

版本信息：
版本
REV2018
时间
04/12/2017

ZYNQ 修炼秘籍

基于米联客系列开发板

第三季 基于 ZYNQ 的裸机应用开发

电子版自学资料

常州一二三电子科技有限公司
溧阳米联电子科技有限公司
版权所有

米联客学院 03QQ 群： 516869816
米联客学院 03QQ 群： 543731097
米联客学院 02QQ 群： 86730608
米联客学院 01QQ 群： 34215299

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

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

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

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

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

版权声明：

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

技术支持：

版主大神们都等着大家去提问--电子资源论坛 www.osrc.cn

微信公众平台：电子资源论坛



目录

ZYNQ 修炼秘籍	1
目录	4
CH01_AXI_DMA_LOOP 环路测试	12
1.1 概述	12
1.2 搭建硬件系统	12
1.2.1 新建 VIVADO 工程	12
1.2.2 创建 VIVADO 硬件构架	14
1.3 PS 部分软件分析	23
1.3.1 新建 SDK 工程	23
1.3.2 main.c 源码的分析	24
1.3.3 dma_intr.c 源码分析	27
1.3.4 dam_intr.h 文件分析	32
1.4 测试结果	34
CH02_AXI_DMA PL 发送数据到 PS	39
2.1 概述	39
2.2 系统构架框图	39
2.2.1 ZYNQ IP 的设置	39
2.3 PS 部分	42
2.4 测试结果	44
CH03_AXI_DMA_OV7725 摄像头采集系统	46
3.1 概述	46
3.2 系统构架	46
3.2.1 构架方案图	46
3.2.2 构 BLOCK 模块化设计方案图	47
3.3 IIC 接口	47
3.4 vid in IP 介绍	48
3.4.1 OV_Sensor_ML 自定义 IP 模块	48
3.4.2 vid in IP 模块	56
3.4.2 VID_IN IP 接口信号的定义	57
3.5 VTC IP 的分析	63
3.5.1 VTC IP 的参数介绍	63
3.5.2 VTC IP 接口信号的定义	65
3.5.3 VTC IP 配置寄存器	69
3.5.4 设置 VTC IP	76
3.6 PLL 时钟设置	77
3.7 VID_OUT IP 的分析	78
3.7.1 VID_OUT 的参数介绍	78
3.7.2 VID_OUT IP 接口信号的定义	79
3.8 FPGA 实现的用户逻辑代码	82

3.8.1 关键信号 1.....	82
3.8.2 关键信号 2.....	83
3.8.3 关键信号 3.....	83
3.8.4 部分关键代码.....	83
3.9 PS 部分	85
3.9.1 DMA 中断函数部分分析.....	85
3.9.2 IIC 驱动	90
3.9.3 main.c 文件.....	94
3.10 实验效果.....	96
CH04_AXI_DMA_OV5640 摄像头采集系统	97
4.1 概述.....	97
4.2 系统构架.....	97
4.2.1 构架方案图.....	97
4.2.2 构 BLOCK 模块化设计方案图	98
4.3 IIC 接口	98
4.4 vid in IP 介绍.....	98
4.4.1 OV_Sensor_ML 自定义 IP 模块.....	98
4.4.2 vid in IP 模块.....	106
4.4.3 VID_IN IP 接口信号的定义	107
4.5 VTC IP 的分析	113
4.5.1 VTC IP 的参数介绍	113
4.5.2 VTC IP 接口信号的定义	115
4.5.3 VTC IP 配置寄存器	119
4.5.5 设置 VTC IP	126
4.6 PLL 时钟设置.....	127
4.7 VID_OUT IP 的分析	128
4.7.1 VID_OUT 的参数介绍.....	128
4.7.2 VID_OUT IP 接口信号的定义	129
4.8 FPGA 实现的用户逻辑代码.....	132
4.8.1 关键信号 1.....	132
4.8.2 关键信号 2.....	133
4.8.3 关键信号 3.....	133
4.8.4 部分关键代码.....	133
4.9 PS 部分	135
4.9.1 DMA 中断函数部分分析.....	135
4.9.2 IIC 驱动	140
4.9.3 main.c 文件.....	147
4.10 实验效果.....	149
CH05_AXI_VDMA_OV7725 摄像头采集系统	150
5.1 为什么要用 VDMA.....	150
5.1.1 什么是帧缓存.....	150
5.1.2 双缓冲机制.....	150
5.1.3 Zynq 硬件架构	152
5.1.4 VDMA 的作用.....	152

5.2 VDMA 概述.....	152
5.3 VDMA 详细介绍.....	154
5.3.1 接口.....	154
5.3.2 VDMA 帧存格式.....	155
5.3.3 读写通道工作时序.....	155
5.3.4 寄存器.....	156
5.3.5 帧同步选项.....	162
5.3.6 Genlock 同步机制	162
5.4 使用 VDMA	164
5.4.1 IP 核配置	164
5.4.2 软件控制流程.....	165
5.5 搭建 VDMA 图像系统.....	166
5.5.1 构架方案图.....	166
5.5.2 构 BLOCK 模块化设计方案图	167
5.6 PS 部分	167
5.7 测试结果.....	169
CH06_AXI_VDMA_OV5640 摄像头采集系统	170
6.1 概述.....	170
6.2 搭建 VDMA 图像系统.....	170
6.2.1 构架方案图.....	170
6.2.2 构 BLOCK 模块化设计方案图	171
6.3 PS 部分	171
6.4 测试结果.....	172
CH07_DMA_LWIP 以太网传输	173
7.1 概述.....	173
7.2 搭建硬件系统.....	173
7.2.1 系统构架.....	173
7.2.2 启用 HP 接口.....	173
7.2.3 启用 PL 到 PS 的中断资源.....	174
7.2.4 启动 PS 部分的以太网接口	174
7.2.5 时钟的设置.....	174
7.2.6 DMA IP 配置	175
7.2.7 GPIO 的配置	175
7.2.8 配置 axi_data_fifo_0	176
7.2.9 设置 S_AXIS 接口	176
7.2.10 地址空间映射.....	177
7.3 FPGA 的发送代码.....	177
7.4 PS 部分 BSP 设置	179
7.4.1 SDK 工程 BSP 设置.....	179
7.4.2 lwip 函数库设置.....	179
7.5 PS 部分程序分析	181
7.5.1 main.c 分析	181
7.5.2 AXI DMA 数据传输过程.....	184
7.5.3 TCP 发送流程.....	187

7.6 连接测试.....	188
CH08_DMA_4_Video_Switch 视频切换系统	192
8.1 概述.....	192
8.2 修改 OV_Sensor_ML 摄像头采集 IP.....	192
8.3 搭建硬件系统.....	194
8.3.1 系统图.....	194
8.3.2 OV_Sensor_ML IP 接线图.....	195
8.3.3 vid_in IP 的接线图	195
8.3.4 DMA 和 FIFO 通路	196
8.3.5 vid_out IP 的通路	197
8.3.6 AXI HP 通道和 DMA 中断.....	197
8.3.7 DMA IP 的设置	198
8.3.8 时钟管理模块.....	198
8.3.9 VTC 图像时序发生模块.....	198
8.4 FPGA 四路输入以及图像切换源码分析	199
8.4.1 按钮输入去抖代码.....	199
8.4.2 DMA 4 路视频输入的 FPGA 代码.....	199
8.4.3 DMA 输出通道.....	202
8.5 4 路视频切换 DMA C 处理源码分析	203
8.5.1 main.c 源码	203
8.5.2 dma_intr.h 源码.....	207
8.5.3 dma_intr.c 中断接收源码.....	210
8.5.4 dma_intr.c 中断发送源码	214
8.6 测试结果.....	217
8.7 本章小结.....	217
CH09_DMA_2_Video_Stitch 视频拼接系统	218
9.1 概述.....	218
9.2 修改 OV_Sensor_ML 摄像头采集 IP.....	218
9.3 搭建硬件系统.....	219
9.3.1 系统图.....	219
9.3.2 OV_Sensor_ML IP 接线图.....	220
9.3.3 vid_in IP 的接线图	220
9.3.4 DMA 和 FIFO 通路	221
9.3.5 vid_out IP 的通路	222
9.3.6 AXI HP 通道和 DMA 中断.....	222
9.3.7 DMA IP 的设置	223
9.3.8 时钟管理模块.....	223
9.3.9 VTC 图像时序发生模块	223
9.4 FPGA 2 路输入以及图像拼接源码分析	224
9.4.1 图像常量参数	224
9.4.2 DMA 2 路视频输入的 FPGA 代码	225
9.4.3 DMA 输出通道	226
9.5 2 路视频切换 DMA C 处理源码分析	228
9.6 测试结果.....	228

CH10_基于 TCP 的 QSPI Flash bin 文件网络烧写.....	229
10.1 概述.....	229
10.2 基本原理.....	229
10.3 Bin 文件.....	229
10.4 QSPI Flash	230
10.5 驱动程序.....	231
10.5.1 建立 TCP Server	231
10.5.2 lwip 库设置.....	231
10.5.3 程序解析.....	232
10.5.4 接收保存 BOOT.bin 文件	233
10.5.5 烧写 QSPI Flash	233
10.5.6 TCP 调试信息输出.....	233
10.6 网络调试助手操作方法.....	234
10.6.1 发送 bin 文件.....	234
10.6.2 发送启动 Flash 烧写命令	235
10.7 Bin 文件更新验证.....	237
10.8 待改进之处.....	237
CH11_基于 UDP 的 QSPI Flash bin 文件网络烧写.....	238
11.1 概述.....	238
11.2 基本原理.....	238
11.2.1 Bin 文件	238
11.2.2 QSPI Flash.....	238
11.3 驱动程序.....	238
11.3.1 main 函数.....	238
11.3.2 建立 UDP 连接.....	239
11.3.3 lwip 库设置.....	239
11.3.4 程序解析.....	240
11.3.5 接收保存 BOOT.bin 文件	240
11.3.6 烧写 QSPI Flash.....	240
11.3.7 UDP 调试信息输出	240
11.4 网络调试助手操作方法.....	241
11.4.1 发送 bin 文件.....	241
11.4.2 发送启动 Flash 烧写命令	241
11.5 Bin 文件更新验证	243
11.6 待改进之处.....	243
CH12_ZYNQ A9 TCP UART 双核 AMP 例程	244
12.1 概述.....	244
12.2 基本原理.....	244
12.2.1 软件中断.....	244
12.2.2 共享内存通信.....	245
12.2.3 双核 BOOT.....	245
12.3 驱动程序.....	246
12.3.3 CORE0 工程.....	246
12.4 CORE1 工程.....	248

12.4.1 main 函数.....	248
12.4.2 初始化软件中断.....	248
12.4.3 响应软件中断.....	248
12.4.4 共享内存数据读出.....	249
12.4.5 触发软件中断.....	249
12.5 工程创建及设置关键步骤.....	249
12.6 工程调试关键步骤.....	251
12.7 网络调试助手操作方法.....	251
12.8 生成 BOOT.bin.....	253
12.9 双核 BOOT 验证.....	254
CH13_通过 BRAM 进行 PS 和 PL 间的数据交互.....	255
13.1 概述.....	255
13.2 基本原理.....	255
13.3 PL 部分设计	256
13.3.1 IP 连线图	256
13.3.2 PS 配置	256
13.3.3 AXI BRAM Controller.....	256
13.3.4 Block Memory Generator.....	257
13.3.5 AXI GPIO.....	259
13.4 逻辑设计.....	260
13.4.1 BRAM 读时序	260
13.4.2 BRAM 写时序	261
13.5 PS 程序设计	261
13.5.1 main 函数.....	261
13.5.2 GPIO 输入输出	261
13.5.3 BRAM 数据写入	261
13.5.4 BRAM 数据读出	262
13.6 程序测试.....	262
13.7 课后习题.....	264
CH14_EMIO 光电通信-FEP 子卡的使用.....	265
14.1 概述.....	265
14.2 基本原理.....	265
14.2.1 88E1512	265
14.2.2 88E1512 RGMII 接口时序.....	267
14.3 PL 部分设计	269
14.3.1 IP 连线图	269
14.3.2 ZYNQ PS 设置	269
14.3.3 GMII to RGMII	270
14.3.4 时序约束.....	272
14.3.4 IO 口	275
14.4 PS 程序设计	276
14.4.1 LWIP 库修改	276
14.4.2 创建工程.....	281
14.5 程序测试.....	283

14.5.1 电口测试.....	283
14.5.2 光口测试.....	285
CH15 PL AXI ETH 光电网络通信.....	287
15.1 概述.....	287
15.2 基本原理.....	287
15.2.1 88E1512	287
15.2.2 88E1512 RGMII 接口时序.....	289
15.3 PL 部分设计	292
15.3.1 IP 连线图	292
15.3.2 ZYNQ PS 设置	292
15.3.3 AXI 1G/2.5G Ethernet Subsystem	293
15.3.4 AXI Direct Memory Access	295
15.3.5 PL 至 PS 的中断.....	297
15.3.6 时序约束.....	298
15.3.7 IO 口	302
15.4 PS 程序设计	302
15.4.1 LWIP 库修改	302
15.4.2 创建工程.....	305
15.5 程序测试.....	307
15.5.1 电口测试.....	307
15.5.2 光口测试.....	309
CH16 基于 μGUI 的触摸屏 GUI 界面设计	311
16.1 概述.....	311
16.2 基本原理.....	311
16.3 LCD 触摸屏.....	311
16.3.1 液晶屏.....	313
16.3.2 触摸屏.....	314
16.3.3 触摸屏唤醒.....	314
16.3.4 触摸中断.....	314
16.3.5 触摸信息获取.....	315
16.3.6 触摸屏与 Miz ZYNQ 开发板的接口	318
16.4 μGUI 概述	319
16.4.1 μGUI 库移植.....	319
16.4.2 颜色空间.....	321
16.4.3 字体大小.....	321
17.5 PL 逻辑框架.....	321
16.5.1 PS 设置	322
16.5.2 GUI 界面显示.....	322
16.5.3 AXI PWM	327
16.5.4 AXI GPIO.....	328
16.5.5 Clocking Wizard.....	329
16.5.6 IO 口	330
16.6 PS 程序设计	332
16.6.1 main 函数.....	333

16.6.2 时钟重配置.....	334
16.6.3 PWM 信号输出	337
16.6.4 GPIO 输入输出	337
16.6.5 I2C 读取触摸信息.....	338
16.6.6 GUI 界面显示.....	338
16.6.6.1 AXI VDMA.....	339
16.6.6.2 显示时序设置.....	339
16.6.7 定时器.....	339
16.6.8 GUI 界面设计.....	340
16.6.8.1 GUI 初始化.....	341
16.6.8.2 窗口 1 设计.....	341
16.6.8.3 窗口 2 设计.....	344
16.6.8.4 窗口 3 设计.....	345
16.6.8.5 窗口 4 设计.....	346
16.6.8.6 窗口 5 设计.....	348
16.7 注意事项.....	349
16.7.1 更改 GUI 分辨率.....	349
16.7.2 SDK 路径设置.....	349

CH01_AXI_DMA_LOOP 环路测试

1.1 概述

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

本课程的设计原理分析。

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

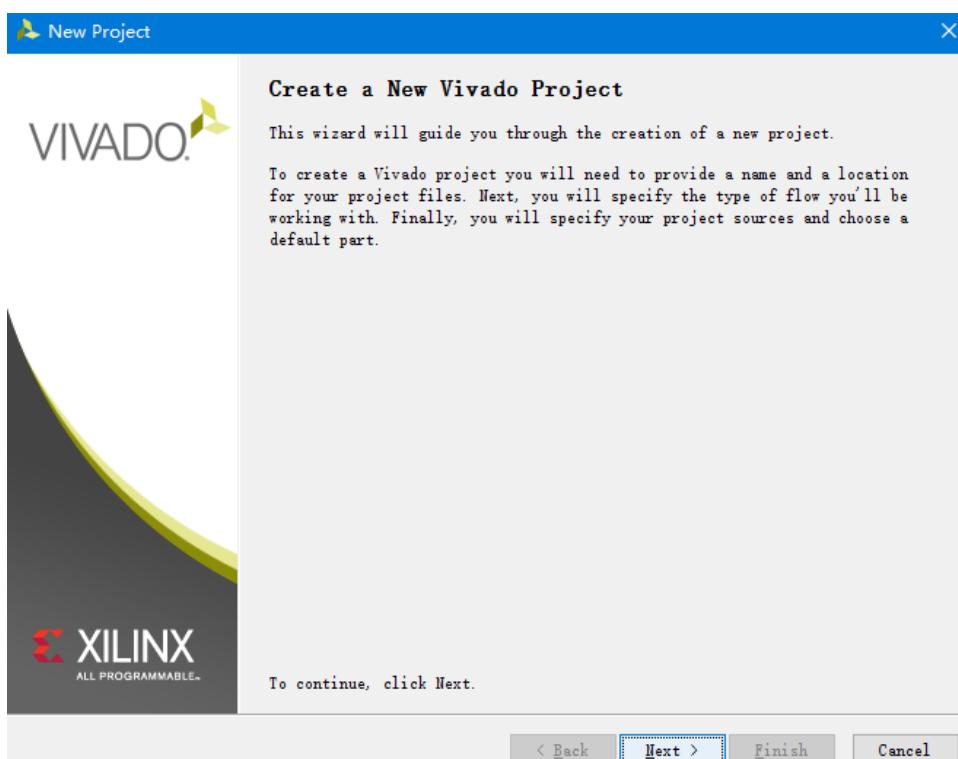
1.2 搭建硬件系统

1.2.1 新建 VIVADO 工程

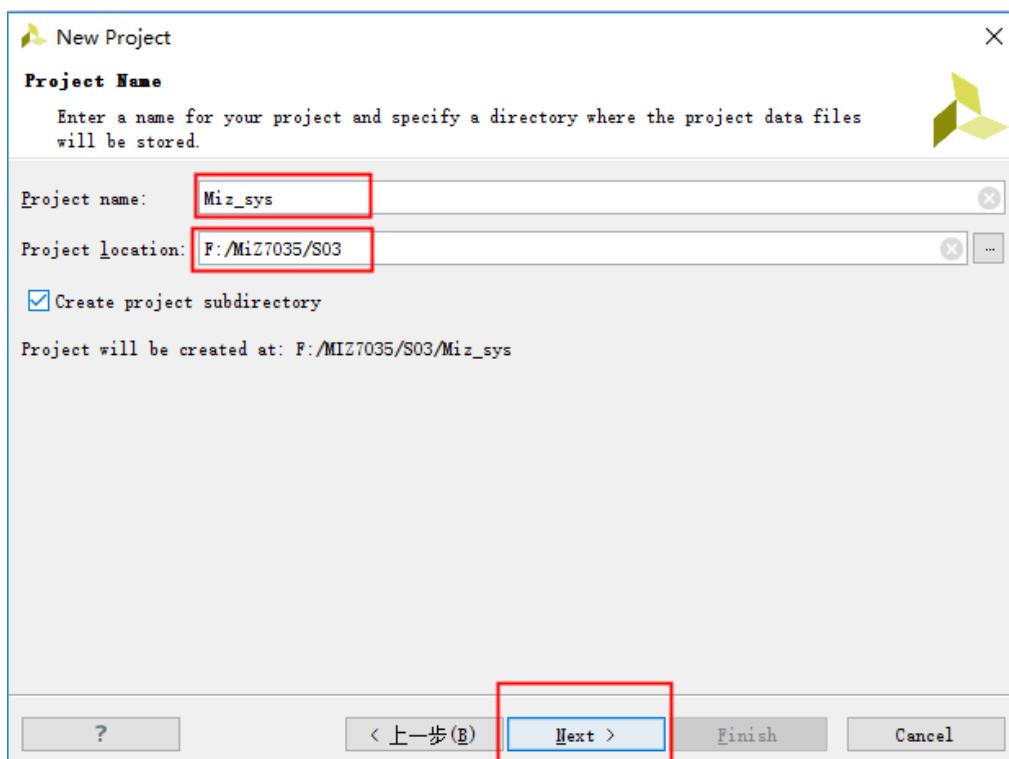
Step1:启动 VIVADO，单击 Create New Project



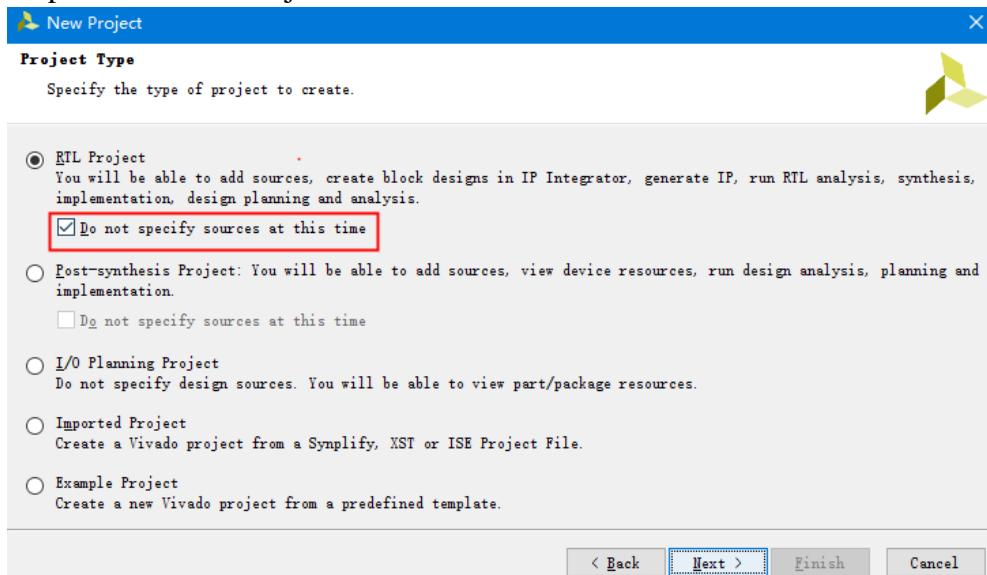
Step2:单击 NEXT



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

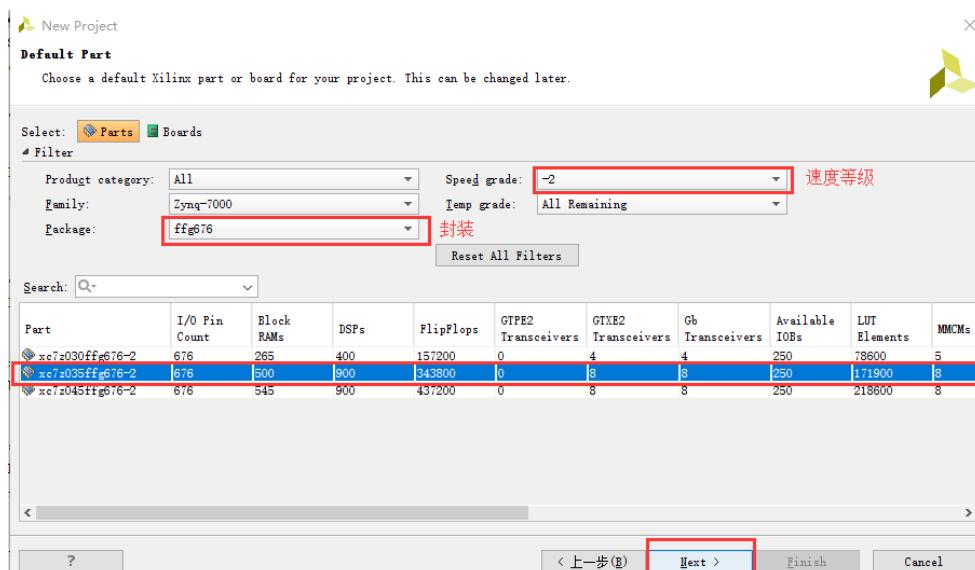


Step 4 :选择 RTL Project 并且勾选复选框，之后单击 NEXT



Step5:选择芯片的型号和封装速度等级。

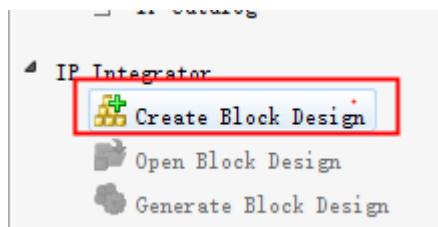
MiZ7035 如下图所示设置:



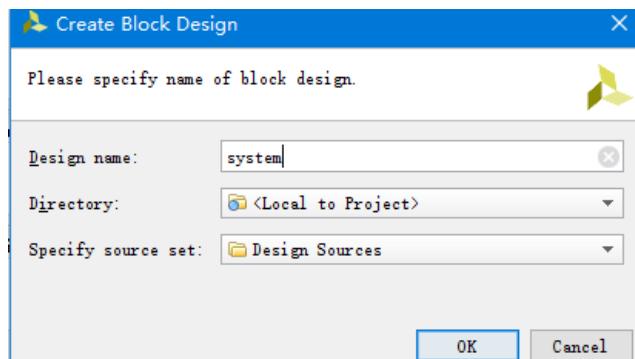
Step:6 单击 Finish 完成工程创建。

1.2.2 创建 VIVADO 硬件构架

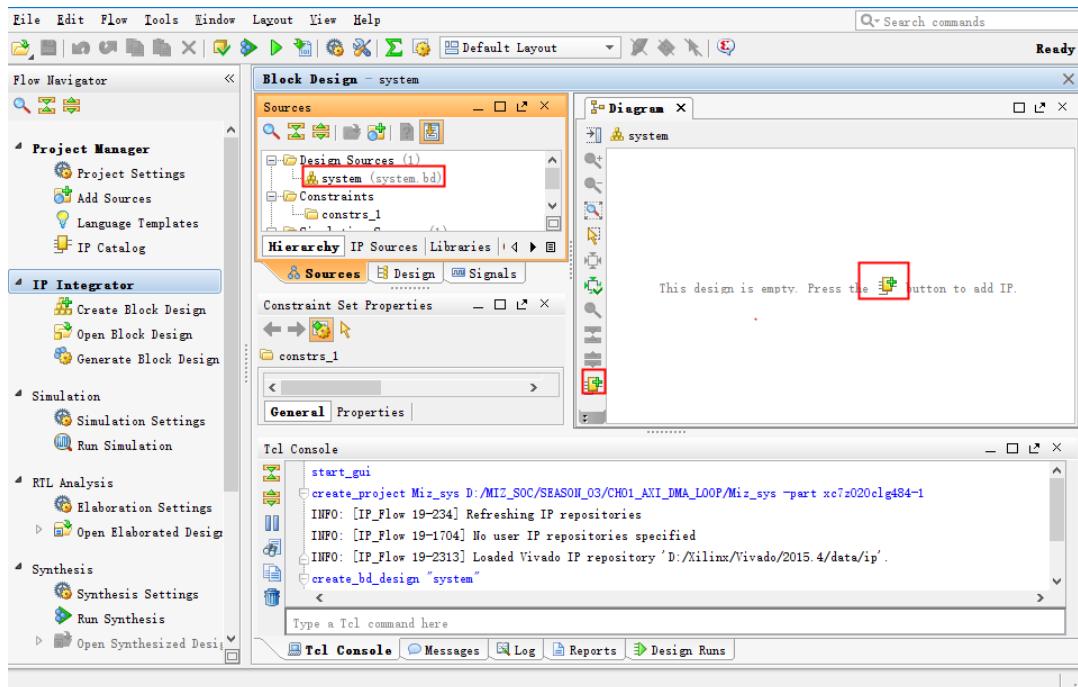
Step1:单击 Create Block Design



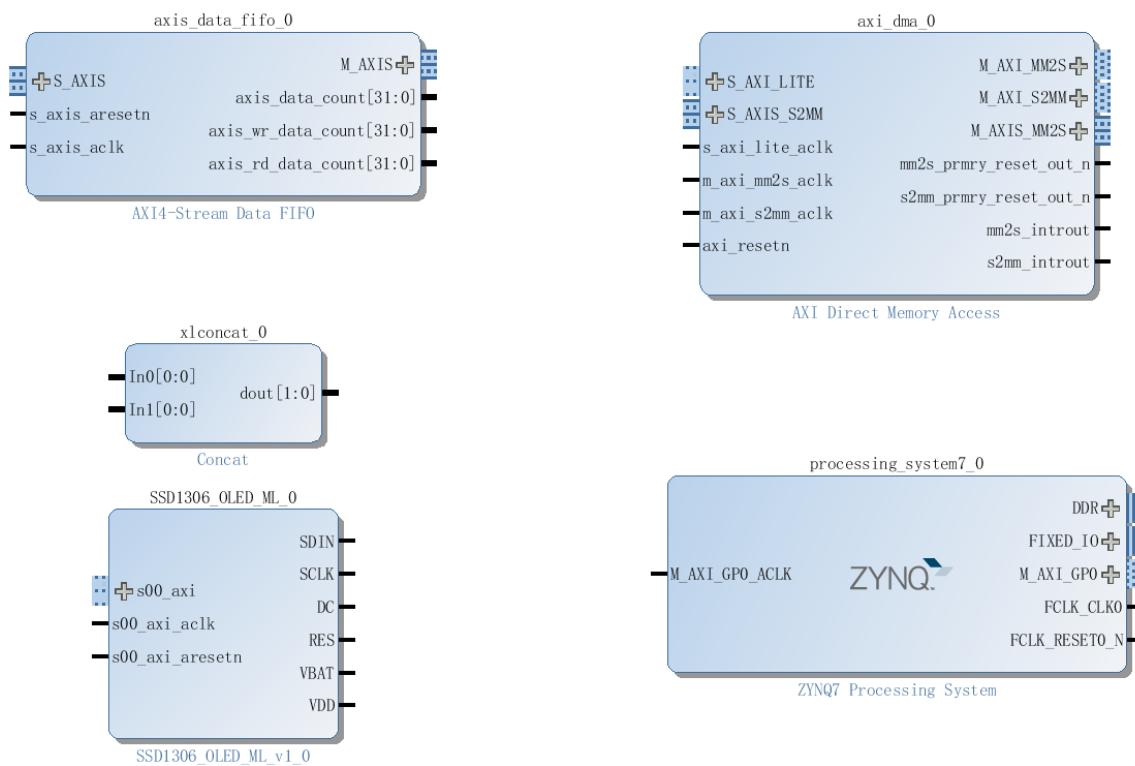
Step2: 命名为 system 之后单击 OK



Step3: 创建完成后如下图所示

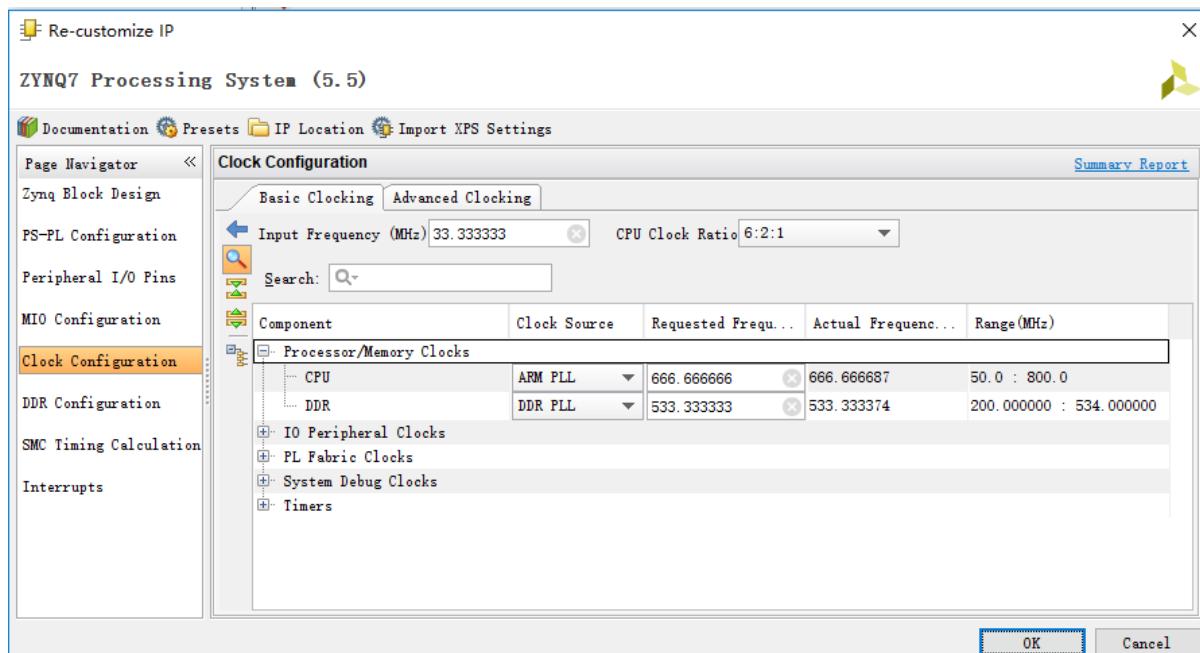


Step3: 添加各个模块如图:

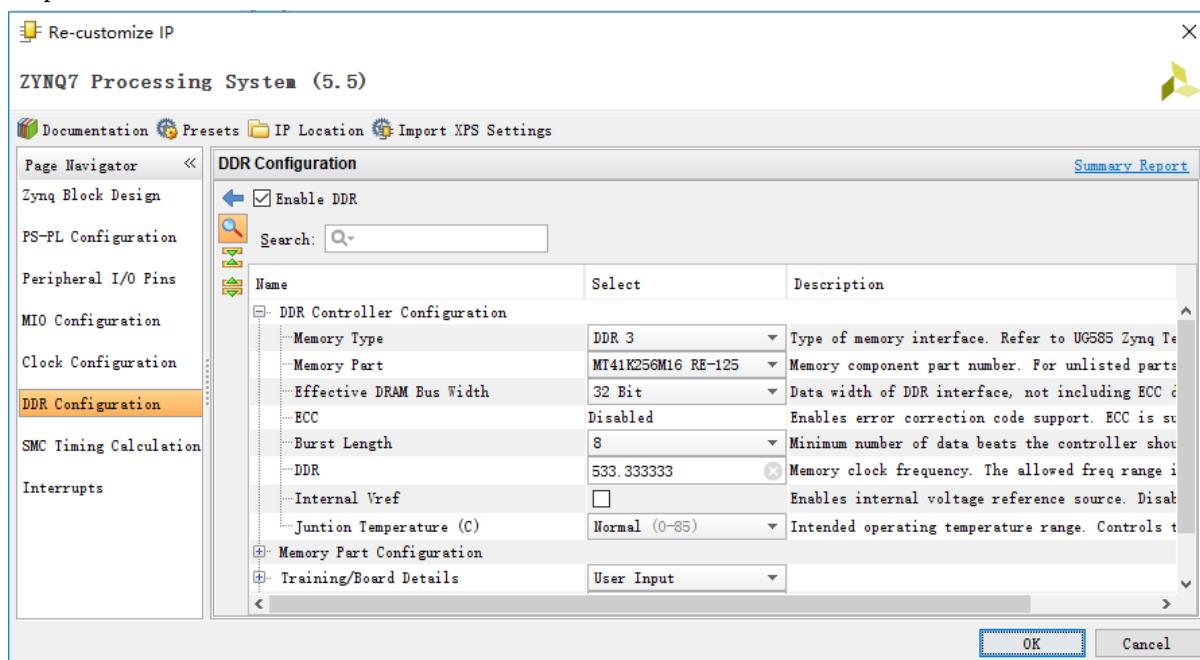


Step4 双击 ZYNQ IP 进行如下步骤配置

Step5: 配置 PS 输入时钟为 33.333333MHZ



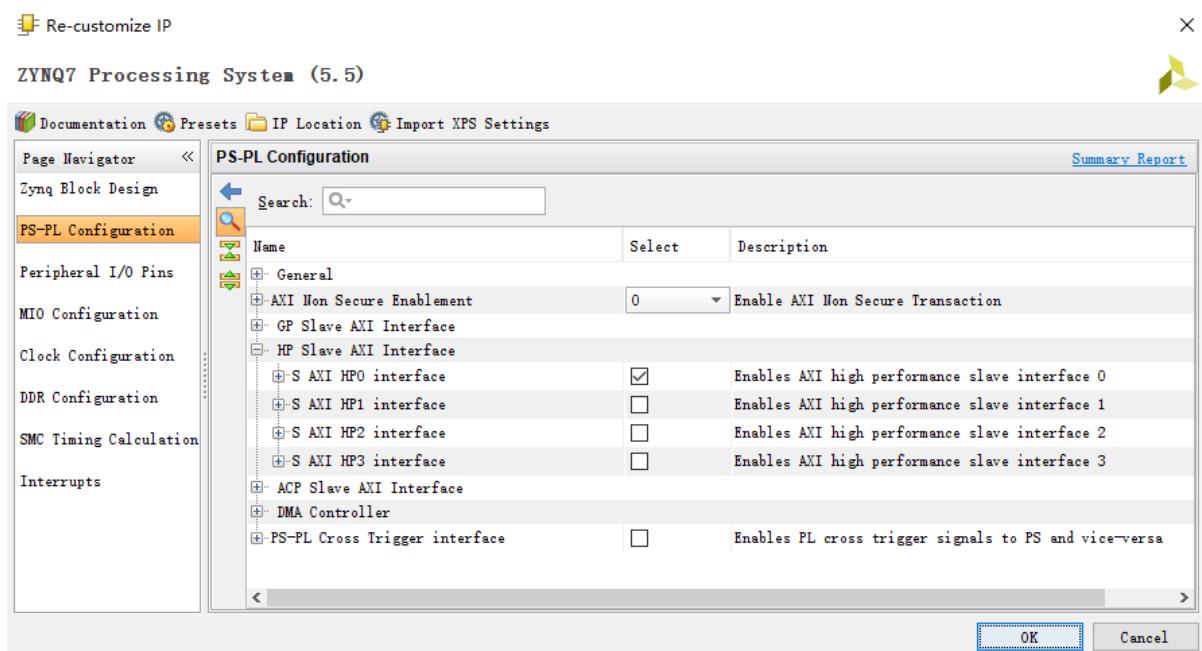
Step6: MiZ7035 开发板内存型号配置为 MT41K256M16 RE-125



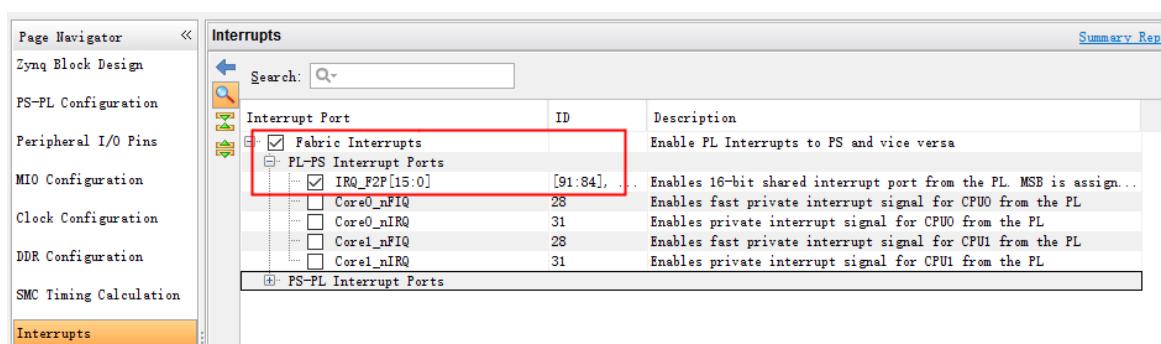
Step7: PS 的 PLL 提供本系统的时钟 100MHZ

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

Step8:启动 1 路 HP 接口，HP 接口是 ZYNQ 个高速数据接口



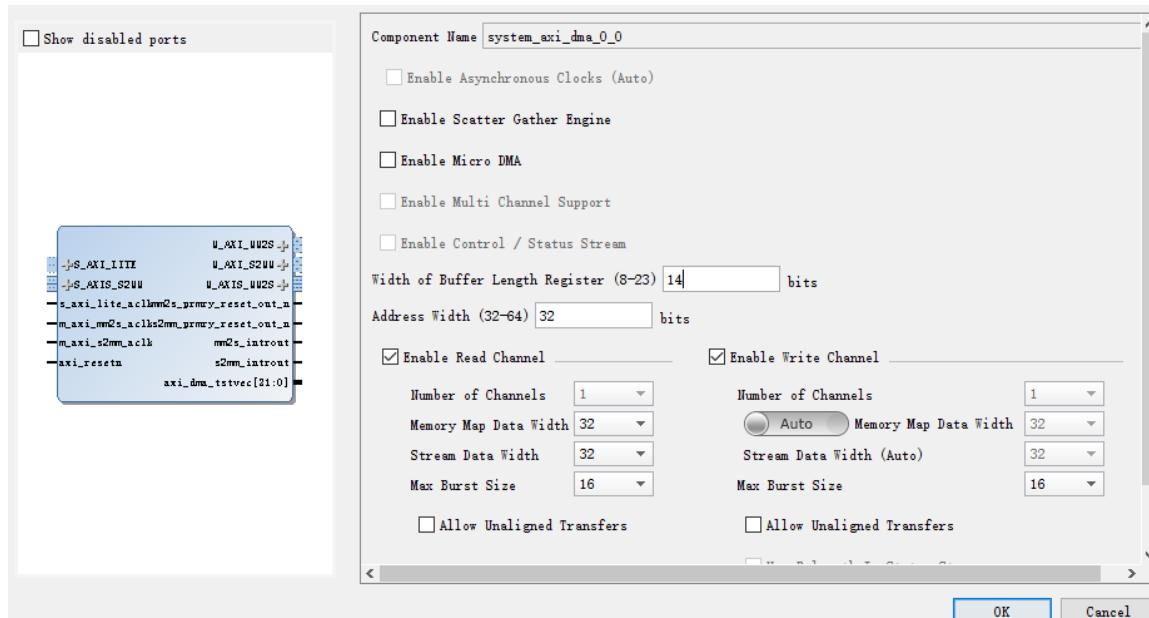
Step9: 勾选 PL 到 PS 的中断资源（关于中断，在第二季的课程中有详细讲解，不熟悉的读者可以到第二季课程中温习一下）



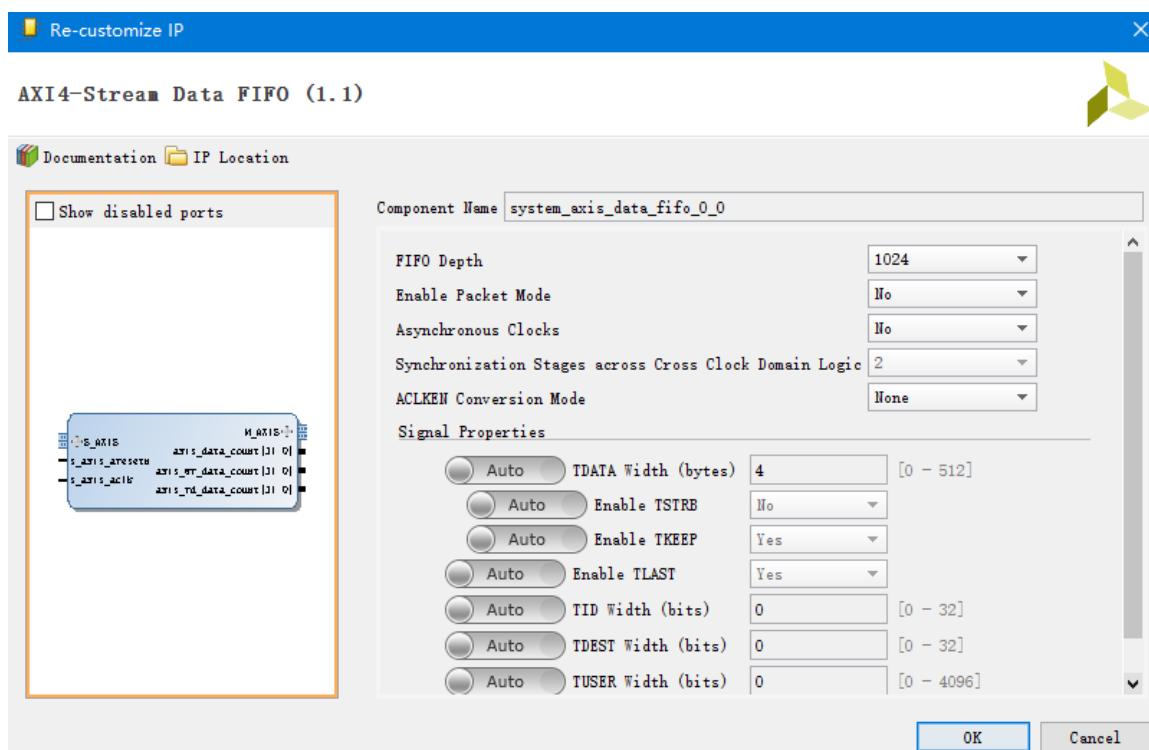
Step10: 设置完成后单击 OK

Step11: 双击 DMA IP 设置如下：

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



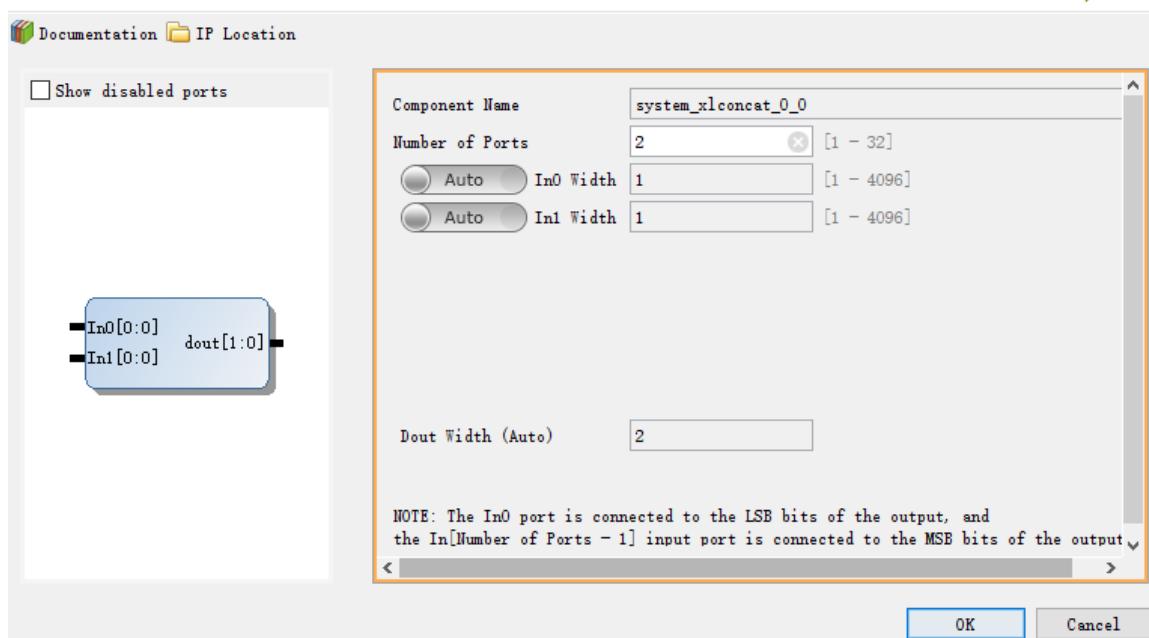
Step12:Data FIFO 设置



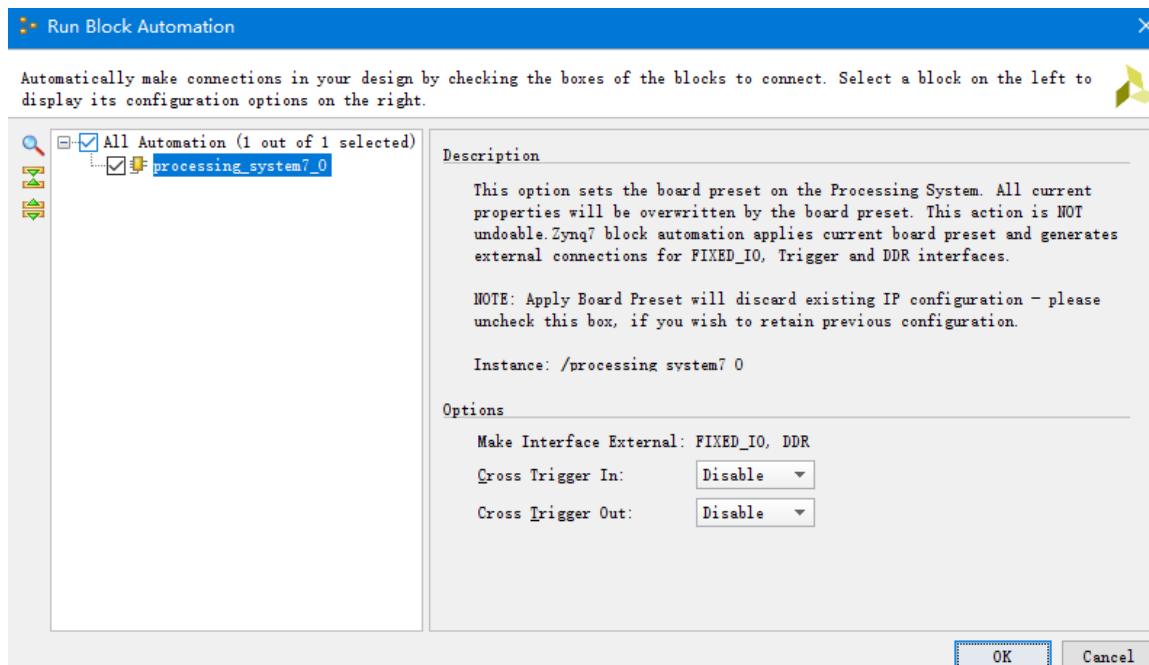
Step13:Concat IP 设置

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

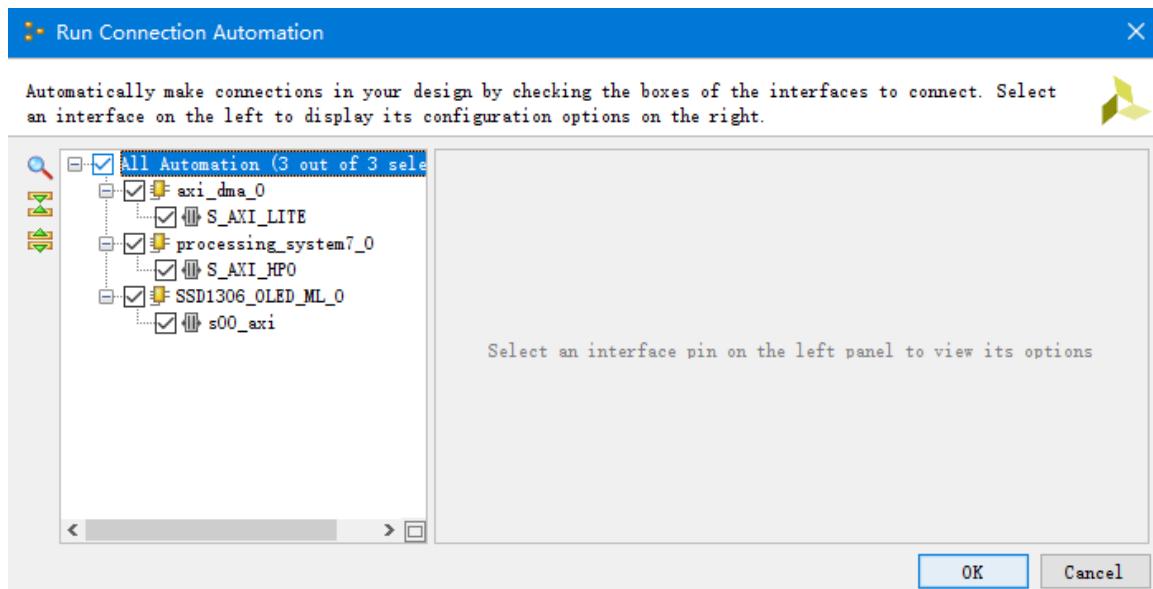
Concat (2.1)



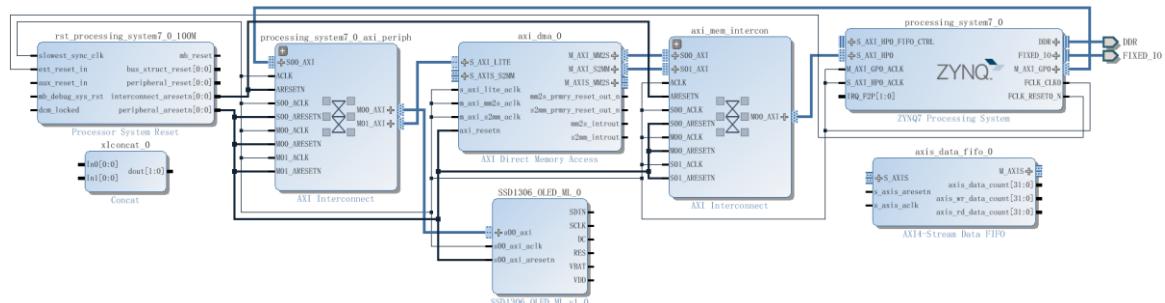
Step14: Run Automation 自动配置 ZYNQ IP 如下图所示



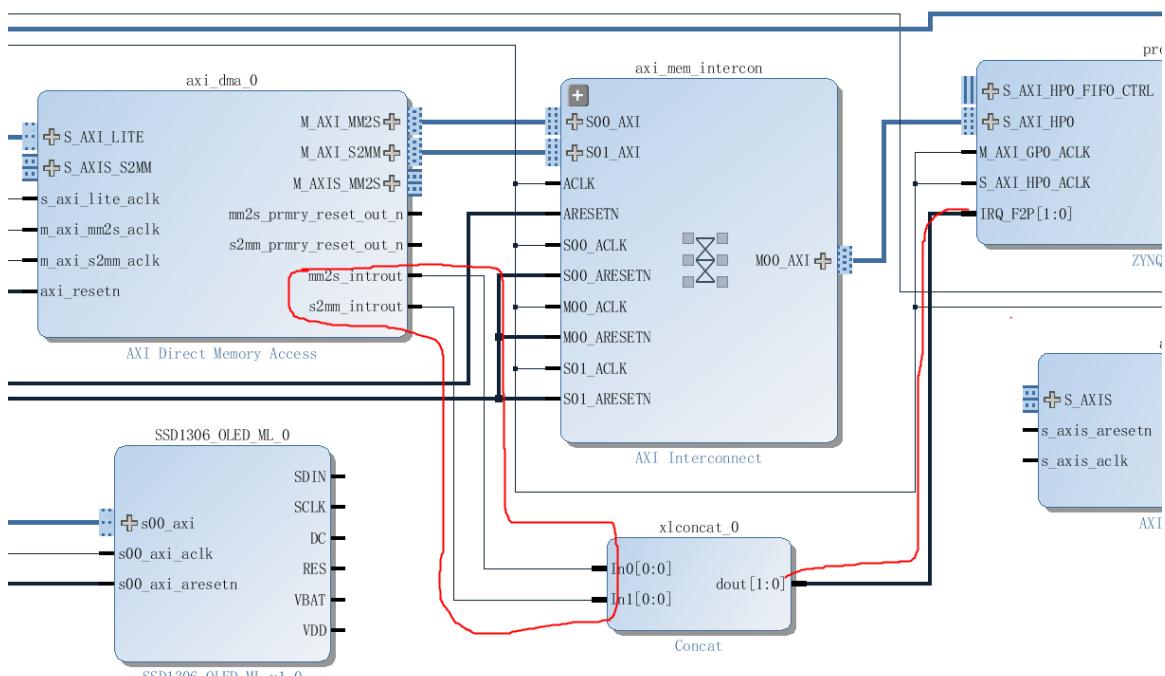
Step15: 单击 Run Connection Automation 自动连线, 只要软件提示你需要自动连线, 一般都需要进行自动连线, 除非自己知道如何连线, 有特殊需求。



Step16: 如果还有提示需要自动连线的继续让软件自动连线，直到出下如下。可以看到，还有未连线的模块。

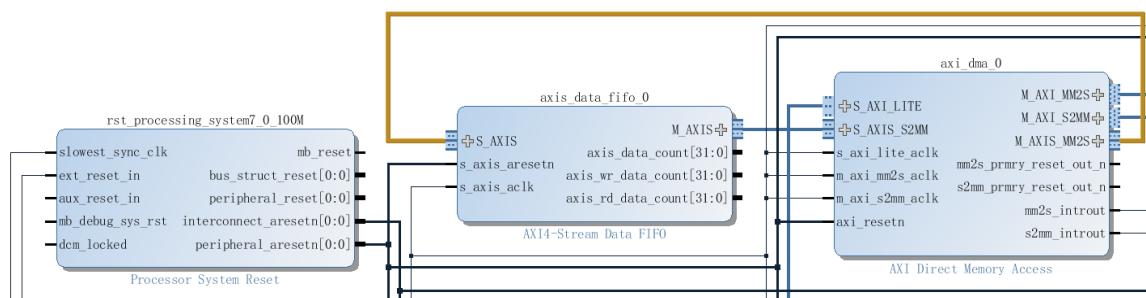


Step17: 把 DMA 收发中断信号，通过 contact IP 连接到 ZYNQ



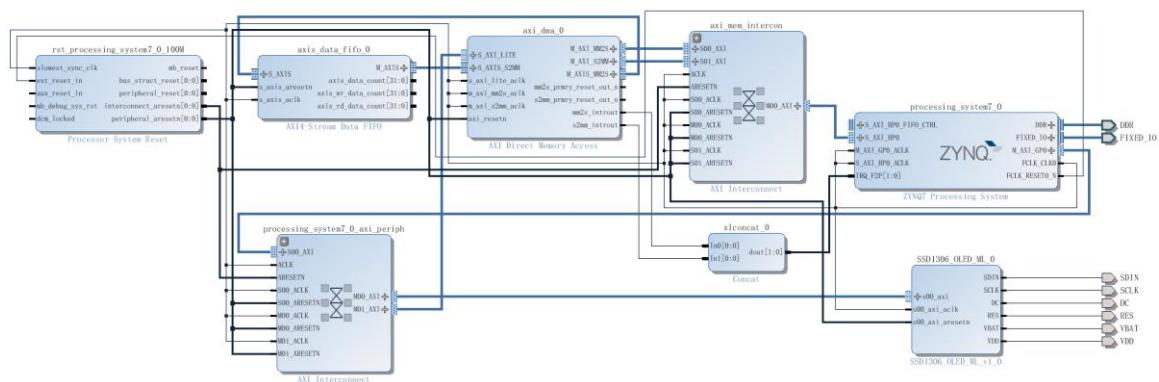
Step18:

- 连接 FIFO 的 S_AXIS(写端口)到 DMA 的 M_AXIS(DMA 读端口);
 连接 FIFO 的 M_AXIS(读端口)到 DMA 的 S_AXIS(DMA 写端口);
 连接 FIFO 的 a_axis_aresetn 到 复位 IP 的 peripheral aresetn ;
 连接 FIFO 的 s_axis_ack 到 ZYNQ IP 的 FCLK0;
 连接完成后如下图

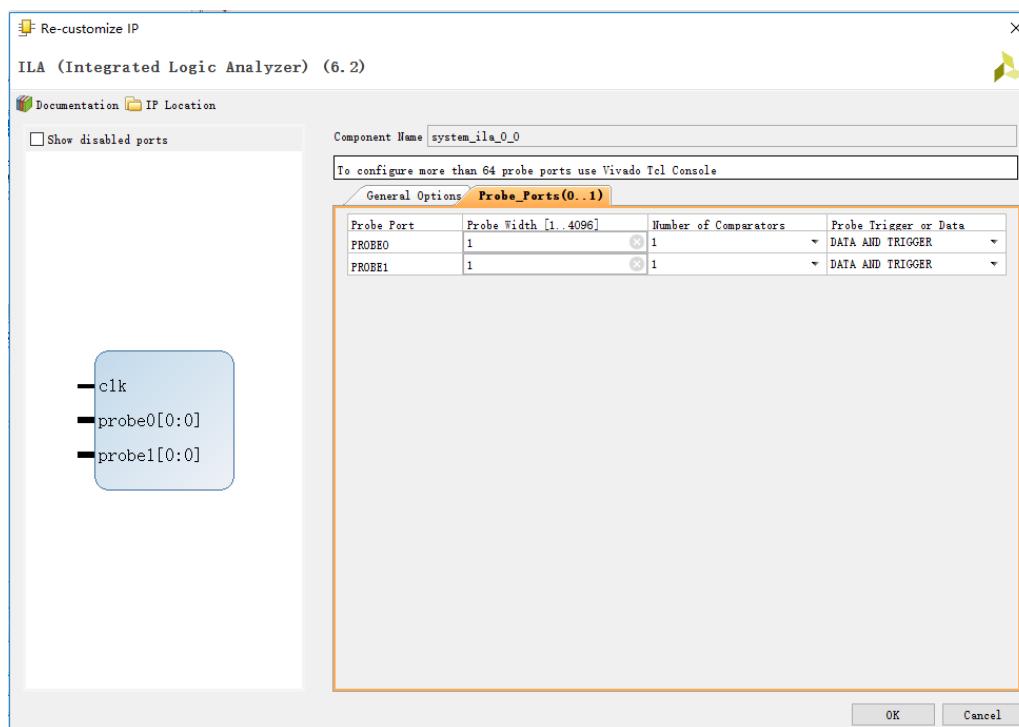
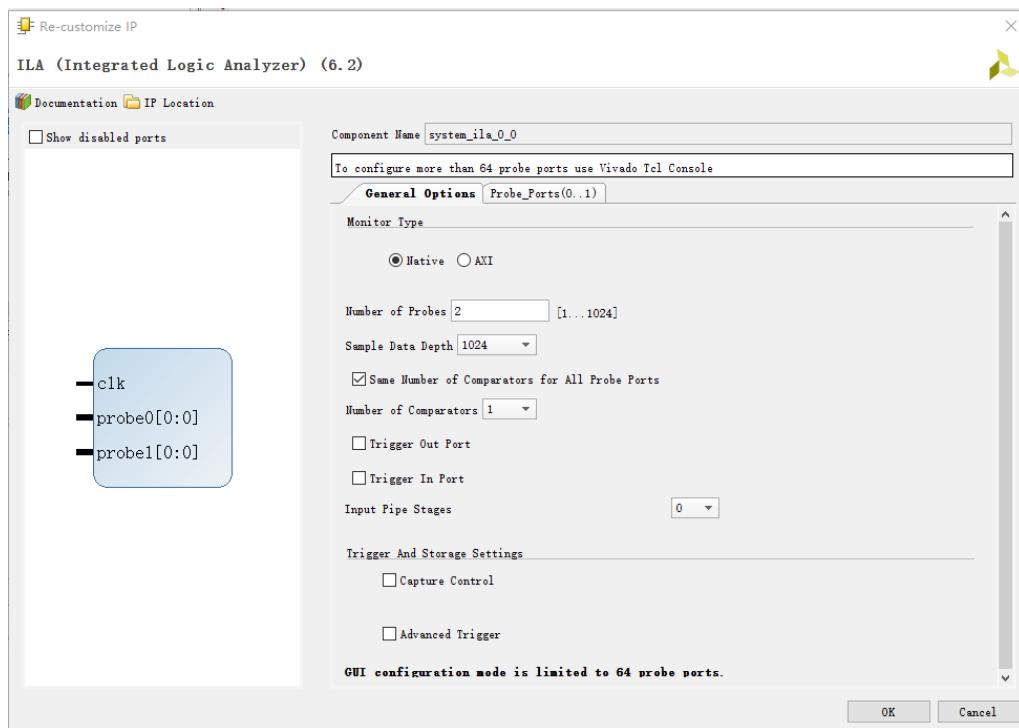


Step19: 把 OLED 模块的 IO 引出来，后面 C 代码部分会用 OLED 显示一些信息。连接完成后的工程如下图

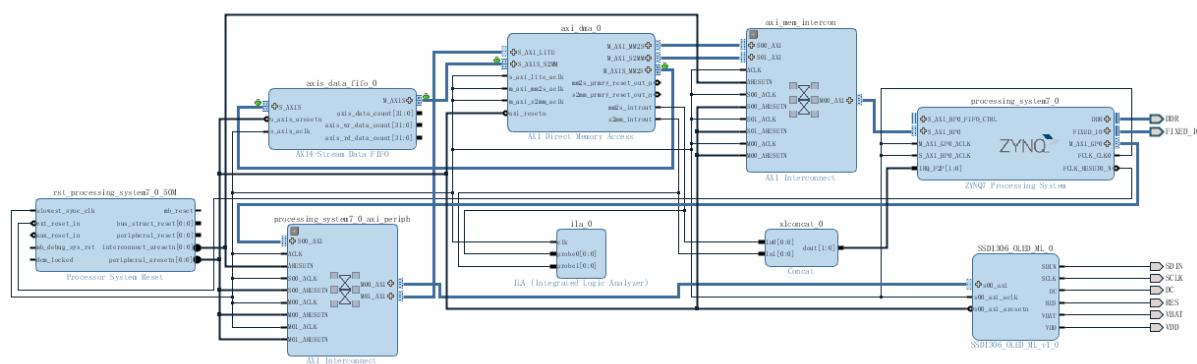
MiZ7035 连线图：



Step20: 未来调试的时候可以观察到中断信号的产生，添加 ila 调试 IP 并且进行如下设置



Step21: 把中断信号连接到 ila IP 上，另外，把时钟信号也连接起来。



Step22:

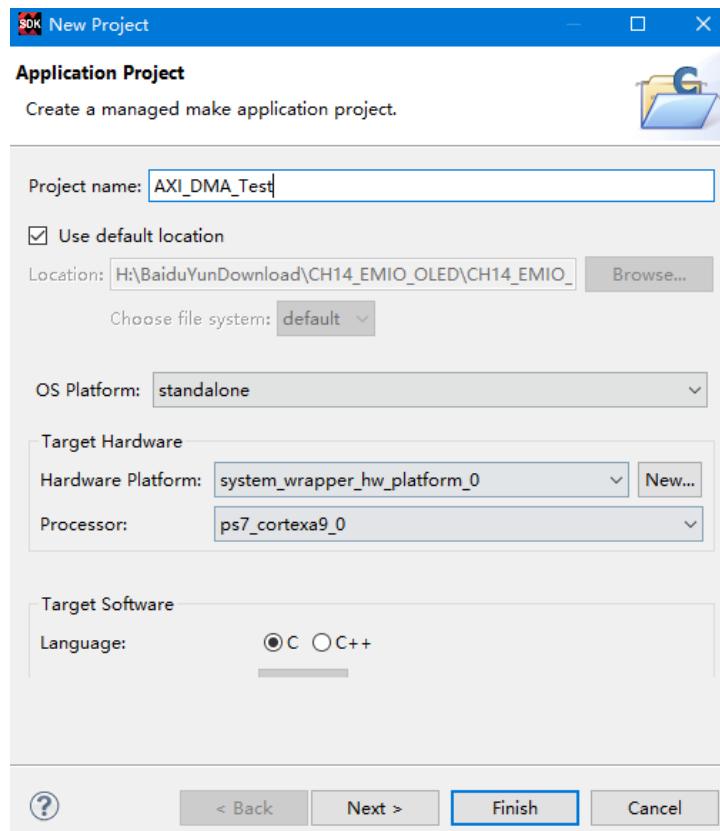
以上就完成独立工程的创建。

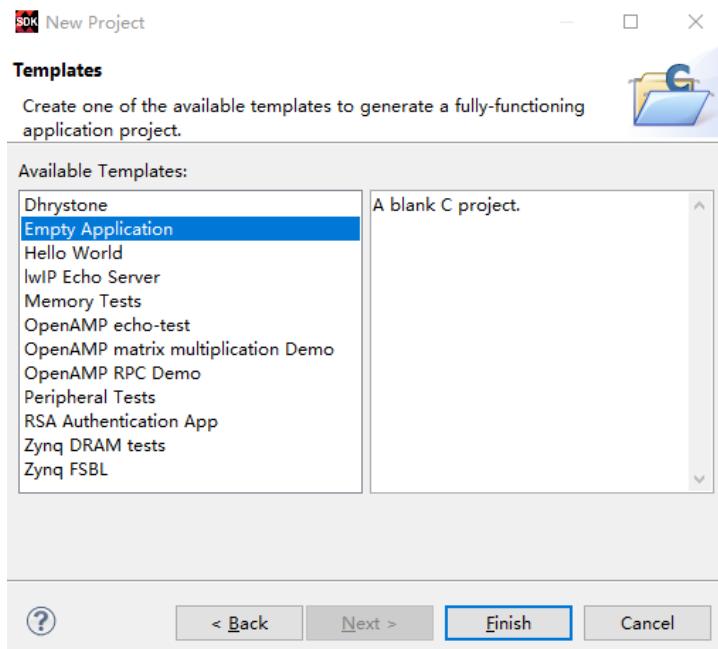
之后的过程是 Validate Design->Generate Out products->Create wrappers->Generate Bitstream 产生完成后导入到 SDK 进行软件开发。

1.3 PS 部分软件分析

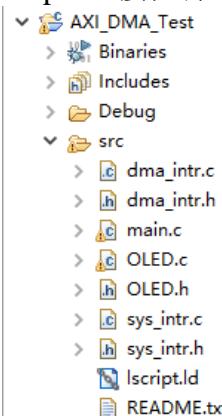
1.3.1 新建 SDK 工程

Step1:新建一个名为 AXI_DMA_Test 的空的软件工程





Step2:直接把源码复制过来，软件会自动编译



1.3.2 main.c 源码的分析

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

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

表 1-3-2-1 init_intr_sys 函数

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

```

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

```

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

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

表1-3-2-1 DMA_CheckData

```

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

    xil_printf("\r\n----DMA Test----\r\n");
    print_message( "----DMA Test----",0); //oled print

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

    sprintf(oled_str,"PKT_LEN=%d",MAX_PKT_LEN);
    print_message(oled_str,1); //oled print

    //while(1)
    for(i = 0; i < Tries; i++)
    {
        Value = TEST_START_VALUE + (i & 0xFF);
        for(Index = 0; Index < MAX_PKT_LEN; Index++) {
            TxBufferPtr[Index] = Value;

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

        /* Flush the SrcBuffer before the DMA transfer, in case the Data Cache
         * is enabled
         */
        Xil_DCacheFlushRange((u32)TxBufferPtr, MAX_PKT_LEN);
    }
}

```

```
 Status = XAxiDma_SimpleTransfer(&AxiDma,(u32) RxBufferPtr,
                                  MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    Status = XAxiDma_SimpleTransfer(&AxiDma,(u32) TxBufferPtr,
                                  MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

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

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

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

    xil_printf("AXI DMA interrupt example test passed\r\n");
    xil_printf("success=%d\r\n", success);
```

```

sprintf(oled_str,"success=%d",success);
print_message(oled_str,2);
/* Disable TX and RX Ring interrupts and return success */
DMA_DisableIntrSystem(&Intc, TX_INTR_ID, RX_INTR_ID);

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

}

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

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

}

```

1.3.3 dma_intr.c 源码分析

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

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

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

如果中断全部正确，`TxDone` 将被置为1表示发送中断完成。

如果有错误，则复位DMA，并且设置超时参数

表 1-3-3-1 DMA_TxIntrHandler 函数

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

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

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

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

    /* Acknowledge pending interrupts */

    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);

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

        return;
    }

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

```

Error = 1;

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

TimeOut = RESET_TIMEOUT_COUNTER;

while (TimeOut) {
    if (XAxiDma_ResetIsDone(AxiDmaInst)) {
        break;
    }

    TimeOut -= 1;
}

return;
}

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

    TxDone = 1;
}
}
}

```

接收中断函数的原理和发送一样

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

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

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

如果中断全部正确，`RxDone` 将被置为1表示接收中断完成。

如果有错误，则复位DMA，并且设置超时参数

表 1-3-3-2 DMA_RxIntrHandler 函数

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

```
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.
*
* @paramCallback is a pointer to RX channel of the DMA engine.
*
* @return None.
*
* @note      None.
*
*****static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

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

    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

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

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

        Error = 1;

        /* Reset could fail and hang
         * NEED a way to handle this or do not call it??
         */
    }
}
```

```

XAxiDma_Reset(AxiDmaInst);

TimeOut = RESET_TIMEOUT_COUNTER;

while (TimeOut) {
    if(XAxiDma_ResetIsDone(AxiDmaInst)) {
        break;
    }

    TimeOut -= 1;
}

return;
}

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

    RxDone = 1;
}
}

```

表 1-3-3-3 DMA_CheckData 函数

```

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

```

```

int Index = 0;
u8 Value;

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

/* Invalidate the DestBuffer before receiving the data, in case the
 * Data Cache is enabled
 */
#ifndef __aarch64__
Xil_DCACHEInvalidateRange((u32)RxPacket, Length);
#endif

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

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

return XST_SUCCESS;
}

```

1.3.4 dam_intr.h 文件分析

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

```

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

```

表 1-3-4 dam_intr.h

/*

```
*  
* www.osrc.cn  
* www.milinker.com  
* copyright by nan jin mi lian dian zi www.osrc.cn  
*/  
#ifndef DMA_INTR_H  
#define DMA_INTR_H  
#include "xaxidma.h"  
#include "xparameters.h"  
#include "xil_exception.h"  
#include "xdebug.h"  
#include "xscugic.h"  
  
/********************* Constant Definitions *****/  
/*  
 * Device hardware build related constants.  
 */  
#define DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID  
  
#define MEM_BASE_ADDR      0x01000000  
  
#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR  
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR  
  
#define TX_BUFFER_BASE      (MEM_BASE_ADDR + 0x00100000)  
#define RX_BUFFER_BASE      (MEM_BASE_ADDR + 0x00300000)  
#define RX_BUFFER_HIGH     (MEM_BASE_ADDR + 0x004FFFFF)  
  
/* Timeout loop counter for reset  
 */  
#define RESET_TIMEOUT_COUNTER    10000  
/* test start value  
 */  
#define TEST_START_VALUE    0xC  
/*  
 * Buffer and Buffer Descriptor related constant definition  
 */  
#define MAX_PKT_LEN      256//4MB  
/*  
 * transfer times  
 */
```

```
#define NUMBER_OF_TRANSFERS 100000

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

int DMA_CheckData(int Length, u8 StartValue);
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr);
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);
#endif
```

1.4 测试结果

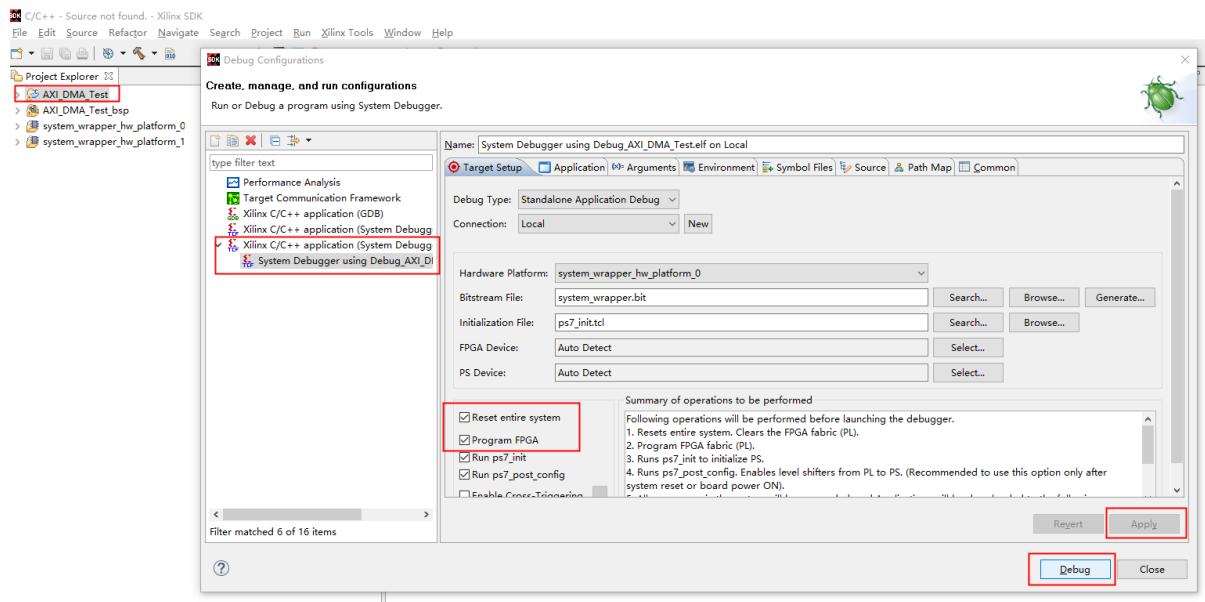
AXI_DMA_LOOP 环路测试。

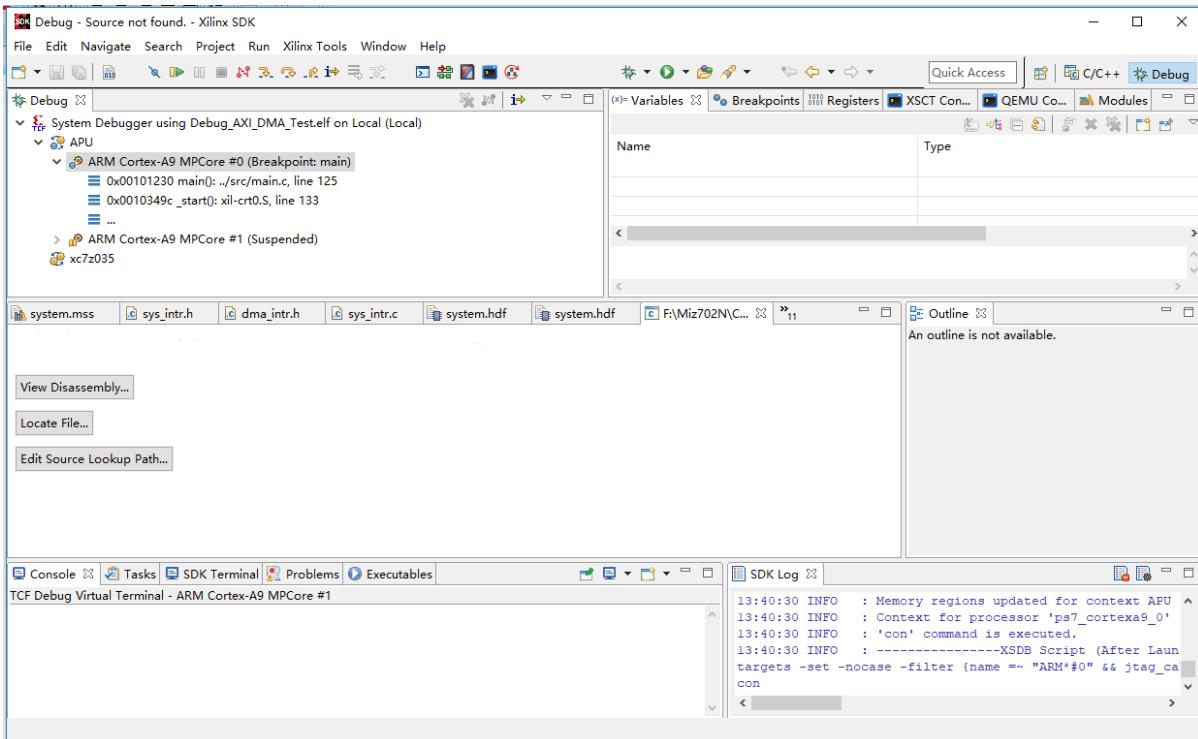
- 1、ZYNQ 的 PS 端（SDK）中发送数据给 DMA
- 2、DMA 把数据发送给 DDR，将数据写入 DDR 中某个地址空间
- 3、PS 端（SDK）从 DDR 中读取这个地址的数据，与 PS 端（SDK）发送数据给 DMA 的数据对比，确认是否一致。

测试是使用软件：VIVADIO 和 SDK

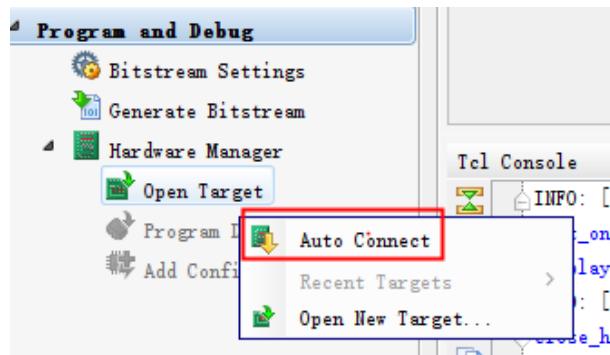
测试步骤如下：

Step1:启动 SDK

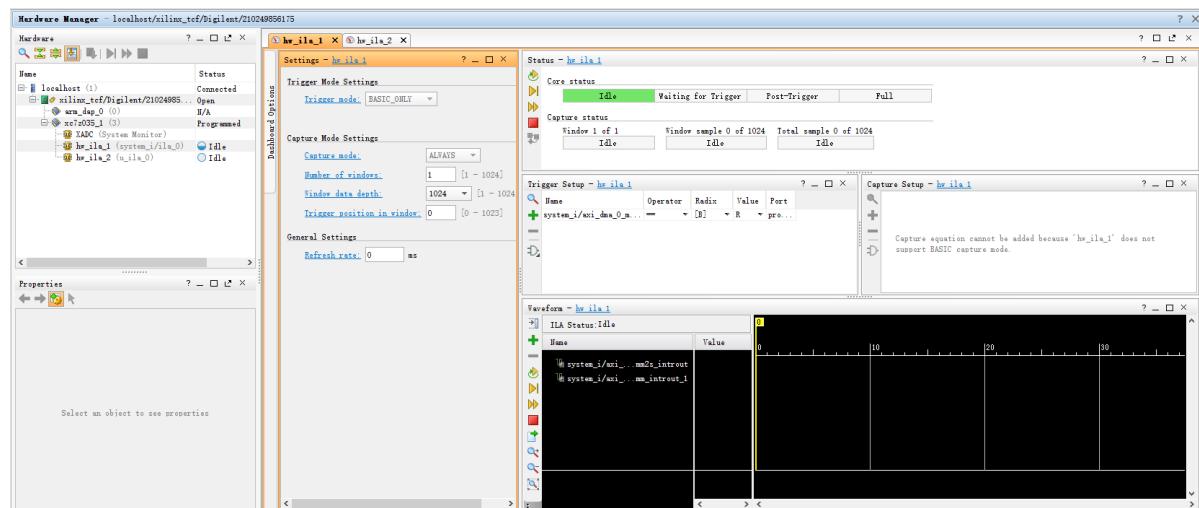




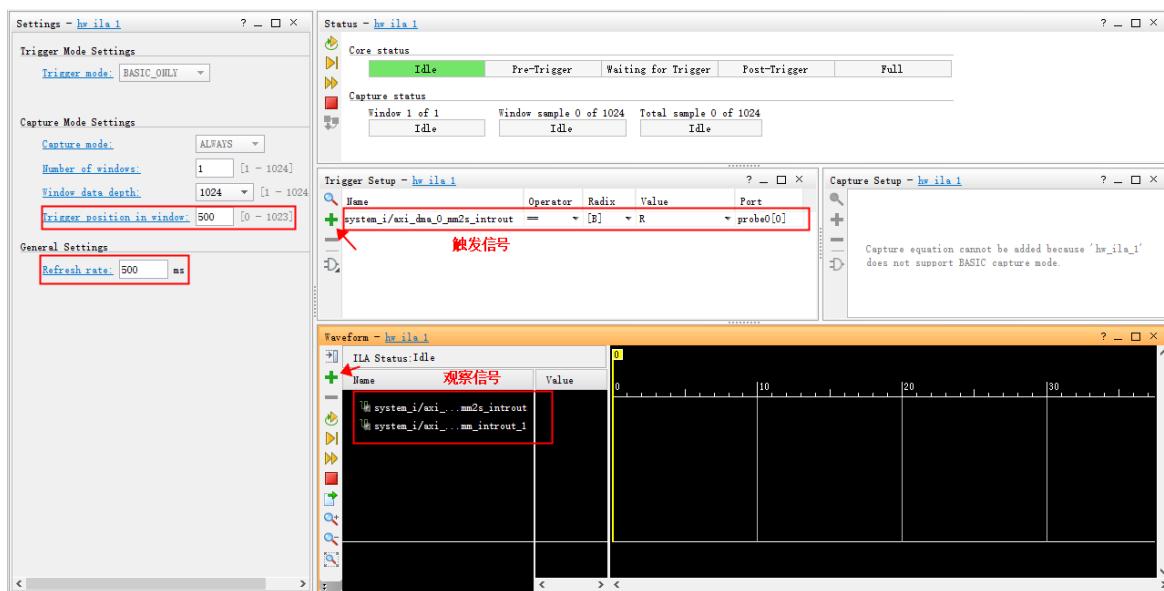
Step2:在 VIVADO 工程中点击 Open Target 然后点击 Auto Connect(前面必须先启动 SDK)



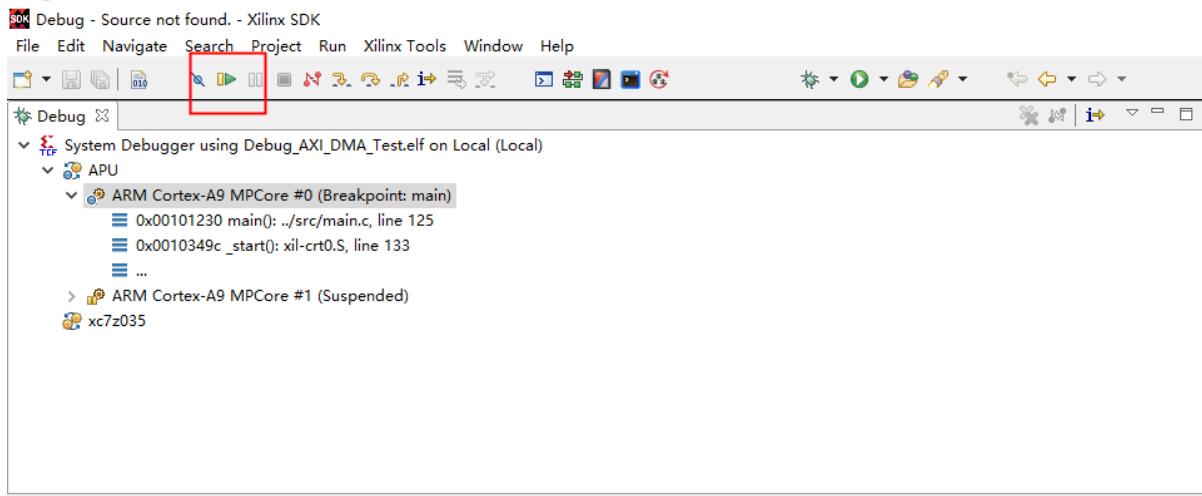
Step3:连接成功后入下图



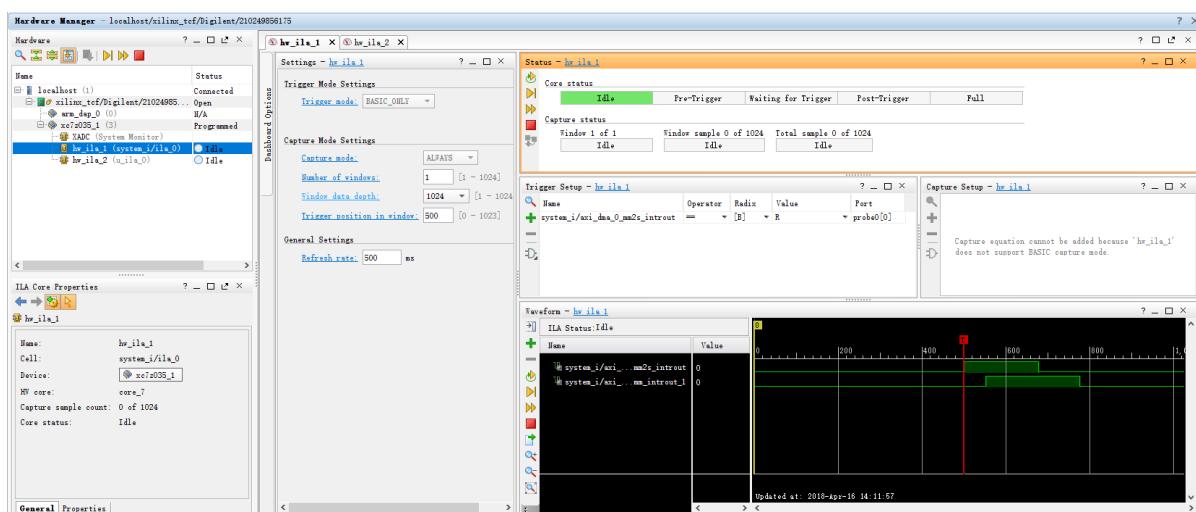
Step4:设置触发条件、观察信号。设置波形偏移 500。



Step5:SDK 中点击运行

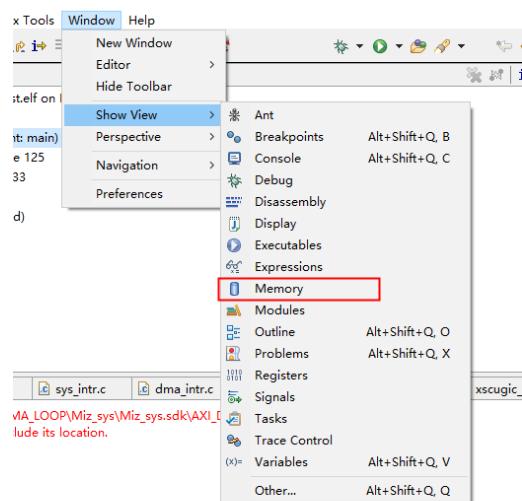


当中断触发的时，VIVADO 中 Hardware Manager 出现捕捉波形，如下图所示

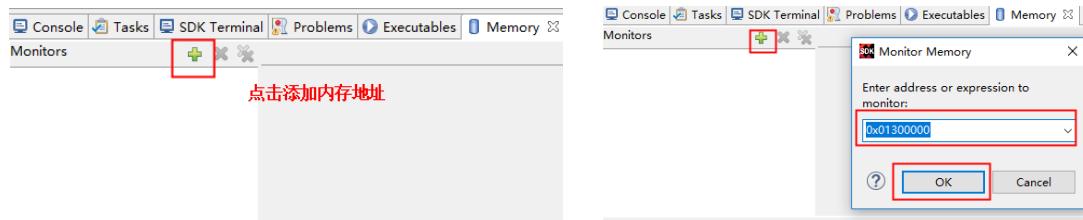


Step6:观察数据

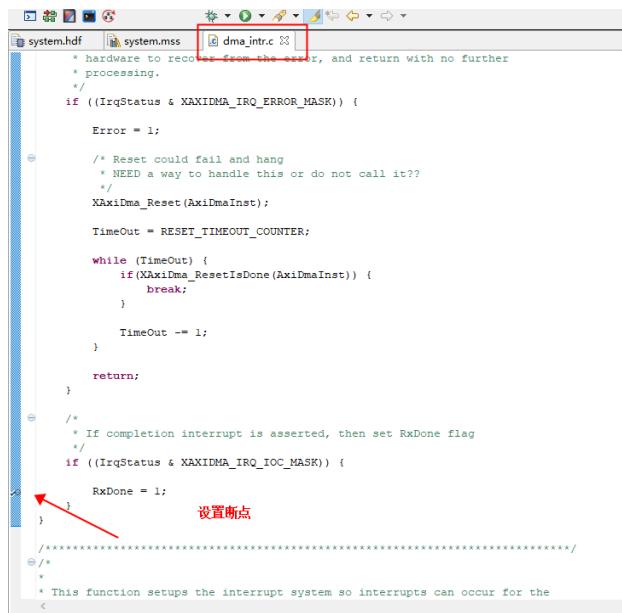
打开 Memory: Window->Show View->Memory

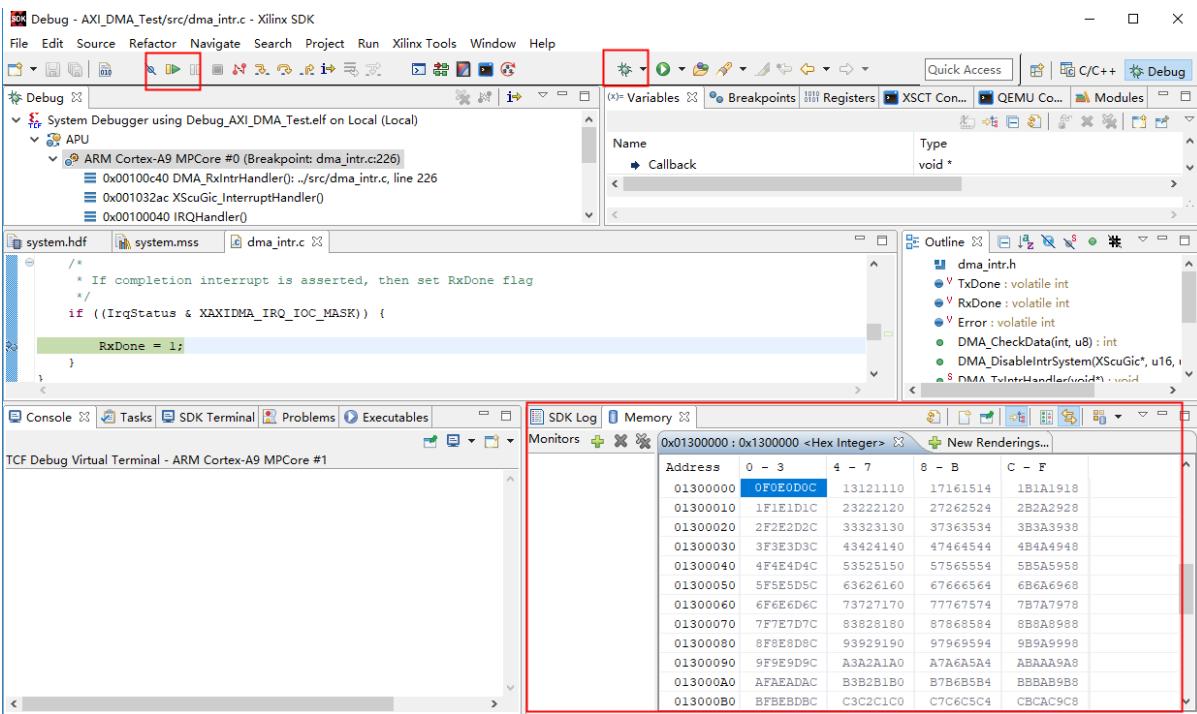


点击添加接收内存部分地址用于观察内存中的数据 地址为 0X01300000



为了观察一次收发数据：设置断点，重新让收发程序跑一次。





收发一次，从内存中读取的数据如图：

0x01300000 : 0x130000 <Hex Integer>					
Address	0 - 3	4 - 7	8 - B	C - F	
012FFFFE	DC596BAC	B50F62E3	7FDDEFA7	0DBDDDBE	
012FFFFF	73D07FDC	78C25E92	0DDE6FB0	FDCAFBD7	
01300000	0FOE0DC	1211110	17161514	1B1A1918	
01300010	1F1E1D1C	23222120	2B2A2928		
01300020	2F2E2D2C	33323130	3B3A3938		
01300030	3F3E3D3C	43424140	4B4A4948		
01300040	4F4E4D4C	53525150	5B5A5958		
01300050	5F5E5D5C	63626160	6B6A6968		
01300060	6F6E6D6C	73727170	7B7A7978		
01300070	7F7E7D7C	83828180	8B8A8988		
01300080	8F8E8D8C	93929190	9B9A9998		
01300090	9F9E9D9C	A3A2A1A0	A7A6A5A4		
013000A0	AFAEAADAC	B3B2B1B0	B7B6B5B4		
013000B0	BFBE8DBC	C7C6C5C4	CBCAC9C8		

0x01300000 : 0x130000 <Hex Integer>					
Address	0 - 3	4 - 7	8 - B	C - F	
0130007B0	BFBE8DBC	C3C2C1C0	C7C6C5C4	CBCAC9C8	
0130007C0	CFCECDC	D9D2D1D0	D7D6D5D4	DEDA5D8	
0130007D0	DFDEDDC	E3E2E1E0	E7E6E5E4	E8EAES8	
0130007E0	EFEEEDC	F3F2F1F0	F7F6F5F4	F8F5F9F8	
0130007F0	FFFEDFC	03020100	07060504	BB0A808	
013000800	57676535	5BBDFF0	5A876535	F35F17E6	
013000810	F54E5F30	CDBE3B46	C9DCB92	BB8BDF1D5	

可以看到第一个数据是 0X0C，后面是依次加 1，一次接收的数据量是 2047。发送数据也是 0X0C，后面依次加 1，发送量是 2047。接收数据一致，测试结束。

CH02_AXI_DMA PL 发送数据到 PS

2.1 概述

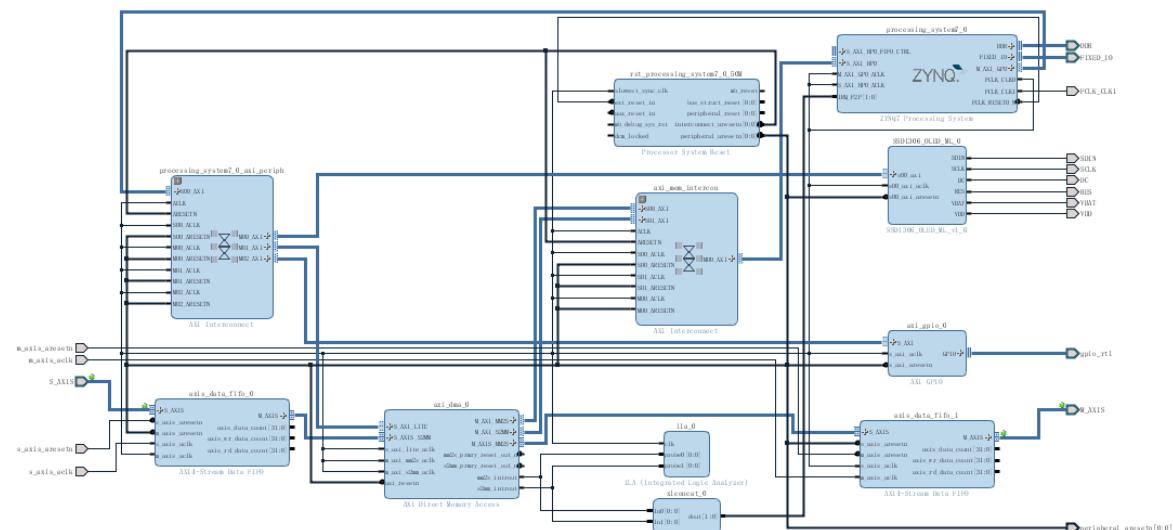
本课程的设计原理分析。

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

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

2.2 系统构架框图

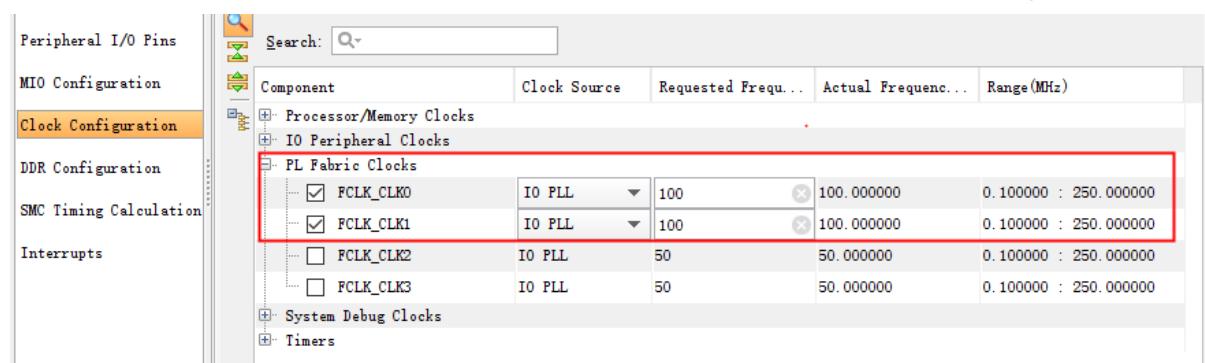
MiZ7035 系列构架框图：



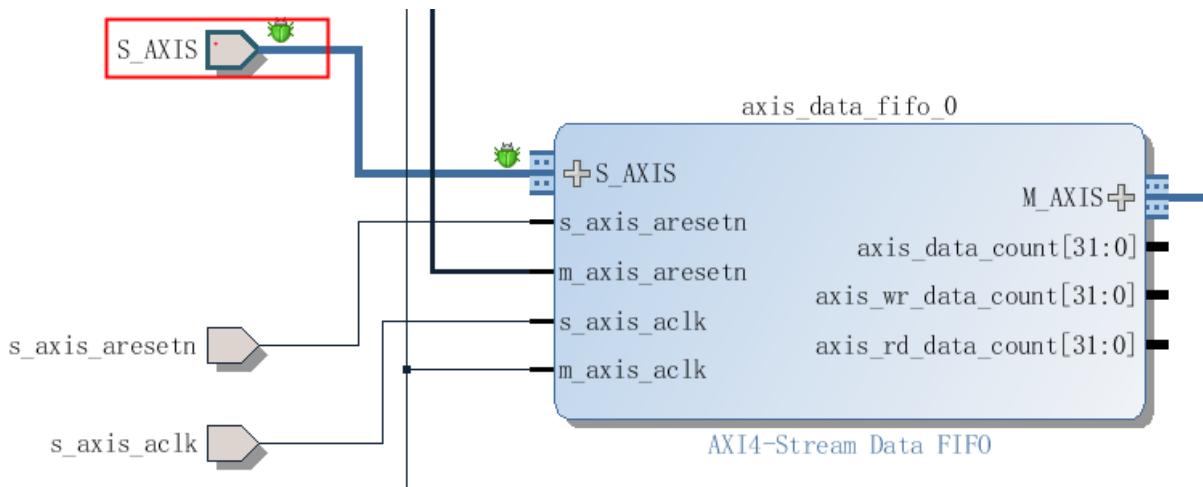
下面看下关键模块的设置

2.2.1 ZYNQ IP 的设置

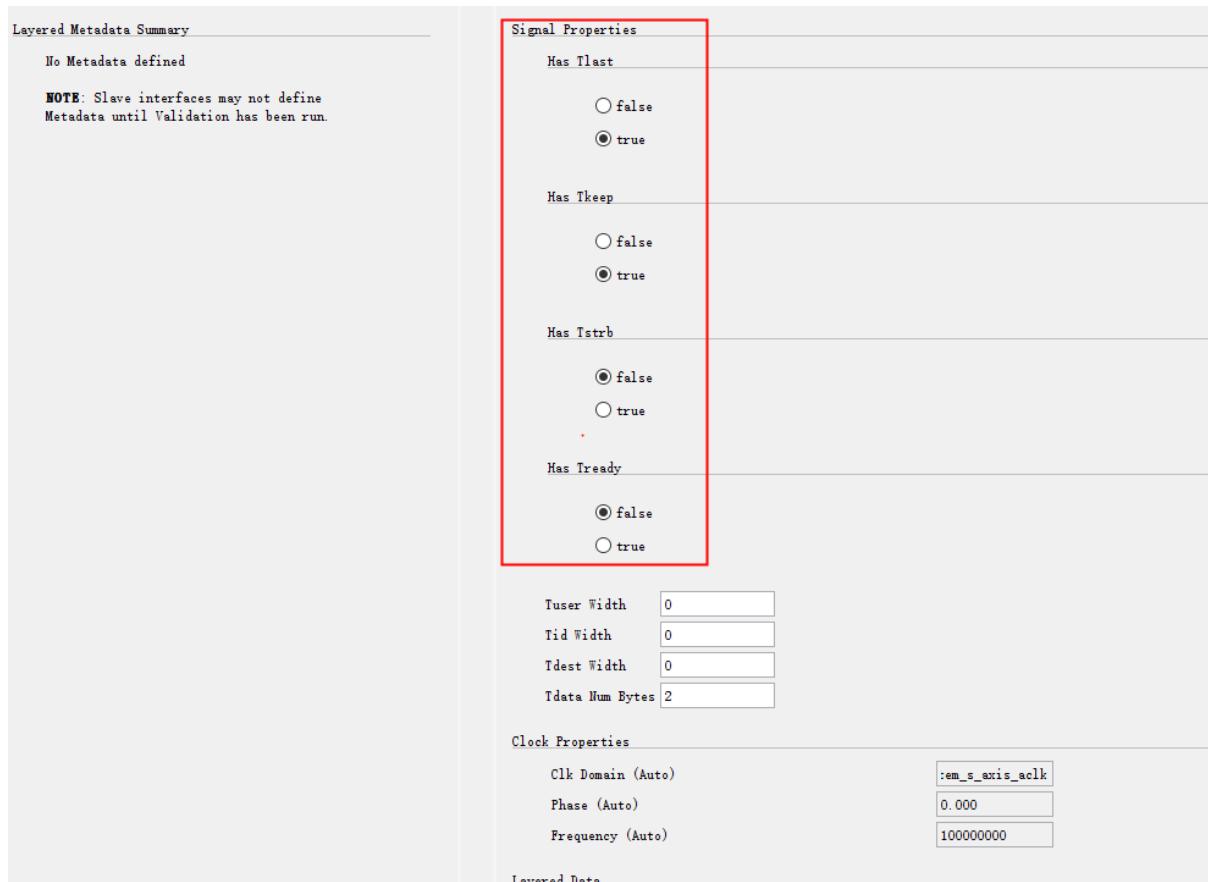
增一路 FCK_CLK1 为 100MHZ（也可以设置其他频率）并且引出到外部提供 verilog 编程时钟。



双击 S_AXIS 设置参数

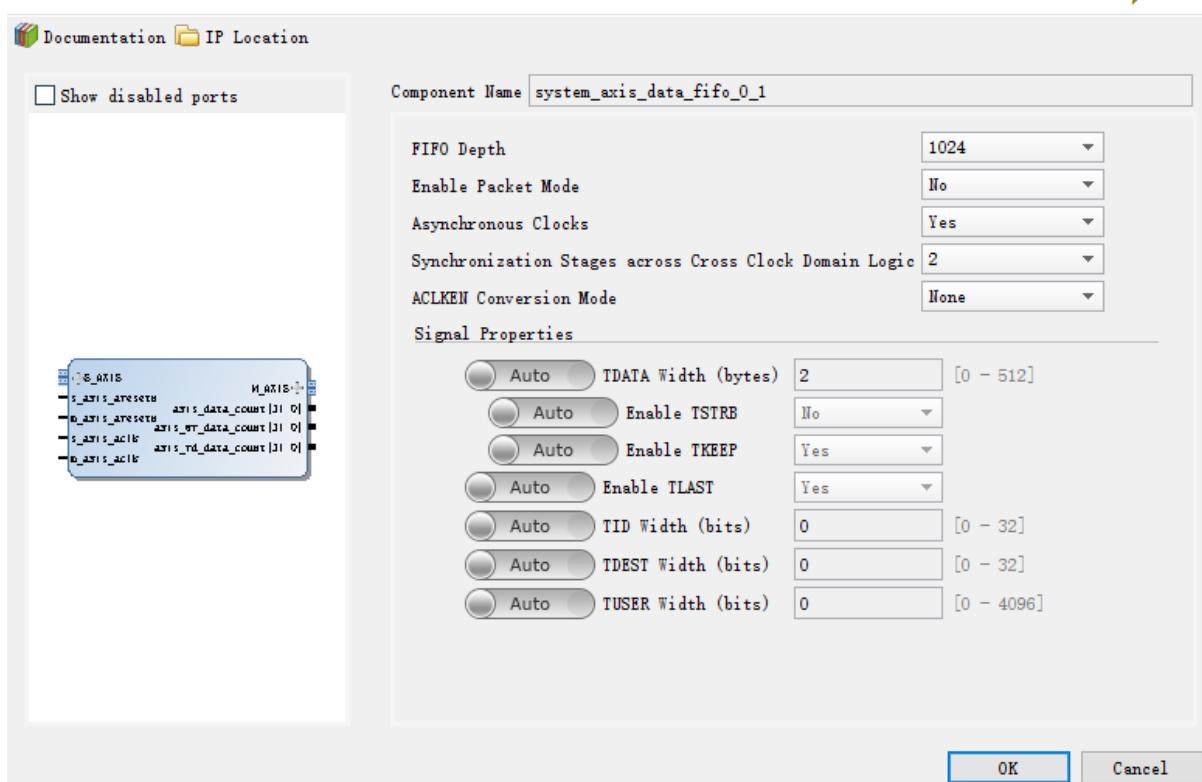


设置如下



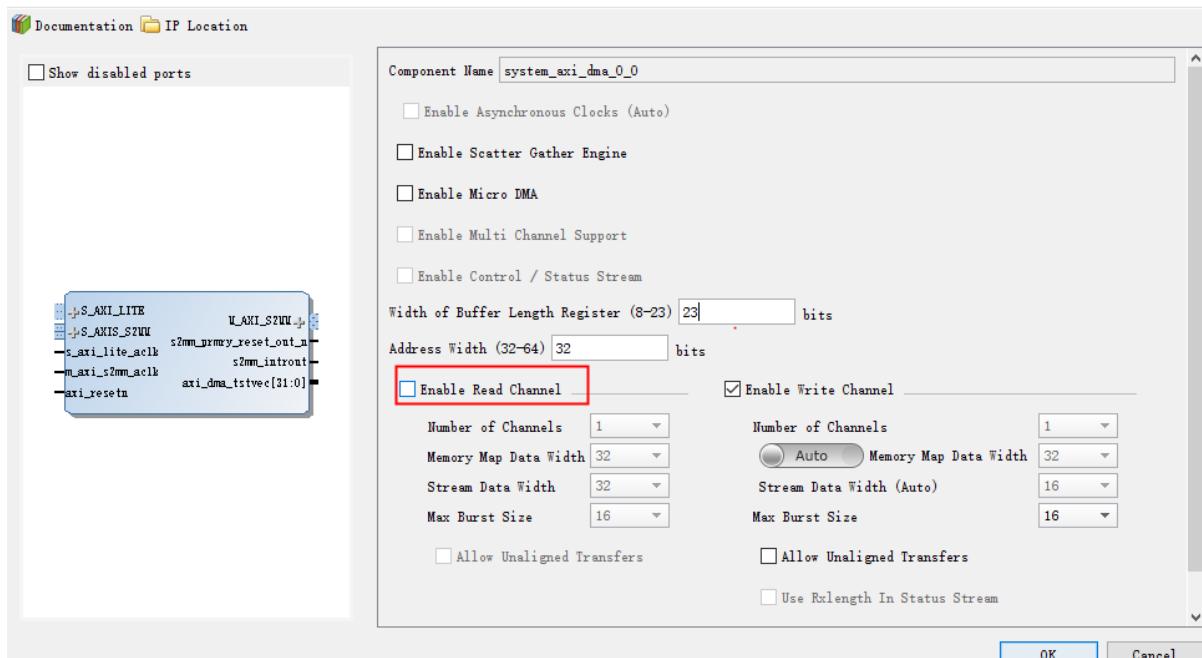
双击 FIFO 进行如下设置

AXI4-Stream Data FIFO (1.1)

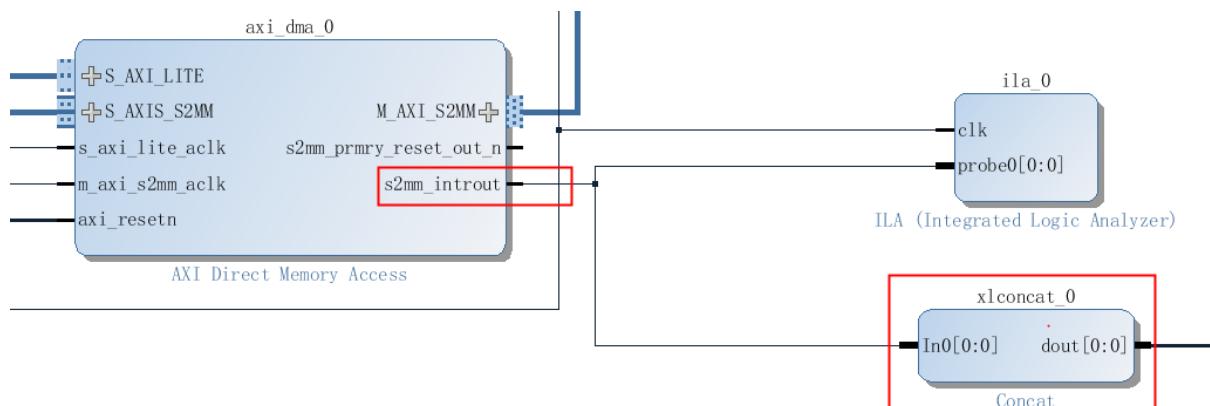


由于只有写 DMA 通道，因此不用勾选读 DMA 通道

AXI Direct Memory Access (7.1)



既然只用到了 DMA 写通道，也就只要使用 1 路中断资源。



2.3 PS 部分

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

中断初始化函数，如下

表 1-3-1 init_intr_sys 函数

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

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

表 1-3-2 测试代码

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

if(TxDone)
{
    TxDone=0;
    TX_ready=1;
    TX_success++;
}
```

```
}

if(usec==2)
{

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

    speed_rx = MAX_PKT_LEN*RX_success/1024/1024;

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

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

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

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

    RX_success=0;
    TX_success=0;

}
```

定时器中断在第二季《S02_CH08_ ZYNQ 定时器中断实验》已经详细讲解过，至于 DMA 中断《S03_CH01_AXI_DMA_LOOP 环路测试》中也已经详细讲解，不在过多复述。

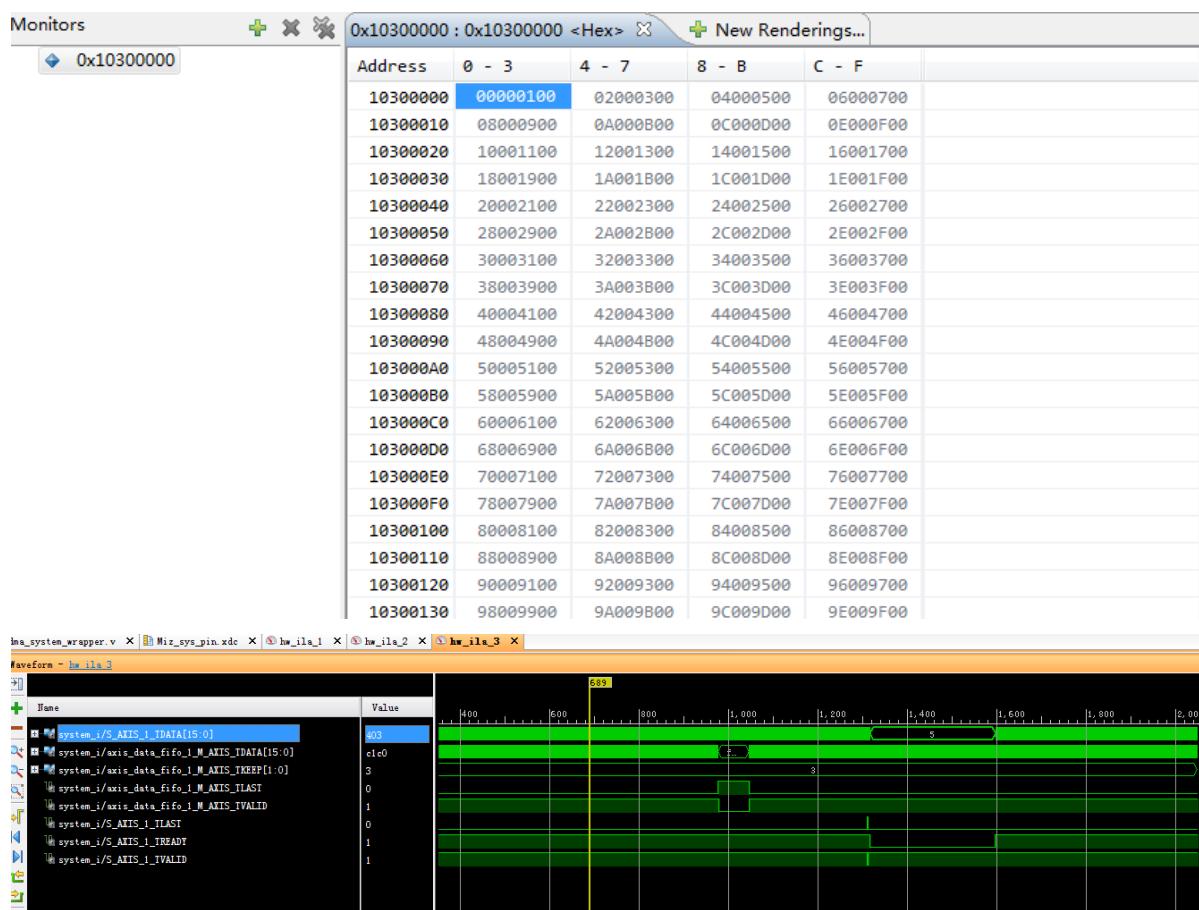
2.4 测试结果

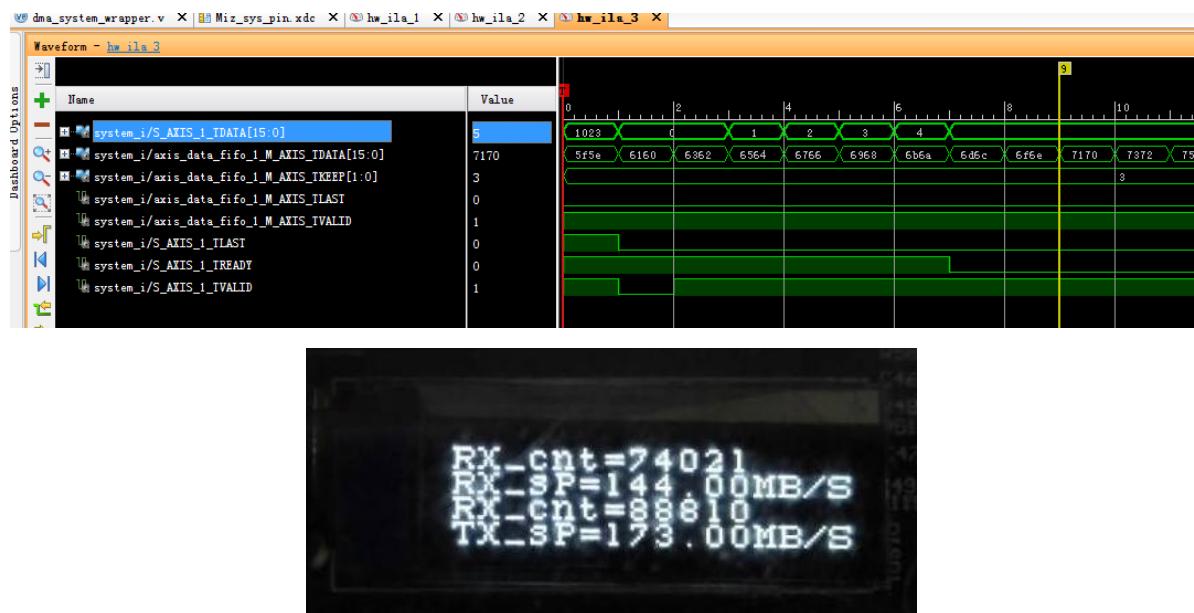
Connected to: Serial (COM69, 115200, 0, 8)

```

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

```





CH03_AXI_DMA_OV7725 摄像头采集系统

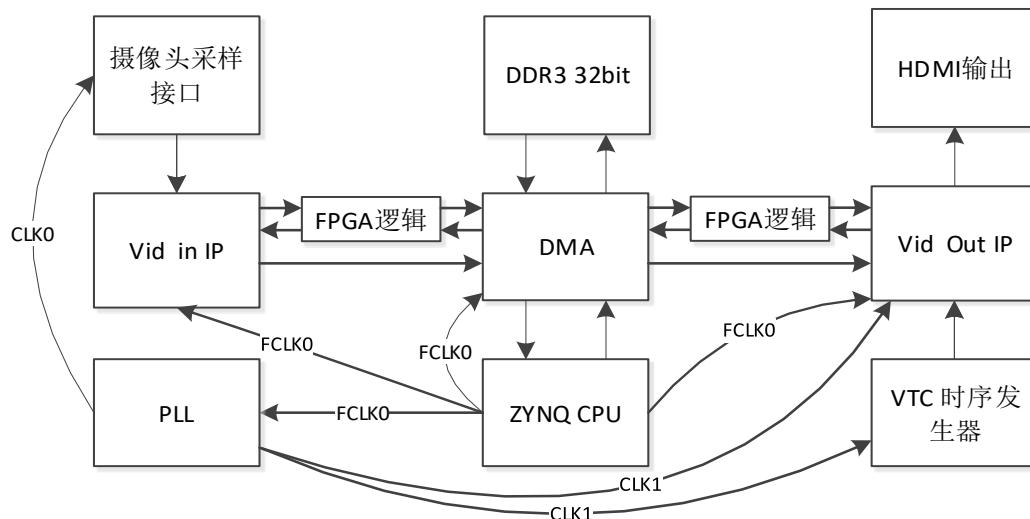
3.1 概述

本课程讲解如何搭建基于 DMA 的图形系统，方案原理如下。

摄像头采样图像数据后通过 DMA 送入到 DDR，在 PS 部分产生 DMA 接收中断，在接收中断里面再把 DDR 里面保持的图形数据 DMA 发送出去。在 FPGA 的接收端口部分产生 VID OUT 时序驱动 HDMI 显示器显示图形。

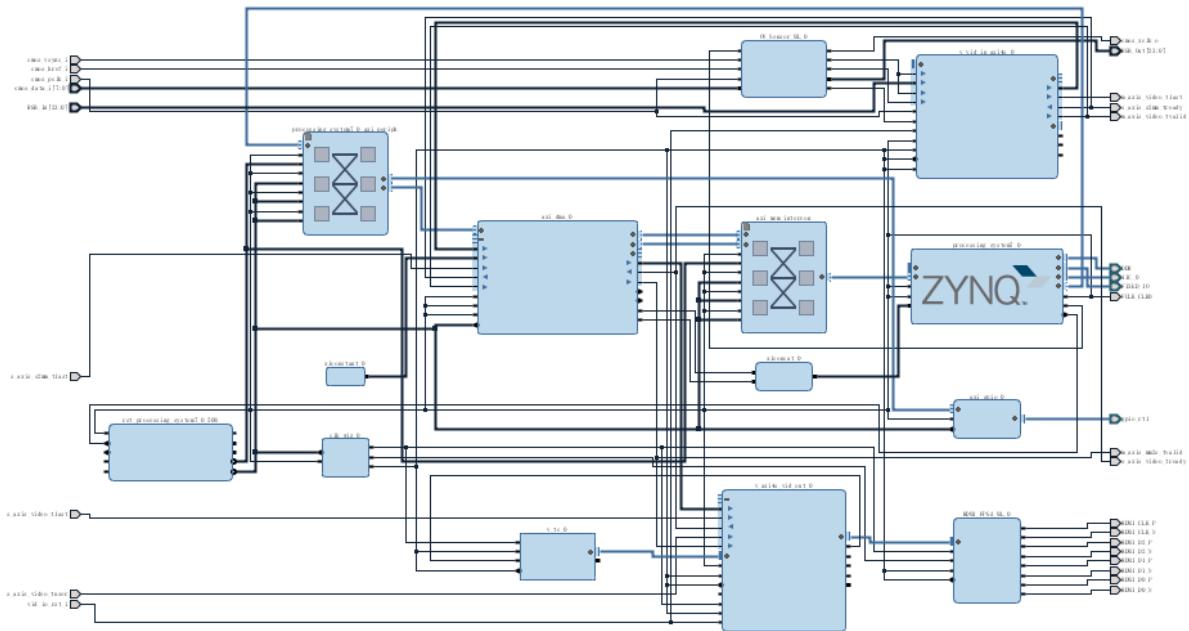
3.2 系统构架

3.2.1 构架方案图



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

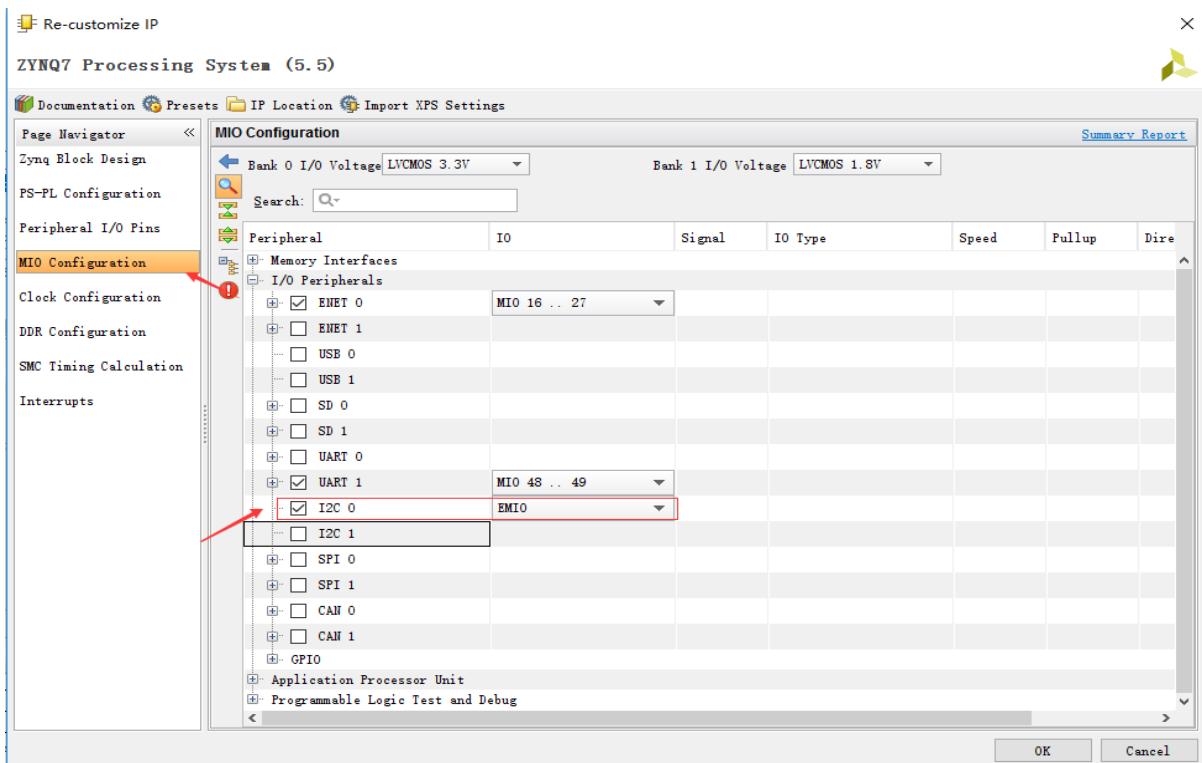
3.2.2 构 BLOCK 模块化设计方案图



3.3 IIC 接口

在 CMOS 摄像头中，有一组用于配置摄像头寄存器的总线名为 SCCB（串行摄像机控制总线）总线，其实这就是我们经常用到的 IIC 总线，在本节当中，使用到一组 IIC 总线来对摄像头进行配置，开启 IIC 总线的方法如下所示：

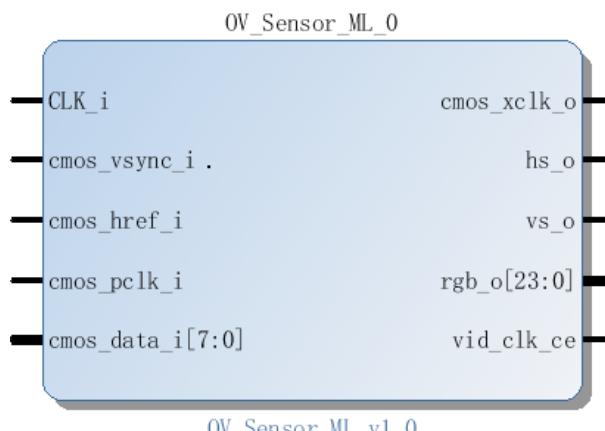
在 ZYNQ Processing system 中自带了很多常用的接口，其中就包括了两组 IIC 接口，所以此处我们只要双击 ZYNQ Processing System，然后如下图所示设置：



设置好了之后按 OK 完成设置，然后将 IIC 接口印出来即可。

3.4 vid in IP 介绍

3.4.1 OV_Sensor_ML 自定义 IP 模块



外部信号接口说明：

CLK_i : 为输入时钟，通常接 24MHZ 或者 25MHZ

Cmos_xclk_o:摄像头工作，通常直接把 CLK_i 连接到 cmos_xclk_o

Cmos_vsync_i: 摄像头场同步输入 上升沿代表场同步开始

Cmos_href_i:摄像头行同步输入 高电平代表行数据有效

Cmos_data[7:0]:摄像头数据输入

Hs_o:采集 OV_Sensor_ML IP 输出的行数据有效

Vs_o:采集 OV_Sensor_ML IP 输出的场同步信号

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

OV_Sensor_ML IP 包含 3 个源程序文件，分别为 OV_Sensor_ML.v、cmos_decode_v1.v、count_reset_v1.v 文件。

OV_Sensor_ML.v 程序中，对 cmos_data_i、cmos_href_i、cmos_vsync_i 做了一次寄存器，笔者发现图像效果有所改观。笔者分析，是因为寄存后有利于去除一些毛刺信号，提高了数据的稳定性。
表 3-3-1-1 OV_Sensor_ML 源码 OV_Sensor_ML.v

```
`timescale 1ns / 1ps
///////////////////////////////
// Company: milinker
// Engineer:tangjinyuan
//
// Create Date:    15:54:59 11/21/2015
// Design Name:
// Module Name:   OV7725_IP_ML
// Project Name: OV7725_IP_ML
// Target Devices: ZYNQ
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////
module OV_Sensor_ML(
    input CLK_i,
    //----- CMOS sensor hardware interface -----
    input      cmos_vsync_i, //cmos vsync
    input      cmos_href_i, //cmos hsync refrence
    input      cmos_pclk_i, //cmos pxiel clock
    output     cmos_xclk_o, //cmos externl clock
    input[7:0]cmos_data_i, //cmos data
    output hs_o, //hs signal.
    output vs_o, //vs signal.
    // output de_o, //data enable.
    output [23:0]rgb_o, //data output,
    output vid_clk_ce
);
```

```
//-----视频输出解码模块-----//
wire [15:0]rgb_o_r;
assign rgb_o = {rgb_o_r[4:0] ,3'd0 ,rgb_o_r[10:5] ,2'd0,rgb_o_r[15:11],3'd0};

reg [7:0]cmos_data_r;
reg cmos_href_r;
reg cmos_vsync_r;

always@(posedge cmos_pclk_i)
begin
    cmos_data_r <= cmos_data_i;
    cmos_href_r <= cmos_href_i;
    cmos_vsync_r<= cmos_vsync_i;
end
//assign rgb_o = 24'b11111111_00000000_11111111;
cmos_decode cmos_decode_u0(
    //system signal.
    .cmos_clk_i(CLK_i),//cmos senseor clock.
    .rst_n_i(RESETn_i2c),//system reset.active low.
    //cmos sensor hardware interface.
    .cmos_pclk_i(cmos_pclk_i),//(cmos_pclk),//input pixel clock.
    .cmos_href_i(cmos_href_r),//(cmos_href),//input pixel hs signal.
    .cmos_vsync_i(cmos_vsync_r),//(cmos_vsync),//input pixel vs signal.
    .cmos_data_i(cmos_data_r),//(cmos_data),//data.
    .cmos_xclk_o(cmos_xclk_o),//(cmos_xclk),//output clock to cmos sensor.
    //user interface.
    .hs_o(hs_o),//hs signal.
    .vs_o(vs_o),//vs signal.
    // .de_o(de_o),//data enable.
    .rgb565_o(rgb_o_r),//data output
    .vid_clk_ce(vid_clk_ce)
);

count_reset_v1#(
    .num(20'hffff0)
)(
    .clk_i(CLK_i),
    .rst_o(RESETn_i2c)
);

endmodule
```

1 cmos_decode_v1.v 是本模块的关键部分, 实现了 RGB565 的解码输出以及 vid_clk_ce 实现了此

模块和 vid_in IP 直接时序匹配的关系。

表 3-3-1-2 OV_Sensor_ML 源码 cmos_decode_v1.v

```
`timescale 1ns / 1ps
///////////////////////////////
// Company: milinker corporation
// WEB:www.milinker.com
// BBS:www.osrc.cn
// Engineer:..
// Create Date: 07:28:50 09/04/2015
// Design Name: cmos_decode_v1
// Module Name: cmos_decode_v1
// Project Name: cmos_decode_v1
// Target Devices: XC6SLX25-FTG256 Mis603
// Tool versions: ISE14.7
// Description: cmos_decode_v1.
// Revision: V1.0
// Additional Comments:
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low
//4) _dg debug signal
//5) _r reg delay
//6) _s state machine
/////////////////////////////
module cmos_decode(
    //system signal.
    input cmos_clk_i,//cmos senseor clock.
    input rst_n_i,//system reset.active low.
    //cmos sensor hardware interface.
    input cmos_pclk_i,//input pixel clock.
    input cmos_href_i,//input pixel hs signal.
    input cmos_vsync_i,//input pixel vs signal.
    input[7:0]cmos_data_i,//data.
    output cmos_xclk_o,//output clock to cmos sensor.
    //user interface.
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    output reg [15:0]rgb565_o,//data output
    output vid_clk_ce
);
parameter[5:0]CMOS_FRAME_WAITCNT = 4'd15;

reg[4:0]rst_n_reg = 5'd0;
//reset signal deal with.
```

```
always@(posedge cmos_clk_i)
begin
    rst_n_reg <= {rst_n_reg[3:0],rst_n_i};
end

reg[1:0]vsync_d;
reg[1:0]href_d;
wire vsync_start;
wire vsync_end;
//vs signal deal with.
always@(posedge cmos_pclk_i)
begin
    vsync_d <= {vsync_d[0],cmos_vsync_i};
    href_d <= {href_d[0],cmos_href_i};
end

assign vsync_start = vsync_d[1]&(!vsync_d[0]);
assign vsync_end = (!vsync_d[1])&vsync_d[0];

reg[6:0]cmos_fps;
//frame count.
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        begin
            cmos_fps <= 7'd0;
        end
    else if(vsync_start)
        begin
            cmos_fps <= cmos_fps + 7'd1;
        end
    else if(cmos_fps >= CMOS_FRAME_WAITCNT)
        begin
            cmos_fps <= CMOS_FRAME_WAITCNT;
        end
end
//wait frames and output enable.
reg out_en;
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        begin
            out_en <= 1'b0;
        end
```

```
else if(cmos_fps >= CMOS_FRAME_WAITCNT)
begin
    out_en <= 1'b1;
end
else
begin
    out_en <= out_en;
end
end

//output data 8bit changed into 16bit in rgb565.
reg [7:0] cmos_data_d0;
reg [15:0] cmos_rgb565_d0;
reg byte_flag;

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        byte_flag <= 0;
    else if(cmos_href_i)
        byte_flag <= ~byte_flag;
    else
        byte_flag <= 0;
end

reg byte_flag_r0;
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        byte_flag_r0 <= 0;
    else
        byte_flag_r0 <= byte_flag;
end

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        cmos_data_d0 <= 8'd0;
    else if(cmos_href_i)
        cmos_data_d0 <= cmos_data_i; //MSB -> LSB
    else if(~cmos_href_i)
        cmos_data_d0 <= 8'd0;
end
```

```

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        rgb565_o <= 16'd0;
    else if(cmos_href_i&byte_flag)
        rgb565_o <= {cmos_data_d0,cmos_data_i}; //MSB -> LSB
    else if(~cmos_href_i)
        rgb565_o <= 8'd0;
end

assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
assign vs_o = out_en ? vsync_d[1] : 1'b0;
assign hs_o = out_en ? href_d[1] : 1'b0;

assign cmos_xclk_o = cmos_clk_i;

endmodule

```

count_reset_v1.v 源文件实现了信号的延迟复位。

表 3-3-1-1 count_reset_v1.v

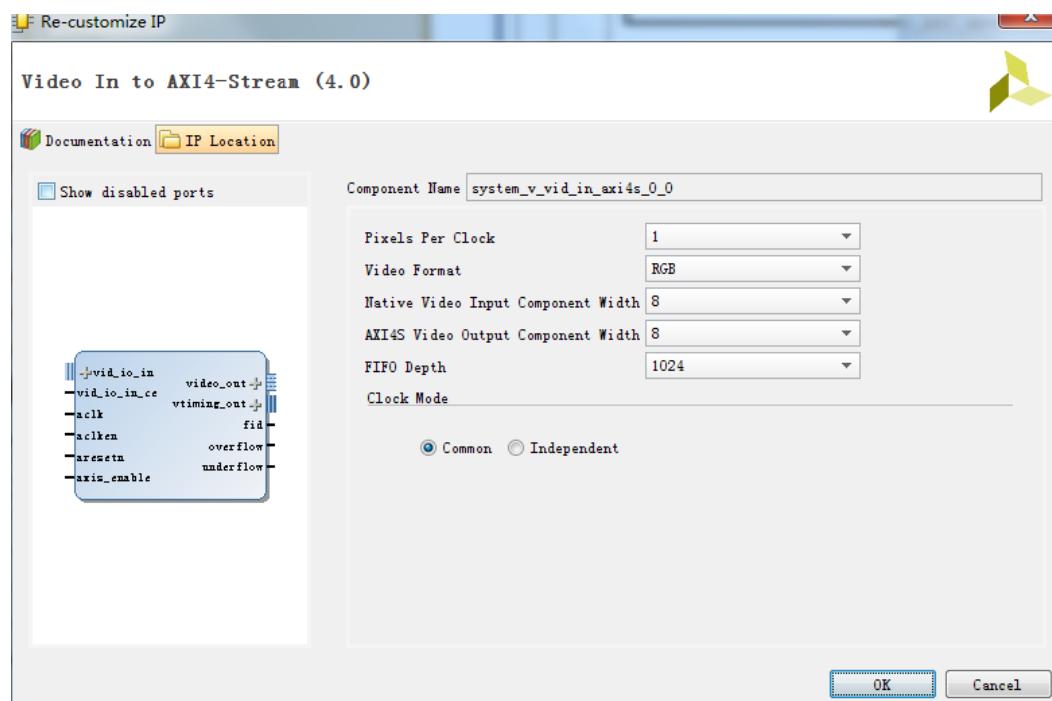
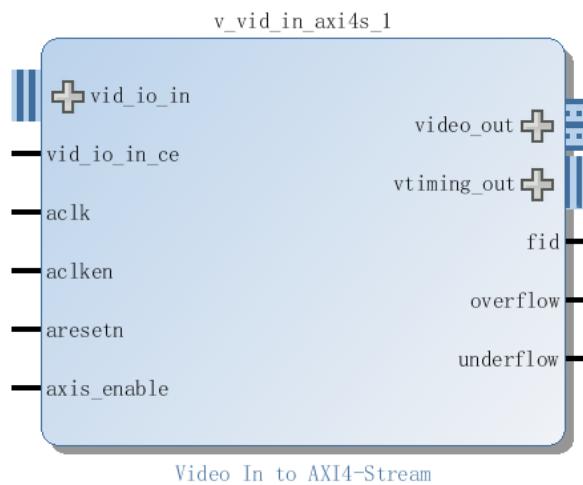
```

`timescale 1ns / 1ps
///////////////////////////////
// Company: milinker corporation
// WEB:www.milinker.com
// BBS:www.osrc.cn
// Engineer:sanliuyaoling.
// Create Date: 07:28:50 12/04/2015
// Design Name: count_reset_v1
// Module Name: count_reset_v1
// Project Name: count_reset_v1
// Target Devices: XC7Z020-CLG484-1I
// Tool versions: vivado2015.4
// Description: count_reset_v1
// Revision: V1.0
// Additional Comments:
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low
//4) _dg debug signal
//5) _r reg delay
//6) _s state machine
/////////////////////////////
module count_reset_v1#

```

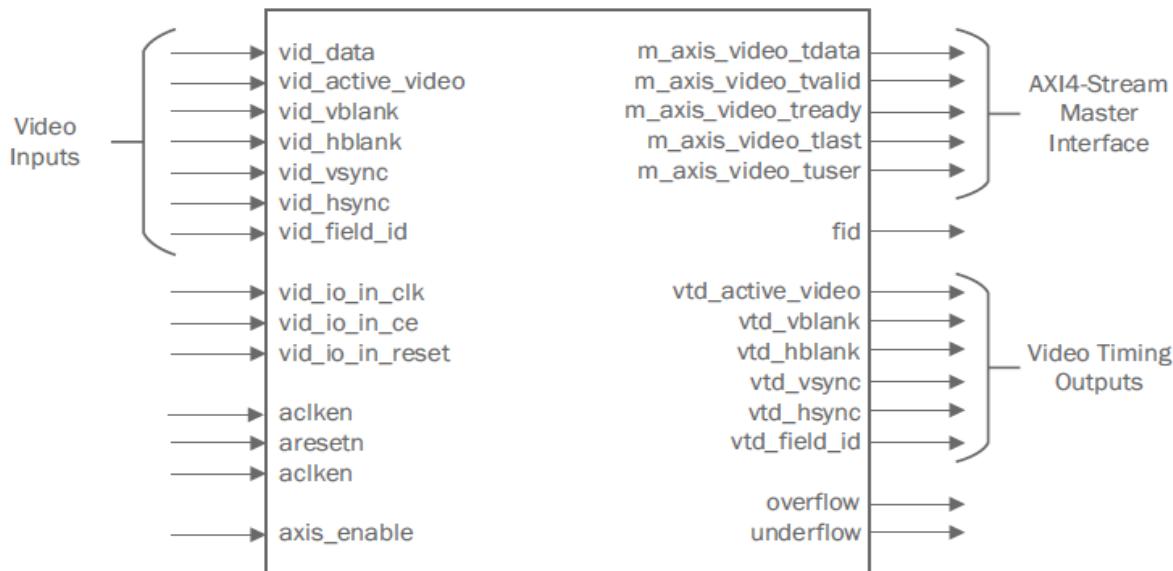
```
(  
    parameter[19:0]num = 20'hffff0  
)  
    input clk_i,  
    output rst_o  
  
reg[19:0] cnt = 20'd0;  
reg rst_d0;  
  
/*count for clock*/  
always@(posedge clk_i)  
begin  
    cnt <= ( cnt <= num)?( cnt + 20'd1 ):num;  
end  
  
/*generate output signal*/  
always@(posedge clk_i)  
begin  
    rst_d0 <= ( cnt >= num)?1'b1:1'b0;  
end  
  
assign rst_o = rst_d0;  
  
endmodule
```

3.4.2 vid in IP 模块



- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Video Format: 视频格式
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- FIFO Depth: FIFO 深度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟

3.4.2 VID_IN IP 接口信号的定义



Common Interface

Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
axis_enable	Input	1	This input should be connected to the VTC detector locked status and is synchronous to vid_io_in_clk. 1 = Enable writes into FIFO 0 = Disable writes into FIFO
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_in_clk	Input	1	Native video clock. Only available in independent clock mode.
vid_io_in_ce	Input	1	Native video clock enable
vid_io_in_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk. If an overflow occurs, this could indicate that the connected AXI4-Stream Slave is creating excessive back-pressure.
underflow	Output	1	Flag indicating that the FIFO has under-flowed. This should never occur under normal operation. Synchronous to aclk.

Video Timing Interface

Signal Name	Direction	Width	Description
vtd_vsync	Out	1	Vertical sync video timing signal.
vtd_hsync	Out	1	Horizontal sync video timing signal.
vtd_vblank	Out	1	Vertical blank video timing signal.
vtd_hblank	Out	1	Horizontal blank video timing signal.
vtd_active_video	Out	1	Active video flag. 1 = active video, 0 = blanked video
vtd_field_id	Out	1	VTC field ID. 0= even field, 1= odd field.

Video Input Interface

Signal Name	Direction	Width	Description
vid_active_video	In	1	Video data valid. 1 = active video, 0 = blanked video
vid_vsync	In	1	Vertical sync video timing signal. Active High
vid_hsync	In	1	Horizontal sync video timing signal. Active High
vid_vblank	In	1	Vertical blank video timing signal. Active High
vid_hblank	In	1	Horizontal blank video timing signal. Active High
vid_data	In	8-256	Parallel video input data.
vid_field_id	In	1	Video field. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

使用到的信号有：

Vid_in IP 输入端信号：

Vid_data: 视频数据输入

Vid_active_video: 视频数据有效

Vid_hsync: 视频行同步信号 (非常关键信号, 下面重点分析对象)

Vid_vsync: 视频场同步信号

Vid_io_in_ce: 数据输入有效 (非常关键信号, 下面重点分析对象)

vid_io_in_clk: 这是时钟信号和摄像头时钟同步

Vid_io_in_reset: 这个信号, 高电平的时候复位

有很多读者会问笔者, 这些官方的 IP 如何使用, 这么没有详细的技术手册。还别说, 官方就是没有非常详细的技术手册, 有时候笔者也是使出浑身解数分析 vid_in IP 内部信号时序, 掌握 OV_Sensor_ML 自定义 IP 时序接口设计。

打开 v_vid_in_axi4s_v4_0_1_formatter.v 这个文件

下面对其关键的部分进行说明。

表 3-3-2-1 v_vid_in_axi4s_v4_0_1_formatter.v

```
`timescale 1ps/1ps
`default_nettype none
(* DowngradeIPIdentifiedWarnings="yes" *)
```

```
module v_vid_in_axi4s_v4_0_1_formatter #(
```

```
parameter C_NATIVE_DATA_WIDTH = 24
)(
// System signals
input wire VID_IN_CLK,           // Native video clock
input wire VID_RESET,            // Native video reset
input wire VID_CE,               // Native video clock enable

// Video input signals
input wire VID_ACTIVE_VIDEO,     // Native video input data enable
input wire VID_VBLANK,           // Native video input vertical blank
input wire VID_HBLANK,           // Native video input horizontal blank
input wire VID_VSYNC,             // Native video input vertical sync
input wire VID_HSYNC,             // Native video input horizontal sync
input wire VID_FIELD_ID,         // Native video input field-id
input wire [C_NATIVE_DATA_WIDTH-1:0] VID_DATA, // Native video input data

// Video timing detector signals
output wire VTD_ACTIVE_VIDEO,    // Native video output data enable
output wire VTD_VBLANK,           // Native video output vertical blank
output wire VTD_HBLANK,           // Native video output horizontal blank
output wire VTD_VSYNC,             // Native video output vertical sync
output wire VTD_HSYNC,             // Native video output horizontal sync
output wire VTD_FIELD_ID,         // Native video output field-id
input wire VTD_LOCKED,            // Native video locked signal from VTD

// FIFO write signals
output wire [C_NATIVE_DATA_WIDTH+2:0] FIFO_WR_DATA, // FIFO write data
output wire FIFO_WR_EN            // FIFO write enable
);

// Wire and register declarations
reg de_1 = 0;
reg vblank_1 = 0;
reg hblank_1 = 0;
reg vsync_1 = 0;
reg hsync_1 = 0;
reg [C_NATIVE_DATA_WIDTH -1:0] data_1 = 0;
reg de_2 = 0;
reg v_blank_sync_2 = 0;
reg [C_NATIVE_DATA_WIDTH -1:0] data_2 = 0;
reg de_3 = 0; // DE output register
reg [C_NATIVE_DATA_WIDTH -1:0] data_3 = 0; // data output register
reg vert_blinking_intvl = 0; // SR, reset by DE rising
reg field_id_1 = 0;
```

```
reg field_id_2 = 0;
reg field_id_3 = 0;

wire v_blank_sync_1; // vblank or vsync
wire de_rising;
wire de_falling;
wire vsync_rising;
reg sof;
reg sof_1;
reg eol;
reg vtd_locked;
wire sof_rising;

// Assignments
assign FIFO_WR_DATA    = {field_id_3,sof_1,eol,data_3};
assign FIFO_WR_EN       = de_3 & ~VID_RESET & vtd_locked;
assign VTD_ACTIVE_VIDEO = de_1;
assign VTD_VBLANK      = vblank_1;
assign VTD_HBLANK      = hblank_1;
assign VTD_VSYNC        = vsync_1;
assign VTD_HSYNC        = hsync_1;
assign VTD_FIELD_ID    = field_id_1;

assign v_blank_sync_1 = vblank_1 || vsync_1;
assign de_rising   = de_1 && !de_2;
assign de_falling  = !de_1 && de_2;
assign vsync_rising = v_blank_sync_1 && !v_blank_sync_2;
assign sof_rising  = sof & ~sof_1;

// VTD locked process
always @(posedge VID_IN_CLK) begin
  if(VID_RESET | ~VTD_LOCKED) begin
    vtd_locked <= 1'b0;
  end else if(VID_CE) begin
    vtd_locked <= (sof_rising & VTD_LOCKED) ? 1'b1 : vtd_locked;
  end
end

// input, output, and delay registers
always @ (posedge VID_IN_CLK) begin
  if(VID_RESET) begin
    de_1      <= 1'b0;
    de_2      <= 1'b0;
    de_3      <= 1'b0;
```

```
vblank_1      <= 1'b0;
hblank_1      <= 1'b0;
vsync_1       <= 1'b0;
hsync_1       <= 1'b0;
field_id_1    <= 1'b0;
field_id_2    <= 1'b0;
field_id_3    <= 1'b0;
data_1        <= {C_NATIVE_DATA_WIDTH{1'b0}};
data_2        <= {C_NATIVE_DATA_WIDTH{1'b0}};
data_3        <= {C_NATIVE_DATA_WIDTH{1'b0}};
v_blank_sync_2 <= 1'b0;
eol           <= 1'b0;
sof            <= 1'b0;
sof_1          <= 1'b0;
end else if(VID_CE) begin
    de_1      <= VID_ACTIVE_VIDEO;
    de_2      <= de_1;
    de_3      <= de_2;
    vblank_1   <= VID_VBLANK;
    hblank_1   <= VID_HBLANK;
    vsync_1    <= VID_VSYNC;
    hsync_1    <= VID_HSYNC;
    field_id_1 <= VID_FIELD_ID;
    field_id_2 <= field_id_1;
    field_id_3 <= field_id_2;
    data_1     <= VID_DATA;
    data_2     <= data_1;
    data_3     <= data_2;
    v_blank_sync_2 <= v_blank_sync_1;
    eol         <= de_falling;
    sof          <= de_rising && vert_blinking_intvl;
    sof_1        <= sof;
end
end

// Vertical back porch SR register
always @ (posedge VID_IN_CLK) begin
    if (VID_CE) begin
        if (vsync_rising) // falling edge of vsync
            vert_blinking_intvl <= 1;
        else if (de_rising) // rising edge of data enable
            vert_blinking_intvl <= 0;
    end
end
```

```
endmodule
```

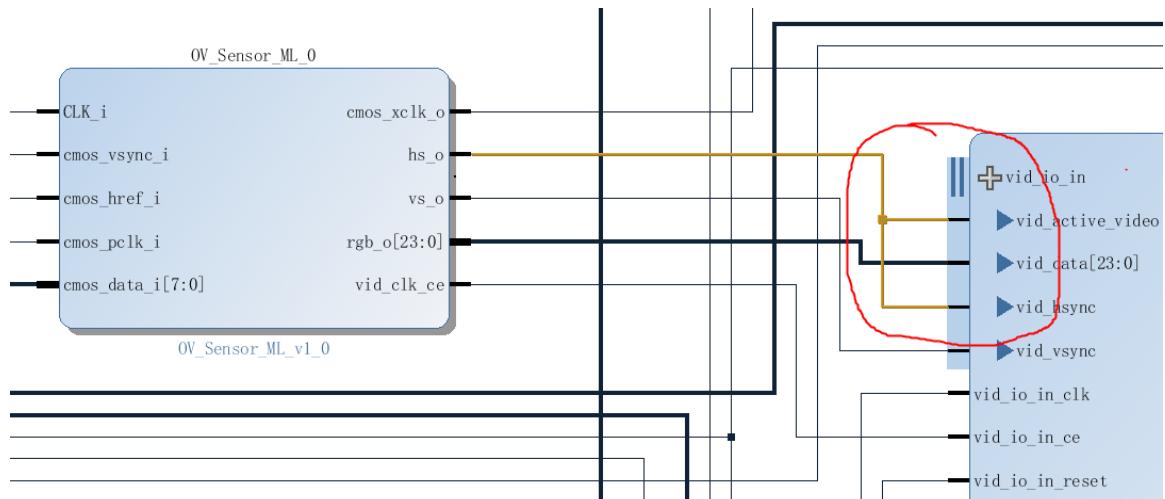
在上面代码中，

```
eol      <= de_falling;
sof      <= de_rising && vert_blanking_intvl;
```

eol 实际就是 tlast 信号, 而 sof 就是 tuser 信号。tlast 信号代表每行图像数据的最后一个数据, tuser 代表每场数据的第一个数据。

所有非常关键的信号都和 de_falling 和 vert_blanking_intvl 有关系。

hs_o 和 vid_in IP 的连接关系。



上图中, 被红色圈起来的 hs_o 信号, 同时接到了 vid_in_ip 的 vid_active_video 和 vid_hsync 信号接口。因此, de 信号就是 hs_o 信号, 而 vid_hsync 我们发现没有任何作用, 也就是说不 hs_o 不连接到 vid_hsync 也不影响这里的程序工作。

VID_CE 这个参数就是前面的 vid_io_in_ce 信号, 可以看出这个芯片有效的时候相对应的时序电路才会执行。在本工程中, 摄像头每 2 个 pc1k 输出 1 个有效的数据, 而 vid_in IP 如果 VID_CE 为 1 则数据输入会每个时钟输入 1 个就错了。因此官方的 IP 设计的还是很不错考虑周到, 通过 VID_CE 这个条件, 控制时钟同步。

```
...
else if(VID_CE) begin
...
end
...
```

现在回到 OV_Sensor_ML 的 cmos_decode_v1.v 文件中有一段红色的代码如下:

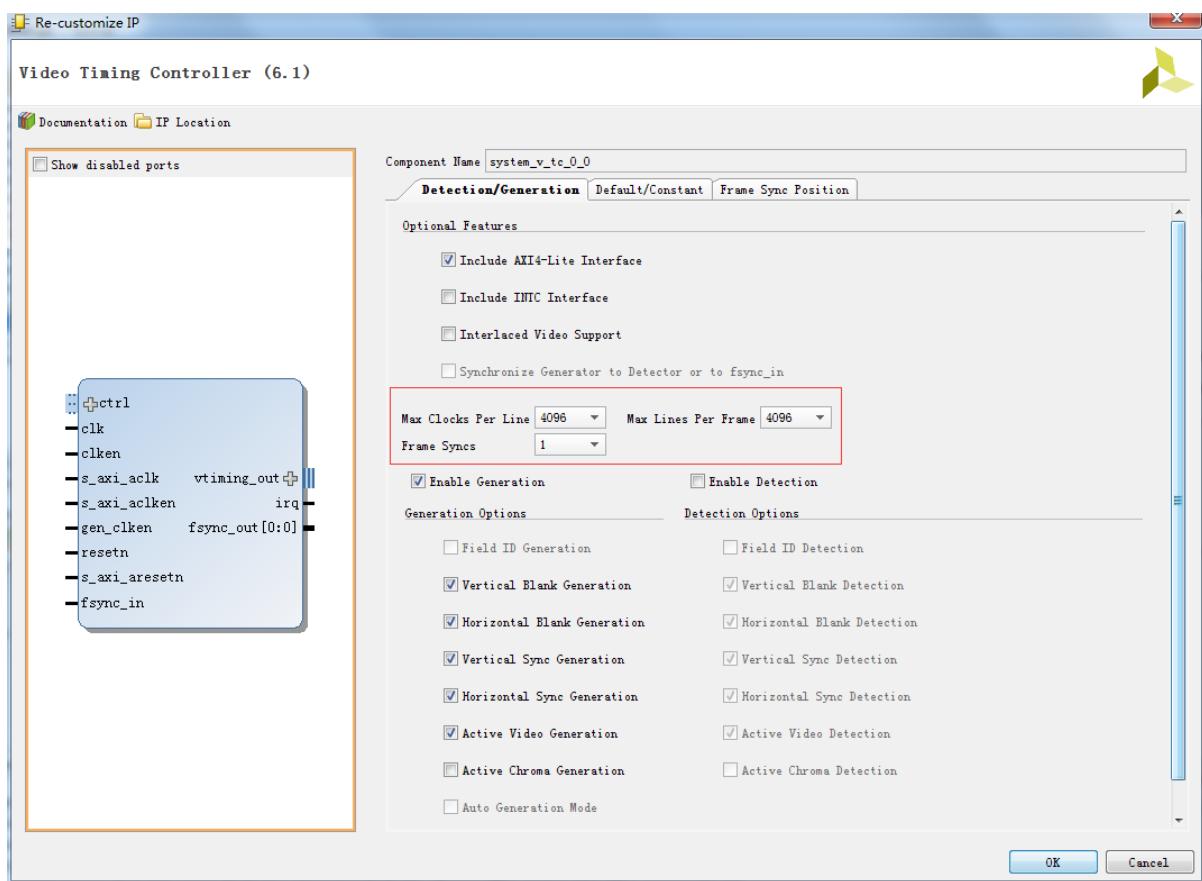
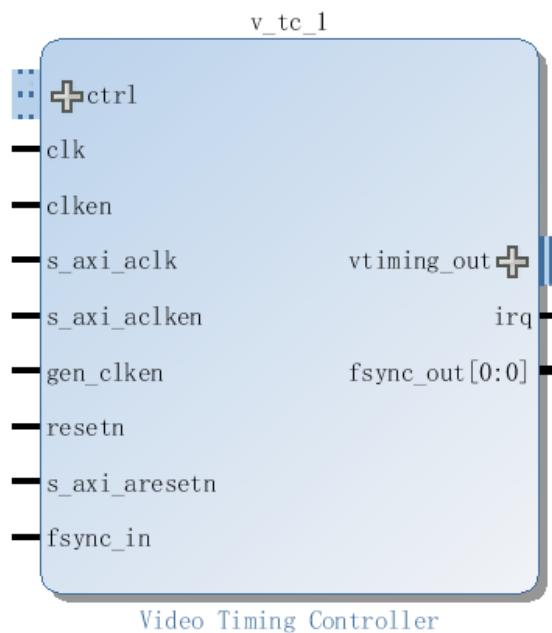
```
assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
```

这段代码控制了 vid_clk_ce 的正确输出, 关键部分是 `(byte_flag_r0&hs_o)||(!hs_o)`。当 hs_o 有效的时候, vid_in 的 VID_CE 信号就有效, 当 hs_o=0 的时候 VID_CE 必须仍然有效, 这样才能检测到 vsync_rising 信号了, 检测到了 vsync_rising 才能有 vert_blanking_intvl 为 1, 才有 tuser 信号。

好了罗嗦了半天, 终于解释完了, 如果有不清楚的, 找我们技术支持吧。

3.5 VTC IP 的分析

3.5.1 VTC IP 的参数介绍



这个 IP 就是一个时序发生器，产生显示器输出所需要的时序信号。

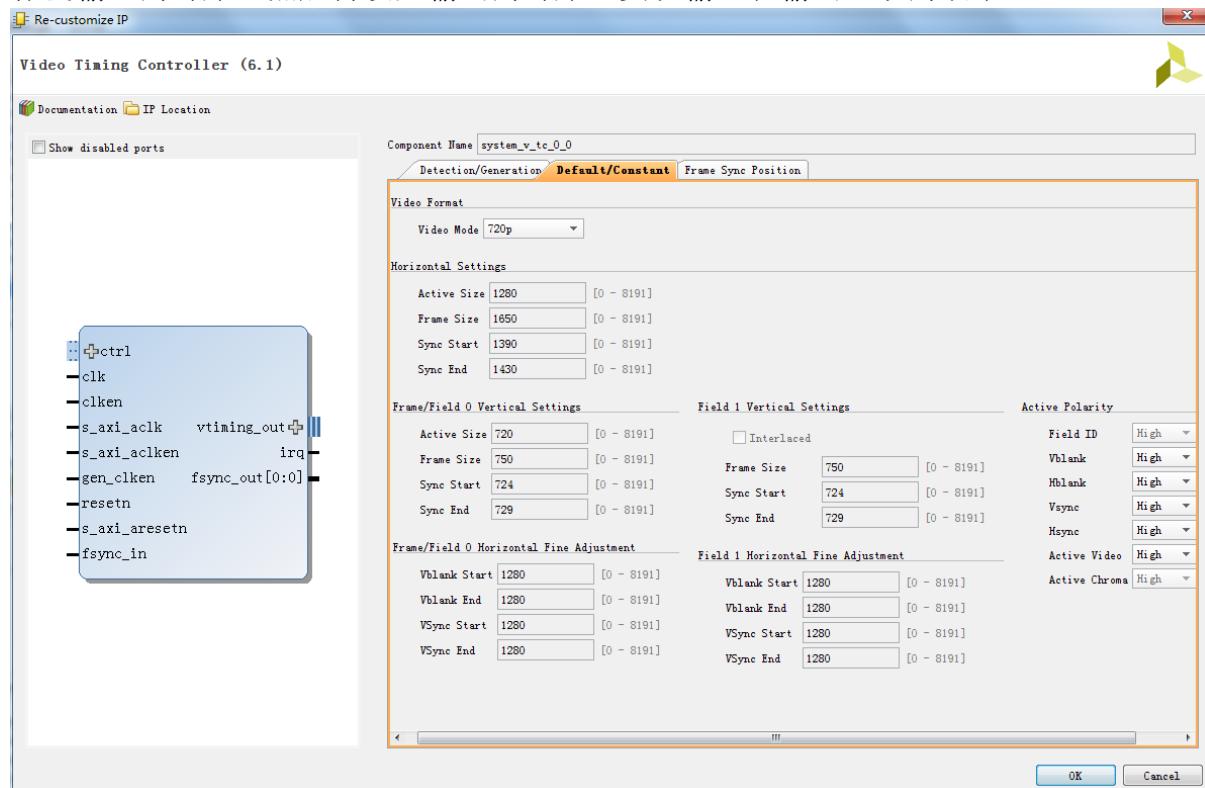
这个页面中，include AXI-lite interface 可以不勾选，不勾选就只能采用默认设置，无法在 C 语言中灵活配置了，所以笔者这里建议大家勾选吧。max clocks per line 和 max_lines per frame 需要设置下，当设置到 4096 的时候可以支持分辨率到最大，当然消耗的资源也更多。笔者这里太奢侈了设置了 4096。实际上设置到 2048 就够用了。本页面的其他信号可以采取默认设置。

Enable Generation:

支持产生时序，这个肯定必须勾选的。

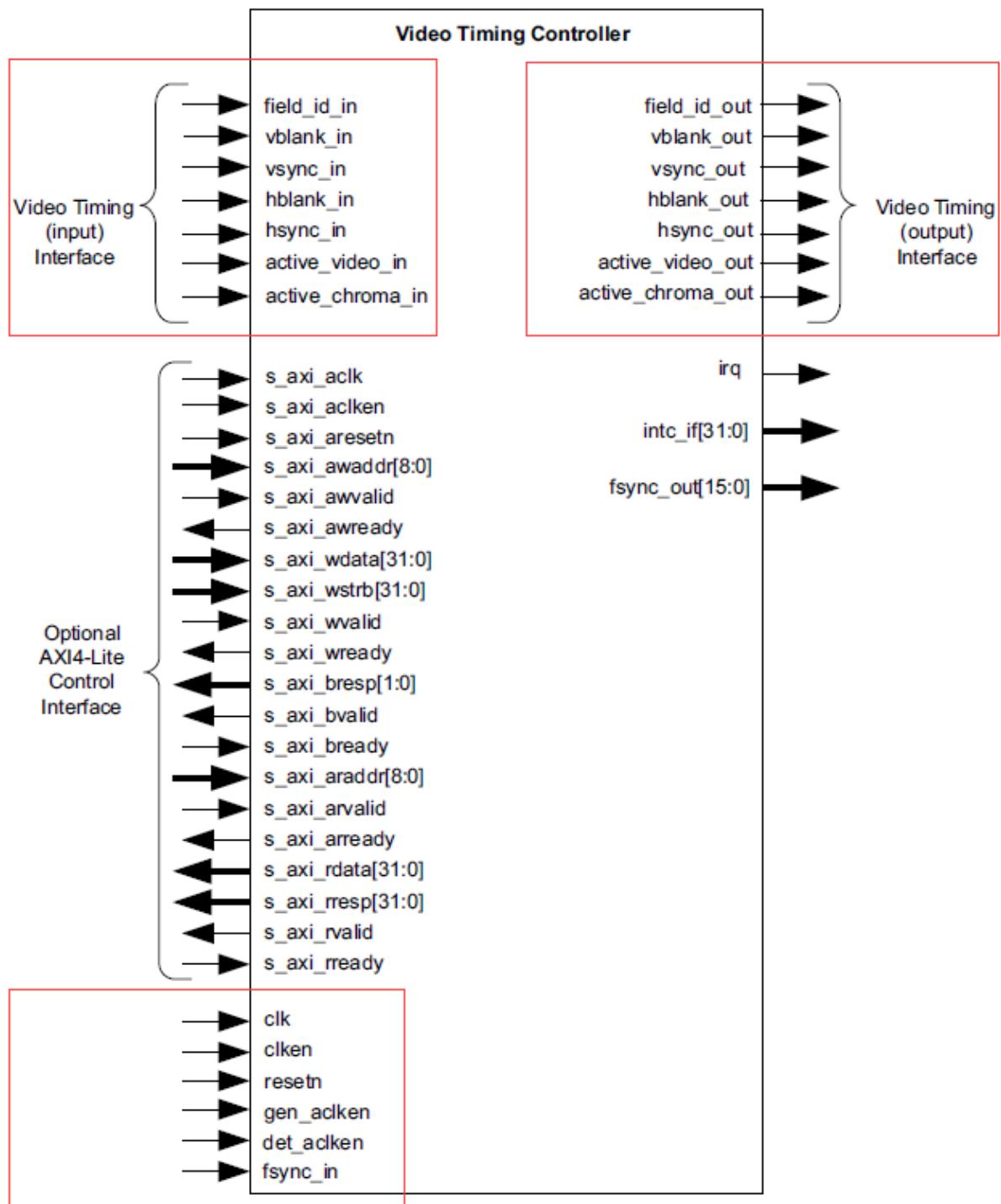
Enable Detection:

支持时序捕捉，这个不是必须的，根据需要而定，如果设置了这个选项，就可以先捕捉输入的时序，然后再设置输出的时序，实现输入和输出一致的效果。



在这个页面中，只要选择需要支持的分辨率就可以了，当然不设置也没关系的，因为我们在 C 代码综合那个会进一步设置的。

3.5.2 VTC IP 接口信号的定义



红色方框内的绝大部分信号需要我们手动联系，所以下面重点是讲解红色方框内的信号作用，至于 AXI4-LITE 接口主要是用来设置参数的。

Common Port Descriptions:

Name	Direction	Width	Description
clk	In	1	Video Core Clock
clken	In	1	Video Core Active High Clock Enable
det_clken	In	1	Video Timing Detection Core Active High Clock Enable
gen_clken	In	1	Video Timing Generator Core Active High Clock Enable
resetn	In	1	Video Core Active Low Synchronous Reset
irq	Output	1	Interrupt request output, active high edge
intc_if	Output	32	OPTIONAL EXTERNAL INTERRUPT CONTROLLER INTERFACE Available when the "Include INTC Interface" or C_HAS_INTC_IF has been selected. Bits [31:8] are the same as the bits [31:8] in the status register (0x0004). Bits [5:0] are the same as bits [21:16] of the error register (0x0008). Bits [7:6] are reserved and are always 0.

Detector Interface (Video Timing Input Interface)			
field_id_in	Input	1	INPUT FIELD ID Used to set the field_id polarity in the Detector Polarity Register (Address Offset 0x002C). Optional. Only valid when interlace support and field id are enabled.
hsync_in	Input	1	INPUT HORIZONTAL SYNCHRONIZATION Used to set the DETECTOR HSYNC register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hsync_in input is not connected, then the "Horizontal Sync Detection" option must be deselected.
hblank_in	Input	1	INPUT HORIZONTAL BLANK Used to set the DETECTOR HSIZE register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hblank_in input is not connected, then the "Horizontal Blank Detection" option must be deselected.
vsync_in	Input	1	INPUT VERTICAL SYNCHRONIZATION Used to set the DETECTOR F0_VSYNC_V and the F0_VSYNC_H registers. Polarity is auto-detected. Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vsync_in input is not connected, then the "Vertical Sync Detection" option must be deselected.

Name	Direction	Width	Description
vblank_in	Input	1	<p>INPUT VERTICAL BLANK Used to set the DETECTOR_VSIZE and the F0_VBLANK_H registers. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vblank_in input is not connected, then the "Vertical Blank Detection" option must be deselected.</p>
active_video_in	Input	1	<p>INPUT ACTIVE VIDEO Used to set the DETECTOR ACTIVE_SIZE register. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the active_video_in input is not connected, then the "Active Video Detection" option must be deselected.</p>
active_chroma_in	Input	1	<p>INPUT ACTIVE CHROMA Used to set the VIDEO_FORMAT and the CHROMA_PARITY bits in the Detector Encoding Register. Polarity is auto-detected.</p> <p>Optional. If the active_chroma_in input is not connected, then the "Active Chroma Detection" option must be deselected.</p>

Generator Interface (Video Timing Output Interface)			
field_id_out	Output	1	<p>OUTPUT FIELD ID Generated field id signal. Polarity configured by the Generator Polarity Register (Address Offset 0x006C) Optional. Only enabled when interlaced support and field id generation is enabled.</p>
hsync_out	Output	1	<p>OUTPUT HORIZONTAL SYNCHRONIZATION Generated horizontal synchronization signal. Polarity configured by the control register. Asserted active during the cycle set by the HSYNC_START bits and deasserted during the cycle set by the HSYNC_END bits in the GENERATOR HSYNC register.</p>
hblank_out	Output	1	<p>OUTPUT HORIZONTAL BLANK Generated horizontal blank signal. Polarity configured by the control register. Asserted active during the cycle set by ACTIVE_HSIZEx and deasserted during the cycle set by the FRAME_HSIZEx bits in the GENERATOR HSIZEx register.</p>
vsync_out	Output	1	<p>OUTPUT VERTICAL SYNCHRONIZATION Generated vertical synchronization signal. Polarity configured by the control register. Asserted active during the line set by the F#_VSYNC_VSTART bits and deasserted during the line set by the F#_VSYNC_VEND bits in the GENERATOR F#_VSYNC_V registers.</p>

Name	Direction	Width	Description
vblank_out	Output	1	OUTPUT VERTICAL BLANK Generated vertical blank signal. Polarity configured by the control register. Asserted active during the line set by the ACTIVE_VSIZE bits and deasserted during the line set by the GENERATOR VSIZE register.
active_video_out	Output	1	OUTPUT ACTIVE VIDEO Generated active video signal. Polarity configured by the control register. Active for non blanking lines. Asserted active during the first cycle of the field/frame and deasserted during the cycle set by the GENERATOR ACTIVE_SIZE register
active_chroma_out	Output	1	OUTPUT ACTIVE CHROMA Generated active chroma signal. Denotes which lines contain valid chroma samples (used for YUV 4:2:0). Polarity configured by the GENERATOR POLARITY register. Active for non-blanking lines configured by the VIDEO_FORMAT and the CHROMA_PARITY bits in the GENERATOR Encoding Register.
Frame Synchronization Interface			
fsync_out	Output	[Frame Syncs - 1:0]	FRAME SYNCHRONIZATION OUTPUT Each Frame Synchronization bit toggles for only one clock cycle during each frame. The number of bits is configured with the Frame Syncs GUI parameter. Each bit is independently configured for horizontal and vertical clock cycle position with the Frame Sync 0-15 Config registers).
fsync_in	Input	1	FRAME SYNCHRONIZATION INPUT This is a one clock cycle pulse (active high) input. The video timing generator will be synchronized to the input if used.

本例子中没有使用到输入时序的捕捉，因此笔者下面只对用到的信号做一些介绍。

hsync_out:

产生行同步输出

hsync_out:

产生行消影

vsync_out:

产生场同步输出

vblank_out:

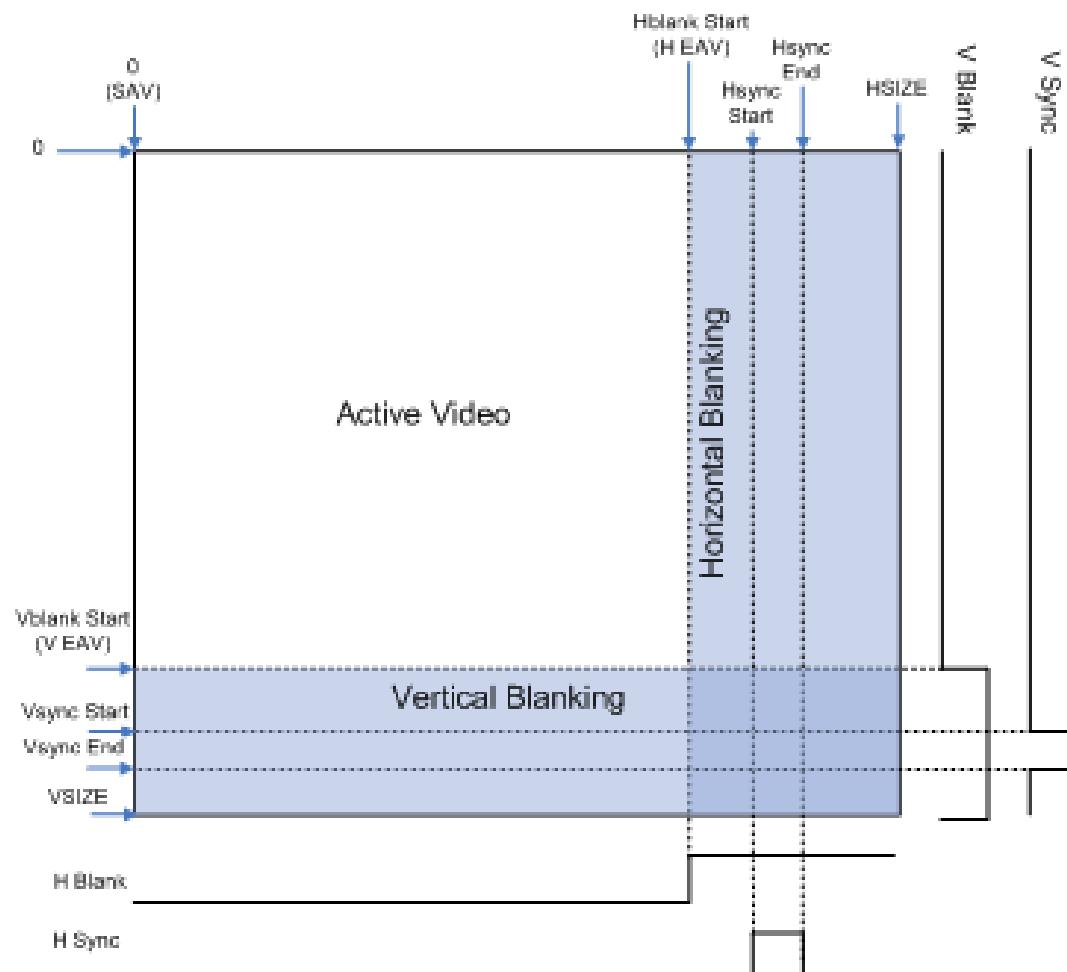
产生场消影

active_video_out:

有效数据输出

3.5.3 VTC IP 配置寄存器

shows the start of the horizontal front porch (Hblank Start), synchronization (Hsync Start), back porch (Hsync End) and active video (SAV). It also shows the start of the vertical front porch (Vblank Start), synchronization (Vsync Start), back porch (Vsync End) and active video (SAV). The total number of horizontal clock cycles is HSIZE and the total number of lines is the VSIZE.



Generator Active Size Register (Address Offset 0x0060)

0x0060	GENERATOR ACTIVE_SIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
ACTIVE_VSIZE	28:16	Generated Vertical Active Frame Size. The height of the frame without blanking in number of lines.
RESERVED	15:13	Reserved
ACTIVE_HSIZE	12:0	Generated Horizontal Active Frame Size. The width of the frame without blanking in number of pixels/clocks.

这是重要的寄存器用来设置有效的行数量和场数量

Generator Timing Status Register (Address Offset 0x0064)

0x0064	GENERATOR TIMING_STATUS	Read
Name	B its	Description
RESERVED	31:3	Reserved
GEN_ACTIVE_VIDEO	2	Generated Active Video Interrupt Status. Set high during the first cycle the output active video is asserted.
GEN_VBLANK	1	Generated Vertical Blank Interrupt Status. Set high during the first cycle the output vertical blank is asserted.
RESERVED	0	Reserved

GEN_ACTIVE_VIDEO:当第一帧图像输出时候置 1

GEN_VBLANK:第一帧有效图像的 blank 信号输出的时候置 1

Generator Encoding Register (Address Offset 0x0068)

0x0068	GENERATOR ENCODING	Read/Write
Name	B its	Description
RESERVED	31:10	Reserved
CHROMA_PARITY	9:8	Generated Chroma Parity 0: Chroma Active during even active-video lines of frame. Active every pixel of active line 1: Chroma Active during odd active-video lines of frame. Active every pixel of active line 2: Chroma Active during even active video lines of frame. Active every even pixel of active line, inactive every odd pixel 3: Chroma Active during odd active video lines of frame. Active every even pixel of active line, inactive every odd pixel
FIELD_ID_PARITY	7	Generated Field ID Parity 0: Field ID input is currently low 1: Field ID input is currently high
INTERLACED	6	Generated Progressive/Interlaced 0: Generated video format is progressive 1: Generated video format is interlaced
RESERVED	5:4	Reserved
VIDEO_FORMAT	3:0	Generated Video Format Denotes when the active_chroma signal is active. 0: YUV 4:2:2 - Active_chroma is active during the same time active_video is active. 1: YUV 4:4:4 - Active_chroma is active during the same time active_video is active. 2: RGB - Active_chroma is active during the same time active_video is active. 3: YUV 4:2:0- Active_chroma is active every other line during the same time active_video is active. See The CHROMA_PARITY bits to control which lines and pixels.

CHROMA_PARITY: 奇偶色度 (读者没明白)

FIELD_ID_PARITY: 奇偶场标志

INTERLACED: 视频格式是渐进式还是各行扫描

VIDEO_FORMAT: 视频格设置, 有 YUV422 YUV444 YUV420 RGB

Generator Polarity Register (Address Offset 0x006C)

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
RESERVED	31:7	Reserved
FIELD_ID_POL	6	Generated Field ID Polarity 0: Low during Field 0 and High during Field 1 1: High during Field 0 and Low during Field 1
ACTIVE_CHROMA_POL	5	Generated Active Chroma Polarity 0: Active Low Polarity 1: Active High Polarity

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
ACTIVE_VIDEO_POL	4	Generated Active Video Polarity 0: Active Low Polarity 1: Active High Polarity
HSYNC_POL	3	Generated Horizontal Sync Polarity 0: Active Low Polarity 1: Active High Polarity
VSYNC_POL	2	Generated Vertical Sync Polarity 0: Active Low Polarity 1: Active High Polarity
HBLANK_POL	1	Generated Horizontal Blank Polarity 0: Active Low Polarity 1: Active High Polarity
VBLANK_POL	0	Generated Vertical Blank Polarity 0: Active Low Polarity 1: Active High Polarity

这个寄存器设置相应的场输出极性和色度输出极性。

Generator Horizontal Frame Size Register (Address Offset 0x0070)

0x0070	GENERATOR HSIZE	Read/Write
Name	B its	Description
RESERVED	31:13	Reserved
FRAME_HSIZE	12:0	Generated Horizontal Frame Size. The width of the frame with blanking in number of pixels/clocks.

一副图像的一行的大小，包括了消隐和有效数据阶段。

Generator Vertical Frame Size Register (Address Offset 0x0074)

0x0074	GENERATOR VSIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
FIELD1_VSIZE	28:16	Generated Vertical Field 1 Size. The height with blanking in number of lines of field 1.
FRAME_VSIZE	12:0	Generated Vertical Frame Size. The height of the frame with blanking in number of lines.

一副图像的一场的大小，包括了消隐和有效数据阶段。

Generator Horizontal Sync Register (Address Offset 0x0078)

0x0078	GENERATOR HSYNC	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
Hsync_End	28:16	Generated Horizontal Sync End End cycle index of horizontal sync. Denotes the first cycle hsync_in is de-asserted.
RESERVED	15:13	Reserved
Hsync_Start	12:0	Generated Horizontal Sync End Start cycle index of horizontal sync. Denotes the first cycle hsync_in is asserted.

设置行的水平同步结束和同步开始

Generator Frame/Field 0 Vertical Blank Cycle Register (Address Offset 0x007C)

0x007C	GENERATOR F0_VBLANK_H	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VBLANK_HEND	28:16	Generated Vertical Blank Horizontal End End Cycle index of vertical blank. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F0_VBLANK_HSTART	12:0	Generated Vertical Blank Horizontal Start Start Cycle index of vertical blank. Denotes the first cycle vblank_in is asserted.

设置 Fram/Field0 的水平消隐结束和开始

Generator Frame/Field 0 Vertical Sync Line Register (Address Offset 0x0080)

0x0080	GENERATOR F0_VSYNC_V	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VSYNC_VEND	28:16	Generated Vertical Sync Vertical End End Line index of vertical sync. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_VSTART	12:0	Generated Vertical Sync Vertical Start Start line index of vertical sync. Denotes the first line vsync_in is asserted.

设置 Fram/Field0 的垂直同步垂直结束和开始

Generator Frame/Field 0 Vertical Sync Cycle Register (Address Offset 0x0084)

0x0084	GENERATOR F0_VSYNC_H	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
F0_VSYNC_HEND	28:16	Generated Vertical Sync Horizontal End End cycle index of vertical sync. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_HSTART	12:0	Generated Vertical Sync Horizontal Start Start cycle index of vertical sync. Denotes the first cycle vsync_in is asserted.

设置 Fram/Field0 的垂直同步水平结束和开始

Generator Field 1 Vertical Blank Cycle Register (Address Offset 0x0088)

0x0088	GENERATOR F1_VBLANK_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VBLANK_HEND	28:16	Generated Field 1 Vertical Blank Horizontal End End Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F1_VBLANK_HSTART	12:0	Generated Field 1 Vertical Blank Horizontal Start Start Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is asserted.

设置 Field1 的水平消隐结束和开始

Generator Field 1 Vertical Sync Line Register (Address Offset 0x008C)

0x008C	GENERATOR F1_VSYNC_V	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_VEND	28:16	Generated Field 1 Vertical Sync Vertical End End Line index of vertical sync for field 1. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_VSTART	12:0	Generated Field 1 Vertical Sync Vertical Start Start line index of vertical sync for field 1. Denotes the first line vsync_in is asserted.

设置 Field1 的垂直同步垂直结束和开始

Generator Field 1 Vertical Sync Cycle Register (Address Offset 0x0090)

0x0090	GENERATOR F1_VSYNC_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_HEND	28:16	Generated Field 1 Vertical Sync Horizontal End End cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_HSTART	12:0	Generated Field 1 Vertical Sync Horizontal Start Start cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is asserted.

设置 Field1 的垂直同步水平结束和开始

Frame Sync 0 - 15 Configuration Registers (Address Offsets 0x0100 - 0x013C)

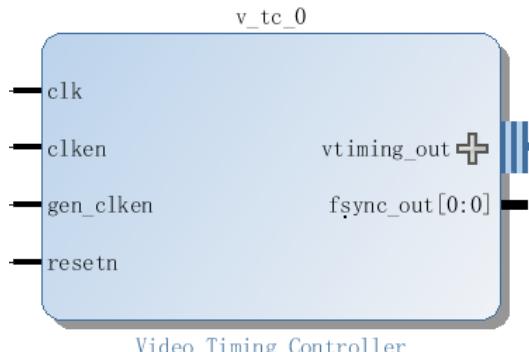
0x0100	FRAME SYNC 0 CONFIG	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
V_START	28:16	FRAME SYNCHRONIZATION VERTICAL START Vertical line during which the fsync_out[0] output port is asserted active-high. Note: Frame Syncs are not active during the complete line, only in the cycle during which both the V_START and H_START are valid each frame.
RESERVED	15:13	Reserved
H_START	12:0	FRAME SYNCHRONIZATION HORIZONTAL START Horizontal Cycle during which fsync_out[0] output port is asserted active-high

Generator Global Delay Register (Address Offset 0x140)

0x140	Generator Global Delay	Read/Write
Name	Bits	Description
Reserved	31:29	Reserved
V_DELAY	28:16	GENERATOR VERTICAL DELAY Vertical line offset. This is the number of lines that the generated output will be shifted relative to the detector (input timing). The vertical delay is only available when both the detector and generator are enabled. Can be combined with the H_DELAY.
Reserved	15:13	Reserved
H_DELAY	12:0	GENERATOR HORIZONTAL DELAY Horizontal cycle offset. This is the number of clock cycles that the generated output will be shifted relative to the detector (input timing). The horizontal delay is only available when both the detector and generator are enabled. Can be combined with the V_DELAY.

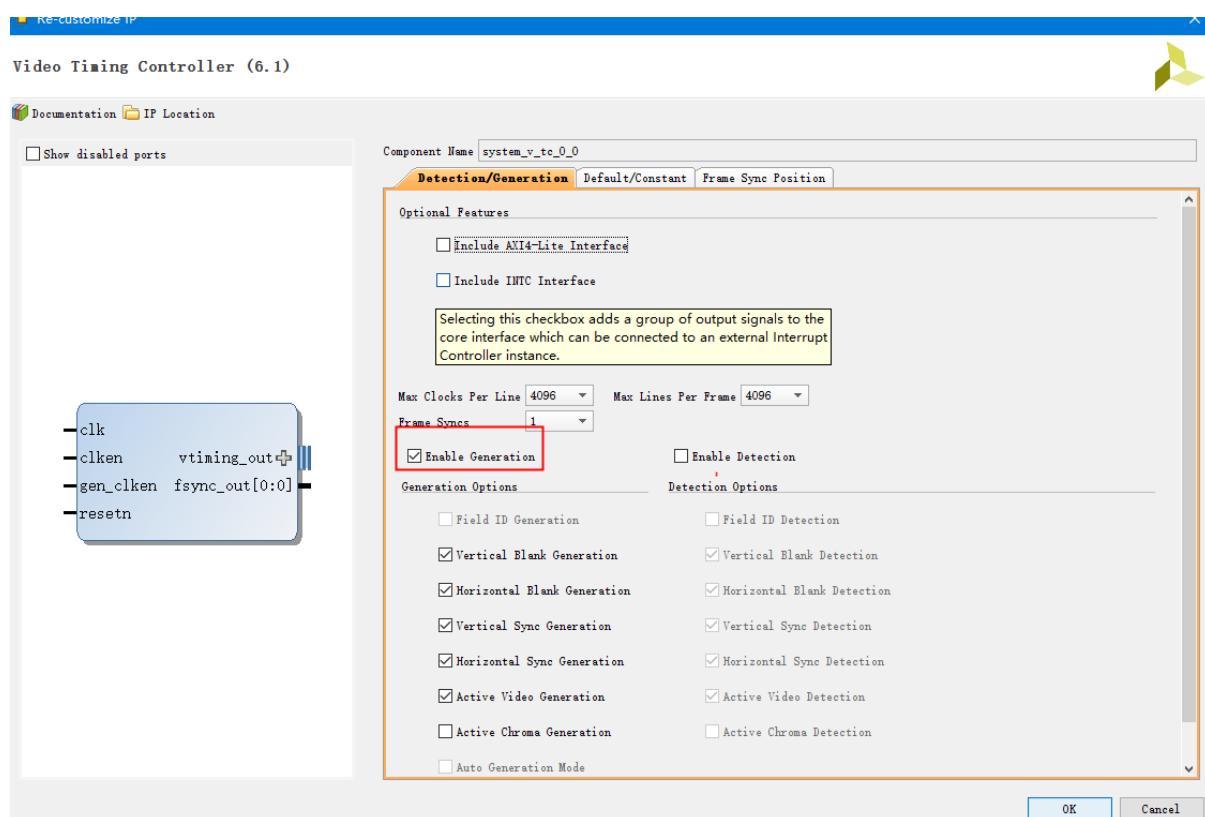
3.5.4 设置 VTC IP

讲了这么多实际上我们用的时候很简单,所以只要这么简单。

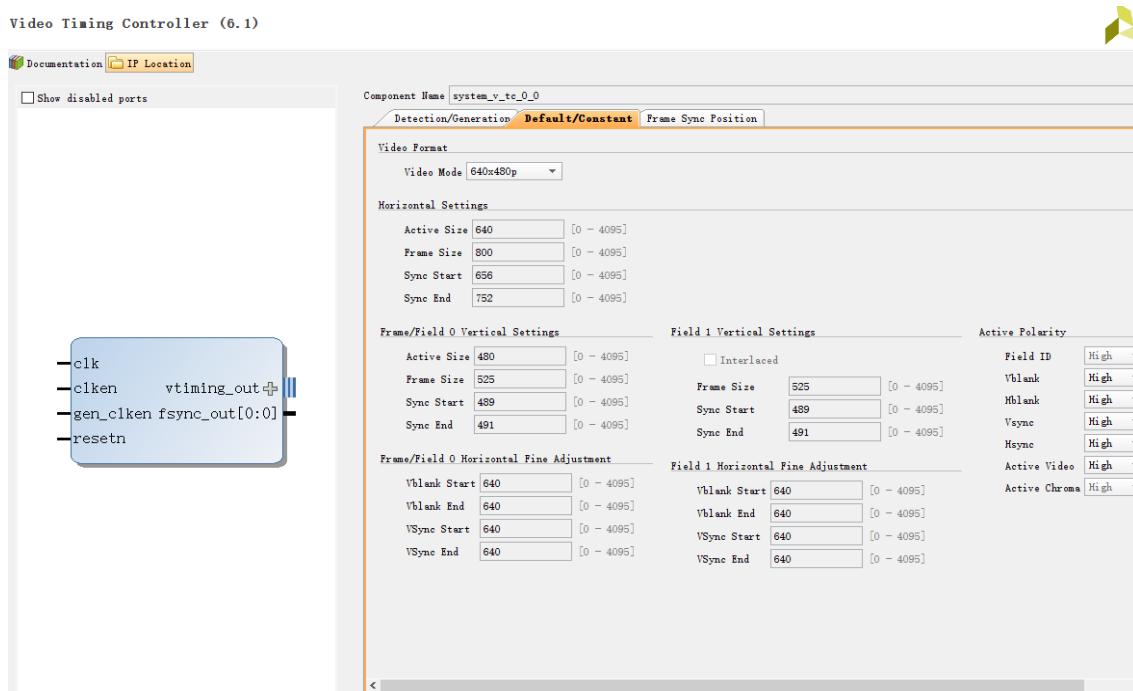


Video Timing Controller

由于不使用动态配置，并且只使用了视频时序产生，所以只要勾选如下复选框。

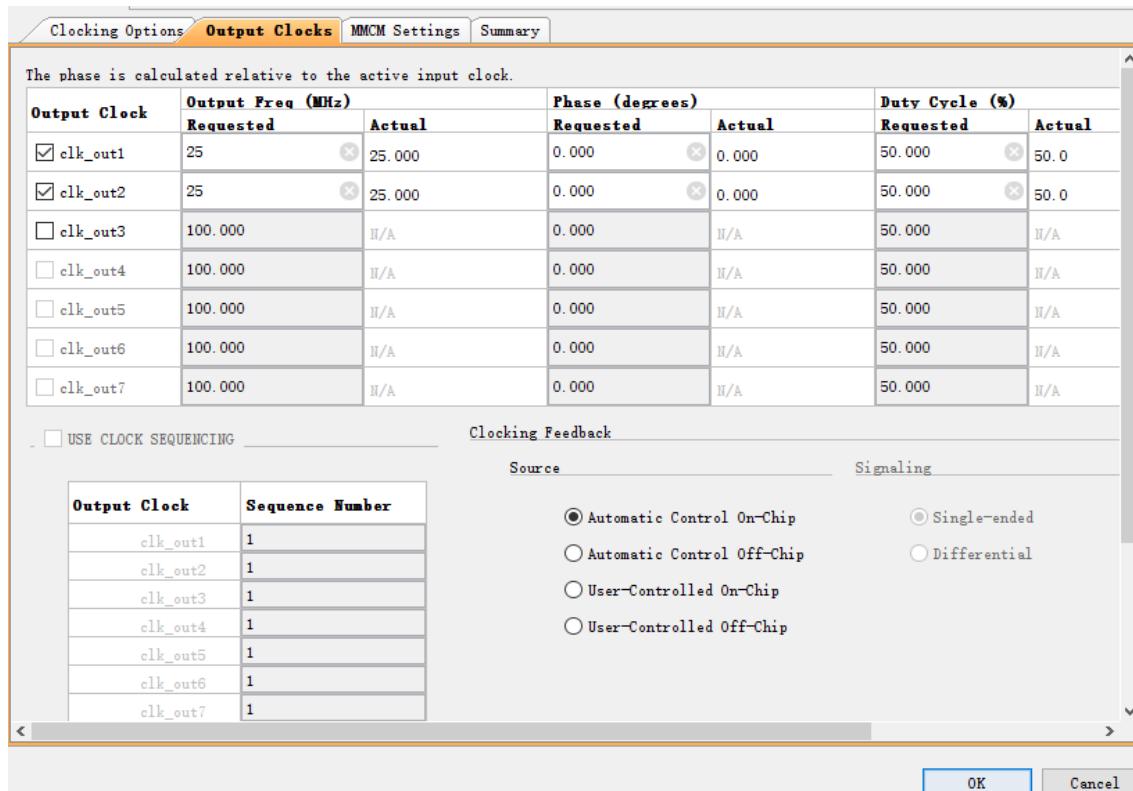


由于 OV7725 分辨率是 640X480 因此直接选择 640PX480 就可以了。



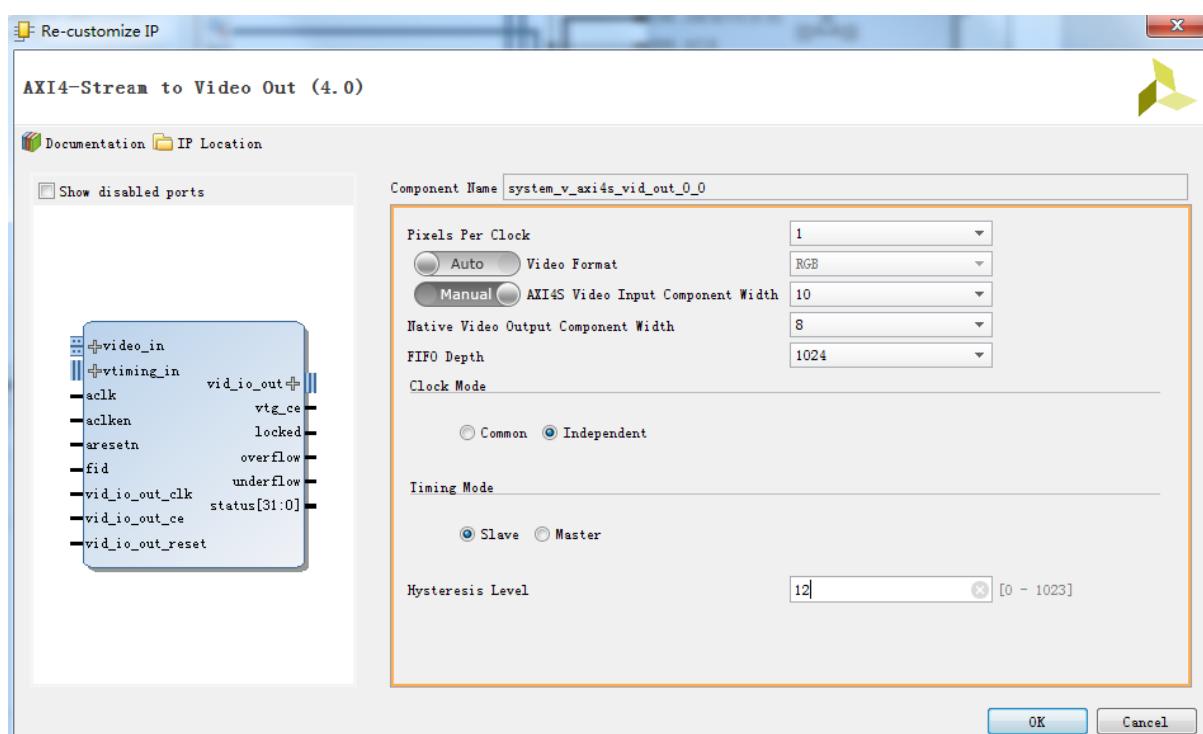
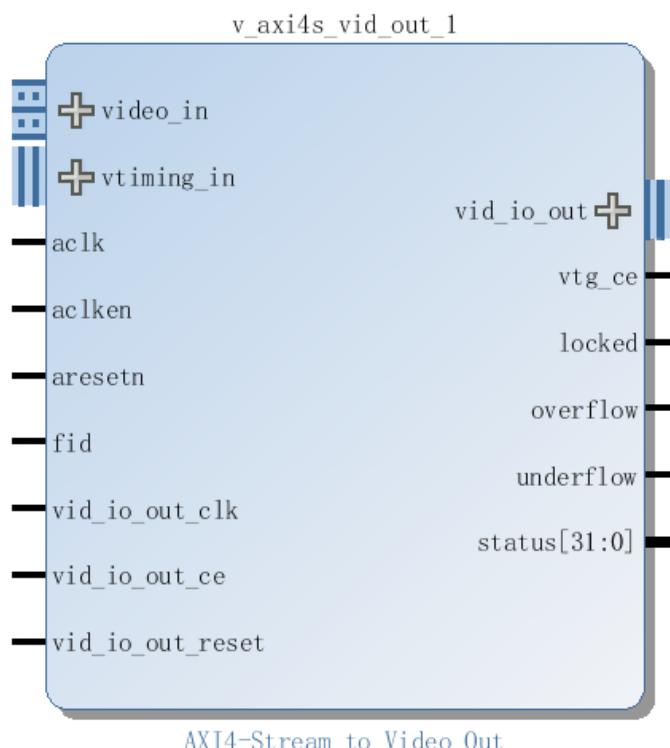
3.6 PLL 时钟设置

由于这里的分辨率是 640X480 因此提供给 VTC IP 和 VID OUTIP 的时钟只要 25M 就可以了



3.7 VID_OUT IP 的分析

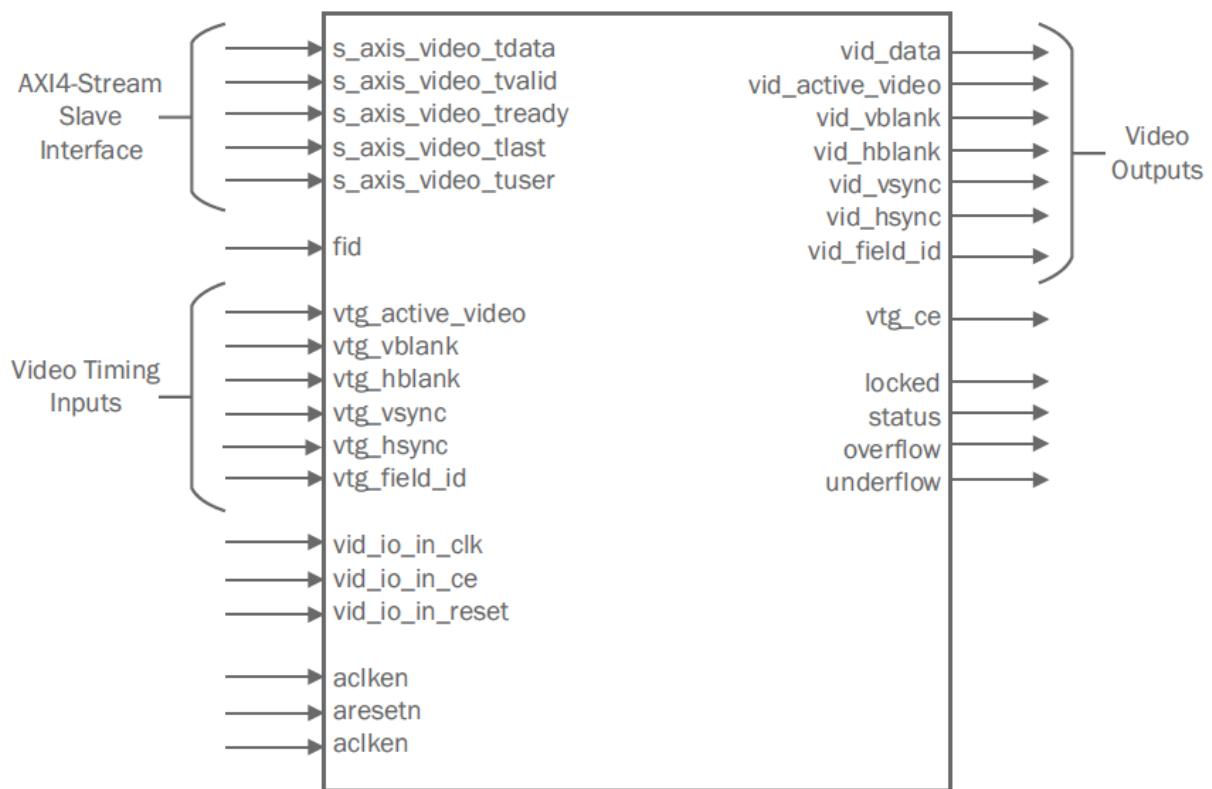
3.7.1 VID_OUT 的参数介绍



这些参数和前面的 V_TPG 参数类似

- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟
- Video Format: 视频格式
- FIFO Depth: FIFO 深度
- Hysteresis Level: 滞后输出

3.7.2 VID_OUT IP 接口信号的定义



Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_out_ce	Input	1	Native video clock enable
vid_io_out_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
vtg_ce	Output	1	VTC clock enable. Used to halt the timing generator for synchronization purposes.
locked	Output	1	Flag indicating whether the VTC is locked to the input timing. 1=locked. Synchronous to vid_io_out_clk.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk.

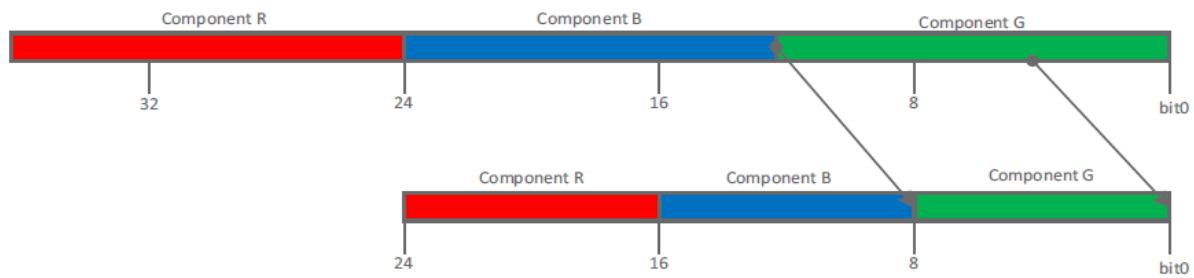
Video Timing Interface

Signal Name	Direction	Width	Description
vtg_vsync	In	1	VTC vertical sync. Active High
vtg_hsync	In	1	VTC horizontal sync. Active High
vtg_vblank	In	1	VTC vertical blank. Active High
vtg_hblank	In	1	VTC horizontal blank. Active High
vtg_act_vid	In	1	VTC active video signal. 1 = active video, 0 = blanked video
vtg_field_id	In	1	VTC field ID. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

AXI4 - Stream Interface

Signal Name	Direction	Width	Description
s_axis_video_tvalid	Input	1	AXI4-Stream TVALID. Active video data enable
s_axis_video_tuser	Input	1	AXI4-Stream TUSER. Start of Frame
s_axis_video_tlast	Input	1	AXI4-Stream TLAST. End of Line
s_axis_video_tready	Output	1	AXI4-Stream TREADY. Inverted FIFO full

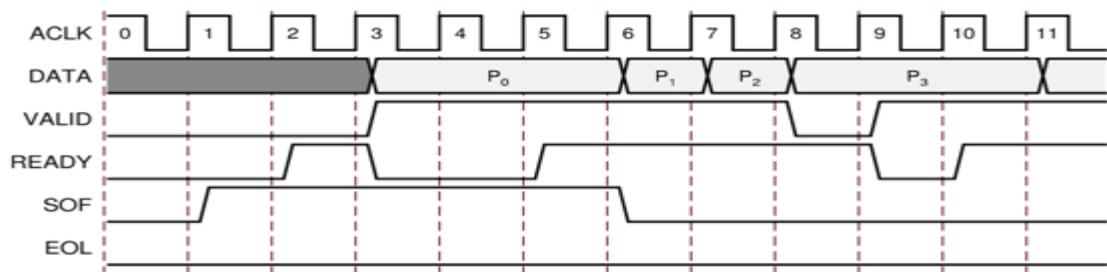
对于 s_axis_video_tdata(TDATA)需要注意一些事情,一般情况下我们的 RGB888 输出,但是,如果 s_axis_video_tdata 是 32bit 那么 VID_OUT IP 会自动截取到 24bit。由于技术手册只给出了 12bit 到 8bit 的截取方式,也就是 RGB 12:12:12 到 RGB 8:8:8 如下图:



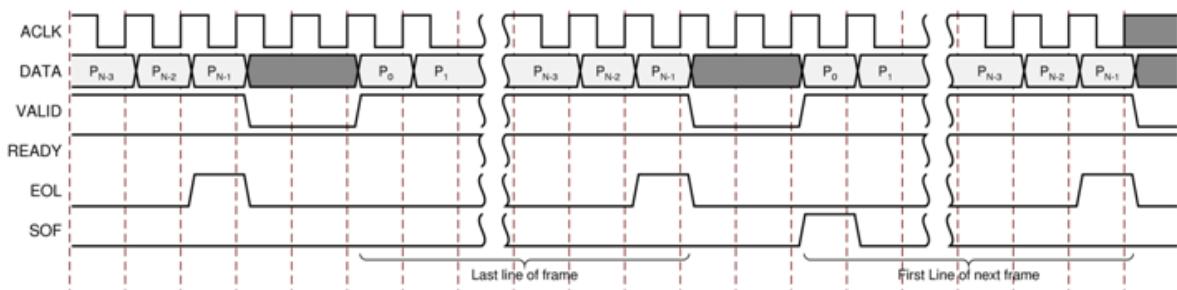
这种截取比较简单，把每个色度的低 4bit 截取就可以了。但是如果是 RGB10:10:10，官方并没有给出截取方式，但是可以通过纯色输出来进行测试。

因此最简单的办法是无需任何截取了，如果 s_axis_video_tdata 是 RGB8:8:8 那就无需任何截取，笔者设计的时候由于 AXI 总线是 32bit 因此数据的低 24bit 为 RGB 8:8:8 只要去掉高 24-31bit 就可以取得 RGB8:8:8,这样最省事。

以下时序图是在 SOF 是一帧图像的开始，当 VALID 和 READY 有效的时候开始传输数据。



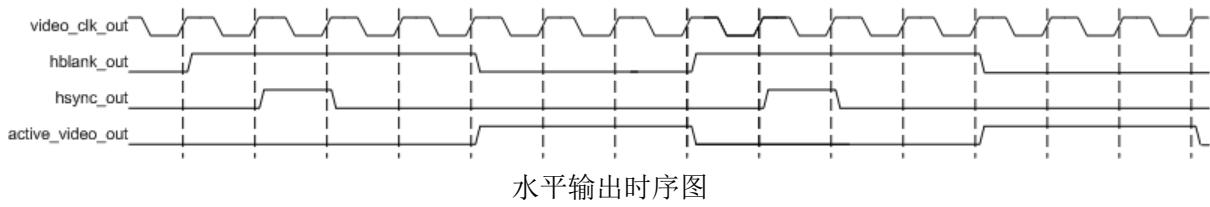
EOL 代表每一行的最后一个数据，SOF 代表前一帧的最后一行的结束，下一帧第一行的开始。SOF 为 1 个 PLUS 有效 (pg044_v_axis_out.pdf 没有描述清楚，而且有错误)。



Example Horizontal Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0003_0003
0x0070	Generator HSize	0x0000_0007
0x0078	Generator HSync	0x0005_0004
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

设置水平输出的相关寄存器

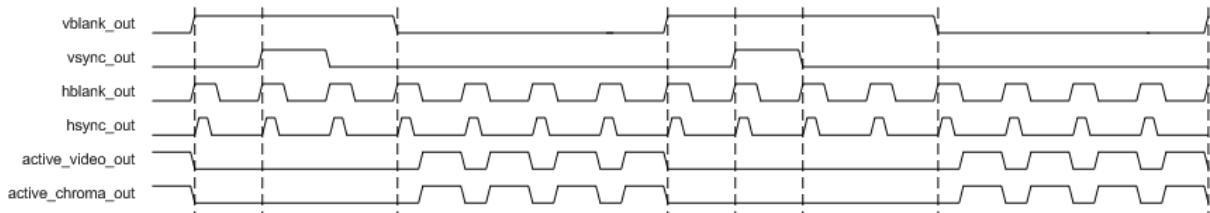


水平输出时序图

Example Vertical Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0004_0003
0x0070	Generator HSize	0x0000_0007
0x0074	Generator VSize	0x0000_0008
0x0078	Generator HSync	0x0005_0004
0x0080	Generator Frame 0 Vsync	0x0006_0005
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

设置垂直输出的相关寄存器



垂直输出时序图

3.8 FPGA 实现的用户逻辑代码

3.8.1 关键信号 1

```
assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready &
m_axis_video_tlast &(vid_in_v_cnt == VID_IN_VS); // dma in last signal
```

`m_axis_video_tvalid`:此信号是 vid in IP 输出的，代表输出数据有效

`s_axis_s2mm_tready`:此信号是 DMA IP 输出的，代表 DMA 可以接收数据

`m_axis_video_tlast`:这是每一行图像数据的最后一个像素的信号标志

`vid_in_v_cnt == VID_IN_VS`:表示一副图像的最后一个像素输出。

`s_axis_s2mm_tlast`:所有这些信号有效的时候代表 DMA 的最后一个数据

s_axis_s2mm_tlast 信号有效。

3.8.2 关键信号 2

```
assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready &
```

```
(vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); //vid out user
```

m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。

s_axis_video_tready: vid out IP 准备好了, 可以接收数据

```
(vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); 行计数器为 0 场计数器也为 0 说明要么这副图像已经结束, 也可以理解为下一副图像开始前。这样结合 s_axis_video_tready, m_axis_mm2s_tvalid 为 1, 基于 FPGA 时序, 下一个时钟输出 s_axis_video_tuser 为 1 正好是一副图像的第一个像素。
```

s_axis_video_tuser: 因此 s_axis_video_tuser 代表了每一副图像开始的第一个像素。

3.8.3 关键信号 3

```
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt ==
```

```
VID_OUT_HS); //vid out last signal
```

m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。

s_axis_video_tready: vid out IP 准备好了, 可以接收数据

vid_out_h_cnt == VID_OUT_HS): 图像一行数据的最后一个像素。

3.8.4 部分关键代码

表 3-6-4-1

<pre>reg [10:0] vid_out_v_cnt; reg [10:0] vid_out_h_cnt; reg [10:0] vid_in_v_cnt; parameter VID_OUT_HS = 11'd639;//图像输出行分辨率 parameter VID_OUT_VS = 11'd479;//图像输出场分辨率 parameter VID_IN_VS = 11'd479; always@(posedge FCLK_CLK0) begin if(!gpio_rtl_tri_o_0) vid_out_v_cnt <= 11'd0; else if(m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == VID_OUT_HS))</pre>
--

```
if(vid_out_v_cnt != VID_OUT_VS)
    vid_out_v_cnt <= vid_out_v_cnt + 1'b1;
else
    vid_out_v_cnt <= 11'd0;
else
    vid_out_v_cnt <= vid_out_v_cnt;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_out_h_cnt <= 11'd0;
    else
        if(m_axis_mm2s_tvalid & s_axis_video_tready)
            if(vid_out_h_cnt != VID_OUT_HS)
                vid_out_h_cnt <= vid_out_h_cnt + 1'b1;
            else
                vid_out_h_cnt <= 11'd0;
        else
            vid_out_h_cnt <= vid_out_h_cnt;
    end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_in_v_cnt <= 11'd0;
    else
        if(m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast)
            if(vid_in_v_cnt != VID_IN_VS)
                vid_in_v_cnt <= vid_in_v_cnt + 1'b1;
            else
                vid_in_v_cnt <= 11'd0;
        else
            vid_in_v_cnt <= vid_in_v_cnt;
    end

assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == 11'd0) &
(vid_out_v_cnt == 11'd0); //vid out user
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt ==
VID_OUT_HS);//vid out last signal

assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast
&(vid_in_v_cnt == VID_IN_VS);//dma in last signal
```

3.9 PS 部分

3.9.1 DMA 中断函数部分分析

为了让图像输出高品质效果，PS 部分设计了 3 缓存处理机制。3 缓存处理机制在大量图像缓冲处理方法是最有效的办法之一。

在 DMA_intr.h 文件中，定义 3 段内存空间用于保存三副最新的图像。

```
#define BUFFER0_BASE (MEM_BASE_ADDR)
#define BUFFER1_BASE (MEM_BASE_ADDR + IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define BUFFER2_BASE (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
```

在 DMA_intr.h 文件中，还定义一下 2 个变量 1 个指针数组。tx_buffer_index; 指示了当前的发送缓存序号，rx_buffer_index; 指示了当前的接收缓存序号。*BufferPtr[3] 会被制定到对应的内存地址空间。

```
extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;
extern u32 *BufferPtr[3];
```

在 main 函数里面有这么一段实现了指针数组指向内存地址空间。

```
BufferPtr[0] = (u32 *)BUFFER0_BASE;
BufferPtr[1] = (u32 *)BUFFER1_BASE;
BufferPtr[2] = (u32 *)BUFFER2_BASE;
```

下面给出 dma_intr.h 的完整代码

表 3-7-1-1 dma_intr.h

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 */

#ifndef DMA_INTR_H
#define DMA_INTR_H
#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
#include "xscugic.h"

/********************* Constant Definitions ********************/
/*
 * Device hardware build related constants.
```

```
/*
#define DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID

#define MEM_BASE_ADDR      0x10000000

#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR

#define IMAGE_WIDTH      640
#define IMAGE_HEIGHT     480
#define BYTES_PER_PIXEL   4
#define BUFFER_NUM       2

#define MEM_BASE_ADDR      0x10000000

#define BUFFER0_BASE      (MEM_BASE_ADDR )
#define BUFFER1_BASE      (MEM_BASE_ADDR +      IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)
#define BUFFER2_BASE      (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)

/* Timeout loop counter for reset
 */
#define RESET_TIMEOUT_COUNTER    10000
/* test start value
 */
#define TEST_START_VALUE    0xC
/*
 * Buffer and Buffer Descriptor related constant definition
 */
#define MAX_PKT_LEN        (IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
/*
 * transfer times
 */
#define NUMBER_OF_TRANSFERS 100000

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

```

extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;

extern u32 *BufferPtr[3];

int DMA_CheckData(int Length, u8 StartValue);
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr);
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);
#endif

```

每次一幅图像通过 DMA 进入 DDR 后，会产生 DMA 中断请求，在 DMA 中断请求中，会指定下一次 DMA 接收的 buffer 位置。

表 3-7-1-2 DMA_RxIntrHandler 函数

```

/****************************************************************************
/*
*
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.
*
* @param    Callback is a pointer to RX channel of the DMA engine.
*
* @return   None.
*
* @note    None.
*
****************************************************************************
static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    u32 Status;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

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

    /* Acknowledge pending interrupts */
}

```

```

XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

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

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
 */
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
    xil_printf("rx error! \r\n");
    return;
}

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

if(rx_buffer_index == 2)
    rx_buffer_index = 0;
else
    rx_buffer_index++;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[rx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma failed! 0 %d\r\n", Status);
    return;
}

}

```

发送函数通过 tx_buffer_index 标记需要发送的缓存部分，并且确保发送的是最新保存的一副图像。

表 3-7-3 DMA_TxIntrHandler

```
*****
/*

```

```
/*
 * This is the DMA TX Interrupt handler function.
 *
 * It gets the interrupt status from the hardware, acknowledges it, and if any
 * error happens, it resets the hardware. Otherwise, if a completion interrupt
 * is present, then sets the TxDone.flag
 *
 * @param    Callback is a pointer to TX channel of the DMA engine.
 *
 * @return   None.
 *
 * @note    None.
 *
 *****/
static void DMA_TxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    u32 Status;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

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

    /* Acknowledge pending interrupts */

    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);

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

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

```

//Error = 1;
xil_printf("tx error! \r\n");
return;
}

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

    TxDone++;
}

if(rx_buffer_index == 0)
    tx_buffer_index = 2;
else
    tx_buffer_index = rx_buffer_index - 1;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[tx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma failed! 0 %d\r\n", Status);
    return;
}
}
}

```

3.9.2 IIC 驱动

IIC 驱动的初始化部分参考于官方的代码，本章当中将其封装成了一个 iic_init 子函数，如下表所示：

<pre> int iic_init(void) { int Status; XIicPs_Config *Config; /* * Initialize the IIC driver so that it's ready to use * Look up the configuration in the config table, * then initialize it. */ Config = XIicPs_LookupConfig(DeviceId); if (NULL == Config) { </pre>
--

```

    return XST_FAILURE;
}

Status = XIicPs_CfgInitialize(&Iic, Config, Config->BaseAddress);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}

/*
 * Set the IIC serial clock rate.
 */
XIicPs_SetSclk(&Iic, IIC_SCLK_RATE);

return XST_SUCCESS;
}

```

I2C_write 函数实现了一组寄存器数据的写入操作，其中包括 8bit 的寄存器地址和 8bit 的数据：

```

int I2C_write_byte(XIicPs *InstancePtr, u8 sensor_slave_addr, void
*write_byte, int byte_num)
{
    int Status;

    Status = XIicPs_MasterSendPolled(InstancePtr, write_byte,
byte_num, sensor_slave_addr);

    if(Status != XST_SUCCESS) {
        xil_printf("IIC Send byte Failed!");
    }

    /*
     * Wait until bus is idle to start another transfer.
     */
    while (XIicPs_BusIsBusy(InstancePtr));

    return XST_SUCCESS;
}

int I2C_write(XIicPs *InstancePtr, u8 addr, u8 data)
{
    u8 buf[2];
    buf[0] = addr;
    buf[1] = data;
    if(I2C_write_byte(InstancePtr, OV_CAM, buf, 2) != XST_SUCCESS)

```

```

    xil_printf("IIC Write bytes Failed");
    return XST_SUCCESS;
}

```

I2C_config_init 子函数完成了整个寄存器的写入操作，其中需要注意，写入的数据被定义为了一个结构体，其中存放的各个元素既是要写入的寄存器数据，如下表所示：

```

struct config_table ov_7725_config_table[]={
    {0x12, 0x80},           // BIT[7]-Reset all the Reg

    //Bingo Init REV
    //Write Data Index
    {0x1c, 0x7f},
    {0x1d, 0xa2},
    {0x12, 0x80},
    {0x3d, 0x03}, //DC offset for analog process
    {0x15, 0x00}, //COM10: href/vsync/pclk/data reverse(Vsync H valid)
    {0x17, 0x22}, //VGA: 0x22; QVGA: 0x3f;
    {0x18, 0xa4}, //VGA: 0xa4; QVGA: 0x50;
    {0x19, 0x07}, //VGA: 0x07; QVGA: 0x03;
    {0x1a, 0xf0}, //VGA: 0xf0; QVGA: 0x78;
    {0x32, 0x00}, //Bit[7]:Mirror image edge alignment
    {0x29, 0xA0}, //VGA: 0xA0; QVGA: 0xF0
    {0x2C, 0xF6}, //VGA: 0xF0; QVGA: 0x78
    {0x0d, 0x41}, //PLL 4x
        //scbb_senddata(0x2a, 0x00}, //PLL 4x
    {0x11, 0x00}, //CLKRC,Finternal clock = Finput clk*PLL
multiplier/[ CLKRC[5:0]+1)*2] = 25MHz*4/[(x+1)*2]
                                            //00: 50fps,
01:25fps, 03:12.5fps (50Hz Fliter)
    {0x12, 0x06}, //BIT[6]: 0:VGA; 1:QVGA
                                            //VGA: 00:YUV;
01:Processed Bayer RGB; 10:RGB; 11:Bayer RAW; BIT[7]-Reset all the Reg
    {0x0c, 0xd0}, //COM3: Bit[6]:Horizontal mirror image ON/OFF,
Bit[0]:Color bar; Default:0x10
        //DSP control
    {0x42, 0x7f}, //BLC Blue Channel Target Value, Default: 0x80
    {0x4d, 0x09}, //BLC Red Channel Target Value, Default: 0x80
    {0x63, 0xf0}, //AWB Control
    {0x64, 0xff}, //DSP_Ctrl1:
    {0x65, 0x00}, //DSP_Ctrl2:
    {0x66, 0x00}, //COM3[4](0x0C), DSP_Ctrl3[7]:00:YUYV; 01:YVYU;
[10:UYVY] 11:VYUY
    {0x67, 0x00}, //DSP_Ctrl4:[1:0]00/01: YUV or RGB; 10: RAW8; 11: RAW10
}

```

```
//AGC AEC AWB
{0x13, 0xff},
{0x0f, 0xc5},
{0x14, 0x11},
{0x22, 0x98}, //Banding Filter Minimum AEC Value; Default: 0x09
{0x23, 0x03}, //Banding Filter Maximum Step
{0x24, 0x40}, //AGC/AEC - Stable Operating Region (Upper Limit)
{0x25, 0x30}, //AGC/AEC - Stable Operating Region (Lower Limit)
{0x26, 0xa1}, //AGC/AEC Fast Mode Operating Region
{0x2b, 0x00}, //Taiwan: 0x00:60Hz Filter; Mainland: 0x9e:50Hz Filter
{0x6b, 0xaa}, //AWB Control 3
{0x13, 0xff}, //0xff: AGC AEC AWB Enable; 0xf0: AGC AEC AWB Disable;

//sccb_senddata(0x0d, 0x41}, //20141206

//matrix sharpness brightness contrast UV
{0x90, 0xa},
{0x91, 0x01},
{0x92, 0x01},
{0x93, 0x01},
{0x94, 0x5f},
{0x95, 0x53},
{0x96, 0x11},
{0x97, 0x1a},
{0x98, 0x3d},
{0x99, 0x5a},
{0x9a, 0x1e},
{0x9b, 0x3f}, //Brightness
{0x9c, 0x25},
{0x9e, 0x81},
{0xa6, 0x06},
{0xa7, 0x65},
{0xa8, 0x65},
{0xa9, 0x80},
{0xaa, 0x80},

//Gamma correction
{0x7e, 0x0c},
{0x7f, 0x16}, //
{0x80, 0x2a},
{0x81, 0x4e},
{0x82, 0x61},
{0x83, 0x6f},
```

```

{0x84, 0x7b},
{0x85, 0x86},
{0x86, 0x8e},
{0x87, 0x97},
{0x88, 0xa4},
{0x89, 0xaf},
{0x8a, 0xc5},
{0x8b, 0xd7},
{0x8c, 0xe8},
{0x8d, 0x20},
    //scrb_senddata(0x1c, 0x7f},//20141206
{0x0e, 0x65},//20141206
{Config_done,0x00}
};

Iic_config_init 子程序的内容如下图所示:

```

```

int I2C_config_init(void)
{
    int Status;
    int i=0;

    Status = iic_init();
    if(Status != XST_SUCCESS)
    {
        xil_printf("I2C init Failed!");
    }

    while(ov_7725_config_table[i].addr != Config_done)
    {

        I2C_write(&Iic,ov_7725_config_table[i].addr,ov_7725_config_table[i].dat
a);
        i++;
    }

    return XST_SUCCESS;
}

```

3.9.3 main.c 文件

这个主程序比较简单，内容比上一个课程的精简很多，这里需要注意的地方是

XGpio_DiscreteWrite(&Gpio, 1, 1);函数这个函数是这只摄像头和 DMA 之间数据同步的，没有这个同步图像容易错位。另外在主函数里面首先启动 DMA 接收和发送中断各一次，以后就可以在中断里面继续触发了。

表 3-7-2-1 main.c

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 * axi dma test
 *
 */

#include "dma_intr.h"
#include "sys_intr.h"
#include "xgpio.h"

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

volatile u8 tx_buffer_index;
volatile u8 rx_buffer_index;

u32 *BufferPtr[3];

static XScuGic Intc; //GIC
static XAxiDma AxiDma;
static XGpio Gpio;

#define AXI_GPIO_DEV_ID           XPAR_AXI_GPIO_0_DEVICE_ID

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

int main(void)
```

{

u32 Status;

```
BufferPtr[0] = (u32 *)BUFFER0_BASE;  
BufferPtr[1] = (u32 *)BUFFER1_BASE;  
BufferPtr[2] = (u32 *)BUFFER2_BASE;
```

```
tx_buffer_index = 0;  
rx_buffer_index = 0;  
TxDone = 0;  
RxDone = 0;  
Error = 0;
```

```
XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);  
XGpio_SetDataDirection(&Gpio, 1, 0);  
init_intr_sys();
```

Miz702_EMIO_init();ov7725_init_rgb();

```
XGpio_DiscreteWrite(&Gpio, 1, 1);  
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[rx_buffer_index],  
MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
```

```
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[tx_buffer_index],  
MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
```

```
while (1);  
    return XST_SUCCESS;
```

}

3.10 实验效果



CH04_AXI_DMA_OV5640 摄像头采集系统

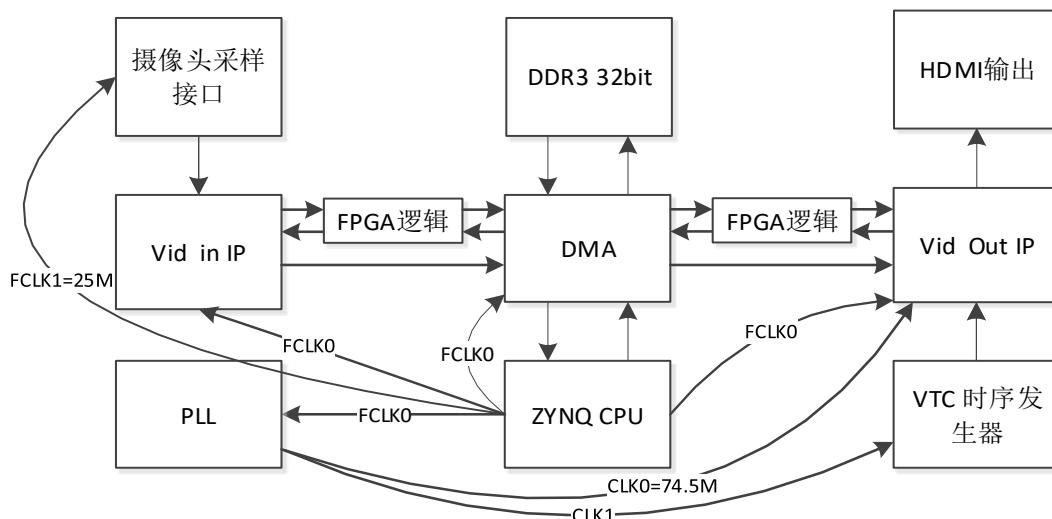
4.1 概述

本课程讲解如何搭建基于 DMA 的图形系统，方案原理和搭建 7725 的一样，只是 OV5640 显示的分辨率是 1280X720 如下，只是购买了 OV5640 摄像头的用户可以直接从本章开始学习。

摄像头采样图像数据后通过 DMA 送入到 DDR，在 PS 部分产生 DMA 接收中断，在接收中断里面再把 DDR 里面保持的图形数据 DMA 发送出去。在 FPGA 的接收端口部分产生 VID OUT 时序驱动 HDMI 显示器显示图形。

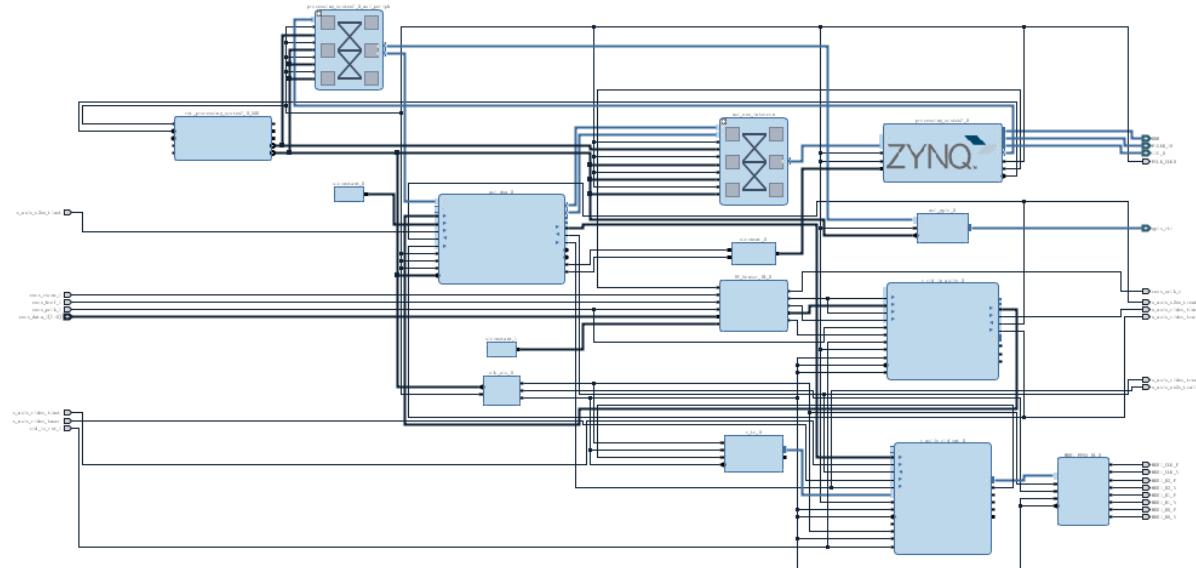
4.2 系统构架

4.2.1 构架方案图



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

4.2.2 构 BLOCK 模块化设计方案图

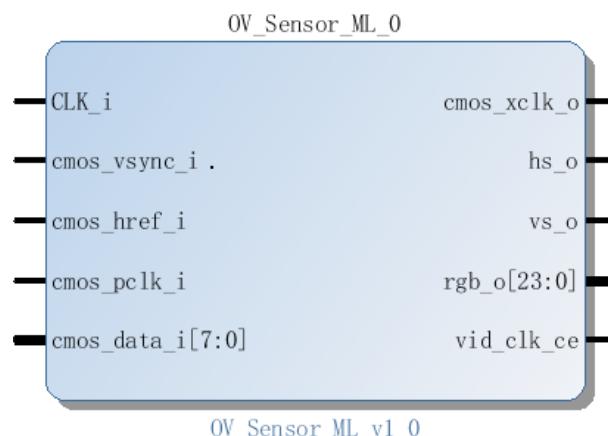


4.3 IIC 接口

IIC 接口配置方法与上一章相同，不记得的可以回到上一章复习一下。

4.4 vid in IP 介绍

4.4.1 OV_Sensor_ML 自定义 IP 模块



外部信号接口说明：

CLK_i : 为输入时钟，通常接 24MHZ 或者 25MHZ

Cmos_xclk_o:摄像头工作，通常直接把 CLK_i 连接到 cmos_xclk_o

Cmos_vsync_i: 摄像头场同步输入 上升沿代表场同步开始

Cmos_href_i:摄像头行同步输入 高电平代表行数据有效

Cmos_data[7:0]:摄像头数据输入

Hs_o:采集 OV_Sensor_ML IP 输出的行数据有效

Vs_o:采集 OV_Sensor_ML IP 输出的场同步信号

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

OV_Sensor_ML IP 包含 3 个源程序文件，分别为 OV_Sensor_ML.v、cmos_decode_v1.v、count_reset_v1.v 文件。

OV_Sensor_ML.v 程序中，对 cmos_data_i、cmos_href_i、cmos_vsync_i 做了一次寄存器，笔者发现图像效果有所改观。笔者分析，是因为寄存后有利于去除一些毛刺信号，提高了数据的稳定性。

表 3-3-1-1 OV_Sensor_ML 源码 OV_Sensor_ML.v

```
`timescale 1ns / 1ps
///////////////////////////////
// Company: milinker
// Engineer:tangjinyuan
//
// Create Date: 15:54:59 11/21/2015
// Design Name:
// Module Name: OV7725_IP_ML
// Project Name: OV7725_IP_ML
// Target Devices: ZYNQ
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////
module OV_Sensor_ML(
    input CLK_i,
    //----- CMOS sensor hardware interface -----
    input      cmos_vsync_i,//cmos vsync
    input      cmos_href_i, //cmos hsync refrence
    input      cmos_pclk_i, //cmos pxiel clock
    output     cmos_xclk_o, //cmos externl clock
    input[7:0]cmos_data_i, //cmos data
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    // output de_o,//data enable.
    output [23:0]rgb_o,//data output,
    output vid_clk_ce
);
```

```
);

//-----视频输出解码模块-----//
wire [15:0]rgb_o_r;
assign rgb_o = {rgb_o_r[4:0] ,3'd0 ,rgb_o_r[10:5] ,2'd0,rgb_o_r[15:11],3'd0};

reg [7:0]cmos_data_r;
reg cmos_href_r;
reg cmos_vsync_r;

always@(posedge cmos_pclk_i)
begin
    cmos_data_r <= cmos_data_i;
    cmos_href_r <= cmos_href_i;
    cmos_vsync_r<= cmos_vsync_i;
end
//assign rgb_o = 24'b11111111_00000000_11111111;
cmos_decode cmos_decode_u0(
    //system signal.
    .cmos_clk_i(CLK_i),//cmos senseor clock.
    .rst_n_i(RESETn_i2c),//system reset.active low.
    //cmos sensor hardware interface.
    .cmos_pclk_i(cmos_pclk_i),//(cmos_pclk),//input pixel clock.
    .cmos_href_i(cmos_href_r),//(cmos_href),//input pixel hs signal.
    .cmos_vsync_i(cmos_vsync_r),//(cmos_vsync),//input pixel vs signal.
    .cmos_data_i(cmos_data_r),//(cmos_data),//data.
    .cmos_xclk_o(cmos_xclk_o),//(cmos_xclk),//output clock to cmos sensor.
    //user interface.
    .hs_o(hs_o),//hs signal.
    .vs_o(vs_o),//vs signal.
    // .
    .de_o(de_o),//data enable.
    .rgb565_o(rgb_o_r),//data output
    .vid_clk_ce(vid_clk_ce)
);

count_reset_v1#(
    .num(20'hffff0)
)(
    .clk_i(CLK_i),
    .rst_o(RESETn_i2c)
);

endmodule
```

1 cmos_decode_v1.v 是本模块的关键部分,实现了RGB565 的解码输出以及 vid_clk_ce 实现了此模块和 vid_in IP 直接时序匹配的关系。

表 3-3-1-2 OV_Sensor_ML 源码 cmos_decode_v1.v

```
`timescale 1ns / 1ps
///////////////////////////////
// Company: milinker corporation
// WEB:www.milinker.com
// BBS:www.osrc.cn
// Engineer:..
// Create Date: 07:28:50 09/04/2015
// Design Name: cmos_decode_v1
// Module Name: cmos_decode_v1
// Project Name: cmos_decode_v1
// Target Devices:
// Tool versions:
// Description: cmos_decode_v1.
// Revision: V1.0
// Additional Comments:
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low
//4) _dg debug signal
//5) _r reg delay
//6) _s state machine
/////////////////////////////
module cmos_decode(
    //system signal.
    input cmos_clk_i,//cmos senseor clock.
    input rst_n_i,//system reset.active low.
    //cmos sensor hardware interface.
    input cmos_pclk_i,//input pixel clock.
    input cmos_href_i,//input pixel hs signal.
    input cmos_vsync_i,//input pixel vs signal.
    input[7:0]cmos_data_i,//data.
    output cmos_xclk_o,//output clock to cmos sensor.
    //user interface.
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    output reg [15:0]rgb565_o,//data output
    output vid_clk_ce
);
parameter[5:0]CMOS_FRAME_WAITCNT = 4'd15;
reg[4:0]rst_n_reg = 5'd0;
```

```
//reset signal deal with.  
always@(posedge cmos_clk_i)  
begin  
    rst_n_reg <= {rst_n_reg[3:0],rst_n_i};  
end  
  
reg[1:0]vsync_d;  
reg[1:0]href_d;  
wire vsync_start;  
wire vsync_end;  
//vs signal deal with.  
always@(posedge cmos_pclk_i)  
begin  
    vsync_d <= {vsync_d[0],cmos_vsync_i};  
    href_d <= {href_d[0],cmos_href_i};  
end  
  
assign vsync_start = vsync_d[1]&(!vsync_d[0]);  
assign vsync_end = (!vsync_d[1])&vsync_d[0];  
  
reg[6:0]cmos_fps;  
//frame count.  
always@(posedge cmos_pclk_i)  
begin  
    if(!rst_n_reg[4])  
        begin  
            cmos_fps <= 7'd0;  
        end  
    else if(vsync_start)  
        begin  
            cmos_fps <= cmos_fps + 7'd1;  
        end  
    else if(cmos_fps >= CMOS_FRAME_WAITCNT)  
        begin  
            cmos_fps <= CMOS_FRAME_WAITCNT;  
        end  
end  
//wait frames and output enable.  
reg out_en;  
always@(posedge cmos_pclk_i)  
begin  
    if(!rst_n_reg[4])  
        begin  
            out_en <= 1'b0;  
        end
```

```
        end
    else if(cmos_fps >= CMOS_FRAME_WAITCNT)
        begin
            out_en <= 1'b1;
        end
    else
        begin
            out_en <= out_en;
        end
end

//output data 8bit changed into 16bit in rgb565.
reg [7:0] cmos_data_d0;
reg [15:0]cmos_rgb565_d0;
reg byte_flag;

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        byte_flag <= 0;
    else if(cmos_href_i)
        byte_flag <= ~byte_flag;
    else
        byte_flag <= 0;
end

reg byte_flag_r0;
always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        byte_flag_r0 <= 0;
    else
        byte_flag_r0 <= byte_flag;
end

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        cmos_data_d0 <= 8'd0;
    else if(cmos_href_i)
        cmos_data_d0 <= cmos_data_i; //MSB -> LSB
    else if(~cmos_href_i)
        cmos_data_d0 <= 8'd0;
end
```

```

always@(posedge cmos_pclk_i)
begin
    if(!rst_n_reg[4])
        rgb565_o <= 16'd0;
    else if(cmos_href_i&byte_flag)
        rgb565_o <= {cmos_data_d0,cmos_data_i}; //MSB -> LSB
    else if(~cmos_href_i)
        rgb565_o <= 8'd0;
end

assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
assign vs_o = out_en ? vsync_d[1] : 1'b0;
assign hs_o = out_en ? href_d[1] : 1'b0;

assign cmos_xclk_o = cmos_clk_i;

endmodule

```

count_reset_v1.v 源文件实现了信号的延迟复位。

表 3-3-1-1 count_reset_v1.v

```

`timescale 1ns / 1ps
///////////////////////////////
// Company: milinker corporation
// WEB:www.milinker.com
// BBS:www.osrc.cn
// Engineer:sanliuyaoling.
// Create Date: 07:28:50 12/04/2015
// Design Name: count_reset_v1
// Module Name: count_reset_v1
// Project Name: count_reset_v1
// Target Devices: XC7Z020-CLG484-1I
// Tool versions: vivado2015.4
// Description: count_reset_v1
// Revision: V1.0
// Additional Comments:
//1) _i PIN input
//2) _o PIN output
//3) _n PIN active low
//4) _dg debug signal
//5) _r reg delay
//6) _s state machine
/////////////////////////////

```

```
module count_reset_v1#
(
    parameter[19:0]num = 20'hffff0
)(
    input clk_i,
    output rst_o
);

reg[19:0] cnt = 20'd0;
reg rst_d0;

/*count for clock*/
always@(posedge clk_i)
begin
    cnt <= ( cnt <= num)?( cnt + 20'd1 ):num;
end

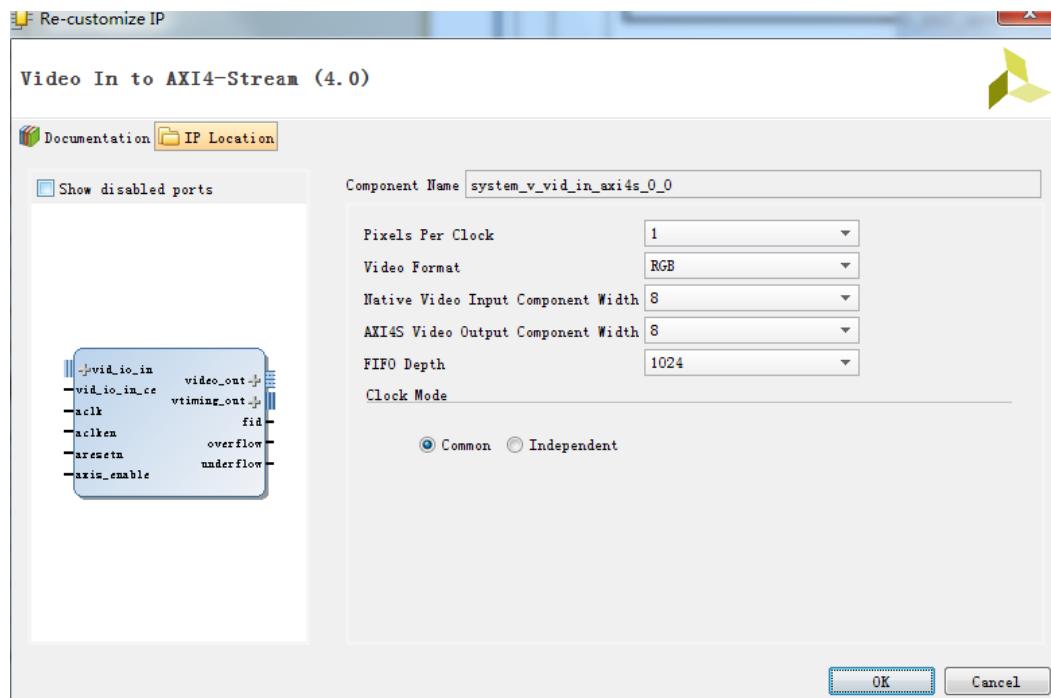
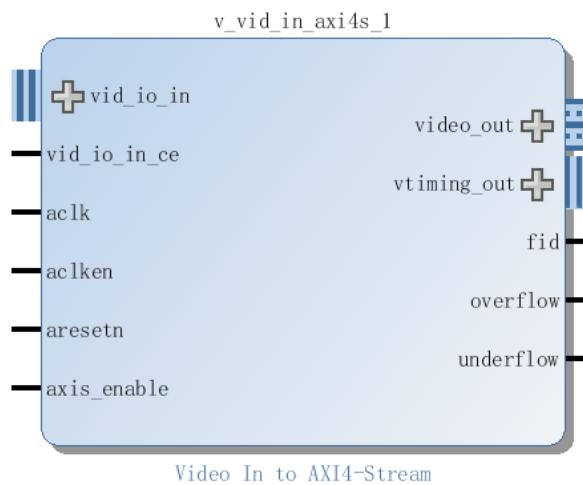
/*generate output signal*/
always@(posedge clk_i)
begin
    rst_d0 <= ( cnt >= num)?1'b1:1'b0;
end

assign rst_o = rst_d0;

endmodule
```

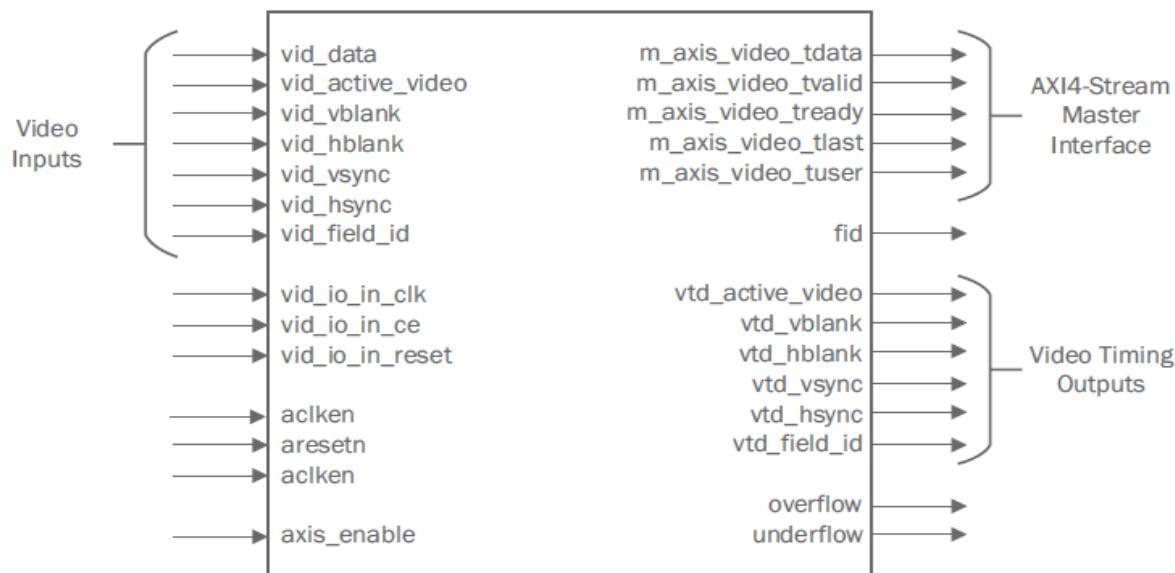
表 3-3-1-2

4.4.2 vid in IP 模块



- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Video Format: 视频格式
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- FIFO Depth: FIFO 深度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟

4.4.3 VID_IN IP 接口信号的定义



Common Interface

Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
axis_enable	Input	1	This input should be connected to the VTC detector locked status and is synchronous to vid_io_in_clk. 1 = Enable writes into FIFO 0 = Disable writes into FIFO
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_in_clk	Input	1	Native video clock. Only available in independent clock mode.
vid_io_in_ce	Input	1	Native video clock enable
vid_io_in_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk. If an overflow occurs, this could indicate that the connected AXI4-Stream Slave is creating excessive back-pressure.
underflow	Output	1	Flag indicating that the FIFO has under-flowed. This should never occur under normal operation. Synchronous to aclk.

Video Timing Interface

Signal Name	Direction	Width	Description
vtd_vsync	Out	1	Vertical sync video timing signal.
vtd_hsync	Out	1	Horizontal sync video timing signal.
vtd_vblank	Out	1	Vertical blank video timing signal.
vtd_hblank	Out	1	Horizontal blank video timing signal.
vtd_active_video	Out	1	Active video flag. 1 = active video, 0 = blanked video
vtd_field_id	Out	1	VTC field ID. 0= even field, 1= odd field.

Video Input Interface

Signal Name	Direction	Width	Description
vid_active_video	In	1	Video data valid. 1 = active video, 0 = blanked video
vid_vsync	In	1	Vertical sync video timing signal. Active High
vid_hsync	In	1	Horizontal sync video timing signal. Active High
vid_vblank	In	1	Vertical blank video timing signal. Active High
vid_hblank	In	1	Horizontal blank video timing signal. Active High
vid_data	In	8-256	Parallel video input data.
vid_field_id	In	1	Video field. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

使用到的信号有：

Vid_in IP 输入端信号：

Vid_data: 视频数据输入

Vid_active_video: 视频数据有效

Vid_hsync: 视频行同步信号 (非常关键信号, 下面重点分析对象)

Vid_vsync: 视频场同步信号

Vid_io_in_ce: 数据输入有效 (非常关键信号, 下面重点分析对象)

vid_io_in_clk: 这是时钟信号和摄像头时钟同步

Vid_io_in_reset: 这个信号, 高电平的时候复位

有很多读者会问笔者, 这些官方的 IP 如何使用, 这么没有详细的技术手册。还别说, 官方就是没有非常详细的技术手册, 有时候笔者也是使出浑身解数分析 vid_in IP 内部信号时序, 掌握 OV_Sensor_ML 自定义 IP 时序接口设计。

打开 v_vid_in_axi4s_v4_0_1_formatter.v 这个文件

下面对其关键的部分进行说明。

表 3-3-2-1 v_vid_in_axi4s_v4_0_1_formatter.v

```
`timescale 1ps/1ps
`default_nettype none
(* DowngradeIPIdentifiedWarnings="yes" *)
```

```
module v_vid_in_axi4s_v4_0_1_formatter #(
```

```
parameter C_NATIVE_DATA_WIDTH = 24
)(
// System signals
input wire VID_IN_CLK,           // Native video clock
input wire VID_RESET,            // Native video reset
input wire VID_CE,               // Native video clock enable

// Video input signals
input wire VID_ACTIVE_VIDEO,     // Native video input data enable
input wire VID_VBLANK,           // Native video input vertical blank
input wire VID_HBLANK,           // Native video input horizontal blank
input wire VID_VSYNC,             // Native video input vertical sync
input wire VID_HSYNC,             // Native video input horizontal sync
input wire VID_FIELD_ID,         // Native video input field-id
input wire [C_NATIVE_DATA_WIDTH-1:0] VID_DATA, // Native video input data

// Video timing detector signals
output wire VTD_ACTIVE_VIDEO,    // Native video output data enable
output wire VTD_VBLANK,           // Native video output vertical blank
output wire VTD_HBLANK,           // Native video output horizontal blank
output wire VTD_VSYNC,             // Native video output vertical sync
output wire VTD_HSYNC,             // Native video output horizontal sync
output wire VTD_FIELD_ID,         // Native video output field-id
input wire VTD_LOCKED,            // Native video locked signal from VTD

// FIFO write signals
output wire [C_NATIVE_DATA_WIDTH+2:0] FIFO_WR_DATA, // FIFO write data
output wire FIFO_WR_EN            // FIFO write enable
);

// Wire and register declarations
reg de_1 = 0;
reg vblank_1 = 0;
reg hblank_1 = 0;
reg vsync_1 = 0;
reg hsync_1 = 0;
reg [C_NATIVE_DATA_WIDTH -1:0] data_1 = 0;
reg de_2 = 0;
reg v_blank_sync_2 = 0;
reg [C_NATIVE_DATA_WIDTH -1:0] data_2 = 0;
reg de_3 = 0; // DE output register
reg [C_NATIVE_DATA_WIDTH -1:0] data_3 = 0; // data output register
reg vert_blinking_intvl = 0; // SR, reset by DE rising
reg field_id_1 = 0;
```

```
reg field_id_2 = 0;
reg field_id_3 = 0;

wire v_blank_sync_1; // vblank or vsync
wire de_rising;
wire de_falling;
wire vsync_rising;
reg sof;
reg sof_1;
reg eol;
reg vtd_locked;
wire sof_rising;

// Assignments
assign FIFO_WR_DATA    = {field_id_3,sof_1,eol,data_3};
assign FIFO_WR_EN       = de_3 & ~VID_RESET & vtd_locked;
assign VTD_ACTIVE_VIDEO = de_1;
assign VTD_VBLANK      = vblank_1;
assign VTD_HBLANK      = hblank_1;
assign VTD_VSYNC        = vsync_1;
assign VTD_HSYNC        = hsync_1;
assign VTD_FIELD_ID    = field_id_1;

assign v_blank_sync_1 = vblank_1 || vsync_1;
assign de_rising   = de_1 && !de_2;
assign de_falling  = !de_1 && de_2;
assign vsync_rising = v_blank_sync_1 && !v_blank_sync_2;
assign sof_rising  = sof & ~sof_1;

// VTD locked process
always @(posedge VID_IN_CLK) begin
  if(VID_RESET | ~VTD_LOCKED) begin
    vtd_locked <= 1'b0;
  end else if(VID_CE) begin
    vtd_locked <= (sof_rising & VTD_LOCKED) ? 1'b1 : vtd_locked;
  end
end

// input, output, and delay registers
always @ (posedge VID_IN_CLK) begin
  if(VID_RESET) begin
    de_1      <= 1'b0;
    de_2      <= 1'b0;
    de_3      <= 1'b0;
```

```
vblank_1      <= 1'b0;
hblank_1      <= 1'b0;
vsync_1       <= 1'b0;
hsync_1       <= 1'b0;
field_id_1    <= 1'b0;
field_id_2    <= 1'b0;
field_id_3    <= 1'b0;
data_1        <= {C_NATIVE_DATA_WIDTH{1'b0}};
data_2        <= {C_NATIVE_DATA_WIDTH{1'b0}};
data_3        <= {C_NATIVE_DATA_WIDTH{1'b0}};
v_blank_sync_2 <= 1'b0;
eol           <= 1'b0;
sof            <= 1'b0;
sof_1          <= 1'b0;
end else if(VID_CE) begin
    de_1      <= VID_ACTIVE_VIDEO;
    de_2      <= de_1;
    de_3      <= de_2;
    vblank_1   <= VID_VBLANK;
    hblank_1   <= VID_HBLANK;
    vsync_1    <= VID_VSYNC;
    hsync_1    <= VID_HSYNC;
    field_id_1 <= VID_FIELD_ID;
    field_id_2 <= field_id_1;
    field_id_3 <= field_id_2;
    data_1     <= VID_DATA;
    data_2     <= data_1;
    data_3     <= data_2;
    v_blank_sync_2 <= v_blank_sync_1;
    eol         <= de_falling;
    sof          <= de_rising && vert_blinking_intvl;
    sof_1        <= sof;
end
end

// Vertical back porch SR register
always @ (posedge VID_IN_CLK) begin
    if (VID_CE) begin
        if (vsync_rising) // falling edge of vsync
            vert_blinking_intvl <= 1;
        else if (de_rising) // rising edge of data enable
            vert_blinking_intvl <= 0;
    end
end
```

```
endmodule
```

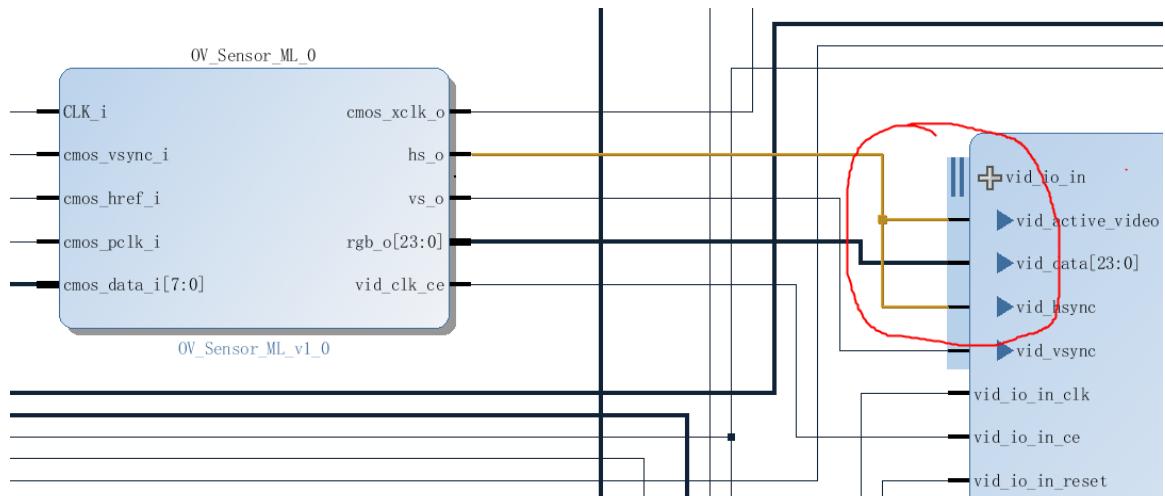
在上面代码中，

```
eol      <= de_falling;
sof      <= de_rising && vert_blanking_intvl;
```

eol 实际就是 tlast 信号, 而 sof 就是 tuser 信号。tlast 信号代表每行图像数据的最后一个数据, tuser 代表每场数据的第一个数据。

所有非常关键的信号都和 de_falling 和 vert_blanking_intvl 有关系。

hs_o 和 vid_in IP 的连接关系。



上图中, 被红色圈起来的 hs_o 信号, 同时接到了 vid_in_ip 的 vid_active_video 和 vid_hsync 信号接口。因此, de 信号就是 hs_o 信号, 而 vid_hsync 我们发现没有任何作用, 也就是说不 hs_o 不连接到 vid_hsync 也不影响这里的程序工作。

VID_CE 这个参数就是前面的 vid_io_in_ce 信号, 可以看出这个芯片有效的时候相对应的时序电路才会执行。在本工程中, 摄像头每 2 个 pc1k 输出 1 个有效的数据, 而 vid_in IP 如果 VID_CE 为 1 则数据输入会每个时钟输入 1 个就错了。因此官方的 IP 设计的还是很不错考虑周到, 通过 VID_CE 这个条件, 控制时钟同步。

```
...
else if(VID_CE) begin
...
end
...
```

现在回到 OV_Sensor_ML 的 cmos_decode_v1.v 文件中有一段红色的代码如下:

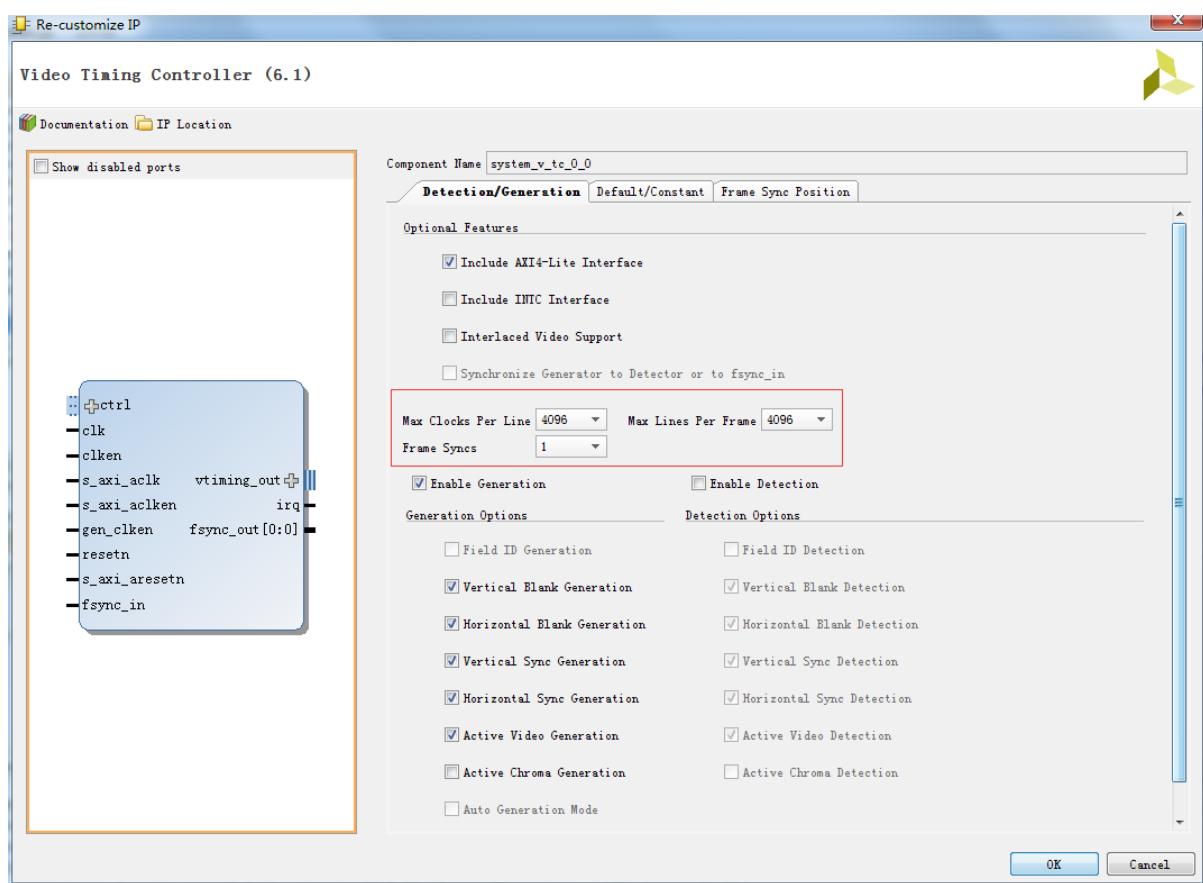
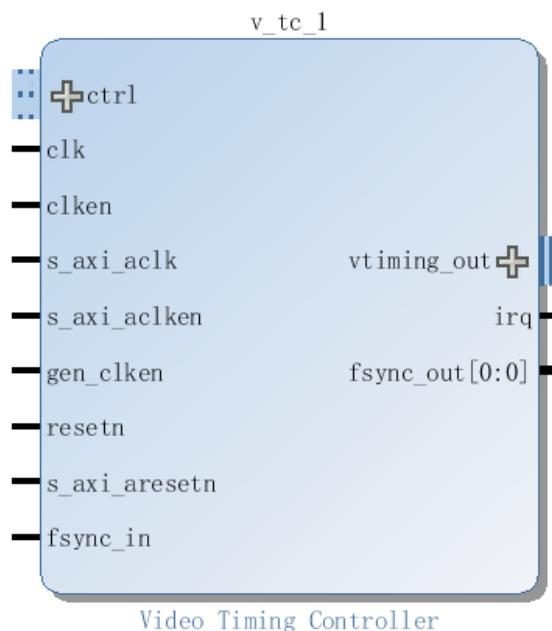
```
assign vid_clk_ce = out_en ? (byte_flag_r0&hs_o)||(!hs_o) : 1'b0;
```

这段代码控制了 vid_clk_ce 的正确输出, 关键部分是 `(byte_flag_r0&hs_o)||(!hs_o)`。当 hs_o 有效的时候, vid_in 的 VID_CE 信号就有效, 当 hs_o=0 的时候 VID_CE 必须仍然有效, 这样才能检测到 vsync_rising 信号了, 检测到了 vsync_rising 才能有 vert_blanking_intvl 为 1, 才有 tuser 信号。

好了罗嗦了半天, 终于解释完了, 如果有不清楚的, 找我们技术支持吧。

4.5 VTC IP 的分析

4.5.1 VTC IP 的参数介绍



这个 IP 就是一个时序发生器，产生显示器输出所需要的时序信号。

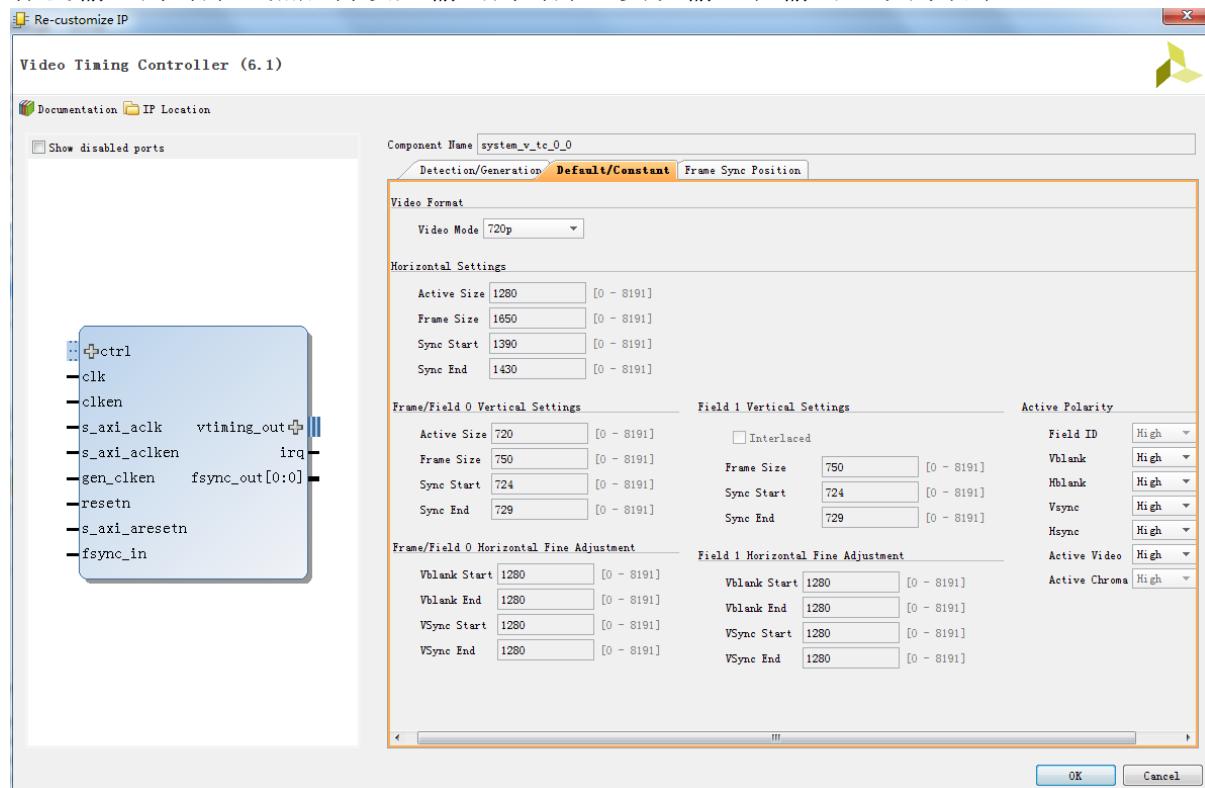
这个页面中，include AXI-lite interface 可以不勾选，不勾选就只能采用默认设置，无法在 C 语言中灵活配置了，所以笔者这里建议大家勾选吧。max clocks per line 和 max_lines per frame 需要设置下，当设置到 4096 的时候可以支持分辨率到最大，当然消耗的资源也更多。笔者这里太奢侈了设置了 4096。实际上设置到 2048 就够用了。本页面的其他信号可以采取默认设置。

Enable Generation:

支持产生时序，这个肯定必须勾选的。

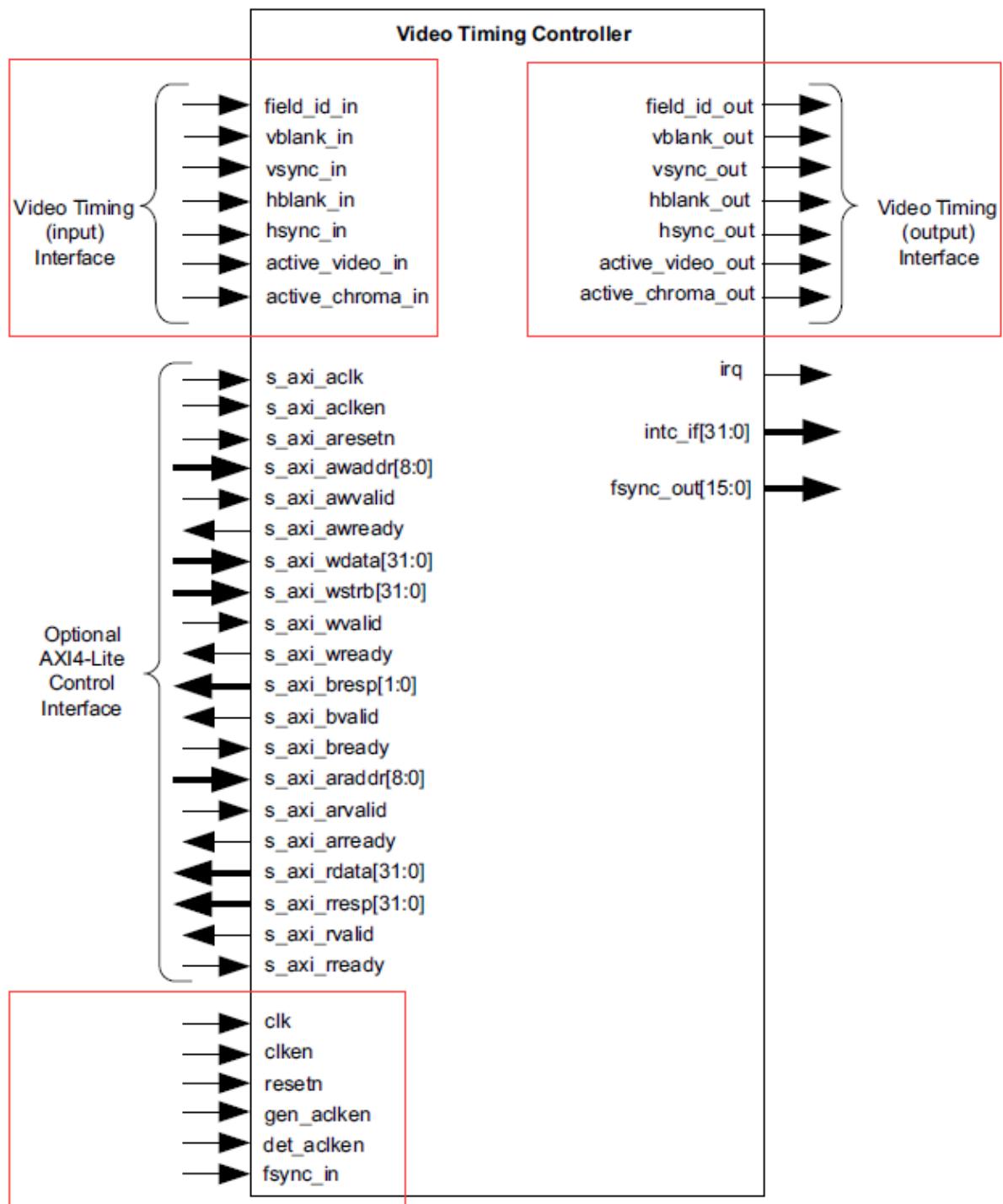
Enable Detection:

支持时序捕捉，这个不是必须的，根据需要而定，如果设置了这个选项，就可以先捕捉输入的时序，然后再设置输出的时序，实现输入和输出一致的效果。



在这个页面中，只要选择需要支持的分辨率就可以了，当然不设置也没关系的，因为我们在 C 代码综合那个会进一步设置的。

4.5.2 VTC IP 接口信号的定义



红色方框内的绝大部分信号需要我们手动联系，所以下面重点是讲解红色方框内的信号作用，至于 AXI4-LITE 接口主要是用来设置参数的。

Common Port Descriptions:

Name	Direction	Width	Description
clk	In	1	Video Core Clock
clken	In	1	Video Core Active High Clock Enable
det_clken	In	1	Video Timing Detection Core Active High Clock Enable
gen_clken	In	1	Video Timing Generator Core Active High Clock Enable
resetn	In	1	Video Core Active Low Synchronous Reset
irq	Output	1	Interrupt request output, active high edge
intc_if	Output	32	OPTIONAL EXTERNAL INTERRUPT CONTROLLER INTERFACE Available when the "Include INTC Interface" or C_HAS_INTC_IF has been selected. Bits [31:8] are the same as the bits [31:8] in the status register (0x0004). Bits [5:0] are the same as bits [21:16] of the error register (0x0008). Bits [7:6] are reserved and are always 0.

Detector Interface (Video Timing Input Interface)			
field_id_in	Input	1	INPUT FIELD ID Used to set the field_id polarity in the Detector Polarity Register (Address Offset 0x002C). Optional. Only valid when interlace support and field id are enabled.
hsync_in	Input	1	INPUT HORIZONTAL SYNCHRONIZATION Used to set the DETECTOR HSYNC register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hsync_in input is not connected, then the "Horizontal Sync Detection" option must be deselected.
hblank_in	Input	1	INPUT HORIZONTAL BLANK Used to set the DETECTOR HSIZE register. Polarity is auto-detected. Optional. Either horizontal blank or horizontal synchronization signal inputs must be present. Both do not have to be present. If the hblank_in input is not connected, then the "Horizontal Blank Detection" option must be deselected.
vsync_in	Input	1	INPUT VERTICAL SYNCHRONIZATION Used to set the DETECTOR F0_VSYNC_V and the F0_VSYNC_H registers. Polarity is auto-detected. Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vsync_in input is not connected, then the "Vertical Sync Detection" option must be deselected.

Name	Direction	Width	Description
vblank_in	Input	1	<p>INPUT VERTICAL BLANK Used to set the DETECTOR_VSIZE and the F0_VBLANK_H registers. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the vblank_in input is not connected, then the "Vertical Blank Detection" option must be deselected.</p>
active_video_in	Input	1	<p>INPUT ACTIVE VIDEO Used to set the DETECTOR ACTIVE_SIZE register. Polarity is auto-detected.</p> <p>Optional. One of the following inputs must be present: active video, vertical blank or vertical synchronization. If the active_video_in input is not connected, then the "Active Video Detection" option must be deselected.</p>
active_chroma_in	Input	1	<p>INPUT ACTIVE CHROMA Used to set the VIDEO_FORMAT and the CHROMA_PARITY bits in the Detector Encoding Register. Polarity is auto-detected.</p> <p>Optional. If the active_chroma_in input is not connected, then the "Active Chroma Detection" option must be deselected.</p>

Generator Interface (Video Timing Output Interface)			
field_id_out	Output	1	<p>OUTPUT FIELD ID Generated field id signal. Polarity configured by the Generator Polarity Register (Address Offset 0x006C) Optional. Only enabled when interlaced support and field id generation is enabled.</p>
hsync_out	Output	1	<p>OUTPUT HORIZONTAL SYNCHRONIZATION Generated horizontal synchronization signal. Polarity configured by the control register. Asserted active during the cycle set by the HSYNC_START bits and deasserted during the cycle set by the HSYNC_END bits in the GENERATOR HSYNC register.</p>
hblank_out	Output	1	<p>OUTPUT HORIZONTAL BLANK Generated horizontal blank signal. Polarity configured by the control register. Asserted active during the cycle set by ACTIVE_HSIZEx and deasserted during the cycle set by the FRAME_HSIZEx bits in the GENERATOR HSIZEx register.</p>
vsync_out	Output	1	<p>OUTPUT VERTICAL SYNCHRONIZATION Generated vertical synchronization signal. Polarity configured by the control register. Asserted active during the line set by the F#_VSYNC_VSTART bits and deasserted during the line set by the F#_VSYNC_VEND bits in the GENERATOR F#_VSYNC_V registers.</p>

Name	Direction	Width	Description
vblank_out	Output	1	OUTPUT VERTICAL BLANK Generated vertical blank signal. Polarity configured by the control register. Asserted active during the line set by the ACTIVE_VSIZE bits and deasserted during the line set by the GENERATOR VSIZE register.
active_video_out	Output	1	OUTPUT ACTIVE VIDEO Generated active video signal. Polarity configured by the control register. Active for non blanking lines. Asserted active during the first cycle of the field/frame and deasserted during the cycle set by the GENERATOR ACTIVE_SIZE register
active_chroma_out	Output	1	OUTPUT ACTIVE CHROMA Generated active chroma signal. Denotes which lines contain valid chroma samples (used for YUV 4:2:0). Polarity configured by the GENERATOR POLARITY register. Active for non-blanking lines configured by the VIDEO_FORMAT and the CHROMA_PARITY bits in the GENERATOR Encoding Register.
Frame Synchronization Interface			
fsync_out	Output	[Frame Syncs - 1:0]	FRAME SYNCHRONIZATION OUTPUT Each Frame Synchronization bit toggles for only one clock cycle during each frame. The number of bits is configured with the Frame Syncs GUI parameter. Each bit is independently configured for horizontal and vertical clock cycle position with the Frame Sync 0-15 Config registers).
fsync_in	Input	1	FRAME SYNCHRONIZATION INPUT This is a one clock cycle pulse (active high) input. The video timing generator will be synchronized to the input if used.

本例子中没有使用到输入时序的捕捉，因此笔者下面只对用到的信号做一些介绍。

hsync_out:

产生行同步输出

hsync_out:

产生行消影

vsync_out:

产生场同步输出

vblank_out:

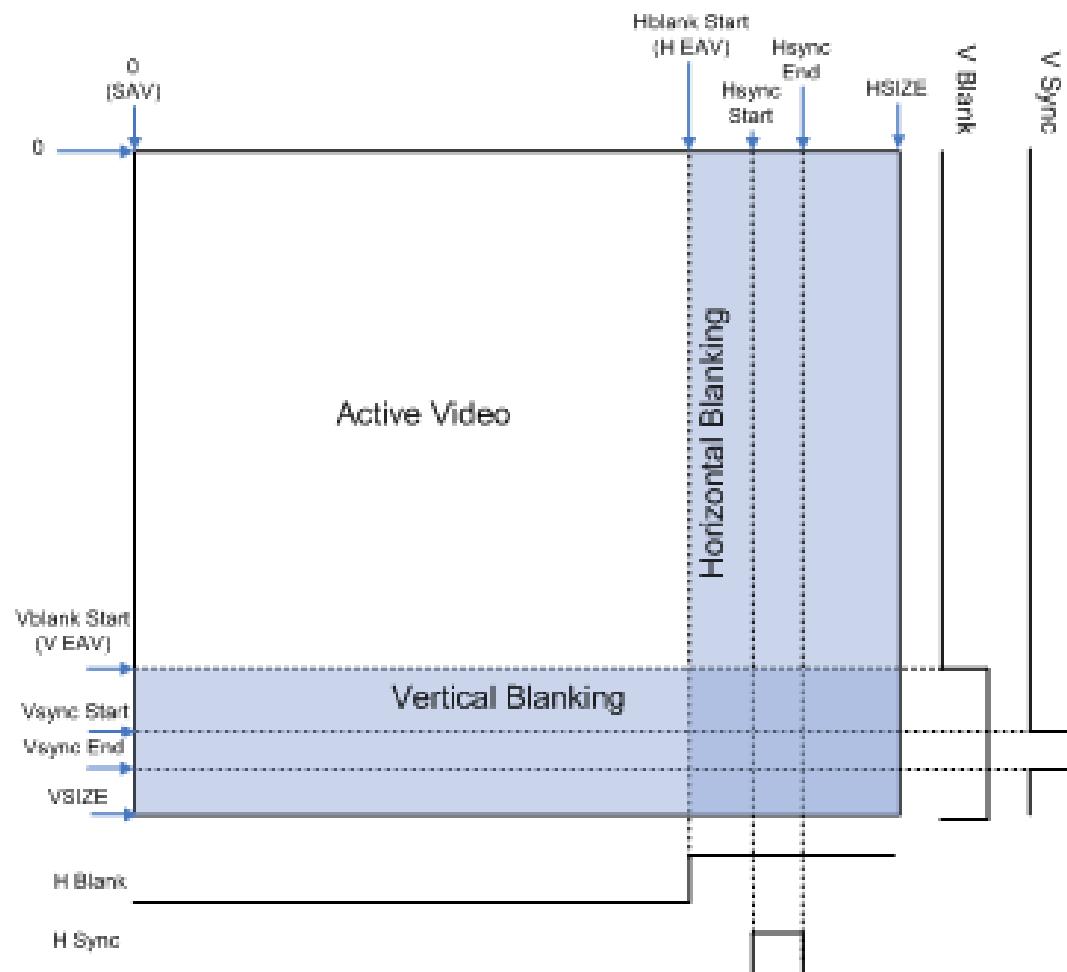
产生场消影

active_video_out:

有效数据输出

4.5.3 VTC IP 配置寄存器

shows the start of the horizontal front porch (Hblank Start), synchronization (Hsync Start), back porch (Hsync End) and active video (SAV). It also shows the start of the vertical front porch (Vblank Start), synchronization (Vsync Start), back porch (Vsync End) and active video (SAV). The total number of horizontal clock cycles is HSIZE and the total number of lines is the VSIZE.



Generator Active Size Register (Address Offset 0x0060)

0x0060	GENERATOR ACTIVE_SIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
ACTIVE_VSIZE	28:16	Generated Vertical Active Frame Size. The height of the frame without blanking in number of lines.
RESERVED	15:13	Reserved
ACTIVE_HSIZE	12:0	Generated Horizontal Active Frame Size. The width of the frame without blanking in number of pixels/clocks.

这是重要的寄存器用来设置有效的行数量和场数量

Generator Timing Status Register (Address Offset 0x0064)

0x0064	GENERATOR TIMING_STATUS	Read
Name	B its	Description
RESERVED	31:3	Reserved
GEN_ACTIVE_VIDEO	2	Generated Active Video Interrupt Status. Set high during the first cycle the output active video is asserted.
GEN_VBLANK	1	Generated Vertical Blank Interrupt Status. Set high during the first cycle the output vertical blank is asserted.
RESERVED	0	Reserved

GEN_ACTIVE_VIDEO:当第一帧图像输出时候置 1

GEN_VBLANK:第一帧有效图像的 blank 信号输出的时候置 1

Generator Encoding Register (Address Offset 0x0068)

0x0068	GENERATOR ENCODING	Read/Write
Name	B its	Description
RESERVED	31:10	Reserved
CHROMA_PARITY	9:8	Generated Chroma Parity 0: Chroma Active during even active-video lines of frame. Active every pixel of active line 1: Chroma Active during odd active-video lines of frame. Active every pixel of active line 2: Chroma Active during even active video lines of frame. Active every even pixel of active line, inactive every odd pixel 3: Chroma Active during odd active video lines of frame. Active every even pixel of active line, inactive every odd pixel
FIELD_ID_PARITY	7	Generated Field ID Parity 0: Field ID input is currently low 1: Field ID input is currently high
INTERLACED	6	Generated Progressive/Interlaced 0: Generated video format is progressive 1: Generated video format is interlaced
RESERVED	5:4	Reserved
VIDEO_FORMAT	3:0	Generated Video Format Denotes when the active_chroma signal is active. 0: YUV 4:2:2 - Active_chroma is active during the same time active_video is active. 1: YUV 4:4:4 - Active_chroma is active during the same time active_video is active. 2: RGB - Active_chroma is active during the same time active_video is active. 3: YUV 4:2:0- Active_chroma is active every other line during the same time active_video is active. See The CHROMA_PARITY bits to control which lines and pixels.

CHROMA_PARITY: 奇偶色度 (读者没明白)

FIELD_ID_PARITY: 奇偶场标志

INTERLACED: 视频格式是渐进式还是各行扫描

VIDEO_FORMAT: 视频格设置, 有 YUV422 YUV444 YUV420 RGB

Generator Polarity Register (Address Offset 0x006C)

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
RESERVED	31:7	Reserved
FIELD_ID_POL	6	Generated Field ID Polarity 0: Low during Field 0 and High during Field 1 1: High during Field 0 and Low during Field 1
ACTIVE_CHROMA_POL	5	Generated Active Chroma Polarity 0: Active Low Polarity 1: Active High Polarity

0x006C	GENERATOR POLARITY	Read/Write
Name	B its	Description
ACTIVE_VIDEO_POL	4	Generated Active Video Polarity 0: Active Low Polarity 1: Active High Polarity
HSYNC_POL	3	Generated Horizontal Sync Polarity 0: Active Low Polarity 1: Active High Polarity
VSYNC_POL	2	Generated Vertical Sync Polarity 0: Active Low Polarity 1: Active High Polarity
HBLANK_POL	1	Generated Horizontal Blank Polarity 0: Active Low Polarity 1: Active High Polarity
VBLANK_POL	0	Generated Vertical Blank Polarity 0: Active Low Polarity 1: Active High Polarity

这个寄存器设置相应的场输出极性和色度输出极性。

Generator Horizontal Frame Size Register (Address Offset 0x0070)

0x0070	GENERATOR HSIZE	Read/Write
Name	B its	Description
RESERVED	31:13	Reserved
FRAME_HSIZE	12:0	Generated Horizontal Frame Size. The width of the frame with blanking in number of pixels/clocks.

一副图像的一行的大小，包括了消隐和有效数据阶段。

Generator Vertical Frame Size Register (Address Offset 0x0074)

0x0074	GENERATOR VSIZE	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
FIELD1_VSIZE	28:16	Generated Vertical Field 1 Size. The height with blanking in number of lines of field 1.
FRAME_VSIZE	12:0	Generated Vertical Frame Size. The height of the frame with blanking in number of lines.

一副图像的一场的大小，包括了消隐和有效数据阶段。

Generator Horizontal Sync Register (Address Offset 0x0078)

0x0078	GENERATOR HSYNC	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
Hsync_End	28:16	Generated Horizontal Sync End End cycle index of horizontal sync. Denotes the first cycle hsync_in is de-asserted.
RESERVED	15:13	Reserved
Hsync_Start	12:0	Generated Horizontal Sync End Start cycle index of horizontal sync. Denotes the first cycle hsync_in is asserted.

设置行的水平同步结束和同步开始

Generator Frame/Field 0 Vertical Blank Cycle Register (Address Offset 0x007C)

0x007C	GENERATOR F0_VBLANK_H	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VBLANK_HEND	28:16	Generated Vertical Blank Horizontal End End Cycle index of vertical blank. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F0_VBLANK_HSTART	12:0	Generated Vertical Blank Horizontal Start Start Cycle index of vertical blank. Denotes the first cycle vblank_in is asserted.

设置 Fram/Field0 的水平消隐结束和开始

Generator Frame/Field 0 Vertical Sync Line Register (Address Offset 0x0080)

0x0080	GENERATOR F0_VSYNC_V	Read/Write
Name	B its	Description
RESERVED	31:29	Reserved
F0_VSYNC_VEND	28:16	Generated Vertical Sync Vertical End End Line index of vertical sync. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_VSTART	12:0	Generated Vertical Sync Vertical Start Start line index of vertical sync. Denotes the first line vsync_in is asserted.

设置 Fram/Field0 的垂直同步垂直结束和开始

Generator Frame/Field 0 Vertical Sync Cycle Register (Address Offset 0x0084)

0x0084	GENERATOR F0_VSYNC_H	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
F0_VSYNC_HEND	28:16	Generated Vertical Sync Horizontal End End cycle index of vertical sync. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F0_VSYNC_HSTART	12:0	Generated Vertical Sync Horizontal Start Start cycle index of vertical sync. Denotes the first cycle vsync_in is asserted.

设置 Fram/Field0 的垂直同步水平结束和开始

Generator Field 1 Vertical Blank Cycle Register (Address Offset 0x0088)

0x0088	GENERATOR F1_VBLANK_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VBLANK_HEND	28:16	Generated Field 1 Vertical Blank Horizontal End End Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is de-asserted.
RESERVED	15:13	Reserved
F1_VBLANK_HSTART	12:0	Generated Field 1 Vertical Blank Horizontal Start Start Cycle index of vertical blank for field 1. Denotes the first cycle vblank_in is asserted.

设置 Field1 的水平消隐结束和开始

Generator Field 1 Vertical Sync Line Register (Address Offset 0x008C)

0x008C	GENERATOR F1_VSYNC_V	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_VEND	28:16	Generated Field 1 Vertical Sync Vertical End End Line index of vertical sync for field 1. Denotes the first line vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_VSTART	12:0	Generated Field 1 Vertical Sync Vertical Start Start line index of vertical sync for field 1. Denotes the first line vsync_in is asserted.

设置 Field1 的垂直同步垂直结束和开始

Generator Field 1 Vertical Sync Cycle Register (Address Offset 0x0090)

0x0090	GENERATOR F1_VSYNC_H	Read
Name	Bits	Description
RESERVED	31:29	Reserved
F1_VSYNC_HEND	28:16	Generated Field 1 Vertical Sync Horizontal End End cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is de-asserted.
RESERVED	15:13	Reserved
F1_VSYNC_HSTART	12:0	Generated Field 1 Vertical Sync Horizontal Start Start cycle index of vertical sync for field 1. Denotes the first cycle vsync_in is asserted.

设置 Field1 的垂直同步水平结束和开始

Frame Sync 0 - 15 Configuration Registers (Address Offsets 0x0100 - 0x013C)

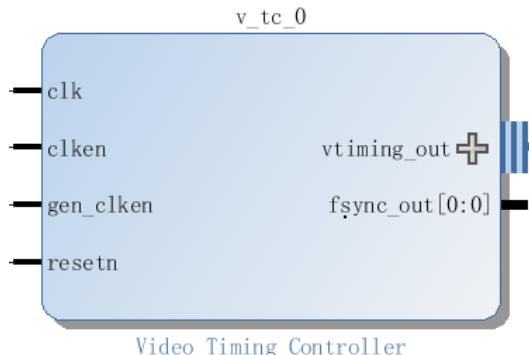
0x0100	FRAME SYNC 0 CONFIG	Read/Write
Name	Bits	Description
RESERVED	31:29	Reserved
V_START	28:16	FRAME SYNCHRONIZATION VERTICAL START Vertical line during which the fsync_out[0] output port is asserted active-high. Note: Frame Syncs are not active during the complete line, only in the cycle during which both the V_START and H_START are valid each frame.
RESERVED	15:13	Reserved
H_START	12:0	FRAME SYNCHRONIZATION HORIZONTAL START Horizontal Cycle during which fsync_out[0] output port is asserted active-high

Generator Global Delay Register (Address Offset 0x140)

0x140	Generator Global Delay	Read/Write
Name	Bits	Description
Reserved	31:29	Reserved
V_DELAY	28:16	GENERATOR VERTICAL DELAY Vertical line offset. This is the number of lines that the generated output will be shifted relative to the detector (input timing). The vertical delay is only available when both the detector and generator are enabled. Can be combined with the H_DELAY.
Reserved	15:13	Reserved
H_DELAY	12:0	GENERATOR HORIZONTAL DELAY Horizontal cycle offset. This is the number of clock cycles that the generated output will be shifted relative to the detector (input timing). The horizontal delay is only available when both the detector and generator are enabled. Can be combined with the V_DELAY.

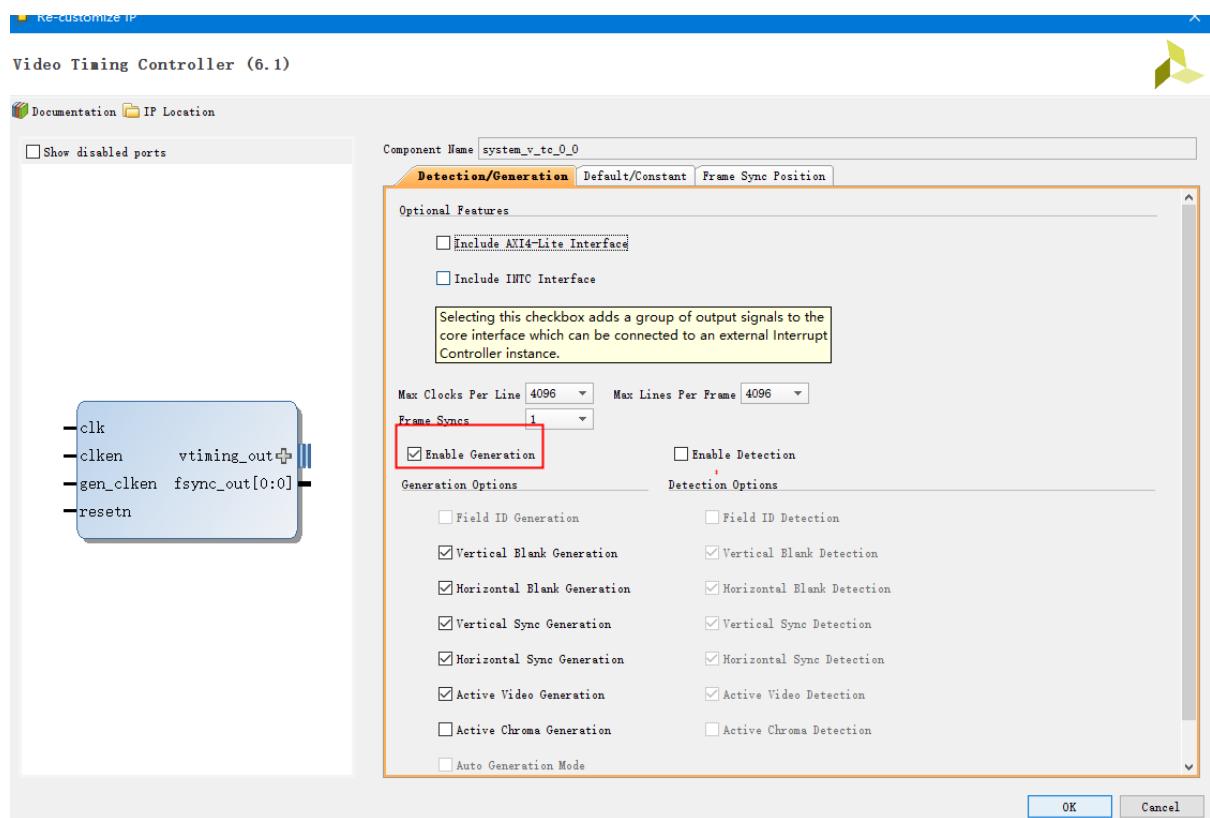
4.5.5 设置 VTC IP

讲了这么多实际上我们用的时候很简单,所以只要这么简单。

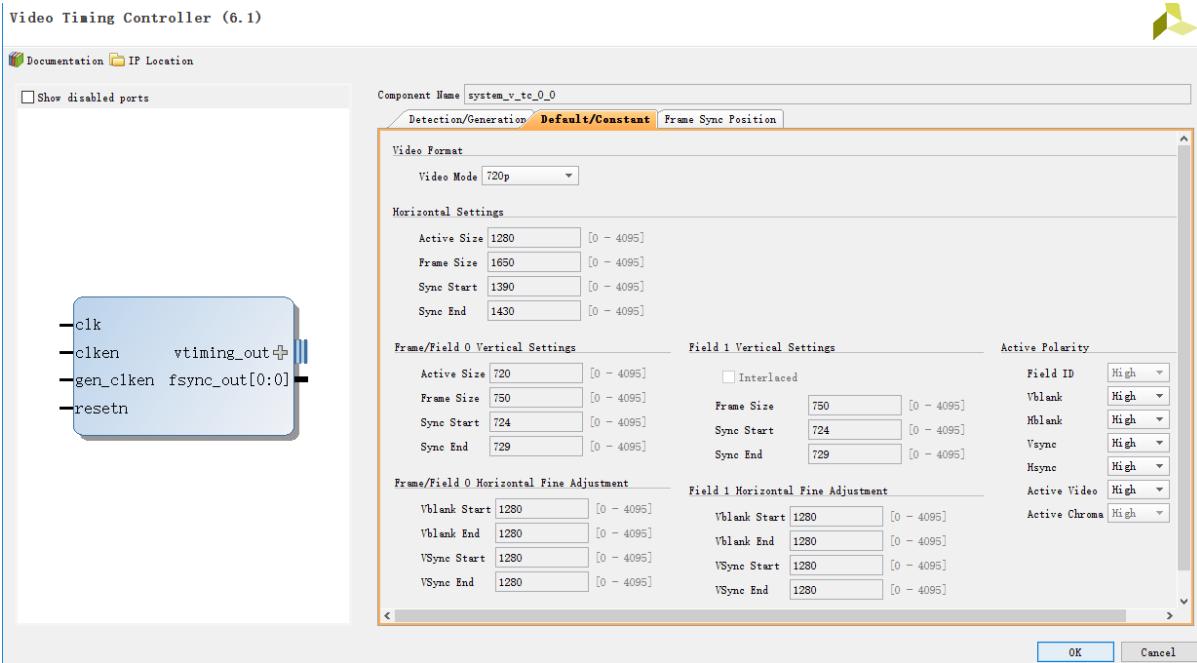


Video Timing Controller

由于不使用动态配置，并且只使用了视频时序产生，所以只要勾选如下复选框。

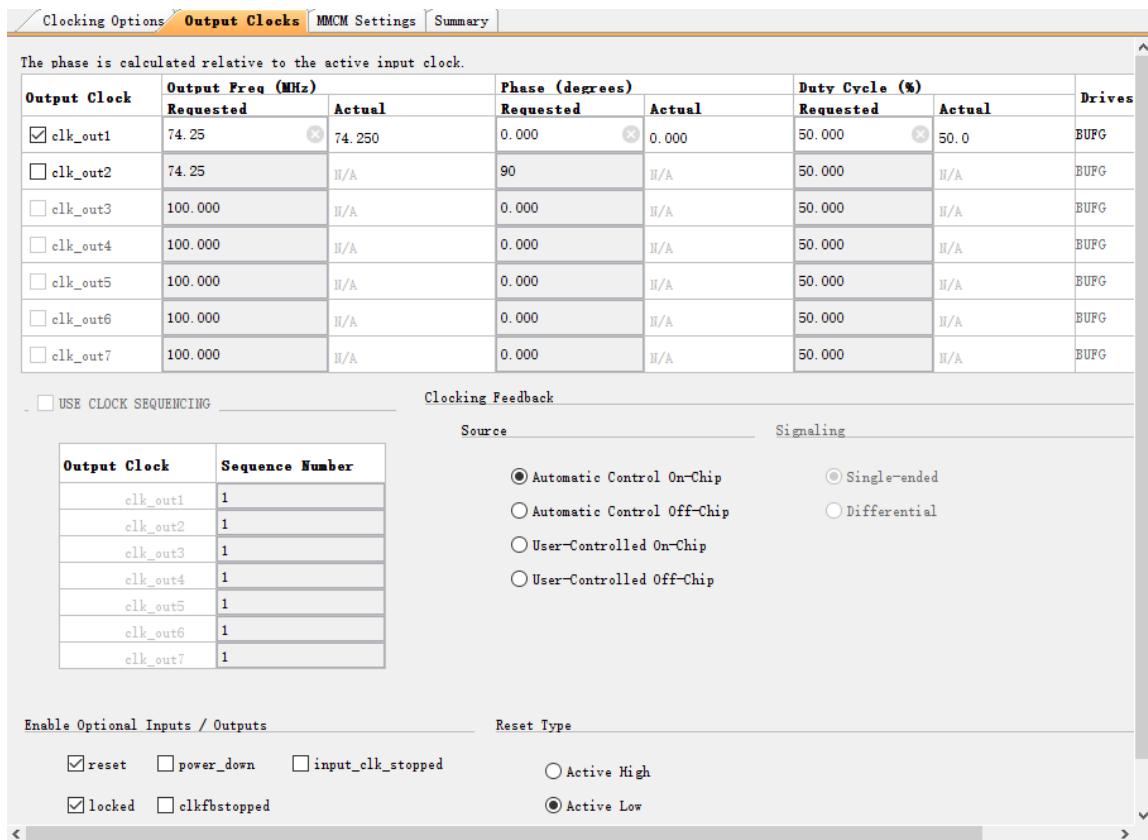


由于 OV5640 分辨率是 1280X720 因此直接选择 720P 即可。



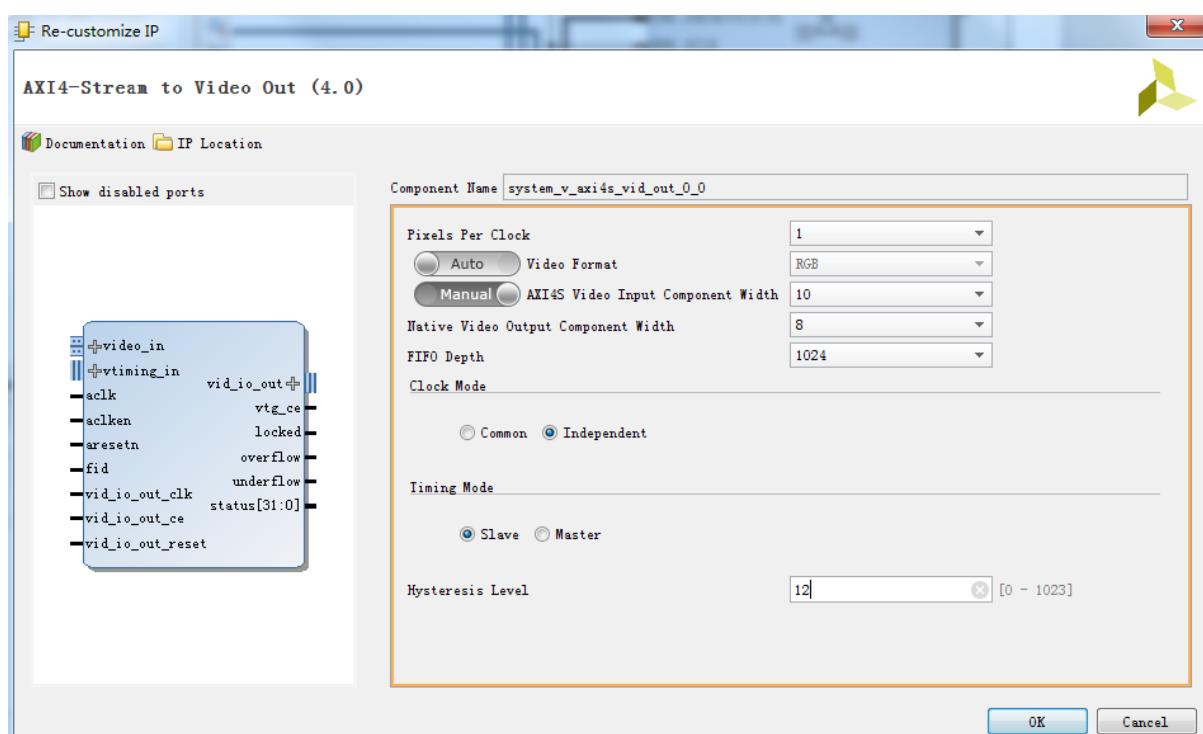
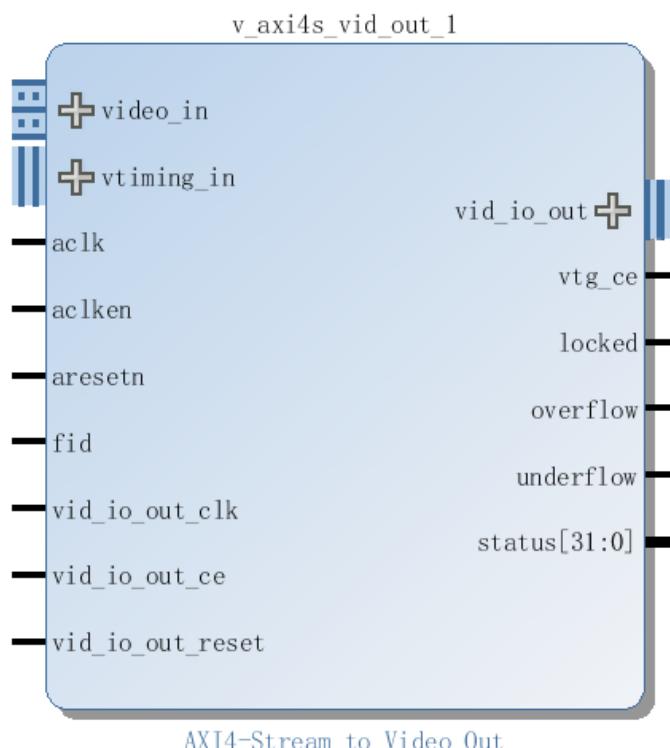
4.6 PLL 时钟设置

由于这里的分辨率是 1280X720 因此提供给 VTC IP 和 VID OUTIP 的时钟 74.5M 就可以了



4.7 VID_OUT IP 的分析

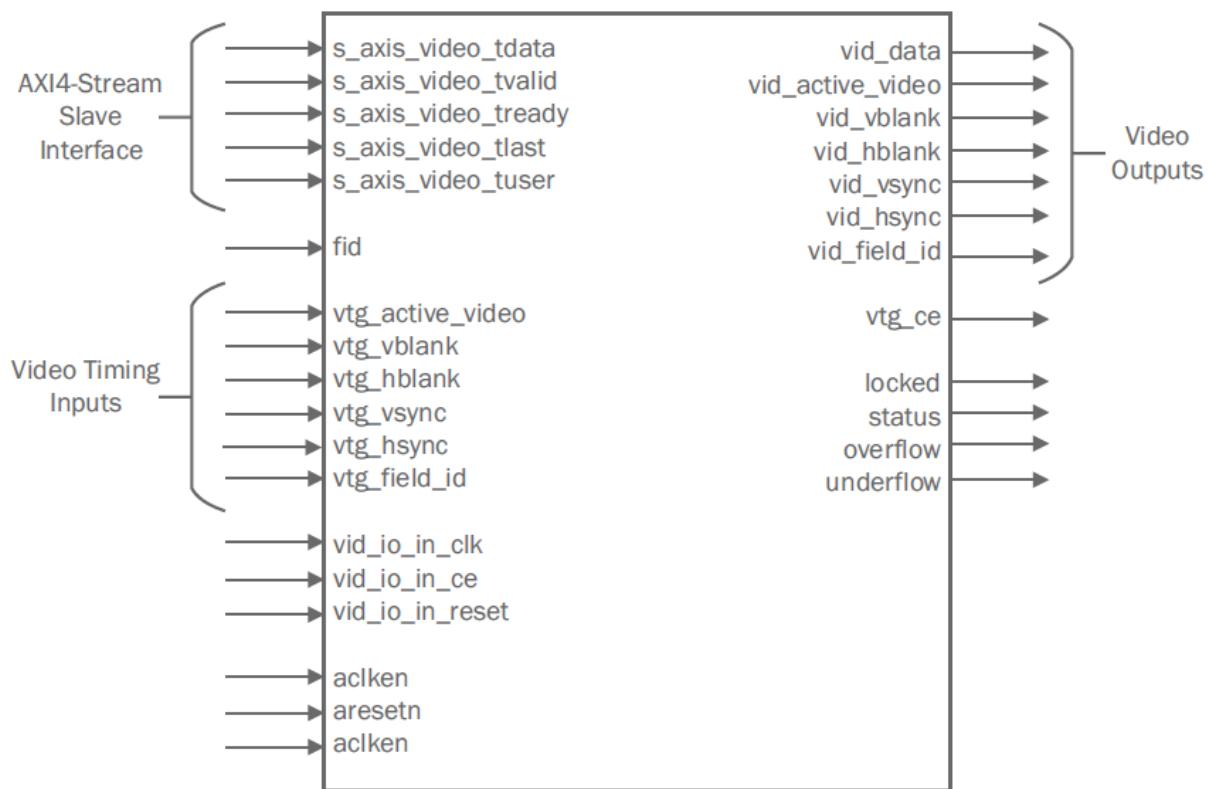
4.7.1 VID_OUT 的参数介绍



这些参数和前面的 V_TPG 参数类似

- Pixels Per Clock: 设置每个时钟输出的像素个数，可以是 1、2、4
- Input Component Width: 输入像素的宽度，这个参数影响 TDATA 的位宽
- Output Component Width: 输出像素的宽度
- Clock Mode: 时钟的模式，可以选择独立时钟，或者共享时钟
- Video Format: 视频格式
- FIFO Depth: FIFO 深度
- Hysteresis Level: 滞后输出

4.7.2 VID_OUT IP 接口信号的定义



Signal Name	Direction	Width	Description
aclk	Input	1	AXI4-Stream ACLK.
aclken	Input	1	AXI4-Stream ACLKEN. Active High.
aresetn	Input	1	AXI4-Stream ARESETN. Active Low. Synchronous to ACLK.
fid	Input	1	Field-ID for AXI4-Stream bus. Used only for interlaced video: 0=even field, 1=odd field. This bit changes coincident with SOF on the AXI4-Stream bus. It should be connected to the field-ID bit of the next device downstream that is field-aware, otherwise it should be left unconnected.
vid_io_out_ce	Input	1	Native video clock enable
vid_io_out_reset	Input	1	Native video clock domain reset. Synchronous to vid_io_out_clk. Only available in independent clock mode. Active High.
vtg_ce	Output	1	VTC clock enable. Used to halt the timing generator for synchronization purposes.
locked	Output	1	Flag indicating whether the VTC is locked to the input timing. 1=locked. Synchronous to vid_io_out_clk.
overflow	Output	1	Flag indicating that the FIFO has over-flowed. Synchronous to vid_io_in_clk.

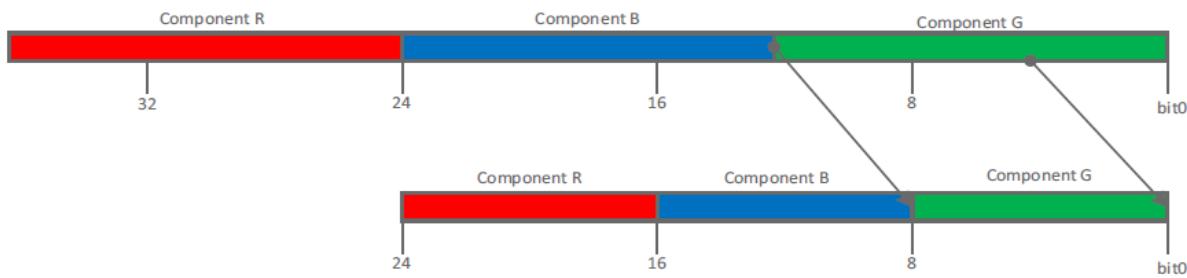
Video Timing Interface

Signal Name	Direction	Width	Description
vtg_vsync	In	1	VTC vertical sync. Active High
vtg_hsync	In	1	VTC horizontal sync. Active High
vtg_vblank	In	1	VTC vertical blank. Active High
vtg_hblank	In	1	VTC horizontal blank. Active High
vtg_act_vid	In	1	VTC active video signal. 1 = active video, 0 = blanked video
vtg_field_id	In	1	VTC field ID. Used only for interlace. 0= even field, 1= odd field. Tie LOW for non-interlace operation.

AXI4 - Stream Interface

Signal Name	Direction	Width	Description
s_axis_video_tvalid	Input	1	AXI4-Stream TVALID. Active video data enable
s_axis_video_tuser	Input	1	AXI4-Stream TUSER. Start of Frame
s_axis_video_tlast	Input	1	AXI4-Stream TLAST. End of Line
s_axis_video_tready	Output	1	AXI4-Stream TREADY. Inverted FIFO full

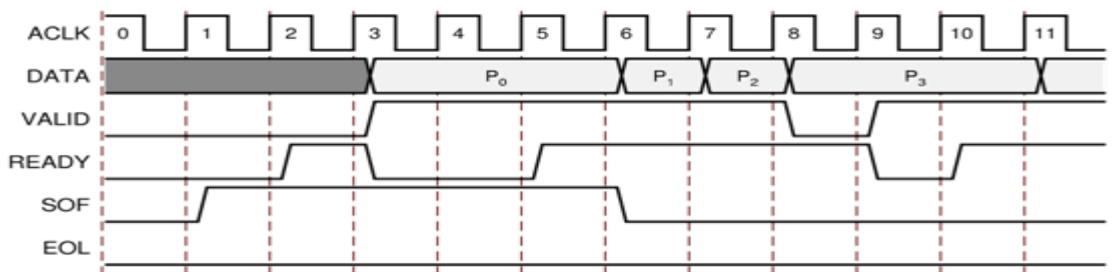
对于 s_axis_video_tdata(TDATA)需要注意一些事情,一般情况下我们的 RGB888 输出,但是,如果 s_axis_video_tdata 是 32bit 那么 VID_OUT IP 会自动截取到 24bit。由于技术手册只给出了 12bit 到 8bit 的截取方式,也就是 RGB 12:12:12 到 RGB 8:8:8 如下图:



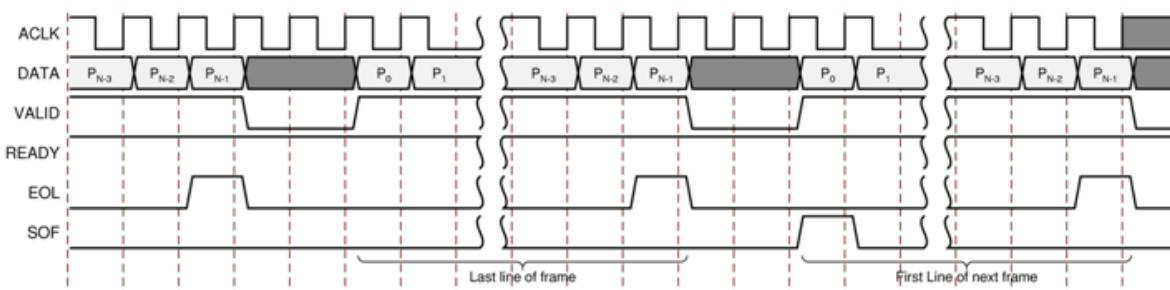
这种截取比较简单，把每个色度的低 4bit 截取就可以了。但是如果是 RGB10:10:10，官方并没有给出截取方式，但是可以通过纯色输出来进行测试。

因此最简单的办法是无需任何截取了，如果 s_axis_video_tdata 是 RGB8:8:8 那就无需任何截取，笔者设计的时候由于 AXI 总线是 32bit 因此数据的低 24bit 为 RGB 8:8:8 只要去掉高 24-31bit 就可以取得 RGB8:8:8 这样最省事。

以下时序图是在 SOF 是一帧图像的开始，当 VALID 和 READY 有效的时候开始传输数据。



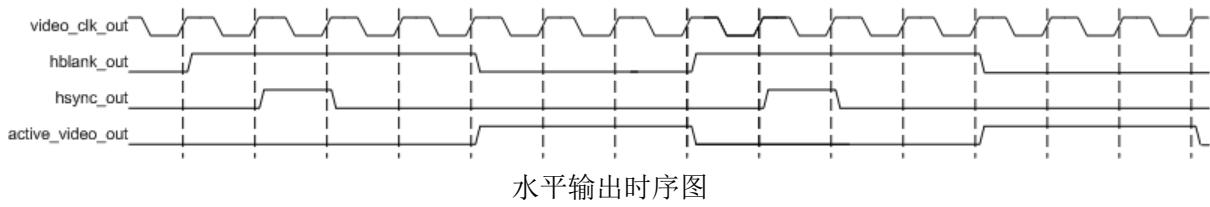
EOL 代表每一行的最后一个数据, SOF 代表前一帧的最后一行的结束, 下一帧第一行的开始。SOF 为 1 个 PLUS 有效 (pg044 v axis out.pdf 没有描述清楚, 而且有错误)。



Example Horizontal Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0003_0003
0x0070	Generator HSize	0x0000_0007
0x0078	Generator HSync	0x0005_0004
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

设置水平输出的相关寄存器

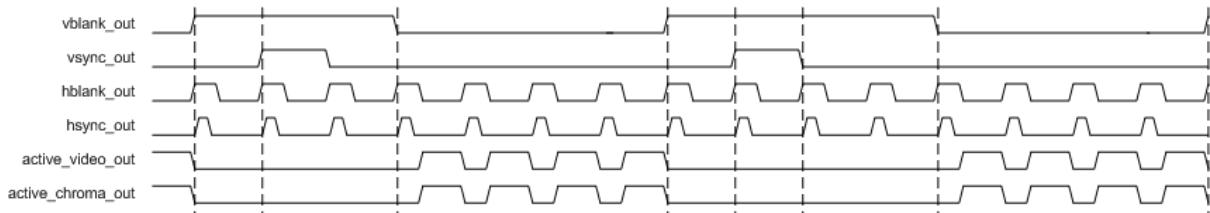


水平输出时序图

Example Vertical Generation Register Inputs

Register Address	Register Name	Value
0x0060	Generator Active Size	0x0004_0003
0x0070	Generator HSize	0x0000_0007
0x0074	Generator VSize	0x0000_0008
0x0078	Generator HSync	0x0005_0004
0x0080	Generator Frame 0 Vsync	0x0006_0005
0x0068	Generator Encoding	0x0000_0000
0x006C	Generator Polarity	0x0000_003f
0x0000	Control	0x01ff_ff07

设置垂直输出的相关寄存器



垂直输出时序图

4.8 FPGA 实现的用户逻辑代码

4.8.1 关键信号 1

```
assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready &
m_axis_video_tlast &(vid_in_v_cnt == VID_IN_VS); // dma in last signal
```

`m_axis_video_tvalid`:此信号是 vid in IP 输出的，代表输出数据有效

`s_axis_s2mm_tready`:此信号是 DMA IP 输出的，代表 DMA 可以接收数据

`m_axis_video_tlast`:这是每一行图像数据的最后一个像素的信号标志

`vid_in_v_cnt == VID_IN_VS`:表示一副图像的最后一个像素输出。

`s_axis_s2mm_tlast`:所有这些信号有效的时候代表 DMA 的最后一个数据

s_axis_s2mm_tlast 信号有效。

4.8.2 关键信号 2

```
assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready &
```

(vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); //vid out user

m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。

s_axis_video_tready: vid out IP 准备好了, 可以接收数据

(vid_out_h_cnt == 11'd0) & (vid_out_v_cnt == 11'd0); 行计数器为 0 场计数器也为 0 说明要么这副图像已经结束, 也可以理解为下一副图像开始前。这样结合 s_axis_video_tready, m_axis_mm2s_tvalid 为 1, 基于 FPGA 时序, 下一个时钟输出 s_axis_video_tuser 为 1 正好是一副图像的第一个像素。

s_axis_video_tuser: 因此 s_axis_video_tuser 代表了每一副图像开始的第一个像素。

4.8.3 关键信号 3

```
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt ==
```

VID_OUT_HS); //vid out last signal

m_axis_mm2s_tvalid: 是 M_AXIS_MM2S 接口(读 DMA 接口)的数据有效标志。

s_axis_video_tready: vid out IP 准备好了, 可以接收数据

vid_out_h_cnt == VID_OUT_HS): 图像一行数据的最后一个像素。

4.8.4 部分关键代码

表 3-6-4-1

<pre>reg [10:0] vid_out_v_cnt; reg [10:0] vid_out_h_cnt; reg [10:0] vid_in_v_cnt; parameter VID_OUT_HS = 11'd1279;//图像输出行分辨率 parameter VID_OUT_VS = 11'd719;//图像输出场分辨率 parameter VID_IN_VS = 11'd719; always@(posedge FCLK_CLK0) begin if(!gpio_rtl_tri_o_0) vid_out_v_cnt <= 11'd0; else if(m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == VID_OUT_HS))</pre>

```
if(vid_out_v_cnt != VID_OUT_VS)
    vid_out_v_cnt <= vid_out_v_cnt + 1'b1;
else
    vid_out_v_cnt <= 11'd0;
else
    vid_out_v_cnt <= vid_out_v_cnt;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_out_h_cnt <= 11'd0;
    else
        if(m_axis_mm2s_tvalid & s_axis_video_tready)
            if(vid_out_h_cnt != VID_OUT_HS)
                vid_out_h_cnt <= vid_out_h_cnt + 1'b1;
            else
                vid_out_h_cnt <= 11'd0;
        else
            vid_out_h_cnt <= vid_out_h_cnt;
    end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        vid_in_v_cnt <= 11'd0;
    else
        if(m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast)
            if(vid_in_v_cnt != VID_IN_VS)
                vid_in_v_cnt <= vid_in_v_cnt + 1'b1;
            else
                vid_in_v_cnt <= 11'd0;
        else
            vid_in_v_cnt <= vid_in_v_cnt;
    end

assign s_axis_video_tuser = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt == 11'd0) &
(vid_out_v_cnt == 11'd0); //vid out user
assign s_axis_video_tlast = m_axis_mm2s_tvalid & s_axis_video_tready & (vid_out_h_cnt ==
VID_OUT_HS);//vid out last signal

assign s_axis_s2mm_tlast = m_axis_video_tvalid & s_axis_s2mm_tready & m_axis_video_tlast
&(vid_in_v_cnt == VID_IN_VS);//dma in last signal
```

4.9 PS 部分

4.9.1 DMA 中断函数部分分析

为了让图像输出高品质效果，PS 部分设计了 3 缓存处理机制。3 缓存处理机制在大量图像缓冲处理方法是最有效的办法之一。

在 DMA_intr.h 文件中，定义 3 段内存空间用于保存三副最新的图像。

```
#define BUFFER0_BASE (MEM_BASE_ADDR)
#define BUFFER1_BASE (MEM_BASE_ADDR + IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
#define BUFFER2_BASE (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
```

在 DMA_intr.h 文件中，还定义一下 2 个变量 1 个指针数组。tx_buffer_index; 指示了当前的发送缓存序号，rx_buffer_index; 指示了当前的接收缓存序号。*BufferPtr[3] 会被制定到对应的内存地址空间。

```
extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;
extern u32 *BufferPtr[3];
```

在 main 函数里面有这么一段实现了指针数组指向内存地址空间。

```
BufferPtr[0] = (u32 *)BUFFER0_BASE;
BufferPtr[1] = (u32 *)BUFFER1_BASE;
BufferPtr[2] = (u32 *)BUFFER2_BASE;
```

下面给出 dma_intr.h 的完整代码

表 3-7-1-1 dma_intr.h

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 */

#ifndef DMA_INTR_H
#define DMA_INTR_H
#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
#include "xscugic.h"

/********************* Constant Definitions ********************/
/*
 * Device hardware build related constants.
*/
```

```
/*
#define DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID

#define MEM_BASE_ADDR      0x10000000

#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR

#define IMAGE_WIDTH      1280
#define IMAGE_HEIGHT     720
#define BYTES_PER_PIXEL   4
#define BUFFER_NUM       2

#define MEM_BASE_ADDR      0x10000000

#define BUFFER0_BASE      (MEM_BASE_ADDR )
#define BUFFER1_BASE      (MEM_BASE_ADDR +      IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)
#define BUFFER2_BASE      (MEM_BASE_ADDR + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *
BYTES_PER_PIXEL)

/* Timeout loop counter for reset
 */
#define RESET_TIMEOUT_COUNTER    10000
/* test start value
 */
#define TEST_START_VALUE    0xC
/*
 * Buffer and Buffer Descriptor related constant definition
 */
#define MAX_PKT_LEN        (IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)
/*
 * transfer times
 */
#define NUMBER_OF_TRANSFERS 100000

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

```

extern volatile u8 tx_buffer_index;
extern volatile u8 rx_buffer_index;

extern u32 *BufferPtr[3];

int DMA_CheckData(int Length, u8 StartValue);
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr);
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);
#endif

```

每次一幅图像通过 DMA 进入 DDR 后，会产生 DMA 中断请求，在 DMA 中断请求中，会指定下一次 DMA 接收的 buffer 位置。

表 3-7-1-2 DMA_RxIntrHandler 函数

```

/****************************************************************************
/*
*
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.
*
* @param    Callback is a pointer to RX channel of the DMA engine.
*
* @return   None.
*
* @note    None.
*
****************************************************************************
static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    u32 Status;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

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

    /* Acknowledge pending interrupts */
}

```

```

XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

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

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
 */
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
    xil_printf("rx error! \r\n");
    return;
}

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

if(rx_buffer_index == 2)
    rx_buffer_index = 0;
else
    rx_buffer_index++;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[rx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma failed! 0 %d\r\n", Status);
    return;
}

}

```

发送函数通过 tx_buffer_index 标记需要发送的缓存部分，并且确保发送的是最新保存的一副图像。

表 3-7-3 DMA_TxIntrHandler

```
*****
/*

```

```
/*
 * This is the DMA TX Interrupt handler function.
 *
 * It gets the interrupt status from the hardware, acknowledges it, and if any
 * error happens, it resets the hardware. Otherwise, if a completion interrupt
 * is present, then sets the TxDone.flag
 *
 * @param    Callback is a pointer to TX channel of the DMA engine.
 *
 * @return   None.
 *
 * @note    None.
 *
 *****/
static void DMA_TxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    u32 Status;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

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

    /* Acknowledge pending interrupts */

    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);

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

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

```

//Error = 1;
xil_printf("tx error! \r\n");
return;
}

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

    TxDone++;
}

if(rx_buffer_index == 0)
    tx_buffer_index = 2;
else
    tx_buffer_index = rx_buffer_index - 1;

Status = XAxiDma_SimpleTransfer(AxiDmaInst, (u32)BufferPtr[tx_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma failed! 0 %d\r\n", Status);
    return;
}
}
}

```

4.9.2 IIC 驱动

本章的 IIC 驱动在结构上与上一章基本一致，在此仅介绍一下改动的部分。

因为 5640 的寄存器是 16bit 的地址，8bit 的数据，因此需要对 I2C_write 函数进行一些改动：

```

int I2C_write_byte(XIicPs *InstancePtr, u8 sensor_slave_addr, void
*write_byte, int byte_num)

{
    int Status;

    Status = XIicPs_MasterSendPolled(InstancePtr, write_byte,
byte_num, sensor_slave_addr);

    if (Status != XST_SUCCESS) {
        xil_printf("IIC Send byte Failed!");
    }
}

```

```

/*
 * Wait until bus is idle to start another transfer.
 */
while (XIicPs_BusIsBusy(InstancePtr));

return XST_SUCCESS;
}

int I2C_write(XIicPs *InstancePtr,u16 addr,u8 data)
{
    u8 buf[3];
    buf[0] = (addr >> 8);
    buf[1] = addr;
    buf[2] = data;
    if(I2C_write_byte(InstancePtr,OV_CAM,buf,3) != XST_SUCCESS)
        xil_printf("IIC Write bytes Failed");
    return XST_SUCCESS;
}

```

相应的寄存器配置表中的内容也要进行修改:

```

struct config_table ov_5640_config_table[]={
    {0x3103,0x11}, // system clock from pad,0x bit[1]
    {0x3008,0x82}, // software reset,0x bit[7]// delay 5ms

    {0x3008,0x42}, // software power down,0x bit[6]
    //usleep(5000),
    {0x3103,0x03}, // system clock from PLL,0x bit[1]
    {0x3017,0xff}, // FREX,0x Vsync,0x HREF,0x PCLK,0x D[9:6] output enable
    {0x3018,0xff}, // D[5:0],0x GPIO[1:0] output enable
    {0x3034,0x1A}, // MIPI 10-bit
    {0x3037,0x13}, // PLL root divider,0x bit[4],0x PLL pre-divider,0x
bit[3:0]
    {0x3108,0x01}, // PCLK root divider,0x bit[5:4],0x SCLK2x root
divider,0x bit[3:2] // SCLK root divider,0x bit[1:0]
    {0x3630,0x36},
    {0x3631,0x0e},
    {0x3632,0xe2},
    {0x3633,0x12},
    {0x3621,0xe0},
    {0x3704,0xa0},
    {0x3703,0x5a},
    {0x3715,0x78},
    {0x3717,0x01},
    {0x370b,0x60},

```

```
{0x3705,0x1a},  
{0x3905,0x02},  
{0x3906,0x10},  
{0x3901,0x0a},  
{0x3731,0x12},  
{0x3600,0x08}, // VCM control  
{0x3601,0x33}, // VCM control  
{0x302d,0x60}, // system control  
{0x3620,0x52},  
{0x371b,0x20},  
{0x471c,0x50},  
{0x3a13,0x43}, // pre-gain = 1.047x  
{0x3a18,0x00}, // gain ceiling  
{0x3a19,0xf8}, // gain ceiling = 15.5x  
{0x3635,0x13},  
{0x3636,0x03},  
{0x3634,0x40},  
{0x3622,0x01}, // 50/60Hz detection      50/60Hz  
{0x3c01,0x34}, // Band auto, 0x bit[7]  
{0x3c04,0x28}, // threshold low sum  
{0x3c05,0x98}, // threshold high sum  
{0x3c06,0x00}, // light meter 1 threshold[15:8]  
{0x3c07,0x08}, // light meter 1 threshold[7:0]  
{0x3c08,0x00}, // light meter 2 threshold[15:8]  
{0x3c09,0x1c}, // light meter 2 threshold[7:0]  
{0x3c0a,0x9c}, // sample number[15:8]  
{0x3c0b,0x40}, // sample number[7:0]  
{0x3810,0x00}, // Timing Hoffset[11:8]  
{0x3811,0x10}, // Timing Hoffset[7:0]  
{0x3812,0x00}, // Timing Voffset[10:8]  
{0x3708,0x64},  
{0x4001,0x02}, // BLC start from line 2  
{0x4005,0x1a}, // BLC always update  
{0x3000,0x00}, // enable blocks  
{0x3004,0xff}, // enable clocks  
{0x300e,0x58}, // MIPI power down, 0x DVP enable  
{0x302e,0x00},  
{0x4300,0x60}, // RGB565  
{0x501f,0x01}, // ISP RGB  
{0x440e,0x00},  
{0x5000,0xa7}, // Lenc on, 0x raw gamma on, 0x BPC on, 0x WPC on, 0x CIP on  
// AEC target  
{0x3a0f,0x30}, // stable range in high  
{0x3a10,0x28}, // stable range in low
```

```
{0x3a1b,0x30}, // stable range out high
{0x3a1e,0x26}, // stable range out low
{0x3a11,0x60}, // fast zone high
{0x3a1f,0x14}, // fast zone low// Lens correction for
{0x5800,0x23},
{0x5801,0x14},
{0x5802,0x0f},
{0x5803,0x0f},
{0x5804,0x12},
{0x5805,0x26},
{0x5806,0x0c},
{0x5807,0x08},
{0x5808,0x05},
{0x5809,0x05},
{0x580a,0x08},
{0x580b,0x0d},
{0x580c,0x08},
{0x580d,0x03},
{0x580e,0x00},
{0x580f,0x00},
{0x5810,0x03},
{0x5811,0x09},
{0x5812,0x07},
{0x5813,0x03},
{0x5814,0x00},
{0x5815,0x01},
{0x5816,0x03},
{0x5817,0x08},
{0x5818,0x0d},
{0x5819,0x08},
{0x581a,0x05},
{0x581b,0x06},
{0x581c,0x08},
{0x581d,0x0e},
{0x581e,0x29},
{0x581f,0x17},
{0x5820,0x11},
{0x5821,0x11},
{0x5822,0x15},
{0x5823,0x28},
{0x5824,0x46},
{0x5825,0x26},
{0x5826,0x08},
{0x5827,0x26},
```

```
{0x5828,0x64},  
{0x5829,0x26},  
{0x582a,0x24},  
{0x582b,0x22},  
{0x582c,0x24},  
{0x582d,0x24},  
{0x582e,0x06},  
{0x582f,0x22},  
{0x5830,0x40},  
{0x5831,0x42},  
{0x5832,0x24},  
{0x5833,0x26},  
{0x5834,0x24},  
{0x5835,0x22},  
{0x5836,0x22},  
{0x5837,0x26},  
{0x5838,0x44},  
{0x5839,0x24},  
{0x583a,0x26},  
{0x583b,0x28},  
{0x583c,0x42},  
{0x583d,0xce}, // _enc BR offset // AWB  
{0x5180,0xff}, // AWB B block  
{0x5181,0x58}, // AWB control  
{0x5182,0x11}, // [7:4] max local counter, 0x [3:0] max fast counter  
{0x5183,0x90}, // AWB advanced  
{0x5184,0x25},  
{0x5185,0x24},  
{0x5186,0x09},  
{0x5187,0x09},  
{0x5188,0x09},  
{0x5189,0x75},  
{0x518a,0x54},  
{0x518b,0xe0},  
{0x518c,0xb2},  
{0x518d,0x42},  
{0x518e,0x3d},  
{0x518f,0x56},  
{0x5190,0x46},  
{0x5191,0xff}, // AWB top limit  
{0x5192,0x00}, // AWB bottom limit  
{0x5193,0xf0}, // red limit  
{0x5194,0xf0}, // green limit  
{0x5195,0xf0}, // blue limit
```

```
{0x5196,0x03}, // AWB control
{0x5197,0x02}, // local limit
{0x5198,0x04},
{0x5199,0x12},
{0x519a,0x04},
{0x519b,0x00},
{0x519c,0x06},
{0x519d,0x82},
{0x519e,0x00}, // AWB control // Gamma
{0x5480,0x01}, // Gamma bias plus on,0x bit[0]
{0x5481,0x08},
{0x5482,0x14},
{0x5483,0x28},
{0x5484,0x51},
{0x5485,0x65},
{0x5486,0x71},
{0x5487,0x7d},
{0x5488,0x87},
{0x5489,0x91},
{0x548a,0x9a},
{0x548b,0xaa},
{0x548c,0xb8},
{0x548d,0xcd},
{0x548e,0xdd},
{0x548f,0xea},
{0x5490,0x1d}, // color matrix
{0x5381,0x1e}, // CMX1 for Y
{0x5382,0x5b}, // CMX2 for Y
{0x5383,0x08}, // CMX3 for Y
{0x5384,0x0a}, // CMX4 for U
{0x5385,0x7e}, // CMX5 for U
{0x5386,0x88}, // CMX6 for U
{0x5387,0x7c}, // CMX7 for V
{0x5388,0x6c}, // CMX8 for V
{0x5389,0x10}, // CMX9 for V
{0x538a,0x01}, // sign[9]
{0x538b,0x98}, // sign[8:1] // UV adjust UV
{0x5580,0x06}, // saturation on,0x bit[1]
{0x5583,0x40},
{0x5584,0x10},
{0x5589,0x10},
{0x558a,0x00},
{0x558b,0xf8},
{0x501d,0x40}, // enable manual offset of contrast// CIP
```

```
{0x5300,0x08}, // CIP sharpen MT threshold 1
{0x5301,0x30}, // CIP sharpen MT threshold 2
{0x5302,0x10}, // CIP sharpen MT offset 1
{0x5303,0x00}, // CIP sharpen MT offset 2
{0x5304,0x08}, // CIP DNS threshold 1
{0x5305,0x30}, // CIP DNS threshold 2
{0x5306,0x08}, // CIP DNS offset 1
{0x5307,0x16}, // CIP DNS offset 2
{0x5309,0x08}, // CIP sharpen TH threshold 1
{0x530a,0x30}, // CIP sharpen TH threshold 2
{0x530b,0x04}, // CIP sharpen TH offset 1
{0x530c,0x06}, // CIP sharpen TH offset 2
{0x5025,0x00},
{0x3008,0x02}, // wake up from standby,0x bit[6]

{0x3035, 0x41}, // PLL
{0x3036, 0x69}, // PLL
{0x3c07, 0x07}, // lightmeter 1 threshold[7:0]
{0x3820, 0x40}, // flip
{0x3821, 0x07}, // mirror
{0x3814, 0x31}, // timing X inc
{0x3815, 0x31}, // timing Y inc
{0x3800, 0x00}, // HS
{0x3801, 0x00}, // HS
{0x3802, 0x00}, // VS
{0x3803, 0xfa}, // VS
{0x3804, 0x0a}, // HW (HE)
{0x3805, 0x3f}, // HW (HE)
{0x3806, 0x06}, // VH (VE)
{0x3807, 0xa9}, // VH (VE)
{0x3808, 0x05}, // DVPHO
{0x3809, 0x00}, // DVPHO
{0x380a, 0x02}, // DVPVO
{0x380b, 0xd0}, // DVPVO
{0x380c, 0x07}, // HTS
{0x380d, 0x64}, // HTS
{0x380e, 0x02}, // VTS
{0x380f, 0xe4}, // VTS
{0x3813, 0x04}, // timing V offset
{0x3618, 0x00},
{0x3612, 0x29},
{0x3709, 0x52},
{0x370c, 0x03},
{0x3a02, 0x02}, // 60Hz max exposure
```

```

{0x3a03, 0xe0}, // 60Hz max exposure
{0x3a14, 0x02}, // 50Hz max exposure
{0x3a15, 0xe0}, // 50Hz max exposure
{0x4004, 0x02}, // BLC line number
{0x3002, 0x1c}, // reset JFIFO, SFIFO, JPG
{0x3006, 0xc3}, // disable clock of JPEG2x, JPEG
{0x4713, 0x03}, // JPEG mode 3
{0x4407, 0x04}, // Quantization scale
{0x460b, 0x37},
{0x460c, 0x20},
{0x4837, 0x16}, // MIPI global timing
{0x3824, 0x04}, // PCLK manual divider
{0x5001, 0x83}, // SDE on, CMX on, AWB on
{0x3503, 0x00}, // AEC/AGC on
{0x3b00, 0x83}, // STROBE CTRL: strobe request ON, 0x Strobe
mode: LED3
{0x3b00, 0x00}, // STROBE CTRL: strobe request OFF
{Config_done, 0x00}
};

```

4.9.3 main.c 文件

这个主程序比较简单，内容比上一个课程的精简很多，这里需要注意的地方是 XGpio_DiscreteWrite(&Gpio, 1, 1); 函数这个函数是这只摄像头和 DMA 之间数据同步的，没有这个同步图像容易错位。另外在主函数里面首先启动 DMA 接收和发送中断各一次，以后就可以在中断里面继续触发了。

表 3-7-2-1 main.c

```

/*
*
* www.osrc.cn
* www.milinker.com
* copyright by nan jin mi lian dian zi www.osrc.cn
* axi dma test
*
*/
#include "dma_intr.h"
#include "sys_intr.h"
#include "xgpio.h"

volatile int TxDone;
volatile int RxDone;

```

```
volatile int Error;

volatile u8 tx_buffer_index;
volatile u8 rx_buffer_index;

u32 *BufferPtr[3];

static XScuGic Intc; //GIC
static XAxiDma AxiDma;
static XGpio Gpio;

#define AXI_GPIO_DEV_ID           XPAR_AXI_GPIO_0_DEVICE_ID

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

int main(void)
{
    u32 Status:
    BufferPtr[0] = (u32 *)BUFFER0_BASE;
    BufferPtr[1] = (u32 *)BUFFER1_BASE;
    BufferPtr[2] = (u32 *)BUFFER2_BASE;

    tx_buffer_index = 0;
    rx_buffer_index = 0;
    TxDone = 0;
    RxDone = 0;
    Error = 0;

    XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);
    XGpio_SetDataDirection(&Gpio, 1, 0);
    init_intr_sys();

Miz702_EMIO_init();
```

```
Ov5640_init_rgb();
```

```
XGpio_DiscreteWrite(&Gpio, 1, 1);  
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[rx_buffer_index],  
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
```

```
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)BufferPtr[tx_buffer_index],  
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
```

```
while (1);  
    return XST_SUCCESS;  
}
```

4.10 实验效果



CH05_AXI_VDMA_OV7725 摄像头采集系统

本课程将对 Xilinx 提供的一款 IP 核——AXI VDMA (Video Direct Memory Access) 进行详细讲解，为后续的学习和开发做好准备。内容安排如下：首先分析为什么要使用 VDMA、VDMA 的作用；然后详细介绍 VDMA 的特点、寄存器空间；最后阐述如何使用 VDMA，包括 IP 核的配置方法、代码编写流程等。

本章主要是理论学习，学习完本章，会对 VDMA 有全面的认识，有利于学习后续的图像生成、视频采集处理系统。由于 VDMA 主要用于视频流数据的存取，单独测试的意义不大，所以在接下来的章节会提供一些样例设计，进一步学习如何使用 VDMA。

5.1 为什么要用 VDMA

在讲解 VDMA 之前，先来探讨一下为什么要学习和使用 VDMA，以明确学习目的。由于使用 VDMA 可以方便地实现双缓冲和多缓冲机制，所以本小节引入了帧缓存和缓冲机制的概念。另外，VDMA 可以很好地契合 Zynq 内部架构，缩短开发周期。再加上 VDMA 本身能够高效地实现数据存取，所以在基于 Zynq（也包括其他 Xilinx FPGA）图像、视频处理系统中，VDMA 可谓是必不可少的。

5.1.1 什么是帧缓存

帧缓冲存储器(Frame Buffer)：简称帧缓存或显存，它是屏幕所显示画面的一个直接映象，又称为位映射图(Bit Map)或光栅。帧缓存的每一存储单元对应屏幕上的一个像素，整个帧缓存对应一帧图像。

在开发者看来，FrameBuffer 是一块显示缓存，往显示缓存中写入特定格式的数据就意味着向屏幕输出内容。所以说 FrameBuffer 就是一块画布，系统在画布上绘制好画面之后，就可以通知显示设备读取 Frame Buffer 进行显示了。

注意，笔者这里所说的 Frame Buffer 和 Linux 的 Frame Buffer 不是同一个概念，这里仅指显示缓存（画布）本身，并不是 Linux 下的一个设备。

5.1.2 双缓冲机制

最早解释多缓冲区如何工作的方式，是通过一个现实生活中的实例来解释的。在一个阳光明媚的日子，你想将水池里的水打满，而又找不到水管的时候，就只能用手边的木桶来灌满水池。水桶满了之后，关掉水龙头，将水提到水池旁边，倒进去，然后走回到水龙头。重复上述工作，如此往复直到将水池灌满。这就类似单缓冲工作过程，当你想将木桶里的水倒出的时候，你必须关掉水龙头。

现在假设你用两个木桶来做上面的工作。你会注满第一个木桶然后将第二个木桶换到水龙头下面，这样，在第二个木桶注满的时间内，你就可以将第一个木桶里面的水倒进水池里面，当你回来的时候，你只需要再将第一个木桶换下第二个注满水木桶，当第一个木桶开始注水的时候你就将第二个木桶里面的水倒进水池里面。重复这个过程直到水池被注满。很容易看得到用这种技术注满水池将会更快，同时也节省了很多等待木桶被注满的时间，而这段时间里你什么也做不了，而水龙头也就不用等待从木桶被注满到你回来的这段时间了。

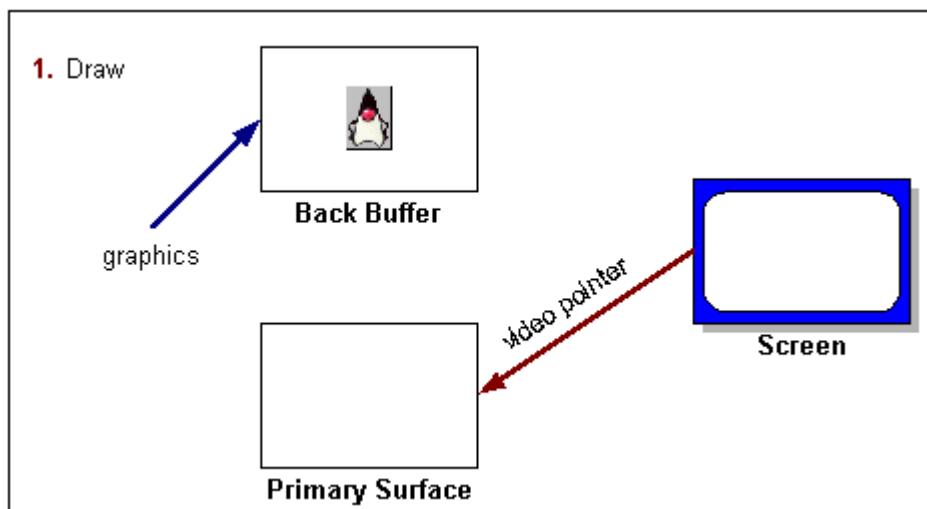
当你雇佣另外一个人来搬运一个被注满的木桶时，这就有点类似于三个缓冲区的工

作原理。如果将搬运木桶的时间很长，你可以用更多的木桶，雇佣更多的人，这样水龙头就会一直开着注满木桶了。

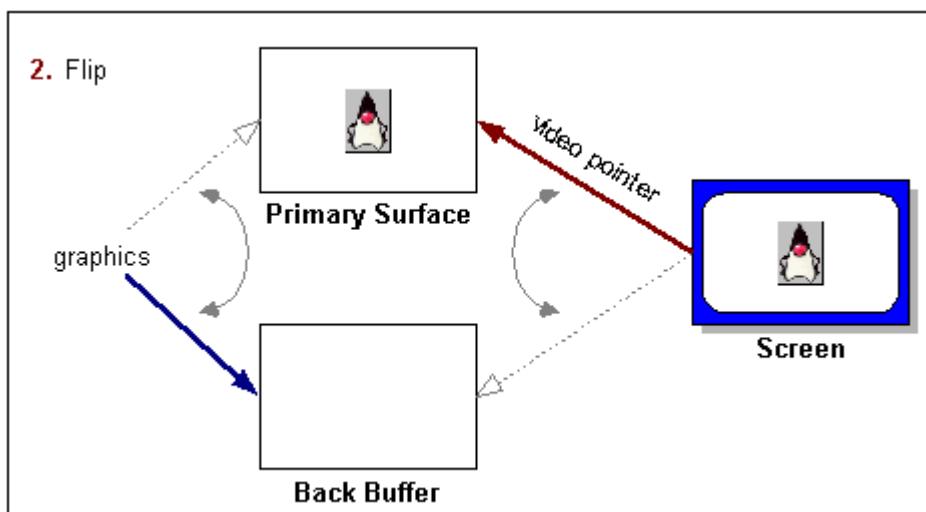
在计算机图形学中，双缓冲是一种画图技术，使用这种技术可以使得画图没有（至少是减少）闪烁、撕裂等不良效果，并减少等待时间。

双缓冲机制的原理大概是：所有画图操作将它们画图的结果保存在一块系统内存区域中，这块区域通常被称作“后缓冲区（back buffer）”，当所有的绘图操作结束之后，将整块区域复制到显示内存中，这个复制操作通常要跟显示器的光栈束同步，以避免撕裂。双缓冲机制必须要求有比单缓冲更多的显示内存和CPU消耗时间，因为“后缓冲区”需要显示内存，而复制操作和等待同步需要CPU时间。

基于双缓冲机制可以实现页交换，页交换初始状态如下图所示：



如上图所示，此时由于处于初始状态，画图操作的结果都在后缓冲区中，而屏幕上显示的则是前缓冲区中的内容。此时画图操作尚未完成，画图操作完成之后，页转换操作开始执行，示意图如下图所示：



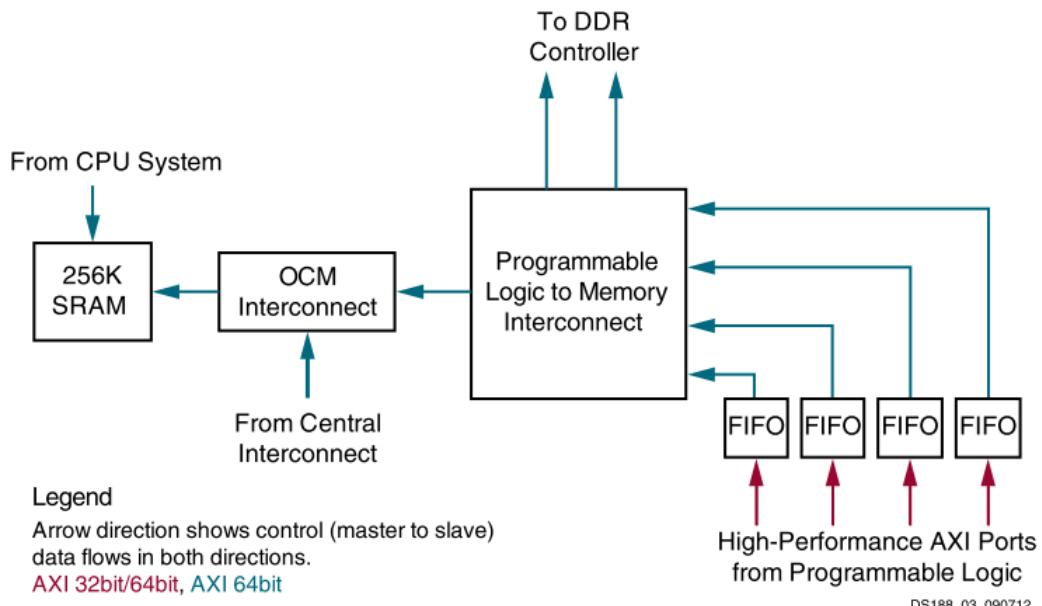
如上图所示，画图操作结束，下一个画图操作的结果保存对象指向前缓冲区，屏幕的显示对象指向后缓冲区，此时前缓冲区变成实际意义上的后缓冲区，后缓冲区变成实际意义上的前缓冲去，即实现“页交换”操作。

有时候也在页交换链中设置多个“后缓冲区”，这是就需要多缓冲区机制的支持。

5.1.3 Zynq 硬件架构

在 Zynq 芯片内部，PS 和 PL 是共享 DDR 控制器的。PS 访问 DDR 十分简单，只要操作 DDR 映射的虚拟地址即可。对于 PL 而言，要接入 DDR，必须通过 AXI_HP 端口。

Zynq 共有四个 AXI_HP 通道，通道数据宽度可以配置为 32 位或 64 位，这些接口通过 FIFO 控制器连接 PL 到存储接口上，其中有两条连接到 DDR 存储控制器上，还有一条是连接到双端口的 OCM 上的，下图是 AXI_HP 访问 DDR 和 OCM 的连接图。



由上图可以看出，AXI_HP 接口也是遵循 AXI 协议的，因此利用 VDMA 可以直接连接 HP 端口。除了使用 VDMA，当然也可以自己开发出符合 AXI 协议的 IP，但是综合考虑设计成本，没太有必要自己实现。此外，自己实现的 IP 功能也不见得比 VDMA 强大。

5.1.4 VDMA 的作用

VDMA 数据接口可以分为读、写通道，用户可以通过写通道将 AXI-Stream 类型的数据流写入 DDR3，通过读通道可以从 DDR3 读取数据，并以 AXI-Stream 类型的格式输出。由此可知，VDMA 本质上是一个数据搬运 IP，为数据进、出 DDR3 提供了一种便捷的方案。

将数据存入 DDR 之后，CPU 就可以进行一些处理（缩放、裁剪等），然后再送至显示设备，达到期望的应用目的。当然，也可能是简单地对捕获的视频进行解析，将数据存入帧缓存，以供显示。

VDMA 可以控制多达 32 个帧存，并可以自由地进行帧存切换，所以就能够轻松地实现双缓冲和多缓冲操作。这也是一个很重要的特性，在后续进行系统设计的时候，通常是采用多缓冲的方式实现显示。

由以上分析可以发现，在基于 Zynq 的图像、视频处理系统中使用 VDMA 是十分有必要的。

5.2 VDMA 概述

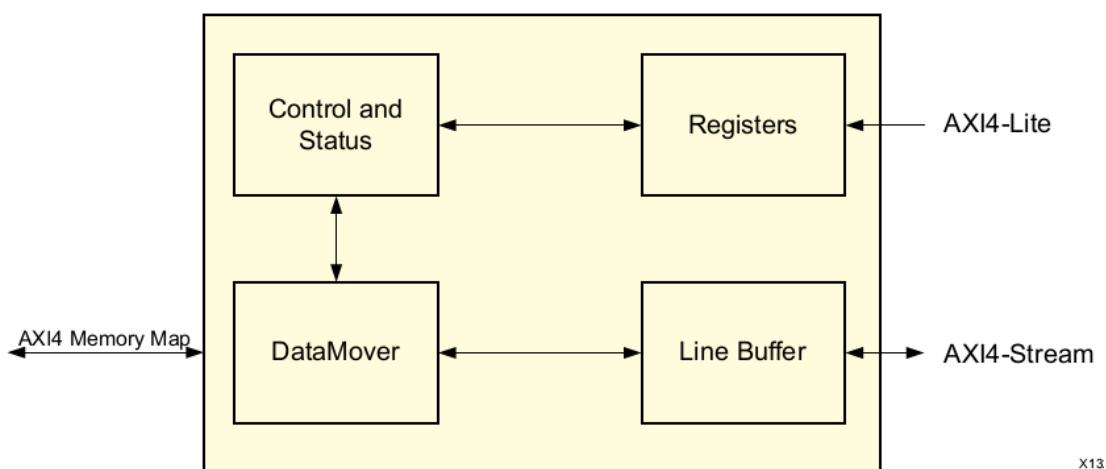
AXI VDMA 是 Xilinx 提供的软核 IP，用于将 AXI Stream 格式的数据流转换为

Memory Map 格式或将 Memory Map 格式的数据转换为 AXI Stream 数据流，从而实现与 DDR3 进行通信。

许多视频类应用都需要帧缓存来处理帧率变化或者进行图像的缩放、裁剪等尺寸变换操作。AXI VDMA 设计的初衷就是用于高效地实现 AXI4-Stream 视频流接口和 AXI4 接口之间的数据传输。

VDMA 的关键特性&优势有以下几点：

- 使视频流能够高带宽直接接入内存
 - 高效的二维 DMA 操作
 - 独立的异步读写通道操作
 - Gen-Lock 帧存同步机制
 - 最多支持 32 个帧存
 - 支持视频格式动态切换
 - 猥发长度和行缓存深度可调节
 - 处理器可以控制 IP 的初始化、状态、中断和管理寄存器
 - 基础 AXI 流数据位宽为 8 的整数倍，如 8,16,24, 32 等，最大可达 1024 个位
- AXI VDMA 框图如下所示。



主要有以下几种接口类型：

- AXI-lite： PS 通过该接口来配置 VDMA
- AXI Memory Map write： 映射到存储器写
- AXI Memory Map read： 映射到存储器读
- AXI Stream Write(S2MM)： AXI Stream 视频流写入图像
- AXI Stream Read(MM2S)： AXI Stream 视频流读出图像

从框图中可以看出，VDMA 主要由控制和状态寄存器、数据搬运模块、行缓冲这几部分构成。数据进出 DDR 要经过行缓冲进行缓存，然后由数据搬运模块写入或者读出数据。数据搬运模块具体如何工作，由相关寄存器负责控制。VDMA 的工作状态可以通过读取状态寄存器进行获取。

5.3 VDMA 详细介绍

5.3.1 接口

5.3.1.1 时钟和复位

各种总线都有自己的时钟信号，不用特别说明，需要指出的是，这些时钟是异步的，并不需要用同一个时钟。但在设计过程中，如无特别需求，可以使用相同的时钟，以降低设计难度。

同步复位信号 axi_resetn，同步时钟为 s_axi_lite_aclk，低电平有效（至少要保持 16 个时钟周期的低电平，才能够生效），有效时复位整个 IP 核。

5.3.1.2 AXI 总线相关信号

- AXI4-Lite 接口 (S_AXI_LITE)
- AXI4 读接口 (M_AXI_MM2S)
- AXI4 写接口 (M_AXI_S2MM)
- AXI4-Stream 主接口 (M_AXI_MM2S)
- AXI4-Stream 从接口 (S_AXI_S2MM)

前缀 S_、M_ 分别表示 Slave 和 Master；后缀 MM2S、S2MM 说明数据流向是从 memory map 到 stream 还是从 stream 到 memory map。具体每个接口所包含的信号，在基础篇第 20 章已有介绍，此处不再重复。

5.3.1.3 视频同步接口信号

信号名称	方向	详细描述
mm2s_fsync	Frame Sync	MM2S 帧同步输入。使能该信号后，VDMA 操作开始于 fsync 每个下降沿。该信号至少要持续一个 m_axis_mm2s_aclk 时钟周期
s2mm_fsync	Frame Sync	S2MM 帧同步输入。使能该信号后，VDMA 操作开始于 fsync 每个下降沿。该信号至少要持续一个 s_axis_s2mm_aclk 时钟周期

5.3.1.4 GenLock 相关信号

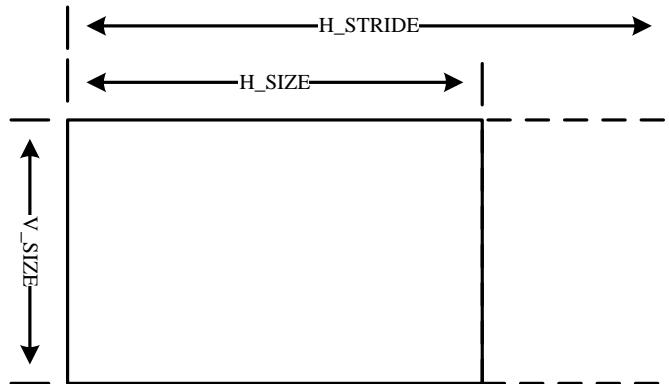
在下一节将详细介绍这些信号的作用和应用场合。

信号名称	方向	详细描述
mm2s_frame_ptr_in(5:0)	输入	输入的帧编号
mm2s_frame_ptr_out(5:0)	输出	输出当前帧的编号
s2mm_frame_ptr_in(5:0)	输入	输入的帧编号

s2mm_frame_ptr_out(5:0)	输出	输出当前帧的编号
-------------------------	----	----------

5.3.2 VDMA 帧存格式

在讲述寄存器时，需要设定和显示（帧存）相关的参数，为了方便读者的理解，这里先介绍 VDMA 数据存放框架，如下图所示，黑色实线内的区域为实际存储画面的帧存。



图中 H_STRIDE 代表水平方向上的跨度，H_SIZE 表示水平方向数据总量，V_SIZE 表示竖直方向总共有多少行。

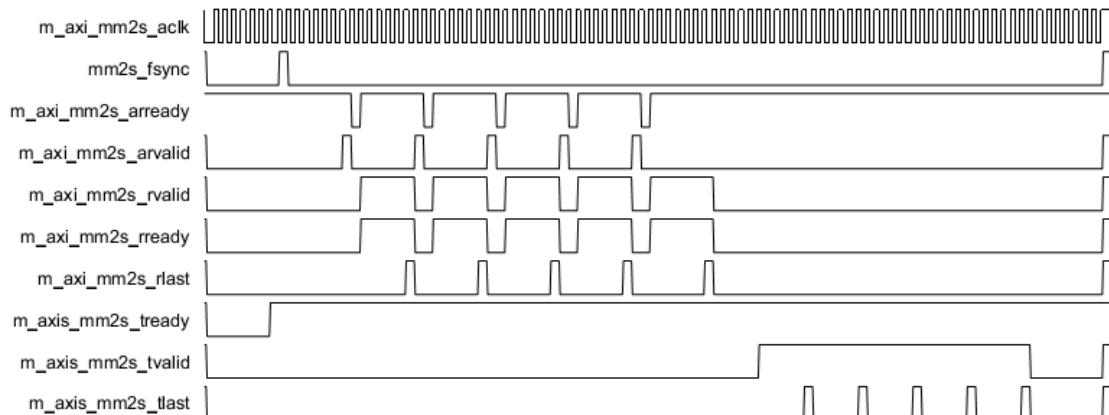
至于帧存内部数据如何组织，就取决于软件代码和硬件逻辑如何匹配了，通常来讲，数据存放格式为 RGB+Alpha 或者 Alpha+RGB。

5.3.3 读写通道工作时序

清晰地理解 VDMA 读写通道的工作时序，对以后的设计有很大的帮助，很多设计都是根据本小节所示的样例时序设计出来的。在下一章，读者就能够有所体会。

5.3.3.1 读通道（MM2S）时序

下图描述了读通道的时序，5 行，每行 16 字节，跨度为 32 字节。

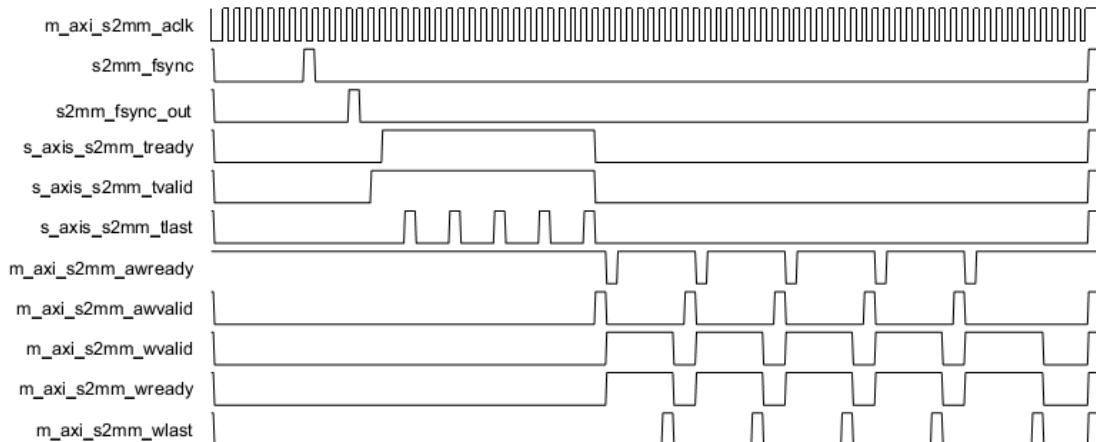


从图中可以看出：在收到 mm2s_fsync 信号后，VDMA 在 m_axi_mm2s_araddr 的起始地址处发出 m_axi_mm2s_arvalid 信号。M_axi_mm2s_arvalid 总共有效 5 次，分别获取一帧的 5 行数据。从 MM 读取的数据存储在行缓存里，当收到来自 axi-stream 端的 m_axis_mm2s_tvalid 信号后，将数据发送到 axi-stream 端。每一行的结束，axi-stream

端会使 `m_axis_mm2s_tlast` 有效。

5.3.3.2 写通道(S2MM)时序

下图描述了写通道的时序，5行，每行16字节，跨度为32字节。



从图中可以看出：在收到 `s2mm_fsync` 信号后，VDMA 发出 `s2mm_fsync_out` 和 `s_axis_s2mm_tready` 表明已经准备好接收来自 axi-stream 端的数据。读取到的数据存储在行缓存里，`m_axi_s2mm_awvalid` 有效后，紧接着有效 `m_axi_s2mm_wvalid` 信号，同时将数据放至 `m_axi_s2mm_wdata`。

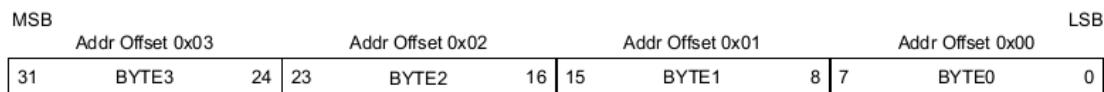
5.3.4 寄存器

VDMA 的寄存器如下表所示。所有寄存器都被映射到非缓存内存空间。该内存空间必须按照 AXI 字(32位)进行对齐，换句话说，寄存器偏移地址至少间隔4个字节。

寄存器名称	偏移地址	详细描述
MM2S_VDMACR	00h	MM2S VDMA 控制寄存器
MM2S_VDMASR	04h	MM2S VDMA 状态寄存器
保留	08h~10h	N/A
MM2S_REG_INDEX	14h	MM2S 寄存器索引
保留	18h~24h	N/A
PARK_PRT_REG	28h	MM2S 和 S2MM Park 指针寄存器
VDMA_VERSION	2Ch	VDMA 版本寄存器
S2MM_VDMACR	30h	S2MM VDMA 控制寄存器
S2MM_VDMASR	34h	S2MM VDMA 状态寄存器
保留	38h	N/A
S2MM_VDMA_IRQ_MASK	3Ch	S2MM 错误中断掩码寄存器
保留	40h	N/A
S2MM_REG_INDEX	44h	S2MM 寄存器索引
保留	48h~4Ch	N/A
MM2S_VSIZE	50h	MM2S 垂直方向显示大小寄存器
MM2S_HSIZE	54h	MM2S 水平方向显示大小寄存器

MM2S_FRMDLY_STRIDE	58h	MM2S 帧延迟和跨度寄存器
MM2S_START_ADDRESS(1~16)	5Ch~98h	MM2S 帧存起始地址 (1~16)
保留	9Ch	N/A
S2MM_VSIZE	A0h	S2MM 垂直方向显示大小寄存器
S2MM_HSIZE	A4h	S2MM 水平方向显示大小寄存器
S2MM_FRMDLY_STRIDE	A8h	S2MM 帧延迟和跨度寄存器
S2MM_START_ADDRESS(1~16)	ACh~E8h	S2MM 帧存起始地址 (1~16)

所有寄存器字节序都是小端格式，如下图所示。

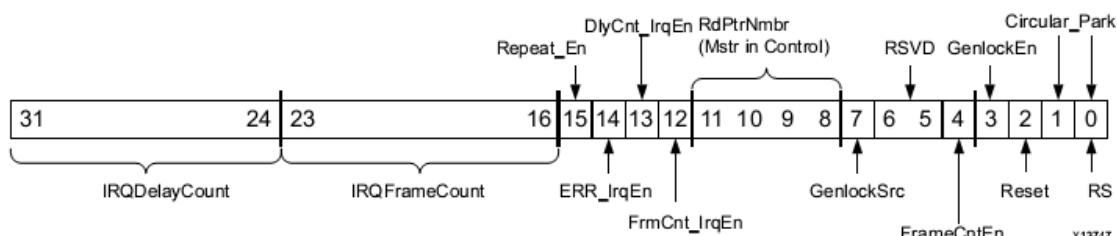


各个寄存器的名称和大致作用从上表就可以看出，接下来，笔者会详细介绍重要寄存器的具体 bit 的作用。明白了每个 bit 的作用之后，自然就知道写入什么值能够达到自己的控制目的。

从上表可以看出，寄存器可以分为两组，分别对应 MM2S 通道和 S2MM 通道，两组寄存器的功能是相似的，区别仅在于偏移地址和所服务的对象。因此，在学习完 MM2S 通道的所有寄存器之后，只要大致浏览一下 S2MM 通道对应的寄存器的关键位即可（个别位不相同），在使用高级功能时，再仔细查阅 VDMA 用户手册。

5.3.2.1 MM2S VDMA 控制寄存器 (00h)

顾名思义，该寄存器用于控制 VDMA，具体可以实现复位、使能锁相同步、设定帧存切换模式、启动 VDMA 读写通道等操作。每一位作用如下图所示，低 4 位是最重要的，接下来会详细介绍。

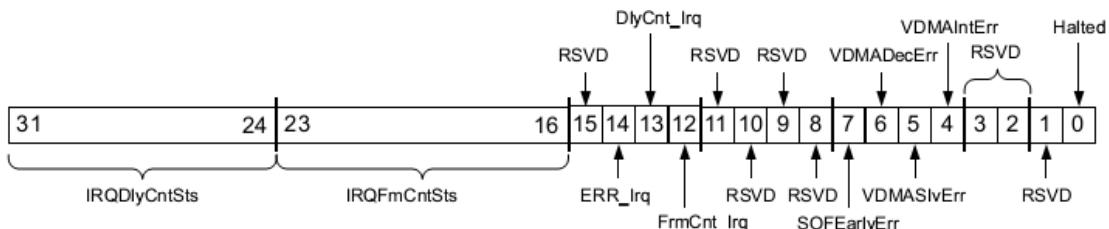


位	名称	默认值	接入类型	描述
31~4				非常用位，请参考 VDMA 使用手册自学
3	GenlockEn	0h	可读可写	使能锁相同步或者动态锁相同步模式。 0: 关闭 Genlock 或动态 Genlock 同步 1: 开启 Genlock 或动态 Genlock 同步 注：该位仅在通道被配置成锁相同步从接口或者动态锁相主、从接口时才起作用。配置成锁相同步主接口时，该位为保留位，值恒为 0。
2	Reset	0h	可读可写	0: 正常操作；1: 复位 MM2S 通道
1	Circular_Park	1h	可读可写	指定帧存为循环模式还是停留模式 0: 停留模式 - 显示用缓存页将停留在

				PARK_PTR_REG.RdFrmPntrRef 指定的帧存； 1：循环模式-循环切换显示用缓存页
0	RS	0h	可读可写	运行/停止，控制 VDMA 通道的运行和停止。 开始任何 VDMA 操作前，该位必须置 1. 0：停止；1：运行。

5.3.2.2 MM2S VDMA 状态寄存器 (04h)

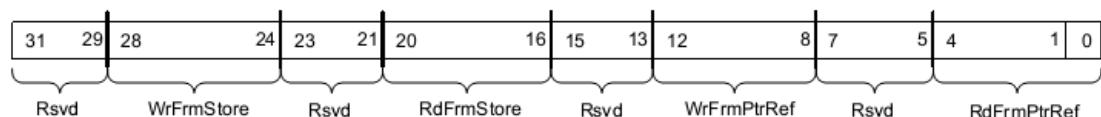
该寄存器用于获取 VDMA 工作状态。每一位作用如下图所示，低 4 位是最重要的，接下来会详细介绍。



位	名称	默认值	接入类型	描述
31~1				非常用位，请参考 VDMA 使用手册自学
0	Halted	1h	只读	指示 VDMA 运行是否停止。 0：运行；1：停止。

5.3.2.3 PARK_PTR_REG 停留指针寄存器 (28h)

该寄存器用于管理读、写通道的数据传输。



位	名称	默认值	接入类型	描述
31~29	保留	0h	只读	
28~24	WrFrmStore	0h	只读	用于存储写通道正在操作的帧的编号。指示 S2MM 通道正在操作的帧。
23~21	保留	0h	只读	
20~16	RdFrmStore	0h	只读	用于存储读通道正在操作的帧的编号。指示 MM2S 通道正在操作的帧。
15~13	保留	0h	只读	
12~8	WrFrmPtrRef	0h	可读可写	通过帧编号指定写通道操作的帧。当工作在停留模式，S2MM 通道操作对象停留在 WrFrmPtrRef 指定的帧。
7~5	保留	0h	只读	

4~0	RdFrmPtrRef	0h	可读可写	通过帧编号指定读通道操作的帧。当工作在停留模式，MM2S 通道操作对象停留在 RdFrmPtrRef 指定的帧。
-----	-------------	----	------	--

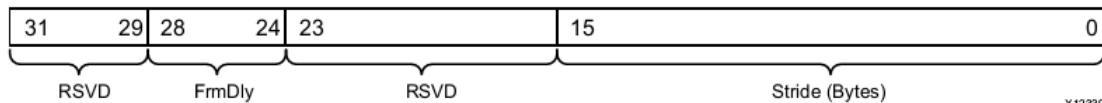
学习了这个寄存器之后，就可以发现：当 VDMA 工作在 Parked 模式下，通过操作该寄存器，就能够实现帧缓存的切换，建立自己想要的缓存切换机制。

5.3.2.4 MM2S 帧存起始地址 (0x5C~0x98)

有最多 32 个寄存器用于存放帧存起始地址，其分别存在于两个寄存器 bank 上：bank0 和 bank1，每个 bank 上有 16 个寄存器。这两个 bank 上有相同的起始偏移地址（0x5C），选择这两个 bank 可以通过 MM2S_REG_INDEX 的值进行选择。假如想访问第 1 个寄存器，则给 MM2S_REG_INDEX 赋值为 0，并设定偏移地址为 0x5C；如果想访问第 17 个寄存器，需要将 MM2S_REG_INDEX 设为 1，并设定初始偏移地址为 0x5C。

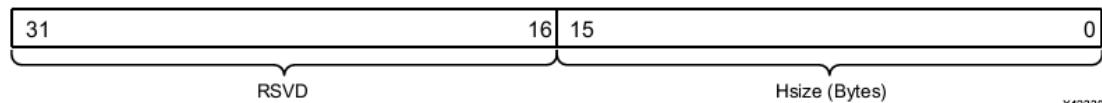
5.3.2.5 MM2S_FRMDLY_STRIDE MM2S 帧延迟和跨度 (58h)

该寄存器有两个作用，第一是 bit24~bit28 指定帧延迟，仅用于 Genlock 从模式，指定从接口比主接口至少要延迟多少个帧；第二是低 16 位指定水平方向的跨度，同样以字节为单位。所谓跨度是指每两行第一个像素之间间隔的数据个数，具体请参考 22.3.2 小节，VDMA 帧存格式。



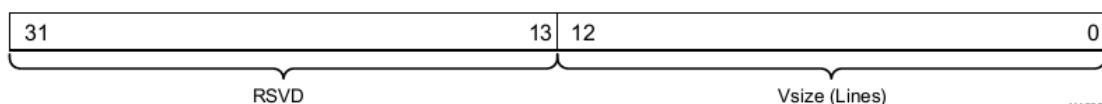
5.3.2.6 MM2S_HSIZE MM2S 水平方向尺寸 (54h)

该寄存器的低 16 位用于指定每一行有多少字节的数据需要传输。例如显示分辨率为 640*480，每个像素 4 个字节（RGB+Alpha），该值应该设定为 640*4。



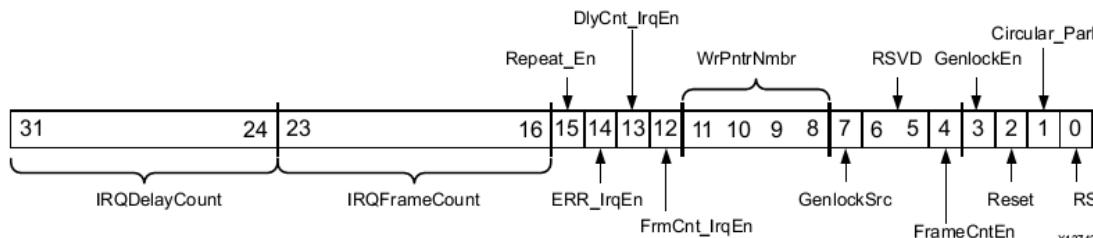
5.3.2.7 MM2S_VSIZE MM2S 垂直方向尺寸 (50h)

该寄存器有两个作用，第一是用低 13 位指定总共有多少行；第二是启动 MM2S 的传输。当 MM2S_VDMACR.RS=1，对该寄存器的写操作会将所有设定参数传递给 VDMA 内部寄存器模块，用于 VDMA 控制。**对某个通道进行配置时，必须在最后一步设置该寄存器。**



5.3.2.8 S2MM VDMA 控制寄存器 (30h)

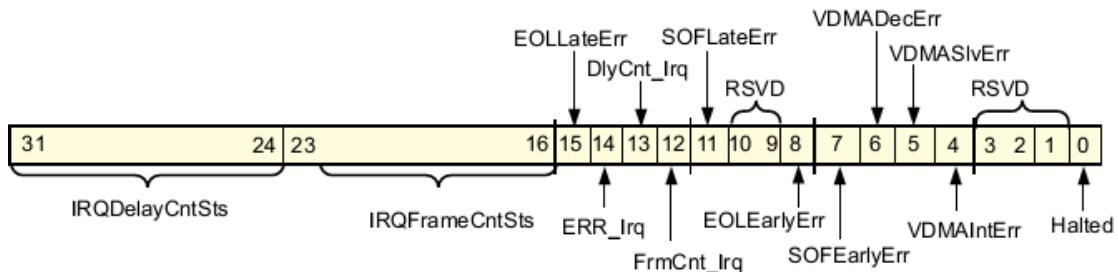
顾名思义，该寄存器用于控制 VDMA S2MM 通道，具体可以实现复位、使能锁相同步、设定帧存切换模式、启动 VDMA 读写通道等操作。每一位作用如下图所示，低 4 位是最重要的，接下来会详细介绍。



位	名称	默认值	接入类型	描述
31~4				非常用位，请参考 VDMA 使用手册自学
3	GenlockEn	0h	可读可写	使能锁相同步或者动态锁相同步模式。 0: 关闭 Genlock 或动态 Genlock 同步 1: 开启 Genlock 或动态 Genlock 同步 注：该位仅在通道被配置成锁相同步从接口或者动态锁相主、从接口时才起作用。配置成锁相同步主接口时，该位为保留位，值恒为 0。
2	Reset	0h	可读可写	0: 正常操作；1: 复位 S2MM 通道
1	Circular_Park	1h	可读可写	指定帧存为循环模式还是停留模式 0: 停留模式 - 显示用缓存页将停留在 PARK_PTR_REG.RdFrmPntrRef 指定的帧存； 1: 循环模式-循环切换显示用缓存页
0	RS	0h	可读可写	运行/停止，控制 VDMA 通道的运行和停止。 开始任何 VDMA 操作前，该位必须置 1. 0: 停止；1: 运行。

5.3.2.9 S2MM VDMA 状态寄存器 (34h)

该寄存器用于获取 S2MM 工作状态。每一位作用如下图所示，低 4 位是最重要的，接下来会详细介绍。



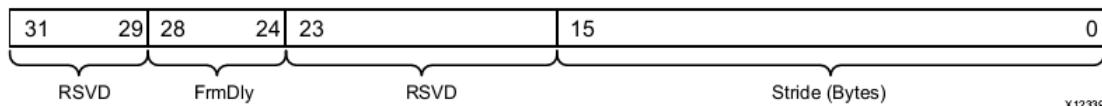
位	名称	默认值	接入类型	描述
31~1				非常用位, 请参考 VDMA 使用手册自学
0	Halted	1h	只读	指示 VDMA 运行是否停止。 0: 运行; 1: 停止。

5.3.2.4 S2MM 帧存起始地址 (0xAC~0xE8)

有最多 32 个寄存器用于存放帧存起始地址, 其分别存在于两个寄存器 bank 上: bank0 和 bank1, 每个 bank 上有 16 个寄存器。这两个 bank 上有相同的起始偏移地址 (0x5C), 选择这两个 bank 可以通过 S2MM_REG_INDEX 的值进行选择。假如想访问第 1 个寄存器, 则给 S2MM_REG_INDEX 赋值为 0, 并设定偏移地址为 0x5C; 如果想访问第 17 个寄存器, 需要将 MM2S_REG_INDEX 设为 1, 并设定初始偏移地址为 0x5C。

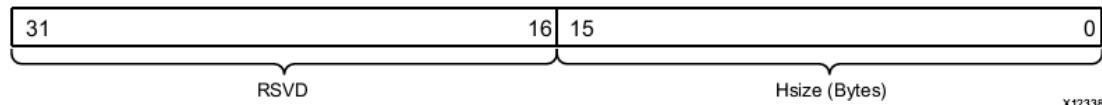
5.3.2.5 S2MM_FRMDLY_STRIDE S2MM 帧延迟和跨度 (A8h)

该寄存器有两个作用, 第一是 bit24~bit28 指定帧延迟, 仅用于 Genlock 从模式, 指定从接口比主接口至少要延迟多少个帧; 第二是低 16 位指定水平方向的跨度, 同样以字节为单位。所谓跨度是指每两行第一个像素之间间隔的数据个数, 具体请参考 22.3.2 小节, VDMA 帧存格式。



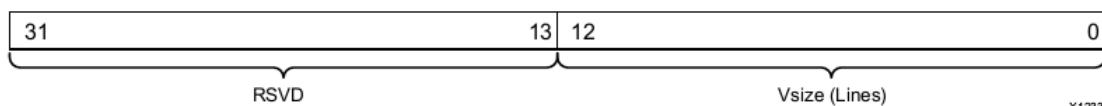
5.3.2.6 S2MM_HSIZE S2MM 水平方向尺寸 (A4h)

该寄存器的低 16 位用于指定每一行有多少字节的数据需要传输。例如显示分辨率为 640*480, 每个像素 4 个字节 (RGB+Alpha), 该值应该设定为 640*4。



5.3.2.7 S2MM_VSIZE S2MM 垂直方向尺寸 (A0h)

该寄存器有两个作用, 第一是用低 13 位指定总共有多少行; 第二是启动 S2MM 的传输。当 S2MM_VDMACR.RS=1, 对该寄存器的写操作会将所有设定参数传递给 VDMA 内部寄存器模块, 用于 VDMA 控制。对某个通道进行配置时, 必须在最后一步设置该寄存器。



5.3.5 帧同步选项

VDMA 支持以下三种帧同步源：

- 基于 AXI4-Stream 的帧同步（使用 tuser(0) 信号）
 - 读通道使用 m_axis_mm2s_tuser(0) 作为帧起始信号
 - 写通道使用 s_axis_s2mm_tuser(0) 作为帧起始信号
- S2MM 帧同步(s2mm_fsync)
- MM2S 帧同步(mm2s_fsync)

5.3.6 Genlock 同步机制

5.3.6.1 什么是 Genlock？

Genlock，同步锁相，可以使一套或多套系统与同一同步源实现同步。能够使视频的刷新和外部视频源保持一致。当提供了一个适当的信号后，系统就会把它的显示刷新率和这个信号进行锁定。

在许多视频应用中，输入端产生数据的速率往往不同于输出端数据速率，为了避免由速率不一致导致的潜在错误，帧缓冲的使用是很有必要的。帧缓冲机制开辟多个缓冲页，用于保存数据，输入和输出端分别操作不同的帧存，从而避免了冲突。

VDMA 的锁相同步特性正是用于阻止读、写通道同时操作同一个帧存。VDMA 的每个通道都可以选择自己的操作类型（同步锁相主/从或者动态同步锁相主/从），利用该特性，禁止主从接口同时访问同一缓存，从而保持同步。

VDMA 支持四种模式的锁相同步，分别为：

- Genlock Master（锁相同步主端）
- Genlock Slave（锁相同步从端）
- Dynamic Genlock Master（动态锁相同步主端）
- Dynamic Genlock Slave（动态锁相同步从端）

5.3.6.2 Genlock Master

读通道(MM2S)：当配置为 Genlock Master 时，该通道不会跳过或者重复任一帧数据，并把当前帧的编号输出在 mm2s_frame_ptr_out 端口。通道不会检测 mm2s_frame_ptr_in 端口提供的帧编号。Genlock Slave 通道应跟随 Genlock Master 通道变化，但有一定的延迟。延迟大小预定义在寄存器中 (*frmldy_stride[28:24])。

写通道(S2MM)：当配置为 Genlock Master 时，该通道不会跳过或者重复任一帧数据，并把当前帧的编号输出到 s2mm_frame_ptr_out 端口。通道不会检测 s2mm_frame_ptr_in 端口提供的帧编号。Genlock Slave 通道应跟随 Genlock Master 通道变化，但有一定的延迟。延迟大小预定义在寄存器中 (*frmldy_stride[28:24])。

5.3.5.3 Genlock Slave

读通道（MM2S）：当配置为 Genlock Slave 时，该通道会通过跳过或者重复一些帧的方式，尝试与 Genlock Master 同步。通道会对 mm2s_frame_ptr_in 端口进行采样，获取 Genlock Master 的帧编号。为了实现状态反馈，通道会把当前帧的编号输出到 mm2s_frame_ptr_out 端口。

指定通道工作在 Genlock Slave 模式，必须进行如下操作。

- 将 GenlockEn 置 1 (MM2S_VDMACR[3]=1)，使能主、从通道之间的 Genlock 同步。
- 将 GenlockSrc 置 1(MM2S_VDMACR[7]=1),使能内部 Genlock 模式。如果在 Vivado IDE 中同时使能读、写通道，该位默认置位。当 GenlockSRC=1 时，VDMA 默认支持内部同步锁相总线。这样一来就没有必要在外部对帧指针端口 (*frame_ptr_out 和*_frame_ptr_in) 进行连接了。
- 根据主从通道的帧率，使用 mm2s_frmldly_stride[28:24] 设定合适的延迟时间。

写通道（S2MM）：当配置为 Genlock Slave 时，该通道会通过跳过或者重复一些帧的方式，尝试与 Genlock Master 同步。通道会对 s2mm_frame_ptr_in 端口进行采样，获取 Genlock Master 的帧编号。为了实现状态反馈，通道会把当前帧的编号输出到 s2mm_frame_ptr_out 端口。

指定通道工作在 Genlock Slave 模式，必须进行如下操作。

- 将 GenlockEn 置 1 (S2MM_VDMACR[3]=1)，使能主、从通道之间的 Genlock 同步。
- 将 GenlockSrc 置 1(S2MM_VDMACR[7]=1),使能内部 Genlock 模式。如果在 Vivado IDE 中同时使能读、写通道，该位默认置位。当 GenlockSRC=1 时，VDMA 默认支持内部同步锁相总线。这样一来就没有必要在外部对帧指针端口 (*frame_ptr_out 和*_frame_ptr_in) 进行连接了。
- 根据主从通道的帧率，使用 mm2s_frmldly_stride[28:24] 设定合适的延迟时间。

5.3.6.4 Dynamic Genlock Master

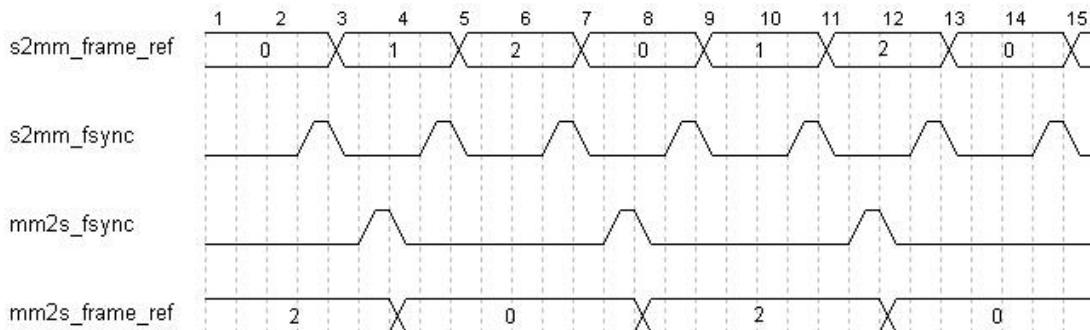
动态 Genlock Master 与 Genlock Master 的区别在于，主通道会跳过从通道正在操作的帧。举例而言，对于三帧存而言，动态 Genlock Master 会按照 0, 1, 2, 0, 1, 2 的顺序循环使用帧存，一旦检测到 Master 即将操作 Slave 正在操作的帧，就会跳过该帧继续循环。因此，如果 Slave 通道一直在操作帧存 1，那么 Master 通道就会在帧 0 和帧 2 之间来回切换。

5.3.6.5 Dynamic Genlock Slave

Dynamic Genlock Slave 通道会操作 Dynamic Genlock Master 通道上一周期操作的

帧。

下图描述了一种简单的 Genlock 操作时序。在这个示例中，S2MM 通道是 Genlock Master，MM2S 通道是 Genlock Slave，并且写通道帧率高于读通道帧率。

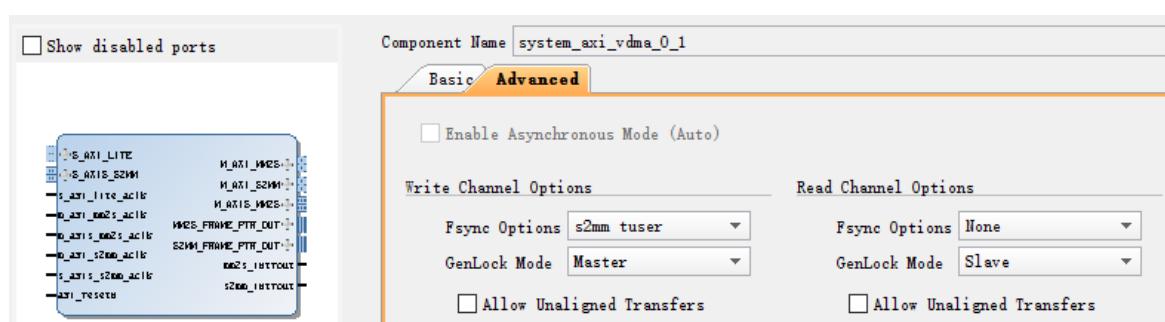
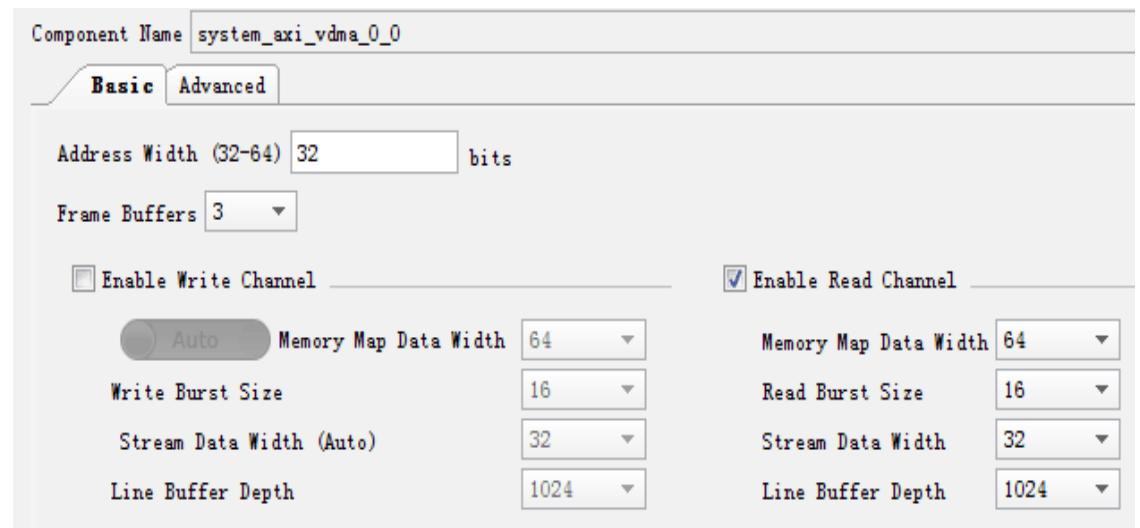


由于读通道帧率慢于写通道，所以读通道仅处理帧 2 和帧 0，跳过帧 1 不做处理。

5.4 使用 VDMA

5.4.1 IP 核配置

Xilinx 集成开发环境升级到 Vivado 之后，VDMA 的配置项比以前少了不少，一定程度上降低了使用难度。主要配置页面如下面两幅图所示。



具体配置项参见下表。

基本配置	高级配置
地址线宽度	是否使能异步模式（自动）
帧存数量	写通道帧同步
是否使能读写通道	写通道 GenLock 模式选择
数据线宽度	写通道是否允许非对齐传输
触发长度	读通道帧同步
AXI-Stream 流数据位宽	读通道 GenLock 模式选择
Line Buffer 深度	读通道是否允许非对齐传输

关于地址线和数据线宽度，需要根据设计的实际情况配置。

Line buffer 深度不能太小。

GenLock 和帧同步前文已经讲解，根据需求自行配置即可。

5.4.2 软件控制流程

以下步骤是最简单的 VDMA 控制初始化操作。

- 写 VDMACR 寄存器，将 VDMACR.RS 设为 1，启动 VDMA 通道。
- 设定有效的帧缓存起始地址。
- 设定帧延迟（仅针对 Genlock 从模式）以及跨度到 FRMDLY_STRIDE 寄存器。
- 设定水平方向字节数到 HSIZE 寄存器。
- 设定竖直方向行数到 VSIZE 寄存器。启动通道的数据传输。

在 VDMA 运行过程中，可以动态的进行显示参数配置，但是需要注意的是，想要使参数生效，必须在设置的最后一步，对 VSIZE 寄存器进行写操作。

最后，给出一段通过 VDMA 对 DDR 读写传输的进行初始化的示例代码：

```
//VDMA configurateAXI VDMA0
//****************************************************************************从 DDR 读数据设置*****
XAxivdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x0, 0x4); //reset
XAxivdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x0, 0x8); //gen-lock

XAxivdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x5C, 0x08000000);
// AXI4 Data Width 为 32 位，是 4 个字节数
// 0xA000000 0x0015F900
XAxivdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x5C+4, 0x0A000000);
// 0x09000000 0x002BF200
XAxivdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x5C+8, 0x09000000);
XAxivdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x54, 640); // 640
XAxivdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x58, 0x01000280);
// 第 0 位： 运行 - 启动 VDMA 操作,在运行 VDMA 时，其状态寄存器中的停止位
赋值为 0 第一位： 循环模式 -通过连续循环帧缓冲
XAxivdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x0, 0x03);
XAxivdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x50, 480); //480

***** 写入 DDR 设置*****
```

```

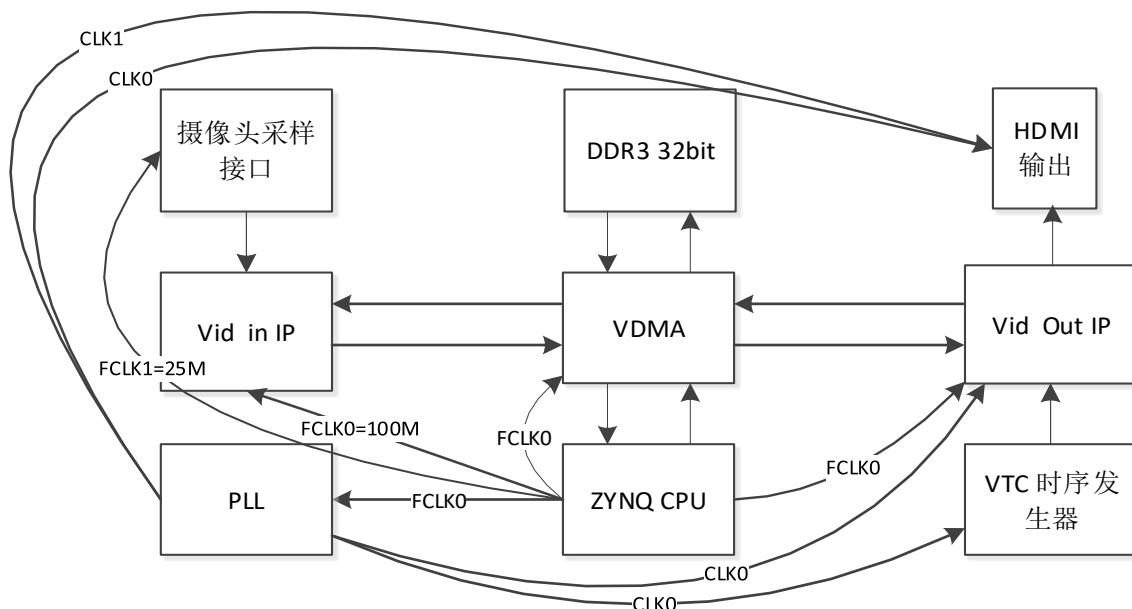
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x30, 0x4); //reset
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x30, 0x8); //genlock

XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xAC, 0x08000000);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xAC+4, 0x0A000000);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xAC+8, 0x09000000);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xA4, 640);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xA8, 0x01000280);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0x30, 0x03);
XAxiVdma_WriteReg(XPAR_AXIVDMA_0_BASEADDR, 0xA0, 480);

```

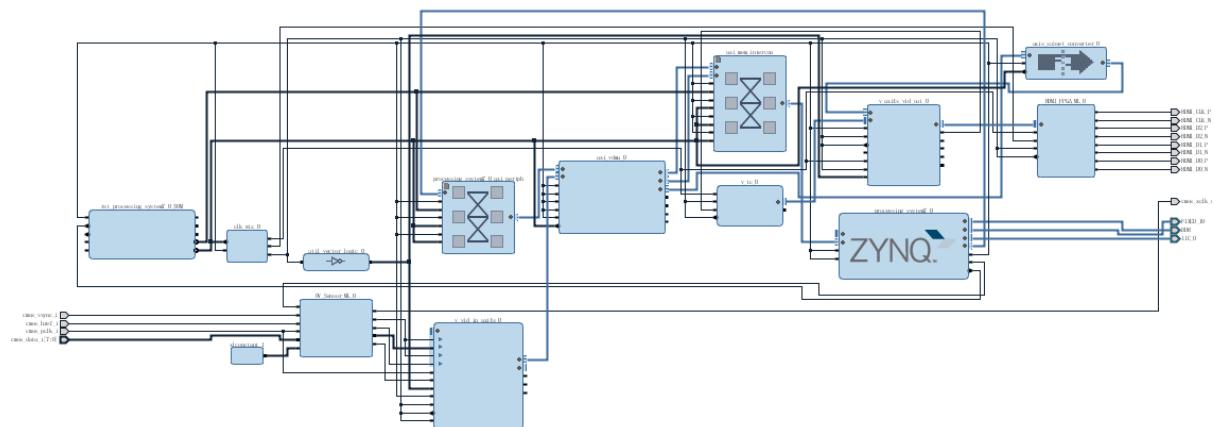
5.5 搭建 VDMA 图像系统

5.5.1 构架方案图



可以看到 VDMA 的图像系统和前面介绍的 DMA 系统相比非常类似。实际上他们都是属于 DMA 系统，只是 VDMA 在配置完成后，可以无需依赖 CPU 可以独立运行，有点类似显卡的功能了。

5.5.2 构 BLOCK 模块化设计方案图



5.6 PS 部分

本课程提供了二种方式启动 VDMA，第一种是通过库函数版本，第二种是通过寄存器版本。寄存器版本主要是验证我们对 VDMA 的寄存器掌握情况。库函数具备更强的功能，和可维护性，但设计过程比较繁琐而且需要对 VDMA 的 api 理解比较透彻，因此此处我们使用寄存器版本比较方便，经过验证效果十分优秀。

表 6-6-1

```
/*
 * main.c
 *
 * Created on: 2017 年 6 月 27 日
 * Author: Administrator
 */

#include "I2C_8bit.h"
#include "xiicps.h"
#include "xil_io.h"
#include "xparameters.h"

#define VDMA_BASEADDR XPAR_AXI_VDMA_0_BASEADDR

#define VIDEO_BASEADDR0 0x01000000
#define VIDEO_BASEADDR1 0x02000000
#define VIDEO_BASEADDR2 0x03000000

#define H_ACTIVE 640
#define V_ACTIVE 480
#define H_STRIDE640
```

```
#define DEMO_MAX_FRAME  (1920*1080*3)
#define DEMO_STRIDE       (1920*3)

XIicPs Iic;

void main()
{
    // Initialize OV5640 regesiter
    I2C_config_init();

    Xil_Out32((VDMA_BASEADDR + 0x030), 0x108B); // enable circular mode
    Xil_Out32((VDMA_BASEADDR + 0x0AC), VIDEO_BASEADDR0); // start address
    Xil_Out32((VDMA_BASEADDR + 0x0B0), VIDEO_BASEADDR1); // start address
    Xil_Out32((VDMA_BASEADDR + 0x0B4), VIDEO_BASEADDR2); // start address
    Xil_Out32((VDMA_BASEADDR + 0x0A8), (H_STRIDE*3)); // h offset (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x0A4), (H_ACTIVE*3)); // h size (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x0A0), V_ACTIVE); // v size (480)

    /*****从 DDR 读数据设置******/
    Xil_Out32((VDMA_BASEADDR + 0x000), 0x8B); // enable circular mode
    Xil_Out32((VDMA_BASEADDR + 0x05c), VIDEO_BASEADDR0); // start address
    Xil_Out32((VDMA_BASEADDR + 0x060), VIDEO_BASEADDR1); // start address
    Xil_Out32((VDMA_BASEADDR + 0x064), VIDEO_BASEADDR2); // start address
    Xil_Out32((VDMA_BASEADDR + 0x058), (H_STRIDE*3)); // h offset (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x054), (H_ACTIVE*3)); // h size (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x050), V_ACTIVE); // v size (480)

    while (1);

}
```

5.7 测试结果



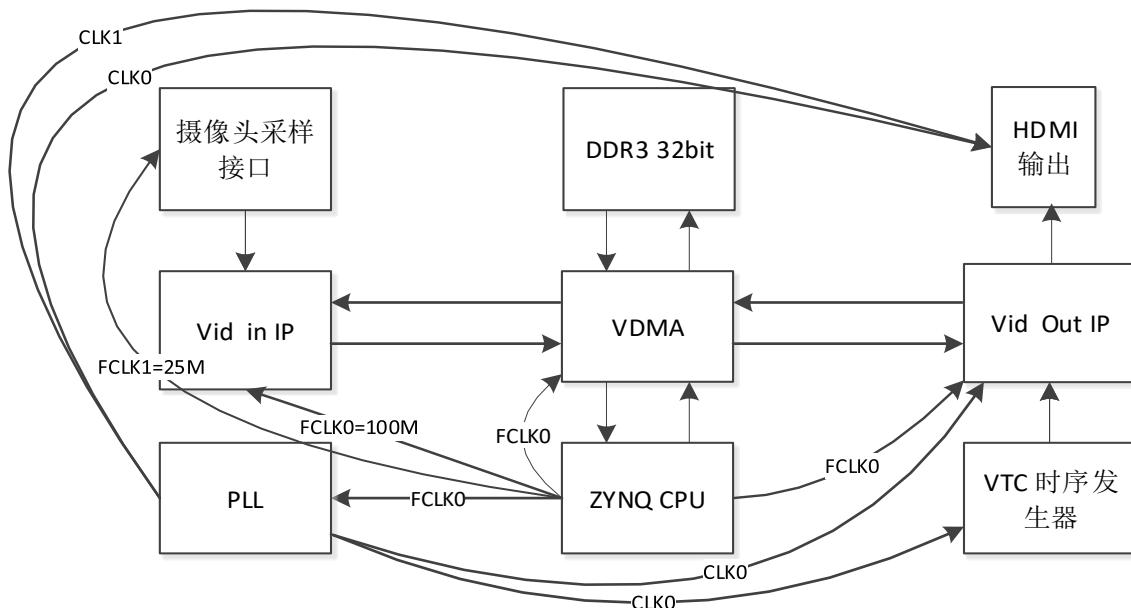
CH06_AXI_VDMA_OV5640 摄像头采集系统

6.1 概述

本章内容和《S03_CH05_AXI_VDMA_OV7725 摄像头采集系统》只是摄像头采用的分辨率不同，其他原理都一样，由于在《S03_CH05_AXI_VDMA_OV7725 摄像头采集系统》中详细介绍了 VDMA 的原理，如果读者只是购买了 OV5640，可以回到《S03_CH05_AXI_VDMA_OV7725 摄像头采集系统》仔细阅读 VDMA 的基础知识。

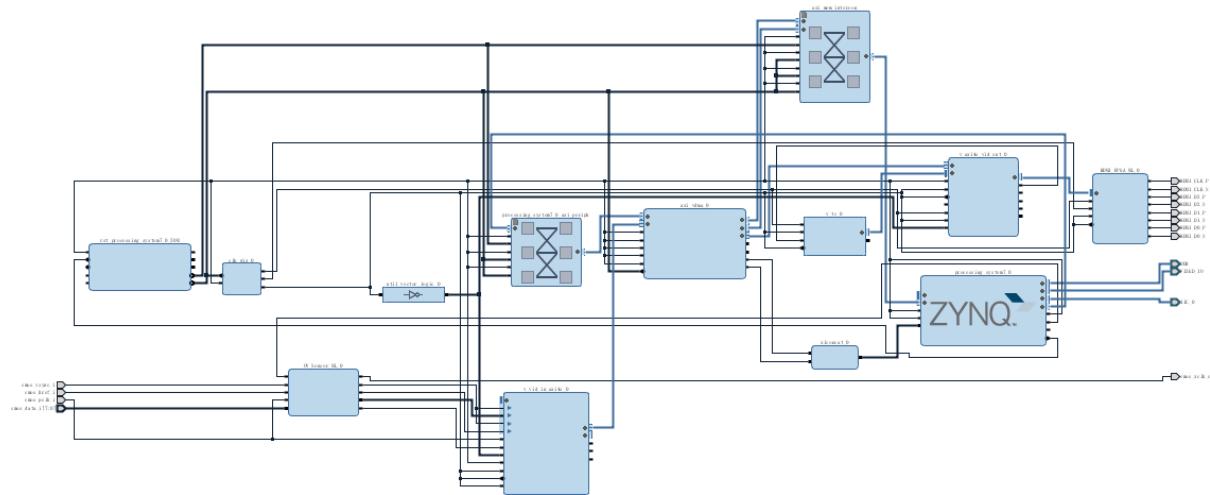
6.2 搭建 VDMA 图像系统

6.2.1 构架方案图



可以看到 VDMA 的图像系统和前面介绍的 DMA 系统相比非常类似。实际上他们都是属于 DMA 系统，只是 VDMA 在配置完成后，可以无需依赖 CPU 可以独立运行，有点类似显卡的功能了。

6.2.2 构 BLOCK 模块化设计方案图



6.3 PS 部分

本课程提供了二种方式启动 VDMA，第一种是通过库函数版本，第二种是通过寄存器版本。寄存器版本主要是验证我们对 VDMA 的寄存器掌握情况。库函数具备更强的功能，和可维护性。和 OV7725 相比，这里的分辨率设置为 1280X720

表 6-6-1

```
/*
 * main.c
 *
 * Created on: 2017 年 6 月 27 日
 * Author: Administrator
 */

#include "I2C_8bit.h"
#include "xiicps.h"
#include "xil_io.h"
#include "xparameters.h"

#define VDMA_BASEADDR    XPAR_AXI_VDMA_0_BASEADDR

#define VIDEO_BASEADDR0 0x01000000
#define VIDEO_BASEADDR1 0x02000000
#define VIDEO_BASEADDR2 0x03000000

#define H_ACTIVE      1280
#define V_ACTIVE      720
#define H_STRIDE640
```

```
#define DEMO_MAX_FRAME  (1920*1080*3)
#define DEMO_STRIDE       (1920*3)

XIicPs Iic;

void main()
{
    // Initialize OV5640 regesiter
    I2C_config_init();

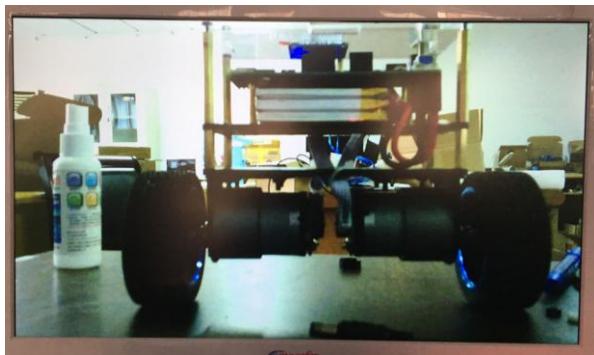
    Xil_Out32((VDMA_BASEADDR + 0x030), 0x108B); // enable circular mode
    Xil_Out32((VDMA_BASEADDR + 0x0AC), VIDEO_BASEADDR0); // start address
    Xil_Out32((VDMA_BASEADDR + 0x0B0), VIDEO_BASEADDR1); // start address
    Xil_Out32((VDMA_BASEADDR + 0x0B4), VIDEO_BASEADDR2); // start address
    Xil_Out32((VDMA_BASEADDR + 0x0A8), (H_STRIDE*3)); // h offset (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x0A4), (H_ACTIVE*3)); // h size (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x0A0), V_ACTIVE); // v size (480)

    /*****从 DDR 读数据设置******/
    Xil_Out32((VDMA_BASEADDR + 0x000), 0x8B); // enable circular mode
    Xil_Out32((VDMA_BASEADDR + 0x05c), VIDEO_BASEADDR0); // start address
    Xil_Out32((VDMA_BASEADDR + 0x060), VIDEO_BASEADDR1); // start address
    Xil_Out32((VDMA_BASEADDR + 0x064), VIDEO_BASEADDR2); // start address
    Xil_Out32((VDMA_BASEADDR + 0x058), (H_STRIDE*3)); // h offset (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x054), (H_ACTIVE*3)); // h size (640 * 4) bytes
    Xil_Out32((VDMA_BASEADDR + 0x050), V_ACTIVE); // v size (480)

    while (1);

}
```

6.4 测试结果



CH07_DMA_LWIP 以太网传输

7.1 概述

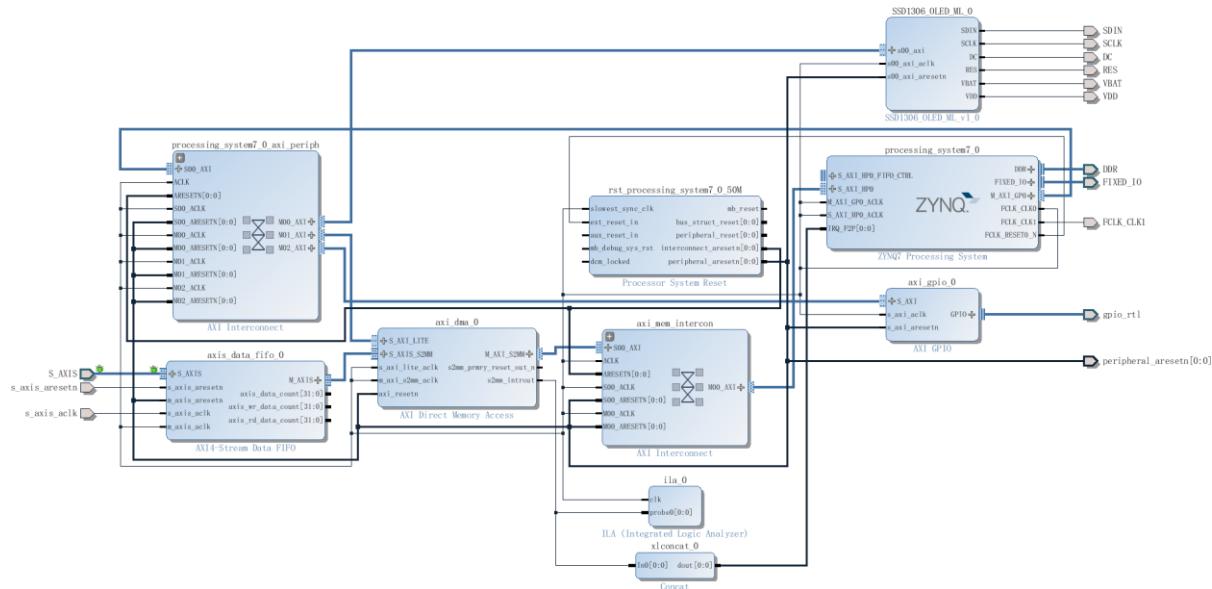
本例程详细创建过程和本季课程第一课《S03_CH01_AXI_DMA_LOOP 环路测试》非常类似，因此如果读者不清楚如何创建工作，请仔细阅读本季第一课时。

本例程的基本原理如下。

PS 通过 AXI GPIO IP 核启动 PL 不间断循环构造 16bit 位宽的 0~1023 的数据，利用 AXI DMA IP 核，通过 PS 的 Slave AXI GP 接口传输至 PS DDR3 的乒乓缓存中。PL 每发完一次 0~1023，AXI DMA IP 核便会产生一个中断信号，PS 得到中断信号后将 DDR3 缓存的数据通过乒乓操作的方式由 TCP 协议发送至 PC 机。

7.2 搭建硬件系统

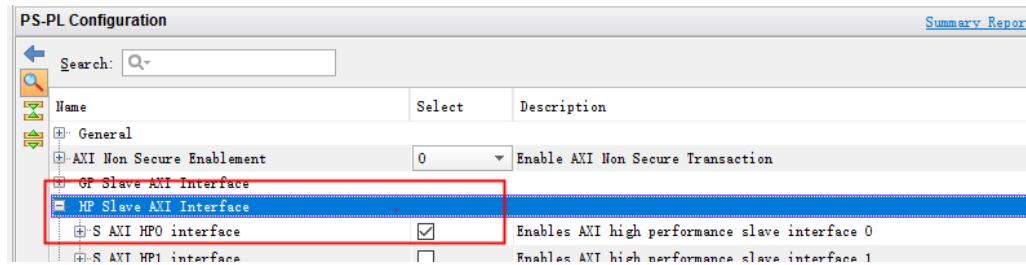
7.2.1 系统构架



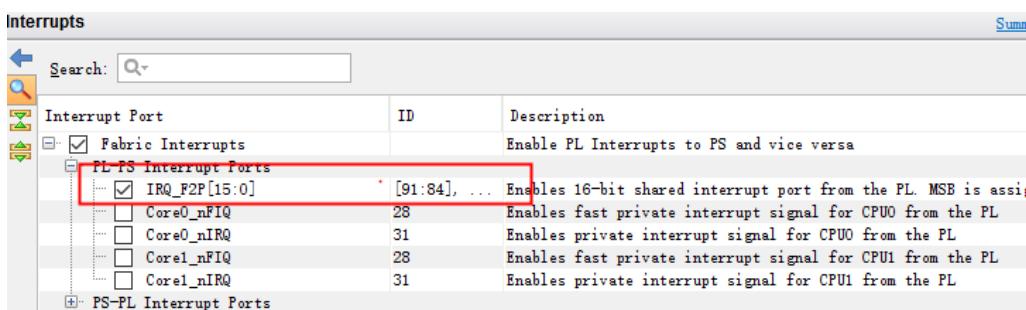
这个系统实际上就是在前面章节 DMA 的基础上去掉 FPGA 读 DMA 通道，只有 FPGA 往 DMA 写输入数据，当 DMA 接收中断产生后，在通过 LWIP 协议，把数据通过网口发送出去。网口是接在 PS 的 ARM 端口的因此，ARM 部分也是要把网口配置好。如下图所示。

7.2.2 启用 HP 接口

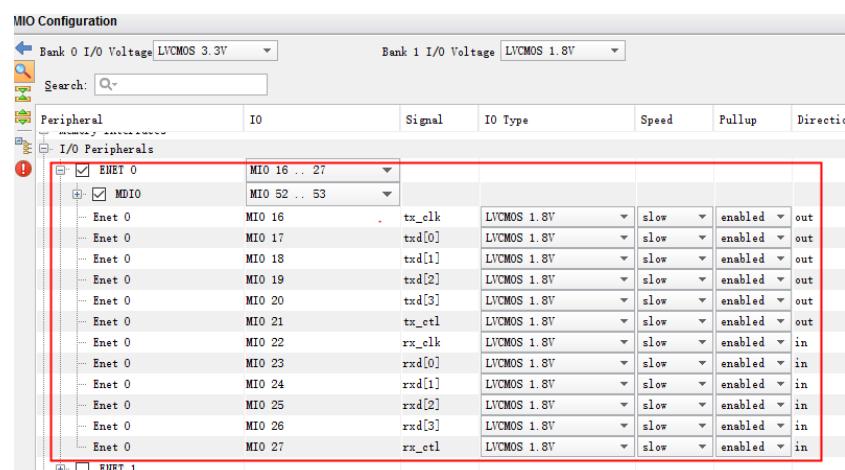
双击 ZYNQ 的 IP 之后启动 HP 接口，这里只要用到 1 个 HP 接口通道



7.2.3 启用 PL 到 PS 的中断资源



7.2.4 启动 PS 部分的以太网接口



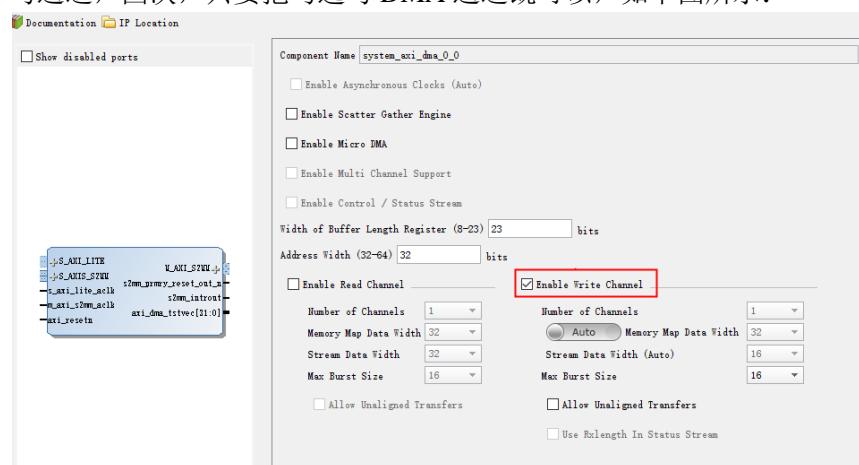
7.2.5 时钟的设置

将 FCLK_CLK0 和 FCLK_CLK1 均设为 100Mhz, 其中 CLK1 为 PL 构造数据部分逻辑的时钟源, 在实际应用中可自由调节时钟频率, 故与 CLK0 分开使用。设置如下图所示。

Component	Clock Source	Requested Frequenc...	Actual Frequenc...	Range (MHz)
+ Processor/Memory Clocks				
+ IO Peripheral Clocks				
- PL Fabric Clocks				
FCLK_CLK0	IO PLL	100.000000	100.000000	0.100000 : 250.000000
FCLK_CLK1	IO PLL	100.000000	100.000000	0.100000 : 250.000000
FCLK_CLK2	IO PLL	50.000000	50.000000	0.100000 : 250.000000
FCLK_CLK3	IO PLL	50	50.000000	0.100000 : 250.000000
+ System Debug Clocks				

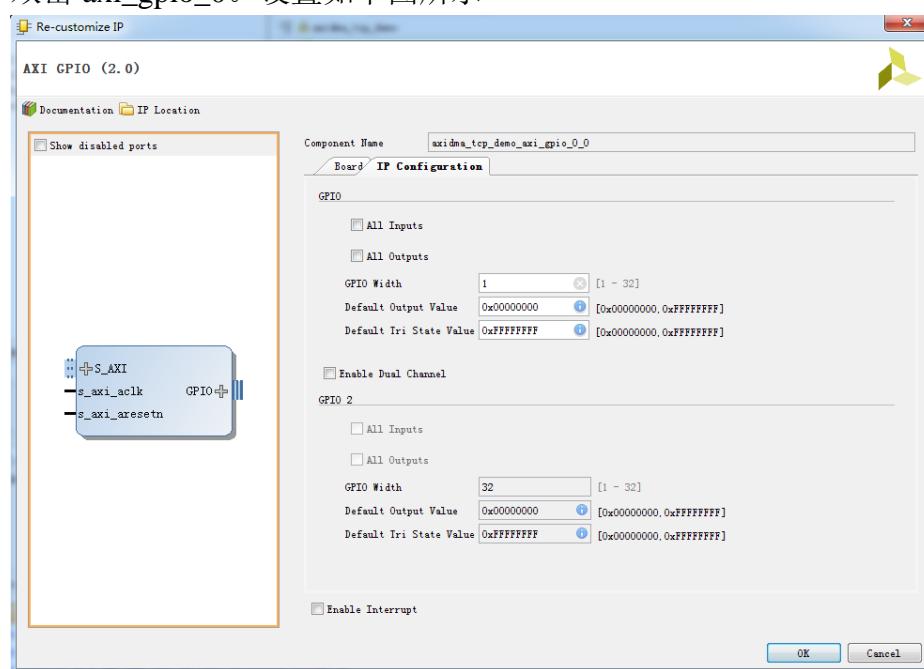
7.2.6 DMA IP 配置

由于只用到了写通道，因次，只要把勾选写 DMA 通道既可以，如下图所示：



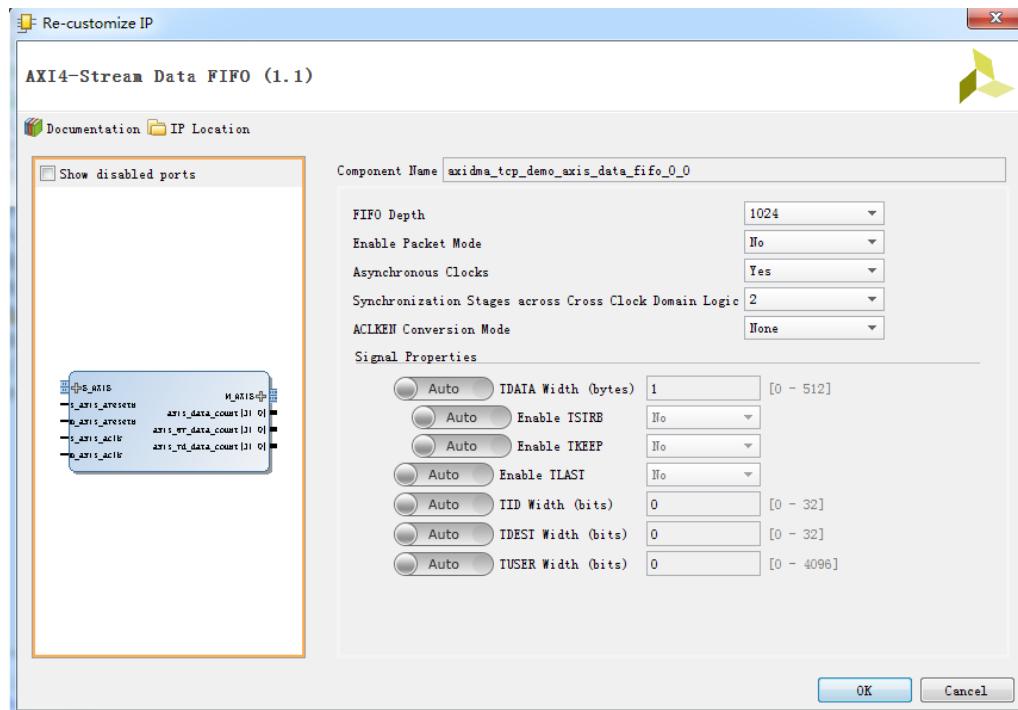
7.2.7 GPIO 的配置

双击 axi_gpio_0。设置如下图所示

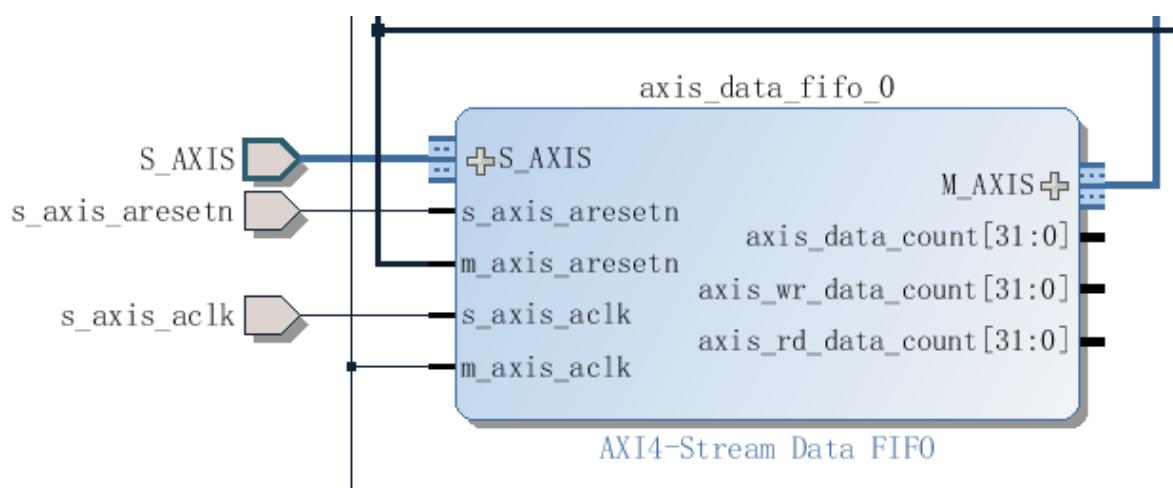


7.2.8 配置 axi_data_fifo_0

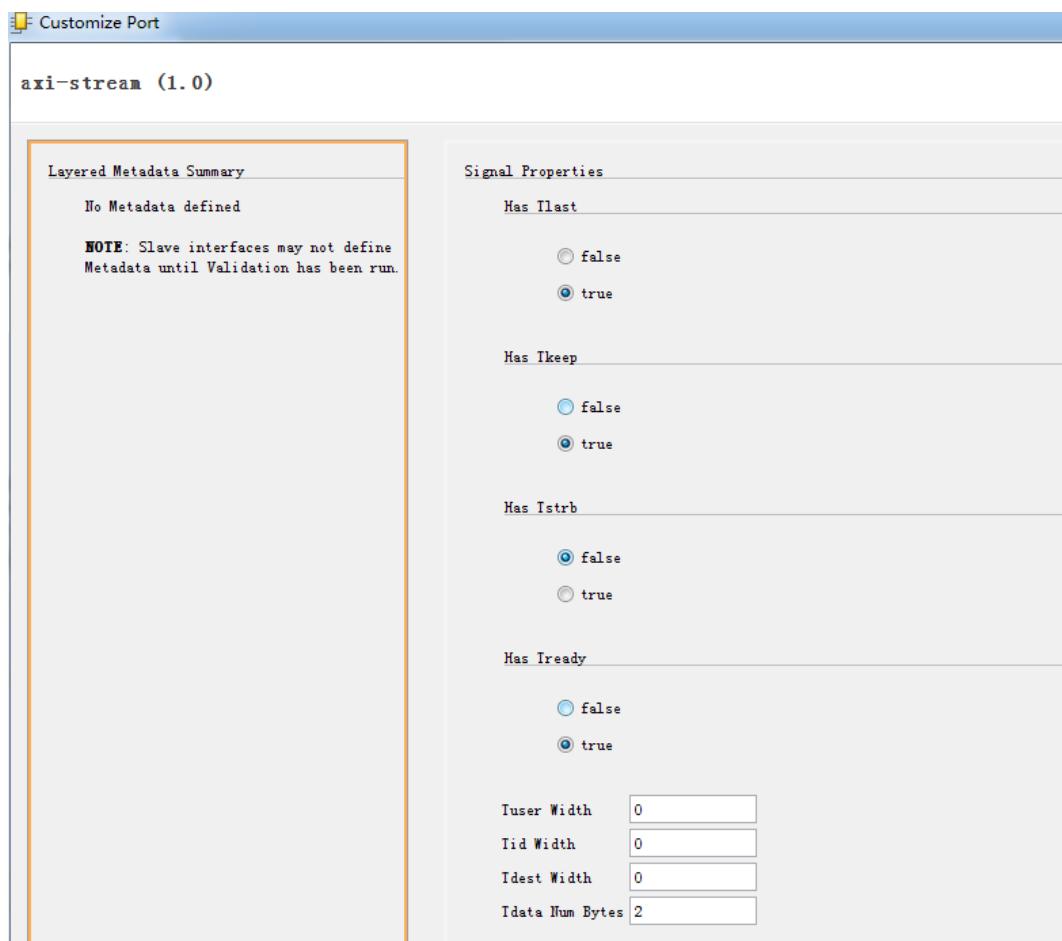
双击 axis_data_fifo_0，设置如下图所示。fifo 在本例程中作为 axi_dma_0 的 S_AXIS_S2MM 接口所在的 FCLK0 时钟域与外部数据生成逻辑 stream 接口所在的 FCLK1 时钟域之间的转换媒介。



7.2.9 设置 S_AXIS 接口

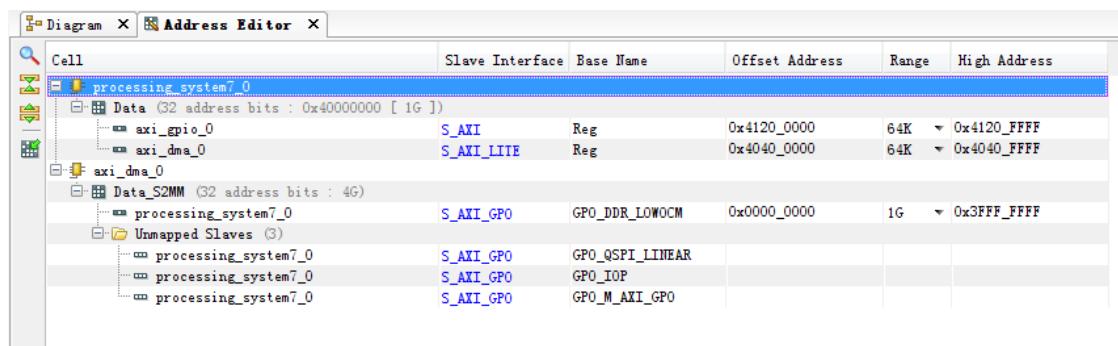


双击 S_AXIS 端口，进行如下图设置



7.2.10 地址空间映射

打开 Address Editor，设置如下图所示



7.3 FPGA 的发送代码

表 8-3-1

```
always@(posedge FCLK_CLK1)
begin
```

```
if(!peripheral_aresetn) begin //系统复位
    S_AXIS_tvalid <= 1'b0;
    S_AXIS_tdata <= 32'd0;
    S_AXIS_tlast <= 1'b0;
    state <=0;
end
else begin
    case(state) //状态机
        0: begin
            if(gpio_tri_o_0&& S_AXIS_tready) begin //当FIFO非满的时候
                S_AXIS_tvalid <= 1'b1;//设置写FIFO数据有效标志为1
                state <= 1;//转入状态1
            end
            else begin
                S_AXIS_tvalid <= 1'b0;
                state <= 0;
            end
        end
        1:begin
            if(S_AXIS_tready) begin //当FIFO非满
                S_AXIS_tdata <= S_AXIS_tdata + 1'b1;
                if(S_AXIS_tdata == 16'd1022) begin //从0-1022一共写入1023个字节
                    S_AXIS_tlast <= 1'b1;//发送最后一个数据
                    state <= 2;
                end
                else begin
                    S_AXIS_tlast <= 1'b0;
                    state <= 1;
                end
            end
            else begin
                S_AXIS_tdata <= S_AXIS_tdata;
                state <= 1;
            end
        end
        2:begin
            if(!S_AXIS_tready) begin //如果FIFO满， 等待
                S_AXIS_tvalid <= 1'b1;
                S_AXIS_tlast <= 1'b1;
                S_AXIS_tdata <= S_AXIS_tdata;
                state <= 2;
            end
            else begin //写入结束
                S_AXIS_tvalid <= 1'b0;
```

```

    S_AXIS_tlast <= 1'b0;
    S_AXIS_tdata <= 16'd0;
    state <= 0;
end
end
default: state <=0;
endcase
end
end

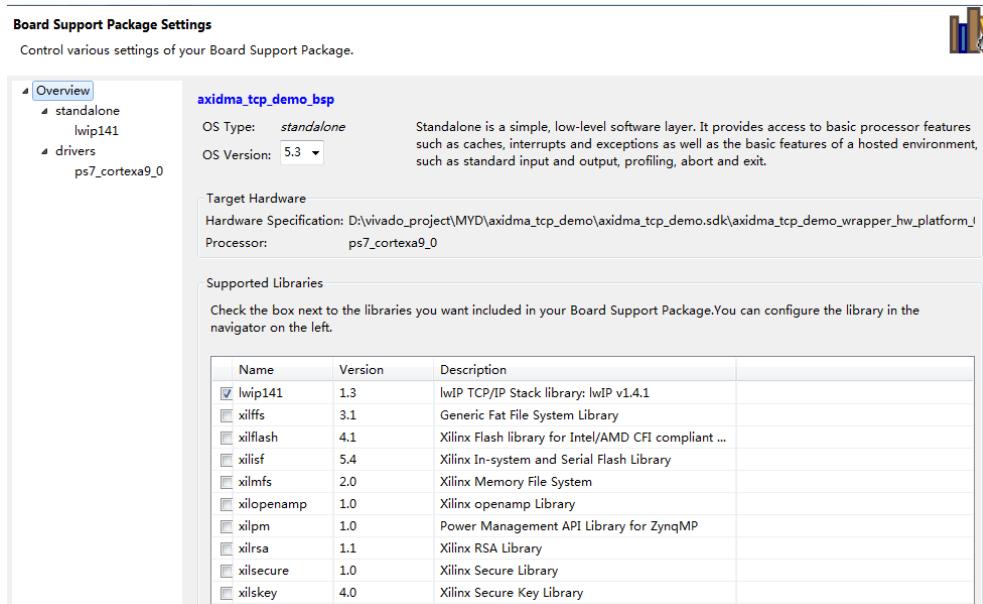
```

发送代码和本季第一课程的代码一样，每次发送 1024X16bit 的数据到通过 DMA 到 DDR 内存。代码比较简单，如果 verilog 语法基础不好的，无法对发送时序精确把我的 FPGA 初学者可以多思考思考，如果还是无法理解可以找我们 FPGA 技术支持。

7.4 PS 部分 BSP 设置

7.4.1 SDK 工程 BSP 设置

进入 sdk 后，新建空工程，命名为 AXI_DMA_PL_PS_Test，同时创建相应的 bsp。修改 AXI_DMA_PL_PS_Test_bsp 的设置，使能 lwip 1.4.1 函数库。如下图所示。



7.4.2 lwip 函数库设置

本例程使用 RAW API，即函数调用不依赖操作系统。传输效率也比 SOCKET API 高，(具体可参考 xapp1026)。将 use_axieth_on_zynq 和 use_emaclite_on_zynq 设为 0。如下图所示。

Configuration for library: lwip141

Name	Value	Default	Type	Description
api_mode	RAW_API	RAW_API	enum	Mode of operation for lwIP (I
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axie
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures em

修改 lwip_memory_options 设置，将 mem_size, memp_n_pbuf, memp_n_tcp_pcb, memp_n_tcp_seg 这 4 个参数值设大，这样会提高 TCP 传输效率。如下图所示。

lwip_memory_options				Options controlling lwIP mem
mem_size	524288	131072	integer	Size of the heap memory (byt
memp_n_pbuf	2048	16	integer	Number of memp struct pbuf
memp_n_sys_timeout	8	8	integer	Number of simultaneously ac
memp_n_tcp_pcb	1024	32	integer	Number of active TCP PCBs. (
memp_n_tcp_pcb_listen	8	8	integer	Number of listening TCP con
memp_n_tcp_seg	1024	256	integer	Number of simultaneously qu
memp_n_udp_pcb	4	4	integer	Number of active UDP PCBs.
memp_num_api_msg	16	16	integer	Number of api msg structure
memp_num_netbuf	8	8	integer	Number of struct netbufs (so
memp_num_netconn	16	16	integer	Number of struct netconns (s
memp_num_tcip_msg	64	64	integer	Number of tcip msg structu

修改 pbuf_options 设置，将 pbuf_pool_size 设大，增加可用的 pbuf 数量，这样同样会提高 TCP 传输效率。如下图所示。

pbuf_options	true	true	boolean	Pbuf Options
pbuf_link_hlen	16	16	integer	Number of bytes that should
pbuf_pool_bufsize	1700	1700	integer	Size of each pbuf in pbuf po
pbuf_pool_size	4096	256	integer	Number of buffers in pbuf po

修改 tcp_options 设置，将 tcp_snd_buf, tcp_wnd 参数设大，这样同样会提高 TCP 传输效率。如下图所示。

tcp_options	true	true	boolean	Is TCP required ?
lwip_tcp	true	true	boolean	Is TCP required ?
tcp_maxrtx	12	12	integer	TCP Maximum retransmissio
tcp_mss	1460	1460	integer	TCP Maximum segment size (
tcp_queue_ooseq	1	1	integer	Should TCP queue segments
tcp_snd_buf	65535	8192	integer	TCP sender buffer space (byt
tcp_synmaxrtx	4	4	integer	TCP Maximum SYN retransmi
tcp_ttl	255	255	integer	TCP TTL value
tcp_wnd	65535	2048	integer	TCP Window (bytes)

修改 temac_adapter_options 设置，将 n_rx_descriptors 和 n_tx_descriptors 参数设大。这样可以提高 zynq 内部 emac dma 的数据迁移效率，同样能提高 TCP 传输效率。如下图所示。

temac_adapter_options	true	true	boolean	Settings for xps_ll-temac/Ax
emac_number	0	0	integer	Zynq Ethernet Interface nun
n_rx_coalesce	1	1	integer	Setting for RX Interrupt coa
n_rx_descriptors	256	64	integer	Number of RX Buffer Descr
n_tx_coalesce	1	1	integer	Setting for TX Interrupt coa
n_tx_descriptors	256	64	integer	Number of TX Buffer Descr
phy_link_speed	CONFIG_LINKSPEED_A...	CONFIG_LINKSPEED_A...	enum	link speed as negotiated by
tcp_ip_rx_checksum_offload	false	false	boolean	Offload TCP and IP Receive
tcp_ip_tx_checksum_offload	false	false	boolean	Offload TCP and IP Transm
tcp_rx_checksum_offload	false	false	boolean	Offload TCP Receive checks
tcp_tx_checksum_offload	false	false	boolean	Offload TCP Transmit check
temac_use_jumbo_frames	false	false	boolean	use jumbo frames

其余选项的参数默认即可，不用修改。点击 OK，重建 bsp。

7.5 PS 部分程序分析

7.5.1 main.c 分析

main 函数的主要流程为：

- 1): 初始化并配置 PL 侧的 AXI GPIO
- 2): 初始化并配置 PL 侧的 AXI DMA
- 3): 初始化并配置 PS 的中断控制器
- 4): 初始化 lwip 协议栈和 PS 的以太网控制器
- 5): 配置 TCP 传输所需的相关参数，并与服务器建立 TCP 连接
- 6): 通过 AXI GPIO 启动 PL 进行数据生成和传输
- 7): 通过 AXI DMA 接收 PL 传输的数据，通过 TCP 发送至 PC 机，并不断循环该过程。

表 8-5-1

```
/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 * axi dma test
 *
 */

#include "dma_intr.h"
#include "timer_intr.h"
#include "sys_intr.h"
#include "xgpio.h"
#include "OLED.h"

#include "lwip/err.h"
#include "lwip/tcp.h"
#include "lwipopts.h"
#include "netif/xadapter.h"
#include "lwipopts.h"

static XScuGic Intc; //GIC
static XScuTimer Timer;//timer
XAxiDma AxiDma;
u16 *RxBufferPtr[2]; /* ping pong buffers*/
```

```
volatile u32 RX_success;
volatile u32 TX_success;

volatile u32 RX_ready=1;
volatile u32 TX_ready=1;

#define TIMER_LOAD_VALUE      XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ / 8 //0.25S

extern void send_received_data(void);
extern unsigned tcp_client_connected;

char oled_str[17]="";
static XGpio Gpio;

#define AXI_GPIO_DEV_ID      XPAR_AXI_GPIO_0_DEVICE_ID

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

}

int main(void)
{

    int Status;
    struct netif *netif, server_netif;
    struct ip_addr ipaddr, netmask, gw;

    /* the mac address of the board. this should be unique per board */
    unsigned char mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };

    /* Initialize the ping pong buffers for the data received from axidma */
    RxBufferPtr[0] = (u16 *)RX_BUFFER0_BASE;
    RxBufferPtr[1] = (u16 *)RX_BUFFER1_BASE;

    XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);
```

```
XGpio_SetDataDirection(&Gpio, 1, 0);
init_intr_sys();
TcpTmrFlag = 0;

netif = &server_netif;

IP4_ADDR(&ipaddr, 192, 168, 1, 10);
IP4_ADDR(&netmask, 255, 255, 255, 0);
IP4_ADDR(&gw, 192, 168, 1, 1);

/*lwip library init*/
lwip_init();
/* Add network interface to the netif_list, and set it as default */
if (!xemac_add(netif, &ipaddr, &netmask, &gw, mac_etherenet_address,
XPAR_XEMACPS_0_BASEADDR)) {
    xil_printf("Error adding N/W interface\r\n");
    return -1;
}
netif_set_default(netif);

/* specify that the network if is up */
netif_set_up(netif);

/* initialize tcp pcb */
tcp_send_init();

XGpio_DiscreteWrite(&Gpio, 1, 1);
oled_fresh_en(); // enable oled
Timer_start(&Timer);

while (1)
{
    /* call tcp timer every 250ms */
    if(TcpTmrFlag)
    {
        tcp_tmr();
        TcpTmrFlag = 0;
    }
    /*receive input packet from emac*/
    xemacif_input(netif); // 将MAC队列里的packets传输到你的LwIP/IP stack里
    /* if connected to the server, start receive data from PL through axidma, then transmit the data to
the PC software by TCP*/
    if(tcp_client_connected)
        send_received_data();
```

```

    }
    return 0;
}

}

```

7.5.2 AXI DMA 数据传输过程

例程中 axi dma 采用了 simple transfer 方式，通过 XAxiDma_SimpleTransfer 函数完成。每次 dma 传输都需要 PS 主动发起，PS 通过 AXI 总线配置 PL 侧 axi dma 内部寄存器，发起一次 dma 传输。dma 传输发起后，axi dma 开始通过 S_AXIS_S2MM 接口接收数据，当其中的 tlast 信号被拉高，则代表当次传输所需要的数据发送完毕，当该次 dma 传输结束，axi dma 通过 s2mm_introut 产生中断信号，触发 PS 中断控制器产生中断，PS 通过中断服务函数 Dma_RxIsr 清除 axi dma 的中断状态，在 DM 中断函数中，拉高 dma 完成指示信号 packet_trans_done，一次完整的 simple transfer 的 dma 传输结束。下表为 dma 中断接收函数，接收来自 PL 的中断信号，并且设置 packet_trans_done。

表：8-5-2-1 DMA_RxIntrHandler DMA 中断接收函数

```

/****************************************************************************
 *
 * This is the DMA RX interrupt handler function
 *
 * It gets the interrupt status from the hardware, acknowledges it, and if any
 * error happens, it resets the hardware. Otherwise, if a completion interrupt
 * is present, then it sets the RxDone flag.
 *
 * @param    Callback is a pointer to RX channel of the DMA engine.
 *
 * @return   None.
 *
 * @note    None.
 *
 */
static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

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

    /* Acknowledge pending interrupts */
}

```

```
XAxIDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

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

/*
 * If error interrupt is asserted, raise error flag, reset the
 * hardware to recover from the error, and return with no further
 * processing.
 */
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
    xil_printf("rx error! \r\n");
    return;
}

/*
 * If completion interrupt is asserted, then set RxDone flag
 */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
    if(packet_trans_done)
        xil_printf("last transmission has not finished!\r\n");
    else
        /*set the axidma done flag*/
        packet_trans_done = 1;
}
}
```

PS 的 dma 数据接收采用了乒乓操作的模式，两个缓冲区交替进行数据接收。

需要注意的是，XAxIDma_SimpleTransfer 函数中 Length，以字节为单位，每次发起 dma 时，所设置的 Length 的值必须大于或等于 PL 实际传输的数据长度，否则会出现错误。本例程中设置的长度为 2048 字节。

first_trans_start 是为了进行第一次先进行一次 DMA 中断传输，这样完成后设置 first_trans_start 为 0。以后每次完成网络传输后，再启动 DMA 接受。

TCP 数据包的发送主要依赖于 tcp_write 和 tcp_output 两个函数，tcp_write 将所需要发送的数据写入 tcp 发送缓冲区等待发送，tcp_output 函数则将缓存区内数据包发送出去。在发送 TCP 数据包时，这两个函数往往要同时配合使用。

收发送函数的具体源码如下表。

8-5-2: send_received_data() 发送函数源码

```
void send_received_data()
{
#if __arm__
    int copy = 3;
#else
    int copy = 0;
#endif
    err_t err;
    int Status;
    struct tcp_pcb *tpcb = connected_pcb;

    /*initial the first axdma transmission, only excuse once*/
    if(!first_trans_start)
    {
        Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)RxBufferPtr[0],
                                         (u32)(PAKET_LENGTH), XAXIDMA_DEVICE_TO_DMA);
        if (Status != XST_SUCCESS)
        {
            xil_printf("axi dma failed! 0 %d\r\n", Status);
            return;
        }
        /*set the flag, so this part of code will not excuse again*/
        first_trans_start = 1;
    }

    /*if the last axidma transmission is done, the interrupt triggered, then start TCP transmission*/
    if(packet_trans_done)
    {

        if (!connected_pcb)
            return;

        /* if tcp send buffer has enough space to hold the data we want to transmit from PL, then start tcp
transmission*/
        if (tcp_sndbuf(tpcb) > SEND_SIZE)
        {
            /*transmit received data through TCP*/
            err = tcp_write(tpcb, RxBufferPtr[packet_index & 1], SEND_SIZE, copy);
            if (err != ERR_OK) {
                xil_printf("txperf: Error on tcp_write: %d\r\n", err);
                connected_pcb = NULL;
                return;
            }
        }
    }
}
```

```

err = tcp_output(tpcb);
if (err != ERR_OK) {
    xil_printf("txperf: Error on tcp_output: %d\r\n",err);
    return;
}

packet_index++;
/*clear the axidma done flag*/
packet_trans_done = 0;

/*initial the other axidma transmission when the current transmission is done*/
Status = XAxiDma_SimpleTransfer(&AxiDma, (u32)RxBufferPtr[(packet_index + 1)&1],
                                (u32)(PAKET_LENGTH), XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS)
{
    xil_printf("axi dma %d failed! %d \r\n", (packet_index + 1), Status);
    return;
}

}
}

```

7.5.3 TCP 发送流程

在本例程中，zynq 作为客户端，PC 作为服务器。由 zynq 向 PC 主动发起 TCP 连接请求，通过 `tcp_connect` 函数便可以完成这个过程。该函数的参数包含了一个回调函数指针 `tcp_connected_fn`，该回调函数将在 TCP 连接请求三次握手完成后被自动调用。该回调函数被调用时代表客户端和服务器之间的 TCP 连接建立完成。在本例程中，该回调函数被定义为 `tcp_connected_callback`，在该函数中，拉高连接建立完成信号 `tcp_client_connected`，并通过 `tcp_sent` 函数配置另一个 TCP 发送完成的回调函数。该回调函数在每个 TCP 包发送完成后会被自动调用，代表 TCP 包发送完成。该回调函数在本例程中被定义为 `tcp_sent_callback`，仅作发送完成数据包的计数。

表 tcp_connected_callback 回调函数

```
static err_t
tcp_connected_callback(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    xil_printf("txperf: Connected to iperf server\r\n");

    /* store state */
    connected_pcb = tpcb;
```

```

/* set callback values & functions */
tcp_arg(tpcb, NULL);
tcp_sent(tpcb, tcp_sent_callback);

tcp_client_connected = 1;

/* initiate data transfer */
return ERR_OK;
}

```

表 tcp_sent_callback 回调函数

```

/*this fuction just used to count the tcp transmission times*/
static err_t
tcp_sent_callback(void *arg, struct tcp_pcb *tpcb, u16_t len)
{
    tcp_trans_done++;
    return ERR_OK;
}

```

7.6 连接测试

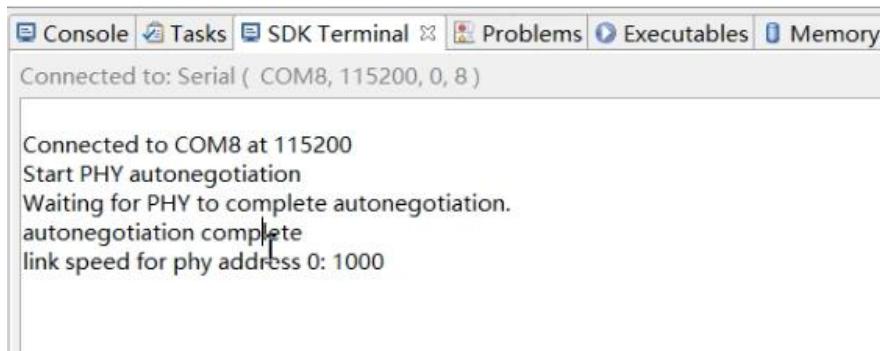
把开发板网卡通过网线接到 PC 网口上，修改 IP 地址如下图



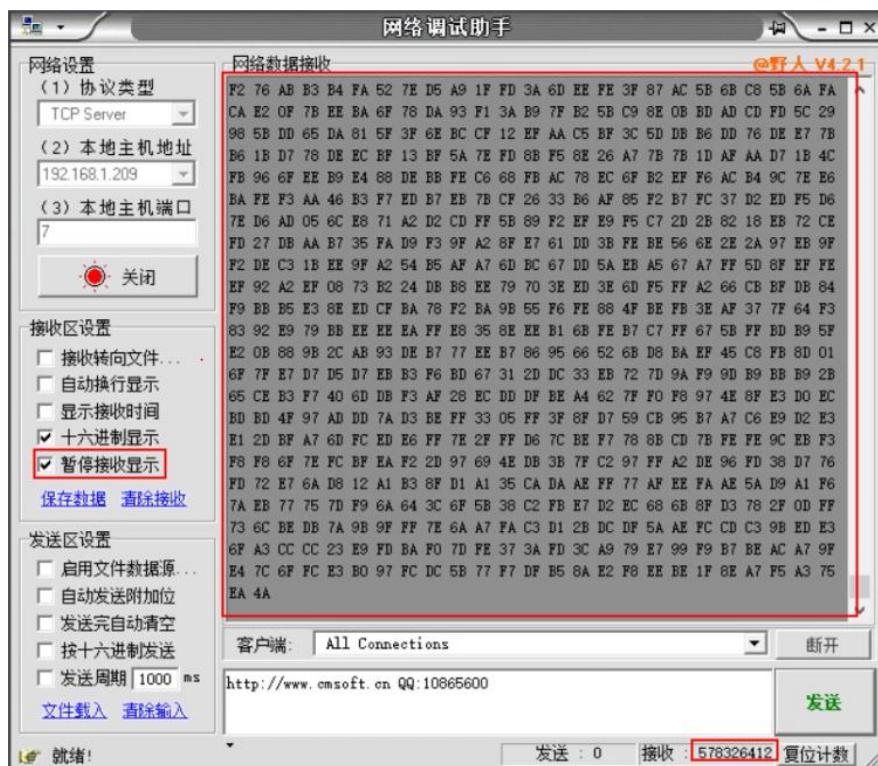
打开网络调试助手，第一次用的时候 windows 会提示你是否允许访问网络一定要选择是，否则你就无法通信了。设置电脑为 TCP Server 本机 IP 为刚才设置的 192.168.1.209 端口号为 7。



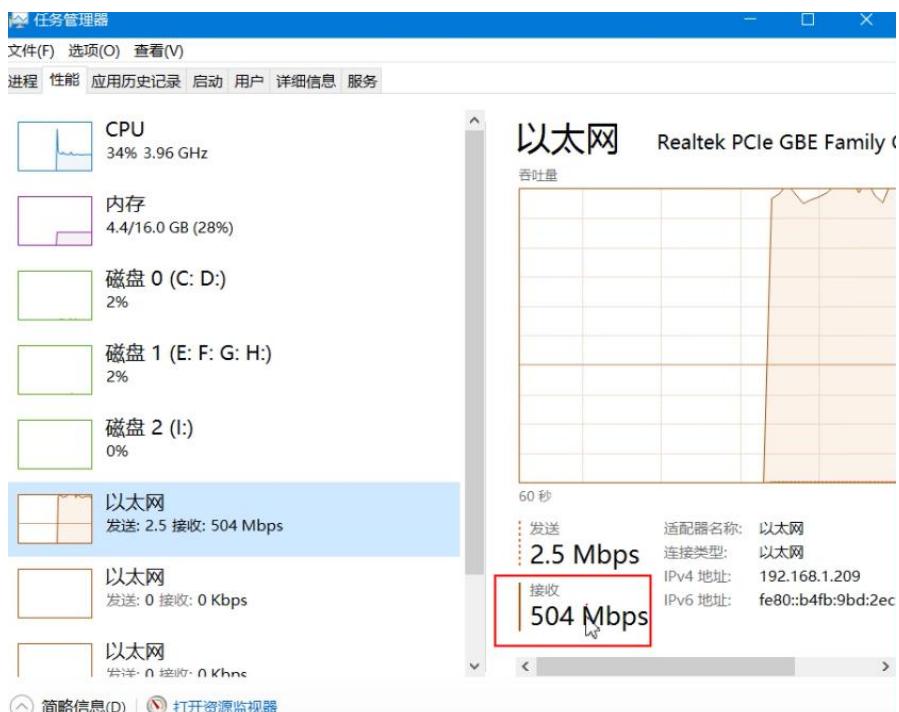
在 SDK 里面打开串口，并且启动 SDK 调试(调试方法前面已经讲过)，板子是 client 可以看到成功连接到了 PCB 上。



这个时候可以看到网络调试助手接收到数据了，由于数据量大，刷新数据显示，会导致电脑变慢，这里把勾选暂停显示。右下角是接收数据的计数器，可以看到计数器在飞奔中。



现在检测下网速，可以看到时间速度在 500Mbps/s 左右查看网速，大概是在 62MB/S-70MB/S 的速度，当然我们也可以通过优化实现更高速度的传输。



在 SDK 里面设定内存空间的查看地址，查看内存中的数据，可以看到正是 PL 发出的数据。

The screenshot shows the Xilinx Vivado IDE interface with the "Memory" tab selected. The left sidebar displays a tree view under "Monitors" with two entries: "0x10320000" and "0x103000". The main window shows a memory dump for the range 0x10320000 to 0x10320000. The data is presented in a grid with columns labeled 0 - 3, 4 - 7, 8 - B, and C - F. The first row of data is highlighted in blue, showing the address 10320000 and the value 00010000.

Address	0 - 3	4 - 7	8 - B	C - F
10320000	00010000	00030002	00050004	00070006
10320010	00090008	000B000A	000D000C	000F000E
10320020	00110010	00130012	00150014	00170016
10320030	00190018	001B001A	001D001C	001F001E
10320040	00210020	00230022	00250024	00270026
10320050	00290028	002B002A	002D002C	002F002E
10320060	00310030	00330032	00350034	00370036
10320070	00390038	003B003A	003D003C	003F003E

CH08_DMA_4_Video_Switch 视频切换系统

8.1 概述

本例程详细创建过程和本季课程第一课《S03_CH01_AXI_DMA_LOOP 环路测试》非常类似，因此如果读者不清楚如何创建工作，请仔细阅读本季第一课时。本例程的基本原理如下。

PL 通过 OV7725 OV7725 实时 采集 1 路 64 0×480 0×480 视频，分别将原始彩色、 视频，分别将原始彩色、 R 分量、 G 分量、 B 分量作为常量输出，实现背景为红、绿、蓝 视频共 4 路视频通过 4 个独立的 AXI DMA IP 核传输至 PS 的 DDR 中进行缓存，然后再通过 AXIDMA AXIDMAAXIDMA 将 4 路视频同时 路视频同时 从 DDR 读出，通过 PL 在 VGA 显示器上 显示器上 切换显示 其中任意 1 路。视频切换通过 。视频切换通过 MZ7X 底板的 SW2 按键实现， 每按 下一次 切换一路视频 。

8.2 修改 OV_Sensor_ML 摄像头采集 IP

外接摄像头字卡后，由于 MIZ735 开发板只有 2 路摄像头视频输入接口，无法满足演示 4 路视频输入的接口需求，因此修改 OV_Sensor_ML ip 使之输出 4 路数据通路。修改的代码如下：

表 9-2-1:

```
module OV_Sensor_ML(
    input CLK_i,
    //----- CMOS sensor hardware interface -----
    input      cmos_vsync_i, //cmos vsync
    input      cmos_href_i, //cmos hsync refrence
    input      cmos_pclk_i, //cmos pxiel clock
    output     cmos_xclk_o, //cmos externl clock
    input[7:0] cmos_data_i, //cmos data
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    // output de_o,//data enable.
    output [23:0] rgb_o1,//data output,
    output [23:0] rgb_o2,//data output,
    output [23:0] rgb_o3,//data output,
    output [23:0] rgb_o4,//data output,
    output vid_clk_ce
);
//-----视频输出解码模块-----
wire [15:0]rgb_o_r;
assign rgb_o1 = {rgb_o_r[4:0],3'd0 ,rgb_o_r[10:5],2'd0,rgb_o_r[15:11],3'd0};
assign rgb_o2 = {5'b11111 ,3'd0 ,rgb_o_r[10:5],2'd0,rgb_o_r[15:11],3'd0};
```

```

assign rgb_o3 = {rgb_o_r[4:0],3'd0 ,rgb_o_r[10:5],2'd0,5'b11111      ,3'd0};
assign rgb_o4 = {rgb_o_r[4:0],3'd0 ,6'b111111     ,2'd0,rgb_o_r[15:11],3'd0};

reg [7:0]cmos_data_r;
reg cmos_href_r;
reg cmos_vsync_r;

always@(posedge cmos_pclk_i)
begin
    cmos_data_r <= cmos_data_i;
    cmos_href_r <= cmos_href_i;
    cmos_vsync_r<= cmos_vsync_i;
end
//assign rgb_o = 24'b11111111_00000000_11111111;
cmos_decode cmos_decode_u0(
    //system signal.
    .cmos_clk_i(CLK_i),//cmos senseor clock.
    .rst_n_i(RESETn_i2c),//system reset.active low.
    //cmos sensor hardware interface.
    .cmos_pclk_i(cmos_pclk_i),//(cmos_pclk),//input pixel clock.
    .cmos_href_i(cmos_href_r),//(cmos_href),//input pixel hs signal.
    .cmos_vsync_i(cmos_vsync_r),//(cmos_vsync),//input pixel vs signal.
    .cmos_data_i(cmos_data_r),//(cmos_data),//data.
    .cmos_xclk_o(cmos_xclk_o),//(cmos_xclk),//output clock to cmos sensor.
    //user interface.
    .hs_o(hs_o),//hs signal.
    .vs_o(vs_o),//vs signal.
    //.
    .de_o(de_o),//data enable.
    .rgb565_o(rgb_o_r),//data output
    .vid_clk_ce(vid_clk_ce)
);

count_reset_v1#(
    .num(20'hffff0)
)(
    .clk_i(CLK_i),
    .rst_o(RESETn_i2c)
);

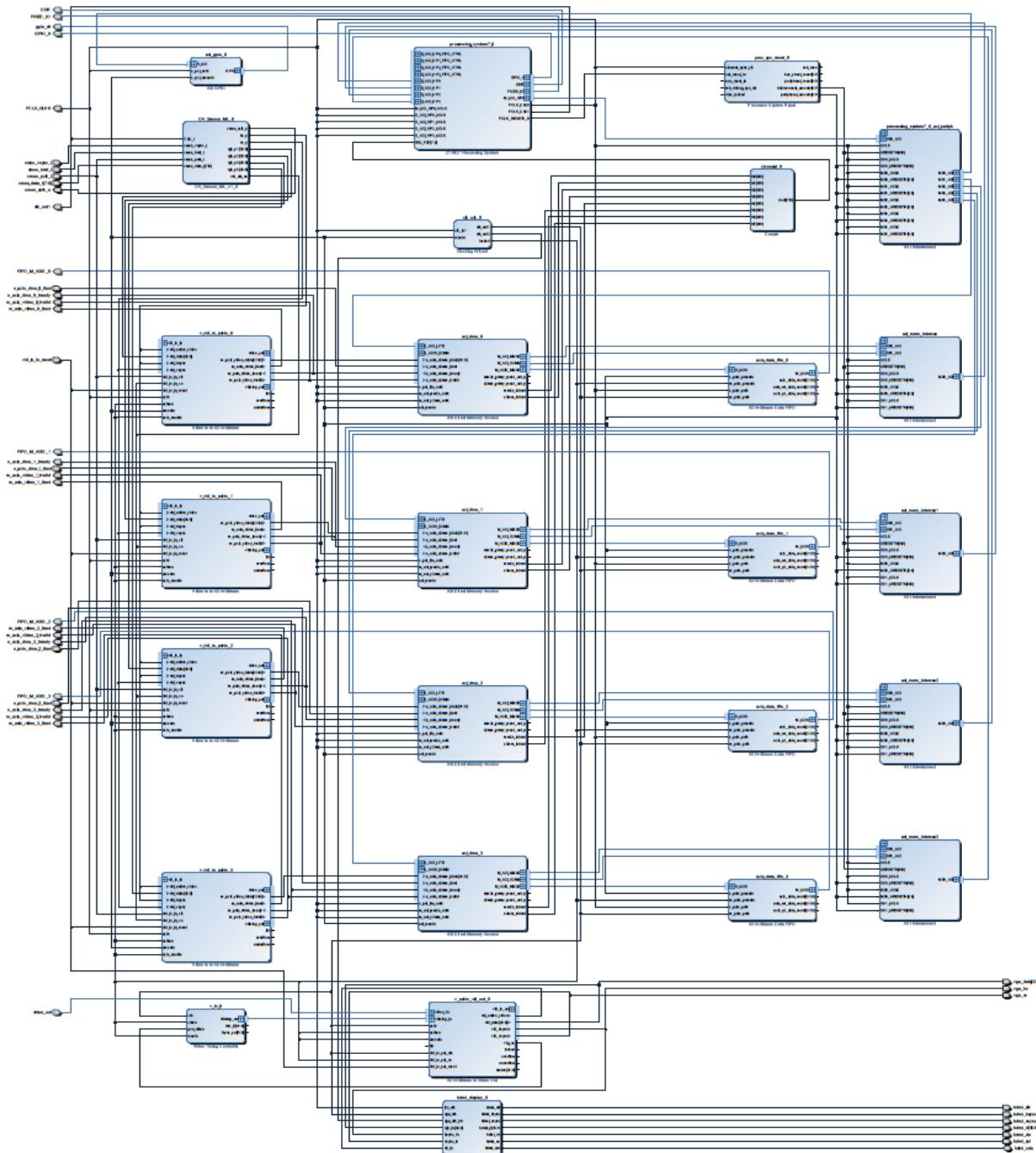
```

修改后代码为的为 OV_Sensor_ML.v 上表中红色字体部分，可以看出，代码作了简单修改，增加了 3 路数据输出，为了让数据颜色有对比，第一路保持原始图像数据颜色，剩余三路增加了颜色背景。这样在做切换测试的时候就可以看到不同的通路变化了。

8.3 搭建硬件系统

8.3.1 系统图

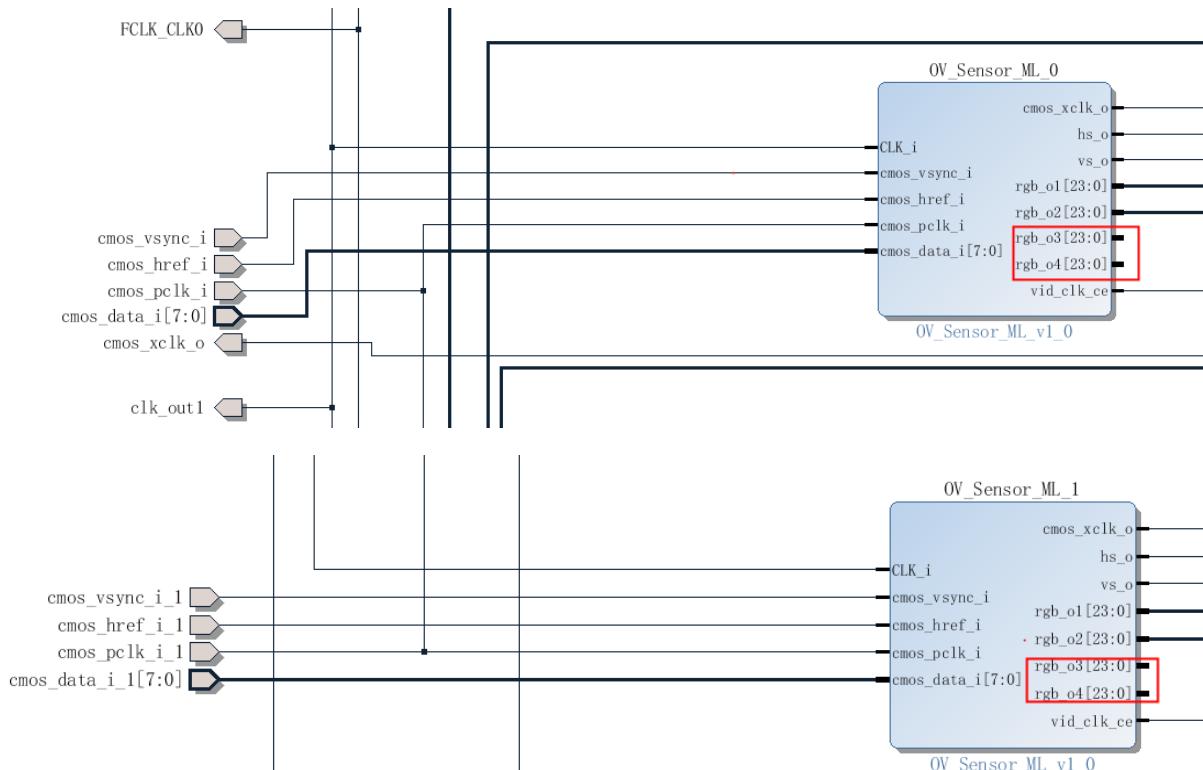
完成了 IP 的修改后，下面就可以搭建硬件系统了，由于 VIVADO 采用了图形化设计，带来了很大便捷。下面把系统构架图贴出来。



由于图片太大，只能看到大概的框架，大家学习的时候可以打开工程放大后去阅读，这里为了分析的时候方便把局部视图放大截图。

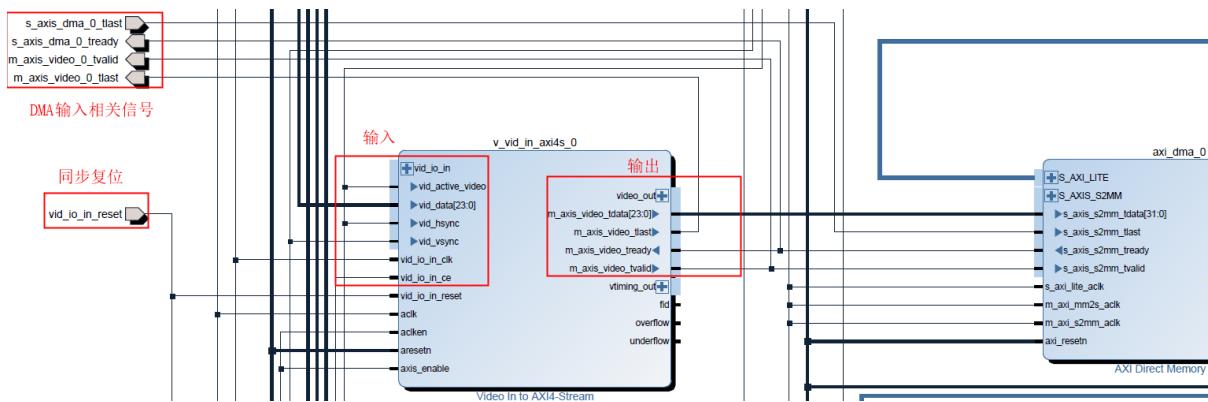
8.3.2 OV_Sensor_ML IP 接线图

下图中主要是前面我们自定义 OV_Sensor_ML 采集 IP 修改后的图形界面。可以看到多出了 rgb_o1、rgb_o2、rgb_o3、rgb_o4 接口这样我们就虚拟了 4 路摄像头数据输入接口啦（本章中只使用了 2 路）。OV_Sensor_ML 前天的信号还是不变。下图中，还有 2 个信号分别是 FCLK_CLK0 和 clk_out1 他们分别是 VID_IN IP 和 VID_OUT IP 相关的时钟，把它们引出去到顶层模块中，后面需要使用到。



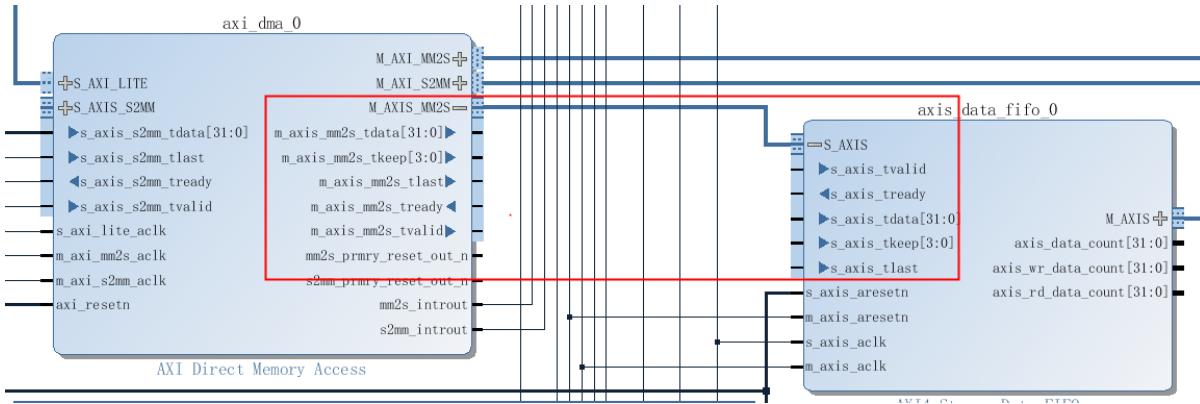
8.3.3 vid_in IP 的接线图

下图大家可以放大后观看，vid_in IP 的输入接口是连接到摄像头采样输出 IP 的。vid_in IP 的输出接口是和和 DMA 链接了。DMA 输入相关的信号被引出到外部，用来添加 FPGA 代码实现写 DMA 时序。还有一个 vid_io_reset 信号，是用来控制所有 vid_in 和 vid_out IP 的同步，也是连接到外部，用 FPGA 代码控制。

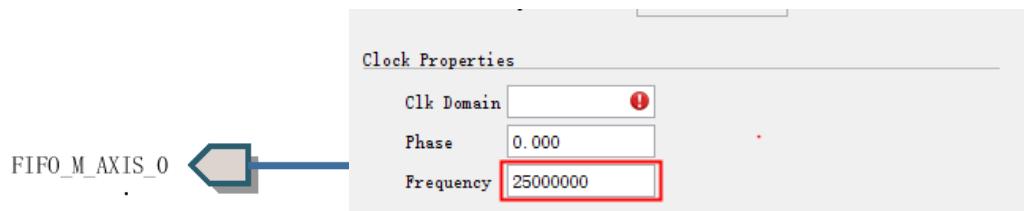


8.3.4 DMA 和 FIFO 通路

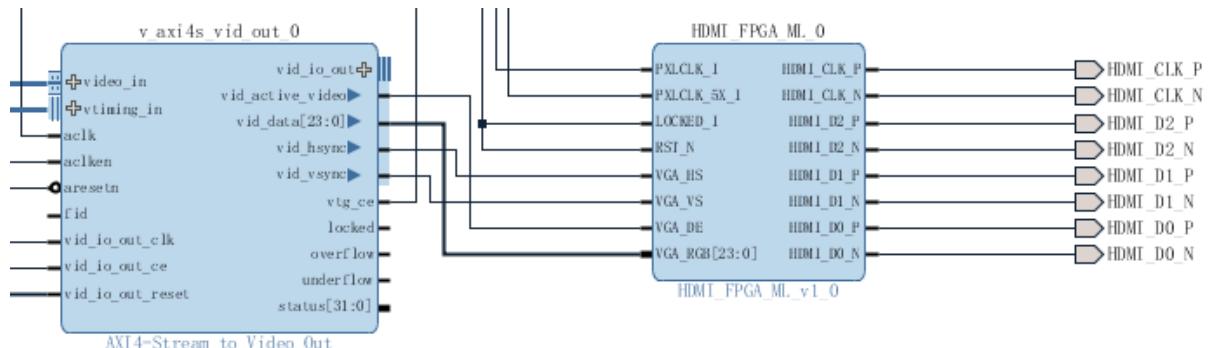
下图是 DMA 和 data fifo 的链接通路。



上图中 data fifo 的 M_AXIS 将被引出到外部受 FPGA 代码控制。如下图, FIFO_M_AXIS_0 就是连接到 axis_data_fifo_0 的 M_AXIS 接口的。双击此接口需要设置时钟, 这里的数据速度时钟是 25MHZ 。不同的分辨率应当设置对应的分辨率时钟。



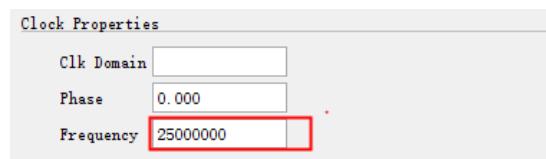
8.3.5 vid_out IP 的通路



上图中的 vid out IP 数据输入通道如图所示

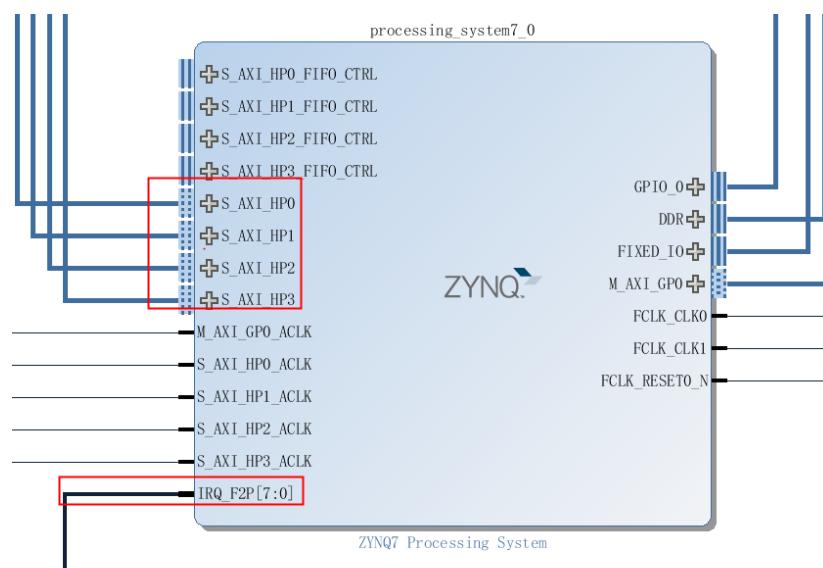


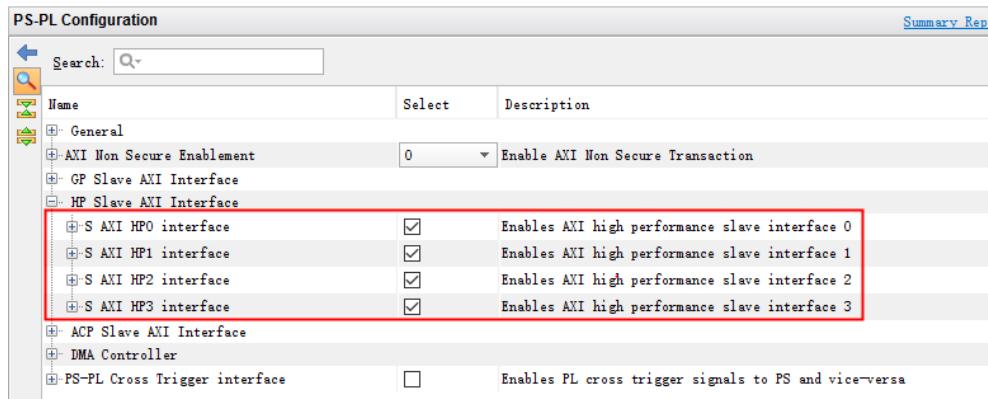
双击这个接口也要设置时钟频率，由于输出像素为 640X480 因此为 25000000HZ



8.3.6 AXI HP 通道和 DMA 中断

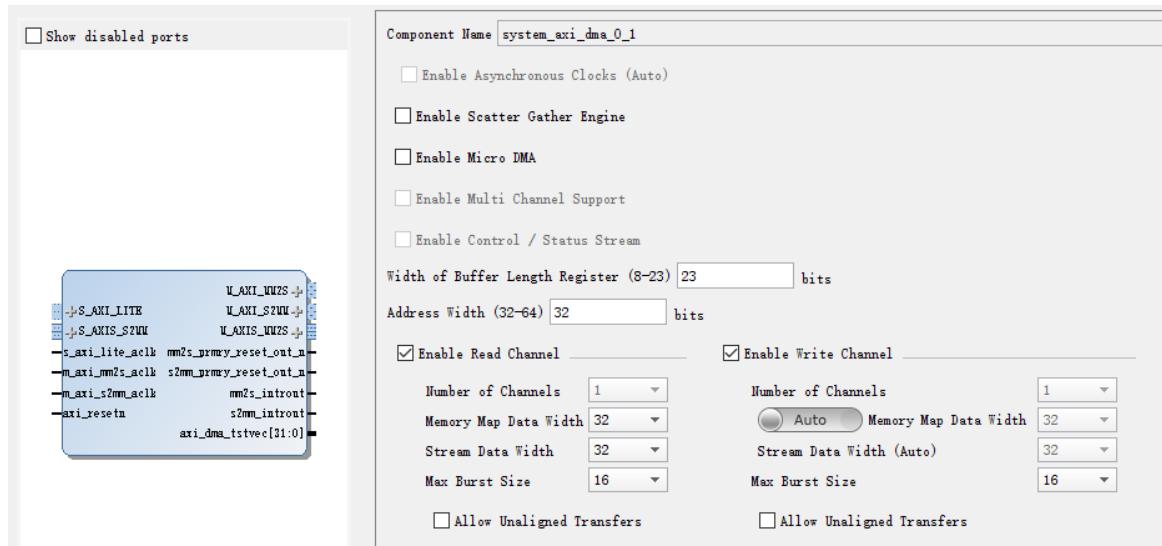
由于是四路视频输入，外接了 4 个 DMA 模块因此使用了 4 个 HP 和 8 个 DMA 中断如图





8.3.7 DMA IP 的设置

下图中，同时勾选读通道和写通道，另外设置，Width of buffer length register 为 23bit 这个含义是 2 的 23 次方 8,388,607bytes 8M 大小。一副 1080P 的图像大小为 $1920 \times 1080 / 1024 / 1024 * 4 = 7.9M$ 因此一次 DMA 就可以传输一副 1080P 的图像。



8.3.8 时钟管理模块

时钟管理模块前面已经讲过了，640X480 的分辨率是设置 25MHZ，不在具体累述。

8.3.9 VTC 图像时序发生模块

VTC 图像时序发生模块的使用只要配置对应的分辨率，这里是设置 640X480 的分辨率，前面章节已经讲过不再累述。

8.4 FPGA 四路输入以及图像切换源码分析

8.4.1 按钮输入去抖代码

表 8-4-1

```
always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0) begin
        button_reg0 <= 1'b0;
        button_reg1 <= 1'b0;
        button_reg2 <= 1'b0;
        button_reg3 <= 1'b0;
    end
    else begin
        button_reg0 <= button;
        button_reg1 <= button_reg0;
        button_reg2 <= button_reg1;
        button_reg3 <= button_reg2;
    end
end

assign button_en = button_reg0 & button_reg1 & ~button_reg2 & ~button_reg3;
```

好简洁的去抖动代码，信号延迟 4 个时钟，连续监测到 4 次，就是认为按下了。关键稳定吗，还真不够稳定，你试试就知道了。有时候按下去会跳过几幅图像。所以按下去的时候要果断点，手指不要抖。哈哈，关键是代码简洁，满足了实验要求。读者如果要做自己的演示系统，还是把去抖动代码写好一些。

8.4.2 DMA 4 路视频输入的 FPGA 代码

表 8-4-2-1

```
always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_0 <= 11'd0;
    else
        if(m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast)
            if(v_cnt_0 != 11'd479)
                v_cnt_0 <= v_cnt_0 + 1'b1;
            else
                v_cnt_0 <= 11'd0;
```

```

    else
        v_cnt_0 <= v_cnt_0;
end

```

上表可以看到 gpio_rtl_tri_o_0 就是可编程的复位信号，可以用 C 代码控制同步时序。上表的代码实现的是视频通路 0 的 vs 行计数器。可以看出来计数器在 m_axis_video_0_tvalid (vid in 输出数据有效)、 s_axis_dma_0_tready(DMA 通道准备好) 、 m_axis_video_0_tlast (vid_in 行结束信号)都有效的时候累加 1。这里的分辨率是 640X480 因此累计一共 480 行数据。由于使用了 4 个输入输入通道，因此 vs 行计数器的完成代码如下表。

表 8-4-2-2

```

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_0 <= 11'd0;
    else
        if(m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast)
            if(v_cnt_0 != 11'd479)
                v_cnt_0 <= v_cnt_0 + 1'b1;
            else
                v_cnt_0 <= 11'd0;
        else
            v_cnt_0 <= v_cnt_0;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_1 <= 11'd0;
    else
        if(m_axis_video_1_tvalid & s_axis_dma_1_tready & m_axis_video_1_tlast)
            if(v_cnt_1 != 11'd479)
                v_cnt_1 <= v_cnt_1 + 1'b1;
            else
                v_cnt_1 <= 11'd0;
        else
            v_cnt_1 <= v_cnt_1;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_2 <= 11'd0;
    else
        if(m_axis_video_2_tvalid & s_axis_dma_2_tready & m_axis_video_2_tlast)

```

```

if(v_cnt_2 != 11'd479)
    v_cnt_2 <= v_cnt_2 + 1'b1;
else
    v_cnt_2 <= 11'd0;
else
    v_cnt_2 <= v_cnt_2;
end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_3 <= 11'd0;
    else
        if(m_axis_video_3_tvalid & s_axis_dma_3_tready & m_axis_video_3_tlast)
            if(v_cnt_3 != 11'd479)
                v_cnt_3 <= v_cnt_3 + 1'b1;
            else
                v_cnt_3 <= 11'd0;
        else
            v_cnt_3 <= v_cnt_3;
end

```

下表是 s_axis_dma_0_tlast、s_axis_dma_1_tlast、s_axis_dma_2_tlast、s_axis_dma_3_tlast 代表每个通道一副图像传输完成后的 last 信号。这个信号为高电平 1 个周期，提交一次 DMA 数据到 DDR，并且会产生一次对应端口的中断信号。

表 8-4-2-4

```

assign s_axis_dma_0_tlast = m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast &(v_cnt_0 ==
11'd479);
assign s_axis_dma_1_tlast = m_axis_video_1_tvalid & s_axis_dma_1_tready & m_axis_video_1_tlast &(v_cnt_1 ==
11'd479);
assign s_axis_dma_2_tlast = m_axis_video_2_tvalid & s_axis_dma_2_tready & m_axis_video_2_tlast &(v_cnt_2 ==
11'd479);
assign s_axis_dma_3_tlast = m_axis_video_3_tvalid & s_axis_dma_3_tready & m_axis_video_3_tlast &(v_cnt_3 ==
11'd479);

```

8.4.3 DMA 输出通道

```
always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0)
        h_cnt <= 11'd0;
    else
        if(video_out_tvalid & video_out_tready)
            if(h_cnt != 11'd639)
                h_cnt <= h_cnt + 1'b1;
            else
                h_cnt <= 11'd0;
        else
            h_cnt <= h_cnt;
end
```

表 8-4-3-1

上表是 vid out ip 输入数据部分的列计数器，一共有 640 列。当 video_out_tvalid(FIFO 输出数据有效信号)和 video_out_tready(vid out IP 准备好接收数据信号)都为 1 的时候开始计数。

表 8-4-3-2

```
always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt <= 11'd0;
    else
        if(video_out_tvalid & video_out_tready & (h_cnt == 11'd639))
            if(v_cnt != 11'd479)
                v_cnt <= v_cnt + 1'b1;
            else
                v_cnt <= 11'd0;
        else
            v_cnt <= v_cnt;
end
```

上表是 vid out IP 输入数据的行计数器，当 video_out_tvalid (FIFO 数据输出有效) video_out_tready (vid out 准备好接收数据信号)和 h_cnt == 11'd639(代表一行数据结束)行计数器 v_cnt 加 1。

表 8-4-3-3

```
assign video_out_tdata = (channel_switch == 2'b00) ? FIFO_M_AXIS_0_tdata[23:0] :
    ((channel_switch == 2'b01) ? FIFO_M_AXIS_1_tdata[23:0] :
    ((channel_switch == 2'b10) ? FIFO_M_AXIS_2_tdata[23:0] :
    FIFO_M_AXIS_3_tdata[23:0]));
```

```

assign video_out_tvalid = video_en & ((channel_switch == 2'b00) ? FIFO_M_AXIS_0_tvalid :
                                         ((channel_switch == 2'b01) ? FIFO_M_AXIS_1_tvalid :
                                         ((channel_switch      ==      2'b10)      ?      FIFO_M_AXIS_2_tvalid      :
                                         FIFO_M_AXIS_3_tvalid)));

assign video_out_tuser = video_out_tvalid & video_out_tready & (h_cnt == 11'd0) & (v_cnt == 11'd0);

assign video_out_tlast = video_out_tvalid & video_out_tready & (h_cnt == 11'd639);

assign FIFO_M_AXIS_0_tready = video_en & video_out_tready;
assign FIFO_M_AXIS_1_tready = video_en & video_out_tready;
assign FIFO_M_AXIS_2_tready = video_en & video_out_tready;
assign FIFO_M_AXIS_3_tready = video_en & video_out_tready;

```

上表中，`video_out_tvalid` 是代表了 FIFO 输出的有效数据的信号，通过 `channel_switch` 切换到当前选定的 FIFO valid 信号上。

上表中，`video_out_tdata` 是代表了 FIFO 输出的数据通道，通过 `channel_switch` 切换到当前选定的 FIFO 数据通道。

上表中，`video_out_tuser` 是代表了 vid out 一帧图像开始信号。每行从 0 开始第一个数据。当 `video_out_tvalid`(FIFO 输出数据有效)、`video_out_tready`(vid out 可以接收数据信号)、`h_cnt==11'd0`(第一行第一个数据)、`v_cnt == 11'd0`(一帧图像的第 0 行)都满足条件 `video_out_tuser` 输出 1，告知 vid_out IP 一帧图像开始。

上表中，`video_out_tlast` 代表了 vid out 输入图像数据的一行结束，每一行结束都要输出 `video_out_tlast` 为 1.

8.5 4 路视频切换 DMA C 处理源码分析

8.5.1 main.c 源码

表 8-5-1 main.c

```

/*
 *
 * www.osrc.cn
 * www.milinker.com
 * copyright by nan jin mi lian dian zi www.osrc.cn
 * axi dma test
 *
 */

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

```

```
#include "xgpio.h"

volatile int TxDone0;
volatile int TxDone1;
volatile int TxDone2;
volatile int TxDone3;
volatile int RxDone0;
volatile int RxDone1;
volatile int RxDone2;
volatile int RxDone3;
volatile u8 tx0_buffer_index;
volatile u8 rx0_buffer_index;
volatile u8 tx1_buffer_index;
volatile u8 rx1_buffer_index;
volatile u8 tx2_buffer_index;
volatile u8 rx2_buffer_index;
volatile u8 tx3_buffer_index;
volatile u8 rx3_buffer_index;
volatile int Error;

u32 *BufferPtr0[3];
u32 *BufferPtr1[3];
u32 *BufferPtr2[3];
u32 *BufferPtr3[3];

XAxiDma AxiDma0;
XAxiDma AxiDma1;
XAxiDma AxiDma2;
XAxiDma AxiDma3;

/********************* Variable Definitions ********************/
static XScuGic Intc; //GIC
static XGpio Gpio;

#define AXI_GPIO_DEV_ID          XPAR_AXI_GPIO_0_DEVICE_ID

int init_intr_sys(void)
{
    // initial DMA interrupt handle
    DMA_Intr_Init(&AxiDma0,XPAR_AXIDMA_0_DEVICE_ID);
    DMA_Intr_Init(&AxiDma1,XPAR_AXIDMA_1_DEVICE_ID);
    DMA_Intr_Init(&AxiDma2,XPAR_AXIDMA_2_DEVICE_ID);
    DMA_Intr_Init(&AxiDma3,XPAR_AXIDMA_3_DEVICE_ID);
```

```
Init_Intr_System(&Intc); // initial DMA interrupt system
Setup_Intr_Exception(&Intc);

DMA_Setup_Intr_System(&Intc,&AxiDma0,TX0_INTR_ID,RX0_INTR_ID);//setup dma interrrpt system
DMA_Setup_Intr_System(&Intc,&AxiDma1,TX1_INTR_ID,RX1_INTR_ID);//setup dma interrrpt system
DMA_Setup_Intr_System(&Intc,&AxiDma2,TX2_INTR_ID,RX2_INTR_ID);//setup dma interrrpt system
DMA_Setup_Intr_System(&Intc,&AxiDma3,TX3_INTR_ID,RX3_INTR_ID);//setup dma interrrpt system

DMA_Intr_Enable(&Intc,&AxiDma0);
DMA_Intr_Enable(&Intc,&AxiDma1);
DMA_Intr_Enable(&Intc,&AxiDma2);
DMA_Intr_Enable(&Intc,&AxiDma3);
}

int main(void)
{
    int Status;

    XGpio_Initialize(&Gpio, AXI_GPIO_DEV_ID);
    XGpio_SetDataDirection(&Gpio, 1, 0);

    BufferPtr0[0] = (u32 *)CH0_BUFFER0_BASE;
    BufferPtr0[1] = (u32 *)CH0_BUFFER1_BASE;
    BufferPtr0[2] = (u32 *)CH0_BUFFER2_BASE;

    BufferPtr1[0] = (u32 *)CH1_BUFFER0_BASE;
    BufferPtr1[1] = (u32 *)CH1_BUFFER1_BASE;
    BufferPtr1[2] = (u32 *)CH1_BUFFER2_BASE;

    BufferPtr2[0] = (u32 *)CH2_BUFFER0_BASE;
    BufferPtr2[1] = (u32 *)CH2_BUFFER1_BASE;
    BufferPtr2[2] = (u32 *)CH2_BUFFER2_BASE;

    BufferPtr3[0] = (u32 *)CH3_BUFFER0_BASE;
    BufferPtr3[1] = (u32 *)CH3_BUFFER1_BASE;
    BufferPtr3[2] = (u32 *)CH3_BUFFER2_BASE;

    /* Initialize flags before start transfer test */
    TxDone0 = 0;
    TxDone1 = 0;
    TxDone2 = 0;
    TxDone3 = 0;
    RxDone0 = 0;
```

```
TxDone1 = 0;
TxDone2 = 0;
TxDone3 = 0;
tx0_buffer_index = 0;
rx0_buffer_index = 0;
tx1_buffer_index = 0;
rx1_buffer_index = 0;
tx2_buffer_index = 0;
rx2_buffer_index = 0;
tx3_buffer_index = 0;
rx3_buffer_index = 0;
Error = 0;

init_intr_sys();
Miz702_EMIO_init();
ov7725_init_rgb();

XGpio_DiscreteWrite(&Gpio, 1, 1);

Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[tx0_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma0 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[tx1_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma1 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[tx2_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma2 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[tx3_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma3 failed! %d\r\n", Status);
```

```

        return XST_FAILURE;
    }

Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[rx0_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma0 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[rx1_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("rx axi dma1 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[rx2_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma2 failed! %d\r\n", Status);
    return XST_FAILURE;
}

Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[rx3_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma3 failed! %d\r\n", Status);
    return XST_FAILURE;
}

while (1) {
    return XST_SUCCESS;
}

```

上表中的代码我们很熟悉了，这里是注册了 4 个 DMA 通道，8 个中断(接收和发送 4 路)。

8.5.2 dma_intr.h 源码

表 8-5-2 dma_intr.h

```

/*
 *
 * www.osrc.cn

```

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

/**************** Constant Definitions *****/
/*
 * Device hardware build related constants.
 */

#define RX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR

#define IMAGE_WIDTH      640
#define IMAGE_HEIGHT     480
#define BYTES_PER_PIXEL  4
#define MAX_BUFFER_NUM   8

#define MEM_BASE_ADDR    0x10000000

#define DMA0_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID
#define DMA1_DEV_ID      XPAR_AXIDMA_1_DEVICE_ID
#define DMA2_DEV_ID      XPAR_AXIDMA_2_DEVICE_ID
#define DMA3_DEV_ID      XPAR_AXIDMA_3_DEVICE_ID

#define RX0_INTR_ID      XPAR_FABRIC_AXI_DMA_0_S2MM_INTROUT_INTR
#define TX0_INTR_ID      XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR
#define RX1_INTR_ID      XPAR_FABRIC_AXI_DMA_1_S2MM_INTROUT_INTR
#define TX1_INTR_ID      XPAR_FABRIC_AXI_DMA_1_MM2S_INTROUT_INTR
#define RX2_INTR_ID      XPAR_FABRIC_AXI_DMA_2_S2MM_INTROUT_INTR
#define TX2_INTR_ID      XPAR_FABRIC_AXI_DMA_2_MM2S_INTROUT_INTR
#define RX3_INTR_ID      XPAR_FABRIC_AXI_DMA_3_S2MM_INTROUT_INTR
#define TX3_INTR_ID      XPAR_FABRIC_AXI_DMA_3_MM2S_INTROUT_INTR

#define CH0_BUFFER0_BASE      (MEM_BASE_ADDR)
#define CH0_BUFFER1_BASE      (CH0_BUFFER0_BASE +      IMAGE_WIDTH * IMAGE_HEIGHT *
                                BYTES_PER_PIXEL)
```

```
#define CH0_BUFFER2_BASE      (CH0_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *  
BYTES_PER_PIXEL)  
  
#define CH1_BUFFER0_BASE      (CH0_BUFFER0_BASE + MAX_BUFFER_NUM * IMAGE_WIDTH *  
IMAGE_HEIGHT * BYTES_PER_PIXEL)  
#define CH1_BUFFER1_BASE      (CH1_BUFFER0_BASE + IMAGE_WIDTH * IMAGE_HEIGHT *  
BYTES_PER_PIXEL)  
#define CH1_BUFFER2_BASE      (CH1_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *  
BYTES_PER_PIXEL)  
  
#define CH2_BUFFER0_BASE      (CH1_BUFFER0_BASE + MAX_BUFFER_NUM * IMAGE_WIDTH *  
IMAGE_HEIGHT * BYTES_PER_PIXEL)  
#define CH2_BUFFER1_BASE      (CH2_BUFFER0_BASE + IMAGE_WIDTH * IMAGE_HEIGHT *  
BYTES_PER_PIXEL)  
#define CH2_BUFFER2_BASE      (CH2_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *  
BYTES_PER_PIXEL)  
  
#define CH3_BUFFER0_BASE      (CH2_BUFFER0_BASE + MAX_BUFFER_NUM * IMAGE_WIDTH *  
IMAGE_HEIGHT * BYTES_PER_PIXEL)  
#define CH3_BUFFER1_BASE      (CH3_BUFFER0_BASE + IMAGE_WIDTH * IMAGE_HEIGHT *  
BYTES_PER_PIXEL)  
#define CH3_BUFFER2_BASE      (CH3_BUFFER0_BASE + 2 * IMAGE_WIDTH * IMAGE_HEIGHT *  
BYTES_PER_PIXEL)  
  
/* Timeout loop counter for reset  
 */  
#define RESET_TIMEOUT_COUNTER    10000  
/* test start value  
 */  
#define TEST_START_VALUE    0xC  
/*  
 * Buffer and Buffer Descriptor related constant definition  
 */  
#define MAX_PKT_LEN        (IMAGE_WIDTH * IMAGE_HEIGHT * BYTES_PER_PIXEL)  
/*  
 * transfer times  
 */  
#define NUMBER_OF_TRANSFERS 100000  
  
extern volatile int TxDone0;  
extern volatile int TxDone1;  
extern volatile int TxDone2;  
extern volatile int TxDone3;
```

```

extern volatile int RxDone0;
extern volatile int RxDone1;
extern volatile int RxDone2;
extern volatile int RxDone3;
extern volatile u8 tx0_buffer_index;
extern volatile u8 rx0_buffer_index;
extern volatile u8 tx1_buffer_index;
extern volatile u8 rx1_buffer_index;
extern volatile u8 tx2_buffer_index;
extern volatile u8 rx2_buffer_index;
extern volatile u8 tx3_buffer_index;
extern volatile u8 rx3_buffer_index;
extern volatile int Error;

extern u32 *BufferPtr0[3];
extern u32 *BufferPtr1[3];
extern u32 *BufferPtr2[3];
extern u32 *BufferPtr3[3];

extern XAxiDma AxiDma0;
extern XAxiDma AxiDma1;
extern XAxiDma AxiDma2;
extern XAxiDma AxiDma3;
int DMA_CheckData(int Length, u8 StartValue);
int DMA_Setup_Intr_System(XScuGic * IntcInstancePtr,XAxiDma * AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
int DMA_Intr_Enable(XScuGic * IntcInstancePtr,XAxiDma *DMAPtr);
int DMA_Intr_Init(XAxiDma *DMAPtr,u32 DeviceId);
#endif

```

上表中主要定义 DMA 用到的变量，每个 DMA 通道的地址分配，DMA 通道对象的定义，以及 DMA 中断函数、DMA 中断使能函数。

8.5.3 dma_intr.c 中断接收源码

表 8-5-3 DMA_RxIntrHandler 源码

```

/*****************************************/
/*
*
* This is the DMA RX interrupt handler function
*
* It gets the interrupt status from the hardware, acknowledges it, and if any
* error happens, it resets the hardware. Otherwise, if a completion interrupt
* is present, then it sets the RxDone flag.

```

```
/*
 * @param Callback is a pointer to RX channel of the DMA engine.
 *
 * @return None.
 *
 * @note      None.
 *
 *****/
static void DMA_RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int Status;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

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

    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
        xil_printf("no interrupt! \r\n");
        return;
    }

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

        // Error = 1;

        xil_printf("rx error! \r\n");

        /* Reset could fail and hang
         * NEED a way to handle this or do not call it??
         */
        XAxiDma_Reset(AxiDmaInst);
    }
}
```

```
// TimeOut = RESET_TIMEOUT_COUNTER;

// while (TimeOut) {
//     if(XAxiDma_ResetIsDone(AxiDmaInst)) {
//         break;
//     }

//     TimeOut -= 1;
// }

return;
}

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

    if(AxiDmaInst == &AxiDma0)
    {
        RxDone0++;
        if(rx0_buffer_index == 2)
            rx0_buffer_index = 0;
        else
            rx0_buffer_index++;

        Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[rx0_buffer_index],
                                         MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

        if (Status != XST_SUCCESS) {
            xil_printf("rx axi dma0 failed! 0 %d\r\n", Status);
            return;
        }
    }

    else if(AxiDmaInst == &AxiDma1)
    {
        RxDone1++;
        if(rx1_buffer_index == 2)
            rx1_buffer_index = 0;
        else
            rx1_buffer_index++;

        Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[rx1_buffer_index],
```

```
    MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        xil_printf("rx axi dma1 failed! 0 %d\r\n", Status);
        return;
    }
}

else if(AxiDmaInst == &AxiDma2)
{
    RxDone2++;
    if(rx2_buffer_index == 2)
        rx2_buffer_index = 0;
    else
        rx2_buffer_index++;

    Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[rx2_buffer_index],
                                    MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        xil_printf("rx axi dma2 failed! 0 %d\r\n", Status);
        return;
    }
}

else if(AxiDmaInst == &AxiDma3)
{
    RxDone3++;
    if(rx3_buffer_index == 2)
        rx3_buffer_index = 0;
    else
        rx3_buffer_index++;

    Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[rx3_buffer_index],
                                    MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        xil_printf("rx axi dma3 failed! 0 %d\r\n", Status);
        return;
    }
}

else
    xil_printf("error!\r\n");
}

}
```

上表中和单独 DMA 视频的却别就是通过 AxiDmaInst 判断当前 DMA 输入的通路，来确定当前输入当道下一次接收的数据需要保存到的 BUFFER 地址。

表 8-5-4-3 dma_intr.c 源码

8.5.4 dma_intr.c 中断发送源码

表 8-5-4-1 DMA_TxIntrHandler 函数源码

```
/****************************************************************************
 * This is the DMA TX Interrupt handler function.
 *
 * It gets the interrupt status from the hardware, acknowledges it, and if any
 * error happens, it resets the hardware. Otherwise, if a completion interrupt
 * is present, then sets the TxDone.flag
 *
 * @param Callback is a pointer to TX channel of the DMA engine.
 *
 * @return None.
 *
 * @note      None.
 *
 ****/
static void DMA_TxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int Status;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

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

    /* Acknowledge pending interrupts */

    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
        xil_printf("no interrupt! \r\n");
        return;
    }
}
```

```
}

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

    //Error = 1;
    xil_printf("tx error! \r\n");

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

    /*
     * TimeOut = RESET_TIMEOUT_COUNTER;
     */

    while (TimeOut) {
        if (XAxiDma_ResetIsDone(AxiDmaInst)) {
            break;
        }
    }

    TimeOut -= 1;
}

return;
}

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

    if(AxiDmaInst == &AxiDma0)
    {
        TxDone0++;
        if(rx0_buffer_index == 0)
            tx0_buffer_index = 2;
        else
            tx0_buffer_index = rx0_buffer_index - 1;

        Status = XAxiDma_SimpleTransfer(&AxiDma0, (u32)BufferPtr0[tx0_buffer_index],

```

```
MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma0 failed! 0 %d\r\n", Status);
    return;
}

else if(AxiDmaInst == &AxiDma1)
{
    TxDone1++;
    if(rx1_buffer_index == 0)
        tx1_buffer_index = 2;
    else
        tx1_buffer_index = rx1_buffer_index - 1;

    Status = XAxiDma_SimpleTransfer(&AxiDma1, (u32)BufferPtr1[tx1_buffer_index],
                                    MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

    if (Status != XST_SUCCESS) {
        xil_printf("tx axi dma1 failed! 0 %d\r\n", Status);
        return;
    }
}

else if(AxiDmaInst == &AxiDma2)
{
    TxDone2++;
    if(rx2_buffer_index == 0)
        tx2_buffer_index = 2;
    else
        tx2_buffer_index = rx2_buffer_index - 1;

    Status = XAxiDma_SimpleTransfer(&AxiDma2, (u32)BufferPtr2[tx2_buffer_index],
                                    MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

    if (Status != XST_SUCCESS) {
        xil_printf("tx axi dma2 failed! 0 %d\r\n", Status);
        return;
    }
}

else if(AxiDmaInst == &AxiDma3)
{
    TxDone3++;
    if(rx3_buffer_index == 0)
```

```
tx3_buffer_index = 2;
else
    tx3_buffer_index = rx3_buffer_index - 1;

Status = XAxiDma_SimpleTransfer(&AxiDma3, (u32)BufferPtr3[tx3_buffer_index],
                                MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

if (Status != XST_SUCCESS) {
    xil_printf("tx axi dma3 failed! 0 %d\r\n", Status);
    return;
}
}
else
xil_printf("error!\r\n");
}

}
```

上表的发送中断函数，和接收中断函数处理机制一致，也是通过 AxiDmaInst 判断当前 DMA 的通道，并且为当前 DMA 通道发送数据，指定对应的 BUFFER。

8.6 测试结果



切换 1



切换 2



切换 3



切换 4

8.7 本章小结

本章给出的是一个实用化的 DMA 应用方案，设计了 4 路视频通过 DMA 输入到 DDR。在 C 代码中实现 3 缓存输出。输出的时候，提供切换 FIFO 的通道，实现把其中一路输出到显示器。本方案的应用场景可以用于电视广播系统、视频会议等。

CH09_DMA_2_Video_Stitch 视频拼接系统

9.1 概述

注意：本课程和上一课程《S03_CH08_DMA_4_Video_Switch 视频切换系统》基本相同，不同部分用紫色字体或者框图注明。

PL 通过 OV7725 实时采集 2 路 640×480 视频。2 路视频通过 2 个独立的 AXI DMA IP 核传输至 PS 的 DDR 中进行缓存，然后再通过 AXIDMA 将 2 路视频同时从 DDR 读出，通过 PL 在 VGA 显示器上以 1080P 分辨率同时显示 2 路原始分辨率拼接而成的视频。

9.2 修改 OV_Sensor_ML 摄像头采集 IP

由于外接字卡后，MiZ7035 开发板有 2 路摄像头视频输入接口，可以进行双路摄像头拼接。

表 9-2-1：

```
module OV_Sensor_ML(
    input CLK_i,
    //----- CMOS sensor hardware interface -----
    input cmos_vsync_i, //cmos vsync
    input cmos_href_i, //cmos hsync refrence
    input cmos_pclk_i, //cmos pxiel clock
    output cmos_xclk_o, //cmos externl clock
    input[7:0] cmos_data_i, //cmos data
    output hs_o,//hs signal.
    output vs_o,//vs signal.
    // output de_o,//data enable.
    output [23:0] rgb_o,//data output,
    output vid_clk_ce
);
//-----视频输出解码模块-----
wire [15:0]rgb_o_r;
//assign rgb_o = {rgb_o_r[4:0],3'd0,rgb_o_r[10:5],2'd0,rgb_o_r[15:11],3'd0};
assign rgb_o = {rgb_o_r[15:11],3'd0,rgb_o_r[10:5],2'd0,rgb_o_r[4:0],3'd0};
reg [7:0]cmos_data_r;
reg cmos_href_r;
reg cmos_vsync_r;

always@(posedge cmos_pclk_i)
begin
    cmos_data_r <= cmos_data_i;
    cmos_href_r <= cmos_href_i;
    cmos_vsync_r <= cmos_vsync_i;
```

```
end

//assign rgb_o = 24'b1111111_0000000_1111111;

cmos_decode cmos_decode_u0(
    //system signal.
    .cmos_clk_i(CLK_i),//cmos senseor clock.
    .rst_n_i(RESETn_i2c),//system reset.active low.
    //cmos sensor hardware interface.
    .cmos_pclk_i(cmos_pclk_i),//(cmos_pclk),//input pixel clock.
    .cmos_href_i(cmos_href_r),//(cmos_href),//input pixel hs signal.
    .cmos_vsync_i(cmos_vsync_r),//(cmos_vsync),//input pixel vs signal.
    .cmos_data_i(cmos_data_r),//(cmos_data),//data.
    .cmos_xclk_o(cmos_xclk_o),//(cmos_xclk),//output clock to cmos sensor.
    //user interface.
    .hs_o(hs_o),//hs signal.
    .vs_o(vs_o),//vs signal.
    //.
    .de_o(de_o),//data enable.
    .rgb565_o(rgb_o_r),//data output
    .vid_clk_ce(vid_clk_ce)
);

count_reset_v1#(
    .num(20'hffff0)
)(

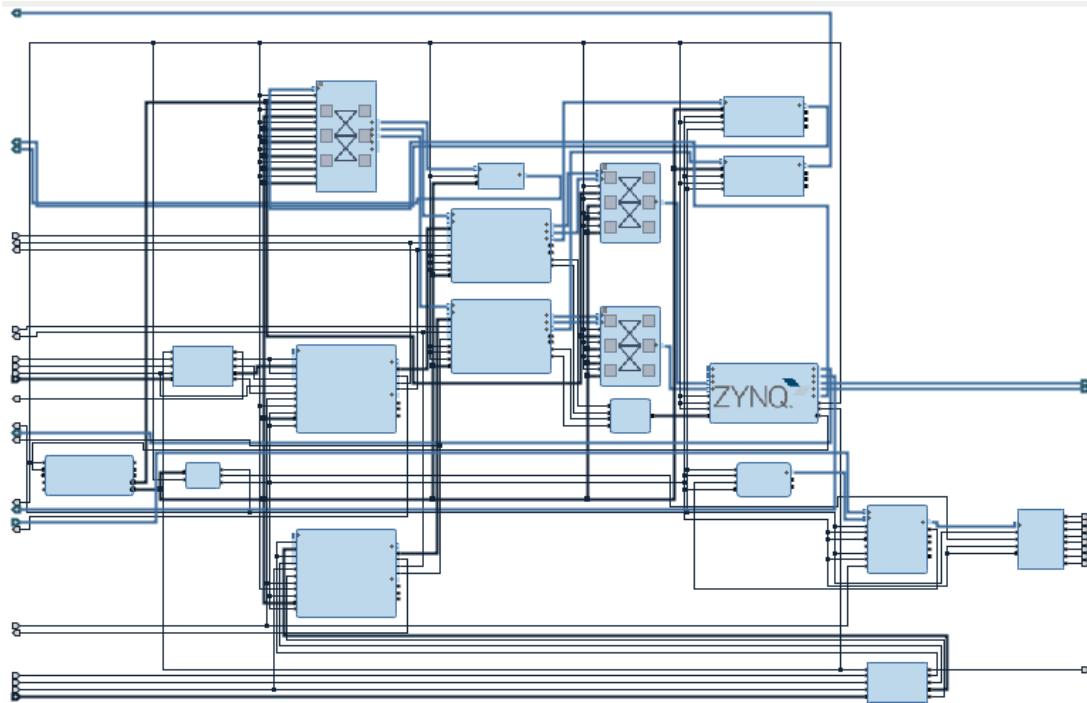
    .clk_i(CLK_i),
    .rst_o(RESETn_i2c)
);

endmodule
```

9.3 搭建硬件系统

9.3.1 系统图

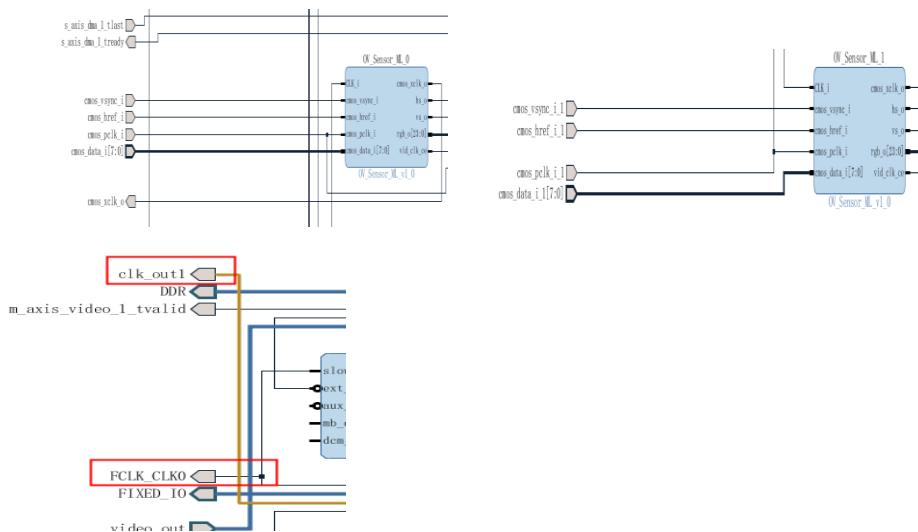
完成了 IP 的修改后，下面就可以搭建硬件系统了，由于 VIVADO 采用了图形化设计，带来了很大便捷。下面把系统构架图贴出来。



由于图片太大，只能看到大概的框架，大家学习的时候可以打开工程放大后去阅读，这里为了分析的时候方便把局部视图放大截图。

9.3.2 OV_Sensor_ML IP 接线图

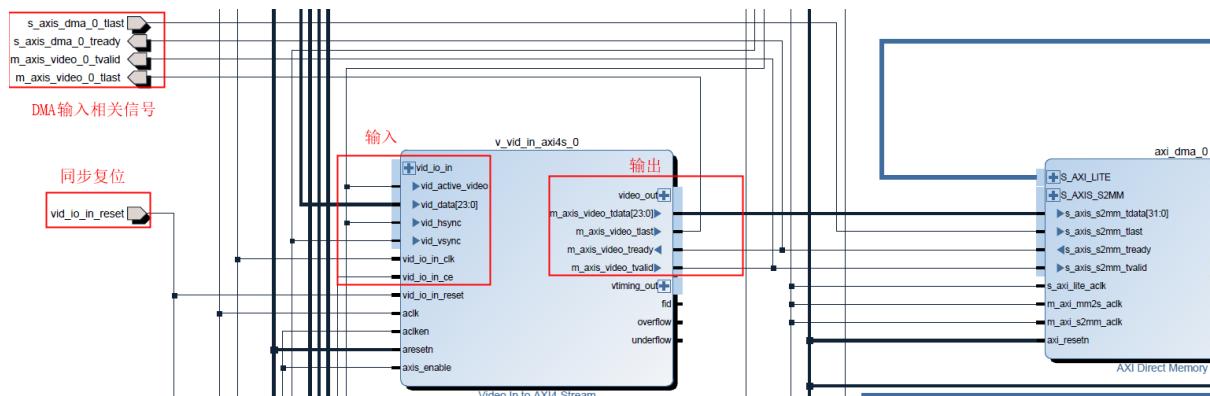
下图中是前面我们自定义 OV_Sensor_ML 采集 IP 图形界面。下图中，还有 2 个信号分别是 FCLK_CLK0 和 clk_out1 他们分别是 VID_IN IP 和 VID_OUT IP 相关的时钟，把它们引出去到顶层模块中，后面需要使用到。



9.3.3 vid_in IP 的接线图

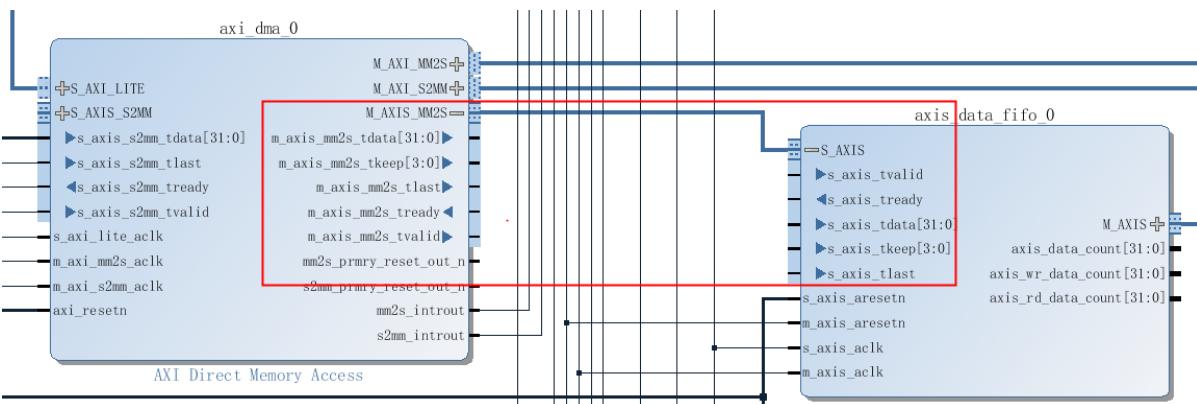
下图大家可以放大后观看，vid_in IP 的输入接口是连接到摄像头采样输出 IP 的。vid_in IP 的输

出接口是和 DMA 链接了。DMA 输入相关的信号被引出到外部,用来添加 FPGA 代码实现写 DMA 时序。还有一个 vid_io_reset 信号,是用来控制所有 vid_in 和 vid_out IP 的同步,也是连接到外部,用 FPGA 代码控制。

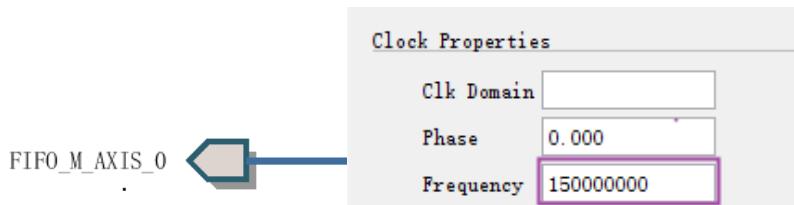


9.3.4 DMA 和 FIFO 通路

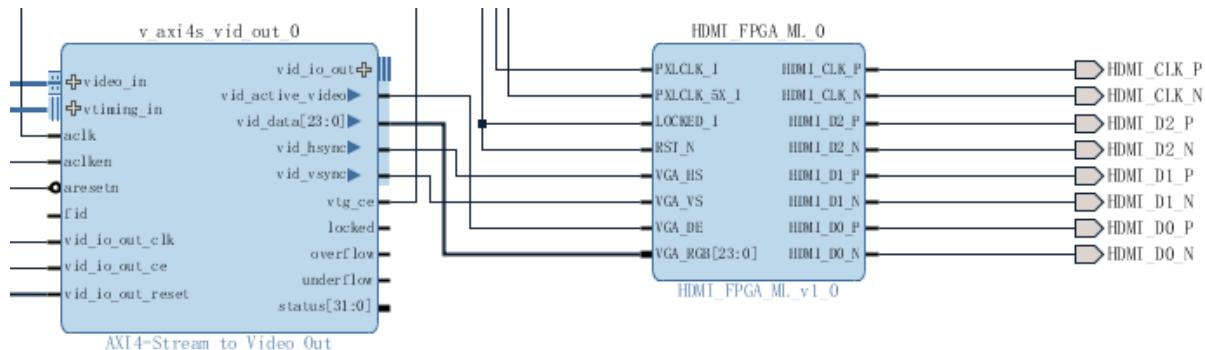
下图是 DMA 和 data fifo 的链接通路。



上图中 data fifo 的 M_AXIS 将被引出到外部受 FPGA 代码控制。如下图, FIFO_M_AXIS_0 就是连接到 axis_data_fifo_0 的 M_AXIS 接口的。双击此接口需要设置时钟,这里的数据速度时钟是 148.5MHZ。不同的分辨率应当设置对应的分辨率时钟。

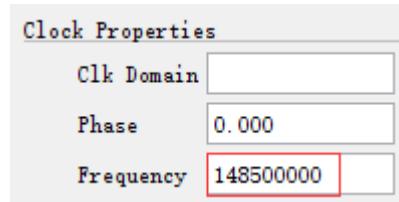


9.3.5 vid_out IP 的通路



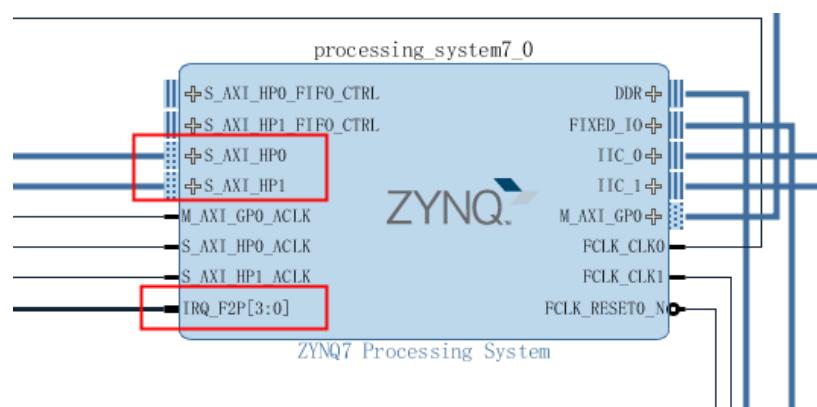
上图中的 vid out IP 数据输入通道如图所示

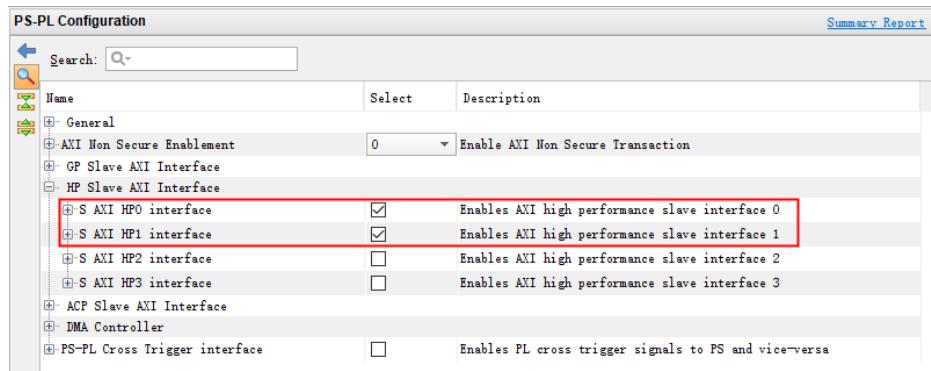
双击这个接口也要设置时钟频率，由于输出像素为 1920X108060HZ 因此为 148500000HZ



9.3.6 AXI HP 通道和 DMA 中断

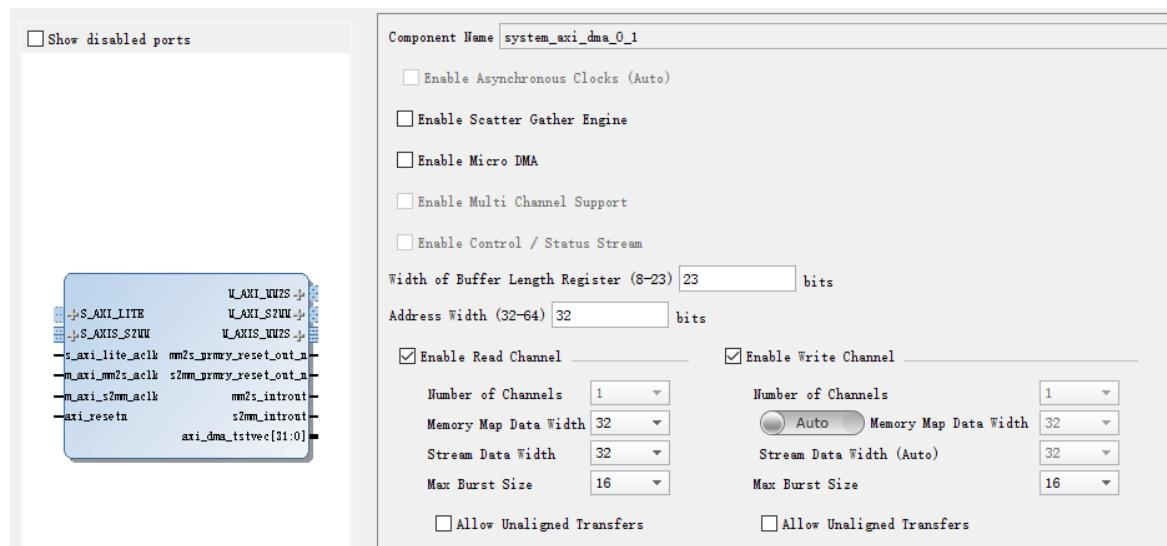
由于是四路视频输入，外接了 2 个 DMA 模块因此使用了 4 个 HP 和 8 个 DMA 中断如图





9.3.7 DMA IP 的设置

下图中，同时勾选读通道和写通道，另外设置，Width of buffer length register 为 23bit 这个含义是 2 的 23 次方 8,388,607bytes 8M 大小。一副 1080P 的图像大小为 $1920 \times 1080 / 1024 / 1024 * 4 = 7.9M$ 因此一次 DMA 就可以传输一副 1080P 的图像。



9.3.7 时钟管理模块

时钟管理模块前面已经讲过了， 1920×1080 的分辨率是设置 148.5MHz，不在具体累述。

9.3.8 VTC 图像时序发生模块

VTC 图像时序发生模块的使用只要配置对应的分辨率，这里是设置 640X480 的分辨率，前面章节已经讲过不再累述。

9.4 FPGA 2 路输入以及图像拼接源码分析

9.4.1 图像常量参数

表 9-4-1

localparam	MONITOR_HEIGHT = 11'd1080;//设置输出行分辨率
localparam	MONITOR_WIDTH = 11'd1920;//设置输出列分辨率
localparam	VIDEO_HEIGHT = 11'd480;//设置输入视频行分辨率
localparam	VIDEO_WIDTH = 11'd640;//设置输入视频列分辨率
localparam	GAP_HEIGHT = 11'd100;//设置四副图形中间的空白尺寸高度
localparam	GAP_WIDTH = 11'd100;//设置四副图形中间的空白尺寸宽度
localparam	BLACK_HEIGHT = (MONITOR_HEIGHT - 2 * VIDEO_HEIGHT - GAP_HEIGHT) >> 1;//上下边框高度 BLACK_HEIGHT = 10
localparam	BLACK_WIDTH = (MONITOR_WIDTH - 2 * VIDEO_WIDTH - GAP_WIDTH) >> 1;//左右边框宽度 BLACK_WIDTH = 270
localparam	CH01_V_START = BLACK_HEIGHT;//第一路图像垂直开始 CH01_V_START = 10
localparam	CH01_V_END = BLACK_HEIGHT + VIDEO_HEIGHT;//第一路图像垂直结束 CH01_V_END=490
localparam	CH23_V_START = BLACK_HEIGHT + VIDEO_HEIGHT + GAP_HEIGHT;//第二路图像垂直开始 CH23_V_START=590
localparam	CH23_V_END = BLACK_HEIGHT + VIDEO_HEIGHT + GAP_HEIGHT + VIDEO_HEIGHT;//第二路图像垂直结束 CH23_V_END= 1070
localparam	CH02_H_START = BLACK_WIDTH;//第一路图像水平开始 CH02_H_START=270
localparam	CH02_H_END = BLACK_WIDTH + VIDEO_WIDTH;//第一路图像水平结束CH02_H_END=910
localparam	CH13_H_START = BLACK_WIDTH + VIDEO_WIDTH + GAP_WIDTH;//第二路图像水平开始1010
localparam	CH13_H_END = BLACK_WIDTH + VIDEO_WIDTH + GAP_WIDTH + VIDEO_WIDTH;第二路图像水平结束//1550

以上代码是对图像的输出分辨率，被拼接图像的分辨率、空白、边框、背景进行设置。

9.4.2 DMA 2 路视频输入的 FPGA 代码

```
always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_0 <= 11'd0;
    else
        if(m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast)
            if(v_cnt_0 != 11'd479)
                v_cnt_0 <= v_cnt_0 + 1'b1;
            else
                v_cnt_0 <= 11'd0;
        else
            v_cnt_0 <= v_cnt_0;
    end

```

表 9-4-2-1

上表可以看到 gpio_rtl_tri_o_0 就是可编程的复位信号，可以用 C 代码控制同步时序。上表的代码实现的是视频通路 0 的 vs 行计数器。可以看出来计数器在 m_axis_video_0_tvalid (vid in 输出数据有效)、s_axis_dma_0_tready(DMA 通道准备好)、m_axis_video_0_tlast (vid_in 行结束信号)都有效的时候累加 1。这里的分辨率是 640X480 因此累计一共 480 行数据。由于使用了 2 个输入输入通道，因此 vs 行计数器的完成代码如下表。

表 9-4-2-2

```
always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_0 <= 11'd0;
    else
        if(m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast)
            if(v_cnt_0 != 11'd479)
                v_cnt_0 <= v_cnt_0 + 1'b1;
            else
                v_cnt_0 <= 11'd0;
        else
            v_cnt_0 <= v_cnt_0;
    end

always@(posedge FCLK_CLK0)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt_1 <= 11'd0;
    else

```

```

if(m_axis_video_1_tvalid & s_axis_dma_1_tready & m_axis_video_1_tlast)
    if(v_cnt_1 != 11'd479)
        v_cnt_1 <= v_cnt_1 + 1'b1;
    else
        v_cnt_1 <= 11'd0;
    else
        v_cnt_1 <= v_cnt_1;
end

```

下表是 `s_axis_dma_0_tlast`、`s_axis_dma_1_tlast`、`s_axis_dma_2_tlast`、`s_axis_dma_3_tlast` 代表每个通道一副图像传输完成后的 last 信号。这个信号为高电平 1 个周期，提交一次 DMA 数据到 DDR，并且会产生一次对应端口的中断信号。

表 9-4-2-3

```

assign s_axis_dma_0_tlast = m_axis_video_0_tvalid & s_axis_dma_0_tready & m_axis_video_0_tlast &(v_cnt_0 ==
11'd479);
assign s_axis_dma_1_tlast = m_axis_video_1_tvalid & s_axis_dma_1_tready & m_axis_video_1_tlast &(v_cnt_1 ==
11'd479);

```

9.4.3 DMA 输出通道

表 9-4-3-1

```

always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0)
        h_cnt <= 11'd0;
    else
        if(video_out_tvalid & video_out_tready)
            if(h_cnt != (MONITOR_WIDTH - 1'b1))
                h_cnt <= h_cnt + 1'b1;
            else
                h_cnt <= 11'd0;
        else
            h_cnt <= h_cnt;
end

```

上表是 vid out ip 输入数据部分的列计数器，一共有 1920 列。当 `video_out_tvalid`(FIFO 输出数据有效信号)和 `video_out_tready`(vid out IP 准备好接收数据信号)都为 1 的时候开始计数。

表 9-4-3-2

```

always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0)
        v_cnt <= 11'd0;

```

```

else
    if(video_out_tvalid & video_out_tready & (h_cnt == (MONITOR_WIDTH - 1'b1)))
        if(v_cnt != (MONITOR_HEIGHT - 1'b1))
            v_cnt <= v_cnt + 1'b1;
        else
            v_cnt <= 11'd0;
    else
        v_cnt <= v_cnt;
end

```

上表是 vid out IP 输入数据的行计数器，当 video_out_tvalid (FIFO 数据输出有效) video_out_tready (vid out 准备好接收数据信号)和 h_cnt == 11'd1919 共计 1920 点(代表一行数据结束) 行计数器 v_cnt 加 1。

```

always@(posedge clk_out1)
begin
    if(!gpio_rtl_tri_o_0)
        channel_switch <= 2'd2;
    else
        if(v_cnt < VIDEO_HEIGHT)
            begin
                if(h_cnt < VIDEO_WIDTH)
                    channel_switch <= 2'd0;
                else
                    channel_switch <= 2'd1;
            end
        else
            channel_switch <= 2'd2;
    end
end

```

表 9-4-3-3

上表代码实现了视频在显示器上的拼接输出，有点类似前面的四路切换方案，区别是这次是把所有视频全部输出到 1080P 的显示器上了。

表 9-4-3

```

assign video_out_tdata = (channel_switch == 2'd0) ? FIFO_M_AXIS_0_tdata :
                           ((channel_switch == 2'd1) ? FIFO_M_AXIS_1_tdata : 24'd0);

assign video_out_tvalid = (channel_switch == 2'd0) ? FIFO_M_AXIS_0_tvalid :
                           ((channel_switch == 2'd1) ? FIFO_M_AXIS_1_tvalid : 1'b1);

assign video_out_tuser = video_out_tvalid & video_out_tready & (h_cnt == 11'd0) & (v_cnt == 11'd0);

assign video_out_tlast = (h_cnt == (MONITOR_WIDTH - 1'b1)) ? 1'b1 : 1'b0;

```

```
assign FIFO_M_AXIS_0_tready = (channel_switch == 2'd0) ? video_out_tready : 1'b0;  
assign FIFO_M_AXIS_1_tready = (channel_switch == 2'd1) ? video_out_tready : 1'b0;
```

上表中，`video_out_tvalid` 是代表了 FIFO 输出的有效数据的信号，通过 `channel_switch` 切换到当前选定的 FIFO valid 信号上。

上表中，`video_out_tdata` 是代表了 FIFO 输出的数据通道，通过 `channel_switch` 切换到当前选定的 FIFO 数据通道。

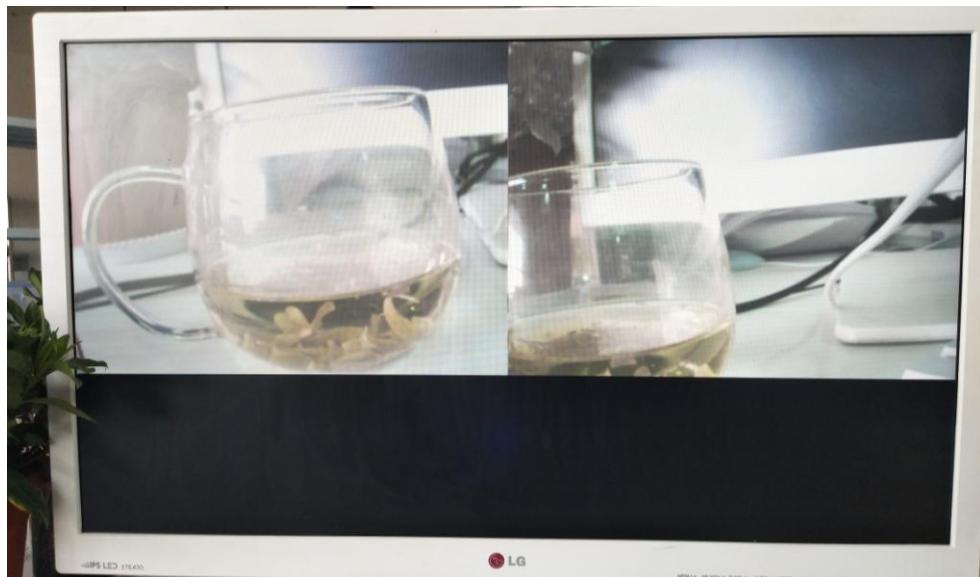
上表中，`video_out_tuser` 是代表了 vid out 一帧图像开始信号。每行从 0 开始第一个数据。当 `video_out_tvalid`(FIFO 输出数据有效)、`video_out_tready`(vid out 可以接收数据信号)、`h_cnt==11'd0`(第一行第一个数据)、`v_cnt ==11'd0`(一帧图像的第 0 行)都满足条件 `video_out_tuser` 输出 1, 告知 `vid_out` IP 一帧图像开始。

上表中，`video_out_tlast` 代表了 vid out 输入图像数据的最后一行最后一个数据，这里是 1920X1080 的图像，因此到 1919，每一行最后一个数据都要输出 `video_out_tlast` 为 1.

9.5 2 路视频切换 DMA C 处理源码分析

见例程源码

9.6 测试结果



CH10_基于 TCP 的 QSPI Flash bin 文件网络烧写

10.1 概述

针对 ZYNQ 中使用 QSPI BOOT 的应用，将 BOOT.bin 文件烧写至 QSPI Flash 基本都是通过 USB Cable 连接 PC，由 JTAG 口连接板卡后，在 SDK 软件中使用“Program Flash”功能进行现场在线烧写。然而，这种常规方法存在两个缺点。

速度慢。Flash 的擦除（Erase）、写入（Program）、校验（Verify）3 个过程所费的时间总和通常都需要若干分钟。

无法脱离 JTAG 口。对于某些产品而言，当产品量产上市后进入维护升级阶段，若需修改、更新 Flash 中的 bin 文件，则需对产品进行拆解才可进行。

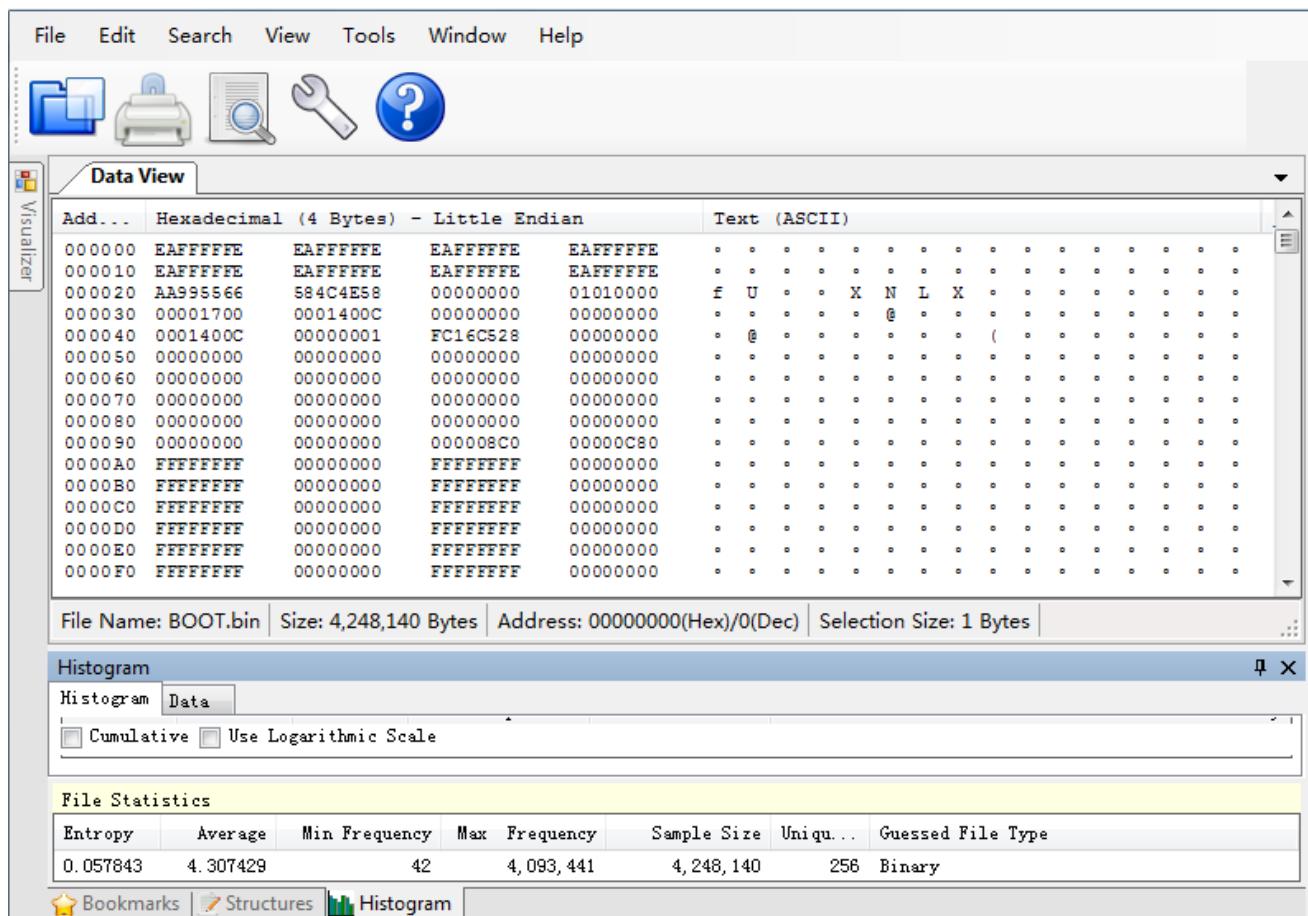
本例程实现了一种基于 TCP 协议的 Flash bin 文件更新方法。一方面，在较大程度上提高了 bin 文件的烧写速度（4MB 大小 bin 文件可缩短至 20 秒左右）；另一方面，提供了一种远程更新 Flash 的方法，可脱离 JTAG 接口，同时避免固件升级时对产品进行拆解。

10.2 基本原理

首先，在 ZYNQ 的 ARM 中基于 LWIP 库建立一个 TCP Server，板卡通过网线与电脑连接。在电脑中通过网络调试助手以 TCP Client 模式与 ZYNQ 中的 TCP Server 建立 TCP 连接。然后，通过网络调试助手将 BOOT.bin 文件以二进制形式发送至 TCP Server，并存储在 ZYNQ 所连接的 DDR 中。最后，当 TCP Server 接收完 bin 文件所有的数据后，网络调试助手发送烧写启动命令，将 bin 文件的数据按顺序一一连续写入 QSPI Flash 中，随后再全部读出与所接收的 bin 文件进行一一比对检验。断电重启板卡，便可验证 bin 文件的更新。

10.3 Bin 文件

在 SDK 中所生成的 BOOT.bin 文件为普通二进制文件，通过 UltraEdit 或 Binary Viewer 软件可打开并查看 bin 文件的内容。将 bin 文件中所有的数据按顺序一一连续写入 QSPI Flash 中，即可完成 bin 文件的烧写，也就是说 Flash 中的数据与 bin 文件中的数据完全是一一对应的。由此可见，烧写 bin 文件的原理并不复杂，不存在类似编码、解码的过程，且与 SDK 中的“Program Flash”的所完成的功能相同。使用 Binary Viewer 软件打开查看 bin 文件的效果如下图所示。



10.4 QSPI Flash

Flash 在写入数据之前必须先进行擦除（Erase），擦除过程以扇区（Sector）为单位。然后以页（Page）为单位，依次将数据写入 Flash 中连续的各页。

以米联 ZYNQ 开发板所使用的 QSPI Flash: S25FL256S 为例。Sector 的大小为 64KB，Page 的大小为 256B。另外，还存在两个重要参数：单位 Sector 擦除时间和单位 Page 写入时间，这决定了 Flash 的烧写速度。S25FL256S 所对应的单位 Sector 擦除时间为 130ms，单位 Page 写入时间 250 μ s。如下图最右侧的两栏所示。具体可参考芯片 datasheet。

Sector Erase Time (typ.)	30 ms (4 kB), 150 ms (64 kB)	500 ms (64 kB)	130 ms (64 kB), 520 ms (256 kB)
Page Programming Time (typ.)	700 μ s (256B)	1500 μ s (256B)	250 μ s (256B), 340 μ s (512B)

以 4MB 大小的 BOOT.bin 文件为例，写入之前需要擦除 $4096/64 = 64$ 个 Sector，最短需耗时 $64 \times 130\text{ms} = 8.32\text{s}$ 。接着，需要写入 $4096 \times 1024/256 = 16384$ 个 Page，最短需耗时 $16384 \times 250\mu\text{s} = 4.096\text{s}$ 。加上 ARM 中应用程序的软件开销，QSPI Flash 的擦除、写入时间总和不超过 15s。若需读出进行校验，则再额外增加读出时间、比对时间。QSPI Flash 的读出速度很快，读出整个 bin 文件耗时小于 1s，ARM 中应用程序的比对时间通常也很短，1 至 2s 即可完成。因此，对于 4MB 大小的 bin 文件，QSPI Flash 的擦除（Erase）、写入（Program）、校验（Verify）3 个过程所耗费的时间总和可以控制在 20s 以内。

10.5 驱动程序

所有的驱动程序文件均包含在 c_driver 文件夹中的 tcp 文件夹。

main 函数的完成的功能如下：

- 关闭 Data Cache，避免 bin 文件在 DDR 拷贝过程中维护 Cache 一致性造成的麻烦
- 配置 QSPI 接口及 QSPI Flash
- 配置 Timer 及其中断
- 初始化中断控制器及系统中断
- 初始化 LWIP 协议栈
- 建立 TCP Server，启动 Timer
- 持续从 LWIP 协议栈接收数据

10.5.1 建立 TCP Server

基于 LWIP 库在 ARM 中建立一个 TCP Server，IP 地址为 192.168.1.10，端口号为 5010。

10.5.2 lwip 库设置

本例程使用 RAW API，即函数调用不依赖操作系统。传输效率也比 SOCKET API 高，(具体可参考 xapp1026)。将 use_axieth_on_zynq 和 use_emaclite_on_zynq 设为 0。如下图所示。

Configuration for library: lwip141

Name	Value	Default	Type	Description
api_mode	RAW_API	RAW_API	enum	Mode of operation for lwIP (
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axi
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures em

修改 lwip_memory_options 设置，将 mem_size, memp_n_pbuf, mem_n_tcp_pcb, memp_n_tcp_seg 这 4 个参数值设大，这样会提高 TCP 传输效率。如下图所示。

lwip_memory_options				Options controlling lwIP m
mem_size	10485760	131072	integer	Size of the heap memory
memp_n_pbuf	2048	16	integer	Number of memp struct p
memp_n_sys_timeout	8	8	integer	Number of simultaneously
memp_n_tcp_pcb	1024	32	integer	Number of active TCP PCE
memp_n_tcp_pcb_listen	8	8	integer	Number of listening TCP c
memp_n_tcp_seg	1024	256	integer	Number of simultaneously
memp_n_udp_pcb	4	4	integer	Number of active UDP PCI
memp_num_api_msg	16	16	integer	Number of api msg struct
memp_num_netbuf	8	8	integer	Number of struct netbufs
memp_num_netconn	16	16	integer	Number of struct netconn
memp_num_tcpip_msg	64	64	integer	Number of tcpip msg str

修改 pbuf_options 设置，将 pbuf_pool_size 设大，增加可用的 pbuf 数量，这样

同样会提高 TCP 传输效率。如下图所示。

◆ pbuf_options	true	true	boolean	Pbuf Options
pbuf_link_hlen	16	16	integer	Number of bytes that should
pbuf_pool_bufsize	1700	1700	integer	Size of each pbuf in pbuf pool
pbuf_pool_size	4096	256	integer	Number of buffers in pbuf pool

修改 tcp_options 设置，将 tcp_snd_buf, tcp_wnd 参数设大，这样同样会提高 TCP 传输效率。如下图所示。

◆ tcp_options	true	true	boolean	Is TCP required ?
lwip_tcp	true	true	boolean	Is TCP required ?
tcp_maxrtx	12	12	integer	TCP Maximum retransmission
tcp_mss	1460	1460	integer	TCP Maximum segment size (
tcp_queue_ooseq	1	1	integer	Should TCP queue segments
tcp_snd_buf	65535	8192	integer	TCP sender buffer space (byt
tcp_synmaxrtx	4	4	integer	TCP Maximum SYN retransmi
tcp_ttl	255	255	integer	TCP TTL value
tcp_wnd	65535	2048	integer	TCP Window (bytes)

修改 temac_adapter_options 设置，将 n_rx_descriptors 和 n_tx_descriptors 参数设大。这样可以提高 zynq 内部 emac dma 的数据搬移效率，同样能提高 TCP 传输效率。如下图所示。

◆ temac_adapter_options	true	true	boolean	Settings for xps-ll-temac/A
emac_number	0	0	integer	Zynq Ethernet Interface number
n_rx_coalesce	1	1	integer	Setting for RX Interrupt coalescence
n_rx_descriptors	256	64	integer	Number of RX Buffer Descriptors
n_tx_coalesce	1	1	integer	Setting for TX Interrupt coalescence
n_tx_descriptors	256	64	integer	Number of TX Buffer Descriptors
phy_link_speed	CONFIG_LINKSPEED_AUTO	CONFIG_LINKSPEED_AUTO	enum	link speed as negotiated by PHY
tcp_ip_rx_checksum_offload	false	false	boolean	Offload TCP and IP Receive checksums
tcp_ip_tx_checksum_offload	false	false	boolean	Offload TCP and IP Transmission checksums
tcp_rx_checksum_offload	false	false	boolean	Offload TCP Receive checksums
tcp_tx_checksum_offload	false	false	boolean	Offload TCP Transmit checksums
temac_use_jumbo_frames	false	false	boolean	use jumbo frames

其余选项的参数默认即可，不用修改。

10.5.3 程序解析

TCP Server 建立由 `tcp_transmission.c` 文件中的 `tcp_recv_init` 和 `connect_accept_callback` 函数完成。

- `tcp_recv_init` 函数：
 - 调用 `tcp_new` 函数建立 1 个建立 TCP 连接所需的结构体。
 - 调用 `tcp_bind` 函数绑定 TCP Server 的本地 IP 地址和 TCP 端口号。
 - 调用 `tcp_listen` 函数启动监听外部任何 TCP Client 向 TCP Server 发送的 TCP 连接请求。
 - 调用 `tcp_accept` 函数指定当 TCP Server 与外部 TCP Client 建立 TCP 连接后的回调函数为 `connect_accept_callback`。
- `connect_accept_callback` 函数：
 - 当 ARM 中建立的 TCP Server 与外部 TCP Client 建立连接后，`connect_accept_callback` 函数将会被调用。
 - 调用 `tcp_recv` 函数指定当 TCP Server 接收来自 TCP Client 所发送数据包的回调函数为 `tcp_recv_callback`。

- 调用 `tcp_sent` 函数指定当 TCP Server 向 TCP Client 发送数据包时的回调函数为 `tcp_sent_callback`, 在例程 `tcp_sent_callback` 为空函数, 无实际作用。

10.5.4 接收保存 BOOT.bin 文件

接收 `BOOT.bin` 文件通过 `tcp_transmission.c` 中的 `tcp_recv_callback` 函数完成, 该函数为 TCP Server 接收数据包的回调函数, 每当接收到 TCP Client 的数据包时该函数都会被调用。该函数将 TCP Server 所接收到的 bin 文件的各数据包依次拷贝至 DDR 中首地址为 `FILE_BASE_ADDR` 的区域中, `FILE_BASE_ADDR` 为宏定义。

```
#define FILE_BASE_ADDR      0x10000000
```

可根据具体要求定义其地址, 注意要与 `lscript.ld` 中的 DDR 区域分开, 不能重合。

10.5.5 烧写 QSPI Flash

烧写 QSPI Flash 由 `qspi_g128_flash.c` 文件中的 `update_flash`、`FlashErase`、`FlashWrite`、`FlashRead` 等函数完成, 可支持 Micron、Spansion 等多个厂商, 128Mb 以上多种容量的 QSPI Flash。`qspi_g128_flash.c` 是根据 SDK 中 QSPI 接口的 `example:xqspips_g128_flash_example.c` 修改而成。

当接收完整个 bin 文件后, 在网络调试助手中输入“start update”, 含空格一共 12 个字符。`tcp_recv_callback` 函数接收到该命令之后便调用 `update_flash` 函数启动 bin 文件至 QSPI Flash 的烧写。该函数需要 1 个读缓存和 1 个写缓存, 分别存放从 flash 中读出的 bin 文件数据和需要写入 flash 中的 bin 文件数据。读缓存和写缓存的地址分别由 `READ_BASE_ADDR` 和 `WRITE_BASE_ADDR` 宏定义指定。可根据具体要求定义其地址, 同样需要注意要与 `lscript.ld` 中的 DDR 区域分开, 不能重合。

```
#define READ_BASE_ADDR      0x11000000
```

```
#define WRITE_BASE_ADDR     0x12000000
```

`update_flash` 函数:

- 将 TCP Server 接收的 bin 文件数据拷贝至写缓存中起始地址为 `WRITE_BASE_ADDR + 4` 的区域中, 增加 4 字节的地址偏移是由于在 `FlashWrite` 函数中, 写缓存的前 4 字节将被用于填充命令和写入地址字段。
- 调用 `FlashErase` 函数将 bin 文件数据所对应数量的扇区擦除。
- 调用 `FlashWrite` 将 bin 文件数据以页为单位依次全部写入 Flash 中。
- 调用 `FlashRead` 函数将刚才写入 Flash 的 bin 文件全部读出存入读缓存中。
- 将读出的 bin 文件数据与 TCP Server 接收的原始 bin 文件数据进行一一比对验证烧写的正确性。

`FlashErase`、`FlashWrite`、`FlashRead` 函数均源自于 SDK 中 QSPI 接口的 `example code`。可参考相应的 `example` 具体分析函数功能。

10.5.6 TCP 调试信息输出

例程中设计了一个 `tcp_printf` 函数, 用于向网络调试助手输出字符串调试信息。该函数暂时只支持字符串以“\n”结尾。

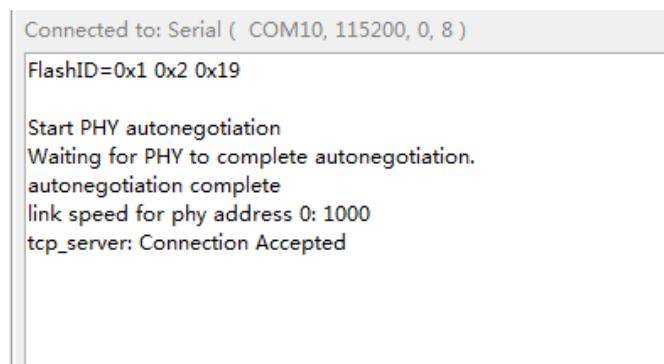
在本例程中使用该函数存在一个问题，即在 Flash 烧写开始直至结束前所有通过 `tcp_printf` 输出的调试信息，将无法及时发送，在烧写结束后应用程序回到 `main` 函数中包含 `xemacif_input(netif)` 函数的 `while` 循环中时，同时全部通过 TCP 发送至网络调试助手。

笔者尝试过几种方法均未能解决这个问题，例如通过 `tcp_nagle_disable()` 函数关闭 nagle 算法功能。笔者认为，一方面，可能跟 LWIP 库 TCP 部分函数的设计原理有关，另一方面，由于 flash 烧写部分应用程序将长时间占用 ARM，使得 `xemacif_input(netif)` 函数长时间无法被调用，而 ZYNQ 中的 LWIP 协议栈所有的数据接收都需要依靠应用程序调用 `xemacif_input()` 函数而实现。因此在这段时间中，可能由于长时间无法调用该函数而对 TCP 部分函数的运行造成某些影响，使数据发送产生阻塞或者延迟。由于 LWIP 中 TCP 部分库函数结构较复杂，笔者对于 TCP 协议研究也不多，限于时间和精力未作深究。

10.6 网络调试助手操作方法

10.6.1 发送 bin 文件

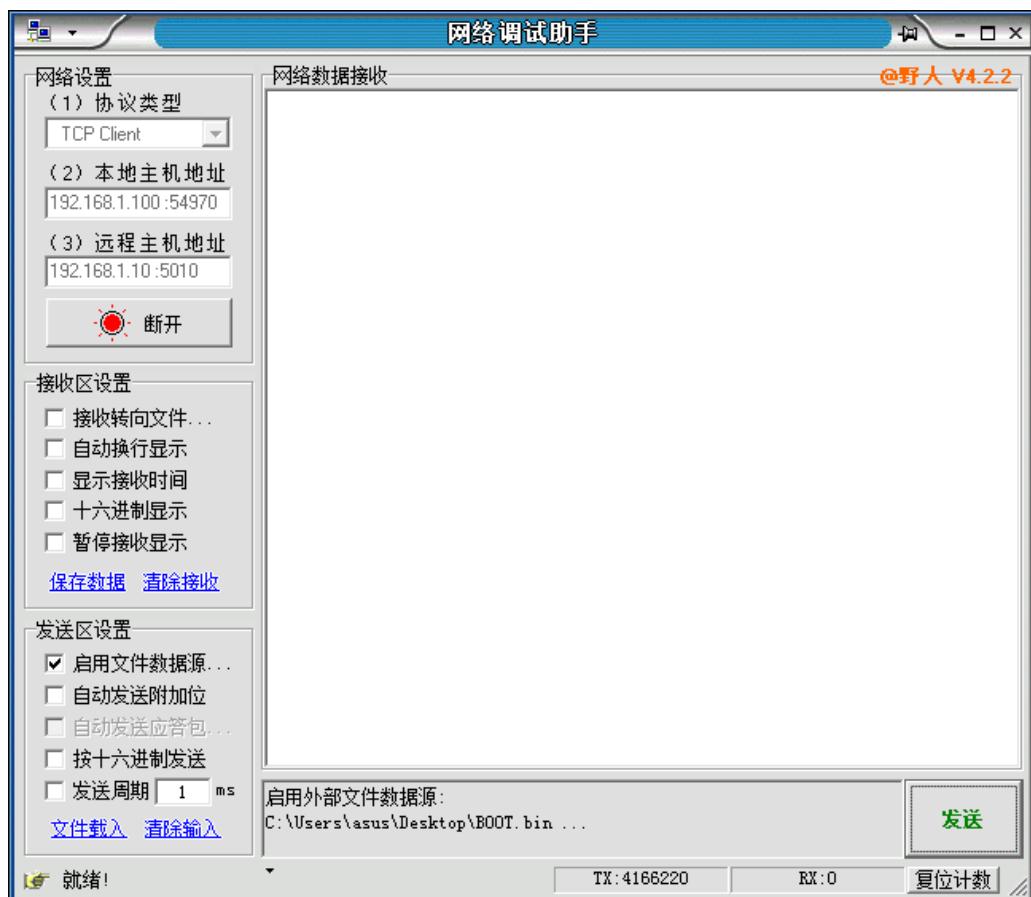
在 SDK 中下载程序至 ZYNQ 中。打开网络调试助手，选择 TCP Client 方式，输入 ARM 中定义的 TCP Server 的 IP 地址和端口号，然后点击连接按键，建立 TCP 连接。SDK 串口终端打印信息如下图所示。



The screenshot shows a terminal window titled "Connected to: Serial (COM10, 115200, 0, 8)". The log output is as follows:

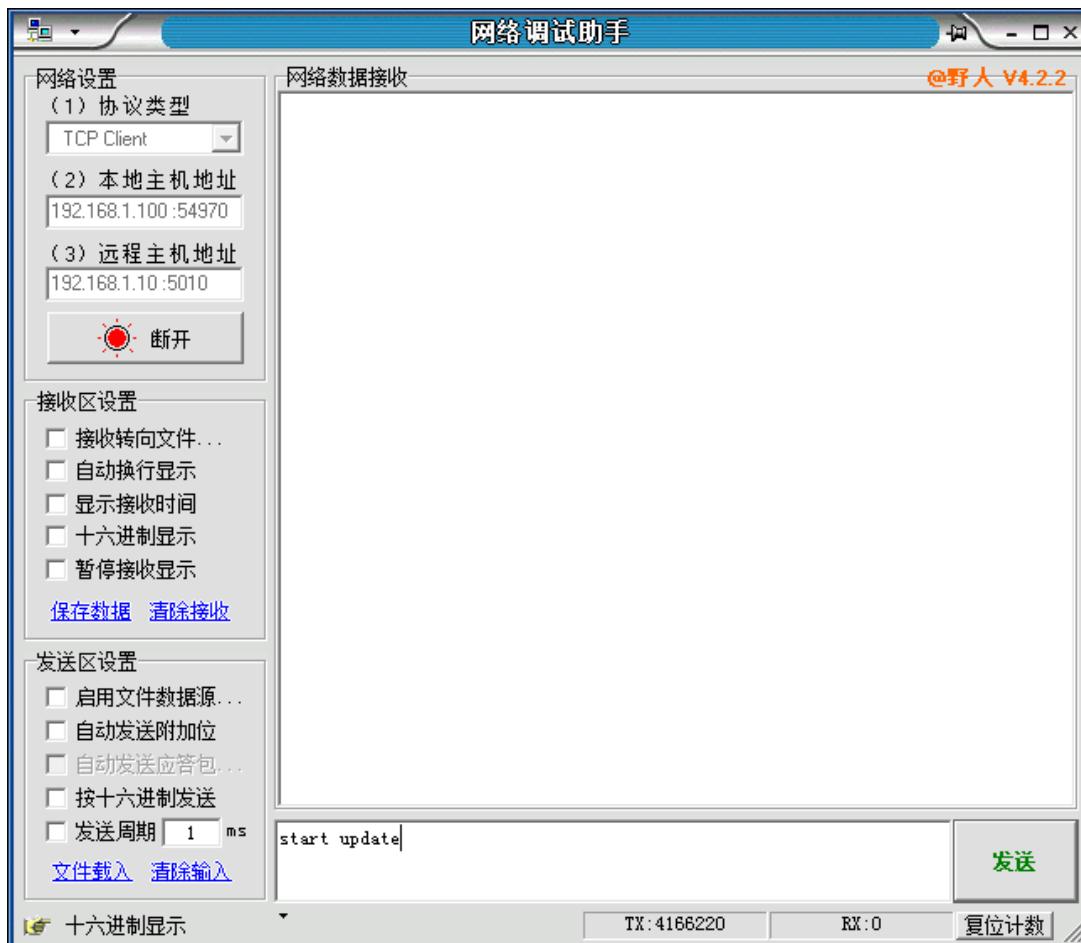
```
FlashID=0x1 0x2 0x19
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
tcp_server: Connection Accepted
```

在网络调试助手发送区设置里选择“启用文件数据源”，选择需要发送的 `BOOT.bin` 文件，然后点击发送。如下图所示。



10.6.2 发送启动 Flash 烧写命令

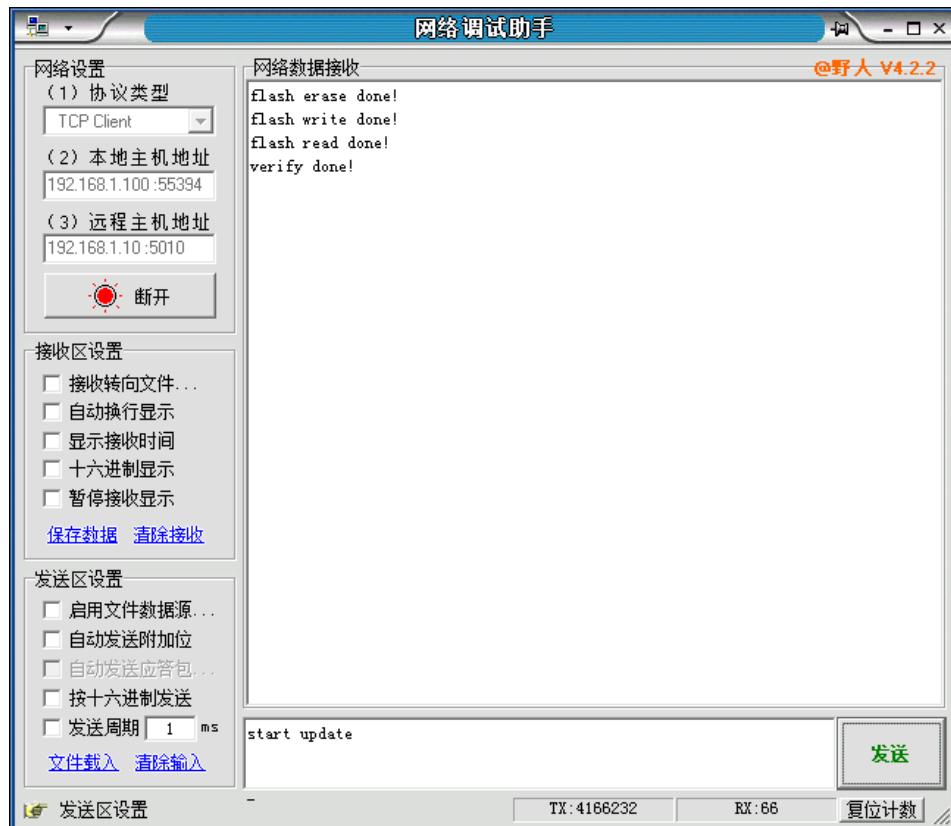
然后输入烧写启动命令“start update”，不要选择“按十六进制发送”，本例程中需要以 ASCII 码形式发送，含空格一共 12 个字符（不要在末尾加回车），千万不要输错，否则需要全部重新再来一遍。如下图所示。



启动烧写后，SDK 串口终端打印信息如下图所示。当提示“verify done!”表示整个烧写过程成功完成。

```
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
tcp_server: Connection Accepted
flash update start!
file length of BOOT.bin is 4166220 Bytes
flash erase done!
flash write done!
flash read done!
verify done!
```

网络调试助手接收 `tcp_printf` 函数的输出的信息如下图所示。



10.7 Bin 文件更新验证

烧写完成后，此时可断电重启验证更新的 BOOT.bin 文件。本例程作为演示，烧入的 bin 文件为 hello world 工程。重启开发板，SDK 串口终端打印信息如下图所示，代表 bin 文件更新成功。



10.8 待改进之处

- tcp_printf 函数在 flash 烧写过程中无法及时发送数据的问题有待解决。
- 无校验差错重新写入功能。当将 bin 文件写入 Flash 之后再读出校验时，若出现错误，则需将错误的 Page 重新写入。
- 由于例程中使用网络调试助手作为传输 BOOT.bin 文件的工具，仅使用了 TCP 协议，无法在 TCP 协议之上设计自定义协议来规范并完善 bin 文件的传输过程，用户可控性较差。例如，启动命令“start update”一旦输入错误并发送至 ZYNQ，则板卡将需断电重启，整个过程需从头重来一遍。理想情况下，用户应根据实际需求在 TCP 之上设计相应的 bin 文件传输协议（例如给每一个 bin 文件数据包加上含有协议信息的包头、包尾等）来提高传输的可靠性和可控性。另外，用户还需设计与之相对应的上位机软件进行 bin 文件的发送，以及与 ZYNQ 的通信。

CH11_基于 UDP 的 QSPI Flash bin 文件网络烧写

11.1 概述

为了满足不同的需求，本例程在“基于 TCP 的 QSPI Flash bin 文件网络烧写”上进行修改，将 bin 文件的传输协议替换为 UDP。与采用 TCP 协议的例程相比，本例程无需使用 ZYNQ 内部的定时器，无定时器中断，LWIP 中 UDP 部分的 API 函数结果也更为简洁，易于使用，简化了 ARM 中的 C 程序设计，但使用 UDP 协议后文件传输的可靠性无法保证，因此需要更具实际应用进行权衡。

本例程基于 Vivado 2016.4 版本开发。

11.2 基本原理

与“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程相同，并将 TCP 协议替换为 UDP 协议。

11.2.1 Bin 文件

见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

11.2.2 QSPI Flash

见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

11.3 驱动程序

所有的驱动程序文件均包含在 c_driver 文件夹中的 udp 文件夹。

11.3.1 main 函数

main 函数的完成的功能如下：

- 关闭 Data Cache，避免 bin 文件在 DDR 拷贝过程中维护 Cache 一致性造成的麻烦
- 配置 QSPI 接口及 QSPI Flash
- 初始化中断控制器及系统中断
- 初始化 LWIP 协议栈
- 建立 UDP 连接
- 持续从 LWIP 协议栈接收数据

11.3.2 建立 UDP 连接

基于 LWIP 库在 ARM 中建立一个 UDP 连接，ZYNQ 的 IP 地址为 192.168.1.10，端口号为 5010，远程 IP 地址为 192.168.1.100，端口号为 8080。

11.3.3 lwip 库设置

本例程使用 RAW API，即函数调用不依赖操作系统。传输效率也比 SOCKET API 高，(具体可参考 xapp1026)。将 use_axieth_on_zynq 和 use_emaclite_on_zynq 设为 0。如下图所示。

Configuration for library: lwip141				
Name	Value	Default	Type	Description
api_mode	RAW_API	RAW_API	enum	Mode of operation for lwIP (I)
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axi
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures em

修改 lwip_memory_options 设置，将 mem_size，memp_n_pbuf 这 2 个参数值设大，这样会提高 UDP 传输效率。如下图所示。

lwip_memory_options					Options control
mem_size	10485760	131072	integer	Size of the heap	
memp_n_pbuf	2048	16	integer	Number of men	
memp_n_sys_timeout	8	8	integer	Number of simu	
memp_n_tcp_pcb	32	32	integer	Number of activ	
memp_n_tcp_pcb_listen	8	8	integer	Number of liste	
memp_n_tcp_seg	256	256	integer	Number of simu	
memp_n_udp_pcb	4	4	integer	Number of activ	
memp_num_api_msg	16	16	integer	Number of api	
memp_num_netbuf	8	8	integer	Number of struc	
memp_num_netconn	16	16	integer	Number of struc	
memp_num_tcip_msg	64	64	integer	Number of tcipi	

修改 pbuf_options 设置，将 pbuf_pool_size 设大，增加可用的 pbuf 数量，这样同样会提高 UDP 传输效率。如下图所示。

pbuf_options					Pbuf Options
pbuf_link_hlen	16	16	integer	Number of bytes that should	
pbuf_pool_bufsize	1700	1700	integer	Size of each pbuf in pbuf po	
pbuf_pool_size	4096	256	integer	Number of buffers in pbuf po	

由于无需使用 TCP 协议，修改 tcp_options 设置，将 lwip_tcp 设置为 false，tcp_queue_ooseq 设为 0，关闭 tcp 功能，如下图所示。

tcp_options					Is TCP required ?
lwip_tcp	true	true	boolean	Is TCP required ?	
tcp_maxrtx	12	12	integer	TCP Maximum retran	
tcp_mss	1460	1460	integer	TCP Maximum segme	
tcp_queue_ooseq	0	1	integer	Should TCP queue se	
tcp_snd_buf	8192	8192	integer	TCP sender buffer sp	
tcp_synmaxrtx	4	4	integer	TCP Maximum SYN re	
tcp_ttl	255	255	integer	TCP TTL value	
tcp_wnd	2048	2048	integer	TCP Window (bytes)	

修改 temac_adapter_options 设置，将 n_rx_descriptors 和 n_tx_descriptors 参数设大。这样可以提高 zynq 内部 emac dma 的数据搬移效率，同样能提高 UDP 传输效率。如下

图所示。

temac_adapter_options	true	true	boolean	Settings for xps_ll_temac/Ax
emac_number	0	0	integer	Zynq Ethernet Interface number
n_rx_coalesce	1	1	integer	Setting for RX Interrupt coalescing
n_rx_descriptors	256	64	integer	Number of RX Buffer Descriptors
n_tx_coalesce	1	1	integer	Setting for TX Interrupt coalescing
n_tx_descriptors	256	64	integer	Number of TX Buffer Descriptors
phy_link_speed	CONFIG_LINKSPEED_Auto	CONFIG_LINKSPEED_Auto	enum	link speed as negotiated by PHY
tcp_ip_rx_checksum_offload	false	false	boolean	Offload TCP and IP Receive checksums
tcp_ip_tx_checksum_offload	false	false	boolean	Offload TCP and IP Transmit checksums
tcp_rx_checksum_offload	false	false	boolean	Offload TCP Receive checksums
tcp_tx_checksum_offload	false	false	boolean	Offload TCP Transmit checksums
temac_use_jumbo_frames	false	false	boolean	use jumbo frames

其余选项的参数默认即可，不用修改。

11.3.4 程序解析

UDP 连接建立由 `udp_transmission.c` 文件中的 `udp_recv_init` 函数完成。

`udp_recv_init` 函数：

调用 `udp_new` 函数建立 1 个建立 UDP 连接所需的结构体。

调用 `udp_bind` 函数绑定本地 IP 地址和 UDP 端口号。

调用 `udp_connect` 函数建立 UDP 连接，并绑定远程 IP 地址和 UDP 端口号。

调用 `udp_recv` 函数指定用于接收 UDP 数据包的回调函数为 `udp_recv_callback`。

11.3.5 接收保存 BOOT.bin 文件

接收 `BOOT.bin` 文件通过 `udp_transmission.c` 中的 `udp_recv_callback` 函数完成，该函数为 UDP 接收数据包的回调函数，每当接收到 UDP 的数据包时该函数都会被调用。保存位置与“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程相同。

11.3.6 烧写 QSPI Flash

见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

11.3.7 UDP 调试信息输出

例程中设计了一个 `udp_printf` 函数，用于向网络调试助手输出字符串调试信息。该函数暂时只支持字符串以“\n”结尾。

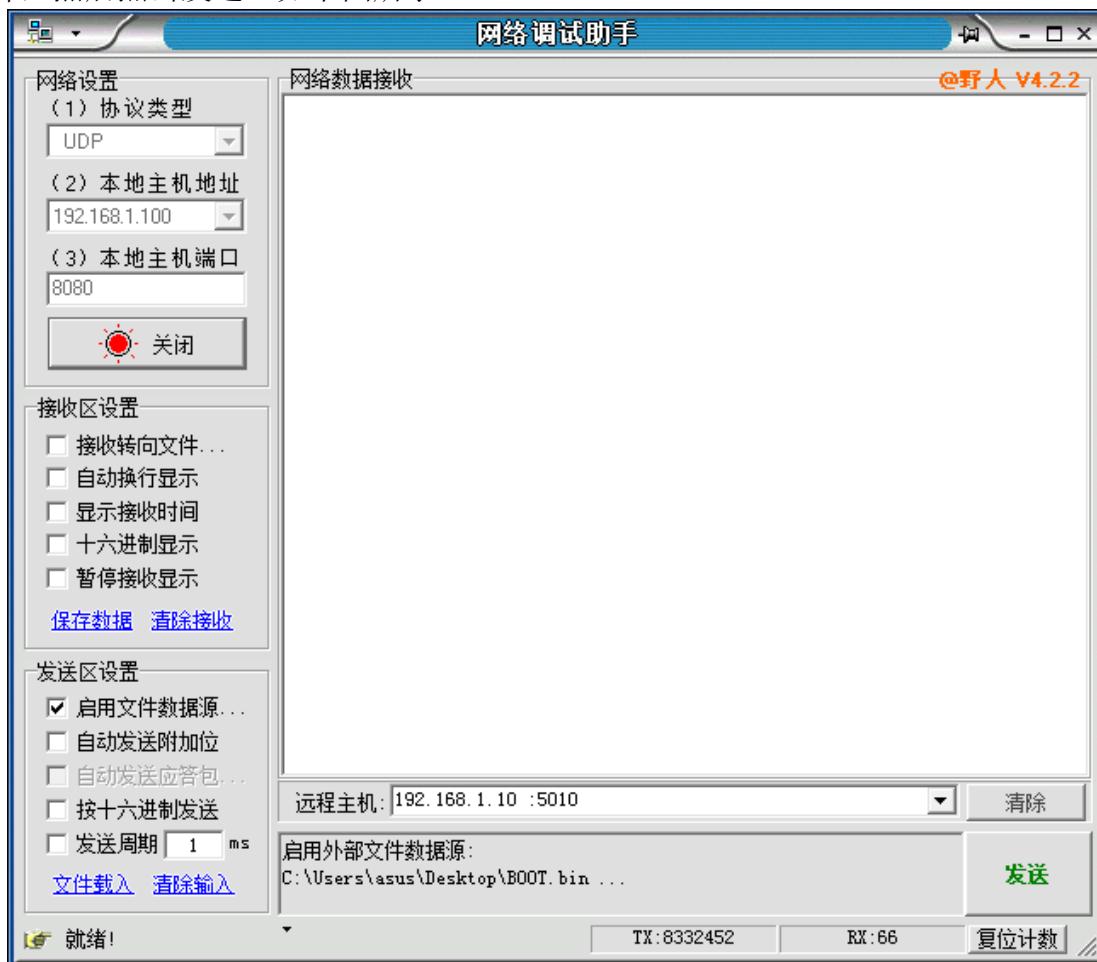
该函数实现思路与“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程中 `tcp_printf` 函数基本相同。不同的是，任何时刻通过 `udp_printf` 发送的数据都会及时通过网络发出，并被网络调试助手接收，而不像 `tcp_printf` 函数的发送会在 flash 烧写过程中被阻塞。

这是由于 UDP 为不可靠传输，不像 TCP 在传输过程中需进行持续握手。因此，UDP 接收和发送过程相互独立，具体在 LWIP 协议栈中反应为 `udp_send` 函数不依赖 `xemacif_input` 函数。

11.4 网络调试助手操作方法

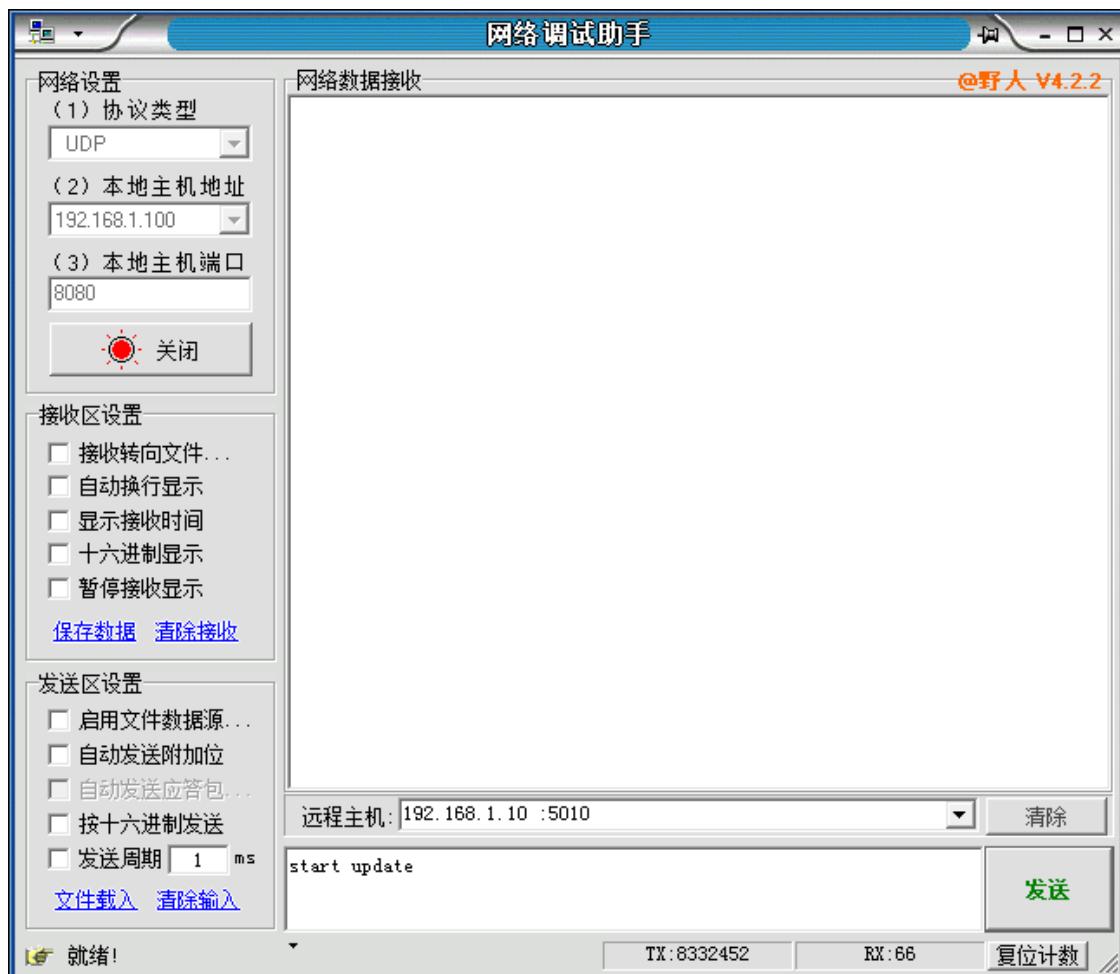
11.4.1 发送 bin 文件

在 SDK 中下载程序至 ZYNQ 中。打开网络调试助手，选择 UDP 方式，输入电脑的 IP 地址和 UDP 端口号，然后点击打开按键，在远程主机中填入 ZYNQ 的 IP 地址和 UDP 端口号。在网络调试助手发送区设置里选择“启用文件数据源”，选择需要发送的 BOOT.bin 文件，然后点击发送。如下图所示。



11.4.2 发送启动 Flash 烧写命令

然后输入烧写启动命令“start update”，不要选择“按十六进制发送”，本例程中需要以 ASCII 码形式发送，含空格一共 12 个字符（不要在末尾加回车），千万不要输错，否则需要全部重新再来一遍。如下图所示。

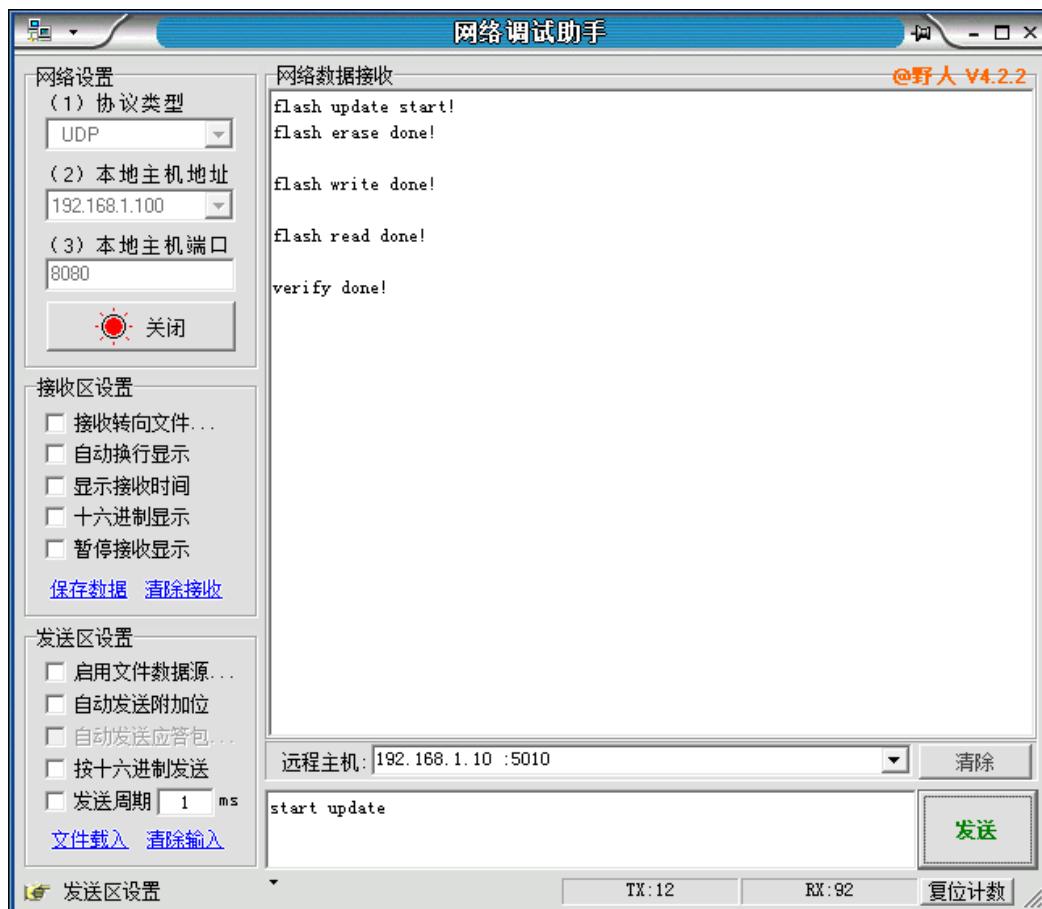


启动烧写后，SDK 串口终端打印信息如下图所示。当提示“verify done!”表示整个烧写过程成功完成。

```
Connected to: Serial ( COM10, 115200, 0, 8 )
FlashID=0x1 0x2 0x19

Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
flash update start!
file length of BOOT.bin is 4166220 Bytes
flash erase done!
flash write done!
flash read done!
verify done!
```

网络调试助手接收 `udp_printf` 函数实时输出的信息如下图所示。



11.5 Bin 文件更新验证

见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

11.6 待改进之处

- UDP 为不可靠传输，为了提高使用 UDP 协议传输 bin 文件的可靠性，可在其之上设计额外的传输协议，并加上类似传输握手的功能。
- 其余见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

CH12_ZYNQ A9 TCP UART 双核 AMP 例程

12.1 概述

ZYNQ 中存在两个独立的 ARM 核，在很多应用场景中往往只需使用其中的 1 个核心即可。然而，对于复杂的设计，例如多任务，并行控制、处理等，单个核心将难以胜任。因此，为了尽可能发挥 ZYNQ 中双 ARM 核的优势和性能，进行双核应用的开发显得尤为重要。同时，也进一步为 Xilinx 下一代 MPSOC 多核异构处理器的使用打下基础。

在 ZYNQ 中实现双 ARM 核 AMP 应用可以参考 Xilinx 官方的 XAPP1078 和 XAPP1079。在 SDK 中也有用于双核应用开发的 openamp 库可以使用。

本例程未使用 openamp 库，通过自行设计的代码实现了一个简单的双核应用，旨在说明：

- 双核通过软件中断进行核间通信的原理及方法。
- 双核通过共享内存进行数据交互的基本原理和设计方法。
- 双核协同工作的基本模式。
- 双核 BOOT 的方法。

本例程基于 vivado 2016.4 开发。

12.2 基本原理

本例程在 ARM 核的 CORE0 建立一个 TCP Server，CORE0 通过 TCP Server 从外部 TCP Client 接收数据。当 CORE0 接收到 1 个 TCP 包后，将数据复制到 DDR3 中的缓存区域内，然后将数据信息（长度、首地址）放入 CORE0 和 CORE1 在 DDR3 的共享内存中。

接着，CORE0 通过触发软件中断通知 CORE1 数据信息已存入共享内存中，CORE1 接到中断后从共享内存读出数据信息，并将对应长度的数据复制到缓存区中，然后通过 UART 将数据输出。当 CORE1 通过串口完成数据输出后，同样通过触发软件中断通知 CORE0 数据发送已完成，CORE0 接到中断后通过串口打印完成信息，然后开始接收下一个 TCP 包，重复上述过程。

最后，通过 FSBL 实现双核的 QSPI BOOT。

12.2.1 软件中断

在 UG585 的 Interrupts 部分 7.2.1 章节可以找到关于软件中断（SGI）的说明。简而言之，软件中断就是 CPU 自己产生的中断，可用于触发自身和其他 CPU。ZYNQ 中共有 16 个软件中断可以使用，对应的中断号为 0~15，如下图所示。通过软件中断可以实现 CPU 之间的相互通信。本例程使用了编号 1 和 2 的软件中断。

Table 7-2: Software Generated Interrupts (SGI)

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

12.2.2 共享内存通信

所谓共享内存就是，CORE0 和 CORE1 在 DDR3 内存中约定一块地址及长度已知的内存区域。然后，两者之间便可通过这片区域进行数据的传递。

两个核心各自拥有独立的 L1 DCache，并且共享同一个 L2 DCache，在 ZYNQ 中存在一个 Snoop Control Unit (SCU) 用于维护 CORE0 和 CORE1 的 L1 DCache 与 L2 DCache 之间的一致性，无需用户干预。因此，虽然 CORE0 和 CORE1 的共享内存区域位于 DDR 中，两者之间的数据传递并不需要考虑 DCache 一致性的维护。但是，为了更好阐明多级存储器结构的特性以及 DCache 一致性维护的问题，本例程在两核间通过 DDR3 共享内存进行数据交互时加入了 DCache 一致性操作，最终达到的效果与不使用 DCache 一致性操作时相同。

DCache 一致性维护的原理为：在多级存储器结构中，CPU 通过 1 级或多级 Cache 与 DDR 产生连接，CPU 本身不直接访问 DDR，而是通过 Cache 访问 DDR。Cache 中始终会暂存一小部分（通常是 KB~几 MB 量级）CPU 最近访问的 DDR 某些地址区域中的数据。因此，在应用程序中对 DDR 进行读或写操作，实际上都是 CPU 对 Cache 进行读或写操作。当 DDR 中某个地址范围内的数据突然被除 CPU 以外的 Master（如 DMA）改变时，若此时 Cache 中保存了这些区域的数据，且这些数据在 Cache 中状态为有效时，当 CPU 需要再次读取 DDR 这片区域的数据时，就不会让 Cache 去读取 DDR 中此区域内最新的数据来更新 Cache，再从 Cache 里读取最新的数据，而是直接从 Cache 中读取原来的旧数据，显然这不是我们所期望的结果。这时，便引入了所谓的 Cache 一致性问题。

ZYNQ 中存在 ICache 和 DCache，ICache 用于缓存可执行程序，DCache 用于缓存数据。一般情况下，用于保存可执行程序的 DDR 地址范围不会被除 CPU 以外的对象访问。因此，一般不存在 ICache 的一致性问题。而 DCache 在很多应用中却经常会被除 CPU 以外的对象访问，所以存在一致性问题。

ZYNQ 中维护 DCache 一致性的方法为：写入方将数据写入 DDR 对应地址区域后，需将残留在 DCache 中相应地址范围内的数据全部刷入 DDR3 中。读取方在从 DDR 相应地址读取数据之前，需将 DCache 中 DDR 相应地址范围内的数据全部设置为 invalid，然后 CPU 会再次通过 DCache 从 DDR3 中读取该地址范围内最新的数据。

12.2.3 双核 BOOT

裸机双核 BOOT 的方法参考自 Xilinx 的 XAPP1079。首先，通过 FSBL 实现 CORE0

的 BOOT。当 CORE0 启动进入 main 函数后，配合 FSBL 再实现 CORE1 的启动。
具体原理参考 XAPP1079，此处不作赘述。

12.3 驱动程序

CORE0 工程的驱动程序文件位于 c_driver 文件夹中的 core0 文件夹中，CORE1 工程的驱动程序文件位于 c_driver 文件夹中的 core1 文件夹中。需要说明的是，core0 和 core1 的工程在 DDR 所占用的地址区域进行如下划分：

- CORE0: 0x00100000~0x01FFFFFF，见下图 core0 的 lscript.ld 文件设置。

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x100000	0x1F00000
ps7_qspi_linear_0_S_AXI_BASEADDR	0xFC000000	0x1000000
ps7_ram_0_S_AXI_BASEADDR	0x0	0x30000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0xFE00

- CORE1: 0x02000000~0x02FFFFFF，见下图 core1 的 lscript.ld 文件设置。

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x2000000	0x1000000
ps7_qspi_linear_0_S_AXI_BASEADDR	0xFC000000	0x1000000
ps7_ram_0_S_AXI_BASEADDR	0x0	0x30000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0xFE00

设计双核应用，两个工程的内存分配是一个关键的前提，程序所占用的 DDR 空间不能发生重合，应完全分隔开。

12.3.3 CORE0 工程

12.3.3.1 main 函数

main 函数的完成的功能如下：

- 配置 Timer 及其中断
- 初始化中断控制器
- 初始化软件中断
- 初始化 LWIP 协议栈
- 建立 TCP Server，启动 Timer
- 配合 FSBL 启动 CORE1
- 持续从 LWIP 协议栈接收数据，若接收到 CORE1 触发的软件中断，则作出响应。

12.3.3.2 建立 TCP Server

基于 LWIP 库在 ARM 中建立一个 TCP Server，IP 地址为 192.168.1.10，端口号为 5010。

- lwip 库设置
见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。
- 程序解析
见“基于 TCP 的 QSPI Flash bin 文件网络烧写”例程。

12.3.3.3 初始化软件中断

通过 software_intr.c 中的 Init_Software_Intr() 函数初始化软件中断，对于 CORE0，该函数完成如下功能。

- 初始化 CORE1 到 CORE0 的软件中断，中断号为 2，设置中断优先级和触发方式。
- 绑定该中断的中断服务函数为 Cpu0_Intr_Handler。
- 将该中断映射至 CORE0，并使能。

12.3.3.4 启动 CORE1

通过 main.c 中的 Start_cpu1() 函数配合 FSBL 完成 CORE1 的启动。Start_cpu1() 原理如下：

- 将 FSBL 工程位于 OCM 中的 stack 区域和 vector table 区域的 cache 禁用。
- 将 CORE1 工程代码的位于 DDR 中的起始地址 0x02000000（见 core1 的 lscript.1d）写入 FSBL 的 vector table 中定义的 cpu1_catch 地址内。CORE1 工程代码的起始地址由如下宏所定义。若 core1 的 lscript.1d 中代码位于 DDR 的起始地址发生更改，则该宏定义也必须进行相应更改。

```
#define APP_CPU1_ADDR 0x02000000
```

- 复位 CORE1 及其时钟
- 使能 CORE1 及其时钟

12.3.3.5 数据写入共享内存

CORE0 通过 TCP 协议接收外部 TCP Client 发送的数据包，并将数据信息写入 CORE0 和 CORE1 共享内存中。共享内存首地址定义为：

```
#define SHARED_BASE_ADDR 0x08000000
```

为共享内存区域在 shared_mem.h 中定义结构体 shared_region，其中包含了两核间所需交互的数据长度及数据指针。

```
typedef struct
{
    u32 data_length;
    u8* dataload;
} shared_region;
```

CORE0 通过 shared_mem.c 中的 put_data_to_region() 函数将所需传递的数据长度及

指针存入 shared region 结构体中。在 put_data_to_region 函数中调用 Xil_DCacheFlushRange 函数将 DCache 中该数据指针所指向内存区域的数据刷入 DDR 中，进行 DCache 一致性维护。

CORE0 将此结构体放置于共享内存首地址 SHARED_BASE_ADDR，CORE1 便可以从该地址读取 CORE0 所需传递的数据信息，从而进一步获取数据。

12.3.3.6 触发软件中断

CORE0 通过调用 shared_mem.c 中的 Gen_Software_Intr 函数触发 CORE0 到 CORE1 的软件中断，中断号为 1，中断目标 CPU 设置为 CORE1。

12.3.3.7 响应软件中断

当 CORE1 向 CORE0 触发中断号为 2 的软件中断时，CORE0 调用 Cpu0_Intr_Hanedler 函数响应此中断。然后，CORE0 通过串口打印相应信息。

12.4 CORE1 工程

12.4.1 main 函数

main 函数的完成的功能如下：

- 配置 Timer 及其中断
- 初始化中断控制器
- 初始化软件中断
- 等待 CORE0 触发的软件中断，将 CORE0 通过共享内存传递的数据由串口输出
- 串口数据输出完成后触发 CORE1 到 CORE0 的软件中断

12.4.2 初始化软件中断

通过 software_intr.c 中的 Init_Software_Intr() 函数初始化软件中断，对于 CORE1，该函数完成如下功能。

- 初始化 CORE0 到 CORE1 的软件中断，中断号为 1，设置中断优先级和触发方式。
- 绑定该中断的中断服务函数为 Cpl_Intr_Hanedler。
- 将该中断映射至 CORE1，并使能。

12.4.3 响应软件中断

当 CORE0 向 CORE1 触发中断号为 1 的软件中断时，CORE1 调用 Cpu1_Intr_Hanedler 函数响应此中断。然后，CORE1 开始从共享内存中读取 CORE0 所传递的数据。

12.4.4 共享内存数据读出

CORE1 从共享内存区域 SHARED_BASE_ADD 地址中获取 CORE0 传递的数据信息，通过 shared_mem.c 中的 get_data_from_region() 函数将 CORE0 传递的数据长度及指针读出，并复制到本地缓存中。

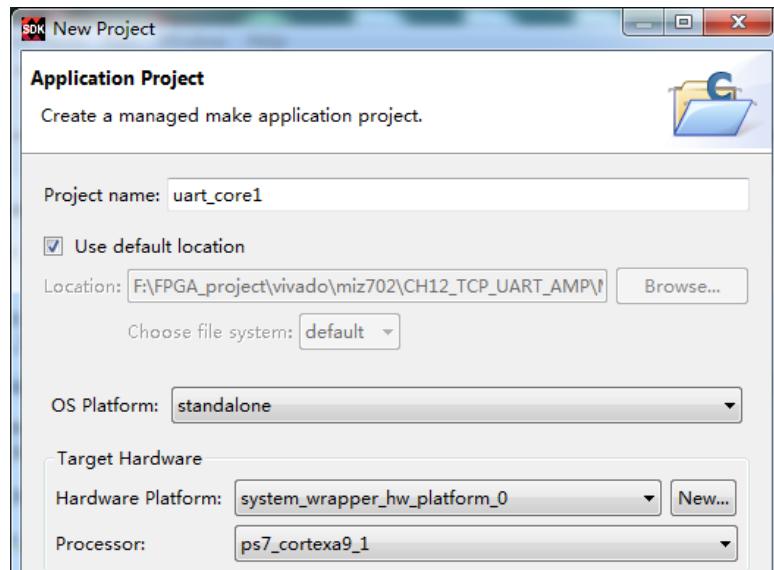
在 get_data_from_region 函数中，在将 CORE0 传递的数据复制到本地缓存区域之前，调用 Xil_DCacheInvalidateRange 函数将 DCache 中该数据指针所指向内存区域的数据设置为 invalid，进行 DCache 一致性维护。

12.4.5 触发软件中断

当 CORE1 将从共享内存中读取的数据通过串口输出完毕后，CORE1 通过调用 shared_mem.c 中的 Gen_Software_Intr 函数触发 CORE1 到 CORE0 的软件中断，中断号为 2，中断目标 CPU 设置为 CORE0。以此通知 CORE0，CORE1 串口输出数据完成。

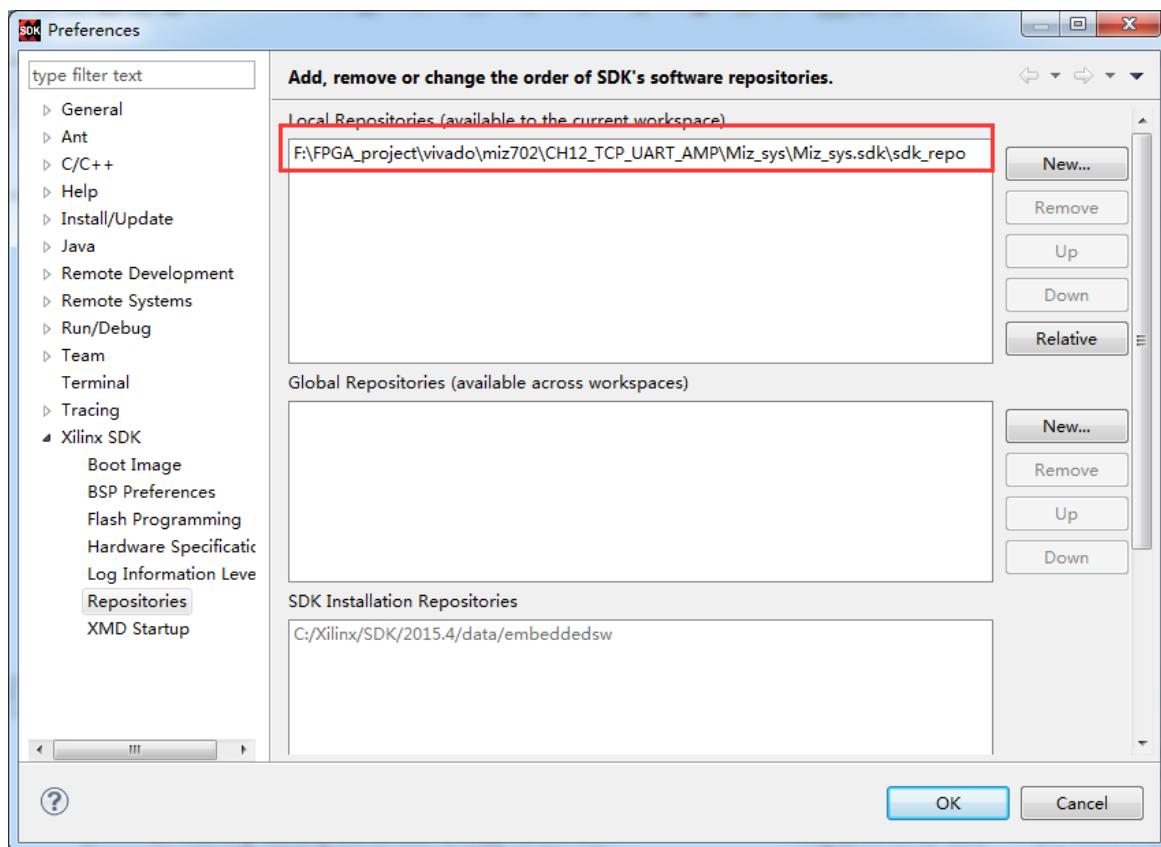
12.5 工程创建及设置关键步骤

- CORE1 工程创建时 Processor 要选择 ps_cortexa7_1，不要选成 ps_cortexa7_0，如下图所示。

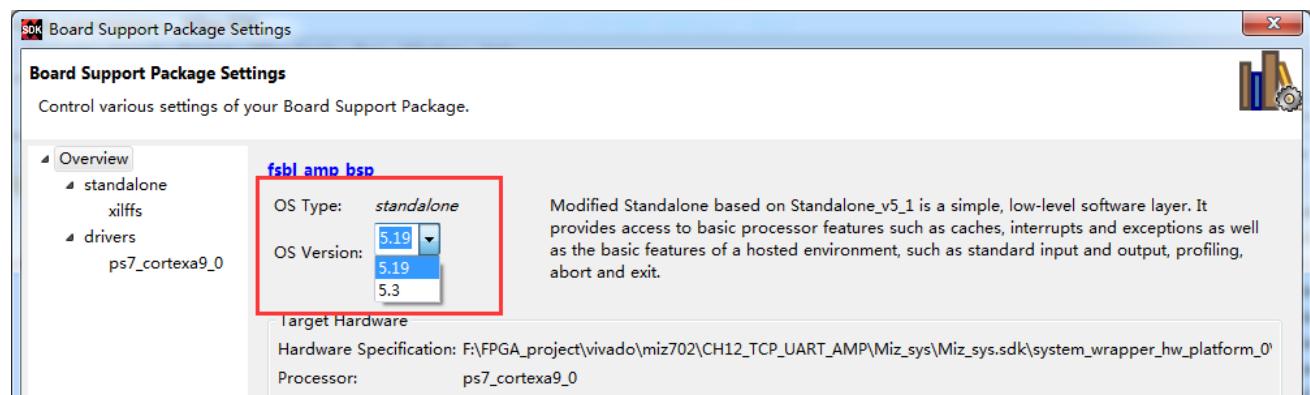


- 创建 FSBL 工程后，需替换其 bsp 的 standalone 库。

首先，设置工程目录下 sdk_repo 文件夹的路径，sdk_repo 文件夹中包含了需要使用的 standalone 库，如下图所示。用户需要根据自己所建工程的实际路径进行修改，使用原工程的路径设置将产生错误。

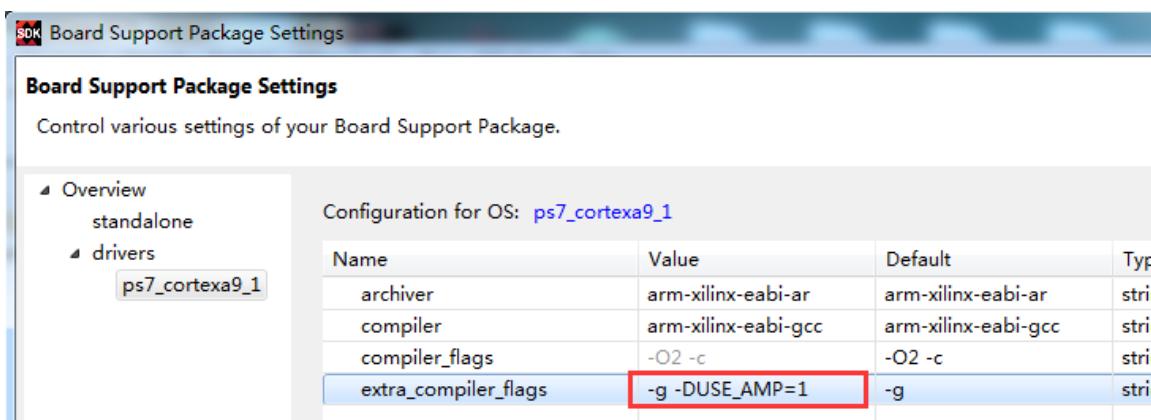


其次，更改 FSBL 工程的 bsp 中 standalone 的版本，将其更换为 sdk_repo 文件夹中的 5.19 版本，如下图所示。该版本 standalone 库来自 xapp1079，只有使用该版本的 standalone 库才可实现双核 BOOT，自带的 standalone 库无法实现。



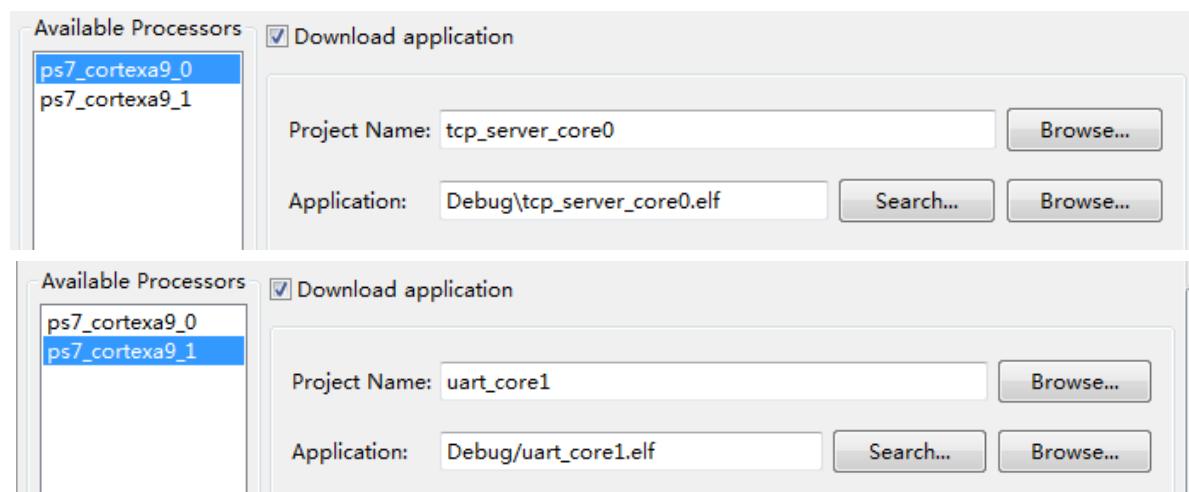
● 设置 CORE1 工程的编译选项

在 CORE1 工程的 bsp 中要增加编译选项 “-DUSE_AMP=1” ，如下图所示。该编译选项将影响到 CORE1 工程代码里中断控制器 SCUGIC 的初始化函数以及 Cache 操作函数的编译，若不增加该选项，可能会出现 CORE0 和 CORE1 中断异常和 Cache 一致性维护异常。



12.6 工程调试关键步骤

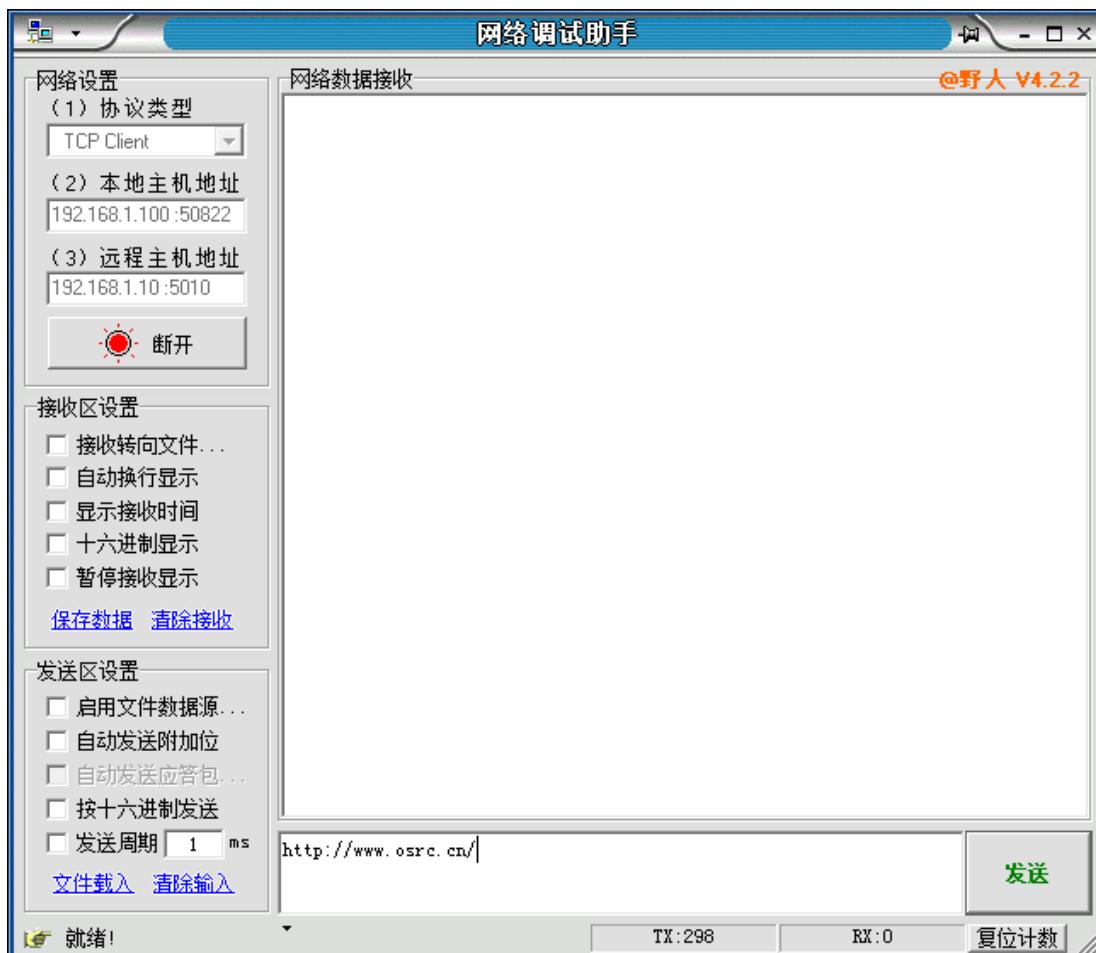
- system debugger 里同时添加 core0 和 core1 工程文件，如下图所示。Debug 时先让 CORE0 运行，再让 CORE1 运行。



- Debug 时一定要注释 CORE0 工程中 main 函数里的 Start_cpu1 函数，否则在 debug 时，CORE1 将无法正常工作。只有当需要生成 BOOT.bin 文件时，才需要使用该函数。

12.7 网络调试助手操作方法

在 SDK 中下载程序至 ZYNQ 中。打开网络调试助手，选择 TCP Client 方式，输入 ARM 中定义的 TCP Server 的 IP 地址和端口号，然后点击连接按键，建立 TCP 连接。输入任意文字信息发送。如图所示。



SDK 终端串口输出的信息如下图所示。

```
Connected to: Serial ( COM10, 115200, 0, 8 )

core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
http://www.osrc.cn/
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
```

继续通过网络调试助手发送信息，串口输出如下图所示。

```
Connected to: Serial ( COM10, 115200, 0, 8 )
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
http://www.osrc.cn/
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
南京米联电子
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
```



```
Connected to: Serial ( COM10, 115200, 0, 8 )
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
南京米联电子
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
南京米联电子ZYNQ开发板
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
```

12.8 生成 BOOT.bin

切记在生成 BOOT.bin 文件时不要注释 CORE0 工程中 main 函数里的 Start_cpu1 函数，否则将无法 BOOT CORE1。生成 BOOT.bin 文件时依次添加 FSBL 工程的 elf，bit 文件，CORE0 和 CORE1 的 elf 文件，如下图所示。

Boot image partitions	
File path	
(bootloader) F:\FPGA_project\vivado\miz702\CH12_TCP_UART_AMP\Miz_sys\Miz_sys.sdk\fsbl_amp\Debug\fsbl_amp.elf	
F:\FPGA_project\vivado\miz702\CH12_TCP_UART_AMP\Miz_sys\Miz_sys.sdk\system_wrapper_hw_platform_0\system_wrapper.bit	
F:\FPGA_project\vivado\miz702\CH12_TCP_UART_AMP\Miz_sys\Miz_sys.sdk\tcp_server_core0\Debug\tcp_server_core0.elf	
F:\FPGA_project\vivado\miz702\CH12_TCP_UART_AMP\Miz_sys\Miz_sys.sdk\uart_core1\Debug\uart_core1.elf	

12.9 双核 BOOT 验证

将 BOOT.bin 文件烧入 QSPI flash 中，重启开发板。SDK 串口终端输出信息，如下图所示。当提示“core1:application start”，代表 CORE0 和 CORE1 都已成功启动。

```
Connected to: Serial ( COM10, 115200, 0, 8 )
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
core0: application start
core1: application start
```

使用网络调试助手进行 TCP 连接并发送数据，串口输出如下图所示。此时，验证了双核 BOOT 后，CORE0 和 CORE1 均运行正常。

```
Connected to: Serial ( COM10, 115200, 0, 8 )
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
core0: application start
core1: application start
core0: tcp_server: Connection Accepted
core1: receive interrupt from core0!
core1: the data received from core0 is:
*****
南京米联电子ZYNQ开发板
*****
core0: receive interrupt from core1!
core0: core1 uart printf finish
-----
```

CH13_通过 BRAM 进行 PS 和 PL 间的数据交互

13.1 概述

在之前的教程中，已经介绍了一种通过使用 AXI DMA 来进行 PS 和 PL 间的高速数据传输的方法。这种方案的特点在于，传输速度快，数据以批量的形式进行传输，无需占用 PS 端的 ARM。

然而，当我们需要在 PS 和 PL 之间传输少量，地址不连续，且长度不规则的数据，比如，配置参数，变量，控制信息等，此时 AXI DMA 便不再适用了。为此，本例程针对这种应用场合，介绍了一种基于 PL 端 BRAM 的方式，来进行 PS 和 PL 间的数据交互。

本例程基于 Vivado 2016.4 版本开发。

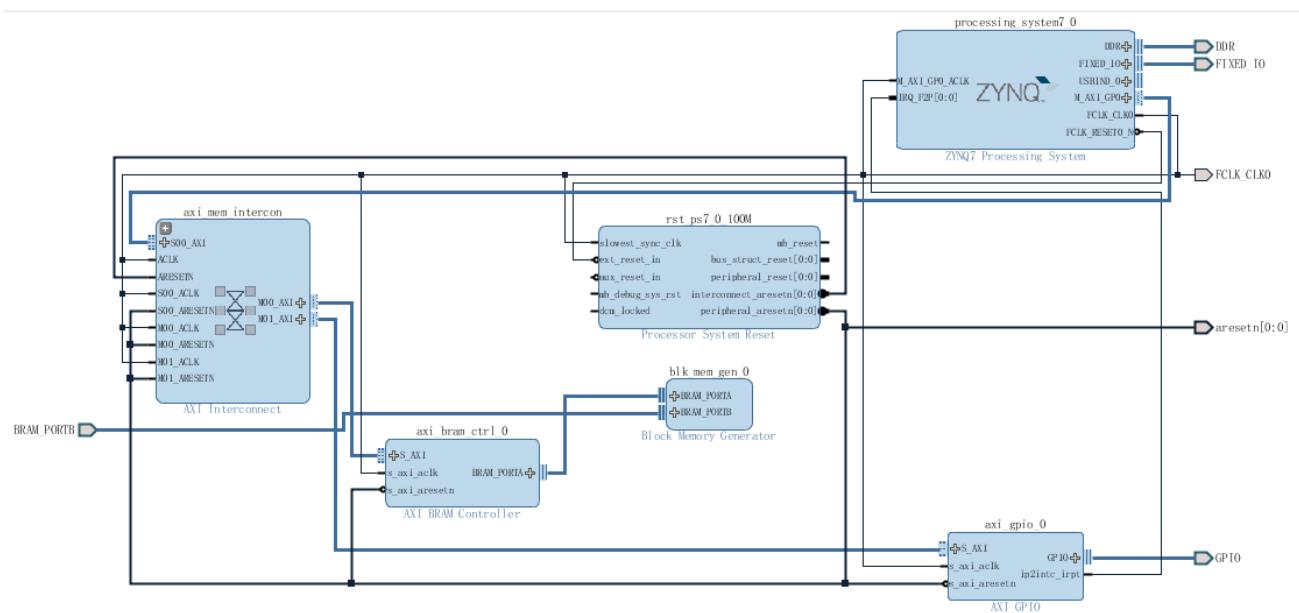
13.2 基本原理

在本例程中，在 PL 端设计了 1 个 4KB（位宽 32，深度 1024）的 BRAM。首先，PS 通过 M_AXI_GP 口向 BRAM 中 1024 个地址依次存入 1024 个 32 位的数据。PS 每向 BRAM 完成写入 1 个 32 位数据后通过 AXI GPIO 输出 1 个上升沿信号，PL 捕获上升沿后立即将 PS 写入的 32 位数据读出，然后加 2，再存入原地址中。存储完成后，PL 通过 AXI GPIO 向 PS 输入 1 个翻转信号，每翻转 1 次，AXI GPIO 便向 PS 触发 1 次中断。PS 触发中断后，再从 BRAM 中读出该数据，判断是否被加了 2，若不一致，则报错。

以上过程重复 1024 次，便将 BRAM 的所有地址都进行了遍历。之后，则不断重复这个过程。

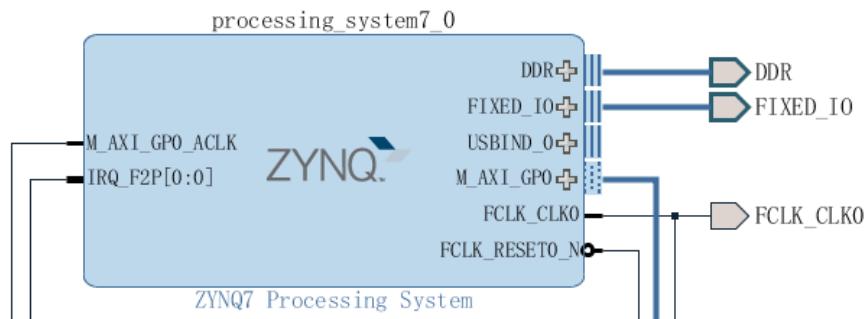
13.3 PL 部分设计

13.3.1 IP 连线图



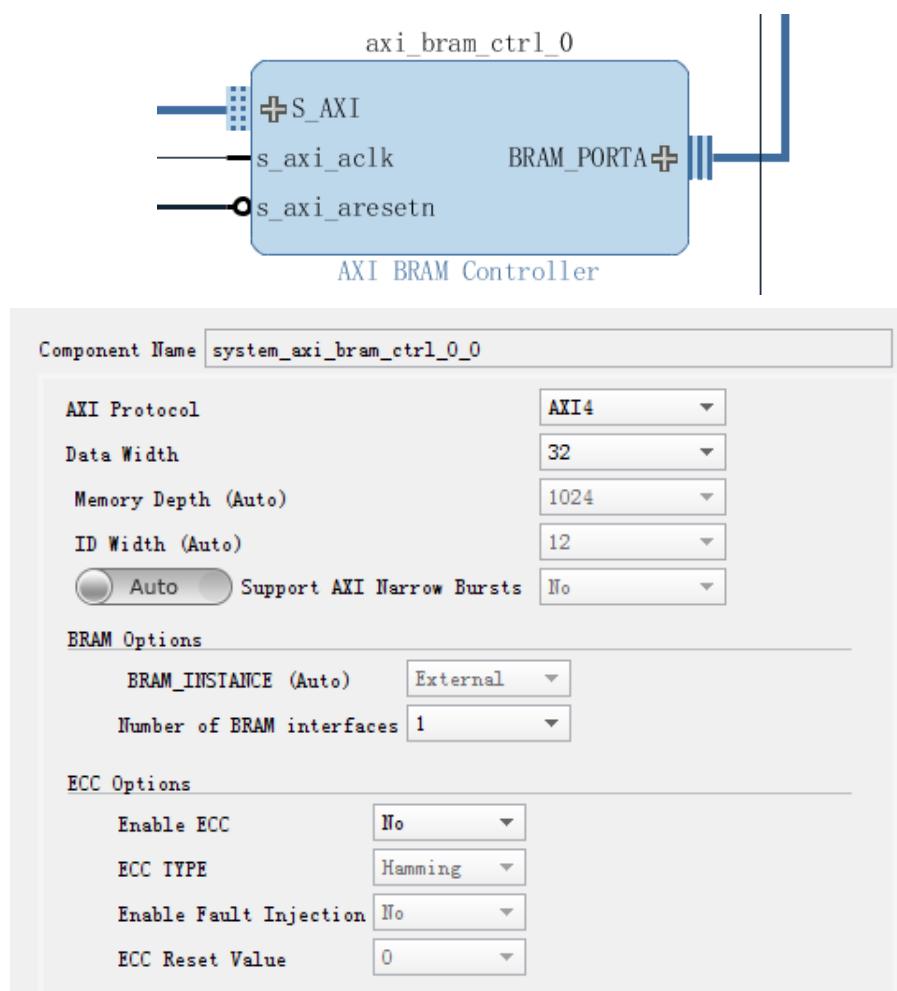
13.3.2 PS 配置

PS 的配置如下图所示。使能 M_AXI_GP0 口，将 FCLK_CLK0 设为 100MHz，使能 PL 至 PS 的中断。



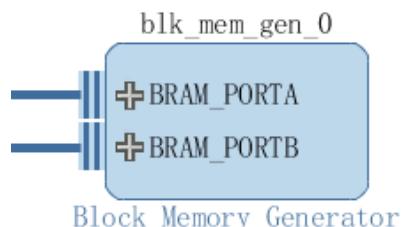
13.3.3 AXI BRAM Controller

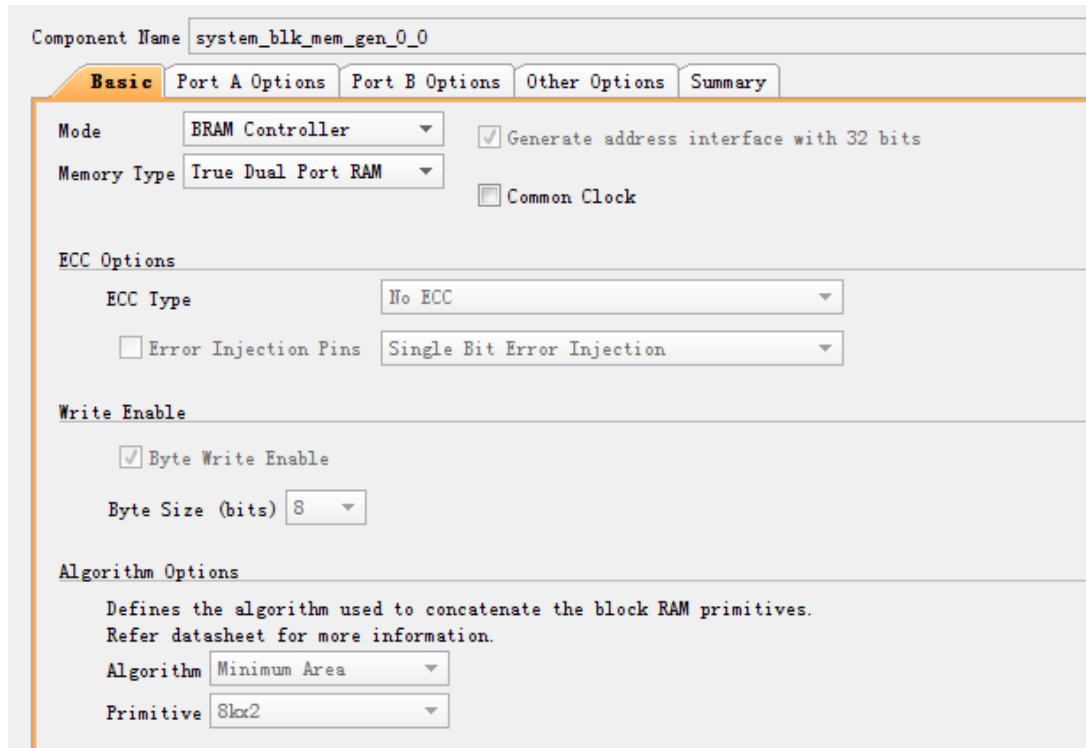
AXI BRAM Controller 是本例程中的关键，该 IP 核连接 PS 的 M_AXI_GP0 口和 BRAM，完成 AXI 接口至 BRAM 接口的转换。设置如下图所示。



13.3.4 Block Memory Generator

添加 BRAM，将 BRAM 设置为双口 RAM，将 PORTA 与 AXI BRAM Controller 连接，PS 通过 PORTA 读写 BRAM，另外，将 PORTB 引出至外部模块，PL 通过 PORTB 读写 BRAM。如下图所示。

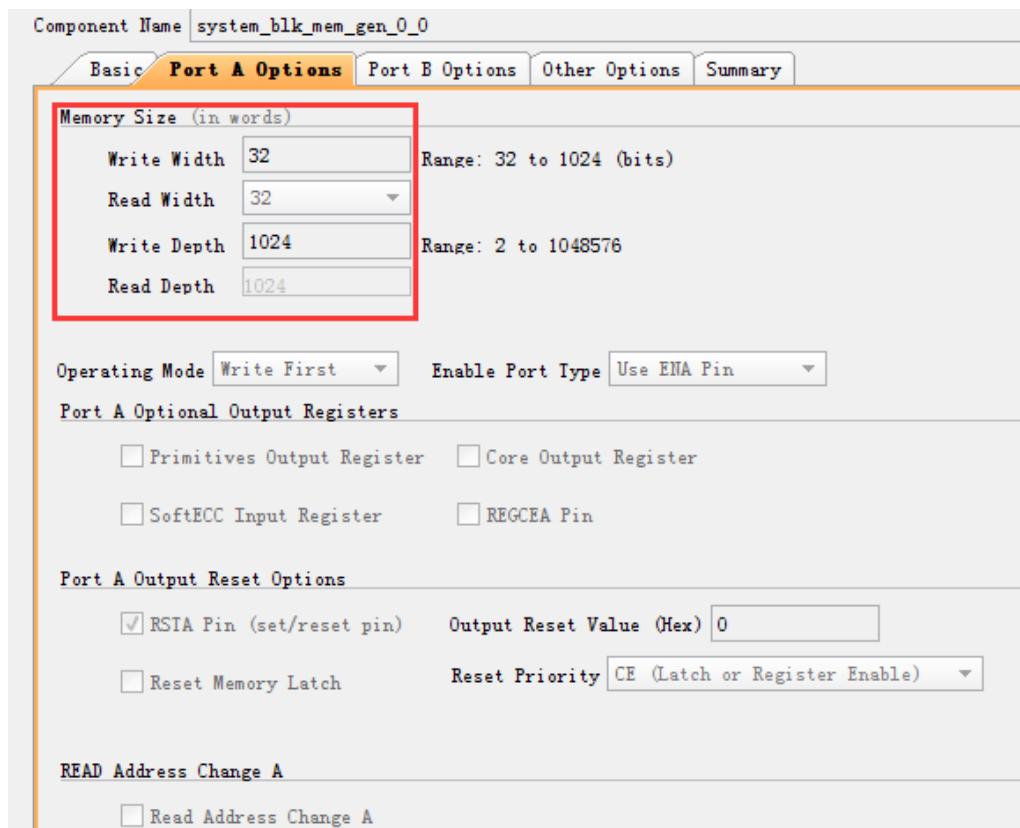




由于要与 AXI BRAM Controller 进行连接，BRAM 接口的位宽固定为 32 位。BRAM 的深度无法在该 IP 中进行设置，需要在所有模块连接完成后，在 Address Editor 里对 AXI BRAM Controller 的地址范围进行设置，本例程中设置为 4KB。该地址范围即对应 BRAM 的深度，如下图所示。

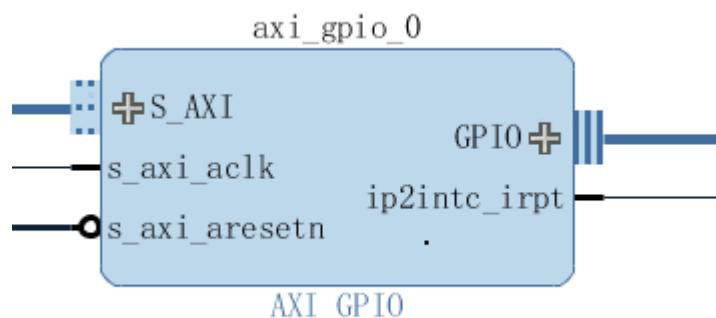
Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	4K	0x4000_0FFF
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF

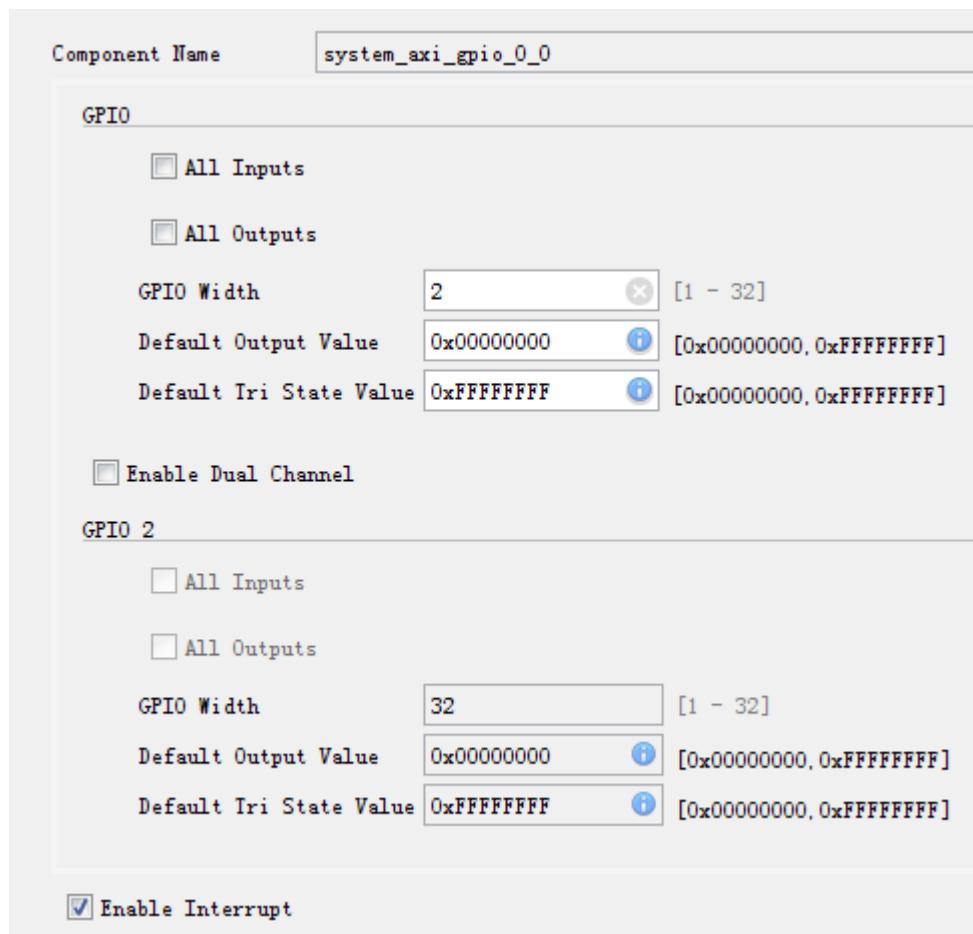
例如， $4\text{KB} = 32\text{bit} \times 1024$ ，因此 BRAM 的深度为 1024。如下图所示。



13.3.5 AXI GPIO

添加 AXI GPIO，位宽设为 2，使能中断，将中断输出与 PS 的中断接口连接，设置如下图所示。其中 GPIO0 作为 PS 向 PL 输出的 PS BRAM 写入完成信号，对于 PL 而言，上升沿有效。GPIO1 作为 PL 向 PS 输入的 BRAM 写入完成信号，该信号为翻转信号，每次翻转向 PS 产生 1 次中断。





13.4 逻辑设计

PL 部分逻辑设计主要包括以下几个过程：

检测 AXI GPIO 输出的 GPIO0 的上升沿；

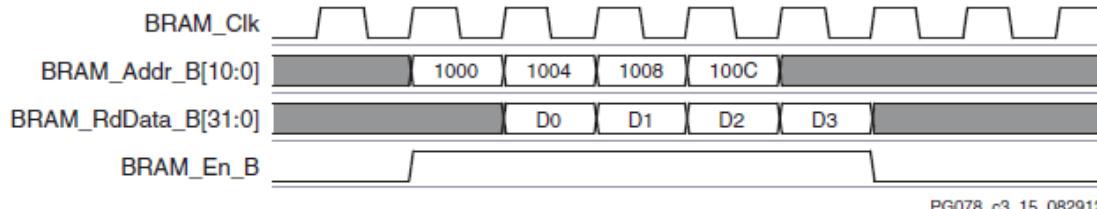
若检测到 GPIO0 的上升沿，则从 BRAM 的某 1 个地址中读出 1 个 PS 写入 32 位数据，然后加 2，存入原地址中；

1 个 32 数据存储完毕后，将 AXI GPIO 的输入信号 GPIO1 进行翻转，告知 PS 一次数据读写完成。

不断循环上述过程，依次遍历 BRAM 中 0~4092 的地址范围。

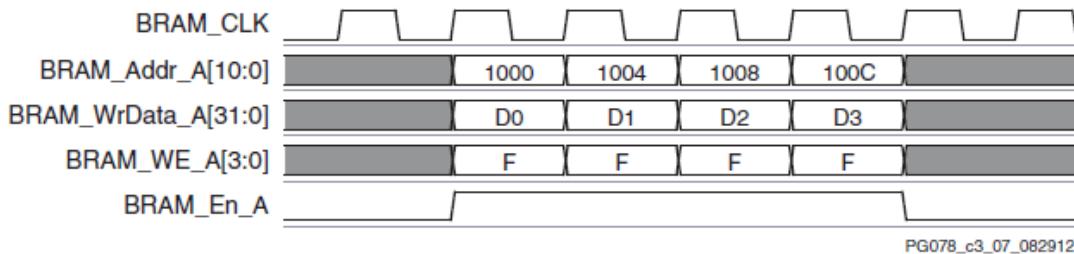
13.4.1 BRAM 读时序

BRAM 读时序如下图所示。



13.4.2 BRAM 写时序

BRAM 写时序如下图所示。



13.5 PS 程序设计

所有的驱动程序文件均包含在 c_driver 文件夹中。

13.5.1 main 函数

main 函数的完成的功能如下：

配置 AXI GPIO 及其中断；

初始化中断控制器及系统中断；

依次向 BRAM 所对应的地址写入 1024 个 32 位整型数据，每写入 1 个数据等待 PL 的读写完成中断来临后继续写入下一个；

依次从 BRAM 所对应的地址读出 1024 个 32 位整型数据，并判断是否被加了 2，若比对不一致则报错。

13.5.2 GPIO 输入输出

在 main 函数调用 Gpiopl_init() 函数初始化 AXI GPIO，设置 2 个 GPIO 的方向，其中 GPIO[0] 为输出，GPIO[1] 为输入。并将 GPIO[0] 置为 0。每个 GPIO 的宏定义在 gpiopl_intr.h 中，如下所示。

```
#define PS_BRAM_MASK          0x00000001
#define PL_INTR_MASK           0x00000002
```

通过 Gpiopl_Setup_Intr_System() 初始化并使能 AXI GPIO 的输入中断，GPIO[1] 的输入发生 1 次改变将触发 1 次中断，GpioplIntrHandler() 为 GPIO 的中断函数。

GpioplIntrHandler() 函数将 PL 读写 BRAM 完成中断标志位 pl_bram_flag 置 1，该信号将在 PS 写入 BRAM 时使用。

13.5.3 BRAM 数据写入

每一个循环 PS 向 BRAM 写入 1024 个 32bit 整型数据，每次循环后将下一次写入的 1024 个数据都加 1。因此，第 1 次写入 BRAM 的数据为 0~1023，第 2 次写入的为

1~1024，第3次为2~1025，以此类推。BRAM写入通过如下函数完成：

```
Xil_Out32(XPAR_BRAM_0_BASEADDR + 4*i, write_data);
```

由于在ZYNQ中最小可寻址单元为字节，因此1个32位数据需占用4个地址，每次写入的地址都需要加4。

每次写入完成后，拉高GPIO0，通过如下代码完成：

```
XGpio_DiscreteWrite(&Gpio, 1, PS_BRAM_MASK);
```

然后，PS等待PL完成BRAM中该地址32bit数据的读写，PL翻转GPIO1使AXI GPIO产生中断，代码如下所示：

```
while(!pl_bram_flag);
```

```
pl_bram_flag = 0;
```

PL完成BRAM读写后，PS拉低GPIO0，通过如下代码完成：

```
XGpio_DiscreteWrite(&Gpio, 1, ~PS_BRAM_MASK);
```

循环1024次，完成1024个数据的写入。

13.5.4 BRAM数据读出

当PS完成1024个数据的写入，此时PL也完成了1024个数据的读出、加2、写入工作。因此，PS需要依次将1024个数据读出进行比对，验证PL给每个数据加2的正确性，因此，第1次读出的1024个数据应该为2~1025，第2次为3~1026，第3次为4~1027，以此类推。若比对出现不一致，则通过串口进行报错。代码如下：

```
for(i      = 0;      4*i      < (XPAR_BRAM_0_HIGHADDR -  
XPAR_BRAM_0_BASEADDR); i++)  
{  
    read_data = Xil_In32(XPAR_BRAM_0_BASEADDR + 4*i);  
    //xil_printf("data at address %d is %d\r\n", 4*i, read_data);  
    /*compare data read from bram if they are add by 2*/  
    if(read_data != (i + 2))  
        xil_printf("error: data at address %d should be %d, but is %d\r\n", 4*i,  
(i + 2), read_data);  
}
```

其中，可通过xil_printf("data at address %d is %d\r\n", 4*i, read_data);查看每一次从BRAM读出的数据的具体值，若无需使用则可注释掉。

13.6 程序测试

程序测试串口打印信息如下图所示。

第1次读出的1024个数据。

Connected to: Serial (COM1)

```
data at address 0 is 2
data at address 4 is 3
data at address 8 is 4
data at address 12 is 5
data at address 16 is 6
data at address 20 is 7
data at address 24 is 8
data at address 28 is 9
data at address 32 is 10
data at address 36 is 11

data at address 4064 is 1018
data at address 4068 is 1019
data at address 4072 is 1020
data at address 4076 is 1021
data at address 4080 is 1022
data at address 4084 is 1023
data at address 4088 is 1024
data at address 4092 is 1025
data at address 0 is 3
data at address 4 is 4
data at address 8 is 5
data at address 12 is 6
data at address 16 is 7
data at address 20 is 8
data at address 24 is 9
data at address 28 is 10
data at address 32 is 11
data at address 36 is 12
data at address 40 is 13
data at address 44 is 14
data at address 48 is 15
data at address 52 is 16
data at address 56 is 17
data at address 60 is 18
```

第2次读出的1024个数据。

```
data at address 4060 is 1018
data at address 4064 is 1019
data at address 4068 is 1020
data at address 4072 is 1021
data at address 4076 is 1022
data at address 4080 is 1023
data at address 4084 is 1024
data at address 4088 is 1025
data at address 4092 is 1026
data at address 0 is 4
data at address 4 is 5
data at address 8 is 6
data at address 12 is 7
data at address 16 is 8
data at address 20 is 9
data at address 24 is 10
data at address 28 is 11
data at address 32 is 12
data at address 36 is 13
data at address 40 is 14
data at address 44 is 15
data at address 48 is 16
data at address 52 is 17
data at address 56 is 18
```

第3次读出的1024个数据。

```
data at address 4052 is 1017  
data at address 4056 is 1018  
data at address 4060 is 1019  
data at address 4064 is 1020  
data at address 4068 is 1021  
data at address 4072 is 1022  
data at address 4076 is 1023  
data at address 4080 is 1024  
data at address 4084 is 1025  
data at address 4088 is 1026  
data at address 4092 is 1027  
data at address 0 is 5  
data at address 4 is 6  
data at address 8 is 7  
data at address 12 is 8  
data at address 16 is 9  
data at address 20 is 10  
data at address 24 is 11  
data at address 28 is 12  
data at address 32 is 13  
data at address 36 is 14  
data at address 40 is 15  
data at address 44 is 16  
data at address 48 is 17
```

后面不作一一列举。

13.7 课后习题

本课读写的速度相对较慢，读者可以尝试修改程序，实现批量数据的读写。

CH14_EMIO 光电通信-FEP 子卡的使用

14.1 概述

目前，在 ZYNQ 中进行以太网开发的方案，大部分都是基于通过 PS 的 MIO 以 RGMII 接口连接外部 PHY 芯片的方式。但是，由于使用 PS 的 MIO 只能以 RGMII 接口连接外部 PHY 芯片，这就限制了支持其他接口 PHY 芯片的使用，如 GMII、SGMII、MII 等等。因此，若将 PS 内部的以太网控制器 ENET0/ENET1 通过 EMIO 的方式扩展至 PL，便可通过 PL 连接更多种类的接口，从而增加了设计的灵活性。

本例程基于米联电子设计的光电双口扩展子卡 FEP_ETH_SFP_CARD，设计了一种通过 EMIO 实现 PL 连接外部 RGMII 接口的 PHY 芯片，实现 PS LWIP 网络通信的方案。扩展子卡使用了 88E1512 芯片，可接光口或电口，且为自适应。

本例程基于 Vivado 2016.4 版本开发，参考了 fpgadeveloper 工程师的应用工程：

<https://github.com/fpgadeveloper/ethernet-fmc-zynq-gem>。

14.2 基本原理

本例程将 PS 的 ETH1 通过 EMIO 方式引出，通过 EMIO 引出的 ETH1 为 GMII 接口，将其与 GMII to RGMII IP 核连接后转换成 RGMII 接口，然后与外部子卡中的 88E1512 芯片连接。在 PS 端通过 SDK 自带的 lwip echo server 例程通过子卡，分别以 RJ45 电口和 SFP 电口两种方式与 PC 机实现 TCP 网络通信。

14.2.1 88E1512

子卡所使用的 88E1512 芯片支持电口和光口，当其上电复位后，自动进入 RGMII to Copper 以及 RGMII to 1000BASEX 自适应状态，这是由其如下图寄存器值决定。当需要使用电口时，将网线与子卡的 RJ45 连接，当需要使用光口时，在子卡的 SFP 屏蔽笼中插入光模块即可。

Fiber/Copper Auto-Selection

The device has a patented feature to automatically detect and switch between fiber and copper cable connections. The auto-selection operates in one of three modes: Copper /1000BASE-X, Copper/100BASE-FX, and Copper/SGMII Media Interface.

Register 20_18.2:0 and register 20_18.6 select the Fiber/Copper auto media modes of operation. See [Table 33](#) for details.

Table 33: Fiber/Copper Auto-media Modes of Operation

Reg 20_18.2:0	Reg 20_18.6	Auto-media Modes of Operation
110	X	Copper/SGMII media
111	0	Copper/1000BASE-X
011	1	Copper/100BASE-FX

The device monitors the signals of the S_INP/N and the MDIP/N[3:0] lines. If a fiber optic cable is plugged in, the device will adjust itself to be in fiber mode. If an RJ-45 cable is plugged in, the device will adjust itself to be in copper mode. If both cables are connected then the first media to establish link, or the preferred media will be enabled. The media which is not enabled will be automatically turned off to save power. If the link on the first media is lost, then the inactive media will be powered up, and both media will once again start searching for link.

Table 220: General Control Register 1
Page 18, Register 20

Bits	Field	Mode	HW Rst	SW Rst	Description
15	Reset	R/W, SC	0x0	SC	Mode Software Reset. Affects page 6 and 18 Writing a 1 to this bit causes the main PHY state machines to be reset. When the reset operation is done, this bit is cleared to 0 automatically. The reset occurs immediately. 1 = PHY reset 0 = Normal operation
14:13	Reserved	R/W	0x0	Retain	Set to 0s.
12:10	Reserved	R/W	0x0	Retain	Reserved for future use.
9:7	Reserved	R/W	0x4	Retain	Set to 100
6	Auto-Media Detect (AMD) 100BASE-FX/ 1000BASE-X	R/W	0x0	Retain	This bit selects the fiber auto-media modes as follows: 1 = mode 011 is AMD between copper and 100BASE-FX 0 = mode 111 is AMD between copper and 1000BASE-X
5:4	Auto Media Detect Preferred Media	R/W	0x0	Retain	00 = Link on first media 01 = Copper Preferred 10 = Fiber Preferred 11 = Reserved
3	Reserved	R/W	0x0	Update	Set to 0
2:0	MODE[2:0]	R/W	See Descr.	Update	Changes to this bit are disruptive to the normal operation; therefore, any changes to these registers must be followed by a software reset to take effect. 1512 device comes up in MODE[2:0] =0x7 on hardware reset. 000 = RGMII (System mode) to Copper 001 = SGMII (System mode) to Copper 010 = RGMII (System mode) to 1000BASE-X 011 = RGMII (System mode) to 100BASE-FX 100 = RGMII (System mode) to SGMII (Media mode) 101 = Reserved 110 = RGMII (System mode) to Auto Media Detect Copper/SGMII (Media mode) 111 = RGMII (System mode) to Auto Media Detect Copper/1000BASE-X/100BASE-FX (see 20_18.6)

14.2.2 88E1512 RGMII 接口时序

88E1512 芯片 RGMII 接口的时序存在延迟和非延迟 2 种模式，由其内部的一个寄存器值来决定，如下图所示。

Table 121: MAC Specific Control Register 2
Page 2, Register 21

Bits	Field	Mode	HW Rst	SW Rst	Description
12:7	Reserved		0x20	0x20	Reserved.
6	Default MAC interface speed (MSB)	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. Also, used for setting speed of MAC interface during MAC side loopback. Requires that customer set both these bits and force speed using register 0 to the same speed. MAC Interface Speed during Link down. Bits 6, 13 00 = 10 Mbps 01 = 100 Mbps 10 = 1000 Mbps
5	RGMII Receive Timing Control	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. 1 = Receive clock transition when data stable 0 = Receive clock transition when data transitions
4	RGMII Transmit Timing Control	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. 1 = Transmit clock internally delayed 0 = Transmit clock not internally delayed

芯片上电通过 RESET 引脚复位后,寄存器中决定 RGMII 输入输出时序关系的 bit 位的值都是 1。也就是说,上电复位后芯片的 RGMII 接口均为延迟时序模式。延迟模式是指 88E1512 芯片在其内部给输出的接收时钟信号 RX_CLK 和输入的发送的时钟信号 TX_CLK 增加了 2ns 左右的延时。这是为了避免通过 PCB 绕线的方式增加时钟信号的延时,使 RX_CLK 和 TX_CLK 都能与相应 RXD 和 TXD 数据窗的中心对齐,从而使得 RGMII 接口数据的建立和保持时间达到余量最大的状态。

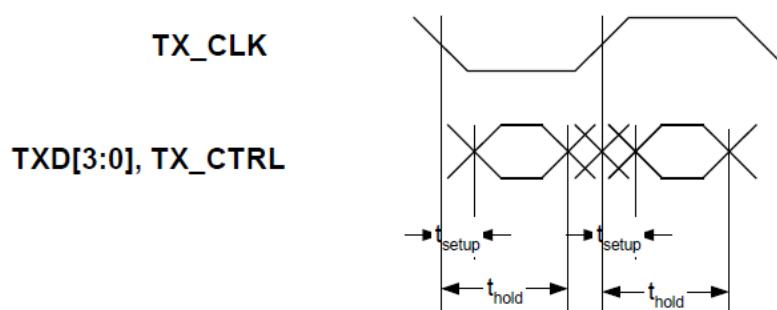
因此,此时 RGMII 接口发送部分信号的时序关系如下图所示。

4.10.2.2 PHY Input - TX_CLK Delay when Register 21_2.4 = 1

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.4 = 1 (add delay)	-0.9			ns
t_{hold}		2.7			ns

Figure 37: TX_CLK Delay Timing - Register 21_2.4 = 1 (add delay)



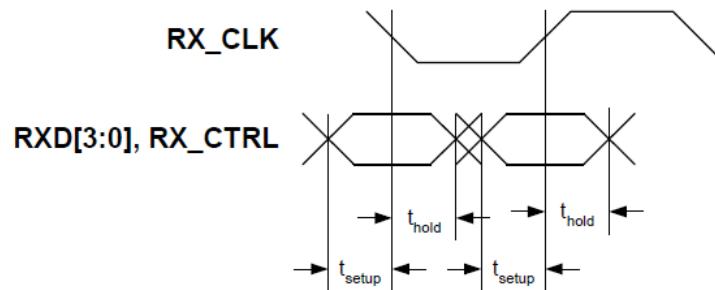
RGMII 接口接收部分信号的时序关系如下图所示。

4.10.2.4 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

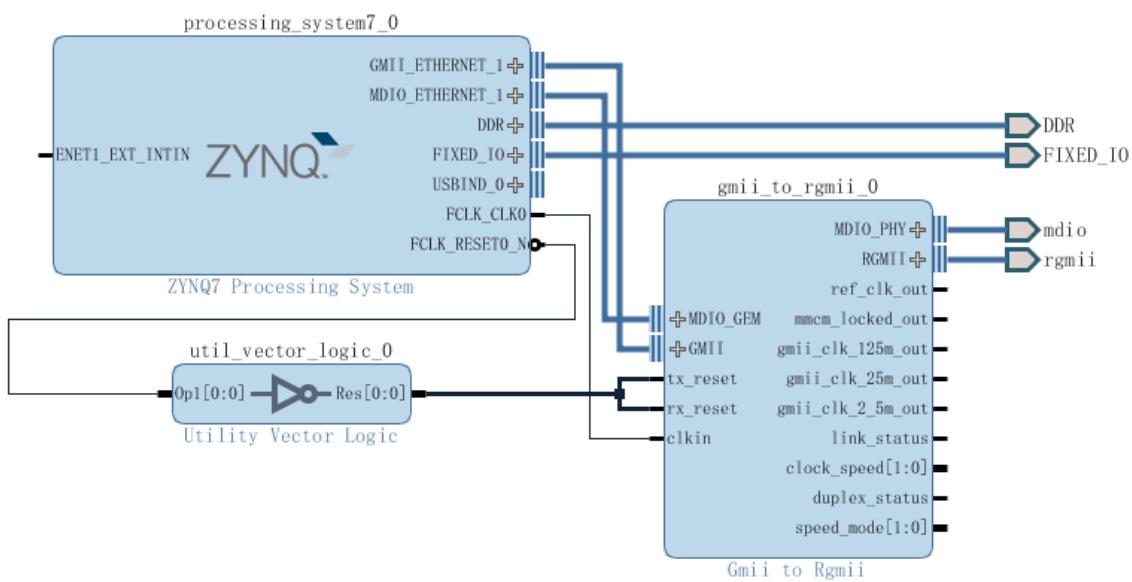
Symbol	Parameter	Min	Typ	Max	Units
t _{setup}	Register 21_2.5 = 1 (add delay)	1.2			ns
t _{hold}		1.2			ns

Figure 39: RGMII RX_CLK Delay Timing - Register 21_2.5 = 1 (add delay)



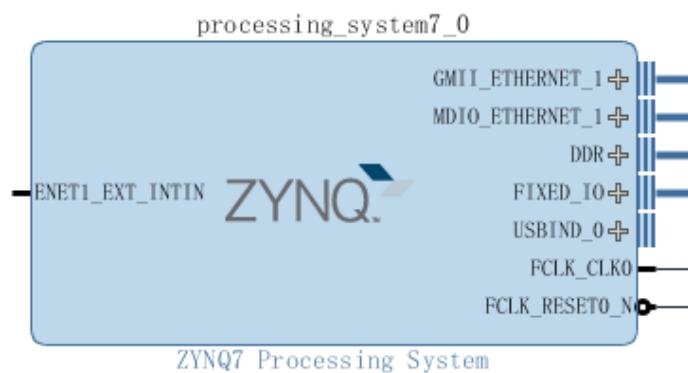
14.3 PL 部分设计

14.3.1 IP 连线图



14.3.2 ZYNQ PS 设置

将 ETH1 及其 DMIO 通过 EMIO 引出，将 FCLK_CLK0 设置为 200M，作为 GMII to RGMII IP 核内部 IDELAYCTRL 的参考时钟，FCLK_RESET0_N 通过 Utility Vector Logic 生成的非门后作为 GMII to RGMII IP 核的复位信号。如下图所示。



14.3.3 GMII to RGMII

14.3.3.1 PS ETH EMIO

PS 的 ETH 通过 EMIO 引出后为标准的 GMII 接口，如下图所示。

Zynq-7000 AP SoC Device

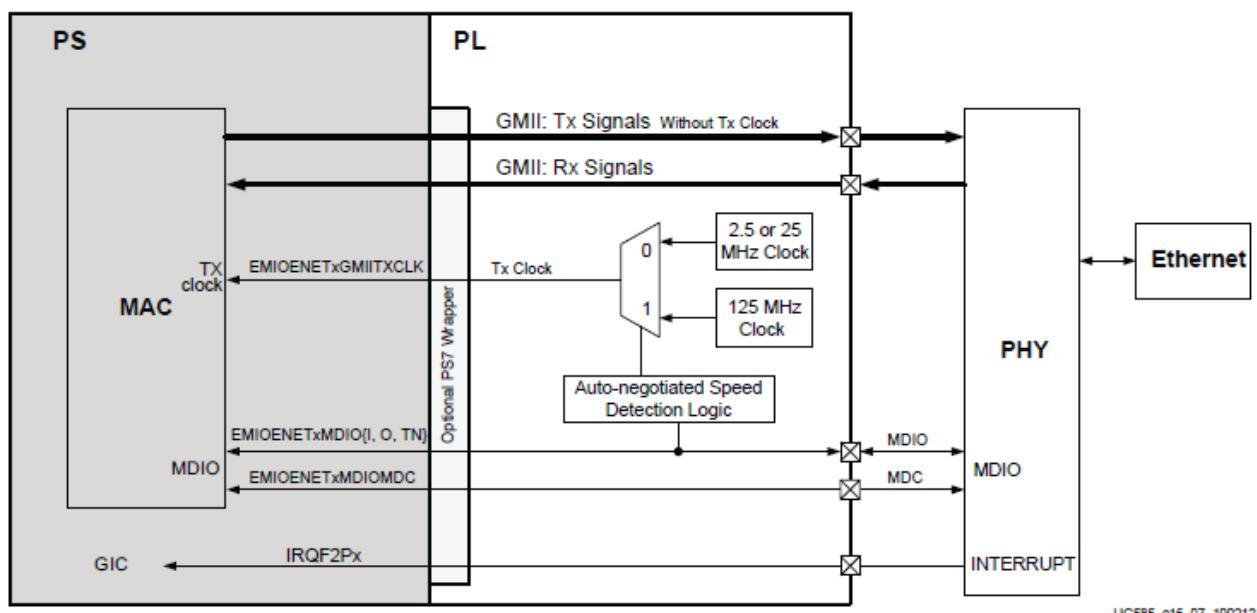


Figure 16-8: GMII Interface via EMIO Connections

需要通过 IP 核 GMII to RGMII 将 GMII 接口转换为 RGMII 接口，才能与 PHY 芯片 88E1512 连接。连接原理如下图所示。

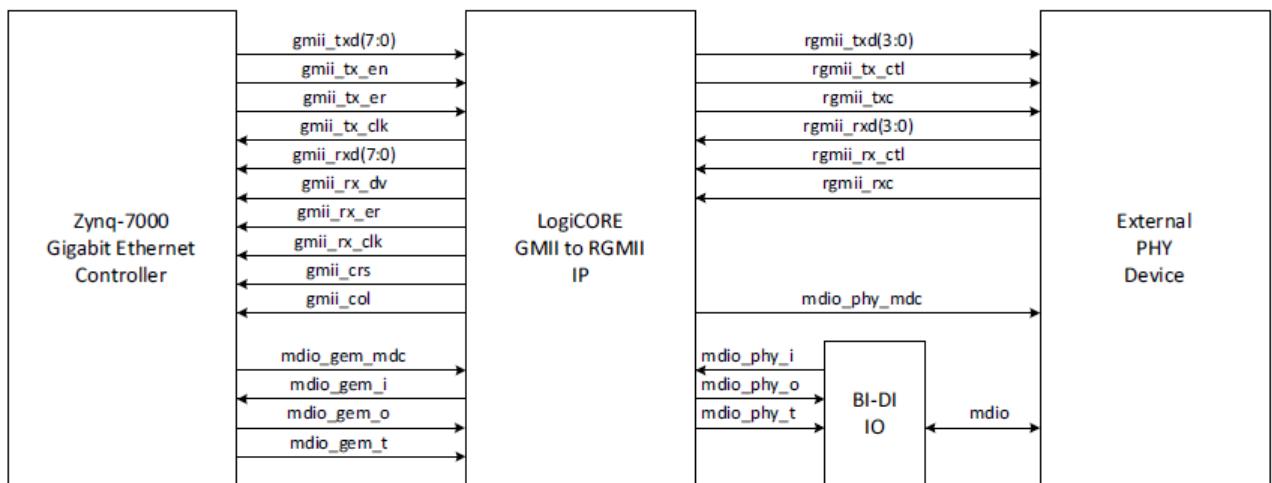
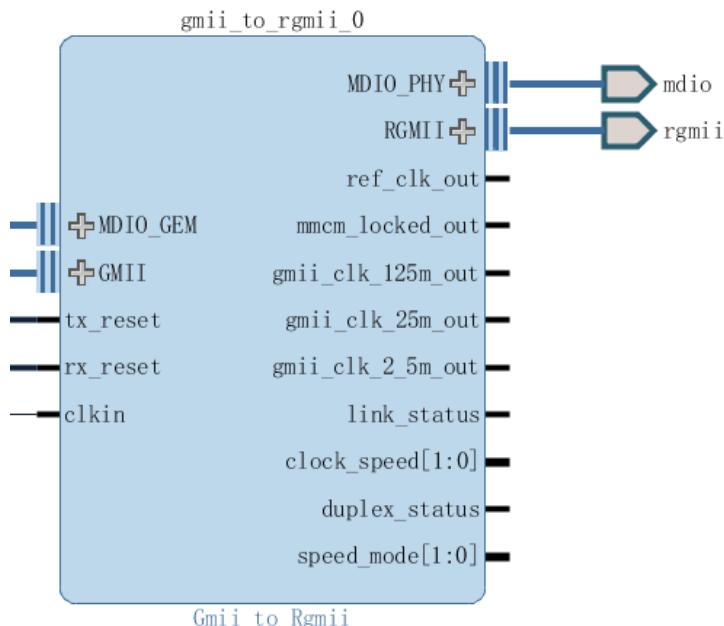


Figure 2-1: GMII to RGMII Core Ports and Interfaces

14.3.3.2 IP 核设置

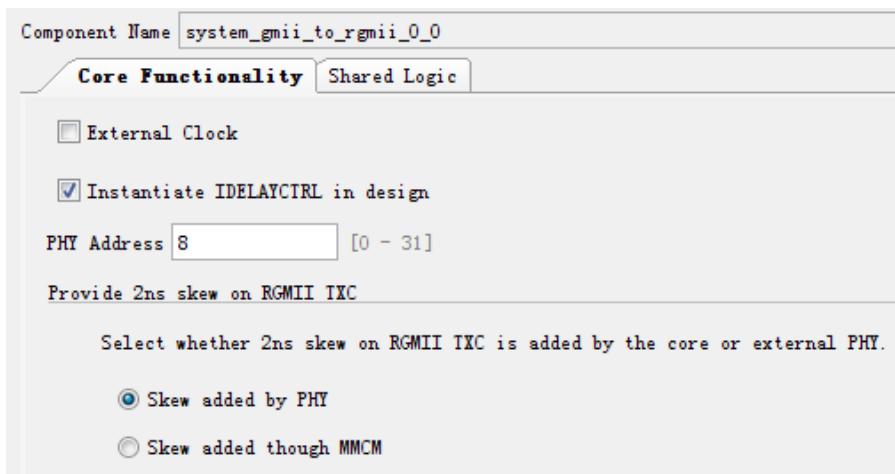
GMII to RGMII IP 核的设置如下图所示。



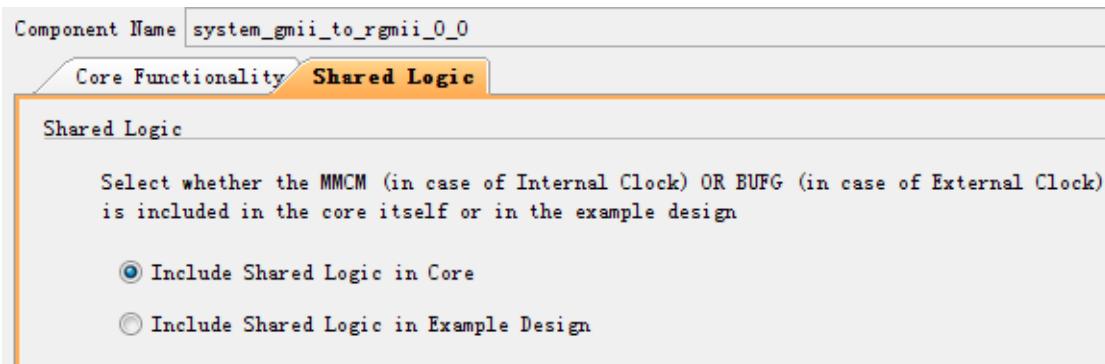
在 HR BANK 中，IP 核中 RGMII 接口的接收数据信号和控制信号需要通过 IDELAYE2 来调整信号输入延时，使其时序满足建立和保持时间约束。因此需要在 IP 核包含与 IDELAYE2 相关的 IDELAYCTRL，用来校准 IDELAYE2 每个延时 tap 的延时值。

将本 IP 核的 PHY address 设置为 8（该值可任意设置，但不能与 88E1512 的 PHY address 相同，否则将产生冲突使 IP 核工作异常，子卡中 88E1512 的 PHY address 为 0）。

由于 88E1512 发送信号延时由芯片内部提供，因此，选择 2ns 的延时 skew 由 PHY 增加。



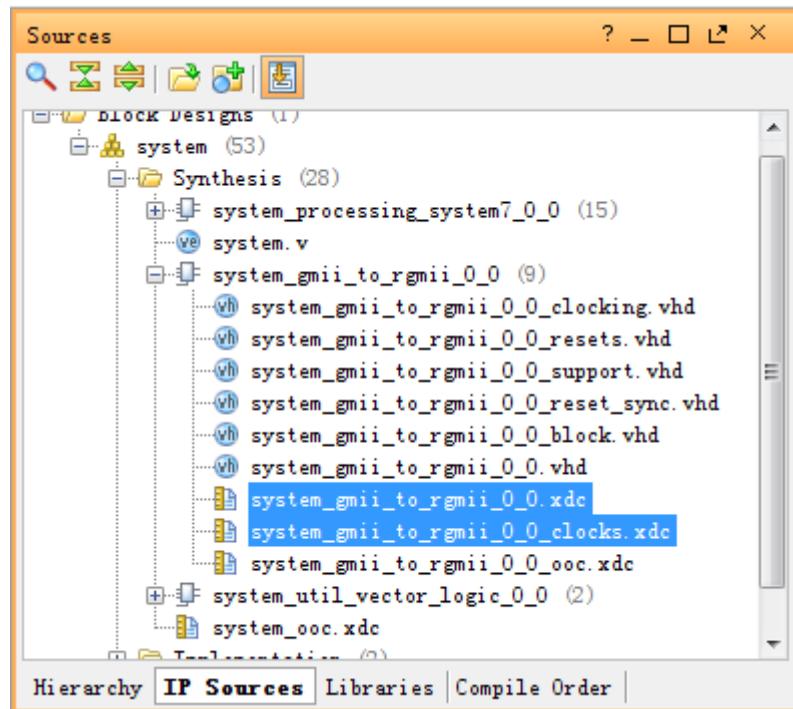
选择 shared logic 包含在 IP 核内部。



14.3.4 时序约束

本例程中，时序约束主要是针对 RGMII 接口进行 input delay 和 output delay 的约束，以及 IDELAYE2 延时 tap 数的设置，使 RGMII 接口的满足建立和保持时间的要求，从而达到时序收敛。

对于 input delay 和 output delay 的约束，GMII to RGMII IP 核所自带的 system_gmii_to_rgmii_0_0_clocks.xdc 文件中已经包含了默认设置。对于 IDELAYE2 延时 tap 数的设置，在 system_gmii_to_rgmii_0_0.xdc 中包含了默认模板。这两个 xdc 文件位置如下图所示。



14.3.4.1 input delay 约束

system_gmii_to_rgmii_0_0_clocks.xdc 文件中，关于 input delay 的约束如下所示。约束范围为：-1.5~15.8ns。

```
# define a virtual clock to simplify the timing constraints
create_clock -name system_gmii_to_rgmii_0_0_rgmii_rx_clk -period 8

# Identify RGMII Rx Pads only.
# This prevents setup/hold analysis being performed on false inputs,
# eg, the configuration_vector inputs.

set_input_delay -clock [get_clocks system_gmii_to_rgmii_0_0_rgmii_rx_clk] -max -1.5 [get_ports {rgmii_rxd[*] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks system_gmii_to_rgmii_0_0_rgmii_rx_clk] -min -2.8 [get_ports {rgmii_rxd[*] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks system_gmii_to_rgmii_0_0_rgmii_rx_clk] -clock_fall -max -1.5 -add_delay [get_ports {rgmii_rxd[*] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks system_gmii_to_rgmii_0_0_rgmii_rx_clk] -clock_fall -min -2.8 -add_delay [get_ports {rgmii_rxd[*] rgmii_rx_ctl}]
```

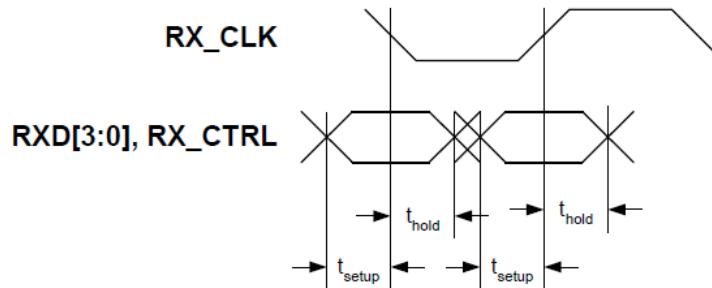
在 15.2 节中已经对 88e1512 芯片的 RGMII 接口时序进行了介绍，对于 RX 接口，如下图。

4.10.2.4 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.5 = 1 (add delay)	1.2			ns
t_{hold}		1.2			ns

Figure 39: RGMII RX_CLK Delay Timing - Register 21_2.5 = 1 (add delay)



按照上图中的 setup time 和 hold time, input delay 的-min 应该为- (4-1.2) =-15.8ns, -max 应该为-1.2ns。显然, IP 核自带 input delay 的约束范围为-1.5ns~15.8ns, 比我们计算的结果-1.2ns~15.8ns 略为宽松, 用于 setup time 计算所需的-max 时间小了 0.3ns, 为了保证时序严格收敛, 需要将 system_gmii_to_rgmii_0_0_clocks.xdc 文件中 set_input_delay -max 的-1.5 全部改为-1.2。需要注意的是, 这些 xdc 文件由 IP 核自动生成, 每次重新配置 IP 后, vivado 都会重新生成 IP 的 output product, 因此会将被修改的 xdc 文件覆盖还原, 需要手动重新再次修改 xdc 文件才行。

14.3.4.2 output delay 约束

system_gmii_to_rgmii_0_0_clocks.xdc 文件中, 关于 output delay 的约束如下所示。约束范围为: -1.0~15.6ns。

```
create_generated_clock -add -name rgmii_tx_clk -divide_by 1 -source [get_pins -of [get_cells -hier -filter {name =~ *rgmii_txc_out}]

set_output_delay -clock [get_clocks rgmii_tx_clk] -max -1.0 [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
set_output_delay -clock [get_clocks rgmii_tx_clk] -min -2.6 [get_ports {rgmii_txd[*] rgmii_tx_ctl}] -add_delay
set_output_delay -clock [get_clocks rgmii_tx_clk] -clock_fall -max -1.0 [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
set_output_delay -clock [get_clocks rgmii_tx_clk] -clock_fall -min -2.6 [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
```

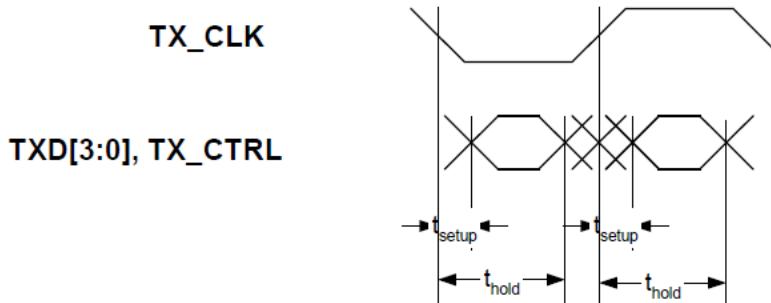
对于 88e1512 RGMII 的 TX 接口, 时序关系如下图。

4.10.2.2 PHY Input - TX_CLK Delay when Register 21_2.4 = 1

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.4 = 1 (add delay)	-0.9			ns
t_{hold}		2.7			ns

Figure 37: TX_CLK Delay Timing - Register 21_2.4 = 1 (add delay)



照上图中的 setup time 和 hold time, output delay 的-min 应该为-0.9ns, -max 应该为-15.7ns。显然, IP 核自带 output delay 的约束范围为-1.0ns~15.6ns, 比我们计算的结果-0.9ns~15.7ns 略为宽松, 用于 setup time 计算所需的-max 时间小了 0.1ns, 用于 hold time 计算所需的-min 时间大了 0.1ns。为了保证时序严格收敛, 需要将 system_gmii_to rgmii_0_0_clocks.xdc 文件中 set_output_delay -max 的 -1.0 全部改为-0.9, -min 的-15.6 全部改为-15.7。

14.3.4.3 IDELAYE2 延时设置

在 GMII to RGMII 的 IP 核内部为 rgmii_rx_ctl 和 rgmii_rxd[3:0]端口都连接了 IDELAYE2 模块, 用来为这些信号进入 PL 内部前引入额外的延时。IDELEYE2 延时值的大小与 xdc 中的 input delay 约束是否能收敛密切相关。一般情况下, 当 input delay 约束的 setup time 出现违例, 应该将 IDELAYE2 的 tap 数减小, 降低延时值; 当 input delay 约束的 hold time 出现违例, 应该将 IDELAYE2 的 tap 数增大, 增加延时值。

system_gmii_to rgmii_0_0.xdc 中包含了设置 IDELAYE2 的 tap 数的模板, 如下图所示。

```
#-----
# To Adjust GMII Rx Input Setup/Hold Timing
# These IDELAY Tap values are given for illustration
# purpose. Please modify as per design requirements
#-----
#set_property IDELAY_VALUE "16" [get_cells -hier -filter {name == *system_gmii_to rgmii_0_0_core/*delay_rgmii_rx_ctl}] ]]
#set_property IDELAY_VALUE "16" [get_cells -hier -filter {name == *system_gmii_to rgmii_0_0_core/*delay_rgmii_rxd*}] ]]
#set_property IDELAY_GROUP "gpr1" [get_cells -hier -filter {name == *system_gmii_to rgmii_0_0_core/*delay_rgmii_rx_ctl}] ]]
#set_property IDELAY_GROUP "gpr1" [get_cells -hier -filter {name == *system_gmii_to rgmii_0_0_core/*delay_rgmii_rxd*}] ]]
#set_property IDELAY_GROUP "gpr1" [get_cells -hier -filter {name == *i_system_gmii_to rgmii_0_0_idelayctrl}]]
```

该段约束默认是被注释的, 默认的延时 tap 数为 16, 我们可以将这段约束复制到工程自己新建的 xdc 文件中, 编译工程后根据时序报告查看 input delay 是否时序收敛, 若存在时序违例, 则需要根据实际情况调整 tap 的值。在本例程中, 针对 miz701n-7010 开发板, 经过尝试后, 最佳的 tap 数为 14。

14.3.4 IO 口

在 block design 自动生成的 system_wrapper.v 文件中, 手动添加 2 个输出端口, 如下所示。

```
output sfp_tx_disable;
output sfp_rate_sel;
```

这两个端口连接到 SFP，将 sfp_tx_disable 置 0，sfp_rate_sel 置 1。如下所示。

```
assign sfp_tx_disable = 1'b0;
assign sfp_rate_sel = 1'b1;
```

14.4 PS 程序设计

所有的驱动程序文件均包含在 c_driver 文件夹中。

14.4.1 LWIP 库修改

在 SDK 2016.4 中所使用的 LWIP 1.4.1 库的版本为 1.7。本例程中，由于 ETH1 通过 EMIO 连接了 GMII to RGMII 这个 IP 核，若直接使用原版 LWIP 库会存在数据收发异常的现象。

原因在于，该 IP 核内部存在 1 个寄存器，如下图所示。

Control Register

This register is 16-bits wide. Its address is 0x10. The composition of this register is similar to the IEEE standard 802.3 MDIO control register 0x0, which is shown in [Table 2-4](#).

Table 2-4: Core-Specific Control Register (Address 0x10)

Bit	Name	Description	R/W
15	Reset	1 = Resets the core and this register 0 = Normal operation	R/W Self-clearing
14	Reserved	Write as 0, ignore on read	R/W
13	Speed Selection (LSB)	6 13 1 1 = Reserved 1 0 = 1,000 Mb/s 0 1 = 100 Mb/s 0 0 = 10 Mb/s	R/W
12:9	Reserved	Write as 0, ignore on read	R/W
8:7	Reserved	Write as 0, ignore on read	R/W
6	Speed Selection (MSB)	6 13 1 1 = Reserved 1 0 = 1,000 Mb/s 0 1 = 100 Mb/s 0 0 = 10 Mb/s	R/W
5:0	Reserved	Write as 0, ignore on read	R/W

本例程中，PS 的 ETH1 的 MDIO 连接到了 GMII to RGMII，PS 需要通过 MDIO 正确配置该寄存器的值，来选择当前 PHY 芯片的工作速率。而 GMII to RGMII IP 核则根据该寄存器的值来切换其于 PHY 芯片连接的 RGMII 接口的时钟频率（125M、25M、15.5M）和数据位宽。因此，若 PS 无法正确配置该寄存器，则 IP 核将无法正常工作。

打开 SDK 自带的 LWIP 库中 xemacpsif_physpeed.c 文件，我们可以发现其中 phy_setup() 函数里其实已经包含了当 PS 的 ETH 通过 EMIO 连接 GMII to RGMII 时，通过 MDIO 配置该寄存器的代码，如下图所示。

```

#ifndef XPAR_GMII2RGMIICON_0N_ETH0_ADDR
    convphyaddr = XPAR_GMII2RGMIICON_0N_ETH0_ADDR;
    conv_present = 1;
#endif
#ifndef XPAR_GMII2RGMIICON_0N_ETH1_ADDR
    convphyaddr = XPAR_GMII2RGMIICON_0N_ETH1_ADDR;
    conv_present = 1;
#endif

#ifndef CONFIG_LINKSPEED_AUTODETECT
    link_speed = get_IEEE_phy_speed(xemacpsp, phy_addr);
    if (link_speed == 1000) {
        SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,1000);
        convspeeddupsetting = XEMACPS_GMII2RGMIISPEED1000_FD;
    } else if (link_speed == 100) {
        SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,100);
        convspeeddupsetting = XEMACPS_GMII2RGMIISPEED100_FD;
    } else if (link_speed != XST_FAILURE){
        SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,10);
        convspeeddupsetting = XEMACPS_GMII2RGMIISPEED10_FD;
    } else {
        xil_printf("Phy setup error \r\n");
        return XST_FAILURE;
    }
#elif defined(CONFIG_LINKSPEED1000)
    SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,1000);
    link_speed = 1000;
    configure_IEEE_phy_speed(xemacpsp, phy_addr, link_speed);
    convspeeddupsetting = XEMACPS_GMII2RGMIISPEED1000_FD;
    sleep(1);
#elif defined(CONFIG_LINKSPEED100)
    SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,100);
    link_speed = 100;
    configure_IEEE_phy_speed(xemacpsp, phy_addr, link_speed);
    convspeeddupsetting = XEMACPS_GMII2RGMIISPEED100_FD;
    sleep(1);
#elif defined(CONFIG_LINKSPEED10)
    SetUpSLCRDivisors(xemacpsp->Config.BaseAddress,10);
    link_speed = 10;
    configure_IEEE_phy_speed(xemacpsp, phy_addr, link_speed);
    convspeeddupsetting = XEMACPS_GMII2RGMIISPEED10_FD;
    sleep(1);
#endif
    if (conv_present) {
        XEmacPs_PhphyWrite(xemacpsp, convphyaddr,
        XEMACPS_GMII2RGMIIREG_NUM, convspeeddupsetting);
    }
}

```

然而，由于相关头文件中缺少了两个宏定义：XPAR_GMII2RGMIICON_0N_ETH0_ADDR 和 XPAR_GMII2RGMIICON_0N_ETH1_ADDR，使得通过 MDIO 配置 GMII to RGMII 寄存器的代码不会被编译执行。因此，直接使用自带 SDK 库将无法完成该寄存器的配置，IP 核也将无法正常工作。按理，vivado 中已经设置了 GMII to RGMII 的 IP 核及其地址，导入 SDK 后应该自动生成这两个宏定义，这可能是软件设计时的疏忽之处。为此，我们需要自己修改 LWIP 库来增加这两个缺失的宏定义。

首先，找到 SDK 安装目录下的 LWIP 库的路径，例如：

C:\Xilinx\SDK\2016.4\data\embeddedsw\ThirdParty\sw_service

将 lwip141_v1_7 文件夹复制一份到工程目录的 sdk_repo\bsp 文件夹下，将其重新命名为 lwip141_v1_73。第一步，修改 lwip141_v1_73\data\lwip141.mld 文件（可用 Notepad++ 等编辑器打开），将其中的版本编号

OPTION VERSION = 1.7;

修改为

OPTION VERSION = 1.73;

然后，在其中增加如下字段：

```
BEGIN CATEGORY emio_options
PARAM name = emio_options, desc = "Settings for ETH using EMIO in PL";
PARAM name = use_gmii2rgmii_core_on_eth0, desc = "Settings for ETH0 using GMII to RGMII ip core in PL", type = bool, default = false;
PARAM name = use_gmii2rgmii_core_on_eth1, desc = "Settings for ETH1 using GMII to RGMII ip core in PL", type = bool, default = false;
PARAM name = gmii2rgmii_core_address_on_eth0, desc = "Settings for ETH0's PHY address of GMII to RGMII ip core in PL", type = int, default = 0;
PARAM name = gmii2rgmii_core_address_on_eth1, desc = "Settings for ETH1's PHY address of GMII to RGMII ip core in PL", type = int, default = 0;
PARAM name = use_1000basex_on_88e1512, desc = "Settings for operation mode of 88e1512, 1000/100/10 baset or 1000basex", type = bool, default = false;
END CATEGORY
```

增加这段代码的目的是为了在 SDK 中 BSP 设置里，lwip 参数设置对话框里增加一栏选项 emio_options，其中包含了 5 个子选项，如下图所示。

Configuration for library: lwip141				
Name	Value	Def...	Type	Description
api_mode	RAW ...	RA...	enum	Mode of operation for lwIP (RAW API/Sockets API)
socket_mode_thread_prio	2	2	integer	Priority of threads in socket mode
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axiethernet adapter being use...
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures emaclite adapter being used in...
▷ arp_options	true	true	boolean	ARP Options
▷ debug_options	true	true	boolean	Turn on lwIP Debug?
▷ dhcp_options	true	true	boolean	Is DHCP required?
▷ emio_options				Settings for ETH using EMIO in PL
gmii2rgmii_core_address_on_eth0	0	0	integer	Settings for ETH0's PHY address of GMII to RGMII ip cor...
gmii2rgmii_core_address_on_eth1	8	0	integer	Settings for ETH1's PHY address of GMII to RGMII ip cor...
use_1000basex_on_88e1512	false	false	boolean	Settings for operation mode of 88e1512, 1000/100/10 b...
use_gmii2rgmii_core_on_eth0	false	false	boolean	Settings for ETH0 using GMII to RGMII ip core in PL
use_gmii2rgmii_core_on_eth1	true	false	boolean	Settings for ETH1 using GMII to RGMII ip core in PL
▷ icmp_options	true	true	boolean	ICMP Options
igmp_options	false	false	boolean	IGMP Options
▷ lwip_ip_options	true	true	boolean	IP Options
▷ lwip_memory_options				Options controlling lwIP memory usage
▷ pbuf_options	true	true	boolean	Pbuf Options
▷ stats_options	true	true	boolean	Turn on lwIP statistics?
▷ tcp_options	true	true	boolean	Is TCP required ?
▷ temac_adapter_options	true	true	boolean	Settings for xps-ll-temac/Axi-Ethernet/Gem lwIP adapter
▷ udp_options	true	true	boolean	Is UDP required ?

其中，use_gmii2rgmii_core_on_eth0 和 use_gmii2rgmii_core_on_eth1 分别表示 ETH0 和 ETH1 是否通过 EMIO 连接了 GMII to RGMII IP 核。gmii2rgmii_core_address_on_eth0 和 gmii2rgmii_core_address_on_eth1 分别表示与 ETH0 和 ETH1 所连接的 GMII to RGMII IP 核中所设置的 PHY address，此处的设置需要和 vivado 中的 IP 核设置完全一致。use_1000basex_on_88e1512 表示子卡中的 88E1512 芯片是否处于 1000BASEX 工作模式，即是否使用使用子卡中的 SFP 接口。

接着，打开 lwip141_v1_73\data\lwip141.tcl 文件，在 proc generate_lwip_opts {libhandle} 所在的大括号内其中增加如下字段：

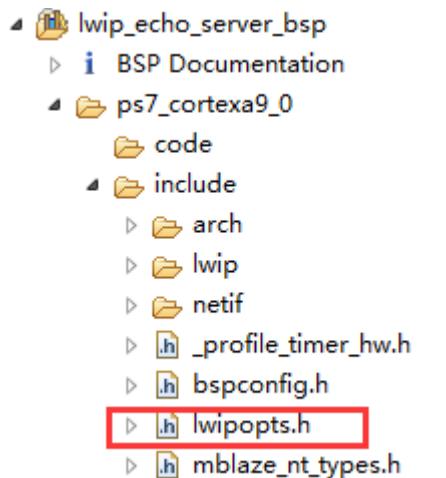
```
# EMIO options
set use_gmii2rgmii_core_on_eth0 [common::get_property CONFIG.use_gmii2rgmii_core_on_eth0 $libhandle]
set use_gmii2rgmii_core_on_eth1 [common::get_property CONFIG.use_gmii2rgmii_core_on_eth1 $libhandle]
set gmii2rgmii_core_address_on_eth0 [common::get_property CONFIG.gmii2rgmii_core_address_on_eth0 $libhandle]
set gmii2rgmii_core_address_on_eth1 [common::get_property CONFIG.gmii2rgmii_core_address_on_eth1 $libhandle]
set use_1000baseX_on_88e1512 [common::get_property CONFIG.use_1000baseX_on_88e1512 $libhandle]

if {$use_gmii2rgmii_core_on_eth0 == true } {
    puts $lwipopts_fd "#define XPAR_GMII2RGMIIICON_ON_ETH1_ADDR $gmii2rgmii_core_address_on_eth0"
}

if {$use_gmii2rgmii_core_on_eth1 == true } {
    puts $lwipopts_fd "#define XPAR_GMII2RGMIIICON_ON_ETH1_ADDR $gmii2rgmii_core_address_on_eth1"
}

if {$use_1000baseX_on_88e1512 == true } {
    puts $lwipopts_fd "#define USE_1000BASEX"
}
puts $lwipopts_fd ""
```

增加这段代码的目的是为了在工程所对应的 bsp 中的 lwipopts.h 头文件里，如下所示。



是否增加

```
#define XPAR_GMII2RGMIIICON_ON_ETH0_ADDR 8 或
#define XPAR_GMII2RGMIIICON_ON_ETH1_ADDR 8 或
#define USE_1000BASEX
```

的宏定义。

由于 88E1512 芯片可以连接 RJ45 电口或者 SFP 光口，且完全自适应，当芯片工作于两种不同模式时，其通过 MDIO 接口所需访问或配置的寄存器便存在较大差异。在 lwip141_v1_73\src\contrib\ports\xilinx\netif\xemacpsif_physpeed.c 源文件中的 get_Marvell_phy_speed()

函数仅包含了当芯片工作于电口模式时的寄存器读写操作。

因此，需要添加当其工作于光口 1000BASEX 模式时的寄存器读写操作。上面提到的宏定义 #define USE_1000BASEX 就是为了用来设置 get_Marvell_phy_speed() 函数对于 88E1512 芯片的寄存器是使用电口配置还是光口配置。

添加的代码如下，增加条件编译#ifndef USE_1000BASEX，如下图所示。

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

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

#ifndef USE_1000BASEX

    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

    XEmacPs_PhphyRead(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, &control);
    control |= IEEE_ASYMMETRIC_PAUSE_MASK;
    control |= IEEE_PAUSE_MASK;
    control |= ADVERTISE_100;
    control |= ADVERTISE_10;
    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_AUTONEGO_ADVERTISE_REG, control);

#endif
```

添加其工作于 1000BASEX 模式时的寄存器读写配置代码，并设置条件编译，如下所示。

```
#else

    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 1);

    XEmacPs_PhphyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;
    control |= IEEE_STAT_AUTONEGOTIATE_RESTART;
//control &= IEEE_CTRL_ISOLATE_DISABLE;
    control &= 0xFBFF;
    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    XEmacPs_PhphyRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);
    control |= IEEE_CTRL_RESET_MASK;
    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);

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

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

    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 1);
    XEmacPs_PhphyRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);
    while ( !(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE) ) {
        sleep(1);
        XEmacPs_PhphyRead(xemacpsp, phy_addr, IEEE_STATUS_REG_OFFSET,
                           &status);
    }
    xil_printf("autonegotiation complete \r\n");
```

```

//XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 1);
XEmacPs_PhphyRead(xemacpsp, phy_addr, IEEE_PARTNER_ABILITIES_1_REG_OFFSET, &temp);
if ((temp & 0x0020) == 0x0020) {
    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    return 1000;
}
else {
    XEmacPs_PhphyWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);
    xil_printf("Link error, temp = %x\r\n", temp);
    return 0;
}

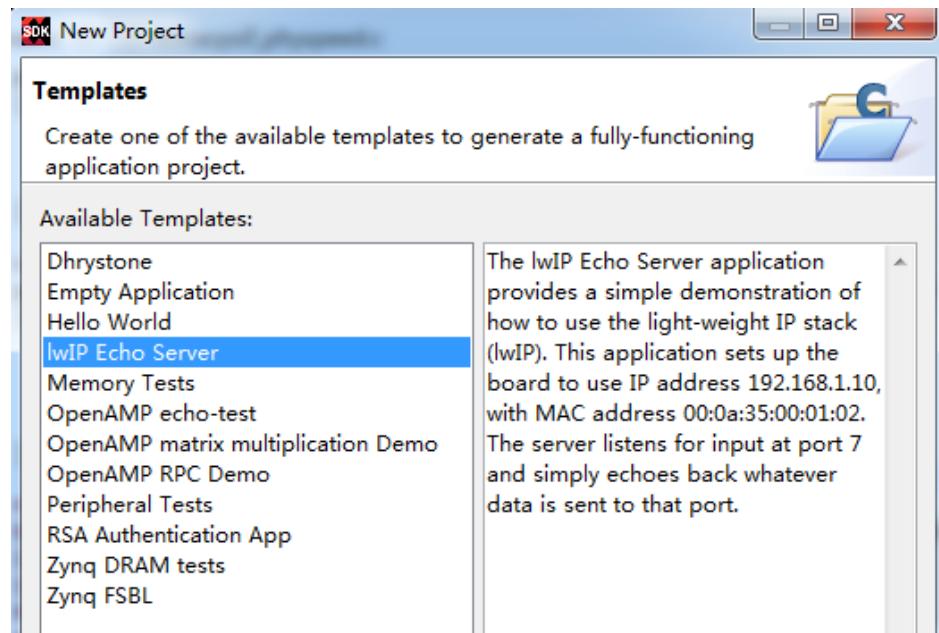
#endif

```

到此，LWIP 库的修改完成。

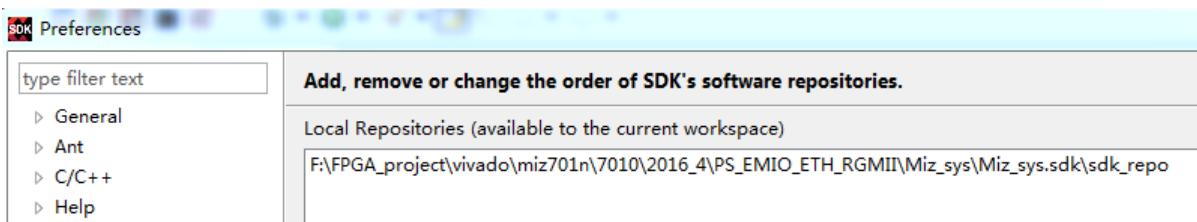
14.4.2 创建工程

本例程使用了 SDK 自带的 lwip echo server 例程来验证子卡电口和光口的功能，因此在创建工作时选择 LwIP Echo Server 模板，如下图所示。该例程基于 LWIP 库在 ARM 中建立一个 TCP Echo Server，IP 地址为 1915.168.1.10，端口号为 7。



14.4.2.1 lwip 库设置

在 SDK 软件中选择修改后的 lwip 库的路径（需根据实际情况更改此路径，若不更改将产生错误），如下图所示。



修改完成后，打开 BSP 设置，此时 lwip echo server 例程使用的是 SDK 自带的 1.7 版本 LWIP 库，为了替换成我们所修改过的 1.73 版本，首先需要取消 lwip141 1.7 的勾选，如下图所示。



然后关闭 SDK，然后重启 SDK 打开该工程目录。然后打开 BSP 设置，此时 lwip 库便变成了所修改过的 1.73 版本，如下图所示。勾选 lwip141 v1.73 版本。

Supported Libraries		
Check the box next to the libraries you want included in your Board Support Package. You can use the search bar at the top to find specific libraries.		
Name	Version	Description
libmetal	1.1	Libmetal Library
<input checked="" type="checkbox"/> lwip141	1.73	lwIP TCP/IP Stack library: lwIP v1.4.1
openamp	1.2	OpenAmp Library
xilffs	3.5	Generic Fat File System Library
xiflash	4.2	Xilinx Flash library for Intel/AMD CFI compliant ...
xilisf	5.7	Xilinx In-system and Serial Flash Library
xilmfs	2.2	Xilinx Memory File System
xilpm	2.0	Power Management API Library for ZynqMP
xilrsa	1.2	Xilinx RSA Library
xilskey	6.1	Xilinx Secure Key Library

将 use_axieth_on_zynq 和 use_emaclite_on_zynq 设为 0。如下图所示。

Configuration for library: lwip141				
Name	Value	Default	Type	Description
api_mode	RAW_API	RAW_API	enum	Mode of operation for lwIP (I
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	0	1	integer	Option if set to 1 ensures axi
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensures em

在 lwip 库增加的选项中，由于子卡通过 GMII to RGMII IP 核与 ETH1 连接，因此，将 use_gmii2rgmii_core_on_eth1 设置为 true，GMII to RGMII IP 核对应的 PHY 地址为 8，即 gmii2rgmii_core_address_on_eth1 设为 8。ETH0 不使用 EMIO，因此 use_gmii2rgmii_core_on_eth0 为 false，gmii2rgmii_core_address_on_eth0 无需进行设置。

当子卡的电口工作时，use_1000basex_on_88e1512 设置为 false；当工作于光口模式时，将其设为 true 即可。如下图所示。

emio_options			
gmii2rgmii_core_address_on_eth0	0	0	integer
gmii2rgmii_core_address_on_eth1	8	0	integer
use_1000basex_on_88e1512	false	false	boolean
use_gmii2rgmii_core_on_eth0	false	false	boolean
use_gmii2rgmii_core_on_eth1	true	false	boolean

其余选项的参数默认即可，不用修改。

14.4.2.2 example 代码修改

将 platform_config.h 的关于 EMAC 的宏定义改为 ETH1，如下图所示。

```
#define PLATFORM_EMAC_BASEADDR XPAR_XEMACPS_1_BASEADDR
```

其余部分无需修改。

14.5 程序测试

14.5.1 电口测试

14.5.1.1 lwip 库设置

电口模式下 lwip 的设置如下图所示。

emio_options			
gmii2rgmii_core_address_on_eth0	0	0	integer
gmii2rgmii_core_address_on_eth1	8	0	integer
use_1000basex_on_88e1512	false	false	boolean
use_gmii2rgmii_core_on_eth0	false	false	boolean
use_gmii2rgmii_core_on_eth1	true	false	boolean

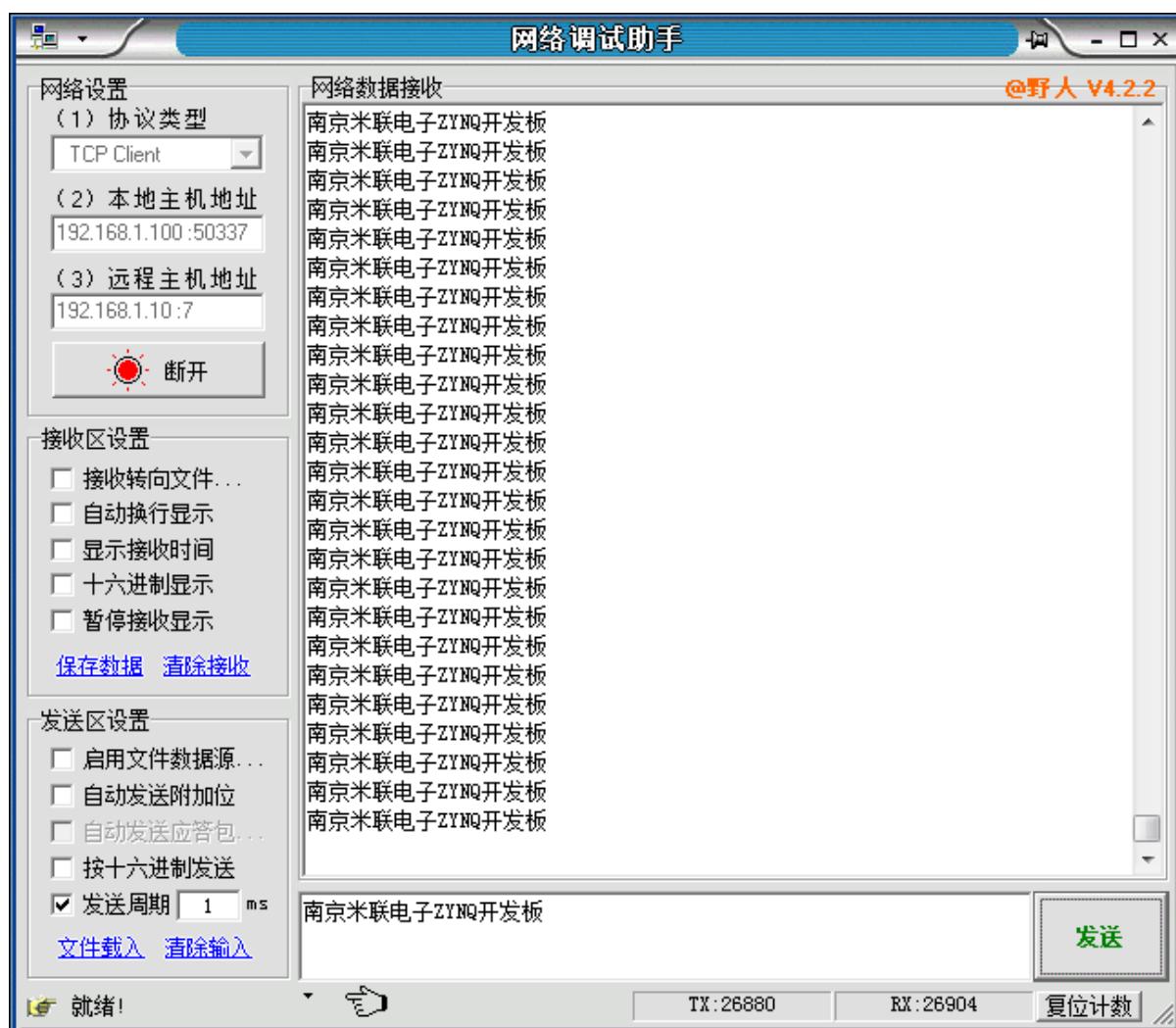
14.5.1.2 网络测试

将千兆网线插入子卡的 RJ45 座中，与电脑连接，将电脑的 ip 地址设为 1915.168.1.100，子网掩码为 255.255.255.0。然后给开发板上电，通过 SDK 将程序下载入开发板后，观察 SDK 串口打印信息，如下图示所示。表示千兆网络连接正常。

Connected to: Serial (COM12, 115200, 0, 8)

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

打开网络调试助手，以 TCP Client 模式连接开发板的 IP 地址 1915.168.1.10，端口号 7，输入文字发送，网络调试助手便可收到开发板返回相同的文字，如下图所示。



14.5.2 光口测试

14.5.15.1 lwip 库设置

光口模式下 lwip 的设置如下图所示，要将 use_1000basex_on_88e1512 设为 true。

emio_options				
gmii2rgmii_core_address_on_eth0	0	0	integer	
gmii2rgmii_core_address_on_eth1	8	0	integer	
use_1000basex_on_88e1512	true	false	boolean	
use_gmii2rgmii_core_on_eth0	false	false	boolean	
use_gmii2rgmii_core_on_eth1	true	false	boolean	

14.5.15.2 网络测试

将 SFP 电口模块插入子卡的 SFP 屏蔽笼中，将千兆网线插入 SFP 电口模块，与电脑连接，将电脑的 ip 地址设为 1915.168.1.100，子网掩码为 255.255.255.0。然后给开发板上电，通过 SDK 将程序下载入开发板后，观察 SDK 串口打印信息，如下图示所示。

```
Connected to: Serial ( COM12, 115200, 0, 8 )

-----lwIP TCP echo server -----

TCP packets sent to port 6001 will be echoed back

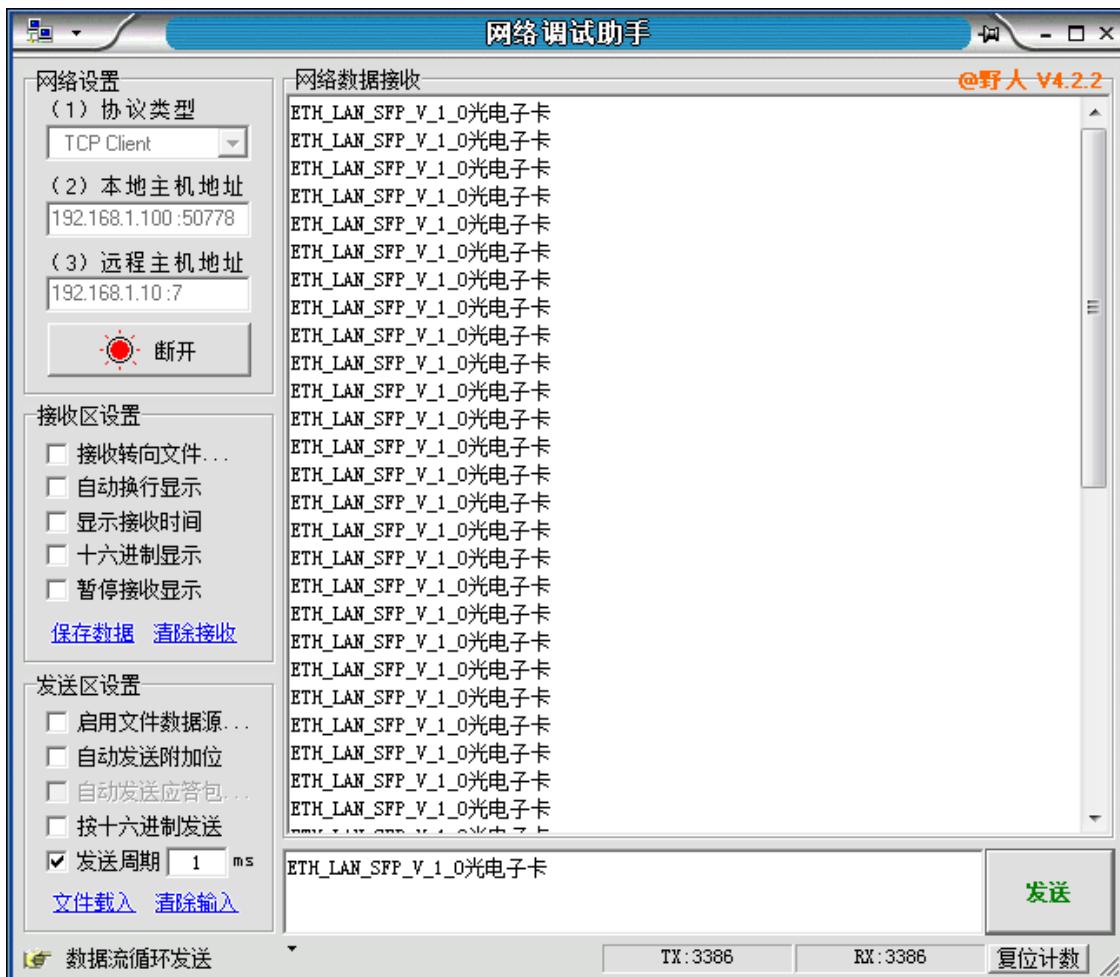
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
Board IP: 192.168.1.10

Netmask : 255.255.255.0

Gateway : 192.168.1.1

TCP echo server started @ port 7
```

同理，打开网络调试助手，以 TCP Client 模式连接开发板的 IP 地址 1915.168.1.10，端口号 7，输入文字发送，网络调试助手便可收到开发板返回相同的文字，如下图所示。



CH15 PL AXI ETH 光电网络通信

15.1 概述

关于在 ZYNQ 中进行以太网应用的开发，目前在米联的教程中已经讲解了 2 种实现方案。如下：

- 将 PS 的 ENET0/ENET1 通过 MIO 以 RGMII 接口连接外部 PHY 芯片。
- 将 PS 的 ENET0/ENET1 通过 EMIO 的方式扩展至 PL，在 PL 中再通过 RGMII 接口连接外部 PHY 芯片。

本例程将基于米联电子设计的光电双口扩展子卡 FEP_ETH_SFP_CARD 介绍第 3 种应用方案。本例程完全脱离 PS 内部的以太网外设资源，在 PL 中分别通过 AXI 1G/2.5G Ethernet Subsystem 和 AXI Direct Memory Access 这两个 IP 核来实现 PS 中以太网外设 GEMAC 和 DMA 的功能，PS 通过 AXI 总线控制这两个 IP 核便可由子卡实现 LWIP 网络通信。

本例程基于 Vivado 2016.4 版本开发，参考 xapp1082 和 fpgadeveloper 工程师的应用工程：

<https://github.com/fpgadeveloper/ethernet-fmc-axi-eth>。

15.2 基本原理

本例程在 PL 中搭建了 1 个 AXI 1G/2.5G Ethernet Subsystem 以及 1 个 AXI Direct Memory Access IP 核。这两个 IP 核均通过 AXI 总线经 S_AXI_HP0 口与 PS 连接，PS 通过 AXI 总线对其进行配置和控制。其中，AXI 1G/2.5G Ethernet Subsystem IP 核通过 RGMII 接口与外部子卡中的 88E1512 芯片连接。在 PS 端通过 SDK 自带的 lwip echo server 例程，通过子卡分别以 RJ45 电口和 SFP 电口两种方式与 PC 机实现 TCP 网络通信。

15.2.1 88E1512

子卡所使用的 88E1512 芯片支持电口和光口，当其上电复位后，自动进入 RGMII to Copper 以及 RGMII to 1000BASEX 自适应状态，这是由其如下图寄存器值决定。当需要使用电口时，将网线与子卡的 RJ45 连接，当需要使用光口时，在子卡的 SFP 屏蔽笼中插入光模块即可。

Fiber/Copper Auto-Selection

The device has a patented feature to automatically detect and switch between fiber and copper cable connections. The auto-selection operates in one of three modes: Copper /1000BASE-X, Copper/100BASE-FX, and Copper/SGMII Media Interface.

Register 20_18.2:0 and register 20_18.6 select the Fiber/Copper auto media modes of operation. See [Table 33](#) for details.

Table 33: Fiber/Copper Auto-media Modes of Operation

Reg 20_18.2:0	Reg 20_18.6	Auto-media Modes of Operation
110	X	Copper/SGMII media
111	0	Copper/1000BASE-X
011	1	Copper/100BASE-FX

The device monitors the signals of the S_INP/N and the MDIP/N[3:0] lines. If a fiber optic cable is plugged in, the device will adjust itself to be in fiber mode. If an RJ-45 cable is plugged in, the device will adjust itself to be in copper mode. If both cables are connected then the first media to establish link, or the preferred media will be enabled. The media which is not enabled will be automatically turned off to save power. If the link on the first media is lost, then the inactive media will be powered up, and both media will once again start searching for link.

Table 220: General Control Register 1
Page 18, Register 20

Bits	Field	Mode	HW Rst	SW Rst	Description
15	Reset	R/W, SC	0x0	SC	Mode Software Reset. Affects page 6 and 18 Writing a 1 to this bit causes the main PHY state machines to be reset. When the reset operation is done, this bit is cleared to 0 automatically. The reset occurs immediately. 1 = PHY reset 0 = Normal operation
14:13	Reserved	R/W	0x0	Retain	Set to 0s.
12:10	Reserved	R/W	0x0	Retain	Reserved for future use.
9:7	Reserved	R/W	0x4	Retain	Set to 100
6	Auto-Media Detect (AMD) 100BASE-FX/ 1000BASE-X	R/W	0x0	Retain	This bit selects the fiber auto-media modes as follows: 1 = mode 011 is AMD between copper and 100BASE-FX 0 = mode 111 is AMD between copper and 1000BASE-X
5:4	Auto Media Detect Preferred Media	R/W	0x0	Retain	00 = Link on first media 01 = Copper Preferred 10 = Fiber Preferred 11 = Reserved
3	Reserved	R/W	0x0	Update	Set to 0
2:0	MODE[2:0]	R/W	See Descr.	Update	Changes to this bit are disruptive to the normal operation; therefore, any changes to these registers must be followed by a software reset to take effect. 1512 device comes up in MODE[2:0] =0x7 on hardware reset. 000 = RGMII (System mode) to Copper 001 = SGMII (System mode) to Copper 010 = RGMII (System mode) to 1000BASE-X 011 = RGMII (System mode) to 100BASE-FX 100 = RGMII (System mode) to SGMII (Media mode) 101 = Reserved 110 = RGMII (System mode) to Auto Media Detect Copper/SGMII (Media mode) 111 = RGMII (System mode) to Auto Media Detect Copper/1000BASE-X/100BASE-FX (see 20_18.6)

15.2.2 88E1512 RGMII 接口时序

88E1512 芯片 RGMII 接口的时序存在延迟和非延迟 2 种模式，由其内部的一个寄存器值来决定，如下图所示。

Table 121: MAC Specific Control Register 2
Page 2, Register 21

Bits	Field	Mode	HW Rst	SW Rst	Description
12:7	Reserved		0x20	0x20	Reserved.
6	Default MAC interface speed (MSB)	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. Also, used for setting speed of MAC interface during MAC side loopback. Requires that customer set both these bits and force speed using register 0 to the same speed. MAC Interface Speed during Link down. Bits 6, 13 00 = 10 Mbps 01 = 100 Mbps 10 = 1000 Mbps
5	RGMII Receive Timing Control	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. 1 = Receive clock transition when data stable 0 = Receive clock transition when data transitions
4	RGMII Transmit Timing Control	R/W	0x1	Update	Changes to these bits are disruptive to the normal operation; therefore, any changes to these registers must be followed by software reset to take effect. 1 = Transmit clock internally delayed 0 = Transmit clock not internally delayed

芯片上电通过RESET引脚复位后，寄存器中决定RGMII输入输出时序关系的bit位的值都是1。也就是说，上电复位后芯片的RGMII接口均为延迟时序模式。延迟模式是指88E1512芯片在其内部给输出的接收时钟信号RX_CLK和输入的发送的时钟信号TX_CLK增加了2ns左右的延时。这是为了避免通过PCB绕线的方式增加时钟信号的延时，使RX_CLK和TX_CLK都能与相应RXD和TXD数据窗的中心对齐，从而使得RGMII接口数据的建立和保持时间达到余量最大的状态。

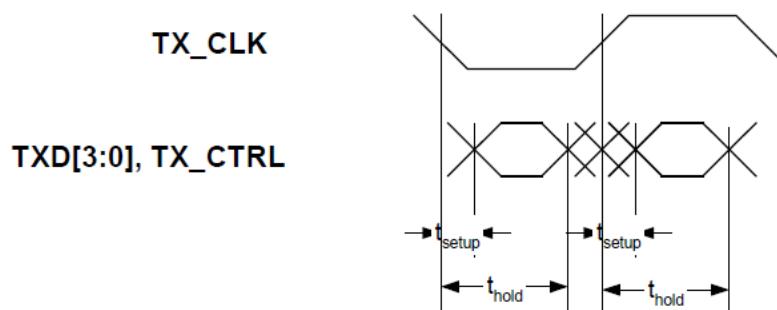
因此，此时RGMII接口发送部分信号的时序关系如下图所示。

4.10.2.2 PHY Input - TX_CLK Delay when Register 21_2.4 = 1

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.4 = 1 (add delay)	-0.9			ns
t_{hold}		2.7			ns

Figure 37: TX_CLK Delay Timing - Register 21_2.4 = 1 (add delay)



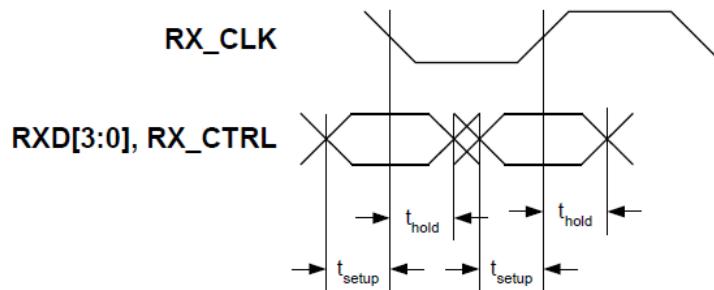
RGMII接口接收部分信号的时序关系如下图所示。

4.10.2.4 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.5 = 1 (add delay)	1.2			ns
t_{hold}		1.2			ns

Figure 39: RGMII RX_CLK Delay Timing - Register 21_2.5 = 1 (add delay)



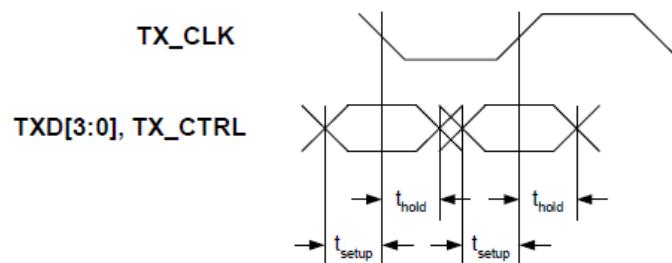
另外，非延迟模式下 RGMII 接口发送部分信号的时序关系如下图所示。

4.10.2.1 PHY Input - TX_CLK Delay when Register 21_2.4 = 0

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.4 = 0	1.0			ns
t_{hold}		0.8			ns

Figure 36: TX_CLK Delay Timing - Register 21_2.4 = 0



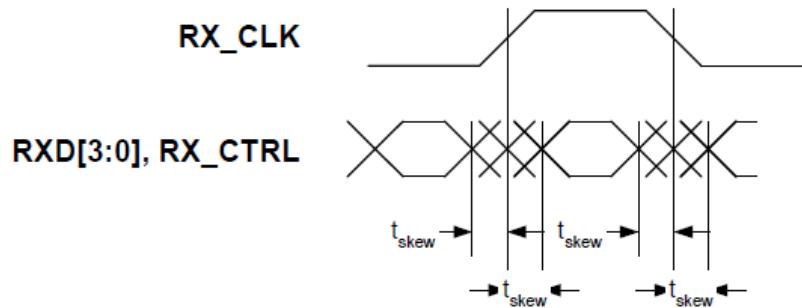
非延迟模式下 RGMII 接口接收部分信号的时序关系如下图所示。

4.10.2.3 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

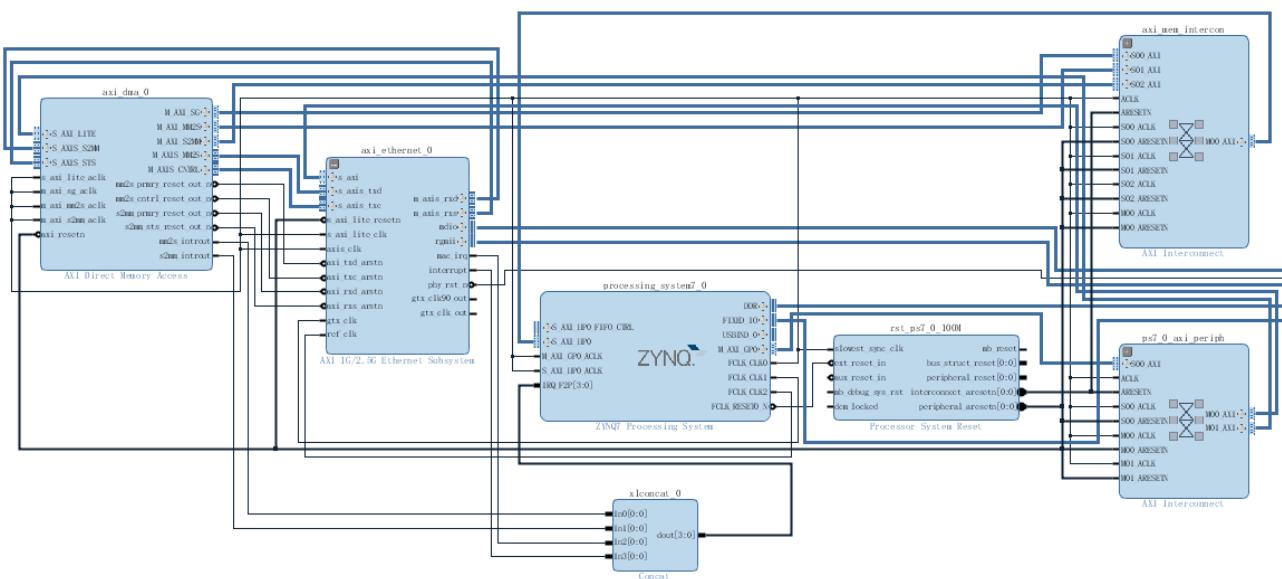
Symbol	Parameter	Min	Typ	Max	Units
t_skew	Register 21_2.5 = 0	- 0.5		0.5	ns

Figure 38: RGMII RX_CLK Delay Timing - Register 21_2.5 = 0



15.3 PL 部分设计

15.3.1 IP 连线图



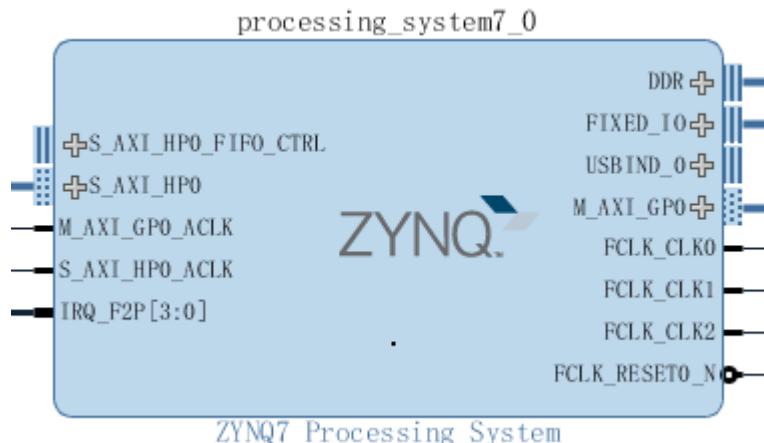
15.3.2 ZYNQ PS 设置

ZYNO PS 需要作如下配置:

- 使能 M_AXI_GP0 接口
 - 使能 S_AXI_HP0 接口
 - 使能 PL 至 PS 的中断接口 IRQ_F2P
 - 使能 FCLK_CLK0，设为 100M，作为 PL 部分所有 AXI 接口的时钟

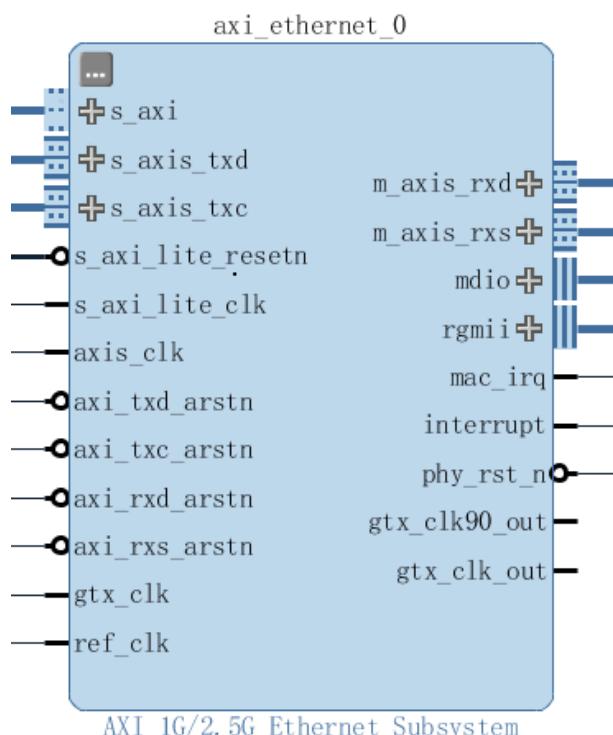
- 使能 FCLK_CLK1，设为 125M，作为 AXI 1G/2.5G Ethernet Subsystem IP 核 gtx_clk 的输入时钟。
- 使能 FCLK_CLK2，设为 200M，作为 AXI 1G/2.5G Ethernet Subsystem IP 核 ref_clk 的输入时钟，作为其内部 IDELAYCTRL 的参考时钟。

如下图所示。

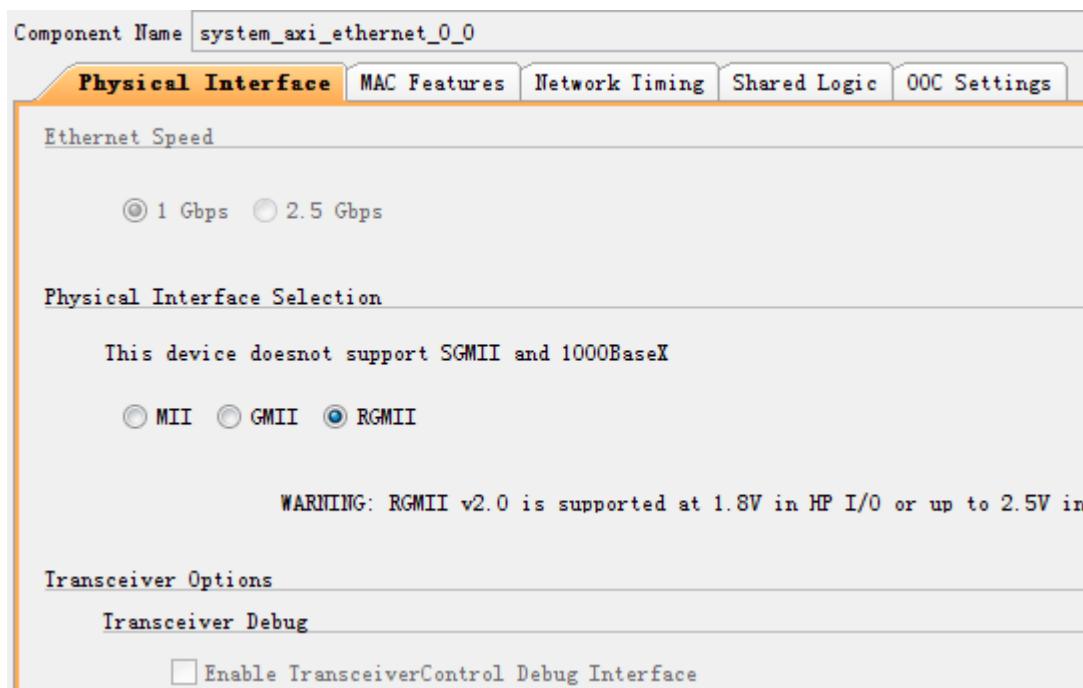


15.3.3 AXI 1G/2.5G Ethernet Subsystem

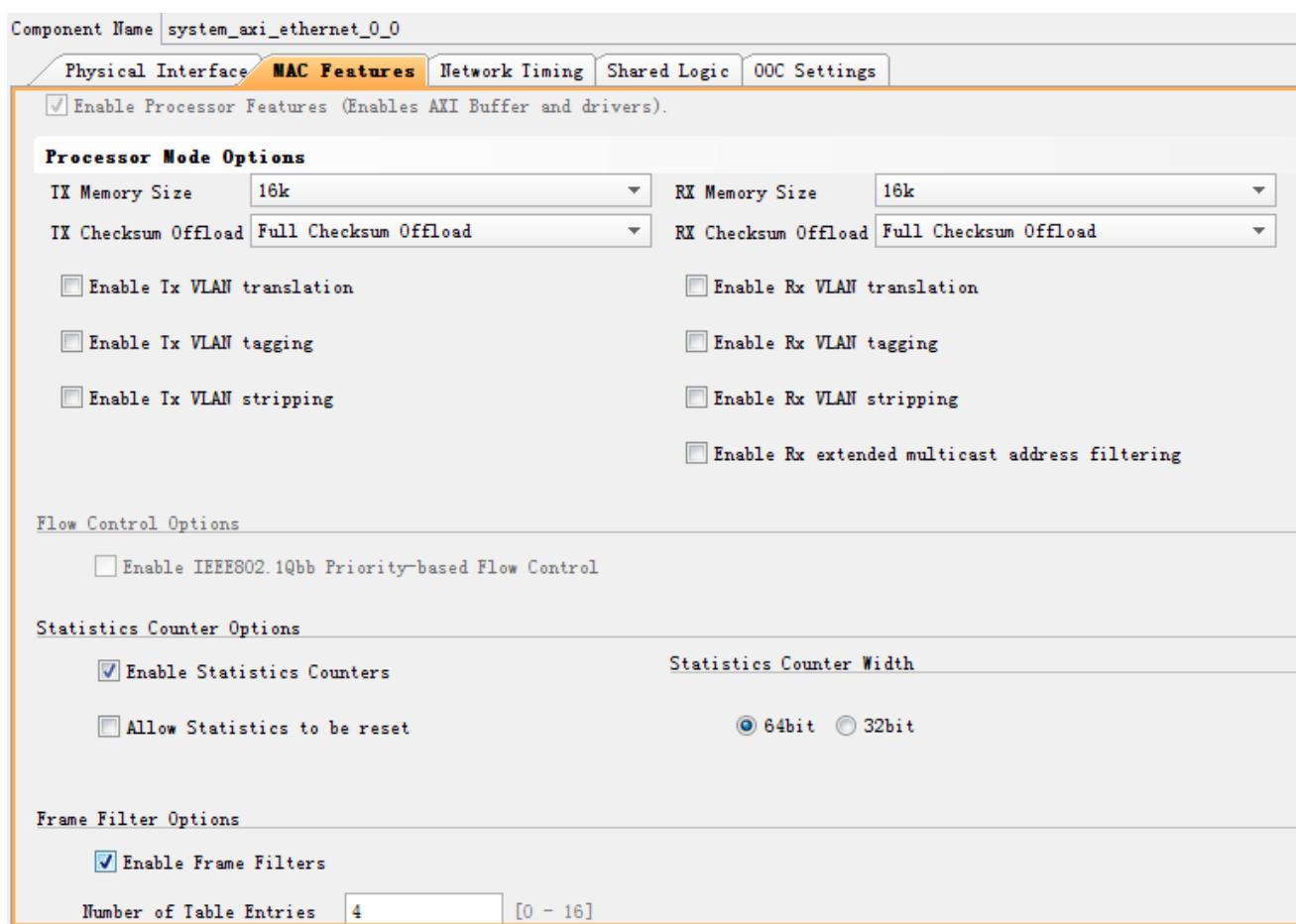
AXI 1G/2.5G Ethernet Subsystem IP 核的设置如下图所示。



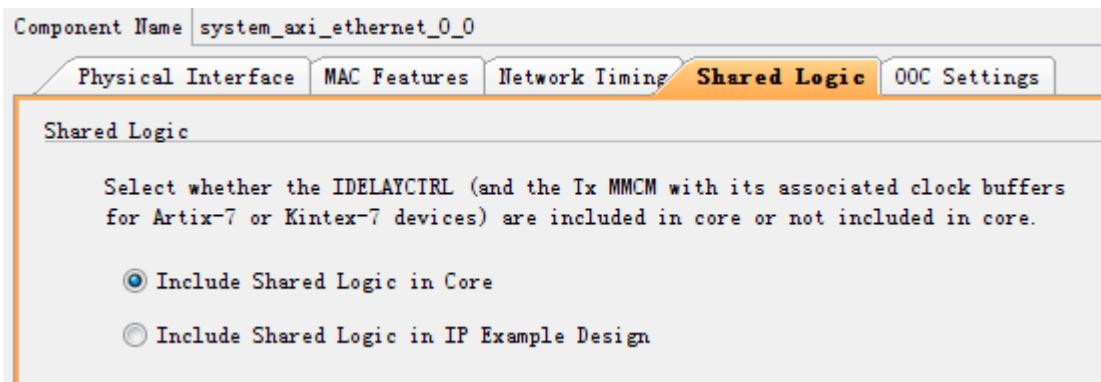
由于子卡中的 88E1512 芯片为 RGMII 接口，因此将 IP 设为 RGMII 接口，如下图所示。



TX Memory 和 RX Memory 对应 IP 核内部的接收和发送缓存，缓存越大将有利于提高接收和发送的速率，不妨将 TX Memory 和 RX Memory 都设为 16k。另外，将 TX 和 RX Checksum Offload 全部设置为 Full Checksum Offload，这将使 IP 核完成所有收发数据包中 TCP UDP IP 协议首部校验和的计算功能，使 PS 无需再进行这些校验和的计算，从而提高 PS 中网络协议栈的运行效率。（PS 内部的 GEMAC 也具有该功能）如下图所示。



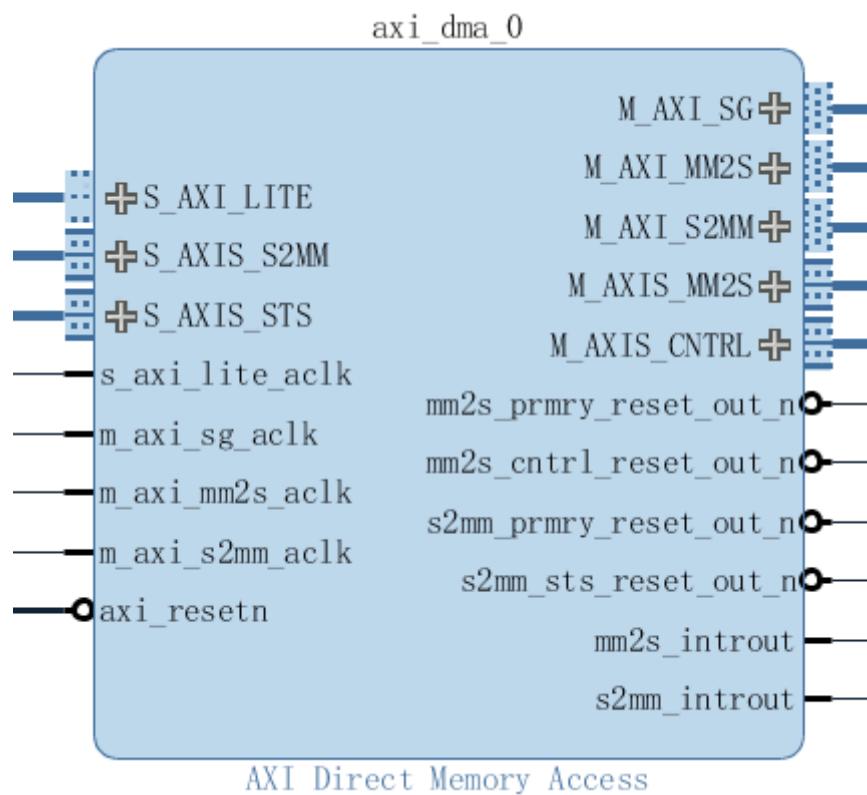
在独立使用单个 IP 核时，应选择将 shared logic 包含在 IP 核内部，如下图所示。



其余选项保持默认即可，无需更改。

15.3.4 AXI Direct Memory Access

AXI Direct Memory Access IP 核的设置如下图所示。



需要说明的是：

- 将 AXI DMA 配置为链式 DMA 工作模式
- 使能 TX Control 和 RX Status 接口，这两个接口是 AXI DMA 与 AXI 1G/2.5G Ethernet Subsystem 专用的连接接口。分别用于发送控制，和接收状态传递。
- 将 Length Register 的位宽设为 16，对应 1 次链式 DMA 传输的最大总数据量为 64KB。若想进一步增大数据传输效率，可将位宽设大，但不应该小于 AXI 1G/2.5G Ethernet Subsystem 中所设置的 TX 和 RX Memory 的值，本例程中为 16KB，对应的最小位宽应为 14。

所有的参数设置如下图所示。

Component Name `system_axi_dma_0_0`

Enable Asynchronous Clocks (Auto)

Enable Scatter Gather Engine

Enable Micro DMA

Enable Multi Channel Support

Enable Control / Status Stream

Width of Buffer Length Register (8-23) `16` bits

Address Width (32-64) `32` bits

Enable Read Channel Enable Write Channel

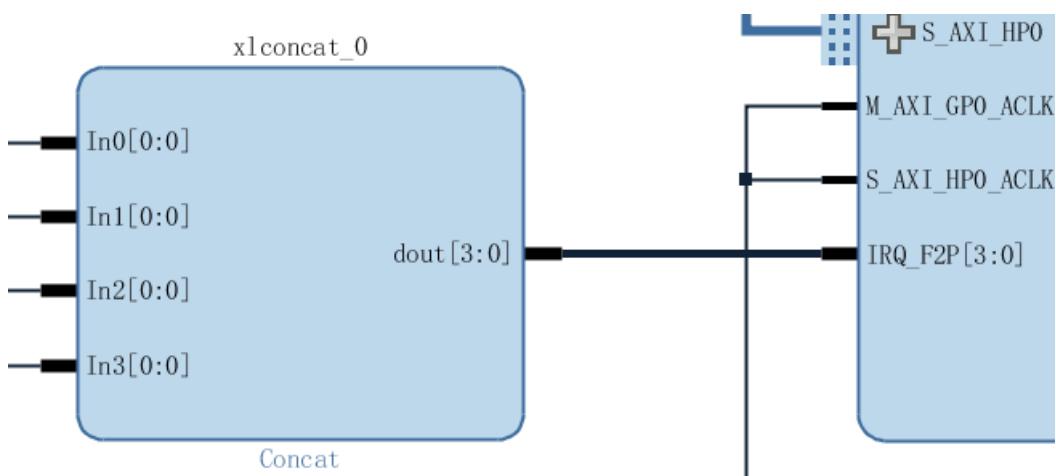
Number of Channels	<code>1</code>	Number of Channels	<code>1</code>
Memory Map Data Width	<code>32</code>	Memory Map Data Width	<code>32</code>
Stream Data Width	<code>32</code>	Stream Data Width (Auto)	<code>32</code>
Max Burst Size	<code>16</code>	Max Burst Size	<code>16</code>

Allow Unaligned Transfers Allow Unaligned Transfers

Use Rxlength In Status Stream

15.3.5 PL 至 PS 的中断

AXI 1G/2.5G Ethernet Subsystem 和 AXI DMA 分别有 2 个输出中断信号，通过 Concat 将这 4 个中断信号与 PS 的 IRQ_F2P 接口连接，如下图所示。



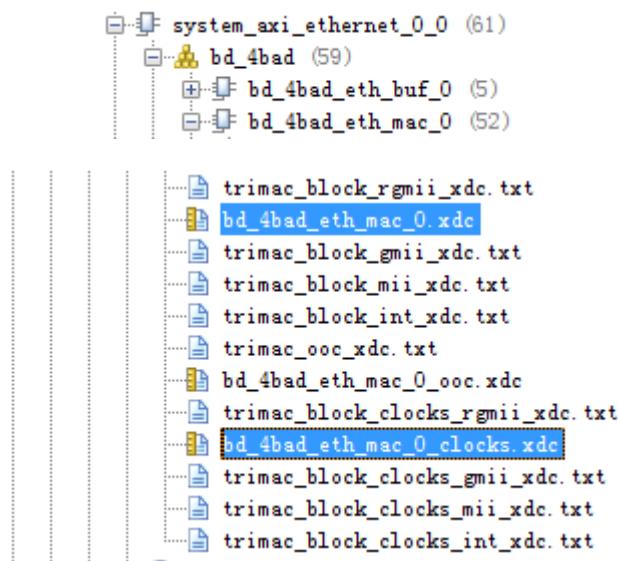
其中，AXI DMA 的 mm2s_intout、s2mm_intout 两个中断信号分别对应 TX 和 RX 两个方向的数据搬移完成、传输错误等中断。AXI 1G/2.5G Ethernet Subsystem 的 interrupt 为其内部各种状态的中断信号，例如自协商完成，接收错误，发送完成等；mac_irq 为 mdio 接口中断信号。实际在 PS

端的 LWIP 库中的驱动程序里并不使用 AXI 1G/2.5G Ethernet Subsystem 的这 2 个中断。用户可根据需求自行设计。

15.3.6 时序约束

本例程中，时序约束主要是针对 RGMII 接口进行 input delay 和 output delay 的约束，以及 IDELAYE2 延时 tap 数的设置，使 RGMII 接口的满足建立和保持时间的要求，从而达到时序收敛。

对于 input delay 和 output delay 的约束，AXI 1G/2.5G Ethernet Subsystem IP 核所自带的 bd_****_eth_mac_0_clock.xdc 文件中已经包含了默认设置。对于 IDELAYE2 延时 tap 数的设置，在 bd_****_eth_mac_0.xdc 中包含了默认模板。这两个 xdc 文件位置如下图所示。



15.3.6.1 input delay 约束

bd_****_eth_mac_0_clock.xdc 文件中，关于 input delay 的约束如下所示。约束范围为：-1.5~ -2.8ns。

```
# define a virtual clock to simplify the timing constraints
create_clock -name [current_instance .]_rgmii_rx_clk -period 8
set rgmii_rx_clk [current_instance .]_rgmii_rx_clk

# Identify RGMII Rx Pads only.
# This prevents setup/hold analysis being performed on false inputs,
# eg, the configuration_vector inputs.

set_input_delay -clock [get_clocks $rgmii_rx_clk] -max -1.5 [get_ports {rgmii_rx[*] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks $rgmii_rx_clk] -min -2.8 [get_ports {rgmii_rx[*] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks $rgmii_rx_clk] -clock_fall -max -1.5 -add_delay [get_ports {rgmii_rx[*] rgmii_rx_ctl}]
set_input_delay -clock [get_clocks $rgmii_rx_clk] -clock_fall -min -2.8 -add_delay [get_ports {rgmii_rx[*] rgmii_rx_ctl}]
```

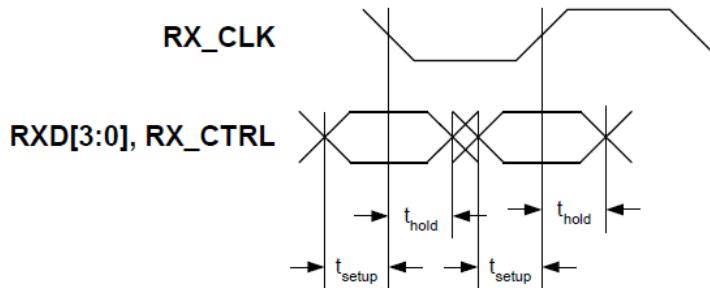
在 2.2 节中已经对 88e1512 芯片的 RGMII 接口时序进行了介绍，对于 RX 接口，如下图。

4.10.2.4 PHY Output - RX_CLK Delay

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t_{setup}	Register 21_2.5 = 1 (add delay)	1.2			ns
t_{hold}		1.2			ns

Figure 39: RGMII RX_CLK Delay Timing - Register 21_2.5 = 1 (add delay)



按照上图中的 setup time 和 hold time, input delay 的-min 应该为 $-(4-1.2) = -2.8\text{ns}$, -max 应该为 -1.2ns 。显然, IP 核自带 input delay 的约束范围为 $-1.5\text{ns} \sim -2.8\text{ns}$, 比我们计算的结果 $-1.2\text{ns} \sim -2.8\text{ns}$ 略为宽松, 用于 setup time 计算所需的 -max 时间小了 0.3ns , 为了保证时序严格收敛, 需要将 bd_****_eth_mac_0_clock.xdc 文件中 set_input_delay -max 的 -1.5 全部改为 -1.2 。需要注意的是, 这些 xdc 文件由 IP 核自动生成, 每次重新配置 IP 后, vivado 都会重新生成 IP 的 output product, 因此会将被修改的 xdc 文件覆盖还原, 需要手动重新再次修改 xdc 文件才行。

15.3.6.2 output delay 约束

bd_****_eth_mac_0_clock.xdc 文件中, 关于 output delay 的约束如下所示。约束范围为: $-0.75 \sim 0.75\text{ns}$ 。

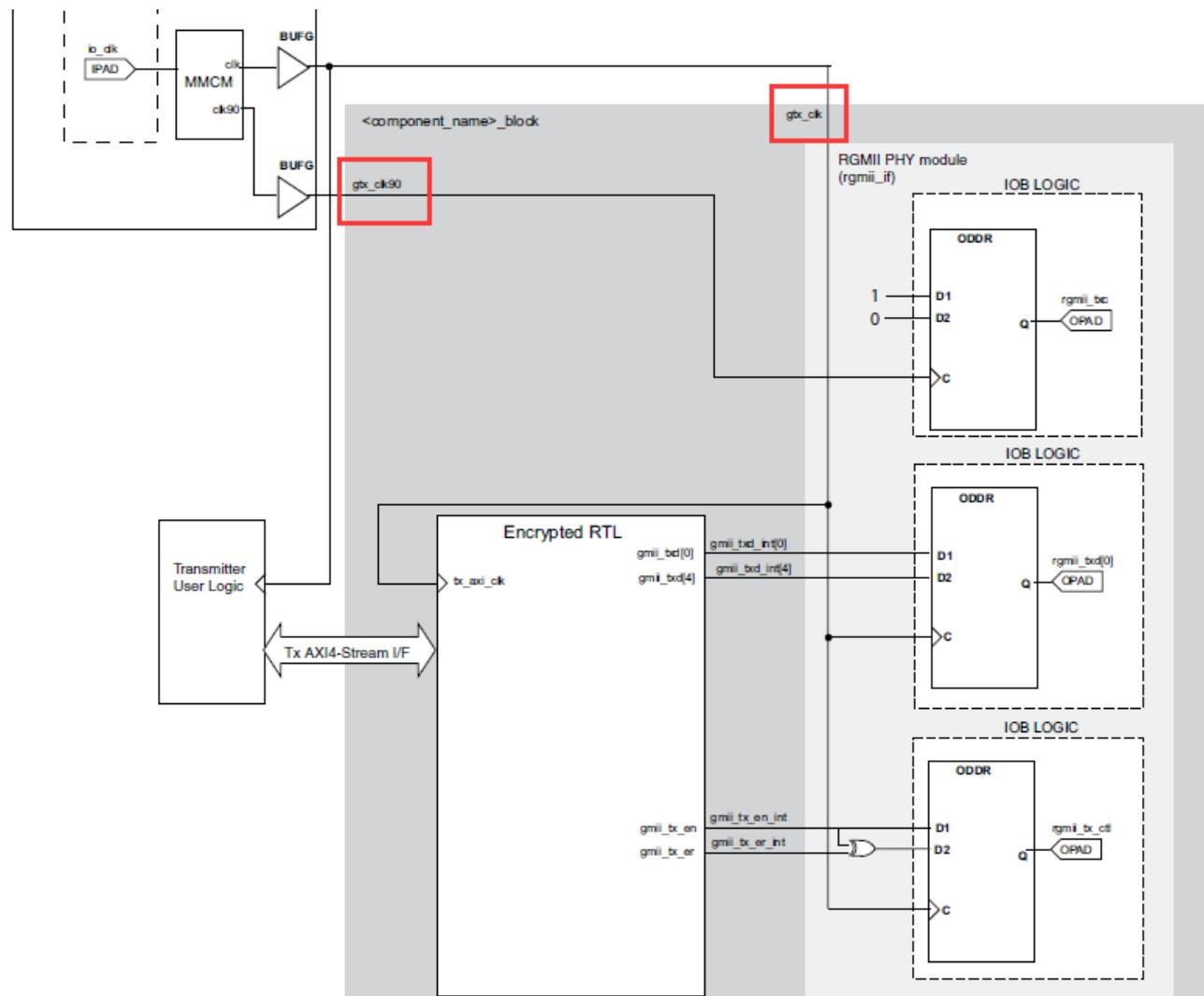
```

create_generated_clock -name [current_instance .]_rgmii_tx_clk -divide_by 1 -source [get_pins {tri_mode_ethernet_mac_i/rgmii_interface/rgmii_txc_ddr/C}]
set rgmii_tx_clk [current_instance .]_rgmii_tx_clk

set_output_delay 0.75 -max -clock [get_clocks $rgmii_tx_clk] [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
set_output_delay -0.75 -min -clock [get_clocks $rgmii_tx_clk] [get_ports {rgmii_txd[*] rgmii_tx_ctl}]
set_output_delay 0.75 -max -clock [get_clocks $rgmii_tx_clk] [get_ports {rgmii_txd[*] rgmii_tx_ctl}] -clock_fall -add_delay
set_output_delay -0.75 -min -clock [get_clocks $rgmii_tx_clk] [get_ports {rgmii_txd[*] rgmii_tx_ctl}] -clock_fall -add_delay

```

在 AXI 1G/2.5G Ethernet Subsystem 中, 对于 HR BANK 所采用的 RGMII 发送接口方案如下图所示。



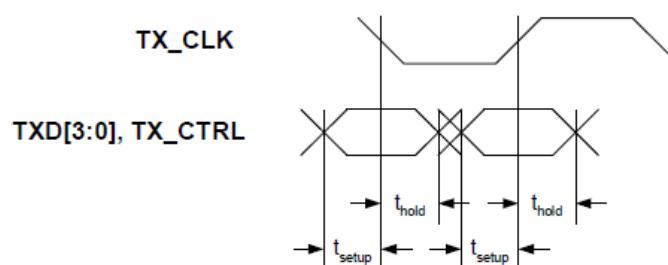
rgmii_txc 的发送时钟 gtx_clk90，与 rgmii_td、rgmii_tx_ctl 的发送时钟 gtx_clk 存在 90° 即 2ns 的相位差。因此，此时 88E1512 芯片 RGMII 的 TX 接口时序不能再使用默认的内部延迟模式，而需要使用外部延迟模式，其时序关系如下图。

4.10.2.1 PHY Input - TX_CLK Delay when Register 21_2.4 = 0

(Over full range of values listed in the Recommended Operating Conditions unless otherwise specified)

Symbol	Parameter	Min	Typ	Max	Units
t _{setup}	Register 21_2.4 = 0	1.0			ns
t _{hold}		0.8			ns

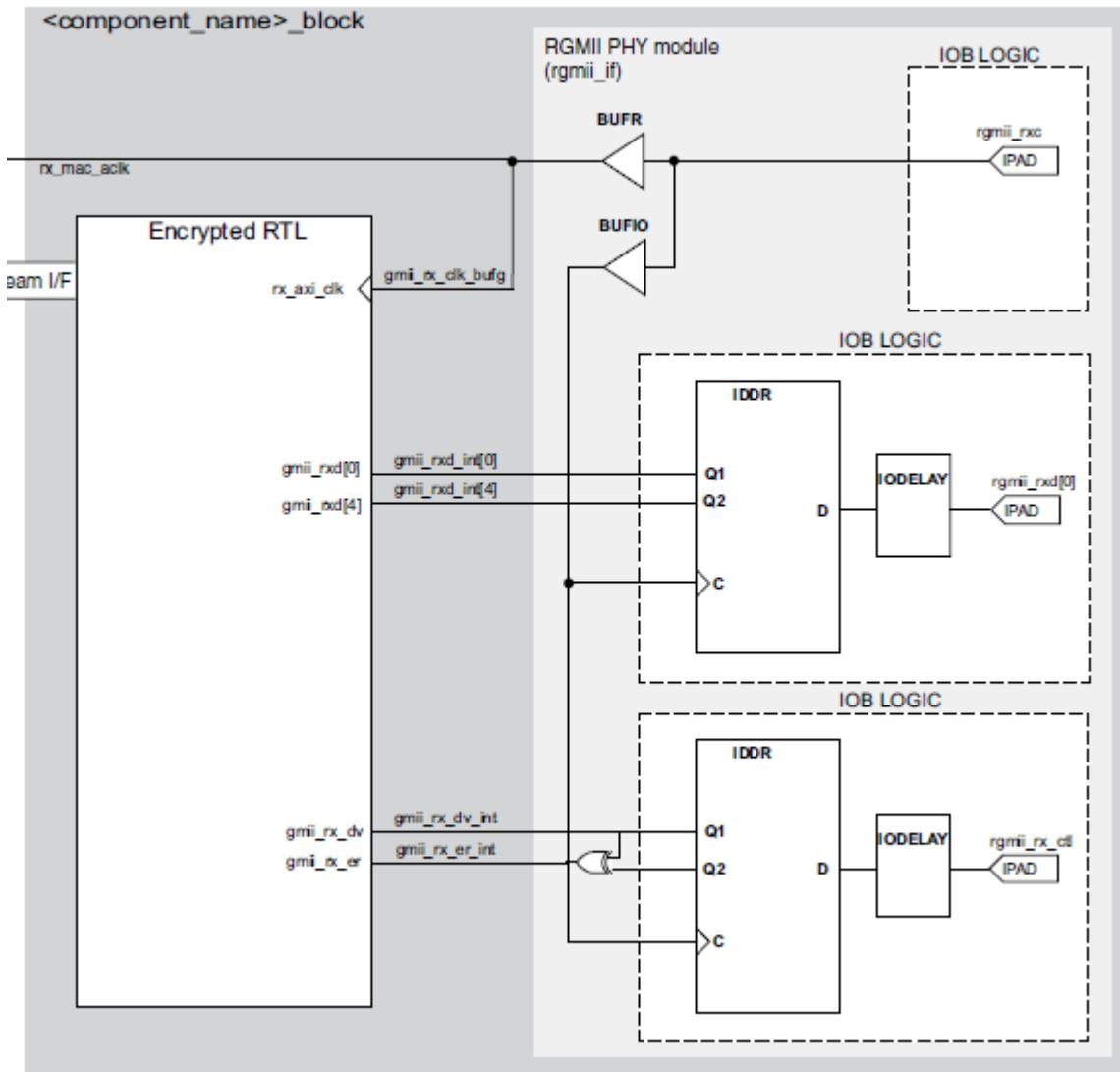
Figure 36: TX_CLK Delay Timing - Register 21_2.4 = 0



照上图中的 setup time 和 hold time, output delay 的 -min 应该为 -0.8ns, -max 应该为 1.0ns。显然, IP 核自带 output delay 的约束范围为 -0.75ns~0.75ns, 比我们计算的结果 -0.8ns~1.0ns 略为宽松, 用于 setup time 计算所需的 -max 时间小了 0.25ns, 用于 hold time 计算所需的 -min 时间大了 0.05ns。为了保证时序严格收敛, 需要将 bd_****_eth_mac_0_clock.xdc 文件中 set_output_delay -max 的 0.75 改为 1.0, -min 的 -0.75 全部改为 -0.8。

15.3.6.3 IDELAYE2 延时设置

在 AXI 1G/2.5G Ethernet Subsystem IP 核内部为 rgmii_rx_ctl 和 rgmii_rxd[3:0] 端口都连接了 IDELAYE2 模块, 如下图所示。



IDELAYE2 用来为这些信号进入 PL 内部前引入额外的延时。IDELAYE2 延时值的大小与 xdc 中的 input delay 约束是否能收敛密切相关。一般情况下, 当 input delay 约束的 setup time 出现违例, 应该将 IDELAYE2 的 tap 数减小, 降低延时值; 当 input delay 约束的 hold time 出现违例, 应该将 IDELAYE2 的 tap 数增大, 增加延时值。

bd_****_eth_mac_0.xdc 中包含了 IDELAYE2 的 tap 数的设置, 如下图所示。

```
# Apply the same IDELAY_VALUE to all RGMII RX inputs.
# This is to provide a similar Clock Path and Data Path delay.
set_property IDELAY_VALUE 15 [get_cells {tri_mode_ethernet_mac_i/rgmii_interface/delay_rgmii_rx* tri_mode_ethernet_mac_i}

# Group IODELAY components
set_property IODELAY_GROUP tri_mode_ethernet_mac_iodelay_grp [get_cells {tri_mode_ethernet_mac_i/rgmii_interface/delay_r
set_property IODELAY_GROUP tri_mode_ethernet_mac_iodelay_grp [get_cells {tri_mode_ethernet_mac_i/delayctrl_common_i}]}
```

该段约束默认是被注释的，默认的延时 tap 数为 15，编译工程后根据时序报告查看 input delay 是否时序收敛，若存在时序违例，则需要根据实际情况调整 tap 的值。经过尝试后，tap 数为 15 时可以满足时序收敛的要求，因此无需修改。

15.3.7 IO 口

AXI 1G/2.5G Ethernet Subsystem IP 核的 rgmii、mdio、phy_reset_n 接口需要作为 IO 口引出与外部 PHY 芯片连接。其中，PHY 芯片的复位信号 phy_reset_n 的低电平有效。

在 block design 自动生成的 system_wrapper.v 文件中，手动添加 2 个输出端口，如下所示。

```
output sfp_tx_disable;
output sfp_rate_sel;
```

这两个端口连接到 SFP，将 sfp_tx_disable 置 0，sfp_rate_sel 置 1。如下所示。

```
assign sfp_tx_disable = 1'b0;
assign sfp_rate_sel = 1'b1;
```

15.4 PS 程序设计

所有的驱动程序文件均包含在 c_driver 文件夹中。

15.4.1 LWIP 库修改

在 SDK 2016.4 中所使用的 LWIP 1.4.1 库的版本为 1.7。其中缺少当 PS 连接 AXI 1G/2.5G Ethernet Subsystem IP 核时对于 88E1512 芯片的配置驱动程序，若直接使用原版 LWIP 库将使 88E1512 芯片无法正常工作，从而无法进行数据传输。因此，需要手动修改库文件添加驱动程序。

首先，找到 SDK 安装目录下的 LWIP 库的路径，例如：

C:\Xilinx\SDK\2016.4\data\embeddedsw\ThirdParty\sw_service

将 lwip141_v1_7 文件夹复制一份到工程目录的 sdk_repo\bsp 文件夹下，将其重新命名为 lwip141_v1_74。第一步，修改 lwip141_v1_74\data\lwip141.mld 文件（可用 Notepad++ 等编辑器打开），将其中的版本编号

OPTION VERSION = 1.7;

修改为

OPTION VERSION = 1.74;

然后，在其中增加如下字段：

```
PARAM name = use_1000basex_on_88e1512, desc = "Settings for operation mode of 88e1512, 1000/100/10 baset
or 1000basex", type = bool, default = false;
```

增加这段代码的目的是为了在 SDK 中 BSP 设置里，lwip 参数设置对话框里增加一个选项 use_1000basex_on_88e1512，如下图所示。

Configuration for library: lwip141

Name	Value	Default	Type	Description
api_mode	RAW API (RAW_API)	RAW_API	enum	Mode of operation for
socket mode thread prio	2	2	integer	Priority of threads in so
use_1000basex_on_88e1512	true	false	boolean	Settings for operation r
use_axieth_on_zynq	1	1	integer	Option if set to 1 ensur
use_emaclite_on_zynq	0	1	integer	Option if set to 1 ensur
▷ arp_options	true	true	boolean	ARP Options
▷ debug_options	true	true	boolean	Turn on lwIP Debug?
▷ dhcp_options	true	true	boolean	Is DHCP required?
▷ icmp_options	true	true	boolean	ICMP Options
igmp_options	false	false	boolean	IGMP Options
▷ lwip_ip_options	true	true	boolean	IP Options
▷ lwip_memory_options				Options controlling lwIP
▷ pbuf_options	true	true	boolean	Pbuf Options
▷ stats_options	true	true	boolean	Turn on lwIP statistics?
▷ tcp_options	true	true	boolean	Is TCP required ?
▷ temac_adapter_options	true	true	boolean	Settings for xps-ll-tema
▷ udp_options	true	true	boolean	Is UDP required ?

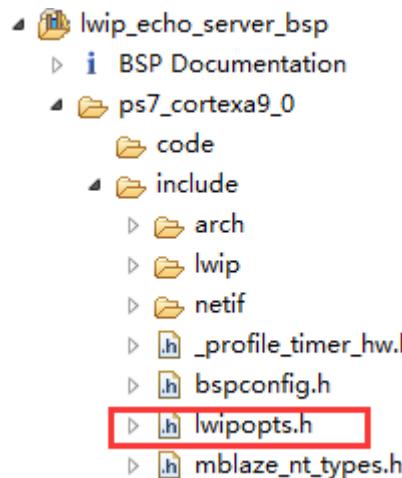
use_1000basex_on_88e1512 表示子卡中的 88E1512 芯片是否处于 1000BASEX 工作模式，即是否使用使用子卡中的 SFP 接口。

接着，打开 lwip141_v1_74\data\lwip141.tcl 文件，在 proc generate_lwip_opts {libhandle} 所在的大括号内其中增加如下字段：

```
set use_1000basex_on_88e1512 [common::get_property CONFIG.use_1000basex_on_88e1512 $libhandle]

if { $use_1000basex_on_88e1512 == true }{
    puts $lwipopts_fd "#define USE_1000BASEX"
}
puts $lwipopts_fd ""
```

增加这段代码的目的是为了在工程所对应的 bsp 中的 lwipopts.h 头文件里，如下所示。



是否增加

```
#define USE_1000BASEX
```

的宏定义。

打开 lwip141_v1_74\src\contrib\ports\xilinx\netif\xaxiemacif_physpeed.c 源文件，在其中添加函数 get_phy_speed_88E1512 ()函数，对 88E1512 芯片进行配置，函数源代码不在此进行列举，详细参考工程目录文件。

需要说明的是，其中对于 88e1512 芯片 RGMII 接口时序的设置部分代码如下：

```
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
control &= ~IEEE_RGMII_TX_CLOCK_DELAYED_MASK;
control |= IEEE_RGMII_RX_CLOCK_DELAYED_MASK;
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_MAC, control);
```

对于 TX 接口不使用内部延迟模式，对于 RX 接口使用延迟模式。

上面提到的宏定义 #define USE_1000BASEX 就是作为条件编译选项，用来设置 get_phy_speed_88E1512 ()函数对于 88E1512 芯片的寄存器是使用电口配置还是光口配置。

另外，添加宏定义

```
#define MARVEL_PHY_88E1512_MODEL 0x01D0
```

该宏定义为 88E1512 芯片的 ID。

然后，找到如下代码：

```
if (phy_identifier == MARVEL_PHY_IDENTIFIER) {
    phy_model = phy_model & MARVEL_PHY_MODEL_NUM_MASK;

    if (phy_model == MARVEL_PHY_88E1116R_MODEL) {
        return get_phy_speed_88E1116R(xaxiemacp, phy_addr);
    } else if (phy_model == MARVEL_PHY_88E1111_MODEL) {
        return get_phy_speed_88E1111(xaxiemacp, phy_addr);
    }
}
```

改为：

```
if (phy_identifier == MARVEL_PHY_IDENTIFIER) {
    phy_model = phy_model & MARVEL_PHY_MODEL_NUM_MASK;

    if (phy_model == MARVEL_PHY_88E1116R_MODEL) {
        return get_phy_speed_88E1116R(xaxiemacp, phy_addr);
    } else if (phy_model == MARVEL_PHY_88E1111_MODEL) {
        return get_phy_speed_88E1111(xaxiemacp, phy_addr);
    } else if (phy_model == MARVEL_PHY_88E1512_MODEL) {
        return get_phy_speed_88E1512(xaxiemacp, phy_addr);
    }
}
```

打开 lwip141_v1_74\src\contrib\ports\xilinx\netif\xaxiemacif_dma.c 源文件，在 init_axi_dma()函数中找到如下代码：

```
dmaconfig = XAxiDma_LookupConfig(XPAR_AXIDMA_0_DEVICE_ID);
```

改为：

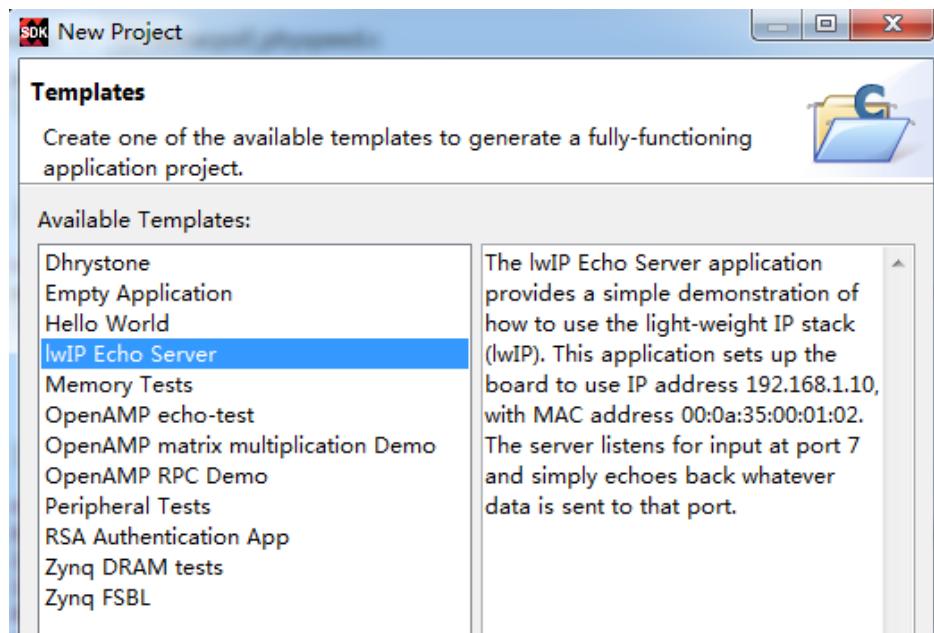
```
dmaconfig = XAxiDma_LookupConfig(xemac->topology_index);
```

这样是为了便于在设计中使用多个 AXI 1G/2.5G Ethernet Subsystem 和 AXI DMA 实现多个网口。

到此，LWIP 库的修改完成。

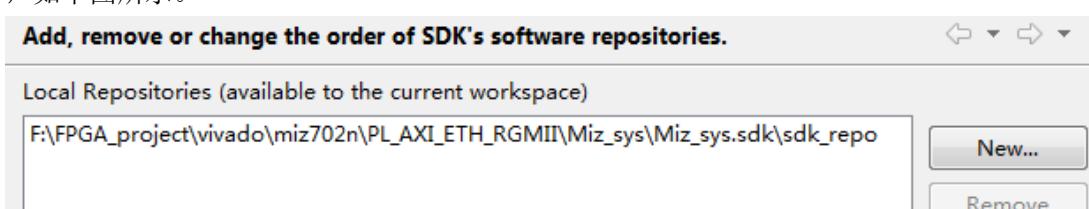
15.4.2 创建工程

本例程使用了 SDK 自带的 lwip echo server 例程来验证子卡电口和光口的功能，因此在创建工程时选择 LwIP Echo Server 模板，如下图所示。该例程基于 LWIP 库在 ARM 中建立一个 TCP Echo Server，IP 地址为 192.168.1.10，端口号为 7。



15.4.2.1 lwip 库设置

在 SDK 软件中选择修改后的 lwip 库的路径（需根据实际情况更改此路径，若不更改将产生错误），如下图所示。



修改完成后，打开 BSP 设置，此时 lwip echo server 例程使用的是 SDK 自带的 1.7 版本 LWIP 库，为了替换成我们所修改过的 1.74 版本，首先需要取消 lwip141 1.7 的勾选，如下图所示。



然后关闭 SDK，然后重启 SDK 打开该工程目录。然后打开 BSP 设置，此时 lwip 库便变成了所修改过的 1.74 版本，如下图所示。勾选 lwip141 v1.74 版本。

Supported Libraries		
Check the box next to the libraries you want included in your Board Support Package. You navigator on the left.		
Name	Version	Description
<input type="checkbox"/> libmetal	1.1	Libmetal Library
<input checked="" type="checkbox"/> lwip141	1.74	lwIP TCP/IP Stack library: lwIP v1.4.1
<input type="checkbox"/> openamp	1.2	OpenAmp Library
<input type="checkbox"/> xliffs	3.5	Generic Fat File System Library
<input type="checkbox"/> xilflash	4.2	Xilinx Flash library for Intel/AMD CFI compliant ...
<input type="checkbox"/> xilisf	5.7	Xilinx In-system and Serial Flash Library
<input type="checkbox"/> xilmfs	2.2	Xilinx Memory File System
<input type="checkbox"/> xilpm	2.0	Power Management API Library for ZynqMP
<input type="checkbox"/> xilrsa	1.2	Xilinx RSA Library
<input type="checkbox"/> xilskey	6.1	Xilinx Secure Key Library

将 use_axieth_on_zynq 设置为 1，表示此时不使用 PS 内部的 GEMAC 和 DMA，而是使用 PL 部分的 AXI 1G/2.5G Ethernet Subsystem 和 AXI DMA 来实现。将 use_emaclite_on_zynq 设为 0。如下图所示。

use_axieth_on_zynq	1	1	integer
use_emaclite_on_zynq	0	1	integer

在 lwip 库增加的选项中，当子卡的电口工作时，use_1000basex_on_88e1512 设置为 false；当工作于光口模式时，将其设为 true 即可。如下图所示。

use_1000basex_on_88e1512	true	false	boolean	Settings for operation mode
--------------------------	------	-------	---------	-----------------------------

在 teamc_adapter_options 中，将 tcp_ip_tx_checksum_offload 和 tcp_ip_rx_checksum_offload 设为 true，这是因为在 AXI 1G/2.5G Ethernet Subsystem 的配置中启用了该功能。

n_rx_coalesce 和 n_tx_coalesce 表示 AXI DMA 在工作于链式 DMA 模式时，触发 1 次中断所需要完成传输的链表中 descriptor 的个数。将该值设大将会减少 AXI DMA 中断的触发次数，提高链式 DMA 的工作效率和 PS 的运行效率，可提高网口传输速率。

n_rx_descriptors 和 n_tx_descriptors 表示 AXI DMA 工作于链式 DMA 模式时，接收和发送链表中 descriptor 的个数，descriptor 的个数越大，链式 DMA 的工作效率将会提高，网口的传输速度也将会提高。最大可设为 256。

设置如下图所示。

temac_adapter_options	true	true	boolean
emac_number	0	0	integer
n_rx_coalesce	8	1	integer
n_rx_descriptors	64	64	integer
n_tx_coalesce	8	1	integer
n_tx_descriptors	64	64	integer
phy_link_speed	Autodetect (CONFIG_LI...)	CONFIG_LINKSPEED_A...	enum
tcp_ip_rx_checksum_offload	true	false	boolean
tcp_ip_tx_checksum_offload	true	false	boolean
tcp_rx_checksum_offload	false	false	boolean
tcp_tx_checksum_offload	false	false	boolean
temac_use_jumbo_frames	false	false	boolean

其余选项的参数默认即可，不用修改。

15.4.2.2 example 代码修改

工程代码无需作任何修改。

15.5 程序测试

15.5.1 电口测试

15.5.1.1 lwip 库设置

电口模式下 lwip 的设置如下图所示。将 use_1000basex_on_88e1512 设为 false。

use_1000basex_on_88e1512	false	false	boolean
--------------------------	-------	-------	---------

15.5.1.2 网络测试

将千兆网线插入子卡的 RJ45 座中，与电脑连接，将电脑的 ip 地址设为 192.168.1.100，子网掩码为 255.255.255.0。然后给开发板上电，通过 SDK 将程序下载入开发板后，观察 SDK 串口打印信息，如下图所示。表示千兆网络连接正常。

```
Connected to: Serial ( COM14, 115200, 0, 8 )
```

```
-----lwIP TCP echo server -----
```

```
TCP packets sent to port 6001 will be echoed back
```

```
Start PHY autonegotiation
```

```
Waiting for PHY to complete autonegotiation.
```

```
autonegotiation complete
```

```
auto-negotiated link speed: 1000
```

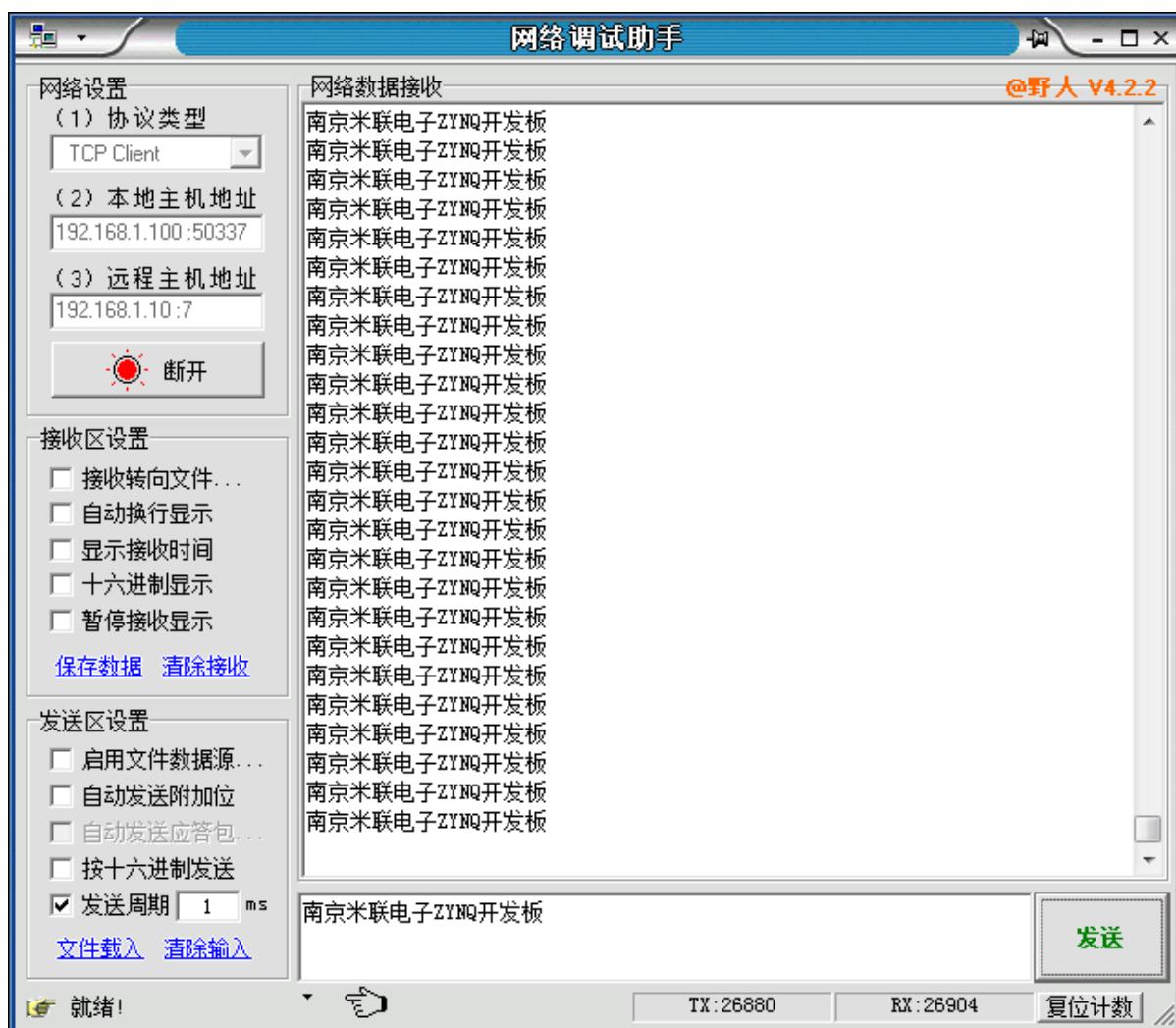
```
Board IP: 192.168.1.10
```

```
Netmask : 255.255.255.0
```

```
Gateway : 192.168.1.1
```

```
TCP echo server started @ port 7
```

打开网络调试助手，以 TCP Client 模式连接开发板的 IP 地址 192.168.1.10，端口号 7，输入文字发送，网络调试助手便可收到开发板返回相同的文字，如下图所示。



15.5.2 光口测试

15.5.2.1 lwip 库设置

光口模式下 lwip 的设置如下图所示，要将 use_1000basex_on_88e1512 设为 true。

use_1000basex_on_88e1512	true	false	boolean	Settings for operation mode
--------------------------	------	-------	---------	-----------------------------

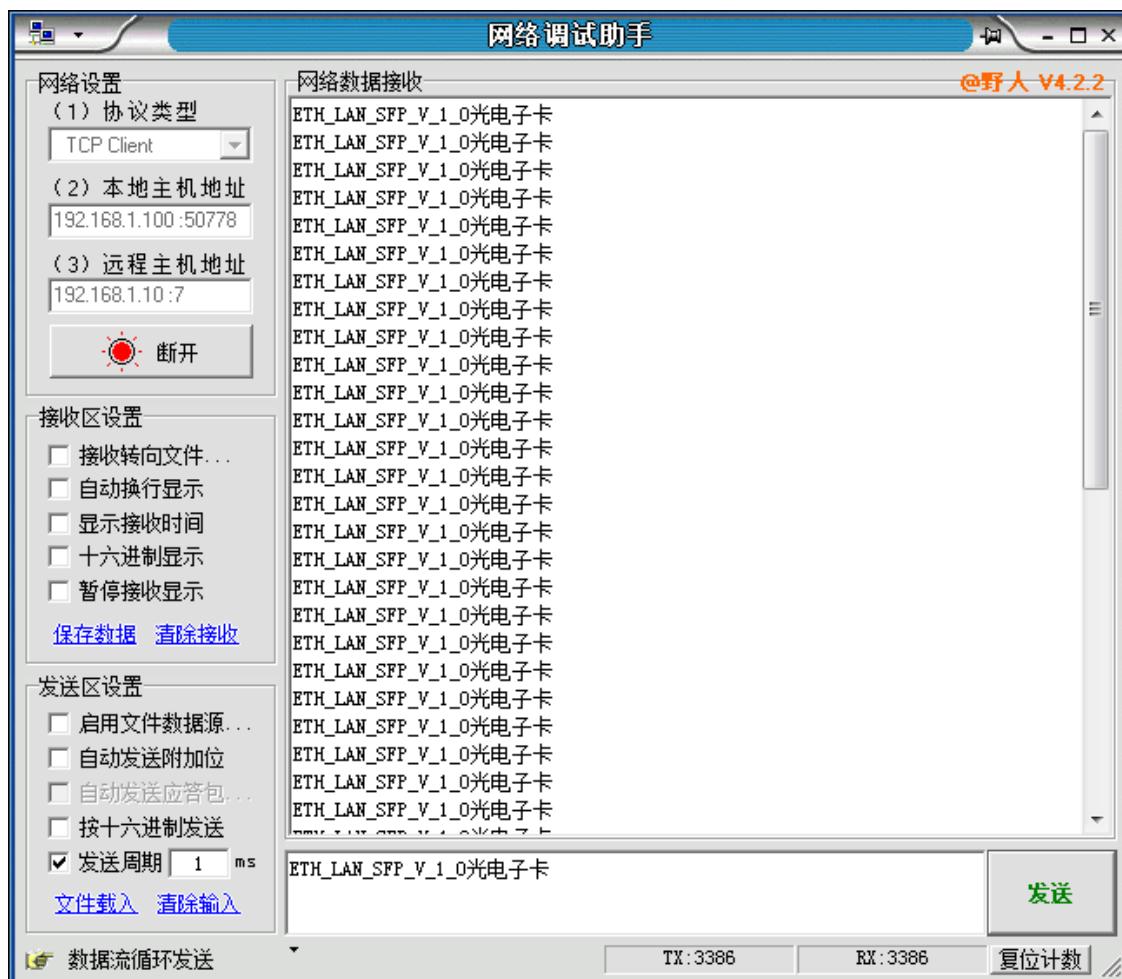
15.5.2.2 网络测试

将 SFP 电口模块插入子卡的 SFP 屏蔽笼中，将千兆网线插入 SFP 电口模块，与电脑连接，将电脑的 ip 地址设为 192.168.1.100，子网掩码为 255.255.255.0。然后给开发板上电，通过 SDK 将程序下载入开发板后，观察 SDK 串口打印信息，如下图所示。

Connected to: Serial (COM14, 115200, 0, 8)

```
-----lwIP TCP echo server -----
TCP packets sent to port 6001 will be echoed back
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
auto-negotiated link speed: 1000
Board IP: 192.168.1.10
Netmask : 255.255.255.0
Gateway : 192.168.1.1
TCP echo server started @ port 7
```

同理，打开网络调试助手，以 TCP Client 模式连接开发板的 IP 地址 192.168.1.10，端口号 7，输入文字发送，网络调试助手便可收到开发板返回相同的文字，如下图所示。



CH16 基于 μGUI 的触摸屏 GUI 界面设计

16.1 概述

很多工程师都在 ZYNQ 上做过 LINUX 相关的应用和开发，在 ZYNQ 运行 LINUX 操作系统，可以实现非常细致的 GUI 图形界面。用户可以通过鼠标、键盘等外部设备与操作系统通过 GUI 界面进行人机交互，与平时使用电脑的体验相当。

但同时，也有很用户只涉及 ZYNQ 的裸机开发，那么在无操作系统支持的情况下，是否可以在裸机环境中构建一个 GUI 图形界面呢？

答案是肯定的。本例程通过利用开源 GUI 库 μ GUI，在 ZYNQ 裸机环境下创建一组简单的 GUI 界面，并通过外部液晶触摸屏实现与 ZYNQ 的人机交互。本例程所涉及的应用知识点如下：

- LCD 触摸屏的使用原理
- 移植开源 μGUI 库，利用 API 函数搭建一组 GUI 图形界面
- 通过 VDMA、AXI-S Video Out、VTC 等 IP 实现 GUI 图形界面的显示
- 通过定时器中断实现 GUI 界面的周期性刷新
- 通过 PS 实现动态配置 MMCM/PLL
- 通过 PS 动态配置 AXI PWM 调节液晶屏亮度
- 通过 AXI GPIO 检查触摸屏的中断信号
- 通过 I2C 读取触摸屏的触摸信息
- 通过 PS 设置 Video Timing Controller 显示分辨率
- 设计 GUI 界面的动态变化机制，将触摸信息反馈至 GUI 界面后，GUI 产生动态变化，形成人机交互。

本例程基于 Vivado 2016.4 版本开发。

16.2 基本原理

本例程通过开源的 μ GUI v0.3 库设计了 1 个 GUI 界面，为该 GUI 界面设计了 5 个窗口，为每个窗口设计了标题、按键、文本、图片 logo 等元素，并给每个按键设定了相应功能，如窗口间切换、LED 灯控制、屏幕亮度调节等，并在 1 个窗口中设计了绘图功能。然后，将设计的 GUI 窗口通过触摸屏显示。用户通过按下触摸屏中所显示的窗口对应的按键位置，便可以实现该按键所设定的相应功能，从而实现人机交互。

16.3 LCD 触摸屏

LCD 触摸屏由 LCD 液晶屏和触摸屏两部分组成，其中 LCD 液晶屏用于画面显示，触摸屏用于实现触摸控制。本例程中所使用的是微雪公司的 7 寸 LCD 电容触摸屏，如下图所示。



其中，LCD 液晶屏采用了 24 位 RGB888 接口。电容触摸屏采用了 I2C 接口。LCD 触摸屏接口定义如下图所示。其中，1~35 为液晶屏接口，37~40 为触摸屏接口。

RGB接口定义

引脚号	标识	描述	类型	功能
1	BL_VDD	电源正	电源型	背光电源，一般接5V
2	BL_VDD			GND
4	VDD			一般接3.3V
5	R0	数据线	输入	红色调色板数据线
6	R1			
7	R2			
8	R3			
9	R4			
10	R5			
11	R6			
12	R7			
13	G0			
14	G1			
15	G2	数据线	输入	绿色调色板数据线
16	G3			
17	G4			
18	G5			
19	G6			
20	G7			

21	B0			
22	B1			
23	B2			
24	B3			
25	B4	数据线	输入	蓝色调色板数据线
26	B5			
27	B6			
28	B7			
29	GND	电源地	电源型	GND
30	DCLK	LCD时钟	输入	LCD时钟信号源
31	DISP	背光控制使能	输入	控制使能背光控制，一般接VDD
32	H SYNC	行同步	输入	水平同步信号输入
33	V SYNC	帧同步	输入	垂直同步信号输入
34	DE	控制模式选择	输入	DE=0:SYNC模式 DE=1:DE模式
35	PWM	背光明暗调节	输入	PWM调节背光信号线
36	GND	电源地	电源型	GND
37	I2C_SDA	I2C数据脚	输出/输入	I2C数据传输脚，读写数据
38	I2C_SCL	I2C时钟脚	输入	I2C时钟控制脚，控制时钟
39	CAP_WAKE	WAKEUP引脚	输出	用于唤醒外部TP控制器
40	CAP_INT	中断引脚	输出	外部触摸中断控制引脚

16.3.1 液晶屏

液晶屏为 1024×600 分辨率，这个分辨率不常用，在网上很难找到统一的标准。在该液晶屏 HJ070NA-13A 的手册中可以找到此液晶屏在该分辨率下的时序要求，如下图所示。

Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Clock Frequency	fclk	40.8	51.2	67.2	MHz	Frame rate =60Hz
Horizontal display area	thd	1024			DCLK	
HS period time	th	1114	1344	1400	DCLK	
HS Blanking	thb	90	320	376	DCLK	
Vertical display area	tvd	600			H	
VS period time	tv	610	635	800	H	
VS Blanking	thb	10	35	200	H	

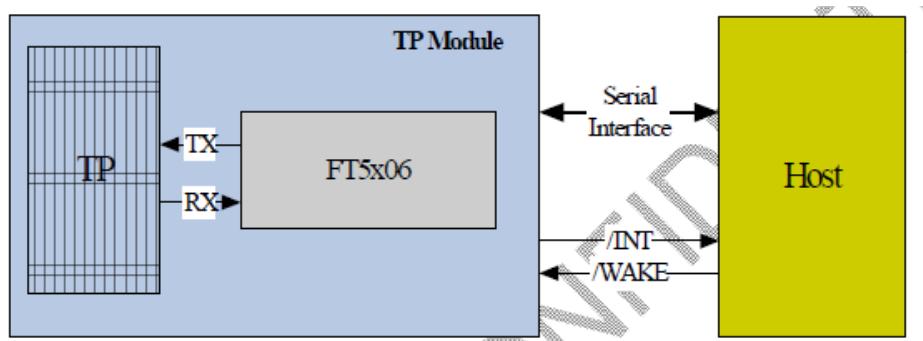
本例程采用了 51.2MHz 像素时钟所对应的参数设置。液晶屏的驱动方法与显示器类似，通过时钟，行、场同步信号，数据有效信号来完成，此处不作赘述。

要使液晶屏正常显示，背光源使能信号 DISP 要拉高，通过调节 PWM 的占空比可以改变背光源的亮度。该液晶屏的 PWM 为负极性，即低电平占空比越高，背光源越亮，

PWM 信号的频率范围为 100Hz~200KHz。

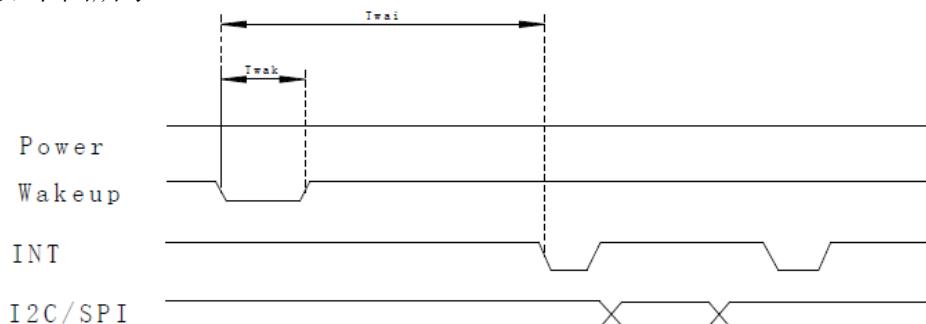
16.3.2 触摸屏

触摸屏为电容屏，采用了 FT5206 作为主控芯片，支持 5 点触控。触摸屏与 ZYNQ 的接口如下图所示。



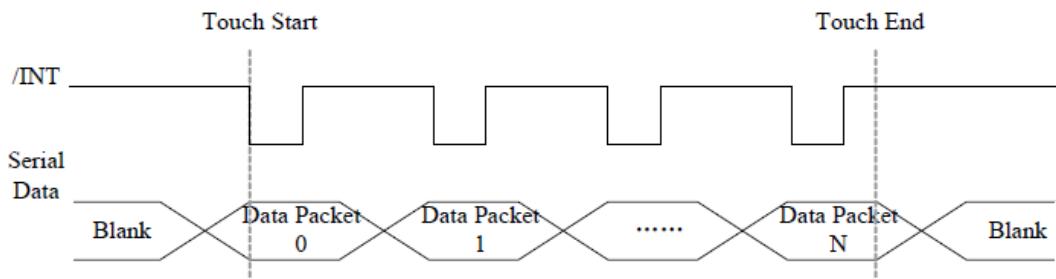
16.3.3 触摸屏唤醒

在触摸屏与 ZYNQ 的接口中，WAKE 信号为低电平有效，ZYNQ 通过拉低 WAKE 信号若干毫秒，再将其拉高来唤醒 FT5026 芯片，当触摸屏正常工作时 WAKE 信号应恒为高电平。如下图所示。



16.3.4 触摸中断

INT 信号也为低电平有效。当手指触摸电容屏时，INT 信号会以固定频率（默认为 60Hz）脉冲信号的形式输出，当手指离开电容屏，INT 信号重新恢复高电平，如下图所示。



16.3.5 触摸信息获取

每当 INT 变为低电平时, ZYNQ 就立即通过 I2C 接口读出 FT5206 芯片内部记录的触摸坐标信息, 完成一次触摸响应。

FT5206 芯片 I2C 接口作为 slave 从设备, 最高速率为 400KHz, I2C 地址为 0x38。这里补充说明一点, 关于 I2C 地址芯片 datasheet 中作了如下图所示的描述, 笔者在搜集到的关于 FT5206 的资料中均未能找到关于 I2CCON register 的说明。后来, 通过网络搜索发现使用过该系列芯片的记录中, 所提到的 I2C 地址均为 0x38, 笔者通过尝试得到了验证。

A[6:0]	Slave address
	A[6:4]: 3'b011
	A[3:0]: data bits are identical to those of I2CCON[7:4] register.

FT5206 芯片通过一系列寄存器记录与触摸相关的信息。相关寄存器如下图所示。

Address	Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Host Access
Op.00h	DEVIDE_MODE				Device Mode[2:0]					RW
Op.01h	GEST_ID				Gesture ID[7:0]					R
Op.02h	TD_STATUS					Number of touch points[3:0]				R
Op.03h	TOUCH1_XH		1 st Event Flag			1 st Touch X Position[11:8]				R
Op.04h	TOUCH1_XL				1 st Touch X Position[7:0]					R
Op.05h	TOUCH1_YH			1 st Touch ID[3:0]		1 st Touch Y Position[11:8]				R
Op.06h	TOUCH1_YL				1 st Touch Y Position[7:0]					R
Op.07h										
Op.08h										
Op.09h	TOUCH2_XH		2 nd Event			2 nd Touch				R

		Flag		X Position[11:8]	
Op,0Ah	TOUCH2_XL	2 nd touch X Position[7:0]		R	
Op,0Bh	TOUCH2_YH	2 nd Touch ID[3:0]	2 nd Touch Y Position[11:8]		R
Op,0Ch	TOUCH2_YL	2 nd Touch Y Position[7:0]		R	
Op,0Dh				R	
Op,0Eh				R	
Op,0Fh	TOUCH3_XH	3 rd Event Flag		3 rd Touch X Position[11:8]	R
Op,10h	TOUCH3_XL	3 rd Touch X Position[7:0]		R	
Op,11h	TOUCH3_YH	3 rd Touch ID[3:0]	3 rd Touch Y Position[11:8]		R
Op,12h	TOUCH3_YL	3 rd Touch Y Position[7:0]		R	
Op,13h				R	
Op,14h				R	
Op,15h	TOUCH4_XH	4 th Event Flag		4 th Touch X Position[11:8]	R
Op,16h	TOUCH4_XL	4 th Touch X Position[7:0]		R	
Op,17h	TOUCH4_YH	4 th Touch ID[3:0]	4 th Touch Y Position[11:8]		R
Op,18h	TOUCH4_YL	4 th Touch Y Position[7:0]		R	
Op,19h				R	
Op,1Ah				R	
Op,1Bh	TOUCH5_XH	5 th Event Flag		5 th Touch X Position[11:8]	R
Op,1Ch	TOUCH5_XL	5 th Touch X Position[7:0]		R	
Op,1Dh	TOUCH5_YH	5 th Touch ID[3:0]	5 th Touch Y Position[11:8]		R
Op,1Eh	TOUCH5_YL	5 th Touch Y Position[7:0]		R	

其中，TD_STATUS 寄存器记录了触摸点数，如下图所示。

2.1.3 TD_STATUS

This register is the Touch Data status register.

Address	Bit Address	Register Name	Description
Op,02h	3:0	Number of touch points[3:0]	How many points detected. 1-5 is valid.
	7:4		

记录触摸坐标、触摸动作的寄存器如下图所示。

2.1.4 TOUCHn_XH (n:1-5)

This register describes MSB of the X coordinate of the nth touch point and the corresponding event flag.

Address	Bit Address	Register Name	Description
Op,03h ~ Op,39h	7:6	Event Flag	00b: Put Down 01b: Put Up 10b: Contact 11b: Reserved
	5:4		Reserved
	3:0	Touch X Position [11:8]	MSB of Touch X Position in pixels

2.1.5 TOUCHn_XL (n:1-5)

This register describes LSB of the X coordinate of the nth touch point.

Address	Bit Address	Register Name	Description
Op,04h ~ Op,3Ah	7:0	Touch X Position [7:0]	LSB of the Touch X Position in pixels

2.1.6 TOUCHn_YH (n:1-5)

This register describes MSB of the Y coordinate of the nth touch point and corresponding touch ID.

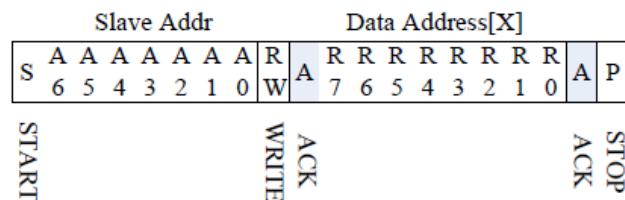
Address	Bit Address	Register Name	Description
Op,05h	7:4	Touch ID[3:0]	Touch ID of Touch Point
~ Op,3Bh	3:0	Touch X Position [11:8]	MSB of Touch Y Position in pixels

2.1.7 TOUCHn YL (n:1-5)

This register describes LSB of the Y coordinate of the nth touch point.

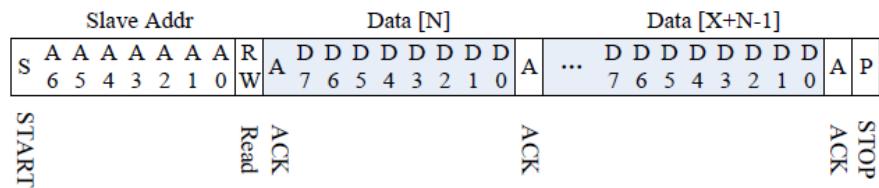
Address	Bit Address	Register Name	Description
Op,06h ~ Op,3Ch	7:0	Touch X Position [7:0]	LSB of The Touch Y Position in pixels

ZYNQ 通过 I2C 读取这些寄存器值分为两个步骤。首先，发送需要连续读取的起始寄存器地址，如下图所示。



然后，连续读取若干个地址连续的寄存器值，如下图所示。

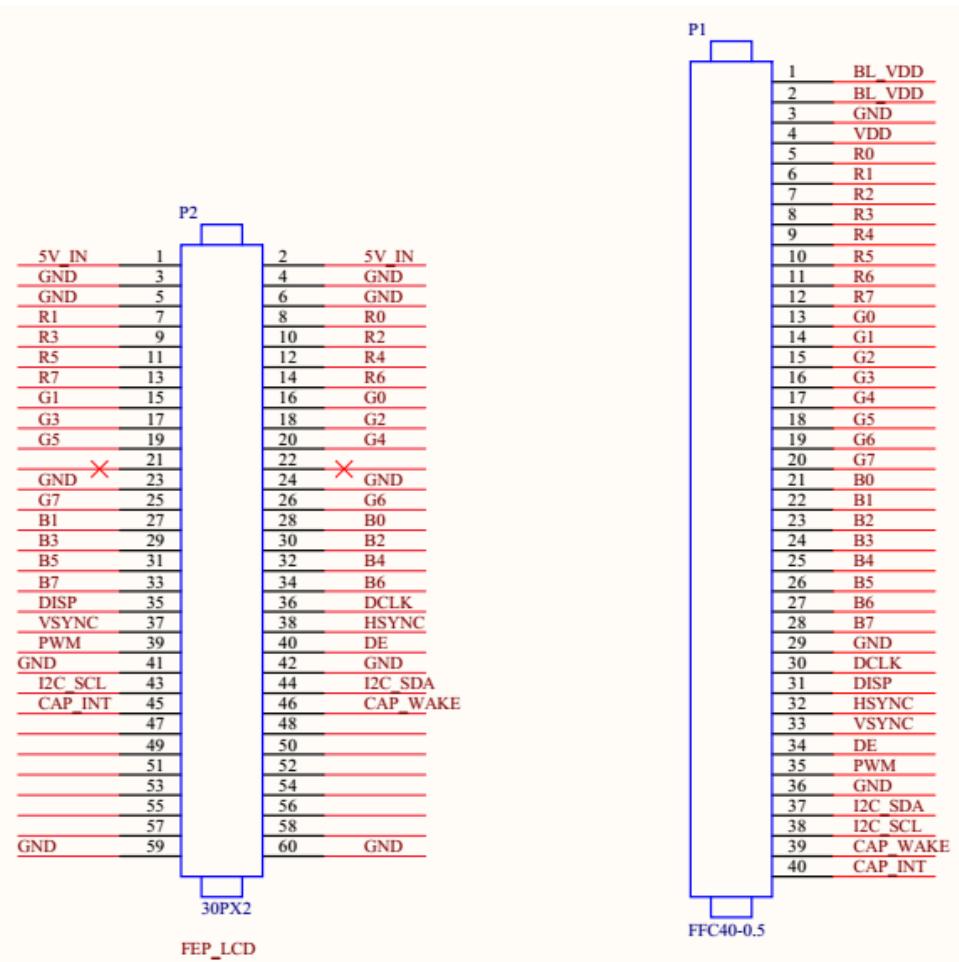
Read X bytes from I²C Slave



16.3.6 触摸屏与 Miz ZYNQ 开发板的接口

由于本例程所使用的 LCD 触摸屏接口为 FPC，而开发板并不具有 FPC 接口。因此，需要设计一个转接板将触摸屏与开发板的 40pin NEP 接口连接。需要注意的是，在该触摸屏内部，I²C 信号线均未接上拉电阻。因此，在转接板中需要将两个 I²C 信号各通过 1 个几 KΩ 的电阻上拉接至 3.3V，否则 I²C 接口将无法正常使用。

NEP 接口定义如下图所示。



16.4 μ GUI 概述

μ GUI 是一个开源的 GUI 图形界面设计库，包含了窗口（window），按键（button），文本框（textbox），图片（image）等 4 种元素。用户可以根据它们所对应的 API 函数自由发挥，对这些元素的数量、颜色、大小、位置、功能等参数进行定义，实现简易的 GUI 界面设计。

μ GUI 具有一个特色，提供了专门的 API 函数来支持外部的二维坐标（X, Y）输入设备，例如，触摸屏。用户所创建的 GUI 界面可以通过这些二维坐标获取外界物体输入的交互信息（比如触摸具有某个特定功能的按键），以此为基础实现人机交互。

该库一共包含 ugui.c 和 ugui.h 两个文件。关于 μ GUI 库更为详细的介绍以及相应 API 函数的使用可参考其使用手册。读者可以访问其所在网站 www.embeddedlightning.com，可以下载源文件、使用手册以及 example project。

16.4.1 μGUI 库移植

由于 μGUI 是一个跨平台（DSP、ARM、MCU、单片机等）的 C 语言库，在将 μGUI 的 ugui.c 和 ugui.h 文件移植到 ZYNQ 的 ARM 中，并通过 SDK 进行程序设计和编译之前，需要完成 2 个重要工作：

16.4.1.1 添加单像素点像素值设置函数

1 个 GUI 界面其实就是 1 幅图像，1 幅图像由很多个像素点组成。要设计 1 个 GUI 界面，就需要设置其中每个像素点的像素值。简单的说，就是要给 GUI 界面对应图像所在存储区域的各个地址赋值。但针对不同的平台，不同的应用，GUI 界面的存储方式会存在不同，因此像素值设置的方式会有区别。出于跨平台的目的，μGUI 给出了这个函数的原型声明，提供了开放接口，由用户自行设计这个函数来完成 GUI 界面各像素点值的设置，μGUI 中所有与界面设计相关的 API 函数都将调用这个函数。本例程在 gui_window.c 中所设计的函数如下：

```
voidPixelSet(UG_S16 x, UG_S16 y, UG_COLOR c)
{
    u32iPixelAddr;
    iPixelAddr = y * GUI_WIDTH + x;
    BufferPtr[0][iPixelAddr] = c;
}
```

其中 x, y 对应像素点的二维坐标（X,Y），c 为该像素点所需设置的像素值，位宽 32bit。由于本例程中 GUI 界面位于 DDR 中连续的内存区域，通过 GUI 界面的指针 BufferPtr [0] 以及 x, y 便可计算出像素点的地址。

16.4.1.2 调整各种变量类型定义

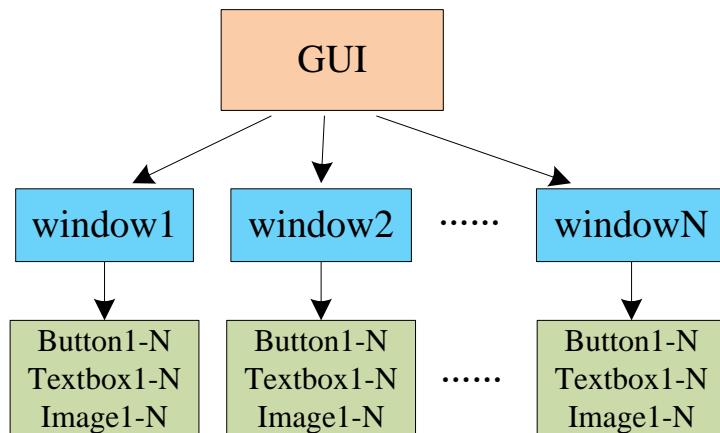
在不同的嵌入式平台中，对于各种变量类型的定义可能会存在区别，例如，int型变量在不同的平台中可能是64位、32位或者16位的。因此，需要在ugui.h中调整各变量类型的定义。ugui.h中默认的变量定义如下：

```
typedef uint8_t U8;
typedef int8_t S8;
typedef uint16_t U16;
typedef int16_t S16;
typedef uint32_t U32;
typedef int32_t S32;
```

上述变量定义与ZYNQ平台编译器一致，因此无需修改，只需在ugui.h文件中将#include "system.h"改为#include "stdint.h"即可。

16.4.1.3 GUI 窗口

窗口是μGUI中GUI界面必需的基本组成，不可缺少，没有窗口就无法创建GUI界面。而文本框、按键、图片等对象则是可选项，在GUI界面中可有可无。μGUI中的窗口包含了文本框、按键、图片3种基本对象，每个窗口都可以拥有若干个多种对象。简而言之，1个窗口可以包含若干个对象，而1个GUI界面可以包含若干个窗口，它们之间的关系如下图所示。



16.4.1.4 GUI 界面刷新

作为GUI界面，应该具有动态变化的特性。例如，不同窗口间切换、文本框内容变化、按键位置调整等。μGUI提供了一个函数UG_Update()来实现整个GUI界面的刷新。当用户通过μGUI库中某些API函数改变当前窗口中某些对象的特性后（例如，文字、颜色、大小），就必须尽快调用UG_Update()来确保这些更改被刷新到GUI界面上。若不调用UG_Update()，则GUI界面将永远不会发生任何改变。

因此，在使用μGUI库进行GUI界面设计时，为了保证其动态变化的特性，用户必须要周期性的调用UG_Update()函数，调用频率的高低可根据实际要求来确定。

16.4.1.5 人机交互

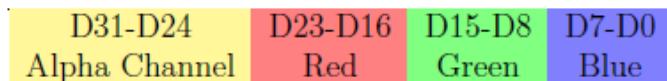
GUI 界面的动态变化特性，其中很大一部分都源自于人机交互，上面提到了μ GUI 库支持外部的二维坐标（X, Y）输入设备，由外部输入的这些二维坐标就是 GUI 界面所需要的人机交互信息。例如，当设备与触摸屏连接，用户触碰了触摸屏的某个区域，触摸屏将这个区域的中心二维坐标反馈给设备，设备再将坐标输入 GUI 界面。

那如何将人机交互与 GUI 界面的动态特性相关联呢？μ GUI 库提供了 UG_TouchUpdate() 函数，用于向 GUI 界面输入当前窗口中外部输入的二维坐标信息。通过该函数，GUI 界面可判断出该坐标对应当前窗口中具体哪个对象，例如，某个按键。那当知道了某个对象被触摸后，是否应该作出，或者具体如何作出动态变化来响应这个输入坐标呢？这完全由用户自己来进行定义。μ GUI 库为每个窗口提供了窗口回调函数来完成这个任务，并给出了回调函数的原型声明（详情参考 μ GUI 使用手册），函数的具体内容完全由用户自己设计。

窗口回调函数在 UG_Update() 中被调用，回调函数获取输入二维坐标对应的对象信息，用户可以据此自由发挥，设计出相应的动态变化效果。因此，GUI 的动态变化最终将由 GUI 界面刷新函数 UG_Update() 来实现。这也说明了为何必须周期性的调用 UG_Update() 来确保 GUI 界面的动态变化特性。

16.4.2 颜色空间

在 μ GUI 库中，GUI 界面里每个像素点的像素值都是 32 位的无符号整型变量，采用 ARGB888 颜色空间，如下图所示。然而，最新的版本 μ GUI v0.3 只使用了其中的 RGB888 部分。因此，对于用户来说，GUI 界面就是一幅 24 位的 RGB888 彩色图像，高 8 位完全忽略。



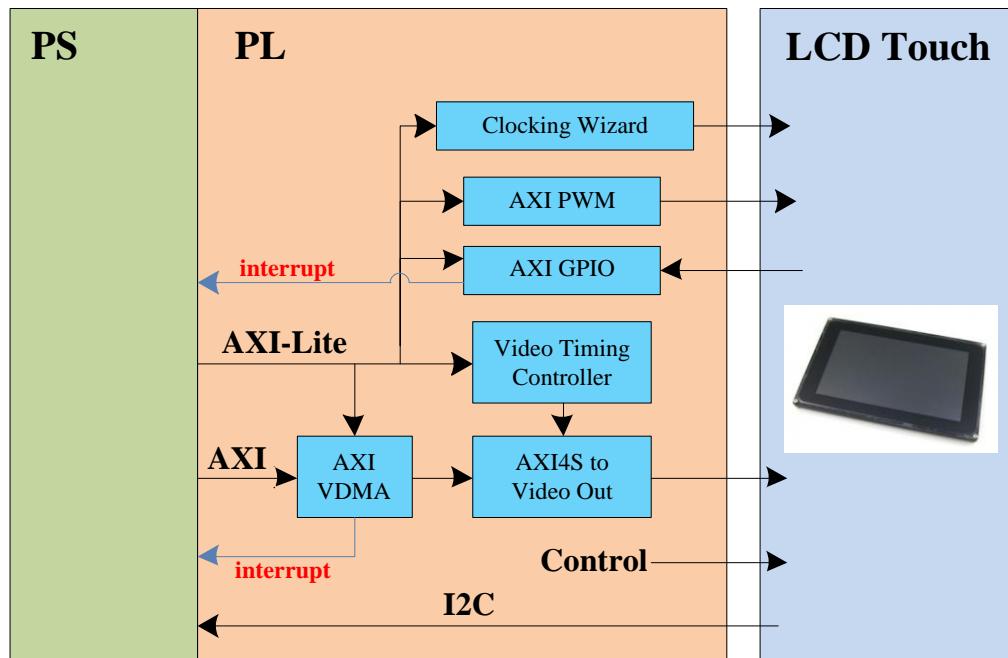
在 μ GUI 中，定义了上百种 RGB888 的颜色种类，用户可根据自己的喜好为 GUI 界面中的窗口、按键、文本等对象选择相应的颜色进行设计。

16.4.3 字体大小

μ GUI 库中定义了十几种大小不同的字体，用于 GUI 界面中窗口、文本框、按键等对象的文本显示。

17.5 PL 逻辑框架

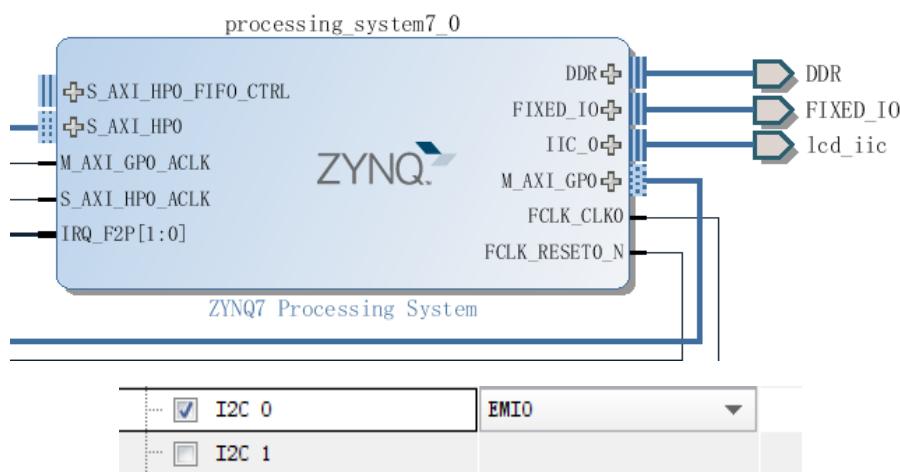
本例程的 PL 部分设计原理如下图所示。



16.5.1 PS 设置

PS 部分的设置下图所示。

- 使能 M_GP0、S_HPO，PL 到 PS 的中断接口。
- 使能 I2C0 接口，并将其通过 EMIO 方式引出。I2C0 接口与触摸屏连接，用于读取触摸屏的触摸坐标信息。
- 使能 FCLK_CLK0，时钟频率设置为 100MHz。



16.5.2 GUI 界面显示

本例程中 GUI 界面的显示通过 AXI VDMA、AXI4-S to Video Out、Video Timing

Controller 三个 IP 核完成。显示分辨率为 1024*600@60Hz。

16.5.2.1 AXI VDMA 设置

- AXI VDMA 与 AXI DMA 使用对比

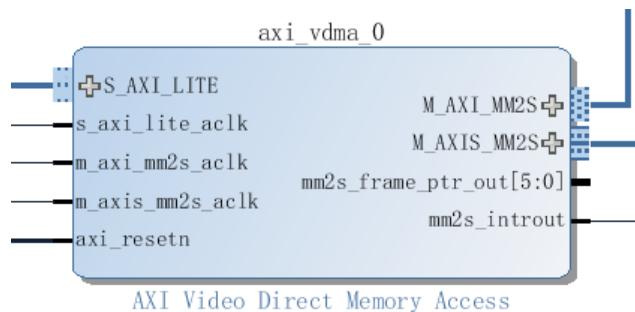
在米联第三季教程中有关于通过 AXI DMA 实现摄像头图像显示的例程。然而在本例程中，AXI DMA 并不适用。原因如下：

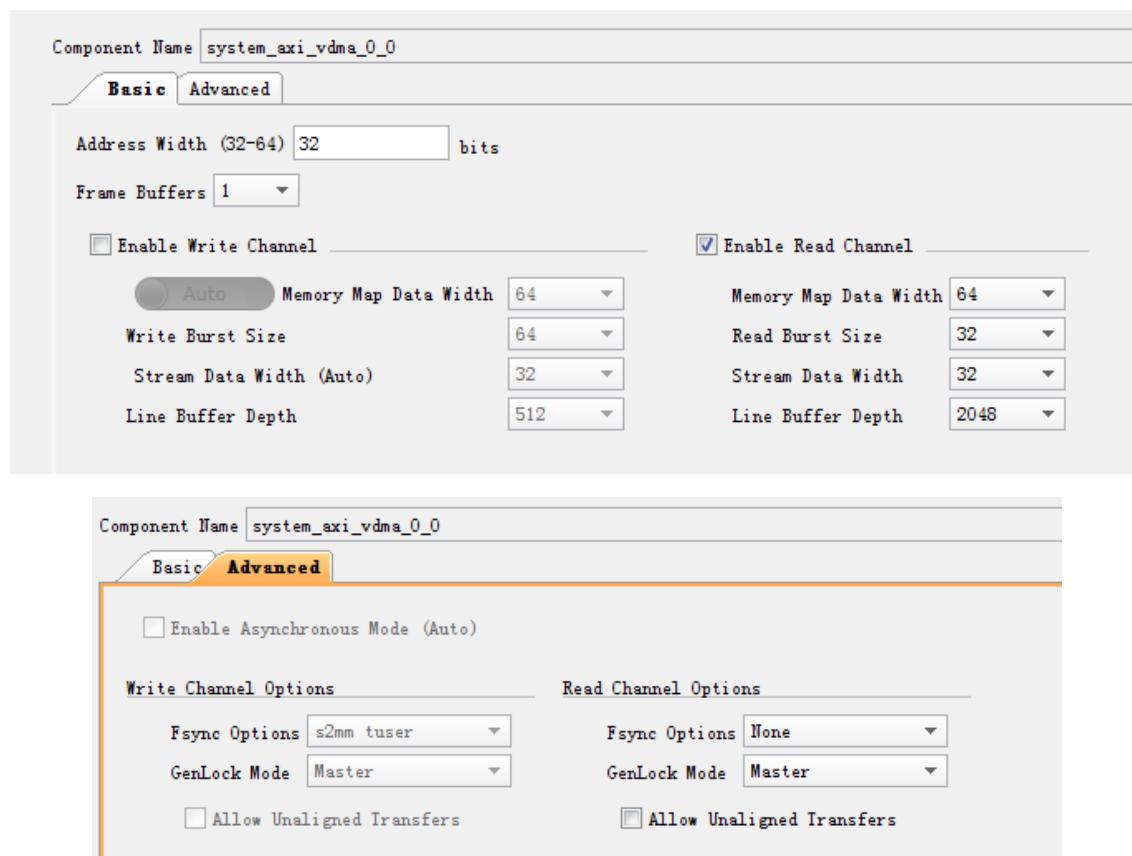
当使用 AXI DMA 时，PS 每次向 PL 发送 GUI 界面图像都需要通过调用函数配置 PL 的 AXI DMA 才能完成。在米联第三季中关于 AXI DMA 图像显示的例程中，都是通过在 AXI DMA 发送完成中断函数中发起下一次图像传输的方式来实现连续的图像发送。同时，在本例程中，GUI 界面的刷新过程是在定时器中断函数中完成，当触摸屏以 60Hz 显示时，1 幅图像的显示时间为 17.67ms。若 GUI 界面刷新的时间超过触摸屏中 1 幅图像显示的时间时，AXI DMA 图像发送完成中断将不能被立即响应执行，会导致图像显示发生中断。当 GUI 界面刷新完成 CPU 退出定时器中断后，AXI DMA 图像发送才能被继续执行，重新恢复 GUI 界面的显示，这样在每次 GUI 界面刷新的时候就会产生“跳屏”的现象。总而言之，定时器中断函数的执行时间会影响 AXIDMA 发送中断函数的执行。笔者已使用 AXI DMA 验证了这个现象。

而 VDMA 具有 free run 的特点，GUI 界面图像的发送可由 PL 的 VDMA 独自完成，无需 PS 参与，因此，定时器中断中 GUI 界面刷新与图像显示不会产生任何冲突，可以保证 GUI 界面显示的连续性。因此，在本例中使用 VDMA 更合适。

- AXI VDMA 设置

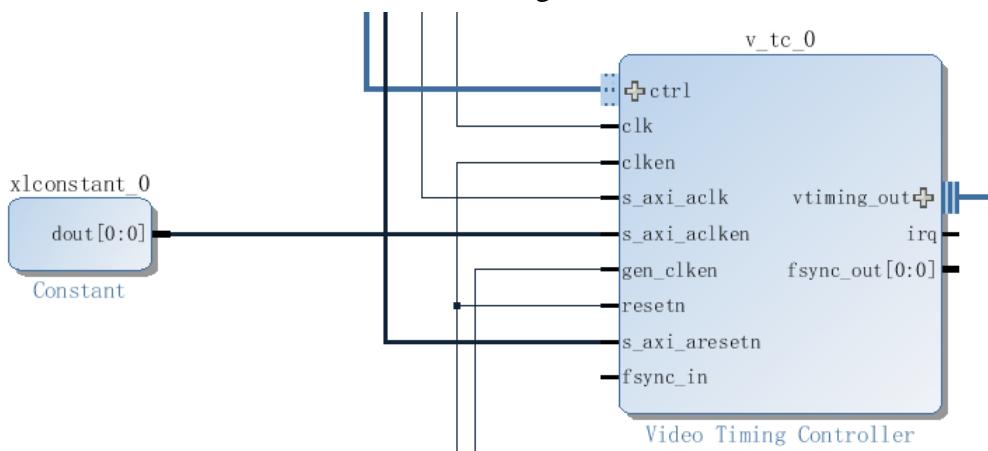
在本例程中，只存在 PS 的 DDR 向 PL 的图像传输，即 Memory Map to Stream (MM2S) 方向。因此，只需使能读通道，可关闭写通道来节省逻辑资源。由于 GUI 界面只对应 DDR 中的同一幅图像，只需 1 个图像缓存，所以将 frame buffer 设置为 1 即可。另外，将中断输出与 PS 连接。AXI VDMA 的设置如下图所示。

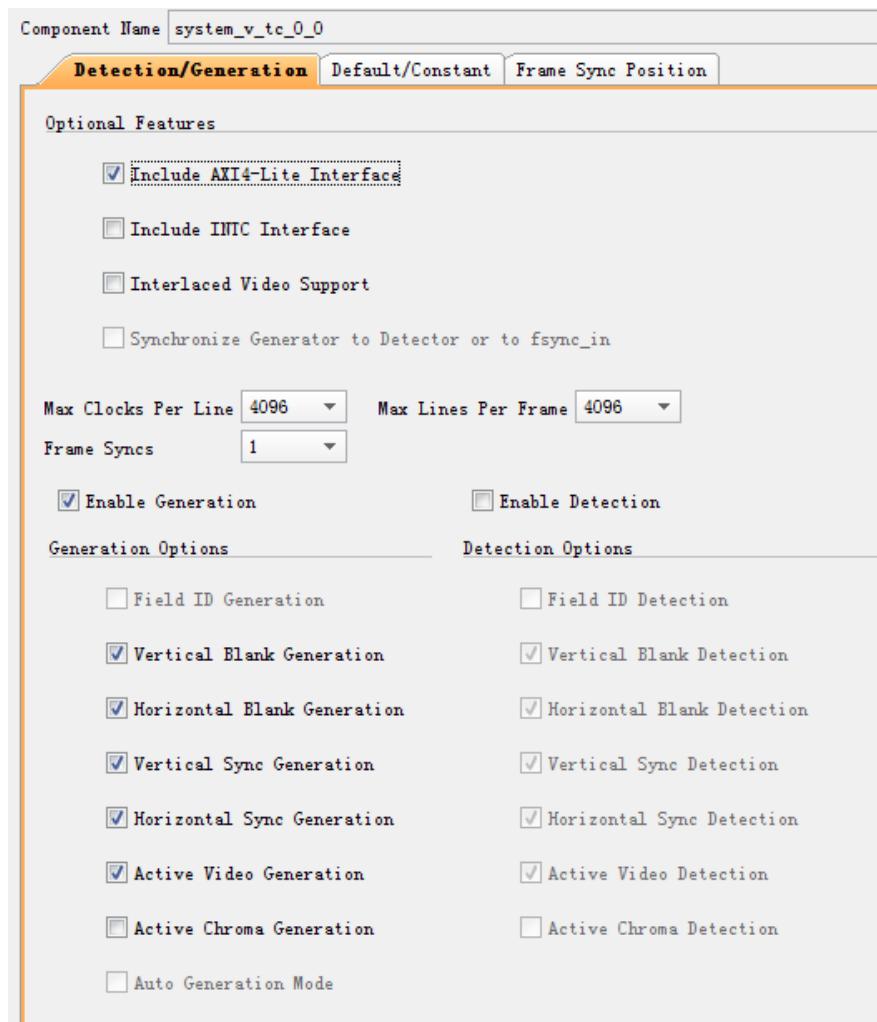




16.5.2.2 Video Timing Controller

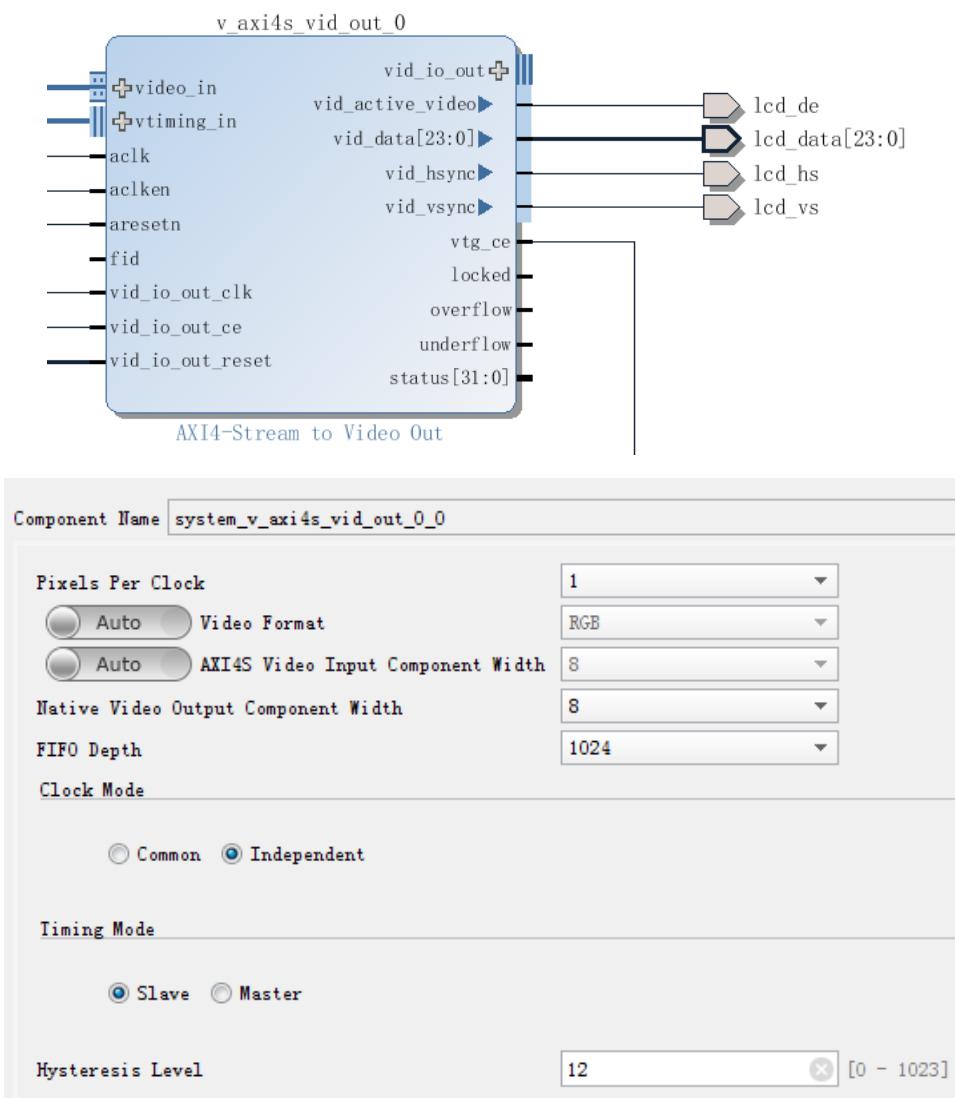
为了方便 PS 动态配置 Video Timing Controller 的显示分辨率，使能 AXI-lite 接口，其默认的分辨率可不用进行设置。Video Timing Controller 的设置如下图所示。





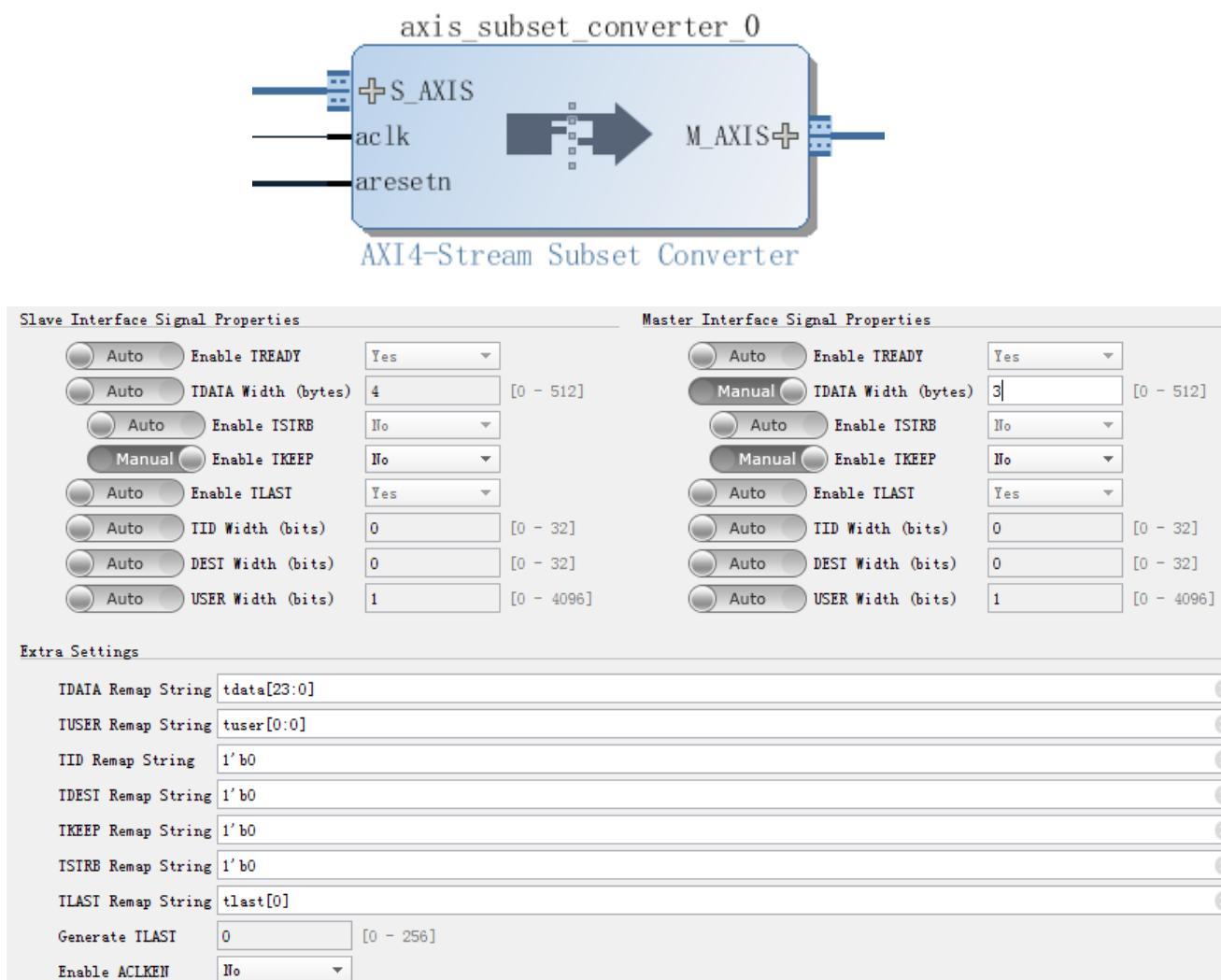
16.5.2.3 AXI4-Sream to Video Out

本例程中，输出至触摸屏的 GUI 图像为 RGB888 格式，AXI4- Sreamto Video Out 设置如下图所示。需要引出 RGB 数据 data、行同步信号 hs、场同步信号 vs 以及数据有效信号 de 与 LCD 触摸屏连接。



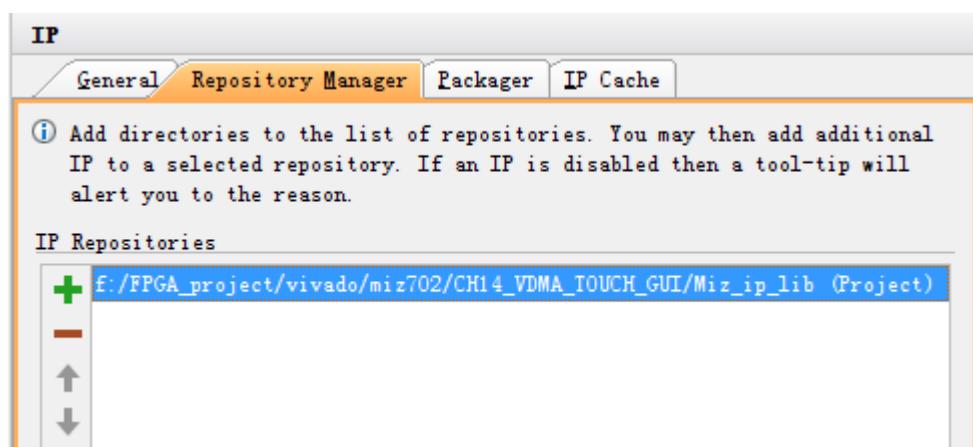
16.5.2.4 AXI-Stream Subset Converter

AXI VDMA 与 AXI4-Stream to Video Out 之间通过 AXI Stream 接口连接, 由于 AXI VDMA 的 M_AXIS_MM2S 接口的数据为 32 位, 且含有 4 位的 tkeep 信号。而 AXI4-Stream to Video Out 的 video_in 接口的数据为 24 位, 且无 tkeep 信号。为了更好的将两个信号之间存在差异的 Stream 接口连接, 通过 AXI-Stream Subset Converter 完成差异信号的重映射。设置如下图所示。

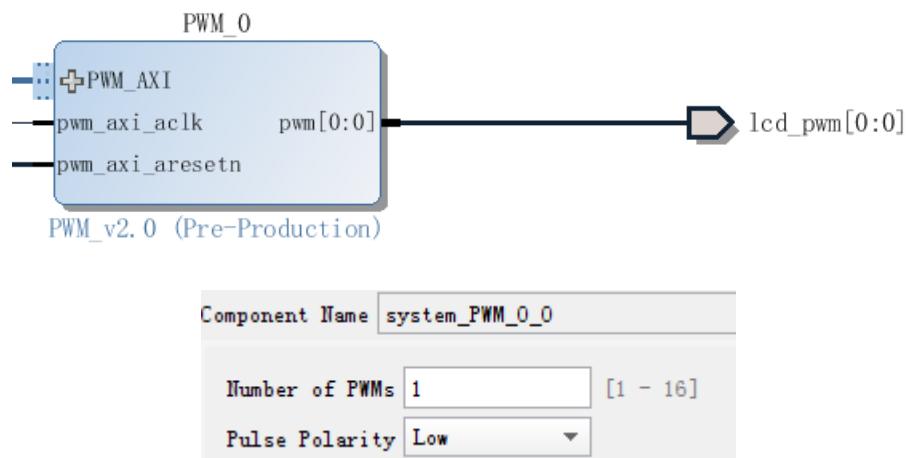


16.5.3 AXI PWM

本例程中，PL 部分通过使用 digilent 公司 AXI PWM 的 IP 核来实现 PWM 信号的输出，用于驱动触摸屏的背光源。该 ip 位于 Miz_ip_lib 文件夹下，IP 的路径设置如下图所示。



很多触摸屏的背光源都是由 PWM 信号驱动的，通过改变 PWM 信号的占空比可以调节触摸屏的亮度。AXI PWM 通过 AXI 总线与 PS 连接，PS 通过 AXI4-Lite 对其进行配置和控制，并可动态调节输出 PWM 信号的占空比。AXI PWM 的设置如下图所示，只需输出 1 路 PWM 信号，由于使用的触摸屏所需的 PWM 信号为负极性，即低电平占空比越大，屏幕越亮，因此将极性设为负。输出的 PWM 信号与触摸屏连接。

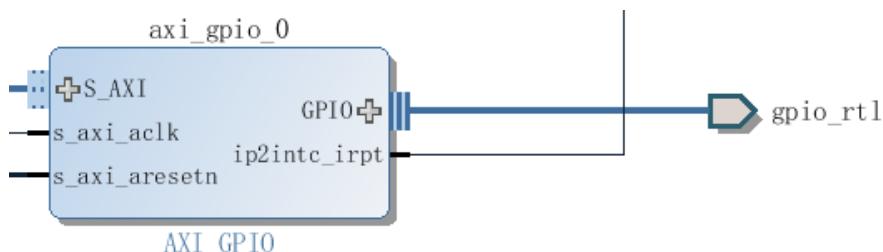


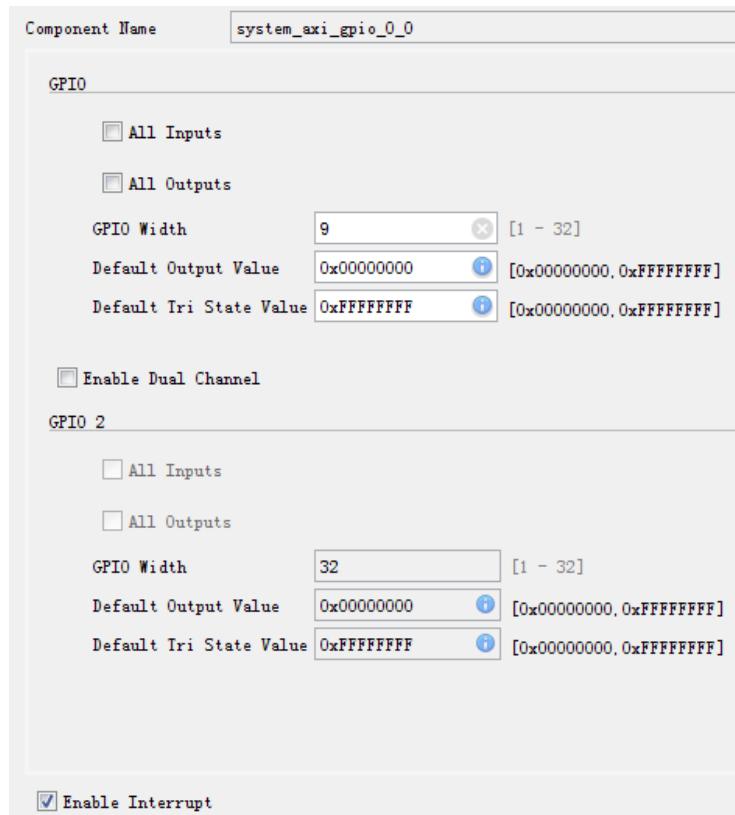
16.5.4 AXI GPIO

本例程中，PL 部分通过 AXI GPIO 实现 1 个触摸屏触摸中断信号输入，2 个按键信号输入，5 个 LED 控制信号输出，1 个液晶屏背光源使能信号输出，共计 9 个 GPIO 端口，定义如下。

- GPIO[0]，触摸屏中断信号输入
- GPIO[1]，按键信号输入，MZ7X 系列对应 SW1
- GPIO[2]，按键信号输入，MZ7X 系列对应 SW2
- GPIO[3]~GPIO[7]，LED1~LED5 控制信号输出，这里只使用 MZ7X 的 LD1~LD4，LD5 不作控制。
- GPIO[8]，液晶屏背光源控制信号输出

每当输入的触摸中断信号或按键信号发生一次改变，AXI GPIO 则会向 PS 触发一次中断。AXI GPIO 的设置如下图所示。使能中断接口，将其与 PS 连接。

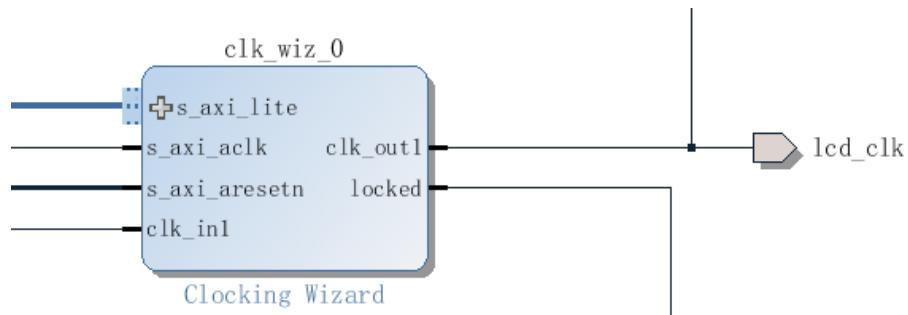


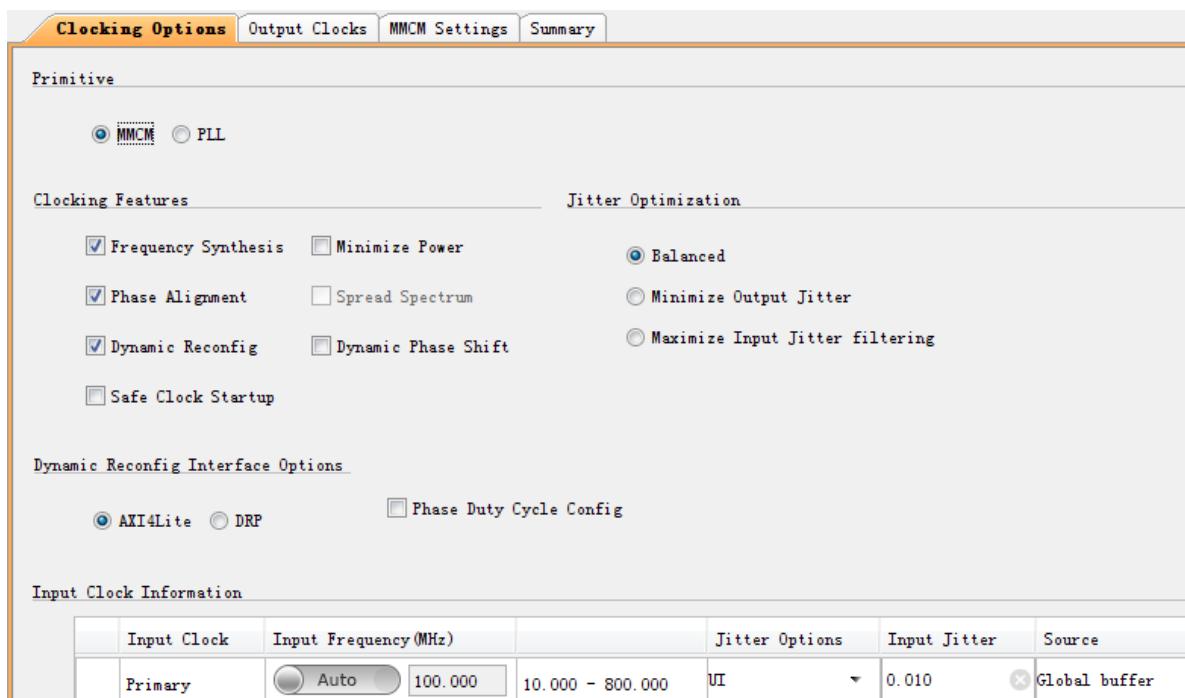


16.5.5 Clocking Wizard

本例程中，Clocking Wizard 用于提供 LCD 触摸屏进行 GUI 显示的参考时钟，该时钟被 Video Timing Controller 以及 AXI4-Sreamto Video Out 两个 IP 核使用，同时引出至顶层模块。将 Clocking Wizard 设置为 MMCM(因为 MMCM 内部倍频系数和 CLKOUT0 分频系数可以为小数，比 PLL 更易获得所需的频率)，并使能 AXI-lite 动态重配置接口，将动态重配置接口通过 AXI 总线与 PS 连接，方便 PS 随时对 MMCM 或 PLL 输出的时钟频率进行重配置，从而避免 PL 部分的重新编译。

Clocking Wizard 设置如下图所示。注意将 input clock 的时钟源设置为 Global buffer，因为输入时钟来自 PS 的 FCLK_CLK0，该时钟也被其他 IP 所使用，在进入 MMCM 之前已经位于全局时钟网络 BUFG。





输出时钟频率可设置也可不作设置，为了避免误解，在这里设置为所 LCD 显示屏需要的 51.2MHz。若用户使用其他触摸屏，需要更高的分辨率，建议将 clk_out1 设置一个大于等于最高分辨率所对应时钟频率的值，这样可以确保时序约束的可靠性。

The phase is calculated relative to the active input clock.					
Output Clock	Output Freq (MHz)		Phase	Request	Response
	Requested	Actual			
clk_out1	51.2	51.200	0.0		

16.5.6 IO 口

16.5.6.1 PL IO 信号定义

本例程除了 PS 部分的固定 IO 口之外，PL 部分引出的 IO 口如下图所示。

```

    input [1:0] button_i;
    input lcd_int_i;
    output lcd_clk_o;
    output [7:0]lcd_r_o;
    output [7:0]lcd_g_o;
    output [7:0]lcd_b_o;
    output lcd_de_o;
    output lcd_hs_o;
    output lcd_vs_o;
    inout lcd_iic_scl_io;
    inout lcd_iic_sda_io;
    output [0:0]lcd_pwm_o;
    output lcd_wake_n_o;
    output lcd_bl_en_o;
    output [4:0]led_o;

```

信号定义如下表所示。

PL 部分 IO 口定义表

端口名称	方向	说明
button_i[1:0]	Input	按键信号, mi702对应SW3, SW4; miz701n对应SW1, SW2
lcd_int_i	Input	触摸屏中断信号
lcd_clk_o	Output	液晶屏数据时钟信号
lcd_r_o[7:0]	Output	液晶屏像素点R分量
lcd_g_o[7:0]	Output	液晶屏像素点G分量
lcd_b_o[7:0]	Output	液晶屏像素点B分量
lcd_de_o	Output	液晶屏数据有效信号
lcd_hs_o	Output	液晶屏行同步信号
lcd_vs_o	Output	液晶屏场同步信号
lcd_I2C_scl_io	Inout	触摸屏I2C时钟信号
lcd_I2C_sda_io	Inout	触摸屏I2C数据信号
lcd_pwm_o	Output	液晶屏背光源PWM信号
lcd_wake_n_o	Output	触摸屏唤醒信号
lcd_bl_en_o	Output	液晶屏背光源使能信号
led_o[4:0]	Output	LED控制信号, mi702对应LD1~LD5, miz701n只使用LD1~LD4, led_o[5]脚并非与LD5连接, 而是连接到40 Pin连接器P1的T10脚

16.5.6.2 LCD 时钟信号输出

Clocking Wizard 引出的时钟信号 lcd_clk, 需要经过 ODDR 产生 1 个反相的时钟 lcd_clk_o 与 LCD 触摸屏连接。目的在于使输出时钟 lcd_clk_o 的边沿位于 LCD 其他控制、数据信号窗口的中心位置, 使建立和保持时间最大化。

```
ODDR #(  
    .DDR_CLK_EDGE("SAME_EDGE"), // "OPPOSITE_EDGE" or "SAME_EDGE"  
    .INIT(1'b0), // Initial value of Q: 1'b0 or 1'b1  
    .SRTYPE("SYNC") // Set/Reset type: "SYNC" or "ASYNC"  
) lcd_clk_addr (  
    .Q(lcd_clk_o), // 1-bit DDR output  
    .C(lcd_clk), // 1-bit clock input  
    .CE(1'b1), // 1-bit clock enable input  
    .D1(1'b0), // 1-bit data input (positive edge)  
    .D2(1'b1), // 1-bit data input (negative edge)  
    .R(1'b0), // 1-bit reset  
    .S(1'b0) // 1-bit set  
);
```

16.5.6.3 LCD RGB 信号映射

AXI4- Streamto Video Out 输出的 24 位像素点数据 lcd_data 与 LCD 触摸屏 RGB 信号的映射关系如下所示。

```
assign lcd_r_o = lcd_data[23:16];  
assign lcd_g_o = lcd_data[15:8];  
assign lcd_b_o = lcd_data[7:0];
```

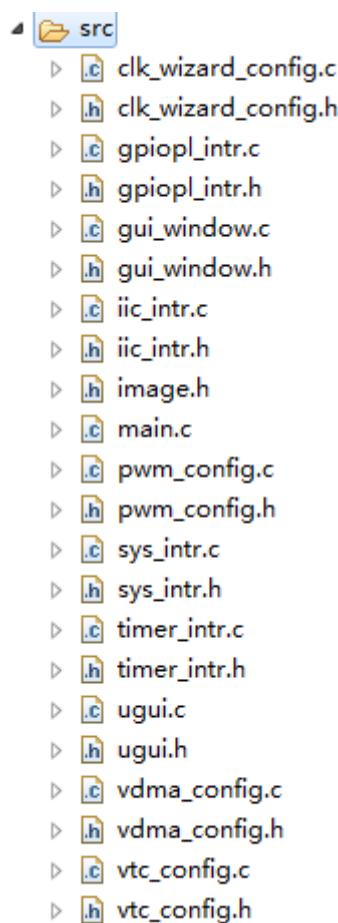
16.5.6.4 LCD 控制信号

LCD 液晶屏的背光源使能信号与 AXI GPIO 的最高位连接，使 PS 可以控制背光源的开关。触摸屏唤醒信号置为 1。

```
assign lcd_wake_n_o = 1'b1;  
assign lcd_bl_en_o = gpio_rtl_tri_o[8];
```

16.6 PS 程序设计

PS 部分程序所有的源文件如下图所示。



16.6.1 main 函数

main 函数主要完成如下功能：

- 初始化 Clocking Wizard，并重新设置其输出时钟
- 初始化中断控制器及系统中断
- 初始化并配置定时器及其中断
- 初始化并配置 AXI GPIO 及其中断
- 初始化并配置 I2C 接口
- 初始化并配置 Video Timing Controller，设置显示分辨率
- 初始化并配置 AXI VDMA 及其中断
- 初始化并配置 AXI PWM，设置输出 PWM 信号频率及占空比
- 启动定时器工作
- 创建 GUI 界面
- 启动 AXI VDMA 工作
- 打开背光源，启动 AXI PWM 输出 PWM 信号点亮触摸屏
- 进入空循环，等待触摸屏中断，根据触摸坐标信息进行相应响应，GUI 界面产生相应变化

16.6.2 时钟重配置

在 PL 中，Clocking Wizard 使能了动态重配置功能，因此 PS 可以通过其 AXI-lite 接口动态重配置 MMCM 或者 PLL，来改变其时钟的输出频率。

16.6.2.1 SDK 库移植

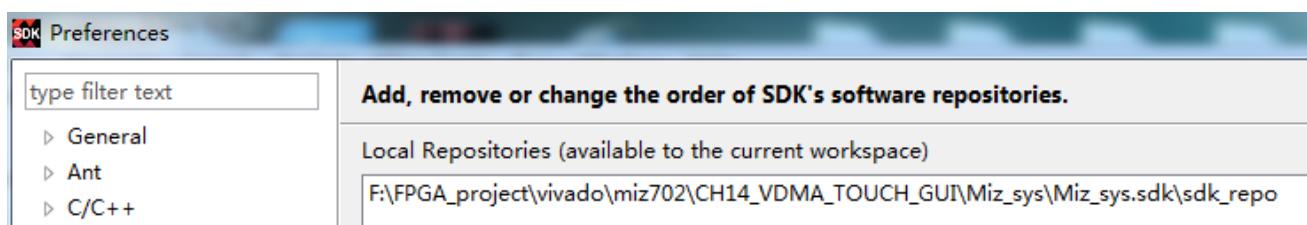
由于在 vivado 2016.4 的 SDK 中还没有 Clocking Wizard 的驱动函数库，缺少可以利用的 API 函数。而 vivado 2016.4 的 SDK 中包含了 Clocking Wizard 的驱动库。因此，为了便于开发，在 2016.4 中借用 2016.4 的驱动库进行设计。

首先，在 SDK 2016.4 的安装目录（如：

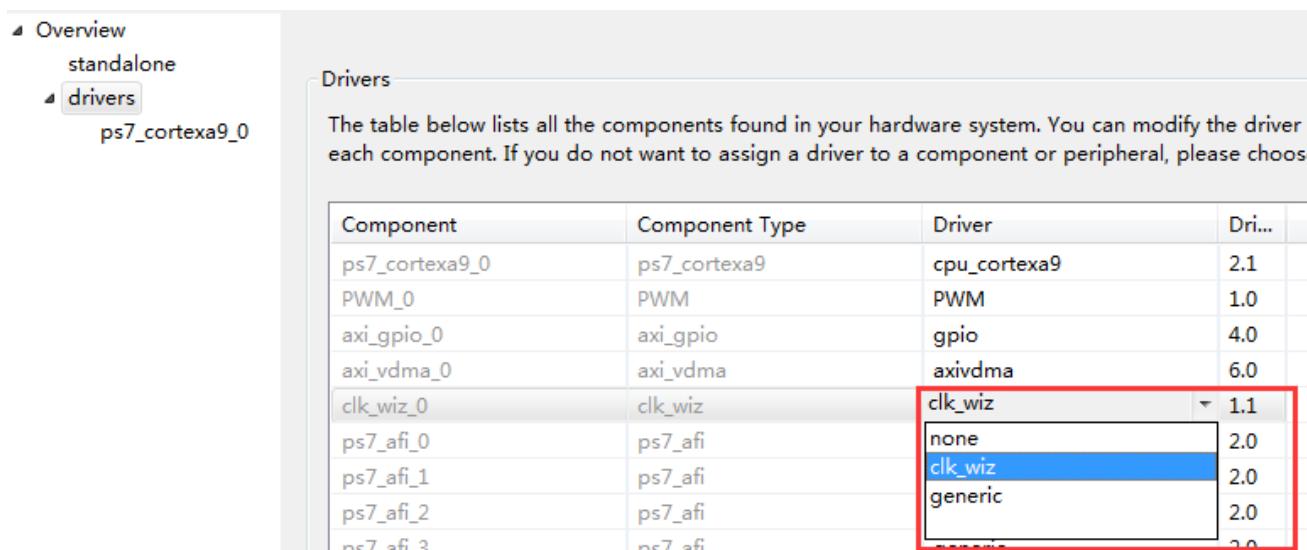
C:\Xilinx\SDK\2017.4\data\embeddedsw\XilinxProcessorIPLib\drivers）下找到 Clocking Wizard 的驱动库文件夹 clk_wiz_v1_1，如下图所示。



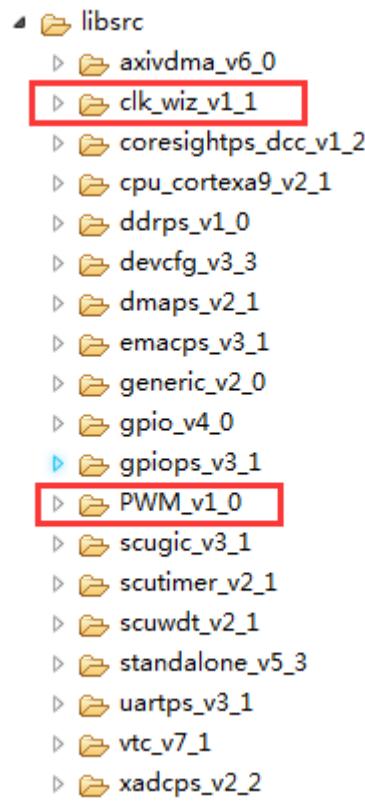
将文件夹 clk_wiz_v1_1 复制到工程目录下的 sdk_repo 的 bsp 文件夹中。然后在 sdk 中设置 sdk_repo 文件夹的路径，如下图所示。



然后更改工程 bsp 中的驱动函数设置，将 clk_wiz_0 的驱动改为 clk_wiz 1.1 版本，如下图所示。



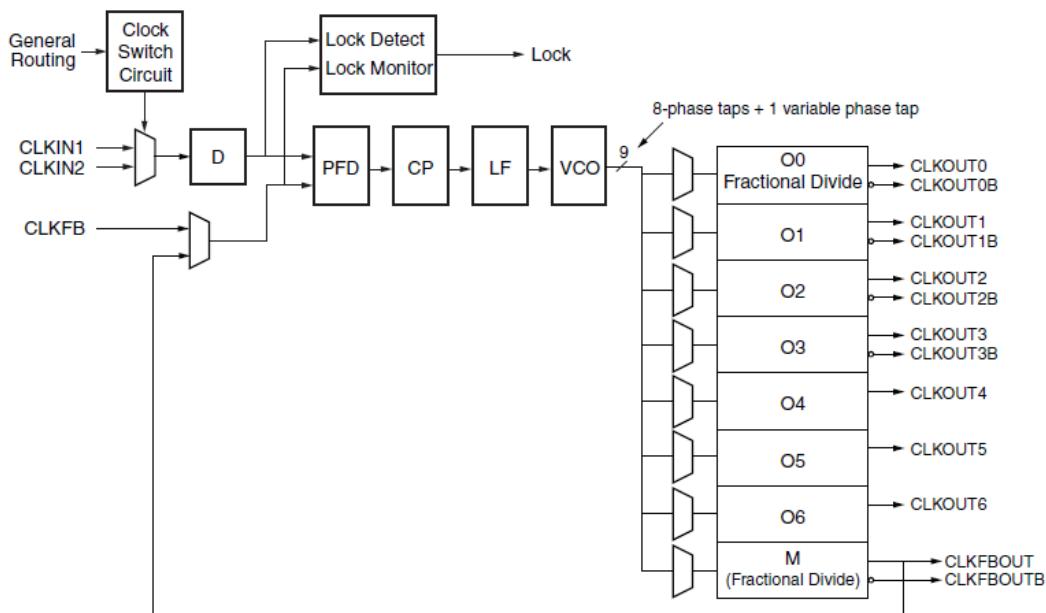
重新生成 bsp 后，在 bsp 的 libsrc 下就会出现 clk_wiz_v1_1 的驱动，如下图所示。



16.6.2.2 驱动程序

Clocking Wizard 的驱动程序由 clk_wizard_config.c 和 clk_wizard_config.h 组成。

本例程中，使用了 MMCM，MMCM 的重配置通过 Clk_Wiz_Reconfig() 函数完成，该函数参考自 SDK 中 clk_wiz_v1_1 的 example: xclk_wiz_intr_example.c。由于 SDK 关于 Clocking Wizard 的驱动还未完善，且 MMCM/PLL 内部的寄存器信息未有官方文档可参考，所以 Clk_Wiz_Reconfig() 不在此做具体分析。MMCM 的基本结构如下图所示。



MMCM 内部的压控振荡器 VCO 频率与 MMCM 的输出时钟频率的公式如下：

$$F_{VCO} = F_{CLKIN} \times \frac{M}{D}$$

$$F_{OUT} = F_{CLKIN} \times \frac{M}{D \times O}$$

ZYNQ 7010/7015/7020 内部 MMCM 的 VCO 频率范围为如下图所示。

Symbol	Description	Speed Grade				Units
		-3	-2	-1C/-1L/-1LI	-1Q	
MMCM_F_VCOMIN	Minimum MMCM VCO frequency	600.00	600.00	600.00	600.00	MHz
MMCM_F_VCOMAX	Maximum MMCM VCO frequency	1600.00	1440.00	1200.00	1200.00	MHz

动态配置 MMCM 其实就是通过 PS 改变其内部倍频系数 M，输入分频系数 D，以及输出分频系数 O 的值来实现。在 clk_wizard_config.h 有 4 个重要的宏定义，如下：

```
#define VCO_FREQ          600
#define DYNAMIC_INPUT_FREQ 100
#define DYNAMIC_OUTPUT_FREQ 51.2
#define CLK_FRAC_EN         1
```

其中 VCO_FREQ 表示 MMCM 工作时的 VCO 频率为 600MHz，DYNAMIC_INPUT_FREQ 表示 MMCM 输入的时钟频率为 100MHz，DYNAMIC_OUTPUT_FREQ 表示 MMCM 输出的时钟频率为 51.2MHz，CLK_FRAC_EN 表示允许 CLKOUT0 的分频系数为 1/8 精度的小数。

Clk_Wiz_Reconfig() 函数根据这 4 个宏定义的值对 MMCM 的 M、D、O 的值进行配置，并通过 Wait_For_Lock() 判断各环节的输出时钟是否 LOCK。用户也可以根据实际需求在允许的范围内调节 VCO 的频率。

16.6.3 PWM 信号输出

16.6.3.1 SDK 库设置

digilent AXI PWM IP 核对应的 PS 驱动库 PWM_v1_0 也位于工程目录 sdk_repo 的 bsp 文件夹中。在 bsp 的 libsrc 下可以看到 PWM_v1_0 驱动库，如 6.2.1 节图所示。

16.6.3.2 驱动程序

AXI PWM 的驱动程序由 pwm_config.c 和 pwm_config.h 组成。

本例程所使用的液晶屏建议输入 PWM 信号频率为 100Hz~200K Hz。在 main 函数中通过 PWM_Init()函数设置初始输出 PWM 信号的周期和占空比。

PWM_Init 函数：

- 通过 PWM_Set_Period 函数设置 PWM 信号周期
- 通过 PWM_Set_Duty 函数设置 PWM 信号低电平的占空比

上述函数都是以时钟周期数来设置。在 pwm_config.h 包含了 PWM 信号周期和占空比的时钟周期数宏定义。如下：

```
#define PERIOD_CLOCK_NUM      409600
#define DUTY_CLOCK_NUM         204800
```

由于本例程中，PL 部分 AXI PWM 的参考时钟频率为 100MHz，一个时钟周期就是 10ns。因此，PWM_Init 所设置的 PWM 信号的周期为 4.096ms，占空比为 50%，频率约为 250Hz。

为了对 PWM 信号的占空比进行动态调节达到改变液晶屏背光源亮度的目的，又设计了 PWM_increase_duty()和 PWM_decrease_duty()函数用于增大和减小 PWM 信号中低电平的占空比。

最后，在 main 函数中调用 PWM_Enable()使能 PWM 信号输出。

16.6.4 GPIO 输入输出

AXI GPIO 的驱动程序由 gpiopl_intr.c 和 gpiopl_intr.h 组成。

在 main 函数调用 Gpiopl_init()函数初始化 AXI GPIO，设置 9 个 GPIO 的方向，其中 GPIO[0]~GPIO[2]为输入，GPIO[3]~GPIO[8]为输出。并将 GPIO[3]~GPIO[8]的输出都置为 0。每个 GPIO 的宏定义在 gpiopl_intr.h 中，如下所示。

```
#define TOUCH_INTR_MASK      0x00000001
#define BUTTON0_INTR_MASK     0x00000002
#define BUTTON1_INTR_MASK     0x00000004
#define LED1_MASK             0x00000008
#define LED2_MASK             0x00000010
#define LED3_MASK             0x00000020
#define LED4_MASK             0x00000040
```

```
#define LED5_MASK          0x00000080
#define LCD_BL_EN_MASK       0x00000100
```

通过 Gpiopl_Setup_Intr_System() 初始化并使能 AXI GPIO 的输入中断，GPIO[0]~GPIO[2] 的任意 1 个信号的输入发生 1 次改变将触发 1 次中断，GpioplIntrHandler()为 GPIO 的中断函数。

GpioplIntrHandler()函数：

- 判断 GPIO[0]输入的触摸屏中断信号是否为 0，若为 0，则调用 I2C_write()和 I2C_read()函数通过 I2C 接口读出触摸屏的触摸信息，并触摸屏中断标志位 touch_flag 置 1，该信号将在定时器中断中使用。
- 判断 GPIO[1]输入的 SW3 按键信号是否为 0，若为 0，将 GPIO[8]信号拉低，打开液晶屏背光源。
- 判断 GPIO[2]输入的 SW4 按键信号是否为 0，若为 0，将 GPIO[8]信号拉高，关闭液晶屏背光源。

最后，在 main 函数中将 GPIO[8]置为 1，开启液晶屏的背光源。

GPIO[3]~GPIO[7]信号的输出在 gui_window.c 中进行控制，用于在 GUI 界面中控制开发包中的 LED 灯。

16.6.5 I2C 读取触摸信息

PS 的 I2C 接口的驱动程序由 iic_intr.c 和 iic_intr.h 组成。

在本例程中，PS 的 I2C 接口不能工作于中断模式，其原因在于，通过 I2C 接口读写触摸屏是在 GPIO 的中断函数 GpioplIntrHandler()中执行，此时若 I2C 接口产生中断，该中断将不能在 GPIO 中断中被嵌套响应，无法完成读写操作。因此，I2C 接口只能工作于轮询 Poll 模式。

触摸屏中 FT5206 控制芯片的 I2C 地址和 I2C 时钟频率设置如 iic_intr.h 宏定义所示。I2C 地址为 0x38，I2C 时钟频率设为 400KHz，为 FT5206 的最高值。

```
#define IIC_SLAVE_ADDR    0x38
#define IIC_SCLK_RATE      400000
```

在 main 函数中调用 Iic_init()函数初始化 PS 的 I2C 接口，设置 I2C 时钟频率。当触摸屏中断信号拉低触发 GPIO 产生中断后，在 GPIO 中断函数 GpioplIntrHandler()中首先调用 I2C_write()向 FT5206 芯片发送需要读取的寄存器首地址。在本例程中，需要从地址为 0x02 的寄存器开始读，因此 I2C_write()发送的首地址为 0x02。然后，调用 I2C_read()函数从 FT5206 连续读取 29 个寄存器的值，每个寄存器的值为 1 个字节，一共读取 29 个字节的触摸信息。

I2C_write()和 I2C_read()函数均以轮询模式控制 I2C 接口。

16.6.6 GUI 界面显示

GUI 界面显示在 PL 部分由 AXI VDMA，Video Timing Controller，AXI4- Sreamto Video Out 三个 IP 协同完成。其中 AXI VDMA 和 Video Timing Controller 需要 PS 进行设置，AXI VDMA 完成 PS 端 DDR3 中 GUI 界面的读取，Video Timing Controller 产生通过 AXI4- Sreamto Video Out 进行 GUI 界面显示的控制时序。

16.6.6.1 AXI VDMA

AXI VDMA 的驱动程序由 vdma_config.c 和 vdma_config.h 组成。

在本例程中，AXI VDMA 仅有 MM2S 方向（PS DDR 到 PL）的读通道工作。另外，需要让 AXI VDMA 工作于 free run 模式，因此，只使能错误中断。

在 main 函数中调用 Vdma_Init() 函数对 AXI VDMA 进行初始化，在 Vdma_Init() 中调用 ReadSetup() 对 MM2S 读通道的参数进行设置。VDMA 读通道的相关设置由 vdma_config.h 中的宏定义所决定，如下所示。

```
#define IMAGE_WIDTH      GUI_WIDTH
#define IMAGE_HEIGHT     GUI_HEIGHT
#define BYTES_PER_PIXEL  4
#define NUMBER_OF_READ_FRAMES 1
#define MEM_BASE_ADDR    0x10000000
#define BUFFER0_BASE     (MEM_BASE_ADDR)
```

- IMAGE_WIDTH，图像宽度，1024
- IMAGE_HEIGHT，图像高度，600
- BYTES_PER_PIXEL，每个像素点的字节数为 4
- NUMBER_OF_READ_FRAMES，图像缓存数量，实际只需使用 1 个缓存
- BUFFER0_BASE，图像缓存的首地址，0x10000000

然后，在 main 函数中调用 Vdma_Setup_Intr_System() 函数，配置使能 VDMA 读通道的错误中断，ReadErrorCallBack() 为中断函数。

最后，调用 Vdma_Start() 函数，启动 AXI VDMA 的读通道开始工作。

16.6.6.2 显示时序设置

Video Timing Controller 的驱动程序由 vtc_config.c 和 vtc_config.h 组成。

在本例程中，液晶屏使用了 $1024 \times 600 @ 60Hz$ 显示分辨率，使用的时序参数如下：

<ul style="list-style-type: none"> ● 行总像素数：1344 ● 行有效像素数：1024 ● 行同步前肩像素数：24 ● 行同步信号像素数：136 ● 行同步后肩像素数：160 	<ul style="list-style-type: none"> ● 场总行数：635 ● 场有效行数：600 ● 场同步前肩行数：8 ● 场同步信号行数：4 ● 场同步后肩行数：23
--	---

在 main 函数中调用 Vtc_init() 函数对 Video Timing Controller 进行初始化，将上述的显示时序参数写入其中，然后使能其产生控制时序信号。

16.6.7 定时器

PS 定时器的驱动程序由 timer_intr.c 和 timer_intr.h 组成。

在本例程中，PS 定时器用于周期性产生中断来实现 GUI 界面的刷新，以固定的频率的调用 UG_Update() 函数实现 GUI 的动态特性。

首先，在 main 函数中调用 Timer_init() 函数对定时器进行初始化，其中断周期由以下宏定义决定，周期为 20ms，即 GUI 的刷新频率为 50Hz，刷新频率越高，GUI 的动

态变化越快，用户可以根据需求进行调整。

```
#define TIMER_LOAD_VALUE XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ / 100
```

然后，在 main 函数中调用 Timer_Setup_Intr_System()函数使能定时器的中，将中断函数设置为 TimerIntrHandler()。

TimerIntrHandler()函数的过程如下：

- 关闭定时器中断，因为定时器中断函数的执行时间可能会超过定时器设置的中断周期，如 20ms。
- 若触摸屏中断标志位 touch_flag 为 1，则对 GPIO 中断函数中读出的 29 字节触摸信息进行判断，若第一个触摸点状态为接触（contact），则读出并计算出第一个触摸点的 x,y 坐标信息，通过 UG_TouchUpdate()函数将该触摸坐标反馈至 GUI 界面，表示 GUI 上的该点被触摸。若第一个触摸点状态为抬起（put up），则将无效坐标（-1, -1）通过 UG_TouchUpdate()反馈至 GUI 界面，表示 GUI 上的无任何点被触摸。目前 μGUI 库的按键触摸功能只支持单点触摸，因此无法同时设计出两个按键同时被触摸的效果。
- 若当前的窗口为 window 4，则为 5 点绘图窗口。此时，读出 5 个触摸点的坐标信息，以每个触摸点为圆心，沿每个手指移动轨迹不断画圆。5 个触摸点的圆形图案颜色按顺序依次为：红、黄、蓝、绿、橙。
- 调用 UG_Update()刷新 GUI 界面，让触摸使 GUI 产生的动态变化更新至对应的图像中。
- 将变化后最新的 GUI 界面对应的图像通过 Xil_DCacheFlushRange()函数刷进 DDR 中，从而保证 VDMA 将最新的 GUI 界面读出，并显示在触摸屏上。
- 重新使能定时器中断。

最后，在 main 函数里调用 Timer_start()函数启动定时器工作。

16.6.8 GUI 界面设计

GUI 界面的驱动程序由 gui_window.c、gui_window.h、image.h、ugui.c 和 ugui.h 组成，其中 ugui.c 和 ugui.h 来自 μ GUI 库。

在本例程中，一共设计了 5 个窗口，对应 window1~window5。同时，也为每个窗口设计了对应的回调函数 window_1_callback()~window_5_callback()。

在 main 函数中，调用 gui_create()函数对 GUI 进行初始化，并创建所有的窗口及窗口所包含的所有对象。

gui_create()函数的流程如下：

- 调用 UG_Init()函数初始化 GUI
- 调用 UG_FillScreen()函数设置所有 GUI 窗口的背景颜色
- 调用 create_window1()~create_window5()函数依次创建 5 个窗口
- 调用 UG_WindowShow()函数设置 GUI 显示的第一个窗口为 window1
- 调用 UG_WaitForUpdate()函数等待定时器中断到来，刷新并显示 GUI 界面

GUI 界面设计以窗口为基础进行，每个界面对应一个窗口，但所有的窗口都对应同一片内存区域。μGUI 库中有很多 API 函数，这里只介绍本例程所涉及的 API 函数，其余的 API 可参考 μGUI 使用手册。5 个窗口的设计在 create_window1()~create_window5()函数中完成。

16.6.8.1 GUI 初始化

通过 UG_Init()函数设置 GUI 界面的长宽分辨率，并绑定在 4.1.1 节所述用户自定义的像素值设置函数 PixelSet()。通过 UG_FillScreen()函数将所有 GUI 窗口的背景设置为浅灰色。

16.6.8.2 窗口 1 设计

窗口 1 为 GUI 的欢迎界面。如下图所示。



窗口 1 的设计在 create_window1()函数中完成，流程如下：

1) 窗口创建

- 调用 UG_WindowCreate()函数绑定窗口 1 的回调函数为 window_1_callback(), 设置窗口可包含对象的最大个数为 15。
- 调用 UG_WindowSetTitleText()函数设置窗口 1 的标题内容。
- 调用 UG_WindowSetTitleTextFont()函数设置标题字体的大小。

2) 按键创建

- 调用 UG_ButtonCreate()函数创建按键 0，并设置按键 0 的坐标范围
- 调用 UG_ButtonSetStyle()函数设置按键 0 的模式为 3D，且按下后背景和字体的颜色会跳变
- 调用 UG_ButtonSetAlternateForeColor 函数设置按键 0 按下后改变的字体颜色
- 调用 UG_ButtonSetAlternateBackColor 函数设置按键 0 按下后改变的按键背景颜色
- 调用 UG_ButtonSetFont 函数设置按键 0 的字体大小
- 调用 UG_ButtonSetText 函数设置按键 0 显示的内容

3) 文本框创建

- 文本框 0
 - 调用 UG_TextboxCreate() 函数创建文本框 0，并设置文本框 0 的坐标范围。
 - 调用 UG_TextboxSetFont() 函数设置文本框 0 内容的字体大小。
 - 调用 UG_TextboxSetText() 函数设置文本框 0 的内容。
 - 调用 UG_TextboxSetAlignment() 函数设置文本框 0 中的内容在文本框中的对齐方式。
- 文本框 1
 - 调用 UG_TextboxSetFontColor() 函数设置文本框 1 内容的字体颜色
 - 调用 UG_TextboxSetHSpace() 函数设置文本框 1 中每个字符之间的横向距离
 - 其余函数调用与文本框 0 同理

4) 图片创建

目前，最新版的μ GUI v0.3 仅支持在 GUI 界面中添加 RGB565 格式的 BMP 图像。由于本设计中所需添加的两个米联 logo 均为 RGB888 格式，为了让 μ GUI 能支持 24 位 RGB888 的图片，需要对 ugui.c 文件中的 UG_DrawBMP() 函数进行修改，如下所示。其中通过阴影部分为添加的代码。

```
voidUG_DrawBMP(UG_S16xp, UG_S16yp, UG_BMP* bmp )
{
    UG_S16x,y,xs;
    UG_U8r,g,b;
    UG_U16* p;
    /*add by osrc 2017.2.18*/
    UG_U32* p1;
    UG_U16tmp;
    /*add by osrc 2017.2.18*/
    UG_U32 tmp1;
    UG_COLOR c;

    if ( bmp->p == NULL ) return;

    /* Only support 16 bpp so far */
    if ( bmp->bpp == BMP_BPP_16 )
    {
        p = (UG_U16*)bmp->p;
    }
    /*add support for 32 bpp, by osrc 2017.2.18*/
    elseif(bmp->bpp == BMP_BPP_32)
    {
        p1 = (UG_U32*)bmp->p;
    }
}
```

```

else
{
    return;
}

xs = xp;
for(y=0;y<bmp->height;y++)
{
    xp = xs;
    for(x=0;x<bmp->width;x++)
    {
        /*add support for RGB888, by osrc 2017.2.18*/
        if(bmp->colors == BMP_RGB565)
        {
            tmp = *p++;
/* Convert RGB565 to RGB888 */
            r = (tmp>>11)&0x1F;
            r<<=3;
            g = (tmp>>5)&0x3F;
            g<<=2;
            b = (tmp)&0x1F;
            b<<=3;
            c = ((UG_COLOR)r<<16) | ((UG_COLOR)g<<8) | (UG_COLOR)b;
        }
        else
        {
            tmp1 = *p1++;
            c = tmp1;
        }
        UG_DrawPixel(xp++ , yp , c );
    }
    yp++;
}
}
➤ 图片 0
添加的图片 0 如下所示，为米联的 logo。图片各像素点的值包含在 image.h 的 logo1_bmp 数组中。

```



定义该图片在 GUI 中的结构体，如下所示。图片的指针 logo1_bmp，为图片的长、宽均为 65 个像素，每个像素点占 32bit，图片格式为 RGB888。

```
constUG_BMP logo1 =
{
    (void*)logo1_bmp,
    65,
    65,
    BMP_BPP_32,
    BMP_RGB888
};
```

- 调用 UG_ImageCreate 函数在窗口 1 中创建该图片对象，并设置图片的坐标位置。
- 调用 UG_ImageSetBMP 函数将上述的 logo 图像与创建的图片对象绑定。

➤ 图片 1

添加的图片 1 如下所示，为米联客的 logo。图片各像素点的值包含在 image.h 的 logo2_bmp 数组中。



定义该图片在 GUI 中的结构体，如下所示。图片的指针 logo2_bmp，为图片的长位 259 像素，宽为 105 个像素，每个像素点占 32bit，图片格式为 RGB888。

```
constUG_BMP logo2 =
{
    (void*)logo2_bmp,
    259,
    105,
    BMP_BPP_32,
    BMP_RGB888
};
```

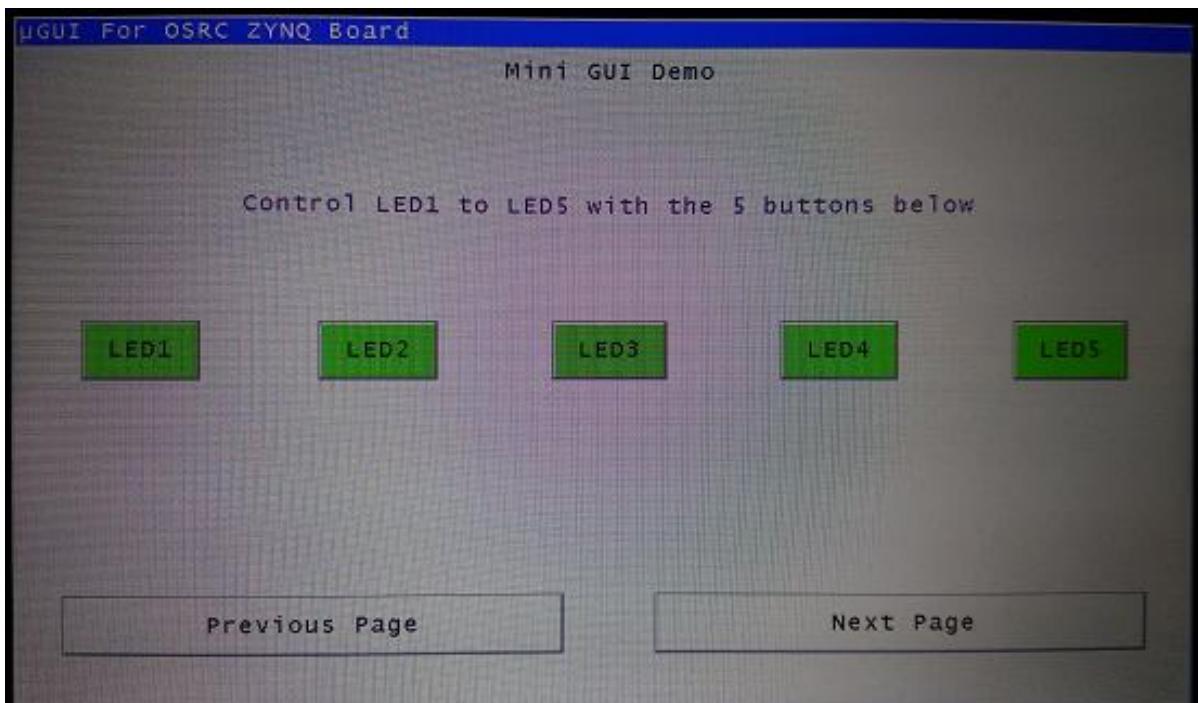
图片对象创建添加方式与图片 0 相同。

5) 回调函数

1 个窗口的回调函数当该窗口中的按键的触摸状态发生改变时，会在 UG_Update() 函数中被调用。在窗口 1 的回调函数 window_1_callback() 中设置了按键 0 的功能，当在触摸屏中按下 start application now 按键便会使 window_1_callback() 被调用，通过 UG_WindowShow 函数，让 GUI 界面刷新后切换到窗口 2。在按下按键的同时，可以观察到按键 0 字体和背景颜色产生的变化。

16.6.8.3 窗口 2 设计

窗口 2 为 LED 灯控制界面，如下图所示。在该窗口中，用户可以通过触摸 LED1~LED5 按键来控制开发板上的标识为 LD1~LD5 的 LED 灯。



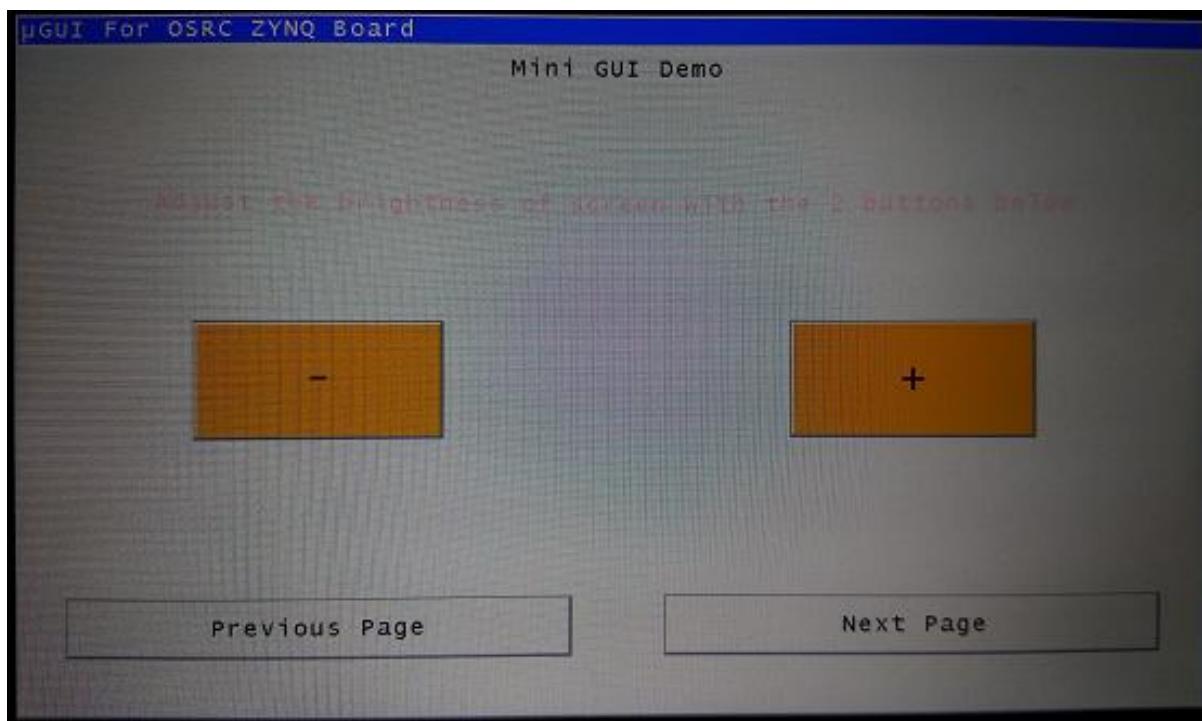
窗口 2 的设计在 `create_window2()` 函数中完成，窗口 2 中的标题、按键、文本框的创建流程与 `create_window1()` 原理相同，详细可参阅工程代码，此处不作赘述。

1) 回调函数

窗口 2 的回调函数 `window_2_callback()` 中设置了 7 个按键 0~6 的功能，其中 LED1~LED5 对应 5 个 LED 灯的控制功能，通过控制 GPIO[3]~GPIO[7]的输出来实现 LED 灯的开关。当 LED 灯亮时，对应按键的背景颜色为红色，当 LED 熄灭时，对应按键的背景颜色还原为绿色。（注意 miz701n 只有 LED1~LED4 受控制，按键 LED5 实际无控制 LD5 的作用）另外，按下 Previous Page 按键让 GUI 界面切换至窗口 1，按下 Next Page 按键让 GUI 界面切换至窗口 3。在按下按键的同时，可以观察到各按键字体和背景颜色产生的变化。

16.6.8.4 窗口 3 设计

窗口 3 为液晶屏亮度调节界面，如下图所示。在该窗口中，用户可以通过触摸加、减按键来调节液晶屏背光源的亮度。



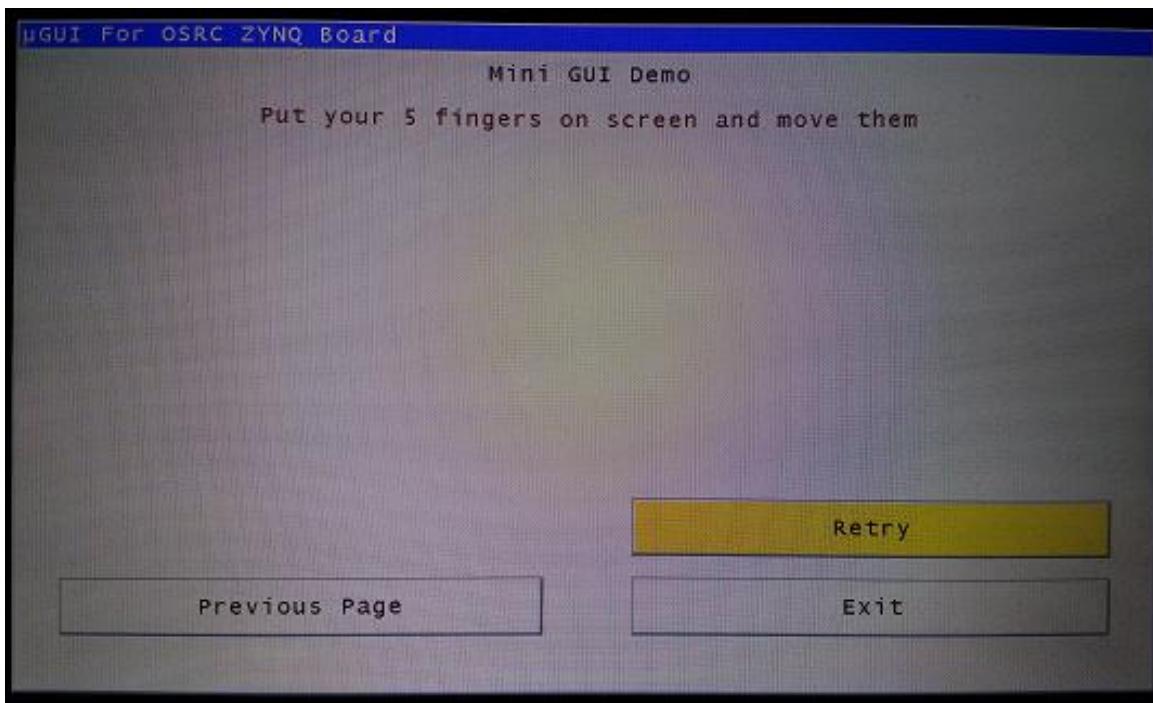
窗口 3 的设计在 `create_window3()` 函数中完成，窗口 3 中的标题、按键、文本框的创建流程与 `create_window1()` 原理相同，详细可参阅工程代码，此处不作赘述。

1) 回调函数

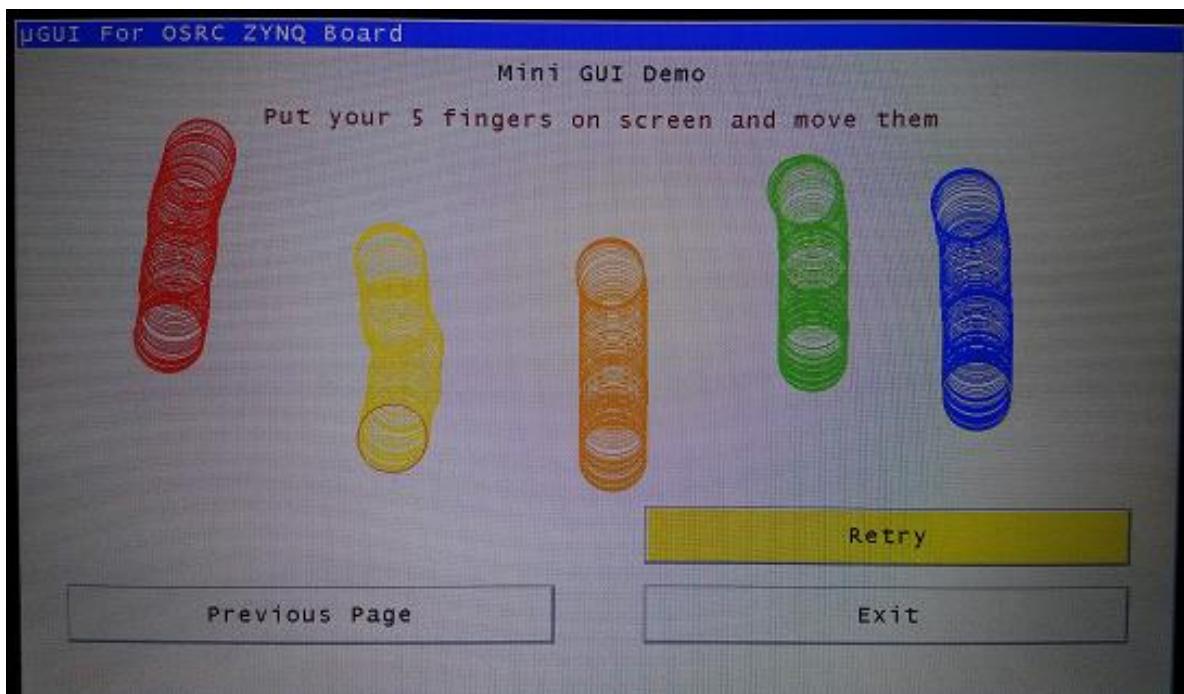
窗口 3 的回调函数 `window_3_callback()` 中设置了 4 个按键 0~3 的功能，其中“-”按键控制 PWM 信号低电平的占空比减小，从而降低液晶屏亮度；“+”按键控制 PWM 信号低电平的占空比增大，从而提高液晶屏亮度。另外，按下“Previous Page”按键让 GUI 界面切换至窗口 2，按下“Next Page”按键让 GUI 界面切换至窗口 4。在按下按键的同时，可以观察到各按键字体和背景颜色产生的变化。

16.6.8.5 窗口 4 设计

窗口 4 为绘图界面，如下图所示。在该窗口中，可实现 5 点触控，用户可以通过 5 个手指同时触摸平面来进行简单画图。



绘图时，以每个手指触摸点为圆心，沿每个手指移动轨迹不断画圆。5个触摸点的圆形图案颜色按顺序依次为：红、黄、蓝、绿、橙。由于 GUI 界面存在一定刷新的时间间隔，因此绘图轨迹中间会出现间断的现象，如下图所示。这里用户自行改进来实现真实的绘图效果。



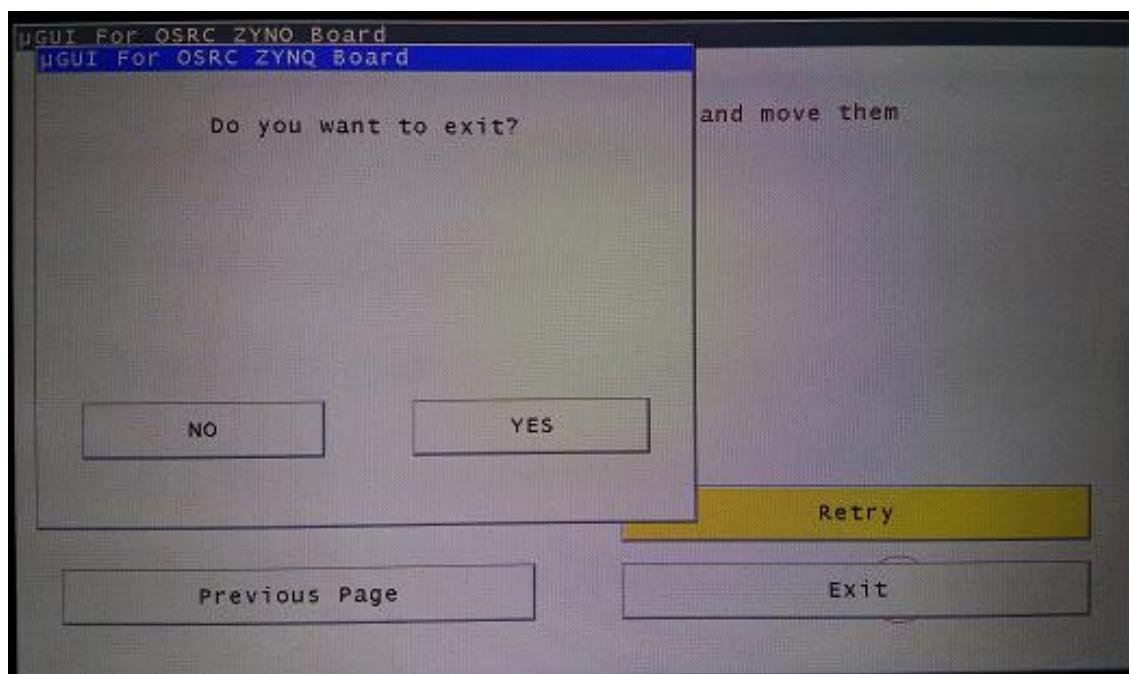
窗口 4 的设计在 `create_window4()` 函数中完成，窗口 4 中的标题、按键、文本框的创建流程与 `create_window1()` 原理相同，详细可参阅工程代码，此处不作赘述。

1) 回调函数

窗口 4 的回调函数 window_4_callback()中设置了 3 个按键 0~2 的功能，其中“Retry”按键用于清除当前窗口中的绘图结果，使用户可以重新进行绘图。另外，按下“Previous Page”按键让 GUI 界面切换至窗口 3，按下“Next Page”按键让 GUI 界面切换至窗口 5。在按下按键的同时，可以观察到各按键字体和背景颜色产生的变化。

16.6.8.6 窗口 5 设计

窗口 5 为退出界面，如下图所示。在该窗口中，用户可以通过触摸“YES”、“NO”按键来选择是否退出回到窗口 1 的欢迎界面。由于窗口 5 的尺寸小于其他窗口，因此当 GUI 界面从窗口 4 切换到窗口 5 时，可以出两个窗口叠加的效果，并且当窗口 5 出现时，窗口 4 的标题栏会由蓝变灰。



窗口 5 的设计在 create_window5()函数中完成，由于窗口 5 的尺寸小于 1024×600 ，因此需要调用 UG_WindowResize()函数重新设置窗口 5 的尺寸大小及位置。窗口 5 中的标题、按键、文本框的创建流程与 create_window1()原理相同，详细可参阅工程代码，此处不作赘述。

2) 回调函数

窗口 5 的回调函数 window_5_callback()中设置了 2 个按键 0~1 的功能，按下“YES”按键让 GUI 界面切换至窗口 1，按下“NO”按键让 GUI 界面切换至窗口 4。在按下按键的同时，可以观察到各按键字体和背景颜色产生的变化。

16.7 注意事项

16.7.1 更改 GUI 分辨率

若用户要接其他分辨率的触摸屏，需要更改触摸屏显示分辨率时，需要更改 3 个地方。

- main 函数中 Vtc_init() 函数中的分辨率参数
- 更改 gui_window.h 中的 2 个宏定义

```
#define GUI_HEIGHT 600
#define GUI_WIDTH 1024
```
- 更改 clk_wizard_config.h 中的 1 个宏定义，将其改为所需要的时钟频率，单位为 MHz

```
#define DYNAMIC_OUTPUT_FREQ 51.2
```

16.7.2 SDK 路径设置

注意 6.2.1 节关于 clk_wiz_v1_1 与 PWM_v1_0 库路径设置的描述，用户将原版程序在自己的电脑上运行时，务必要重新设置该路径，否则 SDK 将提示错误。

一个完美的结束
意味着一个新的开始！

www.osrc.cn

米联客
技术论坛
秀出你的风采！