

# **ZYNQ MPSOC Development Platform**

## **Vitis Application Tutorial**



## Version Record

Version	Date	Release By	Description
Rev1.03	2022-04-30	Rachel Zhou	First Release

The ALINX official Documents are in Chinese, and the English version was translated by **Shanghai Tianhui Trading Company**. They has **not been officially Review by ALINX** and are for reference only. If there are any errors, please send email feedback to [support@aithtech.com](mailto:support@aithtech.com) for correction.

**Amazon Store:** <https://www.amazon.com/alinx>

**Aliexpress Store:**

<https://alinxfpga.aliexpress.com/store/911112202?spm=a2g0o.detail.1000007.1.704e2bedqLBW90>

**Ebay Store:** <https://www.ebay.com/str/alinxfpga>

### Customer Service Information

Skype: [rachelhust@163.com](mailto:rachelhust@163.com)

Wechat: [rachelhust](#)

Email: [support@aithtech.com](mailto:support@aithtech.com) and [rachehust@163.com](mailto:rachehust@163.com)

# Content

Version Record .....	2
Content .....	3
Preparation and precautions .....	15
Software Environment .....	15
Hardware environment .....	15
Batch Download QSPI Flash .....	15
Batch process to build Vitis project .....	16
PS side Peripherals Parts .....	19
Part 1: Experience ARM, bare metal output "Hello World" .....	20
Part 1.1: Hardware Introduction .....	20
FPGA Engineer Job Content .....	21
Part 1.2: Create a Vivado Project .....	21
Part 1.2.1: Low Speed Configuration .....	23
Part 1.2.2: High Speed Configuration .....	26
Part 1.2.3: Clock Configuration .....	28
Part 1.2.4: DDR Configuration .....	30
Software Engineer Job Content .....	37
Part 1.3: Vitis Debugging .....	37
Part 1.3.1: Create Application Project .....	37
Part 1.4: Curing Program .....	51
Part 1.4.1: Generate FSBL .....	52
Part 1.4.2: SD card Startup Test .....	57
Part 1.4.3: QSPI Startup Test .....	59
Part 1.4.4: Programming QSPI Under Vivado .....	61
Part 1.5: Q&A .....	64
Part 1.5.1: Only PL side Logic Solidification .....	64

Part 1.6: Use skills to Share .....	67
Part 1.7: Experimental Summary .....	68
Part 2: PS RTC Interrupt Experiment .....	69
Part 2.1: RTC Introduction .....	69
Part 2.2: Interrupt Introduction .....	71
Software Engineer Job Content.....	76
Part 2.3: Vitis Programming .....	76
Part 2.3.1: Create Platform Project.....	76
Part 2.4: Download and Debug .....	83
Part 2.5: Experimental Summary .....	83
Part 3: PS MIO Experiment .....	84
Software Engineer Job Content.....	84
Part 3.1: Principle Introduction .....	85
Part 3.2: Create a “Vivado” Project .....	85
Part 3.3: Vitis Program Development .....	86
Part 3.3.1: MIO Lights up PS LED .....	86
Part 3.3.2: MIO Key Interrupt .....	95
Part 3.4: Knowledge Sharing .....	98
Part 3.5: Experimental Summary .....	102
Part 4: PS Side UART Read and Write Control .....	103
Software Engineer Job Content.....	103
Part 4.1: UART Module Introduction .....	104
Part 4.2: Vitis Program Development .....	105
Part 4.3: Onboard Verification .....	108
Part 4.4: Experimental Summary .....	109
Part 5: PS Side Use of CAN .....	111
Software Engineer Job Content.....	111
Part 5.1: Vitis Program Development .....	111
Part 5.2: Download and Test .....	112

Part 6: PS Side Use of I2C .....	117
Software Engineer Job Content .....	117
Part 6.1: Vitis Program Development .....	117
Part 6.1.1: Temperature Sensor Test .....	117
Part 6.1.2: EEPROM Read and Write .....	121
Part 7: PS Side Use of Display Port .....	124
Software Engineer Job Content .....	124
Part 7.1: Interface Introduction .....	124
Part 7.2: Example Project Introduction .....	125
Part 7.3: On-board verification .....	126
Part 8: PS Side SD Card Read and Write .....	128
Part 8.1: FatFs Introduction .....	128
Part 8.2: Vitis program development .....	129
Part 8.3: Onboard Verification .....	133
Part 9: PS Side Use of Ethernet (LWIP) .....	135
Software Engineer Job Content .....	135
Part 9.1: Vitis Program Development .....	135
Part 9.1.1: LWIP Library Modification .....	135
Part 9.1.2: Create an APP Based on the LWIP Template	143
Part 9.2: Download Debugging .....	143
Part 9.3: Experimental summary .....	145
Part 10: PS Side Remote Update QSPI Flash by Ethernet .....	146
Software Engineer Job Content .....	146
Part 10.1: Vitis Program Development .....	146
Part 10.1.1: UDP Transmission Mode .....	146
Part 10.1.2: TCP Transmission Method .....	148
Part 10.1.3: QSPI Flash Read and Write Control .....	149
Part 10.2: Onboard Verification .....	150
Part 10.2.1: UDP Mode .....	151

Part 10.2.2: TCP Mode .....	153
Part 11: Use of System Monitor .....	155
FPGA Engineer Job Content .....	155
Part 11.1: Hardware Read System Monitor .....	156
Software Engineer Job Content .....	159
Part 11.2: PS read System Monitor Information .....	159
PS and PL Interconnection Parts .....	161
Part 12: PS Side Use of EMIO .....	162
Part 12.1: Principle Introduction .....	162
FPGA Engineer Job Content .....	163
Part 12.2: Create a Vivado Project .....	163
Part 12.3: XDC File Constraint PL Pin .....	165
Software Engineer Job Content .....	166
Part 12.4: Vitis Programming .....	167
Part 12.4.1: EMIO Lights PL LED .....	167
Part 12.4.2: EMIO Implements PL Key Interrupt .....	168
Part 12.5: Build Project .....	169
Part 12.6: EMIO Usage of UART Serial Port .....	170
Part 12.7: Pin Binding Common Errors .....	172
Part 12.8: Experimental Summary .....	173
Part 13: PL Side Use of AXI GPIO .....	174
Part 13.1: Principle Introduction .....	174
FPGA Engineer Job Content .....	175
Part 13.2.1: Create a “Vivado” Project .....	175
Part 13.2.2: Add “AXI GPIO” .....	176
Part 13.3: XDC File Constraint PL Pin .....	183
Software Engineer Job Content .....	185
Part 13.4: Vitis Programming .....	185
Part 13.4.1: AXI GPIO lights PL LED .....	185

Part 13.4.2: Download and Debug .....	188
Part 13.4.4: PL Side AXI GPIO Key Interrupt .....	191
Part 13.5: Experimental summary .....	193
Part 13.6: Knowledge Sharing .....	193
Part 14: PL Side RS485 Test.....	197
FPGA Engineer Job Content .....	197
Part 14.1: Create a Hardware Project.....	197
Software Engineer Job Content.....	203
Part 14.2: Vitis Program Development.....	203
Part 14.3: Download Test.....	205
Part 14.4: Experimental Summary .....	206
Part 15: PL Side Use of Ethernet .....	207
FPGA Engineer Job Content .....	207
Part 15.1: Create a Hardware Project.....	207
Software Engineer Job Content .....	214
Part 15.2: Vitis Program Development .....	214
Part 15.2.1: PL Side Ethernet test .....	214
Part 15.2.2: PS Side Ethernet Test.....	215
Part 16: Custom IP experiment.....	216
FPGA Engineer Job Content .....	216
Part 16.1: PWM Introduction .....	216
Part 16.2: Building a Vivado roject .....	218
Part 16.2.1: Create a custom IP .....	218
Part 16.2.2: Add a Custom IP to the Project .....	226
Part 16.3: Vitis software Writing and Debugging .....	228
Part 16.4: Experimental Summary .....	232
Part 17: Use of Dual Core AMP .....	233
Software Engineer Job Content .....	233
Part 17.1:Vitis Program Development .....	233

Part 17.1.1: Create CPU0 Vitis Project .....	233
Part 17.1.2: Create a CPU1 Vitis project .....	235
Part 17.1.3: CPU0 Programs Introduction .....	236
Part 17.1.4: CPU1 Programs Introduction .....	237
Part 17.3: Onboard Verification .....	238
Part 17.3: QSPI Flash Startup .....	239
Part 17.4: Experimental Summary .....	241
Part 18: Use of “Free RTOS” under ZYNQ .....	242
Software Engineer Job Content .....	242
Part 18.1: Vitis Program Development .....	242
Part 18.2: Onboard Verification .....	244
Part 18.3: Experimental Summary .....	245
Part 19: PL Read and Write PS DDR Data .....	246
FPGA Engineer Job Content .....	246
Part 19.1: Use of ZYNQ HP Port .....	246
FPGA Engineer Job Content .....	247
Part 19.2: Hardware Environment .....	247
Part 19.3: PL Side AXI Master .....	252
Part 19.4: Verification of ddr Read and Write Data .....	255
Part 19.5: Vivado Software Debugging Skills .....	256
Part 19.6: Vitis Program Development .....	256
Part 19.7: Experimental Summary .....	258
Part 20: Realize PS and PL Data Interaction through BRAM ..	259
FPGA Engineer Job Content .....	260
Part 20.1: Hardware Environment .....	260
Part 20.1.1: Block Design adds logic analyzer method	264
Part 20.2: Vitis Program Development .....	266
Part 20.3: Experimental Result .....	268
Part 20.4: Experimental Summary .....	271

Part 21: DMA Loop Test .....	272
FPGA Engineer Job Content .....	273
Part 21.1: Experimental Description .....	273
Part 21.2: Hardware Environment .....	277
Software Engineer Job Content .....	284
Part 21.3: Vitis program development.....	284
Part 21.4: Program verification .....	288
Part 21.5 Experimental Summary .....	290
Part 22: Use of DMA--DAC Waveform Generator (AN108) ....	292
Part 22.1: Experiment Principle .....	293
Part 22.1.1: Digital to Analog Conversion Circuit.....	293
Part 22.1.2: Analog-to-digital Conversion (AD) Circuit	295
FPGA Engineer Job Content .....	296
Part 22.2: Hardware Environment .....	296
Part 22.2.1: Build Hardware .....	296
Part 22.2.2: DAC Custom IP Function Introduction ...	299
Part 22.3: Custom IP Port Mapping .....	299
Software Engineer Job Content .....	303
Part 22.4: Vitis Program Development.....	303
Part 22.4.1: Add math.h Library .....	306
Part 22.5: Onboard Verification .....	307
Part 22.6: Experimental Summary .....	310
Part 23: Use of DMA--ADC oscilloscope (AN108) .....	311
FPGA Engineer Job Content .....	311
Part 23.1: Hardware Environment .....	312
Part 23.1.1: Build Hardware .....	312
Part 23.1.2: ADC Custom IP Function Introduction ...	316
Software Engineer Job Content .....	317
Part 23.2: Vitis Program Development.....	317

Part 23.2.1: Add the “math.h” Library .....	323
Part 23.3: Onboard Verification .....	324
Part 23.4: Experimental Summary .....	327
Part 24: Use of DMA--ADC oscilloscope (AN9238) .....	328
Part 24.1: Hardware Introduction .....	328
Part 24.1.1: 2-Channel AD Module AN9238 .....	328
Part 24.1.2: AN9238 Module Function Description ...	329
FPGA Engineer Job Content .....	332
Part 24.2: Hardware Environment .....	332
Software Engineer Job Content .....	337
Part 24.3: Vitis Program Development .....	337
Part 24.4: Onboard Verification .....	338
Part 25: Use of DMA--ADC Oscilloscope (AN706) .....	341
Part 25.1: Experimental Principle .....	342
Part 25.1.1: AD7606 Timing .....	343
Part 25.1.2: AD7606 Configuration .....	344
Part 25.1.3: AD7606 AD Conversion .....	346
FPGA Engineer Job Content .....	346
Part 25.2: Hardware Environment .....	347
Part 25.2.1: Hardware Setup .....	347
Part 25.2.2: ADC Custom IP Function Introduction ...	349
Software Engineer Job Content .....	349
Part 25.3: SDK Program Development .....	350
Part 25.4: Onboard Verification .....	350
Part 26: Use of DMA--ADC Oscilloscope (AN108) .....	353
FPGA Engineer Job Content .....	354
Software Engineer Job Content .....	354
Part 26.1: Introduction to SG DMA Principle .....	354
Part 26.2: Vitis Program Development .....	359

Part 26.3: Experimental Summary .....	363
Part 27: Use the Scatter/Gather DMA Based on DAC Module (AN9767) .....	364
FPGA Engineer Job Content .....	369
Part 27.2: Hardware Environment .....	369
Part 27.2.1: Build Hardware .....	369
Part 27.2.2: DAC custom IP function introduction .....	372
Software Engineer Job Content .....	373
Part 27.3: Vitis Program Development .....	373
Part 27.4: Onboard verification .....	374
Part 28: AN5642 Binocular Camera Collection and Display ....	377
Part 28.1: OV5640 Chip Introduction .....	378
Part 28.2: Use of VDMA .....	378
FPGA Engineer Job Content .....	380
Part 28.3: Hardware Environment .....	380
Software Engineer Job Content .....	386
Part 28.4: Vitis Program Development .....	386
Part 28.4.1: Monocular Camera Display .....	386
Part 28.4.2: Binocular Camera Display .....	389
Part 28.5: Experimental Summary .....	390
Part 29: SD card read and write operation - camera capture ..	391
Software Engineer Job Content .....	391
Part 29.1: Vitis Program Development .....	391
Part 21.3: Onboard Verification .....	394
Part 30: Binocular camera Ethernet transmission .....	397
FPGA Engineer Job Content .....	397
Part 30.1: Hardware environment .....	398
Software Engineer Job Content .....	398
Part 30.2: Vitis program development .....	398

Part 30.2.1: Image control section .....	399
Part 30.2.2: LWIP Control Section .....	403
Part 30.3: Onboard verification .....	412
Part 30.4: Experimental Summary .....	416
<b>Part 31: MIPI Acquisition and Display Based on AN5641 Module</b>	<b>417</b>
Part 31.1: Principle Introduction .....	417
Part 31.1.1: MIPI Physical Layer (D-PHY) .....	417
Part 31.1.2: MIPI Protocol Layer (CSI-2) .....	418
FPGA Engineer Job Content .....	419
Part 31.2: Hardware Environment .....	420
Software Engineer Job Content .....	429
Part 31.3: Vitis Program Development .....	429
Part 31.4: Onboard Verification .....	430
<b>Part 32: Audio Module AN831 Recording and Playback</b> .....	<b>433</b>
Part 32.1: Audio Module AN831 Module Description .....	433
FPGA Engineer Job Content .....	436
Part 32.2: Create Hardware Environment .....	437
Software Engineer Job Content .....	443
Part 32.3: Vitis Program Development .....	443
Part 32.3.1: SD Card Playing Music Experiment .....	443
Part 32.3.2: On Board Verification .....	447
Part 32.3.3: Recording and Playback Experiment .....	449
Part 32.3.4: On Board Verification .....	452
<b>Part 33: Use of 7 inch LCD Module</b> .....	<b>454</b>
Part 33.1: 7-inch LCD Module Description .....	454
FPGA Engineer Job Content .....	454
Part 33.2: Hardware Environment .....	454
Software Engineer Job Content .....	466
Part 33.3: Vitis Program Development .....	466

Part 33.4: Onboard Verification .....	467
Part 34: 7-inch Touch Screen GUI and Touch Control .....	469
Software Engineer Job Content .....	469
Part 34.1: Use of µGUI .....	469
Part 34.2: Vitis Program Development .....	471
Part 34.3: Onboard Verification .....	475
Part 35 Ethernet Transmission--ADC Acquisition Based on AN108 Module .....	478
Software Engineer Job Content .....	478
Part 35.1: Develop a Transmission Protocol .....	478
Part 35.2: Vitis program development .....	480
Part 35.2.1: ADC acquisition part .....	480
Part 35.2.2: LWIP control section .....	481
Part 35.3: Onboard Verification .....	484
Part 35.4: PC software instructions .....	488
Part 35.5: Data Saving Demos .....	489
Part 36: Ethernet Transmission--ADC Acquisition Based on AN9238 Module .....	490
Software Engineer Job Content .....	490
Part 36.1: Vitis Program Development .....	490
Part 36.1.1: ADC Acquisition Part .....	490
Part 36.1.2: LWIP Control Section .....	491
Part 36.2: Onboard Verification .....	492
Part 37: Ethernet Transmission--ADC Acquisition Based on AN706 Module .....	495
Software Engineer Job Content .....	495
Part 37.1: Vitis Program Development .....	495
Part 37.1.1: ADC Acquisition Part .....	495
Part 37.1.2: LWIP Control Section .....	496

Part 37.2: Onboard Verification .....	497
---------------------------------------	-----

# Preparation and precautions

## Software Environment

The software development environment is based on Vivado 2020.1

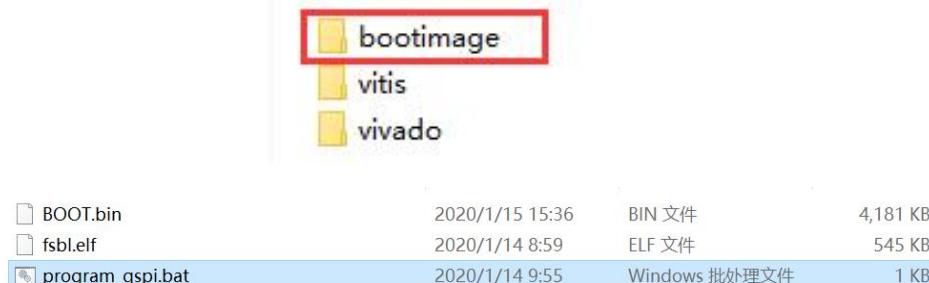


## Hardware environment

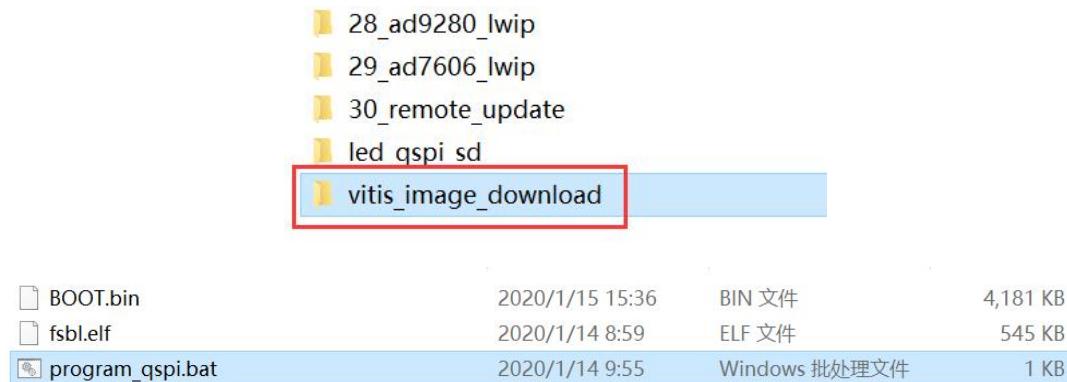
FPGA Development Board Model	FPGA Chip
AXU3EG	Xazu3eg-sfvc784-1-i
AXU4EV	Xczu4ev-sfvc784-1-i
AXU5EV	xazu5ev-sfvc784-1-i

## Batch Download QSPI Flash

There is a **bootimage** folder under all project directories, which stores the corresponding **BOOT.bin** file. You can copy this file to the **Vitis\_image\_download** folder to overwrite the original **BOOT.bin**. You can also put **BOOT.bin** on the SD card to start the verification function



The **vitis\_image\_download** folder is under the **course\_s2** directory, enter the folder, right-click **program\_qspi.bat**, and open it for editing



Change the **program\_flash** path to your own software installation path, save and close.

```
call C:\Xilinx\Vitis\Vitis\2020.1\bin\program_flash] -f BOOT.bin -offset 0 -flash_type qspi-x4-single -fsbl fsbl.elf -verify
pause
```

Double-click **program\_qspi.bat** to download **BOOT.BIN** to **QSPI FLASH**. It is recommended to download in JTAG mode.

```
***** Xilinx Program Flash
***** Program Flash v2017.4 (64-bit)
**** SW Build 2086221 on Fri Dec 15 20:55:39 MST 2017
** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.

Connecting to hw_server @ TCP:localhost:3121
Connected to hw_server @ TCP:localhost:3121
Available targets and devices:
Target 0: jsn-JTAG-HS1-210512180081
    Device 0: jsn-JTAG-HS1-210512180081-4ba00477-0
Retrieving Flash info...
Initialization done, programming the memory
BOOT_MODE REG = 0x00000000
f probe 0 0 0
Performing Erase Operation...
Erase Operation successful.
INFO: [Xicom 50-44] Elapsed time = 1 sec.
Performing Program Operation...
0%...100%
Program Operation successful.
INFO: [Xicom 50-44] Elapsed time = 1 sec.
Performing Verify Operation...
0%...50%...100%
INFO: [Xicom 50-44] Elapsed time = 2 sec.
Verify Operation successful.

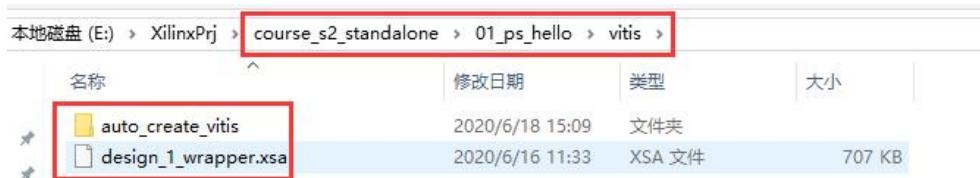
Flash Operation Successful
请按任意键继续 . . .
```

You can also use the SD card boot method to copy the **BOOT.bin** file to the SD to boot.

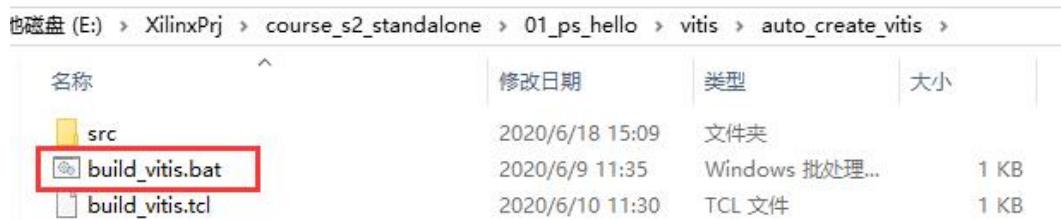
## Batch process to build Vitis project

Since the Vitis project occupies a large space after compilation, in order to save everyone's precious time, we provide the batch **tcl** script of the Vitis project. There is a **vitis** folder under each project, which

contains the hardware description file **xx.xsa**, and the script for automatically creating the project



What you need to do is edit the **build\_vitis.bat** file in the **auto\_create\_vitis** folder



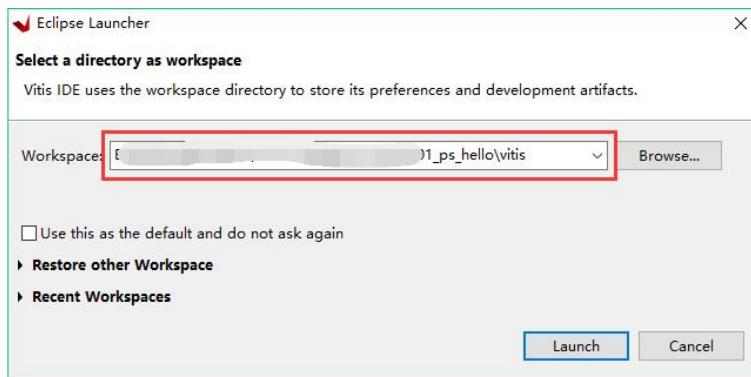
Replace the **xsct.bat** path in the yellow box with the path installed by yourself, the path is **xx\Vitis\2020.1\bin\xsct.bat**

```
call E:\XilinxVitis\Vitis\2020.1\bin\xsct.bat build_vitis.tcl
pause
```

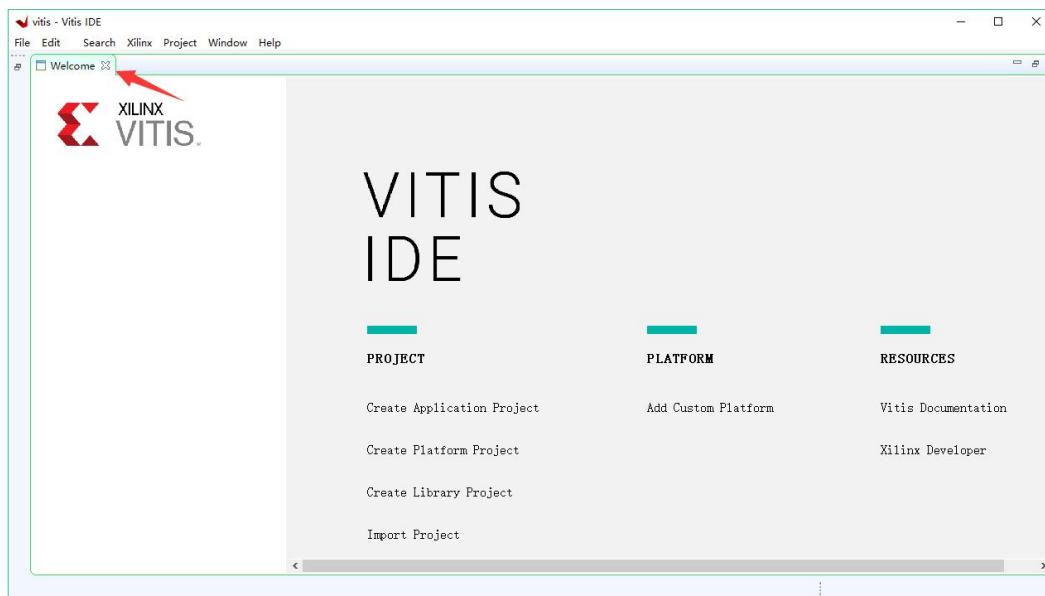
After saving, double-click **build\_vitis.bat** to create the project  
Compilation is over, press any key to exit

```
C:\Windows\system32\cmd.exe
"Compiling uartps"
"Running Make libs in psu_cortexa53_0/libsrc/usbpsu_v1_7/src"
make -C psu_cortexa53_0/libsrc/usbpsu_v1_7/src -s libs "SHELL=CMD" "COMPILER=aarch64-none-elf-gcc" "ASSEMBLER=aarch64-none-elf-as" "ARCHIVER=aarch64-none-elf-ar" "COMPILER_FLAGS=-O2 -c" "EXTRA_COMPILER_FLAGS=-g -Wall -Wextra"
"Compiling usbpsu"
"Running Make libs in psu_cortexa53_0/libsrc/video_common_v4_9/src"
make -C psu_cortexa53_0/libsrc/video_common_v4_9/src -s libs "SHELL=CMD" "COMPILER=aarch64-none-elf-gcc" "ASSEMBLER=aarch64-none-elf-as" "ARCHIVER=aarch64-none-elf-ar" "COMPILER_FLAGS=-O2 -c" "EXTRA_COMPILER_FLAGS=-g -Wall -Wextra"
"Compiling video_common"
"Running Make libs in psu_cortexa53_0/libsrc/zdma_v1_9/src"
make -C psu_cortexa53_0/libsrc/zdma_v1_9/src -s libs "SHELL=CMD" "COMPILER=aarch64-none-elf-gcc" "ASSEMBLER=aarch64-none-elf-as" "ARCHIVER=aarch64-none-elf-ar" "COMPILER_FLAGS=-O2 -c" "EXTRA_COMPILER_FLAGS=-g -Wall -Wextra"
"Compiling zdma"
'Finished building libraries'
请按任意键继续...
```

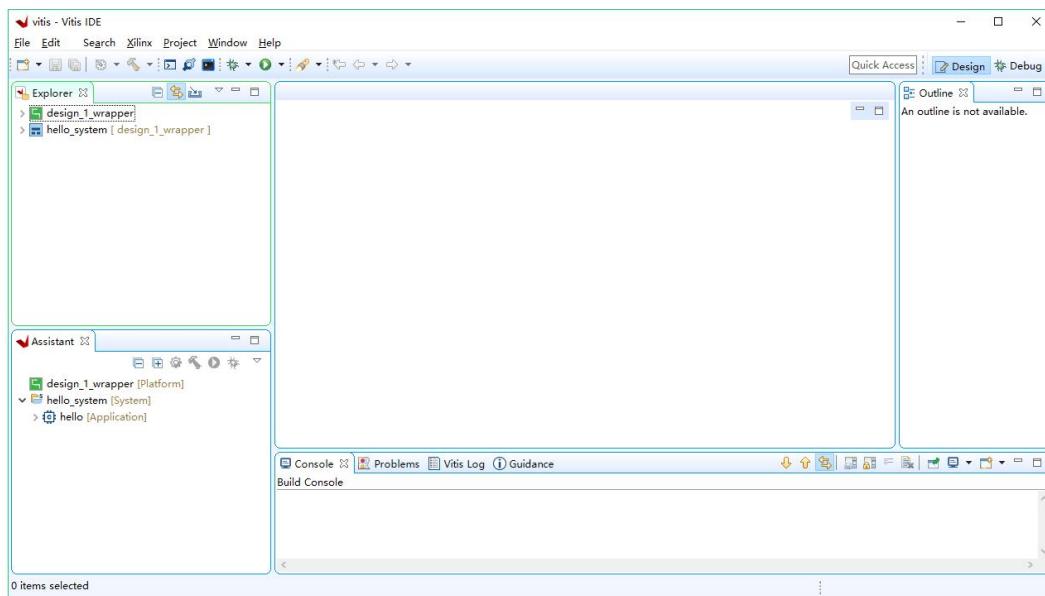
Open the **Vitis** software, select the project path, **Launch**



After opening, close the **Welcome** interface



The project is ready to use



## PS side Peripherals Parts

The basic chapter mainly introduces the configuration of the ZYNQ core, the application of the PS side, such as the basic experiments of MIO, Ethernet, RTC, etc., to lay the foundation for the following applications.

# Part 1: Experience ARM, bare metal output "Hello World"

**The vivado project directory is "ps\_hello/vivado"**

**The vitis project directory is "ps\_hello/vitis"**

**From this chapter, it is implemented by FPGA engineers and software development engineers.**

The previous experiments were carried out on the PL side. It can be seen that there is no difference with the normal FPGA development process. The main advantage of ZYNQ is the reasonable combination of FPGA and ARM, which puts higher demands on developers. From the beginning of this chapter, we started to use ARM, which is what we call PS. In this chapter, we use a simple serial port printing to experience Vivado.Vitis and PS side features.

The previous experiments are all things that FPGA engineers should do. From the beginning of this chapter, there is a division of labor. FPGA engineers are responsible for setting up the Vivado project and providing good hardware to software developers. Software developers can develop applications on this basis program. The division of labor is also conducive to the progress of the project. If a software developer wants to do everything, it may take a lot of time and energy to learn the knowledge of FPGA. It is a painful process to change from software thinking to hardware thinking. It's another matter. A professional person is a good choice for doing professional things.

## Part 1.1: Hardware Introduction

We can see from the schematic diagram that the ZYNQ chip is

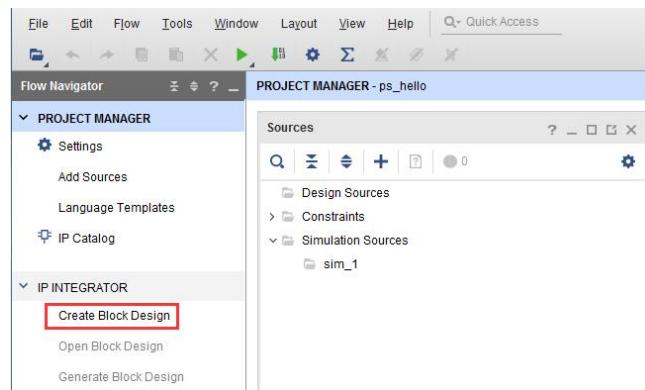
divided into PL and PS. The IO assignment on the PS side is relatively fixed and cannot be arbitrarily assigned, and there is no need to assign pins in the Vivado software. Although this experiment only uses PS, a Vivado project must also be established to configure the PS pins. Although the ARM on the PS side is a hard core, the ARM hard core must also be added to the project in ZYNQ to use it. The previous chapter introduced the project in code form, this chapter begins to introduce the graphical design of ZYNQ to build the project

## FPGA Engineer Job Content

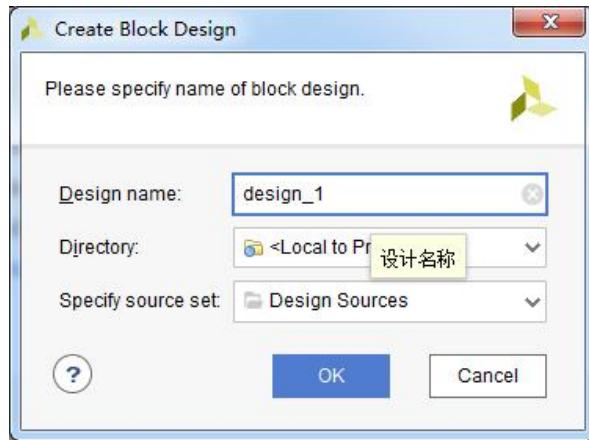
The following is the content that FPGA engineers are responsible for.

### Part 1.2: Create a Vivado Project

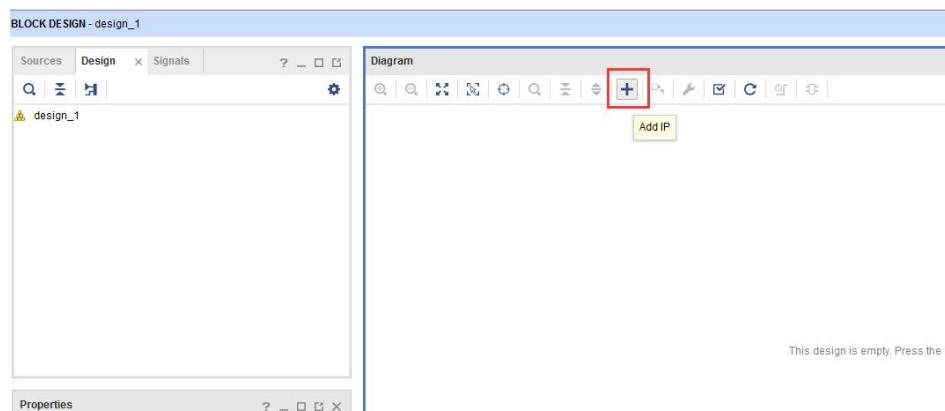
- 1) Create a project called "[ps\\_hello](#)". The creation process will not be repeated, please refer to "PL's "Hello World" LED Experiment".
- 2) Click on "[Create Block Design](#)" to create a block design, which is a graphical design



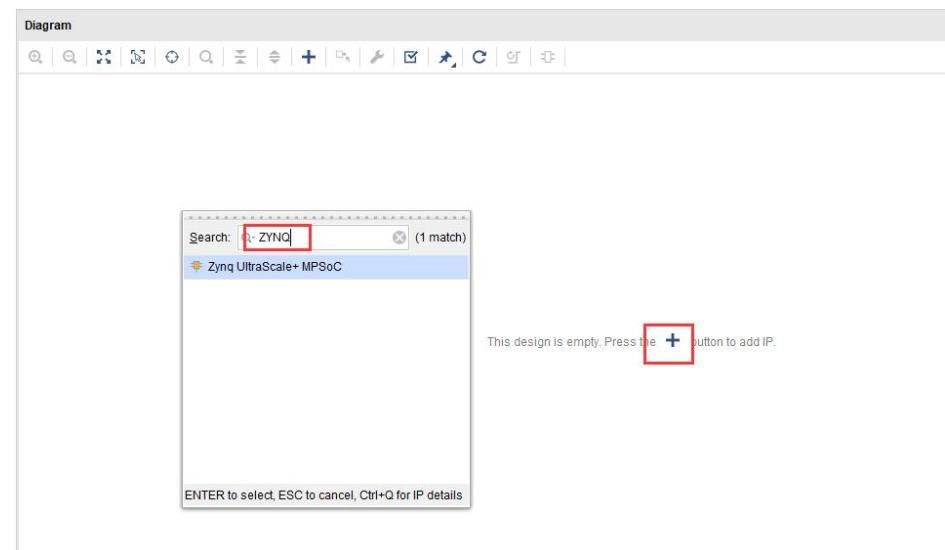
- 3) "Design name" is not modified here, keep the default "design\_1", which can be modified as needed, but the name should be as short as possible, otherwise there will be problems compiling under Windows.



- 4) Click on the "Add IP" shortcut icon



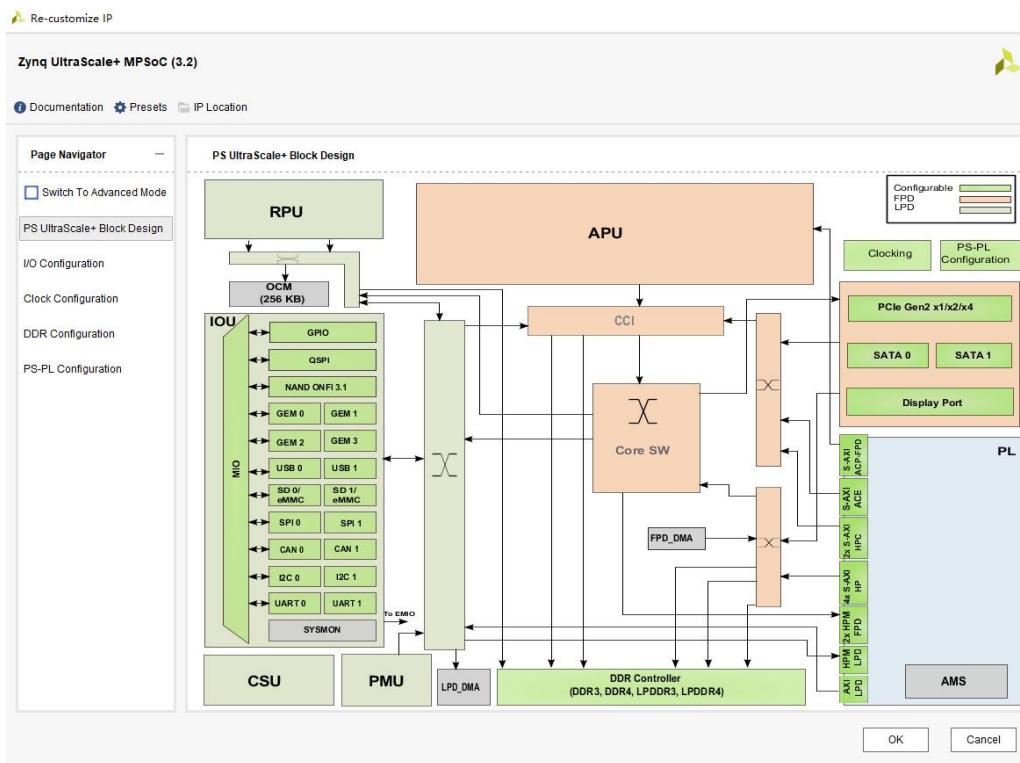
- 5) Search for "zynq" and double-click "ZYNQ UltraScale+ MPSoC" in the search results list



- 6) Double-click "ZYNQ7 UltraScale+ MPSoC" in the block diagram to configure related parameters.



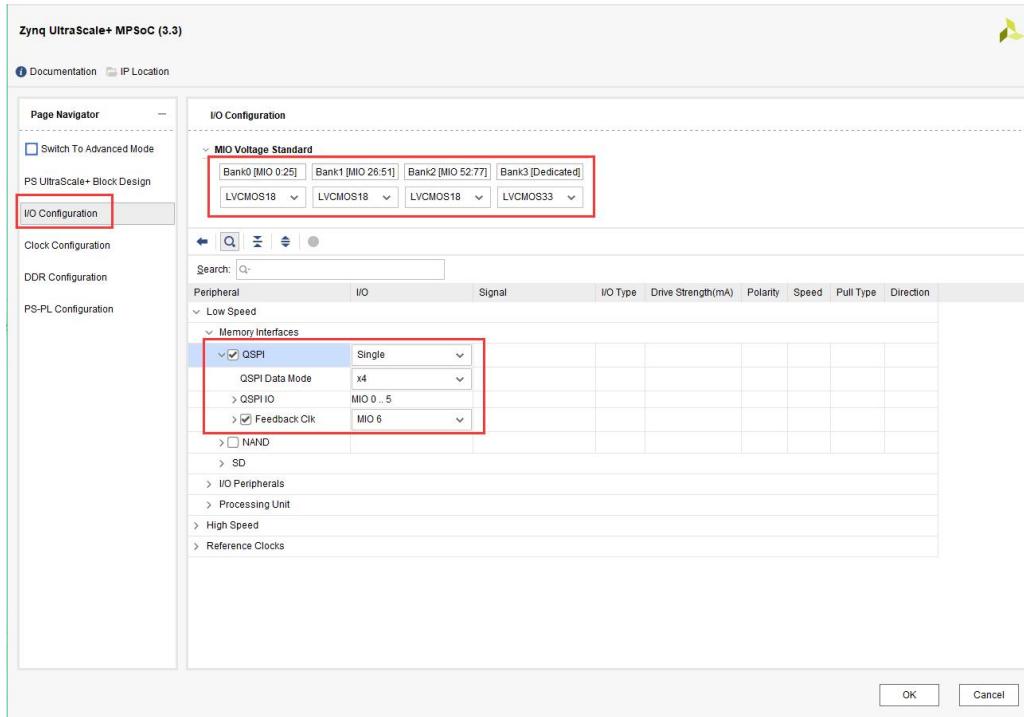
- 7) The first interface that appears is the ZYNQ hard core architecture diagram. You can clearly see its structure. You can refer to the ug585 document, which contains a detailed introduction to ZYNQ. The green part in the picture is the configurable module. You can click to enter the corresponding editing interface. Of course, you can also enter the editing in the window on the left. The functions of each window are introduced below.



### Part 1.2.1: Low Speed Configuration

- 1) In the I/O Configuration window, configure the voltage of BANK0~BANK2 as LVCMOS18, and the voltage of BANK3 as

LVCMS33. First configure Low Speed pin, check QSPI, and set it to "Single" mode, Data Mode is "x4", check Feedback Clk



- 2) Check SD 0 to configure eMMC. Select MIO13..22, Slot Type select eMMC, Data Transfer Mode is 8Bit, check Reset, and select MIO23

Peripheral	I/O	Signal	I/O Type	Drive Strength(mA)	Polarity	Speed	Pull Type	Direction
> NAND								
> SD								
SD 0	MIO 13 .. 22							
Slot Type	eMMC							
Data Transfer Mode	8Bit							
Reset	MIO 23							
SD 0	MIO13	sdio0_data_out[0]	cmc	12	Def	fas	pullu	inout
SD 0	MIO14	sdio0_data_out[1]	cmc	12	Def	fas	pullu	inout
SD 0	MIO15	sdio0_data_out[2]	cmc	12	Def	fas	pullu	inout
SD 0	MIO16	sdio0_data_out[3]	cmc	12	Def	fas	pullu	inout
SD 0	MIO17	sdio0_data_out[4]	cmc	12	Def	fas	pullu	inout
SD 0	MIO18	sdio0_data_out[5]	cmc	12	Def	fas	pullu	inout
SD 0	MIO19	sdio0_data_out[6]	cmc	12	Def	fas	pullu	inout

- 3) Check SD 1 to configure SD card. Select MIO 46..51, Slot Type select SD 2.0, Data Transfer Mode select 4Bit, check CD to detect SD card insertion, select MIO45

Peripheral	I/O	Signal	I/O Type	Drive Strength(mA)	Polarity	Speed	Pull Type	Direction
SD 0	MIO22	sdio0_clk_out	cmc	12	Def	fas	pullu	out
SD 0	MIO23	sdio0_bus_pow	cmc	12	Def	fas	pullu	out
✓ SD 1	MIO 46..51							
Slot Type	SD 2.0							
Data Transfer Mode	4Bit							
✓ CD	MIO 45							
<input type="checkbox"/> Power								
<input type="checkbox"/> WP								
SD 1	MIO45	sdio1_cd_n	cmc	12	Def	fas	pullu	in
SD 1	MIO46	sdio1_data_out[0]	cmc	12	Def	fas	pullu	inout

4) Check CAN 0, select MIO 38..39, check CAN 1, select MIO 32..33

Peripheral	I/O	Signal	I/O Type	Drive Str
<b>Low Speed</b>				
> Memory Interfaces				
> I/O Peripherals				
> CAN				
>✓ CAN 0	MIO 38 .. 39			
>✓ CAN 1	MIO 32 .. 33			
> I2C				
<input type="checkbox"/> PJTAG				

5) Check I2C 1, I2C used for EEPROM, etc., select MIO 24..25

Peripheral	I/O	Signal	I/O Type
<b>Low Speed</b>			
> Memory Interfaces			
> I/O Peripherals			
> CAN			
> I2C			
<input type="checkbox"/> I2C 0			
>✓ I2C 1	MIO 24 .. 25		
<input type="checkbox"/> PJTAG			
> <input type="checkbox"/> PMU			
<input type="checkbox"/> CSU			

6) Check the serial port UART 0, select MIO 42..43, check GPIO1 MIO

> SPI	
✓ UART	
>✓ UART 0	MIO 42 .. 43
> <input type="checkbox"/> UART 1	
✓ GPIO	
<input type="checkbox"/> GPIO EMIO	
<input type="checkbox"/> GPIO0 MIO	
>✓ GPIO1 MIO	MIO 26 .. 51
> <input type="checkbox"/> GPIO2 MIO	
> Processing Unit	

7) Check TTC 0 ~ TTC 3



## Part 1.2.2: High Speed Configuration

- In the **High Speed** part, first configure the PS-side Ethernet, check **GEM 3**, select **MIO 64..75**, check **MDIO 3**, select **MIO 76..77**

Peripheral	I/O	Signal	I/O Type	Drive Strength(mA)	Speed	Pull Type	Direction
<b>High Speed</b>							
<b>GEM</b>							
<b>GEM 0</b>							
<input type="checkbox"/>							
<b>GEM 1</b>							
<input type="checkbox"/>							
<b>GEM 2</b>							
<input type="checkbox"/>							
<b>GEM 3</b>							
<input checked="" type="checkbox"/> <b>MIO 64..75</b>							
<input checked="" type="checkbox"/> <b>MIO 76..77</b>							
Gem 3	MIO64	rgmii_tx_clk	sc...	12	s...	pull...	out
Gem 3	MIO65	rgmii_tx[0]	sc...	12	s...	pull...	out
Gem 3	MIO66	rgmii_tx[1]	sc...	12	s...	pull...	out
Gem 3	MIO67	rgmii_tx[2]	sc...	12	s...	pull...	out
Gem 3	MIO68	rgmii_tx[3]	sc...	12	s...	pull...	out
Gem 3	MIO69	rgmii_tx_ctl	sc...	12	s...	pull...	out

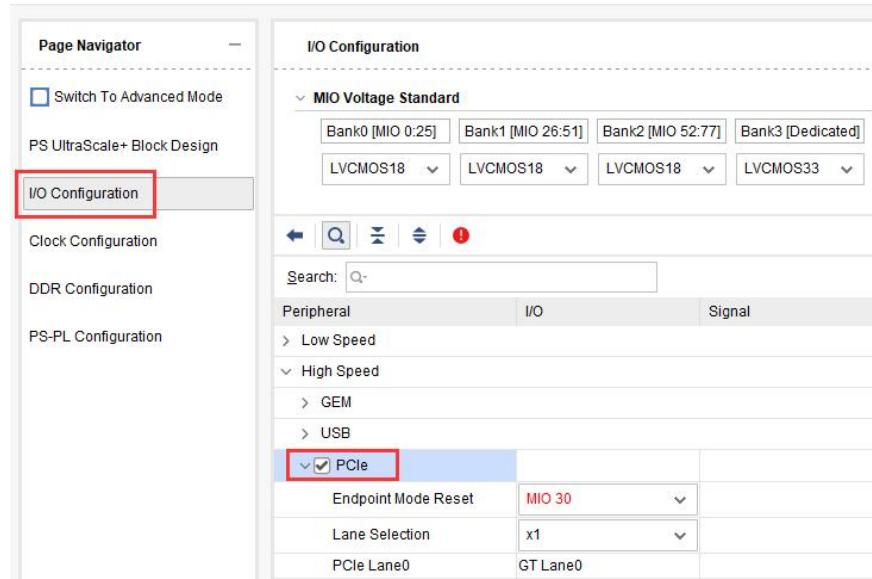
- Check **USB 0**, select **MIO 52..63**, check **USB 3.0**, select **GT Lane1**

Peripheral	I/O	Signal	I/O Type	Drive Strength(mA)	Polarity	Speed
<b>USB</b>						
<b>USB0</b>						
<input checked="" type="checkbox"/> <b>USB 0</b>						
USB 0	MIO52	ulpi_clk_in	cmc	12	Def	fas
USB 0	MIO53	ulpi_dir	cmc	12	Def	fas
USB 0	MIO54	ulpi_tx_data[2]	cmc	12	Def	fas
USB 0	MIO55	ulpi_nxt	cmc	12	Def	fas
USB 0	MIO56	ulpi_tx_data[0]	cmc	12	Def	fas
USB 0	MIO57	ulpi_tx_data[1]	cmc	12	Def	fas
USB 0	MIO58	ulpi_stp	cmc	12	Def	fas
USB 0	MIO59	ulpi_tx_data[3]	cmc	12	Def	fas
USB 0	MIO60	ulpi_tx_data[4]	cmc	12	Def	fas
USB 0	MIO61	ulpi_tx_data[5]	cmc	12	Def	fas
USB 0	MIO62	ulpi_tx_data[6]	cmc	12	Def	fas
USB 0	MIO63	ulpi_tx_data[7]	cmc	12	Def	fas
<input checked="" type="checkbox"/> <b>USB 3.0</b>						

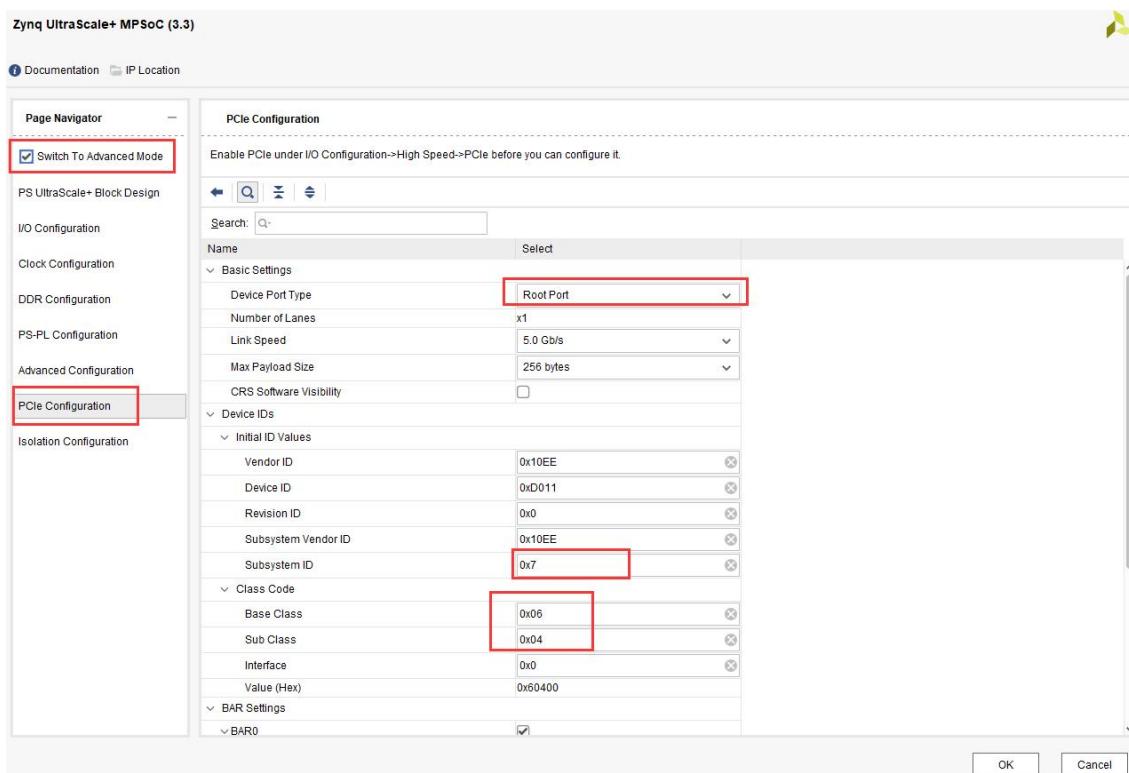
## USB reset select MIO 31



## 3) Check PCIe



## 4) Click Switch To Advanced Mode, select PCIe Configuration, modify the following parameters, and configure it to ROOT mode



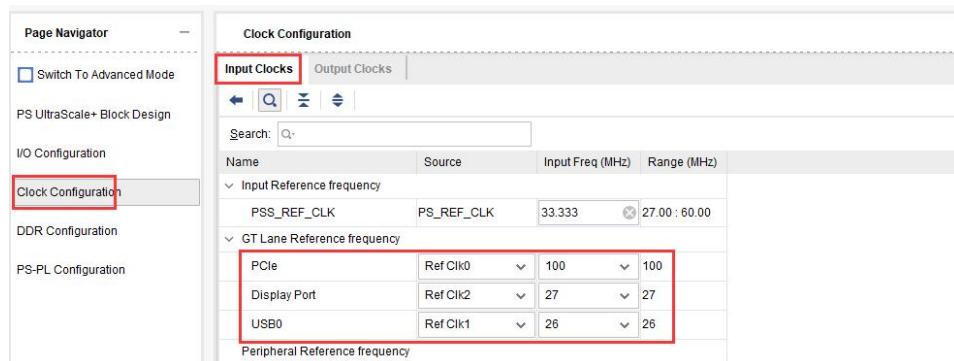
- 5) Go back to I/O Configuration, reset and select MIO 37; check Display Port, select MIO 27..30, and Lane Selection select Dual Higher



At this point, the I/O part is configured

### Part 1.2.3: Clock Configuration

- 1) In the Clock Configuration interface, the Input Clocks window configures the reference clock, where PSS\_REF\_CLOCK is the ARM reference clock and the default is 33.333MHz; PCIe selects Ref Clk0, 100MHz; Display Port selects Ref Clk2, 27MHz; USB0 selects Ref Clk1, 26MHz.



- 2) In the Output Clocks window, if it is not IOPLL, change to IOPLL, keep the same, use the same PLL

The screenshot shows the 'Clock Configuration' tab in the Alinx software. The 'Output Clocks' tab is selected. In the 'Processor/Memory Clocks' section, the 'CPU\_R5' row has its source dropdown highlighted with a red box, showing 'IOPPLL' as the selected option. Other rows in this section include QSPI, SDIO0, SDIO1, SD DLL, UART0, I2C1, CAN0, CAN1, USB0, USB3\_DUAL, Gem3, GEM\_TSU, TTC0, and TTC1. The 'PL Fabric Clocks' section below also has a red box around the PL0 row, where 'PL0' is checked and 'RPLL' is selected as the source.

- 3) The PL clock remains the default, which is the clock provided to the PL side logic.

This screenshot shows the same 'Clock Configuration' tab as above, but focusing on the 'PL Fabric Clocks' section. A red box highlights the first row for 'PL0', where the 'PL0' checkbox is checked and the 'RPLL' dropdown is selected. The other three rows for PL1, PL2, and PL3 show their respective checkboxes unchecked and RPLL selected as the source.

- 4) For the Full Power part, keep the default for others, change DP\_VIDEO to VPLL, DP\_AUDIO and DP\_STC are changed to RPLL.

This screenshot shows the 'Clock Configuration' tab for the 'Full Power Domain Clocks'. A red box highlights the 'DP\_VIDEO', 'DP\_AUDIO', and 'DP\_STC' rows. 'DP\_VIDEO' is set to 'VPLL', 'DP\_AUDIO' is set to 'RPLL', and 'DP\_STC' is also set to 'RPLL'. Other rows in this section include ACPU (set to 'APLL') and DDR (set to 'DPLL'). The 'Peripherals/I/O Clocks' and 'System Debug Clocks' sections are also visible below.

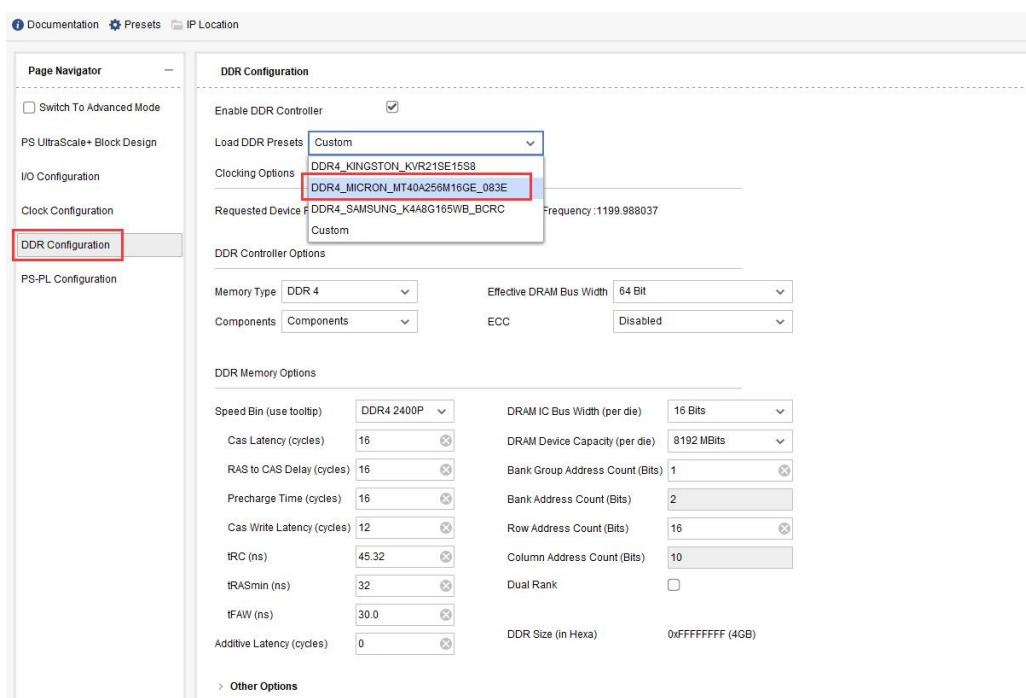
The bottom Interconnect is modified as follows

FPD_DMA	DPLL	600	X	2	599.994019	0.0000...
DPDMA	DPLL	600	X	2	599.994019	0.0000...
TOPSW_MAIN	APLL	533.333	X	2	533.328003	0.0000...
TOPSW_LSBUS	IOPLL	100	X	5	99.999001	0.0000...

Keep the other parts as default, so far, the clock part configuration is complete.

#### Part 1.2.4: DDR Configuration

- 1) In the DDR Configuration window, select Load DDR Presets "DDR4\_MICRON\_MT40A256M16GE\_083E"



The parameters are modified as follows

Page Navigator

- Switch To Advanced Mode
- PS UltraScale+ Block Design
- I/O Configuration
- Clock Configuration
- DDR Configuration**
- PS-PL Configuration

DDR Configuration

Enable DDR Controller

Load DDR Presets: Custom

Clocking Options

Requested Device Frequency (MHz): 1200 Actual Device Frequency: 1199.988037

DDR Controller Options

Memory Type	DDR 4	Effective DRAM Bus Width	64 Bit
Components	Components	ECC	Disabled

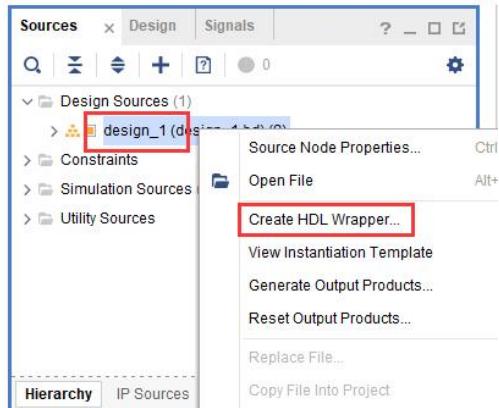
DDR Memory Options

Speed Bin (use tooltip)	DDR4 2400P	DRAM IC Bus Width (per die)	16 Bits
Cas Latency (cycles)	16	DRAM Device Capacity (per die)	8192 MBits
RAS to CAS Delay (cycles)	16	Bank Group Address Count (Bits)	1
Precharge Time (cycles)	16	Bank Address Count (Bits)	2
Cas Write Latency (cycles)	12	Row Address Count (Bits)	16
tRC (ns)	45.32	Column Address Count (Bits)	10
tRASmin (ns)	32	Dual Rank	<input type="checkbox"/>
tFAW (ns)	30.0	DDR Size (in Hexa)	0xFFFFFFFF (4GB)
Additive Latency (cycles)	0		

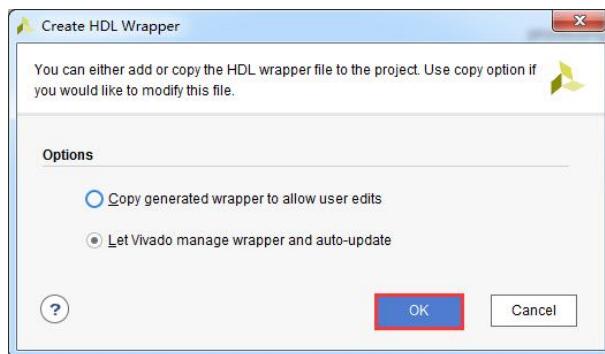
Keep the others as default, click OK, the configuration is complete, and connect the clock as follows:



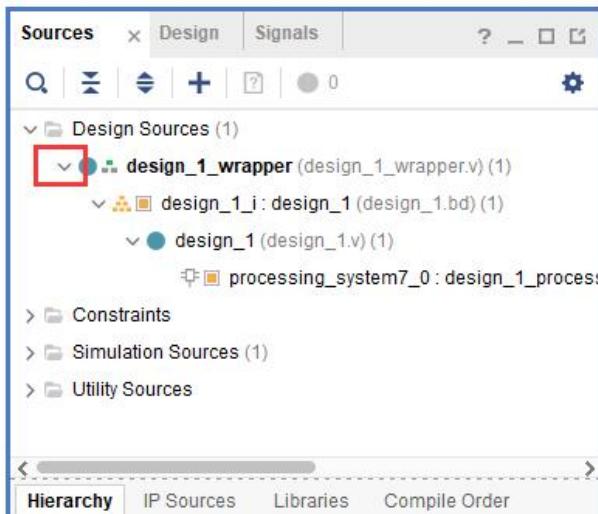
- 2) Select Block design, right click "Create HDL Wrapper...", create a Verilog or VHDL file, and generate HDL top-level file for block design.



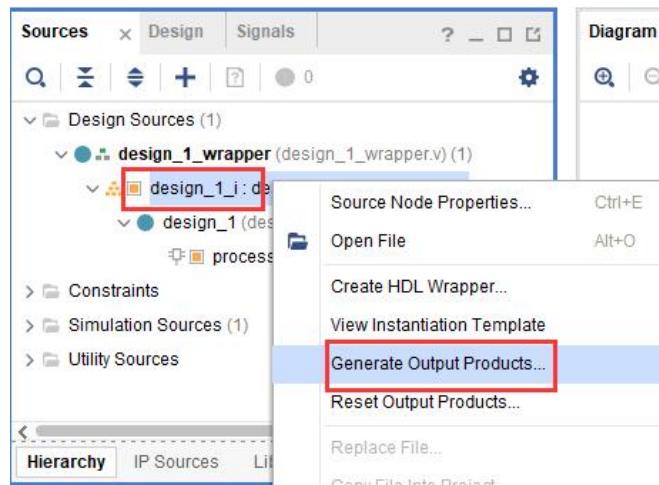
3) Keep the default options and click "OK"



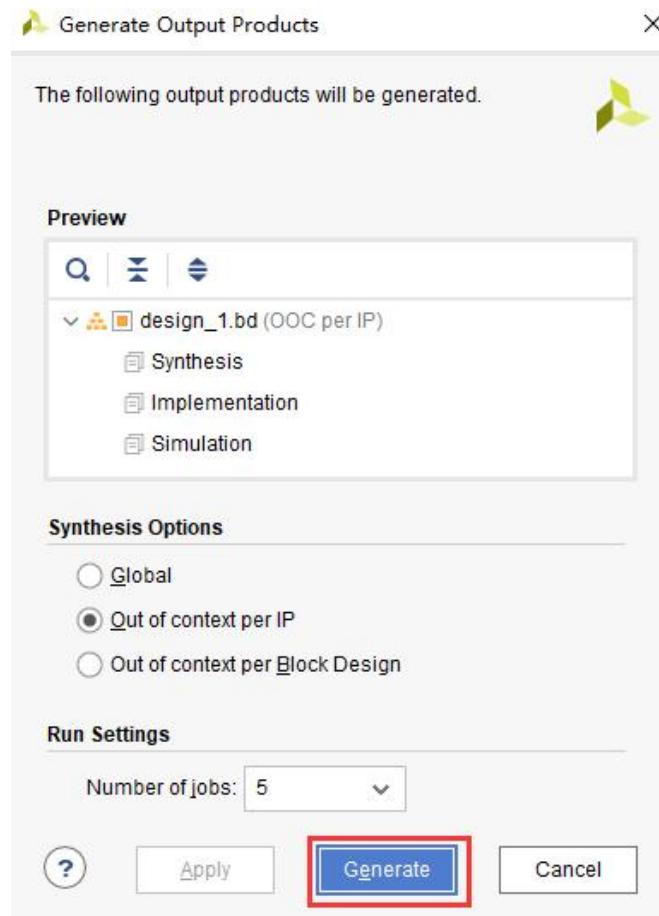
4) Expand the design and you can see that the PS is used as an ordinary IP.



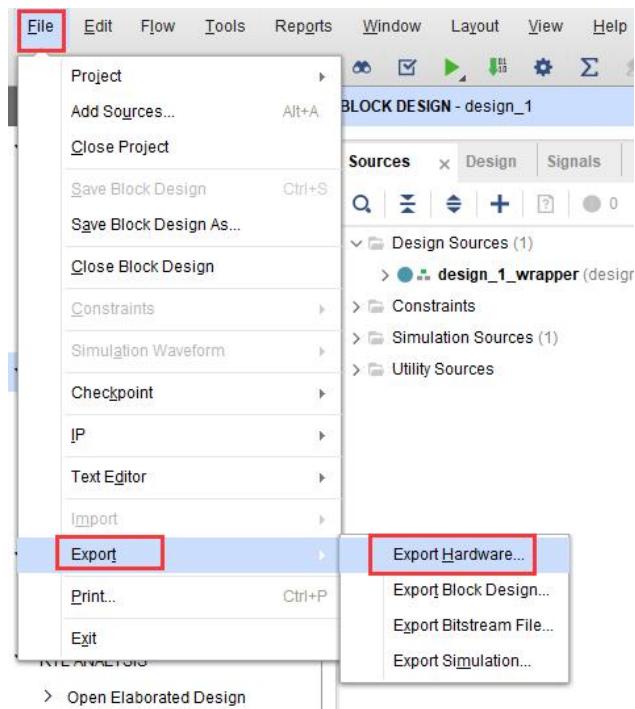
5) Select the block design and right click "Generate Output Products". This step will generate block output files, including IP, instantiation templates, RTL source files, XDC constraints, third-party integrated source files, and so on. For subsequent operations.



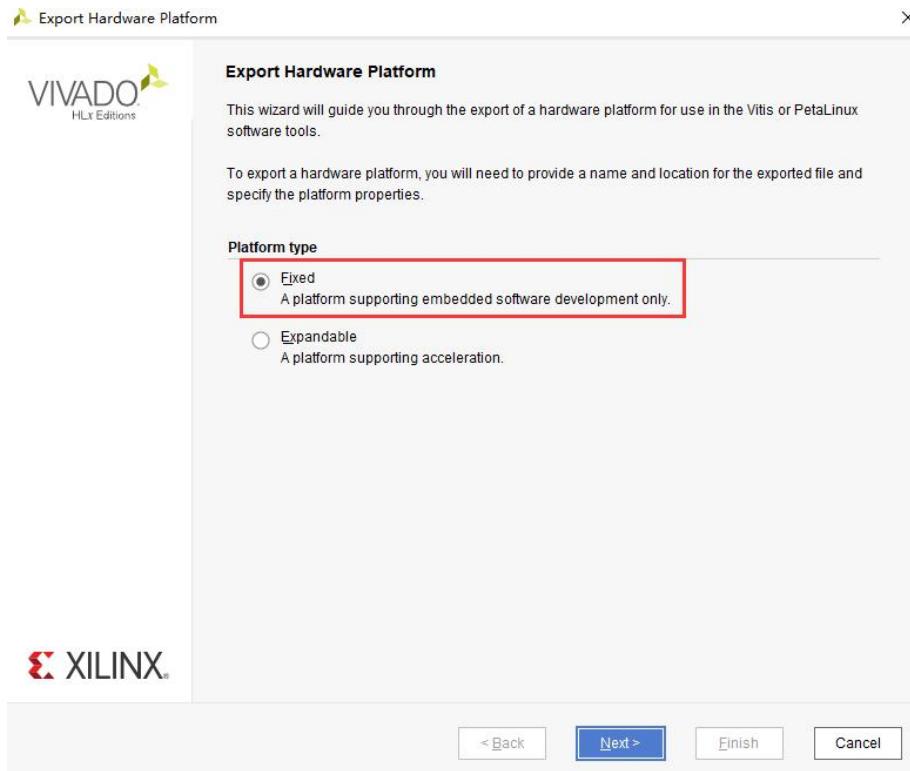
6) Click "Generate"



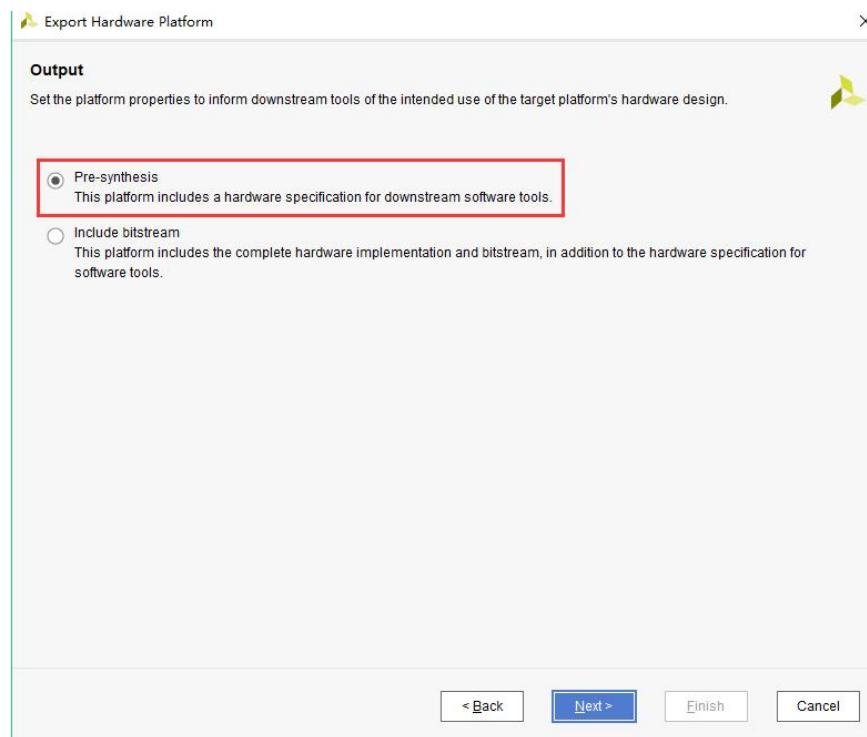
7) Export the hardware information in the menu bar "File -> Export -> Export Hardware...", here it contains the configuration information of the PS side.



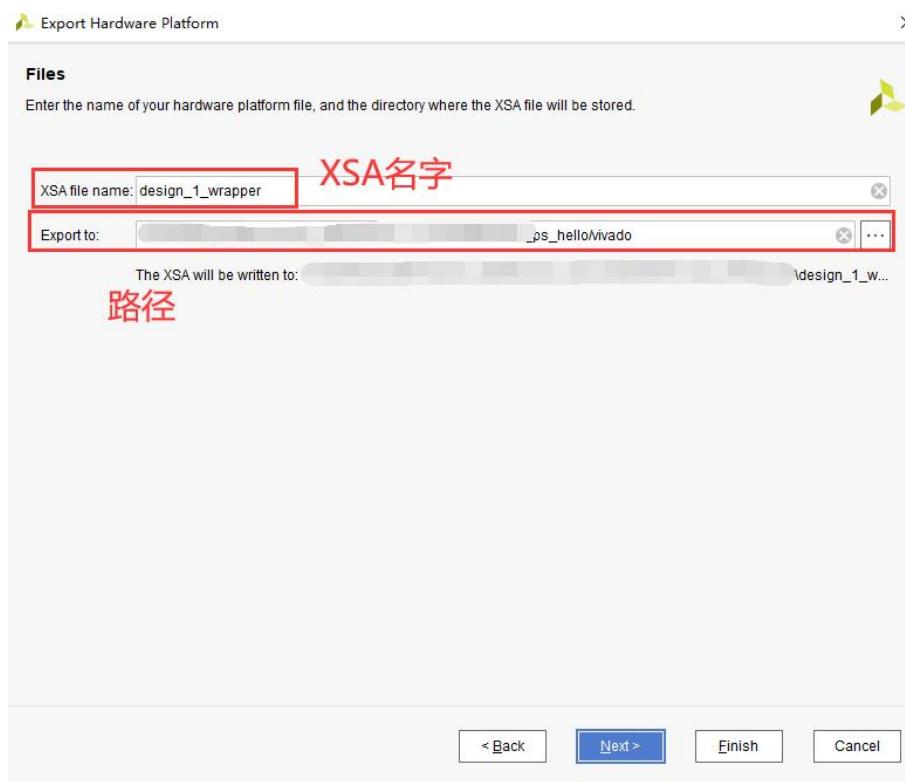
### 8) Select Fixed in the pop-up window, click Next



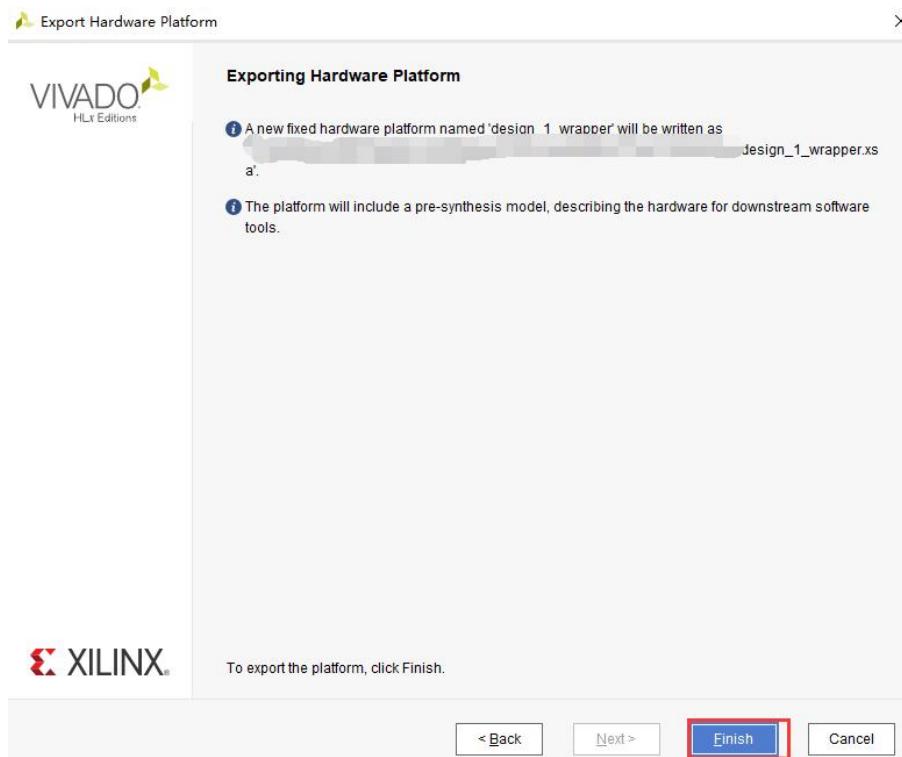
### 9) Click "OK" in the pop-up dialog box, because the experiment only uses the PS serial port and does not require PL to participate. There is no enable here. Do not select "Include bitstream", click Next



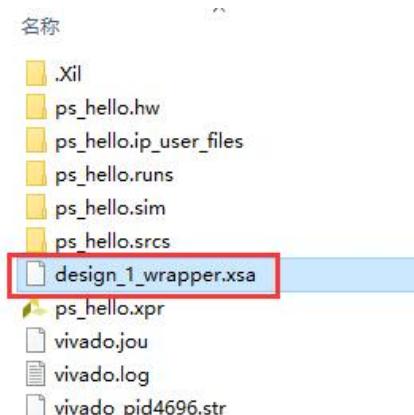
10) The export name and export path can be modified. The default is in the vivado project directory. This file can be placed in a suitable location according to your needs. It does not have to be placed under the vivado project. The vivado and vitis software are independent. Here we choose not to change by default. Click Next



Click Finish



11)At this time, you can see the xsd file in the project directory. This file contains the information of Vivado hardware design and can be used by software developers.



At this point, the FPGA engineer's job comes to an end.

## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

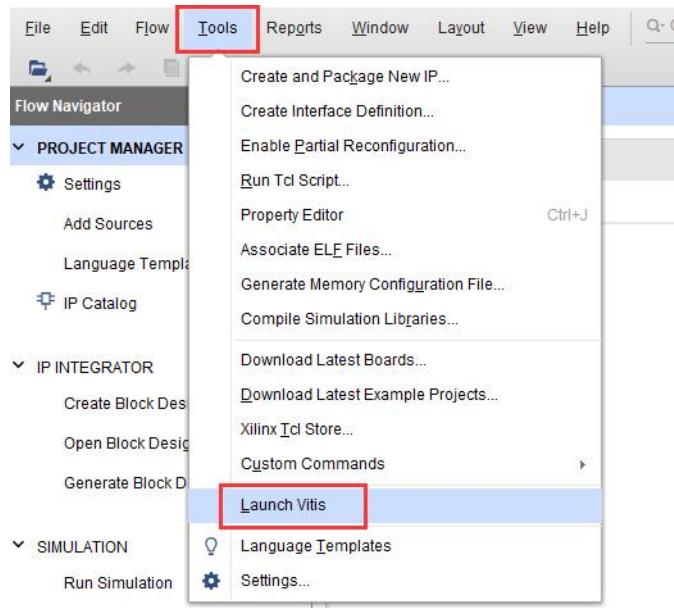
### Part 1.3: Vitis Debugging

#### Part 1.3.1: Create Application Project

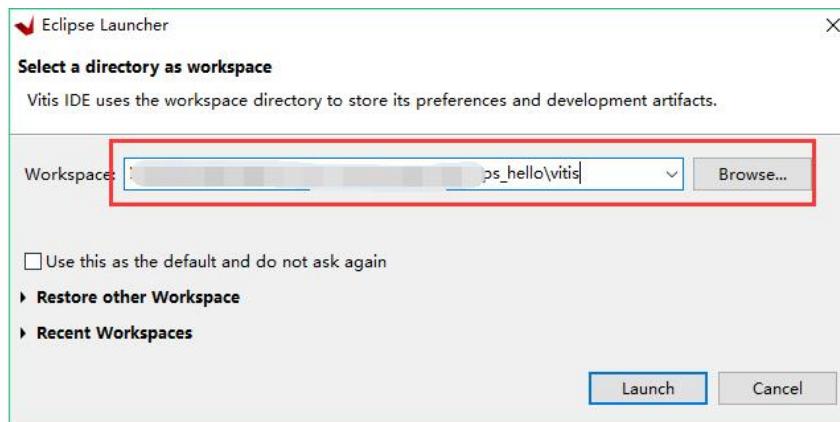
- 1) Create a new folder and copy in the `xx.xsa` file exported by vivado
- 2) Vitis is an independent software, you can double-click the Vitis software to open



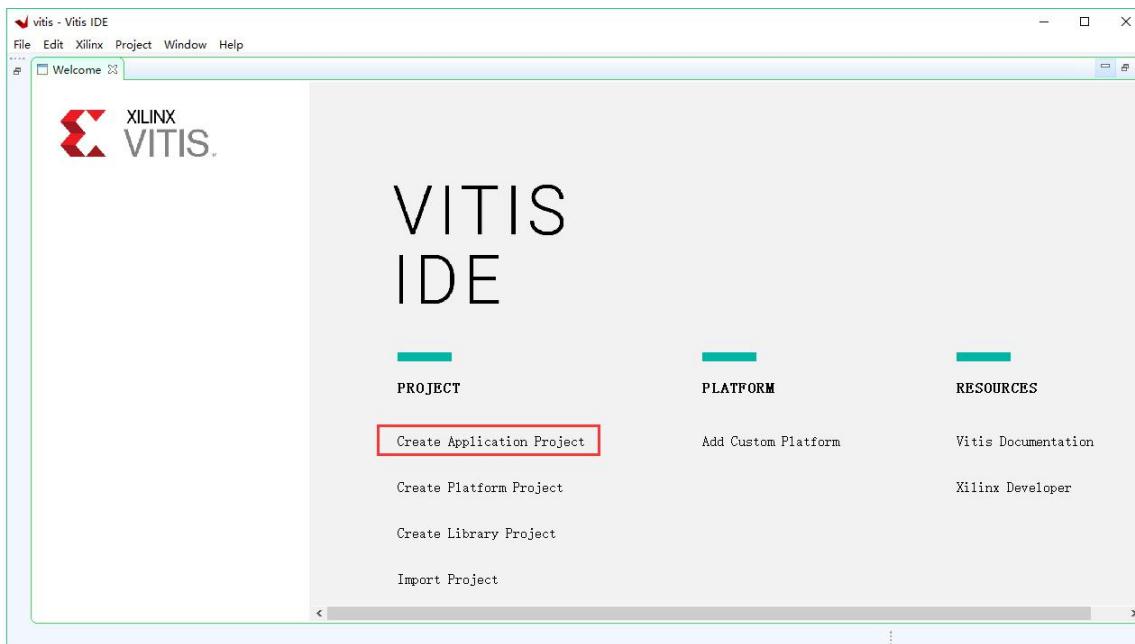
You can also open the Vitis software by selecting Tools → Launch Vitis in the Vivado software



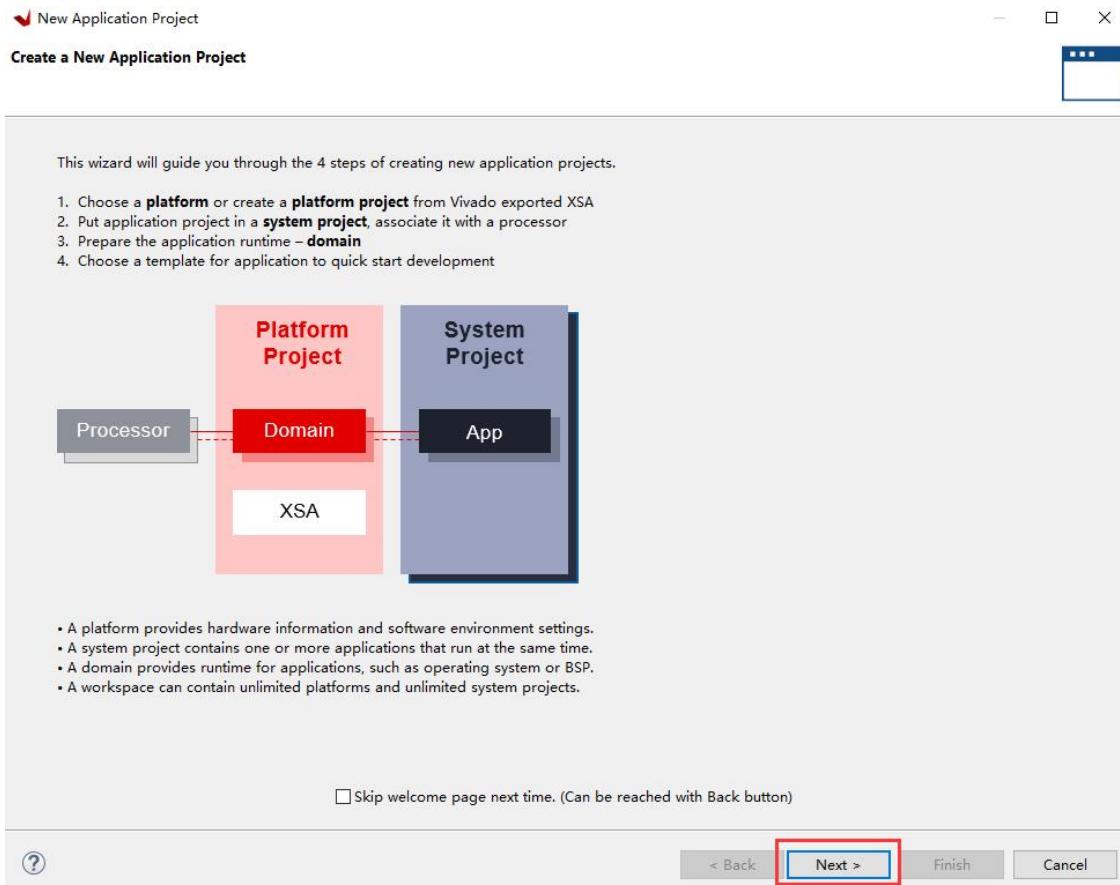
Select the newly created folder and click "Launch"



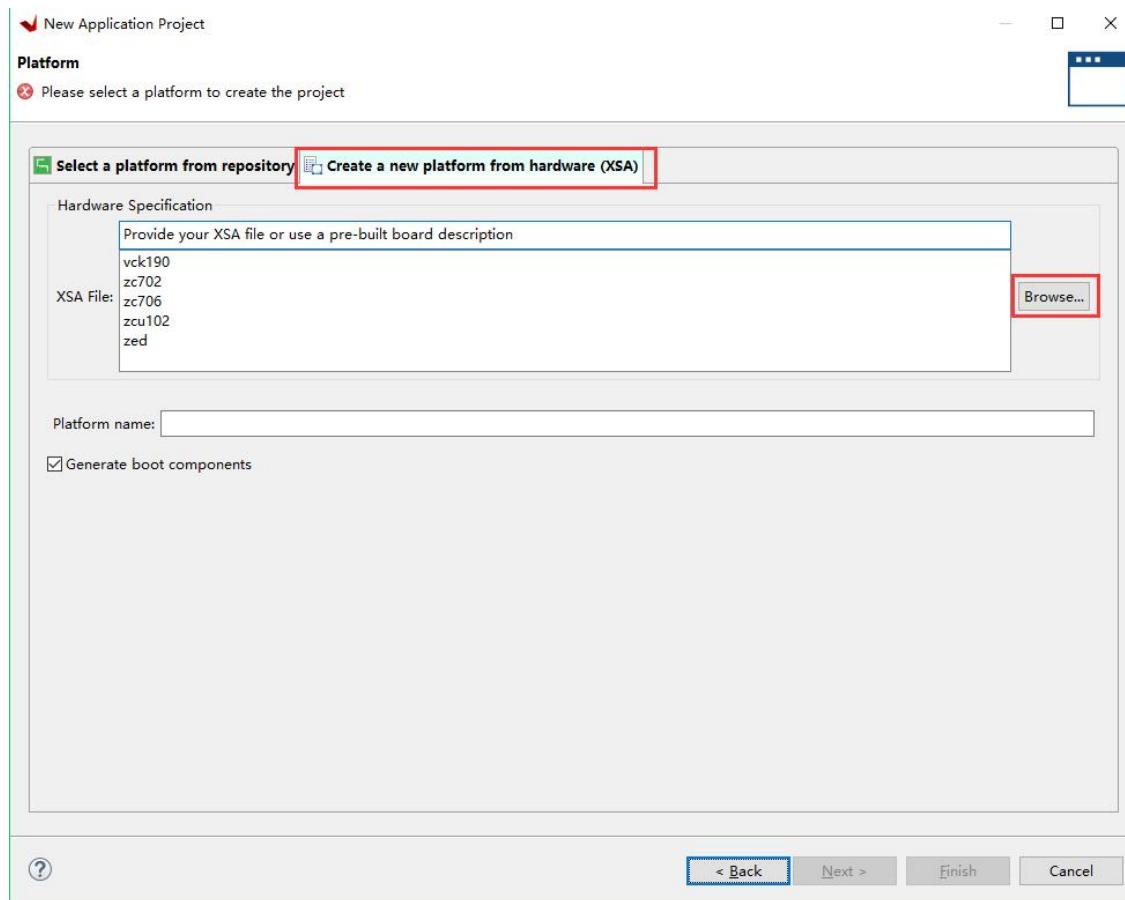
- 3) After starting Vitis, the interface is as follows, click "Create Application Project", this option will generate APP project and Platfrom project, Platform project is similar to the previous version of hardware platform, including hardware support related files and BSP.



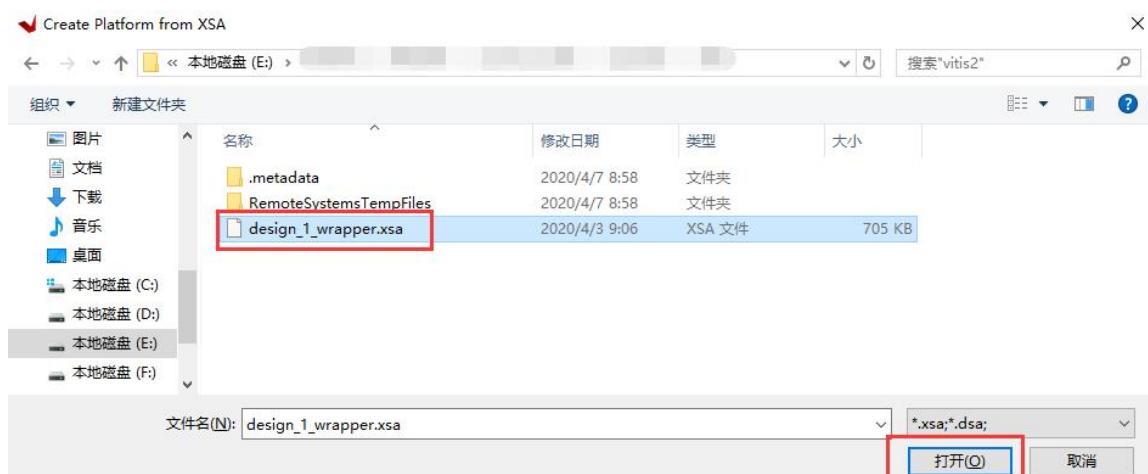
- 4) The first page is the introduction page, skip directly, click Next



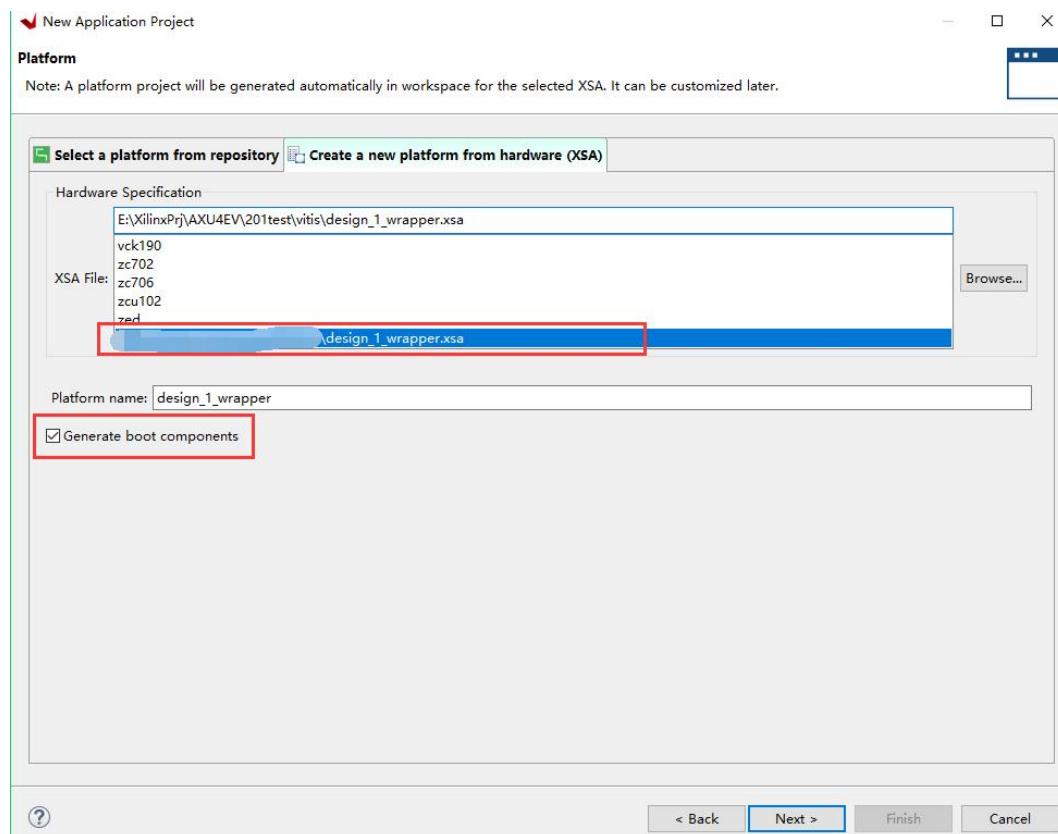
- 5) Select "Create a new platform from hardware(XSA)", select "Browse"



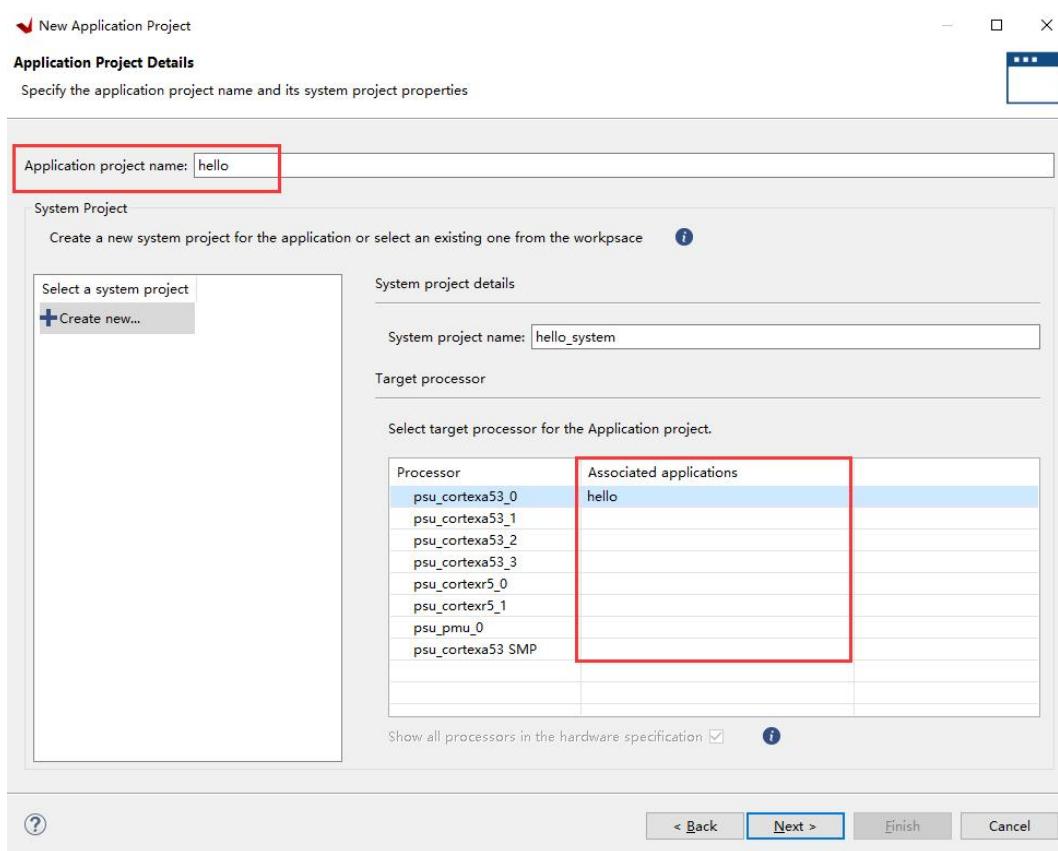
Select the previously generated xsa, click to open



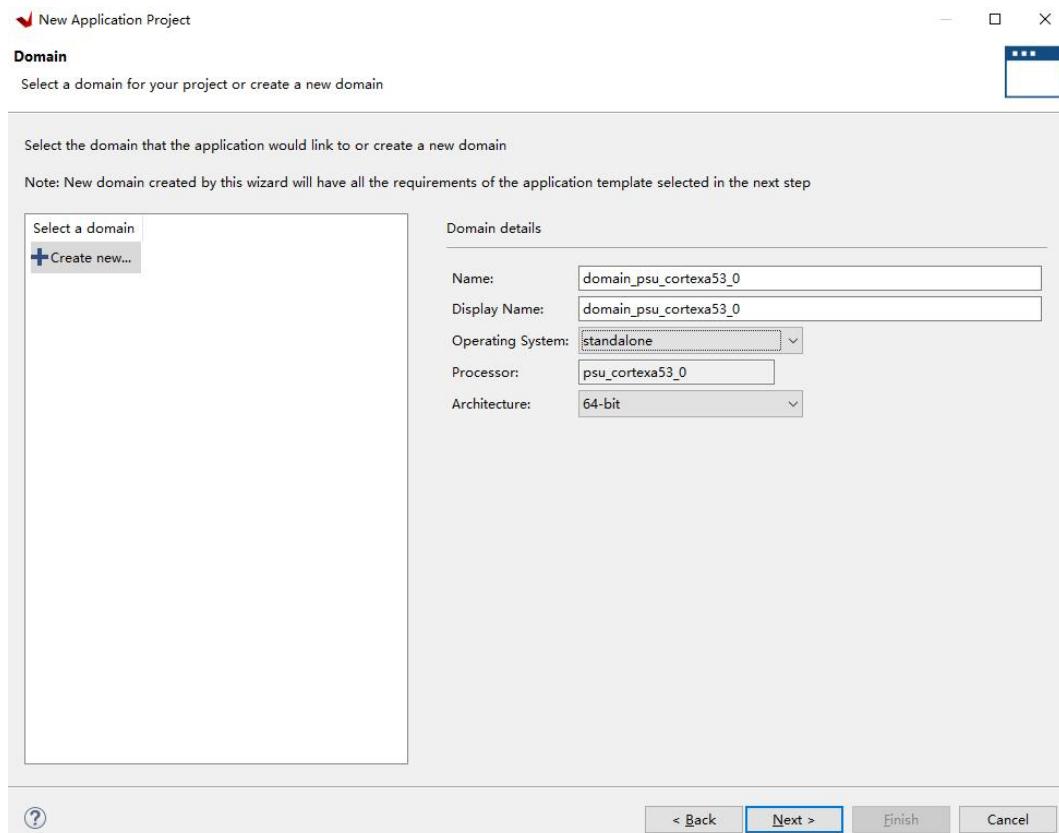
- 6) The Generate boot components option at the bottom, if checked, the software will automatically generate the fsbl project, we generally choose to check it by default



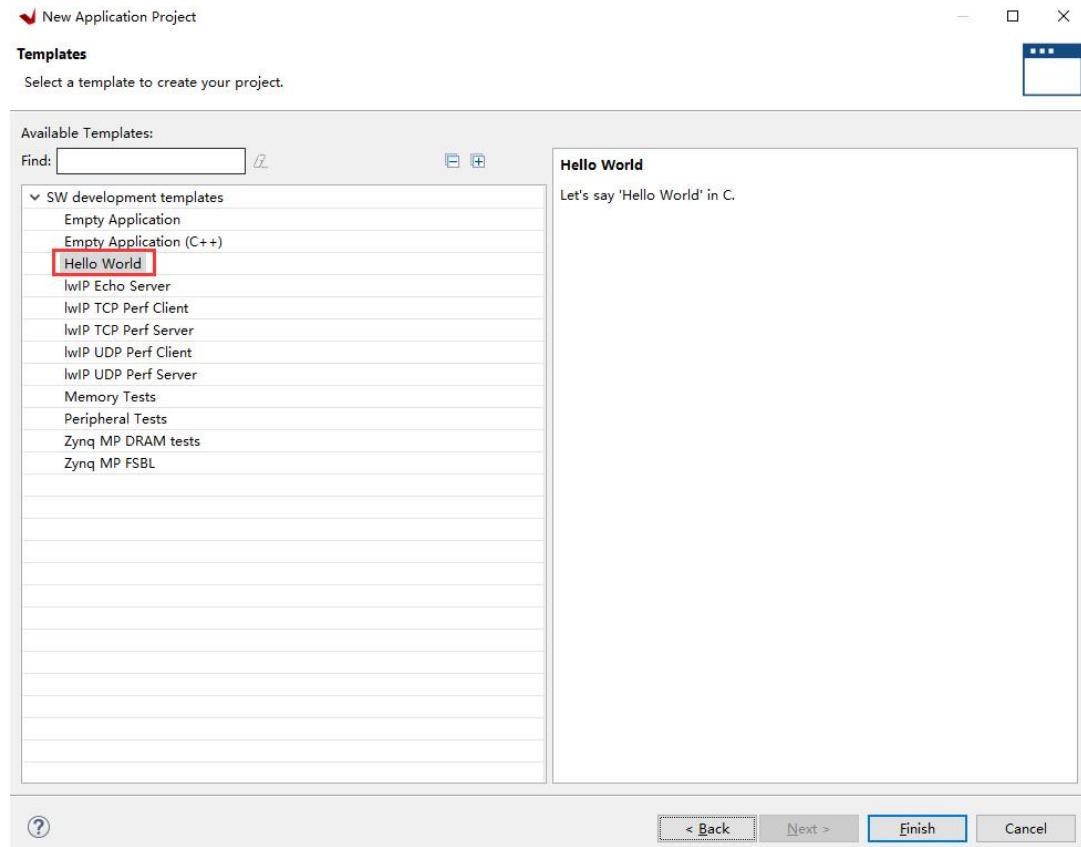
- 7) Fill in the APP project name, click in the box to select the corresponding processor, we keep the default here



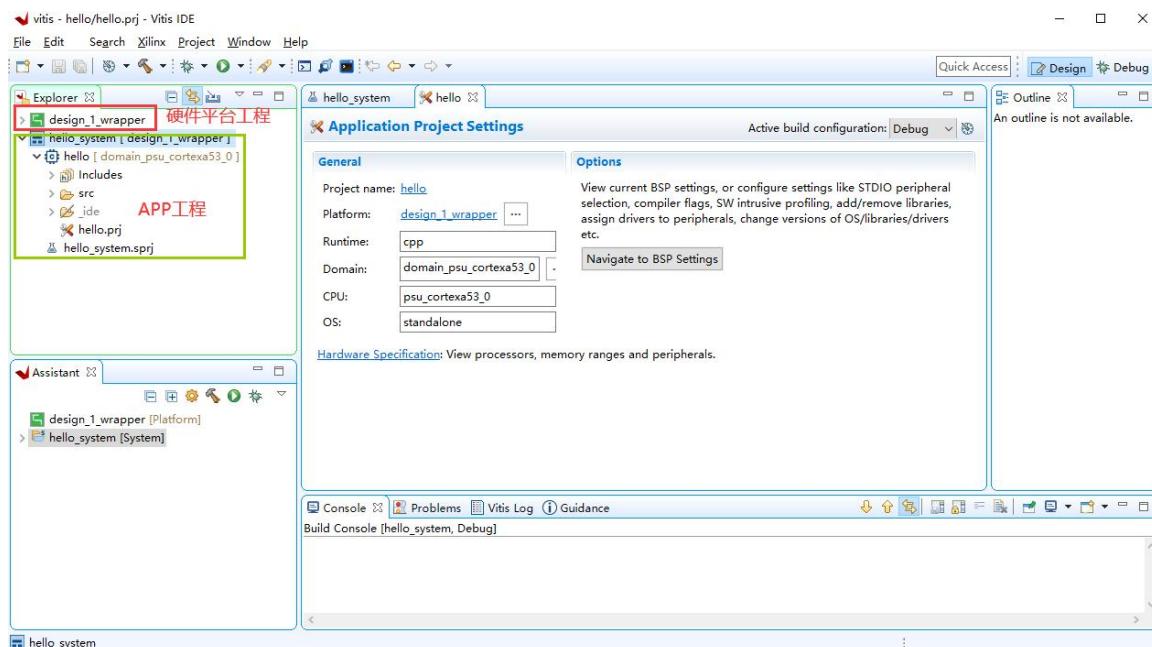
- 8) In this interface, you can modify the domain name, select the operating system, ARM architecture, etc., keep the default here, and select standalone for the operating system, which is bare metal.



- 9) Select the "Hello World" template and click "Finish" to complete

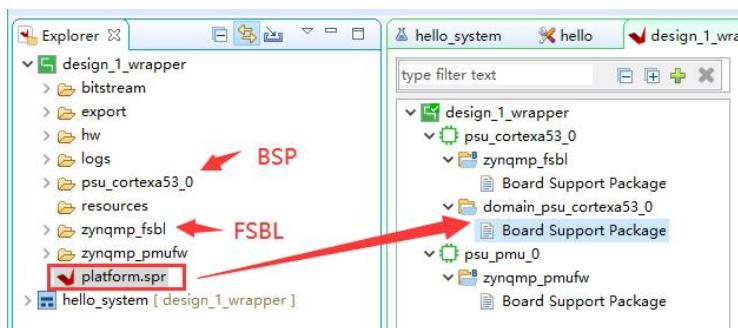


10) After completion, you can see that two projects have been generated, one is the hardware platform project, which is the Platform project mentioned before, and the other is the APP project.

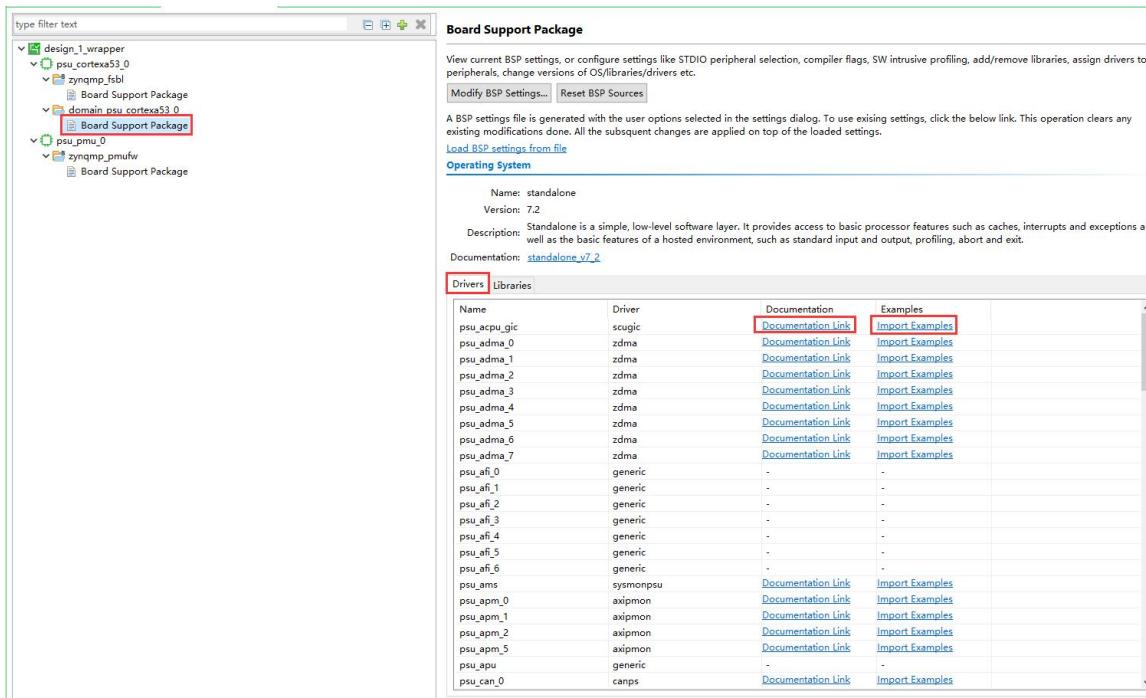


11) After expanding the Platform project, you can see that it contains

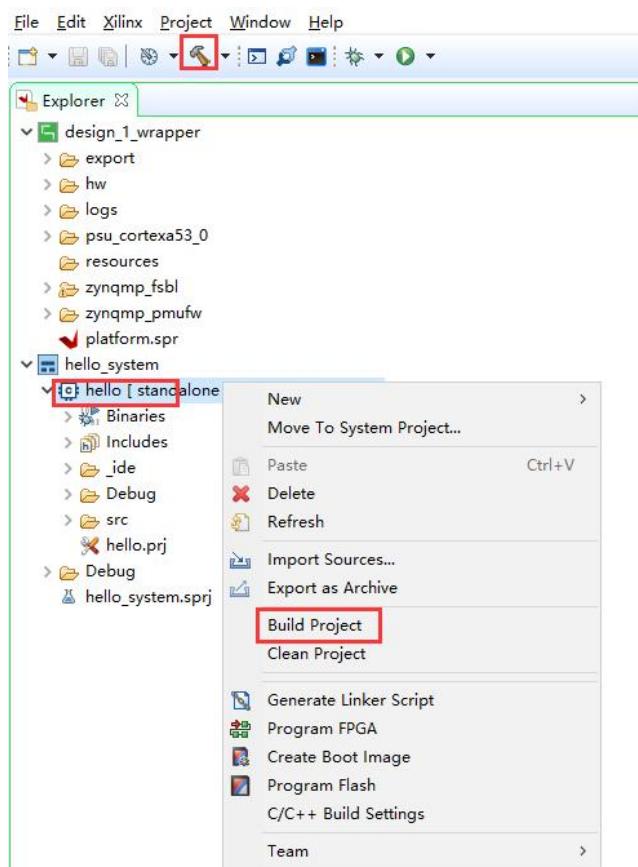
the BSP project and the zynq\_fsbl project (this project is the result of selecting Generate boot components). Double-click platform.spr to see the BSP project generated by Platform, where you can configure the BSP. Software developers are more aware that BSP is also the meaning of Board Support Package, which contains the driver files needed for development and is used for application development. You can see that there are multiple BSPs under the Platform, which is different from the previous SDK software. Among them, zynqmp\_fsbl is the BSP of fsbl, and domain\_psu\_cortexa53\_0 is the BSP of the APP project. You can also add BSP to the Platform, and I will talk about it later in the routine.



12) Click on the BSP to see the peripheral drivers of the project. Documentation is the driver documentation provided by xilinx, and Import Examples is the example project provided by xilinx to speed up learning.



13)Select the APP project, right-click Build Project, or click the "hammer" button in the menu bar to compile the project



14)You can see the compilation process in the Console

```
Build Console [design_1_wrapper]
XSDB Server Channel: tcfchan#1
Building the zynq_fsbl application.
make -C zynq_fsbl_bsp
```

Compile is over, generate elf file

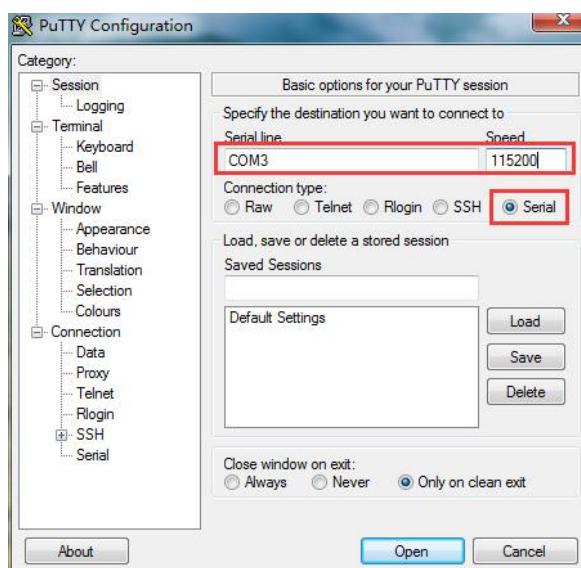
```
Build Console [hello, Debug]
'Finished building: ../src/platform.c'
'
'Building target: hello.elf'
'Invoking: ARM v8 gcc linker'
aarch64-none-elf-gcc -Wl,-T -Wl,.../src/lscript.ld -LE:
'Finished building target: hello.elf'
'

'Invoking: ARM v8 Print Size'
aarch64-none-elf-size hello.elf | tee "hello.elf.size"
    text      data      bss      dec      hex filename
    30308     2048     20616    52972    c000 hello.elf
'Finished building: hello.elf.size'
'

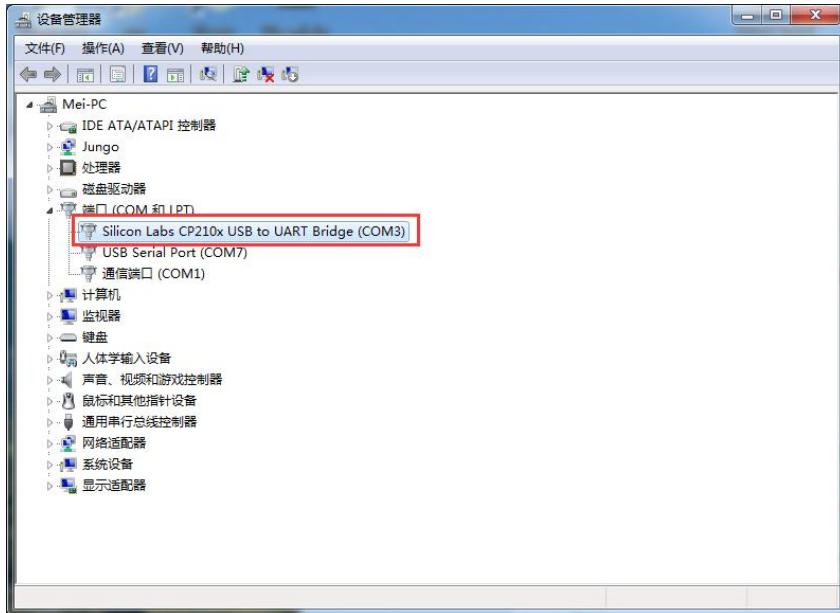
10:51:14 Build Finished (took 1s.121ms)
```

15) Connect the JTAG Cable to the development board, and the UART USB Cable to the PC

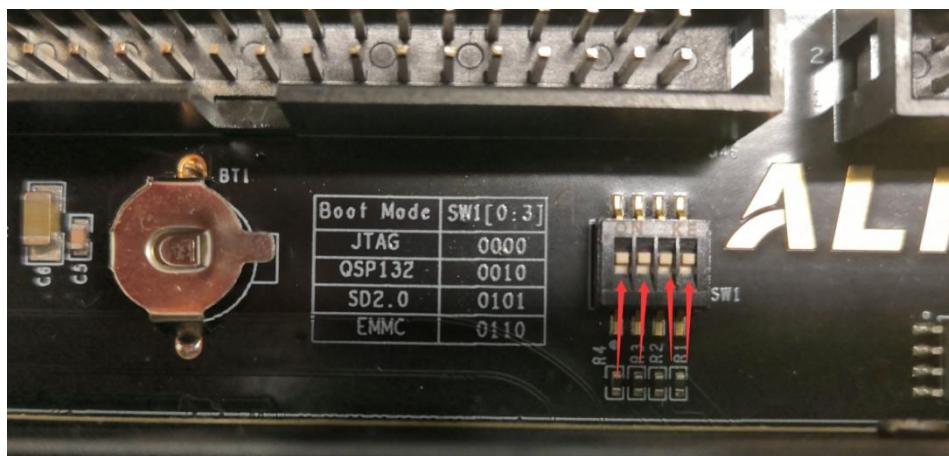
16) Use PuTTY software as a serial port terminal debugging tool, PuTTY is a small software that does not require installation



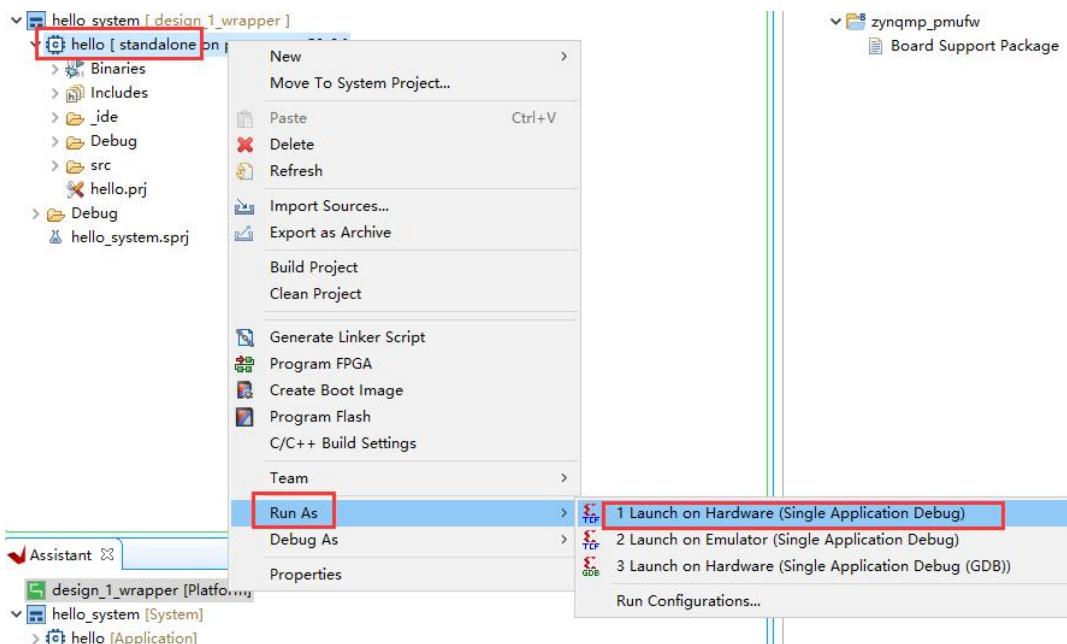
17) Select Serial, fill in COM3 for Serial line, 115200 for Speed, and fill in the COM3 serial port number as shown in the device manager, and click "Open"



18) Before powering on, set the startup mode of the development board to JTAG mode and pull it to the "ON" position



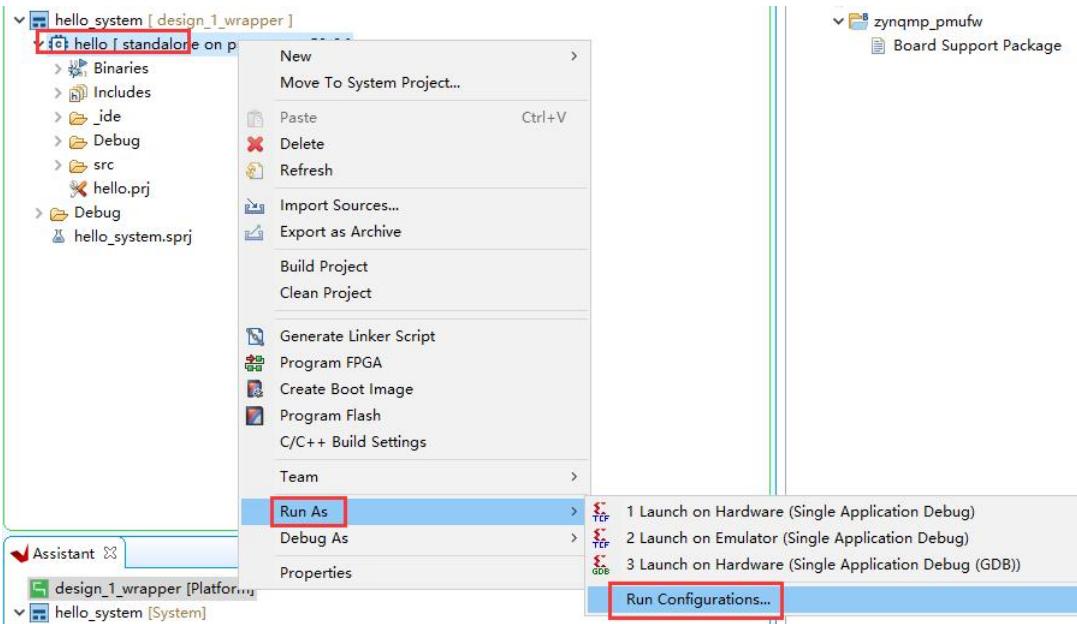
19) Power on the FPGA development board and prepare to run the program. The FPGA development board comes with a program when it leaves the factory. **Here you can select the JTAG mode as the operating mode, and then power on again.** Select "hello", right click, and you can see many options. The "Run as" used in this experiment is to run the program. There are many options in "Run as". Select the first "Launch on Hardware( Single Application Debug)", use system debugging to run the program directly.



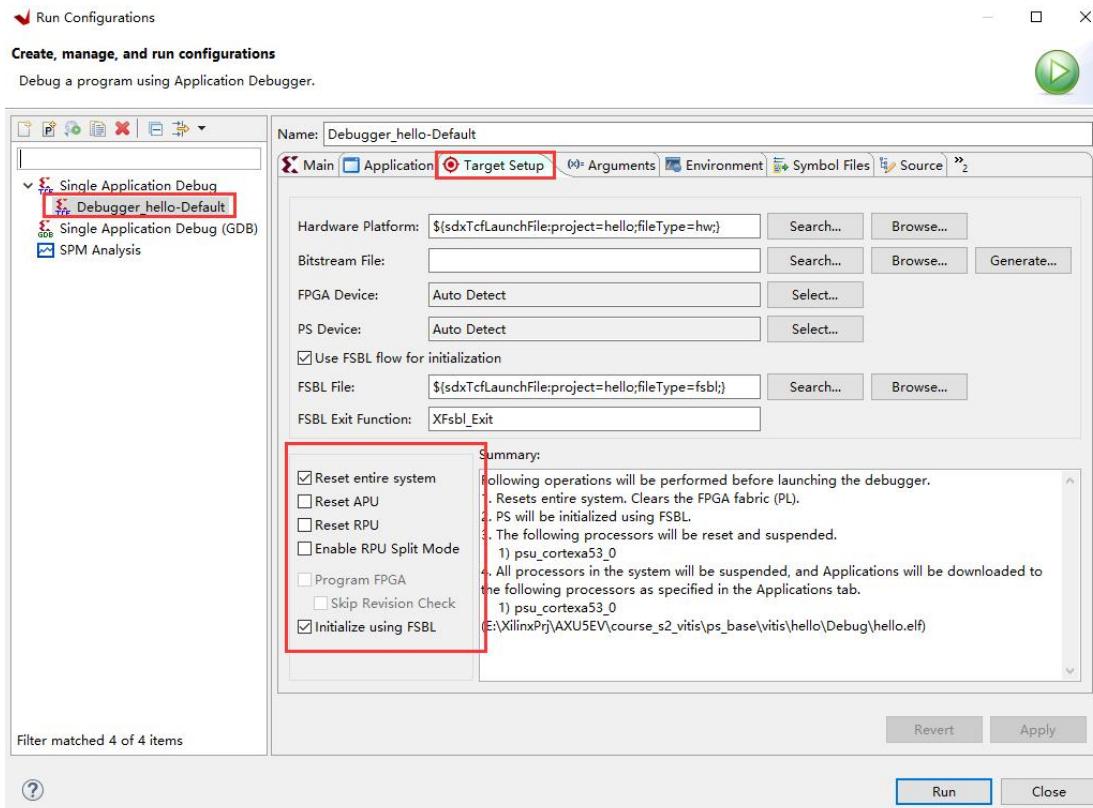
20) Observe the serial port software at this time, you can see the output "Hello World"

```
Hello World  
Successfully ran Hello World application
```

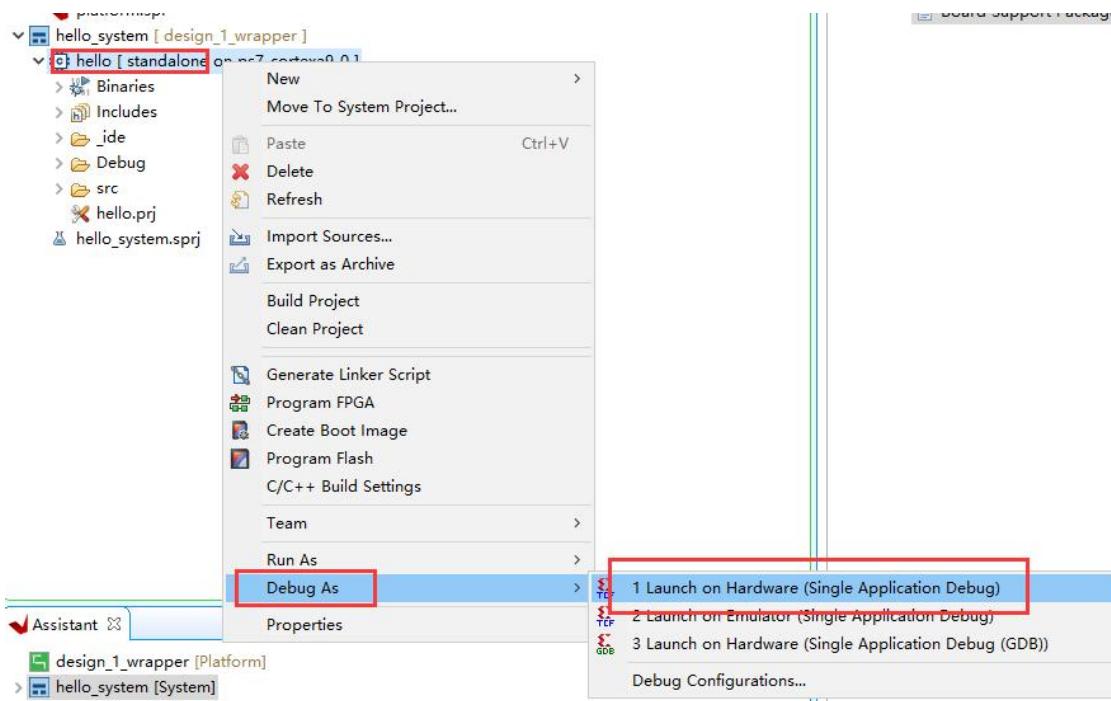
21) In order to ensure the reliable debugging of the system, it is best to right-click "Run As -> Run Configuration..."



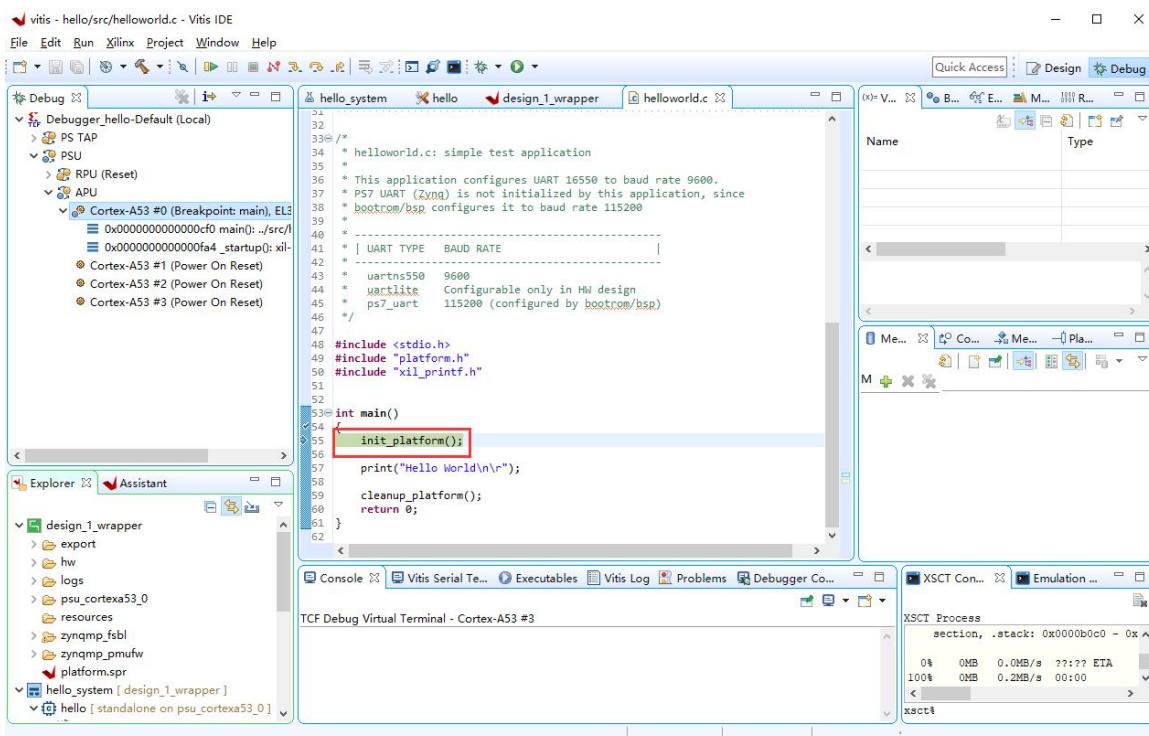
22) We can take a look at the configuration inside, where Reset entire system is selected by default, which is different from the previous SDK software. If there is a PL design in the system, you must also select "Program FPGA".



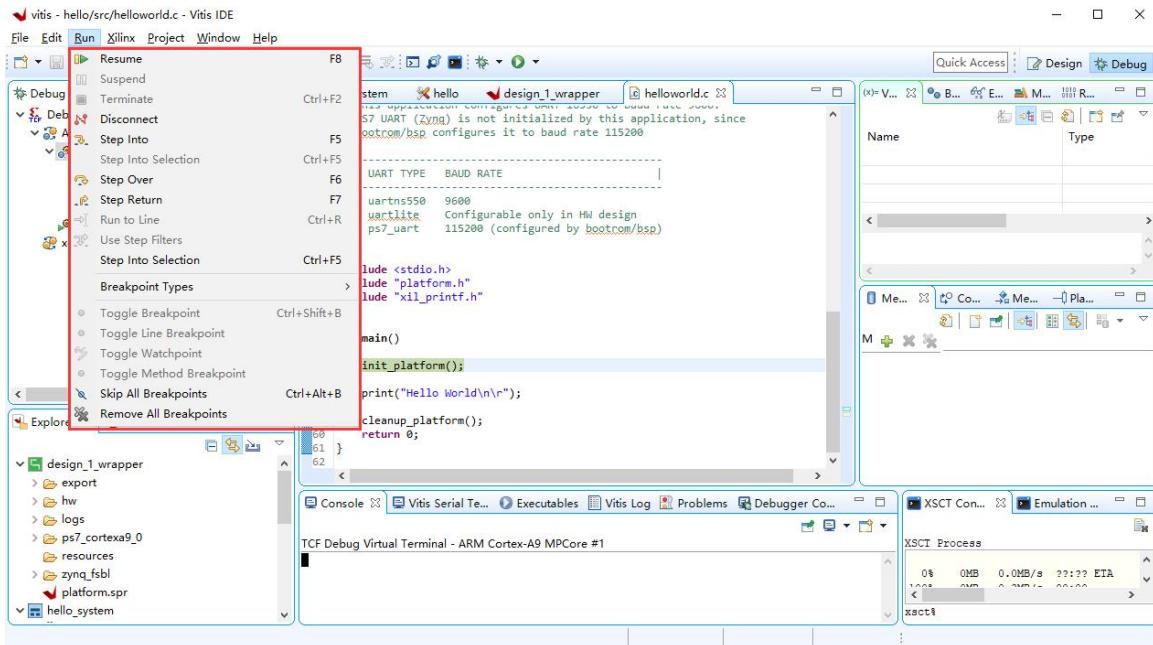
23) In addition to "Run As", you can also "Debug As" so that you can set breakpoints and run in a single step



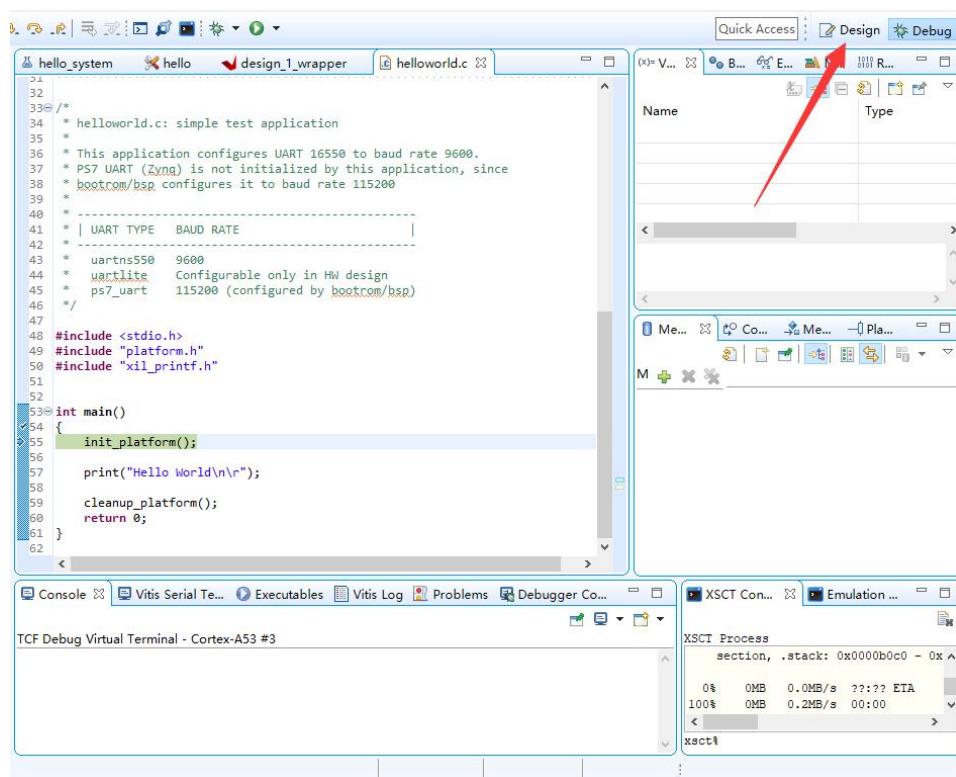
## 24) Enter Debug mode



## 25) Like other C language development IDEs, it can be run step by step, set breakpoints, etc.



26)The IDE mode can be switched in the upper right corner



## Part 1.4: Curing Program

Ordinary FPGAs can be booted from flash, or passively loaded. ZYNQ is started by ARM, including the loading of FPGA programs. ZYNQ MPSoC startup generally involves three steps, which are also

introduced in UG1085:

**Pre-configuration stage:** The pre-loading stage is controlled by the PMU and executes the code in the PMU ROM to set up the system. The PMU handles all reset and wake-up processes.

**Configuration stage:** The next step is to enter the most important step. After BootRom (part of the CSU ROM code) moves FSBL to OCM, the processor starts to execute the FSBL code. FSBL mainly has the following functions:

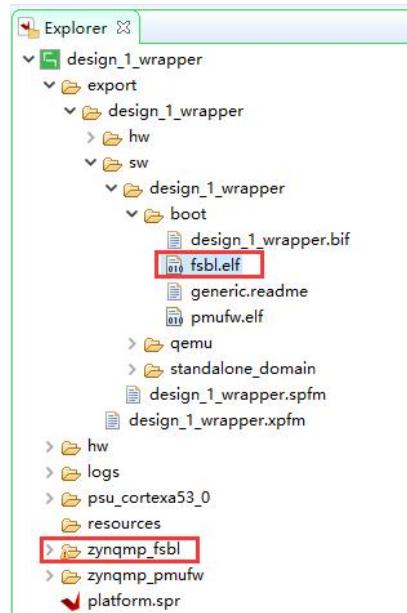
- Initialize PS configuration, MIO, PLL, DDR, QSPI, SD, etc.
- If there is a PL end program, load the PL end bitstream
- Transfer user program to DDR, and jump to execute

**Post-configuration stage:** After the FSBL starts to execute, the CSU ROM code enters the post-configuration stage and is responsible for system intervention response. CSU provides continuous hardware support for verifying file correctness, loading PL through PCAP, storing management security keys, decryption, etc. .

#### **Part 1.4.1: Generate FSBL**

FSBL is a second-level boot program that completes “MIO” allocation, clock, PLL, DDR controller initialization, SD, QSPI controller initialization, finds “bitstream” configuration FPGA through startup mode, then searches for user program to load into DDR, and finally hand over to application carried out.

- 1) Since the Generate boot components option was selected when creating a new project, Platform has imported the fsbl project and generated the corresponding elf file.



- 2) Modifying the debugging macro definition
- FSBL\_DEBUG\_INFO\_VAL can output some status information of FSBL at startup, which is beneficial to debugging, but it will cause the startup time to become longer. save document. You can take a look at the files of many peripherals in fsbl, including psu\_init.c, qspi, sd, etc. You can read the code carefully. Of course, this fsbl template can also be modified, as for how to modify it according to your own needs.

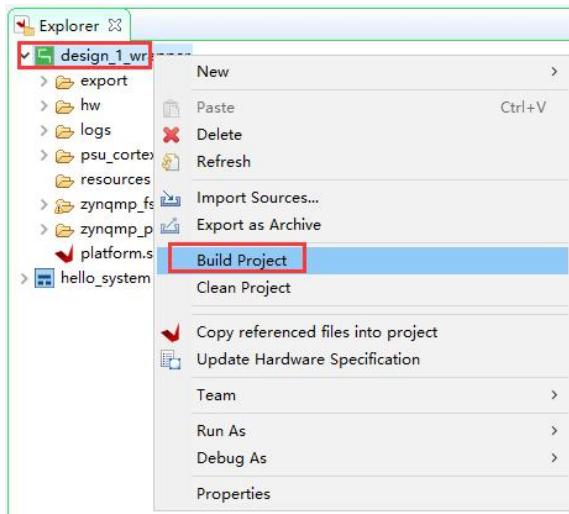
The screenshot shows the Vitis Explorer interface and a code editor side-by-side. The code editor displays the 'xfsbl\_config.h' file with the following content:

```

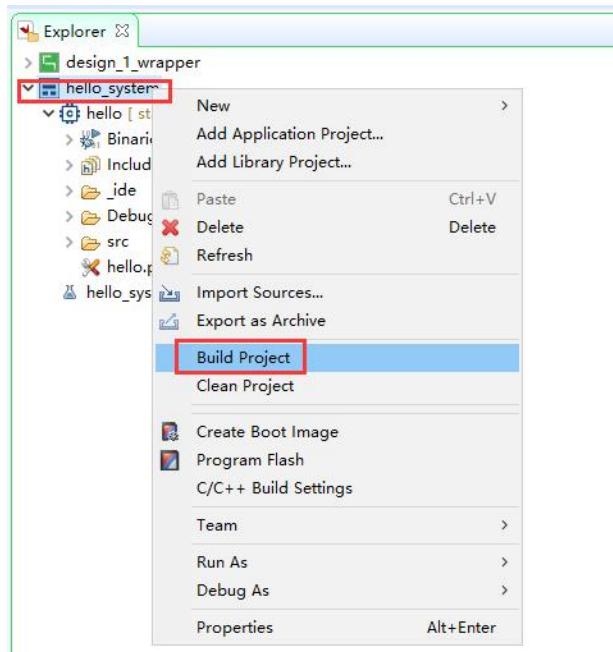
70 * @name FSBL Debug options
71 *
72 * FSBL supports an unconditional print
73 * - FSLB_PRINT Used to print FSBL header and any mandatory prints
74 * - Hence FSLB_DEBUG_VAL should always be 1
75 * Further FSBL by default doesn't have any debug prints enabled. If user
76 * wants to enable the debug prints, they can define the following
77 * options
78 * FSBL supports three types of debug levels.
79 * - FSLB_DEBUG Defining this will print basic information and
80 * - error prints if any
81 * - FSLB_DEBUG_INFO Defining this will have prints enabled with format
82 * - specifiers in addition to the basic information
83 * - FSLB_DEBUG_DETAILED Defining this will print information with
84 * - all data exchanged.
85 */
86 #define FSLB_PRINT_VAL (1U)
87 #define FSLB_DEBUG_VAL (0U)
88 #define FSLB_DEBUG_INFO_VAL (1U)
89 #define FSLB_DEBUG_DETAILED_VAL (0U)
90 */
91 /**
92 * FSBL Debug options
93 */
94
95 #if FSLB_DEBUG_VAL
96 #define FSLB_DEBUG
97#endif
98
99 #if FSLB_DEBUG_INFO_VAL
100 #define FSLB_DEBUG_INFO
101#endif
102
103 #if FSLB_DEBUG_DETAILED_VAL
104 #define FSLB_DEBUG_DETAILED
105#endif

```

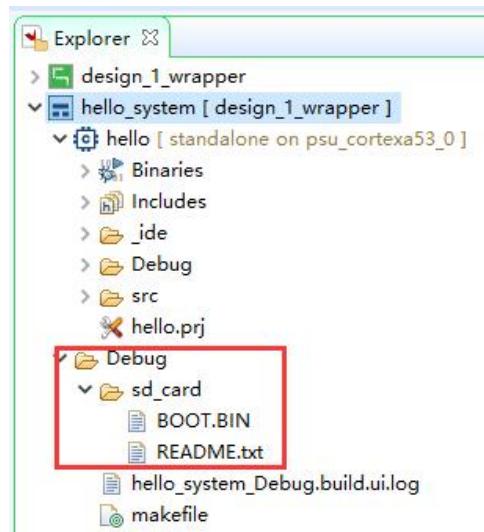
- 3) Build Project



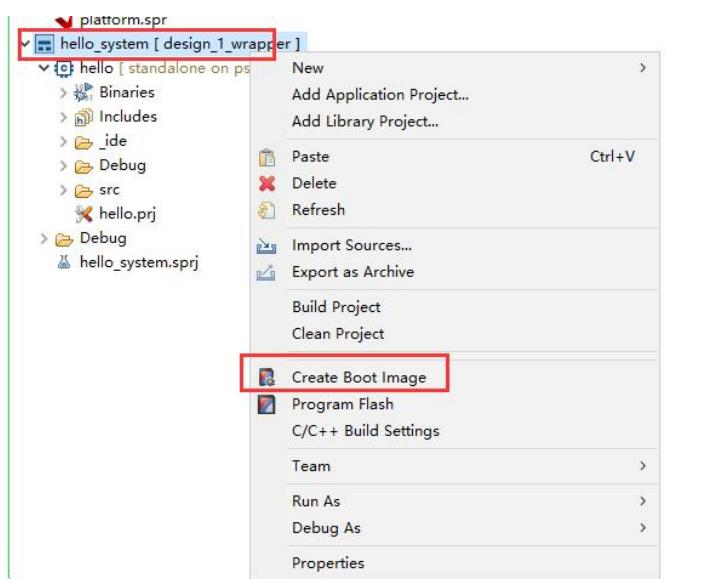
- 4) Next, we can click on the system of the APP project, right-click and select Build project

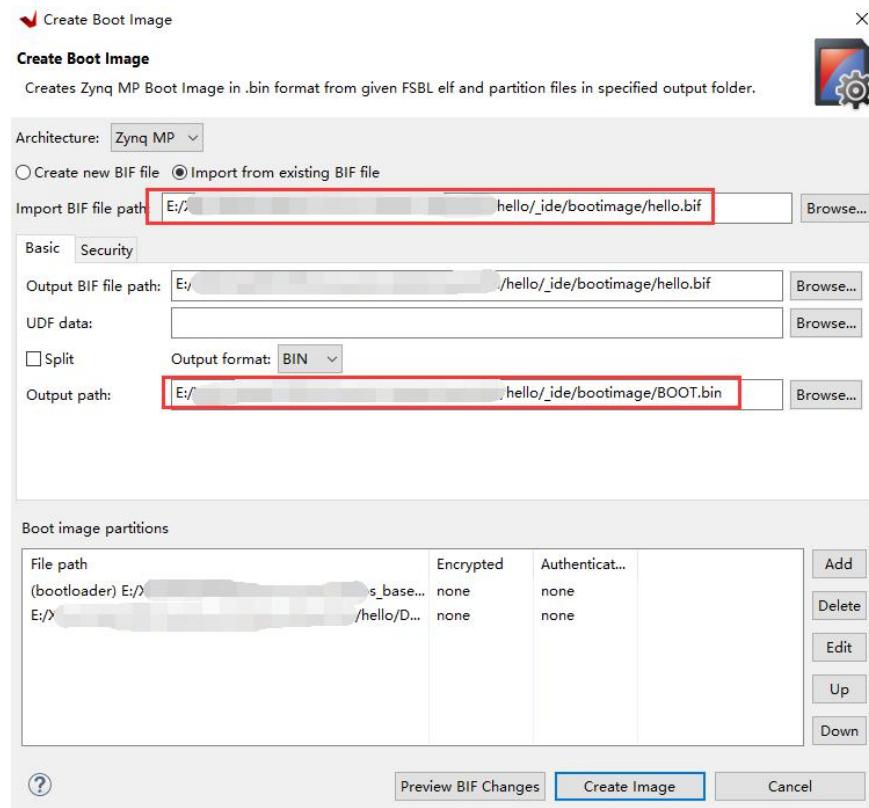


- 5) At this time, there will be an extra Debug folder, and the corresponding BOOT.BIN will be generated

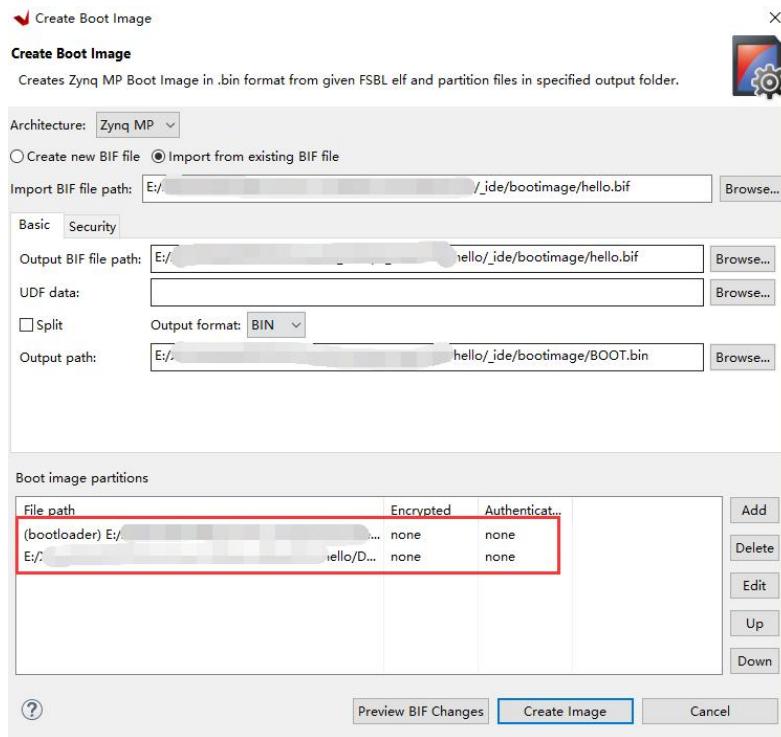


- 6) Another way is to click on the system of the APP project, right-click and select Creat Boot Image. In the pop-up window, you can see the path of the generated BIF file. The BIF file is the configuration file for generating the BOOT file, as well as the path of the generated BOOT.bin file. , The BOOT.bin file is the boot file we need, which can be placed on the SD card to boot, or burned to QSPI Flash.





- 7) There are files to be synthesized in the Boot image partitions list. The first file must be the bootloader file, which is the fsbl.elf file generated above, and the second file is the FPGA configuration file bitstream. In this experiment, there is no FPGA bitstream. No need to add, the third one is the application, in this experiment it is hello.elf, because there is no bitstream, only the bootloader and application are added in this experiment. Click Create Image to generate



8) The BOOT.bin file can be found in the generated directory



### Part 1.4.2: SD card Startup Test

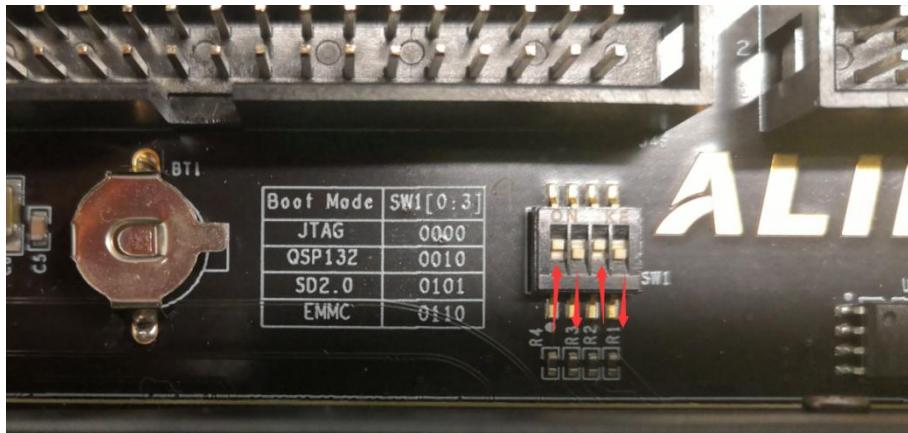
1) Format SD card, can only be formatted as FAT32, other formats cannot be started



2) Put in the “BOOT.bin” file and put it in the root directory



- 3) Insert the SD card into the SD card slot of the development board
- 4) Adjust the boot mode to SD card boot

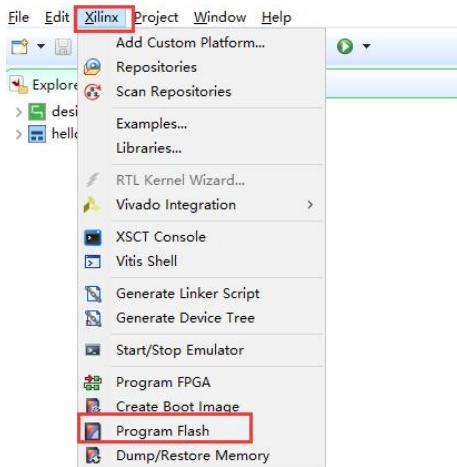


- 5) Open the serial port software, power on and start, you can see the print information, the red box is the FSBL startup information, the yellow arrow part is the executed application “helloworld”

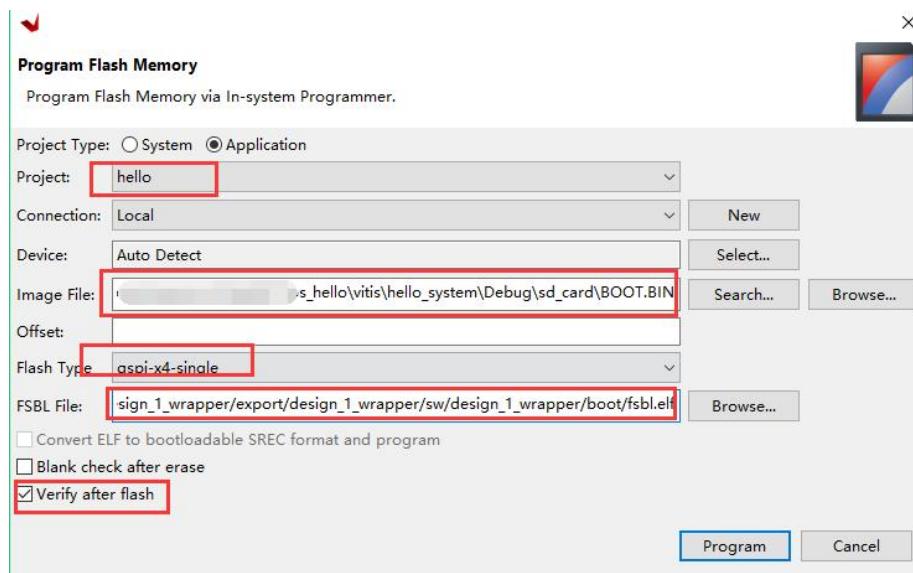
```
Xilinx Zynq MP First Stage Boot Loader
Release 2020.1 Jul 8 2020 - 14:22:13
Reset Mode : System Reset
Platform: Silicon (4.0), Cluster ID 0x00000000
Running on A53-0 (64-bit) Processor, Device Name: XCZU4EV
Processor Initialization Done
===== In Stage 2 =====
SD1 Boot Mode
SD: rc= 0
File name is l:/BOOT.BIN
Multiboot Reg : 0x0
Image Header Table Offset 0x8C0
*****Image Header Table Details*****
Boot Gen Ver: 0x1020000
No of Partitions: 0x2
Partition Header Address: 0x440
Partition Present Device: 0x0
Initialization Success
===== In Stage 3, Partition No:1 =====
UnEncrypted data Length: 0x2412
Data word offset: 0x2412
Total Data word length: 0x2412
Destination Load Address: 0x0
Execution Address: 0x0
Data word offset: 0x8290
Partition Attributes: 0x116
Partition 1 Load Success
All Partitions Loaded
===== In Stage 4 =====
PMU-FW is not running, certain applications may not be supported.
Protection configuration applied
Running Cpu Handoff address: 0x0, Exec State: 0
Exit from FSBL
Hello World
Successfully ran Hello World application
```

### Part 1.4.3: QSPI Startup Test

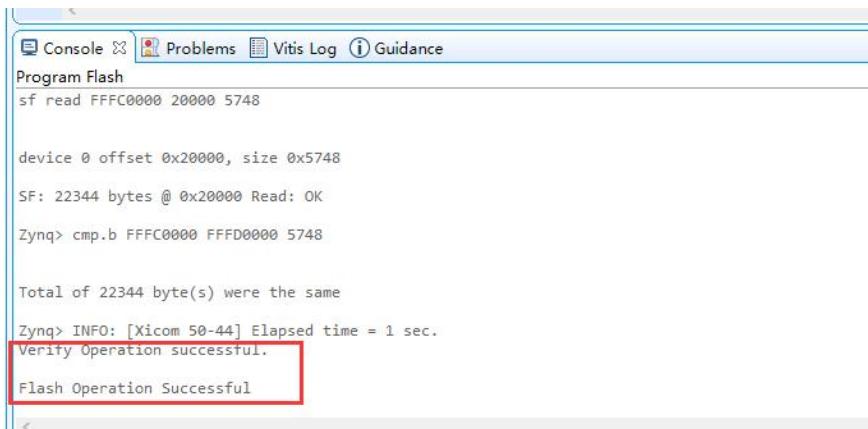
- 1) In the Vitis menu “Xilinx-> Program Flash”



- 2) The “Hardware Platform” selects the latest, the “Image File” file selects the “BOOT.bin” to be flashed, and the “FSBL file” selects the “fsbl.elf”. Select “Verify after flash” to verify the “flash” after programming is complete.



- 3) Click “Program” and wait for programming to complete



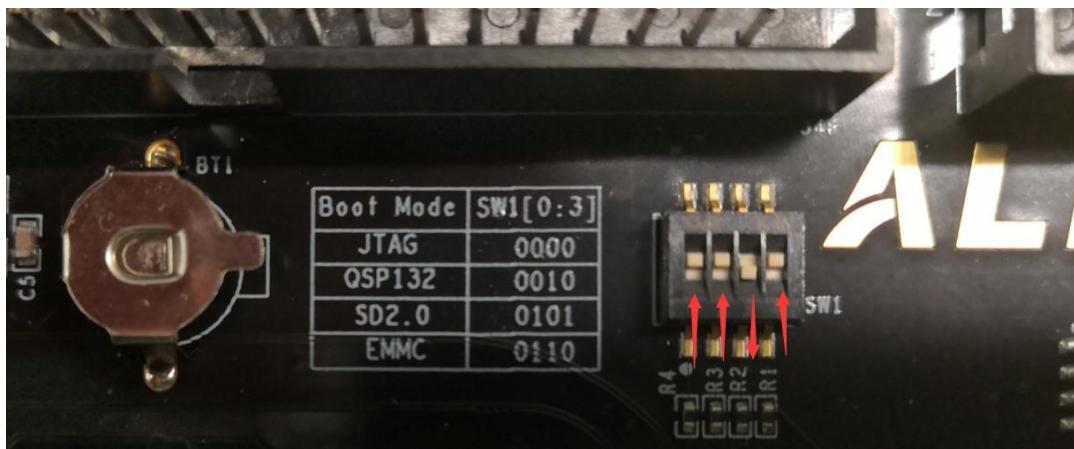
```
Console ✘ Problems Vitis Log Guidance
Program Flash
sf read FFFC0000 20000 5748

device 0 offset 0x20000, size 0x5748
SF: 22344 bytes @ 0x20000 Read: OK
Zynq> cmp.b FFFC0000 FFFD0000 5748

Total of 22344 byte(s) were the same
Zynq> INFO: [Xicom 50-44] Elapsed time = 1 sec.
Verity Operation successful.

Flash Operation Successful
```

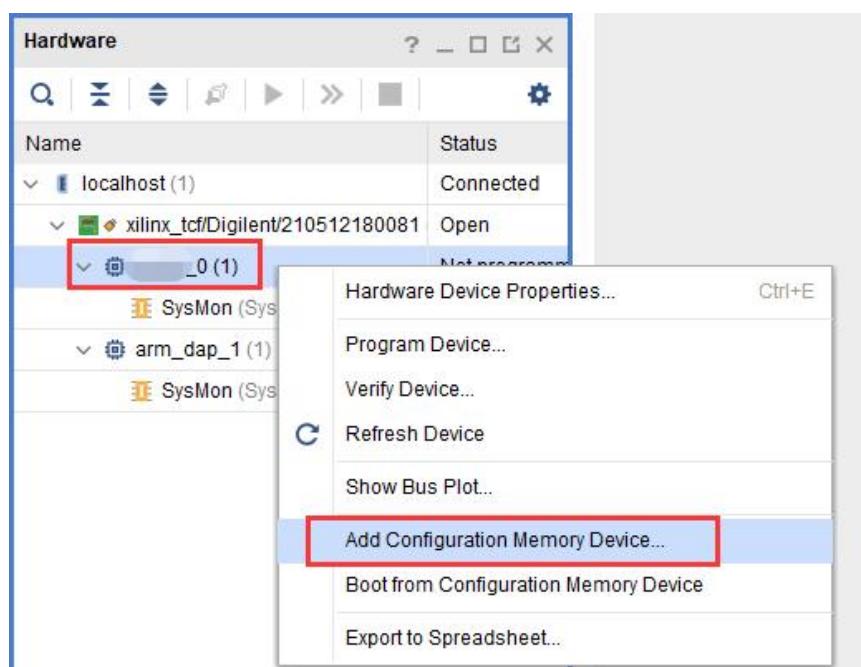
- 4) Set the startup mode to QSPI and start again, You can see the same startup effect as SD in the serial port software.



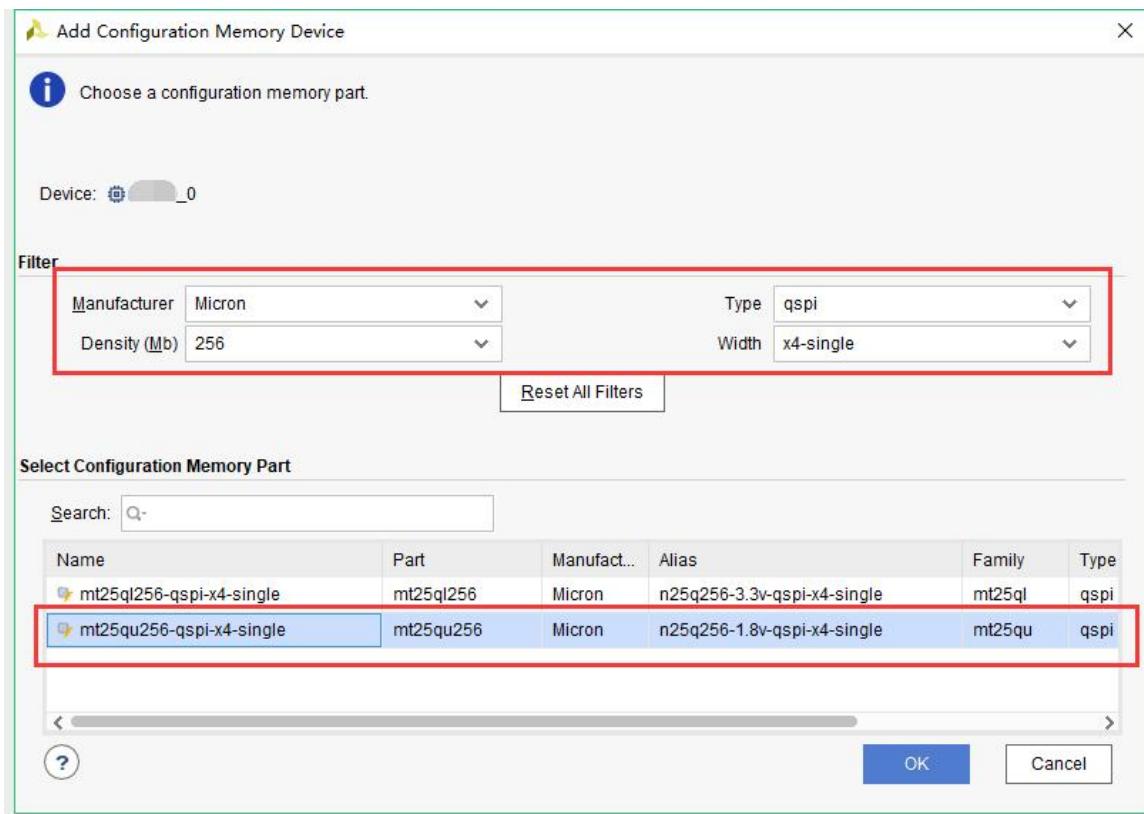
```
Xilinx Zynq MP First Stage Boot Loader
Release 2020.1 Jul 8 2020 - 14:22:13
Reset Mode : System Reset
Platform: Silicon (4.0), Cluster ID 0x80000000
Running on A53-0 (64-bit) Processor, Device Name: XCZU4EV
Processor Initialization Done
==== In Stage 2 ====
QSPI 32 bit Boot Mode
QSPI is in single flash connection
QSPI is using 4 bit bus
FlashID=0x20 0xBB 0x19
MICRON 256M Bits
Multiboot Reg : 0x0
QSPI Reading Src 0x0, Dest FFFF0040, Length E0
.Image Header Table Offset 0x8C0
QSPI Reading Src 0x8C0, Dest FFFDD0C8, Length 40
*****Image Header Table Details*****
Boot Gen Ver: 0x1020000
No of Partitions: 0x2
Partition Header Address: 0x440
Partition Present Device: 0x0
QSPI Reading Src 0x1100, Dest FFFDD108, Length 40
.QSPI Reading Src 0x1140, Dest FFFDD148, Length 40
.Initialization Success
===== In Stage 3, Partition No:1 =====
UnEncrypted Data Length: 0x2412
Data word offset: 0x2412
Total Data word length: 0x2412
Destination Load Address: 0x0
Execution Address: 0x0
Data word offset: 0x8290
Partition Attributes: 0x116
QSPI Reading Src 0x20A40, Dest 0, Length 9048
.Partition 1 Load Success
All Partitions Loaded
===== In Stage 4 =====
PMU-FW is not running, certain applications may not be supported.
Protection configuration applied
Running Cpu Handoff address: 0x0, Exec State: 0
Exit from FSBI
Hello World
Successfully ran Hello World application
```

#### Part 1.4.4: Programming QSPI Under Vivado

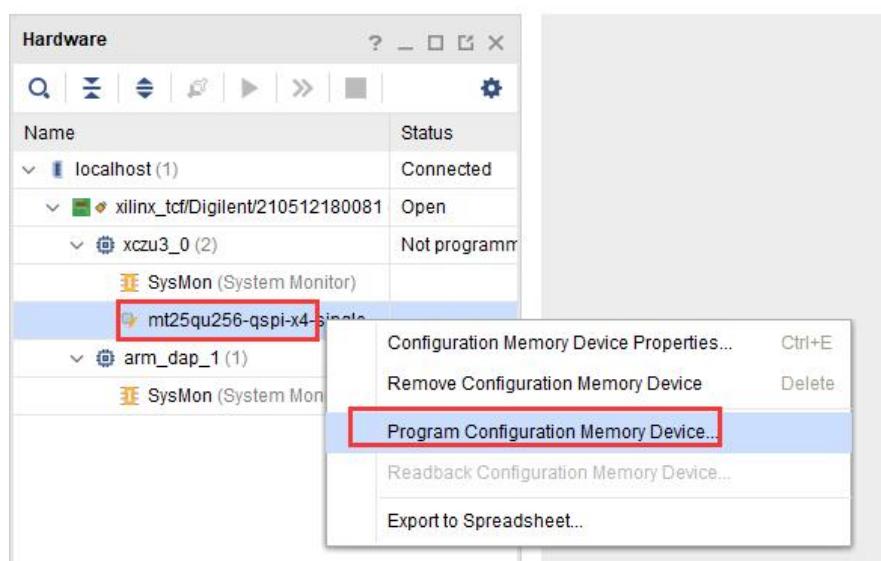
- 1) Select the device under “HARDWARE MANGER”, right-click “Add Configuration Memory Device”



- 2) Choose to try “Micron”, select “qspi” for the type, select “x4-single” for the width, and select “256” for Density. At this time, “w25q128” appears, select the red frame model, and the FPGA development board uses “w25q256”, but it does not affect the burning.

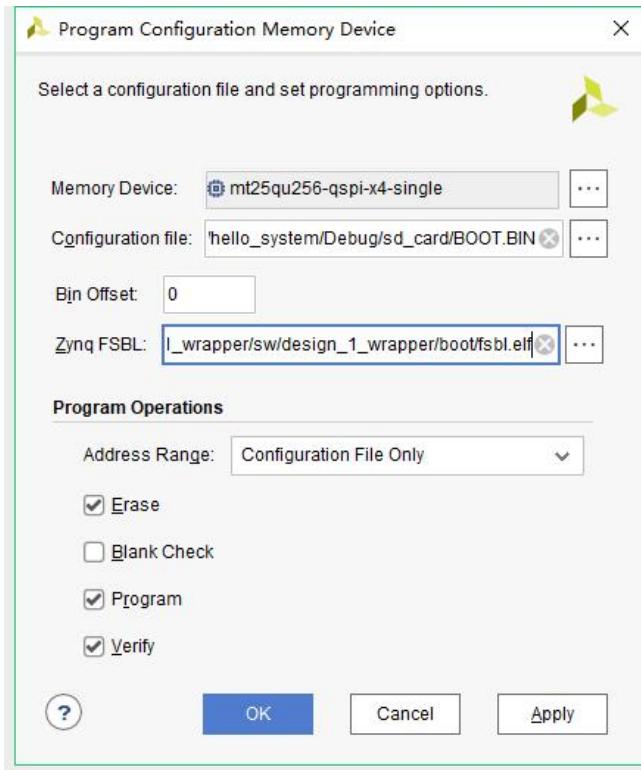


- 3) Right-click and select the programming file



- 4) Select the file to be flashed and the “fsbl” file, then you can flash. If

the flash is not in JTAG boot mode, the software will give a warning, so it is recommended to set to JTAG boot mode when flashing QSPI



#### Part 1.4.6: Quickly Flash QSPI Using Batch Files

- 1) Create a new “program\_qspi.txt” text file, change the extension to “bat”, and fill in the following content, where “E:\XilinxVitis\Vitis\2020.1\bin\program\_flash” is our tool path, modify it appropriately according to the installation path, “-f” is the file to be flashed, and “-fsbl” is the fsbl file (ALINX specific files) to be flashed. “-blank\_check –verify” is the check option.

```
call E:\XilinxVitis\Vitis\2020.1\bin\program_flash -f BOOT.bin      -offset 0 -flash_type
qspi-x4-single -fsbl fsbl.elf -verify
pause
```

- 2) Put together the “BOOT.bin”, “fsbl”, and “bat” files to be burned

BOOT.bin	2020/1/14 9:22	BIN 文件	150 KB
fsbl.elf	2020/1/14 8:59	ELF 文件	545 KB
program_qspi.bat	2020/1/14 9:55	Windows 批处理...	1 KB

- 3) Plug in the JTAG cable and power on. Double-click the “bat” file to flash.

```
C:\Windows\system32\cmd.exe
F:\xilinx_project
image_download>call F:\Xilinx_Vitis\Vitis\2019.2\bin\program_flash -f BOOT.bin -offset 0 -flash_type qspi-x4-single -fsbl fsbl.elf -verify
***** Xilinx Program Flash
***** Program Flash v2019.2 (64-bit)
**** SW Build 2708376 on Wed Nov 6 21:40:23 MST 2019
** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.

Connected to hw_server @ TCP:localhost:3121
Available targets and devices:
Target 0 : jsn-JTAG-HS1-210512180081
Device 0: jsn-JTAG-HS1-210512180081-4ba00477-0

Retrieving Flash info...

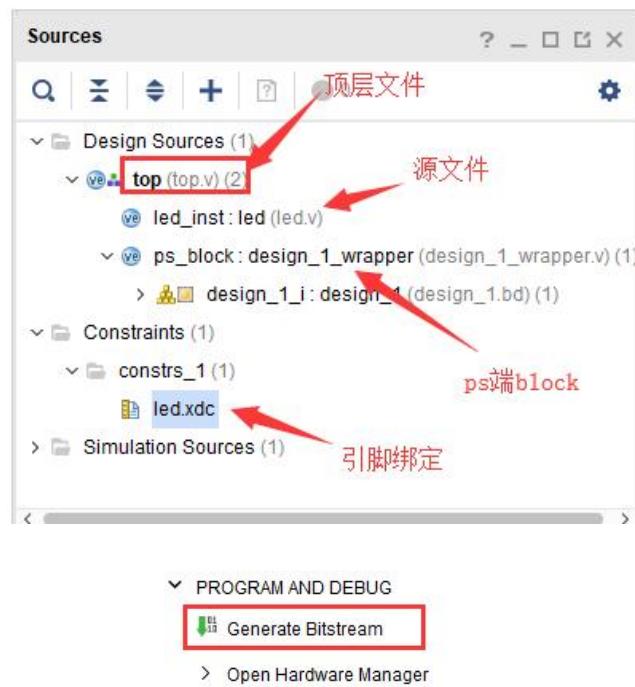
Initialization done, programming the memory
Using default mini u-boot image file - F:/Xilinx_Vitis/Vitis/2019.2/data\xicom\cfgmem\uboot\zynq_qspi_x4_single.bin
===== mrd->addr=0xF800025C, data=0x00000001 =====
BOOT_MODE_REG = 0x00000001
WARNING: [Xicom 50-100] The current boot mode is QSPI.
If flash programming fails, configure device for JTAG boot mode and try again.
===== mrd->addr=0xF8007080, data=0x30800100 =====
===== mrd->addr=0xF8000B18, data=0x80000000 =====
Downloading FSBL...
Running FSBL...
Finished running FSBL.
===== mrd->addr=0xF8000110, data=0x0000FA220 =====
READ: ARM PLL CFG (0xF8000110) = 0x0000FA220
```

## Part 1.5: Q&A

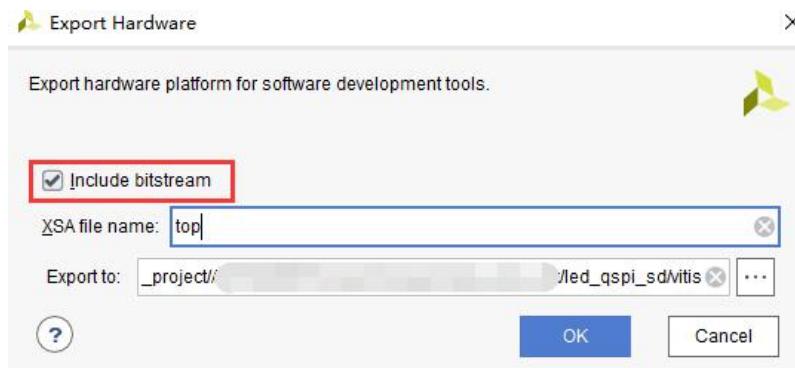
### Part 1.5.1: Only PL side Logic Solidification

Many people will ask, if there is only the logic on the PL side, how to fix the program on the PS side? There is no problem in FPGA without ARM, but for ZYNQ, the cooperation of PS side is required to solidify the program. So how to fix the program for the previous "PL" Hello World "LED experiment"?

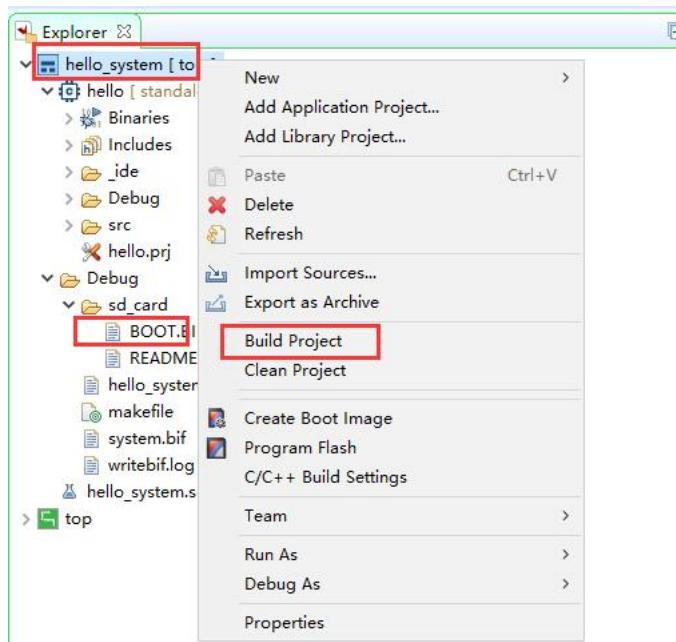
- 1) According to the PS side of this chapter, add the ZYNQ core and configure it. The easiest way is to add the verilog source file of the LED experiment on the basis of the project in this chapter and instantiate it to form a system, and the bitstream needs to be generated.



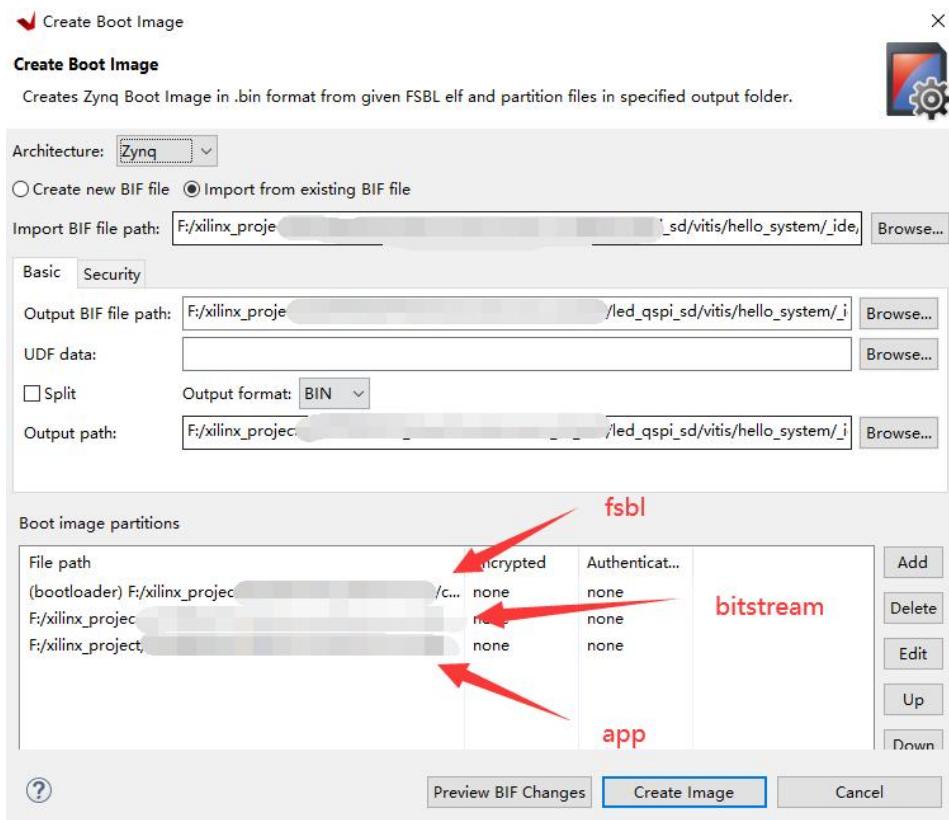
- 2) After generating the bitstream, export the hardware and select "include bitstream"



- 3) When generating BOOT.BIN, you still need an app project hello, just to generate "BOOT.BIN". By default, right-click Build Project in the system to generate BOOT.BIN containing bitstream.



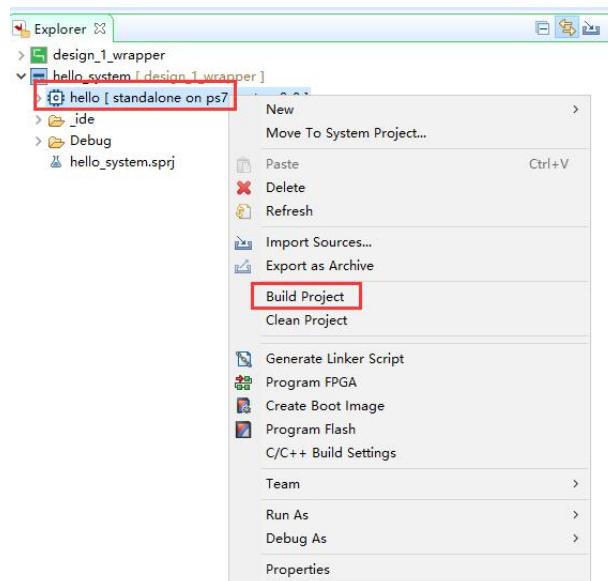
- 4) Open the Create Boot Image interface and you can see, the file order of the “BoolImage Partitions” is “fsbl”, “bitstream”, “app”, pay attention not to reverse the order. Using the BOOT.BIN generated in this way, you can test and start according to the previous startup method.



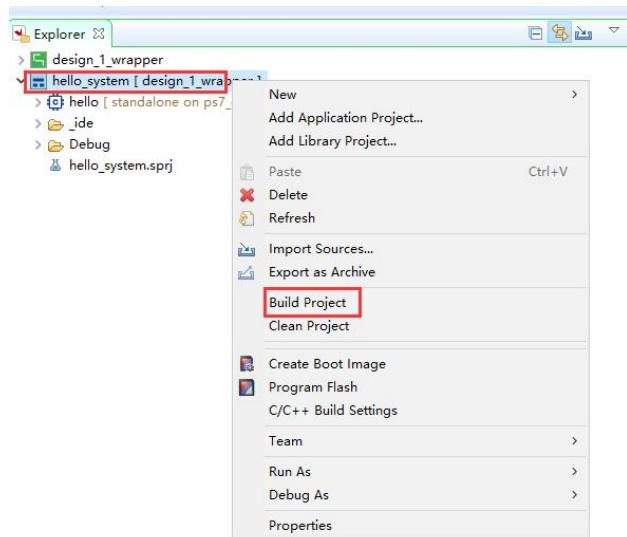
- 5) In the “course\_s2” folder, a project named “led\_qspi\_sd” is provided for your reference.

## Part 1.6: Use skills to Share

When frequently modifying source files and compiling, it is best to select APP project for Build Project. In this case, only elf files will be generated.



If you want to generate the BOOT.BIN file, you can choose system to compile. In this case, both elf and BOOT.BIN will be generated. I suffered a lot when I first used it. Every time I compiled, I chose system. You have to wait for the BOOT.BIN to be generated, a waste of time, everyone can pay attention to it.



## Part 1.7: Experimental Summary

This chapter introduces the classic process of ZYNQ development from the perspective of both FPGA engineers and software engineers. The main job of FPGA engineers is to build a hardware platform, provide hardware description files “hdf” to software engineers, and software engineers develop applications on this basis. This chapter is a simple example that introduces the collaborative work of FPGA and software engineers. The follow-up will also involve joint debugging between PS and PL, which is more complicated and is the core part of ZYNQ development.

It also introduces FSBL, boot file creation, SD card boot method, QSPI download and boot method, and Vivado download BOOT.BIN method. There is no FPGA load file in this chapter. We will introduce adding FPGA load file to make BOOT.BIN later .

Subsequent projects will be subject to the configuration in this chapter. The basic configuration of ZYNQ will not be described later.

## Part 2: PS RTC Interrupt Experiment

The vivado project directory is "ps\_hello/vivado"

The vitis project directory is "ps\_rtc/vitis"

### Part 2.1: RTC Introduction

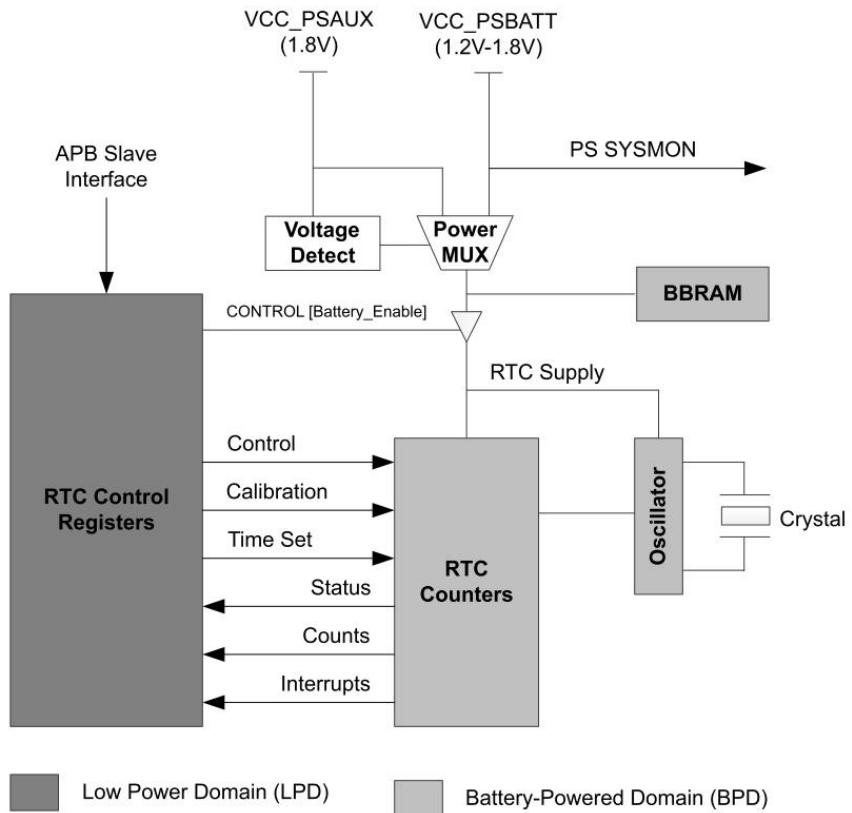
The real-time clock unit provides an accurate time reference for the system and application software. In order to meet the need for high precision, the real-time clock also includes a calibration circuit to compensate for temperature and voltage fluctuations. RTC is powered by VCC-PSAUX or VCC-PSBATT power supply. When auxiliary power is available, the RTC uses it to keep the counter active. When the auxiliary power supply is not available, the RTC automatically switches to the VCC PSBATT power supply. RTC functions are as follows:

- 1) When the system is powered off, the unit will automatically switch to battery power supply to realize the uninterrupted operation of the clock
- 2) Support alarm setting and periodic interrupt setting
- 3) Calibrate the circuit to ensure accurate time
- 4) Three counters

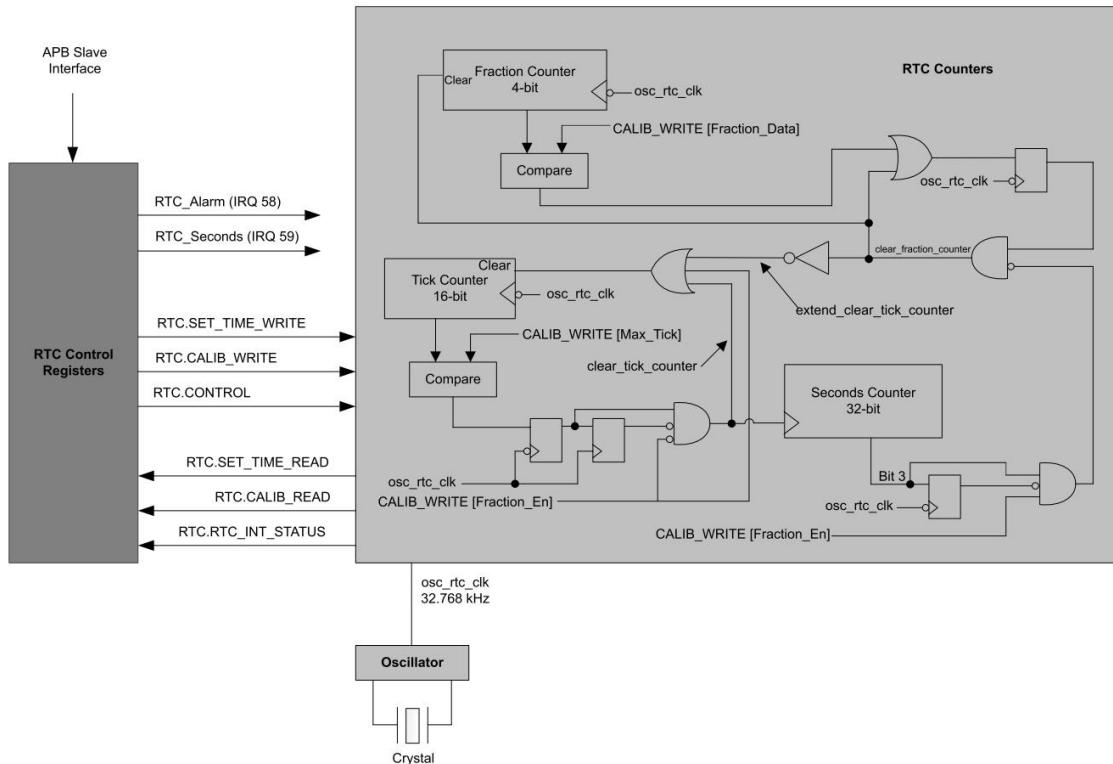
Time seconds counter, 32 bits, can count about 136 years

32 KHz reference clock counter, which means 1 second count

4-digit fraction counter for calibration



RTC Controller Block Diagram

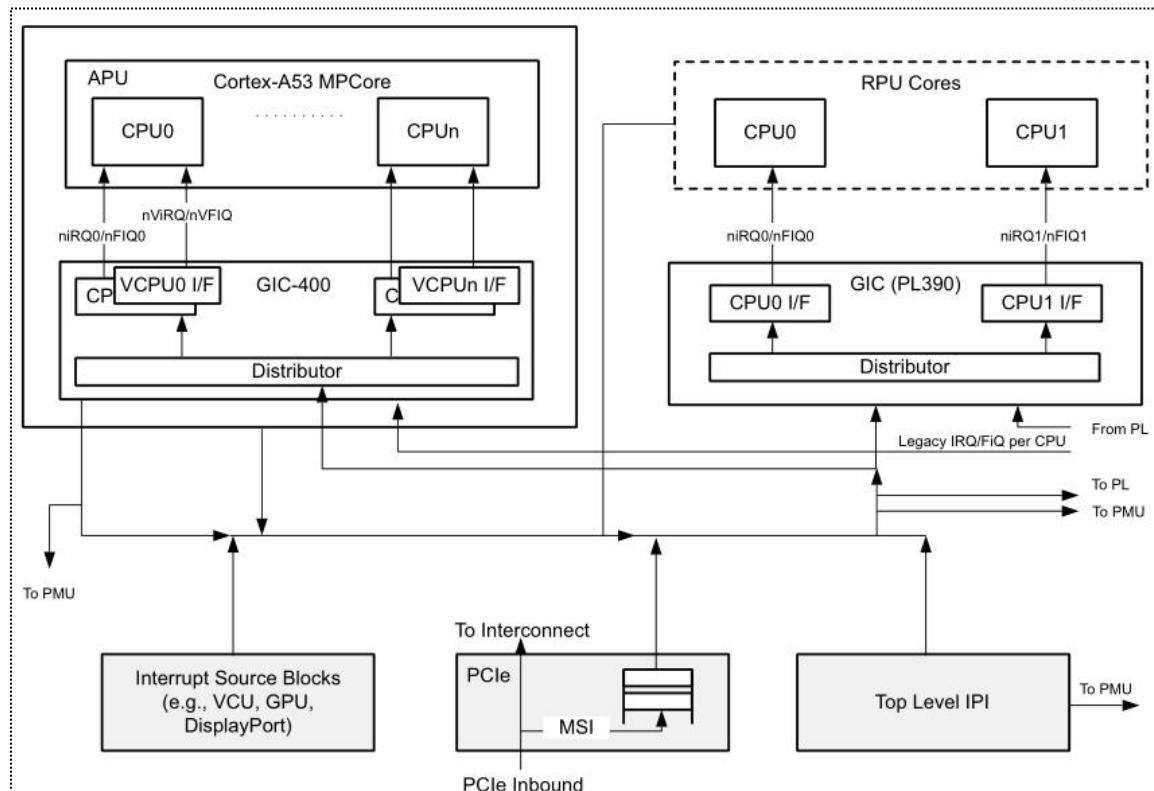


RTC Block Diagram

## Part 2.2: Interrupt Introduction

- 1) ARM cortex-A series processors provide 4 pins for soc to realize the transfer of external interrupts. They are: nIRQ, nFIQ, nVIRQ, nVFIQ. In the arm system, there will be multiple peripherals, all of which may generate interrupts and send them to the core. Therefore, an interrupt controller is needed as an intermediate bridge to collect all the interrupt signals of the soc, and then arbitrate to select the appropriate (high priority) Interrupt, and then send to the CPU, waiting for the CPU to process.
- 2) The bridge in the middle here is the famous gic (general interrupt controller) launched by the arm company. gic is actually an architecture, and the version has gone through gicv1, gicv2, gicv3, and gicv4.
- 3) Ultrascale+ interrupt block diagram is as follows

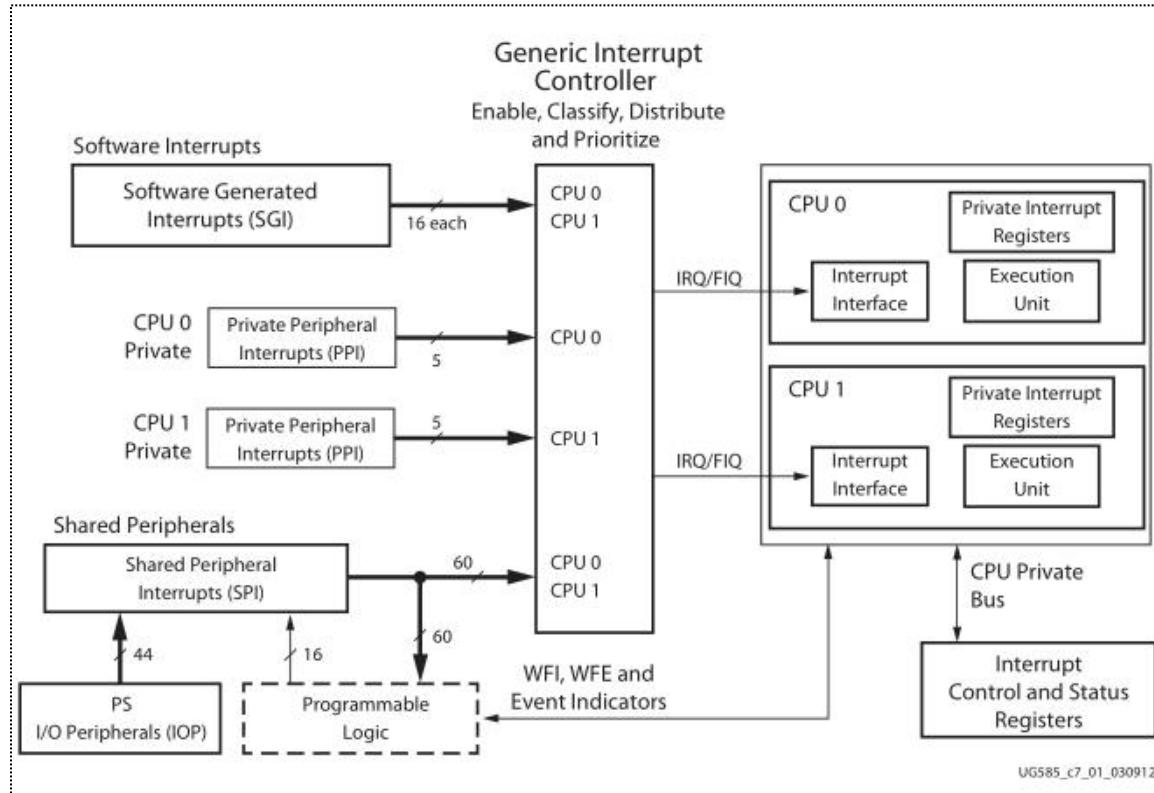
The figure contains two GICs:



RPU GIC: PL390 (corresponding to GICv1 IP designed by arm)

APU GIC: GIC-400 (corresponding to GICv2 IP designed by arm company)

4) RPU GIC, its system functional block diagram is as follows:



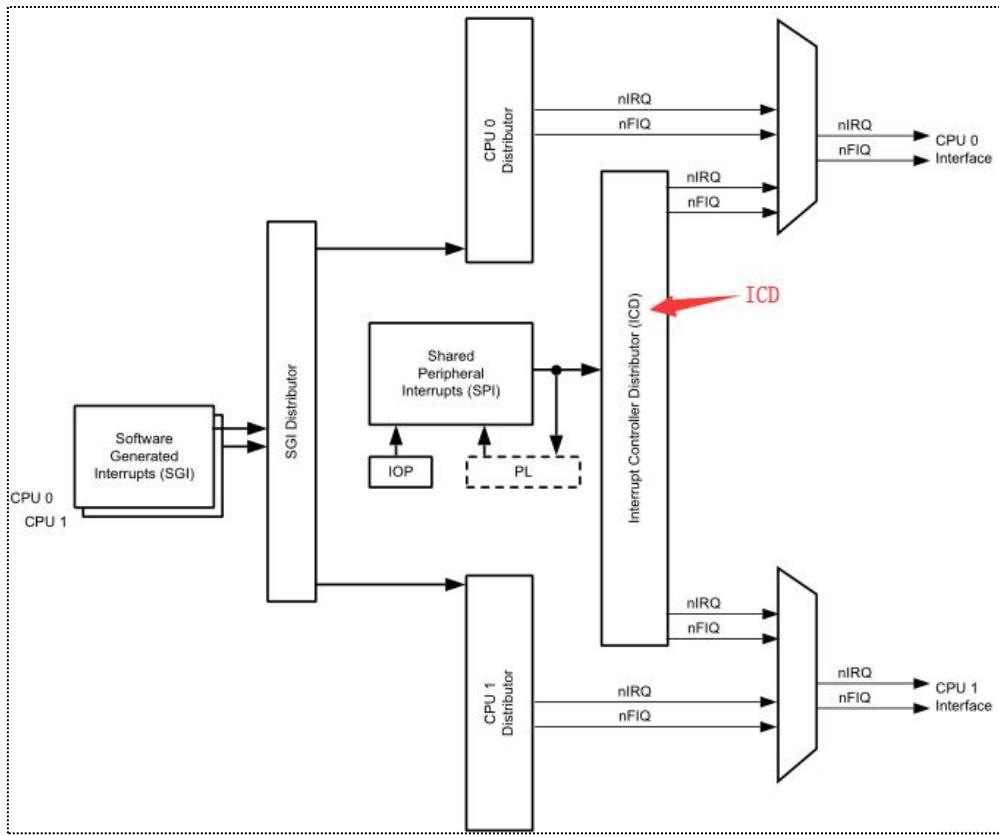
As can be seen from the figure, there are three main interrupt sources:

PPI: private peripheral interrupt, the interrupt originates from the peripheral and is only valid for a fixed core.

SPI: shared peripheral interrupt, the interrupt originates from the peripheral and can be effective for all cores.

SGI: software-generated interrupt, the interrupt generated by the software, used to transmit an interrupt signal to the specified core

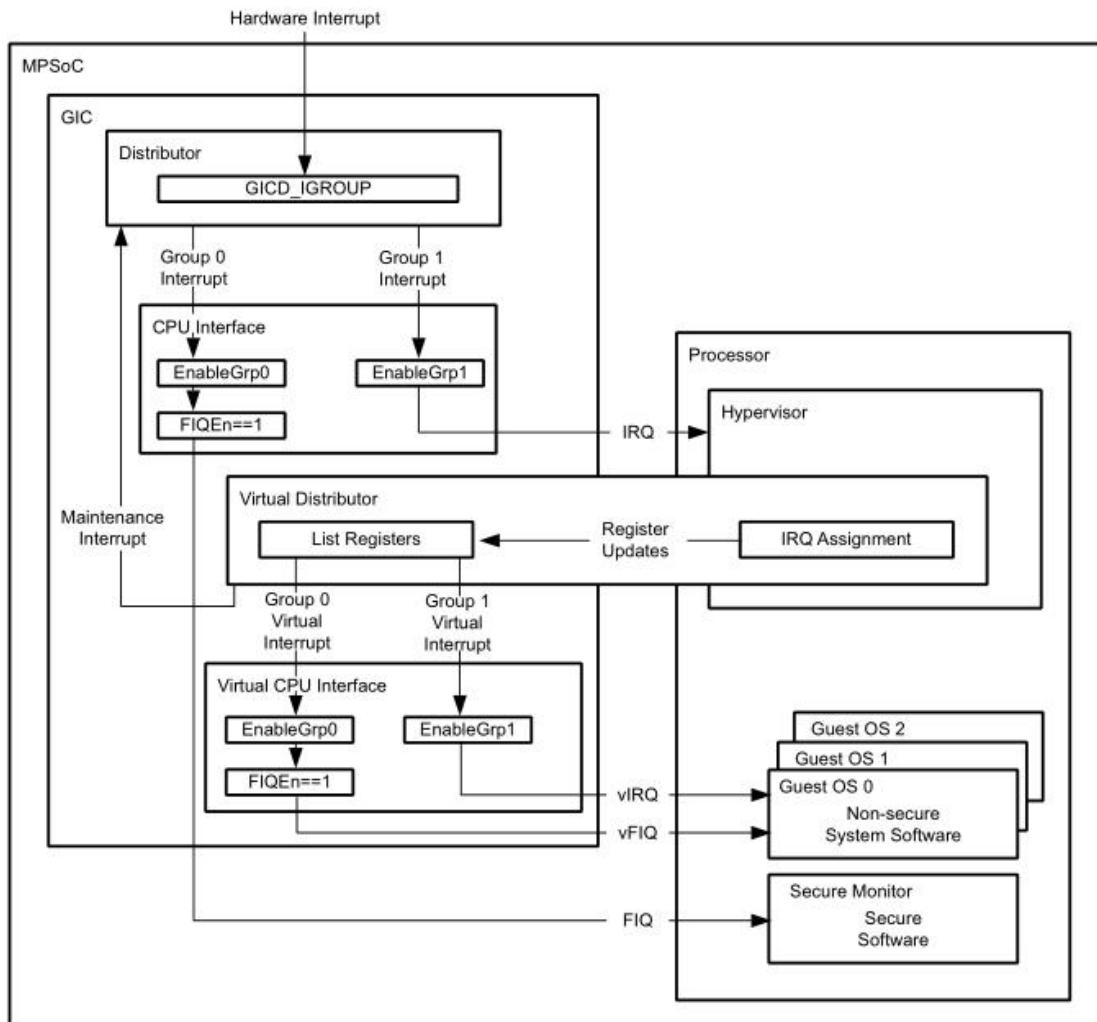
The functional block diagram of the controller is as follows:



In the above figure, the control registers of ICD are mainly as follows:

Starting Address	Register Set	Count	Description
<b>RPU - Private CPU Bus for RPU MPCore</b>			
0xF900_0000	PL390.enable	1	Interrupt control register (ICDICR).
0xF900_0080	PL390.sgi_security_if_n	1	SGI interrupt security register (ICDISR).
0xF900_0084	PL390.spi_security	5	SPI interrupt security register (ICDISR).
0xF900_0104	PL390.spi_enable_set	5	SPI enable set register (ICDISER).
0xF900_0184	PL390.spi_enable_clr	5	SPI interrupt clear-enable registers (ICDICER).
0xF900_0200	PL390.sgi_pending_set_if_n	1	SGI interrupt set-pending registers (ICDISPR).
0xF900_0204	PL390.spi_pending_set	5	SPI interrupt set-pending registers (ICDISPR).
0xF900_0280	PL390.sgi_pending_clr_if_n	1	SGI pending clear register (ICDICPR).
0xF900_0284	PL390.spi_pending_clr	5	SPI pending clear register (ICDICPR).
0xF900_0300	PL390.sgi_active_if_n	1	SGI active bit registers (ICDABR).
0xF900_0304	PL390.spi_active	5	SPI active bit registers (ICDABR).
0xF900_0400	PL390.priority_sgi_if_n	16	SGI interrupt priority registers (ICDIPR).
0xF900_0420	PL390.priority_spi	160	SPI interrupt priority registers (ICDIPR).
0xF900_0820	PL390.targets_spi	160	SPI target register interrupt (ICDIPTR).
0xF900_0C08	PL390.spi_config	5	SPI interrupt configuration register Interrupt (ICDICR).

## 5) APU GIC, the functional block diagram is as follows



GICv2 divides the interrupt into group0 and group1. Use the register GICD\_IGROUPRn to set the group for each interrupt. Where group0: safety interrupt, driven by nFIQ. group1: non-secure interrupt, driven by nIRQ. Supports up to 1020 interrupts. The interrupt number is allocated as follows:

Interrupt Number	Allocation	Interrupt Number	Registers
ID0-ID7	Non-safe Soft Interrupt	Software	GICD_SGIR
ID8-ID15	Safe Soft Interrupt	Software	GICD_SGIR
ID16-ID31	Private Interrupt	Peripherals	no
ID32-ID1019	Shared Interrupt	Peripherals	no

GICv2 is mainly composed of two parts: distributor and cpu

interface.

Distributor, used to collect all interrupt sources, and transmit interrupt priority, interrupt grouping, and interrupt destination core for each interrupt source. When an interrupt is generated, the current highest priority interrupt is transmitted to the corresponding cpu interface. Its functions are: global interrupt enable, each interrupt enable, interrupt priority, interrupt grouping, interrupt purpose core, interrupt trigger mode, for SGI interrupt, transfer interrupt to specified core, status of each interrupt Manage and provide software, can modify the pending status of interrupt

cpu interface, the interrupt information transmit by GICD is transmitted to the core connected to the cpu interface through IRQ and FIQ pins. Its functions include: transmitting interrupt requests to cpu, acknowledging an interrupt, indicating completion of an interrupt, setting interrupt priority shielding, defining interrupt preemption strategies, and determining the highest priority interrupt currently in the pending state

gicv2 defines some of its own registers. These registers are all accessed in a memory-mapped way, that is, in the soc, there will be a space for gic. The cpu accesses this part of the space to operate the gic. The main registers are as follows:

APU - AXI Interconnect with Access Restricted to APU MPCore			
0xF901_0000	GIC400.GICD	180	Display controller.
0xF902_0000	GIC400.GICC	15	CPU interface.
0xF904_0000	GIC400.GICH	99	Hypervisor.
0xF906_0000	GIC400.GICV	14	Virtual machine.

The interrupt here is just a brief introduction. For detailed understanding, please refer to the document provided by xilinx: ug1085-zynq-ultrascale-trm.pdf.

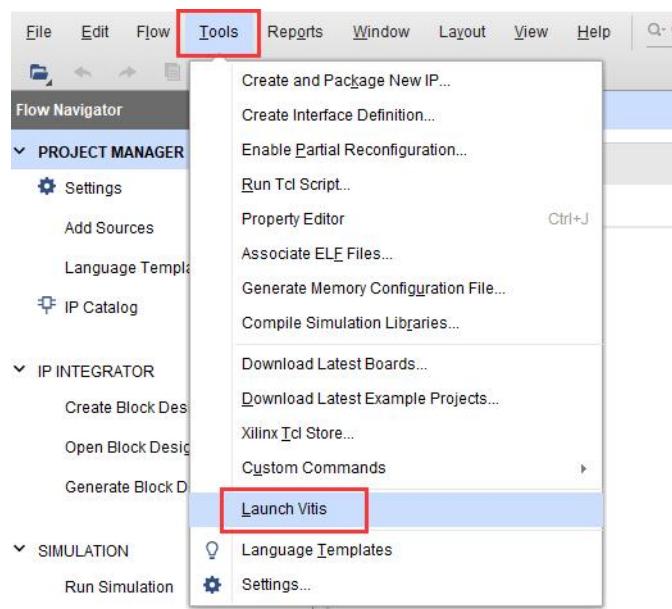
## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

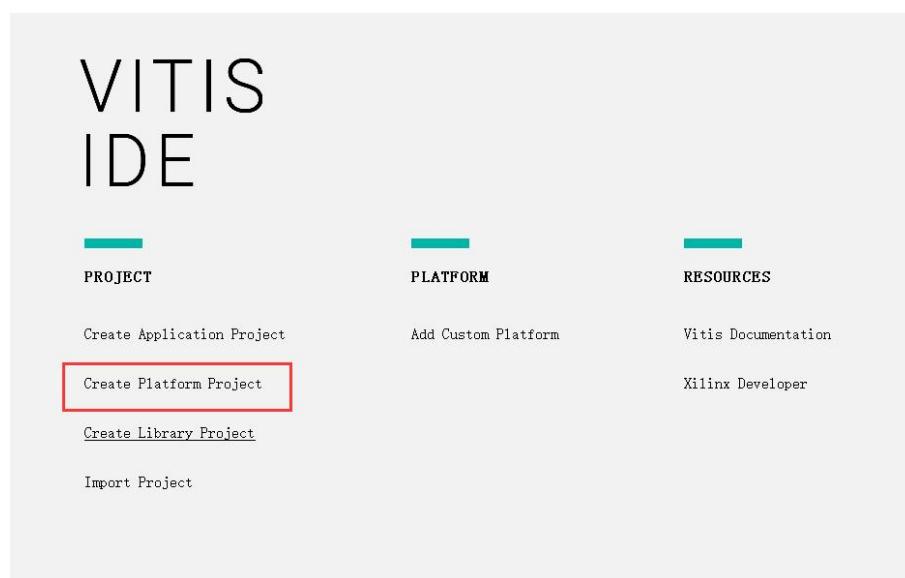
### Part 2.3: Vitis Programming

#### Part 2.3.1: Create Platform Project

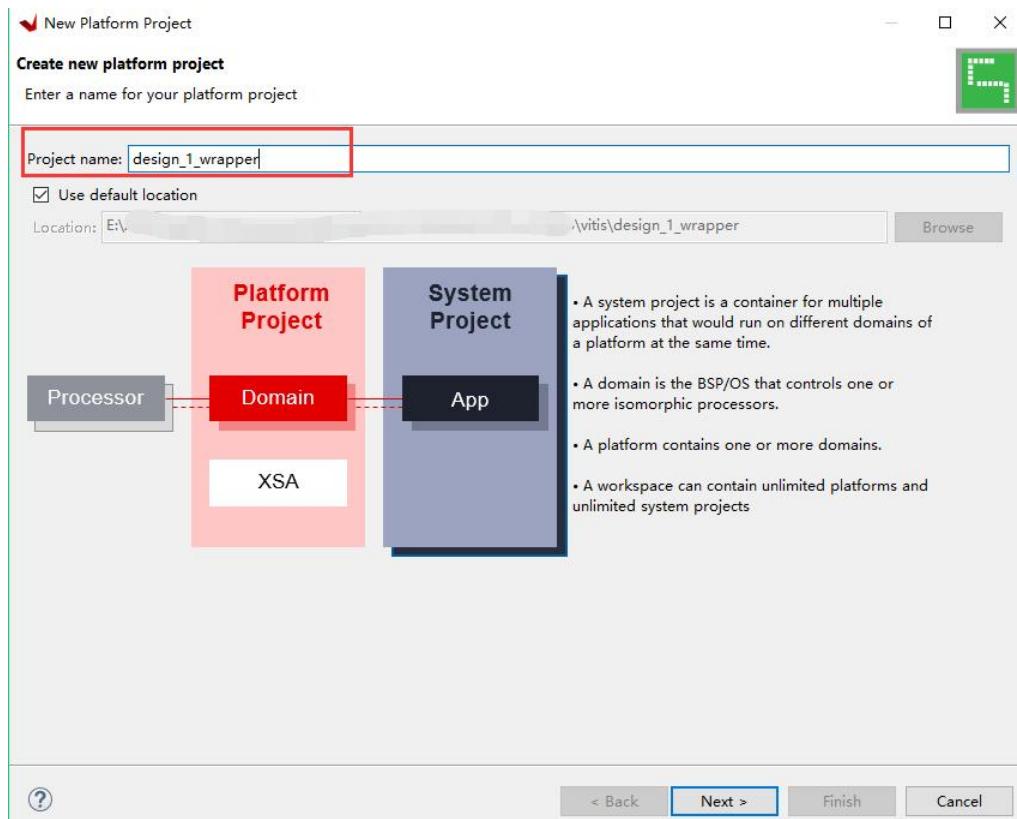
- 1) Click Tools→Launch Vitis



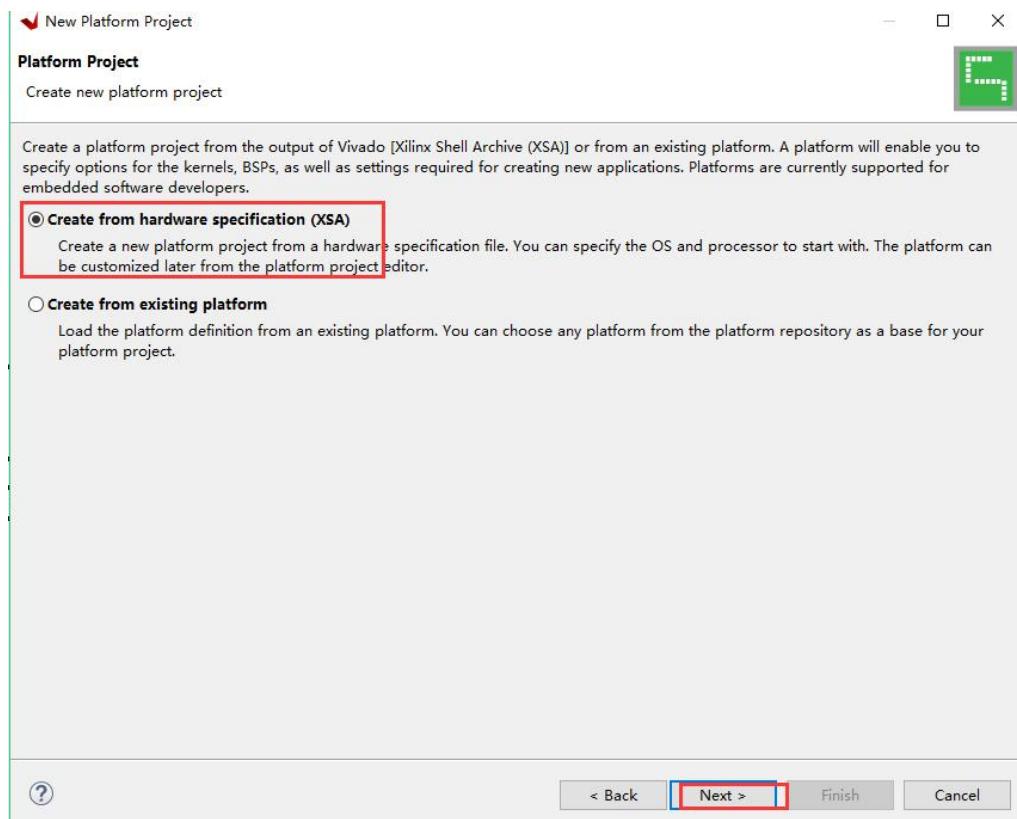
- 2) Unlike the previous Hello World experiment, we only build the Platform project



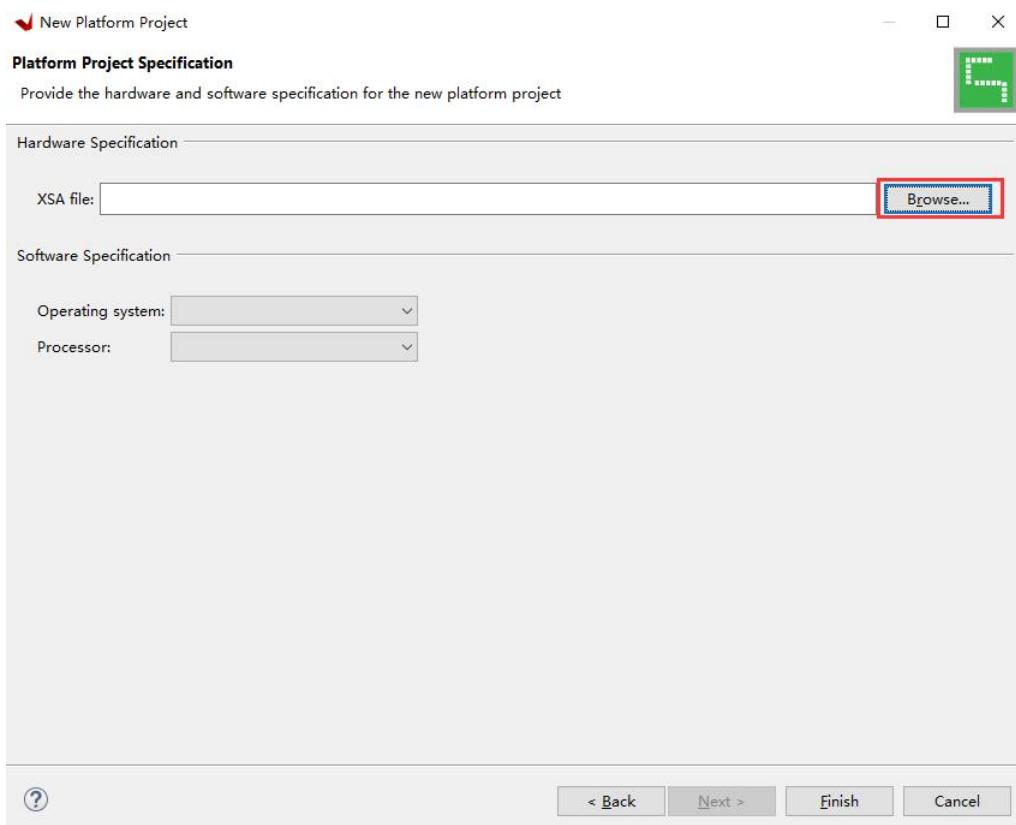
- 3) Fill in the project name, which should be the same as the name of the XSA file, click Next



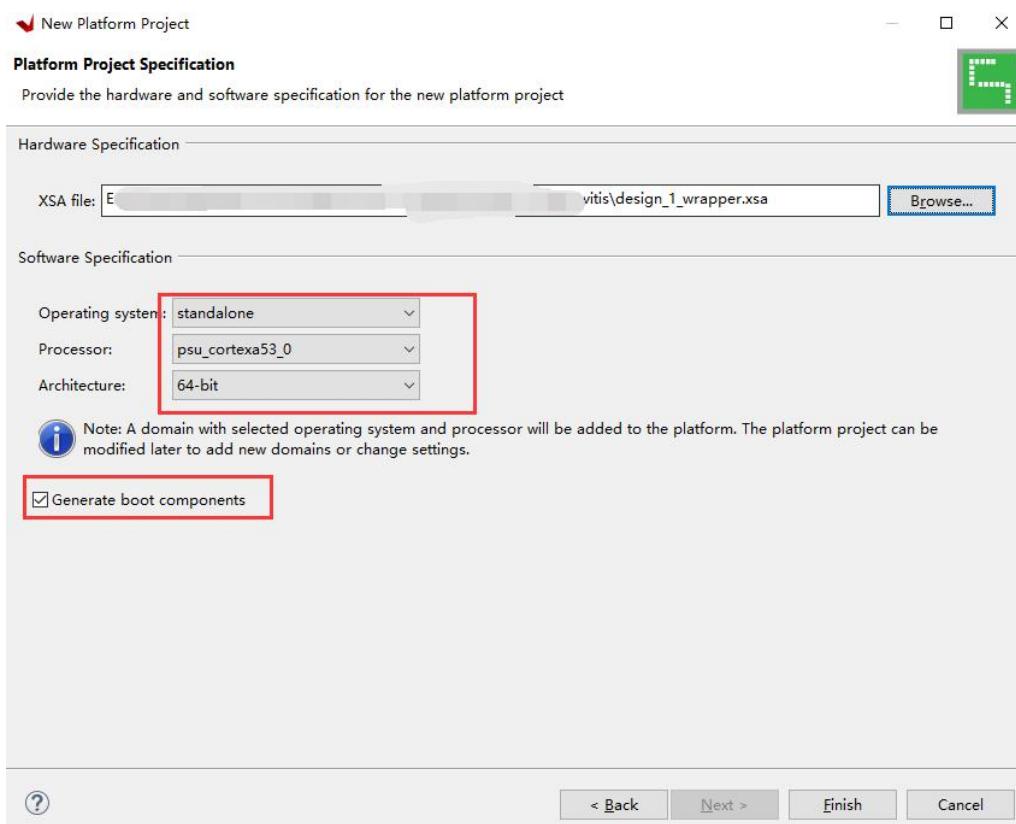
- 4) Click Next



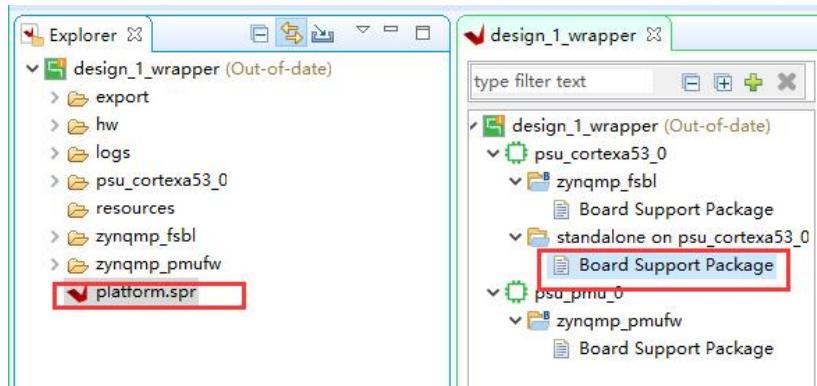
## 5) Select XSA file



Keep the default, click Finish



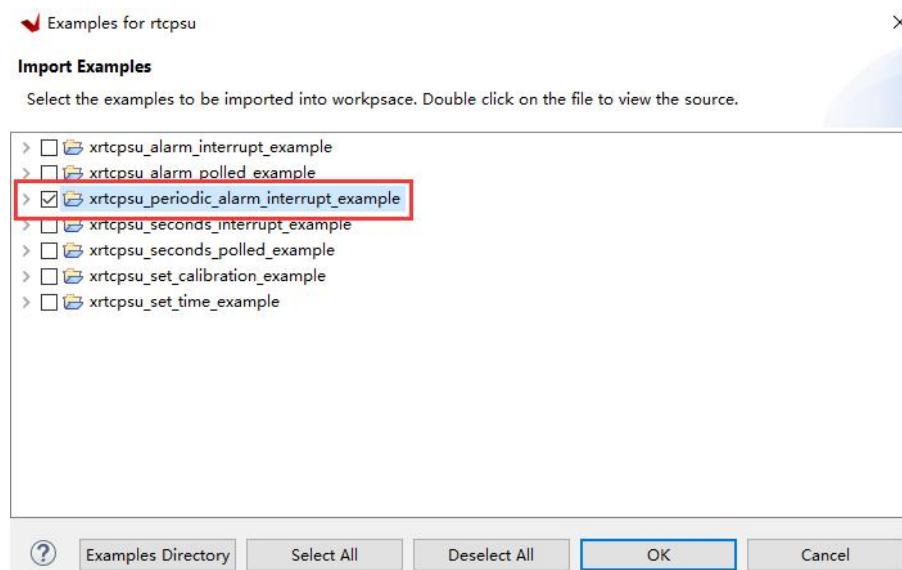
- 6) Click to open platform.spr, and click to open BSP



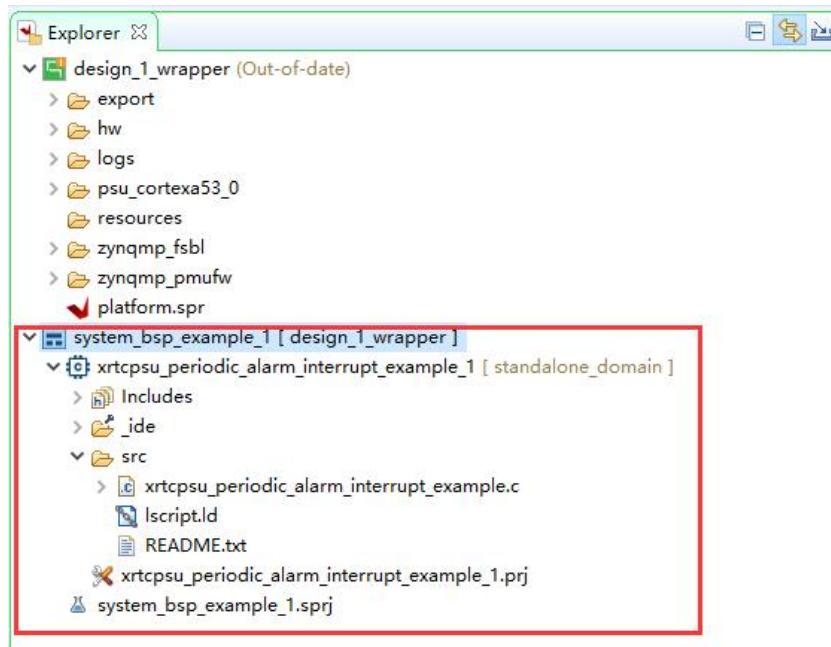
- 7) Find the RTC driver, and click Import Examples

Name	Driver	Documentation	Examples
psu_qspi_linear_0	generic	-	-
psu_r5_0_atcm_global	generic	-	-
psu_r5_0_btcm_global	generic	-	-
psu_r5_1_atcm_global	generic	-	-
psu_r5_1_btcm_global	generic	-	-
psu_r5_tcm_ram_global	generic	-	-
psu_rcpu_gic	scugic	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>
psu_rpu	generic	-	-
psu_rsa	generic	-	-
psu_RTC	rtcpsu	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>
psu_sd_0	sdps	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>
psu_sd_1	sdps	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>
psu_serdes	generic	-	-
psu_siou	generic	-	-
psu_smmu_gpv	generic	-	-

- 8) Fortunately, there are examples of interrupts. How do you know that this example is an interrupted example? It is guessed by "intr", so the basic skills are very important, otherwise you won't even know how to find a routine.



## 9) Import the example project here



The following is to read the code, and then modify the code. Of course, you may not fully understand the code at a time, and you can only practice repeatedly in future applications.

- 10) Get the system seconds counter value through the function “**XRtcPsu\_GetCurrentTime**”, and use the function “**XRtcPsu\_SecToDate**” to convert the count value into a year, month, day, hour, minute, and second that we can see clearly

```
xil_printf("\n\rDay Convention : 0-Fri, 1-Sat, 2-Sun, 3-Mon, 4-Tue, 5-Wed, 6-Thur\r\n");
xil_printf("Current RTC time is..\r\n");
CurrentTime = XRtcPsu_GetCurrentTime(RtcInstPtr);
XRtcPsu_SecToDate(CurrentTime,&dt0);
xil_printf("YEAR:MM:DD HR:MM:SS \t %04d:%02d:%02d:%02d:%02d\t Day = %d\r\n",
dt0.Year,dt0.Month,dt0.Day,dt0.Hour,dt0.Min,dt0.Sec,dt0.WeekDay);
```

- 11) Set the interrupt time, the interrupt time “**PERIODIC\_ALARM\_PERIOD**” macro is defined as **2**, that is, interrupt once every **2** seconds

```

/*
 * Setup the handlers for the RTC that will be called from the
 * interrupt context when alarm and seconds interrupts are raised,
 * specify a pointer to the RTC driver instance as the callback reference
 * so the handlers are able to access the instance data
 */
XRtcPsu_SetHandler(RtcInstPtr, (XRtcPsu_Handler)Handler, RtcInstPtr);

/*
 * Enable the interrupt of the RTC device so interrupts will occur.
 */
XRtcPsu_SetInterruptMask(RtcInstPtr, XRTC_INT_EN_ALRM_MASK );

CurrentTime = XRtcPsu_GetCurrentTime(RtcInstPtr);
Alarm = CurrentTime + PERIODIC_ALARM_PERIOD;           // 设置中断时间

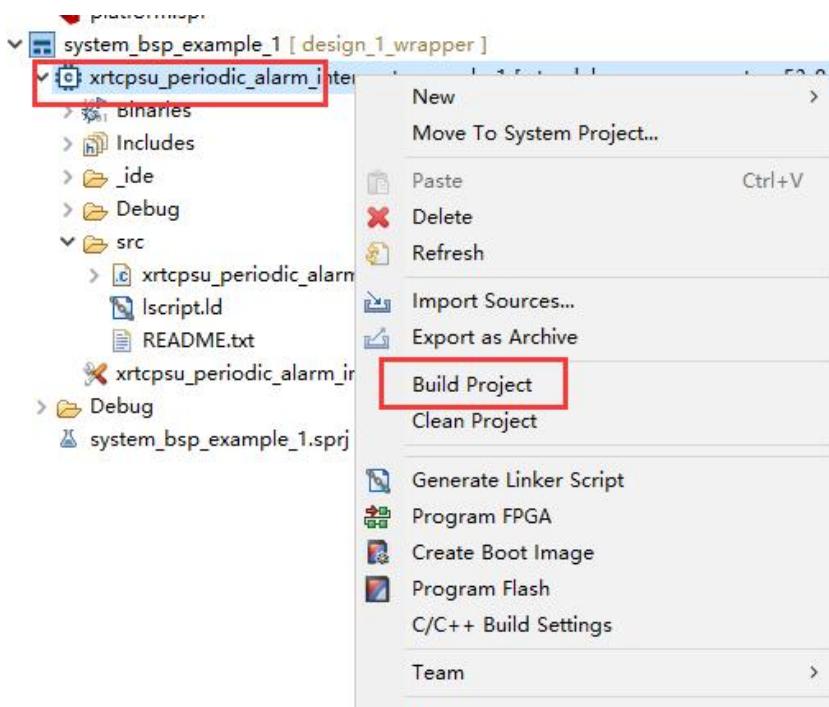
XRtcPsu_SetAlarm(RtcInstPtr, Alarm, 1U);               // 连续中断

while( PeriodicAlarms != REPETITIONS);                 // 等待10次中断

/*
 * Disable the interrupt of the RTC device so interrupts will not occur.
 */
XRtcPsu_ClearInterruptMask(RtcInstPtr,XRTC_INT_DIS_ALRM_MASK);
XRtcPsu_ResetAlarm(RtcInstPtr);

```

## 12)Build Project



13)To understand the use of the interrupt controller, it is mainly divided into several steps, Initialize the interrupt controller GIC → Initialization interrupt exception → Interrupt service function registration → Enable interrupt enable in the interrupt controller → Interrupt exception. There are two steps need to pay attention, "Enable interrupt enable in the interrupt controller" is to enable the

corresponding interrupt according to the interrupt number, such as the RTC introduced in this chapter, which is the enable interrupt in the interrupt controller GIC. The latter enabling peripheral interrupts refers to opening its interrupts in the peripherals. Under normal circumstances, it is not be open. After opening, interrupts can be generated and passed to the interrupt controller GIC. This way of writing can be used for reference in future experiments.

```
static int SetupInterruptSystem(XScuGic *IntcInstancePtr,
                               XRtcPsu *RtcInstancePtr,
                               u16 RtcIntrId)
{
    int Status;

#ifndef TESTAPP_GEN
    XScuGic_Config *IntcConfig; /* Config for interrupt controller */

    /* Initialize the interrupt controller driver */
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (NULL == IntcConfig) {
        return XST_FAILURE;
    }
    Interrupt Initialization
    Status = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig,
                                   IntcConfig->CpuBaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Connect the interrupt controller interrupt handler to the
     * hardware interrupt handling logic in the processor.
     */
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler) XScuGic_InterruptHandler,
                                IntcInstancePtr);
#endif
    Interrupt Exception Service Function Registration
    /*
     * Connect a device driver handler that will be called when an
     * interrupt for the device occurs, the device driver handler
     * performs the specific interrupt processing for the device
     */
    Status = XScuGic_Connect(IntcInstancePtr, RtcIntrId,
                            (Xil_ExceptionHandler) XRtcPsu_InterruptHandler,
                            (void *) RtcInstancePtr);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    Interrupt Service Function Registration

    /* Enable the interrupt for the device */
    XScuGic_Enable(IntcInstancePtr, RtcIntrId);

Enable RTC Interrupt
#ifndef TESTAPP_GEN
    /* Enable interrupts */
    Xil_ExceptionEnable();
#endif
Enable Interrupt Exception
    return XST_SUCCESS;
}
```

## Part 2.4: Download and Debug

- 1) Open the serial terminal
- 2) The method of downloading the debugger has been explained in the previous tutorial and will not be repeated
- 3) As we expected, the serial port will be checked every two seconds

```
Day Convention : 0-Fri, 1-Sat, 2-Sun, 3-Mon, 4-Tue, 5-Wed, 6-Thur
Current RTC time is..
YEAR:MM:DD HR:MM:SS      2000:01:03 00:18:09      Day = 3
2Sec Periodic Alarm generated.
Successfully ran RTC Periodic Alarm Interrupt Example Test
```

## Part 2.5: Experimental Summary

In the experiment, by simply modifying the routines of Vitis, the application of RTC and interrupt was completed. It seems to be a simple operation, but it contains a wealth of knowledge. We need to understand the principle of RTC and the principle of interrupt. These basic knowledge are necessary conditions for learning ZYNQ well.

## Part 3: PS MIO Experiment

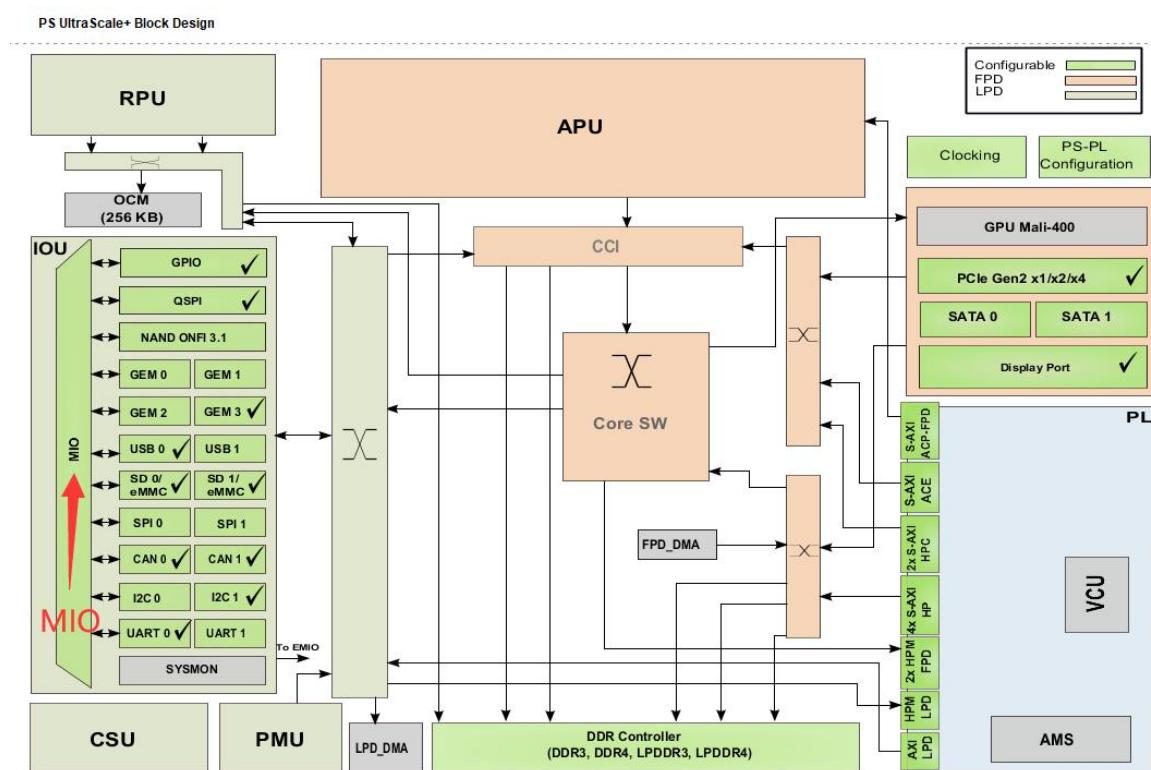
The vivado project directory is "ps\_hello/vivado"

The vitis project directory is "ps\_mio/vitis"

### Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

This chapter introduces the operation of the MIO on the PS side. MIO is the basic peripheral IO. It can be connected to SPI, I2C, UART, GPIO, etc. Via VIVADO software settings, the software can export signals through MIO. The signal can also be connected to the PL pin through EMIO.



This experiment demonstrates the operation of MIO by implementing the blinking of the LED light on the PS side.

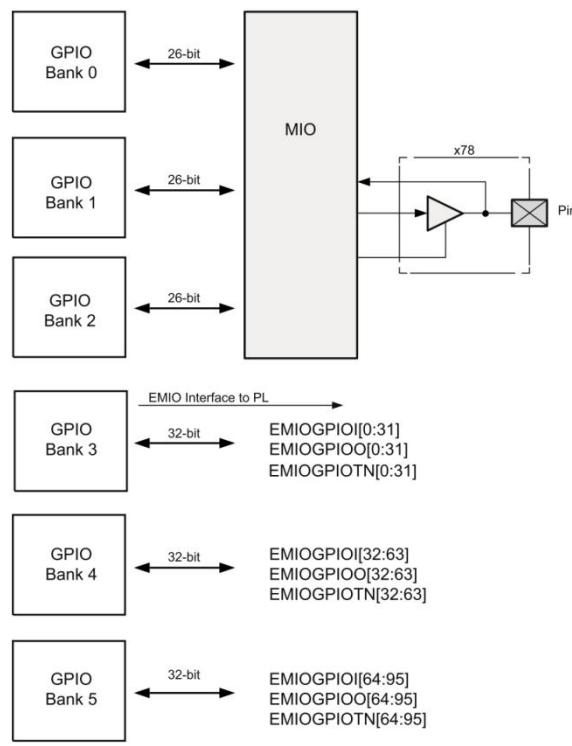
## Part 3.1: Principle Introduction

Let's first understand the bank distribution of GPIO. In the GPIO chapter of the UG1085 document, you can see that there are 6 banks for GPIO.

BANK0 controls 26 signals, and BANK1 controls 22 signals, which are 54 pins of MIO in total, that is, PS-side peripheral interfaces such as SPI, I2C, USB, SD and so on.

BANK3~BANK5 can control 96 PL pins in total. Note that each group has three signals, input EMIOGPIOI, output EMIOGPIOO, output enable EMIOGPIOTN, similar to a three-state gate, with a total of 288 signals. It can be connected to the PL terminal pin and controlled by the PS terminal.

This chapter mainly talks about MIO control.



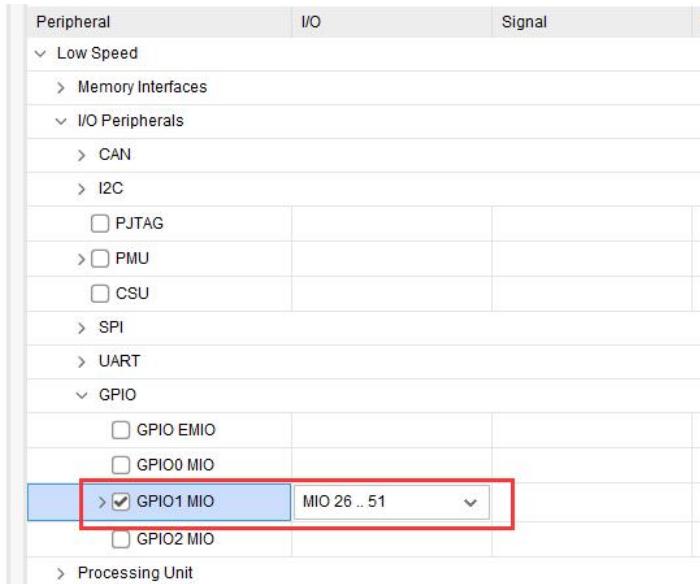
X15456-092516

Figure 27-1: GPIO Block Diagram

## Part 3.2: Create a “Vivado” Project

This experiment is based on the "ps\_hello" project. Through the

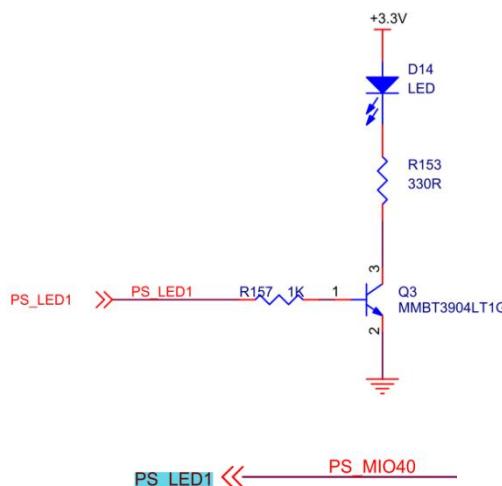
schematic diagram, we know that the LEDs and Keys on the PS side are on GPIO1, which is MIO BANK1, so we need to turn on the GPIO1 MIO, which has been configured in the "ps\_hello" project.



## Part 3.3: Vitis Program Development

### Part 3.3.1: MIO Lights up PS LED

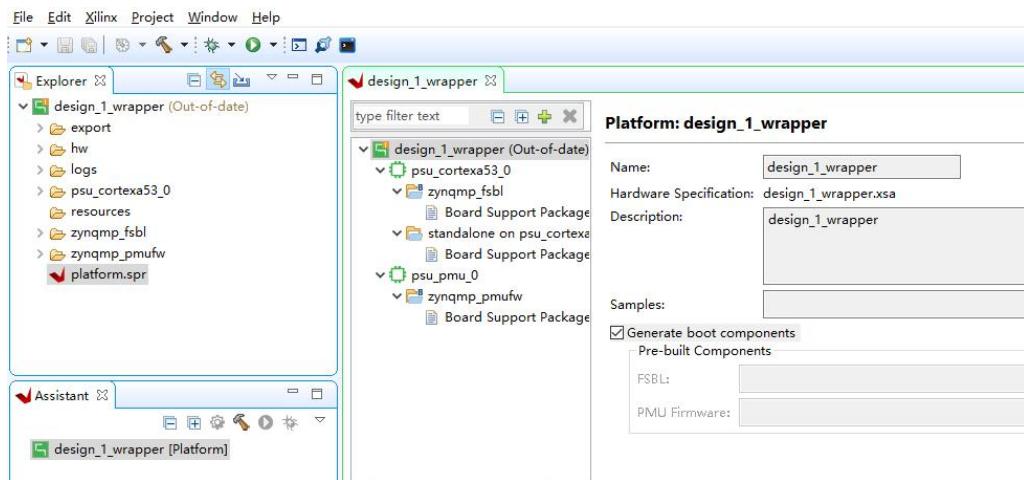
According to the schematic, the LED is connected to MIO40 on the PS side, and can be controlled according to the position of the corresponding FPGA development board MIO.



AXU3EG/AXU4EV/AXU5EV Schematic

- 1) The process of creating a new platform project will not be repeated,

please refer to the chapter "PS-side RTC interrupt experiment"



- 2) The following figure is the control block diagram of “GPIO”. In the experiment, the output register, data register “DATA”, data mask register “MASK\_DATA\_LSW”, “MASK\_DATA\_MSW”, direction control register “DIRM”, and output enable controller “OEN” will be used.

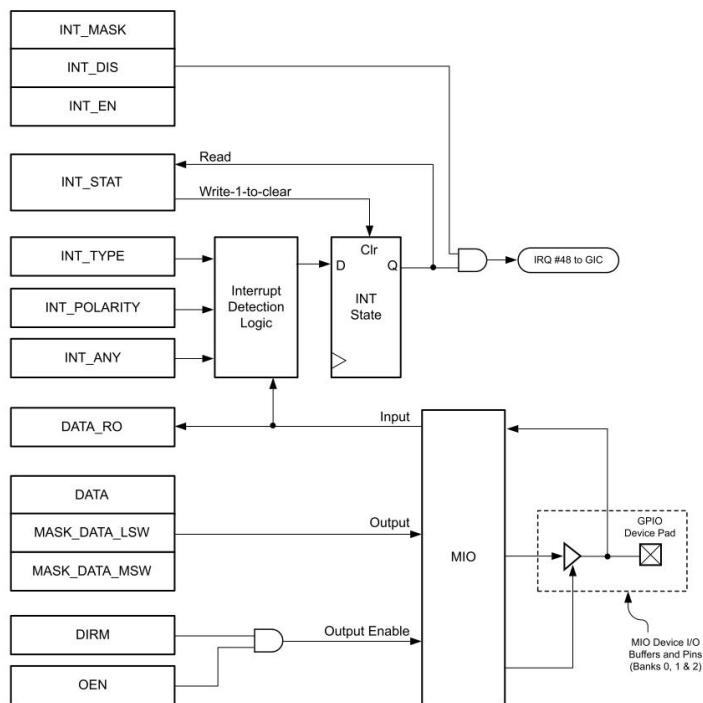
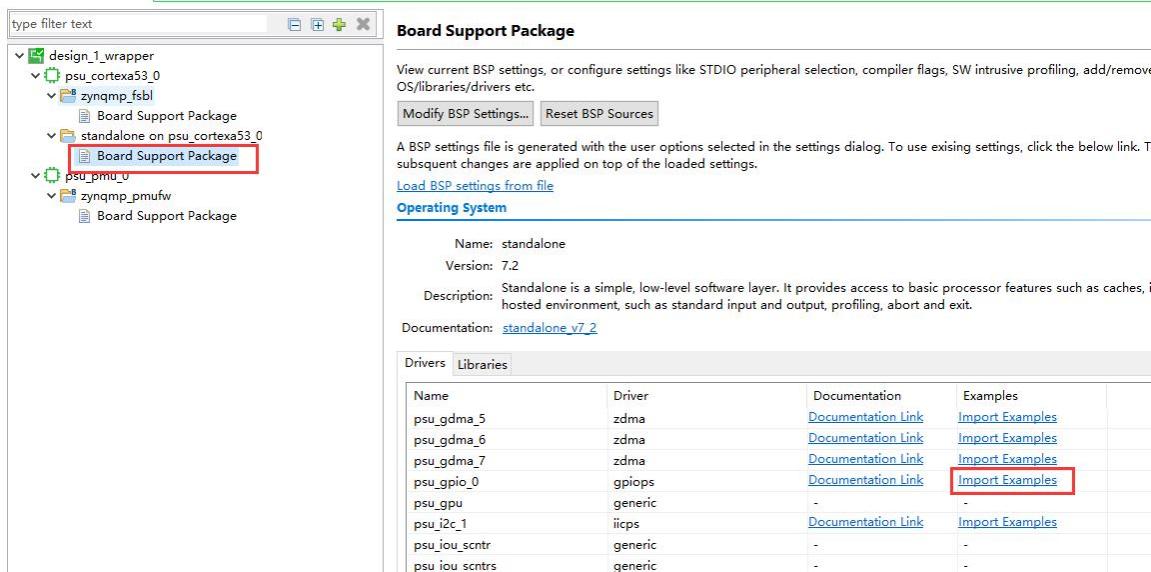


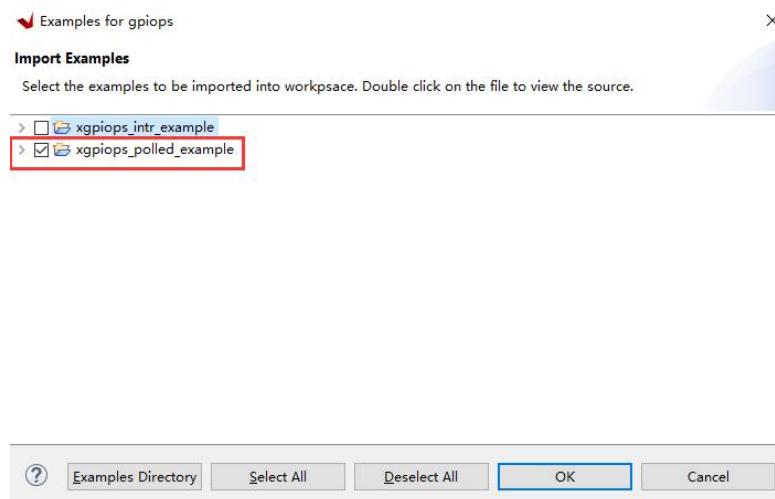
Figure 27-2: GPIO Channel

- 3) At the beginning, you may not be able to write the code. You can

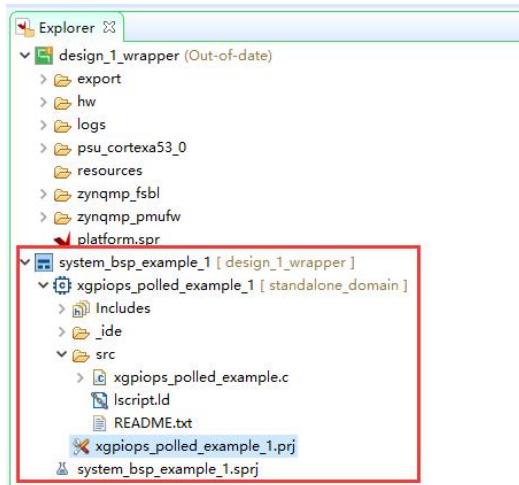
import the example project provided by Xilinx, find “ps7\_gpio\_0” in BSP, and click “Import Examples”



Select "xgpiops\_polled\_example" in the pop-up window and click OK



The new APP project will appear



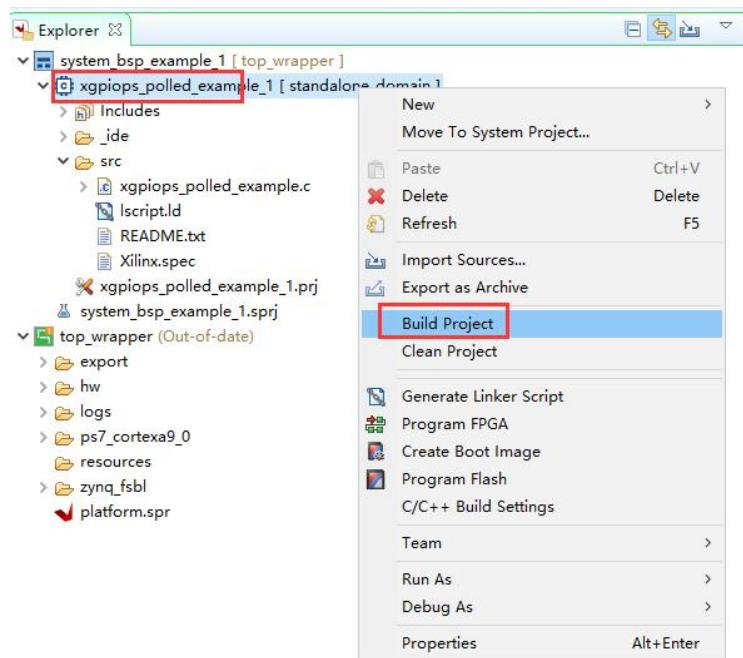
- 4) This example project tests the input and output of the MIO on the PS side. Since the LED on the PS side of the FPGA development board is MIO40, you need to modify the Output\_pin to 40 in the file to test the MIO40 LED.

```
72 //*****
73 int GpioPolledExample(u16 DeviceId, u32 *DataRead)
74 {
75     int Status;
76     XGpioPs_Config *ConfigPtr;
77     int Type_of_board;
78
79     /* Initialize the GPIO driver. */
80     ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
81 #ifdef versal
82     if(ConfigPtr->DeviceId == 0x0U)
83     {
84         /* Accessing PMC GPIO by setting 1 value*/
85         Gpio.PmcGpio=1;
86     }
87 #endif
88     Type_of_board = XGetPlatform_Info();
89     switch (Type_of_board) {
90         case XPLAT_ZYNQ_ULTRA_MP:
91             Input_Pin = 22;
92             Output_Pin = 40; // Line 92
93             break;
94
95         case XPLAT_ZYNQ:
96             Input_Pin = 14;
97             Output_Pin = 10;
98             break;
99     }
```

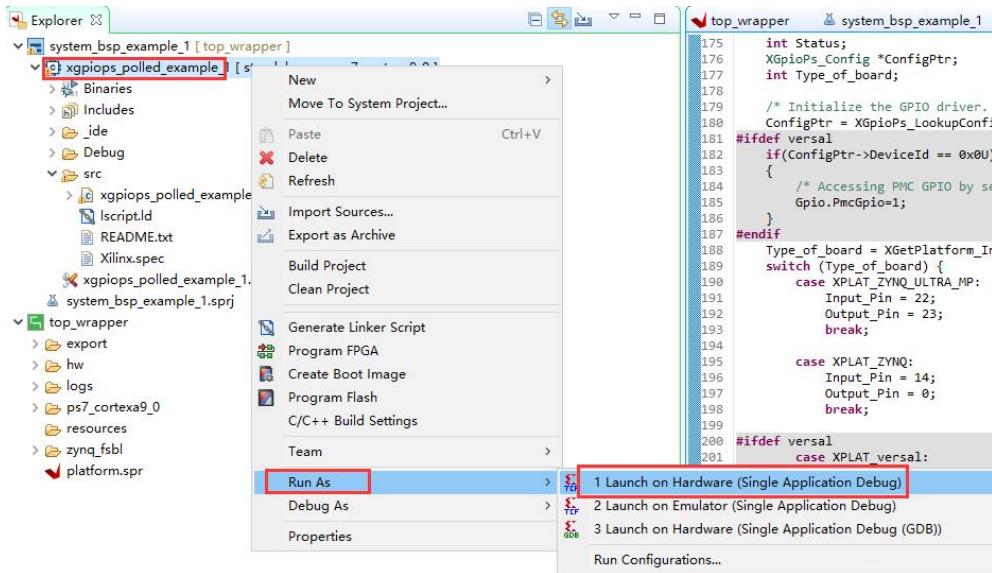
Since only the LED light is tested, that is, the output, comment out the input function and save the document.

```
214     Status = XGpioPs_CfgInitialize(&Gpio, ConfigPtr,
215                                     ConfigPtr->BaseAddr);
216     if (Status != XST_SUCCESS) {
217         return XST_FAILURE;
218     }
219     /* Run the Output Example. */
220     Status = GpioOutputExample();
221     if (Status != XST_SUCCESS) {
222         return XST_FAILURE;
223     }
224
225     /* Run the Input Example. */
226     // Status = GpioInputExample(DataRead);
227     // if (Status != XST_SUCCESS) {
228     //     return XST_FAILURE;
229     // }
230
231     return XST_SUCCESS;
232 }
```

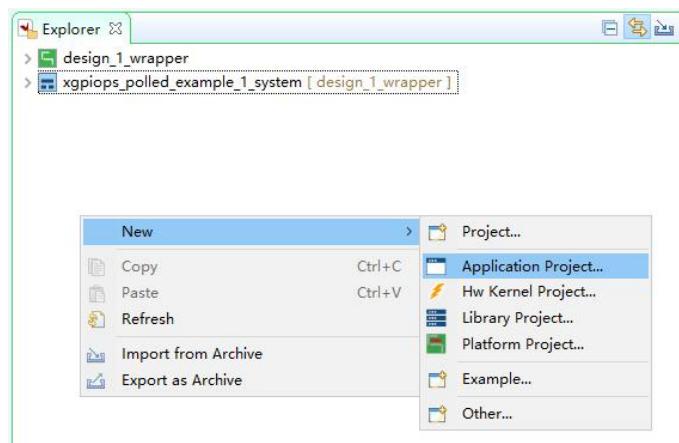
## 5) Build Project



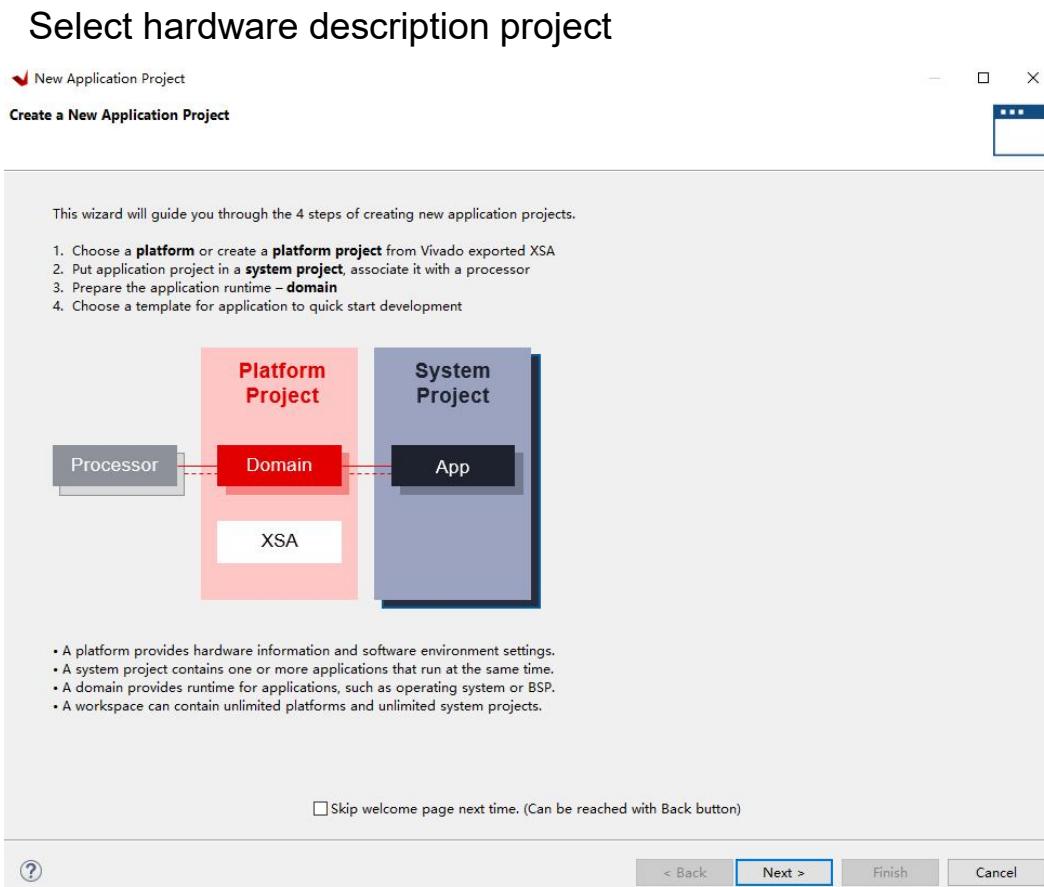
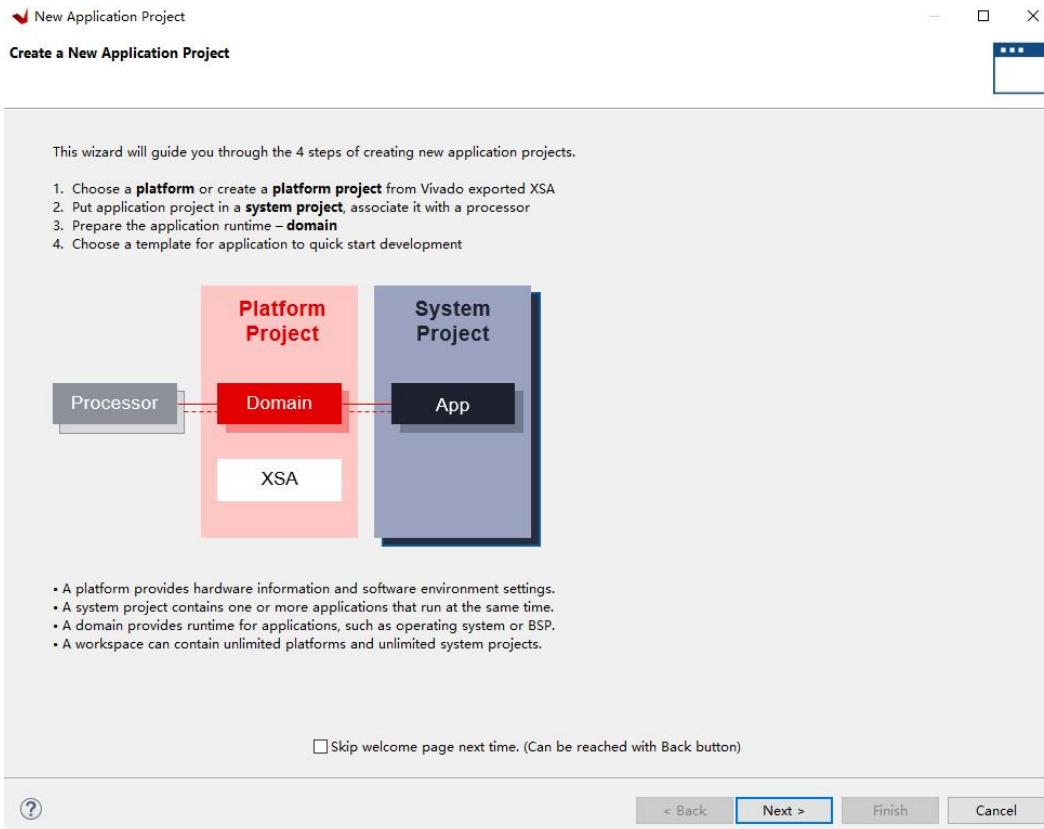
## 6) Run AsLaunch on Hardware (Single Application Debug), after downloading, you can see PS\_LED flashing 16 times quickly



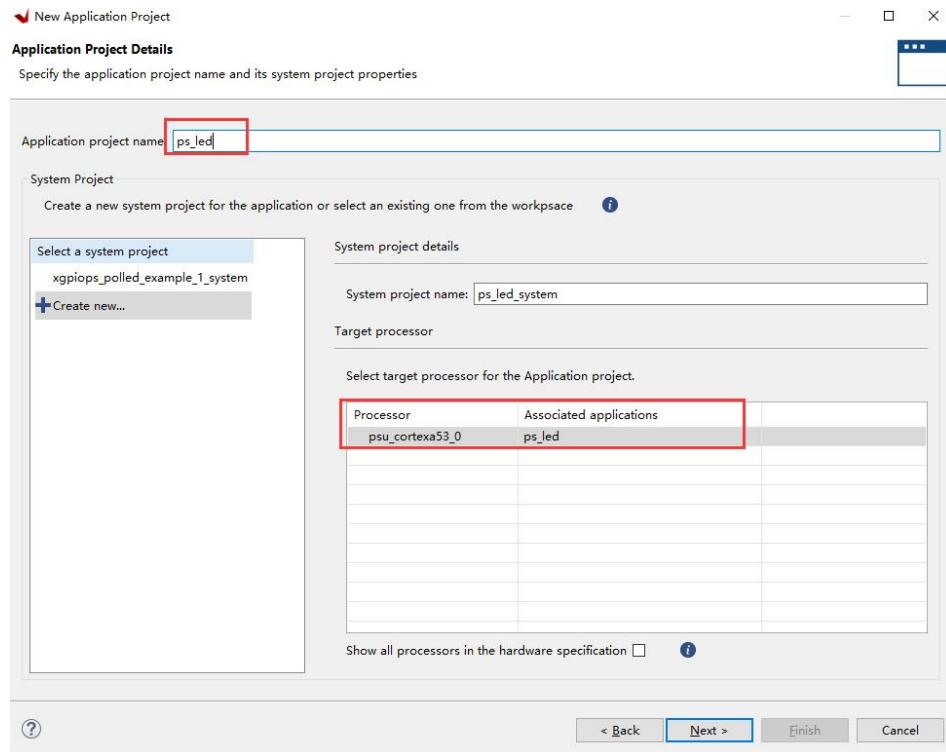
- 7) Although it is more convenient to use official examples, its code looks bloated. You can learn it and simplify it yourself. Modify it in "helloworld.c" of "ps\_led\_test". In fact, the procedure is very simple. "Initialize the GPIO→set the direction→output enable→control the GPIO output value". We create a new APP project. You can right-click New→Application Project in the blank space. Modify it in [helloworld.c](#) of [ps\\_led\\_test](#). In fact, the program steps are very simple, initialize GPIO→set direction→output enable→control GPIO output value



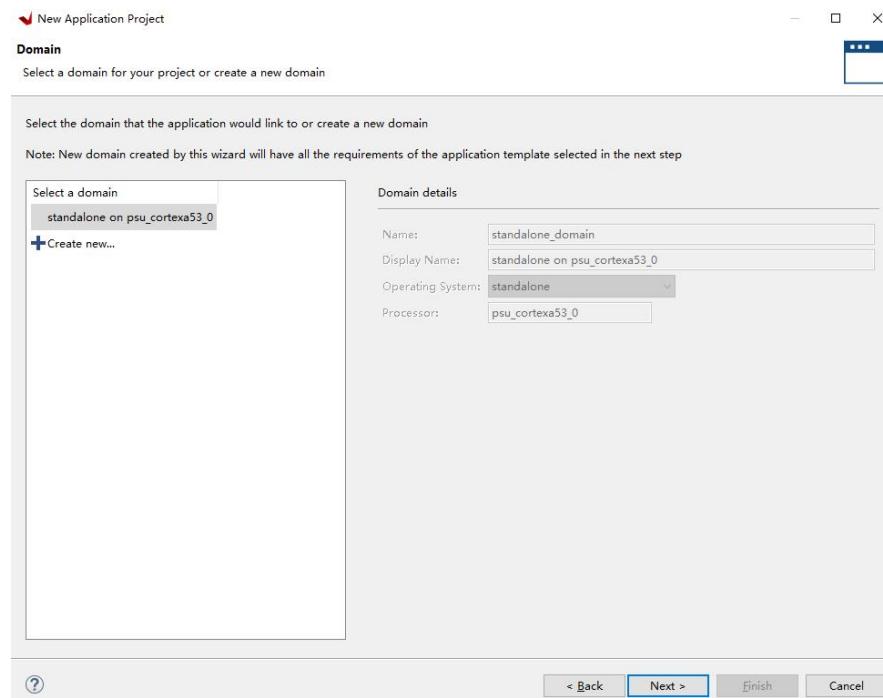
Skip the first page



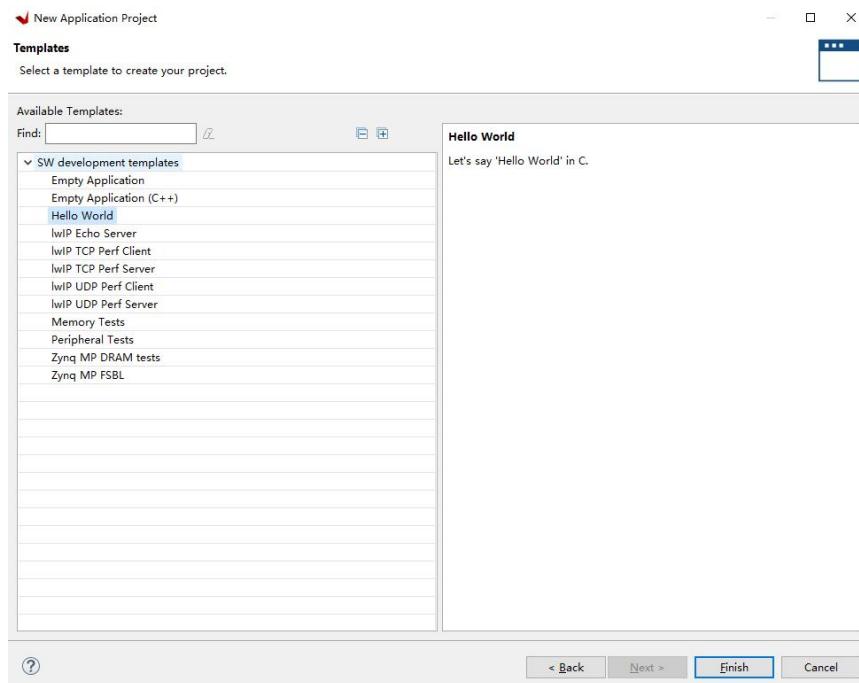
Fill in the project name and select the corresponding CPU



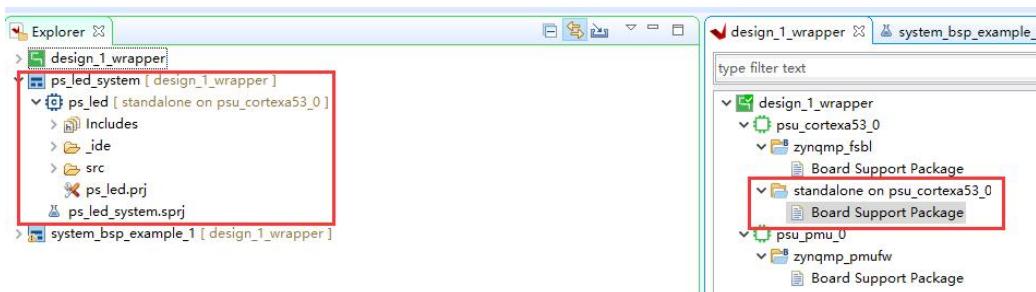
## Next step



Choose Hello World as the template



- 8) You can see that there is one more APP project, which is still based on the BSP named standalone on psu\_cortexa53\_0, which is a Domain, and shares the same BSP with the previous example project



- 9) Copy the code of the routine to helloworld.c, save and Build Project

```

int main()
{
    init_platform();

    int Status;
    XGpioPs_Config *ConfigPtr;

    print("Hello World\n\r");
    /* Initialize the GPIO driver. */
    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);

    Status = XGpioPs_CfgInitialize(&Gpio, ConfigPtr,
        ConfigPtr->BaseAddr);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Set the direction for the pin to be output and
     * Enable the Output enable for the LED Pin.
     */
    XGpioPs_SetDirectionPin(&Gpio, 40, 1);
    XGpioPs_SetOutputEnablePin(&Gpio, 40, 1);

    while(1){
        /* Set the GPIO output to be low. */
        XGpioPs_WritePin(&Gpio, 40, 0x0);
        sleep(1);
        /* Set the GPIO output to be high. */
        XGpioPs_WritePin(&Gpio, 40, 0x1);
        sleep(1);
    }

    cleanup_platform();
    return 0;
}

```

初始化  
GPIO

设置方向为  
输出，并使能  
输出

控制输出值

10) Download method is the same as before. You can see that the LED on the PS side start to blink.

### Part 3.3.2: MIO Key Interrupt

The MIO was introduced as an output to control the LED light. Here we will talk about using the MIO as a key input to control the LED light.

1) Look at the structure diagram of GPIO through the ug1085 document, the interrupt register:

INT\_MASK: interrupt mask

INT\_DIS: Interrupt off

INT\_EN: interrupt enable

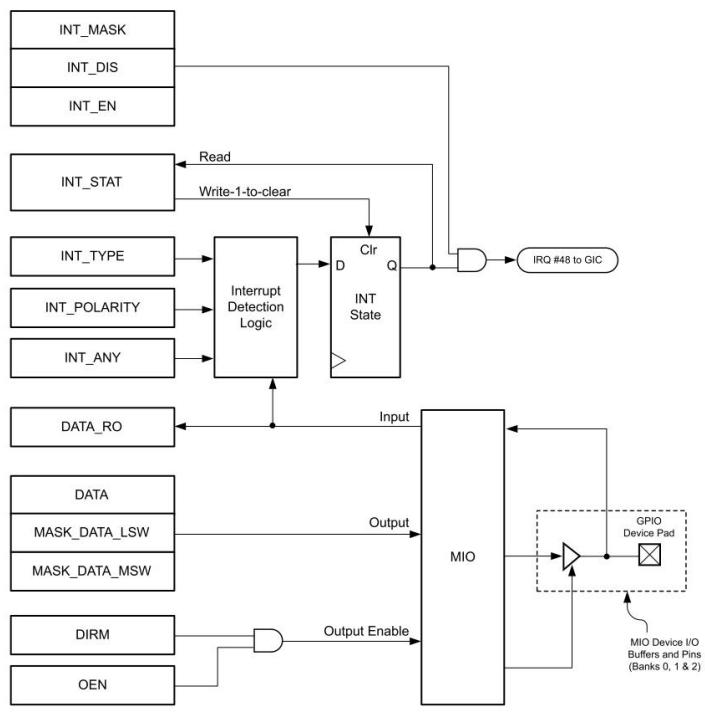
INT\_TYPE: interrupt type, set level sensitive or edge sensitive

INT\_POLARITY: Interrupt polarity, set low or falling edge or high or rising edge

INT\_ANY: edge trigger mode, you need to set INT\_TYPE as edge sensitive to use

When setting the interrupt generation mode, INT\_TYPE, INT\_POLARITY, and INT\_ANY must be used together. Please refer to

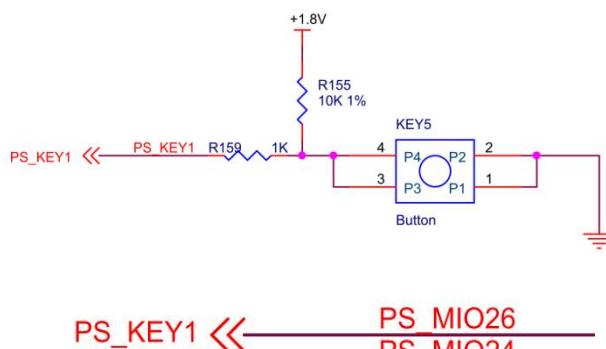
“ug1085 Register Details” for specific register meaning.



X18806-080318

Figure 27-2: GPIO Channel

It can be seen in the schematic diagram that the keys on the PS end is connected to MIO26. Pressing the key is low level, not pressing it is high level.



### AXU3EG/AXU4EV/AXU5EV Schematic

- 2) This experiment is designed to press the button, the LED lights up, and then press the LED to turn off.

**The main program design process is as follows:**

GPIO initialization → Set the direction of the keys and LED → Set

the interrupt method → Set the interrupt → Turn on the interrupt controller → Turn on the interrupt abnormal → Determine the KEY\_FLAG value, if it is 1, write the LED

### Interrupt processing flow:

Query the interrupt status register → determine the status → clear the interrupt → Set the KEY\_FLAG value

- 3) Create a new Vitis project
- 4) Define PS key number as 26 and PS LED as 40

```
/* GPIO parameter */
#define MIO_ID XPAR_XGPIOPS_0_DEVICE_ID
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
#define KEY_INTR_ID XPAR_XGPIOPS_0_INTR
#define PS_KEY_MIO 26
#define PS_LED_MIO 40

#define GPIO_INPUT 0
#define GPIO_OUTPUT 1
```

- 5) In the main function, set the LED and keys, and set the key interrupt type to a rising edge to generate an interrupt. In this experiment, the rising edge of the key signal generates an interrupt.

```
/*
 * Set the direction for the pin to be output and
 * Enable the Output enable for the LED Pin.
 */
XGpioPs_SetDirectionPin(&GPIO_PTR, PS_LED_MIO, GPIO_OUTPUT) ;
XGpioPs_SetOutputEnablePin(&GPIO_PTR, PS_LED_MIO, GPIO_OUTPUT) ;
/*
 * Set the direction for the pin to be input.
 * Set interrupt type as rising edge and enable gpio interrupt
 */
XGpioPs_SetDirectionPin(&GPIO_PTR, PS_KEY_MIO, GPIO_INPUT) ;
XGpioPs_SetIntrTypePin(&GPIO_PTR, PS_KEY_MIO, XGPIOPS_IRQ_TYPE_EDGE_RISING) ;
XGpioPs_IntrEnablePin(&GPIO_PTR, PS_KEY_MIO) ;
```

- 6) The interrupt controller setting function “IntrInitFuntions” is made with reference to the PS timer interrupt experiment, and the following statement sets the interrupt priority and trigger mode. That is, the “ICDIPR” and “ICDICFR” registers are operated.

```
XScuGic_SetPriorityTriggerType(InstancePtr, KEY_INTR_ID,
    0xA0, 0x3);
```

- 7) In the interrupt service program “GpioHandler”, determine the interrupt status register, clear the interrupt, and set the key flag to

1.

```
void GpioHandler(void *CallbackRef)
{
    XGpioPs *GpioInstancePtr = (XGpioPs *)CallbackRef ;
    int Int_val ;

    Int_val = XGpioPs_IntrGetStatusPin(GpioInstancePtr, PS_KEY_MIO) ;
    /*
     * Clear interrupt.
     */
    XGpioPs_IntrClearPin(GpioInstancePtr, PS_KEY_MIO) ;
    if (Int_val)
        key_flag = 1 ;
}
```

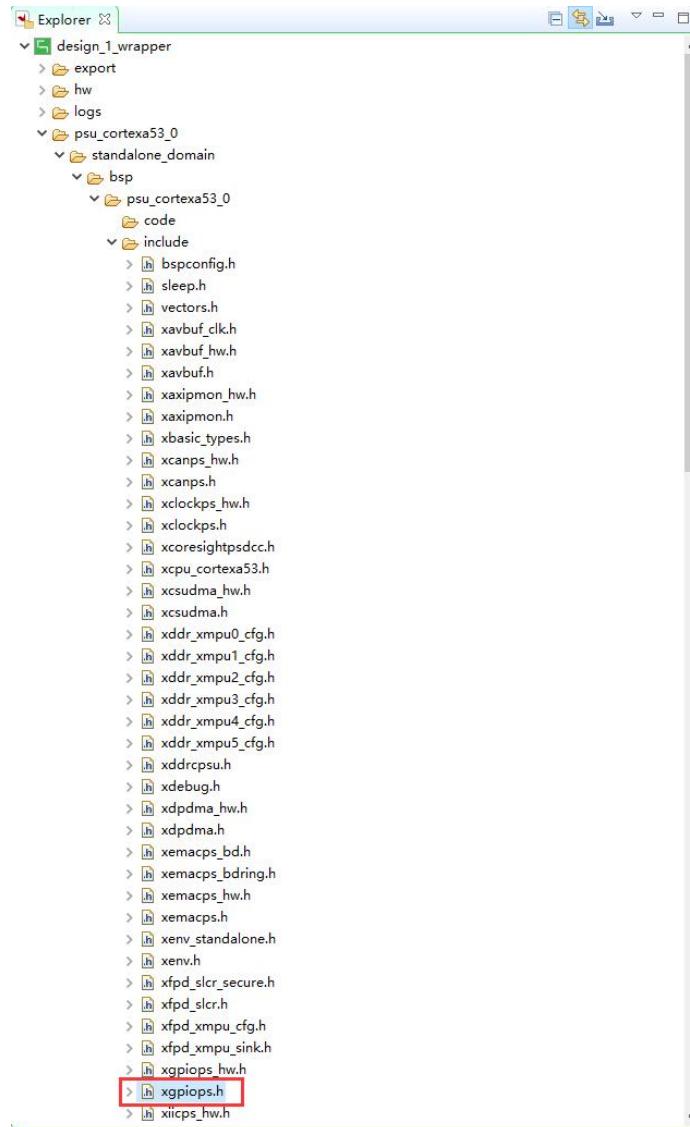
- 8) In the main function, judge the “key\_flag” and write data to the “LED”.

```
while(1)
{
    if (key_flag)
    {
        XGpioPs_WritePin(&GPIO_PTR, 9, key_val) ;
        key_val = ~key_val ;
        key_flag = 0 ;
    }
}
```

- 9) Build project and download the program.  
10) Observe the experimental phenomenon, press the PS\_Key, you can control the PS\_LED on and off.

### Part 3.4: Knowledge Sharing

- 1) Various header files of xilinx are included in the “bsp” “include” folder, such as the GPIO used in this chapter, “xgpiops.h” is used, and various macro definitions can be seen in this file. These macro definitions can be used when calling GPIO functions to improve readability.

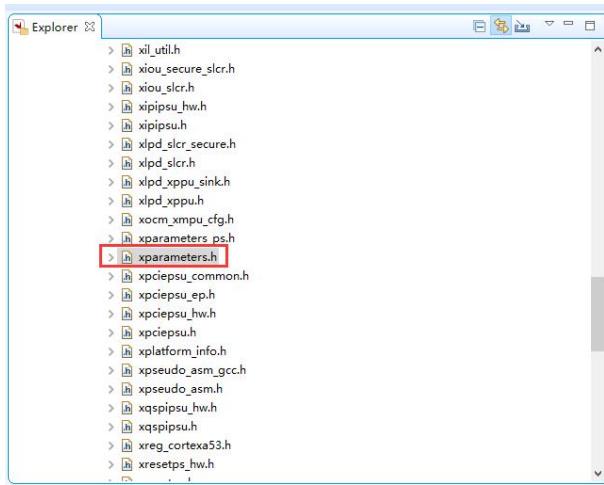


Also contains the function declarations that come with the peripherals

```
system.mss  xgpiops.h
----- Macros (inline functions) Definitions -----
***** Function Prototypes *****
/* Functions in xgpiops.c */
s32 XGpioPs_CfgInitialize(XGpioPs *InstancePtr, XGpioPs_Config *ConfigPtr,
                           u32 EffectiveAddr);

/* Bank APIs in xgpiops.c */
u32 XGpioPs_Read(XGpioPs *InstancePtr, u8 Bank);
void XGpioPs_Write(XGpioPs *InstancePtr, u8 Bank, u32 Data);
void XGpioPs_SetDirection(XGpioPs *InstancePtr, u8 Bank, u32 Direction);
u32 XGpioPs_GetDirection(XGpioPs *InstancePtr, u8 Bank);
void XGpioPs_SetOutputEnable(XGpioPs *InstancePtr, u8 Bank, u32 OpEnable);
void XGpioPs_GetBankPin(u8 PinNumber, u8 *BankNumber, u8 *PinNumberInBank);
```

- 2) The “xparameters.h” header file defines the base address of each peripheral, device ID, interrupt, etc.

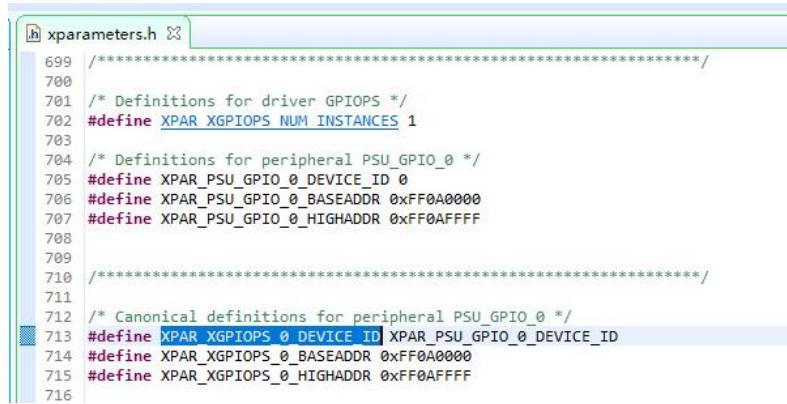


```

xparameters.h
1 #ifndef XPARETERS_H /* prevent circular inclusions */
2 #define XPARETERS_H /* by using protection macros */
3
4 /* Definition for CPU ID */
5 #define XPAR_CPU_ID 0U
6
7 /* Definitions for peripheral PSU_CORTEXA53_0 */
8 #define XPAR_PSU_CORTEXA53_0_CPU_CLK_FREQ_HZ 1066656005
9 #define XPAR_PSU_CORTEXA53_0_TIMESTAMP_CLK_FREQ 33333000
10
11 ****
12 ****
13
14 /* Canonical definitions for peripheral PSU_CORTEXA53_0 */
15 #define XPAR_CPU_CORTEXA53_0_CPU_CLK_FREQ_HZ 1066656005
16 #define XPAR_CPU_CORTEXA53_0_TIMESTAMP_CLK_FREQ 33333000
17
18 ****
19 ****
20
21 /* Definition for PSS REF CLK FREQUENCY */
22 #define XPAR_PSU_PSS_REF_CLK_FREQ_HZ 33333000U
23
24 #include "parameters_ps.h"
25
26
27 /* Number of Fabric Resets */
28 #define XPAR_NUM_FABRIC_RESETS 1
29
30 #define STDIN_BASEADDRESS 0xFFFF00000
31 #define STDOUT_BASEADDRESS 0xFFFF00000
32

```

For example, the DEVICE\_ID macro definition in the program is found in this file.



```

699 ****
700
701 /* Definitions for driver XGPIOPS */
702 #define XPAR_XGPIOPS_NUM_INSTANCES 1
703
704 /* Definitions for peripheral PSU_GPIO_0 */
705 #define XPAR_PSU_GPIO_0_DEVICE_ID 0
706 #define XPAR_PSU_GPIO_0_BASEADDR 0xFF0A0000
707 #define XPAR_PSU_GPIO_0_HIGHADDR 0xFF0AFFFF
708
709 ****
710
711 /* Canonical definitions for peripheral PSU_GPIO_0 */
712 #define XPAR_XGPIOPS_0_DEVICE_ID XPAR_PSU_GPIO_0_DEVICE_ID
713 #define XPAR_XGPIOPS_0_BASEADDR 0xFF0A0000
714 #define XPAR_XGPIOPS_0_HIGHADDR 0xFF0AFFFF
715
716

```

- 3) In the “libs” folder, contains definitions of peripheral functions, instructions for use

```

xparameters.h xpiops.c xscript.ld
94 * @note None.
95 ****
96 ****
97 #define XGPIOPS_CfgInitialize(XGpioPs *InstancePtr, const XGpioPs_Config *ConfigPtr,
98 *                           u32 EffectiveAddr)
99 {
100     s32 Status = XST_SUCCESS;
101     u8 i;
102     Xil_AssertNonvoid(InstancePtr != NULL);
103     Xil_AssertNonvoid(ConfigPtr != NULL);
104     Xil_AssertNonvoid((EffectiveAddr != (u32)0));
105     /*
106      * Set some default values for instance data, don't indicate the device
107      * is ready to use until everything has been initialized successfully.
108      */
109     InstancePtr->ISReady = 0U;
110     InstancePtr->GpioConfig.BaseAddr = EffectiveAddr;
111     InstancePtr->GpioConfig.DeviceId = ConfigPtr->DeviceId;
112     InstancePtr->Handler = (XGpioPs_Handler)StubHandler;
113     InstancePtr->Platform = XgetPlatform_Info();
114
115     /* Initialize the Bank data based on platform */
116     if (InstancePtr->Platform == (u32)XPLAT_ZYNQ_ULTRA_NP) {
117         /*
118          * Max pins in the ZynqMP GPIO device
119          * 0 - 25, Bank 0
120          * 26 - 51, Bank 1
121          * 52 - 77, Bank 2
122          * 78 - 109, Bank 3
123          * 110 - 141, Bank 4
124          * 142 - 173, Bank 5
125          */
126     InstancePtr->MaxPinNum = (u32)174;
127     InstancePtr->MaxBanks = (u8)6;
128     }
129     else if (InstancePtr->Platform == (u32)XPLAT_VERSAL) {
130         /*
131          * If InstancePtr->PmcGpio == (u32)FALSE
132          */
133         /*
134          * Max pins in the PS_GPIO devices
135          * 0 - 25, Bank 0
136          * 26-57, Bank 3
137          */
138         InstancePtr->MaxPinNum = (u32)58;
139         InstancePtr->MaxBanks = (u8)4;
140     }
141     else {
142         /*
143          * Max pins in the PMC_GPIO devices
144          * 0 - 25, Bank 0
145          * 26 - 51, Bank 1
146          * 52 - 83, Bank 3
147          * 84 - 115, Bank 4
148          */
149         InstancePtr->MaxPinNum = (u32)116;
150         InstancePtr->MaxBanks = (u8)5;
151     }
152     /*
153      * Max pins in the GPIO device
154      */

```

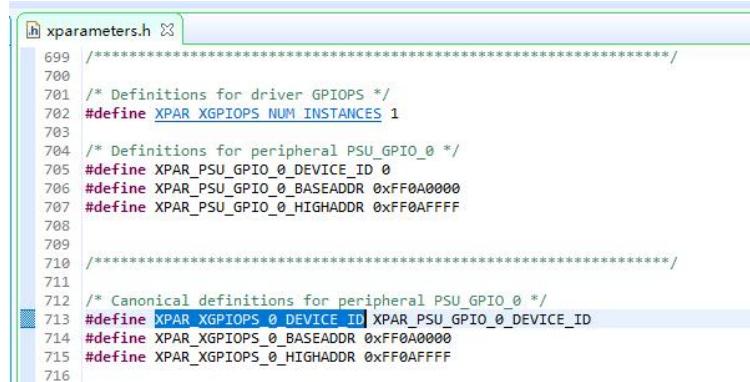
- 4) The “lscript.ld” under the “src” folder defines the available “memory” space, stack and heap space sizes, etc., which can be modified as needed.

Name	Base Address	Size
psu_ddr_0_MEMORY_0	0x0	0x7FFFFFFF
psu_ddr_1_MEMORY_0	0x80000000	0x80000000
psu_ocm_MEMORY_0	0xFFFFC000	0x4000
psu_qspi_linear_MEMORY_0	0xC0000000	0x20000000

Section Name	Memory Region
.text	psu_ddr_0_MEMORY_0
.init	psu_ddr_0_MEMORY_0
.fini	psu_ddr_0_MEMORY_0
.interp	psu_ddr_0_MEMORY_0
.note-ABI-tag	psu_ddr_0_MEMORY_0
.rodata	psu_ddr_0_MEMORY_0
.rodata1	psu_ddr_0_MEMORY_0
.sdata2	psu_ddr_0_MEMORY_0
.sbss2	psu_ddr_0_MEMORY_0
.data	psu_ddr_0_MEMORY_0
.data1	psu_ddr_0_MEMORY_0
.got	psu_ddr_0_MEMORY_0

- 5) Place the mouse cursor on the macro definition or function, press “F3” to see where it is defined, or press “Ctrl + left mouse” key to enter. For example, the following “DEVICE\_ID” can be entered into “xparameter.h”

```
48 #include <stdio.h>
49 #include "platform.h"
50 #include "xil_printf.h"
51 #include "xgpiops.h"
52 #include "sleep.h"
53
54 #define GPIO_DEVICE_ID XPAR_XGPIOPS_0_DEVICE_ID
55
```



```
#include "xparameters.h"
/*
 * Definitions for driver GPIOPS */
#define XPAR_XGPIOPS_NUM_INSTANCES 1

/* Definitions for peripheral PSU_GPIO_0 */
#define XPAR_PSU_GPIO_0_DEVICE_ID 0
#define XPAR_PSU_GPIO_0_BASEADDR 0xFF0A0000
#define XPAR_PSU_GPIO_0_HIGHADDR 0xFF0AFFFF

/*
 * Canonical definitions for peripheral PSU_GPIO_0 */
#define XPAR_XGPIOPS_0_DEVICE_ID XPAR_PSU_GPIO_0_DEVICE_ID
#define XPAR_XGPIOPS_0_BASEADDR 0xFF0A0000
#define XPAR_XGPIOPS_0_HIGHADDR 0xFF0AFFFF
```

## Part 3.5: Experimental Summary

This chapter introduces the input and output control of MIO and the use of GPIO. I believe that everyone also has a certain understanding. During the learning process, be sure to look at the documentation, combine module structure and register meaning to deepen understanding. Reference document ug1085.

## Part 4: PS Side UART Read and Write Control

The vivado project directory is "ps\_hello/vivado"

The vitis project directory is "ps\_uart/vitis"

### Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

In the previous experiment, the printing information was mainly called "xil\_printf" or "printf", but how to print the information? We still remember that the serial port was set up before printing the information. Yes, it is indeed a serial port, but how do these functions call the serial port? In fact, we can see in the "xil\_ function definition". Note that the outbyte function calls UART to print.

```
④ void xil_printf( const char8 *ctrl1, ...)  
{  
    s32 Check;  
    #if defined (_aarch64_) || defined (_arch64_)  
    s32 long_flag;  
    #endif  
    s32 dot_flag;  
  
    params_t par;  
  
    char8 ch;  
    va_list argp;  
    char8 *ctrl = (char8 *)ctrl1;  
  
    va_start( argp, ctrl1);  
  
    while ((ctrl != NULL) && (*ctrl != (char8)0)) {  
        /* move format string chars to buffer until a */  
        /* format control is found. */  
        if ((*ctrl != '%') {  
            #ifdef STDOUT_BASEADDRESS  
            outbyte(*ctrl);  
            #endif  
            ctrl += 1;  
            continue;  
        }  
    }  
}
```

Then enter the “outbyte” function, you can see that the function of the UART on the PS side is called, which can be displayed in the serial port.

```
④ void outbyte(char c) {  
    XUartPs_SendByte(STDOUT_BASEADDRESS, c);  
}
```

In addition to printing information, how to use UART for data transmission? This chapter introduces read and write control of the UART on the PS side. In the experiment, a string of characters is sent out every 1S. If data is received, an interrupt is generated and the received data is sent out again.

## Part 4.1: UART Module Introduction

The following is a block diagram of the “UART” module. Both the “TxFIFO” and the “RxFIFO” are 64 bytes.

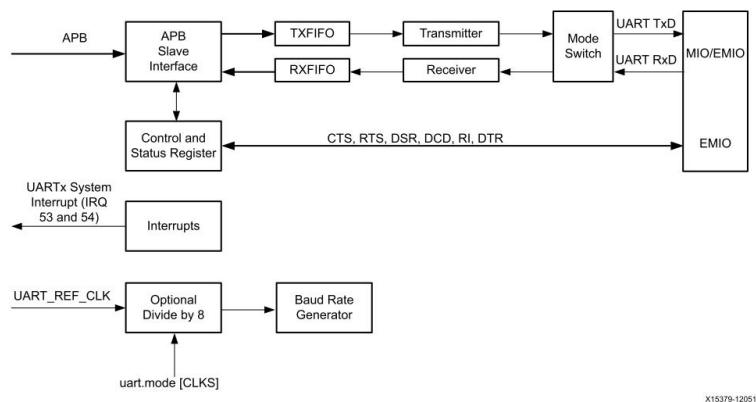


Figure 21-1: UART Controller

The following figure shows the four modes of the UART.

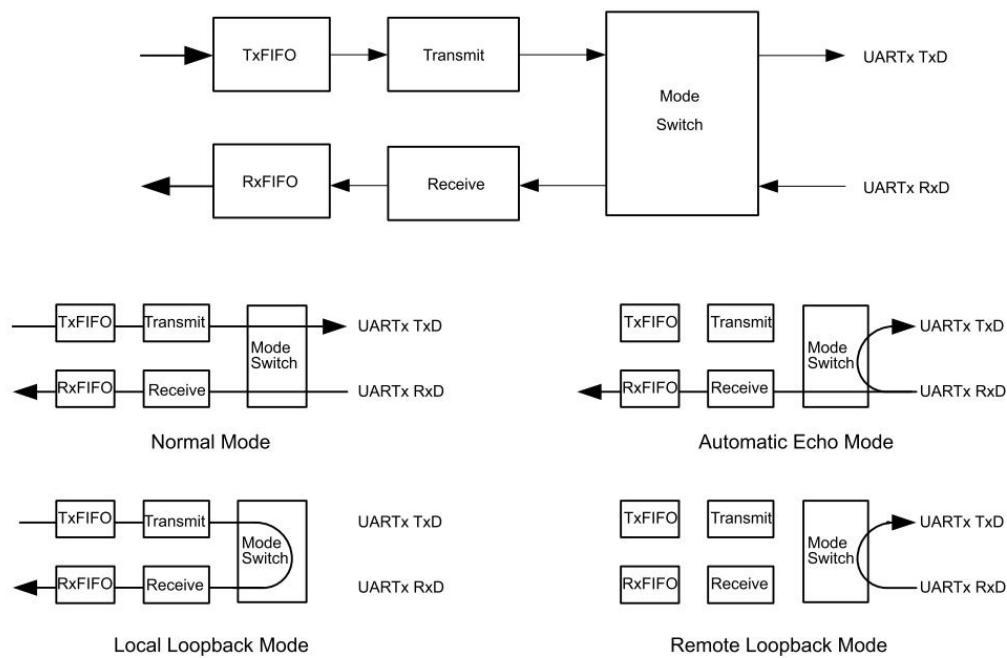


Figure 21-6: UART Mode Switch for TxD and RxD

You can use the “remote loopback mode” to test whether the physical circuit is normal, use the “API” function “**XUartPs\_SetOperMode**”

```
XUartPs_SetOperMode(&Uart_PS, XUARTPS_OPER_MODE_REMOTE_LOOP);
```

## Part 4.2: Vitis Program Development

- 1) The experimental process is as follows:

### Main program flow:

UART initialization → Set UART mode → Set data format → Set interrupt → Transmit UART data → Check if data is received → If received, transmit received data, if not, waiting for 1 second, continue to transmit data

### Interrupt program flow:

Interrupt initialization → Set the receive FIFO trigger interrupt register, set to 1, that is, receive a data interrupt → open receive trigger interrupt “REMPIY” and receive “FIFO” empty interrupt “RTRIG”

### Interrupt service routine:

Determine whether the status register is trigger or empty → clear the corresponding interrupt → trigger state, read RxFIFO data, and the empty state sets “ReceivedFlag” to 1.

- 2) In the “main” function to set the mode, you can directly call the function, set to normal mode, the data format is set to baud rate “115200” data 8bit, no parity, 1bit stop bit. “UartFormat” is defined in “uart\_parameter.h”.

```
/* Use Normal mode. */  
XUartPs_SetOperMode(&Uart_PS, XUARTPS_OPER_MODE_NORMAL);  
/* Set uart mode Baud Rate 115200, 8bits, no parity, 1 stop bit */  
XUartPs_SetDataFormat(&Uart_PS, &UartFormat) ;
```

```
XUartPsFormat UartFormat =
{
    115200,
    XUARTPS_FORMAT_8_BITS,
    XUARTPS_FORMAT_NO_PARITY,
    XUARTPS_FORMAT_1_STOP_BIT
};
```

- 3) The interrupt controller program initialization can refer to the key interrupt mode, and the usage is similar.
- 4) Set the “trigger level” to 1 in the main function, and turn on the “trigger” and “empty” interrupts.

```
/*Set receiver FIFO interrupt trigger level, here set to 1*/
XUartPs_SetFifoThreshold(UartInstancePtr,1);
/* Enable the receive FIFO trigger level interrupt and empty interrupt for the device */
XUartPs_SetInterruptMask(UartInstancePtr,XUARTPS_RXOVR|XUARTPS_RXEMPTY);
```

- 5) The transmit and receive functions of the data refer to the “XUartPs\_Send” and “XUartPs\_Rev” functions of the “UARTPS”, but they open some interrupts that are not as expected, so they are modified.

```
int UartPsSend(XUartPs *InstancePtr, u8 *BufferPtr, u32 NumBytes) ;
int UartPsRev (XUartPs *InstancePtr, u8 *BufferPtr, u32 NumBytes) ;
```

A buffer of up to 2000 bytes is set in the receive buffer and can be modified as needed

```
#define MAX_LEN    2000      /* UART receiver buffer */
u8 ReceivedBuffer[MAX_LEN] ;
```

- 6) In the interrupt service routine, add the “ReceivedBufferPtr” pointer address to the number received by “ReceivedByteNum”. If the FIFO is empty, set the “ReceivedFlag” to 1.

```
void Handler(void *CallBackRef)
{
    XUartPs *UartInstancePtr = (XUartPs *) CallBackRef ;
    u32 ReceivedCount = 0 ;
    u32 UartSrValue ;

    UartSrValue = XUartPs_ReadReg(UartInstancePtr->Config.BaseAddress, XUARTPS_SR_OFFSET) & (XUARTPS_RXOVR|XUARTPS_RXEMPTY) ;
    ReceivedFlag = 0 ;

    if (UartSrValue & XUARTPS_RXOVR) /* check if receiver FIFO trigger */
    {
        ReceivedCount = UartPsRev(&Uart_PS, ReceivedBufferPtr, MAX_LEN) ;
        ReceivedByteNum += ReceivedCount ;
        ReceivedBufferPtr += ReceivedCount ;
        /* clear trigger interrupt */
        XUartPs_WriteReg(UartInstancePtr->Config.BaseAddress, XUARTPS_ISR_OFFSET, XUARTPS_RXOVR) ;
    }
    else if (UartSrValue & XUARTPS_RXEMPTY) /*check if receiver FIFO empty */
    {
        /* clear empty interrupt */
        XUartPs_WriteReg(UartInstancePtr->Config.BaseAddress, XUARTPS_ISR_OFFSET, XUARTPS_RXEMPTY) ;
        ReceivedFlag = 1 ;
    }
}
```

***Interrupt and Status Registers***

There are two status registers that can be read by software. Both show raw status. The Chnl\_int\_sts register can be read for status and generate an interrupt. The Channel\_sts register can only be read for status.

The Chnl\_int\_sts register is sticky; once a bit is set, the bit stays set until software clears it. Write a **1** to clear a bit. This register is bit-wise AND'ed with the Inrprt\_mask mask register. If any of the bit-wise AND functions have a result = 1, then the UART interrupt is asserted to the PS interrupt controller.

**Ug 1085 UART Clear Interrupt**

- 7) In the main function, the “ReceivedFlag” and “ReceivedByteNum” are cleared, and the “ReceivedBufferPtr pointer” is reset.

```
case UART_RXCHECK : /* Check receiver flag, send received data */
{
    if (ReceivedFlag)
    {
        /* Reset receiver pointer, flag, byte number */
        ReceivedBufferPtr = ReceivedBuffer ;
        SendBufferPtr = ReceivedBuffer ;
        SendByteNum = ReceivedByteNum ;
        ReceivedFlag = 0 ;
        ReceivedByteNum = 0 ;
        UartPsSend(&Uart_PS, SendBufferPtr, SendByteNum);
    }
}
```

- 8) In the “Uart” transmit function, determine if the “TxFIFO” is full, otherwise continue to send until the count reaches “NumBytes”

```
int UartPsSend(XUartPs *InstancePtr, u8 *BufferPtr, u32 NumBytes)
{
    u32 SentCount = 0U;

    /* Setup the buffer parameters */
    InstancePtr->SendBuffer.RequestedBytes = NumBytes;
    InstancePtr->SendBuffer.RemainingBytes = NumBytes;
    InstancePtr->SendBuffer.NextBytePtr = BufferPtr;

    while (InstancePtr->SendBuffer.RemainingBytes > SentCount)
    {
        /* Fill the FIFO from the buffer */
        if (!XUartPs_IsTransmitFull(InstancePtr->Config.BaseAddress))
        {
            XUartPs_WriteReg(InstancePtr->Config.BaseAddress,
                            XUARTPS_FIFO_OFFSET,
                            ((u32)InstancePtr->SendBuffer.
                            NextBytePtr[SentCount]));

            /* Increment the send count. */
            SentCount++;
        }
    }

    /* Update the buffer to reflect the bytes that were sent from it */
    InstancePtr->SendBuffer.NextBytePtr += SentCount;
    InstancePtr->SendBuffer.RemainingBytes -= SentCount;

    return SentCount;
}
```

- 9) In the “Uart” receiving function, it is judged whether the receiving “Rx FIFO” is empty, otherwise the data is read continuously, and “NumBytes” is the number of data to be read, but if the received “FIFO” is empty, the counting does not reach this value, and the function is terminated.

```

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

    /* Setup the buffer parameters */
    InstancePtr->ReceiveBuffer.RequestedBytes = NumBytes;
    InstancePtr->ReceiveBuffer.RemainingBytes = NumBytes;
    InstancePtr->ReceiveBuffer.NextBytePtr = BufferPtr;

    /*
     * Read the Channel Status Register to determine if there is any data in
     * the RX FIFO
     */
    CsrRegister = XUartPs_ReadReg(InstancePtr->Config.BaseAddress,
                                  XUARTPS_SR_OFFSET);

    /*
     * Loop until there is no more data in RX FIFO or the specified
     * number of bytes has been received
     */
    while((ReceivedCount < InstancePtr->ReceiveBuffer.RemainingBytes)&&
          (((CsrRegister & XUARTPS_SR_RXEMPTY) == (u32)0)))
    {
        InstancePtr->ReceiveBuffer.NextBytePtr[ReceivedCount] =
            XUartPs_ReadReg(InstancePtr->Config.BaseAddress,XUARTPS_FIFO_OFFSET);

        ReceivedCount++;

        CsrRegister = XUartPs_ReadReg(InstancePtr->Config.BaseAddress,
                                      XUARTPS_SR_OFFSET);
    }
}

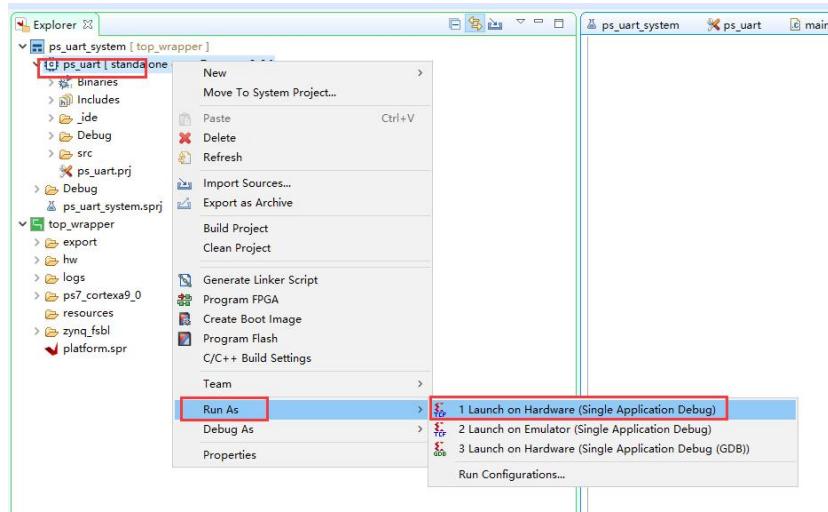
```

10) In addition to writing the program yourself, you can also import the module example from “system.mss”, refer to the program provided by Xilinx for easy learning.

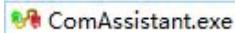
			Documentation Link	Import Examples
psu_sd_1	sdps	-	-	-
psu_serdes	generic	-	-	-
psu_siou	generic	-	-	-
psu_smmu_gpv	generic	-	-	-
psu_smmu_reg	generic	-	-	-
psu_ttc_0	ttcps	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>	
psu_ttc_1	ttcps	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>	
psu_ttc_2	ttcps	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>	
psu_ttc_3	ttcps	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>	
psu_uart_0	uartps	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>	
psu_usb_0	generic	-	-	
psu_usb_xhci_0	usbpsu	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>	

## Part 4.3: Onboard Verification

### 1) Next download the program



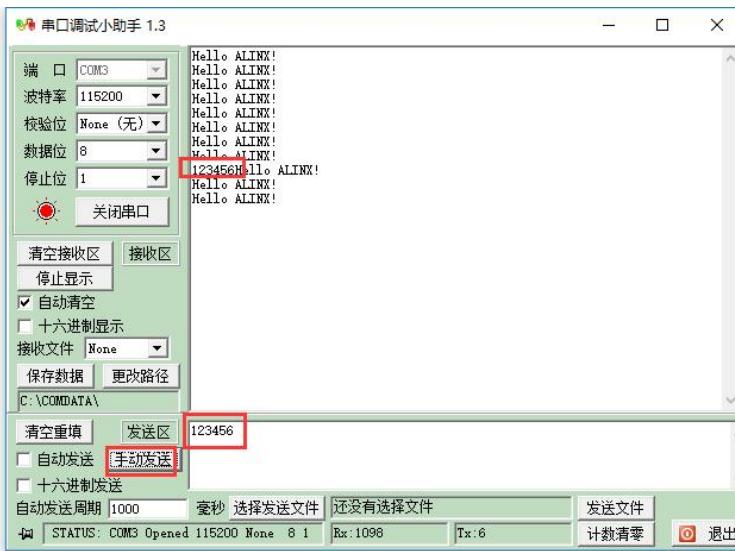
2) Open the serial debugging tool in the project directory.



3) Set the parameters as follows, open the serial port, you can see the print information.



4) Fill in the data in the sending area, click on the manual to send, you can see the data in the receiving area



## Part 4.4: Experimental Summary

This chapter has learned about the transmission and reception of UART, as well as the use of interrupts. I hope that everyone can develop good habits, read more documents, understand the principles,

and greatly improve the understanding of the system.

## Part 5: PS Side Use of CAN

The vivado project directory is "ps\_hello/vivado"

The vitis project directory is "ps\_can/vitis"

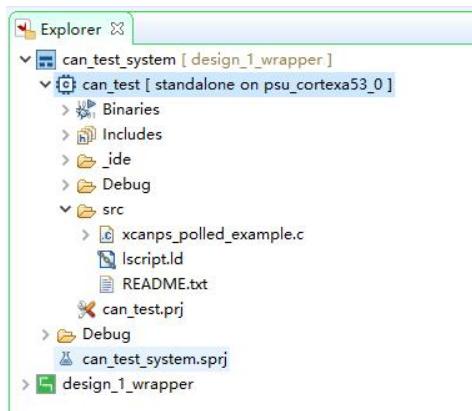
This chapter introduces the use of two CAN interfaces on the board for loopback testing.

### Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

#### Part 5.1: Vitis Program Development

Create a new project can\_test in the VITIS software. This experiment is a modification based on this example. Both CAN interfaces are set to Normal Mode and connected externally for loopback testing.



- 1) The function is actually relatively simple, mainly the initialization of the two-way CAN, and set it to Normal Mode, and then loopback.

```

int main()
{
    int Status;

    xil_printf("CAN Polled Mode Example Test \r\n");

    /*
     * Run the Can Polled example, specify the Device ID that is generated
     * in xparameters.h .
     */
    Status = CanInitial(XPAR_XCANPS_0_DEVICE_ID, &Can0); 初始化CAN0
    if (Status != XST_SUCCESS) {
        xil_printf("CAN Initial Failed\r\n");
        return XST_FAILURE;
    }

    Status = CanInitial(XPAR_XCANPS_1_DEVICE_ID, &Can1); 初始化CAN1
    if (Status != XST_SUCCESS) {
        xil_printf("CAN Initial Failed\r\n");
        return XST_FAILURE;
    }

    Status = CanLoopback(&Can0, &Can1); CAN0和CAN1 loopback
    if (Status != XST_SUCCESS) {
        xil_printf("CAN Loopback Failed\r\n");
        return XST_FAILURE;
    }

    xil_printf("Successfully ran CAN Polled Mode Example Test\r\n");
    return XST_SUCCESS;
}

```

- 2) The CanLoopback function is also relatively simple. First, CAN0 sends data, CAN1 receives data and compares data, then clears RxBuffer data, then CAN1 sends data, CAN0 receives data and compares data, and the test ends.

```

int CanLoopback(XCanPs *CanInstPtr0, XCanPs *CanInstPtr1)
{
    int Status;
    /*
     * Send a frame, receive the frame via the loop back and verify its
     * contents.
     */
    Status = SendFrame(CanInstPtr0);
    if (Status != XST_SUCCESS) {
        return Status;
    }

    Status = RecvFrame(CanInstPtr1);

    /*
     * Clear RxFrame buffer
     */
    memset(RxFrame, 0, XCANPS_MAX_FRAME_SIZE) ;

    /*
     * Send a frame, receive the frame via the loop back and verify its
     * contents.
     */
    Status = SendFrame(CanInstPtr1);
    if (Status != XST_SUCCESS) {
        return Status;
    }

    Status = RecvFrame(CanInstPtr0);

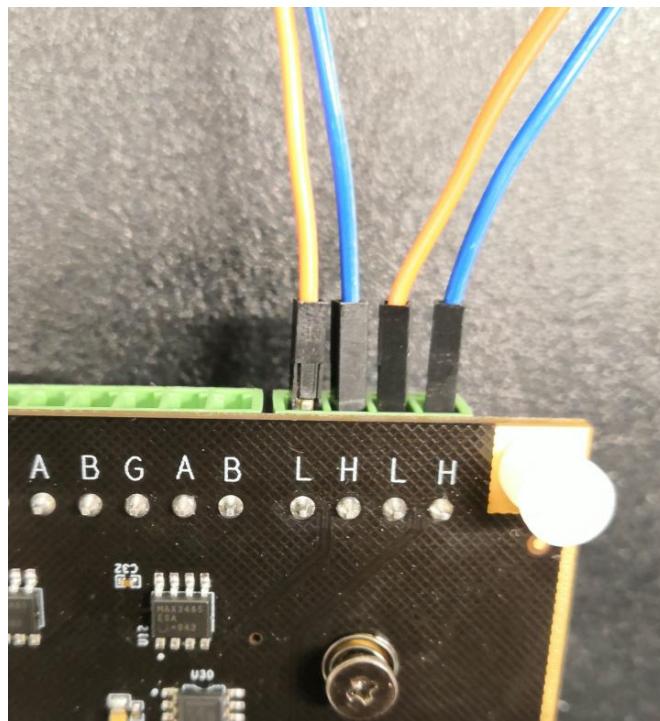
    return Status;
}

```

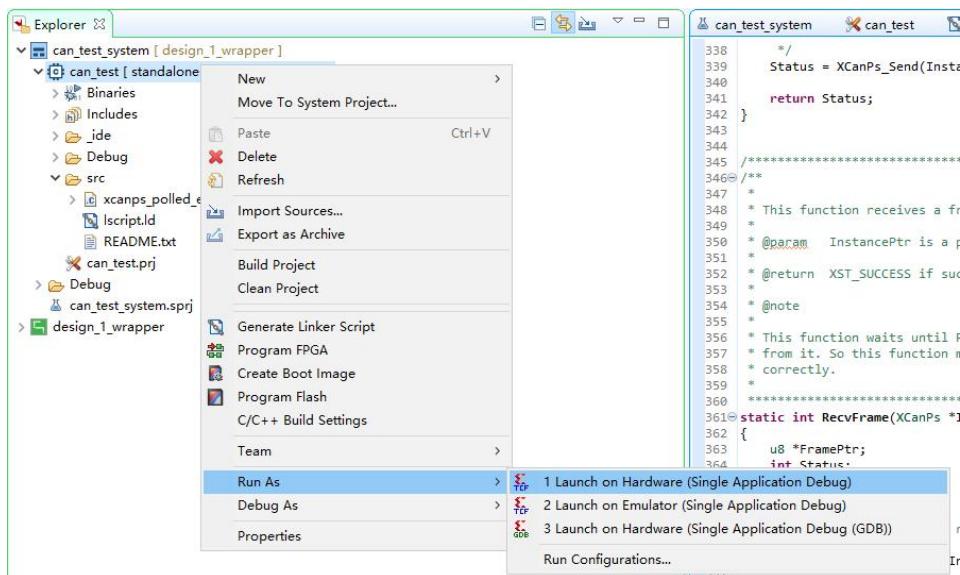
## Part 5.2: Download and Test

- 1) Connect H and L of CAN0 with H and L of CAN1 by using du-tie

wire



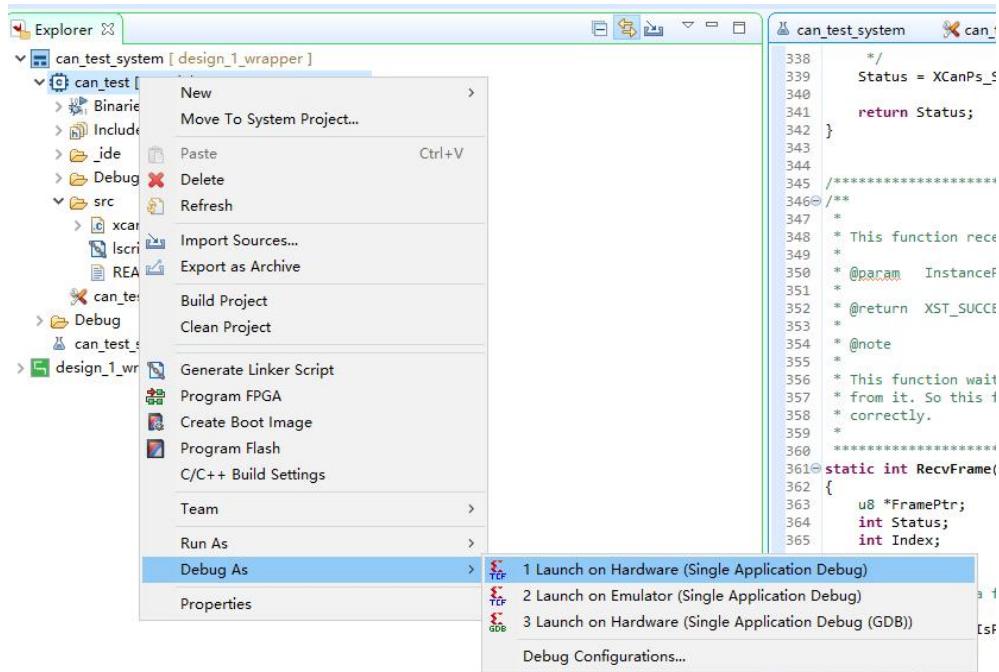
## 2) Download program



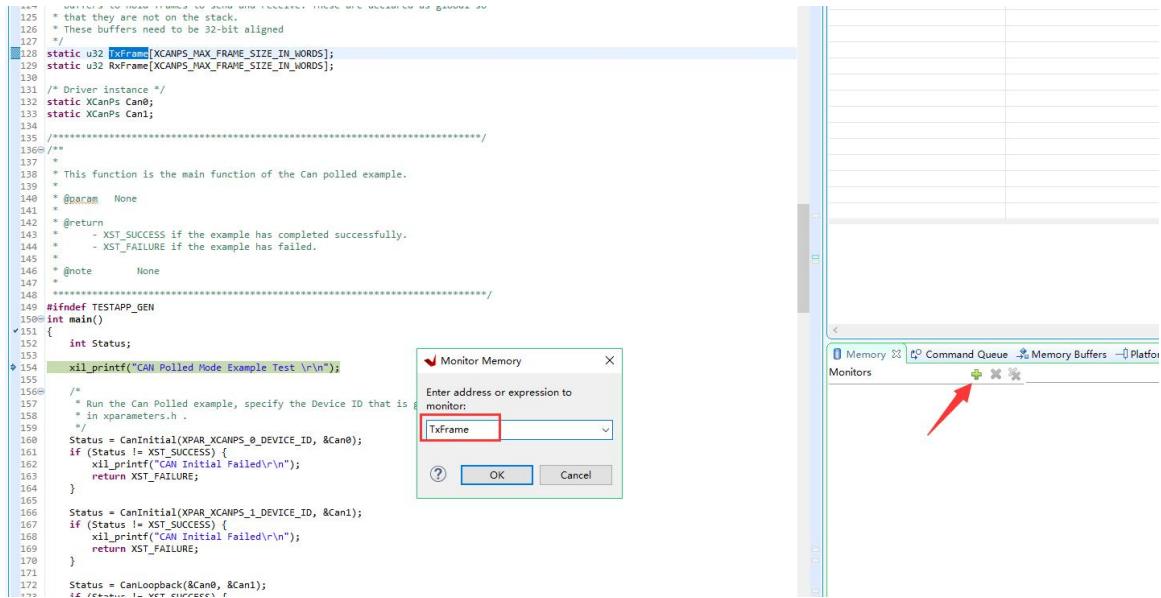
## 3) You can see the print information on the serial port

```
CAN Polled Mode Example Test
Successfully ran CAN Polled Mode Example Test
```

## 4) We can also use Debug to see the running status of the program, select Debug As to enter the Debug mode



## 5) Add the sent buffer in Memory



In the same way, add RxFrame and observe the data in the two buffers

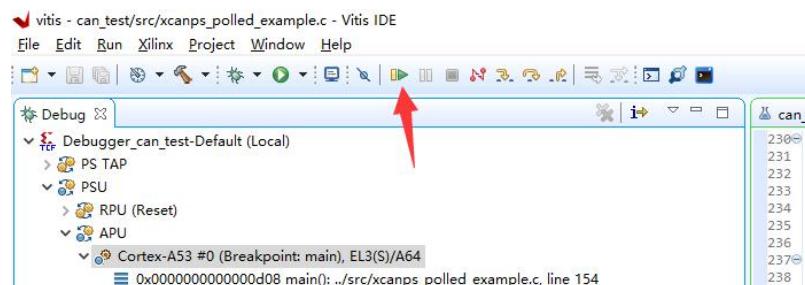
Monitors	RxFrame : 0xC0F8 <Hex Integer>	New Renderings...
TxFrame	000000000000C0F0	00000000
RxFrame	000000000000C100	00000000
	000000000000C110	00000000
	000000000000C120	00000000
	000000000000C130	00000000
	000000000000C140	00000000
	000000000000C150	00000000
	000000000000C160	00000000
	000000000000C170	00000000
	000000000000C180	00000000
	000000000000C190	00000000
	000000000000C1A0	00000000
	000000000000C1B0	00000000
	000000000000C1C0	00000000
	000000000000C1D0	00000000
	.....	.....

## 6) In the CanLoopback function, add a breakpoint at SendFrame

```

255
256 int CanLoopback(XCanPs *CanInstPtr0, XCanPs *CanInstPtr1)
257 {
258     int Status;
259     /*
260     * Send a frame, receive the frame via the loop back and verify its
261     * contents.
262     */
263     Status = SendFrame(CanInstPtr0);
264     if (Status != XST_SUCCESS) {
265         return Status;
266     }
267
268     Status = RecvFrame(CanInstPtr1);
269
270     /*
271     * Clear RxFrame buffer
272     */
273     memset(RxFrame, 0, XCANPS_MAX_FRAME_SIZE) ;
274
275     /*
276     * Send a frame, receive the frame via the loop back and verify its
277     * contents.
278     */
279     Status = SendFrame(CanInstPtr1);
280     if (Status != XST_SUCCESS) {
281         return Status;
282     }
283
284     Status = RecvFrame(CanInstPtr0);
285
286     return Status;
287 }
288 }
```

## 7) Click Run



## 8) The program runs to here, and then set a breakpoint at RecvFrame

```

255
256 int CanLoopback(XCanPs *CanInstPtr0, XCanPs *CanInstPtr1)
257 {
258     int Status;
259     /*
260     * Send a frame, receive the frame via the loop back and verify its
261     * contents.
262     */
263     Status = SendFrame(CanInstPtr0);
264     if (Status != XST_SUCCESS) {
265         return Status;
266     }
267
268     Status = RecvFrame(CanInstPtr1);
269
270     /*
271     * Clear RxFrame buffer
272     */
273 }
```

9) Click Run again, you can see the data generated by TxFrame

Address	0 - 3	4 - 7	8 - B	C - F
0000000000C0B0	00000000	00000000	FA000000	80000000
0000000000C0C0	03020100	07060504	00000000	00000000
0000000000C0D0	00000000	00000000	00000000	00000000
0000000000C0E0	00000000	00000000	00000000	00000000
0000000000C0F0	00000000	00000000	00000000	00000000

10) Press F6 to run in a single step, and you can see the data received by RxFrame, thus completing the data transmission and reception from CAN0 to CAN1.

Address	0 - 3	4 - 7	8 - B	C - F
0000000000C0F0	00000000	00000000	FA000000	8000274C
0000000000C100	03020100	07060504	00000000	00000000
0000000000C110	00000000	00000000	00000000	00000000
0000000000C120	00000000	00000000	00000000	00000000

11) Similarly, you can try the data transmission from CAN1 to CAN0

```

256 int CanLoopback(XCanPs *CanInstPtr0, XCanPs *CanInstPtr1)
257 {
258     int Status;
259     /*
260     * Send a frame, receive the frame via the loop back and verify its
261     * contents.
262     */
263     Status = SendFrame(CanInstPtr0);
264     if (Status != XST_SUCCESS) {
265         return Status;
266     }
267
268     Status = RecvFrame(CanInstPtr1);
269
270     /*
271     * Clear RxFrame buffer
272     */
273     memset(RxFrame, 0, XCANPS_MAX_FRAME_SIZE) ;
274
275     /*
276     * Send a frame, receive the frame via the loop back and verify its
277     * contents.
278     */
279     Status = SendFrame(CanInstPtr1);
280     if (Status != XST_SUCCESS) {
281         return Status;
282     }
283
284     Status = RecvFrame(CanInstPtr0);
285
286     return Status;
287 }
288 }
```

# Part 6: PS Side Use of I2C

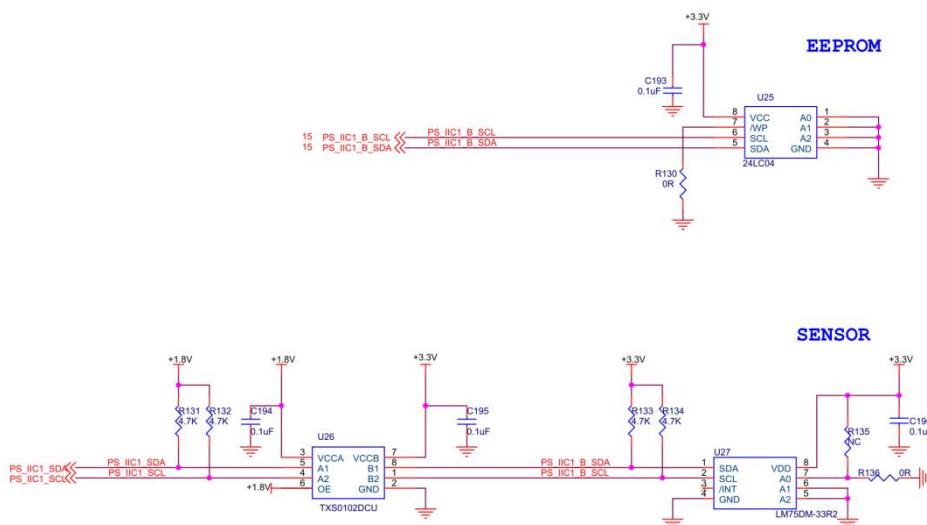
**The vivado project directory is "ps\_hello/vivado"**

**The vitis project directory is "ps\_i2c/vitis"**

# Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

In the schematic diagram, the I2C1 on the PS side is connected to two peripherals, namely EEPROM, temperature sensor. This chapter introduces how to use I2C to read and write peripherals.



## Part 6.1: Vitis Program Development

### Part 6.1.1: Temperature Sensor Test

The temperature sensor uses LM75. The following is its register description. Just read the value of the Temperature register, so its address is 0x48+r/w

**Table 1. Slave Address**

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
1	0	0	1	A2	A1	A0	R/W

**Table 2. Register Functions**

REGISTER NAME	ADDRESS (hex)	POR STATE (hex)	POR STATE (binary)	POR STATE (°C)	READ/ WRITE
Temperature	00	000X	0000 0000 XXXX XXXX	—	Read only
Configuration	01	00	0000 0000	—	R/W
THYST	02	4BX0	0100 1011 XXXX XXXX	75	R/W
Tos	03	500X	0101 0000 XXXX XXXX	80	R/W

X = Don't care.

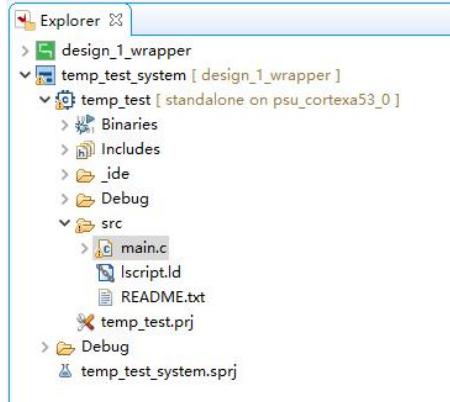
And its data has two bytes, the first byte is the integer bit, the second byte is the decimal place, the highest bit is 1, that is, 0.5 degrees Celsius, if it is 0, the decimal place is 0

**Table 4. Temperature Data Output Format**

TEMPERATURE (°C)	DIGITAL OUTPUT	
	BINARY	HEX
+125	0111 1101 XXXX XXXX	7D0X
+25	0001 1001 XXXX XXXX	190X
+0.5	0000 0000 1XXX XXXX	008X
0	0000 0000 XXXX XXXX	000X
-0.5	1111 1111 1XXX XXXX	FF8X
-25	1110 0111 XXXX XXXX	E70X
-55	1100 1001 XXXX XXXX	C90X

X = Don't care.

## 1) Create a new Vitis project



## 2) In main.c, the i2c\_init function is used for i2c initialization and rate setting

```

int i2c_init(XIicPs *Iic, short DeviceID ,u32 IIC_SCLK_RATE)
{
    XIicPs_Config *Config;
    int Status;
    /*
     * Initialize the Can device.
     */
    Config = XIicPs_LookupConfig(DeviceID);
    if (NULL == Config) {
        xil_printf("XIicPs_LookupConfig failure\r\n");
        return XST_FAILURE;
    }

    Status = XIicPs_CfgInitialize(Iic, Config, Config->BaseAddress);
    if (Status != XST_SUCCESS) {
        xil_printf("XIicPs_CfgInitialize failure\r\n");
        return XST_FAILURE;
    }
    /*
     * Set i2c clock rate
     */
    XIicPs_SetSclk(Iic, IIC_SCLK_RATE);
    while (XIicPs_BusIsBusy(Iic)); // Wait
    return XST_SUCCESS;
}

```

- 3) The temp\_read function sets the read data to 2 bytes, which is used to read i2c data

```

int i2c_init(XIicPs *Iic, short DeviceID ,u32 IIC_SCLK_RATE)
{
    XIicPs_Config *Config;
    int Status;
    /*
     * Initialize the Can device.
     */
    Config = XIicPs_LookupConfig(DeviceID);
    if (NULL == Config) {
        xil_printf("XIicPs_LookupConfig failure\r\n");
        return XST_FAILURE;
    }

    Status = XIicPs_CfgInitialize(Iic, Config, Config->BaseAddress);
    if (Status != XST_SUCCESS) {
        xil_printf("XIicPs_CfgInitialize failure\r\n");
        return XST_FAILURE;
    }
    /*
     * Set i2c clock rate
     */
    XIicPs_SetSclk(Iic, IIC_SCLK_RATE);
    while (XIicPs_BusIsBusy(Iic)); // Wait
    return XST_SUCCESS;
}

void temp_read(XIicPs *InstancePtr, u8 *rd_data, char IIC_ADDR)
{
    /*
     * i2c receiver
     */
    XIicPs_MasterRecvPolled(InstancePtr, rd_data, 2, IIC_ADDR);
    while (XIicPs_BusIsBusy(InstancePtr));
}

```

- 4) The program is very simple. According to the received data value, it will be printed out every second and displayed. It should be noted that the device address of ZYNQ is 7bit, for example, the address of LM75 is 0x48+r/w, that is, the first 7 bits of 0x48 are taken.

```

int main()
{
    //temperature point
    float point ;
    //temperature
    float temp ;

    //Initial i2c
    i2c_init(&rtc_i2c, XPAR_XIICPS_0_DEVICE_ID ,100000) ;//100KHz

    while(1){

        //read temperature
        temp_read(&rtc_i2c, Readbuf, 0x48) ;
        //if bit 7 equals to 1, point is 0.5C, or 0
        if (Readbuf[1] & 0x80)
            point = 0.5 ;
        else
            point = 0 ;

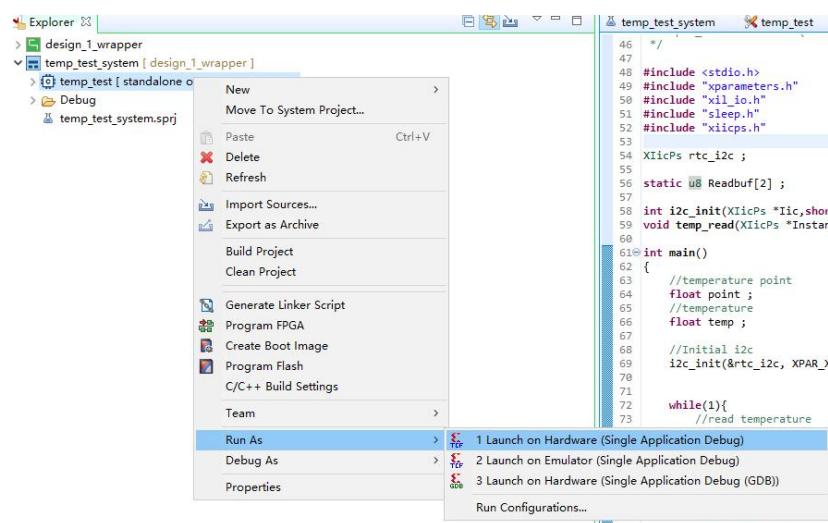
        //if bit7 equals to 1, then it is negative value
        if (Readbuf[0] & 0x80){
            temp = (float)(Readbuf[0]-256) + point ;
        }
        else{
            temp = (float)(Readbuf[0]) + point ;
        }

        printf("current temp is:%.1f\t\n",temp);

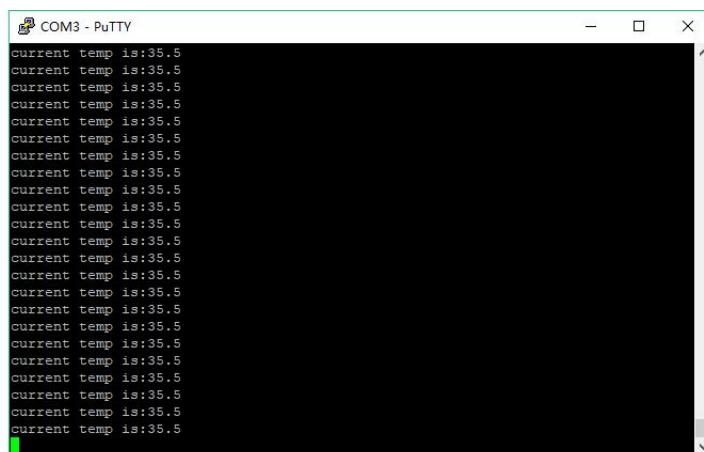
        sleep(1) ;
    }
    return 0;
}

```

## 5) Download the program

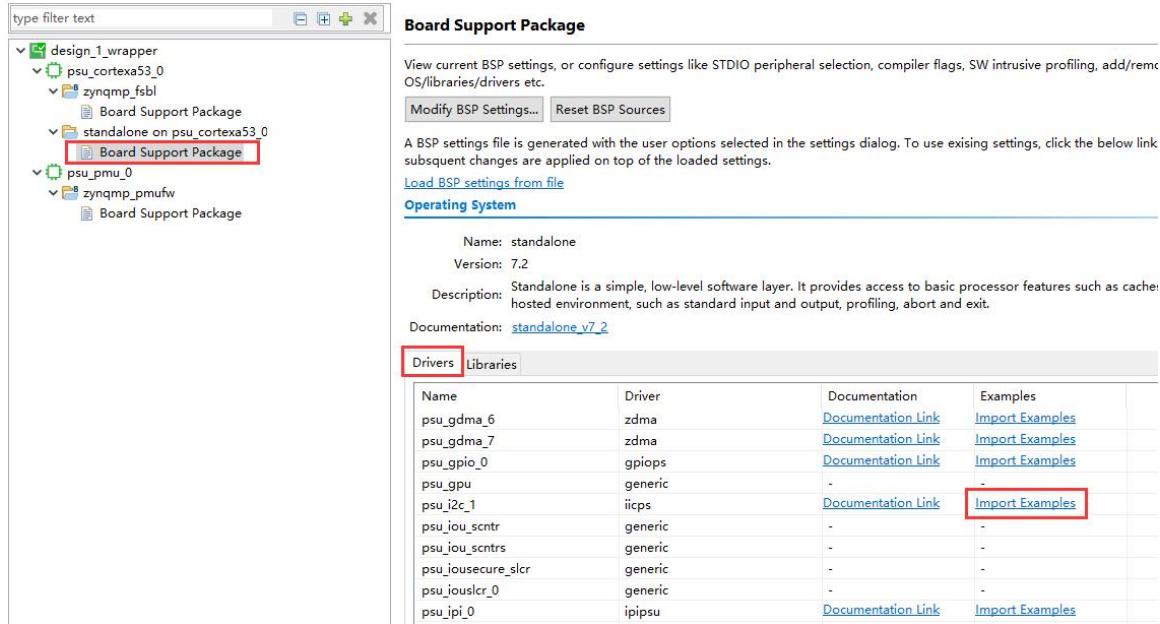


6) The serial port print information is as follows

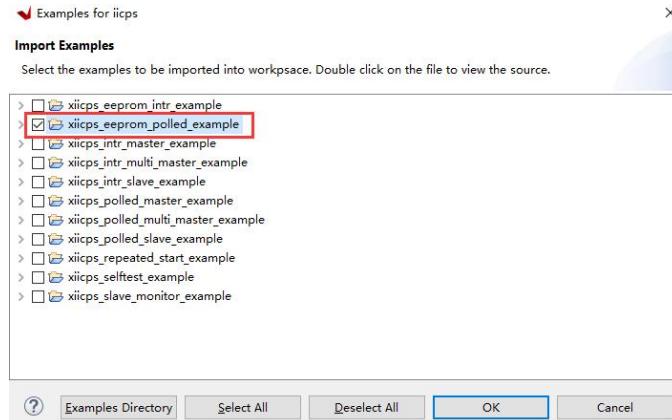


## Part 6.1.2: EEPROM Read and Write

### 1) Import the example project



### 2) Import the xiicps\_eeprom\_polled\_example project



The EEPROM program is relatively simple. You can read the specific code by yourself. I won't go into details here. The following is only for the program. The functions and some key points of the "are introduced:

- 3) The EEPROM device address is defined in the front of the program. This address is the device address for the system to access external IIC peripherals. Here, the EEPROM address is 0x54, which is equivalent to 8bit 0xA8.

```
/* **Searching for the required EEPROM Address and use
 * their own EEPROM Address in the below array list*
 u16 EepromAddr[] = {0x54,0x55,0};
```

The device address of the EEPROM can be found in the chip manual of the 24LC04. The upper 4 bits are A, and the last 3 bits are the Block address. Because 24LC04 has only 2 blocks, the upper 2 bits of the Block Address are invalid.

Operation	Control Code	Block Select	R/W
Read	1010	Block Address	1
Write	1010	Block Address	0

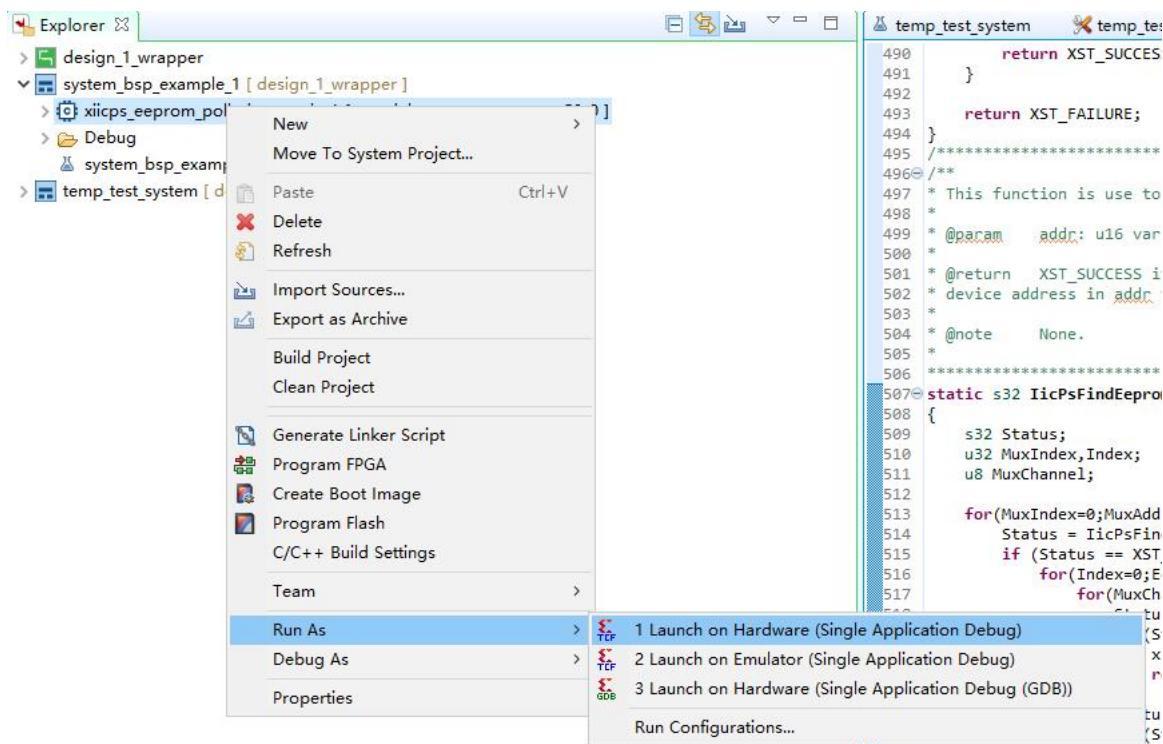
- 4) Since the address of EEPROM is 1 byte, modify it as follows in the program

```
513@ static s32 IicPsFindEeprom(u16 *Eeprom_Addr, u32 *PageSize)
514 {
515     s32 Status;
516     u32 MuxIndex, Index;
517     u8 MuxChannel;
518
519     for(MuxIndex=0;MuxAddr[MuxIndex] != 0;MuxIndex++){
520         Status = IicPsFindDevice(MuxAddr[MuxIndex]);
521         if (Status == XST_SUCCESS) {
522             for(Index=0;EepromAddr[Index] != 0;Index++) {
523                 for(MuxChannel = MAX_CHANNELS; MuxChannel > 0x0; MuxChannel = MuxChannel >> 1) {
524                     Status = MuxInitChannel(MuxAddr[MuxIndex], MuxChannel);
525                     if (Status != XST_SUCCESS) {
526                         xil_printf("Failed to enable the MUX channel\r\n");
527                         return XST_FAILURE;
528                     }
529                     Status = FindEepromDevice(EepromAddr[Index]);
530                     if (Status == XST_SUCCESS) {
531                         *Eeprom_Addr = EepromAddr[Index];
532                         *PageSize = PAGE_SIZE_16;
533                         return XST_SUCCESS;
534                     }
535                 }
536             }
537         }
538     }
539     for(Index=0;EepromAddr[Index] != 0;Index++) {
540         Status = IicPsFindDevice(EepromAddr[Index]);
541         if (Status == XST_SUCCESS) {
542             *Eeprom_Addr = EepromAddr[Index];
543             *PageSize = PAGE_SIZE_16; ←
544             return XST_SUCCESS;
545         }
546     }
547     return XST_FAILURE;
548 }
```

- 5) The program flow is as follows:

ReadBuffer is cleared, WriteBuffer is assigned FF → write 16 bytes to EEPROM → read 16 bytes of EEPROM to ReadBuffer → check whether it is correct → Readbuffer is cleared, WriteBuffer is assigned 10~25 → write 16 bytes to EEPROM → read 16 bytes Section to ReadBuffer → check whether it is correct → return

- 6) Download program



## 7) Serial result

```
IIC EEPROM Polled Mode Example Test
Successfully ran IIC EEPROM Polled Mode Example Test
```

- 8) You can also debug according to the Debug method in the chapter "PS Side Use of CAN"

## Part 7: PS Side Use of Display Port

The vivado project directory is "ps\_hello/vivado"

The vitis project directory is "ps\_dp/vitis"

This chapter introduces the use of DisplayPort on the PS side.

The Vivado project is still based on "ps\_hello"

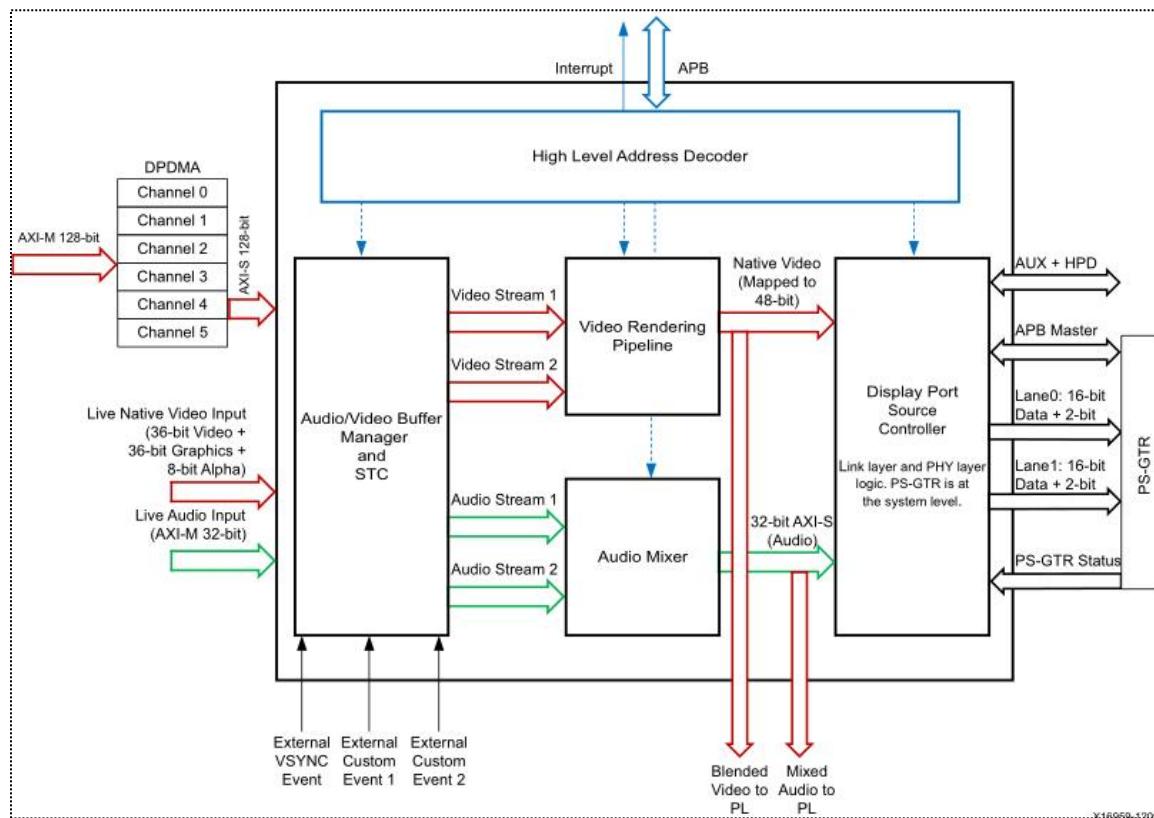
### Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

#### Part 7.1: Interface Introduction

DisplayPort v1.2 protocol supports 4 lanes of 5.4G, but this controller only supports two lanes, and the maximum resolution is 4096\*2160@30.

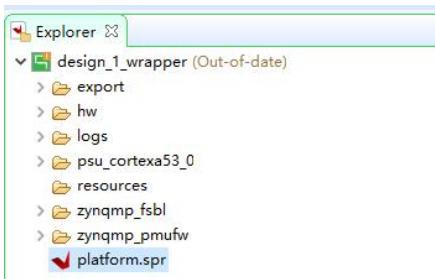
The controller data interface is as follows:



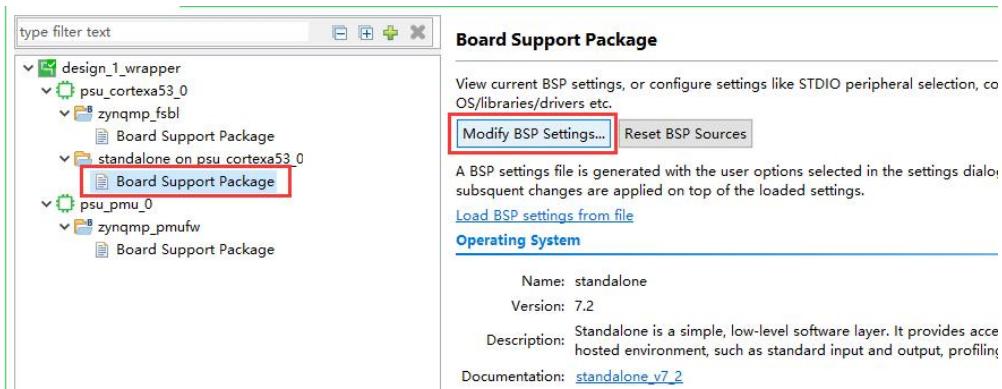
In the figure, AXI-M is used to read the video and audio data in the memory, here called non-real-time audio and video, DPDMA has six channels, of which 3 channels are used for video, 1 channel is used for graphics, and 2 channels are used for audio.

## Part 7.2: Example Project Introduction

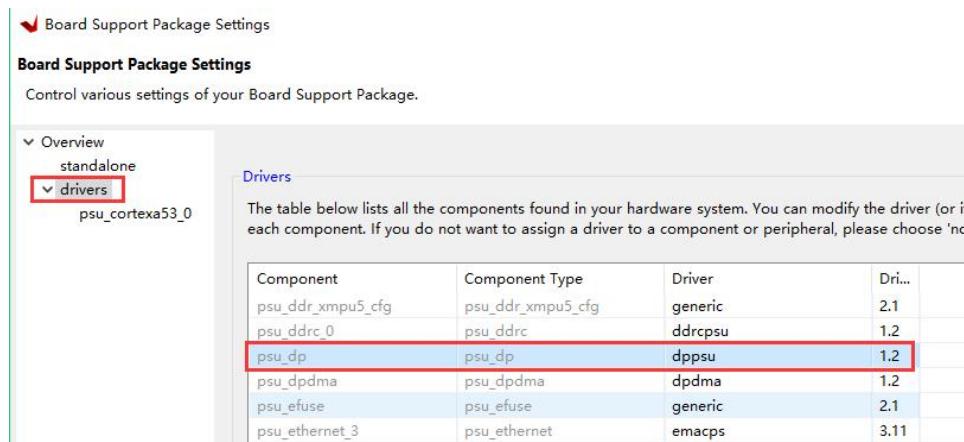
- 1) The process of creating a new platform will not be introduced. It has been introduced in "PS Side RTC Interrupt Experiment".



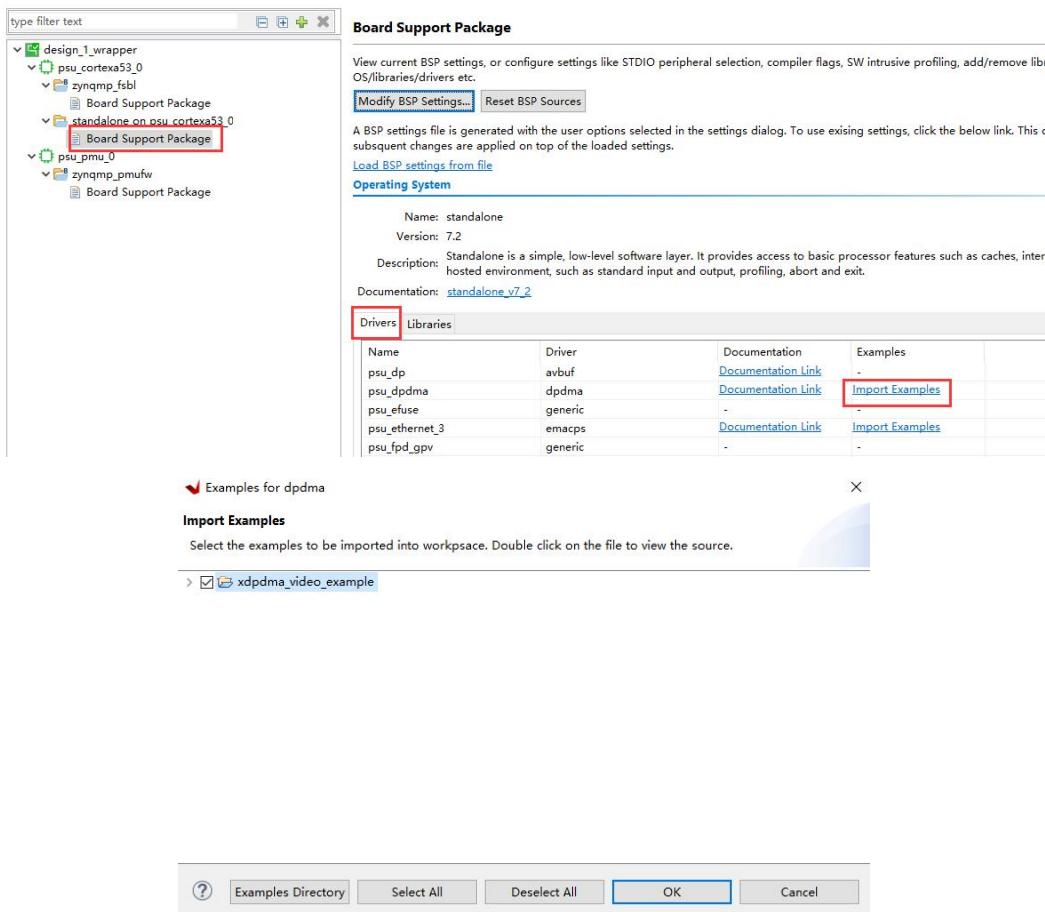
- 2) Configure BSP



And change the psu\_dp driver to dppsu, then click OK



- 3) Import example project

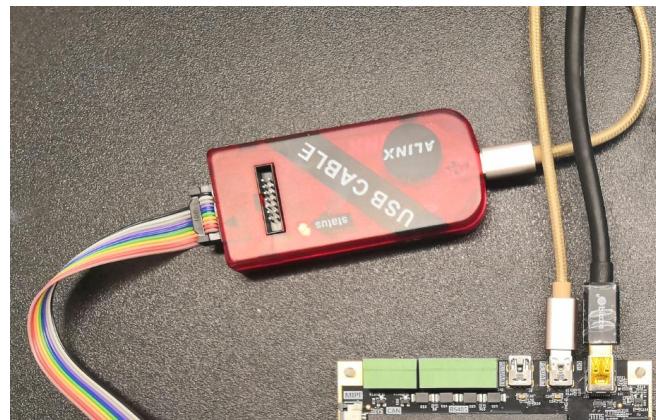


- 4) The example defaults to 1080P, RGBA display, you can change the Alpha value of RGBA to FF to make the display effect better, save it, and compile the project.

```
/*
*****
 * u8 *GraphicsOverlay(u8* Frame, Run_Config *RunCfgPtr)
 {
    u64 Index;
    u32 *RGBA;
    RGBA = (u32 *) Frame;
    /*
     * Red at the top half
     * Alpha = 0x0F
     */
    for(Index = 0; Index < (BUFFERSIZE/4) /2; Index++) {
        RGBA[Index] = 0xFF0000FF;
    }
    for(; Index < BUFFERSIZE/4; Index++) {
        /*
         * Green at the bottom half
         * Alpha = 0xFe
         */
        RGBA[Index] = 0xFF00FF00;
    }
    return Frame;
}
```

### Part 7.3: On-board verification

Connect to the MINI DP interface on the FPGA development board



After downloading, the display effect is as follows



In the serial port tool, you can see that the DP port has been trained and successfully operated.

```
DPDMA Generic Video Example Test
Generating Overlay.....
HPD event ..... ! Connected.
Lane count =      2
Link rate =      20

Starting Training...
      ! Training succeeded.
DONE!
..... HPD event
Successfully ran DPDMA Video Example Test
```

## Part 8: PS Side SD Card Read and Write

The vivado project directory is "ps\_hello/vivado"

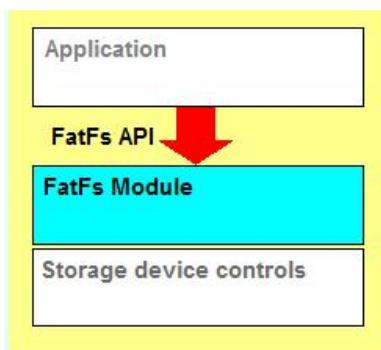
The vitis project directory is "ps\_sd/vitis"

This chapter describes the use of the “FatFs” file system module to read the “BMP” picture of the “SD” card and display it via “DP”.

### Part 8.1: FatFs Introduction

“FatFs” is a general-purpose file system module for implementing “FAT” file systems in small embedded systems. “FatFs” is written to follow “ANSI C” and therefore does not depend on the hardware platform. It can be embedded in inexpensive microcontrollers such as the 8051, PIC, AVR, SH, Z80, H8, ARM, etc. without any modifications.

The application calls the “FatFs” system module through an “API” function to control the SD card storage devices.



The FatFs system provides a number of API functions, and we've listed the API functions that will be used in our routines below.

F\_mount - register/logout a work area

F\_open - open/create a file

F\_close - close a file

F\_read - read the file

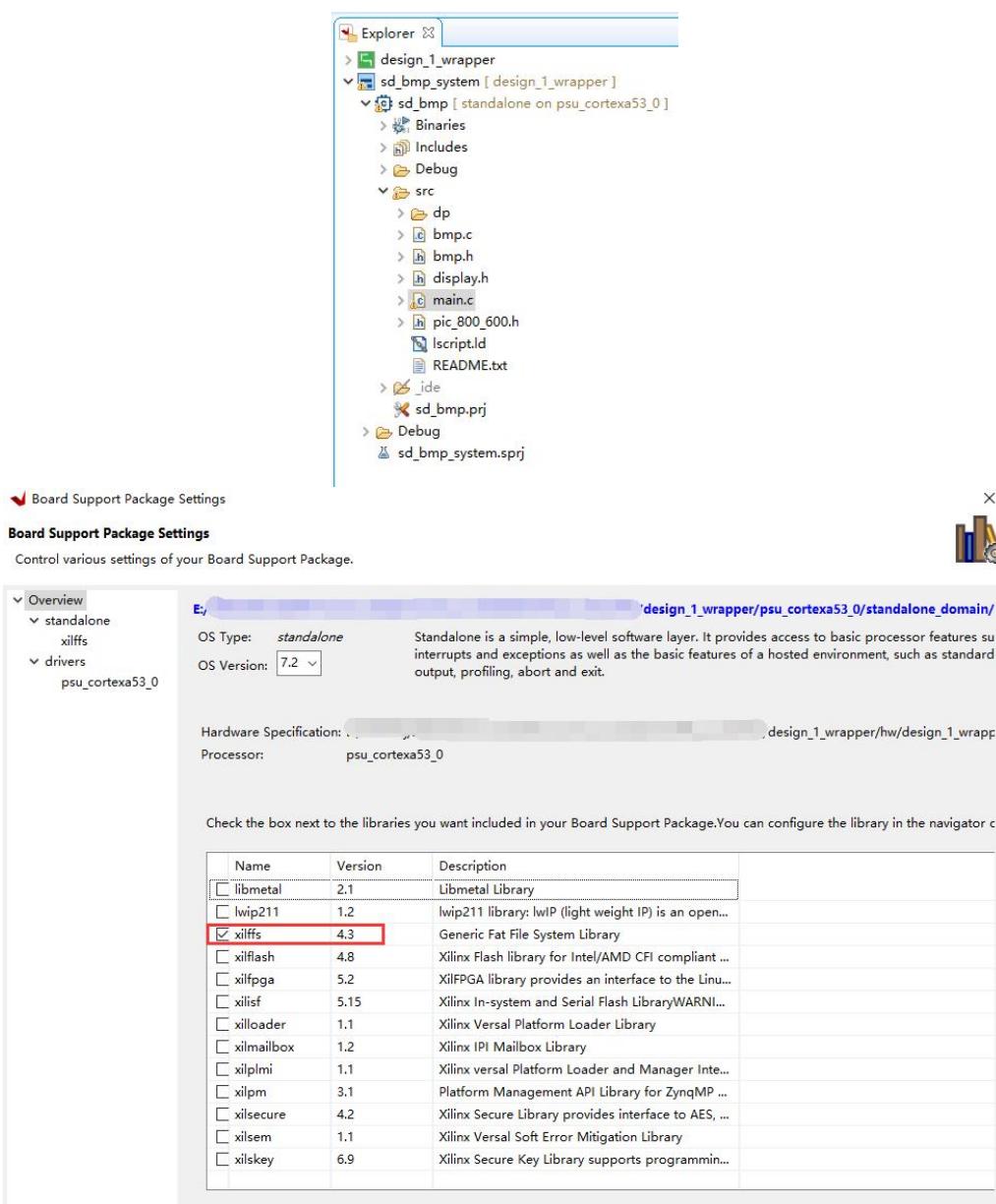
F\_write - write file

For an introduction and explanation of API functions, you can refer to the following website for a deeper understanding. This website gives instructions and examples for each API function.

[http://elm-chan.org/fsw/ff/00index\\_e.html](http://elm-chan.org/fsw/ff/00index_e.html)

## Part 8.2: Vitis program development

- 1) Open the “Vitis” software, we have generated a “sd\_bmp” project for everyone. Here you need to configure the properties of the bsp support package, select the “xilffs” in the “Board Support Package Settings”, and enable the project to support the “xilffs” file system



About the “xilffs” library is the “FAT” file system support package provided by Xilinx. Users can call the API functions in the library to implement operations on “SD/eMMC” and other devices. The “xilffs” library mainly contains the “File System Files” and “Glue Layer Files” of “FAT”

- 2) For the introduction and application of the xilffs library, you can refer to the following Xilinx official website link:

<http://www.wiki.xilinx.com/xilffs>

- 3) Next we look at the project code for “sd bmp”. In the project program code, we need to read out the “bmp” format image data stored in the SD card, remove the image header and put it into the “DP” display buffer, and then display the image in the DP Monitor.
- 4) In the “main.c” file, we added a “bmp\_read” function. In this function, we first use the “f\_open” function to open the “bmp” image file in the SD card. Then read the first 54 bytes of this file, because the first 54 bytes of the BMP image file are the image header file, which contains the pixel size information of the image. Then read image data one by line is stored in the DP frame display buffer.

Since the storage of the “BMP” is upside down, the order is adjusted in the bmp\_read function and stored in the “frame” buffer.

```

1 #include "xil_types.h"
2 #include "ff.h"
3 #include "stdio.h"
4 #include "string.h"
5
6 unsigned char read_line_buf[1920 * 3];
7 unsigned char Write_line_buf[1920 * 3];
8 void bmp_read(char * bmp,u8 *frame,u32 stride, FIL *fil)
9 {
10     short y,x;
11     short Ximage;
12     short Yimage;
13     u32 iPixelAddr = 0;
14     HRESULT res;
15     unsigned char TMPBUF[64];
16     unsigned int br;
17
18     res = f_open(fil, bmp, FA_OPEN_EXISTING | FA_READ);
19     if(res != FR_OK)
20     {
21         printf("error: f_open Failed!\r\n");
22         return ;
23     }
24     res = f_read(fil, TMPBUF, 54, &br);
25     if(res != FR_OK)
26     {
27         f_close(fil);
28         printf("Failed to Read!\r\n");
29         return ;
30     }
31     Ximage=(unsigned short int)TMPBUF[19]*256+TMPBUF[18];
32     Yimage=(unsigned short int)TMPBUF[23]*256+TMPBUF[22];
33     iPixelAddr = (Yimage-1)*stride ;
34
35     for(y = 0; y < Yimage ; y++)
36     {
37         f_read(fil, read_line_buf, Ximage * 3, &br);
38         for(x = 0; x < Ximage; x++)
39         {
40             frame[x * 4 + iPixelAddr + 0] = read_line_buf[x * 3 + 2];
41             frame[x * 4 + iPixelAddr + 1] = read_line_buf[x * 3 + 1];
42             frame[x * 4 + iPixelAddr + 2] = read_line_buf[x * 3 + 0];
43             frame[x * 4 + iPixelAddr + 3] = 0xff ;
44         }
45         iPixelAddr -= stride;
46     }
47     f_close(fil);
48     printf("BMP read successfully!\r\n");
49 }

```

- 5) At the same time, we also prepared the “BMP” file header structure. As well as some common resolution image header settings, placed in the “bmp.h” file.

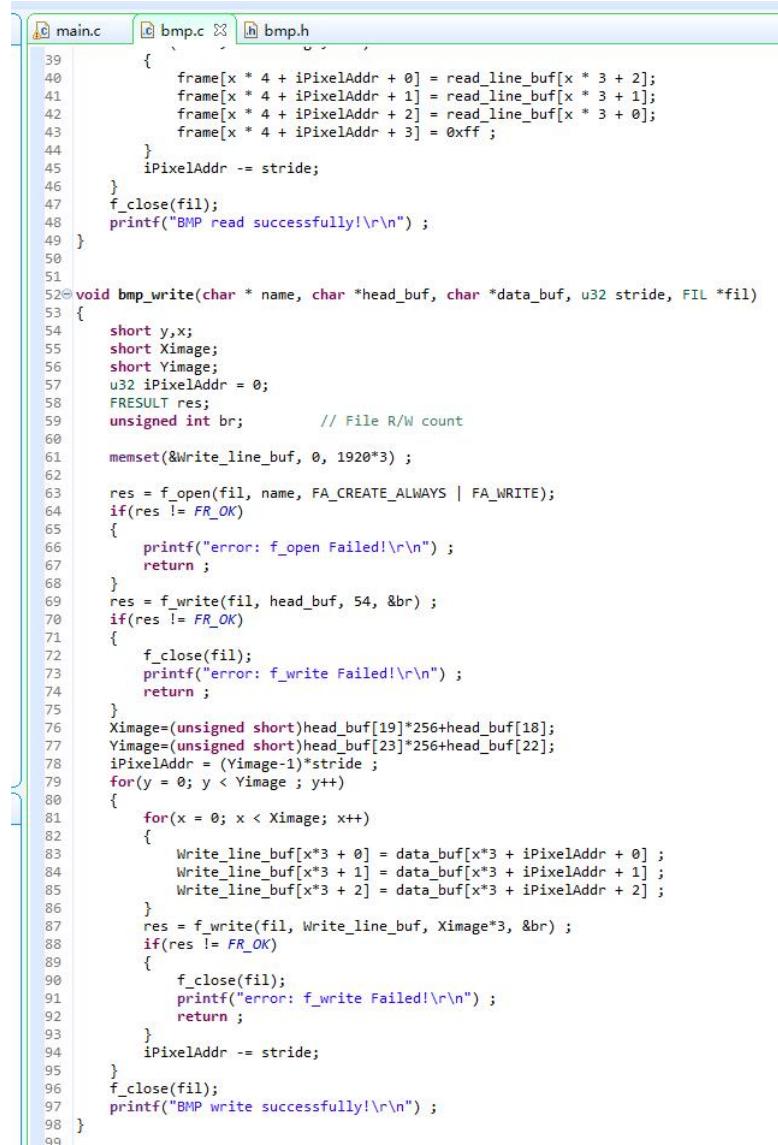
```

19     char pixel_width[2]; /* pixel width, 24bit, 32bit, 16bit and etc */
20     char compress[4]; /* compressed information 0: not compressed 1: 8bit RLE compress */
21     char bm_bytes[4]; /* bmp frame total length in byte, for 800*600,24bit, 800*600*3 */
22     char meter_width[4]; /* bmp frame width in pixel/meter, could be ignored */
23     char meter_height[4]; /* bmp frame height in pixel/meter, could be ignored */
24     char color_index[4]; /* color index number, if set 0, will use all color */
25     char index_num[4]; /* importance color index number, if set 0, all are important */
26 }BmpMode;
27
28
29 static const BmpMode BMODE_640x480 = {
30     .bm_header = {0x42, 0x4d},
31     .bm_len = {0x36, 0x10, 0x00, 0x00}, /* file length 921600+54 bytes */
32     .reserved = {0x00, 0x00, 0x00, 0x00},
33     .offset = {0x36, 0x00, 0x00, 0x00}, /* 54 bytes */
34     .bm_infolen = {0x28, 0x00, 0x00, 0x00}, /* 40 bytes */
35     .bm_width = {0x80, 0x02, 0x00, 0x00}, /* width 640 */
36     .bm_height = {0x00, 0x01, 0x00, 0x00}, /* height 480 */
37     .color_plane = {0x01, 0x00},
38     .pixel_width = {0x18, 0x00}, /* pixel 24 bit true color */
39     .compress = {0x00, 0x00, 0x00, 0x00}, /* not compressed */
40     .bm_bytes = {0x00, 0x10, 0x00, 0x00}, /* frame length 921600 bytes */
41     .meter_width = {0x00, 0x00, 0x00, 0x00},
42     .meter_height = {0x00, 0x00, 0x00, 0x00},
43     .color_index = {0x00, 0x00, 0x00, 0x00},
44     .index_num = {0x00, 0x00, 0x00, 0x00}
45 };
46
47 static const BmpMode BMODE_800x600 = {
48     .bm_header = {0x42, 0x4d},
49     .bm_len = {0x36, 0xf9, 0x15, 0x00}, /* file length 1440000+54 bytes */
50     .reserved = {0x00, 0x00, 0x00, 0x00},
51     .offset = {0x36, 0x00, 0x00, 0x00}, /* 54 bytes */
52     .bm_infolen = {0x28, 0x00, 0x00, 0x00}, /* 40 bytes */
53     .bm_width = {0x20, 0x03, 0x00, 0x00}, /* width 800 */
54     .bm_height = {0x58, 0x02, 0x00, 0x00}, /* height 600 */
55     .color_plane = {0x01, 0x00}
56 };

```

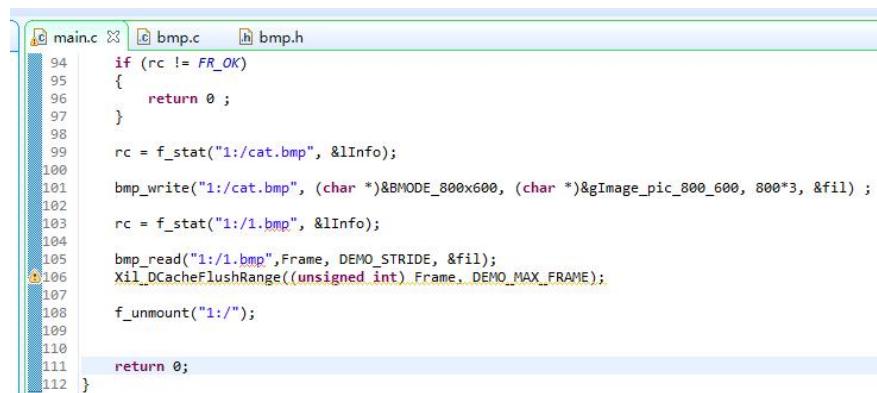
- 6) In combination with the display of the previous cat picture, save

the cat picture in “bmp” format, save it to the “SD” card, and combine it with the “bmp” header and “bmp” data in the “bmp\_write” function to save to the SD card.



```
main.c  bmp.c  bmp.h
39
40
41
42
43
44
45
46
47
48
49
50
51
52 void bmp_write(char * name, char *head_buf, char *data_buf, u32 stride, FIL *fil)
53 {
54     short y,x;
55     short Ximage;
56     short Yimage;
57     u32 iPixelAddr = 0;
58     FRESULT res;
59     unsigned int br;           // File R/W count
60
61     memset(&Write_line_buf, 0, 1920*3);
62
63     res = f_open(fil, name, FA_CREATE_ALWAYS | FA_WRITE);
64     if(res != FR_OK)
65     {
66         printf("error: f_open Failed!\r\n");
67         return;
68     }
69     res = f_write(fil, head_buf, 54, &br);
70     if(res != FR_OK)
71     {
72         f_close(fil);
73         printf("error: f_write Failed!\r\n");
74         return;
75     }
76     Ximage=(unsigned short)head_buf[19]*256+head_buf[18];
77     Yimage=(unsigned short)head_buf[23]*256+head_buf[22];
78     iPixelAddr = (Yimage-1)*stride;
79     for(y = 0; y < Yimage ; y++)
80     {
81         for(x = 0; x < Ximage; x++)
82         {
83             Write_line_buf[x*3 + 0] = data_buf[x*3 + iPixelAddr + 0];
84             Write_line_buf[x*3 + 1] = data_buf[x*3 + iPixelAddr + 1];
85             Write_line_buf[x*3 + 2] = data_buf[x*3 + iPixelAddr + 2];
86         }
87         res = f_write(fil, Write_line_buf, Ximage*3, &br);
88         if(res != FR_OK)
89         {
90             f_close(fil);
91             printf("error: f_write Failed!\r\n");
92             return;
93         }
94         iPixelAddr -= stride;
95     }
96     f_close(fil);
97     printf("BMP write successfully!\r\n");
98 }
```

- 7) In the “main” function, the “bmp\_read” function is called to implement the storage of an image from the SD card to the VDMA display buffer. The file name “1.bmp” of the “BMP” image here needs to be the same as the file name stored in the SD card. Write the cat picture to the SD card with “bmp\_write”.



```

main.c  main.c  bmp.c  bmp.h
94     if (rc != FR_OK)
95     {
96         return 0 ;
97     }
98
99     rc = f_stat("1:/cat.bmp", &lInfo);
100
101    bmp_write("1:/cat.bmp", (char *)&BMODE_800x600, (char *)&gImage_pic_800_600, 800*3, &fil) ;
102
103    rc = f_stat("1:/1.bmp", &lInfo);
104
105    bmp_read("1:/1.bmp",Frame, DEMO_STRIDE, &fil);
106    Xil_DCacheFlushRange((unsigned int) Frame, DEMO_MAX_FRAME);
107
108    f_unmount("1:/");
109
110
111    return 0;
112

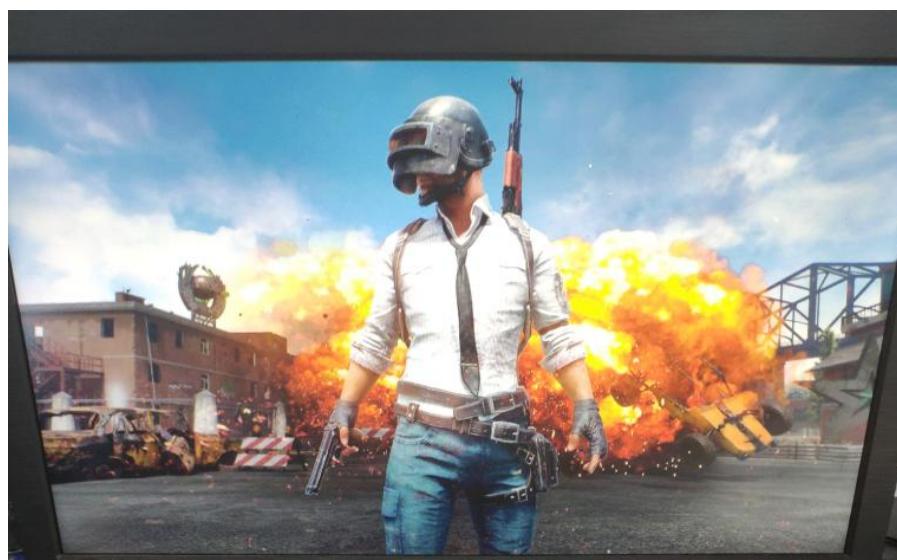
```

### Part 8.3: Onboard Verification

- 1) First, you need to save a pair of 1920\*1080 pixels and 24bit BMP files to the SD card. The file name is 1.bmp (the file is in the project directory). When the development board is powered off, insert the SD card into the SD Card socket.



- 2) The FPGA development board is connected to the DP Monitor, and then powered on. After the download program runs, we can display the image of the “1.bmp” file stored in the SD card on the DP Monitor.



- 3) After that, you can power off the FPGA development board, insert

the SD card into the computer, you can see additional “CAT.BMP”



1.bmp



CAT.BMP

## Part 9: PS Side Use of Ethernet (LWIP)

**The vivado project directory is "ps\_hello/vivado"**

**The vitis project directory is "ps\_net/vitis"**

### Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

The FPGA development board has 2 Gigabit Ethernet and is connected through the RGMII interface. This experiment demonstrates how to use the LWIP template that comes with the Vitis for Gigabit Ethernet TCP communication on the PS side..

Although LWIP is a lightweight protocol stack, if it has never been used before, there will be some difficulties in using it. It is recommended to be familiar with the relevant knowledge of LWIP first.

### Part 9.1: Vitis Program Development

#### Part 9.1.1: LWIP Library Modification

Since the built-in LWIP library can only identify some phy chips, if the phy chip used by the FPGA development board is not within the default support, modify the library file. You can also replace the original library directly with the modified library.

1) Find the library file directory  
"X:\xxx\Vitis\2020.1\data\embeddedsw\ThirdParty\sw\_services"

名称	修改日期	类型	大小
libmetal_v1_3	2020/6/5 18:46	文件夹	
libmetal_v1_4	2020/6/5 19:06	文件夹	
libmetal_v1_5	2020/6/5 18:46	文件夹	
libmetal_v2_0	2020/6/5 18:46	文件夹	
libmetal_v2_1	2020/6/5 18:46	文件夹	
lwip211_v1_0	2020/6/5 18:46	文件夹	
lwip211_v1_1	2020/6/5 18:46	文件夹	
lwip211_v1_2	2020/6/5 18:46	文件夹	
openamp_v1_2	2020/6/5 18:46	文件夹	
openamp_v1_3	2020/6/5 18:46	文件夹	
openamp_v1_4	2020/6/5 18:46	文件夹	
openamp_v1_5	2020/6/5 18:46	文件夹	
openamp_v1_6	2020/6/5 18:46	文件夹	

- 2) Find the files "xaxiemarkif\_physpeed.c" and "xemacpsif\_physpeed.c" in the file directory "lwip211\_v1\_2\src\contrib\ports\xilinx\netif" to be modified.

名称	修改日期	类型	大小
xadapter.c	2020/5/28 7:54	C 文件	12 KB
xaxiemarkif.c	2020/5/28 7:54	C 文件	18 KB
xaxiemarkif_dma.c	2020/5/28 7:54	C 文件	30 KB
xaxiemarkif_fifo.c	2020/5/28 7:54	C 文件	12 KB
xaxiemarkif_fifo.h	2020/5/28 7:54	H 文件	2 KB
xaxiemarkif_hw.c	2020/5/28 7:54	C 文件	5 KB
xaxiemarkif_hw.h	2020/5/28 7:54	H 文件	2 KB
xaxiemarkif_mdma.c	2020/5/28 7:54	C 文件	24 KB
xaxiemarkif_physpeed.c	2020/5/28 7:54	C 文件	25 KB
xemac_ieee_reg.h	2020/5/28 7:54	H 文件	5 KB
xemacliteif.c	2020/5/28 7:54	C 文件	24 KB
xemacpsif.c	2020/5/28 7:54	C 文件	20 KB
xemacpsif_dma.c	2020/5/28 7:54	C 文件	27 KB
xemacpsif_hw.c	2020/5/28 7:54	C 文件	9 KB
xemacpsif_hw.h	2020/5/28 7:54	H 文件	2 KB
xemacpsif_physpeed.c	2020/5/28 7:54	C 文件	37 KB
xpqueue.c	2020/5/28 7:54	C 文件	3 KB

- 3) Modify the "xaxiemarkif\_physpeed.c" file on the PL side and add related macro definitions

```

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

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

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

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

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

#define MICREL_PHY_IDENTIFIER            0x22
#define MICREL_PHY_KSZ9031_MODEL        0x220

#define TI_PHY_REGCR                   0xD
#define TI_PHY_ADDDR                   0xE

```

#### 4) Add “phy” Speed acquisition function

```

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

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

    XAxiEthernet_Physize(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

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

    XAxiEthernet_Physize(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,

```

```
    &control);  
  
control |= ADVERTISE_1000;  
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,  
                      control);  
  
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);  
XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,  
                     control);  
control |= (7 << 12); /* max number of gigabit attempts */  
control |= (1 << 11); /* enable downshift */  
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_COPPER_SPECIFIC_CONTROL_REG,  
                      control);  
XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);  
control |= IEEE_CTRL_AUTONEGOTIATE_ENABLE;  
control |= IEEE_STAT_AUTONEGOTIATE_RESTART;  
  
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);  
  
XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);  
control |= IEEE_CTRL_RESET_MASK;  
XAxiEthernet_PhyWrite(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, control);  
  
while (1) {  
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_CONTROL_REG_OFFSET, &control);  
    if (control & IEEE_CTRL_RESET_MASK)  
        continue;  
    else  
        break;  
}  
xil_printf("Waiting for PHY to complete autonegotiation.\r\n");  
  
XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET, &status);  
while ( !(status & IEEE_STAT_AUTONEGOTIATE_COMPLETE) ) {  
    sleep(1);  
    XAxiEthernet_PhyRead(xaxiemacp, phy_addr, IEEE_STATUS_REG_OFFSET,  
                         &status);  
}  
  
xil_printf("autonegotiation complete \r\n");  
  
XAxiEthernet_PhyRead(xaxiemacp, phy_addr, 0x1f, &partner_capabilities);  
  
if ( (partner_capabilities & 0x40) == 0x40)/* 1000Mbps */  
    return 1000;  
else if ( (partner_capabilities & 0x20) == 0x20)/* 100Mbps */  
    return 100;  
else if ( (partner_capabilities & 0x10) == 0x10)/* 10Mbps */  
    return 10;  
else  
    return 0;  
}
```

## 5) Modify the function "get\_IEEE\_phy\_speed" to add support for

## KSZ9031

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

#ifndef XPAR_AXIETHERNET_0_BASEADDR
    u32 phy_addr = detect_phy(xaxiemacp);

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

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

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

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

        if (phy_model == TI_PHY_DP83867_MODEL) {
            return get_phy_speed_TI_DP83867(xaxiemacp, phy_addr);
        }
    }
    else if(phy_identifier == MICREL_PHY_IDENTIFIER)
    {
        xil_printf("Phy %d is KSZ9031\n\r", phy_addr);
        get_phy_speed_ksz9031(xaxiemacp, phy_addr);
    }
    else {
        LWIP_DEBUGF(NETIF_DEBUG, ("XAxiEthernet get_IEEE_phy_speed: Detected PHY with unknown
identifier/model.\n\r"));
    }
#endif
#endif
#endif
}
```

- 6) Modify the "xemacpsif\_physpeed.c" file on the PS side to add

## macro definitions

```

#define PHY_DETECT_REG          1
#define PHY_IDENTIFIER_1_REG    2
#define PHY_IDENTIFIER_2_REG    3
#define PHY_DETECT_MASK         0x1808
#define PHY_MARVELL_IDENTIFIER 0x0141
#define PHY_TI_IDENTIFIER       0x2000
#define PHY_REALTEK_IDENTIFIER  0x001c
#define PHY_XILINX_PCS_PMA_ID1 0x0174
#define PHY_XILINX_PCS_PMA_ID2 0x0C00

#define XEMACPS_GMII2RGMII_SPEED1000_FD   0x140
#define XEMACPS_GMII2RGMII_SPEED100_FD     0x2100
#define XEMACPS_GMII2RGMII_SPEED10_FD       0x100
#define XEMACPS_GMII2RGMII_REG_NUM         0x10

#define PHY_REGCR      0x0D
#define PHY_ADDAR      0x0E
#define PHY_RGMIIDCTL 0x86
#define PHY_RGMIICTL   0x32
#define PHY_STS        0x11
#define PHY_TI_CR      0x10
#define PHY_TI_CFG4    0x31

#define MICREL_PHY_IDENTIFIER          0x22
#define MICREL_PHY_KSZ9031_MODEL     0x220

#define PHY_REGCR_ADDR 0x001F
#define PHY_REGCR_DATA 0x401F
#define PHY_TI_CRVAL  0x5048
#define PHY_TI_CFG4RESVDBIT7 0x80

```

## 7) Add “phy” Speed acquisition function

```

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

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

    XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 2);
    XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, &control);
    control |= IEEE_RGMII_TXRX_CLOCK_DELAYED_MASK;
    XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_CONTROL_REG_MAC, control);

    XEmacPs_PhWrite(xemacpsp, phy_addr, IEEE_PAGE_ADDRESS_REGISTER, 0);

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

    XEmacPs_PhRead(xemacpsp, phy_addr, IEEE_1000_ADVERTISE_REG_OFFSET,
                   &control);

```

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

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

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

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

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

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

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

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

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

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

```
    return 10;
else
    return 0;
return XST_SUCCESS;
}
```

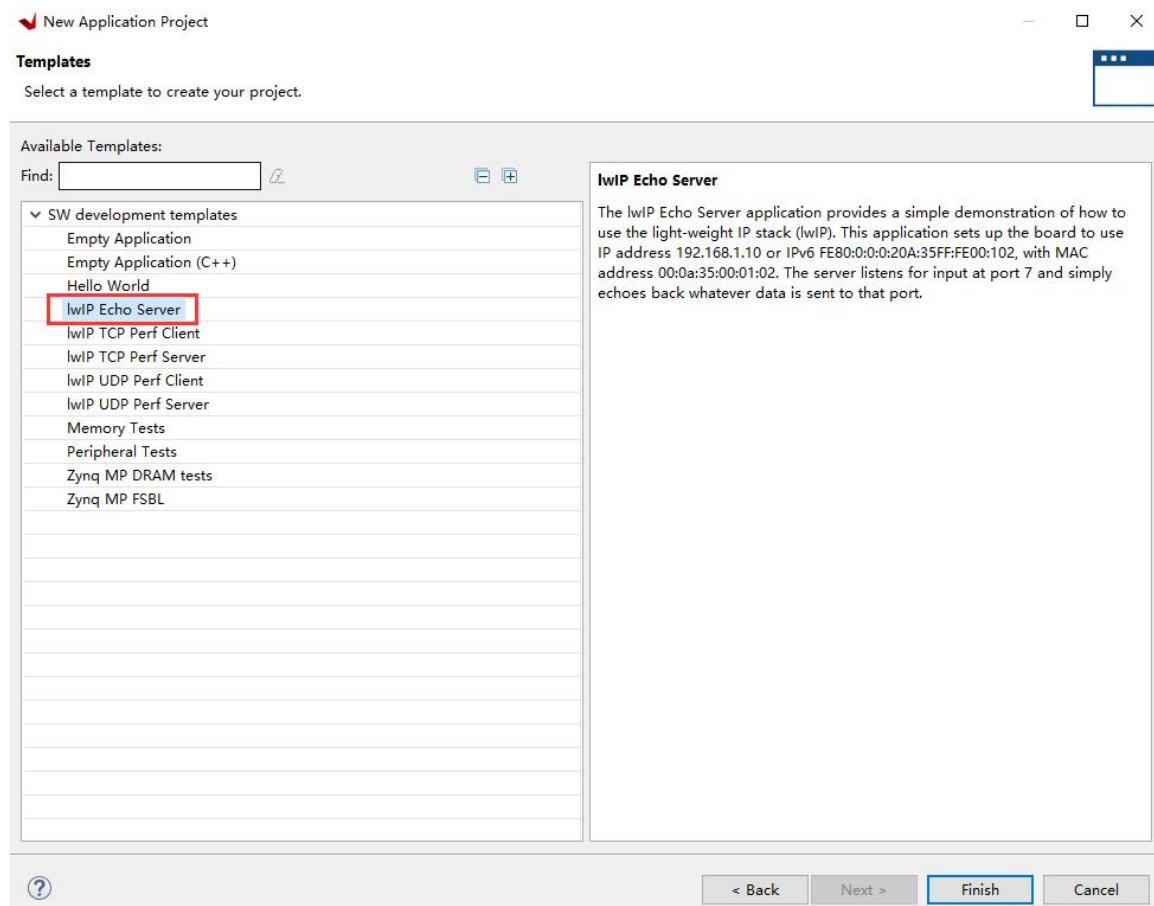
- 8) Modify the function "get\_IEEE\_phy\_speed" to add support for KSZ9031

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

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

    return RetStatus;
}
```

## Part 9.1.2: Create an APP Based on the LWIP Template

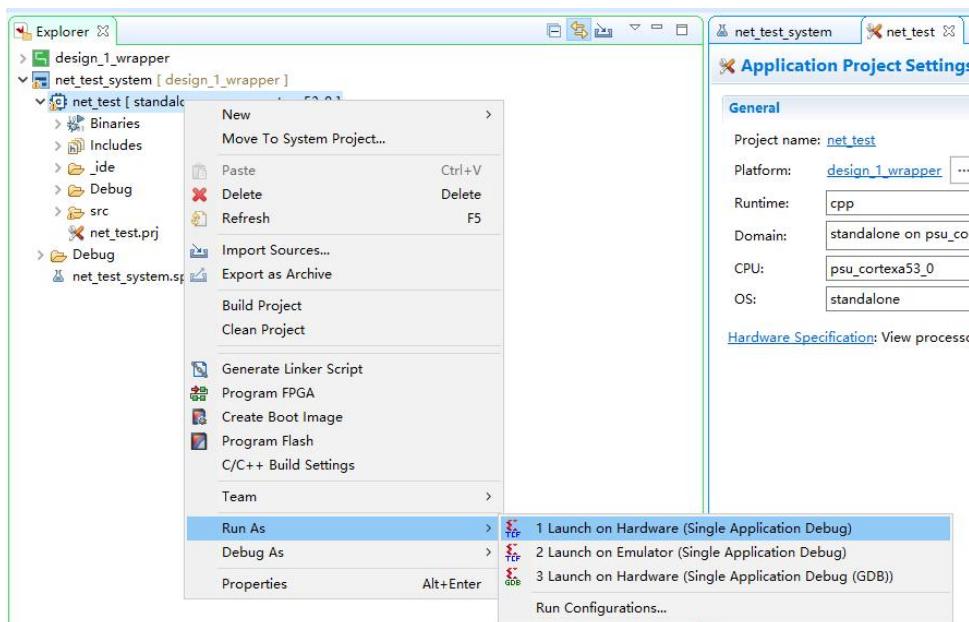


## Part 9.2: Download Debugging

The test environment requires a router that supports “dhcp”, The FPGA development board connects to the router to automatically obtain an IP address. The experimental host and the FPGA development board is in the same network and can communicate with each other.

### Part 9.2.1: Ethernet Testing

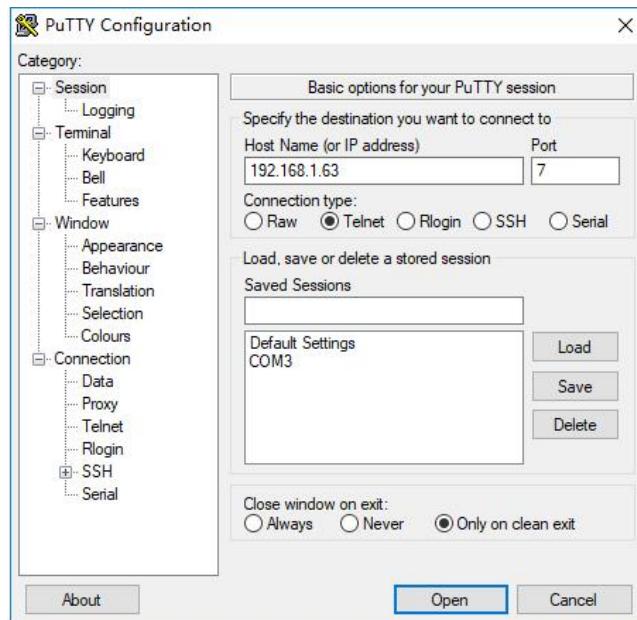
- 1) Connect the serial port to open the serial debugging terminal, connect the PS Ethernet cable to the router. Run the Vitis to Download the Programs



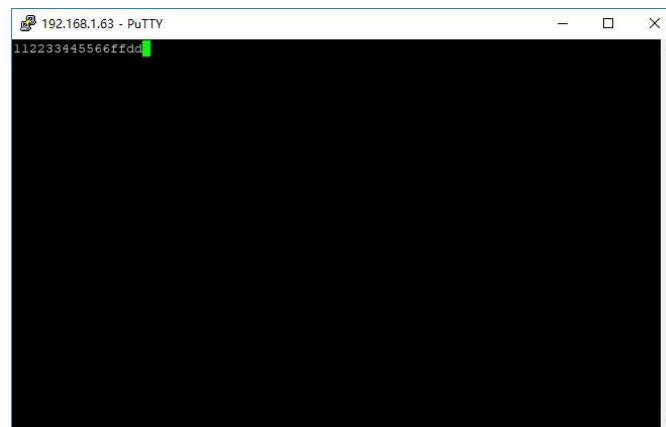
- 2) You can see that the serial port prints some information, you can see that the address is automatically obtained as "192.168.1.68", the connection speed is 1000Mbps, and the tcp port is 7

```
----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 1: 1000
Board IP: 192.168.1.68
Netmask : 255.255.255.0
Gateway : 192.168.1.1
TCP echo server started @ port 7
```

### 3) Connect using “telnet”



### 4) The FPGA development board returns the same character when entering a character



## Part 9.3: Experimental summary

Through the experiments we have a deeper understanding of the development of the Vitis program. This experiment simply explains how to create an LWIP application. LWIP can complete protocols such as UDP and TCP. In subsequent tutorials we will provide specific applications based on Ethernet. For example, the ADC collected data is transmitted via Ethernet, and the camera data is transmitted to the host computer for display via Ethernet.

# Part 10: PS Side Remote Update QSPI Flash by Ethernet

The vivado project directory is "ps\_hello/vivado"

The vitis project directory is "ps\_remote/vitis"

In actual work, you will encounter product upgrade problems. If you follow the programming method, you may need to open the product shell, which is undoubtedly. This chapter introduces a method for remotely updating FLASH programs through the network, including UDP and TCP methods.

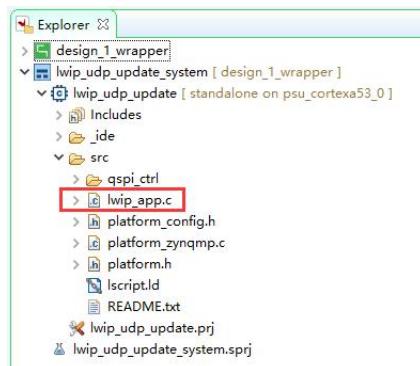
## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 10.1: Vitis Program Development

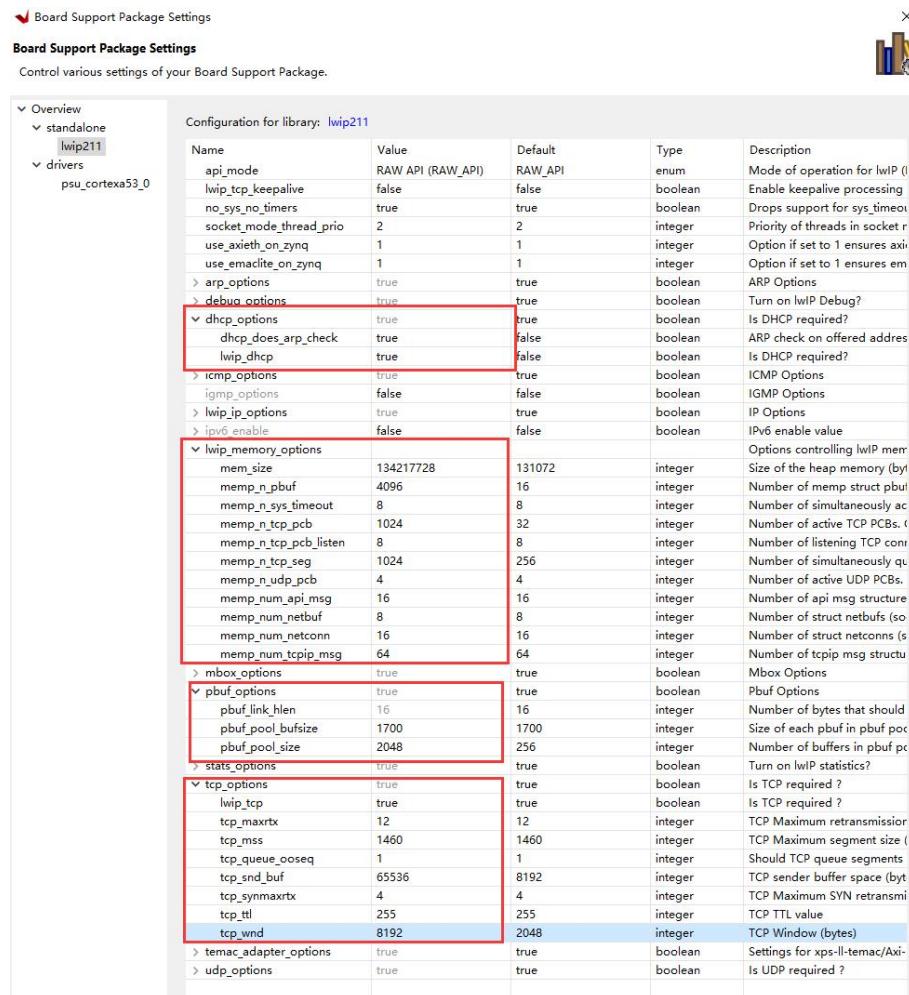
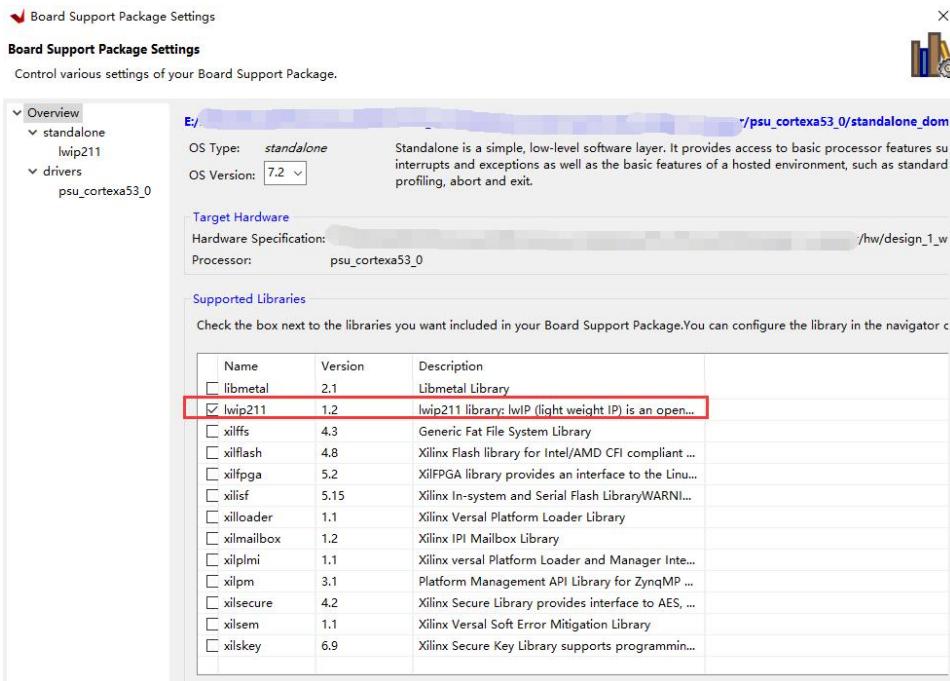
#### Part 10.1.1: UDP Transmission Mode

- 1) The LWIP part mainly handles the reception of “BIN” files, and the program is “lwip\_app.c”



- 2) After creating the project, you need to enable the “lwip” library and set it to enable the “DHCP” function. Set the “memory” space as large as possible to increase the cache space and improve

efficiency.

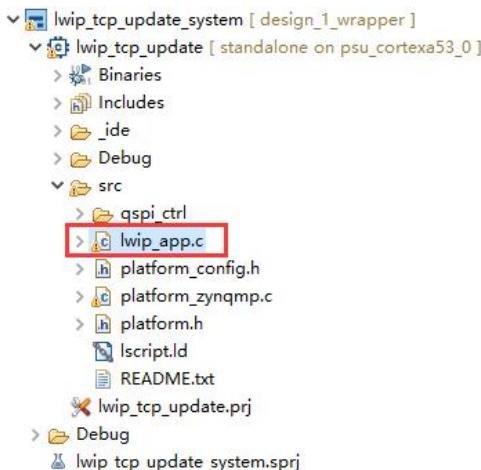


- 3) The “udp\_receive” function is the set receive callback function. The main function is to receive the data, and buffer the received data into the “FlashRxBuffer” space, leaving it to be used for updating the “Flash”. After sending the data, send the "update" command to start updating the “flash”. Judge this command.
- 4) In the “while” loop statement, determine the value of the “StartUpdate” variable and update the “Flash”.

```
while (1)
{
    xemacif_input(echo_netif);
    if (StartUpdate)
    {
        Status = update_qspi(&QspiInstance, QSPI_DEVICE_ID, ReceivedCount, FlashRxBuffer) ;
        if (Status != XST_SUCCESS)
            xil_printf("Write Flash Error!\r\n");
        else
        {
            StartUpdate = 0 ;
            ReceivedCount = 0;
        }
    }
}
```

### Part 10.1.2: TCP Transmission Method

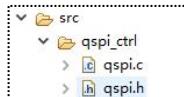
- 1) The “LWIP” part of “TCP” is also the “lwip\_app.c” file. The control part refers to the “lwip echo” server routine to establish a “TCP Server”.



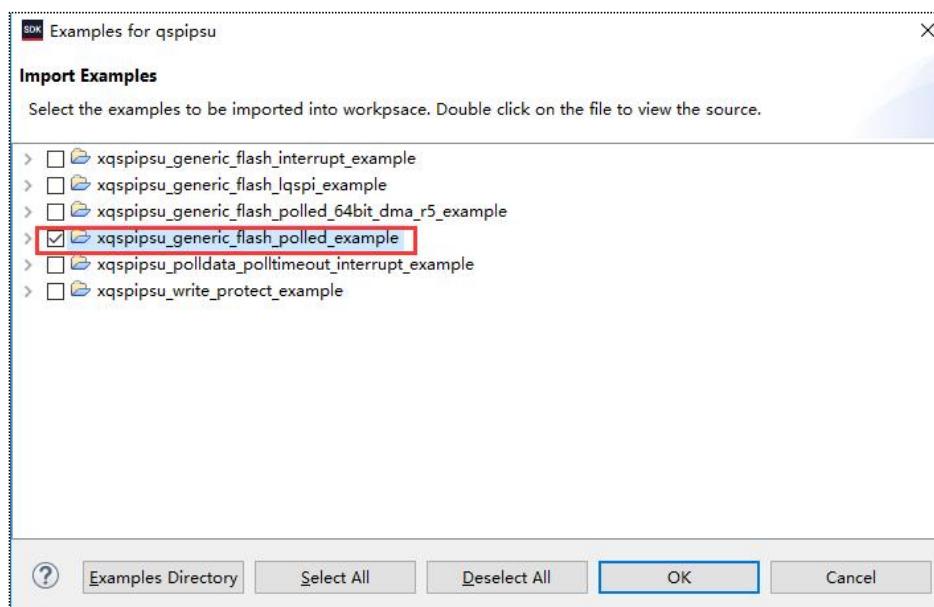
- 2) Similar to UDP, in the recv\_callback receive callback function, the received BIN file is cached. The update command is also updated, and other parts are similar to UDP.

### Part 10.1.3: QSPI Flash Read and Write Control

UDP and TCP use the same QSPI read and write files “qspi.c” and “qspi.h”



- 1) The qspi.c file is modified according to “xqspipsu\_flash\_polled\_example”



- 2) There are mainly the following functions, write enable and disable, flash erase, flash write, flash read, read flash ID and so on

```
***** Function Prototypes *****/
int update_qspi(XQspiPsu *QspiPsuInstancePtr, u16 QspiPsuDeviceId, unsigned int TotalLen, char *FlashDataToSend) ;
int FlashReadID(XQspiPsu *QspiPsuPtr);
int FlashErase(XQspiPsu *QspiPsuPtr, u32 Address, u32 ByteCount, u8 *WriteBfrPtr);
int FlashWrite(XQspiPsu *QspiPsuPtr, u32 Address, u32 ByteCount, u8 Command,
              u8 *WriteBfrPtr);
int FlashRead(XQspiPsu *QspiPsuPtr, u32 Address, u32 ByteCount, u8 Command,
              u8 *WriteBfrPtr, u8 *ReadBfrPtr);
u32 GetRealAddr(XQspiPsu *QspiPsuPtr, u32 Address);
int BulkErase(XQspiPsu *QspiPsuPtr, u8 *WriteBfrPtr);
int DieErase(XQspiPsu *QspiPsuPtr, u8 *WriteBfrPtr);
int FlashEnterExit4BAddMode(XQspiPsu *QspiPsuPtr,unsigned int Enable);
int FlashEnableQuadMode(XQspiPsu *QspiPsuPtr);

void print_percent(int percent) ;
```

- 3) The main function is “update\_qspi”, where “TotalLen” is the total number of bytes to be updated, and “FlashDataToSend” is the cache area for storing updated data. The flow is also relatively simple. First, it is erased. Here, there is no choice to erase the

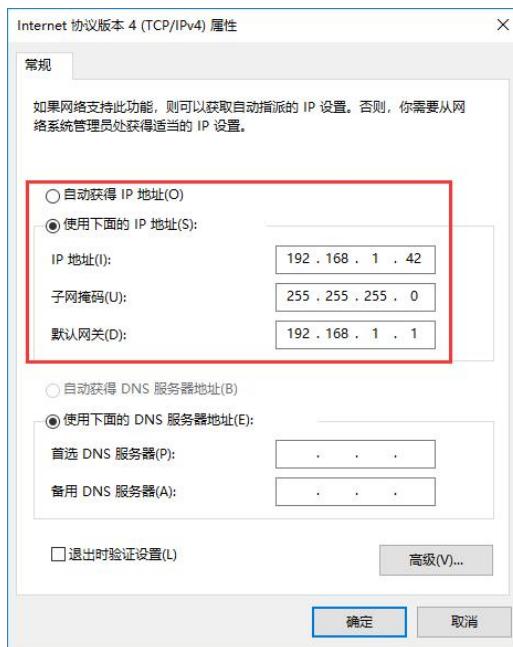
entire Flash. Instead, the Sector erase is done according to the TotalLen size, so the erased space will be slightly larger than the TotalLen; then the Flash is written, the FlashWrite function is used for writing; the final is the check, the data is read from the Flash, and written. The data is compared.

```
int update_qspi(XQspiPsu *QspiPsuInstancePtr, u16 QspiPsuDeviceId, unsigned int TotoalLen, char *FlashDataToSend)
```

## Part 10.2: Onboard Verification

We can choose the BOOT.bin file of other routines. We set up the experiment under the ideal state of the network environment. When doing this experiment, do not open other upper computer software related to Ethernet transmission. Because the port number is the same, it may cause conflicts.

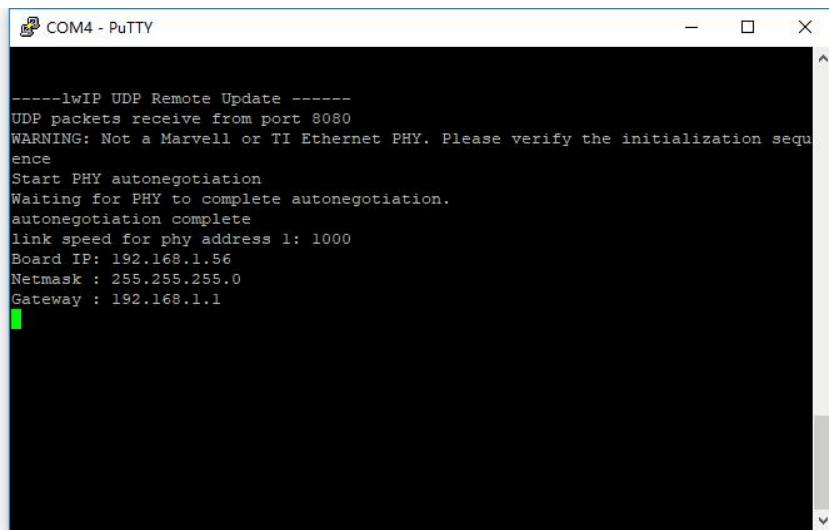
- 1) First connect the development board as follows, connect the network cable to port.
- 2) If there is a DHCP server, the IP will be automatically assigned to the development board; if there is no DHCP server, the default FPGA development board IP address is 192.168.1.10, and the IP address of the PC needs to be set to the same network segment, as shown in the following figure. Also make sure that there is no IP address of 192.168.1.10 in the network, otherwise IP conflict will occur and the image will not be displayed. You can enter “ping 192.168.1.10” in the CMD to check whether the ping can be pinged before the board is powered on. If the ping is successful, the IP address in the network cannot be verified.



Open the putty software after no problem

#### Part 10.2.1: UDP Mode

- 1) Download program, You can see the information in putty



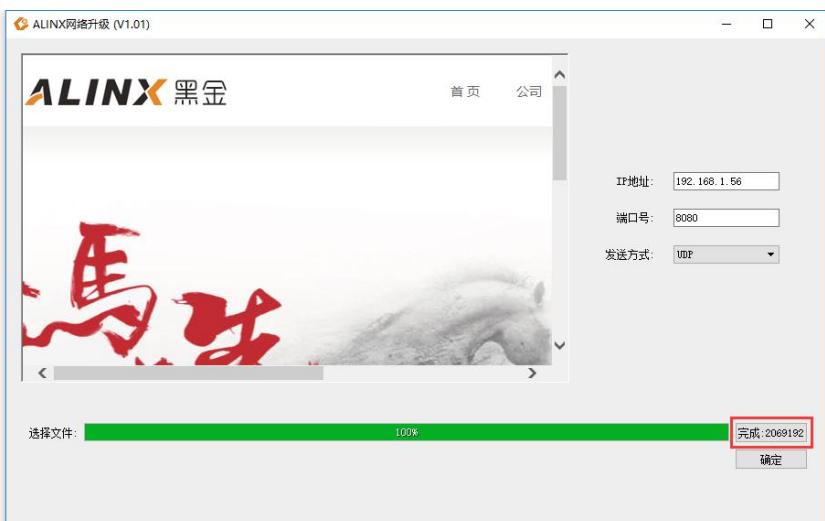
- 2) Open the board network upgrade software in the project directory.



- 3) Fill in the IP address and port number of the board, select the UDP sending method, select the BOOT.bin file, and click Transmit.



4) After Transmitting, the number of bytes sent will be displayed.



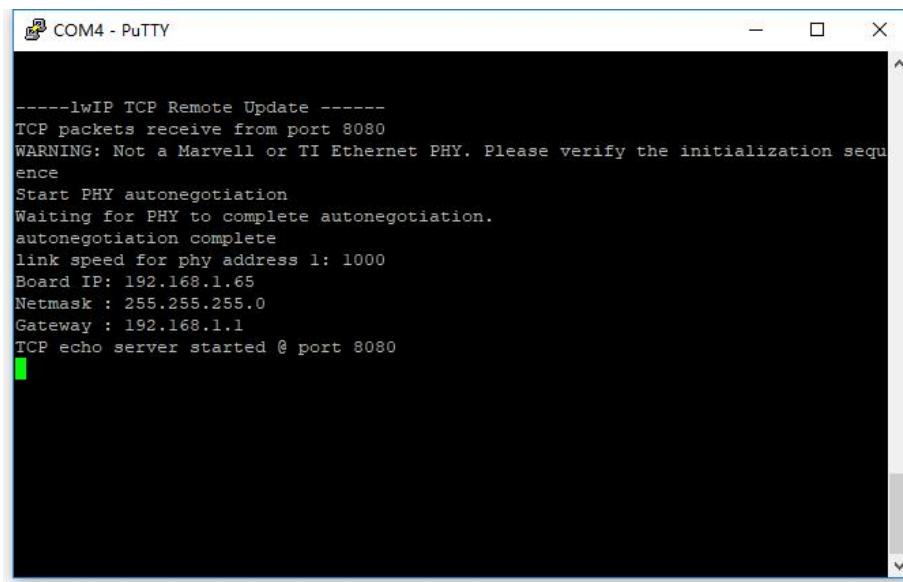
5) In the putty window you can see the number of bytes received by the board, as well as the erase, burn, and verify processes

```
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 1: 1000
Board IP: 192.168.1.56
Netmask : 255.255.255.0
Gateway : 192.168.1.1
Received Size is 2069192 Bytes
Initialization done, programming the memory
FlashID=0xEF 0x40 0x19
Performing Erase Operation...
Erase Size is 2097152 Bytes
0%..10%..20%..30%..40%..50%..60%..70%..80%..90%..100%
INFO:Elapsed time = 4.73 sec
Erase Operation Successful.
Performing Program Operation...
0%..10%..20%..30%..40%..50%..60%..70%..80%..90%..100%
INFO:Elapsed time = 3.88 sec
Program Operation Successful.
Performing Verify Operation...
0%..10%..20%..30%..40%..50%..60%..70%..80%..90%..100%
INFO:Elapsed time = 0.79 sec
Verify Operation Successful.
```

- 6) The power-off pass-through code switch selects the QSPI startup mode, and when the power is turned on, the program can be seen to run.

### Part 10.2.2: TCP Mode

- 1) Download the program, You can see putty information



```
-----lwIP TCP Remote Update -----  
TCP packets receive from port 8080  
WARNING: Not a Marvell or TI Ethernet PHY. Please verify the initialization sequence  
Start PHY autonegotiation  
Waiting for PHY to complete autonegotiation.  
autonegotiation complete  
link speed for phy address 1: 1000  
Board IP: 192.168.1.65  
Netmask : 255.255.255.0  
Gateway : 192.168.1.1  
TCP echo server started @ port 8080
```

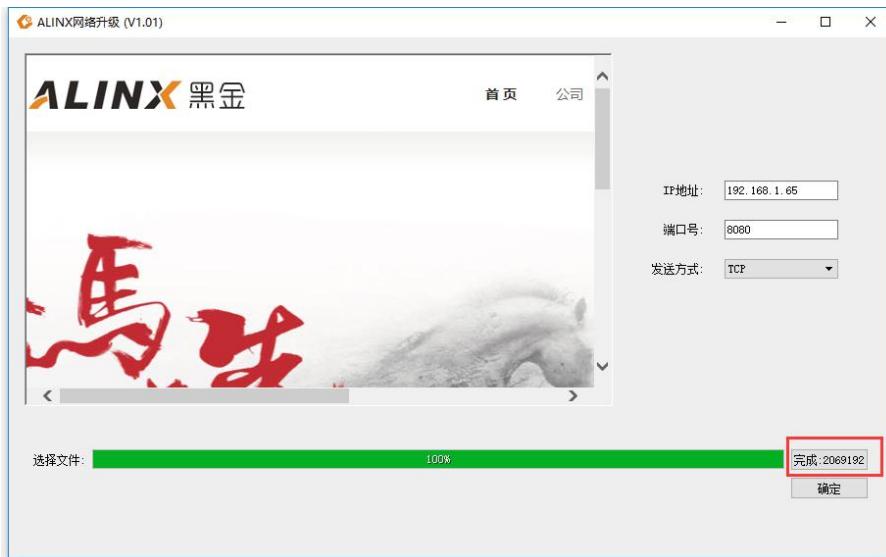
- 2) Open the board network upgrade software in the project directory.



- 3) Fill in the IP address and port number, select the TCP transmitting method, select the BOOT.bin file, and click Transmit.



- 4) As with UDP, you can also see the number of bytes sent.



- 5) In the putty window you can see the number of bytes received by the board, as well as the erase, burn, and verify processes.

```
COM4 - PuTTY
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 1: 1000
Board IP: 192.168.1.65
Netmask : 255.255.255.0
Gateway : 192.168.1.1
TCP echo server started @ port 8080
Received Size is 2069192 Bytes
Initialization done, programming the memory
FlashID=0xEF Ox40 0x19
Performing Erase Operation...
Erase Size is 2097152 Bytes
0%..10%..20%..30%..40%..50%..60%..70%..80%..90%..100%
INFO:Elapsed time = 4.73 sec
Erase Operation Successful.
Performing Program Operation...
0%..10%..20%..30%..40%..50%..60%..70%..80%..90%..100%
INFO:Elapsed time = 3.88 sec
Program Operation Successful.
Performing Verify Operation...
0%..10%..20%..30%..40%..50%..60%..70%..80%..90%..100%
INFO:Elapsed time = 0.79 sec
Verify Operation Successful.
```

- 6) Power off through the DIP switch to select the QSPI startup mode, turn on the power to start, you can see the program running.

## Part 11: Use of System Monitor

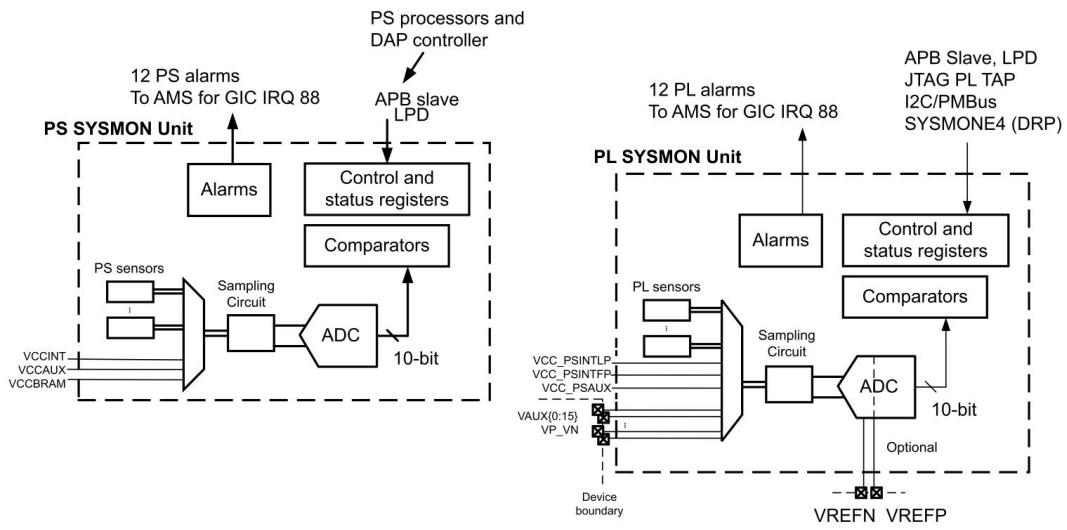
The vivado project directory is "ps\_hello/vivado"

The vitis project directory is "ps\_sysmon/vitis"

### FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

This chapter introduces the use of system monitors, which are used to monitor the voltage and temperature values of the chip, and can also be used as external signal acquisition through the ADC pin on the PL end. The PL end can do 17-channel ADC acquisition, but the FPGA development board does not connect devices to these pins, so this chapter will not explain it. As shown in the figure, the voltage sensor can monitor VCCINT, VCCAUX, VCCBRAM, etc. of the chip. VP\_0 and VN\_0 of PL\_SYSMON are a pair of dedicated ADC analog input ports. VAUXP[\*] and VAUXN[\*] are also ADC input ports, but when not used as ADC input ports, they can be used as ordinary IO. This experiment mainly measures the value of temperature and voltage.



X19410-103117

Figure 9-1: PS SYSMON and PL SYSMON Block Diagram

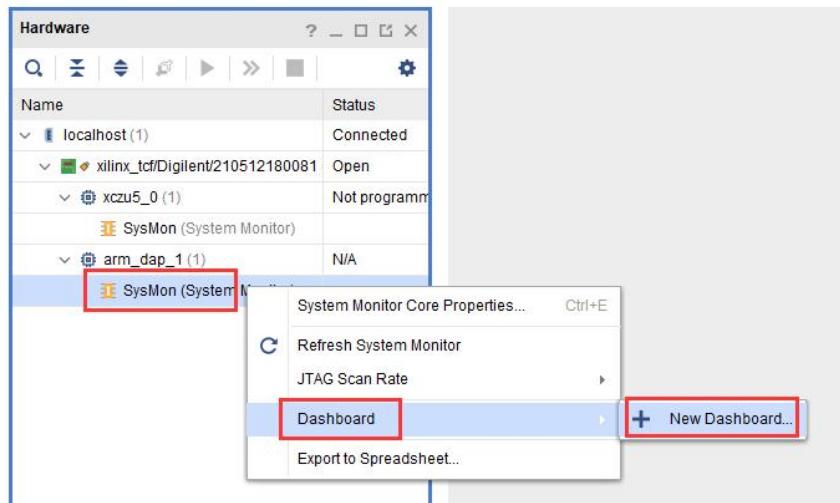
The Vivado project is also based on the "ps\_hello" project.

### Part 11.1: Hardware Read System Monitor

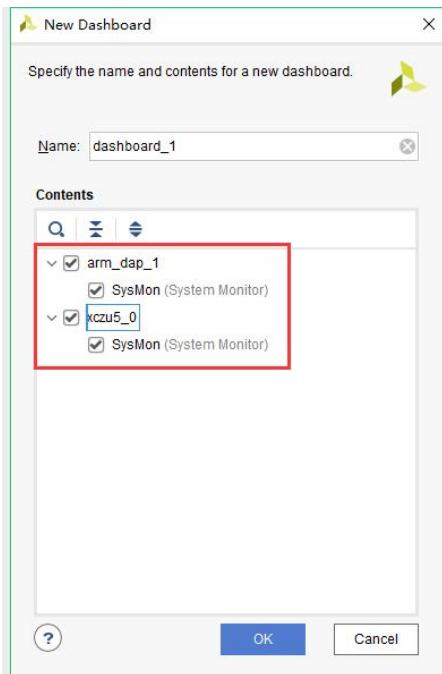
- 1) Open the project, connect the FPGA development board power supply, JTAG downloader, and adjust the development board to **JTAG mode**, power on the development board, click **Open Hardware Manager**, and then click **Auto Connect** to find the hardware.



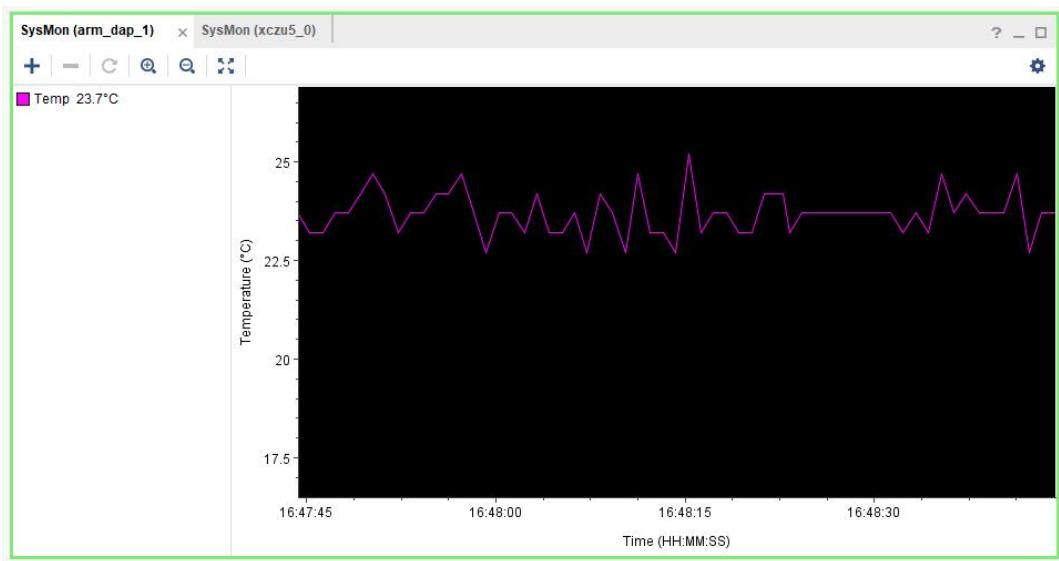
- 2) Right-click and select SysMon, create a new Dashboard



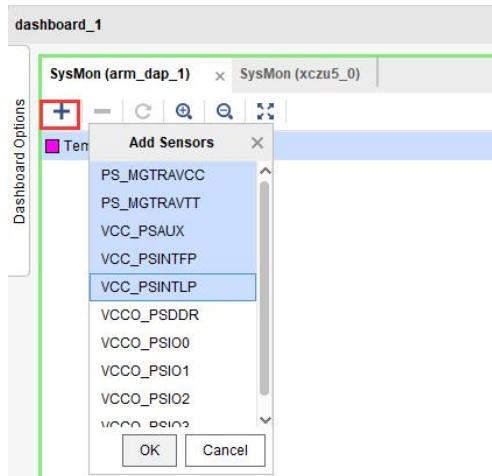
3) Select both PS and PL, click OK



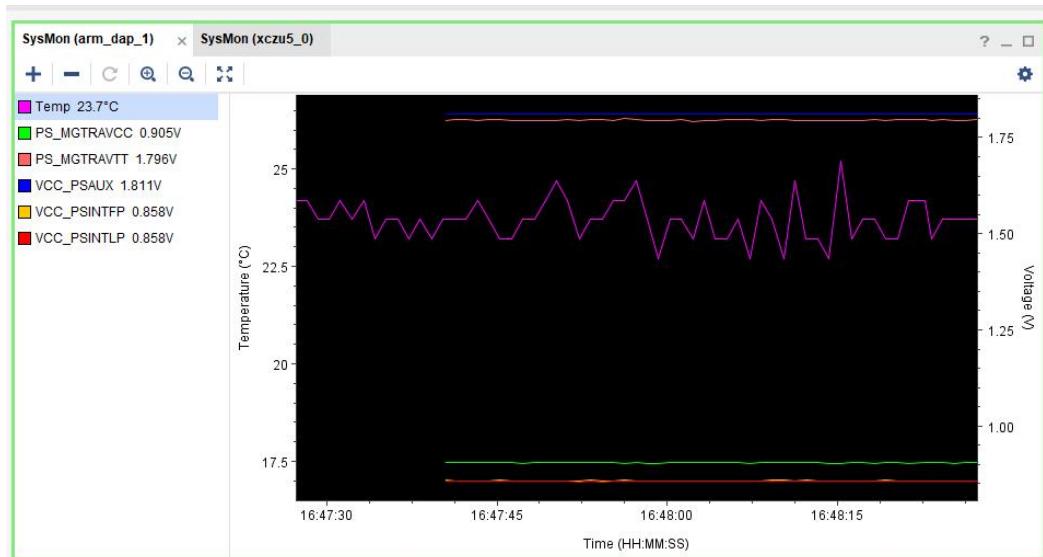
4) There will be temperature information by default



5) Click + to add the voltage value to the window



6) The display is as follows



The advantage of this method is graphical display, which is more

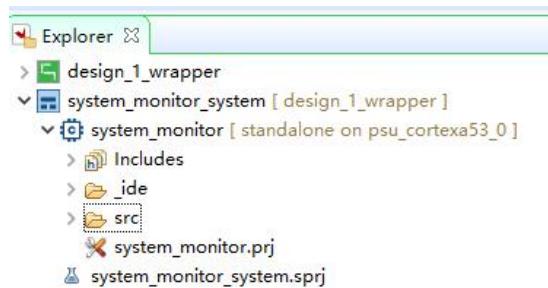
intuitive, but the disadvantage is that the data value cannot be obtained. The following describes how PS reads XADC information.

## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 11.2: PS read System Monitor Information

- 1) Open the Vitis software and create a new Vitis project



- 2) You can see the system monitor in the BSP, import Example to learn

Name	Driver	Documentation	Examples
psu_afi_6	generic	-	-
psu_ams	sysmonpsu	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>
psu_apm_0	axipmon	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>
psu_apm_1	axipmon	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>
psu_apm_2	axipmon	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>
psu_apm_5	axipmon	<a href="#">Documentation Link</a>	<a href="#">Import Examples</a>

- 3) This experimental result is to read the temperature and voltage data, and print it out through the serial port every 1S. Read the original value through the XSysMonPsu\_GetAdcData function, and use the XSysMonPsu\_RawToTemperature\_OnChip macro to convert the ADC value to a temperature value. Use XSysMonPsu\_RawToVoltage to convert to a voltage value.

```
while(1)
{
    /*
     * Read the on-chip Temperature Data (Current/Maximum/Minimum)
     * from the ADC data registers.
     */
    TempRawData = XSysMonPsu_GetAdcData(SysMonInstPtr, XSM_CH_TEMP, XSYSMON_PS);
    TempData = XSysMonPsu_RawToTemperature_OnChip(TempRawData);
    printf("\r\nThe Current Temperature is %d.%03d Centigrades.\r\n",
          (int)(TempData), SysMonPsuFractionToInt(TempData));

    /*
     * Read the VccInt Voltage Data (Current/Maximum/Minimum) from the
     * ADC data registers.
     */
    VccIntRawData = XSysMonPsu_GetAdcData(SysMonInstPtr, XSM_CH_SUPPLY1, XSYSMON_PS);
    VccIntData = XSysMonPsu_RawToVoltage(VccIntRawData);
    printf("\r\nThe Current VCCINT is %d.%03d Volts. \r\n",
          (int)(VccIntData), SysMonPsuFractionToInt(VccIntData));

    /*
     * Read the VccAux Voltage Data (Current/Maximum/Minimum) from the
     * ADC data registers.
     */
    VccAuxRawData = XSysMonPsu_GetAdcData(SysMonInstPtr, XSM_CH_SUPPLY3, XSYSMON_PS);
    VccAuxData = XSysMonPsu_RawToVoltage(VccAuxRawData);
    printf("\r\nThe Current VCCAUX is %d.%03d Volts. \r\n",
          (int)(VccAuxData), SysMonPsuFractionToInt(VccAuxData));

    sleep(1) ;
}
```

- 4) After downloading, you can see the print information in the serial port tool as follows, read the value of temperature, VCCINT, VCCAUX

```
The Current Temperature is 28.811 Centigrades.  
The Current VCCINT is 0.849 Volts.  
The Current VCCAUX is 1.814 Volts.  
The Current Temperature is 29.028 Centigrades.  
The Current VCCINT is 0.849 Volts.  
The Current VCCAUX is 1.812 Volts.
```

## PS and PL Interconnection Parts

In the PS-side peripherals article, the application of PS-side peripherals is introduced. On this basis, we learn the interconnection between PS and PL, which is also the focus of ZYNQ applications, such as image display, camera image acquisition, ADC data acquisition and display , Data acquisition and Ethernet transmission and other applications.

## Part 12: PS Side Use of EMIO

The experimental Vivado project directory is "ps\_emio/vivado"

The experimental vitis project directory is "ps\_emio/vitis".

The experiment of lighting the LED light on the PS side was introduced earlier, but if you want to light the LED light of the PL with PS, how to do it? One is that you can control the PL-side LED lights through EMIO, and the other is through the IP of AXI GPIO. This chapter describes how to use the EMIO to control the on and off of the LED lights on the PL side. It also introduced the use of EMIO to connect the PL key to control the PL LED.

### Part 12.1: Principle Introduction

The structure of the PS-side MIO is introduced as follows. From the figure, we can see that there are 78 MIOs for BANK0 to BANK2. There are 96 EMIOs in BANK3 to BANK5. This chapter is to use EMIO to control the PL LED.

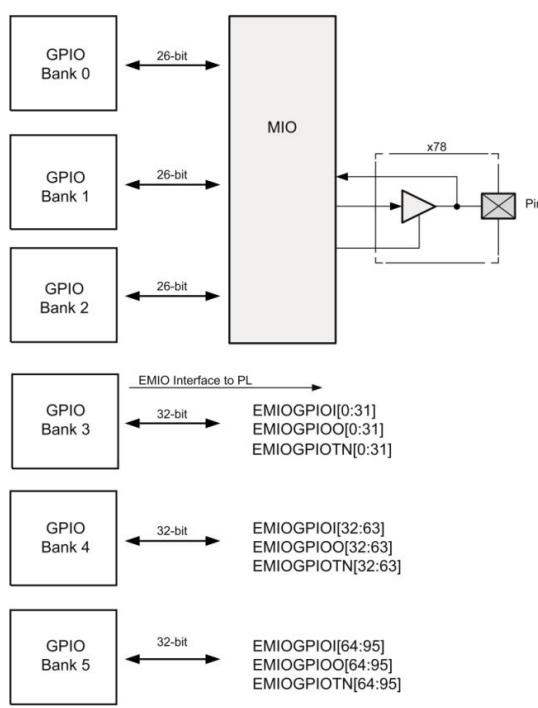


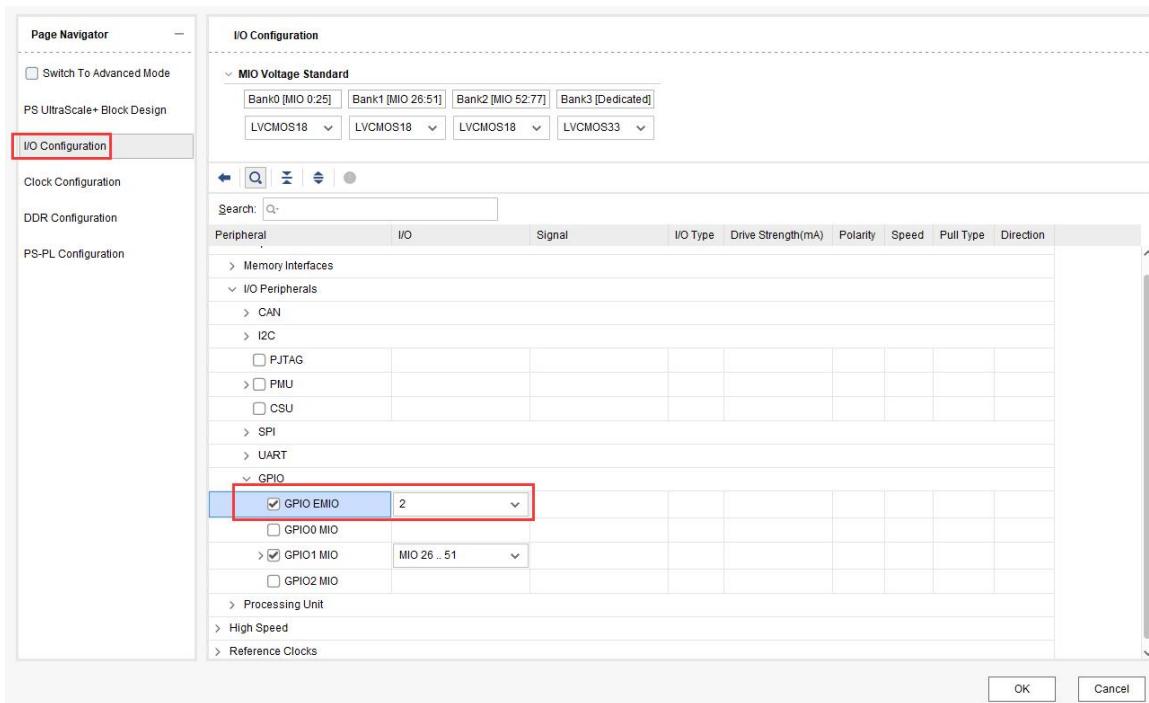
Figure 27-1: GPIO Block Diagram

## FPGA Engineer Job Content

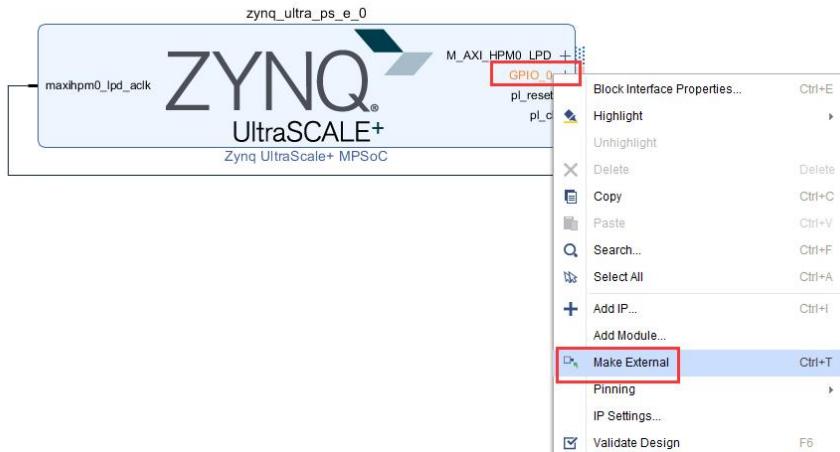
The following is the content that FPGA engineers are responsible for.

### Part 12.2: Create a Vivado Project

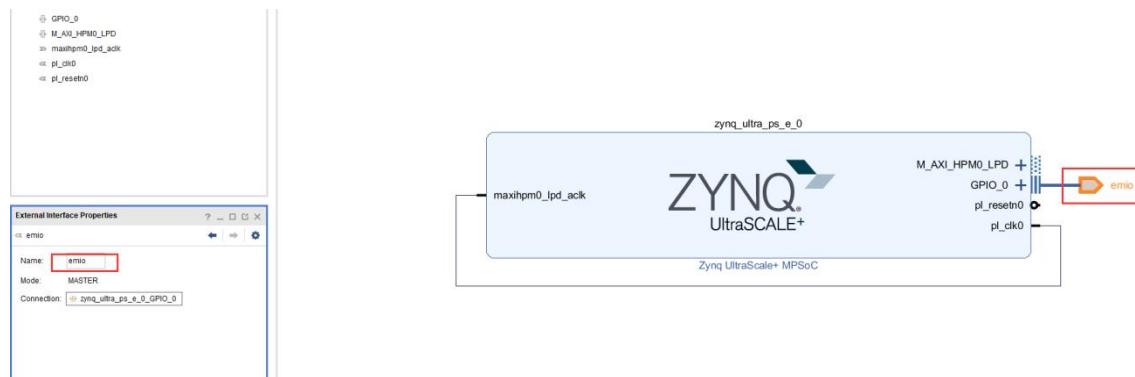
- 1) Based on the “ps\_hello” project, save as a project named “ps\_emio”, open the ZYNQ configuration, and check “GPIO EMIO”. Because there is 1 LED and 1 Key on the PL side, the bit width of EMIO is selected as 2 bits in the MIO configuration. After the configuration is completed, click OK.



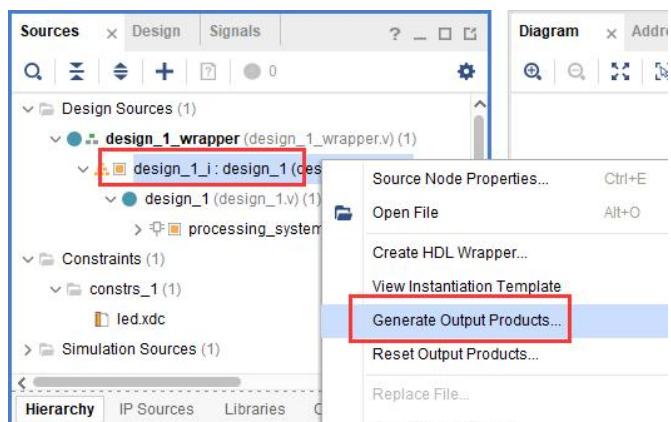
- 2) Click the extra “GPIO\_0” port, right-click and select “Make External” to export the port signal



- 3) Click on the pin and modify the pin name to emio, or you can modify the name according to your needs. Save design



- 4) Right click on “xx.bd” and select “Generate Output Products” to regenerate the output file



- 5) After finishing, the top-level file will be updated with new pins, which need to be pin-bound below

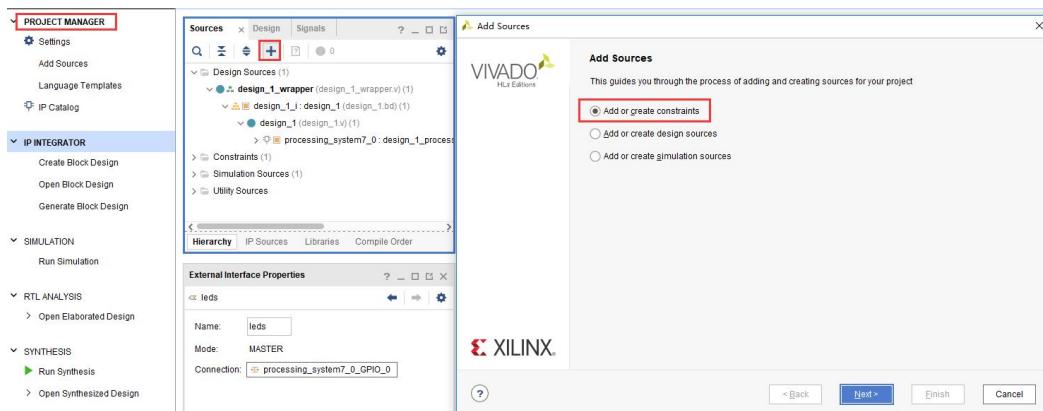
```

1 //Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
2 //
3 //Tool Version: Vivado v.2019.2 (win64) Build 2708876 Wed Nov 6 21:40:23 MST 2019
4 //Date        : Wed Apr 22 09:22:49 2020
5 //Host        : DESKTOP-OPF260C running 64-bit major release (build 9200)
6 //Command    : generate_target design_1_wrapper.bd
7 //Design     : design_1_wrapper
8 //Purpose    : IP block netlist
9 //
10 `timescale 1 ps / 1 ps
11
12 module design_1_wrapper
13   (emio_tri_io);
14   inout [1:0]emio_tri_io;
15
16   wire [0:0]emio_tri_i_0;
17   wire [1:1]emio_tri_i_1;
18   wire [0:0]emio_tri_o_0;
19   wire [1:1]emio_tri_o_1;
20   wire [0:0]emio_tri_o_0;
21   wire [1:1]emio_tri_o_1;
22   wire [0:0]emio_tri_t_0;
23   wire [1:1]emio_tri_t_1;

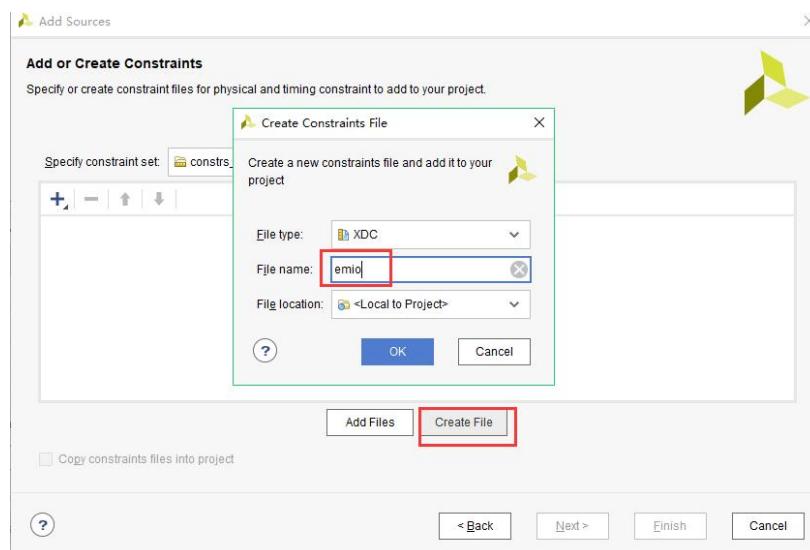
```

## Part 12.3: XDC File Constraint PL Pin

### 6) Create a new XDC file and bind the PL pin



Set the file name to “emio”



- 7) “emio.xdc” add the following content, the port name must be consistent with the top-level file port

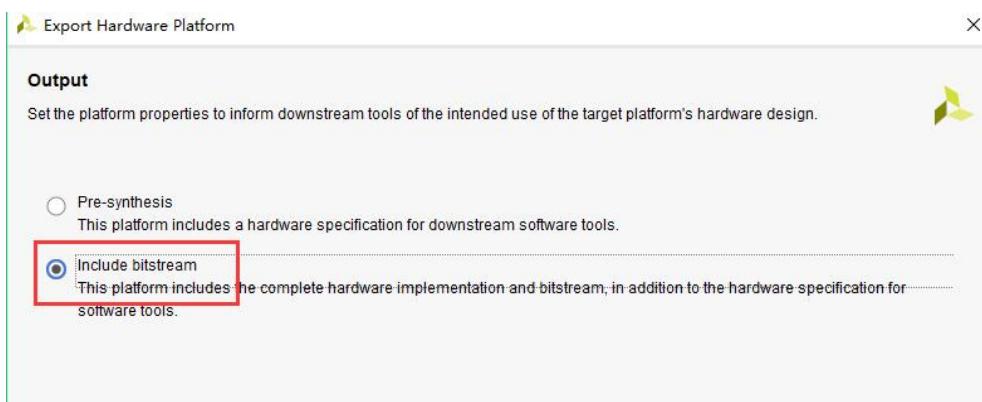
```
#####Compress Bitstream#####
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]

set_property IOSTANDARD LVCMOS33 [get_ports {emio_tri_io[*]}]
#pl led
set_property PACKAGE_PIN AE12 [get_ports {emio_tri_io[0]}]
#pl key
set_property PACKAGE_PIN AF12 [get_ports {emio_tri_io[1]}]
```

- 8) Generate a bit file



- 9) Although the logic on the PL side is not used, the pins on the PL side are used. Therefore, to export the hardware, select "Include bitstream"



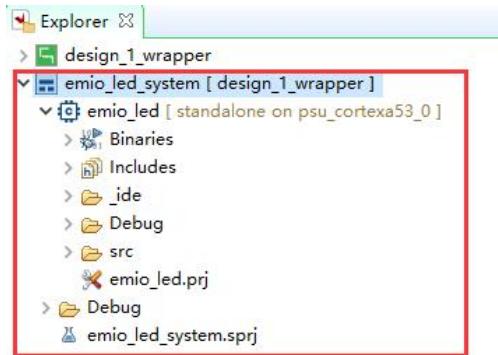
## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

## Part 12.4: Vitis Programming

### Part 12.4.1: EMIO Lights PL LED

- 1) Enter the Vitis software and create a new project named emio\_led



- 2) The code part is similar to the MIO operation on the PS side to turn on the LED. Since the number of MIO is 0 ~ 77, the number of EMIO starts from 78. Just make the following changes

```
int main()
{
    init_platform();

    int Status;
    XGpioPs_Config *ConfigPtr;

    print("Hello World\n\r");
    /* Initialize the GPIO driver. */
    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);

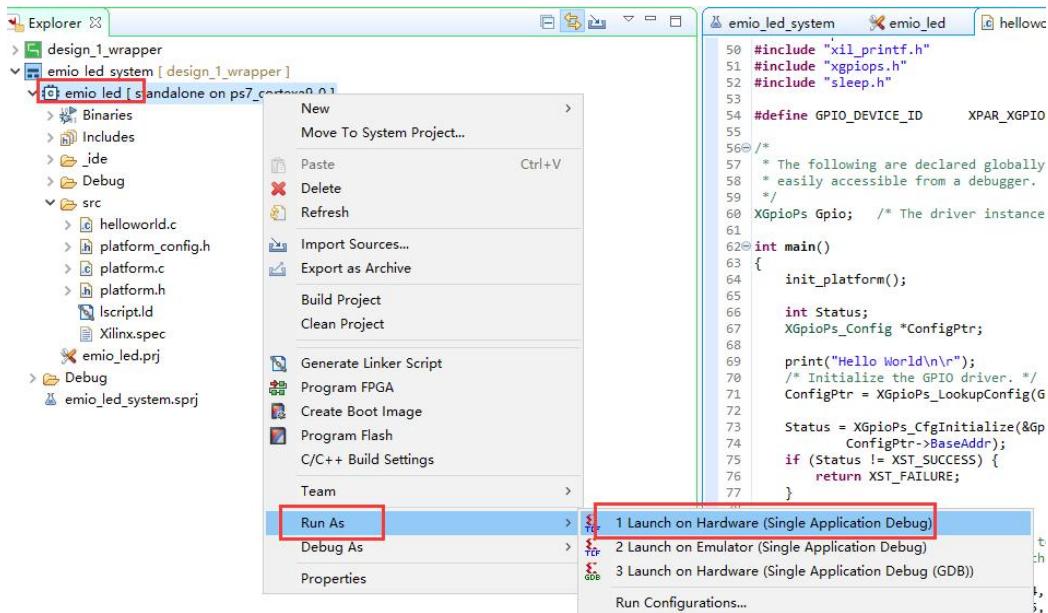
    Status = XGpioPs_CfgInitialize(&Gpio, ConfigPtr,
                                    ConfigPtr->BaseAddr);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Set the direction for the pin to be output and
     * Enable the Output enable for the LED Pin.
     */
    XGpioPs_SetDirectionPin(&Gpio, 78, 1);
    XGpioPs_SetOutputEnablePin(&Gpio, 78, 1);

    while(1){
        /* Set the GPIO output to be low. */
        XGpioPs_WritePin(&Gpio, 78, 0x0);
        sleep(1) ;
        /* Set the GPIO output to be high. */
        XGpioPs_WritePin(&Gpio, 78, 0x1);
        sleep(1) ;
    }

    cleanup_platform();
    return 0;
}
```

- 3) Compile and Download

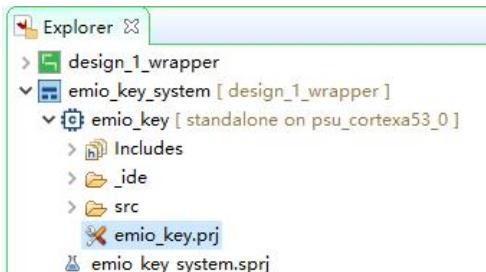


You can see the PL end LED Flashing.

#### Part 12.4.2: EMIO Implements PL Key Interrupt

Controlling the LED light on the PL terminal through the key on the PL terminal

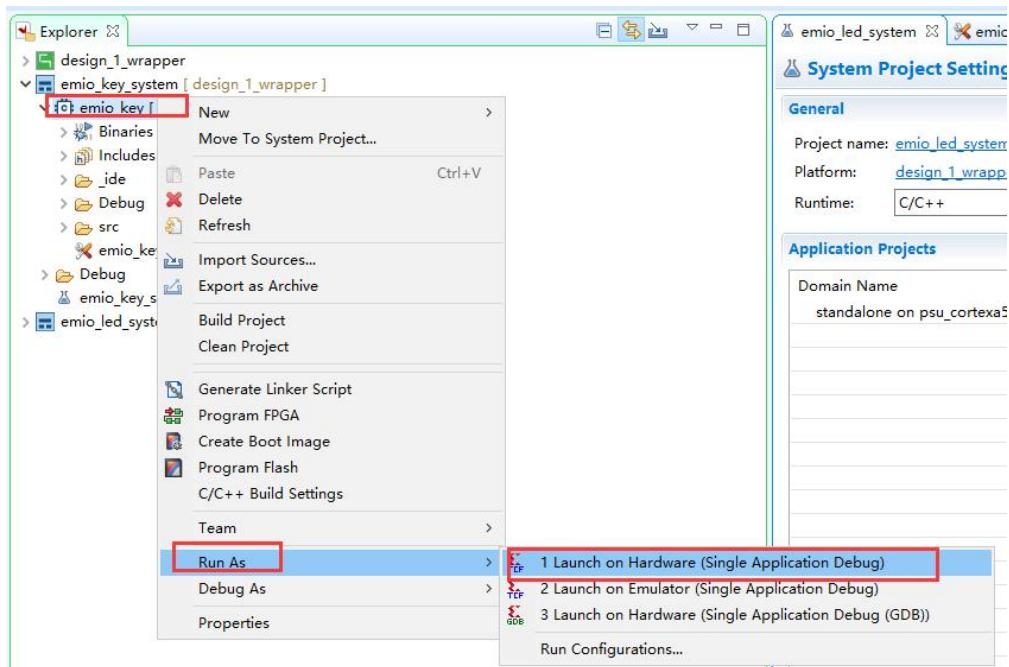
- 1) Create a new project named “emio\_key” with a template of “hello world. Copy the routine program, save and compile



- 2) The MIO key interrupt program used by the PS-side MIO is migrated, and the key number is changed to 58 and the LED light number is 54 to save and regenerate elf.

```
/* GPIO parameter */
#define MIO_ID      XPAR_XGPIOPS_0_DEVICE_ID
#define INTC_DEVICE_ID  XPAR_SCUGIC_SINGLE_DEVICE_ID
#define KEY_INTR_ID  XPAR_XGPIOPS_0_INTR
#define PS_KEY_MIO   79
#define PS_LED_MIO   78
```

- 3) “Run Configurations” Select “Program FPGA” and click “Run”

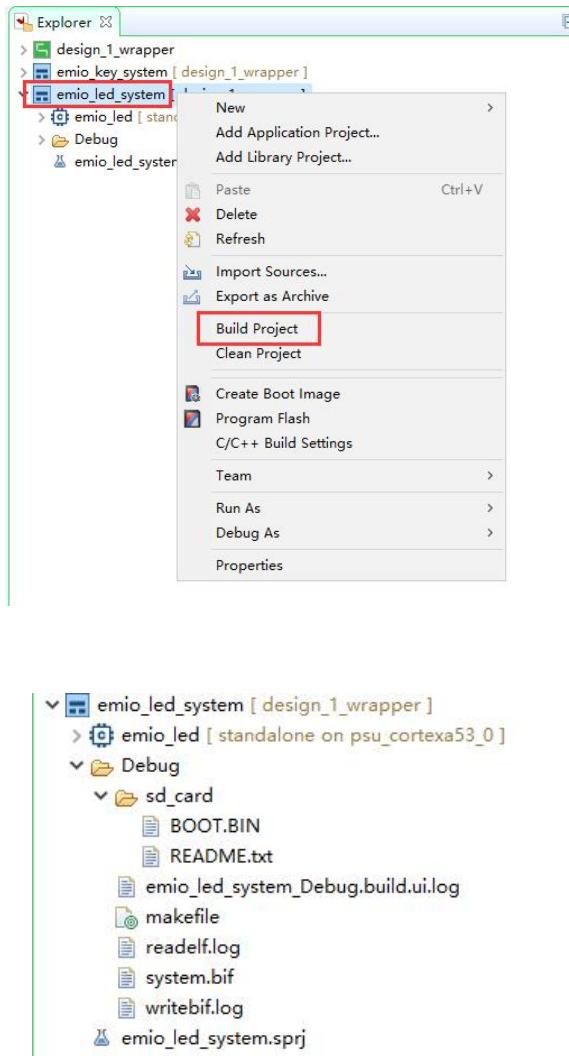


- 4) Observe the experimental phenomenon, press the PL end key PL\_KEY, you can control the PL end LED PL-LED on and off.

### Part 12.5: Build Project

We have introduced how to generate the firmware without the FPGA loading file (for details, refer to the chapter "Experiencing ARM, Bare Metal Output" Hello World"). The content of this chapter has generated the FPGA loading file, here demonstrates how to generate the firmware.

As before, click system and right click Build Project.

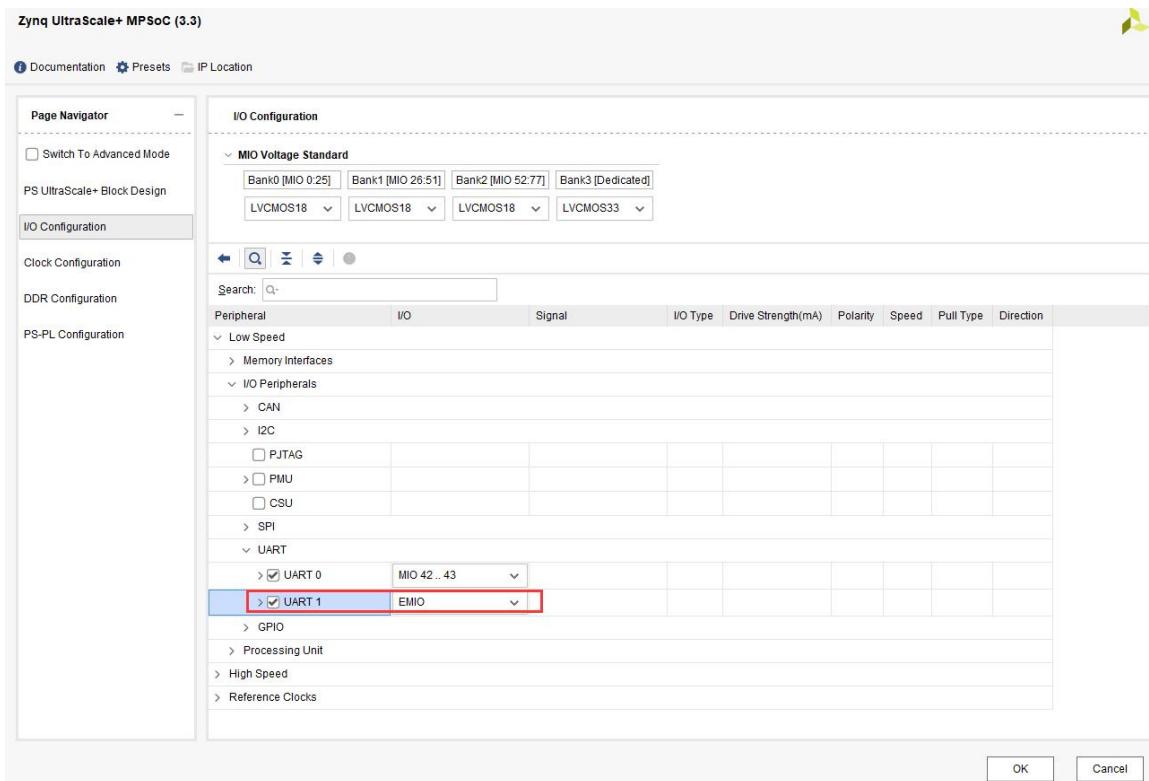


The software will automatically add three files, the first boot program is “fsbl.elf”, the second is the “bitstream” of FPGA, and the third is the application program “xx.elf”. Click “Create Image”, the download method is the same as above, and will not be repeated here.

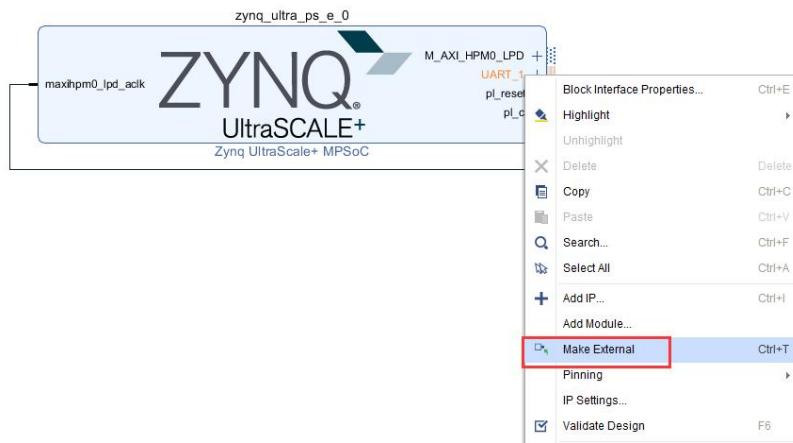
## Part 12.6: EMIO Usage of UART Serial Port

In addition to GPIO that can export EMIO, there are many MIOs on the PS side that can be connected to the PL side through EMIO. For example, if the PL side of the development board is connected to a UART, it can be connected in this way.

In the configuration, set UART1 to EMIO



## Export pin

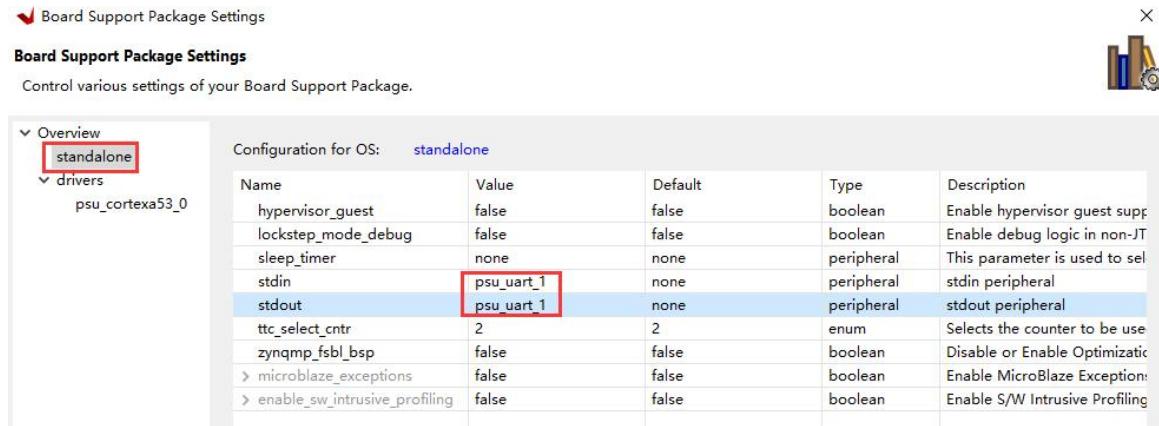


Bind the pins according to the schematic and recompile.

```
set_property PACKAGE_PIN AH11 [get_ports {uart_rxd}]
set_property IOSTANDARD LVCMS33 [get_ports {uart_rxd}]
```

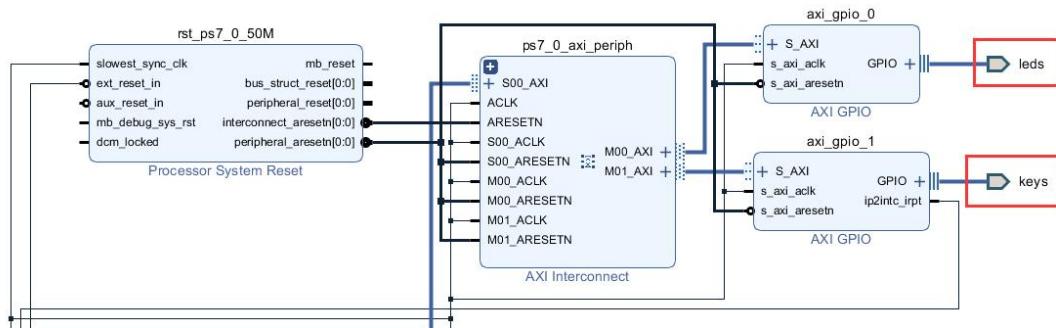
```
set_property PACKAGE_PIN AH12 [get_ports {uart_txd}]
set_property IOSTANDARD LVCMS33 [get_ports {uart_txd}]
```

The usage is the same as uart0, but if you want to use uart1 to print information, you need to set it to uart1 in the BSP

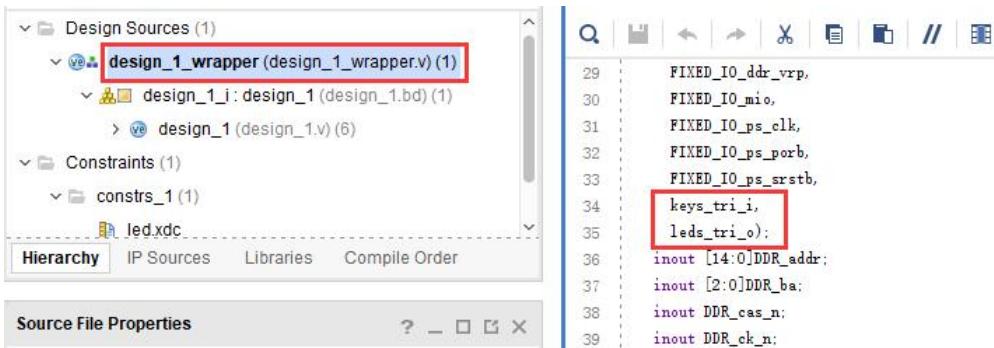


## Part 12.7: Pin Binding Common Errors

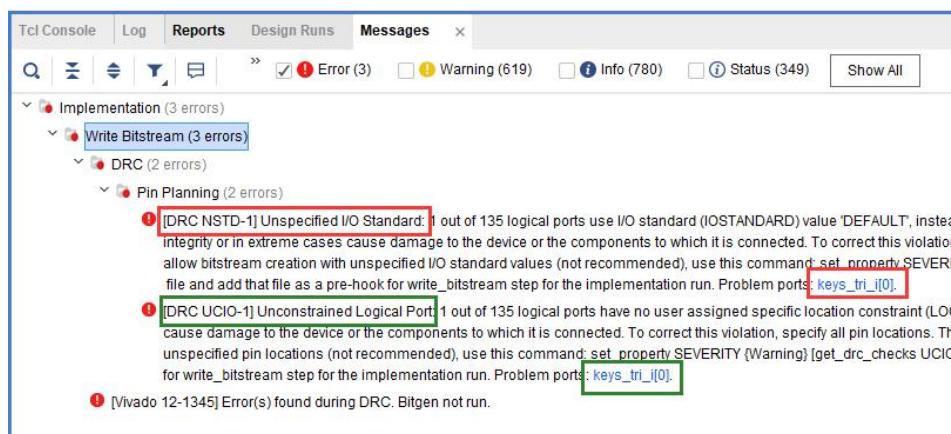
- 1) In the block design design, for example, the pin name of the GPIO module is set to leds and keys. Many people bind pins according to such names in XDC files.



If you open the top-level file, you will find that the pin names are different. Be sure to check it carefully. The pin names in the top-level file will prevail.



Otherwise, the following pin unbound error will occur



- 2) If you are writing by hand XDC files, remember to pay attention to spaces, this is also a very common mistake

```

1  set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[3]}]
2  set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[2]}]
3  set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[1]}]
4  set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[0]}]
5  set_property PACKAGE_PIN M14 [get_ports {leds_tri_o[0]}]
6  set_property PACKAGE_PIN M15 [get_ports {leds_tri_o[1]}]
7  set_property PACKAGE_PIN K16 [get_ports {leds_tri_o[2]}]
8  set_property PACKAGE_PIN J16 [get_ports {leds_tri_o[3]}]
9
10 set_property IOSTANDARD LVCMOS33 [get_ports {keys_tri_i[0]}]
11 set_property PACKAGE_PIN N15 [get_ports {keys_tri_i[0]}]

```

## Part 12.8: Experimental Summary

This chapter further studies the use of EMIO on the PS side. Although the EMIO is connected to the pins on the PL side, the usage in the SDK is the same. From this example, we can also see that once the connection with the PL side occurs, we need to generate bitstream, although produces almost no logic.

## Part 13: PL Side Use of AXI GPIO

The experimental Vivado project directory is "ps\_axi\_gpio /vivado"

The experimental vitis project directory is "ps\_axi\_gpio /vitis".

Some people may ask, how to talk about GPIO and LED lights, which is too cumbersome, but GPIO is the basic operation of ZYNQ. This tutorial tries to share various methods with you, MIO on PS side, EMIO, axi gpio on PL side, Including the two directions of input and output, as well as the basic operation of PS and PL, so I still hope that everyone will learn patience.

We introduced how to use the PS-side EMIO to light the PL-side LED, but did not interact with the PL-side. This chapter introduces another control method. You can use AXI GPIO in ZYNQ to control the LED lights on the PL side through the AXI bus. The use of the keys on the PL side is also introduced.

The biggest doubt in using zynq is how to use PS and PL together. GPIO is generally used in other SOC chips. This experiment uses an AXI GPIO IP core to let the PS side control the LED lights on the PL side through the AXI bus. Although the experiment is simple, it can let us understand how PL and PS are combined.

### Part 13.1: Principle Introduction

An AXI GPIO module has two GPIOs: GPIO and GPIO2, that is, channel1 and channel2, which are bidirectional IO.

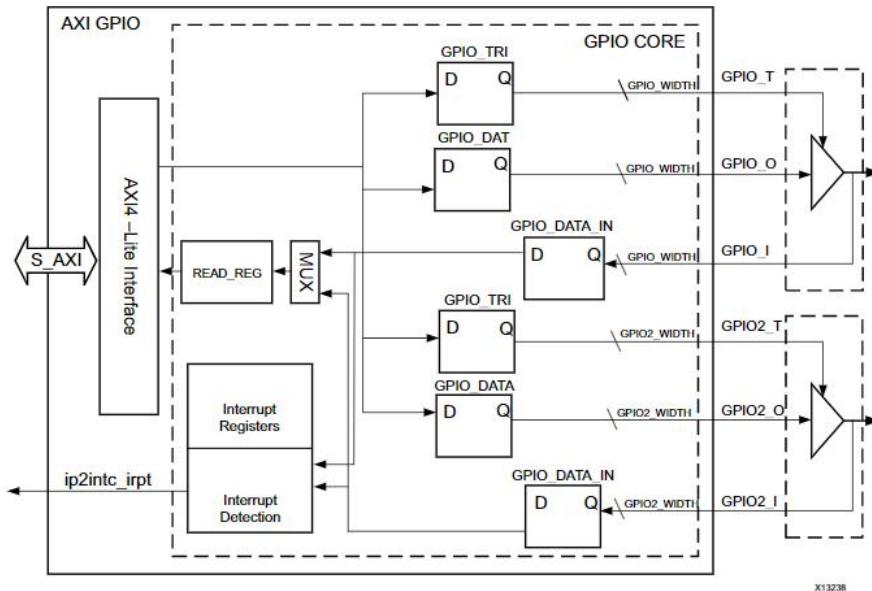


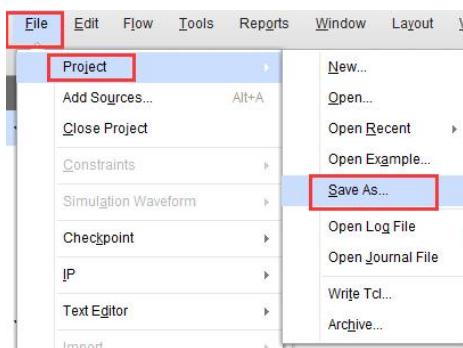
Figure 1-1: AXI GPIO Block Diagram

## FPGA Engineer Job Content

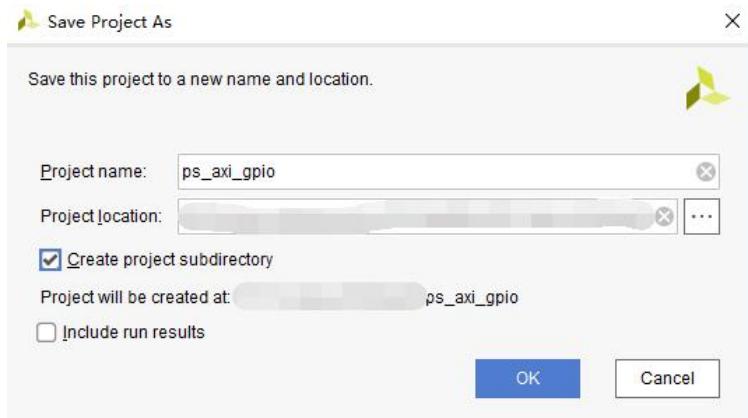
The following is the content that FPGA engineers are responsible for.

### Part 13.2.1: Create a “Vivado” Project

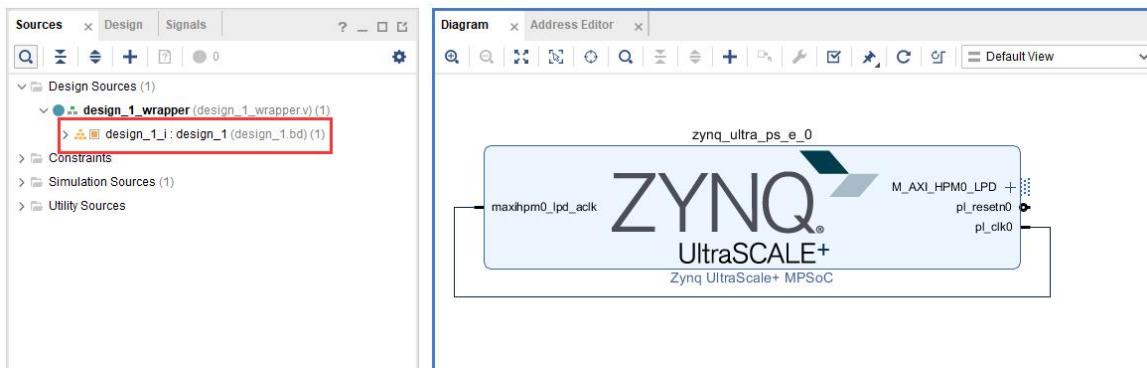
- 1) Open "ps\_hello" and save as a "ps\_axi\_gpio" Vivado project, indicating that PS controls gpio through the AXI bus



"Create project subdirectory" is checked to create a subdirectory under the directory, and "Include run results" is checked to include the compiled results

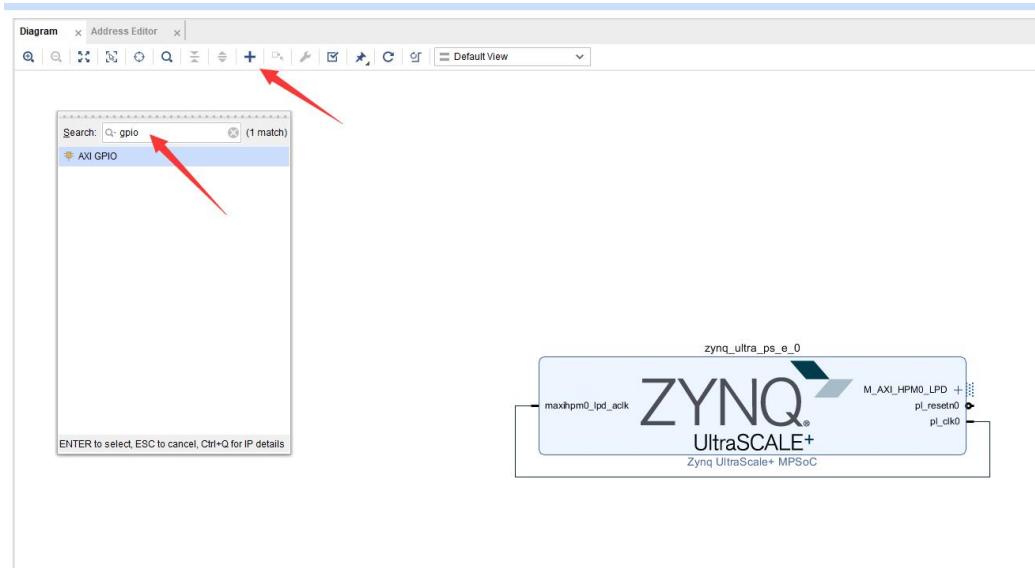


2) Double-click "xx.bd" to open the "block design"

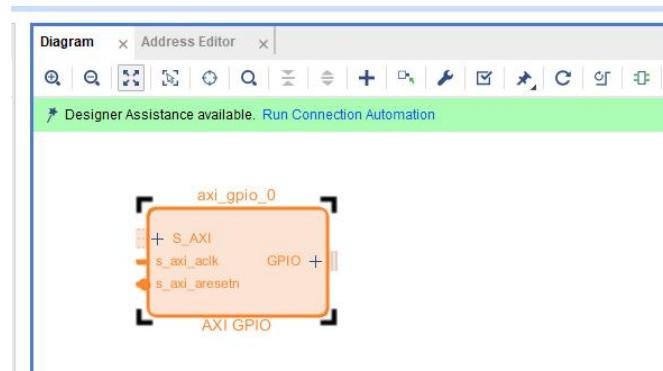


### Part 13.2.2: Add “AXI GPIO”

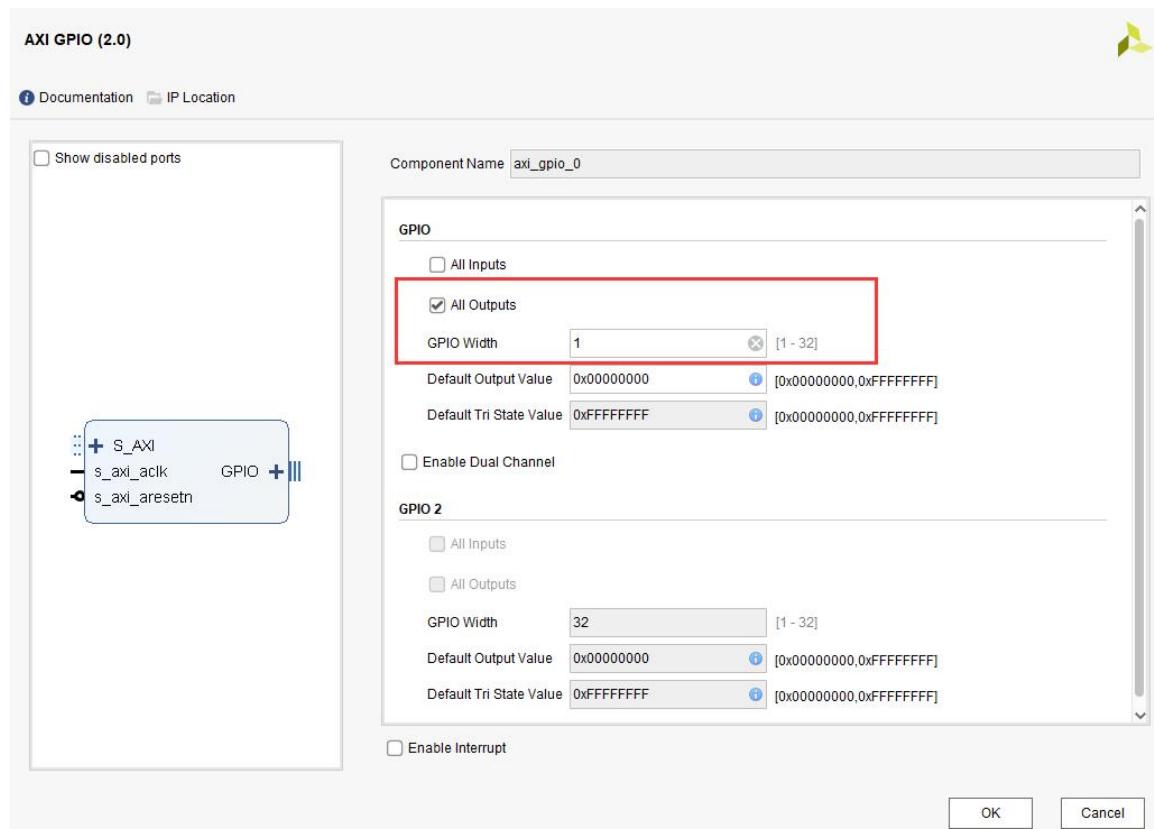
3) Add an AXI GPIO IP core



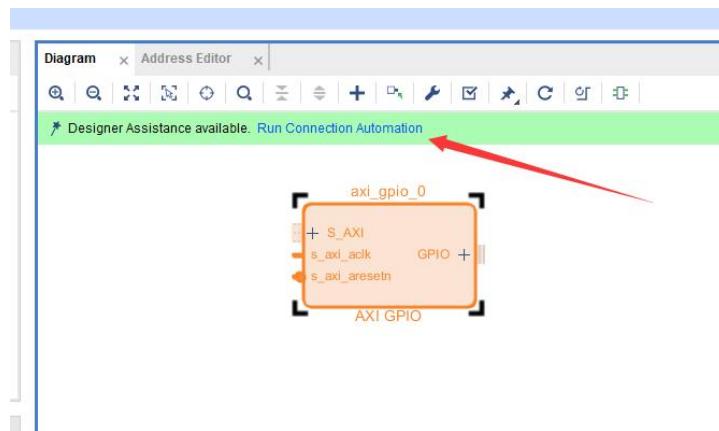
4) Double-click the "axi\_gpio\_0" just added, configure the parameters



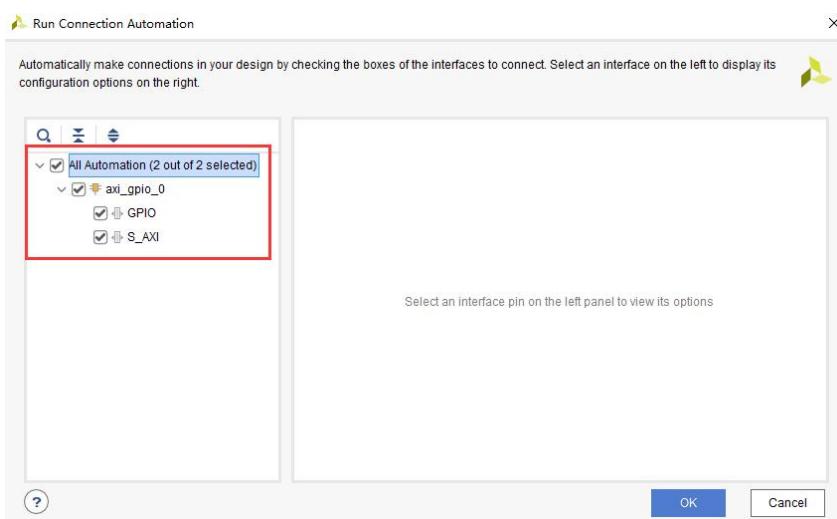
- 5) Select "All Outputs", because the LED is controlled here, as long as the output is OK. Fill in "GPIO Width" as 1, control 1 LED, and click OK. If you want to use "channel2", you need to turn on "Enable Dual Channel", which also enables GPIO2



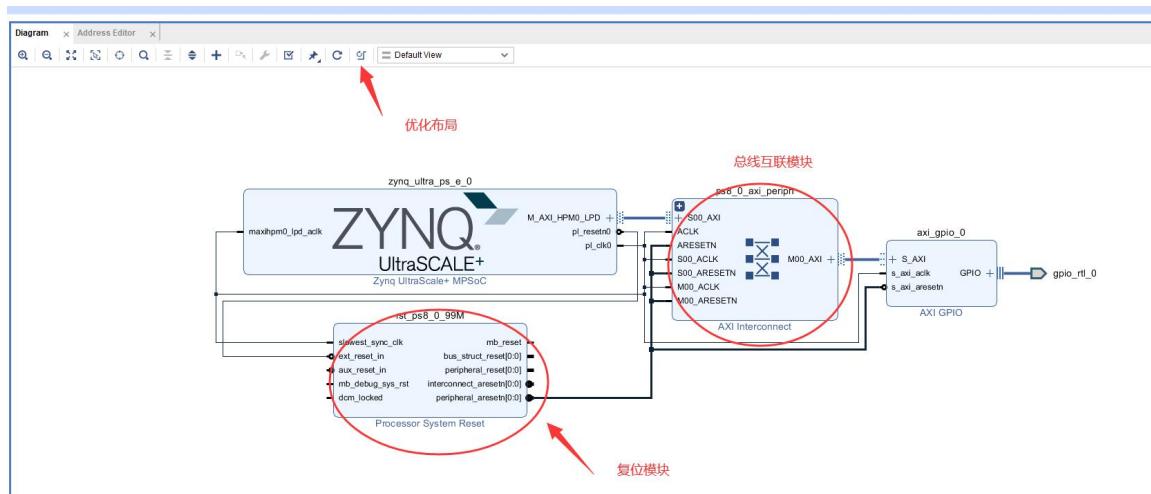
- 6) Click "Run Connection Automation" to complete some automatic connections



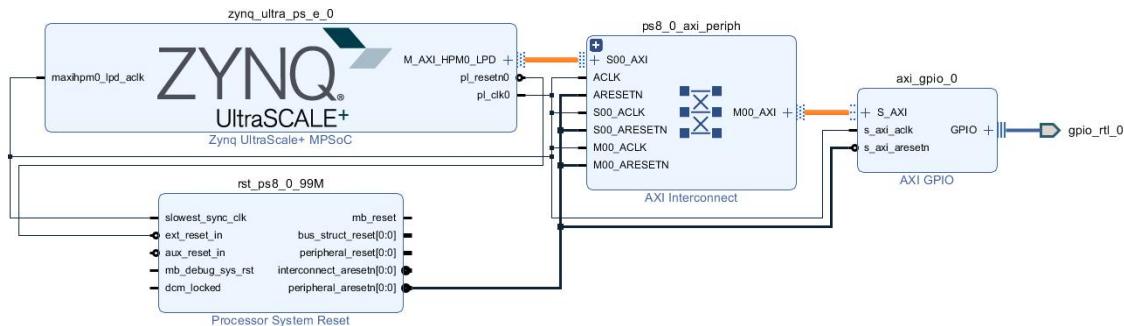
- 7) Select the port to be automatically connected, select all here, click OK



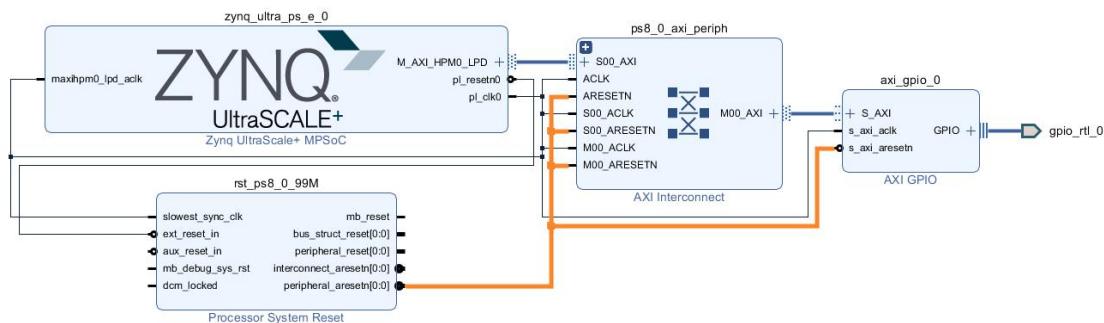
- 8) Click "Optimize Routing", you can optimize the layout, and you can see that there are two more modules. One is a “Processor System Reset” module, which provides a reset signal in the same clock domain as a synchronous reset module. The “AXI Interconnect module” is an AXI bus interconnection module, which is used for cross interconnection of AXI modules.



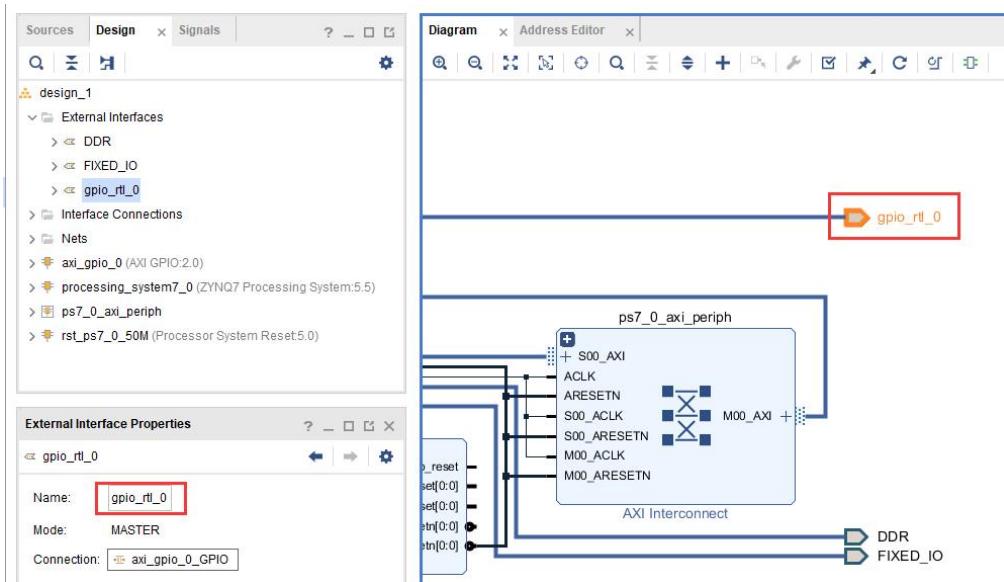
In this application, we can see that the HPM0\_LPD port of ZYNQ is used. This interface is used to access PL data. In most applications, it is used to configure the registers of the PL module.



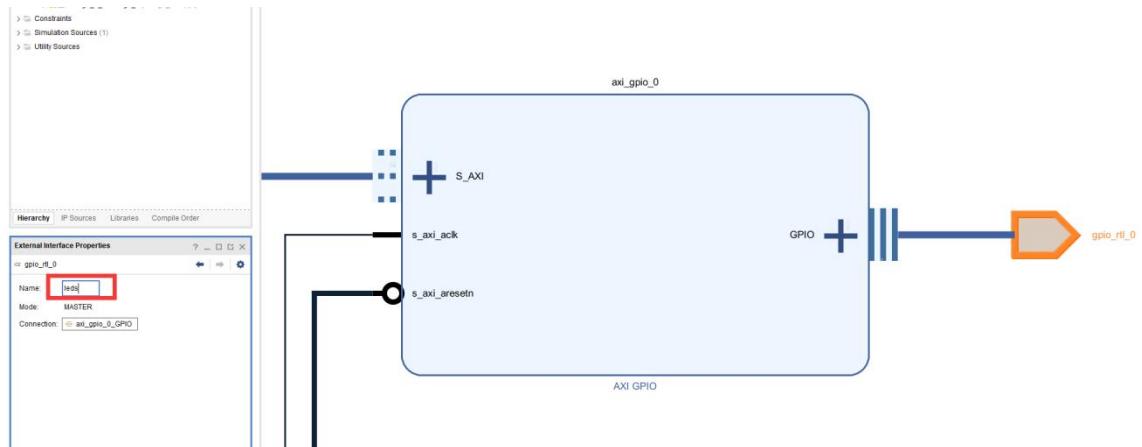
The reset signal is provided by the reset output of ZYNQ. It is best to add a reset module for each clock domain. You can search and add according to the name of the module.



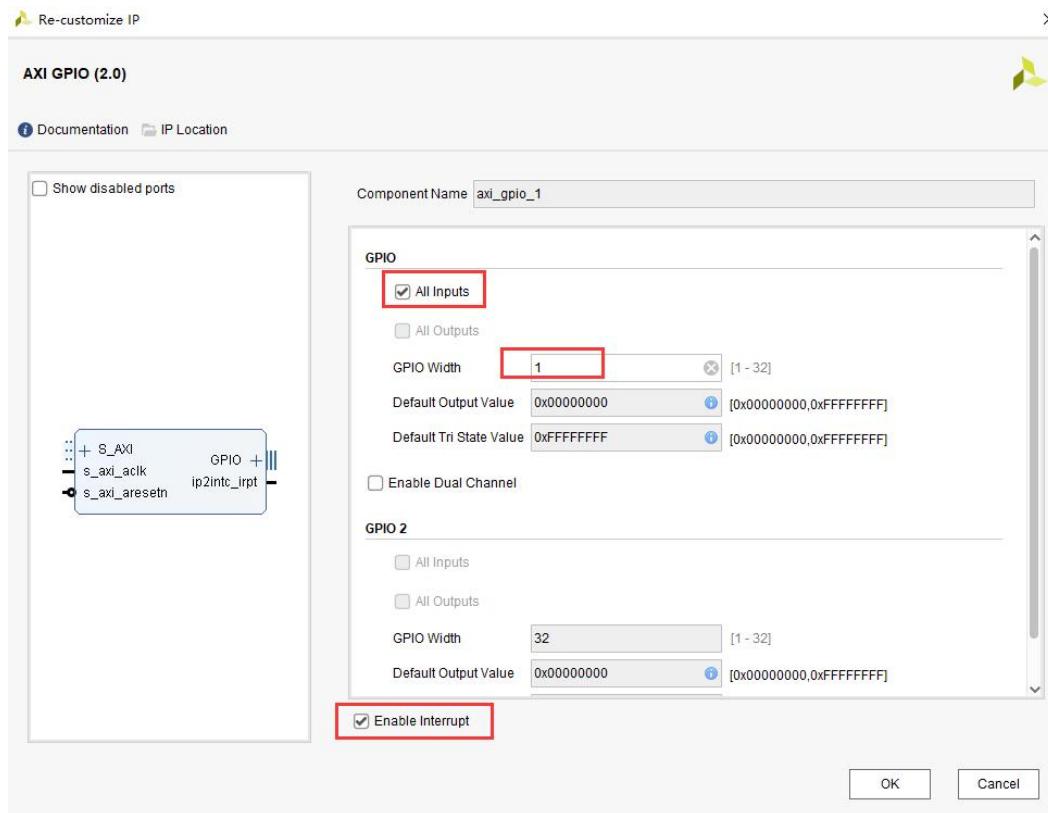
## 9) Modify the name of the GPIO port



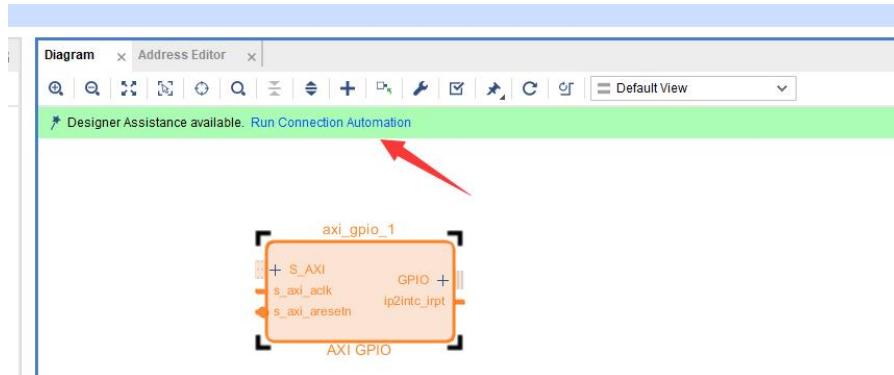
## 10) Name changed to leds



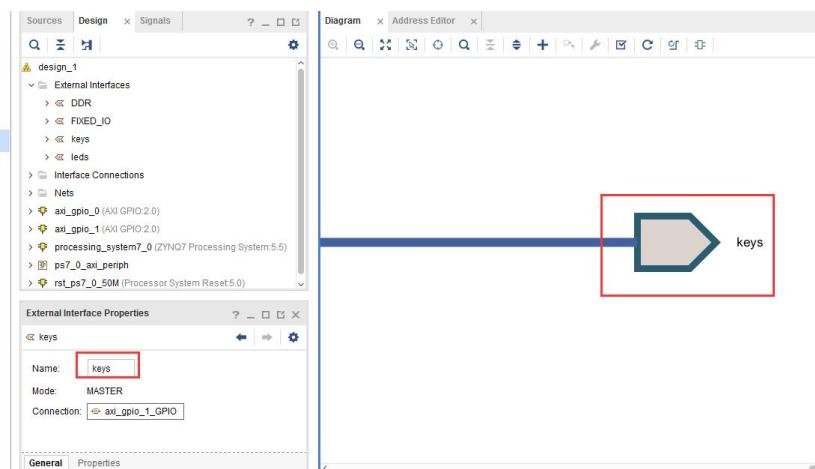
## 11) Add another “AXI GPIO”, connect the “PL” side key, configure “GPIO” parameters, all of which are input, width is 1, enable interrupt



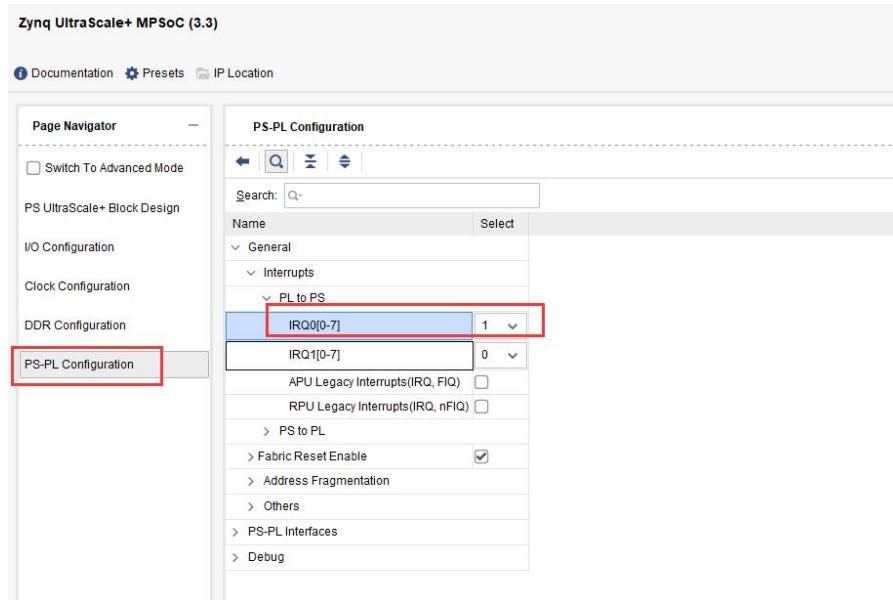
## 12) Use automatic connection



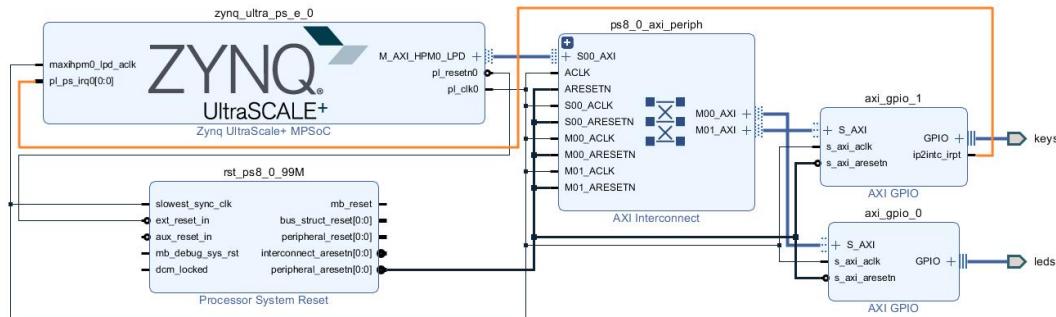
## 13) Change the port name to "keys"



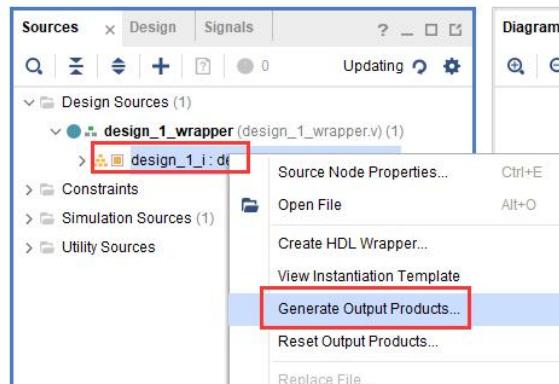
14) Since it is an interrupt from the PL side, here you need to configure the interrupt of the ZYNQ processor, Set IRQ0[0-7] to 1.



15) Connect “ip2intc\_irpt” to “pl\_ps\_irq”

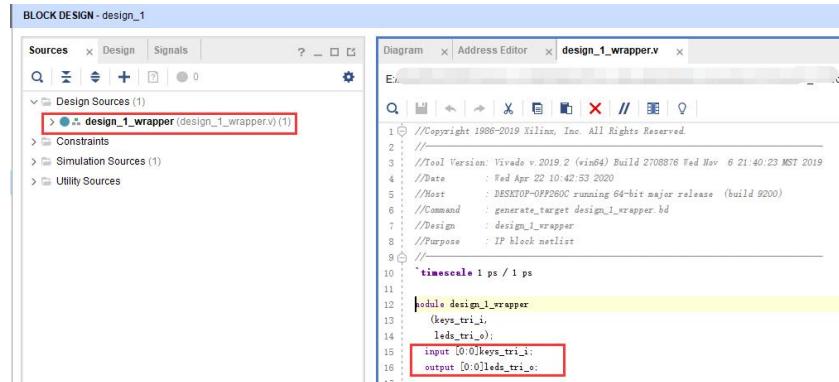


16) Save the design, click “xx.bd”, right-click “Generate Output Products”



17) In the generated “Verilog” file, you can see that there are ports of

"leds\_tri\_o" and "keys\_tri\_i". You must assign pins to them. When binding the pins, the pin names in this file is correct and should same with it.



```

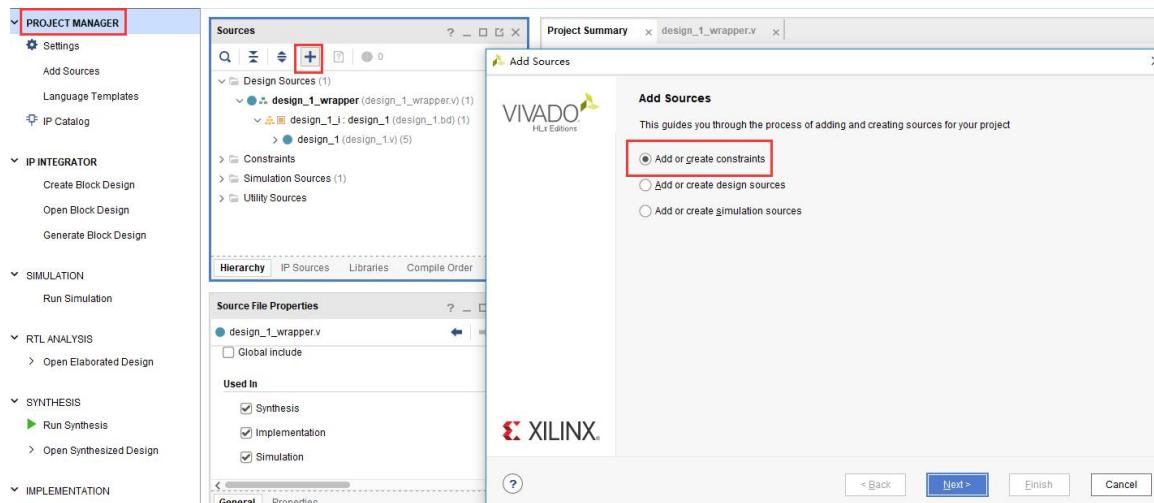
BLOCK DESIGN - design_1
Sources | Design | Signals | ? - □ 0
Design Sources (1)
> design_1_wrapper (design_1_wrapper.v)(1)
Constraints
Simulation Sources (1)
Utility Sources

Diagram x Address Editor x design_1_wrapper.v x
Diagram | Address Editor | design_1_wrapper.v | 
1 //Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
2 //
3 //Tool Version: Vivado v.2019.2 (win64) Build 2708976 Wed Nov 6 21:40:23 MST 2019
4 //Date : Wed Apr 22 10:42:53 2020
5 //Host : DESKTOP-OPP260C running 64-bit major release (build 9200)
6 //Command : generate_target design_1_wrapper.bd
7 //Design : design_1_wrapper
8 //Purpose : IP block netlist
9 //
10 'timescale 1 ps / 1 ps
11
12 module design_1_wrapper
13   (keys_tri_i,
14    leds_tri_o);
15   input [0:0]keys_tri_i;
16   output [0:0]leds_tri_o;
17

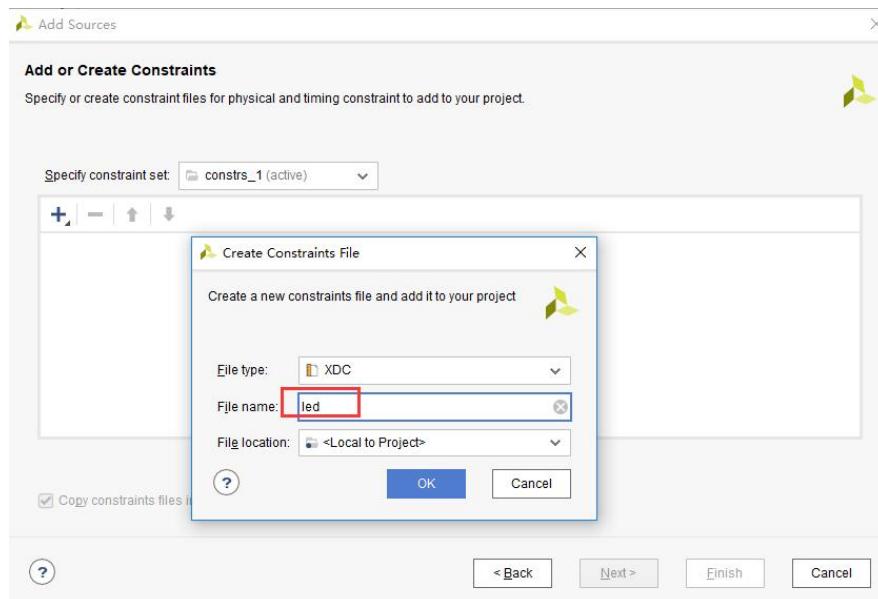
```

### Part 13.3: XDC File Constraint PL Pin

#### 1) Create a new xdc constraint file



#### 2) File name is led



- 3) “led.xdc” add the following content, the port name must be the same as the top-level file port

```
#####Compress Bitstream#####
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]

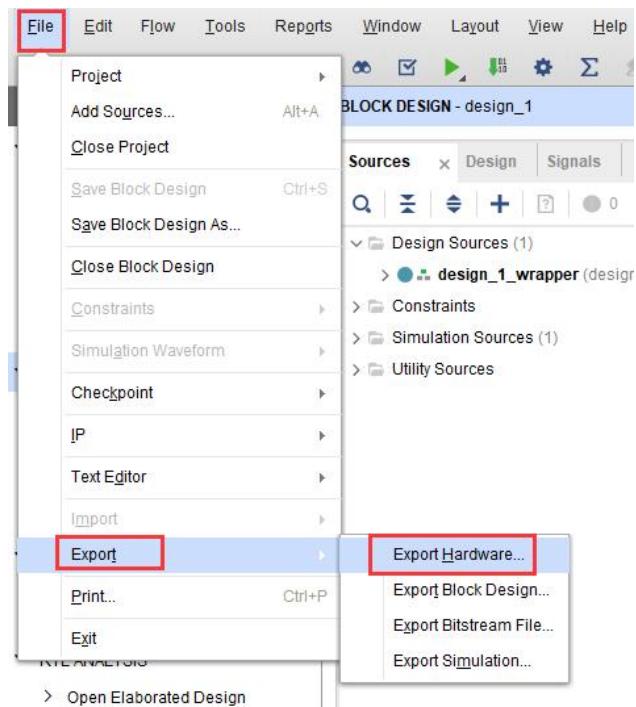
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[0]}]
set_property PACKAGE_PIN AE12 [get_ports {leds_tri_o[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {keys_tri_i[0]}]
set_property PACKAGE_PIN AF12 [get_ports {keys_tri_i[0]}]
```

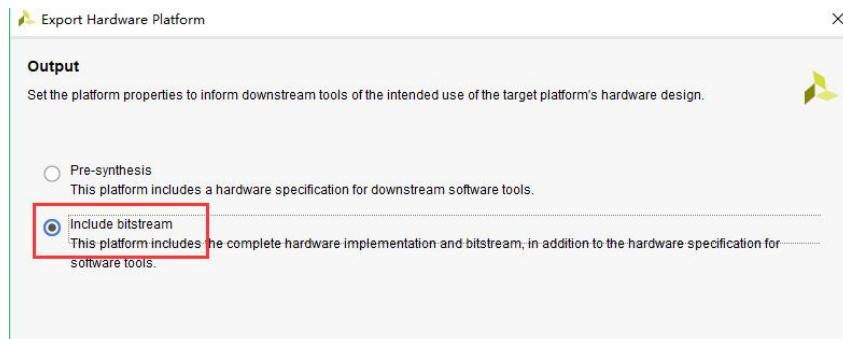
- 4) Generate bit file



- 5) Export the hardware “File→Export→Export Hardware”



- 6) Since PL is used, select "Include bitstream" and click "OK"



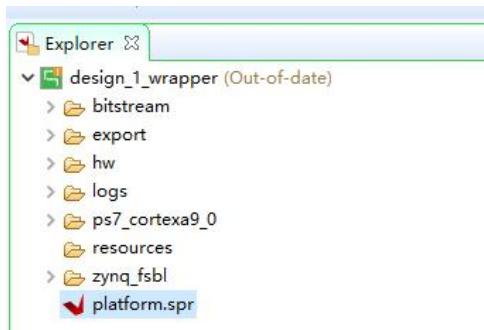
## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

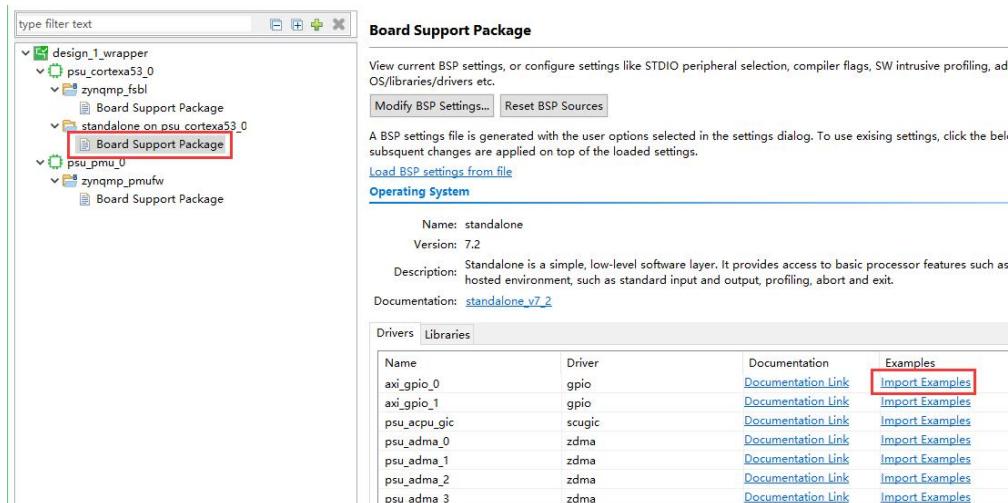
### Part 13.4: Vitis Programming

#### Part 13.4.1: AXI GPIO lights PL LED

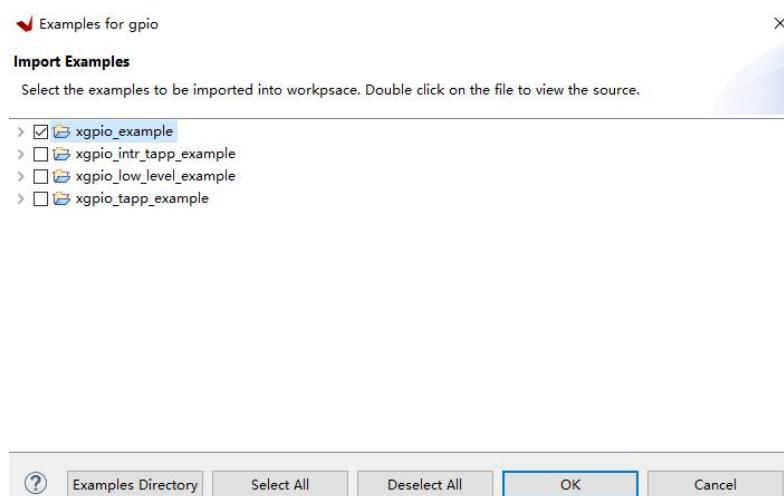
- 1) Create a platform, refer to the chapter "PS Side RTC Interrupt Experiment" for the creation process



- 2) Faced with an unfamiliar “AXI GPIO”, how do we control it? We can try the routines that come with the Vitis
- 3) Double-click "system.mss" and find "axi\_gpio\_0". You can click "Documentation" here to see related documents. I won't show it here. Click "Import Examples"



- 4) There are multiple routines in the pop-up dialog box, you can guess from the name, here select the first "xgpio\_example"



5) You can see that the routine is relatively simple, just a few lines of code, and completed the operation of the AXI GPIO

The screenshot shows the Vitis IDE interface with the following details:

- Project Explorer:** Shows the project structure:
  - design\_1\_wrapper (Out-of-date)
  - system\_bsp\_example\_1 [design\_1\_wrapper]
  - xgpio\_example\_1 [ standalone\_domain ]
    - Includes
    - \_ide
    - src
      - xgpio\_example.c
      - lscript.tcl
      - Xilinx.spec
      - README.txt
      - xgpio\_example\_1.prj
    - system\_bsp\_example\_1.sprj
- Code Editor:** Displays the xgpio\_example.c file with the following C code:

```
147 if (Status != XST_SUCCESS) {
148     xil_printf("Gpio Initialization Failed\r\n");
149     return XST_FAILURE;
150 }
151
152 /* Set the direction for all signals as inputs except the LED output */
153 XGpio_SetDataDirection(&Gpio, LED_CHANNEL, ~LED);
154
155 /* Loop forever blinking the LED */
156
157 while (1) {
158     /* Set the LED to High */
159     XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, LED);
160
161     /* Wait a small amount of time so the LED is visible */
162     for (Delay = 0; Delay < LED_DELAY; Delay++);
163
164     /* Clear the LED bit */
165     XGpio_DiscreteClear(&Gpio, LED_CHANNEL, LED);
166
167     /* Wait a small amount of time so the LED is visible */
168     for (Delay = 0; Delay < LED_DELAY; Delay++);
169 }
170
171 }
```
- Console:** Shows the build console output for system\_bsp\_example\_1\_Debug.

There are many GPIO related API functions used in it. You can learn more about it through the documentation, or you can select this function and press "F3" to see the specific definition. If you don't understand how to use AXI GPIO with this information, it means that you need to supplement the C language foundation.

In fact, these functions are all operating GPIO registers, not many AXI GPIO registers. It is Mainly the data registers “GPIO\_DATA” and “GPIO2\_DATA” of two channels. The directions of the two channels control “GPIO\_TRI” and “GPIO2\_TRI”, as well as the global interrupt enable register GIER, IP interrupt enable IP IER, and interrupt status register ISR. For specific functions, see page 144 of the AXI GPIO documentation.

Table 2-4: Registers

Address Space Offset <sup>(3)</sup>	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register.
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register.
0x0008	GPIO2_DATA	R/W	0x0	Channel 2 AXI GPIO Data Register.
0x000C	GPIO2_TRI	R/W	0x0	Channel 2 AXI GPIO 3-state Control.
0x011C	GIER <sup>(1)</sup>	R/W	0x0	Global Interrupt Enable Register.
0x0128	IP IER <sup>(1)</sup>	R/W	0x0	IP Interrupt Enable Register (IP IER).
0x0120	IP ISR <sup>(1)</sup>	R/TOW <sup>(2)</sup>	0x0	IP Interrupt Status Register.

For example, when entering the function of setting the “GPIO” direction, you can see that you are writing data to the “GPIO\_TRI” register to control the direction.

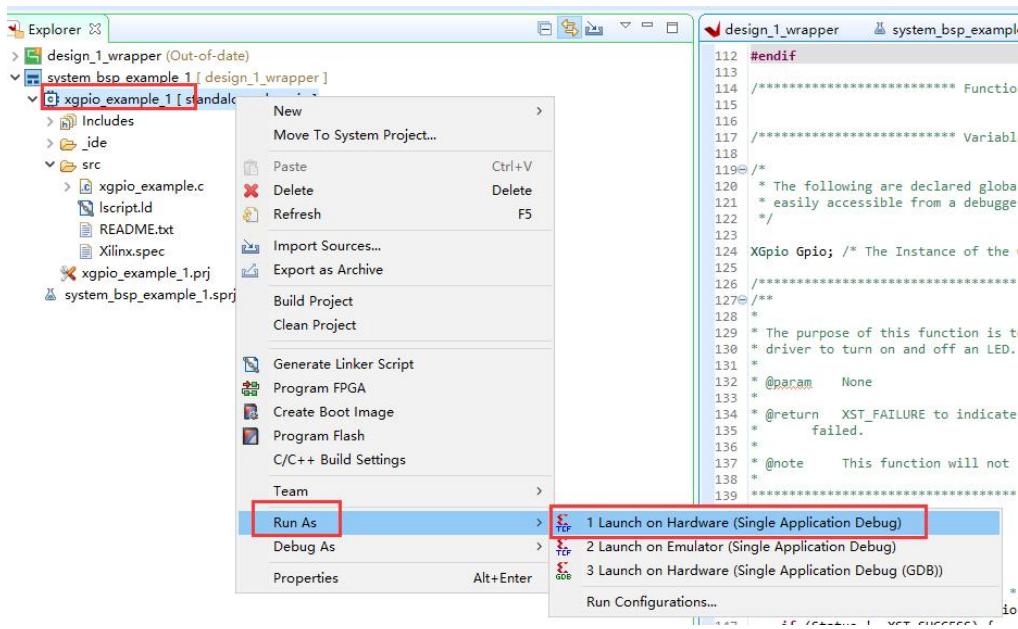
```
@ void XGpio_SetDataDirection(XGpio *InstancePtr, unsigned Channel,
                               u32 DirectionMask)
{
    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
    Xil_AssertVoid((Channel == 1) ||
                  ((Channel == 2) && (InstancePtr->IsDual == TRUE)));

    XGpio_WriteReg(InstancePtr->BaseAddress,
                   ((Channel - 1) * XGPIO_CHAN_OFFSET) + XGPIO_TRI_OFFSET,
                   DirectionMask);
}
```

Other functions can be studied by this method.

### Part 13.4.2: Download and Debug

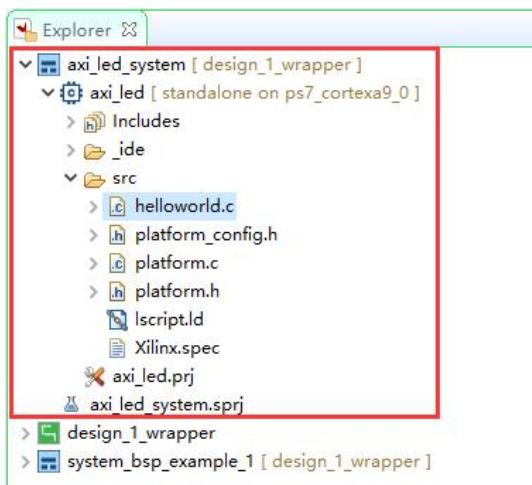
- 1) First compile the APP project, the compilation method has been introduced in the previous example. Although the Vitis can provide some routines, some of them need to be modified by yourself. This simple LED routine is not modified. Try to run it and find that it does not achieve the expected results, and even prompts some errors. After downloading, you can see the rapid flashing of the development board PL LED.



### Part 13.4.3: Register Mode Implementation

If you feel that the API functions provided by Xilinx are tedious and inefficient, you can also implement the control of LEDs by operating registers.

For example, below we have created a new axi\_led project and modified helloworld.c as follows.



```

#define GPIO_BASEADDR 0x80000000    GPIO基地址
#define DATA_OFFSET 0x0             偏移地址
#define TRI_OFFSET 0x4

#define LED_DELAY 10000000

int main()
{
    volatile int Delay;
    /* Set the direction for all signals as outputs */
    *(int*)(GPIO_BASEADDR + TRI_OFFSET) = 0x0 ;

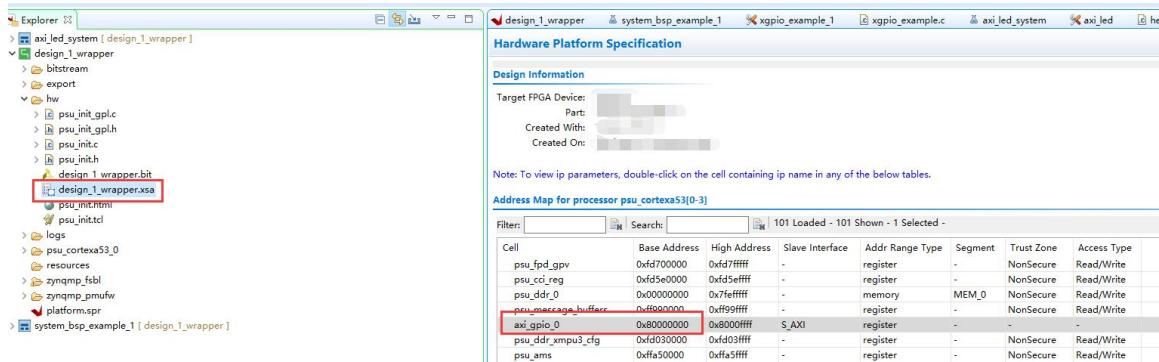
    while (1)
    {
        /* Set the LED to High */
        *(int*)(GPIO_BASEADDR + DATA_OFFSET) = 0xF ;    向数据寄存器
        /* Wait a small amount of time so the LED is visible */
        for (Delay = 0; Delay < LED_DELAY; Delay++);

        /* Set the LED to Low */
        *(int*)(GPIO_BASEADDR + DATA_OFFSET) = 0x0 ;
        /* Wait a small amount of time so the LED is visible */
        for (Delay = 0; Delay < LED_DELAY; Delay++);
    }

    return 0;
}

```

The base address “GPIO\_BASEADDR” defined in it can be found in “xxx.xsa”



Since we only enable “channel1”, the following register address is defined

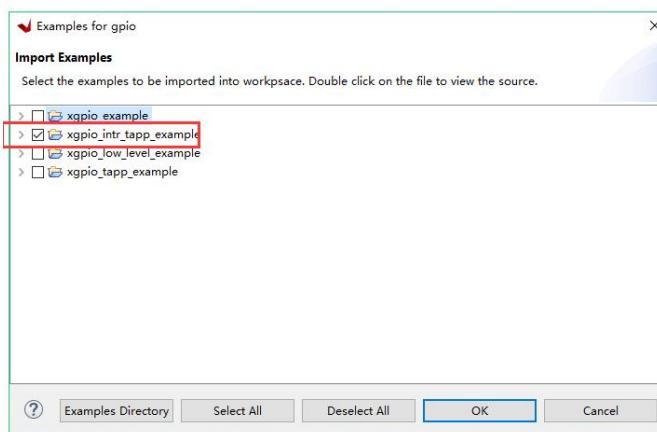
Address Space Offset <sup>(3)</sup>	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register.
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register.

In this way, the way of directly operating the register will be more efficient than calling Xilinx API functions, and it is more intuitive, which is very helpful for understanding how the program runs. However, for large projects, this method is more complicated to use and is mainly selected based on individual needs.

#### Part 13.4.4: PL Side AXI GPIO Key Interrupt

The interrupt of the previous timer interrupt experiment belongs to the PS internal interrupt. In this experiment, the interrupt comes from the PL. The PS can receive up to 16 interrupt signals from the PL, which are triggered by rising edges or high levels

- 1) As in the previous tutorial, if you are not familiar with Vitis programming, we try to use the Vitis's own routines to modify it. Select“xgpio\_intr\_tapp\_example”



- 2) Part of the code needs to be modified, the button axi gpio module is called axi\_gpio\_1, find its “device id” in “xparameters.h”

```
221 /* **** Definitions for driver GPIO ****/
222
223 /* Definitions for peripheral AXI_GPIO_0 */
224 #define XPAR_XGPIO_NUM_INSTANCES 2
225
226 /* Definitions for peripheral AXI_GPIO_0 */
227 #define XPAR_AXI_GPIO_0_BASEADDR 0x41200000
228 #define XPAR_AXI_GPIO_0_HIGHADDR 0x4120FFFF
229 #define XPAR_AXI_GPIO_0_DEVICE_ID 0
230 #define XPAR_AXI_GPIO_0_INTERRUPT_PRESENT 0
231 #define XPAR_AXI_GPIO_0_IS_DUAL 0
232
233
234 /* Definitions for peripheral AXI_GPIO_1 */
235 #define XPAR_AXI_GPIO_1_BASEADDR 0x41210000
236 #define XPAR_AXI_GPIO_1_HIGHADDR 0x41211FFF
237 #define XPAR_AXI_GPIO_1_DEVICE_ID 1
238 #define XPAR_AXI_GPIO_1_INTERRUPT_PRESENT 1
239 #define XPAR_AXI_GPIO_1_IS_DUAL 0
240
```

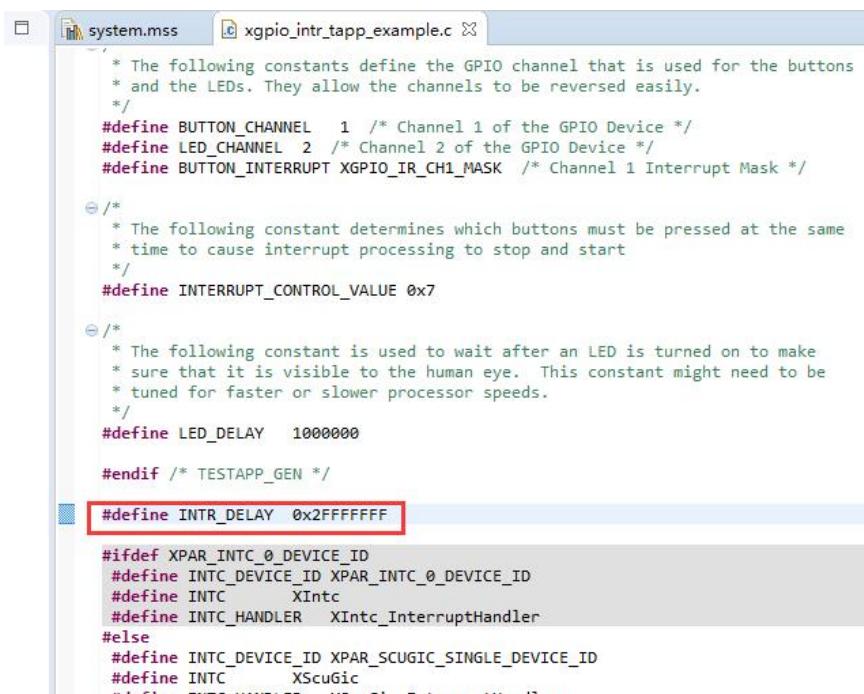
```
997
998
999 /* **** Definitions for Fabric interrupts connected to psu_acpu_gic ****/
1000
1001 /* Definitions for Fabric interrupts connected to psu_acpu_gic */
1002 #define XPAR_FABRIC_AXI_GPIO_1_IP2INTC_IRPT_INTR 121U
1003
```

- 3) Then you can modify the macro definition of GPIO and interrupt number as follows

```
#define GPIO_DEVICE_ID XPAR_GPIO_1_DEVICE_ID
#define GPIO_CHANNEL1 1

#ifndef XPAR_INTC_0_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID XPAR_INTC_0_GPIO_0_VEC_ID
#define INTC_DEVICE_ID XPAR_INTC_0_DEVICE_ID
#else
#define INTC_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_1_IP2INTC_IRPT_INTR
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
#endif /* XPAR_INTC_0_DEVICE_ID */
```

- 4) Modify the test delay time so that we have enough time to press the button



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

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

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

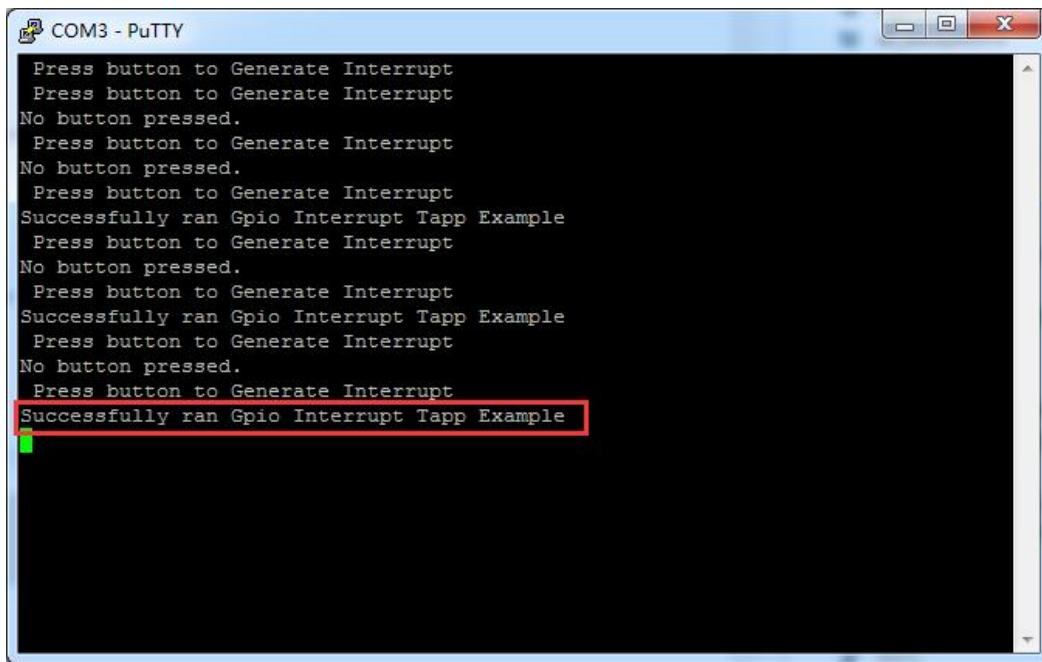
#endif /* TESTAPP_GEN */

#define INTR_DELAY 0xFFFFFFFF

#ifndef XPAR_INTC_0_DEVICE_ID
#define INTC_DEVICE_ID XPAR_INTC_0_DEVICE_ID
#define INTC_XIntc
#define INTC_HANDLER XIntc_InterruptHandler
#else
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
#define INTC_Xscugic
#define INTC_HANDLER Xscugic_InterruptHandler
#endif
```

#### Part 13.4.5: Download and Debug

Save the file, compile the project, open the serial terminal, and download the program. If you do not press the key, the serial port displays "No button pressed." If you press the "PL KEY1" key, it displays "Successfully ran Gpio Interrupt Tapp Example"



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

### Part 13.5: Experimental summary

Through experiments, we learned that PS can control PL through AXI bus, but it has hardly reflected the advantages of ZYNQ, because it can be easily completed for controlling LED lights, whether it is ARM or FPGA, but if you change the LED to a serial port, control 100 Serial communication, 8 Ethernet and other applications, I don't think any SOC can complete this function, only ZYNQ can, this is the difference between ZYNQ and ordinary SOC.

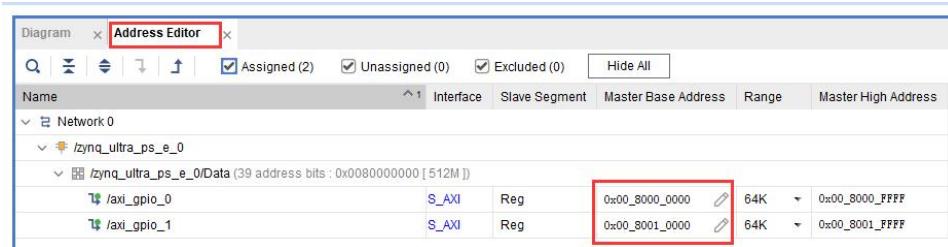
The PL end can send interrupt signals to the PS, which improves the efficiency of PL and PS data interaction. Interrupt processing is needed in applications that require large numbers and low latency.

By the end of this chapter, I have already explained how to use the PS-side MIO, EMIO, and PL-side GPIO of ZYNQ, including input and output, and interrupt handling. These are the most basic operations. You still have to think and understand.

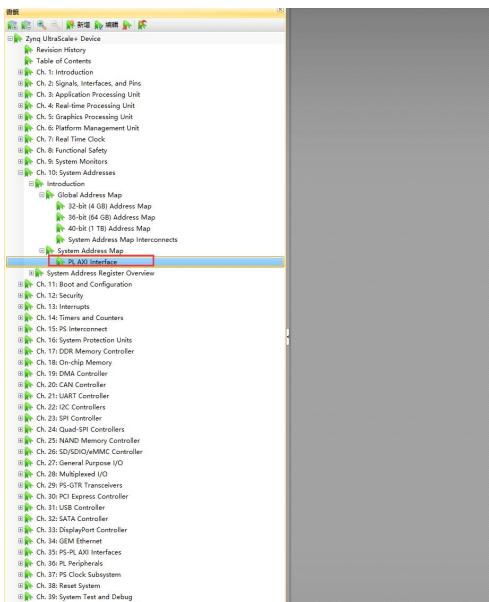
### Part 13.6: Knowledge Sharing

- 1) After designing, you can see that the address space has been

allocated for the AXI peripherals in the Address Editor, where the offset address and space size can be modified



However, there is a limit to modifying the offset address. For details, refer to the System Address chapter of the UG585 document. The AXI peripheral is connected to the M\_AXI\_GP0 port and modified in the 8000\_0000 to 9FFF\_FFFF address space.



#### PL AXI Interface

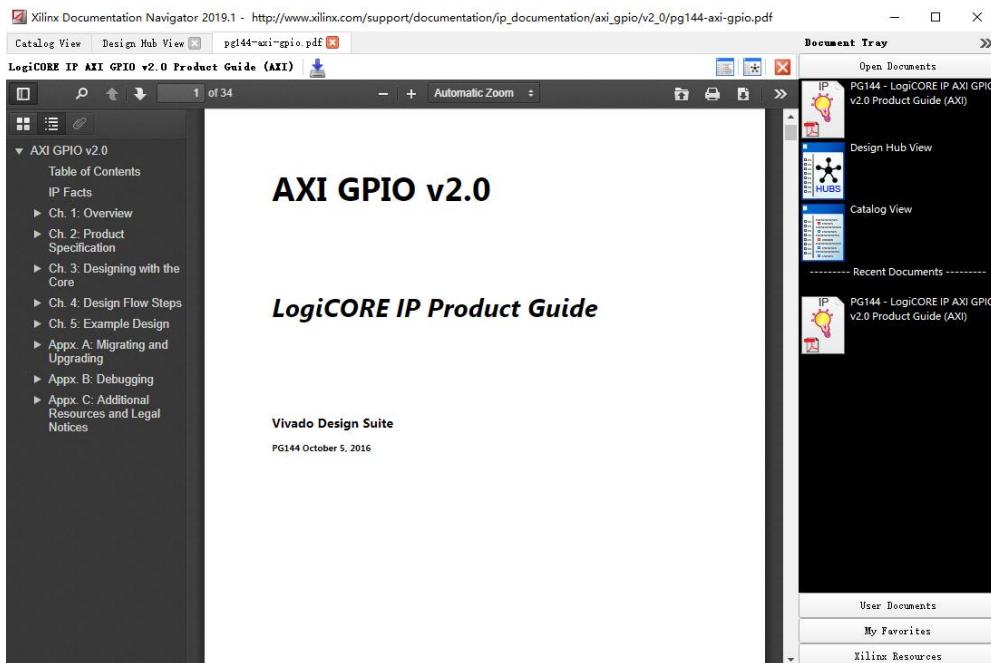
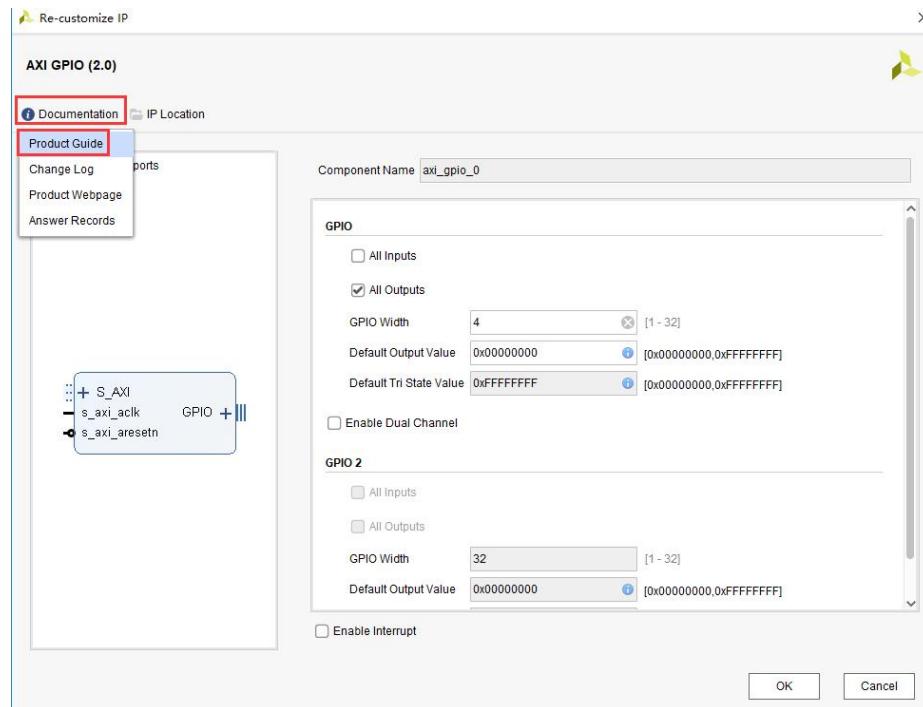
The AXI interface from the LPD to PL is assigned a fixed address space of 512 MB in the lower 4 GB address space. It is typically used for LPD to PL communications because it provides a low-latency path from the LPD masters like the RPU and the LPD DMA unit to the PL. The AXI interfaces from the FPD to PL are assigned multiple address ranges.

The comprehensive system-level addresses map is shown in Table 10-1.

Table 10-1: Top-Level System Address Map

Slave Name	Size	Start Address	End Address
DDR Low	2 GB	0x0000_0000	0x7FFF_FFFF
M_AXI_HPM0_LPD (LPD_PL)	512 MB	0x8000_0000	0x9FFF_FFFF
VCU <sup>(1)</sup>	64 MB	0xA000_0000	0xA3FF_FFFF
M_AXI_HPM0_FPD (HPM0) interface <sup>(1)</sup>	192 MB	0xA400_0000	0xAFFF_FFFF
M_AXI_HPM1_FPD (HPM1) interface	256 MB	0xB000_0000	0xBF00_0000
Quad-SPI	512 MB	0xC000_0000	0xDFFF_FFFF
PCIe Low	256 MB	0xE000_0000	0xEFFF_FFFF
Reserved	128 MB	0xF000_0000	0xF7FF_FFFF
STM CoreSight	16 MB	0xF800_0000	0xF8FF_FFFF
APU GIC	1 MB	0xF900_0000	0xF90F_FFFF
Reserved	63 MB	0xF910_0000	0xFCFF_FFFF
FPD slaves	16 MB	0xFD00_0000	0xFFFF_FFFF
Upper LPD slaves	16 MB	0xFE00_0000	0xFEFF_FFFF
Lower LPD slaves	12 MB	0xFF00_0000	0xFFBF_FFFF
CSU, PMU, TCM, OCM	4 MB	0xFFC0_0000	0xFFFF_FFFF
Reserved	12 GB	0x0001_0000_0000	0x0003_FFFF_FFFF
M_AXI_HPM0_FPD (HPM0)	4 GB	0x0004_0000_0000	0x0004_FFFF_FFFF
M_AXI_HPM1_FPD (HPM1)	4 GB	0x0005_0000_0000	0x0005_FFFF_FFFF
PCIe High	8 GB	0x0006_0000_0000	0x0007_FFFF_FFFF
DDR High	32 GB	0x0008_0000_0000	0x000F_FFFF_FFFF
M_AXI_HPM0_FPD (HPM0)	224 GB	0x0010_0000_0000	0x0047_FFFF_FFFF
M_AXI_HPM1_FPD (HPM1)	224 GB	0x0048_0000_0000	0x007F_FFFF_FFFF
PCIe High	256 GB	0x0080_0000_0000	0x00BF_FFFF_FFFF

- 2) When using a module, supporting documents are needed to assist in development, but how to find these documents, such as the IP of XILINX, open the module configuration, click Documentation in the upper left corner, and then click Product Guide. If DocNav is installed when installing Vivado, it will jump to open the document.



- 3) This function requires a computer to connect to the Internet, and DocNav will load the document from the website. You can click the

download button to download to local. Another method is to search for the information on the Xilinx official website according to the name of the module to download (the page may change)

The screenshot shows the Xilinx website's search interface. At the top, there is a navigation bar with links for 应用 (Applications), 产品 (Products), 开发者 (Developers), 技术支持 (Technical Support), and 关于 Xilinx (About Xilinx). Below the navigation bar is a search bar with a placeholder '网站关键字搜索' (Search website keywords) and a magnifying glass icon. A red box highlights the search input field. The main content area is titled '网站关键字搜索' (Website Keyword Search) and contains a search bar with the query 'axi gpio'. To the right of the search bar is a red search button. On the left, there is a sidebar titled 'Filter Results' with a '结果类型' (Result Type) dropdown set to 'Document'. Below it is a list of categories with counts: Document (278), Forums (93), Answer Record (51), Product Information (2), and Partner Information (1). In the center, the search results are displayed. The first result is highlighted with a red box and labeled 'PG144 - AXI GPIO v2.0 Product Guide (v2.0)'. It includes a thumbnail, the title, the date 'Oct 05, 2016', and a 'See All Versions' link. Below the result, there is a brief description: 'This core provides a general purpose input/output interface to the AXI interface. This 32-bit soft Intellectual Property (IP) core is designed to interface with the AXI4-Lite interface.'

## Part 14: PL Side RS485 Test

The experimental Vivado project directory is "rs485\_test/vivado".

The experiment vitis project directory is "rs485\_test/vitis".

This chapter introduces the use of two CAN interfaces on the board for loopback testing.

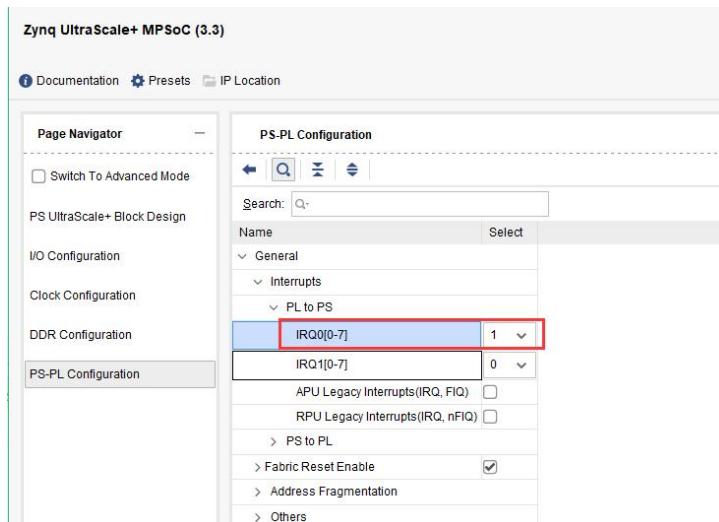
### FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

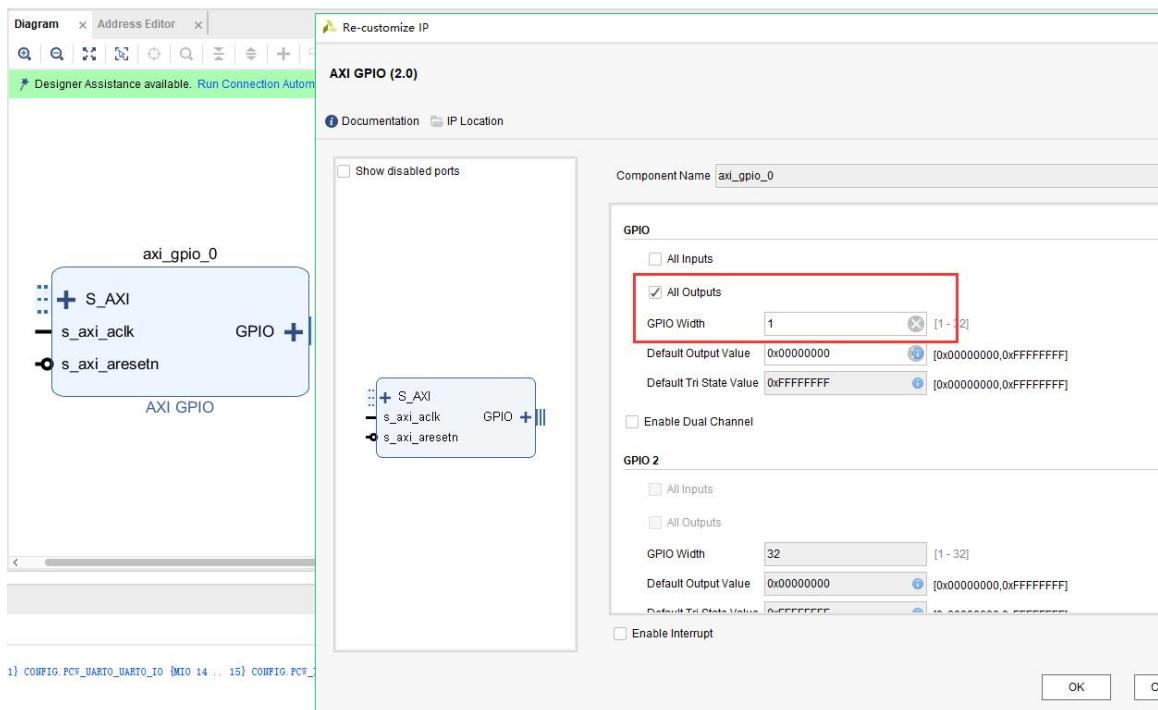
#### Part 14.1: Create a Hardware Project

The hardware project is based on "ps\_ hello", save as a project "rs485\_test".

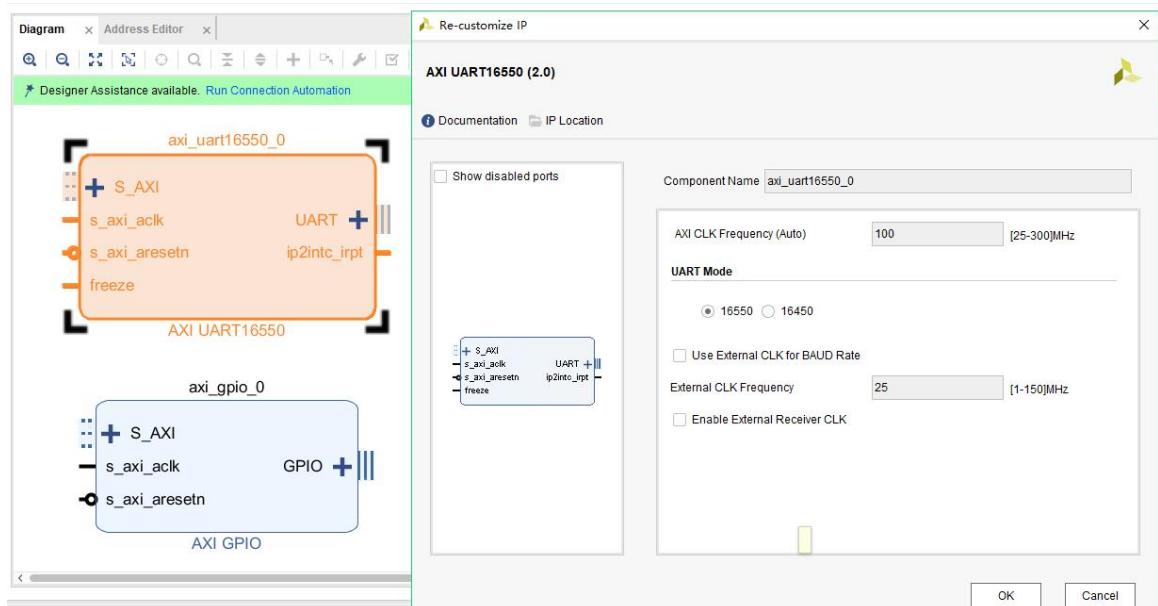
##### 1) Open the PL end interrupt



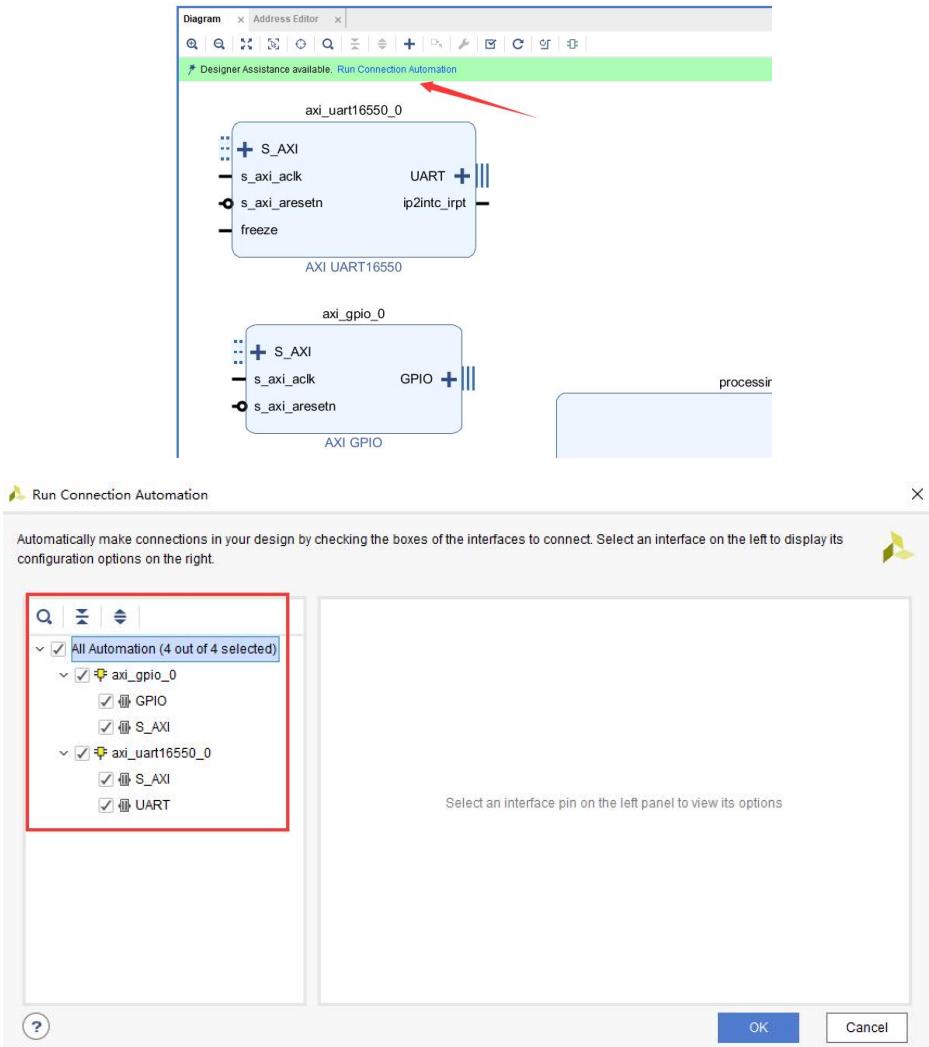
##### 2) Add an AXI GPIO module and configure it as an output with a bit width of 1, used for DE control of the First RS485



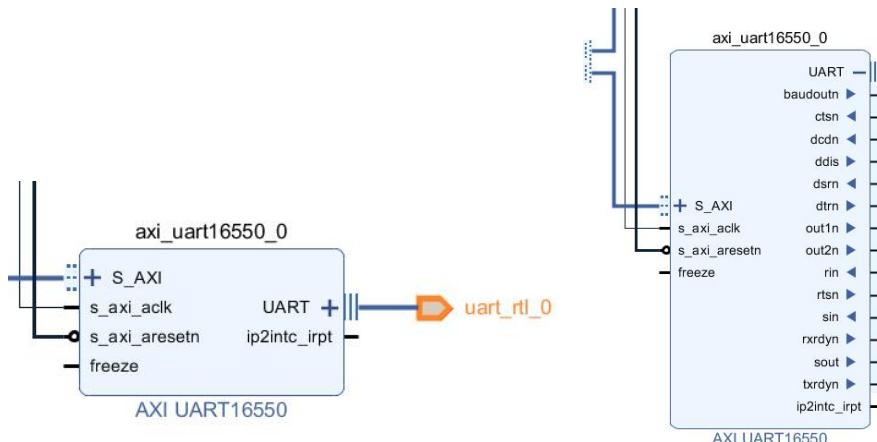
### 3) Add UART16550 module for the second RS485 data port



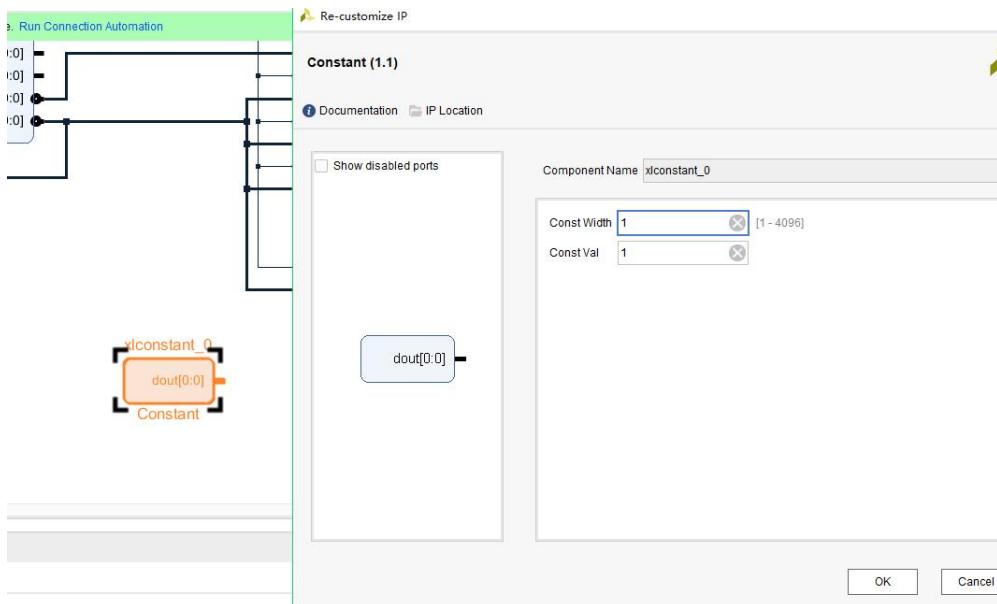
### 4) Auto Connect



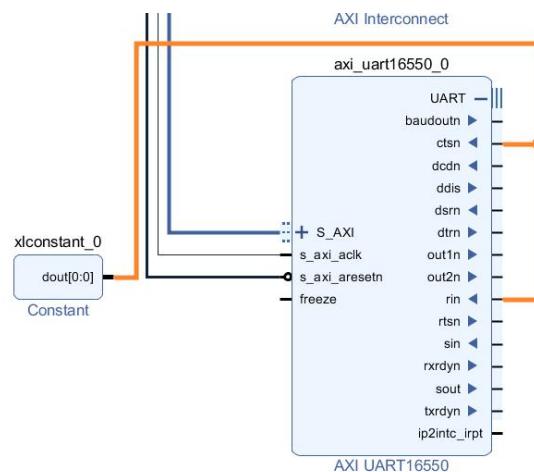
## 5) Delete the UART pin and expand the UART interface



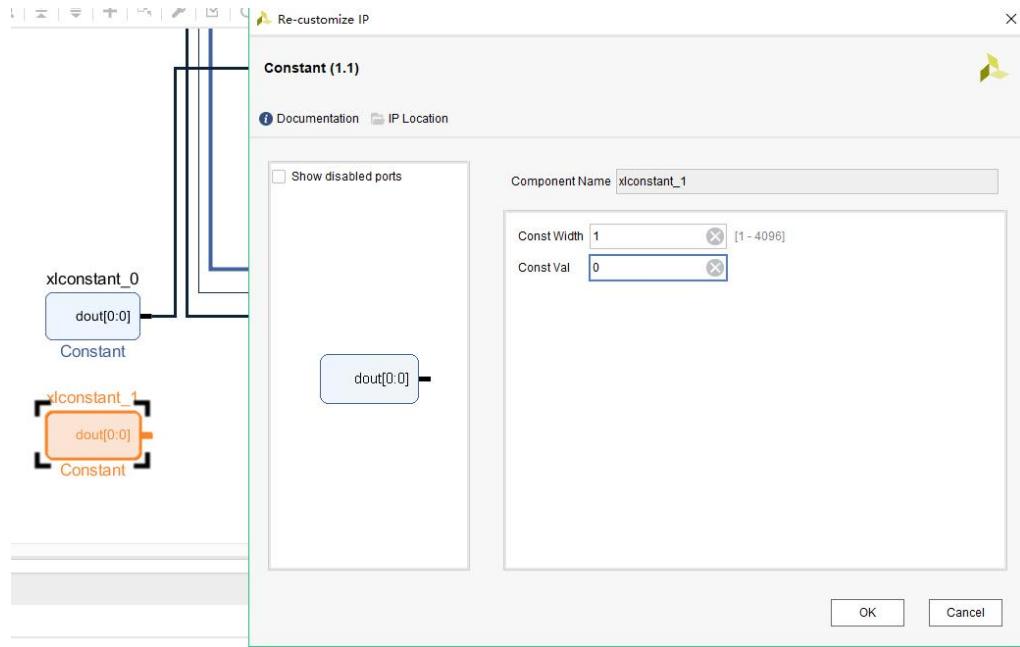
## 6) Add a constant module, and set the bit width to 1, the value is 1, which is constant



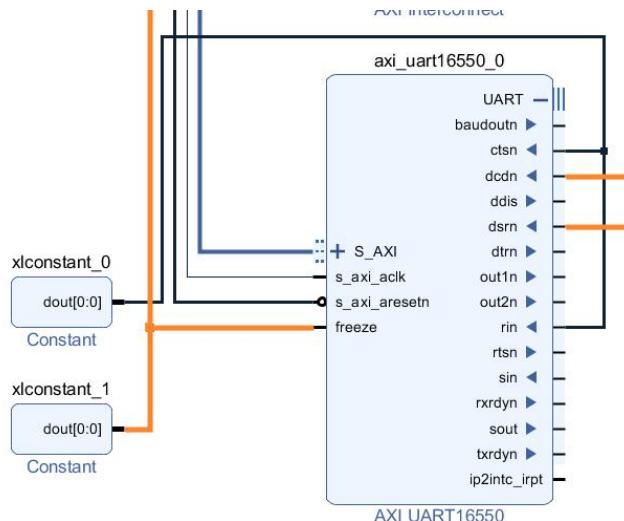
## 7) Connect ctsn, rin signal



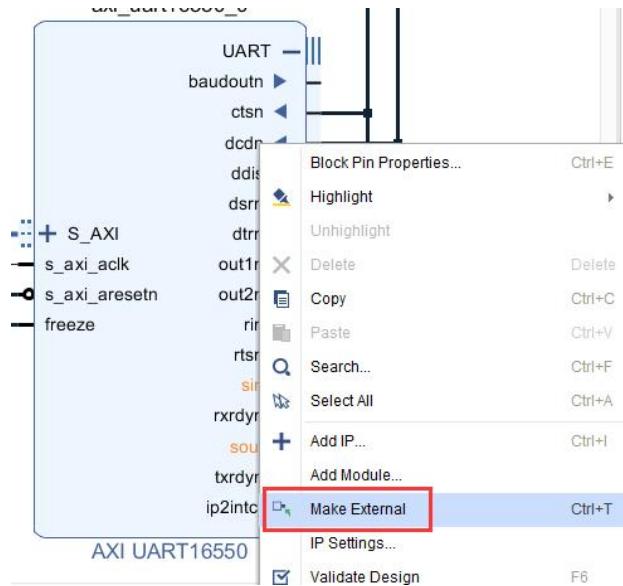
## 8) Add another constant module and set the value to 0



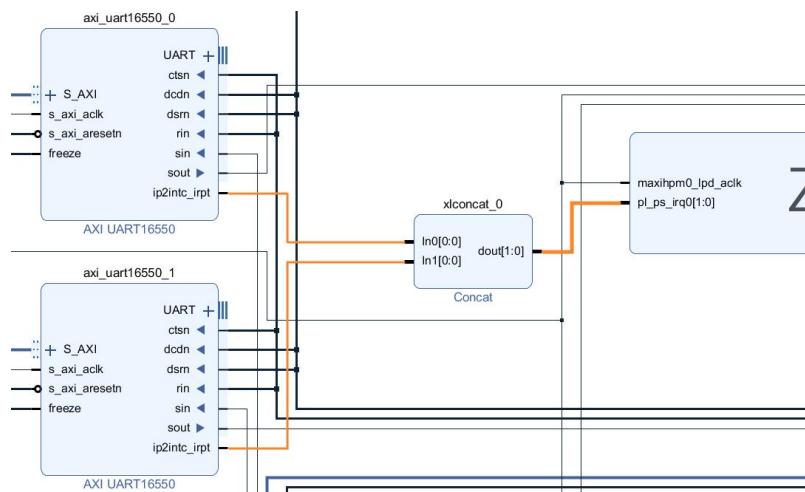
### 9) Connect "freeze", "dcdn", "dsrn" signals



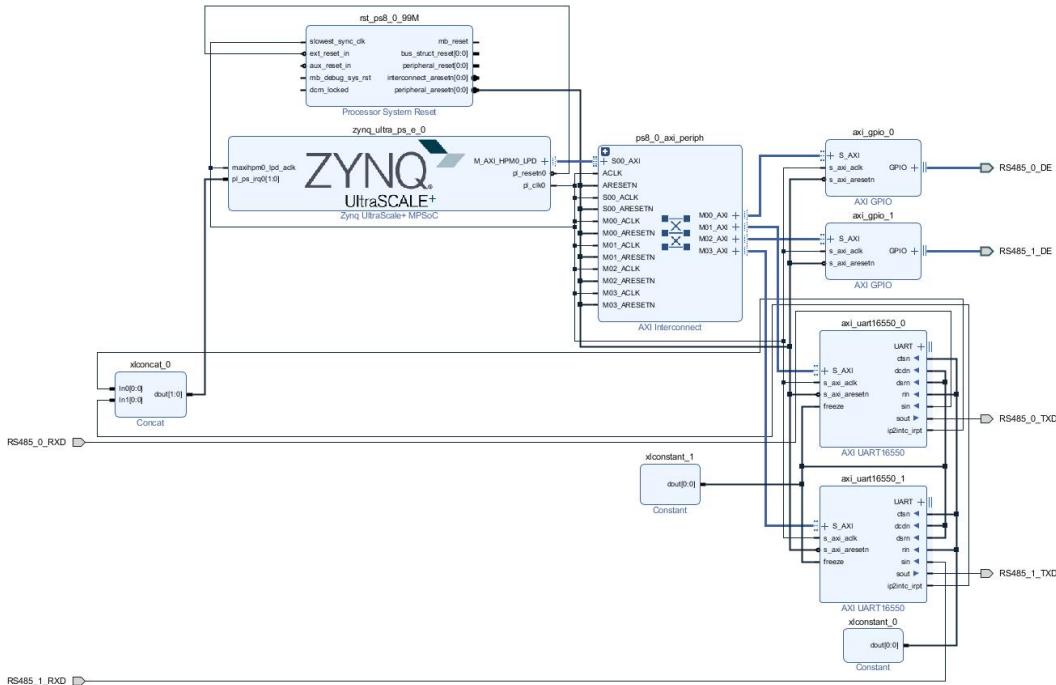
### 10) Configure the second 485 interface in the same way



- 11) Configure the second 485 interface in the same way
- 12) Add concat module, connect two interrupts to pl\_ps\_irq0



- 13) Automatically connect and modify the pin name



#### 14) Pin binding, and Generate bitstream



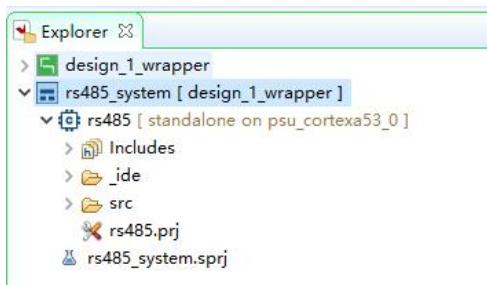
#### 15) Export hardware platform information

### Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 14.2: Vitis Program Development

- Create a new Vitis project “rs 485\_test”. This experiment is modified from the example project of the “UART” and “UART16550” modules. The program flow is: Initialize RS485 DE and UART → set RS485 0 to transmit , set RS485 1 to receive, send 16 bytes of data, compare data after receiving → reverse, set RS485 1 to transmit, set RS485 0 to receive, transmit 16 bytes of data, compare the data after receiving→the program ends



## 2) PL Side GPIO Setting

```
int PLGpioInitial(XGpio *GpioInstPtr, u16 DeviceId)
{
    int Status ;

    /* initial gpio */
    Status = XGpio_Initialize(GpioInstPtr, DeviceId) ;
    if (Status != XST_SUCCESS)
        return XST_FAILURE ;

    /* set gpio as output */
    XGpio_SetDataDirection(GpioInstPtr, 1, 0x0);

    return XST_SUCCESS ;
}
```

## 3) In the initialization of the UART, the baud rate is set to 115200, 8bit, no parity bit, 1 stop bit

```
XUartNs550Format UartNs550Format =
{
    115200,
    XUN_FORMAT_8_BITS,
    XUN_FORMAT_NO_PARITY,
    XUN_FORMAT_1_STOP_BIT
};
```

## 4) In the loopback function, first set RS485\_0 as output and RS485\_1 as input. After setting, wait for 1ms before transmitting data because of the switching delay.

```
int UartLoopback(void)
{
    unsigned int SentCount;
    unsigned int ReceivedCount = 0 ;
    u16 Index;
    u32 LoopCount = 0;

    /*
     * Initialize the send buffer bytes with a pattern and zero out
     * the receive buffer.
     */
    for (Index = 0; Index < TEST_BUFFER_SIZE; Index++) {
        SendBuffer[Index] = '0' + Index;
        RecvBuffer[Index] = 0;
    }

    /*
     * Set rs485_0 to tx
     */
    XGpio_DiscreteWrite(&rs485_0_de, 1, 1) ;
    /*
     * Set rs485_1 to rx
     */
    XGpio_DiscreteWrite(&rs485_1_de, 1, 0);

    /* wait 1ms */
    usleep(1000) ;

    memset(RecvBuffer, 0, TEST_BUFFER_SIZE) ;
    /* Block sending the buffer. */
    SentCount = XUartNs550_Send(&UartNs550_0, SendBuffer, TEST_BUFFER_SIZE);
    if (SentCount != TEST_BUFFER_SIZE) {
        return XST_FAILURE;
    }
}
```

- 5) After the test, switch the direction and set RS485\_0 as input and RS485\_1 as output, and then transmit data again

```
xil_printf("RS485_0 to RS485_1 Check Done!\r\n");

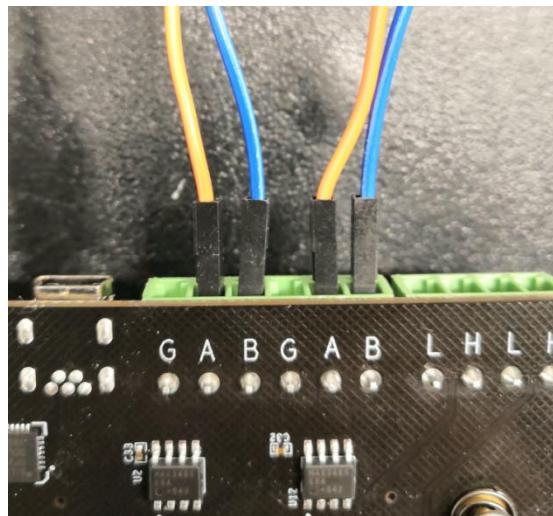
/*
 * Set rs485_0 to rx
 */
XGpio_DiscreteWrite(&rs485_0_de, 1, 0) ;
/*
 * Set rs485_1 to tx
 */
XGpio_DiscreteWrite(&rs485_1_de, 1, 1);

/* wait 1ms */
usleep(1000) ;

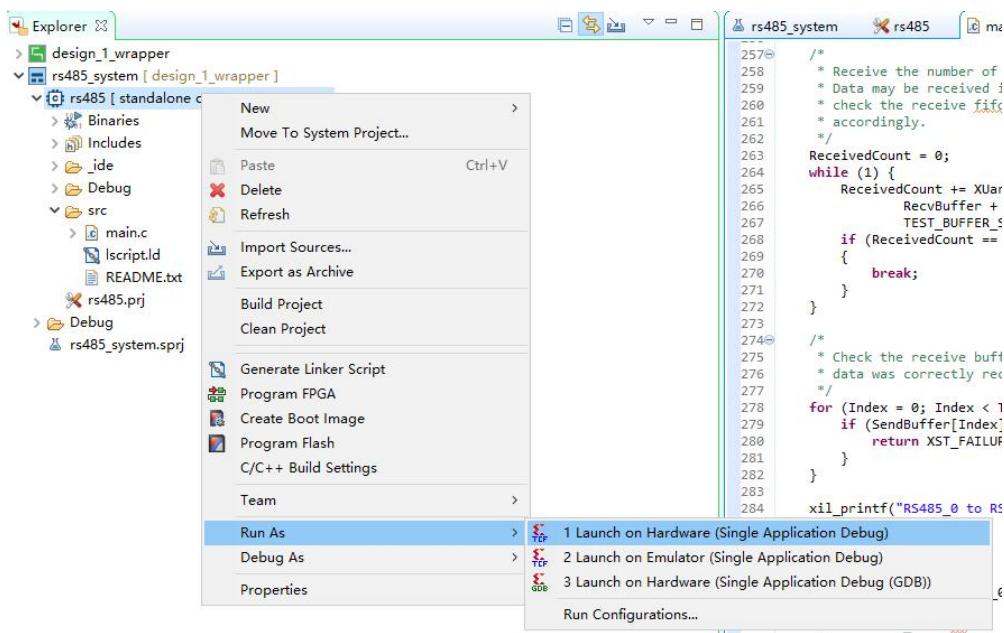
memset(RecvBuffer, 0, TEST_BUFFER_SIZE) ;
/*
 * Send the buffer thru the UART waiting till the data can be
 * sent (block), if the specified number of bytes was not sent
 * successfully, then an error occurred
 */
SentCount = XUartNs550_Send(&UartNs550_1, SendBuffer, TEST_BUFFER_SIZE);
if (SentCount != TEST_BUFFER_SIZE) {
    return XST_FAILURE;
}
```

### Part 14.3: Download Test

- 1) Connect A1 and A2 with Dupont cable, and connect B1 and B2 as follows:



## 2) Download



## 3) Serial print information

```
Start UART Loopback Test!
RS485_0 to RS485_1 Check Done!
RS485_1 to RS485_0 Check Done!
UART Loopback Done!
```

### Part 14.4: Experimental Summary

This chapter introduces the use of two RS485 channels on the board for loopback testing. You can expand on this basis for functional testing.

## Part 15: PL Side Use of Ethernet

The experimental Vivado project directory is "pl\_net/vivado".

The experiment vitis project directory is "pl\_net/vitis".

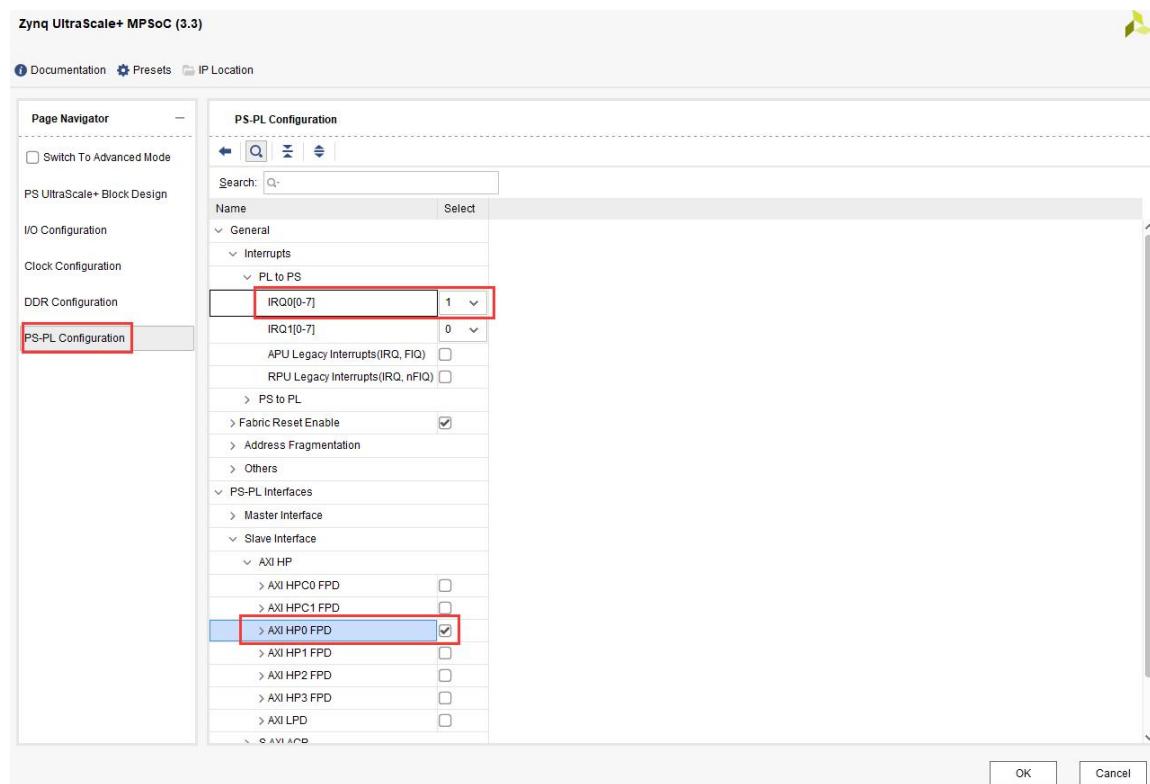
Previously introduced the PS Side use of Ethernet, this chapter introduces PL side use of Ethernet.

### FPGA Engineer Job Content

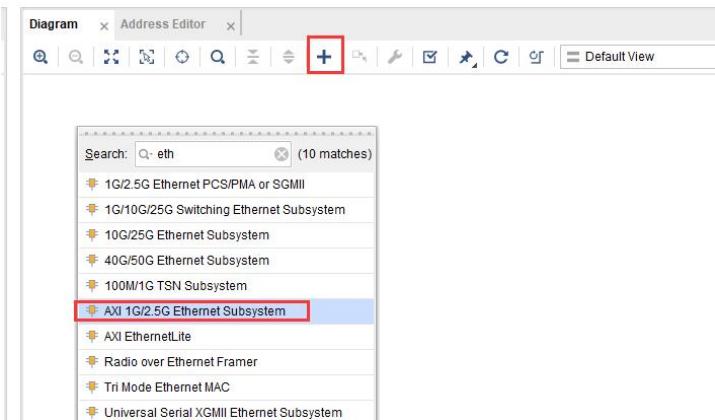
The following is the content that FPGA engineers are responsible for.

#### Part 15.1: Create a Hardware Project

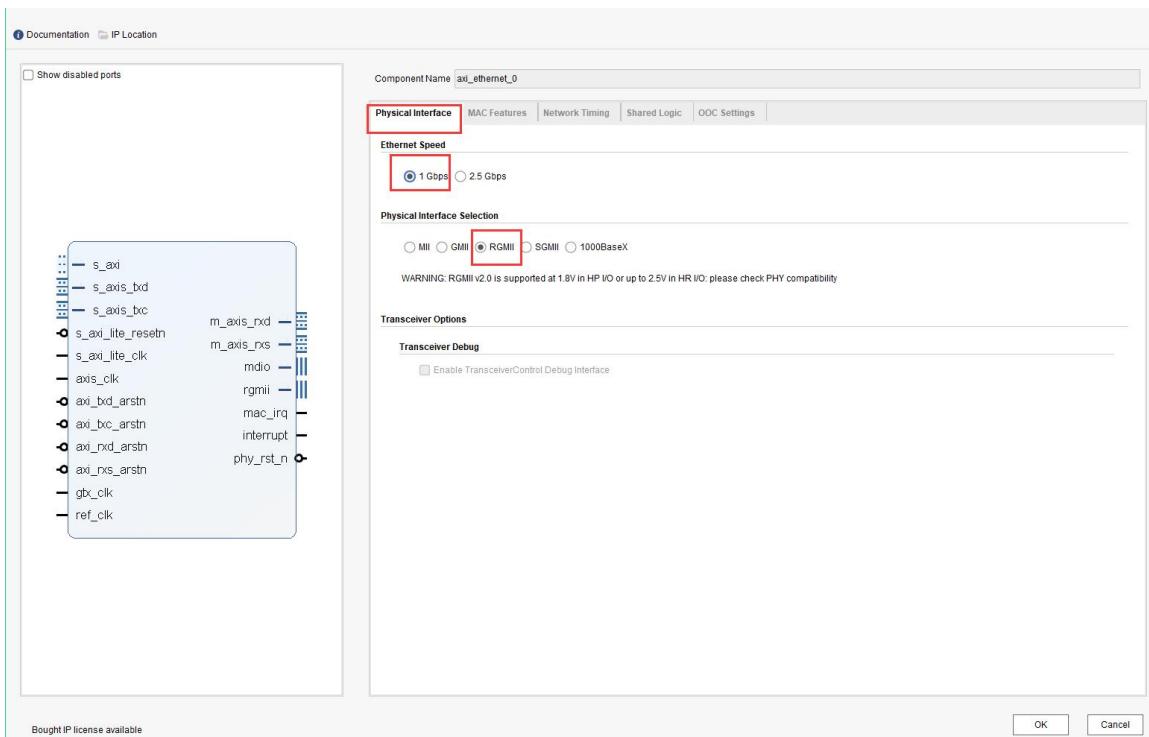
- 1) Based on "ps\_hello", configure interrupts and AXI HP0 FPD



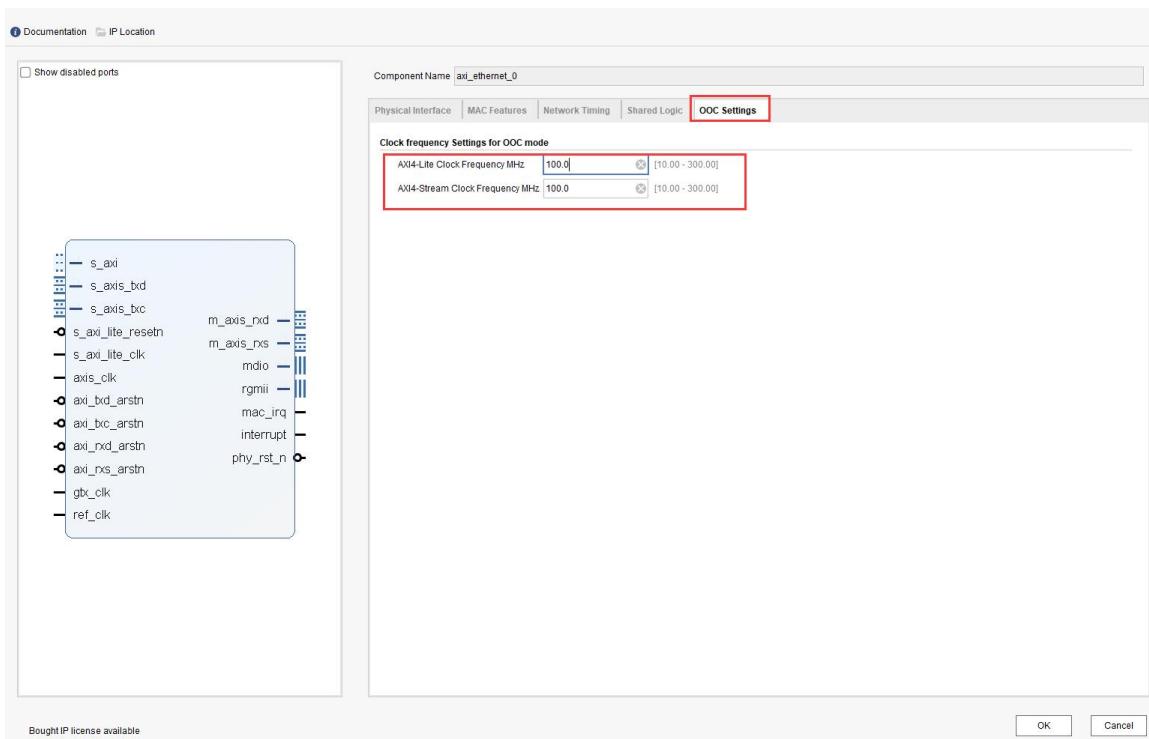
- 2) Add AXI 1G/2.5G Ethernet Subsystem module



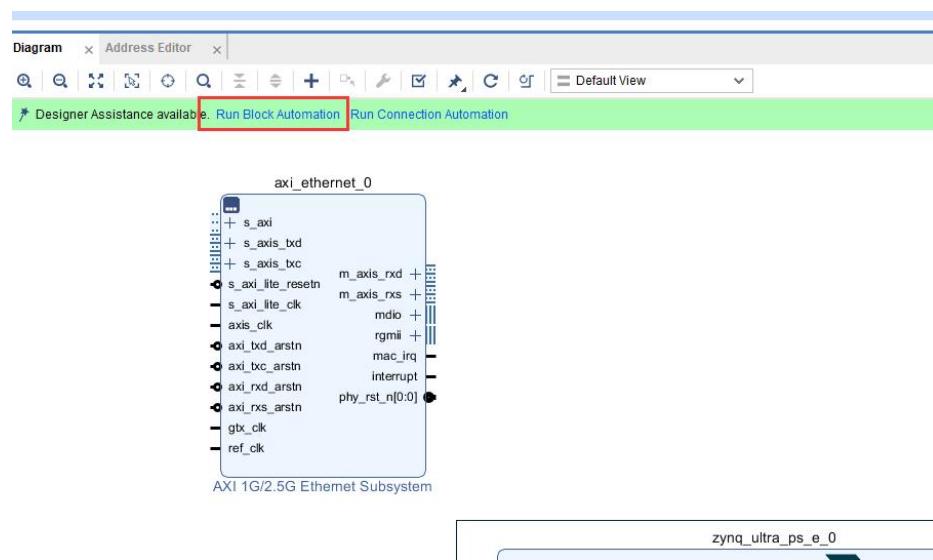
### 3) The configuration speed is 1Gbps, and the PHY interface is RGMII

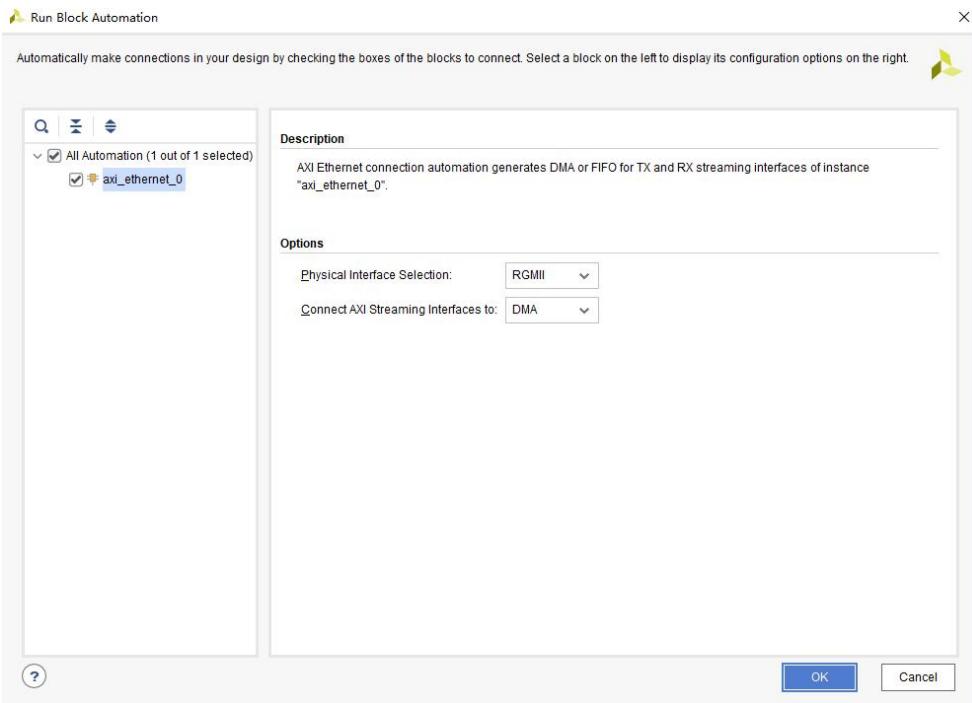


### 4) The clock defaults to 100MHz, other configurations keep the default, click OK

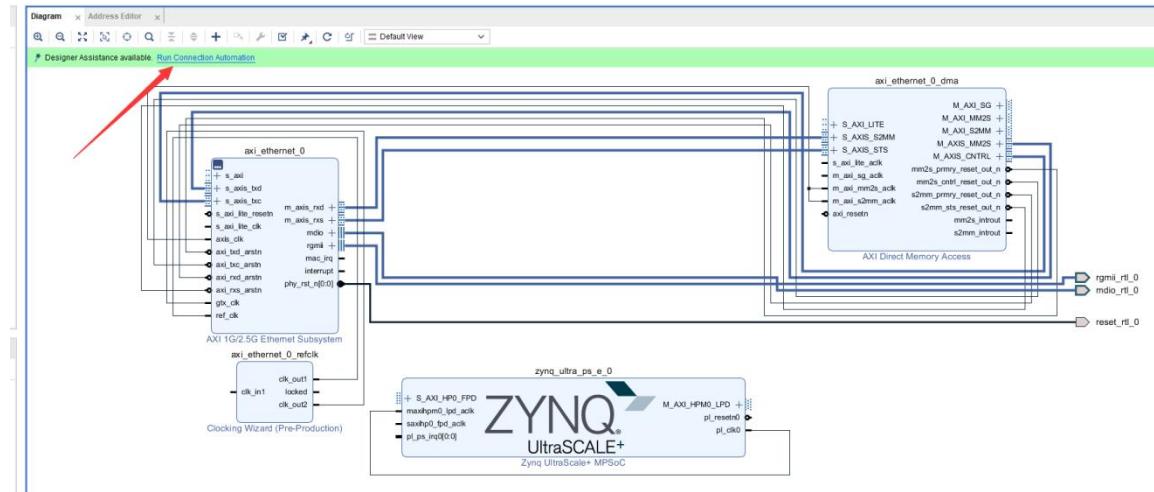


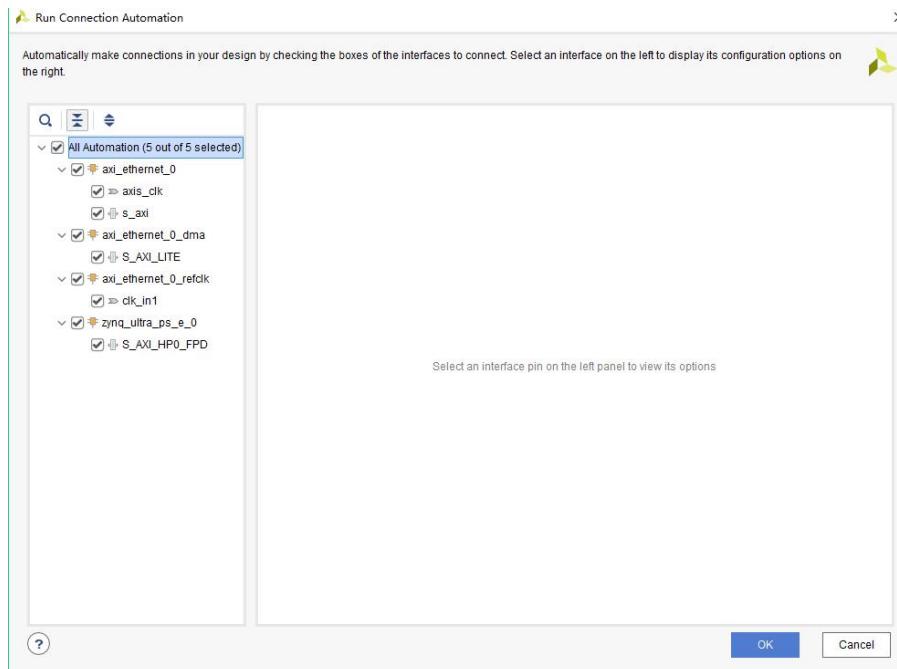
## 5) Click Run Block Automation



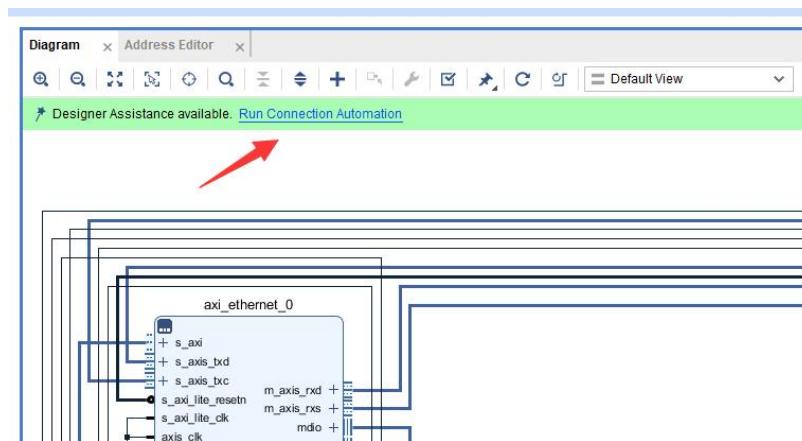


6) Click Run Connectino Automation to automatically connect

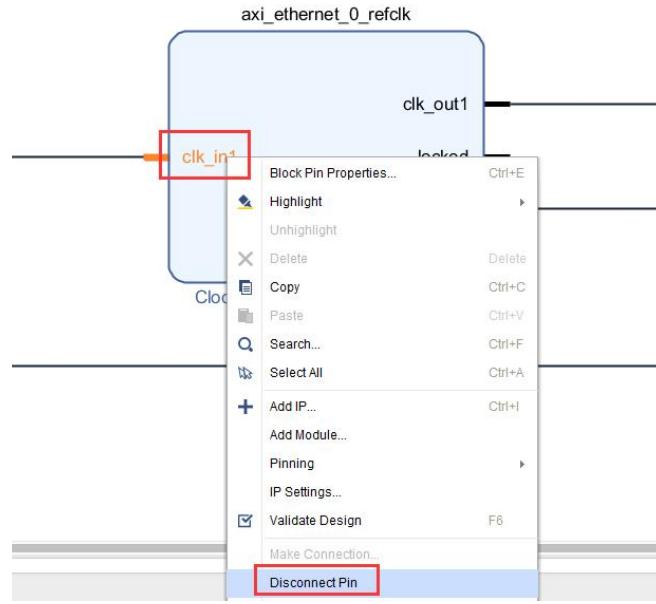




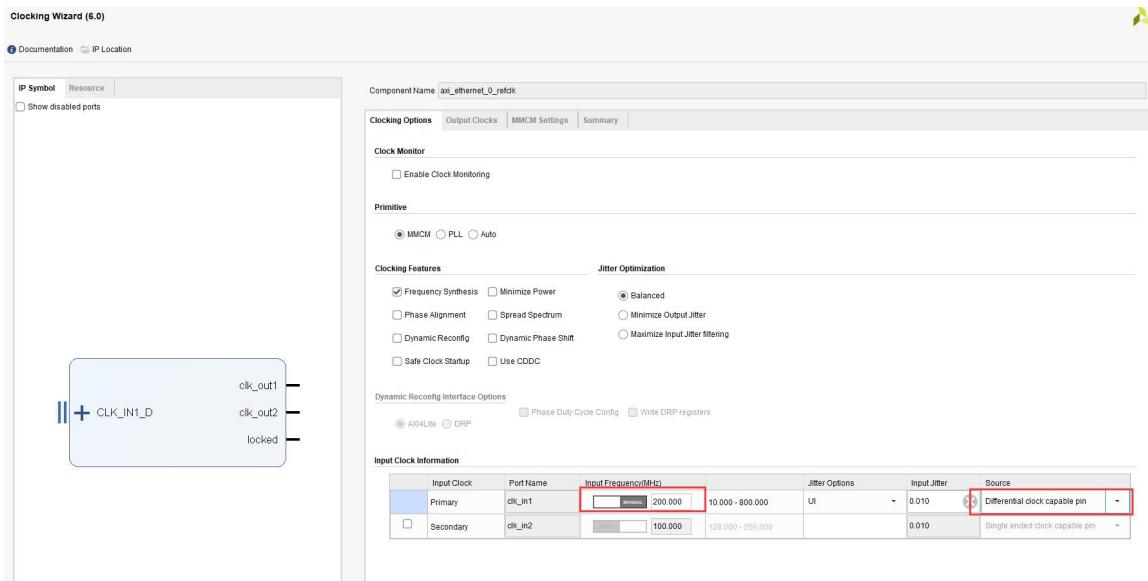
## 7) Continue to connect automatically



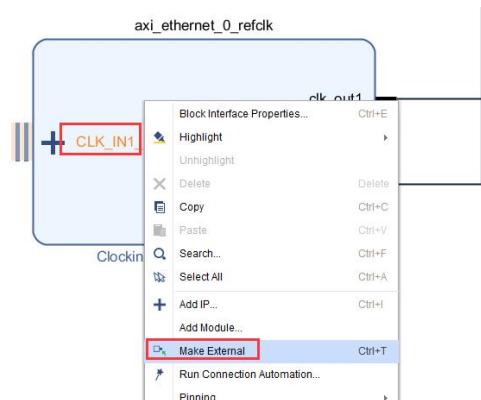
## 8) Select the input clock of PII, Disconnect Pin



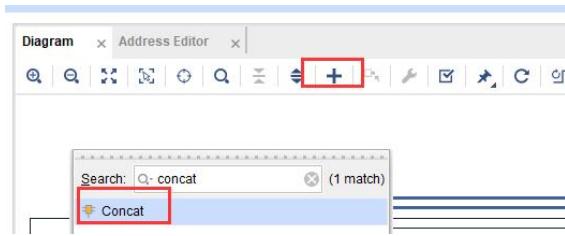
Click to open the PLL, configure the input clock to 200MHz, and set it as a differential input



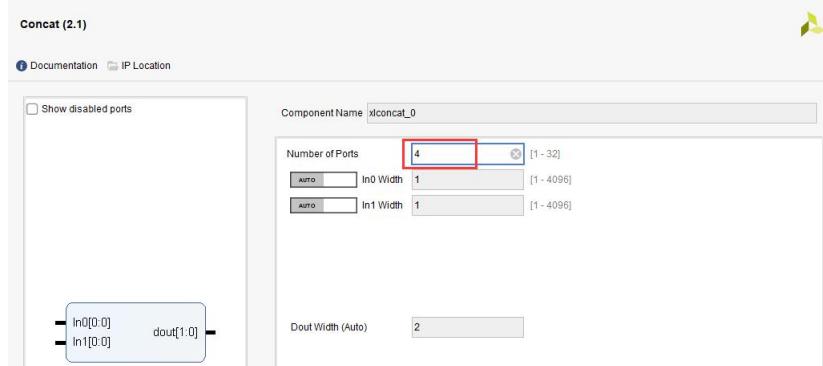
## 9) Export pin



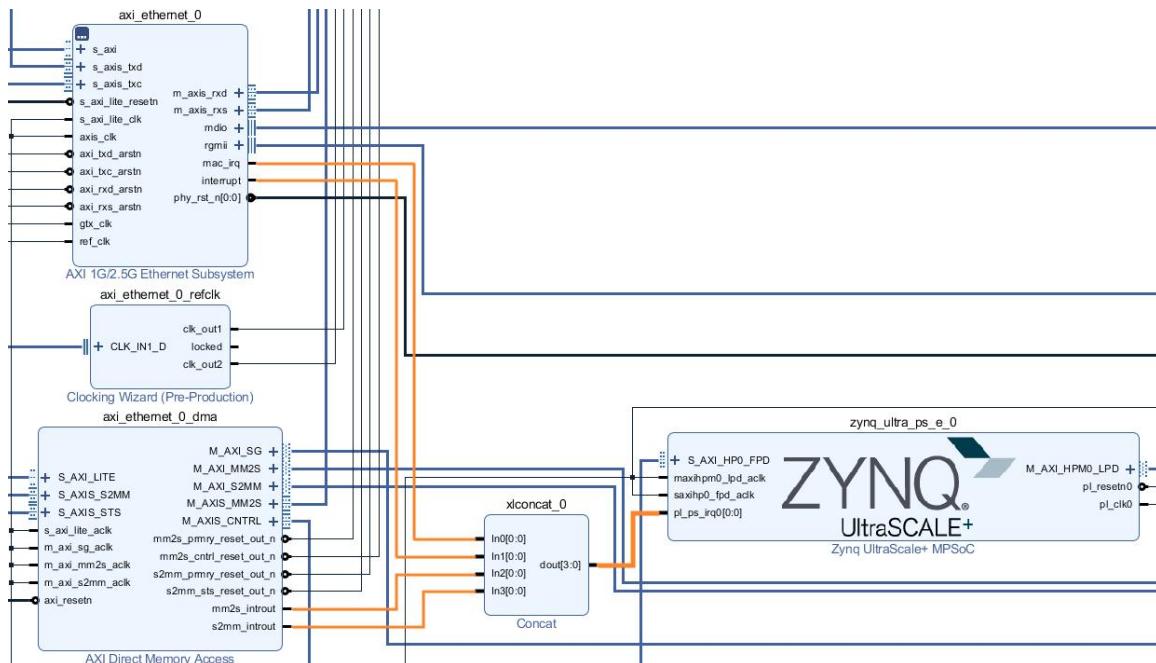
## 10)Add concat module



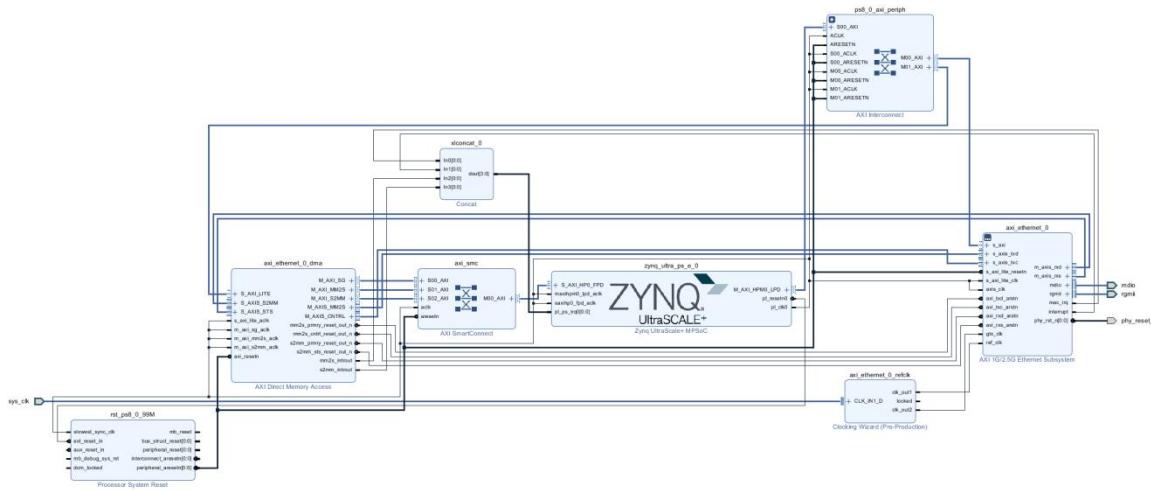
Configured to 4



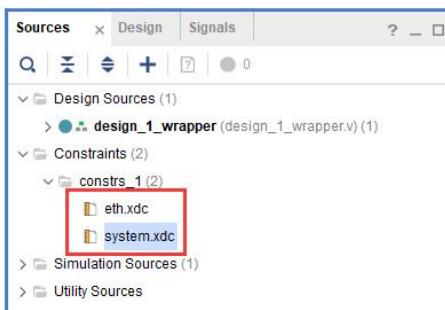
## 11)Connection interruption signal



## 12)Modify the pin name



13)Bind pins, generate bitstream, and export hardware information



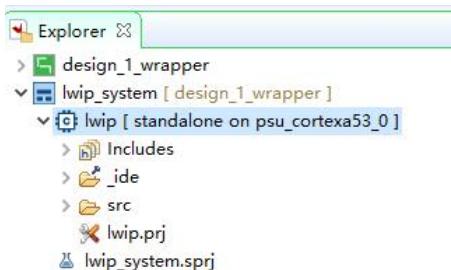
# Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

## Part 15.2: Vitis Program Development

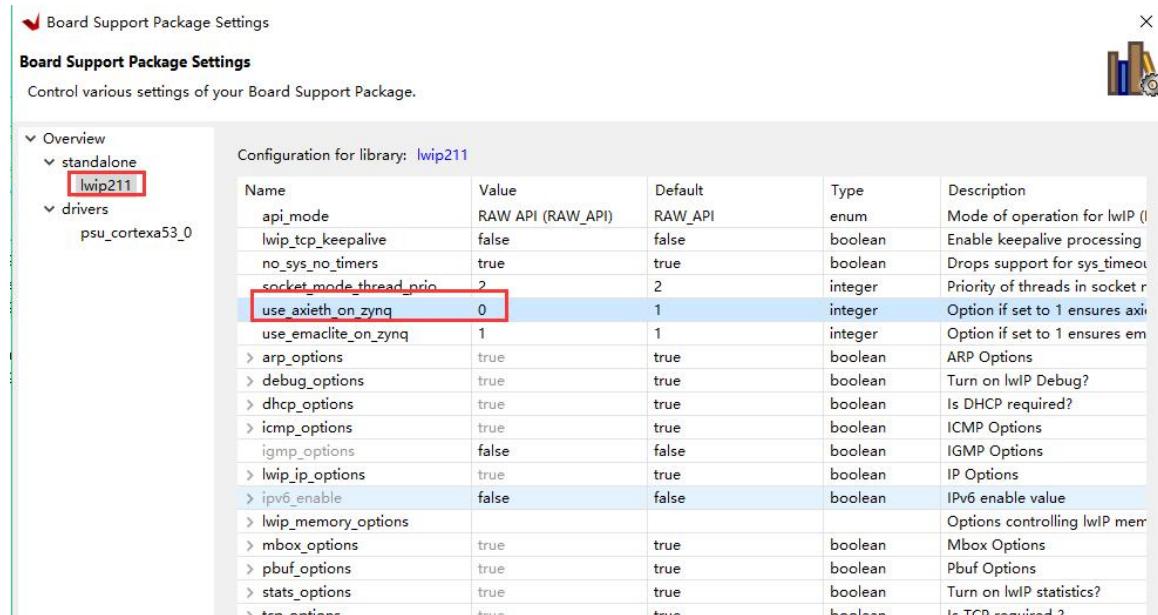
## Part 15.2.1: PL Side Ethernet test

The Vitis project creation method is the same as that of the PS side. The process will not be repeated. The `lwip` library has been modified before. The test method has been introduced earlier.

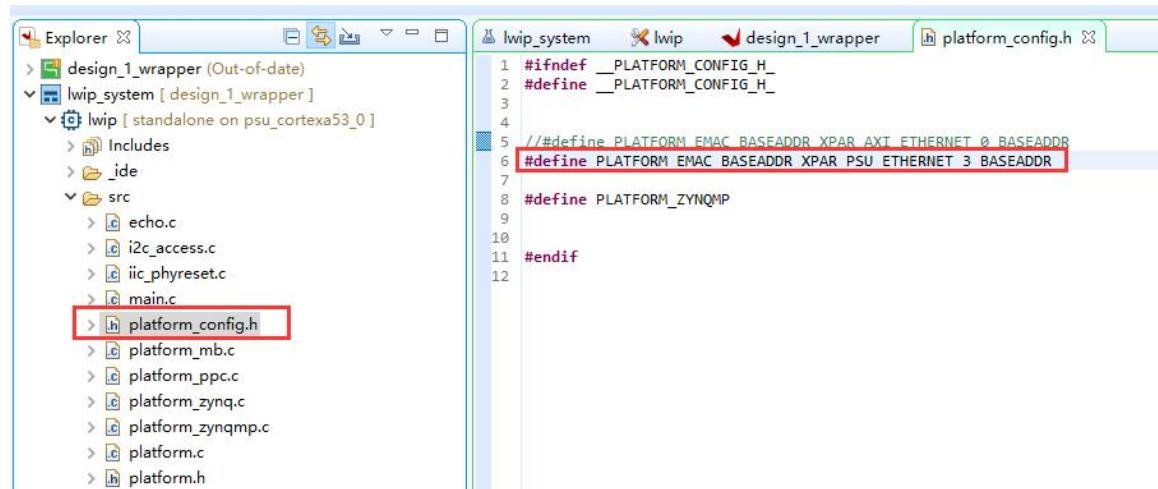


### Part 15.2.2: PS Side Ethernet Test

If you want to test the PS side Ethernet in this project, you need to modify the BSP settings, the default value of `use_axieth_onzynq` is 1, which is the AXI Ethernet on the PL side, change the value to 0, and click OK



Modify `platform_config.h` and recompile



## Part 16: Custom IP experiment

The experimental Vivado project directory is "custom\_pwm\_ip /vivado".

The experimental vitis project directory is "custom\_pwm\_ip /vitis".

Xilinx officially provides a lot of IP cores. You can view these IP cores in Vivado's IP Catalog. Users can build their own systems. It is impossible to use only Xilinx's official free IP core. In many cases, you need to create your own user IP. There are many benefits to creating your own IP core, such as system design customization; design reuse, you can add a license to the IP core, and provide it to others for payment; simplify system design and shorten design time. To design an IP core with the ZYNQ system, the most common one is to connect the PS to the IP core of the PL part using the AXI bus. This lab will show you how to build an AXI bus type IP core in Vivado. This IP core is used to generate a PWM. Use this to control the LED on the development board to make a breathing light effect.

### FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

#### Part 16.1: PWM Introduction

We often use PWM to control LEDs, buzzers, etc., by adjusting the duty cycle of the pulses to adjust the brightness of the LEDs. One of the pwm modules we have used in other development boards is as follows:

```
////////////////////////////////////////////////////////////////
//                                                     //
//                                                     //
// Author: meisq                                     //
//         msq@qq.com                                 //
//         ALINX(shanghai) Technology Co.,Ltd        //
//         heijin                                      //
// WEB: http://www.alinx.cn/                          //
// BBS: http://www.heijin.org/                        //
//                                                     //
//                                                     //
// Copyright (c) 2017,ALINX(shanghai) Technology Co.,Ltd   //
// All rights reserved                                //
//                                                     //
// This source file may be used and distributed without restriction provided   //
// that this copyright statement is not removed from the file and that any      //
// derivative work contains the original copyright notice and the associated    //
// disclaimer.                                         //
//                                                     //
////////////////////////////////////////////////////////////////

//=====================================================================
// Description: pwm model
// pwm out period = frequency(pwm_out) * (2 ** N) / frequency(clk);
// 
//=====================================================================
// Revision History:
// Date      By      Revision      Change Description
// -----
// 2017/5/3   meisq    1.0      Original
//*****
`timescale 1ns / 1ps
module ax_pwm
#(
  parameter N = 32 //pwm bit width
)
(
  input      clk,
  input      rst,
  input[N - 1:0]period,
  input[N - 1:0]duty,
  output     pwm_out
);

reg[N - 1:0] period_r;
reg[N - 1:0] duty_r;
reg[N - 1:0] period_cnt;
reg pwm_r;
assign pwm_out = pwm_r;
always@(posedge clk or posedge rst)
begin
  if(rst==1)
  begin
    period_r <= { N {1'b0} };
    duty_r <= { N {1'b0} };
  end
  else
  begin
    period_r <= period;
    duty_r  <= duty;
  end
end
end
```

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

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

It can be seen that this PWM module requires two parameters "period" and "duty" to control the frequency and duty cycle. We need to design some registers to control these parameters. Here we need to use the AXI bus, and the PS reads and writes the registers through the AXI bus.

$$\text{PWM Frequency} = \frac{\text{period}}{2^N} \times \text{clk Frequency} (\text{The unit is "Hz"})$$

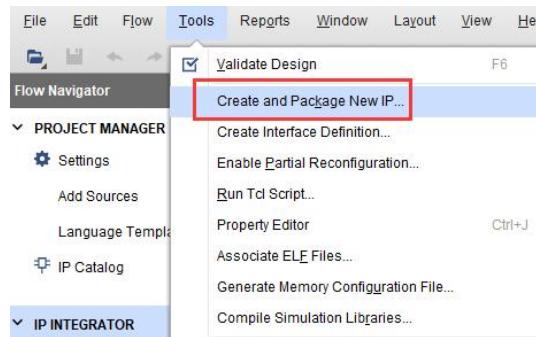
$$\text{PWM Duty Cycle} = 1 - \frac{\text{duty}+1}{2^N}$$

## Part 16.2: Building a Vivado project

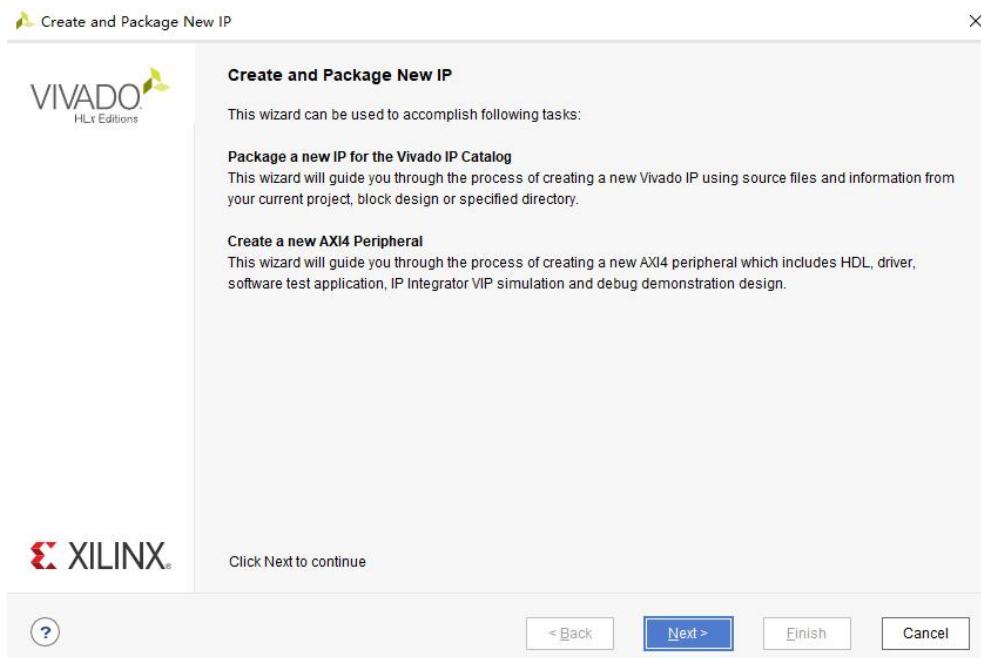
Save the project "ps\_hello" as a project named "custom\_pwm\_ip"

### Part 16.2.1: Create a custom IP

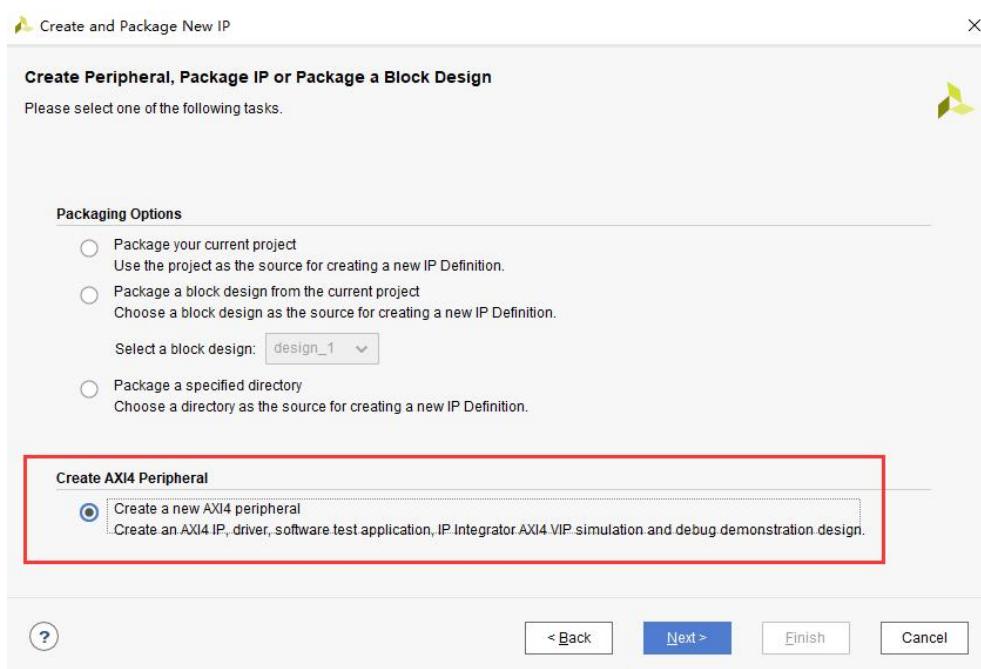
- 1) Click on the menu "Tools->Create and Package IP..."



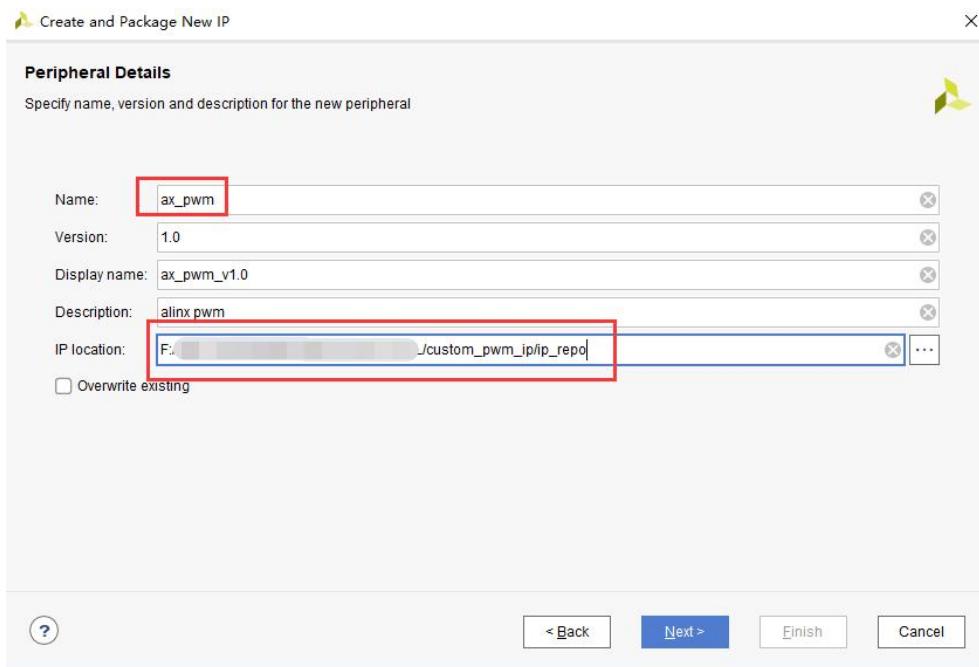
## 2) Select "Next"



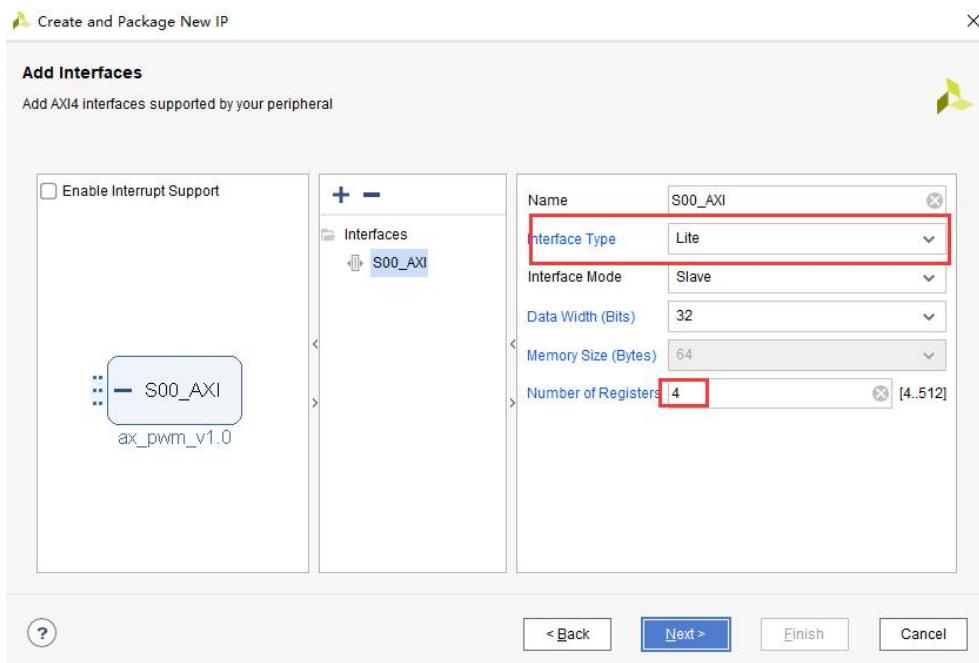
## 3) Choose to create a new AXI4 device



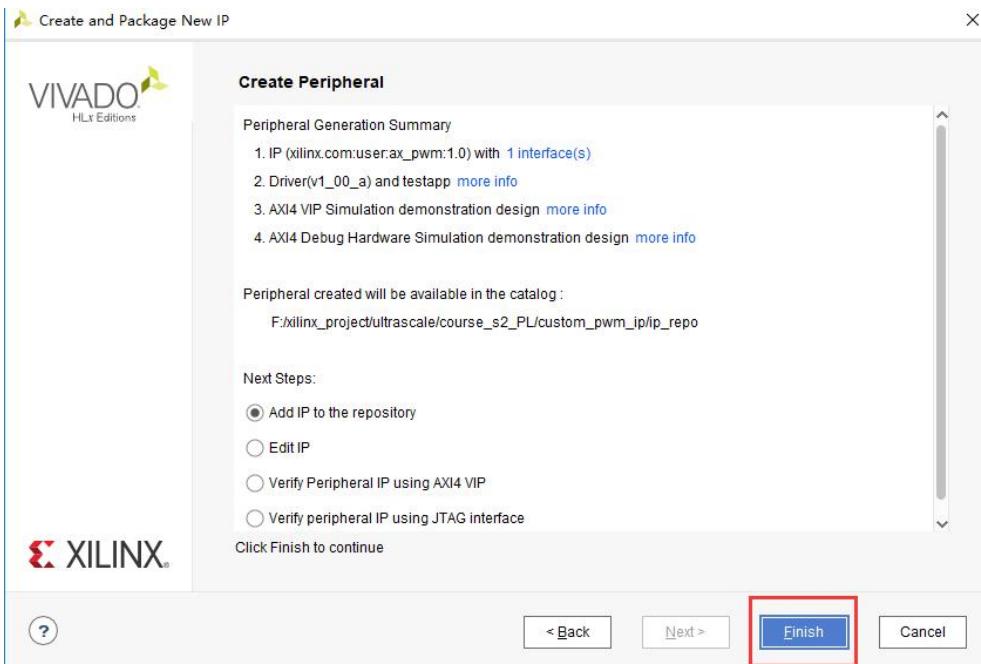
- 4) Fill in the name "ax\_pwm", describe "alinx pwm", and then select a suitable location for IP



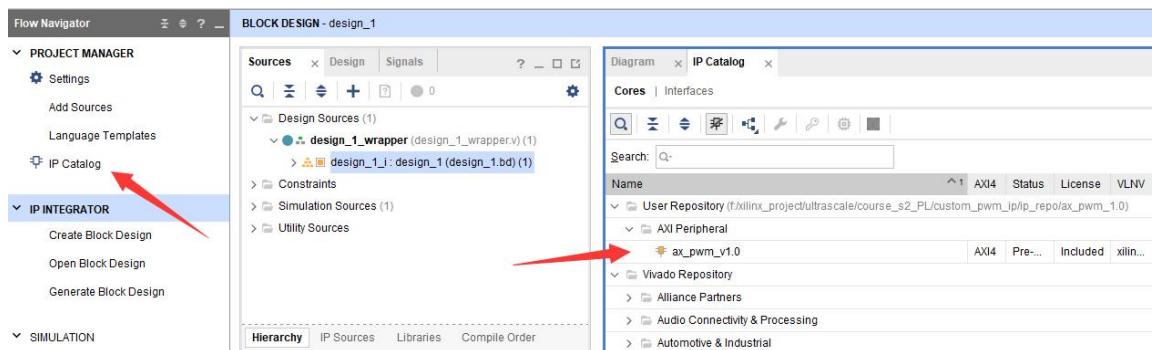
- 5) The following parameters can specify the interface type, the number of registers, etc., do not need to be modified here, use the AXI Lite Slave interface, 4 registers.



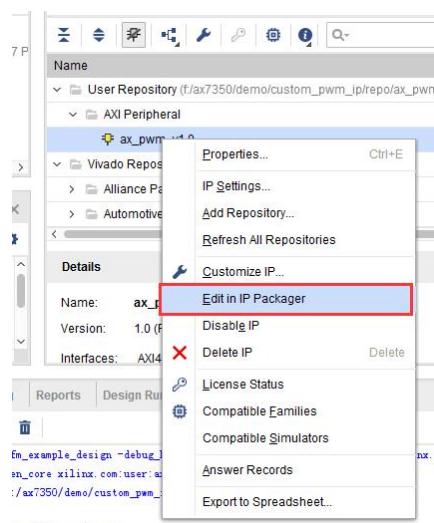
- 6) Click "Finish" to complete the creation of the IP.



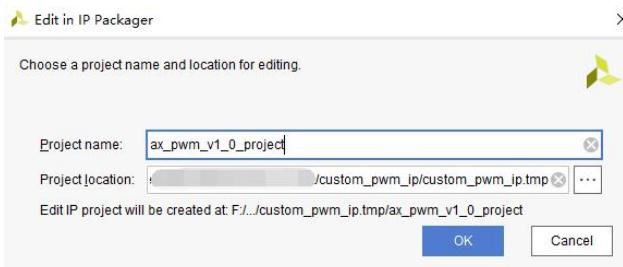
7) You can see the IP just created in "IP Catalog"



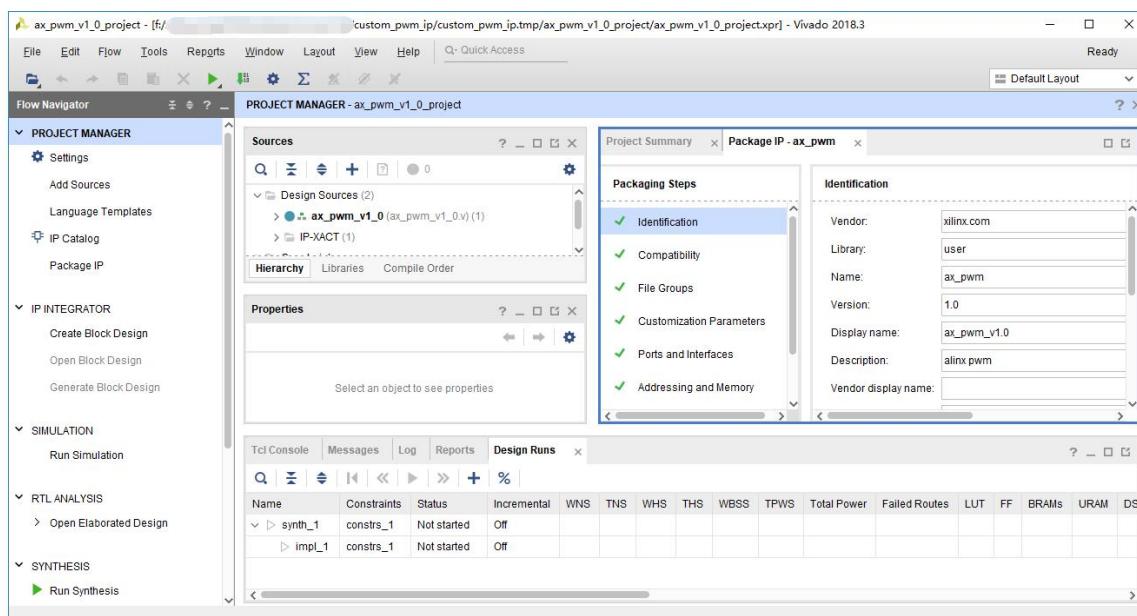
8) At this time, the IP only has a simple register read and write function. We need to modify the IP, select the IP, and right click "Edit in IP Package".



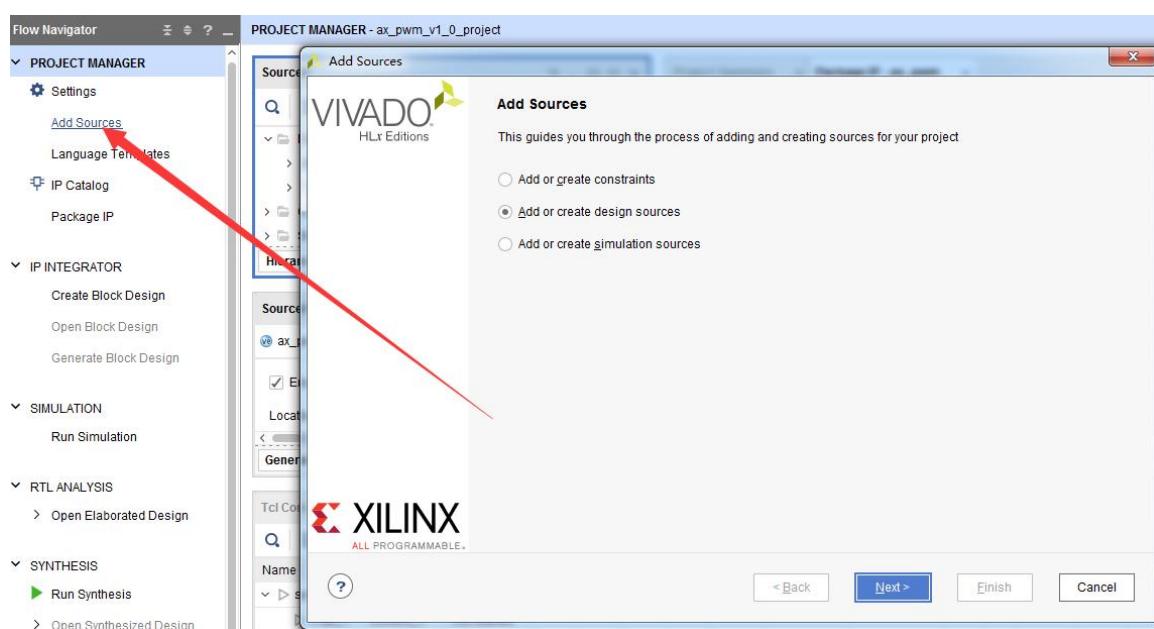
- 9) This is a pop-up dialog box where you can fill in the project name and path. By default, click "OK"



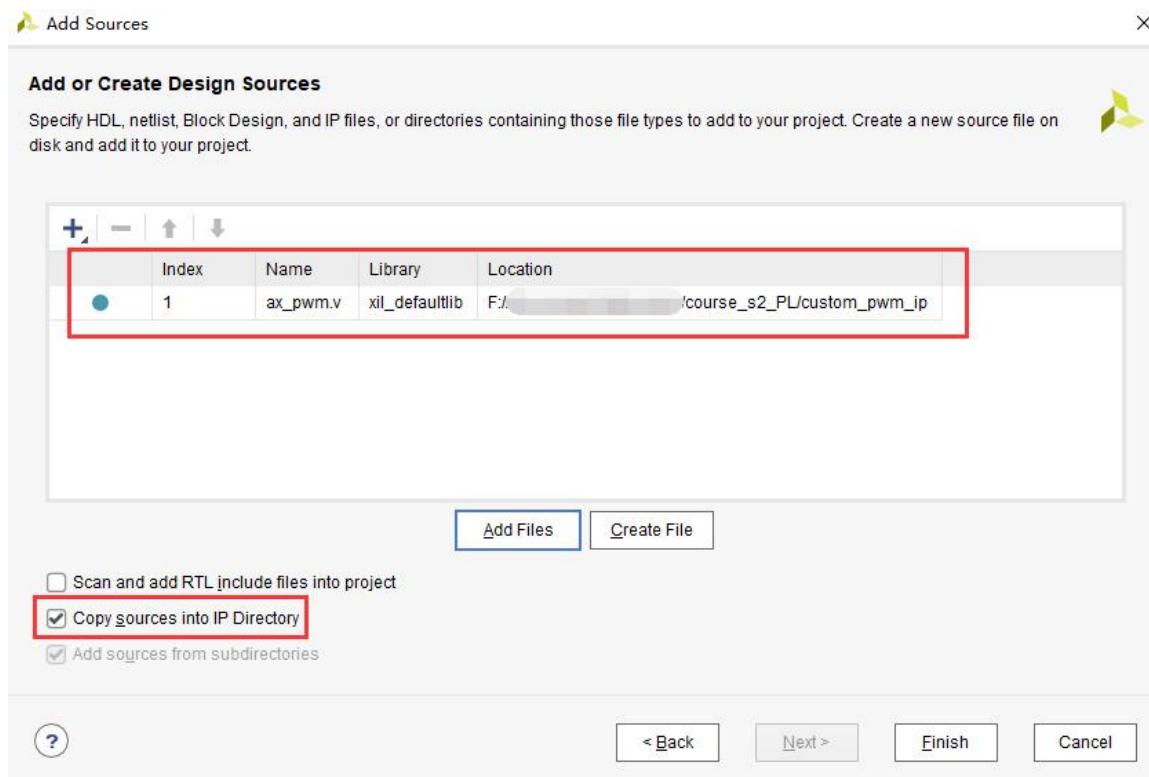
- 10)Vivado opens a new project



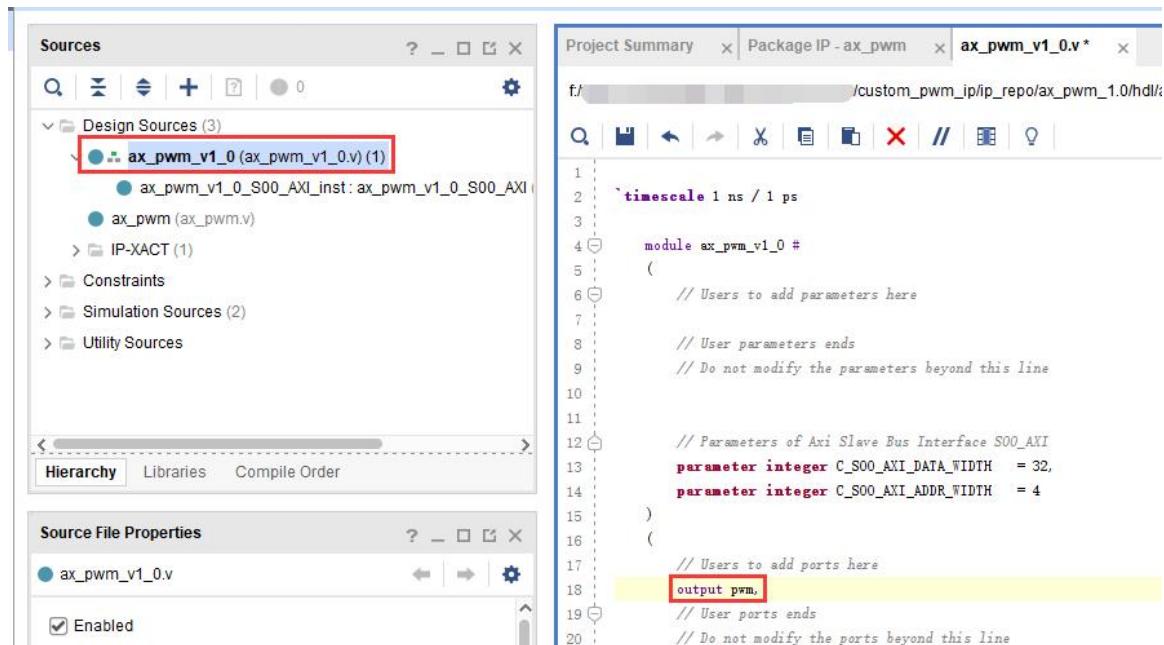
- 11)Add the core code of the PWM function



## 12) Select copy code to IP directory when adding code



## 13) Modify "ax\_pwm\_v1\_0.v" to add a pwm output port



## 14) Modify "ax\_pwm\_v1\_0.v" to add the pwm port routine to the routine "ax\_pwm\_V1\_0\_S00\_AXI"

```

Project Summary x Package IP - ax_pwm x ax_pwm_v1_0.v* x
f:/xilinx_project/ultrascale/course_s2_PL/custom_pwm_ip/ip_repo/ax_pwm_1.0/hdl/ax_pwm_v1_0.v

43     output wire s00_axi_rvalid,
44     input wire s00_axi_rready
45   );
46   // Instantiation of Axi Bus Interface S00_AXI
47   ax_pwm_v1_0_S00_AXI #(
48     .C_S_AXI_DATA_WIDTH(C_S_AXI_DATA_WIDTH),
49     .C_S_AXI_ADDR_WIDTH(C_S_AXI_ADDR_WIDTH)
50   ) ax_pwm_v1_0_S00_AXI_inst (
51     .pwm(pwm),
52     .S_AXI_ACLK(s00_axi_aclk),
53     .S_AXI_ARSTEN(s00_axi_arstn),
54     .S_AXI_AWADDR(s00_axi_awaddr),
55     .S_AXI_AWPROT(s00_axi_awprot),
56     .S_AXI_AWVALID(s00_axi_awvalid),
57     .S_AXI_AWREADY(s00_axi_awready),
58     .S_AXI_WDATA(s00_axi_wdata),
59     .S_AXI_WSTRB(s00_axi_wstrb),

```

15) Modify the "ax\_pwm\_v1\_0\_s00\_AXI.v" file and add the pwm port, which is the core code for implementing AXI4 Lite Slave.

```

Project Summary x Package IP - ax_pwm x ax_pwm_v1_0.v x ax_pwm_v1_0_S00_AXI.v* x
f:/xilinx_project/ultrascale/course_s2_PL/custom_pwm_ip/ip_repo/ax_pwm_1.0/hdl/ax_pwm_v1_0_S00_AXI.v

7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

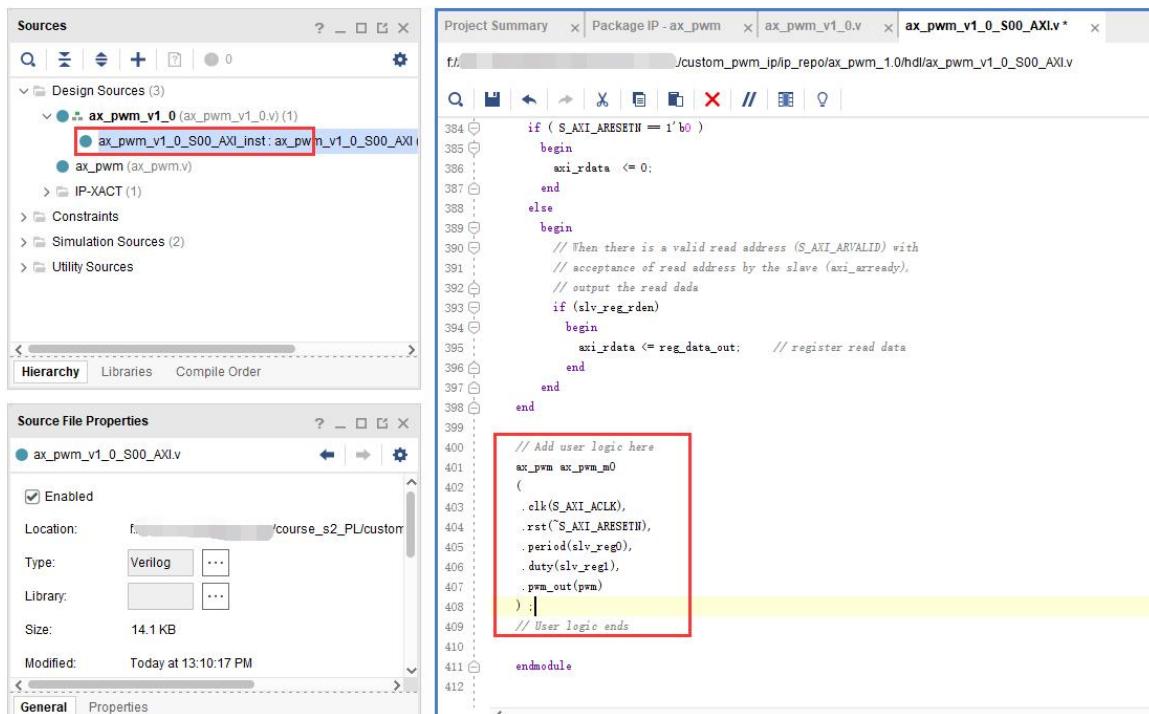
// User parameters ends
// Do not modify the parameters beyond this line

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

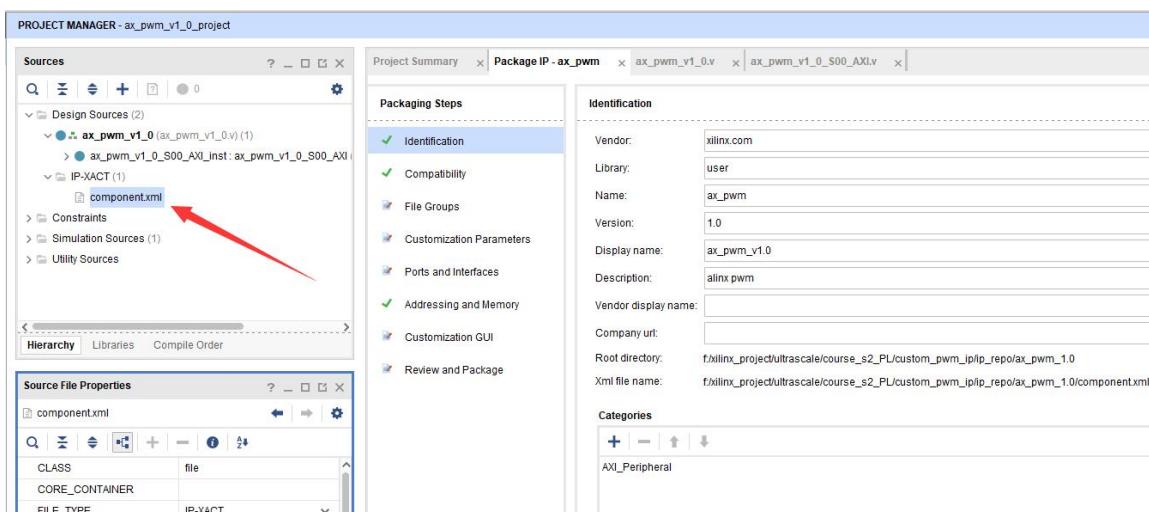
(
  // Users to add ports here
  output pwm;
  // User ports ends
  // Do not modify the ports beyond this line
  // Global Clock Signal

```

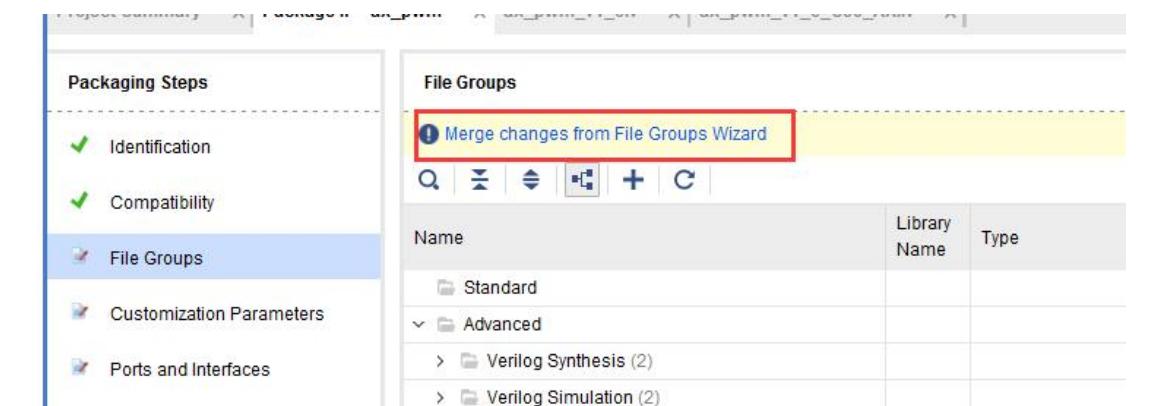
16) Modify the "ax\_pwm\_v1\_0\_s00\_AXI.v" file, the routine pwm core function code, and use the registers slv\_reg0 and slv\_reg1 for parameter control of the pwm module.



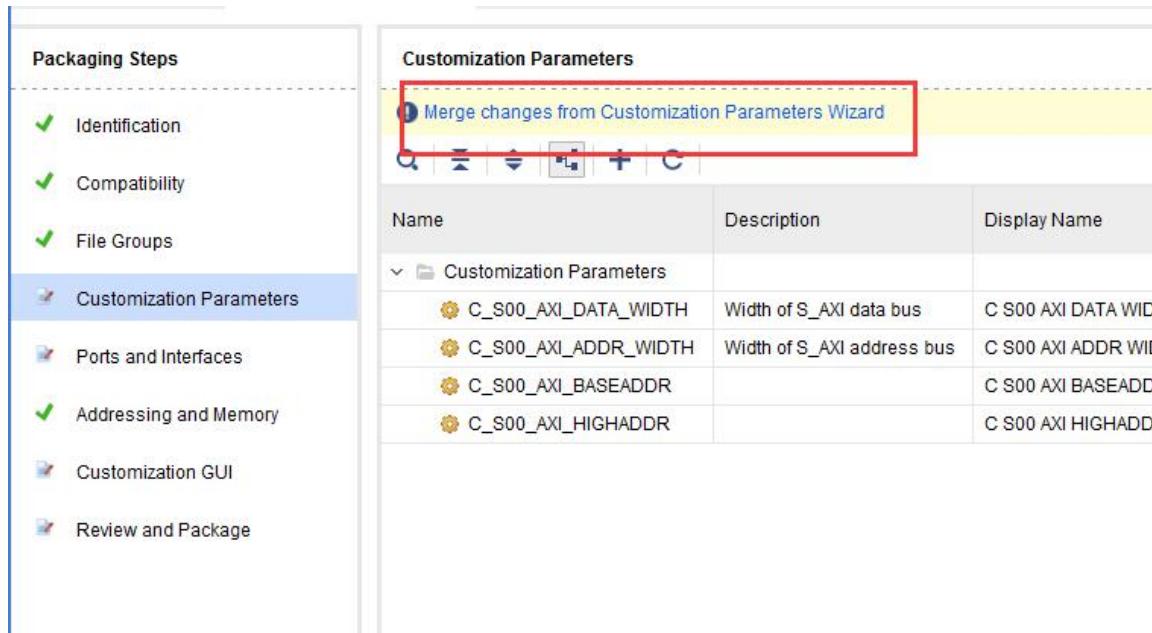
## 17) Double-click the "component.xml" file



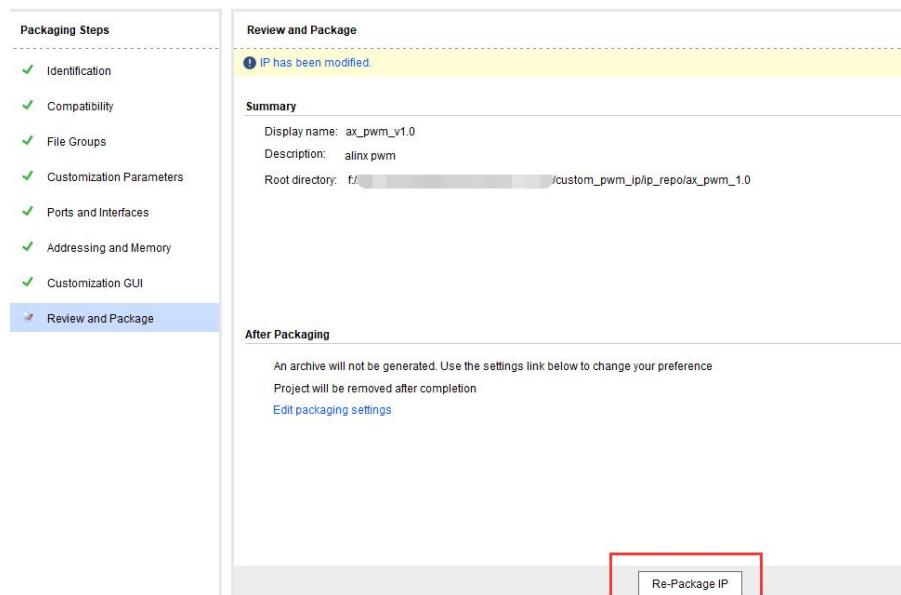
## 18) Click on "Merge changes from File Groups Wizard" in the "File Groups" option.



19) Click on "Merge changes from Customization Parameters Wizard" in the "Customization Parameters" option.

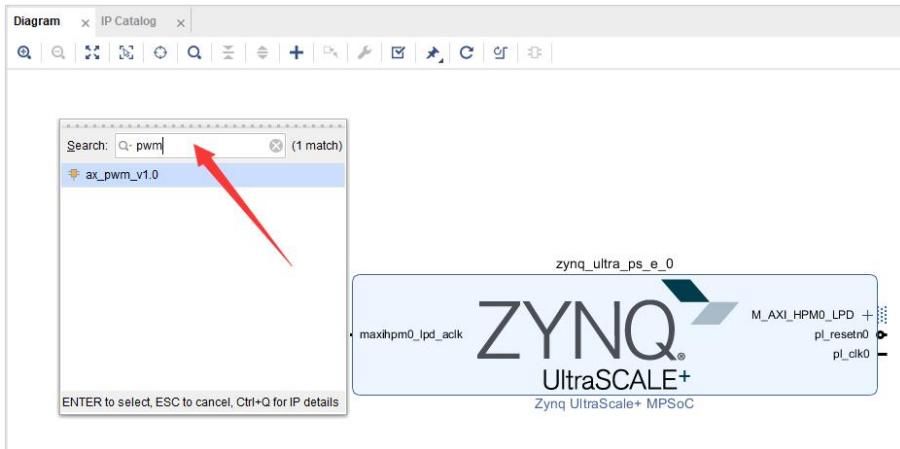


20) Click "Re-Package IP" to complete the IP modification.

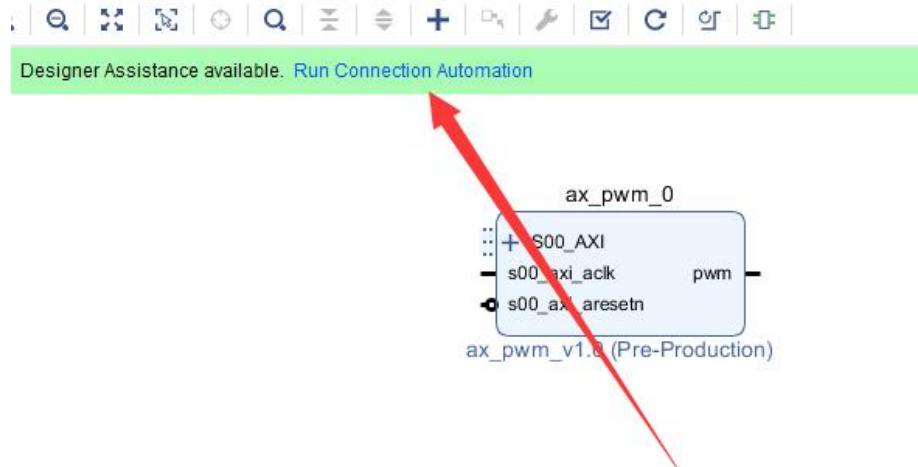


### Part 16.2.2: Add a Custom IP to the Project

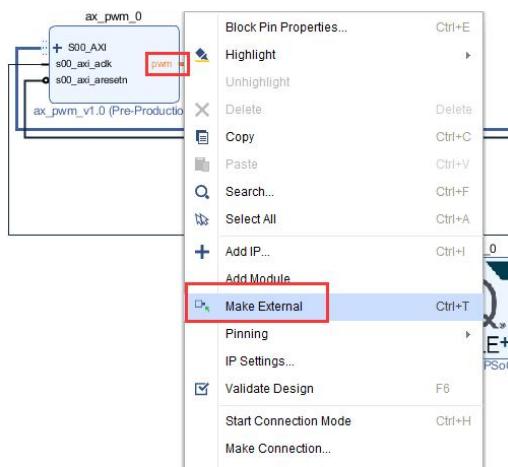
- 1) Search for "pwm" and add "ax\_pwm\_v1.0"

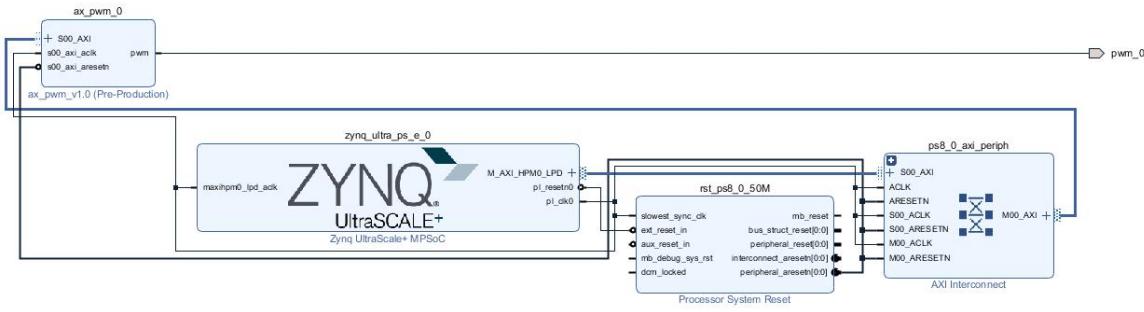


## 2) Click on "Run Connection Automation"

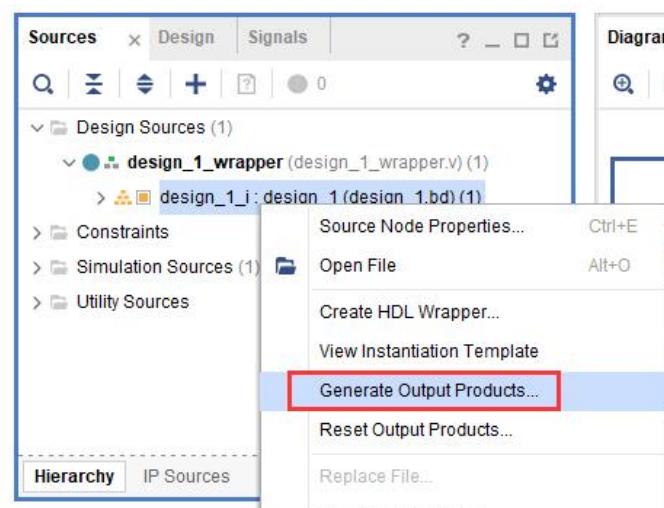


## 3) Export pwm port





#### 4) Save the design and Generate Output Products



#### 5) Add xdc file assignment pins, assign pwm\_0 output port to PLLED1, and make a breathing light

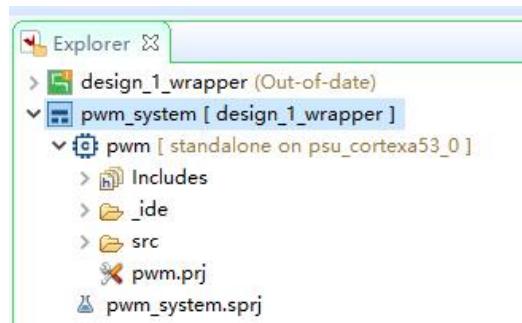
```
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports pwm_0]
set_property PACKAGE_PIN AE12 [get_ports pwm_0]
```

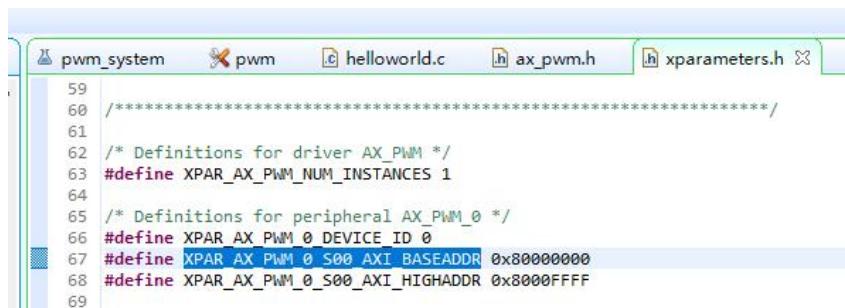
#### 6) Compile and generate a bit file and export the hardware

### Part 16.3: Vitis software Writing and Debugging

- 1) Start the Vitis to create a new APP and select Hello World as the template.



- 2) Find the "xparameters.h" file in bsp. This very important file, which finds the register base address of the custom IP, can find the base address of the custom IP



- 3) There is a register read and write macro and the base address of the custom IP. We start writing code to test the custom IP. We first control the PWM output frequency by writing to the register "AX\_PWM\_S00\_AXI\_SLV\_REG0\_OFFSET", and then controlling the duty cycle of the PWM output by writing to the register "AX\_PWM\_S00\_AXI\_SLV\_REG1\_OFFSET"

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

unsigned int duty;

int main()
{
    init_platform();
}
```

```

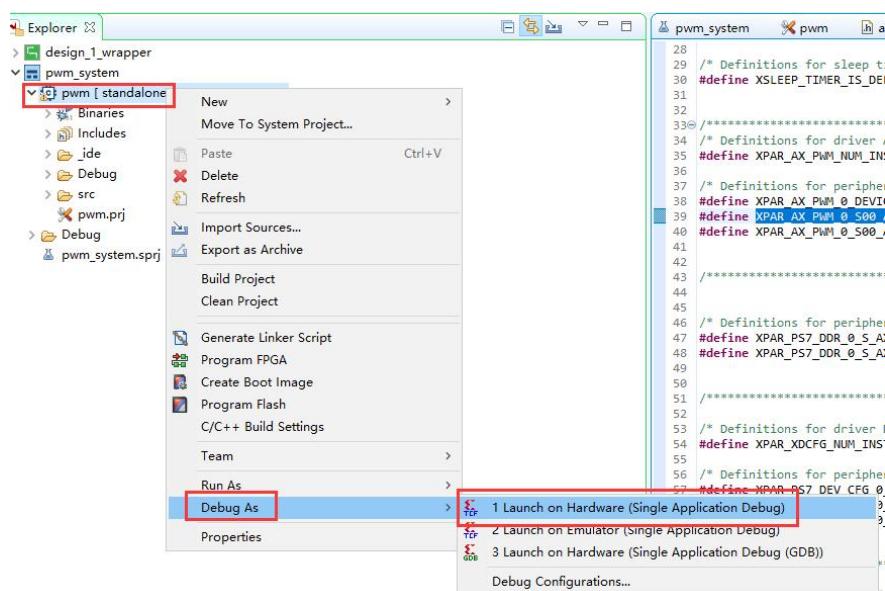
print("Hello World\n\r");

//pwm out period = frequency(pwm_out) * (2^N) / frequency(clk);
AX_PWM_mWriteReg(XPAR_AX_PWM_0_S00_AXI_BASEADDR,
AX_PWM_S00_AXI_SLV_REG0_OFFSET, 17179); //200hz
//duty = (2^N) * (1 - (duty cycle)) - 1
while (1) {
    for (duty = 0xffffffff; duty < 0xffffffff; duty = duty + 100000)
    {
        AX_PWM_mWriteReg(XPAR_AX_PWM_0_S00_AXI_BASEADDR,
AX_PWM_S00_AXI_SLV_REG1_OFFSET, duty);
        usleep(100);
    }
}

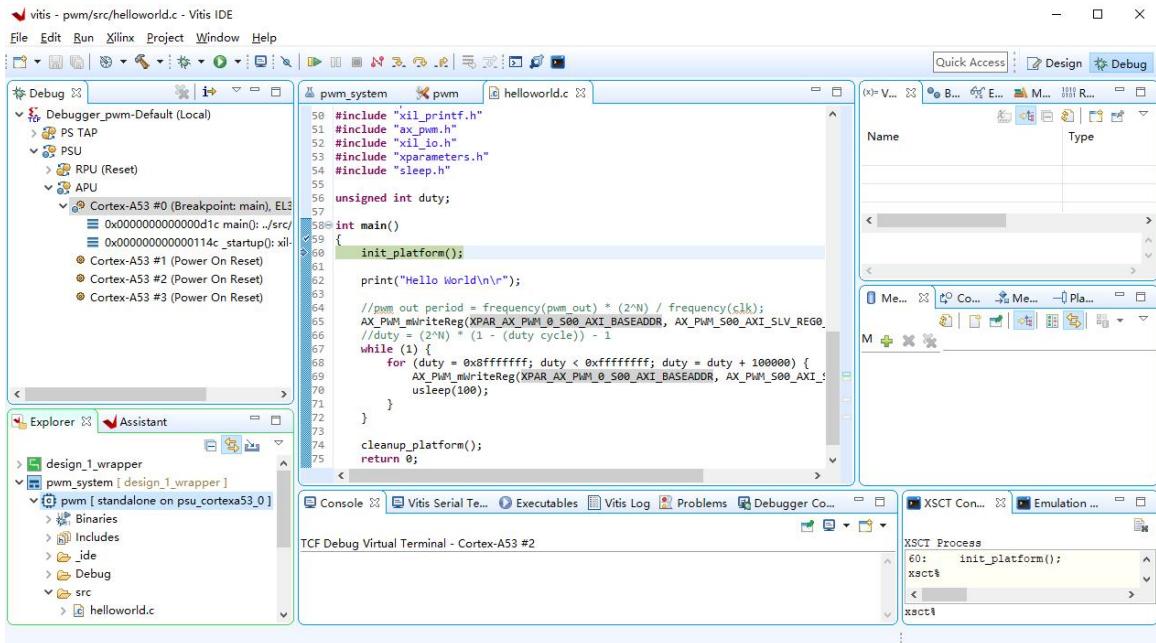
cleanup_platform();
return 0;
}

```

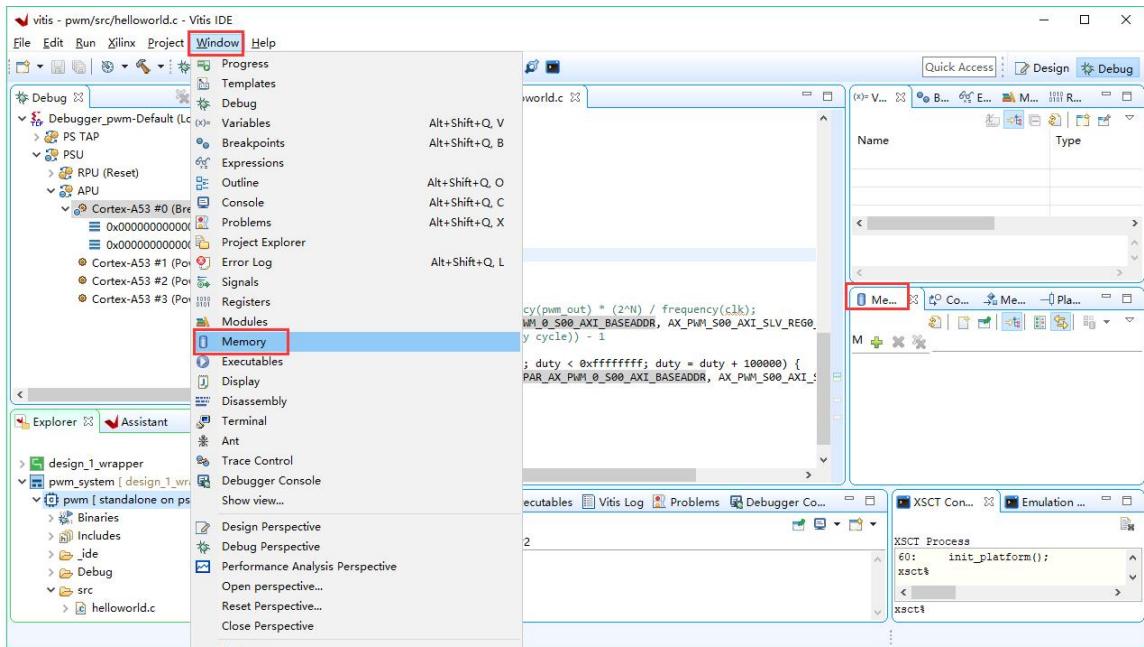
- 4) By running the code, we can see that PL LED shows the effect of a breathing light.
- 5) Through debug, let's look at the register



- 6) Enter the debug state and press "F6" to step through

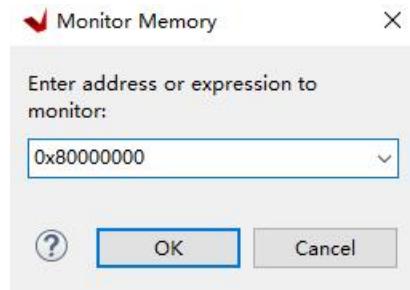


## 7) View the "Memory" window through the menu

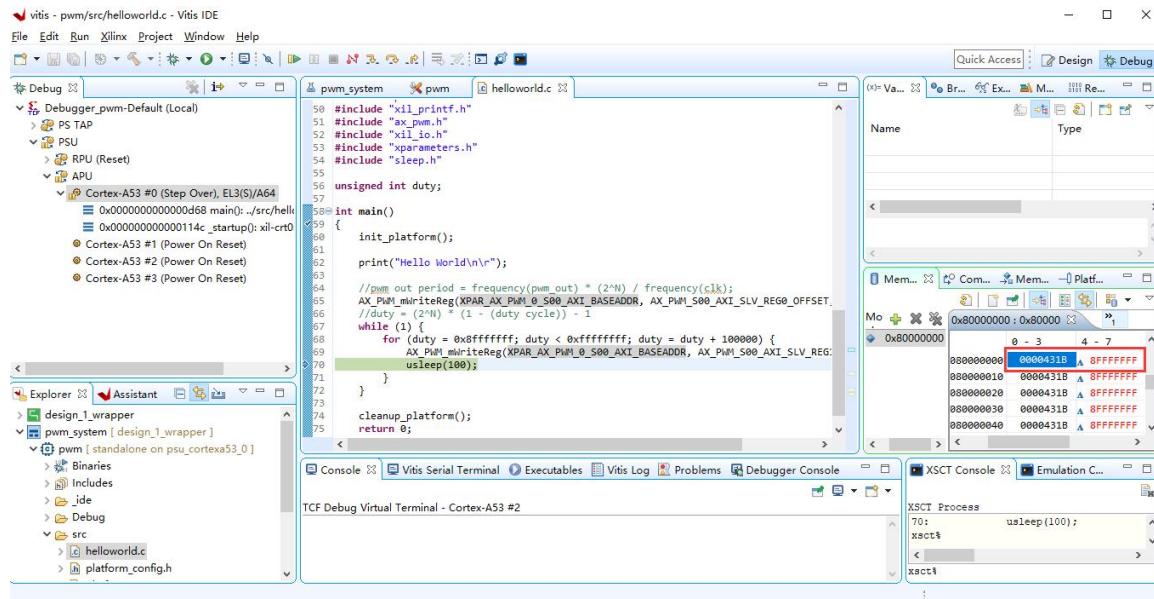


## 8) Add a monitoring address "0x80000000"





## 9) Single step, observe changes



## Part 16.4: Experimental Summary

Through this experiment, we have mastered more Vitis debugging skills, and mastered the core content of ARM + FPGA development, which is ARM and FPGA data interaction.

## Part 17: Use of Dual Core AMP

The experimental Vivado project directory is "ps\_axi\_gpio /vivado".

The experiment vitis project directory is "dualcore\_amp /vitis".

The previous routines are all single-core CPUs. In some cases, such as multitasking, parallel processing, etc., dual-core CPUs are required. This chapter will briefly introduce the use of dual-core. And will implement the following features:

CPU0 realizes the PS side key interruption, controls the PS side LED, and transmits a software interrupt to the CPU1, allowing the CPU1 to print a string of characters in the CPU0 memory space.

CPU1 realizes the PL side key interruption, controls the PL side LED, and transmits a software interrupt to the CPU0, allowing the CPU0 to print a string of characters in the CPU0 memory space.

Division of memory space, use of shared memory space

FSBL starts Flash

Vivado uses the project of "PL Side Use of AXI GPIO"

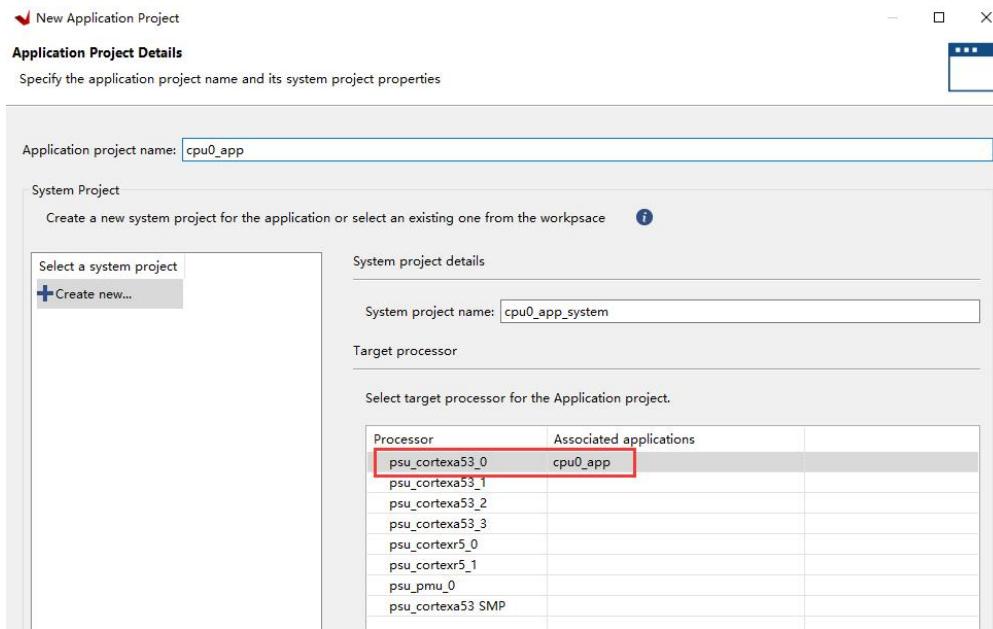
### Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

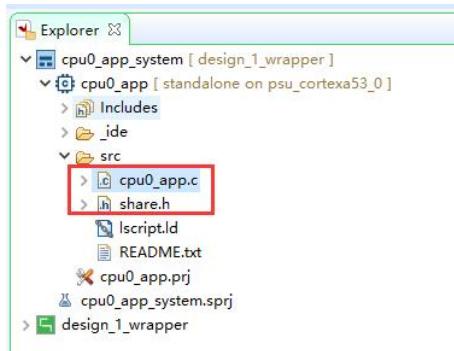
#### Part 17.1:Vitis Program Development

##### Part 17.1.1: Create CPU0 Vitis Project

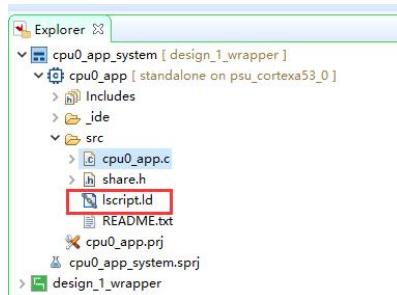
- 1) Create a new project, Note that the CPU selects "ps7\_cortexa53\_0" , which is "CPU0".

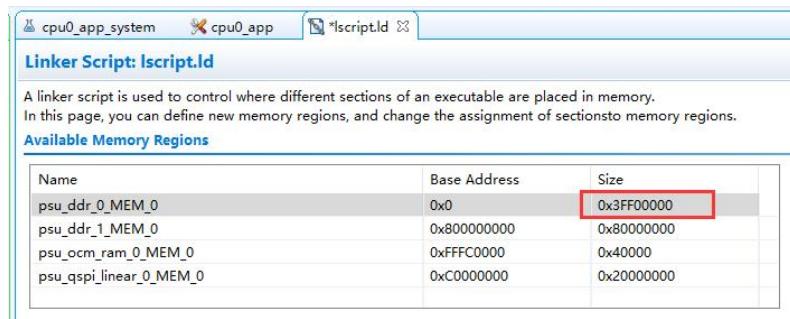


- 2) The code has been prepared for everyone, “cpu0\_app.c” and “share.h”, “share.h” contains the shared memory structure, which will be discussed later.



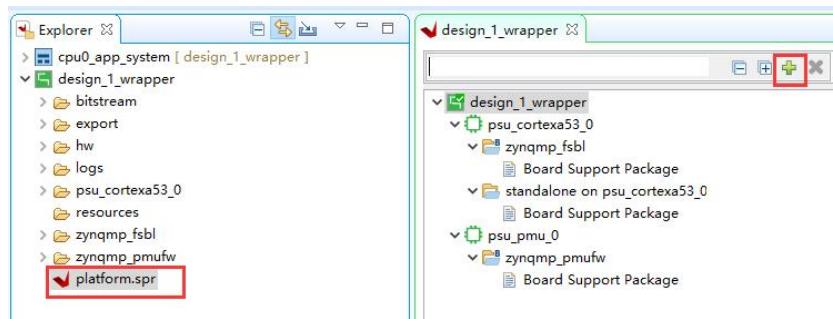
- 3) Set the access space of “CPU0” in “lscript.ld”. For example, psu\_ddr\_0\_MEM\_0 is 2GByte, and the CPU0 space is set to half. Of course, it can be modified as needed.



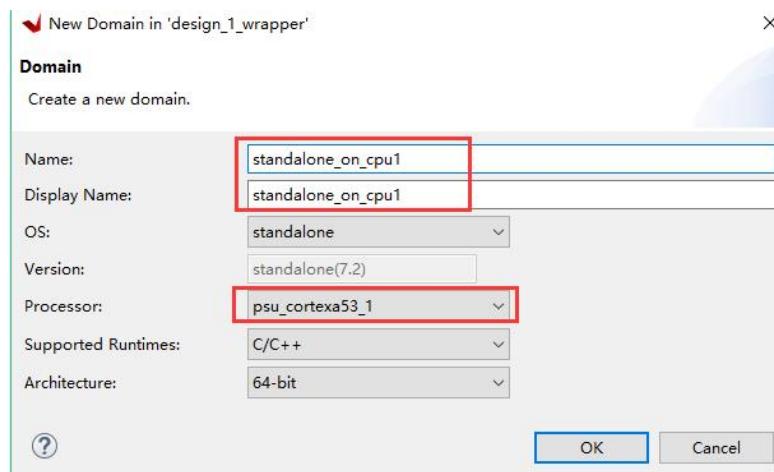


### Part 17.1.2: Create a CPU1 Vitis project

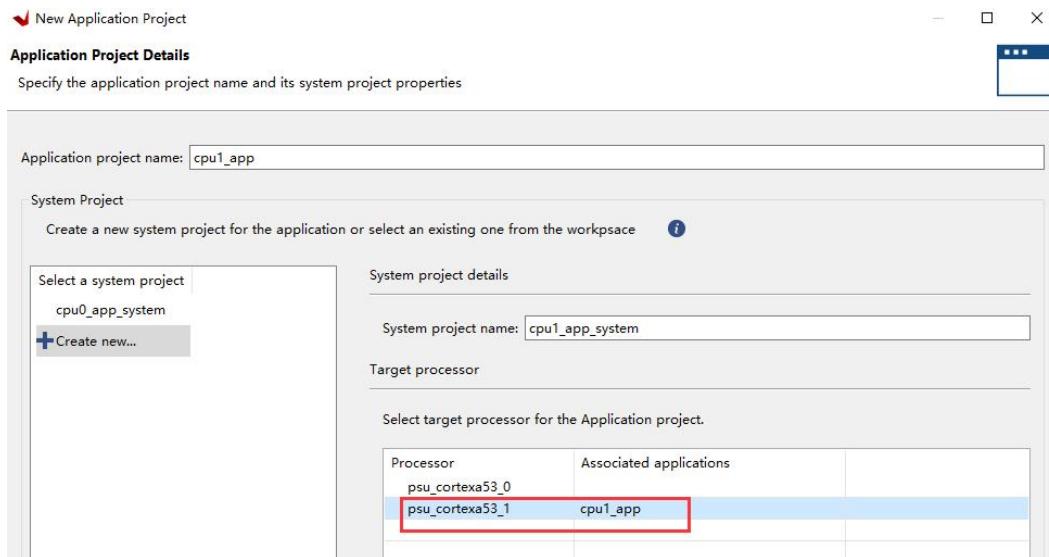
- Before creating a new CPU1 APP project, we can create a new domain based on CPU1, which is the so-called BSP, click "+" in platform.spr



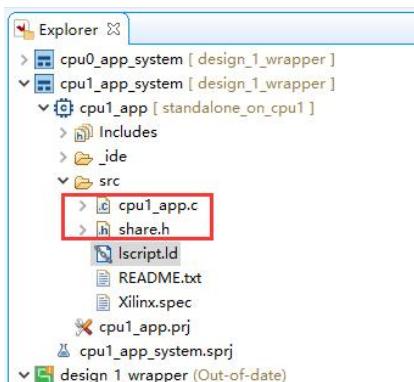
- Fill in the name, and the Processor selects ps7\_cortexa9\_1, which is CPU1, click OK



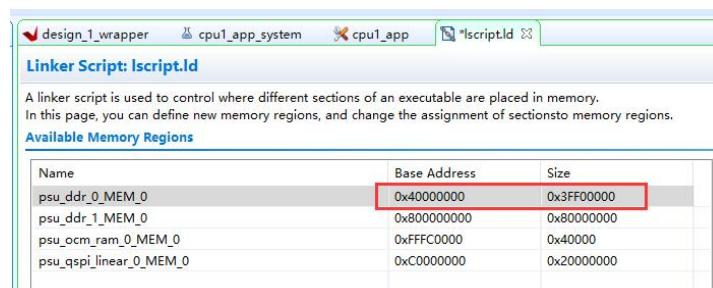
- When creating a new CPU1 project, select CPU1



- 4) Also prepared the code, cpu1\_app.c and share.h



- 5) Set the memory space of CPU1, be careful not to overlap with CPU0, the space between 0x3FF00000~0x3FFFFFFF is reserved for shared memory



Compile the APP project of CPU0 and CPU1

### Part 17.1.3: CPU0 Programs Introduction

- 1) Software interrupts use ID numbers 1 and 2.

```
u16 SoftIntrIdToCpu0 = 1 ;
u16 SoftIntrIdToCpu1 = 2 ;
```

And connect the interrupt number 1 to the software interrupt service function.

```
InterruptConnnect(&InterruptInst, SoftIntrIdToCpu0, (void *)SoftHandler, (void *)&InterruptInst) ;
```

- 2) In the “**while**” loop statement, the address and length of the character array are assigned to the shared structure. Here, the shared memory structure is mentioned. The structure “**ShareMem**” is defined in “**share.h**” for passing information in the shared memory.

```
while(1)
{
    if (key_flag)
    {
        /*
         * initial shared Struct
         */
        SharePtr->addr = (unsigned int *)&Cpu0_Data ;
        SharePtr->length = sizeof(Cpu0_Data) ;
        /*
         * Write led value
         */
        XGpioPs_WritePin(&GPIO_PTR, 9, LedVal) ;
        LedVal = ~LedVal ;
        /*
         * Software interrupt to CPU1
         */
        XScuGic_SoftwareIntr(&InterruptInst, SoftIntrIdToCpu1, CPU1) ;
        key_flag = 0 ;
    }

    typedef struct
    {
        unsigned int length;
        unsigned int *addr;
    }ShareMem;
```

And the dual core agrees to share the address so that parameters can be passed.

```
#define SHARE_BASE 0x3FFFFF00
```

- 3) In the “**while**” loop, it is judged that there is a software interrupt from “**CPU1**”, and a character string in the memory space of **CPU1** is printed.

```
else if (soft_flag)
{
    /*
     * When Software interrupt, print data in CPU1
     */
    Cpu1Data = (unsigned char *)SharePtr->addr ;
    xil_printf("This is CPU0, Now Start to Print:\r\n") ;
    xil_printf("%s\r\n", Cpu1Data) ;
    soft_flag = 0 ;
}
```

#### Part 17.1.4: CPU1 Programs Introduction

- 1) There is also a character array in the “**CPU1**” program, and

“Cpu0Data” points to the string address of the “CPU0” memory space.

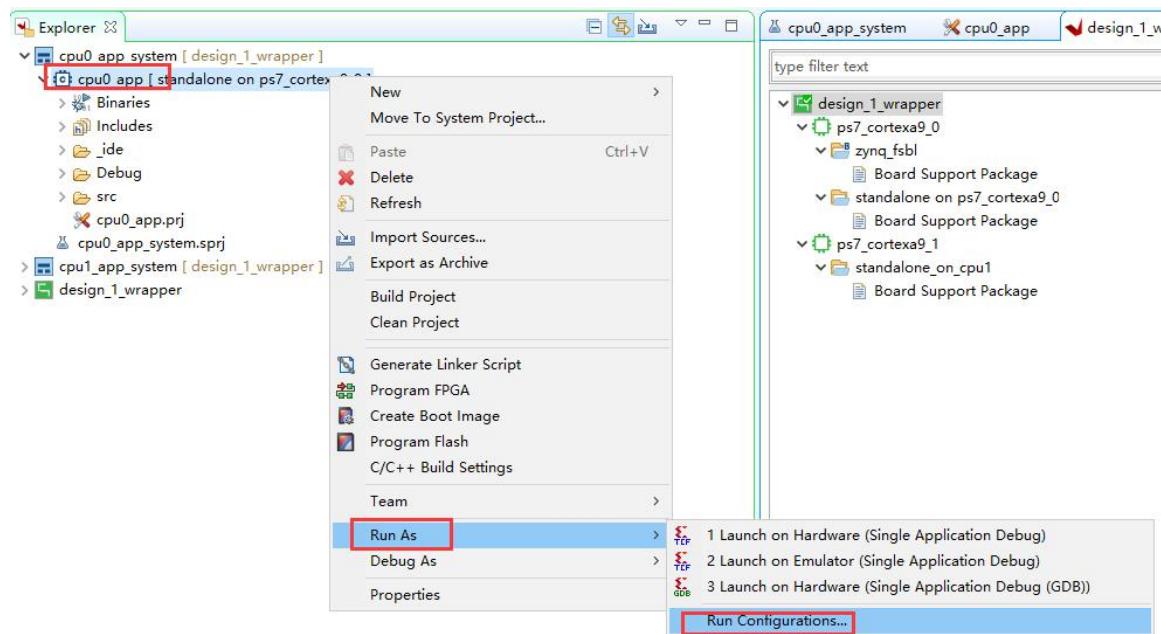
```
unsigned char Cpu1_Data[12] = "Hello Cpu0!" ;
unsigned char *Cpu0Data ;
```

- 2) In the “PLGpioSetup” function, you need to bind the key interrupt number to “CPU1”, and other parts are similar to “CPU0”, and will not be described again.

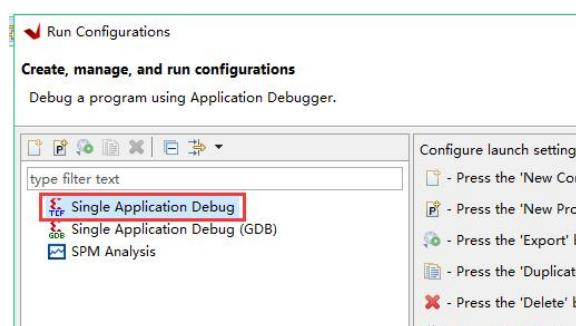
```
/* Connect key interrupt to CPU1 */
XScuGic InterruptMapToCpu(InstancePtr, XPAR_CPU_ID, KEY_INTR_ID) ;
```

### Part 17.3: Onboard Verification

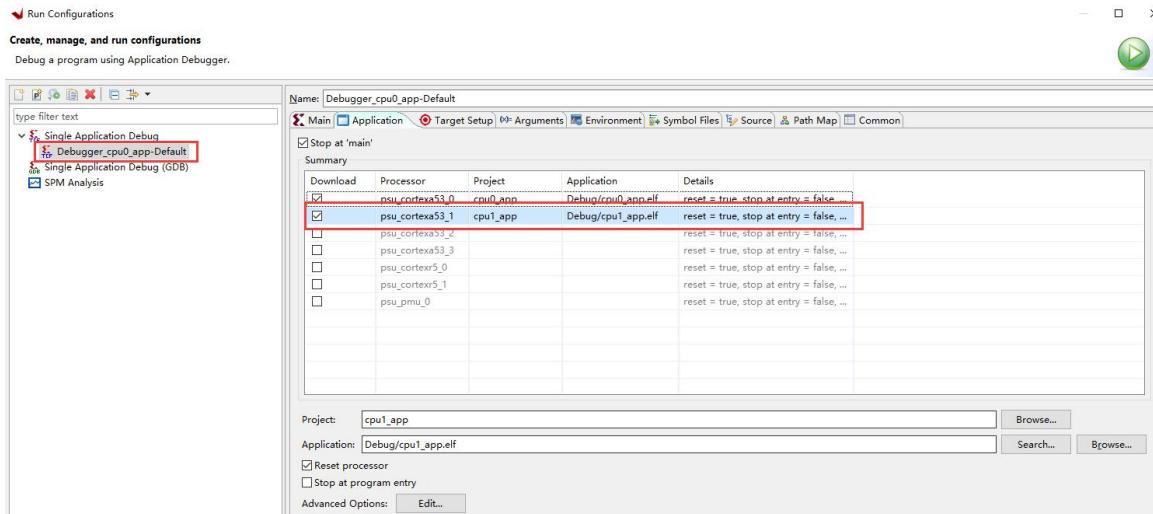
- 1) Run Configurations, and Configuration is as follows:



- 2) Double click Single Application Debug



- 3) Check CPU1, other defaults, click Run



- 4) Open the serial port software, test CPU0, press the PS side button, and control the PS side LED light to turn on, indicating that CPU0 is running, and CPU1 receives the software interrupt set by CPU0 at the same time, and prints out the information.

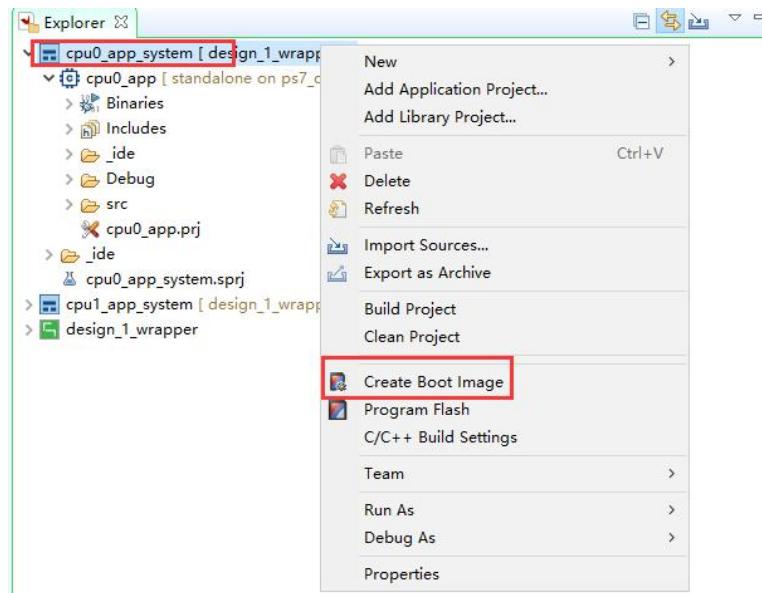
```
Soft Interrupt from CPU0
This is CPU1, Now Start to Print:
Hello Cpu1!
```

- 5) Test CPU1, press the button on the PL side, and control the LED on the PL side to turn on, indicating that CPU1 is running, and at the same time CPU0 receives the software interrupt set by CPU1 and prints out the information.

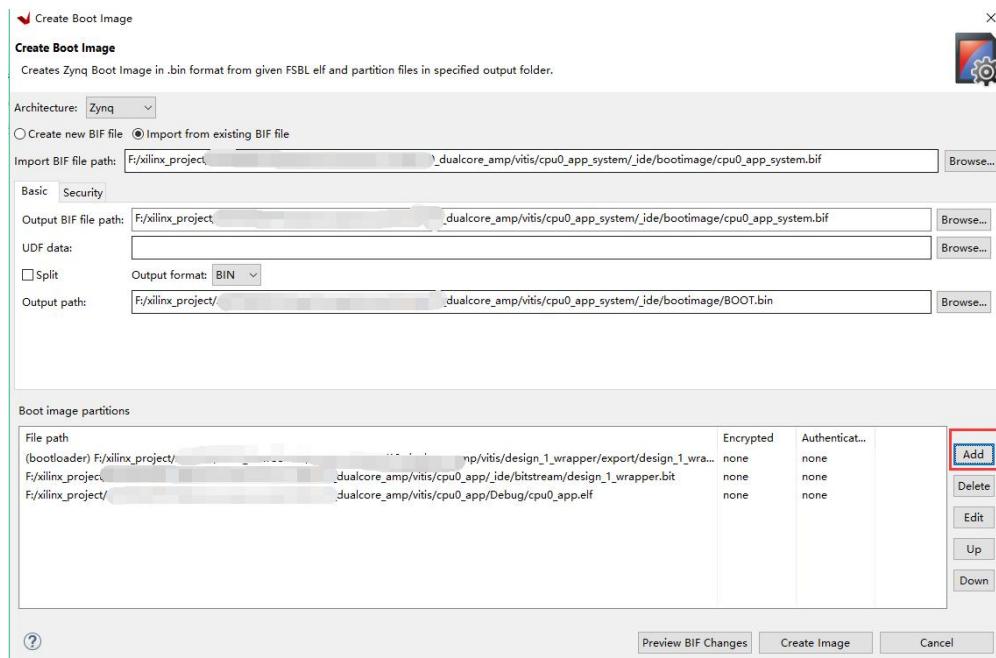
```
Soft Interrupt from CPU0
This is CPU1, Now Start to Print:
Hello Cpu1!
Soft Interrupt from CPU1
This is CPU0, Now Start to Print:
Hello Cpu0!
```

### Part 17.3: QSPI Flash Startup

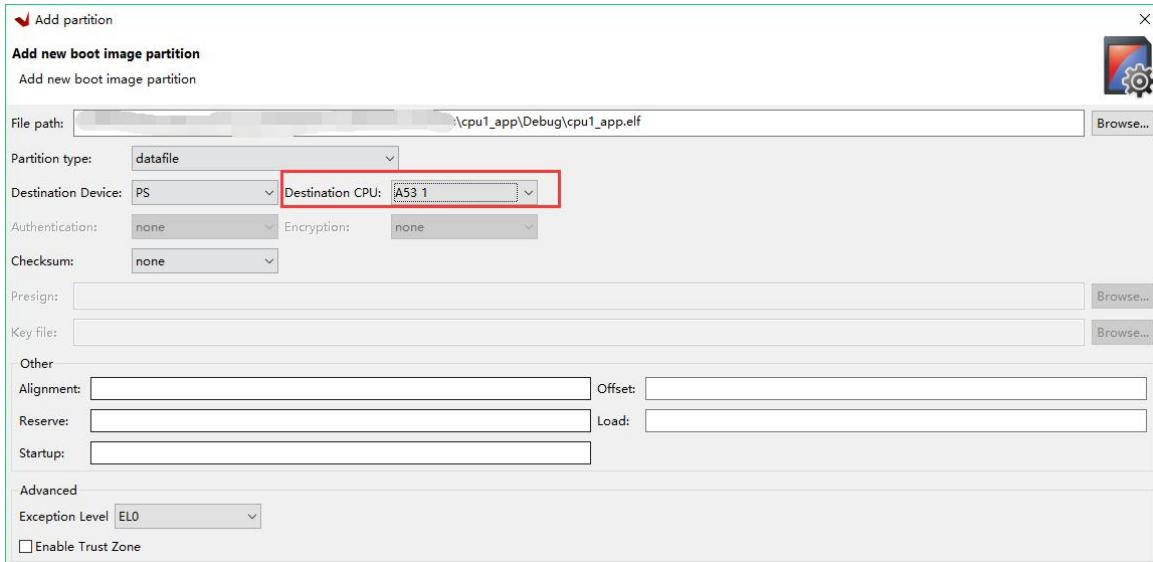
The way to generate BOOT.BIN is different from the previous Build Project generation, we need to configure it. Right-click on the system of CPU0 and select Create Boot Image



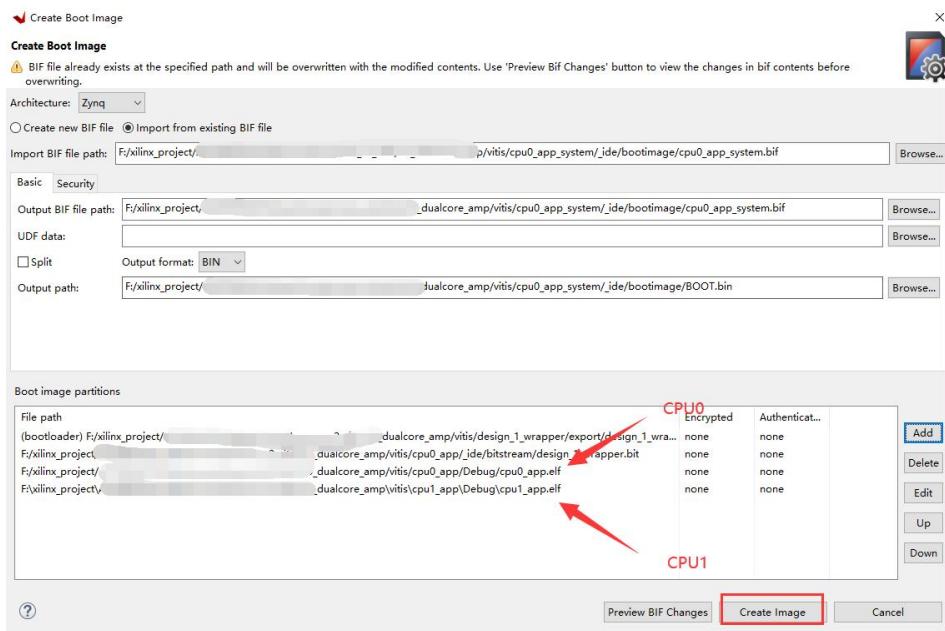
Click Add to add the elf file of CPU1



Select the elf of the corresponding CPU1, select the directory  
CPU as A53 1



The result after adding is as follows, click Create Image



## Part 17.4: Experimental Summary

This chapter provides a brief introduction to how to use dual cores in bare metal, as well as interrupt use and communication between dual cores. In this experiment, the length members in the shared memory structure are not used. You can try to copy the data of the two cores according to the length and address.

## Part 18: Use of “Free RTOS” under ZYNQ

The experimental Vivado project directory is "ps\_axi\_gpio /vivado".

The experimental vitis project directory is "freertos /vitis".

Learning “ZYNQ”, a large part of the FPGA developers, is not very good at using Linux, so I suggest that you still use the real-time operating system or bare-metal operation, which also has more flexibility. This chapter explains how to set up the “Free RTOS” real-time operating system running environment. I will not delve into the specific use of “Free RTOS”. This experiment uses “FreeRTOS Hello World” as an example, and experiments the LEDs on the PS and PL sides continue to blink at different intervals.

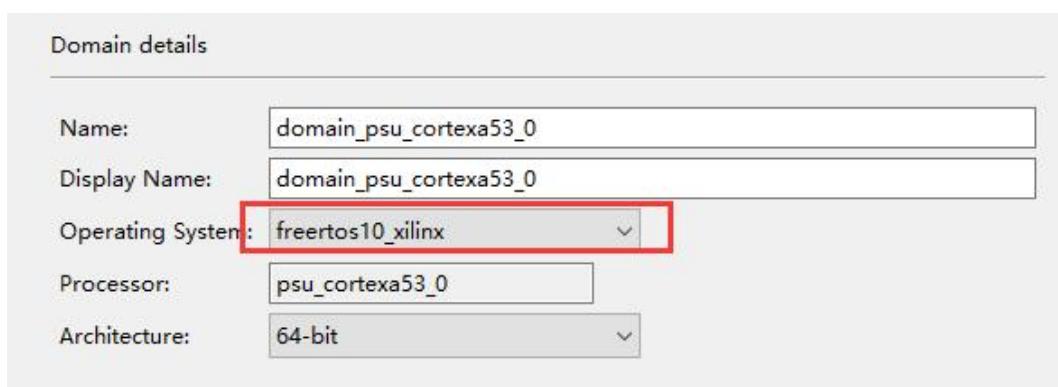
This experiment is based on the " PS Side Use of Dual-Core AXI GPIO" tutorial, the hardware environment does not need to be modified.

### Software Engineer Job Content

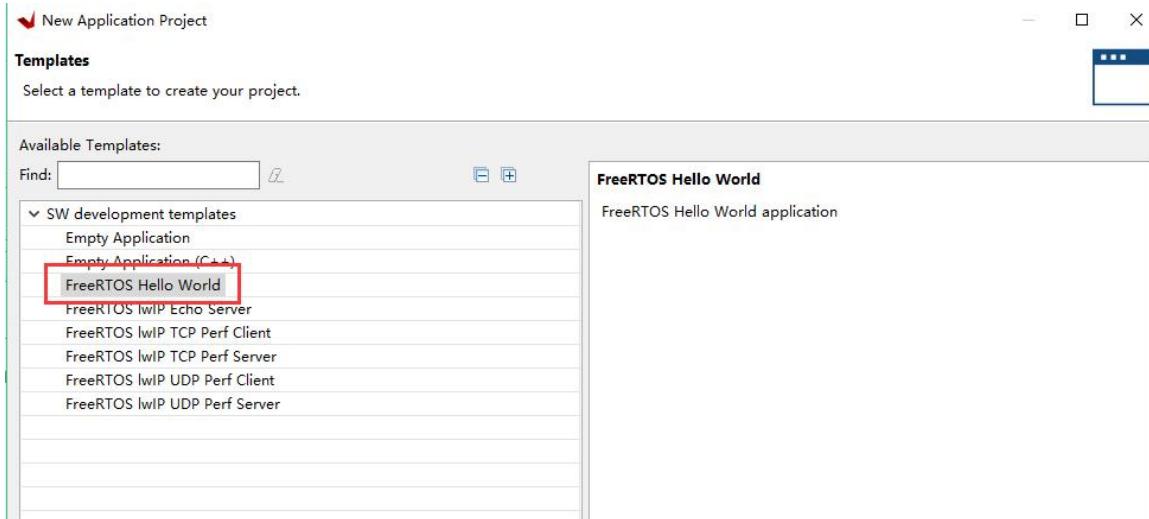
The following is the content that FPGA engineers are responsible for.

#### Part 18.1: Vitis Program Development

- 1) Create a new project, “OS Platform” select “freertos10\_xilinx”



- 2) This experiment chooses “FreeRTOS Hello World” as an example.



In the “Hello World” example, two tasks are created, transmitting tasks and receiving tasks. The priority of receiving tasks is higher than that of transmitting tasks. And create a queue, the data is transmitted to the queue by the transmitting task, and the receiving task reads the data from the queue and prints. In the example, the timer is set, but in this experiment, the timer is deleted, and the transmitting and receiving tasks are always working.

```
xTaskCreate( prvTxTask,
    ( const char * ) "Tx",
    configMINIMAL_STACK_SIZE,
    NULL,
    tskIDLE_PRIORITY,
    /* The task parameter is not used, so set to NULL. */
    /* The task runs at the idle priority. */
    &xTxTask );

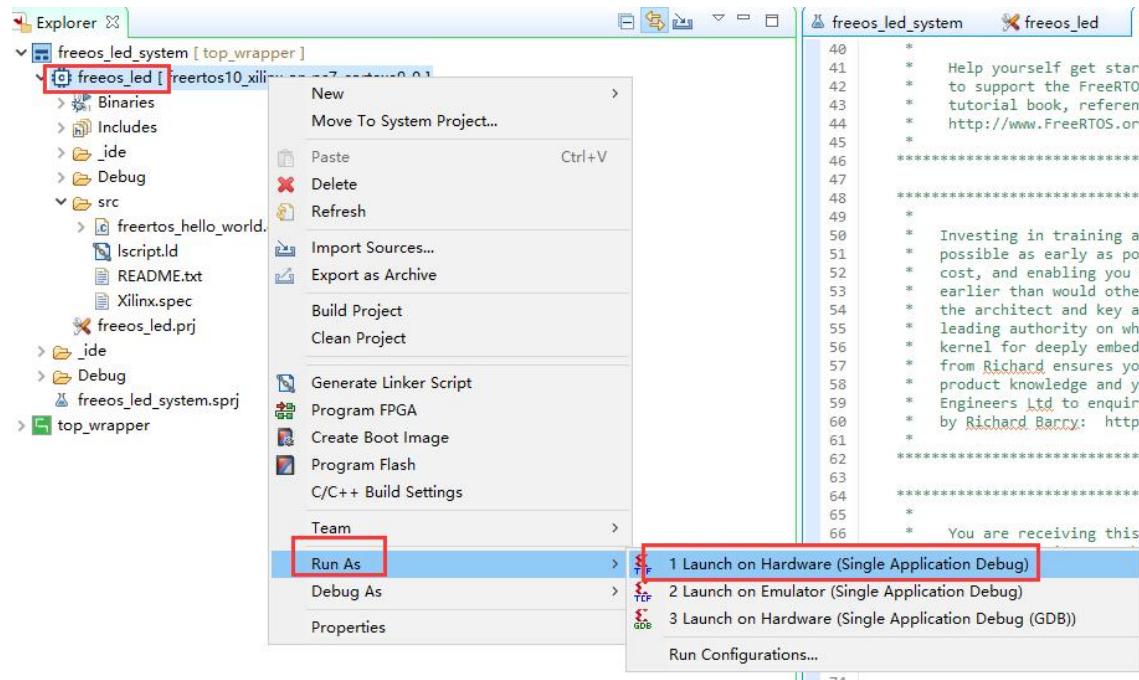
xTaskCreate( prvRxTask,
    ( const char * ) "Rx",
    configMINIMAL_STACK_SIZE,
    NULL,
    tskIDLE_PRIORITY + 1,
    &xRxTask );
```

- 3) On this basis, the PS and PL LED blinking tasks are added. The flicker interval of the PS end is 100ms, and the flicker interval of the PL end is 1S.

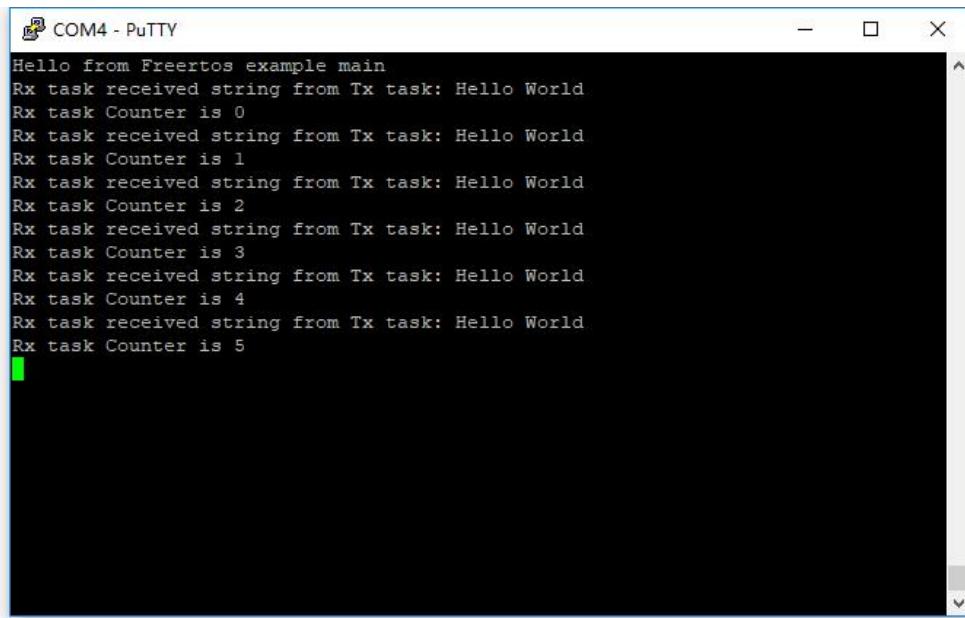
```
xTaskCreate( prvPsLedTask,
    ( const char * ) "Ps_Led",
    configMINIMAL_STACK_SIZE,
    NULL,
    tskIDLE_PRIORITY + 1,
    NULL);
xTaskCreate( prvPlLedTask,
    ( const char * ) "PL_Led",
    configMINIMAL_STACK_SIZE,
    NULL,
    tskIDLE_PRIORITY + 1,
    NULL);
```

## Part 18.2: Onboard Verification

### 1) Download interface settings, download the program



### 2) Open the serial port and continue to print data



```

Hello from Freertos example main
Rx task received string from Tx task: Hello World
Rx task Counter is 0
Rx task received string from Tx task: Hello World
Rx task Counter is 1
Rx task received string from Tx task: Hello World
Rx task Counter is 2
Rx task received string from Tx task: Hello World
Rx task Counter is 3
Rx task received string from Tx task: Hello World
Rx task Counter is 4
Rx task received string from Tx task: Hello World
Rx task Counter is 5

```

### 3) At the same time, you can also see the LED flashing on the PS side and PL side on the development board, which intuitively reflects the multi-task parallel processing.

### Part 18.3: Experimental Summary

Compared with complex Linux, real-time operating systems such as FreeRTOS bring us more flexible and convenient development, and can interact with the underlying FPGA more directly, but FreeRTOS itself has a little difficulty. To be skilled, it is necessary to combine specific projects. Practice more.

## Part 19: PL Read and Write PS DDR Data

The experimental Vivado project directory is "pl\_read\_write\_ps\_ddr/vivado".

The experiment vitis project directory is "pl\_read\_write\_ps\_ddr /vitis".

The efficient interaction between PL and PS is the top priority of zynq soc development. We often need to transmit a large amount of data from the PL side to the PS side for processing in real time, or transmit the PS side processing results to the PL side for processing in real time.

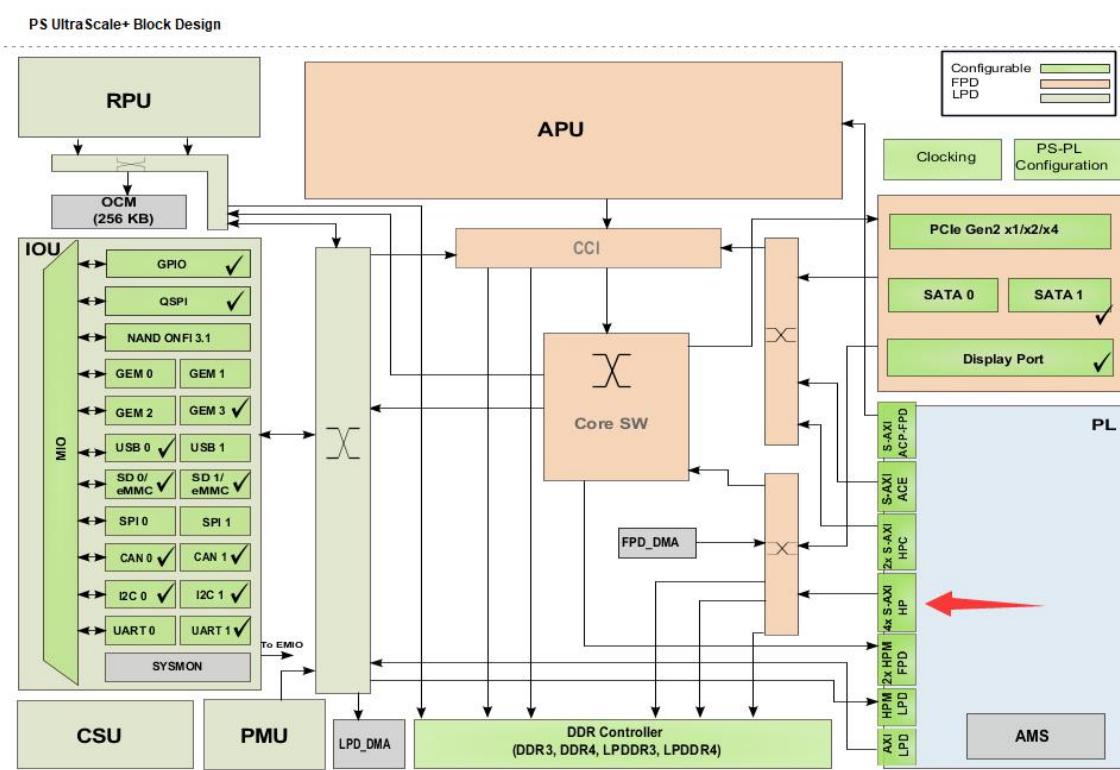
Normally we would think of using DMA to do this, but various protocols are very troublesome and flexibility is relatively poor. This section of the course explains how to directly read and write data on the PS side ddr through the AXI bus. This involves the AXI4 protocol, vivado FPGA debugging and so on.

### FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

#### Part 19.1: Use of ZYNQ HP Port

The “HP” port of the “zynq 7000 SOC” is the abbreviation of “High-Performance Ports”. As shown in the figure below, there are 4 HP ports. The HP port is “AXI Slave” device. We can realize high-bandwidth data interaction through these 4 HP interfaces.

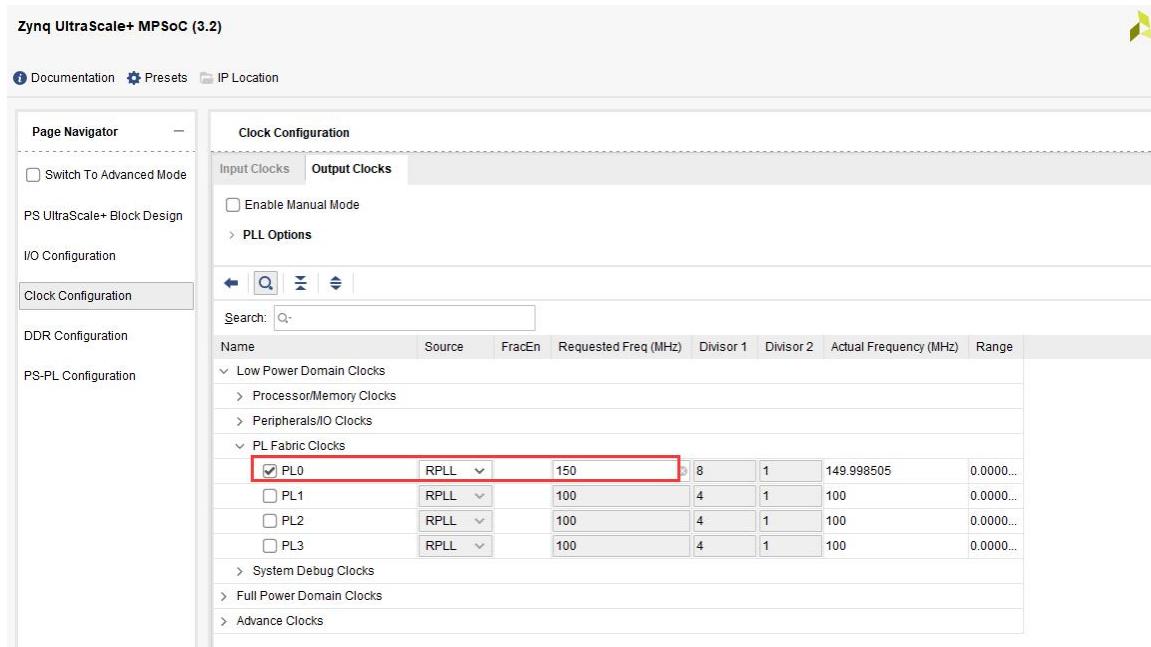
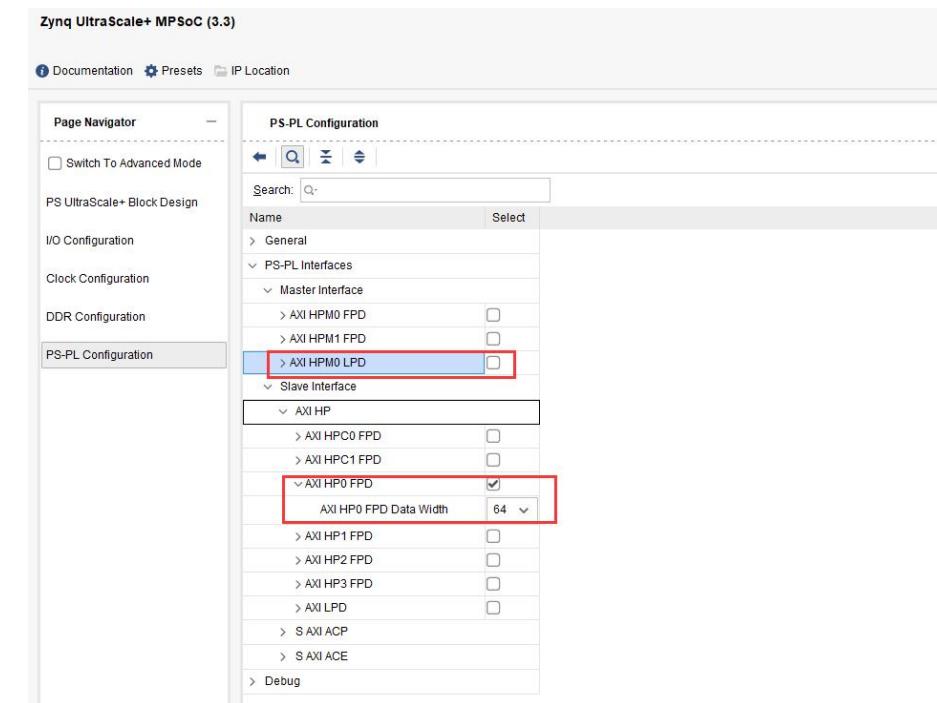


## FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 19.2: Hardware Environment

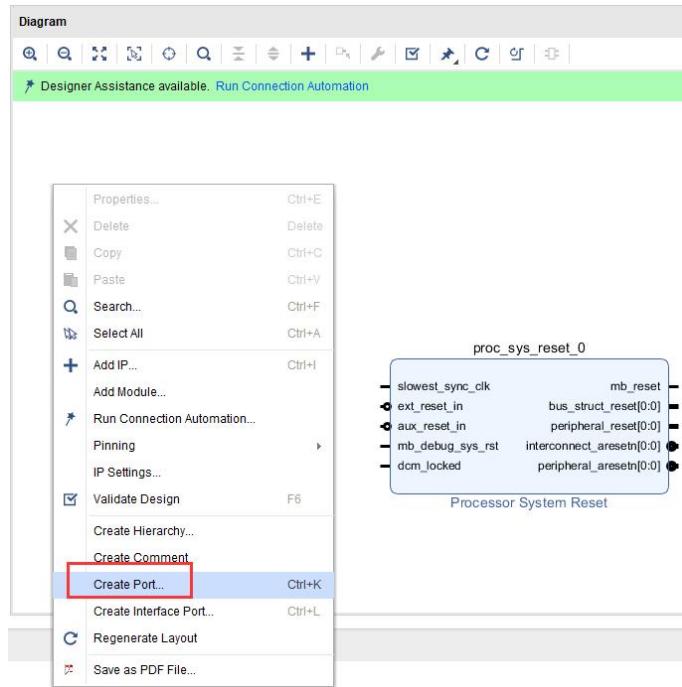
- 1) Based on the "ps\_hello" project, the HP configuration in the vivado interface is as shown in the figure below (HP0~HP3). There are enable control and data bit width selection. You can choose 32bit, 64bit or 128bit. In our experiment, HP0 is configured to be 64bit wide, the clock used is 150Mhz, and the bandwidth of HP is 150Mhz \* 64bit, which has sufficient bandwidth for video processing, ADC data acquisition and other applications. No need for AXI HPM0 LPD, deselect it.



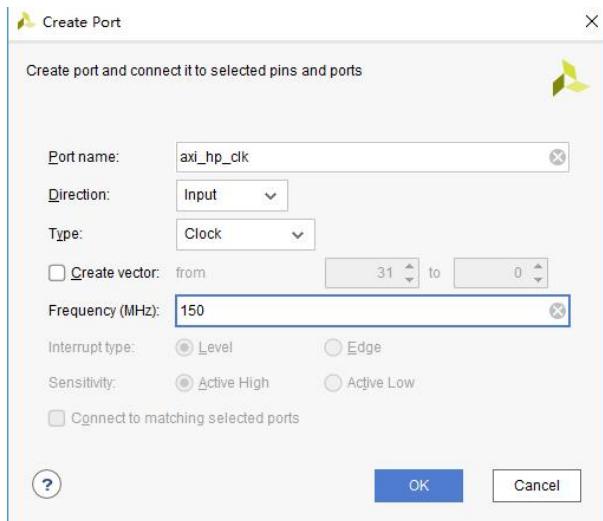
## 2) Add reset module for reset



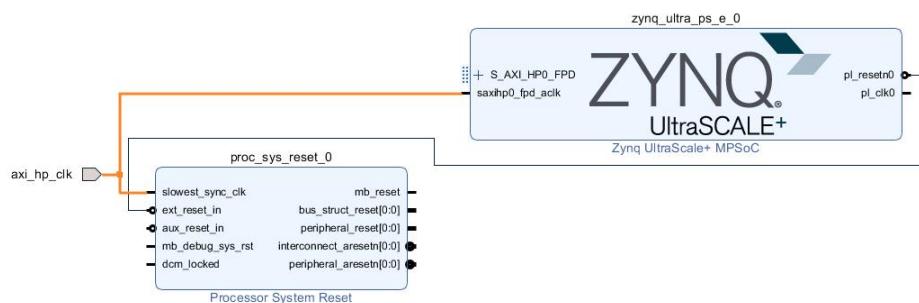
- 3) Right-click on the blank space and select "Create Port"



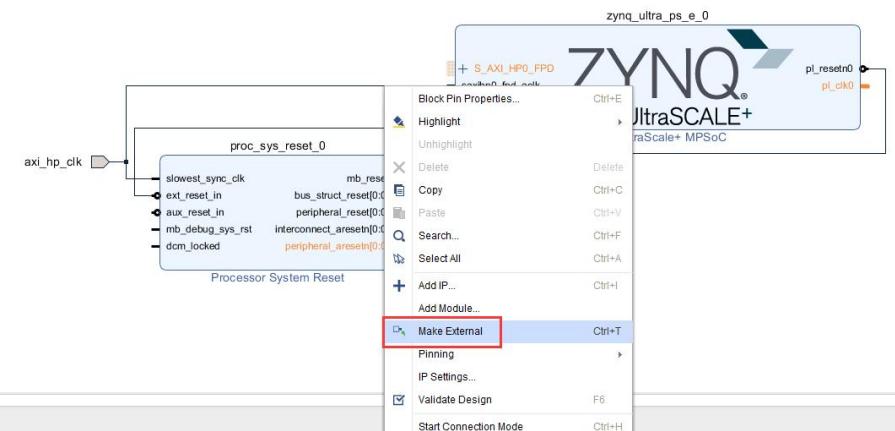
Configuration as shown



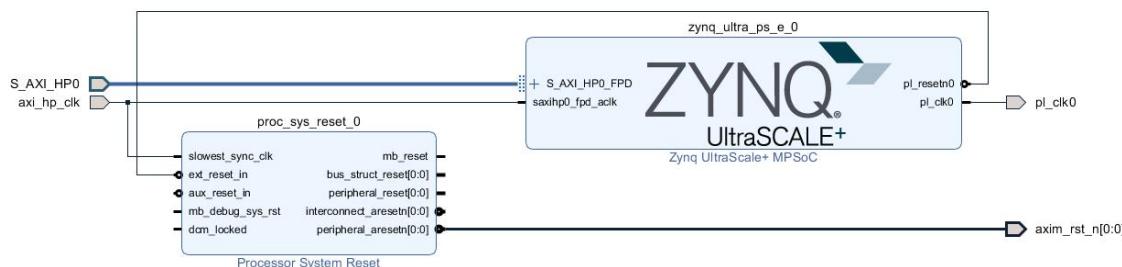
- 4) Connect the clock and reset



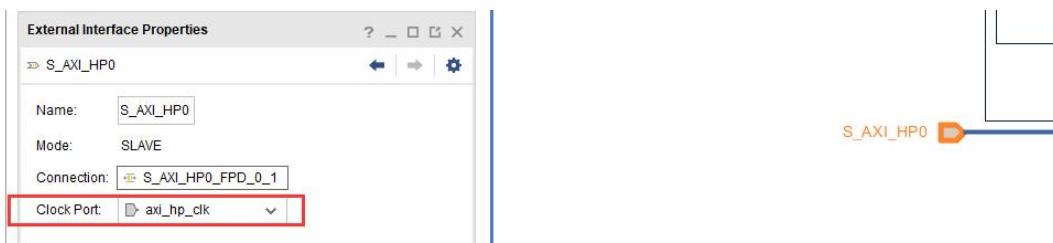
- 5) Select the pin and click Make External to export the signal



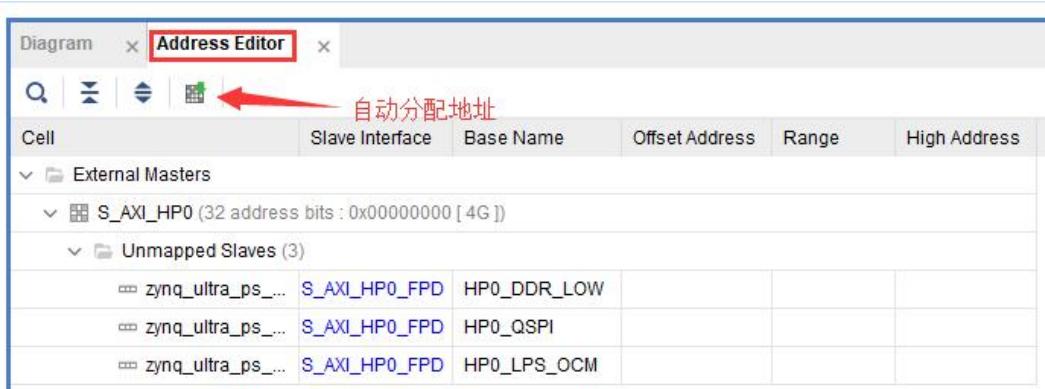
And modify the pin name as shown below



And select the bus synchronization clock as axi\_hp\_clk



- 6) Click on the Address Editor, if you find that the address is not assigned, click the button to automatically assign an address

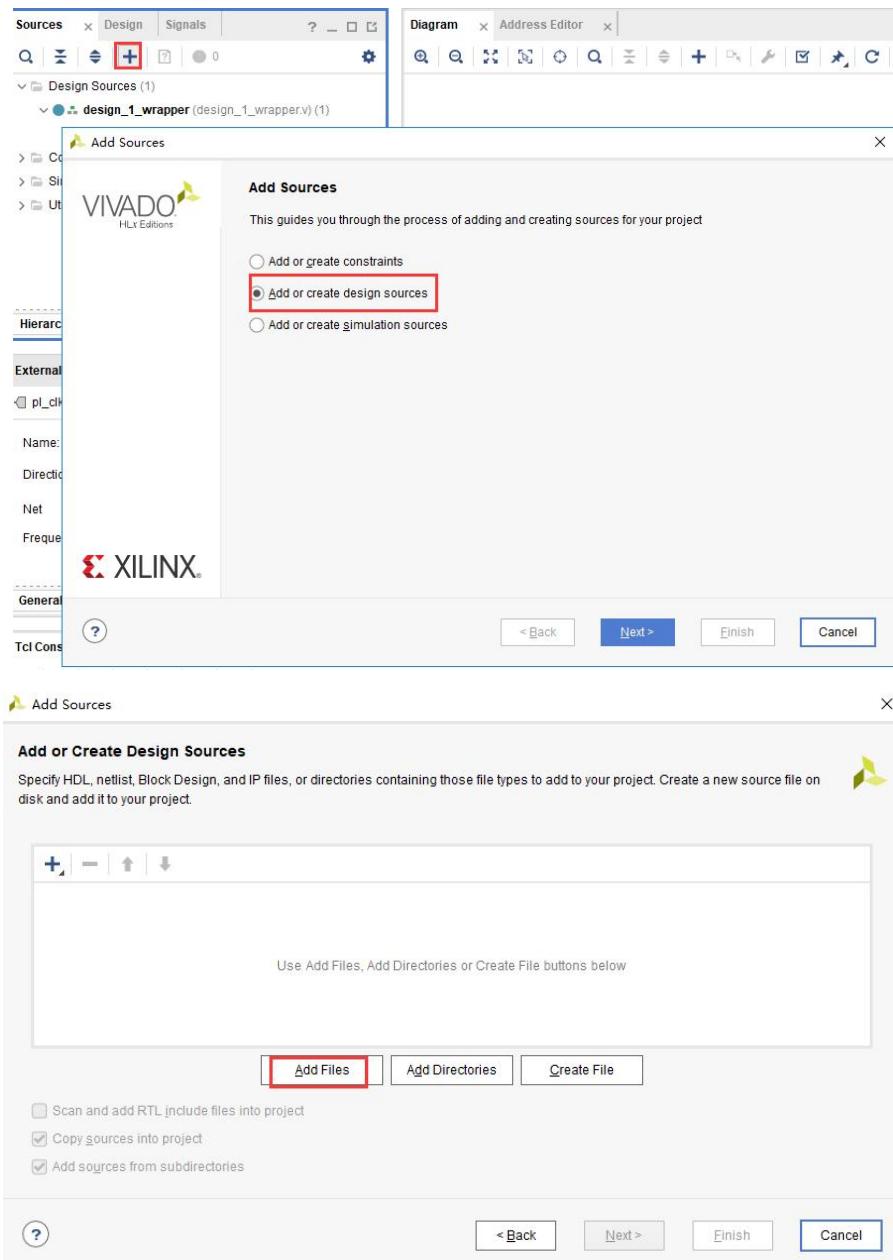


After allocation, you can see the address space for accessing DDR, QSPI, OCM

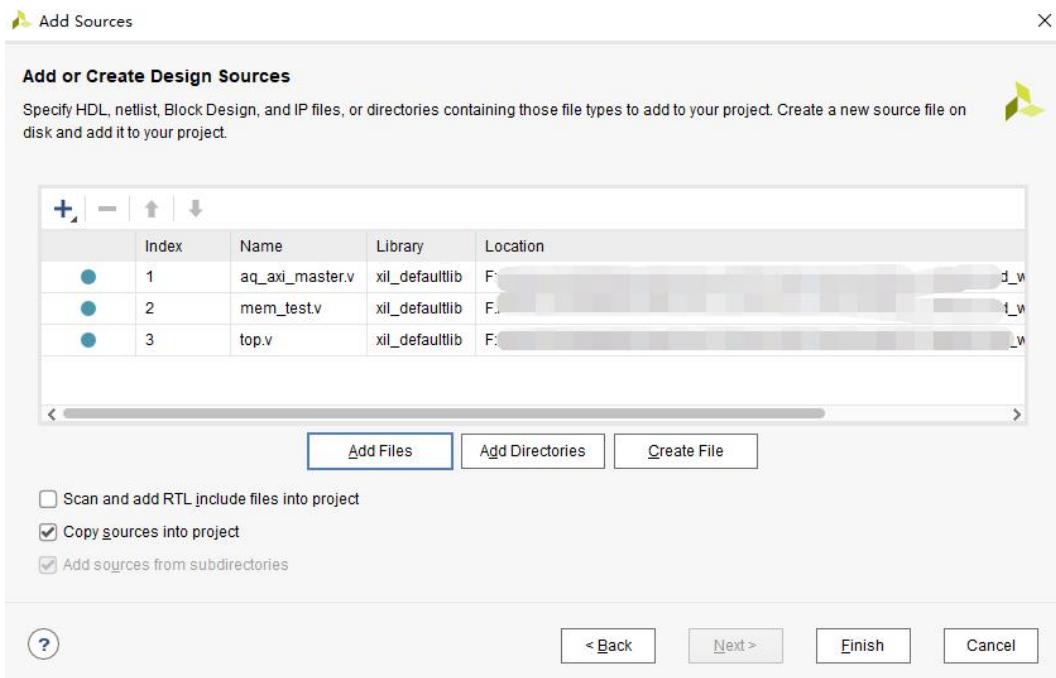
Cell	Slave Interface	Base Name	Offset Address	Range	High Address
<b>External Masters</b>					
S_AXI_HP0 (32 address bits : 0x00000000 [ 4G ])					
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_DDR_LOW	0x0000_0000	2G	0x7FFF_FFFF
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_QSPI	0xC000_0000	512M	0xFFFF_FFFF
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_LPS_OCM	0xFF00_0000	16M	0xFFFF_FFFF

Save the design and regenerate Output Product

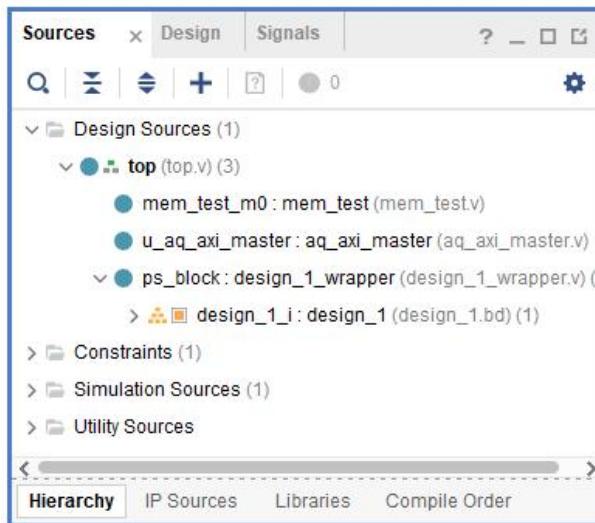
## 7) Add hdl files



## 8) Click Finish



HDL hierarchical relationship update results

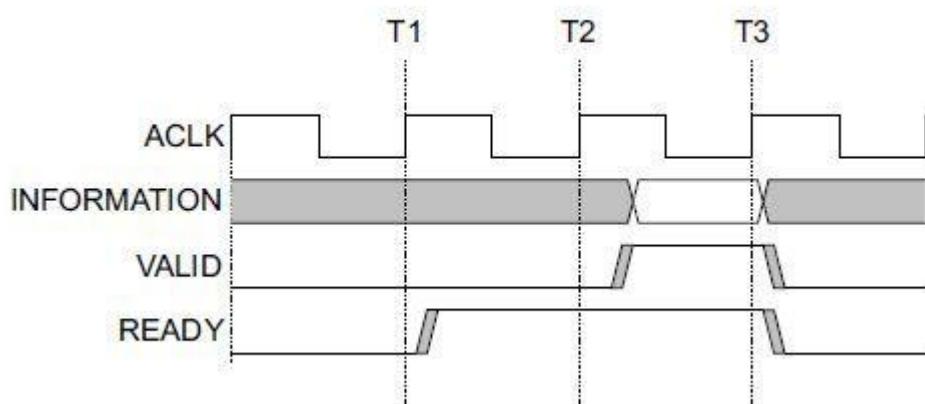


## Part 19.3: PL Side AXI Master

AXI4 is relatively complex, but SOC developers must master. For developers of zynq, I suggest that you can modify it based on some existing template code. For details on the AXI protocol, refer to the "Xilinx UG761 AXI Reference Guide". Here we can briefly understand.

AXI4 adopts a “READY”, “VALID” handshake communication

mechanism, that is, before the master-slave module performs data communication, it first handshakes the data and address channels used according to the operation. The main operation includes transmitting the READY signal of the receiver B received by the sender A, and A sends the data and the VALID signal to B at the same time, which is a typical handshake mechanism.



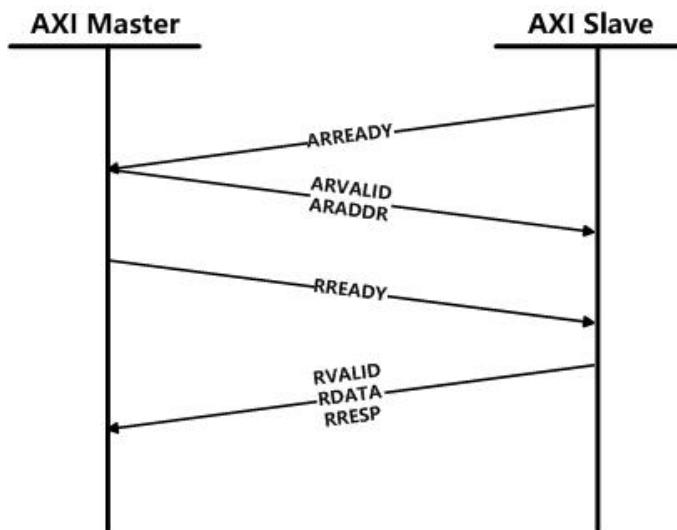
The AXI bus is divided into five channels:

- Read address channel, include: ARVALID, ARADDR, ARREADY signals
- Write address channel, include: AWVALID, AWADDR, AWREADY signals
- Read data channel, include: RVALID, RDATA, RREADY, RRESP signals
- Write data channel, include : WVALID, WDATA , WSTRB, WREADY signals
- Write response channel, include : BVALID, BRESP, BREADY signals
- System channel, include: ACLK, ARESETN signals

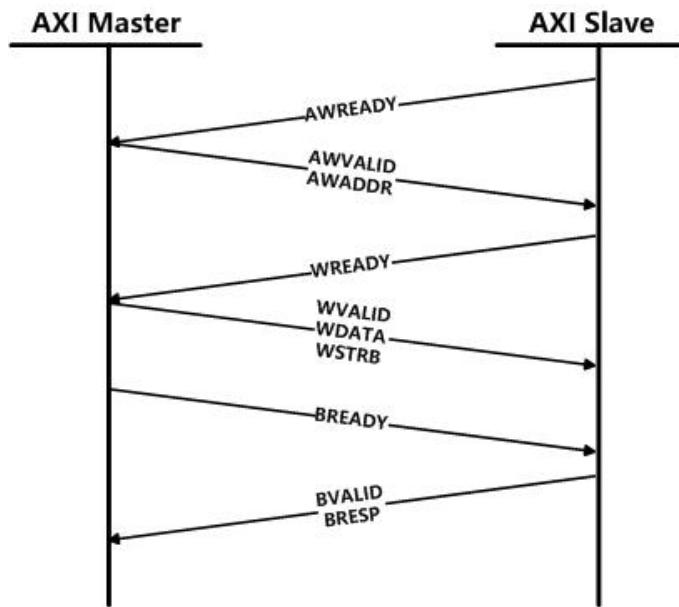
“ACLK” is “axi” bus clock, “ARESETN” is “axi” bus reset signal, active low; read/write data and read/write address class signal width

are both 32bit; “READY” and “VALID” are corresponding channel handshake signals; “WSTRB” signal is 1 bit corresponding “WDATA” valid data byte, “WSTRB” width is 32bit/8=4bit; “BRESP” and “RRESP” are write response signals, read response signals, width is “2bit”, 'h0 stands for success, others are errors.

The read operation is dominated by the handshake from the read address channel and the address content is transferred, and then the handshake is read in the read data channel, and the response of the read content and the read operation is transmitted, and the rising edge of the clock is valid. Details as the picture below:



The write operation is dominated by the handshake from the write address channel and the address content is transferred, then the data channel is handshaked and the read content is transferred, and finally the response channel handshake is written, and the write response data is transmitted, and the rising edge of the clock is valid. Details as the picture below:



When we are not good at writing FPGA code , we need to learn from other people's code or use IP core. Here I found an AXI master code from github, the address is

[https://github.com/aquaxis/IPCORE/tree/master/aq\\_axi\\_vdma](https://github.com/aquaxis/IPCORE/tree/master/aq_axi_vdma)

This project is a VDMA written by itself, which contains a lot of code that can be referenced. I mainly use the “aq\_axi\_master.v” code for “AXI master” read and write operations. Learning from other people's code sometimes saves a lot of time, but if you can't understand it, it's hard to solve the problem. The “aq\_axi\_master.v” code is as follows, with some modifications.

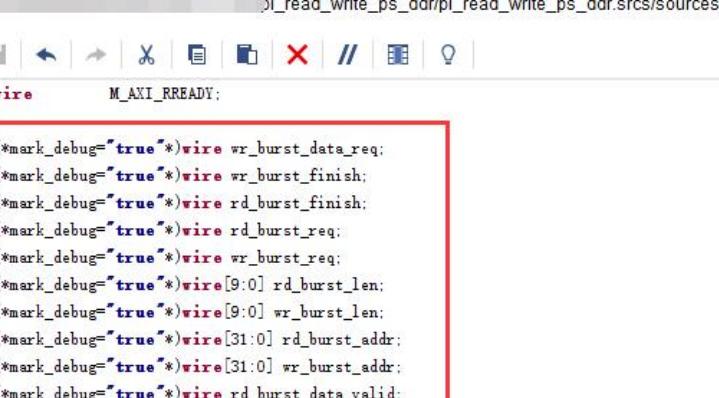
#### Part 19.4: Verification of ddr Read and Write Data

With the “AXI Master” read and write interface, it is easy to write a simple verification module. This verification module was used to verify the “ddr ip”. After each 8-bit write, the data is incremented and then read out for comparison. The important thing to note here is the starting address and size of the PS side DDR. Also, the unit of the address is byte or word, the address unit of the AXI bus is “byte”, and the address unit of the test module is “word” (the word here is not

necessarily 4 bytes). The file name “mem\_test.v”.

## Part 19.5: Vivado Software Debugging Skills

The AXI read-write verification module has only one error signal to indicate the error. If there is a data error, we hope to have more accurate information. There is a signal tap tool in altera's quartus II software, and a chipscope tool in xilinx's ISE. These are all embedded. The logic analyzer is very helpful to our debugging, and it is more convenient to debug in the vivado software. Some information may be optimized when inserting a debug signal, or the signal name may not be easily recognized when the signal name is changed. At this time, we can add the attribute `*mark_debug="true"` to the program code, as shown in the signal below:



```
Diagram x Address Editor x top.v x  
F:\pl_read_write_ps_ddr\pl_read_write_ps_ddr.sr...sources_1\rtl\top.v  
Q | H | ← | → | X | // | || |  
81 wire M_AXI_READY;  
82  
83 (*mark_debug="true")*wire wr_burst_data_req;  
84 (*mark_debug="true")*wire wr_burst_finish;  
85 (*mark_debug="true")*wire rd_burst_finish;  
86 (*mark_debug="true")*wire rd_burst_req;  
87 (*mark_debug="true")*wire wr_burst_req;  
88 (*mark_debug="true")*wire [9:0] rd_burst_len;  
89 (*mark_debug="true")*wire [9:0] wr_burst_len;  
90 (*mark_debug="true")*wire [31:0] rd_burst_addr;  
91 (*mark_debug="true")*wire [31:0] wr_burst_addr;  
92 (*mark_debug="true")*wire rd_burst_data_valid;  
93 (*mark_debug="true")*wire [63 : 0] rd_burst_data;  
94 (*mark_debug="true")*wire [63 : 0] wr_burst_data;
```

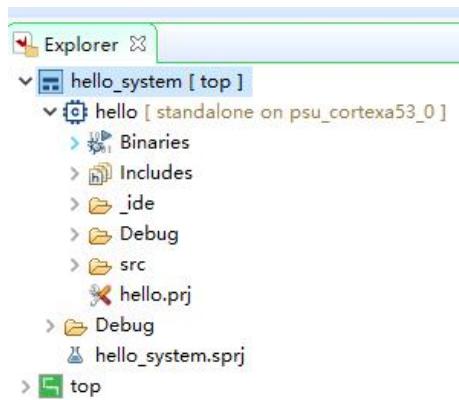
The specific adding method has been discussed in "PL's "Hello World" LED Experiment", you can refer to it.

And bind the error signal to the LED light on the PL side in the XDC file.

## Part 19.6: Vitis Program Development

Use hello world as a template to create a new vitis project as

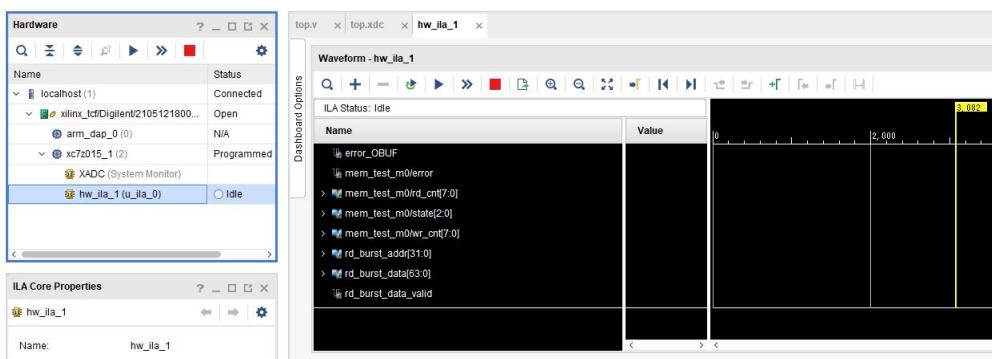
follows



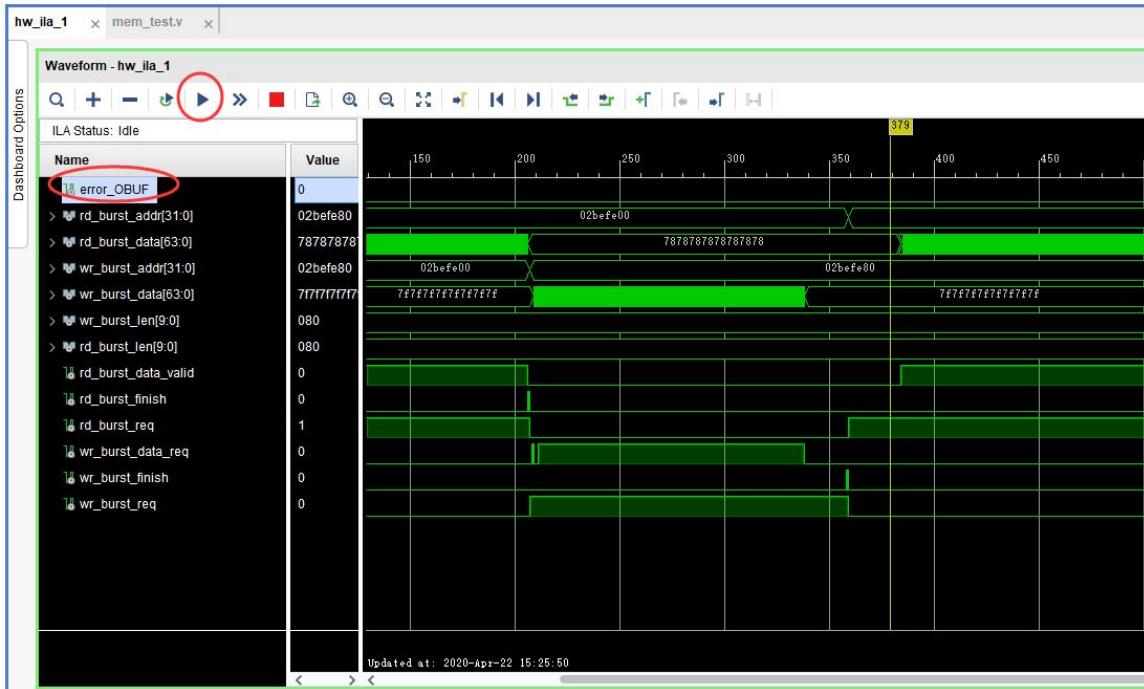
After downloading the program through vitis, the system will reset and download the bit file of FPGA. Then return to the vivado interface and click on the Program and Debug column to automatically connect to the target as shown in the figure below:



After the hardware is automatically connected, you can find the devices connected to JTAG, including a device named `hw_ila_1`, which is our debug device. After selecting it, you can click the upper yellow triangle button to capture the waveform. If some signals are not displayed completely, you can click the "+" button next to the waveform to add them.



After clicking the capture waveform, as shown in the figure below, if the error is always low and the read and write status changes, it means that the reading and writing of the DDR data is normal. The user can check other signals here to observe the data written to the DDR and read from the DDR. The data.



## Part 19.7: Experimental Summary

The zynq system is quite complex compared to a single FPGA or a single ARM. It requires a high level of basic knowledge for developers. This chapter covers the AXI protocol, zynq interconnect resources, vivado and sdk debugging techniques. These are just basic knowledge. Everyone still has to practice a lot and master the skills in constant practice

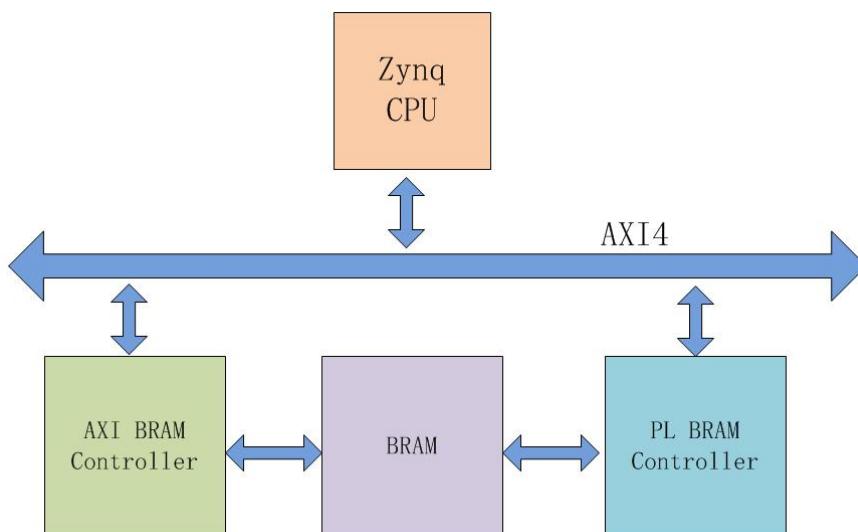
## Part 20: Realize PS and PL Data Interaction through BRAM

The experimental Vivado project directory is "bram\_test /vivado"

The experiment vitis project directory is "bram\_test /vitis"

Sometimes the "CPU" needs to exchange data in small batches with the "PL". This can be done through the "BRAM" module, which is the "Block RAM". This chapter reads and writes the "BRAM" of the PL side through the "GP Master" interface of Zynq, to realize the interaction with the PL. A custom FPGA program was added to this experiment and configured with the "AXI4" bus to inform it when to read and write "BRAM".

The following is the schematic diagram of the experiment. The "CPU" reads the "BRAM" data through the "AXI BRAM Controller". The "CPU" only configures the register of the custom "PL BRAM Controller", and does not read and write data through it.



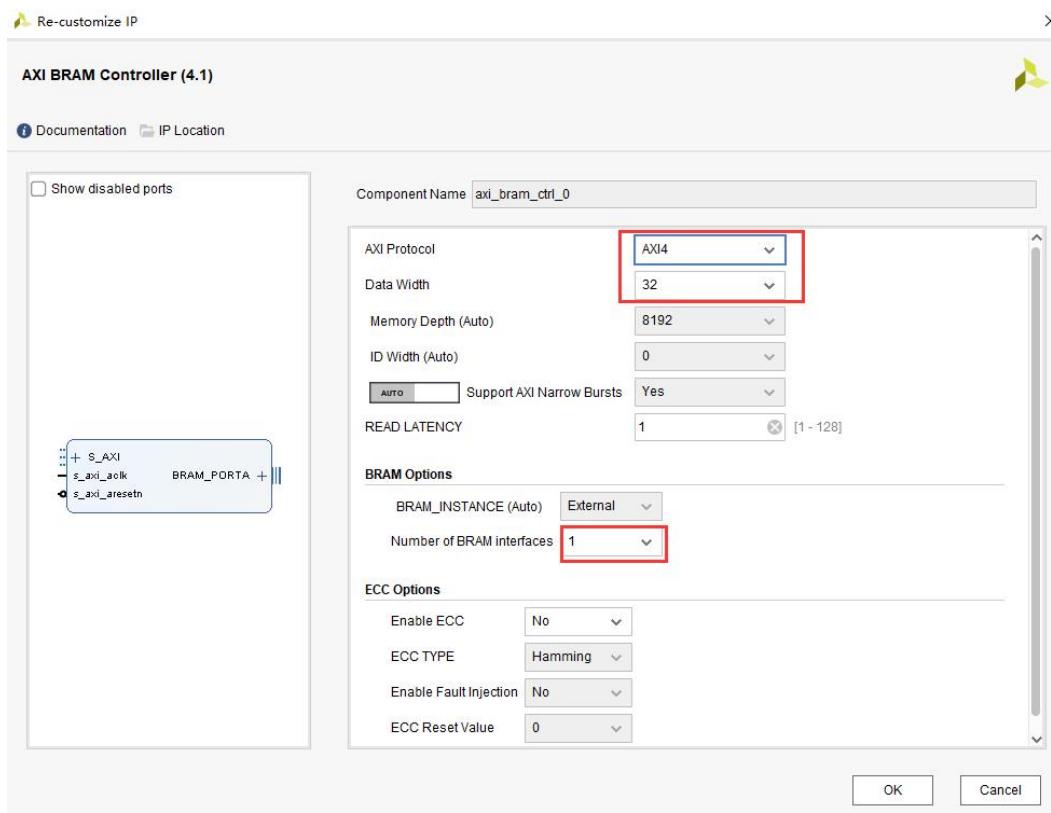
## FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 20.1: Hardware Environment

Based on "ps\_hello", save as a project, and configure the interrupt to open ZYNQ

- 1) First add the AXI BRAM Controller module to control the BRAM on the PS side. Double-click to open the configuration and connect to the AXI bus. It can be used to read and write the BRAM module. The AXI mode is set to AXI4, the data width is set to 32 bits, and the memory depth is not set here. Need to set in the Address Editor. The number of BRAM ports is set to 1, which is used to connect to PORTA of dual-port RAM. The “ECC” function is not enabled.



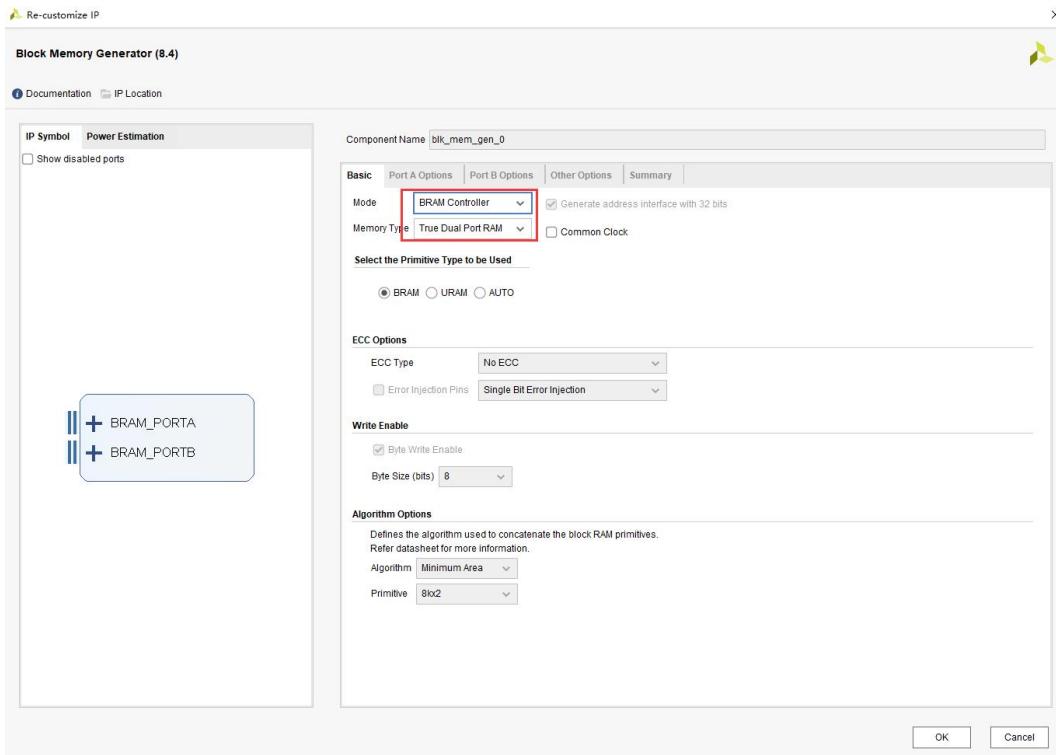
Since the “AXI4” bus is byte-addressed, the “BRAM” data width

setting is also “32” bits, which is also 32-bit data width. Therefore, when mapping to the BRAM address, it needs to be addressed by 4 bytes, that is, the last two bits are removed. The following figure shows the mapping relationship between BRAM controller and BRAM.

Table 3-1: BRAM Configuration for UltraScale, 7 Series, or Zynq-7000 Devices

Supported Memory Sizes / BRAM Memory Configuration	Number of BRAM Primitives (36k/each)	Size of BRAM_Addr (each port)	Typical BRAM_Addr Bit Usage with BRAM Configuration
32-bit Data BRAM Data Width			
4k / (1024 x 32)	1	10	BRAM_Addr [11:2]
8k / (2048 x 32)	2	11	BRAM_Addr [12:2]
16k / (4096 x 32)	4	12	BRAM_Addr [13:2]
32k / (8192 x 32)	8	13	BRAM_Addr [14:2]
64k / (16384 x 32)	16	14	BRAM_Addr [15:2]
128K / (32,768x32)	32	15	BRAM_Addr [16:2]

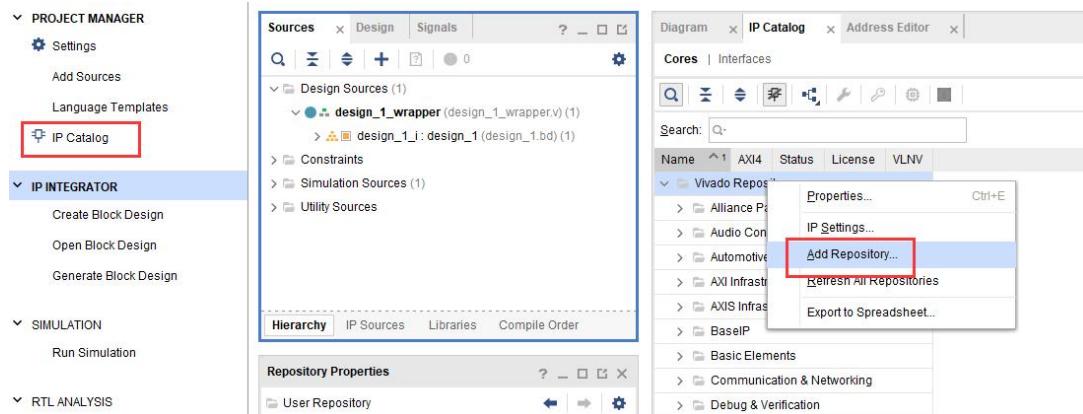
- 2) Add BRAM module, the BRAM settings are as follows, there are two mode options, “standalone” mode, which can freely configure the data width and depth of the RAM. “BRAM Controller” mode. In this mode, the address line and data port default to 32 bits. In this experiment, since the BRAM controller is used, the “BRAM Controller” mode is selected. The Memory type selects “dual-port” RAM with a “BRAM” controller on one end and a “PL RAM” controller on one end.



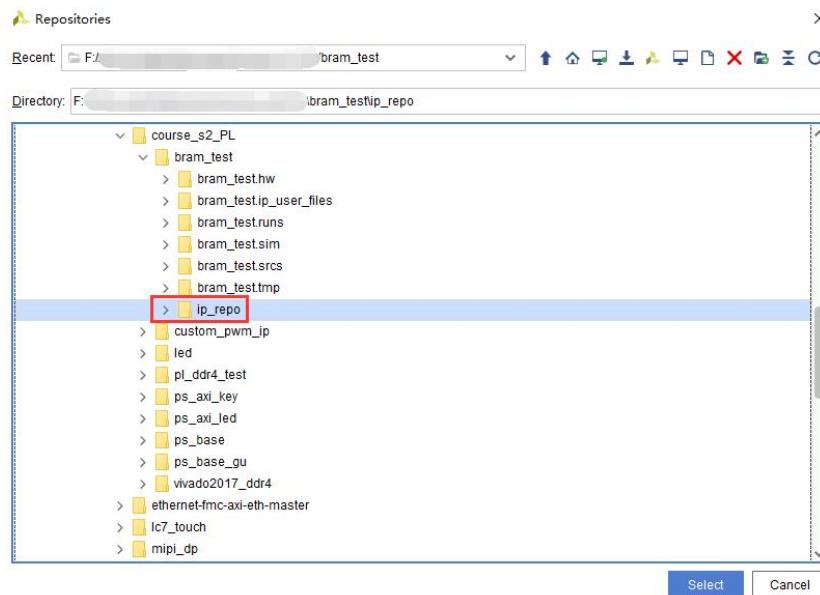
- 3) Add a custom “PL RAM” controller “pl\_ram\_ctrl”, the function is very simple, start the “BRAM” data after the “start” signal is valid, the read data can be observed by the “ILA” logic analyzer, and after the “PL RAM” controller reads the “BRAM”, it starts to write data to the BRAM. After writing the data enable “write\_end” signal, the “GPIO” generates an interrupt, and the “CPU” can read the “BRAM” data. Connect the “PL” controller signal to the “PORTB” of the “BRAM”. Custom IP in the “ip\_repo” folder

.Xil	2018/8/2 11:10	文件夹
bootimage	2018/8/1 15:24	文件夹
bram_intr.cache	2018/8/2 9:19	文件夹
bram_test.cache	2018/8/2 9:19	文件夹
bram_test.hw	2018/8/2 11:10	文件夹
bram_test.ip_user_files	2018/8/2 9:21	文件夹
bram_test.runs	2018/8/2 9:22	文件夹
bram_test.sdk	2018/8/2 11:08	文件夹
bram_test.sim	2018/8/2 9:19	文件夹
bram_test.srcs	2018/8/2 9:19	文件夹
ip_repo	2018/8/1 14:56	文件夹
bram_test.xpr	2018/8/2 9:27	Vivado Project Fi... 14 KB

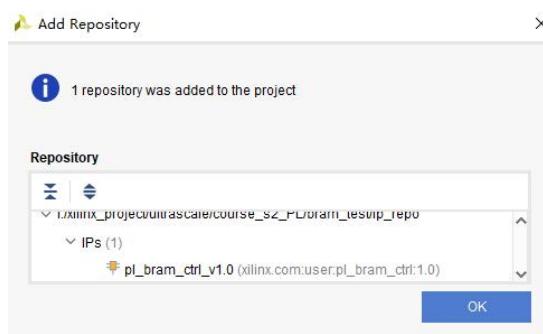
If you want to add a custom IP to IP library, click on the “IP Catalog”, right click the “Add Repository” in “Vivado Repository”



Find the folder where the custom IP is located and click Select



Pop up window, select IP, Click OK

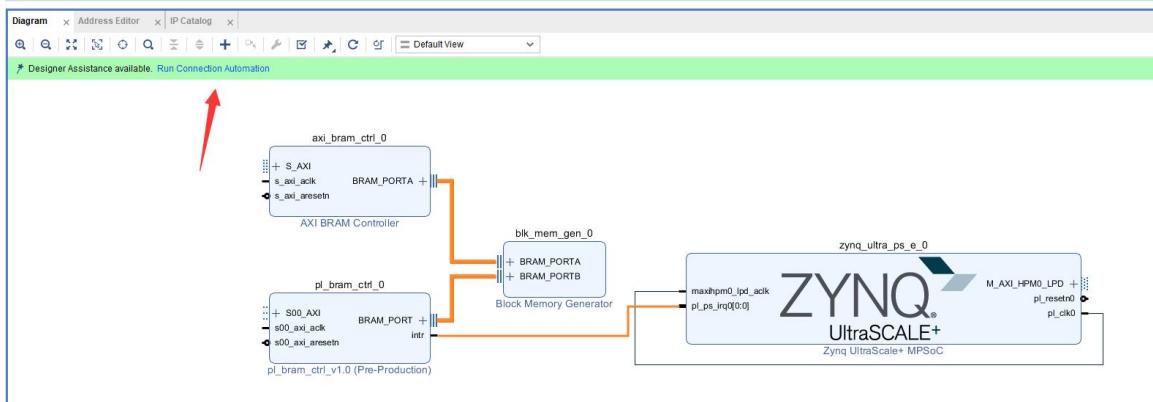


You can see that the newly added IP appears

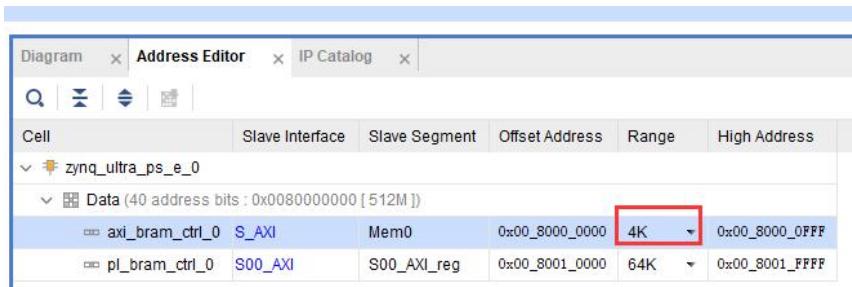


- 4) Connect BRAM\_PORTA of AXI BRAM Controller to PORTA of BRAM, and connect BRAM\_PORT of pl\_bram\_ctrl to PORTB of

BRAM. Connect the interrupt signal intr of the pl\_bram\_ctrl module to the interrupt port of ZYNQ. And click auto connect

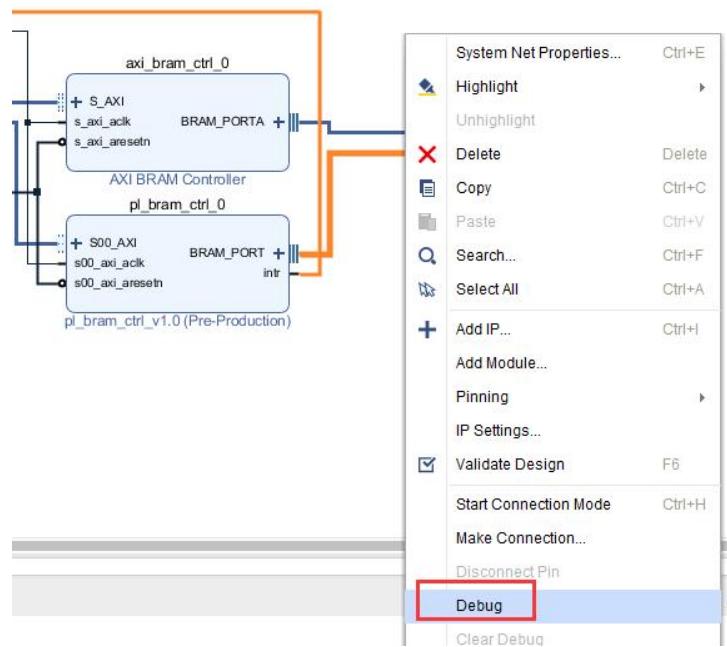


- 5) Select the BRAM address size in the Address Editor. If you set 4K, the address BRAM space is 1K deep.

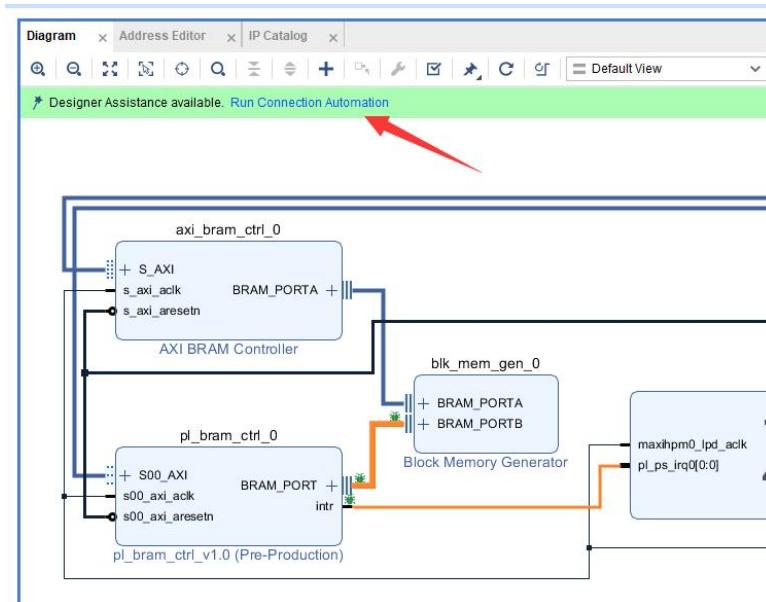


### Part 20.1.1: Block Design adds logic analyzer method

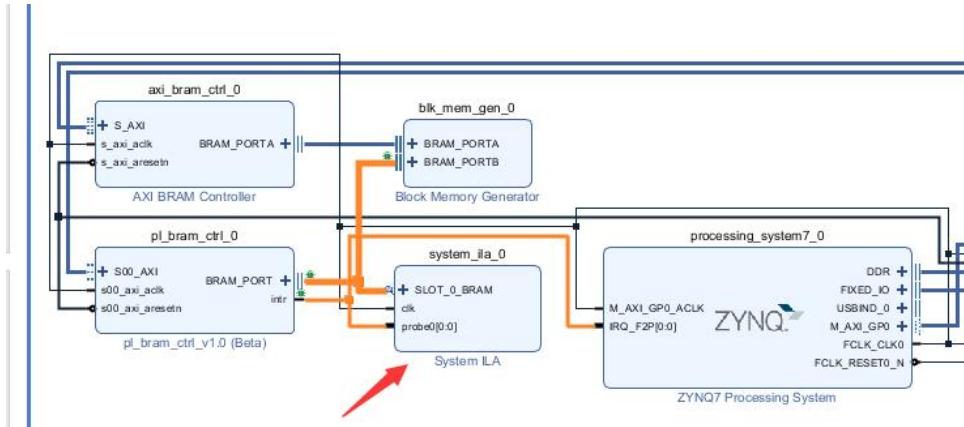
- 6) Then introduce a method to add a logic analyzer, select BRAM\_PORT bus and intr interrupt, right click and select Debug



- 7) You can see that there are more small insects on the bus, click Run Connection Automation to automatically connect



An ILA module is automatically added, and there is a bus interface and a signal interface



- 8) Save the design, and then click Generate Bitstream to generate the bit file and export the Hardware information.

▼ PROGRAM AND DEBUG

Generate Bitstream

## Part 20.2: Vitis Program Development

- 1) The programming flow is: input start address and length → CPU writes BRAM data through BRAM controller → informs PL controller to read BRAM data → PL internally reads and writes data to the same position, the initial data is informed by the CPU → Enable “write\_end” signal after writing, trigger “GPIO” interrupt → Interrupt reading BRAM data, print display
- 2) After entering the Vitis, create a new project under the Vitis, and the program is ready. The program is also relatively simple, the first is to interrupt settings

```
int main()
{
    int Status;
    Intr_flag = 1 ;
    IntrInitFuntion(INTC_DEVICE_ID) ;
    while(1)
    {
```

- 3) In the “While” statement, you need to enter the starting address and length, then call the “bram\_write” function.

```

while(1)
{
    if (Intr_flag)
    {
        Intr_flag = 0 ;
        printf("Please provide start address\t\n") ;
        scanf("%d", &Start_Addr) ;
        printf("Start address is %d\t\n", Start_Addr) ;
        printf("Please provide length\t\n") ;
        scanf("%d", &Len) ;
        printf("Length is %d\t\n", Len) ;
        Status = bram_read_write() ;
        if (Status != XST_SUCCESS)
        {
            xil_printf("Bram Test Failed!\r\n") ;
            xil_printf("*****\r\n");
            Intr_flag = 1 ;
        }
    }
}

```

- 4) In the “bram\_read\_write();” function, the data is first written by the BRAM controller. The initial value of the data is “TEST\_START\_VAL”, and then the “PL RAM” controller parameters are configured with length, start address, initial data, and start signal. And in the function to determine whether the test length exceeds the “BRAM” controller address range, if it exceeds, will report an error, you need to re-enter the address and length.

```

int bram_read_write()
{
    u32 Write_Data = TEST_START_VAL ;
    int i ;

    /*
     * if exceed BRAM address range, assert error
     */
    if ((Start_Addr + Len) > (BRAM_CTRL_HIGH - BRAM_CTRL_BASE + 1)/4)
    {
        xil_printf("*****\r\n");
        xil_printf("Error! Exceed Bram Control Address Range!\r\n");
        return XST_FAILURE ;
    }
    /*
     * Write data to BRAM
     */
    for(i = BRAM_BYTENUM*Start_Addr ; i < BRAM_BYTENUM*(Start_Addr + Len) ; i += BRAM_BYTENUM)
    {
        XBram_WriteReg(XPAR_BRAM_0_BASEADDR, i , Write_Data) ;
        Write_Data += 1 ;
    }
    //Set ram read and write length
    PL_RAM_CTRL_mWriteReg(PL_RAM_BASE, PL_RAM_LEN , BRAM_BYTENUM*Len) ;
    //Set ram start address
    PL_RAM_CTRL_mWriteReg(PL_RAM_BASE, PL_RAM_ST_ADDR , BRAM_BYTENUM*Start_Addr) ;
    //Set pl initial data
    PL_RAM_CTRL_mWriteReg(PL_RAM_BASE, PL_RAM_INIT_DATA , (Start_Addr+1)) ;
    //Set ram start signal
    PL_RAM_CTRL_mWriteReg(PL_RAM_BASE, PL_RAM_START , 1) ;

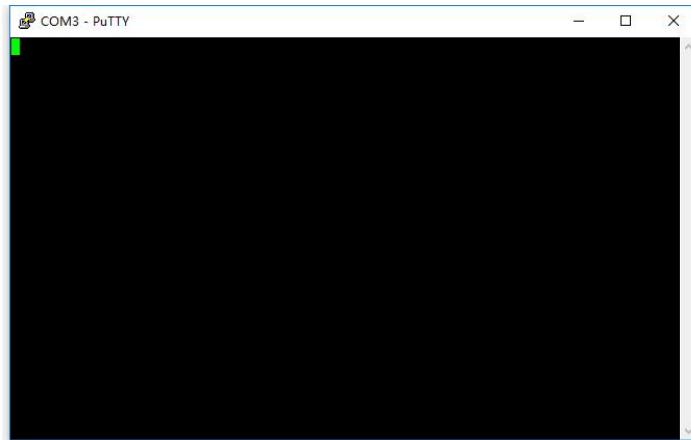
    return XST_SUCCESS ;
}

```

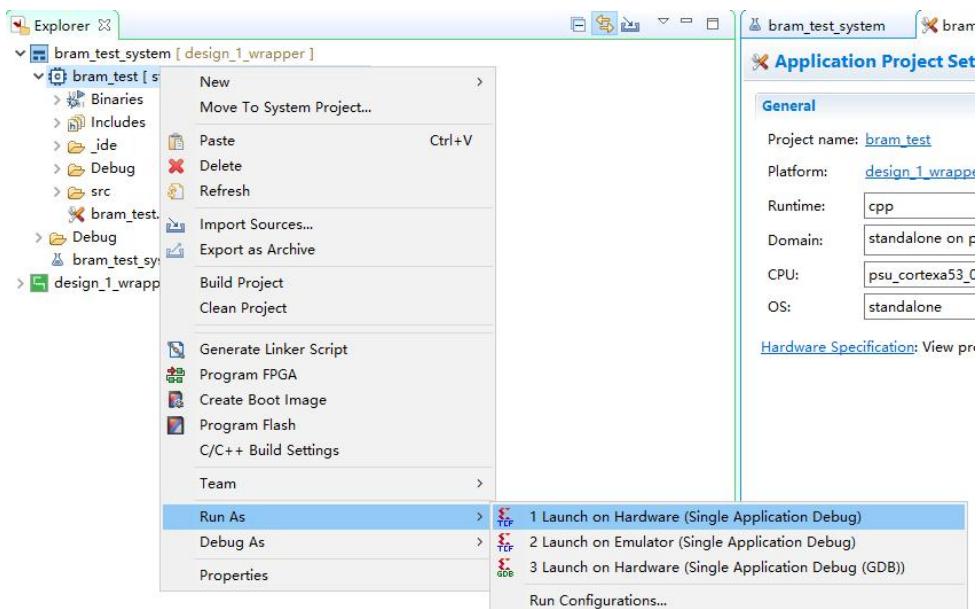
- 5) In the interrupt service routine, the “BRAM” controller reads the “BRAM” data and prints

## Part 20.3: Experimental Result

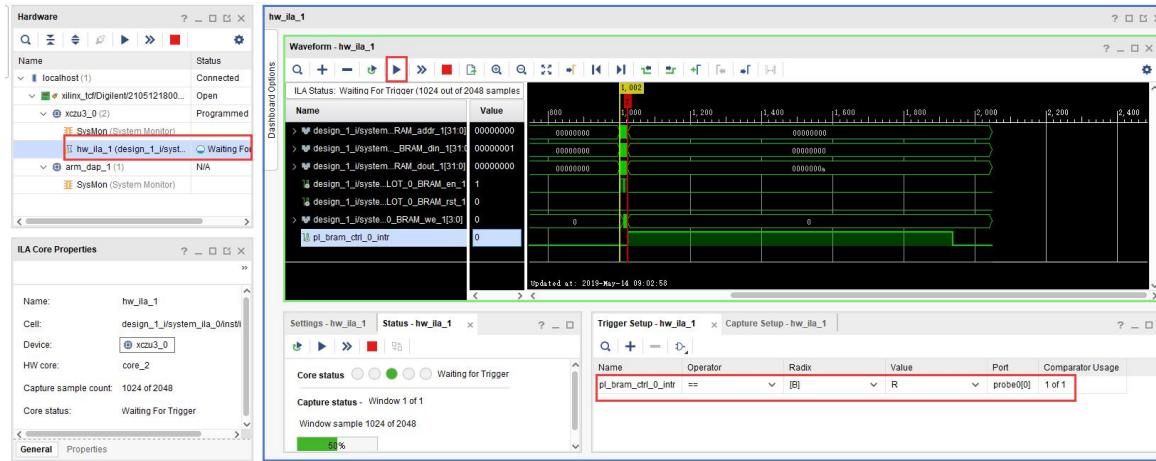
- 1) Open the serial port



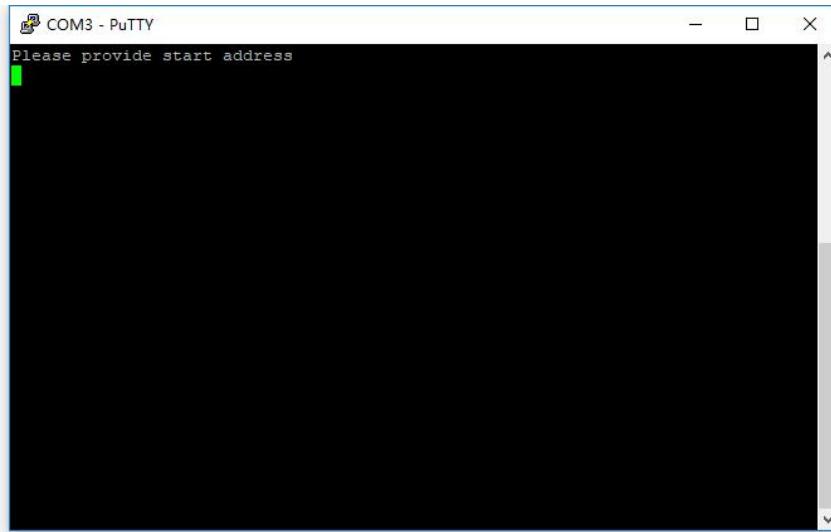
- 2) Download the program through Run Configurations



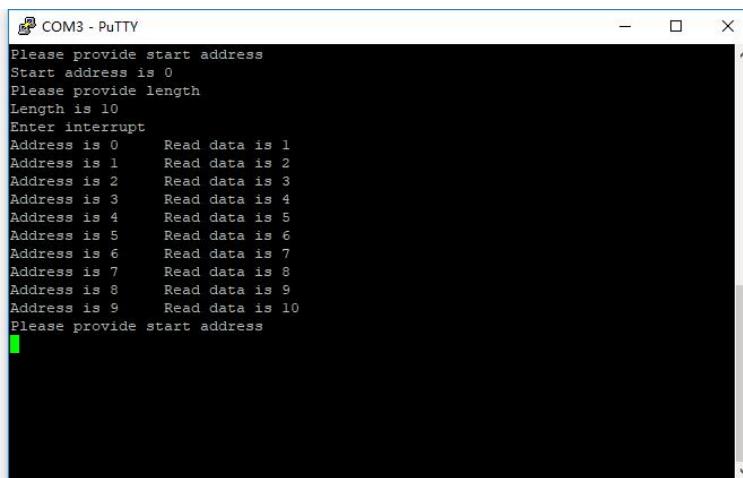
- 3) Open Vivado's Hardware Manager (joint debugging with PL), set the intr interrupt signal as the trigger signal, select the rising edge trigger, click the start button, you can see that hw\_il\_1 becomes the Waiting for trigger state



- 4) In the serial port software, enter the starting address. Since the BRAM query address is 1K, it can be set to 0~1023, and the length is set to 1~1024. Note that the starting address + length should not exceed 1024 because it exceeds the query address space.

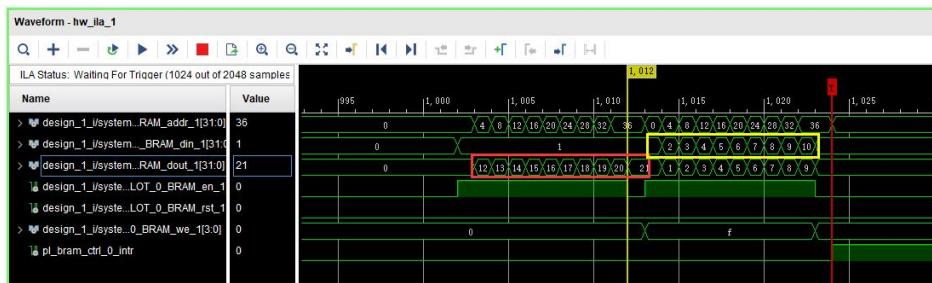


- 5) The input data is a decimal number, press Enter after the input

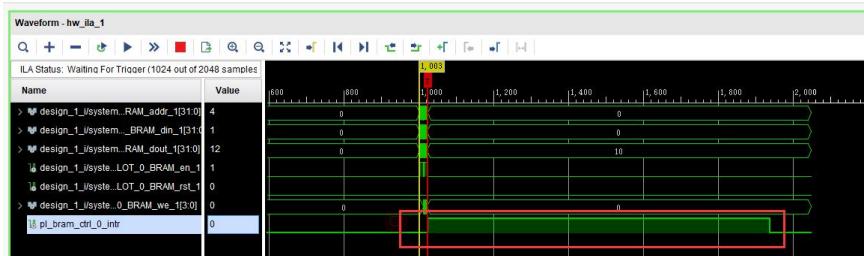


- 6) Open the ILA logic analyzer, you can see that it has been triggered.

First, the PL controller reads data from BRAM, and then writes data. You can see that the BRAM data read by PL in red is exactly the data written by the CPU, starting from 12. , A total of 10 data, the yellow part of the data written by the PL starts from 1, a total of 10 data, which is consistent with the data read by the CPU in the above figure.



You can also see the status of the interrupt signal



- 7) If it exceeds the range, print the error message, you need to re-enter valid information

```

COM3 - PuTTY
Start address is 0
Please provide length
Length is 10
Enter interrupt
Address is 0      Read data is 1
Address is 1      Read data is 2
Address is 2      Read data is 3
Address is 3      Read data is 4
Address is 4      Read data is 5
Address is 5      Read data is 6
Address is 6      Read data is 7
Address is 7      Read data is 8
Address is 8      Read data is 9
Address is 9      Read data is 10
Please provide start address
Start address is 0
Please provide length
Length is 1025
*****
Error! Exceed Bram Control Address Range!
Bram Test Failed!
*****
Please provide start address

```

## Part 20.4: Experimental Summary

The above is the experiment that “PS” and “PL” realize low-bandwidth data interaction through BRAM. The two are interconnected through GP port, which can realize small batch data interaction.

Knowledge points are the use of logic analyzers, the use of GPIO interrupts, custom IP, etc.

## Part 21: DMA Loop Test

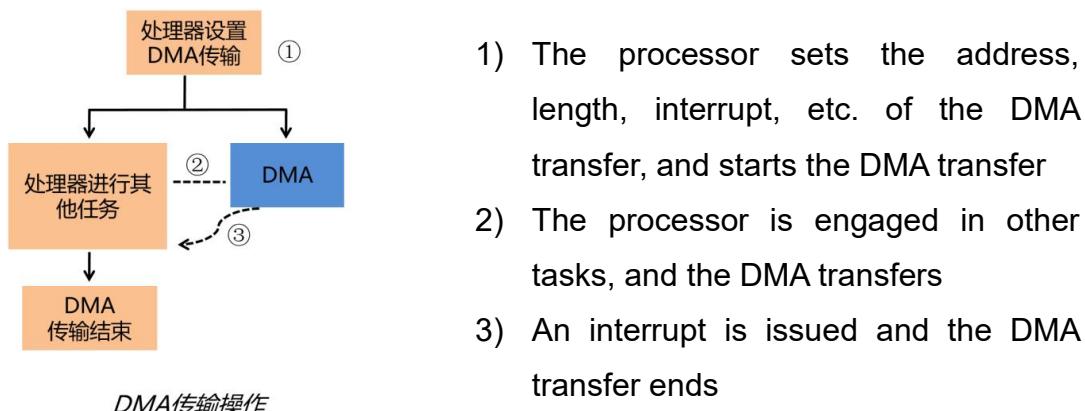
The experimental Vivado project directory is "dma\_loopback /vivado".

The experiment vitis project directory is "dma\_loopback /vitis".

This chapter introduces an important function module, DMA (Direct Memory Access), which refers to the interface technology that external devices do not exchange data directly with system memory through the CPU. To read peripheral data into memory or transfer memory to peripherals, it is usually done through CPU control, such as query or interrupt mode. Such as "BRAM" experiment as mentioned before.

Although the interrupt mode can improve the CPU utilization, it also has efficiency problems. For the case of bulk data transfer, the DMA method can solve the efficiency and speed problem. The CPU only needs to provide the address and length for the DMA, and the DMA can take over the bus, access memory, wait for the DMA to complete the work, tell the CPU, hand over the bus control.

In this chapter, the DMA example given by the SDK is slightly modified. After the DMA is finished, the end interrupt is issued, and the CPU is notified to the CPU to read the memory data.



## FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 21.1: Experimental Description

#### Refer to DMA document PG021

- 1) First come to know the “**AXI DMA**” module. This module uses three kinds of buses. “**AXI4-Lite**” is used to configure the registers. “**AXI4 Memory Map**” is used to interact with the memory. In this module, “**AXI4 Memory Map Read**” and “**AXI4 Memory Map Write**” are separately separated, which are called “**M\_AXI\_MM2S**” and “**M\_AXI\_S2MM**”. One is to read and another is to write, here to be clear, can not be confused. The “**AXI4 Stream**” interface is used to read and write peripherals. The “**AXI4 Stream Master (MM2S)**” is used to write to peripherals, and the “**AXI4-Stream Slave (S2MM)**” is used to read peripherals. The “**Scatter/Gather**” feature is also supported, but this experiment is not covered and is left for user research. (“**MM2S**” means “**Memory Map to Stream**”, “**S2MM**” means “**Stream to Memory Map**”)  
“**AXI Memory Map**” Data width support “32, 64, 128, 256, 512, 1024bits”  
“**AXI Stream**” Data width support “8, 16, 32, 64, 128, 256, 512, 1024bits”

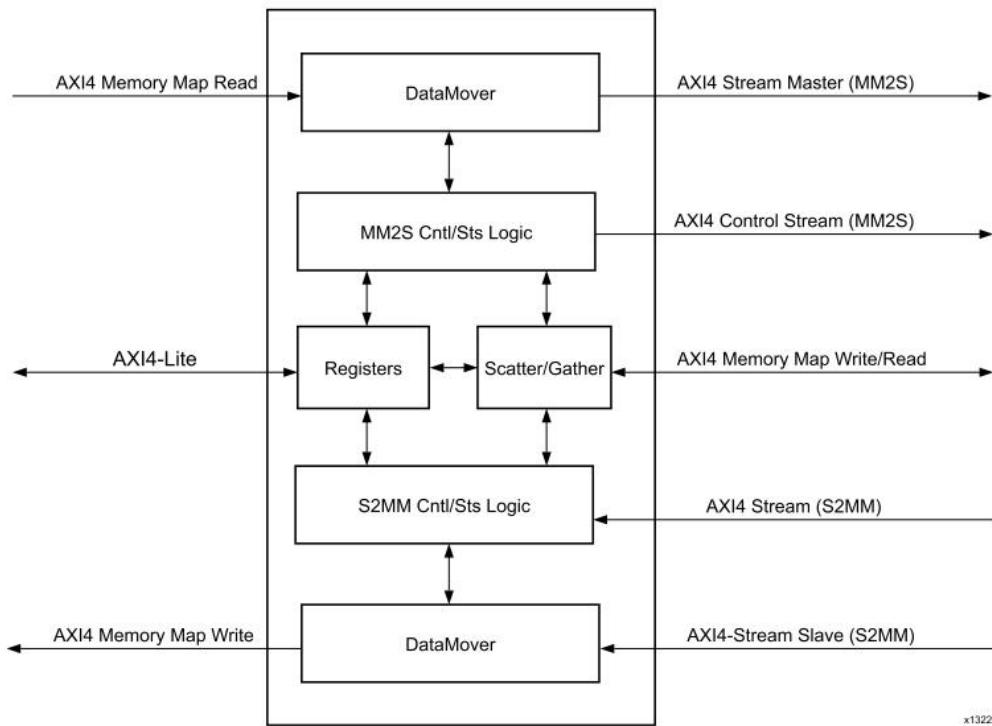


Figure 1-1: AXI DMA Block Diagram

- 2) In this experiment, the direct register mode is adopted. The following figure shows the register description. It is mainly divided into two parts. One is “MM2S”, including “Control Register”, “Status Register”, “Source Address”, “Transfer Length”. The second is “S2MM”, which also includes “Control Register”, “Status Register”, “Destination Address”, “Buffer Length” four parts, note that the “Source Address” and “Destination Address” here refers to the memory address.

Table 2-6: Direct Register Mode Register Address Map

Address Space Offset <sup>(1)</sup>	Name	Description
00h	MM2S_DMCR	MM2S DMA Control register
04h	MM2S_DMASR	MM2S DMA Status register
08h – 14h	Reserved	N/A
18h	MM2S_SA	MM2S Source Address. Lower 32 bits of address.
1Ch	MM2S_SA_MSB	MM2S Source Address. Upper 32 bits of address.
28h	MM2S_LENGTH	MM2S Transfer Length (Bytes)
30h	S2MM_DMCR	S2MM DMA Control register

Address Space Offset <sup>(1)</sup>	Name	Description
34h	S2MM_DMASR	S2MM DMA Status register
38h – 44h	Reserved	N/A
48h	S2MM_DA	S2MM Destination Address. Lower 32 bit address.
4Ch	S2MM_DA_MSB	S2MM Destination Address. Upper 32 bit address.
58h	S2MM_LENGTH	S2MM Buffer Length (Bytes)

- 3) The following is the description of the “MM2S\_DMACR” control register. The more important ones are “Bit0”, “Run/Stop”, which means that “DMA” is turned on or off. Other content is no longer described.

### **MM2S\_DMACR (MM2S DMA Control Register – Offset 00h)**

This register provides control for the Memory Map to Stream DMA Channel.

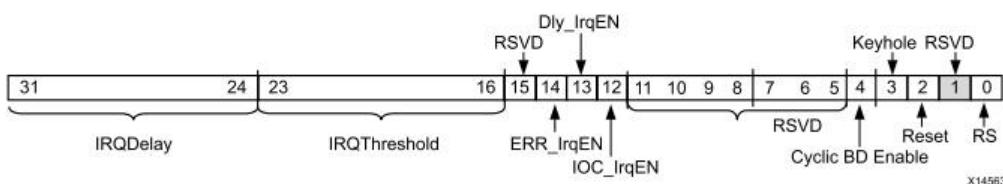


Figure 2-2: MM2S DMACR Register

Bits	Field Name	Default Value	Access Type	Description
0	RS	0	R/W	<p>Run / Stop control for controlling running and stopping of the DMA channel.</p> <ul style="list-style-type: none"> <li>0 = Stop – DMA stops when current (if any) DMA operations are complete. For Scatter / Gather Mode pending commands/transfers are flushed or completed. AXI4-Stream outs are potentially terminated early. Descriptors in the update queue are allowed to finish updating to remote memory before engine halt.</li> <li>1 = Run – Start DMA operations. The halted bit in the DMA Status register deasserts to 0 when the DMA engine begins operations.</li> </ul> <p>For Direct Register mode pending commands/transfers are flushed or completed. AXI4-Stream outs are potentially terminated.</p> <p>The halted bit in the DMA Status register asserts to 1 when the DMA engine is halted. This bit is cleared by AXI DMA hardware when an error occurs. The CPU can also choose to clear this bit to stop DMA operations.</p>

There are several interrupts that can be set, “IOC\_IrqEn”, enable interrupts, “Dly\_IrqEn” enable delayed interrupts, “Err\_IrqEn” enable error interrupts.

12	IOC_IrqEn	0	R/W	Interrupt on Complete (IOC) Interrupt Enable. When set to 1, allows DMASR.IOC_Irq to generate an interrupt out for descriptors with the IOC bit set. <ul style="list-style-type: none"><li>• 0 = IOC Interrupt disabled</li><li>• 1 = IOC Interrupt enabled</li></ul>
13	Dly_IrqEn	0	R/W	Interrupt on Delay Timer Interrupt Enable. When set to 1, allows DMASR.Dly_Irq to generate an interrupt out. <ul style="list-style-type: none"><li>• 0 = Delay Interrupt disabled</li><li>• 1 = Delay Interrupt enabled</li></ul> <b>Note:</b> This bit is ignored when AXI DMA is configured for Direct Register Mode.
14	Err_IrqEn	0	R/W	Interrupt on Error Interrupt Enable. <ul style="list-style-type: none"><li>• 0 = Error Interrupt disabled</li><li>• 1 = Error Interrupt enabled</li></ul>

- 4) “MM2S\_DMASR” is the status register description, “bits 12, 13, 14” are interrupt status, write “1” to clear the interrupt.

#### MM2S\_DMASR (MM2S DMA Status Register – Offset 04h)

This register provides the status for the Memory Map to Stream DMA Channel.

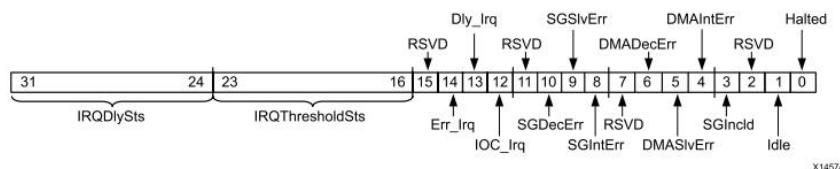
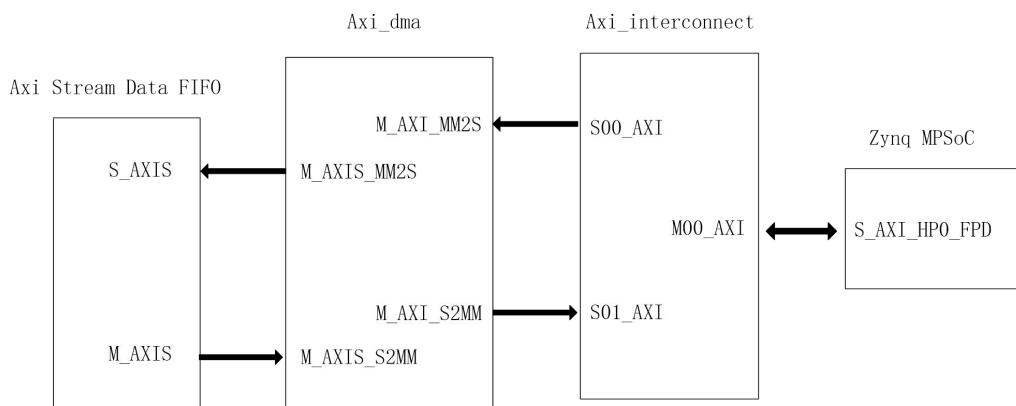


Figure 2-3: MM2S DMASR Register

12	IOC_Irq	0	R/WC	Interrupt on Complete. When set to 1 for Scatter/Gather Mode, indicates an interrupt event was generated on completion of a descriptor. This occurs for descriptors with the End of Frame (EOF) bit set. When set to 1 for Direct Register Mode, indicates an interrupt event was generated on completion of a transfer. If the corresponding bit is enabled in the MM2S_DMACR (IOC_IrqEn = 1) and if the interrupt threshold has been met, causes an interrupt out to be generated from the AXI DMA. <ul style="list-style-type: none"><li>• 0 = No IOC Interrupt.</li><li>• 1 = IOC Interrupt detected.</li></ul> Writing a 1 to this bit clears it.
13	Dly_Irq	0	R/WC	Interrupt on Delay. When set to 1, indicates an interrupt event was generated on delay timer timeout. If the corresponding bit is enabled in the MM2S_DMACR (Dly_IrqEn = 1), an interrupt out is generated from the AXI DMA. <ul style="list-style-type: none"><li>• 0 = No Delay Interrupt.</li><li>• 1 = Delay Interrupt detected.</li></ul> Writing a 1 to this bit clears it. <b>Note:</b> This bit is not used and is fixed at 0 when AXI DMA is configured for Direct Register Mode.
14	Err_Irq	0	R/WC	<ul style="list-style-type: none"><li>• Interrupt on Error. When set to 1, indicates an interrupt event was generated on error. If the corresponding bit is enabled in the MM2S_DMACR (Err_IrqEn = 1), an interrupt out is generated from the AXI DMA.</li></ul> Writing a 1 to this bit clears it. <ul style="list-style-type: none"><li>• 0 = No error Interrupt.</li><li>• 1 = Error interrupt detected.</li></ul>

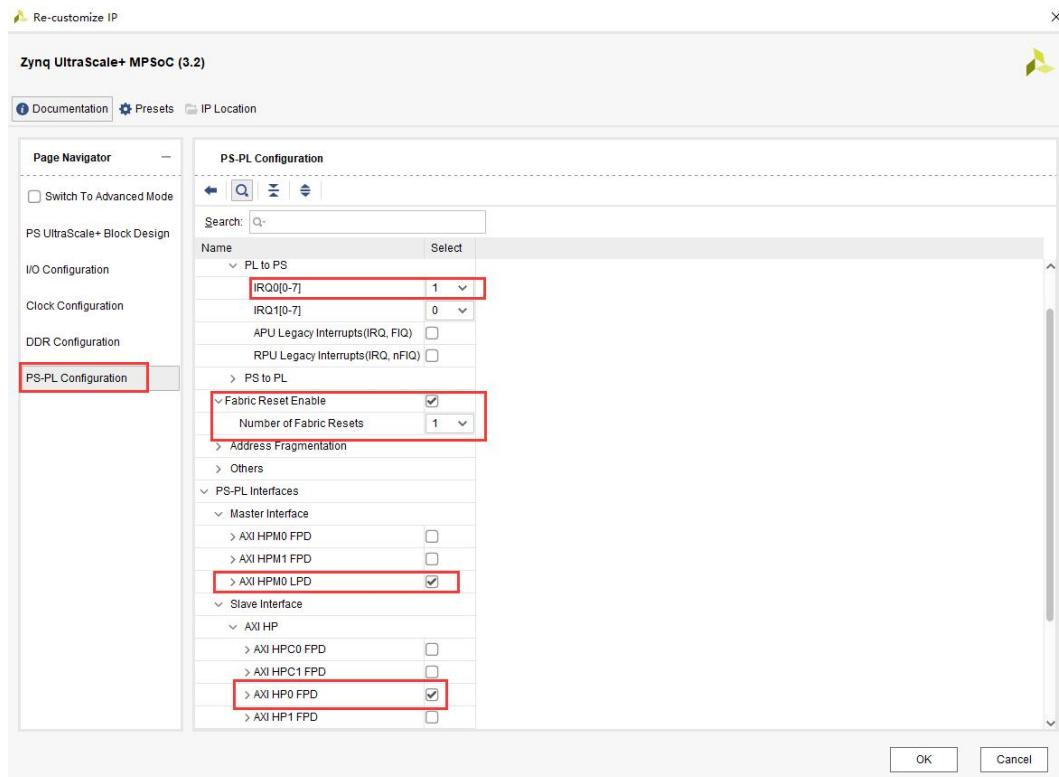
- 5) “MM2S\_DA”, “MM2S\_LENGTH” stands for address and length

setting. The register of “S2MM” is similar to “MM2S”. It is not described. The function of this experiment is that “MM2S” reads data from “DDR3”, writes to “AXI Stream Data FIFO”, then read from the “FIFO” to write to “DDR3”, to achieve loop-through test, you need to open the "IOC\_Irq" of “S2MM\_DMCR”, that is, write the memory end of the interrupt, the functional block diagram is as follows:

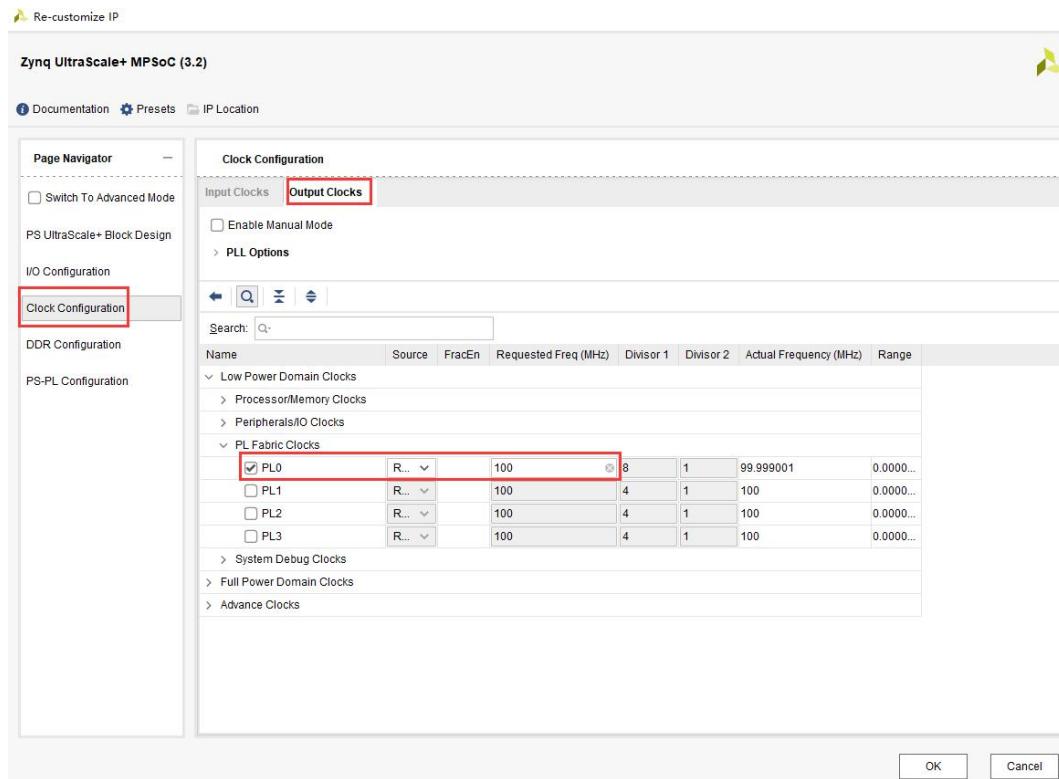


## Part 21.2: Hardware Environment

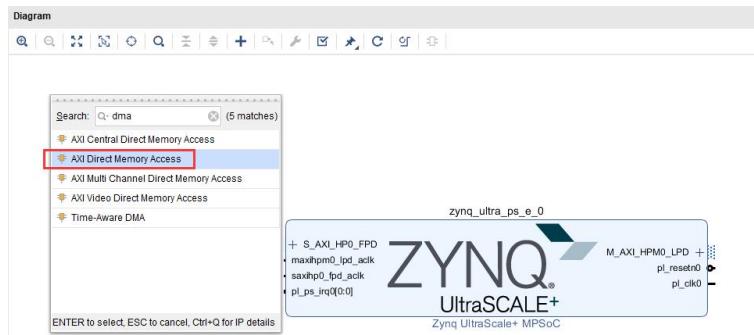
- 1) Based on the "ps\_base" project, save as the project "dma\_loopback", turn on reset, HPM0, HP0, PL to PS interrupt IRQ0



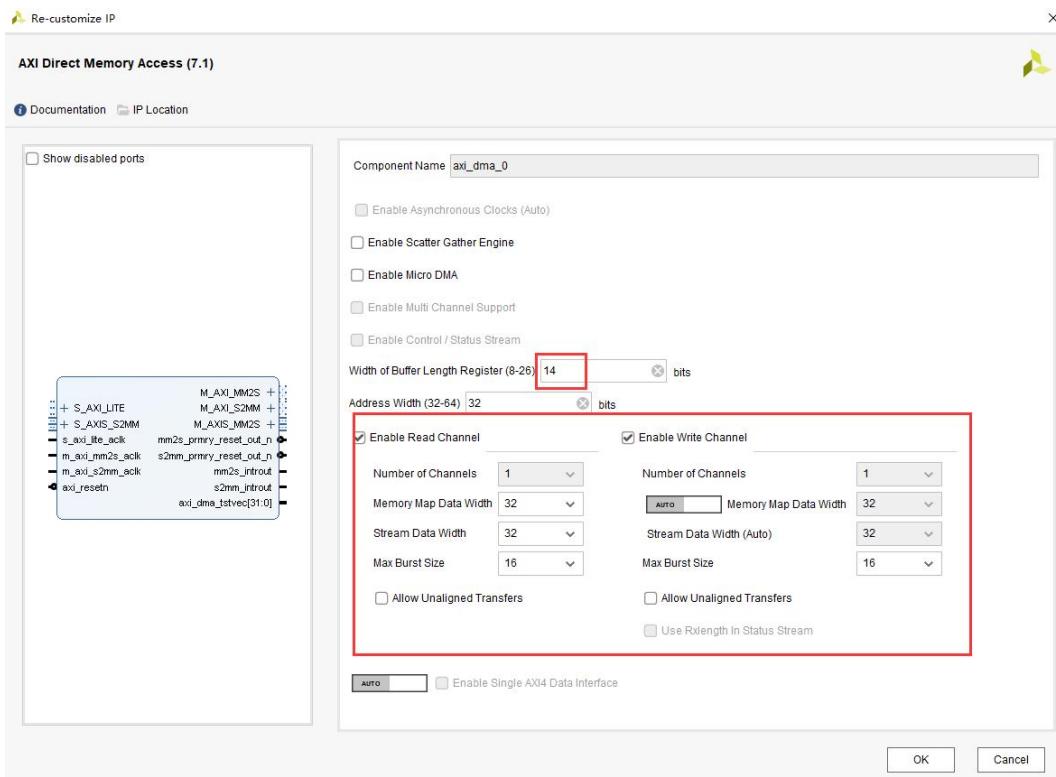
## 2) Set the clock to 100MHz



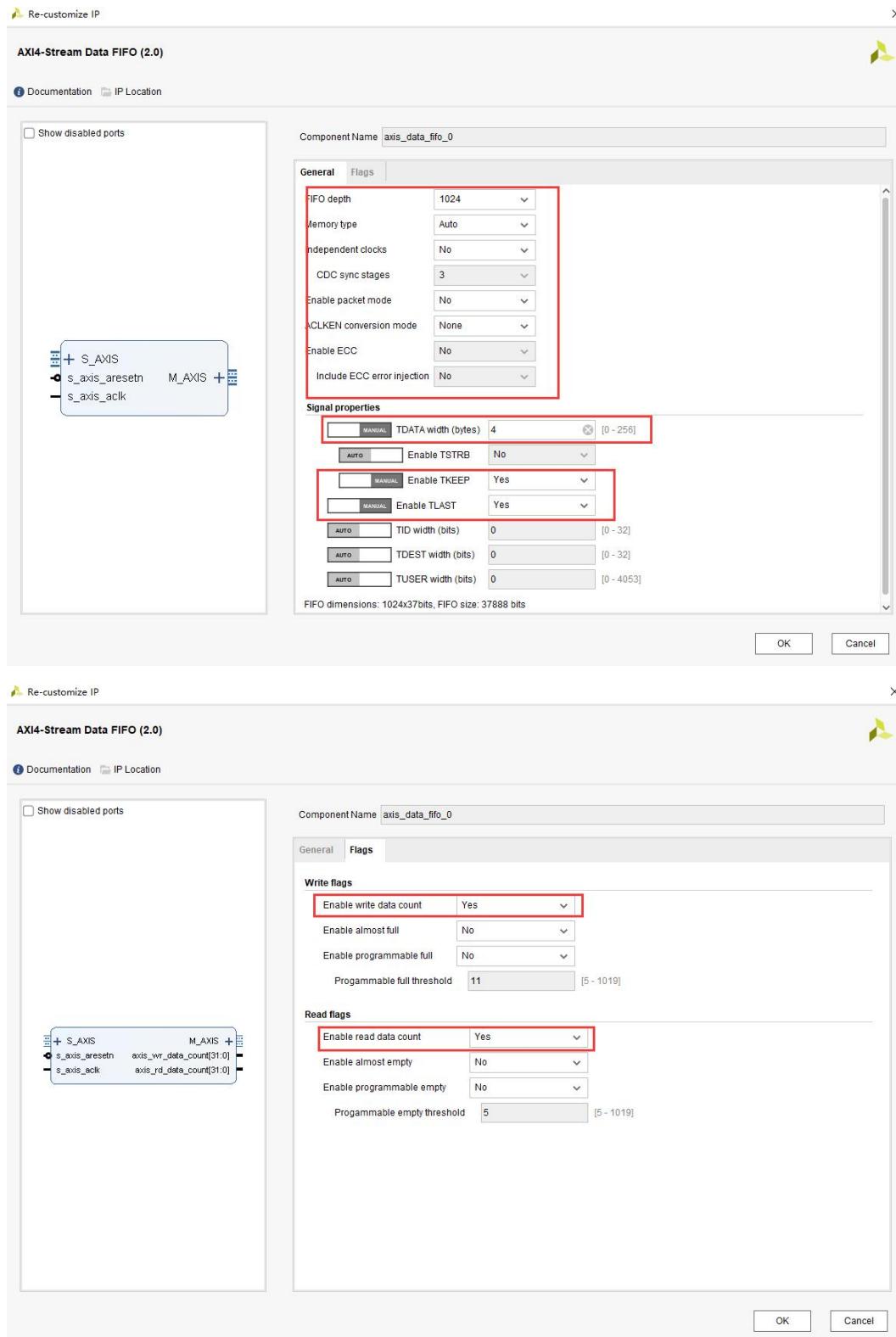
## 3) Click on "+" to add the “DMA” module.



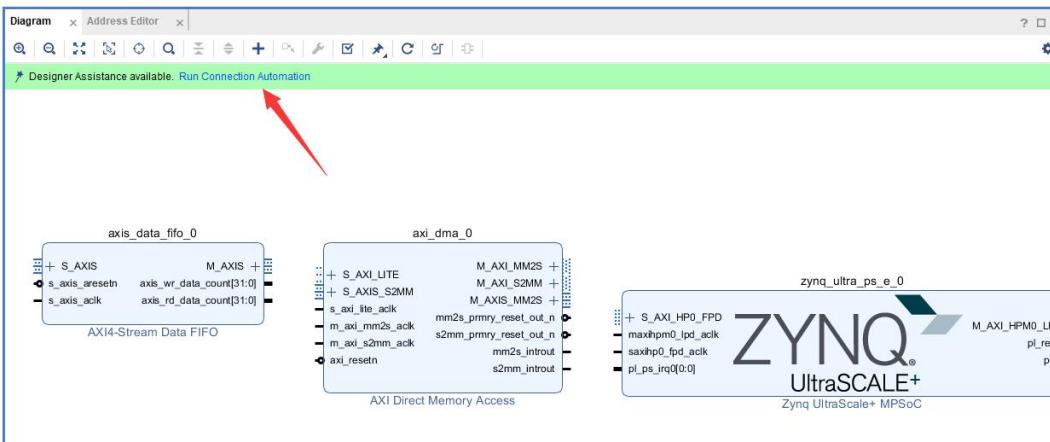
- 4) The “DMA” settings are as follows, “Width of Buffer Length Register” refers to the width of the “LENGTH” register, such as “26bits”, that is, the maximum transmission  $2^{26}=67,108,864\text{bytes}$ . Here, according to the default setting of “14”, open the read and write channels, do not open “Allow Unaligned Transfers”, if not open, “Memory Map Data Width” is set to “32”, then the address must be “0x0”, “0x4”, “0x8”, “0xC”, etc., aligned by 4 bytes. The “Max Burst Size” can be set to “2, 4, 8, 16, 32, 64, 128, 256”.



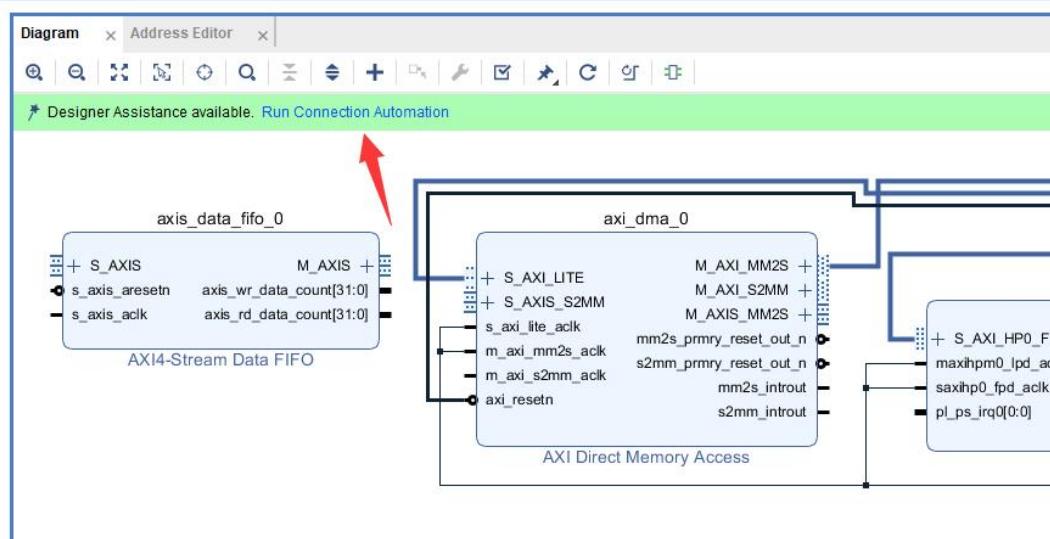
- 5) The “AXI Stream Data FIFO” is set as follows, setting the depth to “1024”, “TDATA Width” to 4 bytes, turning on “TKEEP”, “TLAST”



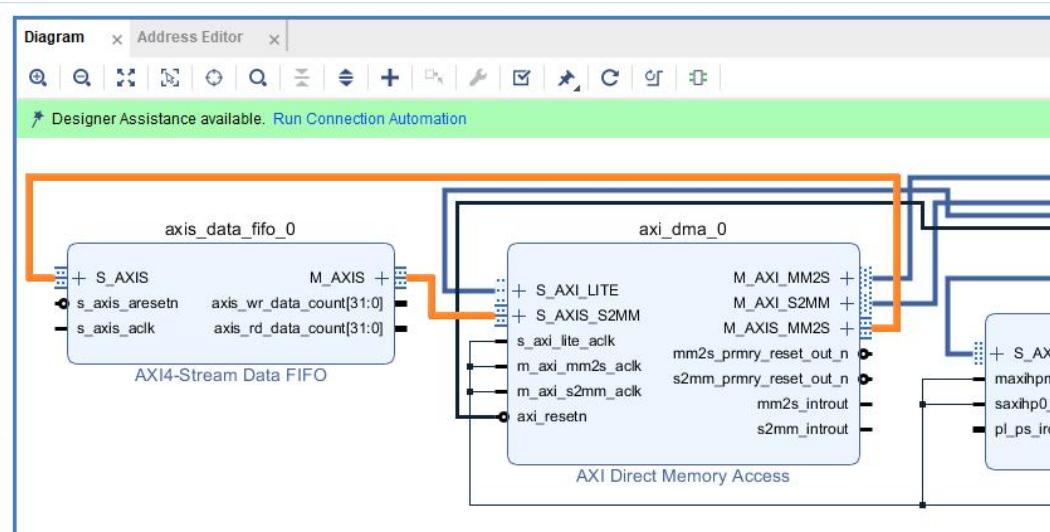
## 6) Auto connect



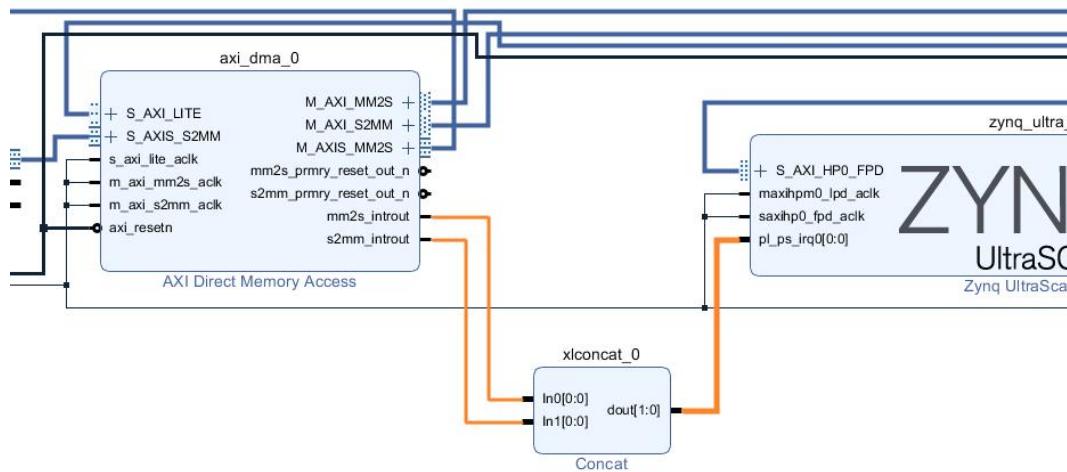
Continue to connect automatically



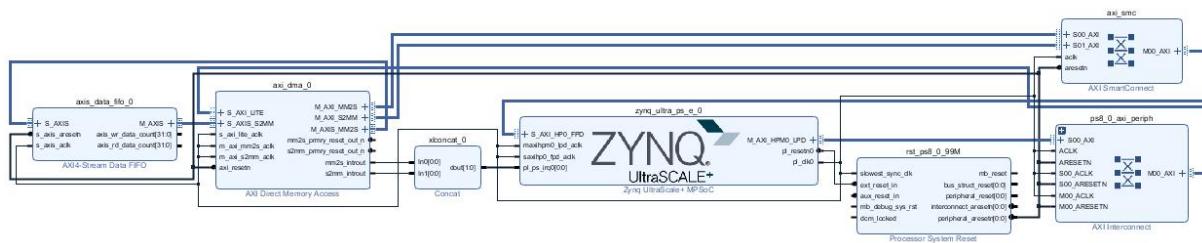
- 7) Connect **S\_AXIS** and **M\_AXIS** of **FIFO** to **dma**, Continue to click **Run Connection Automation**



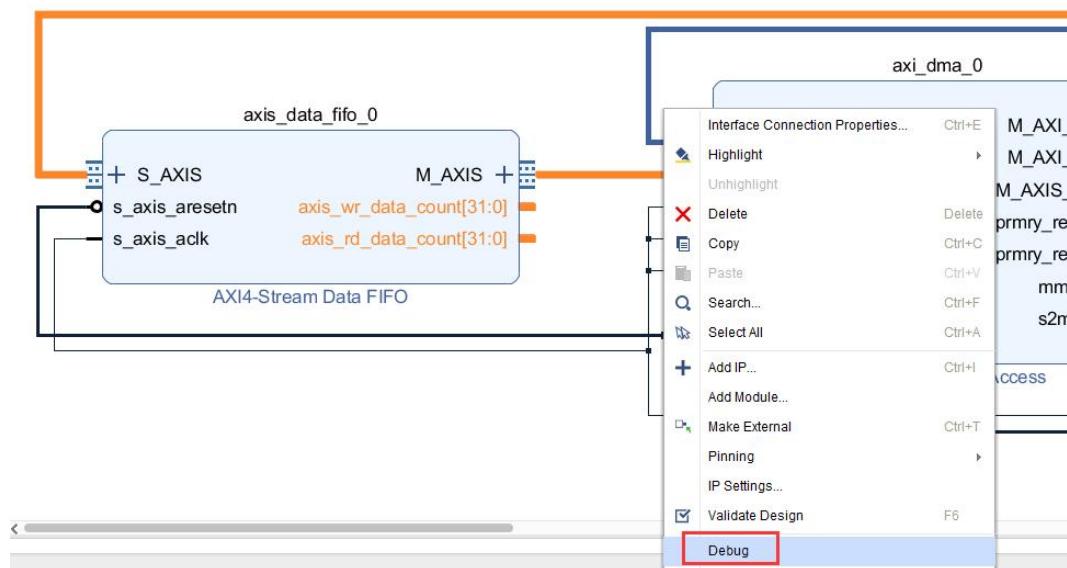
- 8) Add “Concat”, connect “MM2S” and “S2MM” interrupt output to “pl\_ps\_irq”



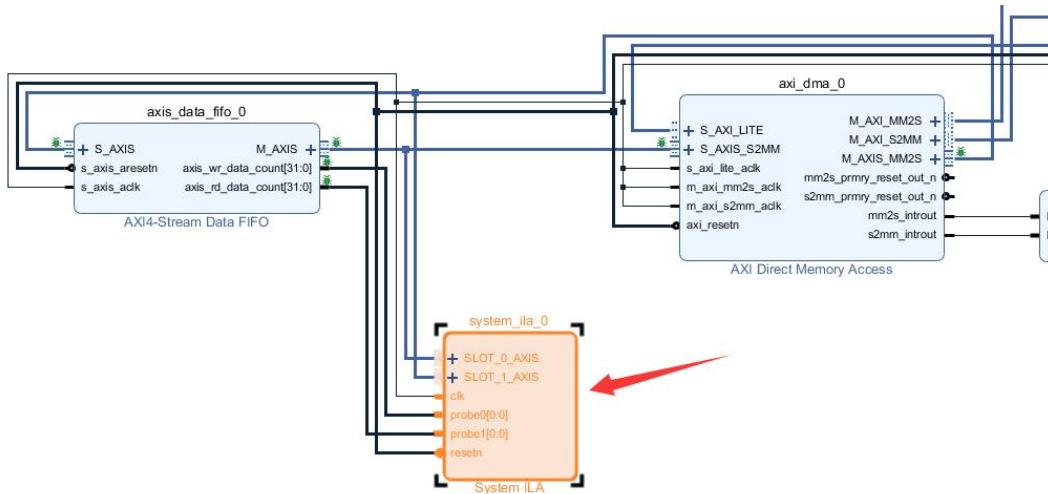
- 9) The final connection is shown below



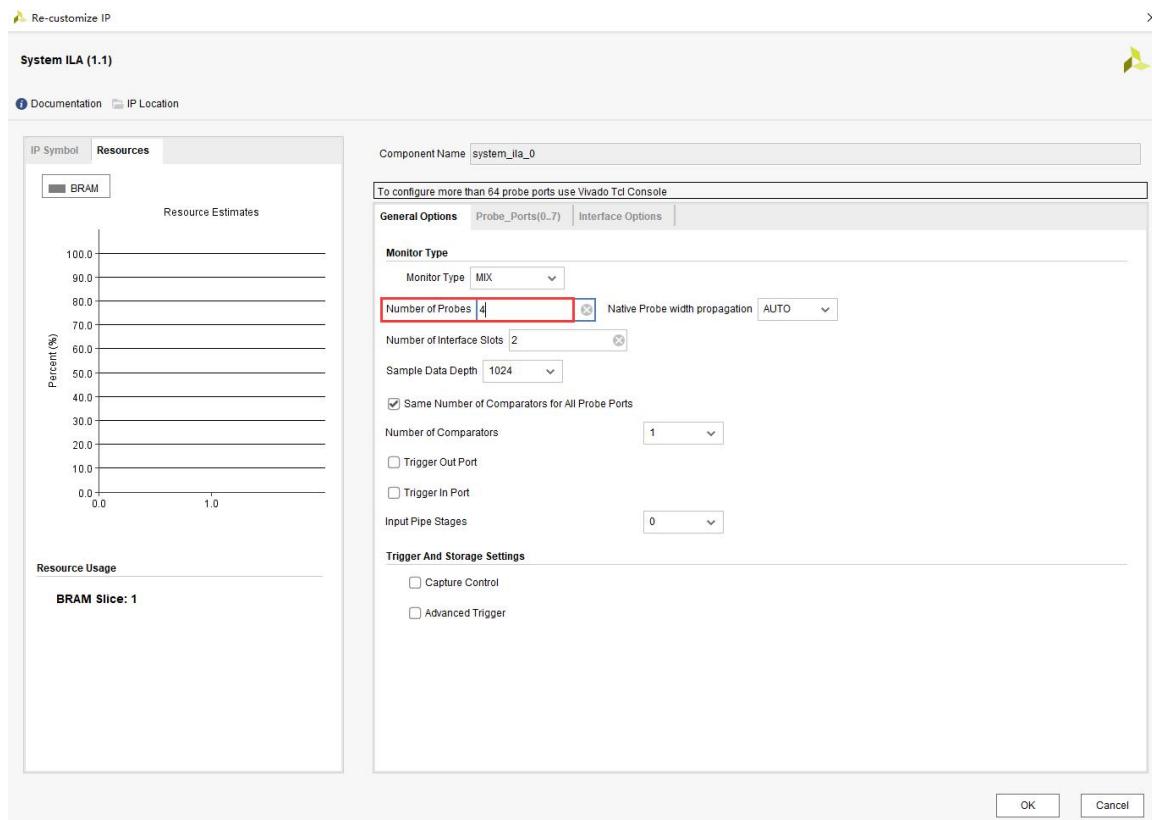
- 10) Select “**S\_AXI**”, “**M\_AXI**”, “**count**” signals of fifo, right-click to select “**Debug**”, add **ILA** logic analyzer, and observe the data changes.



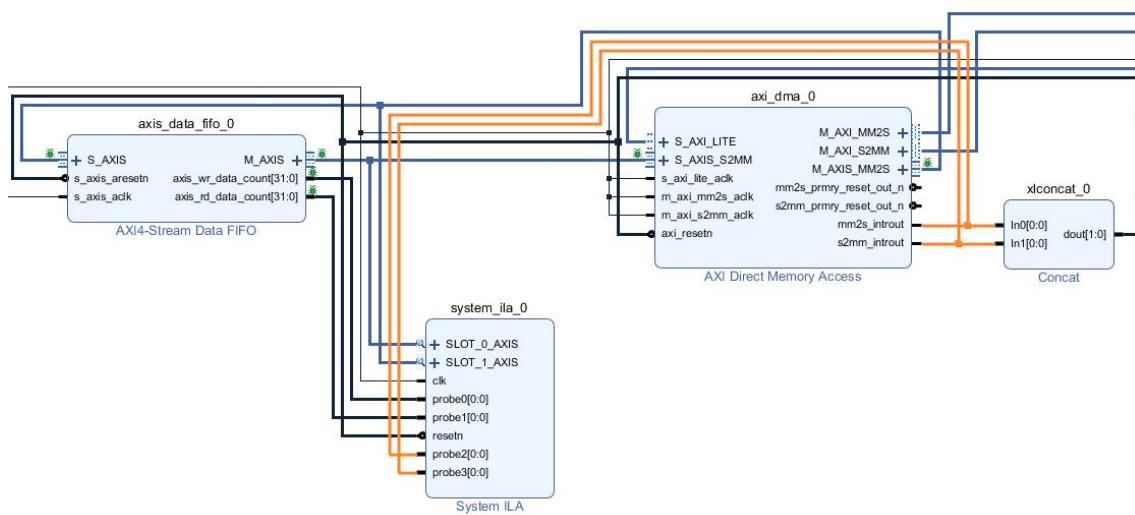
11) After automatic connection, open **ila** configuration



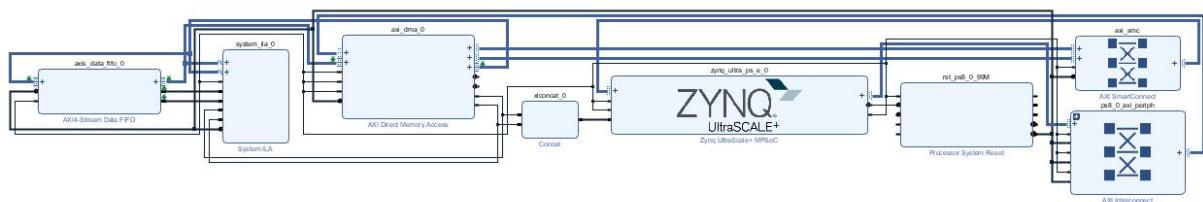
Change the Number of Probes to 4 and add two **Probe** interfaces



Connect the two newly added **Probes** to the interrupt output of **DMA**



The final connection result is as follows



## 12) Save the design and generate bitstream

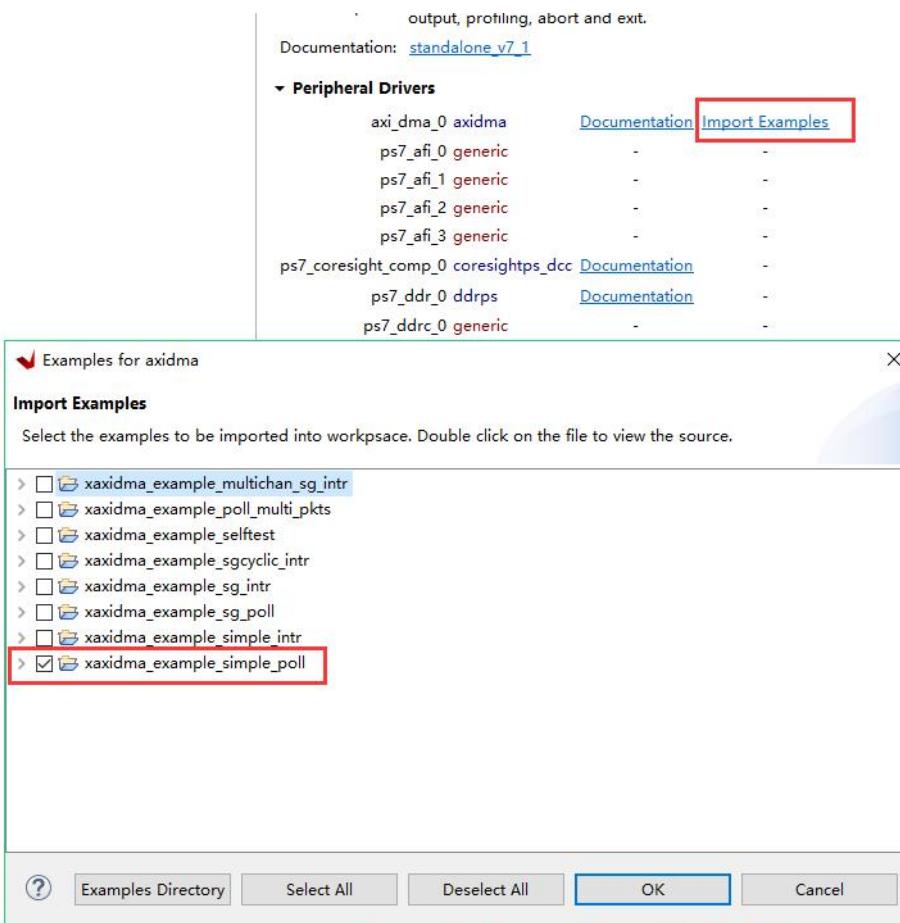


## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 21.3: Vitis program development

- 1) This experimental program is modified according to the **“simple\_poll”** example. In the BSP, you can learn the application of the module by importing examples.



- 2) Set “**MAX\_PKT\_LEN**”, which is the length in bytes, “**TEST\_START\_VALUE**” is the starting data value, and “**NUMBER\_OF\_TRANSFERS**” is the number of tests.

```
#define FIFO_DATABYTE    4
#define TEST_COUNT        80
#define MAX_PKT_LEN       TEST_COUNT*FIFO_DATABYTE

#define TEST_START_VALUE   0xC
#define NUMBER_OF_TRANSFERS 2
```

- 3) Define transmit and receive arrays

```
u8 TxBufferPtr[MAX_PKT_LEN] ;
u8 RxBufferPtr[MAX_PKT_LEN] ;
```

- 4) In the “**XAxiDma\_Setup**” function, open the “**IOC**” interrupt of the “**S2MM**” and close all interrupts of the “**MM2S**”. An interrupt is issued after the “**S2MM**” receives the data.

```
/* Disable MM2S interrupt, Enable S2MM interrupt */
XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_IOC_MASK,
                    XAXIDMA_DEVICE_TO_DMA);
XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
                     XAXIDMA_DMA_TO_DEVICE);
```

- 5) In the “**XAxiDma\_Setup**” function, after the “**TxBufferPtr**” is initialized, the data in the “**Cache**” needs to be flushed into the memory. This is very important. Because the “**DMA**” needs to access “**DDR3**”, and the “**CPU**” and “**DDR3**” are interacted through the “**Cache**”. The data is temporarily stored in the “**Cache**”, and may not actually be refreshed to “**DDR3**”. If the external device, that is, the “**DMA**” wants to read the value of “**DDR3**”, the data in the “**Cache**” must be flushed to “**DDR3**”, so that the “**DMA**” can read the correct value. Calling the “**Xil\_DCacheFlushRang**” function requires a memory address and length.

```
Value = TEST_START_VALUE;

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((UINTPTR)TxBufferPtr, MAX_PKT_LEN);
```

- 6) Open “**MM2S**” path and “**S2MM**” path

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

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

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

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

while ((XAxiDma_Busy(&AxiDma,XAXIDMA_DEVICE_TO_DMA)) ||
       (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE)))
{
    /* Wait */
}
```

- 7) The interrupt setting method is the same as the previous routine

```

int SetInterruptInit(XScuGic *InstancePtr, u16 IntrID, XAxiDma *XAxiDmaPtr)
{
    XScuGic_Config * Config ;
    int Status ;

    Config = XScuGic_LookupConfig(INT_DEVICE_ID) ;

    Status = XScuGic_CfgInitialize(&INST, Config, Config->CpuBaseAddress) ;
    if (Status != XST_SUCCESS)
        return XST_FAILURE ;

    Status = XScuGic_Connect(InstancePtr, IntrID,
                           (Xil_ExceptionHandler)CheckData,
                           XAxiDmaPtr) ;

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

    XScuGic_Enable(InstancePtr, IntrID) ;

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler) XScuGic_InterruptHandler,
                                InstancePtr);

    Xil_ExceptionEnable();

    return XST_SUCCESS ;
}

```

- 8) In the interrupt service program, the interrupt is first cleared. Since the data in “DDR3” has been updated, but the data in the “Cache” is not updated, if the “CPU” wants to read data from “DDR3”, it needs to call the “Xil\_DCachelnvalidateRang” function to invalidate the “Cache” data, so that the “CPU” Can read the correct data from “DDR3”. Also give the memory address and length.

```

xil_printf("Enter Interrupt\r\n");
/*Clear Interrupt*/
XAxiDma_IntrAckIRQ(&AxiDma, XAXIDMA_IRQ_IOC_MASK,
                    XAXIDMA_DEVICE_TO_DMA) ;
/* Invalidate the DestBuffer before receiving the data, in case the
   * Data Cache is enabled
   */
Xil_DCachelnvalidateRange((UINTPTR)RxPacket, MAX_PKT_LEN);

```

- 9) The **CPU** then reads the data from **DDR3** for comparison and verifies the correctness of the data

```

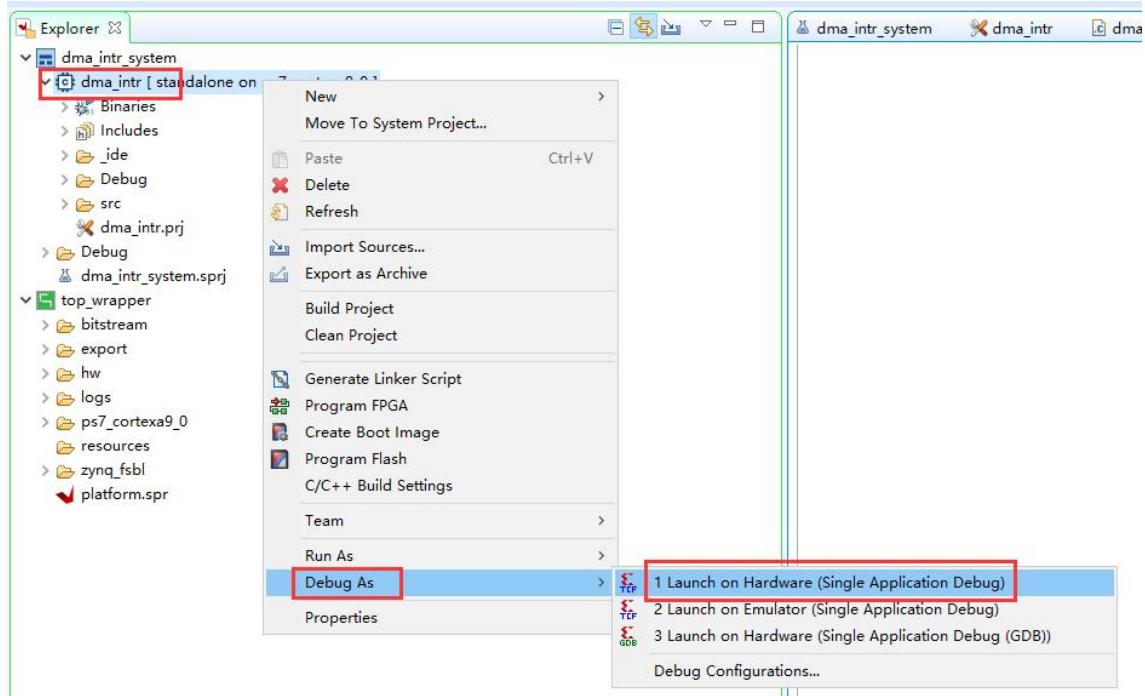
for(Index = 0; Index < MAX_PKT_LEN; Index++) {
    if (RxPacket[Index] != Value) {
        xil_printf("Data error %d: %x/%x\r\n",
                   Index, (unsigned int)RxPacket[Index],
                   (unsigned int)Value);

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

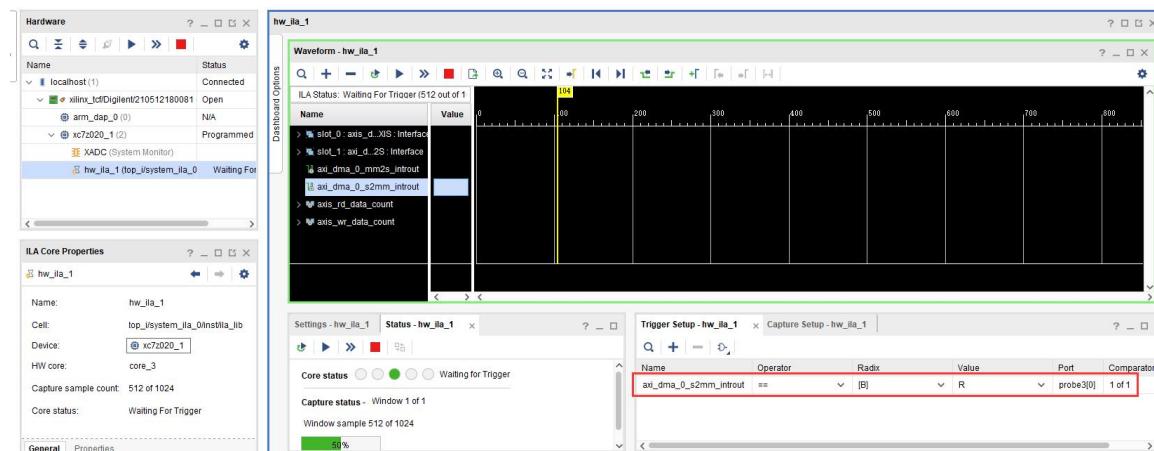
```

## Part 21.4: Program verification

- 1) Select “Debug Configurations”, use “Debug” mode, click “Debug”



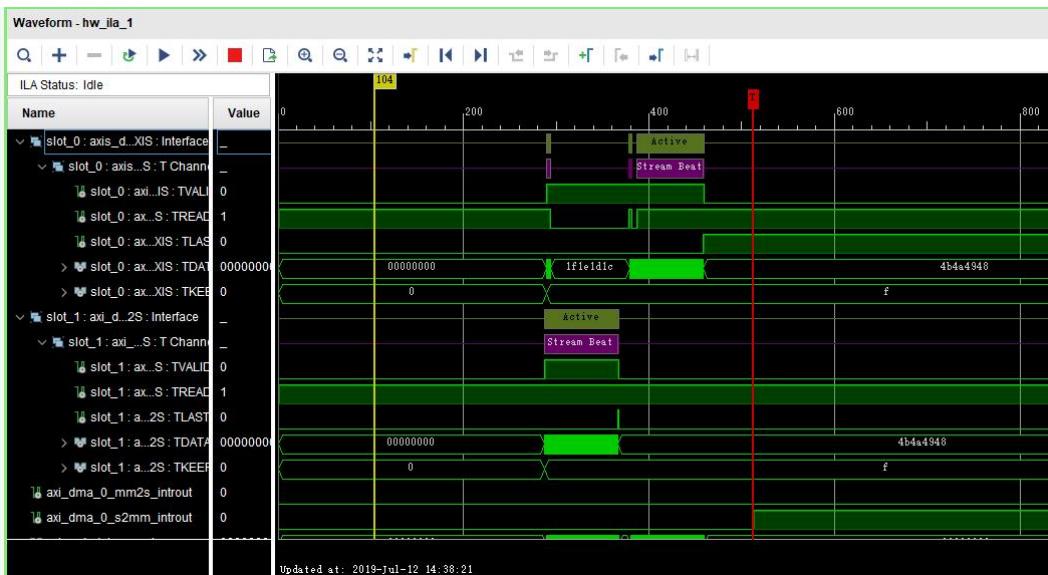
- 2) Open “ILA” and set the trigger condition “m\_axi\_mm2s\_rvalid” rising edge



- 3) Go back to the “Debug” interface of the “Vitis”, without setting a breakpoint, click on “Resume”



- 4) At this point, you can see that “ILA” has been triggered, and you can observe the collected data.



- 5) In the serial debugging tool, you can see the print information, interrupted twice, and the test is successful.

```
--- Entering main() ---
Enter Interrupt
Enter Interrupt
Successfully Ran XAxiDma Test
--- Exiting main() ---
```

- 6) You can also observe the “memory” information in the “SDK” debugging, set the breakpoint as shown below, and set the breakpoint in the interrupt service function.

```

static int CheckData(void)
{
    u8 *RxPacket;
    int Index = 0;
    u8 Value;

    RxPacket = RxBufferPtr;
    Value = TEST_START_VALUE;

    xil_printf("Enter Interrupt\r\n");
    /*Clear Interrupt*/
    XAxiDma_IntrAckIrq(&AxiDma, XAXIDMA IRQ_IOC_MASK,
                        XAXIDMA_DEVICE_TO_DMA);

    /* Invalidate the DestBuffer before receiving the data, i
     * Data Cache is enabled
     */
    Xil_DCacheInvalidateRange((UINTPTR)RxPacket, MAX_PKT_LEN);

    for(Index = 0; Index < MAX_PKT_LEN; Index++) {
        if (RxPacket[Index] != Value) {

```



- 7) Restart “Run Configurations”, click the “Resume” key to run to the breakpoint, add “TxBUFFERptr” and “RxBUFFERptr” to the “Memory” window to observe the comparison data.

Memory					
Monitors		TxBUFFERptr : 0xFC68 <Hex Integer>			
<input type="checkbox"/> TxBUFFERptr					
<input type="checkbox"/> RxBUFFERptr		Address	0 - 3	4 - 7	8 - B C - F
		000000000000FC60	00000000	00000020	0F0E0D0C 13121110
		000000000000FC70	A 17161514	A 1B1A1918	A 1F1E1D1C A 23222120
		000000000000FC80	A 27262524	A 2B2A2928	A 2F2E2D2C A 33323130
		000000000000FC90	A 37363534	A 3B3A3938	A 3F3E3D3C A 43424140
		000000000000FCAA	A 47464544	A 4B4A4948	A 4F4E4D4C A 53525150
		000000000000FCB0	A 57565554	A 5B5A5958	A 5F5E5D5C A 63626160

## Part 21.5 Experimental Summary

This chapter has a lot of knowledge points, using DMA for memory access, and using DMA interrupt, combined with ILA logic analyzer to observe data, CPU processing Cache processing when reading and writing memory, we can do more exercises, flexible use of DMA.

As mentioned earlier, the AXI bus accesses the DDR on the PS side through the HP port, which is a way of interacting PS and PL data, and the DMA in this chapter is another way of interacting PS and PL data. In essence, the two methods are the same. All access to the PS side DDR, the difference is that the AXI bus accesses the PS side

DDR through the HP port by the PL side code, which is more flexible and controllable for the user. The disadvantage is that the code is required, which is more difficult for those who are not familiar with FPGA. The control of the DMA mode is mainly on the PS side. The PS configures the read and write of DMA. The advantage is that it is more intuitive, but it requires better software knowledge.

## Part 22: Use of DMA--DAC Waveform Generator (AN108)

The experimental Vivado project directory is "ad9708\_dma/vivado"

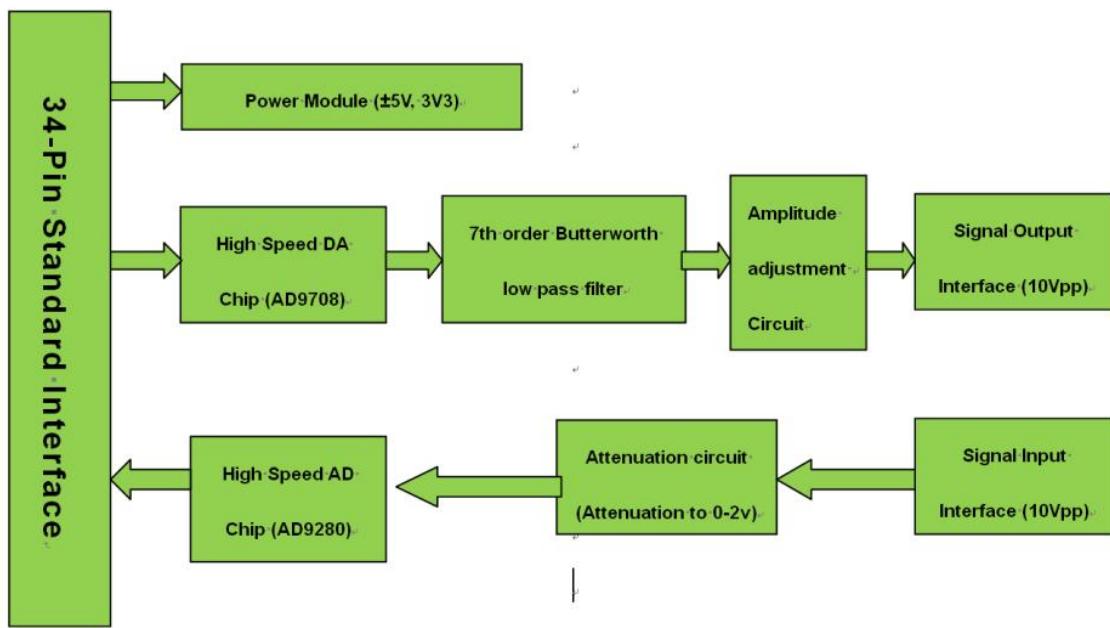
The experimental vitis project directory is "ad9708\_dma/vitis"

I have talked about using DMA for loop-through testing to understand DMA read/write control. This chapter uses DMA read channel to write waveform data to DAC, realize DA conversion, observe waveform with oscilloscope, and the waveform is switched by the key on the PS side to realize a simple waveform generator. The model of the ADDA module used in the experiment is AN108. The maximum sampling rate of the ADC is 32Mhz, the precision is 8 bits, the maximum sampling rate of the DAC is 125Mhz, and the precision is 8 bits.



ADDA Module

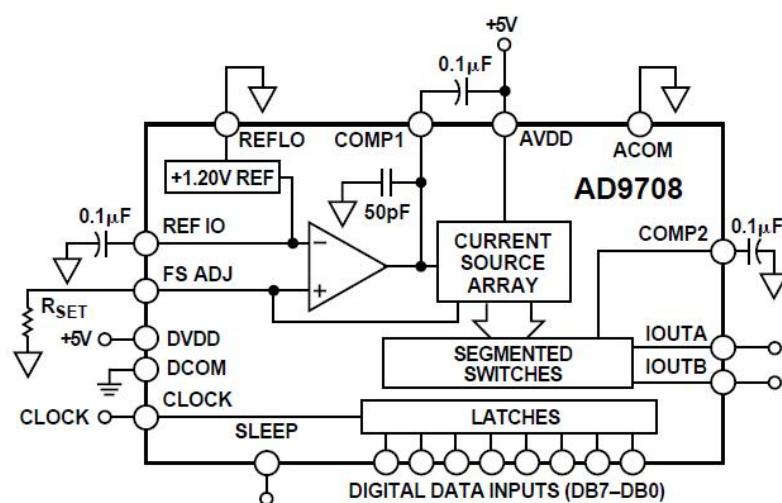
## Part 22.1: Experiment Principle



### Part 22.1.1: Digital to Analog Conversion Circuit

As shown in Figure the hardware block diagram, the DA circuit consists of a high-speed DA chip, a 7th-order Butterworth low-pass filter, an amplitude adjustment circuit, and a signal output interface.

The high-speed DA chip we use is the AD9708 from AD. The AD9708 is a 8-bit, 125MSPS DA conversion chip with a built-in 1.2V reference and differential current output. Figure below is the functional block diagram of chip AD9708.



After the differential output of the AD9708 chip, in order to prevent noise interference, a 7th-order Butterworth low-pass filter is connected to the circuit with a bandwidth of 40MHz. The frequency response is shown in the figure below:

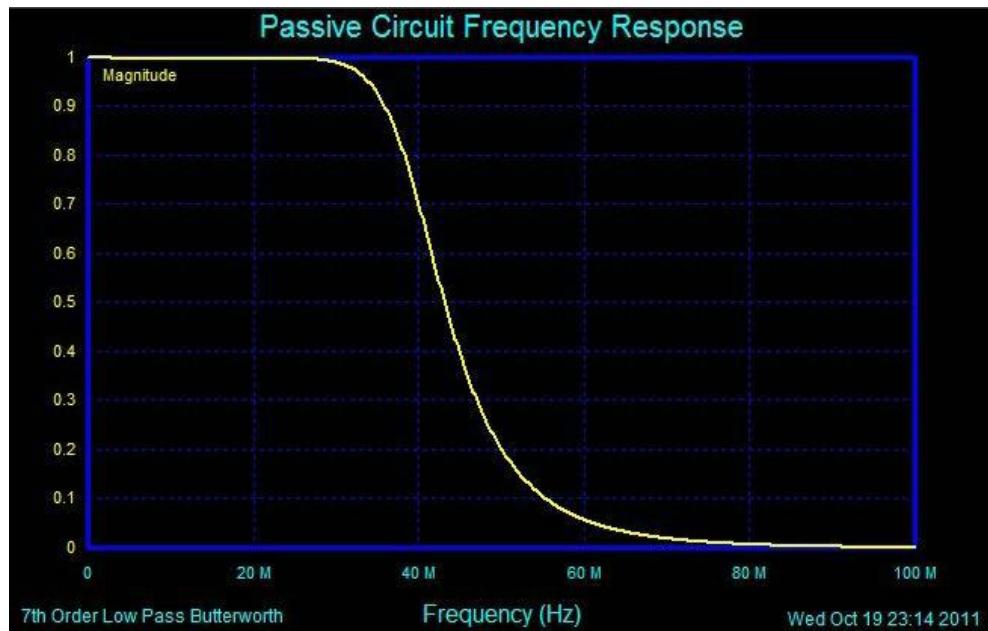
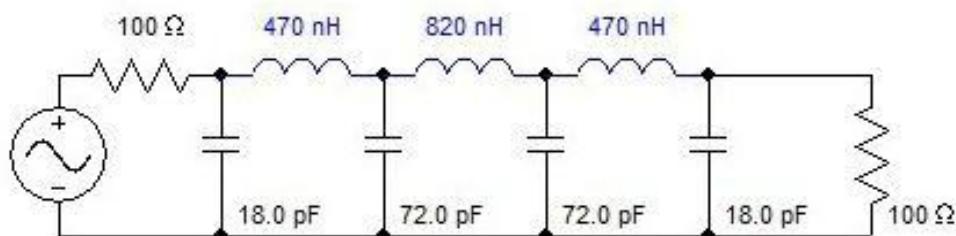


Figure below detailed 7th-order Butterworth low-pass filter parameter:



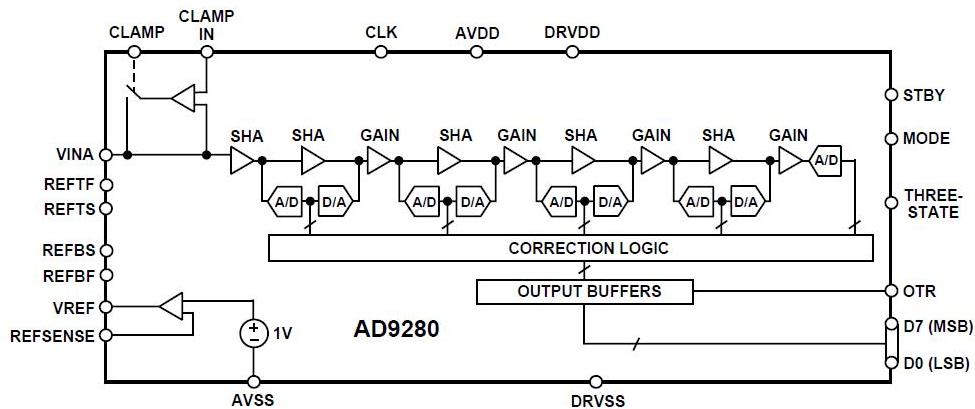
After the filter, we used two high-performance 145MHz bandwidth op amp AD8056 to achieve differential convert to single-ended, amplitude adjustment and other functions, so that the overall circuit performance has been maximized. The amplitude is adjusted using a 5K potentiometer and the final output range is -5V to 5V (10Vpp).

Note: Since the accuracy of the potentiometer is not very accurate, the final output has a certain error. It is possible that the waveform amplitude cannot reach 10Vpp, and there may be problems such as waveform clipping. These are normal conditions.

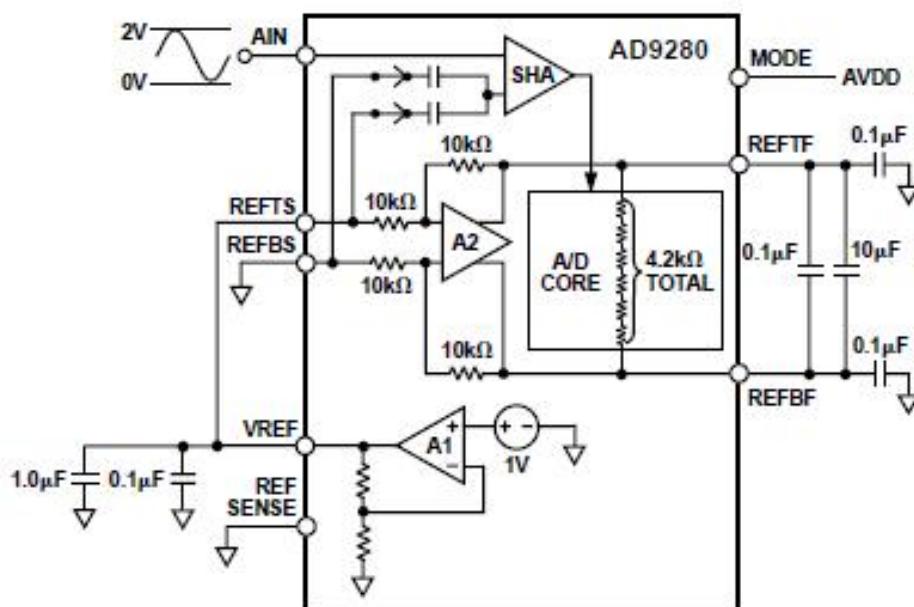
### Part 22.1.2: Analog-to-digital Conversion (AD) Circuit

As shown in the hardware structure diagram, the AD circuit consists of a high-speed AD chip, an attenuation circuit, and a signal input interface.

The high-speed AD chip we use is an 8-bit, ADMS280 chip with a maximum sampling rate of 32MSPS. The internal structure diagram is shown below



According to the configuration shown below, we set the AD voltage input range to: 0V~2V



Before the signal enters the AD chip, an attenuation circuit is constructed with a piece of AD8056 chip. The input range of the

interface is -5V~+5V (10Vpp). After attenuation, the input range meets the input range of the AD chip (0~2V). The conversion formula is as follow

$$V_{AD} = \frac{1}{5} V_{IN} + 1$$

When the input signal  $V_{IN}=5(V)$ , the signal input to AD  $V_{AD}=2(V)$ ;

When the input signal  $V_{IN}=-5(V)$ , the signal input to AD  $V_{AD}=0(V)$ ;

## FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 22.2: Hardware Environment

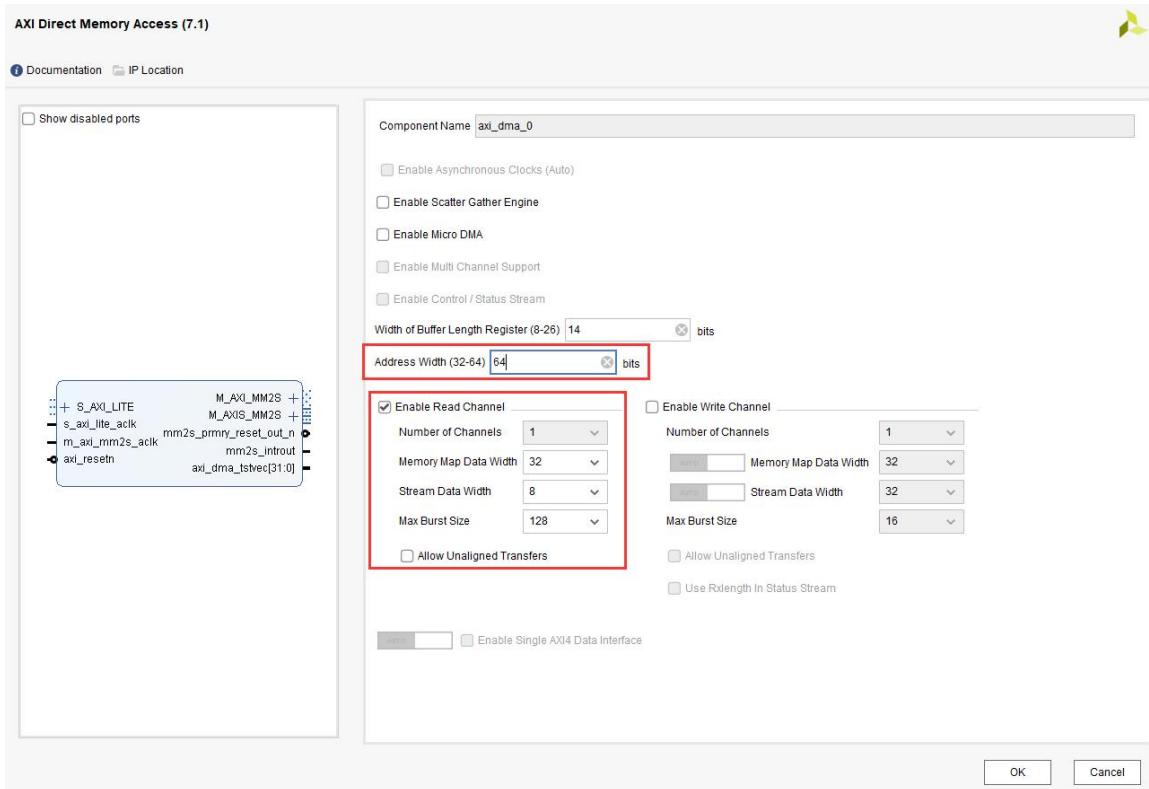
#### Part 22.2.1: Build Hardware

- 1) Set the PL end clock PL0 to 150MHz, add the clock PL1 to the DAC, and set it to the maximum clock frequency of the DAC 125MHz

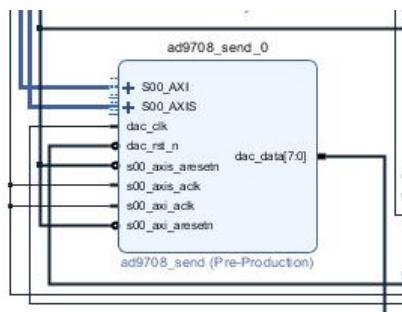
Name	Source	FracEn	Requested Freq (MHz)	Divisor 1	Divisor 2	Actual Frequency (MHz)	Range
PL0	R...		150	7	1	142.855713	0.0000...
PL1	R...		125	8	1	124.998749	0.0000...
PL2	R...		100	4	1	100	0.0000...
PL3	R...		100	4	1	100	0.0000...

- 2) Add DMA, and set the DMA as follows, only open the read channel,

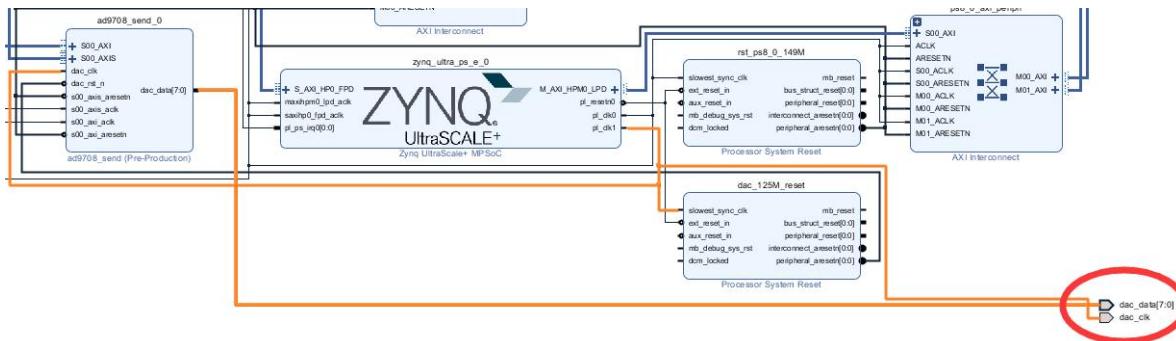
and set the Stream Data Width to 8, because the DAC data width is 8 bits.



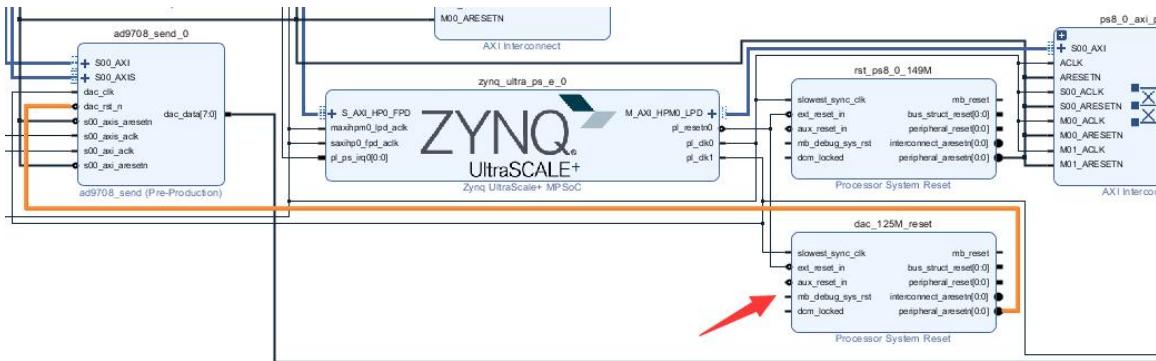
- 3) A custom IP is added to buffer “AXI Stream” data into the FIFO, read data from the FIFO and transmit it to the DAC, and output the DAC data to the port. In the program, once the “start” signal is valid, it starts to read data from the FIFO and send it to the DAC interface. Because the bandwidth of the AXI write FIFO is larger than the DAC bandwidth, it is guaranteed that the read DAC data is continuous. The custom IP is in the “ip\_repo” folder.



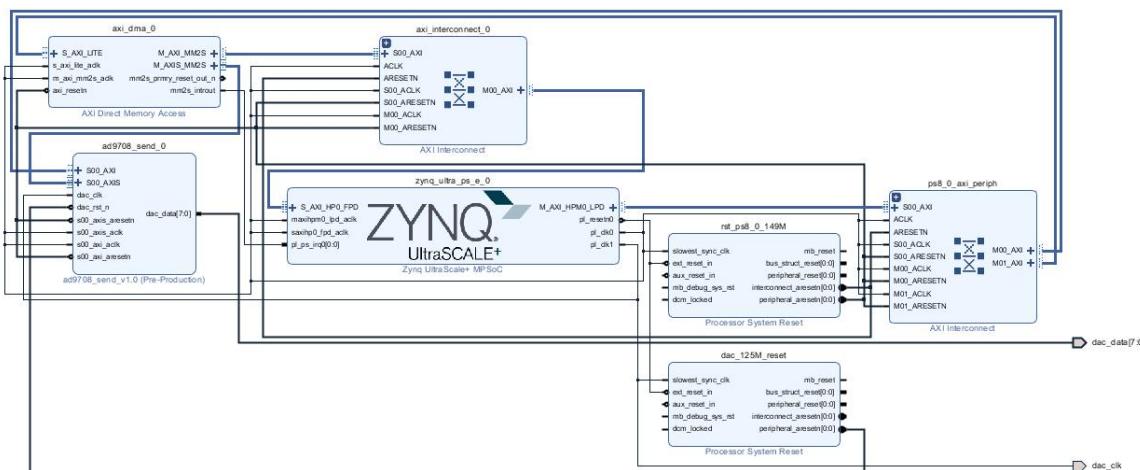
- 4) Lead the DAC clock and data pins as follows



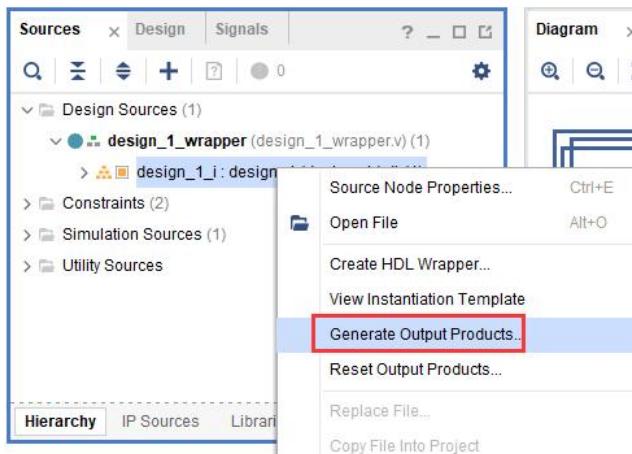
## 5) Add a reset module to reset the dac module



## 6) Connect the corresponding signal, and the final connection result is as follows:



## 7) Then Generate Output Products, and Create HDL Wrapper to generate top-level files



- 8) Bind the DAC pin, compile and generate the bit file, and “Export Hardware”.

### Part 22.2.2: DAC Custom IP Function Introduction

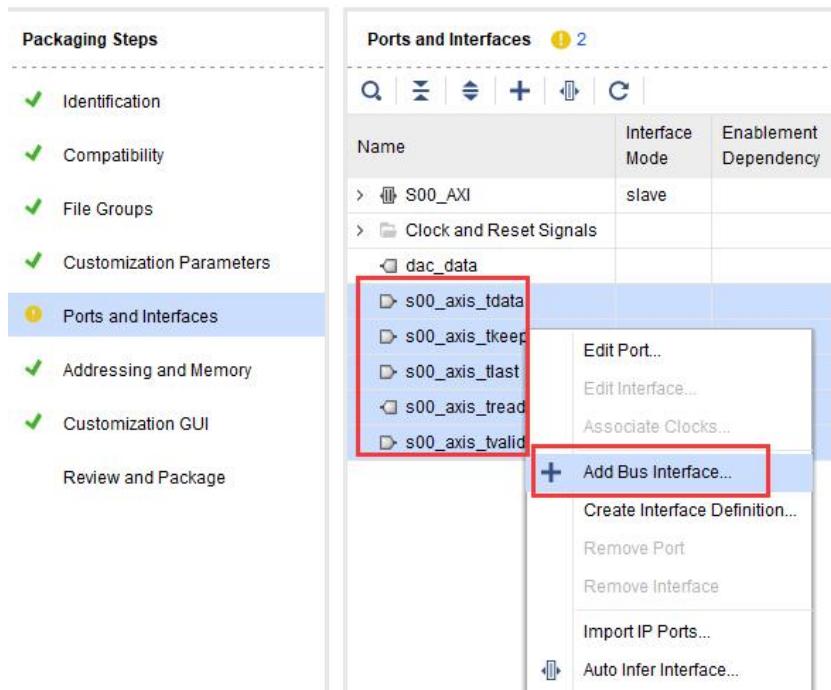
Since the waveform data needs to be transferred to the DAC through DMA, the interface with the DMA is “AXIS” stream interface, so the AXIS stream data needs to be converted into DAC data, and the clock of the DAC is different from the AXIS clock frequency. Therefore, it is necessary to add a FIFO for data processing across clock domains. At the same time, you need to implement the “AXIS Slave” function. The workflow is:

- 1) ARM configuration DAC start register and data length register
- 2) DAM uses AXIS interface to write data to FIFO
- 3) After the DAC state machine queries the FIFO for certain data, it begins to read the data. Since the clock frequency of AXIS is fast, the data read by the DAC can be guaranteed to be continuous.

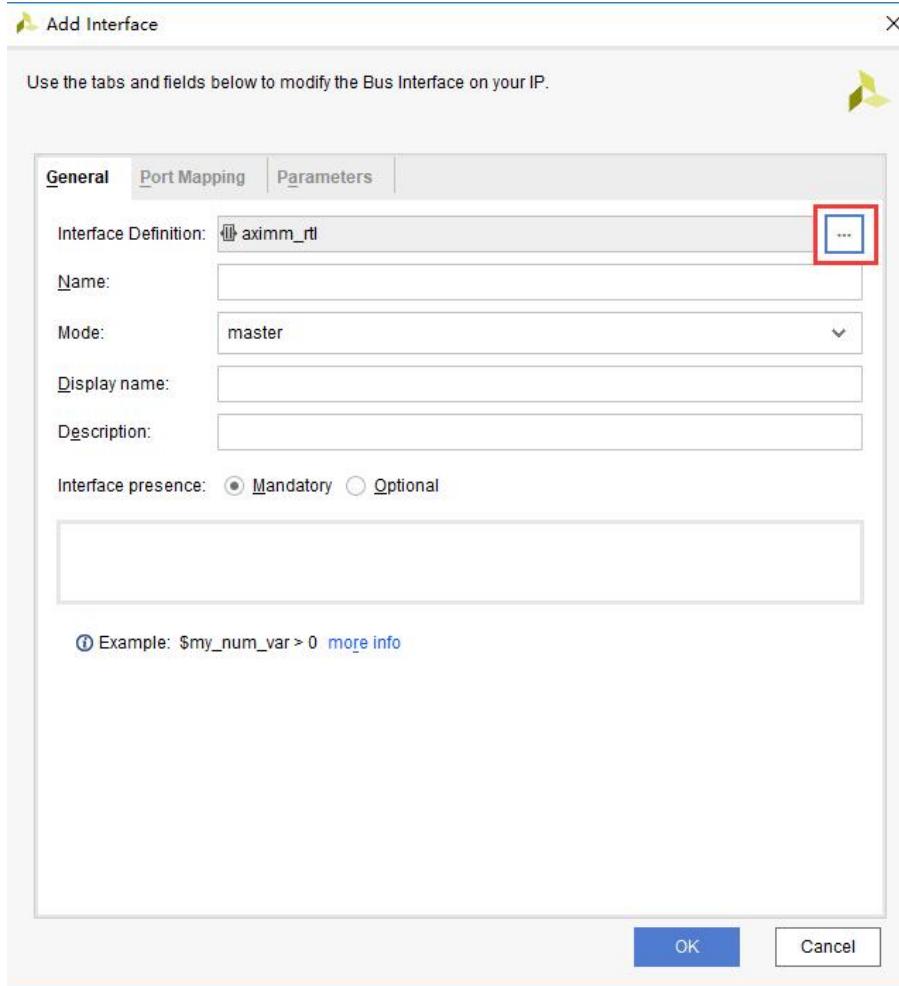
### Part 22.3: Custom IP Port Mapping

- 1) When doing custom IP, for the bus interface, we can encapsulate its signal into an interface to avoid signals being connected one by one. In this experiment, you need to package the “axis slave” interface. In the “Ports and Interfaces” interface, select the signal

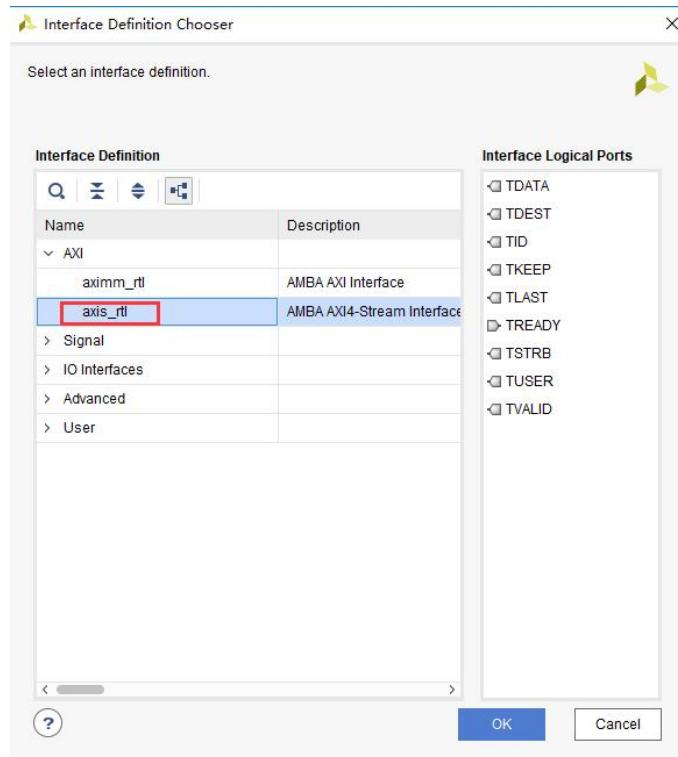
to be packaged, right click and select “Add Bus Interface”.



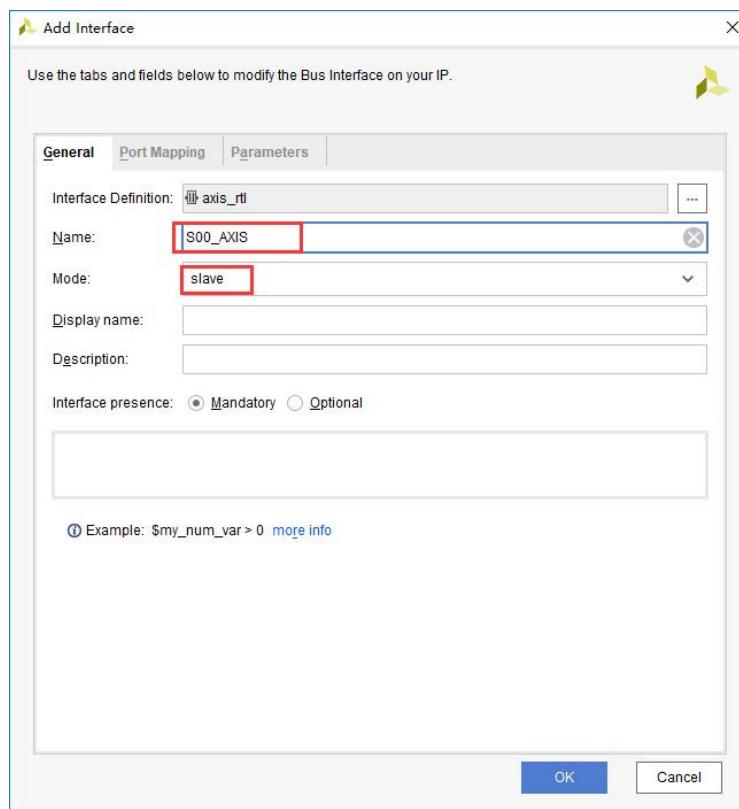
## 2) Pop-up window, click to browse



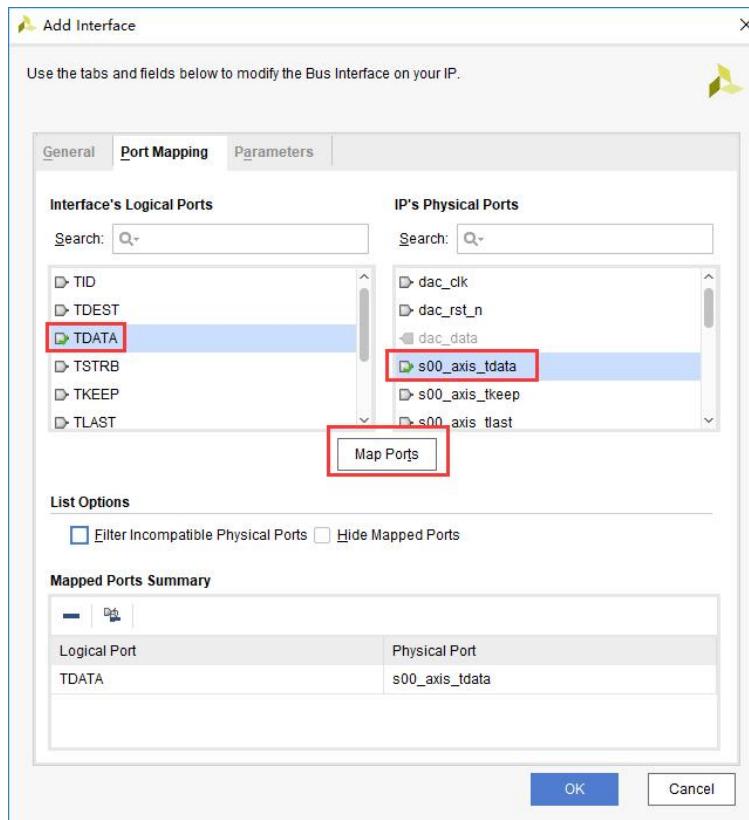
Select the type of bus to be packaged, select “axis\_rtl” here, click OK



- 3) Enter the name in the “Name” and the mode in the “Mode”. Select the “slave” in this experiment.



- 4) Map on the “Port Mapping” interface, the bus signal name on the left, the signal name in the code on the right, select “Map Ports” mapping, and map the remaining signals one by one, and finally click OK.



- 5) You can see the package as a bus form in “Ports and Interfaces”.

Name	Interface Mode
> S00_AXI	slave
S00_AXIS	slave
s00_axis_tdata	
s00_axis_tlast	
s00_axis_valid	
s00_axis_tkeep	
s00_axis_tready	
> Clock and Reset Signals	
dac_data	

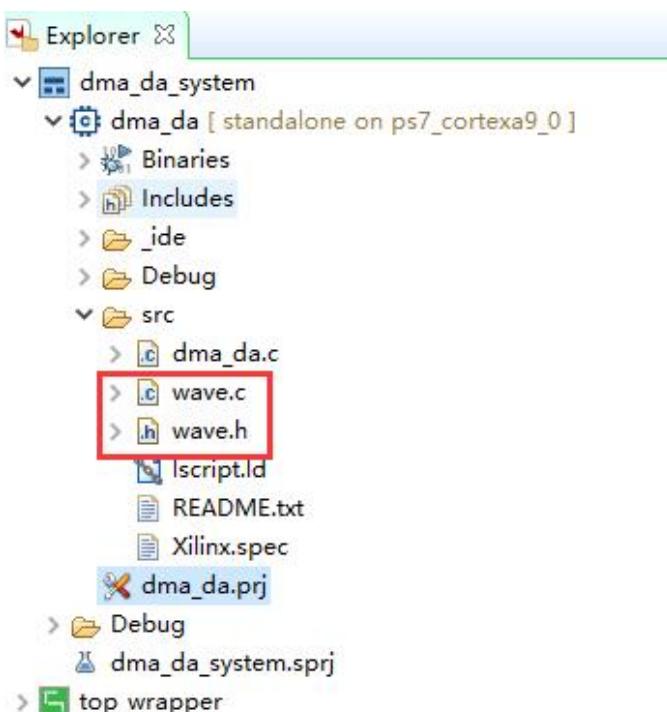
- 6) Of course, other signals can also be mapped, provided that they are familiar with these mapped ports.

## Software Engineer Job Content

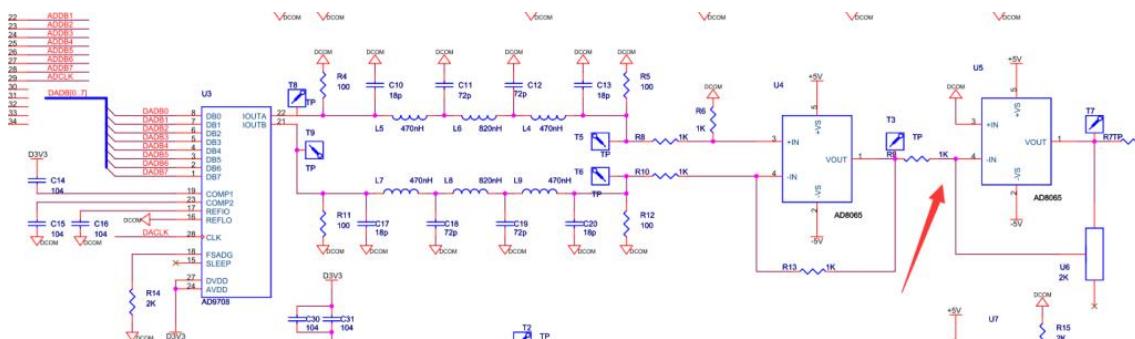
The following is the content that FPGA engineers are responsible for.

### Part 22.4: Vitis Program Development

- 1) “wave.c” and “wave.h” are added to the Vitis program to generate waveforms. Five waveform functions, sine wave, square wave, triangle wave, sawtooth wave, negative sawtooth wave, and other waveform functions can be added.



The data sent to the “DAC” is unsigned. Since the output is reversed during circuit design, if the data is “0x00”, the voltage is a positive maximum and “0xFF” is a negative maximum.



## 2) “SetInterruptInit” function interrupt setting, open interrupt controller

```

int SetInterruptInit(XScuGic *InstancePtr, u16 IntID)
{
    XScuGic_Config * Config ;
    int Status ;

    Config = XScuGic_LookupConfig(IntID) ;

    Status = XScuGic_CfgInitialize(&INST, Config, Config->CpuBaseAddress) ;
    if (Status != XST_SUCCESS)
        return XST_FAILURE ;

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler) XScuGic_InterruptHandler,
                                InstancePtr);

    Xil_ExceptionEnable();

    return XST_SUCCESS ;
}

```

## 3) “KeySetup” function to set the key, set the PS key, register interrupt, enable key interrupt

```

int KeySetup(XScuGic *InstancePtr, u16 IntrID, XGpioPs *GpioInstancePtr)
{
    XGpioPs_Config *GPIO_CONFIG ;
    int Status ;
    key_flag = 0 ;

    GPIO_CONFIG = XGpioPs_LookupConfig(MIO_0_ID) ;
    Status = XGpioPs_CfgInitialize(GpioInstancePtr, GPIO_CONFIG, GPIO_CONFIG->BaseAddr) ;
    if (Status != XST_SUCCESS)
    {
        return XST_FAILURE ;
    }
    //set MIO 11 as input
    XGpioPs_SetDirectionPin(GpioInstancePtr, 11, 0x0) ;
    //set interrupt type
    XGpioPs_SetIntrTypePin(GpioInstancePtr, 11, XGPIOPS IRQ_TYPE_EDGE_RISING) ;

    //set priority and trigger type
    XScuGic_SetPriorityTriggerType(InstancePtr, IntrID,
                                  0xA0, 0x3);
    Status = XScuGic_Connect(InstancePtr, IntrID,
                            (Xil_ExceptionHandler)GpioHandler,
                            (void *)GpioInstancePtr) ;

    XScuGic_Enable(InstancePtr, IntrID) ;

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

    XGpioPs_IntrEnablePin(GpioInstancePtr, 11) ;
    return XST_SUCCESS ;
}

```

- 4) In the “XAxiDma\_Initial” function, open the “MM2S” interrupt

```
/* Disable S2MM interrupt, Enable MM2S interrupt */
XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
                     XAXIDMA_DEVICE_TO_DMA);
XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_IOC_MASK,
                     XAXIDMA_DMA_TO_DEVICE);
```

- 5) In the “XAxiDma\_DAC” function, in the initial state, first write sine wave data to the DAC, write 1 to the “AD9708\_START” register, enable the start signal, start the state machine running the “AD9708”, and then the “AD9708” is always in the transmit state.

```
/* Create initial wave data */
GetSinWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ;
/* Copy wave data to DMA buffer */
memcpy(DmaTxBuffer, WaveBuffer, MAX_PKT_LEN) ;
Xil_DCacheFlushRange((UINTPTR)DmaTxBuffer, MAX_PKT_LEN);

AD9708_SEND_mWriteReg(AD9708_BASE, AD9708_START, 1) ;
```

- 6) In the “while” statement in the “XAxiDma\_DAC” function, the “key\_flag” is generated by the key interrupt, indicating that there is a key press. If there is a key press, the waveform function is switched and refreshed into the memory again.

```
if (key_flag)
{
    switch(wave_sel)
    {
        case 0 : GetSquareWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
        case 1 : GetTriangleWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
        case 2 : GetSawtoothWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
        case 3 : GetSubSawtoothWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
        case 4 : GetSinWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
        default: GetSinWave(MAX_PKT_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
    }

    memcpy(DmaTxBuffer, WaveBuffer, MAX_PKT_LEN) ;
    Xil_DCacheFlushRange((UINTPTR)DmaTxBuffer, MAX_PKT_LEN);

    if (wave_sel == 4)
        wave_sel = 0 ;
    else
        wave_sel++ ;

    /* Clear flag */
    key_flag = 0 ;
}
```

- 7) Start the next “DMA” transfer in the DMA interrupt service function “DAC\_Interrupt\_Handler”

```

if (mm2s_sr & XAXIDMA_IRQ_IOC_MASK)
{
    /* Clear interrupt */
    XAxiDma_IntrAckIrq(XAxiDmaPtr, XAXIDMA_IRQ_IOC_MASK,
        XAXIDMA_DMA_TO_DEVICE) ;

    XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) DmaTxBuffer,
        MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
}

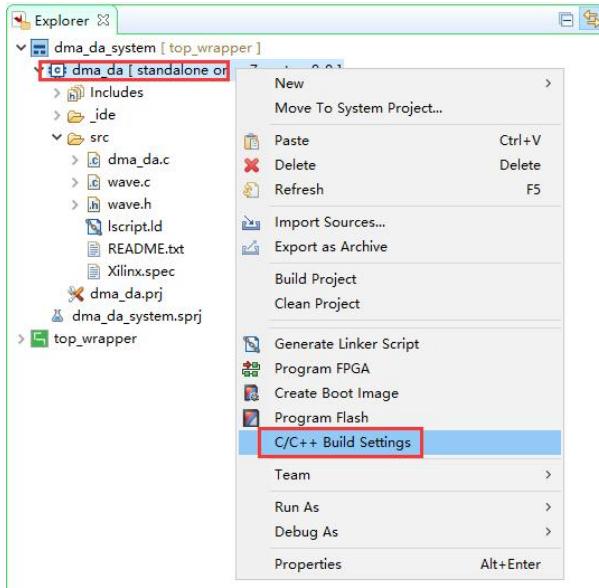
```

- 8) The “MAX\_AMP\_VAL” in the macro definition is the maximum amplitude value. Since the DAC data width is 8, it is set to 256 and should not be changed. The “AMP\_VAL” value is the current amplitude value, which can change the amplitude of the current display, taking care not to exceed 256.

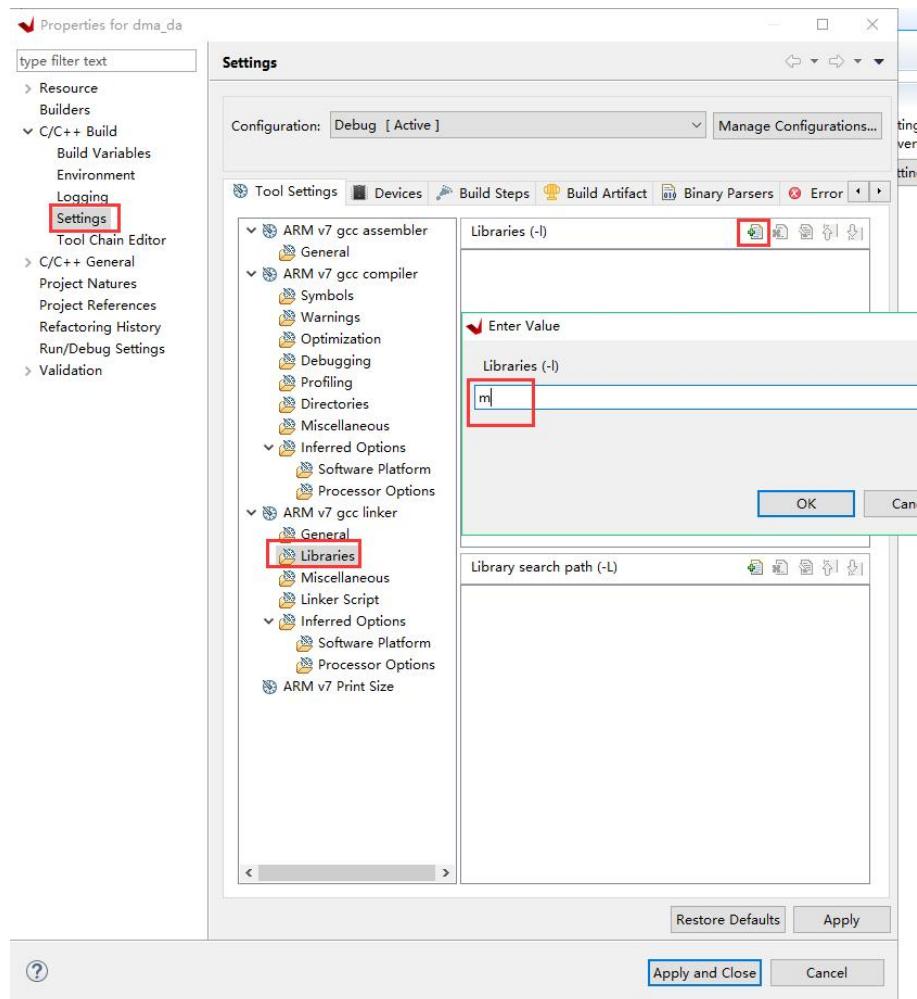
#define MAX_AMP_VAL	256
#define AMP_VAL	256

#### Part 22.4.1: Add math.h Library

Note: The function of “math.h” is used in the program. You need to make the following settings to use it. Right click on the “C/C++ Build Settings” option.

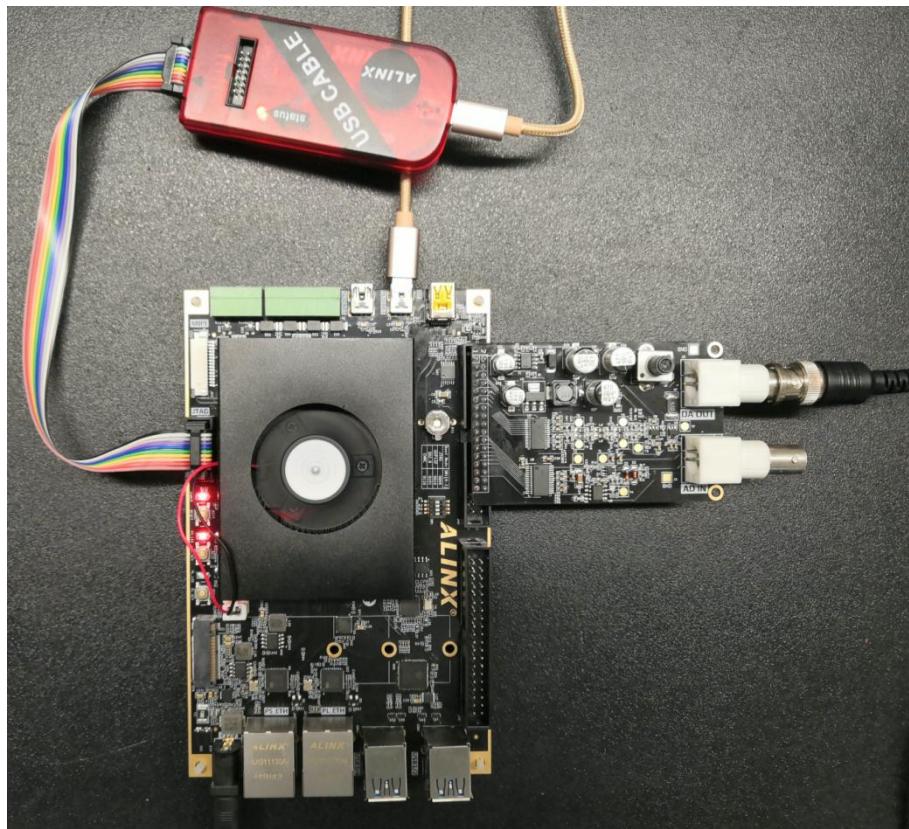


Add “m” in the “Libraries” of the “Settings“ option, click OK



## Part 22.5: Onboard Verification

- 1) Connect the AN108 module to the carrier board as follows. Note that the pin of the module is aligned with the pin1 of the FPGA Carrier board. The dedicated shield cable is used to connect to the DA output port, and the other end is connected to the oscilloscope.

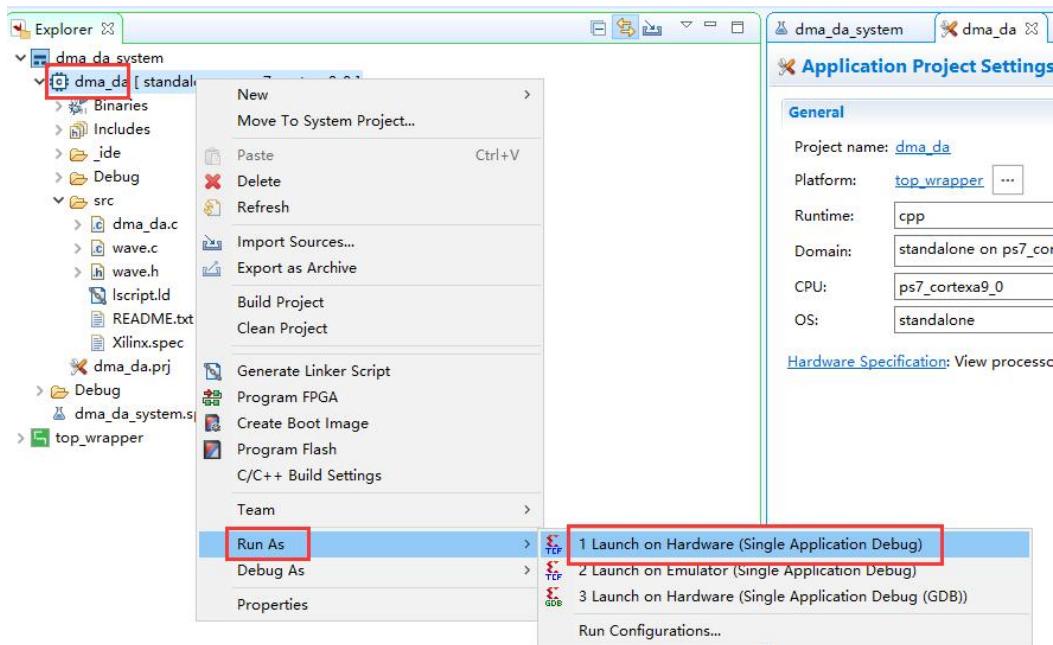


Hardware Connection (Expansion port J46)

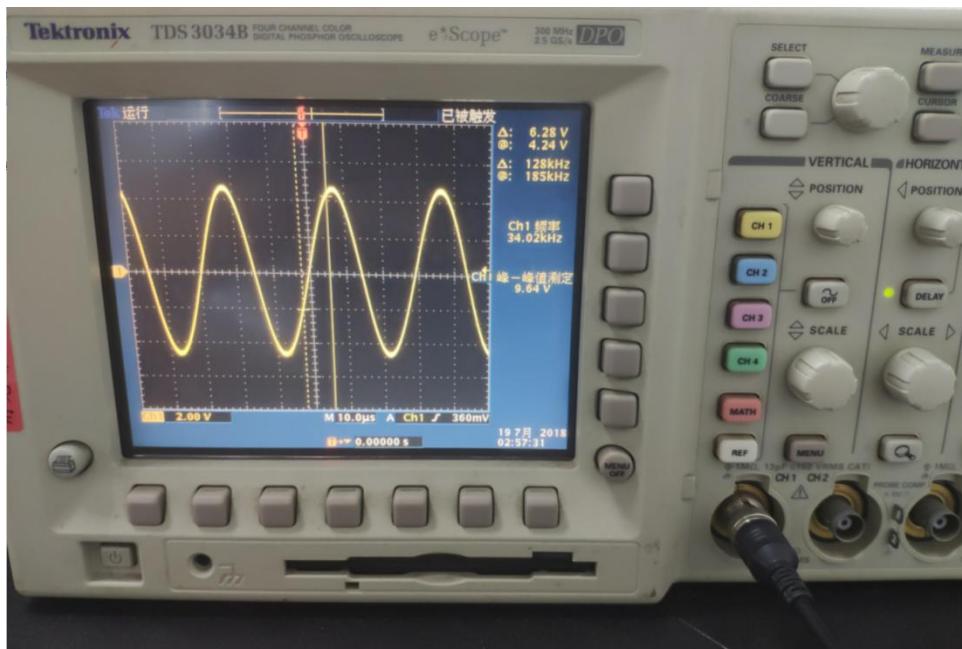


Note that the Pin1 is aligned

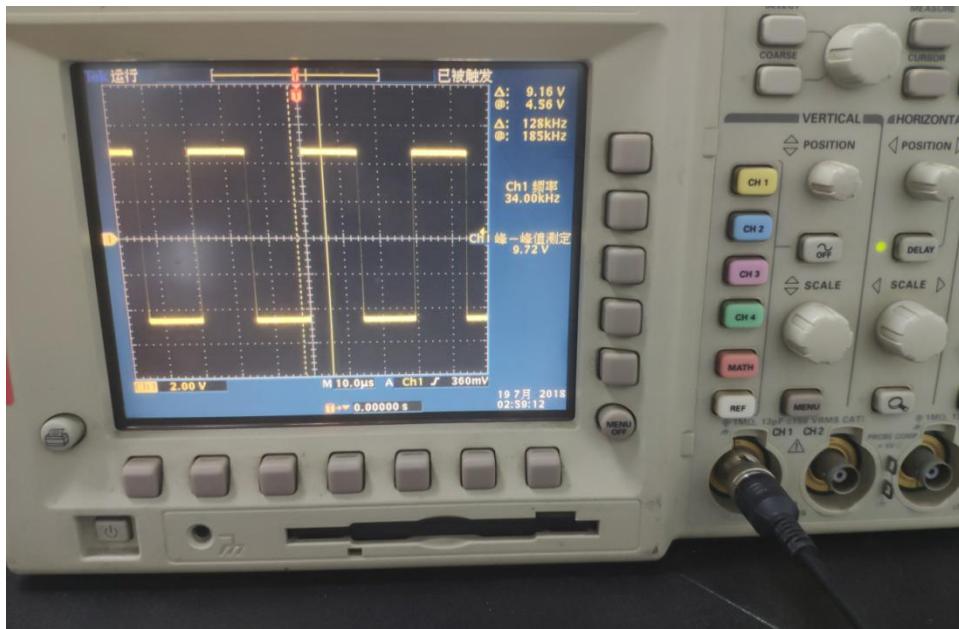
- 2) Download the program



3) The sine wave is displayed by default on the oscilloscope



4) Press the PS\_KEY to switch the waveform



- 5) The amplitude of the waveform can also be adjusted by the potentiometer on the AN108.

## Part 22.6: Experimental Summary

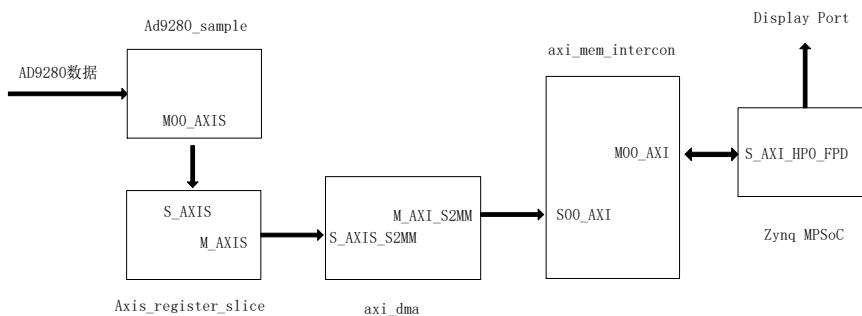
This chapter introduces the use of DMA to transmit waveform data to the DAC, and observe the waveform with an oscilloscope. It is the prototype of the waveform generator. The main knowledge points are the use of DMA interrupts, “AXI Stream” to DAC data conversion.

## Part 23: Use of DMA--ADC oscilloscope (AN108)

The experimental Vivado project directory is "ad9280\_dma\_dp /vivado".

The experiment vitis project directory is "ad9280\_dma\_dp /vitis".

The previous chapter talked about the use of the DAC of the AN108 module. This chapter introduces the ADC acquisition and superimposes the waveform onto the color bar. The final result is as follows:



Block diagram

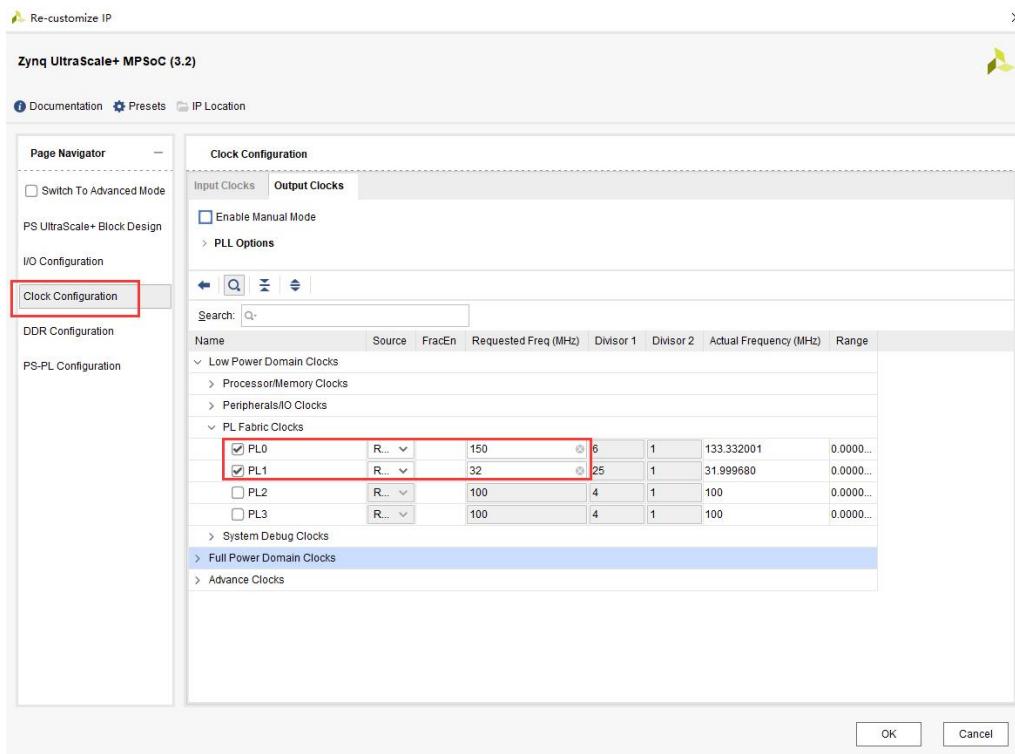
### FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

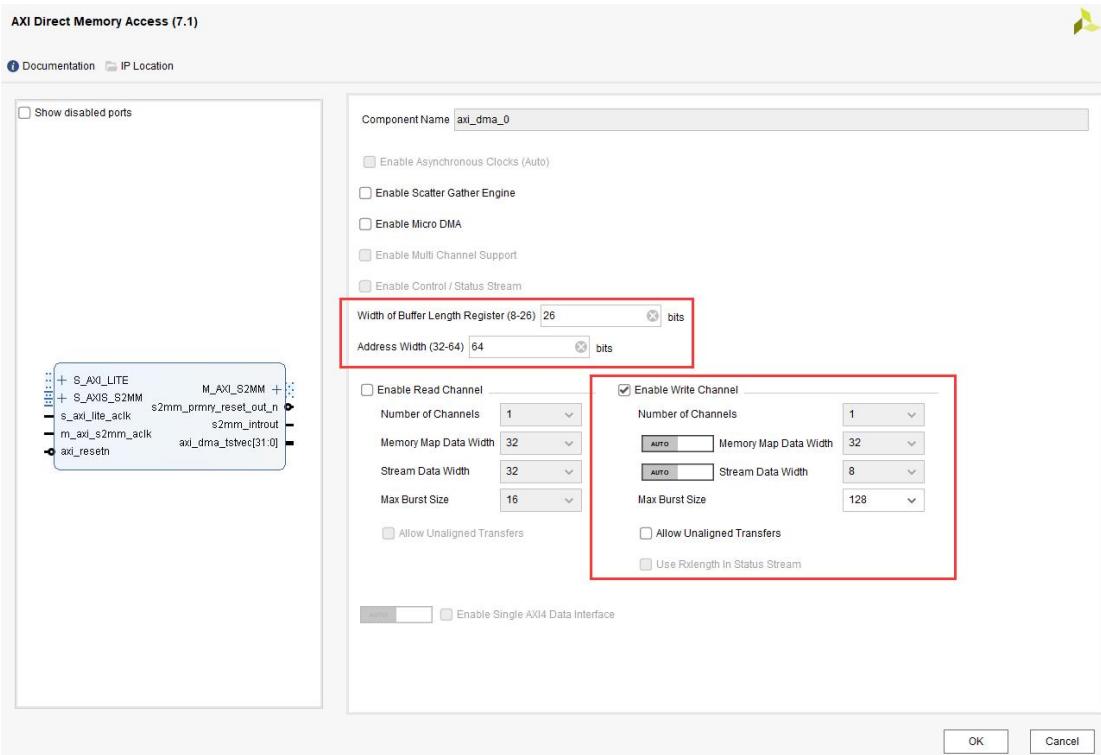
## Part 23.1: Hardware Environment

### Part 23.1.1: Build Hardware

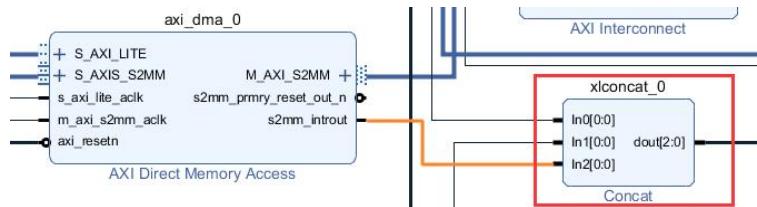
- 1) Based on the "DMA loop-through test" project, delete all modules and peripherals except the ZYNQ core. Open **Zynq** settings, add a clock, set the main clock **PL0** to **150MHz**, and set the maximum frequency of the ADC clock **PL1** to **32MHz**



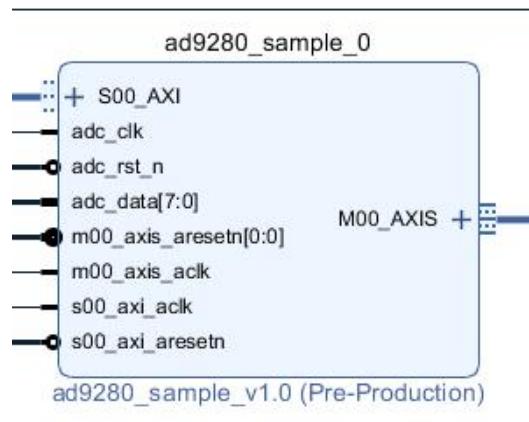
- 2) Add DMA module, the configuration is as follows



### 3) Add Concat interface, connect to the interrupt port of dma

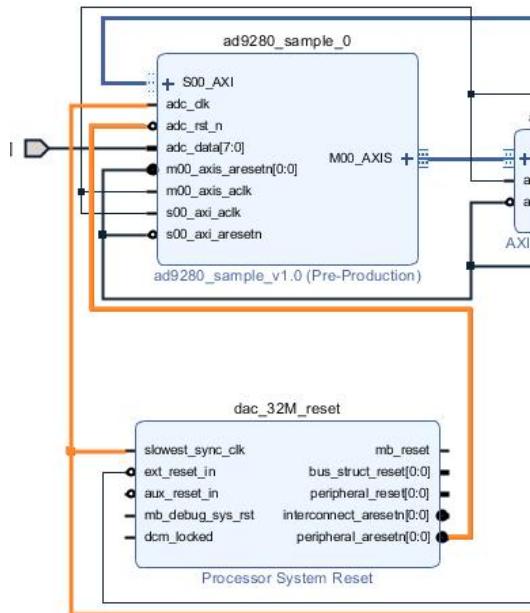


### 4) Add the custom IP module “ad9280\_sample”, the function is to collect “ad9280” data, cache it into the FIFO, and read from the FIFO to convert to “AXI4-Stream” stream data. The custom IP is in the “ip\_repo” folder.

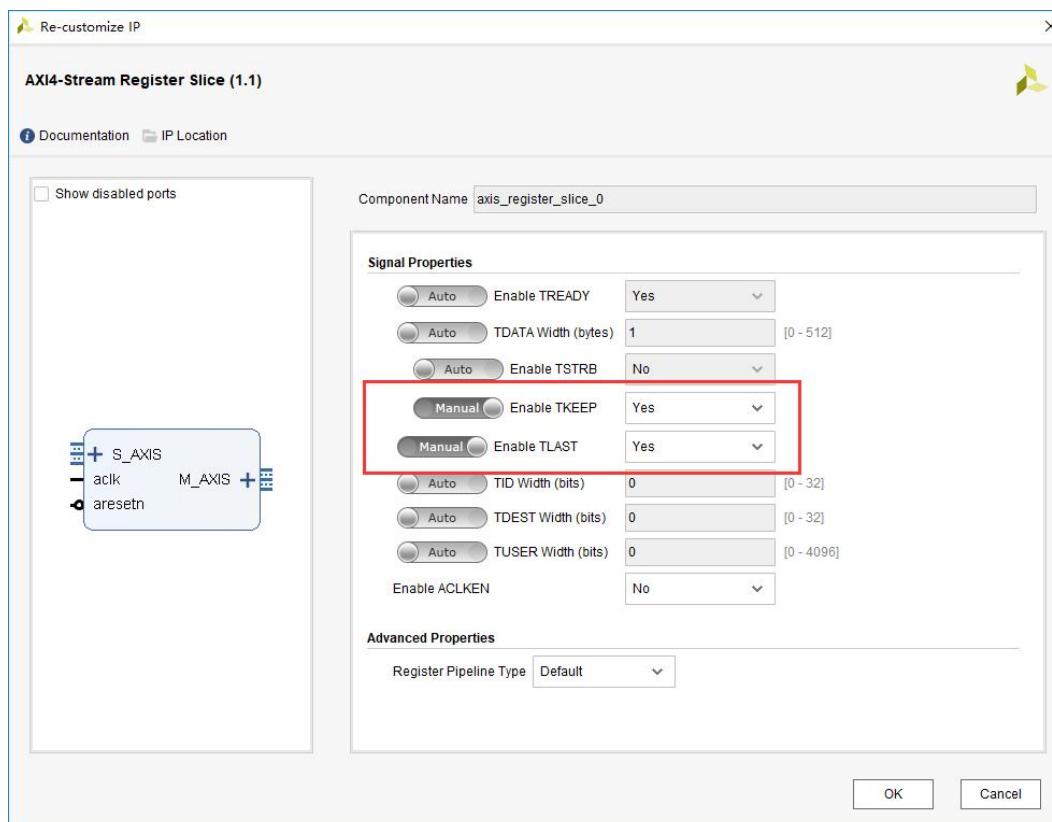


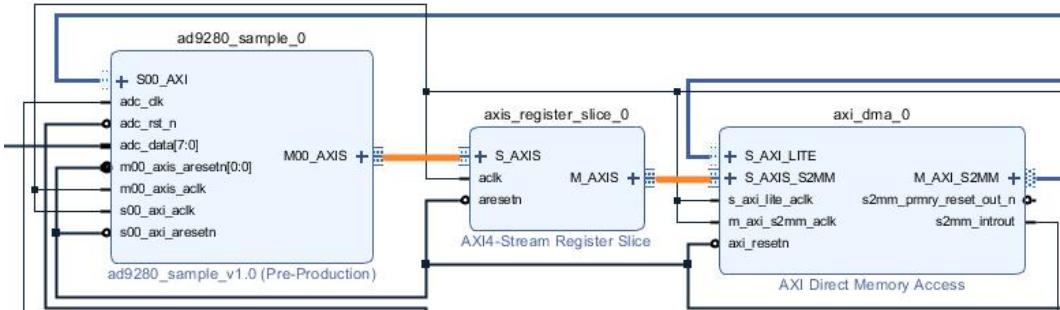
### 5) Add a synchronous reset module and connect the reset clock to

“adc\_clk”. The reset output is connected to the “adc\_rst\_n” reset interface of “ad9280”.

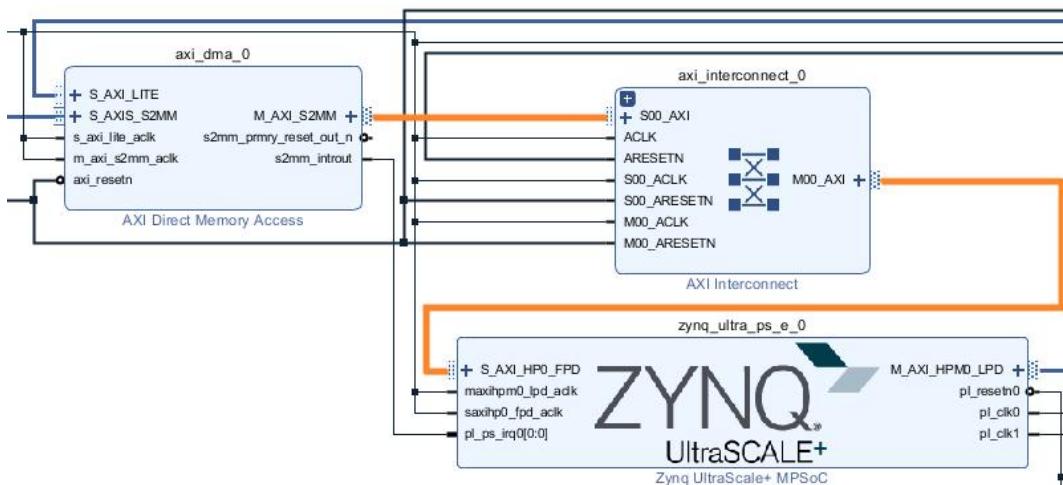


- 6) Adding the “AXI4-Stream Register Slice” module (optional) will improve the timing of the “Stream” interface.

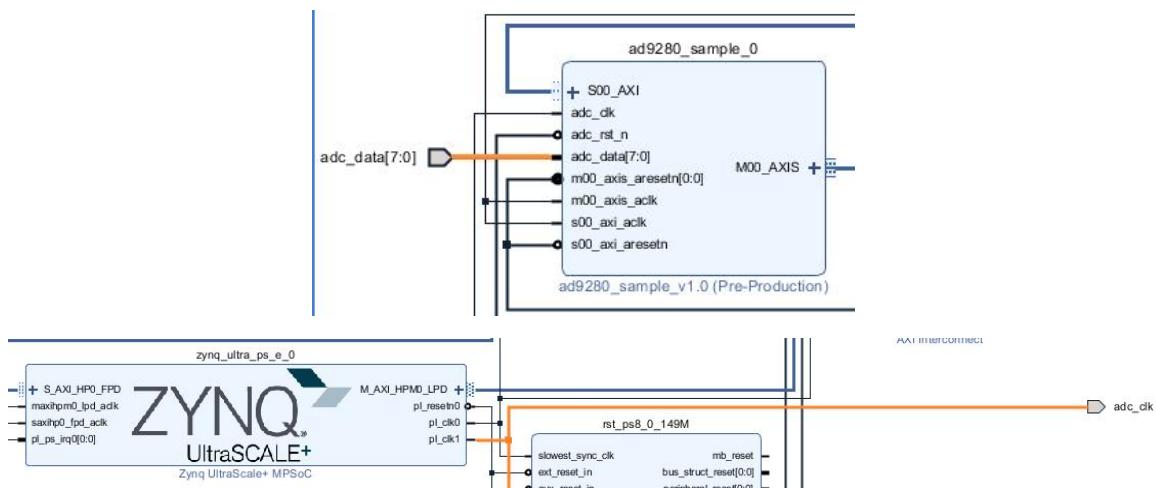




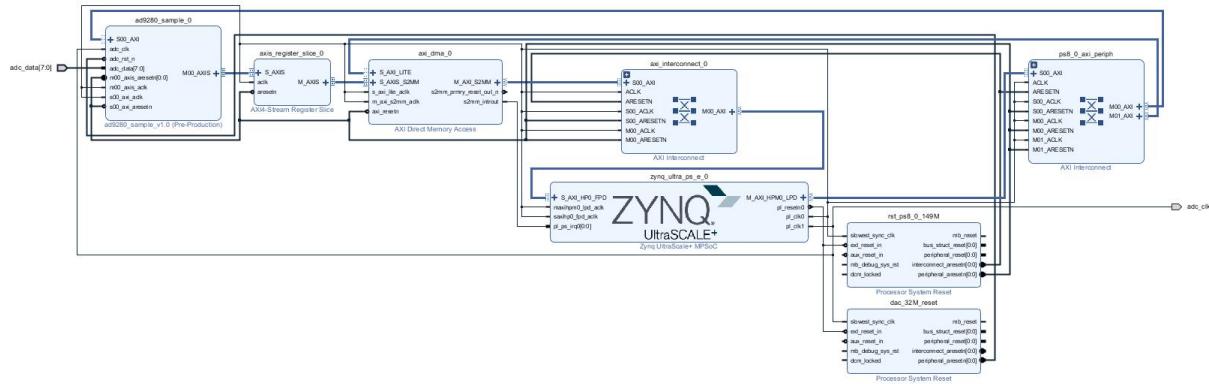
- 7) Add AXI Interconnect module, connect **S00\_AXI** to the **S2MM** interface of **DMA**, and connect **M00\_AXI** to **HP0** port



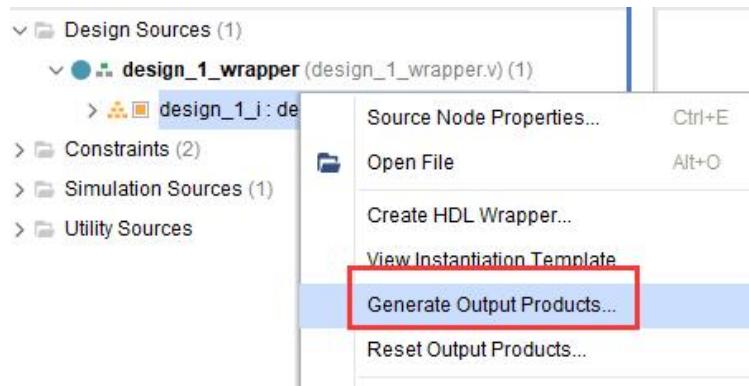
- 8) Lead the ADC data interface and clock, and modify the pin name



The final connection result is as follows



- 9) Connect other corresponding signals, save, and regenerate Output Products



- 10) Bind the “AD9280” pin in the “XDC”, then “Generate Bitstream”

### Part 23.1.2: ADC Custom IP Function Introduction

Since the data collected by the “ADC” needs to be transferred to “ZYNQ” through “DMA”, the interface with the “DMA” is “AXIS” stream interface, so the “ADC” data needs to be converted into “AXIS” stream data, and the clock of the “ADC” is different from the “AXIS” clock frequency. Therefore, it is necessary to add a “FIFO” for data processing across clock domains. At the same time, you need to implement the “AXIS” Master function. The workflow is:

- 1) The ARM configures the start register and the acquisition length register.
- 2) ADC collects data and stores it in FIFO
- 3) The DMA uses the “AXIS” interface to read the data in the “FIFO”

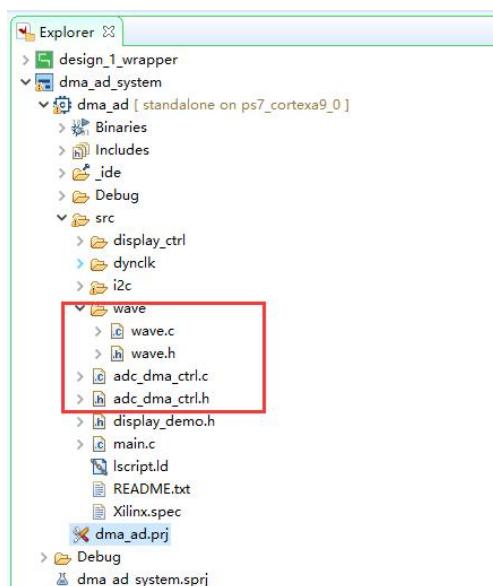
until the configured amount of data is read.

## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 23.2: Vitis Program Development

- 1) The experimental flow is: writing color bar data to the frame buffer → superimposing the grid → superimposing the waveform data.
- 2) Added “adc\_dma\_ctrl.c” and “adc\_dma\_ctrl.h” files to the program. And the “wave.c” and “wave.h” files, which can be found under the Vitis folder.



- 3) The first thing to do is to display the background. In this experiment, select the color bar as the background, and use the “DemoPrintTest” function of the “main.c” file, delete the others, leaving only the color bar part.

```
void DemoPrintTest(u8 *frame, u32 width, u32 height, u32 stride, int pattern)
{
    u32 xcoi, ycoi;
    u32 iPixelAddr = 0;
    u8 wRed, wBlue, wGreen;
    u32 xInt;

    xInt = width*BYTES_PIXEL / 8; //each with width/8 pixels

    for(ycoi = 0; ycoi < height; ycoi++)
    {
```

4) Open the interrupt controller in “main.c” for the “DMA” interrupt.

```
int SetInterruptInit(XScuGic *InstancePtr, u16 IntrID)
{
    XScuGic_Config * Config ;
    int Status ;

    Config = XScuGic_LookupConfig(IntrID) ;

    Status = XScuGic_CfgInitialize(InstancePtr, Config, Config->CpuBaseAddress) ;
    if (Status != XST_SUCCESS)
        return XST_FAILURE ;

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        (Xil_ExceptionHandler) XScuGic_InterruptHandler,
        InstancePtr);

    Xil_ExceptionEnable();

    return XST_SUCCESS ;
}
```

5) The next step is to superimpose the grid and waveform. “adc\_dma\_ctrl.c” is based on the previous DMA control. The “XAxiDma\_Adc\_Wave” function is used to initialize the “DMA”, control the “ADC” acquisition, and waveform overlay. Since the “DMA” has only a write interface, the “S2MM” interrupt is turned on in the “XAxiDma\_Initial” function.

```
/* Disable MM2S interrupt, Enable S2MM interrupt */
XAxiDma_IntrEnable(&xAxiDma, XAXIDMA IRQ_IOC_MASK,
    XAXIDMA_DEVICE_TO_DMA);
XAxiDma_IntrDisable(&xAxiDma, XAXIDMA IRQ_ALL_MASK,
    XAXIDMA_DMA_TO_DEVICE);
```

6) In the “adc\_dma\_ctrl.c” call the “draw\_grid” function overlay grid, “draw\_grid” in the “wave.c” file, you need to provide the parameter width, height, that is, to display the width and height of the grid. Each square set in the function is 32\*32 pixels, and is displayed every 4 points in the horizontal and vertical directions. The grid is grayed out and the background is black, and the image data is written to the canvas (CanvasBuffer) buffer.

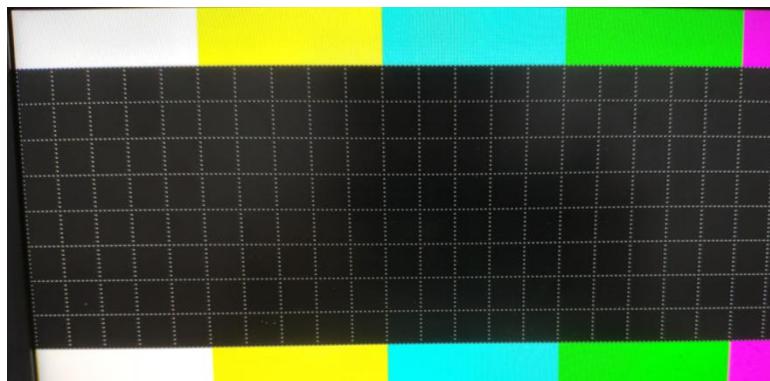
```

void draw_grid(u32 width, u32 height, u8 *CanvasBufferPtr)
{
    u32 xcoi, ycoi;
    u8 wRed, wBlue, wGreen;
    /*
     * overlay grid on canvas, background set to black color, grid color is gray.
     */
    for(ycoi = 0; ycoi < height; ycoi++)
    {
        for(xcoi = 0; xcoi < width; xcoi++)
        {

            if (((ycoi == 0 || (ycoi+1)%32 == 0) && (xcoi == 0 || (xcoi+1)%4 == 0))
                || ((xcoi == 0 || (xcoi+1)%32 == 0) && (ycoi+1)%4 == 0))
            {
                /* gray */
                wRed = 150;
                wGreen = 150;
                wBlue = 150;
            }
            else
            {
                /* Black */
                wRed = 0;
                wGreen = 0;
                wBlue = 0;
            }
            draw_point(CanvasBufferPtr, xcoi, ycoi, width, wBlue, wGreen, wRed);
        }
    }
}

```

The effect is as follows:



- 7) The superimposed waveform function is “draw\_wave”, “width” is width, “height” is height, and “BufferPtr” is waveform data pointer. In this experiment, it points to the data received by the “ADC”. “CanvasBufferPtr” is the canvas pointer, and the processed data is superimposed on it. “Sign” is the sign bit of “BufferPtr” data, “Bits” is a valid data bit. For example, if the “ADC” has a data width of 8, you can set this parameter to 8. The parameter color is used to select the color to be displayed, and the “coe” is the coefficient.

The height of the waveform can be changed by adjusting the value of the coe. Since the AD9280 has a data width of 8, the coe is set to 1 in this experiment.

```
@void draw_wave(u32 width, u32 height, void *BufferPtr, u8 *CanvasBufferPtr, u8 Sign, u8 Bits, u8 color, u16 coe)
{
```

Determine the Sign bit and assign it to a different pointer.

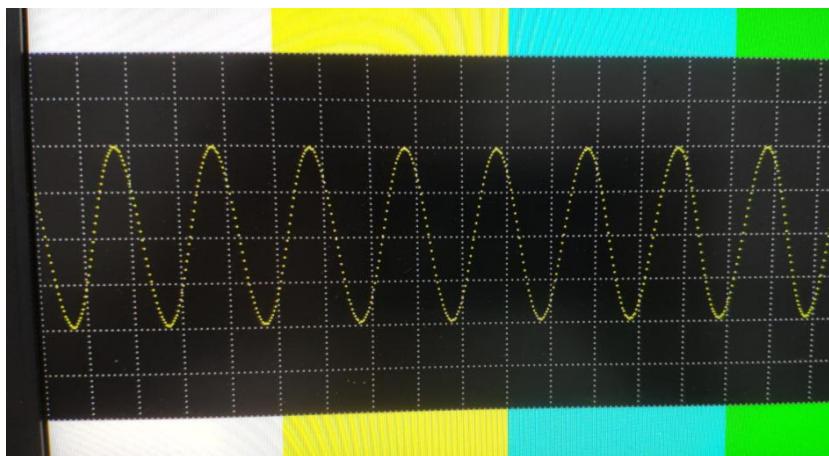
```
    char *CharBufferPtr ;
    short *ShortBufferPtr ;

    if(Sign == UNSIGNEDCHAR || Sign == CHAR)
        CharBufferPtr = (char *)BufferPtr ;
    else
        ShortBufferPtr = (short *)BufferPtr ;
```

Since the obtained data is displayed as discrete points on the image, in order to make the waveform display smoother, the drawing processing is performed, and the data is compared with the previous data to obtain a difference, and the points are drawn in the same column.

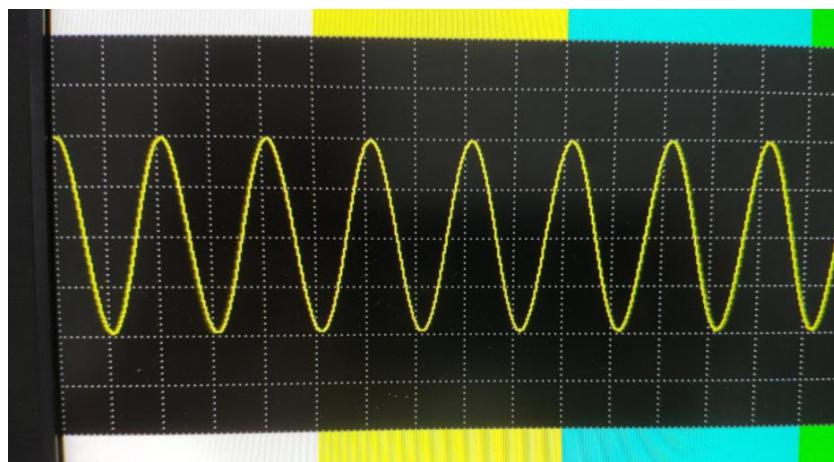
```
/* Convert char data to u8 */
if (i == 0)
{
    if(Sign == UNSIGNEDCHAR || Sign == CHAR)
    {
        last_data = (u8)(CharBufferPtr[i] + adder)*data_coe ;
        curr_data = (u8)(CharBufferPtr[i] + adder)*data_coe ;
    }
    else
    {
        last_data = (u8)((u16)(ShortBufferPtr[i] + adder)*data_coe) ;
        curr_data = (u8)((u16)(ShortBufferPtr[i] + adder)*data_coe) ;
    }
}
```

The following are 500KHz unspoken sine waves, all of which are discrete points:



The figure below shows the effect after the point is drawn,

smoothing out a bit



The drawing function is “draw\_point”, and you need to provide parameters such as abscissa, ordinate, width, height, etc.

```
void draw_point(u8 *PointBufferPtr, u32 hor_x, u32 ver_y, u32 width, u8 wBlue, u8 wGreen, u8 wRed)
{
    PointBufferPtr[(hor_x + ver_y*width)*BYTES_PIXEL + 0] = wBlue;
    PointBufferPtr[(hor_x + ver_y*width)*BYTES_PIXEL + 1] = wGreen;
    PointBufferPtr[(hor_x + ver_y*width)*BYTES_PIXEL + 2] = wRed;
}
```

- 8) In the “XAxiDma\_Adc\_Wave” function of “adc\_dma\_ctrl.c”, call the “frame\_copy” function to copy the canvas data to the image space, and refresh the “Cache”, then open the ADC acquisition

```
if (s2mm_flag)
{
    /* clear s2mm_flag */
    s2mm_flag = 0 ;
    /* Copy grid to Wave buffer */
    memcpy(WaveBuffer, GridBuffer, WAVE_LEN) ;
    /* wave Overlay */
    draw_wave(wave_width, WAVE_HEIGHT, (void *)DmaRxBuffer, WaveBuffer, UNSIGNEDCHAR, ADC_BITS, YELLOW, ADC_COE) ;
    /* Copy Canvas to frame buffer */
    frame_copy(wave_width, WAVE_HEIGHT, stride, WAVE_START_COLUMN, WAVE_START_ROW, frame, WaveBuffer) ;
    /* delay 100ms */
    usleep(100000) ;
    /* Start sample */
```

- 9) The previous experiment has already been said, modify the display resolution method, modify “vMode” in “display\_ctrl.c”

```
/*
 * Initialize all the fields in the DisplayCtrl struct
 */
dispPtr->curFrame = 0;
dispPtr->dynClkAddr = dynClkAddr;
for (i = 0; i < DISPLAY_NUM_FRAMES; i++)
{
    dispPtr->framePtr[i] = framePtr[i];
}
dispPtr->state = DISPLAY_STOPPED;
dispPtr->stride = stride;
dispPtr->vMode = VMODE_1280x720;

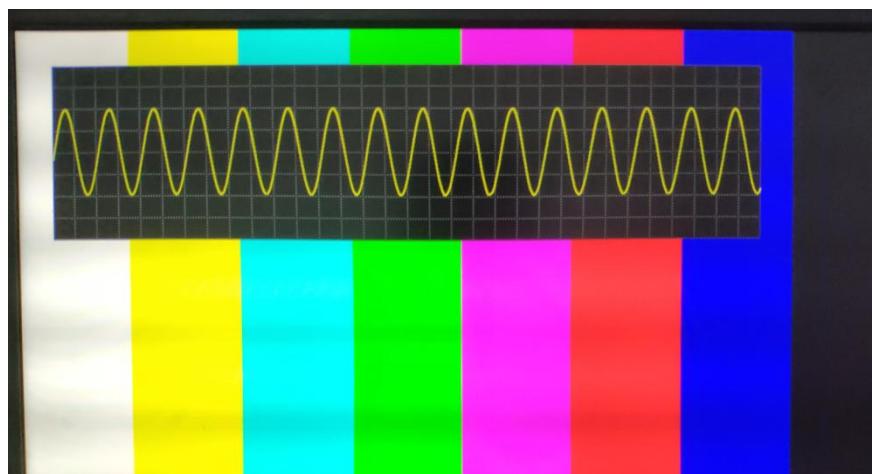
ClkFindParams(dispPtr->vMode.freq, &clkMode);
```

10) If you want to change the waveform background display area, you can modify the grid waveform start position, modify “WAVE\_START\_ROW” to change the start line position, modify “WAVE\_START\_COLUMN” to modify the start column position. Note that “WAVE\_HEIGHT\+WAVE\_START\_ROW” cannot be greater than the resolution height, such as “1280\*720”, which cannot be greater than “720”, otherwise the display is not normal.

```
/*
 *Wave defines
 */
#define WAVE_LEN          1920*256*3      /* Wave total length in byte */
#define WAVE_START_ROW     50                 /* Grid and Wave start row in frame */
#define WAVE_START_COLUMN  0                  /* Grid and Wave start column in frame */
#define WAVE_HEIGHT         256                /* Grid and Wave height */
```

The width of the waveform can also be modified in the “XAxidma\_Adc\_Wave” function, such as changing it to “1024” and changing “WAVE\_START\_COLUMN” to “50”. You can see the effect is as follows

```
int XAxidma_Adc_Wave(u32 width, u8 *frame, u32 stride, XScuGic *InstancePtr)
{
    int Status;
    u32 wave_width = width ;
```



11) When calling the “draw\_wave” function, the “Sign” symbol is set to “UNSIGNEDCHAR”

```
/* wave Overlay */
draw_wave(wave_width, WAVE_HEIGHT, (void *)DmaRxBuffer, CanvasBuffer, UNSIGNEDCHAR, ADC_BITS, YELLOW, ADC_COE) ;
/* Copy Canvas to frame buffer */
```

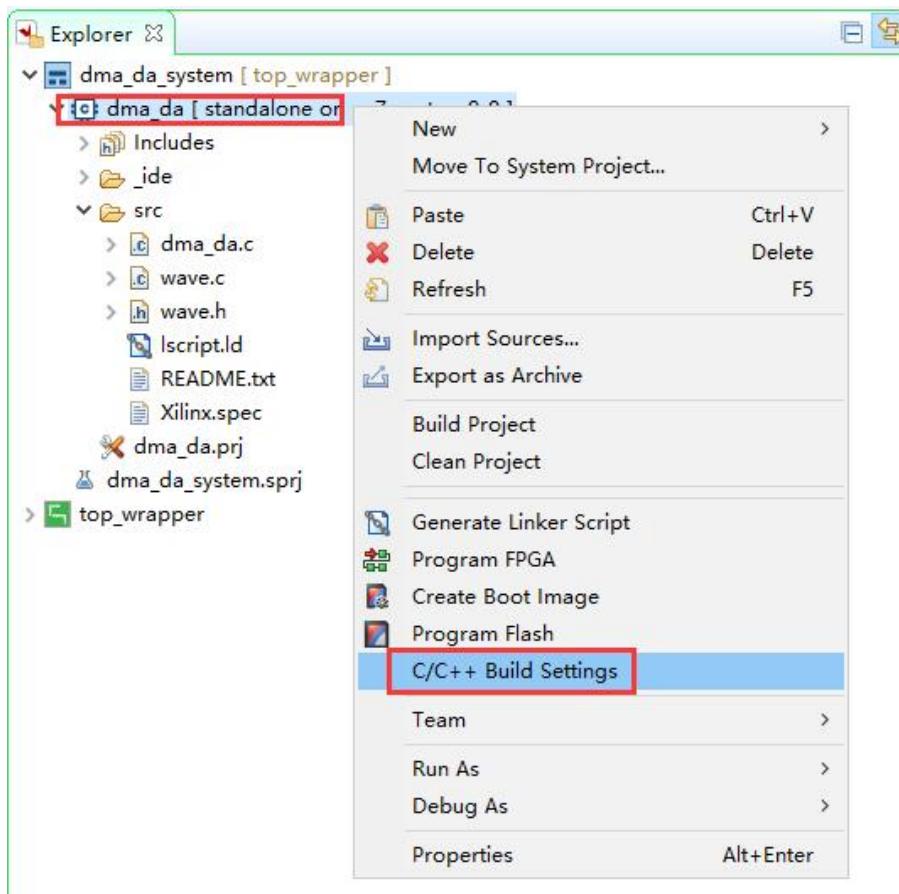
In the “adc\_dma\_ctrl.h” file, the ADC parameters are set as

follows:

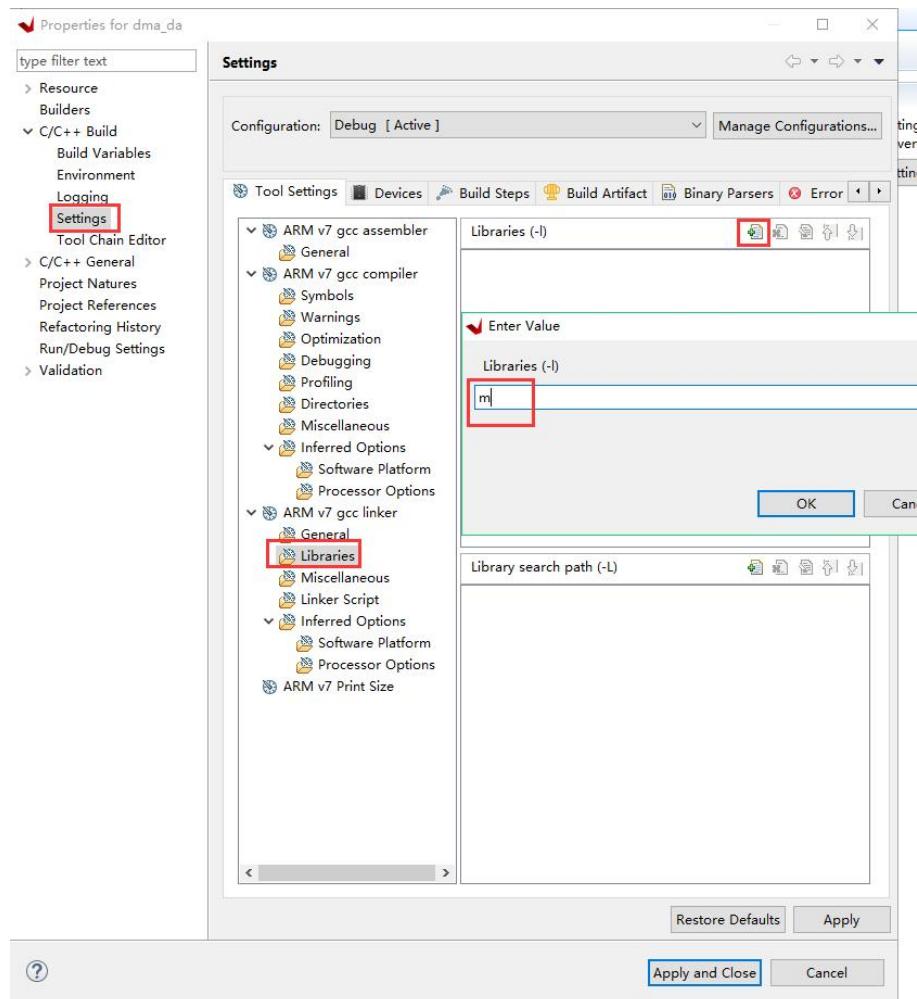
```
#define AD9280_BASE      XPAR_AD9280_SAMPLE_0_S00_AXI_BASEADDR
#define AD9280_START     AD9280_SAMPLE_S00_AXI_SLV_REG0_OFFSET
#define AD9280_LENGTH     AD9280_SAMPLE_S00_AXI_SLV_REG1_OFFSET
#define ADC_CAPTURELEN   1920 /* ADC capture length */
#define ADC_COE          1 /* ADC coefficient */
#define ADC_BYTE         1 /* ADC data byte number */
#define ADC_BITS          8
```

### Part 23.2.1: Add the “math.h” Library

Note: The function of **math.h** is used in the program. You need to make the following settings to use it. Right click on the “C/C++ Build Settings” option.

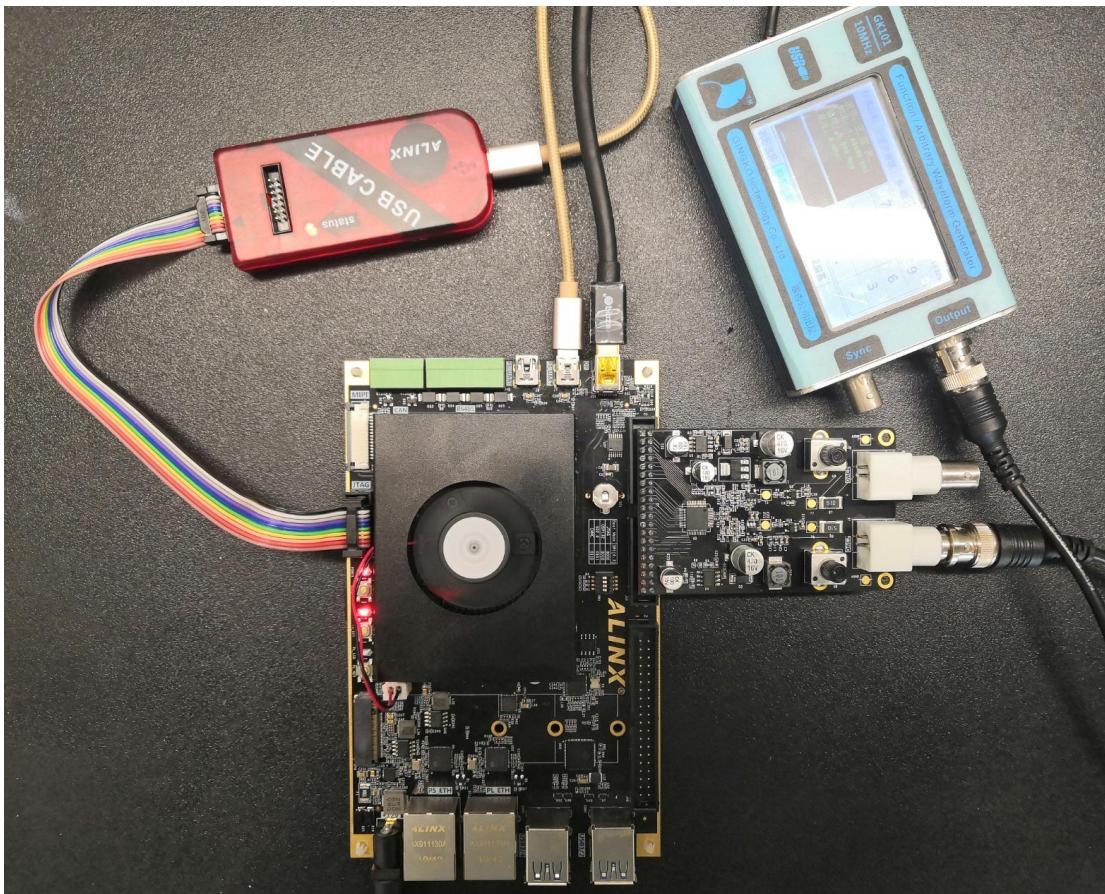


Add “m” in the Libraries of the “Settings” option, click “OK”

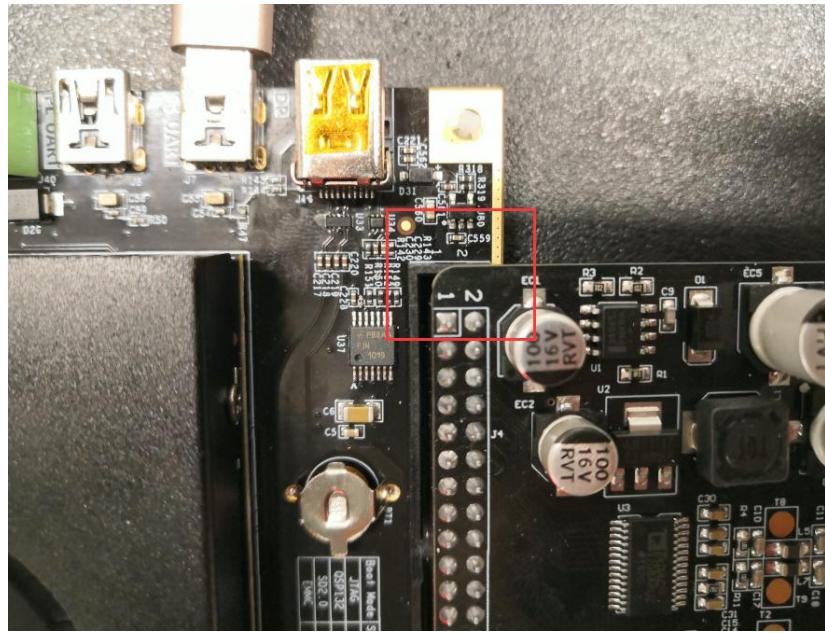


### Part 23.3: Onboard Verification

- 1) Connect the “AN108” to the FPGA development board, connect the waveform generator to the “ADC” interface with the dedicated shielded cable, and connect the HDMI cable. In order to facilitate the observation of the display effect, the waveform generator sampling frequency can be set from “100KHz to 1MHz”, and the voltage amplitude is up to “10V”.



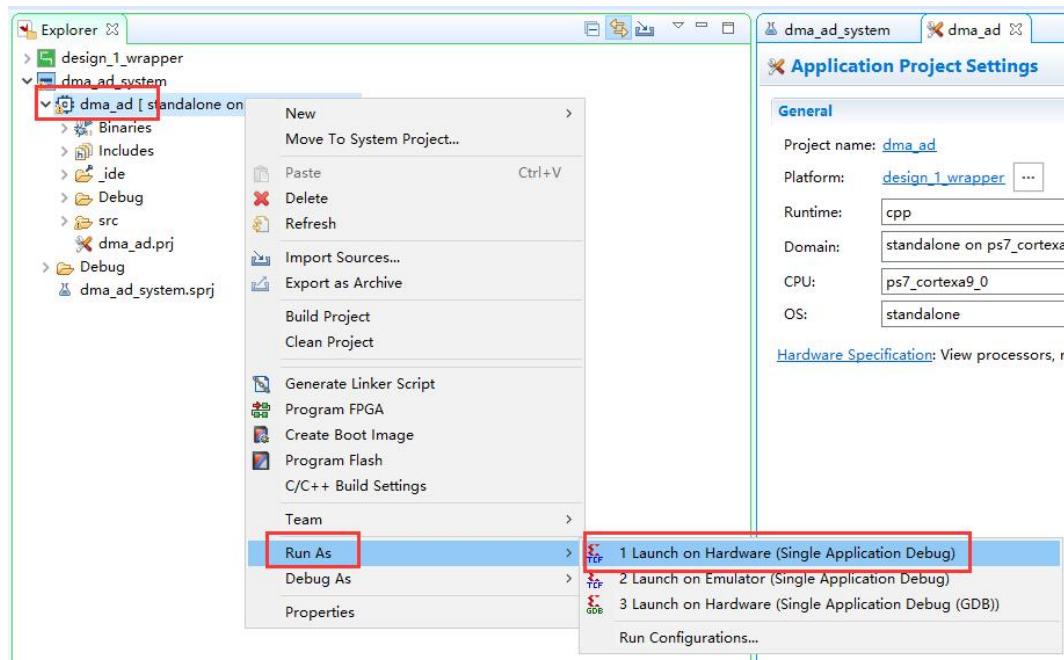
Hardware Connection (Expansion Port J46)

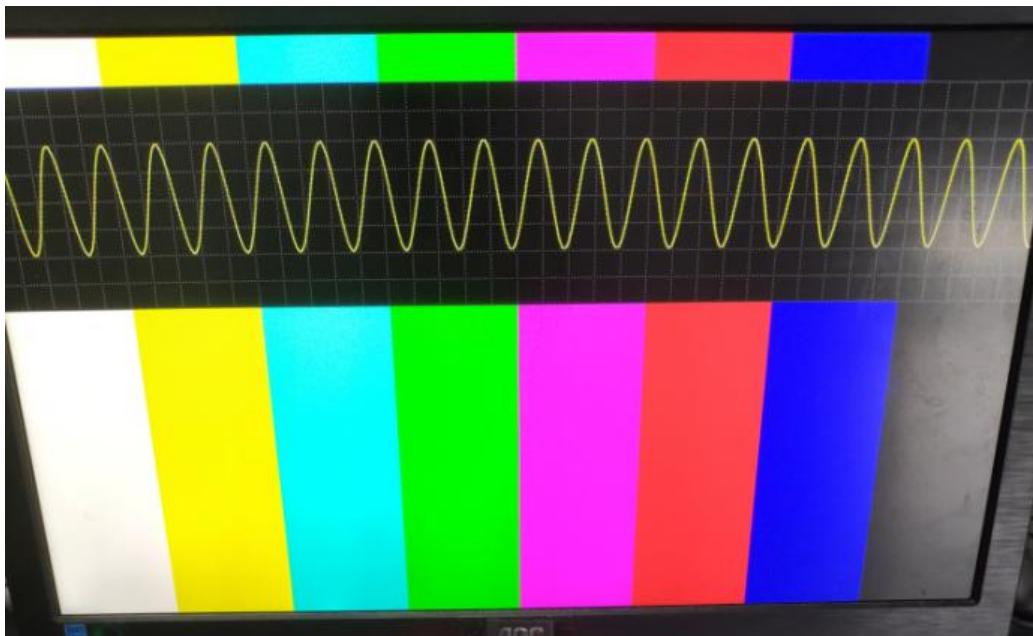


Note that the Pin1 is aligned



- 2) Run Configurations, download the program, you can see the experimental result as below





#### **Part 23.4: Experimental Summary**

This chapter introduces the simple ADC acquisition and display, the overall function is not complicated, users can complete and optimize the function on this basis.

# Part 24: Use of DMA--ADC oscilloscope (AN9238)

The experimental Vivado project directory is "ad9238\_dma\_dp /vivado"

The experimental vitis project directory is "ad9238\_dma\_dp /vitis"

## Part 24.1: Hardware Introduction

### Part 24.1.1: 2-Channel AD Module AN9238

ALINX high-speed AD module AN9238 is a 2-channel 65MSPS, 12-bit analog signal to digital signal module. The AD conversion of the module adopts the AD9238 chip of ADI Company. The AD9238 chip supports 2-channel AD input conversion, so one AD9238 chip supports 2-channel AD input conversion. The analog signal input supports single-ended analog signal input, the input voltage range is -5V~+5V, and the interface is an SMA socket.

The module has a standard 0.1 pitch 40-pin female header for connecting to the FPGA development board



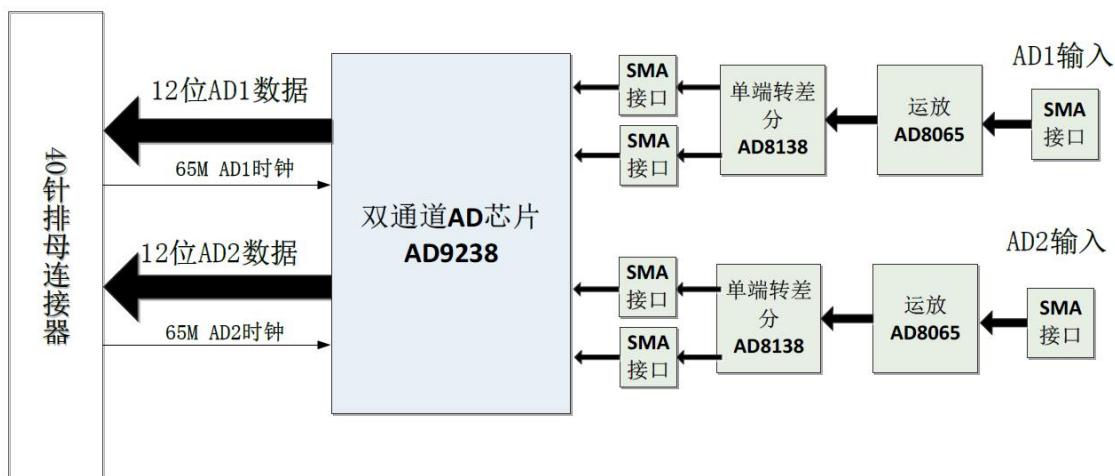
AN9238 Module Product Image

The following are the detailed parameters of the high-speed AD module AN9238:

- AD conversion chip: 1 piece of AD9238
- AD conversion channel: 2 channels;
- AD sampling rate: 65MSPS;
- AD sampling data bits: 12 bits;
- Digital interface level standard: +3.3V CMOS level
- AD analog signal input range: -5V~+5V
- Analog signal input interface: SMA interface
- Measurement accuracy: about 10Mv
- Working temperature: -40°~85°

### Part 24.1.2: AN9238 Module Function Description

AN9238 Module Hardware Block Diagram



For the specific reference design of the AD9238 circuit, please refer to the AD9238 chip manual.

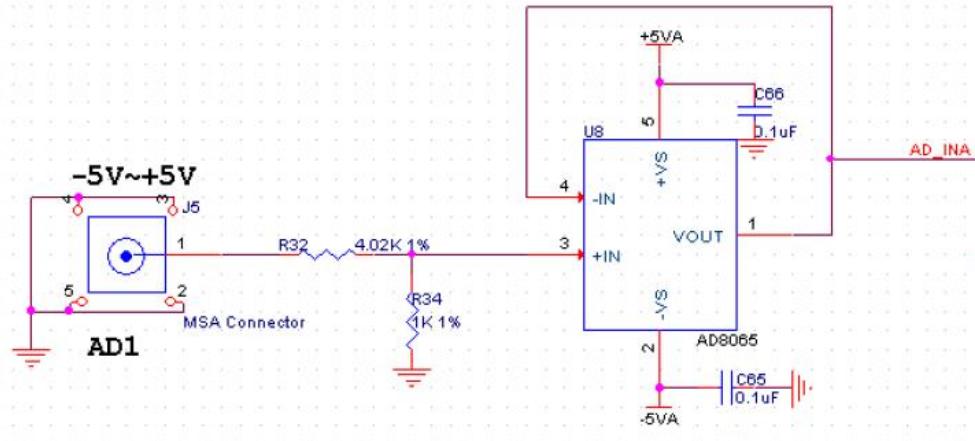
#### 1) Single-ended Input and Operational Amplifier Circuit

The single-ended input AD1 and AD2 are input through two SMA headers, J5 or J6, and the voltage of the single-ended input is -5V~+5V.

On the FPGA development board, the input voltage of -5V~+5V is

reduced to -1V~+1V through the AD8065 chip and voltage divider resistors. If the user wants to input a wider range of voltage, just modify the resistance of the front-end voltage divider resistor.

**Conversion Formula:**  $V_{OUT} = (1.0/5.02) \cdot V_{IN}$

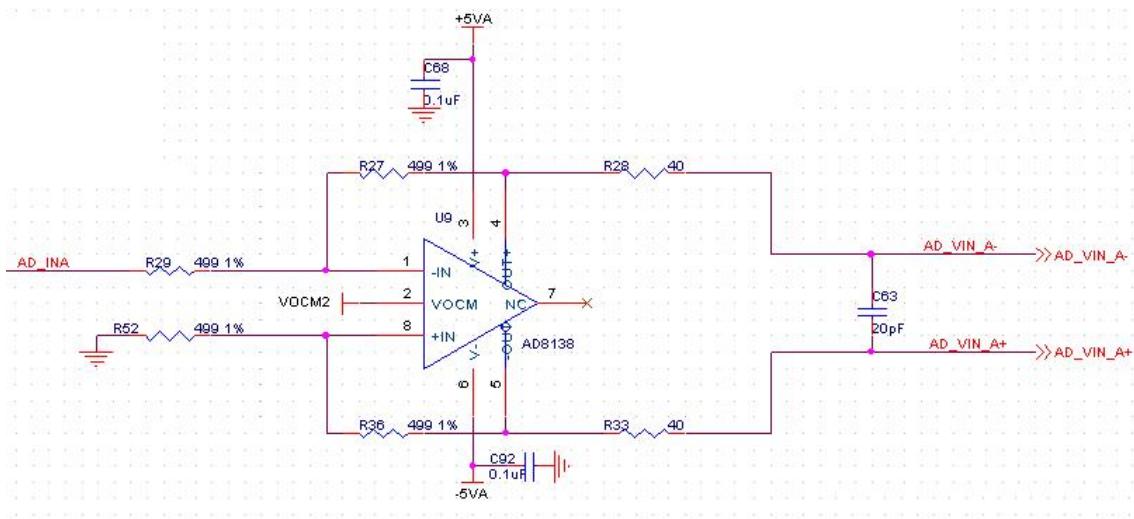


The following table is the voltage comparison table after analog input signal and AD8065 operational amplifier output:

AD Analog Input Value	AD8065 Operational Amplifier Output
-5 V	-1 V
0 V	0 V
+5 V	+1 V

## 2) Single-ended to Differential and AD Conversion

The input voltage of -1V~+1V is converted into a differential signal ( $V_{IN+} - V_{IN-}$ ) by the AD8138 chip, and the common mode level of the differential signal is determined by the CML pin of AD.



The following table is the voltage comparison table after analog input signal to AD8138 differential output:

AD Analog Input Value	AD8065 Operational Amplifier Output	AD8138 Differential Output (VIN+-VIN-)
-5 V	-1 V	-1 V
0 V	0 V	0 V
+5 V	+1 V	+1 V

### 3) AD9238 Conversion

The default AD is configured as offset binary, and the value of AD conversion is shown in the figure below:

Table 16. Output Data Format

Input (V)	Condition (V)	Offset Binary Output Mode
VIN+ – VIN-	< -VREF – 0.5 LSB	0000 0000 0000
VIN+ – VIN-	= -VREF	0000 0000 0000
VIN+ – VIN-	= 0	1000 0000 0000
VIN+ – VIN-	= +VREF – 1.0 LSB	1111 1111 1111
VIN+ – VIN-	> +VREF – 0.5 LSB	1111 1111 1111

In the module circuit design, the VREF value of AD9238 is 1V, so the final analog signal input and AD conversion data are as follows

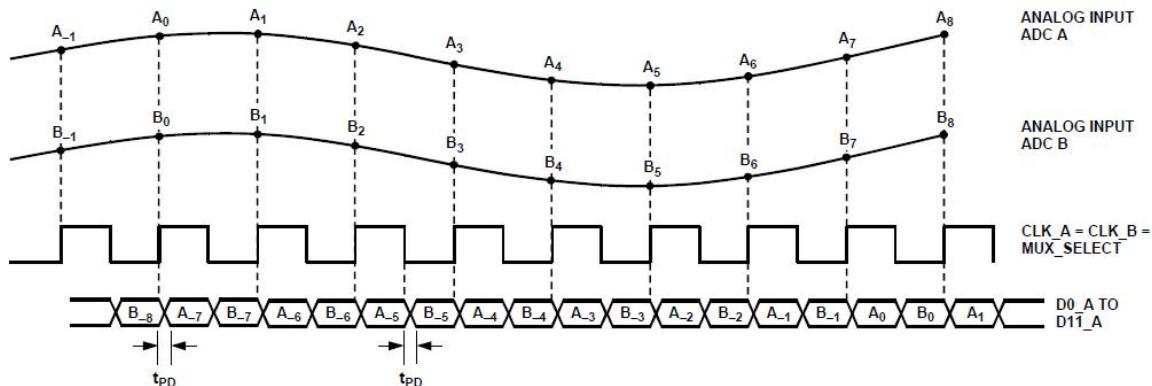
AD Analog Input Value	AD8065 Operational Amplifier Output	AD8138 Differential Output (VIN+-VIN-)	AD9238 Digital Output
-5 V	-1 V	-1 V	000000000000
0 V	0 V	0 V	100000000000
+5 V	+1 V	+1 V	111111111111

From the table, we can see that the digital value converted by AD9238 is the smallest when -5V is input, and the digital value converted by AD9238 is the largest when it is +5V.

#### 4) AN9238 Digital Output Timing

The digital output of AD9238 dual-channel AD is +3.3VCMOS output mode, 2 channels (A and B) independent data and clock. AD data converts data on the rising and falling edges of the clock, and the FPGA end can sample the AD data with the AD clock.

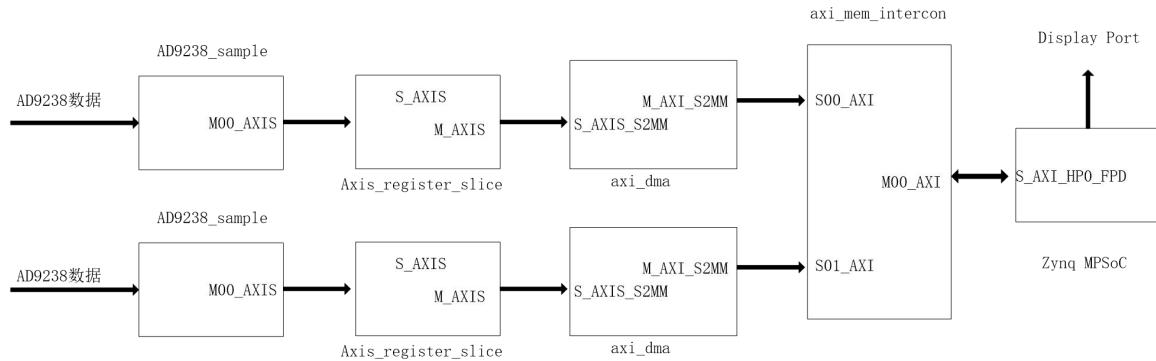
Decoupling capacitors on REF1 and REF2.



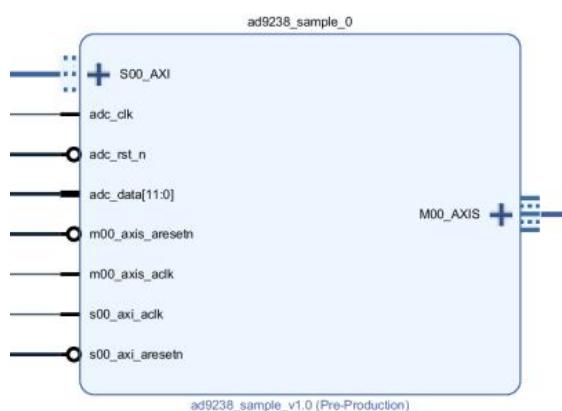
#### FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

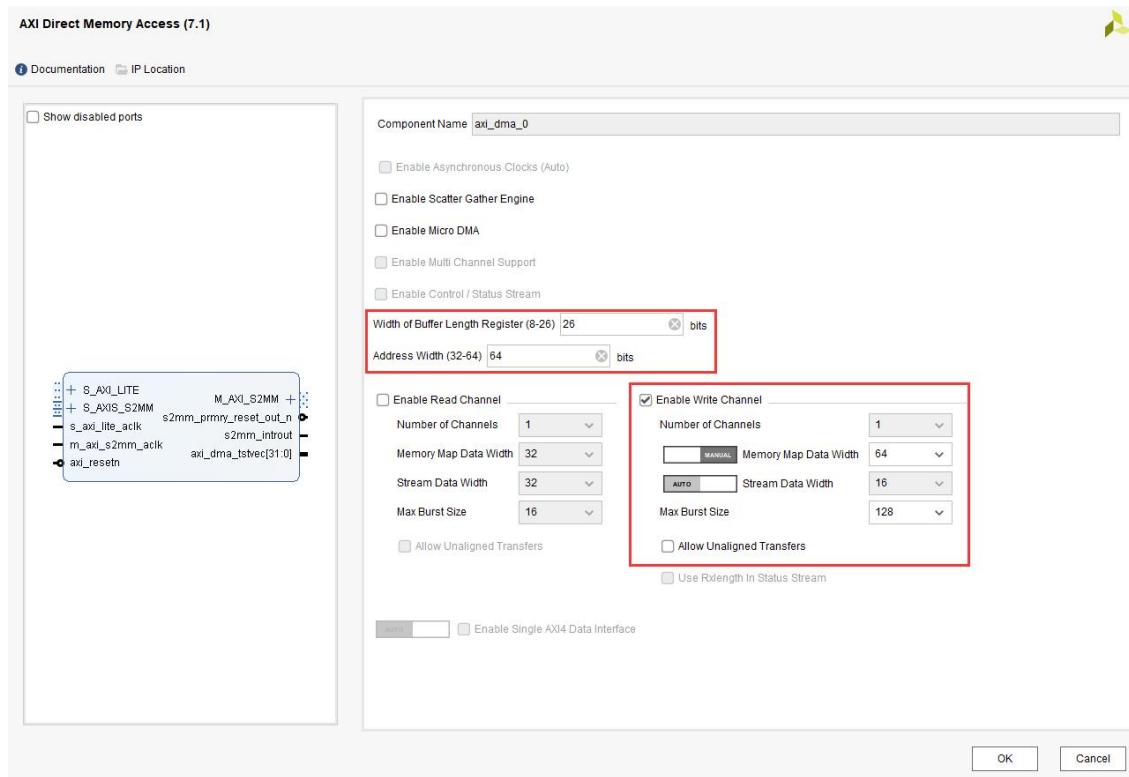
#### Part 24.2: Hardware Environment



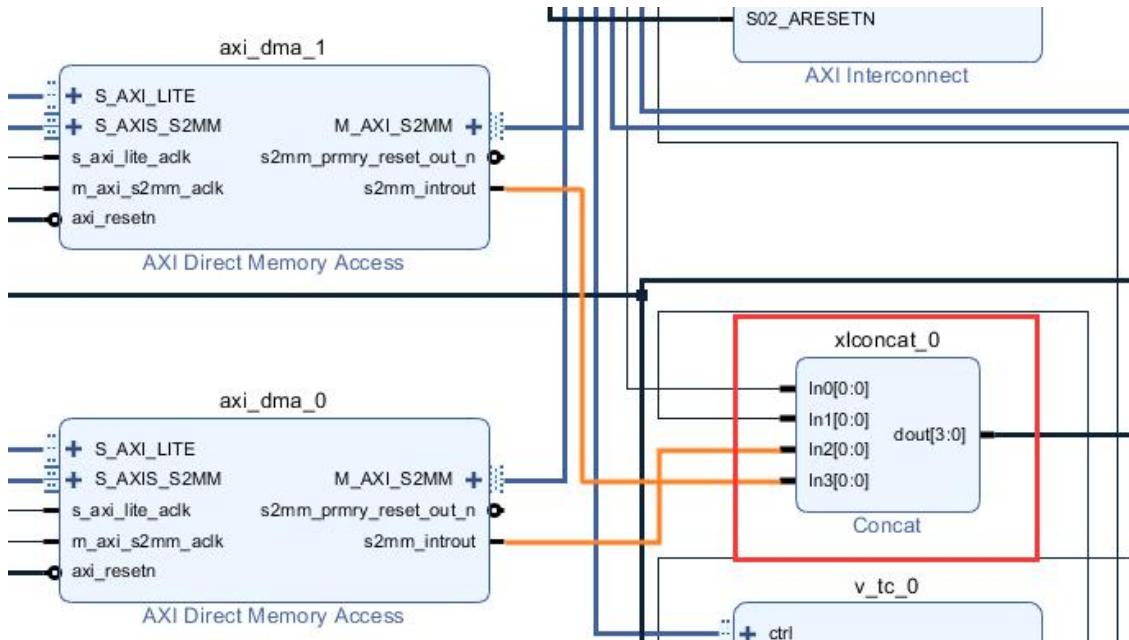
- 1) Based on the “AN108” project to build the hardware environment, delete the “ad9280\_sample” module. Since it is two “AD9238” channels, you need to add two “ad9238\_sample” modules, and the “IP” core is in the “repo” folder.



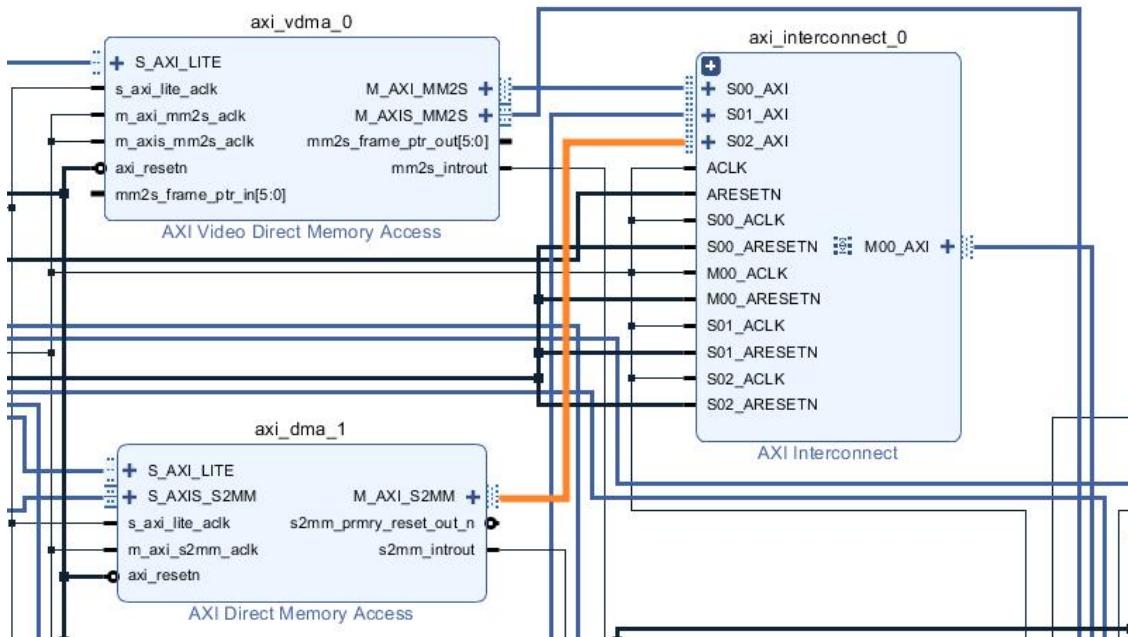
- 2) Add another DMA module, set the Buffer Length of the two DMAs to 26, the Address width to 64, and the Memory Map DataWidth to 64 to increase the bandwidth. The two modules are configured as follows:



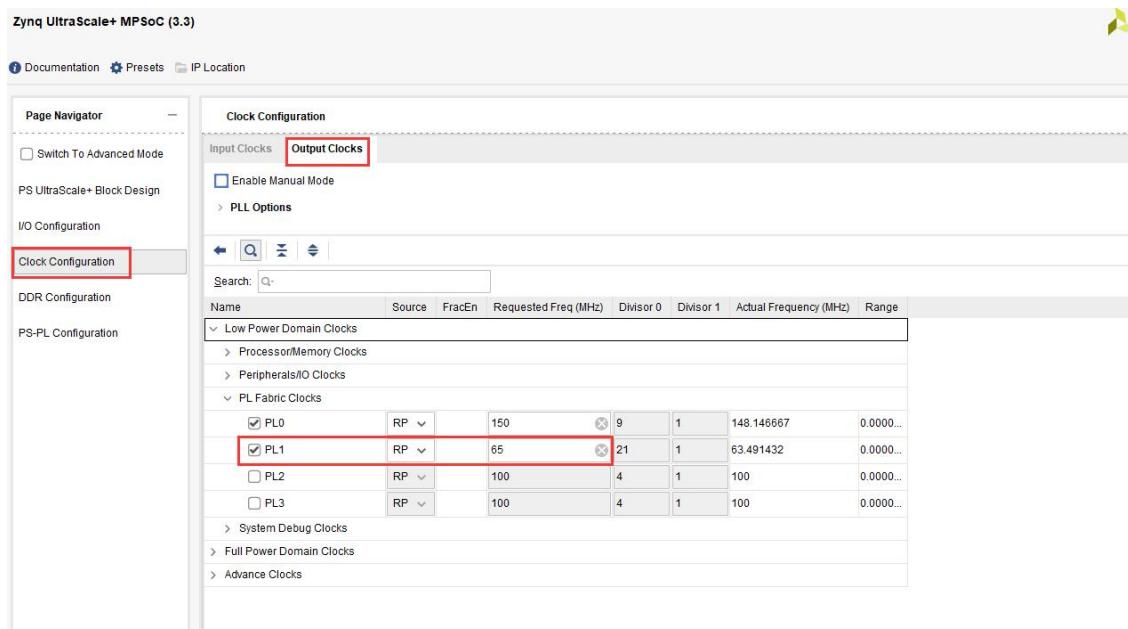
- 3) Add an interface to the “Concat” module and connect to the newly added “dma” interrupt



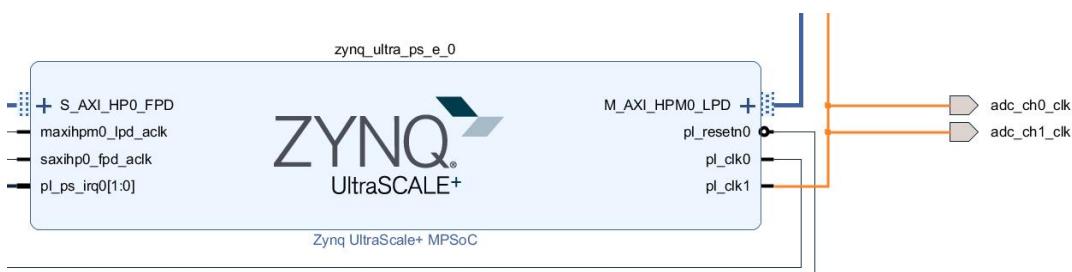
- 4) Add “AXI SLAVE” interface to “AXI Interconnect” module



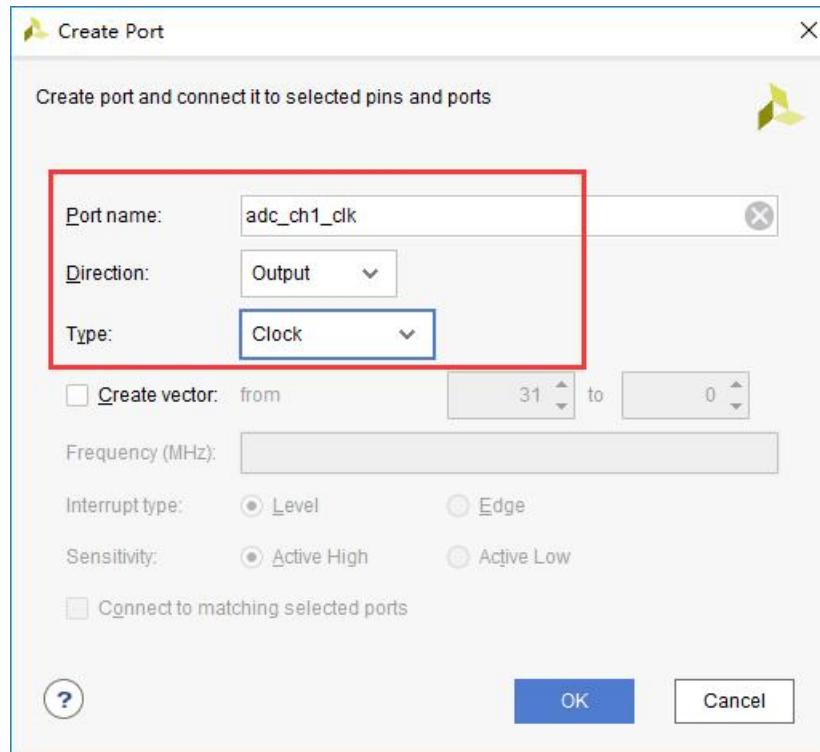
- 5) Modify the clock configuration of the “CPU”, and change “FCLK\_CLK2” to “65MHz” for the ADC clock.



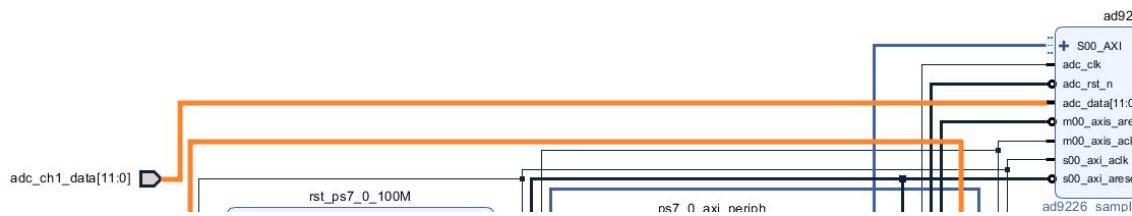
- 6) Take the clock out two ways and provide it to two AD9238 chip



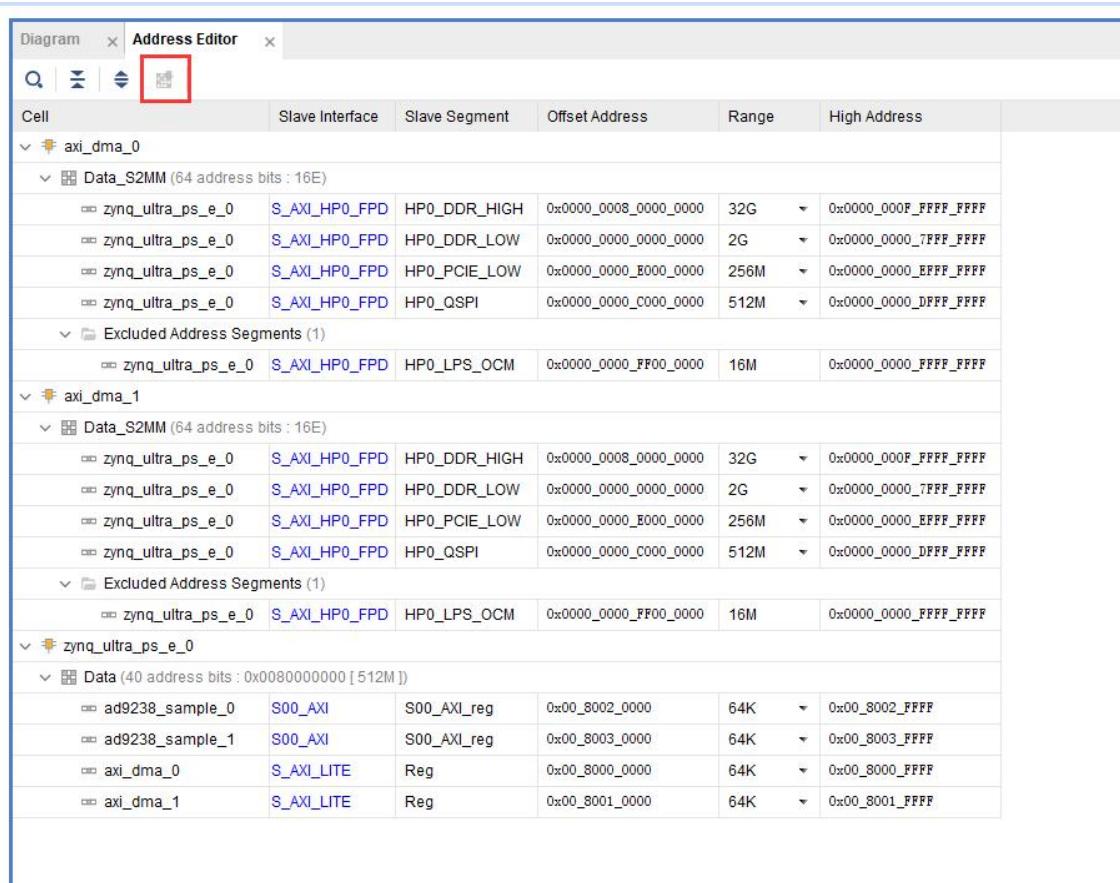
Since “FCLK\_CLK2” can only lead to one pin, you need to right-click on the blank to select “create port”, fill in the pin name, direction, type, and connect the pin to “FCLK\_CLK2”.



- 7) Take the data pin of the “AD9238” and modify the name.



- 8) Connect the rest of the signal, save, click on “Address Editor”, check the address configuration, if some modules do not have an address, click “Auto Assign Address”



The screenshot shows the Xilinx Vivado Address Editor interface. The top menu bar includes 'Diagram' and 'Address Editor'. Below the menu is a toolbar with icons for search, filter, and copy/paste. The main table has columns for 'Cell', 'Slave Interface', 'Slave Segment', 'Offset Address', 'Range', and 'High Address'. The table lists memory regions for various components:

- axi\_dma\_0** (64 address bits : 16E)
  - Data\_S2MM (64 address bits : 16E)
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_DDR\_HIGH 0x0000\_0008\_0000\_0000 32G 0x0000\_000F\_FFFF\_FFFF
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_DDR\_LOW 0x0000\_0000\_0000\_0000 2G 0x0000\_0000\_7FFF\_FFFF
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_PCIE\_LOW 0x0000\_0000\_E000\_0000 256M 0x0000\_0000\_EFFF\_FFFF
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_QSPI 0x0000\_0000\_C000\_0000 512M 0x0000\_0000\_DFFF\_FFFF
  - Excluded Address Segments (1)
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_LPS\_OCM 0x0000\_0000\_FF00\_0000 16M 0x0000\_0000\_FFFF\_FFFF
- axi\_dma\_1** (64 address bits : 16E)
  - Data\_S2MM (64 address bits : 16E)
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_DDR\_HIGH 0x0000\_0008\_0000\_0000 32G 0x0000\_000F\_FFFF\_FFFF
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_DDR\_LOW 0x0000\_0000\_0000\_0000 2G 0x0000\_0000\_7FFF\_FFFF
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_PCIE\_LOW 0x0000\_0000\_E000\_0000 256M 0x0000\_0000\_EFFF\_FFFF
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_QSPI 0x0000\_0000\_C000\_0000 512M 0x0000\_0000\_DFFF\_FFFF
  - Excluded Address Segments (1)
    - zynq\_ultra\_ps\_e\_0 S\_AXI\_HP0\_FPD HP0\_LPS\_OCM 0x0000\_0000\_FF00\_0000 16M 0x0000\_0000\_FFFF\_FFFF
- zynq\_ultra\_ps\_e\_0**
  - Data (40 address bits : 0x0080000000 [ 512M ])
    - ad9238\_sample\_0 S00\_AXI S00\_AXI\_reg 0x00\_8002\_0000 64K 0x00\_8002\_FFFF
    - ad9238\_sample\_1 S00\_AXI S00\_AXI\_reg 0x00\_8003\_0000 64K 0x00\_8003\_FFFF
    - axi\_dma\_0 S\_AXI\_LITE Reg 0x00\_8000\_0000 64K 0x00\_8000\_FFFF
    - axi\_dma\_1 S\_AXI\_LITE Reg 0x00\_8001\_0000 64K 0x00\_8001\_FFFF

- 9) “Generate Output Products” and “Create HDL Wrapper”, bind the “AD9238” pin in the “XDC”, and then generate the “bit” file

## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 24.3: Vitis Program Development

- 1) Similar to the Vitis development of the AN108 experiment, but the need to superimpose two ADC waveforms, defining two DMA receive buffers

```
/* 
 * DMA s2mm receiver buffer
 */
short CH0DmaRxBuffer[MAX_DMA_LEN/sizeof(short)] __attribute__ ((aligned(64)));
short CH1DmaRxBuffer[MAX_DMA_LEN/sizeof(short)] __attribute__ ((aligned(64)));
```

- 2) Set “ADC\_COE” to “16”, “2^4”, that is, convert “12bit” of “AD9226”

to “8bit”, “ADC\_BYTEx” to “2”, “ADC\_BITS” to valid data width, set to “12”, and expand “12bit” data to “16bit” in FPGA program.

```
#define ADC_COE          16           /* ADC coefficient */
#define ADC_BYTEx         2            /* ADC data byte number */
#define ADC_BITS          12
```

When calling the waveform superposition function draw\_wave, the Sign symbol is set to UNSIGNEDSHORT

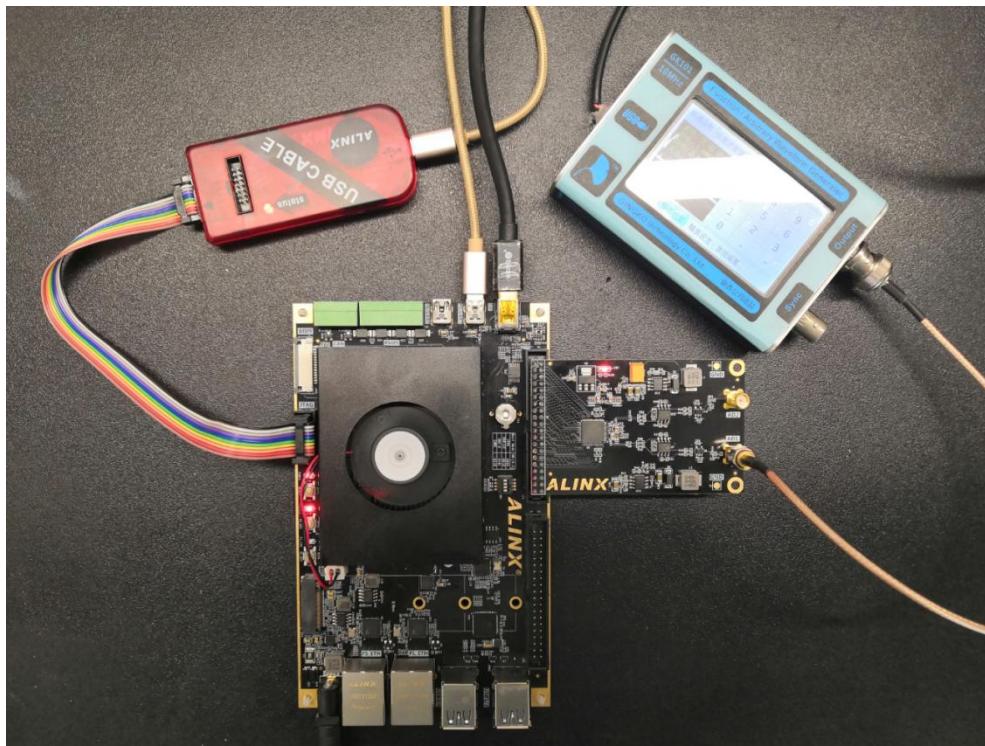
```
/* Copy grid to Wave buffer */
memcpy(WaveBuffer, GridBuffer, WAVE_LEN) ;
/* channel 0 Overlay */
draw_wave(wave_width, WAVE_HEIGHT, (void *)CH0DmaRxBuffer, WaveBuffer, UNSIGNEDSHORT, ADC_BITS, YELLOW, ADC_COE) ;
/* channel 1 Overlay */
draw_wave(wave_width, WAVE_HEIGHT, (void *)CH1DmaRxBuffer, WaveBuffer, UNSIGNEDSHORT, ADC_BITS, RED, ADC_COE) ;

/* Copy Canvas to frame buffer */
frame_copy(wave_width, WAVE_HEIGHT, stride, WAVE_START_COLUMN, WAVE_START_ROW, frame, WaveBuffer) ;
```

Other operations are similar to AN108

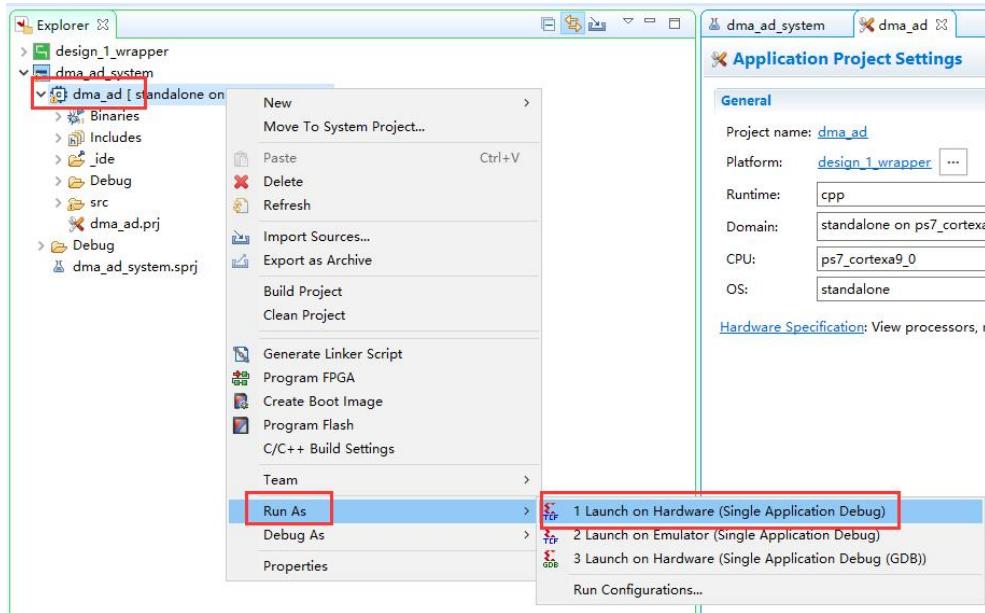
## Part 24.4: Onboard Verification

- 1) Connect the circuit board as shown below, plug the AN9238 module into the expansion port, connect the SMA interface to the waveform generator, connect the HDMI cable to the display, and turn on the power. In order to observe the display effect, the waveform generator sampling frequency is set from 100KHz to 1MHz, and the voltage amplitude is up to 10V.

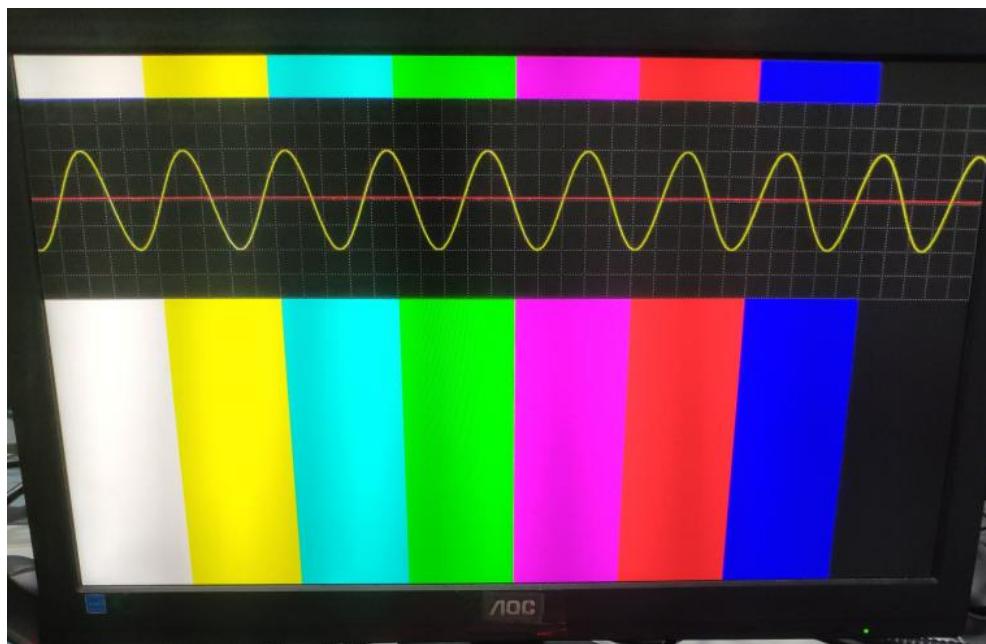


Hardware Connection (Port J46)

2) Run Configurations as below, click “Run” to download the program



3) Experimental result as below

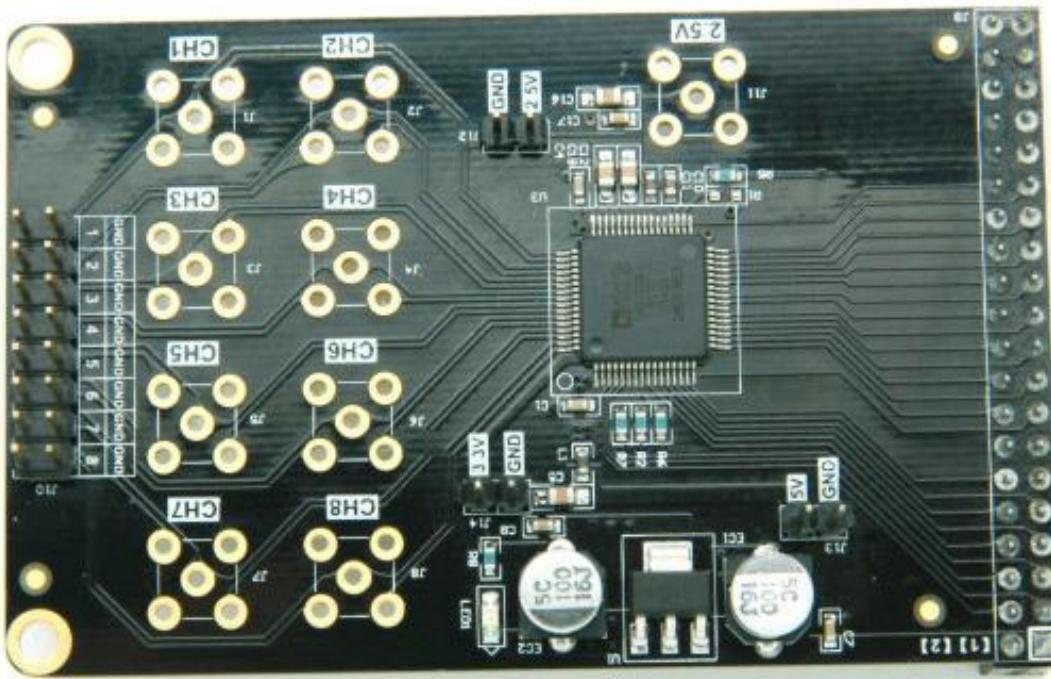


# Part 25: Use of DMA--ADC Oscilloscope (AN706)

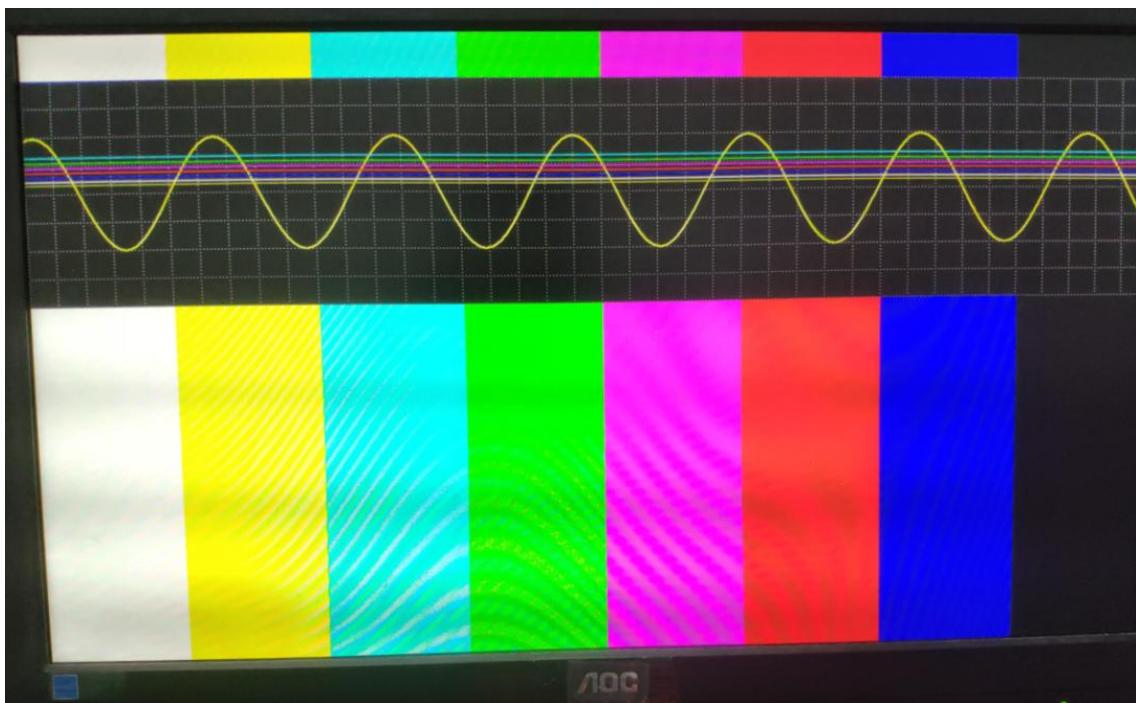
The experimental Vivado project directory is "ad7606\_dma\_dp /vivado".

The experiment vitis project directory is "ad7606\_dma\_dp/vitis".

This experiment uses ADC. The ADC module used in the experiment is AN706, the maximum sampling rate is 200Khz, and the precision is 16 bits. In the experiment, the eight inputs of AN706 are displayed on the VGA in a waveform mode. We can observe the waveform in a more intuitive way, which is a prototype of a digital oscilloscope.



8-channel 200K sampling 16-bit ADC module



Experimental expected effect

### Part 25.1: Experimental Principle

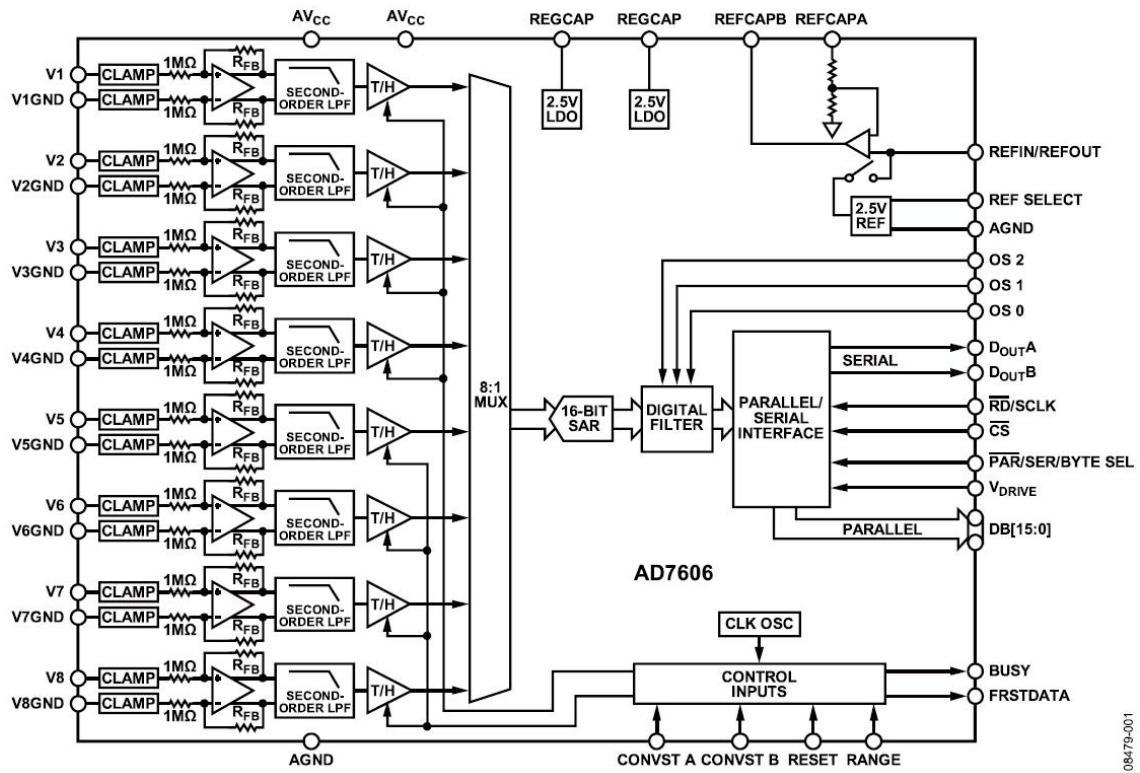
The AD7606 is an integrated 8-channel simultaneous sampling data acquisition system with on-chip integrated input amplifier, overvoltage protection circuitry, second-order analog anti-aliasing filter, analog multiplexer, 16-bit 200 kSPS SAR ADC and a digital Filter, 2.5 V reference, reference buffer, and high speed serial and parallel interfaces.

The AD7606 operates from a single +5V supply and can handle  $\pm 10V$  and  $\pm 5V$  true bipolar input signals, while all channels are sampled at throughput rates up to 200KSPS. The input clamp protection circuit can withstand voltages up to  $\pm 16.5V$ .

The analog input impedance of the AD7606 is 1M ohm regardless of the sampling frequency. It operates from a single supply and features on-chip filtering and high input impedance. There is therefore no need to drive an op amp and an external bipolar supply.

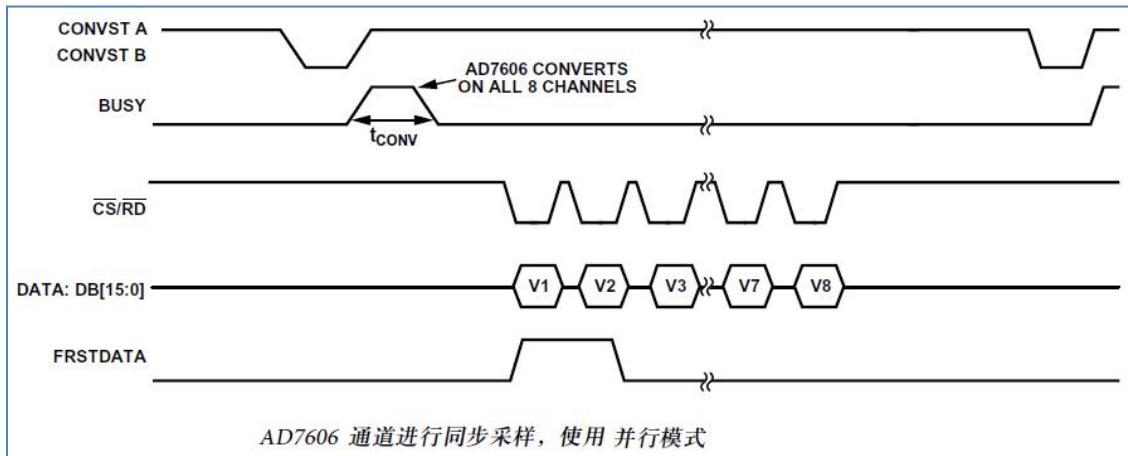
The AD7606 anti-aliasing filter has a 3dB cutoff frequency of

22kHz. It has 40dB anti-aliasing rejection when the sampling rate is 200kSPS. The flexible digital filter is pin-driven to improve signal-to-noise ratio (SNR) and reduce 3dB bandwidth.



08479-001

### Part 25.1.1: AD7606 Timing



The AD7606 can simultaneously sample all eight analog input channels. When two CONVST pins (CONVSTA and CONVSTB) are tied together, all channels are sampled simultaneously. The rising

edge of this shared “CONVST” signal initiates simultaneous sampling (V1 to V8) of all analog input channels.

The AD7606 contains an on-chip oscillator for conversion. The conversion time for all ADC channels is “Tconv”. The BUSY signal informs the user that a conversion is in progress, so when a rising edge of “CONVST” is applied, “BUSY” goes to a logic high level and goes low at the end of the entire conversion process. The falling edge of the “BUSY” signal is used to return all eight sample-and-hold amplifiers to tracking mode. The falling edge of BUSY also indicates that it is now possible to read data from eight channels from the parallel bus DB[15:0].

### Part 25.1.2: AD7606 Configuration

In the AN706 8-channel AD module hardware circuit design, add the pull-up resistor or pull-down resistor to the three configuration pins of the AD7606 to set the working mode of the AD7606.

The AD7606 supports an external reference input or an internal reference. If an external reference is used, the “REFIN/REFOUT” of the chip requires an external 2.5V reference. If using an internal reference voltage. The “REFIN/REFOUT” pin is an internal 2.5V reference. The “REF SELECT” pin is used to select the internal reference or external reference. In this module AN706, because the accuracy of the internal reference voltage of the AD7606 is also very high (2.49V~2.505V), the circuit design chooses to use the internal reference voltage.

Pin	Set Level	Description
REF SELECT	High Level	Use internal reference voltage 2.5V

The AD7606's AD conversion data acquisition can be in parallel

mode or serial mode. The user can set the communication mode by setting the “PAR/SER/BYTE SEL” pin level. When designing, we chose parallel mode to read AD data of AD7606.

Pin	Set Level	Description
PAR/SER/BYTE SEL	Low Level	Select parallel interface

The input range of the AD analog signal of the AD7606 can be set to  $\pm 5V$  or  $\pm 10V$ . When the  $\pm 5V$  input range is set,  $1LSB=152.58\mu V$ ; when the  $\pm 10V$  input range is set,  $1LSB=305.175\mu V$ . The user can set the range of the analog input voltage by setting the RANGE pin level. When designing, we chose an analog voltage input range of  $\pm 5V$ .

Pin	Set Level	Description
RANGE	Low Level	Analog signal input range selection: $\pm 5V$

The AD7606 includes an optional digital first-order sinc filter, which should be used in applications that use lower throughput or require a higher signal-to-noise ratio. The oversampling ratio of the digital filter is controlled by the oversampling pin OS[2:0]. The following table provides oversampling bit decoding to select different oversampling ratios.

表9. 过采样位解码

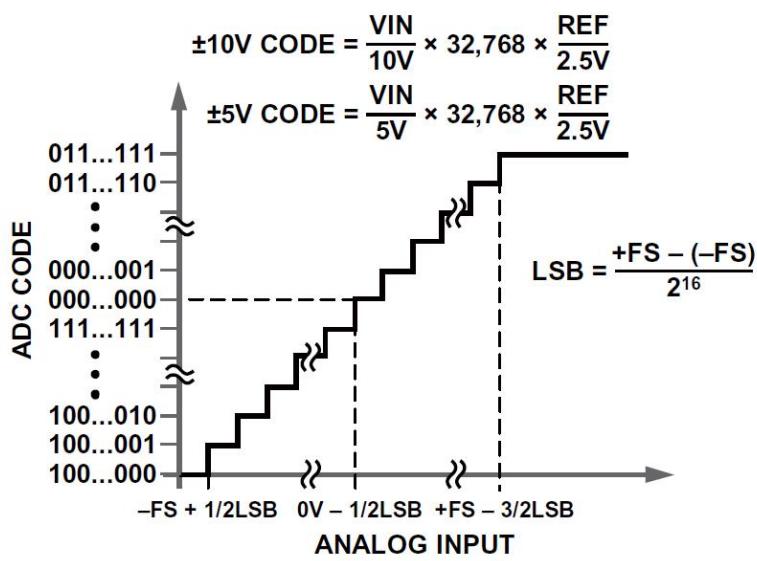
OS[2:0]	过采样倍率	5 V范围SNR(dB)	10 V范围SNR(dB)	5 V范围3 dB带宽(kHz)	10 V范围3 dB带宽(kHz)	最大吞吐量CONVST频率(kHz)
000	No OS	89	90	15	22	200
001	2	91.2	92	15	22	100
010	4	92.6	93.6	13.7	18.5	50
011	8	94.2	95	10.3	11.9	25
100	16	95.5	96	6	6	12.5
101	32	96.4	96.7	3	3	6.25
110	64	96.9	97	1.5	1.5	3.125
111	无效					

In the hardware design of the AN706 module, OS[2:0] has been introduced to the external interface. The FPGA or CPU can control whether to use the filter by controlling the pin level of OS[2:0] to

achieve higher Measurement accuracy.

### Part 25.1.3: AD7606 AD Conversion

The output coding method of the AD7606 is two's complement. The designed code conversion is performed in the middle of consecutive LSB integers (1/2 LSB and 3/2 LSB). The LSB size of the AD7606 is FSR/65536. The ideal transfer characteristics of the AD7606 are shown below:

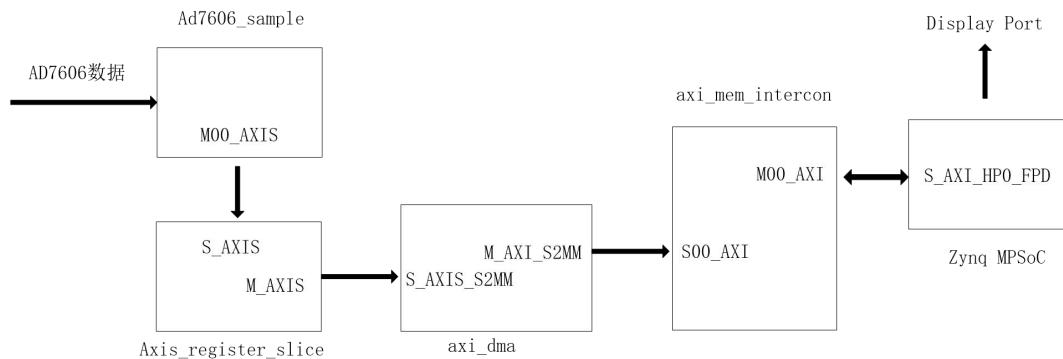


	+FS	MIDSCALE	-FS	LSB
±10V RANGE	+10V	0V	-10V	305µV
±5V RANGE	+5V	0V	-5V	152µV

### FPGA Engineer Job Content

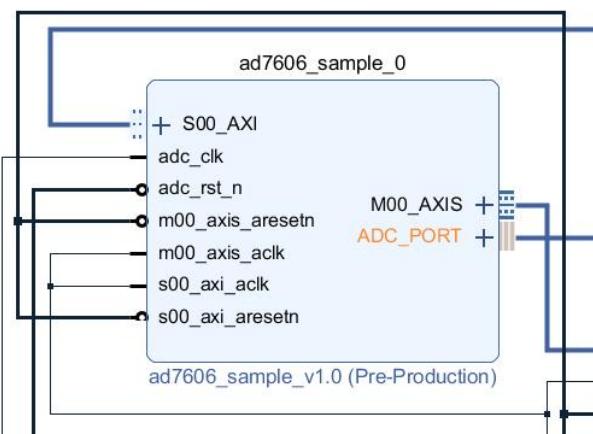
The following is the content that FPGA engineers are responsible for.

## Part 25.2: Hardware Environment

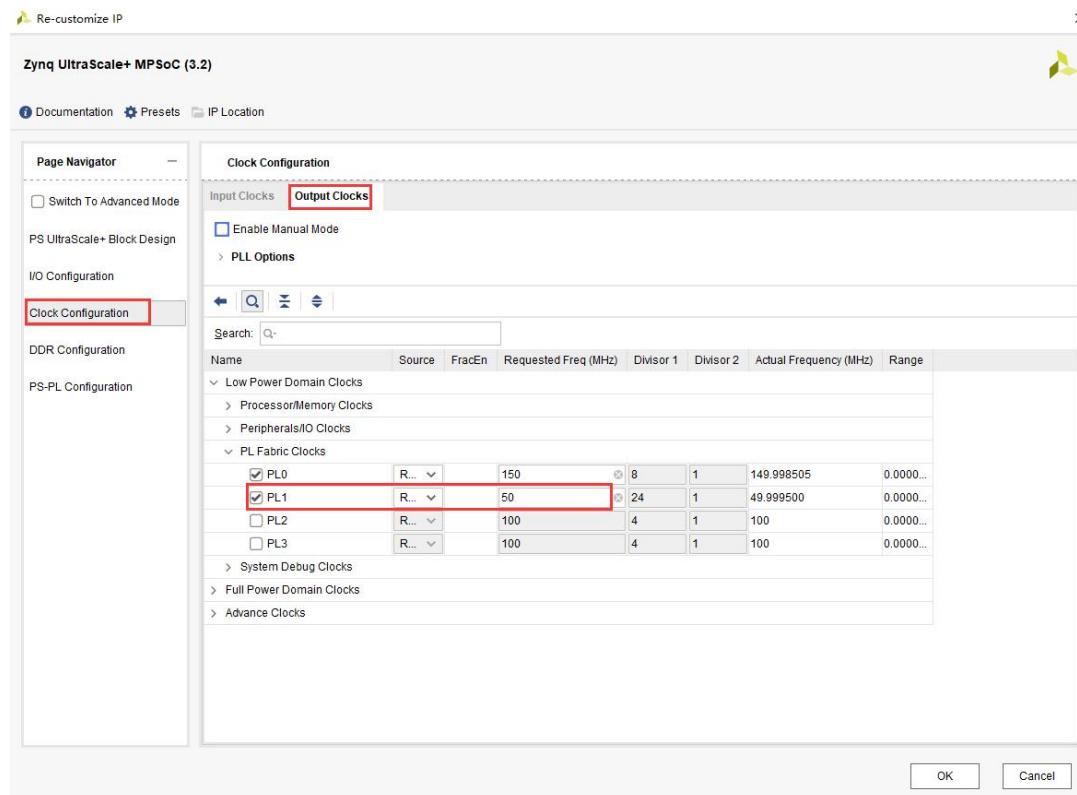


### Part 25.2.1: Hardware Setup

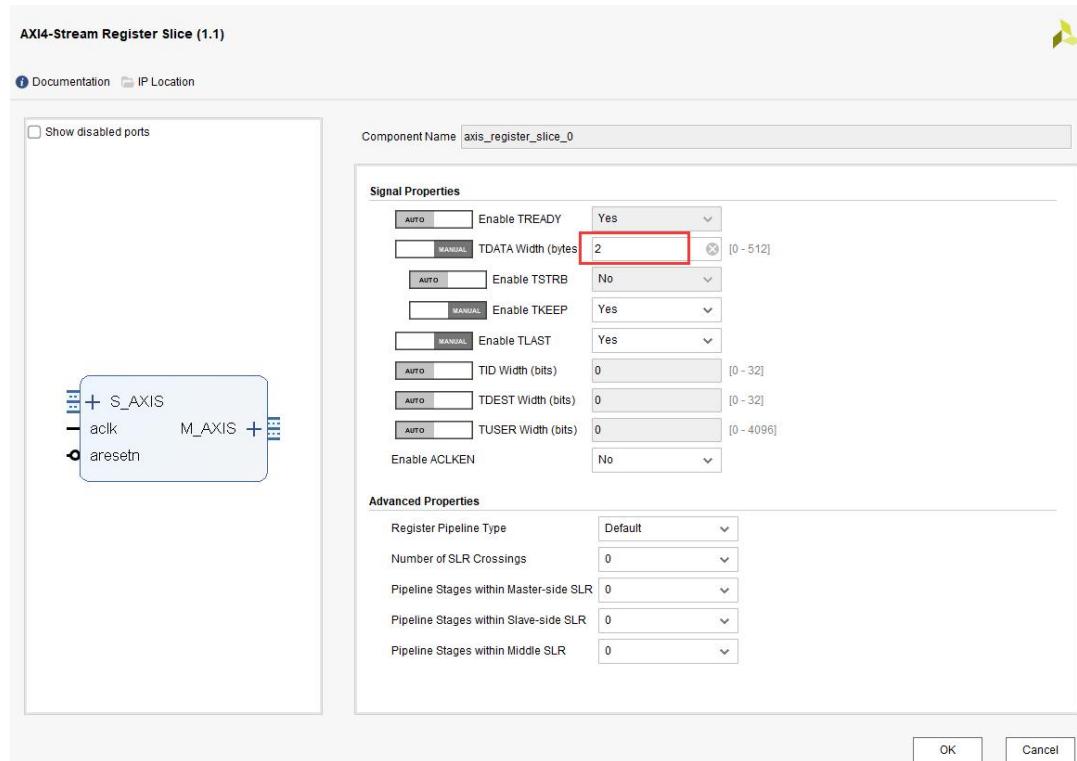
- 1) Based on the “AN108” project, delete the “ad9280\_sample” module, add the “ad7606\_sample” module, and pull out the pins of the “AD7606” to modify the name. The IP core is in the “repo” folder.



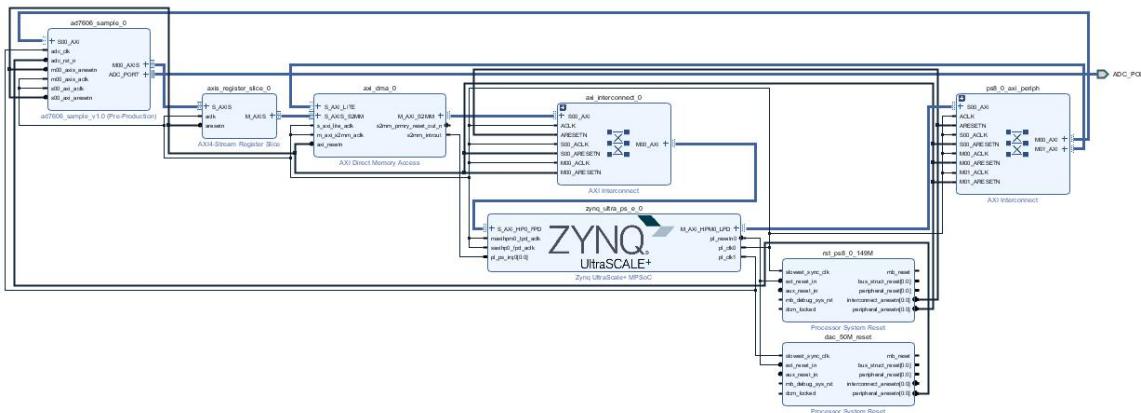
- 2) Modify “FCLK\_CLK2” frequency to “50MHz” for “AD7606” acquisition



- 3) Modify the TDATA width of AXIS Stream Register Slice to 2 because the data width of ad7606 is 16bit. The DMA stream interface width does not need to be modified, and it automatically adapts to the bit width.



- 4) Connection related signal , "Generate Output Products" , Create HDL Wrapper



- 5) Bind the AD7606 pin to the XDC file and generate a bit file.

### Part 25.2.2: ADC Custom IP Function Introduction

Since the data collected by the “ADC” needs to be transferred to “ZYNQ” through “DMA”, the interface with the “DMA” is “AXIS” stream interface, so the “ADC” data needs to be converted into “AXIS” stream data, and the clock of the “ADC” is different from the “AXIS” clock frequency. Therefore, it is necessary to add a “FIFO” for data processing across clock domains. At the same time, you need to implement the “AXIS” Master function. The workflow is:

- 1) The ARM configures the start register and the acquisition length register.
- 2) The 8 channel data is sequentially stored in the FIFO.
- 3) The DMA uses the “AXIS” interface to read the data in the “FIFO” until the configured amount of data is read.

### Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 25.3: SDK Program Development

- 1) Similar to the use of AN108, 8-channel superposition is required.

Since the ADC data is written in the FIFO in the order of “CH1~CH8”, a two-dimensional array is defined, and each channel data is separated, and the order is adjusted in the “XAxisDma\_Adc\_Wave” function.

```
/* Adjust ADC order */
for(i = 0; i < 8 ; i++)
{
    for(j = 0 ; j < width ; j++)
    {
        DmaRxBufferTmp[i][j] = DmaRxBuffer[8*j + i] ;
    }
}
```

- 2) In the “XAxisDma\_ADC” function, in order to be able to see the data of each channel, the coefficient coe of each channel is fine-tuned, so the position of the display is somewhat offset.

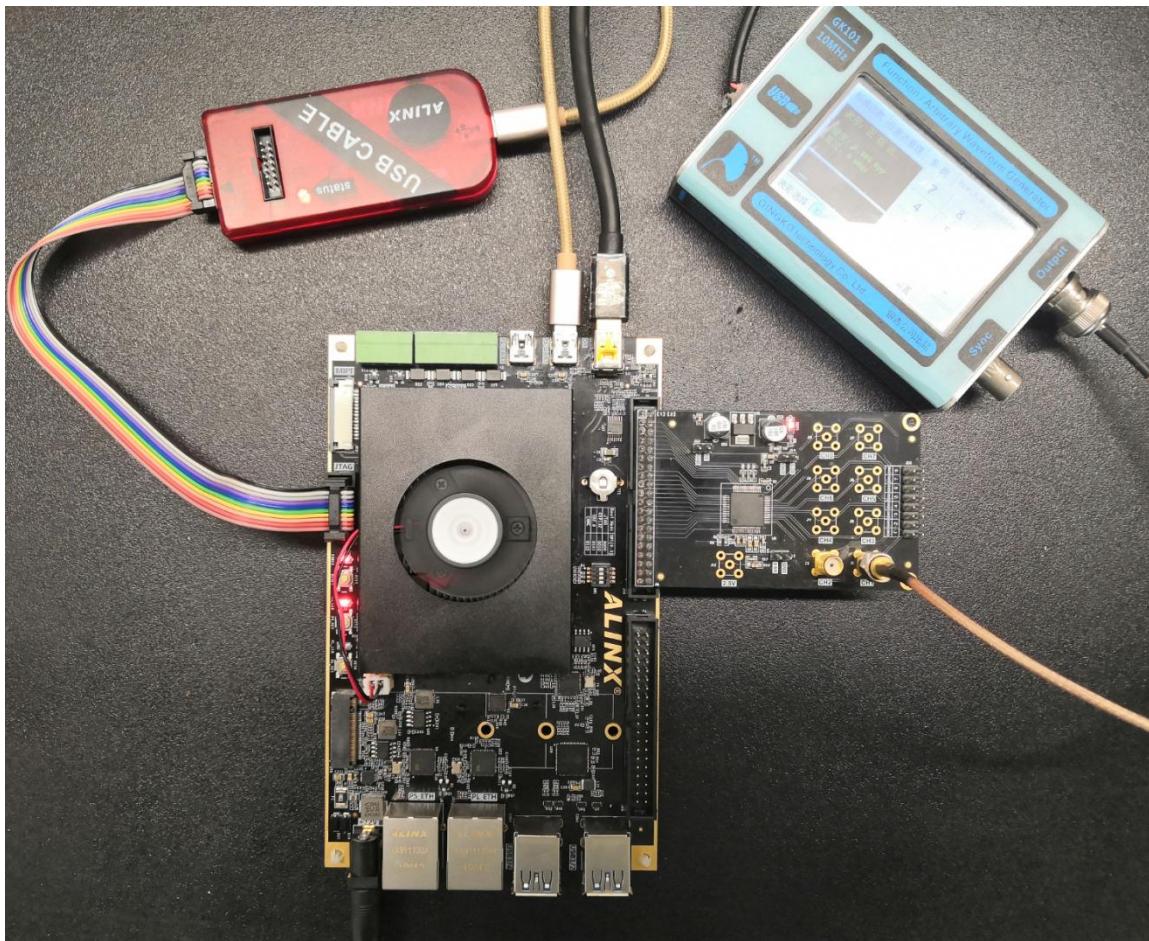
```
/* Wave Overlay */
for(i = 0; i < 8 ; i++)
{
    draw_wave(wave_width, WAVE_HEIGHT, (void *)DmaRxBufferTmp[i], CanvasBuffer, SHORT, ADC_BITS, i, ADC_COE+10*i) ;
}
```

- 3) The parameters of the ADC are defined as follows

```
#define AD7606_BASE          XPAR_AD7606_SAMPLE_0_S00_AXI_BASEADDR
#define AD7606_START           AD7606_SAMPLE_S00_AXI_SLV_REG0_OFFSET
#define AD7606_LENGTH           AD7606_SAMPLE_S00_AXI_SLV_REG1_OFFSET
#define ADC_CAPTURELEN          1920           /* ADC capture length */
#define ADC_COE                  256            /* ADC coefficient */
#define ADC_BYTET                2               /* ADC data byte number */
#define ADC_BITS                 16
#define ADC_CH_COUNT              8
```

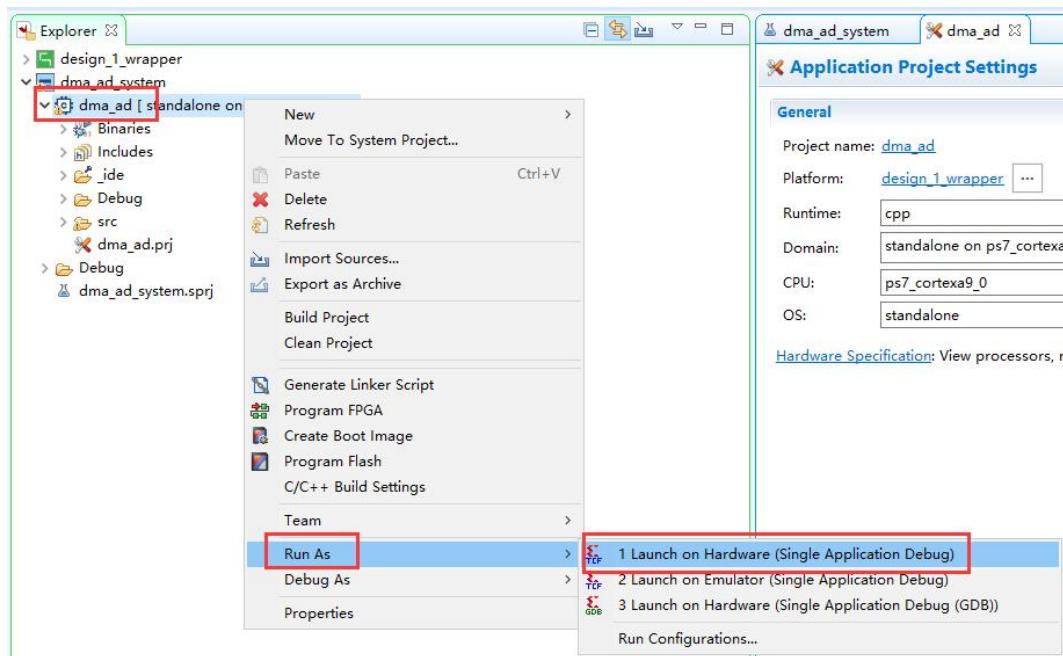
### Part 25.4: Onboard Verification

- 1) Connect the circuit as follows, insert the AN706 module, connect the SMA to the waveform generator, in order to observe the display effect, the waveform generator sampling frequency setting range is 500Hz~10KHz, and the voltage amplitude is up to 10V.

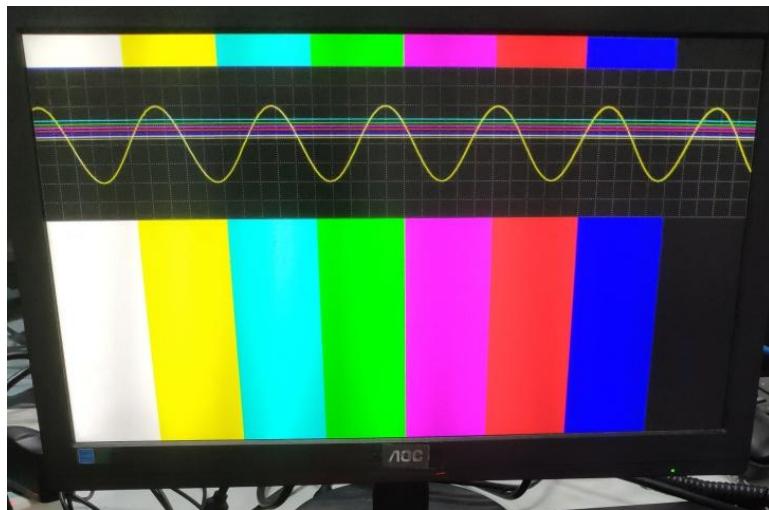


Hardware Connection (Expansion port J46)

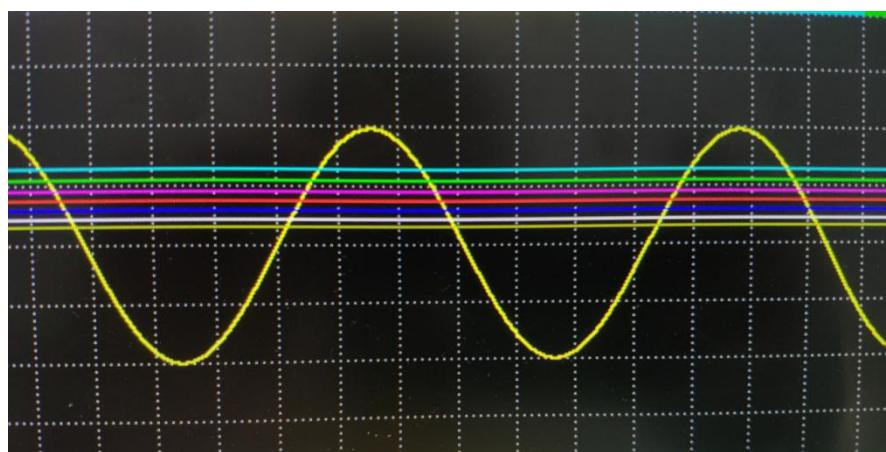
## 2) Download the Program



## 3) Experimental Results



Experimental Results



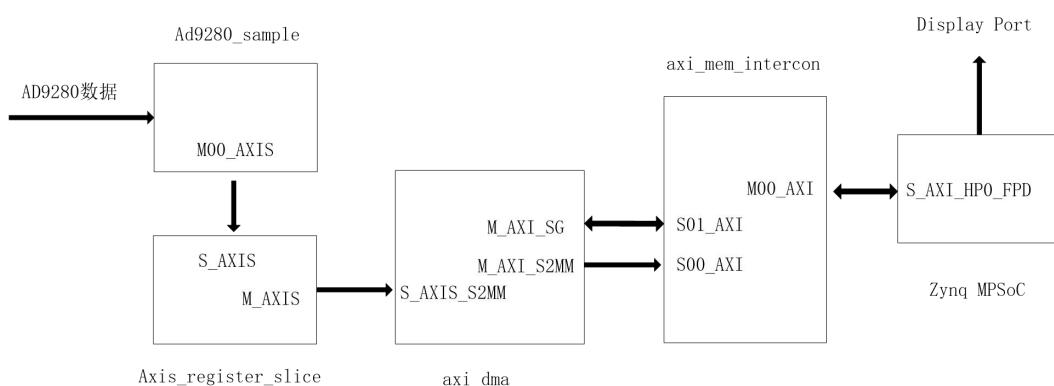
Waveform detail

## Part 26: Use of DMA--ADC Oscilloscope (AN108)

The experimental Vivado project directory is "ad9280\_sg\_dma\_dp /vivado".

The experimental vitis project directory is "ad9280\_sg\_dma\_dp /vitis".

In the previous DMA experiments, the simple DMA mode was used, which is the direct memory access mode. In this chapter, we will learn a more efficient but more complex SG mode, which is Scatter/Gather. The direct access method accesses one memory space at a time, and interrupts to notify the CPU after each transfer, which increases the burden on the CPU and is less efficient. The SG mode allows a single DMA transfer to access multiple memory spaces. After all tasks are completed , The interrupt is issued, which improves efficiency.

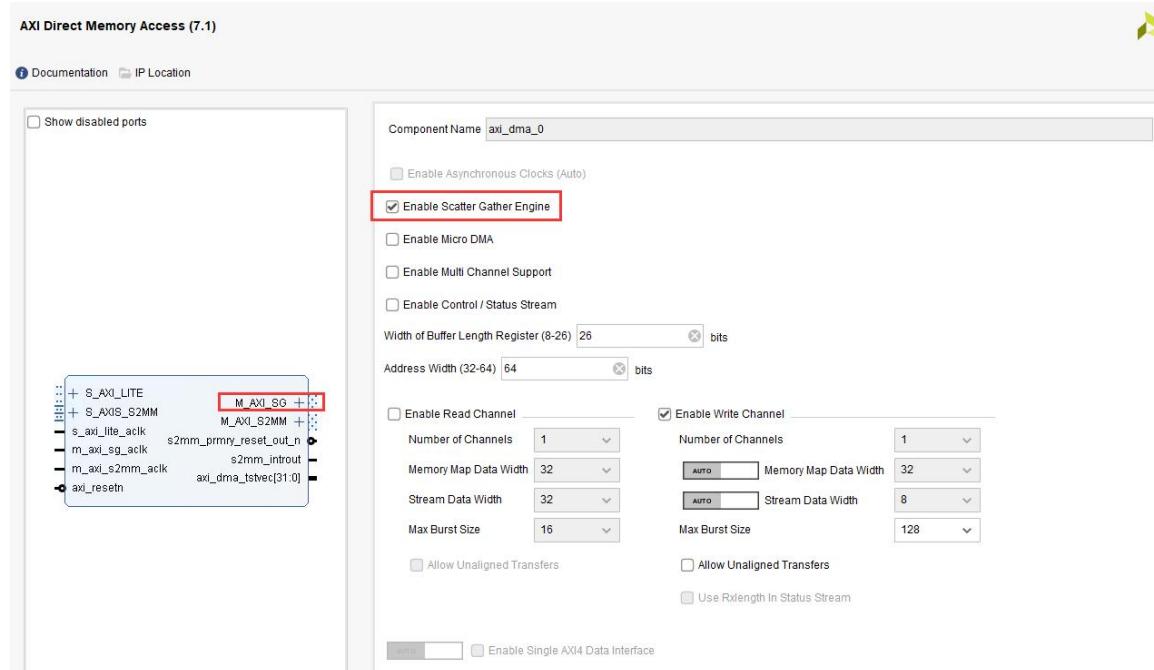


Block Diagram

## FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

Based on the AN108 project, two DMAs need to be configured, Enable Scatter Gather Engine, then the M\_AXI\_SG interface will appear for reading and writing the linked list (will be introduced later).



## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 26.1: Introduction to SG DMA Principle

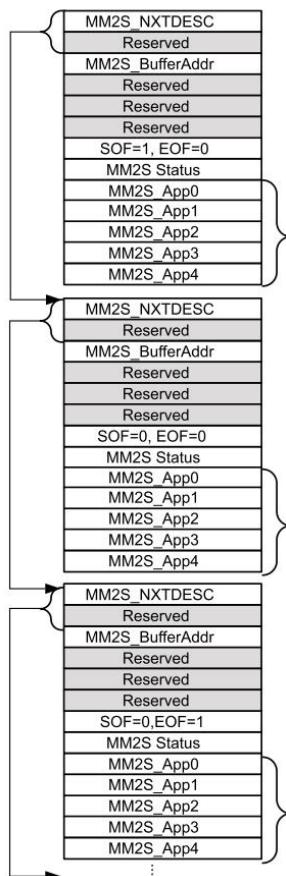
- 1) First to understand the linked list, you need to open a space in the memory to save the linked list, the contents of the linked list are as follows, “Descriptor” as the basic unit, each Descriptor has 13 registers, but each “Descriptor” address needs to be aligned in 64 bytes, which is “0x00”, “0x40”, “0x80”; “NXTDESC” represents the next descriptor pointer address, “BUFFER\_ADDRESS” is the

address of the data cache, in this experiment represents the “ADC” data buffer space received. Control stores the start and end of the Frame, the length information of each Pachage (in bytes), the Status stores the error, the end and other information, and “APP0~APP4” are the user space (only valid when the “Status Control Stream” is opened, otherwise the “DMA” does not Will grab this information).

Table 2-26: Descriptor Fields (Non-multichannel Mode)

Address Space Offset <sup>(1)</sup>	Name	Description
00h	NXTDESC	Next Descriptor Pointer
04h	NXTDESC_MSB	Upper 32 bits of Next Descriptor Pointer
08h	BUFFER_ADDRESS	Buffer Address
0Ch	BUFFER_ADDRESS_MSB	Upper 32 bits of Buffer Address.
10h	RESERVED	N/A
14h	RESERVED	N/A
18h	CONTROL	Control
1Ch	STATUS	Status
20h	APP0	User Application Field 0 <sup>(2)</sup>
24h	APP1	User Application Field 1
28h	APP2	User Application Field 2
2Ch	APP3	User Application Field 3
30h	APP4	User Application Field 4

The following figure shows the linked list structure of “MM2S”. After the “SG DMA” is started, the “DMA” will fetch the first Descriptor through “M\_AXI\_SG”, read the “BufferAddr”, “Control” and other information, and then transfer the next Descriptor information after the length of the control setting is transmitted. And so on, until the last Descriptor.



- 2) It is important to note that the “MX2S\_CONTRL” “TXSOF” (Frame Start) and “TXEOF” (Frame End) require software settings.

Table 2-31: MM2S\_CONTROL Details

Bits	Field Name	Description
25 to 0	Buffer Length	<p>Indicates the size in bytes of the transfer buffer. This value indicates the amount of bytes to transmit out on the MM2S stream. The usable width of buffer length is specified by the parameter <b>Width of Buffer Length Register</b>. A maximum of 67,108,863 bytes of transfer can be described by this field. When configuring the AXI_DMA in Micro mode, this value should not exceed the following equation:</p> $(\text{MM2S Memory Mapped Data width}/8)*\text{Burst\_length}$ <p><b>Note:</b> Setting the buffer length register width smaller than 26 reduces FPGA resource utilization.</p>
26	Transmit End Of Frame (TXEOF)	<p>End of Frame. Flag indicating the last buffer to be processed. This flag is set by the CPU to indicate to AXI DMA that this descriptor describes the end of the packet. The buffer associated with this descriptor is transmitted last.</p> <ul style="list-style-type: none"> <li>• 0 = Not End of Frame.</li> <li>• 1 = End of Frame.</li> </ul> <p><b>Note:</b> For proper operation, there must be a Start of Frame (SOF) descriptor (TXSOF=1) and an End of Frame (EOF) descriptor (TXEOF=1) per packet. It is valid to have a single descriptor describe an entire packet that is a descriptor with both TXSOF=1 and TXEOF=1.</p>
27	TXSOF	<p>Start of Frame. Flag indicating the first buffer to be processed. This flag is set by the CPU to indicate to AXI DMA that this descriptor describes the start of the packet. The buffer associated with this descriptor is transmitted first.</p> <ul style="list-style-type: none"> <li>• 0 = Not start of frame.</li> <li>• 1 = Start of frame.</li> </ul> <p><b>Note:</b> When Status Control Stream is enabled, user application data from APP0 to APP4 of the Start of Frame (SOF) descriptor (TXSOF=1) is transmitted on the control stream output.</p>
31 to 28	Reserved	This bit is reserved and should be written as zero.

The “RXSOF” and “RXEOF” of “S2MM\_CTRL” do not require software settings and are controlled by the “DMA” after receiving data.

Table 2-38: S2MM\_CONTROL Details

Bits	Field Name	Description
31 to 28	Reserved	These bits are reserved and should be set to zero.
27	RXSOF	<p>Start of Frame. Flag indicating the first buffer to be processed. This flag is set by the sw/user to indicate to AXI DMA that this descriptor describes the start of the packet. The buffer associated with this descriptor is received first.</p> <ul style="list-style-type: none"> <li>• 0 = Not End of Frame.</li> <li>• 1 = End of Frame.</li> </ul> <p>This is applicable only when AXI_DMA is configured in Micro mode.</p>
26	Receive End Of Frame	<p>End of Frame. Flag indicating the last buffer to be processed. This flag is set by the sw/user to indicate to AXI DMA that this descriptor describes the end of the packet. The buffer associated with this descriptor is received last.</p> <ul style="list-style-type: none"> <li>• 0 = Not End of Frame.</li> <li>• 1 = End of Frame.</li> </ul> <p>This is applicable only when AXI_DMA is configured in Micro mode.</p>
25 to 0	Buffer Length	<p>This value indicates the amount of space in bytes available for receiving data in an S2MM stream. The usable width of buffer length is specified by the parameter <b>Width of Buffer Length Register</b>. A maximum of 67,108,863 bytes of transfer can be described by this field.</p> <p><b>Note:</b> The total buffer space in the S2MM descriptor chain (that is, the sum of buffer length values for each descriptor in a chain) must be, at a minimum, capable of holding the maximum receive packet size. Undefined results occur if a packet larger than the defined buffer space is received.</p> <p><b>Note:</b> Setting the Buffer Length Register Width smaller than 23 reduces FPGA resource utilization.</p> <p><b>Note:</b> When configuring the AXI_DMA in Micro mode, this value should not exceed the following equation:  <math>(\text{S2MM Memory Mapped Datawidth}/8)*\text{Burst\_length}</math></p>

- 3) Looking at the “SG DMA” register, “MM2S\_CURDESC” indicates the current “Descriptor” address, and “MM2S\_TAILDESC” indicates the tail Descriptor address, which is the last Descriptor. Similarly, “S2MM\_CURDESC” and “S2MM\_TAILDESC” refer to the S2MM path register. These registers are configured via the “AXI4\_LITE” bus.

Table 2-5: Scatter / Gather Mode Register Address Map

Address Space Offset <sup>(1)</sup>	Name	Description
00h	MM2S_DMCR	MM2S DMA Control register
04h	MM2S_DMASR	MM2S DMA Status register
08h	MM2S_CURDESC	MM2S Current Descriptor Pointer. Lower 32 bits of the address.
0Ch	MM2S_CURDESC_MSB	MM2S Current Descriptor Pointer. Upper 32 bits of address.
10h	MM2S_TAILDESC	MM2S Tail Descriptor Pointer. Lower 32 bits.
14h	MM2S_TAILDESC_MSB	MM2S Tail Descriptor Pointer. Upper 32 bits of address.
2Ch <sup>(2)</sup>	SG_CTL	Scatter/Gather User and Cache
30h	S2MM_DMCR	S2MM DMA Control register
34h	S2MM_DMASR	S2MM DMA Status register
38h	S2MM_CURDESC	S2MM Current Descriptor Pointer. Lower 32 address bits
3Ch	S2MM_CURDESC_MSB	S2MM Current Descriptor Pointer. Upper 32 address bits.
40h	S2MM_TAILDESC	S2MM Tail Descriptor Pointer. Lower 32 address bits.
44h	S2MM_TAILDESC_MSB	S2MM Tail Descriptor Pointer. Upper 32 address bits.

#### 4) SG DMA usage process

##### MM2S Side:

- 1) Open up cache space in memory, make linked list
- 2) Write the address of the first “Descriptor” to the “MM2S\_CURDESC” register via the “AXI4\_LITE” bus
- 3) Write register “MM2S\_DMCR.RS=1” to start DMA, if necessary, open “MM2S\_DMCR.IOC\_IrqEn” interrupt, will issue an interrupt after the end
- 4) Write the last “Descriptor” address to the “MM2S\_TAILDESC” register, trigger the “DMA” to start the Descriptor of the linked list through the “M\_AXI\_SG” bus, wait for the package transmission to end, and read the next Descriptor information until the end.

##### S2MM Side:

- 1) Open up cache space in memory, make linked list
- 2) Write the address of the first “Descriptor” to the

“S2MM\_CURDESC” register via the “AXI4\_LITE” bus

- 3) Write register “S2MM\_DMACR.RS=1” to start DMA, if necessary, open “S2MM\_DMACR.IOC\_IrqEn” interrupt, will issue an interrupt after the end
- 4) Write the last “Descriptor” address to the “S2MM\_TAILDESC” register, trigger the “DMA” to start the Descriptor of the linked list through the “M\_AXI\_SG” bus, wait for the package transmission to end, and read the next Descriptor information until the end.

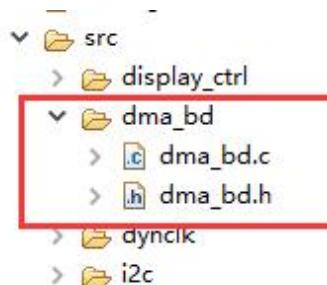
At this point, maybe everyone will wonder how the list should be produced. The following is an introduction through the Vitis program.

## Part 26.2: Vitis Program Development

- 1) Based on the AN108 experiment, two linked list arrays are defined, each linked to a maximum of 16 Descriptors, and the “BD\_ALIGNMENT” macro is defined as 0x40.

```
/*  
 * BD buffer  
 */  
u32 BdChainBuffer[BD_ALIGNMENT*16] __attribute__ ((aligned(64)));
```

- 2) Added two files “dma\_bd.c” and “dma\_bd.h”



“CreateBdChain” is to create a linked list function, “BdCount” is the number of “Descriptors” to be created, “TotalByteLen” is the total number of bytes transferred by “DMA”

```
/*  
int CreateBdChain(u32 *BdDesptr, u16 BdCount, u32 TotalByteLen, u8 *DmaBufferPtr, u32 Direction)  
{
```

Only need to configure “NEXDESC”, “BUFFER\_ADDRESS”, “CONTROL” three parts, if “TXPATH” need to set “TXSOF” and “TXEOP”, this experiment is “RXPATH”, no need to set.

```
/* Write to Next descriptor point Lower 32bit */
XAxiDma_BdWrite(BdPtrCurr, XAXIDMA_BD_NDESC_OFFSET, (u32)BdPtrNext & XAXIDMA_DESC_LSB_MASK) ;
/* Write to Buffer Address Lower 32bit */
XAxiDma_BdWrite(BdPtrCurr, XAXIDMA_BD_BUFA_OFFSET, (u32)RxBufCurr) ;
/* Write to length field */
if (Direction == TXPATH)
{
    if (i == 0)
        XAXIDMA_BdWrite(BdPtrCurr, XAXIDMA_BD_CTRL_LEN_OFFSET, (u32)(Len/Count) | XAXIDMA_BD_CTRL_TXSOF_MASK) ;
    else if (i == Count - 1)
        XAXIDMA_BdWrite(BdPtrCurr, XAXIDMA_BD_CTRL_LEN_OFFSET, (u32)(Len/Count) | XAXIDMA_BD_CTRL_TXEOP_MASK) ;
    else
        XAXIDMA_BdWrite(BdPtrCurr, XAXIDMA_BD_CTRL_LEN_OFFSET, (u32)(Len/Count) ) ;
}
else
    XAXIDMA_BdWrite(BdPtrCurr, XAXIDMA_BD_CTRL_LEN_OFFSET, (u32)(Len/Count) ) ;
```

To match the “Cyclic DMA Mode”, point the last “Descriptor” to the address of the first “Descriptor”.

```
if (i < BdCount - 2)
{
    BdPtrCurr += Bd_Align ;
    BdPtrNext += Bd_Align ;
}
else
{
    BdPtrCurr += Bd_Align ;
    BdPtrNext = BdDesptr ;
}
```

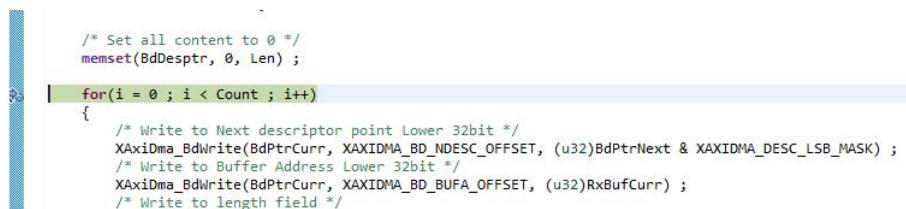
- 3) In order to facilitate understanding, in this experiment, the buffer area of the ADC is divided into four consecutive spaces. The following is the “Debug” to view the “memory” information. First “Run Debug” enters the “Debug” interface and sets a breakpoint in “CreateBdChain”.



```
/* Set all content to 0 */
memset(BdDesptr, 0, BdCount*BD_ALIGNMENT) ;

for(i = 0 ; i < BdCount ; i++)
{
    /* Write to Next descriptor point Lower 32bit */
    XAXIDMA_BdWrite(BdPtrCurr, XAXIDMA_BD_NDESC_OFFSET, (u32)BdPtrNext & XAXIDMA_DESC_LSB_MASK) ;
    /* Write to Buffer Address Lower 32bit */
    XAXIDMA_BdWrite(BdPtrCurr, XAXIDMA_BD_BUFA_OFFSET, (u32)RxBufCurr) ;
    /* Write to length field */
```

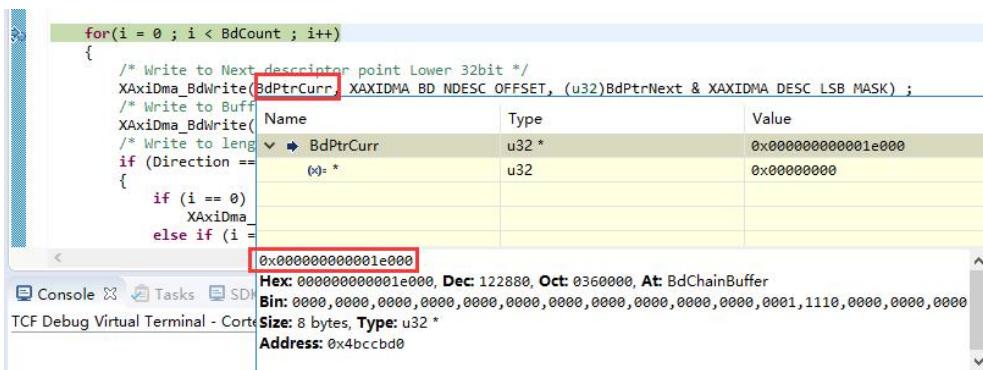
- 4) Click the “Resume” key to run to the breakpoint

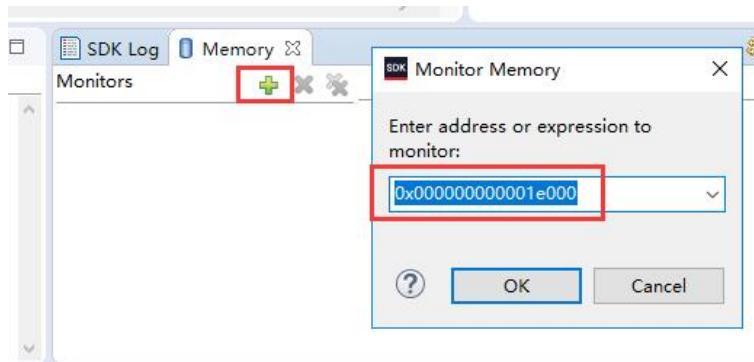
```
/* Set all content to 0 */
memset(BdDesptr, 0, Len) ;

for(i = 0 ; i < Count ; i++)
{
    /* Write to Next descriptor point Lower 32bit */
    XAXIDMA_BdWrite(BdPtrCurr, XAXIDMA_BD_NDESC_OFFSET, (u32)BdPtrNext & XAXIDMA_DESC_LSB_MASK) ;
    /* Write to Buffer Address Lower 32bit */
    XAXIDMA_BdWrite(BdPtrCurr, XAXIDMA_BD_BUFA_OFFSET, (u32)RxBufCurr) ;
    /* Write to length field */
```

5) Find the address of the linked list



6) Click the “Add” key in the “Memory” window to fill in the linked list address.



7) Cancel the current breakpoint, add a breakpoint to the end of the function, click the “Resume” key again to run to the breakpoint.

```

    }
    Xil_DCacheFlushRange((INTPTR) BdDesptr, Count*BD_ALIGNMENT) ;
    return XST_SUCCESS ;

```

8) You can see that the first Descriptor's NEXDESC points to the next “Descriptor” address, which is “0x00277700”, the first “BUFFER\_ADDRESS” is “0x002767C0”, and the “CONTROL” is “0x00000280”. In this experiment, the buffer space is set to be continuous, “0x002767C0+0x00000280=0x00276A40”, which is the next one. “BUFFER\_ADDRESS”, the user can also be set to a discontinuous space.

Address	0 - 3	4 - 7	8 - B	C - F
000000000001E000	0001E040	00000000	A 003DFB80	00000000
000000000001E010	00000000	00000000	A 000001E0	00000000
000000000001E020	00000000	00000000	00000000	00000000
000000000001E030	00000000	00000000	00000000	00000000
000000000001E040	0001E080	00000000	A 003DFD60	00000000
000000000001E050	00000000	00000000	A 000001E0	00000000
000000000001E060	00000000	00000000	00000000	00000000
000000000001E070	00000000	00000000	00000000	00000000

9) The last Descriptor's NEXDESC points to the address of the first Descriptor

Address	0 - 3	4 - 7	8 - B	C - F
000000000001E090	00000000	00000000	A 000001E0	00000000
000000000001E0A0	00000000	00000000	00000000	00000000
000000000001E0B0	00000000	00000000	00000000	00000000
000000000001E0C0	0001E000	00000000	A 003E0120	00000000
000000000001E0D0	00000000	00000000	A 000001E0	00000000
000000000001E0E0	00000000	00000000	00000000	00000000
000000000001E0F0	00000000	00000000	00000000	00000000
000000000001E100	00000000	00000000	00000000	00000000

10)The above is the setting of “Buffer Descriptor”. The following is the start configuration register to start “SG DMA”. Using “Bd\_Start” function, only need to write “CURDESC”, “DMACR”, “TAILDESC” register.

```

/* Write current descriptor pointer to DMA register */
XAxiDma_BdWrite(RingPtr->ChanBase, XAXIDMA_CDESC_OFFSET, (u32)BdDesptr & XAXIDMA_DESC_LSB_MASK) ;
/* Start DMA */
XAxiDma_BdWrite(RingPtr->ChanBase, XAXIDMA_CR_OFFSET,
XAxiDma_ReadReg(RingPtr->ChanBase, XAXIDMA_CR_OFFSET) | XAXIDMA_CR_RUNSTOP_MASK) ;

/* Write Tail descriptor pointer to DMA register, once it is written, SG will start fetching current descriptor pointer */
if (XAxiDma_BdRingHwIsStarted(RingPtr))
XAxiDma_BdWrite(RingPtr->ChanBase, XAXIDMA_TDESC_OFFSET, (u32)BdPtrLast & XAXIDMA_DESC_LSB_MASK) ;

```

11)After the end of a package transfer, the “DMA” will write information to the linked list “STATUS” via “M\_AXI\_SG”. You can see that the value of the first “Descriptor” is “0x88000280” and “RXSOF” is 1, which is the start of the packet.

Address	0 - 3	4 - 7	8 - B	C - F
000000000001E000	0001E040	00000000	003DFB80	00000000
000000000001E010	00000000	00000000	000001E0	880001E0
000000000001E020	00000000	00000000	00000000	00000000
000000000001E030	00000000	00000000	00000000	00000000
000000000001E040	0001E080	00000000	003DFD60	00000000
000000000001E050	00000000	00000000	000001E0	800001E0
000000000001E060	00000000	00000000	00000000	00000000
000000000001E070	00000000	00000000	00000000	00000000

12) After each processing of the data, you need to clear the status, which is the “STATUS” content, the program uses the “Bd\_StatusClr” function

```
void Bd_StatusClr(u32 *BdDesptr, u16 BdCount)
{
    int i ;
    u32 *BdPtrCurr = BdDesptr ;
    /* BD is 16 words alignment, every word is 4 bytes*/
    u32 Bd_Align = BD_ALIGNMENT/sizeof(u32) ;

    for(i = 0 ; i < BdCount ; i++)
    {
        XAxiDma_BdWrite(BdPtrCurr, XAXIDMA_BD_STS_OFFSET, 0) ;
        BdPtrCurr += Bd_Align ;
    }
    Xil_DCacheFlushRange((INTPTR) BdDesptr, BdCount*BD_ALIGNMENT) ;
}
```

### Part 26.3: Experimental Summary

The Scatter/Gather DMA mode needs to understand a lot of content. The first is the generation of the linked list. The difference between the linked list and the DMA register needs to be distinguished. Based on this experiment, users can write data to different address spaces and use the SG DMA mode flexibly.

In the routine, we also provide you with the use of TXPATH's SG DMA. Based on the AN108 DAC experiment, it will be easier to understand after finishing this experiment, so I won't go into details here.

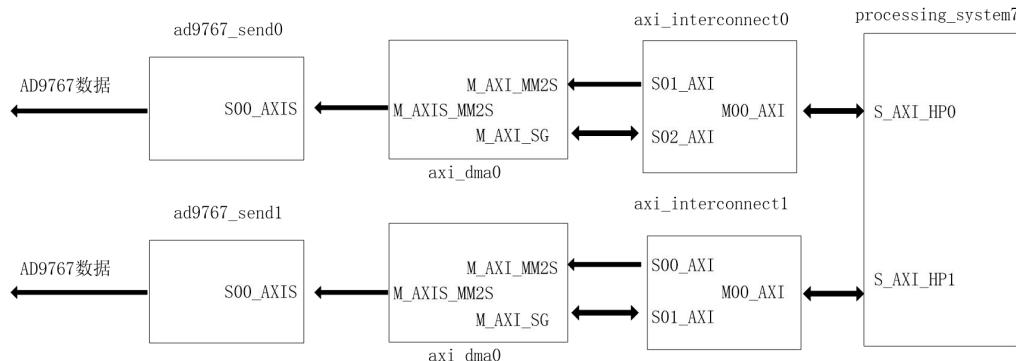
The SG project corresponding to AD7606 is also prepared for your reference.

# Part 27: Use the Scatter/Gather DMA Based on DAC Module (AN9767)

The experimental Vivado project directory is "ad9767\_sg\_dma /vivado".

The experiment vitis project directory is "ad9767\_sg\_dma /vitis".

This chapter uses the "AN9767" module as an example to demonstrate the use of the "SG DMA" read path, namely the "MM2S path". The following figure is the principle block diagram. Since the "AD9767" is dual channel and the bandwidth is relatively large, consider using two HP ports. This experiment is to simultaneously output the waveforms of the dual channel DAC and observe it through the oscilloscope.

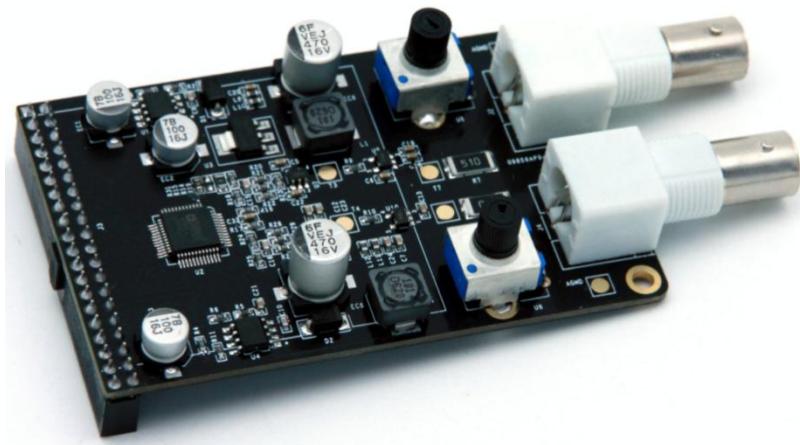


## Part 27.1: Dual 14-Bit DA Module Description

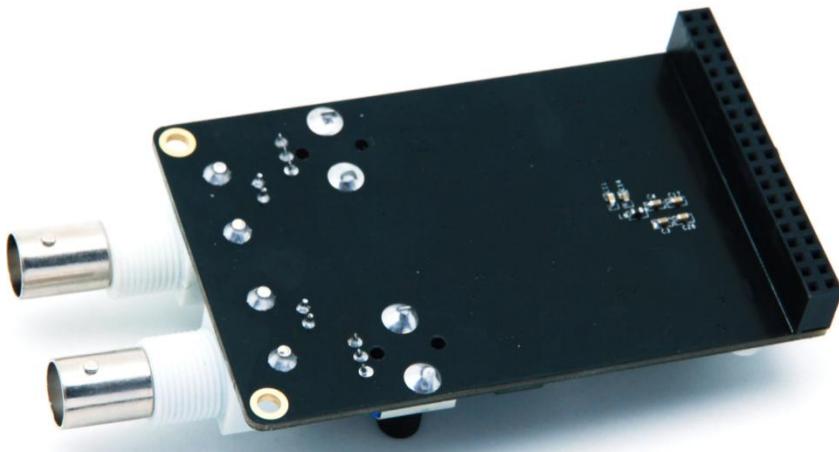
### 1) AN9767 Module Introduction

ALINX 14-bit dual port DA output module AN9767, use AD9767 chip of Analogy Devices, which supporting independent dual ports,

14-bit, 125MPS DA. The module reserves a 40-pin female header that to connect FPGA development kit, 2 BNC connectors to output analogy signals.



AN9767 module product photo (front side)



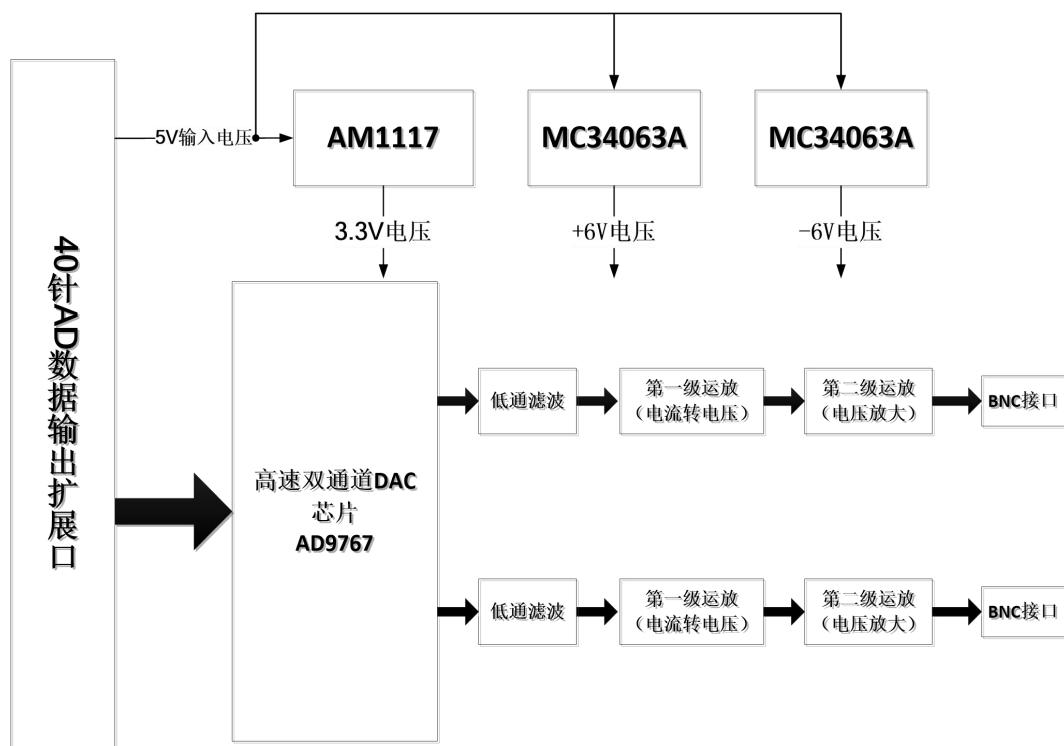
AN9767 module product photo (back side)

**AN9767 dual port DA module detail parameter listed as below:**

- DA chip: AD9767
- Channel: 2-channel

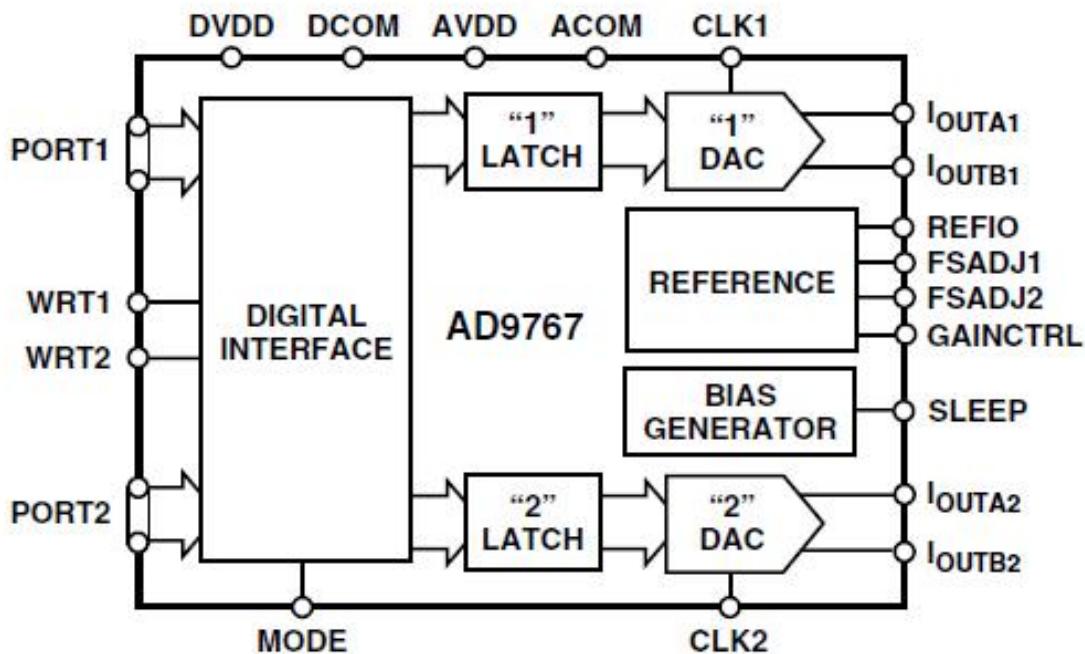
- DA bits: 14-Bits
- DA update rate: 125 MSPS;
- Output Voltage range: -5V~+5V;
- PCB layers of Module: 4-Layer, independent power layer and GND layer
- Module Interface: 40-pin 0.1 spacing female header, download direction
- Ambient Temperature (with power applied: -40°~85°, all the chips on module to meet the industrial requirements.
- Output interface: 2-Port BNC analog output interface (use the BNC line connect to the oscilloscope directly)

## 2) AN9767 Module Block Diagram



- 3) The AD9767 is a dual port, high speed, two channel, 14-bit CMOS DAC. It integrates two high quality 14-bit TxDAC+ cores, a voltage reference and digital interface circuitry into a 48-lead LQFP package. The AD9767 offers exceptional ac and dc performance

while supporting update rates up to 125 MSPS. The functional block diagram of AD9767 as below:



#### 4) Current and voltage conversion and amplification

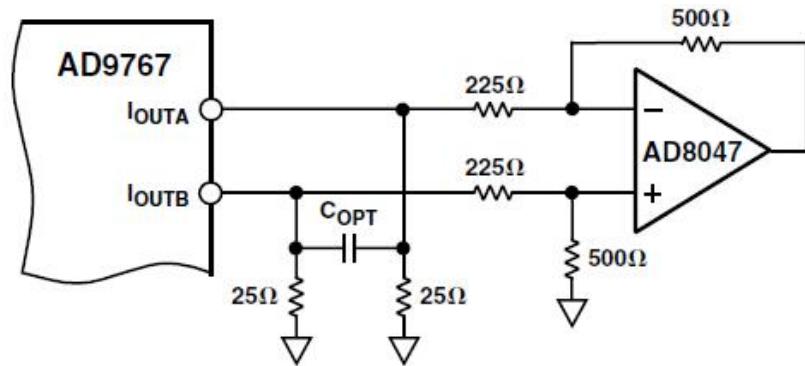
The two DA outputs of the AD9767 are the complementary current outputs “IoutA” and “IoutB”. When the digital input of the AD9767 is full-scale (14-bit data input to the DAC is high), “IoutA” outputs a full-scale current output of 20mA. The current output from “IoutB” is 0 mA. The relationship between the specific current and the DAC data is as follows:

$$I_{OUTA} = (DAC\ CODE / 16384) \times I_{OUTFS}$$

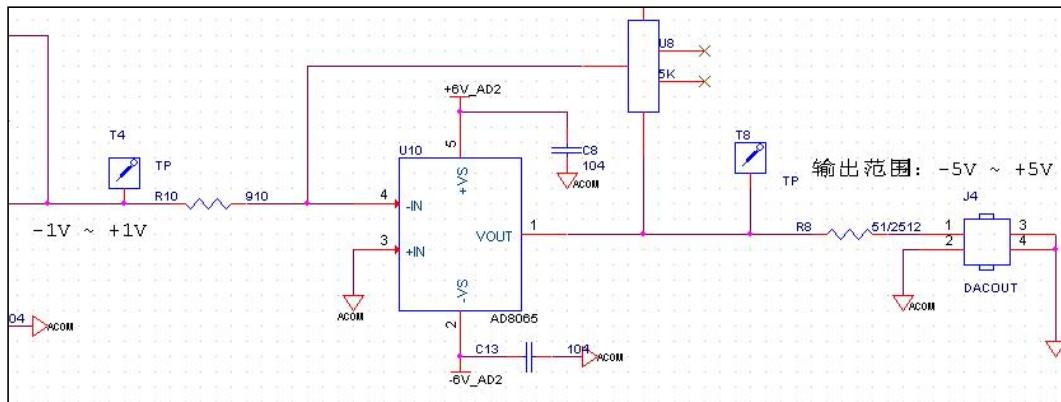
$$I_{OUTB} = (16383 - DAC\ CODE) / 16384 \times I_{OUTFS}$$

Where  $I_{OUTFS}=32 \times I_{ref}$ . In the design of AN9767,  $I_{ref}$  is set by the resistor of “R16”, if the is  $R16=19.2K$ , then the value of  $I_{ref}$  is 0.625mA,  $I_{OUTFS}$  is 20mA.

The output current of AD9767 is converted to voltage -1V~+1V through the first stage operational amplifier AD6045. The specific conversion circuit is shown below:



The voltage of -1V~+1V after the conversion of the first stage operational amplifier is converted to a higher amplitude voltage signal by the second stage operational amplifier. The amplitude of the second operational amplifier can be change by adjusting the adjustable resistor on the board. With the second stage operation amplifier, the output of range of the analog signal as high as -5V~+5V



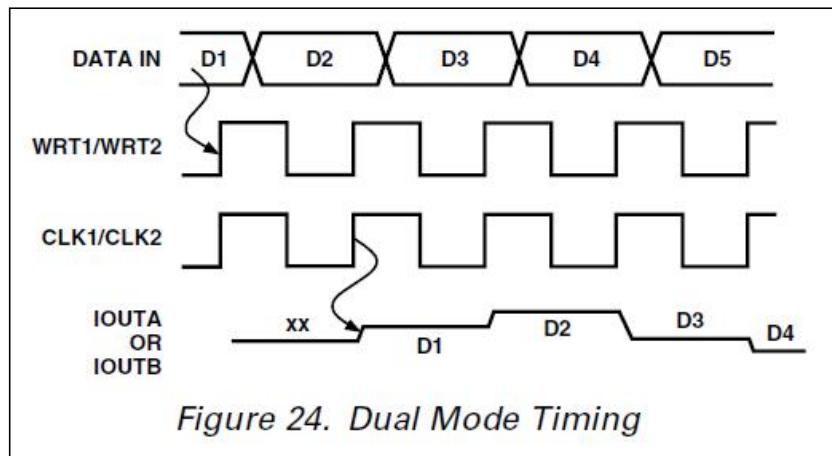
The following table shows the voltage input table after the digital input signal and the output of each level of the op amp:

DAC data input value	AD9767 current output	The output of first stage OP	The output of second stage OP
3fff (14 bits are high)	+20mA	-1V	+5V
0 (14 bits are low)	-20mA	+1V	-5V
2000 (Median)	0mA	0V	0V

## 5) Interface timing

The digital inputs of AD9767 chip is used in “Dual” mode or

“Interleaved” mode by setting PIN (MODE)。In the design of AN9767 module, the AD9767 chip operated in “Dual” mode. The DA digital inputs of dual ports are separated and independent. Time specifications for dual port mode are give in figure below



The DA data of the AD9767 chip is input to the chip for DA conversion by the rising edge of the clock “CLK” and the write signal “WRT”.

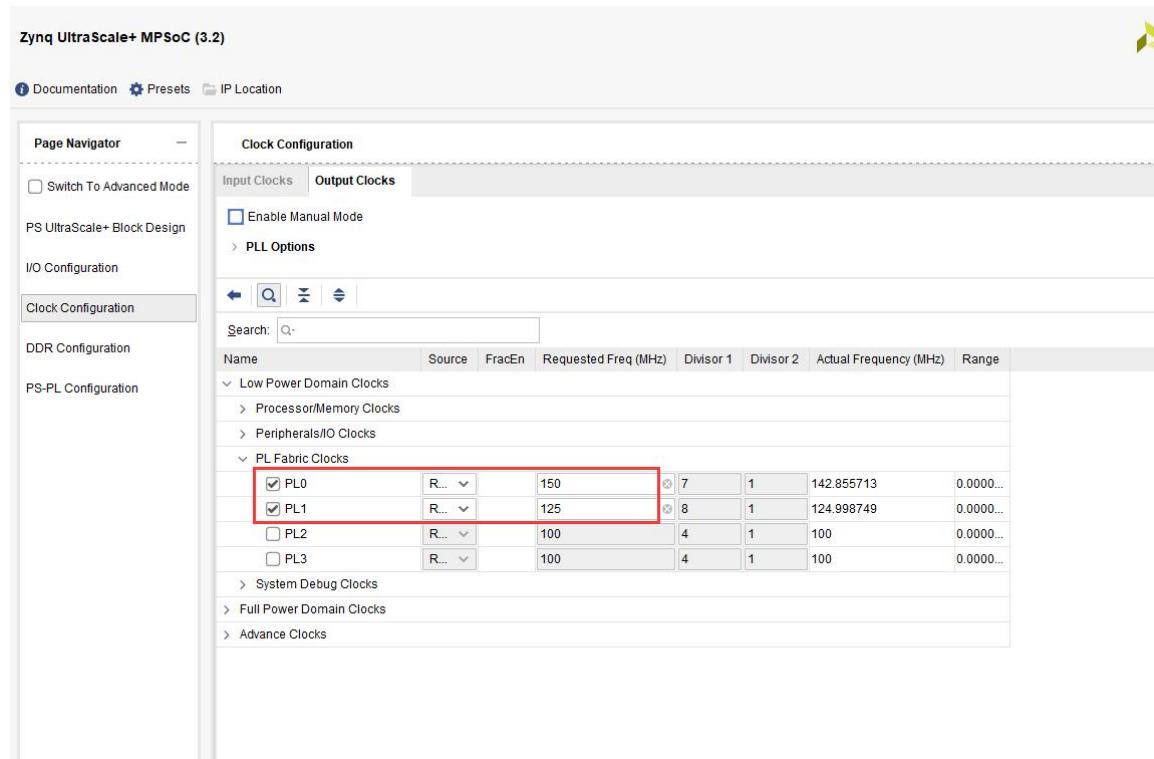
## FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

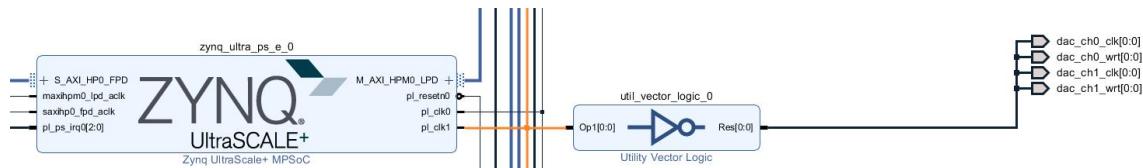
### Part 27.2: Hardware Environment

#### Part 27.2.1: Build Hardware

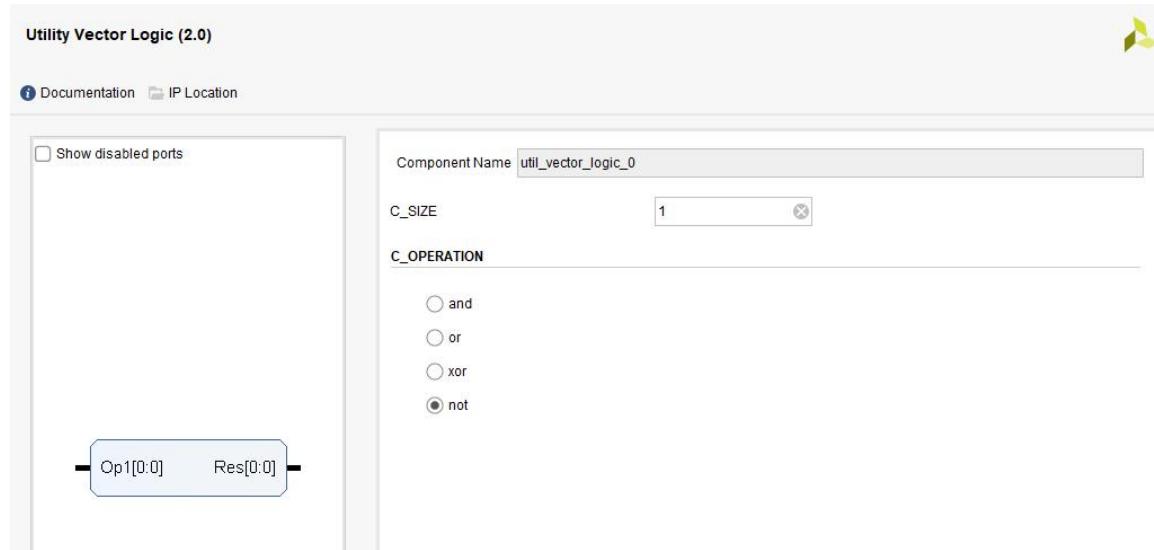
- 1) Based on the "Use of DMA--DAC Waveform Generator (AN108)" project, the changes were made. Set "FCLK\_CLK0" to "150MHz" for "AXI" bus use and set "FCLK\_CLK1" to "125MHz" to connect to the "AD9767" clock.



Lead out two clocks and two wrt channels, add a reverse module in the middle, in order to provide a better clock for the module

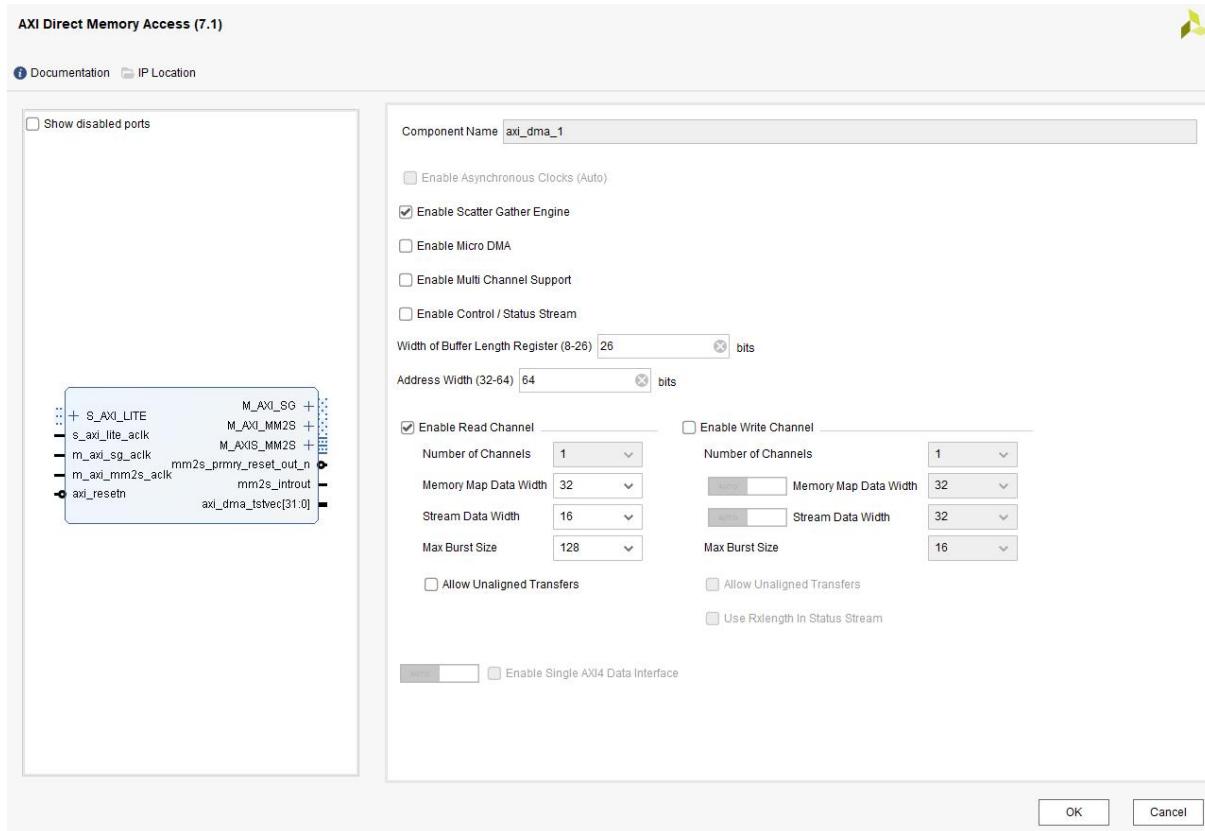


Utility Vector Logic module is set to not, C\_SIZE is 1

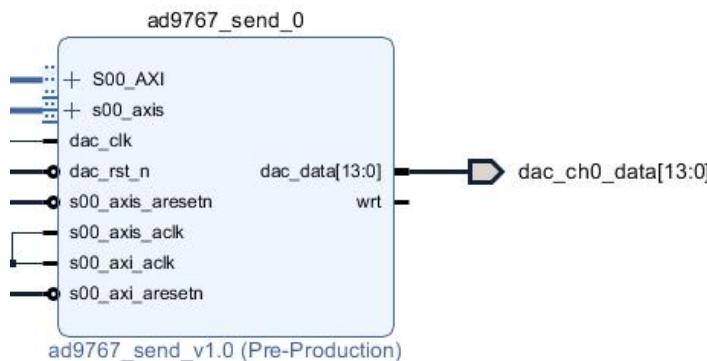


- 2) Set the “DMA” as follows, enable the “SG” function, set the “buffer” width to the maximum, open the read channel, set the “Stream

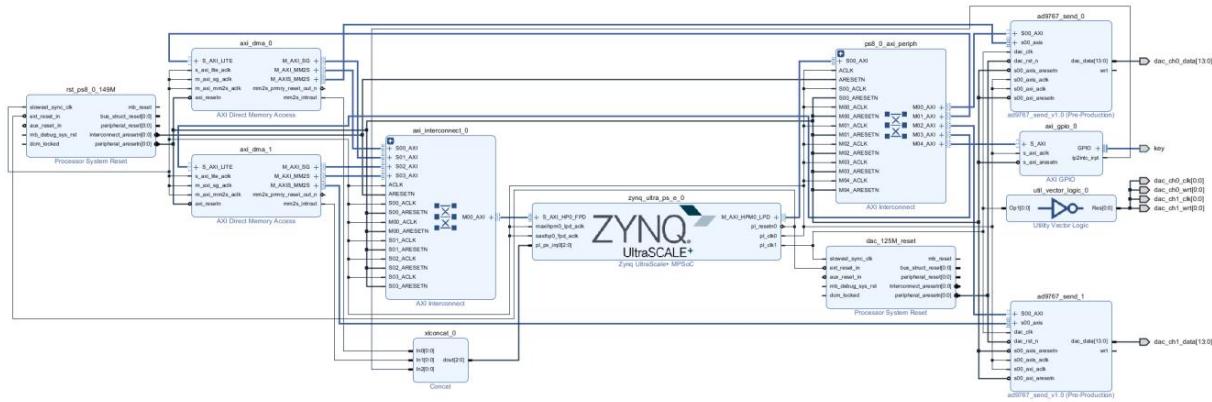
Data Width” to 16, connect the “ad9767” transmit module, and connect one “DMA” to each module.



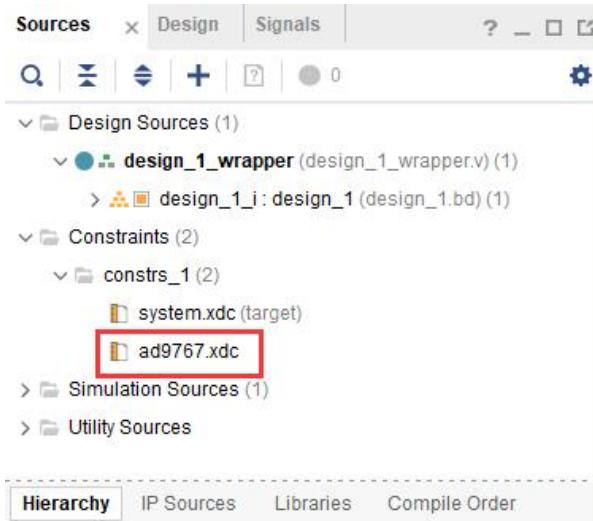
- 3) Add two “ad9767\_send” modules. This module functions to continuously read the data in the internal “FIFO” after the “ARM” sends the start command. The “FIFO” data is written by the “DMA” through the “AXIS MM2S” interface. And the data interface and “wrt” interface are brought out.



- 4) Connect each module, the final connection results are as follows:



## 5) Write ad9767.xdc file to bind pins to generate Bitstream



### Part 27.2.2: DAC custom IP function introduction

Since the waveform data needs to be transferred to the “DAC” through “DMA”, the interface with the “DMA” is “AXIS” stream interface, so the “DAC” data needs to be converted into “AXIS” stream data, and the clock of the “ADC” is different from the “AXIS” clock frequency. Therefore, it is necessary to add a “FIFO” for data processing across clock domains. At the same time, you need to implement the “AXIS Master” function. The workflow is:

- 1) The ARM configures the start register and the acquisition length register of the DAC
- 2) DMA uses AXIS interface to write data to FIFO

- 3) After the DAC state machine queries the FIFO for certain data, it begins to read the data. Since the clock frequency of AXIS is fast, the data read by the DAC can be guaranteed to be continuous.

## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 27.3: Vitis Program Development

For the use of SG DMA, refer to the chapter “Use the Scatter/Gather DMA based on ADC module (AN108)”

- 1) Here, set the maximum amplitude of two channels to “ $2^{14}$ ”, which is “16384”. Note that “AMP\_VAL” cannot be greater than “MAX\_AMP\_VAL”. You can adjust “AMP\_VAL” to change the amplitude. “MAX\_PKT\_LEN” is the amount of data collected. Note that it cannot be less than 1024. Because there is a period of time after the DMA interrupt, there is no data written to the “FIFO”, which will cause the “FIFO” to be read. The larger the collected value, the longer the effective data time. The “DMA” clock is “150MHz” and the “DAC” clock is “125MHz”, so the “FIFO” is guaranteed to be empty.

```
#define CH0_MAX_PKT_LEN      4096
#define CH0_MAX_AMP_VAL       16384
#define CH0_AMP_VAL           16384
#define CH1_MAX_PKT_LEN       4096
#define CH1_MAX_AMP_VAL       16384
#define CH1_AMP_VAL           16384
```

- 2) In the “main” function, the “PS” and “PL” keys are set, and the interrupt is enabled, the “PS” key controls the “channel0” waveform switching, and the “PL” key controls the “channel1” waveform switching.

```
PsKeySetup(&INST, PS_KEY_INTR_ID, &PsGpio) ;  
PLKeySetup(&INST, PL_KEY_INTR_ID, &PlGpio) ;
```

- 3) In the “XAxiDma\_DAC” function, create a “BD” linked list and start “DMA” transfer, here select “TXPATH”. In the “While” loop, if a key is pressed, the new waveform data is loaded and flushed to the memory.

```
CreateBdChain(Ch0BdTxChainBuffer, BD_COUNT, ADC_BYTE*CH0_MAX_PKT_LEN, (u8 *)Ch0DmaTxBuffer, TXPATH) ;  
Bd_Start(Ch0BdTxChainBuffer, BD_COUNT, &Ch0AxiDma, TXPATH) ;  
  
CreateBdChain(Ch1BdTxChainBuffer, BD_COUNT, ADC_BYTE*CH1_MAX_PKT_LEN, (u8 *)Ch1DmaTxBuffer, TXPATH) ;  
Bd_Start(Ch1BdTxChainBuffer, BD_COUNT, &Ch1AxiDma, TXPATH) ;
```

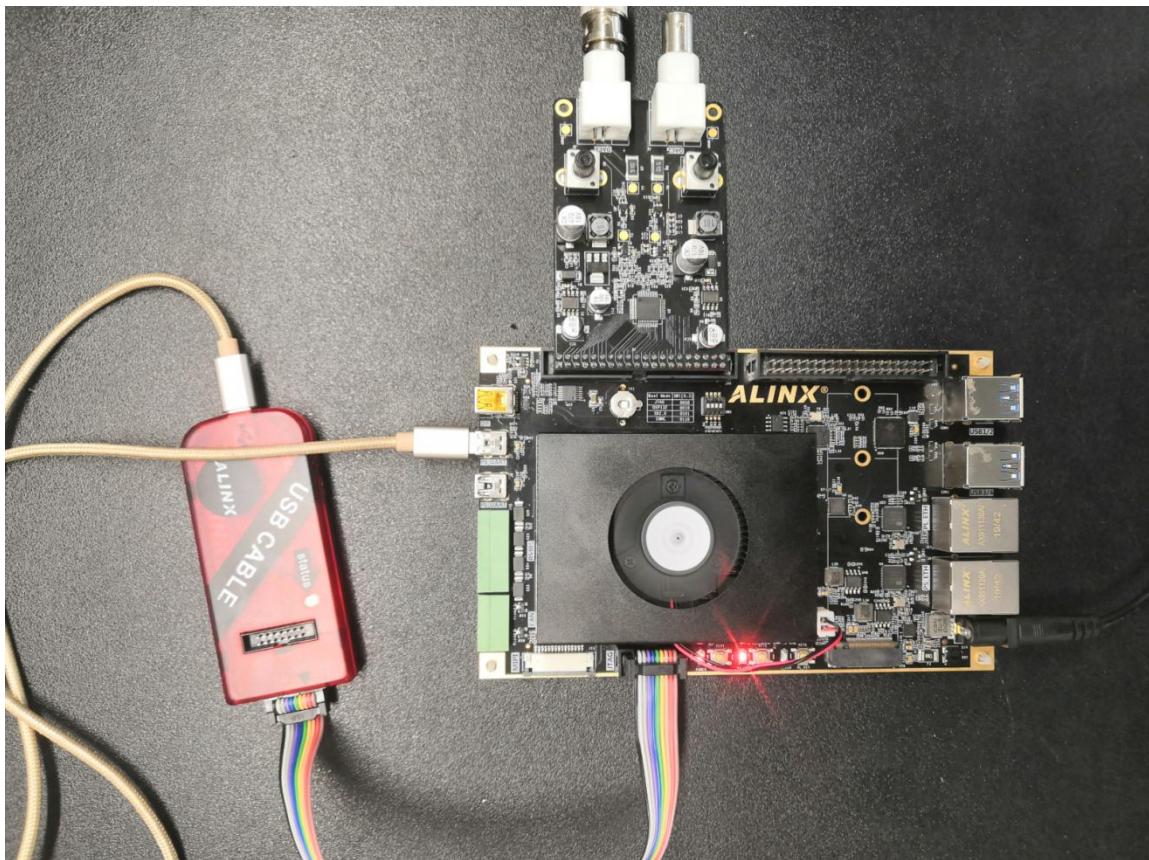
- 4) In the “DAC Interrupt Handler” function, the interrupt is cleared, the BD state is cleared, and the next DMA transfer is started.

```
if (XAxiDmaPtr->RegBase == CH0_DMA_BASE)  
{  
    Bd_StatusClr(Ch0BdTxChainBuffer, BD_COUNT) ;  
    Bd_Start(Ch0BdTxChainBuffer, BD_COUNT, &Ch0AxiDma, TXPATH) ;  
}  
/*  
 * Clear BD Status and Start Channel 1  
 */  
else if (XAxiDmaPtr->RegBase == CH1_DMA_BASE)  
{  
    Bd_StatusClr(Ch1BdTxChainBuffer, BD_COUNT) ;  
    Bd_Start(Ch1BdTxChainBuffer, BD_COUNT, &Ch1AxiDma, TXPATH) ;  
}
```

- 5) The other sections are not explained too much, please refer to the previous routine.

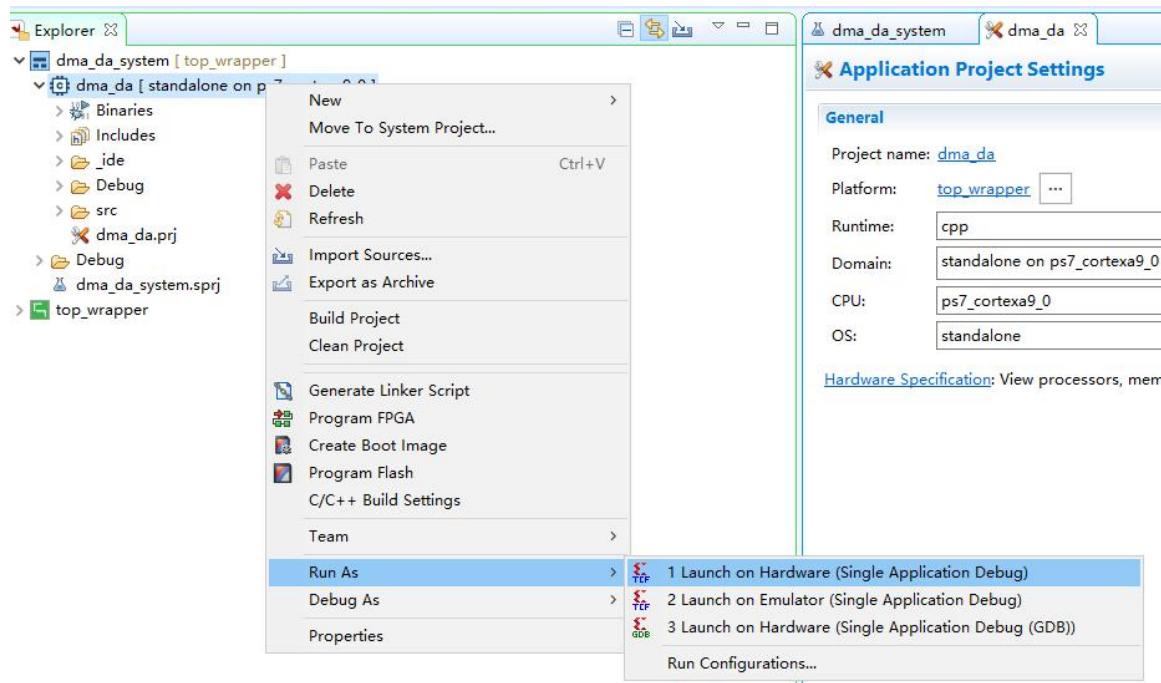
#### Part 27.4: Onboard verification

- 1) Connect the module to the FPGA development board and connect it to the oscilloscope with a dedicated shield.

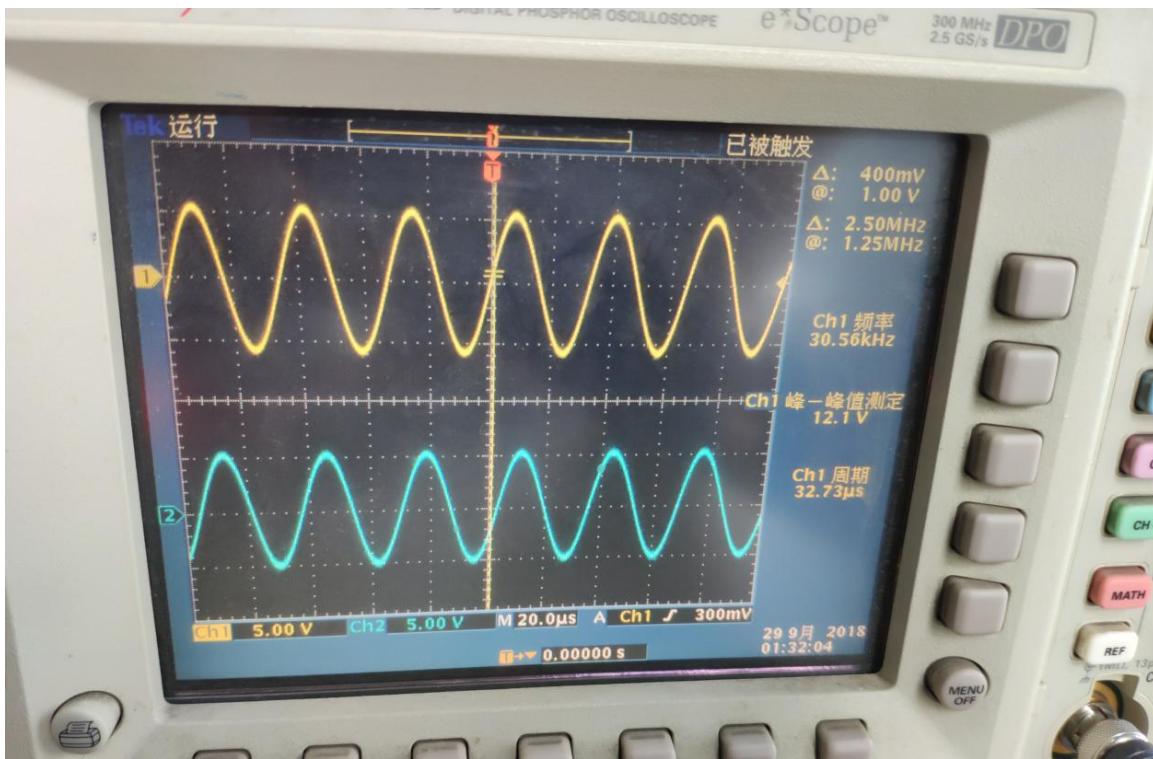


Hardware Connection (Expansion Port J46)

## 2) Download program:



3) After downloading, the oscilloscope displays as follows, and the waveform can be switched by pressing the key.



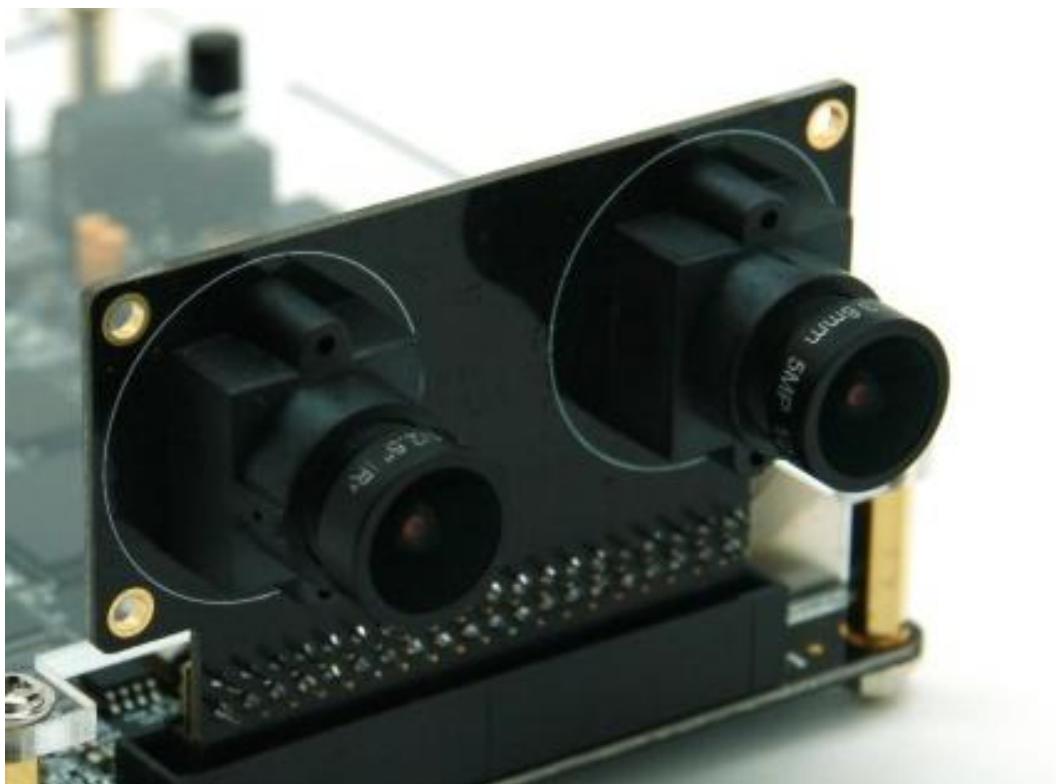
- 4) The amplitude can also be adjusted via the potentiometer on the module.

## Part 28: AN5642 Binocular Camera Collection and Display

The experimental Vivado project directory is "an5642\_double /vivado".

The experiment vitis project directory is "an5642\_double /vitis".

The application of binocular cameras is developing rapidly, and is widely used in 3D vision, VR, video intelligent analysis, industrial automation and other fields. However, binocular applications often have high requirements for system computing performance and interface scalability and flexibility. No asic or soc can solve a series of problems in binocular applications. Zynq integrates FPGA and arm, which brings great convenience to the preliminary verification of binocular applications.



## Part 28.1: OV5640 Chip Introduction

The image sensor is the core component of the camera. The image sensor in the ALINX binocular camera module “AN5642” is a “CMOS” type digital image sensor model “OV5640”. The sensor supports output of up to 5 megapixel images (2592x1944 resolution), supports output image data using VGA timing, and supports DUV (DC), MIPI interface output image data formats supporting YUV (422/420), YCbCr422, RGB565, RAW and JPEG formats. If the JPEG format image is directly output, the amount of data can be greatly reduced, which facilitates network transmission. It also compensates for acquired images and supports basic processing such as gamma curve, white balance, saturation, and chromaticity. Depending on the resolution configuration, the frame rate of the sensor output image data is adjustable from 15-60 frames, and the working power is between 150mW and 200mW.

The OV5640 uses the SCCB bus configuration, the SCCB is compatible with the I2C bus, the AN5642 module uses DVP to transmit video, PCLK is the pixel clock, and HREF is the line sync signal. When HREF is high, the video data is valid and VSYNC is the field sync signal. The data line is 10Bit, and our program only uses 8Bit. In the ALINX FPGA development board, we often configure it as RGB565 output. It is configured as YCbCr422 in ALINX professional video processing board. Because the data bus is 8Bit, it transmits one pixel of data in 2 clock cycles. It is then spliced into complete pixel data at the receiving end of the FPGA.

## Part 28.2: Use of VDMA

In the previous tutorial we have used VDMA for HDMI display. VDMA is a key IP in xilinx video processing. VDMA is a special DMA,

specially designed for video processing. As shown in the figure below, we see that VDMA has an AXI4 Memory Map interface for reading and writing video data to the memory, an AXI4-Lite interface for reading VDMA status and configuring VDMA parameters, and an AXI4-Stream interface. Used for video input and output.

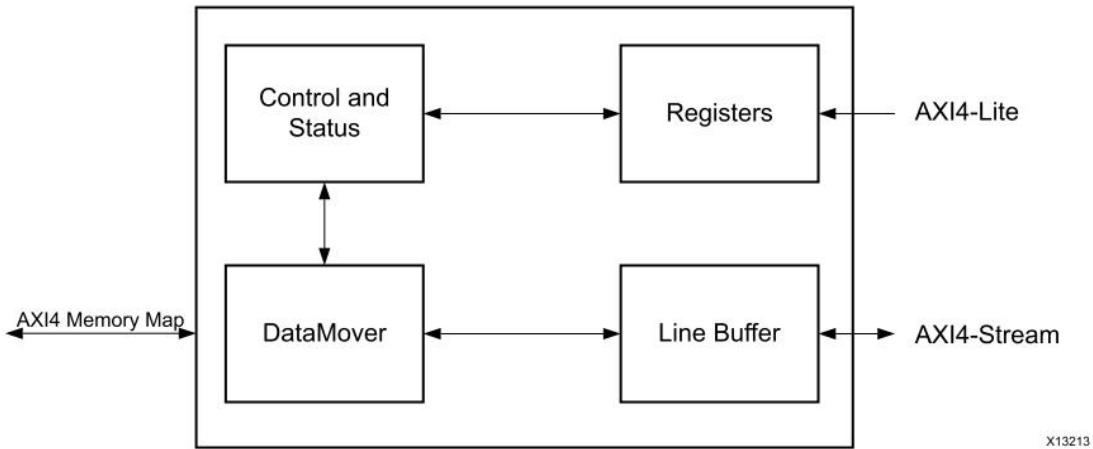


Figure 1-1: AXI VDMA Block Diagram

The configuration of VDMA is also very important. I extracted the registers of VMDA. We wrote a simple VDMA control program according to the document "AXI Video Direct Memory Access v6.3", as shown below:

Address Space Offset	Name	Description
00h	MM2S_VDMACR	MM2S VDMA Control Register
04h	MM2S_VDMASR	MM2S VDMA Status Register
08h to 10h	Reserved	N/A
14h	MM2S_REG_INDEX	MM2S Register Index
18h to 24h	Reserved	See <a href="#">Additional Design Information in Chapter 4</a> for more information.
28h	PARK_PTR_REG	MM2S and S2MM Park Pointer Register
2Ch	VDMA_VERSION	Video DMA Version Register
30h	S2MM_VDMACR	S2MM VDMA Control Register
34h	S2MM_VDMASR	S2MM VDMA Status Register
38h	Reserved	N/A
3Ch	S2MM_VDMA_IRQ_MASK	S2MM Error Interrupt Mask Register
40h	Reserved	N/A
44h	S2MM_REG_INDEX	S2MM Register Index

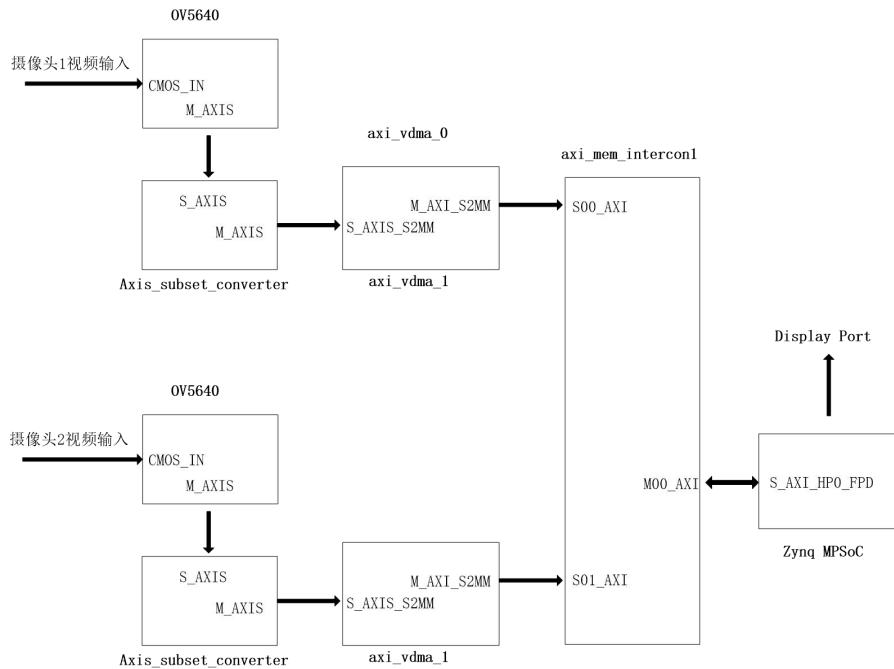
Address Space Offset	Name	Description
48h to 4Ch	Reserved	N/A
50h	MM2S_VSIZE	MM2S Vertical Size Register
54h	MM2S_HSIZE	MM2S Horizontal Size Register
58h	MM2S_FRMDLY_STRIDE	MM2S Frame Delay and Stride Register
5Ch to 98h	MM2S_START_ADDRESS (1 to 16) <sup>(1)(2)</sup>	MM2S Start Address (1 to 16)
9Ch	Reserved	N/A
A0h	S2MM_VSIZE	S2MM Vertical Size Register
A4h	S2MM_HSIZE	S2MM Horizontal Size Register
A8h	S2MM_FRMDLY_STRIDE	S2MM Frame Delay and Stride Register
ACh to E8h	S2MM_START_ADDRESS (1 to 16) <sup>(2)</sup>	S2MM Start Address (1 to 16)
ECh	ENABLE VERTICAL FLIP <sup>(3)</sup>	Vertical Flip Register
F0h to F4h	Reserved	N/A

## FPGA Engineer Job Content

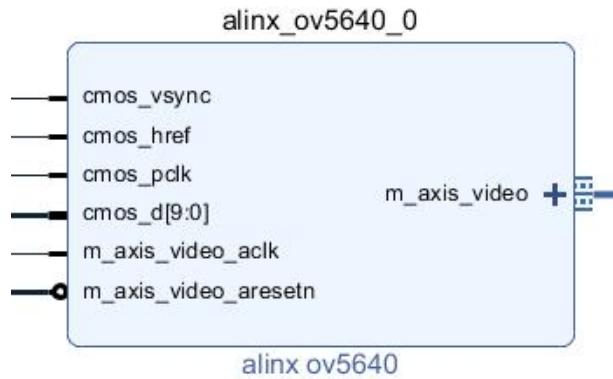
The following is the content that FPGA engineers are responsible for.

### Part 28.3: Hardware Environment

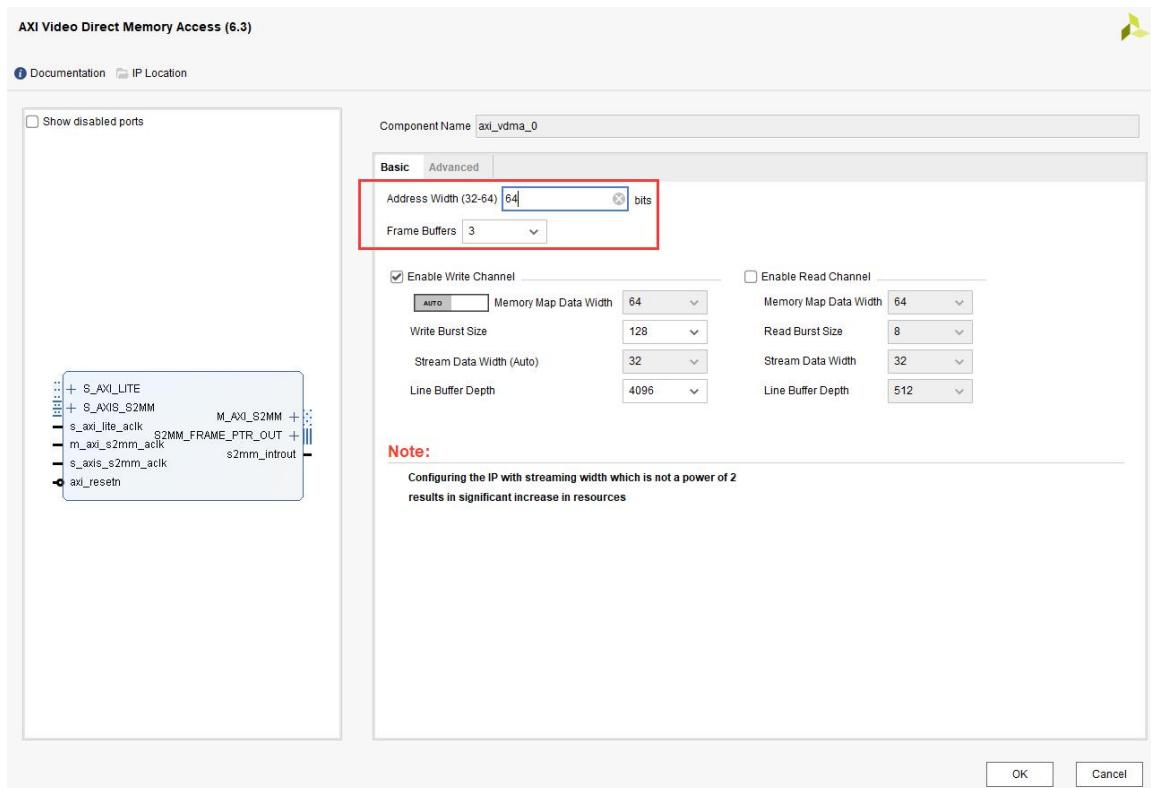
Based on the "ps\_hello" project, the reset, HP port, HPM, interrupt configuration will not be repeated. Two VMDAs are mainly added for the input of OV5640, the custom IP module of ov5640, and the subset is used for the data length conversion of the axis stream interface.

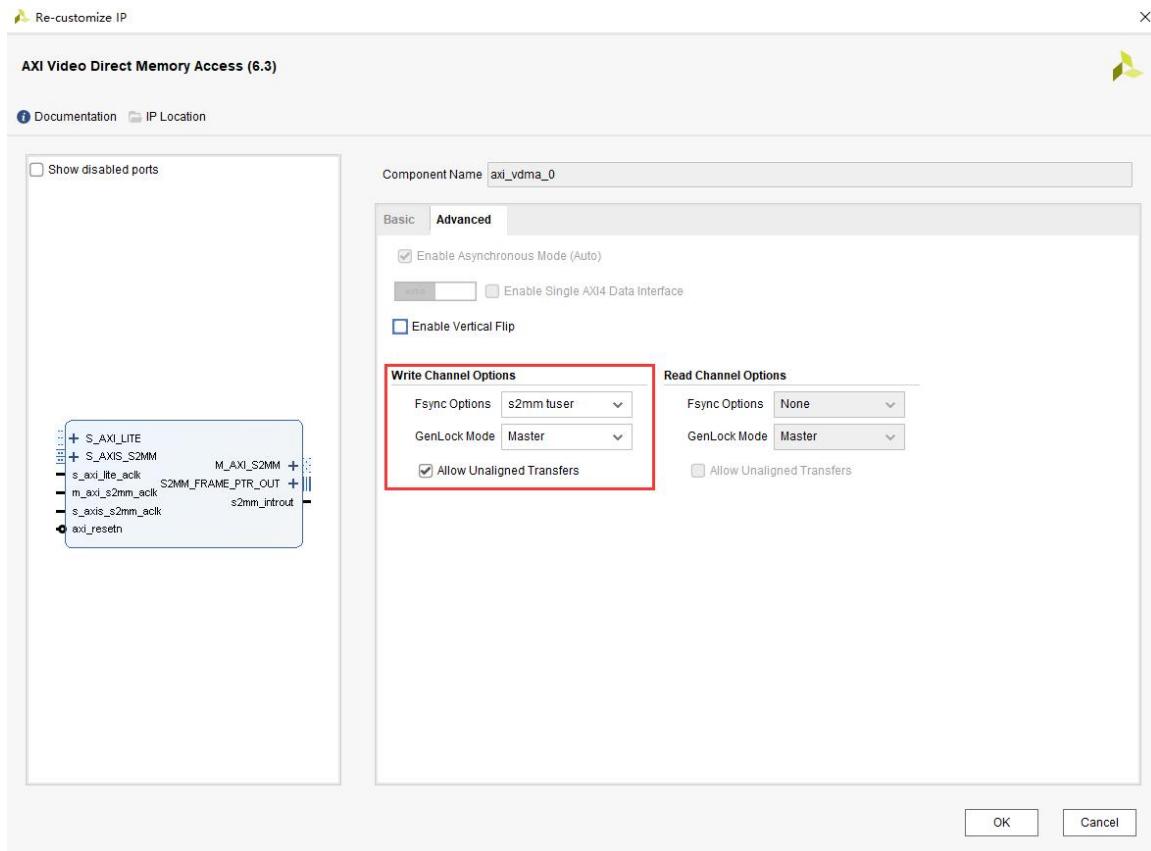


- 1) The alinx\_ov5640 module is a custom IP. The function is to convert the RGB565 input from the camera into AX4-Stream format. It contains a cmos\_8\_16bit module to convert the data input by the 8-bit camera into a 16-bit RGB565 format. At this time, the output RGB565 is one pixel of 2 clock cycles, one clock cycle is useless data, cmos\_8\_16bit uses de\_o to indicate whether the pixel data is valid, de\_o is not continuous in one line of video data, and hblank of cmos\_8\_16bit is a line synchronization signal. Use to indicate pixel data for a row. Another module, cmos\_in\_axi4s, is a module that converts the camera's RGB565 data into AXI4-Stream. The code of this module is modified by xilinx's "Video In to AXI4-Stream" IP. This IP input is a standard video format. The camera's RGB565 cannot be entered directly.

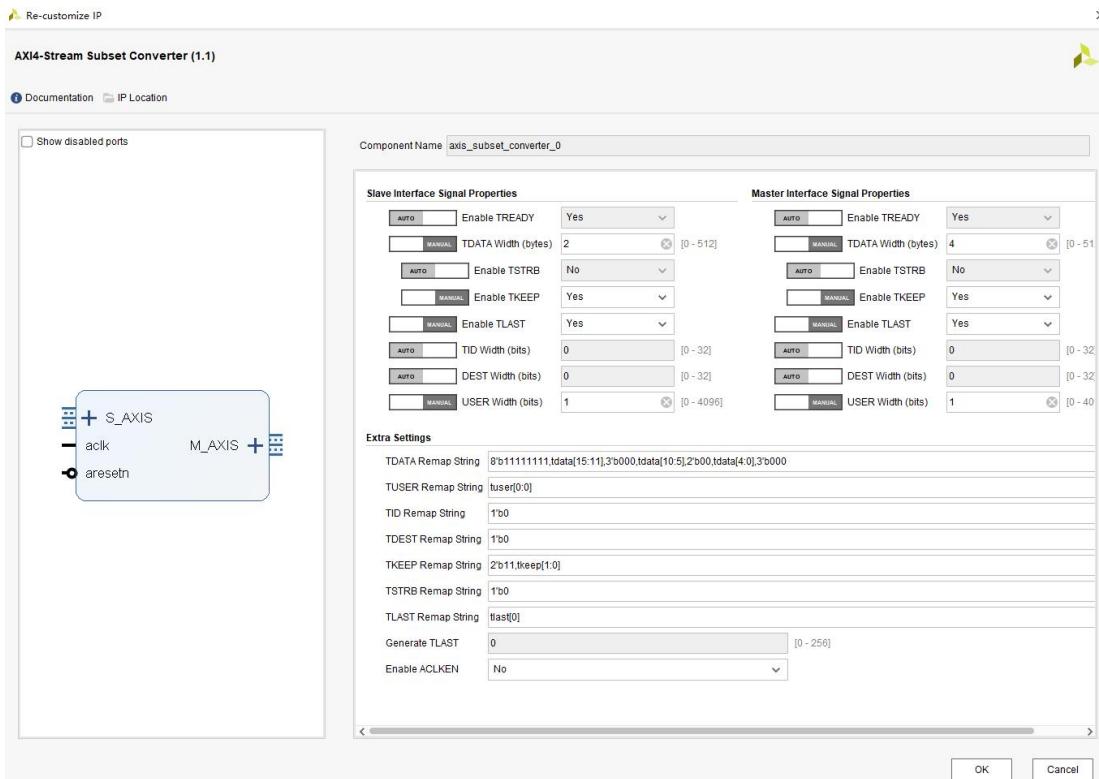


2) “AXI\_VMDA1” configuration, we only enable write channel, read channel is not enabled, in the “Advanced” tab our “Fsync Options” select “s2mm tuser” for data synchronization, while enabling misaligned transmission. It should be noted that if the misaligned transfer is not enabled, the buffer address of the “VDMA” must be 8-byte aligned.

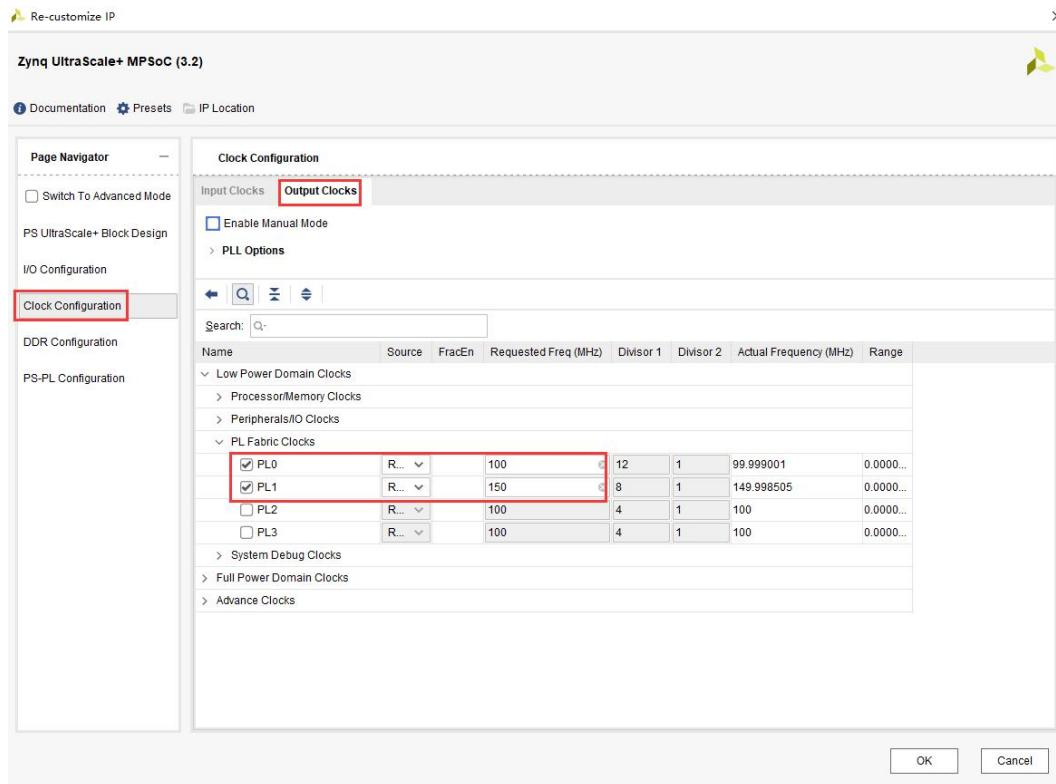




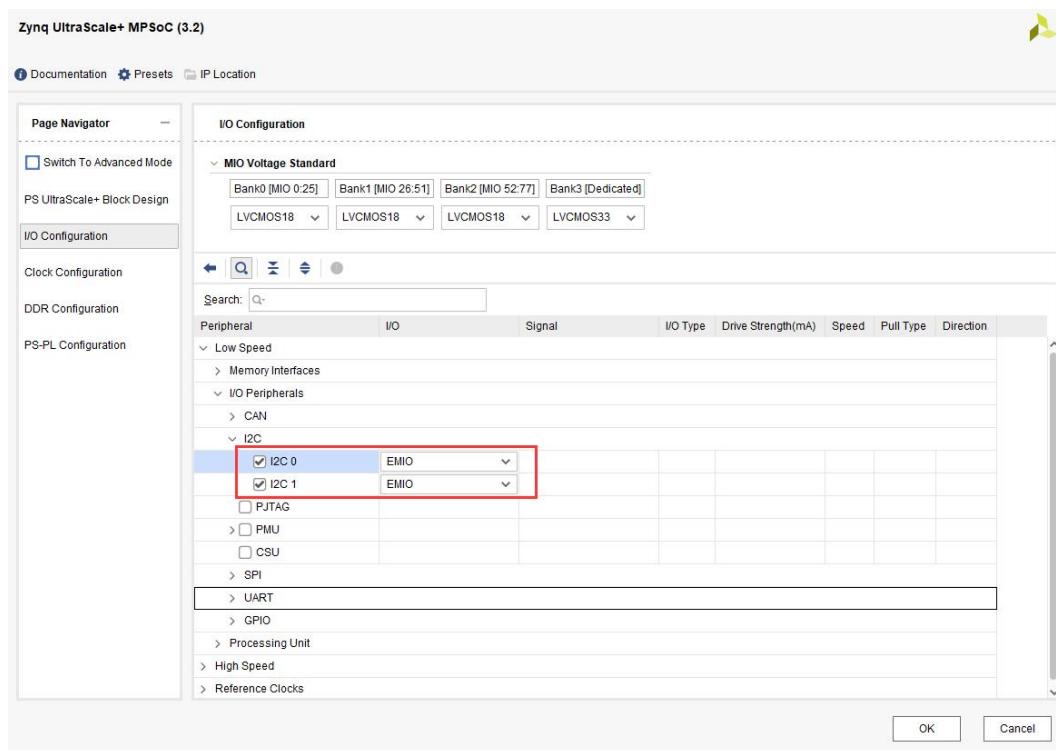
3) Since the default output of DP is set to RGBA (4 bytes), it is necessary to add the AXI4-Stream Subset Converter module to convert 2 bytes to 4 bytes



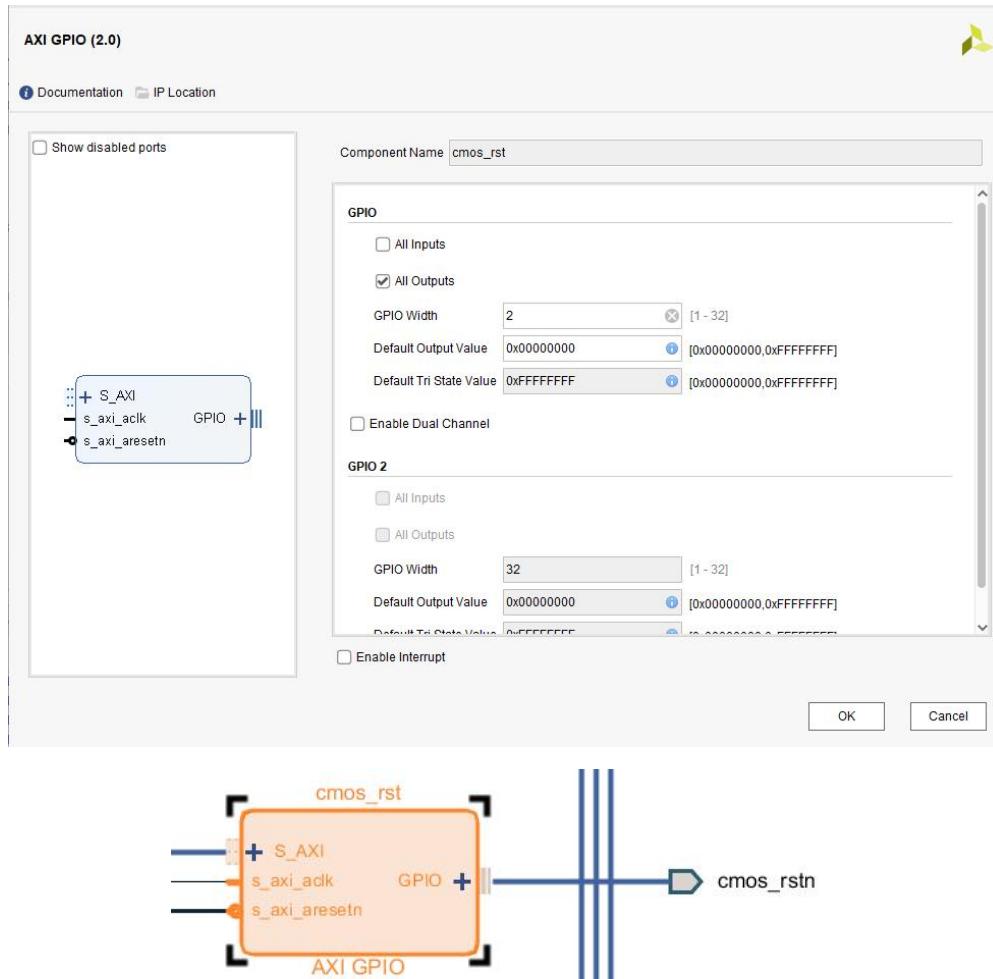
- 4) Zynq core clock configuration, PL0 is configured as 100MHz, used for register configuration of the module, and PL1 is configured as 150MHz for AXIS stream data processing



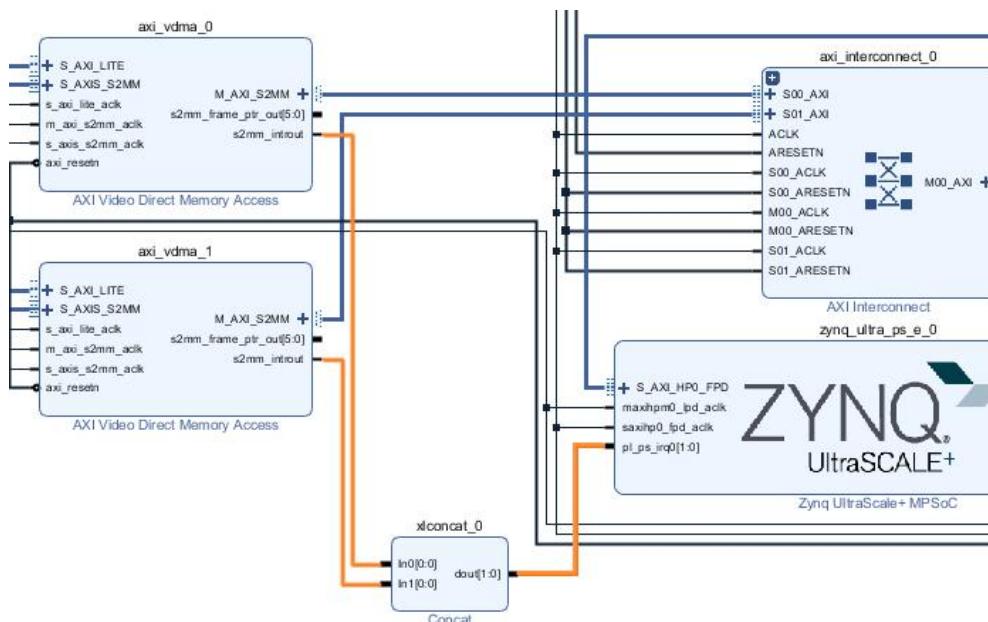
- 5) Add two I2C and set it as EMIO for configuring ov5640



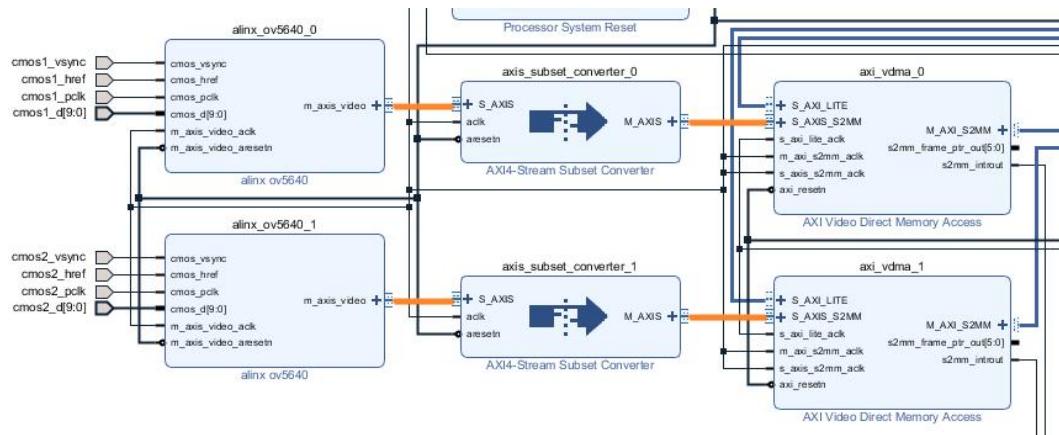
6) Add GPIO, set as output, width is 1, used for cmos reset



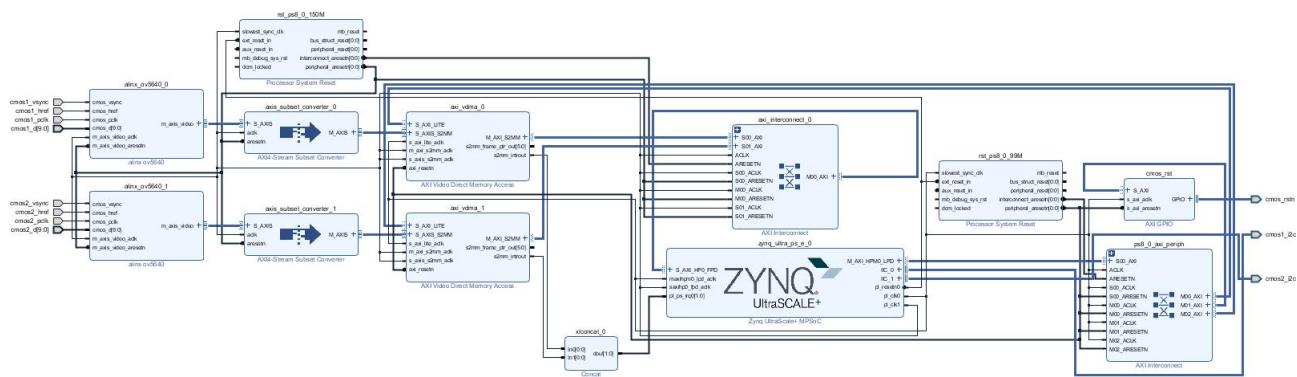
7) Add concat, connect two VDMA interrupts to ZYNQ core



Connect the corresponding interface



The other connections will not be repeated, the vivado project after the connection is as shown in the figure below:



Save the design, bind the pins, and generate bitstream

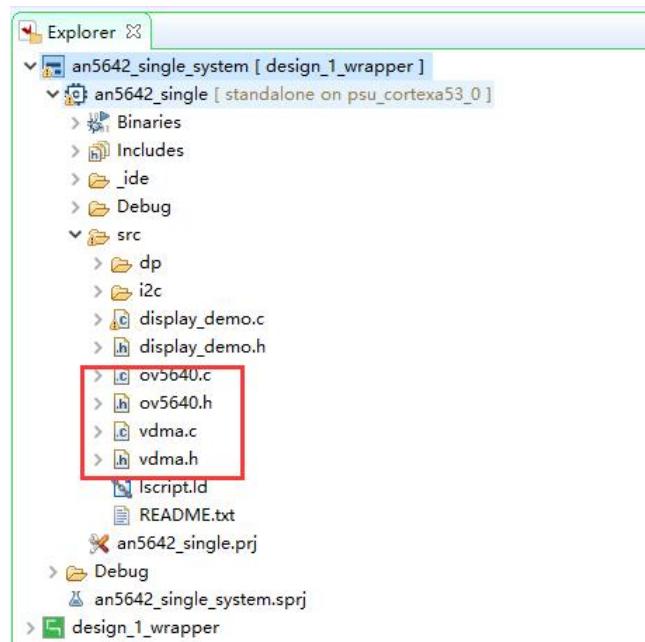
## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 28.4: Vitis Program Development

#### Part 28.4.1: Monocular Camera Display

- 1) ov5640.c and ov5640.h, used for i2c initialization camera, vdma.c and vdma.h, used for vdma initialization and read-write control



## 2) In dp.c, set the display resolution to 1280\*720

```

75 //##define BUFFERSIZE          1920 * 1080 * 4    /* HTotal * VTotal * BPP */
76 //##define LINESIZE           1920 * 4             /* HTotal * BPP */
77
78 #define BUFFERSIZE          1280 * 720 * 4    /* HTotal * VTotal * BPP */
79 #define LINESIZE           1280 * 4             /* HTotal * BPP */
80
81 #define STRIDE              LINESIZE           /* The stride value should
82                                         be aligned to 256*/
83
84
85 ****
86 void InitRunConfig(Run_Config *RunCfgPtr)
87 {
88     /* Initial configuration parameters. */
89     RunCfgPtr->DpPsuPtr = &DpPsu;
90     RunCfgPtr->IntrPtr = &Intr;
91     RunCfgPtr->AVBufPtr = &AVBuf;
92     RunCfgPtr->DpDmaPtr = &DpDma;
93     RunCfgPtr->VideoMode = XVIDC_VM_1280x720_60_P; //XVIDC_VM_1920x1080_60_P;
94     RunCfgPtr->Bpc      = XVIDC_BPC_8;
95     RunCfgPtr->ColorEncode = XDPPSU_CENC_RGB;
96     RunCfgPtr->UseMaxCfgCaps = 1;
97     RunCfgPtr->LaneCount = LANE_COUNT_2;
98     RunCfgPtr->LinkRate = LINK_RATE_540GBPS;
99     RunCfgPtr->EnSyncHClkMode = 0;
100    RunCfgPtr->UseMaxLaneCount = 1;
101    RunCfgPtr->UseMaxLinkRate = 1;
102 }

```

## 3) In the main function of display\_demo.c, first reset the sensor and initialize the sensor.

```

int main(void)
{
    int Status;

    Xil_DCacheDisable();
    Xil_ICacheDisable();

    /*
     * Clear frame buffer
     */
    memset(Frame, 0, DEMO_MAX_FRAME);

    i2c_init(&ps_i2c0, XPAR_XIICPS_0_DEVICE_ID, 40000);
    XGpio_Initialize(&cmos_rstn, XPAR_CMOS_RST_DEVICE_ID);
    XGpio_SetDataDirection(&cmos_rstn, 1, 0x0);
    XGpio_DiscreteWrite(&cmos_rstn, 1, 0x1);
    usleep(500000);
    XGpio_DiscreteWrite(&cmos_rstn, 1, 0x0);

    usleep(500000);
    XGpio_DiscreteWrite(&cmos_rstn, 1, 0x1);
    usleep(500000);

    /*
     * Initialize Sensor
     */
    sensor_init(&ps_i2c0);
}

```

#### 4) And then set the display

```

/*
 * DP dma demo
 */
xil_printf("DPDMA Generic Video Example Test \r\n");
Status = DpdmaVideoExample(&RunCfg, Frame);
if (Status != XST_SUCCESS) {
    xil_printf("DPDMA Video Example Test Failed\r\n");
    return XST_FAILURE;
}

```

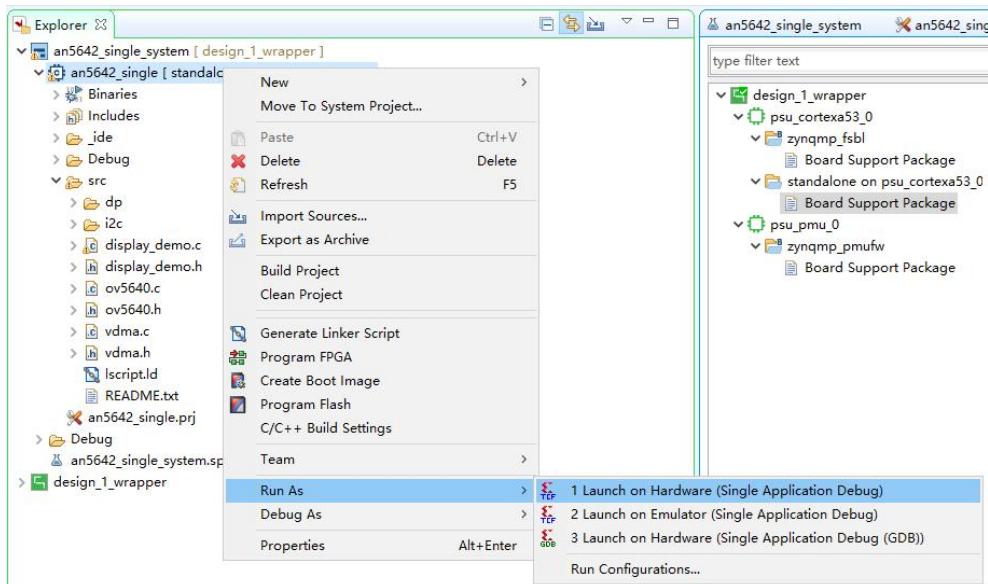
#### 5) Finally, read the VDMA settings of the Sensor

```

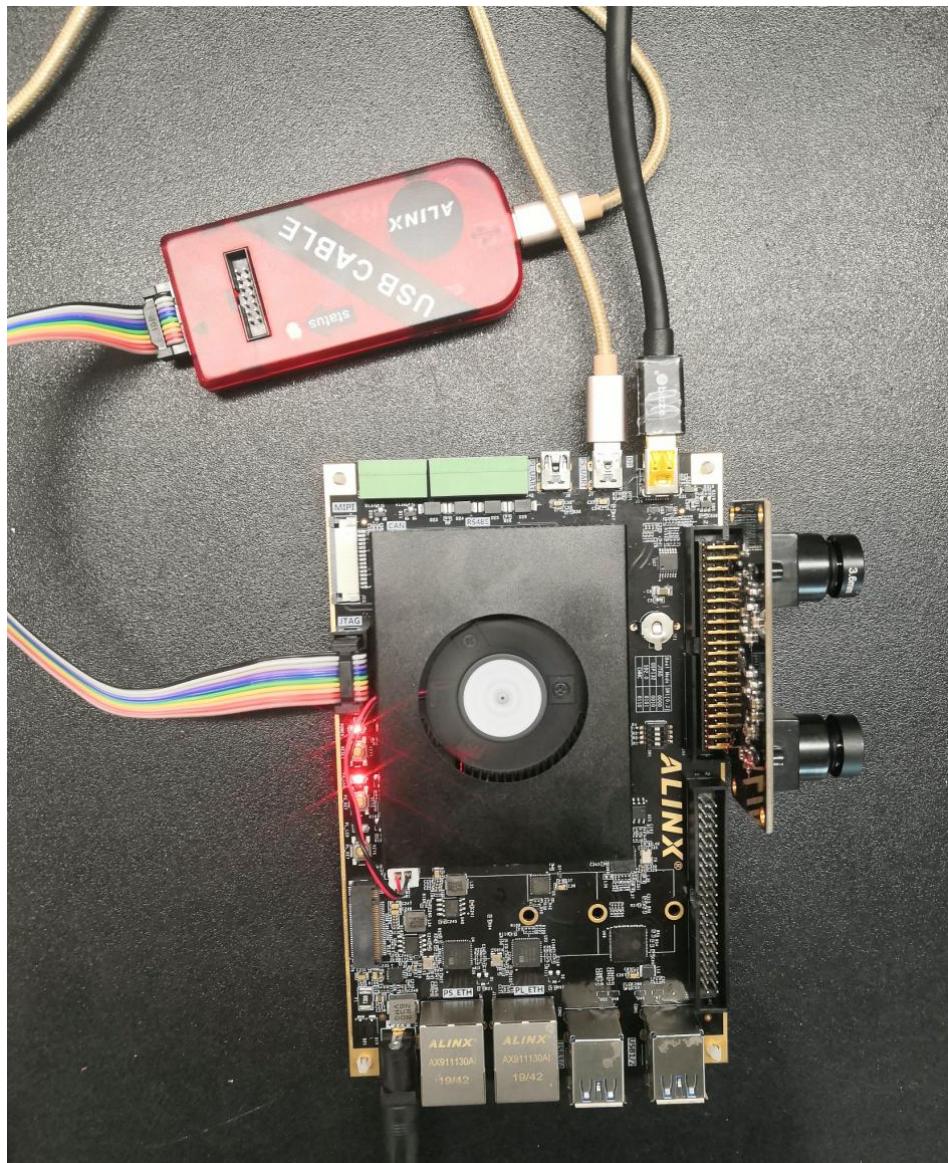
/* Start Sensor Vdma */
vdma_write_init(XPAR_AXIVDMA_0_DEVICE_ID, 1280 * 4, 720, 1280 * 4, (unsigned int)Frame);

```

#### 6) Download the programs



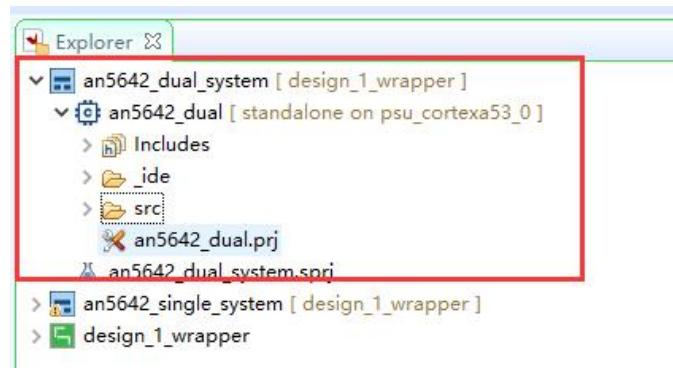
After the program runs, the camera image can be displayed on the DP monitor.



Hardware Connection and Display(Expansion Port J46)

#### Part 28.4.2: Binocular Camera Display

Create a new APP project



Modified on the basis of the monocular display program, the DP display is set to 1080P, and the two cameras are configured to 960\*540, and they are placed on the upper left and lower right of the screen.

```
/* Reset Camera Resolution */
ResetCamRes(&ps_i2c0, 960, 540) ;
ResetCamRes(&ps_i2c1, 960, 540) ;
/* Start Sensor Vdma */
vdma_write_init(XPAR_AXIVDMA_0_DEVICE_ID,960 * 4,540,1920 * 4,(unsigned int)Frame);
vdma_write_init(XPAR_AXIVDMA_1_DEVICE_ID,960 * 4,540,1920 * 4,(unsigned int)Frame+DEMO_MAX_FRAME/2+960*4);
```

Download the program, verify on the board

## Part 28.5: Experimental Summary

This chapter covers a wide range of topics. For developers who are new to zynq, especially those with only FPGA or only arm, it is difficult to understand. The main knowledge points are video basic knowledge, RGB656, video timing, AXI bus, I2C, VDMA, etc. It takes a long time to master. Through such routines, we can recognize the flexibility of the zynq soc system and solve many problems that are difficult to solve with ARM or FPGA.

## Part 29: SD card read and write operation

### - camera capture

The experimental Vivado project directory is "an5642\_double /vivado".

The experiment vitis project directory is "an5642\_sd /vitis".

This chapter combines the OV5640 camera module to capture images and save them to the SD card.

### Software Engineer Job Content

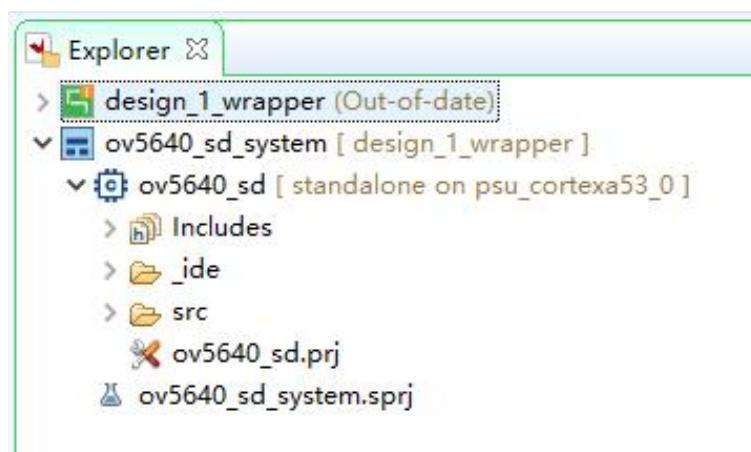
The following is the content that FPGA engineers are responsible for.

#### Part 29.1: Vitis Program Development

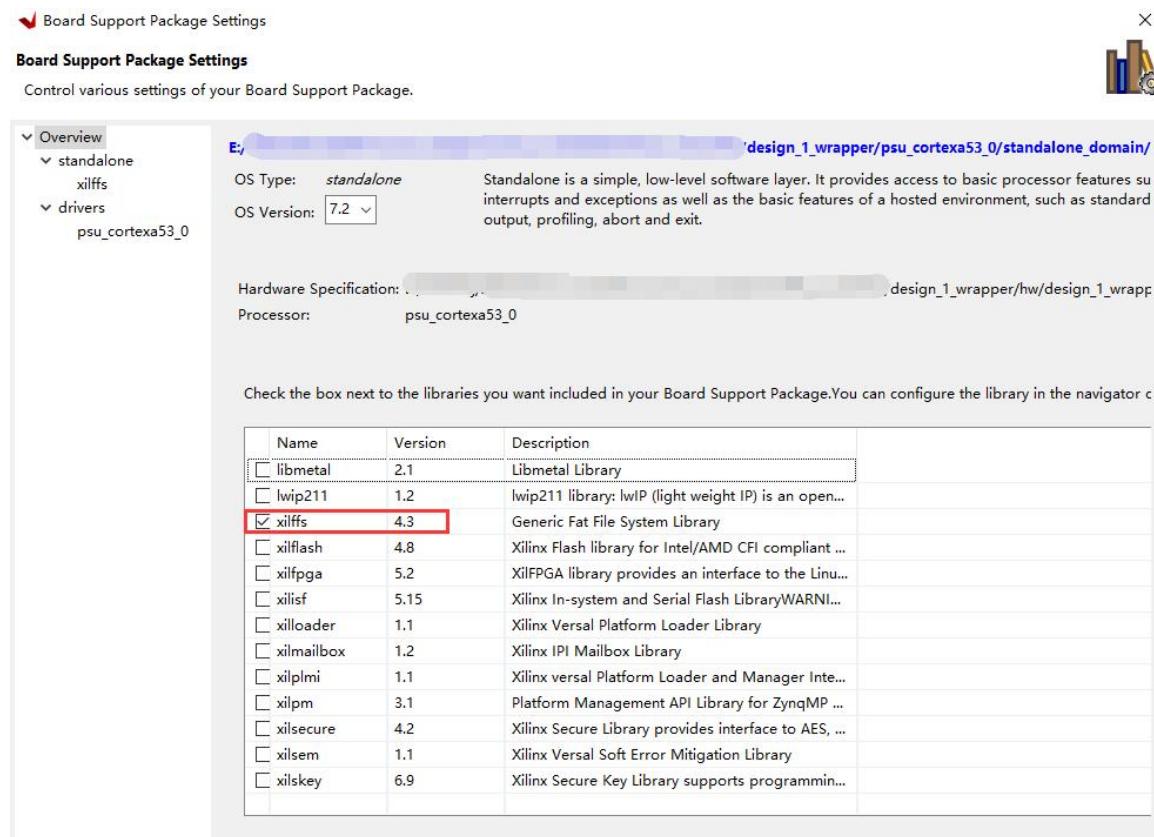
The experimental process is as follows:

Initialize VDMA → three-frame image switching → detect snap key interrupt, stop image switching → copy frame image data before current frame to photo buffer → continue three-frame image switching, and simultaneously write image data to SD card

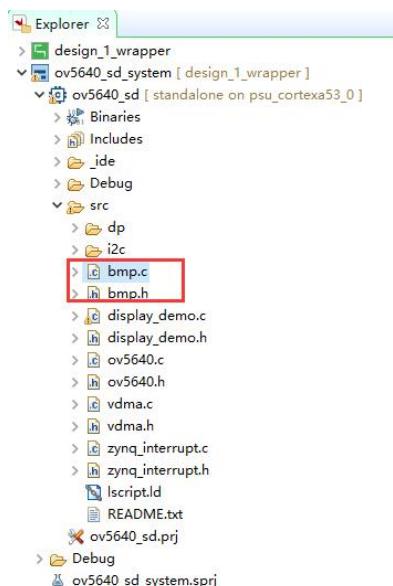
- 1) Create a new vitis project based on the hardware project of "AN5642 Binocular Camera Acquisition and Display"



- 2) In this experiment, you also need to open the “xilffs” library, as shown below, set in the “Board Support Package Settings”



- 3) “bmp.c” and “bmp.h” have been added to the project, which is the “BMP header information” and “read and write BMP functions” in the BMP image display chapter.



- 4) In the “main” function of the “display\_demo.c” file, open the

camera and display the interrupt of VDMA at the same time, and set the interrupt service function of the camera to “WriteCallBack”, and the interrupt service function displayed is “ReadCallBack”. In the “WriteCallBack” function, it is judged whether the key is pressed, and then the value of the “key\_flag” is changed, and the read/write frame is not switched at this time.

```
static void WriteCallBack(void *CallbackRef, u32 Mask)
{
    if (Mask & XAXIVDMA_IXR_FRMCNT_MASK)
    {
        if(key_flag == 1)
        {
            key_flag = 2 ;
            return;
        }
        else if(key_flag == 2)
        {
            return ;
        }

        if(wr_index==2){
            wr_index=0;
        }
        else{
            wr_index++;
        }
        /* Set park pointer */
        XAxivdma_StartParking((XAxivdma*)CallbackRef, wr_index, XAXIVDMA_WRITE);
    }
}
```

- 5) In the “while” loop of the main function, determine the value of “key\_flag”, add 1 to the photo name, copy the valid frame before the current frame to the photo buffer, then clear the “key\_flag”, write the “BMP” image data to the SD card, and print out the required time value.

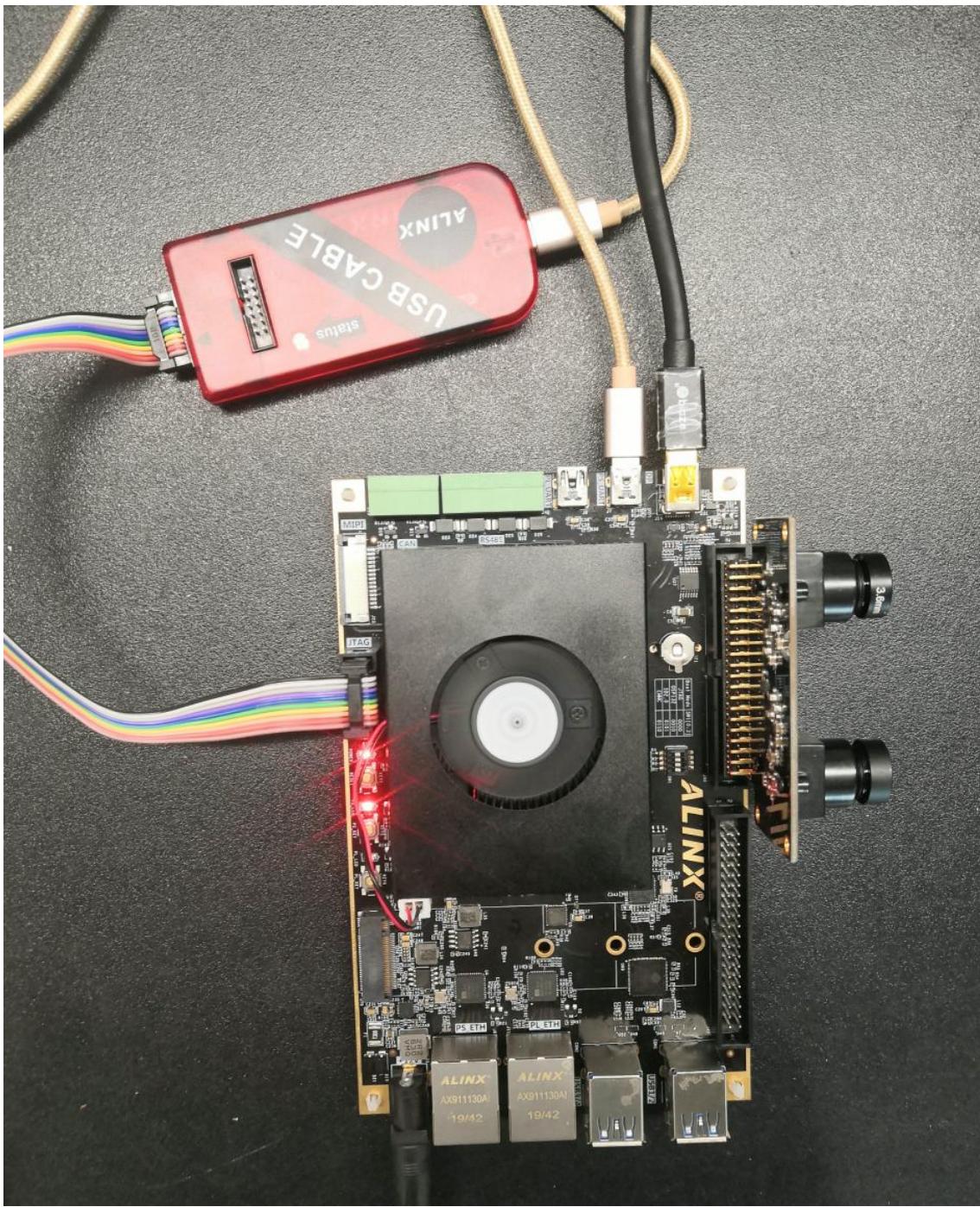
```
while(1)
{
    if (key_flag == 2)
    {
        KeyFlagHold = 0;
        /*
         * increment of photo name
         */
        PhotoCount++ ;
        sprintf(PhotoName, "%04u.bmp", PhotoCount) ;
        for(i = 0;i < 8;i++)
            PhotoPath[i+3] = PhotoName[i] ;
        /* Set PS LED on */
        XGpioPs_WritePin(&GpioInstance, PS_LED_MIO, 1) ;
        printf("Successfully Take Photo, Photo Name is %s\r\n", PhotoName) ;
        printf("Write to SD Card... \r\n") ;
        /*
         * Set timer
         */
        XTime_SetTime(0) ;
        XTime_GetTime(&TimerStart) ;

        Xil_DCacheInvalidateRange((INTPTR) pFrames[(wr_index+1)%3], (INTPTR)DEMO_MAX_FRAME) ;
        /*
         * Copy frame data to photo buffer
         */
        memcpy(&photobuf, pFrames[(wr_index+1)%3], DEMO_MAX_FRAME) ;

        /*
         * Clear key flag
         */
        key_flag = 0 ;
        /*
         * Write to SD Card
         */
        bmp_write(PhotoPath, (char *)&BMODE_1280x720, (char *)&photobuf, DEMO_STRIDE, &fil) ;
        /*
         * Print Elapsed time
         */
        XTime_GetTime(&TimerEnd) ;
        elapsed_time = ((float)(TimerEnd-TimerStart))/((float)COUNTS_PER_SECOND) ;
        printf("INFO:Elapsed time = %.2f sec\r\n", elapsed_time) ;
    }
    /* Set PS LED off */
    XGpioPs_WritePin(&GpioInstance, PS_LED_MIO, 0) ;
    KeyFlagHold = 1;
}
```

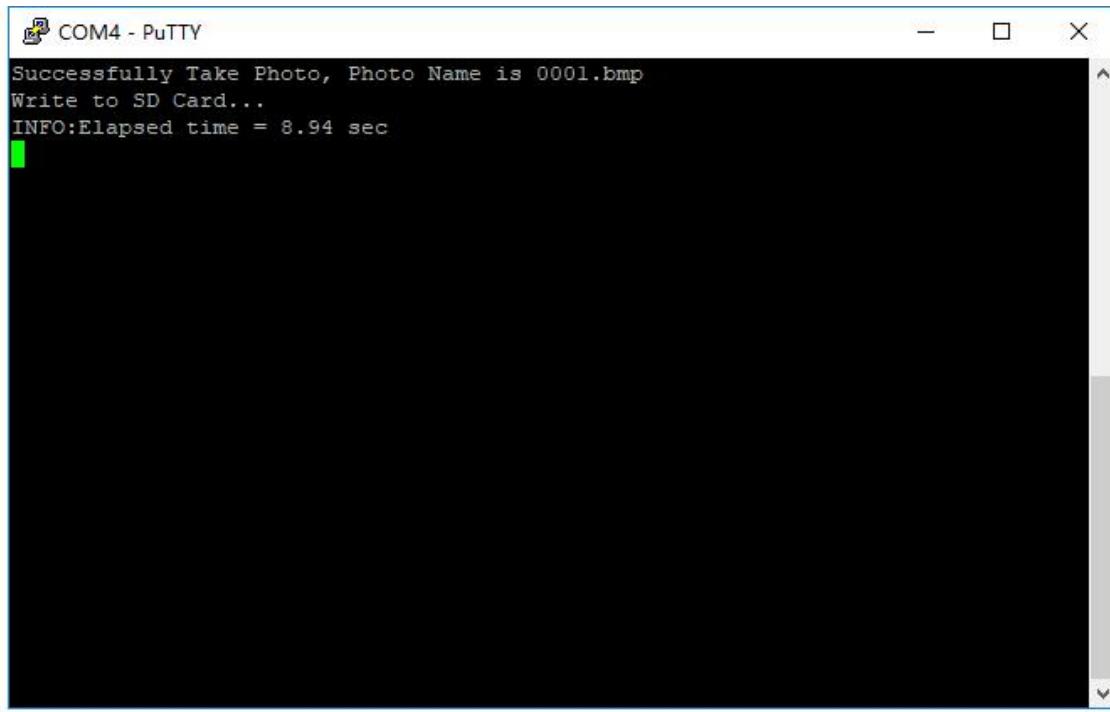
### Part 21.3: Onboard Verification

- 1) Connect the FPGA development board as shown below, insert the SD card into the SD card holder of the FPGA development board, turn on the power, and turn on the “putty”.



Hardware Connection (Expansion Port J46)

- 2) Download the program, after the image is displayed, press the key, the PS LED will light when writing SD, and will be extinguished after writing.
- 3) In putty, you can see the print information



- 4) Remove the SD card after power off, and you can see the BMP picture captured in the SD card on the computer.

名称	修改日期	类型	大小
0001.BMP	2010/1/1 0:00	BMP 文件	2,701 KB
0002.BMP	2010/1/1 0:00	BMP 文件	2,701 KB
0003.BMP	2010/1/1 0:00	BMP 文件	2,701 KB

## Part 30: Binocular camera Ethernet transmission

The experimental Vivado project directory is "an5642\_lwip\_double /vivado".

The experiment vitis project directory is "an5642\_lwip\_double /vitis".

The HDMI display of the OV5640 camera is introduced in the previous section. However, in some cases, the video needs to be transmitted to the host computer, and the data can be transmitted by using Ethernet. This chapter uses the udp of LWIP to transmit the camera data to the host computer.



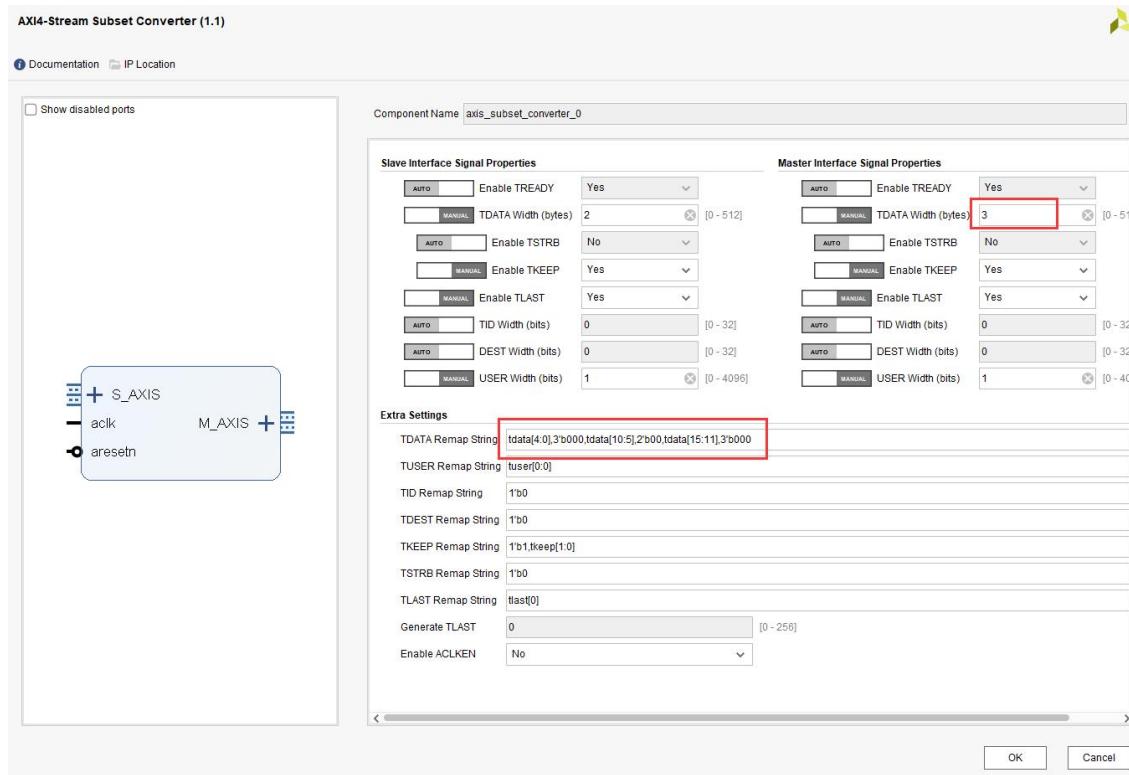
Host computer display effect

### FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

## Part 30.1: Hardware environment

Based on the acquisition and display project of the AN5642 binocular camera, because the host computer processes the data according to RGB, which is 3 bytes, the two AXI4-Stream Subset Converters are mainly modified as follows



Regenerate bitstream and export hardware information

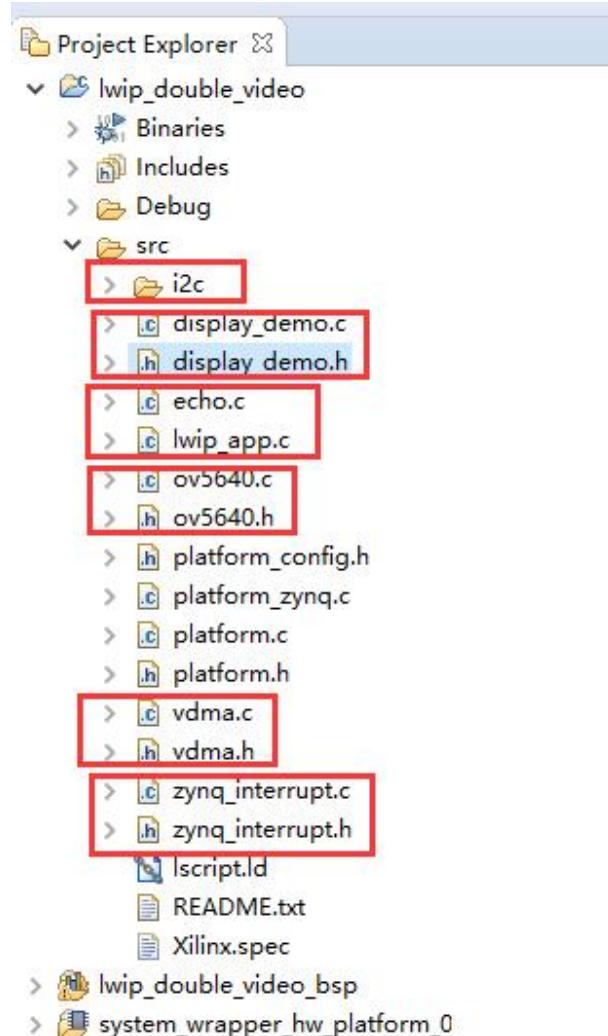
## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

## Part 30.2: Vitis program development

The program design is more complicated than before, mainly divided into i2c control, main control module display\_demo.c; lwip control module echo.c, lwip\_app.c; ov5640 module ov5640.c; vdma module vdma.c; interrupt module zynq\_interrupt.c. The whole

program is mainly divided into two parts, one is the control of the image, and the other is the transmission of Ethernet data.



### Part 30.2.1: Image control section

- 1) Both the OV5640 module and the VDMA module have been used before. In this experiment, a VDMA interrupt is added, and the interrupt controller is initialized in `zynq_interrupt.c`

```

#include "xparameters.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "zynq_interrupt.h"
@ int InterruptSystemSetup(XScuGic *XScuGicInstancePtr)
{
    // Register GIC interrupt handler
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,(Xil_ExceptionHandler)XScuGic_InterruptHandler,XScuGicInstancePtr);
    Xil_ExceptionEnable();
    return XST_SUCCESS;
}

@ int InterruptInit(u16 DeviceId,XScuGic *XScuGicInstancePtr)
{
    XScuGic_Config *IntcConfig;
    int status;
    // Interrupt controller initialization
    IntcConfig = XScuGic_LookupConfig(DeviceId);
    status = XScuGic_CfgInitialize(XScuGicInstancePtr, IntcConfig, IntcConfig->CpuBaseAddress);
    if(status != XST_SUCCESS) return XST_FAILURE;

    // Call interrupt setup function
    status = InterruptSystemSetup(XScuGicInstancePtr);
    if(status != XST_SUCCESS) return XST_FAILURE;
    return XST_SUCCESS;
}

@ int InterruptConnect(XScuGic *XScuGicInstancePtr,u32 Int_Id,void * Handler,void *CallBackRef)
{
    int status;
    // Register GPIO interrupt handler
    status = XScuGic_Connect(XScuGicInstancePtr,Int_Id,(Xil_InterruptHandler)Handler,CallBackRef);
    if(status != XST_SUCCESS) return XST_FAILURE;
    XScuGic_Enable(XScuGicInstancePtr, Int_Id);
    return XST_SUCCESS;
}

```

- 2) In the main function of display\_demo.c, the initialization of the three-frame buffer is performed. The DISPLAY\_NUM\_FRAMES macro definition is set to 3, the initial initialization of pFrame0 and pFrame1 pointers, pFrame0 points to the cache of camera 1, and pFrame1 points to the cache of camera 2, clear the cached data.

```

int i;
/*
 * Initialize an array of pointers to the 3 frame buffers
 */
for (i = 0; i < DISPLAY_NUM_FRAMES; i++)
{
    pFrames0[i] = frameBuf0[i];
    pFrames1[i] = frameBuf1[i];
    memset(pFrames0[i], 0, DEMO_MAX_FRAME);
    memset(pFrames1[i], 0, DEMO_MAX_FRAME);
}

```

- 3) Next is the initialization interrupt, cmos reset, initialize i2c, configure the sensor, reset the image, set to 1280\*720

```
/*
 * Interrupt initialization
 */
InterruptInit(XPAR_XSCUTIMER_0_DEVICE_ID,&XScuGicInstance);
/*
 * cmos reset
 */
/*initialize GPIO*/
XGpio_Initialize(&cmos_rstn, XPAR_CMOS_RST_DEVICE_ID);
/* set GPIO as output */
XGpio_SetDataDirection(&cmos_rstn, 1, 0x0);
XGpio_DiscreteWrite(&cmos_rstn, 1, 0x3);
usleep(500000);
/* set GPIO output value to 0 */
XGpio_DiscreteWrite(&cmos_rstn, 1, 0x0);
usleep(500000);
XGpio_DiscreteWrite(&cmos_rstn, 1, 0x3);
usleep(500000);
/*
 * initial i2c and sensor
 */
i2c_init(&ps_i2c0, XPAR_XIICPS_0_DEVICE_ID,40000);
i2c_init(&ps_i2c1, XPAR_XIICPS_1_DEVICE_ID,40000);
sensor_init(&ps_i2c0);
sensor_init(&ps_i2c1);
/*
 * Reconfiguration Sensor and VDMA
 */
resetVideoFmt(1280, 720, 0);
resetVideoFmt(1280, 720, 1);
```

- 4) In the resetVideoFmt function, stop vdma, close the interrupt, and reconfigure the resolution of the sensor according to the values of w, h, and ch.

```
void resetVideoFmt(int w, int h, int ch)
{
    frame_length_curr = 0;
    /* Stop vdma write process, disable vdma interrupt */
    vdma_write_stop(&vdma_vin[ch]);
    XAxiVdma_IntrDisable(&vdma_vin[ch], XAXIVDMA_IXR_ALL_MASK, XAXIVDMA_WRITE);

    /* reconfig Sensor horizontal width and vertical height */
    if(ch == 0)
    {
        i2c_reg16_write(&ps_i2c0, 0x3c, 0x3808, (w>>8)&0xff);
        i2c_reg16_write(&ps_i2c0, 0x3c, 0x3809, (w>>0)&0xff);
        i2c_reg16_write(&ps_i2c0, 0x3c, 0x380a, (h>>8)&0xff);
        i2c_reg16_write(&ps_i2c0, 0x3c, 0x380b, (h>>0)&0xff);
    }
    else
    {
        i2c_reg16_write(&ps_i2c1, 0x3c, 0x3808, (w>>8)&0xff);
        i2c_reg16_write(&ps_i2c1, 0x3c, 0x3809, (w>>0)&0xff);
        i2c_reg16_write(&ps_i2c1, 0x3c, 0x380a, (h>>8)&0xff);
        i2c_reg16_write(&ps_i2c1, 0x3c, 0x380b, (h>>0)&0xff);
    }
}
```

And initialize the vdma write channel, write the three cache addresses of pFrame to the S2MM StartAddresses register. Use the xAxiVdma\_SetCallBack function to set the callback function, WriteCallBack and WriteErrorCallBack

```
/*
 * Initial vdma write path, set call back function and register interrupt to GIC
 */
if(ch == 0)
{
    vdma_write_init(XPAR_AXIVDMA_0_DEVICE_ID, &vdma_vin[ch], w * 3, h, w * 3,
        (unsigned int)pFrames0[0], (unsigned int)pFrames0[1], (unsigned int)pFrames0[2]);
    XAxiVdma_SetCallBack(&vdma_vin[ch], XAXIVDMA_HANDLER_GENERAL,WriteCallBack0, (void *)&vdma_vin[ch], XAXIVDMA_WRITE);
    XAxiVdma_SetCallBack(&vdma_vin[ch], XAXIVDMA_HANDLER_ERROR,WriteErrorCallBack, (void *)&vdma_vin[ch], XAXIVDMA_WRITE);
    InterruptConnect(&XScuGicInstance,XPAR_FABRIC_AXI_VDMA_0_S2MM_INTROUT_INTR,XAxiVdma_WriteIntrHandler,(void *)&vdma_vin[ch]);
}
else
{
    vdma_write_init(XPAR_AXIVDMA_1_DEVICE_ID, &vdma_vin[ch], w * 3, h, w * 3,
        (unsigned int)pFrames1[0], (unsigned int)pFrames1[1], (unsigned int)pFrames1[2]);
    XAxiVdma_SetCallBack(&vdma_vin[ch], XAXIVDMA_HANDLER_GENERAL,WriteCallBack1, (void *)&vdma_vin[ch], XAXIVDMA_WRITE);
    XAxiVdma_SetCallBack(&vdma_vin[ch], XAXIVDMA_HANDLER_ERROR,WriteErrorCallBack, (void *)&vdma_vin[ch], XAXIVDMA_WRITE);
    InterruptConnect(&XScuGicInstance,XPAR_FABRIC_AXI_VDMA_1_S2MM_INTROUT_INTR,XAxiVdma_WriteIntrHandler,(void *)&vdma_vin[ch]);
}
/* Start vdma write process, enable vdma interrupt */
XAxiVdma_IntrEnable(&vdma_vin[ch], XAXIVDMA_IXR_ALL_MASK, XAXIVDMA_WRITE);
vdma_write_start(&vdma_vin[ch]);
frame_length_curr = w*h*3;
```

- 5) The WriteCallBack callback function is used to determine whether the interrupt status is a FrameCount interrupt. If so, add wr\_index to 1, and use XAxiVdma\_StartParking to set the WrFrmPtrRef of the park pointer register. Modify the current cached Start Addresses, the parameter is wr\_index[0]; there is a very important variable WriteOneFrameEnd, the initial value is -1, which can be simply understood as the end of a frame write, this signal is used for the interaction between image processing and LWIP I will talk about it later in the LWIP processing section.

```
static void WriteCallBack0(void *CallbackRef, u32 Mask)
{
    if (Mask & XAXIVDMA_IXR_FRMCNT_MASK)
    {
        if(WriteOneFrameEnd[0] >= 0)
        {
            return;
        }
        int hold_rd = rd_index[0];
        if(wr_index[0]==2)
        {
            wr_index[0]=0;
            rd_index[0]=2;
        }
        else
        {
            rd_index[0] = wr_index[0];
            wr_index[0]++;
        }
        /* Set park pointer */
        XAxiVdma_StartParking((XAxiVdma*)CallbackRef, wr_index[0], XAXIVDMA_WRITE);
        WriteOneFrameEnd[0] = hold_rd;
    }
}
```

### Part 30.2.2: LWIP Control Section

1) The following describes the contents of the LWIP. When communicating with the host computer, UDP transmission is used, and the protocol is customized in the UDP packet as follows:

1) Get board information

a) **Inquiry command** (5 bytes in total, sent by the host computer via Ethernet)

Number of bytes	1	4
Command information	Header	0x00020001

b) **Answer command** (16 bytes in total, sent by the development board via Ethernet)

Number of bytes	Command Information
1	Header 0x01
4	0x00020001
6	Board MAC address
4	Board IP address
1	0x02

2) Obtain data

a) **Control command** (send data request by host computer)

Number of bytes	Command Information
1	Header
4	0x00020002
6	Board MAC address
1	Camera channel selection, the value 1 means that only the camera 1 is turned on, the value 2 means that only the camera 2 is turned on, and the value 3 means that the two cameras are turned on at the same time.
1	Start signal, 0 means to turn off the upper image display, other means to turn on the image display

b) **Answer command** (sent by the development board)

Number of bytes	Command Information
1	Header 0x 01
3	0x 000200
1	Channel identification, the value 2 represents the camera 1, and the value 3 represents the camera 2
3	Serial number, Ethernet packet serial number, used for host computer identification
N	Image data

Each UDP packet contains a header. In the first byte, the format is as follows:

Bit	Value (0)	Value (1)
bit 0	Query or control	Answer
bit1~bit7	Random data	

Note: When answering, the high 7-bit random data remains unchanged, bit0 is set to 1

The workflow is:

- 1) The host computer sends an inquiry command
  - 2) Development board answering inquiry
  - 3) The host computer sends a control command request data
  - 4) Development board sends data
  - 5) Loop steps 3 and 4
- 2) The LWIP control part mainly consists of two parts, one is the udp read/write part echo.c, and the other is the interactive part of the image cache lwip\_app.c. Before understanding the program, first need to understand several structures, netif, udp\_pcb, pbuf. In Lwip, many structures exist in a linked list.

Each network interface has a corresponding structure netif

representation, which is the protocol stack and the underlying driver interface module. The next structure in the linked list, IP address, subnet mask, gateway, input function, output function, maximum transmission unit, etc. are defined in the structure. The corresponding files are netif.h and netif.c

```

④ /** Generic data structure used for all lwIP network interfaces.
 *   The following fields should be filled in by the initialization
 *   function for the device driver: hwaddr_len, hwaddr[], mtu, flags */
④ struct netif {
    /** pointer to next in linked list */
    struct netif *next;

    /** IP address configuration in network byte order */
    ip_addr_t ip_addr;
    ip_addr_t netmask;
    ip_addr_t gw;

    /** This function is called by the network device driver
     *   to pass a packet up the TCP/IP stack. */
    netif_input_fn input;
    /** This function is called by the IP module when it wants
     *   to send a packet on the interface. This function typically
     *   first resolves the hardware address, then sends the packet. */
    netif_output_fn output;
    /** This function is called by the ARP module when it wants
     *   to send a packet on the interface. This function outputs
     *   the pbuf as-is on the link medium. */
    netif_linkoutput_fn linkoutput;

```

---

- 3) The pbuf structure is used to store data received or sent, and is also in the form of a linked list. Pbuf \*next points to the address of the next pbuf; payload points to the address of the payload data. For example, udp is to remove the valid data of the frame header, IP header, and udp header; tot\_len is the sum of the current data plus all the linked list pbuf data. If there is no linked list, tot\_len is equal to len; len refers to the current pbuf data length; Type refers to the pbuf type, which is divided into PBUF\_RAM, PBUF\_ROM, PBUF\_REF and PBUF\_POOL;

Related files are pbuf.h and pbuf.c

```

struct pbuf {
    /** next pbuf in singly linked pbuf chain */
    struct pbuf *next;

    /** pointer to the actual data in the buffer */
    void *payload;

    /**
     * total length of this buffer and all next buffers in chain
     * belonging to the same packet.
     *
     * For non-queue packet chains this is the invariant:
     * p->tot_len == p->len + (p->next? p->next->tot_len: 0)
     */
    u16_t tot_len;

    /** length of this buffer */
    u16_t len;

    /** pbuf_type as u8_t instead of enum to save space */
    u8_t /*pbuf_type*/ type;

    /** misc flags */
    u8_t flags;

    /**
     * the reference count always equals the number of pointers
     * that refer to this pbuf. This can be pointers from an application,
     * the stack itself, or pbuf->next pointers from a chain.
     */
    u16_t ref;
};

```

- 4) Udp\_pcb refers to the protocol control block of udp. The main members include, the next pcb, the local port number, the peer port number, the receive callback function, etc. The related files are udp.h and udp.c.

```

@struct udp_pcb {
    /* Common members of all PCB types */
    IP_PCB;

    /* Protocol specific PCB members */

    struct udp_pcb *next;

    u8_t flags;
    /** ports are in host byte order */
    u16_t local_port, remote_port;

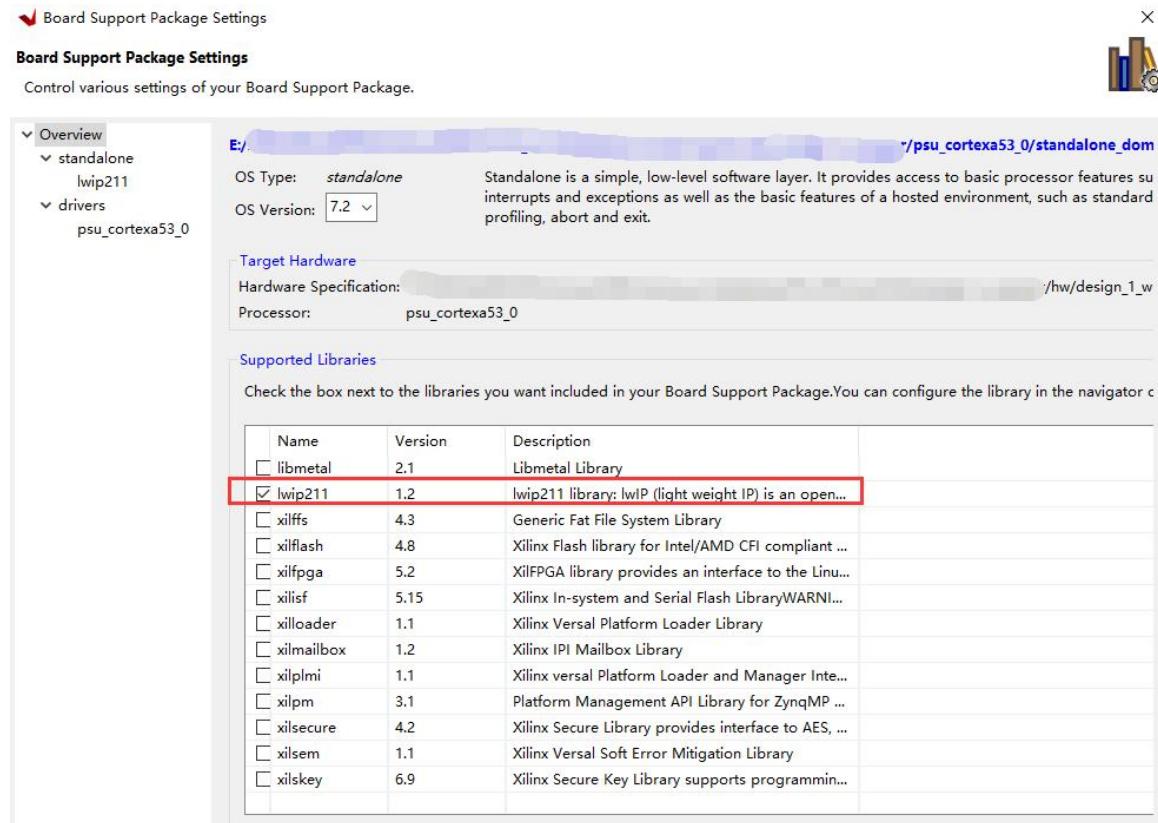
#ifndef LWIP_IGMP
    /** outgoing network interface for multicast packets */
    ip_addr_t multicast_ip;
#endif /* LWIP_IGMP */

#ifndef LWIP_UDPLITE
    /** used for UDP_LITE only */
    u16_t checksum_len_rx, checksum_len_tx;
#endif /* LWIP_UDPLITE */

    /** receive callback function */
    udp_recv_fn recv;
    /** user-supplied argument for the recv callback */
    void *recv_arg;
};

```

5) After some concepts about the above structure, the following describes the use of the program part. First need to set lwip, click to open Board Support Package Settings. After opening, click lwip211, api\_mode is set to RAW API, dhcp\_options opens the dhcp function, pbuf\_options option sets pbuf\_pool\_size larger to increase the cache space and improve efficiency. Click OK



Board Support Package Settings

Control various settings of your Board Support Package.

Overview

- standalone
  - lwip211**
- drivers
  - ps7\_cortexa9\_0

Configuration for library: **lwip211**

Name	Value	Default	Type	Description
api_mode	RAW API (RAW_API)	RAW_API	enum	Mode of operation for lwIP (I
lwip_tcp_keepalive	false	false	boolean	Enable keepalive processing
no_sys_no_timers	true	true	boolean	Drops support for sys_timeo
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	1	1	integer	Option if set to 1 ensures axi
use_emaclite_on_zynq	1	1	integer	Option if set to 1 ensures em
arp_options	true	true	boolean	ARP Options
debug_options	true	true	boolean	Turn on lwIP Debug?
dhcp options	true	true	boolean	Is DHCP required?
<b>dhcp_does_arp_check</b>	<b>true</b>	false	boolean	ARP check on offered address
<b>lwip_dhcp</b>	<b>true</b>	false	boolean	Is DHCP required?
icmp_options	true	true	boolean	ICMP Options
igmp_options	false	false	boolean	IGMP Options
lwip_ip_options	true	true	boolean	IP Options
ipv6_enable	false	false	boolean	IPv6 enable value
lwip_memory_options				Options controlling lwIP mem
mbox_options	true	true	boolean	Mbox Options
pbuf_options	true	true	boolean	Pbuf Options
<b>pbuf_link_hlen</b>	<b>16</b>	16	integer	Number of bytes that should
<b>pbuf_pool_bufsize</b>	<b>2000</b>	1700	integer	Size of each pbuf in pbuf po
<b>pbuf_pool_size</b>	<b>4096</b>	256	integer	Number of buffers in pbuf po
stats_options	true	true	boolean	Turn on lwIP statistics?
tcp_options	true	true	boolean	Is TCP required ?
temac_adapter_options	true	true	boolean	Settings for xps-ll-temac/Axi-
udp_options	true	true	boolean	Is UDP required ?

- 6) Next, the lwip is initialized. In the `lwip_app.c` file, the `lwip_loop` function is initialized. The `echo_netif` is a pointer to the defined netif structure type. First, set the MAC address, IP address, subnet mask, and gateway information of the board. Use the `xemac_add` function to add to the netif structure, and use `netif_set_default` to set `echo_netif` as the default NIC. Open this network port with `netif_set_up`.

```

int lwip_loop()
{
    struct ip_addr ipaddr, netmask, gw;

    /* the mac address of the board. this should be unique per board */
    unsigned char mac_ethernet_address[] = { 0x00, 0xa, 0x35, 0x00, 0x01, 0x02 };
    echo_netif = &server_netif;
    init_platform();

#ifndef LWIP_DHCP
    ipaddr.addr = 0;
    gw.addr = 0;
    netmask.addr = 0;
#else
    /* initialize IP addresses to be used */
    IP4_ADDR(&ipaddr, 192, 168, 1, 11);
    IP4_ADDR(&netmask, 255, 255, 255, 0);
    IP4_ADDR(&gw, 192, 168, 1, 1);
#endif
    print_app_header();

    lwip_init();

    /* Add network interface to the netif_list, and set it as default */
    if (!xemac_add(echo_netif, &ipaddr, &netmask, &gw, mac_ethernet_address, PLATFORM_EMAC_BASEADDR)) {
        xil_printf("Error adding N/W interface\n\r");
        return -1;
    }
    netif_set_default(echo_netif);

    /* now enable interrupts */
    platform_enable_interrupts();

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

```

## Make DHCP settings

```
#if (LWIP_DHCP==1)
    /* Create a new DHCP client for this interface.
     * Note: you must call dhcp_fine_tmr() and dhcp_coarse_tmr() at
     * the predefined regular intervals after starting the client.
     */
    dhcp_start(echo_netif);
    dhcp_timeoutcntr = 24;

    while(((echo_netif->ip_addr.addr) == 0) && (dhcp_timeoutcntr > 0))
        xemacif_input(echo_netif);

    if (dhcp_timeoutcntr <= 0) {
        if ((echo_netif->ip_addr.addr) == 0) {
            xil_printf("DHCP Timeout\r\n");
            xil_printf("Configuring default IP of 192.168.1.11\r\n");
            IP4_ADDR(&(echo_netif->ip_addr), 192, 168, 1, 11);
            IP4_ADDR(&(echo_netif->netmask), 255, 255, 255, 0);
            IP4_ADDR(&(echo_netif->gw), 192, 168, 1, 1);
        }
    }

    ipaddr.addr = echo_netif->ip_addr.addr;
    gw.addr = echo_netif->gw.addr;
    netmask.addr = echo_netif->netmask.addr;
#endif
```

At this point, the setup is basically complete.

- 7) After that, start calling the udp application function, which is defined in the echo.c file.

```
/* start the application (web server, rxtest, txttest, etc..) */
start_udp8080();
```

In this function, first create a pcb structure with udp\_new, return a pointer of type udp\_pcb, and assign it to udp8080\_pcb; use udp\_bind function to bind the address and port to the structure, there are three parameters, the first one is pcb structure Variables, the second is the IP address, and each of the three is the port number. Usually the IP address is filled in with IP\_ADDR\_ANY. Use the udp\_recv function to bind the callback function. The callback function bound in this experiment is udp\_receive

```

int start_udp8080()
{
    err_t err;
    unsigned port = 8080;
    /* Create new pcb, allocate memory space to pcb */
    udp8080_pcb = udp_new();
    if (!udp8080_pcb) {
        xil_printf("Error creating PCB. Out of Memory\n\r");
        return -1;
    }
    /* bind to specified @port */
    err = udp_bind(udp8080_pcb, IP_ADDR_ANY, port);
    if (err != ERR_OK) {
        xil_printf("Unable to bind to port %d: err = %d\n\r", port, err);
        return -2;
    }
    /* Set call back function for udp receive */
    udp_recv(udp8080_pcb, udp_receive, 0);
    IP4_ADDR(&target_addr, 192,168,1,35);

    return 0;
}

```

The above is the initialization process of udp.

- 8) The udp receiving function is udp\_receive, which has been set to the udp receiving callback function. The function is the received host udp command and determines whether it is a custom protocol. If it is a query command, the response is initiated. If it is a control command, reset the camera resolution according to the command.

```

void udp_receive(void *arg,
                 struct udp_pcb *pcb,
                 struct pbuf *p_rx,
                 struct ip_addr *addr,
                 u16_t port)

```

- 9) The transfer\_data function is called in the udp\_receive function to send “udp” data. The parameter “pData” points to the address where the data will be transmit, and len is the length of the transmitted data.

```

int transfer_data(const char *pData, int len, struct ip_addr *addr)
{

```

First, determine whether “len” is greater than “udp8080\_qlen”. If yes, use “pbuf\_alloc” to re-allocate space to “pbuf” “udp8080\_q”. The parameter of “pbuf\_alloc” is an enumerated type. You can use F3 to find out which enumeration members are available. Then copy the data to the “payload” of “udp8080\_q” and assign values to “len” and “tot\_len”. Use “udp\_sendto” to start sending data.

10)The receiving and sending part of the command has been finished.

Let's learn how to send image data. Sending images using “sendpic” in “echo.c” is similar to the “transfer\_data” function, but since the image data may vary from package to packet, “pbuf” is released each time, and the image header “targetPicHeader” is added.

```
④ int sendpic(const char *pic, int piclen, int sn)
{
    if(!targetPicHeader[0])
    {
        return -1;
    }
    targetPicHeader[5] = sn>>16;
    targetPicHeader[6] = sn>>8;
    targetPicHeader[7] = sn>>0;

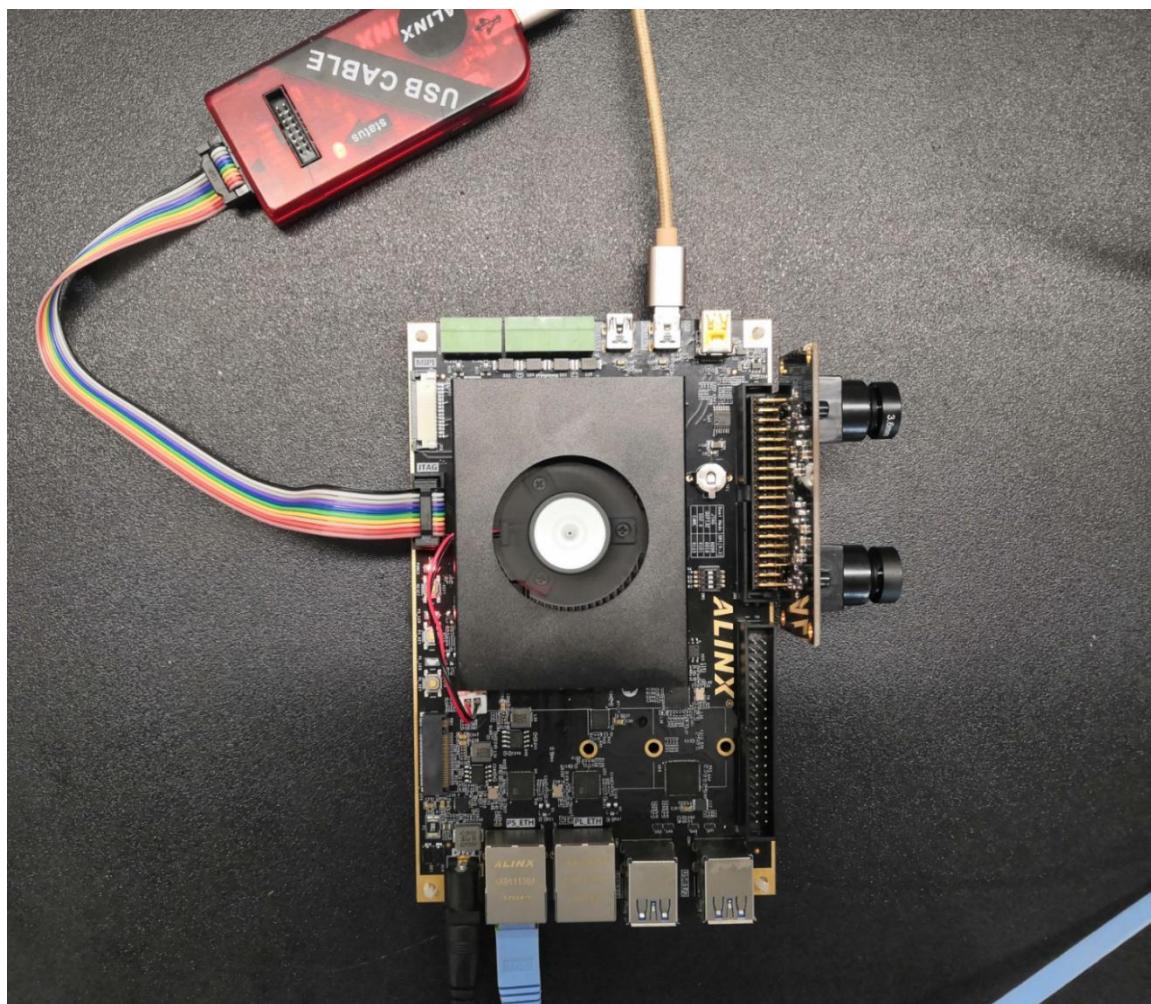
    struct pbuf *q;
    q = pbuf_alloc(PBUF_TRANSPORT, 8+piclen, PBUF_POOL);
    if(!q)
    {
        //xil_printf("pbuf_alloc %d fail\n\r", piclen+8);
        return -3;
    }
```

11)Back to “lwip\_app.c” file, still “lwip\_loop” function, there is a “while” loop, use “xemacif\_input” to start data reception, here uses the interaction variable “WriteOneFrameEnd” with the image, if the image is received and “sendchannel[0]” is valid, start dividing Package, call “sendpic” to send the image. The following “if” is to determine if the second camera is valid. At this point, the transmission of the image data is completed

```
while (1) {
    xemacif_input(echo_netif);
    if((WriteOneFrameEnd[0] >= 0) && (sendchannel[0]) )
    {
        targetPicHeader[4] = 2;
        index = WriteOneFrameEnd[0];
        int sn = 1;
        int cot;
        Xil_DCacheInvalidateRange((u32)pFrames0[index], frame_length_curr);
        /* Separate camera 1 frame in package */
        for(int i=0;i<frame_length_curr;i+=1440)
        {
            if((i+1440)>frame_length_curr)
            {
                cot = frame_length_curr-i;
            }
            else
            {
                cot = 1440;
            }
            sendpic((const char *)pFrames0[index]+i, cot, sn++);
        }
        WriteOneFrameEnd[0] = -1;
    }
```

### Part 30.3: Onboard verification

- 1) Before verifying, first make sure that the HDMI display of the binocular camera is ok, otherwise the experiment will not be possible.
- 2) Connect the development board as follows, insert the PS end network port, you need to ensure that the PC's network card is a Gigabit network card, otherwise the network speed will be too low, resulting in the inability to display images.

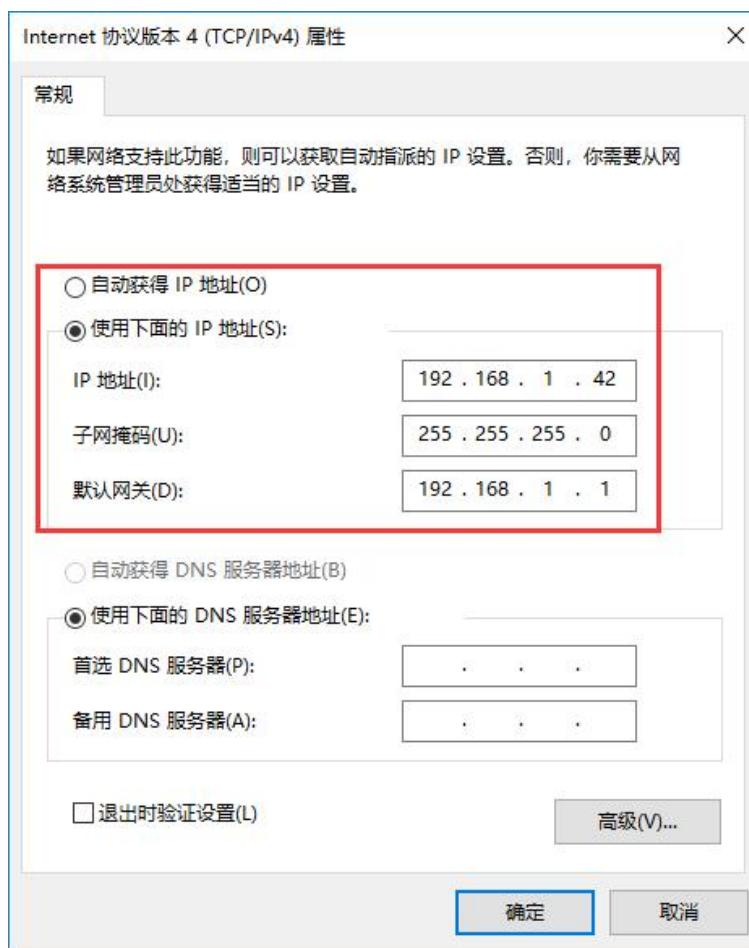


Hardware Connection (Expansion Port J46)

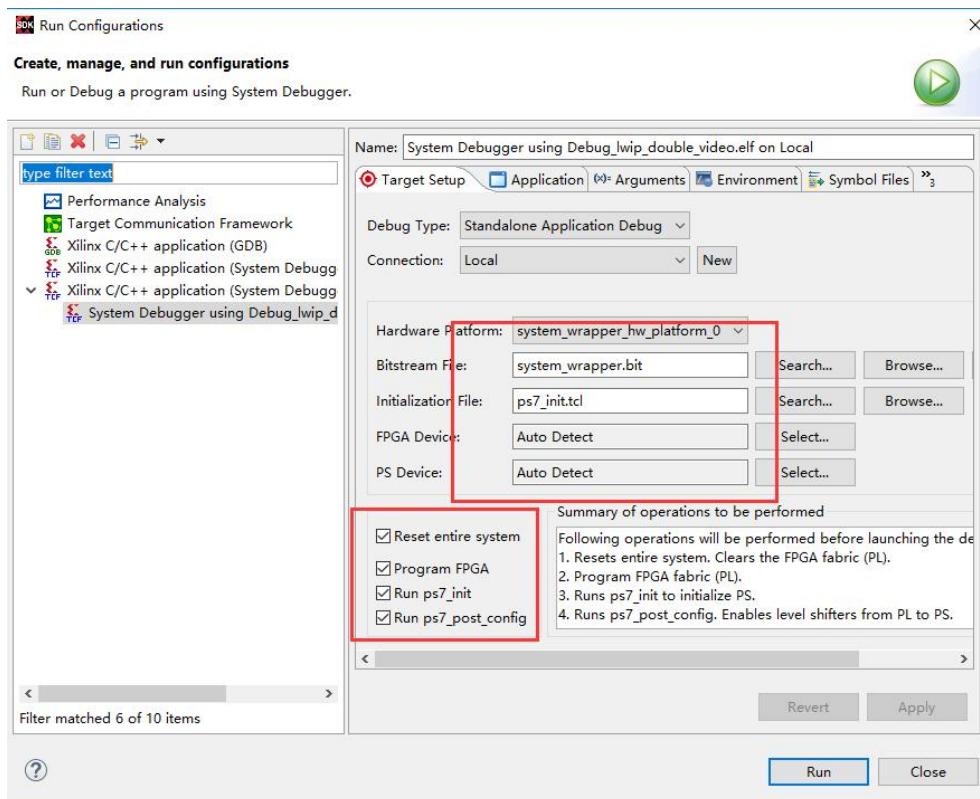
- 3) If there is a DHCP server, the IP will be automatically assigned to the development board; if there is no DHCP server, the default FPGA development board IP address is 192.168.1.11, and the IP

address of the PC needs to be set to the same network segment, as shown in the following figure. Also make sure that there is no IP address of 192.168.1.11 in the network, otherwise IP conflict will occur and the image will not be displayed. You can enter ping 192.168.1.11 in the CMD to check whether the ping can be pinged before the board is powered on. If the ping is successful, the IP address in the network cannot be verified.

Open the putty software after no problem



4) Run Configurations as below:



- 5) The serial port print information is as follows, the network card speed is detected, and the set IP address is set.

```

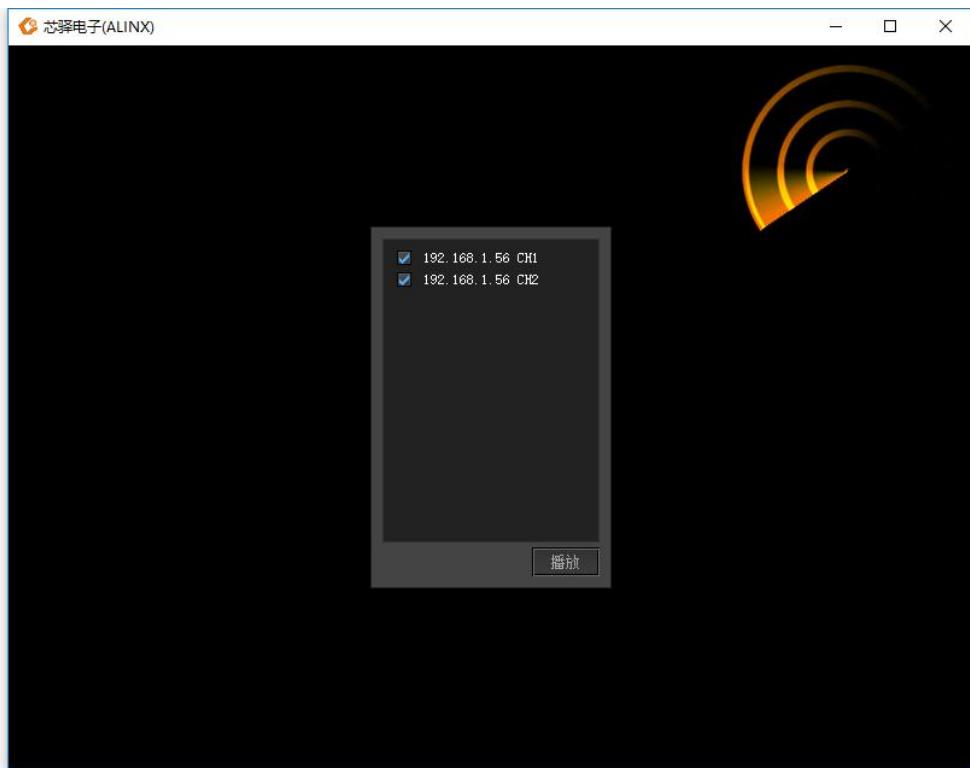
-----AN5642 lwIP UDP DEMO -----
UDP packets sent to port 8080
WARNING: Not a Marvell or TI Ethernet PHY. Please verify the initialization sequence
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 1: 1000
Board IP: 192.168.1.56
Netmask : 255.255.255.0
Gateway : 192.168.1.1

```

Open the Vivado project folder and open videoshow.exe

.Xil	2018/8/13 9:05	文件夹
an5642_lwip.cache	2018/8/13 9:05	文件夹
an5642_lwip.hw	2018/8/13 9:05	文件夹
an5642_lwip.ip_user_files	2018/8/13 9:05	文件夹
an5642_lwip.runs	2018/8/13 9:05	文件夹
an5642_lwip.sdk	2018/8/13 11:43	文件夹
an5642_lwip.sim	2018/8/13 9:05	文件夹
an5642_lwip.srcs	2018/8/13 9:05	文件夹
repo	2018/8/2 13:46	文件夹
an5642_lwip.xpr	2018/8/2 14:43	Vivado Project Fi...
<b>videoshow.exe</b>	2018/5/28 12:39	应用程序
		16,484 KB

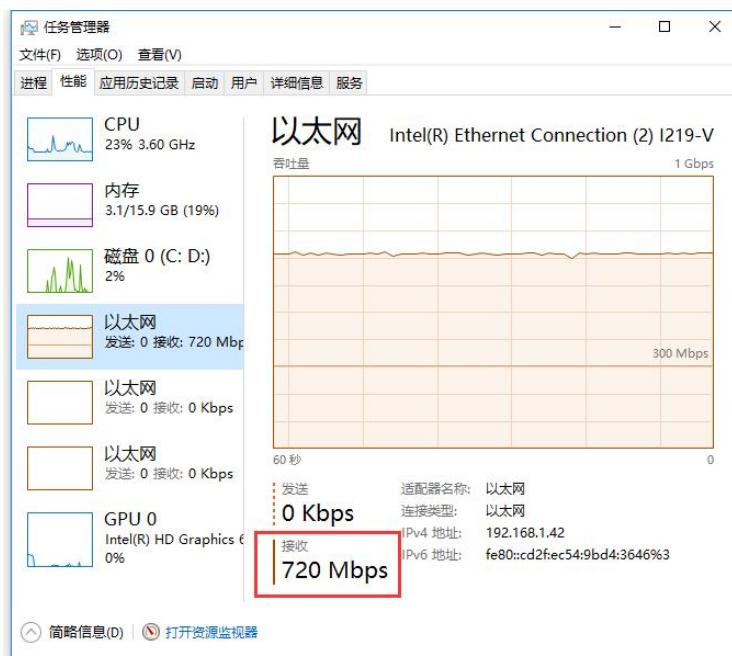
The software scans to two cameras, you can select the corresponding camera display by clicking the check, click to play



- 6) The display effect is as follows. If you want to reselect the display path, double-click on the software screen to return to the selection interface and select the image to be displayed again.



- 7) Open the task manager, you can see that the network bandwidth is about 720Mbps



## Part 30.4: Experimental Summary

In this experiment, we introduced the udp transmission video using lwip, and the content is more. If you want to use lwip well, you still need to study the internal structure and use it flexibly.

# Part 31: MIPI Acquisition and Display

## Based on AN5641 Module

The experimental Vivado project directory is "an5641\_mipi\_dp /vivado".

The experiment vitis project directory is "an5641\_mipi\_dp /vitis".

In the previous chapters, the use of the camera's DVP interface for image acquisition was introduced. This chapter introduces the MIPI CSI-2 image acquisition based on the AN5641 module. The MIPI protocol is more complicated. This chapter aims to introduce the basic concepts for users to learn in depth.

### Part 31.1: Principle Introduction

*MIPI Alliance*, Mobile Industry Processor Interface, MIPI Alliance have formulated a set of interface standards to standardize the interfaces of mobile devices, such as cameras and displays. The one used for camera capture is called the CSI interface, and the one used for display is called DSI. Since it is a camera, what needs to be learned is the CSI interface. CSI interface is divided into physical layer (D-PHY) and protocol layer (CSI-2)

#### Part 31.1.1: MIPI Physical Layer (D-PHY)

The following is the structure diagram of the physical layer. Dp/Dn is a differential interface, and the interface part is divided into **LP (low power)**, which is a low power mode, which can control the interface to enter the sleep state or switch the state. The voltage swing is 1.2V; **HS (high speed)**, the high-speed interface, is mainly used for image

transmission, with a voltage swing of 200mV. The main function of DPHY is to switch modes and convert data from serial to parallel.

The specific content can refer to the following project catalog

*mipi\_D PHY\_specification\_v01 00 00.pdf*

### Part 31.1.2: MIPI Protocol Layer (CSI-2)

The following is the structure diagram of the protocol layer. CSI-2 needs to analyze the parallel data protocol from DPHY, including bit sequence adjustment, long packet and short packet data analysis, and image data analysis.

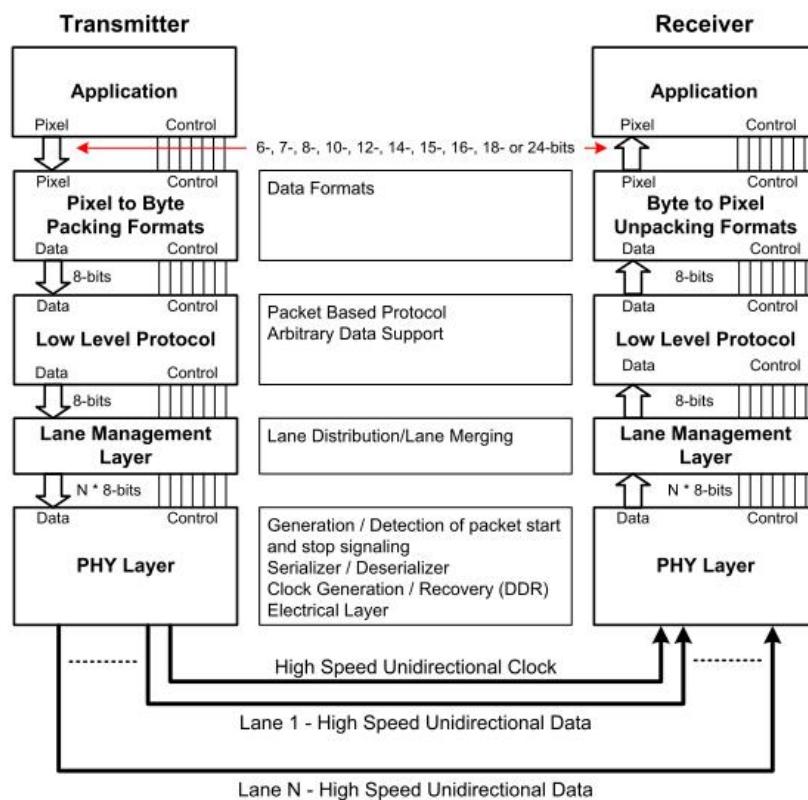


Figure 2 CSI-2 Layer Definitions

The important thing in the CSI-2 protocol layer is the short packet and the long packet. The short packet can be used to explain the frame start position and line number of the image, and is used for image synchronization. The format is as follows:

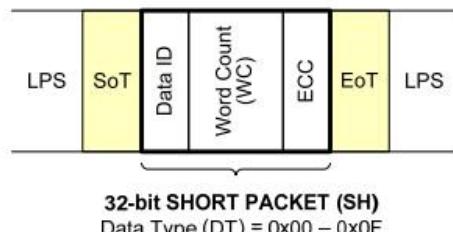


Figure 31 Short Packet Structure

Long packets are mainly used for image data transmission, and specify the image format, such as RGB888/RGB565/RAW10, etc., which can be specified by Data ID. The format is as follows:

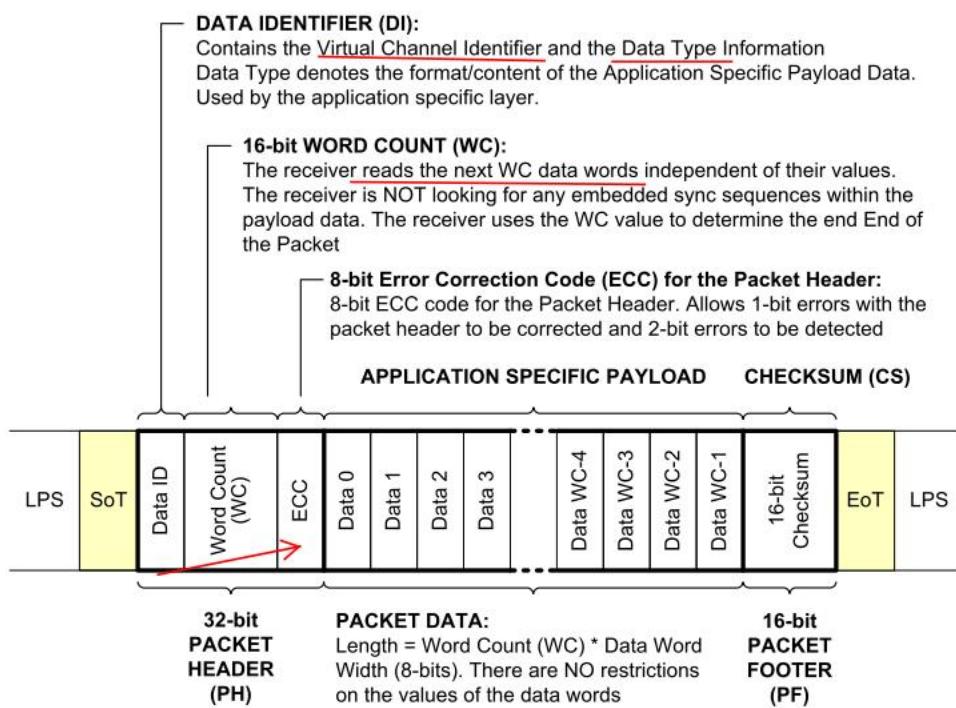


Figure 30 Long Packet Structure

For the specific content of MIPI CSI-2, please refer to

**“MIPI Alliance Specification for Camera Serial Interface 2 (CSI-2).pdf”**

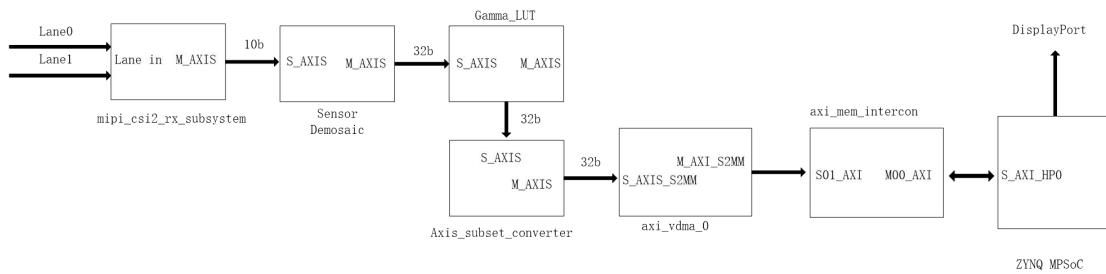
Saved in the the project catalog

## FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

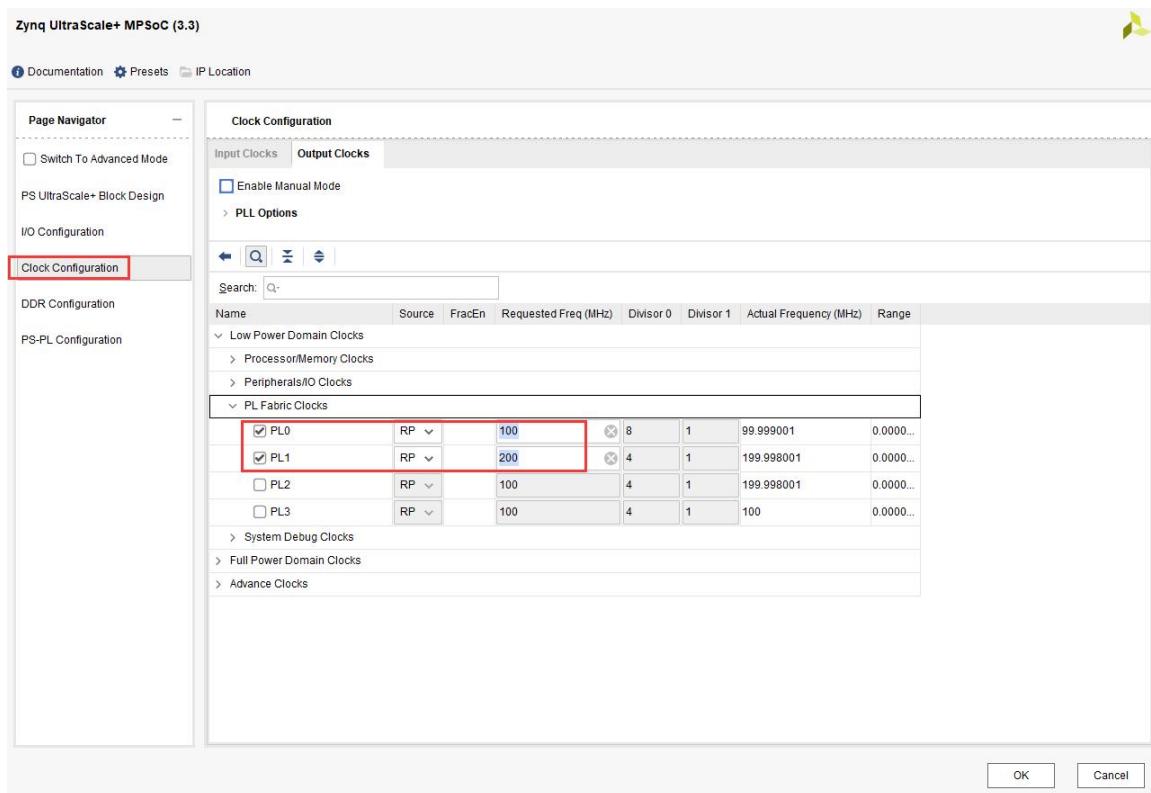
## Part 31.2: Hardware Environment

The project uses two lanes of MIPI input, and the MIPI camera is configured for RAW10 output. The `mipi_csi2_rx_subsystem` module is used for protocol analysis and converted into AXIS stream data, and the `bayerToRGB` module is used to convert RAW into RGB data. After the Gamma calibration and other modules, enter the VDMA, and then enter the HP port.

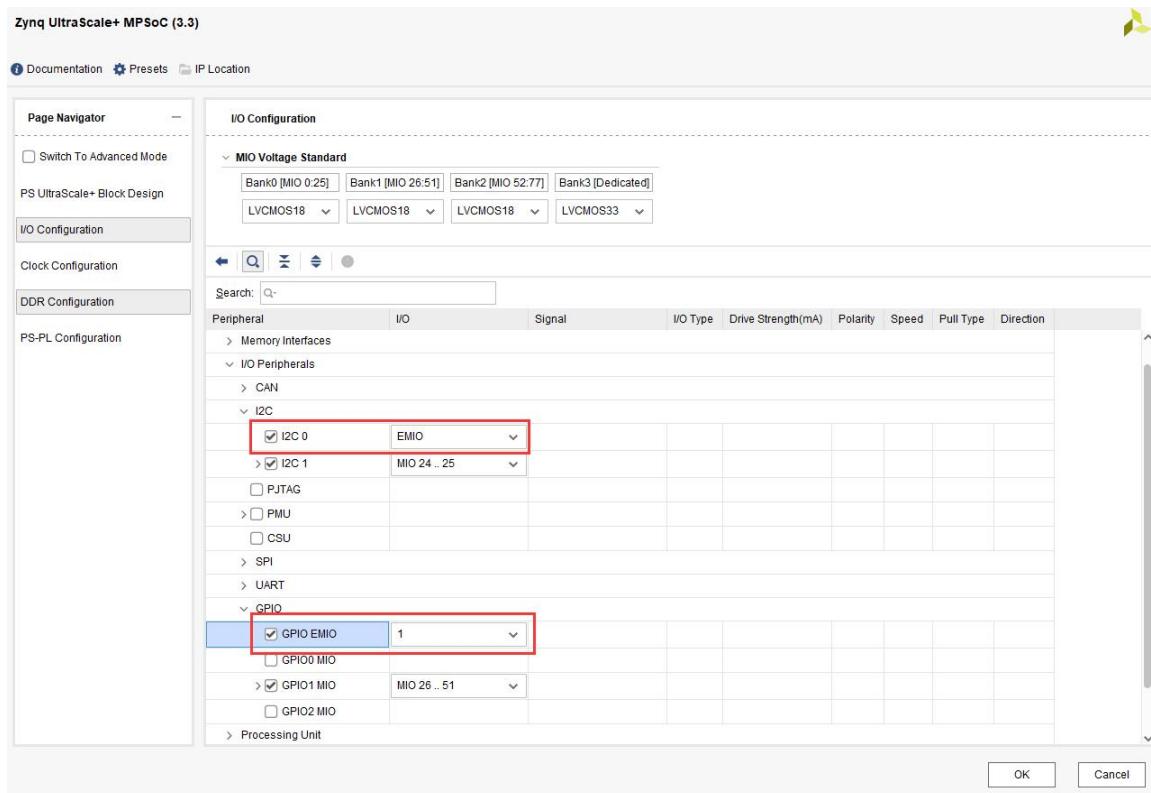


Based on the "VDMA driven HDMI display" project, we add a module for the MIPI acquisition part

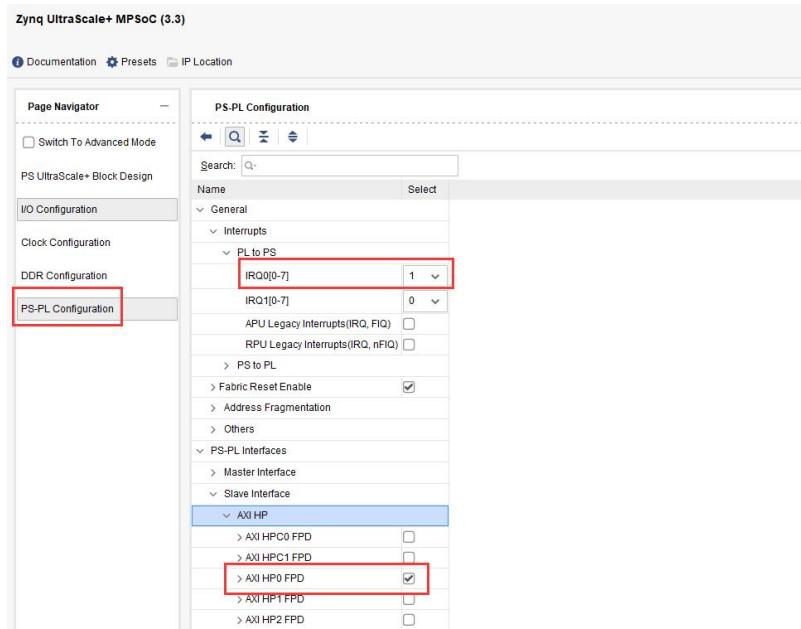
- 1) First configure the ZYNQ core, 100MHz is used for data transmission, and 200MHz is used for the MIPI module reference clock.



Configure i2c as EMIO, used to configure the camera register,  
GPIO EMIO set to 1, used to configure the camera enable



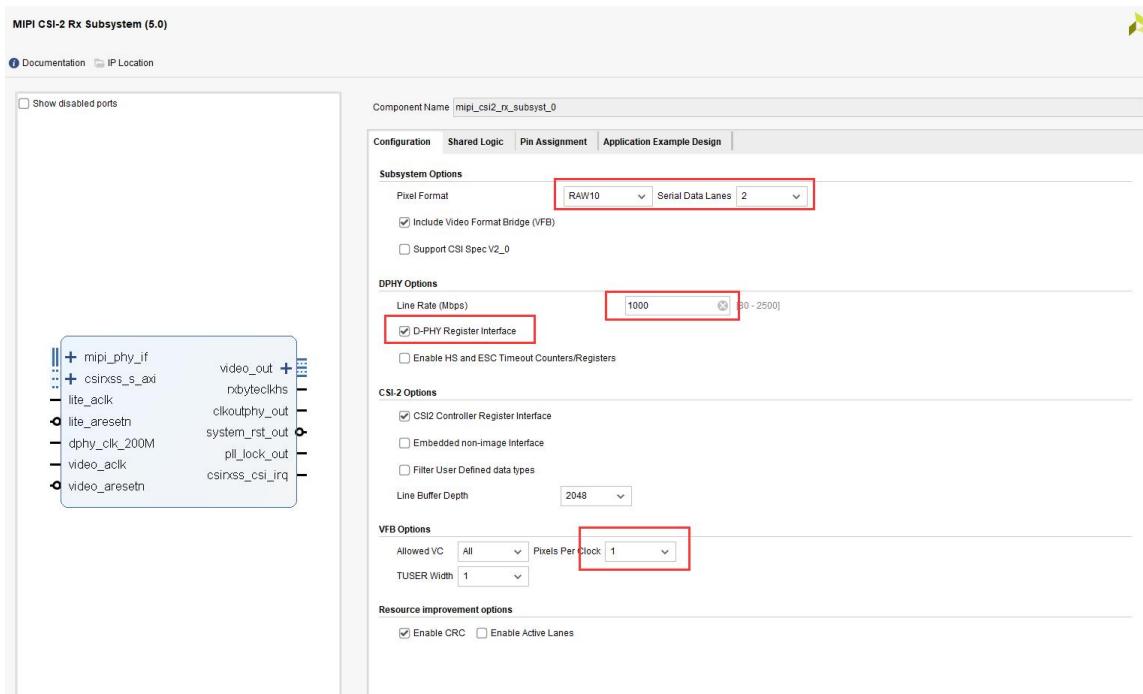
Configure interrupt and HP port



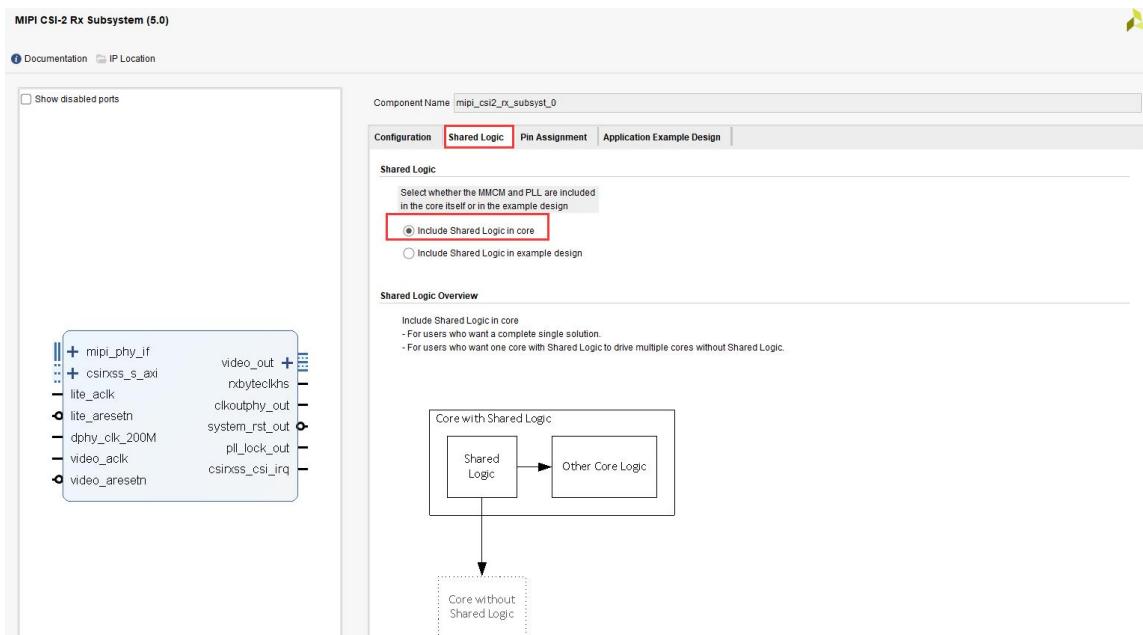
And connect the clock



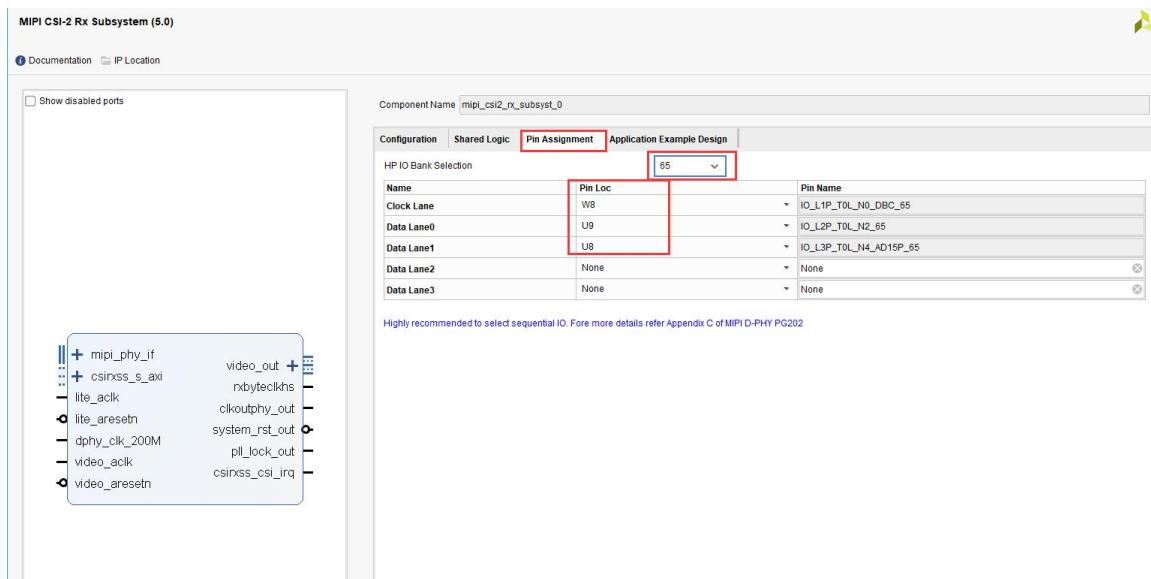
- 2) Add `mipi_csi2_rx_subsystem` module for receiving and parsing MIPI data and converting it to axi-stream interface. The configuration is as follows, the data format is RAW10, 2 lanes are selected, and the line rate is configured as 1000Mbps, which refers to the maximum supported rate. You can also fill in according to your needs, ranging from 80 to 2500; the default configuration of Pixels Per Clock is 1, which means 1 Each period is 1 pixel;



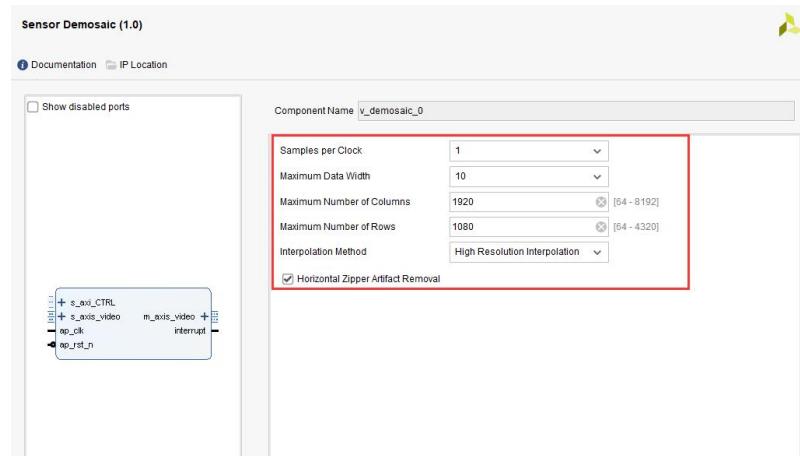
Shared Logic select "Include Shared Logic in core"



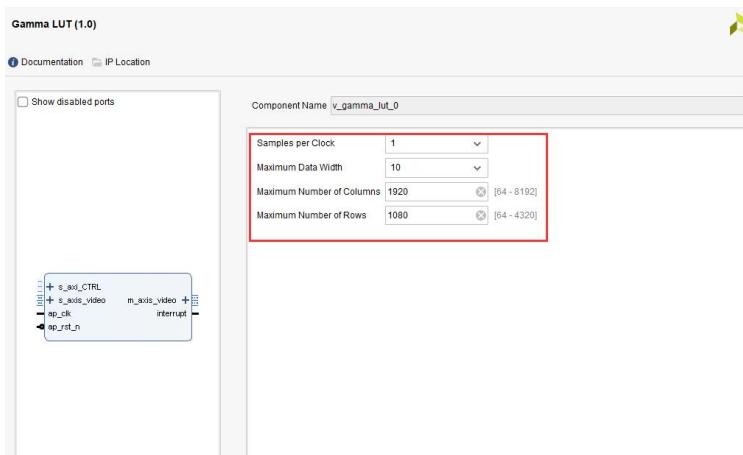
Pin Assignment is configured as follows according to the schematic



- 3) Add the Sensor Demosaic module to convert RAW10 to RGB, the configuration is as follows, the data bit width is set to 10, and the maximum resolution is 1920\*1080

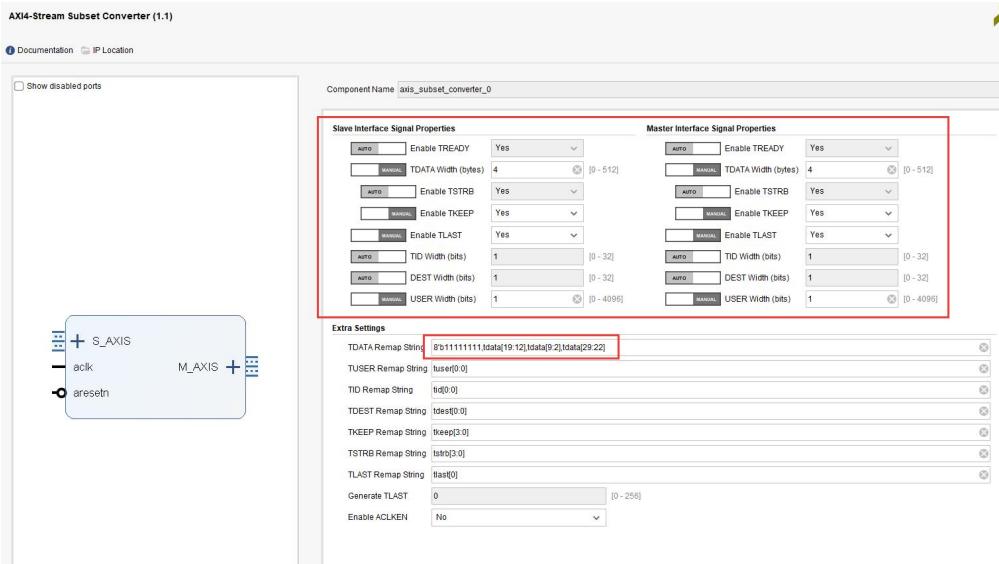


- 4) Add Gamma LUT module for Gamma correction

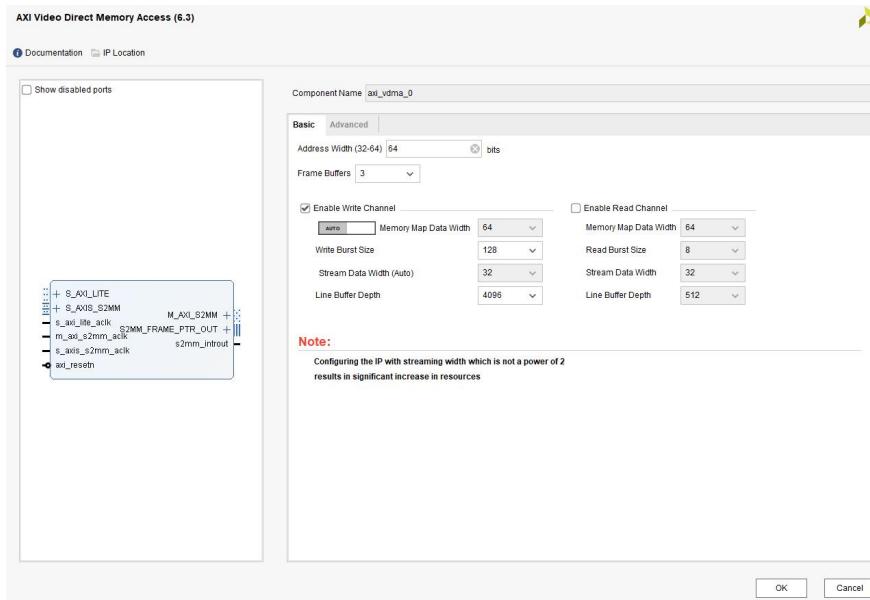


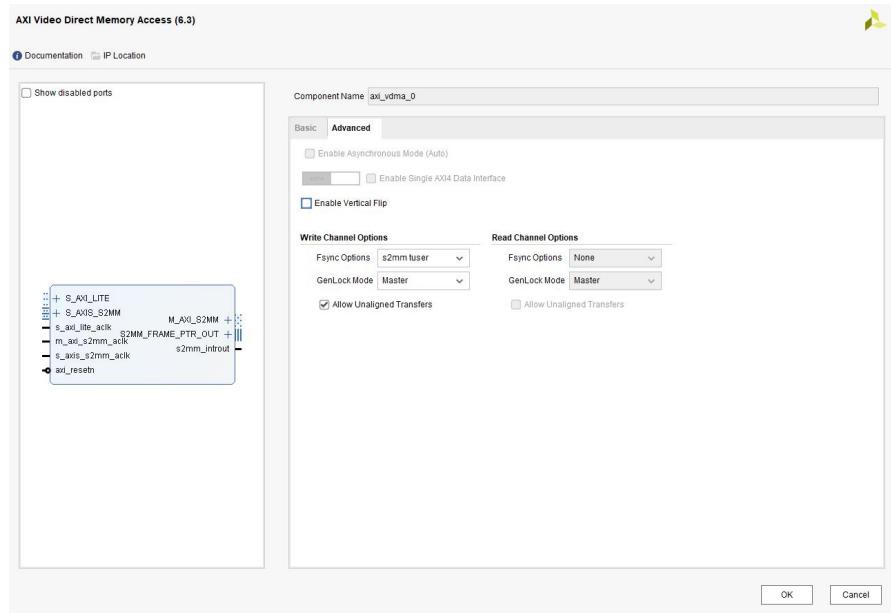
- 5) Add the subset module to adjust the order of the image data,

because after actual operation, it is found that the order of the RGB data of the image needs to be adjusted.

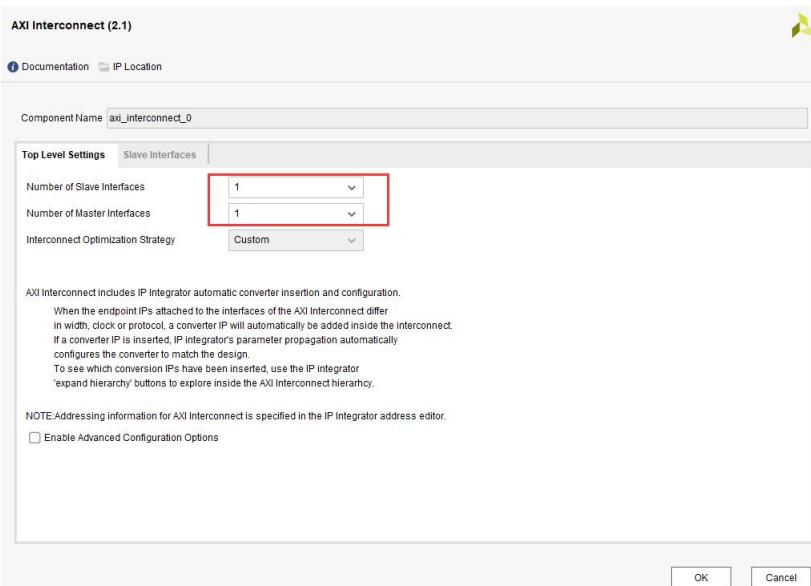


## 6) Add the VDMA configuration as follows

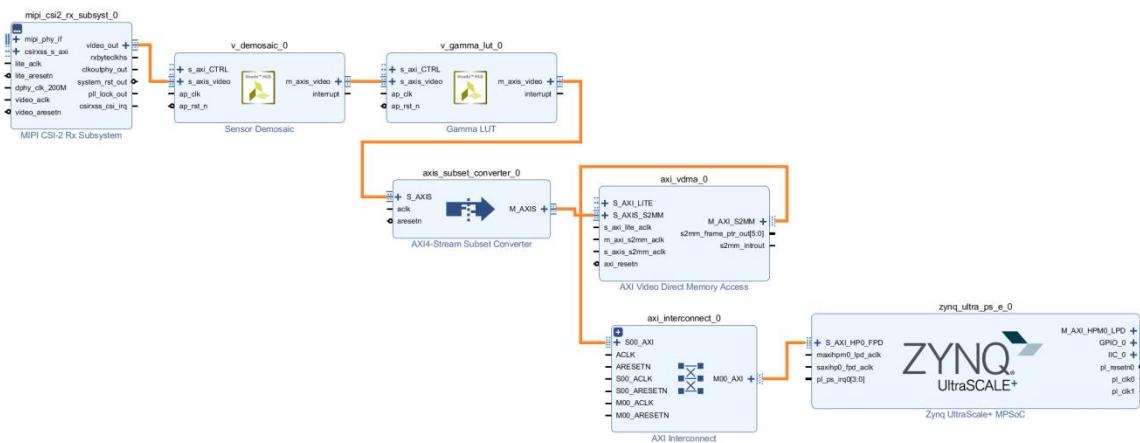




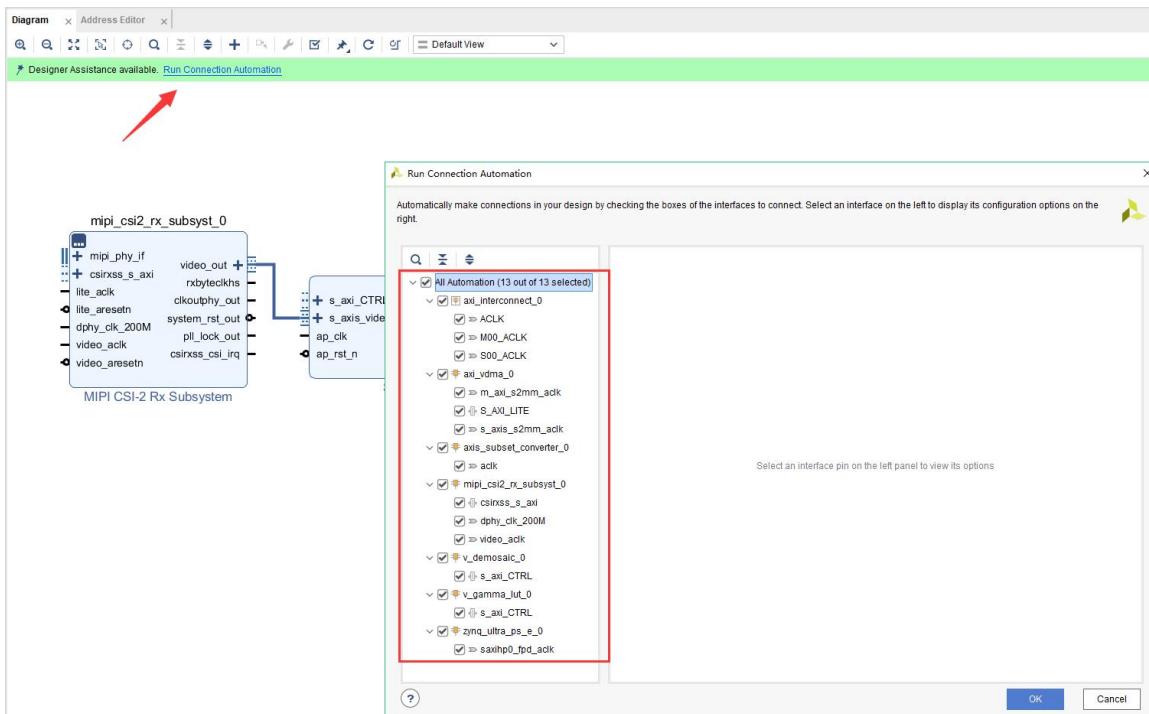
- 7) Add an AXI Interconnect module and configure it as 1 master and 1 slave



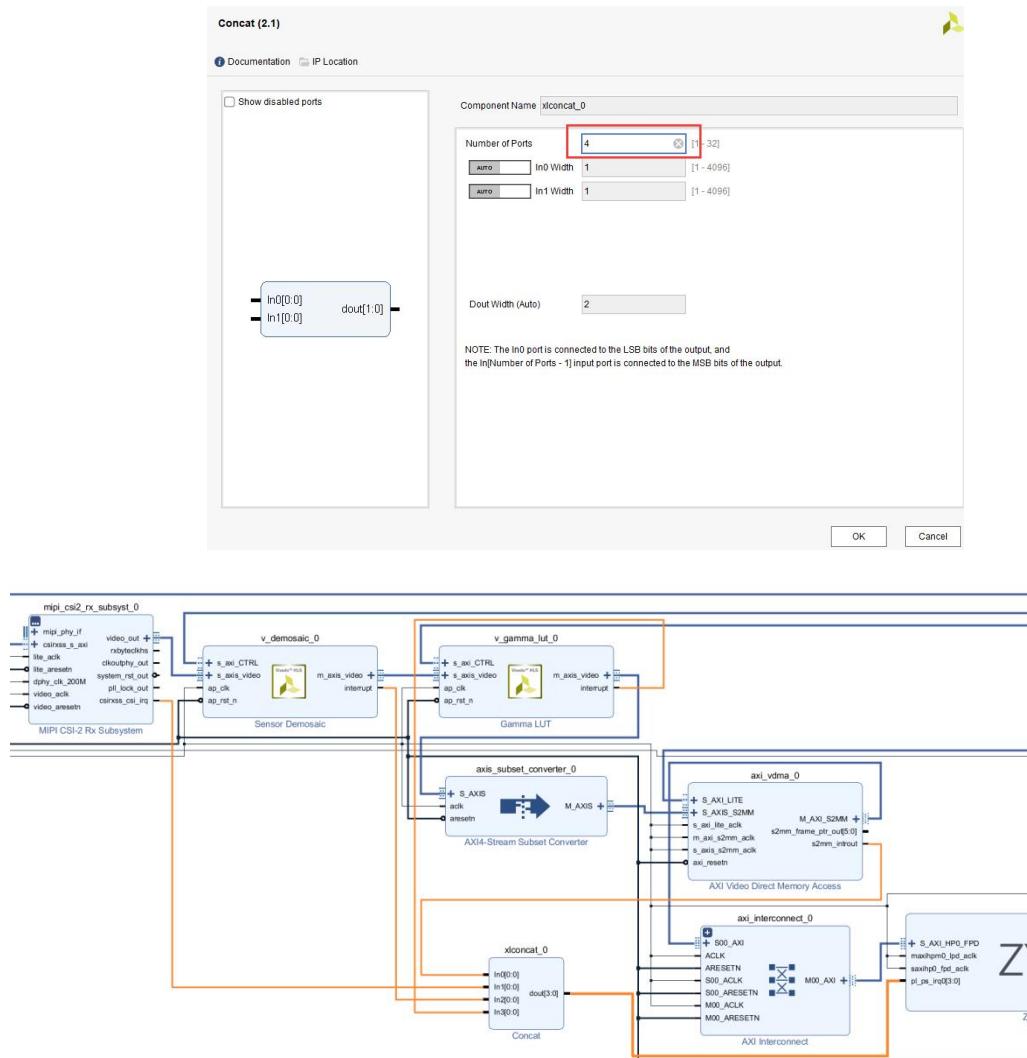
- 8) Connect key signals



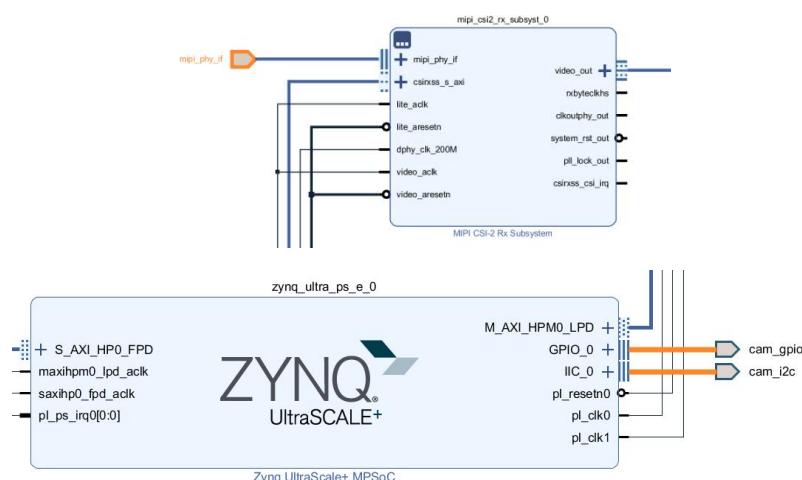
## 9) Automatic connection



## 10) Add concat module connection interrupted



## 11) Export the pin and modify the name



## 12) At this point, the hardware is built, bitstream is generated, and hardware information is exported.

## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 31.3: Vitis Program Development

The Vitis program is also relatively simple. On the basis of VDMA, add the initialization of the camera, the configuration of the VDMA, and the reset of the camera and the initialization of I2C are required.

```
PsGpioSetup() ;
/*
 * Reset sensor
 */
XGpioPs_WritePin(&Gpio, CAM_EMIO, 0) ;
usleep(1000000);
XGpioPs_WritePin(&Gpio, CAM_EMIO, 1) ;
usleep(1000000);

i2c_init(&ps_i2c0, XPAR_XIICPS_0_DEVICE_ID,1000000);
```

At the end, configure the MIPI camera and start the VDMA of the camera. The camera here is configured to 720p, 30fps

```
/*
 * DP dma demo
 */
xil_printf("DPDMA Generic Video Example Test \r\n");
Status = DpdmaVideoExample(&RunCfg, pFrames[0]);
if (Status != XST_SUCCESS) {
    xil_printf("DPDMA Video Example Test Failed\r\n");
    return XST_FAILURE;
}

gamma_lut_init();
demosaic_init();

/* Start Sensor Vdma */
vdma_write_init(XPAR_AXIVDMA_0_DEVICE_ID,HORSIZE,VERSIZE,DEMO_STRIDE,(unsigned int)pFrames[0]);

/*
 * Initialize Sensor
 */
sensor_init(&ps_i2c0);

return 0;
```

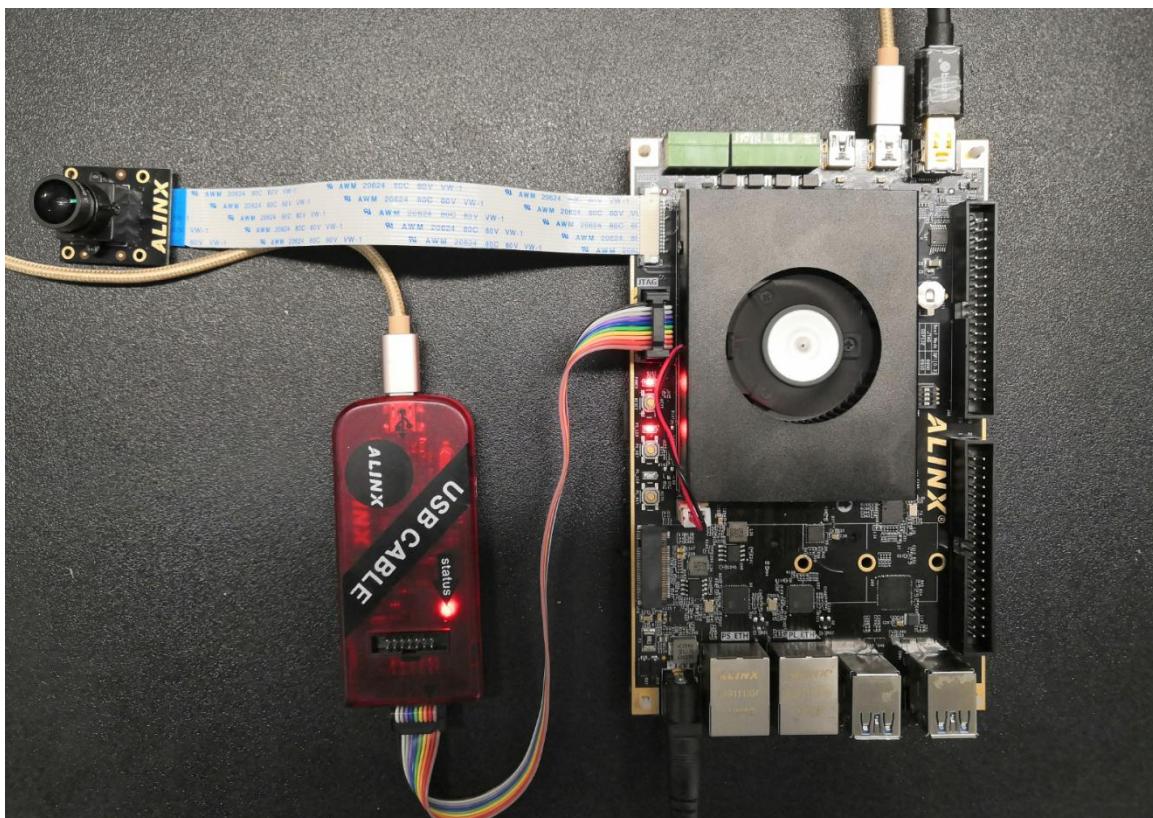
Currently the program supports two resolution configurations, 720p @60fps and 1080p@30fps. If you want to change to 1080p, you need to modify three places. One is in **ov5640.c**, comment out 720p and enable 1080p

The screenshot shows the Vitis IDE interface. On the left, the Explorer view displays a project structure under 'design\_1\_wrapper'. Inside 'mipi\_system' (under 'mipi'), there is a folder 'src' containing several source files like 'dp.c', 'i2c.c', and 'config.h'. A red box highlights 'config.h'. On the right, the code editor shows the content of 'config.h'. A specific line, '#define P1080 1', is highlighted with a red box.

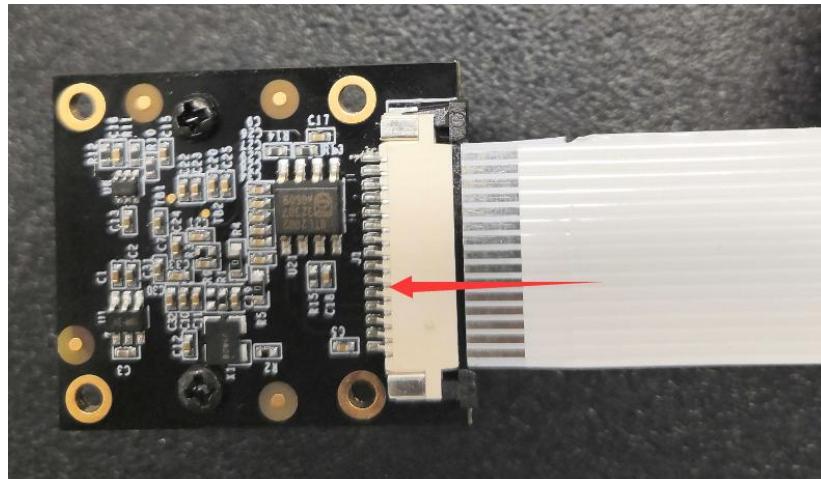
```
1 #ifndef SRC_CONFIG_H_
2 #define SRC_CONFIG_H_
3
4 #define P1080 1
5
6
7
8 #if P1080 == 1
9 #define VIDEO_COLUMNS 1920
10#define VIDEO_ROWS 1080
11#else
12#define VIDEO_COLUMNS 1280
13#define VIDEO_ROWS 720
14#endif
15
16
17#endif /* SRC_CONFIG_H_ */
```

#### Part 31.4: Onboard Verification

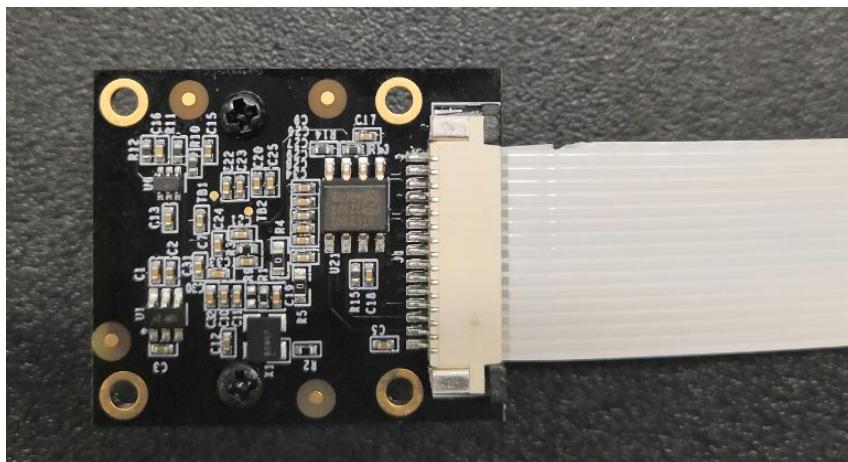
Connect the MIPI camera as shown in the figure below.



Hardware Connection

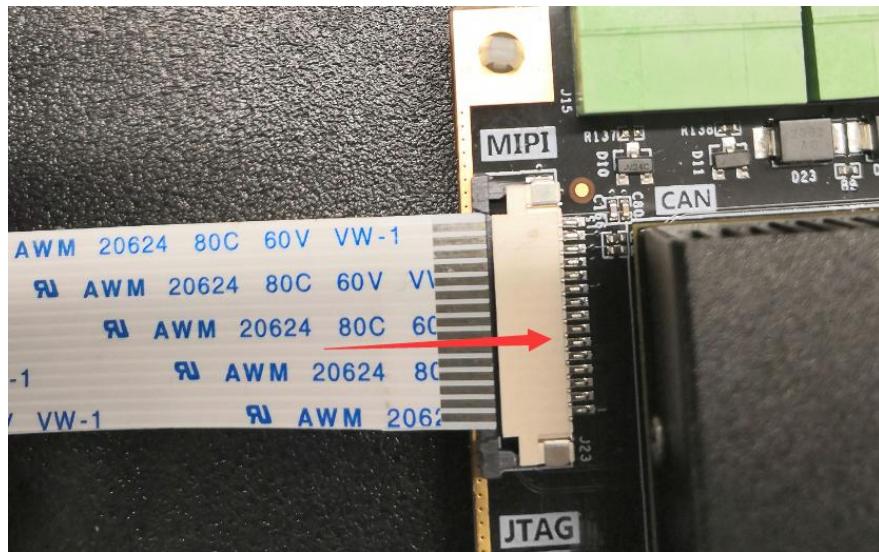


Camera Cable Connection (Not Connected)

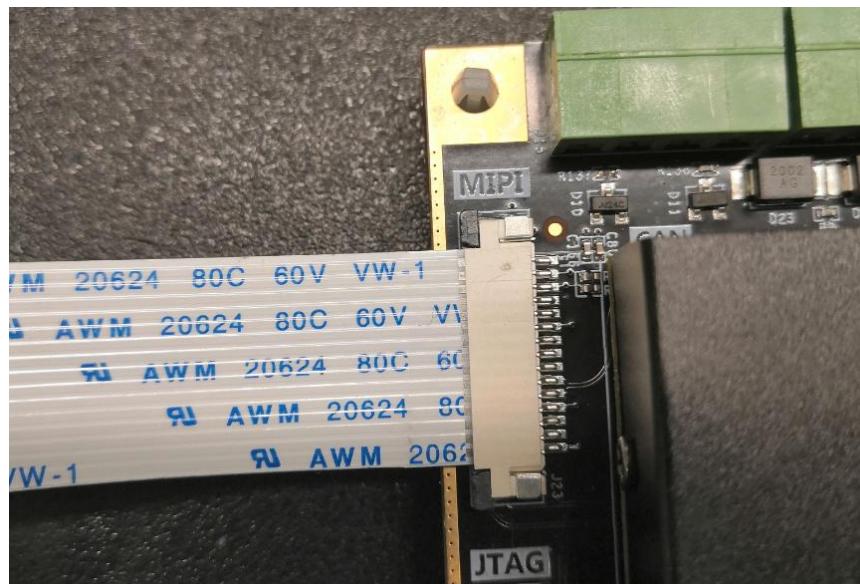


Camera Cable Connection (Connected)

**Pay attention to the direction of the cable, and be sure not to reverse it! ! !**



Board Cable Connection (Not Connected)



Board Cable Connection (Connected)

After downloading the program, the monitor will display the image.

## Part 32: Audio Module AN831 Recording and Playback

The experimental Vivado project directory is "audio\_i2s\_stream /vivado".

The experiment vitis project directory is "audio\_i2s\_stream /vitis".

Since there is no audio circuit on the FPGA development board, an external ALINX audio module AN831 is required. Based on this module, this experiment reads wav format music files from the SD card, and uses DMA to realize the music playback function.



There are three audio connectors on the audio module AN831, the pink interface is the microphone input; the green interface is the headphone output; the blue interface is the audio input, which is used to connect the audio output port such as DVD

### Part 32.1: Audio Module AN831 Module Description

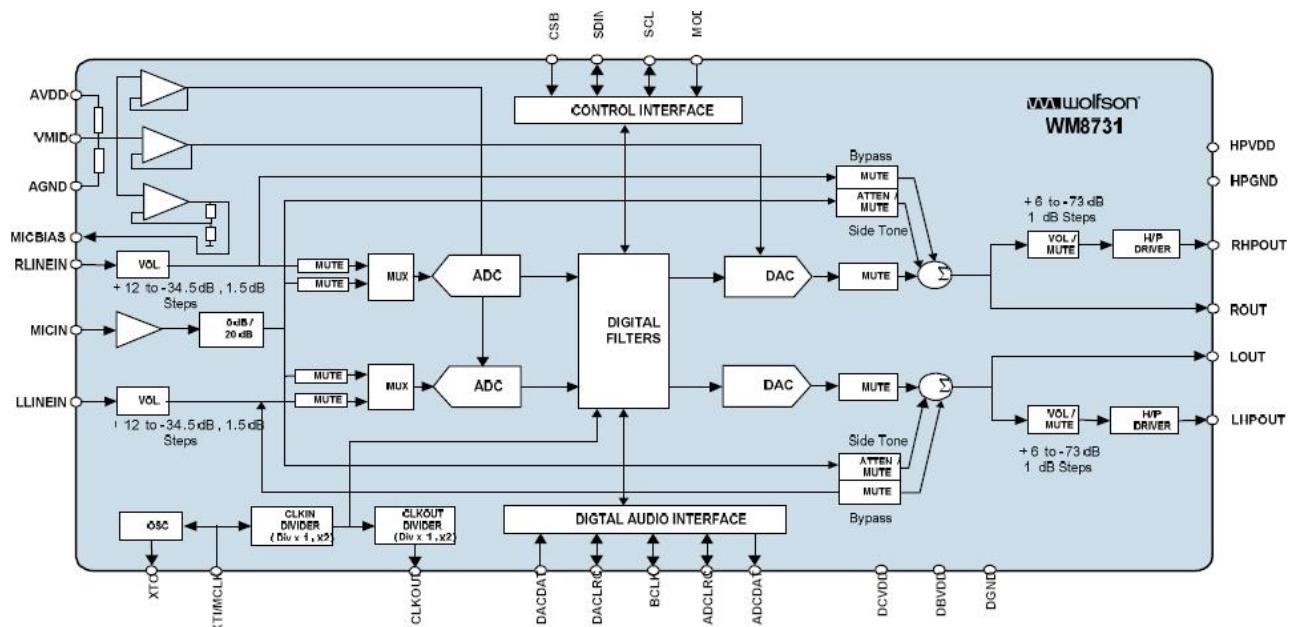
Here briefly introduce the audio encoding/decoding chip WM8731

used in the audio module AN831, which mainly perform A/D and D/A conversion function of the sound signal during recording and playing. The WM8731, stereo 24-bit multi-bit sigma delta ADCs and DACs are used with oversampling digital interpolation and decimation filters. Digital audio input word lengths from 16-32 bits and sampling rates from 8 kHz to 96 kHz are supported. There are 11 registers with 16 bits per register (7 bit address+9 bits of data).

In this design, WM8731 works in slave mode, the sampling frequency is set to 48KHZ, and the converted data bit length is 16 bits

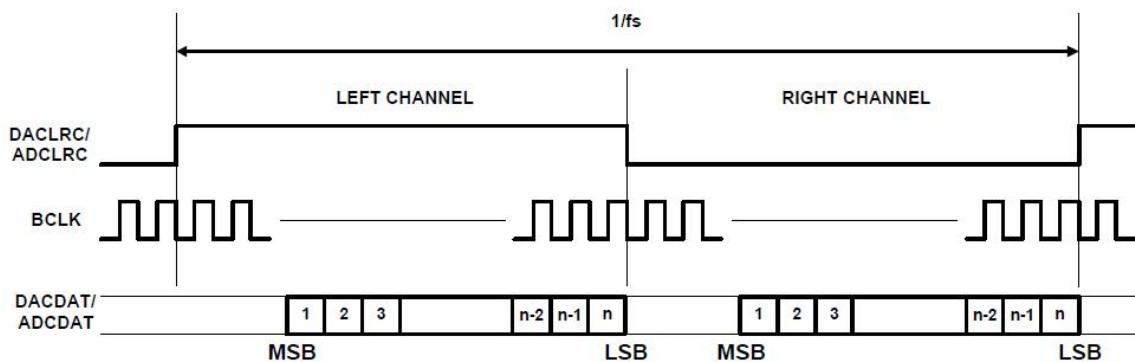
There are 5 digital audio interfaces on chip WM8731 : BCLK (Digital Audio Bit Clock); DACDAT (DAC Digital Audio Data Input); DACLRC (DAC Sample Rate Left/Right Clock); ADCDAT(ADC Digital Audio Data Onput); ADCLRC(ADC Sample Rate Left/Right Clock).

In this design, the Zynq FPGA is a slave device and the WM8731 is a master device. ADCDAT、DACDAT、ADCLRC, and DACLRC are synchronized with bit clock BCLK, data transfer on the falling edge of each BCLK. BCLK, DACDAT, DACLRC, ADCLRC are the input signal of WM8731. ADCDAT is the output signal of WM8731.

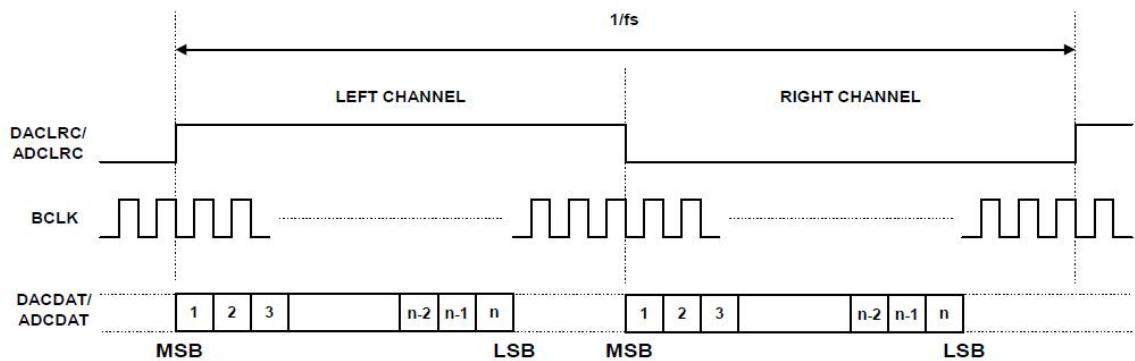


In this system, WM8731 control and data communication will use I2C and digital audio bus interface. Configure the registers of WM8731 through the I2C interface, and communicate audio data through the I2S bus interface. The digital audio interface can provide 4 modes:

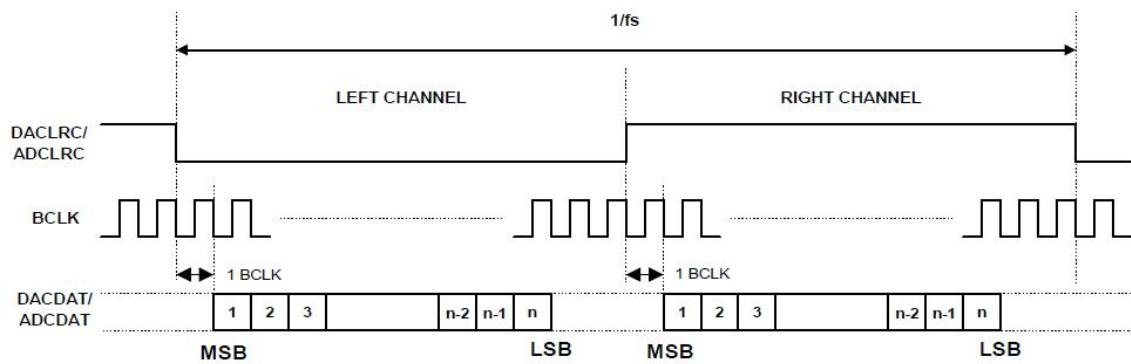
- Right Justified Mode
- Left Justified
- I2S Mode
- DSP Mode



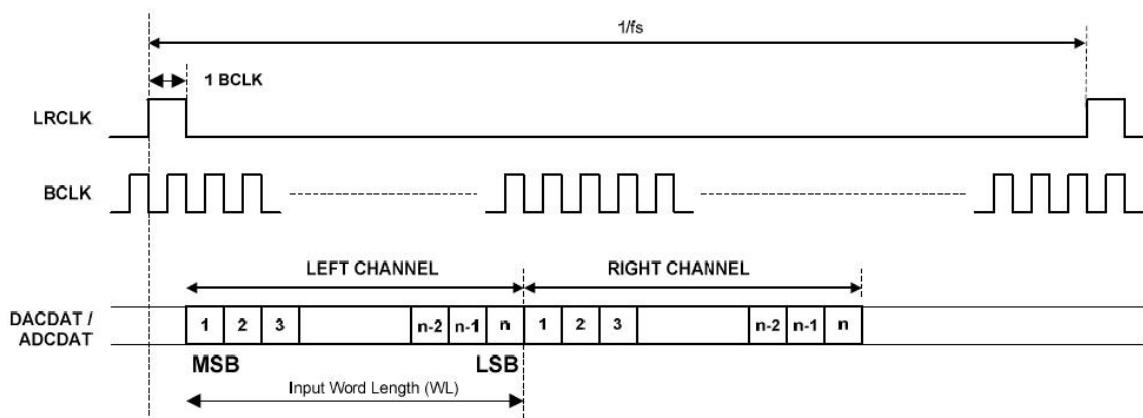
Right Justified Mode



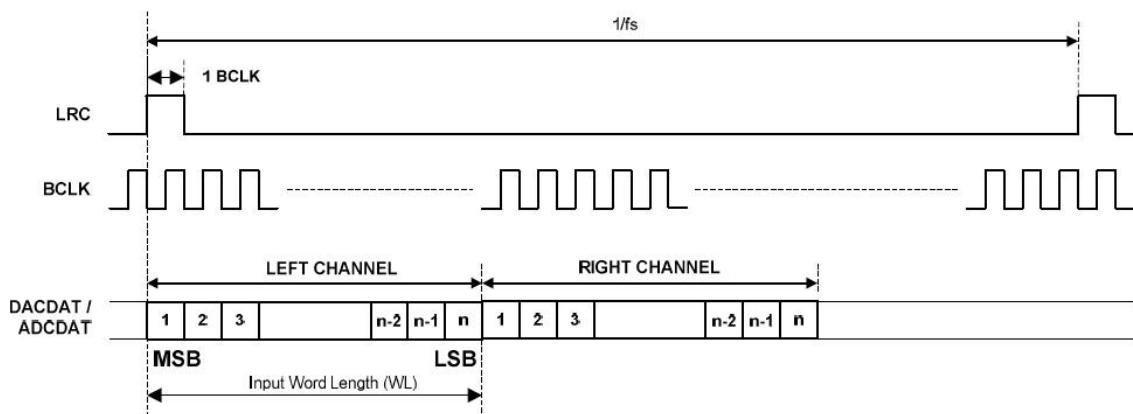
Left Justified



I2S Mode



DSP/PCM (Mode A)



DSP/PCM (Mode B)

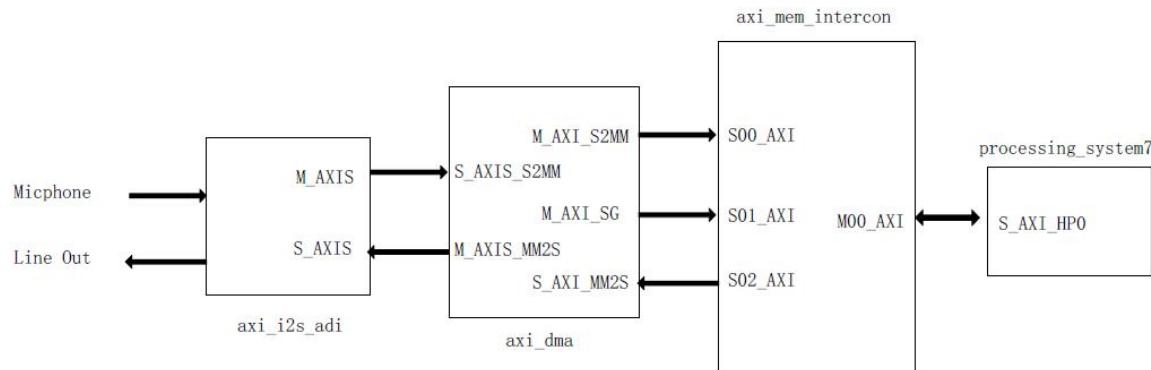
The I2S mode is selected for this experiment.

## FPGA Engineer Job Content

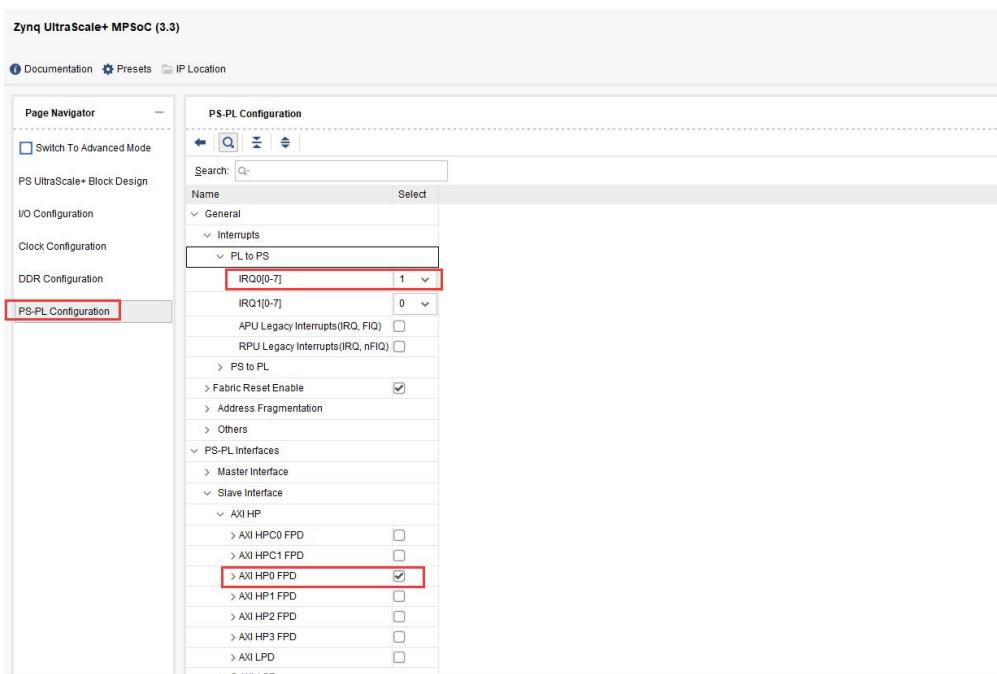
The following describes what the FPGA engineer is responsible for

## Part 32.2: Create Hardware Environment

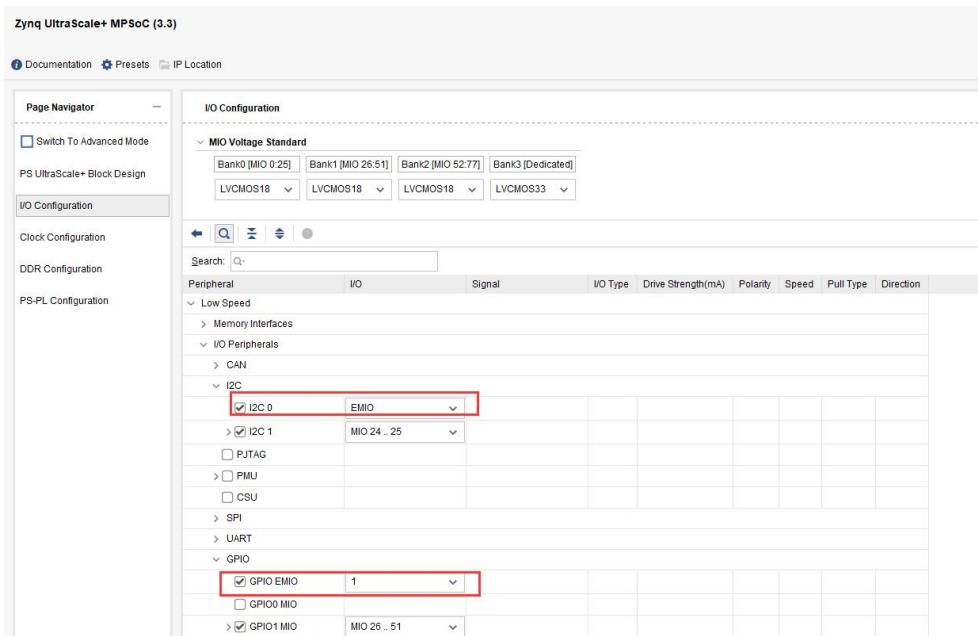
The overall block diagram is as follows, using the I2S IP core provided by ADI, connecting the microphone and audio output, and using a DMA to process stream data.



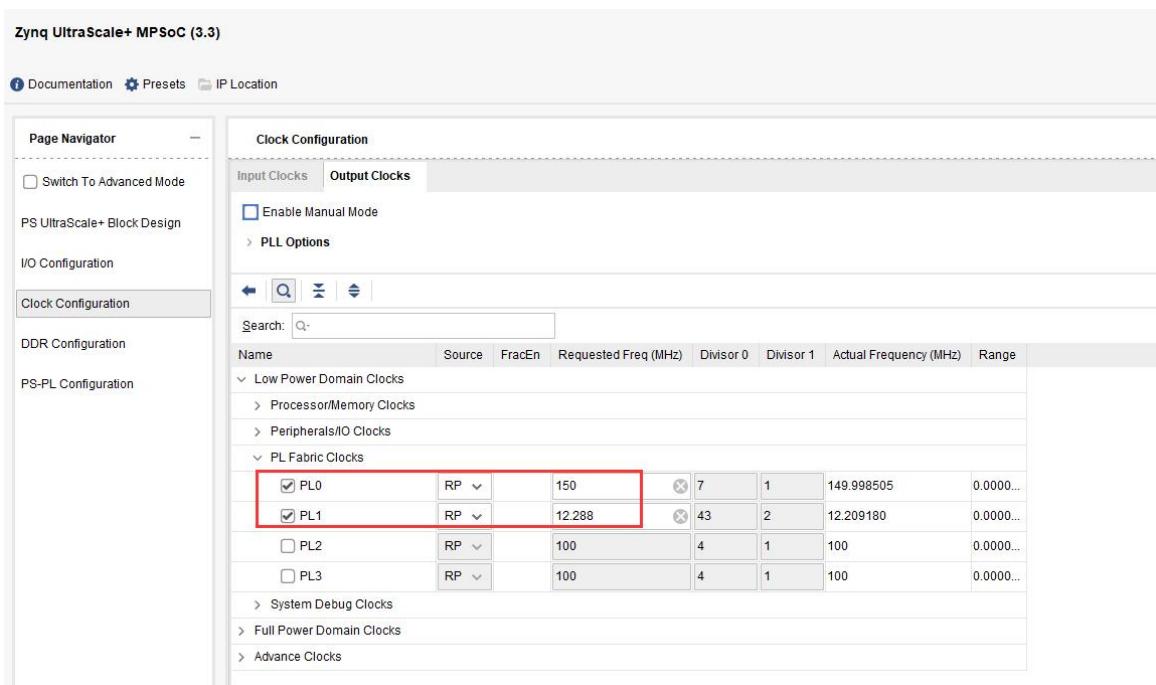
- 1) Based on the "ps\_hello" project. In the experiment, open the HPO port and interrupt



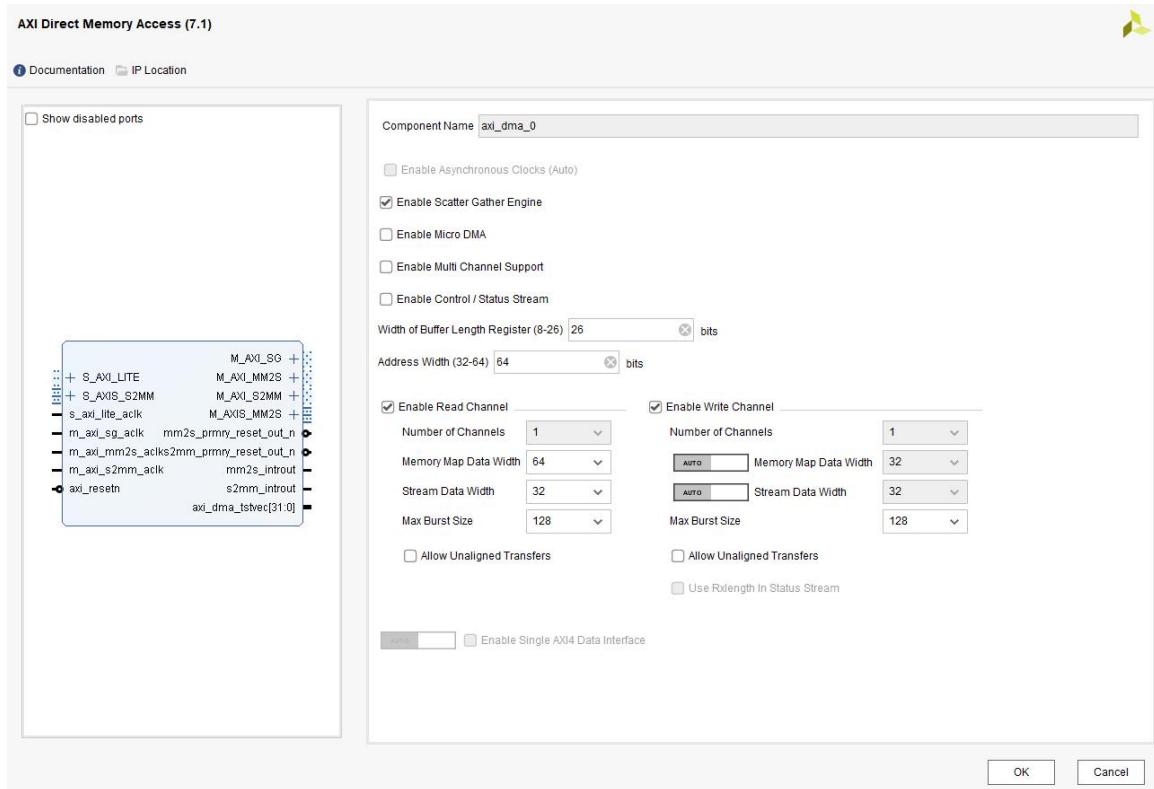
- 2) Enable I2C0 and select EMIO, used to configure WM8731, turn on GPIO EMIO, used to connect LED lights



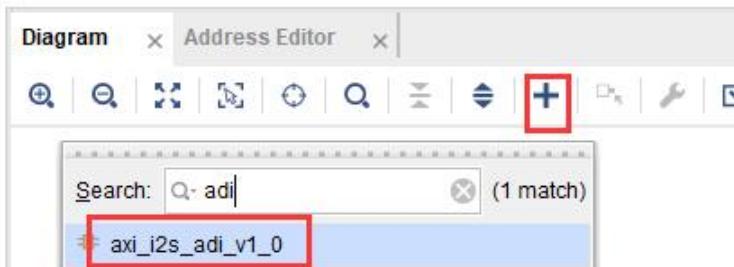
- 3) Configure FCLK\_CLK1 to 12.288MHz, which is the main clock provided to I2S, and the I2S module can divide BCLK and LRCLK according to this clock



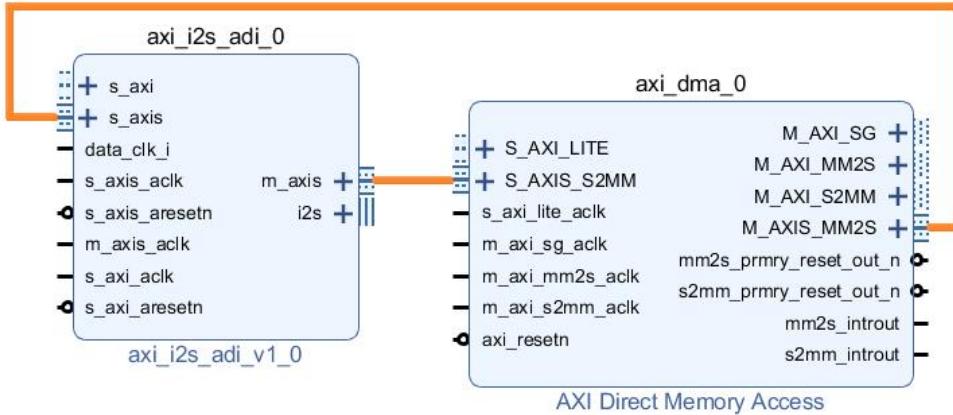
- 4) Configure DMA as follows, open the read and write ports, select SG mode, because the audio data is 32 bits, set the stream data width to 32



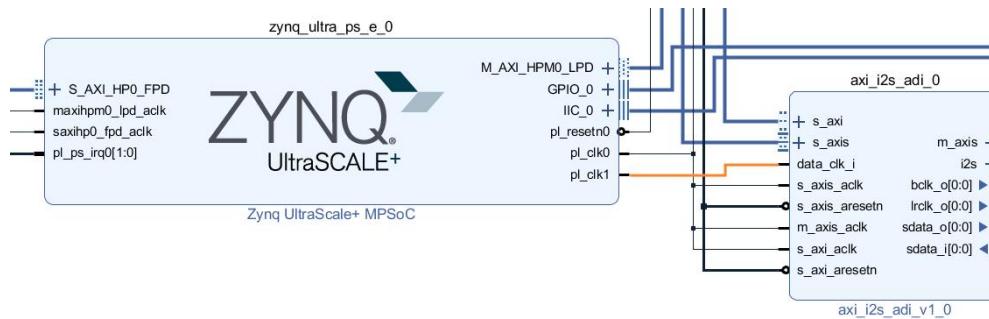
- 5) Add ADI's I2S IP, this ip is in the repo folder of the project directory, how to add it to vivado? I have already mentioned it and will not repeat it, hereafter referred to as I2S module.



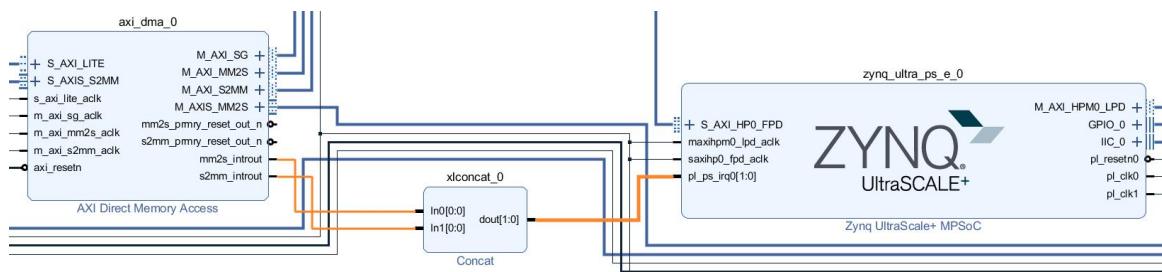
- 6) Connect M\_AXIS\_MM2S of axi dma to s\_axis of i2s. This is the data stream interface for audio playback, that is, DMA transfers the audio data in the DDR memory to the I2S module; Connect S\_AXIS\_S2MM of axi dma to m\_axis of i2s, which is the recording data stream interface, and transfer the collected microphone data to DDR memory through DMA.



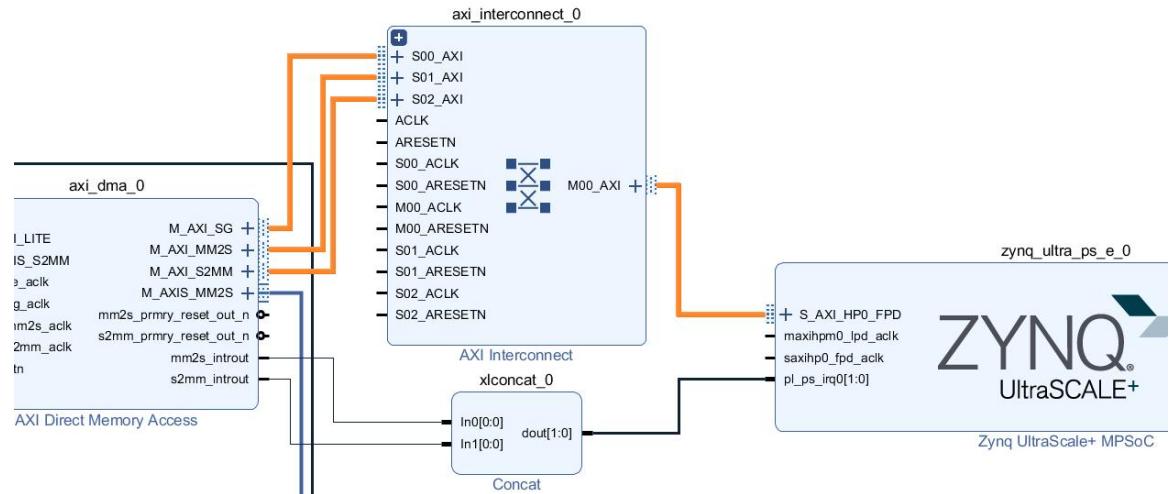
7) Connect **pl\_clk1** to **data\_clk\_i**, the frequency is 12.288MHz



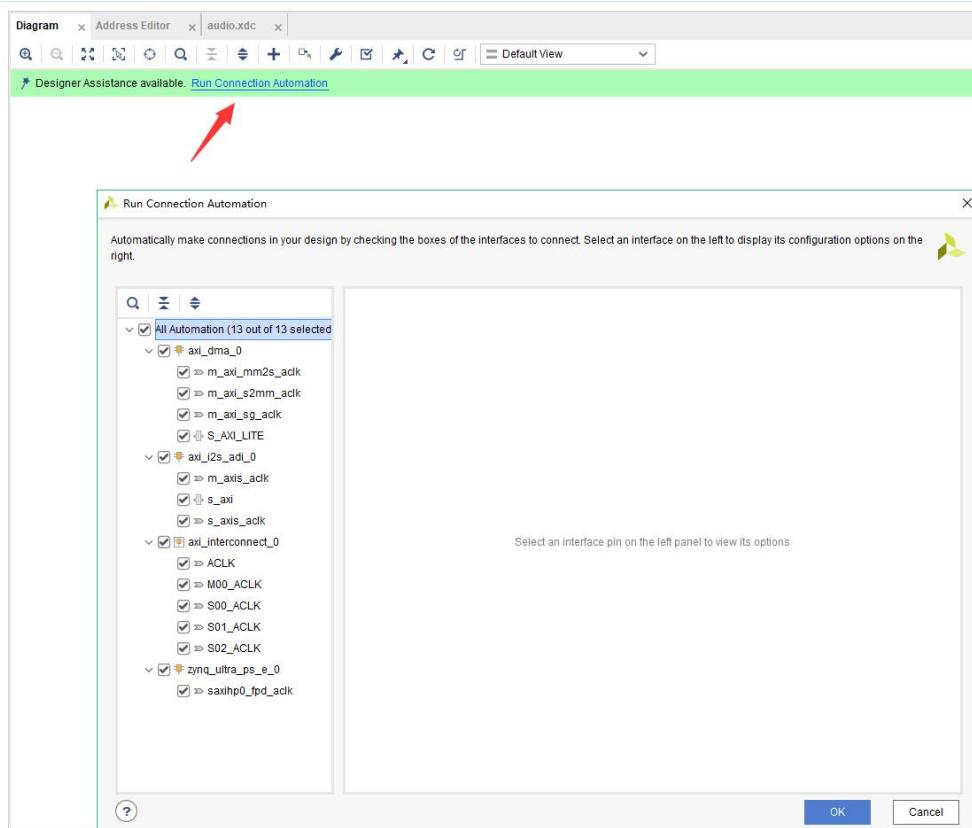
8) Add the concat module, connect the two interrupts of dma to the ZYNQ interrupt port



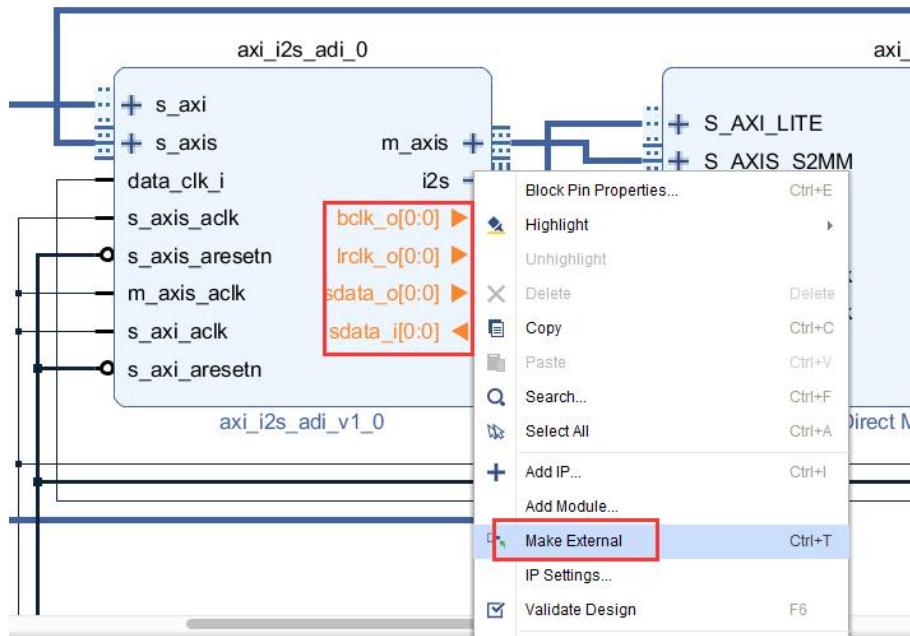
9) Add axi interconnect module to connect the HP port of dma and zynq



## 10) Auto connect



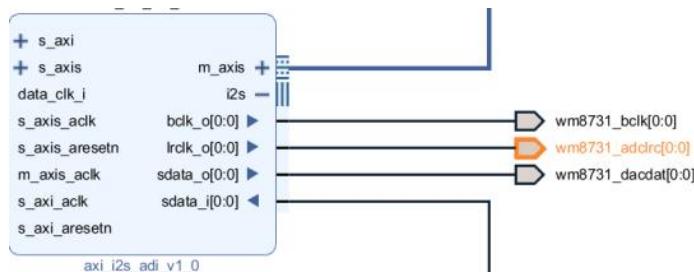
## 11) Export I2S interface



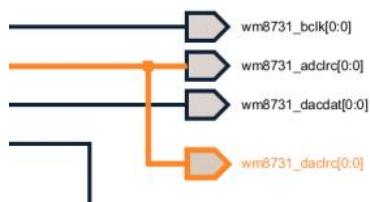
## 12) Modify pin name



## 13) Select the pin of Irclk\_o, "ctrl+c,ctrl+v" to copy a pin



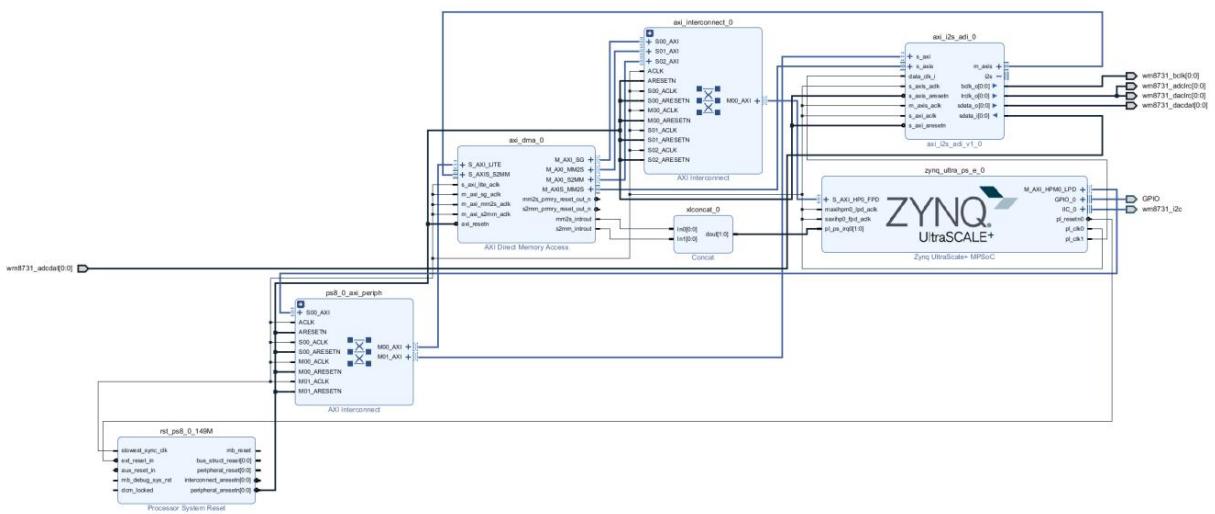
Modify the name and connect with it, this is the clock of the left and right channels of dac



## 14) Export GPIO and I2C, and modify the name



15)The final connection result is as follows



16)Bind pins and generate bitstream, then export hardware information

## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 32.3: Vitis Program Development

#### Part 32.3.1: SD Card Playing Music Experiment

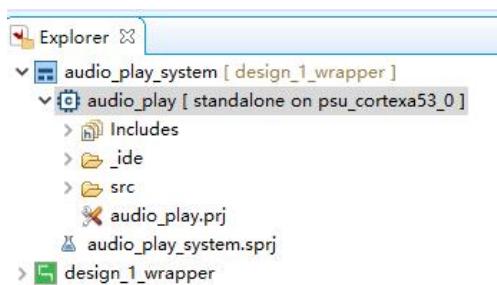
After the hardware is created, how to play music? The reading and writing of SD card and the use of SG DMA have been introduced before. We can read and write audio files in the SD card to DDR, and then transfer the audio data to the audio module for playback through

DMA.

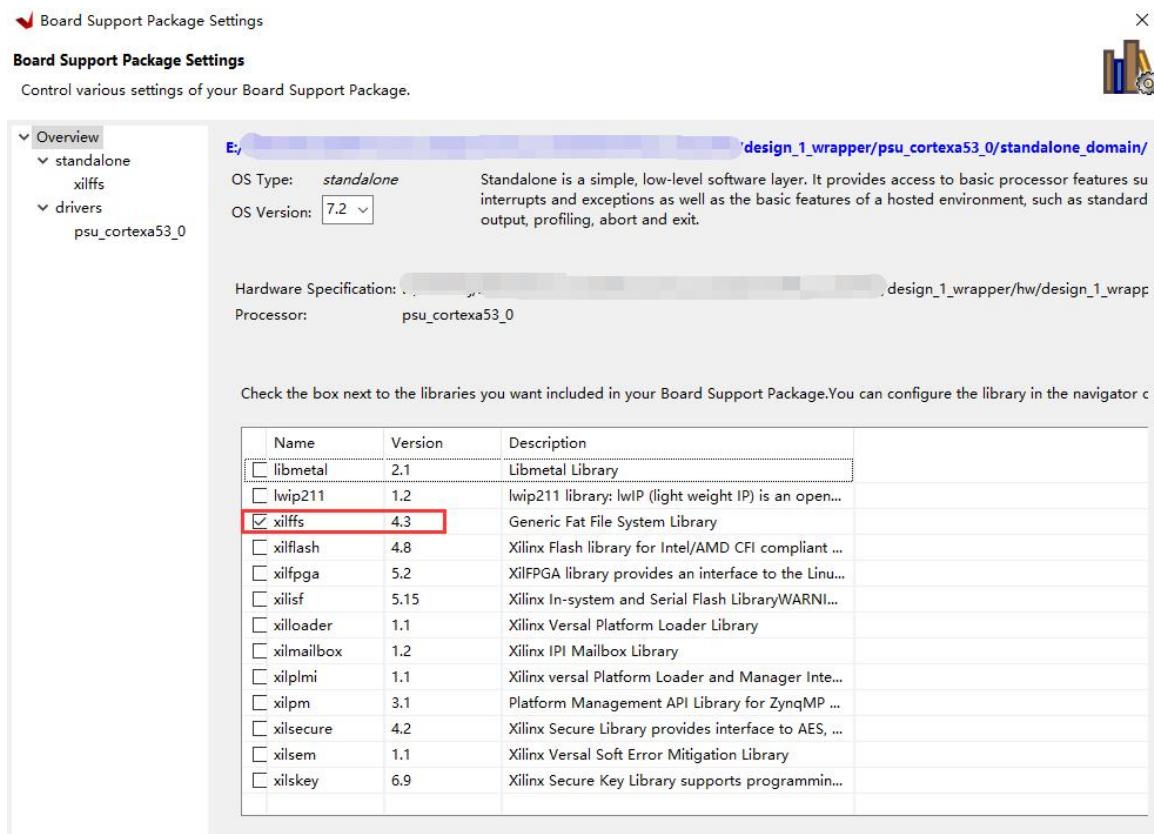
### The following is the flow of the program:

Initialize I2C and I2S modules → Close the transmit and receive channels of the I2S module → Initialize the interrupt controller → Initialize GPIO → Read wav audio files in SD card → Initialize DMA and open MM2S interrupt → Start SG DMA transfer → Enable the transmitting channel of the I2S module

#### 1) Create a new “audio\_play” project



#### 2) Enable the **xilffs** library in the **bsp** setting



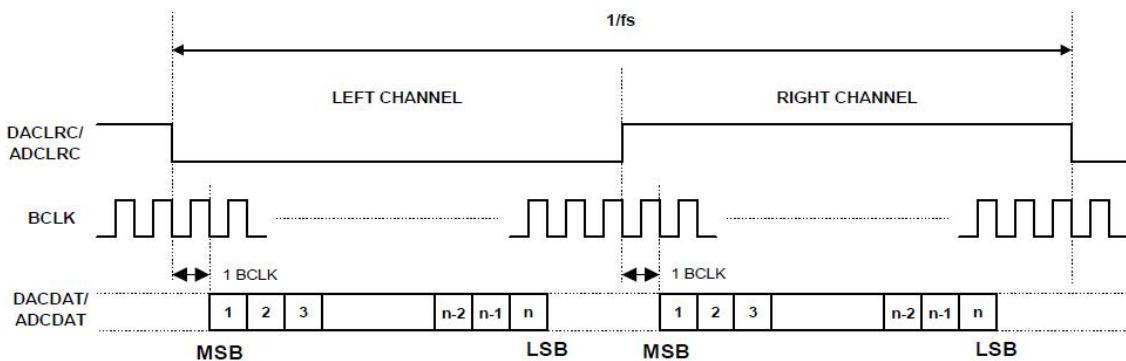
- 3) In the I2S module initialization function, the audio\_reg\_init function performs the register configuration of wm8731 and configures it as slave mode with a 16bit sampling rate of 48KHz. And configure the clock register of the I2S module, and configure it to 48KHz, which is the frequency of LRCLK.

```
int audio_init(Audio *AudioPtr, XIicPs *IicInstance)
{
    AudioPtr->BaseAddr = XPAR_AXI_I2S_ADI_0_BASEADDR ;
    AudioPtr->IicDevAddr = WM8731_DEVADDR ;
    /* Initial audio registers */
    audio_reg_init(IicInstance, AudioPtr, wm8731_init_data);
    /* MCLK:12.288MHz. 256fs. setting to 48Khz */
    Audio_WriteReg(AudioPtr->BaseAddr, AUDIO_REG_I2S_CLK_CTRL, (64/2 - 1)<<16 | (4/2 - 1)) ;/* LRCLK = BCLK / 64 and BCLK = MCLK / 4. */
    return XST_SUCCESS ;
}
```

How to correctly set the bit clock BCLK and the left and right channel clock LRCLK? As mentioned earlier, FCLK\_CLK1 is set to 12.288MHz, which is the main clock provided to the I2S module. Both BCLK and LRCLK are frequency-divided based on it. In order to get the sampling frequency of 48KHz, choose BOSR as 256fs, that is, MCLK frequency=256\*LRCLK.

SAMPLING RATE		MCLK FREQUENCY	SAMPLE RATE REGISTER SETTINGS				DIGITAL FILTER TYPE
ADC	DAC		BOSR	SR3	SR2	SR1	
kHz	kHz	MHz					
48	48	12.288	0 (256fs)	0	0	0	0
		18.432	1 (384fs)	0	0	0	0
48	8	12.288	0 (256fs)	0	0	0	1
		18.432	1 (384fs)	0	0	0	1
8	48	12.288	0 (256fs)	0	0	1	0
		18.432	1 (384fs)	0	0	1	0

One channel has 32 BCLKs, and two are 64 BCLKs, so the BCLK frequency = 64 LRCLK frequency, then the MCLK frequency 4\*BCLK can be obtained. Configure the I2S module according to this proportional relationship.



- 4) Before transmitting data, the TX and RX channels of the I2S module should be closed, otherwise it will cause abnormal DMA transmission.

```
/*
 * disable tx and rx channel before transfer
 */
audio_txrx_disable(&AudioInst, TX_ENABLE_MASK|RX_ENABLE_MASK) ;
```

- 5) In the function of SD card reading data, judge whether it is a wav file and record the length.

```
if(TMPBUF[0] == 'R' && TMPBUF[1] == 'I' && TMPBUF[2] == 'F' && TMPBUF[3] == 'F'
    && TMPBUF[8] == 'W' && TMPBUF[9] == 'A' && TMPBUF[10] == 'V' && TMPBUF[11] == 'E'){
    FrameLength = ((unsigned int)TMPBUF[7]<<24) + ((unsigned int)TMPBUF[6]<<16) + ((unsigned int)TMPBUF[5]<<8) + TMPBUF[4] ;
    xil_printf("wave length is %x\r\n", FrameLength) ;
}
```

As one of the sound wave file formats used in multimedia, WAV files are based on the RIFF format. RIFF is the abbreviation of Resource Interchange File Format in English. The first four bytes of each WAV file are "RIFF", so this experiment judges whether the file is a WAV file based on whether the first 4 bytes are "RIFF", and then 4 bytes represent the size of the file, so we can determine the amount of data to be read. The WAV file header size is 88 bytes, and the first 88 bytes of the file header should be removed during playback.

OFFSET	LENGTH	VALUE	DESCRIPTION
0	4 bytes	'RIFF'	The file format ID.
4	4 bytes		Length of the file minus (-) 8 bytes.
8	4 bytes	'WAVE'	The data format ID.
12	4 bytes	'fmt '	The chunk ID.
16	4 bytes	32	Length of the chunk excluding the 8 bytes for the ID and length.
20	4 bytes		The codec ID.
24	4 bytes		The number of channels.
28	8 bytes		Samples per second.
36	8 bytes		Average bytes per second.
44	4 bytes		Block alignment.
48	4 bytes		Bits per sample.
52	4 bytes	'data'	The chunk ID.
56	4 bytes		Length of the data (chunk size minus (-) 8 bytes.
60	4 bytes	'fact'	The chunk ID.
	4 bytes	8	Chunk size minus (-) 8 bytes.
	8 bytes		Sample length.

- 6) Bd\_start starts DMA, and then turns on the TX channel of I2S, then the audio data stream can be transmitted to the AN831 module

```
int XAxiDma_Audio_Play(XScuGic *InstancePtr, Audio *AudioInstance, u32 length, u32 *Databuf)
{
    XAxiDma_Initial(DMA_DEV_ID, MM2S_INTR_ID, &AxiDma, InstancePtr) ;

    /* Create BD chain */
    CreateBdChain(BdTxChainBuffer, BD_COUNT, length * sizeof(u32) , (u8 *)Databuf, TXPATH) ;

    XGpioPs_WritePin(&Gpio, PLAY_LED, LED_ON);
    /* Start DMA transfer */
    Bd_Start(BdTxChainBuffer, BD_COUNT, &AxiDma, TXPATH) ;
    /* Enable TX channel */
    audio_txrx_enable(AudioInstance, TX_ENABLE_MASK) ;
```

### Part 32.3.2: On Board Verification

- 1) Format the SD card according to FAT32 format

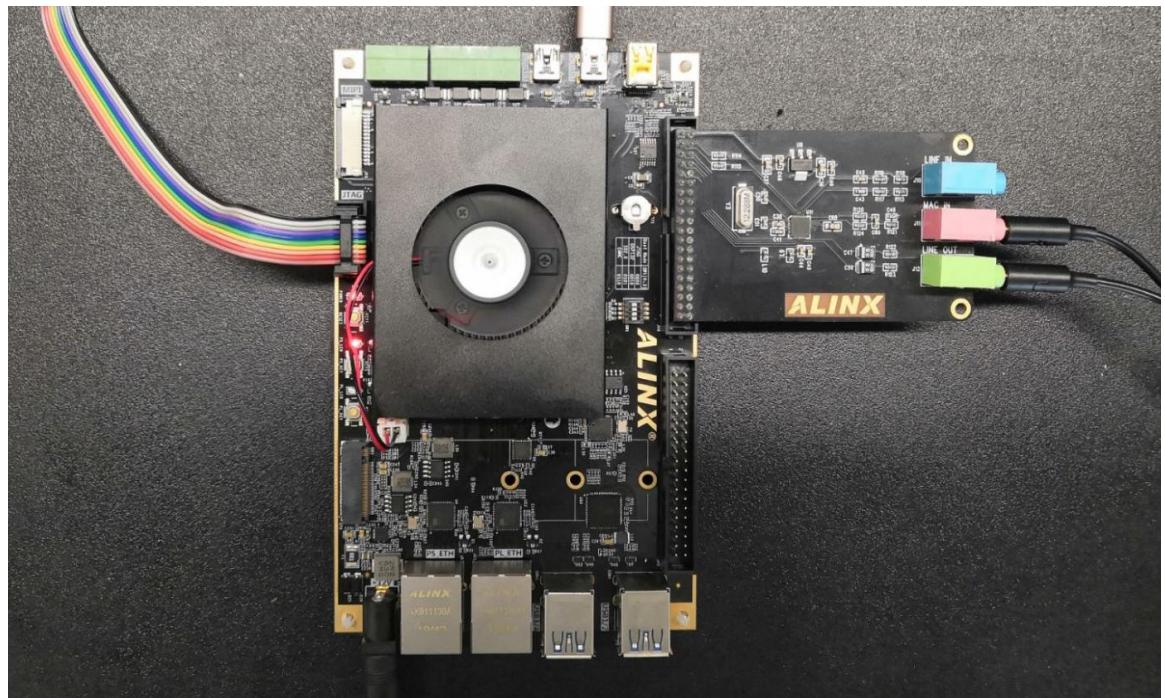


Copy 1.wav to the SD card, and insert the SD card into the SD card slot of the FPGA development board

BOOT (E):

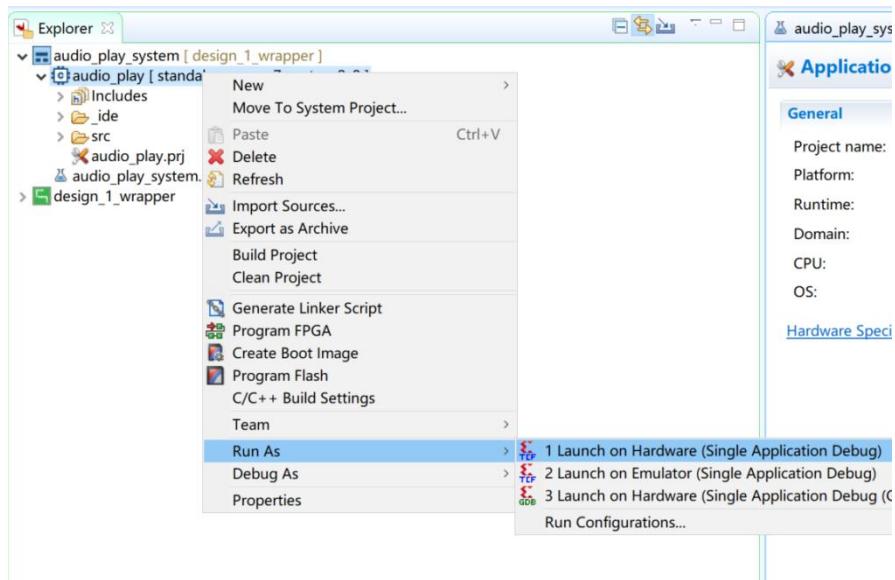
名称	修改日期	类型	大小
1.wav	2017/1/14 21:50	WAV 文件	52,212 KB

- 2) Connect the AN831 audio module with the FPGA development board, insert the headset and microphone



Hardware Connection (Expansion Port J46)

### 3) Download the program



### 4) Wait for a while, the PS\_LED light will light up and start to play music, and the PS\_LED will go out when the music is finished.

#### Part 32.3.3: Recording and Playback Experiment

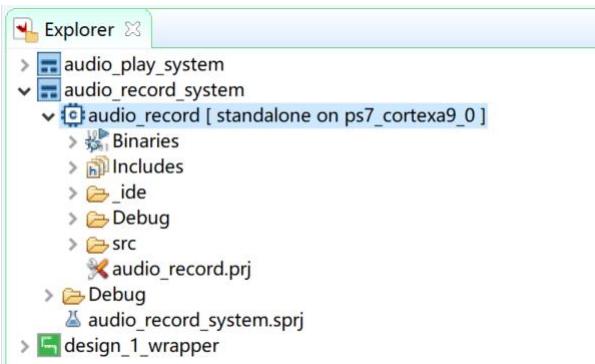
The previous experiment was about playing music experiments, and only used the MM2S channel of DMA, which is the channel for

transmitting data to peripherals. In this experiment, both channels of DMA are used to realize the functions of recording and playback. The final phenomenon is to press the key to start recording. The recording duration is fixed at about 10 seconds. After that, press the key again to play the recording.

### The following is the flow of the program:

Initialize I2C and I2S modules → Close the transmit and receive channels of the I2S module → Initialize the interrupt controller → Initialize GPIO → Read wav audio files in SD card → Initialize DMA, open MM2S and S2MM interrupt → Wait for the Key to start the S2MM channel SG DMA transmission and enable the I2S module to receive the channel, start recording → After recording, wait for the key to start MM2S channel SG DMA transmission and enable I2S module to transmit channel, start playing → Loop recording and playback operations

#### 1) Create a new “audio\_record” project



#### 2) First, the length of RECORD is defined, which is about 10 seconds

```
#define PLAY_LEN 0x4000000
#define RECORD_LEN 960000 //record length about 10 second for 48K sample
```

#### 3) Also before transmission, close the TX and RX channels of the I2S module, and send the length of the data to be received

```
audio_txrx_disable(&AudioInst, TX_ENABLE_MASK|RX_ENABLE_MASK) ;
/* Rx Stream data length */
Audio_RxStreamLengthSetting(&AudioInst, RECORD_LEN) ;
```

- 4) The difference from the previous experiment is that key initialization and interruption are added

```

Status = XScuGic_Connect(InstancePtr, KEY_INTR_ID,
    (Xil_ExceptionHandler)GpioHandler,
    (void *)Gpioinstance) ;
if (Status != XST_SUCCESS)
    return XST_FAILURE ;

/*
 * Enable the interrupt for the device.
 */
XScuGic_Enable(InstancePtr, KEY_INTR_ID) ;
XGpioPs_SetDirectionPin(Gpioinstance, KEY, 0) ;
XGpioPs_SetIntrTypePin(Gpioinstance, KEY, XGPIOPS_IRQ_TYPE_EDGE_RISING) ;

XGpioPs_IntrEnablePin(Gpioinstance, KEY) ;

```

And the key interrupt service function

```

int GpioHandler(void *CallbackRef)
{
    XGpioPs *GpioInstancePtr = (XGpioPs *)CallbackRef ;
    int value ;
    float Interval_time ;

    value = XGpioPs_IntrGetStatusPin(GpioInstancePtr, KEY) ;
    /*

```

- 5) Initialize DMA, open S2MM and MM2S interrupts, and create BD chain of TX and RX

```

int XAxiDma_Audio_Record_Play(XScuGic *InstancePtr, Audio *AudioInstance, u32 length, u32 *Databuf)
{
    /* Initial DMA, enable s2mm and mm2s interrupt */
    XAxiDma_Initial(DMA_DEV_ID, S2MM_INTR_ID, MM2S_INTR_ID, &AxiDma, InstancePtr) ;

    /* Create BD chain */
    CreateBdChain(BdTxChainBuffer, BD_COUNT, length * sizeof(u32) , (u8 *)Databuf, TXPATH) ;

    /* Create BD chain */
    CreateBdChain(BdRxChainBuffer, BD_COUNT, length * sizeof(u32) , (u8 *)Databuf, RXPATH) ;

```

- 6) Press the key to start DMA transfer and enable the corresponding channel of the I2S module

```

if (key_flag == 1){

    key_flag = 2 ;
    key_hold = 1 ;

    XGpioPs_WritePin(&Gpio, RECORD_LED, LED_ON);
    /* Start DMA transfer */
    Bd_Start(BdRxChainBuffer, BD_COUNT, &AxiDma, RXPATH) ;
    /* Enable RX channel */
    audio_txrx_enable(AudioInstance, RX_ENABLE_MASK) ;
}

```

- 7) When the DMA transfer is over, close the corresponding I2S module channel and clear the BD status.

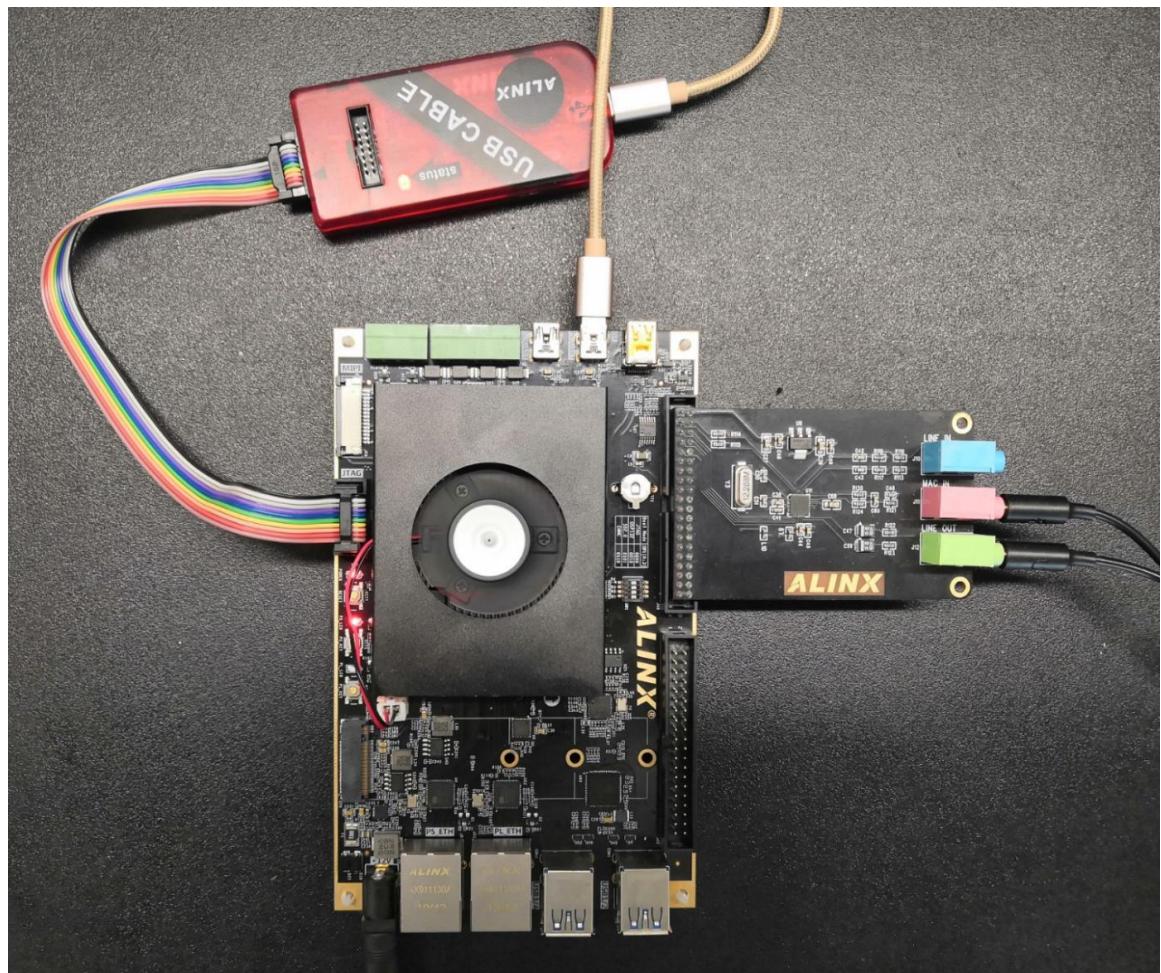
```
if (s2mm_flag)
{
    /* clear s2mm_flag */
    s2mm_flag = 0 ;
    /* disable RX channel */
    audio_txrx_disable(AudioInstance, RX_ENABLE_MASK) ;

    XGpioPs_WritePin(&Gpio, RECORD_LED, LED_OFF);
    /* Clear BD Status */
    Bd_StatusClr(BdRxChainBuffer, BD_COUNT) ;

    key_hold = 0 ;
}
```

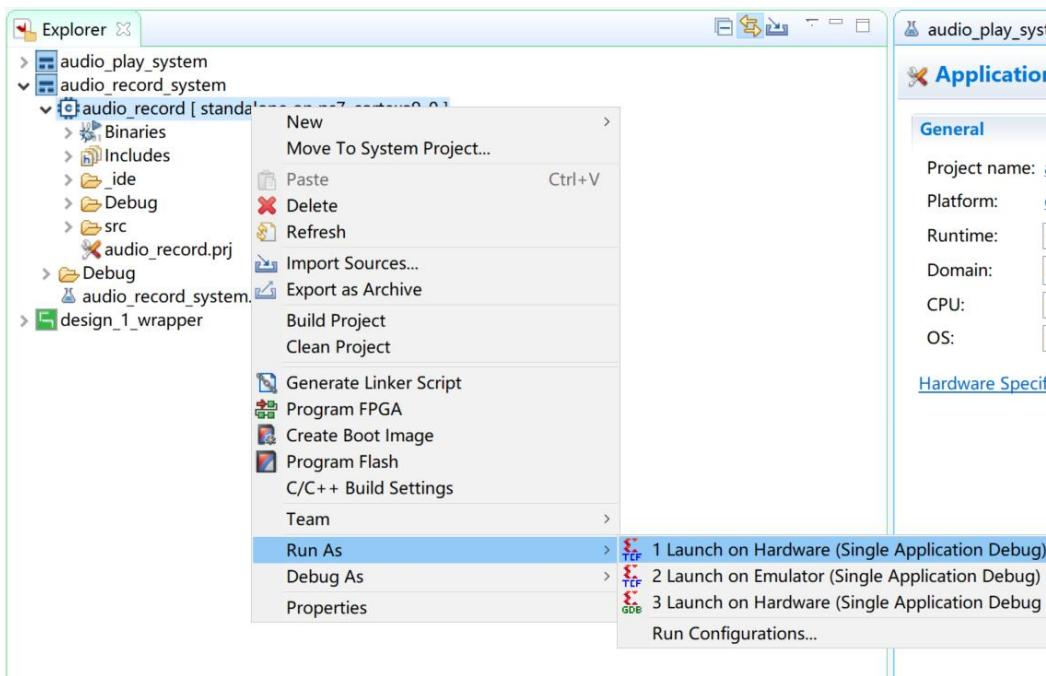
#### Part 32.3.4: On Board Verification

- 1) This experiment does not need to use the SD card, the hardware connection is as follows, insert the headset, microphone



Hardware Connection (Expansion Port 46)

## 2) The download interface is as follows



3) Press the key, the recording PL\_LED will light up, and then go out after 10 seconds. Press the key again, the playback PL\_LED will light up and go out after 10 seconds

## Part 33: Use of 7 inch LCD Module

The experimental Vivado project directory is "lcd7\_touch /vivado".

The experiment vitis project directory is "lcd7\_touch /vitis".

### Part 33.1: 7-inch LCD Module Description

The AN970 LCD touch screen module consists of a TFT LCD screen, a capacitive touch screen and a driver board. For more information, please refer to the AN970 user manual. The real photos of AN970 are as follows:



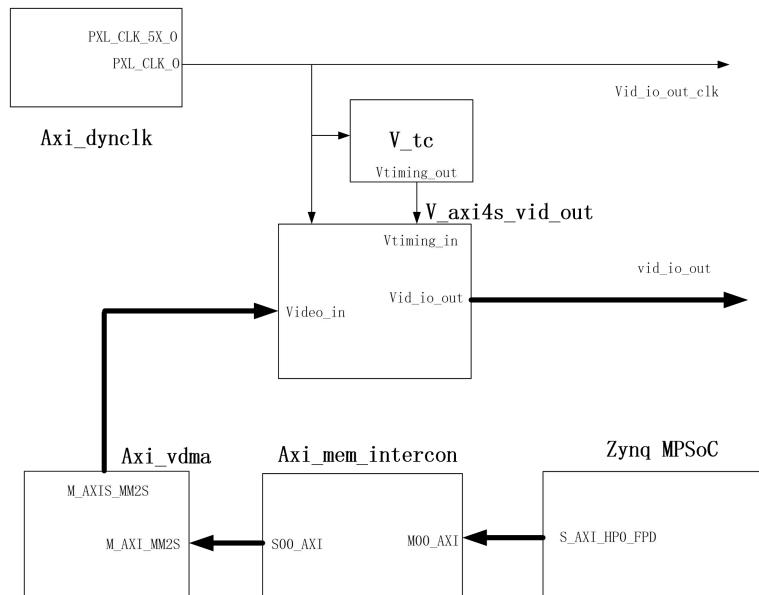
### FPGA Engineer Job Content

The following is the content that FPGA engineers are responsible for.

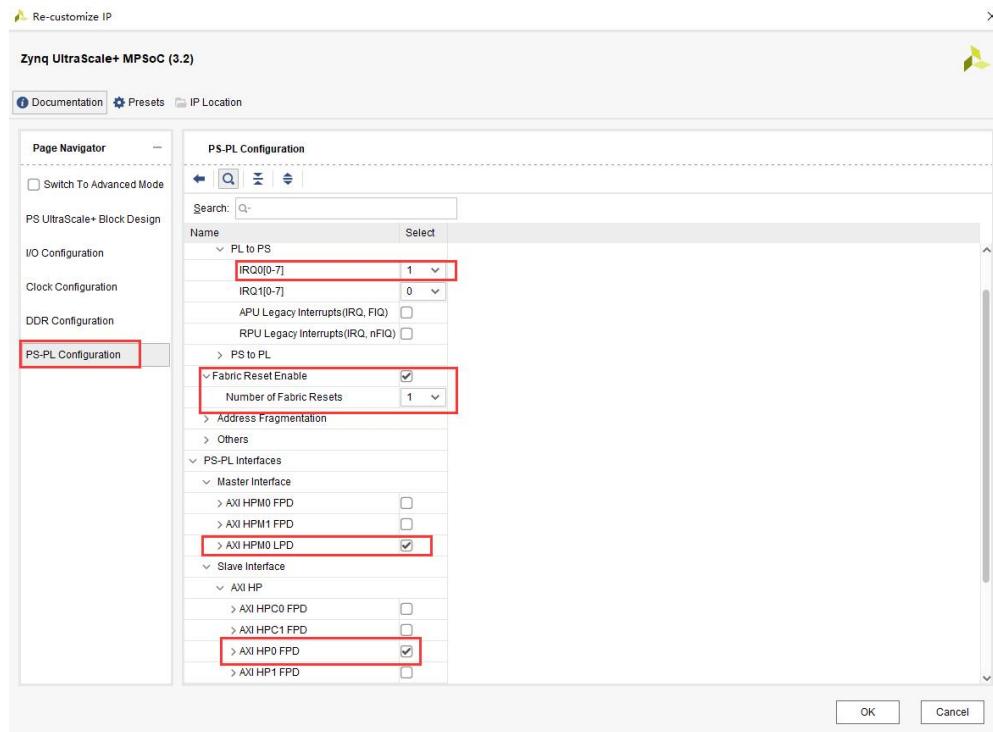
### Part 33.2: Hardware Environment

The video or image data is output from the high-speed AXI\_HPO port of the ZYNQ system. The axi\_vdma IP is connected via the axi

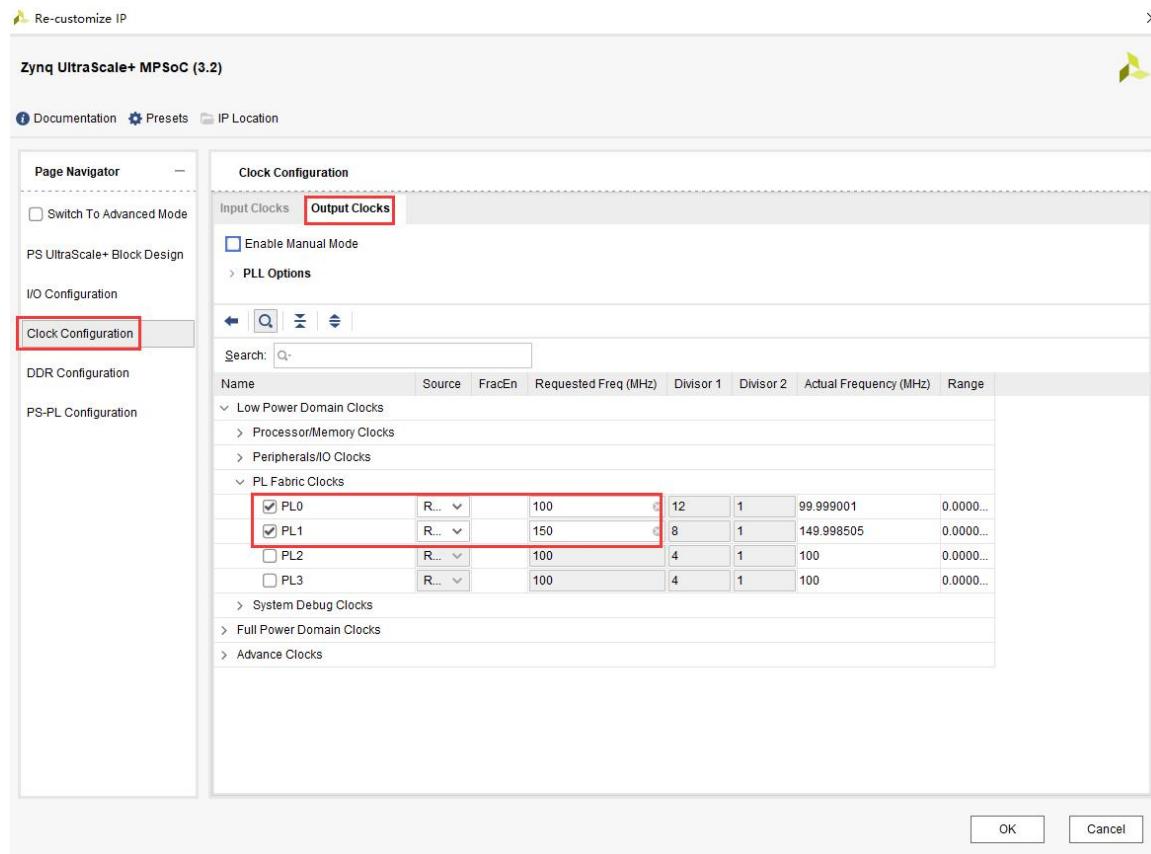
interconnect. The axi\_vdma is connected to the axi4s\_vid\_out IP through the AXI-Stream interface and outputs a drive signal to the LCD module to display the image. The system uses v\_tc IP to control the timing of the display format and resolution of the output, and uses axi\_dynclk IP to generate the image display clock signal of the LCD screen. The structured system is roughly as follows:



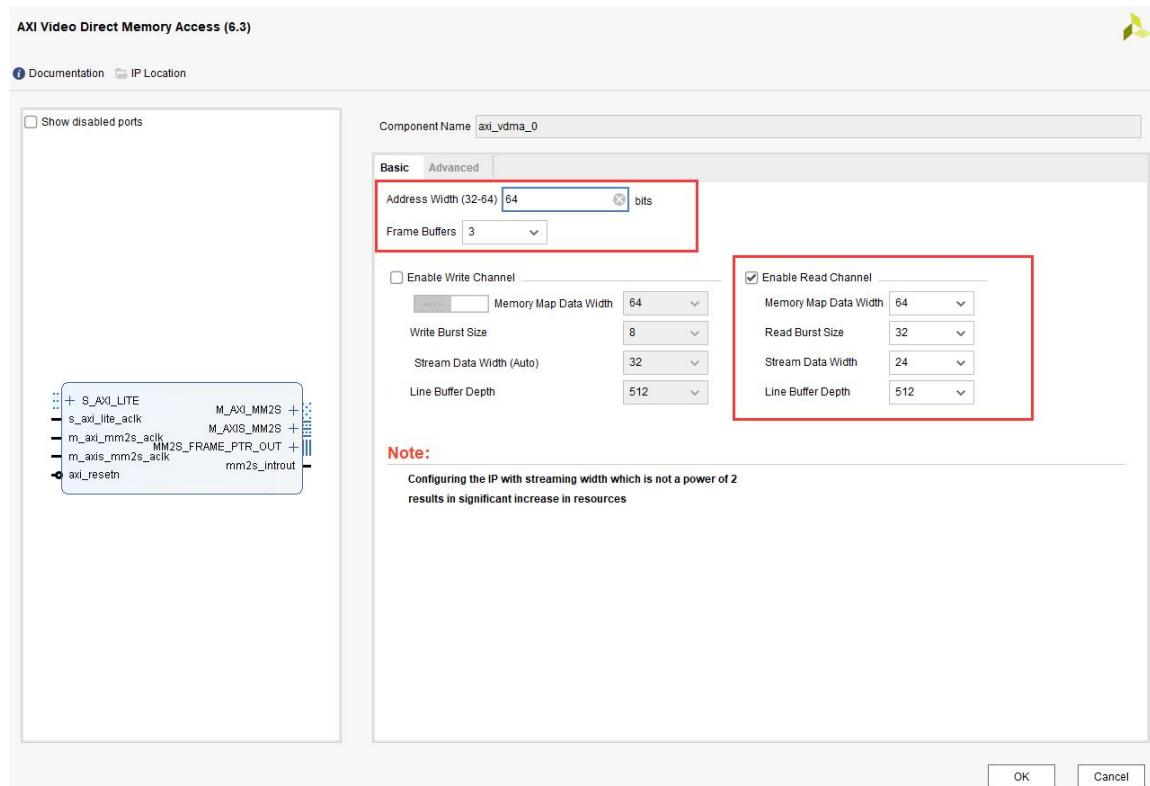
- 1) Based on the "ps\_base" project, save as the project "lcd7\_touch", open the reset, HPM0, HP0, PL to PS interrupt IRQ0



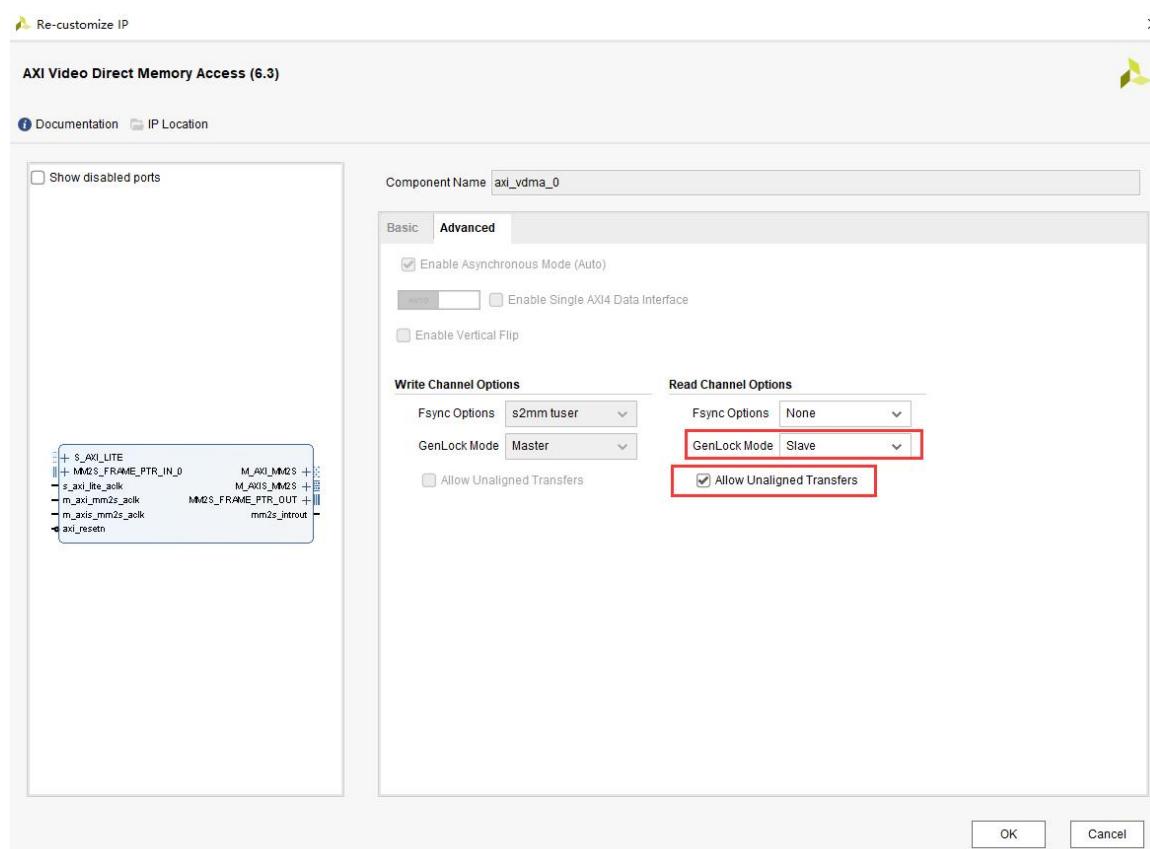
- 2) Zynq core clock configuration, PL0 is configured as 100MHz, used for module register configuration, PL1 is configured as 150MHz, used for VDMA data processing



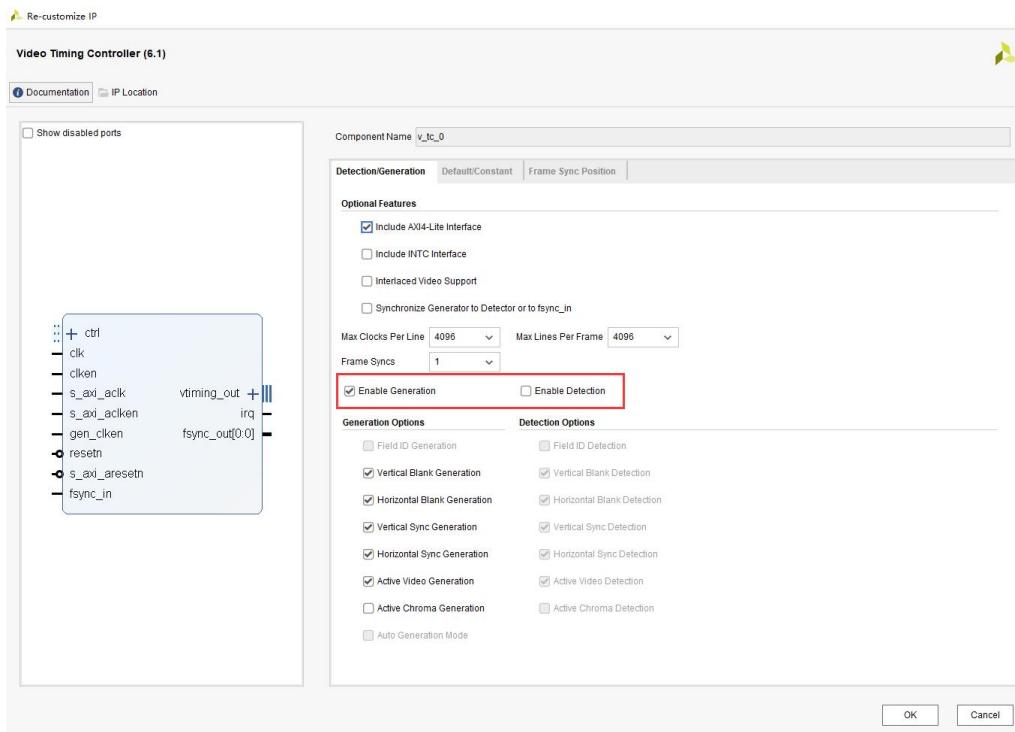
3) Add VDMA IP, configure the basic parameters of VDMA according to the figure below



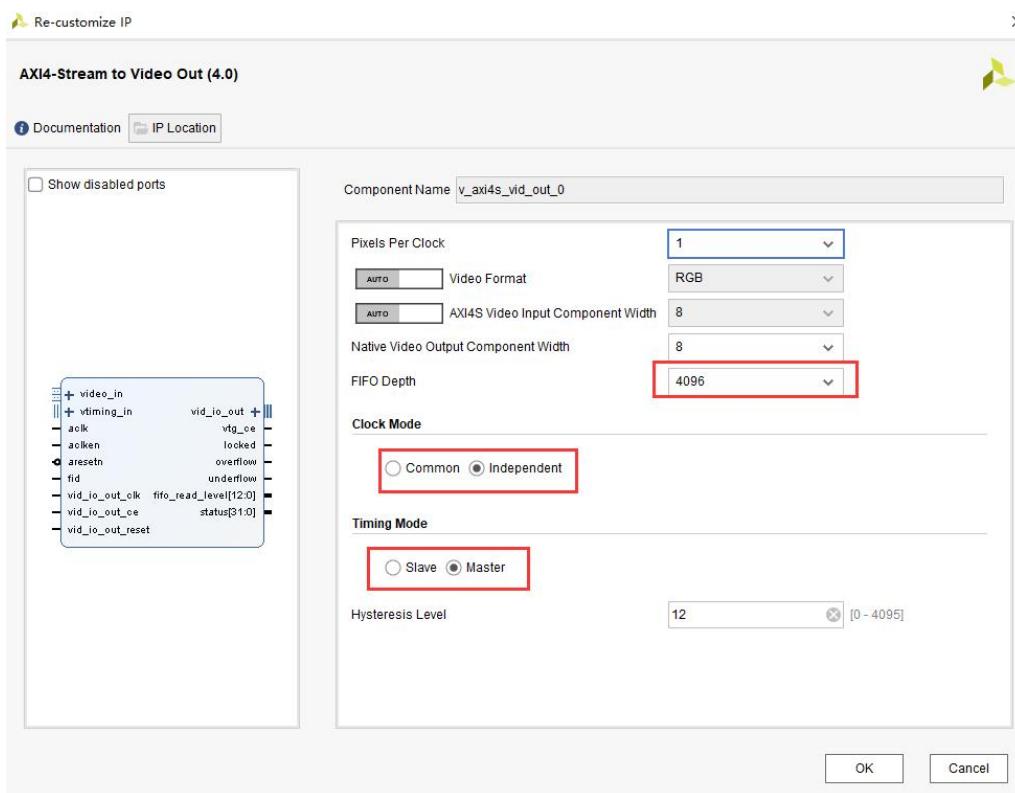
4) Configure VDMA advanced parameters



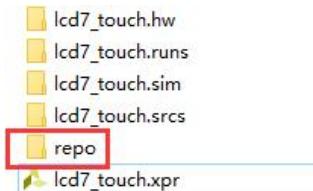
5) Add video timing controller, configure video timing controller parameters



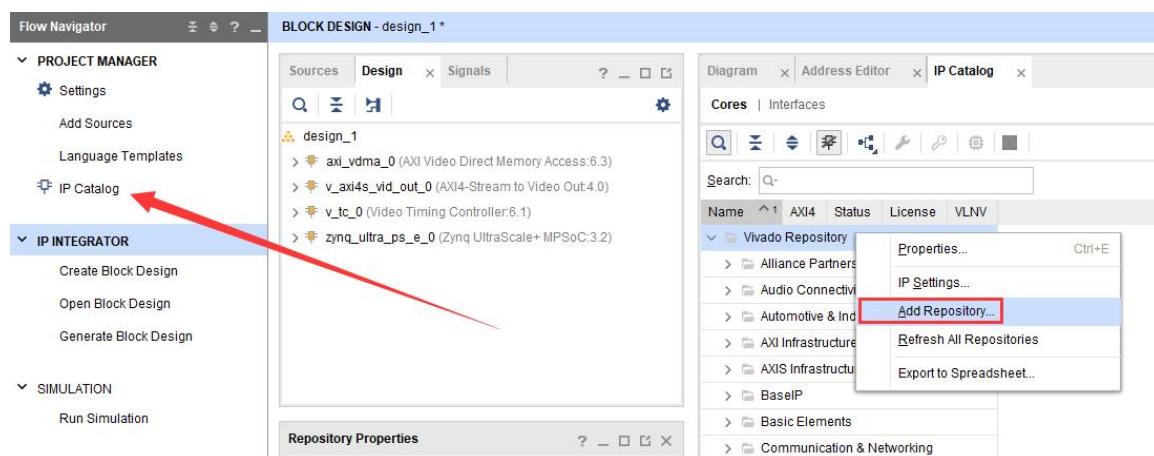
6) Add AXI streaming video output controller, configure AXI streaming video output controller parameters



- 7) Since the video has many resolutions, the clock frequencies of various resolutions are not the same, you need to use a dynamic clock controller, this IP is from open source software, find the repo directory in the routine, and copy it to your own directory



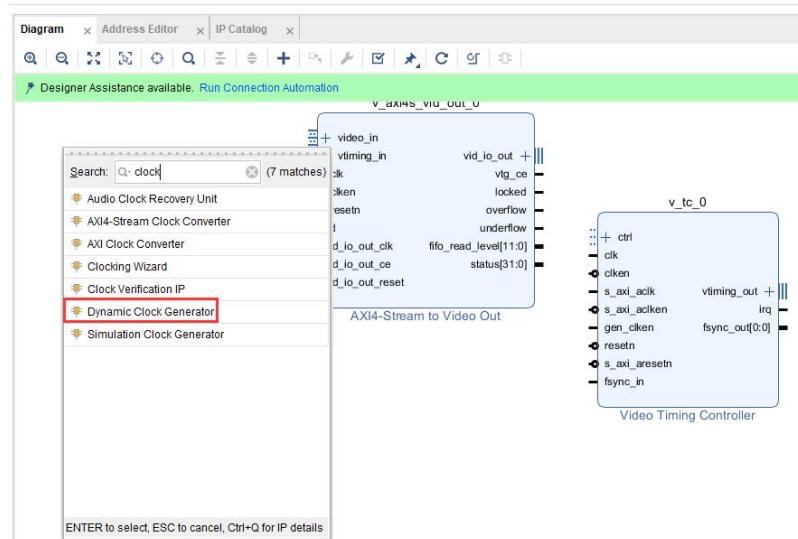
- 8) Add IP Catalog



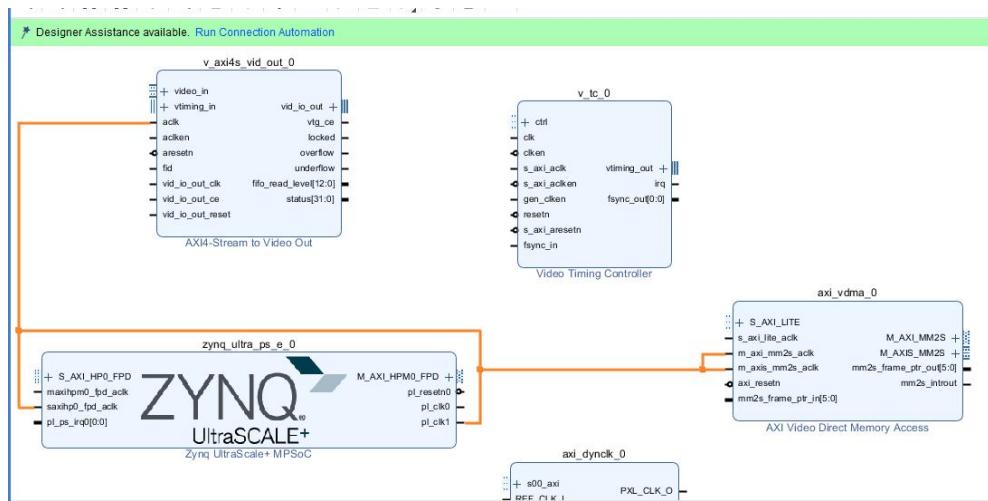
- 9) After the addition is complete, you can see many IPs

Name	AXI4	Status	License	VLAN
User Repository (f...)				
AXI Peripheral				
ax_pwm_v1.0	AXI4	Prod...	Included	xilin...
FPGA Features and Design				
Clocking				
Dynamic Clock Generator	AXI4	Beta	Included	digital...
UserIP				
Dynamic Clock Generator	AXI4	Beta	Included	digital...

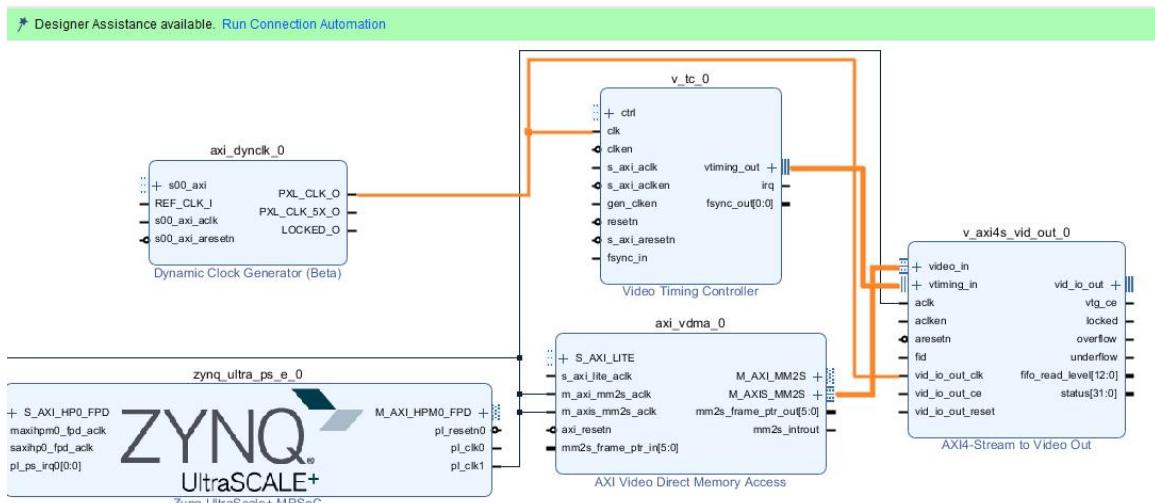
- 10) Add dynamic clock controller



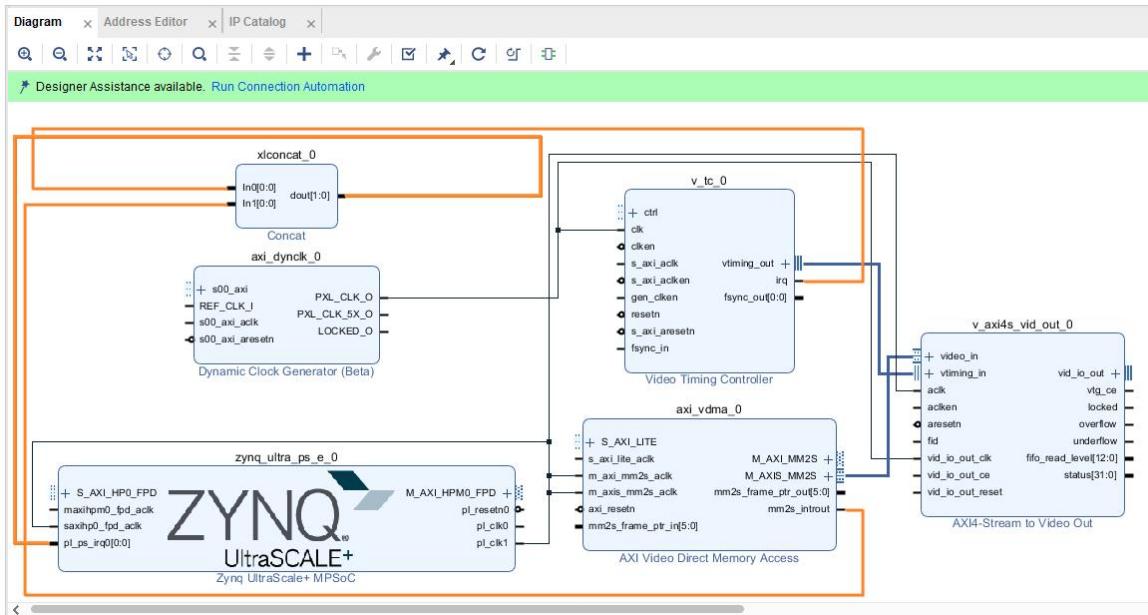
11) Connect to clock signals that Vivado may not be able to connect automatically



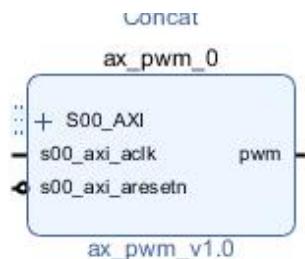
12) Connect some other key signals



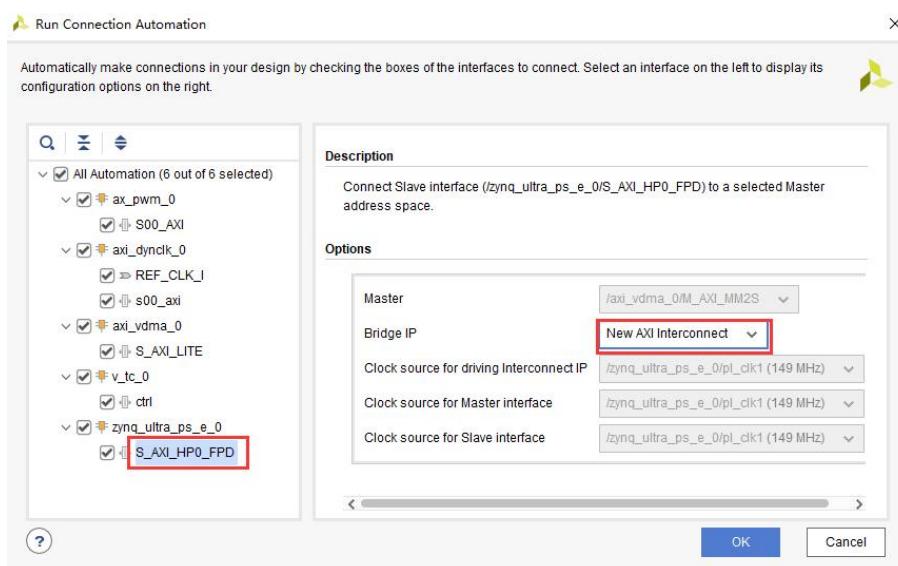
13) Connection interrupt signal, you need to add a Concat IP for signal connection



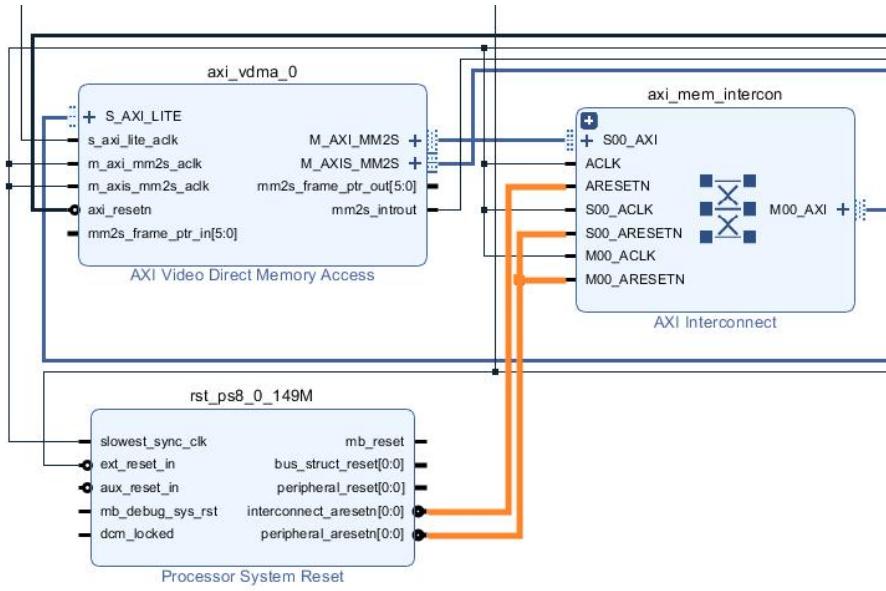
14) Add a PWM module to adjust the brightness of the screen



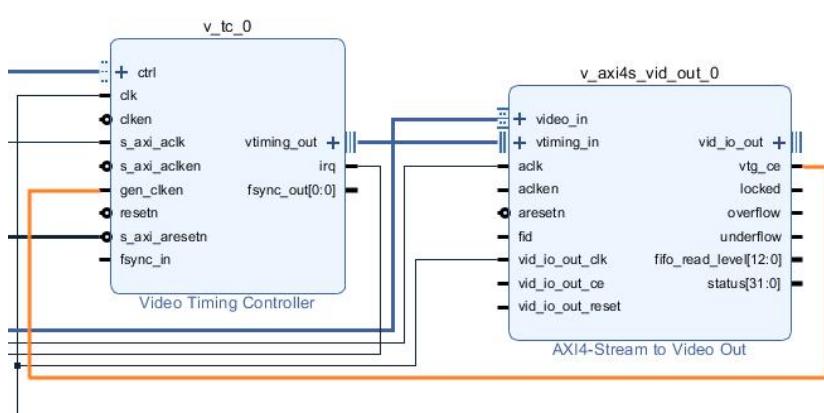
15) Select all modules to automatically connect, and select the bridge IP of S\_AXI\_HP0\_FPD as AXI Interconnect



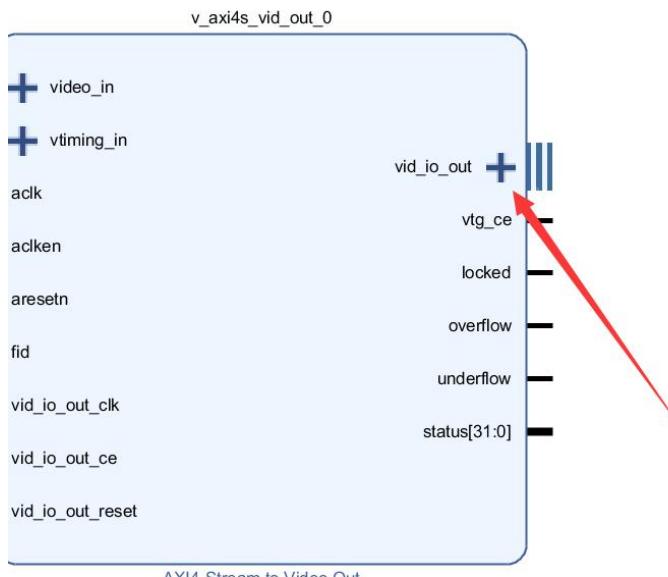
## 16) Adjust the reset of the bridge module



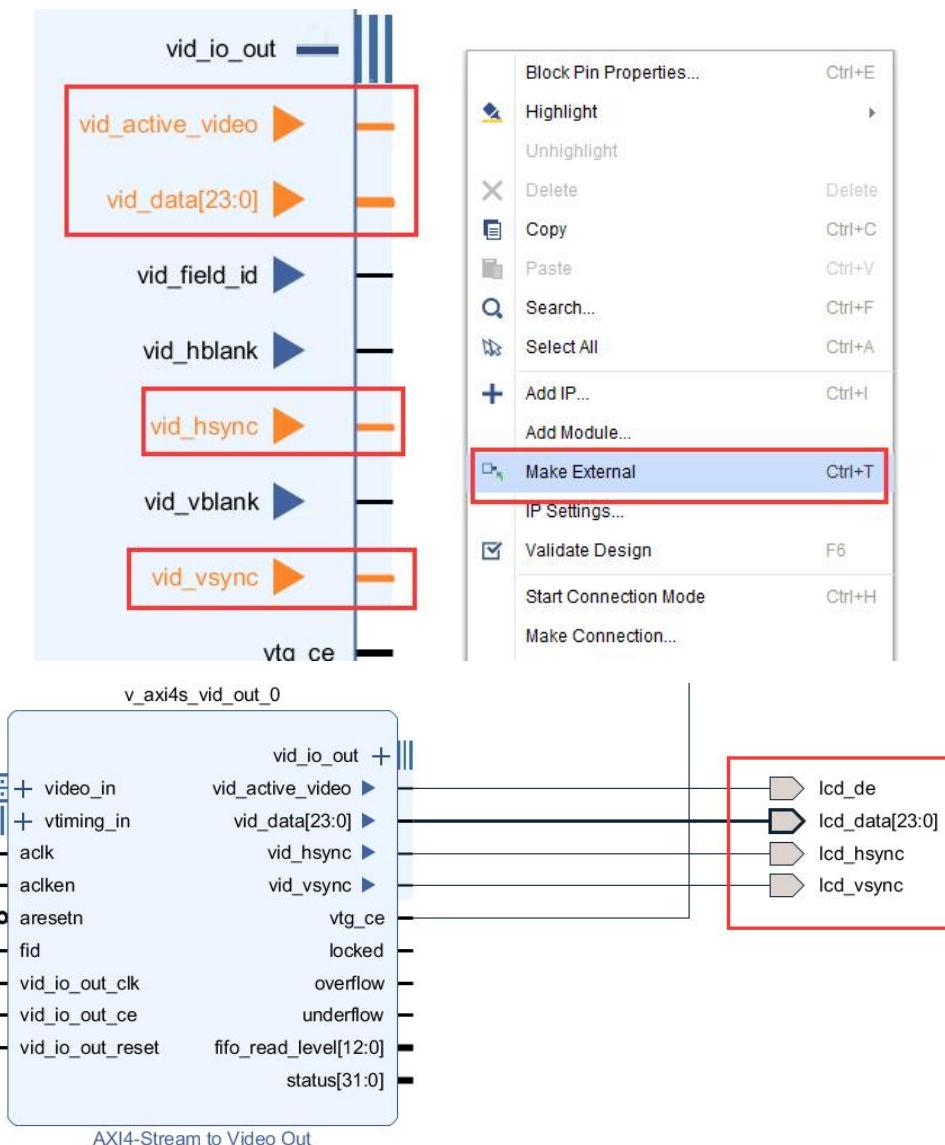
Connect some unconnected pins



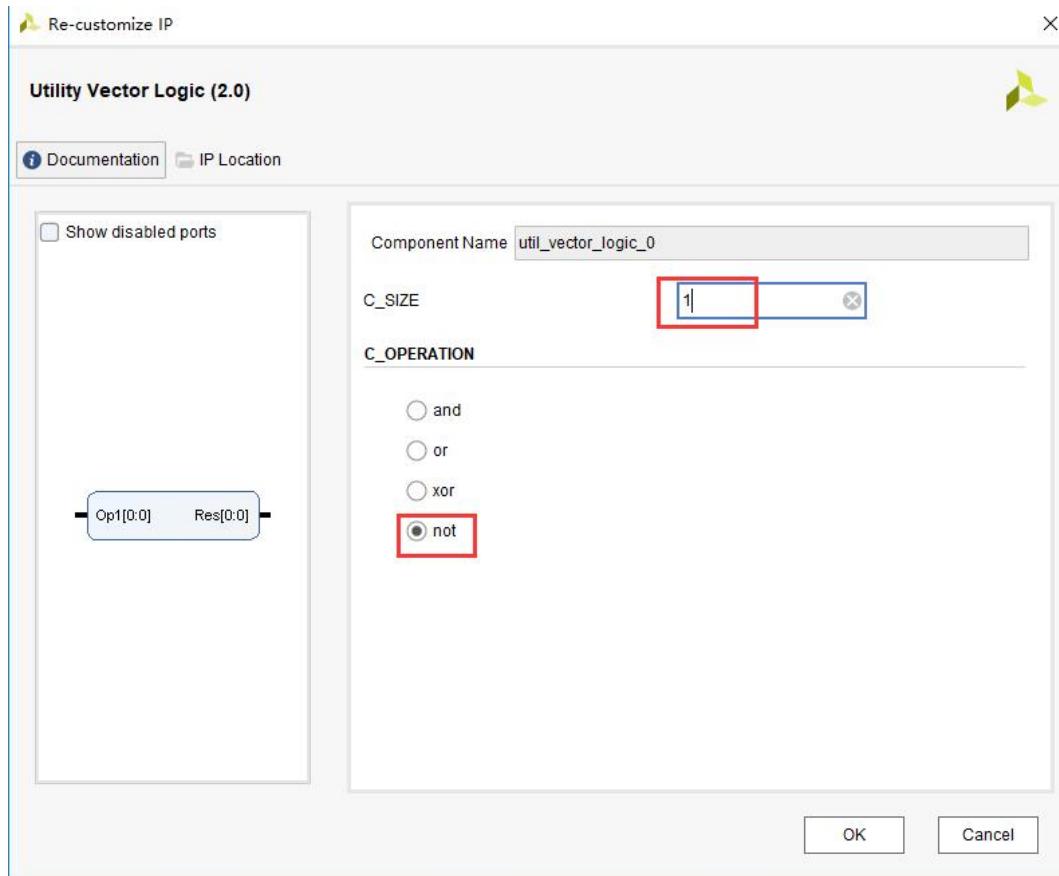
## 17) Expand the vid\_io\_out port



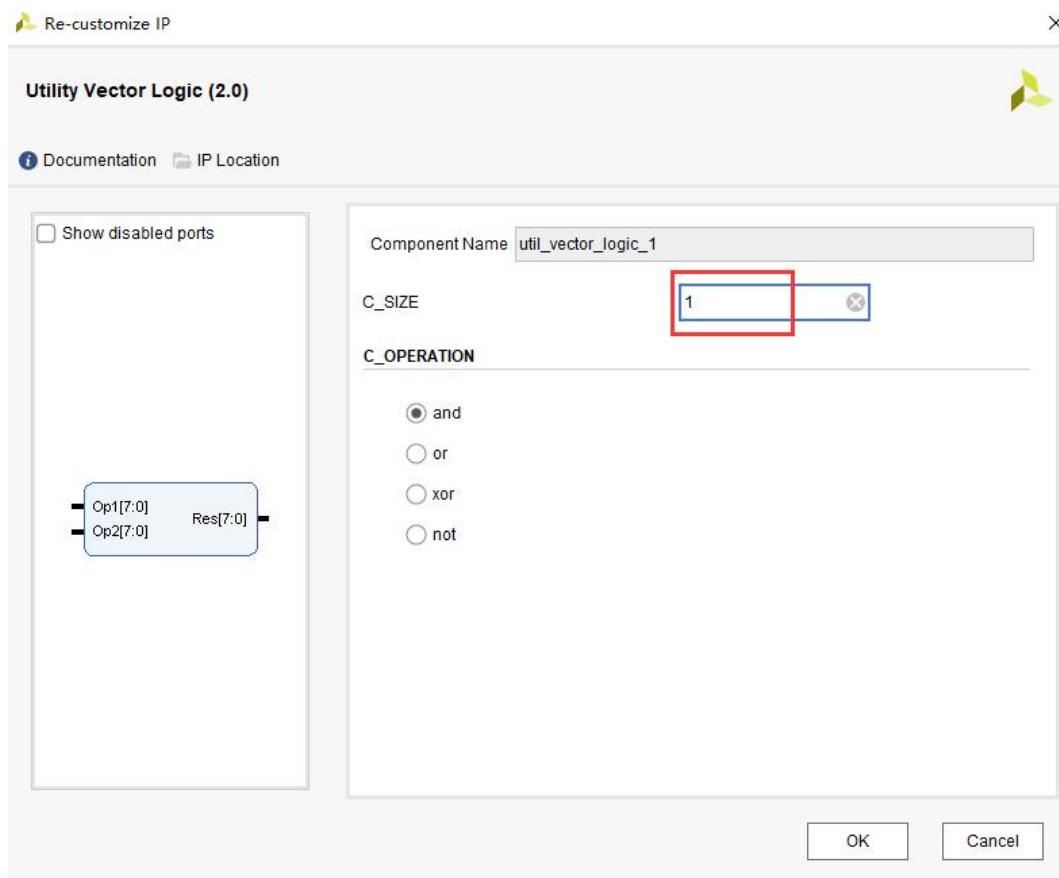
18) Select the port export we need, and modify the pin name



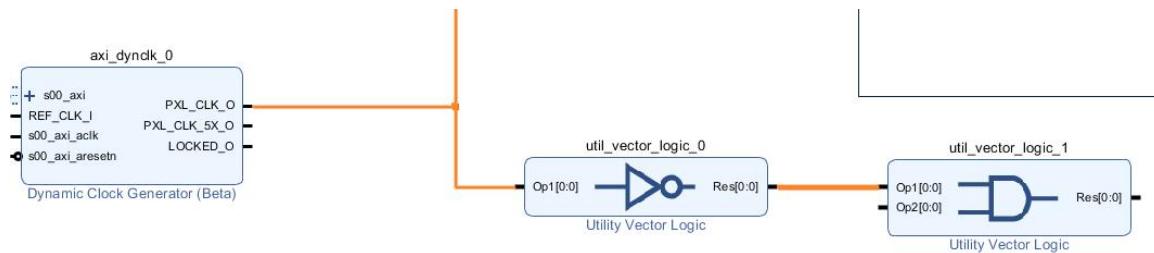
19) Add Utility Vector Logic module, set to “not”, 1 bit



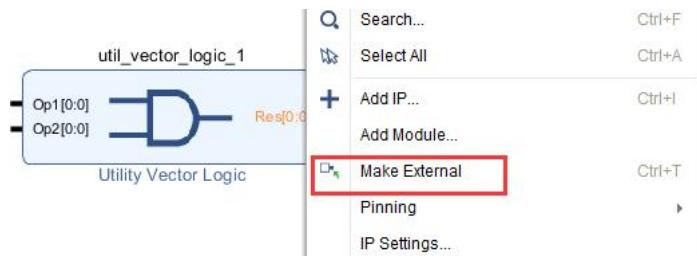
Add one more, set to “and”, 1 bit



Connect as shown below



## 20) Export video clock port



And modify the name to lcd\_dclk



## 21) Modify the names of other ports



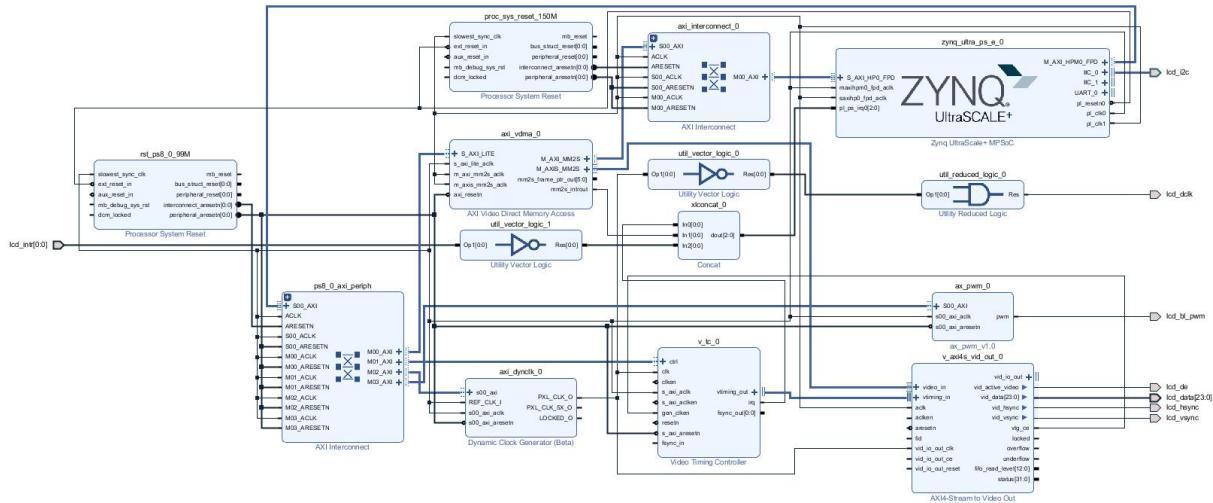
Configure IIC in the ZYNQ core and set it to EMIO, export the pin, and modify the name



Add the interface lcd\_intr, connect to the interrupt signal of the touch screen, low level means that an interrupt is generated, so an inverter is added to connect to the interrupt



The designed vivado project is shown in the figure below:



22) After saving the design, press F6 to check the design, and constrain the pins in XDC to generate bitstream

## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 33.3: Vitis Program Development

The software program of Vitis is to output a picture to a 7-inch screen. The picture data is stored in the "pic\_800\_600.h" file. The picture is 800x600 pixels, and the image size of our LCD screen is 800x480, so the whole picture cannot be displayed. If the user wants to change another picture, he can regenerate it himself.

In the "vga\_modes.h" file we need to add the timing parameters of the 7-inch screen "800\*480". The software will configure this parameter to "v\_tc IP" to generate the correct "LCD" screen drive timing.

```

Explorer [ lcd_display_system [ system_wrapper ]
  <--> Binaries
  <--> Includes
  <--> ide
  <--> Debug
  <--> src
    <--> display_ctrl
      <--> display_ctrl.c
      <--> display_ctrl.h
      <--> vga_modes.h
    <--> dynclk
    <--> display_demo.c
    <--> display_demo.h
    <--> pic_800_600.h
    <--> lscript.ld
    <--> README.txt
    <--> Xilinx.spec
    <--> lcd_display.prj
  <--> Debug
  <--> lcd_display_system.sprj

lcd_display_system [ lcd_display ] [ vga_modes.h ]
52  };
53
54 static const VideoMode VMODE_800x480 = {
55   .label = "800x480@60Hz",
56   .width = 800,
57   .height = 480,
58   .hps = 840,
59   .hpe = 968,
60   .hmax = 1056,
61   .hpol = 0,
62   .vps = 481,
63   .vpe = 484,
64   .vmax = 505,
65   .vpol = 0,
66   .freq = 33.0
67 };
68
69 static const VideoMode VMODE_800x600 = {
70   .label = "800x600@60Hz",
71   .width = 800,
72

```

Modify the value of “`dispPtr->vMode`” in the “`display_ctrl.c`” file to “`VMOD_800_480`”.

```

Explorer [ lcd_display_system [ system_wrapper ]
  <--> Binaries
  <--> Includes
  <--> ide
  <--> Debug
  <--> src
    <--> display_ctrl
      <--> display_ctrl.c
      <--> display_ctrl.h
      <--> vga_modes.h
    <--> dynclk
    <--> display_demo.c
    <--> display_demo.h
    <--> pic_800_600.h
    <--> lscript.ld
    <--> README.txt
    <--> Xilinx.spec
    <--> lcd_display.prj
  <--> Debug
  <--> lcd_display_system.sprj

lcd_display_system [ lcd_display ] [ display_ctrl.c ]
322   * Initialize all the fields in the DisplayCtrl struct
323   /*
324   dispPtr->curFrame = 0;
325   dispPtr->dynClkAddr = dynClkAddr;
326   for (i = 0; i < DISPLAY_NUM_FRAMES; i++)
327   {
328     dispPtr->framePtr[i] = framePtr[i];
329   }
330   dispPtr->state = DISPLAY_STOPPED;
331   dispPtr->stride = stride;
332   dispPtr->vMode = VMODE_800x480;
333
334   ClkFindParams(dispPtr->vMode.freq, &clkMode);
335
336   /*
337   * Store the obtained frequency to pxFreq. It is possible that the PLL was not able to
338   * exactly generate the desired pixel clock, so this may differ from vMode.freq.
339   */
340   dispPtr->pxlFreq = clkMode.freq;
341
342   /*

```

Configure the frequency and duty cycle of the “PWM” in the “`main.c`” file. The second parameter of “`set_pwm_frequency`” is the base clock. The connection to Vivado is 100MHz. The third parameter is the PWM frequency to be set, in Hz.

The second parameter of “`Set_pwm_duty`” is the duty cycle.

```

int main(void)
{
    int Status;
    XAxiVdma_Config *vdmaConfig;
    int i;
    //pwm out period = frequency(pwm_out) * (2 ** N) / frequency(clk);
    set_pwm_frequency(XPAR_AX_PWM_0_S00_AXI_BASEADDR,100000000,200);
    set_pwm_duty(XPAR_AX_PWM_0_S00_AXI_BASEADDR,0.5);

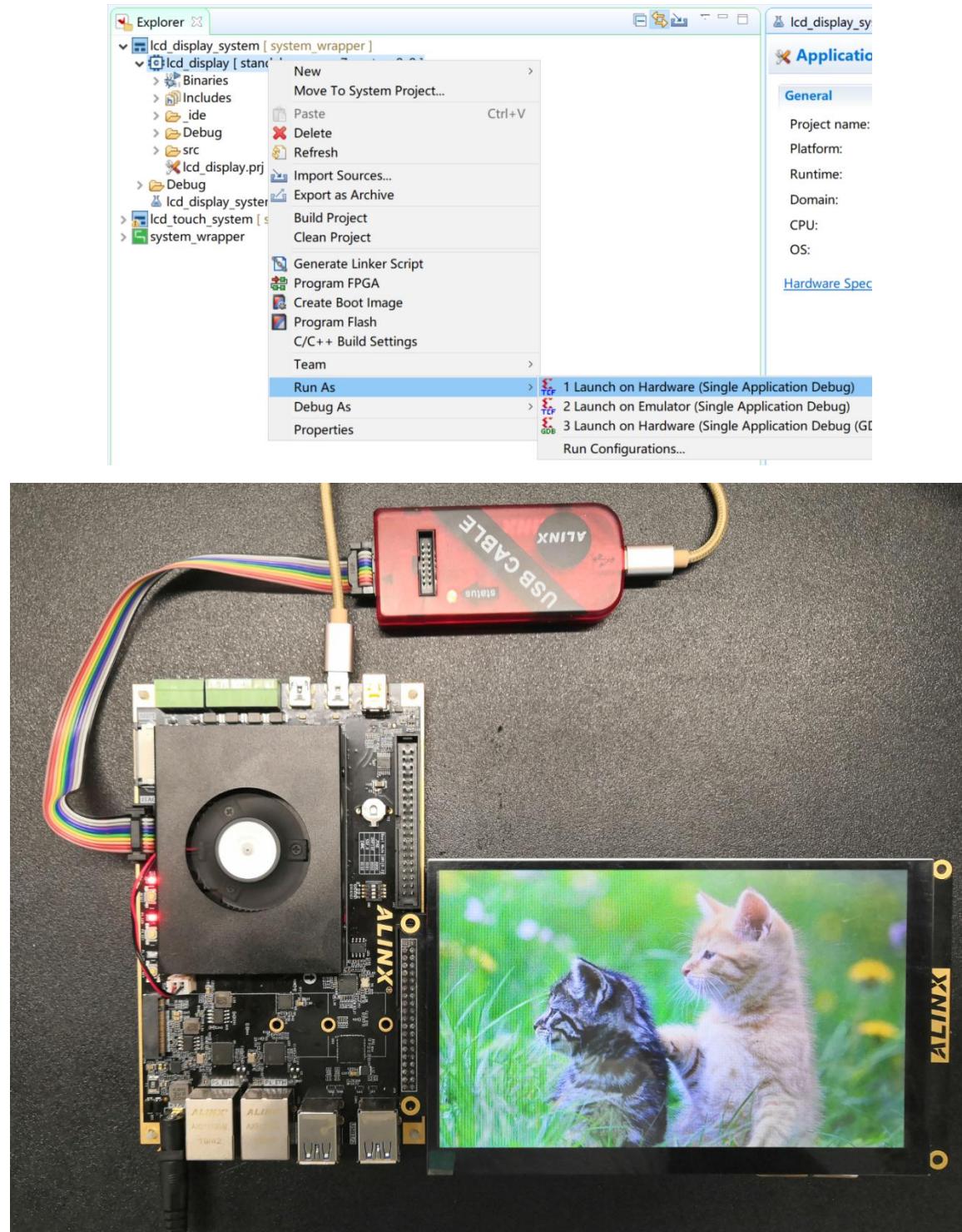
    /*
    * Initialize an array of pointers to the 3 frame buffers
    */
    for (i = 0; i < DISPLAY_NUM_FRAMES; i++)
    {
        pFrames[i] = frameBuf[i];
    }
}

```

## Part 33.4: Onboard Verification

As with the previous routines, compile and generate the bit file,

then export the hardware and run the Vitis. The 7-inch LCD screen is connected to the expansion port of the FPGA development board. Click the “run as” to download the programs. You can see that just like the VDMA test routine, a picture of a kitten will be displayed on the LCD screen.



## Part 34: 7-inch Touch Screen GUI and Touch Control

The experimental Vivado project directory is "lcd7\_touch /vivado".

The experiment vitis project directory is "lcd7\_touch /vitis".

The focus of this chapter is to master the use of touch controllers and the use of GUI with touch functions.

### Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

#### Part 34.1: Use of μGUI

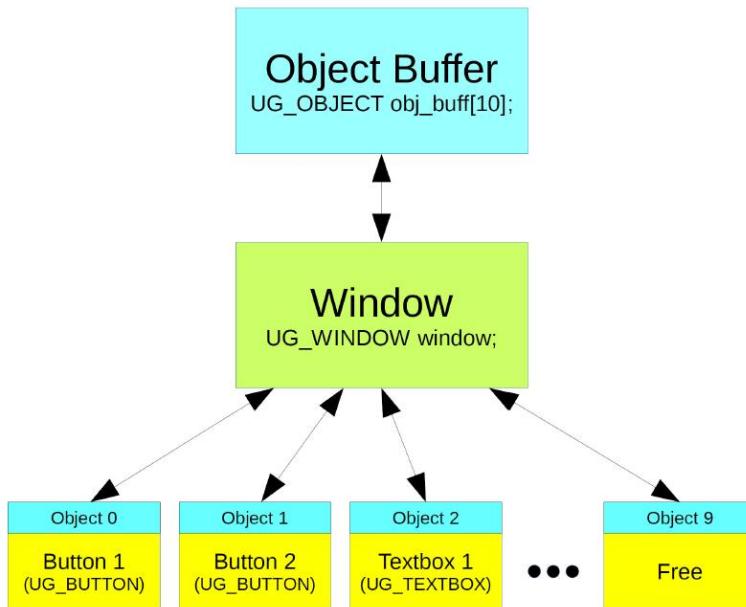


μGUI Running Result

μGUI is a lightweight human-machine interface open source library, suitable for single-chip, no operating system environment. The official website of μGUI is

<http://embeddedlightning.com/ugui/>

μGUI has a simple window management system that can define regular image interface elements such as keys and text boxes.



```

void window_1_callback( UG_MESSAGE* msg )
{
    // ...
}

#define MAX_OBJECTS 10
int main( void )
{
    UGWINDOW window_1; /* Window */
    UGBUTTON button_1; /* Button container */
    UGBUTTON button_2; /* Button container */
    UGBUTTON button_3; /* Button container */
    UGOBJECT obj_buff_wnd_1[MAX_OBJECTS]; /* Object buffer */

    // ...

    /* Create the window (link the object buffer and the callback function to the
     * window) */
    UGWindowCreate( &window_1, obj_buff_wnd_1, MAX_OBJECTS, window_1_callback );

    /* Create some buttons (link each object container to the window) */
    UGButtonCreate( &window_1, &button_1, BTN_ID_0, 10, 10, 110, 60 );
    UGButtonCreate( &window_1, &button_2, BTN_ID_1, 10, 80, 110, 130 );
    UGButtonCreate( &window_1, &button_3, BTN_ID_2, 10, 150, 110, 200 );

    /* Finally, show the window */
    UGWindowShow( &window_1 );

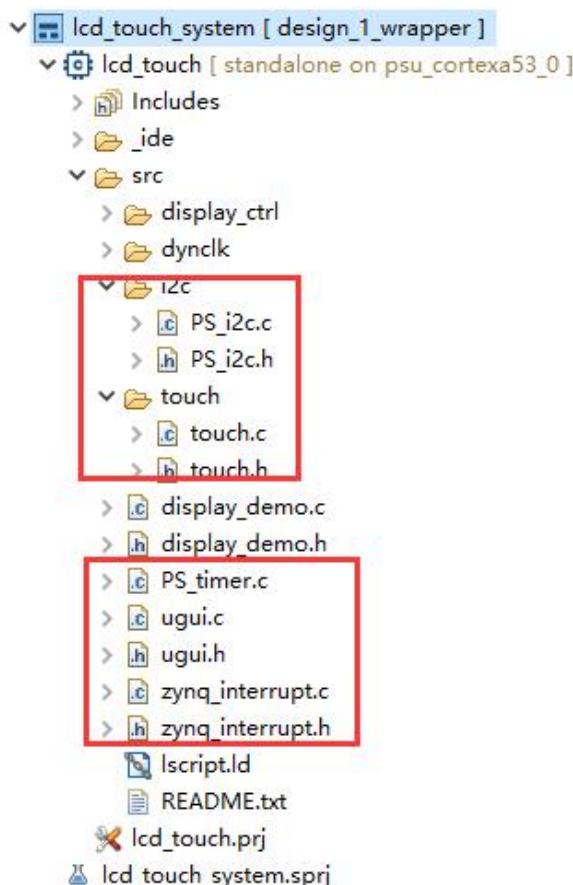
    // ...
}

```

For detailed usage, refer to µGUI's "Reference Guide", which is not the focus of this article.

## Part 34.2: Vitis Program Development

In the touch screen Vitis program, we added the following code to the 7-inch screen display routine. PS I2C communication program (PS\_i2c.c and PS\_i2c.h), touch program (touch.c and touch.h), PS timer code (PS\_timer.c), interrupt program (zynq\_interrupt.c) and uGUI library Programs (ugui.c and ugui.h).



Here's a brief introduction to the newly added code:

### 1) PS I2C communication program (PS\_i2c.c and PS\_i2c.h)

The PS I2C communication program contains three functions: the I2C initialization function `i2c_init`, the I2C byte write function `i2c_wrtie_bytes`, and the I2C byte read function `i2c_read_bytes`.

These three functions are used to read and write data from the

registers of the touch screen.

```
int i2c_init(XIicPs *IicInstance, u16 DeviceId, u32 Fsc1Hz);

int i2c_wrtie_bytes(XIicPs *IicInstance, u8 i2c_slave_addr, void *buf, int byte_num);

int i2c_read_bytes(XIicPs *IicInstance, u8 i2c_slave_addr, void *buf, int byte_num);
```

## 2) Touch program (touch.c and touch.h)

The touch program contains three functions, one is the register read function of the touch screen I2C, and the device address of the touch screen is “0xA0”. Here, the “i2c\_write\_bytes” and “i2c\_read\_bytes” in the “I2C” communication program are called to complete the register read operation of the touch screen.

One is the initialization function of the touch screen, which connects the interrupt response function. The other is the response function of the touch screen interrupt, setting the variable “touch\_sig” to high.

```
int touch_i2c_read_bytes(u8 *BufferPtr, u8 address, u16 ByteCount);
int touch_init (void);
void Touch_Intr_Handler(void *InstancePtr);
```

## 3) Timer program (PS\_timer.c)

There is only one timer initialization function (PS\_timer\_init) in the program. The function initializes the PS timer, resets the initial value of the timer, sets the auto load, and enables the timer to start counting.

```
int PS_timer_init(XScuTimer *Timer, u16 DeviceId, u32 timer_load);
```

## 4) uGUI library program (ugui.c)

This is the open source uGUI library program we downloaded online. It defines a lot of functions and fonts for drawing lines, frames,

keys, and texts. Please refer to the “uGUI Reference Guide” documentation for the use and description of the specific functions.

## 5) ZYNQ interrupt program (zynq\_interrupt.c)

Three functions are defined in the interrupt program, one is the interrupt system setup function, one is the interrupt initialization function, and the other is the interrupt response function connection.

## 6) Main Program( display\_demo.c )

First let's take a look at the main function. In the main function we added the “pwm” setting, interrupt, touch initialization, timer initialization and timer interrupt enable.

```
int main(void)
{
    int Status;
    XAxiVdma_Config *vdmaConfig;
    int i;
    /*
     * Initialize an array of pointers to the 3 frame buffers
     */
    for (i = 0; i < DISPLAY_NUM_FRAMES; i++)
    {
        pFrames[i] = frameBuf[i];
    }
    set_pwm_frequency(XPAR_AXI_PWM_0_S00_AXI_BASEADDR,100000000,200);
    duty = 0.5;
    set_pwm_duty(XPAR_AXI_PWM_0_S00_AXI_BASEADDR,duty);
    InterruptInit(XPAR_SCUGIC_SINGLE_DEVICE_ID,&XScuGicInstance);
    i2c_init(&ps_i2c0, XPAR_XIICPS_0_DEVICE_ID,100000);
    touch_init(&ps_i2c0,&XScuGicInstance,123);
    PS_timer_init(&Timer, XPAR_XTTCPS_0_DEVICE_ID ,100000); //100KHz. 10ms interval
    InterruptConnect(&XScuGicInstance,XPAR_XTTCPS_0_INTR,Timer_Handler,(void *)&Timer);
    XTTCPS_EnableInterrupts(&Timer, XTTCPS_IXR_INTERVAL_MASK);
    /*
     * Initialize VDMA driver
     */
    vdmaConfig = XAxiVdma_LookupConfig(VGA_VDMA_ID);
    if (!vdmaConfig)
    {
        xil_printf("No video DMA found for ID %d\r\n", VGA_VDMA_ID);
    }
    Status = XAxiVdma_CfgInitialize(&vdma, vdmaConfig, vdmaConfig->BaseAddress);
    if (Status != XST_SUCCESS)
    {
        xil_printf("VDMA Configuration Initialization failed %d\r\n", Status);
    }
}
```

After the main function, we call the “uGUI” function to implement a window on the 7-inch LCD screen, and display different patterns on this window. Here we add a menu bar, which displays the "%UI For ALINX!" character on the menu bar. Add 6 keys at different positions on the window, and display different text on each key to indicate different functions.

```
cpFrames =(u8 *)dispCtrl.framePtr[dispCtrl.curFrame];
UG_Init(&gui , PixelSet, 800 , 480);
UG_FillScreen( 0xFF0000 );
UG_DrawCircle(100 , 100 , 30 , C_WHITE) ;
Xil_DCacheFlushRange((unsigned int) dispCtrl.framePtr[dispCtrl.curFrame], DEMO_MAX_FRAME);
/* Create Window 1 */
UG_WindowCreate( &window_1, obj_buff_wnd_1, MAX_OBJECTS, window_1_callback );
UG_WindowSetTitleText( &window_1, "UI For ALINX!" );
UG_WindowSetTitleTextFont( &window_1, &FONT_12X20 );

/* Create some Buttons */
UG_ButtonCreate( &window_1, &button1_1, BTN_ID_0, 10, 10, 110, 60 );
UG_ButtonCreate( &window_1, &button1_2, BTN_ID_1, 10, 80, 110, 130 );
UG_ButtonCreate( &window_1, &button1_3, BTN_ID_2, 10, 150, 110,200 );
UG_ButtonCreate( &window_1, &button1_4, BTN_ID_3, 120, 10, 360 , 60 );
UG_ButtonCreate( &window_1, &button1_5, BTN_ID_4, 120, 80, 360, 130 );
UG_ButtonCreate( &window_1, &button1_6, BTN_ID_5, 120, 150,360, 200 );

/* Configure Button 1 */
UG_ButtonSetFont( &window_1, BTN_ID_0, &FONT_12X20 );
UG_ButtonSetBackColor( &window_1, BTN_ID_0, C_LIME );
UG_ButtonSetText( &window_1, BTN_ID_0, "Green\nLED" );
/* Configure Button 2 */
UG_ButtonSetFont( &window_1, BTN_ID_1, &FONT_12X20 );
UG_ButtonSetBackColor( &window_1, BTN_ID_1, C_RED );
UG_ButtonSetText( &window_1, BTN_ID_1, "Red\nLED" );
/* Configure Button 3 */
UG_ButtonSetFont( &window_1, BTN_ID_2, &FONT_12X20 );
UG_ButtonSetText( &window_1, BTN_ID_2, "About\nUI" );
UG_WindowShow( &window_1 );
/* Configure Button 4 */
UG_ButtonSetFont( &window_1, BTN_ID_3, &FONT_12X20 );
UG_ButtonSetForeColor( &window_1, BTN_ID_3, C_RED );
UG_ButtonSetText( &window_1, BTN_ID_3, "LCD brightness\n - " );
/* Configure Button 5 */
UG_ButtonSetFont( &window_1, BTN_ID_4, &FONT_8X14 );
UG_ButtonSetText( &window_1, BTN_ID_4, "LCD brightness\n + " );
/* Configure Button 6 */
UG_ButtonSetFont( &window_1, BTN_ID_5, &FONT_10X16 );
UG_ButtonSetText( &window_1, BTN_ID_5, "Resize\nWindow" );

UG_WindowShow( &window_1 );
```

There are two functions in the “display\_demo.c” program, one is the timer interrupt handler “Timer\_Handler”. When the timer overflows, an interrupt will be generated, and the “Timer\_Handler” interrupt program will be entered. The value of the touch screen register will be read in the interrupt program to calculate the touch state and the touch position.

```
static void Timer_Handler(void *CallBackRef);
```

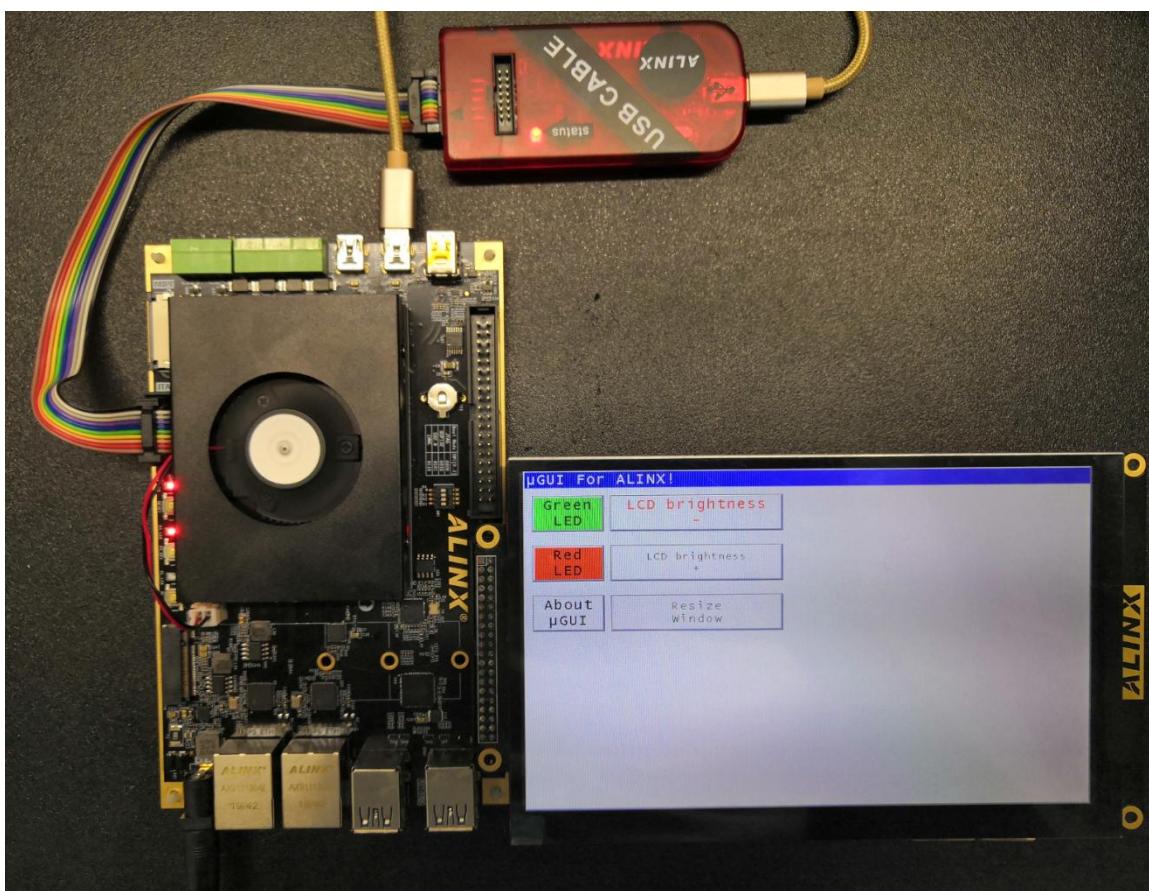
There is also a window handler program “window\_1\_callback”, which is used to process the touch event, and generate different processing results according to the touched key position. For example, if the key of the “green led” is touched, the number on the key will increase, and each time you press it, the number is increased by 1; If you touch the key of the “Resize Window”, the size of the window will

change. If you press the “LCD brightness+/-“ key, the screen brightness will change.

```
void window_1_callback( UG_MESSAGE* msg );
```

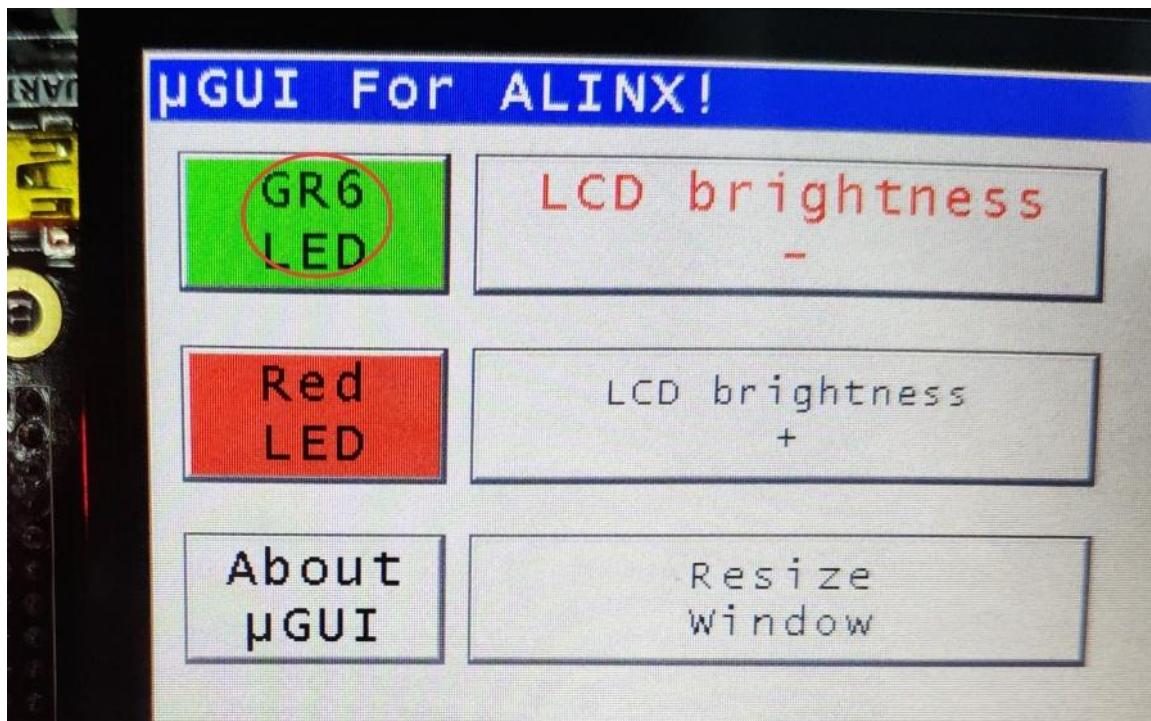
### Part 34.3: Onboard Verification

As with the previous routines, compile and generate the “bit” file, then export the hardware and run the “Vitis”. Connect the 7-inch touch LCD screen to the expansion port of the FPGA development board. Run the configuration according to the previous chapters. Click “run” to see the following window interface on the 7-inch screen.

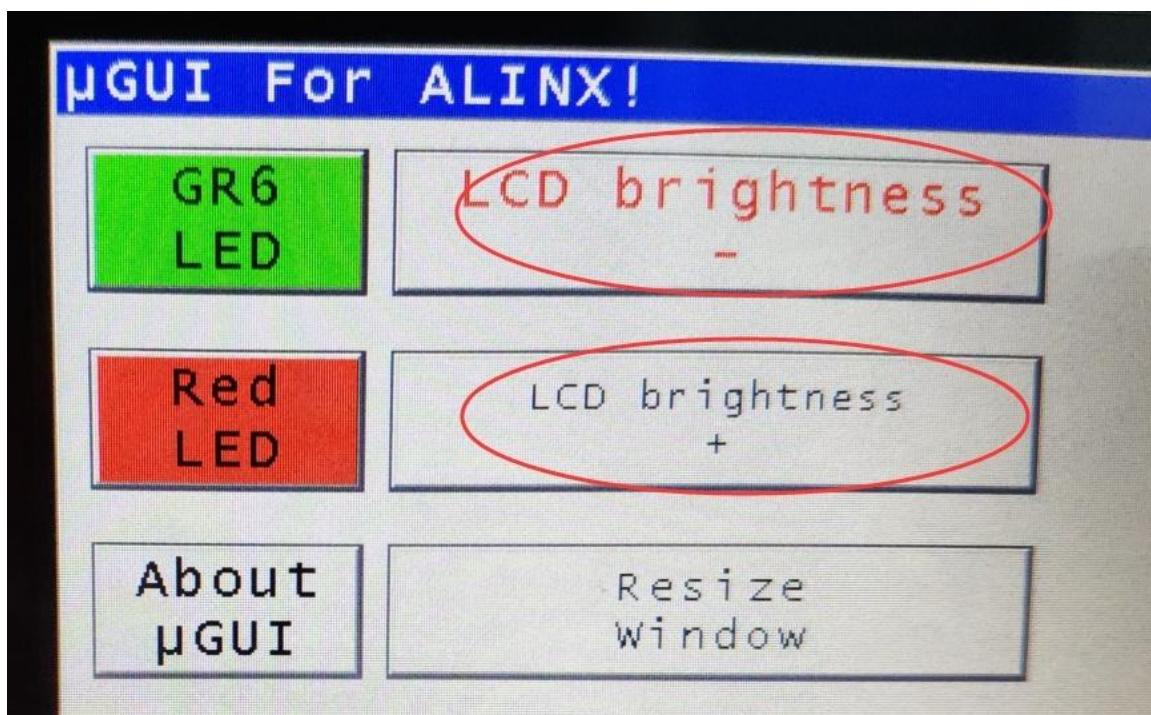


Hardware Connection (Expansion Port 45)

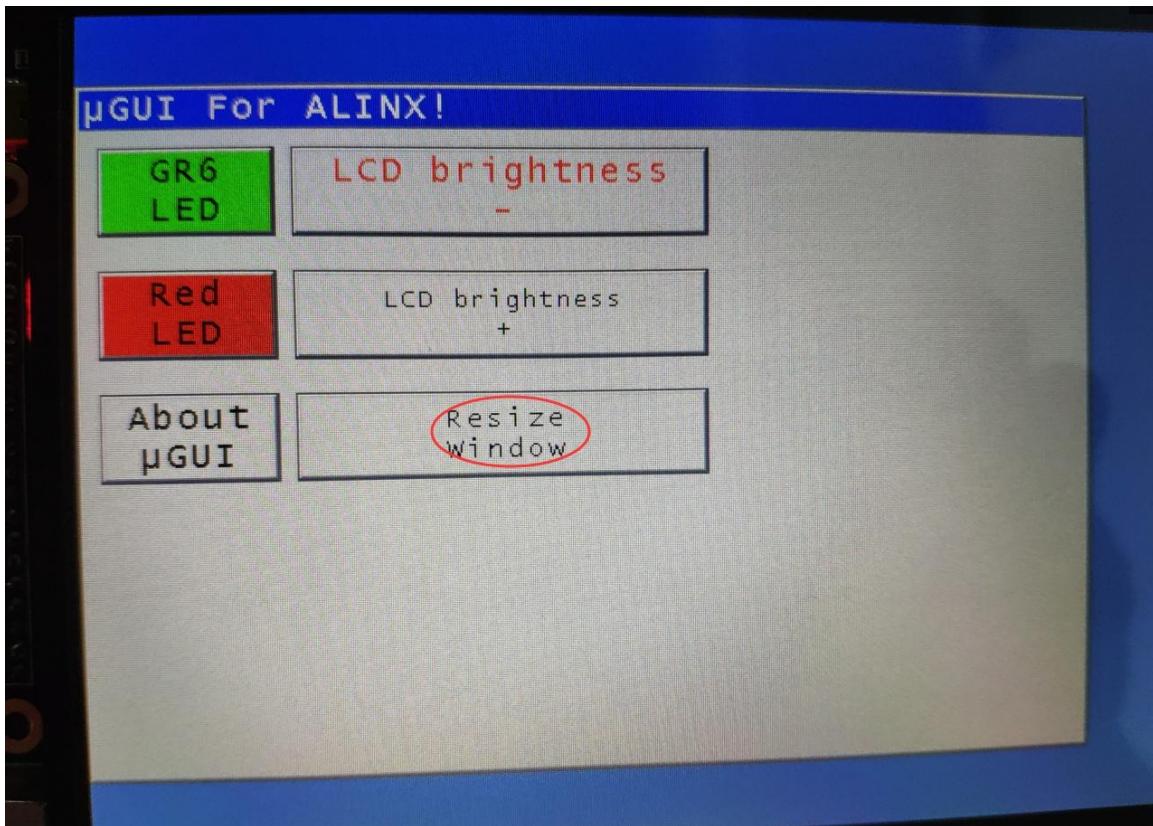
At this time, we can touch the keys on the touch screen by hand, such as touching the key of the green led. Each time the key is touched, the number on the key will increase by 1.



If we press the “LCD brightness+/-” key, you can change the LCD display brightness.



If you press “Resize Window”, the window will become smaller, and pressing it again will restore it.



# Part 35 Ethernet Transmission--ADC Acquisition Based on AN108 Module

The experimental Vivado project directory is "ad9280\_sg\_dma\_dp /vivado".

**The experiment vitis project directory is "ad9280\_lwip /vitis".**

This chapter takes the AN108 module as an example to introduce the data collected by the ADC to the host computer via Ethernet.

## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 35.1: Develop a Transmission Protocol

In order to reflect the flexibility of the host computer, based on UDP transmission, the following communication protocol is developed, which is included in the UDP packet.

- 1) Get board information
  - a) **Inquiry command** (5 bytes in total, sent by the host computer via Ethernet)

Number of bytes	1	4
Command information	Header	0x00000000 or 0x00010001

- b) **Answer command** (27 bytes in total, sent by the development board via Ethernet)

Number of bytes	Command Information
1	Header 0x01
4	0x00020001
6	Board MAC address

4	Board IP address
1	Sign bit 0x00, Unsigned number 0x01, Signed number is invalid. The host computer requires a signed number
1	ADC Valid data length, such as 12 bits for the AD9226, which is 12
1	The number of bytes that the ADC acquires once. This function is invalid. The host computer requires the ADC data bit width to be two bytes.
1	Sampling channel (this function is not implemented by the host computer)
4	The sampling rate, that is, the sampling frequency, is set to 65M in the program.
4	Cache ADC data length in bytes

## 2) Obtain data

### a) Control command (send data request by host computer)

Number of bytes	Command Information
1	Header
4	0x00010002
6	Board MAC address
4	Sampling channel (this function is not implemented)
4	Number of samples (16 bits of data collected, the number of samples is half of the length of the cached data)

### b) Answer command (sent by the development board)

Number of bytes	Command Information
1	Header 0x 01
4	0x00010002
1024	ADC data

Each UDP packet contains a header. In the first byte, the format is as follows:

Bit	Value (0)	Value (1)
bit 0	Query or control	Answer
bit1~bit7		Random data

**Note:** When answering, the high 7-bit random data remains unchanged, bit0 is set to 1

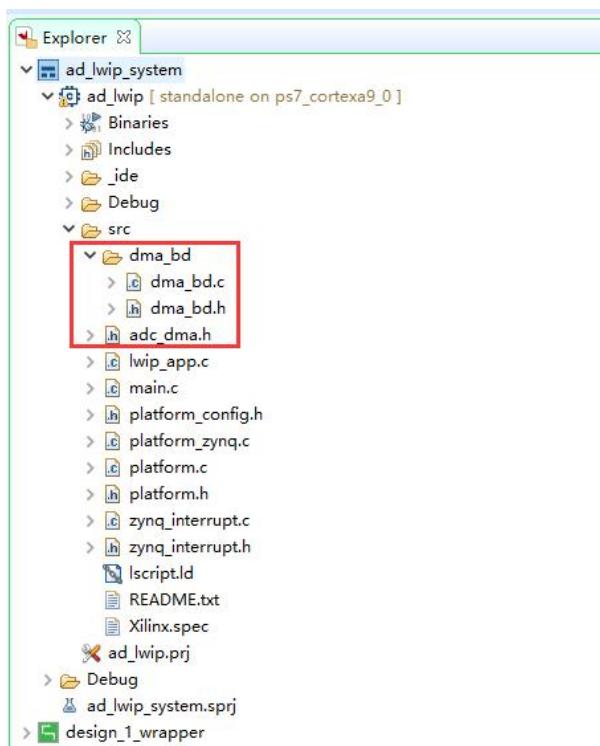
The workflow is:

- 1) The host computer sends an inquiry command
- 2) Development board answering inquiry
- 3) The host computer sends a control command request data
- 4) Development board sends data
- 5) Loop steps 3 and 4

## Part 35.2: Vitis program development

### Part 35.2.1: ADC acquisition part

- 1) The acquisition of ADC has been mentioned before, adding “dma\_bd” and “adc\_dma.h” to the “Vitis” in this chapter



- 2) In the “main” function of the “main.c” file, the interrupt is initialized, the “DMA” is initialized, "interrupt" is connected and BD linked list is created.

```

int main()
{
    init_platform();

    InterruptInit(INT_DEVICE_ID,&XScuGicInstance);
    /* Initialize DMA */
    XAxiDma_Initial(DMA_DEV_ID, S2MM_INTR_ID, &AxiDma, &XScuGicInstance) ;

    /* Interrupt register */
    InterruptConnect(&XScuGicInstance,S2MM_INTR_ID,Dma_Interrupt_Handler, &AxiDma,0,3);

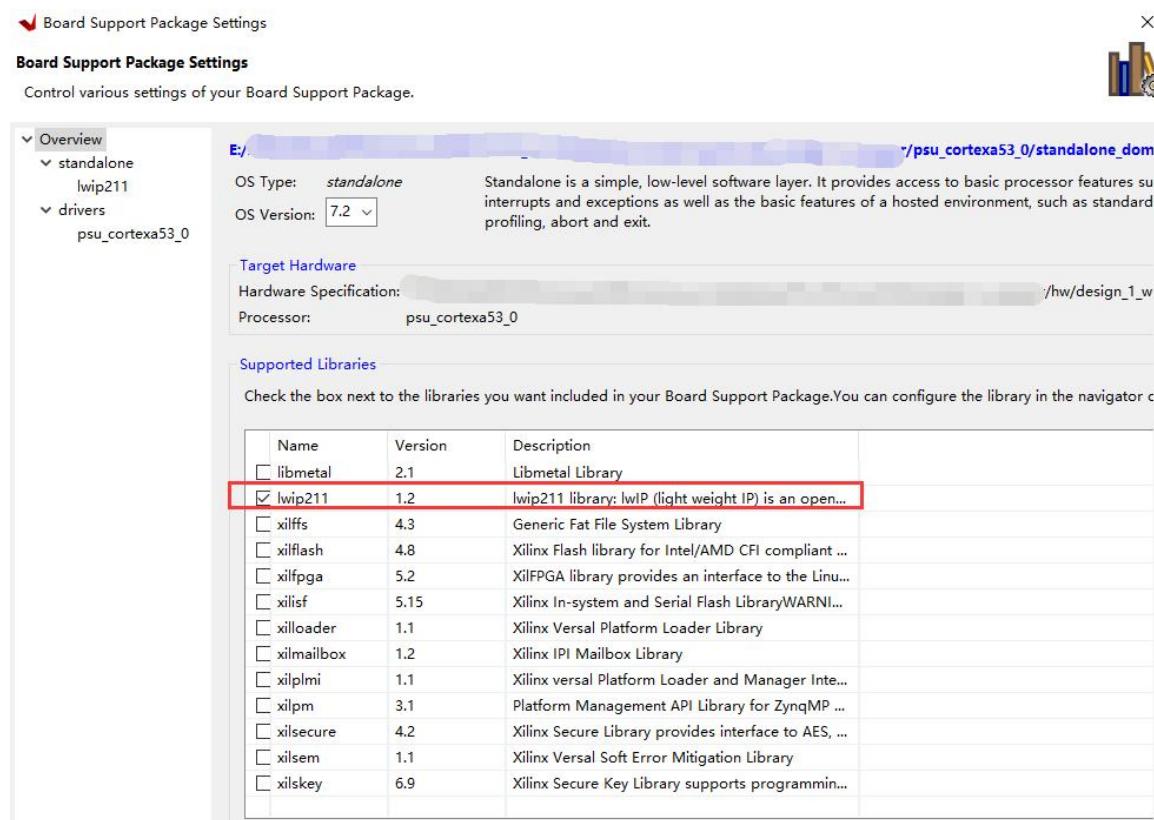
    /* Create BD chain */
    CreateBdChain(BdChainBuffer, BD_COUNT, ADC_SAMPLE_NUM, (unsigned char *)DmaRxBuffer, RXPATH) ;

    lwip_loop();
    cleanup_platform();
    return 0;
}

```

### Part 35.2.2: LWIP control section

- 1) In this experiment, you need to enable the lwip library and set it in the Board Support Package Setting



- 2) Configure “DHCP” “pbuf” settings

Name	Value	Default	Type	Description
api_mode	RAW API (RAW_API)	RAW_API	enum	Mode of operation for lwIP (I)
lwip_tcp_keepalive	false	false	boolean	Enable keepalive processing
no_sys_no_timers	true	true	boolean	Drops support for sys_timeo
socket_mode_thread_prio	2	2	integer	Priority of threads in socket r
use_axieth_on_zynq	1	1	integer	Option if set to 1 ensures axi
use_emaclite_on_zynq	1	1	integer	Option if set to 1 ensures em
> arp_options	true	true	boolean	ARP Options
> debug_options	true	true	boolean	Turn on lwIP Debug?
dhcp_options	true	true	boolean	Is DHCP required?
dhcp_does arp_check	true	false	boolean	ARP check on offered addres
lwip_dhcp	true	false	boolean	Is DHCP required?
> icmp_options	true	true	boolean	ICMP Options
igmp_options	false	false	boolean	IGMP Options
> lwip_ip_options	true	true	boolean	IP Options
ipv6_enable	false	false	boolean	IPv6 enable value
> lwip_memory_options				Options controlling lwIP mem
> mbox_options	true	true	boolean	Mbox Options
pbuff_options	true	true	boolean	Pbuff Options
pbuff_link_hlen	16	16	integer	Number of bytes that should
pbuff_pool_bufsize	2000	1700	integer	Size of each pbuf in pbuf po
pbuff_pool_size	2048	256	integer	Number of buffers in pbuf po
> stats_options	true	true	boolean	Turn on lwIP statistics?
> tcp_options	true	true	boolean	Is TCP required ?
> temac_adapter_options	true	true	boolean	Settings for xps-ll-temac/Axi
> udp_options	true	true	boolean	Is UDP required ?

- 3) In the lwip\_app.c file, start\_udp is used for udp initialization, which has been mentioned before

```
int start_udp(unsigned int port) {
    err_t err;
    udp8080_pcb = udp_new();
    if (!udp8080_pcb) {
        xil_printf("Error creating PCB. Out of Memory\n\r");
        return -1;
    }
}
```

- 4) The udp\_receive function is a callback function for receiving. In this function, the command sent by the host computer is judged. For the format, please refer to the transmission protocol established earlier.

```
void udp_receive(void *arg, struct udp_pcb *pcb, struct pbuf *p_rx,
                 struct ip_addr *addr, u16_t port) {
```

- 5) The transfer\_data function is used to answer the commands of the host computer, in udp\_receive

```
int transfer_data(const char *pData, int len, struct ip_addr *addr)
{
```

- 6) The send\_adc\_data function is used to send ADC data to the host computer. The first 5 bytes are TargetHeader, which can refer to

the transmission protocol.

```

int send_adc_data(const char *frame, int data_len)
{
    if (!TargetHeader[0]) {
        return -1;
    }
    struct pbuf *q;
    q = pbuf_alloc(PBUF_TRANSPORT, 8 + data_len, PBUF_POOL);
    if (!q) {
        xil_printf("pbuf_alloc %d fail\n\r", data_len + 8);
        return -3;
    }

    memcpy(q->payload, TargetHeader, 5);
    for (int i = 0; i < data_len; i += 2) {
        ((char *) q->payload)[5 + i] = frame[i + 1];
        ((char *) q->payload)[6 + i] = frame[i + 0];
    };
    q->len = q->tot_len = 5 + data_len;
    udp_sendto(udp8080_pcb, q, &target_addr, 8080);
    pbuf_free(q);
    return 0;
}

```

- 7) In the lwip\_loop function, start UDP transmission. Since the host computer can only display one channel of ADC data, only the CH1 channel is opened in this experiment. In the while loop, judge whether the ADC data collection is completed, and then sub-packet transmission

```

/* start the application (web server, pxtest, txtest, etc..) */
start_udp(8080);

XAxiDma_Adc(Ch1BdChainBuffer, AD9226_CH1_BASE, ADC_SAMPLE_NUM, BD_COUNT, &AxiDmaCh1) ;
/* receive and process packets */
while (1)
{
    xemacif_input(echo_netif);
    /* Wait for times */
    usleep(1000);
    /* Check current frame length */
    if (FrameLengthCurr > 0)
    {
        /* Check if DMA completed */
        if (ch0_s2mm_flag >= 0)
        {
            ...
        }
    }
}

```

- 8) It should be noted that the cache size set by the host computer is 1MB, because the host computer is fixed to the data bit width of two bytes and unsigned bits. Therefore, in adc\_dma.h, set the number of acquisitions of AD9280 to 1024\*512, and set ADC\_BYTE to 2

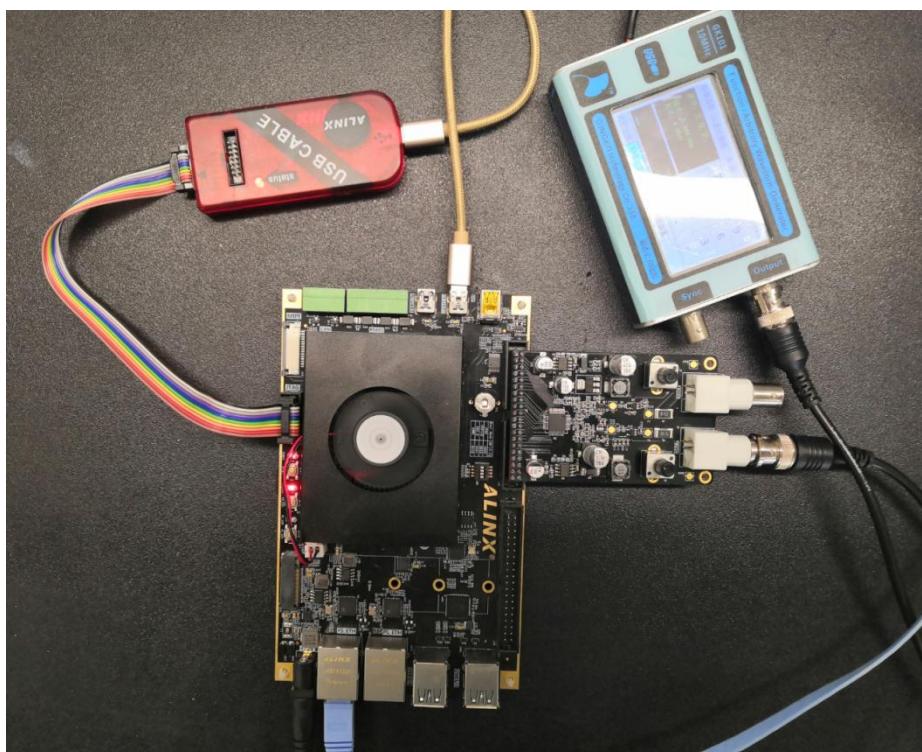
```
#define ADC_SAMPLE_NUM (1024*512)
```

And convert the data into two bytes, a signed number. It is implemented in the while loop of lwip\_app.c.

```
while (1)
{
    xemacif_input(echo_netif);
    /* Wait for times */
    usleep(1000) ;
    /* Check current frame length */
    if (FrameLengthCurr > 0)
    {
        /* Check if DMA completed */
        if (s2mm_flag >= 0)
        {
            int udp_len;
            Xil_DCacheInvalidateRange((u32) DmaRxBuffer, FrameLengthCurr);
            for(int i = 0 ; i < ADC_SAMPLE_NUM ; i++)
            {
                DmaBufferTmp[i] = (short)(DmaRxBuffer[i] - 128) ;
            }
        }
    }
}
```

### Part 35.3: Onboard Verification

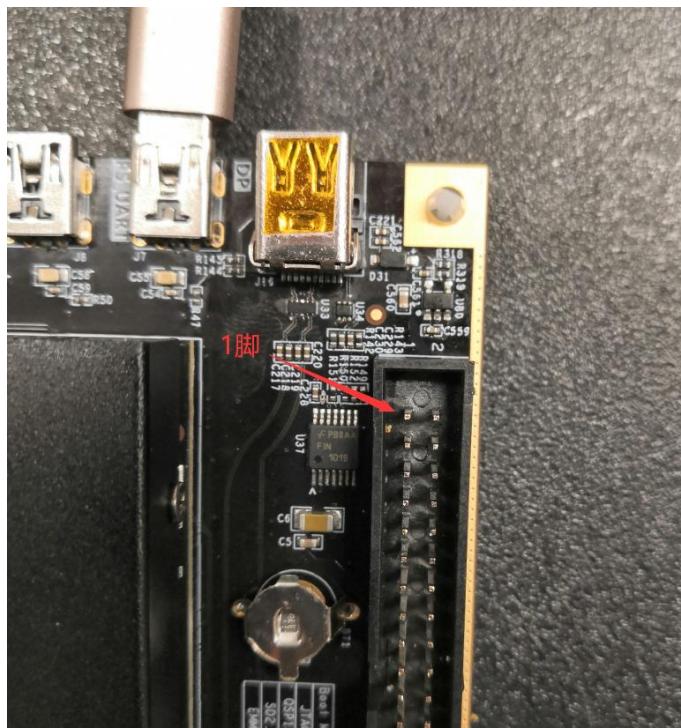
- 1) Connect the development board as shown below, you need to ensure that the PC network card is a Gigabit network card, otherwise it will not be displayed because the network speed is too low. Insert the AN108 module into the expansion port and connect a dedicated shield to the waveform generator. In order to observe the display effect, the waveform generator sampling frequency can be set from 1KHz to 1MHz, and the voltage amplitude is up to 10V.



Hardware Connection (Expansion Port J46)



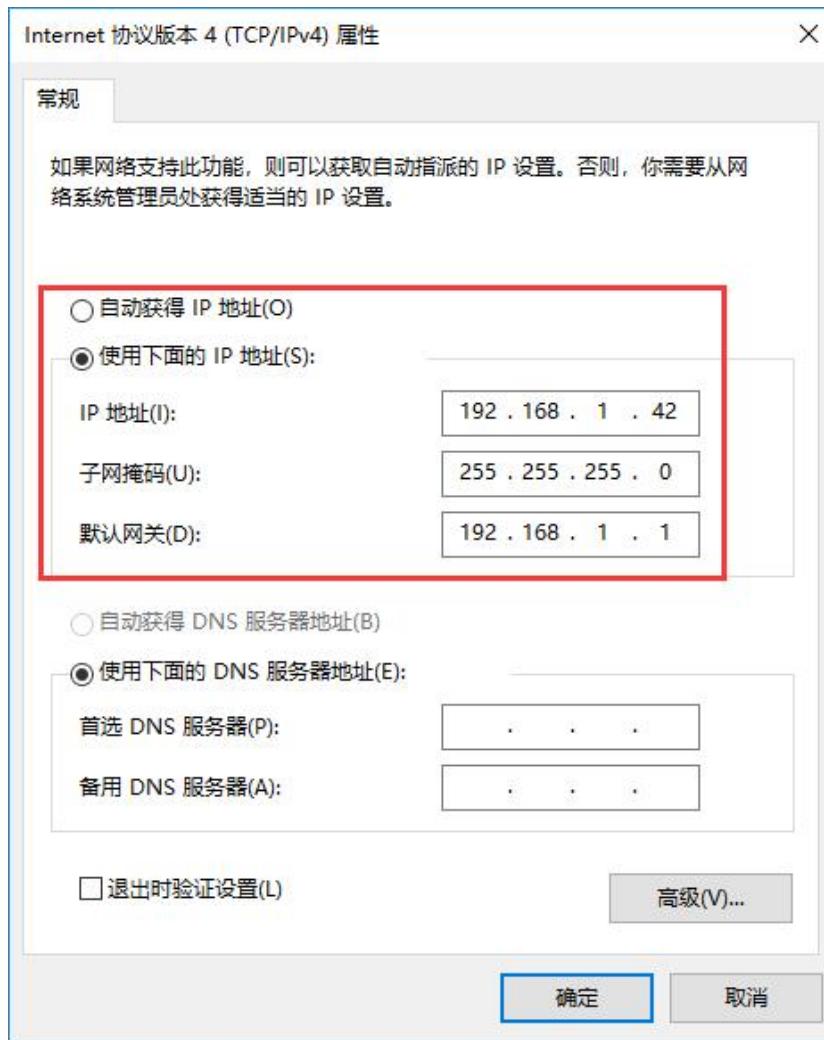
Noted that PIN 1 Alignment



- 2) If there is a DHCP server, the IP will be automatically assigned to the development board; if there is no DHCP server, the default FPGA development board IP address is 192.168.1.11, and the IP address of the PC needs to be set to the same network segment, as shown in the following figure. Also make sure that there is no IP address of 192.168.1.11 in the network, otherwise IP conflict will

occur and the image will not be displayed. You can enter ping 192.168.1.11 in the CMD to check whether the ping can be pinged before the board is powered on. If the ping is successful, the IP address in the network cannot be verified.

Open the putty software after no problem.



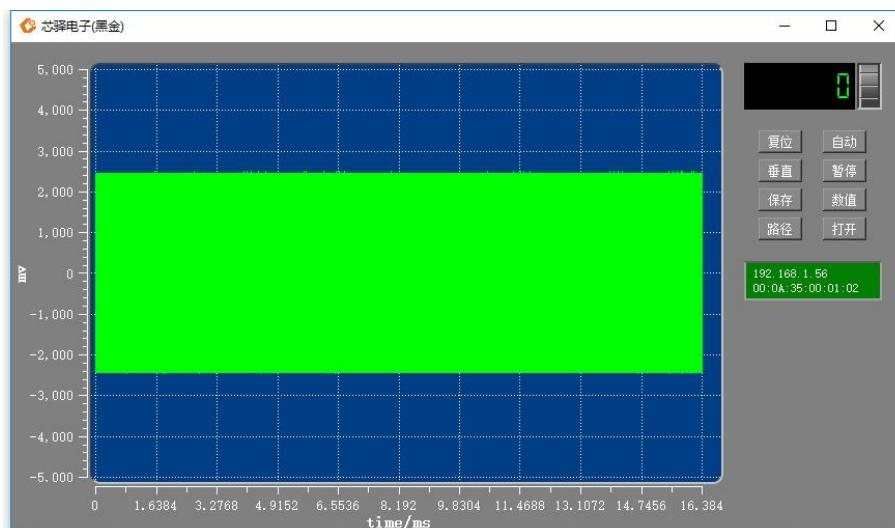
- 3) Download the program to the FPGA development board, you can see the print information in putty

```
-----AN108 lwIP UDP DEMO -----
UDP packets sent to port 8080
WARNING: Not a Marvell or TI Ethernet PHY. Please verify the initialization sequence
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 1: 1000
Board IP: 192.168.1.56
Netmask : 255.255.255.0
Gateway : 192.168.1.1
```

4) In the project directory, open the “oscilloscope .exe”

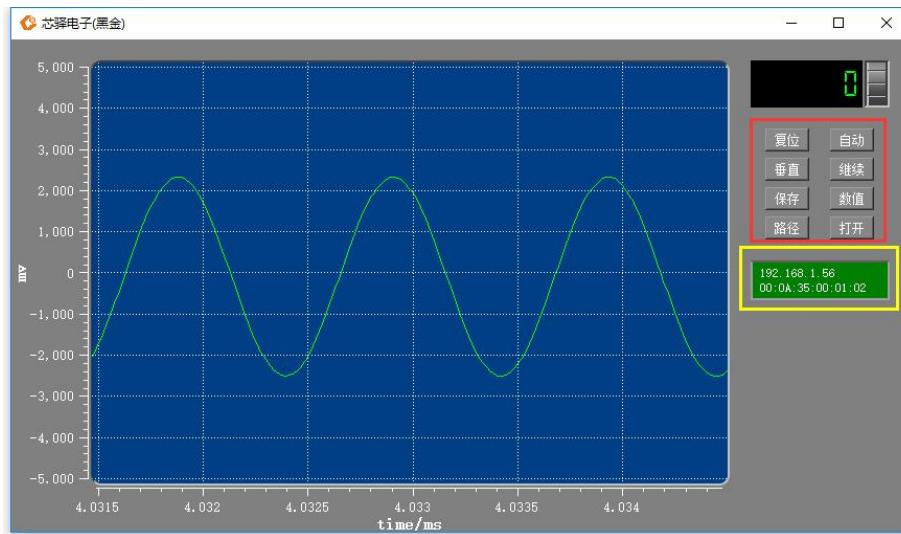
.Xil	2018/9/14 11:30	文件夹
ad9280_lwip.cache	2018/9/14 11:05	文件夹
ad9280_lwip.hw	2018/9/14 11:05	文件夹
ad9280_lwip.ip_user_files	2018/9/14 11:05	文件夹
ad9280_lwip.runs	2018/9/14 11:05	文件夹
ad9280_lwip.sdk	2018/9/14 11:10	文件夹
ad9280_lwip.sim	2018/9/14 11:05	文件夹
ad9280_lwip.srcs	2018/9/14 11:05	文件夹
ad9280_lwip.tmp	2018/9/14 11:31	文件夹
repo	2018/9/11 11:54	文件夹
ad9280_lwip.xpr	2018/8/16 11:18	Vivado Project Fi... 27 KB
示波器.exe	2018/5/14 16:36	应用程序 17,903 KB

5) The results are as follows



For the use of the host computer software, please refer to the section of the host computer software instructions for AN108 Ethernet transmission.

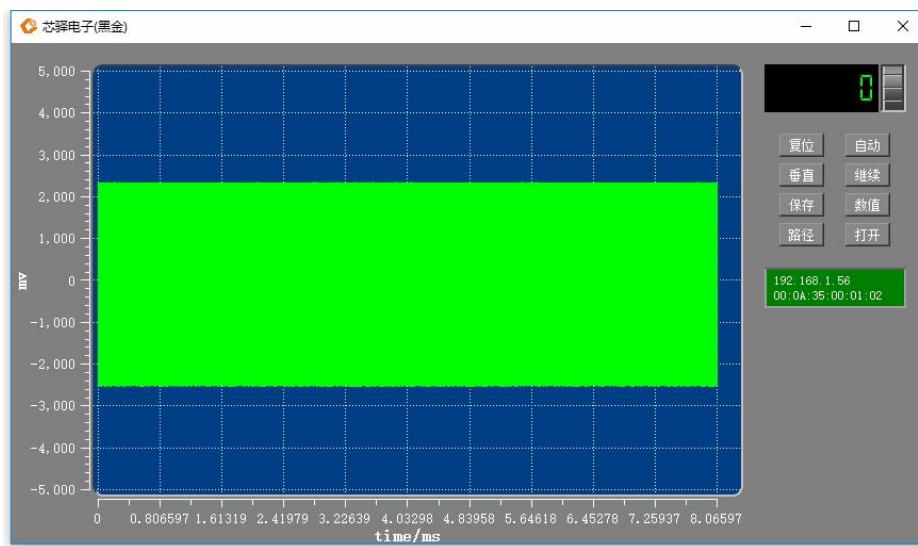
## Part 35.4: PC software instructions



The yellow box shows the “MAC” and “IP” address of the transmitting FPGA development board. If the background turns red, the network connection is broken or the data is lost.

The red box is the control key and functions as follows:

**复位:** Click Reset to display the waveform to the initial state, as shown below.



**自动:** useless

**垂直:** Click this key to switch between horizontal and vertical zoom. In the vertical state, scroll the mouse scroll to zoom in the vertical direction. In the horizontal state, the mouse scroll can be

scrolled horizontally. 暂停: “暂停” and “继续” to switch, click “暂停” to pause the waveform, and then click “继续” to display the waveform.

**保存:** Save the ADC data as a TXT file. The save path is set at the “Path” key. The default is the path where the software is located.

**数值:** “数值” and “电压” are switched. The Y-direction coordinate unit is the original value, that is, the received raw data value. Click “电压” to display the voltage value.

**路径:** Select save path

**打开:** Open saved TXT waveform file

## Part 35.5: Data Saving Demos

After clicking the save key, a “TXT” file will appear in the saved path of the settings.

.Xil	2018/9/13 16:55	文件夹
ad9226_lwip.cache	2018/9/13 16:54	文件夹
ad9226_lwip.hw	2018/9/13 16:54	文件夹
ad9226_lwip.ip_user_files	2018/9/13 16:54	文件夹
ad9226_lwip.runs	2018/9/13 16:54	文件夹
ad9226_lwip.sdk	2018/9/13 16:54	文件夹
ad9226_lwip.sim	2018/9/13 16:54	文件夹
ad9226_lwip.srcs	2018/9/13 16:54	文件夹
repo	2018/9/11 11:54	文件夹
2018-09-13 20_17_18_00.txt	2018/9/13 20:17	文本文档 2,592 KB
ad9226_lwip.xpr	2018/9/13 15:46	Vivado Project Fi... 43 KB
示波器.exe	2018/5/14 16:36	应用程序 17,903 KB

Open the file to see the original ADC data

```
2018-09-13 20_17_18_00.txt
1 b1d3 0184 012a 00d4 0079 0016 ffbd ff5e ff00 fea9 fe53 fdf9 fd8e fd62 fd1b fcd9
2 fc6a fc76 fc4e fc29 fc13 fc02 rbf9 fbf8 fc05 fc1a fc31 fc57 fc83 fc87 fcfc7 fcfc2 fc3d
3 fd7e fdc3 fe16 fe6f fec6 ffle ff7d ffd5 0035 0091 00ed 0144 019b 01ec 0235 0280
4 02c1 02fa 0331 035b 037b 039a 03ab 03b7 03b9 03b1 03a2 0385 036d 0340 030d 02db
5 0295 0254 020a 01b5 0166 010f 00b3 0058 fff8 ff9d ff3f fedf fe8c fe36 fddd fd97
6 fd49 fd09 fc99 fc67 fc43 fc20 fc00 fbfb fbfe fc0b fc23 fc3b fc65 fc96
7 fcfa fd07 fd4a fd95 fd82 fe8a feeo ff3f ff9b ff99 0054 00b3 010a 0160 01b7
8 0206 0253 0295 02d7 030c 033c 0368 0385 03a3 03b0 03b7 03ba 03af 039a 037f 0358
9 0331 02fd 02bf 0285 0239 01ef 019f 0148 00f1 008f 0039 ffdd5 ff7d ff23 fec5 fe6f
10 fe12 fdc5 fd7d fd31 fcfc3 fcba fc83 fc58 fc39 fc18 fc06 fbf8 fbf9 fc01 fc12 fc2a
11 fc4c fc74 fcac fdcc fd1d fd66 fdac fdfe fe51 fea7 fe01 ff5e ffba 0018 0073 00d0
12 0129 017f 01d2 0220 0269 02e8 031c 034c 0372 0393 03a7 03b2 03b9 03bb 03a6
13 0394 0374 034d 0321 02ea 02ad 026a 0225 01d0 0180 0128 00d0 0073 001a ffbc ff5d
14 ff00 fea7 fe4f fdff fd8a fd62 fd1d fd9d fea7 fc75 fc4b fc2c fc11 fc01 fbf8 fbf9
15 fc08 fc1a fc38 fc57 fc83 fcba fcfc1 fd31 fd7d fdc5 fe17 fe72 fec5 ff20 ff7e ffd8
16 0037 0094 00ef 0147 019c 01eb 023a 0280 02be 02fe 0331 035b 0380 039a 03ad 03b7
17 03b7 03b2 03a1 038a 036b 033d 030d 02d8 0297 0254 0207 01b5 0164 010d 00b2 0055
18 ff88 ff9a feee fe87 fe31 ffde2 fd90 fd49 fd07 ffcc9 fc95 fc65 fc40 fc21 fc0b
19 fbf7 fbf7 fc00 fc0f fc21 fc41 fc66 fc98 ffce fd07 fd4b fd95 fdel fe35 fe8b feee4
20 ff3f ff9a fffd 0054 00b0 010e 0167 01b3 0208 0253 0298 02d5 030b 033d 0367 0389
21 03a1 03ae 03b8 03b8 03ab 039d 037e 035b 032e 02fd 02c3 0280 023a 01ed 019e 0148
22 00ef 0092 0037 ffdb ff7c ff1d fec5 fe69 fe17 fd66 fd78 fd2e fcf2 fcb6 fc85 fc57
23 fc34 fc18 fc02 fbf8 fbf9 fc00 fc13 fc2b fc4a fc78 fc7 fcel fd1e fd64 fdaf fdff
24 fe52 fea8 ff04 ff61 ffb8 0019 0076 00d4 012b 017e 01d3 0222 0266 02af 02e9 031e
25 034e 0372 039a 03a8 03b3 03b9 03a7 0395 0373 034c 0320 02e7 02ab 026b 021f
26 01d4 017b 012b 00d0 0073 0018 ffbe ff5b ff04 fea3 fe4e fdff fda9 fd61 fd19 fcd9
```

# Part 36: Ethernet Transmission--ADC Acquisition Based on AN9238 Module

The experimental Vivado project directory is "ad9238\_sg\_dma\_dp /vivado".

**The experiment vitis project directory is "ad9238\_lwip /vitis".**

This chapter takes the AN9238 module as an example to introduce the data collected by ADC to be transmitted to the host computer via Ethernet. For Develop a transmission protocol, please refer to **Part 35.1**

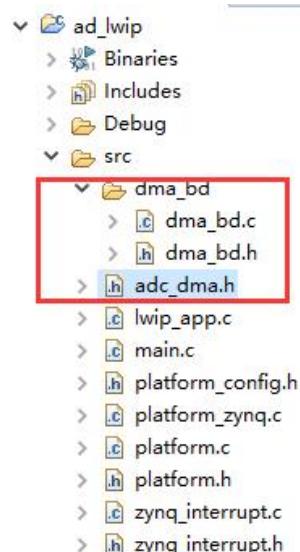
## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 36.1: Vitis Program Development

#### Part 36.1.1: ADC Acquisition Part

- 1) The acquisition of “ADC” has been mentioned before, adding “dma\_bd” and “adc\_dma.h” to the “Vitis” in this chapter.



- 2) In the “main” function of the “main.c” file, the interrupt is initialized, the “DMA” is initialized, "interrupt" is connected and BD linked list is created.

```

int main()
{
    init_platform();

    InterruptInit(INT_DEVICE_ID,&XScuGicInstance);
    /* Initialize DMA */
    XAxiDma_Initial(DMA_DEV_ID, S2MM_INTR_ID, &AxiDma, &XScuGicInstance) ;

    /* Interrupt register */
    InterruptConnect(&XScuGicInstance,S2MM_INTR_ID,Dma_Interrupt_Handler, &AxiDma,0,3);

    /* Create BD chain */
    CreateBdChain(BdChainBuffer, BD_COUNT, ADC_BYTE*ADC_CH_COUNT*ADC_SAMPLE_NUM, (unsigned char *)DmaRxBuffer, RXPATH) ;

    lwip_loop();
    cleanup_platform();
    return 0;
}

```

### Part 36.1.2: LWIP Control Section

For the LWIP control part, please refer to AN108 Ethernet transmission, which is basically the same.

It should be noted that the cache size set by the host computer is 1MB. Because the sampling speed of AD 9238 is relatively high, for the display effect of the host computer, set the number of samples in adc\_dma.h to  $1024 * 512$

```
#define ADC_SAMPLE_NUM (1024*512)
```

In the lwip\_app.c file, only the data of CH0 is taken.

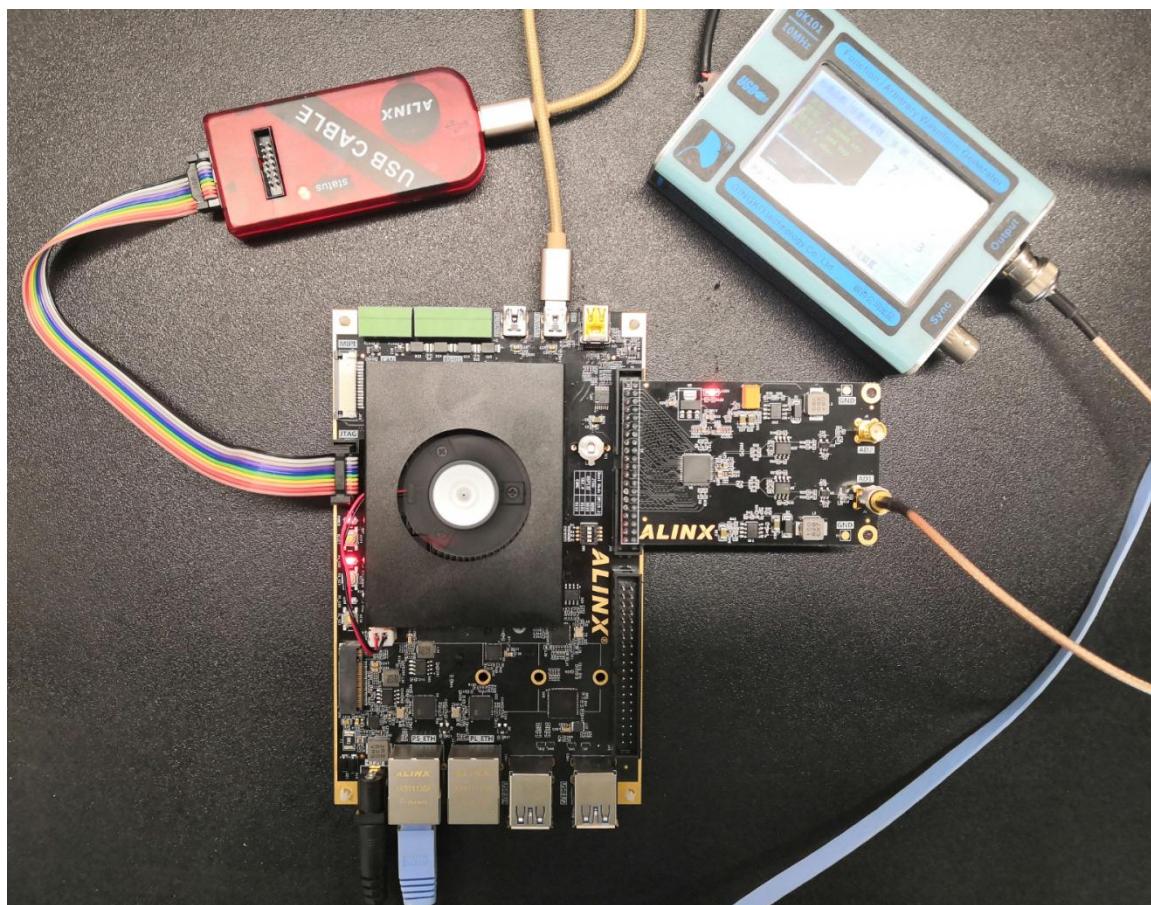
```

XAxiDma_Adc(Ch0BdChainBuffer, AD9238_CH0_BASE, ADC_SAMPLE_NUM, BD_COUNT, &AxiDmaCh0) ;
/* receive and process packets */
while (1)
{
    xemacif_input(echo_netif);
    /* Wait for times */
    usleep(1000) ;
    /* Check current frame length */
    if (FrameLengthCurr > 0)
    {
        /* Check if DMA completed */
        if (ch0_s2mm_flag >= 0)
        {
            int udp_len;
            Xil_DCacheInvalidateRange((u32) CH0DmaRxBuffer, FrameLengthCurr);
            /* change data to short */
            for(int i = 0 ; i < ADC_SAMPLE_NUM ; i++)
            {
                CH0DmaBufferTmp[i] = (short)(CH0DmaRxBuffer[i] - 2048) ;
            }
        }
    }
}

```

## Part 36.2: Onboard Verification

- 1) Connect the development board as shown below, you need to ensure that the PC network card is a **Gigabit network card**, otherwise it will not be displayed because the network speed is too low. Insert the AN9238 module into the expansion port and connect the SMA interface to the waveform generator. In order to observe the display effect, the waveform generator sampling frequency can be set from 1KHz to 5MHz, and the voltage amplitude is up to 10V.

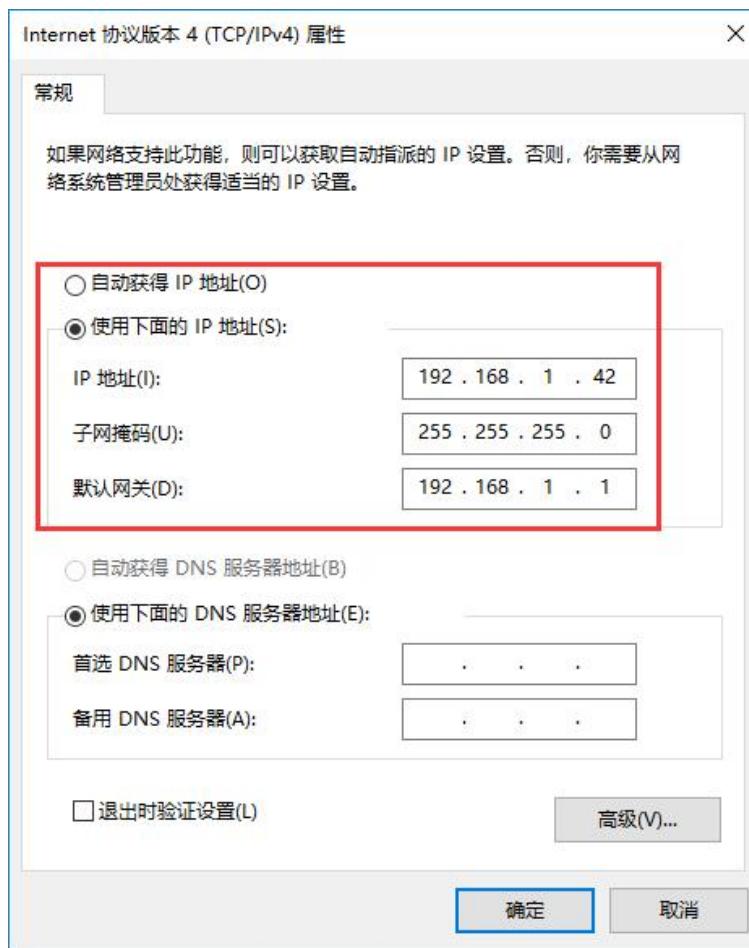


Hardware Connection (Port J46)

- 2) If there is a DHCP server, the IP will be automatically assigned to the development board; if there is no DHCP server, the default FPGA development board IP address is 192.168.1.11, and the IP address of the PC needs to be set to the same network segment,

as shown in the following figure. Also make sure that there is no IP address of 192.168.1.11 in the network, otherwise IP conflict will occur and the image will not be displayed. You can enter ping 192.168.1.11 in the CMD to check whether the ping can be pinged before the board is powered on. If the ping is successful, the IP address in the network cannot be verified.

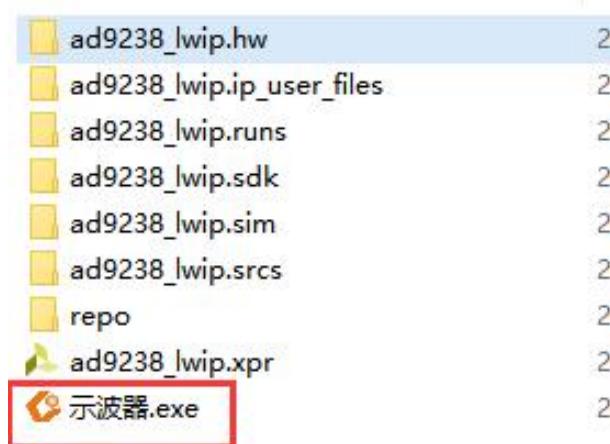
Open the putty software after no problem.



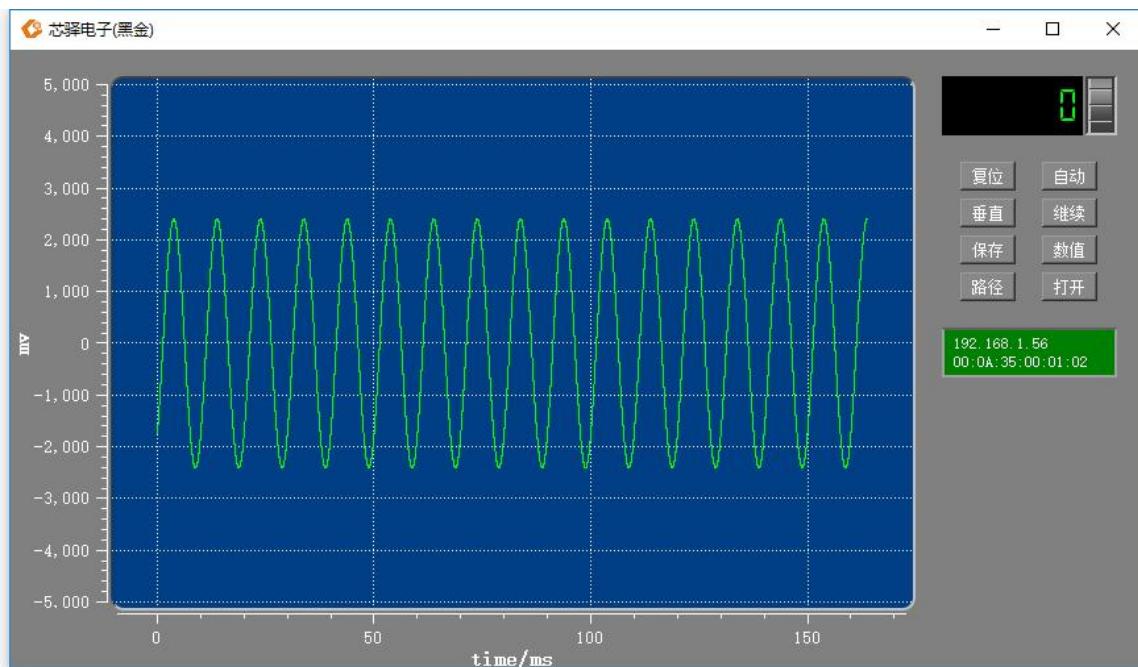
- 3) Download the program to the FPGA development board, you can see the print information in putty

```
-----AN9238 lwIP UDP DEMO -----  
UDP packets sent to port 8080  
WARNING: Not a Marvell or TI Ethernet PHY. Please verify the initialization sequence  
Start PHY autonegotiation  
Waiting for PHY to complete autonegotiation.  
autonegotiation complete  
link speed for phy address 1: 1000  
Board IP: 192.168.1.135  
Netmask : 255.255.255.0  
Gateway : 192.168.1.1
```

- 4) In the project directory, open the “oscilloscope .exe”



- 5) The results are as follows



For PC software instructions refer to **Part 35.4**.

# Part 37: Ethernet Transmission--ADC Acquisition Based on AN706 Module

The experimental Vivado project directory is "ad7606\_sg\_dma\_dp /vivado".

**The experiment vitis project directory is "ad7606\_lwip /vitis".**

This chapter takes the AN706 module as an example to introduce the data collected by the ADC to the host computer via Ethernet. For Develop a transmission protocol, please refer to **Part 35.1**

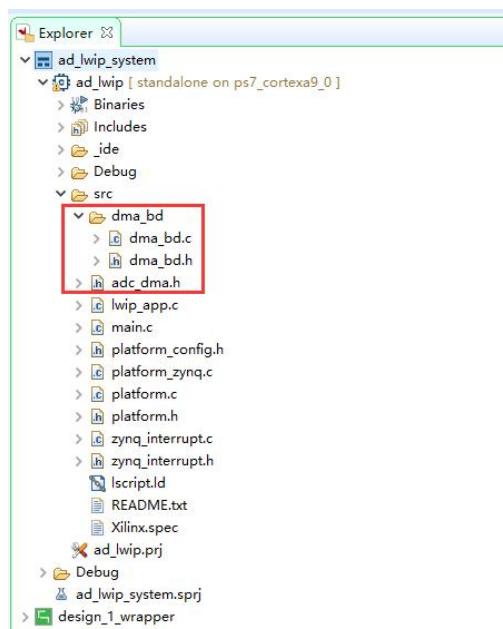
## Software Engineer Job Content

The following is the content that FPGA engineers are responsible for.

### Part 37.1: Vitis Program Development

#### Part 37.1.1: ADC Acquisition Part

The acquisition of ADC has been mentioned before, adding "dma\_bd" and "adc\_dma.h" to the "Vitis" in this chapter.



In the “main” function of the “main.c” file, the interrupt is initialized, the “DMA” is initialized, "interrupt" is connected and BD linked list is created.

```
@int main()
{
    init_platform();

    InterruptInit(INT_DEVICE_ID,&XScuGicInstance);
    /* Initialize DMA */
    XAxiDma_Initial(DMA_DEV_ID, S2MM_INTR_ID, &AxiDma, &XScuGicInstance) ;

    /* Interrupt register */
    InterruptConnect(&XScuGicInstance,S2MM_INTR_ID,Dma_Interrupt_Handler, &AxiDma,0,3);

    /* Create BD chain */
    CreateBdChain(BdChainBuffer, BD_COUNT, ADC_BYTE*ADC_CH_COUNT*ADC_SAMPLE_NUM, (unsigned char *)DmaRxBuffer, RXPATH) ;

    lwip_loop();
    cleanup_platform();
    return 0;
}
```

### Part 37.1.2: LWIP Control Section

For the LWIP control part, please refer to the AN926 Ethernet transmission, which is basically the same.

It should be noted that the cache size set by the host computer is “1MB”. Because the sampling speed of the “AD7606” is relatively low. In order to display the effect of the host computer, set the number of samples in “adc\_dma.h” to “1024\*32”.

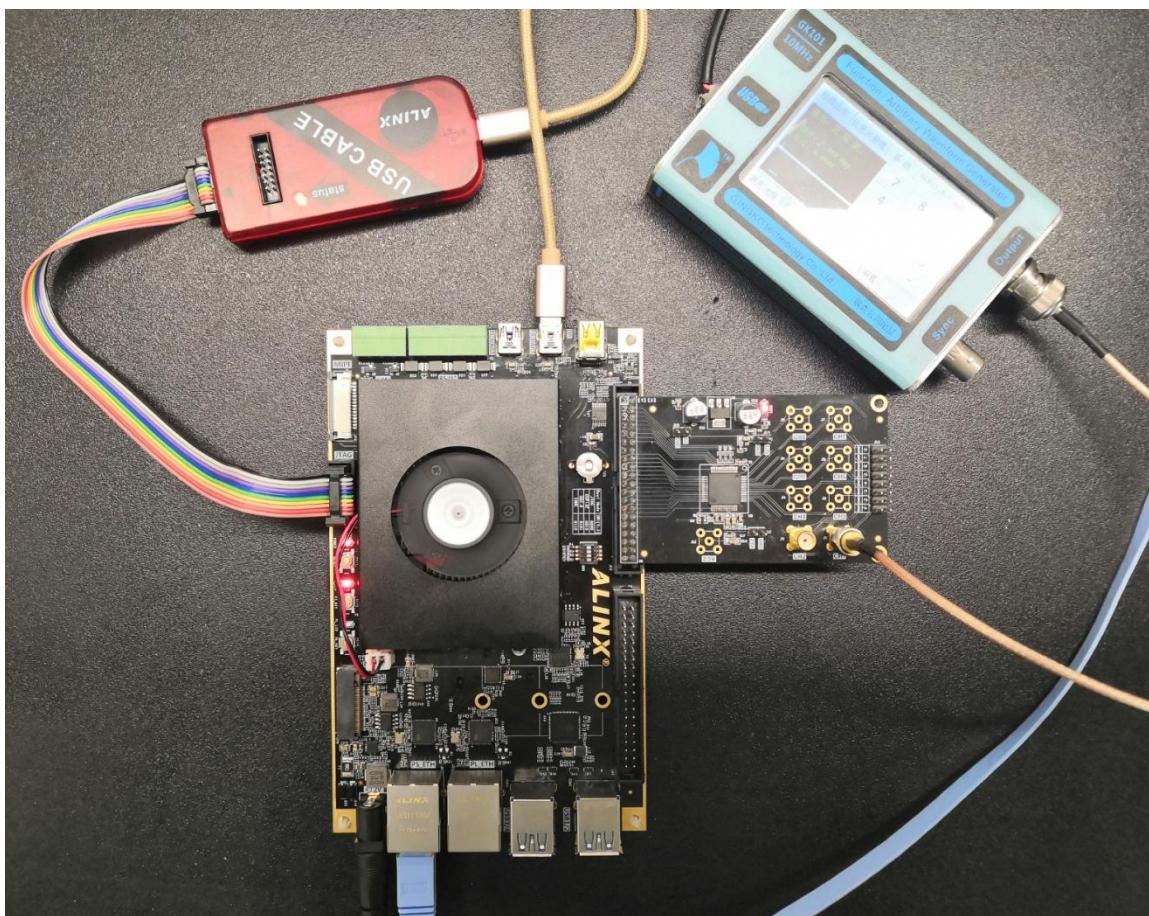
```
#define ADC_SAMPLE_NUM (1024*32)
```

In the “while” loop of the “lwip\_app.c” file, only the CH1 data is taken.

```
while (1)
{
    xemacif_input(echo_netif);
    /* Wait for times */
    usleep(1000) ;
    /* Check current frame length */
    if (FrameLengthCurr > 0)
    {
        /* Check if DMA completed */
        if (s2mm_flag >= 0)
        {
            int udp_len;
            Xil_DCacheInvalidateRange((u32) DmaRxBuffer, ADC_BYTE*ADC_CH_COUNT*ADC_SAMPLE_NUM);
            for(int i = 0 ; i < ADC_SAMPLE_NUM ; i++)
            {
                DmaRxBufferTmp[i] = DmaRxBuffer[8*i] ;
            }
        }
    }
}
```

## Part 37.2: Onboard Verification

- 1) Connect the development board as shown below, you need to ensure that the PC network card is a Gigabit network card, otherwise it will not be displayed because the network speed is too low. Insert the AN706 module into the expansion port and connect the SMA interface to the waveform generator. In order to observe the display effect, the waveform generator sampling frequency can be set from 50Hz to 1Hz, and the voltage amplitude is up to 10V

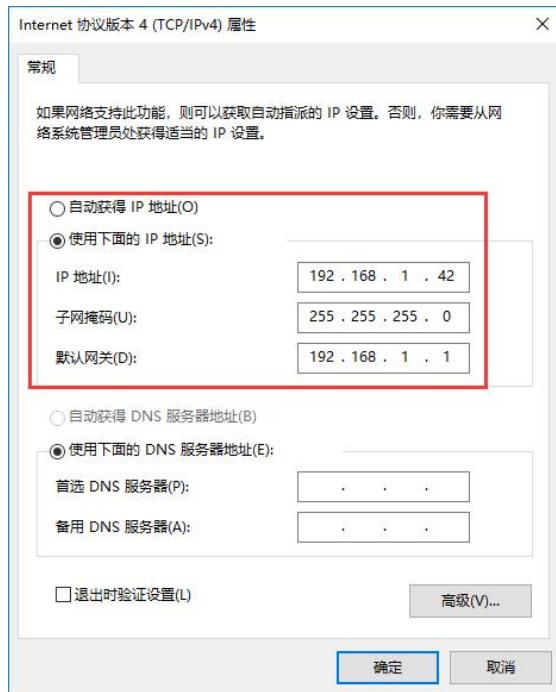


Hardware Connection (Expansion Port J46)

- 2) If there is a DHCP server, the IP will be automatically assigned to the development board; if there is no DHCP server, the default FPGA development board IP address is 192.168.1.11, and the IP address of the PC needs to be set to the same network segment, as shown in the following figure. Also make sure that there is no IP

address of 192.168.1.11 in the network, otherwise IP conflict will occur and the image will not be displayed. You can enter ping 192.168.1.11 in the CMD to check whether the ping can be pinged before the board is powered on. If the ping is successful, the IP address in the network cannot be verified.

Open the putty software after no problem.



- 3) Download the program to the FPGA development board, you can see the print information in putty

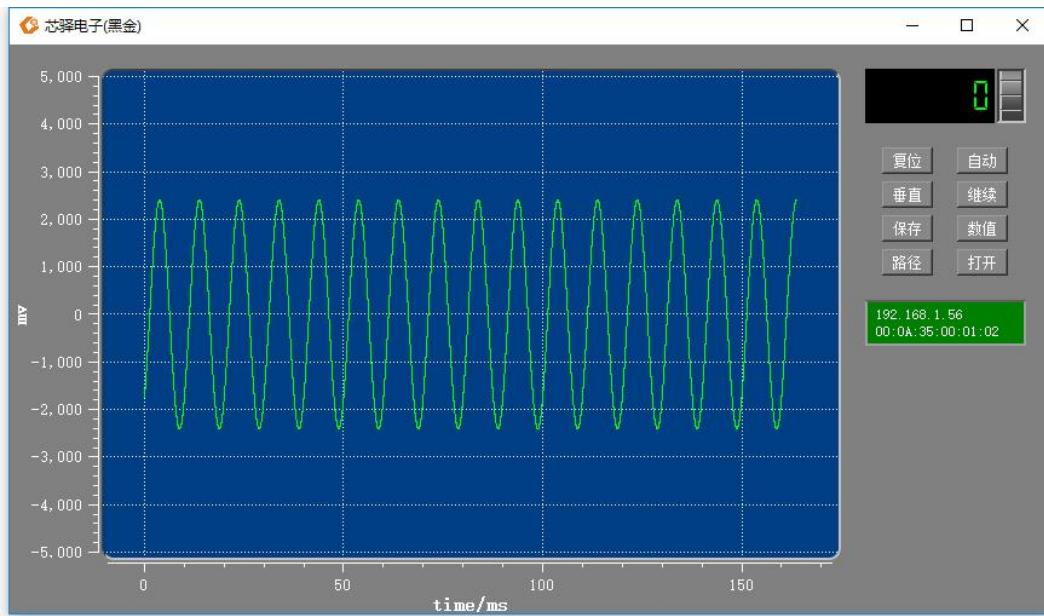
A screenshot of a PuTTY terminal window titled "COM4 - PuTTY". The window displays the following text:

```
-----AN706 lwIP UDP DEMO -----  
UDP packets sent to port 8080  
WARNING: Not a Marvell or TI Ethernet PHY. Please verify the initialization sequence  
Start PHY autonegotiation  
Waiting for PHY to complete autonegotiation.  
autonegotiation complete  
link speed for phy address 1: 1000  
Board IP: 192.168.1.56  
Netmask : 255.255.255.0  
Gateway : 192.168.1.1
```

- 4) In the project directory, open the “oscilloscope .exe”

.Xil	2018/9/14 8:29	文件夹
ad7606_lwip.cache	2018/9/14 8:28	文件夹
ad7606_lwip.hw	2018/9/14 8:28	文件夹
ad7606_lwip.ip_user_files	2018/9/14 8:28	文件夹
ad7606_lwip.runs	2018/9/14 8:28	文件夹
ad7606_lwip.sdk	2018/9/14 8:30	文件夹
ad7606_lwip.sim	2018/9/14 8:28	文件夹
ad7606_lwip.srcs	2018/9/14 8:28	文件夹
repo	2018/9/11 11:54	文件夹
ad7606_lwip.xpr	2018/8/16 11:17	Vivado Project Fi...
示波器.exe	2018/5/14 16:36	应用程序 17,903 KB

- 5) The results are as follows



For PC software instructions refer to **Part 35.4.**