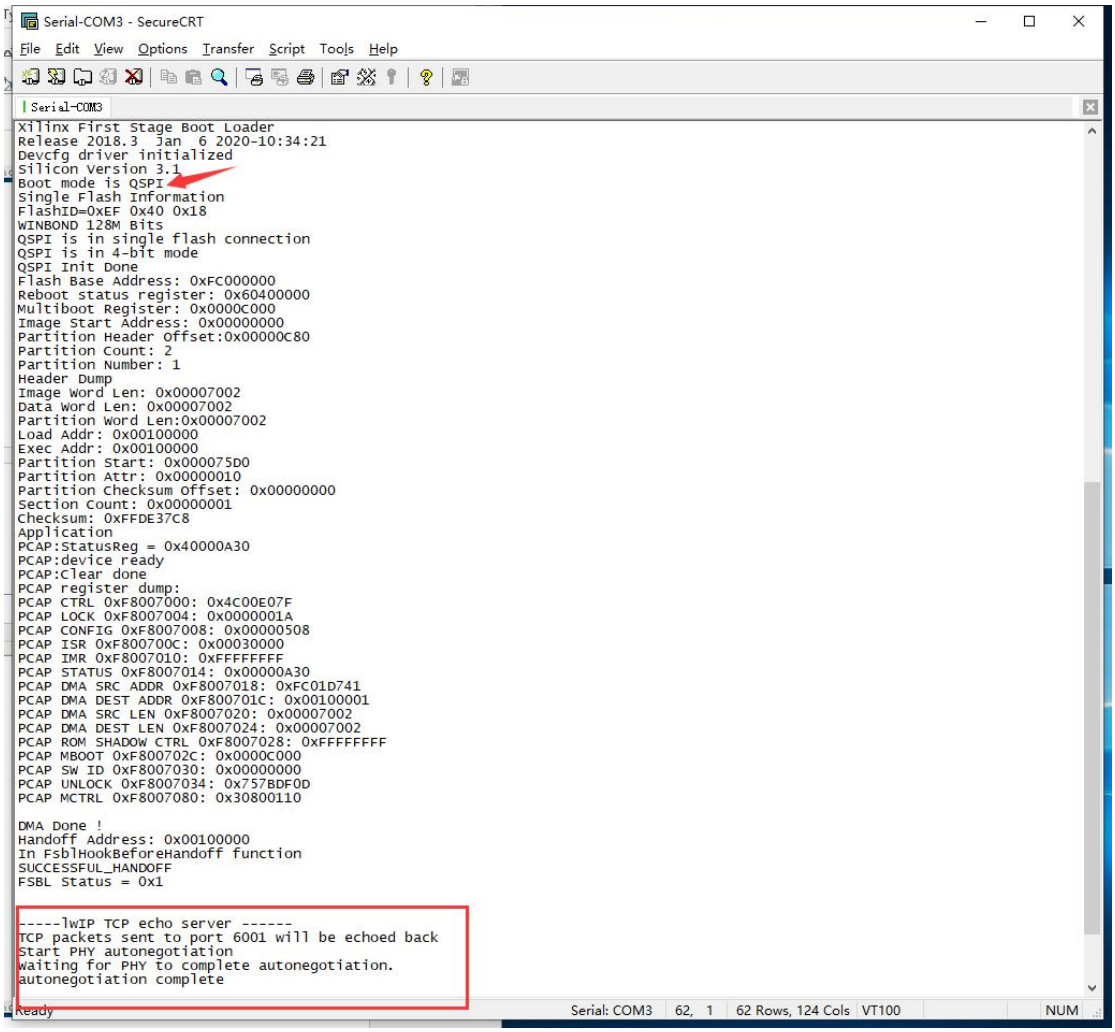
之后点击菜单栏 Xilinx，选择 Program Flash。



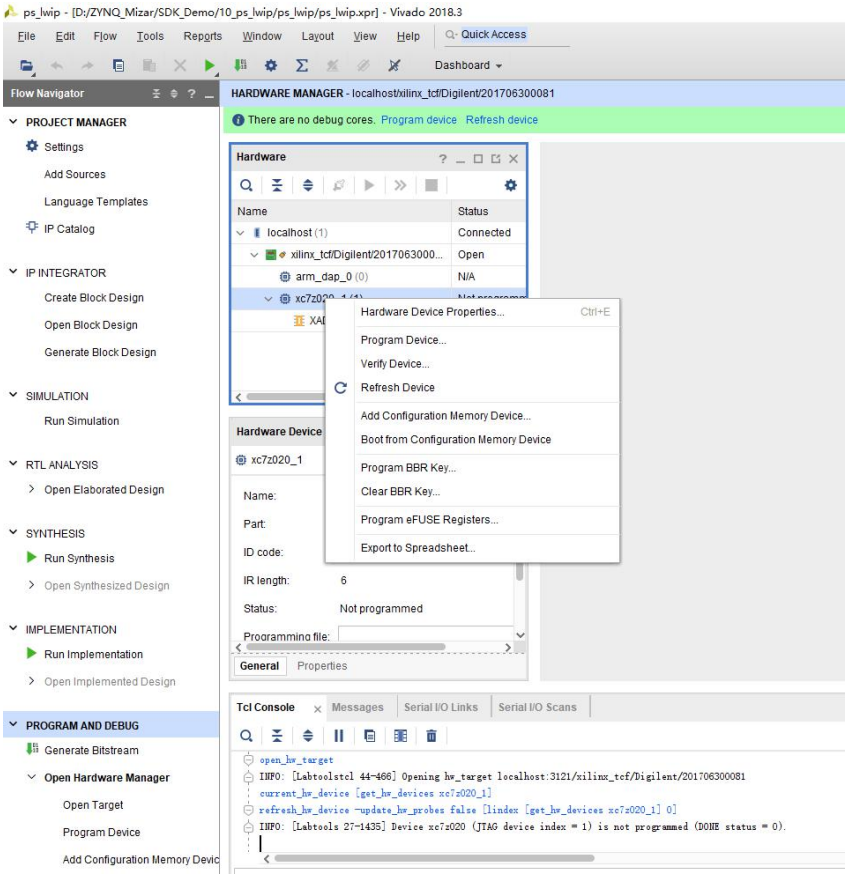
在弹出的对话框中按照下图所示配置。配置完成后点击 Program。



烧录完成后关闭电源，将拨码开关调整为 QSPI 启动，然后上电，可以看到串口打印的信息，前半部分为通过 QSPI 启动的信息。可以看到启动方式为 QSPI。启动完成后开始执行主程序，打印出 lwip 相关的信息。

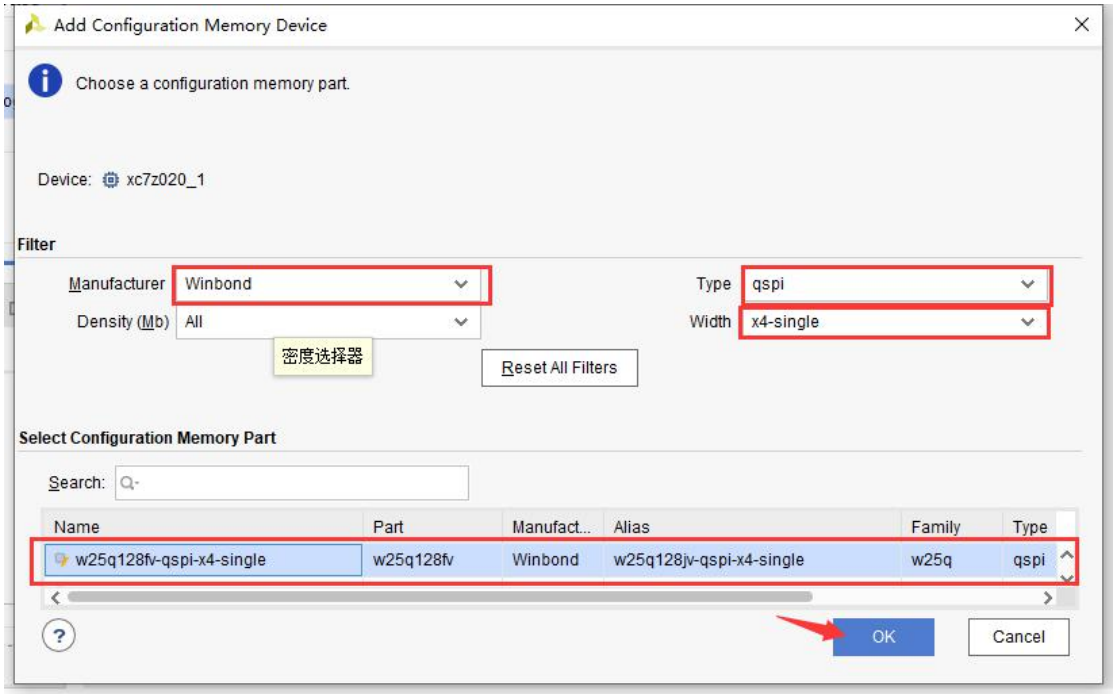


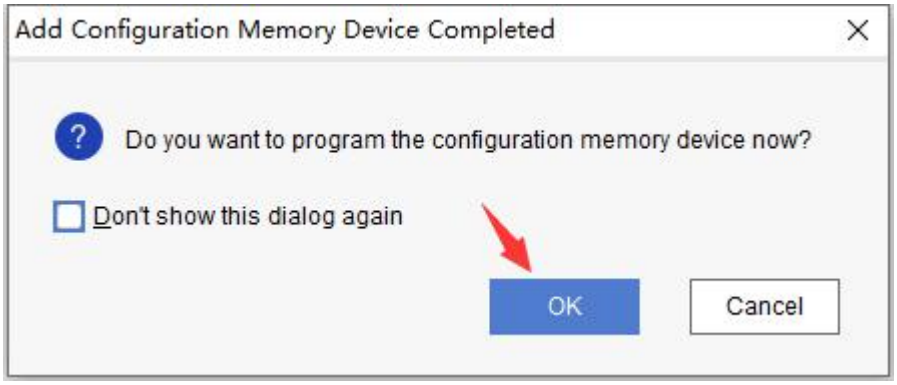
QSPI 启动的另一种下载方式为利用 Vivado 软件，点击 Open Hardware Manager->Open Target->Auto Connect，之后再 Hardware 窗口右键点击 xc7z020，选择 Add Configuration Memory Device。


ps_lwip - [D:/ZYNQ_Mizar/SDK_Demo/10_ps_lwip/ps_lwip/ps_lwip.xpr] - Vivado 2018.3
File Edit Flow Tools Reports Window Layout View Help
Quick Access
Dashboard
Flow Navigator
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
Open Target
Program Device
Add Configuration Memory Devic
HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/201706300081
There are no debug cores. Program device Refresh device
Hardware
Name
Status
localhost (1)
Connected
xilinx_tcf/Digilent/2017063000...
Open
arm_dap_0 (0)
N/A
Hardware Device Properties... Ctrl+E
Program Device...
Verify Device...
Refresh Device
Add Configuration Memory Device...
Boot from Configuration Memory Device
Program BBR Key...
Clear BBR Key...
Program eFUSE Registers...
Export to Spreadsheet...
Hardware Device
xc7z020_1
Name:
Part:
ID code:
IR length: 6
Status: Not programmed
Programming file:
General
Properties
Tcl Console
Messages
Serial I/O Links
Serial I/O Scans
open_hw_target
INFO: [Labtoolstcl 44-466] Opening hw_target localhost:3121/xilinx_tcf/Digilent/201706300081
current_hw_device [get_hw_devices xc7z020_1]
refresh_hw_device -update_hw_probes false [lindex [get_hw_devices xc7z020_1] 0]
INFO: [Labtools 27-1435] Device xc7z020 (JTAG device index = 1) is not programmed (DONE status = 0).

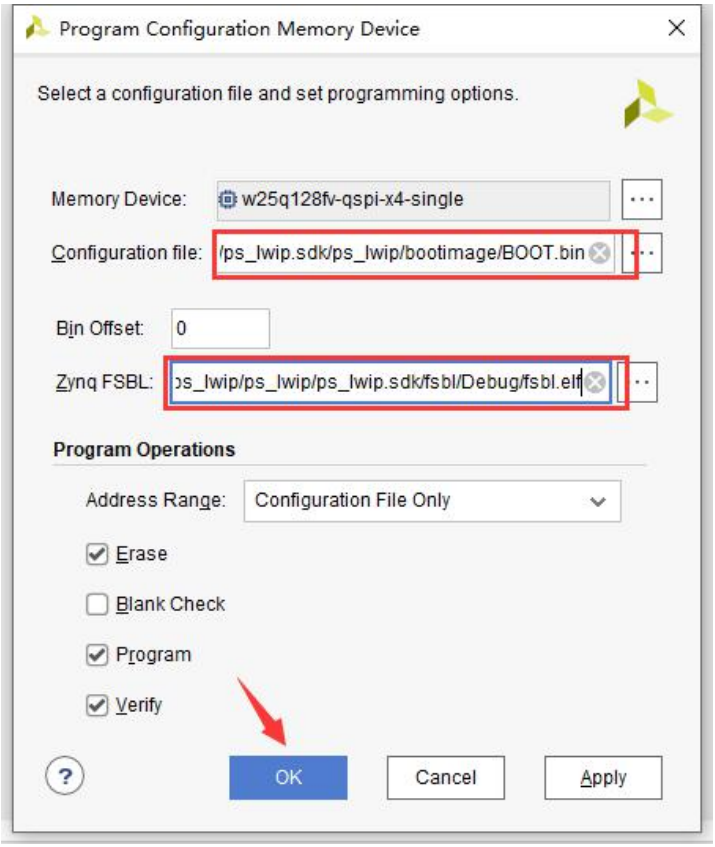
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager

在弹出的窗口按照下图所示顺序选择 Winbond->qspi->x4-single，下方会出现 W25Q128FV，选择该项，点击“OK”，在弹出的对话框中选择“OK”。



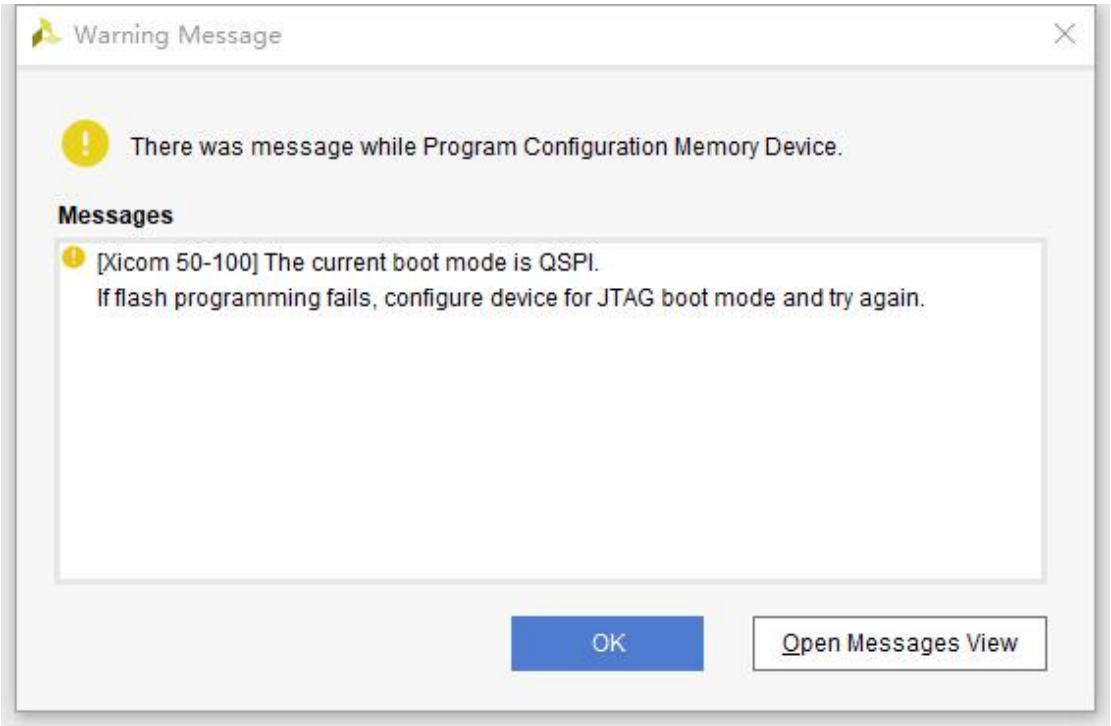


之后在弹出的配置页面如下图所示选择需要烧录的 BOOT.bin 文件和 fsbl.elf



文件的路径，设置完成后点击“OK”开始烧录。

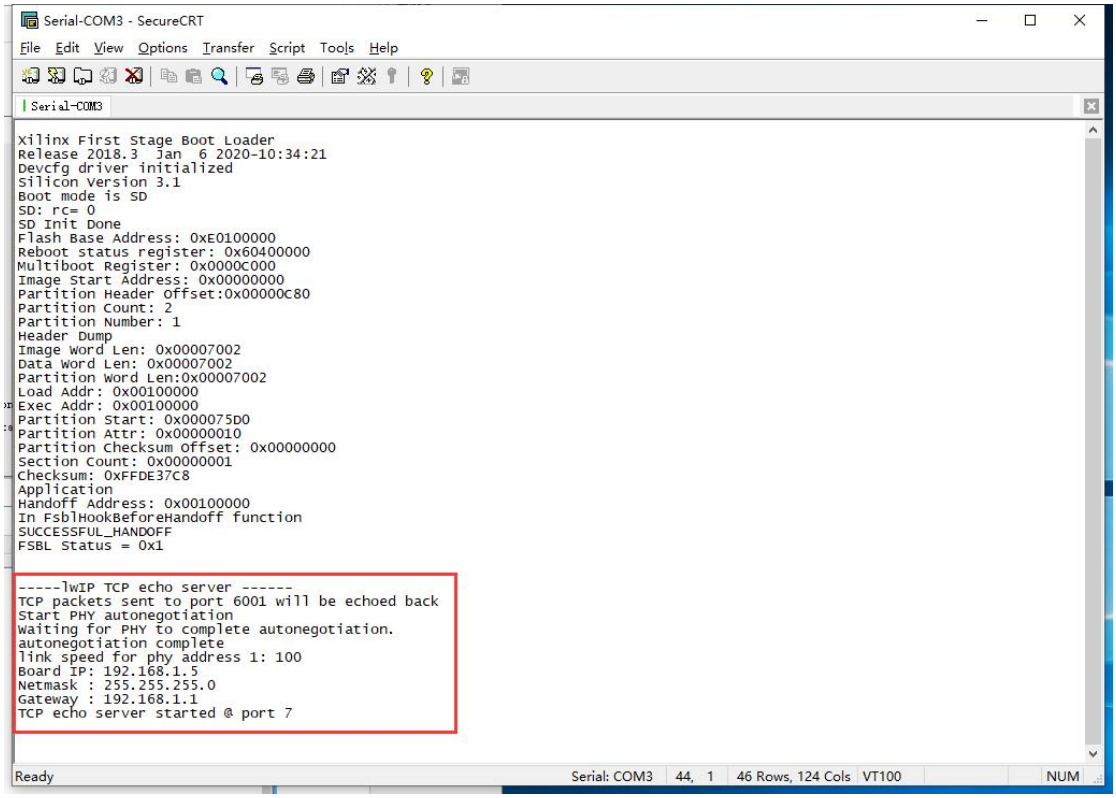
如果烧录时开发板设置为 QSPI 启动模式，会弹出如下图所示警告，可能会导致烧录失败，建议在烧录前将开发板设置为 JTAG 模式。



烧录完成后，关闭开发板电源，修改启动模式为 QSPI，然后上电即可看到与之前相同的启动信息。

最后一种启动方式为 SD 卡启动，将开发板配套的 SD 卡格式化，格式化时文件系统选择 FAT32，格式化完成之后将之前生成的 BOOT.bin 文件拷贝到 SD 卡中，然后关闭电源的情况下将 SD 卡插入开发板上的卡槽。调整跳线帽为 SD 卡启动模式。然后给开发板上电。

然后可以看到开发板通过 SD 卡启动的信息和主函数运行的信息。



10.5 章节小结

通过本章，我们初步了解了 LWIP 的使用方式，同时掌握了其他两种 ZYNQ 的启动方式，希望读者多练习几次，熟练掌握。

第 11 章　　双核 AMP 实验

在 ZYNQ 7000 SOC 芯片中集成了两个 Cortex-A9 的处理器，这种设计解决了在面对复杂设计时，单核 cpu 无法处理多任务、并行控制等任务的问题。这也是 ZYNQ 芯片的优势所在，本实验中我们就来学习双核 CPU 的使用方法。

本章包括以下内容：

11.1 实验简介

11.2 硬件设计

11.3 软件设计

11.4 下载验证

11.5 章节小结

11.1 实验简介

在日常生活中，我们经常听到多核处理器的概念，例如电脑的四核处理器等。多核处理器从多核的结构上区分可分为同构多核架构和异构多核架构两种基本架构。同构多核处理器是指系统中的多个处理器在结构上是相同的；而异构处理器是指系统中的处理器在结构上是不同的。同构多核架构相比于异构多核架构，在硬件和软件设计上较为简单，通用性较高。但在某些特定应用场合下，如异构多核架构具有专用的硬件加速硬核，会在某些应用上获得更好的性能。

我们使用的 ZYNQ SOC 结合了这两种结构，它具备两个相同的独立的 Cortex-A9 处理器，同时还具有可编程的逻辑单元（PL）。同时兼具通用性和高性能。参考官方文档，我们了解到多核处理器的运行模式分为三种，分别为：AMP（非对称多处理）、 SMP（对称多处理）和 BMP（受约束多处理）。

AMP: 非对称多处理，每个 CPU 内核运行一个独立的操作系统或同一操作系统的独立实例

SMP: 对称多处理，一个操作系统实例可以管理所有 CPU 内核，且应用并不绑定某一内核

BMP: 混合多处理，一个操作系统实例可以同时管理所有 CPU 内核，但每个应用被锁定于某个指定的核心。

本次试验采用的是双核 AMP 的运行模式，两个 CPU 分别运行不同的裸机应用程序。

ZYNQ SOC 提供的两个 Cortex-A9 处理器，都具有各自私有的资源，同时也有一些共享的资源。私有资源有 L1 指令缓存、 L1 数据缓存、私有定时器等。共享的资源有 L2 Cache、 DDR 存储器、外设和 OCM（On Chip Memory）等。在 AMP 运行模式下，这两个处理器彼此隔离、分别运行。但在访问共享资源或者外设时，要注意避免冲突，否则程序在运行时可能会出现问题。

在本实验中，我们通过中断触发的方式来控制两个 CPU 的运行，然后通过读写共享内存 OCM 的数据来实现互相通信。在 Xilinx 的官方文档中有两个相关文档描述了双核 AMP 的应用，供用户参考。“XAPP1078”是基于两个 Linux 操作系统的双核 AMP 应用文档，“XAPP1079”是基于两个裸机应用程序的双核 AMP 应用文档。

本次实验的目的是利用双核 AMP，CPU0 接收串口发送的数据，将数据写入共享内存，并产生中断触发 CPU1；CPU1 接收到中断之后，读取共享内存的数据，控制修改呼吸灯的闪烁频率。

11.2 硬件设计

本次实验的硬件设计基于之前的实验 7 自定义 IP 核实验，打开实验 7 的 Vivado 工程另存为一个新的工程并命名为“AMP”。之后删除工程文件夹下的.sdk 文件夹。创建完成后，点击“Open Block Design”打开设计页面，双击 ZYNQ7 Processing System 修改配置，此处使能 QSPI 和 SD0，参考之前的实验。配置完成后如下图所示：

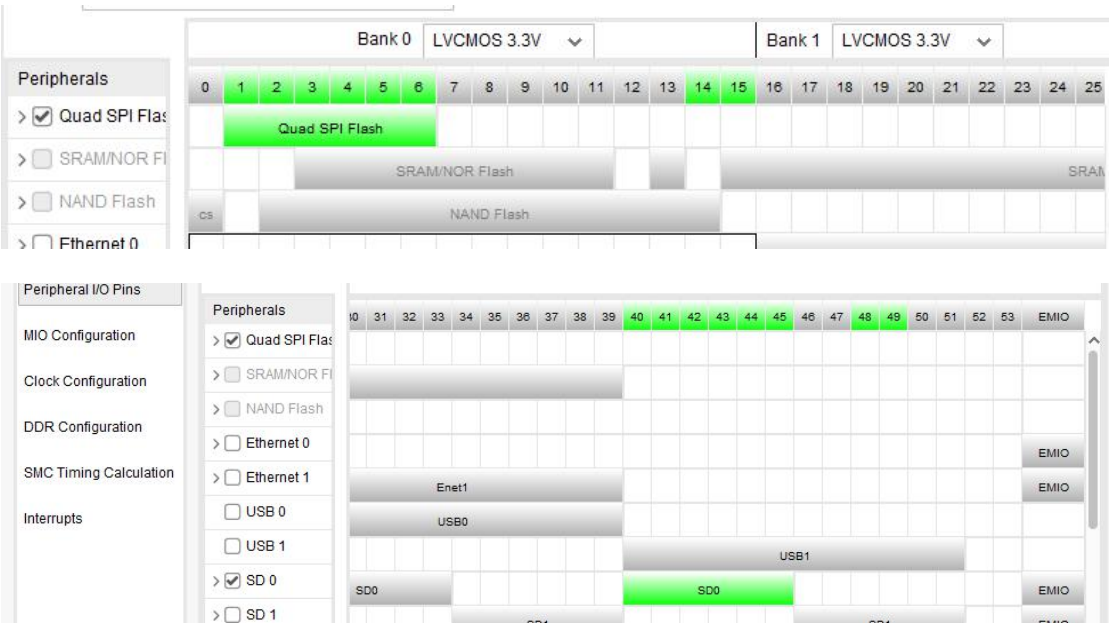


图 11.2.1 配置 QSPI 和 SD0

此处使能 QSPI 和 SD0 是为了后续通过 QSPI 和 SD 卡启动，实现实验不需要这些资源，此处不使能也可。

配置完成后快捷键 Ctrl + S 保存设计，重新生成顶层 HDL 文件。之后点击“Generate Bitstream”重新编译综合生成新的 Bitstream 文件。生成结束之后导出硬件描述文件，并打开 SDK。

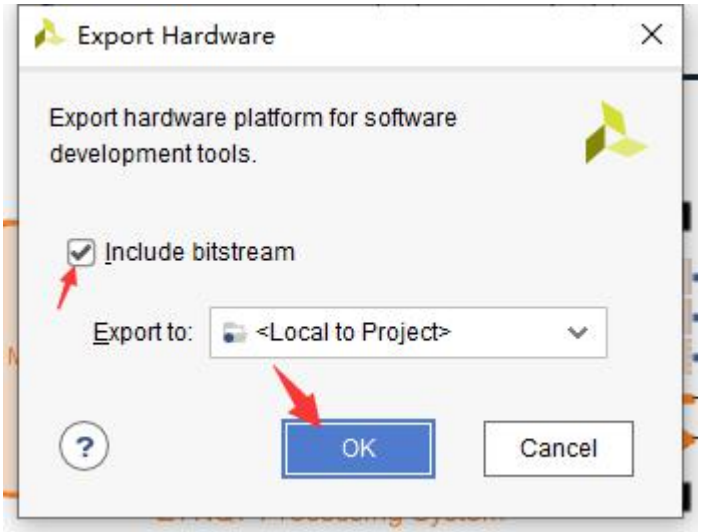


图 11.2.2 导出硬件描述文件

从这里可以看出，在硬件设计上双核 CPU 的使用和单核 CPU 是一样的，实现双核 AMP 的重点在于软件上的设计。

11.3 软件设计

在进行软件设计之前，我们首先来了解本次实验的程序流程。在本次实验中，我们将同时运行两个 cpu 核心，并且两个核心实现不同的功能，同时还要实现两个核心之间的数据交互。在了解实验需求之后，我们就知道了在软件设计时需要完成的工作。

1、首先我们需要创建两个工程分别对应 cpu0 和 cpu1；2、设置两个 cpu 的 DDR3 地址，切记地址不能重合，否则运行会出现异常；3、在 cpu0 的程序中设置接收 cpu1 中断相关设置，设置启动 cpu1，接收串口发来的数据并写入共享内

存，向 cpu1 发送软件中断；4、在 cpu1 的程序中设置接收 cpu0 中断相关设置，读取共享内存中的数据改变呼吸灯的频率，向 cpu0 发送软件中断。

明确设计细节之后我们来完成软件设计。在打开的 SDK 软件中新建工程并命名为“cpu0”，处理器选择“ps7_cortexa9_0”，工程模板选择空白模板。设置如下图所示：

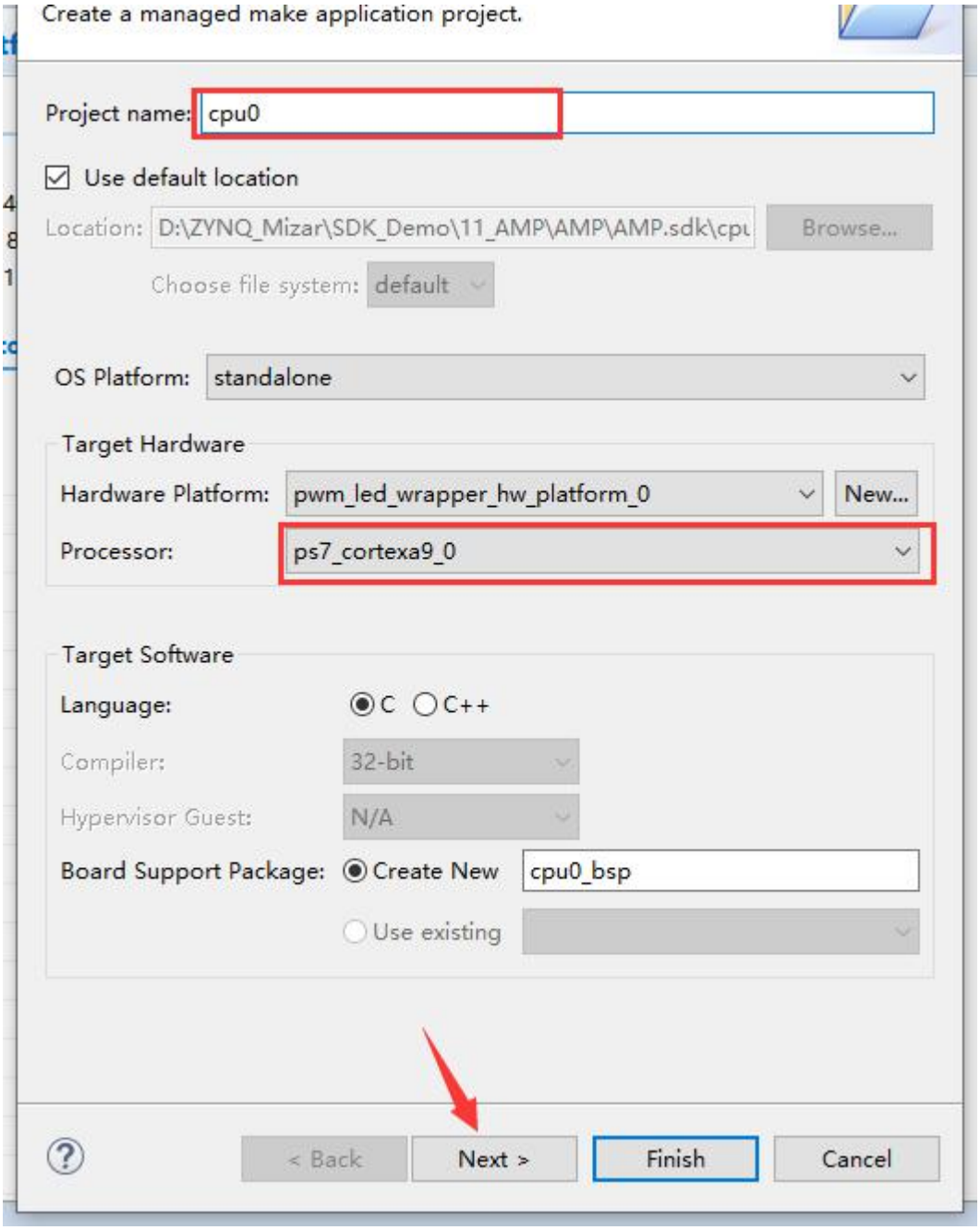


图 11.3.1 新建“cpu0”工程

创建完成后双击 cpu0->src->lscript.ld，如下图所示。在打开的页面中设置 cpu0 的 DDR3 访问空间，基地址为：0x100000；空间大小为：0XFF00000。修改完成后按下快捷键 Ctrl + S 保存设置。

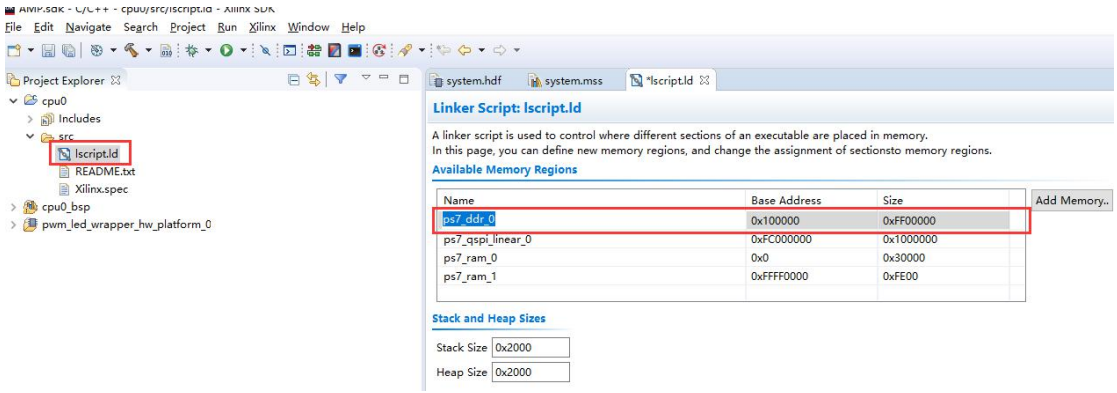
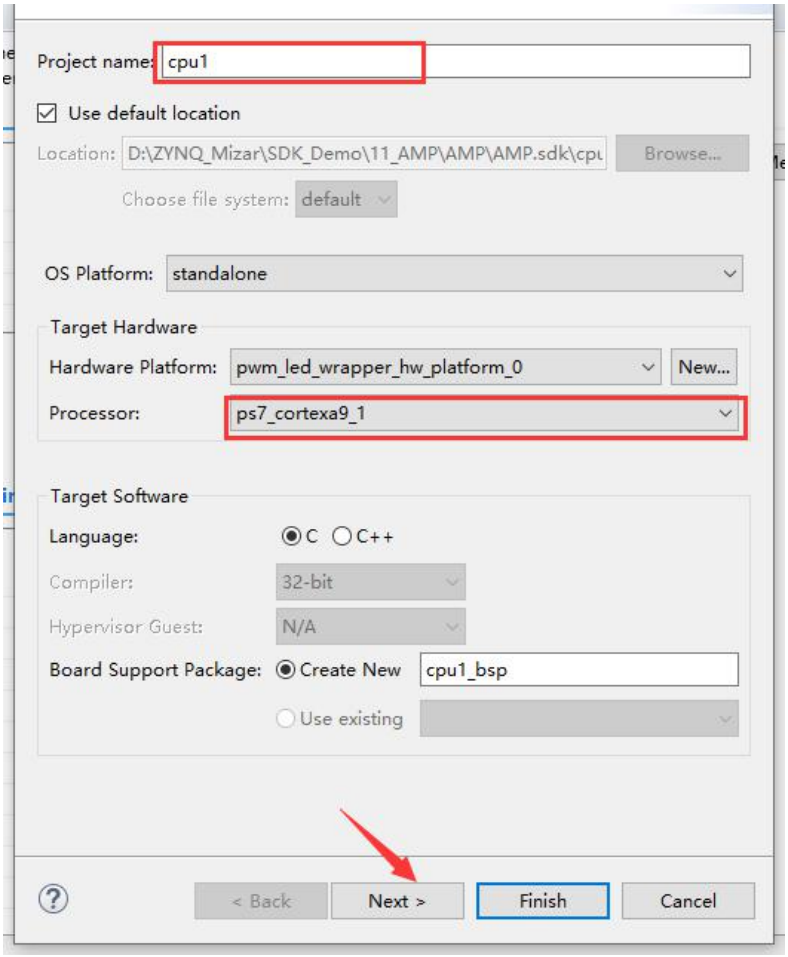


图 11.3.2 设置 cpu0 访问空间

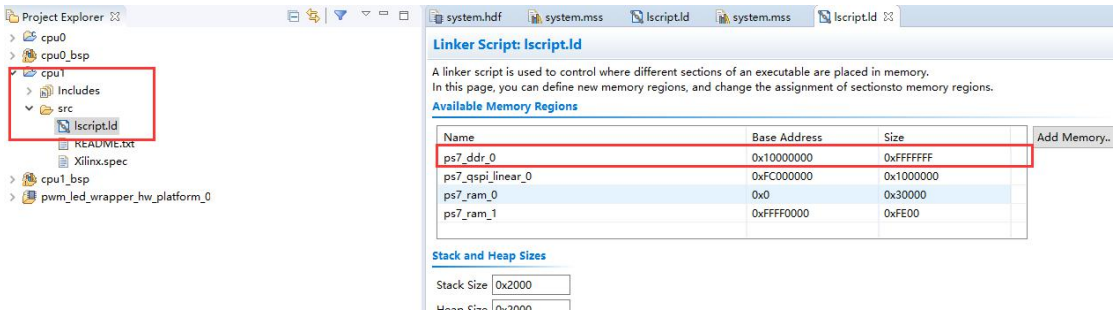
之后再创建一个新的工程并命名为“cpu1”，处理器选择“ps7_cortexa9_1”，



工程模板选择空白模板，如下图所示：

图 11.3.3 新建“cpu1”工程

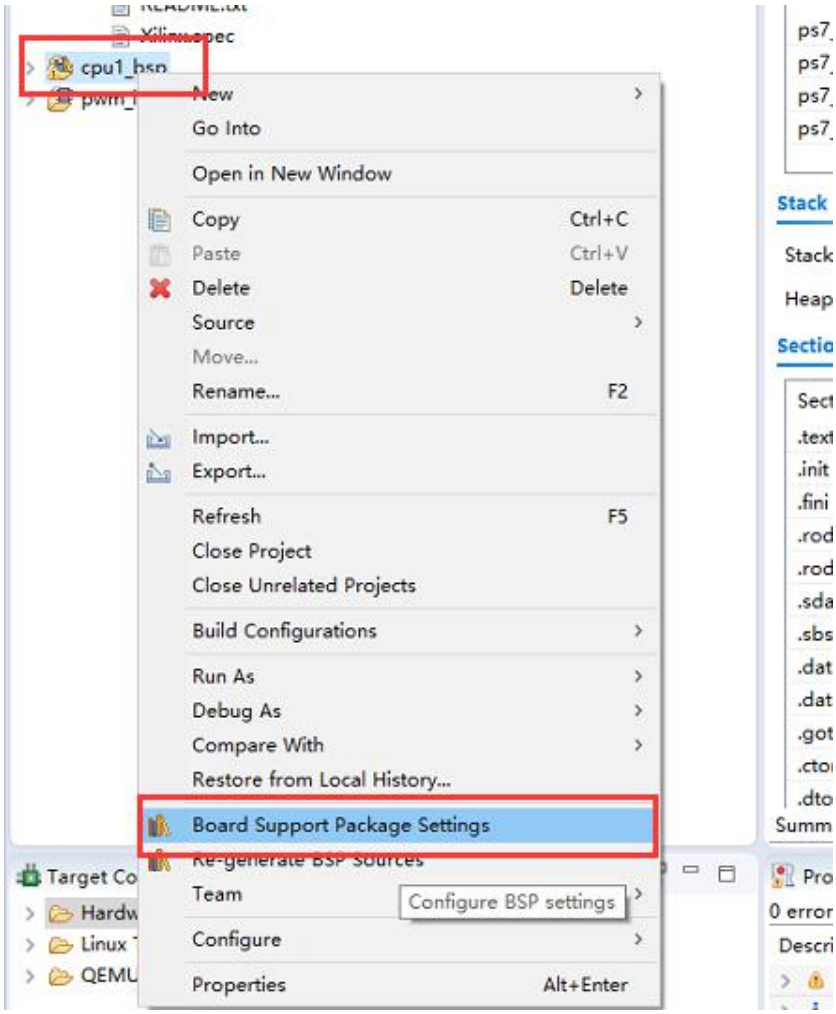
创建完成后双击 cpu1->src->lscript.ld，如下图所示。在打开的页面中设置 cpu0 的 DDR3 访问空间，基地址为：0x10000000；空间大小为：0XFFFFFFF。修



改完成后按下快捷键 Ctrl + S 保存设置。

图 11.3.4 设置 cpu1 访问空间

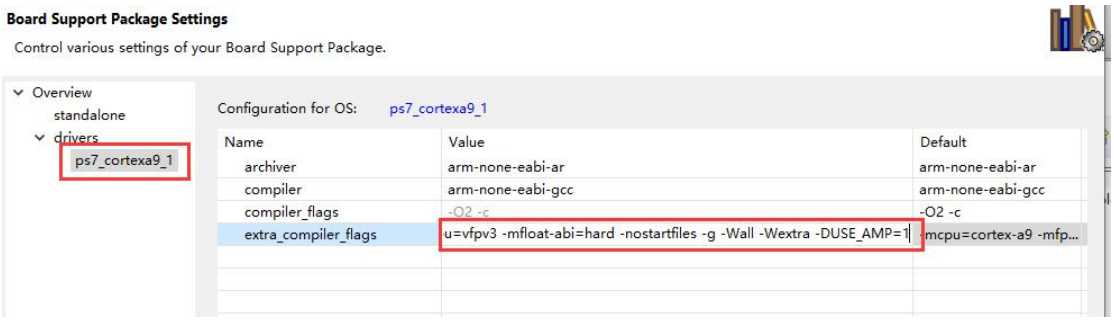
之后设置 cpu1 的板级支持包，使其支持双核同时工作。右键点击 cpu1_bsp，



选择 Board Support Package Settings 点击。

图 11.3.5 设置 cpu1 板级支持包

在弹出的界面中，点击“ps7_cortexa9_1”，在“extra_compiler_flags”一行的 Value 一列下，在末尾输入“-DUSE_AMP=1”，设置完成后点击“OK”按钮，



如下图所示：

图 11.3.6 使能支持双核启动

之后在目录 cpu0/src 上点击右键添加主程序源文件并命名为“cpu0.c”。添加完成后自动打开编辑窗口，在其中输入以下代码：

```c
#include "xparameters.h"
#include "xscugic.h"
#include "xil_printf.h"
#include "xil_exception.h"
#include "xil_mmu.h"
#include "stdio.h"

//宏定义
#define INTC_DEVICE_ID        XPAR_SCUGIC_SINGLE_DEVICE_ID //中断ID
#define SHARE_BASE            0xffff0000                   //共享OCM首地址
#define CPU1_START_ADDR       0xfffffff0                   //CPU1开始地址
#define CPU1_START_MEM        0x10000000                   //CPU1程序开始地址

u16 SoftIntrIdToCpu0 = 0 ;
u16 SoftIntrIdToCpu1 = 1 ;
#define CPU1_ID               2   //CPU1 ID,0bxxxxxx1x指向CPU1
//"SEV"指令唤醒CPU1并跳转至相应的程序
#define sev()                 __asm__("sev")

//函数声明
void start_cpu1();
void cpu0IntrInit(XScuGic *intc_ptr);
void IntrHandler(void *CallbackRef);

//全局变量
XScuGic Intc;                     //中断控制器驱动
int rec_freq_flag = 0;            //接收到来自cpu1中断的标志
int freq;                         //频率调节

//CPU0 main函数
int main()
{
    //S=b1 TEX=b100 AP=b11, Domain=b1111, C=b0, B=b0
    Xil_SetTlbAttributes(SHARE_BASE,0x14de2);    //禁用Cache缓存

    //启动CPU1
    start_cpu1();
    //CPU0中断初始化
    cpu0IntrInit(&Intc);
    while(1){
        if(rec_freq_flag == 0){
            xil_printf("This is CPU0,Please input the numbers 1~3 to change "
                    "breath led frequency\r\n");
            scanf("%d",&freq);
            if(freq >= 1 && freq <=3){
                xil_printf("You input number is %d\r\n",freq);
                //将参数写入共享的内存
                Xil_Out8(SHARE_BASE,freq);
                //给CPU1发送软件中断
                XScuGic_SoftwareIntr(&Intc,SoftIntrIdToCpu1,CPU1_ID);
                rec_freq_flag = 1;
            }
            else{
                xil_printf("Error,The number range is 1~3, please input another number\r\n");
                xil_printf("\r\n");
            }
        }
    }
    return 0 ;
}

//启动CPU1
void start_cpu1()
{
    //向 CPU1_START_ADDR(0Xfffffff0)地址写入 CPU1 的访问内存基地址
    Xil_Out32(CPU1_START_ADDR, CPU1_START_MEM);
    dmb();
    sev();  //唤醒CPU1并跳转至相应的程序
}
```

```
    sev();  //唤醒CPU1并跳转至相应的程序
}

//CPU0中断初始化
void cpu0IntrInit(XScuGic *intc_ptr)
{
    //初始化中断控制器
    XScuGic_Config *intc_cfg_ptr;
    intc_cfg_ptr = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(intc_ptr, intc_cfg_ptr,
            intc_cfg_ptr->CpuBaseAddress);

    //设置并打开中断异常处理功能
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler)XScuGic_InterruptHandler, intc_ptr);
    Xil_ExceptionEnable();
    //关联中断源及中断处理函数
    XScuGic_Connect(intc_ptr, SoftIntrIdToCpu0,
          (Xil_ExceptionHandler)IntrHandler, (void *)intc_ptr);

    XScuGic_Enable(intc_ptr, SoftIntrIdToCpu0); //CPU0软件中断
}

//中断服务函数
void IntrHandler(void *CallbackRef)
{
    xil_printf("This is CPU0,Receive software interrupt from CPU1\r\n");
    xil_printf("\r\n");
    rec_freq_flag = 0;
}
```

输入完成后快捷键 Ctrl + S 保存。

之后在目录 cpu1/src 上点击右键添加主程序源文件并命名为“cpu1.c”。添

```
#include "xparameters.h"
#include "xscugic.h"
#include "xil_printf.h"
#include "xil_exception.h"
#include "xil_mmu.h"
#include "stdio.h"
#include "pwm_ip.h"

//宏定义
#define INTC_DEVICE_ID       XPAR_SCUGIC_SINGLE_DEVICE_ID //中断ID
#define SHARE_BASE           0xffff0000                   //共享OCM首地址
u16 SoftIntrIdToCpu0 = 0 ;
u16 SoftIntrIdToCpu1 = 1 ;
#define CPU0_ID              1                            //CPU1 ID,0bxxxxxxx1指向CPU0
    //CPU1中断初始化
    cpu1IntrInit(&Intc);
    while(1){
        if(intr_flag){
            freq = Xil_In8(SHARE_BASE);     //从共享内存中读出数据
            xil_printf("CUP1 Received data is %d\r\n",freq) ;
            switch(freq){
                case 1 : freq_step = 50;break;
                case 2 : freq_step = 200;break;
                case 3 : freq_step = 400;break;
                default : freq_step = 50;break;
            }
            //设置呼吸灯频率
            PWM_IP_mWriteReg(PWM_IP_BASEADDR,PWM_IP_REG1,(0x00000000|freq_step));
            //给CPU0发送软件中断
            XScuGic_SoftwareIntr(&Intc,SoftIntrIdToCpu0,CPU0_ID);
            intr_flag = 0;
        }
    }
    return 0 ;
}
```

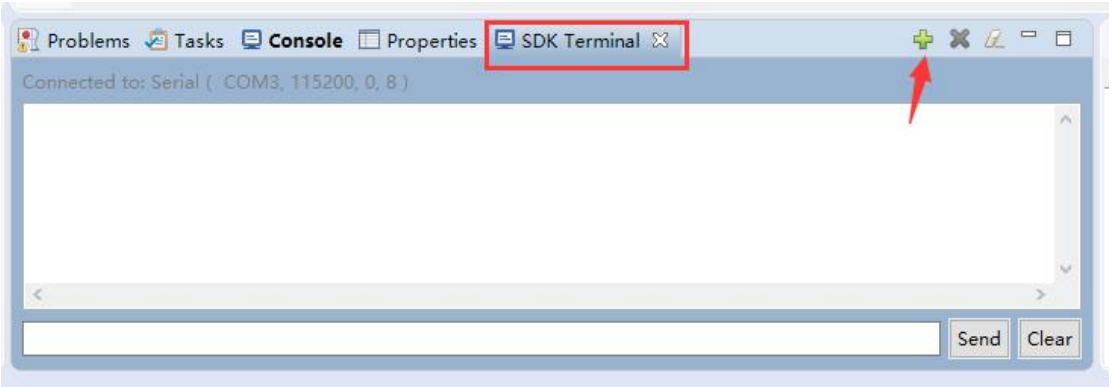
加完成后自动打开编辑窗口，在其中输入以下代码：

```
//CPU1中断初始化
void cpu1IntrInit(XScuGic *intc_ptr)
{
    //初始化中断控制器
    XScuGic_Config *intc_cfg_ptr;
    intc_cfg_ptr = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(intc_ptr, intc_cfg_ptr,
            intc_cfg_ptr->CpuBaseAddress);
    //设置并打开中断异常处理功能
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler)XScuGic_InterruptHandler, intc_ptr);
    Xil_ExceptionEnable();
    //关联中断源及中断处理函数
    XScuGic_Connect(intc_ptr, SoftIntrIdToCpu1,
          (Xil_ExceptionHandler)IntrHandler, (void *)intc_ptr);

    XScuGic_Enable(intc_ptr, SoftIntrIdToCpu1); //使能CPU1软件中断
}

//中断服务函数
void IntrHandler(void *CallbackRef)
{
    xil_printf("This is CUP1,Soft Interrupt from CPU0\r\n") ;
    intr_flag = 1;
}
```

输入完成后快捷键 Ctrl + S 保存。工具会自动进行编译，可以在下方控制台（Console）看到编译过程。

11.4 下载验证

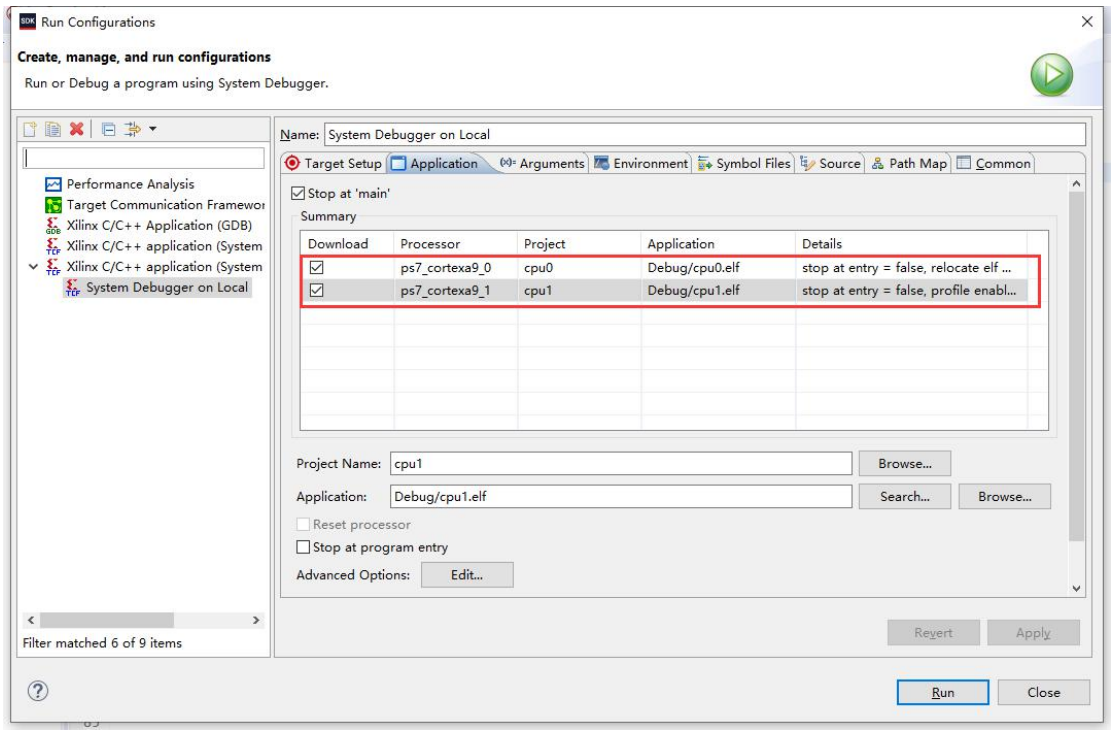
首先按照之前的实验连接好烧录线和串口线。之后打开下方 SDK Terminal



窗口，点击加号连接串口。

图 11.4.1 连接串口

点击菜单栏 Run->Run Configurations…，在弹出的界面中双击 Xilinx C/C++ application（System Debug），即对下载的选项进行设置。在打开的页面按照如下图所示数字的顺序修改相关设置，在 Application 下选择两个处理器 ps7_cortexa9_0 和 ps7_cortexa9_1。


Run Configurations
Create, manage, and run configurations
Run or Debug a program using System Debugger.
Performance Analysis
Target Communication Framewor
Xilinx C/C++ Application (GDB)
Xilinx C/C++ application (System
Xilinx C/C++ application (System
System Debugger on Local
Filter matched 6 of 9 items
Name: System Debugger on Local
Target Setup
Application
Arguments
Environment
Symbol Files
Source
Path Map
Common
Stop at 'main'
Summary
Download
Processor
Project
Application
Details
ps7_cortexa9_0
cpu0
Debug/cpu0.elf
stop at entry = false, relocate elf ...
ps7_cortexa9_1
cpu1
Debug/cpu1.elf
stop at entry = false, profile enabl...
Project Name: cpu1
Browse...
Application: Debug/cpu1.elf
Search...
Browse...
Reset processor
Stop at program entry
Advanced Options:
Edit...
Revert
Apply
Run
Close

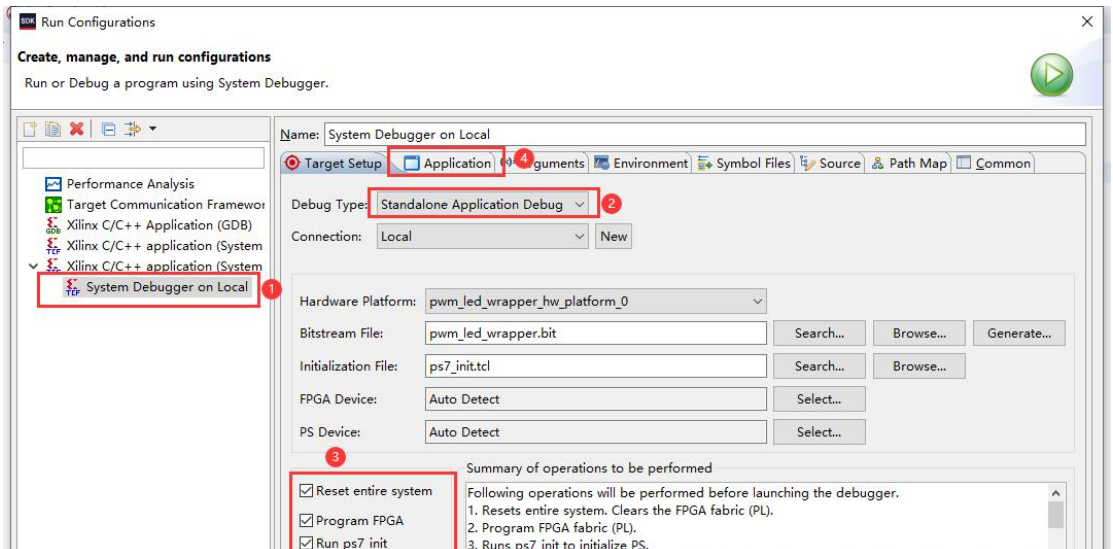
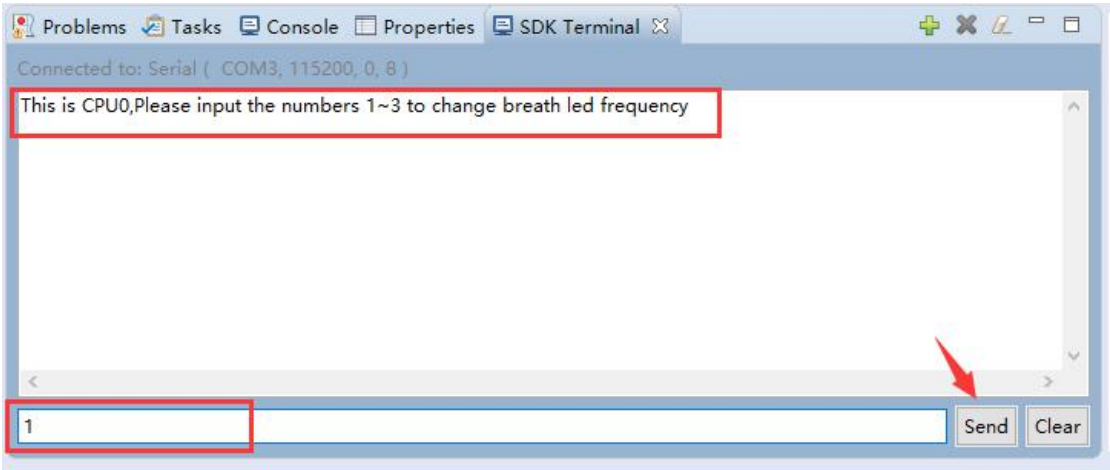

Run Configurations
Create, manage, and run configurations
Run or Debug a program using System Debugger.
Performance Analysis
Target Communication Framewor
Xilinx C/C++ Application (GDB)
Xilinx C/C++ application (System
Xilinx C/C++ application (System
System Debugger on Local
1
Name: System Debugger on Local
Target Setup
Application
4
Environment
Symbol Files
Source
Path Map
Common
Debug Type: Standalone Application Debug
2
Connection: Local
New
Hardware Platform: pwm_led_wrapper_hw_platform_0
Bitstream File: pwm_led_wrapper.bit
Search...
Browse...
Generate...
Initialization File: ps7_init.tcl
Search...
Browse...
FPGA Device: Auto Detect
Select...
PS Device: Auto Detect
Select...
3
Reset entire system
Program FPGA
Run ps7_init
Summary of operations to be performed
Following operations will be performed before launching the debugger.
1. Resets entire system. Clears the FPGA fabric (PL).
2. Program FPGA fabric (PL).
3. Runs ps7_init to initialize PS.

图 11.4.2 设置烧录程序相关设置

设置完成后点击“Run”烧录程序，烧录完成后可以看到下方 SDK Terminal



窗口打印的信息，如下图所示：

图 11.4.3 烧录完成

在编辑栏输入“1”并点击“Send”，可以看到 pl_led1 出现呼吸效果，同时打印出相关信息。发送 2 或者 3 可以看到呼吸频率的变化。说明实验成功。

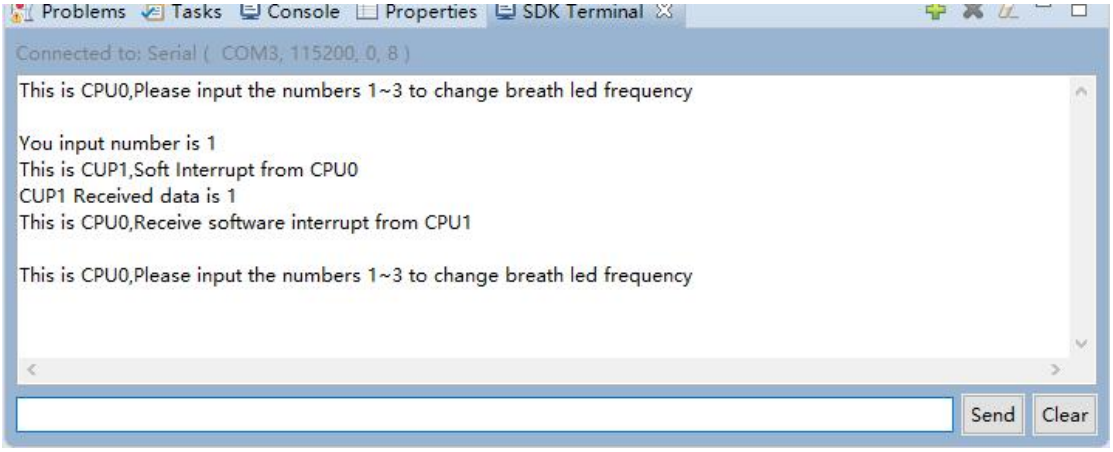


图 11.4.4 改变呼吸频率

之后介绍一下双核 AMP 实验程序固化到 Flash 或 SD 卡的方法。在本次实验中有两个板级支持包，在新建 FSBL 的时候可以选择创建一个新的 fsbl_bsp 或者选择已有的 cpu0_bsp，（不可以选择 cpu1_bsp）。此处我们创建一个新的 FSBL 工程，如下图所示：

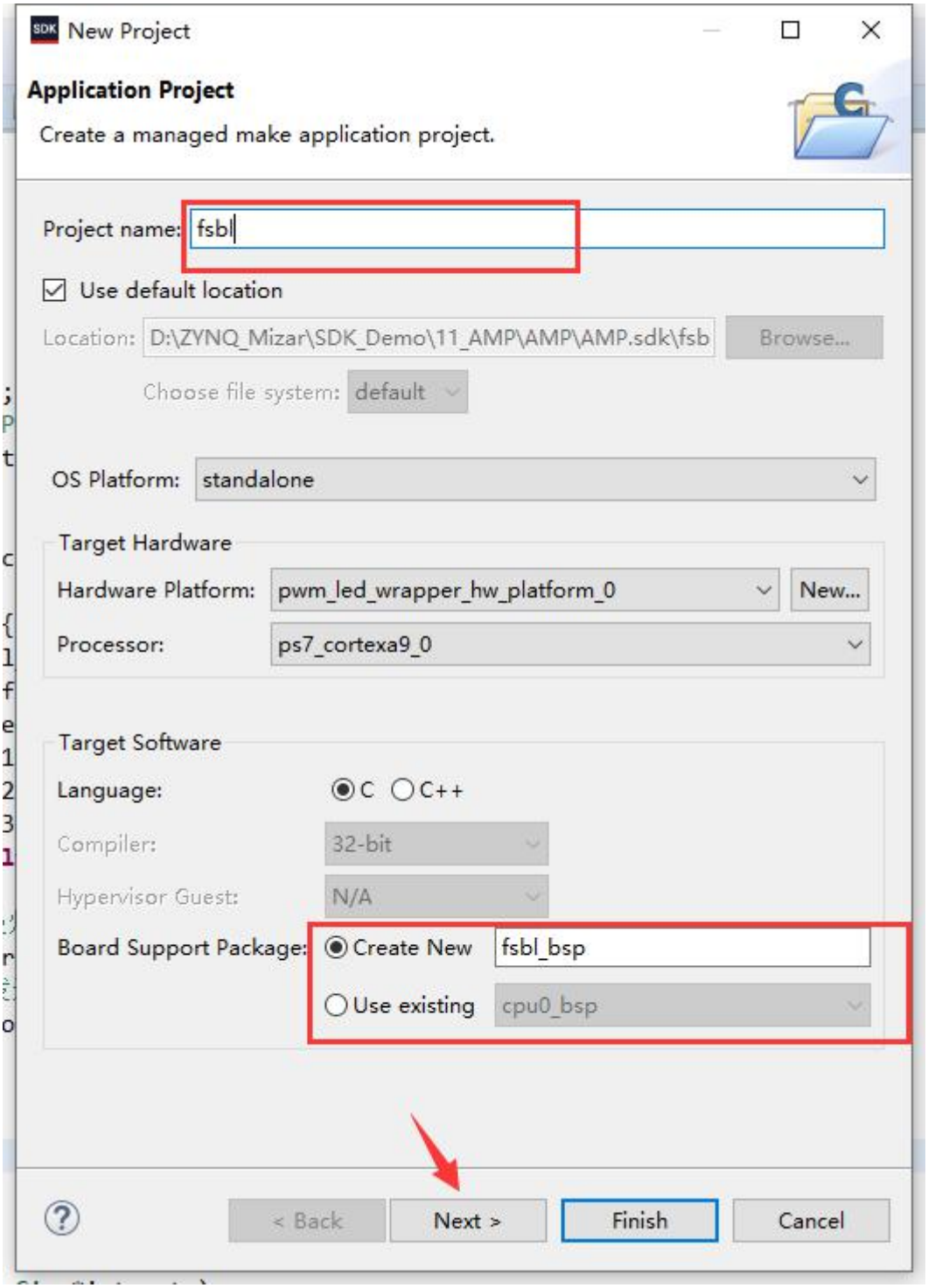


图 11.4.5 创建 FSBL 工程

创建完成之后按照之前的操作步骤生成 BOOT.bin 文件，在添加文件时加入 cpu1.elf，如下图所示，切记添加顺序不能出错，否则无法运行。

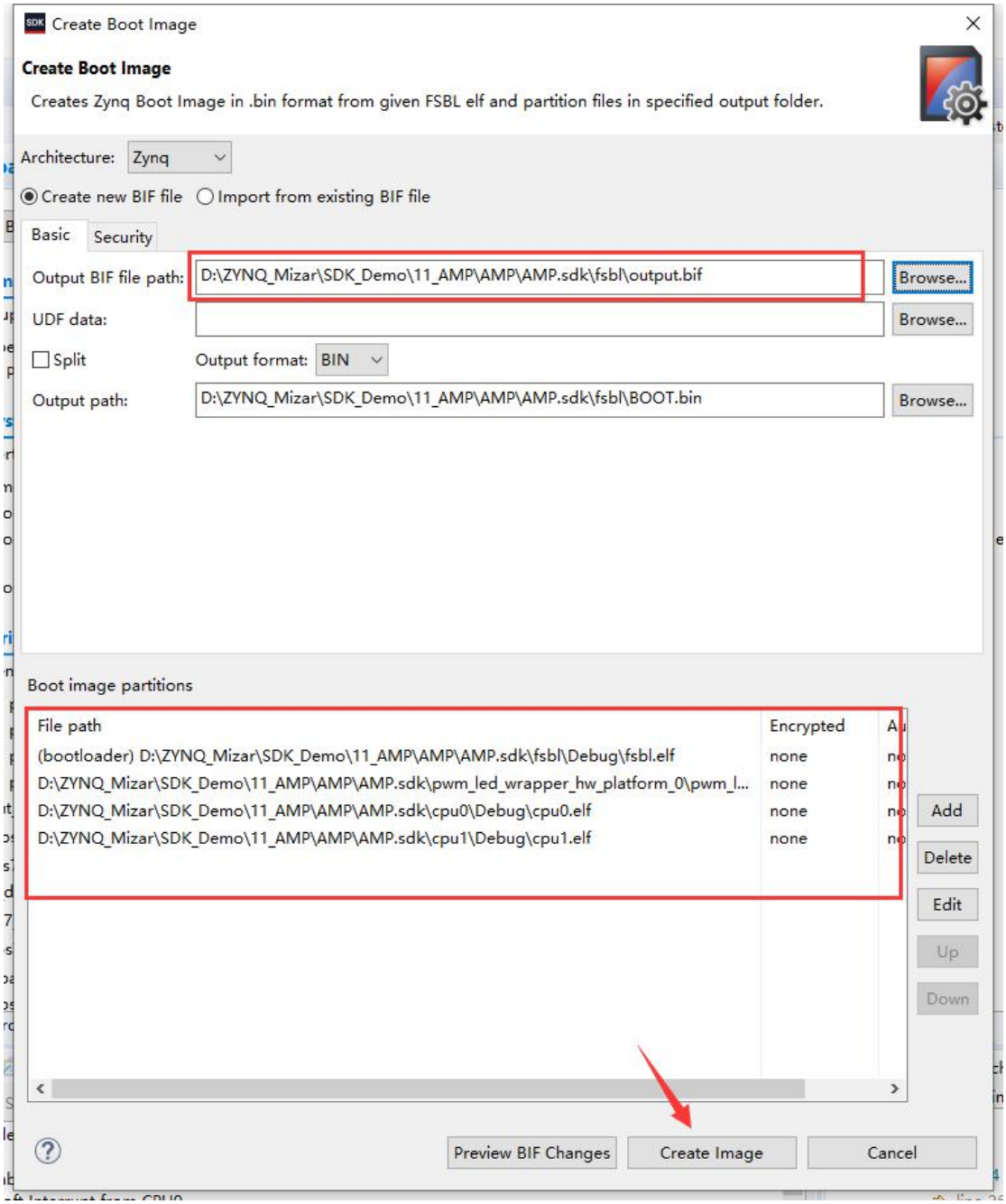


图 11.4.6 生成 BOOT.bin 文件

生成完成之后即可烧录到 Flash 或拷贝到 SD 卡中启动，操作步骤参照之前的实验。

11.5 本章小结

在本次实验中我们学习了怎样实现双核 AMP 的运行，重点在于双核之间的数据交互。请读者在本实验的基础上查阅官方文档，加深对于双核使用的理解。

第 12 章　　FLASH 数据读写实验

在之前的实验中，我们使用外部 Flash 作为程序存储的位置来实现程序掉电不删除，除此之外，Flash 还通常用来进行数据存储。在本次实验中，我们来学习如何对 Flash 进行数据读写。

本章包括以下内容：

12.1 实验简介

12.2 硬件设计

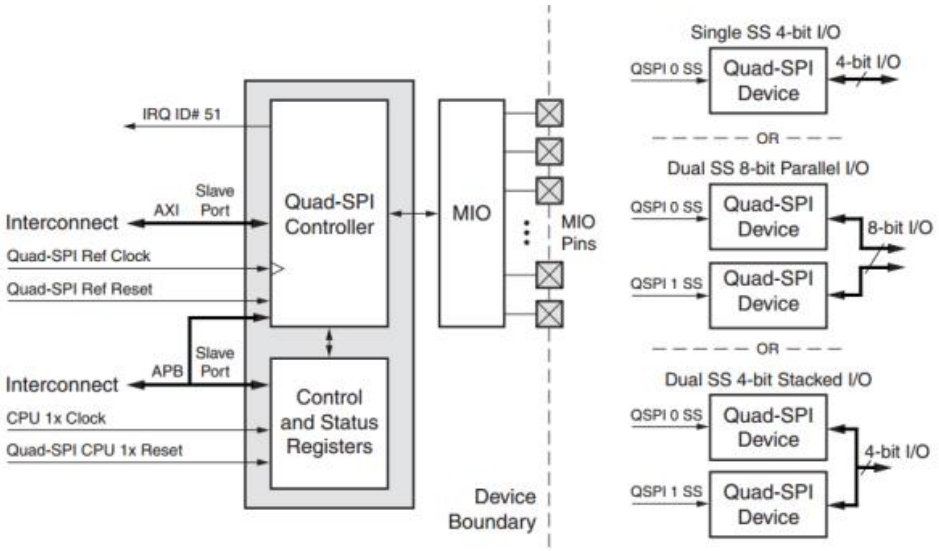
12.3 软件设计

12.4 下载验证

12.5 章节小结

12.1 实验简介

在 ZYNQ SOC 中集成了一个 QSPI FLASH 控制器来实现对于 Flash 的读写控制，



QSPI FLASH 控制器的系统框图如下图所示：

图 12.1.1 QSPI FLASH 控制器系统框图

从图中我们可以看出 QSPI FLASH 控制器通过 MIO 和外部 Flash 相连接，支持三种模式：单个从器件模式、双从器件并行模式和双从器件堆模式。通过使用双从器件模式可以扩展 QSPI Flash 的存储容量。另外还可以看到控制器还有 AXI 和 APB 两种类型的接口。

12.2 硬件设计

本实验重点在于对于 Flash 的操作，只需使能 QSPI FLASH 控制器即可，无需其他的外设，所以本实验硬件设计基于实验 1 的工程，添加 QSPI FLASH 即可。打开实验 1 的 Vivado 工程，另存为一个新的工程并命名为“qspi_flash_test”。创建完成后删除工程目录下的.sdk 文件夹。之后点击“Open Block Design”打开设计页面，双击 ZYNQ7 Processing System 打开配置页面，在 Peripheral I/O Pins 下使

能 QSPI FLASH，如下图所示：

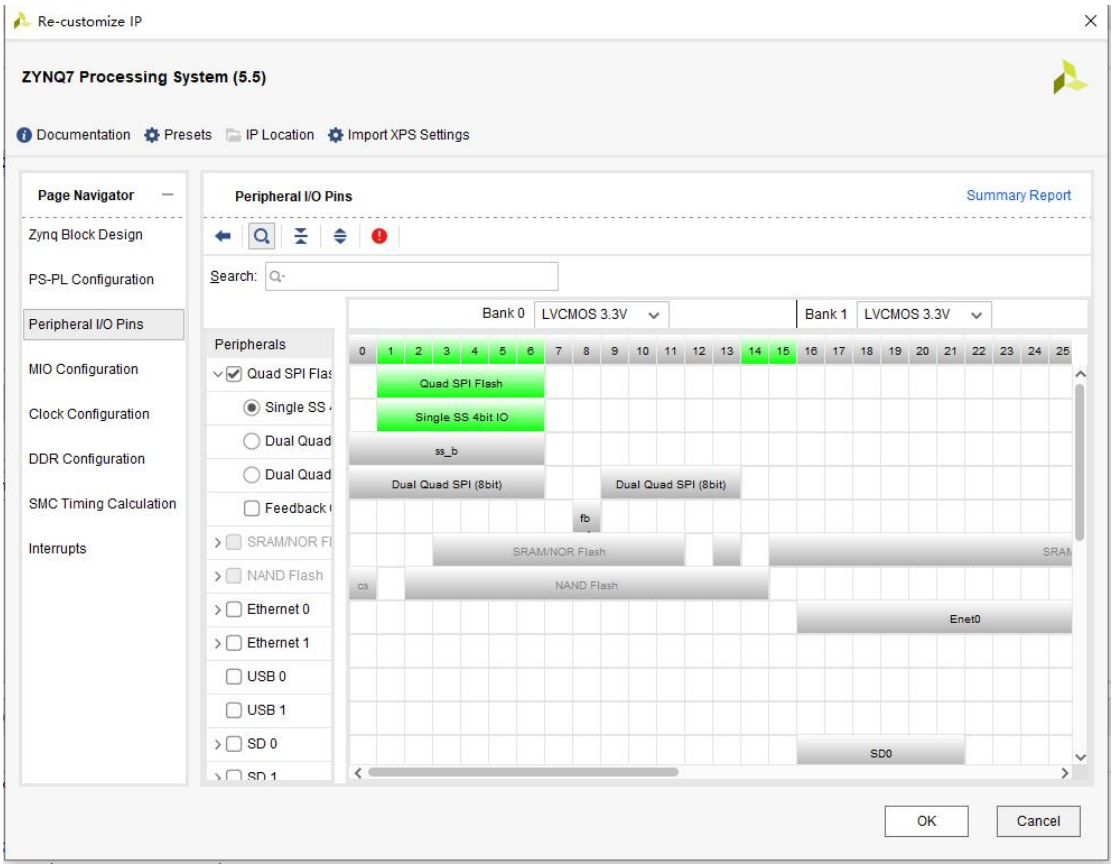


图 12.2.1 使能 QSPI FLASH 控制器

配置完成后快捷键 Ctrl + S 保存，之后在 Source 窗口中右键点击 Block Design 设计文件“.bd”， 然后依次执行“Generate OutputProducts”和“Create HDL Wrapper”重新生成顶层 HDL。

然后在菜单栏中选择 File > Export > Export hardware 导出硬件， 最后在菜单栏选择 File > Launch SDK，启动 SDK 软件。

12.3 软件设计

在 SDK 软件中新建一个工程并命名为“qspi_flash_test”，工程模板选择空白模板。之后为应用工程新建一个主函数源文件“main.c”。创建完成后在 mainc

system.hdf system.mss main.c xqspips_flash_polled_example.c

```c
#include "xparameters.h"    /* SDK generated parameters */
#include "xqspips.h"        /* QSPI device driver */
#include "xil_printf.h"

#define QSPI_DEVICE_ID      XPAR_XQSPIPS_0_DEVICE_ID

//向Flash发送的指令
#define WRITE_STATUS_CMD    0x01
#define WRITE_CMD       0x02
#define READ_CMD        0x03
#define WRITE_DISABLE_CMD   0x04
#define READ_STATUS_CMD     0x05
#define WRITE_ENABLE_CMD    0x06
#define FAST_READ_CMD       0x0B
#define DUAL_READ_CMD       0x3B
#define QUAD_READ_CMD       0x6B
#define BULK_ERASE_CMD      0xC7
#define SEC_ERASE_CMD       0xD8
#define READ_ID         0x9F


#define COMMAND_OFFSET      0 /* FLASH instruction */
#define ADDRESS_1_OFFSET    1 /* MSB byte of address to read or write */
#define ADDRESS_2_OFFSET    2 /* Middle byte of address to read or write */
#define ADDRESS_3_OFFSET    3 /* LSB byte of address to read or write */
#define DATA_OFFSET         4 /* Start of Data for Read/Write */
#define DUMMY_OFFSET        4 /* Dummy byte offset for fast, dual and quad reads*/
#define DUMMY_SIZE          1 /* Number of dummy bytes for fast, dual and quad reads  */
#define RD_ID_SIZE          4 /* Read ID command + 3 bytes ID response */
#define BULK_ERASE_SIZE     1 /* Bulk Erase command size */
#define SEC_ERASE_SIZE      4 /* Sector Erase command + Sector address */

#define OVERHEAD_SIZE       4

#define SECTOR_SIZE     0x10000
#define NUM_SECTORS     0x100
#define NUM_PAGES       0x10000
#define PAGE_SIZE       256

#define PAGE_COUNT      16

#define TEST_ADDRESS        0x00055000
#define UNIQUE_VALUE        0x05
#define MAX_DATA        (PAGE_COUNT * PAGE_SIZE)

void FlashErase(XQspiPs *QspiPtr, u32 Address, u32 ByteCount);
void FlashWrite(XQspiPs *QspiPtr, u32 Address, u32 ByteCount, u8 Command);
void FlashRead(XQspiPs *QspiPtr, u32 Address, u32 ByteCount, u8 Command);
int FlashReadID(void);
void FlashQuadEnable(XQspiPs *QspiPtr);
int QspiFlashPolledExample(XQspiPs *QspiInstancePtr, u16 QspiDeviceId);


static XQspiPs QspiInstance;
int Test = 5;

u8 ReadBuffer[MAX_DATA + DATA_OFFSET + DUMMY_SIZE];
u8 WriteBuffer[PAGE_SIZE + DATA_OFFSET];

int main(void)
{
    int Status;

    xil_printf("QSPI FLASH Polled Example Test \r\n");
```

中输入以下代码：

```
65
66      /* Run the Qspi Interrupt example.*/
67      Status = QspiFlashPolledExample(&QspiInstance, QSPI_DEVICE_ID);
68      if (Status != XST_SUCCESS) {
69          xil_printf("QSPI FLASH Polled Example Test Failed\r\n");
70          return XST_FAILURE;
71      }
72
73      xil_printf("Successfully ran QSPI FLASH Polled Example Test\r\n");
74      return XST_SUCCESS;
75  }
76
77  int QspiFlashPolledExample(XQspiPs *QspiInstancePtr, u16 QspiDeviceId)
78  {
79      int Status;
80      u8 *BufferPtr;
81      u8 UniqueValue;
82      int Count;
83      int Page;
84      XQspiPs_Config *QspiConfig;
85
86      /* Initialize the QSPI driver so that it's ready to use*/
87      QspiConfig = XQspiPs_LookupConfig(QspiDeviceId);
88      if (QspiConfig == NULL) {
89          return XST_FAILURE;
90      }
91
92      Status = XQspiPs_CfgInitialize(QspiInstancePtr, QspiConfig,
93                      QspiConfig->BaseAddress);
94      if (Status != XST_SUCCESS) {
95          return XST_FAILURE;
96      }
```

```
98      /* Perform a self-test to check hardware build*/
99      Status = XQspiPs_SelfTest(QspiInstancePtr);
100     if (Status != XST_SUCCESS) {
101         return XST_FAILURE;
102     }
103
104     for (UniqueValue = UNIQUE_VALUE, Count = 0; Count < PAGE_SIZE;
105          Count++, UniqueValue++) {
106         WriteBuffer[DATA_OFFSET + Count] = (u8)(UniqueValue + Test);
107     }
108     memset(ReadBuffer, 0x00, sizeof(ReadBuffer));
109
110     //设置手动启动和手动片选模式
111     XQspiPs_SetOptions(QspiInstancePtr, XQSPIPS_MANUAL_START_OPTION |
112             XQSPIPS_FORCE_SSELECT_OPTION |
113             XQSPIPS_HOLD_B_DRIVE_OPTION);
114
115     // 设置时钟分频系数
116     XQspiPs_SetClkPrescaler(QspiInstancePtr, XQSPIPS_CLK_PRESCALE_8);
117
118     /* Assert the FLASH chip select.*/
119     XQspiPs_SetSlaveSelect(QspiInstancePtr);
120
121
122     FlashReadID();
123
124     FlashQuadEnable(QspiInstancePtr);
125
126     /* 擦除 flash.*/
127     FlashErase(QspiInstancePtr, TEST_ADDRESS, MAX_DATA);
```

```
128
129     //向Flash中写入数据
130     for (Page = 0; Page < PAGE_COUNT; Page++) {
131         FlashWrite(QspiInstancePtr, (Page * PAGE_SIZE) + TEST_ADDRESS,
132                PAGE_SIZE, WRITE_CMD);
133     }
134
135     //使用普通读取命令读出数据
136     FlashRead(QspiInstancePtr, TEST_ADDRESS, MAX_DATA, READ_CMD);
137
138     BufferPtr = &ReadBuffer[DATA_OFFSET];
139
140     for (UniqueValue = UNIQUE_VALUE, Count = 0; Count < MAX_DATA;
141          Count++, UniqueValue++) {
142         if (BufferPtr[Count] != (u8)(UniqueValue + Test)) {
143             return XST_FAILURE;
144         }
145     }
146     //使用快速读取命令读出数据
147     memset(ReadBuffer, 0x00, sizeof(ReadBuffer));
148     FlashRead(QspiInstancePtr, TEST_ADDRESS, MAX_DATA, FAST_READ_CMD);
149
150     BufferPtr = &ReadBuffer[DATA_OFFSET + DUMMY_SIZE];
151
152     for (UniqueValue = UNIQUE_VALUE, Count = 0; Count < MAX_DATA;
153          Count++, UniqueValue++) {
154         if (BufferPtr[Count] != (u8)(UniqueValue + Test)) {
155             return XST_FAILURE;
156         }
157     }
```

```
159     //使用双重读取命令读取数据
160     memset(ReadBuffer, 0x00, sizeof(ReadBuffer));
161     FlashRead(QspiInstancePtr, TEST_ADDRESS, MAX_DATA, DUAL_READ_CMD);
162     BufferPtr = &ReadBuffer[DATA_OFFSET + DUMMY_SIZE];
163
164     for (UniqueValue = UNIQUE_VALUE, Count = 0; Count < MAX_DATA;
165          Count++, UniqueValue++) {
166         if (BufferPtr[Count] != (u8)(UniqueValue + Test)) {
167             return XST_FAILURE;
168         }
169     }
170     //使用块读取命令读取数据
171     memset(ReadBuffer, 0x00, sizeof(ReadBuffer));
172     FlashRead(QspiInstancePtr, TEST_ADDRESS, MAX_DATA, QUAD_READ_CMD);
173
174     BufferPtr = &ReadBuffer[DATA_OFFSET + DUMMY_SIZE];
175
176     for (UniqueValue = UNIQUE_VALUE, Count = 0; Count < MAX_DATA;
177          Count++, UniqueValue++) {
178         if (BufferPtr[Count] != (u8)(UniqueValue + Test)) {
179             return XST_FAILURE;
180         }
181     }
182
183     return XST_SUCCESS;
184 }
185
186 void FlashWrite(XQspiPs *QspiPtr, u32 Address, u32 ByteCount, u8 Command)
187 {
188     u8 WriteEnableCmd = { WRITE_ENABLE_CMD };
189     u8 ReadStatusCmd[] = { READ_STATUS_CMD, 0 };  /* must send 2 bytes */
190     u8 FlashStatus[2];
191
192
```

```
    XQspiPs_PolledTransfer(QspiPtr, &WriteEnableCmd, NULL,
                sizeof(WriteEnableCmd));

    WriteBuffer[COMMAND_OFFSET]   = Command;
    WriteBuffer[ADDRESS_1_OFFSET] = (u8)((Address & 0xFF0000) >> 16);
    WriteBuffer[ADDRESS_2_OFFSET] = (u8)((Address & 0xFF00) >> 8);
    WriteBuffer[ADDRESS_3_OFFSET] = (u8)(Address & 0xFF);

    XQspiPs_PolledTransfer(QspiPtr, WriteBuffer, NULL,
                ByteCount + OVERHEAD_SIZE);

    while (1) {
        XQspiPs_PolledTransfer(QspiPtr, ReadStatusCmd, FlashStatus,
                    sizeof(ReadStatusCmd));
        if ((FlashStatus[1] & 0x01) == 0) {
            break;
        }
    }
}

void FlashRead(XQspiPs *QspiPtr, u32 Address, u32 ByteCount, u8 Command)
{
    WriteBuffer[COMMAND_OFFSET]   = Command;
    WriteBuffer[ADDRESS_1_OFFSET] = (u8)((Address & 0xFF0000) >> 16);
    WriteBuffer[ADDRESS_2_OFFSET] = (u8)((Address & 0xFF00) >> 8);
    WriteBuffer[ADDRESS_3_OFFSET] = (u8)(Address & 0xFF);

    if ((Command == FAST_READ_CMD) || (Command == DUAL_READ_CMD) ||
        (Command == QUAD_READ_CMD)) {
        ByteCount += DUMMY_SIZE;
    }
    XQspiPs_PolledTransfer(QspiPtr, WriteBuffer, ReadBuffer,
                ByteCount + OVERHEAD_SIZE);
```

```
                ByteCount + OVERHEAD_SIZE);
}

void FlashErase(XQspiPs *QspiPtr, u32 Address, u32 ByteCount)
{
    u8 WriteEnableCmd = { WRITE_ENABLE_CMD };
    u8 ReadStatusCmd[] = { READ_STATUS_CMD, 0 };  /* must send 2 bytes */
    u8 FlashStatus[2];
    int Sector;

    if (ByteCount == (NUM_SECTORS * SECTOR_SIZE)) {
        XQspiPs_PolledTransfer(QspiPtr, &WriteEnableCmd, NULL,
                    sizeof(WriteEnableCmd));
        WriteBuffer[COMMAND_OFFSET]   = BULK_ERASE_CMD;
        XQspiPs_PolledTransfer(QspiPtr, WriteBuffer, NULL,
                    BULK_ERASE_SIZE);

        while (1) {
            XQspiPs_PolledTransfer(QspiPtr, ReadStatusCmd,
                        FlashStatus,
                        sizeof(ReadStatusCmd));
            if ((FlashStatus[1] & 0x01) == 0) {
                break;
            }
        }
        return;
    }

    for (Sector = 0; Sector < ((ByteCount / SECTOR_SIZE) + 1); Sector++) {
        XQspiPs_PolledTransfer(QspiPtr, &WriteEnableCmd, NULL,
                    sizeof(WriteEnableCmd));
```

```
256
257         WriteBuffer[COMMAND_OFFSET]   = SEC_ERASE_CMD;
258         WriteBuffer[ADDRESS_1_OFFSET] = (u8)(Address >> 16);
259         WriteBuffer[ADDRESS_2_OFFSET] = (u8)(Address >> 8);
260         WriteBuffer[ADDRESS_3_OFFSET] = (u8)(Address & 0xFF);
261
262         XQspiPs_PolledTransfer(QspiPtr, WriteBuffer, NULL,
263                     SEC_ERASE_SIZE);
264
265         while (1) {
266             XQspiPs_PolledTransfer(QspiPtr, ReadStatusCmd,
267                         FlashStatus,
268                         sizeof(ReadStatusCmd));
269
270             if ((FlashStatus[1] & 0x01) == 0) {
271                 break;
272             }
273         }
274
275         Address += SECTOR_SIZE;
276     }
277 }
278
279 int FlashReadID(void)
280 {
281     int Status;
282
283     /* Read ID in Auto mode.*/
284     WriteBuffer[COMMAND_OFFSET]   = READ_ID;
285     WriteBuffer[ADDRESS_1_OFFSET] = 0x23;       /* 3 dummy bytes */
286     WriteBuffer[ADDRESS_2_OFFSET] = 0x08;
287     WriteBuffer[ADDRESS_3_OFFSET] = 0x09;
```

```
289     Status = XQspiPs_PolledTransfer(&QspiInstance, WriteBuffer, ReadBuffer,
290                 RD_ID_SIZE);
291     if (Status != XST_SUCCESS) {
292         return XST_FAILURE;
293     }
294
295     xil_printf("FlashID=0x%x 0x%x 0x%x\n\r", ReadBuffer[1], ReadBuffer[2],
296            ReadBuffer[3]);
297
298     return XST_SUCCESS;
299 }
300
301 void FlashQuadEnable(XQspiPs *QspiPtr)
302 {
303     u8 WriteEnableCmd = {WRITE_ENABLE_CMD};
304     u8 ReadStatusCmd[] = {READ_STATUS_CMD, 0};
305     u8 QuadEnableCmd[] = {WRITE_STATUS_CMD, 0};
306     u8 FlashStatus[2];
307
308     if (ReadBuffer[1] == 0x9D) {
309
310         XQspiPs_PolledTransfer(QspiPtr, ReadStatusCmd,
311                     FlashStatus,
312                     sizeof(ReadStatusCmd));
313
314         QuadEnableCmd[1] = FlashStatus[1] | 1 << 6;
315
316         XQspiPs_PolledTransfer(QspiPtr, &WriteEnableCmd, NULL,
317                   sizeof(WriteEnableCmd));
318
319         XQspiPs_PolledTransfer(QspiPtr, QuadEnableCmd, NULL,
320                     sizeof(QuadEnableCmd));
321     }
322 }
```

在该程序中，我们首先定义了控制 Flash 读写的一些指令，部分数据的偏移量，以及 Flash 的一些相关参数，这些参数根据 Flash 型号的不同有所区别，具体请查阅芯片资料即可。之后定义了一些读写、擦除、使能等相关函数。在主函数中运行函数 QspiFlashPolledExample(XQspiPs *QspiInstancePtr, u16 QspiDeviceId)，该函数对 FLASH 进行读写操作，对比写入和读出的数据是否一致，完成对于 FLASH 的测试。

在该函数中 1、初始化 QSPI 驱动；2、初始化读写 buffer；3、设置 QSPI 工作模式，分频系数，使能片选信号，读出 Flash ID，使能 Quad 模式；4、向 Flash 写入数据，之后通过不同方式从 Flash 中读取数据，对比写入的数据与读出的数据是否相同，验证读写的正确性。

该程序是由官方的例程 qspi_flash_test_bsp_xqspips_flash_polled_example_1 修改而来，例程中由详细的注释说明，读者可以自行查阅。

程序编辑完成后快捷键 Ctrl + S 保存并编译程序。

12.4 下载验证

首先我们将使用 Micro USB 线与 JTAG 接口连接，另外一端与电脑连接。然后使用 Micro USB 连接线将开发板的 UART 接口与电脑连接，用于串口通信。最后连接开发板的电源，并打开电源开关。

在 SDK 软件下方的 SDK Terminal 窗口中点击右上角的加号设置并连接串口。然后下载本次实验软件程序， 下载完成后， 在下方的 SDK Terminal 中可

以看到应用程序打印的信息如下图所示：

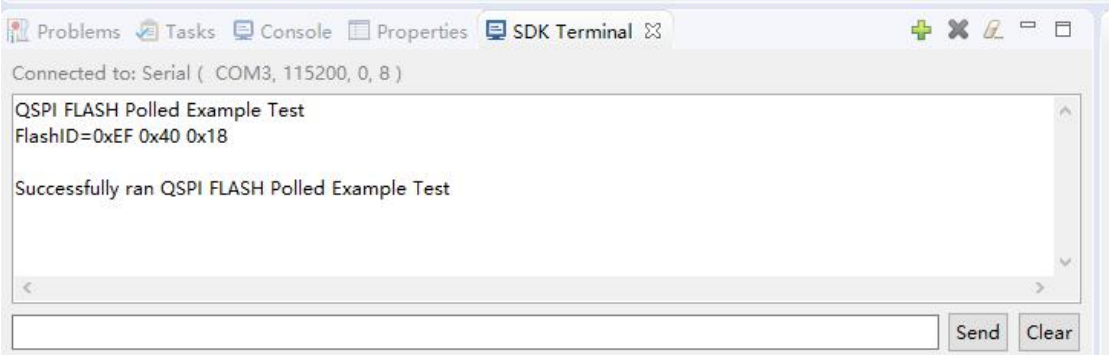


图 12.4.1 实验结果

说明实验成功，可以对 Flash 进行正常的读写操作，写入的数据与读出的数据相同。

12.5 章节小结

Flash 作为一种常见的数据存储介质，经常被用到各种嵌入式系统之中，这需要研发人员熟练掌握关于它的各种操作。重点是数据的读写、擦除以及校验，希望读者在此实验的基础上更加深入的学习相关知识。

第 13 章　　PL 读写 PS 端 DDR 数据

将高性能 ARM Cotex-A 系列处理器与高性能 FPGA 集成到同一块芯片上是 ZYNQ SOC 的最大优势，那么两者之间数据交互的速度、准确度等方面的因素就是能否发挥出这种优势的关键所在。在本实验中，我们就通过 PL 端读写 PS 端 DDR 数据的实验，来更加深入的了解 PS 和 PL 端数据交互的相关知识。

本章包括以下内容：

13.1 实验简介

13.2 硬件设计

13.3 软件设计

13.4 下载验证

13.5 本章小结

13.1 实验简介

在本次实验中，我们使用 AXI 总线来实现实验目的，接下简单讲解关于 AXI 总线的相关知识。

AXI 简介：

AXI（Advanced eXtensible Interface）是一种总线协议，该协议是 ARM 公司提出的 AMBA（Advanced Microcontroller Bus Architecture）3.0 协议中最重要的部分，是一种面向高性能、高带宽、低延迟的片内总线。它的地址/控制和数据相位是分离的，支持不对齐的数据传输，同时在突发传输中，只需要首地址，同时分离的读写数据通道、并支持显著传输访问和乱序访问，并更加容易就行时序收敛。AXI 是 AMBA 中一个新的高性能协议。AXI 技术丰富了现有的 AMBA 标准内容，满足超高性能和复杂的片上系统（SoC）设计的需求。

AXI 特点：

单向通道体系结构。信息流只以单方向传输，简化时钟域间的桥接，减少门数量。当信号经过复杂的片上系统时，减少延时。

支持多项数据交换。通过并行执行猝发操作，极大地提高了数据吞吐能力，可在更短的时间内完成任务，在满足高性能要求的同时，又减少了功耗。

独立的地址和数据通道。地址和数据通道分开，能对每一个通道进行单独优化，可以根据需要控制时序通道，将时钟频率提到最高，并将延时降到最低。

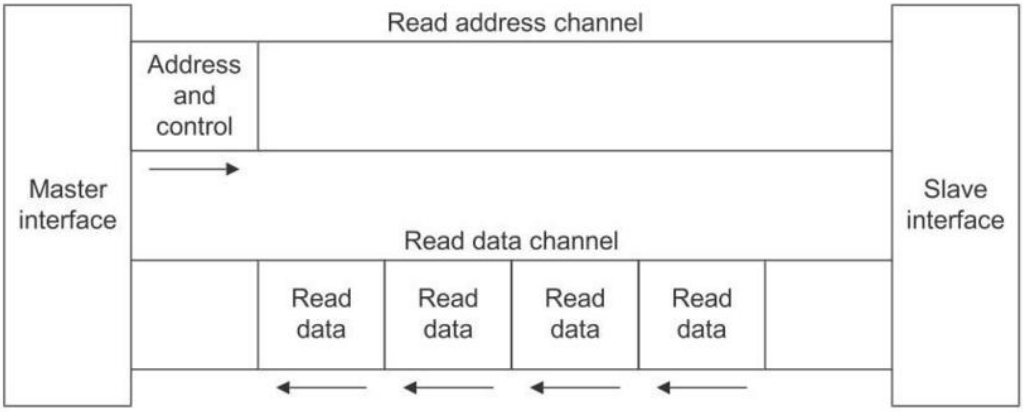
增强的灵活性。AXI 技术拥有对称的主从接口，无论在点对点或在多层系统中，都能十分方便地使用 AXI 技术。

AXI 协议和 AXI 协议定义的基础事务：

AXI 总线共有 5 个独立的通道,分别是 read address channel (ARxxx)，write address channel(AWxxx),read data channel(Rxxx), write data hannel(Wxxx),

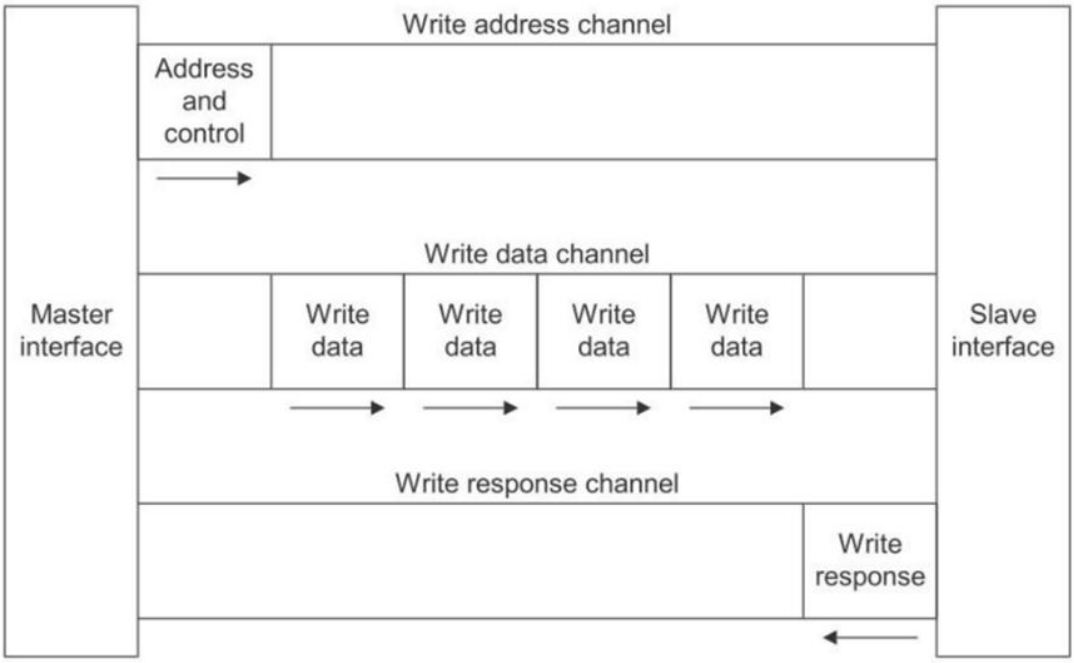
write response channel(Bxxx)。每一个 AXI 传输通道都是单方向的。

每一个 transaction 都有地址和控制信息在地址通道（address channel）中，用来描述被传输数据的性质。



读 transcation 的结构图如下：读请求和数据返回

图 13.1.1 读 transcation 的结构图



写 transcation 的结构如下：

图 13.1.2 写 transcation 的结构图

这 5 条独立的通道都包含一个信息信号和一个双路的 VALID、READY 握手机制。信息源通过 VALID 信号来指示通道中的数据和控制信息什么时候有效。目地源用 READY 信号来表示何时能够接收数据。读数据和写数据通道都包括一个 LAST 信号，用来指明一个事物传输的最后一个数据。

读和写事务都有他们自己的地址通道，这地址通道携带着传输事务所必须的地址和控制信息。

读数据通道传送着从设备到主机的读数据和读响应信息，读数据总线宽度可以是 8,16,32,64,128,256,512,1024 位。读响应信息指明读事务的完成状态。

写数据通路传送着主机向设备的写数据，写数据总线宽度可以是 8,16,32,64,128,256,512,1024 位。每八位数据都会有一个 byte lane strobe ，用来指明数据总线上面的哪些 byte 有效。写响应通道提供了设备响应写事务的一种方式。完成信号每一次事务只会产生一个。

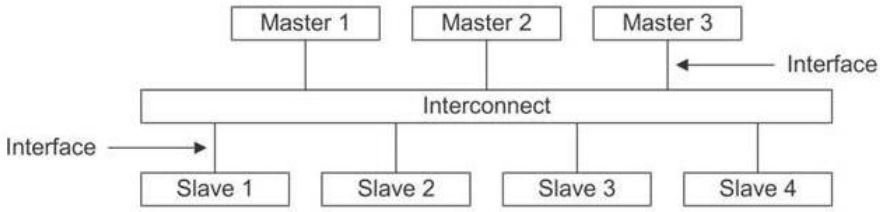


Figure 1-3 Interface and interconnect

主机和设备的接口和互联图如下：

图 13.1.3 主机和设备的接口和互联图

传输地址信息和数据都是在 VALID 和 READY 同时为高时有效。

突发式读的时序图如下：

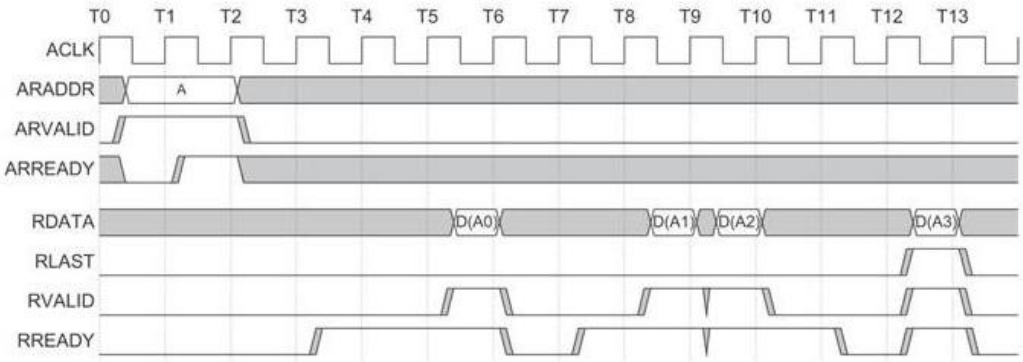
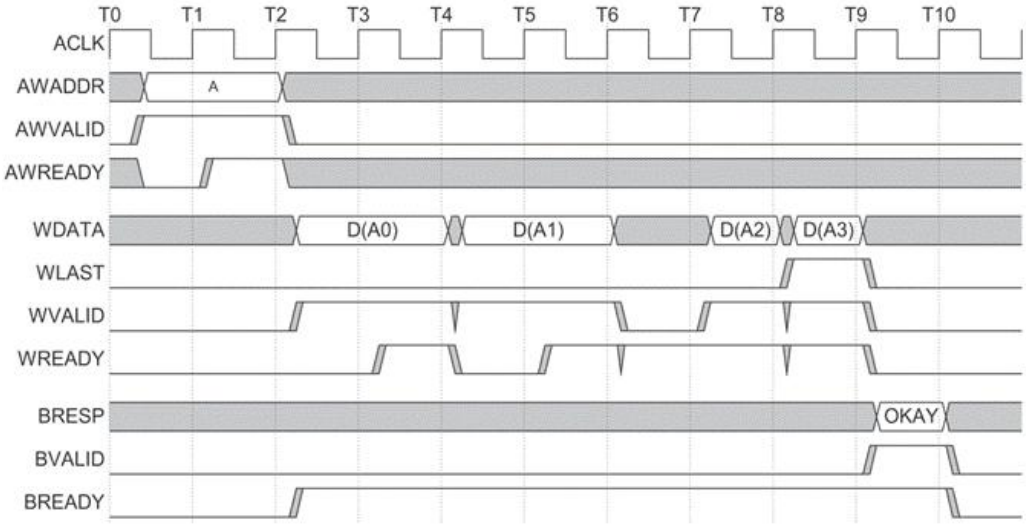


图 13.1.4 突发式读的时序图

设备会在第一次突发式读完成后处理第二次突发式读数据。也就意味着，主机一开始传送了两个地址给设备。设备在完全处理完第一个地址的数据之后才开始处理第二个地址的数据。



突发式写时序图如下：

图 13.1.5 突发式写时序图

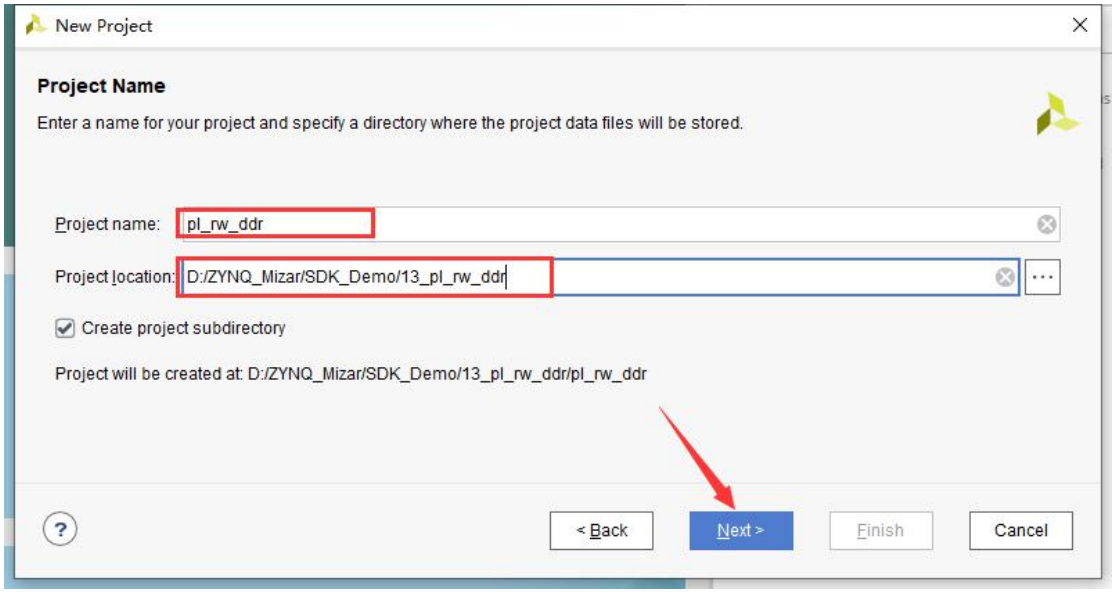
这一过程的开始时，主机发送地址和控制信息到写地址通道中，然后主机发送每一个写数据到写数据通道中。当主机发送最后一个数据时，WLAST 信号就变为高。当设备接收完所有数据之后他将一个写响应发送回主机来表明写事务完成。

AXI 协议支持乱序传输。它给每一个通过接口的事务一个 IDtag。协议要求相同 ID tag 的事务必须有序完成，而不同 ID tag 可以乱序完成。

受篇幅所限，这里关于 AXI 总线协议的相关知识无法详尽的向读者说明，希望读者能够自行查阅相关资料，学习相关的知识。

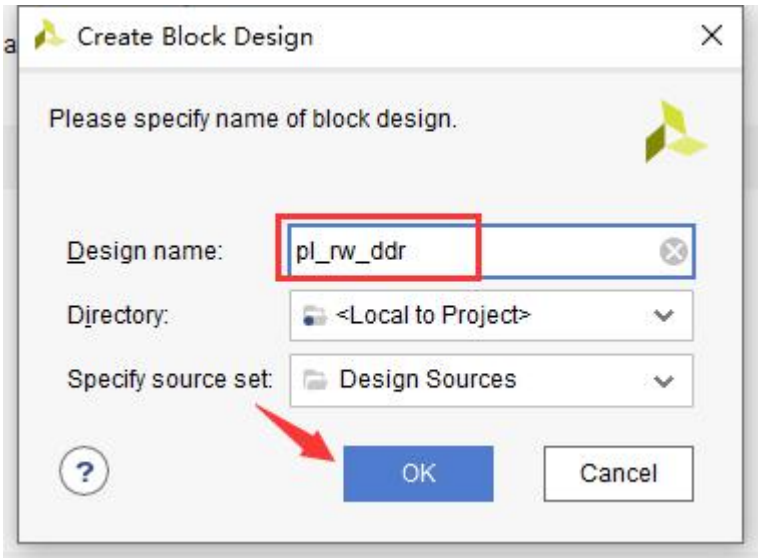
13.2 硬件设计

双击打开 Vivado 2018.3 软件，创建一个新的工程并命名为“pl_rw_ddr”，如



下图所示，芯片型号根据所用开发板的型号选取。

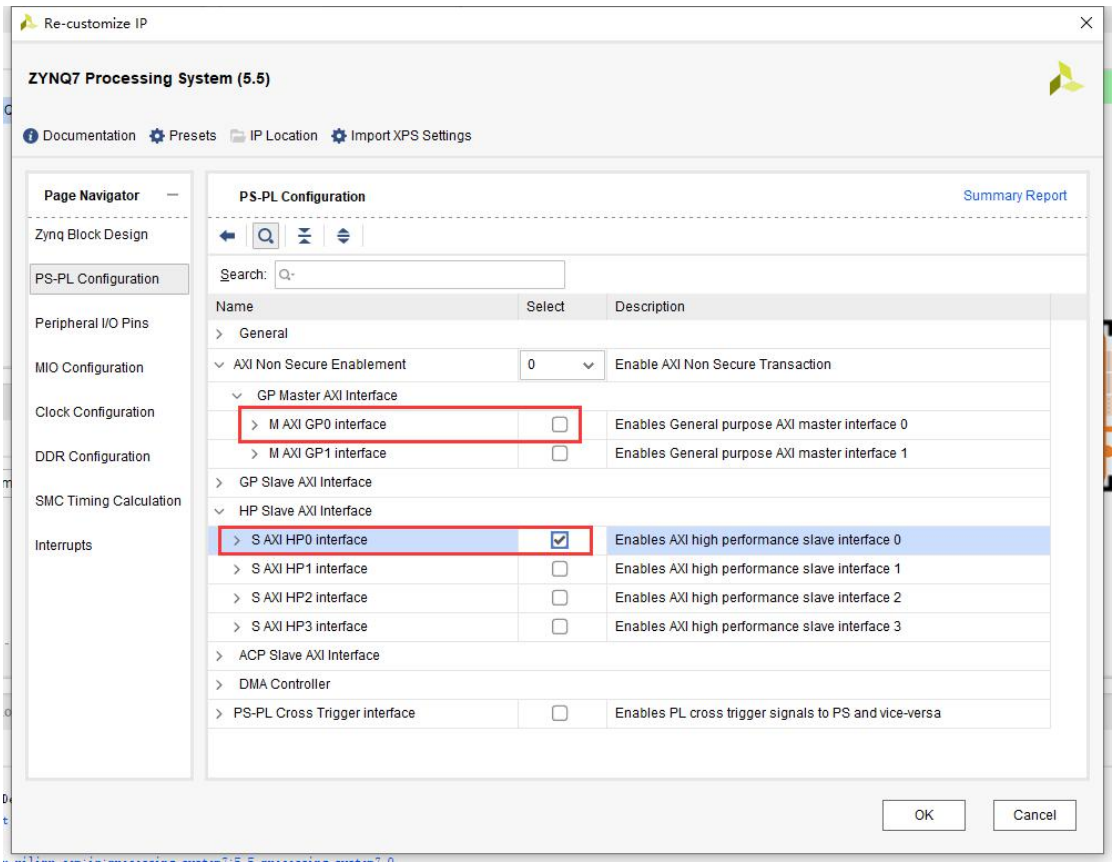
图 13.2.1 创建新工程



创建完成后新建 Block Design，并命名为“pl_rw_ddr”，如下图所示：

图 13.2.2 创建 Block Design

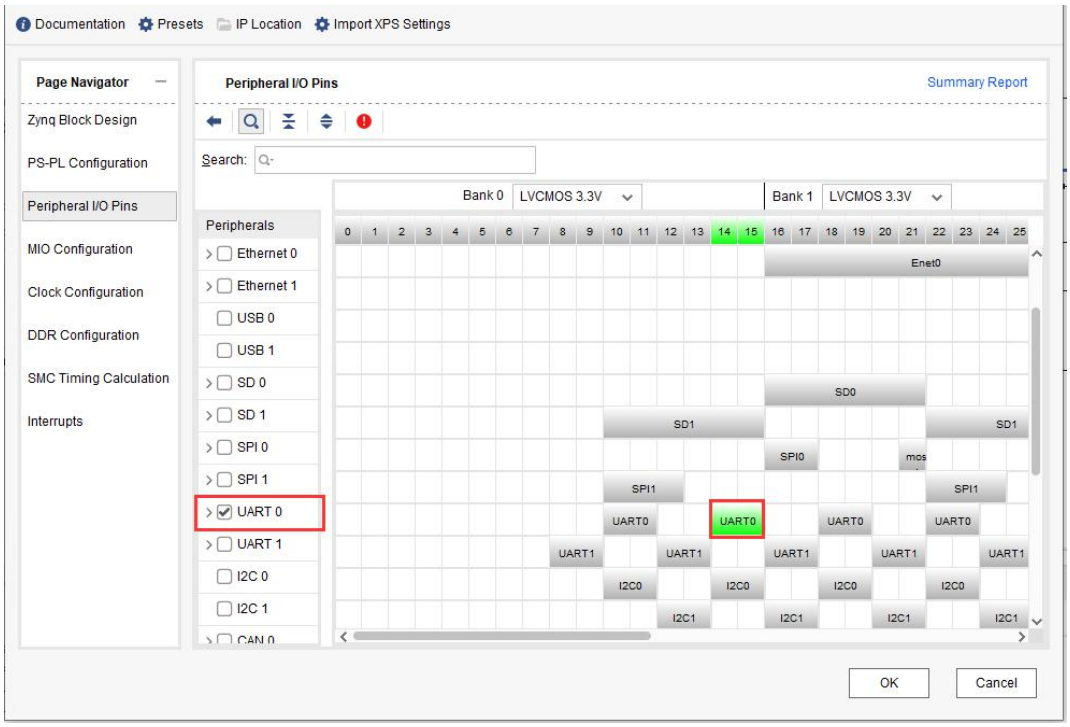
创建完成后添加 ZYNQ7 Processing System 模块，并按照下图所示配置模块



参数。

图 13.2.3 移除 GP Master AXI Interface 接口，勾选 S AXI HP0 Interface

图 13.2.4 配置 UART



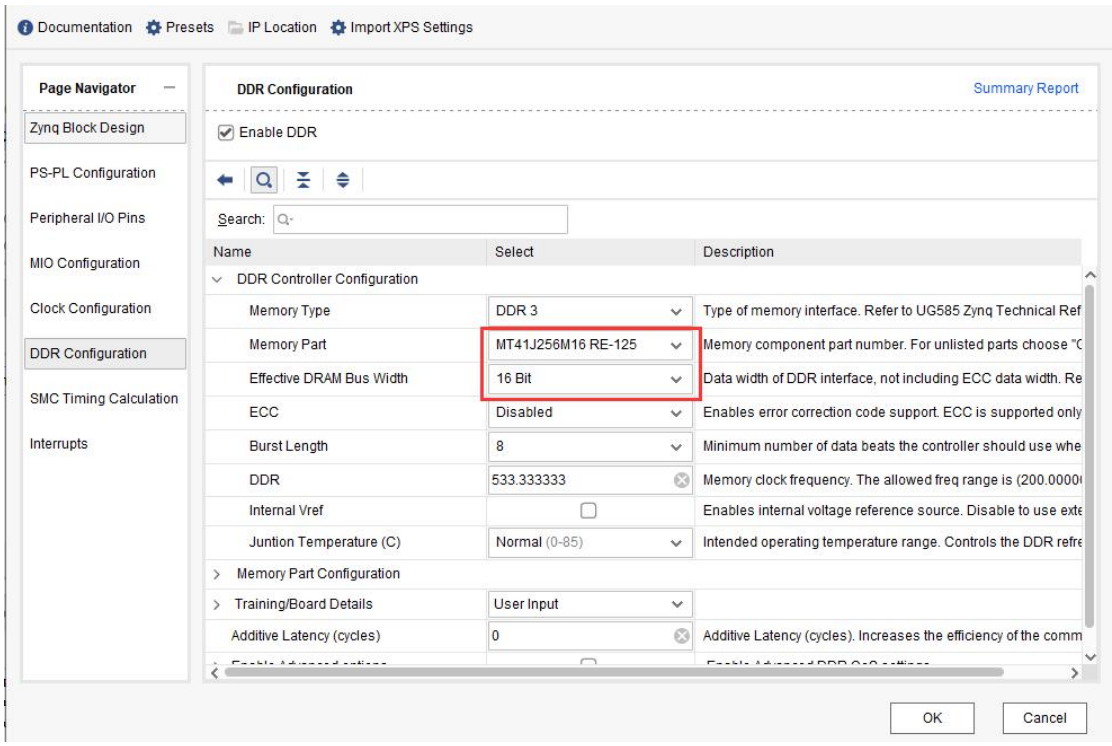


图 13.2.5 配置 DDR 控制器

配置完成后点击“OK”，完成配置。

配置完成后的接口如下图所示：

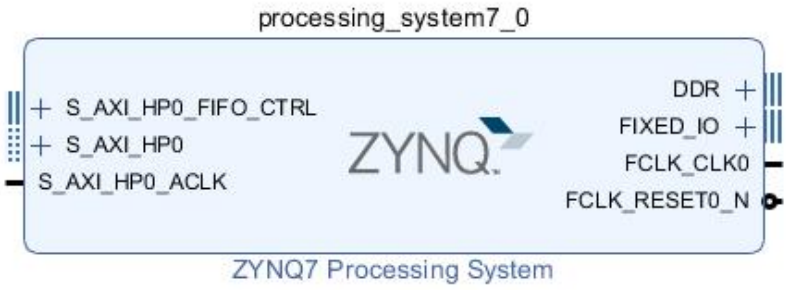
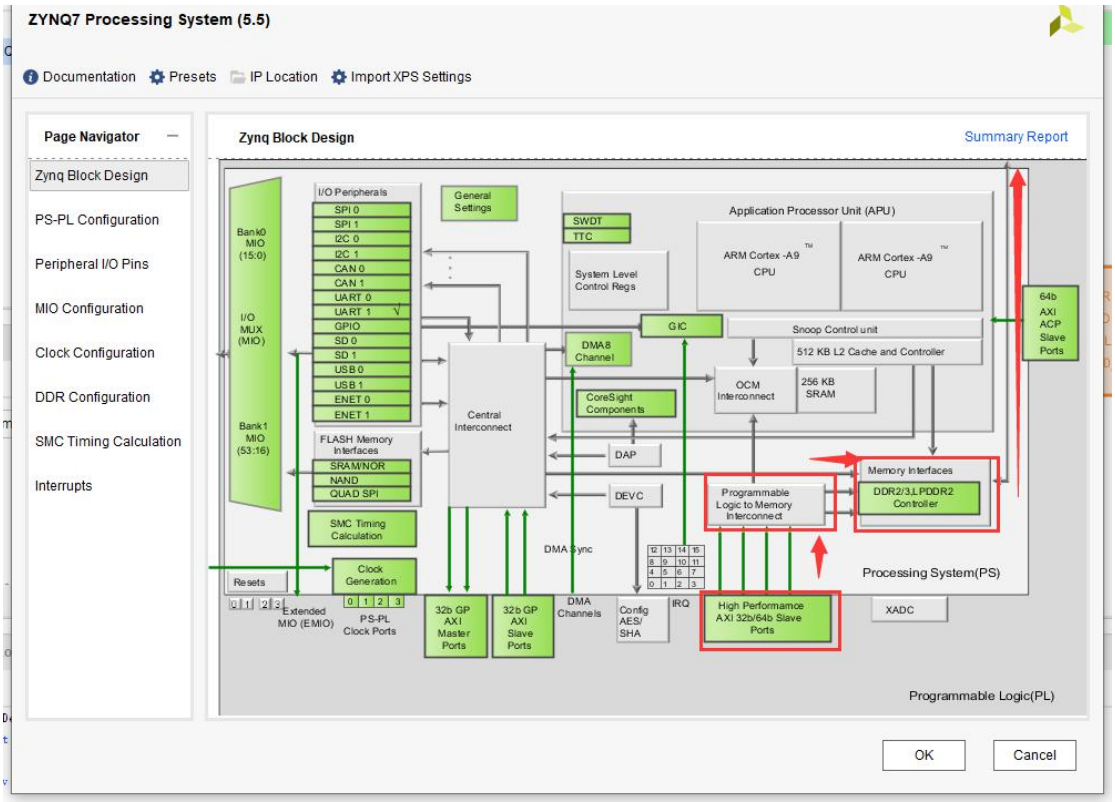


图 13.2.6 ZYNQ7 模块接口

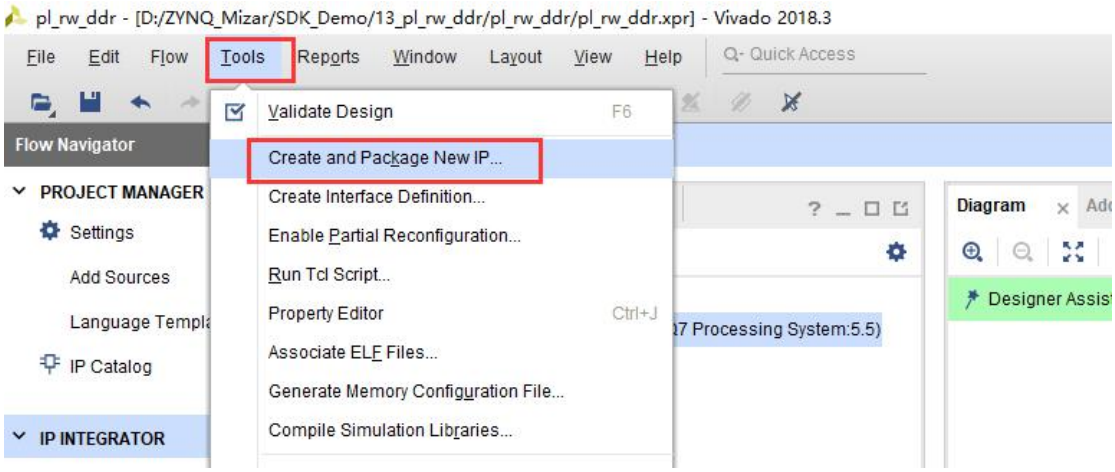
在本次实验中，我们使能了 S_AXI_HP0 接口，它是 PS 端的 AXI 高性能接口，作为一个从接口，PL 可以通过它访问 PS 内的存储设备，包括 OCM 和 DDR。从下图的数据通路可以看出，我们需要在 PL 内自定义一个 IP 核作为主机，通过



PS 的 S_AXI_HP0 从机接口来实现与 DDR 的通信，对 DDR3 存储器进行读写操作。

图 13.2.7 数据通路

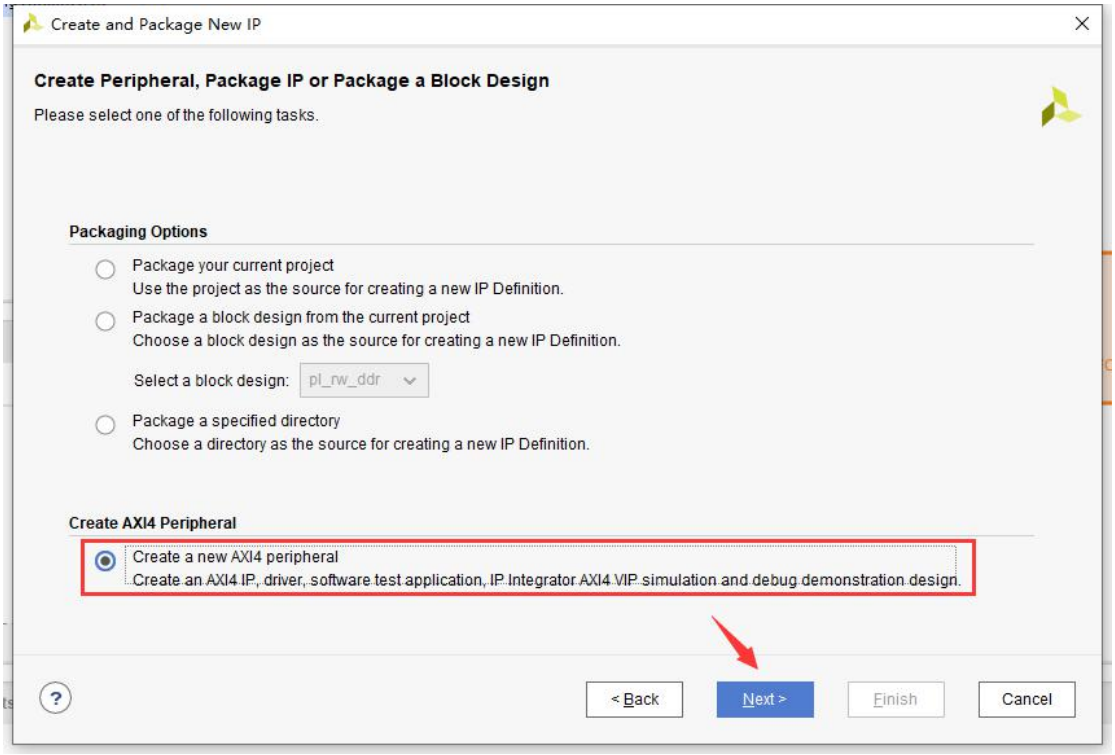
接下来创建自定义 IP 核，在菜单栏中点击“Tools”，在弹出的列表中选择



“Create and Package New IP”，如下图所示：

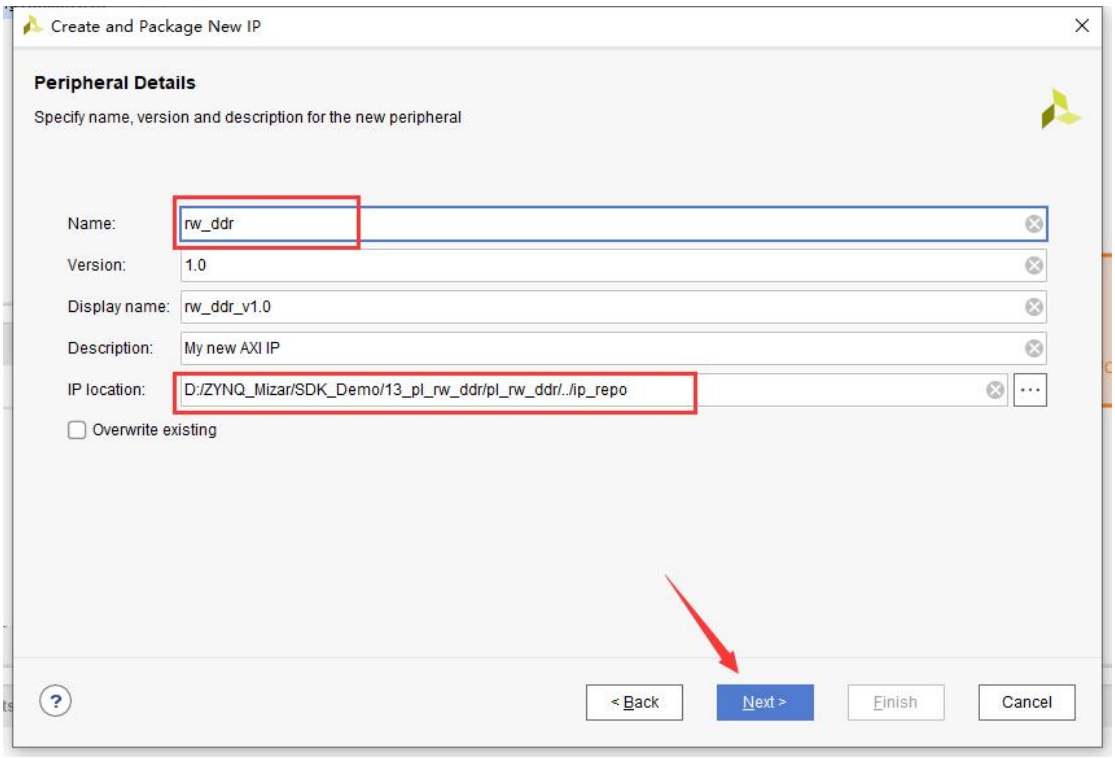
图 13.2.8 创建自定义 IP 核

在弹出的对话框中点击“Next”，之后在下图所示的界面中选择“Create a



new AXI4 peripheral”，点击“Next”：

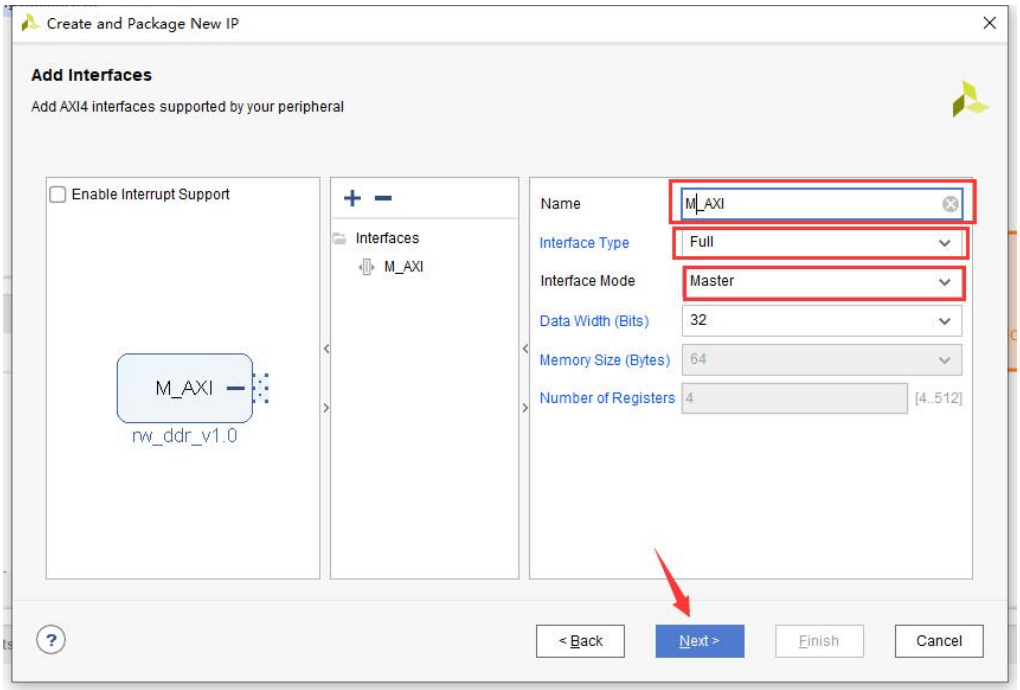
图 13.2.9 创建 AXI4 协议 IP 核



接下来为 IP 核命名为“rw_ddr”。保存路径默认即可，如下图所示：

图 13.2.10 为 IP 核命名

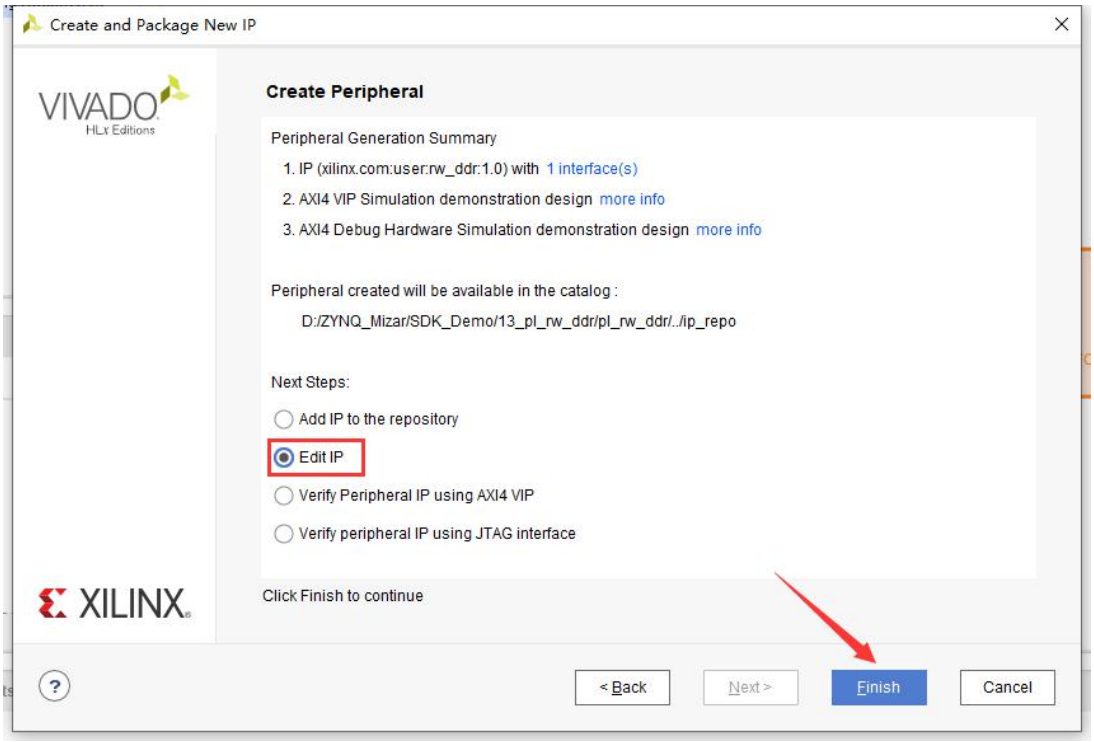
之后点击“Next”进入参数设置页面，修改接口名称为“M_AXI”，选择接



口类型为“Full”，接口模式为“Master”，数据位宽为“32”。如下图所示：

图 13.2.11 IP 核参数设置

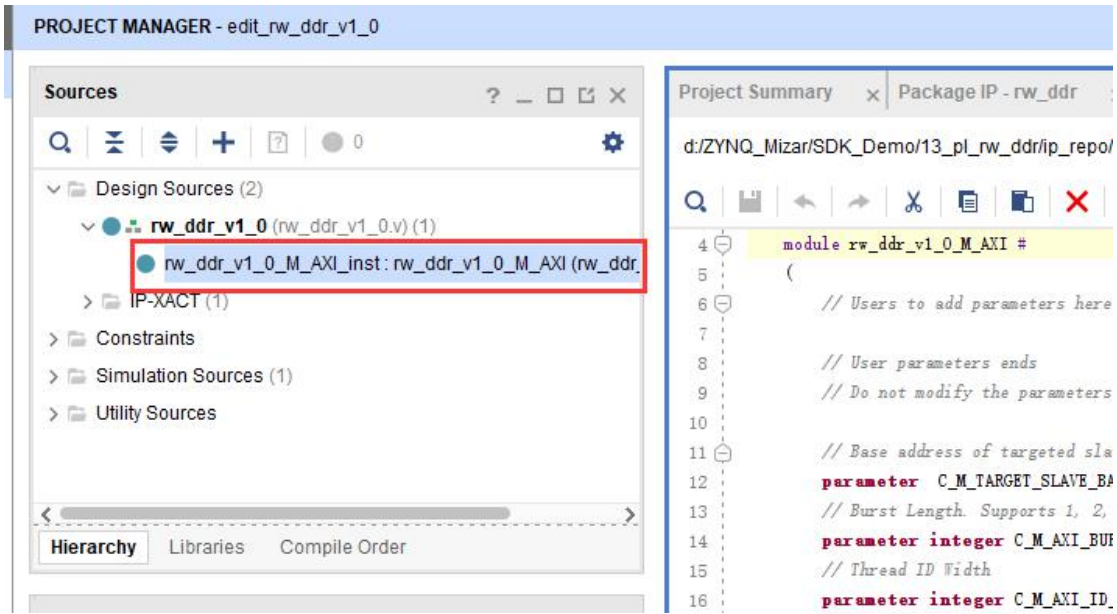
设置完成后点击“Next”，在之后弹出的页面选择“Edit IP”，之后点击“Finish”



完成创建。

图 13.2.12 完成创建

之后软件自动打开以一个新的工程，在该工程中，我们可以对刚创建的 IP 核进行编辑。双击下图所示位置可以看到创建 IP 核时生成的测试程序，该程序



实现了 AXI4 协议下对于 DDR 的读写测试。因为该实验中无需该 IP 核实现其他功能，所以不需要对代码进行修改，直接关闭即可，IP 核就创建完成了。

图 13.2.13 IP 核测试 DDR 读写代码

回到之前工程的页面在左侧“Flow Navigator”一栏点击“IP Catalog”，然后在右侧的 IP 目录中可以看到我们前面所创建的 IP 核——rw_ddr，该 IP 已经自动添加到了当前工程的 IP 库中。

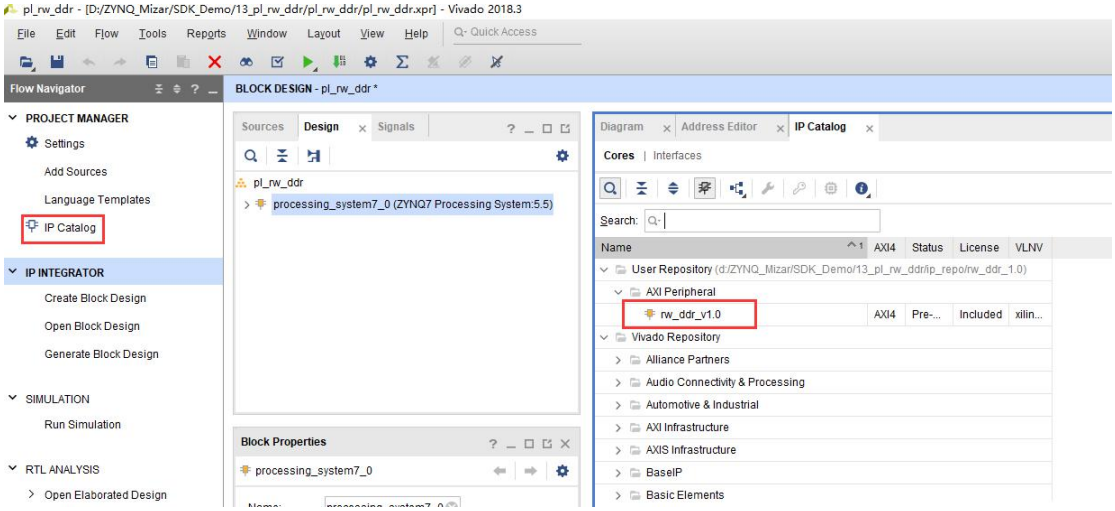
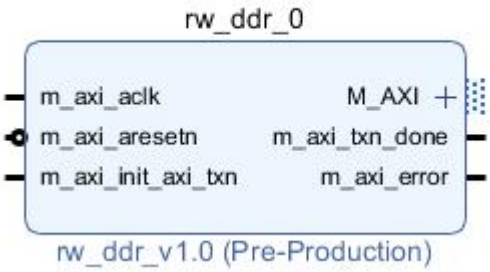


图 13.2.14 IP 核目录

在 Diagram 页面打开 IP 核搜索添加窗口，查找刚创建的 IP 核，双击添加到工程内。

图 13.2.15 添加 IP 核



添加到工程中的 IP 核如下图所示：

图 13.2.16 IP 核添加完成

这里简单说明以下模块各接口的作用。M_AXI 是 AXI-Full 类型的主机接口，通过该接口可以实现对 PS 端的 DDR3 进行读写操作。IP 核在检测到 m_axi_init_axi_txn 端口的上升沿信号后会启动读写过程，并将读出的数据与写入的数据进行比较，比较完成后 m_axi_txn_done 输出高电平，表明该过程结束。另外，m_axi_error 信号会指示整个过程是否出错。如果在读写过程中出错，或者在比较的过程中发现读出的数据与写入的数据不一致，那么 m_axi_error 将会输出高电平。

之后双击该模块进行参数配置，具体参数如下图所示：

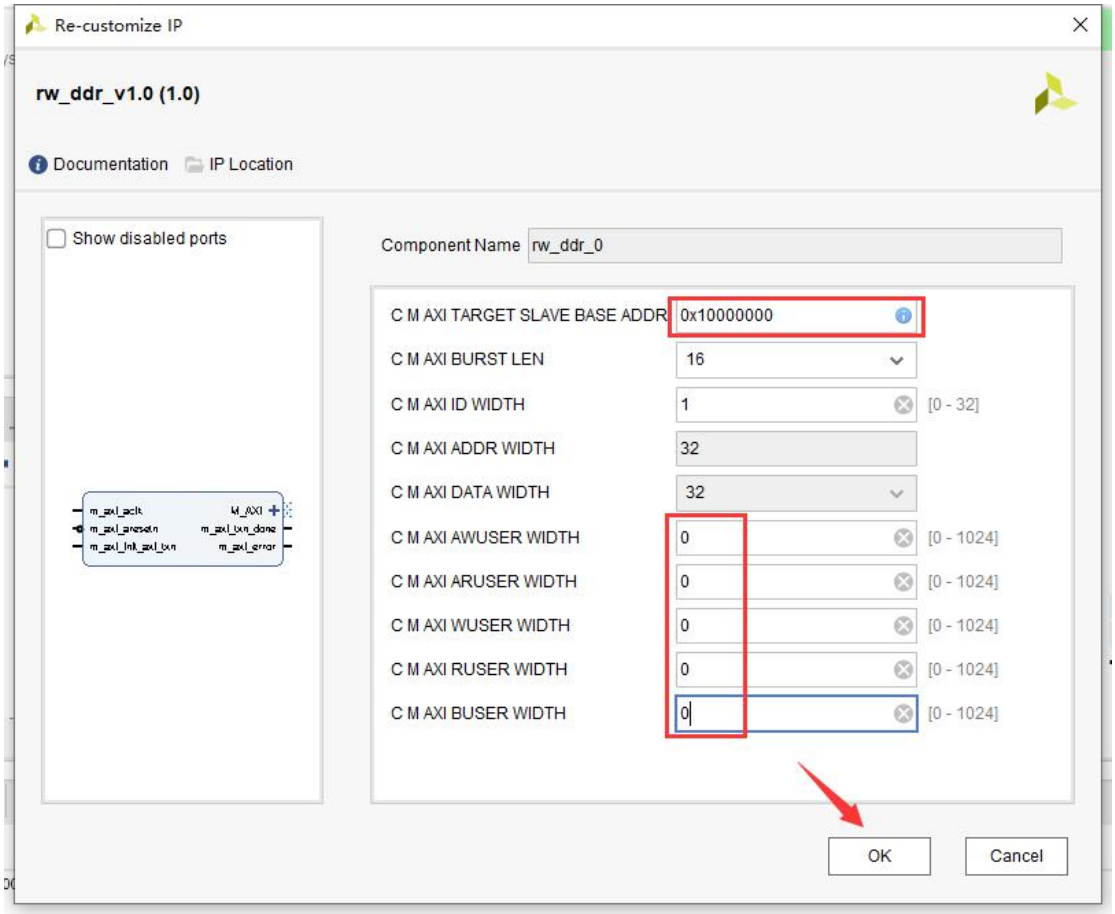


图 13.2.17 IP 核参数

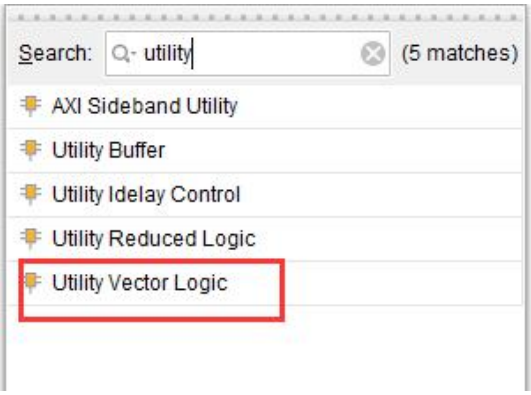
为什么此处的基地址设置为 0x1000_0000 而不是从 0x0000_0000 开始呢？这就要根据芯片内存的地址映射来决定了。查看下图，我们可以看到相关的地址映射关系，从图中可以看到，从 0x0000_0000 开始并不是 DDR 的地址而是 OCM 的地址，如果从 0x0000_0000 开始读写的就是 OCM 的数据。此处我们设置基地址为 0x1000_0000，之前的内存保留为 OCM 和程序运行的内存。

Table 29-1: **Initial OCM/DDR Address Map**

Address Range (Hex)	Size	CPUs/ACP	Other Masters
0000_0000 - 0000_FFFF	64 KB	OCM	OCM
0001_0000 - 0001_FFFF	64 KB	OCM	OCM
0002_0000 - 0002_FFFF	64 KB	OCM	OCM
0003_0000 - 0003_FFFF	64 KB	Reserved	Reserved
0004_0000 - 0007_FFFF	256 KB	Reserved	Reserved
0008_0000 - 000B_FFFF	256 KB	Reserved	DDR
000C_0000 - 000C_FFFF	64 KB	Reserved	DDR
000D_0000 - 000D_FFFF	64 KB	Reserved	DDR
000E_0000 - 000E_FFFF	64 KB	Reserved	DDR
000F_0000 - 000F_FFFF	64 KB	OCM3 (alias)	DDR
0010_0000 - 3FFF_FFFF	1,023 MB	DDR	DDR
FFFC_0000 - FFFC_FFFF	64 KB	Reserved	Reserved
FFFD_0000 - FFFD_FFFF	64 KB	Reserved	Reserved
FFFE_0000 - FFFE_FFFF	64 KB	Reserved	Reserved
FFFF_0000 - FFFF_FFFF	64 KB	OCM3	OCM3

图 13.2.18 地址映射关系

设置完成之后，添加三个 Utility Vector Logic IP 核，然后将其配置成非门，位宽为 1，作为反向器使用。在前面我们讲到模块的三个信号“m_axi_init_axi_txn”，“m_axi_txn_done”和“m_axi_error”，“m_axi_init_axi_txn”检测到上升沿开始读写 DDR，“m_axi_txn_done”和“m_axi_error”指示读写的状态。此处我们将这三个信号分别连接按键和两个 LED 等来控制及观察模块运行状态。查看原理图可知，按键按下产生下降沿信号，LED 灯低电平点亮。这与我们希望的按下按键读写 DDR，读写完成输出高电平 LED 点亮，读写出错输出高电平 LED 点亮刚好相反，所以此



处添加反向器来改变信号状态。

图 13.2.19 添加 Utility Vector Logic IP 核

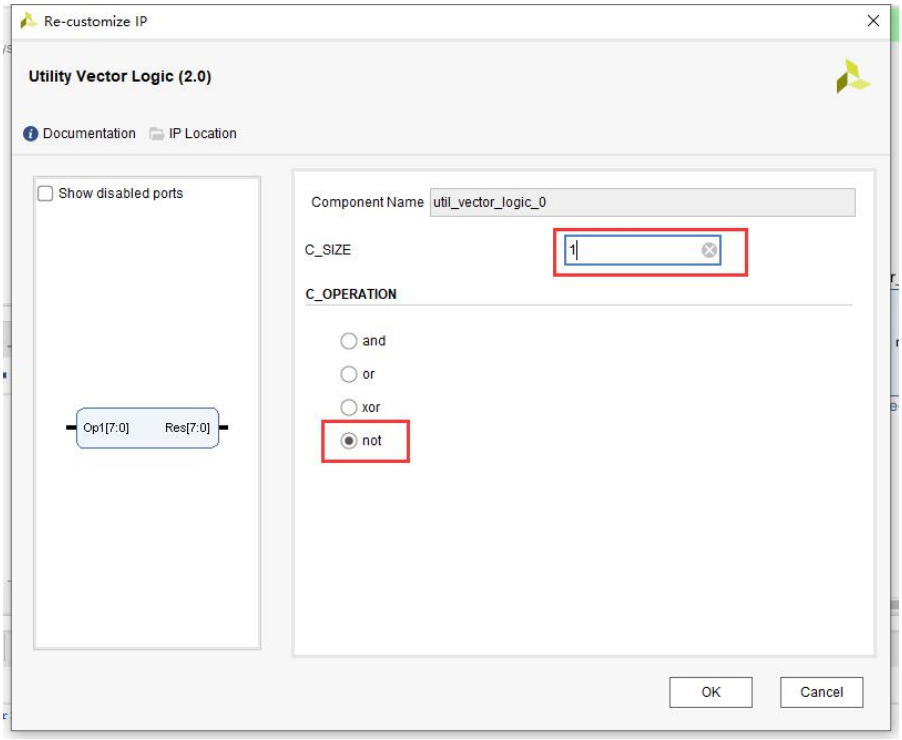
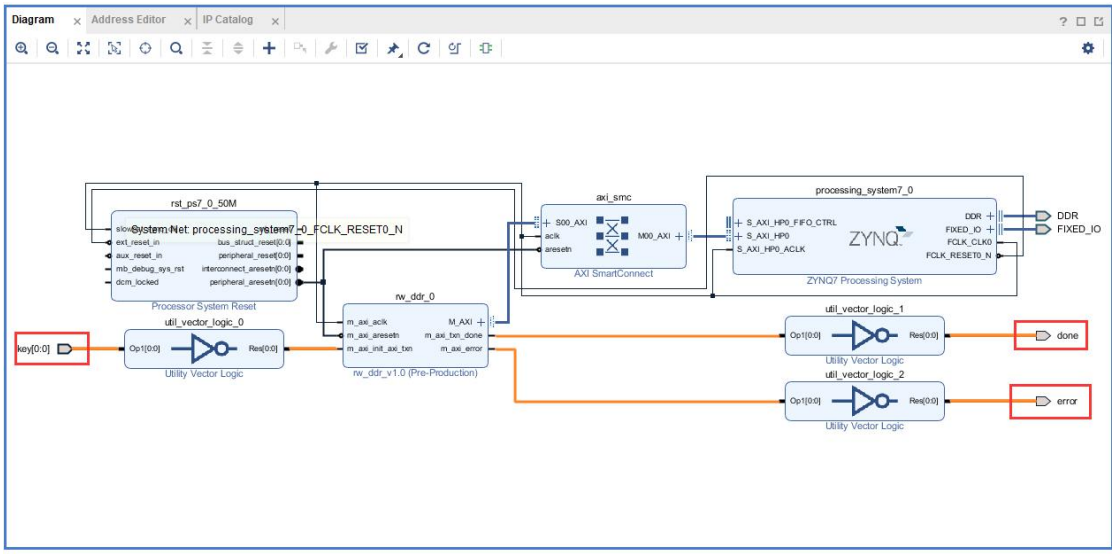


图 13.2.20 设置 IP 核参数

之后执行“Run Block Automation”和“Run Connection Automation”自动连接各模块。执行完成后手动连接 rw_ddr IP 核以及三个非门的端口，并将三个信



号引出，修改信号名称便于识别。连接完成后的设计图如下图所示：

图 13.2.21 连接各信号

连接完成后保存并验证设计是否正确。之后生成顶层 HDL。

点击左侧“Open Elaborated Design”打开引脚约束设置窗口。

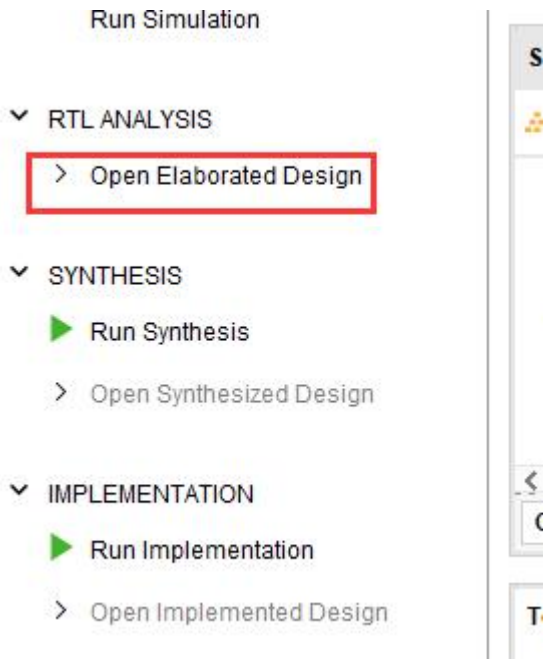
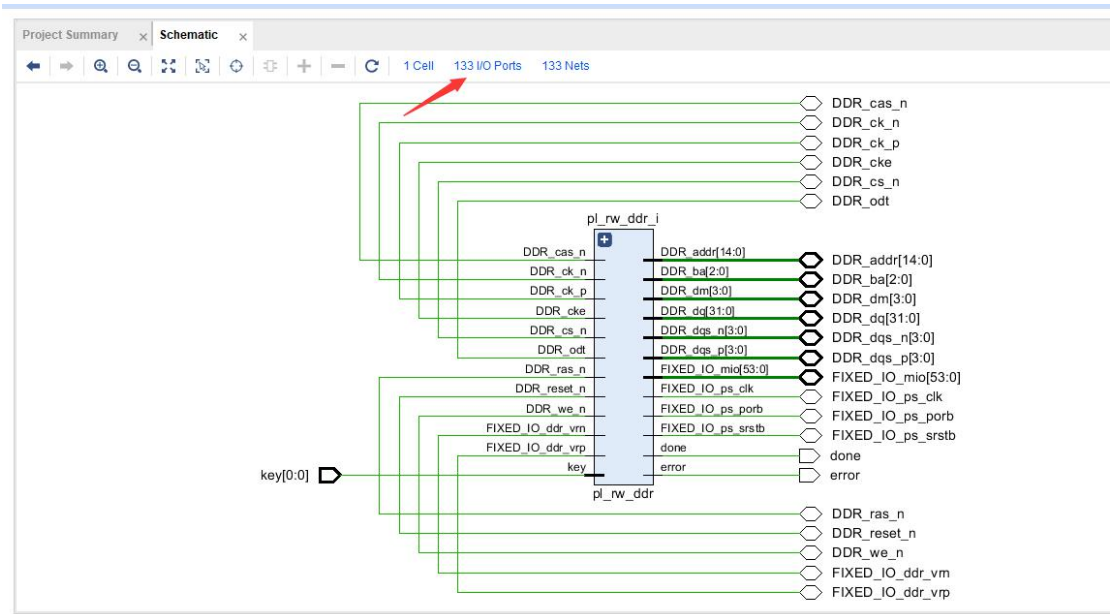
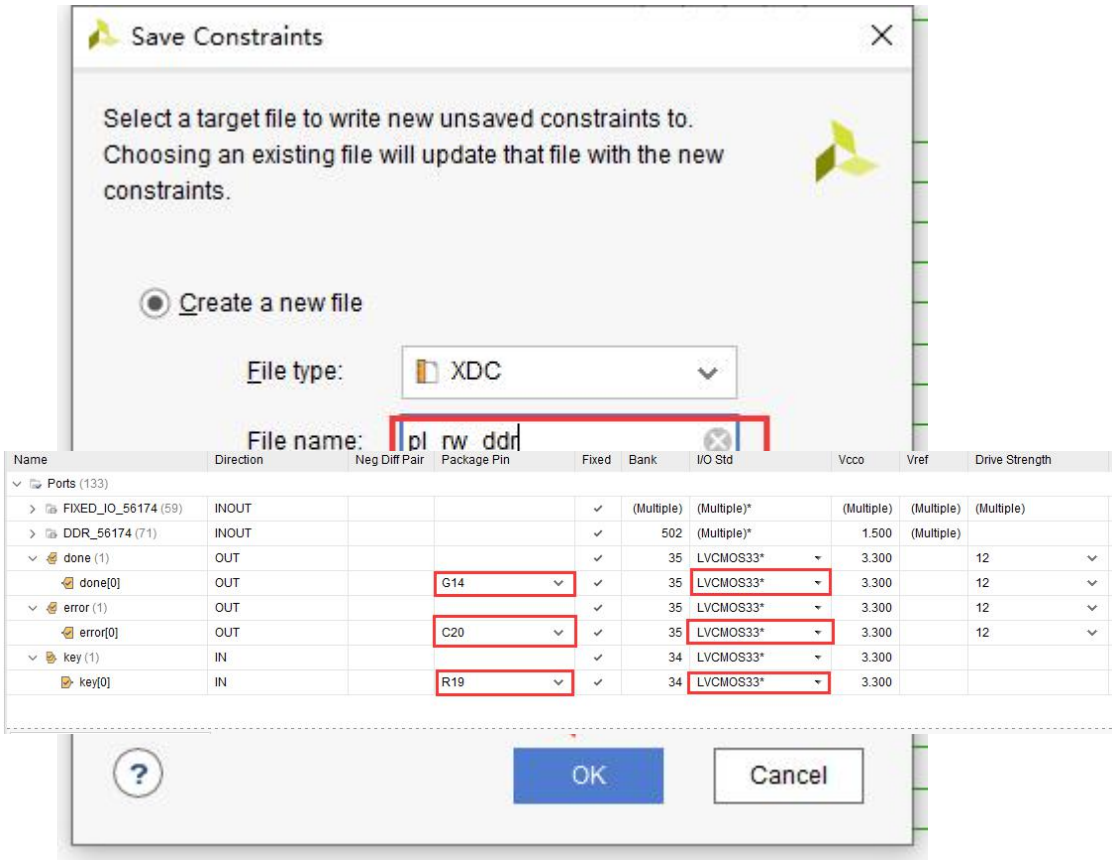


图 13.2.22 打开引脚约束



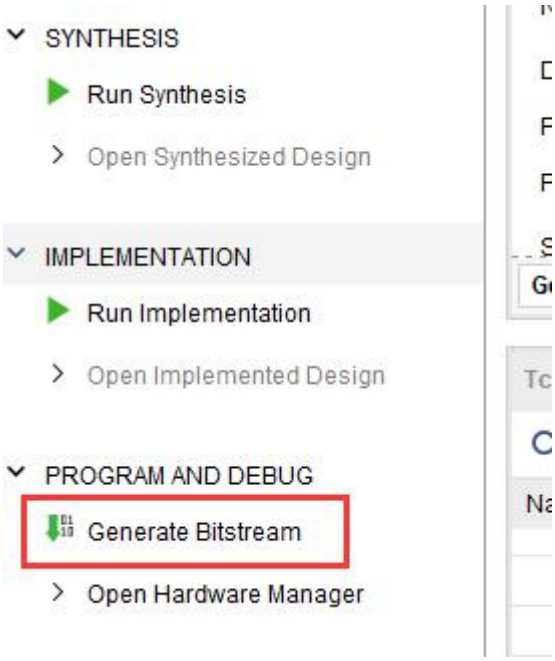
点击 I/O port 打开设置页面，如下图所示设置引脚约束：

图 13.2.23 设置引脚约束



设置完成后快捷键 Ctrl + S 保存，文件命名为“pl_rw_ddr”。如下图所示：

图 13.2.24 保存并命名



之后点击 Generate Bitstream 编译综合生成 Bitstream 文件。

图 13.2.25 生成 Bitastream

在生成 Bitstream 之后， 在菜单栏中选择 File > Export > Export hardware 导出硬件，并在弹出的对话框中，勾选“Include bitstream”。然后在菜单栏选择 File > Launch SDK，启动 SDK 软件。

13.3 软件设计

在打开的 SDK 软件中新建一个工程并命名为“pl_rw_ddr”，工程模板选择空白。新建完成后为工程添加一个源文件并命名为“main.c”。之后在编辑框中输

```
system.hdf    system.mss    *main.c
1  #include <stdio.h>
2  #include "xil_cache.h"
3  #include "xparameters.h"
4  #include "xparameters_ps.h"
5  #include "xil_printf.h"
6  #include "xil_io.h"
7
8  #define DDR_BASEARDDR  0x10000000 //从设置基地址开始读取
9  #define DDR_HIGHADDR   XPAR_DDR_MEM_HIGHADDR
10
11 int main()
12 {
13     int i=0;
14     char A;
15     int rev;
16
17     Xil_DCacheDisable();
18     print("PL RW DDR TEST!\n\r");
19     print("Please input A to get data\n\r");
20     while(1){
21     scanf("%c",&A);
22     if(A=='A'){
23         printf("start\n\r");
24         while(i*4<4096){
25                     rev = Xil_In32(DDR_BASEARDDR+i*4);
26                     xil_printf("the address at  %x data is : %d \r\n" ,DDR_BASEARDDR+i*4, rev);
27                     ++i;
28             }
29             i=0;
30         }
31     }
32     return 0;
33
34 }
```

入以下代码：

在代码中检查串口收到的数据，到收到“A”时打印 DDR 中的数据。函数中我们用到了 Xil_In32(INTPTR Addr)这个函数，打开 xil_io.h 这个头文件，这个头文件是 cortexA9 可以直接控制的内存映射或者映射到了地址空间的 IO。在里面可以看到如下代码：

//从某个地址读数据

u8 Xil_In8(INTPTR Addr);

u16 Xil_In16(INTPTR Addr);

u32 Xil_In32(INTPTR Addr);

//向某个地址写数据。

void Xil_Out8(INTPTR Addr, u8 Value);

void Xil_Out16(INTPTR Addr, u16 Value);

void Xil_Out32(INTPTR Addr, u32 Value);

其中 8、16、32 表示一次读取的位数。

编辑完成后快捷键 Ctrl + S 保存并编译程序。

13.4 下载验证

按照之前的操作连接好电源等连接线，并在下方 SDK Terminal 窗口连接串口，连接完成后上电，将程序烧录到开发板上。

烧录完成后可以看到串口打印的数据，PL_LED1 和 PL_LED2 都是熄灭的，输入“A”并点击“Send”。

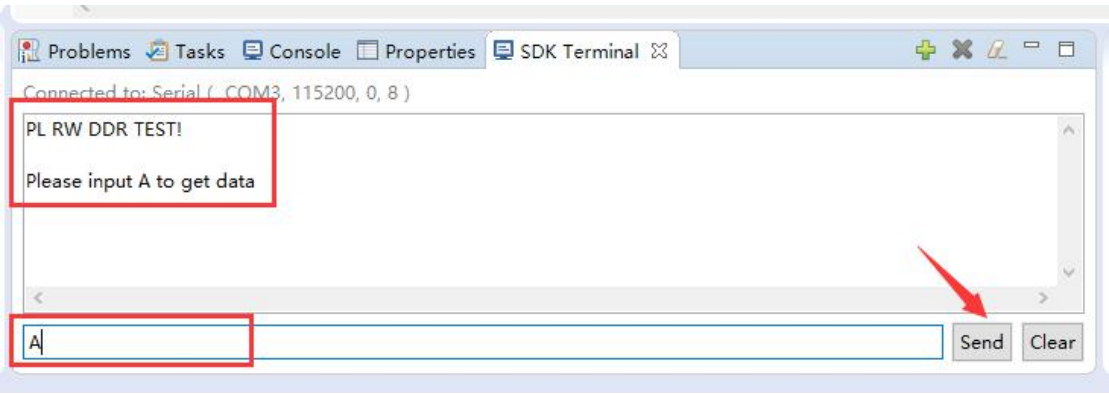
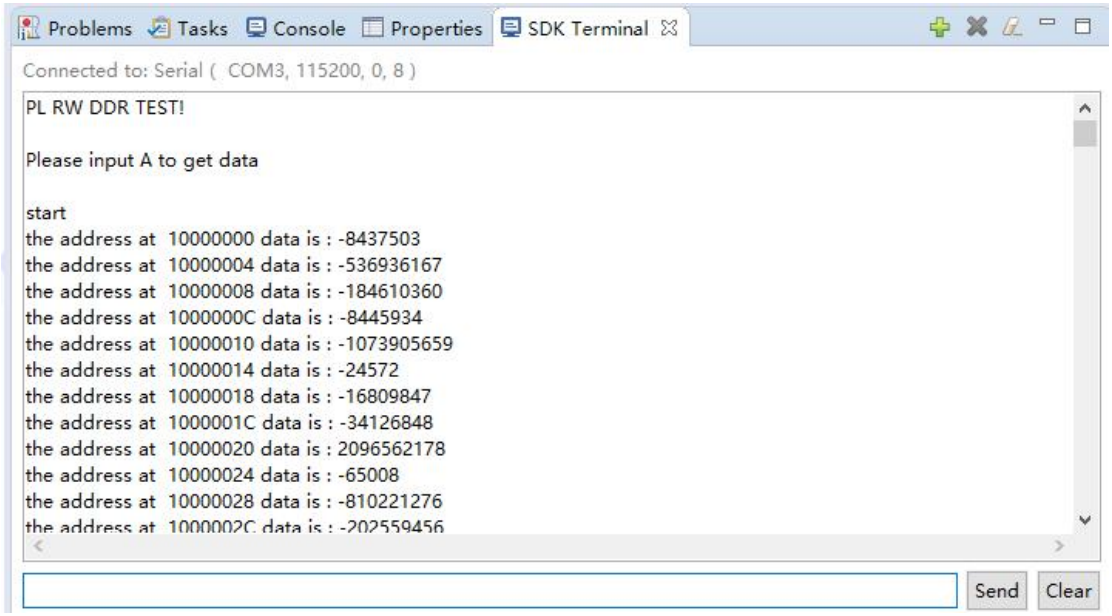


图 13.4.1 程序运行

可以看到串口打印出来的从 DDR 读取的数据，如下图所示，从图中可以看



到数据是随机的。

图 13.4.2 第一次读取数据

之后按下按键 PL_KEY1 触发 IP 核读写 DDR 数据，可以看到 PL_LED1 点亮，PL_LED2 未点亮，说明读写完成，未发现错误。如下图所示：

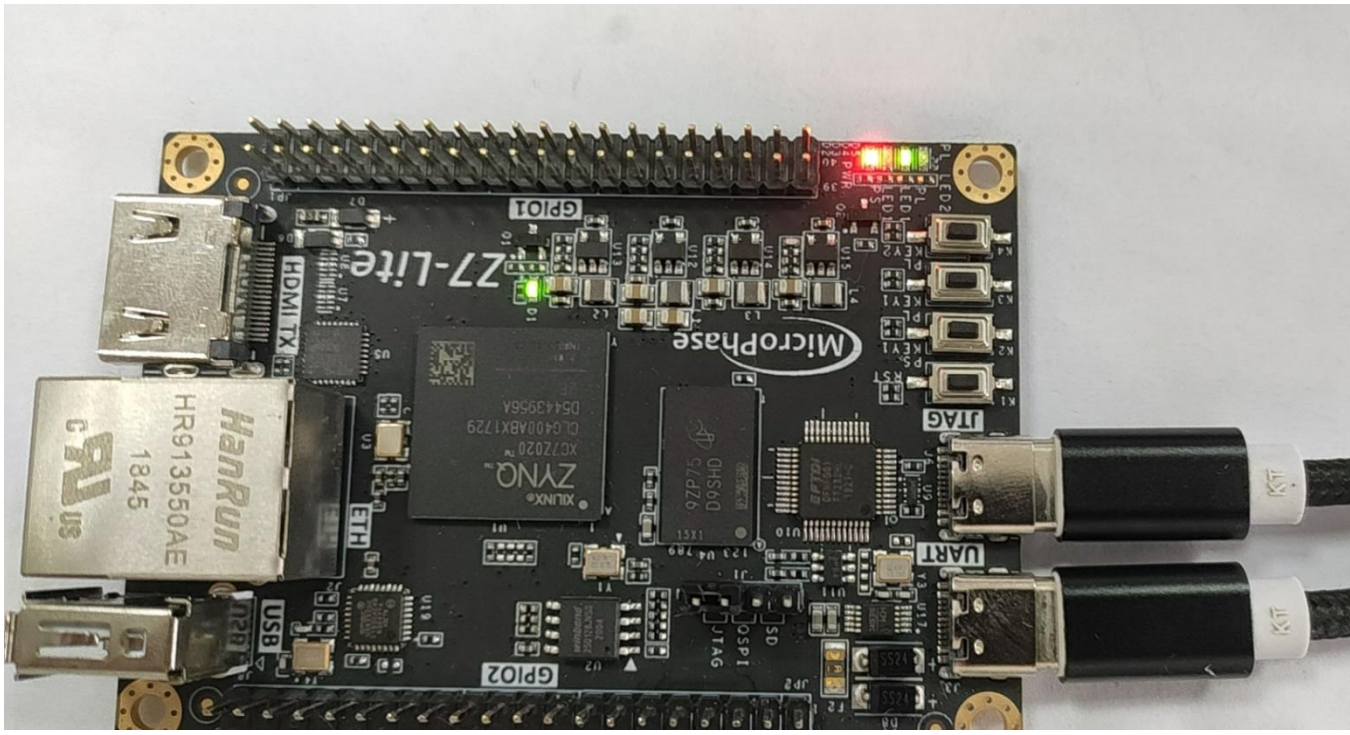
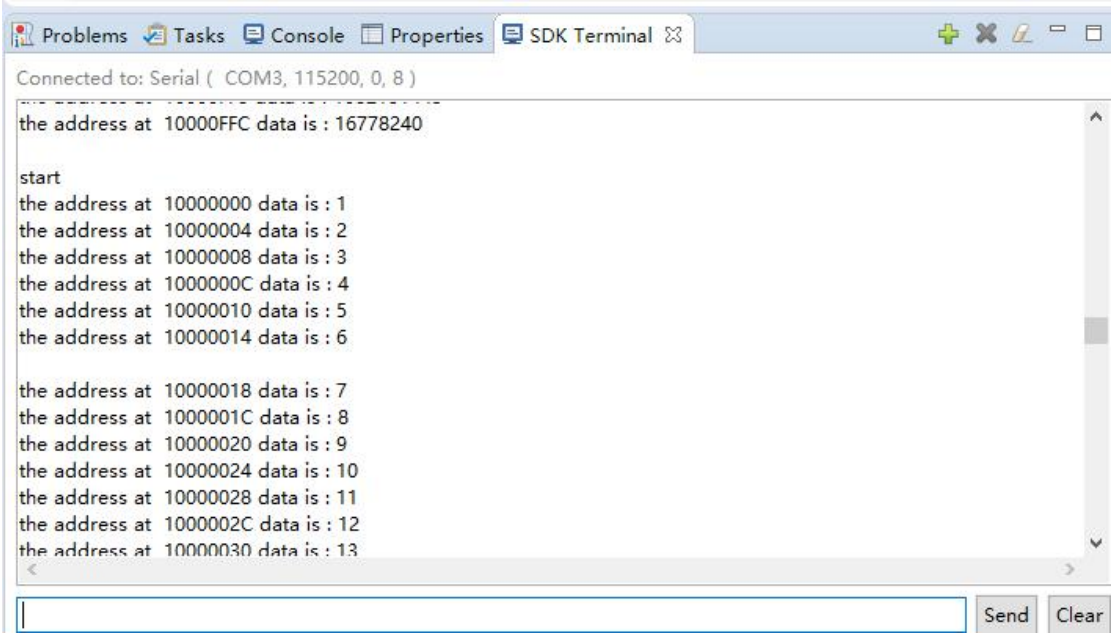


图 13.4.3 IP 核读写 DDR

之后再次串口发送“A”到开发板，可以看到打印出来的数据依次为 1、2、



3…1024，说明数据写入成功。

13.5 章节小结

在本实验中，我们通过 AXI4 总线实现了 PL 端读写 PS 端 DDR 数据的功能。读者可以自行尝试单独操作某一位的数据，或者修改自定义的 IP 核，改变写入的数据。通过这种方式熟练掌握相关的操作。

第 14 章　　Micro SD 卡读写 TXT 文本实验

在之前的实验中，我们介绍了将程序拷贝到 Micro SD 卡中，通过 Micro SD 卡启动系统的操作方式。除此之外，由于 Micro SD 卡具有容量大，操作简单，可插拔等的特点，经常作为大容量的存储介质用来保存数据。本实验我们来学习 Micro SD 卡的读写操作。

本实验包括以下内容：

14.1 实验简介

14.2 硬件设计

14.3 软件设计

14.4 下载验证

14.5 章节小结

14.1 实验简介

在本次实验中，我们开发板上使用的是 Micro SD 卡（也称 TF 卡），接下来我们简单介绍一下 SD 卡的相关知识。

SD 卡相关知识

SD 存储卡(Secure Digital Memory Card)是一种基于半导体快闪存储器的新一代高速存储设备。SD 存储卡的技术是从 MMC 卡(MultiMedia Card)格式上发展而来，在兼容 SD 存储卡基础上发展了 SDIO(SD Input/ Output)卡，此兼容性包括机械，电子，电力，信号和软件，通常将 SD、SDIO 卡俗称 SD 存储卡。

SD 卡具有高记忆容量、快速数据传输率、极大的移动灵活性以及很好的安全性，它被广泛地应用于便携式装置上，例如数码相机、平板电脑和多媒体播放器等。

SD 卡的结构能保证数字文件传送的安全性，也很容易重新格式化，所以有着广泛的应用领域。音乐、电影等多媒体文件都可以方便地保存到 SD 卡中。

SD 卡作为一种新型的存储设备，具有以下特点:

●高存储容量，最常用的容量：8GB、16GB、32GB、128GB、256GB 等。

●内置加密技术，适应基于 SDMI 协议的著作版权保护功能。

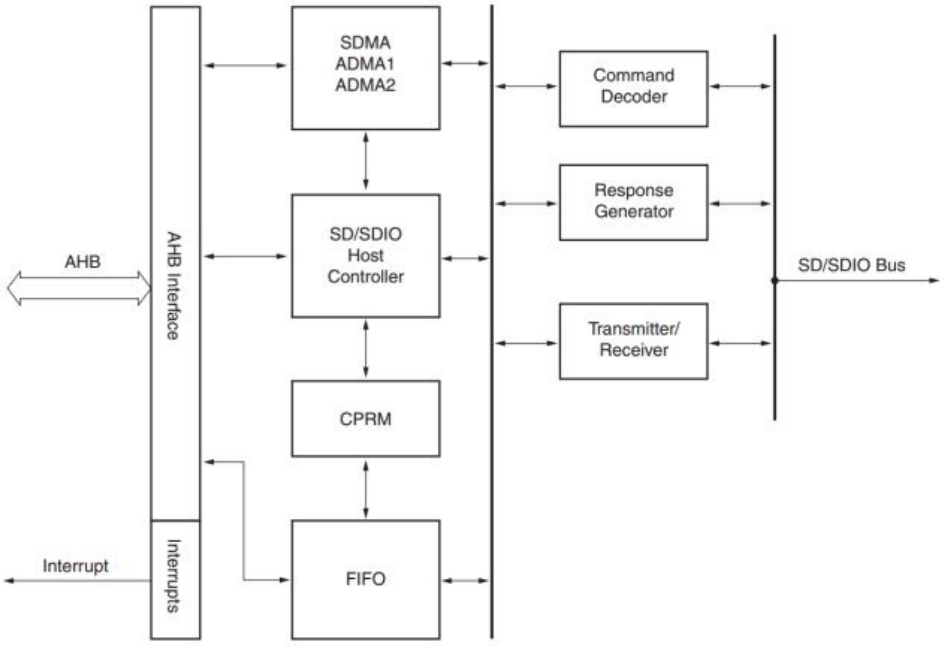
●高速数据传送；最大读写速率为 100MB/s。

●体积轻小，便于携带，具有很强的抗冲击能力。

SD 卡控制器（SD/SDIO Controller）

ZYNQ 中的 SD 卡控制器符合 SD2.0 协议规范， 接口兼容 eMMC、MMC3.31、 SDIO2.0、 SD2.0、 SPI，支持 SDHC、 SDHS 器件。 SD 卡控制器

支持 SDMA（单操作 DMA）、 ADMA1（4K 边界限制 DMA）和 ADMA2（在 32 位系统中允许任何位置和任意大小）。 ARM 处理器通过 AHB 总线访问 SD 卡



控制器， SD 控制器采用读和写通道各自双缓冲 FIFO 的机制提高吞吐带宽。其内部框图如下图所示：

图 14.1.1 SD 卡控制器内部框图

SD 控制器读写通道采用独立的 512 字节深度的双缓冲 FIFO 执行读和写操作。在写操作时，处理器向其中一个 FIFO 写数据，将另一个 FIFO 的数据写到 SD 总线；在读操作时，SD 总线上的数据向其中一个 FIFO 写数据，处理器将数据从另一个 FIFO 读出数据。SD 卡控制器通过双缓冲机制以保证最大带宽。

FATFS 文件系统

负责管理和存储文件信息的软件机构，在磁盘上组织文件的方法叫做文件系统。常用的文件系统有：（1）FAT/FATFS 适用于小型嵌入式系统；（2） NTFS 适用于 WINDOWS；（3）CDFS 适用于光盘；（4）exFAT 更适用于闪存。本次实验用到的 FATFS 文件系统具有以下优点：免费开源，专门为小型嵌入式系统设计，c 编写，支持 FAT12, FAT16 与 FAT32，支持多种存储媒介,有独立的缓冲区，可对多个文件进行读写，是一个可裁剪的文件系统（极为重要）。

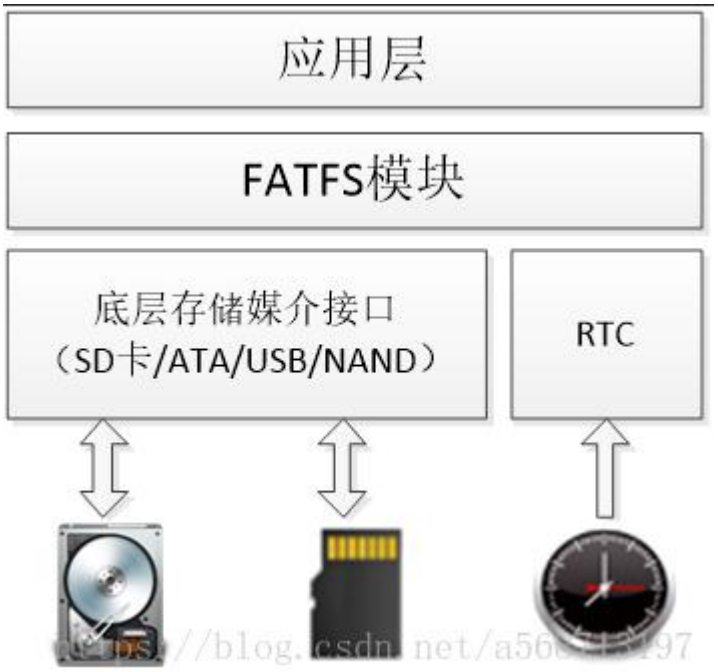
FATFS 具有以下特点：

Windows 兼容的 FAT 文件系统（支持 FAT12/FAT16/FAT32）

与平台无关，移植简单

代码量少、效率高

多种配置选项：支持多卷（物理驱动器或分区，最多 10 个卷）；多个 ANSI/OEM 代码页，包括 DBCS；支持长文件名、ANSI/OEM 或 Unicode；支持 RTOS；支持多种扇区大小；只读、最小化的 API 和 I/O 缓冲区。



FATFS 层次结构如下图所示：

图 14.1.2 FATFS 层次结构

①底层接口，包括存储媒介读 / 写接口（disk I/O）和供给文件创建修改时间的实时时钟，需要我们根据平台和存储介质编写移植代码。

②中间层 FATFS 模块，实现了 FAT 文件读 / 写协议。FATFS 模块提供的是 ff.c 和 ff.h。除非有必要，使用者一般不用修改，使用时将头文件直接包含进去即可。

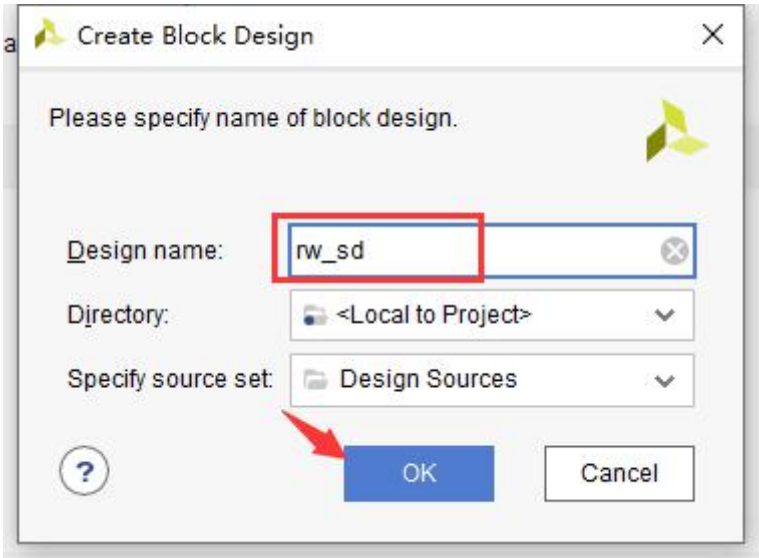
③最顶层是应用层，使用者无需理会 FATFS 的内部结构和复杂的 FAT 协议，只需要调用 FATFS 模块提供给用户的一系列应用接口函数，如 f_open，f_read，f_write 和 f_close 等，就可以像在 PC 上读 / 写文件那样简单。

FATFS 的整个系统包可以在 FATFS 的官网下载：http://elm-chan.org/fsw/ff/00index_e.html。

了解了相关知识之后，我们来进行硬件设计。

14.2 硬件设计

按照之前的步骤创建新的 Vivado 工程并命名为“rw_sd”，根据开发板型号选择相应的芯片型号。工程创建完成后创建新的 Block Design 并命名为“rw_sd”。



如下图所示：

图 14.2.1 创建 Block Design

创建完成后添加 ZYNQ7 Processing System IP 核，之后双击打开 IP 核设置页面，按照下面图片所示配置相关设置。取消 PS-PL Configuration 下 FCLK_RESET0_N 和 M AXI GP0 interface 的勾选；Perpheral I/O Pins 下配置 SD0 和 UART，具体分配引脚参考开发板原理图；取消 Clock Configuration 下 FCLK_CLK0 的勾选；DDR

Congifuration 下配置 DDR 控制器。

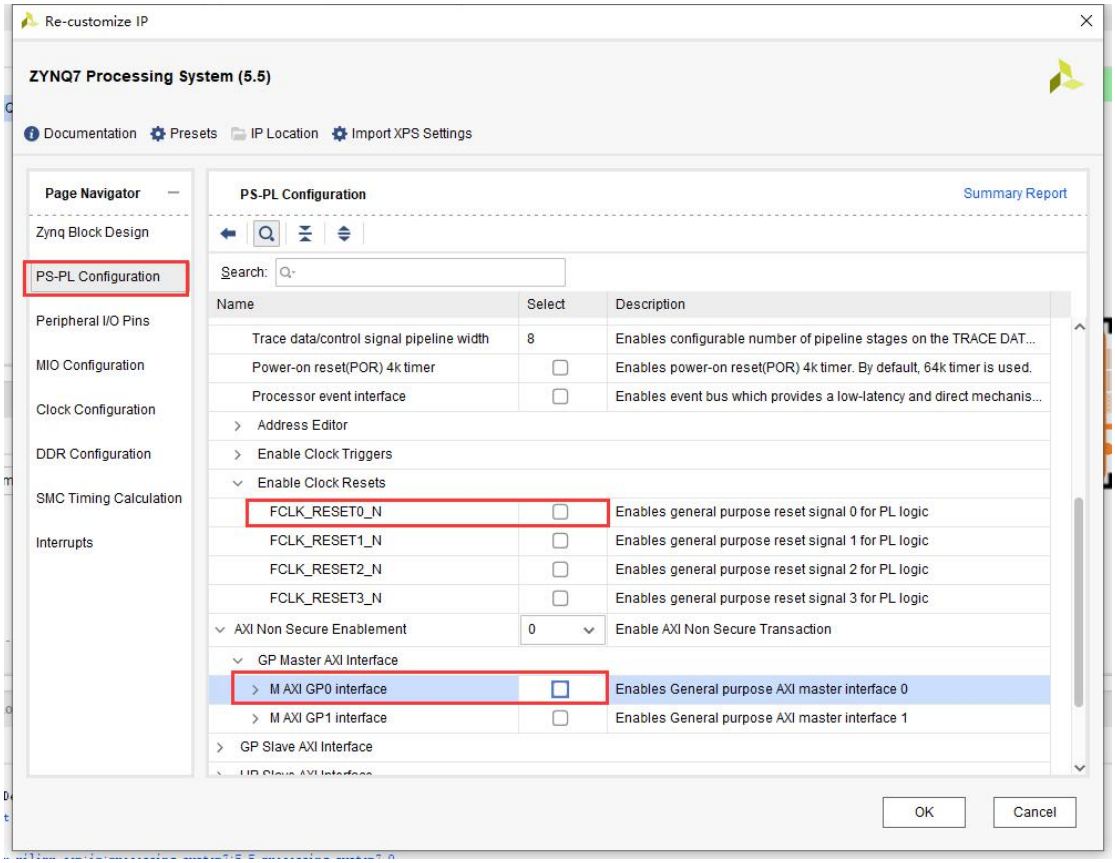


图 14.2.2 配置 PS-PL Configuration

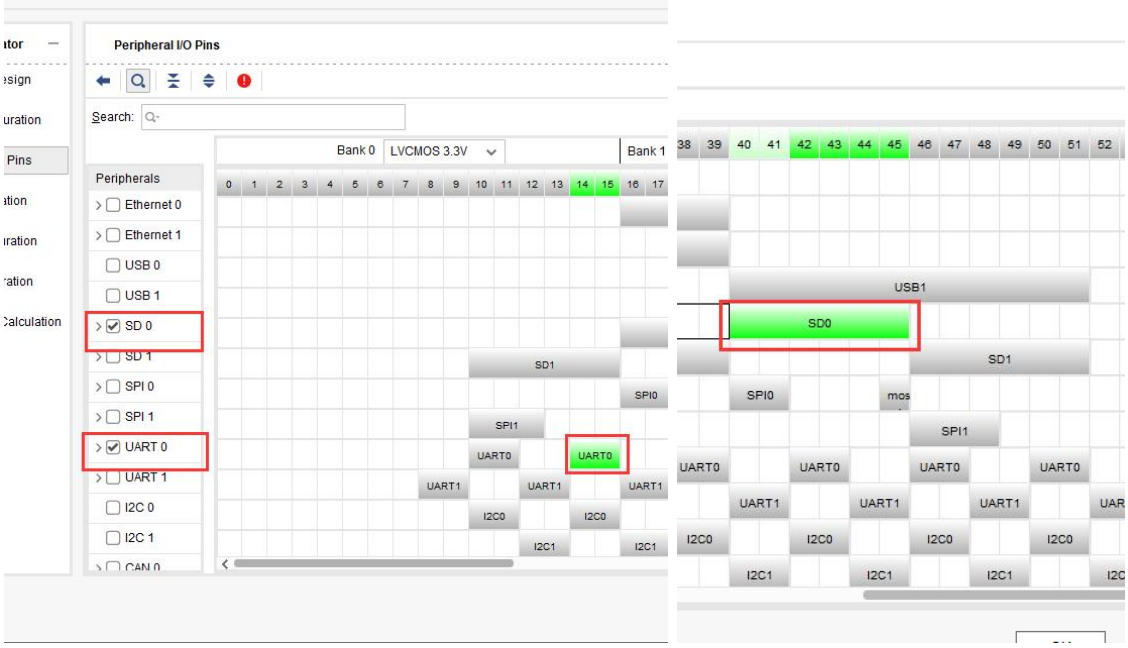


图 14.2.3 配置 Perpheral I/O Pins

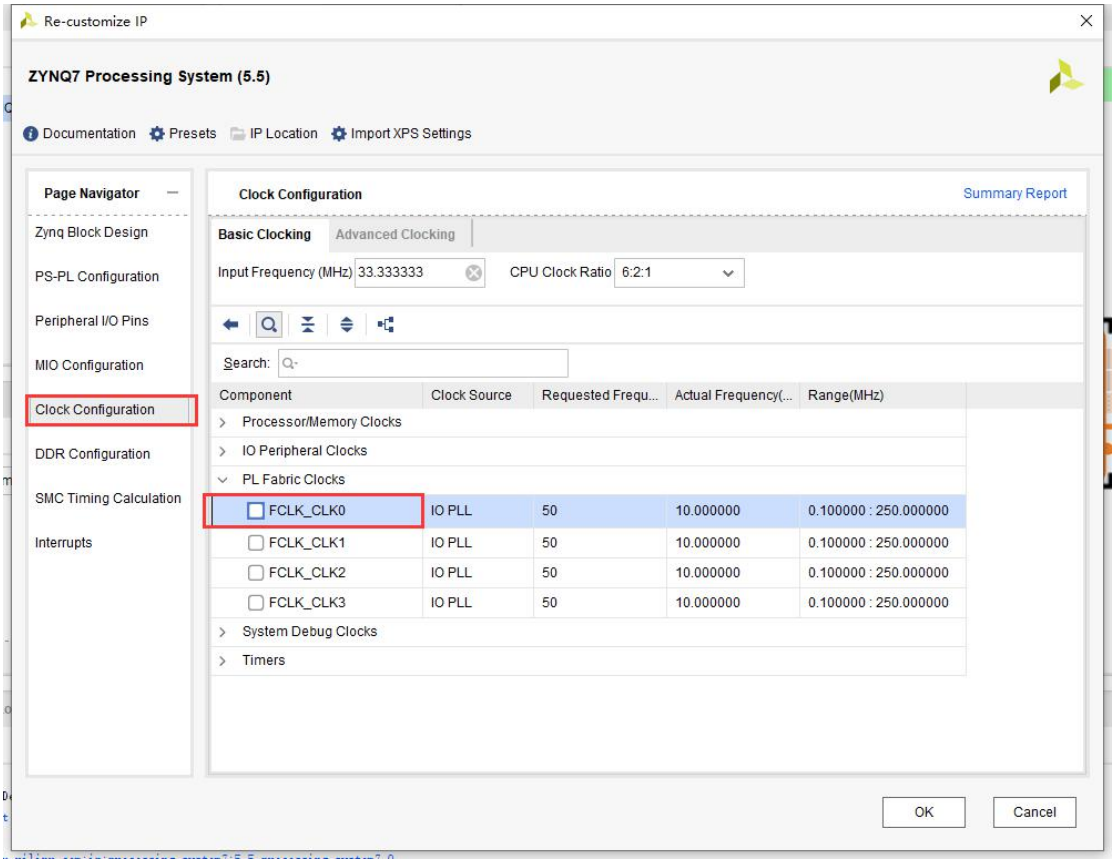


图 14.2.4 配置 Clock Configuration

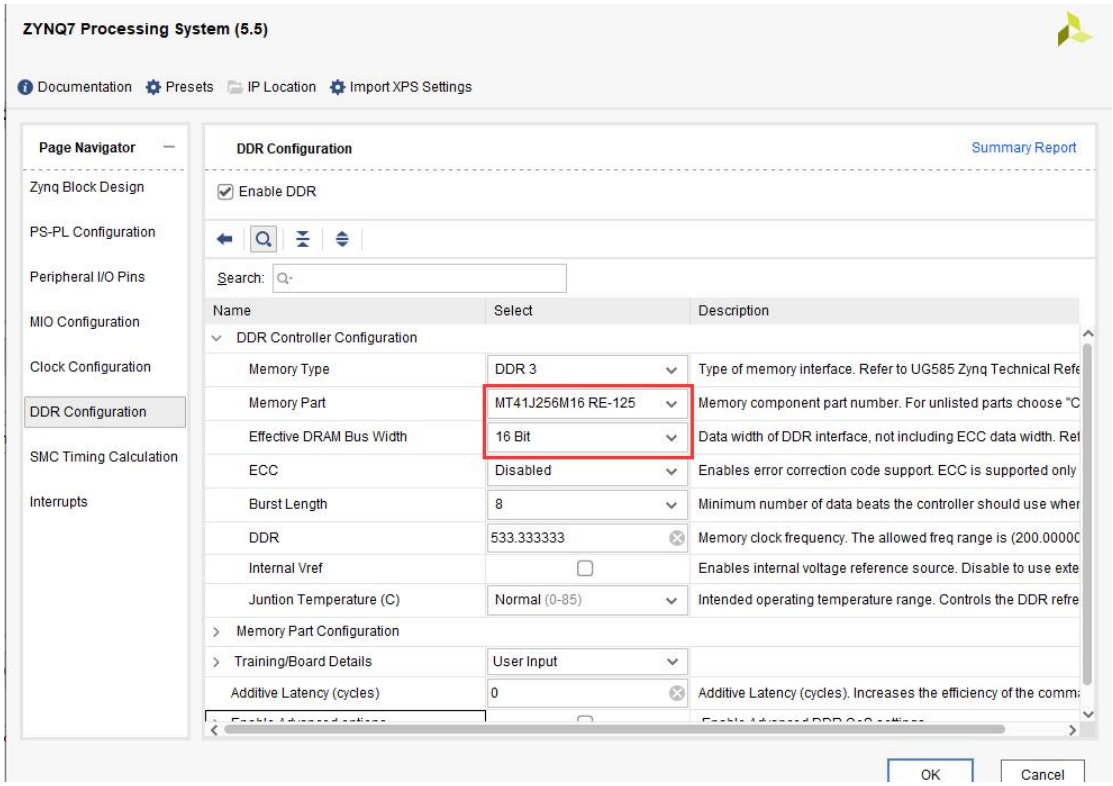
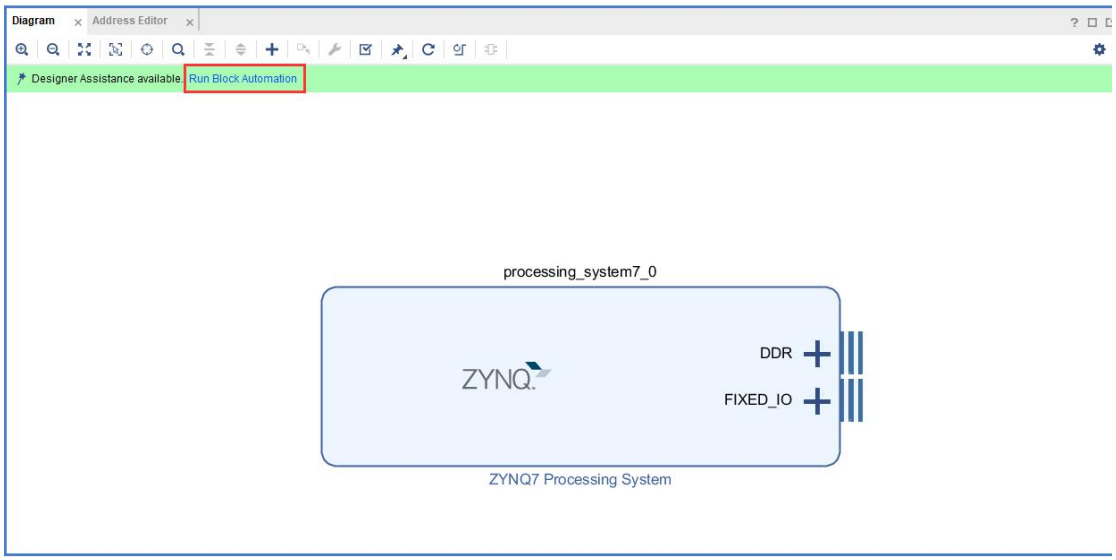
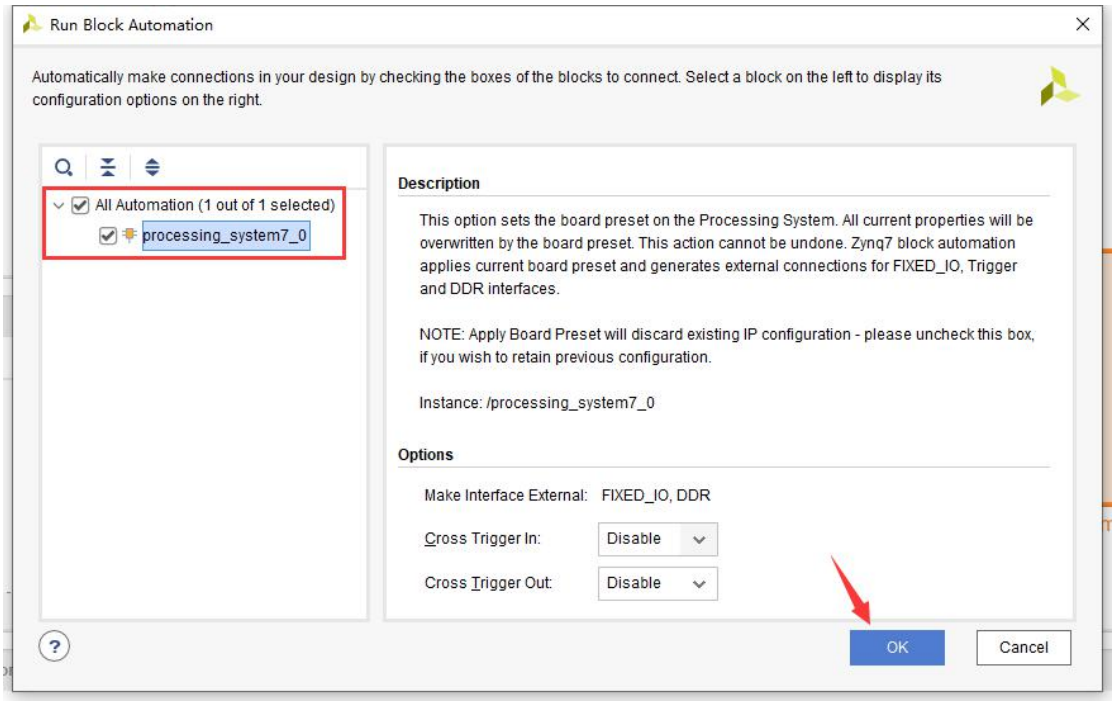


图 14.2.5 配置 DDR Congifuration



配置完成后点击“OK”，之后点击“Run Block Automation”，如下图所示。

图 14.2.6 点击 Run Block Automation



在弹出的页面全选所有端口，点击“OK”。

图 14.2.7 配置相关选项

之后保存并验证设计的正确性。检查无误后生成顶层 HDL。之后导出硬件描述文件并打开 SDK。

14.3 软件设计

在打开的 SDK 软件中创建一个新的工程并命名为“rw_sd”，工程模板选择空白模板。因为本实验需要用到 FATFS 文件系统，所以在板级支持包中添加文件系统的相关库函数。右键点击 rw_sd_bsp，在弹出的选项中选择“Board Support Package Setting”，如下图所示：

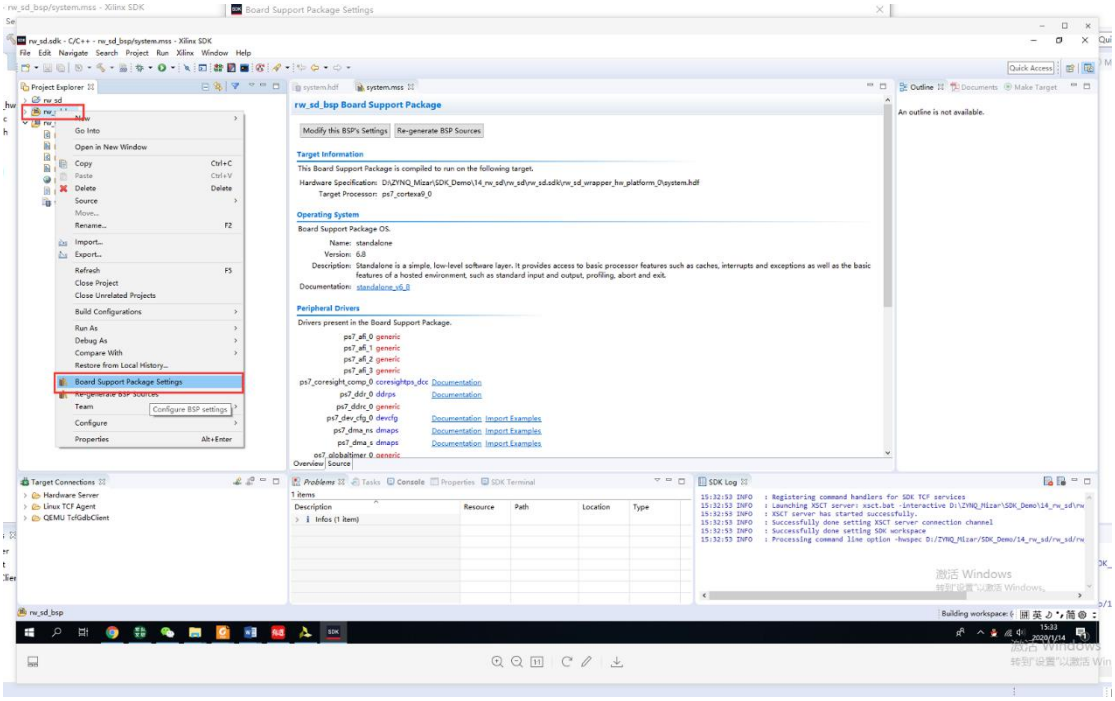
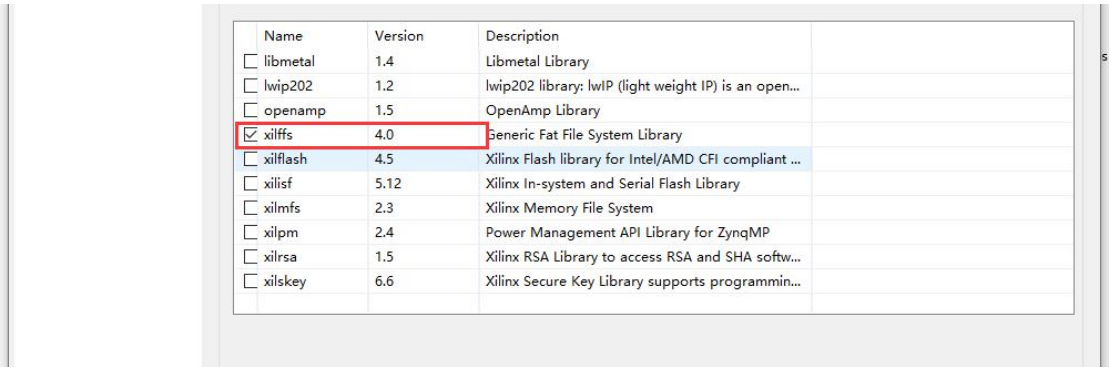


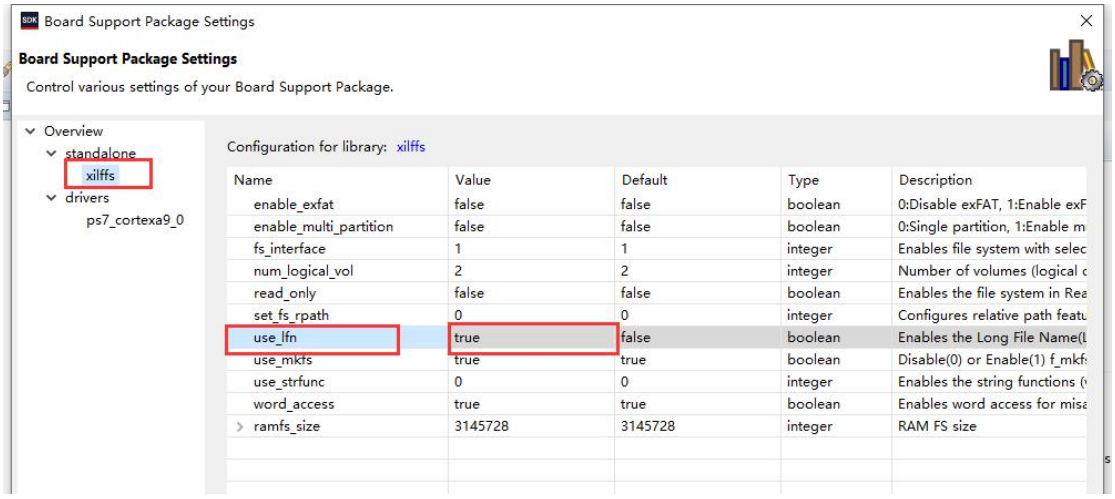
图 14.3.1 打开板级支持包设置



在弹出的界面中勾选“xilffs”， xilffs 即为 FATFS 库，如图 14.3.2 所示：

图 14.3.2 勾选 xilffs

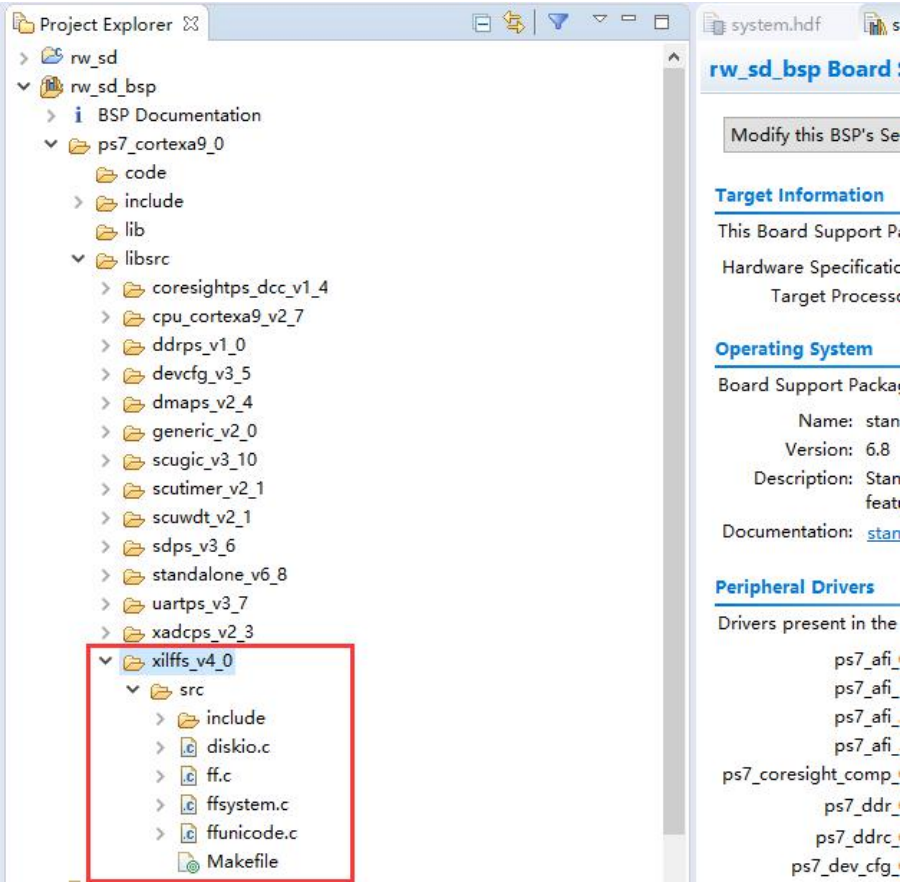
勾选后，会在左侧 Overview 的 standalone 一栏出现 xilffs，点击 xilffs。将 use_lfn 设置为 true，使能长文件名以及文件名的小写字母，点击“OK”按钮完



成设置。如图 14.3.3 所示：

图 14.3.3 使能 use_lfn

设置完成后，在 rw_sd_bsp→ps_cortexa9_0→libsrc 一栏下，会多出 FATFS



的库函数。如图 14.3.4 所示:

图 14.3.4 文件系统库函数

文件系统库函数添加完成后，右键点击 rw_sd->src 添加主程序源文件并命

```c
/***************************** Include Files *********************************/
#include "xparameters.h"    /* SDK generated parameters */
#include "xsdps.h"      /* SD device driver */
#include "xil_printf.h"
#include "ff.h"
#include "xil_cache.h"
#include "xplatform_info.h"

/************************** Function Prototypes ******************************/
int FfsSdPolledExample(void);

/**************************** Variable Definitions ***************************/
static FIL fil;     /* File object */
static FATFS fatfs;

static char FileName[32] = "Test.txt";
static char *SD_File;

char DestinationAddress[20] ;
const char SourceAddress[20]= "hello mizar !";

#define TEST 7

int main(void)
{
    int Status;

    xil_printf("SD Polled File System Example Test \r\n");

    Status = FfsSdPolledExample();
    if (Status != XST_SUCCESS) {
        xil_printf("SD Polled File System Example Test failed \r\n");
        return XST_FAILURE;
    }

    xil_printf("Successfully ran SD Polled File System Example Test \r\n");

    return XST_SUCCESS;
}
```

名为“main.c”，之后在编辑框中输入以下程序：

```c
int FfsSdPolledExample(void)
{
    FRESULT Res;
    UINT NumBytesRead;
    UINT NumBytesWritten;
    u32 BuffCnt;
    BYTE work[FF_MAX_SS];

     int FileSize = strlen(SourceAddress);

    TCHAR *Path = "0:/";

    //初始化文件系统
    Res = f_mount(&fatfs, Path, 0);

    if (Res != FR_OK) {
        return XST_FAILURE;
    }

    //格式化SD卡
    Res = f_mkfs(Path, FM_FAT32, 0, work, sizeof work);
    if (Res != FR_OK) {
        return XST_FAILURE;
    }

    //打开一个文件，如果文件不存在，则创建一个文件，该文件的权限为可读写
    SD_File = (char *)FileName;

    Res = f_open(&fil, SD_File, FA_CREATE_ALWAYS | FA_WRITE | FA_READ);
    if (Res) {
        return XST_FAILURE;
    }
```

```
    //关闭文件
    Res = f_close(&fil);
    if (Res) {
        return XST_FAILURE;
    }
    return XST_SUCCESS;
}
```

```
    //指针指向文件开头
    Res = f_lseek(&fil, 0);
    if (Res) {
        return XST_FAILURE;
    }

    //向文件中写入数据
    Res = f_write(&fil, (const void*)SourceAddress, FileSize,
            &NumBytesWritten);
    if (Res) {
        return XST_FAILURE;
    }

    //指针指向文件开头
    Res = f_lseek(&fil, 0);
    if (Res) {
        return XST_FAILURE;
    }

    //从SD卡中的文件读出数据
    Res = f_read(&fil, (void*)DestinationAddress, FileSize,
            &NumBytesRead);
    if (Res) {
        return XST_FAILURE;
    }

    //比较写入的数据与读出的数据是否相同
```

输入完成后快捷键 Ctrl + S 保存并编译程序。该程序是官方例程修改后得到的，读者可以在以下路径找到官方例程，里面的注释非常详尽，可仔细阅读。

C:\Xilinx\SDK\2018.3\data\embeddedsw\lib\sw_services\xilffs_v4_0\example

14.4 下载验证

首先我们将使用 type-c 线与 JTAG 接口连接，另外一端与电脑连接。然后使用 type-c 线将开发板的 UART 接口与电脑连接，用于串口通信。最后连接开发板的电源，将 SD 卡插入开发板 SD 卡插座，SD 卡槽位于开发板背面，如下图所示。

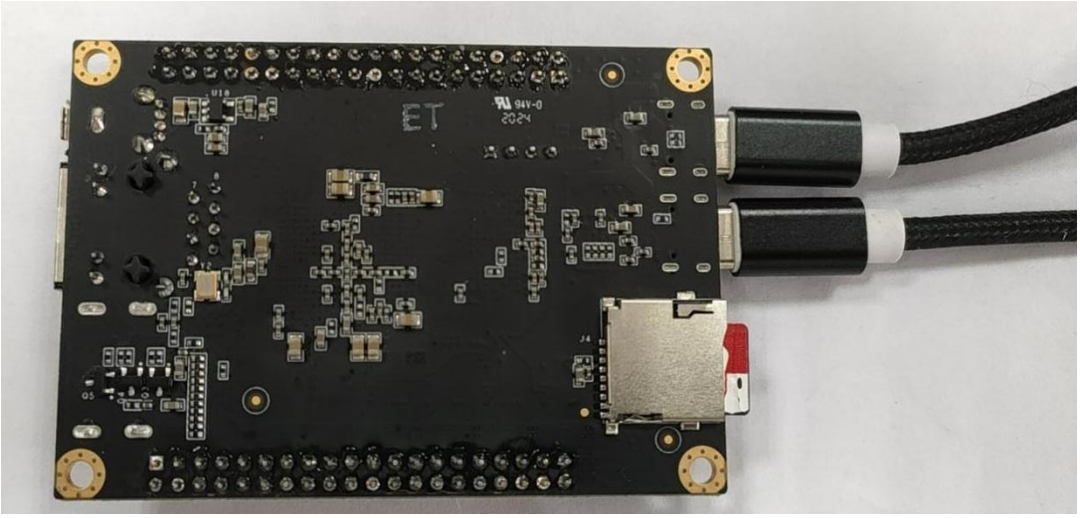
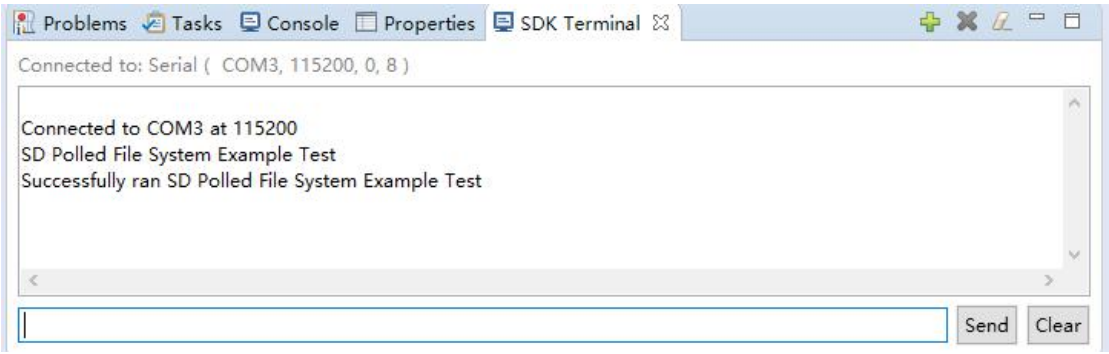


图 14.4.1 开发板 SD 卡接口

在 SDK 软件下方的 SDK Terminal 窗口中点击右上角的加号设置并连接串口。然后在应用工程 rw_sd 上右击，选择“Run As”，然后选择第一项“1 Launch on Hardware (System Debugger)”即可烧录程序。

烧录完成后可以看到 SDK Terminal 窗口打印出“SD Polled File System Example Test”的信息，一段时间后打印出“Successfully ran SD Polled File System Example



Test”的信息，如下图所示，表明读写测试成功。

图 14.4.2 串口打印信息

接下来从开发板的卡槽中取出 SD 卡，通过连接读卡器来插入电脑的 USB

接口，此时可以看到 SD 卡中出现一个 Test.txt 文件，打开的文件内容如下图所示。

图 14.4.3 Test.txt 文件内容

文件内容与程序中定义的一致，说明本次实验验证成功。

14.5 章节小结

在本次实验中，我们简单了解了 Micro SD 卡的读写操作方法，在后续实验中，我们还将利用这些操作来实现更多的功能。

第 15 章 BRAM 应用实验

在之前的实验中，我们介绍了一些用于 PS 和 PL 端数据交互的方法，不同的方法具有各自的特点，适用于不同的场合。在本次实验中，我们讲解一种适用于数据量较少、地址不连续、长度不规则的情况下的数据交互。那就是通过 BRAM 来实现 PS 和 PL 的数据交互。

本章包括以下几部分内容：

15.1 实验简介

15.2 硬件设计

15.3 软件设计

15.4 下载验证

15.5 章节小结

15.1 实验简介

Block RAM 是 PL 部分的存储器阵列，为了与 DRAM（分布式 RAM）区分开，称之为块 RAM（Block RAM）。ZYNQ 的每一个 BRAM 36KB，7020 的 BRAM 有 140 个（4.9M），7030 有 265 个（9.3M），7045 有 545 个（19.2M）。每一个 BRAM 都有两个共享数据的独立端口，当然是可以配置的，可用于片内数据缓存、FIFO 缓冲。

在使用时，PS 并不直接读写 BRAM 的数据。PS 和 BRAM 之间的事务由 AXI BRAM Controller IP 核来控制，AXI BRAM Controller IP 核可以是 AXI 总线形式，也可以是 AXILite 总线形式。AXI4 接口模式的 BRAM 控制器支持的数据位宽为 32 位、 64 位、 128 位、 512 位和 1024 位，而 AXI4-Lite 接口仅支持 32 位数据位宽。

15.2 硬件设计

按照之前的步骤创建一个新的 Vivado 工程并命名为“BRAM_test”，根据开发板型号选取合适的芯片型号。工程创建完成后创建新的 Block Design 并命名为“BRAM_test”。

创建完成后添加 ZYNQ7 Processing System 控制系统 IP。双击打开配置页面，

参照之前的实验配置 DDR 控制器和 UART，配置完成后点击“OK”。如下图所示：

图 15.2.1 ZYNQ7 Processing System

之后点击 IP 核添加按钮，再弹出的 IP 核列表的搜索框中输入“axi bram”搜索添加 AXI BRAM Controller IP 核，该 IP 核用来控制对于 BRAM 的读写。此处添加两个该 IP 核，分别控制读取和写入两种操作。

图 15.2.2 添加 AXI BRAM Controller IP 核

添加完成后双击 IP 和进入配置界面，按照下图所示配置相关参数。

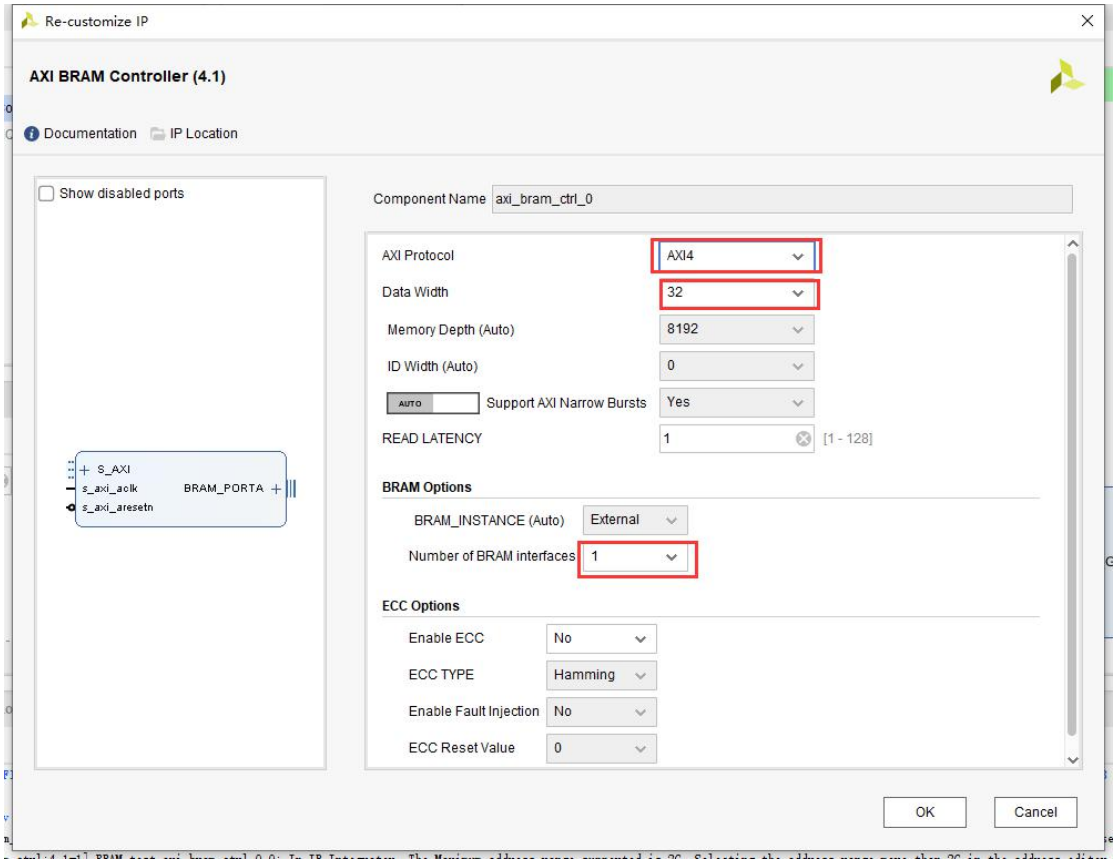


图 15.2.3 配置相关参数

配置完成后添加 BRAM 的 IP 核，点击添加按钮，在搜索框中输入“Block Memory Generator”，之后双击 Block Memory Generator 添加此 IP 核。

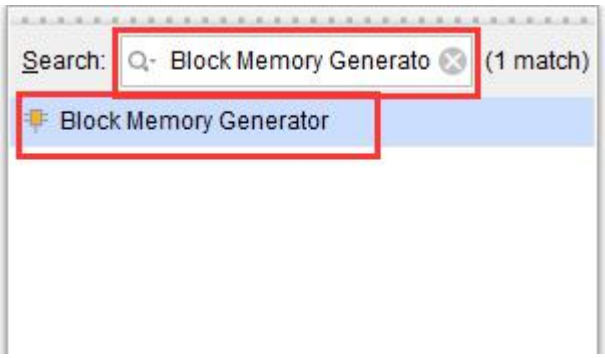


图 15.2.4 添加 Block Memory Generator IP 核

添加完成后双击此 IP 核进入配置界面，按照下图所示分别在 Basic 和 Other Options 标签页配置相关参数。配置完成后点击“OK”。

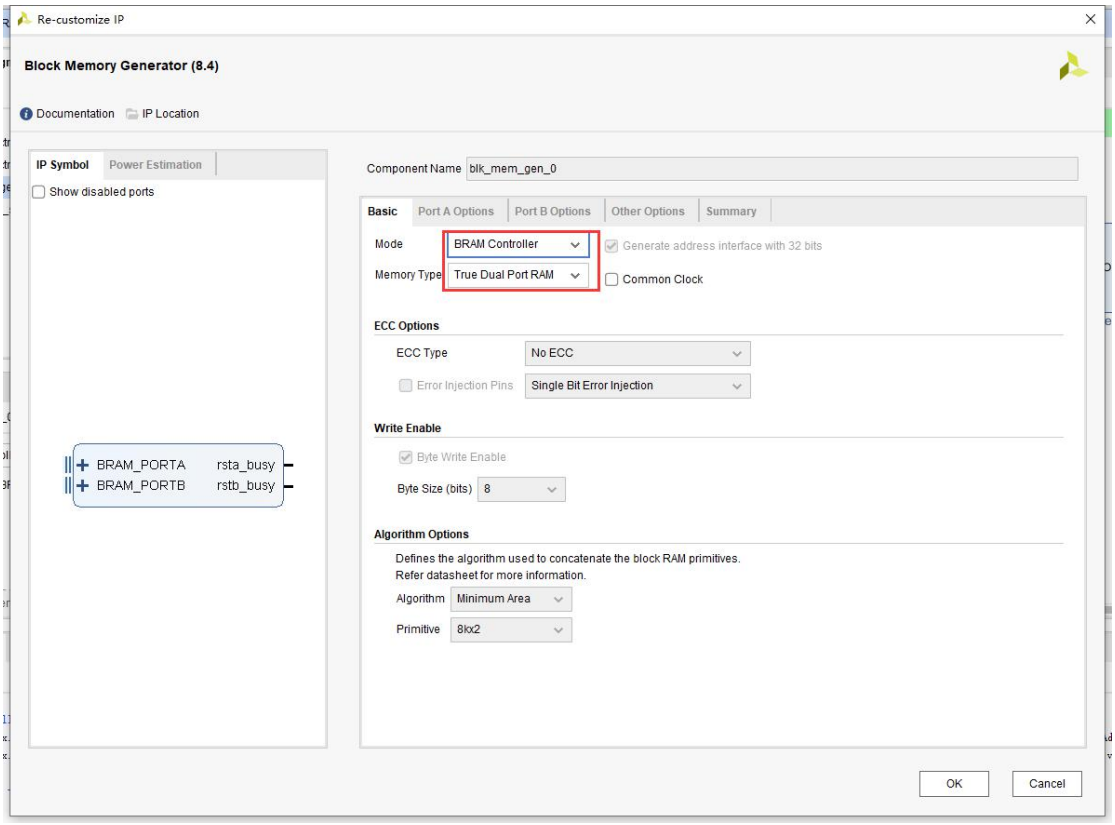


图 15.2.5 配置 Basic

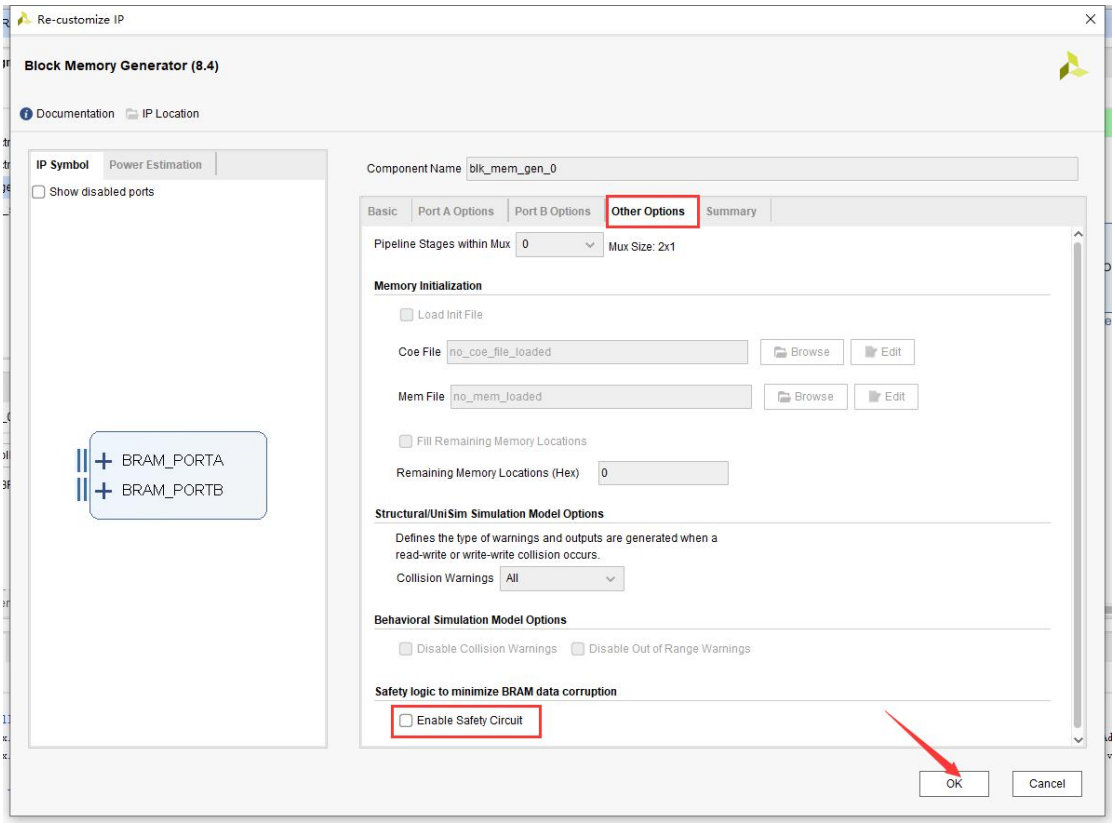


图 15.2.6 配置 Other Options

之后依次点击“Run Block Automation”和“Run Connection Automation”，如下图所示：

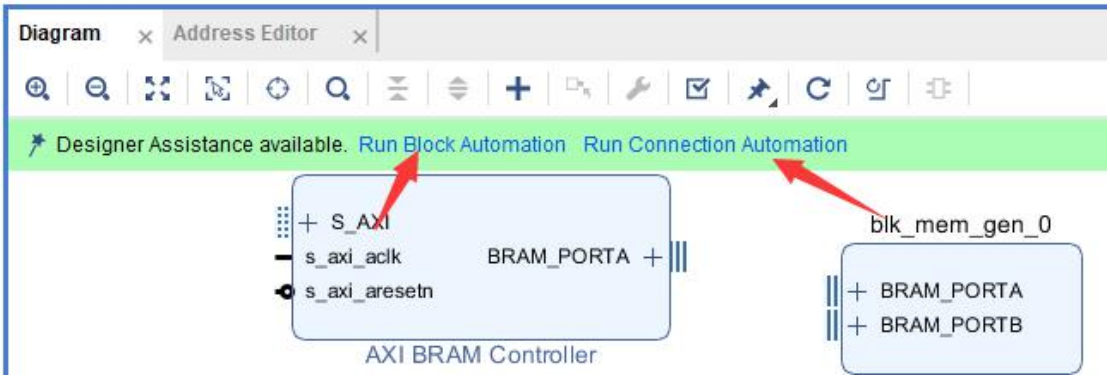


图 15.2.7 自动连线

在弹出的对话框左侧确认勾选 processing_system7_0，然后点击“OK”。这一步骤会将 ZYNQ7 PS 模块的 DDR 和 FIXED_IO 端口引出。

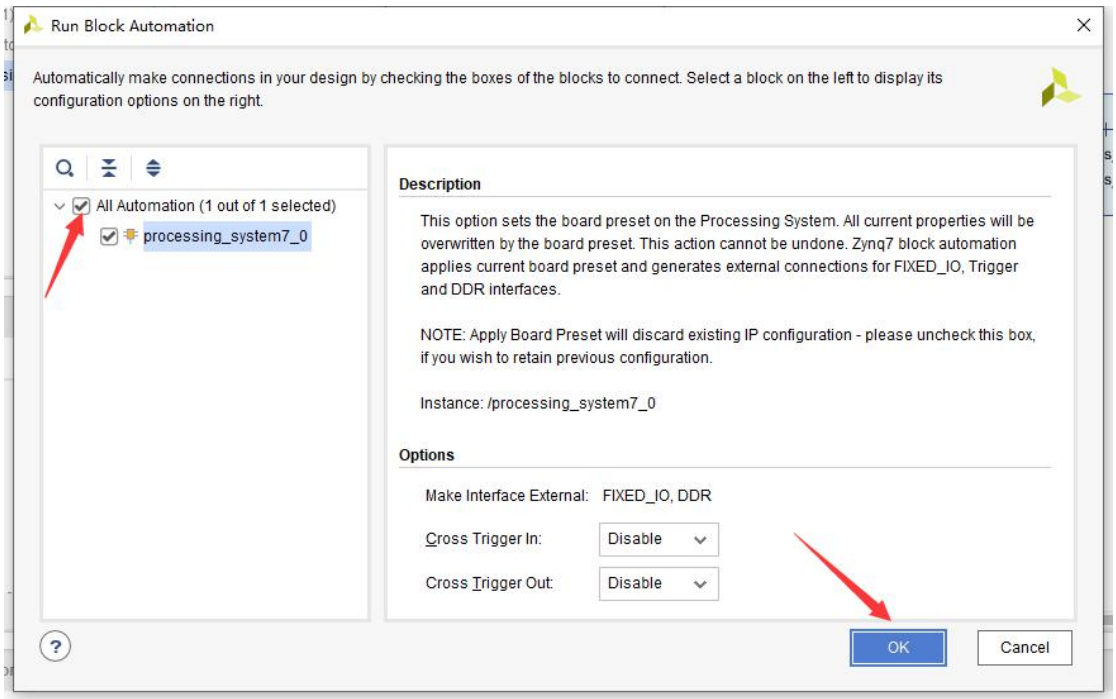


图 15.2.8 引出端口

在弹出的对话框左侧全选连接所有端口。

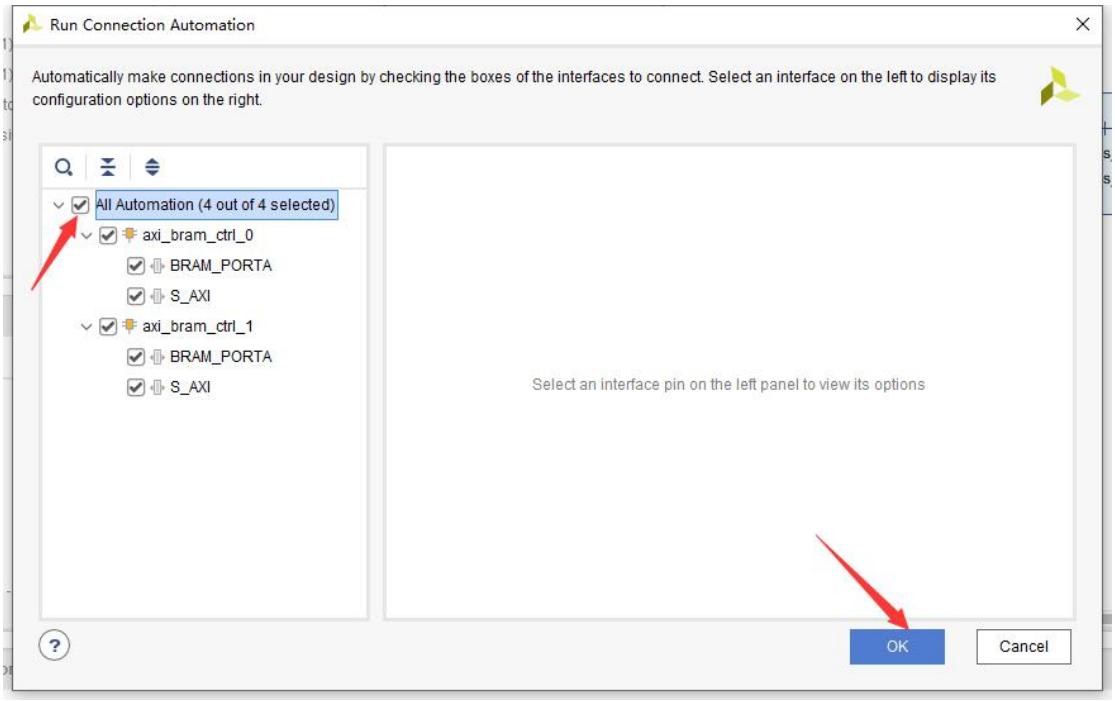
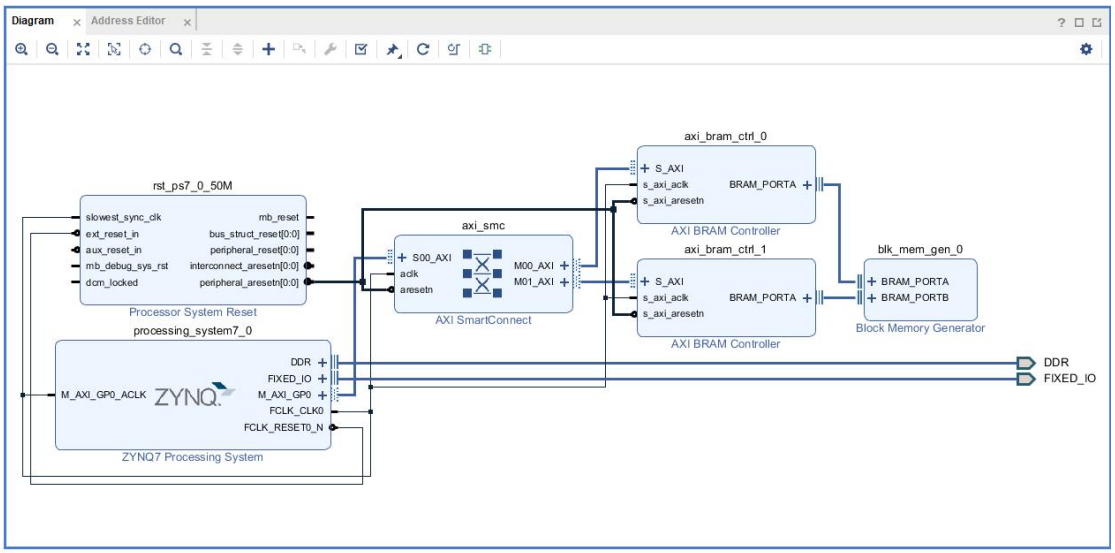


图 15.2.9 连接所有端口

连接完成后点击 Regenerate Layout 重新布局，布局完成后的设计界面如下图



所示：

图 15.2.10 完成后的设计界面

点击切换到 Address Editor 可以看到两个 BRAM 控制器的地址和空间大小设

置，此处保持默认，在实际项目中根据项目要求设置合适的大小。

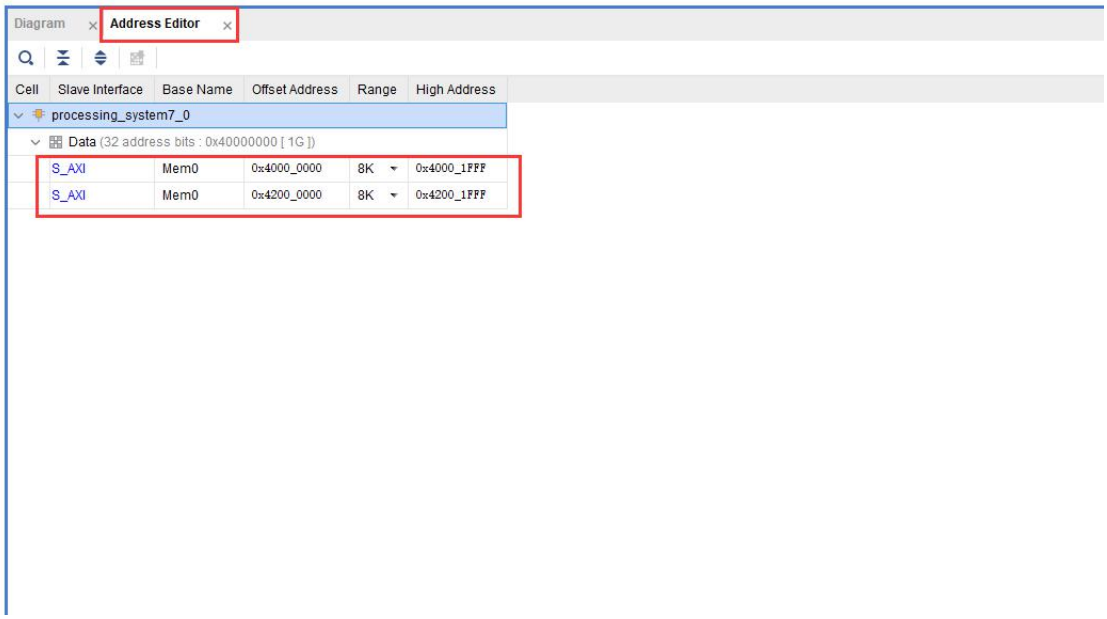


图 15.2.11 地址及空间设置

到这里我们的 Block Design 就设计完成了，在 Diagram 窗口空白处右击，然后选择“Validate Design”验证设计。验证完成后弹出对话框提示“Validation Successful”表明设计无误， 点击“OK”确认。最后按快捷键“Ctrl + S”保存设计。

接下来在 Source 窗口中右键点击 Block Design 设计文件“BRAM_test.bd”，然后依次执行“Generate OutputProducts”和“Create HDL Wrapper”生成顶层 HDL。

最后在左侧 Flow Navigator 导航栏中找到 PROGRAM AND DEBUG，点击该选项中的“GenerateBitstream”，对设计进行综合、实现、并生成 Bitstream 文件。

在生成 Bitstream 之后， 在菜单栏中选择 File > Export > Export hardware 导出硬件，并在弹出的对话框中，勾选“Include bitstream”。然后在菜单栏选择 File > Launch SDK，启动 SDK 软件。

15.3 软件设计

在打开的 SDK 软件中，创建一个新的工程并命名为“BRAM_test”，工程模板选择空白。创建完成后右键点击 BRAM_test->src 为工程添加主函数源文件，命

```
system.hdf   system.mss   *main.c   xparameters.h
1  #include <stdio.h>
2  #include "xil_io.h"
3  #include "xparameters.h"
4  #include "xbram.h"
5
6  #define BRAM_CTRL_0_BASE      XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR
7  #define BRAM_CTRL_1_BASE      XPAR_AXI_BRAM_CTRL_1_S_AXI_BASEADDR
8  char ch_date[]="Hello Mizar !";
9  int main()
10 {
11         int num,len;
12         int rev;
13
14     xil_printf("------The test is start...-------\r\n");
15     len=strlen(ch_date);//计算数据长度
16     //通过控制axi_bram_ctrl_0，向blk_mem_gen_0写数据
17     for( num=0; num<len; num++ )
18     {
19         XBram_WriteReg(BRAM_CTRL_0_BASE , num*4, ch_date[num]);
20     }
```

名为“main.c”。之后在打开的编辑窗口中输入以下程序：

```
21
22     //通过控制axi_bram_ctrl_1，把blk_mem_gen_0里的数据读出来
23    for( num=0; num<len; num++ )
24        {
25                rev = XBram_ReadReg( BRAM_CTRL_1_BASE, num*4);
26                xil_printf( "The data at %x is %c \r\n",XPAR_AXI_BRAM_CTRL_1_S_AXI_BASEADDR + num*4,rev);
27        }
28     xil_printf("------The test is end!------\r\n");
29     return 0;
30 }
```

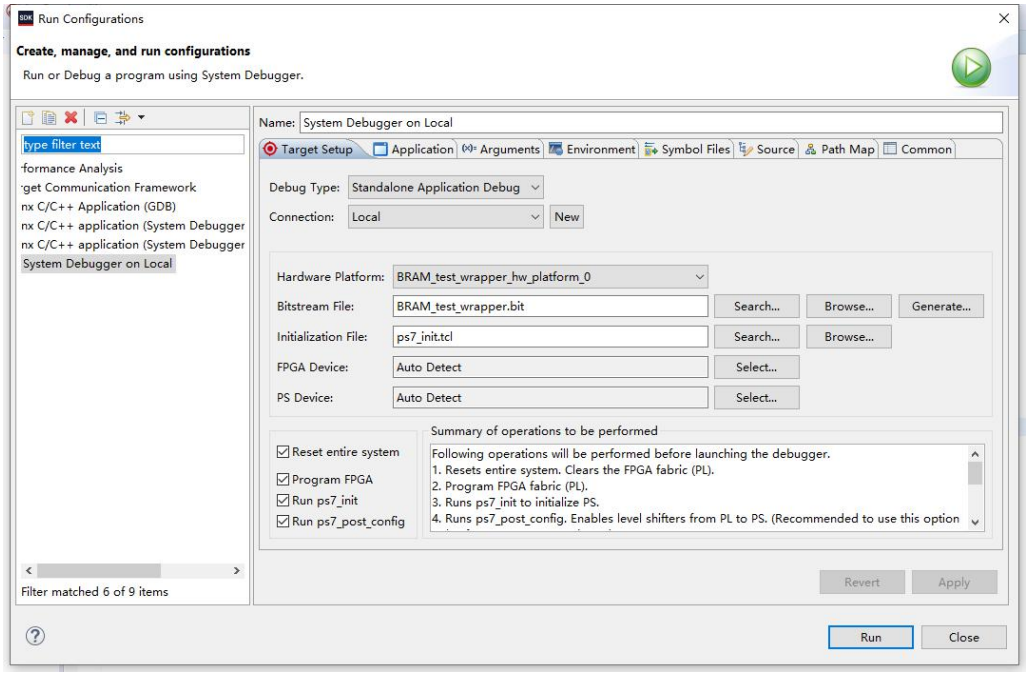
该程序实现了向 BRAM 中写入一串字符串并读出来的功能。

程序设计完成后，按快捷键 Ctrl+S 保存 main.c 文件，工具会自动进行编译。编译完成后控制台（Console）中会出现提示信息“Build Finished”，同时在应用工程的 Binaries 目录下可以看到生成的 elf 文件。

15.4 下载验证

首先我们将使用 Micro USB 线与 JTAG 接口连接，另外一端与电脑连接。然后使用 Micro USB 连接线将开发板的 UART 接口与电脑连接，用于串口通信。最后连接开发板的电源。

在下方 SDK Terminal 窗口连接串口。之后进入 Run Configurations，按照如下



图所示配置相关设置。

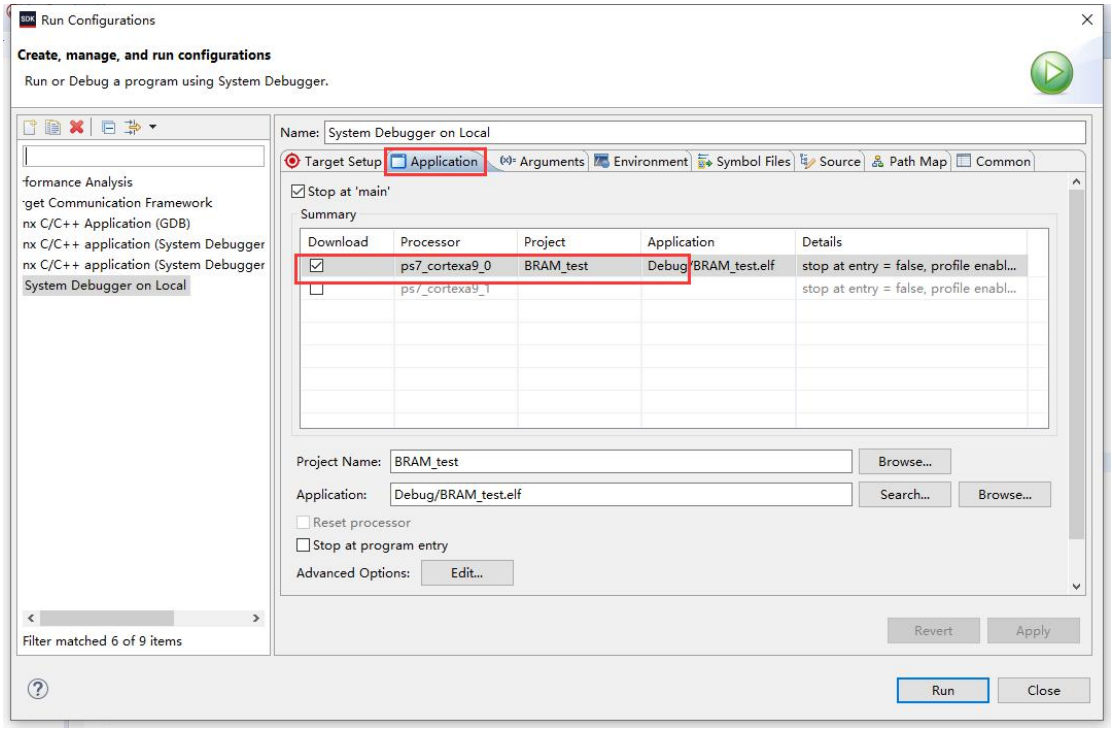


图 15.4.1 Run Configurations 配置

配置完成后点击“Run”烧录程序，烧录完成后可以在下方 SDK Terminal 窗口看到如下图所示的打印信息，说明读写测试成功。

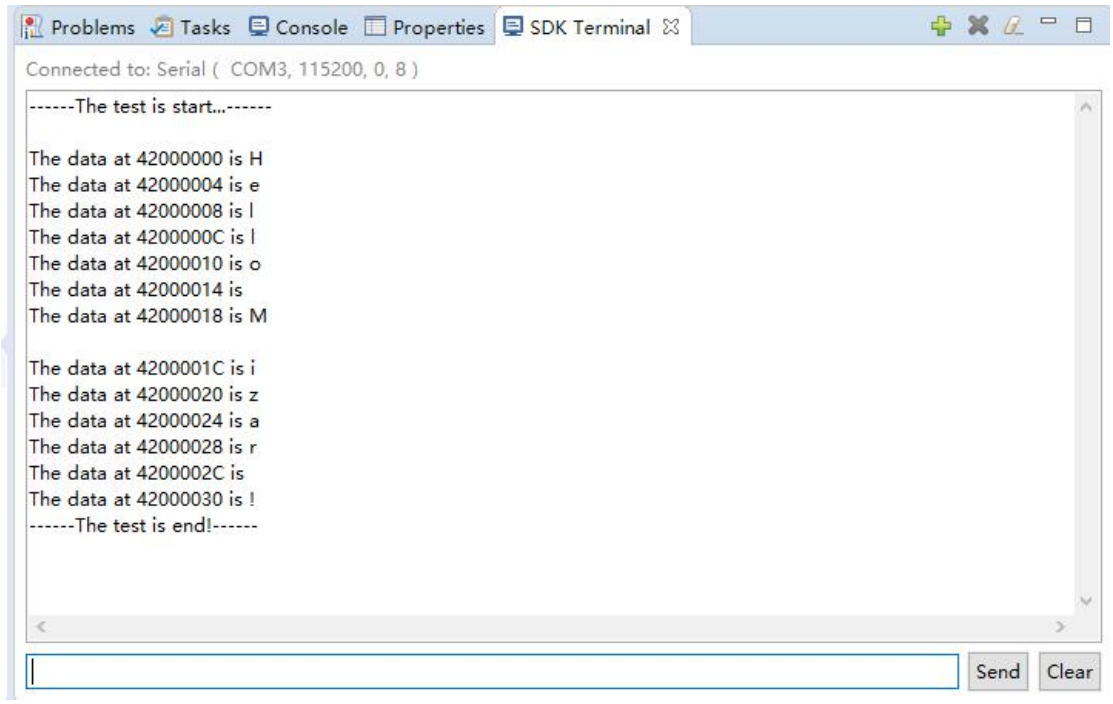


图 15.4.2 运行结果

15.5 章节小结

在本实验中，我们学习了一种新的 PS 和 PL 端数据交互的方式。本实验仅实现了基本读写的功能使读者对于 BRAM 有一个基本的了解，希望读者在此基础上尝试实现将串口接收的数据写入 BRAM，BRAM 的中断功能等拓展实验，熟练掌握 BRAM 的使用。

第 16 章　　DMA 环路实验

DMA(Direct Memory Access，直接内存存取) 是所有现代电脑的重要特色，它允许不同速度的硬件装置来沟通，而不需要依赖于 CPU 的大量中断负载。否则，CPU 需要从来源把每一片段的资料复制到暂存器，然后把它们再次写回到新的地方。在这个时间中，CPU 对于其他的工作来说就无法使用。

DMA 的使用极大提高了数据传输的速度，同时很少占用 CPU 的资源。当前在许多传输大量数据的使用场合中基本都会使用 DMA。本实验我们就来学习 DMA 的使用方法。

本章包括以下几部分内容：

16.1 实验简介

16.2 硬件设计

16.3 软件设计

16.4 下载验证

16.5 章节小结

16.1 实验简介

DMA 传输将数据从一个地址空间复制到另外一个地址空间。当 CPU 初始化这个传输动作，传输动作本身是由 DMA 控制器来实行和完成。典型的例子就是移动一个外部内存的区块到芯片内部更快的内存区。像是这样的操作并没有让处理器工作拖延，反而可以被重新排程去处理其他的工作。DMA 传输对于高效能嵌入式系统算法和网络是很重要的。

在实现 DMA 传输时，是由 DMA 控制器直接掌管总线，因此，存在着一个总线控制权转移问题。即 DMA 传输前，CPU 要把总线控制权交给 DMA 控制器，而在结束 DMA 传输后，DMA 控制器应立即把总线控制权再交回给 CPU。一个完整的 DMA 传输过程必须经过 DMA 请求、DMA 响应、DMA 传输、DMA 结束 4 个步骤。

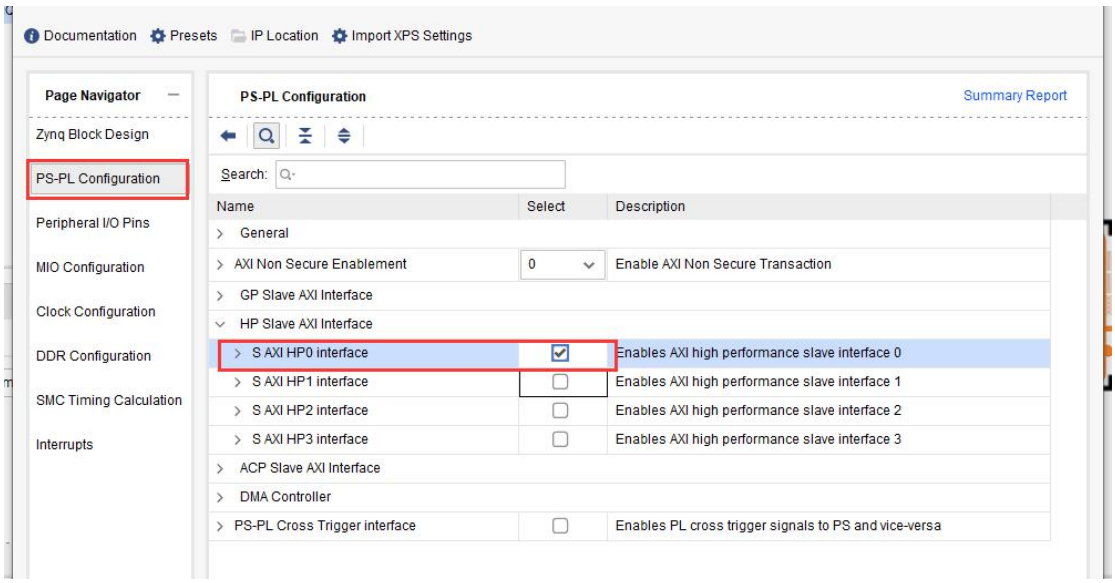
ZYNQ 提供了两种 DMA 的实现方式，一种是使用集成在 PS 端的 DMA 外设，另一种是在 PL 端添加 AXI DMA IP 核。PS 端 DMA 的使用通过 DMA 控制器 DAMC 来实现，需要 CPU 的参与。DAMC 最多支持 8 个通道，可以在一个 DAMC 上同时挂载 8 个具备 DMA 结构的外设。DAMC 与 PL 的连接是通过 AXI_GP 接口，这个接口最高支持到 32 位宽度，理论最高速率为 600MB/s。PL 端 DMA 的实现不需要 CPU 的参与，CPU 只需发出控制信号和相关参数即可。

本次实验中，我们使用 AXI DMA IP 核，通过该 IP 核读取 DDR 的数据，之后再通过 IP 核向 DDR 写入数据，实现 DMA 的环通测试。

16.2 硬件设计

双击打开 Vivado 开发平台，按照之前的步骤创建新的设计工程并命名为“dma_loop_test”，根据开发板型号选取相应的芯片型号。创建完成后点击 Create Block Design 创建新的硬件设计，并命名为“dma_loop_test”。

之后在 Diagram 页面添加 ZYNQ7 Processing System 模块，双击该模块进入



配置页面。使能 S AXA HP 总线端口 0，如下图所示：

图 16.2.1 使能 S AXI HP 总线

配置串口，如下图所示：

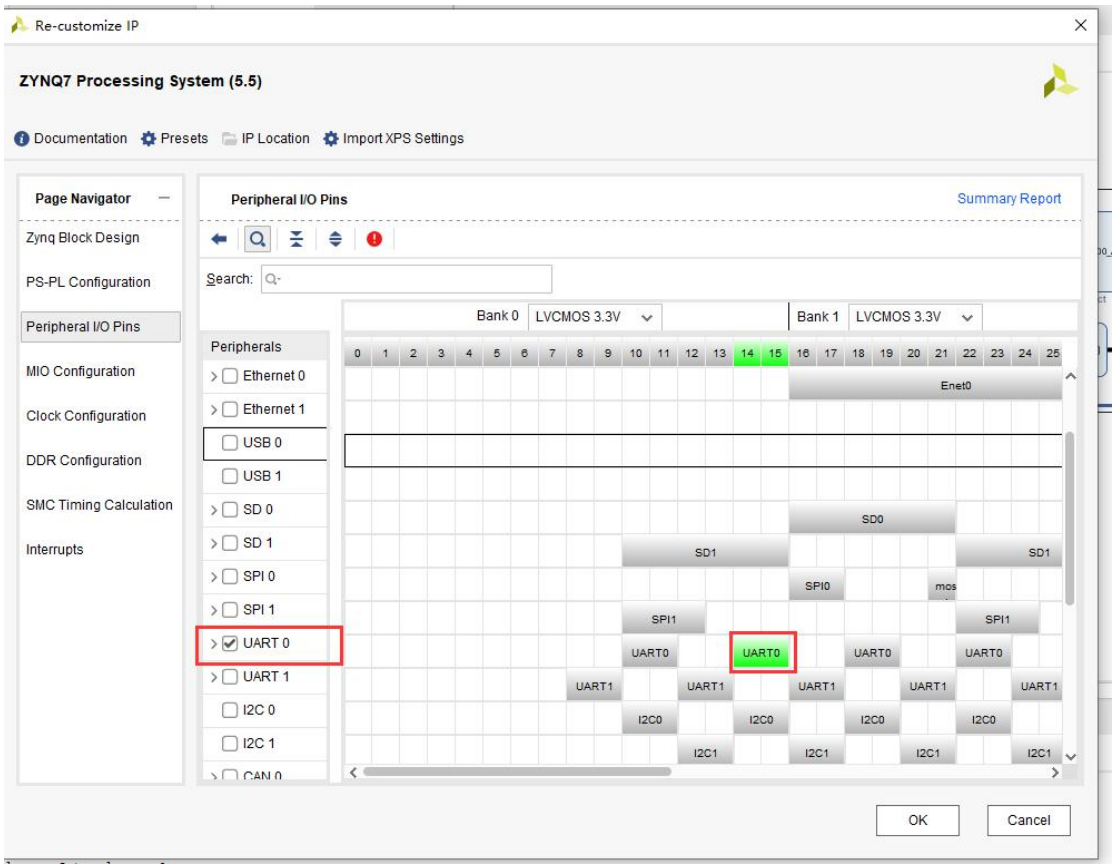
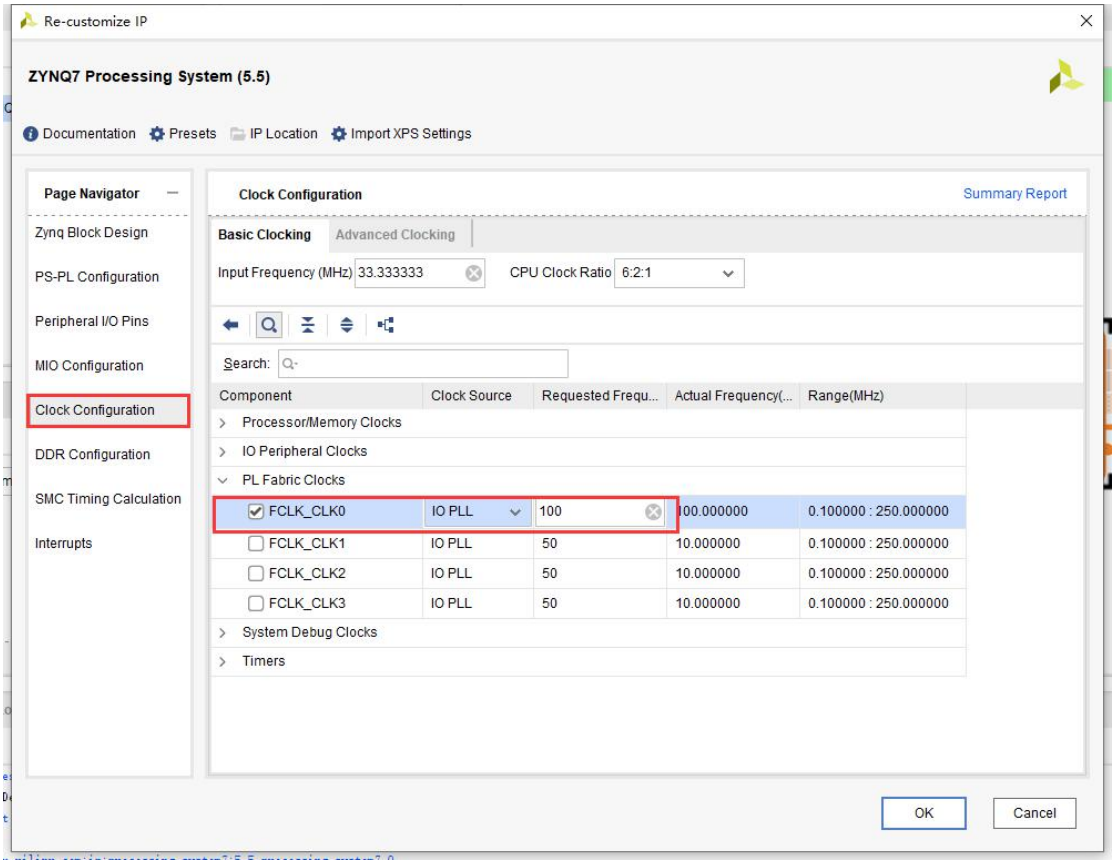
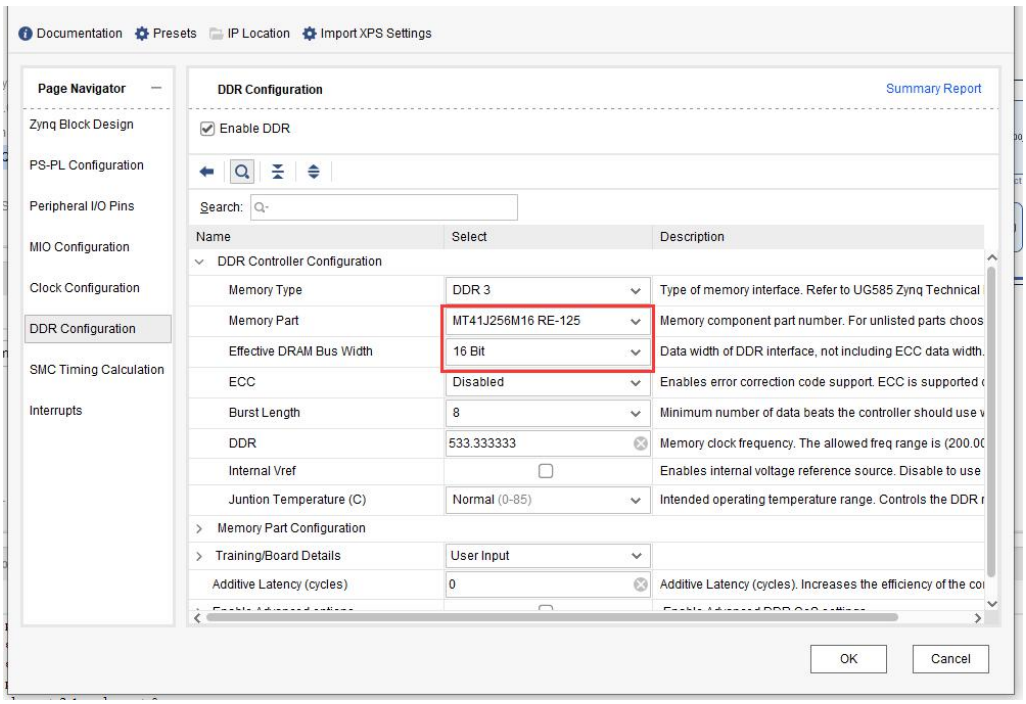


图 16.2.2 配置串口



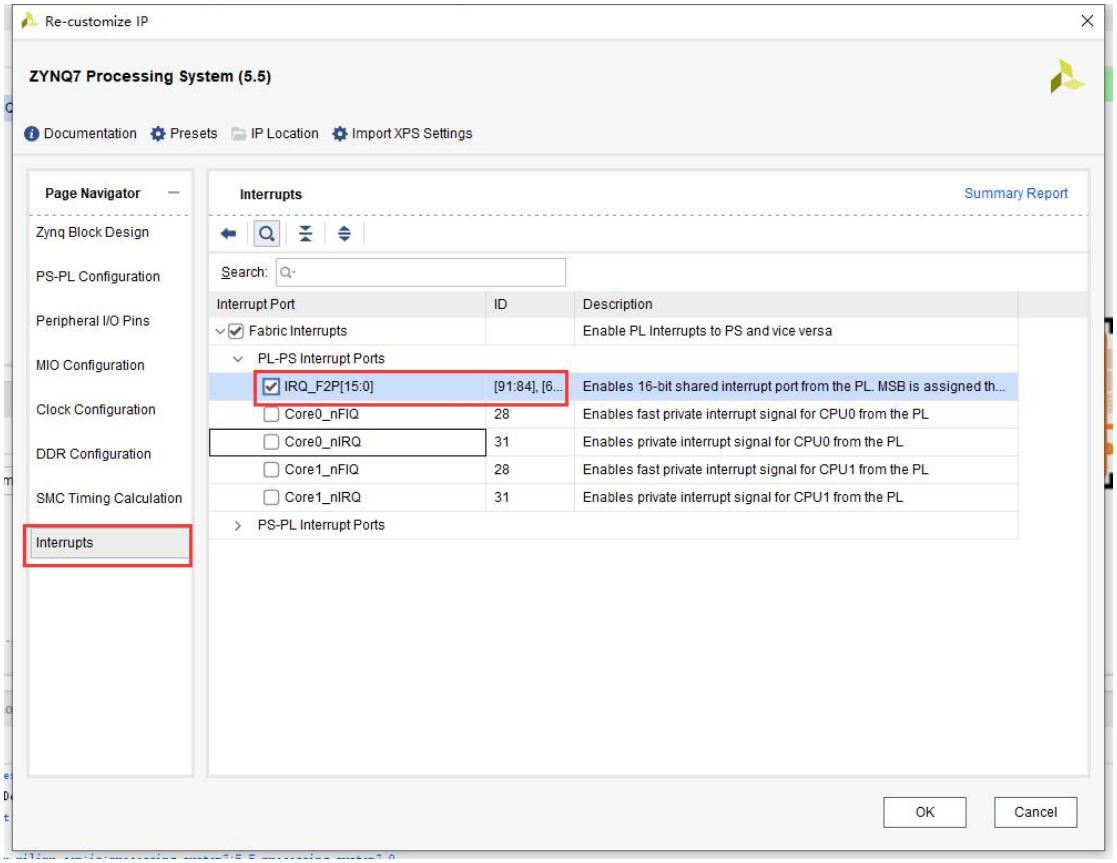
使能 FCLK_CLK0 并配置时钟频率为 100Mhz，如下图所示：

图 16.2.3 配置时钟频率



配置 DDR 控制器型号，如下图所示：

图 16.2.4 配置 DDR 控制器



开启 PL 到 PS 的中断，如下图所示：

图 16.2.5 使能中断

配置完成的 ZYNQ7 Processing System 模块如下图所示：

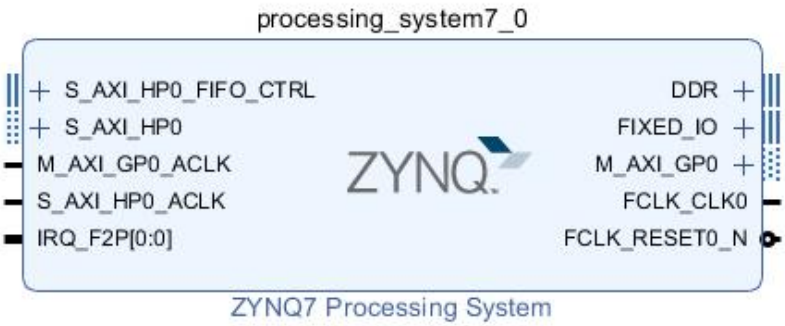


图 16.2.6 ZYNQ7 Processing System 模块

之后点击“ADD IP”打开 IP 核列表，搜索框中输出 DMA 查找 AXI Direct Memory Access IP 并添加到设计中，如下图所示：

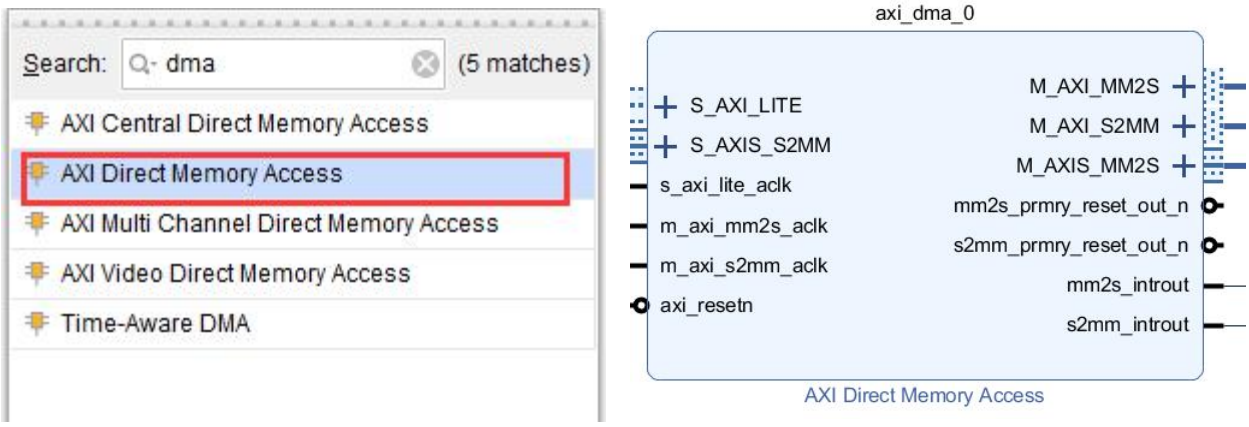


图 16.2.7 添加 AXI Direct Memory Access IP

双击该 IP 核打开配置页面，如下图所示配置相关设置。

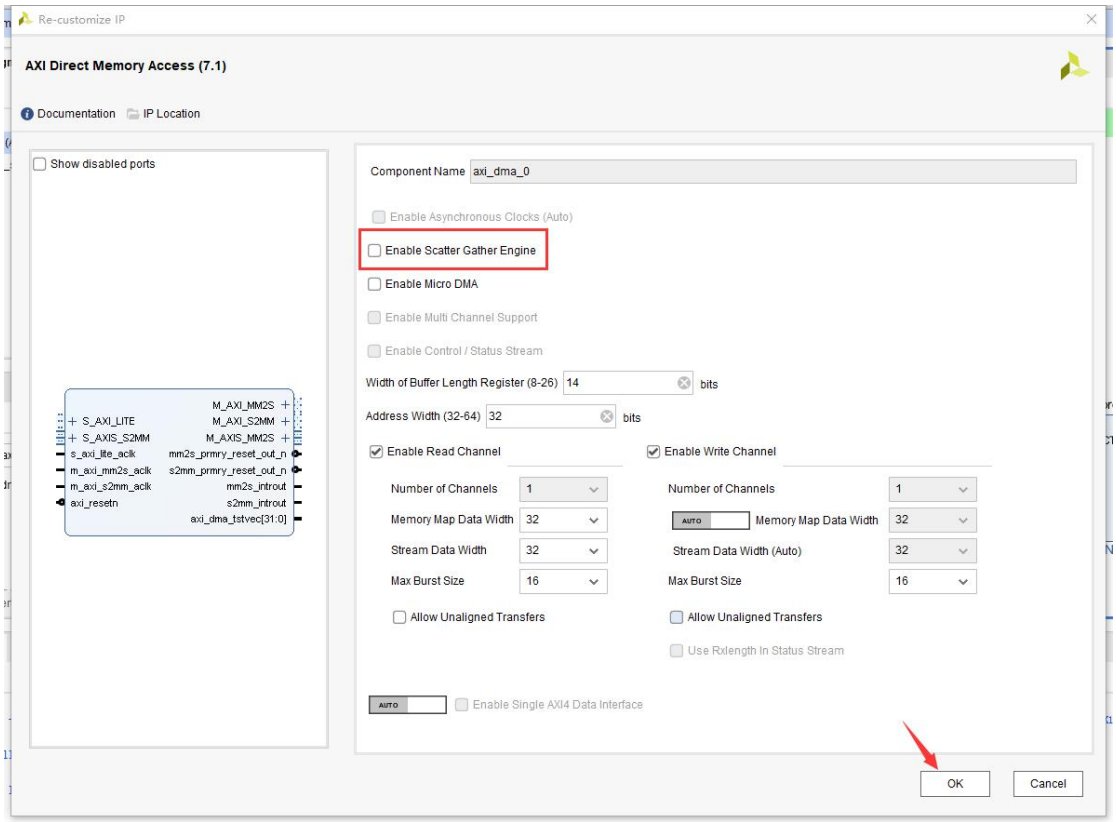


图 16.2.8 配置 DMA IP 核相关参数及设置

添加 axis_data_fifo IP，在输入框中输入 DATA，双击 AXI4-Stream Data FIFO 添加，如下图所示:

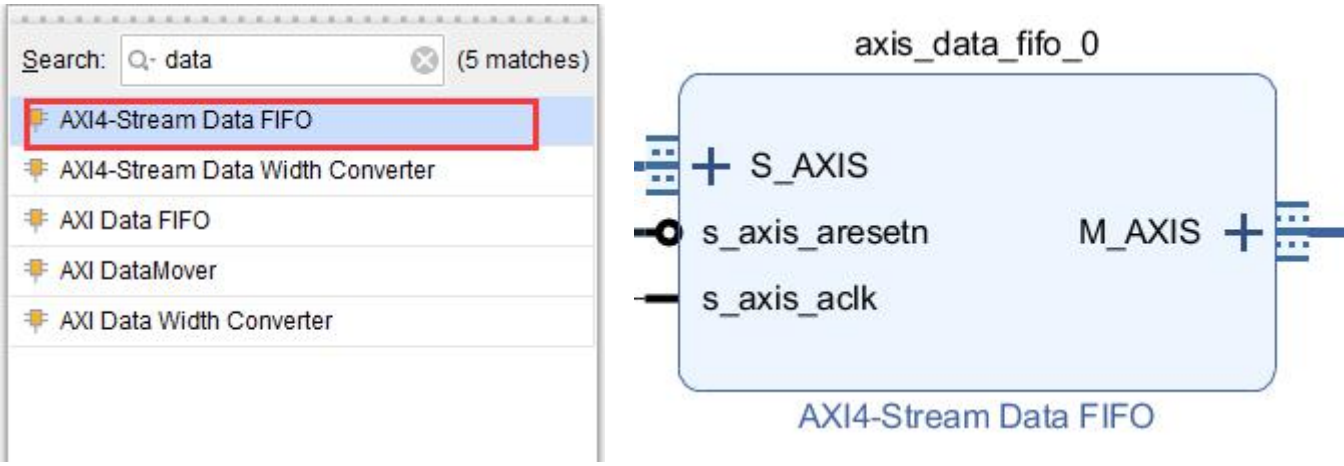
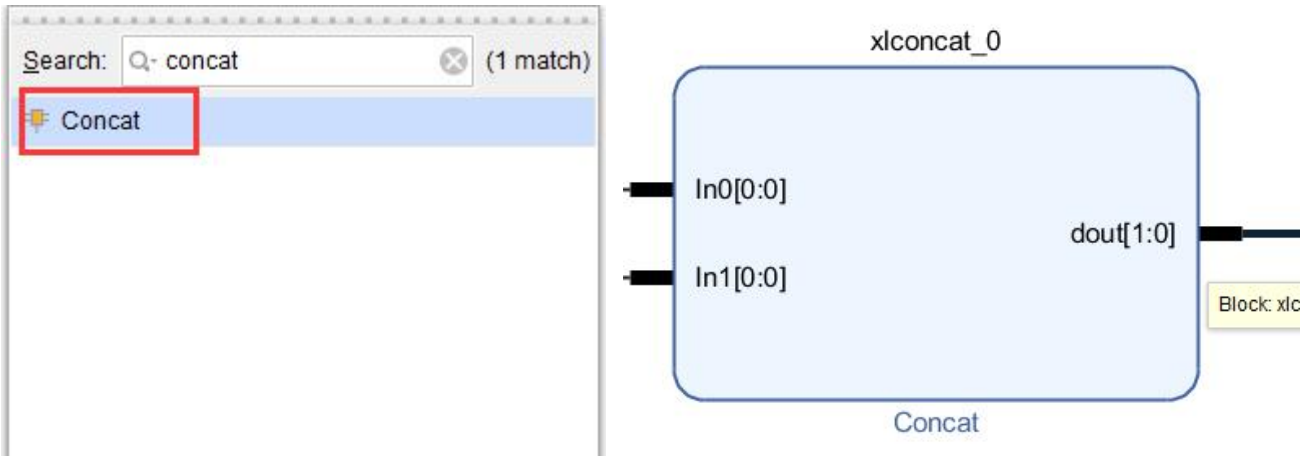


图 16.2.9 添加 axis_data_fifo IP



之后再添加 Concat IP 核，如下图所示：

图 16.2.10 添加 Concat IP 核

添加完成后分别点击 Run Block Automation 和 Run Connection Automation 引出端口并连接相关端口。

在 Run Block Automation 弹出的窗口左侧选择 processing_system7_0，如下图所示：

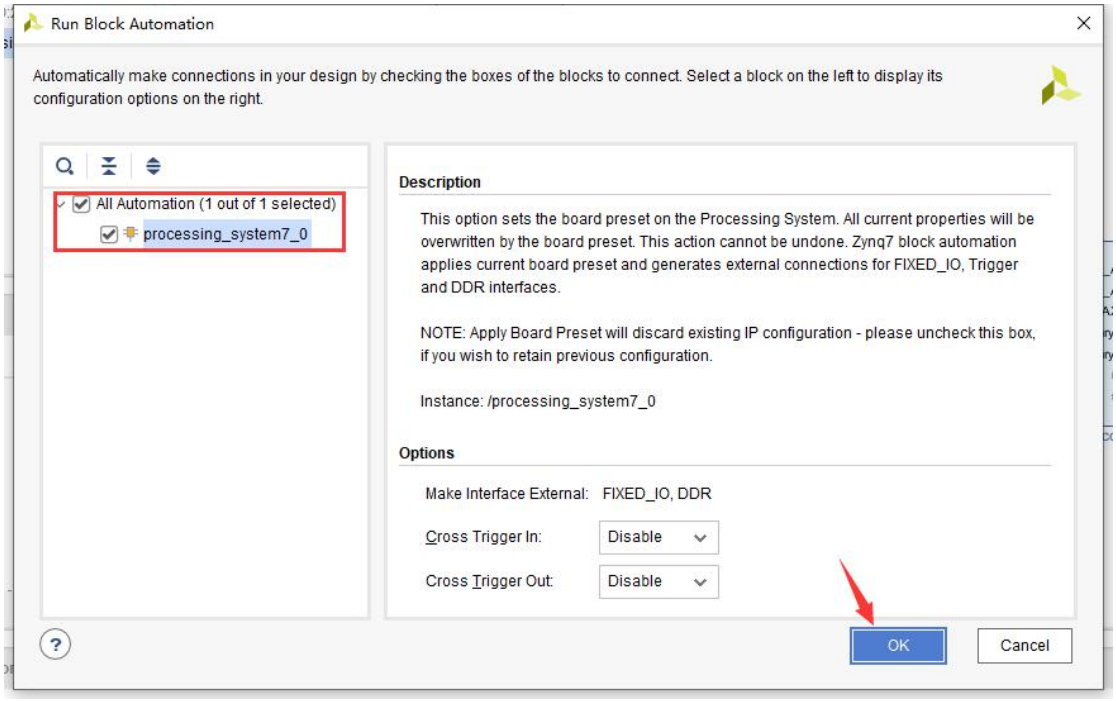


图 16.2.11 引出端口

在 Run Connection Automation 弹出的窗口左侧全选所以端口，点击“OK”，如下图所示：

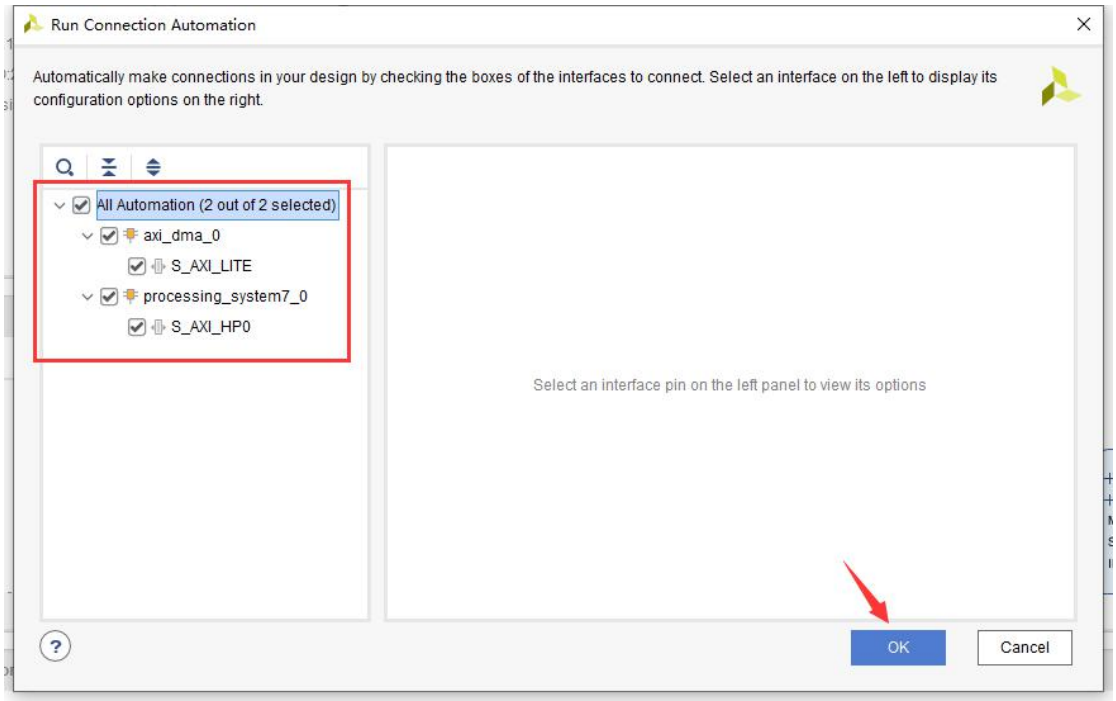


图 16.2.12 连接端口

自动连接完成后会发现我们添加的 concat IP 核以及 AXI4-Stream Data FIFO IP 核并未连接，这是因为我们添加的这两个 IP 核的端口并没有强制规定固定的连接关系，它们各端口的连接是根据实际设计时的情况来确定的，需要手动连接。如下图所示连接 DMA IP 的中断输出接口和 concat IP 的信号输入接口，连接 concat IP 的输出端口和 ZYNQ7 模块的中断输入接口。

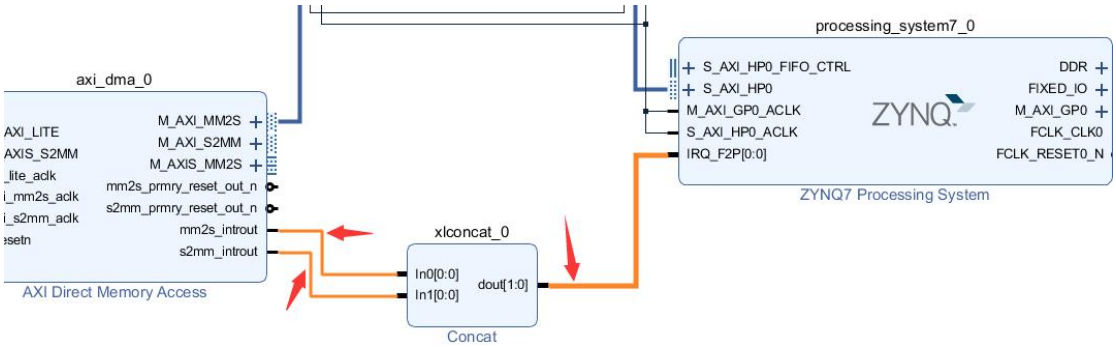
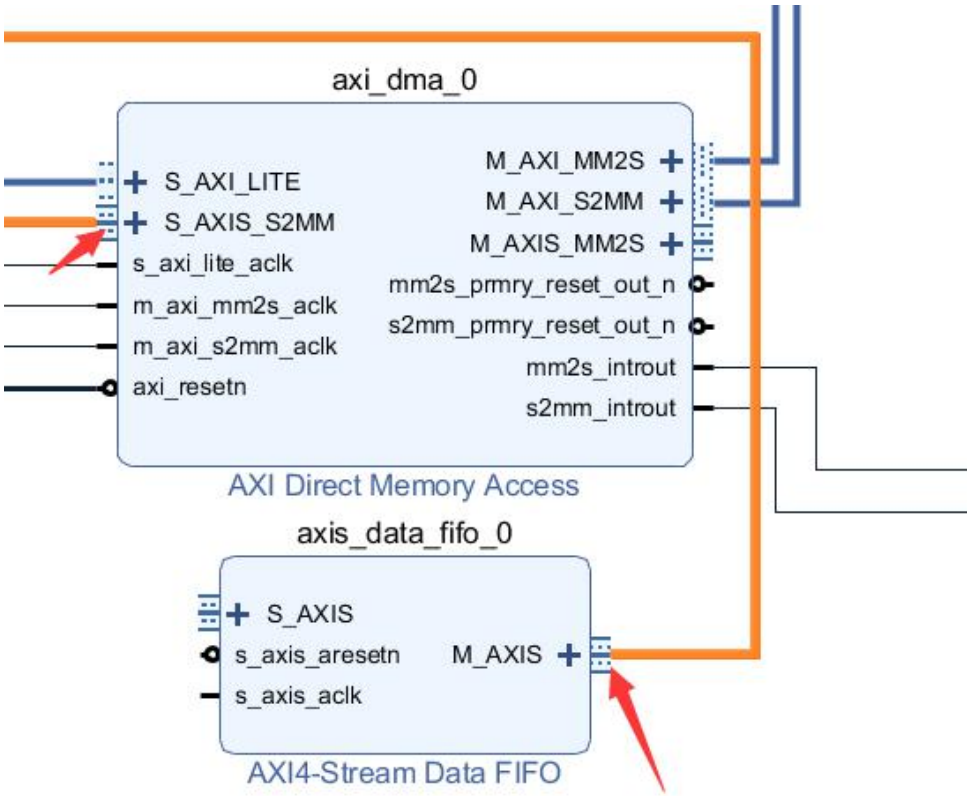


图 16.2.13 连接中断端口信号

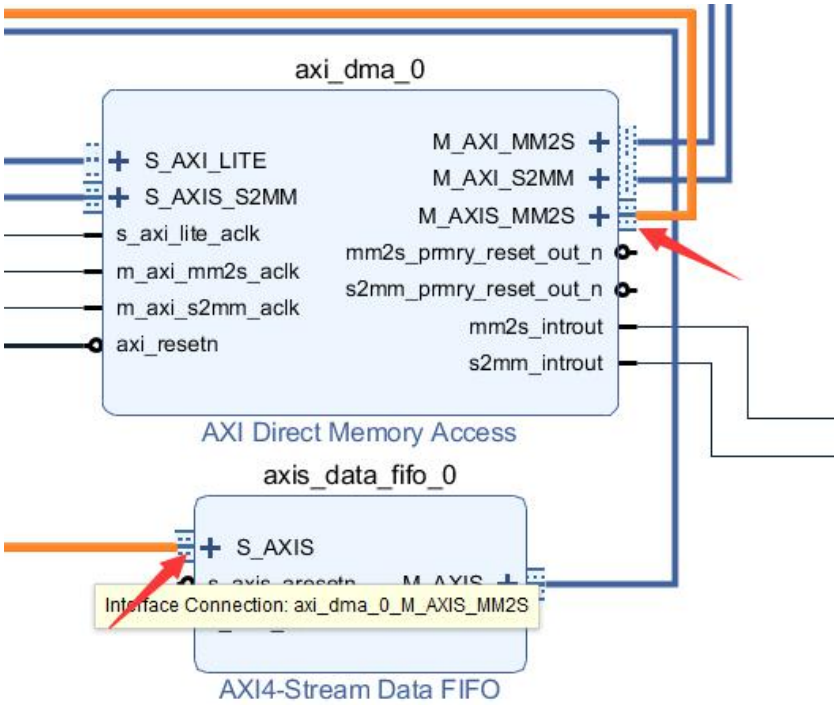
连接 DMA IP 的 S_AXIS_S2MM 端口和 AXI4-Stream Data FIFO IP 的 M_AXIS 端



口，如下图所示：

图 16.2.14 连接 S_AXIS_S2MM 和 M_AXIS

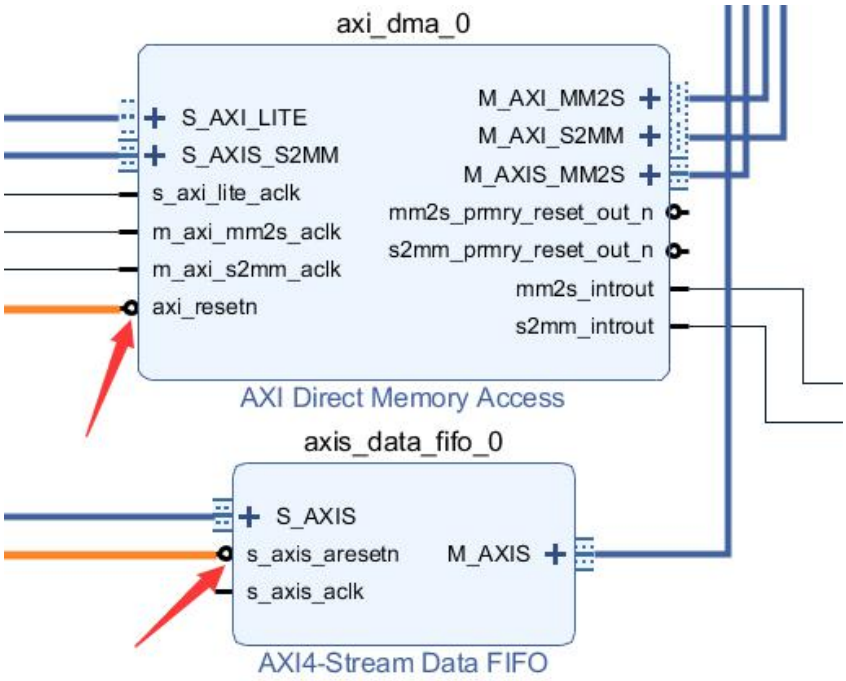
连接 DMA IP 的 M_AXIS_MM2S 端口和 AXI4-Stream Data FIFO IP 的 S_AXIS 端



口，如下图所示：

图 16.2.15 连接 M_AXIS_MM2S 和 S_AXIS

连接 DMA IP 的 axi_resetn 端口和 AXI4-Stream Data FIFO IP 的 s_axis_aresetn



端口，如下图所示：

图 16.2.16 连接 axi_resetn 和 s_axis_areset

连接 DMA IP 的 m_axi_s2mm_aclk 端口和 AXI4-Stream Data FIFO IP 的 s_axis_aclk 端口，如下图所示：

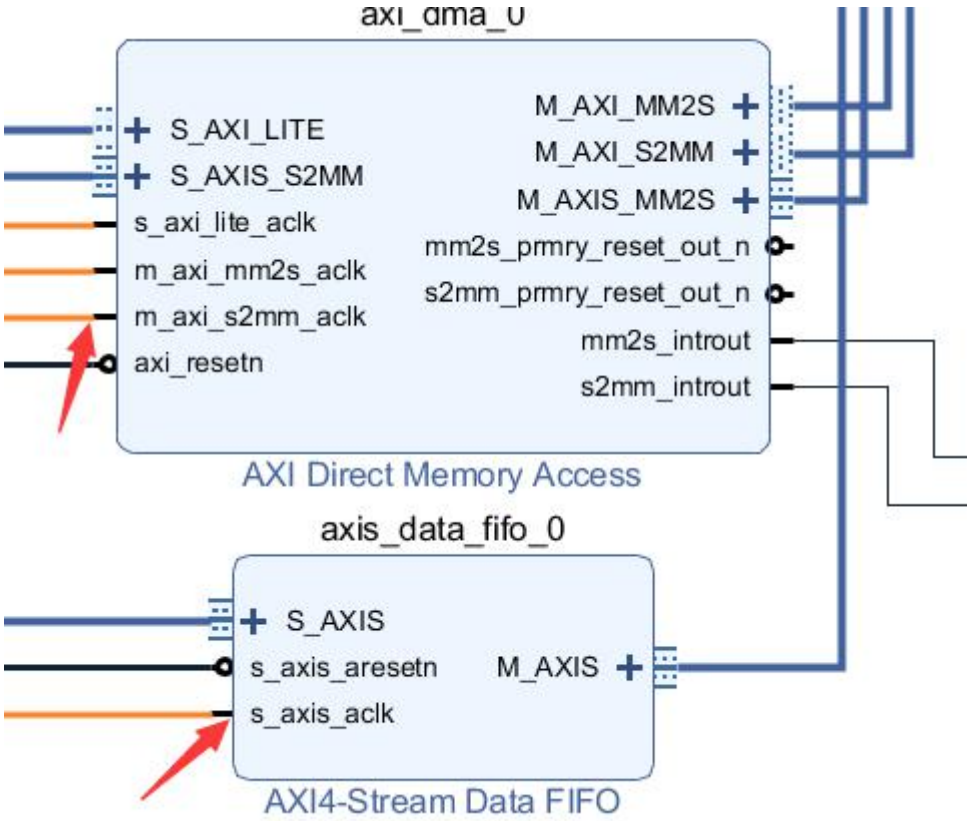


图 16.2.17 连接 m_axi_s2mm_aclk 和 s_axis_aclk

连接完成后点击重新布局，完成的设计如下图所示：

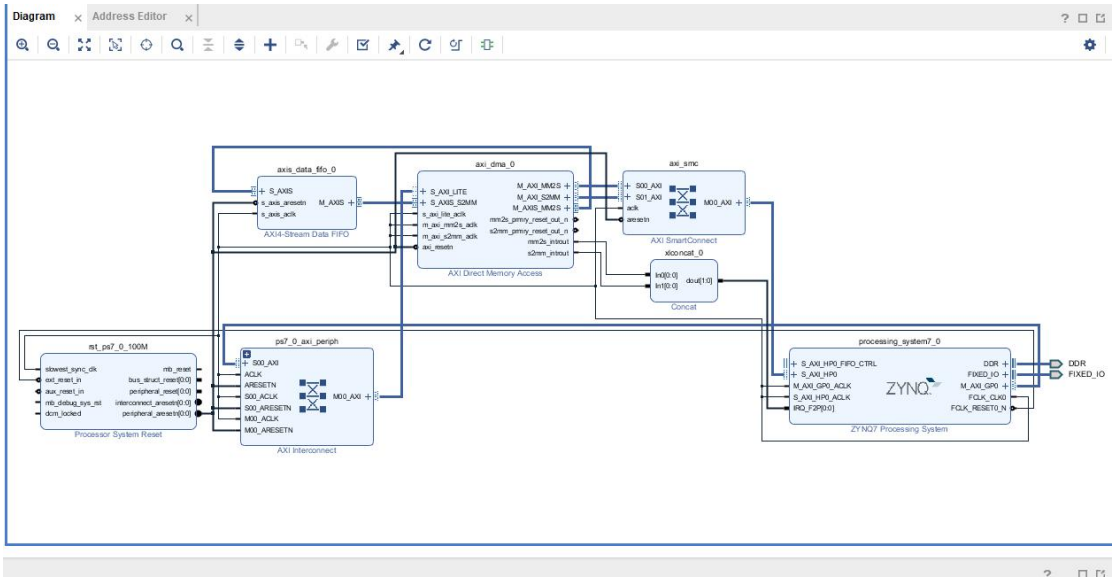


图 16.2.18 Block Design 设计图

到这里我们的 Block Design 就设计完成了，按快捷键“Ctrl + S”保存设计。之后按“F6”键进行“Validate Design”验证设计。验证完成后弹出对话框提示“Validation Successful”表明设计无误，点击“OK”确认。

在 Sources 窗口中，选中 Design Sources 下的 dma_loop_test.bd，右键点击 dma_loop_test.bd， 然后依次执行“Generate Output Products”和“Create HDL Wrapper”。生成顶层 HDL。

在左侧导航栏中找到 PROGRAM AND DEBUG，点击该选项中的“Generate Bitstream”。在连续弹出的对话框中依次点击“YES”、“OK”。然后 Vivado 工具开始依次对设计进行综合、实现、并生成 Bitstream 文件。

在生成 Bitstream 之后， 在菜单栏中选择 File > Export > Export hardware 导出硬件，并在弹出的对话框中，勾选“Include bitstream”。然后在菜单栏选择 File > Launch SDK，启动 SDK 软件。

16.3 软件设计

在打开的 SDK 软件中新建一个工程并命名为“dma_loop_test”，工程模板选择空白模板。创建完成后右键点击 dma_loop_test->src 向工程中添加主函数源文件并命名为“main.c”，之后再打开的编辑窗口中输入以下程序：

```
#include "dma_intr.h"
#include "sys_intr.h"

static XScuGic Intc; //GIC中断控制器
static XAxiDma AxiDma;

int Tries = 6;
int i;
int Index;
u8 *TxBufferPtr = (u8 *) TX_BUFFER_BASE;
u8 *RxBufferPtr = (u8 *) RX_BUFFER_BASE;
u8 Value;

int axi_dma_test();
```

```c
int axi_dma_test()
{
    int Status;
    TxDone = 0;
    RxDone = 0;
    Error = 0;

    xil_printf("\r\n----DMA Test----\r\n");
    xil_printf("PKT_LEN=%d\r\n", MAX_PKT_LEN);

    for (i = 0; i < Tries; i++) {
        Value = 0x55 + (i & 0xFF);
        for (Index = 0; Index < MAX_PKT_LEN; Index++) {
            TxBufferPtr[Index] = Value;

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

        Xil_DCacheFlushRange((u32) TxBufferPtr, MAX_PKT_LEN); //数据刷新到DDR中

        Status = XAxiDma_SimpleTransfer(&AxiDma, (u32) TxBufferPtr, //配置发送通道
                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }

        Status = XAxiDma_SimpleTransfer(&AxiDma, (u32) RxBufferPtr, //配置接收通道
                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }

        while (!TxDone || !RxDone) {
        } //等待收发中断都到达
        TxDone = 0;
        RxDone = 0;
        Xil_DCacheInvalidateRange((u32) RxBufferPtr, MAX_PKT_LEN); //刷新cache，观察DDR的最新数据

        if (Error) {
            return XST_FAILURE;
        }
    }

    DMA_DisableIntrSystem(&Intc, TX_INTR_ID, RX_INTR_ID); //失能DMA中断
    xil_printf("--- Exiting Test --- \r\n");
    return XST_SUCCESS;
}
```

```c
int main(void)
{
    DMA_Intr_Init(&AxiDma, 0); //DMA初始化
    Init_Intr_System(&Intc); //系统初始化
    Setup_Intr_Exception(&Intc); //使能硬件中断
    DMA_Setup_Intr_System(&Intc, &AxiDma, TX_INTR_ID, RX_INTR_ID); //注册DMA收发中断
    DMA_Intr_Enable(&Intc, &AxiDma); //使能系统中断

    axi_dma_test();
}
```

在本次实验中，我们没有将所有程序都放到 main.c 中，而是将相关的初始化设置放到其他两个 dma_intr.c 和 sys_intr.c 文件中，读者使用时只需要将例程中

的 dma_intr.c、dma_intr.h、sys_intr.c 和 sys_intr.h 四个文件拷贝到工程目录下的 src 文件夹，与 main.c 在同一个文件夹内即可，SDK 软件会自动添加。此处不再显示全部程序，请读者自行查阅。

程序输入完成后快捷键 Ctrl + S 保存并编译程序。

16.4 下载验证

首先我们将使用 Micro USB 线与 JTAG 接口连接，另外一端与电脑连接。然后使用 Micro USB 连接线将开发板的 UART 接口与电脑连接，用于串口通信。最后连接开发板的电源。连接串口并配置相关设置，之后将程序烧录到芯片中，可以在 SDK Terminal 窗口中看到打印的信息如下图所示，说明实验成功。

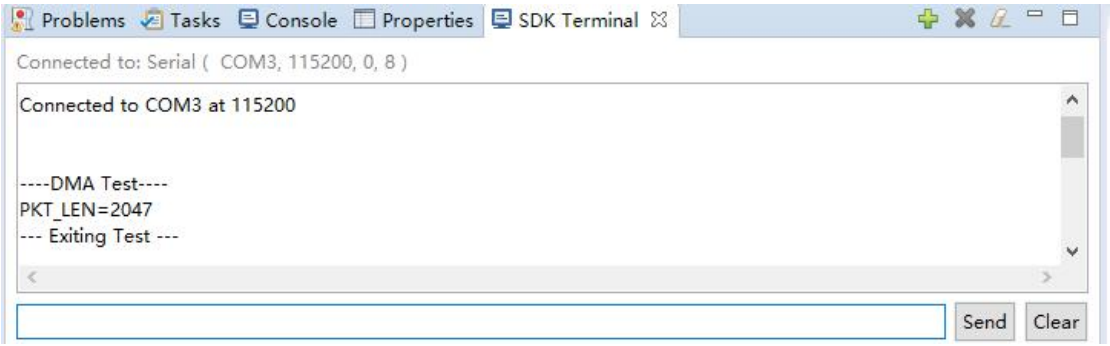


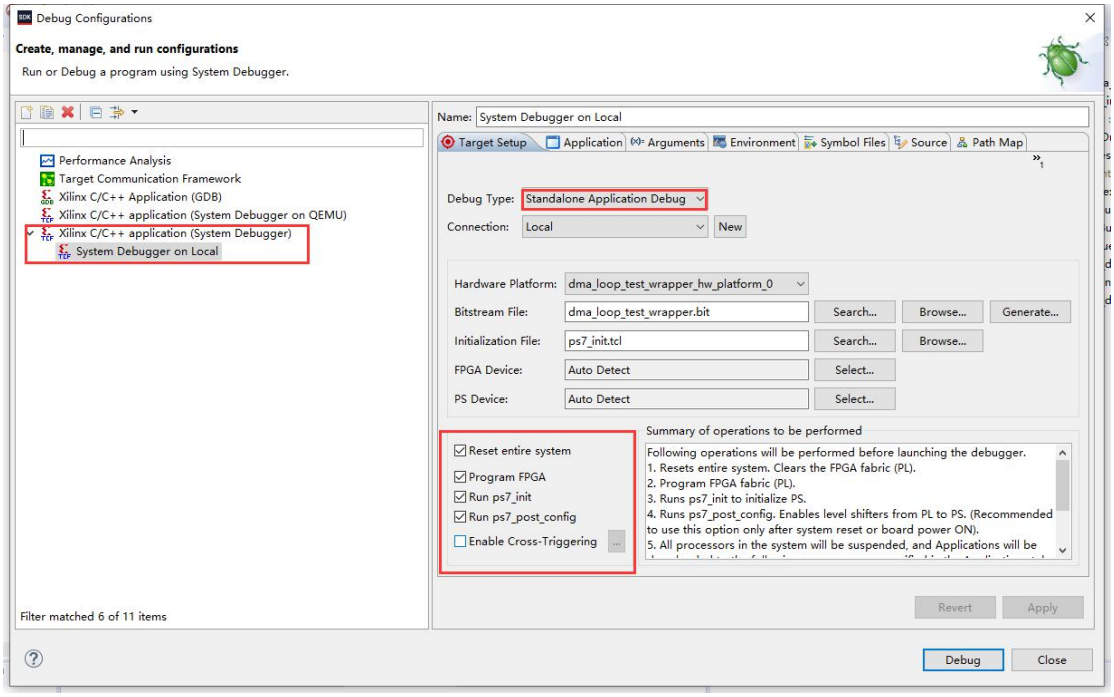
图 16.4.1 串口打印信息

但是这样我们并不了解数据传输的具体过程，接下来我们介绍怎样利用 SDK 的在线调试功能查看数据传输的过程。在主函数源文件内，双击行数前面的位置，设置几个断点，如下图所示：

```
44
45      Xil_DCacheFlushRange((u32) TxBufferPtr, MAX_PKT_LEN); //数据刷新到DDR中
46
47      Status = XAxiDma_SimpleTransfer(&AxiDma, (u32) TxBufferPtr, //配置接收通道
48                 MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
49      if (Status != XST_SUCCESS) {
50          return XST_FAILURE;
51      }
52
53      Status = XAxiDma_SimpleTransfer(&AxiDma, (u32) RxBufferPtr, //配置发送通道
54                 MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
55      if (Status != XST_SUCCESS) {
56          return XST_FAILURE;
57      }
58
59      while (!TxDone || !RxDone) {
60      } //等待收发中断都到达
61
62      TxDone = 0;
63      RxDone = 0;
64      Xil_DCacheInvalidateRange((u32) RxBufferPtr, MAX_PKT_LEN); //刷新cache，观察DDR的最新数据
65      if (Error) {
66          return XST_FAILURE;
67      }
```

图 16.4.2 设置断点

设置完成后点击菜单栏 Run->Debug Configurations..，进入 Debug 配置页面，双击 Xilinx C/C++ application(System Debugger)，按照下图所示配置相关设置。配置完成后点击“Debug”进入在线调试。



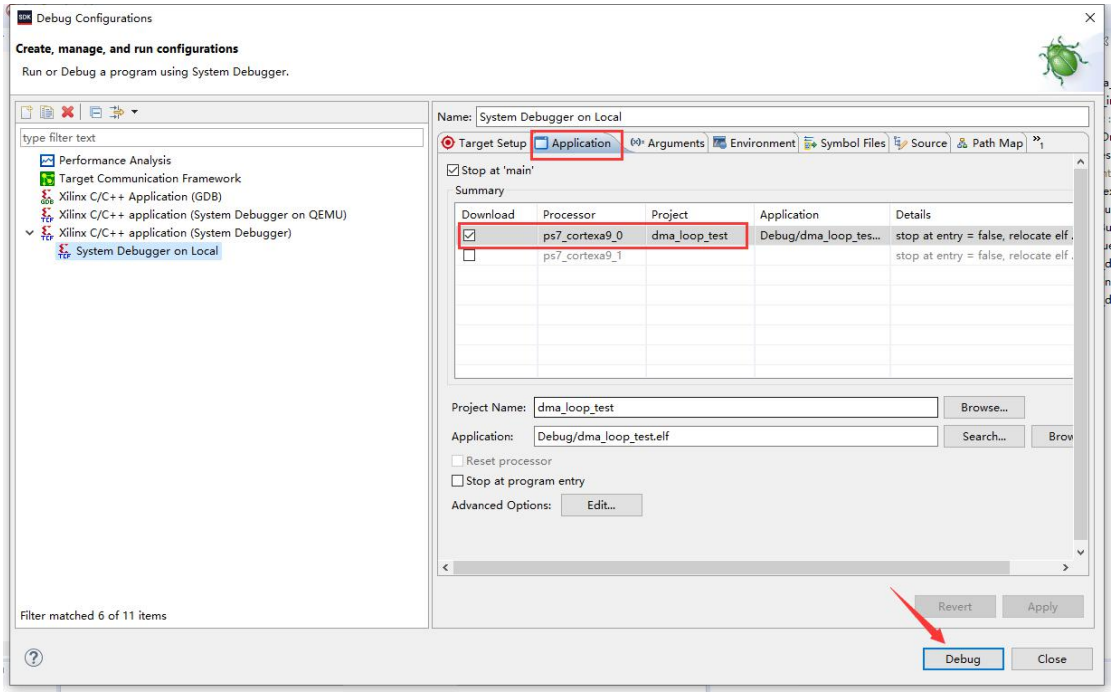


图 16.4.3 配置 Debug 相关设置

在开始调试之前在页面左下角的 Menory 窗口内添加内存地址以便于观察数据的变化。点击绿色的“+”，之后再弹出的窗口输入“0x01100000”，点击“OK”，如下图所示，即可将地址添加到内存数据观察窗口。

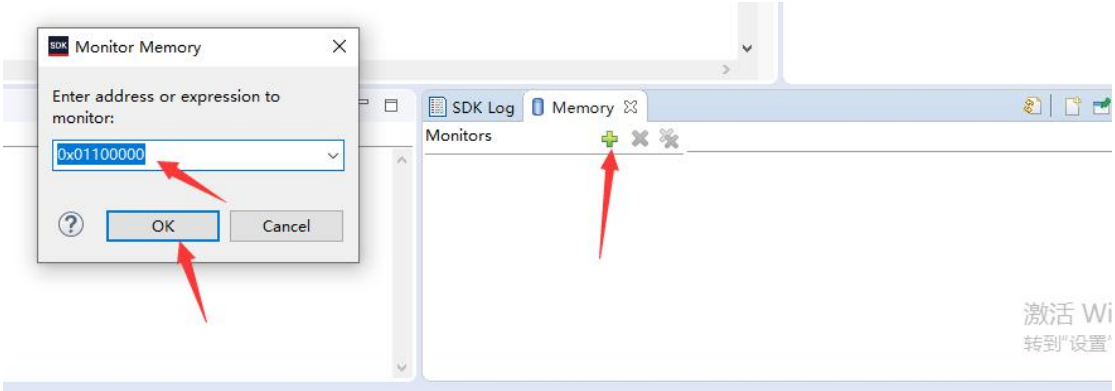


图 16.4.4 添加内存地址

右键点击右侧的空白处，选择 Format...设置数据显示形式，如下图所示：

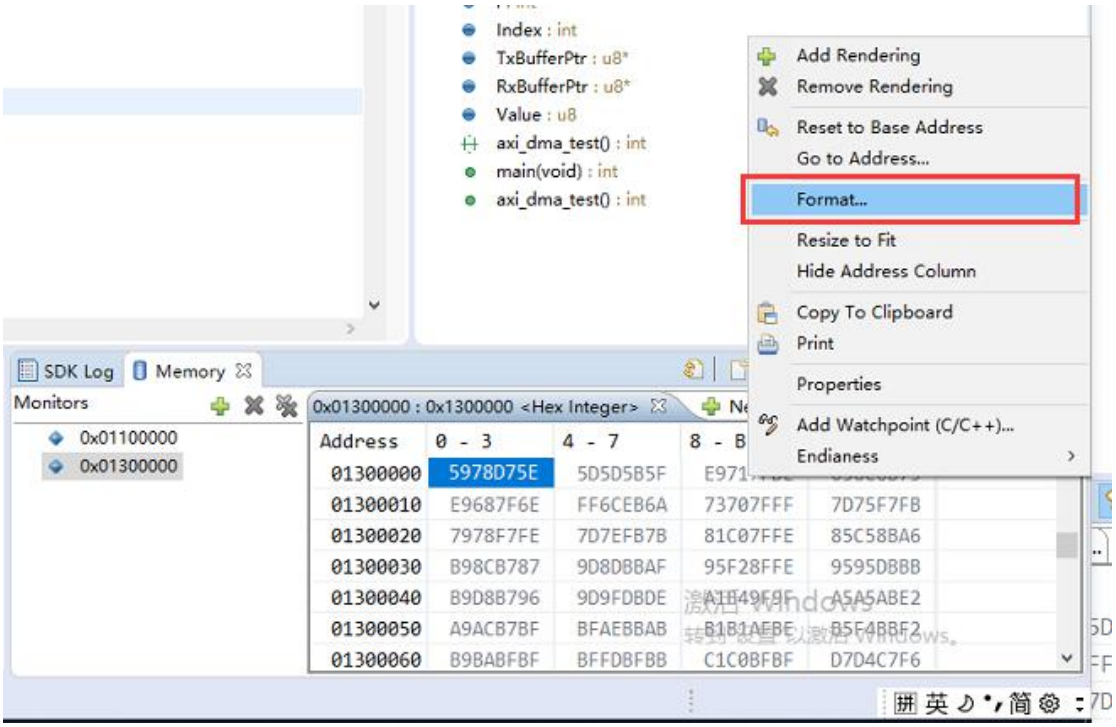
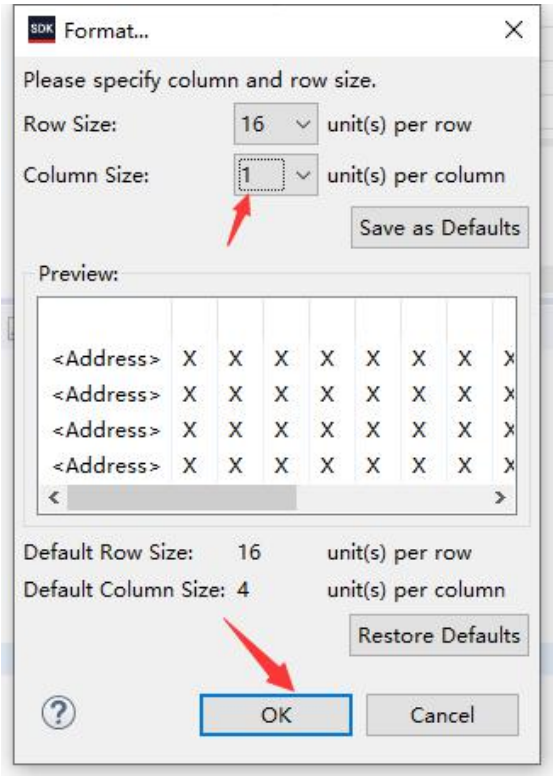
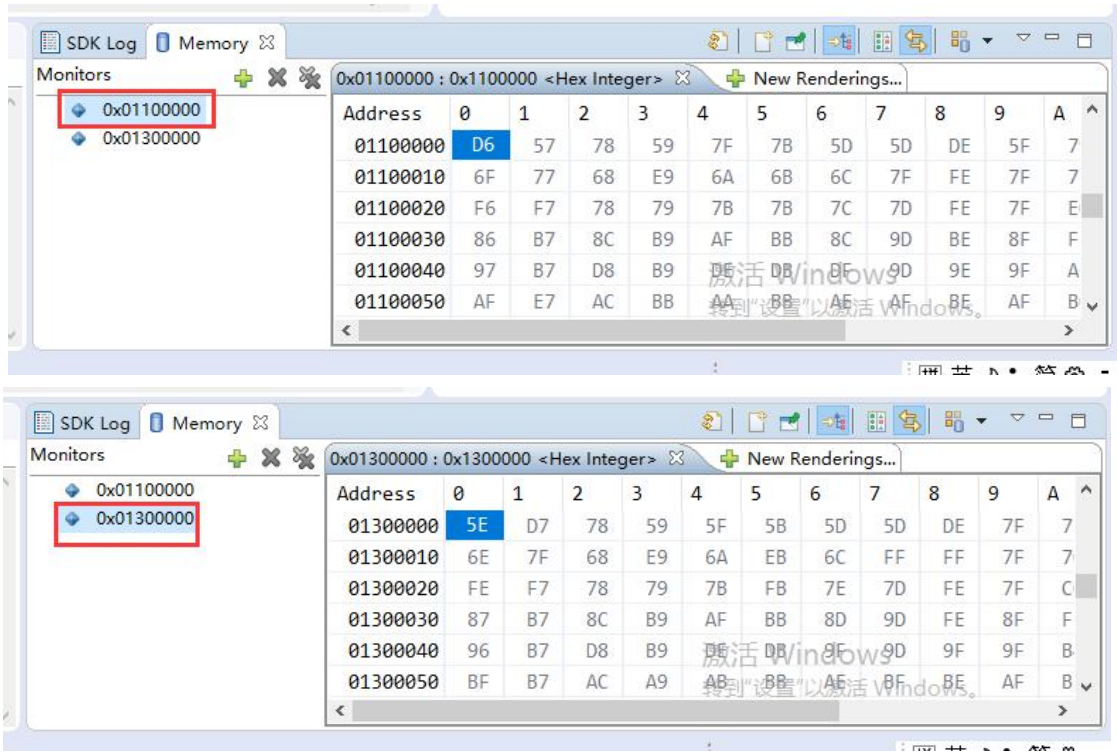


图 16.4.5 修改显示形式



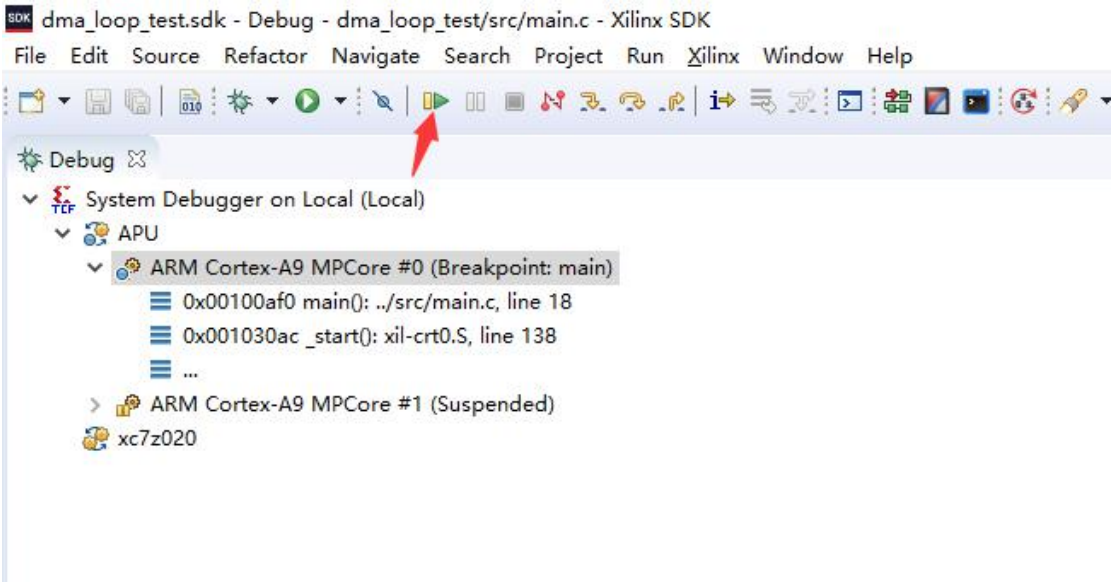
在弹出的窗口中将下图所示位置参数改为“1”。

图 16.4.6 修改参数



可以看到每个地址的数据分开显示，并且所有数据无规律为随机数。

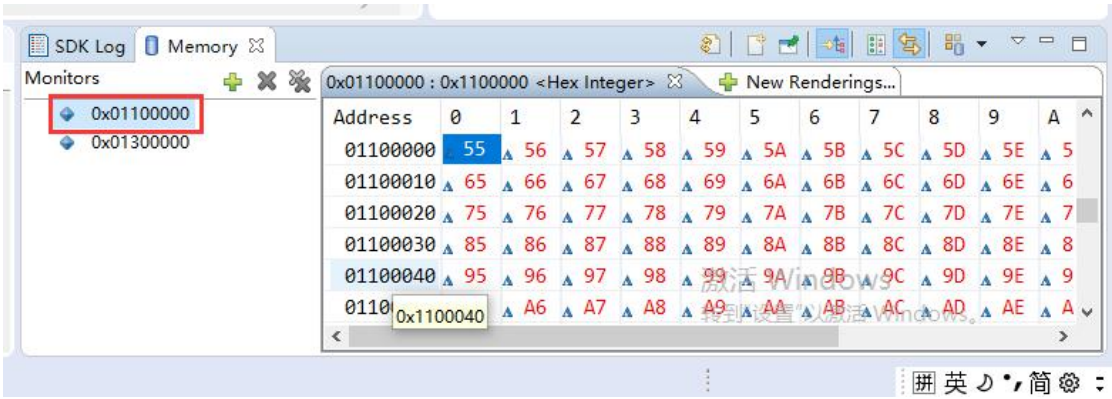
图 16.4.7 初始数据



之后点击运行按钮使程序运行到第一个断点处，如下图所示。

图 16.4.8 开始运行

在第一个断点处，此时我们可以看到下方窗口的数据已经更新，此时从程序中可以看出此时我们还未向 DDR3 中写入数据，此时看到的数据是 Cache 中的数据。CPU 与 DDR3 之间是通过 Cache 交互的，先将数据暂存在 Cache 中，之



后刷新 Data Cache 数据到 DDR3。

图 16.4.9 0x01100000 地址数据变化

接着点击运行使程序运行到断点 2 处，在数据监视窗口切换到观察地址 0x01300000 的数据，点击左侧的地址即可。我们观察它的数据何时变化。运行到断点 3 处，执行完 DMA 发送函数，完成从内存中读取数据传输给外设，即 DMA 从地址 0x01100000 处读取数据传输给外设 DDR3，此时地址 0x01300000 处的数据未更新。运行到断点 4 处，执行完 DMA 接收函数，完成从外设读取数据写入到内存，即将刚才写入到外设的数据读取出来并从 DDR3 的地址 0x01300000 处开始写入，不过此时我们从下图发现地址 0x01300000 处的数据还是未更新。其实此时 DDR3 中的数据已经更新，只不过我们 Data Cache 中的数据未更新，而 Memory Monitors 窗口显示的正是 Data Cache 中的数据，所以需要刷新 Data Cache。

接着运行到断点 5 处，刷新 Data Cache 后，此时我们发现地址 0x01300000 处的值变为 55，紧随其后的地址处的数据都变成预期的值。这就是 DMA 传输数据的整个过程。在程序中我们设计的是循环读写 6 次测试连续读写，上面描述的是执行一次的过程。运行 6 次之后可以在 SDK Terminal 窗口看到打印信息表明执行结束。

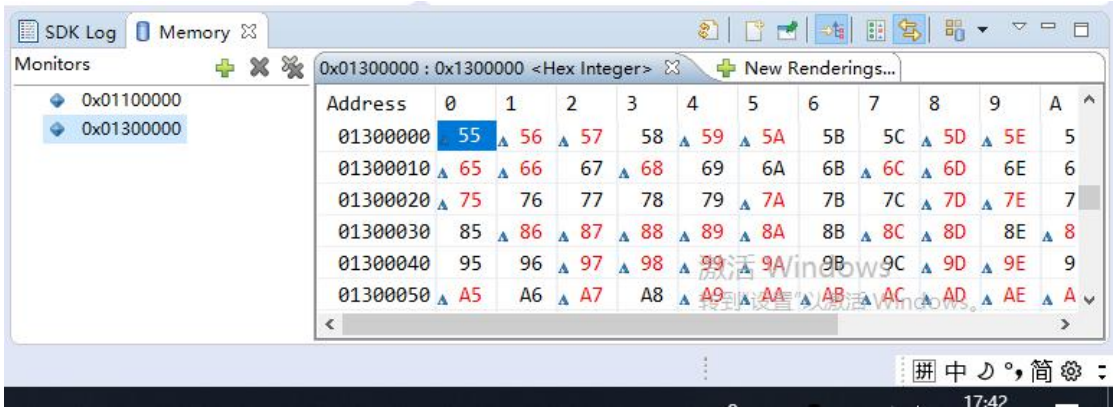


图 16.4.10 0x01300000 地址数据变化

16.5 章节小结

在本实验中，我们简单了解了 PL 端 DMA 的使用方法，并未结合实际的应用场景，DMA 读取和写入两个不同的外设，例如读取 SD 卡内的图片传输到 HDMA 在屏幕上显示出来。读者可以在此实验的基础上尝试实现该应用。

第 17 章　　HDMI 显示实验

在 ZYNQ 的众多应用中，视频图像显示是经常用到的方面，在我们的开发板上也设计了 HDMI 视频输入输出接口。在关于 fpga 的视频显示应用中，常见的方式有两种，针对单一的 HDMI 显示，可以使用编写 IP 核的方式解码视频数据，输出 HDMI 信号；对于显示终端多种多样的时候，可以使用转码芯片的方式，这样使用起来更加灵活。针对不同客户的需求，我司出品的不同型号的产品采用不同的解决方案。对于 Mizar 型号的开发板采用 IP 核的方式，对于 XME0715-PE100 的开发板采用 ADV7513 解码芯片的方式。在本实验中，我们将讲解利用 VDMA 的 IP 在另种不同方式下实现 HDMI 的图像输出。读者可根据自己购买的开发板查阅相应内容。

本实验包括以下几部分：

17.1 实验简介

17.2 硬件设计

17.3 软件设计

17.4 下载验证

17.5 章节小结

17.1 实验简介

AXI VDMA（AXI Video Direct Memory Access，以下简称 VDMA）， 是 Xilinx 提供的软核 IP。该 IP 可以看作 DMA 的升级版，提供了一些适用于视频图像应用的功能。与 DMA 类似，该 IP 可以为存储器或者 AXI4-Stream 类目标外设之间提供高带宽直接存储器存取。在此基础上增加了帧缓存的缓冲机制和同步锁相（GenLock）等功能，同时集成了视频专用功能，如帧同步和 2D DMA 传输等，非常适合基于 ZYNQ 架构上的图像和视频处理 q 应用。

接下来介绍一些相关概念。

帧缓存

在日常生活中我们知道想要实现流畅的视频播放至少需要 24 帧/S，即每秒播放 24 幅图像。一般在图像处理设备中图像输入源和图像显示的传输速率是不匹配的（如图像输入源传输速度较快或者图像显示端传输速度较快），在这种时候直接读取输入源来显示显然是不合适，时候就需要一片存储区域来缓存输入的数据，以便显示设备读取使用，同时方便后续对视频数据做图像处理。这就是帧缓存存储器（Frame Buffer），简称帧缓存，也常被称作显存。帧缓存的每一个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。

在使用帧缓存来缓存图像数据时，可以采用单帧缓存或多帧缓存的方案。单帧缓存是指图像的输入和图像的显示都是通过读写同一片存储区域来实现的。而多帧缓存是指将不同的图像保存在不同的存储区域，显示时按顺序读取。显而易见，单帧缓存存在一个缺陷，那就是当数据源连续输入的时候，帧缓存保存的就可能是两帧或更多帧图像数据叠加的结果，当显示设备读取显示的时候就会出现图像割裂的现象。所以单帧缓存适用于输入速率小于读取速率的应用场景。对于摄像头图像显示或视频播放就需要用到多帧缓存。

同步锁相

实际应用场景中，为解决图像输入端和输出端的数据速率不匹配导致的潜在错误通常使用多帧缓存来保存数据。图像输入端在写入其中一个帧缓存时，输出端读取其它的帧缓存。这就涉及到帧缓存的读写策略，即同步锁相模式。

VDMA 支持四种同步锁相模式，分别是 Genlock Master（同步锁相主模式）、Genlock Slave（同步锁从模式）、 Dynamic Genlock Master（动态同步锁相主模式）和 Dynamic Genlock Slave（动态同步锁相从模式）。

VDMA 有一个写通道（S2MM）和一个读通道（MM2S），用户通过写通道将输入端数据写入帧缓存，通过读通道将从帧缓存中读出数据。 VDMA 的每一个通道都可以选择以上四种模式中的一种，接下来我们分别向大家介绍这四种同步模式。

Genlock Master（同步锁相主模式）

当写通道（S2MM）或者读通道（MM2S）配置为 Genlock Master 时，该通道不会跳过或者重复任一帧缓存区域，按照帧缓存顺序读出数据。配置为 Genlock Slave 的通道应当紧跟 Genlock Master 通道变化，但有一定的延迟，延迟的大小在寄存器（*frmdly_stride[28:24]）中配置。

Genlock Slave（同步锁从模式）

当写通道（S2MM）或者读通道（MM2S）配置为 Genlock Slave 时，该通道会通过跳过或者重复一些帧缓存区域的方式，尝试与 Genlock Master 通道同步。

Dynamic Genlock Master（动态同步锁相主模式）

当写通道（S2MM）或者读通道（MM2S）配置为 Dynamic Genlock Master 时，该通道会跳过 Dynamic Genlock Slave 通道正在操作的帧缓存，通过跳过或者重复一些帧缓存区域的方式来完成。

Dynamic Genlock Slave（动态同步锁相从模式）

当写通道（S2MM）或者读通道（MM2S）配置为 Dynamic Genlock Slave 时，该通道会操作 Dynamic Genlock Master 通道上一周期操作的帧。

在实际应用中，如果避开读通道和写通道同时访问同一帧缓存，那么 VMDA 必须配置成动态同步锁相的模式，且帧缓存数量要大于等于 3。

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：

Ø 使视频流能够高带宽直接接入内存

Ø 高效的二维 DMA 操作

Ø 独立的异步读写通道操作

Ø Gen-Lock 帧缓存同步机制

Ø 最多支持 32 个帧存

Ø 支持视频格式动态切换

Ø 猝发长度和行缓存深度可调节

Ø 处理器可以控制 IP 的初始化、状态、中断和管理寄存器

Ø 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 的框图如下所示：

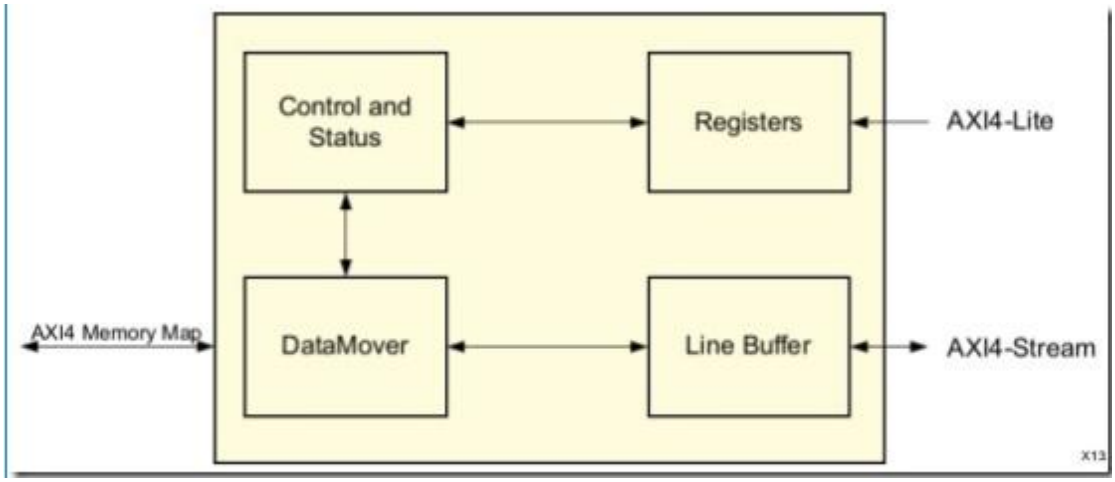


图 17.1.1 AXI VDMA 框图

主要有以下几种接口类型：

Ø AXI-lite： PS 通过该接口来配置 VDMA

Ø AXI Memory Map write：映射到存储器写

Ø AXI Memory Map read：映射到存储器读

Ø AXI Stream Write(S2MM)：AXI Stream 视频流写入图像

Ø 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。

在本实验中，我们将利用 AXI VDMA 和 HDMI 信号输出的 IP，利用开发板输出 HDMI 信号，在电脑显示器上显示图片。

17.2 硬件设计

1.1.1 IP 核解码方式

首先按照之前的步骤建立一个新的工程并命名为“hdmi_out”，选取相应的芯片型号。新建完成后，复制开发板配套文件中的 SDK_Demo/17_hdmi_out /hdmi_out 文件夹下的 repo 文件夹，粘贴到新建工程文件夹下的相同位置，如下图所示。

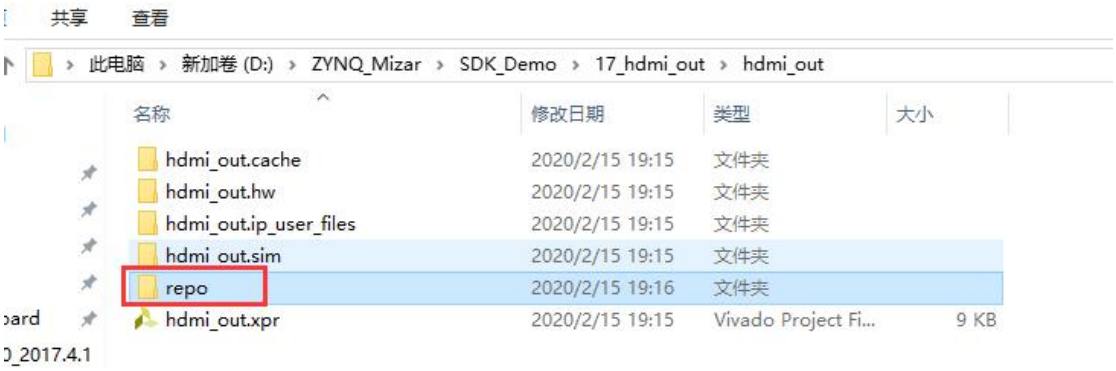
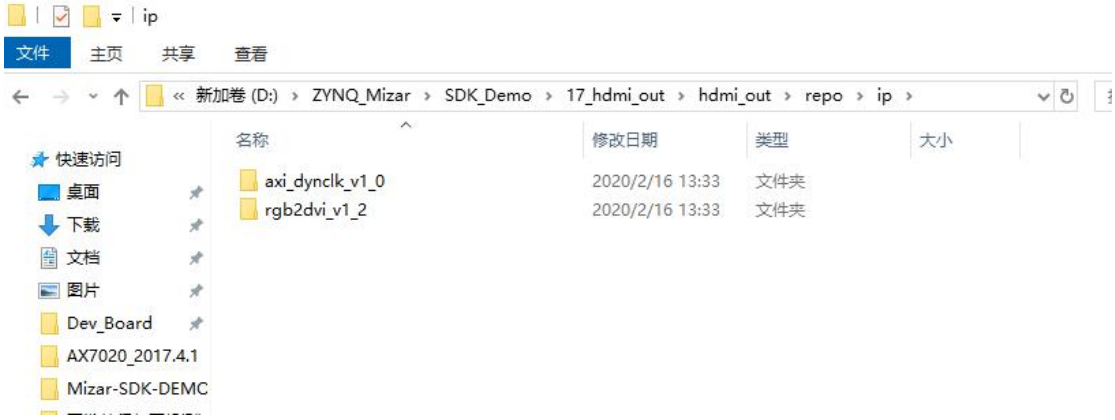


图 17.2.1 复制粘贴 repo 文件夹

该文件夹内有 Digilent 提供的两个自定义 IP，其中 axi_dynclk_v1_0 是时钟发生模块，会根据不同的屏幕分辨率，自动生成相应的像素时钟和串行时钟，而 rgb2dvi_v1_2 模块会将图像的红绿蓝信号转换成 TMDS 信号，送往 HDMI 端口



发出，如下图所示：

图 17.2.2 自定义 IP

在主窗口左侧边栏 Project Manager 下点击 Project Settings 选项，向工程中添加这两个自定义 IP，如下图所示：

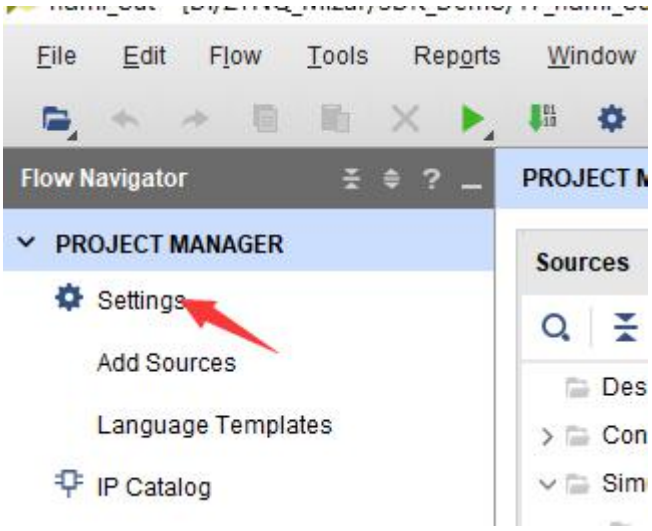
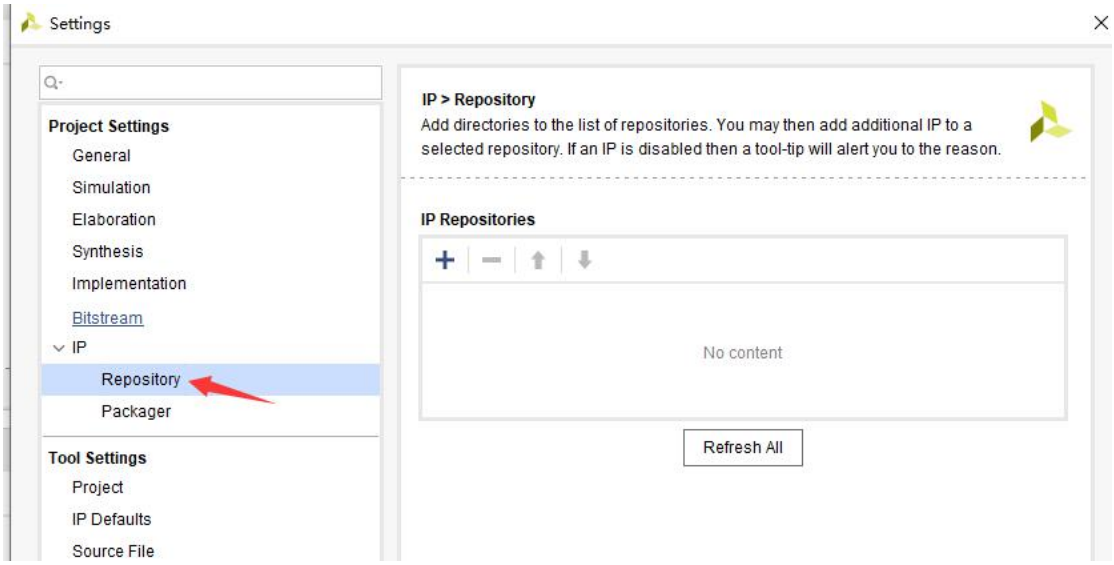


图 17.2.3 进入设置页面

点击“IP”打开下拉菜单，之后点击“Repository”，之后点击左上角的“+”，



选择工程文件夹下的“repo”文件夹。

图 17.2.4 添加自定义 IP

选择完成之后软件会自动识别文件夹下包含的 IP，点击“OK”，添加到工程中，如下图所示：

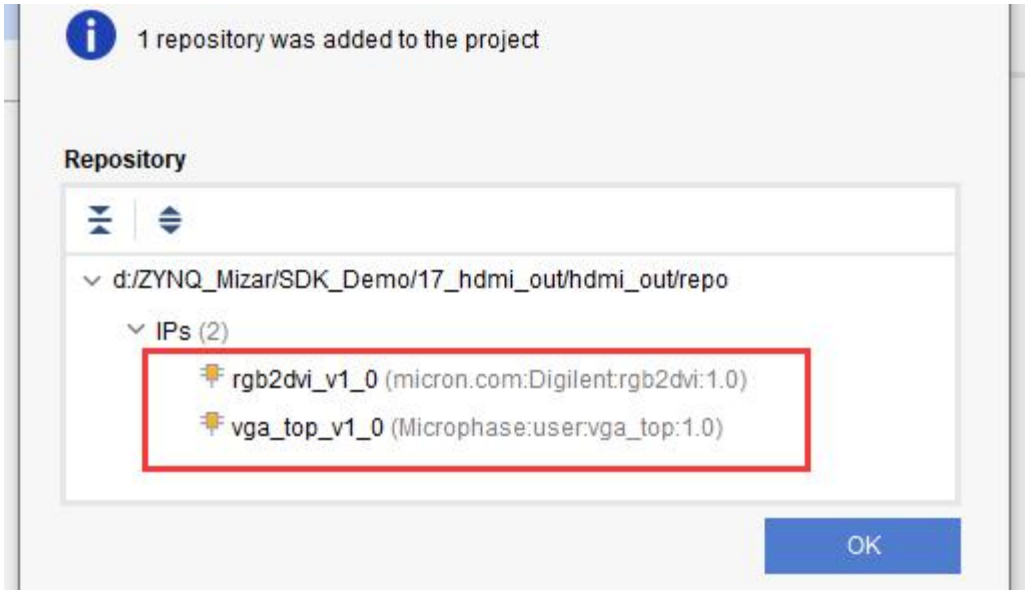
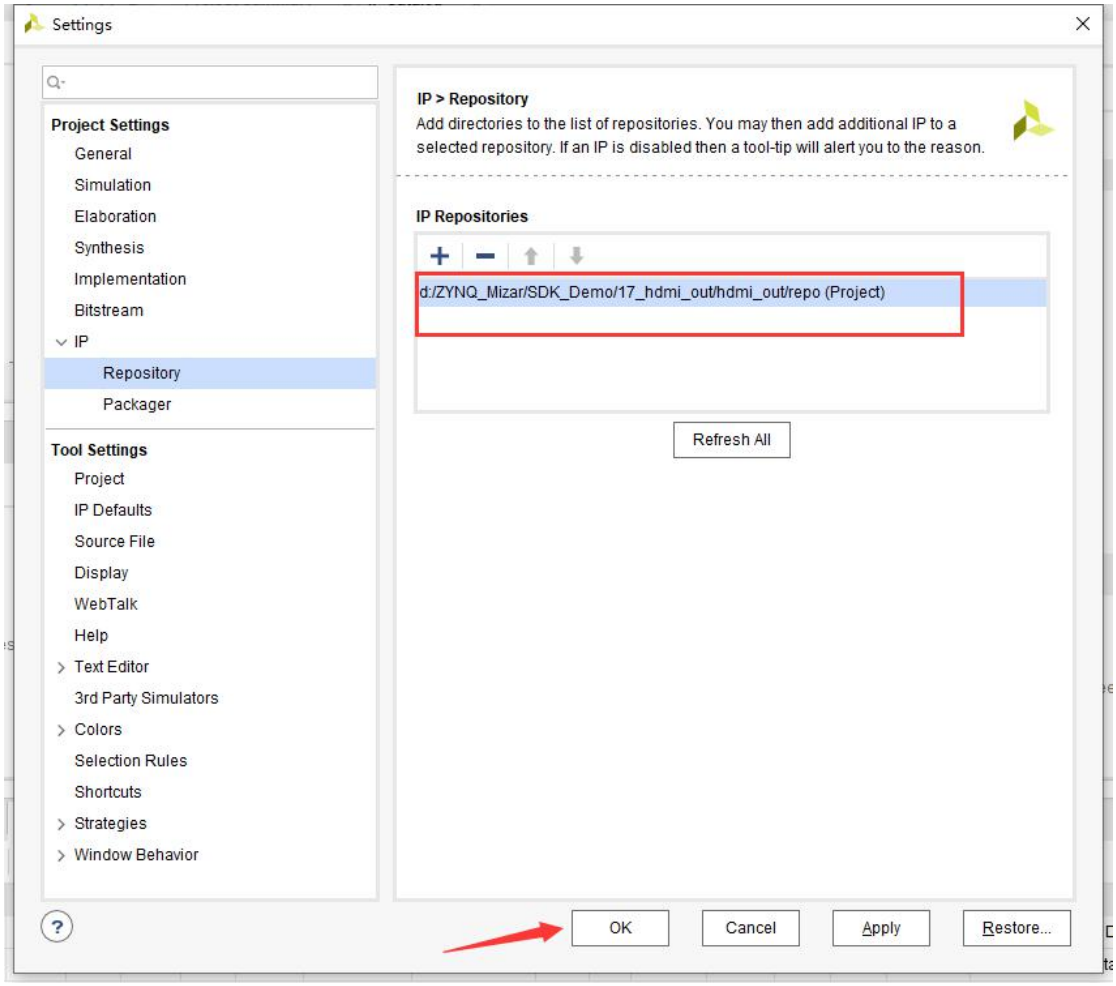


图 17.2.5 添加 IP



添加完成后如下图所示：

图 17.2.6 IP 添加完成

点击 IP Catalog 可以看到添加到工程中的 IP，如下图所示：

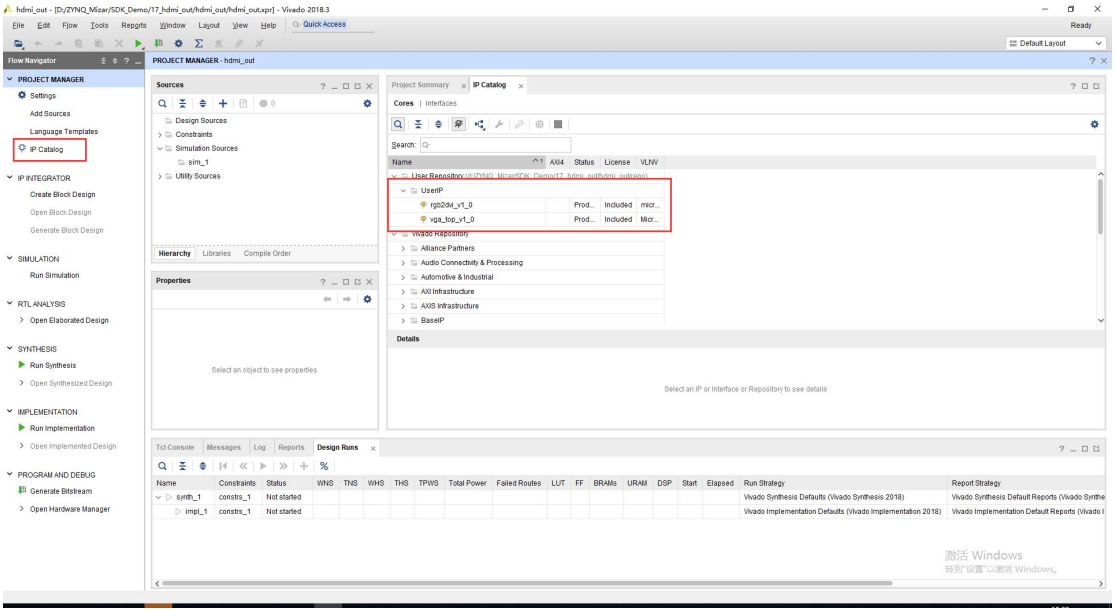


图 17.2.7 IP 添加成功

之后点击“Create Block Design”创建设计文件并命名为“hdmi_out”，创建完成后向其中添加 ZYNQ7 Processing System，如下图所示：

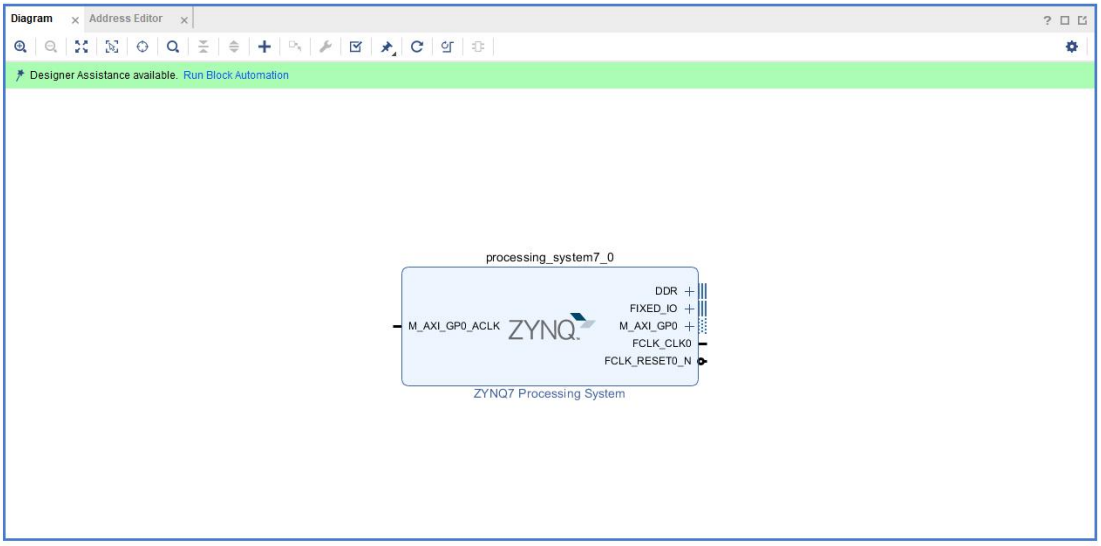


图 14.2.8 添加 ZYNQ7 Processing System

双击 ZYNQ，进行如下参数设置。

点击 PS-PL Configuration 选项， 在 HP Slave AXI interface 下勾选 S AXI

HP0 interface， 本节实验要用这个端口获取 DDR 中存储的图像数据。

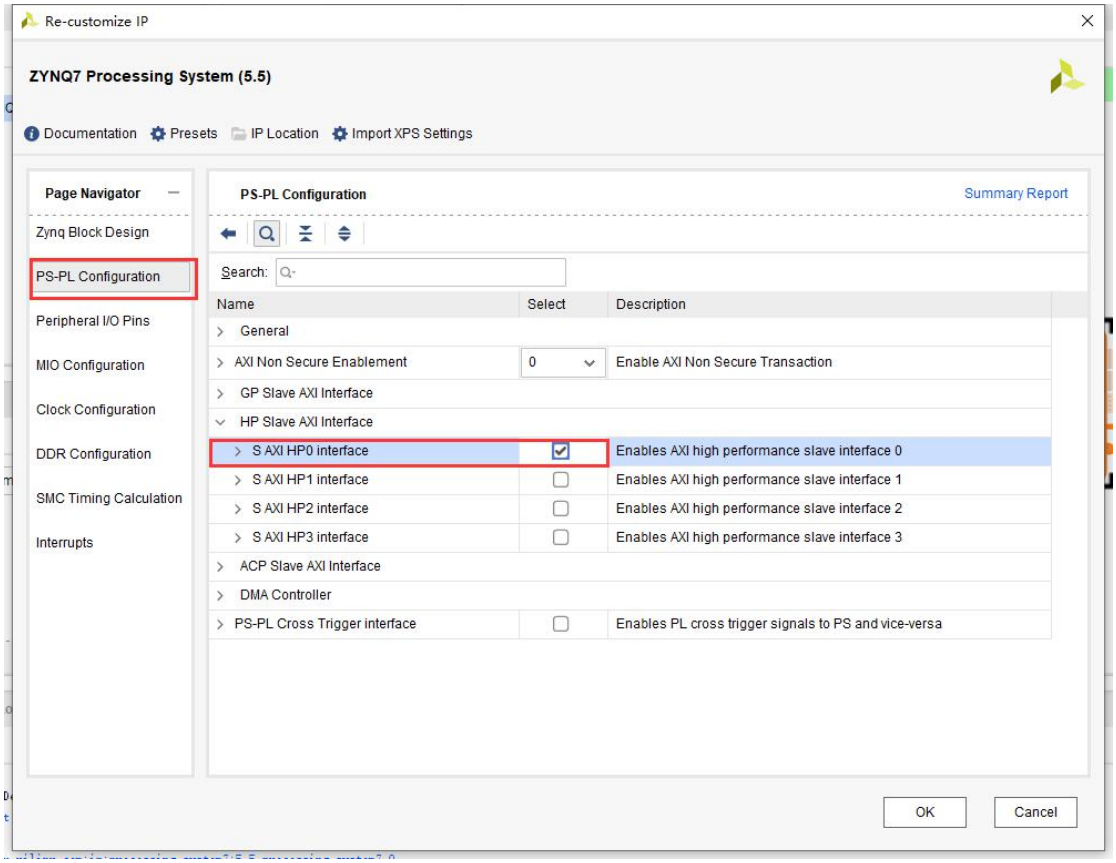
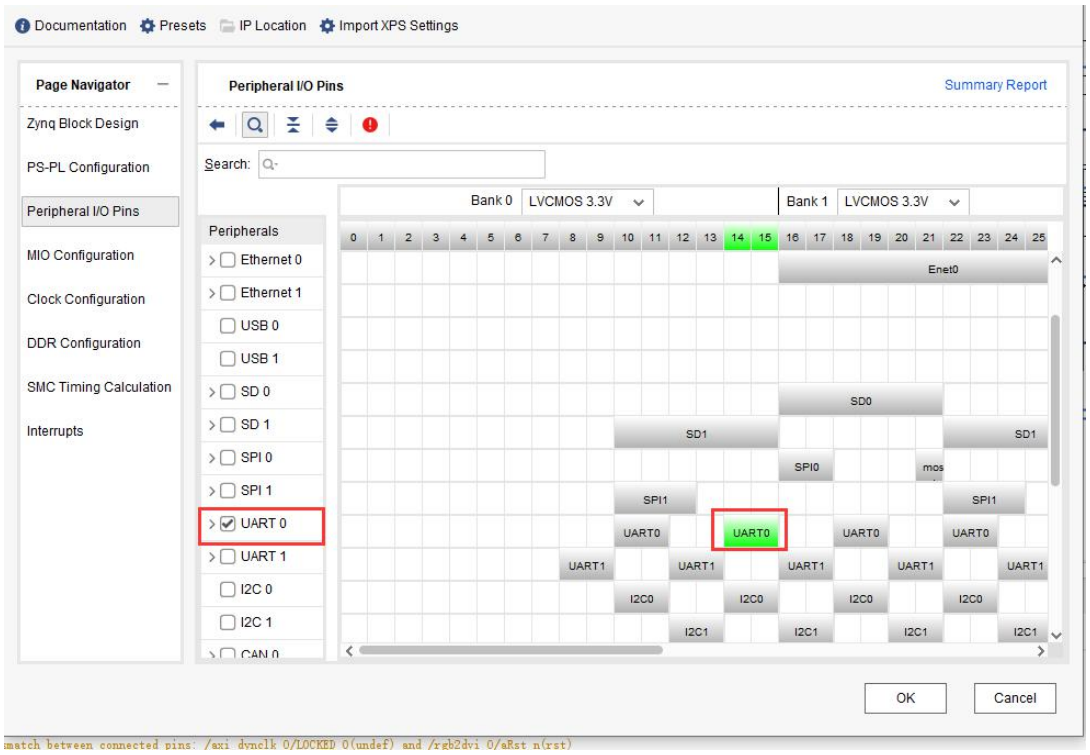


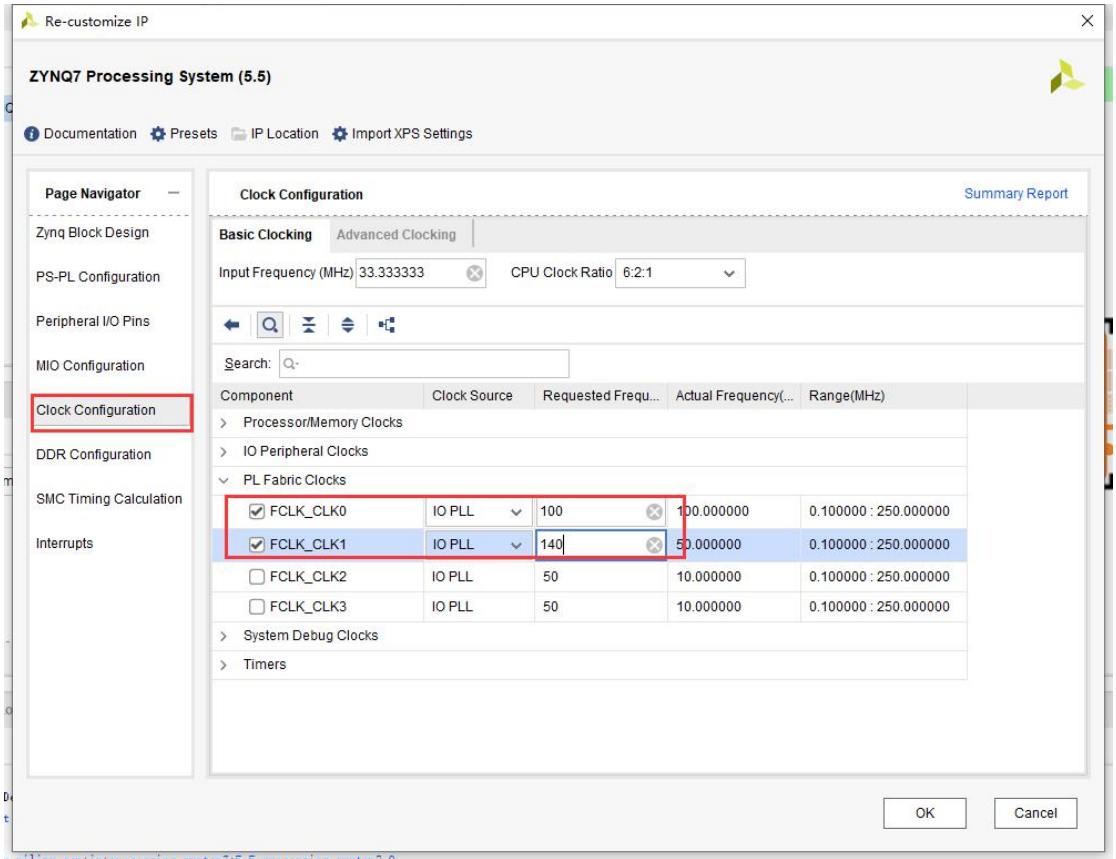
图 17.2.9 使能高速 HP 总线



使能 UART0。

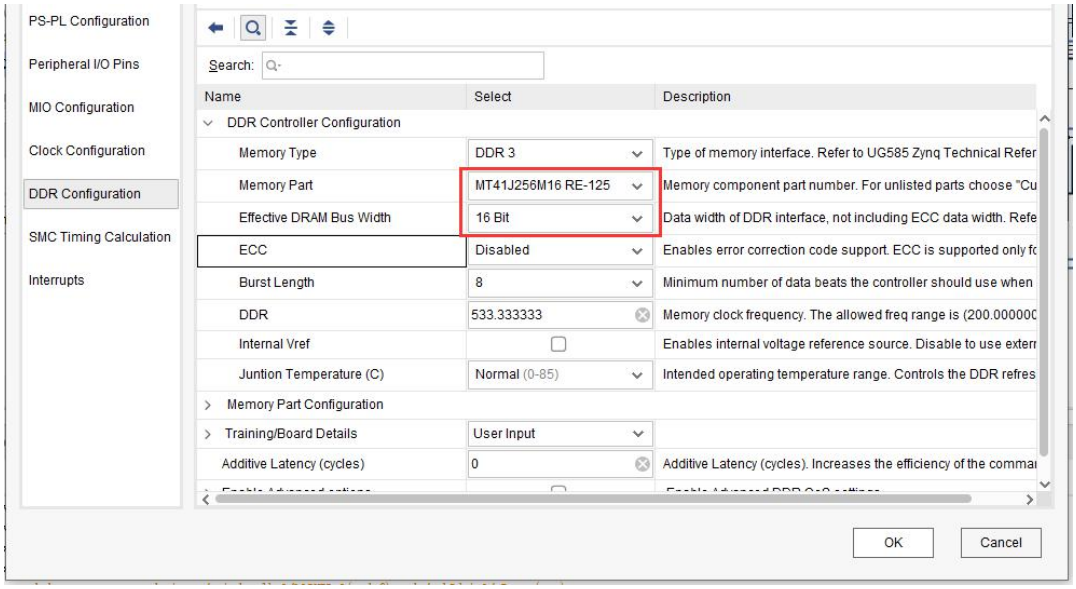
图 17.2.10 使能串口

点击 Clock Configuration 选项，在 PL Fabric Clocks 窗口，勾选 FCLK_CLK0 和 FCLK_CLK1，并将其分别设为 100， 140MHz。 其中 FCLK_CLK0 将作为 Zynq



配置各个模块的时钟， 而 FCLK_CLK1 将作为图像数据流的时钟，

图 17.2.11 设置相关时钟频率



再点击 DDR Configuration， 选择跟开发板一致的 DDR 型号

图 17.2.12 设置 DDR 型号

再点击 Interrupt 选项，激活 IRQ_F2P[15:0]选项，点击 OK，配置完成。

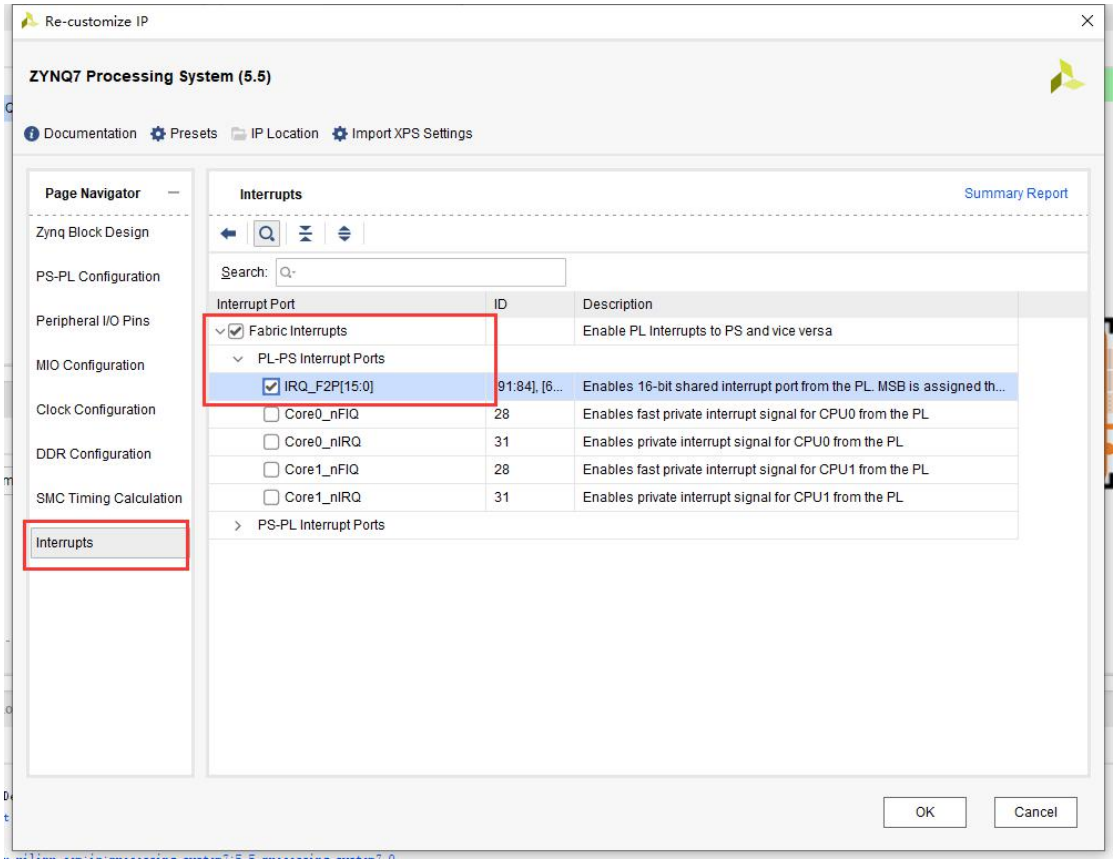


图 17.2.13 使能中断

手动连接 FCLK_CLK0 到 MAXI_GPO_ACLK，连接 FCLK_CLK1 到 S_AXI_HP0_ACLK，

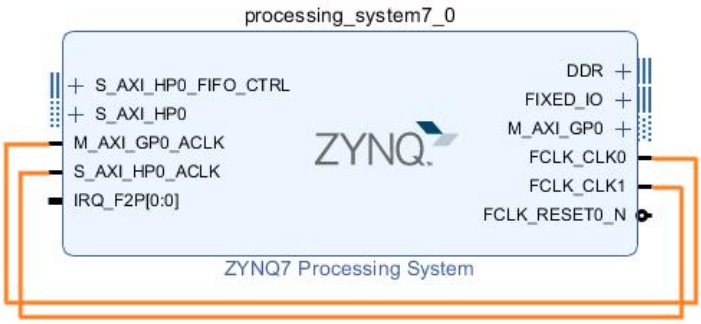


图 17.2.14 连接时钟信号

点击上方“+”打开 IP 列表，输入 VDMA 点击添加 VDMA IP。如下图所示：

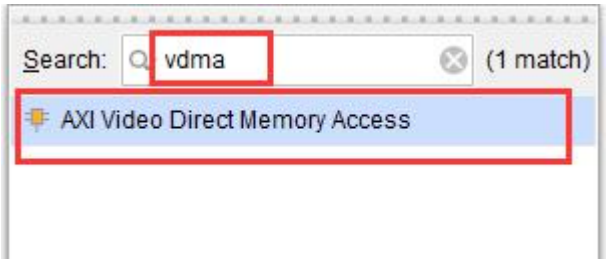


图 17.2.15 添加 IP

VDMA 添加完成如下图所示：

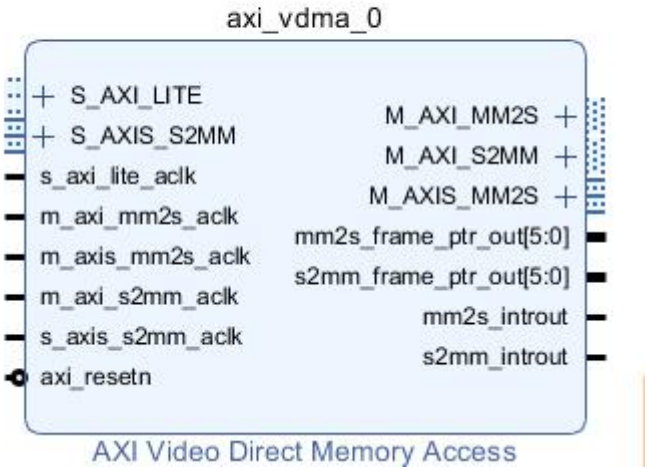


图 17.2.16 VDMA

双击上图 axi_vdma_0，进行参数配置。首先 Basic 选项，Address Width 设为 32 bits，寻址空间可以达到 4GB。Frame Buffers 设为 1，取消 Enable WriteChannel 选项，因为本实验 VDMA 只从 DDR 中读取图像数据。在 Enable Read Channel 下，将 Line Buffer Depth 改为 4096，其它默认，如下图所示：

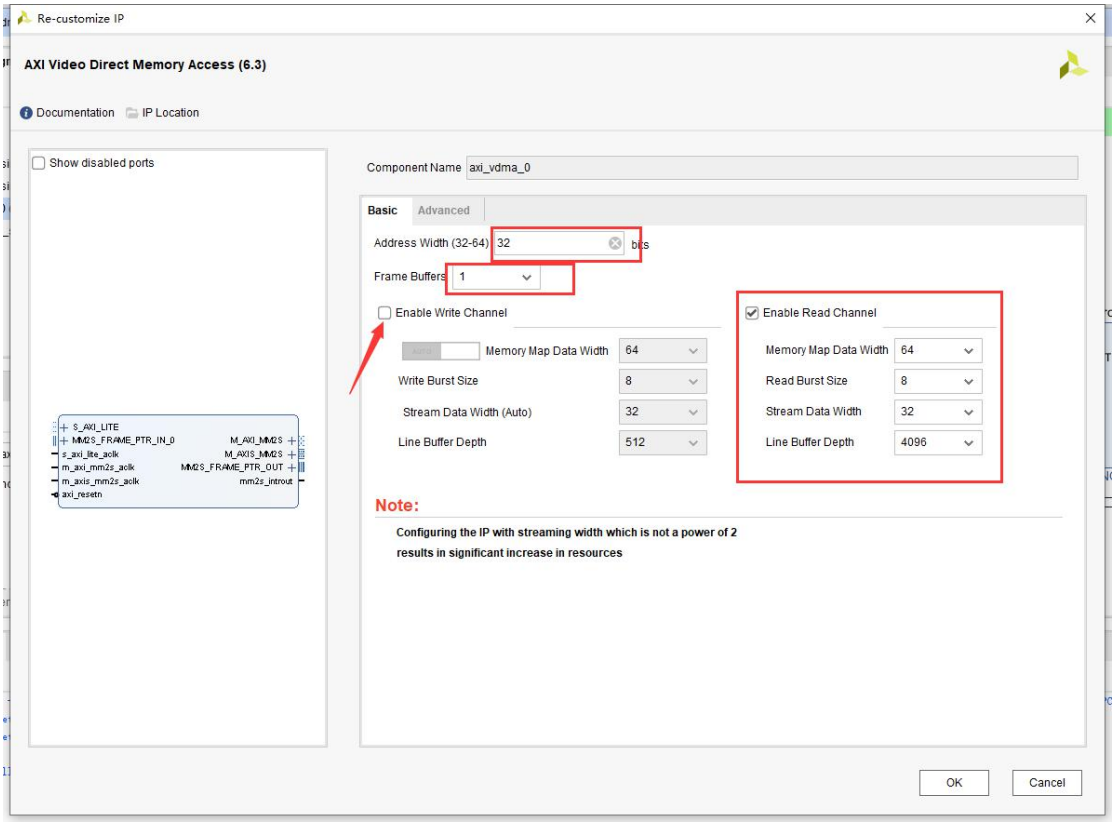
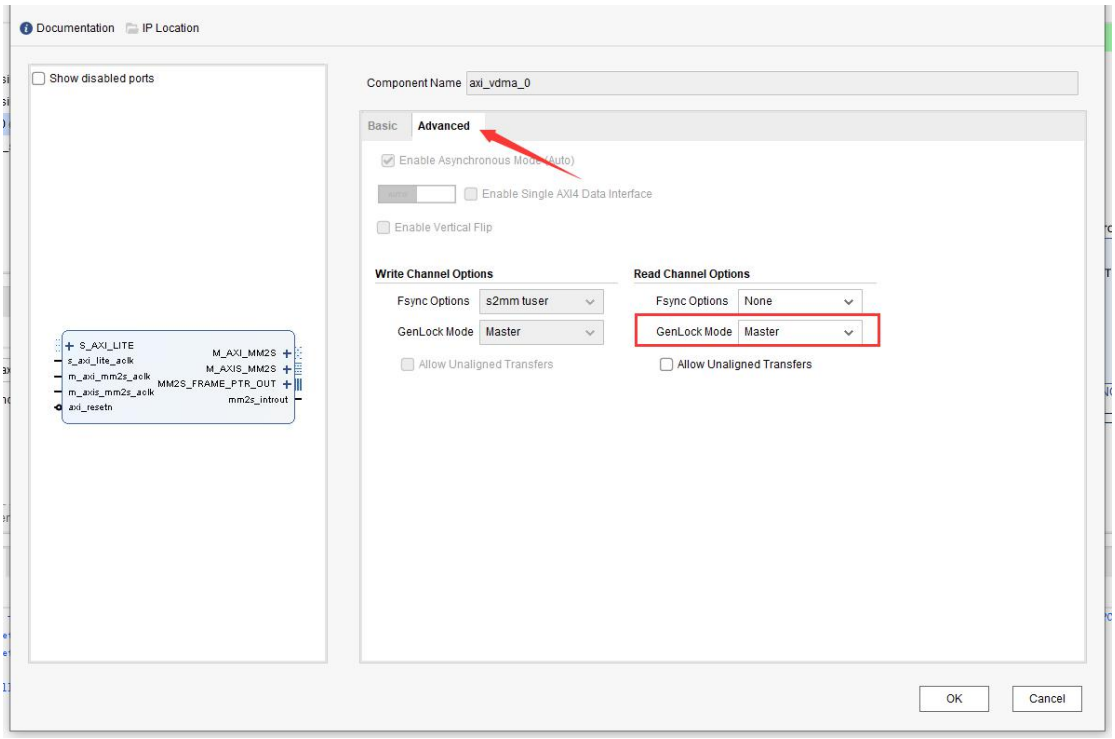


图 17.2.17 设置 Basic 参数

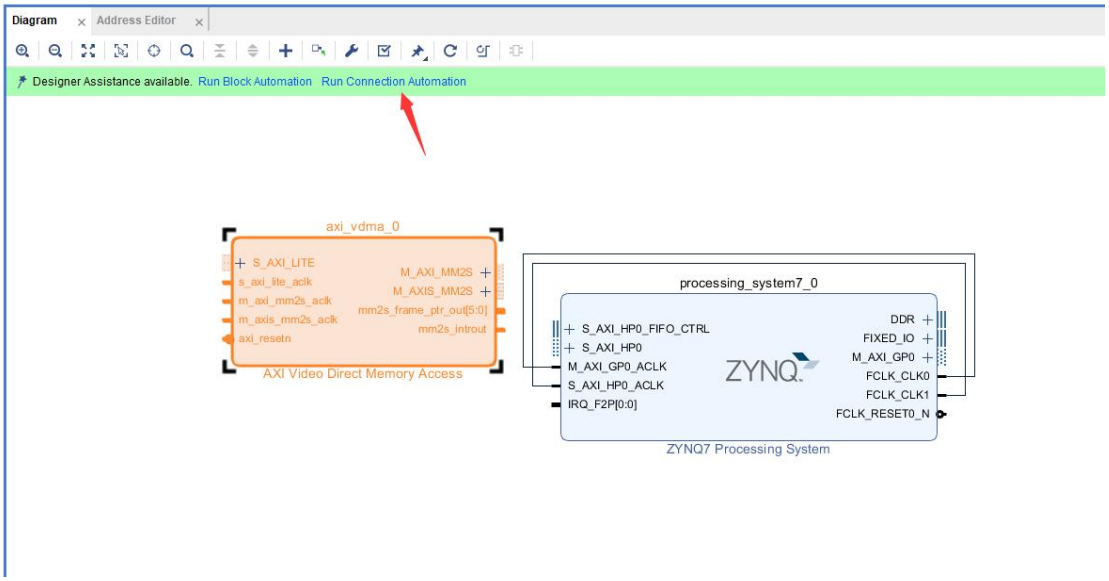
再点击 Advanced 选项卡， 将 GenLock Mode 改为 Master，点击 OK 完成



配置。

图 17.2.18 设置 Advanced

在 Diagram 窗口，点击 Run Connection Automation，自动生成 AXI 互联总



线及系统复位电路，

图 17.2.19 生成 AXI 互联总线及系统复位电路

在弹出的对话框中勾选 All Automation，点击 OK。

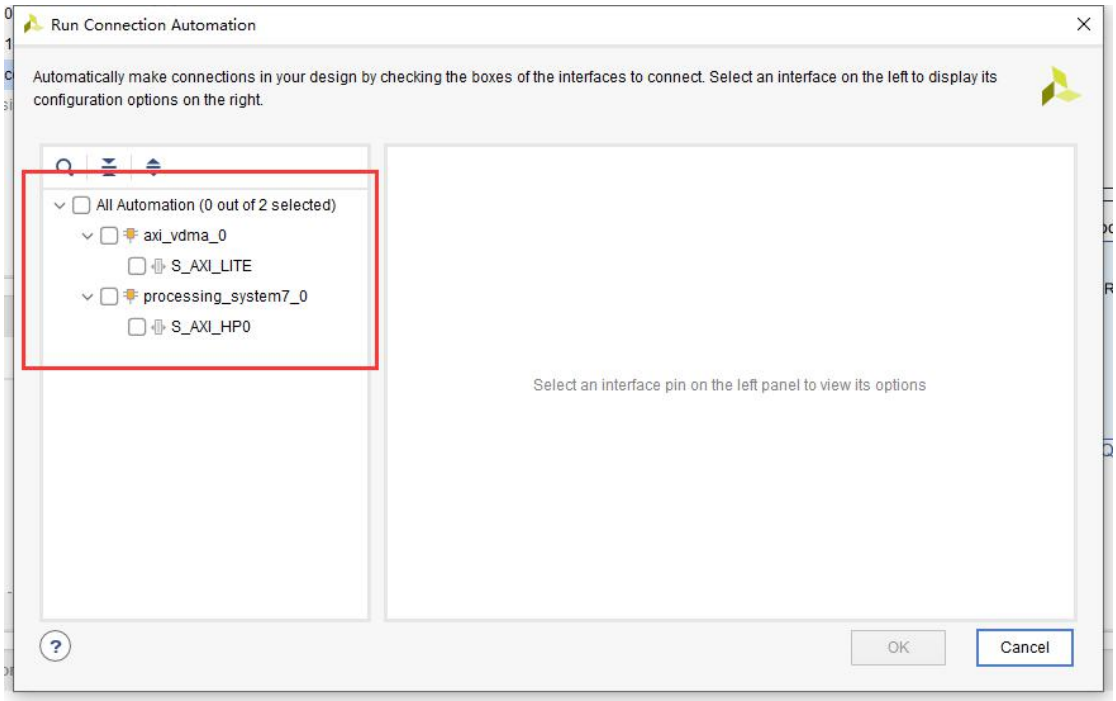


图 17.2.20 勾选 All Automation

手动将 m_axis_mm2s_aclk 和 m_axi_mm2s_aclk 两个时钟连在一起。

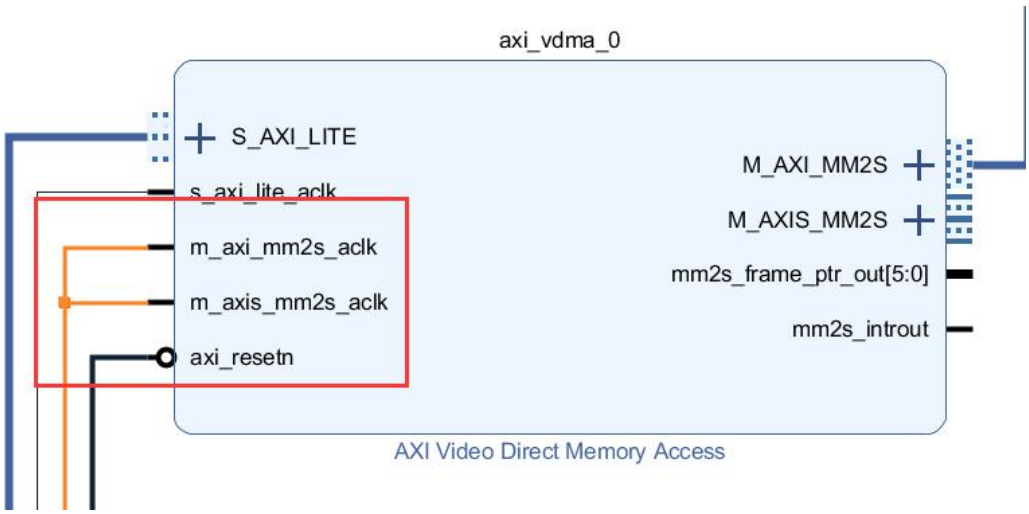
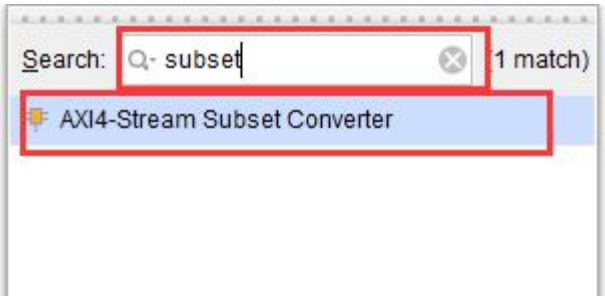


图 17.2.21 连接 m_axis_mm2s_aclk 和 m_axi_mm2s_aclk

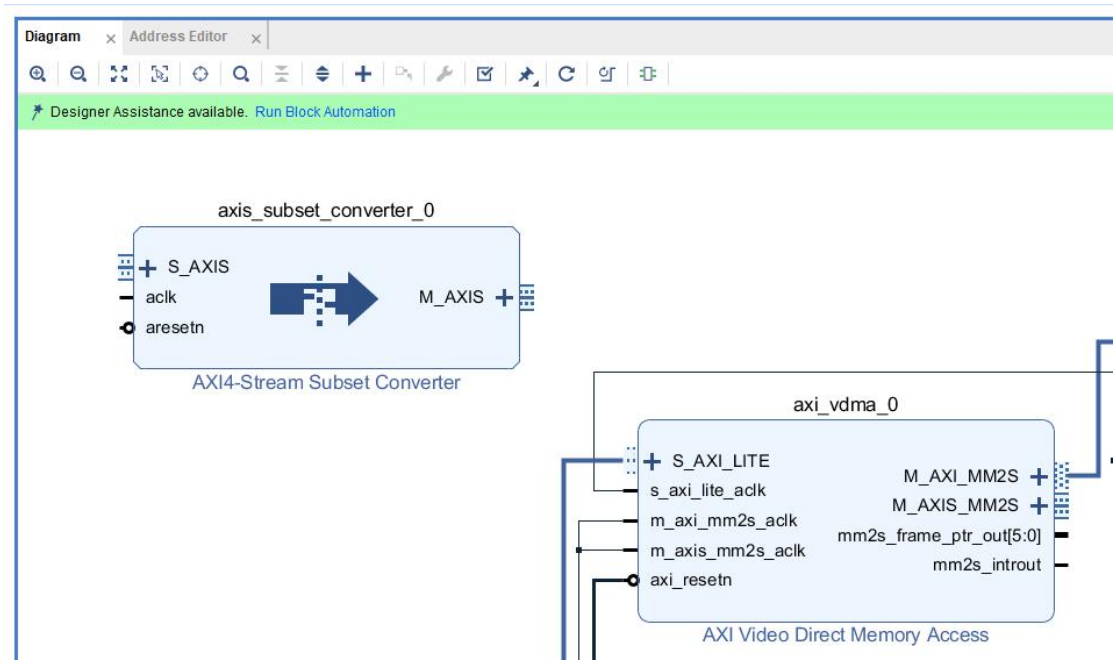


添加 AXI-Stream Subset Converter 模块

添加完成后如下图所示：

图 17.2.22 添加 AXI-Stream Subset Converter 模块

双击上图的 axis_subset_converter_0，如下配置。 TDATA 输入为 32-bit，即 4-Byte，输出为 24-Bit， 即 3-Byte。将 TKEEP、TLAST 修改为 yes，并将 USER



Width 置为 1，

这里要重点说一下 TDATA Remap String 项，因为要显示的图像在 DDR 中的存储方式为 24-bit 默认方式 B，G，R，分别以[7:0]，[15:8]，[23:16]组成 24-bit，而本例中使用的产生 TMDS 信号模块 rgb2dvi 的像素排序为 R，B，G，所以需要在 TDATA Remap String 这一项做顺序调整。如下图所示：

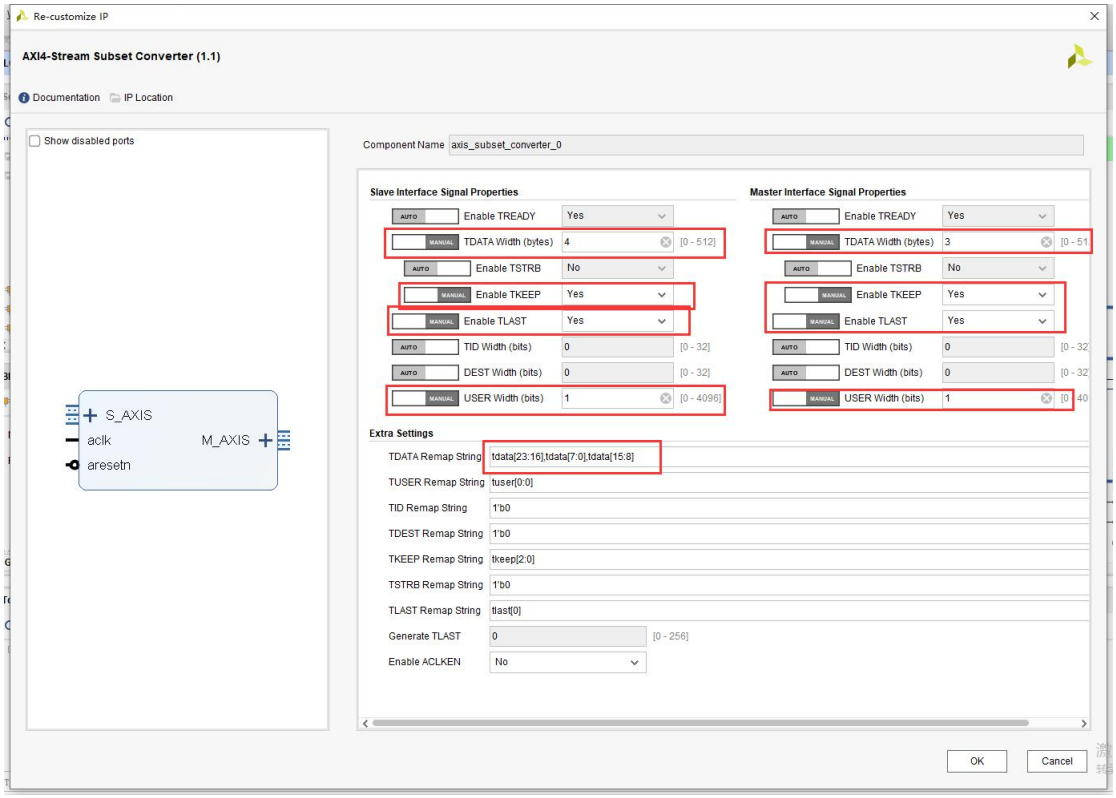


图 17.2.23 设置 subset 相关设置

手动连接 axis_subset_converter_0 的 S_AXIS 端口到 axi_vdma_0 的 M_AXIS_MM2S 端口。并将其 aclk 连接到 FCLK_CLK1 时钟网络。 如下图所示：

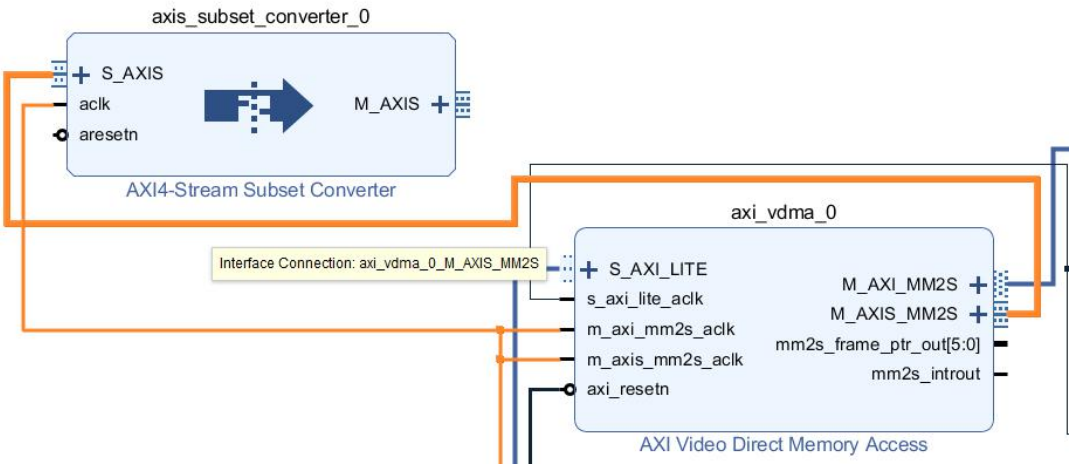
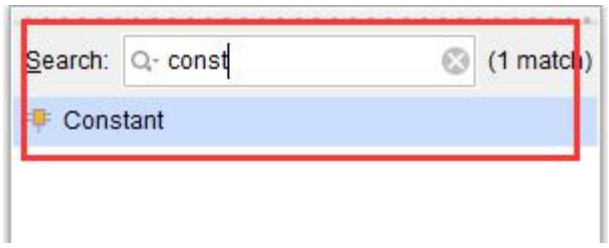


图 17.2.24 连接信号

增加一个常量输出模块 Constant， 将 axis_subset_convert_0 的复位端口 aresetn 拉高，使其始终处于工作状态，



添加后如下图所示：

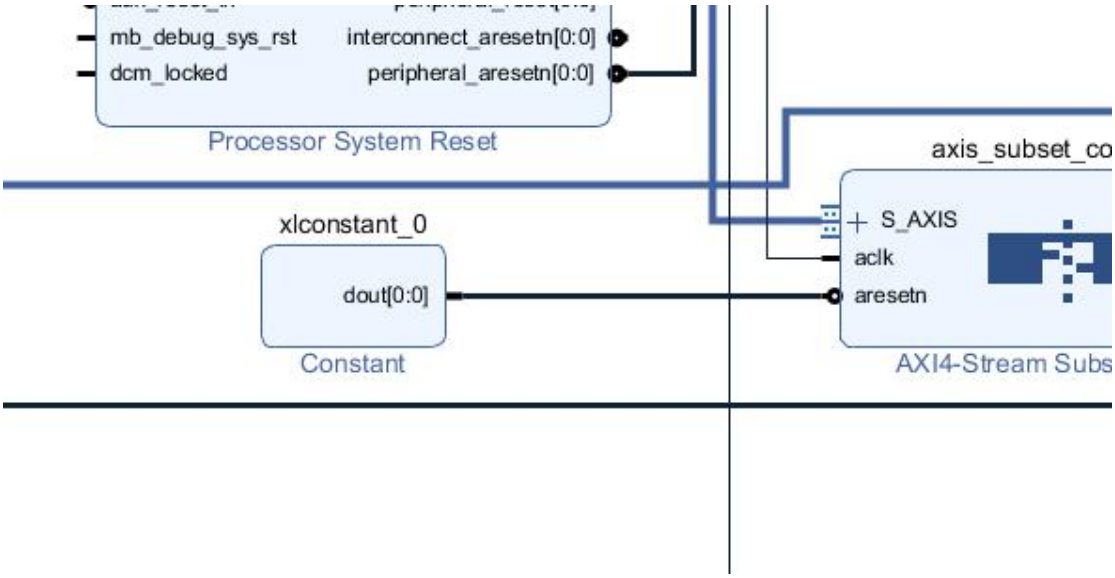
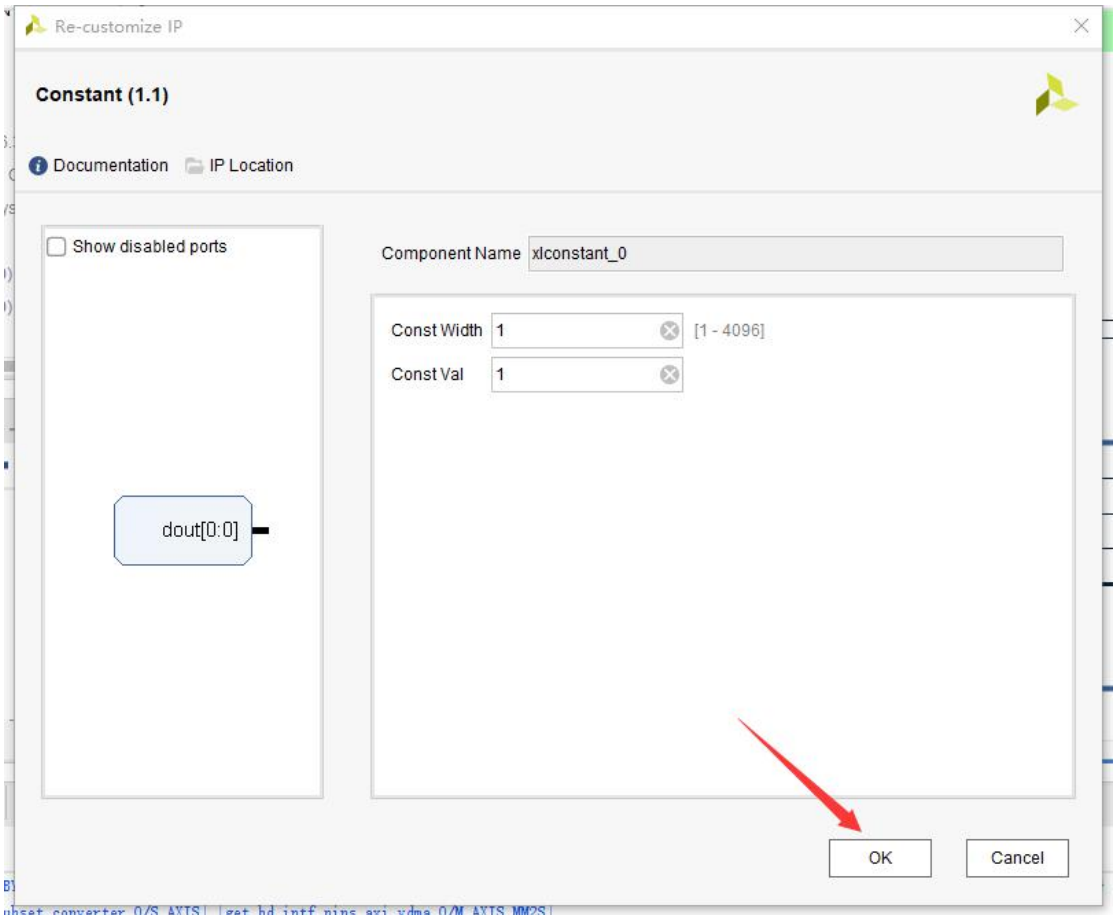


图 17.2.25 添加 Constant

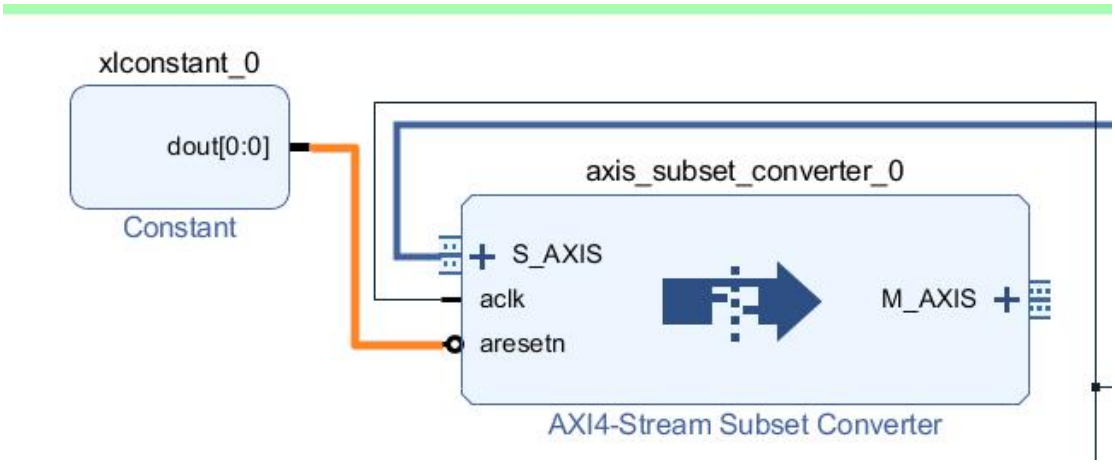
双击上图的 xlconstant_0，保持默认的 1bit 位宽输出高电平 1 的配置，点



击 OK，

图 17.2.26 设置默认输出高

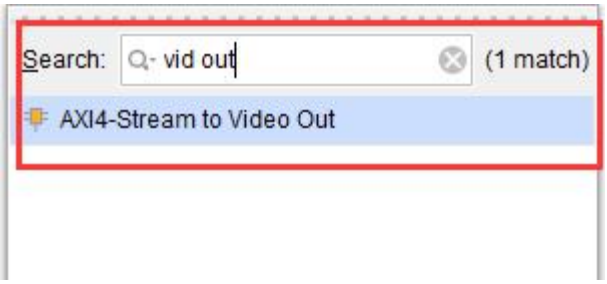
将 xlconstant_0 的 dout 端口连到 axis_subset_converter_0 的 aresetn



端口，

图 17.2.27 连接信号

添加 AXI4-Stream to Video Out 模块。本例用这个模块将 VDMA 从 DDR 读



出的 AXI4-Stream 转换成 RGB 图像数据。

双击 v_axi4s_vid_out_0 进行配置。将 FIFO Depth 改为 4096，Clock Mode 设为 Independent，Timing Mode 设为 Master，点击 OK。

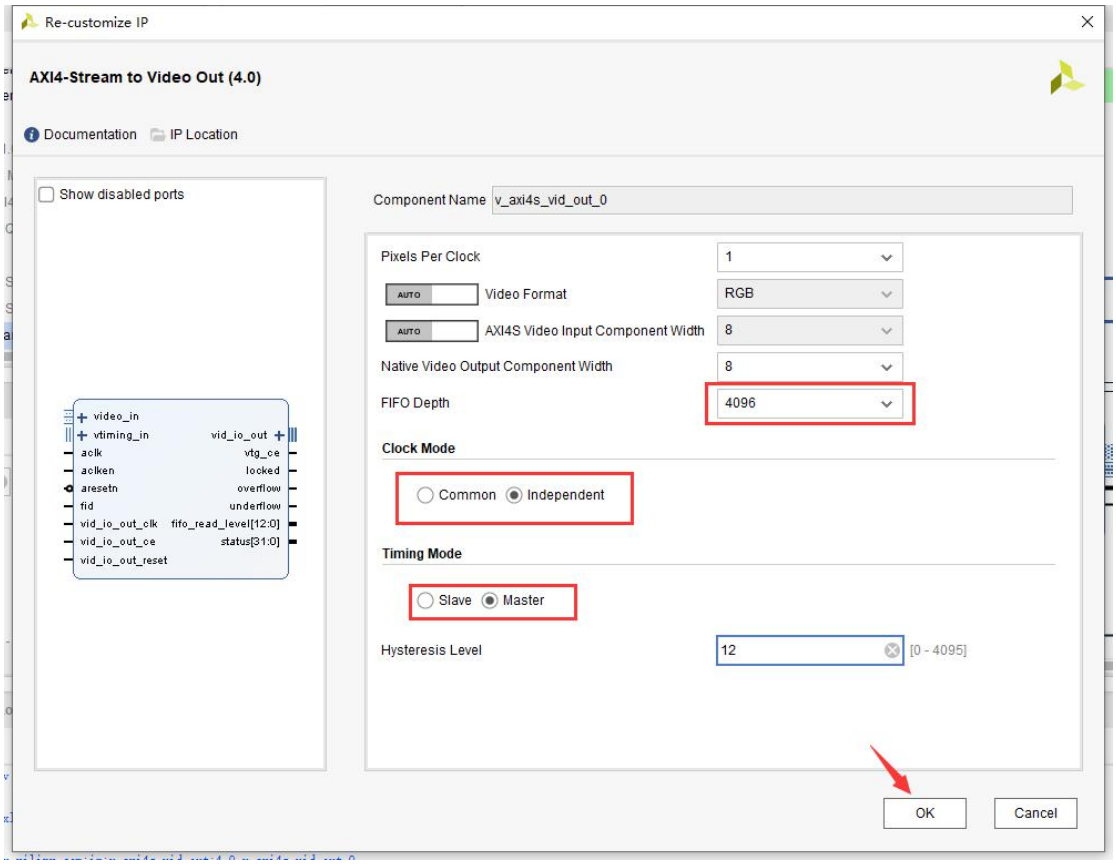


图 17.2.28 设置相关设置

手动将 v_axi4s_vid_out_0 的 video_in 端口连接到 axis_subset_converter_0 的 M_AXIS 端口，再将其 aclk 连接到 FCLK_CLK1 时钟网络，如下图，

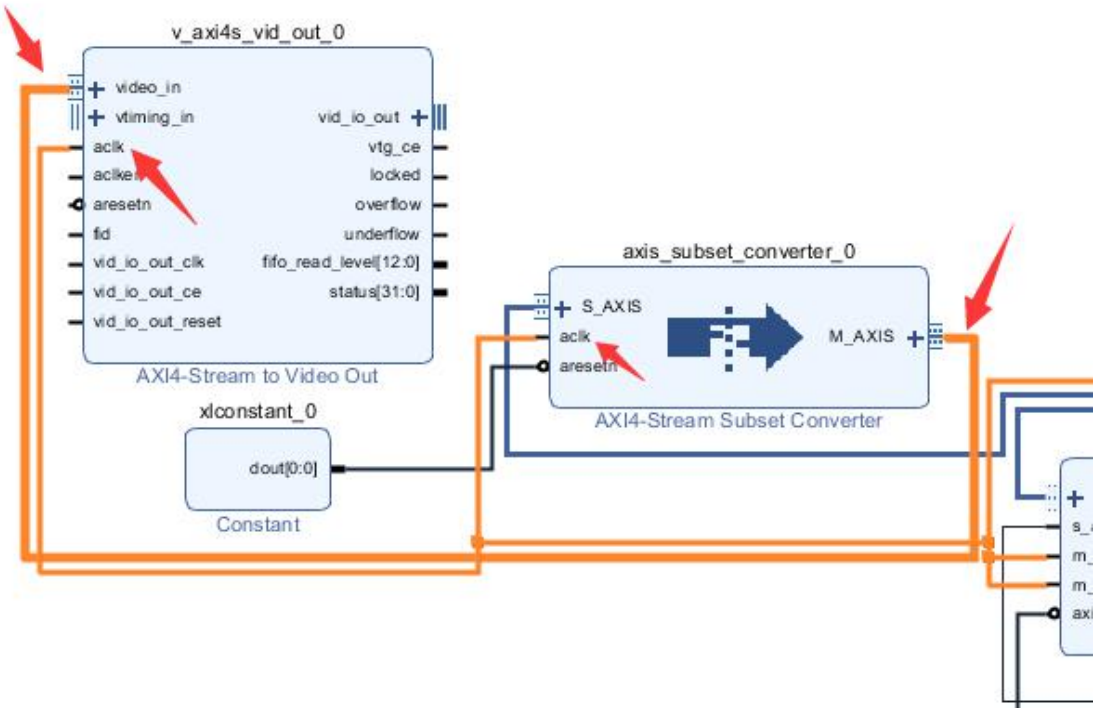
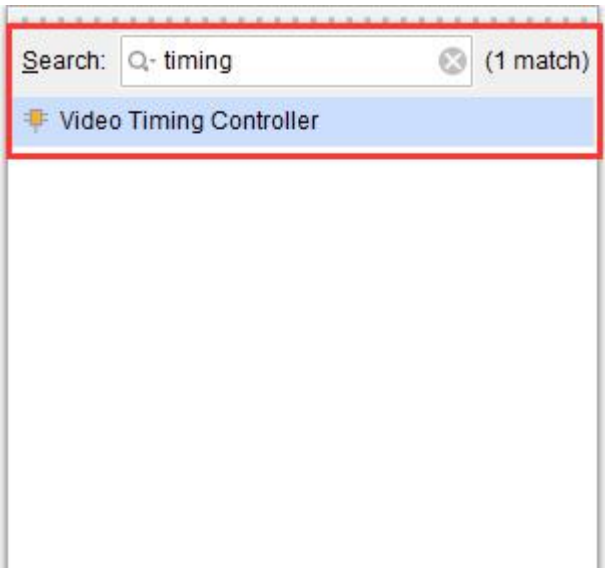


图 17.2.29 连接相关信号

添加 Video Timing Controller 模块。本例使用这个模块，来产生不同的分辨率下时序控制信号。



添加完成后如下图所示：

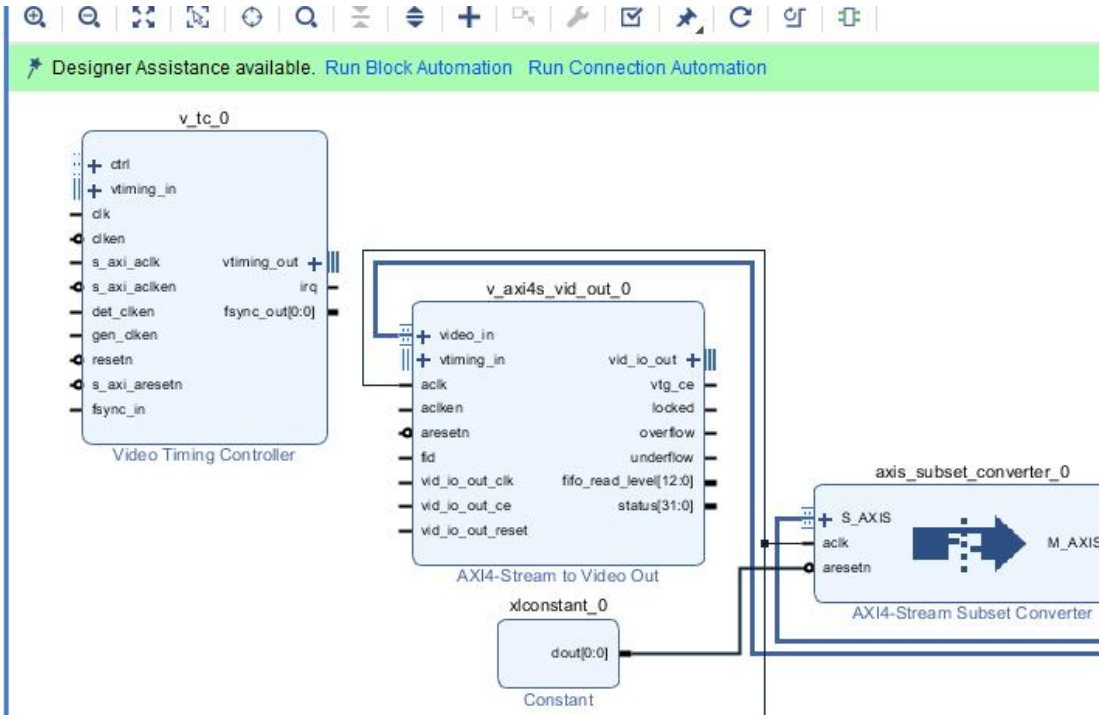


图 17.2.30 Video Timing Controller 模块

双击上图的 v_tc_0 进行配置。取消掉 Enable Detection 选项的勾选，其它保持默认，点击 OK，

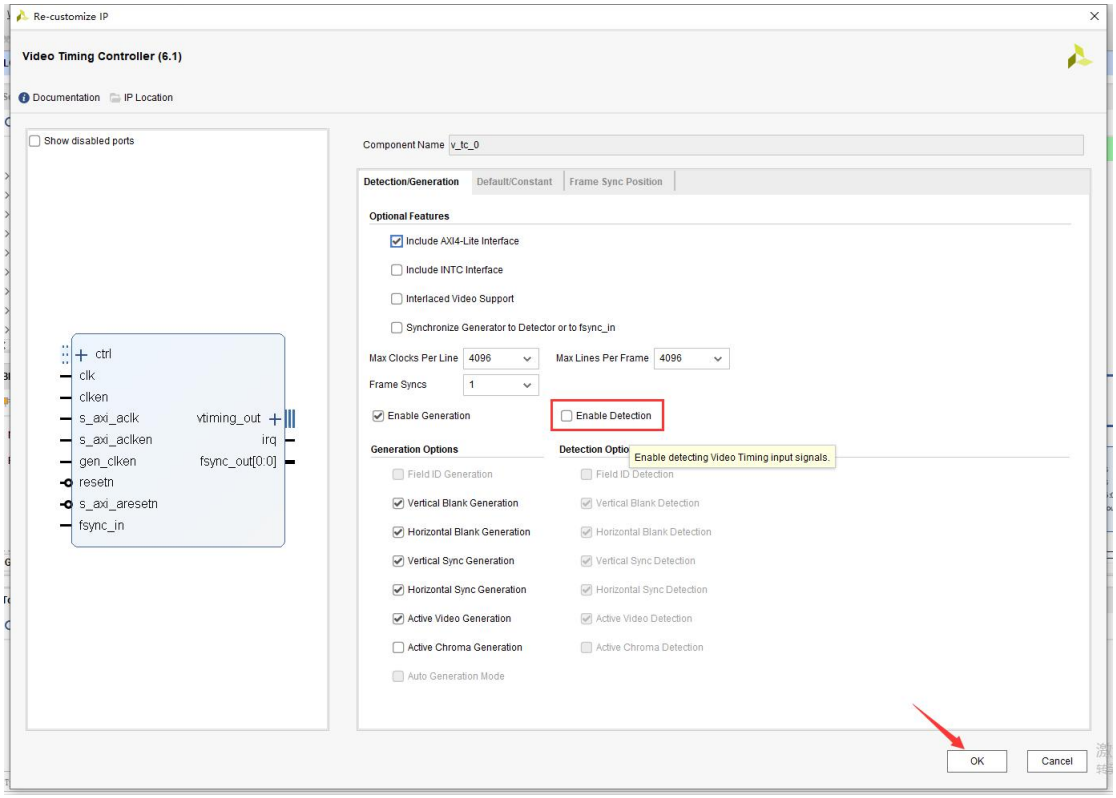
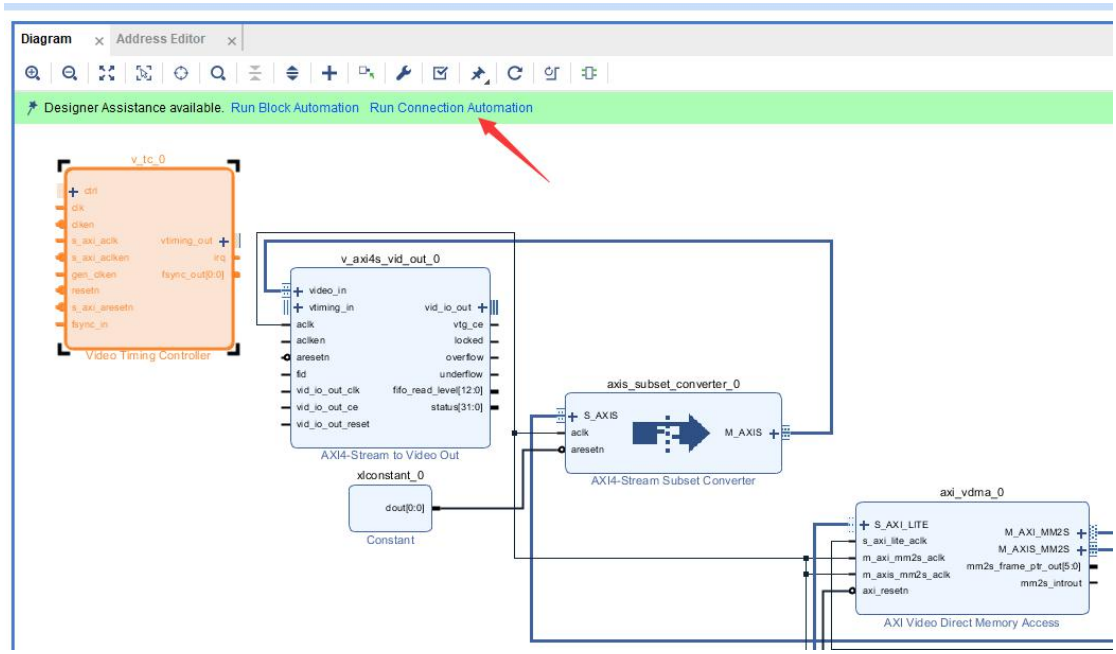


图 17.2.31 取消 Enable Detection 的勾选

点击 Diagram 窗口的 Run Connection Automation，自动生成 v_tc_0 模块



的相关连接。

勾选 All Automation，点击“OK”。

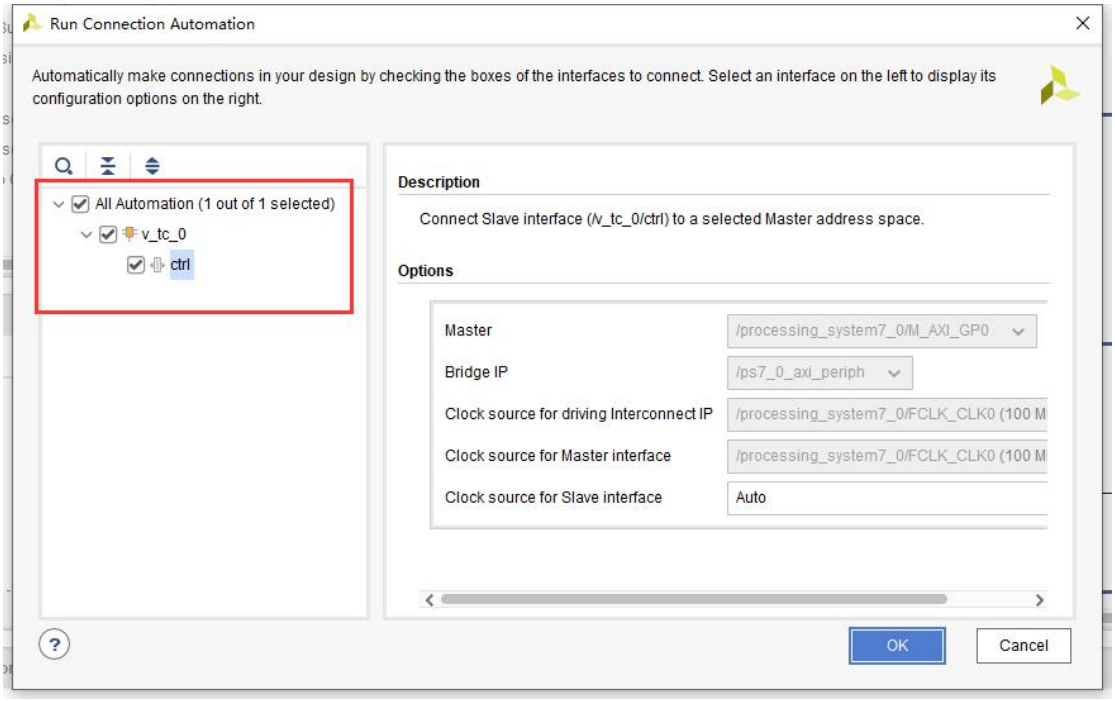
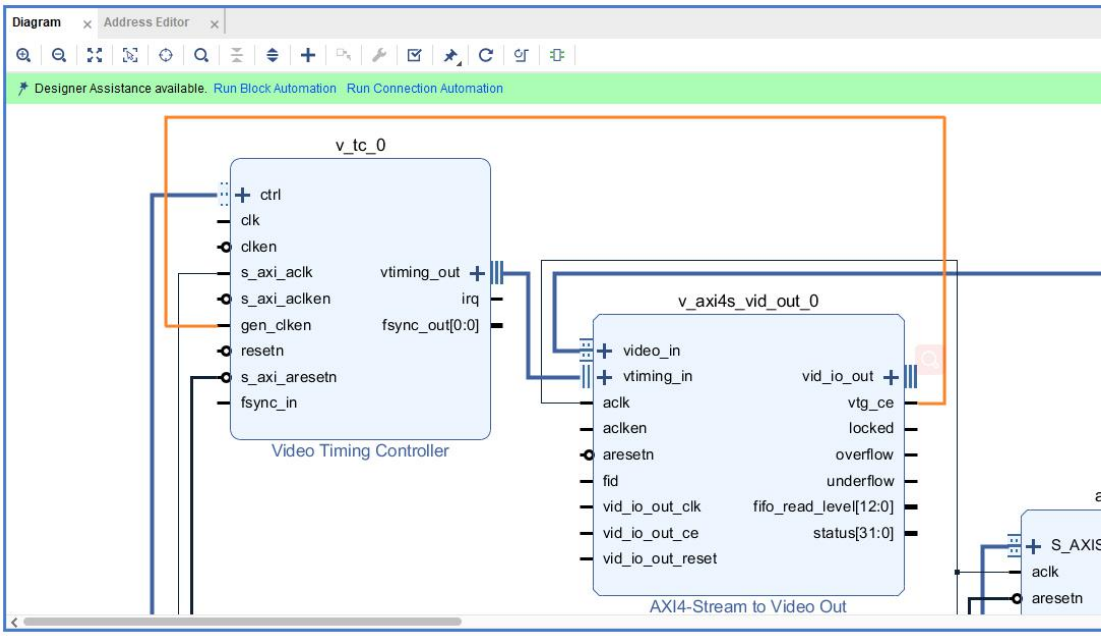


图 17.2.32 生成相关连接

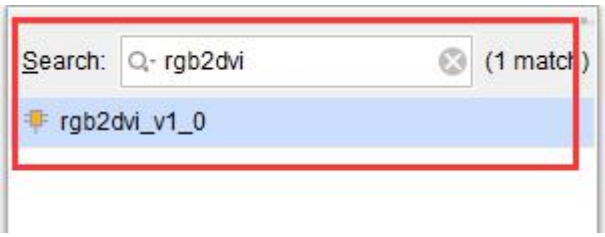
连接 v_tc_0 的 vtiming_out 端口到 v_axi4s_vid_out_0 的 vtiming_in 端



口，连接 v_tc_0 的 gen_clken 端口到 v_axi4s_vid_out_0 的 vtg_ce 端口，

图 17.2.33 连接相关信号

添加 rgb2dvi 模块。此模块将 RGB 信号转换为 DVI 信号，即 TMDS 格式。



添加完成后如下图所示：

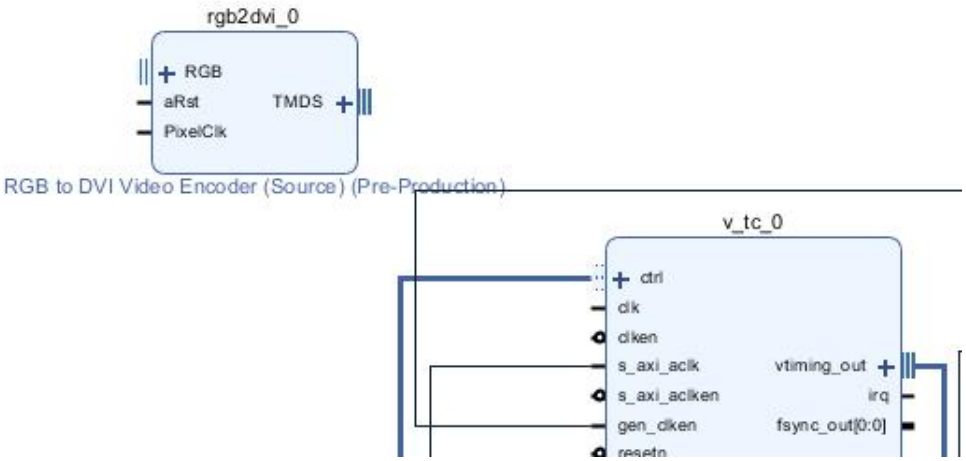


图 17.2.34 rgb2dvi 模块

双击 rgb2dvi_0 进行配置。 这里取消勾选 Reset active high 和 Generate SerialClk internally from pixel clock，信号频率设置选择<120MHz(720p)这一项。

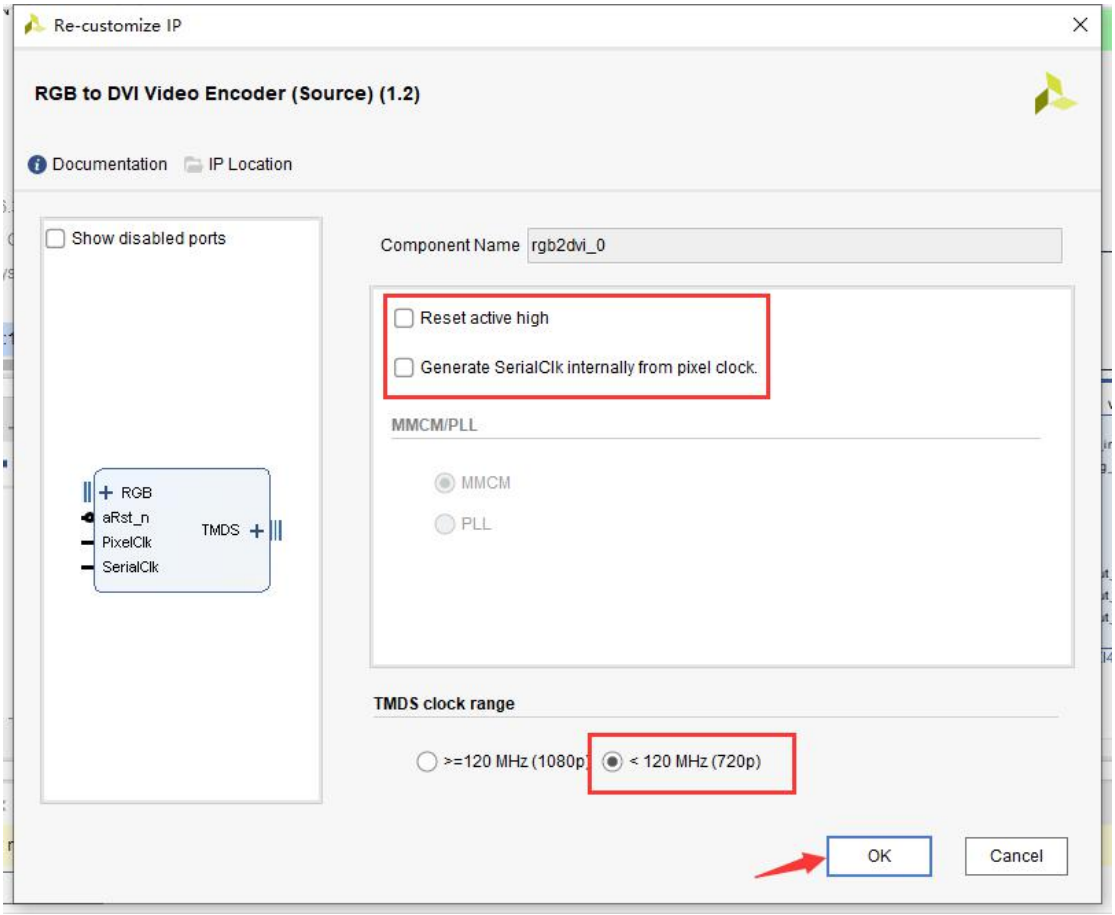


图 17.2.35 rgb2dvi_0 模块相关设置

连接 v_axi4s_vid_out_0 的 vid_io_out 端口到 rgb2dvi_0 端口，

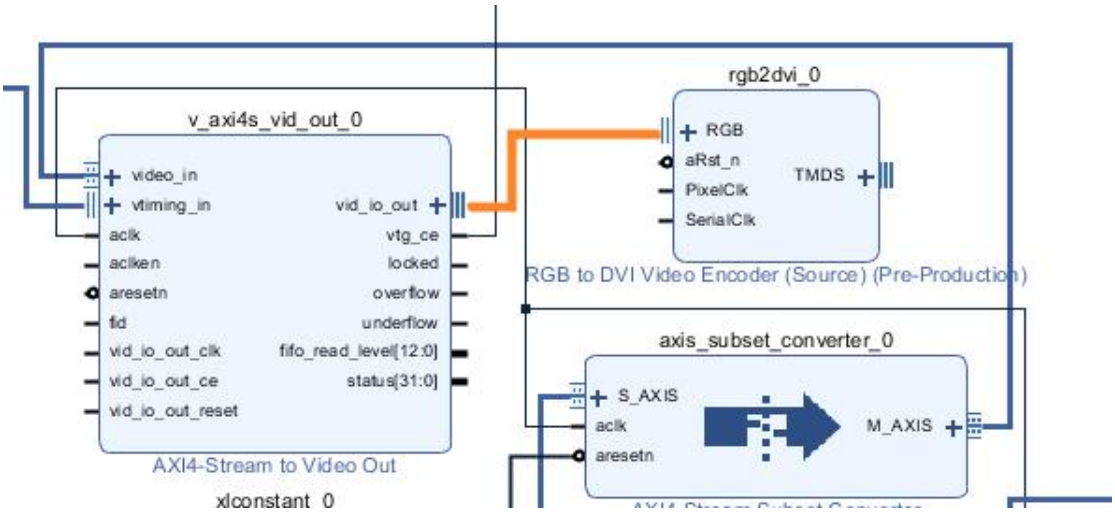
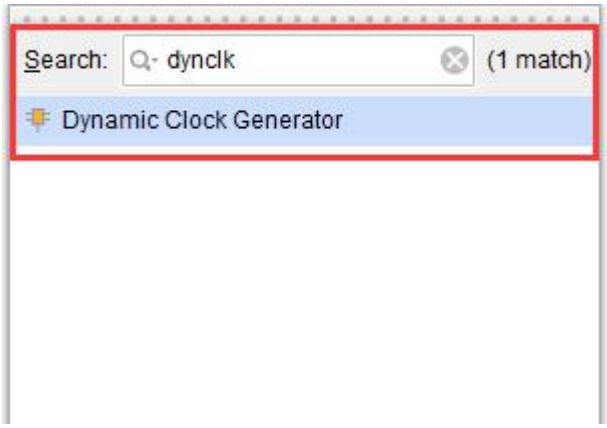


图 17.2.36 连接相关信号

添加时钟生成模块，点击上方工具栏“+”，在搜索框输入 dynclk，回车添加



模块。

添加完成后如下图所示，此模块使用默认参数，点击 Run Connection Automation

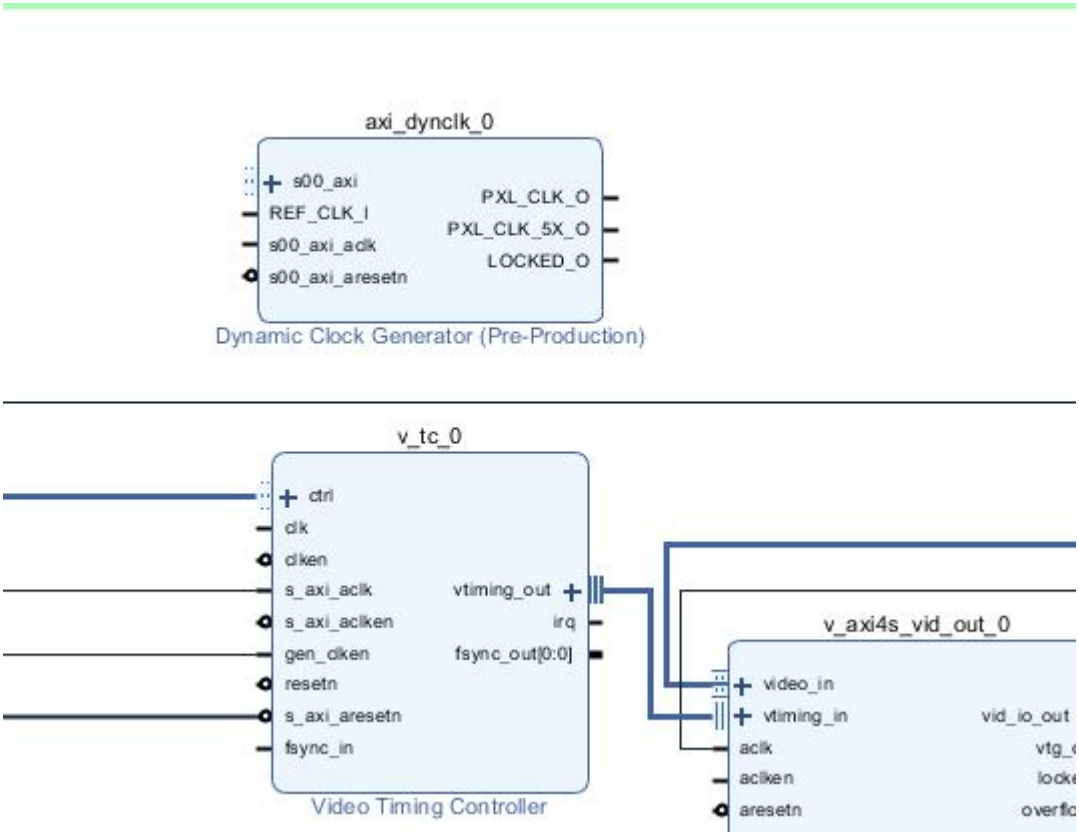
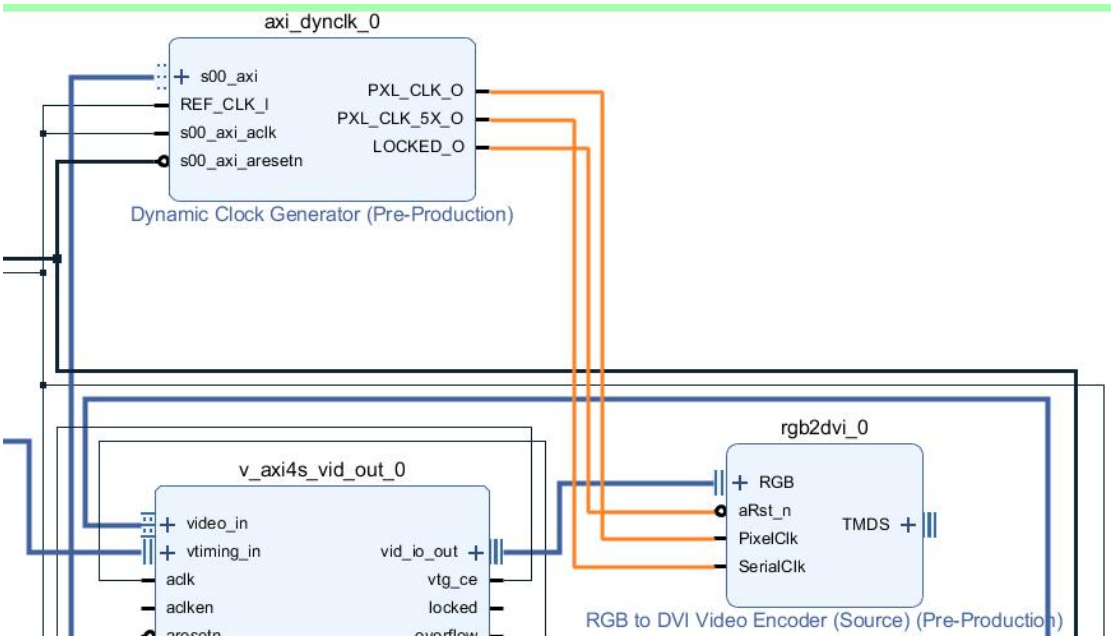


图 17.2.37 时钟生成模块

将 axi_dynclk_0 的 REF_CLK_I 连接到 FCLK_CLK0 时钟网络，将 axi_dynclk_0 的

像素时钟 PXL_CLK_O 端口连接到 rgb2dvi_0 的 PixelClk 端口， 并将其串行时钟 PXL_CLK_5X_O 端口连接到 SerialCLk 端口，再将 axi_dynclk_0 的 LOCKED_O 端口连接到 rgb2dvi_0 的 aRst_n 端口，这样当锁相环锁定，即时钟信号稳定输出时，rgb2dvi 模块开始工作.



将 v_tc_0 的 clk 端口连接到 axi_dynclk_0 的 PXL_CLK_O 端口上。同样，将 v_axi4s_vid_out_0 的 vid_io_out_clk 也连接到 PXL_CLK_O 这一网络上，

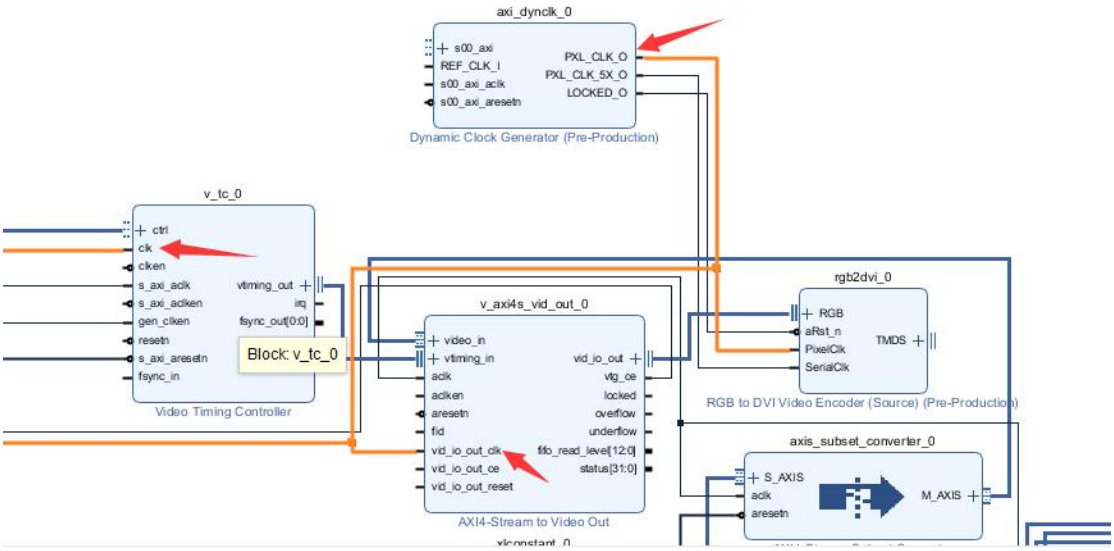
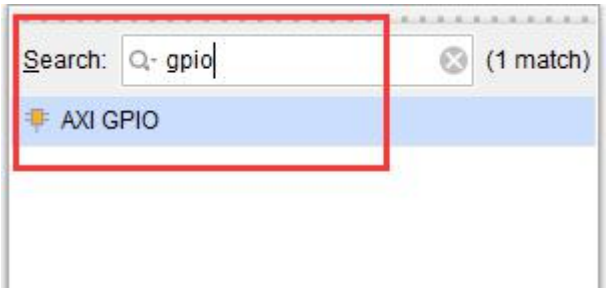


图 17.2.38 连接相关信号

添加 GPIO 模块，用作 HDMI 的热拔插检测信号 HPD



双击 axi_gpio_0 进行配置，将其设为 1 位输入， 使能 Interrupt，点击 OK，

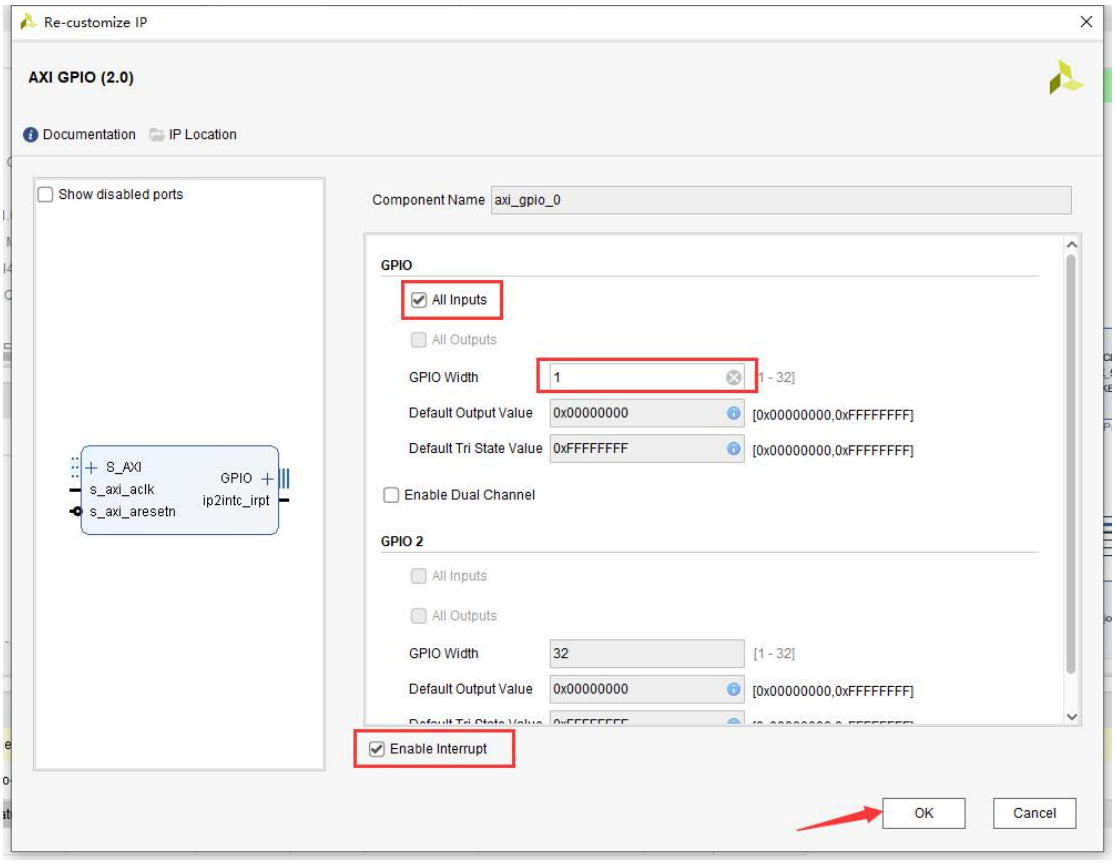


图 17.2.39 设置 GPIO 参数

点击 Run Connection Automation，将 axi_gpio_0 连入系统，

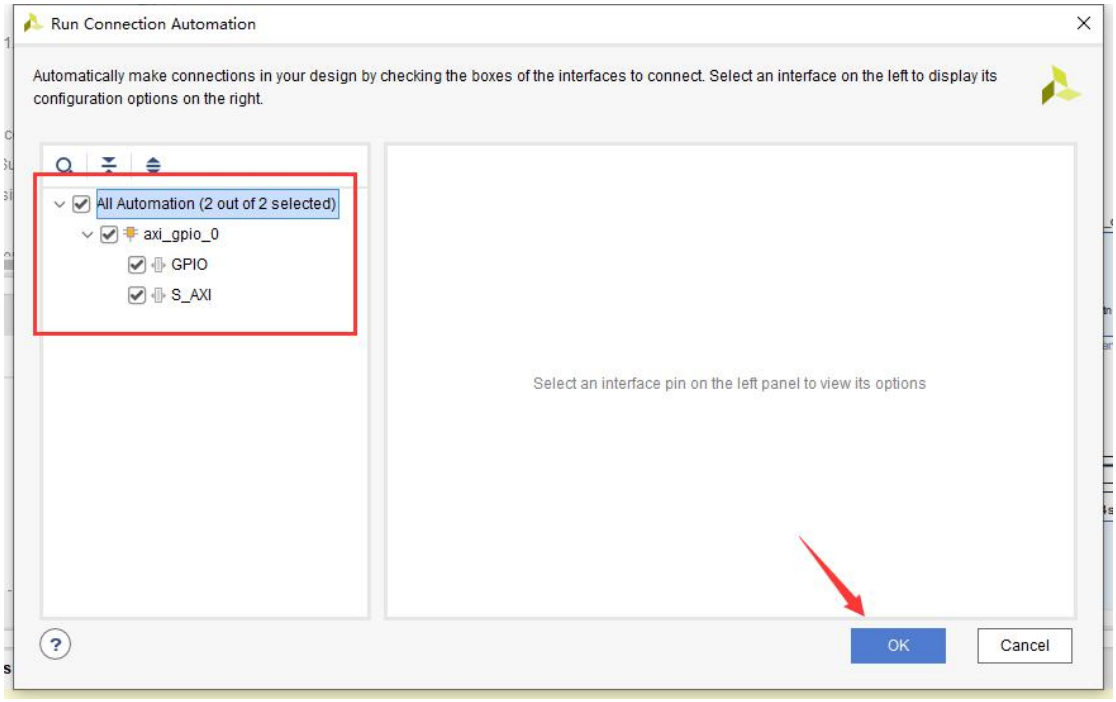
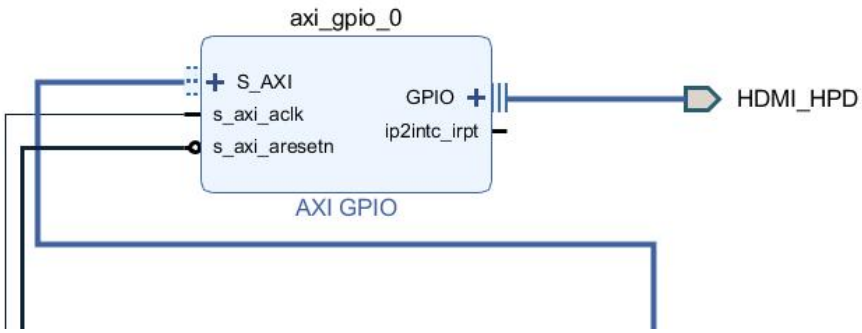


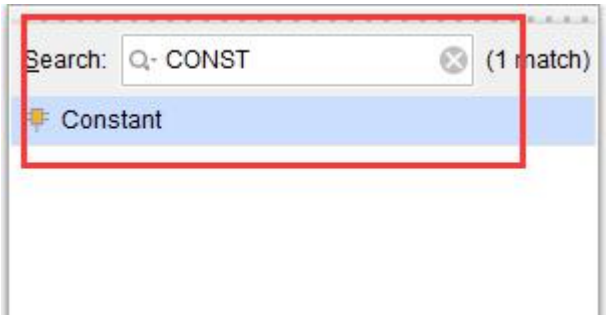
图 17.2.40 连接信号



将端口名称修改为“HDMI_HPD”，如下图所示：

图 17.2.41 修改信号名称

添加一个常量输出模块，作为 HDMI 输出使能信号，



双击 xlconstant_1 进行配置，使用默认的 1 位高电平即可，点击 OK，

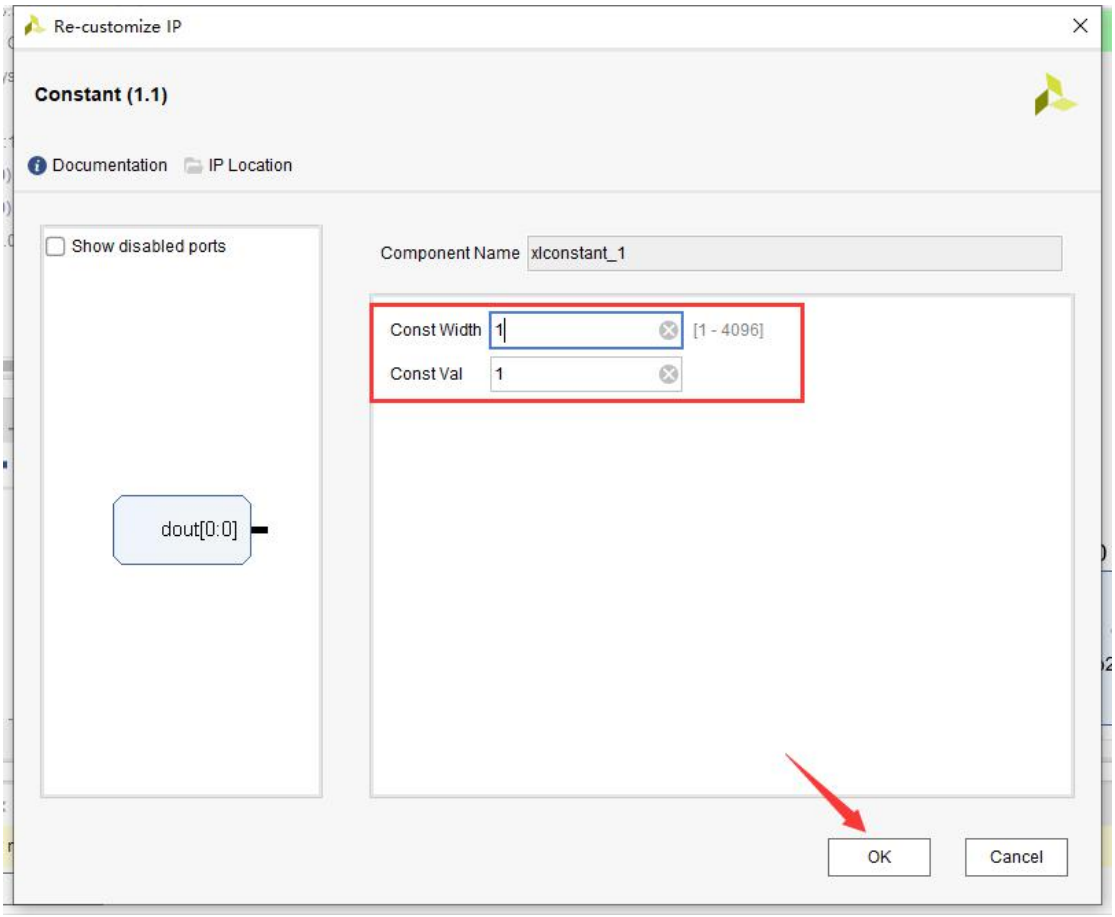
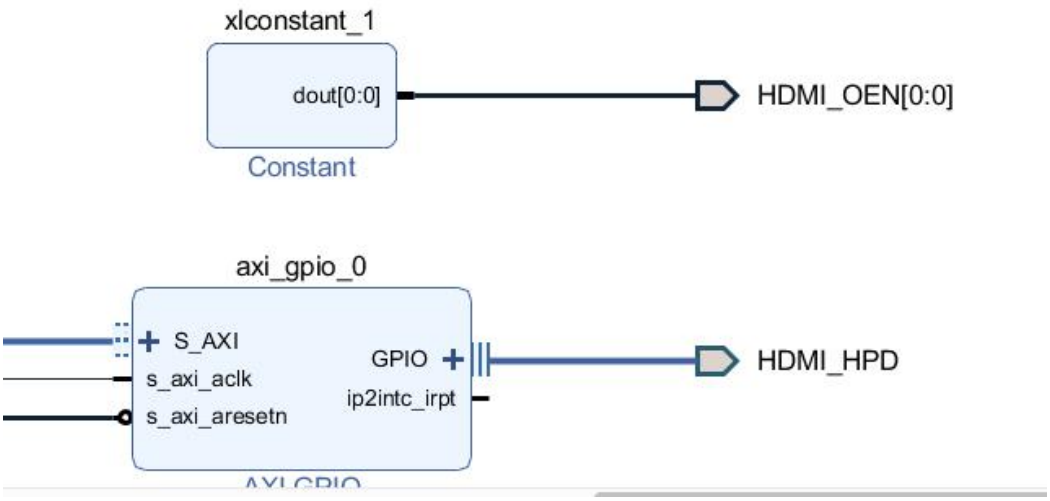


图 17.2.42 设置 xlconstant_1

右键单击 xlconstant_1 的 dout[0:0]端口， 选择 Make External 将其引出，



并重命名为 HDMI_OEN， 如下图所示：

图 17.2.43 引出信号并修改名称

添加一个 concat 模块，将所有中断信号集中起来，然后再连接到 Zynq 处理器的中断输入端口。

双击 xlconcat_0 进行配置。本例中有 3 个中断信号，设置端口数为 3， 点击 OK，如下图所示：

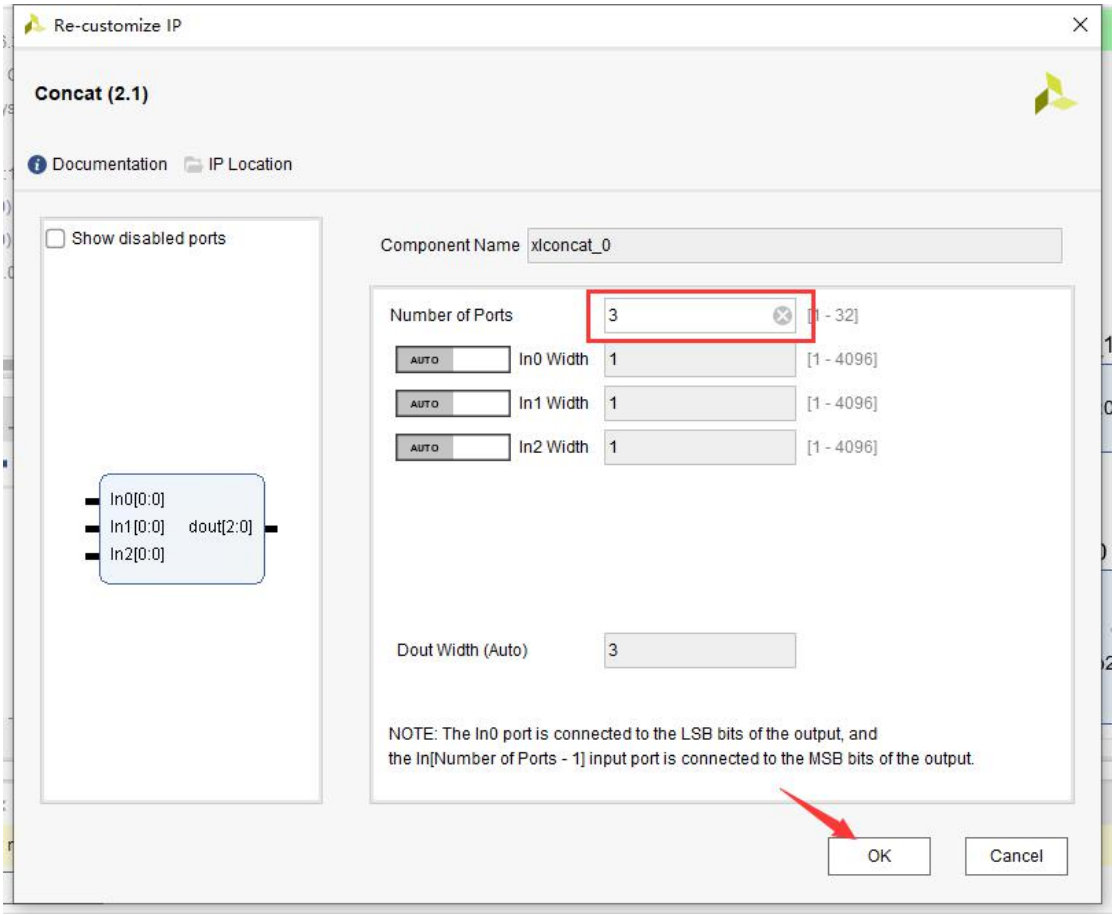


图 17.2.44 设置 concat

将 axi_gpio_0 的中断端口 ip2intc_irpt， axi_vdma_0 的中断端口 mm2s_introut，v_tc_0 的中断端口 irq，分别连接到 xlconcat_0 的 In0，In1，In2 上，并将 xlconcat_0 的输出端口 dout 连接到 Zynq 的 IRQ_F2P 上， 如下图所示：

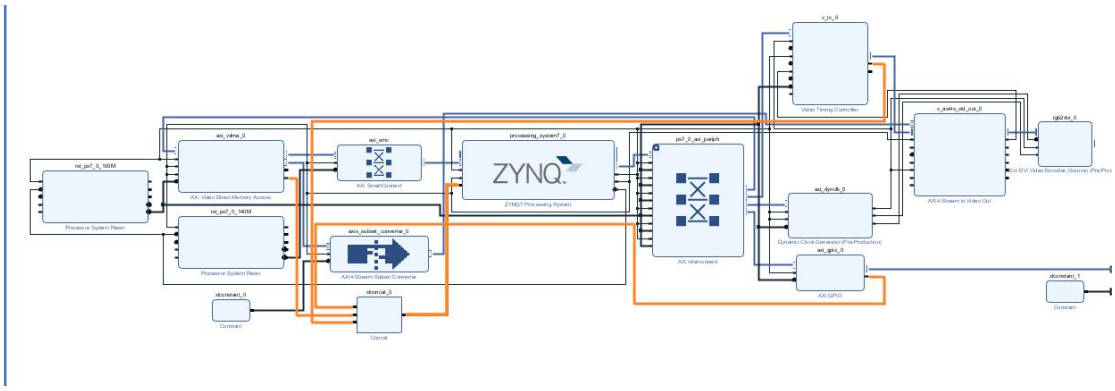


图 17.2.45 连接各中断信号

点击 Run Block Automation，自动生成外设管脚，如下图所示：

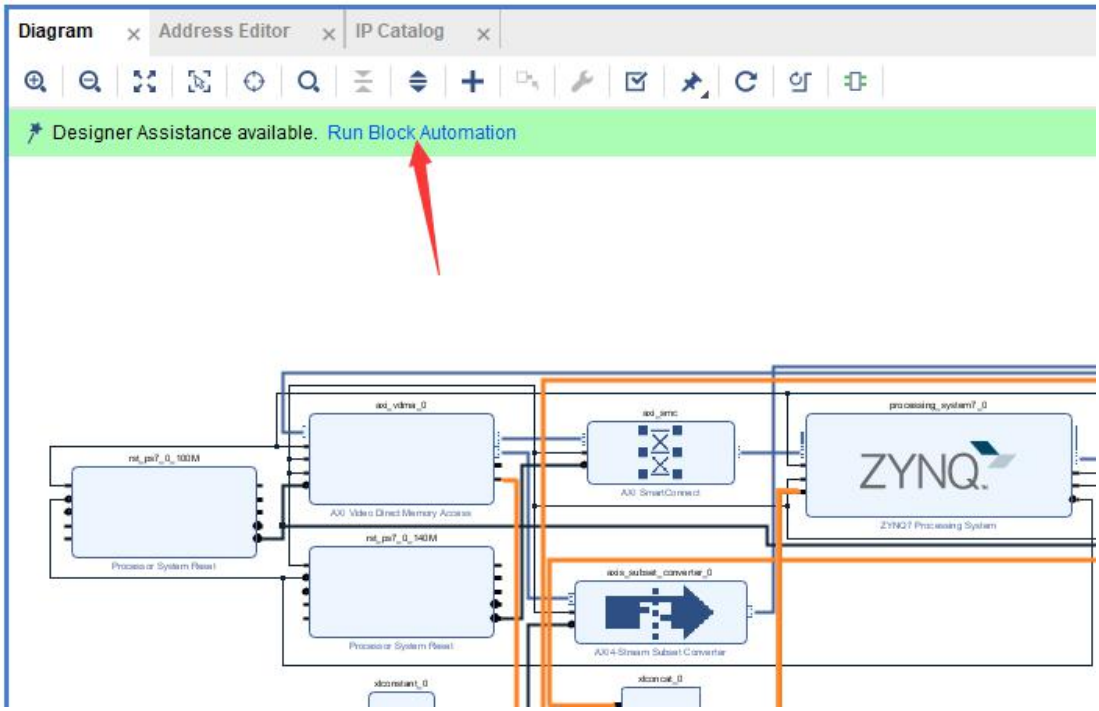


图 17.2.46 生成外设管脚

在弹出的对话框中全选所有信号端口，点击“OK”。

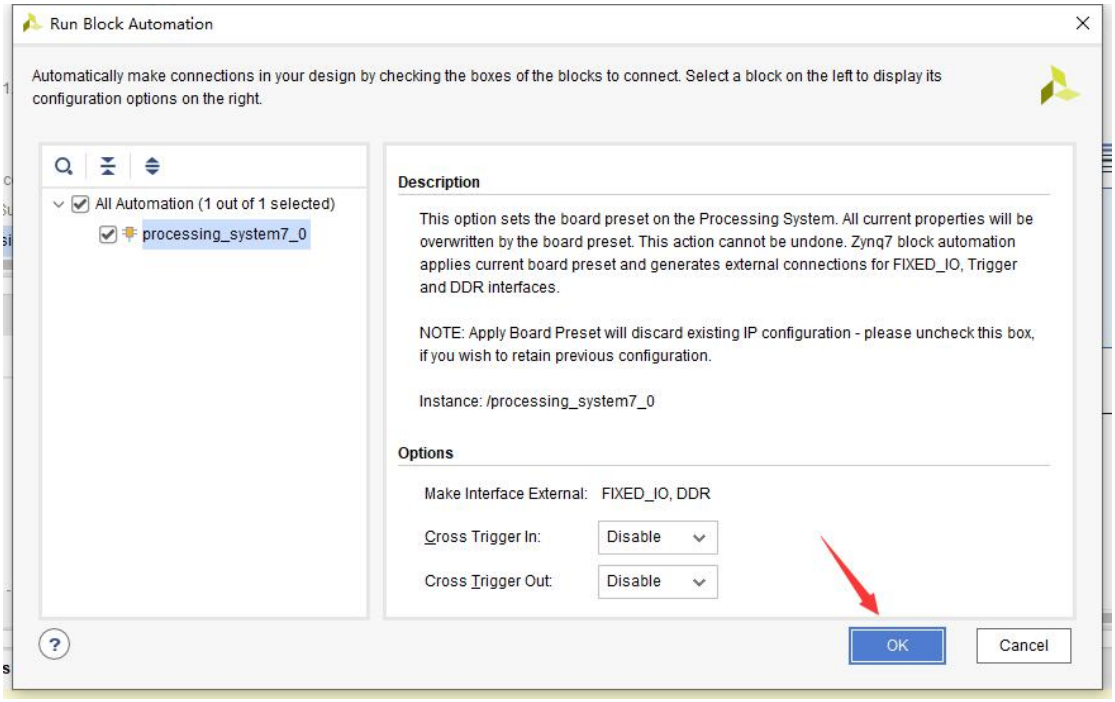


图 17.2.47 全选所有信号

右键单击 rgb2dvi_0 的 TMDS 端口， 选择 Make External 生成外部引脚，

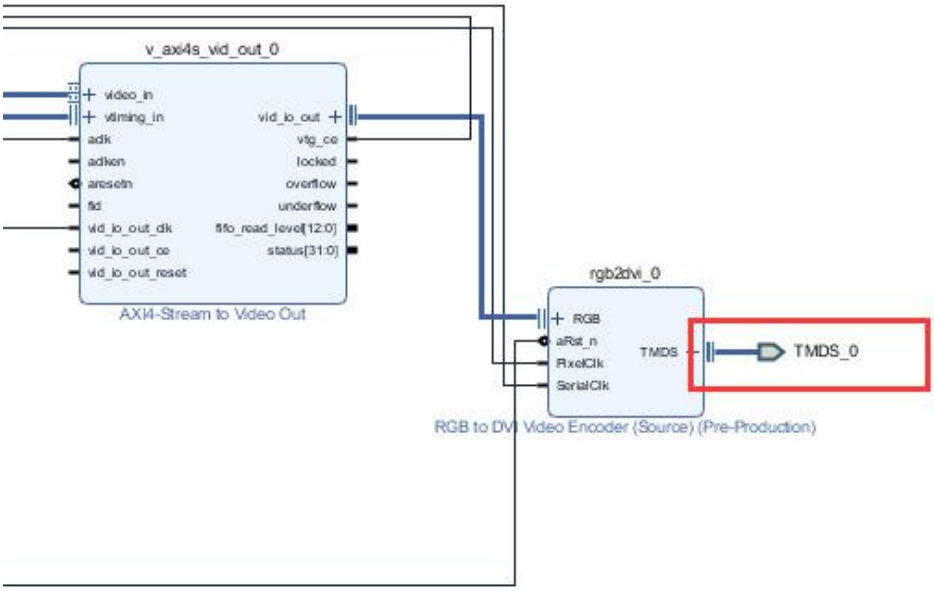


图 17.2.48 生成外部引脚

点击 Regenerate Layout 生成标准布局如下图， 再点击 Validate Design 验证设计：

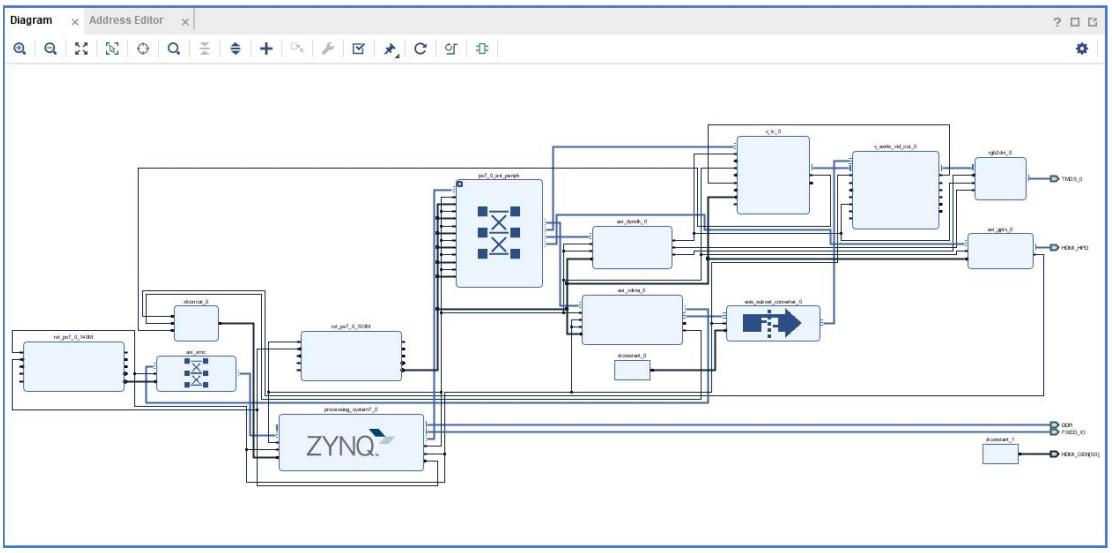


图 17.2.49 标准布局

验证成功后，弹窗点击 OK， Ctrl+S 保存设计。

来到 Source 窗口，生成设计代码和顶层文件，右键点击 hdmi_out 选择 Generate Output Producs， 在弹出的窗口点击 Generate， 点击 OK。右键点击

hdmi_out 选择 Create HDL Wrapper，在弹出的窗口点击 OK。

之后添加管脚约束，单击左侧边栏 RTL Analysis 底下的 Open Elaborated Design，如下图所示：

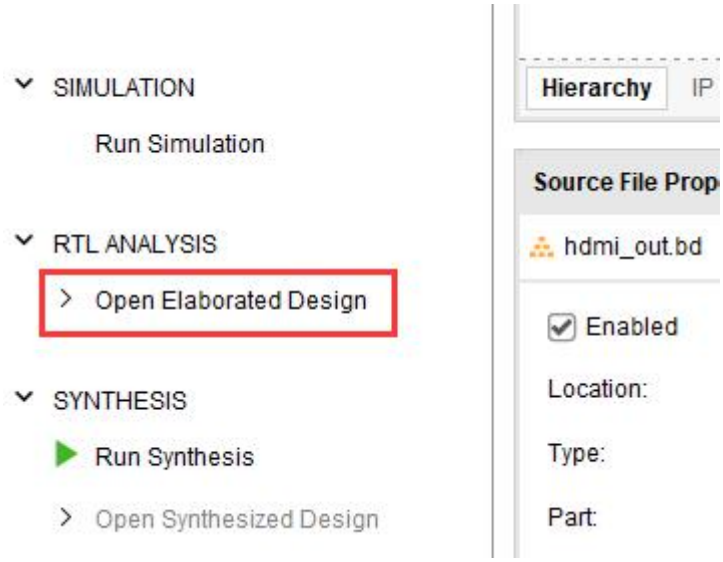
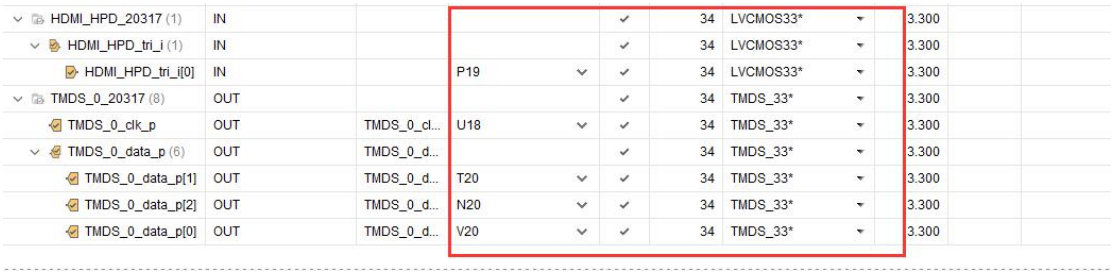


图 17.2.50 打开管脚约束窗口

在设置串口根据原理图设置 Package Pin 和 I/O Std，如下图所示：



HDMI_HPD_20317 (1)	IN			✓	34	LVCMOS33*		3.300
HDMI_HPD_tri_i (1)	IN			✓	34	LVCMOS33*		3.300
HDMI_HPD_tri_i[0]	IN		P19	✓	34	LVCMOS33*		3.300
TMDS_0_20317 (8)	OUT			✓	34	TMDS_33*		3.300
TMDS_0_clk_p	OUT	TMDS_0_cl...	U18	✓	34	TMDS_33*		3.300
TMDS_0_data_p (6)	OUT	TMDS_0_d...		✓	34	TMDS_33*		3.300
TMDS_0_data_p[1]	OUT	TMDS_0_d...	T20	✓	34	TMDS_33*		3.300
TMDS_0_data_p[2]	OUT	TMDS_0_d...	N20	✓	34	TMDS_33*		3.300
TMDS_0_data_p[0]	OUT	TMDS_0_d...	V20	✓	34	TMDS_33*		3.300

图 17.2.51 修改管脚约束

编辑完成，Ctrl+S 保存，在 Save Constraints 弹窗，填入约束文件名 hdmi_out，生成的约束文件如下图所示：

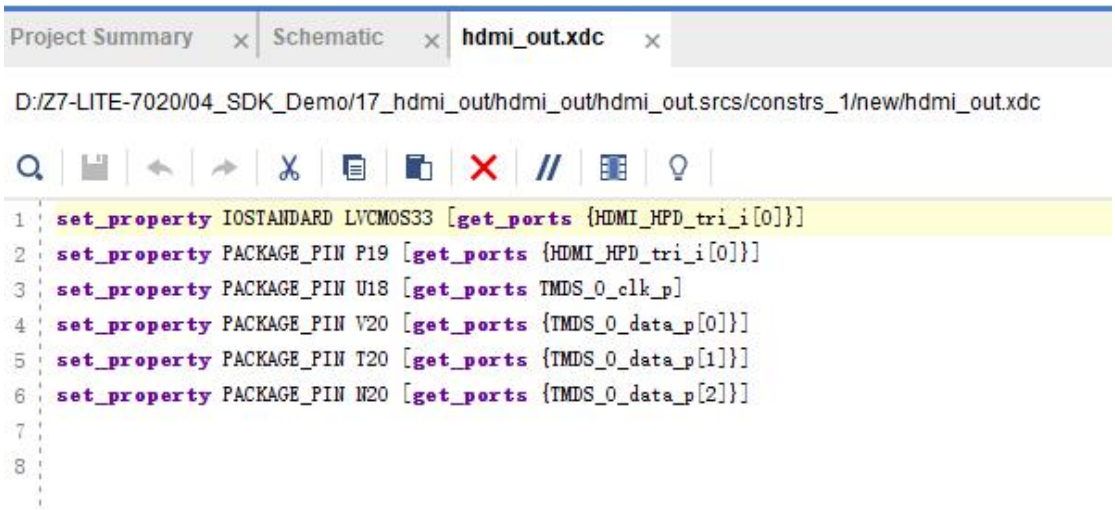


图 17.2.52 管脚约束文件

单击左侧边栏 Program and Debug 底下的 Generate Bitstream 生成 Bit 流文件。生成完成之后点击 Vivado 菜单栏 File > Export > Export Hardware，因为本节设计包含 PL 部分，所以在 Export Hardware 弹窗勾选 Include bitstream，再点击 OK，如下图所示：

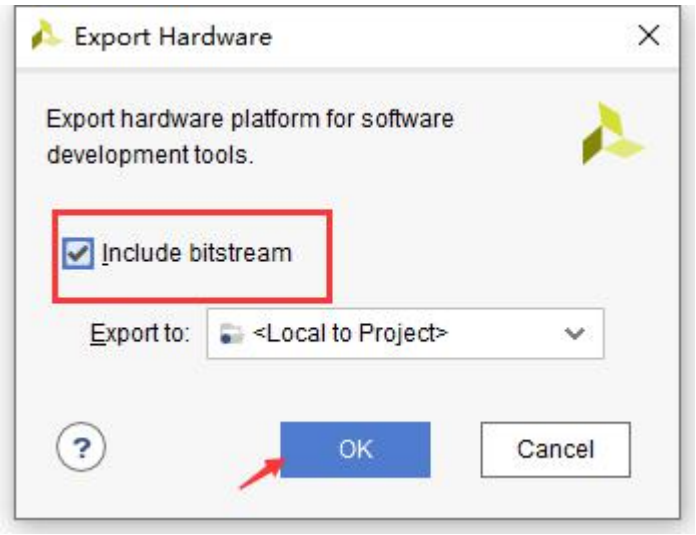
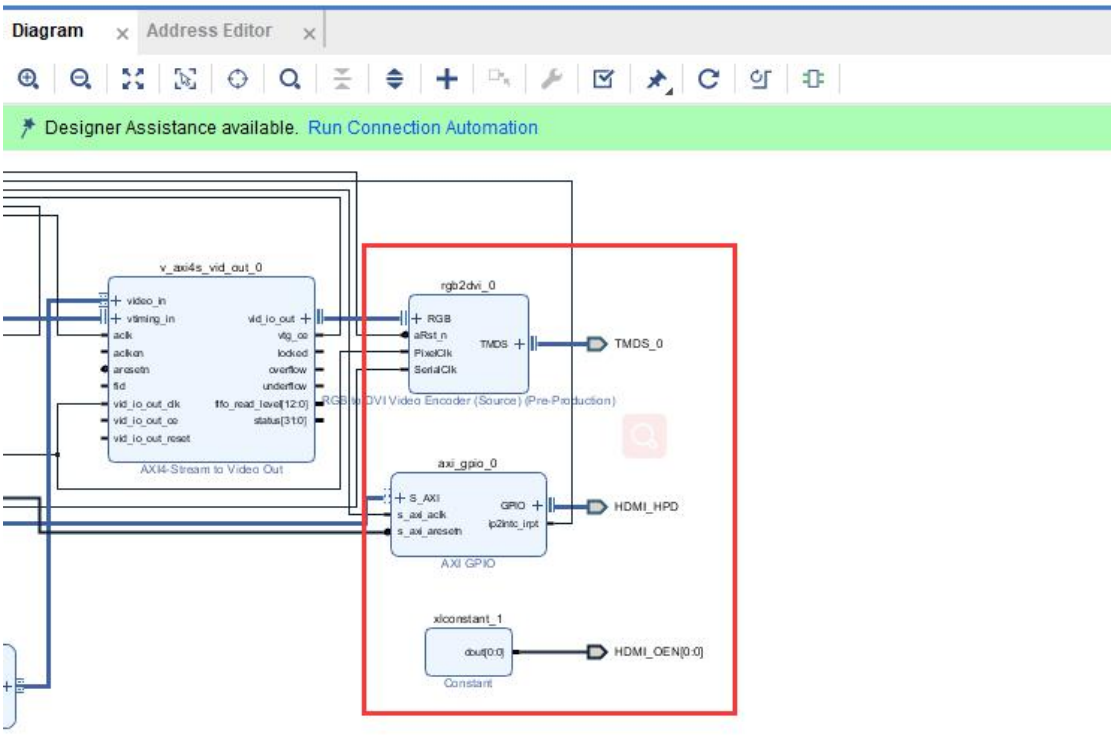


图 17.2.53 导出 Bit 流文件

点击 Vivado 菜单栏 File > Launch SDK，打开的 SDK 工程窗口，到此本实验的硬件设计就完成了。

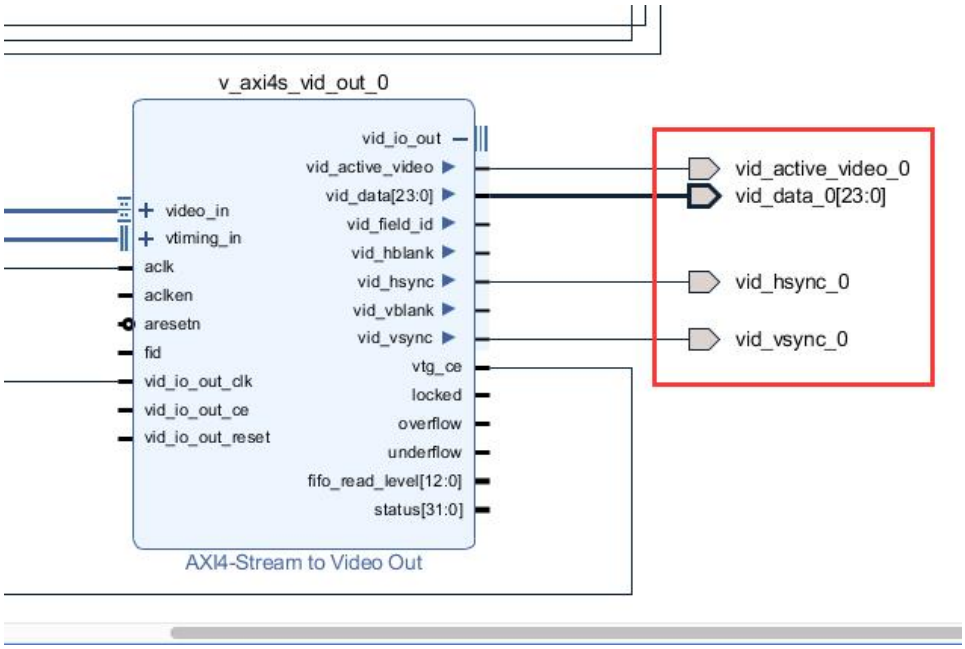
1.1.2 ADV7513 解码芯片

对于使用视频数据解码芯片的开发板，硬件设计在前面的基础上进行简单修



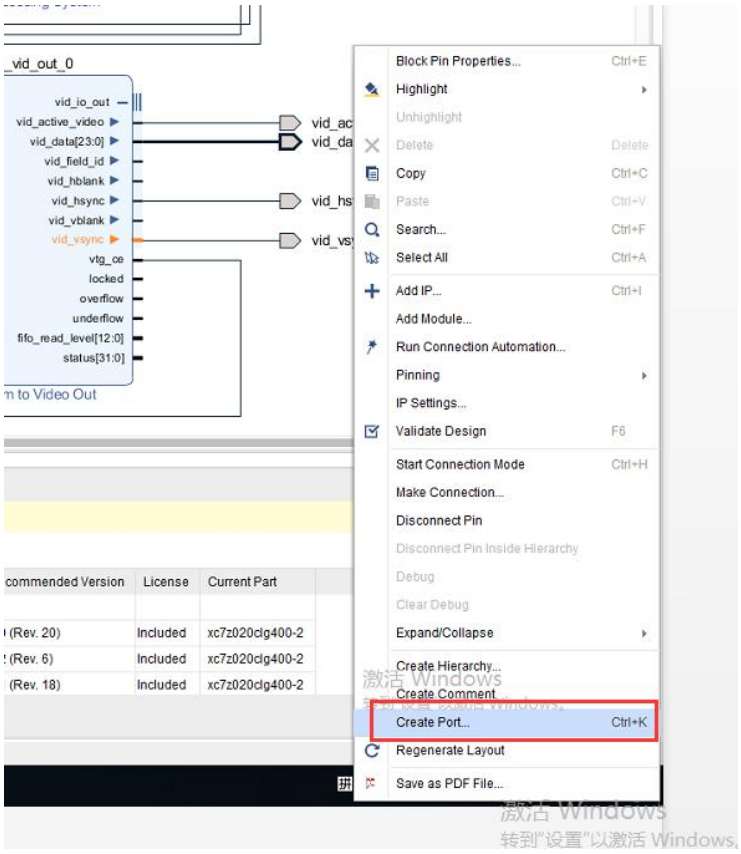
改即可。首先删除如下图所示的几个 IP 核，

图 17.2.54 删除部分 IP



将下图所示几个相关信号引出，

图 17.2.55 引出相关信号



右键点击空白处，在弹出的菜单中选择 Create Port 创建信号端口，

图 17.2.56 创建信号端口

设置端口名称“HDMI_TX_CLK”，端口方向设置为输出，端口属性设置为时钟端口，如下图所示：

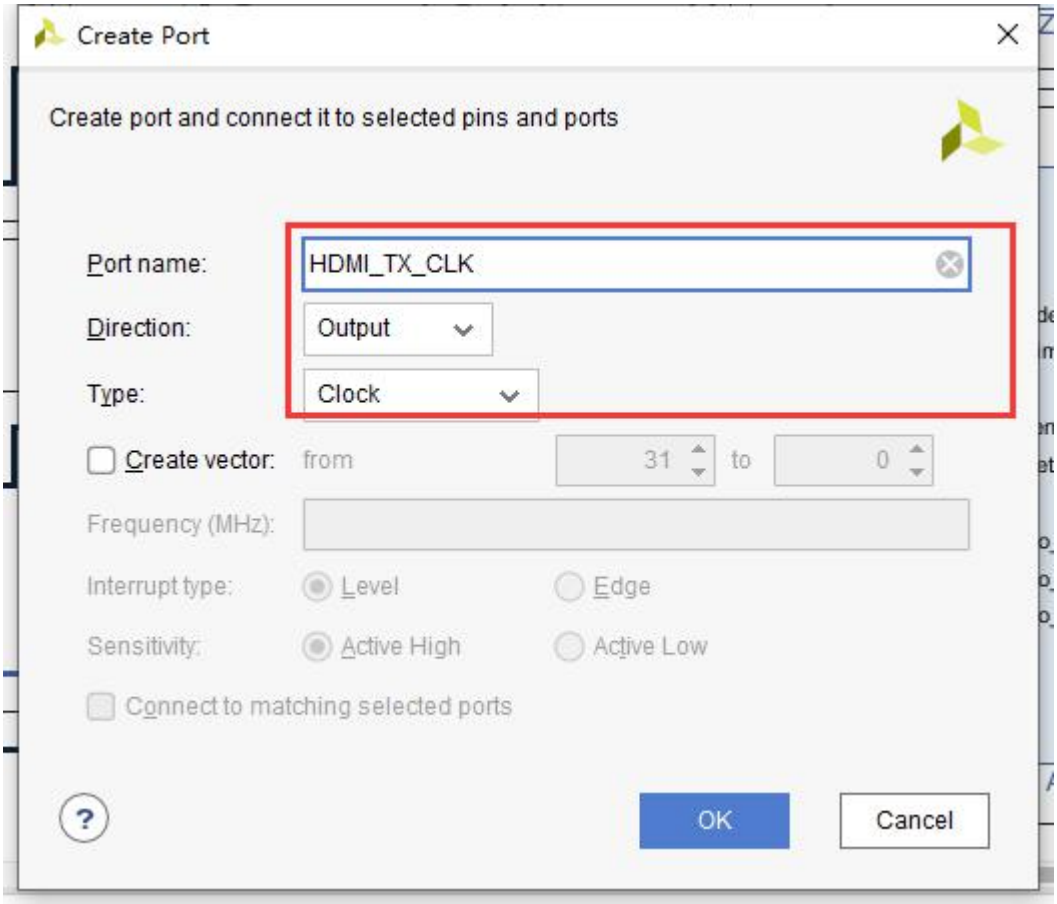


图 17.2.57 配置端口设置

如下图所示，连接时钟信号，

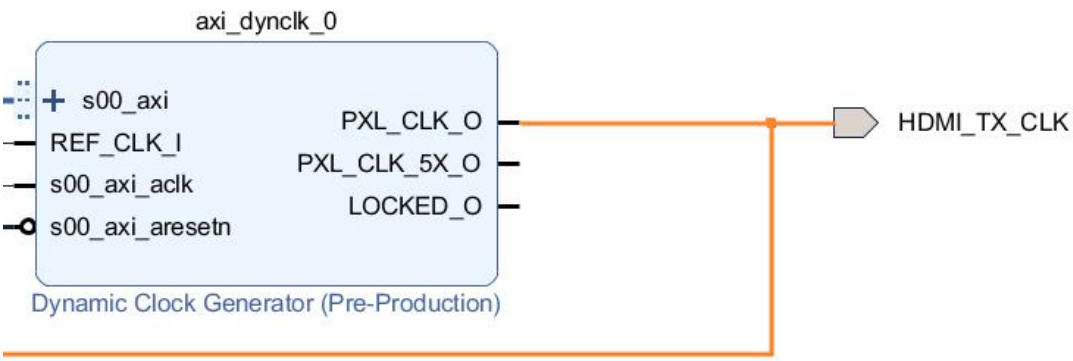
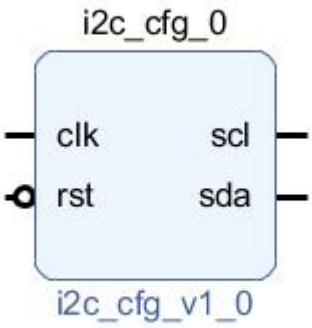


图 17.2.58 连接信号

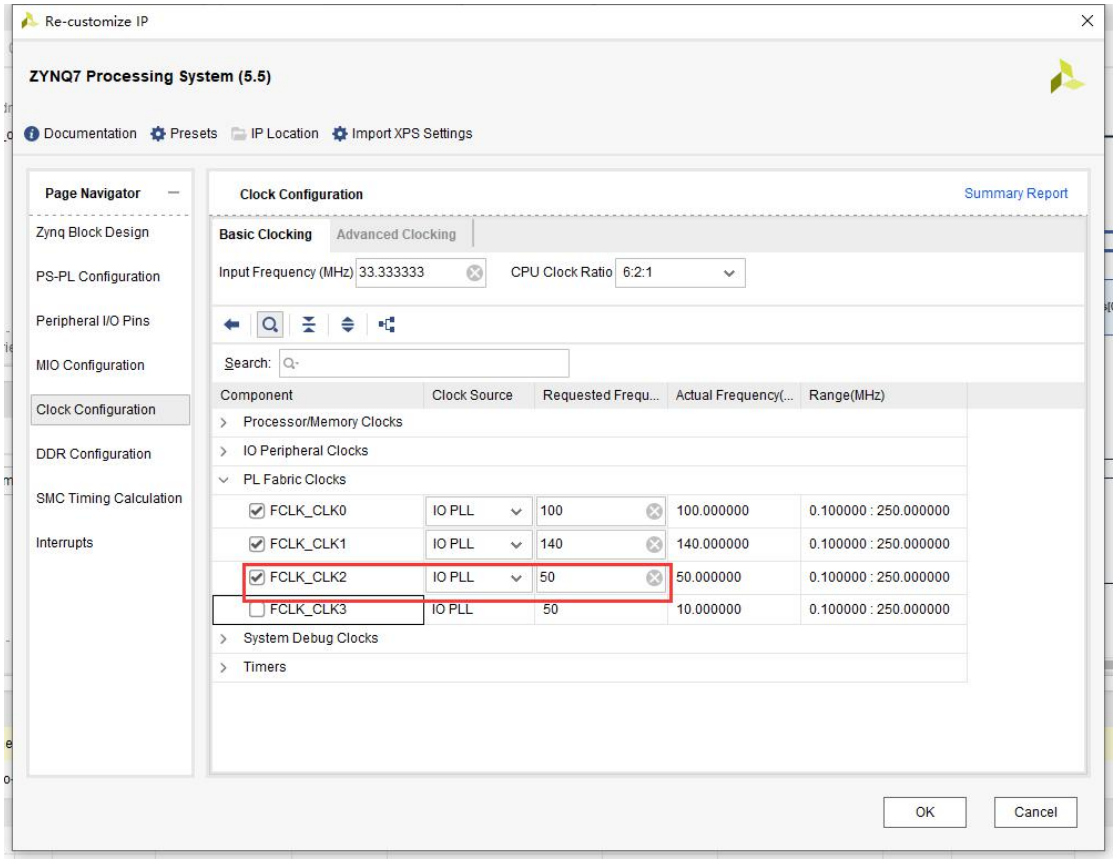
添加 ADV7513 IIC 配置 IP 核，该 IP 保存在该实验相关例程文件夹内的 repo



文件夹，对于不使用 ADV7513 的开发板例程中无此 IP，

图 17.2.59 添加 IIC 配置 IP

双击 ZYNQ7 Processing System IP 打开设置页面，使能 FCLK_CLK2 输出 50Mhz



时钟作为 IIC IP 的驱动时钟，

图 17.2.60 使能 FCLK_CLK2 时钟

添加一个非门 IP，之后如下图所示引出相关信号及连接其他信号，

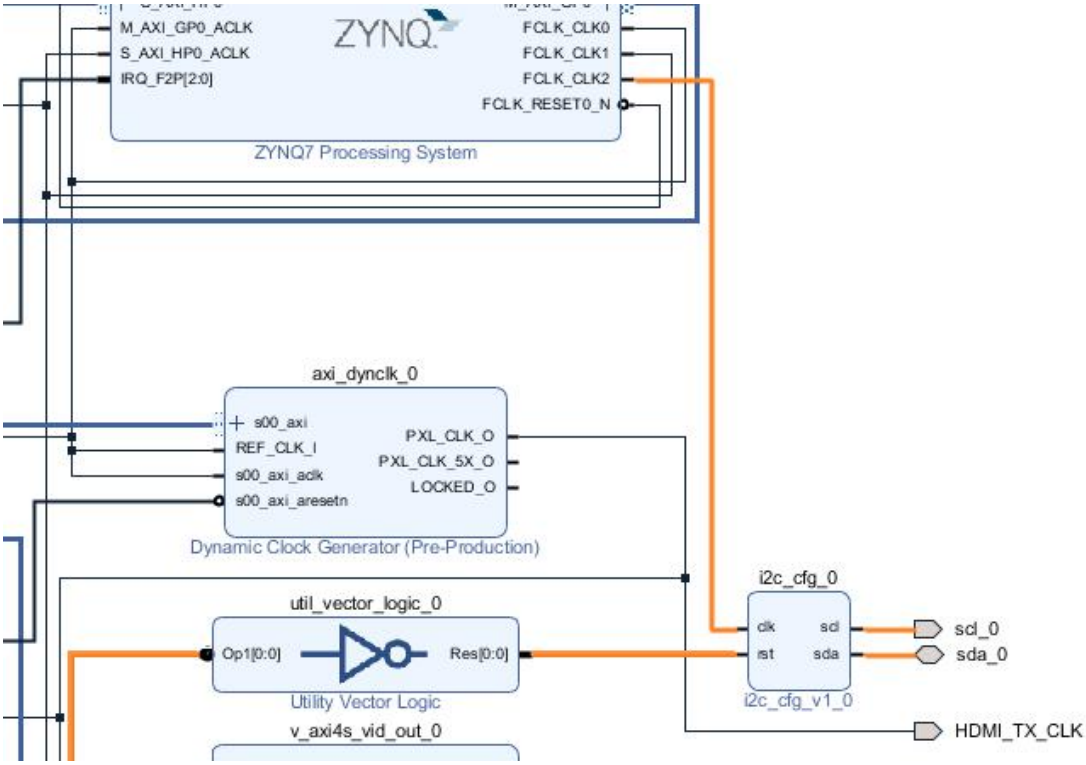


图 17.2.61 引出及连接相关信号

右键点击非门输入端口将信号引出，之后连接一个 PL 端按键作为触发信号，

图 17.2.62 引出非门输入端口

之前为适应 HDMI 转码 IP，在设置 AXI4-Stream Subset Converter 时调整了输出数据的顺序，现在不需要进行调整，按照默认即可，如下图示所示：

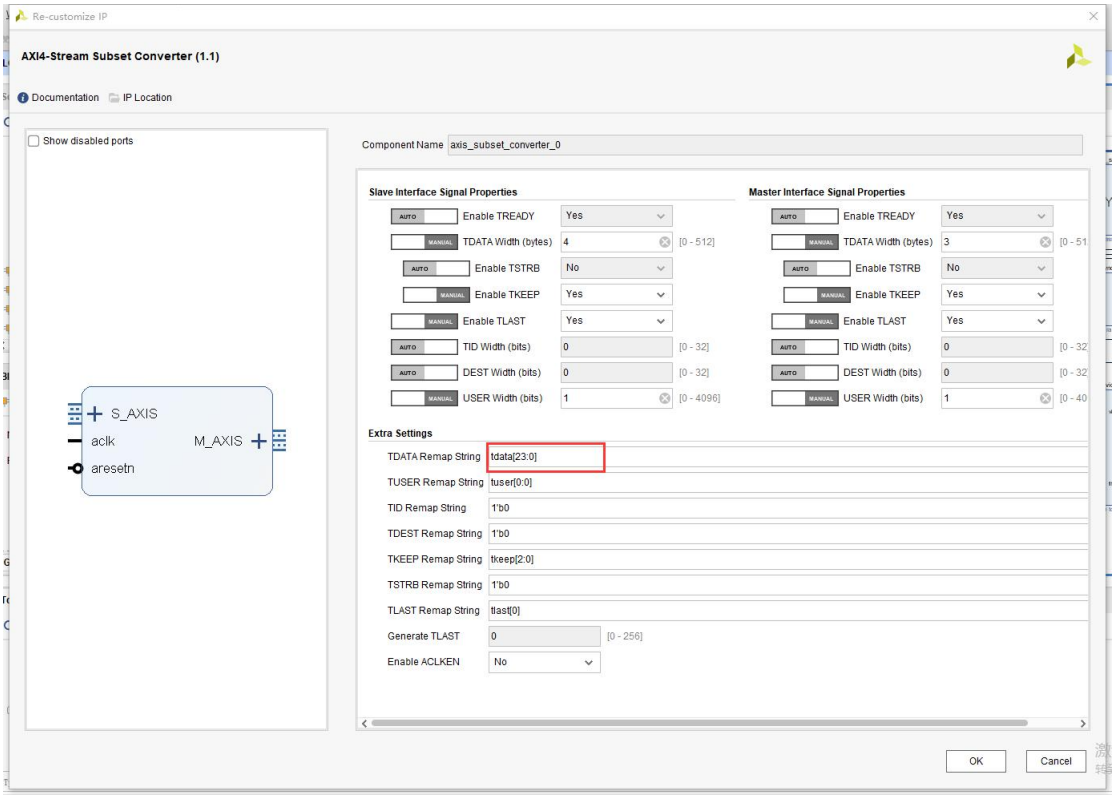


图 17.2.63 设置输出数据顺序

设置完成后保存设计并重新生成顶层 HDL 文件，之后打开引脚约束文件.xdc，输

入下面所示内容进行引脚约束，

```
set_property PACKAGE_PIN D5 [get_ports HDMI_TX_CLK]
set_property PACKAGE_PIN C6 [get_ports {vid_data_0[0]}]
set_property PACKAGE_PIN C5 [get_ports {vid_data_0[1]}]
set_property PACKAGE_PIN E2 [get_ports {vid_data_0[2]}]
set_property PACKAGE_PIN D2 [get_ports {vid_data_0[3]}]
set_property PACKAGE_PIN D1 [get_ports {vid_data_0[4]}]
set_property PACKAGE_PIN C1 [get_ports {vid_data_0[5]}]
set_property PACKAGE_PIN E4 [get_ports {vid_data_0[6]}]
set_property PACKAGE_PIN E3 [get_ports {vid_data_0[7]}]
set_property PACKAGE_PIN C4 [get_ports {vid_data_0[8]}]
set_property PACKAGE_PIN A2 [get_ports {vid_data_0[9]}]
set_property PACKAGE_PIN A1 [get_ports {vid_data_0[10]}]
set_property PACKAGE_PIN A6 [get_ports {vid_data_0[11]}]
set_property PACKAGE_PIN A7 [get_ports {vid_data_0[12]}]
set_property PACKAGE_PIN A4 [get_ports {vid_data_0[13]}]
set_property PACKAGE_PIN A5 [get_ports {vid_data_0[14]}]
set_property PACKAGE_PIN B1 [get_ports {vid_data_0[15]}]
set_property PACKAGE_PIN B2 [get_ports {vid_data_0[16]}]
set_property PACKAGE_PIN B3 [get_ports {vid_data_0[17]}]
set_property PACKAGE_PIN B4 [get_ports {vid_data_0[18]}]
set_property PACKAGE_PIN F1 [get_ports {vid_data_0[19]}]
set_property PACKAGE_PIN F2 [get_ports {vid_data_0[20]}]
set_property PACKAGE_PIN F4 [get_ports {vid_data_0[21]}]
set_property PACKAGE_PIN G4 [get_ports {vid_data_0[22]}]
set_property PACKAGE_PIN G2 [get_ports {vid_data_0[23]}]
set_property PACKAGE_PIN G1 [get_ports vid_active_video_0]
set_property PACKAGE_PIN H1 [get_ports vid_hsync_0]
set_property PACKAGE_PIN H3 [get_ports vid_vsync_0]
set_property PACKAGE_PIN C3 [get_ports scl_0]
set_property PACKAGE_PIN G3 [get_ports sda_0]


set_property IOSTANDARD LVCMOS33 [get_ports HDMI_TX_CLK]
set_property IOSTANDARD LVCMOS33 [get_ports {vid_data_0[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports vid_active_video_0]
set_property IOSTANDARD LVCMOS33 [get_ports vid_hsync_0]
set_property IOSTANDARD LVCMOS33 [get_ports vid_vsync_0]
set_property IOSTANDARD LVCMOS33 [get_ports scl_0]
set_property IOSTANDARD LVCMOS33 [get_ports sda_0]
set_property PACKAGE_PIN J1 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
```

之后保存并重新生成 Bitstream。生成完成之后导出到 SDK 即可进行软件设计。

17.3 软件设计

在打开的 SDK 软件内点击 File > New > Application Project，工程名填入“hdmi_out”，工程模板选择 Empty Application 工程， 点击 Finish，如下图所示：

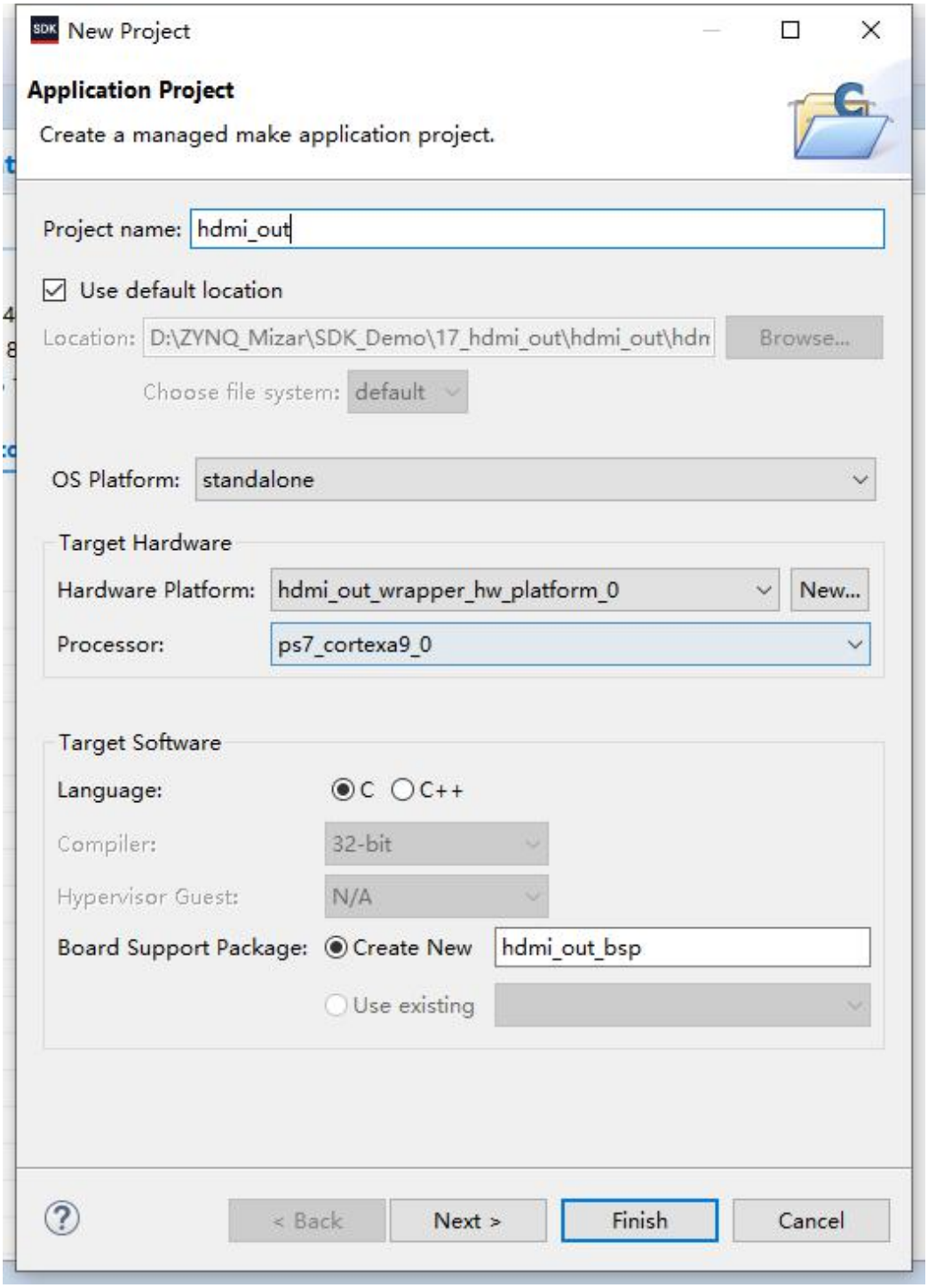
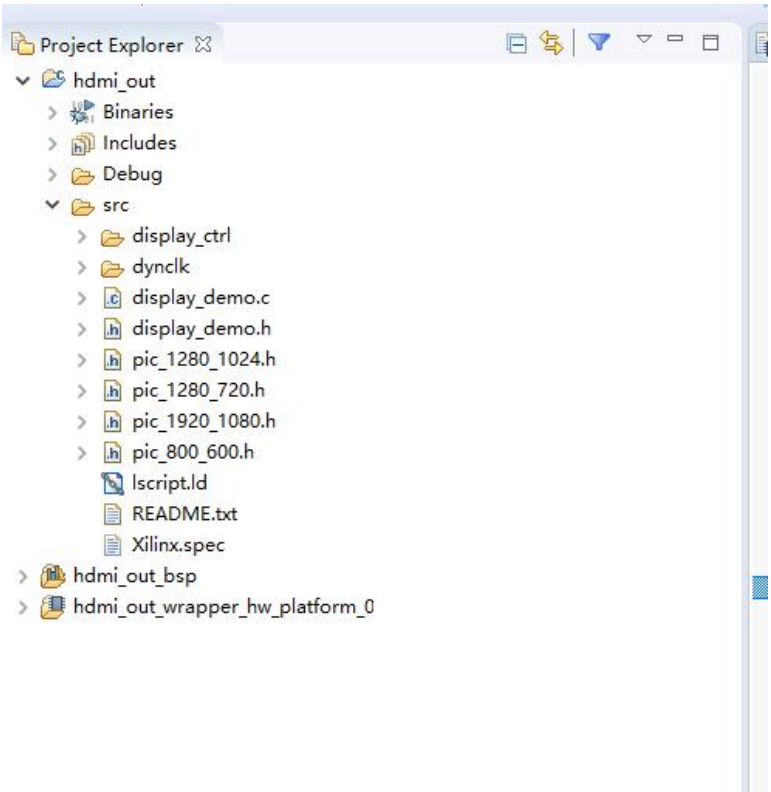


图 17.3.1 创建“hdmi_out”工程

创建完成后，打开资料目录 SDK_Demo/17_hdmi_out/hdmi_out/

hdmi_out.sdk/hdmi_out/src 文件夹，复制文件夹内的所有文件，粘贴到当前工程同样的目录下(注意，不要覆盖本工程下的 lscript.ld)， 右键单击工程 hdmi_out 选择 Refresh，则开始自动编译。双击打开 src 目录底下的 display_demo.c， 这就是 SDK 主程序。

接下来讲解代码，展开工程下的 src 可以看到文件结构如下图所示，其中 display_ctrl 文件夹包含不同视频分辨率情况下的时序控制， dynclk 包含不同分



辨率情况下的像素时钟和串行时钟生成。

图 17.3.2 源程序文件结构图

打开 display_demo.c 文件，这就是本实验的主程序，我们分段讲解各部分的作用。

```
// Image data for each resolution
//@@@
```

这部分提供了 4 个不同分辨率图像的像素数据，这些数据以头文件的形式包含在工程目录中，比如 pic_800_600.h。 以 800x600 图像像素为例，如下图，每个像素以三基色表示，一共 800x600x3 = 1440000 个数据点。屏幕左上角坐标(0, 0)处第一个像素的蓝色，绿色，红色数值依次是 0xF1, 0xDC, 0xDB， 其它点的像

system.hdf　system.mss　*display_demo.c　pic_800_600.h

```
const unsigned char Pixel_800_600[1440000] = {
0XF1,0XDC,0XDB,0XF1,0XDC,0XDB,0XF3,0XDE,0XDD,0XF4,0XDF,0XDE,0XF5,0XE0,0XDF,0XF6,0XE1,0XE0,
0XF7,0XE2,0XE1,0XF7,0XE2,0XE1,0XFC,0XE7,0XE6,0XFB,0XE7,0XE6,0XFB,0XE7,0XE6,0XFC,0XE8,0XE7,
0XFC,0XE8,0XE7,0XFC,0XE8,0XE7,0XFC,0XE8,0XE7,0XFD,0XE9,0XE8,0XFD,0XE9,0XE8,0XFE,0XEA,0XE9,
0XFF,0XEB,0XEA,0XFF,0XED,0XEC,0XFF,0XED,0XEC,0XFF,0XED,0XEC,0XFF,0XEC,0XEB,0XFF,0XEB,0XEA,
0XFF,0XEB,0XEA,0XFF,0XEB,0XEA,0XFF,0XEC,0XEB,0XFF,0XEC,0XEB,0XFF,0XEC,0XEB,0XFF,0XEB,0XEA,
0XFF,0XEB,0XEA,0XFE,0XEA,0XE9,0XFF,0XEC,0XEA,0XFF,0XEB,0XE9,0XFF,0XEA,0XE8,0XFF,0XEB,0XE9,
0XFF,0XEC,0XEA,0XFF,0XEE,0XEB,0XFF,0XEE,0XEB,0XFF,0XED,0XEA,0XFD,0XEC,0XE9,0XFC,0XEB,0XE8,
0XFC,0XEB,0XE8,0XFC,0XEB,0XE8,0XFB,0XEC,0XE9,0XFC,0XED,0XEA,0XFD,0XEE,0XEB,0XFD,0XEE,0XEB,
0XFA,0XF0,0XF0,0XFB,0XF1,0XF1,0XFD,0XF3,0XF3,0XFB,0XF3,0XF3,0XFB,0XF3,0XF3,0XFC,0XF4,0XF4,
0XFD,0XF5,0XF5,0XFD,0XF8,0XF7,0XFC,0XF7,0XF6,0XFD,0XF8,0XF7,0XFC,0XF8,0XF7,0XFD,0XF9,0XF8,
0XFD,0XF9,0XF8,0XFD,0XF9,0XF8,0XFC,0XFA,0XF9,0XFC,0XFA,0XF9,0XFB,0XF9,0XF9,0XFB,0XF9,0XF9,
0XFB,0XF9,0XF9,0XFB,0XF9,0XF9,0XFC,0XFA,0XFA,0XFD,0XFB,0XFB,0XFC,0XFA,0XFA,0XFC,0XFA,0XFA,
0XFB,0XF9,0XF9,0XFB,0XF9,0XF9,0XFC,0XFA,0XFA,0XFC,0XFA,0XFA,0XFD,0XFB,0XFB,0XFE,0XFC,0XFC,
0XFF,0XFD,0XFD,0XFF,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,
0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFF,0XFF,0XFF,0XFF,0XFF,0XFF,
0XFF,0XFF,0XFF,0XFF,0XFF,0XFF,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,0XFE,
```

素值依次类推。

显示控制结构体 DisplayCtrl 和 VDMA 例化（读者可以按住 Ctrl 键的同时鼠标左键点进去看看这两个结构体包含的内容）。

定义帧缓冲空间，和指向它的指针数组。

初始化指针数组。

```
 * Frame buffers for video data
 */
u8 frameBuf[DISPLAY_NUM_FRAMES][DEMO_MAX_FRAME];
     * Initialize an array of pointers to the 3 frame buffers    uffers
     */
    for (i = 0; i < DISPLAY_NUM_FRAMES; i++)
    {
        pFrames[i] = frameBuf[i];
    }
XAxiVdma vdma;

```

初始化 VDMA 模块

```
 * Initialize VDMA driver, get the hardware VDMA configurations
 */
vdmaConfig = XAxiVdma_LookupConfig(VGA_VDMA_ID);
if (vdmaConfig == NULL)
{
    xil_printf("No video DMA found for ID %d\r\n", VGA_VDMA_ID);
}

/*
 * Use hardware VDMA configurations to initialize the driver
 */
Status = XAxiVdma_CfgInitialize(&vdma, vdmaConfig, vdmaConfig->BaseAddress);
if (Status != XST_SUCCESS)
{
    xil_printf("VDMA Configuration Initialization failed %d\r\n", Status);
}
```

本例支持的所有显示模式。 注意， 这个要和头文件的分辨率选择对应。比如头文件选择的是 pic_800_600.h， 这里就要选择 VMODE_800x600 这一项。

```
// Video Mode for each resolution
//@@@
```

根据不同的分辨率，选择使用不同的像素文件，填充帧缓冲空间。

```
124     for(ycoi = 0; ycoi < height; ycoi++)
125     {
126         for(xcoi = 0; xcoi < (width * 4); xcoi+=4)
127         {
128
129             //@@@
```

编译完成即可烧录到开发板中进行验证。

针对不包含 ADV7513 和包含 ADV7513 的两种类型的开发板，软件程序完全相同。

17.4 下载验证

连接电脑 USB 到开发板 JTAG 口，连接电脑另一个 USB 到开发板 UART 口，确保开发板启动模式设置为 JTAG，连接开发板 HDMI 口到 HDMI 显示器，给开发板上电。按照之前实验的操作方式连接串口（在本实验中，如果未发送错误则串口不打印信息）。

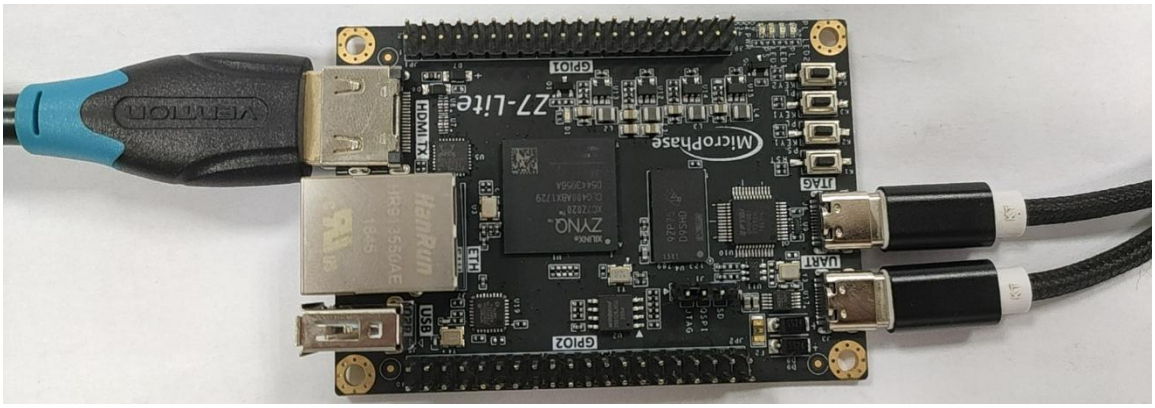
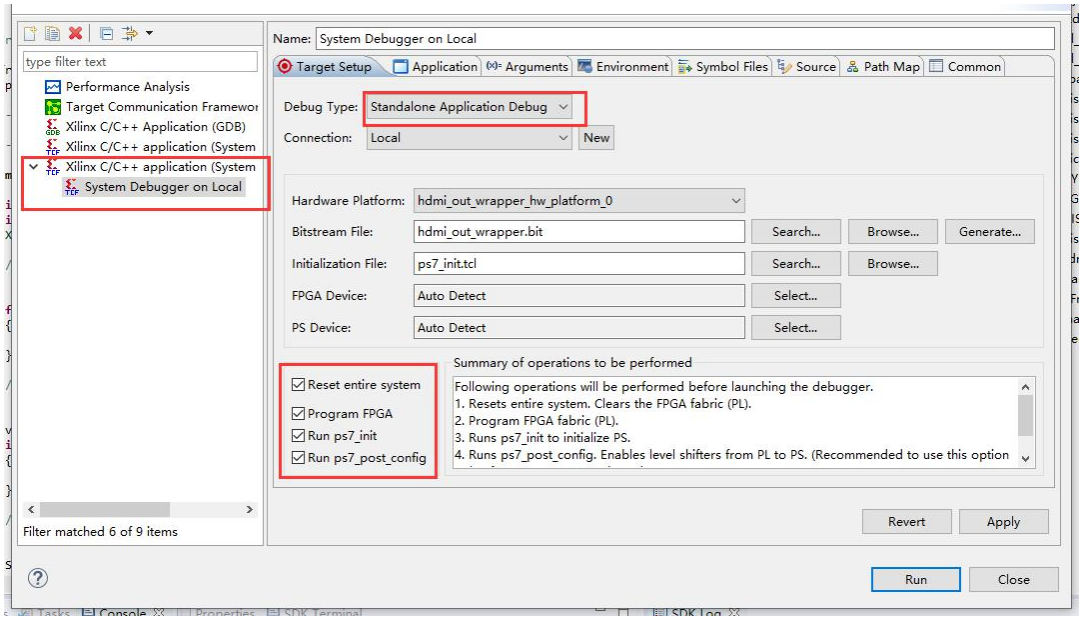
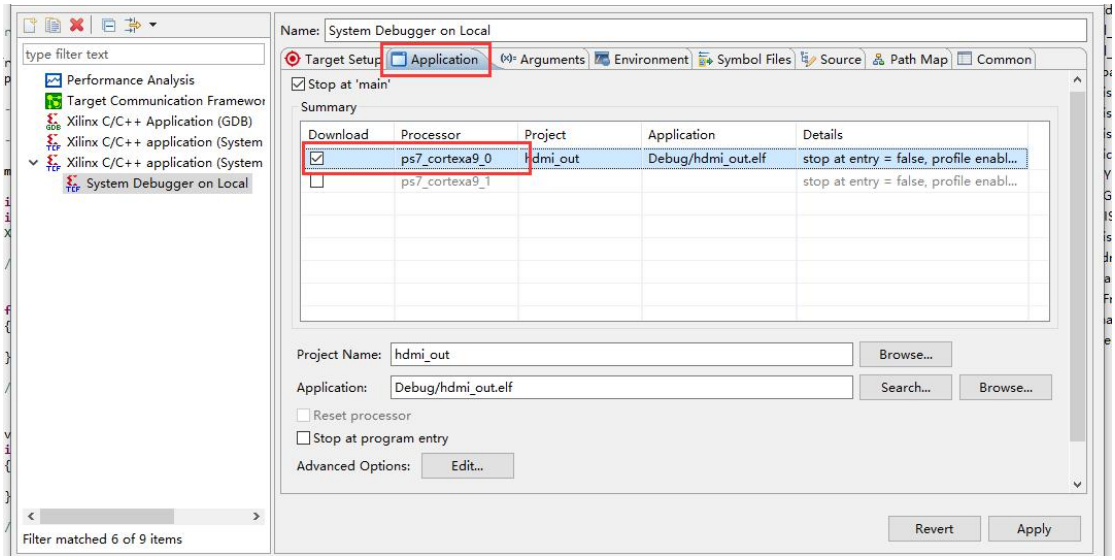


图 17.4.1 端口连接



之后点击 Run->Run Configurations...，之后如下图所示设置相关参数。

图 17.4.1 设置 fpga 烧录设置



在 Application 页面设置软件烧录设置，如下图所示：

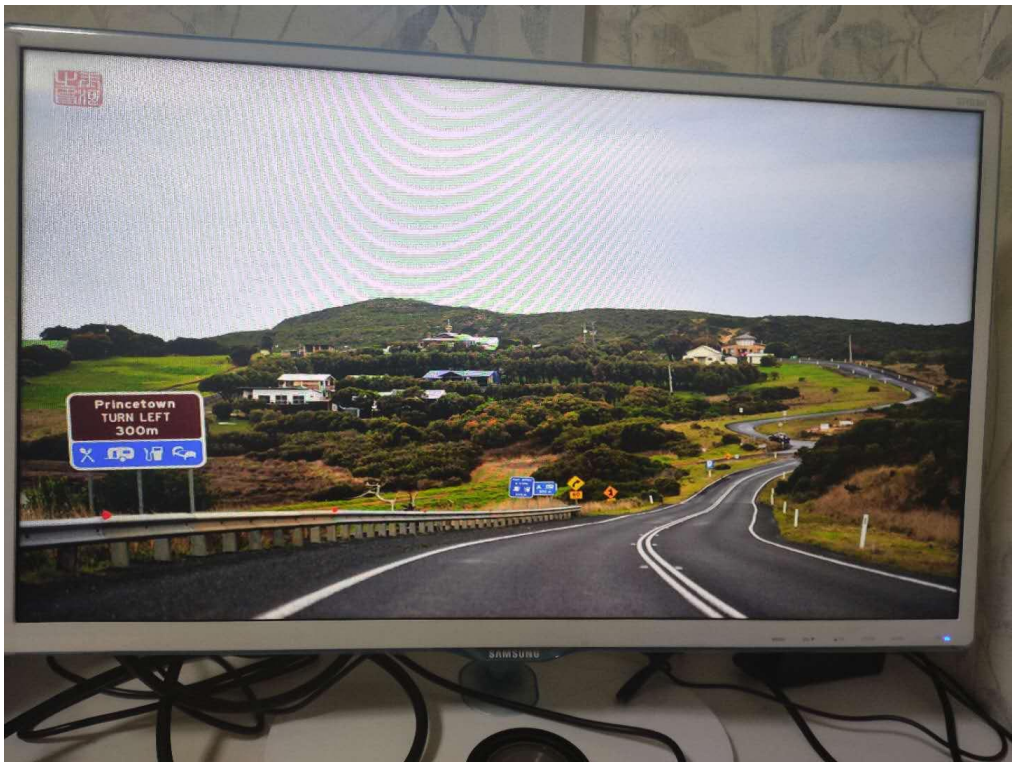
设置完成后点击“Run”即可烧录，烧录完成后可以在 HDMI 显示器上看到指定图像。

若想修改显示的图像请按照以下步骤操作：

第一， 选择想要显示的图像头文件，

```
// Image data for each resolution
//@@@
```

800 x 600：

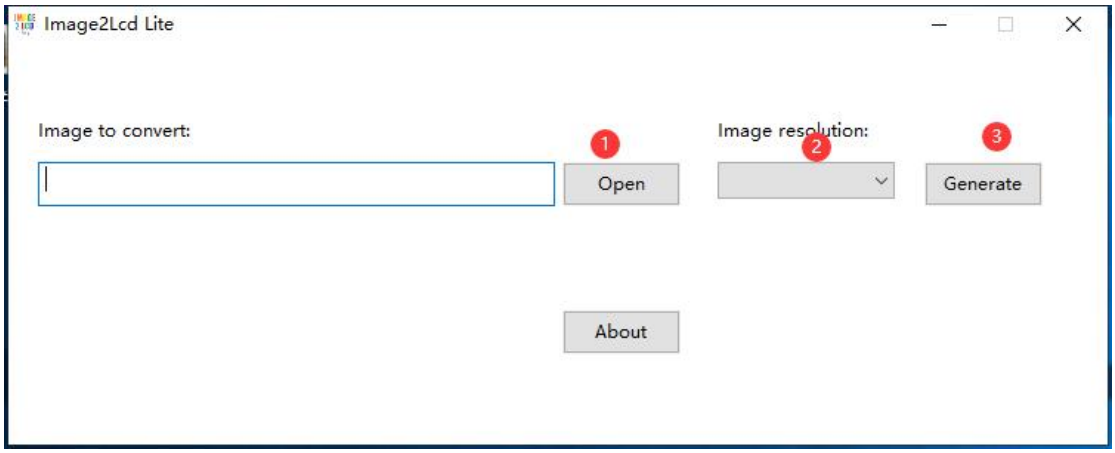


1280 x 720：

1280 x 1024：

1920 x 1080：

为方便读者显示自己喜欢的图片，在我们提供的 Tools 文件夹下，Image2Lcd Lite 的安装包，双击 Image2Lcd Lite.exe 安装。为方便起见，安装过程中注意勾选 Createa desktop shortcut。安装完成后，双击桌面 Image2Lcd Lite 图标，打开操作窗口，如下图所示：



首先点击① Open 按钮，选择图片。

再点击②选择当前图片的分辨率，这里只支持 800x600， 1280x720，

1280x1024， 1920x1080 四种。如果图片的实际分辨率和这里指定的分辨率不同，则生成像素文件时会有弹窗错误信息。

最后点击③Generate 按钮生成图片的像素文件。生成的像素文件以 .h 头文件的形式给出，存储在和图片相同的路径下。

之后就可以将头文件添加到工程中。

17.5 章节小结

在本实验中，我们初步实现了非常基础的单张图片静态显示，通过这种方式来了解 HDMI 的使用方法，相较于之前的实验，本次实验的硬件设计比较复杂，步骤众多，希望读者在操作时仔细阅读本实验教程，按步骤慢慢操作。并在此基础上尝试开发更多的应用。

第 18 章 读取 SD 卡内 BMP 图片 HDMI 显示实验

在之前的实验中，我们学习了怎样通过 HDMI 的接口在显示屏上显示图片，我们直接将需要显示的图片的数据保存在头文件中，然后刷新到帧缓存中。但在实际应用中，通常将图片或视频保存在外部存储设备如 SD 卡中，或者直接显示摄像头采集的图像。本实验我们就来学习读取显示保存在 SD 卡中的图片。

本实验包括以下几部分内容：

18.1 实验简介

18.2 硬件设计

18.3 软件设计

18.4 下载验证

18.5 章节小结

18.1 实验简介

在本次实验中我们将读取 SD 卡中的一张 BMP 格式的图片显示在 HDMI 屏幕上。

BMP(Bitmap-File)图形文件是 Windows 采用的图形文件格式，在 Windows 环境下运行的所有图象处理软件都支持 BMP 图象文件格式。Windows 系统内部各图像绘制操作都是以 BMP 为基础的。它采用位映射存储格式，除了图像深度可选以外，不采用其他任何压缩，因此， BMP 文件所占用的空间很大，但是没有失真。 BMP 文件的图像深度可选 lbit、 4bit、 8bit、 16bit、 24bit 及 32bit。BMP 文件存储数据时，图像的扫描方式是按从下到上、从左到右的顺序。

BMP 文件的数据按照从文件头开始的先后顺序分为四个部分：

- 位图文件头(bmp file header)： 提供文件的格式、大小等信息

◆ 位图信息头(bitmap information)：提供图像数据的尺寸、位平面数、压缩方式、颜色索引等信息

◆ 调色板(color palette)：可选，如使用索引来表示图像，调色板就是索引与其对应的颜色的映射表

◆ 位图数据(bitmap data)：图像数据区

BMP 图片文件数据表如下：

数据段名称	大小（byte）	开始地址	结束地址
位图文件头(bitmap-file header)	14	0000h	000Dh
位图信息头(bitmap-information header)	40	000Eh	0035h
调色板(color table)	由biBitCount决定	0036h	未知
图片点阵数据(bitmap data)	由图片大小和颜色定	未知	未知

在本实验中，我们使用的图片为 24 位图像，即 R、 G、 B 三种颜色各用 8 个 bit 来表示，这样的图像我们称之为真彩色。在这种情况下是不需要调色板的，位图信息头后面紧跟的就是位图数据了。也就是说，位图文件从文件头开始偏移 54 个字节就是图像数据了。

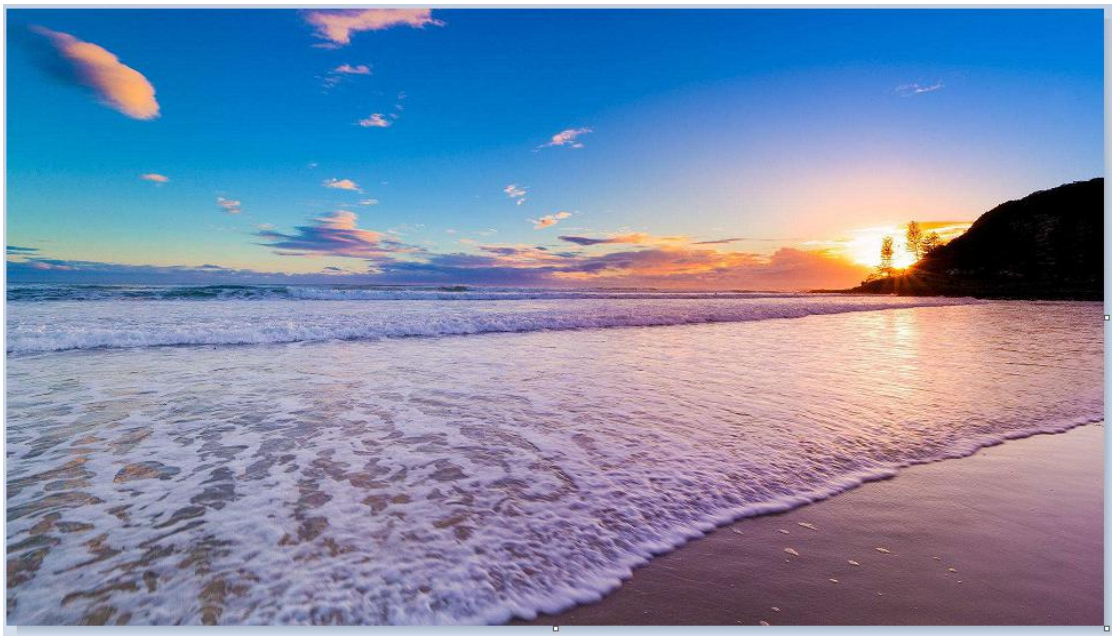
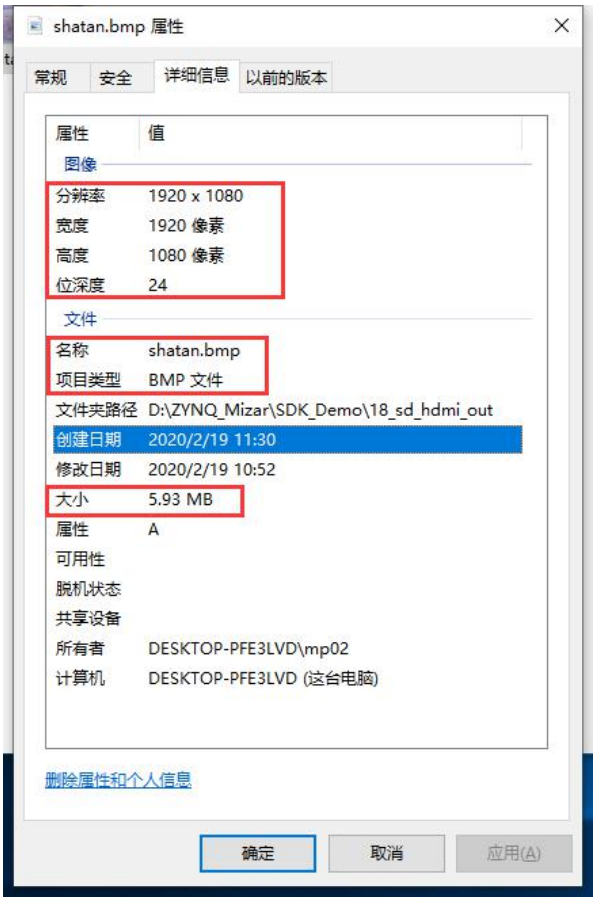


图 18.1.1 示例图片



我们可以看到图片的属性信息如下图所示：

图 18.1.2 示例图片属性

在实验中，我们通过读取图片的信息头来了解图片的属性信息，图片的信

变量名	地址偏移	大小	作用说明
biSize	000Eh	4Bytes	BNP信息头即BMP_INFOHEADER结构体所需要的字节数（以字节为单位）
biWidth	0012h	4Bytes	说明图像的宽度（以像素为单位）
biHeight	0016h	4Bytes	说明图像的高度（以像素为单位）。这个值还有一个用处，指明图像是正向的位图还是倒向的位图，该值是正数说明图像是倒向的即图像存储是由下到上；该值是负数说明图像是倒向的即图像存储是由上到下。大多数BMP位图是倒向的位图，所以此值是正值。
biPlanes	001Ah	2Bytes	为目标设备说明位面数，其值总设置为1
biBitCount	001Ch	2Bytes	说明一个像素点占几位（以比特位/像素位单位），其值可为1,4,8,16,24或32
biCompression	001Eh	4Bytes	说明图像数据的压缩类型，取值范围为： 0 BI_RGB 不压缩（最常用） 1 BI_RLE8 8比特游程编码（BLE），只用于8位位图 2 BI_RLE4 4比特游程编码（BLE），只用于4位位图 3 BI_BITFIELDS比特域（BLE），只用于16/32位位图 4
biSizeImage	0022h	4Bytes	说明图像的大小，以字节为单位。当用BI_RGB格式时，总设置为0
biXPelsPerMeter	0026h	4Bytes	说明水平分辨率，用像素/米表示，有符号整数
biYPelsPerMeter	002Ah	4Bytes	说明垂直分辨率，用像素/米表示，有符号整数
biClrUsed	002Eh	4Bytes	说明位图实际使用的调色板索引数，0：使用所有的调色板索引
biClrImportant	0032h	4Bytes	说明对图像显示有重要影响的颜色索引的数目，如果是0，表示都重要。

息头数据如下表所示：

18.2 硬件设计

本实验的硬件设计基本与实验十七相同，所以不必重新建立新的工程，打开实验十七的 Vivado 工程，另存为一个新的工程并明命名为“sd_hdmi_out”,如下图所示，完成之后打开工程文件夹，删除.sdk 文件夹，这是上一个实验的程序，本实验不需要。

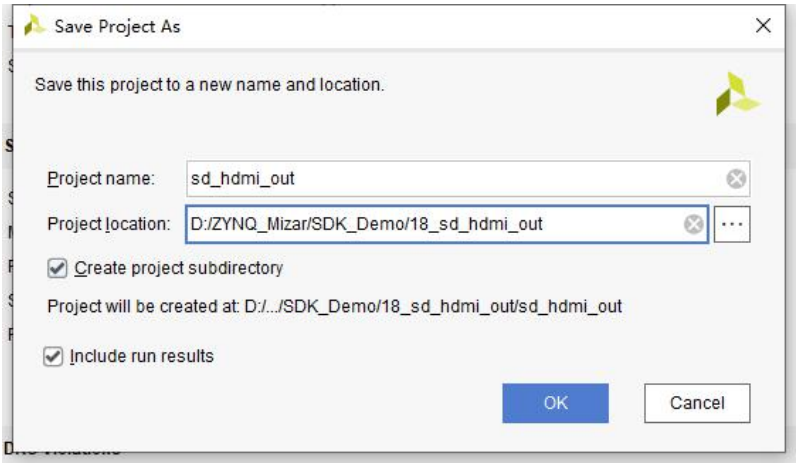
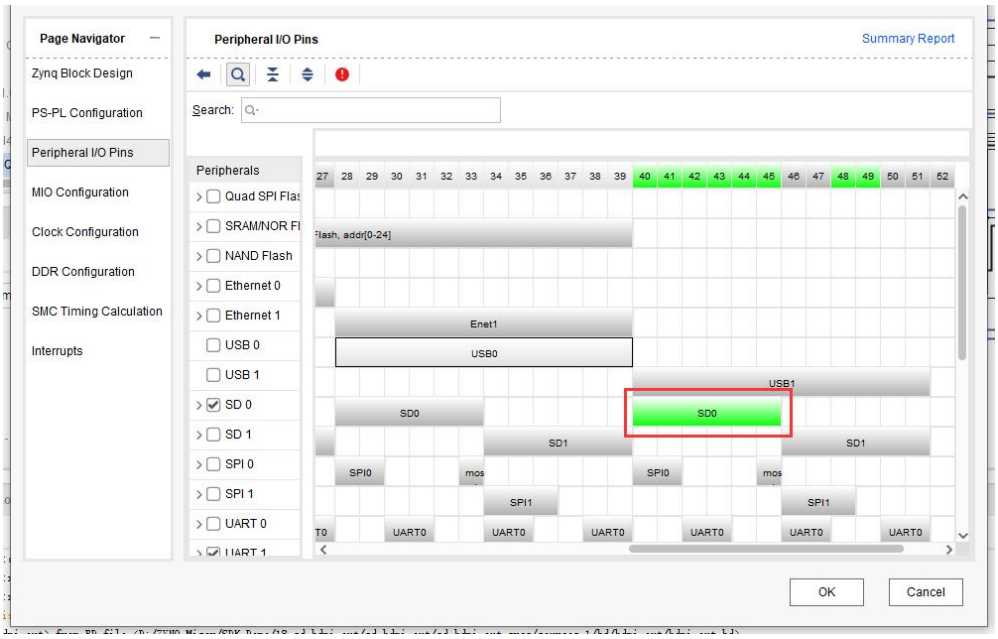


图 18.2.1 另存为新的工程

之后软件会自动打开这个工程，点击“Open Block Design”，打开设计文件。因为本实验要读取 SD 卡的数据，所以要使能 SD0。双击 ZYNQ7 Processing System



打开设置页面，使能 SD0，如下如图所示：

图 18.2.2 使能 SD0

之后点击“OK”并保存设置。接着重新生成顶层 HDL。点击左侧 Generate Bitstream 生成新的 BIT 流文件，如下图所示：

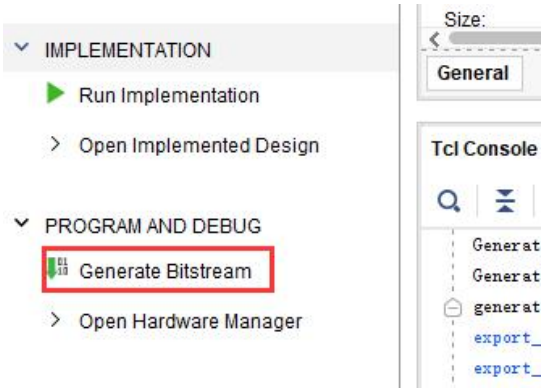
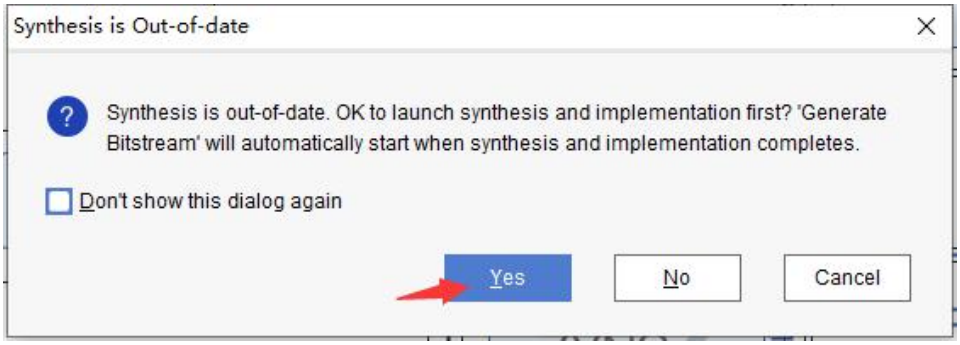


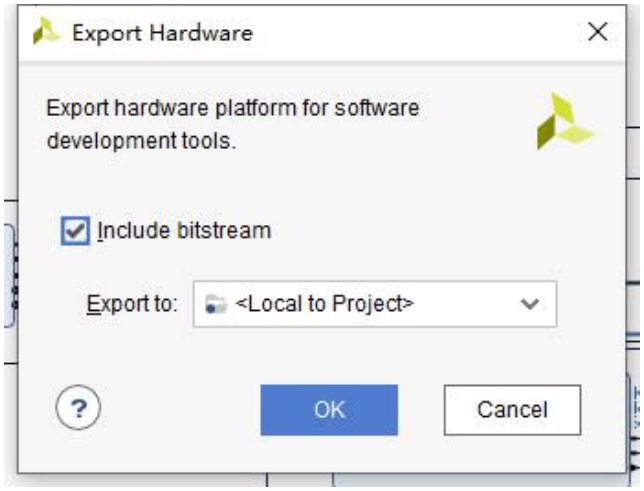
图 18.2.3 生成新的 Bit 流文件



在弹出的对话框中点击“yes”，对工程进行重新综合编译。

图 18.2.4 重新综合编译

生成完成后导出硬件描述文件到 SDK 进行软件设计，导出时勾选 Include



bitstream。

图 18.2.5 导出硬件描述文件

18.3 软件设计

在打开的 SDK 软件中新建一个应用工程并命名为“sd_hdmi_out”，工程模板选择空白。在本实验中，我们需要读取 SD 卡中的文件，所以添加文件系统以

支持对于文件的操作，点击 Modify this BSP’s Settings 进入设置页面，如下图所示：

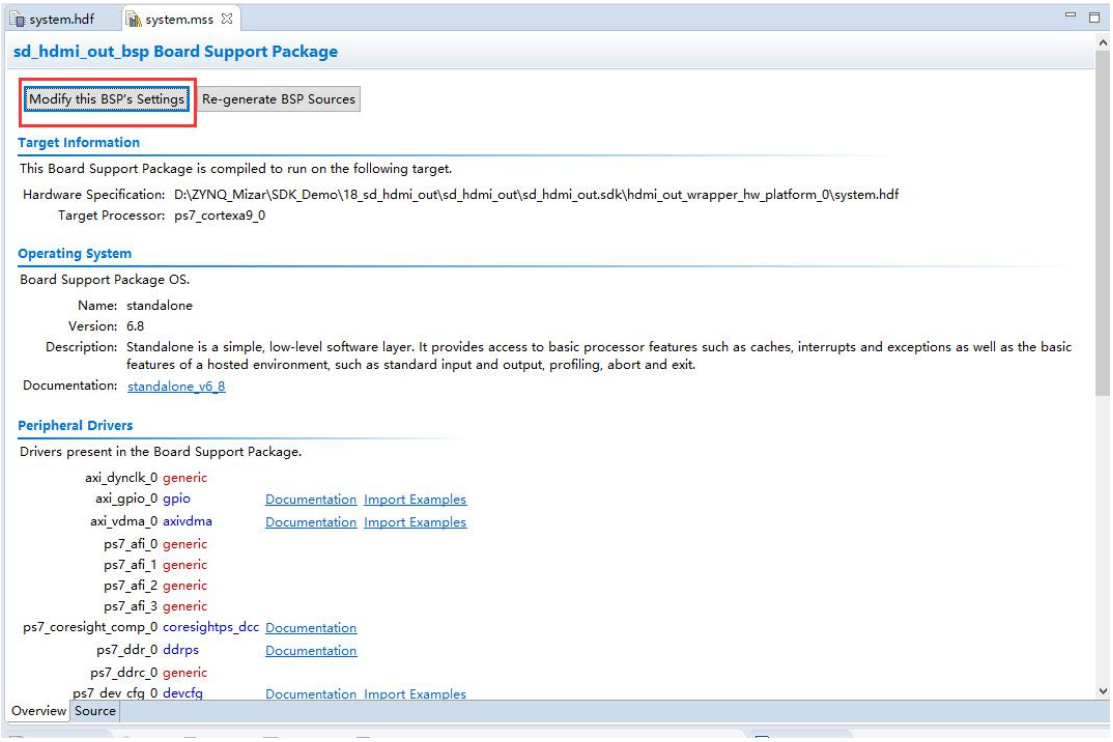


图 18.3.1 进入板级支持包设置页面

勾选下方的 xilffs，软件会自动添加相关文件到工程中，如下图所示：

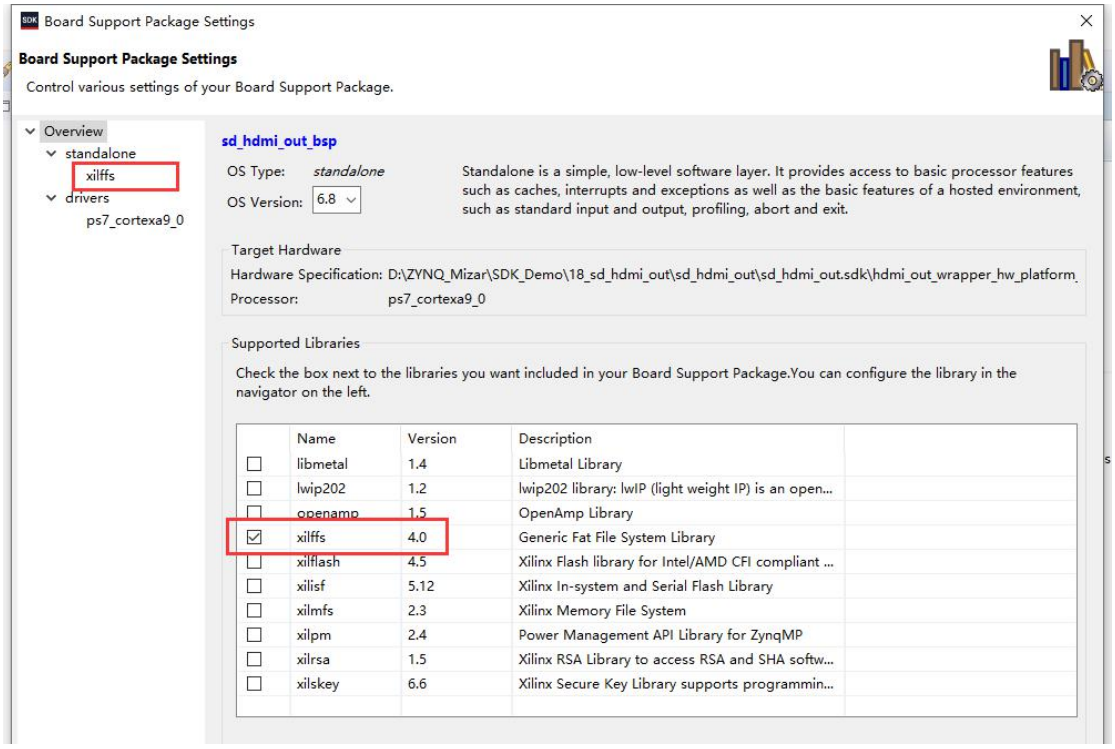
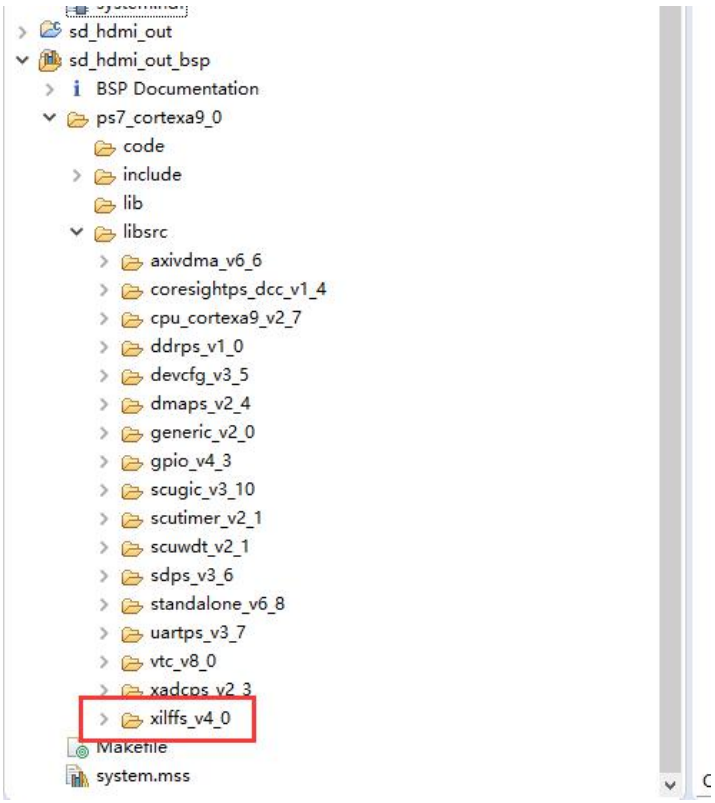


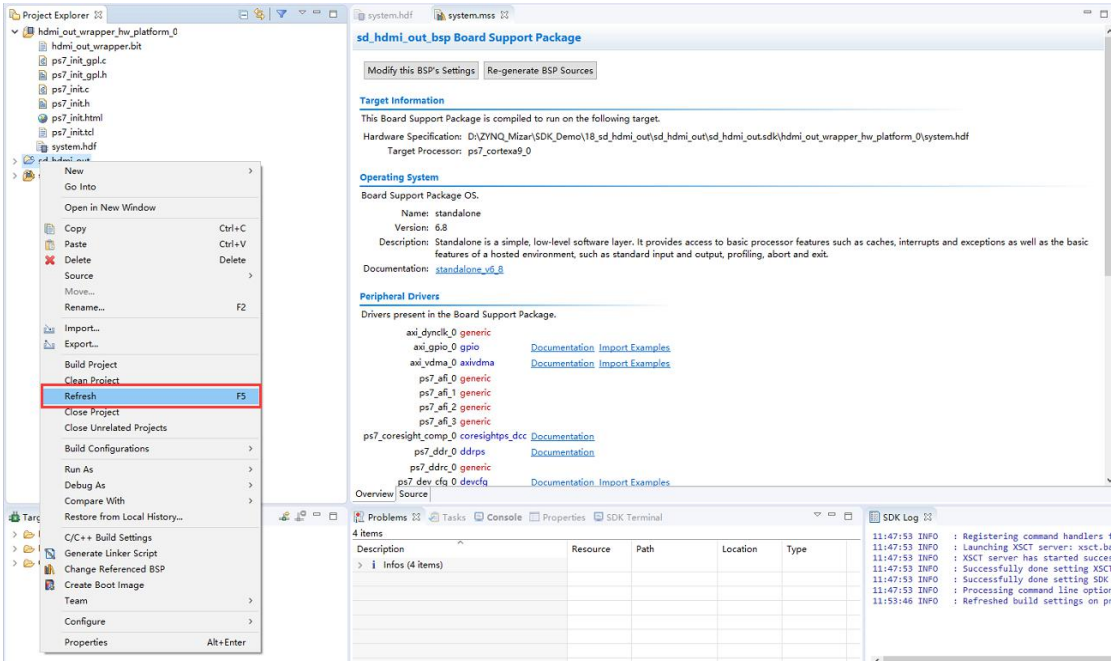
图 18.3.2 添加文件系统



之后可以看到相关文件已加入工程。

图 18.3.3 文件系统相关文件

之后打开资料例程目录 SDK_Demo\ 18_sd_hdmi_out\sd_hdmi_out\sd_hdmi_out.sdk，复制其中的 src 文件夹，保存到新建工程的相同位置，提示相同文件跳过不要替换。之后在 SDK 软件下右键点击工程，点击 Refresh 重新编译



即可，如下图所示：

图 18.3.4 编译工程

打开 display_demo.c 主程序文件，我们可以看到本实验在之前实验的基础上增加了读取图片文件的函数 bmp_read。在函数中使用 f_open 文件操作函数打开图片文件，然后读取前 54 字节的图像头文件，获取图像像素等属性信息，之后按照从下往上，从左往右的顺序读取像素数据。这样我们就获得了图片的全部像素数据，相当于之前实验用到的几个图片数据的头文件。

在成功获取到图片的数据之后，我们就可以按照之前实验的步骤，将数据写入 VDMA 的缓存中，再通过 HDMI 接口显示到屏幕上。

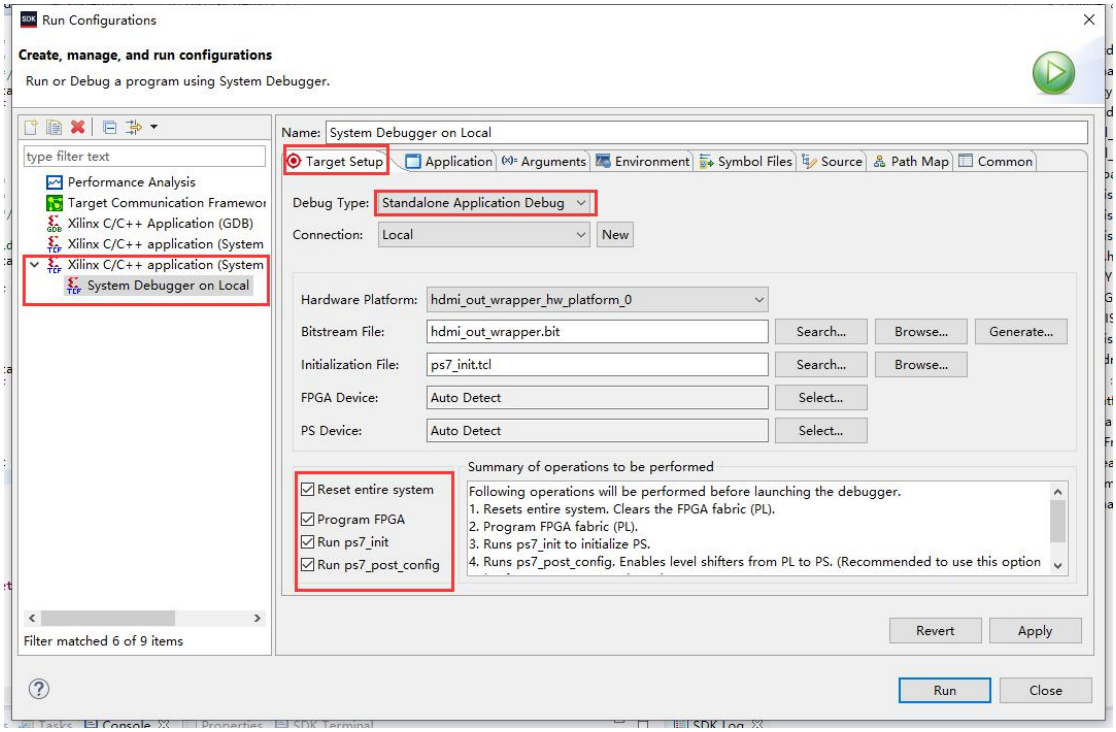
```
void bmp_read(char * bmp,u8 *frame,u32 stride)
{
    short y,x;
    short Ximage;
    short Yimage;
    u32 iPixelAddr = 0;
    FRESULT res;
    unsigned char TMPBUF[64];
    unsigned int br;         // File R/W count

    res = f_open(&fil, bmp, FA_OPEN_EXISTING | FA_READ);
    if(res != FR_OK)
    {
        return ;
    }
    res = f_read(&fil, TMPBUF, 54, &br);
    if(res != FR_OK)
    {
        return ;
    }
    Ximage=(unsigned short int)TMPBUF[19]*256+TMPBUF[18];
    Yimage=(unsigned short int)TMPBUF[23]*256+TMPBUF[22];
    iPixelAddr = (Yimage-1)*stride ;

    for(y = 0; y < Yimage ; y++)
    {
        f_read(&fil, read_line_buf, Ximage * 3, &br);
        for(x = 0; x < Ximage; x++)
        {
            frame[x * BYTES_PIXEL + iPixelAddr + 0] = read_line_buf[x * 3 + 0];
            frame[x * BYTES_PIXEL + iPixelAddr + 1] = read_line_buf[x * 3 + 1];
            frame[x * BYTES_PIXEL + iPixelAddr + 2] = read_line_buf[x * 3 + 2];
        }
        iPixelAddr -= stride;
    }
    f_close(&fil);
}
```

图 18.3.5 读取图片数据

程序编译完成之后，将例程文件夹下的示例图片拷贝到 SD 卡中，就可在开发板上下载验证了。

18.4 下载验证

连接好开发板的串口线和下载线，使用 HDMI 线连接 HDMI 接口和显示器。将拷贝好示例图片的 SD 卡插入卡槽。点击菜单栏 Run->Run Configuration 进入烧录设置页面，按照下图所示设置相关设置。



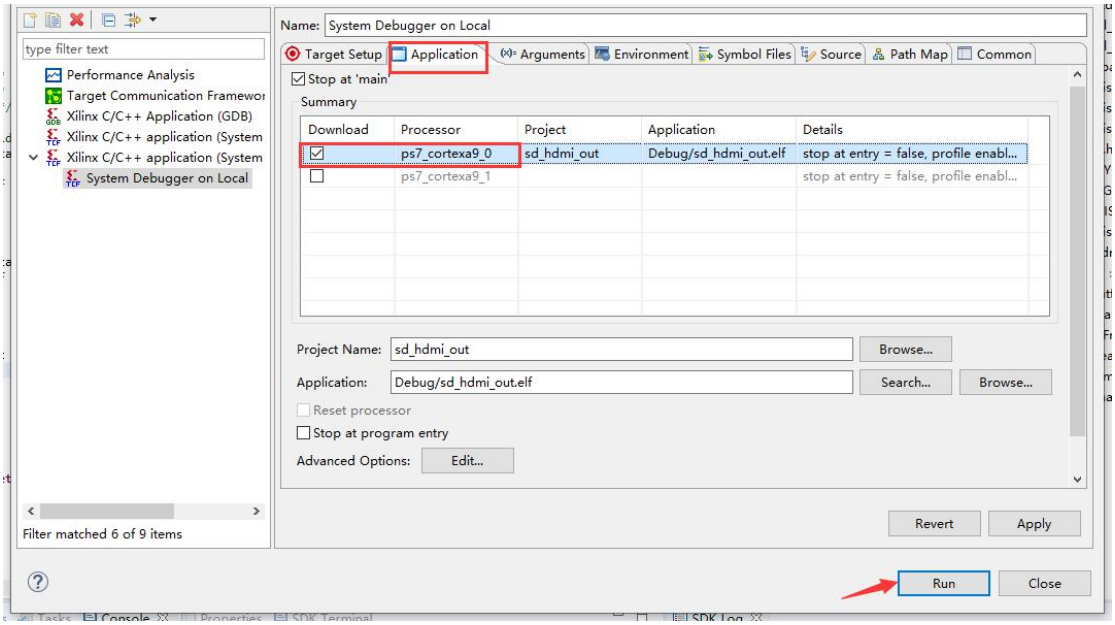


图 18.4.1 设置烧录选项

图 18.4.2 烧录设置

设置完成后点击“Run”烧录程序，即可看到屏幕上如下图所示显示示例图

片。

图 18.4.3 显示示例图片

18.5 章节小结

在本实验中，我们在之前实验的基础上结合文件系统对 SD 卡数据的读取，实现了读取 SD 卡图片数据显示在屏幕上的功能。希望读者可以在此基础上尝试实现串口控制显示不同图片等拓展实验。

第 19 章　　读取 SD 卡内 BMP 图片 LCD 显示实验

在之前的实验中我们实现了 SD 卡中的图片通过 HDMI 接口显示在屏幕上，HDMI 显示通常应用在视频图像等方面的应用。在其他的设备中，通常使用 LCD 屏幕来作为图像信息的输出端。本次实验我们就来学习怎样在 LCD 屏幕上显示图片。

本章包括以下几部分内容：

19.1 实验简介

19.2 硬件设计

19.3 软件设计

19.4 下载验证

19.5 章节小结

19.1 实验简介

在本次实验中我们同样使用 VDMA IP 核来实现对于 AXI4-Stream 类目标外设的高带宽直接存储器存取来读取 DDR 中的数据。关于 VDMA IP 的相关知识之前已经进行过讲解，此处不再赘述。VDMA 读取到数据之后通过 AXI4-Stream to Video Out IP 核将数据流转换成视频协议的数据流，之后在将该数据流转换为符合 RGB LCD 接口的时序就可以在 LCD 屏幕上显示了。

本次实验使用的 LCD 屏幕是我公司出产的一款 4.3 寸 LCD 显示屏，分辨率为 480*272，该屏幕的相关资料保存在“原理图及相关硬件资料”的文件夹中，客户可自行查阅。

19.2 硬件设计

首先双击打开 Vivado，新建一个 Vivado 工程并命名为“sd_bmp_lcd”，之后根据开发板的型号选择相应的芯片型号。创建完成之后点击 Create Block Design 新建硬件设计文件。在本次实验中需要用到一些自定义的 IP 核，我们首先将这些 IP 添加到 Vivado 中为之后的使用做准备。复制本实验例程文件内的 repo 文件夹，将其保存到新建工程的文件夹内，如下图所示：

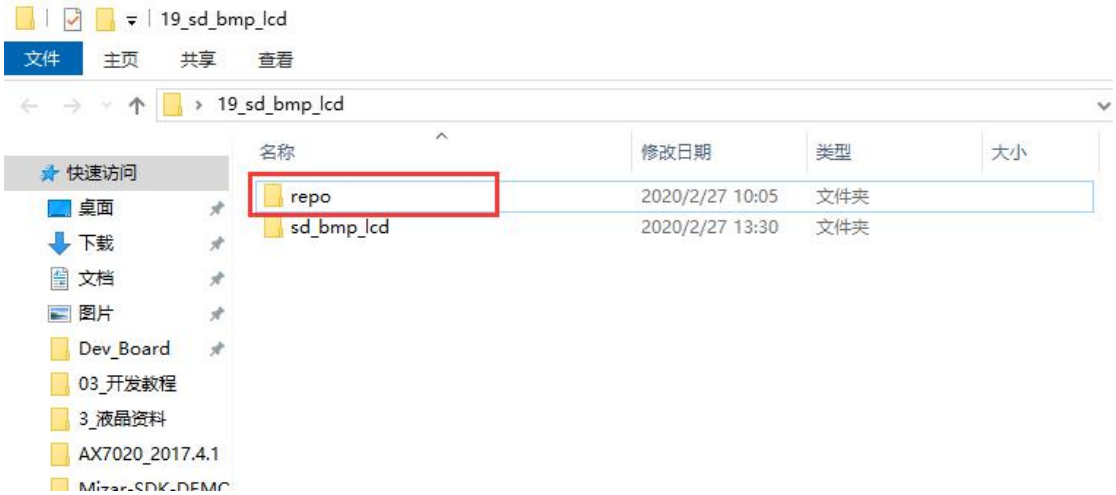
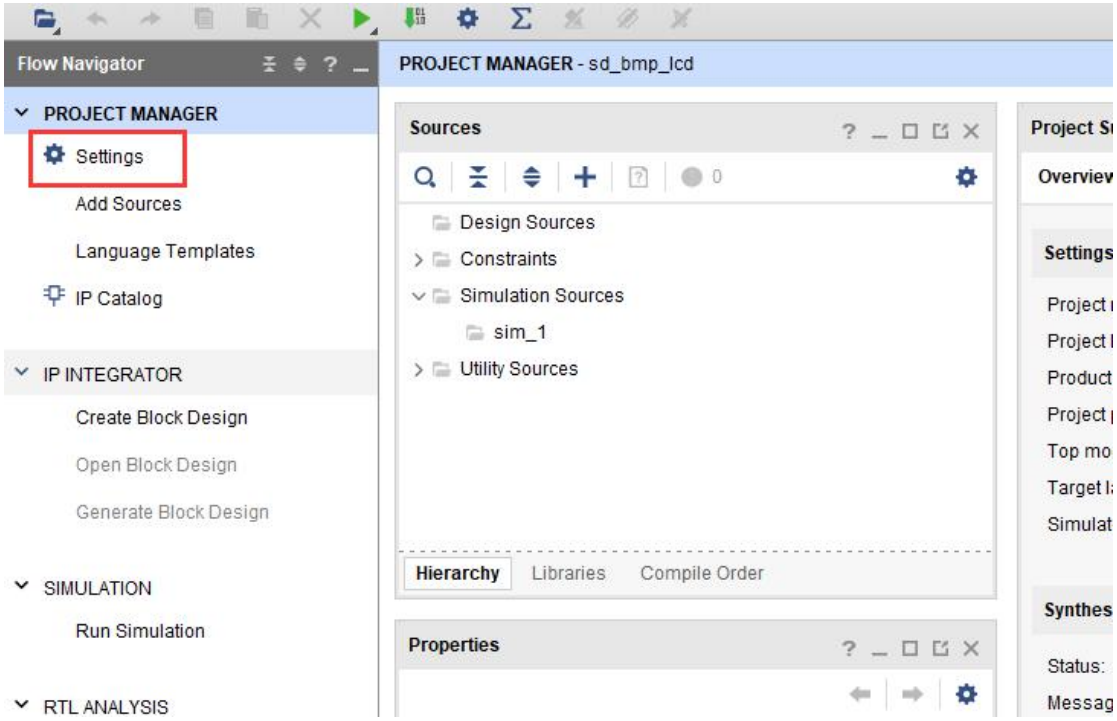


图 19.2.1 复制 repo 文件夹



之后在 Vivado 工程页面左侧点击 Settings 进入设置页面，如下图所示：

图 19.2.2 打开设置页面

展开 IP 下拉菜单，单击 Repository 打开添加界面，点击“+”选择 IP 核所在文件夹 repo 的路径，如下图所示：

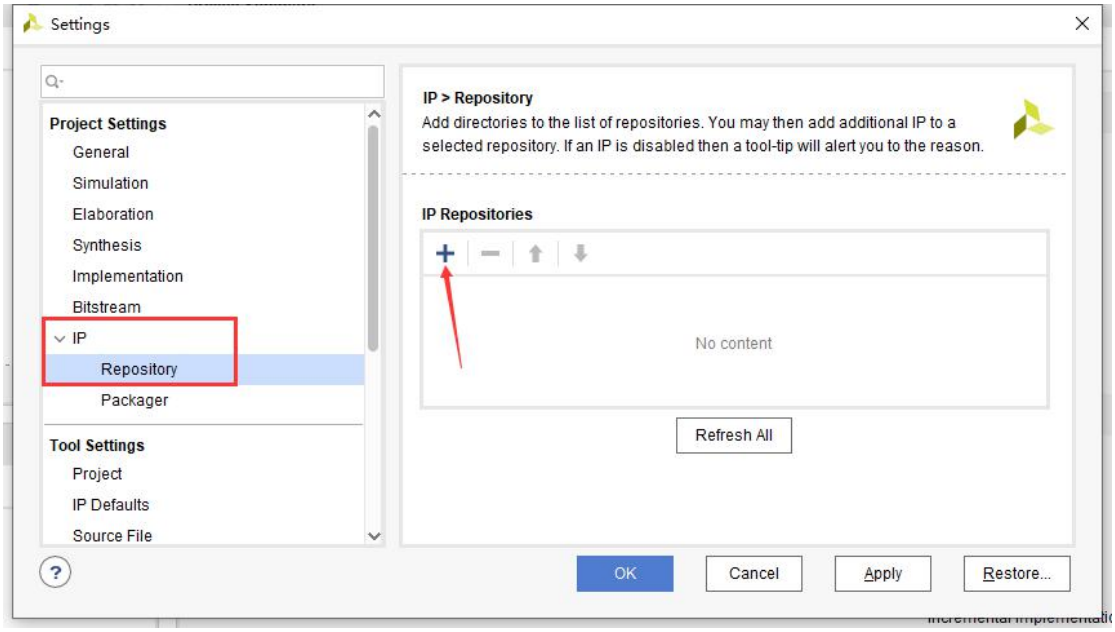


图 19.2.3 打开添加界面

选择完成后点击“OK”，软件会自动识别该文件夹下包含的 IP 核，如下图所示，点击“OK”将 IP 添加到软件中。

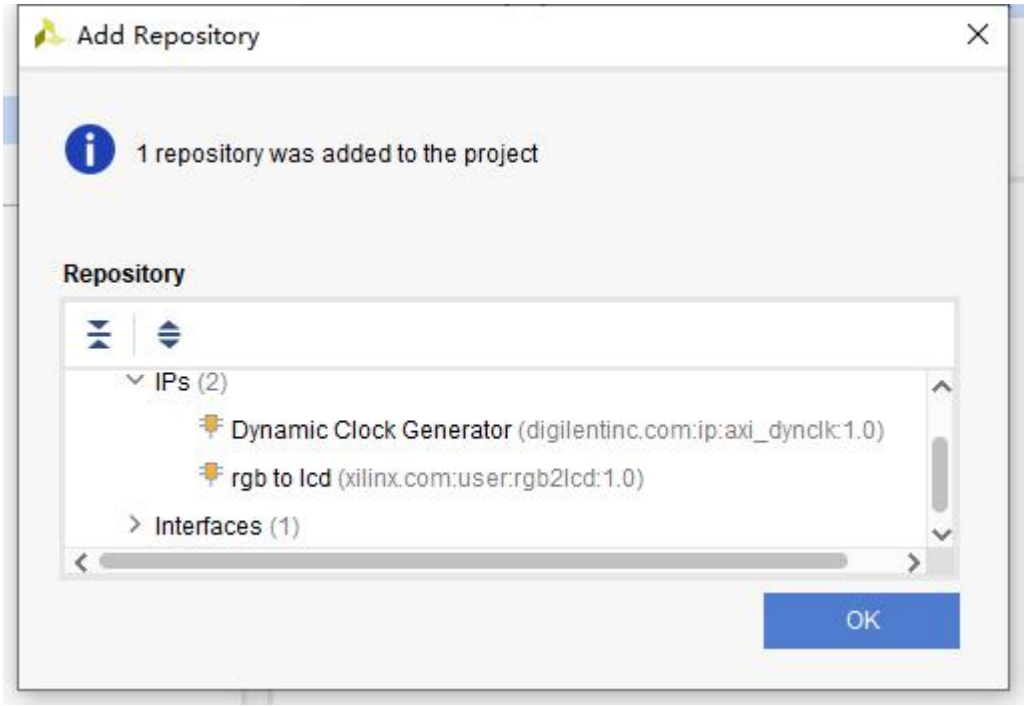
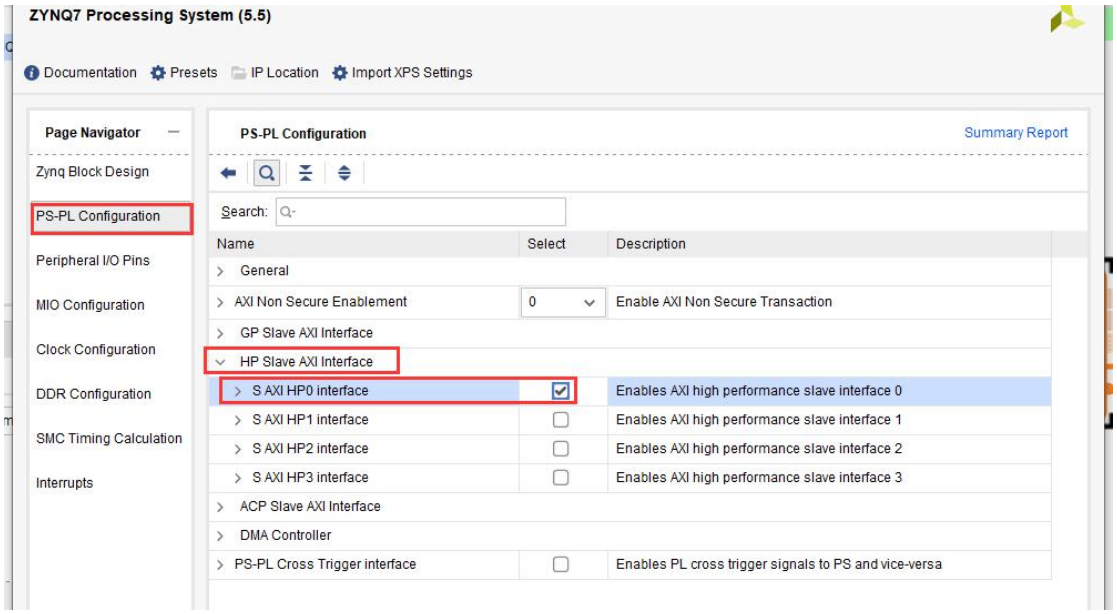


图 19.2.4 添加自定义 IP 核

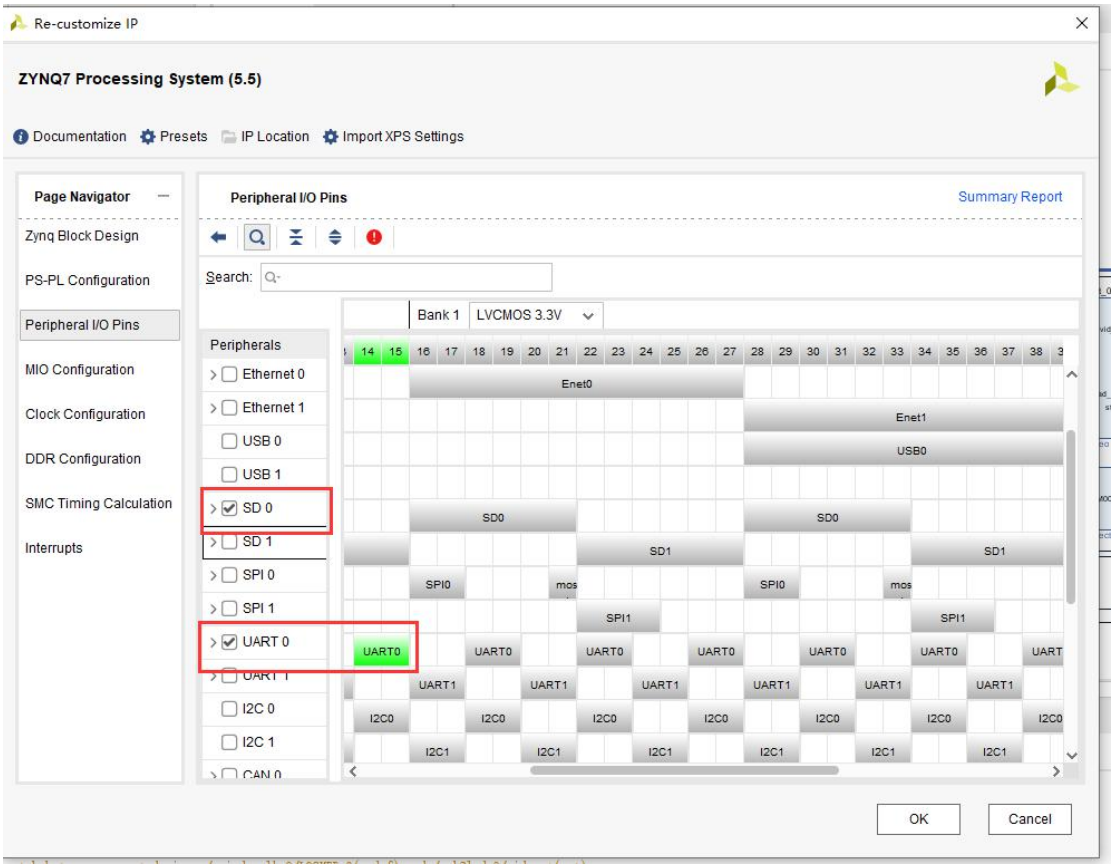
添加完成后返回 Diagram 窗口，点击“+”添加 ZYNQ7 Processing System IP，双击此 IP 进入配置页面。单击打开 PS-PL Configuration 栏，使能 S_AXI_HP0



Interface，如下图所示：

图 19.2.5 使能 S_AXI_HP0 Interface

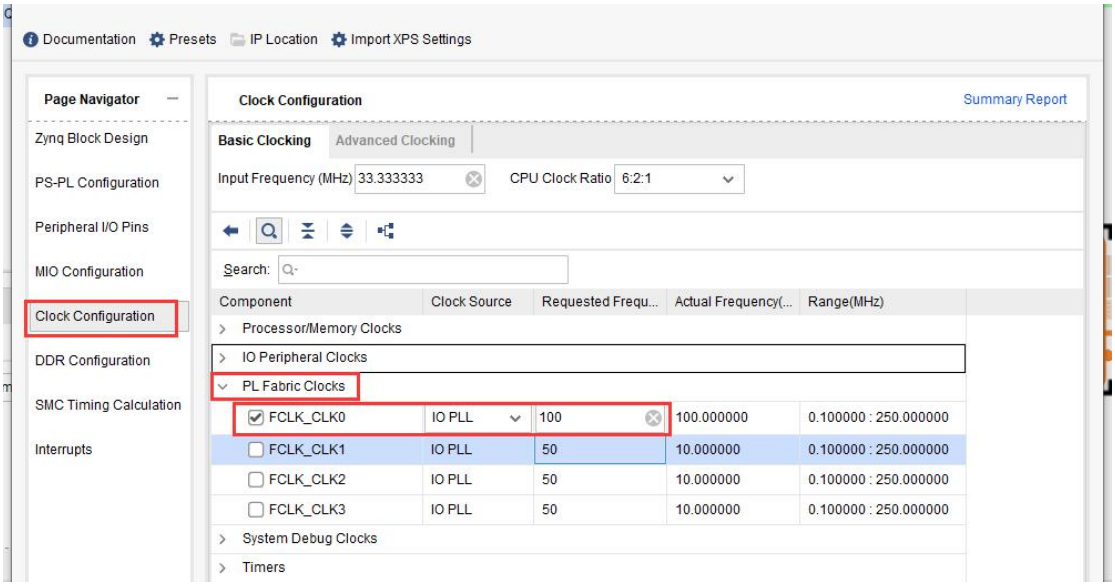
单击打开 Peripheral I/O Pins 栏，使能 SD0（对应 MIO40-MIO45）和 UART0，



如下图所示：

图 19.2.6 使能 SD0 和 UART1

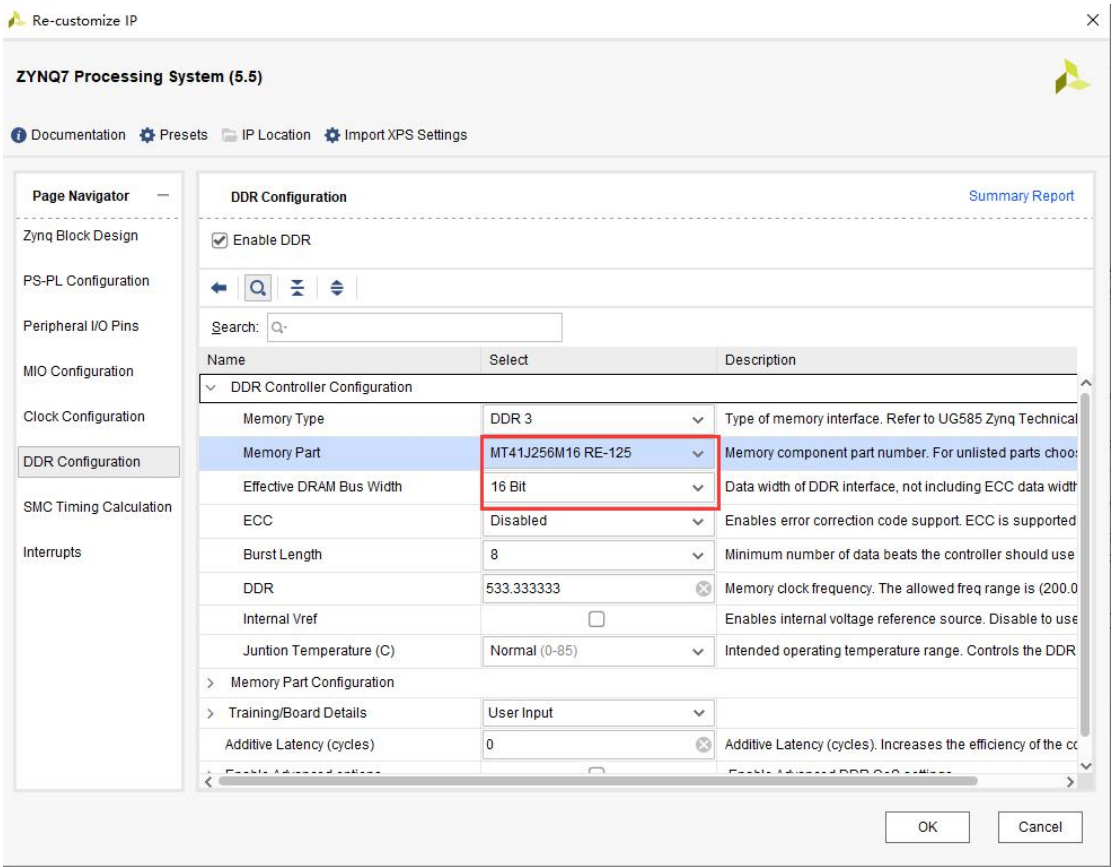
单击 Clock Configuration 打开时钟配置栏，配置 FCLK_CLK0 为 100M，如下



图所示：

图 19.2.7 配置时钟频率

单击 DDR Configuration 配置 DDR 控制器，选择 MT41J256M16RE-125，如



下图所示：

图 19.2.8 配置 DDR 控制器

配置完成后如下图所示：

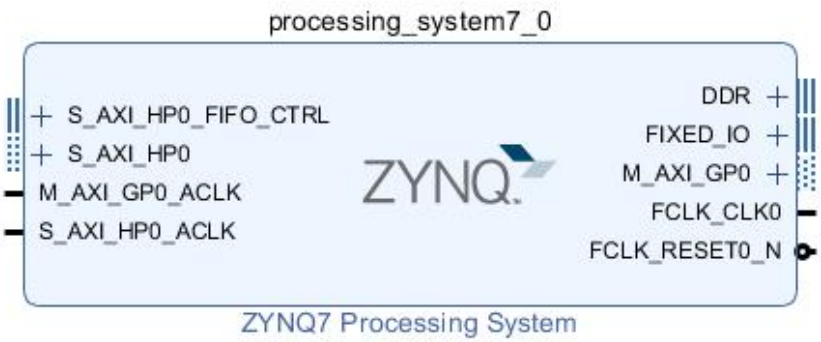
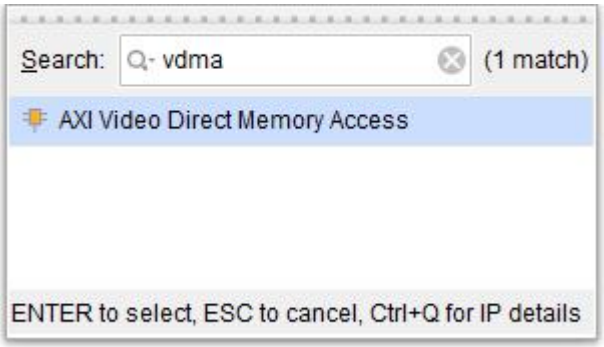


图 19.2.9 ZYNQ7 Processing System

点击“+”打开 IP 列表，在搜索框中输入“vdma”添加 VDMA IP，如下图



所示：

图 19.2.10 添加 VDMA IP

双击 IP 打开配置页面，如下图所示配置相关参数。Address Width 保持默认 32 即可。 Frame Buffers 选项可以选择 AXI VDMA 要处理的帧缓冲存储位置的数量。由于本次显示实验只显示一张图片，数据只需要写入一次，因此不需要设置多个帧缓存区域，这里设置为 1。因为本实验是从 DDR3 中读取数据输出

给 LCD，所以只需要勾选 Enable Read Channel 就可以了，无需勾选 Enable Write Channel。

Memory Map Data Width 选项可以为 MM2S 通道选择所需的 AXI4 数据宽度。此处保持默认 64 即可。

Read Burst Size 用于指定突发读的大小，此处选择 64。Stream Data Width 选项可以选择 MM2S 通道的 AXI4-Stream 数据宽度。 有效值是 8 的倍数，最大到 1024。 必须注意的是该值必须小于或等于 Memory Map Data Width。此处因输出数据格式为 RGB888，设置为 24。

Line Buffer Depth 选项可以选择 MM2S 通道的行缓冲深度（行缓冲区宽度为 stream data 的大小），此处设置为 2048 即可。

Advanced 界面用于配置读写通道的同步锁相模式，由于本次实验只有一个帧缓存，且只开启了读通道，所以同步锁相模式无需配置，保持默认就可以了。

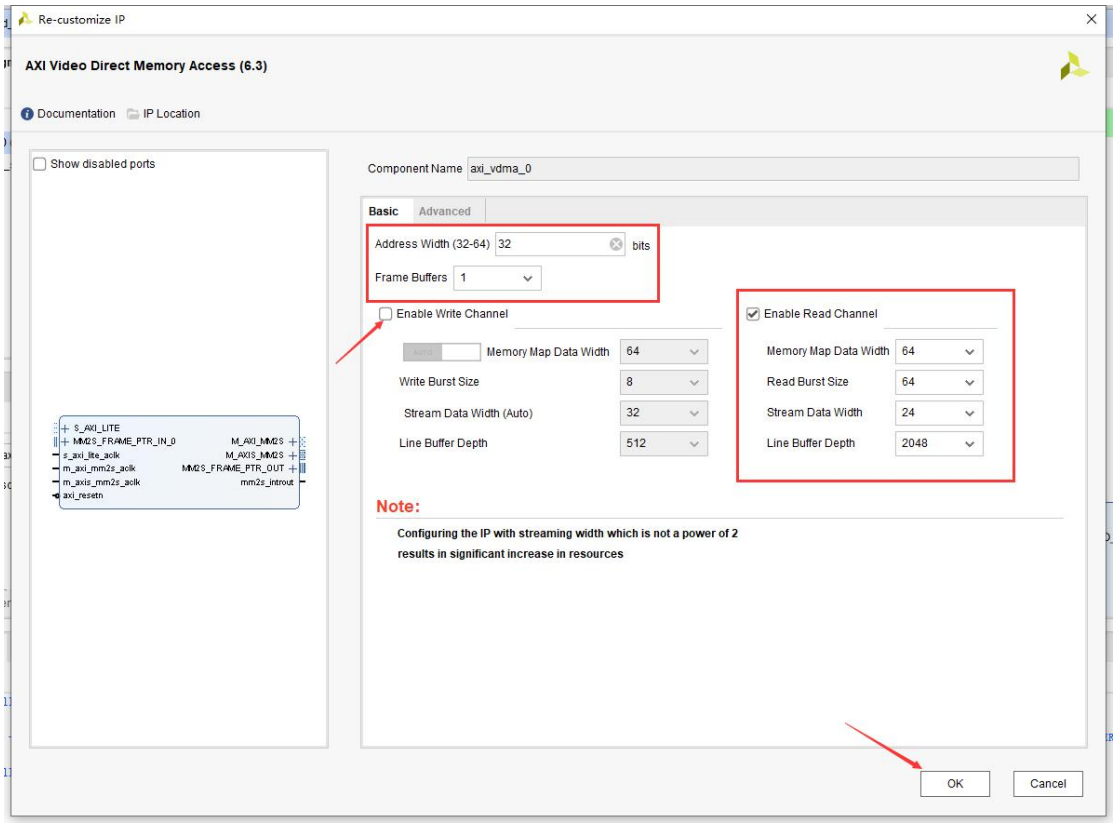


图 19.2.11 VDMA 相关参数

配置完成后 VDMA 如下图所示：

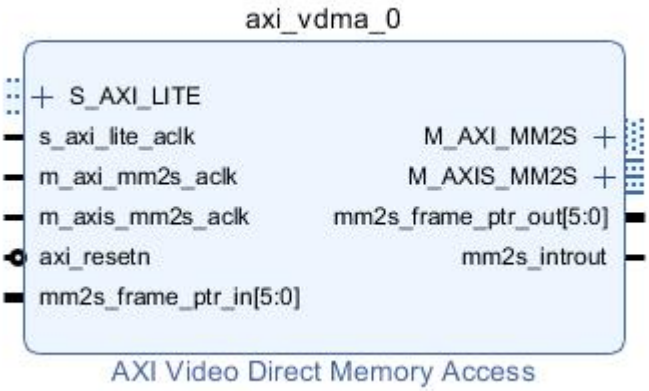


图 19.2.12 VDMA IP

单击“+”，打开 IP 列表，在搜索框输入“timing”，添加 Video Timing Controller IP 核，如下图所示：

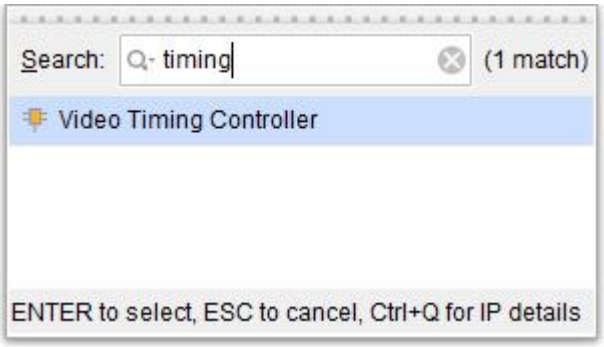
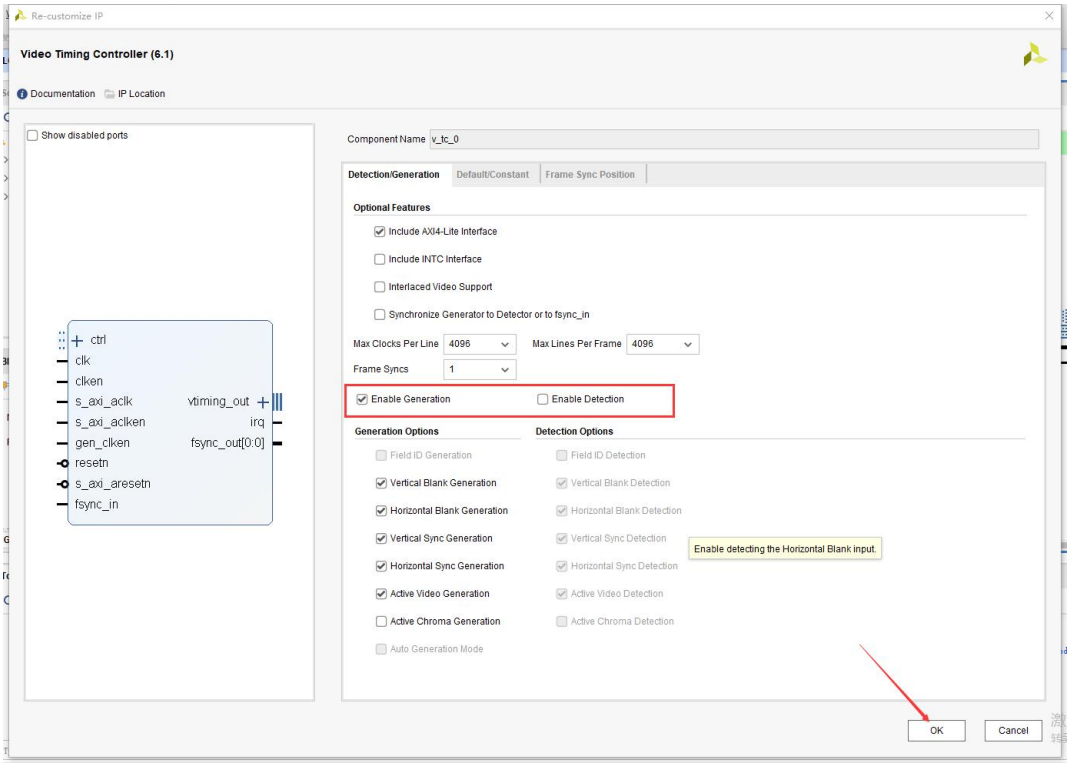
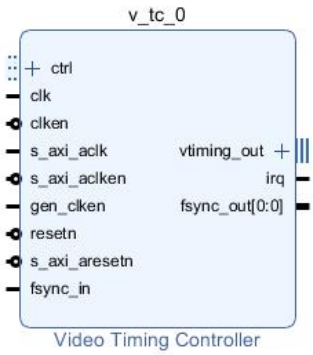


图 19.2.13 添加视频时序控制器



双击 IP 打开配置页面，如下图所示配置，取消勾选 Enable Detection。

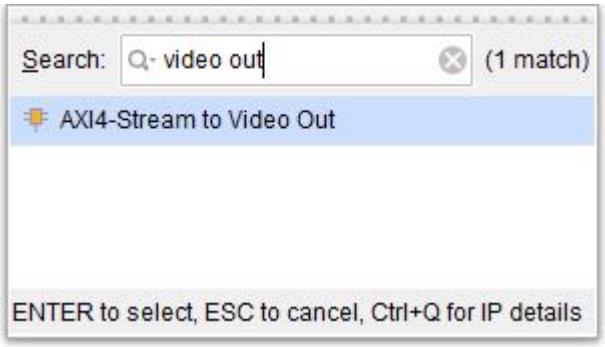
图 19.2.14 配置相关设置



配置完成后如下图所示：

图 19.2.15 Video Timing Controller IP

单击“+”，打开 IP 列表，在搜索框中输入“video out”，添加视频输出控



制器，如下图所示：

图 19.2.16 添加视频输出控制器

双击该 IP 打开配置页面，此处选择 Indeopendent（独立时钟），不使用 Common（公共时钟），如下图所示：

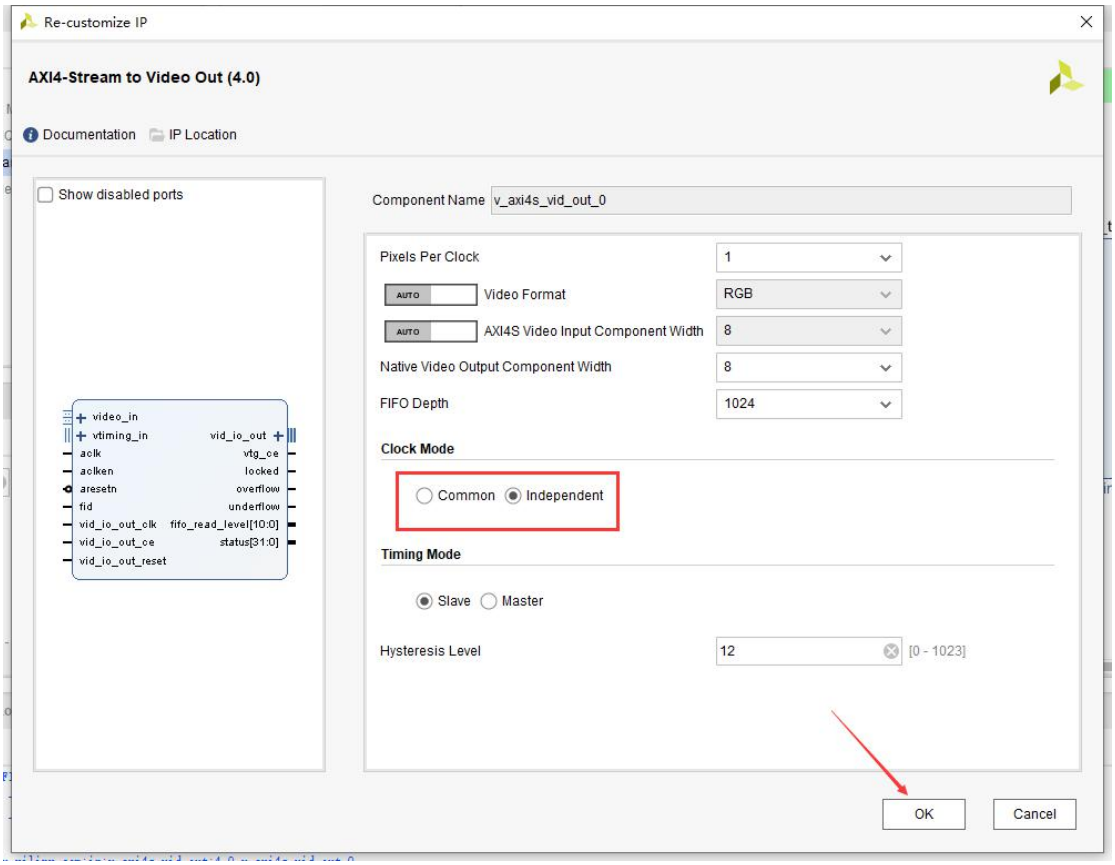


图 19.2.17 配置相关设置

配置完成后如下图所示：

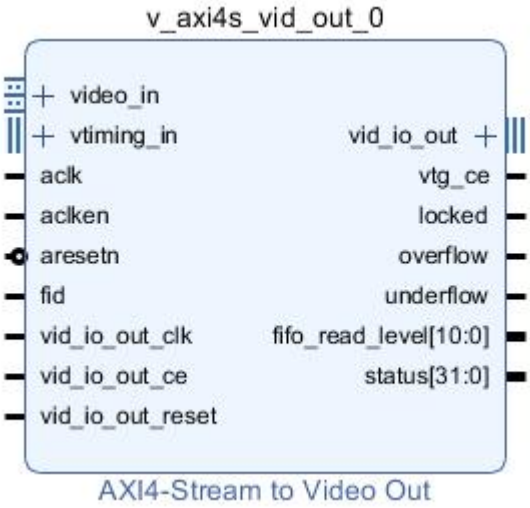
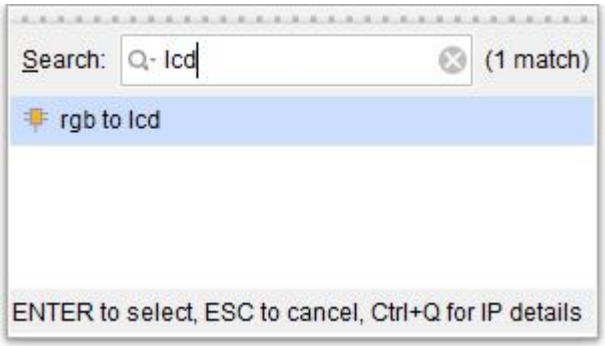


图 19.2.18 AXI4-Steram to Video Out IP

单击“+”打开 IP 列表，在搜索框中输入“lcd”添加自定义 IP 核 rgb to lcd，



如下图所示：

图 19.2.19 添加 rgb to lcd IP 核

双击 IP 打开配置界面。其中 Reddepth、 Greendepth 和 Buledepth 用于设置输出的 Red、Green 和 Blue 颜色的位数，这里我们使用 RGB888，所以，分别设置为 8、8、8。Vid In Data Width 用于指定输入的 RGB 总线宽度，为 24 位。Vid Out Data Width 用于指定输出的 RGB 总线宽度，为 24 位。此处无需修改，保持默认即可，如下图所示：

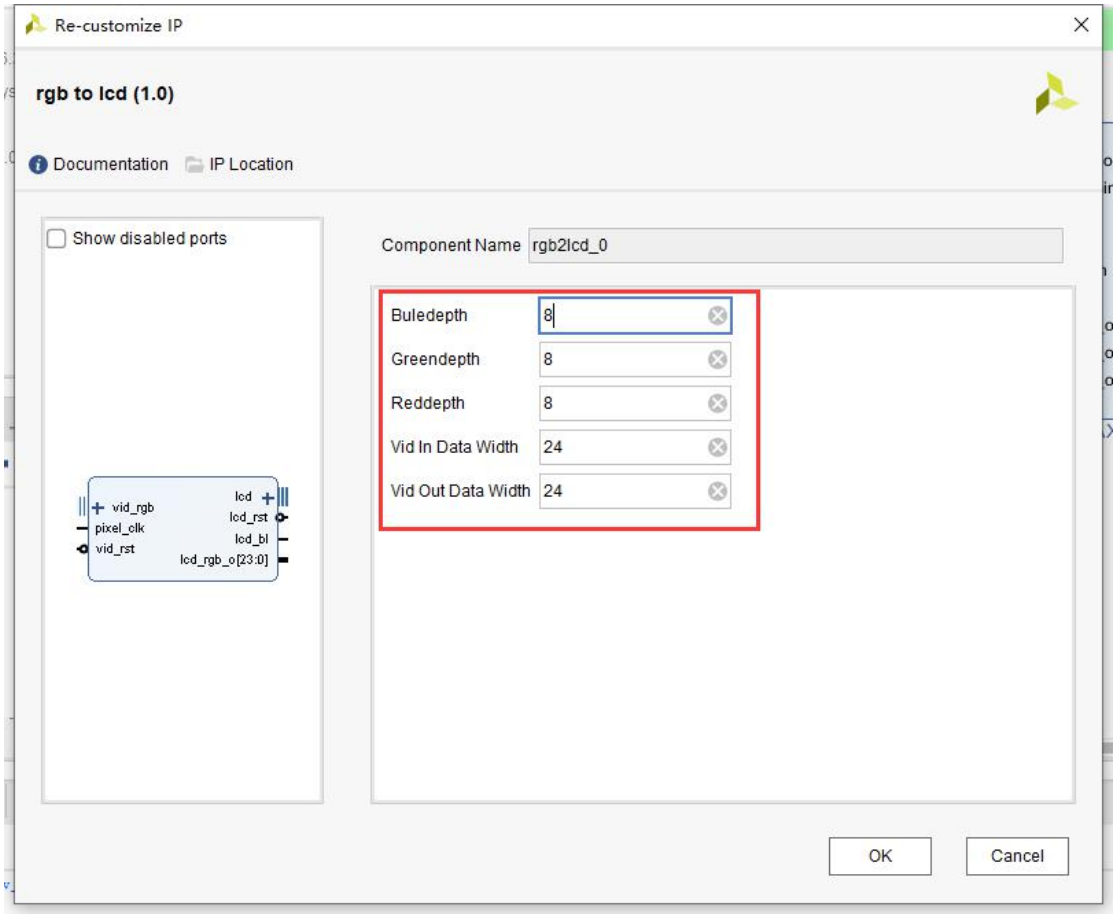


图 19.2.20 IP 核配置

单击“+”打开 IP 列表，在搜索框中输入“dynamic”添加动态时钟控制器，如下图所示：

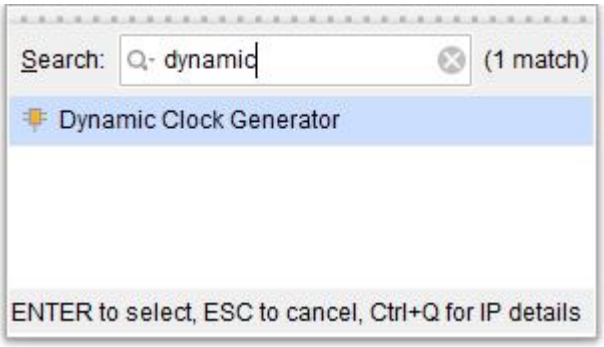


图 19.2.21 添加动态时钟控制器 IP

该 IP 无需配置，保持默认即可。

Vivado 的自动连线功能可能不符合我们的需求，所以在自动连线之前手动连接部分接口，确保连线的正确。按照下图所示手动连接部分接口：

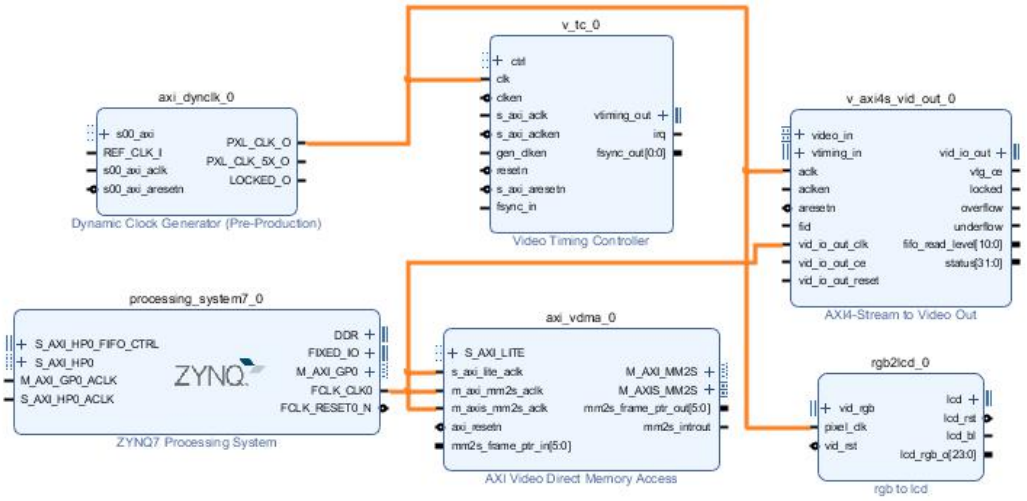
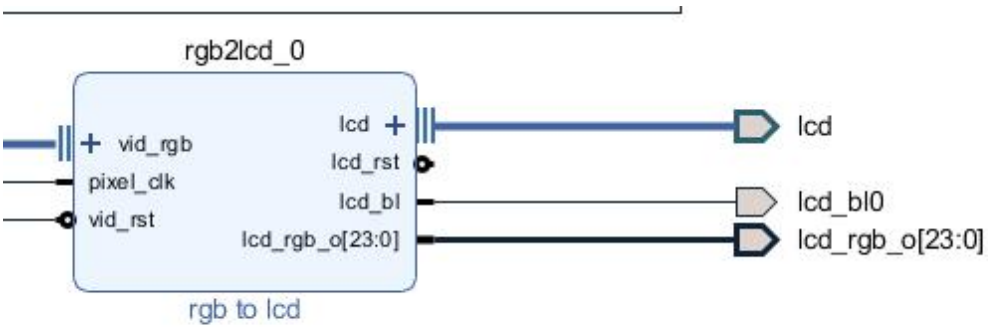


图 19.2.22 手动连接接口

接下来点击界面上方“Run Block Automation”，在弹出的对话框左侧确认勾选 processing_system7_0，然后点击“OK”。这一步骤会将 ZYNQ7 PS 模块的 DDR



和 FIXED_IO 端口引出。之后将 rgb to lcd IP 的相关端口引出，如下图所示：

图 19.2.23 引出相关端口

然后点击“Run Connnection Automation”，下面列出了会自动连接的模块及其接口，勾选“All Automation”，然后点击“OK”按钮。如下图所示：

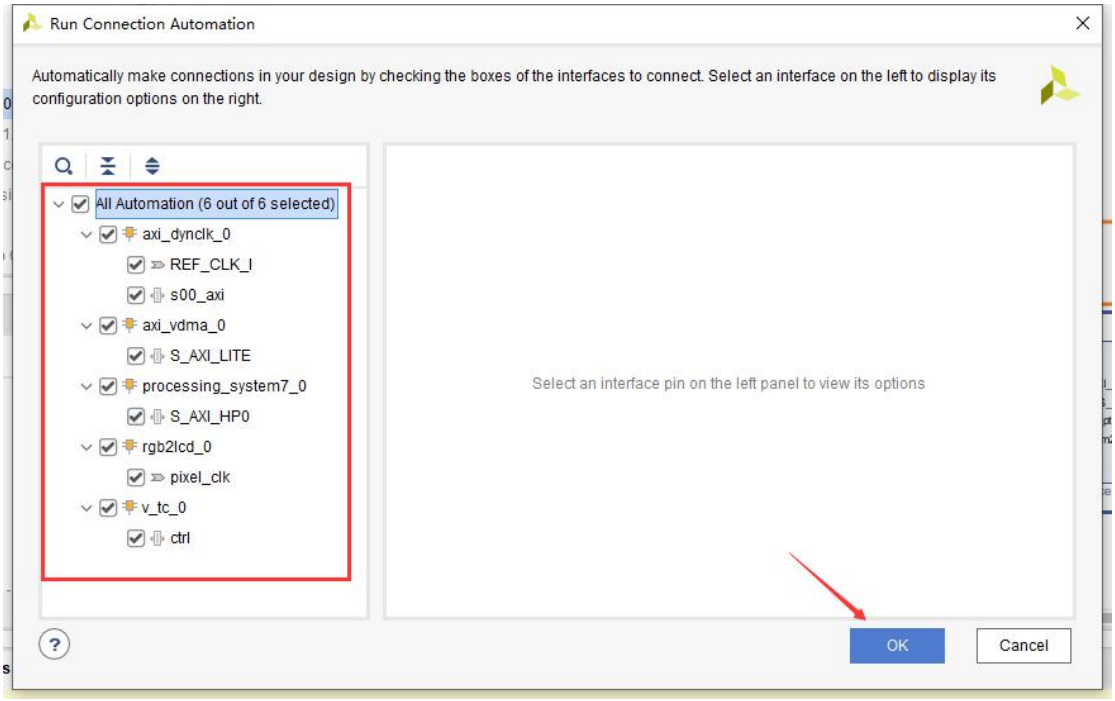


图 19.2.24 自动连接所有模块及端口

连接完成后部分信号无法自动连接，需手动连接，需要连接的线如下图所示：

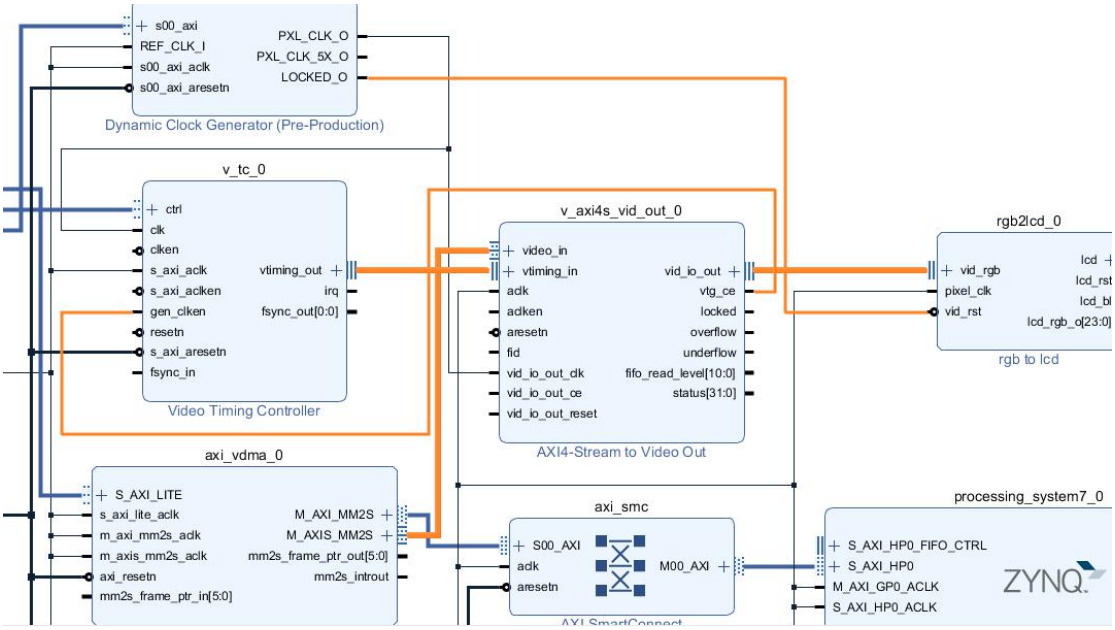


图 19.2.25 手动连接信号

连接完成后点击 Regenerate Layout 重新布局，布局完成后整体设计如下图所示：

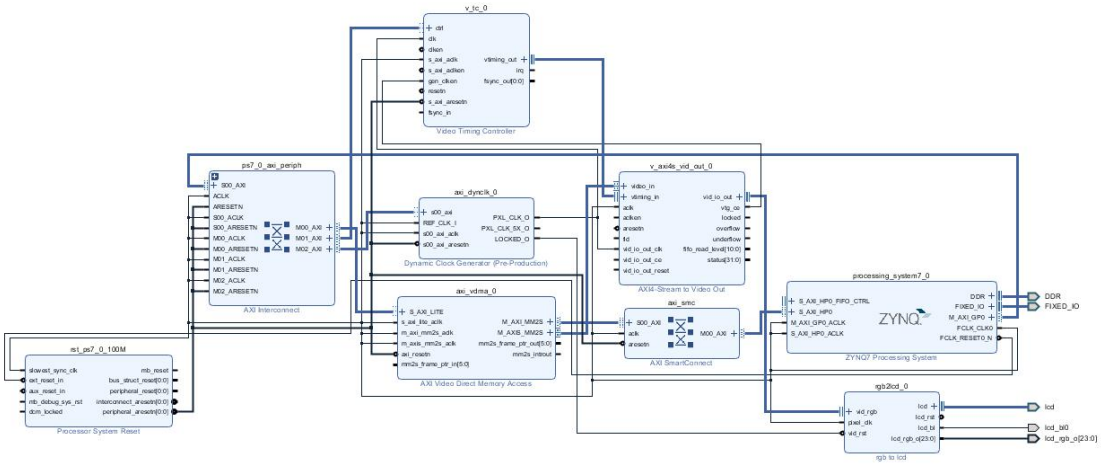
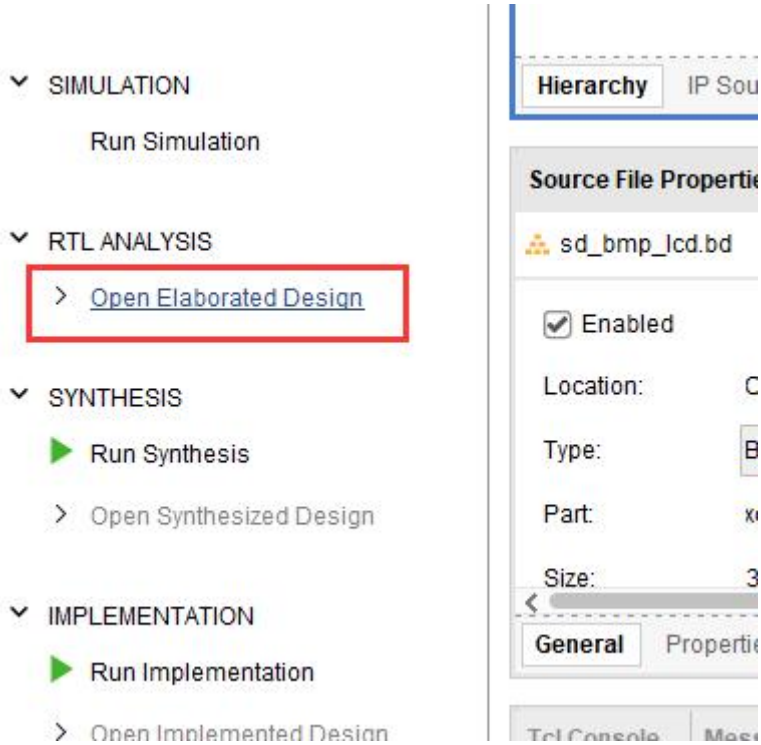


图 19.2.26 系统整体设计结构图

之后验证设计，没有错误之后保存并生成顶层 HDL。接着进行引脚约束，



点击左侧“Open Elaborated Design”打开引脚约束窗口，如下图所示：

图 19.2.27 Open Elaborated Design

引脚约束如下图所示：

```
27 set_property IOSTANDARD LVCMOS33 [get_ports {lcd_rgb_o[2]}]
28 set_property IOSTANDARD LVCMOS33 [get_ports {lcd_rgb_o[1]}]
29 set_property IOSTANDARD LVCMOS33 [get_ports {lcd_rgb_o[0]}]
30 set_property PACKAGE_PIN R14 [get_ports lcd_lcd_de]
31 set_property PACKAGE_PIN W15 [get_ports lcd_lcd_hs]
32 set_property PACKAGE_PIN T10 [get_ports lcd_lcd_pclk]
33 set_property PACKAGE_PIN V15 [get_ports lcd_lcd_vs]
34 set_property PACKAGE_PIN T11 [get_ports lcd_bl0]
35 set_property PACKAGE_PIN W16 [get_ports {lcd_rgb_o[0]}]
36 set_property PACKAGE_PIN V16 [get_ports {lcd_rgb_o[1]}]
37 set_property PACKAGE_PIN R18 [get_ports {lcd_rgb_o[2]}]
38 set_property PACKAGE_PIN T17 [get_ports {lcd_rgb_o[3]}]
39 set_property PACKAGE_PIN W13 [get_ports {lcd_rgb_o[4]}]
40 set_property PACKAGE_PIN V12 [get_ports {lcd_rgb_o[5]}]
41 set_property PACKAGE_PIN T15 [get_ports {lcd_rgb_o[6]}]
42 set_property PACKAGE_PIN T14 [get_ports {lcd_rgb_o[7]}]
43 set_property PACKAGE_PIN P18 [get_ports {lcd_rgb_o[8]}]
```

```
1 set_property IOSTANDARD LVCMOS33 [get_ports lcd_lcd_hs]
2 set_property IOSTANDARD LVCMOS33 [get_ports lcd_lcd_vs]
3 set_property IOSTANDARD LVCMOS33 [get_ports lcd_lcd_de]
4 set_property IOSTANDARD LVCMOS33 [get_ports lcd_lcd_pclk]
5 set_property IOSTANDARD LVCMOS33 [get_ports lcd_bl0]
6 set_property IOSTANDARD LVCMOS33 [get_ports {lcd_rgb_o[23]}]
7 set_property IOSTANDARD LVCMOS33 [get_ports {lcd_rgb_o[22]}]
8 set_property IOSTANDARD LVCMOS33 [get_ports {lcd_rgb_o[21]}]
```

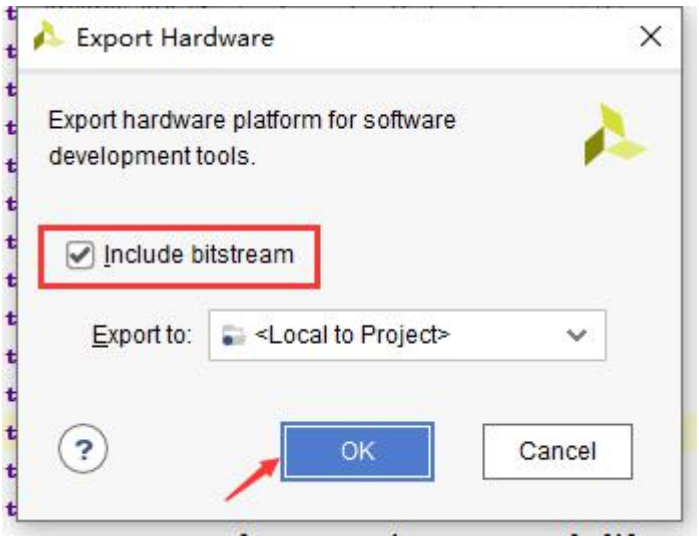
```
44 set_property PACKAGE_PIN Y17 [get_ports {lcd_rgb_o[9]}]
45 set_property PACKAGE_PIN Y16 [get_ports {lcd_rgb_o[10]}]
46 set_property PACKAGE_PIN V18 [get_ports {lcd_rgb_o[11]}]
47 set_property PACKAGE_PIN N17 [get_ports {lcd_rgb_o[12]}]
48 set_property PACKAGE_PIN V17 [get_ports {lcd_rgb_o[13]}]
49 set_property PACKAGE_PIN Y14 [get_ports {lcd_rgb_o[14]}]
50 set_property PACKAGE_PIN W14 [get_ports {lcd_rgb_o[15]}]
51 set_property PACKAGE_PIN Y19 [get_ports {lcd_rgb_o[16]}]
52 set_property PACKAGE_PIN Y18 [get_ports {lcd_rgb_o[17]}]
53 set_property PACKAGE_PIN W19 [get_ports {lcd_rgb_o[18]}]
54 set_property PACKAGE_PIN W18 [get_ports {lcd_rgb_o[19]}]
55 set_property PACKAGE_PIN U17 [get_ports {lcd_rgb_o[20]}]
56 set_property PACKAGE_PIN T16 [get_ports {lcd_rgb_o[21]}]
57 set_property PACKAGE_PIN R17 [get_ports {lcd_rgb_o[22]}]
58 set_property PACKAGE_PIN R16 [get_ports {lcd_rgb_o[23]}]
```

图 19.2.28 引脚约束

在本例程中，我们将 LCD 屏幕插到 GPIO1 插座上，根据原理图得到上图所示的引脚约束。如使用其他型号的开发板或插到其他 GPIO 插座上使用，请对照开发板原理图和 LCD 原理图确定引脚约束，此处引脚较多，请仔细查看，保证分配正确。

完成引脚约束之后点击“Generate Bitstream”，对设计进行综合、实现、并生成 Bitstream 文件。

在生成 Bitstream 之后， 在菜单栏中选择 File > Export > Export hardware



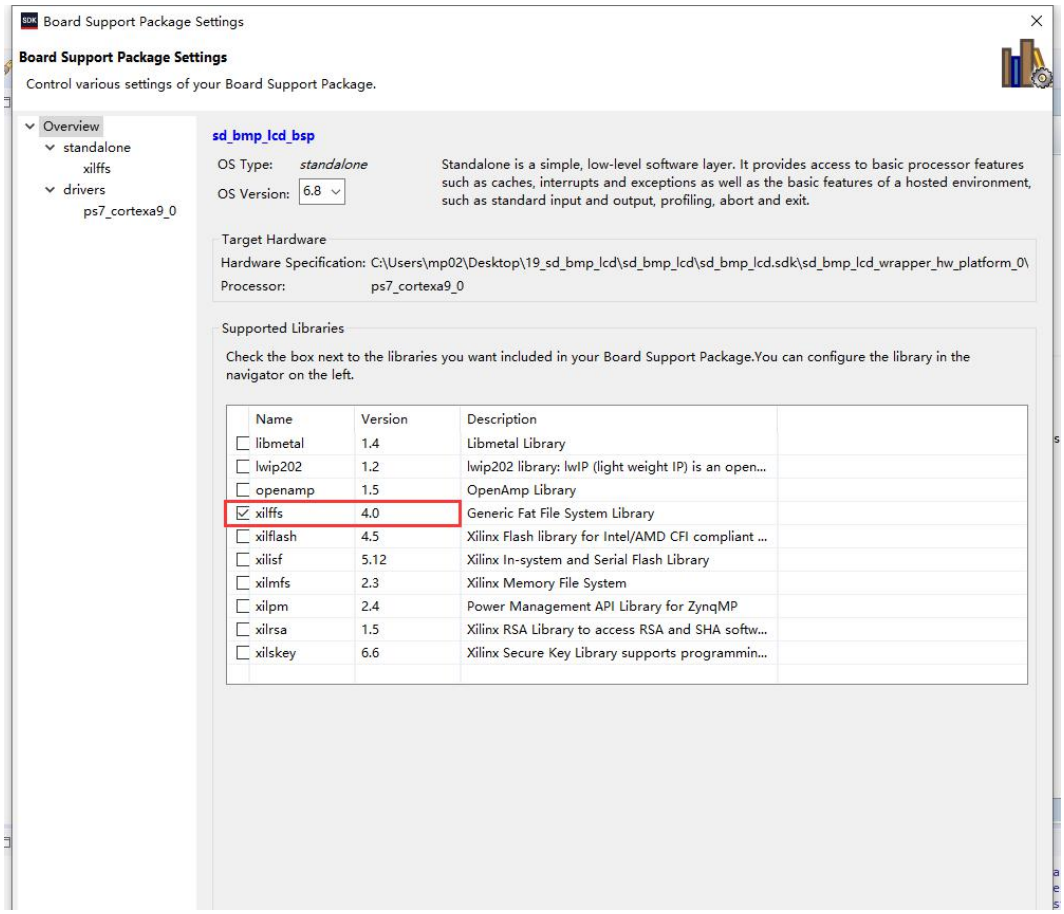
导出硬件，并在弹出的对话框中，勾选“Include bitstream”。如下图所示：

图 19.2.29 勾选“Include bitstream”

然后在菜单栏选择 File > Launch SDK，启动 SDK 软件。

19.3 软件设计

在打开的 SDK 软件中创建新的空白的应用工程并命名为“sd_bmp_lcd”，创



建完成后打开 sd_bmp_lcd_bsp 设置界面，添加文件系统库函数，如下图所示：

图 19.3.1 添加文件系统库函数

之后将源文件拷贝到 src 文件夹下，需要拷贝的文件如下图所示：

图 19.3.2 源文件

主函数的代码如下：

```c
int main(void)
{
    //设置分辨率
    vd_mode = VMODE_480x272;
    //配置VDMA
    run_vdma_frame_buffer(&vdma, VDMA_ID, vd_mode.width, vd_mode.height,
                          frame_buffer_addr,0, 0,ONLY_READ);

    //初始化Display controller
    DisplayInitialize(&dispCtrl, DISP_VTC_ID, DYNCLK_BASEADDR);
    //设置VideoMode
    DisplaySetMode(&dispCtrl, &vd_mode);
    DisplayStart(&dispCtrl);

    //读取SD卡图片并显示
    load_sd_bmp((u8*)frame_buffer_addr);

    return 0;
}
```

在主函数中，首先设置屏幕的分辨率，然后设置 VDMA 的相关参数并初始化显示控制器。之后读取 SD 卡中的图片并将其数据写入 DDR 中。

读取 SD 卡中图片数据的函数如下图所示：

```c
void load_sd_bmp(u8 *frame)
{
    static  FATFS fatfs;
    FIL     fil;
    u8      bmp_head[54];
    UINT    *bmp_width,*bmp_height,*bmp_size;
    UINT    br;
    int     i;

    f_mount(&fatfs,"",1);//挂载文件系统

    f_open(&fil,"shatan.bmp",FA_READ);  //打开文件
    xil_printf("open bmp\n\r");
    f_lseek(&fil,0);//移动文件读写指针到文件开头

    f_read(&fil,bmp_head,54,&br);//读取BMP文件头

    //BMP图片的分辨率和大小
    bmp_width  = (UINT *)(bmp_head + 0x12);
    bmp_height = (UINT *)(bmp_head + 0x16);
    bmp_size   = (UINT *)(bmp_head + 0x22);
    xil_printf("bmp information:\n\r");
    xil_printf(" width  = %d,\n\r height = %d,\n\r size   = %d bytes \n\r",
            *bmp_width,*bmp_height,*bmp_size);

    //读出图片，写入DDR
    for(i=*bmp_height-1;i>=0;i--){
        f_read(&fil,frame+i*(*bmp_width)*3,(*bmp_width)*3,&br);
    }
    //关闭文件
    f_close(&fil);

    Xil_DCacheFlush();      //刷新Cache，将数据更新至DDR3中
    xil_printf("display bmp\n\r");

}
```

源文件复制完成后重新编译工程，没有报错就可以烧录到开发板进行验证了。

19.4 下载验证

首先将 LCD 显示屏插入 GPIO1 插座中，注意插接的方向。然后我们将使用 TYPE-C 线与 JTAG 接口连接，另外一端与电脑连接。然后使用 TYPE-C 线连接线将开发板的 UART 接口与电脑连接，用于串口通信。显示屏连接如下图所示：

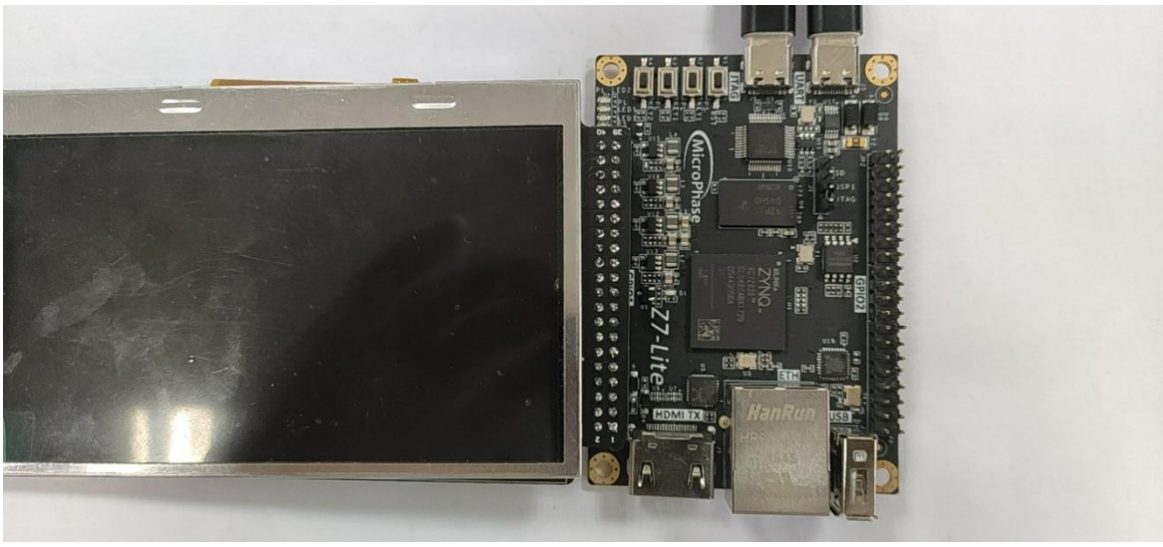
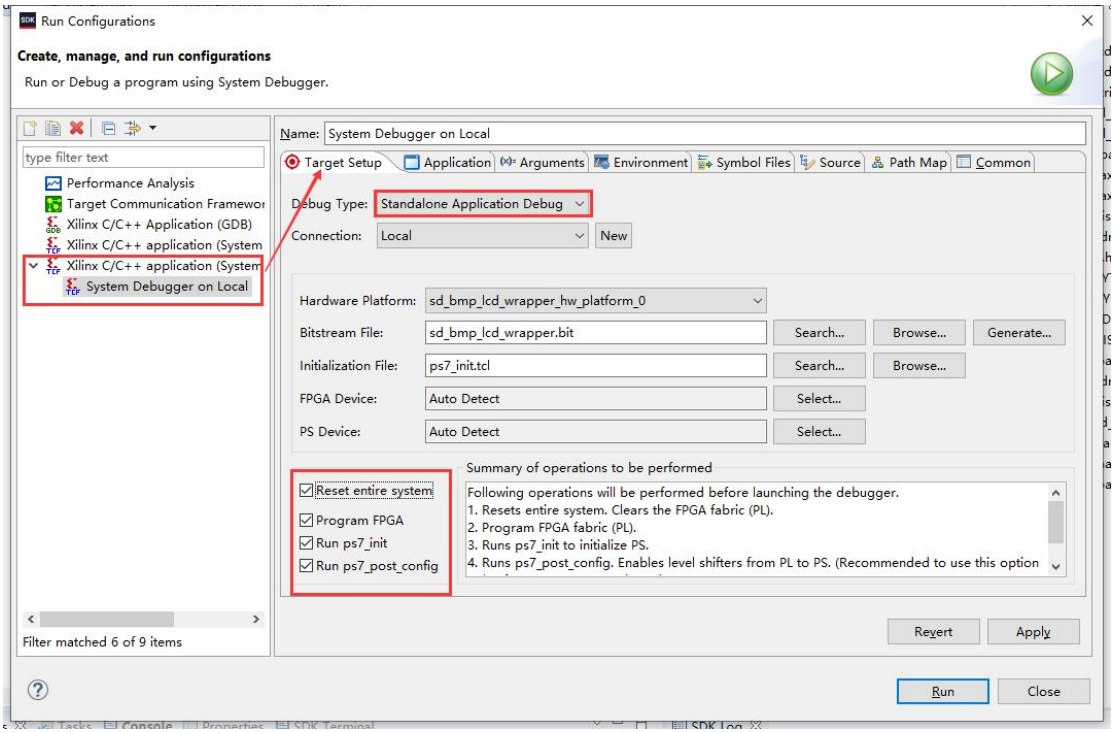


图 19.4.1 显示屏连接

在烧录之前将例程中给出的示例图片拷贝到 SD 卡中并将 SD 卡插入卡槽。之后打开窗口软件并连接串口。打开 Run Configuration 下载设置页面，如下图所示设置相关设置。



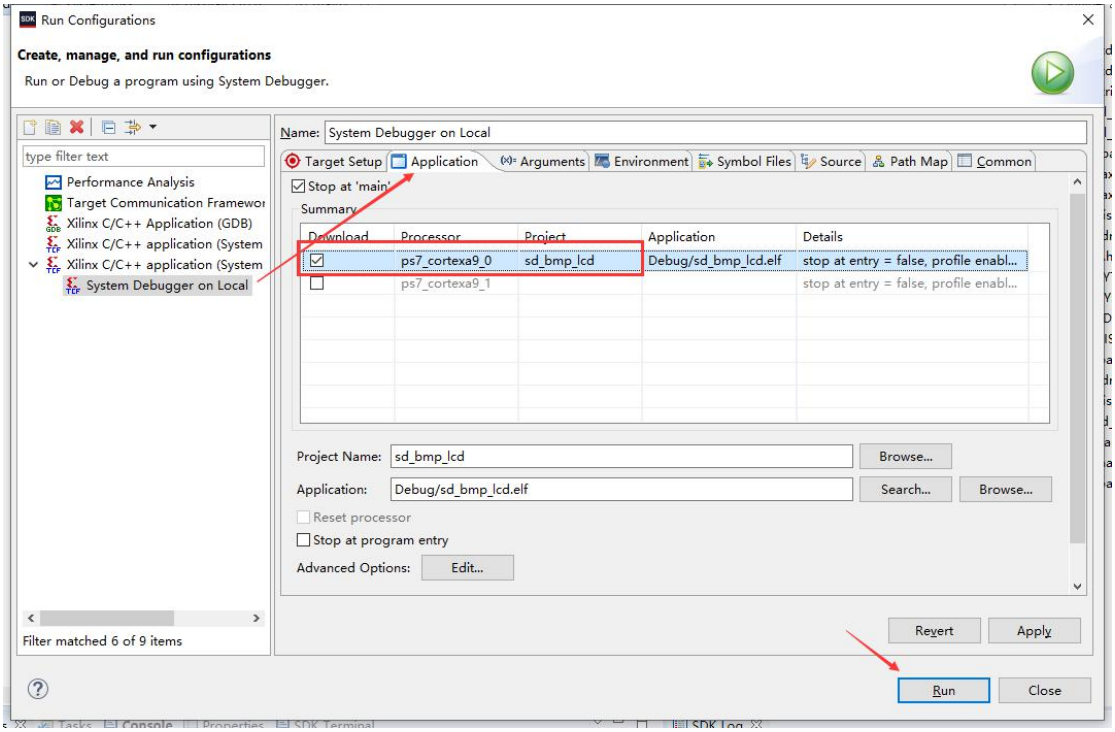
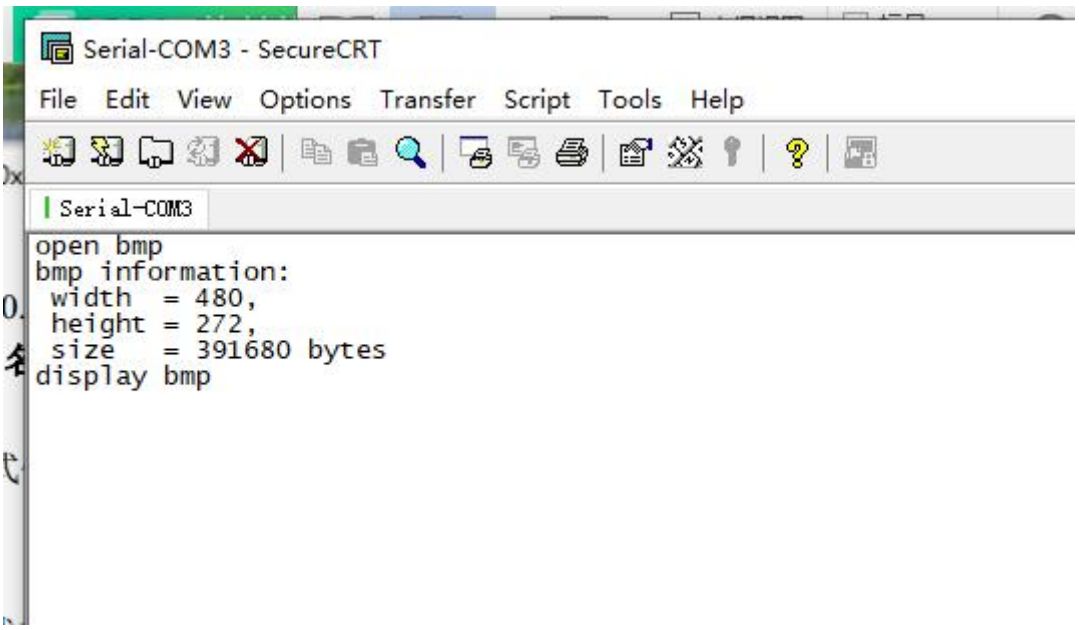


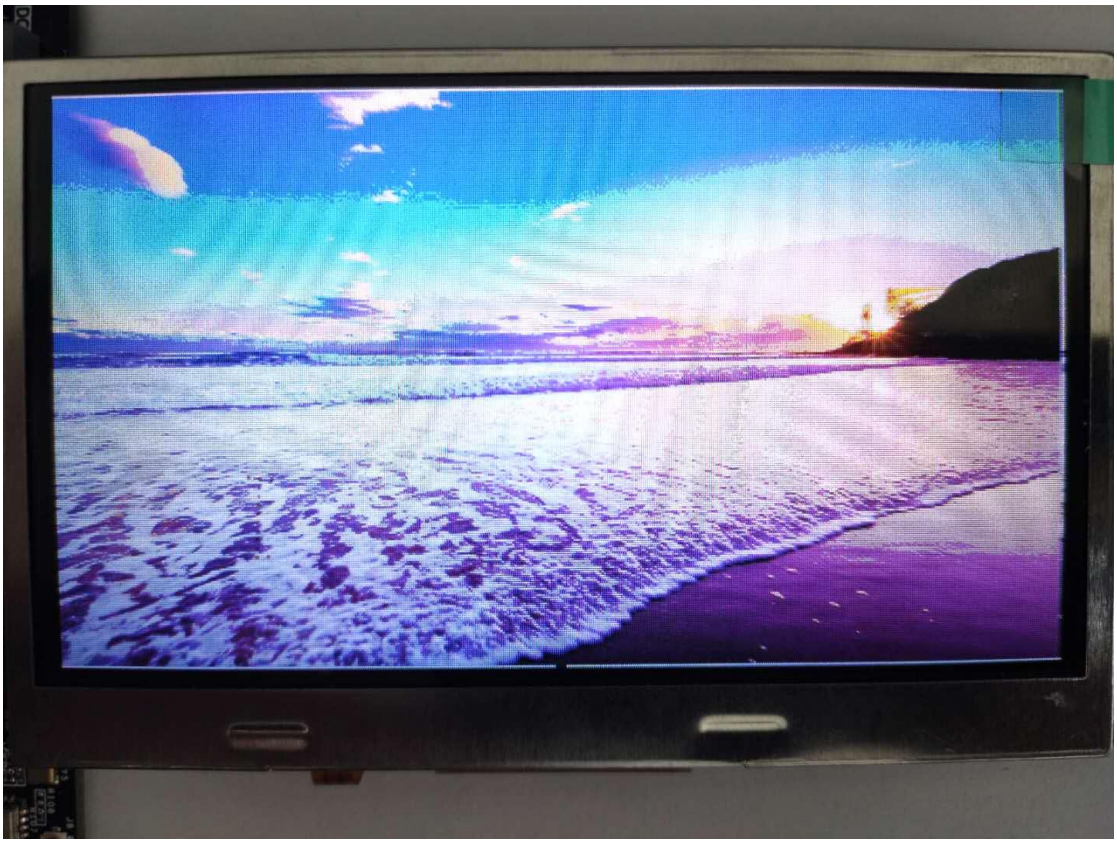
图 19.4.2 烧录相关设置

设置完成后点击“Run”烧录程序，烧录完成后可以看到串口打印出如下信



息，显示出图片的分辨率即大小。

图 19.4.3 图片信息



同时屏幕上显示出该图片，如下图所示：

图 19.4.4 LCD 屏幕显示图片

如果图片分辨率与屏幕不同，则会出现显示异常或无法显示。

19.5 章节小结

本实验的硬件设计及其连线比较复杂，而且引脚众多，需要在实验时多加小心，仔细操作。同时思考各个设计的原理及不同 IP 的作用，查阅相关资料了解接口定义和参数设置的依据，才能知其然也知其所以然。

第 20 章　　OV5640 摄像头 HDMI 显示实验

在常见的视频相关应用中，摄像头图像采集显示是一种常见的应用方式，不论是视频监控、人脸识别或者其他应用，都需要摄像头来采集图像，针对具有这部分需求的客户，我们公司出品了一款具有双摄像头的模块，上面有两个型号为 OV5640 的摄像头。本次实验我们就将使用其中一个来实现视频图像的采集及显示功能。

本章包含以下几部分内容：

20.1 实验简介

20.2 硬件设计

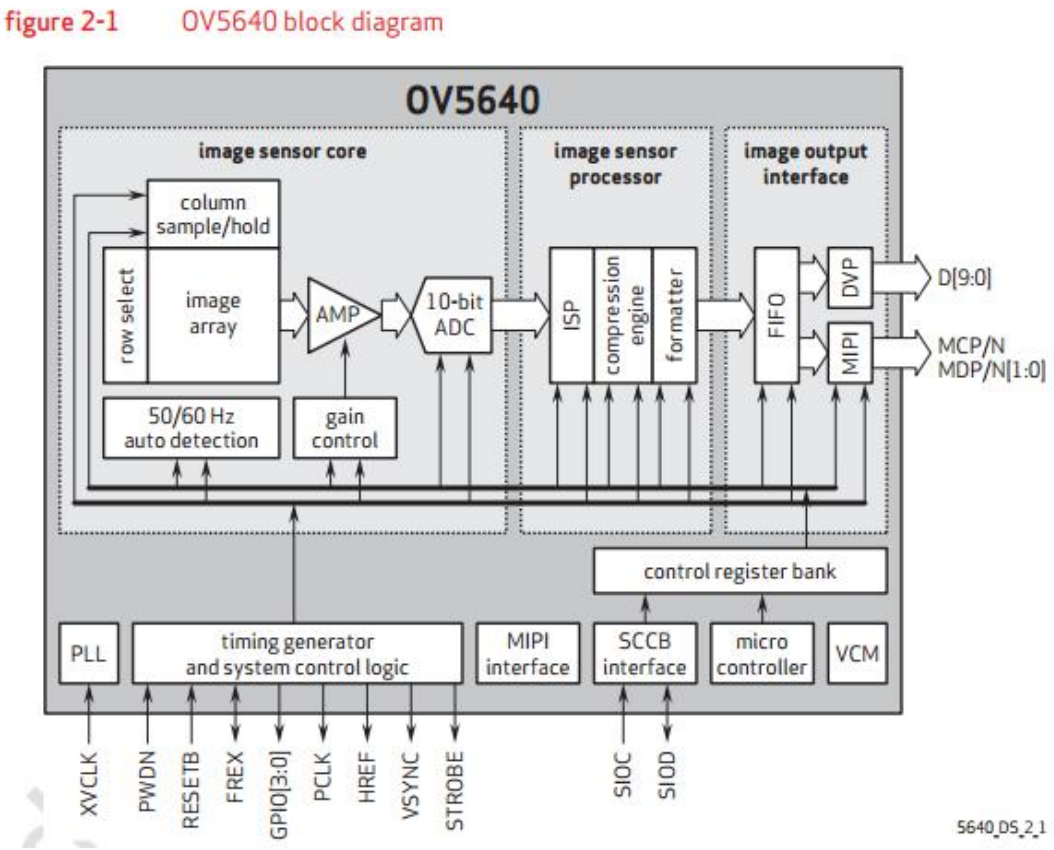
20.3 软件设计

20.4 下载验证

20.5 章节小结

20.1 实验简介

OV5640 是一款 1/4 英寸单芯片图像传感器，其感光阵列达到 2592*1944（即 500W 像素），能实现最快 15fps QSXVGA（2592*1944）或者 90fps VGA（640*480）分辨率的图像采集。传感器采用 OmniVision 推出的 OmniBSI（背面照度）技术，使传感器达到更高的性能，如高灵敏度、低串扰和低噪声。传感器内部集成了图像处理的功能，包括自动曝光控制（AEC）、自动白平衡（AWB） 等。同时该传感器支持 LED 补光、MIPI（移动产业处理器接口）输出接口和 DVP（数字视频并行）输出接口选择、ISP（图像信号处理）以及 AFC（自动聚焦控制）等功能。



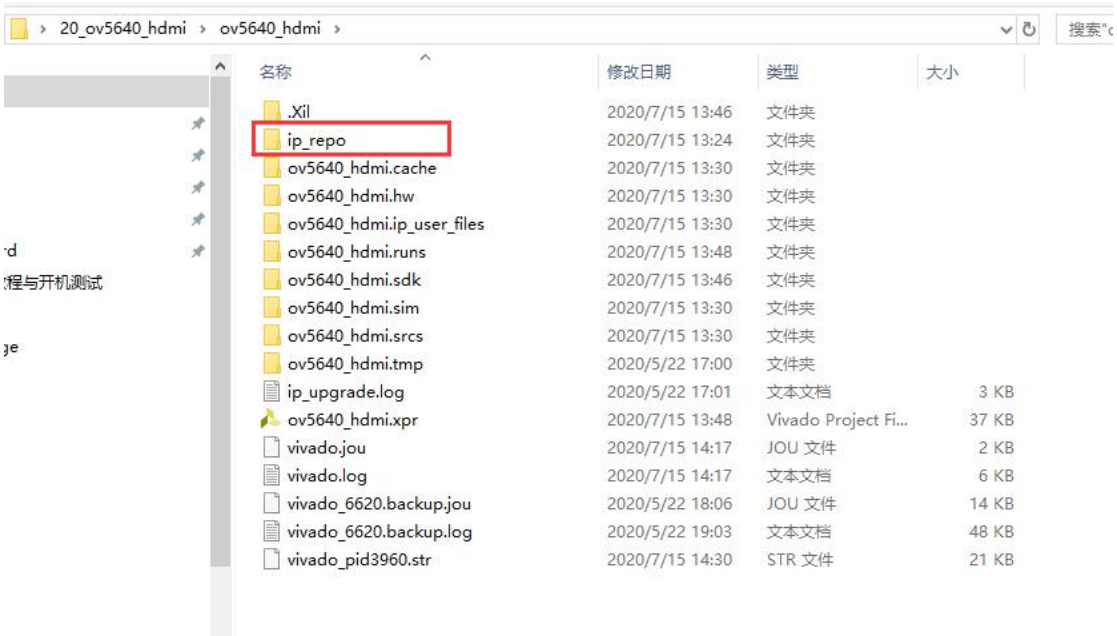
OV5640 的功能框图如下图所示：

图 20.1.1 OV5640 功能框图

从图中我们可以看到这个摄像头的组成及控制关系，外部通过 SCCB 总线来进行寄存器的配置，SCCB 总线接口协议兼容 IIC 协议，所以本实验中我们使用模拟 IIC 的方式来配置相关寄存器。 IIC 协议的接口时序以及相关寄存器的配置可以参考相关的数据手册，此处不再详细说明。

20.2 硬件设计

首先创建一个 Vivado 工程，并命名为“ov5640_hdmi”，根据开发板型号选择相应的芯片型号。工程创建完成后将例程目录下的 ip_repo 文件夹复制到新建的工程目录下，该文件夹包含了一下自定义 IP 核，我们将在后面的硬件设计中



用到这些 IP。自定义 IP 文件夹如下图所示：

图 20.2.1 自定义 IP 文件夹

自定义 IP 文件夹复制完成之后。打开 Vivado 工程界面。点击左侧导航栏的 Settings 选项，打开设置页面。之后展开 IP 下拉菜单，点击 Repository 打开自定义 IP 添加窗口，如下图所示：

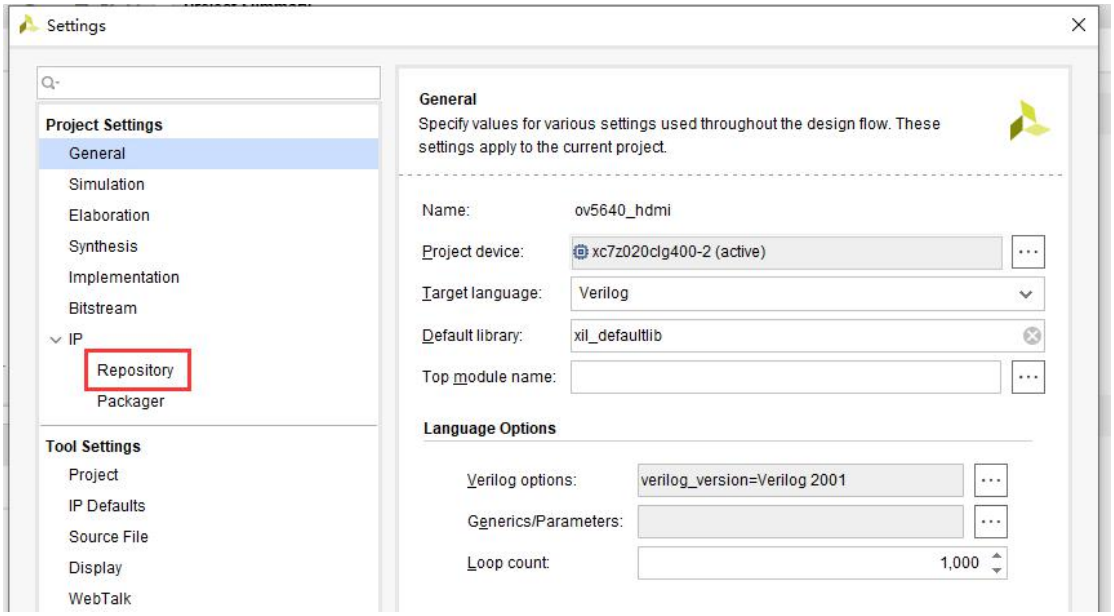
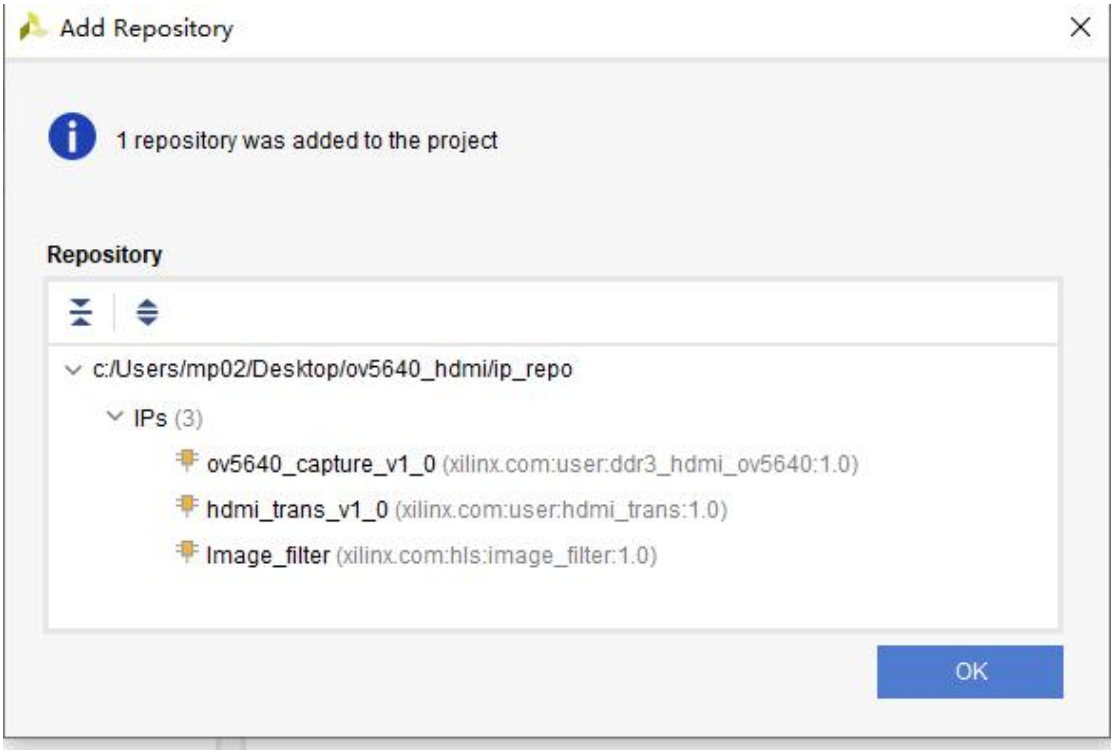


图 20.2.2 自定义 IP 添加选项

之后点击“+”添加自定义 IP 文件夹，定位到复制到工程路径下的 ip_repo

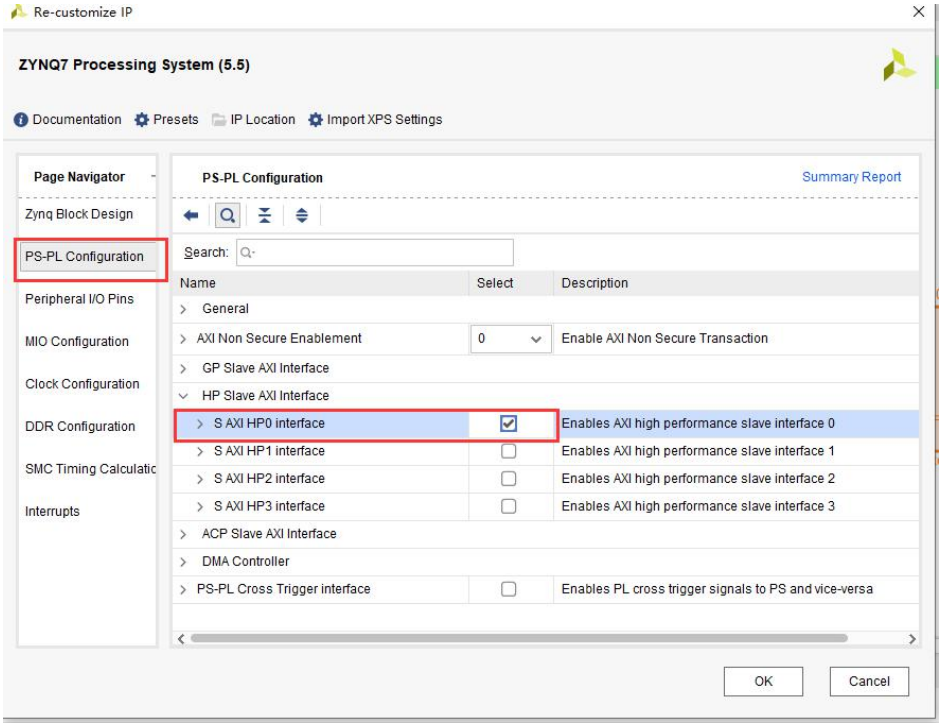


文件夹路径，之后软件自动识别该文件夹内包含的 IP 核，如下图所示：

图 20.2.3 自定义 IP 核

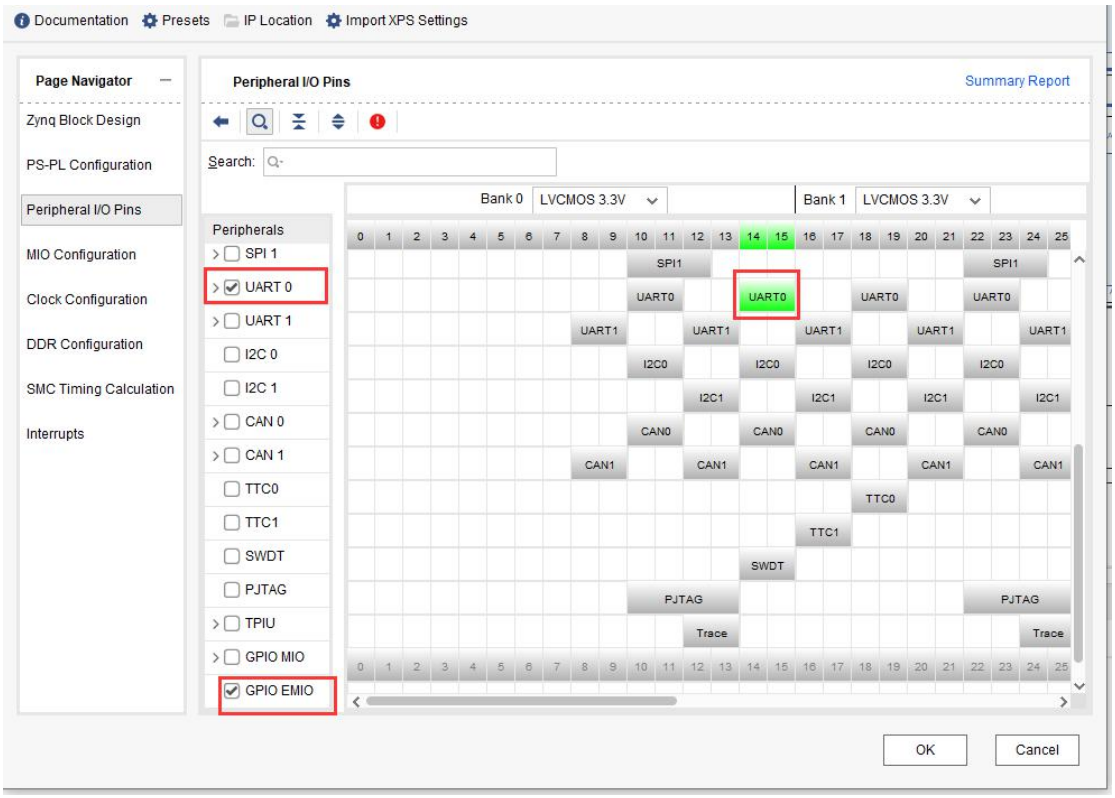
自定义 IP 核添加完成之后，点击 Create Block Design 创建新的设计文件。

打开设计文件添加 ZYNQ7 Processing System，之后双击 IP 打开设置页面。 使能



告诉 AIX 总线接口。如下图所示：

图 20.2.4 使能 AXI 总线接口



使能 UART 串口，使能 EMIO 作为 SCCB 接口配置摄像头。如下图所示：

图 20.2.5 使能外设

在 MIO Configuration 设置项内设置 EMIO 的位宽为 2。

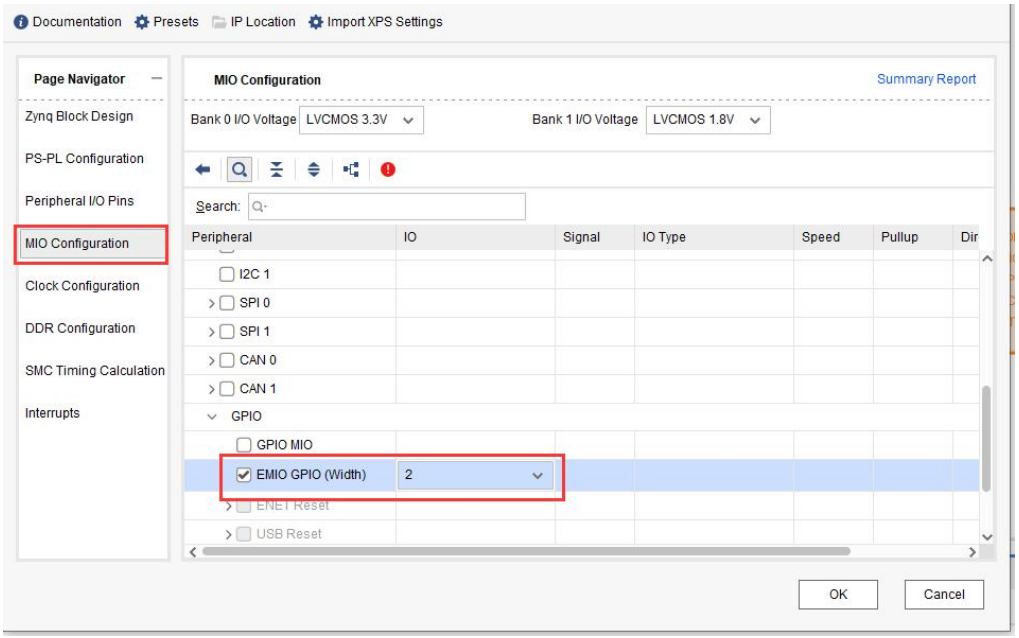


图 20.2.6 设置 EMIO 位宽

保持输出 FCLK_CLK0 默认输出 50Mhz 时钟不变，设置 DDR 控制器。

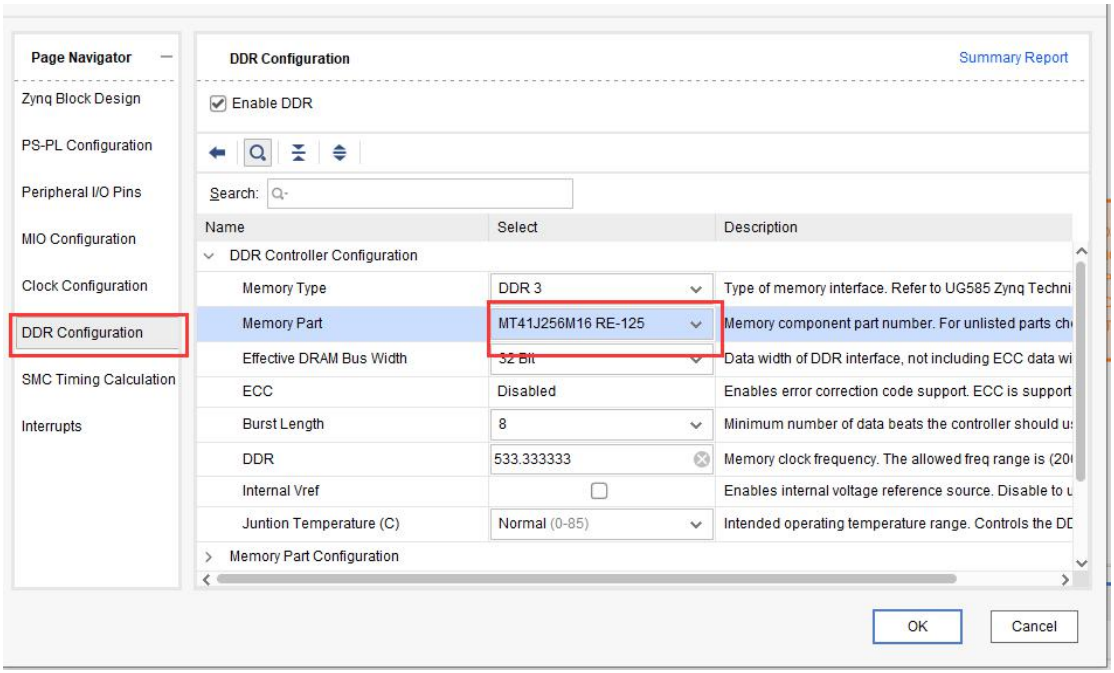


图 20.2.7 设置 DDR 控制器

设置完成后的 ZYNQ7 Peocseeing Sysrtem 如下图所示，点击上方 Run Block Automation 将端口引出，之后手动引出 EMIO 接口并修改接口名称为“emio_sccb”。

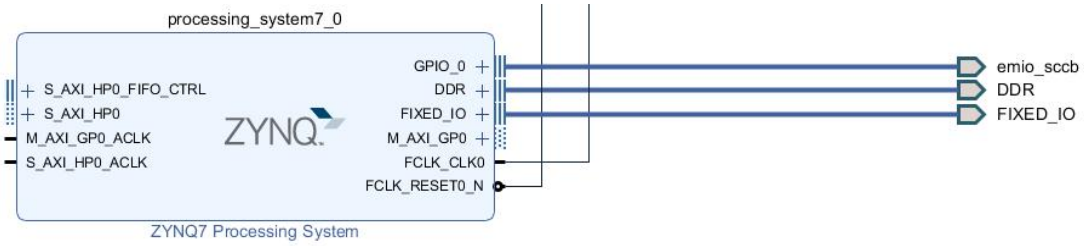


图 20.2.8 引出接口并修改接口信号名称

添加 ov5640_capture IP 核，并将连接外部引脚的端口信号引出，修改信号名称方便辨识及引脚约束。需引出信号如下图所示：

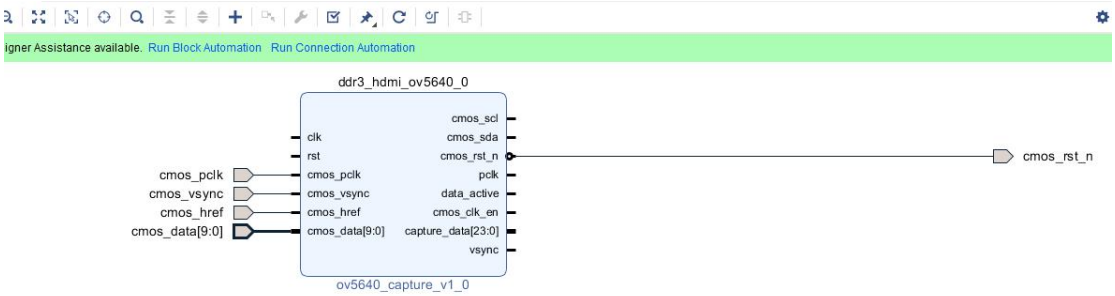


图 20.2.9 引出相关信号

将 PS 端的输出时钟作为该 IP 的驱动时钟，连接时钟信号如下图所示：

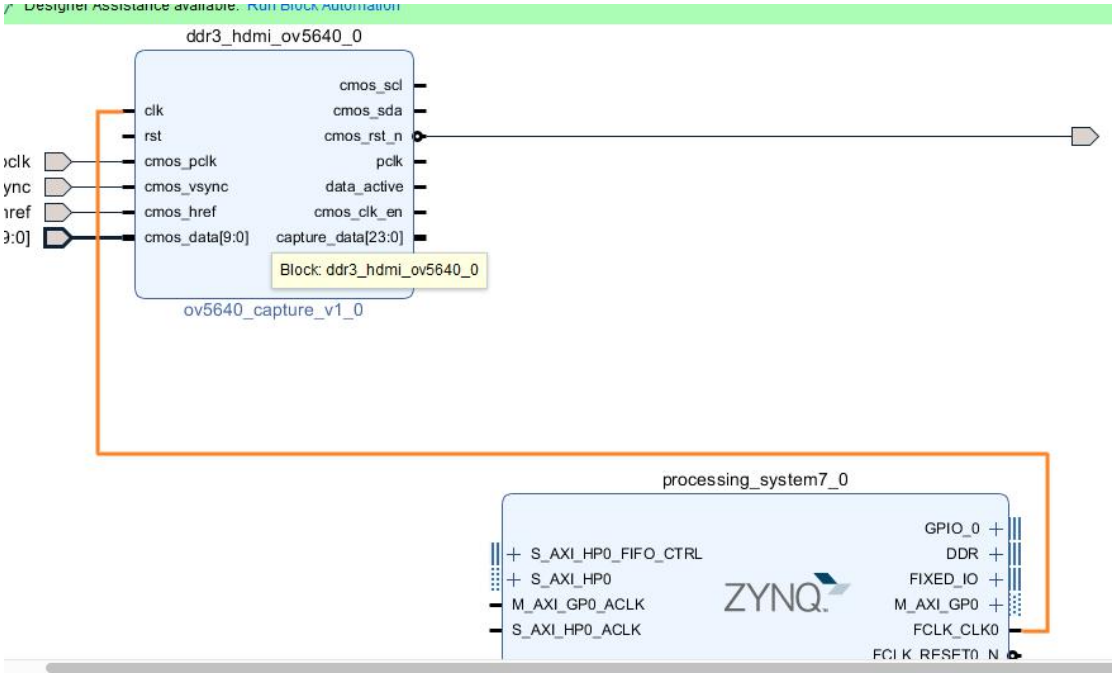


图 20.2.10 连接时钟信号

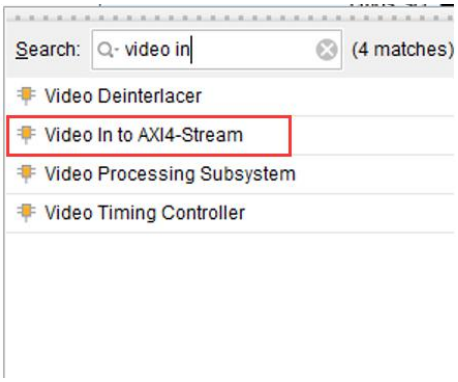


图 20.2.11 搜索添加 video in to AXI4-Stream IP 核

双击 IP 打开设置页面，设置时钟模式为独立模式（Independent），如下图

所示：

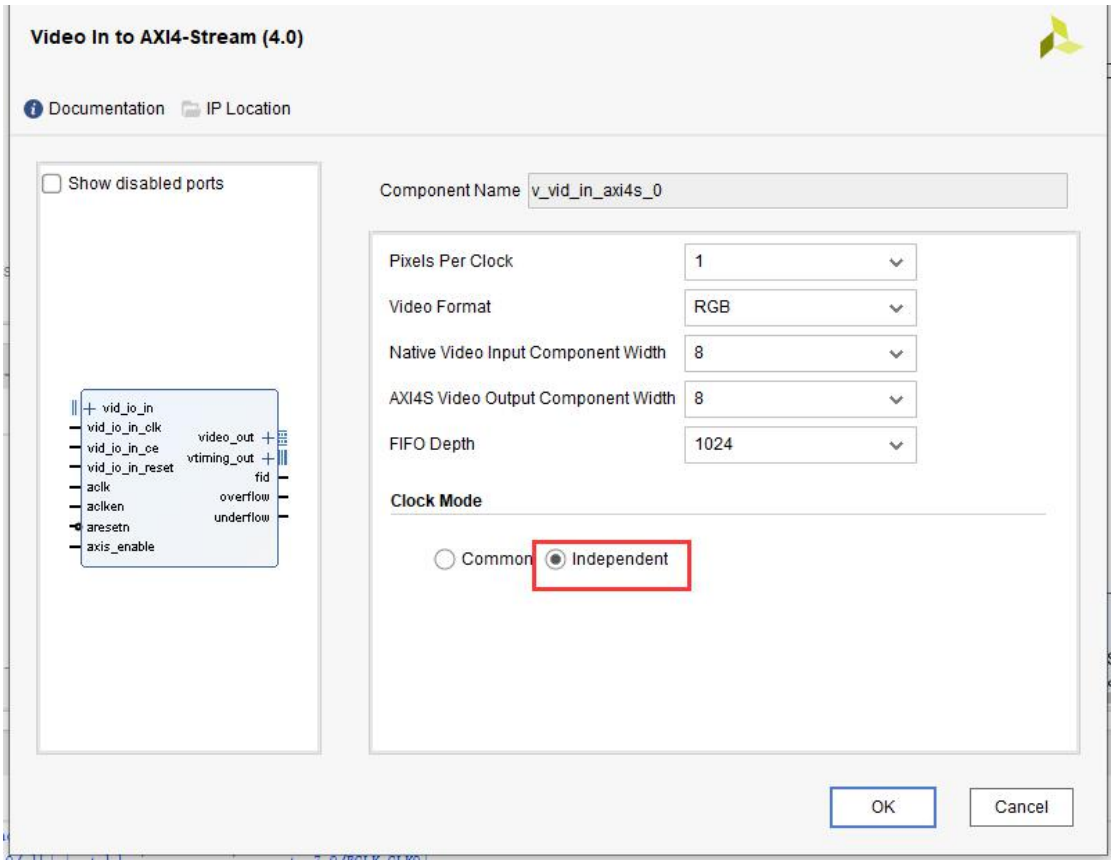


图 20.2.12 设置 IP 的时钟模式

连接 ov5640_capture 和 video in to AXI4-Steram 之间的信号。pclk 连接 vid_io_in_clk；data_active 连接 vid_active_video；cmos_clk_en 连接 vid_io_in_ce；capture_data[23:0]连接 vid_data[23:0]；vsync 连接 vid_vsync。如下图所示：

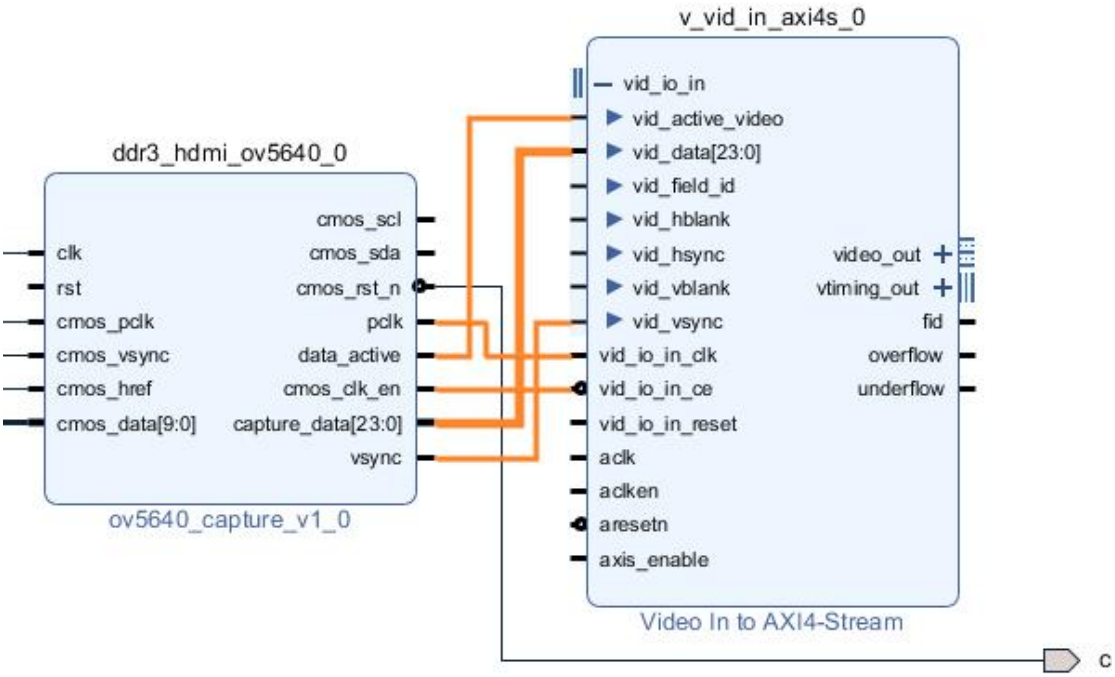


图 20.2.13 连接 ov5640_capture 和 video in to AXI4-Steram 之间的信号

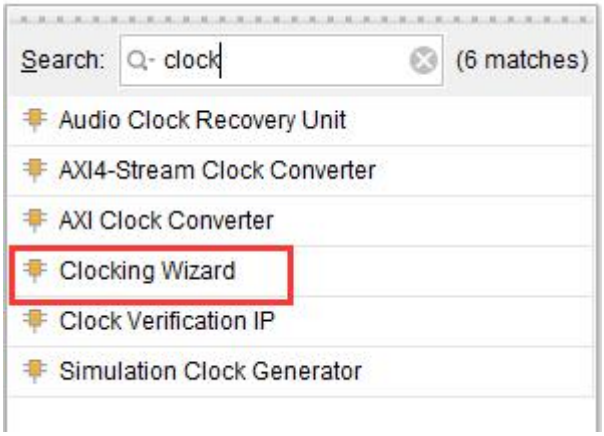
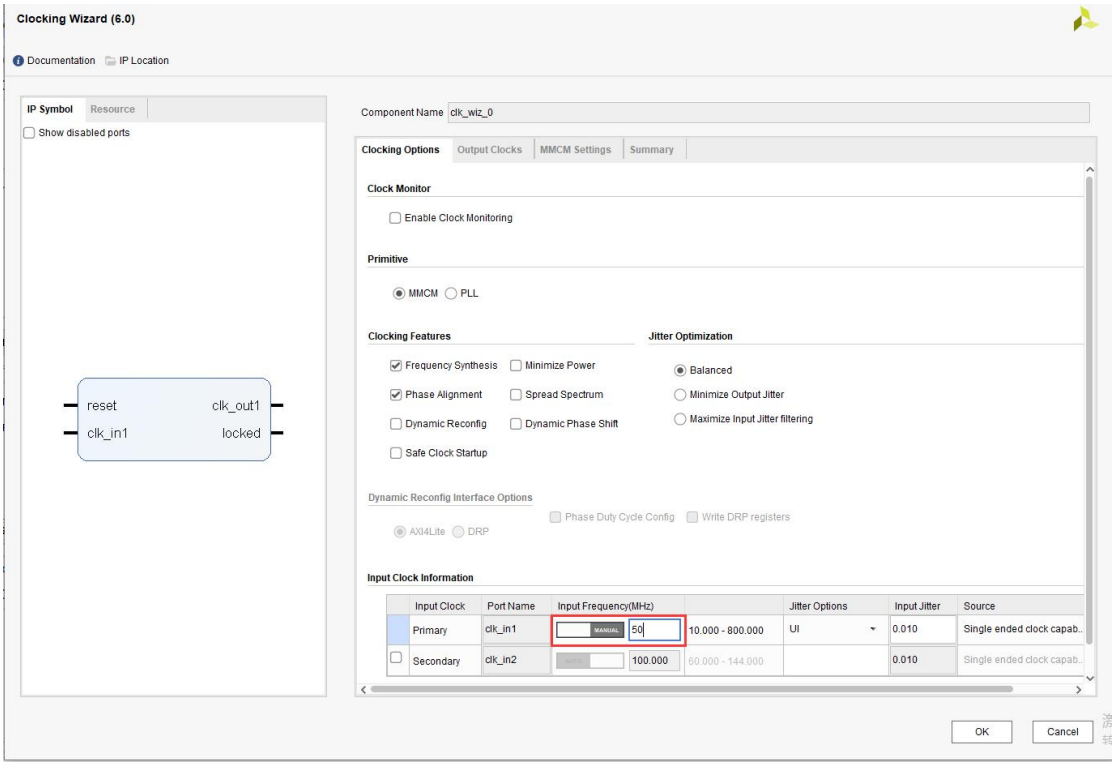


图 20.2.14 搜索添加时钟 IP 核

添加完成之后打开 IP 核设置页面，设置输入时钟频率为 50Mhz，设置输出时钟 1 的输出频率为 65Mhz，输出时钟 2 的频率为 325Mhz，设置复位信号为低电平有效。如下图所示：



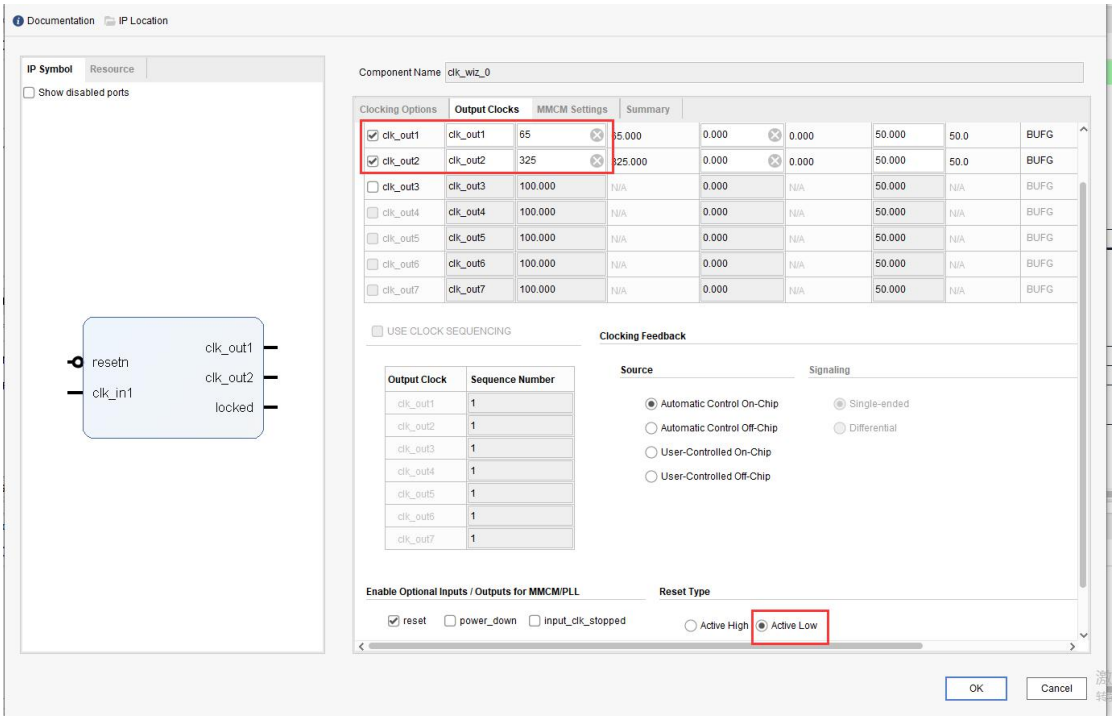


图 20.2.15 设置输入时钟频率

图 20.2.16 设置输出时钟频率及复位有效电平

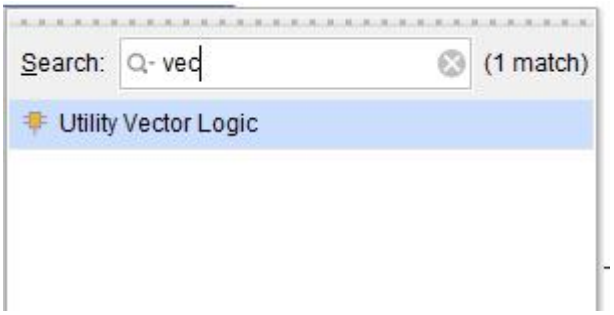


图 20.2.17 搜索添加逻辑门电路 IP

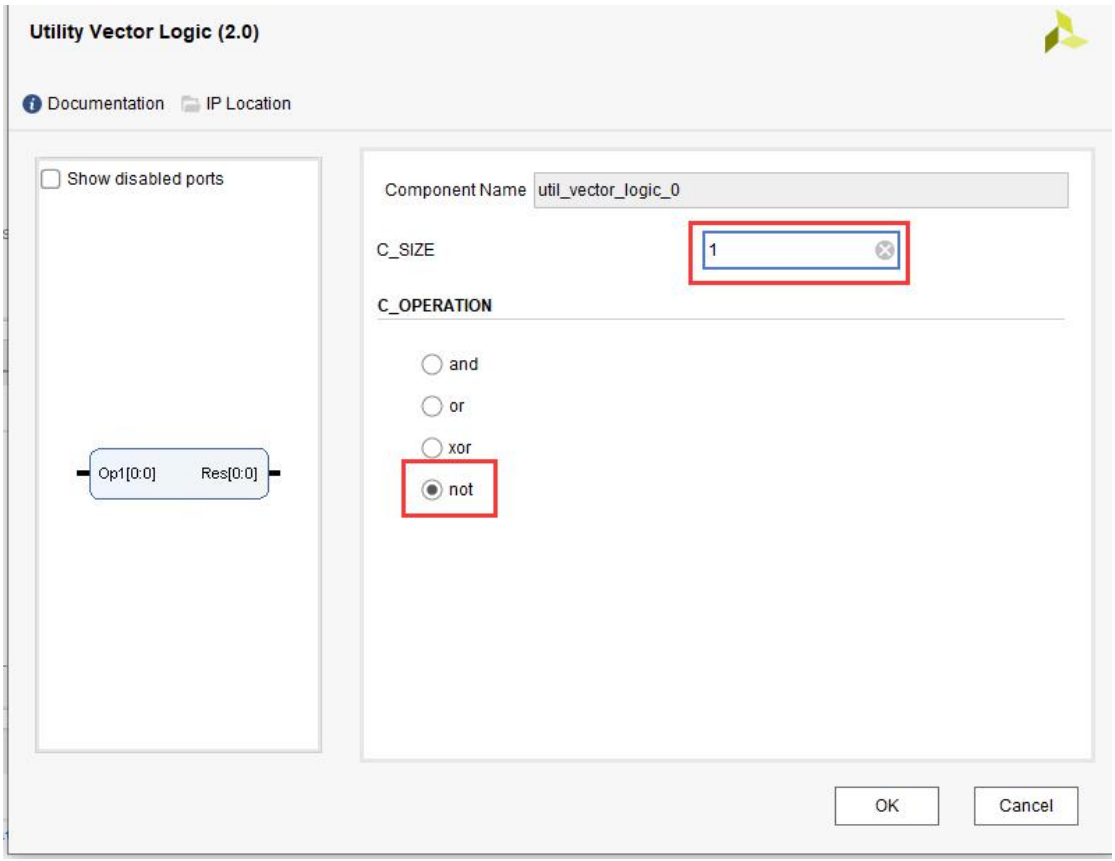
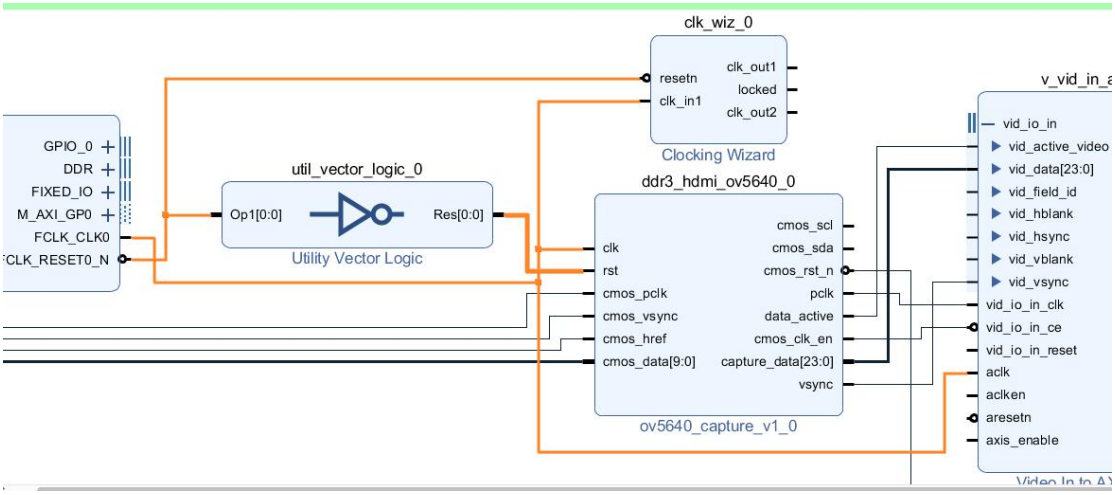


图 20.2.18 设置门电路类型为非门，位宽为 1

设置完成后连接相关信号。时钟 IP 的时钟信号 clk_in1 连接 ps 端输出时钟，复位信号连接 ps 端输出复位信号。ov5640_capture 的时钟输入 clk 连接 ps 端输出时钟信号，复位信号为高有效，将 ps 端输出复位信号经过非门之后连接倒复位信号输入端。video in to AXI4-Stream IP 的时钟信号 aclk 连接 ps 端的输出时钟。信号连接如下图所示：



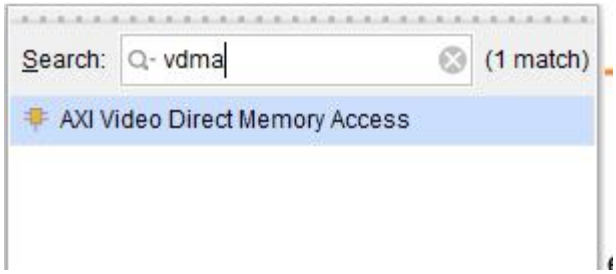
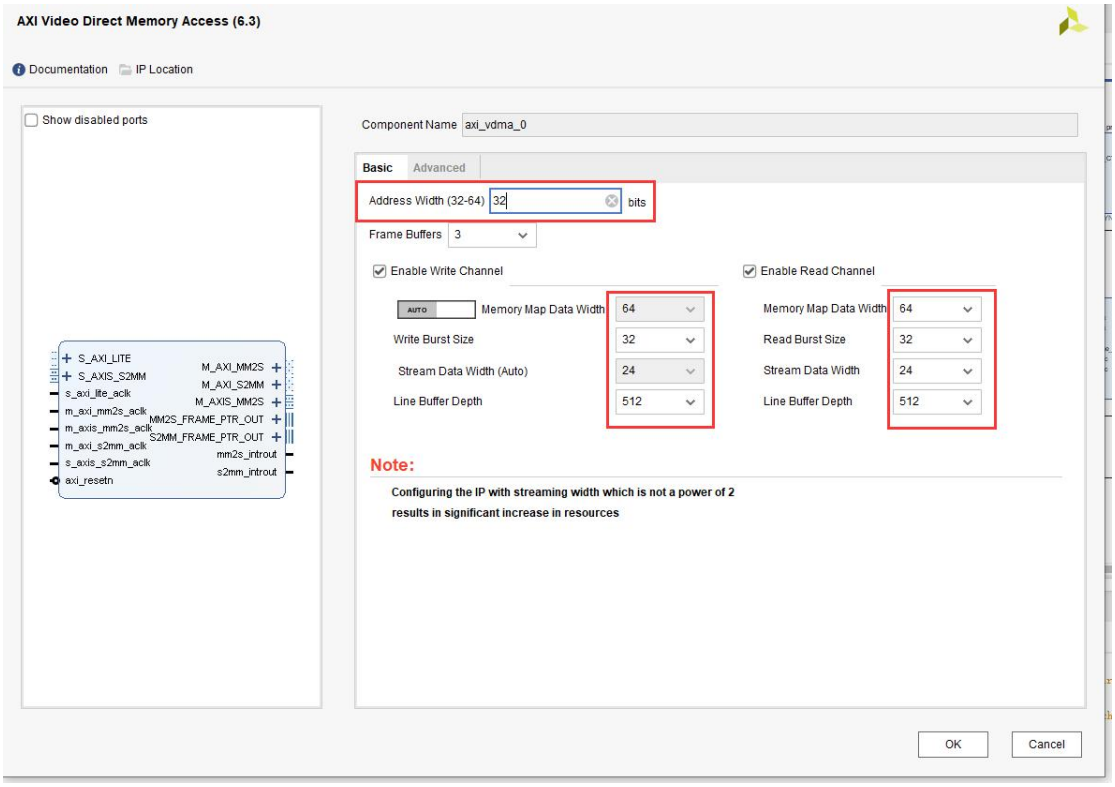


图 20.2.19 部分相关信号连接

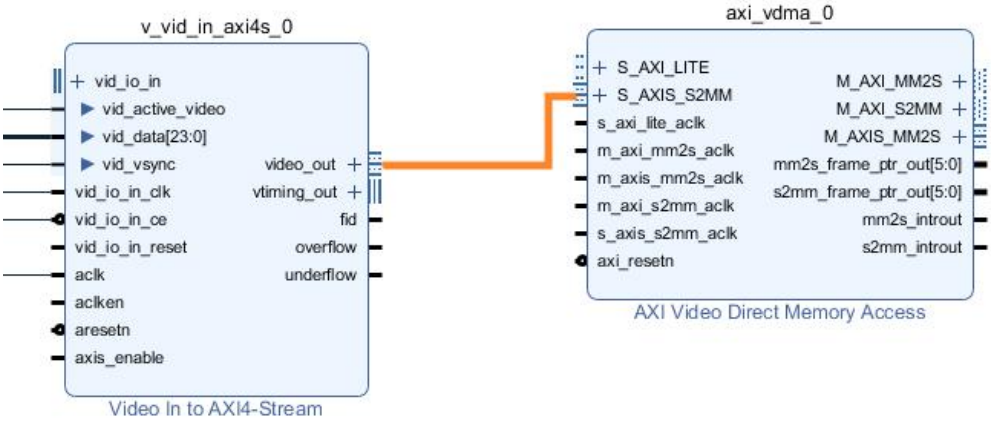
图 20.2.20 搜索添加 VDMA IP



双击打开设置页面，具体设置如下图所示：

图 20.2.21 VDMA 设置项

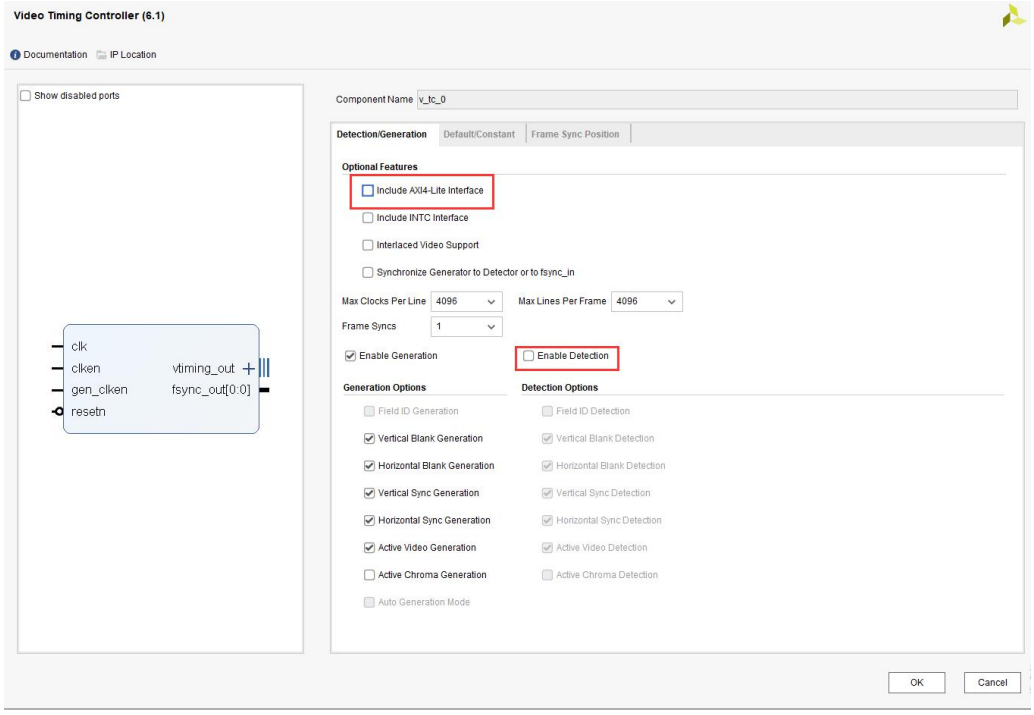
连接 video in to AXI4-Stream IP 的视频信号输出端口 video_out 和 VDMA 的



数据输入端口 S_AXIS_S2MM。如下图所示：

图 20.2.22 连接视频数据信号

图 20.2.23 搜索添加 video timing controller IP 核



双击打开 IP 核设置页面，如下图所示进行设置：

图 20.2.24 取消选项勾选

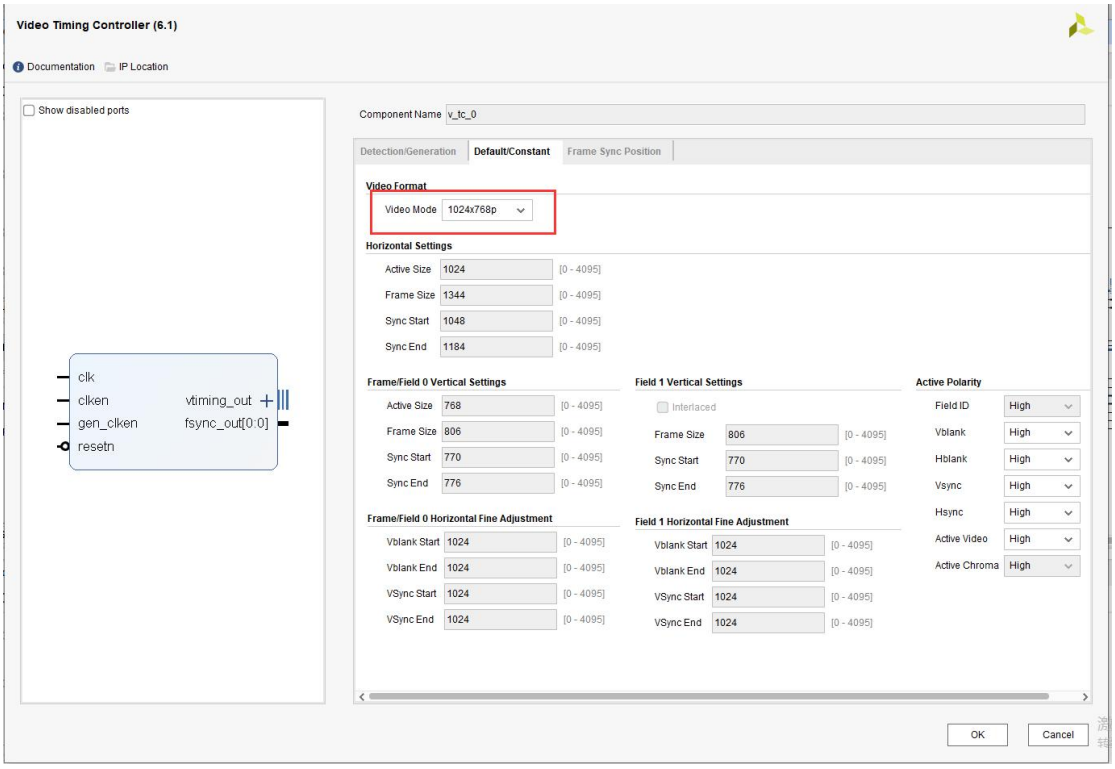
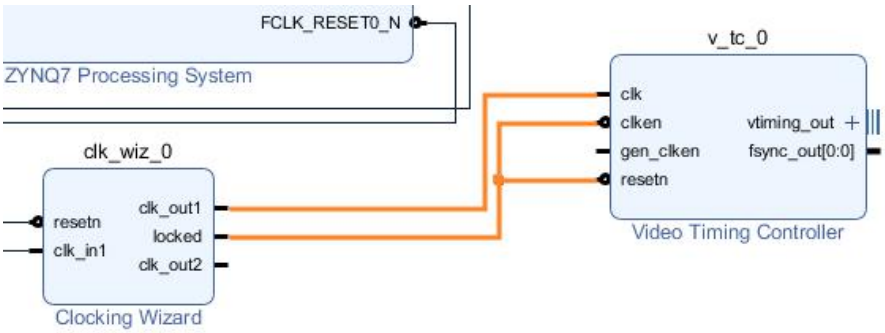


图 20.2.25 设置视频分辨率为 1024*768

连接 IP 核的输入时钟信号及复位信号，如下图所示:



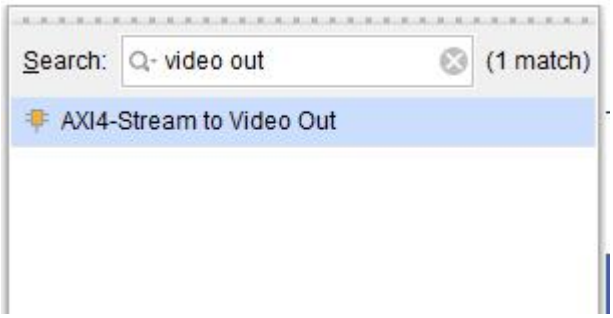
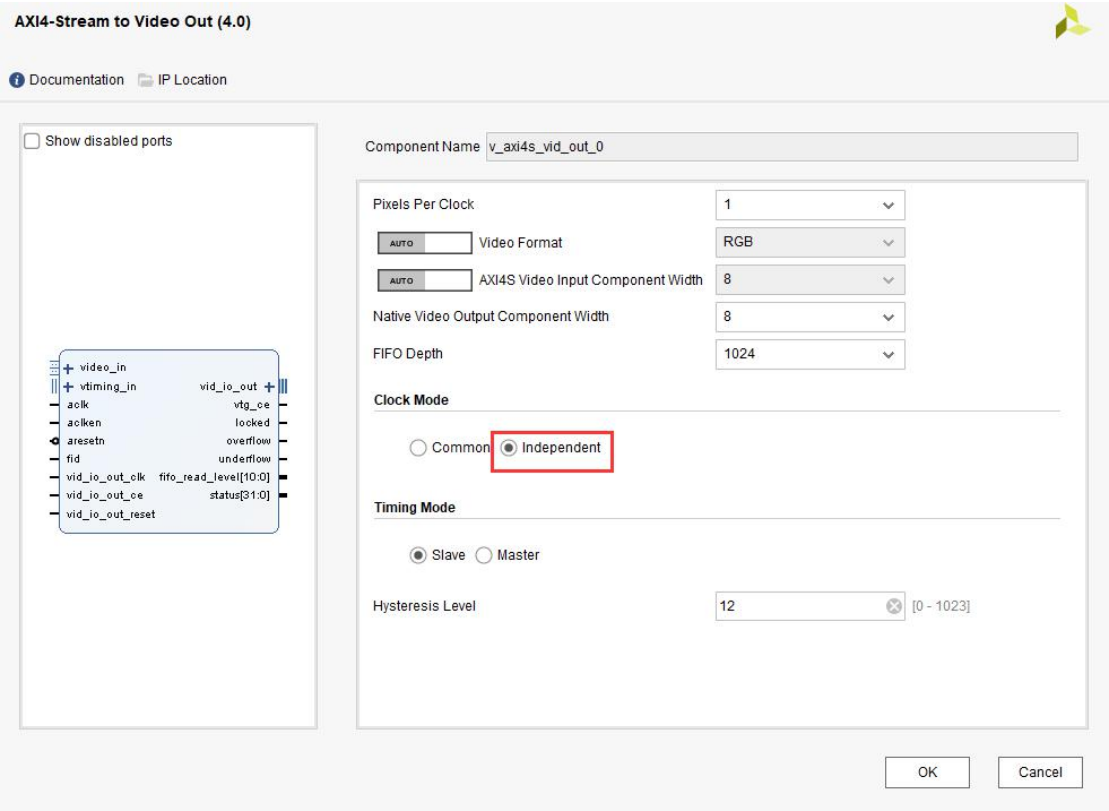


图 20.2.26 连接相关信号

图 20.2.27 搜索添加 AXI4-Stream to video out IP



双击打开设置页面，设置时钟模式为独立模式。

图 20.2.28 设置时钟模式

添加非门逻辑电路，之后如下图所示连接相关信号。

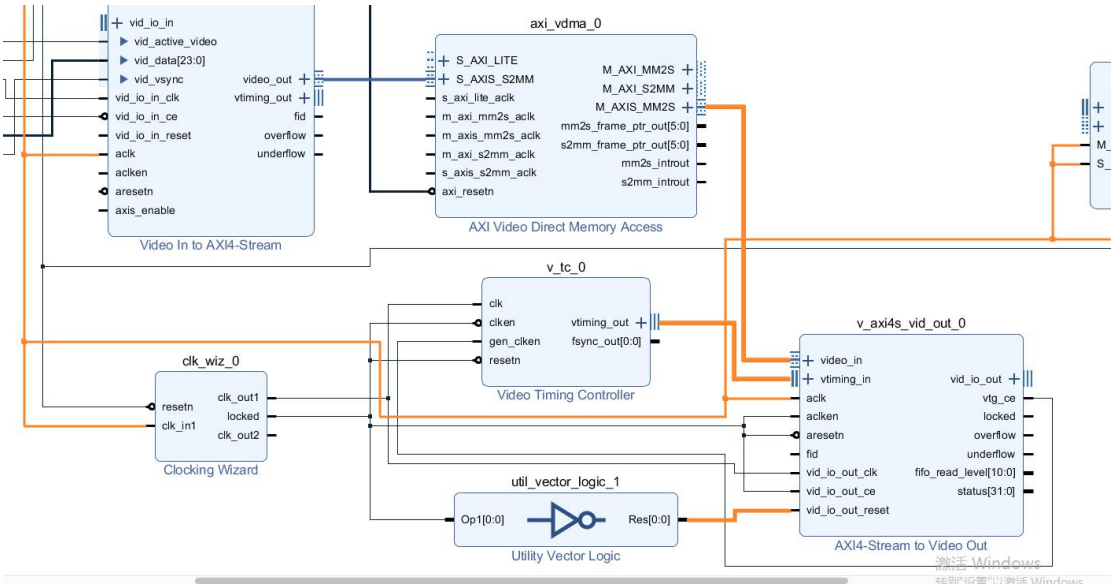
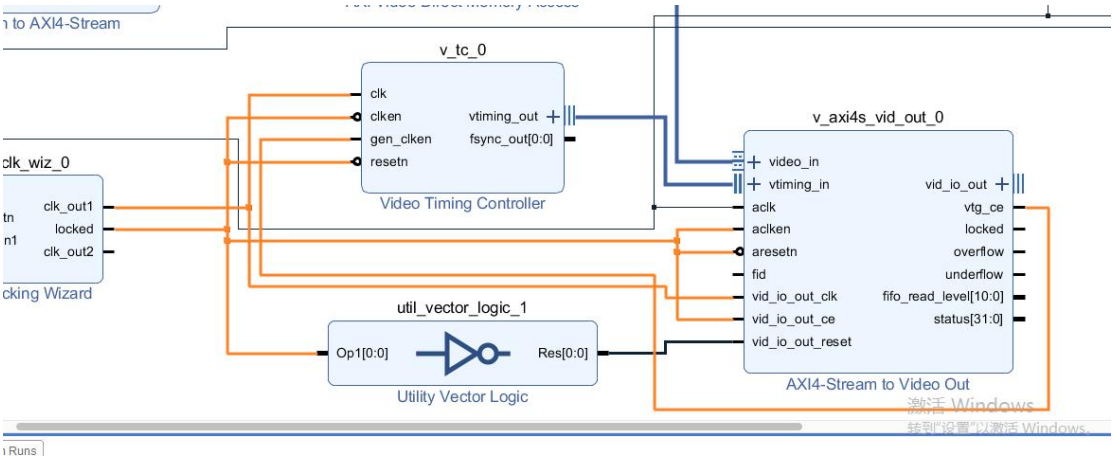


图 20.2.29 信号连接 1



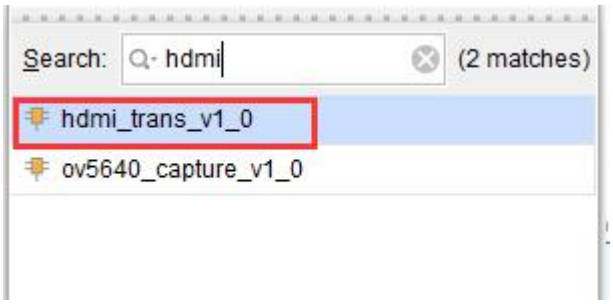


图 20.2.30 信号连接 2

图 20.2.31 搜索添加 hdmi_trans IP

将所有输出信号引出并修改信号名称。

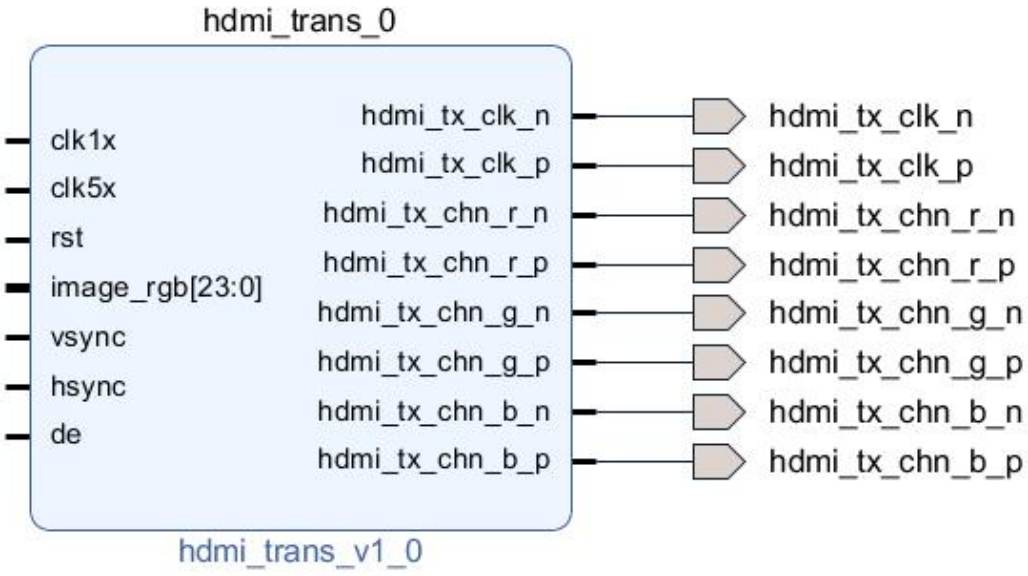
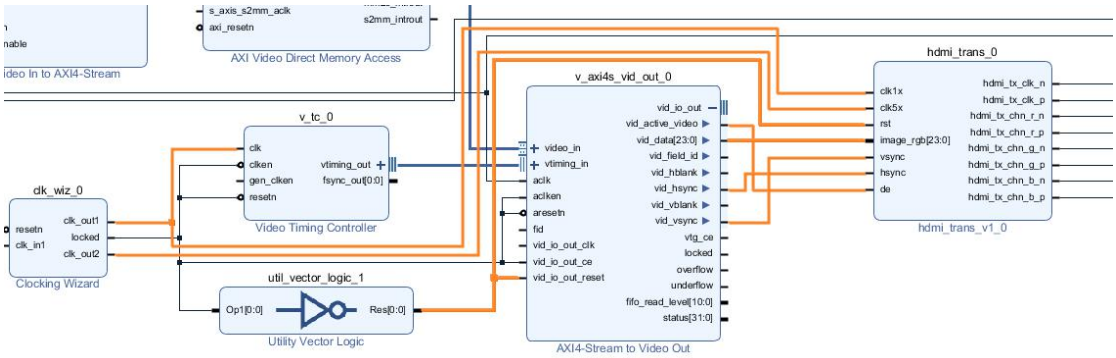


图 20.2.32 引出信号端口并修改名称

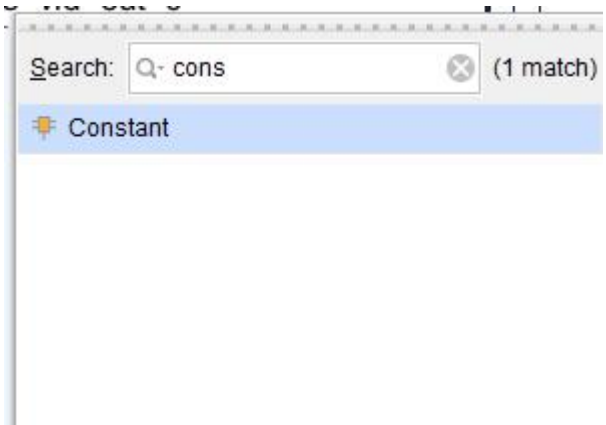
连接该 IP 其他信号，clk1x 连接时钟 IP 输出时钟信号 1，clk5x 连接时钟 IP 输出时钟信号 2，时钟 IP locked 信号经非门取反后连接 rst 复位信号。展开 vid_io_out，连接 de 和 vid_active_video，连接 image_rgb[23:0]和 vid_data[23:0]，

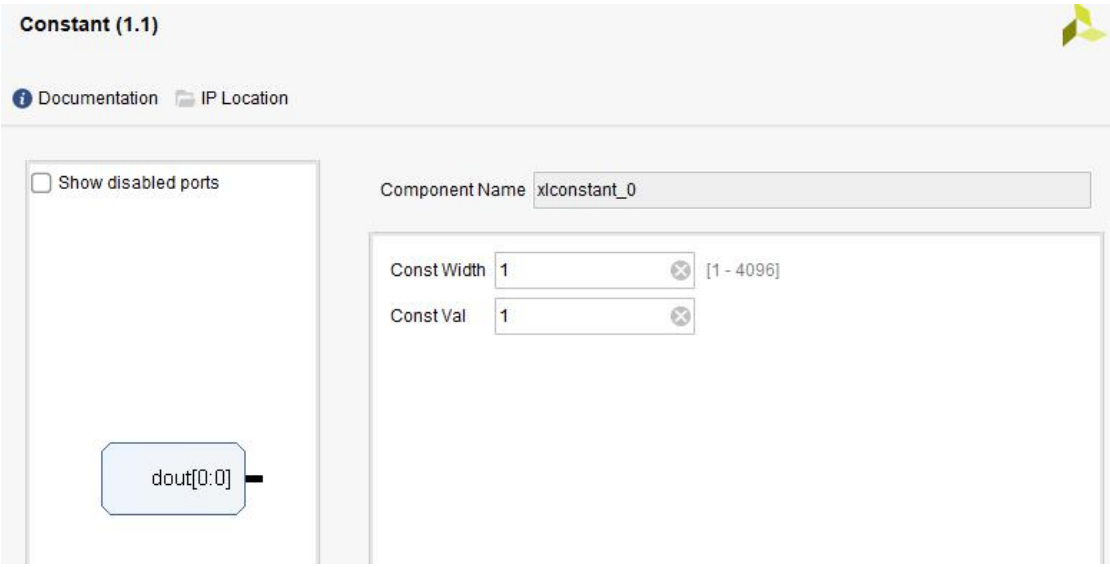


连接 vsync 和 vid_vsync，连接 hsync 和 vid_hsync。

图 20.2.33 连接相关信号

图 20.2.34 搜索添加 constant IP

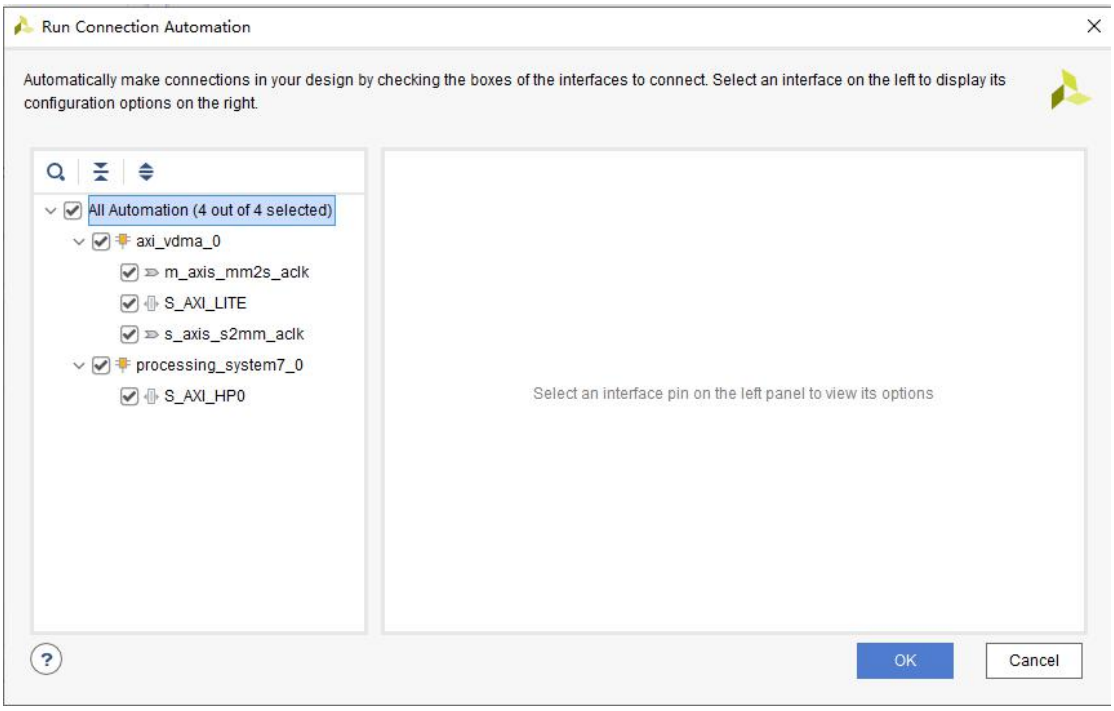




设置输出位宽为 1，输出高电平作为 HDMI 输出使能信号。

图 20.2.35 设置输出位宽和输出电平

图 20.2.36 将输出信号引出并修改信号名称



点击 Run Connection Automation 自动连接信号。

图 20.2.37 全选所有信号并连接

连接完成后再次点击 Run Connection Automation 自动连接信号。

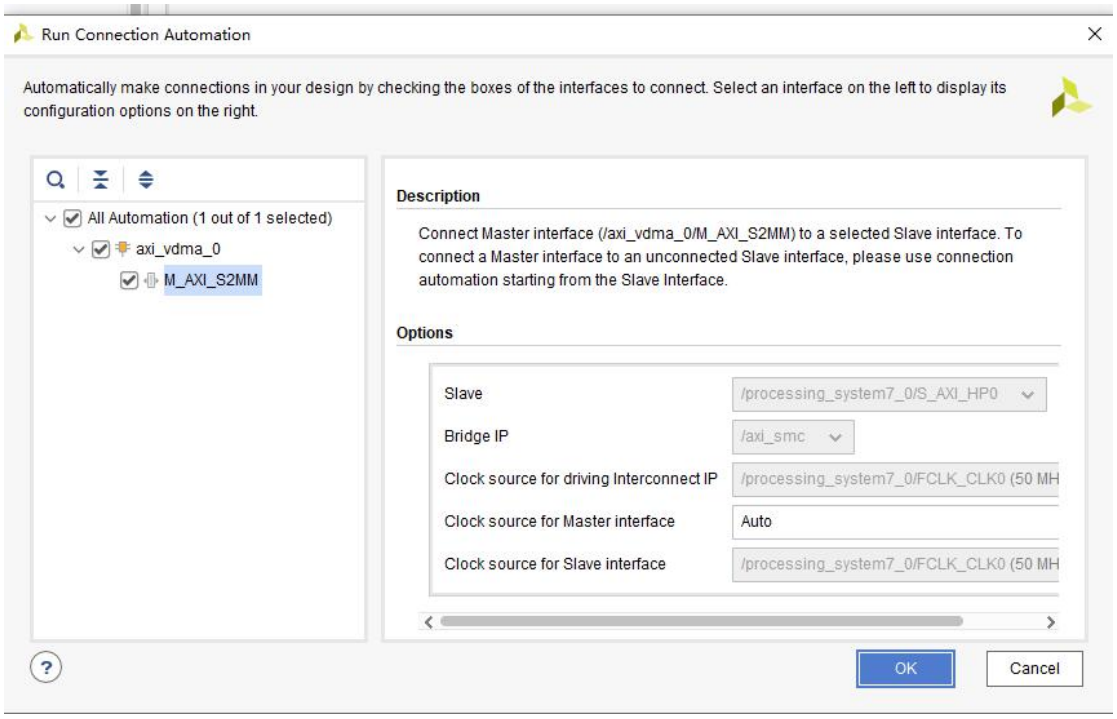
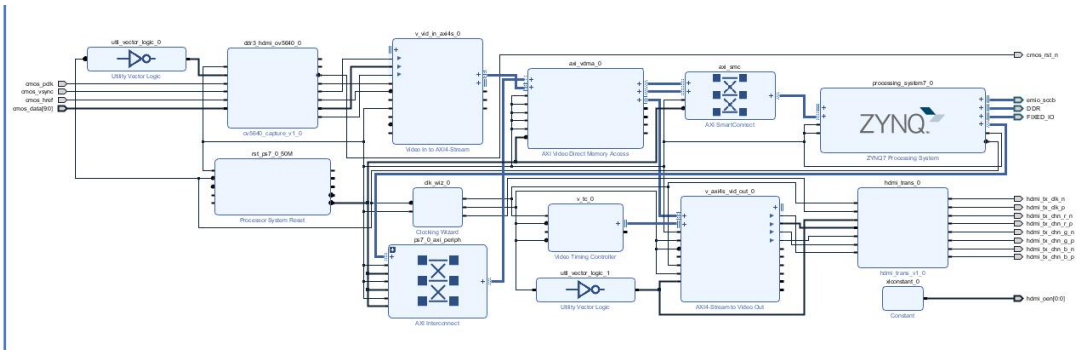


图 20.2.38 连接 M_AXI_S2MM 信号



连接完成后重新生成标准布局如下图所示：

图 20.2.39 设计完成后的标准布局

之后进行引脚约束，具体约束如下图所示，本次实验使用 GPIO1 这一组引脚。

```
set_property PACKAGE_PIN N17 [get_ports cmos_pclk]
set_property PACKAGE_PIN W14 [get_ports cmos_href]
set_property PACKAGE_PIN V18 [get_ports cmos_vsync]
set_property PACKAGE_PIN Y14 [get_ports cmos_rst_n]


set_property PACKAGE_PIN U17 [get_ports {cmos_data[9]}]
set_property PACKAGE_PIN T16 [get_ports {cmos_data[8]}]
set_property PACKAGE_PIN W18 [get_ports {cmos_data[7]}]
set_property PACKAGE_PIN W19 [get_ports {cmos_data[6]}]
set_property PACKAGE_PIN Y18 [get_ports {cmos_data[5]}]
set_property PACKAGE_PIN Y16 [get_ports {cmos_data[4]}]
set_property PACKAGE_PIN R17 [get_ports {cmos_data[3]}]
set_property PACKAGE_PIN Y19 [get_ports {cmos_data[2]}]
set_property PACKAGE_PIN V17 [get_ports {cmos_data[1]}]
set_property PACKAGE_PIN Y17 [get_ports {cmos_data[0]}]


set_property IOSTANDARD LVCMOS33 [get_ports cmos_rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports cmos_pclk]
set_property IOSTANDARD LVCMOS33 [get_ports cmos_vsync]
set_property IOSTANDARD LVCMOS33 [get_ports cmos_href]
set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data[*]}]

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets cmos_pclk_IBUF]


set_property PACKAGE_PIN U18 [get_ports hdmi_tx_clk_p]
set_property PACKAGE_PIN V20 [get_ports hdmi_tx_chn_b_p]
set_property PACKAGE_PIN T20 [get_ports hdmi_tx_chn_g_p]
set_property PACKAGE_PIN N20 [get_ports hdmi_tx_chn_r_p]

set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_r_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_g_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_b_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_clk_p]

set_property IOSTANDARD LVCMOS33 [get_ports {emio_sccb_tri_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {emio_sccb_tri_io[0]}]
set_property PACKAGE_PIN P18 [get_ports {emio_sccb_tri_io[0]}]
set_property PACKAGE_PIN R16 [get_ports {emio_sccb_tri_io[1]}]
```

图 20.2.40 引脚约束

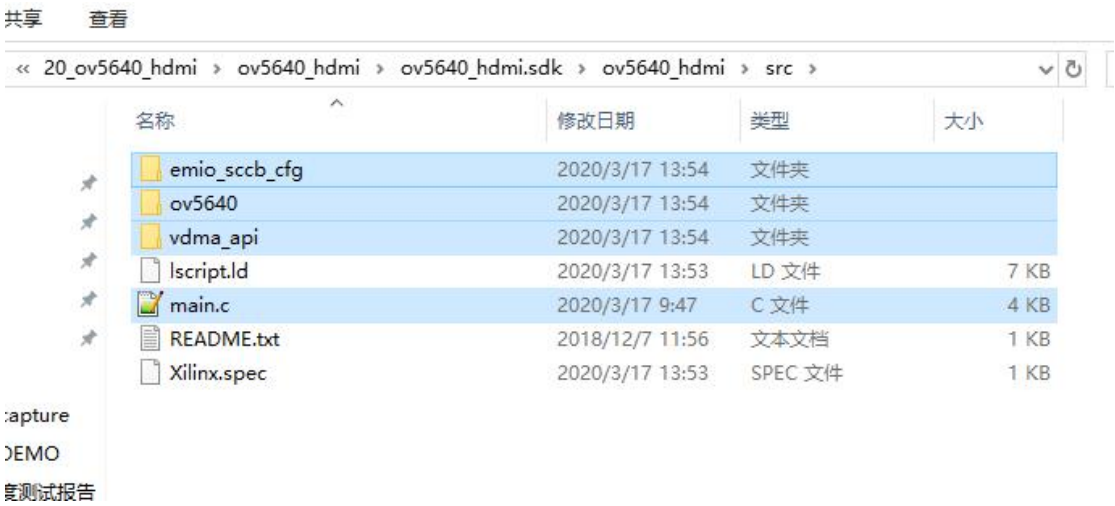
修改完成后关于硬件设计部分的修改就完成了，之后重新生成 Bitstream 文件并导出到 SDK 中，即可进行软件设计的工作。

对于使用 HDMI 解码芯片 ADV7513 的 XME7015-PE100 开发板，请查阅之前 HDMI 显示实验的硬件设计，结合示例工程进行修改即可。

20.3 软件设计

在之前 FPGA 的实验中，摄像头的配置全部是通过 IP 核来实现的，现在我们将尝试通过软件来实现对于摄像头的配置，这样我们就可根据实际情况修改摄像头的各个参数，在使用时更加灵活方便。

在打开的 SDK 开发平台中创建新应用工程并命名为“ov5640_hdmi”，工程



模板选择空白。创建完成之后如下图所示复制以下程序源文件到工程内 src 文件夹。

图 20.3.1 程序源文件

复制的文件中主要包括以下几个文件：

1、 emio_sccb_cfg：该文件夹包含 SCCB 总线协议（IIC 总线协议）的相关程序，通过程序模拟实现通讯协议的时序约束，通过端口发送或接收数据；

2、 ov5640：该文件夹包含摄像头初始化设置的函数以及读写寄存器的相关函数；

3、 vdma_api：该文件包含 VDMA 的相关驱动程序；

4、 main.c：主函数程序

程序各部分的内容比较简单，主要是将之前 IP 核中的部分功能通过软件程序来实现。主函数的代码如下图所示：

```
47 #include <stdio.h>
48 #include <stdlib.h>
49 #include <string.h>
50 #include "xil_types.h"
51 #include "xil_cache.h"
52 #include "xparameters.h"
53 #include "xaxivdma.h"
54 #include "xaxivdma_i.h"
55 #include "vdma_api/vdma_api.h"
56 #include "emio_sccb_cfg/emio_sccb_cfg.h"
57 #include "ov5640/ov5640_init.h"
58
59 ////宏定义
60 #define VDMA_ID                XPAR_AXIVDMA_0_DEVICE_ID    //VDMA器件ID
61 #define VDMA_BASE_ADDR         0x08000000                  //VDMA基地址
62 #define SCREEN_X               1024
63 #define SCREEN_Y               768
64 //全局变量
65 XAxiVdma     vdma;
66 int main(void)
67 {
68     u32 status;
69     u16 cmos_h_pixel;                    //ov5640 DVP 输出水平像素点数
70     u16 cmos_v_pixel;                    //ov5640 DVP 输出垂直像素点数
71     u16 total_h_pixel;                   //ov5640 水平总像素大小
72     u16 total_v_pixel;                   //ov5640 垂直总像素大小
73
74     cmos_h_pixel = 1024;                 //设置OV5640输出分辨率
75     cmos_v_pixel = 768;
76     total_h_pixel = 2570;
77     total_v_pixel = 980;
78
79     emio_init();                         //初始化EMIO
```

```
    //初始化ov5640
    status = ov5640_init( cmos_h_pixel,
                          cmos_v_pixel,
                         total_h_pixel,
                         total_v_pixel);
    if(status == 0)
        xil_printf("OV5640 detected successful!\r\n");
    else
        xil_printf("OV5640 detected failed!\r\n");


    //配置VDMA
    run_vdma_frame_buffer(&vdma, VDMA_ID, SCREEN_X, SCREEN_Y,
            VDMA_BASE_ADDR,0,0,BOTH);
    return 0;
}
```

在程序中定义了四个变量来控制输出图像分辨率的大小，这样可以根据显示器的不同来调整合适的图像分辨率。

在 ov5640_init（）函数中设置了摄像头的相关参数。

文件复制完成后重新编译工程，接下来就可以下载到开发板上进行验证。

20.4 下载验证

按照之前的方式连接好各个接口。将 ov5640 摄像头模块插到 GPIO1 接口上。连接方式如下图所示：

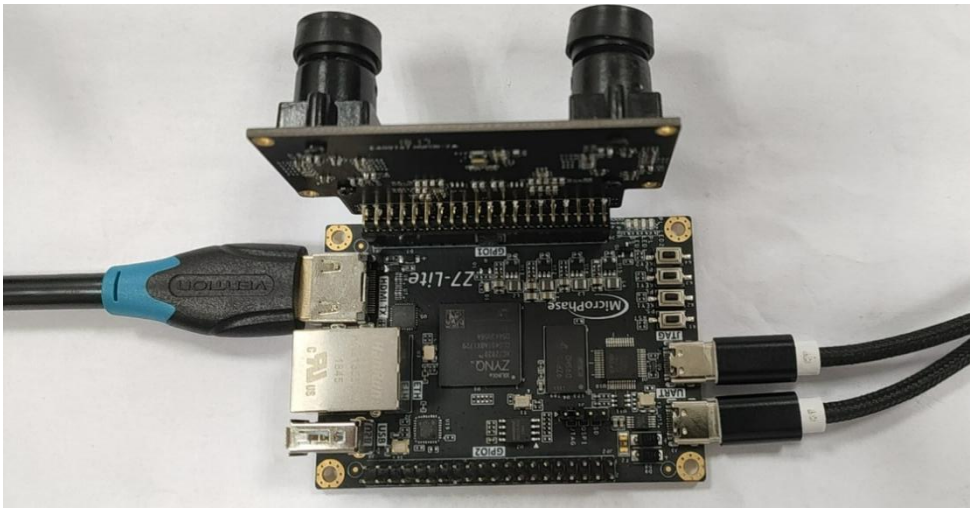


图 20.4.1 摄像头连接方式

最后将 JTAG 线插入电脑 USB 接口给开发板上电，之后将程序烧录到开发板中。烧录完成后通过 HDMI 连接线将开发板 HDMI 接口和显示器连接起来，即可在显示器上看到摄像头采集到的实时画面，如下图所示：

图 20.4.2 摄像头实时画面

20.5 章节小结

在本实验中，我们通过软件设置的方式实现了对于摄像头的参数配置。而在我们的 ov5640_capture IP 核上也具有 SCCB 的信号接口，可以利用 IP 核来实现对于 ov5640 进行设置。但是如果要对 ov5640 的设置进行修改就需要重新修改并重新封装 IP 核。由此我们也可以看出 zynq 芯片将 arm 和 fpga 结合的优势。它既有 arm 带来的软件设计的灵活，又有 FPGA 硬件编程带来的速度优势。这就要求我们在实际使用过程中认真分析项目各部分的内在需求，用最合适的方式去实现性能要求。

第 21 章　　双目摄像头 HDMI 显示实验

在之前的实验中，我们实现了摄像头采集图像通过 HDMI 接口显示在显示屏上的功能。相信读者已经发现我们的摄像头模块上面是两个 OV5640 摄像头，这是因为在安防监控、人脸识别、三位视觉建模等应用领域，单个摄像头无法实现。为满足不同用户需求，特意设计了集成两个摄像头的模组。在本实验中，我们就利用该模组实现双目摄像头采集图像并同时显示在 HDMI 显示屏上的功能。

本章包括以下几部分内容：

21.1 实验简介

21.2 硬件设计

21.3 软件设计

21.4 下载验证

21.5 章节小结

21.1 实验简介

在我们的日常生活中，摄像头的身影出现在各个角落，各种种类也是花样繁多，在不同场景下分为单目摄像头、双目摄像头和多目摄像头。单目摄像头常见于监控，摄像拍照，违章监控等领域；在三维空间建模、双目视觉等单目摄像头无法胜任的场合使用双目摄像头；在一些更加复杂的场合下，会使用结合红外摄像头、长焦摄像头等各种不同类型摄像头的多目摄像头。

在本次实验中，我们将使用两个摄像头分别采集两路视频数据，之后将数据进行合成，在显示屏上同时显示两路视频图像。在图像处理上，我们使用一个 Xilinx 官方提供的专门用于视频混合显示以及文本叠加的视频处理 IP，OSD（On Screen Display）IP。该 IP 可最大支持 8 层显示图层的叠加，包括外部输入视频和内部图形控制器，所有图层位置、大小、背景色、透明度都可进行配置。每个图层可以设置优先级，显示在同一位置的图层，优先级高的图层会覆盖到优先级低的图层之上。

该 IP 核的输入输出接口全部支持 AXI4-Stream 接口协议。输入接口为 AXI4-Stream Slave Interface，输出接口为 AXI4-Stream Master Interface，使其具有很好的适配性，可以方便与其他支持 AXI4-Stream 协议的 IP 核集成在一起。

21.2 硬件设计

本实验的硬件设计是在之前实验的基础上，增加一路摄像头图像数据读取和增加一个 OSD IP 来实现图像的叠加处理。所以此处不需要新建 Vivado 工程，只需打开前一实验的工程，另存为一个新的工程并命名为“double_ov5640”，即可，之后我们在此基础上进行设计。如下图所示：

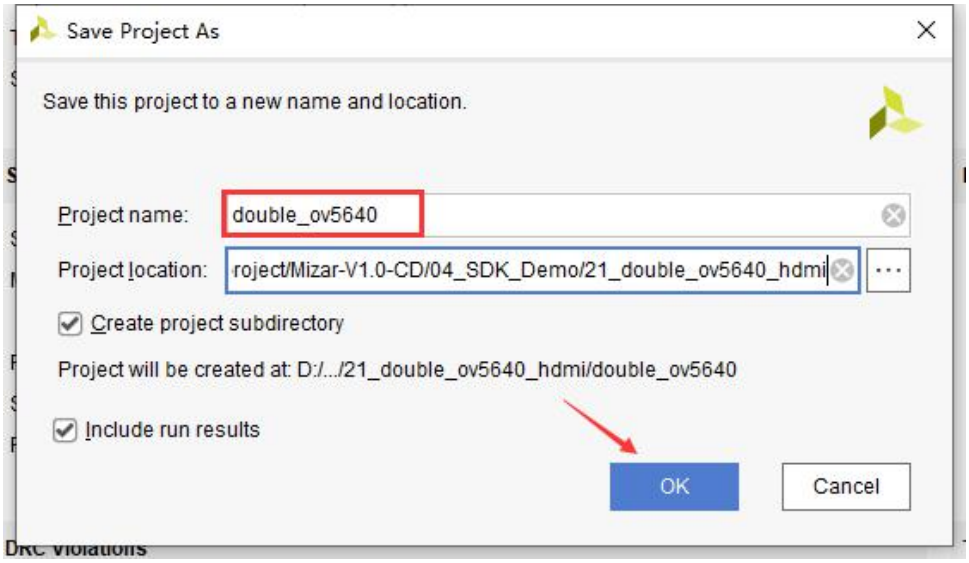


图 21.2.1 另存为“double_ov5640”工程

之后删除工程文件夹下的.sdk 文件夹，这是之前实验的应用工程，我们不需要，如下图所示：

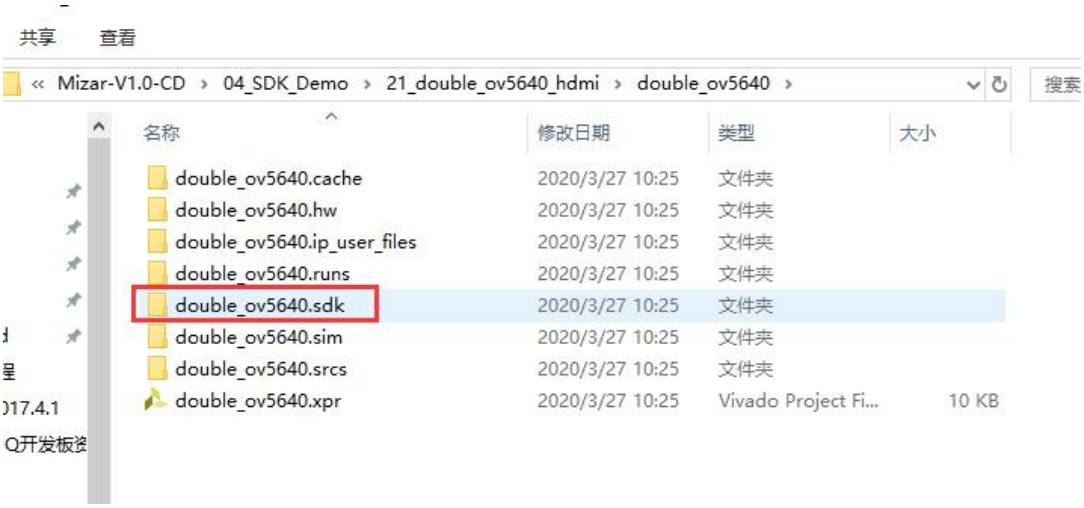


图 21.2.2 删除.sdk 文件夹

另存完成后自动打开另存的“double_ov5640”工程，点击 Open Block Design 打开设计界面。双击 ZYNQ7 Processing System 打开处理器配置窗口。点击 MIO Configuration 选项，将 EMIO 的位宽从 2 修改为 4，增加两个 I/O 作为 sccb 接口模拟 IIC 协议配置第二个摄像头。如下图所示：

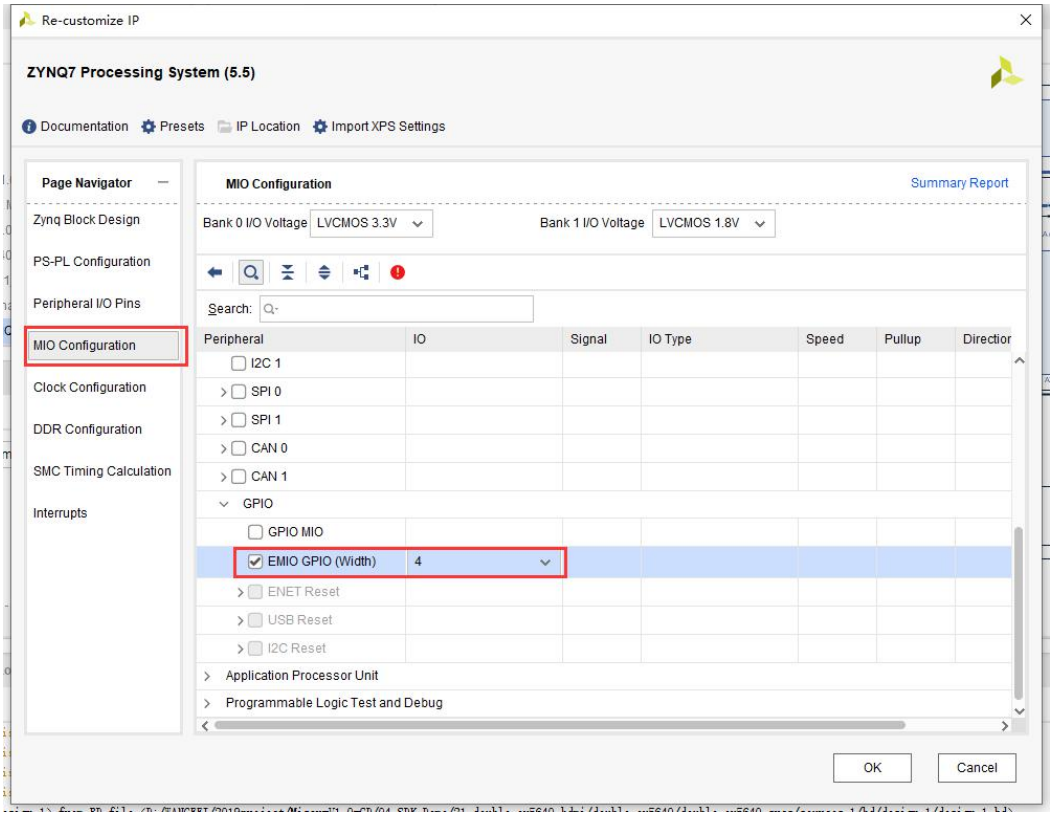
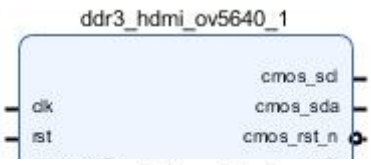
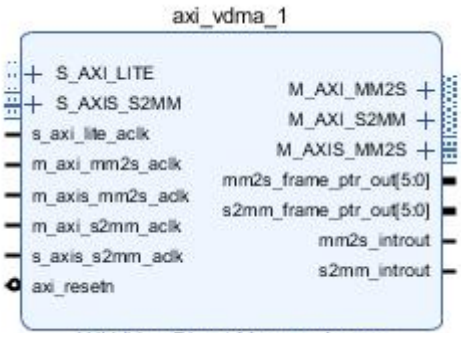
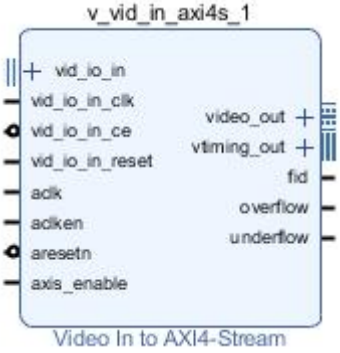


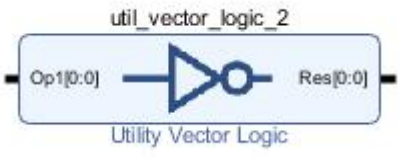
图 21.2.3 设置 EMIO 位宽为 4

设置完成后点击 OK。之后添加如下几个 IP 作为第二路摄像头的图像采集，





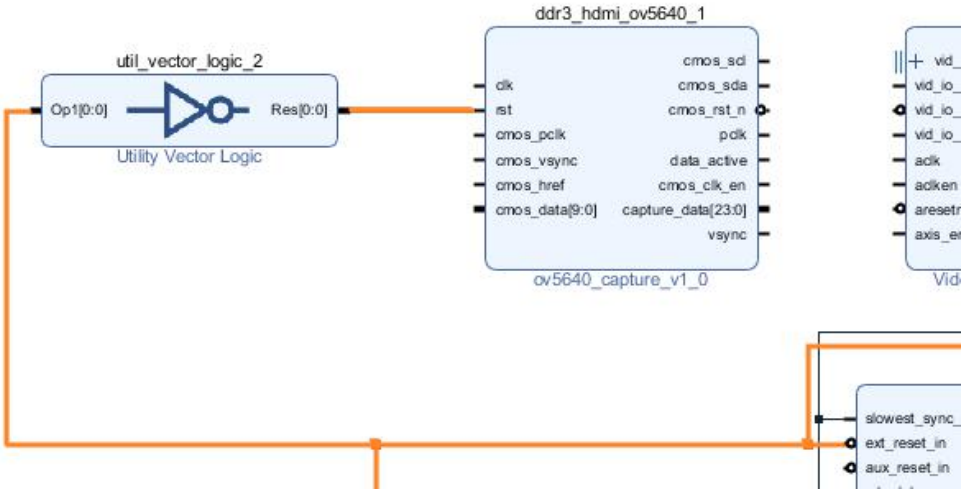




如下图所示：

图 21.2.4 添加 IP

这四个 IP 的设置与已存在的 IP 设置相同，参照相同 IP 进行设置即可。设置完成后手动连接各 IP 之间的相关接口。如下图所示，连接 ddr3_hdmi_ov5640_1 的 rst 接口和 util_vector_logic_2 的 Res[0:0]接口，连接



util_vector_logic_2 的 Op1[0:0]接口和系统的复位信号。

图 21.2.5 连接复位信号

连接 ddr3_hdmi_ov5640_1 和 v_vid_in_axi4s_1 的相关接口，如下图所示：

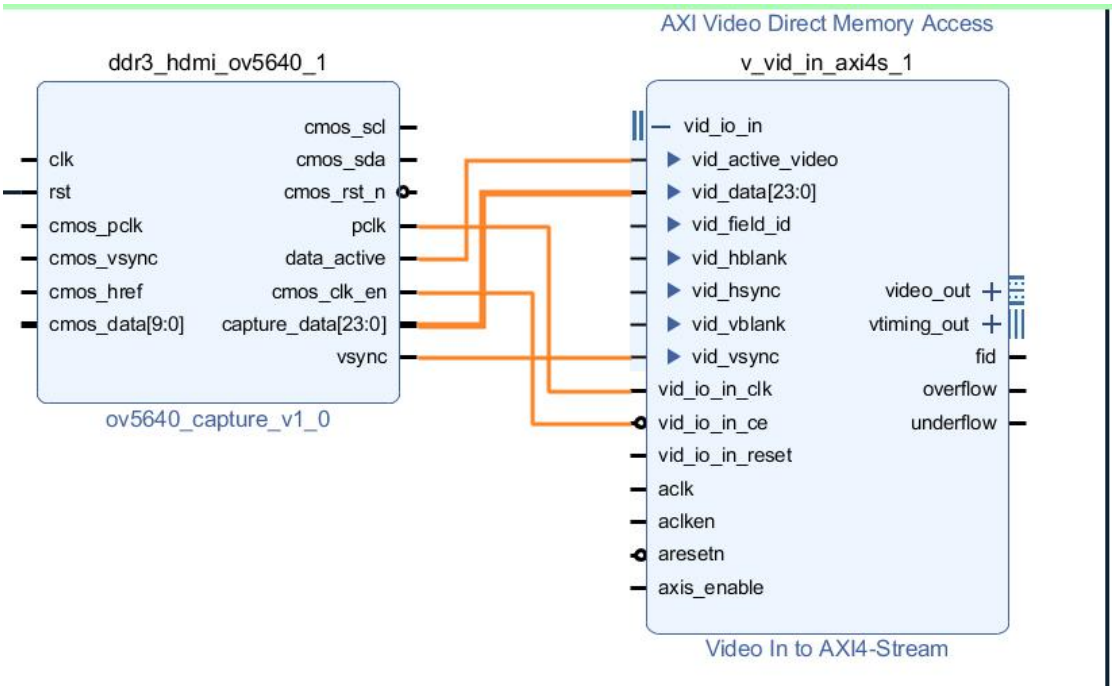
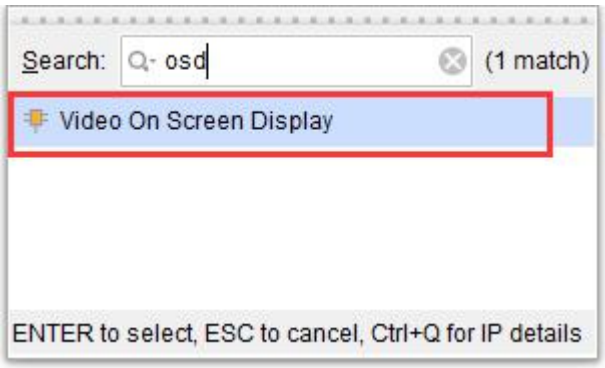


图 21.2.6 ddr3_hdmi_ov5640_1 和 v_vid_in_axi4s_1 相关接口的连接关系

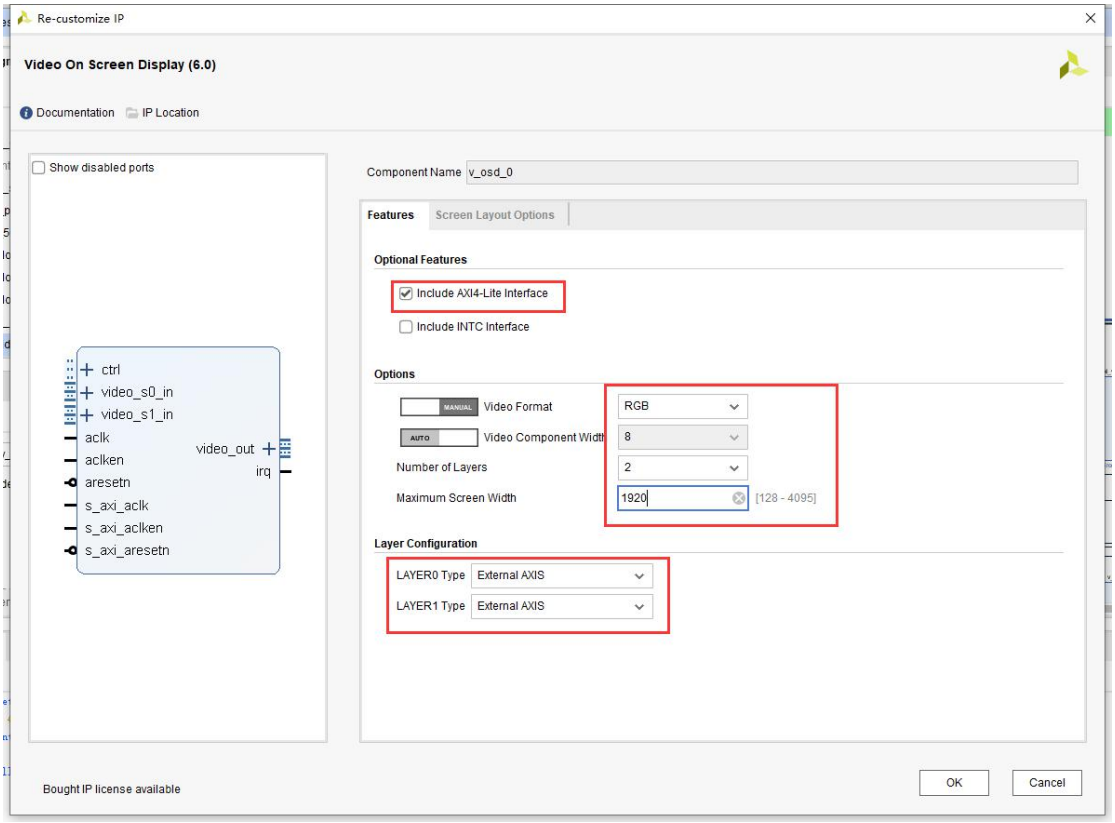
连接信号完成后添加 OSD IP 核，点击“+”打开 IP 列表，在搜索框中输入



“osd”，搜索添加 Video On Screen Display IP 核，如下图所示：

图 21.2.7 搜索添加 Video On Screen Display IP 核

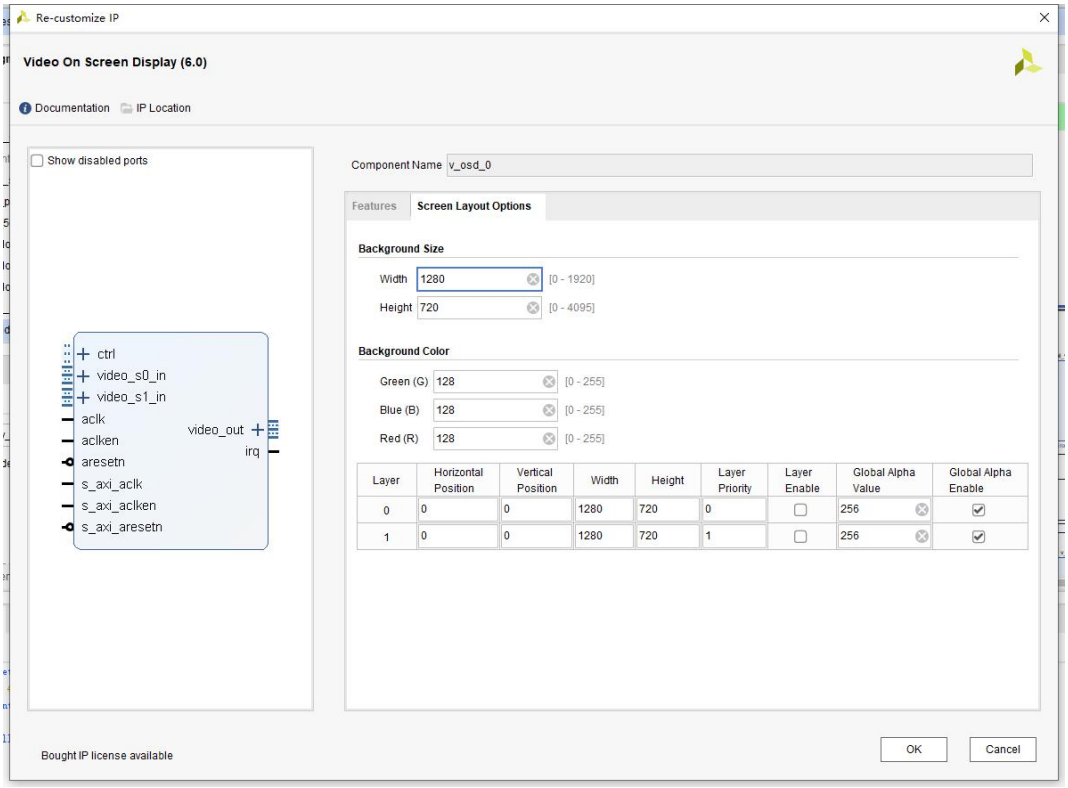
添加完成后双击该 IP 核打开设置页面，如下图所示配置相关设置。使能 AXI4-Lite 总线接口，设置输入视频格式为 RGB，位宽为 8，即 RGB888 格式也就是摄像头图像采集 VDMA 输出的视频格式。设置图层数量，我来使用双目摄像头，所以设置图层数量为 2。最下方设置图层类型，即图像是来自外部输入还



是内部图形控制器，此处全部设置为外部输入。

图 21.2.8 设置相关设置

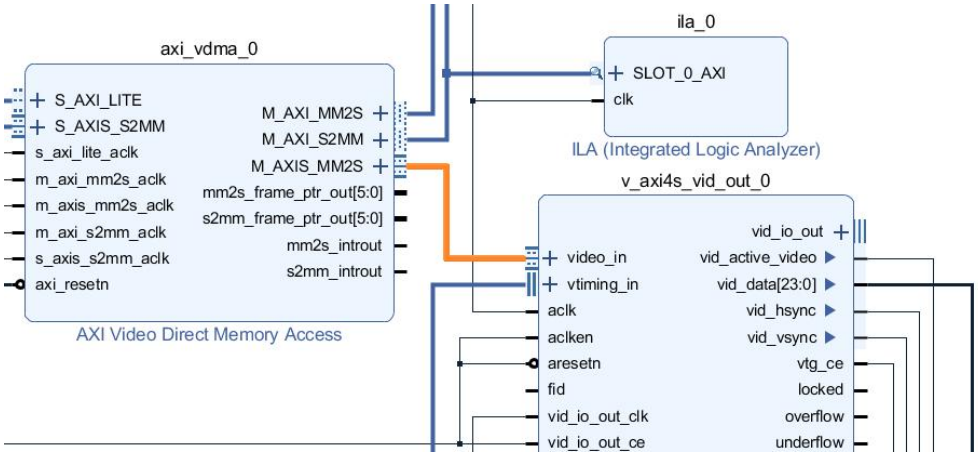
点击打开 Screen Layout Options 选项页面，在此处可以设置背景的大小即颜色，每个图层的大小、位置、透明度等参数，这些参数也可通过程序设置。此处我们不进行设置，在之后的软件设计中再设置这些参数，学习怎样通过软件



控制 osd IP 核。

图 21.2.9 图层参数设置

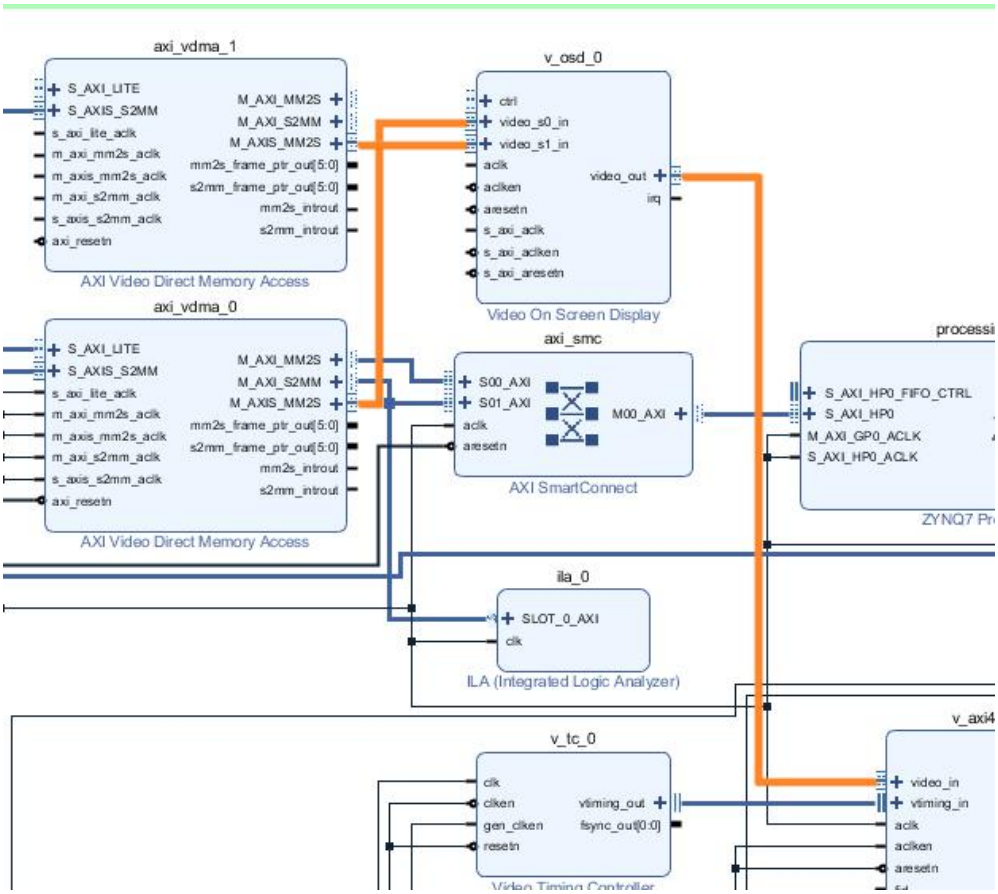
之后删除 VDMA_0 的 M_AXIS_MM2S 接口和 v_axi4_vid_out_0 的 video_in 之间的连线。本实验中我们将 VDMA 输出的视频数据传输到 OSD IP 中进行处理，



之后再传输到 v_axi4_vid_out_0 进行显示。

图 21.2.10 删除连线

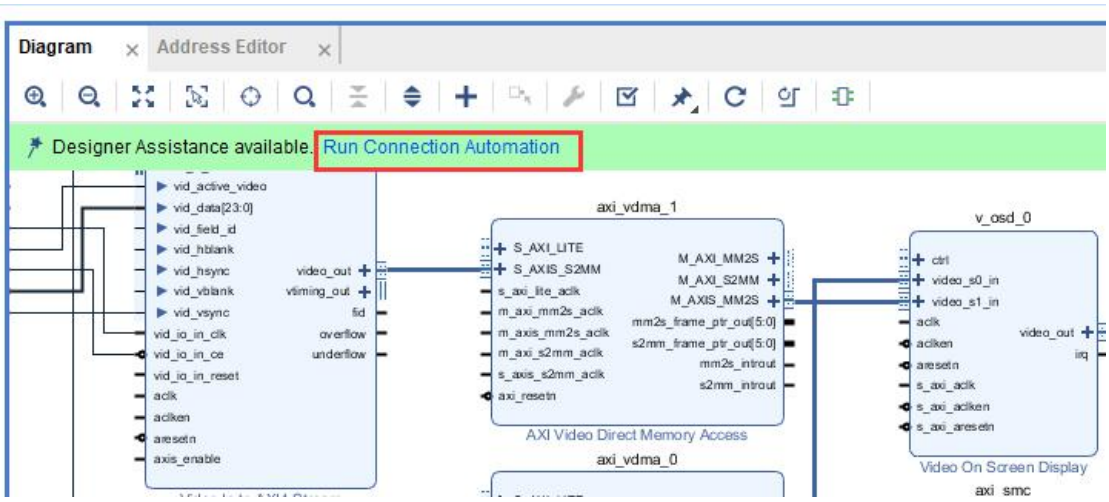
之后再将 VDMA_0 和 VDMA_1 输出的视频传输到 OSD IP，处理之后传输到



v_axi4_vid_out_0，如下图所示：

图 21.2.11 连接视频数据传输

完成需要手动连接的信号之后，点击 Run Connection Automation 自动连接



其他信号。再弹出的对话框中全选所有信号，点击 OK 完成连接。如下图所示：

图 21.2.12 自动连接信号

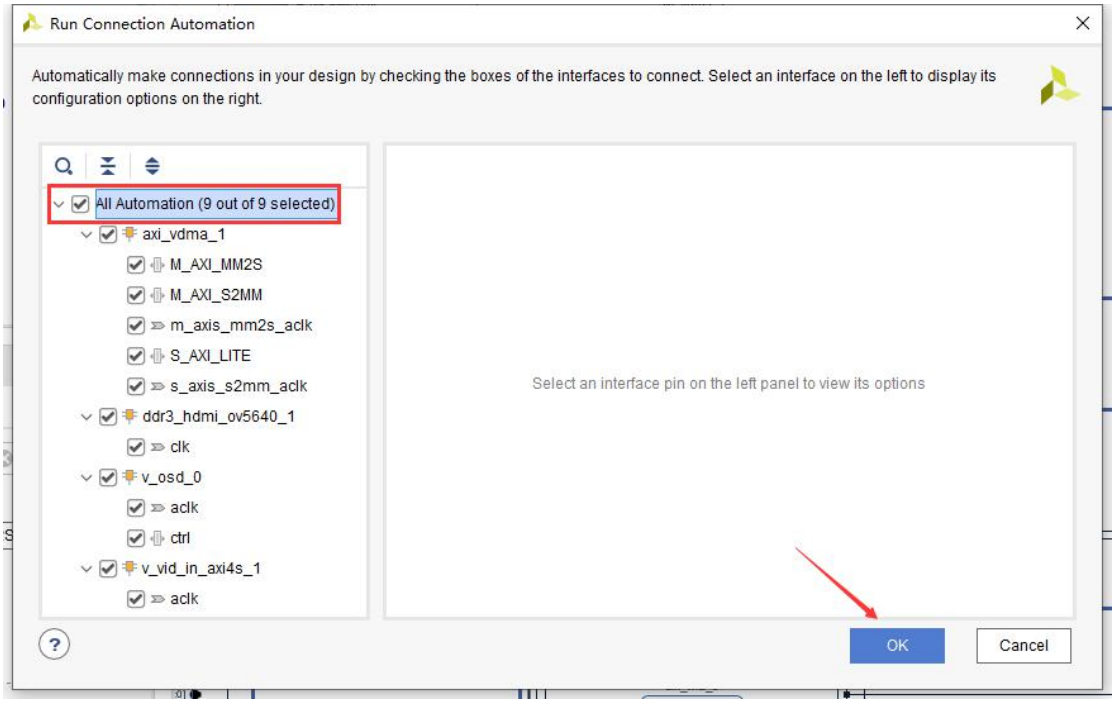


图 21.2.13 全选所有信号

完成信号连接之后，将需要和模块连接的端口引出。右键点击 ddr3_hdmi_ov5640_1IP 的相关端口，选择 Make External 引出端口信号。需要引出的信号如下图所示：

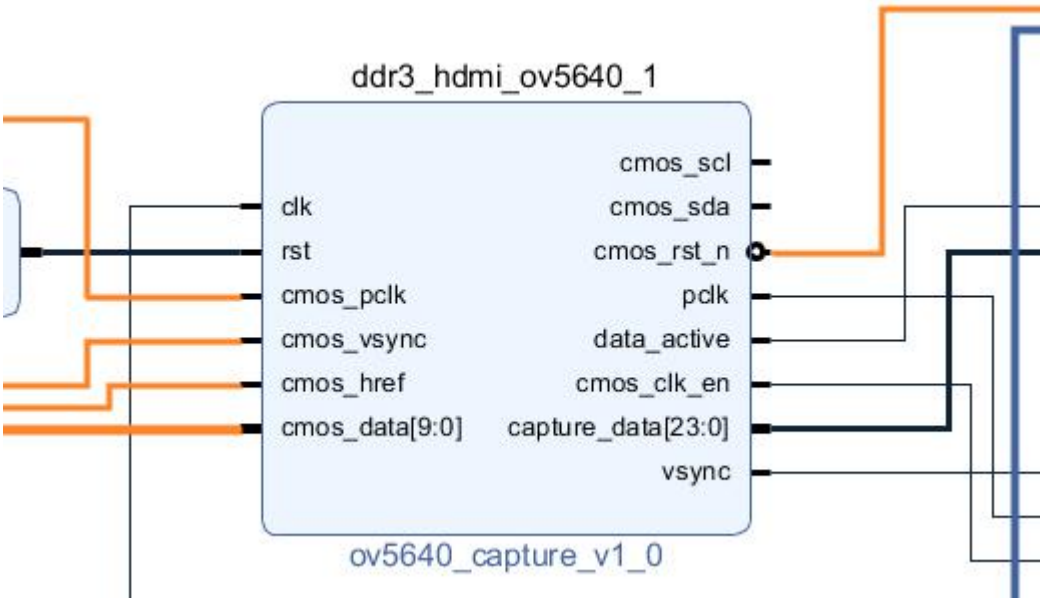


图 21.2.14 引出端口信号

完成之后点击 Regenerate Layout 重新布局，布局完成后的整个设计如下图所示：

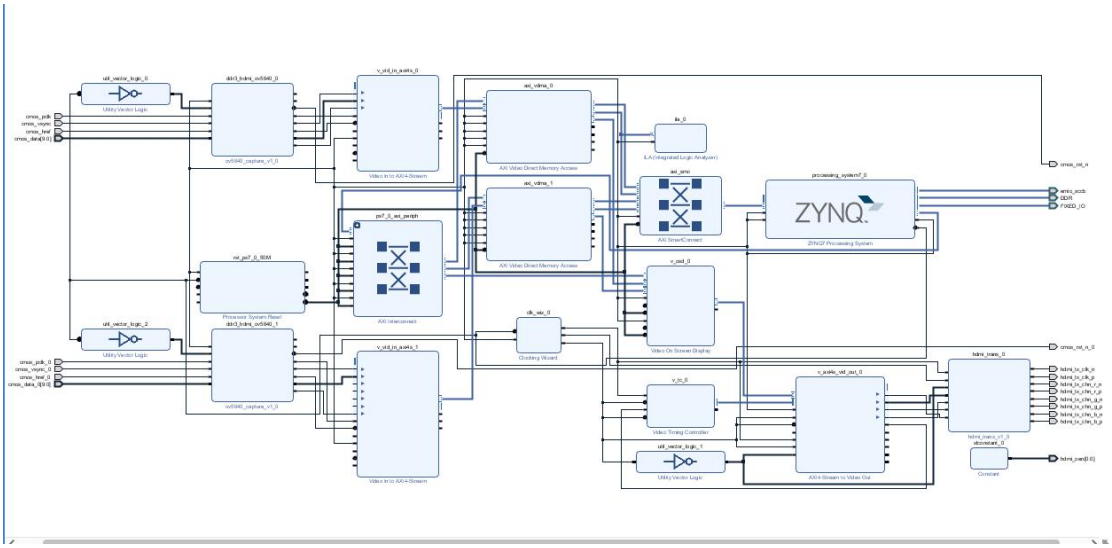
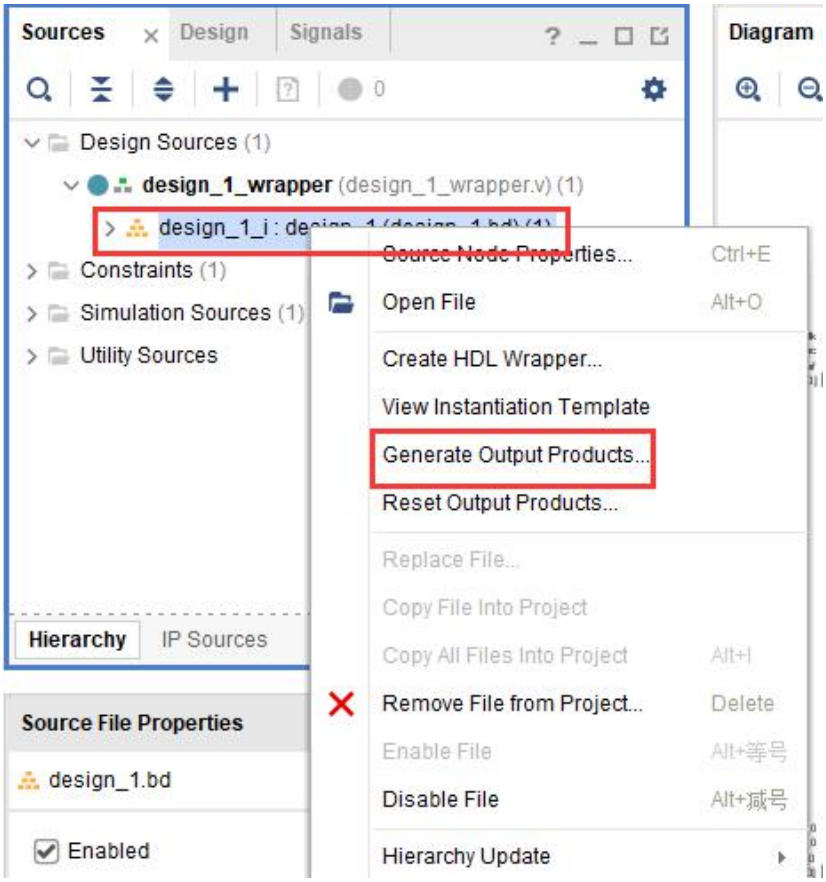


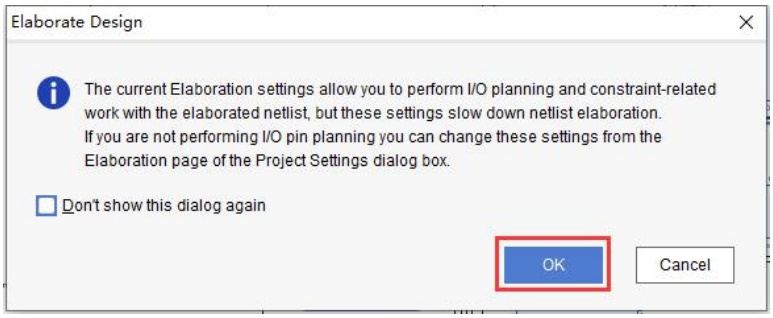
图 21.2.15 系统整体框图

点击 Validate Design 验证设计是否存在错误，没有错误则在 Source 窗口右



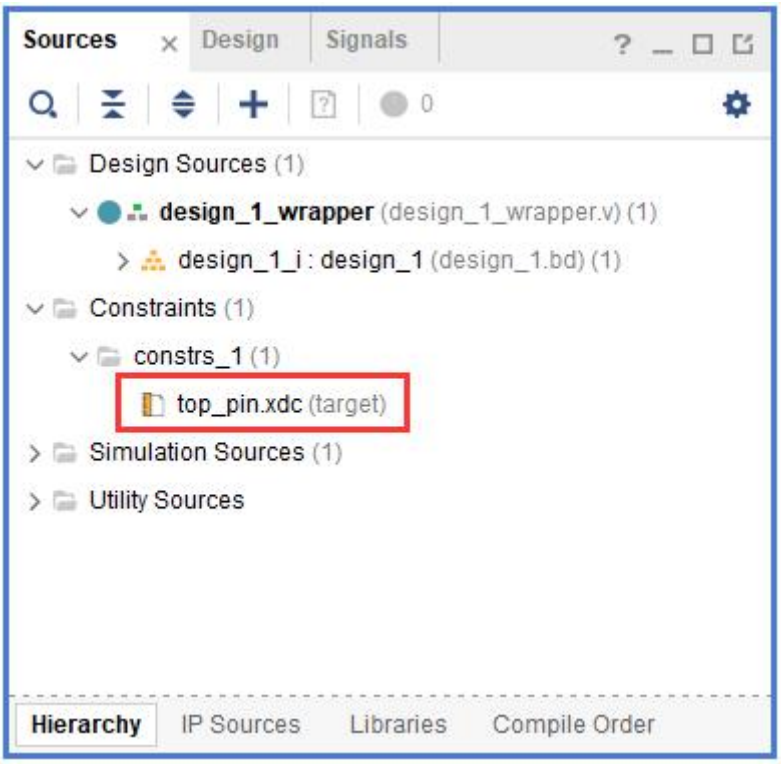
键点击设计重新生成输出工程，软件自动更新顶层 HDL 文件，如下图所示：

图 21.2.16 重新生成输出文件



在弹出的对话框中点击 OK。

图 21.2.17 确认重新生成



之后双击打开引脚约束文件.xdc，如下图所示：

图 21.2.18 引脚约束文件

```
set_property PACKAGE_PIN N17 [get_ports cmos_pclk]
set_property PACKAGE_PIN W14 [get_ports cmos_href]
set_property PACKAGE_PIN V18 [get_ports cmos_vsync]
set_property PACKAGE_PIN Y14 [get_ports cmos_rst_n]


set_property PACKAGE_PIN U17 [get_ports {cmos_data[9]}]
set_property PACKAGE_PIN T16 [get_ports {cmos_data[8]}]
set_property PACKAGE_PIN W18 [get_ports {cmos_data[7]}]
set_property PACKAGE_PIN W19 [get_ports {cmos_data[6]}]
```

向其中添加第二个摄像头相关的引脚约束，全部引脚分配如下图所示：

```
38 set_property IOSTANDARD LVCMOS33 [get_ports {emio_sccb_tri_io[1]}]
39 set_property IOSTANDARD LVCMOS33 [get_ports {emio_sccb_tri_io[0]}]
40 set_property PACKAGE_PIN P18 [get_ports {emio_sccb_tri_io[0]}]
41 set_property PACKAGE_PIN R16 [get_ports {emio_sccb_tri_io[1]}]
42
43 set_property IOSTANDARD LVCMOS33 [get_ports {emio_sccb_tri_io[2]}]
44 set_property IOSTANDARD LVCMOS33 [get_ports {emio_sccb_tri_io[3]}]
45 set_property PACKAGE_PIN T17 [get_ports {emio_sccb_tri_io[2]}]
46 set_property PACKAGE_PIN W13 [get_ports {emio_sccb_tri_io[3]}]
47
48 set_property IOSTANDARD LVCMOS33 [get_ports cmos_href_0]
49 set_property IOSTANDARD LVCMOS33 [get_ports cmos_vsync_0]
50 set_property IOSTANDARD LVCMOS33 [get_ports cmos_rst_n_0]
51 set_property IOSTANDARD LVCMOS33 [get_ports cmos_pclk_0]
```

```
11 set_property PACKAGE_PIN Y18 [get_ports {cmos_data[5]}]
12 set_property PACKAGE_PIN Y16 [get_ports {cmos_data[4]}]
13 set_property PACKAGE_PIN R17 [get_ports {cmos_data[3]}]
14 set_property PACKAGE_PIN Y19 [get_ports {cmos_data[2]}]
15 set_property PACKAGE_PIN V17 [get_ports {cmos_data[1]}]
16 set_property PACKAGE_PIN Y17 [get_ports {cmos_data[0]}]
17
18 set_property IOSTANDARD LVCMOS33 [get_ports cmos_rst_n]
19 set_property IOSTANDARD LVCMOS33 [get_ports cmos_pclk]
20 set_property IOSTANDARD LVCMOS33 [get_ports cmos_vsync]
21 set_property IOSTANDARD LVCMOS33 [get_ports cmos_href]
22 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data[*]}]
23
```

```
52 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[9]}]
53 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[8]}]
54 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[7]}]
55 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[6]}]
56 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[5]}]
57 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[4]}]
58 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[3]}]
59 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[2]}]
60 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[1]}]
61 set_property IOSTANDARD LVCMOS33 [get_ports {cmos_data_0[0]}]
62 set_property PACKAGE_PIN U14 [get_ports {cmos_data_0[0]}]
63 set_property PACKAGE_PIN U15 [get_ports {cmos_data_0[1]}]
64 set_property PACKAGE_PIN W15 [get_ports {cmos_data_0[2]}]
65 set_property PACKAGE_PIN T15 [get_ports {cmos_data_0[3]}]
66 set_property PACKAGE_PIN R14 [get_ports {cmos_data_0[4]}]
67 set_property PACKAGE_PIN P14 [get_ports {cmos_data_0[5]}]
68 set_property PACKAGE_PIN T11 [get_ports {cmos_data_0[6]}]
69 set_property PACKAGE_PIN V15 [get_ports {cmos_data_0[7]}]
70 set_property PACKAGE_PIN T10 [get_ports {cmos_data_0[8]}]
71 set_property PACKAGE_PIN T14 [get_ports {cmos_data_0[9]}]
72 set_property PACKAGE_PIN U12 [get_ports cmos_rst_n_0]
73 set_property PACKAGE_PIN T12 [get_ports cmos_href_0]
74 set_property PACKAGE_PIN V13 [get_ports cmos_vsync_0]
75 set_property PACKAGE_PIN V12 [get_ports cmos_pclk_0]
```

图 21.2.19 引脚分配及引脚电压设置

设置完成之后点击保存即可。对于其他型号的开发板，请参照原理图进行引脚分配。

最后在左侧 Flow Navigator 导航栏中点击 “GenerateBitstream”，对设计进行综合、实现、并生成 Bitstream 文件。

在生成 Bitstream 之后， 在菜单栏中选择 File -> Export -> Export hardware 导出硬件，并在弹出的对话框中，勾选“Include bitstream”。然后在菜单栏选择 File- > Launch SDK，启动 SDK 软件。

21.3 软件设计

将 Bitstream 导出到 SDK 后打开 SDK，按照之前的方式创建新的应用工程并

命名为“double_ov5640”，工程模板选择空白模板。创建完成之后，打开打开资料文件内的例程文件夹，如下图所示将以下几个文件及文件夹拷贝到新建工程的相同位置。

图 21.3.1 工程源代码

相较于之前的实验，因为新增加了一个摄像头，所以摄像头配置程序进行了修改，同时增加了对于 osd IP 核配置的程序。

main 文件的程序如下所示：

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xil_types.h"
#include "xil_cache.h"
#include "xparameters.h"
#include "xaxivdma.h"
#include "xaxivdma_i.h"
#include "vdma_api/vdma_api.h"
#include "emio_sccb_cfg/emio_sccb_cfg.h"
#include "ov5640/ov5640_init.h"
#include "osd/osd.h"
////宏定义
#define VDMA_ID0                XPAR_AXIVDMA_0_DEVICE_ID    //VDMA器件ID
#define VDMA_ID1                XPAR_AXIVDMA_1_DEVICE_ID    //VDMA器件ID
#define VDMA_BASE_ADDR0         XPAR_PS7_DDR_0_S_AXI_BASEADDR+0x1000000*0       //VDMA基地址
#define VDMA_BASE_ADDR1         XPAR_PS7_DDR_0_S_AXI_BASEADDR+0x1000000*1       //VDMA基地址
#define SCREEN_X                1024
#define SCREEN_Y                768
//全局变量
XAxiVdma     vdma0,vdma1;

int main(void)
{
    u32 status0,status1;
    u16 cmos_h_pixel;                       //ov5640 DVP 输出水平像素点数
    u16 cmos_v_pixel;                       //ov5640 DVP 输出垂直像素点数
    u16 total_h_pixel;                      //ov5640 水平总像素大小
    u16 total_v_pixel;                      //ov5640 垂直总像素大小
    cmos_h_pixel    = 1024;                 //设置OV5640输出分辨率
    cmos_v_pixel    = 768;
    total_h_pixel   = 2570;
    total_v_pixel   = 980;
```

```c
    emio_init();                                    //初始化EMIO
    status0 = ov5640_init(CMOS_CH0,cmos_h_pixel,//初始化ov5640 0
                          cmos_v_pixel,
                         total_h_pixel,
                         total_v_pixel);
    status1 = ov5640_init(CMOS_CH1,cmos_h_pixel,//初始化ov5640 1
                          cmos_v_pixel,
                         total_h_pixel,
                         total_v_pixel);
    if(status0 == 0 && status1 == 0)
        xil_printf("Dual OV5640 detected successful!\r\n");
    else
        xil_printf("Dual OV5640 detected failed!\r\n");

    //配置VDMA
    run_vdma_frame_buffer(&vdma0, VDMA_ID0, SCREEN_X/2, SCREEN_Y,
            VDMA_BASE_ADDR0,0,0,BOTH);
    run_vdma_frame_buffer(&vdma1, VDMA_ID1, SCREEN_X/2, SCREEN_Y,
            VDMA_BASE_ADDR1,0,0,BOTH);

    //OSD初始化及设置
    osd_init(SCREEN_X,SCREEN_Y);

    return 0;
}
```

在程序中首先分别配置两个摄像头的相关参数，之后配置两个 VDMA，最

后设置 OSD IP 的相关设置。即可将两路视频显示到显示屏上。

```
//OSD初始化
void osd_init(int screen_width,int screen_height)
{
    OsdCfgPtr = XOSD_LookupConfig(OSD_DEVICE_ID);
    XOSD_CfgInitialize(&Osd, OsdCfgPtr, OsdCfgPtr->BaseAddress); //初始化OSD IP核
    XOSD_RegUpdateEnable(&Osd);                  //寄存器更新使能
    XOSD_Enable(&Osd);                           //使能OSD IP核
    osd_layer_cfg(screen_width,screen_height); //OSD参数及各图层参数配置
}

//OSD配置
void osd_layer_cfg(int screen_width,int screen_height)
{
    //配置OSD输出分辨率
    XOSD_SetScreenSize(&Osd, screen_width, screen_height);
    xil_printf("OSD output resolution is %d,%d\r\n",screen_width,screen_height);
    //配置OSD背景色   颜色参数:RED BLUE GREEN
    XOSD_SetBackgroundColor(&Osd,0x00,0xff,0x00);
    //图层一设置
    OSD_LCfg = OSD_Layer0;
    XOSD_SetLayerAlpha(&Osd,OSD_LCfg.index,OSD_LCfg.alpha_enable,OSD_LCfg.alpha_value);//设置透明度
    XOSD_SetLayerPriority(&Osd,OSD_LCfg.index,OSD_LCfg.priority);//设置优先级
    XOSD_SetLayerDimension(&Osd,OSD_LCfg.index,0,0,screen_width/2,screen_height);//设置图层大小
    XOSD_EnableLayer(&Osd,OSD_LCfg.index);
```

设置 OSD IP 的程序如下所示：

```
    //图层二设置
    OSD_LCfg = OSD_Layer1;
    XOSD_SetLayerAlpha(&Osd,OSD_LCfg.index,OSD_LCfg.alpha_enable,OSD_LCfg.alpha_value);//设置透明度
    XOSD_SetLayerPriority(&Osd,OSD_LCfg.index,OSD_LCfg.priority);//设置优先级
    XOSD_SetLayerDimension(&Osd,OSD_LCfg.index,screen_width/2,0,screen_width/2,screen_height);//设置图层大小
    XOSD_EnableLayer(&Osd,OSD_LCfg.index);

    xil_printf("OSD initial finish!\r\n" );
}
```

配置流程如下：首先初始化 IP 核，之后使能寄存器更新，否则之后写入的参数无效，它会使用在硬件设计时配置的参数，之后使能寄存器，接下来设置 IP 核输出视频的分辨率大小和背景色，以及各图层的各个参数。

源文件复制完成后重新编译工程，没有错误即可下载到开发板上进行验证。

21.4 下载验证

按照之前的方式连接好各个接口。将 ov5640 摄像头模块插到 GPIO1 接口上。连接方式如下图所示：

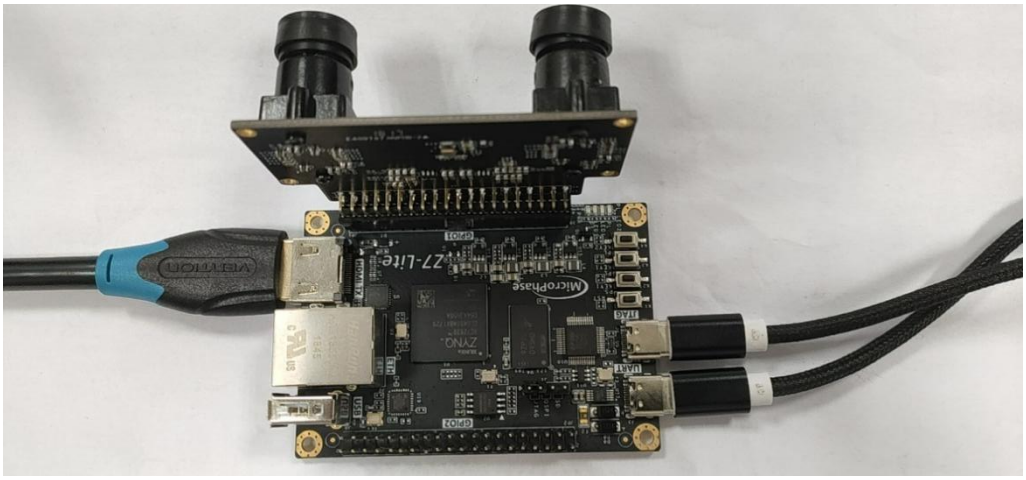


图 21.4.1 摄像头连接方式

连接完成，将程序烧录到开发板中。烧录完成后通过 HDMI 连接线将开发板 HDMI 接口和显示器连接起来，即可在显示器上看到摄像头采集到的实时画面，如下图所示：

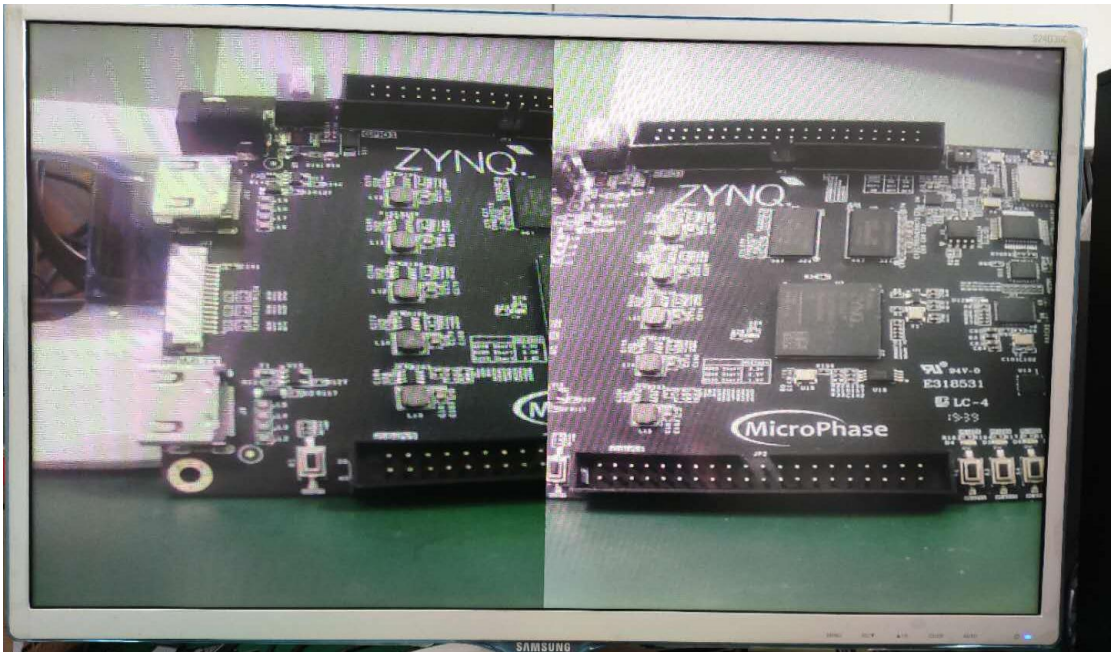


图 21.4.2 摄像头采集画面

说明实验成功。

21.5 章节小结

在本实验中，我们实现了双目摄像头采集图像，并同时将两路图像同时显示到屏幕上的功能。同时还学习了 OSD IP 的使用方式，更加深入了解 Xilinx 提供的丰富的 IP 资源。希望读者可以在此实验的基础上尝试实现添加更多图层，例如添加文本显示等内容。

第 22 章　　DAC 波形发生器实验

在嵌入式开发过程中，模数转换和数模转换是经常会使用到的两种功能。模数转换可以用来获取数据，检测运行状态，数模转换能产生合适的控制波形来控制其他部分的运行。为满足部分用户的需求，我们推出了一款高速 ADDA 的子卡 ADA-106，用户可以借助这款子卡来学习 AD、DA 的使用方法，或者利用它来开发自己的项目。本实验我们就使用其 DAC 的功能来实现一个波形发生器。

本章包括以下几部分内容：

22.1 实验简介

22.2 硬件设计

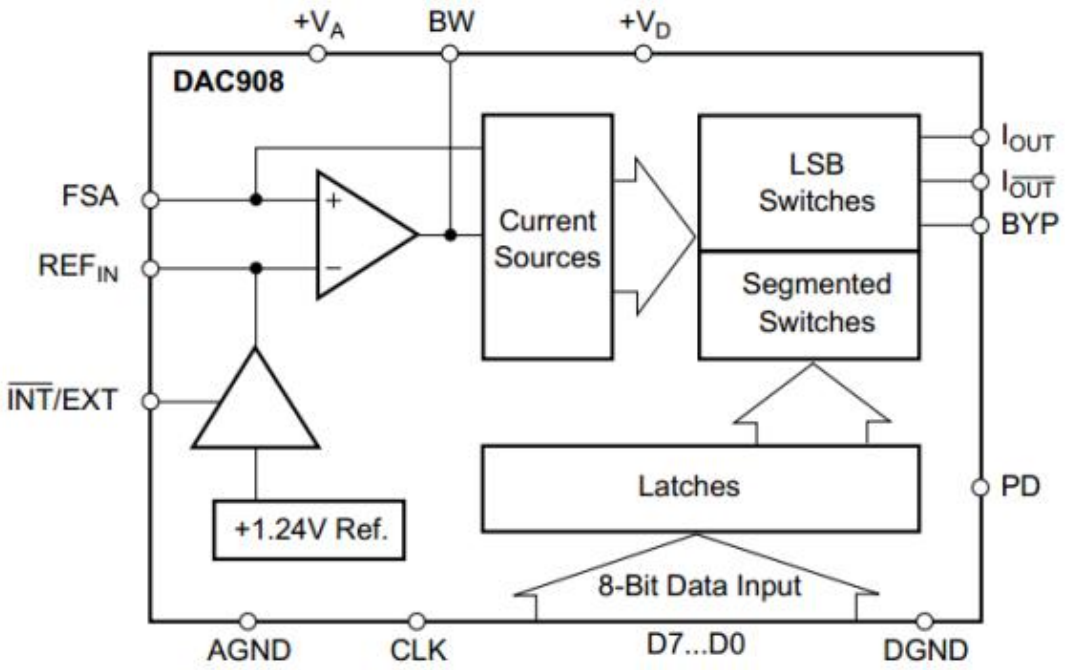
22.3 软件设计

22.4 下载验证

22.5 章节小结

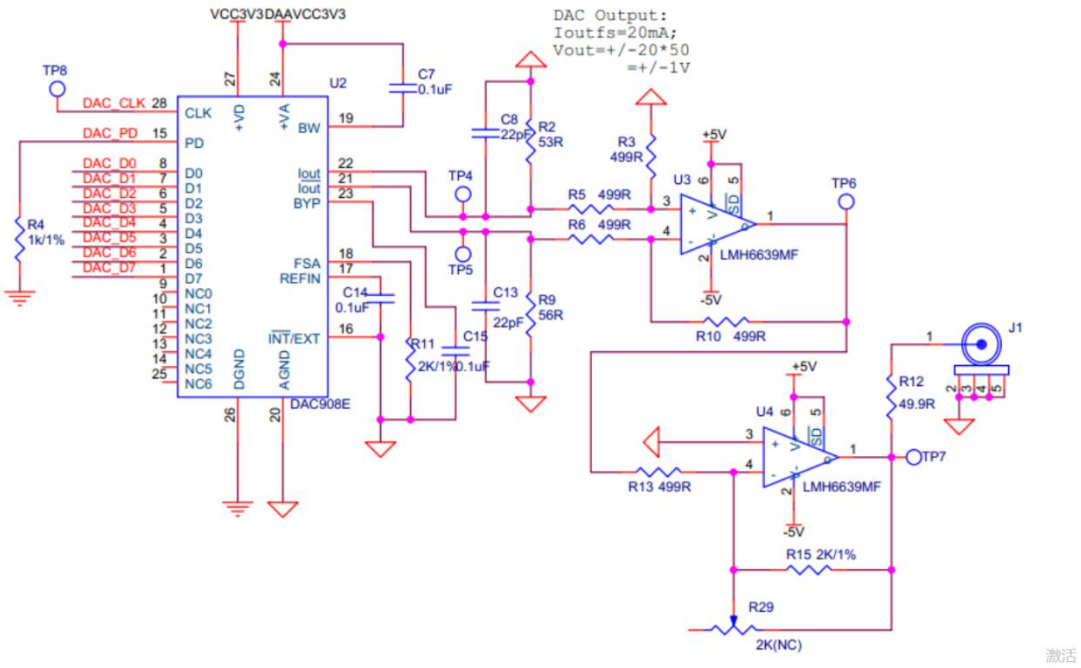
22.1 实验简介

该子卡使用的高速 DA 转换芯片是 TI 出产的 DAC908E，最大转换速率



165MSPS，模拟输出范围为+/-5Vpp，其结构图如下图所示：

图 22.1.1 DAC908E 结构图



相关部分原理图如下图所示：

图 22.1.2 DAC 部分原理图

高速 ADDA 子卡采用内部参考电压 1.24V, Rset（R11）取值 2K 欧姆，根据以下计算公式（计算公式来自于芯片 datasheet），其中 Ioutfs 为 DAC 单端最大

$$I_{OUTFS} = 32 \cdot I_{REF} = 32 \cdot V_{REF} / R_{SET}$$

量程电流输出。

可得 Ioutfs=20mA， DAC 电流输出如下：

$$I_{OUT} = I_{OUTFS} \cdot (Code/256)$$

$$I_{\overline{OUT}} = I_{OUTFS} \cdot (255 - Code/256)$$

算上负载电路 Rload，可得输出电压如下：

本次实验需要实现的功能为通过自定义 IP 核向 DAC 芯片传输波形数据使芯片输出波形，并可以通过按键切换输出的波形。为实现该功能，我们添加 DMA IP

$$V_{OUT} = I_{OUT} \cdot R_{LOAD}$$

$$V_{\overline{OUT}} = I_{\overline{OUT}} \cdot R_{LOAD}$$

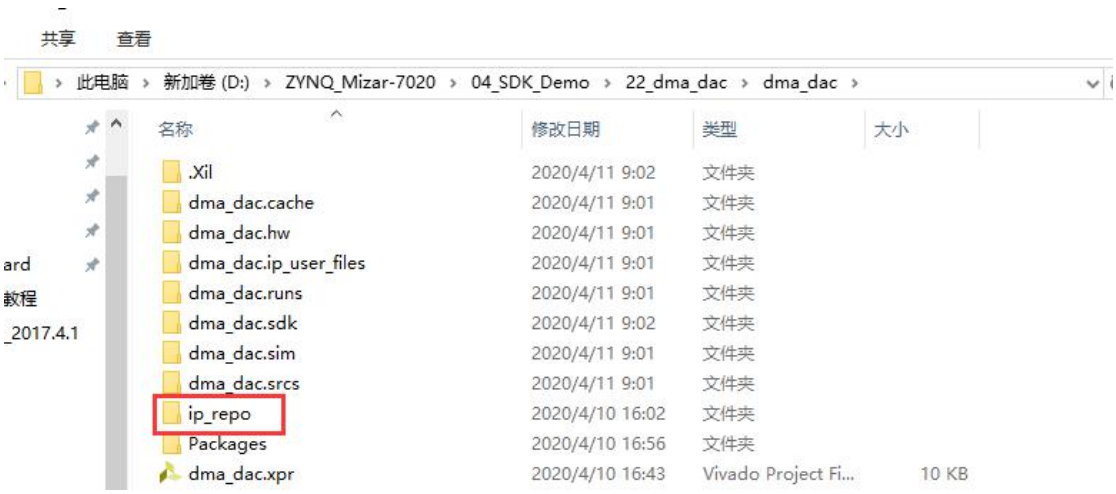
核读取 DDR 中的数据并向自定义 IP 核传输数据，之后自定义 IP 核将该数据转换为 DAC 芯片可以接受的数据形式传送给芯片实现波形输出。自定义 IP 的创建不是本实验的重点，此处不再详细介绍，读者可以直接使用资料中包含的文件。并在此基础上尝试自行编写该 IP 并拓展其功能。

22.2 硬件设计

按照之前的步骤创建新的 Vivado 工程并命名为“dma_dac”，根据自己开发板的型号选择对应芯片型号，本教程以 xc7z020clg400-1 为例，创建完成的工程如下图所示：

图 22.2.1 创建 Vivado 工程

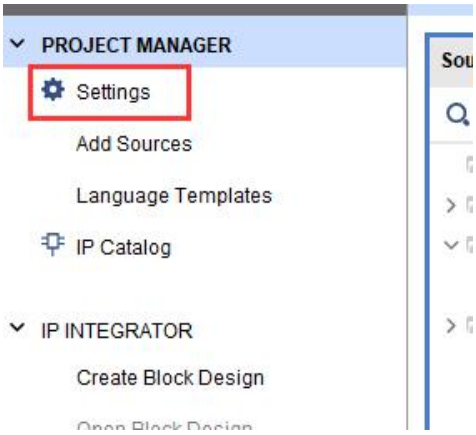
工程创建完成后，拷贝资料例程文件下的 IP 核文件夹到新建完成的工程目



录下相同位置，如下图所示：

图 22.2.2 拷贝 IP 核文件夹

之后向工程中添加自定义 IP 核，再左侧导航窗格点击 Settings 打开设置窗口，



如下图所示：

图 22.2.3 打开设置窗口

如下图所示点击 Repository，之后点击“+”添加自定义 IP 核，IP 核路径定位到之前复制的 IP 核文件夹，

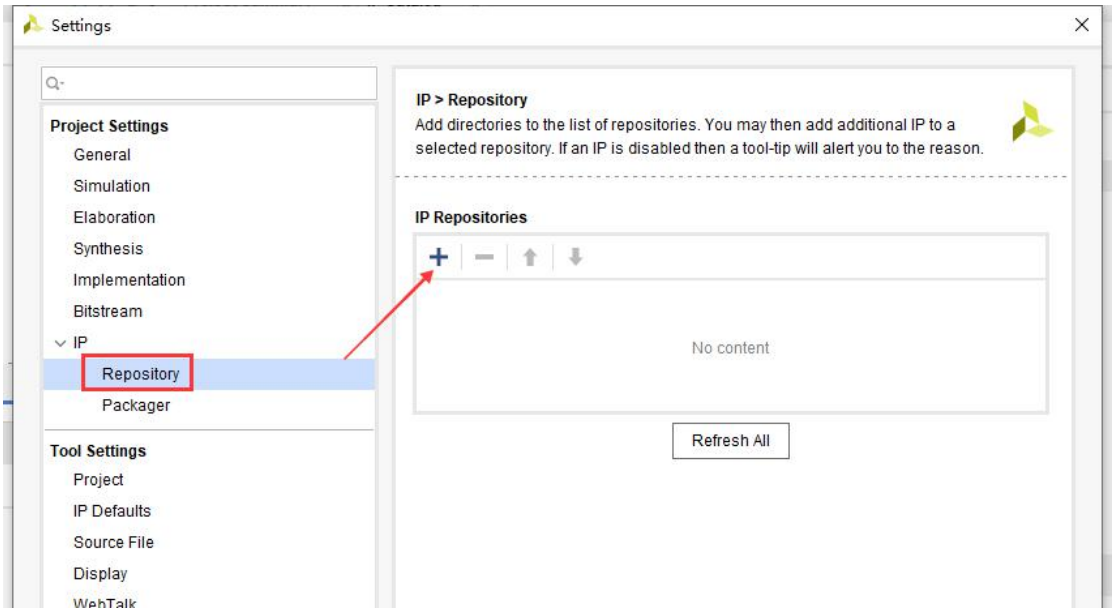
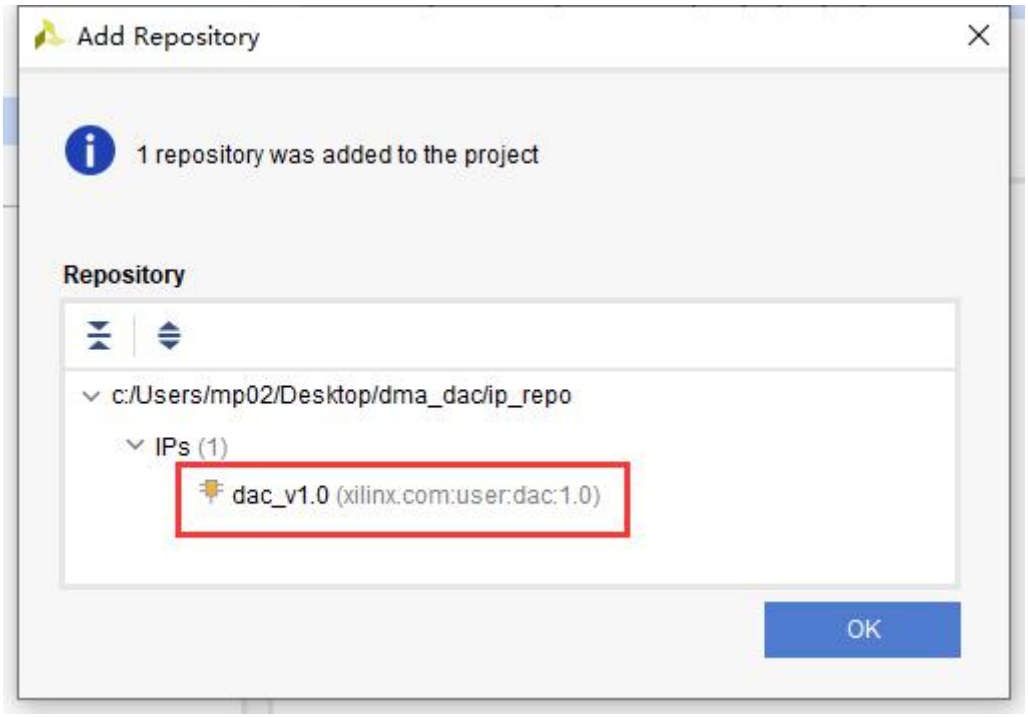


图 22.2.4 添加自定义 IP 核设置



之后软件会自动搜索识别文件夹下包含的 IP 核，如下图所示：

图 22.2.5 识别出的 IP 核

之后点击“OK”将该 IP 核添加到项目中，如下图所示：

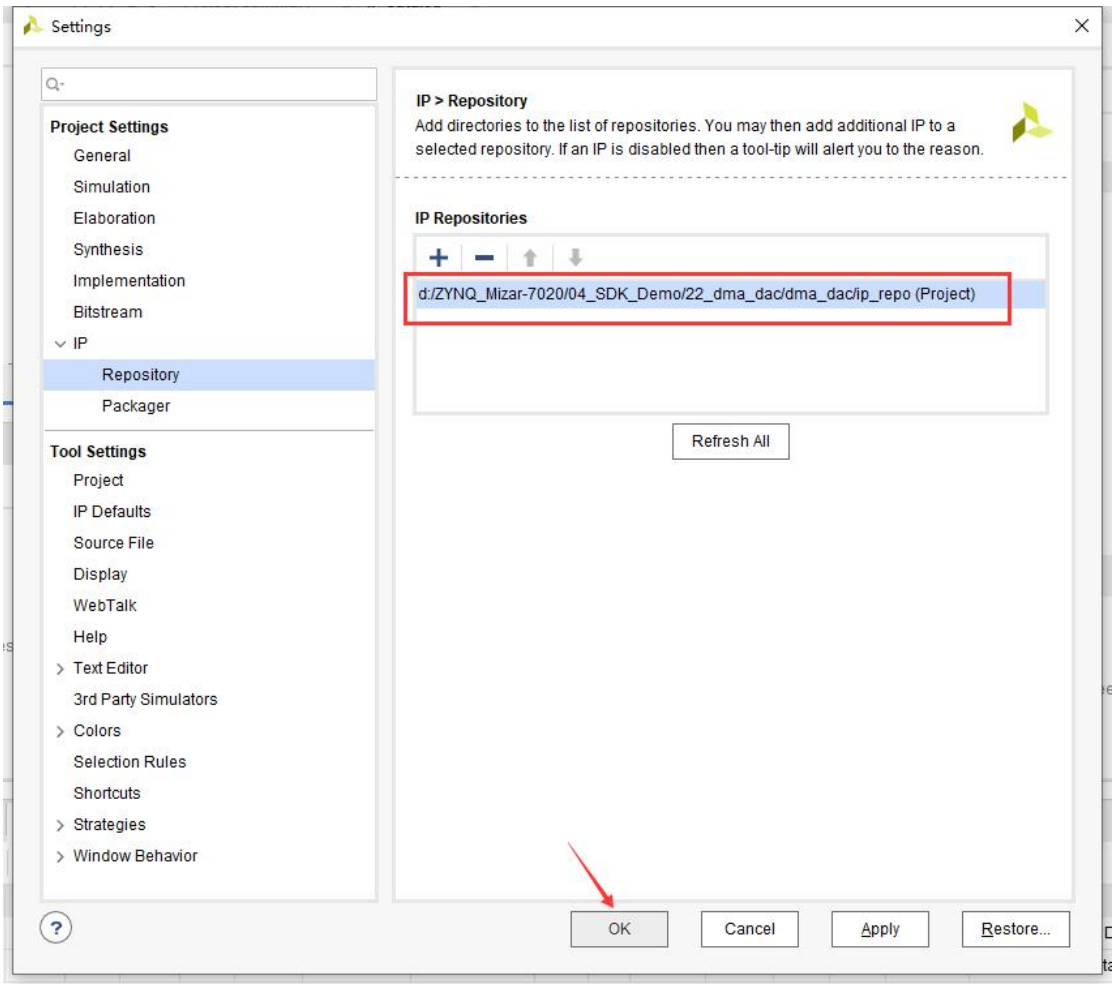


图 22.2.6 添加 IP 核

添加完成就可以开始进行硬件设计，点击 Create Block Design 创建设计文件，并将其命名为“dma_dac”，创建完成后添加 ZYNQ7 Processing System IP，添加完成后双击该 IP 打开配置页面。

点击 PS-PL Configuration，使能 S AXI HP0 interface，如下图所示：

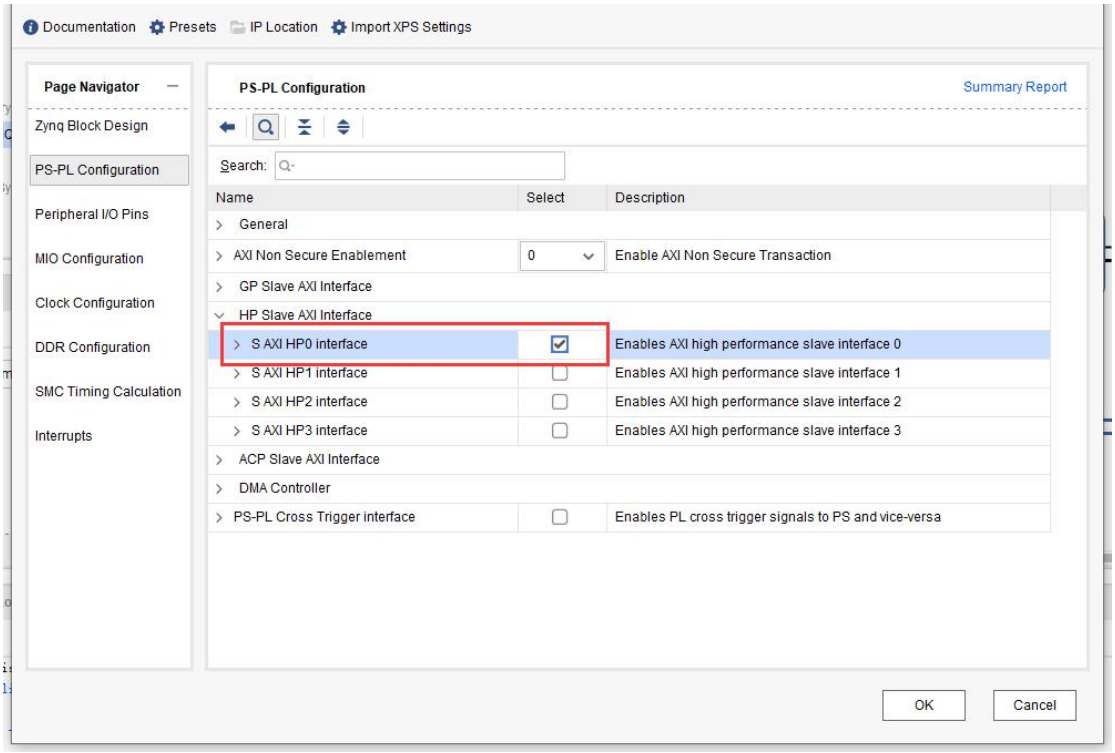


图 22.2.7 使能 S AXI HP0 interface

点击 Peripheral I/O Pins，使能串口 0，对应 MIO14，MIO15，使能 QSPI Flash，

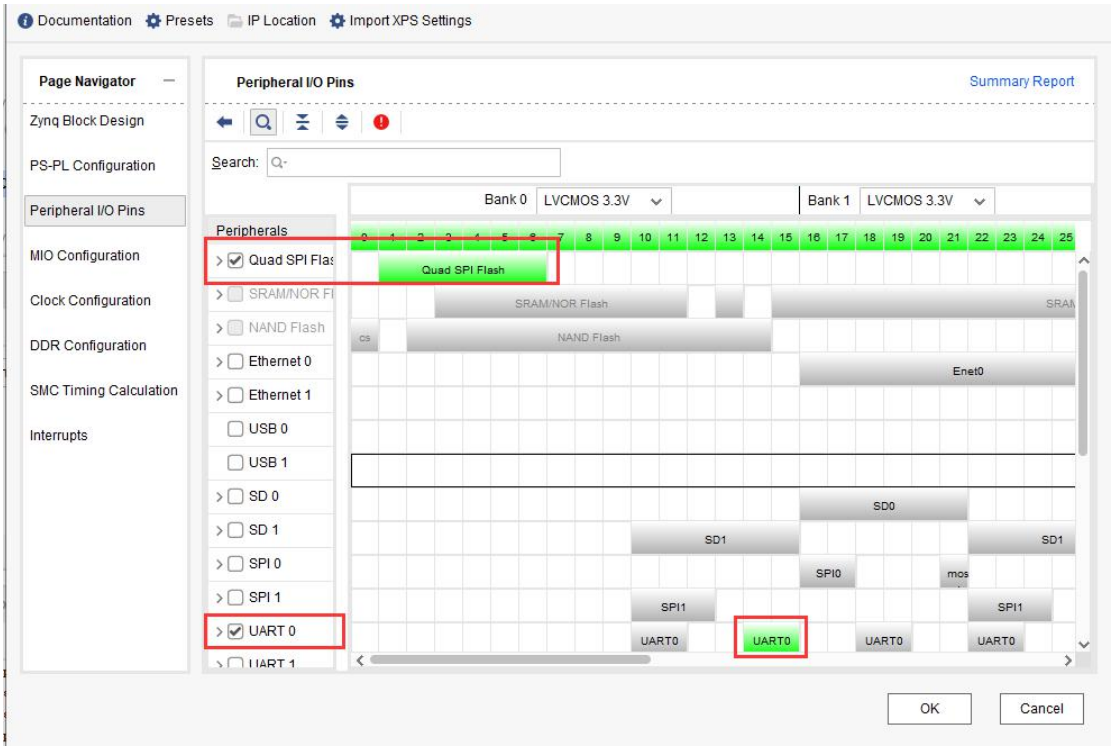
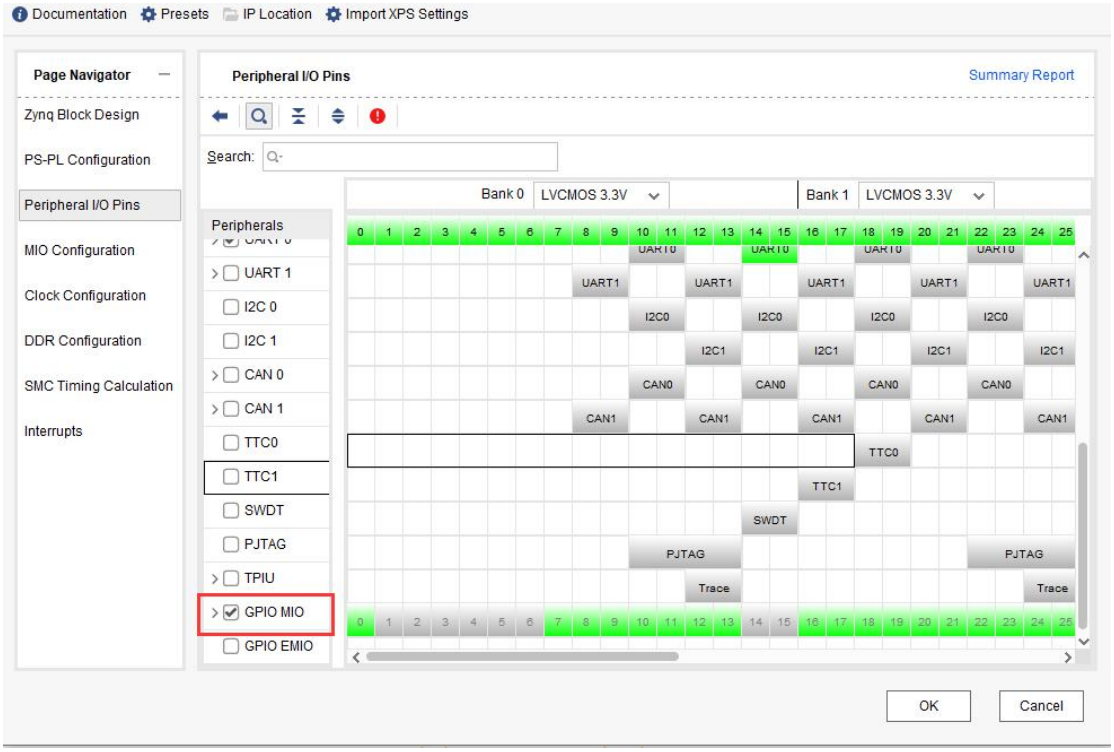
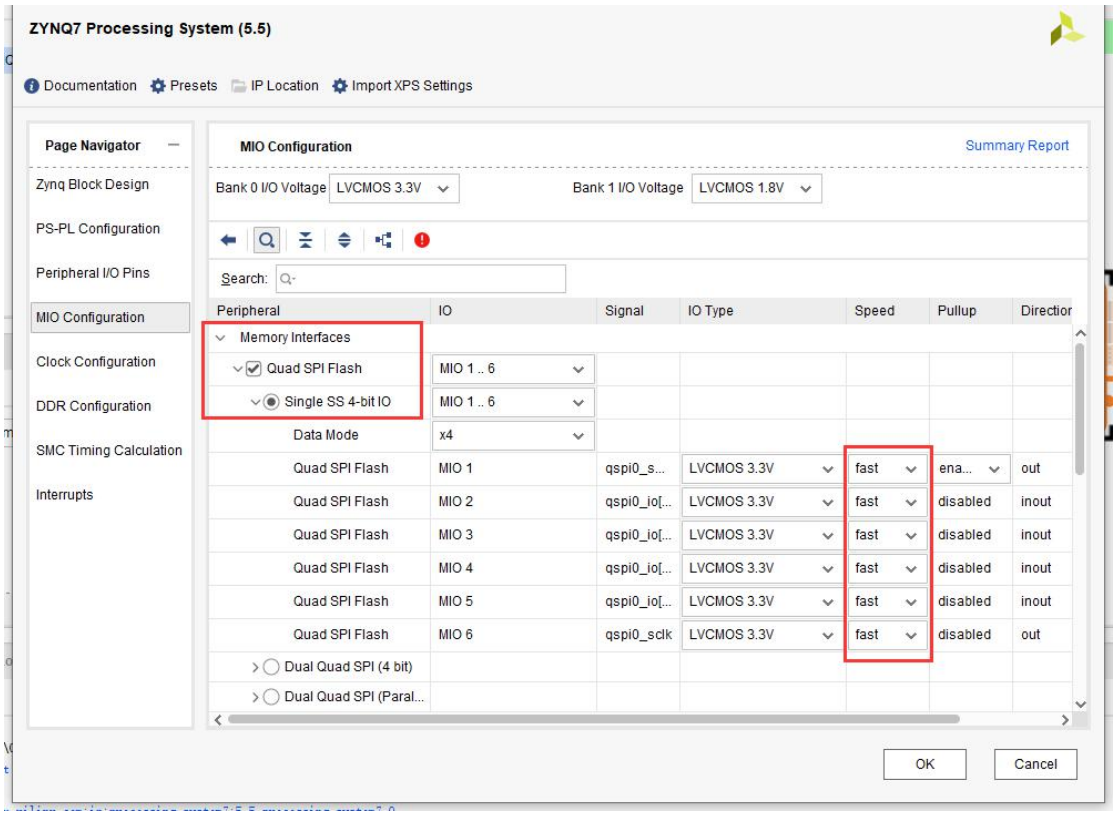


图 22.2.8 使能串口 0，使能 QSPI Flash



使能 MIO，

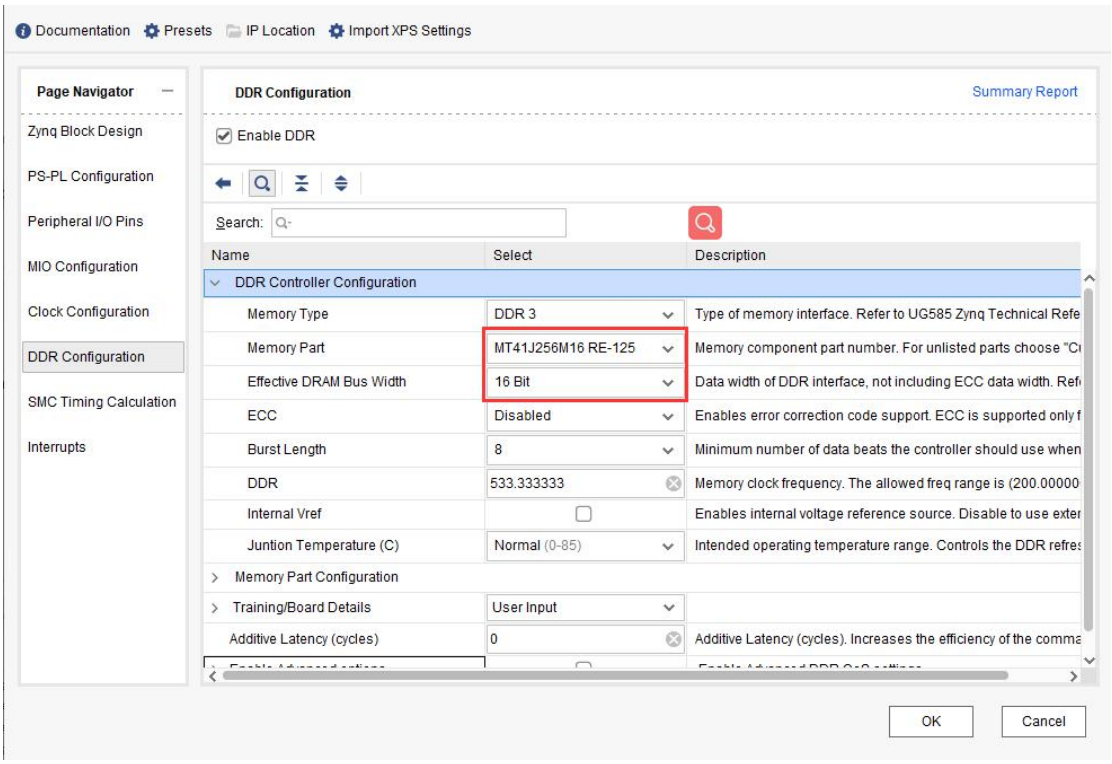
图 22.2.9 使能串口和 MIO



点击 MIO Configuration，修改所有 QSPI Flash 端口速率为 fast，

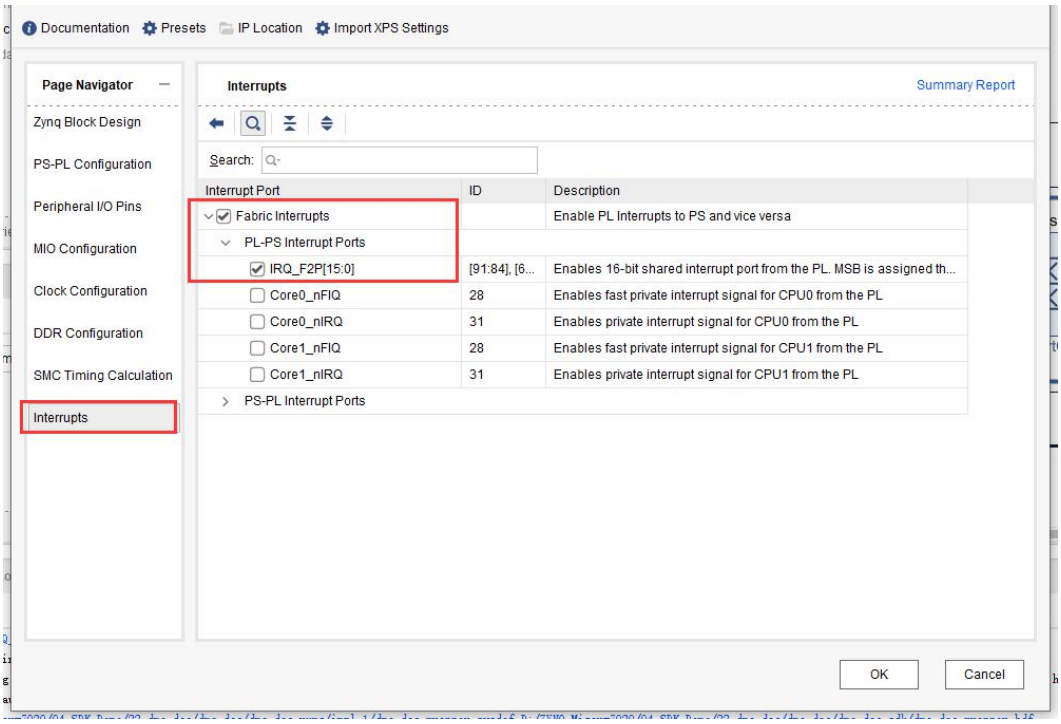
图 22.2.10 修改端口速率

点击 DDR Configuration，设置 DDR 控制器为 MT41J256M16 RE-125，不同型



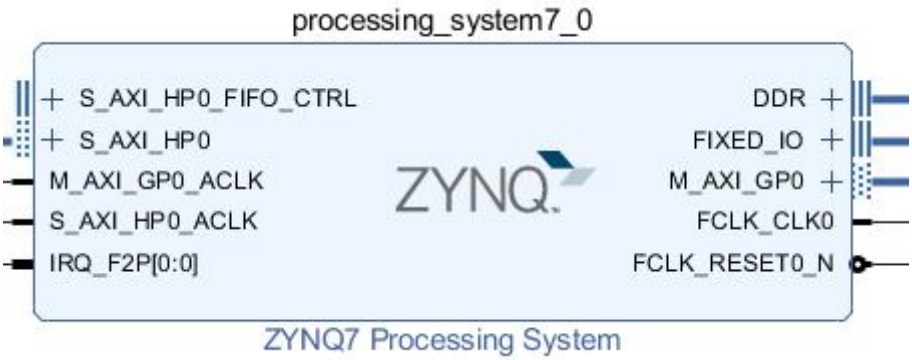
号开发板选择不同，此处请注意。

图 22.2.11 设置 DDR 控制器



点击 interrupts，使能 PL 到 PS 的中断，如下图所示：

图 22.2.12 使能中断



设置完成后 ZYNQ7 Processing System 如下图所示：

图 22.2.13 设置完成

点击窗口上方“+”打开 IP 核添加列表，搜索并添加 DMA IP，如下图所示：

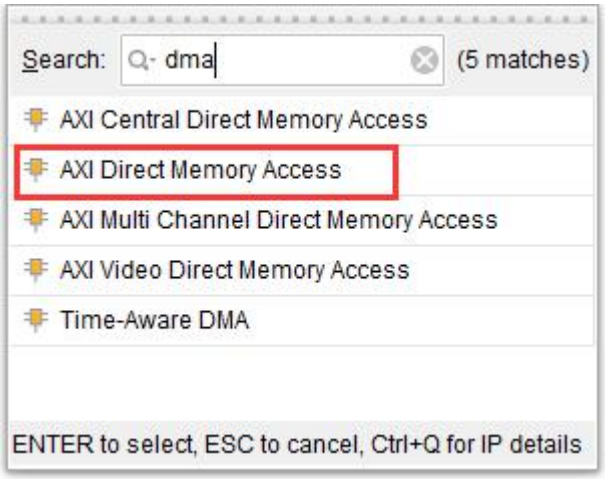


图 22.2.14 添加 DMA IP

双击该 IP 打开设置窗口，如下图所示配置各选项参数，只打开读通道，将 Stream Data Width 设置为 8，因为 DAC 数据宽度是 8 位。

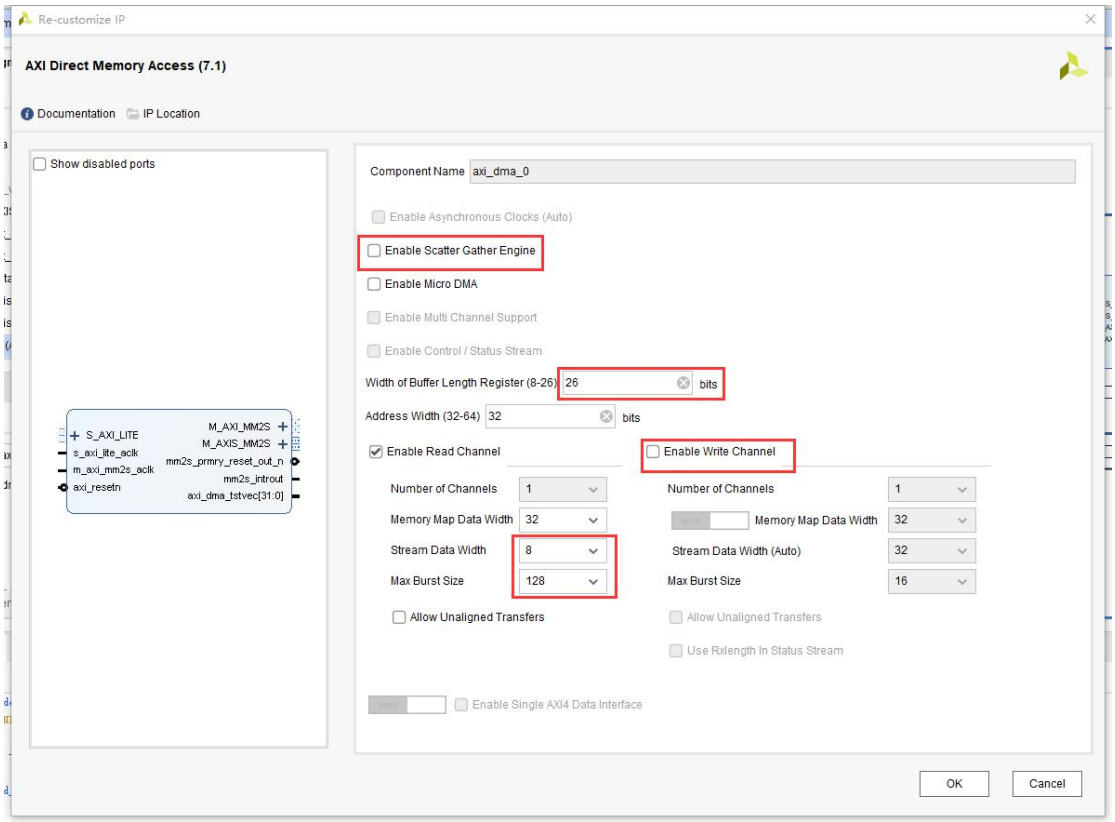


图 22.2.15 设置 DMA 参数

添加自定义 IP 核到设计中，如下图所示：

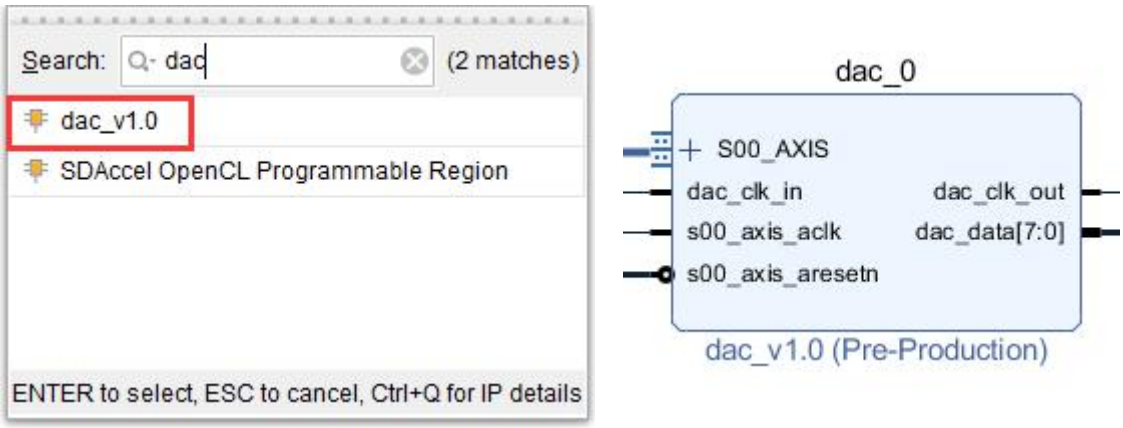
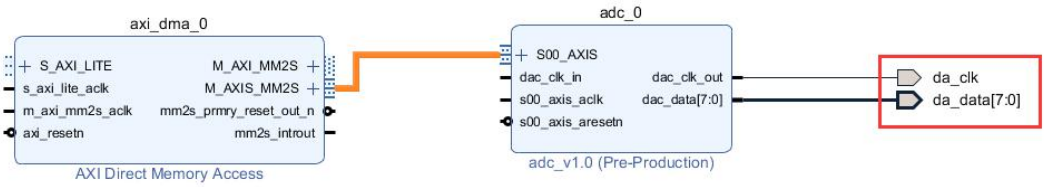


图 22.2.16 添加自定义 IP 到设计中

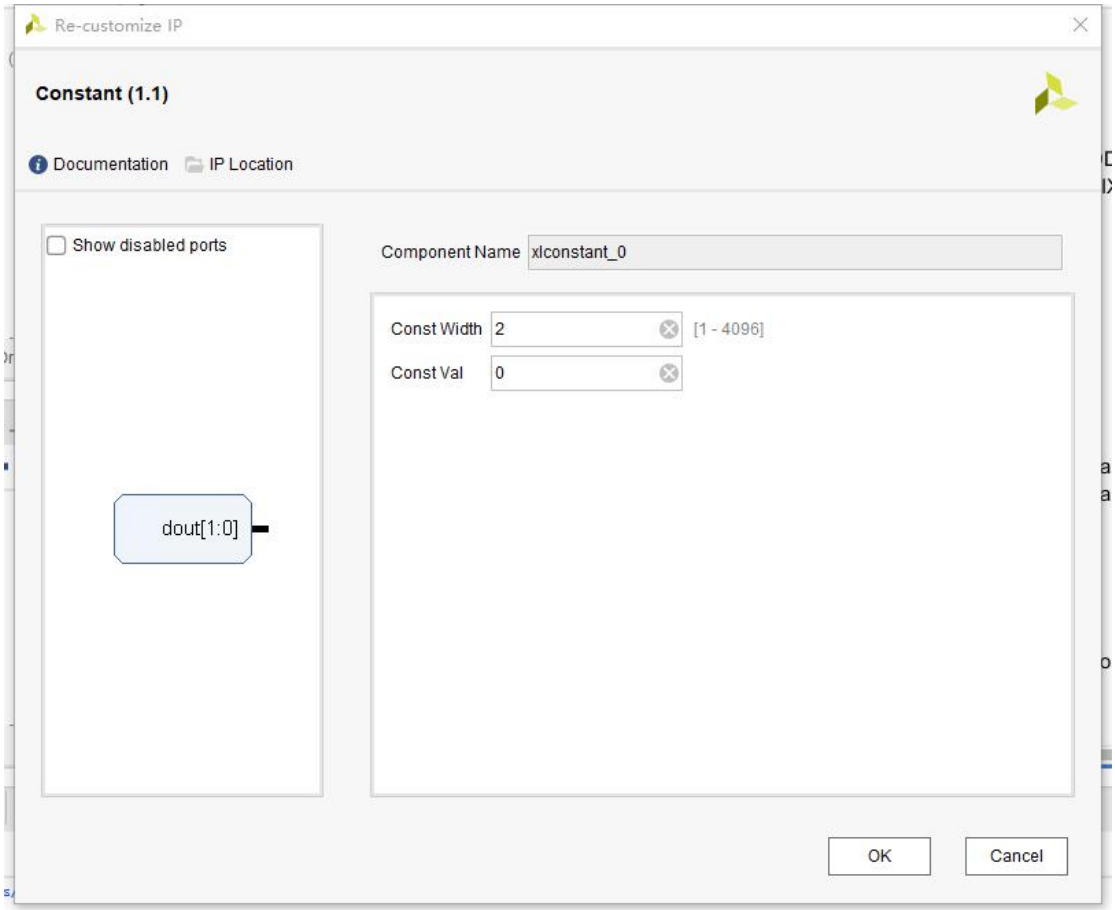
该 IP 无需设置，保持默认即可。之后如下图所示连接相关信号，并将 IP 核



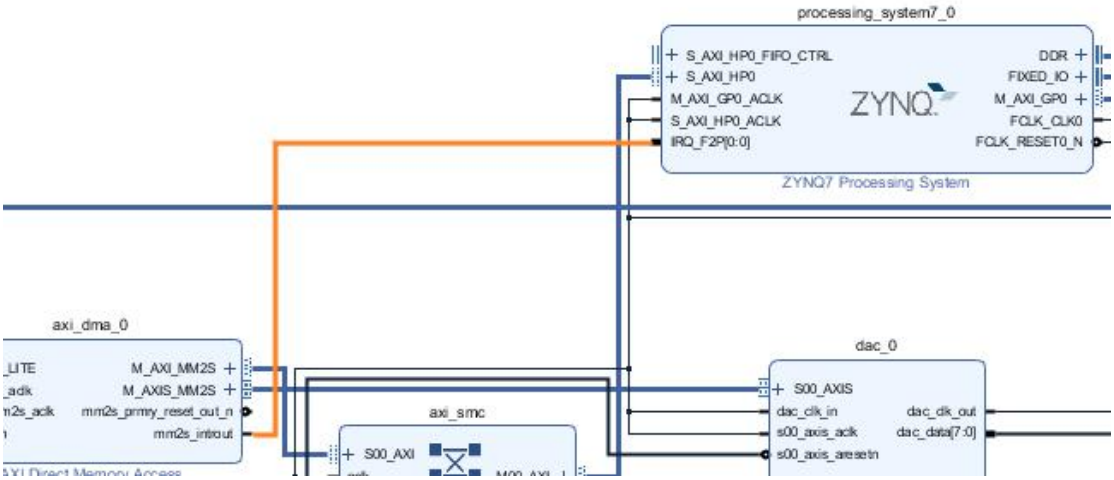
信号引出，修改信号端口名称，

图 22.2.17 连接并引出信号端口

添加一个 xlconstant IP 核模块，并对其进行设置如图 a 1 所示。位宽是 2，输出 Const Val 0 为低电平。如果不增加这一步，后续 ADA 将无法工作。

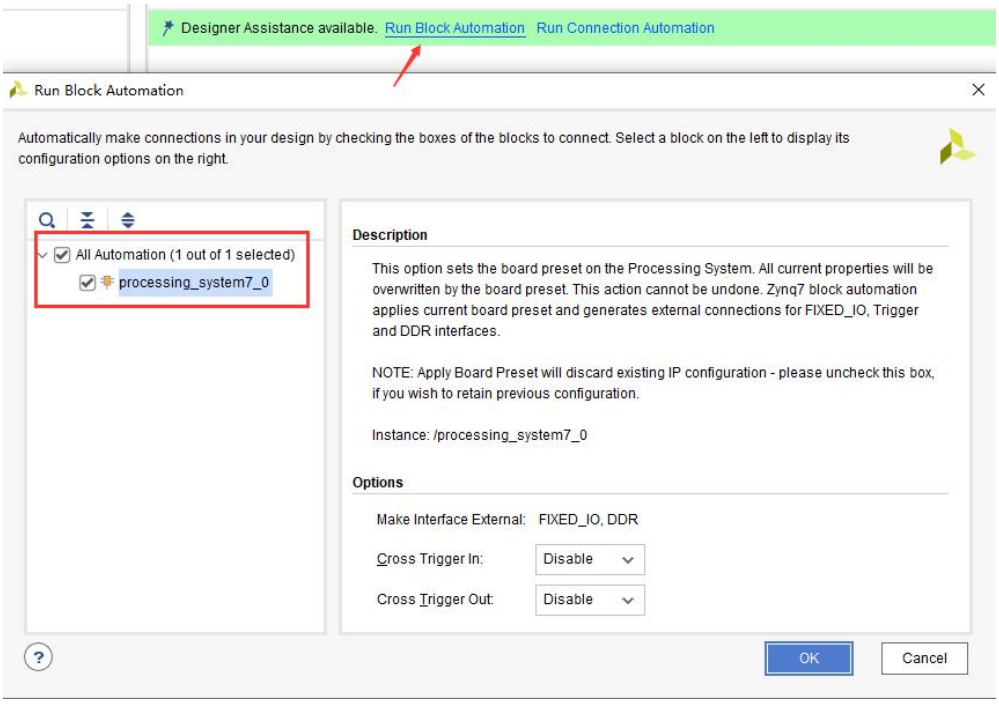


将 DMA 的 mm2s_introut 中断输出信号连接到 ZYNQ7 Processing Sysrem 的中



断信号输入端，如下图所示：

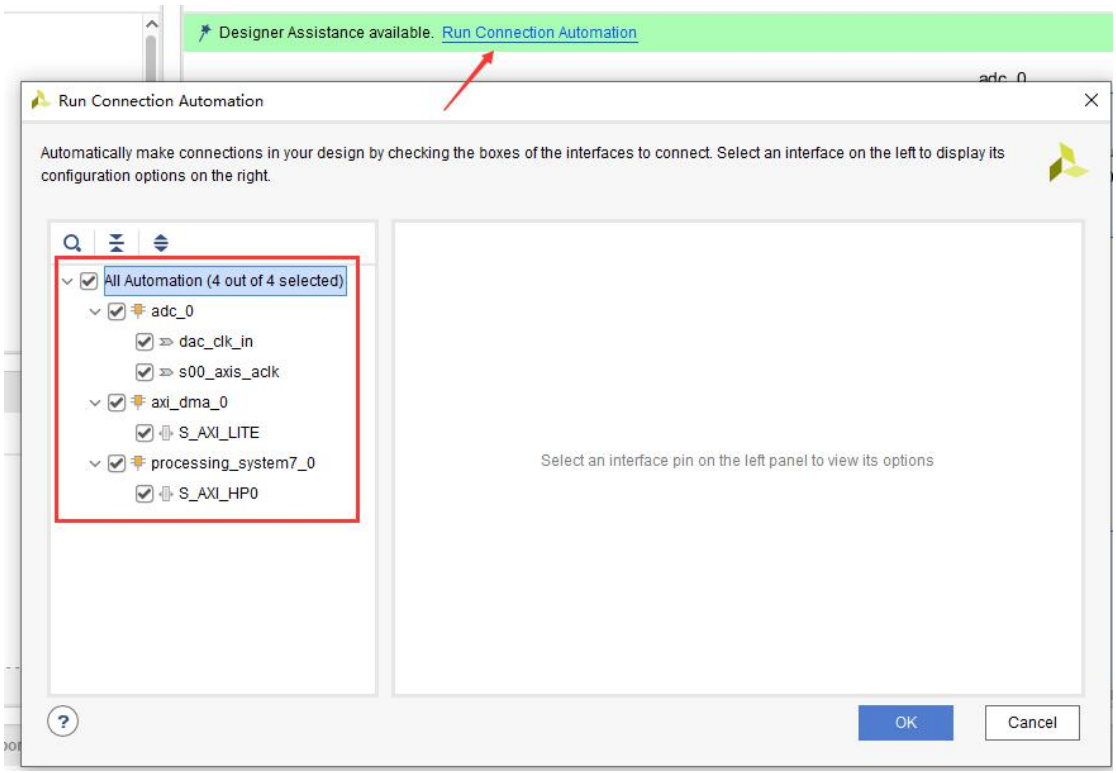
图 22.2.18 连接中断信号



点击 Run Block Automstion，选择 All Automation，如下图所示：

图 22.2.19 自动添加相关端口信号

点击 Run Connection Automation，选择 All Automation，如下图所示，自动



添加必要 IP 并连接相关信号，

图 22.2.20 自动添加必要 IP 并连接信号

创建完成的整个硬件设计如下图所示：

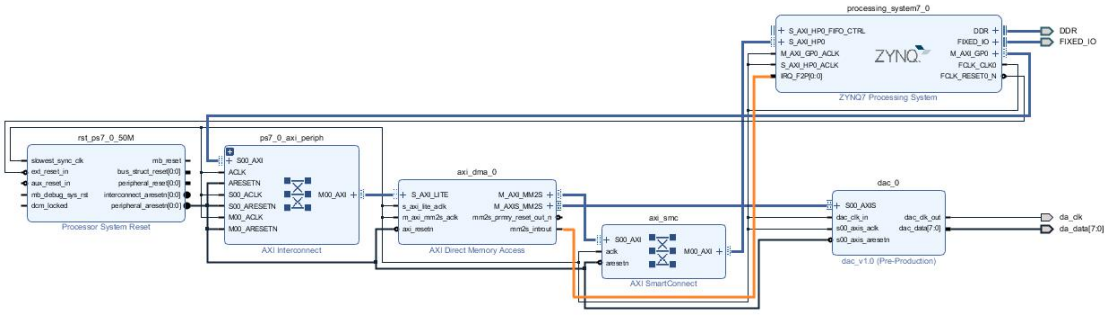


图 22.2.21 硬件设计图

之后点击导航窗格下的 Open Elaborated Dseign 打开电气设计串口窗口配置引脚约束，在弹出的窗口选择 OK，

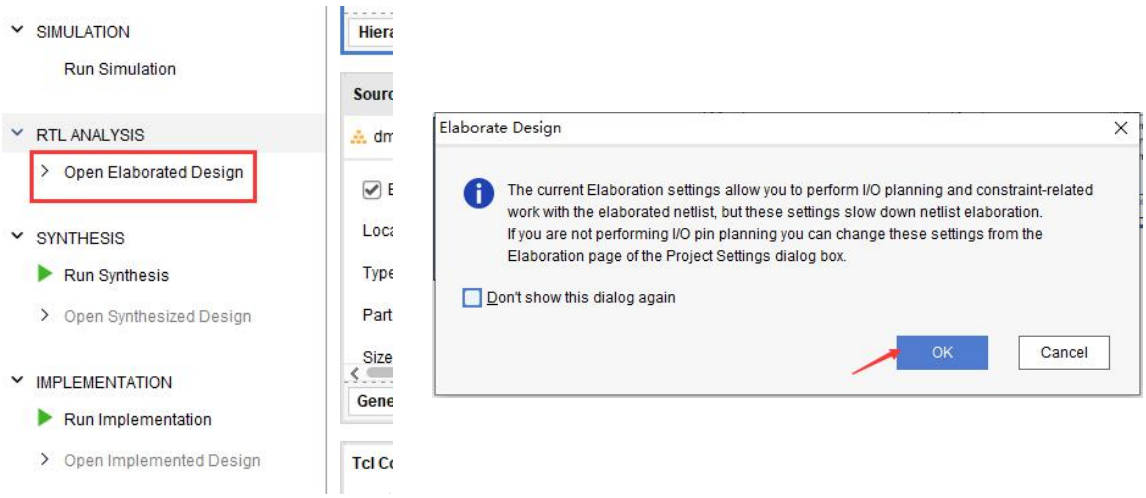


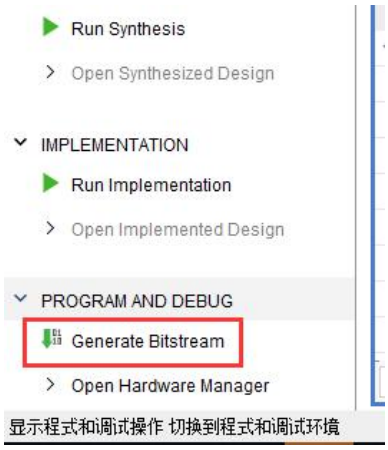
图 22.2.22 打开引脚约束配置

参考原理图配置引脚约束，本例使用 GPIO1 端口，引脚约束如下图所示：

```
set_property IOSTANDARD LVCMOS33 [get_ports {da_data[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_data[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_data[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_data[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_data[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_data[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_data[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_data[0]}]
set_property PACKAGE_PIN W19 [get_ports {da_data[0]}]
set_property PACKAGE_PIN W18 [get_ports {da_data[1]}]
set_property PACKAGE_PIN U17 [get_ports {da_data[2]}]
set_property PACKAGE_PIN T16 [get_ports {da_data[3]}]
set_property PACKAGE_PIN R17 [get_ports {da_data[4]}]
set_property PACKAGE_PIN R16 [get_ports {da_data[5]}]
set_property PACKAGE_PIN P18 [get_ports {da_data[6]}]
set_property PACKAGE_PIN N17 [get_ports {da_data[7]}]
set_property PACKAGE_PIN Y18 [get_ports da_clk]

set_property SLEW FAST [get_ports {da_data[*]}]
set_property SLEW FAST [get_ports da_clk]

set_property IOSTANDARD LVCMOS33 [get_ports {da_data[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports da_clk]

set_property IOSTANDARD LVCMOS33 [get_ports {dout[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[1]}]
set_property PACKAGE_PIN Y19 [get_ports {dout[0]}]
set_property PACKAGE_PIN T12 [get_ports {dout[1]}]
```

图 22.2.23 引脚约束

设置完成之后即可综合、编译生成 BIT 文件，点击左侧导航窗格



Generate Bitstream，在弹出的对话框选择 yes，如下图所示：

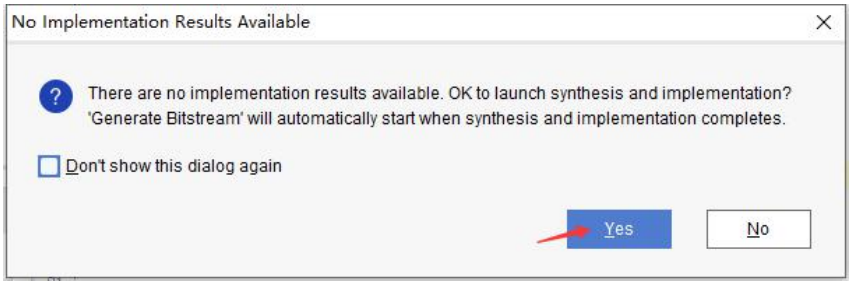


图 22.2.24 生成 BIT 文件

BIT 文件生成之后点击 File->Export->Export Hardware，在弹出的对话框中勾选 include bitstream，导出 BIT 文件。

点击 File->Launch SDK 打开 SDK 开发软件。

22.3 软件设计

在打开的 SDK 软件中，创建新的应用工程并命名为“dma_dac”，工程模板选择空白。在本次实验中，我们需要用到一些数学计算的相关函数，需要向工程中添加相关文件。在左侧项目管理器窗口右键点击工程，在弹出的菜单中点击 C/C++ Build Settings，如下图所示：

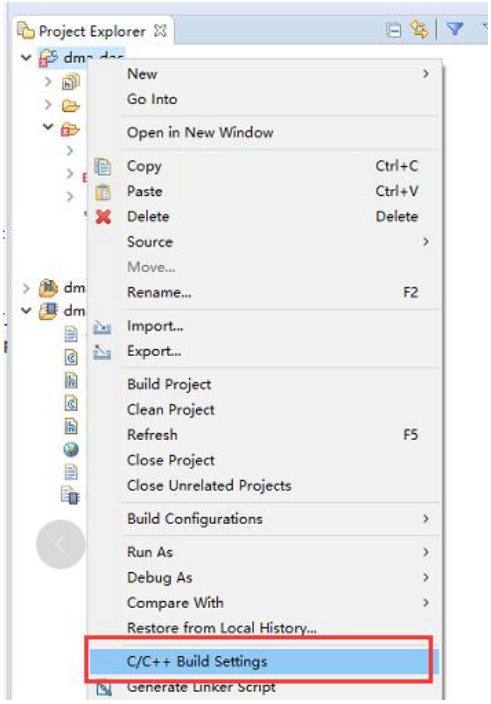


图 22.3.1 项目设置菜单

在打开的窗口中点击 Tool Settings 下的 Libraries，点击箭头所指的“+”，

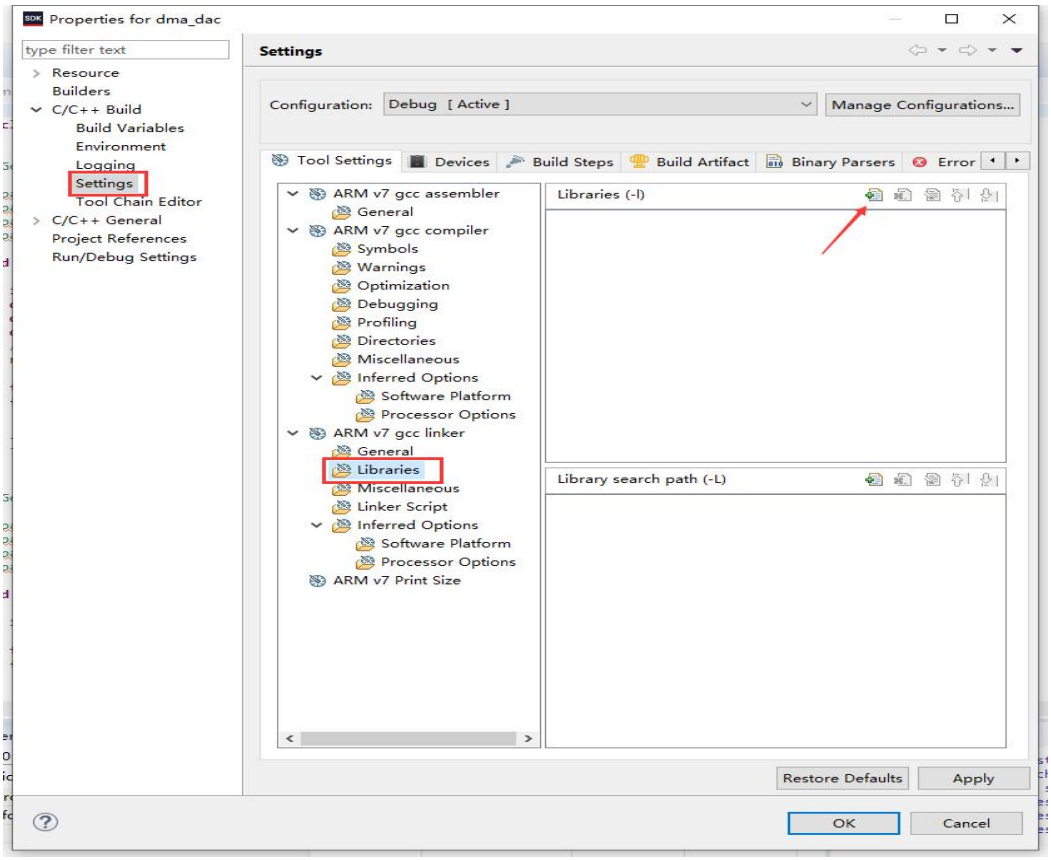
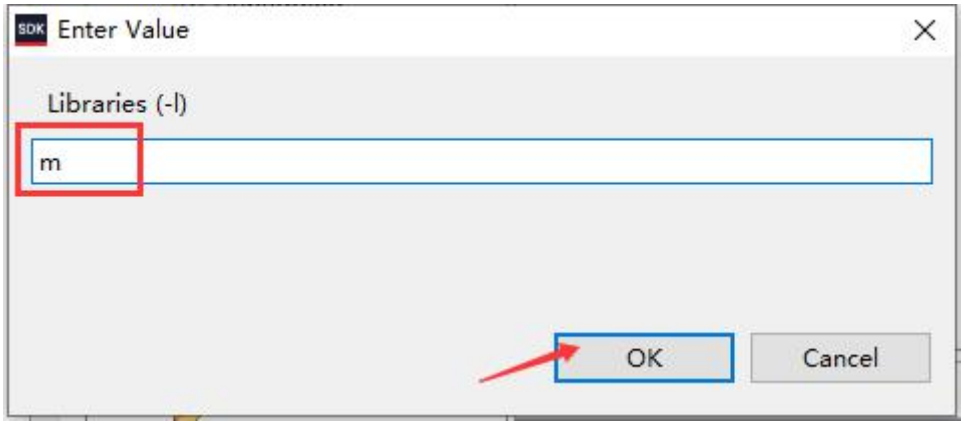


图 22.3.2 选择 Libraries 项

在弹出的窗口中输入“m”，点击 OK，软件会自动搜索添加数学运算相关文



件，如下图所示：

图 22.3.3 添加数学运算相关文件。

复制资料例程下的源码文件到新建工程的相同位置，其中 main.c 是程序运行的相关函数，wave.c 和 wave.h 提供了正弦波、方波、三角波、锯齿波、负锯齿波这五种波形函数。

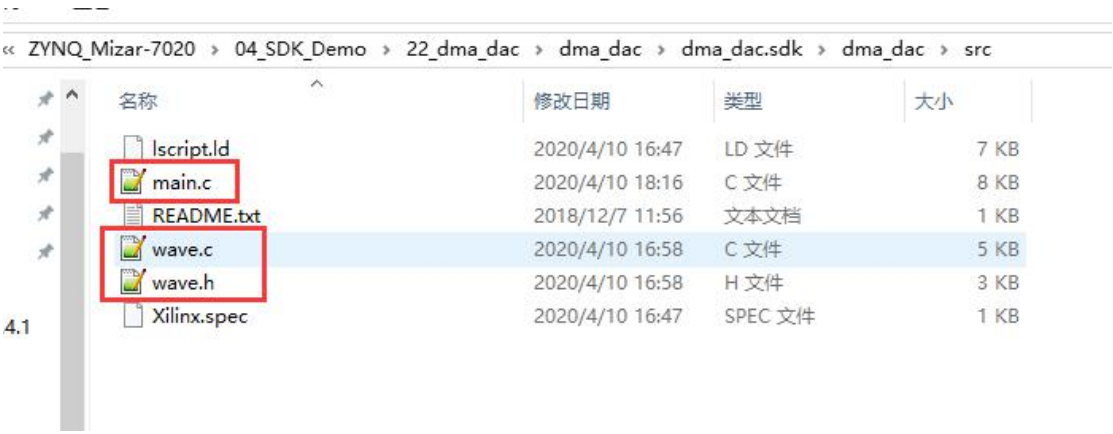


图 22.3.4 程序代码源文件

在软件设计中，为实现实验目标，我们需要配置 dma 相关设置，使其读取 ddr 的数据并将其传输给自定义 IP，配置中断相关设置，使其可以读取按键中断并修改波形数据，实现按键切换波形的功能。根据这些目标来编写相关函数。

接下来讲解一下 main.c 文件下各个函数：

int XAxiDma_DAC(u16 DeviceId)：初始化 DMA，获取正弦波数据写入到 ddr

中，开启传输，使正弦波作为默认输出波形。

```
int XAxiDma_DAC(u16 DeviceId)
{

    /* Initialize DMA and enable MM2S interrupt */
    XAxiDma_Initial(DeviceId, MM2S_INTR_ID, &AxiDma, &INST) ;
    /* Create initial wave data */
    GetSinWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ;
    /* Copy wave data to DMA buffer */
    memcpy(DmaTxBuffer, WaveBuffer, MAX_PKT_LEN) ;
    Xil_DCacheFlushRange((UINTPTR)DmaTxBuffer, MAX_PKT_LEN);

    XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) DmaTxBuffer,
            MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);


    return XST_SUCCESS;
}
```

int KeySetup(XScuGic *InstancePtr, u16 IntrID, XGpioPs *GpioInstancePtr)：初始化 GPIO 控制器，设置 MIO0 即 PS_KEY1 为输入，设置 GPIO 中断为下降沿，设置

```
int KeySetup(XScuGic *InstancePtr, u16 IntrID, XGpioPs *GpioInstancePtr)
{
    XGpioPs_Config *GPIO_CONFIG ;
    int Status ;
    key_flag = 0 ;


    GPIO_CONFIG = XGpioPs_LookupConfig(MIO_0_ID) ;
    Status = XGpioPs_CfgInitialize(GpioInstancePtr, GPIO_CONFIG, GPIO_CONFIG->BaseAddr) ;
    if (Status != XST_SUCCESS)
    {
        return XST_FAILURE ;
    }
    //set MIO 0 as input
    XGpioPs_SetDirectionPin(GpioInstancePtr, 0, GPIO_INPUT) ;
    //set interrupt type
    XGpioPs_SetIntrTypePin(GpioInstancePtr, 0, XGPIOPS_IRQ_TYPE_EDGE_FALLING) ;


    //set priority and trigger type
    XScuGic_SetPriorityTriggerType(InstancePtr, IntrID,
            0xA0, 0x3);
    Status = XScuGic_Connect(InstancePtr, IntrID,
            (Xil_ExceptionHandler)GpioHandler,
            (void *)GpioInstancePtr) ;

    XScuGic_Enable(InstancePtr, IntrID) ;

    if (Status != XST_SUCCESS) {
        return Status;
    }

    XGpioPs_IntrEnablePin(GpioInstancePtr, 0) ;

    return XST_SUCCESS ;
}
```

中断类型和优先级，使能中断。

int XAxiDma_Initial(u16 DeviceId, u16 IntrID, XAxiDma *XAxiDma, XScuGic *InstancePtr)：初始化 DMA，使能 MM2S 中断；

```
int XAxiDma_Initial(u16 DeviceId, u16 IntrID, XAxiDma *XAxiDma, XScuGic *InstancePtr)
{
    XAxiDma_Config *CfgPtr;
    int Status;
    /* Initialize the XAxiDma device. */
    CfgPtr = XAxiDma_LookupConfig(DeviceId);
    if (!CfgPtr) {
        xil_printf("No config found for %d\r\n", DeviceId);
        return XST_FAILURE;
    }

    Status = XAxiDma_CfgInitialize(XAxiDma, CfgPtr);
    if (Status != XST_SUCCESS) {
        xil_printf("Initialization failed %d\r\n", Status);
        return XST_FAILURE;
    }
```

```
    Status = XScuGic_Connect(InstancePtr, IntrID,
                (Xil_ExceptionHandler)DAC_Interrupt_Handler,
                (void *)XAxiDma) ;

        if (Status != XST_SUCCESS) {
            return Status;
        }

        XScuGic_Enable(InstancePtr, IntrID);

        /* Disable S2MM interrupt, Enable MM2S interrupt */
        XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
                XAXIDMA_DEVICE_TO_DMA);
        XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_IOC_MASK,
                XAXIDMA_DMA_TO_DEVICE);
    return XST_SUCCESS ;
}
```

int SetInterruptInit(XScuGic *InstancePtr, u16 IntID)：函数中断设置，打开中断

```
int SetInterruptInit(XScuGic *InstancePtr, u16 IntID)
{

    XScuGic_Config * Config ;
    int Status ;

    Config = XScuGic_LookupConfig(IntID) ;

    Status = XScuGic_CfgInitialize(&INST, Config, Config->CpuBaseAddress) ;
    if (Status != XST_SUCCESS)
        return XST_FAILURE ;

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler) XScuGic_InterruptHandler,
            InstancePtr);

    Xil_ExceptionEnable();

    return XST_SUCCESS ;
}
```

控制器。

void GpioHandler(void *CallbackRef)：按键中断服务函数，监测中断状态。

```
    while(1)
        {
            if (key_flag)
            {
                switch(wave)
                {
                case 0 : GetSquareWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
                case 1 : GetTriangleWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
                case 2 : GetSawtoothWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
                case 3 : GetSubSawtoothWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
                case 4 : GetSinWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
                default: GetSinWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
                }

                memcpy(DmaTxBuffer, WaveBuffer, MAX_PKT_LEN) ;
                Xil_DCacheFlushRange((UINTPTR)DmaTxBuffer, MAX_PKT_LEN);

                if (wave == 4)
                    wave = 0 ;
                else
                    wave++ ;

                /* Clear flag */
                key_flag = 0 ;

            }
        }
    return XST_SUCCESS;

}
```

while(1)中一但获取到中断，则修改波形数据，改变输出波形。

DMA 中断服务函数，一次传输完成之后检测到中断开启下一次传输；

```
void DAC_Interrupt_Handler(void *CallBackRef)
{
    XAxiDma *XAxiDmaPtr = (XAxiDma *) CallBackRef ;
    int mm2s_sr ;

    mm2s_sr = XAxiDma_IntrGetIrq(XAxiDmaPtr, XAXIDMA_DMA_TO_DEVICE) ;

    if (mm2s_sr & XAXIDMA_IRQ_IOC_MASK)
    {
        /* Clear interrupt */
        XAxiDma_IntrAckIrq(XAxiDmaPtr, XAXIDMA_IRQ_IOC_MASK,
                XAXIDMA_DMA_TO_DEVICE) ;

        XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) DmaTxBuffer,
                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
    }
}
```

代码源文件复制到工程中后，重新编译工程，没有错误后即可烧录验证。

22.4 下载验证

首先将 ADA-106 子卡插入 GPIO1 插座中，注意插接的方向。然后使用 type-c 线线将开发板的 UART 接口与电脑连接，用于串口通信。然后我们将使用 type-c 线与 JTAG 接口连接，另外一端与电脑连接。ADA-106 子卡连接如下图所示：

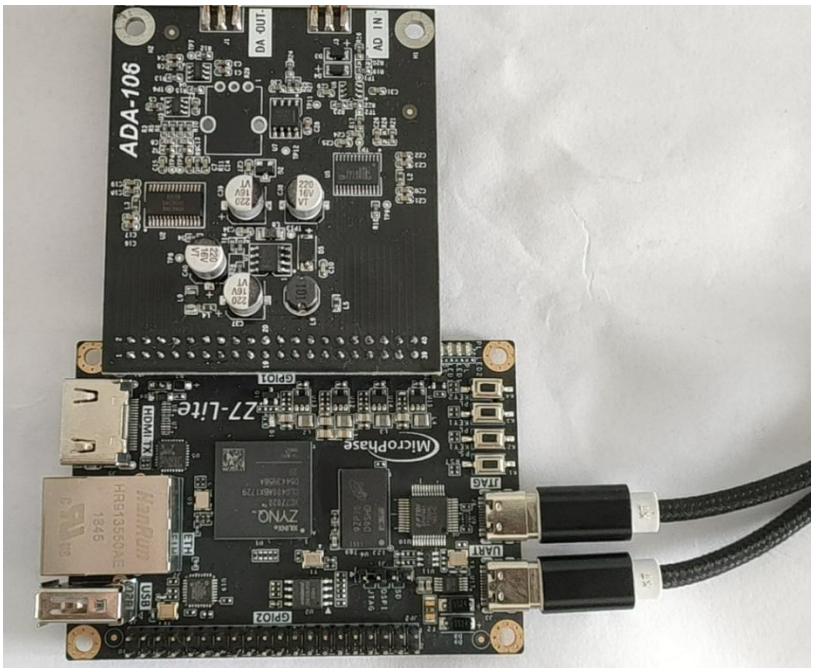
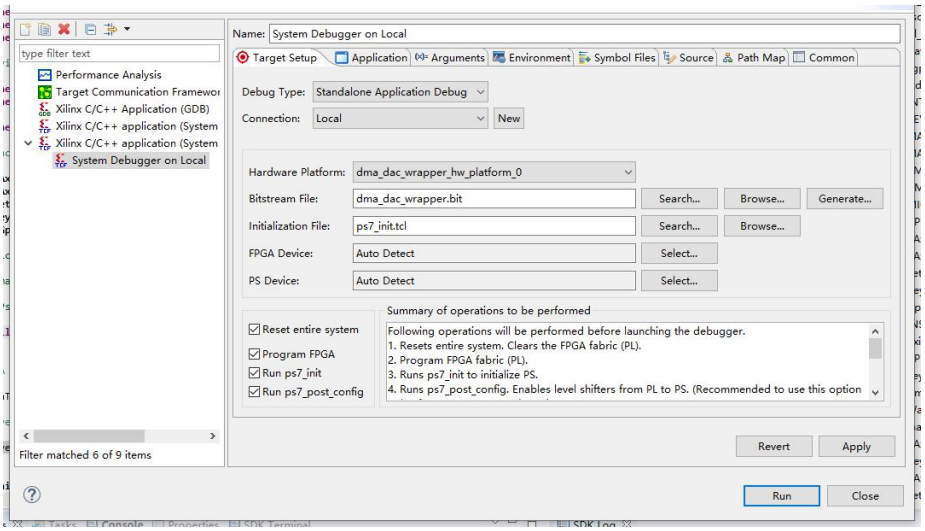
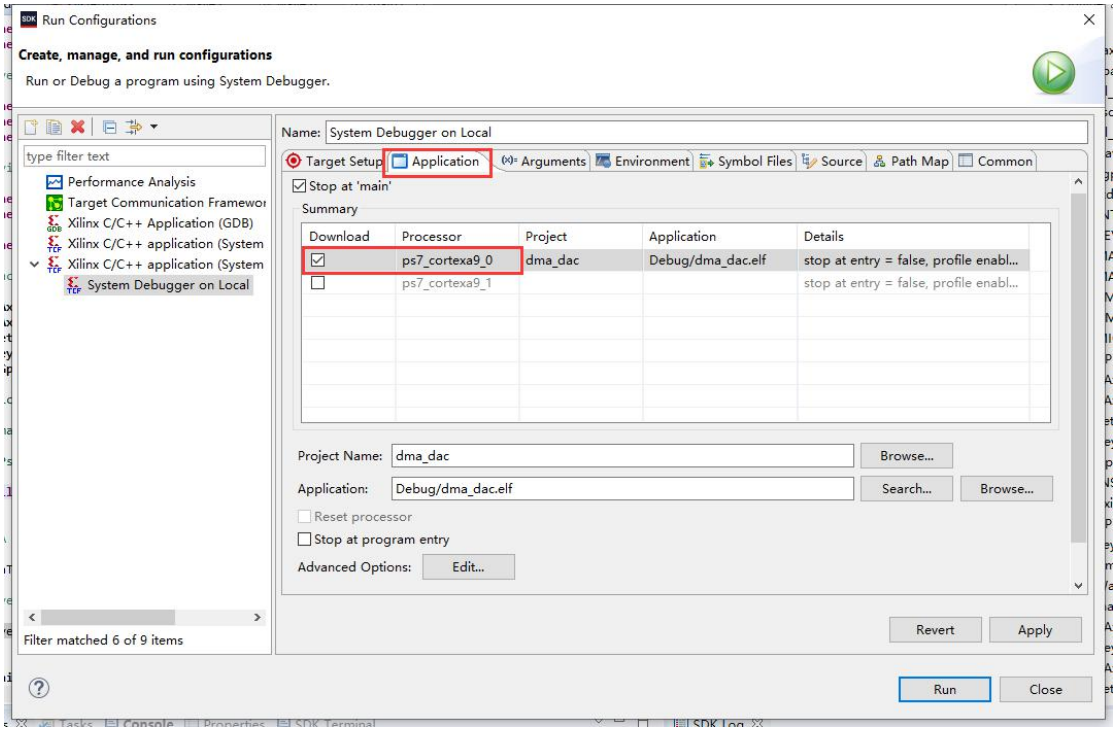


图 22.4.1 连接子卡及其他连接线



点击 Run->Run Configurations，如下图所示设置 bit 文件烧录选项：

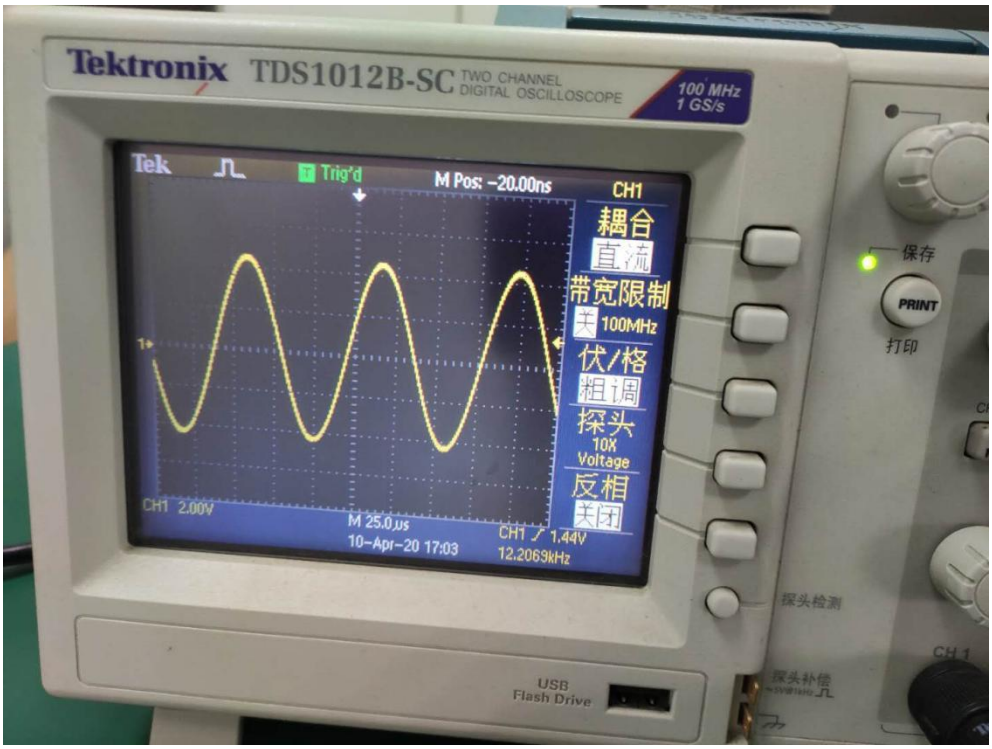
图 22.4.2 设置烧录选项



之后设置应用工程烧录选项，如下图所示：

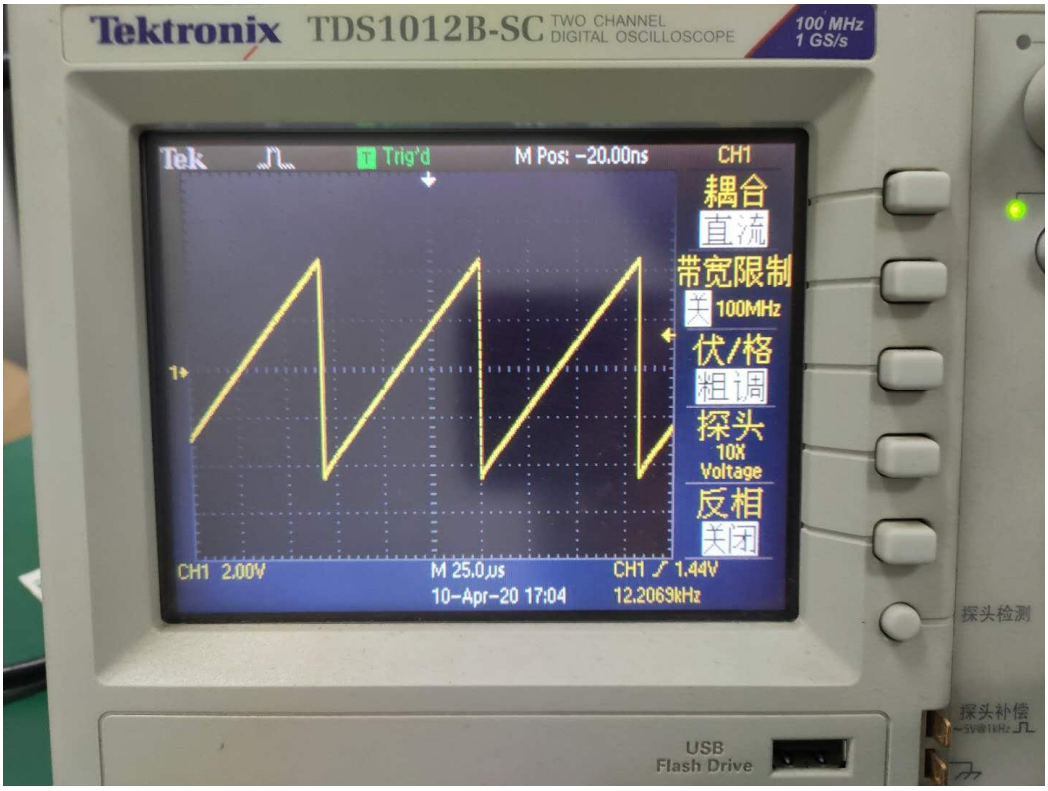
图 22.4.3 设置应用程序烧录选项

设置完成后点击 Run 将程序烧录到开发板中，利用示波器即可看到输出的波

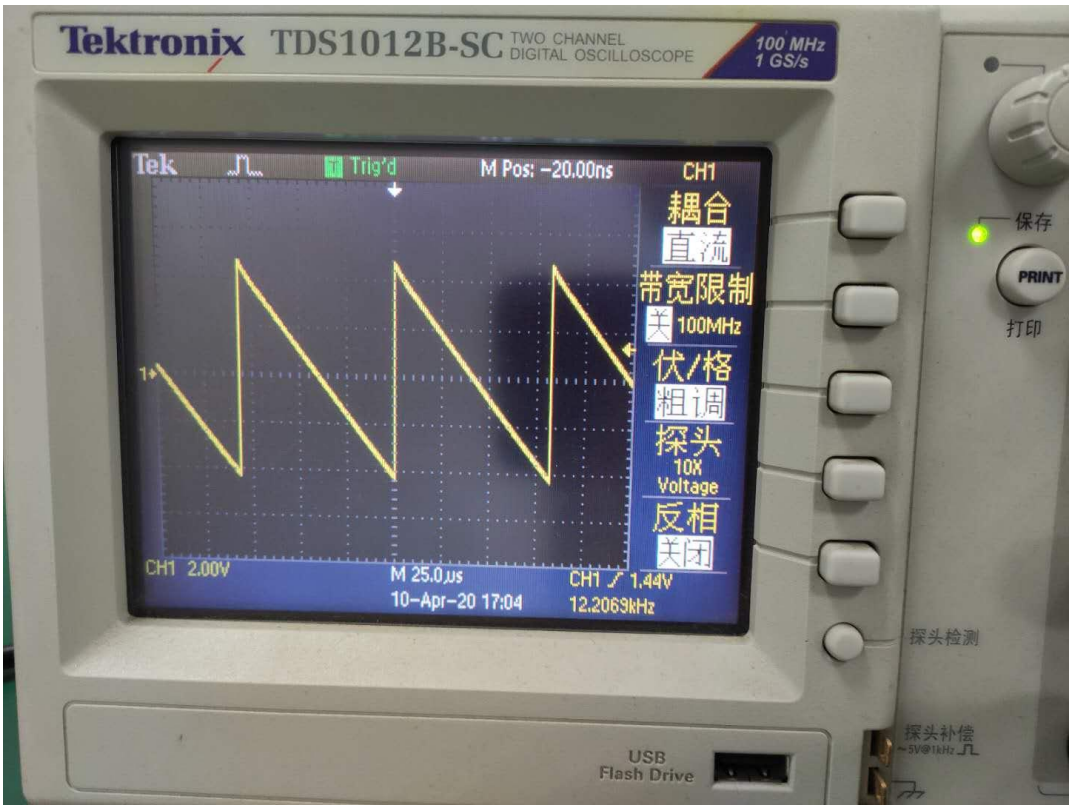


形如下图所示，可以通过 PS_KEY1 按键切换其他波形。

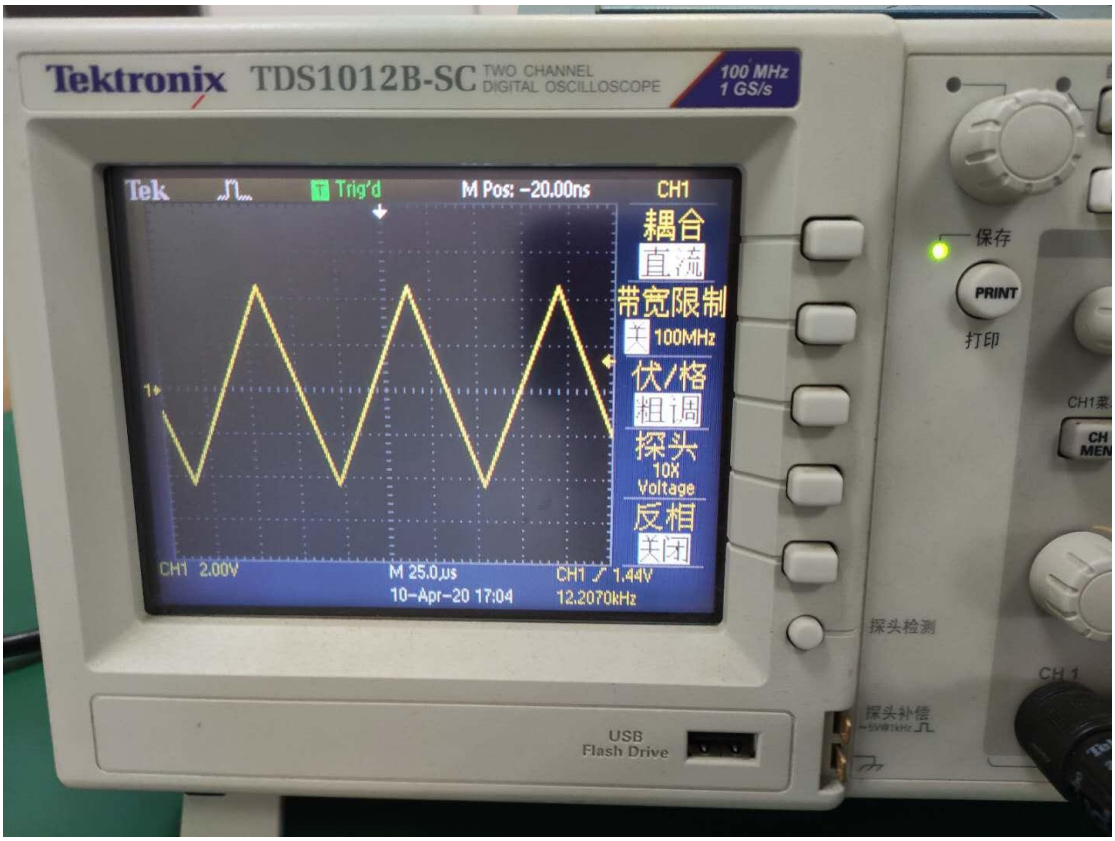
正弦波

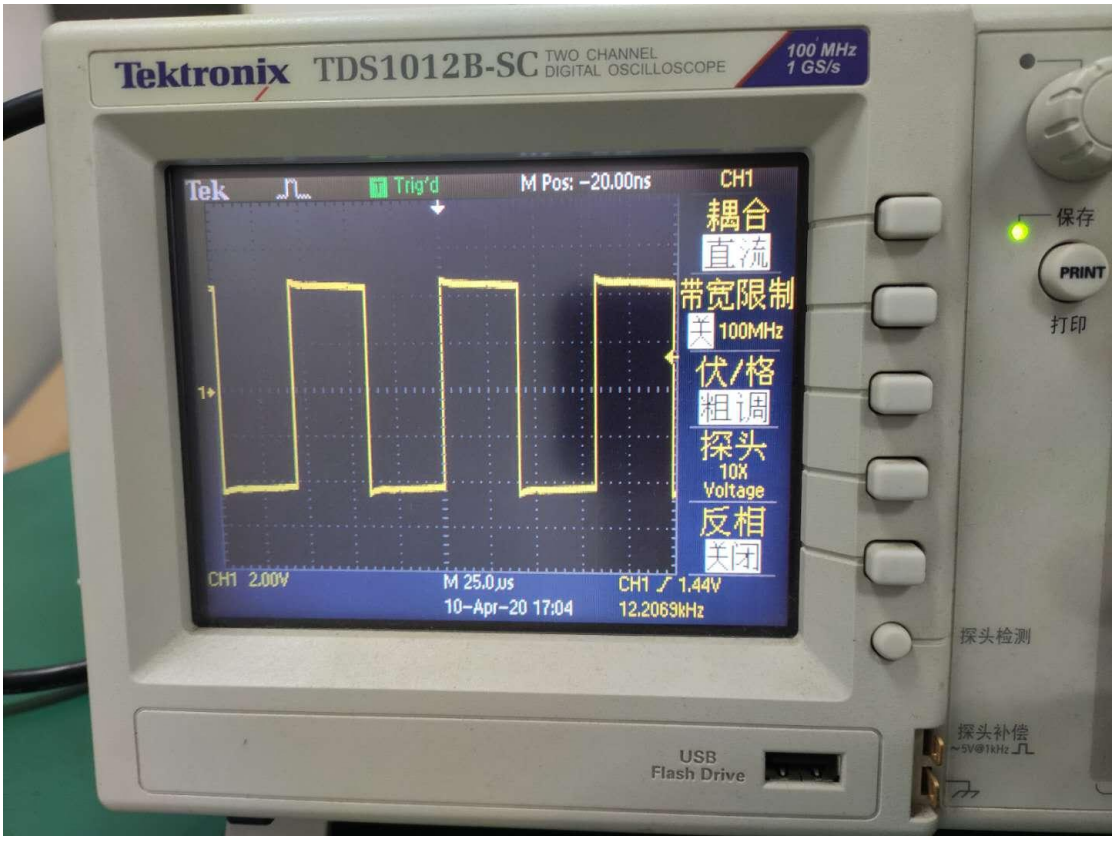

Tektronix TDS1012B-SC TWO CHANNEL DIGITAL OSCILLOSCOPE
100 MHz 1 GS/s
Tek
Trig'd
M Pos: -20.00ns
CH1
耦合
直流
带宽限制
关 100MHz
伏/格
粗调
探头
10X
Voltage
反相
关闭
CH1 2.00V
M 25.0µs
10-Apr-20 17:04
CH1 ∕ 1.44V
12.2069kHz
探头检测
探头补偿
USB
Flash Drive

负锯齿波


Tektronix TDS1012B-SC TWO CHANNEL DIGITAL OSCILLOSCOPE
100 MHz 1 GS/s
Tek
Trig'd
M Pos: -20.00ns
CH1
耦合
直流
带宽限制
关 100MHz
伏/格
粗调
探头
10X
Voltage
反相
关闭
CH1 2.00V
M 25.0µs
10-Apr-20 17:04
CH1 ∕ 1.44V
12.2069kHz
保存
PRINT
打印
探头检测
探头补偿
USB
Flash Drive

锯齿波


Tektronix TDS1012B-SC TWO CHANNEL DIGITAL OSCILLOSCOPE
100 MHz 1 GS/s
Tek
Trig'd
M Pos: -20.00ns
CH1
耦合
直流
带宽限制
关 100MHz
伏/格
粗调
探头
10X
Voltage
反相
关闭
CH1 2.00V
M 25.0µs
10-Apr-20 17:04
CH1 ∕ 1.44V
12.2070kHz
保存
PRINT
打印
探头检测
USB
Flash Drive


Tektronix TDS1012B-SC TWO CHANNEL DIGITAL OSCILLOSCOPE
100 MHz 1 GS/s
Tek
Trig'd
M Pos: -20.00ns
CH1
耦合
直流
带宽限制
关 100MHz
伏/格
粗调
探头
10X
Voltage
反相
关闭
CH1 2.00V
M 25.0µs
10-Apr-20 17:04
CH1 ∕ 1.44V
12.2069kHz
保存
PRINT
打印
探头检测
USB
Flash Drive

三角波

方波

22.5 章节小结

在本实验中，我们利用 DMA 和自定义 IP 通过 ADA-106 子卡的 DAC 芯片实现了一些简单波形的输出。在此基础上，读者可以继续尝试实现更复杂的波形或者利用它输出的波形来控制其他设备模块，熟悉该子卡的使用方法。

第 23 章　　ADC 示波器实验

在之前的实验中，我们利用 ADA-106 子卡的 DAC 芯片实现了波形的输出，在本实验中，我们将学习使用该子卡上的另一个芯片 ADC08060，利用该芯片来读取 DAC 芯片的输出波形将其转化为数据，之后在 HDMI 屏幕上描绘出来，实现一个简单的示波器。

本章包括以下几部分内容：

23.1 实验简介

23.2 硬件设计

23.3 软件设计

23.4 下载验证

23.5 章节小结

23.1 实验简介

在实际应用中 ADC 模数转换是一种应用非常广泛的功能，在传感器数据采集，超声信号采集及处理、医疗影像等领域都有他的身影。他能将连续的模拟量转化为离散的有效的数字量来加以利用。

ADA-106 子卡包含一路高速 AD 转换，采用 TI 出产的 ADC08060 模数转换芯片，支持最大 60M 的转换速率，ADC 模拟输入范围为+/-5Vpp，其结构图如下：

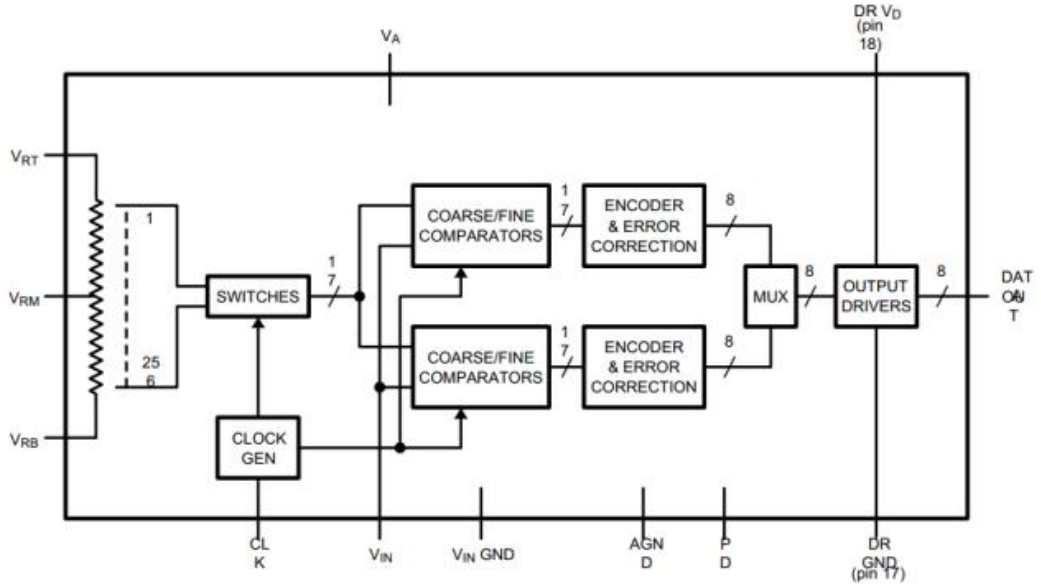


图 23.1.1 ADC08060 结构图

外部原理图设计如下图所示：

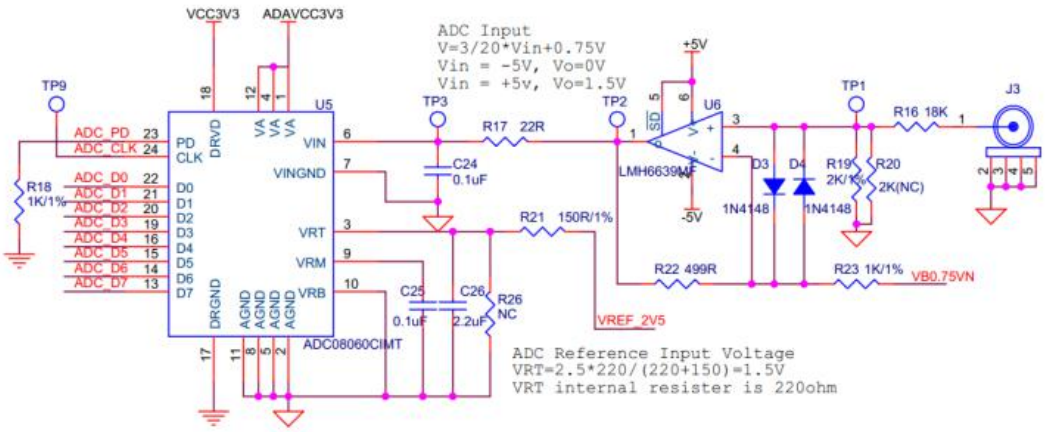


图 23.1.2 ADC 原理图

ADC 部分采用外部参考电压， AD 芯片内部 VRT 到 VRB 内部电阻为 220 欧姆，采用 2.5V 参考电压输入经过 R21(150 欧姆)与 AD 内部电阻（220 欧姆）分压，最终得到 AD 的参考电压为 1.5V。

所以 ADC 转换数字输出与 ADC 模拟输入的关系：

Code=256*Vinadc/1.5 (Vinadc 为 AD 芯片模拟输入脚的电压，即测试点 TP3 处的电压)

那么高速 ADDA 子卡模块模拟输入（即测试点 TP1 处电压）与 ADC08060 Pin6 Vinadc（即测试点 TP3 处的电压）出的输入关系为：

Vinadc=3/20*Vin+0.75V

Vinadc = -5V, Vo=0V

Vinadc = +5v, Vo=1.5V

更多关于芯片的信息请查阅数据手册。

23.2 硬件设计

在本实验中，我们在之前 HDMI 输出实验的基础上，分别添加关于 ADC 模数转换和 DAC 数模转换的自定义 IP 及 DMA IP 核，最终实现一个简单的示波器功能。首先打开之前实验 17 的 HDMI 显示实验的 Vivado 工程，将其另存为一个新的工程并命名为“ADDA_HDMI”，如下图所示：

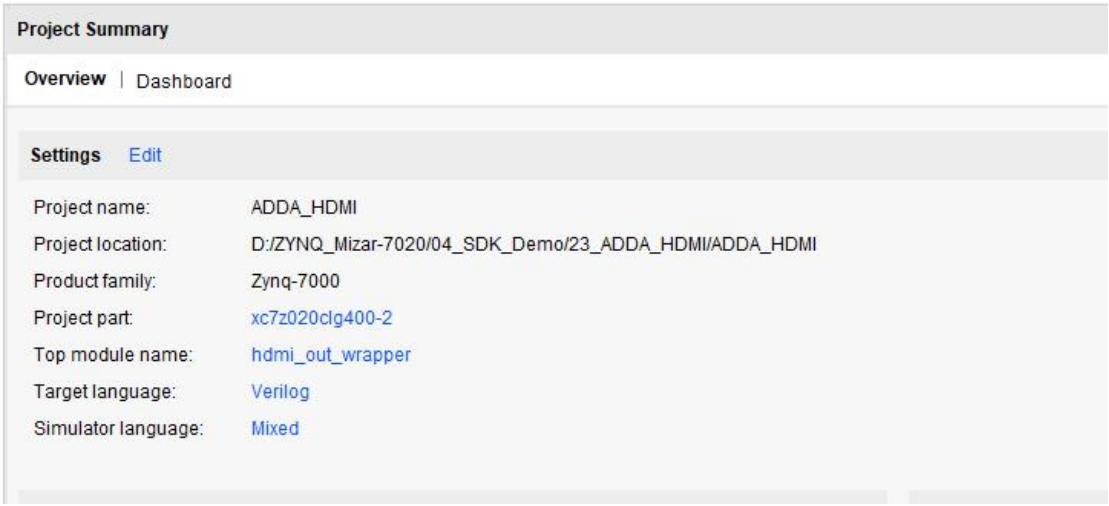


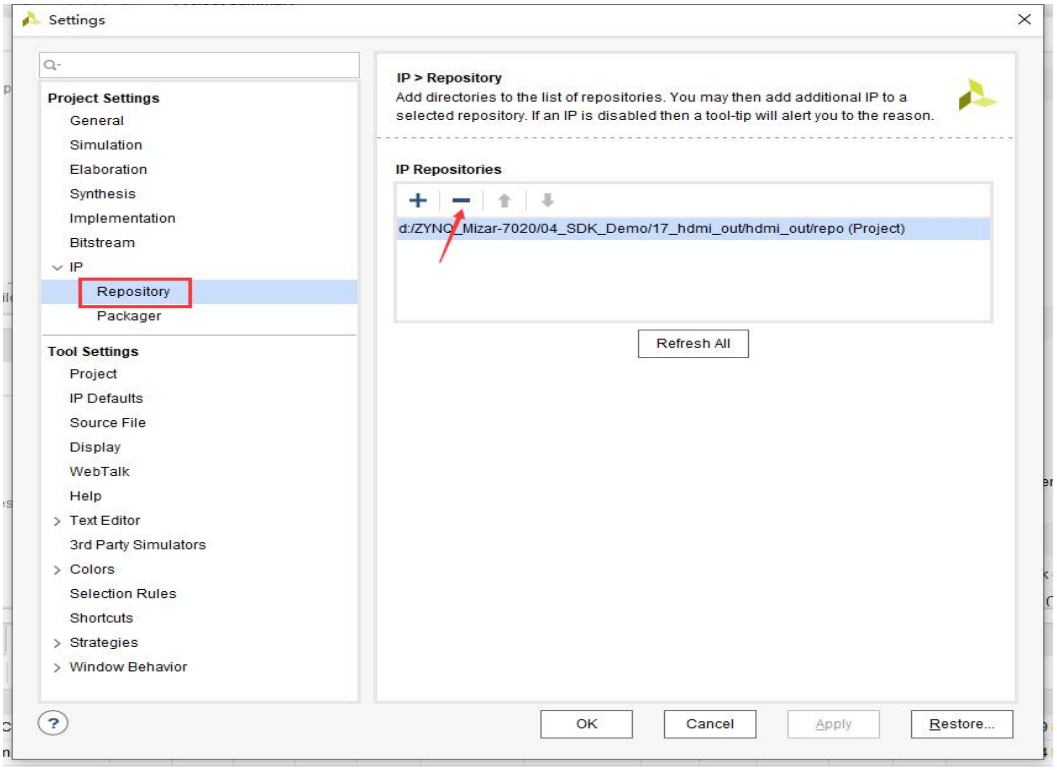
图 23.2.1 另存的“ADDA_HDMI”工程

之后删除新工程下的.sdk 文件夹这是之前实验的应用程序，此处不需要。之后将资料中该例程下的 repo 文件夹复制到该工程下，此文件夹包含 HDMI 显示

及 ADC、DAC 控制的自定义 IP 核，如下图所示：

图 23.2.3 删除.sdk，复制 repo 文件夹

之后在打开的 Vivado 软件中，点击左侧导航栏 Settings 打开设置页面，点击“-”删除并重新导入自定义 IP 核，将新添加的 IP 核导入工程，如下图所示：



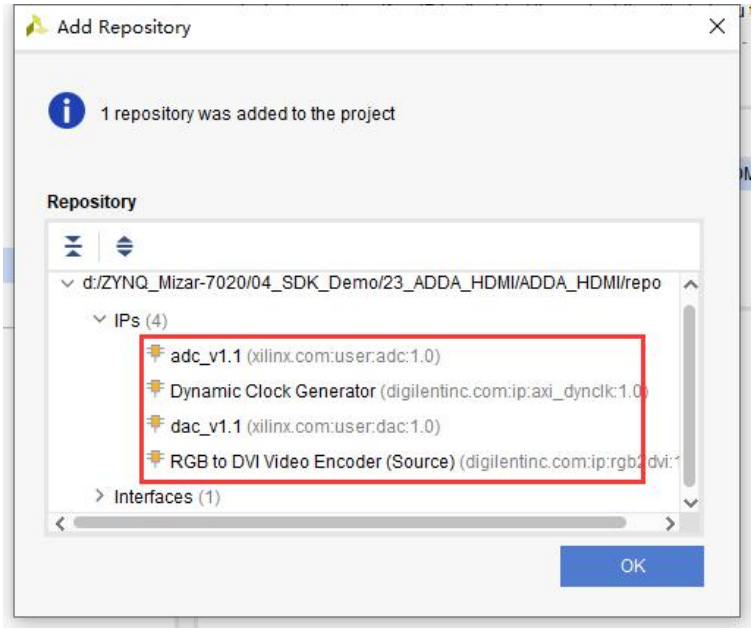


图 23.2.4 删除旧 IP

图 23.2.5 导入新 IP

导入完成后点击 Open Block Design 打开设计窗口，搜索并添加 ADC IP 核，如下图所示：

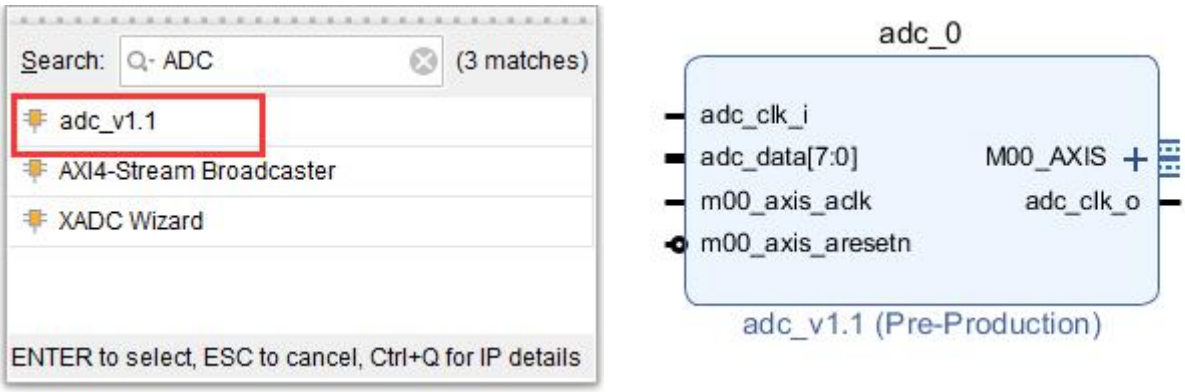
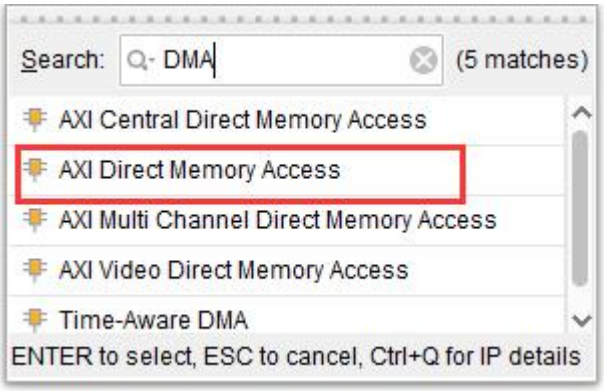


图 23.2.6 搜索并添加 ADC IP 核

搜索并添加 DAC IP 核，如下图所示：

图 23.2.7 搜索并添加 DAC IP 核

这两个 IP 用来分别向 dac 芯片写数据以及从 adc 芯片读取数据，无需修改设置，保持默认即可。之后搜索并添加 DMA IP 核，用来将 DDR 的数据写入 dac_v1.1



以及从 adc_v1.1 读取数据并写入 DDR 中。

图 23.2.8 搜索添加 DMA IP 核

添加完成后双击该 IP 核打开设置窗口，如下图所示设置各项相关设置，

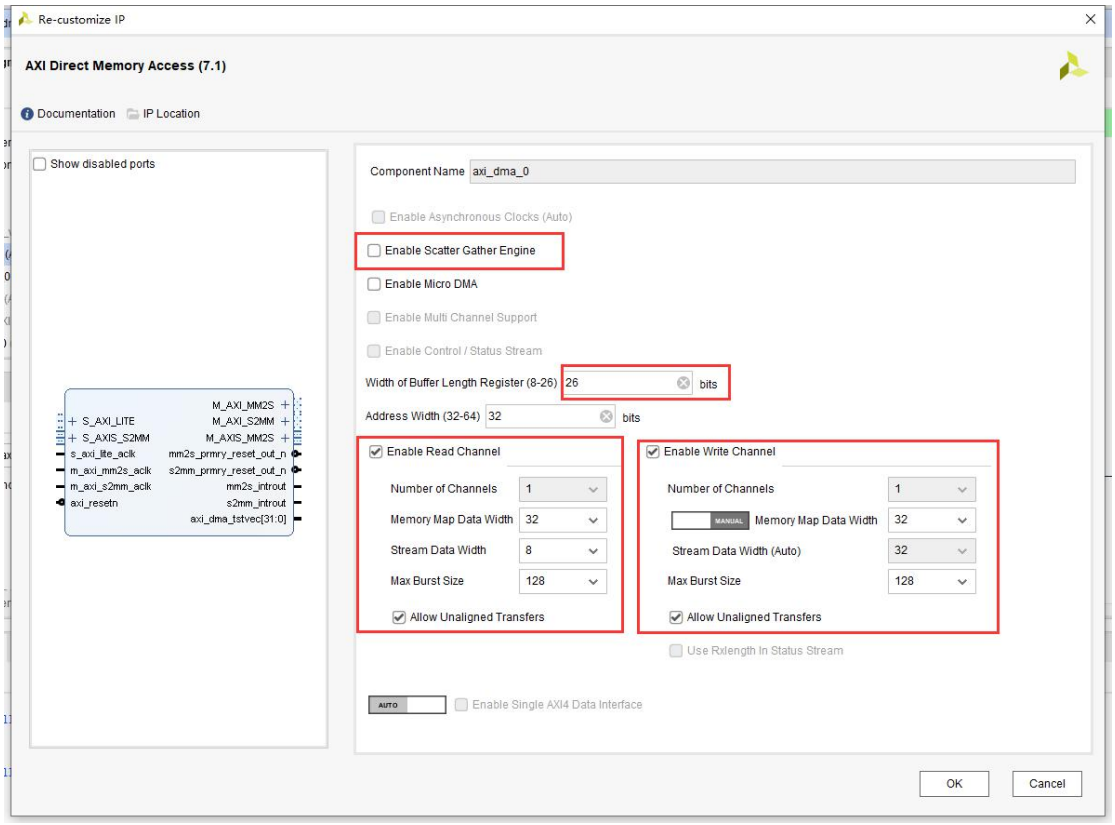


图 23.2.9 设置 DMA IP 核相关设置

连接相关信号接口，连接 adc_0 的 M00_AXIS 端口和 DMA IP 的 S_AXIS_S2MM 端口，连接 dac_0 的 S00_AXIS 接口和 DMA IP 的 M_AXI_S2MM 接口，如下图所示：

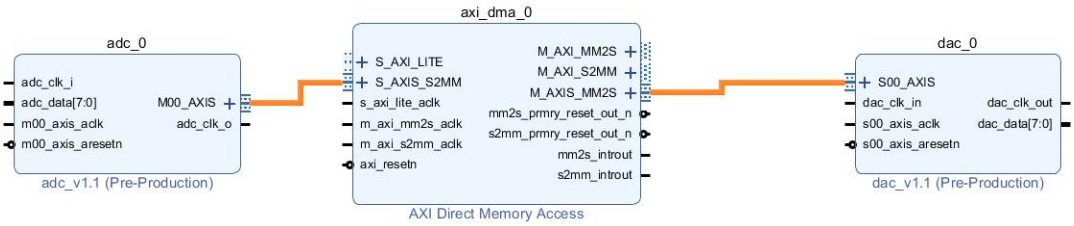


图 23.2.10 连接各 IP 核的 AXI 总线接口

在本实验中，我们需要用到 DMA 的接收中断和发送中断，即 mm2s_introut 和 s2mm_introut，将这两个中断信号连接到 PS 端，需要修改 Concat IP 核的设置，双击 Concat IP 核打开设置页面，将端口数修改为 5，如下图所示：

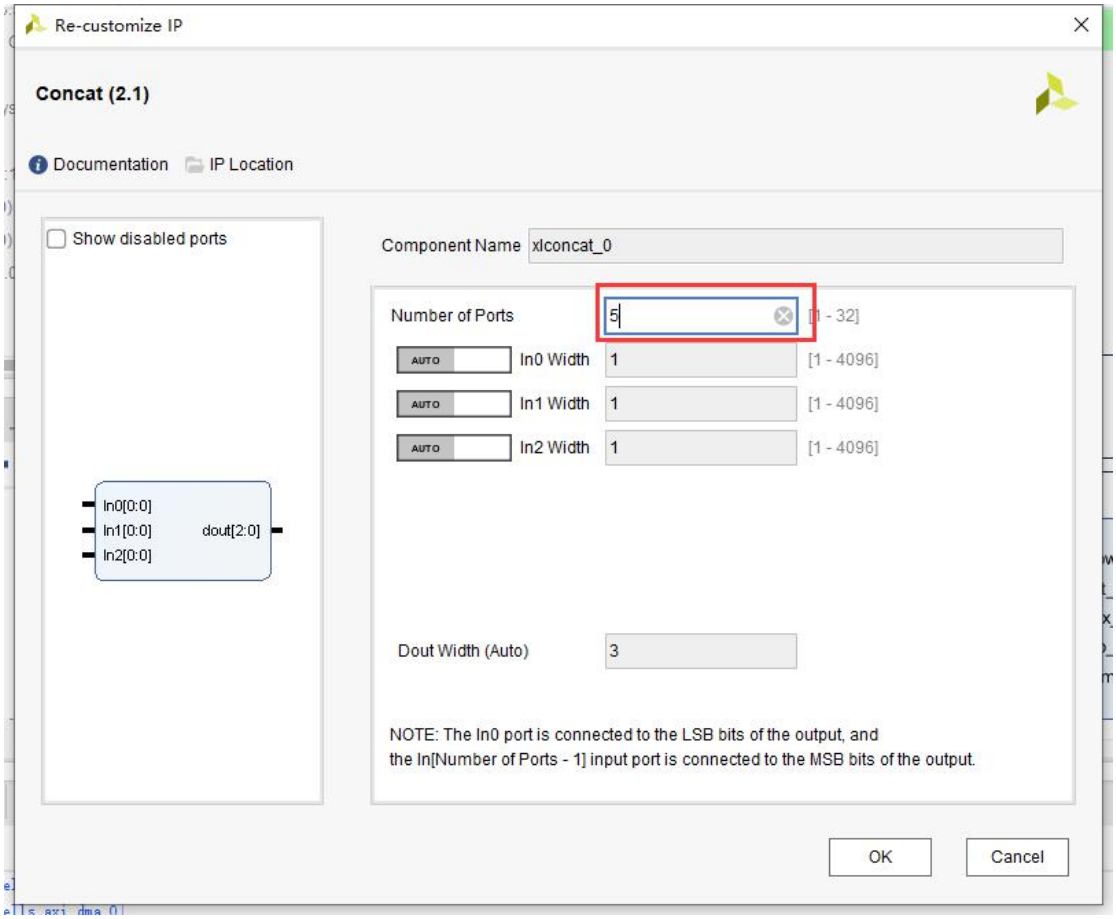
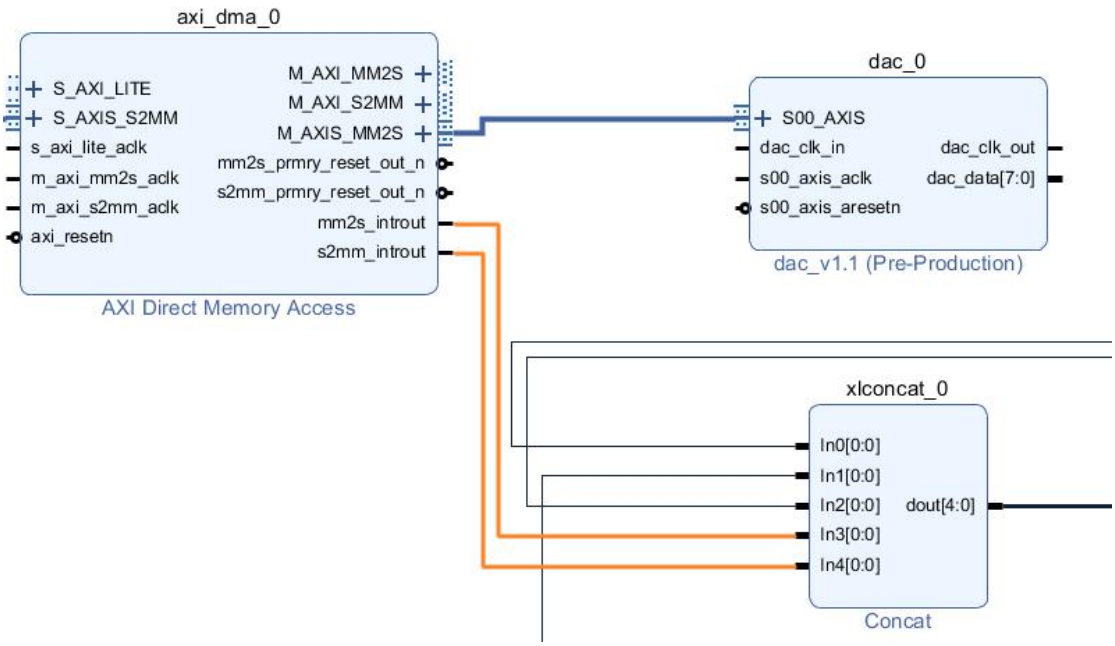


图 23.2.11 修改 Concat IP 端口数

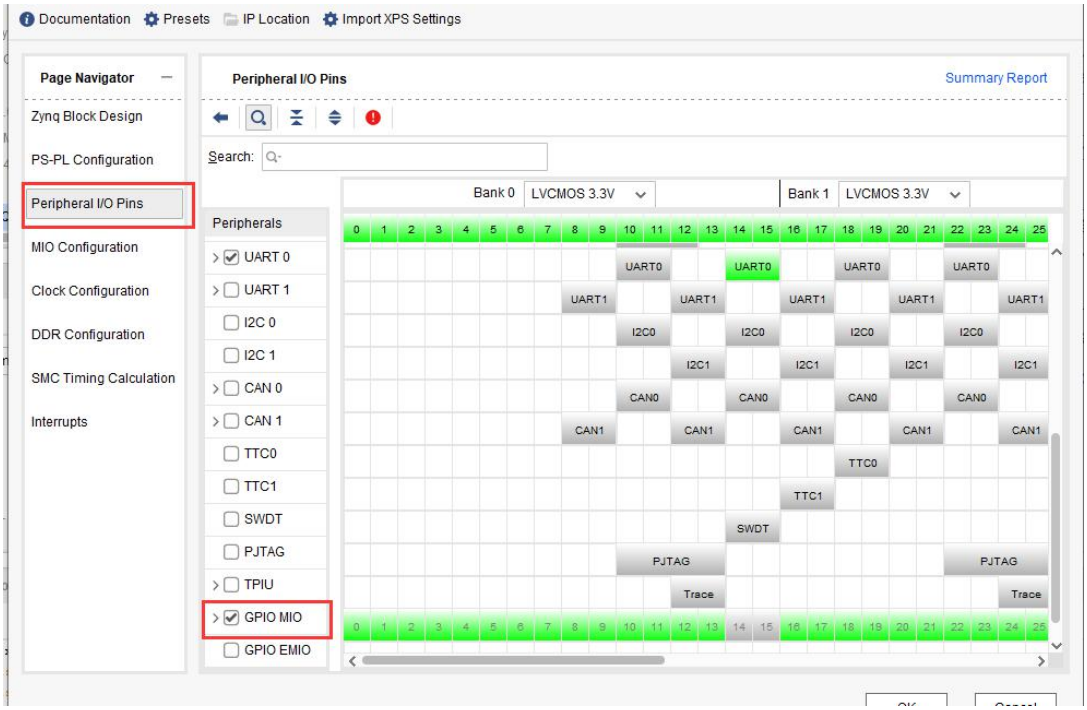
修改完成后将 mm2s_introut 和 s2mm_introut 这两个中断信号连接到 Concat



IP 的输入端口，如下图所示：

图 23.2.12 连接中断信号

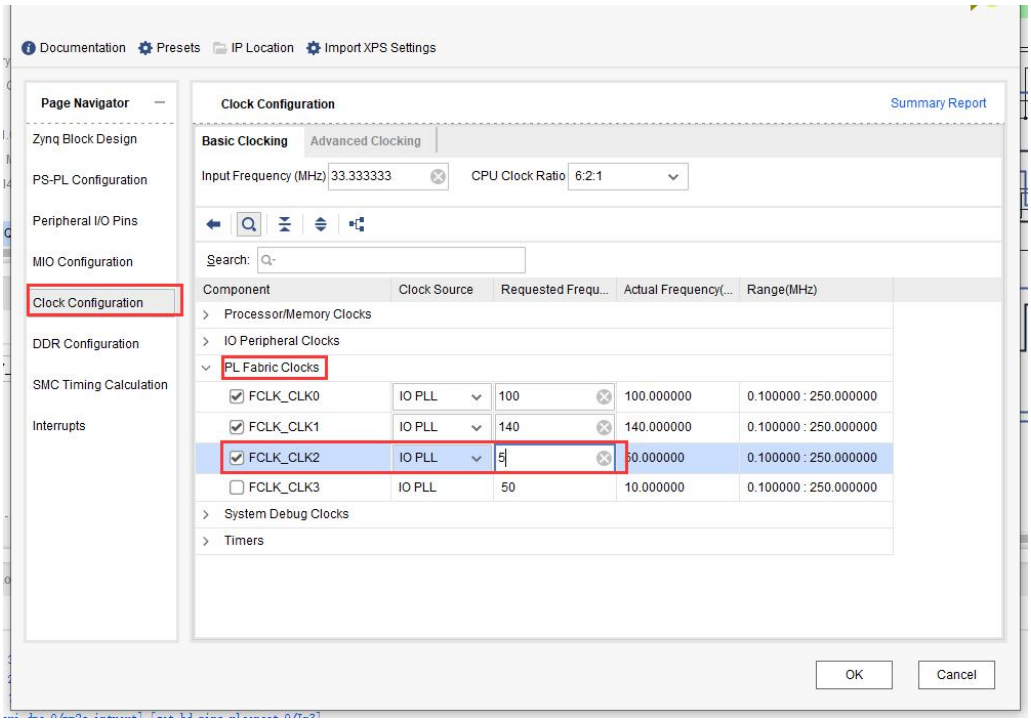
双击 ZYNQ7 Processing System IP 打开设置页面，点击 Perpheral I/O Pins，使



能 GPIO MIO，连接 PS_KEY1，用来切换 DAC 的输出波形，如下图所示：

图 23.2.13 使能 GPIO MIO

点击 Clock Configurations，使能 FCLK_CLK2，设置输出频率为 5M，作为 ADC



的采样频率，如下图所示：

图 23.2.14 设置 FCLK_CLK2 输出频率

设置完成后将 FCLK_CLK2 连接到 adc_0 的 adc_clk_i 时钟信号输入端口，如下图所示：

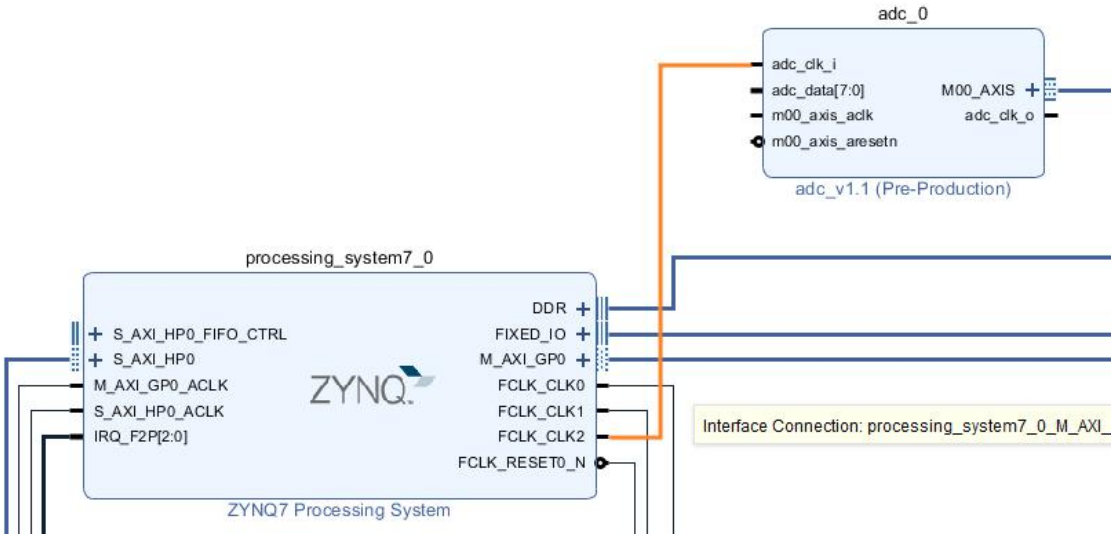
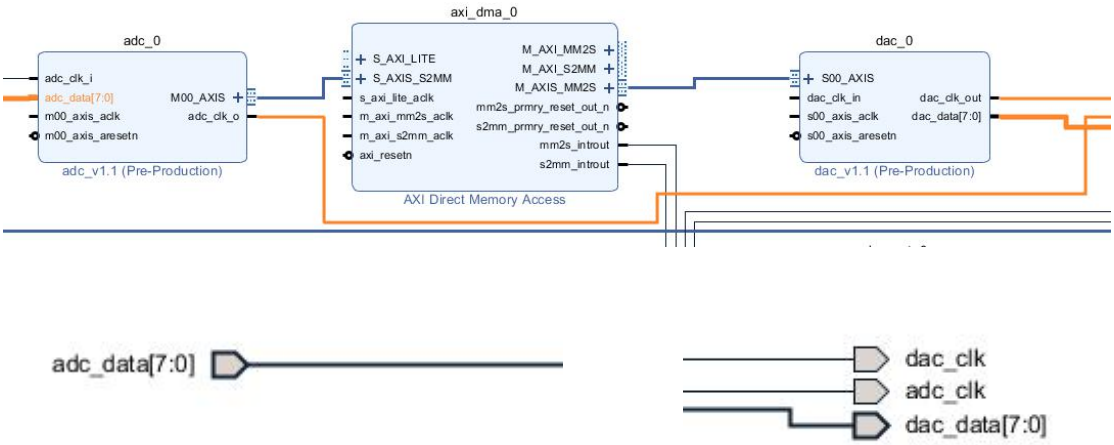


图 23.2.15 连接时钟信号

将 adc_0 和 dac_0 两个 IP 核的时钟及数据信号端口引出，鼠标右键点击信



号端口，选择 Make External，之后修改各信号的名称，如下图所示：

图 23.2.16 引出信号端口及修改信号名称

点击 Run Connection Automation，全选所有信号，自动连接其他相关信号，如下图所示：

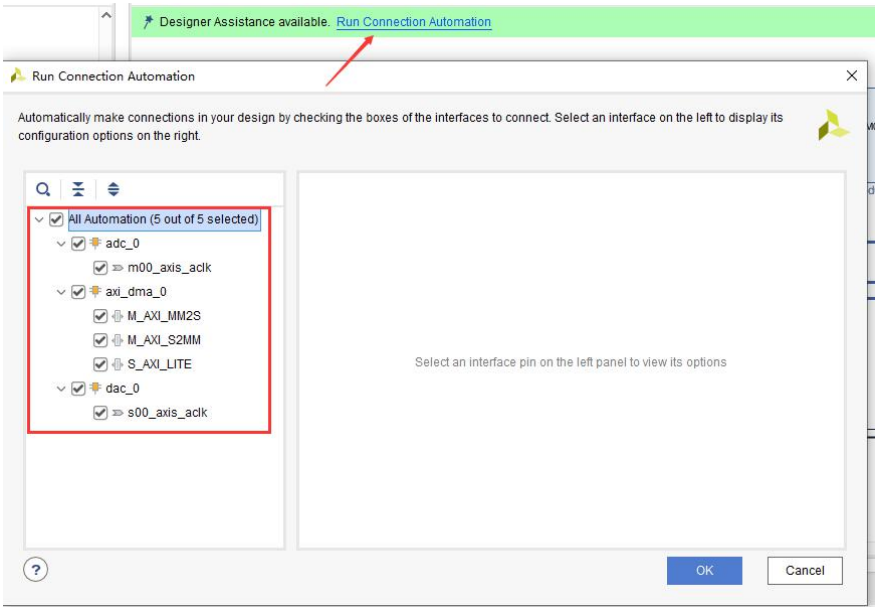


图 23.2.17 自动连接信号

连接完成后，手动连接 dac_clk_in 到 s00_axis_aclk，即系统时钟，如下图所示：

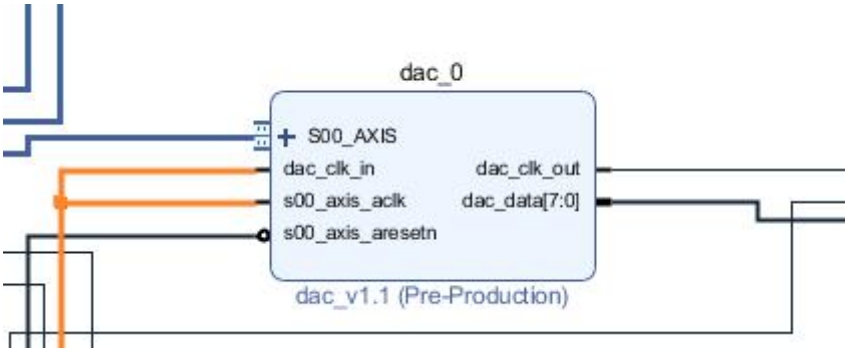


图 23.2.18 连接 dac_clk_in 信号到系统时钟

添加一个 xlconstant IP 核模块，并对其进行设置如图 a 1 所示。位宽是 2，输出 Const Val 0 为低电平。如果不增加这一步，后续 ADA 将无法工作。

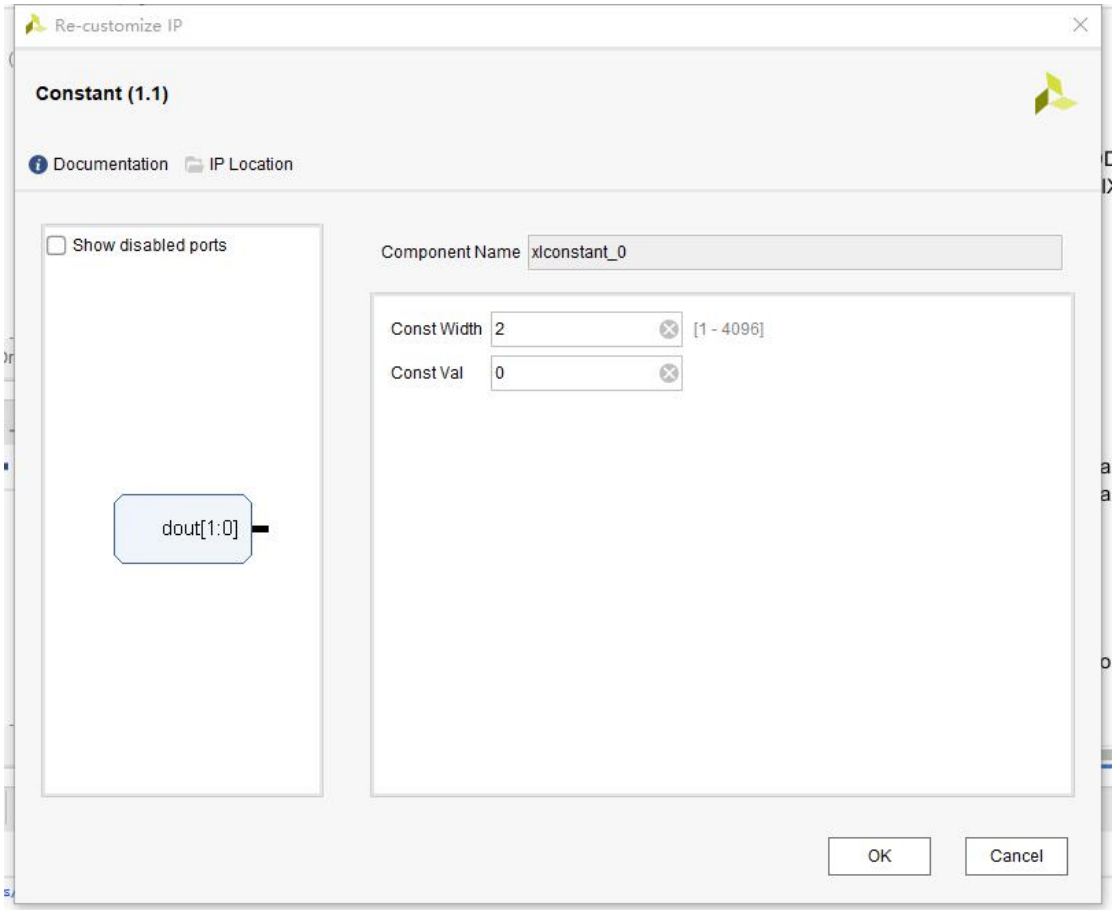


图 a 1

到此硬件系统设计就完成了，验证并保存设计。完整的硬件设计如下图所示：

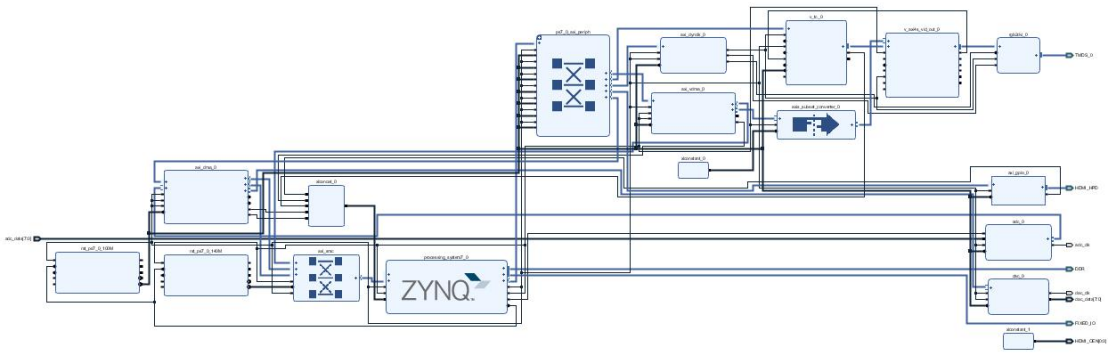
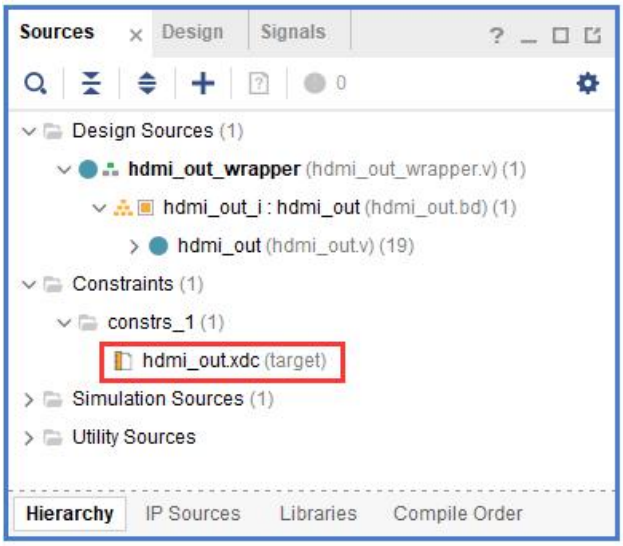


图 23.2.19 硬件设计系统图

之后为新添加的一些信号绑定引脚，在 Source 串口双击打开引脚约束文件



hdmi_out.xdc，如下图所示：

图 23.2.20 打开引脚约束文件

在文件中输入以下内容绑定引脚约束：

```
set_property IOSTANDARD LVCMOS33 [get_ports {HDMI_HPD_tri_i[0]}]
set_property PACKAGE_PIN P19 [get_ports {HDMI_HPD_tri_i[0]}]
set_property PACKAGE_PIN U18 [get_ports TMDS_0_clk_p]
set_property PACKAGE_PIN V20 [get_ports {TMDS_0_data_p[0]}]
set_property PACKAGE_PIN T20 [get_ports {TMDS_0_data_p[1]}]
set_property PACKAGE_PIN N20 [get_ports {TMDS_0_data_p[2]}]

set_property PACKAGE_PIN W19 [get_ports {dac_data[0]}]
set_property PACKAGE_PIN W18 [get_ports {dac_data[1]}]
set_property PACKAGE_PIN U17 [get_ports {dac_data[2]}]
set_property PACKAGE_PIN T16 [get_ports {dac_data[3]}]
set_property PACKAGE_PIN R17 [get_ports {dac_data[4]}]
set_property PACKAGE_PIN R16 [get_ports {dac_data[5]}]
set_property PACKAGE_PIN P18 [get_ports {dac_data[6]}]
set_property PACKAGE_PIN N17 [get_ports {dac_data[7]}]
set_property PACKAGE_PIN Y18 [get_ports dac_clk]

set_property SLEW FAST [get_ports {dac_data[*]}]
set_property SLEW FAST [get_ports dac_clk]

set_property IOSTANDARD LVCMOS33 [get_ports {dac_data[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports dac_clk]

set_property PACKAGE_PIN U12 [get_ports adc_clk]
set_property PACKAGE_PIN V13 [get_ports {adc_data[0]}]
set_property PACKAGE_PIN U13 [get_ports {adc_data[1]}]
set_property PACKAGE_PIN U15 [get_ports {adc_data[2]}]
set_property PACKAGE_PIN U14 [get_ports {adc_data[3]}]
set_property PACKAGE_PIN R14 [get_ports {adc_data[4]}]
set_property PACKAGE_PIN P14 [get_ports {adc_data[5]}]
set_property PACKAGE_PIN W15 [get_ports {adc_data[6]}]
set_property PACKAGE_PIN V15 [get_ports {adc_data[7]}]
```

```

set_property SLEW FAST [get_ports adc_clk]

set_property IOSTANDARD LVCMOS33 [get_ports adc_clk]
set_property IOSTANDARD LVCMOS33 [get_ports {adc_data[*]}]


set_property IOSTANDARD LVCMOS33 [get_ports {dout[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {dout[1]}]
set_property PACKAGE_PIN Y19     [get_ports {dout[0]}]
set_property PACKAGE_PIN T12     [get_ports {dout[1]}]

```

之后点击左侧导航栏 Generate Bitstream 生成 BIT 文件。

点击 File->Export->Export Hardware 导出硬件设计。

导出完成后点击 File->Launch SDK 打开 SDK。

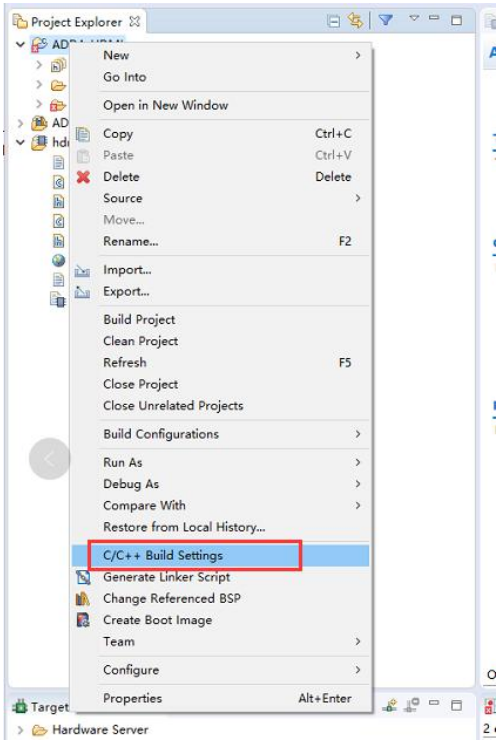
23.3 软件设计

在打开的 SDK 软件中创建新的应用工程并命名为“ADDA_HDMI”，工程模板选择空白。之后打开资料例程文件，将下列源代码文件复制到新建工程下的相同

位置，如下图所示：

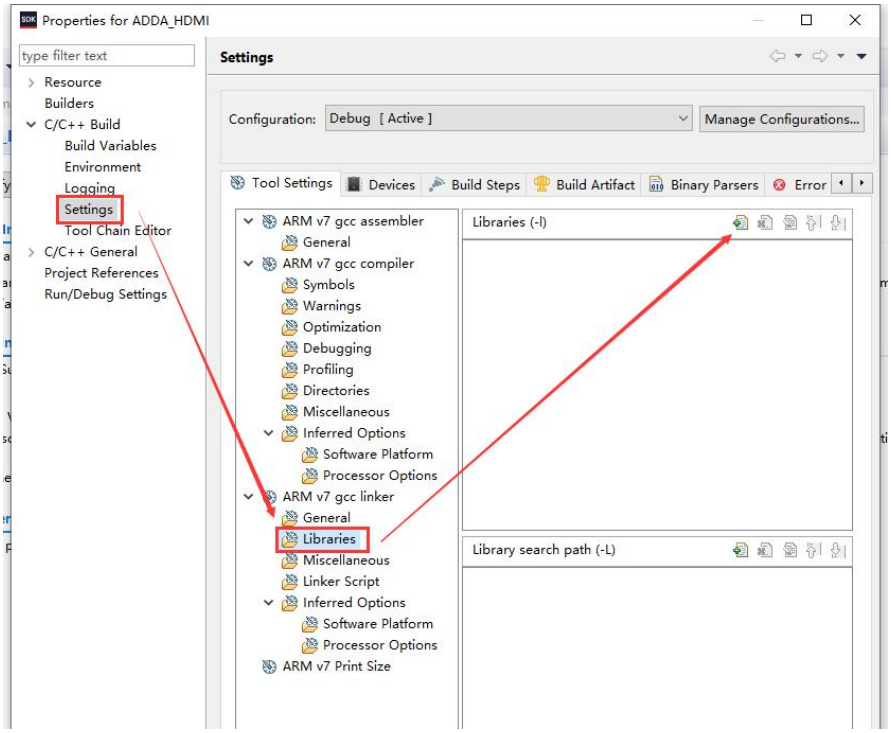
图 23.3.1 代码源文件

该实验需要使用一些数学计算函数，需添加相关库函数。右键点击应用工程，



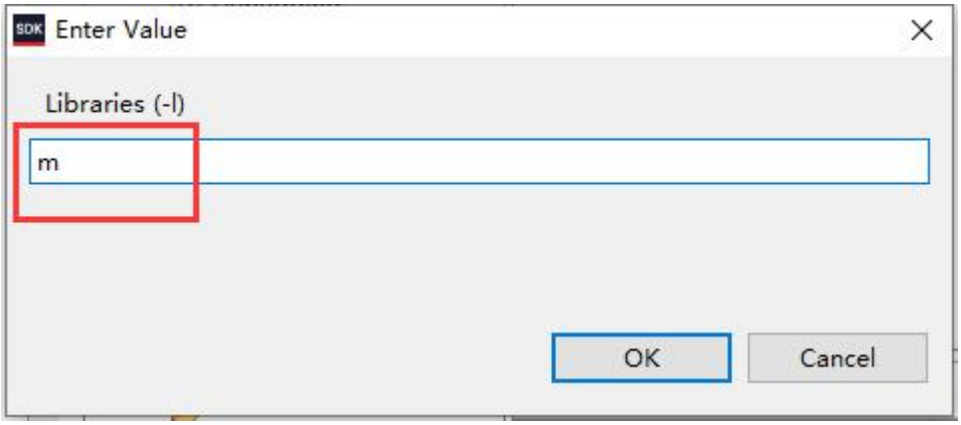
选择 C/C++ Build Settings，如下图所示：

图 23.3.2 打开设置页面



在打开的页面中，点击 Libraries，点击右上侧窗口的“+”，如下图所示：

图 23.3.3 添加库函数



在弹出的窗口中输入“m”，点击 OK 即可，如下图所示：

图 23.3.4 添加数学计算相关库函数

在本实验的软件设计中主要包括 dac 波形输出及 adc 数据采集并通过 HDMI 输出两部分。波形输出部分与上一实验相同，显示部分初始化设置与实验 17 相同，此处不再重复讲解，遗忘的读者可以查阅之前的实验。adc 数据采集并通过

HDMI 输出部分主要包含 adc 数据获取和将获取的数据描绘成显示在屏幕上的波形。

在 int XAxiDma_Init()函数中使能 S2MM 中断，用来判断 DMA 读取数据的状

```
/* Enable S2MM interrupt, Enable MM2S interrupt */
XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_IOC_MASK,
        XAXIDMA_DMA_TO_DEVICE);
XAxiDma_IntrEnable(XAxiDma, XAXIDMA_IRQ_IOC_MASK,
        XAXIDMA_DEVICE_TO_DMA);
```

态，

在 DMA 中断处理函数中添加对于 s2mm 中断的处理，设置中断标志 s2mm_flag 为 1，清空显示波形缓冲区，

```
s2mm_sr = XAxiDma_IntrGetIrq(XAxiDmaPtr, XAXIDMA_DEVICE_TO_DMA) ;

if (s2mm_sr & XAXIDMA_IRQ_IOC_MASK)
{
    /* Clear interrupt */
    XAxiDma_IntrAckIrq(XAxiDmaPtr, XAXIDMA_IRQ_IOC_MASK,
            XAXIDMA_DEVICE_TO_DMA) ;
    /* Invalidate the Data cache for the given address range */
    Xil_DCacheInvalidateRange((INTPTR)DmaRxBuffer, ADC_CAPTURELEN);

    s2mm_flag = 1 ;
}
```

对于波形显示部分，首先设置显示背景。本次实验我们选用 HDMI 输出显示彩条作为显示背景，void DemoPrintTest(u8 *frame, u32 width, u32 height, u32 stride)函数将在屏幕上显示彩条。

之后我们在彩条背景上叠加灰色的网格作为波形背景，在该函数中包括宽度参数 width 和高度参数 height，用来控制显示网格高与宽。网格设置为每格 32*32 像素，灰色网格点每隔 4 个像素显示。

```c
void draw_grid(u32 width, u32 height, u8 *CanvasBufferPtr)
{

    u32 xcoi, ycoi;
    u8 wRed, wBlue, wGreen;
    /*
     * overlay grid on canvas, background set to black color, grid color is gray.
     */
    for(ycoi = 0; ycoi < height; ycoi++)
    {
        for(xcoi = 0; xcoi < width; xcoi++)
        {

            if (((ycoi == 0 || (ycoi+1)%32 == 0) && (xcoi == 0 || (xcoi+1)%4 == 0))
                    || ((xcoi == 0 || (xcoi+1)%32 == 0) && (ycoi+1)%4 == 0))
            {
                /* gray */
                wRed = 150;
                wGreen = 150;
                wBlue = 150;
            }
            else
            {
                /* Black */
                wRed = 0;
                wGreen = 0;
                wBlue = 0;
            }
            draw_point(CanvasBufferPtr, xcoi, ycoi, width, wBlue, wGreen, wRed);
        }
    }
}
```

叠加波形的函数为 void draw_wave()；函数功能及各参数介绍如下：

```c
 * Draw wave on canvas
 *
 *@param BufferPtr        data buffer for drawing wave
 *@param CanvasBufferPtr  this is CanvasBuffer pointer, draw wave on CanvasBuffer
 *@param Sign             data sign: unsigned char 0; char 1; unsigned short 2 ;short 3
 *@param Bits             data valid bits
 *@param color            color select for wave
 *@param coe              coefficient
 *
 *@note  this function draw line between two point through checking last data and current data value,
 *       convert data to u8 by adder and coe
 */
void draw_wave(u32 width, u32 height,  void *BufferPtr, u8 *CanvasBufferPtr, u8 Sign, u8 Bits, u8 color, u16 coe)
```

最后当 s2mm 中断触发表明接收到 ADC 数据之后，描绘波形，并将波形叠

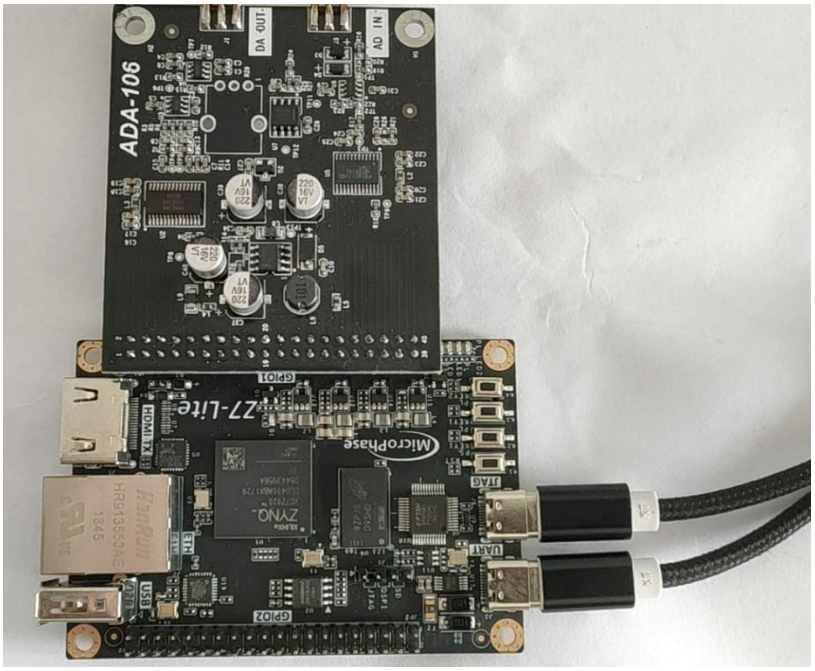
```c
if (s2mm_flag)
{
    /* clear s2mm_flag */
    s2mm_flag = 0 ;
    /* Copy grid to Wave buffer */
    memcpy(RxWaveBuffer, GridBuffer, WAVE_LEN) ;
    /* wave Overlay */
    draw_wave(dispCtrl.vMode.width, WAVE_HEIGHT, (void *)DmaRxBuffer, RxWaveBuffer, UNSIGNEDCHAR, ADC_BITS, YELLOW, ADC_COE) ;
    /* Copy Canvas to frame buffer */
    frame_copy(dispCtrl.vMode.width, WAVE_HEIGHT, dispCtrl.stride, WAVE_START_COLUMN, WAVE_START_ROW,
            dispCtrl.framePtr[dispCtrl.curFrame], RxWaveBuffer) ;
    /* delay 100ms */
    usleep(100000) ;
    /* start DMA translation from ADC08060 to DDR3 */
    Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) DmaRxBuffer,
            MAX_DMA_LEN, XAXIDMA_DEVICE_TO_DMA);
```

加到之前显示的图像上就完成了波形的显示。

程序设计完成之后保存编译，没有错误即可烧录验证。

23.4 下载验证

首先将 ADA-106 子卡插入 GPIO1 插座中，注意插接的方向。然后使用 type-c 线线将开发板的 UART 接口与电脑连接，用于串口通信。然后我们将使用 type-c



线与 JTAG 接口连接，另外一端与电脑连接。ADA-106 子卡连接如下图所示：

图 23.4.1 连接子卡及其他连接线

点击 Run->Run Configurations，如下图所示设置 bit 文件烧录选项：

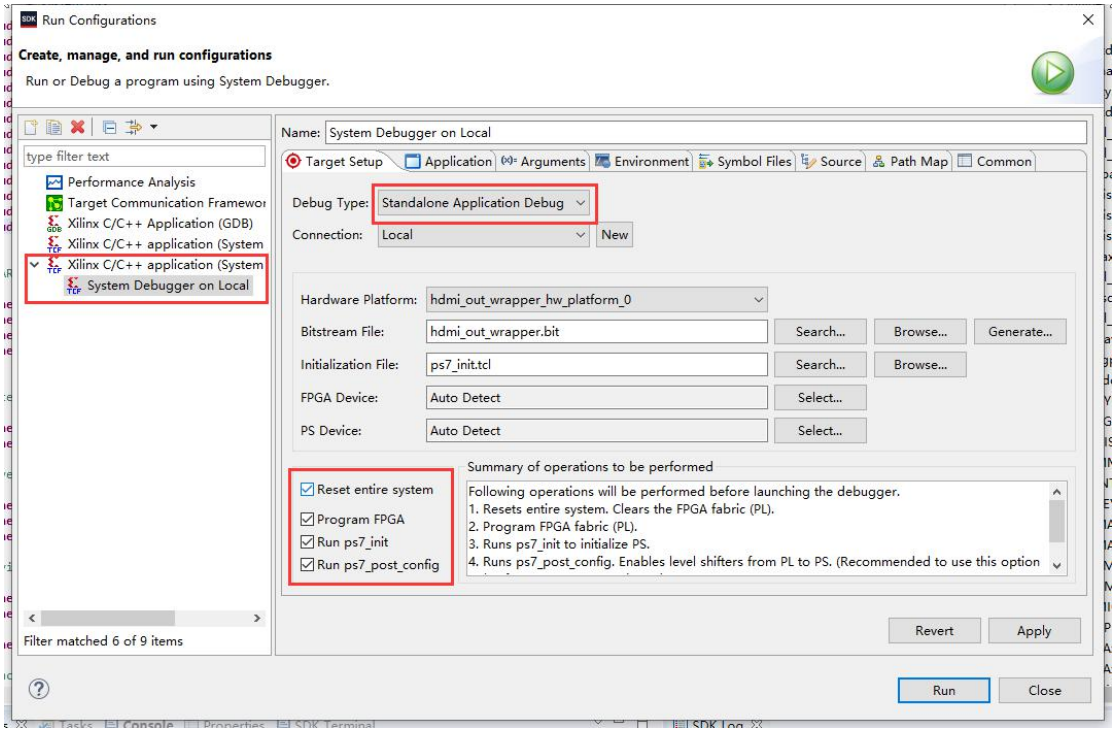


图 23.4.2 设置烧录选项

之后设置应用工程烧录选项，如下图所示：

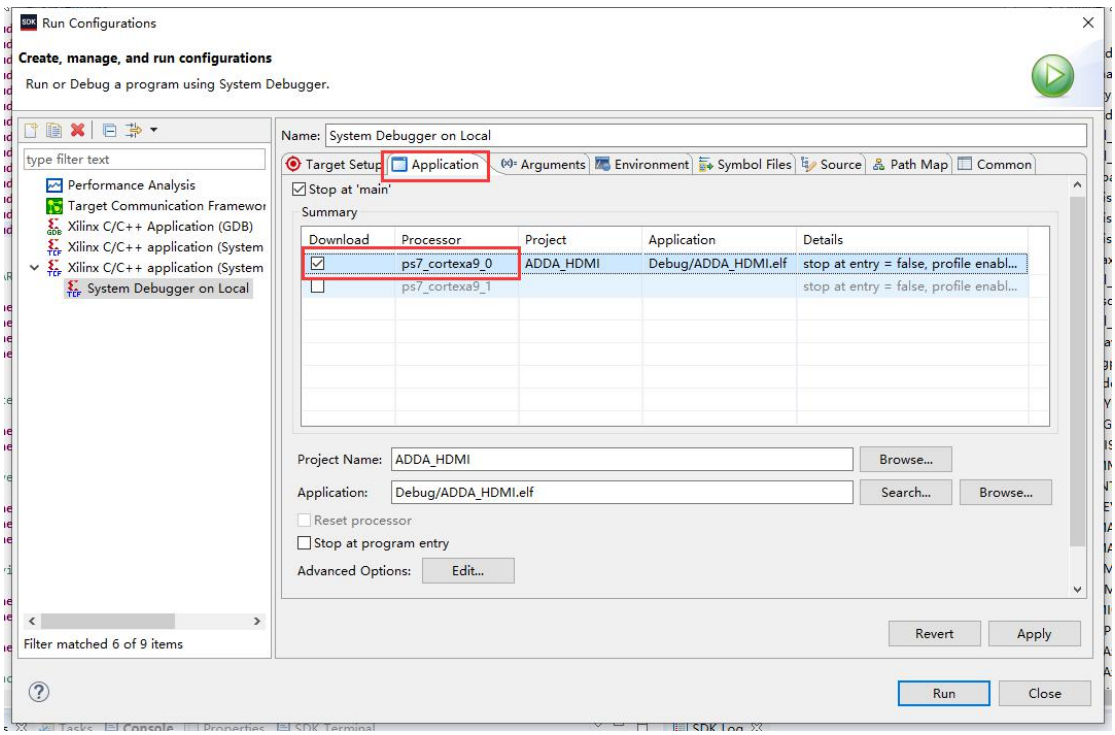


图 23.4.3 设置应用程序烧录选项

设置完成后点击 Run 将程序烧录到开发板中，之后使用 HDMI 连接线连接开

发板 HDMI TX 接口及显示器接口，之后即可看到屏幕上显示出当前 DAC 芯片输出的波形，如下图所示：

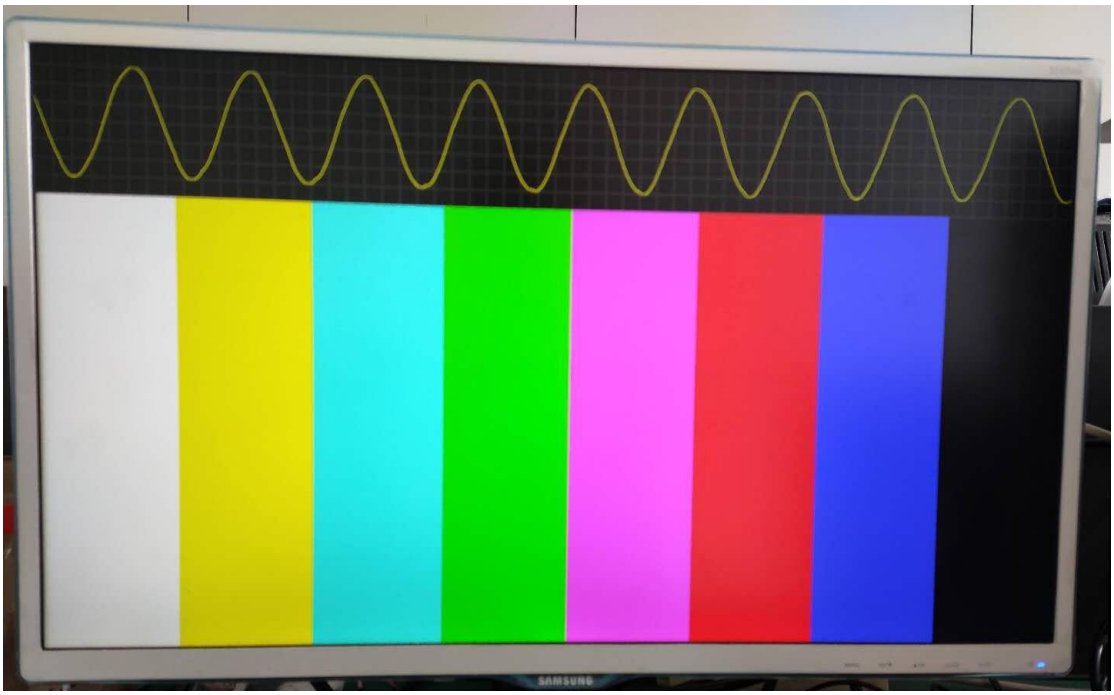


图 23.4.4 屏幕显示波形

按下按键 PS_KEY1 可以切换输出波形，如下图所示：

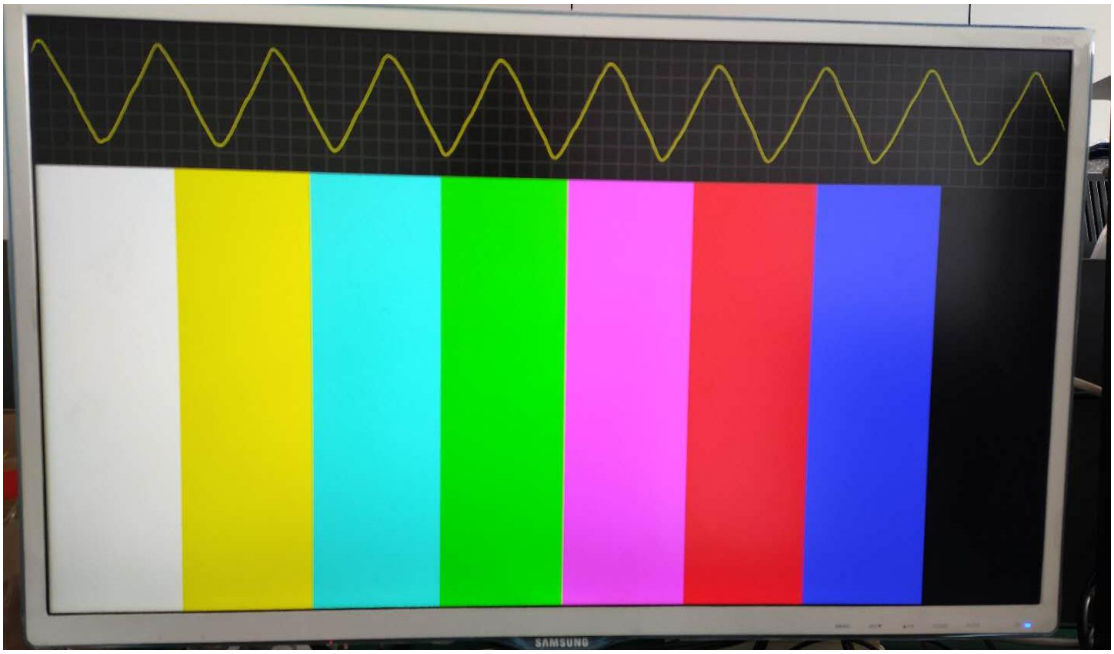


图 23.4.5 输出波形为三角波时显示图像

到此我们就实现可一个简单的示波器实验。

23.5 章节小结

在本实验中，我们结合之前的 HDMI 显示实验、DAC 波形发生器实验实现了从波形产生、数据采集到显示器显示的一个简单的示波器，学习掌握 ADC 芯片的使用方法，并结合其他外设实现相应的功能。

第 24 章　　基于 TCP 协议的远程更新 QSPI Flash 实验

在之前的实验中我们通常使用 JTAG 下载的方式将程序烧录到开发板上，这种方法方便快捷，但缺点也很突出，那就是程序在掉电后丢失。为解决这种问题，我们在第 10 章的实验中分别介绍了将程序固化到 QSPI 和 SD 卡中这两种方法。这两种方法很实用，重新固化也非常方便。但在实际应用中，这种方法并不一定可行，例如产品安装在设备内部；程序预先固化到 QSPI 中，未预留 JATG 接口；需更新的设备众多并分布在全国各地，在这些情况下现场通过 SDK 软件重新固化的方式就不再可行。由此可见一种可以通过网络远程更新 QSPI 的方式是非常必要而且重要的，而且这种方式操作方便，通过网络即可服务全国各地的客户。在本实验中，我们将使用 TCP 协议实现远程更新 QSPI。

本章包括以下几部分内容：

24.1 实验简介

24.2 硬件设计

24.3 软件设计

24.4 下载验证

24.5 章节小结

24.1 实验简介

程序固化的实质就是将程序保存在非易失性存储设备中，例如 QSPI 和 SD 卡，在开发板启动时读取程序然后运行。在之前的实验中我们利用 SDK 软件，通过 JATG 接口直接将程序写入 QSPI，另外，我们还可以调用相关函数的方式将数据写入 QSPI 中。本实验我们将使用后一种方式。

在远程更新过程中，我们将电脑作为客户端，将客户端生成的 BOOT.bin 文件通过常见的 TCP、UDP 等网络协议发送给远端联网的文件接收端，接收端将受到的文件保存在 DDR3 中，等待文件传输完成并接收到相应的更新命令后，将文件写入到 QSPI 中去。之后接收端以 QSPI 启动的方式重新启动即可完成远程更新。为防止数据写入过程中出错，在写入完成后需要将数据读出进行校验，确保数据的正确性。

所以在本实验中需要完成的任务有：1、通过 TCP 协议将 BOOT.bin 文件传输到接收端；2、发送“update”命令表示传输完成或发送“clear”命令表示传输无效，删除接收到的数据；3、如收到“update”命令，将数据写入 QSPI，并读出校验；4、以 QSPI 方式重启完成远程更新。

24.2 硬件设计

在之前的 PS 端以太网实验中，我们使能了 QSPI 等相关接口，所以在本实验的硬件设计与第 10 章 PS 端以太网实验相同。

打开《PS 端以太网实验》的 Vivado 工程，另存为一个新的工程并命名为“tcp_update”，点击“OK”。如下图所示：

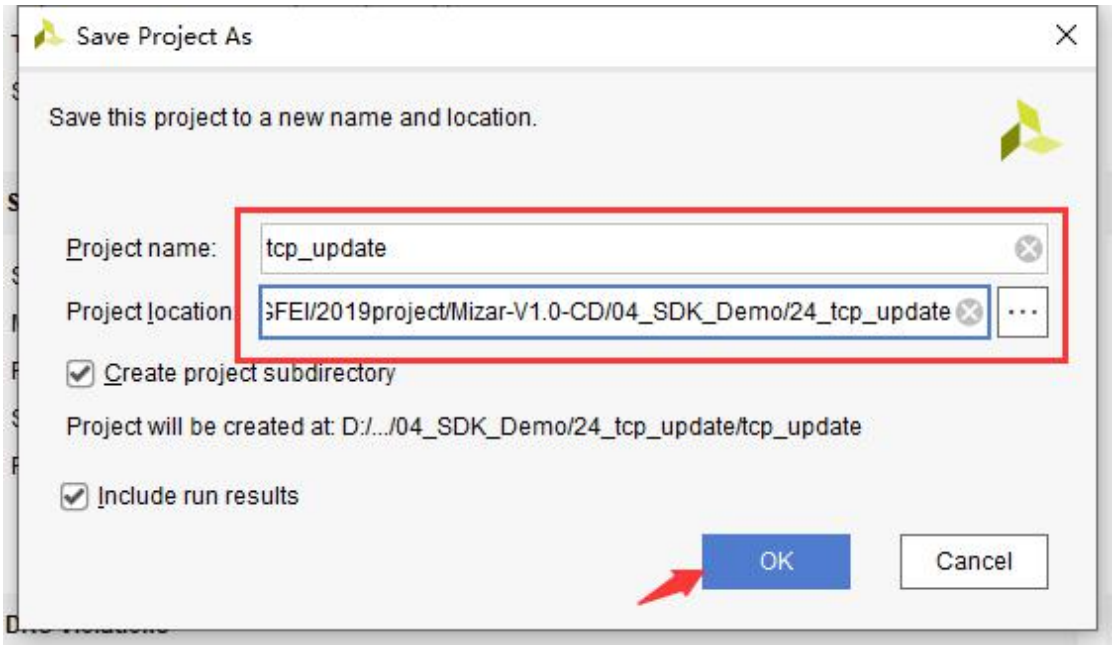


图 24.2.1 另存为新工程

另存完成后软件会自动打开，因为本实验硬件设计无需修改，所以可以直接导出到 SDK 进行软件设计。导出硬件设计之前删除工程目录下的.sdk 文件夹，这是之前实验的程序，本实验不需要。点击 File->Export->Export Hardware 导出硬件设计。导出完成后点击 File->Launch SDK 打开 SDK。

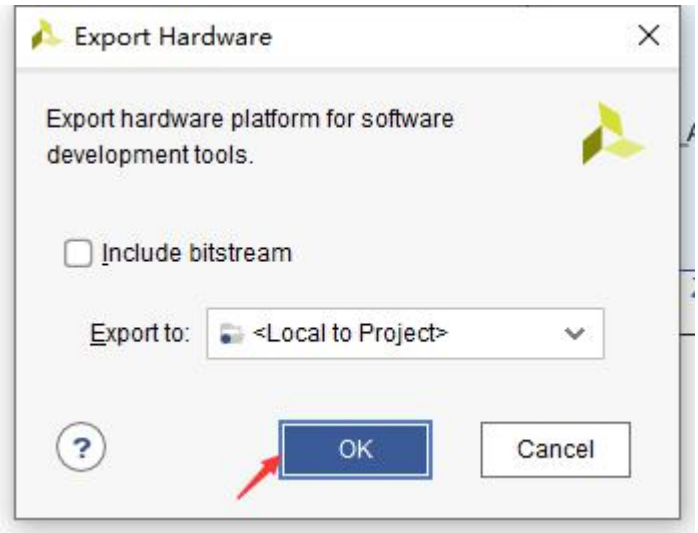
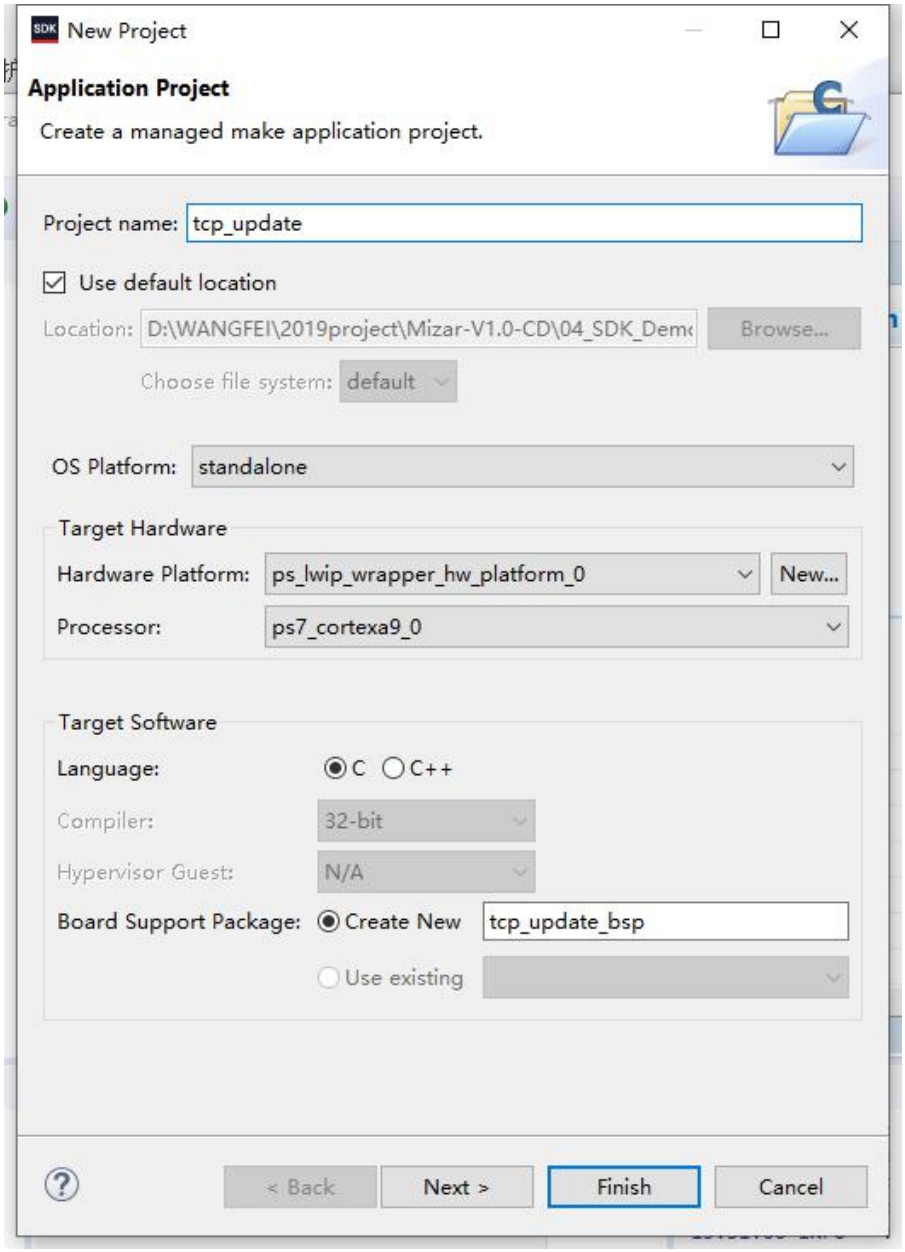


图 24.2.2 导出硬件设计

24.3 软件设计

在本次实验中我们主要将从网络接收的文件进行保存，之后再根据不同的命令执行删除文件或将数据写入 QSPI 的操作。从程序框架而言与之前的以太网实验相同，只是将之前返回接收数据的功能改为远程更新 QSPI 的功能。

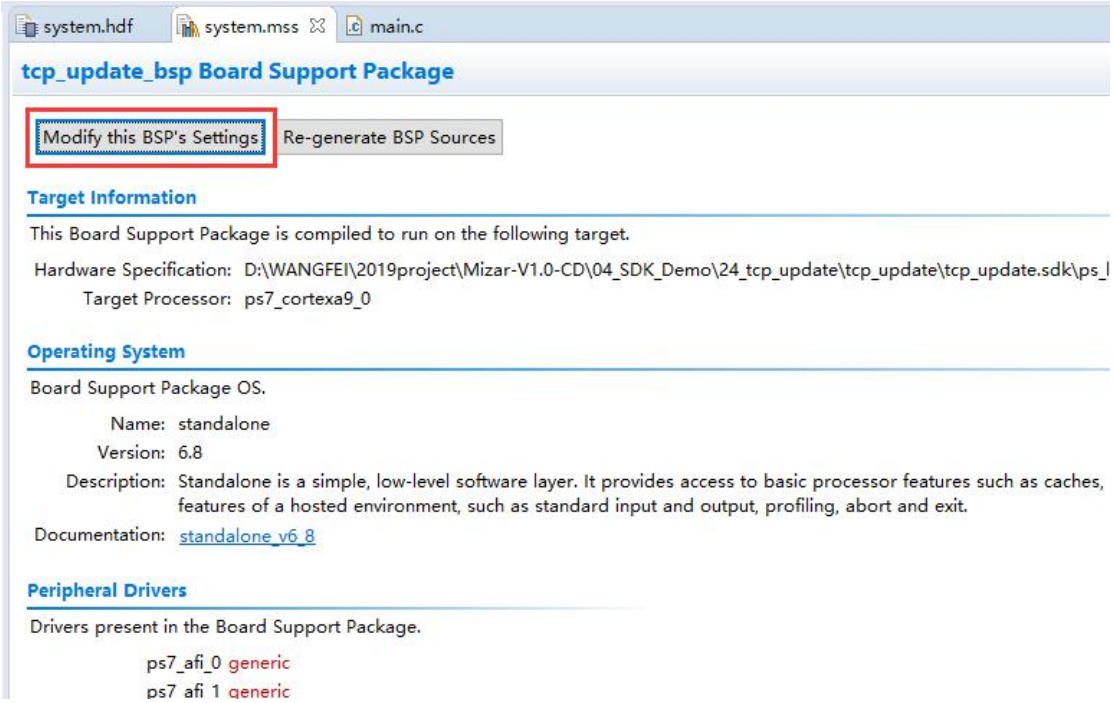
在菜单栏点击 File->New->Application Project 创建一个新的应用工程并命名



为“tcp_update”，工程模板选择空白模板“Empty Application”。

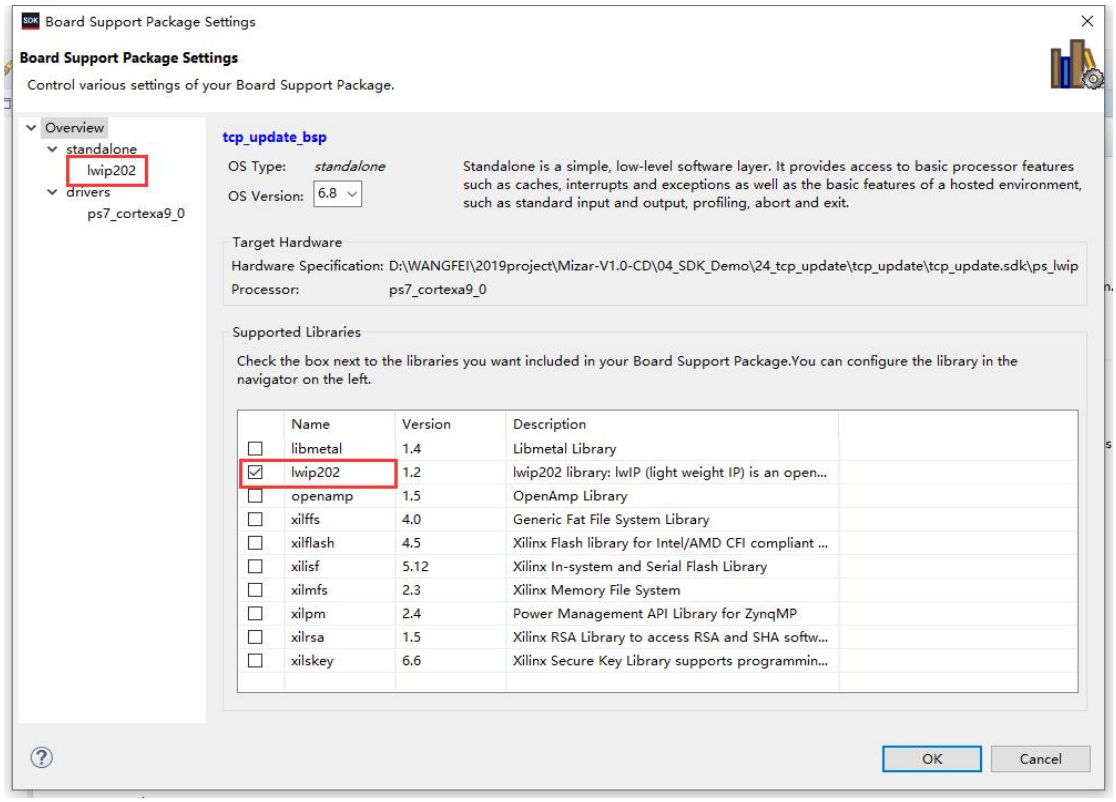
图 24.3.1 创建新应用工程

创建完成后自动打开 system.mss 文件，点击 Modify this BSP’s Settings 进入板



级支持包设置界面。如下图所示：

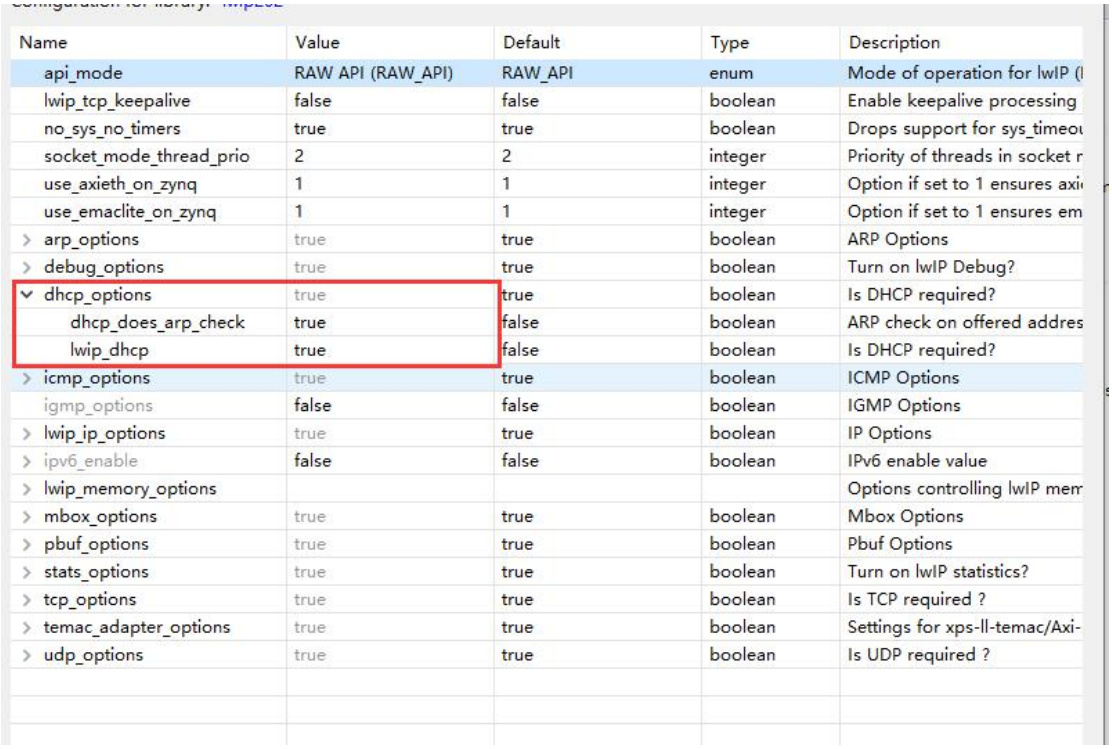
图 24.3.2 进入板级支持包设置界面



勾选 lwip202 添加以太网相关库函数。

图 25.3.3 添加以太网库函数

之后点击 standalone 下的 lwip202 进入参数设置页面。设置 dhcp_options 选项。将 dhcp_does_arp_check 和 lwip_dhcp 的值设置为 true，使能 dhcp 功能。



Name	Value	Default	Type	Description
api_mode	RAW API (RAW_API)	RAW_API	enum	Mode of operation for lwIP (
lwip_tcp_keepalive	false	false	boolean	Enable keepalive processing
no_sys_no_timers	true	true	boolean	Drops support for sys_timeou
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	1	1	integer	Option if set to 1 ensures axi
use_emaclite_on_zynq	1	1	integer	Option if set to 1 ensures em
arp_options	true	true	boolean	ARP Options
debug_options	true	true	boolean	Turn on lwIP Debug?
dhcp_options	true	true	boolean	Is DHCP required?
dhcp_does_arp_check	true	false	boolean	ARP check on offered addres
lwip_dhcp	true	false	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
ipv6_enable	false	false	boolean	IPv6 enable value
lwip_memory_options				Options controlling lwIP mem
mbox_options	true	true	boolean	Mbox Options
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-
udp_options	true	true	boolean	Is UDP required ?

如下图所示：

图 25.3.4 使能 DHCP

设置 lwip_memory_options 选项。将 mem_size 设置为 524288，增加可得到的总的堆空间；将 memp_n_pbuf 设置为 1024， 增加 pbuf 数；将

igmp_options	false	false	boolean	IGMP Options
lwip_ip_options	true	true	boolean	IP Options
ipv6_enable	false	false	boolean	IPv6 enable value
lwip_memory_options				Options controlling lwIP mem
mem_size	524288	131072	integer	Size of the heap memory (byt
memp_n_pbuf	1024	16	integer	Number of memp struct pbuf
memp_n_sys_timeout	8	8	integer	Number of simultaneously ac
memp_n_tcp_pcb	32	32	integer	Number of active TCP PCBs. (
memp_n_tcp_pcb_listen	8	8	integer	Number of listening TCP conn
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_tcpip_msg	64	64	integer	Number of tcpip msg structu
mbox_options	true	true	boolean	Mbox Options
pbuf_options	true	true	boolean	Pbuf Options
stats_options	true	true	boolean	Turn on lwIP statistics?
tcp_options	true	true	boolean	Is TCP required ?

memp_n_tcp_seg 设置为 1024， 提高同时排队的 TCP 段数。 如下图所示：

图 24.3.5 设置 lwip_memory_options 选项

设置 pbuf_options 选项。将 pbuf_pool_size 为 pbuf 池中的缓冲区数量。对于高性能系统，可以将 pbuf 池大小增加到一个较高的值，此处设为 16384，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 poo
pbuf_pool_size	16384	256	integer	Number of buffers in pbuf po

图 24.3.6 设置 pbuf_options 选项

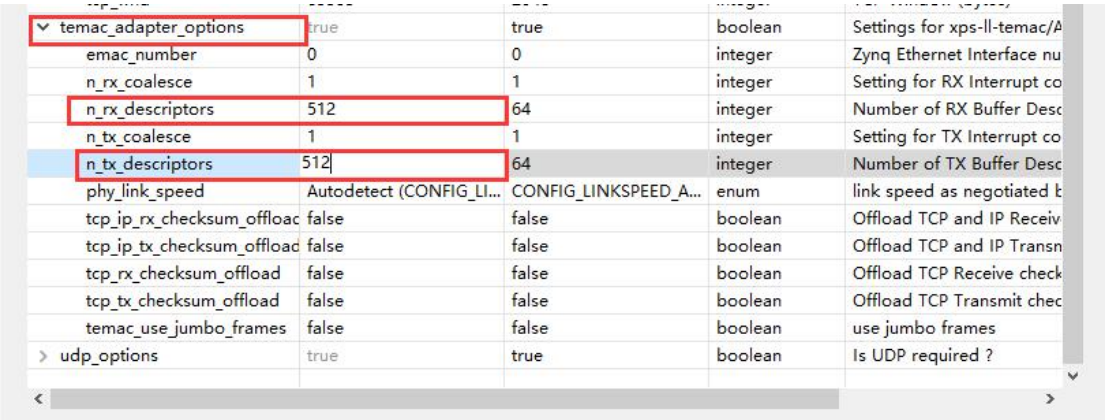
设置 tcp_options 选项，将 tcp_snd_buf 和 tcp_wnd 设为 65535，增大 tcp 发

tcp_options	true	true	boolean	Is TCP required ?
lwip_tcp	true	true	boolean	Is TCP required ?
tcp_maxrtx	12	12	integer	TCP Maximum retransmiss
tcp_mss	1460	1460	integer	TCP Maximum segment si
tcp_queue_ooseq	1	1	integer	Should TCP queue segmer
tcp_snd_buf	65535	8192	integer	TCP sender buffer space (
tcp_synmaxrtx	4	4	integer	TCP Maximum SYN retrans
tcp_ttl	255	255	integer	TCP TTL value
tcp_wnd	65535	2048	integer	TCP Window (bytes)

送缓冲空间和窗口大小，如下图所示：

图 24.3.7 设置 tcp_options 选项

设置 temac_adapter_options 选项，将 n_rx_descriptors 和 n_tx_descriptors 设置为 512，以提高系统性能，如下图所示：

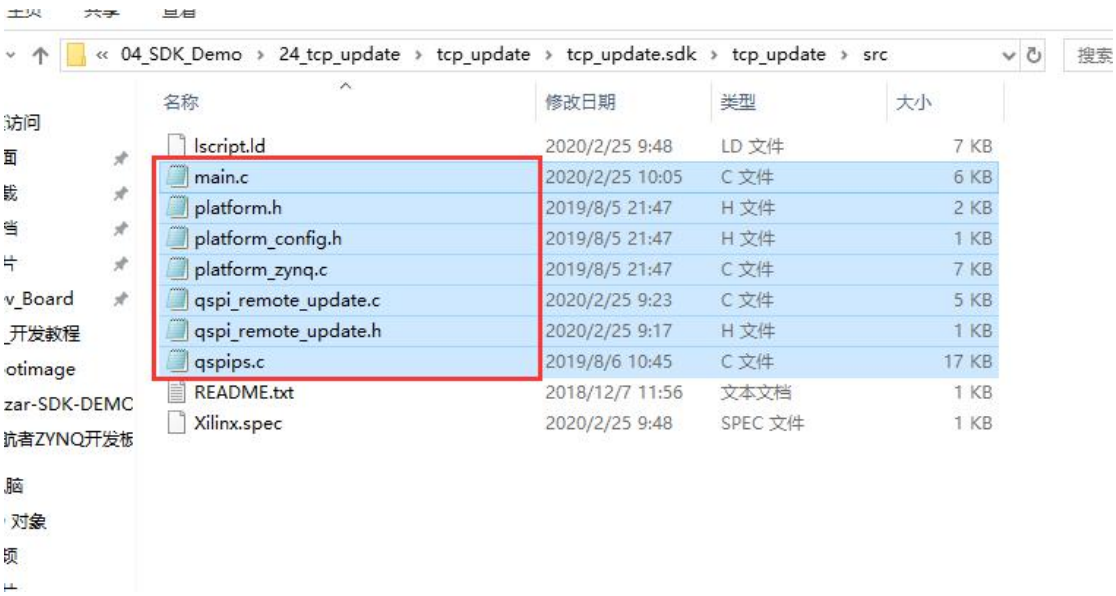


temac_adapter_options	true	true	boolean	Settings for xps-ll-temac/A
emac_number	0	0	integer	Zynq Ethernet Interface nu
n_rx_coalesce	1	1	integer	Setting for RX Interrupt co
n_rx_descriptors	512	64	integer	Number of RX Buffer Desc
n_tx_coalesce	1	1	integer	Setting for TX Interrupt co
n_tx_descriptors	512	64	integer	Number of TX Buffer Desc
phy_link_speed	Autodetect (CONFIG_LI...	CONFIG_LINKSPEED_A...	enum	link speed as negotiated b
tcp_ip_rx_checksum_offloac	false	false	boolean	Offload TCP and IP Receiv
tcp_ip_tx_checksum_offload	false	false	boolean	Offload TCP and IP Transn
tcp_rx_checksum_offload	false	false	boolean	Offload TCP Receive check
tcp_tx_checksum_offload	false	false	boolean	Offload TCP Transmit chec
temac_use_jumbo_frames	false	false	boolean	use jumbo frames
udp_options	true	true	boolean	Is UDP required ?

图 24.3.8 设置 temac_adapter_options 选项

其余选项保持默认，点击 OK 完成设置。

之后打开例程文件，复制以下几个文件，如下图所示：



« 04_SDK_Demo › 24_tcp_update › tcp_update › tcp_update.sdk › tcp_update › src

名称	修改日期	类型	大小
lscript.ld	2020/2/25 9:48	LD 文件	7 KB
main.c	2020/2/25 10:05	C 文件	6 KB
platform.h	2019/8/5 21:47	H 文件	2 KB
platform_config.h	2019/8/5 21:47	H 文件	1 KB
platform_zynq.c	2019/8/5 21:47	C 文件	7 KB
qspi_remote_update.c	2020/2/25 9:23	C 文件	5 KB
qspi_remote_update.h	2020/2/25 9:17	H 文件	1 KB
qspips.c	2019/8/6 10:45	C 文件	17 KB
README.txt	2018/12/7 11:56	文本文档	1 KB
Xilinx.spec	2020/2/25 9:48	SPEC 文件	1 KB

图 24.3.9 工程源文件

将复制的源文件保存到新建 SDK 工程的相同位置下，之后右键点击 tcp_update，在弹出的菜单中点击 Refresh 重新编译工程。如下图所示：

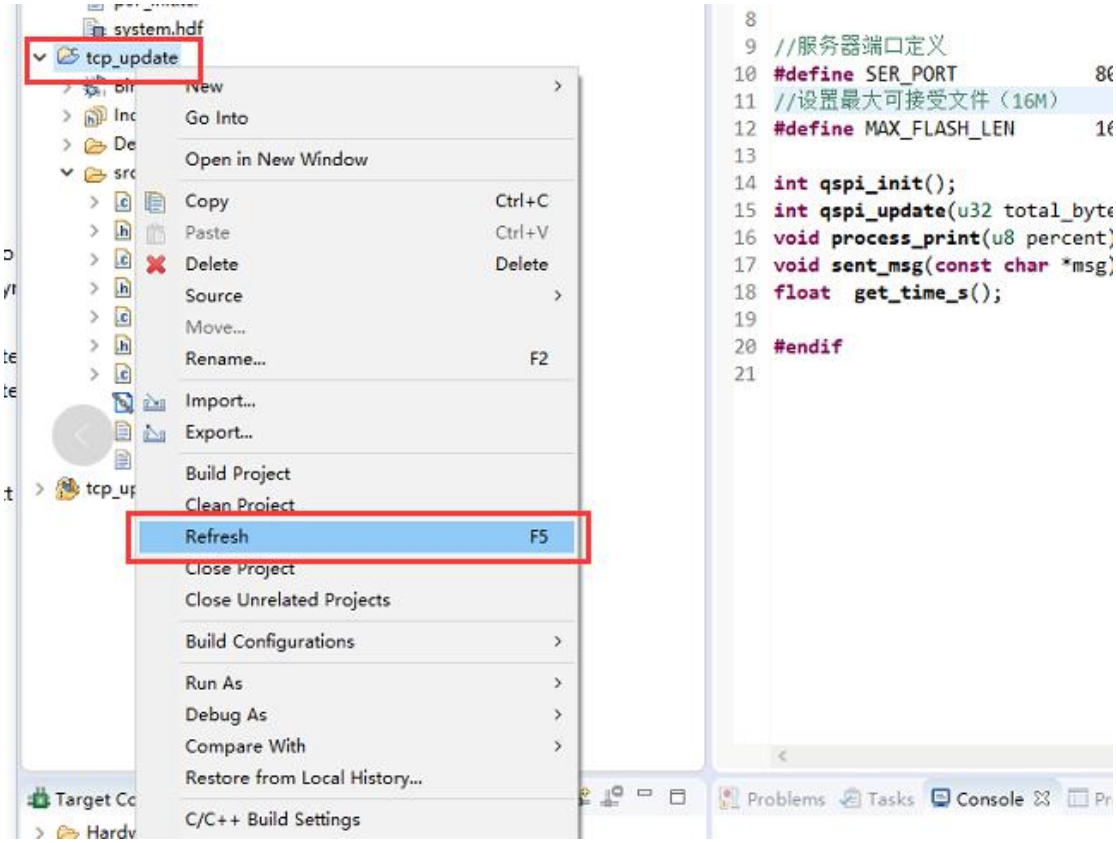


图 24.3.10 重新编译工程

编译完成后即可烧录到开发板上进行验证。

24.4 下载验证

首先我们将使用 Micro USB 线与 JTAG 接口连接，另外一端与电脑连接。然后使用 type-c 线将开发板的 UART 接口与电脑连接，用于串口通信。使用一根网线一端连接开发板以太网接口，另一端与电脑或路由器连接。最后连接开发板的电源，并打开电源开关。

打开 SecureCRE 软件连接开发板串口，之后右键点击 tcp_update->Run As->1 Launch on Hardware (Sysrtem Debugger)将程序烧录到开发板中，如下图所示：

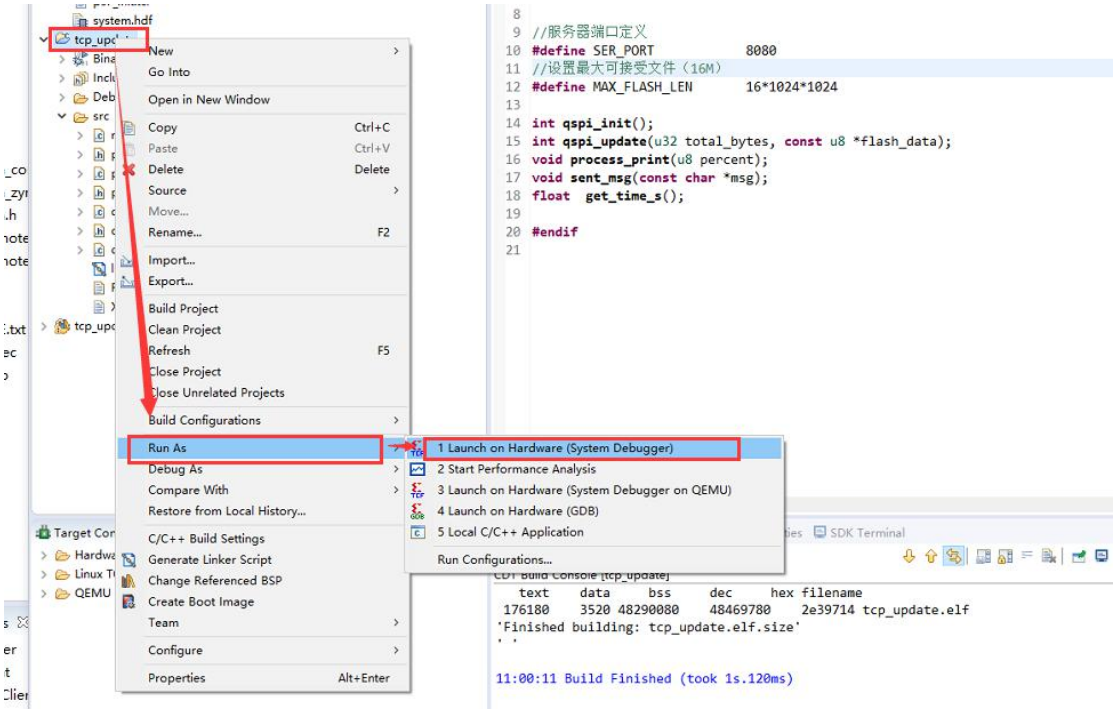


图 24.4.1 烧录程序

烧录完成后可以在 SecureCRT 窗口看到串口打印的信息。如果开发板连接路由器，那么路由器将自动为开发板分配 IP 地址；如果开发板直接连接电脑则设置为默认 IP 地址：192.168.1.10，端口号设置为：8080。打印信息如下图所示，表示 IP 设置完成，QSPI 初始化成功。（每次重新烧录需关闭重新打开电源，否则可能初始化失败）

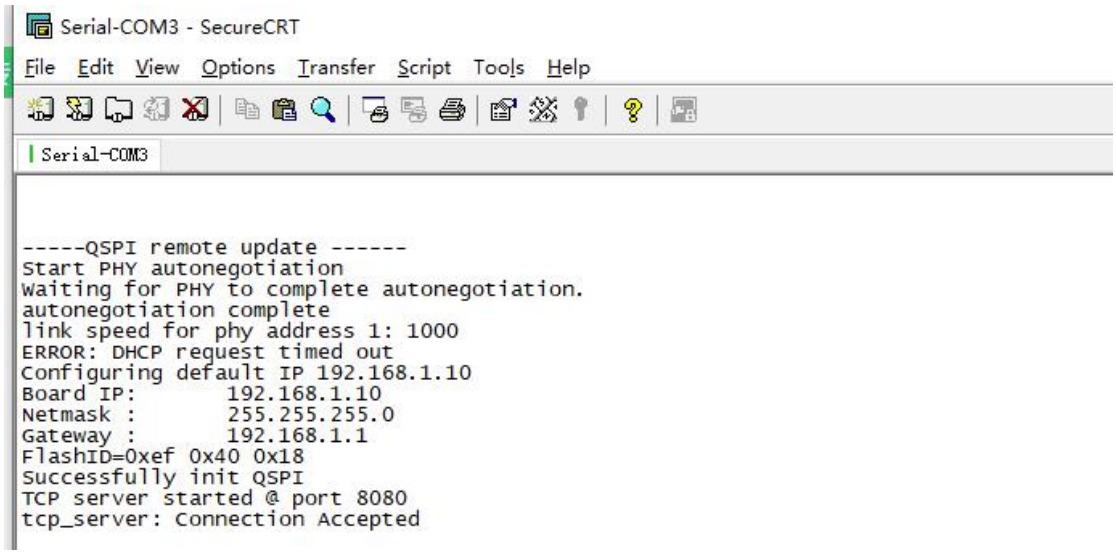
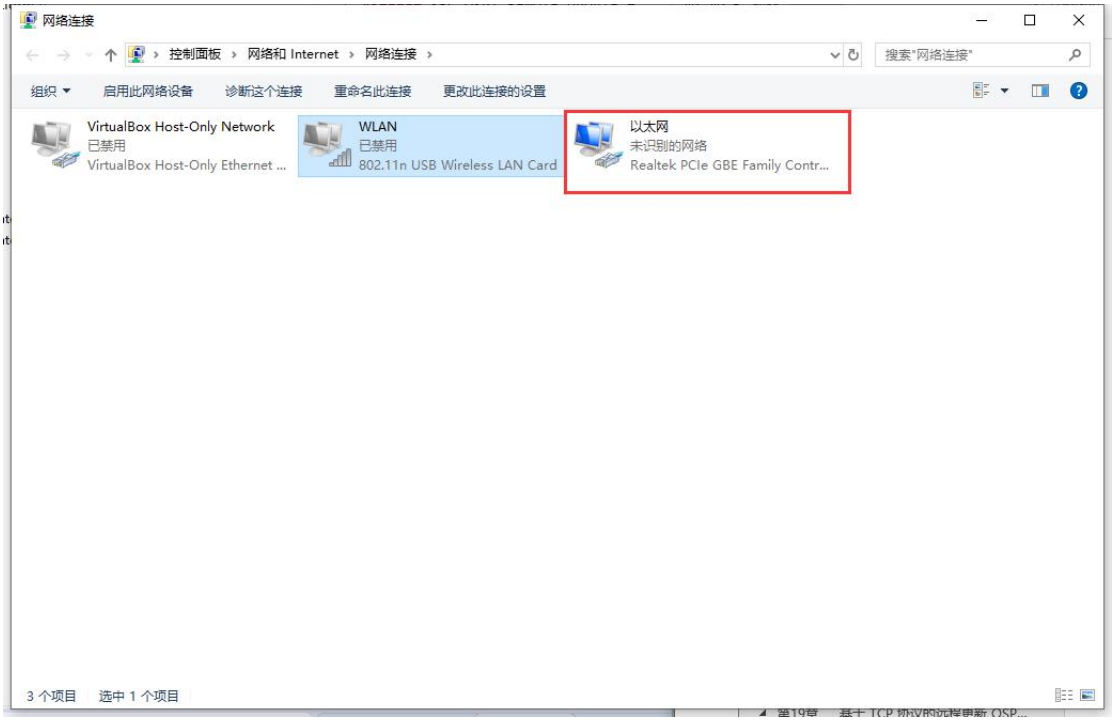


图 24.4.2 初始化设置信息

之后打开电脑控制面板->网络连接可以看到一个未识别的网络，那就是开



发板的以太网，为保证连接，禁用其他网络，如下图所示：

图 24.4.3 网络连接

右键点击该网络打开属性页面，双击 Internet 协议版本 4（TCP/IPv4），如下图所示：

图 24.4.4 属性页面

按照下图所示设置相关设置，确保电脑 IP 和开发板 IP 在同一网段，且 IP 地址不同。

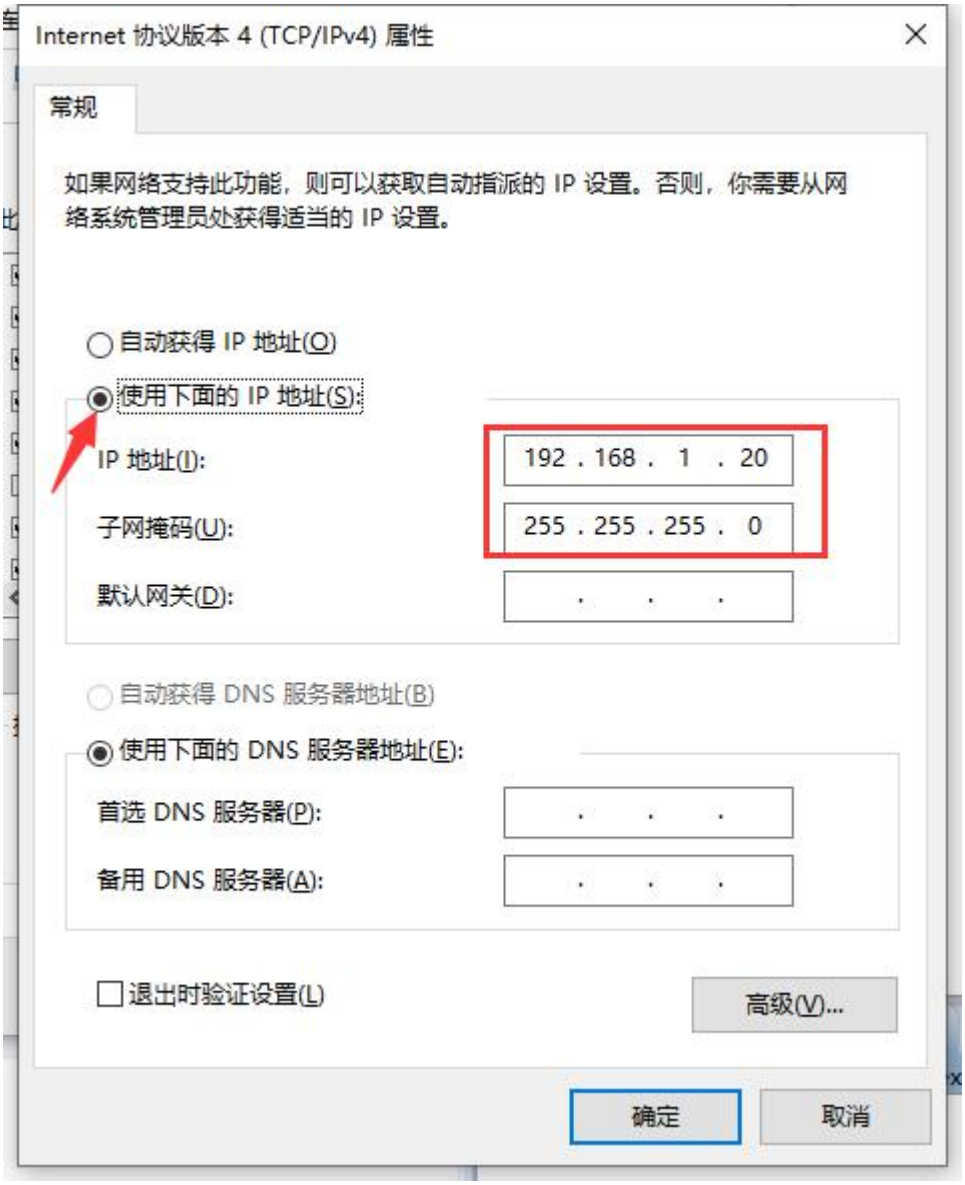


图 24.4.5 设置电脑 IP 地址

打开网络调试助手，如下图所示设置协议类型、服务器 IP 地址、端口号等参数，点击“连接”即可连接开发板。

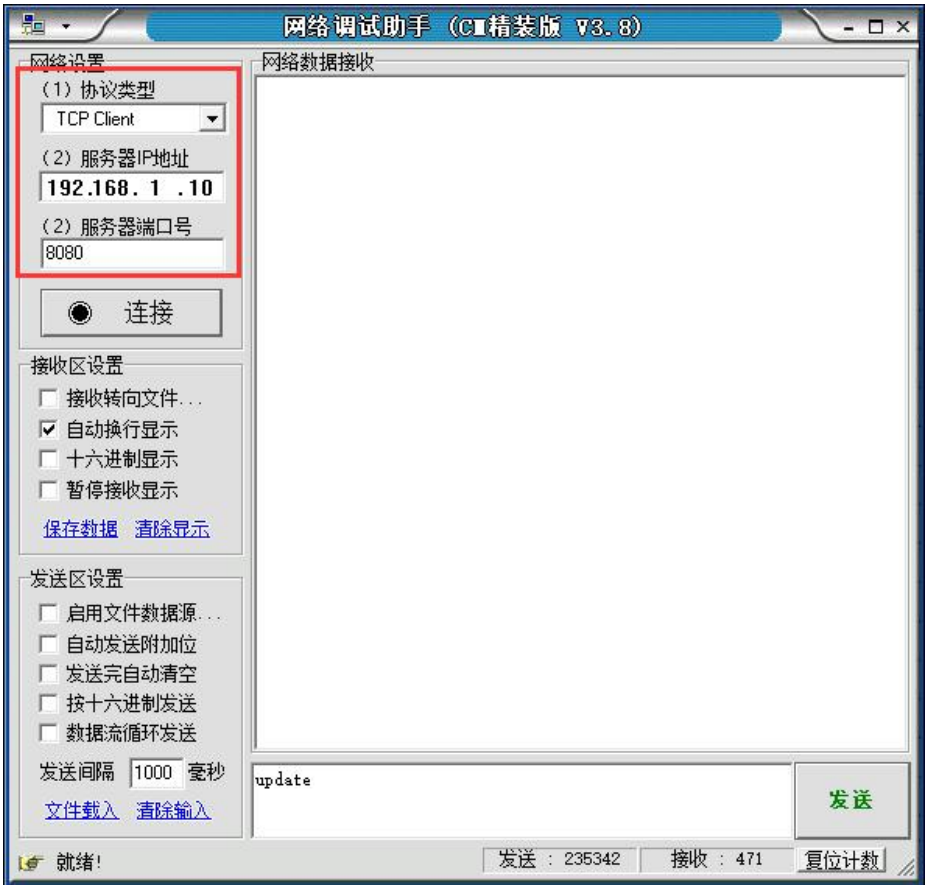


图 24.4.6 设置并连接开发板

连接成功后选择“启用文件数据源”，在弹出的窗口选择需要发送的文件，此处我们选择第 10 章生成的 BOOT.bin 文件，然后点击发送，如下图所示：

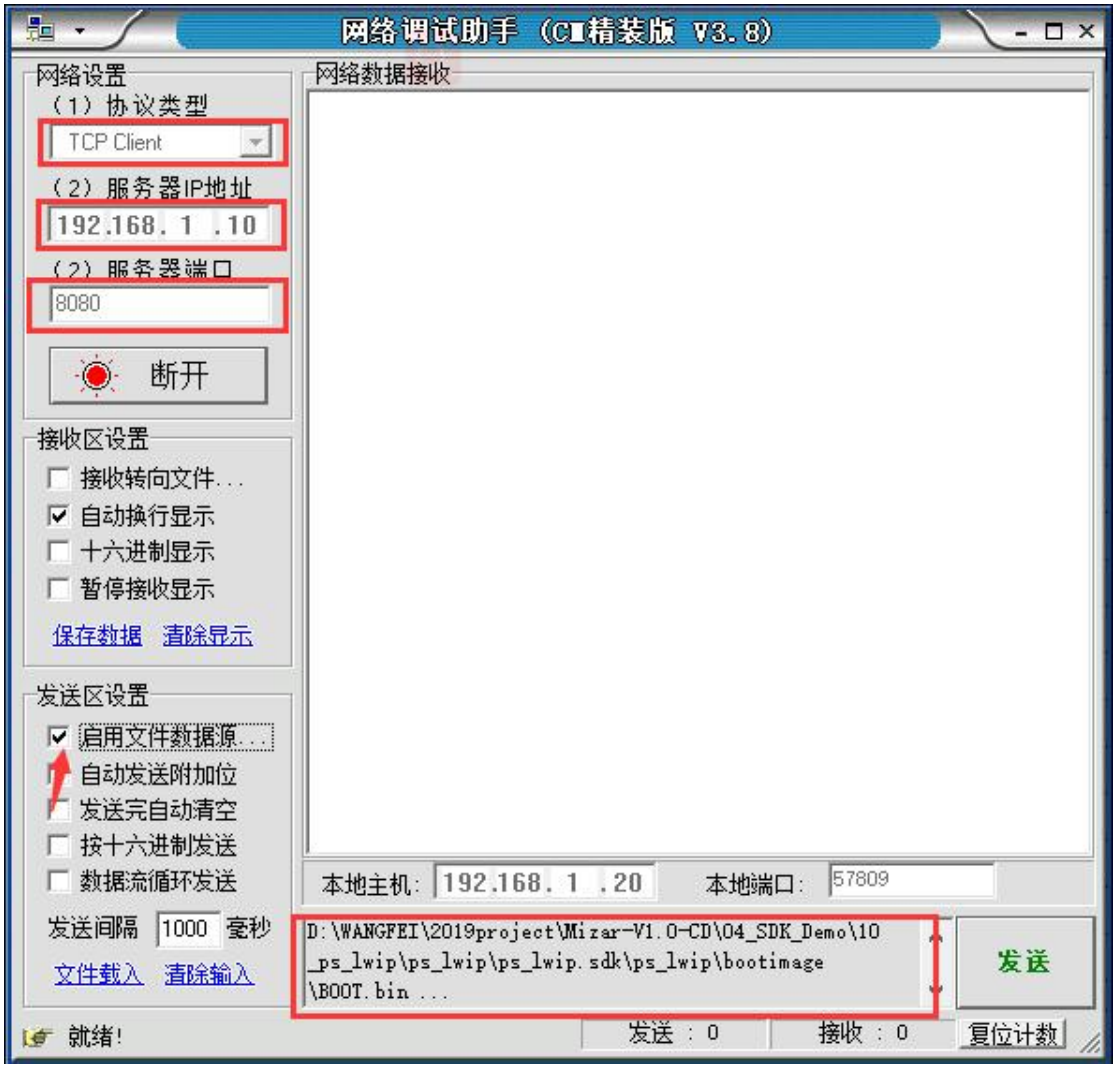


图 24.4.7 发送 BOOT.bin 文件

发送完成后点击取消“启用文件数据源”，软件自动清除发送区，之后在发送区输入“update”，点击发送，如下图所示：

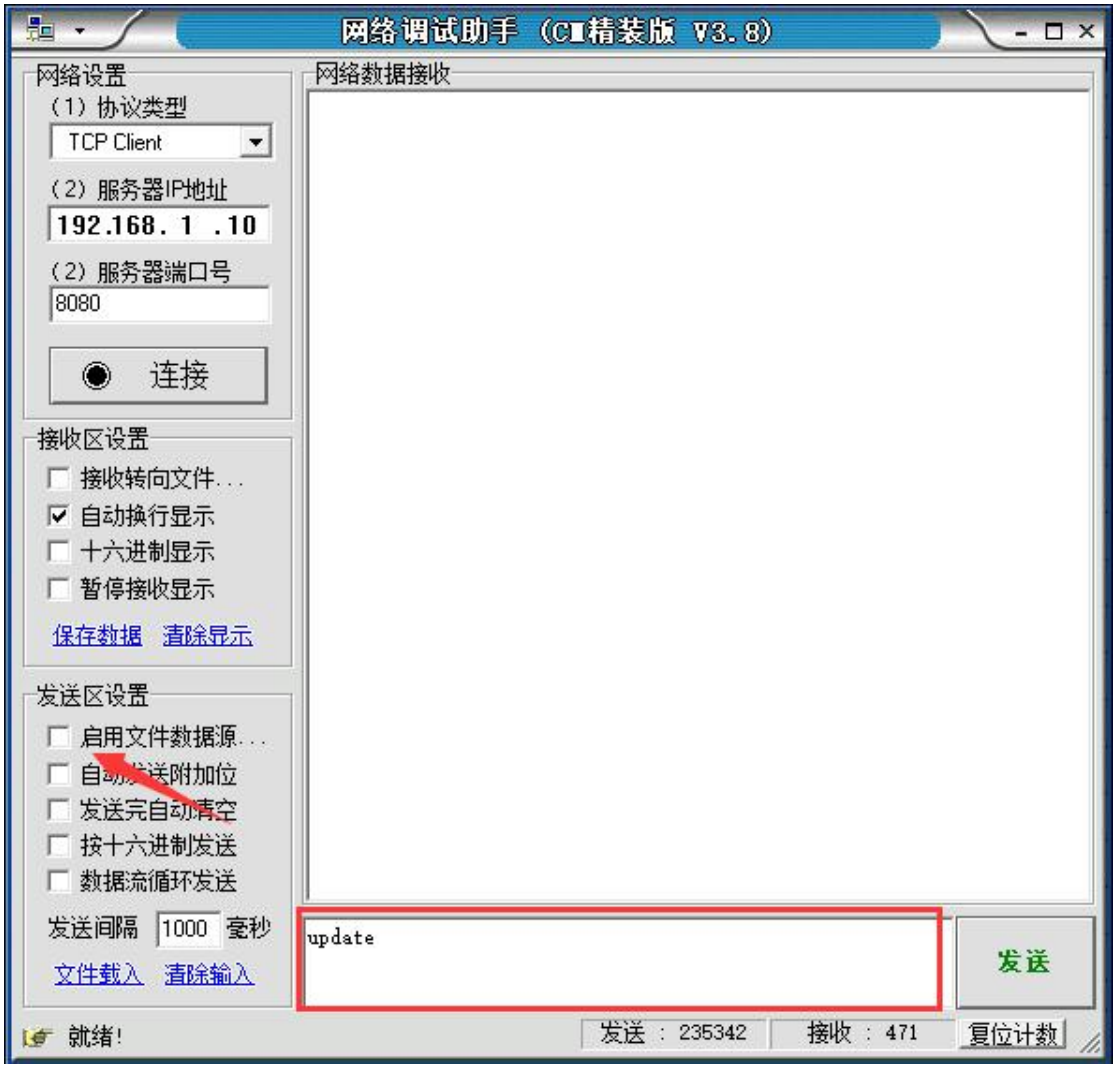


图 24.4.8 发送 update 命令

开发板收到命令后启动 QSPI 更新，并将更新的实时进度信息发送回发送方，我们在网络助手中可以看到 QSPI 更新的进度信息，如下图所示。从这些信息中我们可以得知传送过程中有没有丢包，更新进度信息中的 Elapsed time 表明每个操作（擦除、写入、校验）所花费的时间。

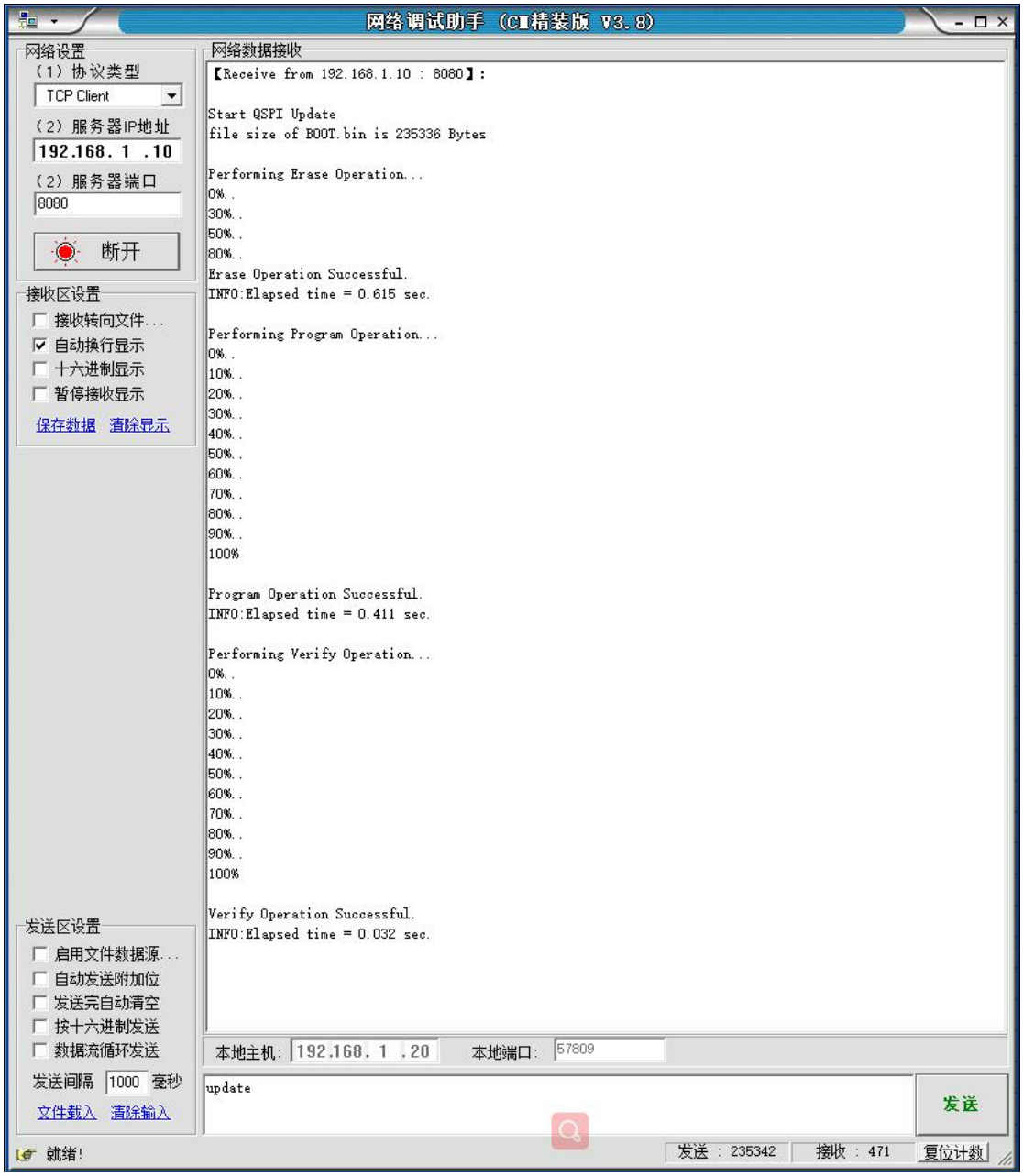


图 24.4.9 反馈更新进度信息

同时在开发板也会通过串口输出更新信息，如下图所示：

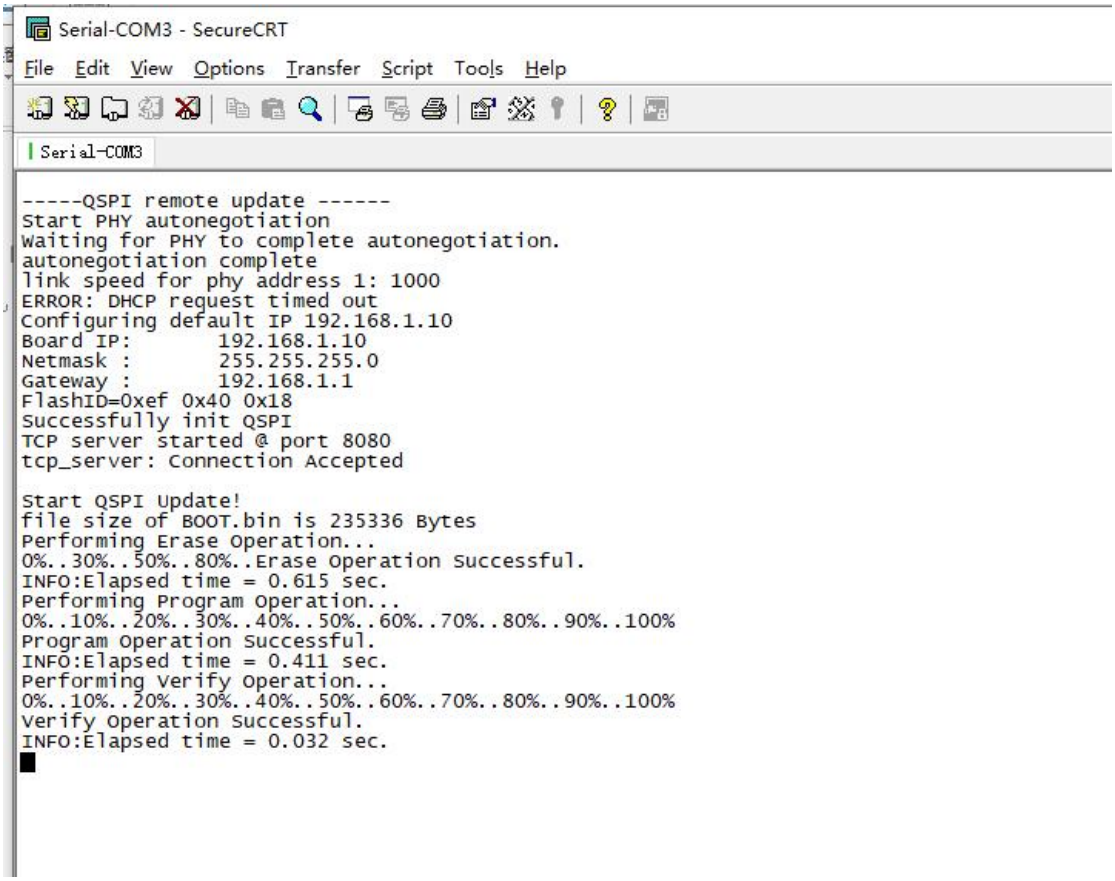
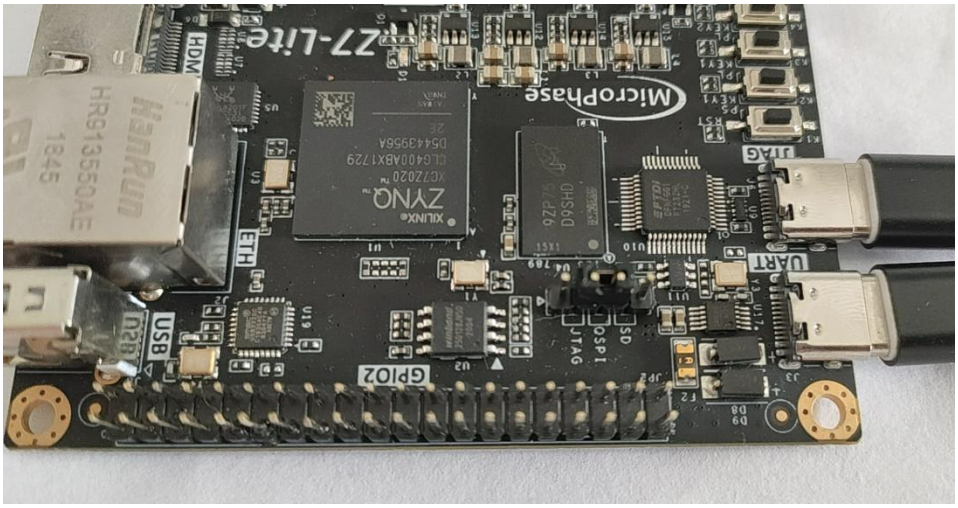


图 24.4.10 串口打印更新信息

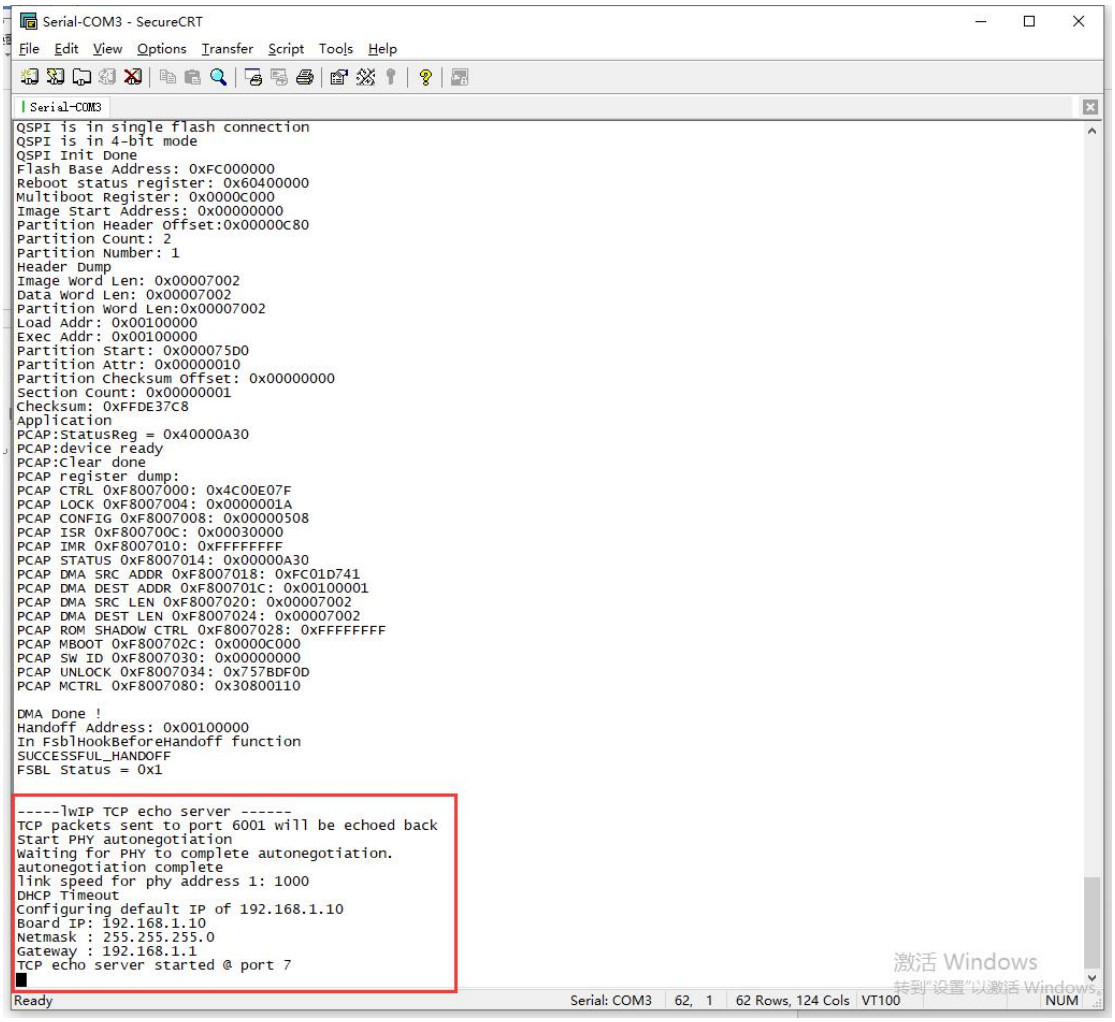
更新完成后，关闭开发板电源。调整跳线帽位置，将跳线帽插入中间两根



排针，设置启动方式为 QSPI 启动。

图 24.4.11 设置 QSPI Flash 启动

之后重新连接 JTAG 线对开发板供电，可以在 SecureCRT 窗口看到打印信息



如下图所示，说明远程更新 QSPI 成功。

图 24.4.12 QSPI 启动

24.5 章节小结

在本实验中，我们实现了通过网络远程更新 QSPI 的方式进行程序固化。相当于之前的以太网实验的拓展，读者可以对照之前的实验，比较程序上的差异，更加深入的了解以太网的相关知识。

第 25 章　　基于 UDP 协议的远程更新 QSPI Flash 实验

上一个实验中，我们实现了利用 TCP 协议远程更新 QSPI 的功能。在实际的应用中使用的网络协议不止一种，所以在本次实验中，我们将利用 UDP 协议实现相同的功能。

本章包括以下几部分内容：

25.1 实验简介

25.2 硬件设计

25.3 软件设计

25.4 下载验证

25.5 章节小结

25.1 实验简介

相较于 TCP 协议，UDP 协议的可靠性性较差，在网络性能较好时，能够保障数据的完整性，不丢失数据，而网络性能较差时就会面临数据丢失的问题。所以 UDP 协议不适合进行文件数据的传输，它更适合于传输音视频这种对实时性要求高而对数据完整性要求不高的应用场合。或者在 UDP 协议的基础上增加应用层协议来提高它的可靠性。

本实验中只是将 TCP 协议改为 UDP 协议来实现相同的功能，目的是在于让读者了解 UDP 协议的使用。在实际使用 UDP 协议来实现远程更新功能时需要增加保障数据传输可靠性的机制，才能实现性能要求。

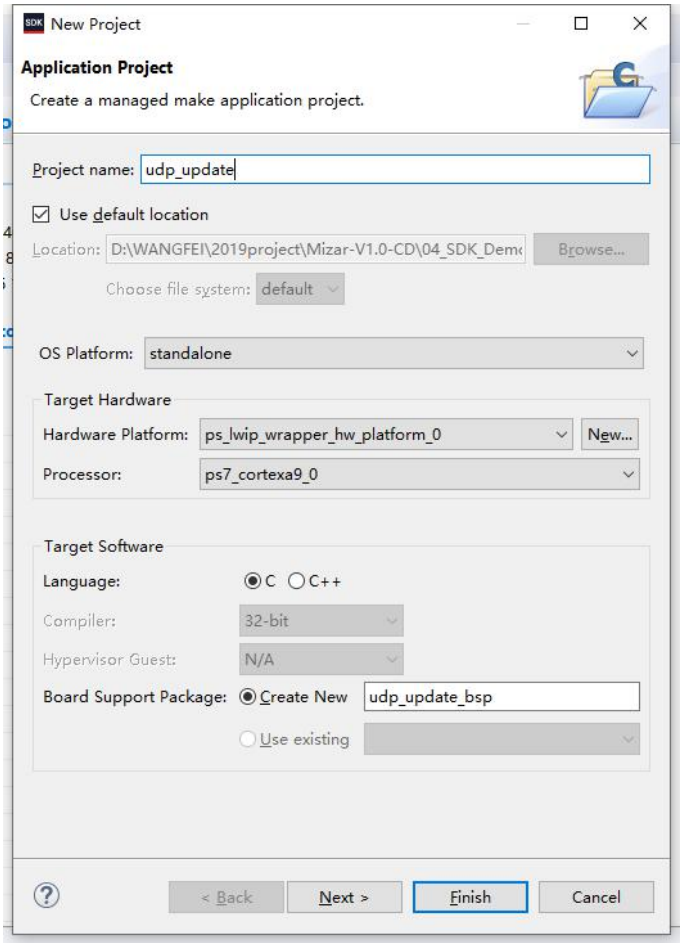
25.2 硬件设计

本次实验的硬件设计与《基于 TCP 协议的远程更新 QSPI Flash 实验》相同。只需打开之前的工程另存为新的工程即可。

保存完成后删除上一实验的.sdk 文件夹，导出硬件描述文件并打开 SDK。

25.3 软件设计

在打开的 SDK 软件中创建一个新的应用工程并命名为“udp_update”，如图 25.3.1 所示，之后在模板选择界面，选择“lwip UDP Perf Server”模板，如图 25.3.2 所示。在之前的实验中，为提高数据传送效率，我们设置了一些 lwip 的相关参数。这些参数就是来自于官方模板，这是官方优化好的参数。在实际的开发过要善于利用这些模板里提高开发的效率。



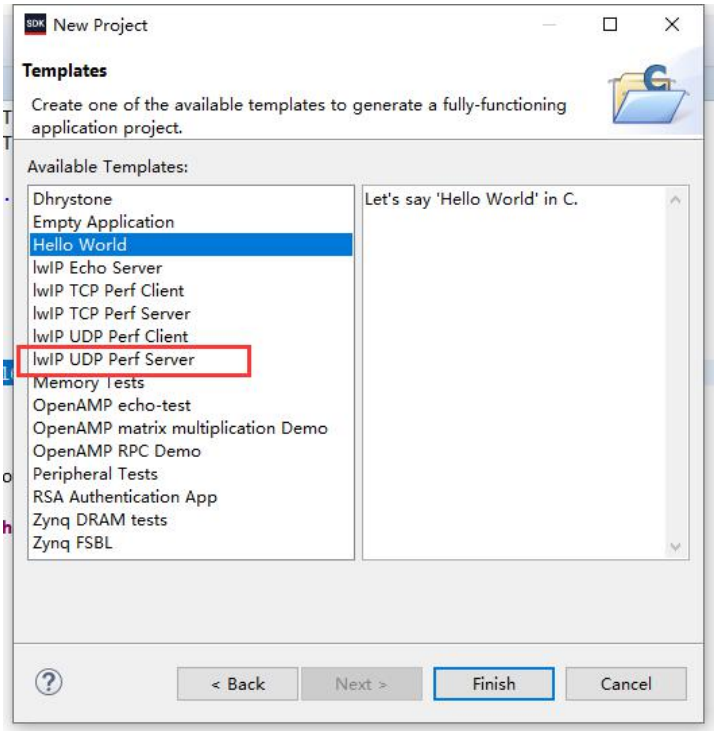


图 25.3.1 创建新应用工程

图 25.3.2 选择“lwip UDP Perf Server”模板

工程创建完成后，删除应用工程“upd_update”的 src 目录下所有的.c 和.h 文件，只保留 lscript.ld、README.txt 和 Xilinx.spec 三个文件即可。之后打开提供的例程文件，复制相关的源文件到 src 文件夹中，需要复制的文件如下图所示：

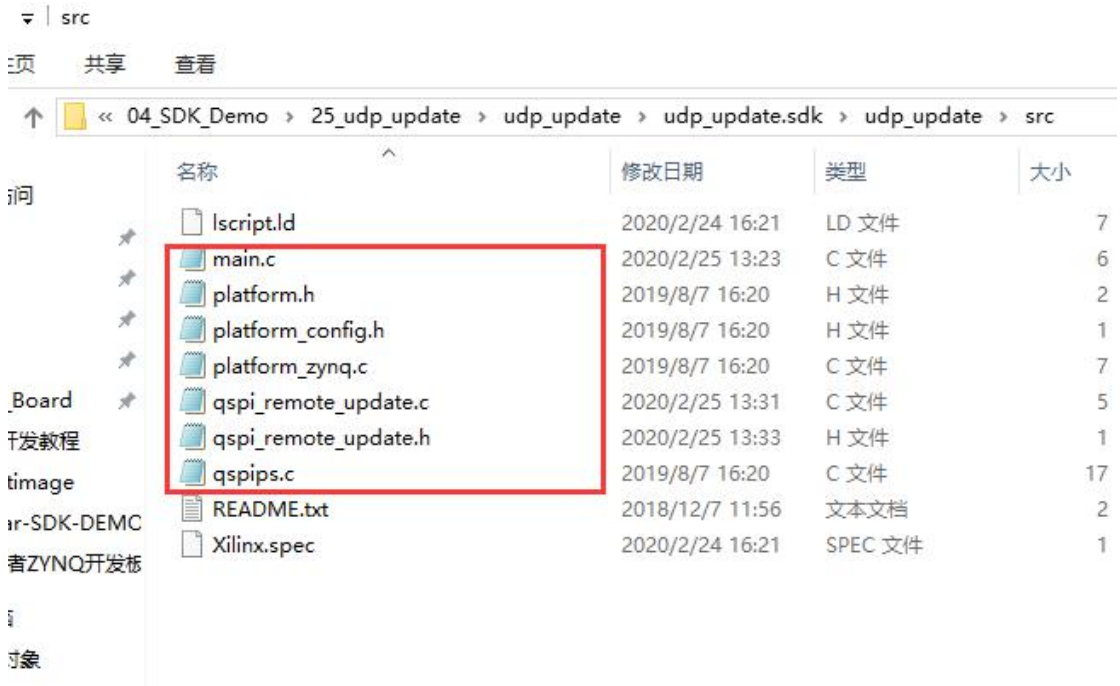


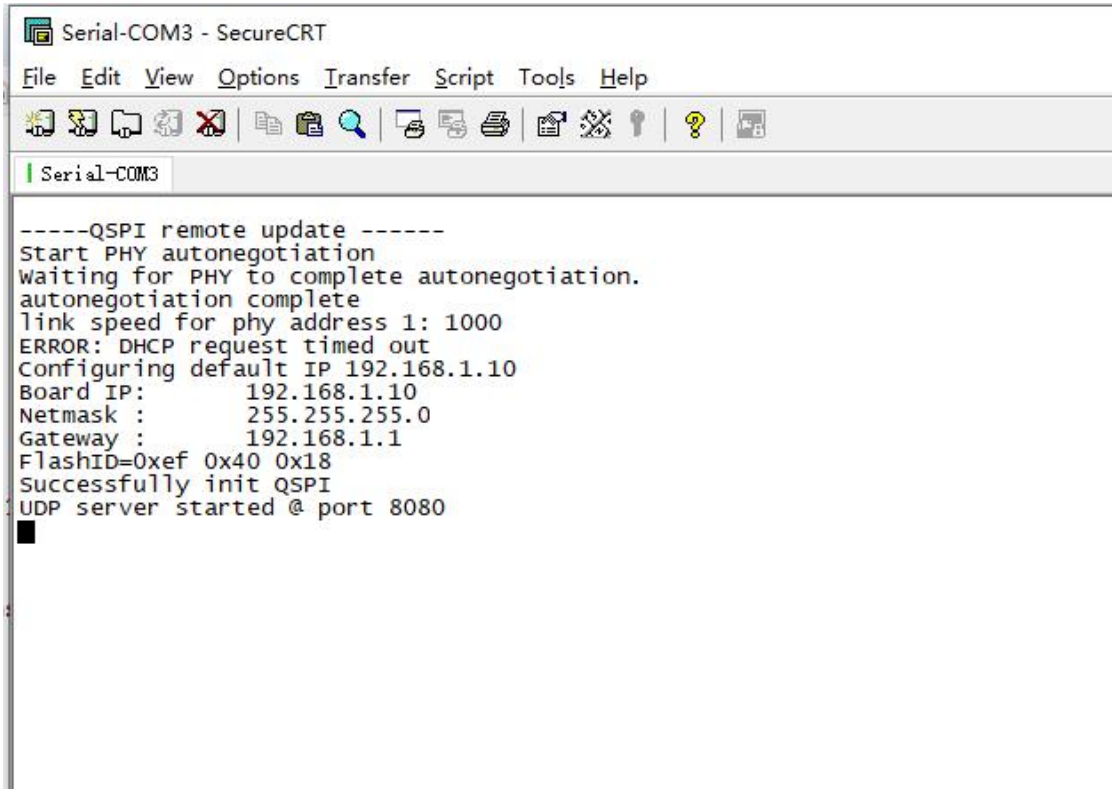
图 25.3.3 应用程序源文件

复制完成后在 SDK 软件中右键点击 udp_update，之后点击 Refresh 重新编译工程。编译完成后即可下载到开发板上进行验证。

25.4 下载验证

首先我们将使用 type-c 线与 JTAG 接口连接，另外一端与电脑连接。然后使用 type-c 线将开发板的 UART 接口与电脑连接，用于串口通信。使用一根网线一端连接开发板以太网接口，另一端与电脑或路由器连接。打开 SecureCRE 软件连接开发板串口，之后右键点击 udp_update->Run As->1 Launch on Hardware (Sysrtem Debugger)将程序烧录到开发板中。

烧录完成后可以在 SecureCRT 窗口看到串口打印的信息。如果开发板连接路由器，那么路由器将自动为开发板分配 IP 地址；如果开发板直接连接电脑则设置为默认 IP 地址：192.168.1.10，端口号设置为：8080。打印信息如下图所示，表示 IP 设置完成，QSPI 初始化成功。（每次重新烧录需关闭重新打开电源，



否则可能初始化失败）

图 25.4.1 初始化设置信息

按照之前实验设置电脑以太网 IP 地址等相关设置，之后打开网络调试助手，按照下图所示设置相关参数，设置完成后点击“连接”。

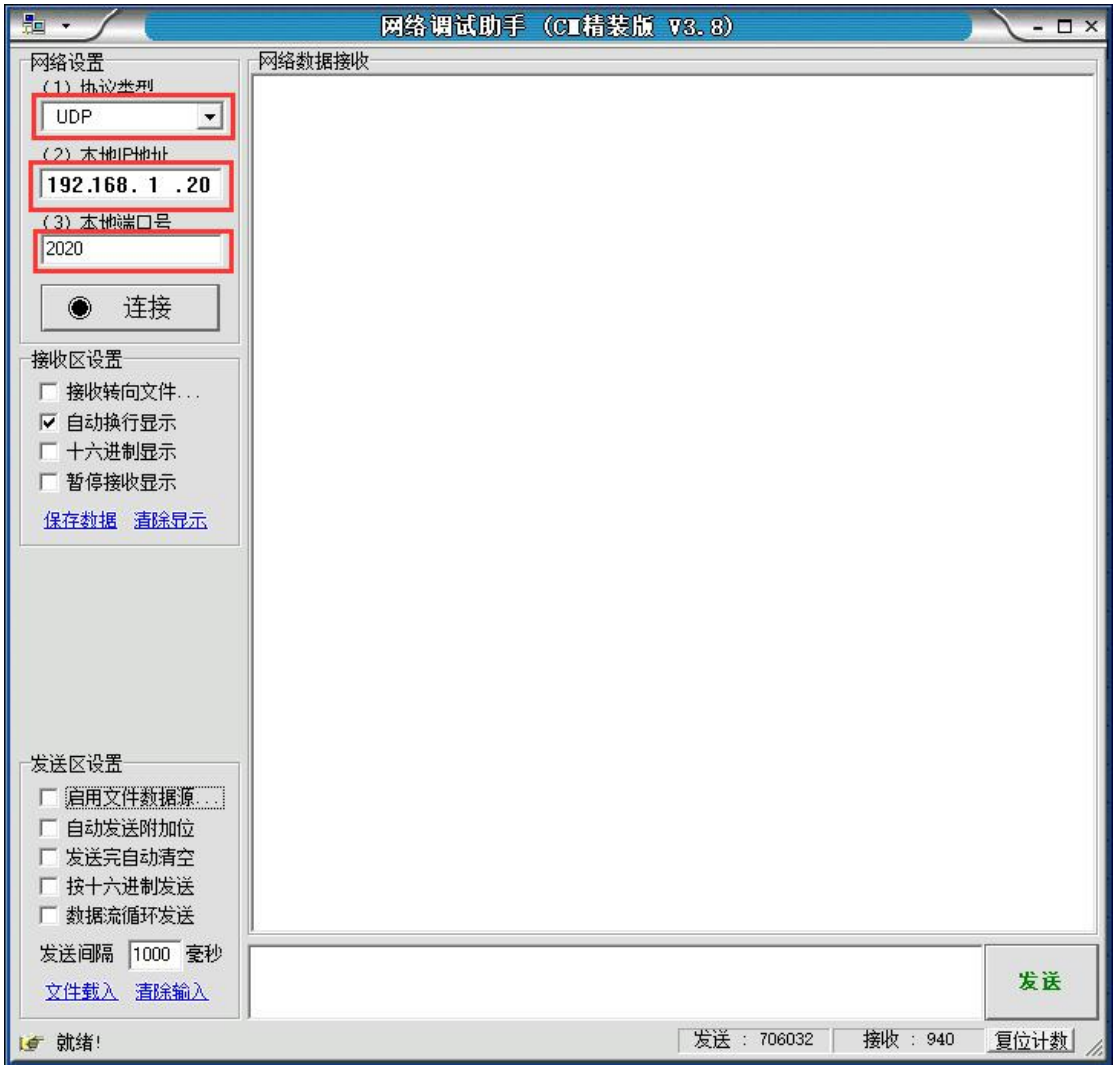


图 25.4.2 设置相关连接参数

连接成功后如下图所示，在发送区上方输入目标主机的 IP 地址和端口号，即可向目标主机发送数据。

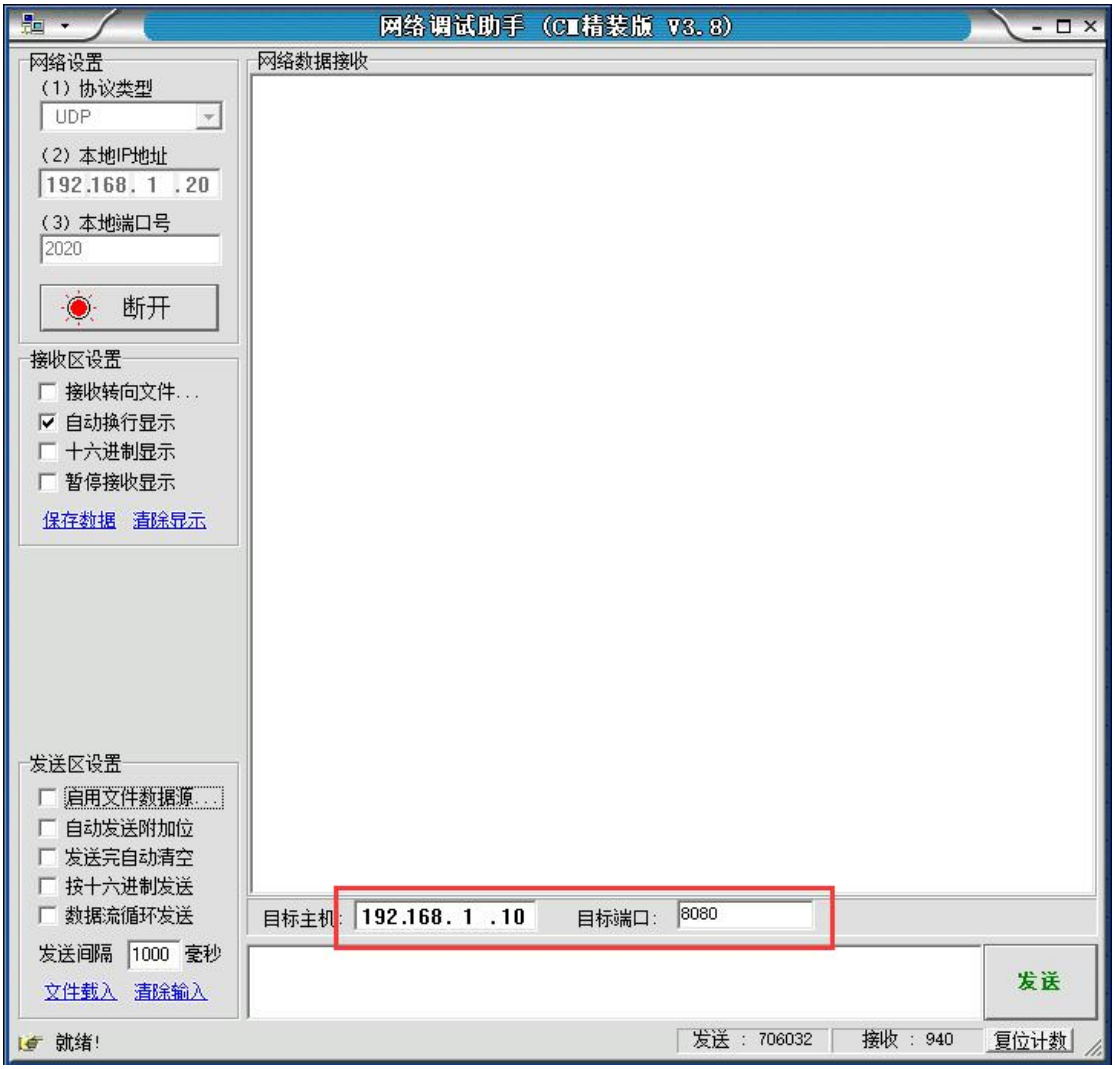


图 25.4.3 设置目标主机

之后点击“启用文件数据源”，选择要发送的 BOOT.bin 文件，如下图所示：

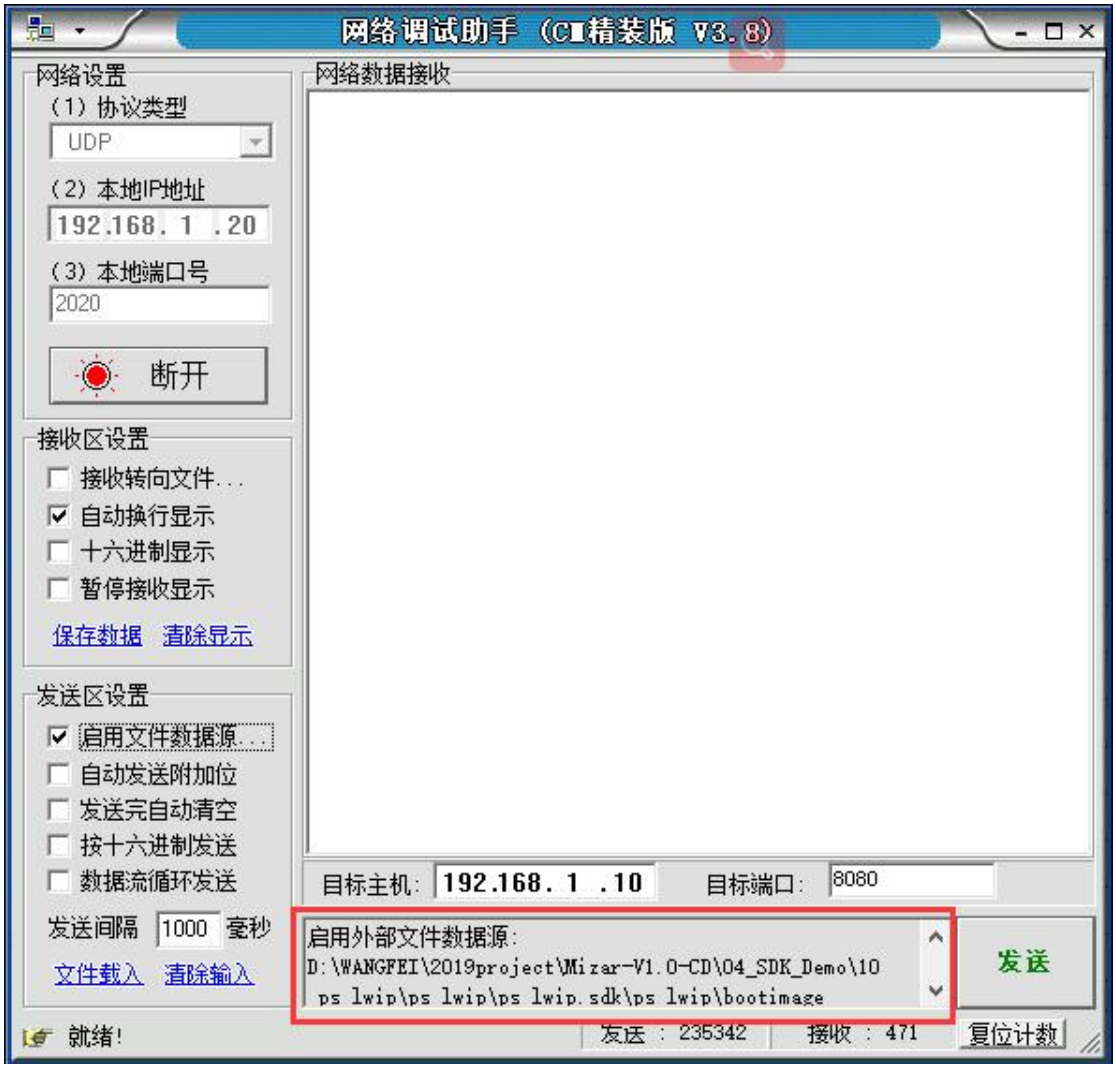


图 25.4.4 发送 BOOT.bin 文件

发送完成后取消“启用文件数据源”，清空发送区，输入“update”，点击发送。向目标主机发送更新命令。在接收到命令后开发板向 QSPI 写入数据并进行校验，同时向发送端返回更新进度数据如下图所示：

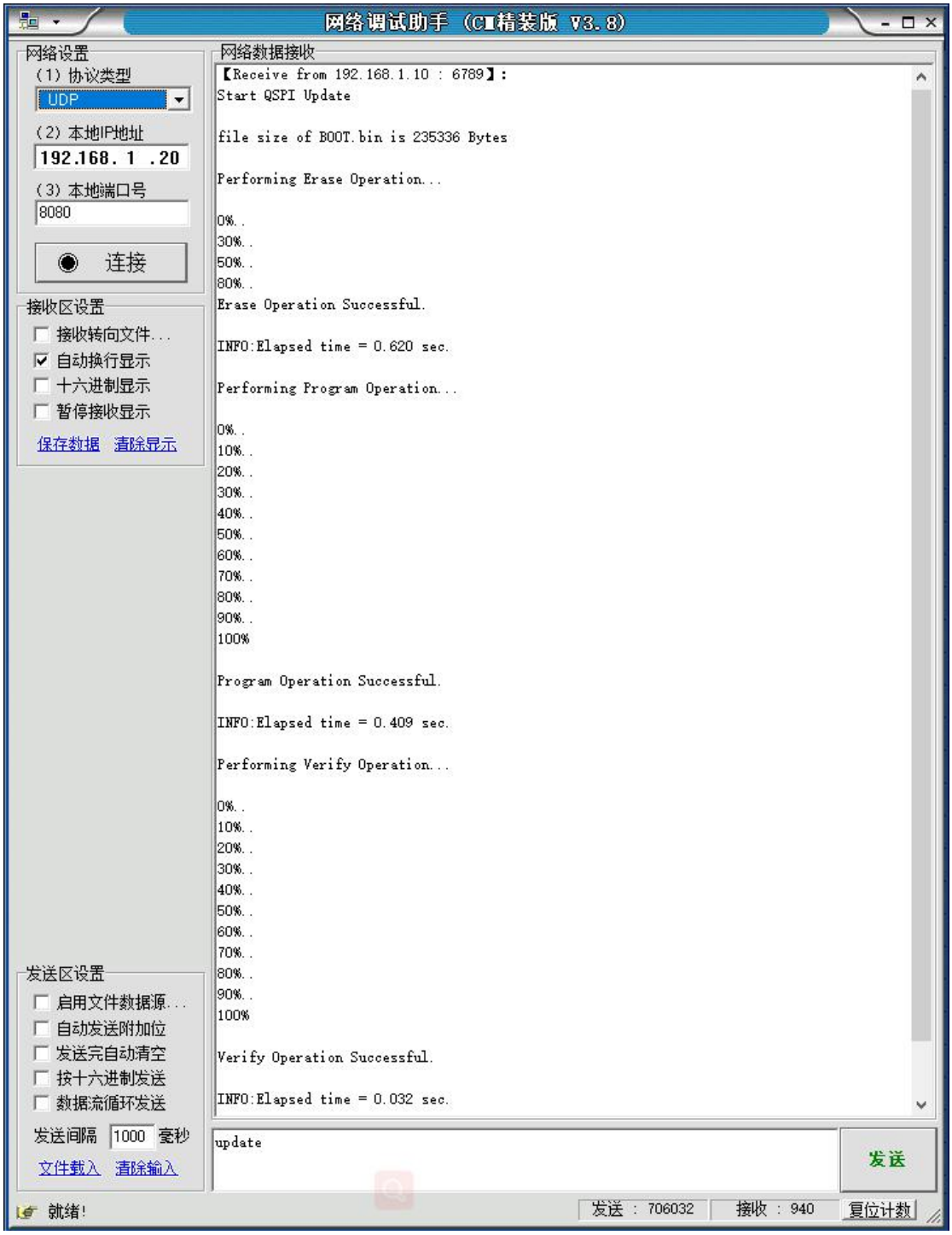


图 25.4.5 返回更新进度数据

同时开发板通过串口打印出更新进度数据，可以通过 SecureCRT 窗口查看，如下图所示：

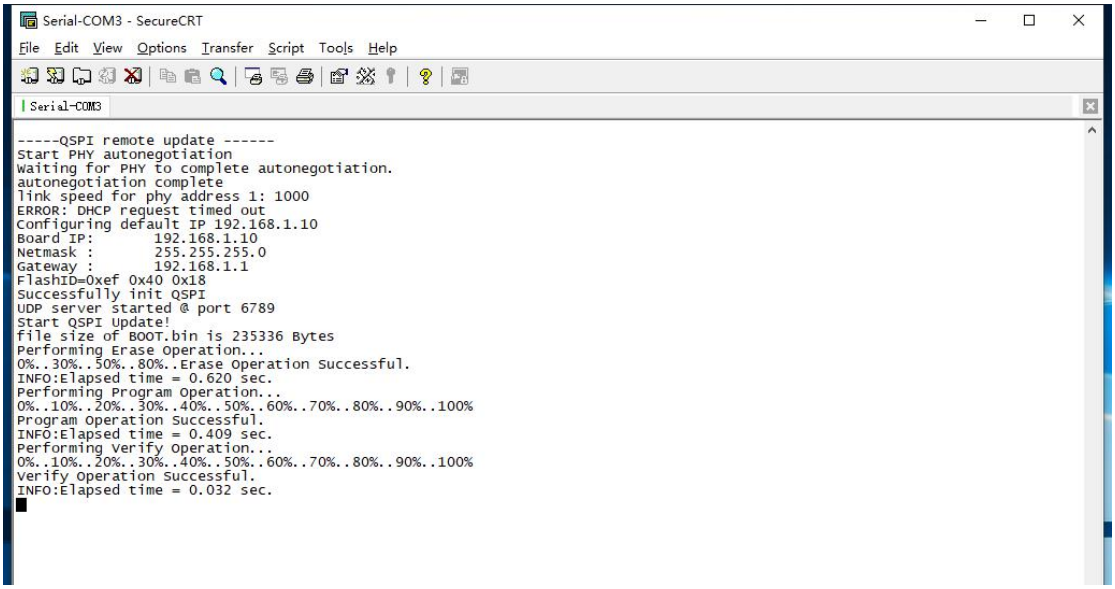
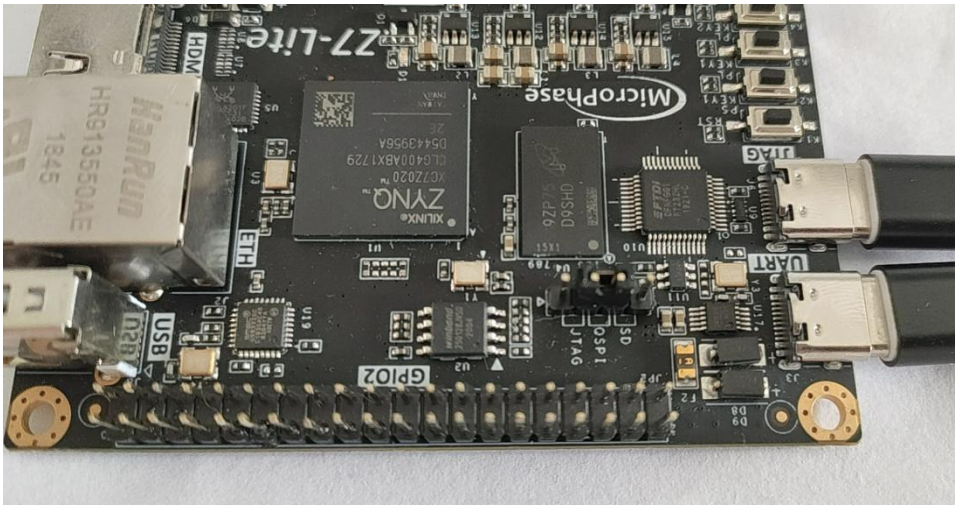


图 25.4.6 串口打印更新信息

更新完成后，断开 JTAG 连接线。更新完成后，关闭开发板电源。调整跳线



帽位置，将跳线帽插入中间两根排针，设置启动方式为 QSPI 启动。如下图所示：

图 25.4.7 设置 QSPI Flash 启动

之后重新打开电源，可以在 SecureCRT 窗口看到打印信息如下图所示，说明远程更新 QSPI 成功。

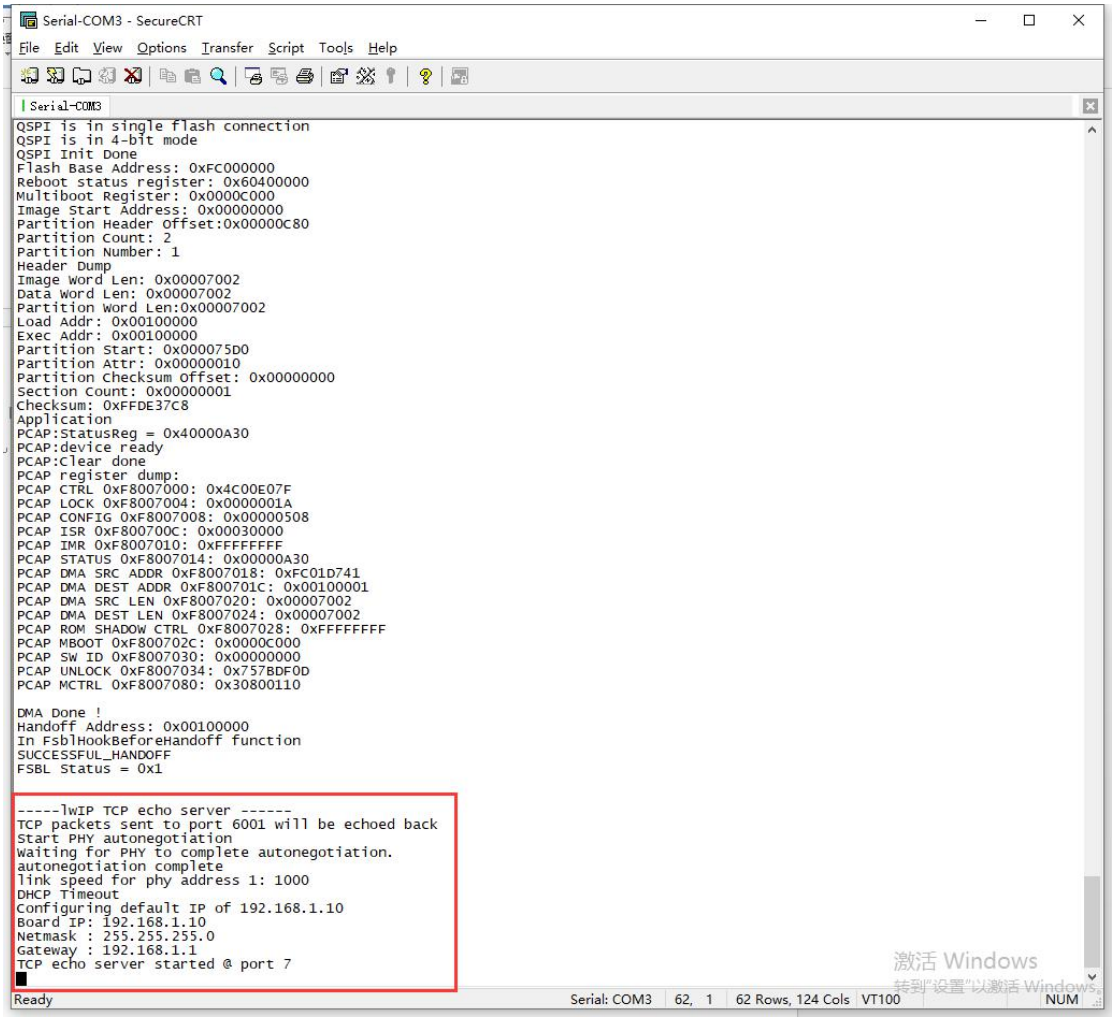


图 25.4.8 QSPI 启动

25.5 章节小结

在本实验中我们利用 UDP 协议实现了远程更新 QSPI Flash 的功能，旨在提高读者对于网络协议的理解，方便在以后的工作中熟练应用相关的知识。

第一版本小结

感谢您选择微相科技（上海）有限公司 Mizar 系列产品，通过前面的内容，相信您对我们的开发板已经比较熟悉了，也能够熟练的运用起板上的资源。由于本人水平有限，不能保证文中的内容全都正确，若您发现了其中的错误，请您与我们联系，我们会尽快更改。同时由于本人水平很低，暂时能够提供的例程还十分有限，但请给予我们多一点时间，我们会努力丰富和完善本开发板的相关资源。

本文档的内容，作为我的初次编写的内容，还请大家包涵，本文档会随着时间，逐步丰富，还请您继续关注。

MircoPhase