

ZYNQ MPSoC Development Platform

Vitis Application Tutorial



Version Record

Version	Date	Release By	Description
Rev1.03	2021-04-17	Rachel Zhou	First Release

We promise that this tutorial is not a permanent, consistent document. We will continue to revise and optimize the tutorial based on the feedback of the forum and the actual development experience.

Content

Version Record.....	2
Content.....	3
Preparation and precautions.....	11
Software Environment.....	11
Hardware environment.....	11
Batch Download QSPI Flash.....	11
Batch process to build Vitis project.....	12
PS side Peripherals Parts.....	15
Part 1: Experience ARM, bare metal output "Hello World".....	16
Part 1.1: Hardware Introduction.....	16
FPGA Engineer Job Content.....	17
Part 1.2: Create a Vivado Project.....	17
Part 1.2.1: Low Speed Configuration.....	19
Part 1.2.2: High Speed Configuration.....	22
Part 1.2.3: Clock Configuration.....	24
Part 1.2.4: DDR Configuration.....	26
Software Engineer Job Content.....	33
Part 1.3: Vitis Debugging.....	33
Part 1.3.1: Create Application Project.....	33
Part 1.4: Curing Program.....	47
Part 1.4.1: Generate FSBL.....	48
Part 1.4.2: SD card Startup Test.....	53
Part 1.4.3: QSPI Startup Test.....	55
Part 1.4.4: Programming QSPI Under Vivado.....	57
Part 1.5: Q&A.....	60
Part 1.5.1: Only PL side Logic Solidification.....	60

Part 1.6: Use skills to Share.....	63
Part 1.7: Experimental Summary.....	64
Part 2: PS RTC Interrupt Experiment.....	65
Part 2.1: RTC Introduction.....	65
Part 2.2: Interrupt Introduction.....	67
Software Engineer Job Content.....	72
Part 2.3: Vitis Programming.....	72
Part 2.3.1: Create Platform Project.....	72
Part 2.4: Download and Debug.....	79
Part 2.5: Experimental Summary.....	79
Part 3: PS MIO Experiment.....	80
Software Engineer Job Content.....	80
Part 3.1: Principle Introduction.....	81
Part 3.2: Create a “Vivado” Project.....	81
Part 3.3: Vitis Program Development.....	82
Part 3.3.1: MIO Lights up PS LED.....	82
Part 3.3.2: MIO Key Interrupt.....	91
Part 3.4: Knowledge Sharing.....	94
Part 3.5: Experimental Summary.....	98
Part 4: PS Side UART Read and Write Control.....	99
Software Engineer Job Content.....	99
Part 4.1: UART Module Introduction.....	100
Part 4.2: Vitis Program Development.....	101
Part 4.3: Onboard Verification.....	104
Part 4.4: Experimental Summary.....	105
Part 5: PS Side Use of CAN.....	107
Software Engineer Job Content.....	107
Part 5.1: Vitis Program Development.....	107
Part 5.2: Download and Test.....	108

Part 6: PS Side Use of I2C.....	113
Software Engineer Job Content.....	113
Part 6.1: Vitis Program Development.....	113
Part 6.1.1: Temperature Sensor Test.....	113
Part 6.1.2: EEPROM Read and Write.....	117
Part 7: PS Side Use of Display Port.....	120
Software Engineer Job Content.....	120
Part 7.1: Interface Introduction.....	120
Part 7.2: Example Project Introduction.....	121
Part 7.3: On-board verification.....	122
Part 8: PS Side SD Card Read and Write.....	124
Part 8.1: FatFs Introduction.....	124
Part 8.2: Vitis program development.....	125
Part 8.3: Onboard Verification.....	129
Part 9: PS Side Use of Ethernet (LWIP).....	131
Software Engineer Job Content.....	131
Part 9.1: Vitis Program Development.....	131
Part 9.1.1: LWIP Library Modification.....	131
Part 9.1.2: Create an APP Based on the LWIP Template	139
Part 9.2: Download Debugging.....	139
Part 9.3: Experimental summary.....	141
Part 10: PS Side Remote Update QSPI Flash by Ethernet.....	142
Software Engineer Job Content.....	142
Part 10.1: Vitis Program Development.....	142
Part 10.1.1: UDP Transmission Mode.....	142
Part 10.1.2: TCP Transmission Method.....	144
Part 10.1.3: QSPI Flash Read and Write Control.....	145
Part 10.2: Onboard Verification.....	146
Part 10.2.1: UDP Mode.....	147

Part 10.2.2: TCP Mode.....	149
Part 11: Use of System Monitor.....	151
FPGA Engineer Job Content.....	151
Part 11.1: Hardware Read System Monitor.....	152
Software Engineer Job Content.....	155
Part 11.2: PS read System Monitor Information.....	155
PS and PL Interconnection Parts.....	157
Part 12: PS Side Use of EMIO.....	158
Part 12.1: Principle Introduction.....	158
FPGA Engineer Job Content.....	159
Part 12.2: Create a Vivado Project.....	159
Part 12.3: XDC File Constraint PL Pin.....	161
Software Engineer Job Content.....	162
Part 12.4: Vitis Programming.....	163
Part 12.4.1: EMIO Lights PL LED.....	163
Part 12.4.2: EMIO Implements PL Key Interrupt.....	164
Part 12.5: Build Project.....	165
Part 12.6: EMIO Usage of UART Serial Port.....	166
Part 12.7: Pin Binding Common Errors.....	168
Part 12.8: Experimental Summary.....	169
Part 13: PL Side Use of AXI GPIO.....	170
Part 13.1: Principle Introduction.....	170
FPGA Engineer Job Content.....	171
Part 13.2: Create a “Vivado” Project.....	171
Part 13.2.1: Add “AXI GPIO”.....	172
Part 13.3: XDC File Constraint PL Pin.....	179
Software Engineer Job Content.....	181
Part 13.4: Vitis Programming.....	181
Part 13.4.1: AXI GPIO lights PL LED.....	181

Part 13.4.2: Download and Debug.....	184
Part 13.4.4: PL Side AXI GPIO Key Interrupt.....	187
Part 13.5: Experimental summary.....	189
Part 13.6: Knowledge Sharing.....	189
Part 14: PL Side RS485 Test.....	193
FPGA Engineer Job Content.....	193
Part 14.1: Create a Hardware Project.....	193
Software Engineer Job Content.....	199
Part 14.2: Vitis Program Development.....	199
Part 14.3: Download Test.....	201
Part 14.4: Experimental Summary.....	202
Part 15: PL Side Use of Ethernet.....	203
FPGA Engineer Job Content.....	203
Part 15.1: Create a Hardware Project.....	203
Software Engineer Job Content.....	210
Part 15.2: Vitis Program Development.....	210
Part 15.2.1: PL Side Ethernet test.....	210
Part 15.2.2: PS Side Ethernet Test.....	211
Part 16: Custom IP experiment.....	212
FPGA Engineer Job Content.....	212
Part 16.1: PWM Introduction.....	212
Part 16.2: Building a Vivado project.....	214
Part 16.2.1: Create a custom IP.....	214
Part 16.2.2: Add a Custom IP to the Project.....	222
Part 16.3: Vitis software Writing and Debugging.....	224
Part 16.4: Experimental Summary.....	228
Part 17: Use of Dual Core AMP.....	229
Software Engineer Job Content.....	229
Part 17.1:Vitis Program Development.....	229

Part 17.1.1: Create CPU0 Vitis Project.....	229
Part 17.1.2: Create a CPU1 Vitis project.....	231
Part 17.1.3: CPU0 Programs Introduction.....	232
Part 17.1.4: CPU1 Programs Introduction.....	233
Part 17.2: Onboard Verification.....	234
Part 17.3: QSPI Flash Startup.....	235
Part 17.4: Experimental Summary.....	237
Part 18: Use of “Free RTOS” under ZYNQ.....	238
Software Engineer Job Content.....	238
Part 18.1: Vitis Program Development.....	238
Part 18.2: Onboard Verification.....	240
Part 18.3: Experimental Summary.....	241
Part 19: PL Read and Write PS DDR Data.....	242
FPGA Engineer Job Content.....	242
Part 19.1: Use of ZYNQ HP Port.....	242
FPGA Engineer Job Content.....	243
Part 19.2: Hardware Environment.....	243
Part 19.3: PL Side AXI Master.....	248
Part 19.4: Verification of ddr Read and Write Data.....	251
Part 19.5: Vivado Software Debugging Skills.....	252
Part 19.6: Vitis Program Development.....	252
Part 19.7: Experimental Summary.....	254
Part 20: Realize PS and PL Data Interaction through BRAM... ..	255
FPGA Engineer Job Content.....	256
Part 20.1: Hardware Environment.....	256
Part 20.1.1: Block Design adds logic analyzer method	260
Part 20.2: Vitis Program Development.....	262
Part 20.3: Experimental Result.....	264
Part 20.4: Experimental Summary.....	267

Part 21: Use VDMA to drive HDMI display.....	268
FPGA Engineer Job Content.....	268
Part 21.1: Create a Vivado Project.....	268
Part 21.1.1: Configure VDMA.....	270
Part 21.1.2: Add custom IP.....	274
Software Engineer Job Content.....	281
Part 21.2: Vitis Software Writing and Debugging.....	282
Part 21.3: Onboard Verification.....	284
Part 21.4: Experimental Summary.....	285
Part 22: Use VDMA to Drive HDMI Acquisition and Display....	286
FPGA Engineer Job Content.....	286
Part 22.1: Create a Vivado Project.....	286
Software Engineer Job Content.....	289
Part 22.2: Vitis Software Writing and Debugging.....	289
Part 22.3: Onboard Verification.....	290
Part 23: MIPI Acquisition and DP Display Based on AN5641 Module.....	291
Part 23.1: Principle Introduction.....	291
Part 23.1.1: MIPI Physical Layer (D-PHY).....	291
Part 23.1.2: MIPI Protocol Layer (CSI-2).....	292
FPGA Engineer Job Content.....	294
Part 23.2: Hardware Environment.....	294
Software Engineer Job Content.....	304
Part 23.3: Vitis Program Development.....	304
Part 33.4: Onboard Verification.....	305
Part 24: MIPI Acquisition and HDMI Display Based on AN5641 Module.....	309
FPGA Engineer Job Content.....	309
Part 24.1: Create a Vivado Project.....	309

Software Engineer Job Content.....	310
Part 24.2: Vitis Software Writing and Debugging.....	310
Part 24.3: Onboard Verification.....	311
Part 25: PCIe Test.....	313
FPGA Engineer Job Content.....	313
Part 25.1: Create a Vivado Project.....	313
Part 25.1.1: PCIe xdma Configuration.....	313
Part 25.1.2: ZNYQ Configuration.....	316
Part 25.1.3: Module Connection.....	317
Part 25.2: Generate and burn BOOT.....	321
Part 25.3: Set the computer to enter the test mode.....	321
Part 25.4: Install PCIe Driver.....	323
Part 25.5: Testing PCIe.....	326
Part 25.6: Experiment Summary.....	328

Preparation and precautions

Software Environment

The software development environment is based on Vivado 2020.1

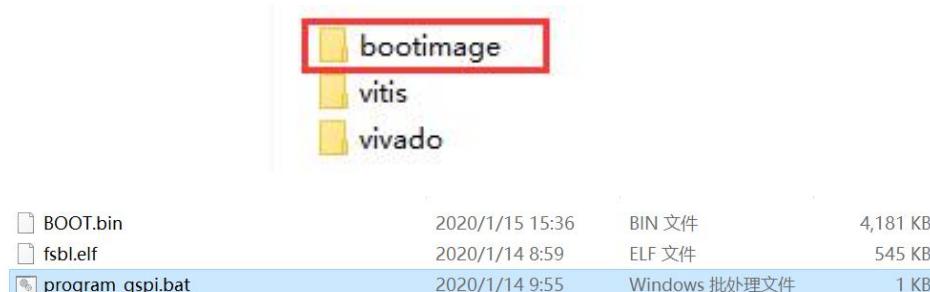


Hardware environment

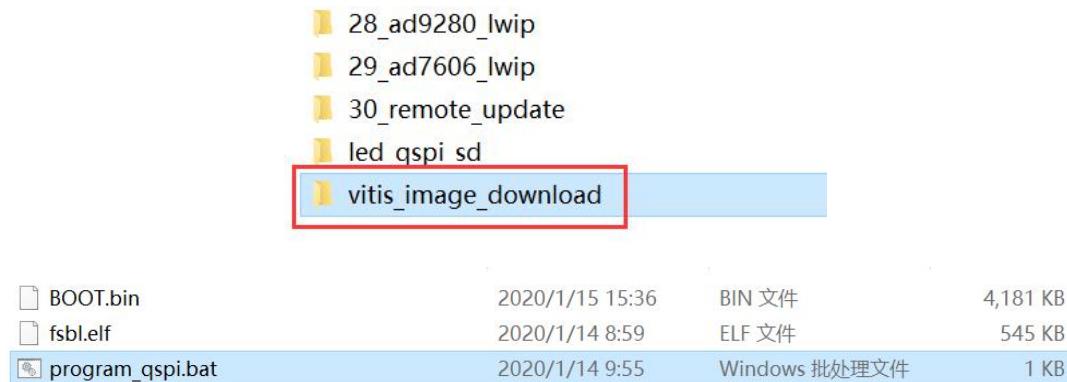
FPGA Development Board Model	FPGA Chip
AXU3EG	Xazu3eg-sfvc784-1-i
AXU4EV	Xczu4ev-sfvc784-1-i
AXU5EV	xazu5ev-sfvc784-1-i

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.

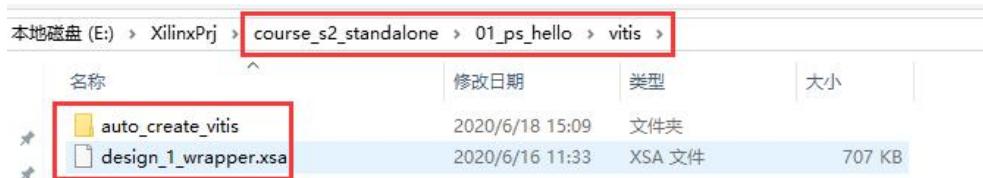
Flash Operation Successful
请按任意键继续 . . .
```

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

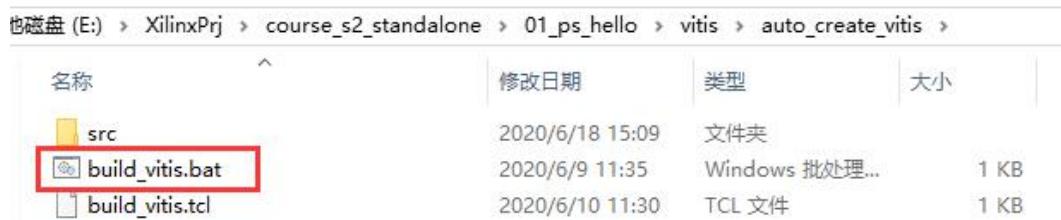
Batch process to build Vitis project

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**

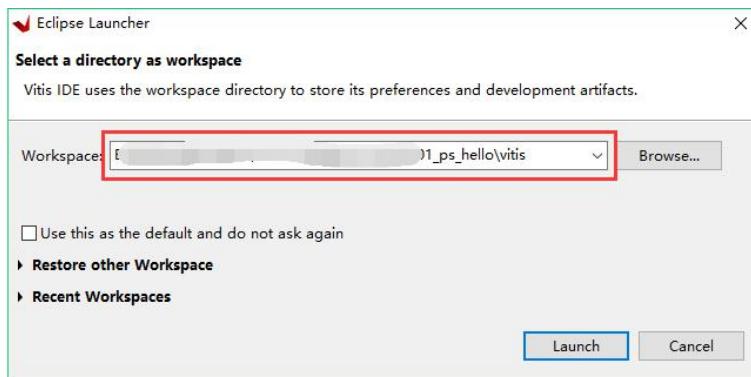
```
call E:\XilinxVitis\Vitis\2020.1\bin\xsct.bat build_vitis.tcl
pause
```

After saving, double-click **build_vitis.bat** to create the project

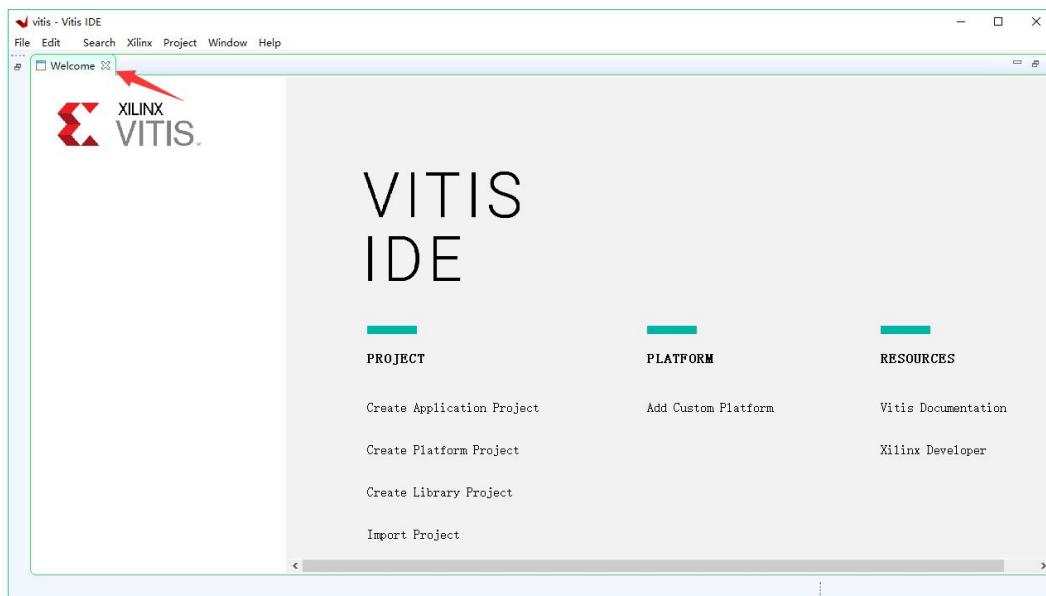
Compilation is over, press any key to exit

```
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...
```

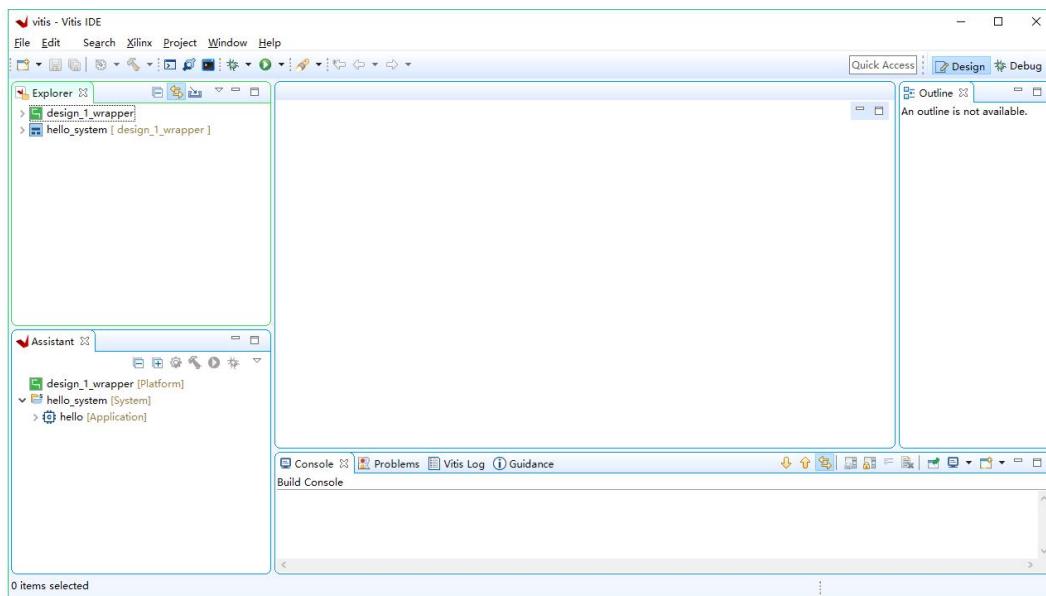
Open the **Vitis** software, select the project path, **Launch**



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

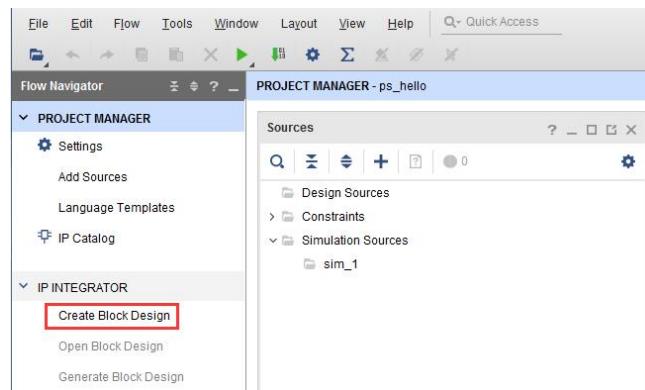
divided into PL and PS. The IO assignment on the PS side is relatively fixed and cannot be arbitrarily assigned, and there is no need to assign pins in the Vivado software. Although this experiment only uses PS, a Vivado project must also be established to configure the PS pins. Although the ARM on the 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

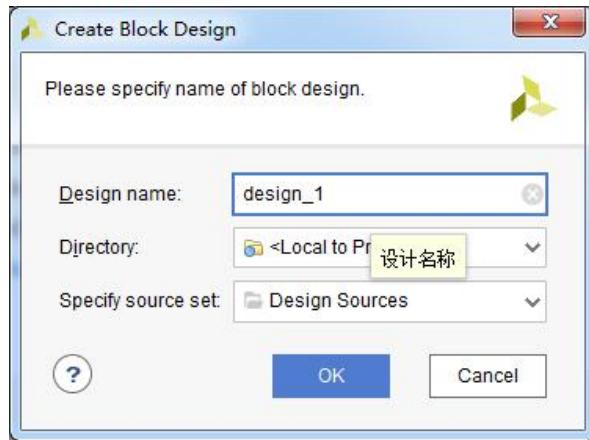
The following is the content that FPGA engineers are responsible for.

Part 1.2: Create a Vivado Project

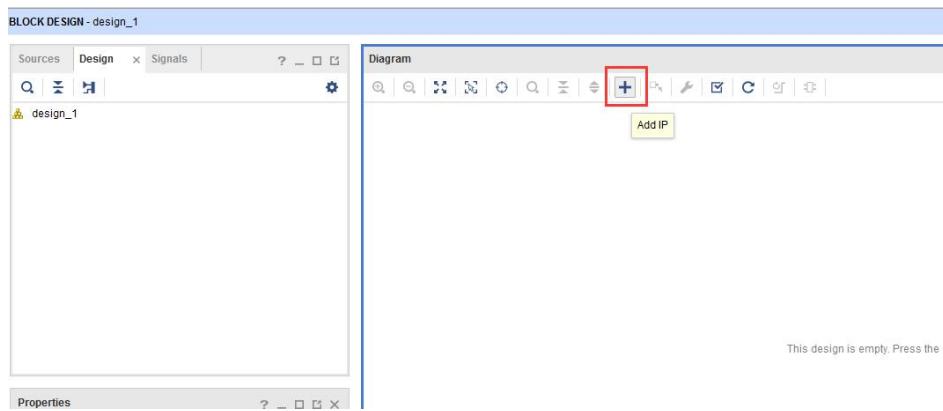
- 1) Create a project called "[ps_hello](#)". The creation process will not be repeated, please refer to "PL's "Hello World" LED Experiment".
- 2) Click on "[Create Block Design](#)" to create a block design, which is a graphical design



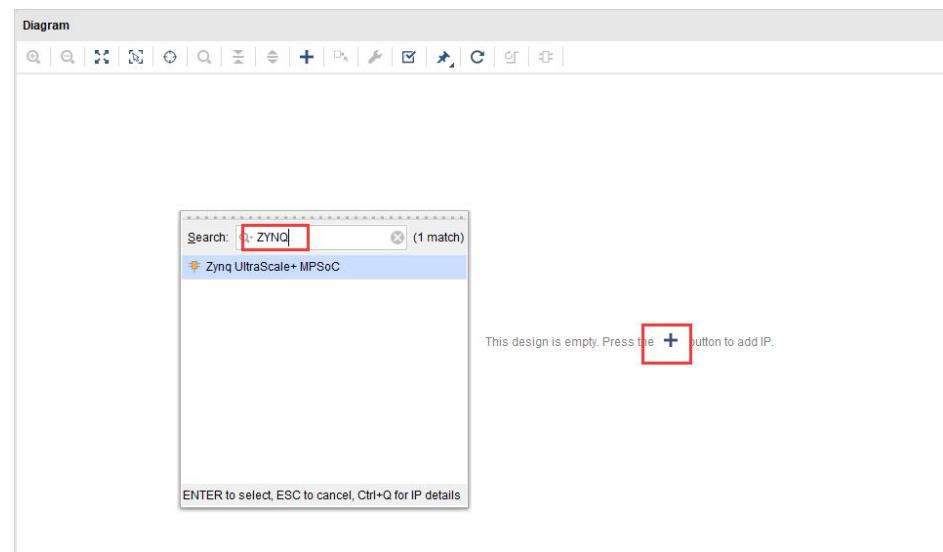
- 3) "Design name" is not modified here, keep the default "design_1", which can be modified as needed, but the name should be as short as possible, otherwise there will be problems compiling under Windows.



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



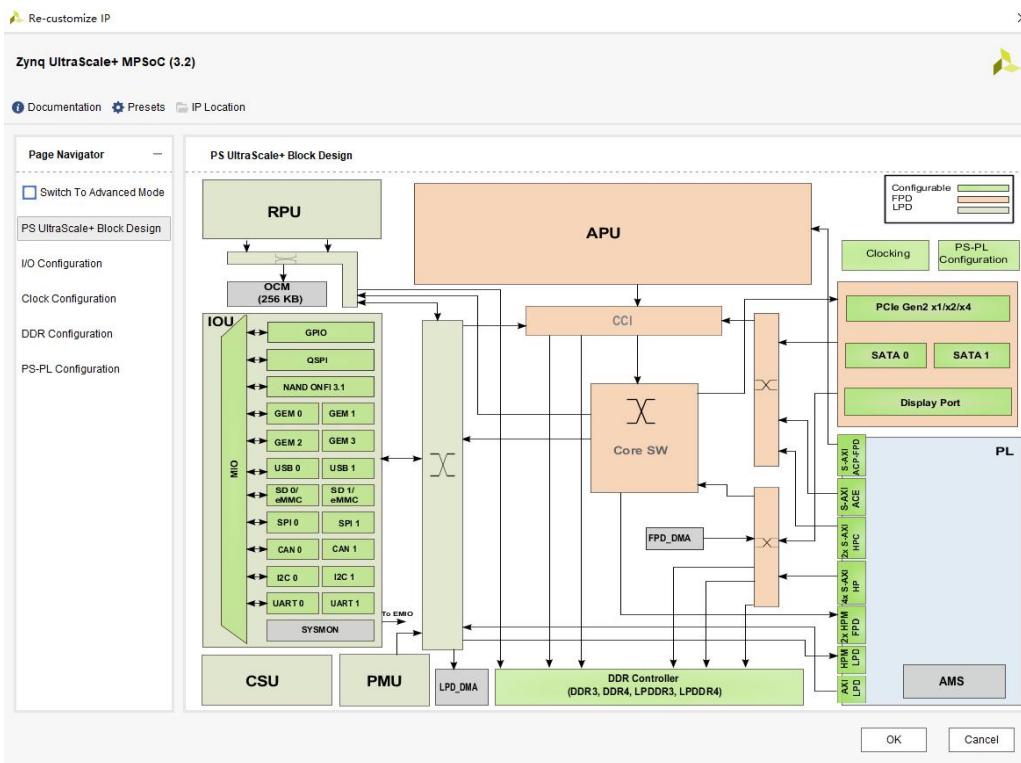
- 5) Search for "zynq" and double-click "ZYNQ UltraScale+ MPSoC" in the search results list



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



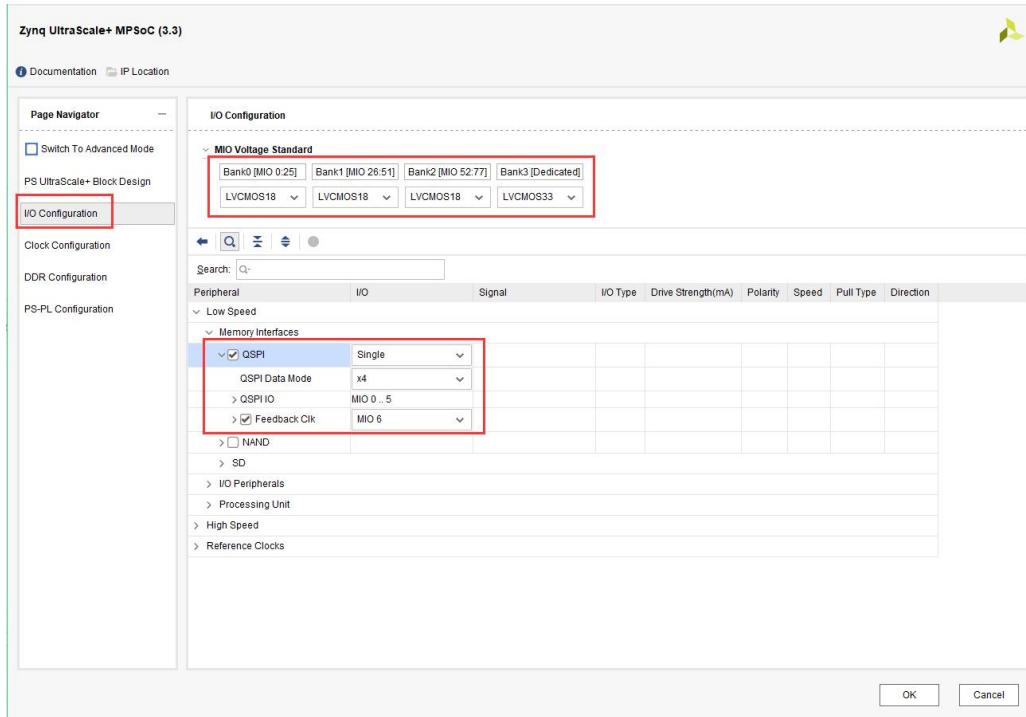
- 7) The first interface that appears is the ZYNQ hard core architecture diagram. You can clearly see its structure. You can refer to the ug585 document, which contains a detailed introduction to ZYNQ. The green part in the picture is the configurable module. You can click to enter the corresponding 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	MIO40	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
Low Speed				
> 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
Low Speed			
> 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
High Speed							
GEM							
GEM 0							
<input type="checkbox"/>	MDIO 0						
GEM 1							
<input type="checkbox"/>	MDIO 1						
GEM 2							
<input type="checkbox"/>	MDIO 2						
GEM 3							
<input checked="" type="checkbox"/>	GEM 3	MIO 64 .. 75					
<input checked="" type="checkbox"/>	MDIO 3	MIO 76 .. 77					
Gem 3	MIO64	rgmii_tx_clk	sc...	12	s...	pull...	out
Gem 3	MIO65	rgmii_txd[0]	sc...	12	s...	pull...	out
Gem 3	MIO66	rgmii_txd[1]	sc...	12	s...	pull...	out
Gem 3	MIO67	rgmii_txd[2]	sc...	12	s...	pull...	out
Gem 3	MIO68	rgmii_txd[3]	sc...	12	s...	pull...	out
Gem 3	MIO69	rgmii_tx_ctl	sc...	12	s...	pull...	out

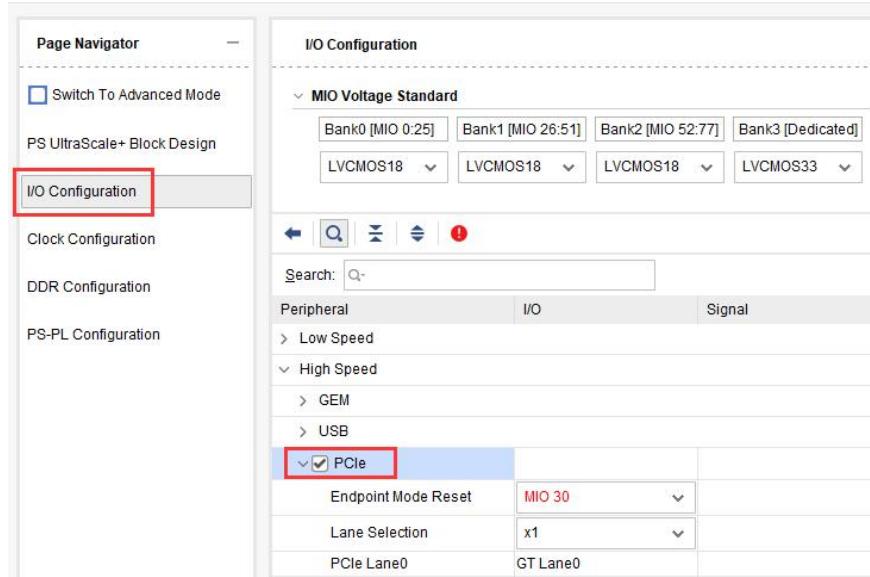
- Check **USB 0**, select **MIO 52..63**, check **USB 3.0**, select **GT Lane1**

Peripheral	I/O	Signal	I/O Type	Drive Strength(mA)	Polarity	Speed
USB						
USB0						
<input checked="" type="checkbox"/>	USB 0	MIO 52 .. 63				
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"/>	USB 3.0	GT Lane1				

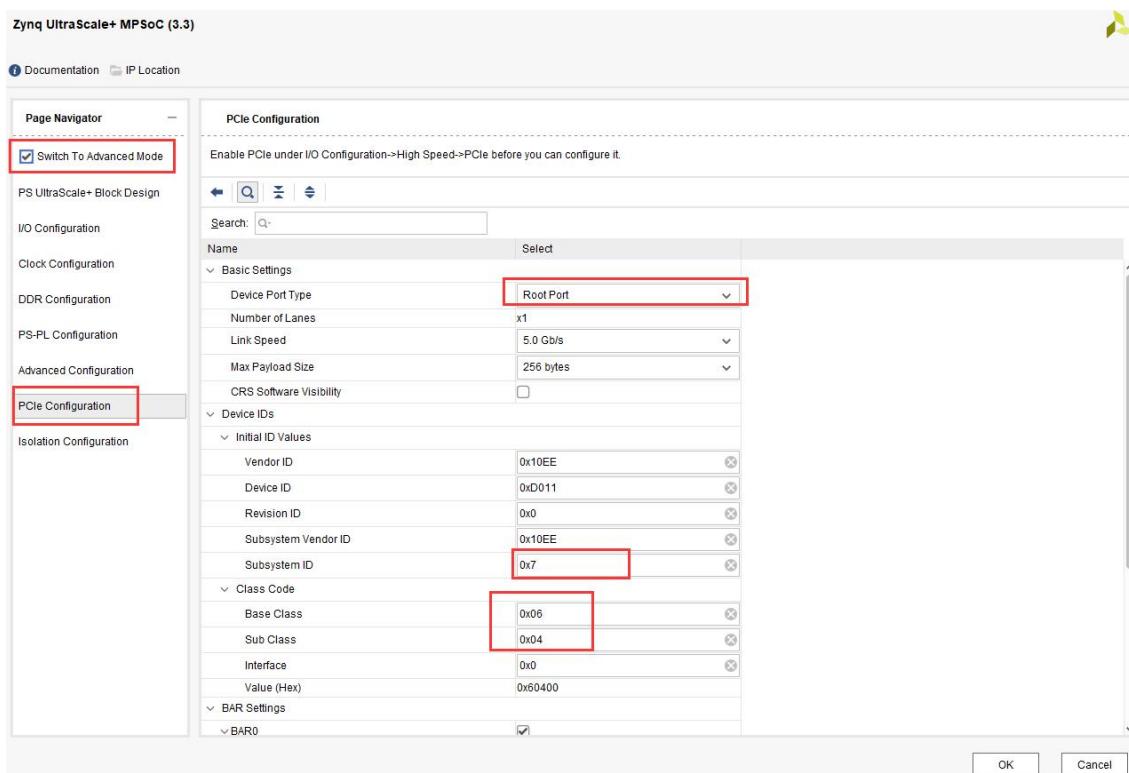
USB reset select MIO 31



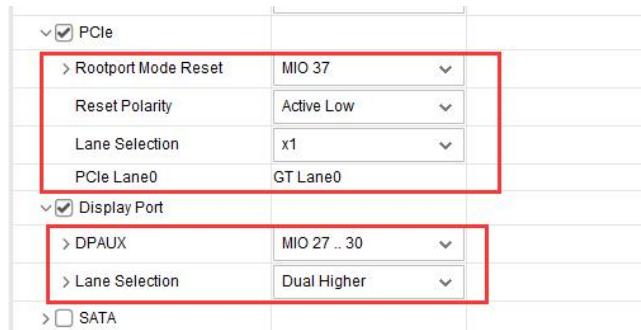
3) Check PCIe



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



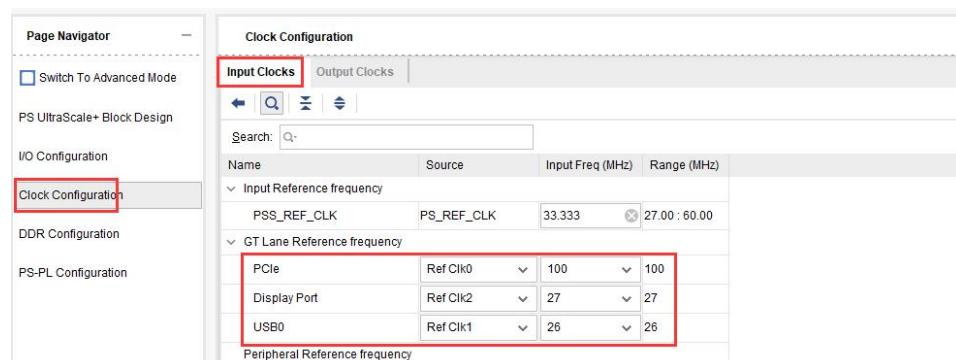
- 5) Go back to I/O Configuration, reset and select MIO 37; check Display Port, select MIO 27..30, and Lane Selection select Dual Higher



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

Name	Source	FracEn	Requested Freq (MHz)	Divisor 0	Divisor 1	Actual Frequency (MHz)	Range
CPU_R5	IOPLL		500	3		499.994995	0.0000...
QSPI	IOPLL		300	5	1	299.997009	0.0000...
SDIO0	IOPLL		200	8	1	187.498123	
SDIO1	IOPLL		200	8	1	187.498123	
SD DLL	IOPLL		1500			1499.984985	
UART0	IOPLL		100	15	1	99.999001	0.0000...
I2C1	IOPLL		100	15	1	99.999001	0.0000...
CAN0	IOPLL		100	15	1	99.999001	0.0000...
CAN1	IOPLL		100	15	1	99.999001	0.0000...
USB0	IOPLL		250	6	1	249.997498	0.0000...
USB3_DUAL	IOPLL		20	25	3	19.999800	0.0000...
Gem3	IOPLL		125	12	1	124.998749	0.0000...
GEM_TSU	IOPLL		250	6	1	249.997498	0.0000...
TTC0	APB		100.000000			100.000000	0.0000...
TTC1	APB		100.000000			100.000000	0.0000...

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

PL	Source	Requested Freq (MHz)	Divisor 0	Divisor 1	Actual Frequency (MHz)	Range
PL0	RPLL	100	8	1	99.999001	0.0000...
PL1	RPLL	100	4	1	149.998505	0.0000...
PL2	RPLL	100	4	1	100	0.0000...
PL3	RPLL	100	4	1	100	0.0000...

- 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.

Processor/Memory Clocks	Source	Requested Freq (MHz)	Divisor 0	Divisor 1	Actual Frequency (MHz)	Range
ACPU	APLL	1200	1		1199.988037	0.0000...
DDR	DPLL	400.000	3		399.996002	100.00...
DP_VIDEO	VPLL	300	5	1	299.997009	0.0000...
DP_AUDIO	RPLL	25	16	1	24.999750	0.0000...
DP_STC	RPLL	27	15	1	26.666401	0.0000...
SATA	IOPLL	250	2		249.997498	0.0000...
System Debug Clocks						
DBG_FPD	IOPLL	250	2		249.997498	0.0000...
DBG_TSTMP	IOPLL	250	2		249.997498	0.0000...

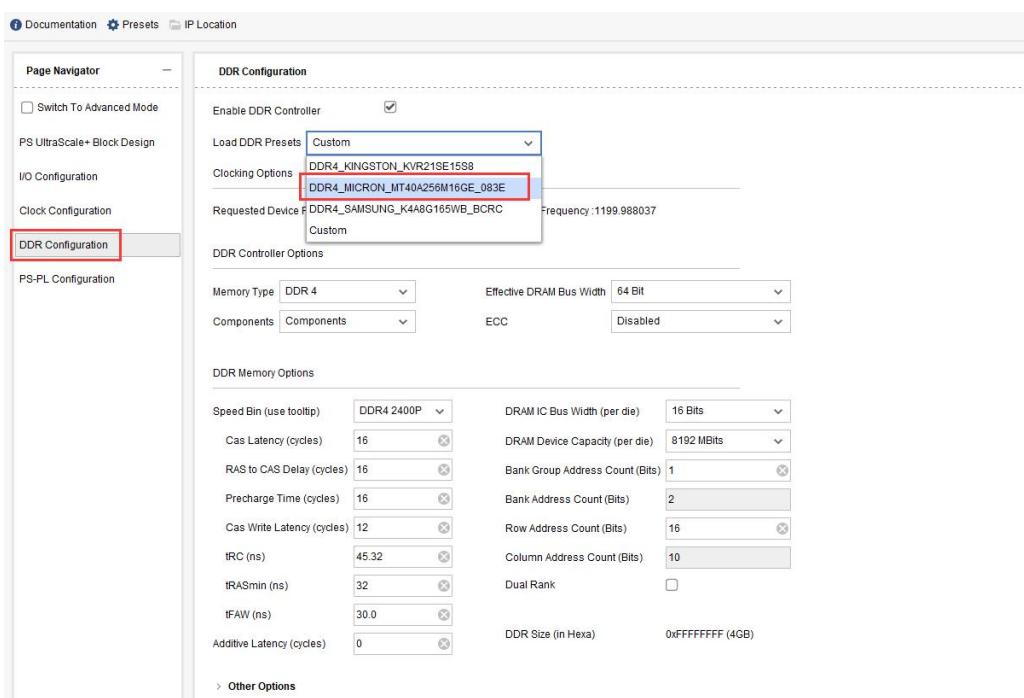
The bottom Interconnect is modified as follows

FPD_DMA	DPLL	600	<input type="button" value="X"/>	2	599.994019	0.0000...
DPDMA	DPLL	600	<input type="button" value="X"/>	2	599.994019	0.0000...
TOPSW_MAIN	APLL	533.333	<input type="button" value="X"/>	2	533.328003	0.0000...
TOPSW_LSBUS	IOPLL	100	<input type="button" value="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

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

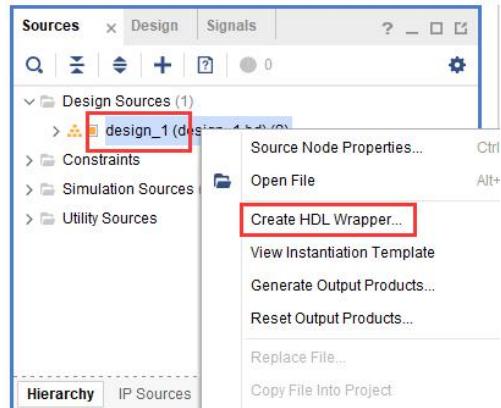
DDR Memory Options

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		

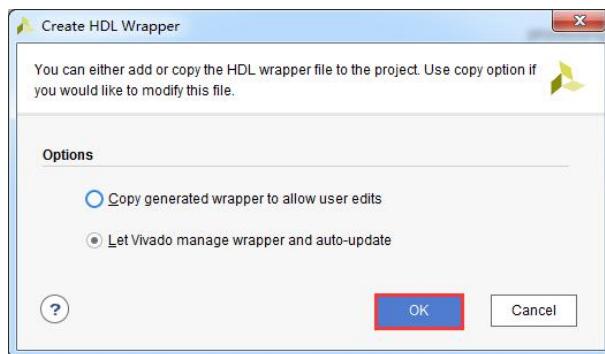
Keep the others as default, click OK, the configuration is complete, and connect the clock as follows:



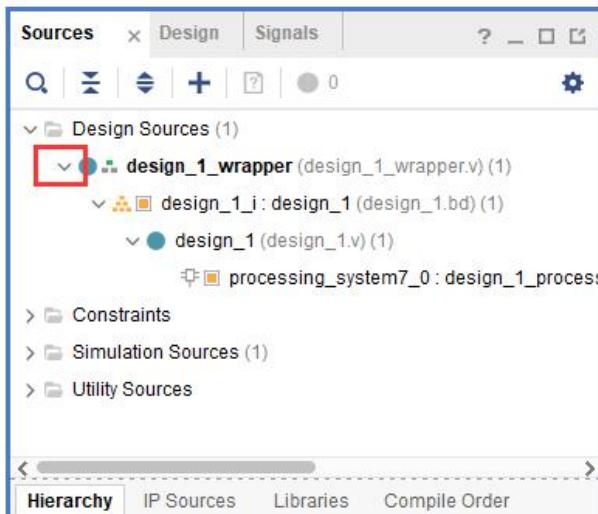
- 2) Select Block design, right click "Create HDL Wrapper...", create a Verilog or VHDL file, and generate HDL top-level file for block design.



3) Keep the default options and click "OK"

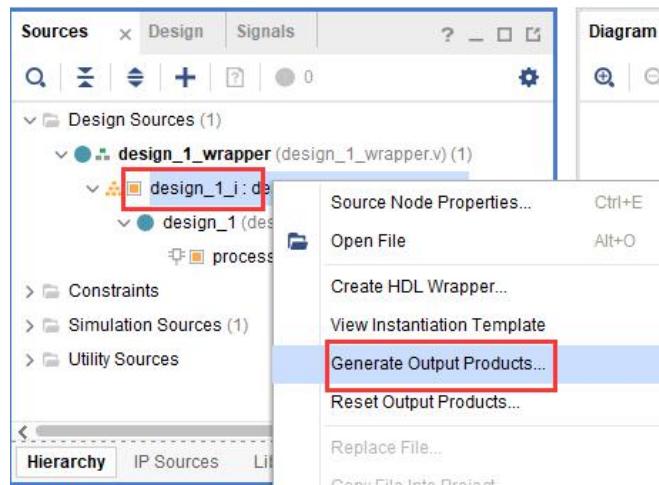


4) Expand the design and you can see that the PS is used as an ordinary IP.

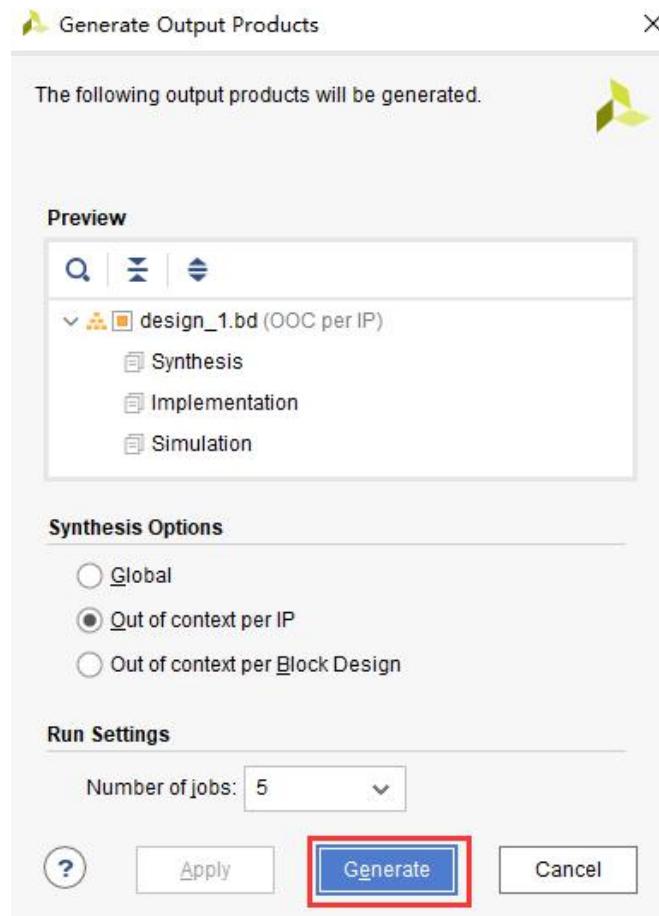


5) Select the block design and right click "Generate Output Products".

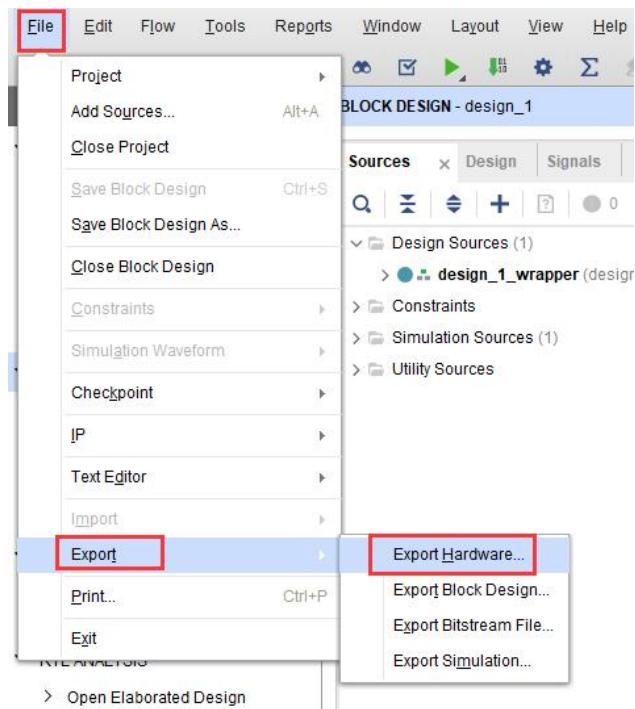
This step will generate block output files, including IP, instantiation templates, RTL source files, XDC constraints, third-party integrated source files, and so on. For subsequent operations.



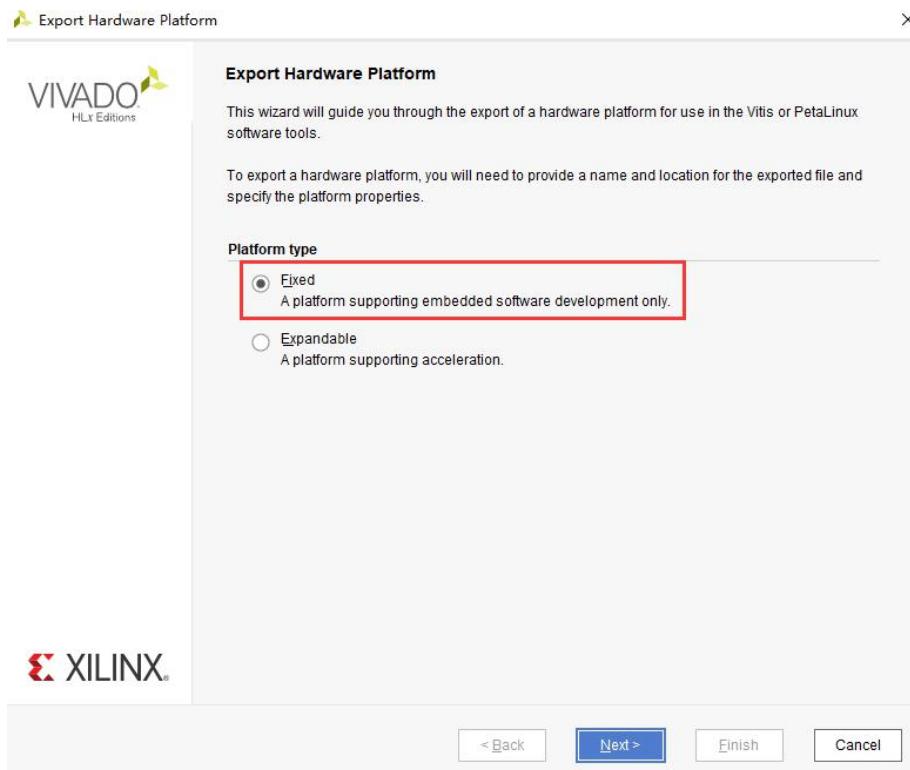
6) Click "Generate"



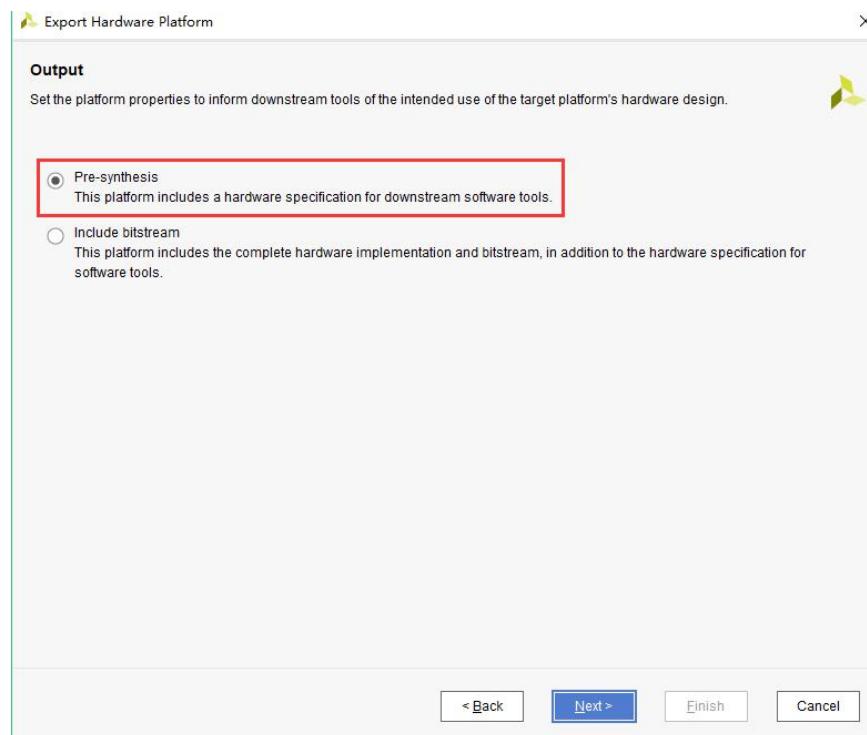
7) Export the hardware information in the menu bar "File -> Export -> Export Hardware...", here it contains the configuration information of the PS side.



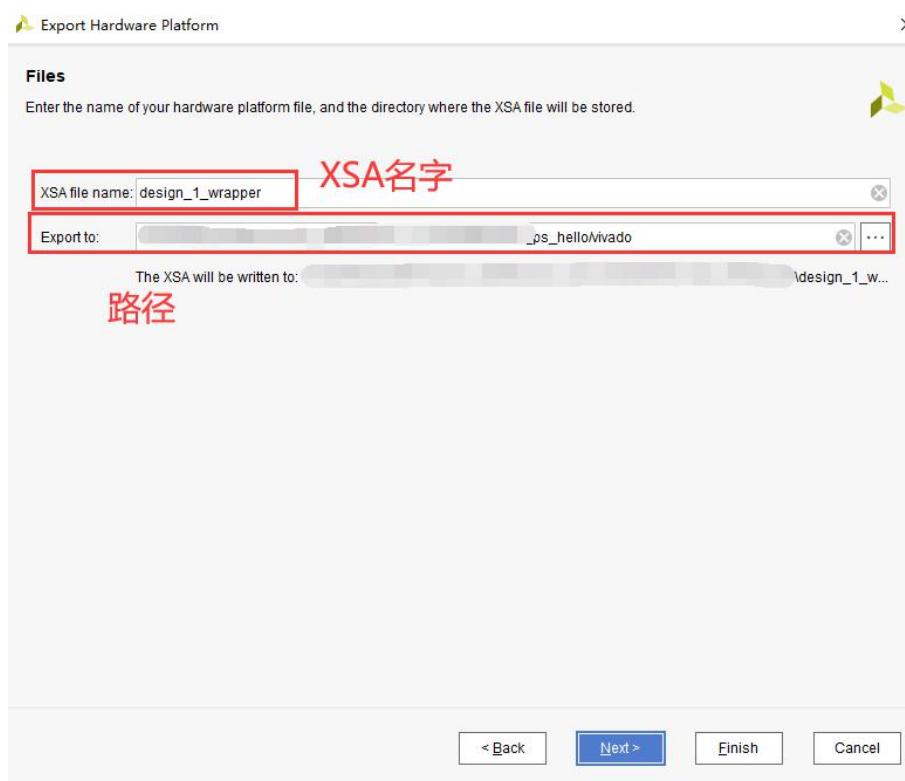
8) Select Fixed in the pop-up window, click Next



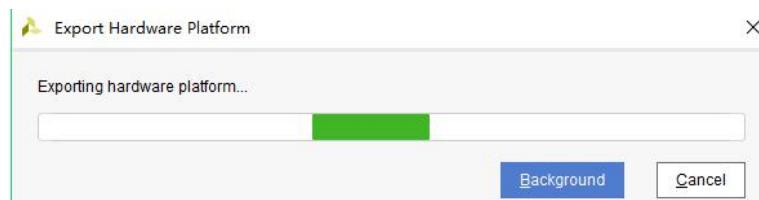
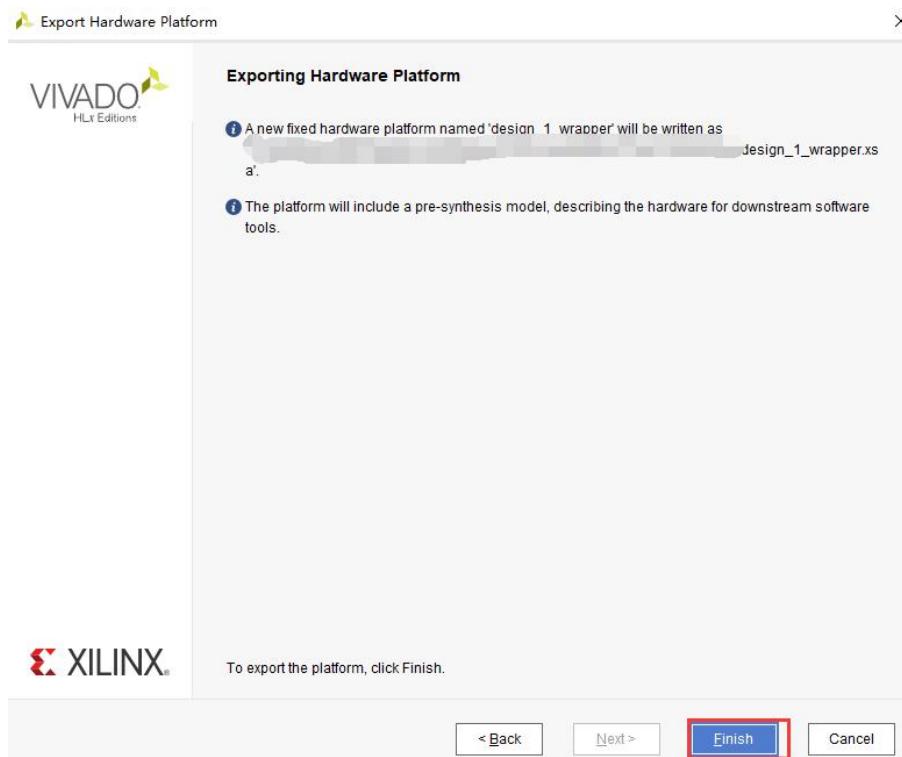
9) Click "OK" in the pop-up dialog box, because the experiment only uses the PS serial port and does not require PL to participate. There is no enable here. Do not select "Include bitstream", click Next



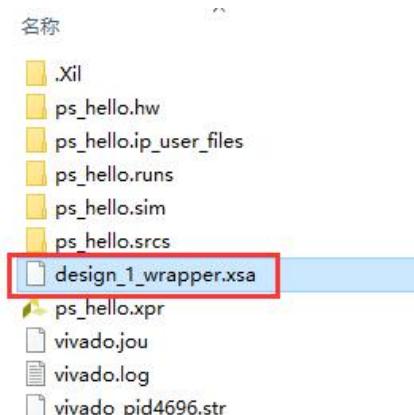
10) The export name and export path can be modified. The default is in the vivado project directory. This file can be placed in a suitable location according to your needs. It does not have to be placed under the vivado project. The vivado and vitis software are independent. Here we choose not to change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



Click Finish



11)At this time, you can see the xsa 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.

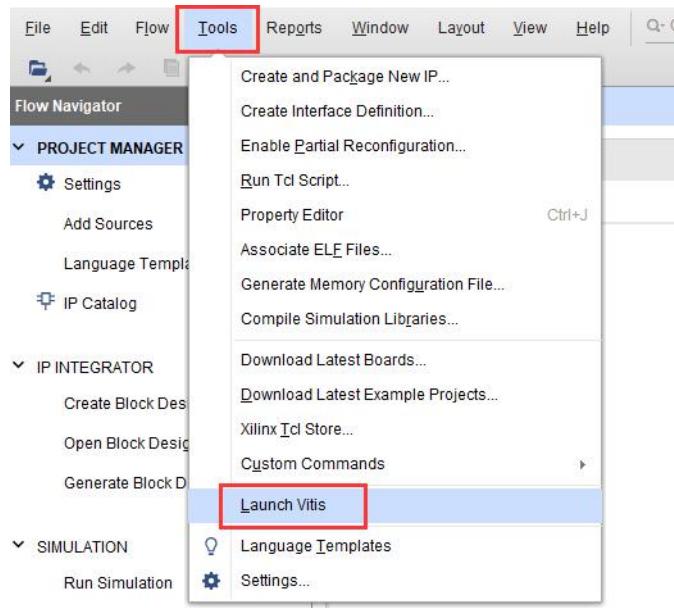
Part 1.3: Vitis Debugging

Part 1.3.1: Create Application Project

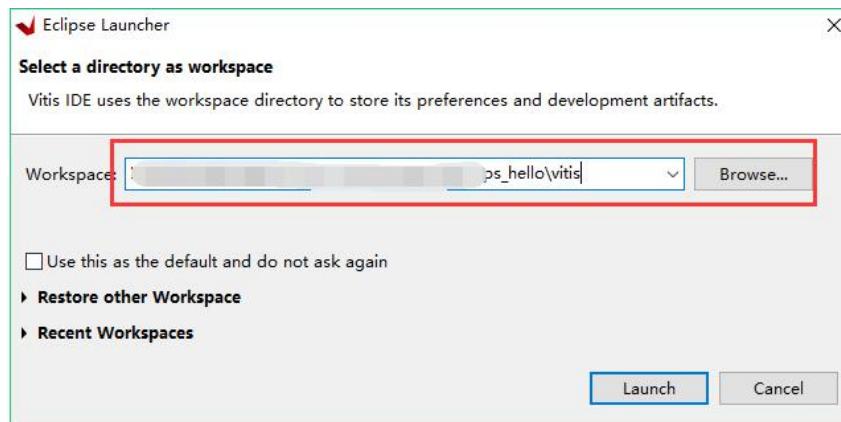
- 1) Create a new folder and copy in the `xx.xsa` file exported by vivado
- 2) Vitis is an independent software, you can double-click the Vitis software to open



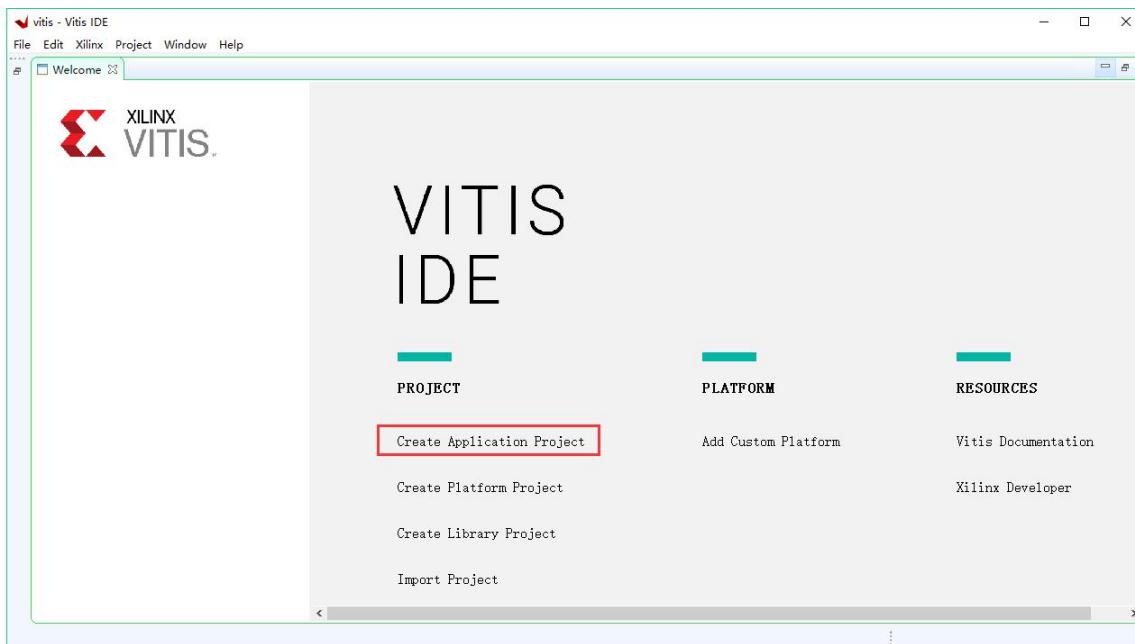
You can also open the Vitis software by selecting Tools → Launch Vitis in the Vivado software



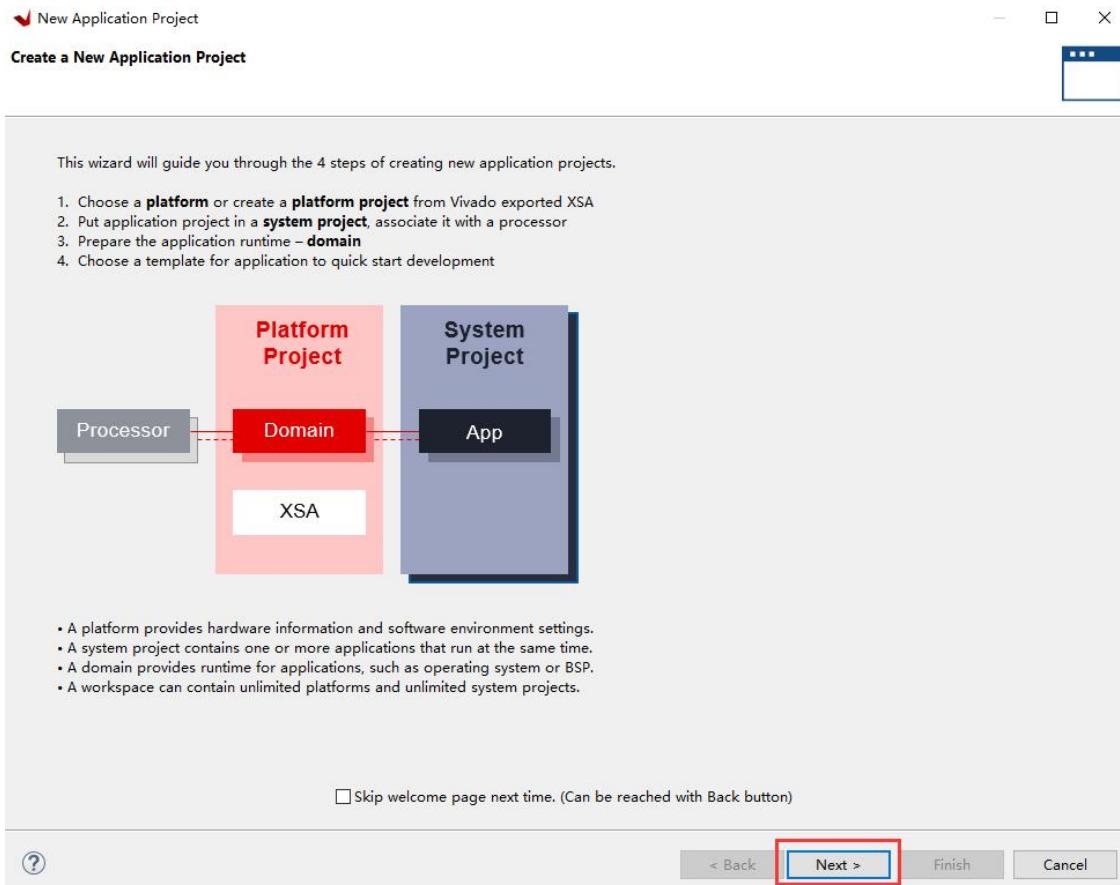
Select the newly created folder and click "Launch"



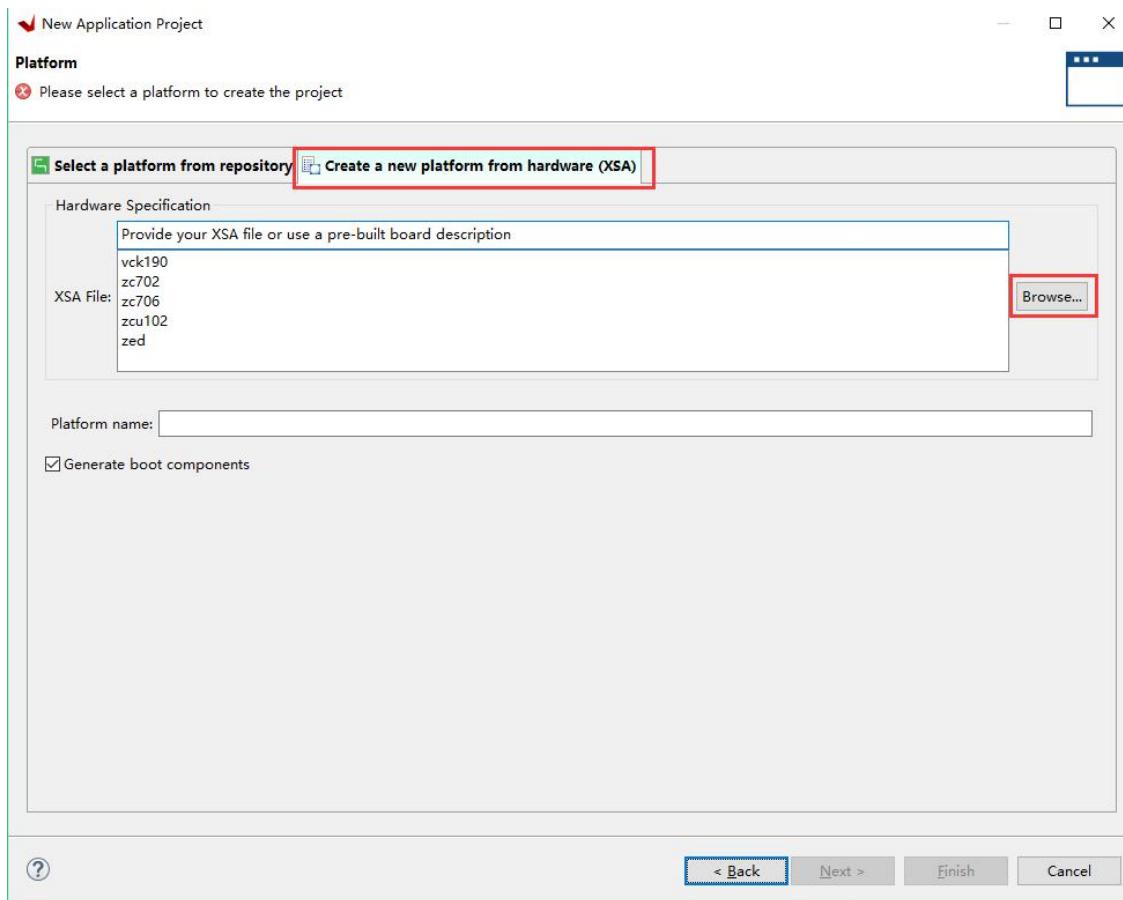
- 3) After starting Vitis, the interface is as follows, click "Create Application Project", this option will generate APP project and Platfrom project, Platform project is similar to the previous version of hardware platform, including hardware support related files and BSP.



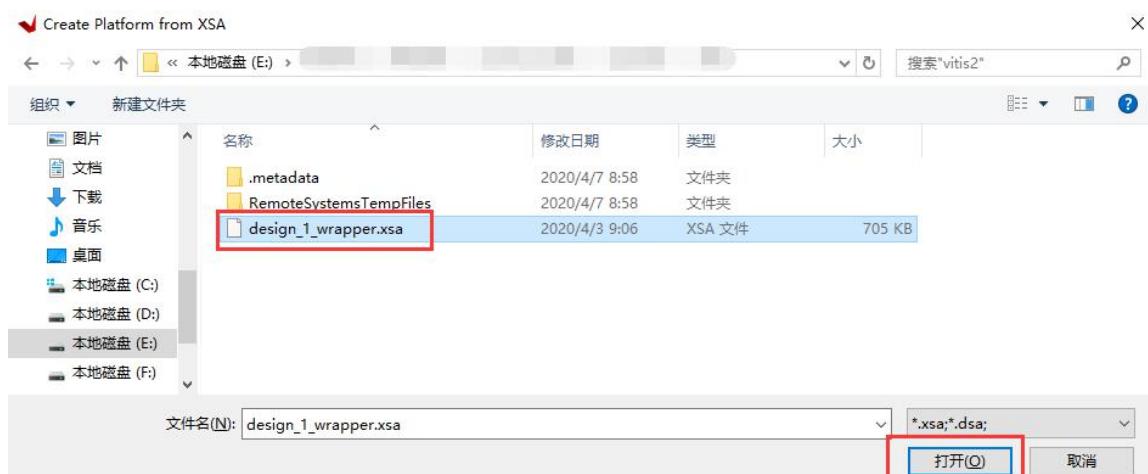
- 4) The first page is the introduction page, skip directly, click Next



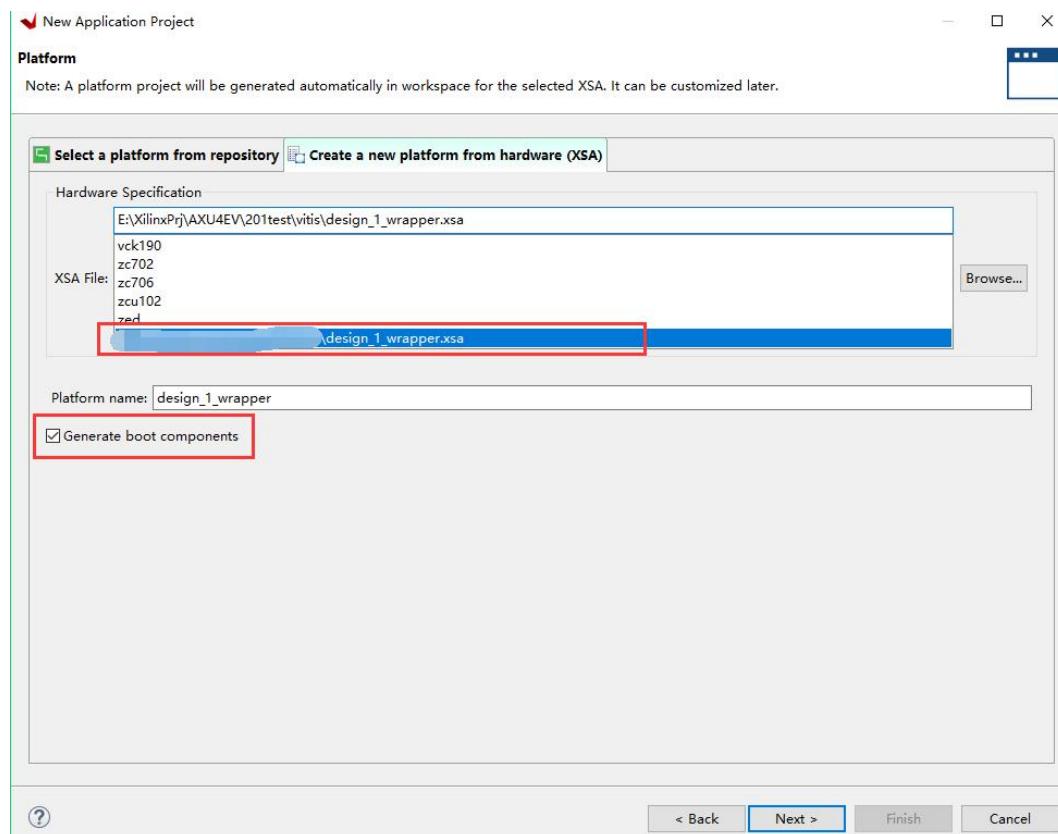
- 5) Select "Create a new platform from hardware(XSA)", select "Browse"



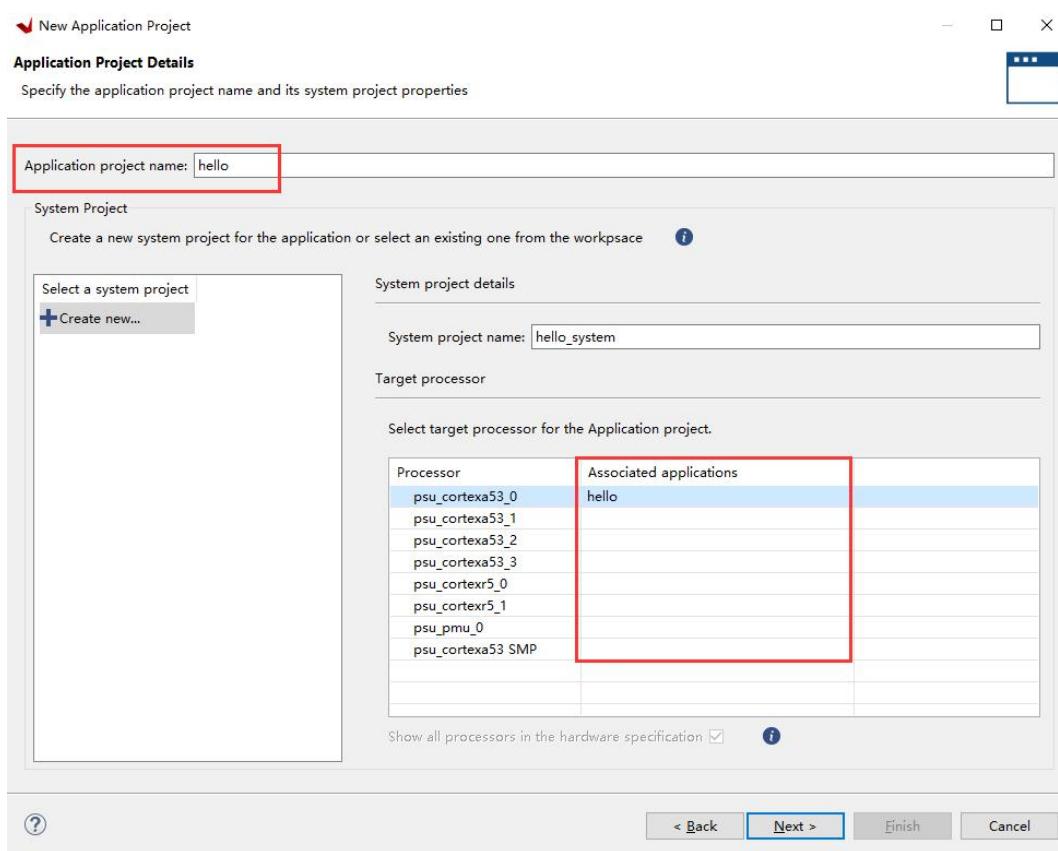
Select the previously generated xsa, click to open



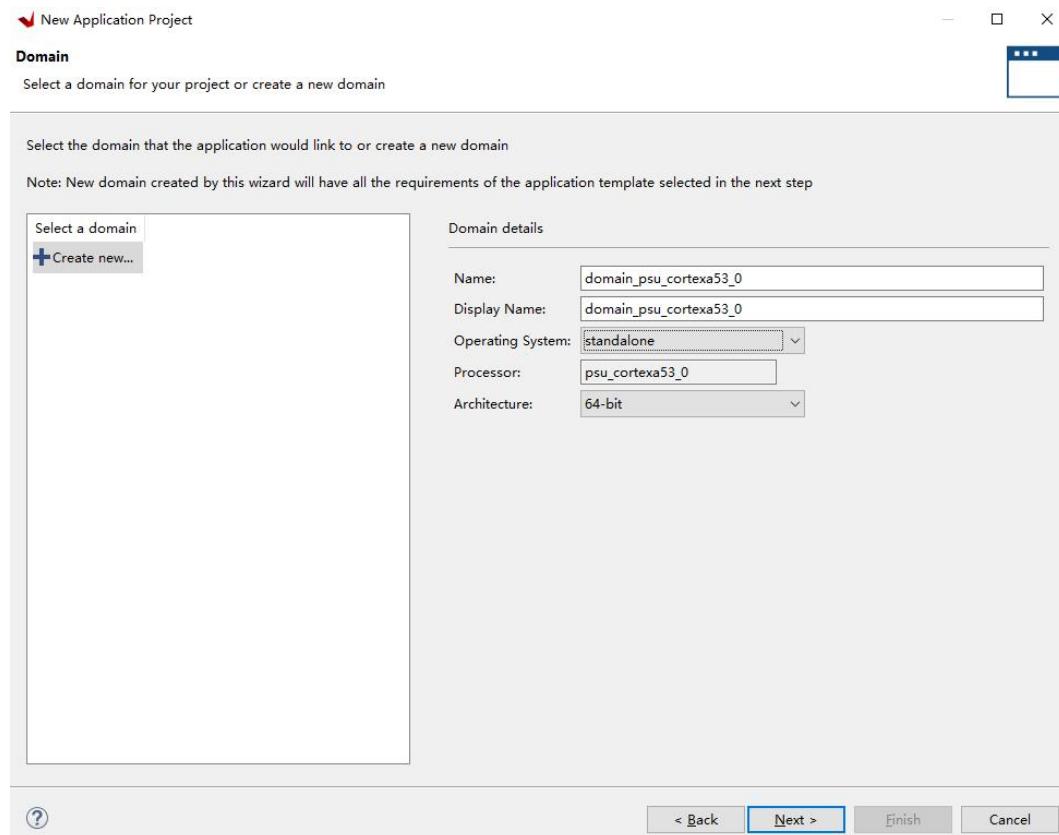
- 6) The Generate boot components option at the bottom, if checked, the software will automatically generate the fsbl project, we generally choose to check it by default



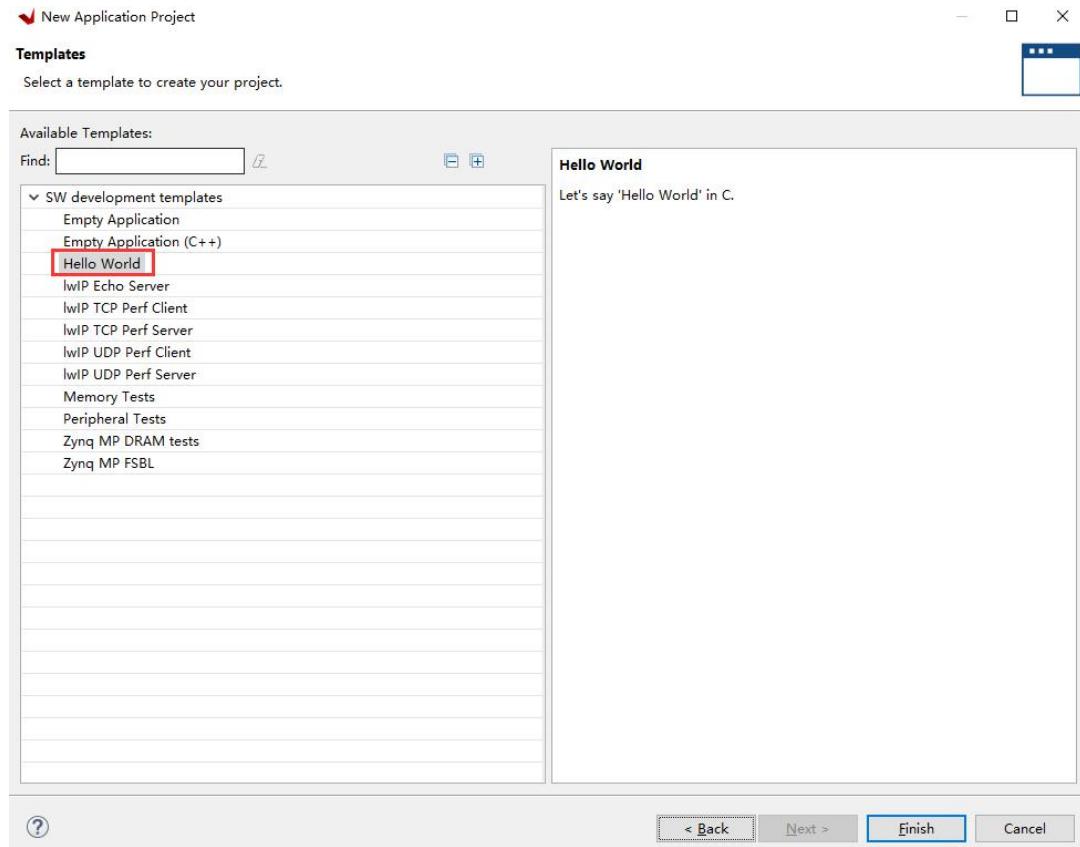
- 7) Fill in the APP project name, click in the box to select the corresponding processor, we keep the default here



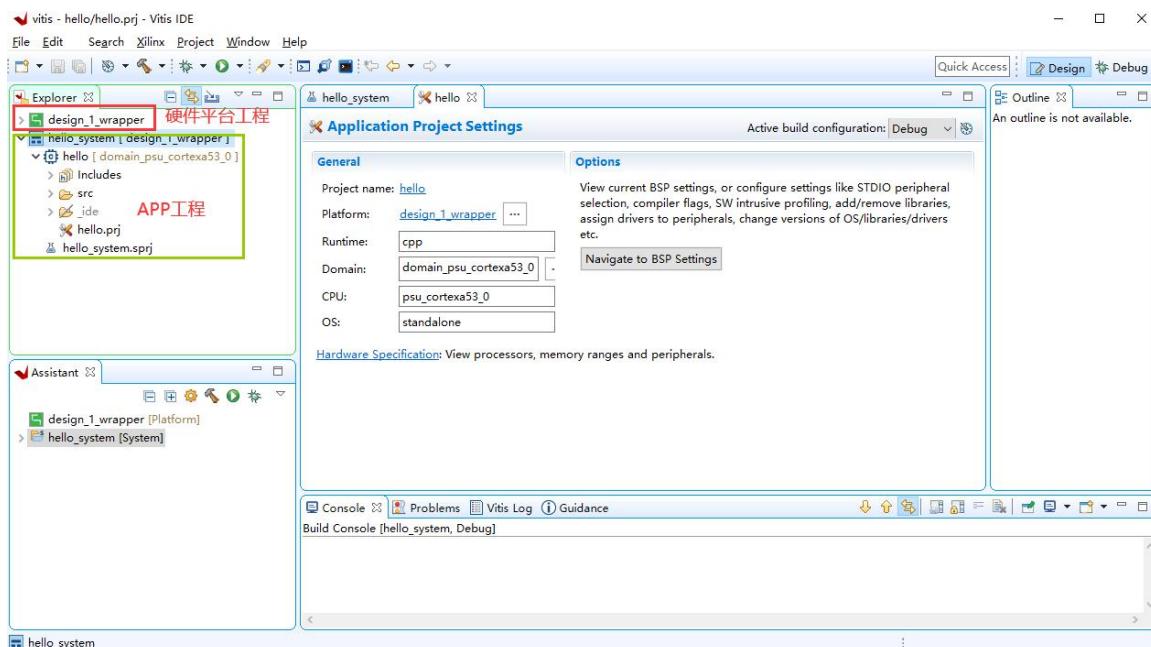
- 8) In this interface, you can modify the domain name, select the operating system, ARM architecture, etc., keep the default here, and select standalone for the operating system, which is bare metal.



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

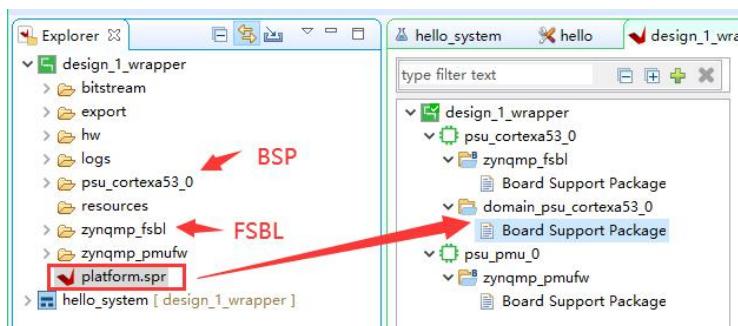


10) After completion, you can see that two projects have been generated, one is the hardware platform project, which is the Platform project mentioned before, and the other is the APP project.

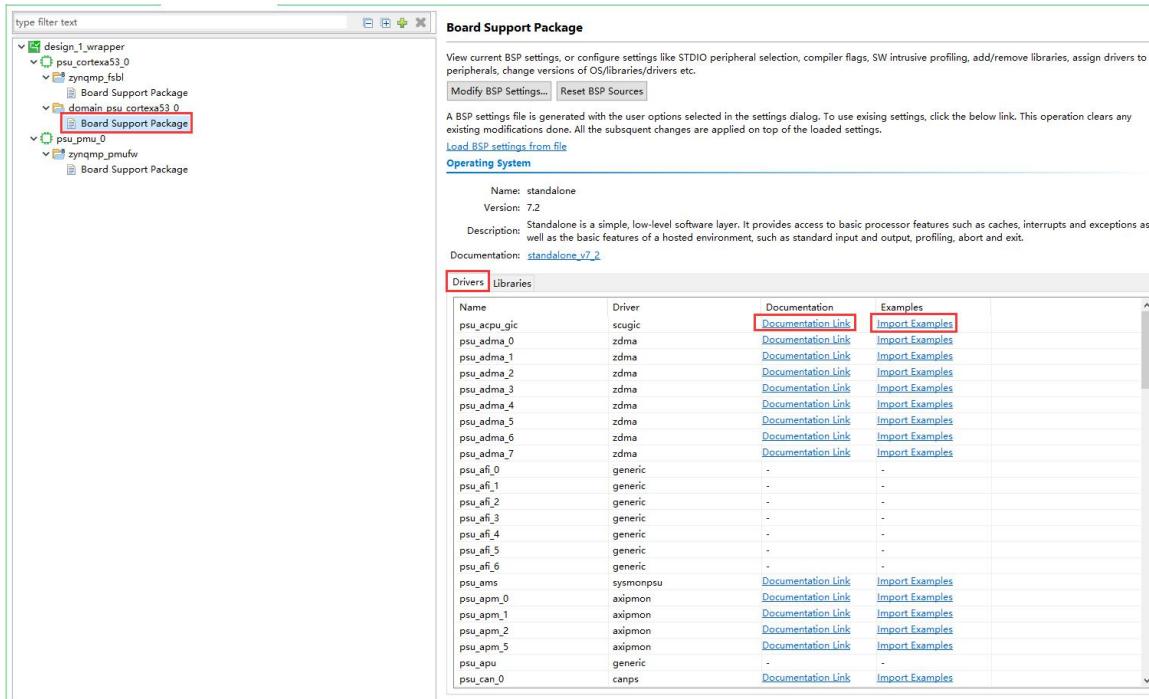


11) After expanding the Platform project, you can see that it contains

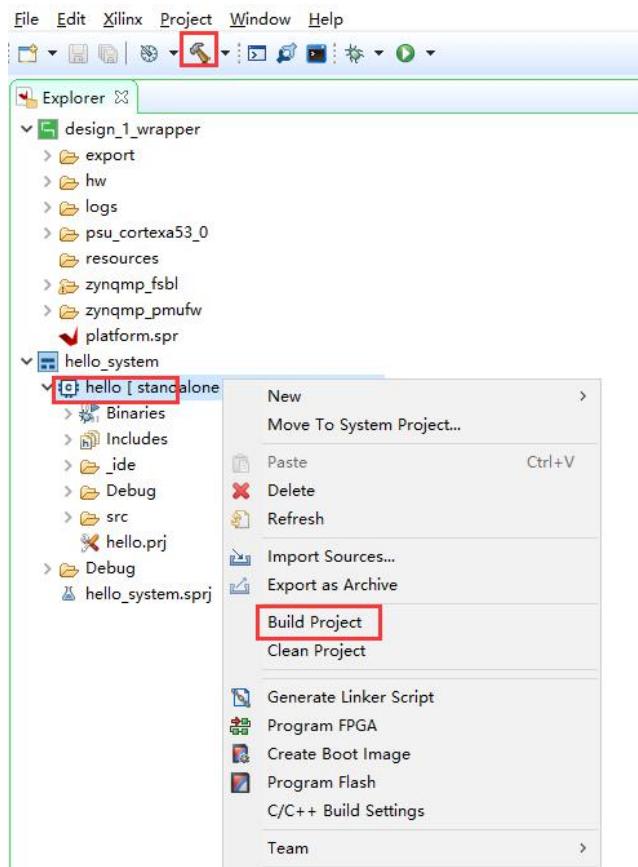
the BSP project and the zynq_fsbl project (this project is the result of selecting Generate boot components). Double-click platform.spr to see the BSP project generated by Platform, where you can configure the BSP. Software developers are more aware that BSP is also the meaning of Board Support Package, 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

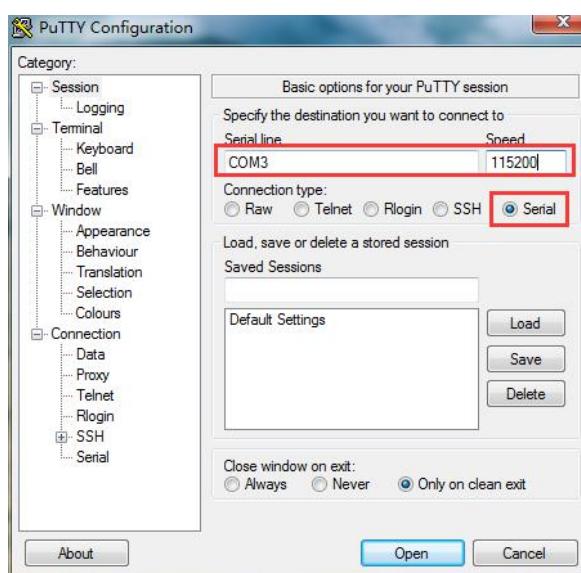
```
Build Console [hello, Debug]
'Finished building: ../src/platform.c'
'
'Building target: hello.elf'
'Invoking: ARM v8 gcc linker'
aarch64-none-elf-gcc -Wl,-T -Wl,../src/lscript.ld -LE:
'Finished building target: hello.elf'
'

'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   c0ec hello.elf
'Finished building: hello.elf.size'
'

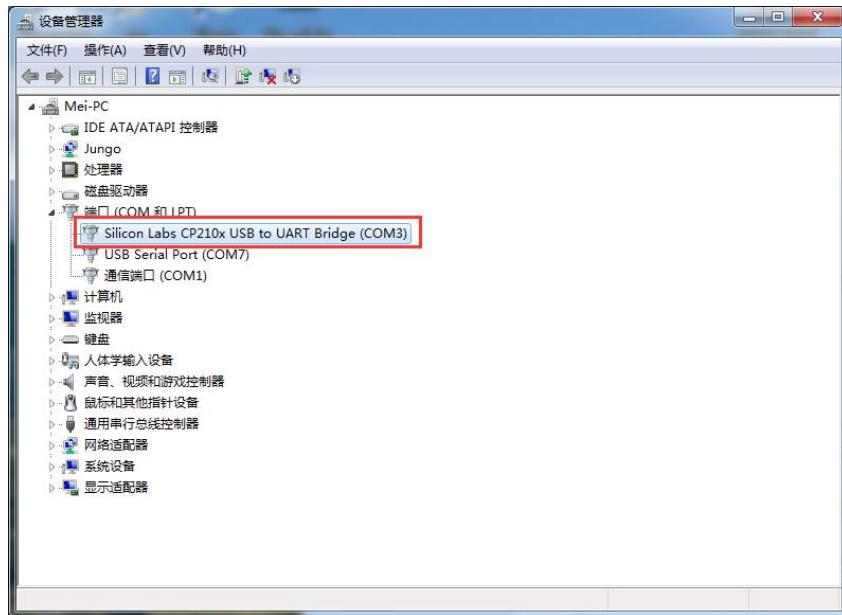
10:51:14 Build Finished (took 1s.121ms)
```

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

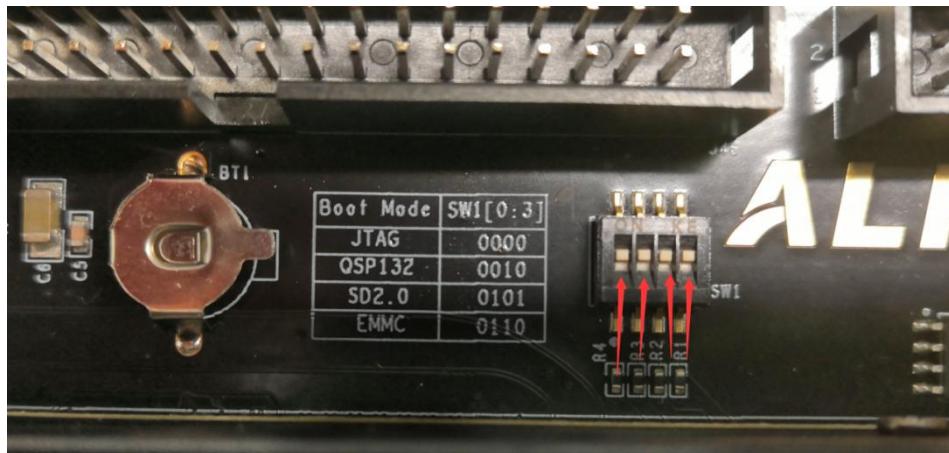
16) Use PuTTY software as a serial port terminal debugging tool, PuTTY is a small software that does not require installation



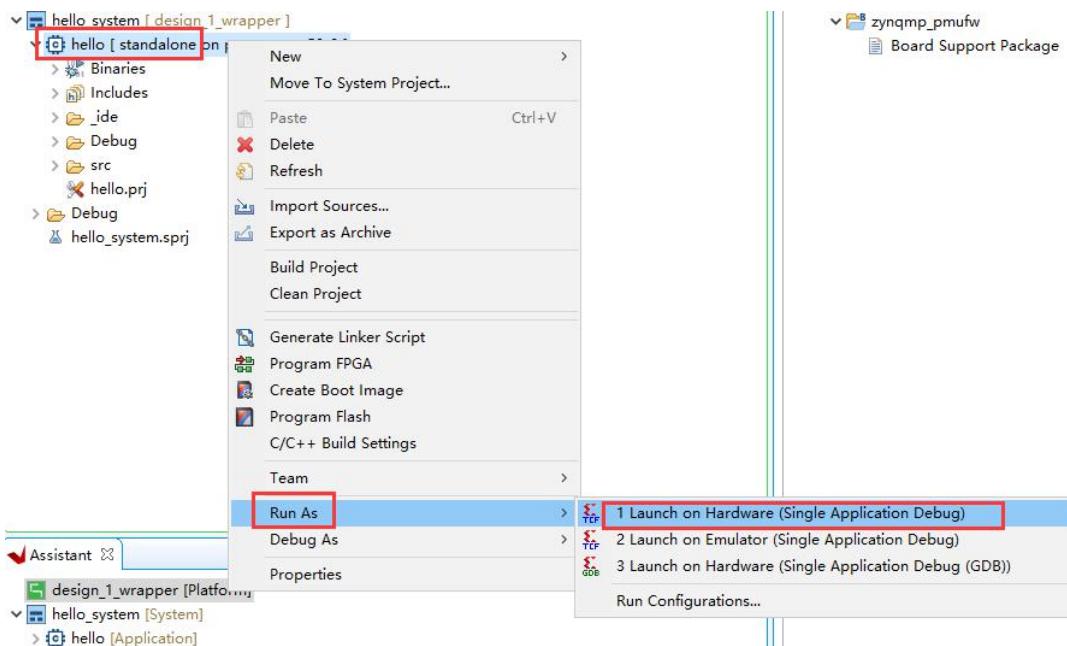
17) Select Serial, fill in COM3 for Serial line, 115200 for Speed, and fill in the COM3 serial port number as shown in the device manager, and click "Open"



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



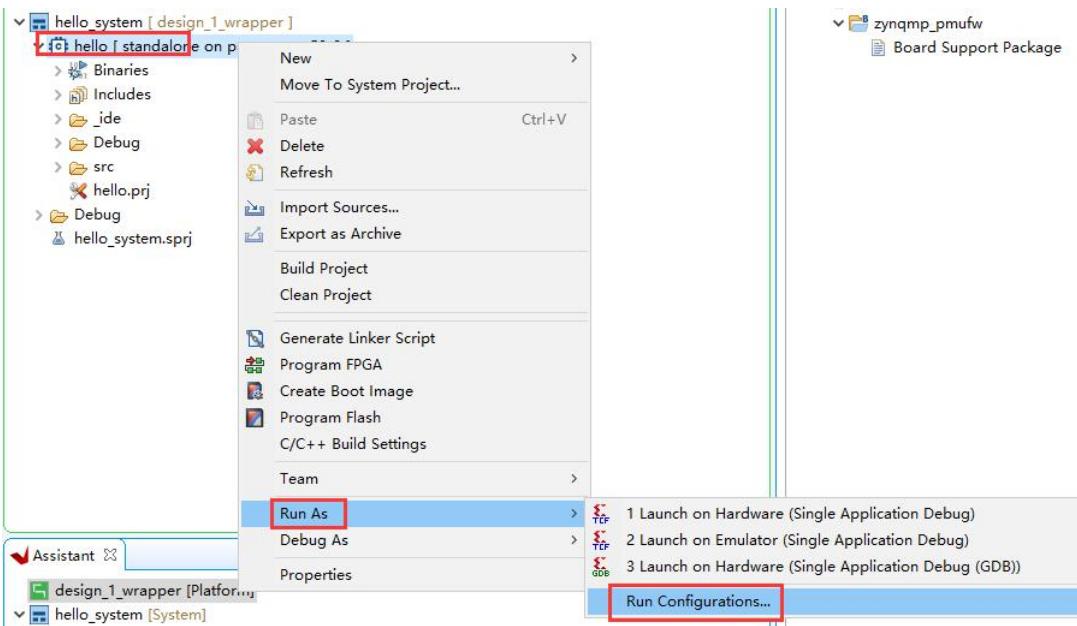
19) Power on the FPGA development board and prepare to run the program. The FPGA development board comes with a program when it leaves the factory. **Here you can select the JTAG mode as the operating mode, and then power on again.** Select "hello", right click, and you can see many options. The "Run 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.



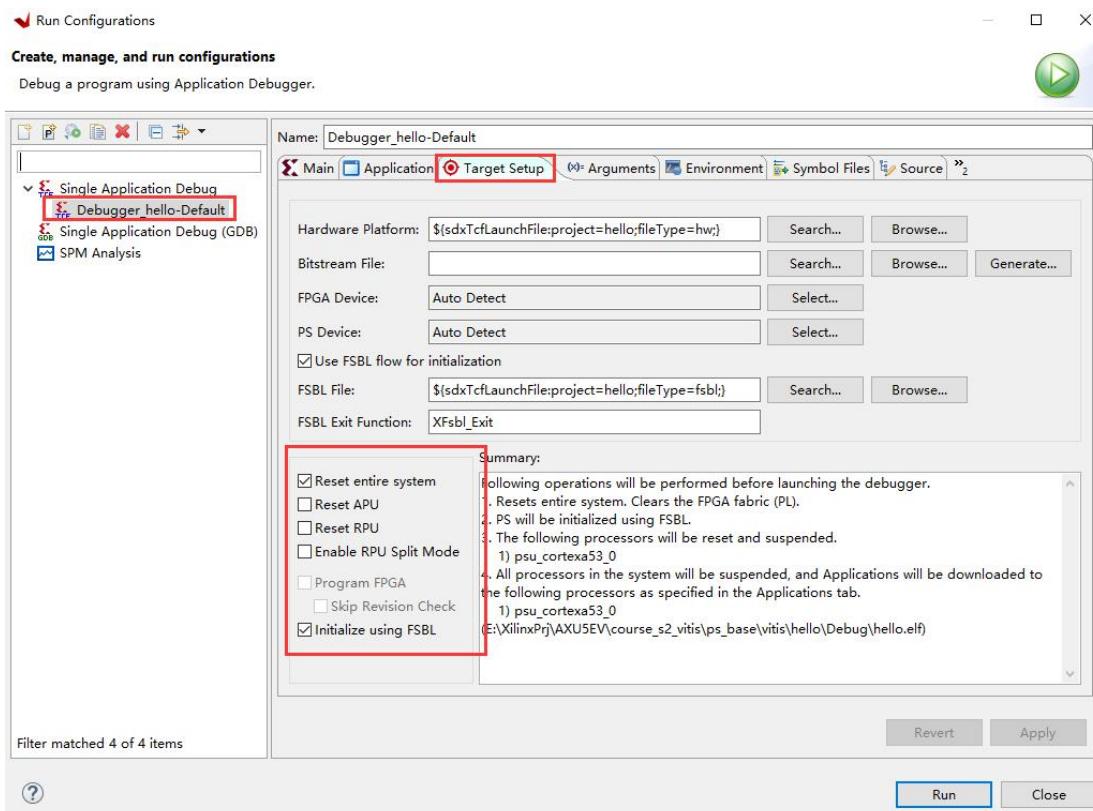
20) Observe the serial port software at this time, you can see the output "Hello World"

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

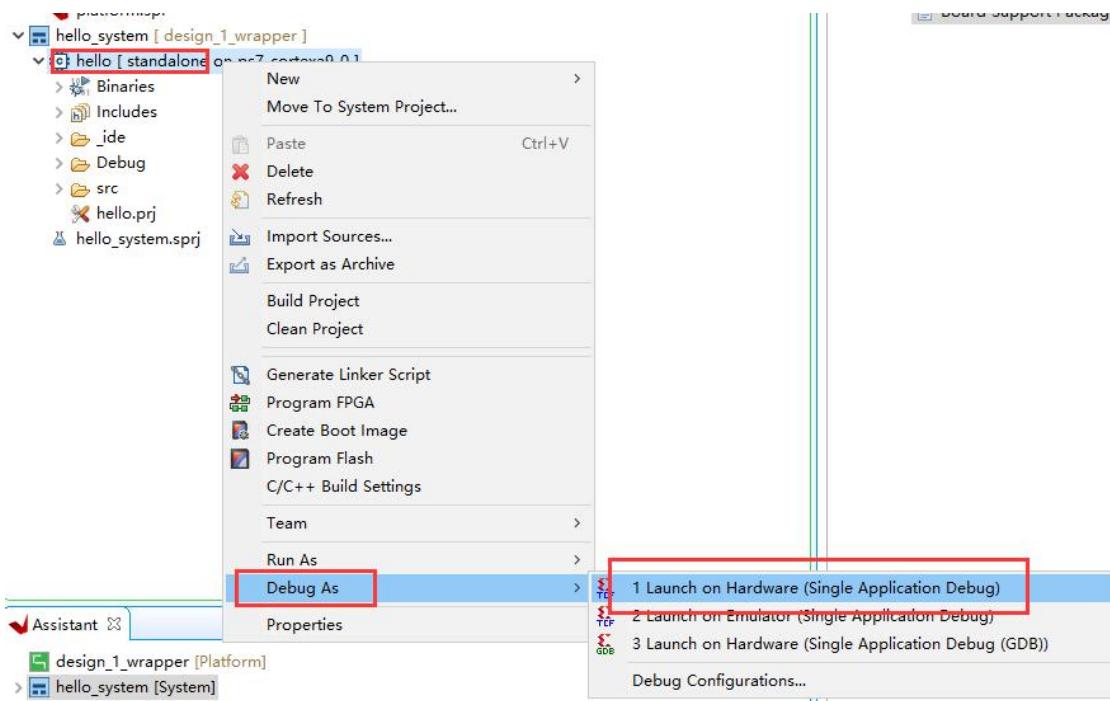
21) In order to ensure the reliable debugging of the system, it is best to right-click "Run As -> Run Configuration..."



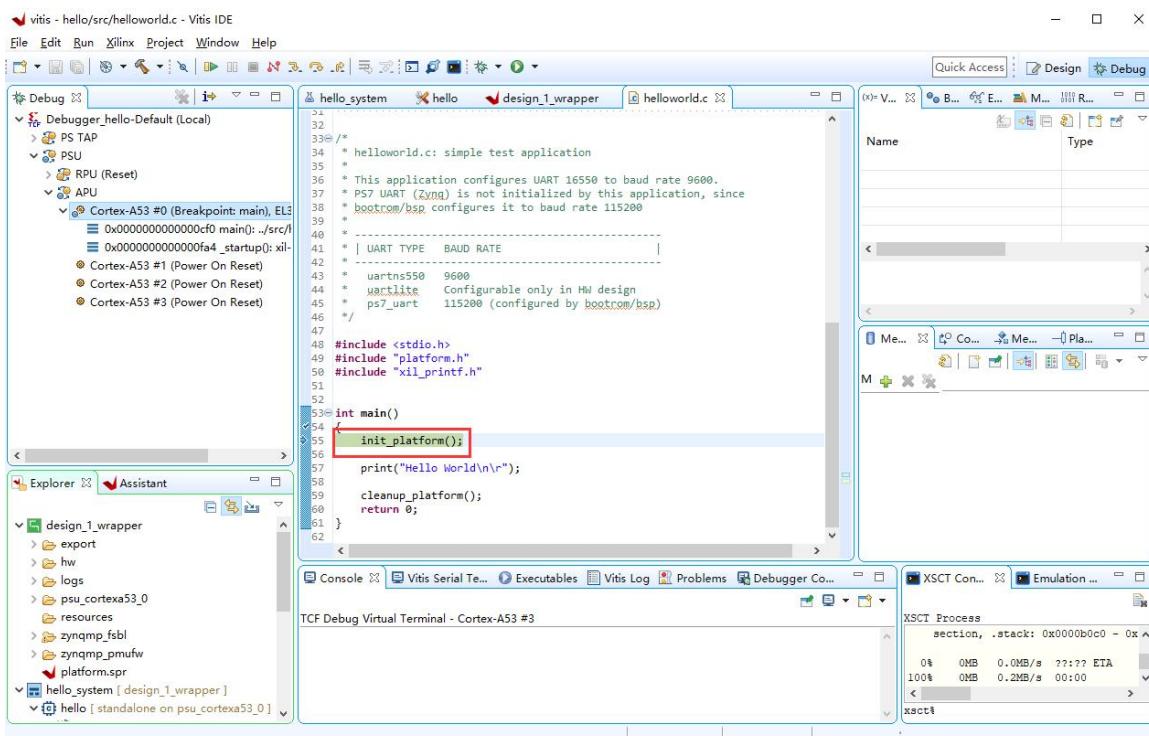
22) We can take a look at the configuration inside, where Reset entire system is selected by default, which is different from the previous SDK software. If there is a PL design in the system, you must also select "Program FPGA".



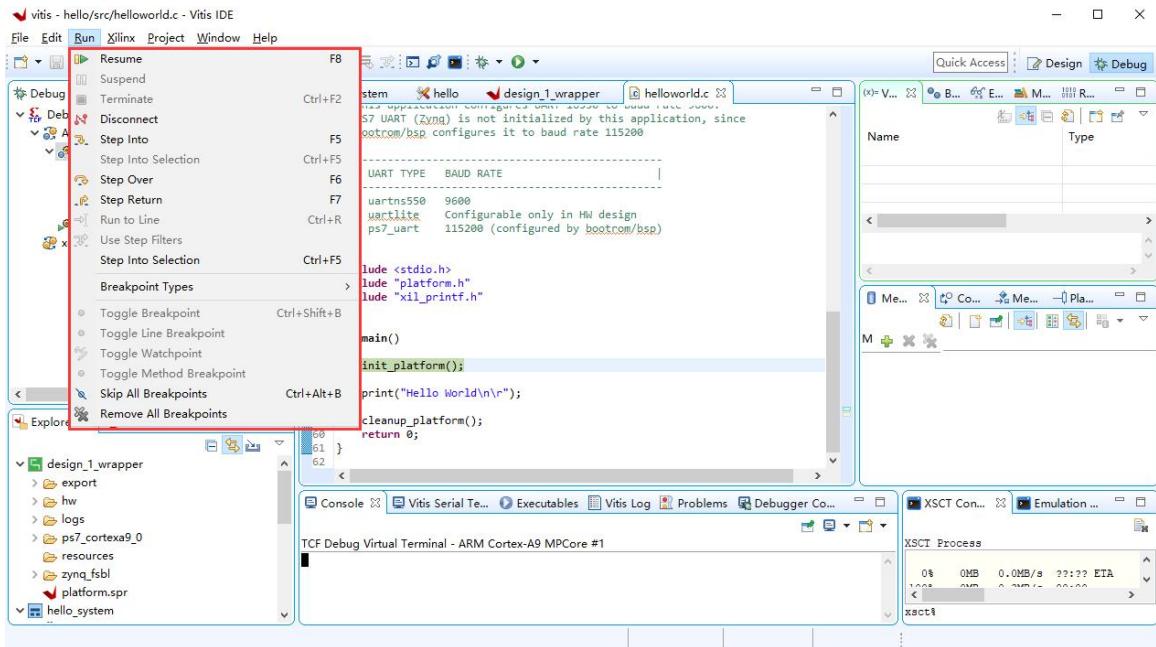
23) In addition to "Run As", you can also "Debug As" so that you can set breakpoints and run in a single step



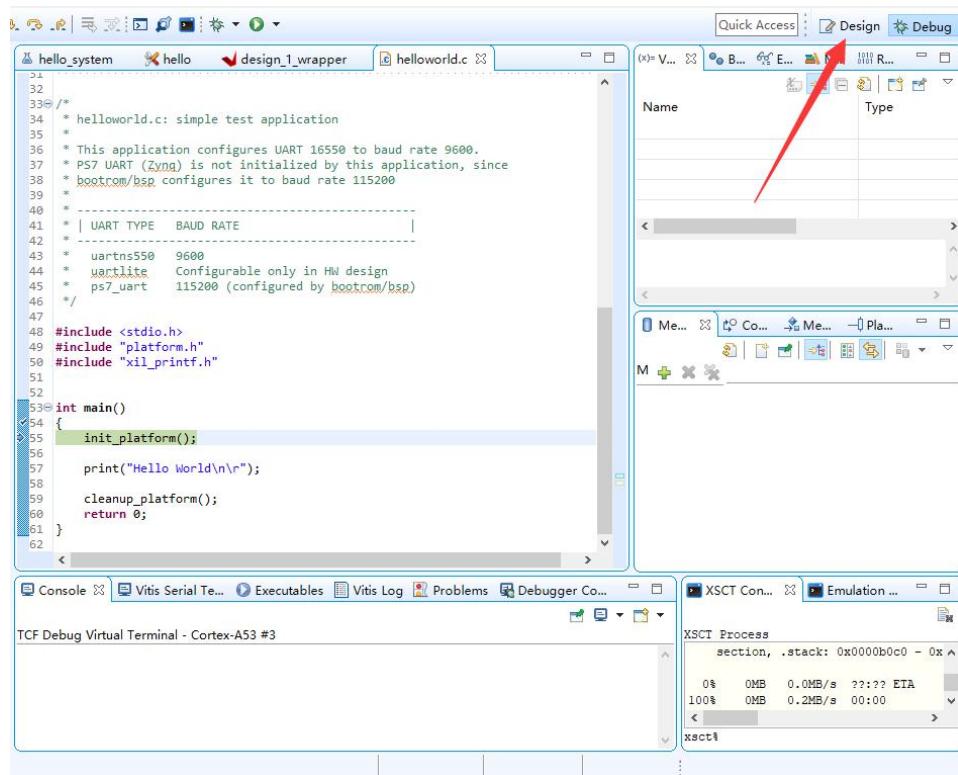
24) Enter Debug mode



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:

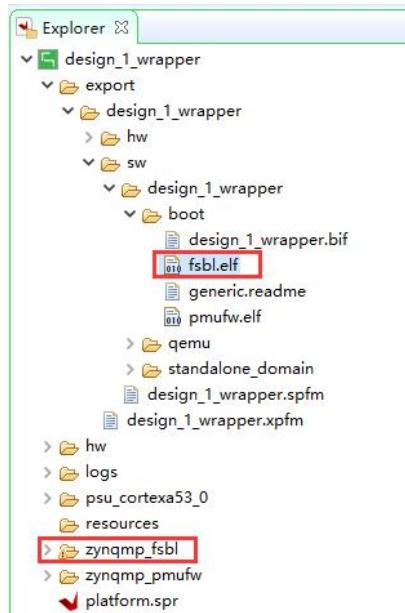
- Initialize PS configuration, MIO, PLL, DDR, QSPI, SD, etc.
- If there is a PL end program, load the PL end bitstream
- Transfer user program to DDR, and jump to execute

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

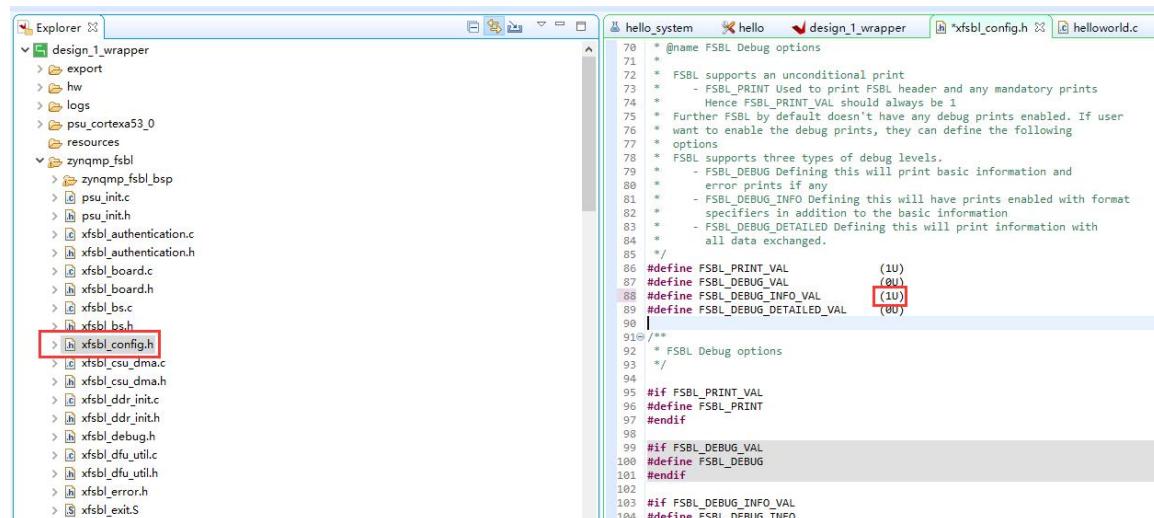
FSBL is a second-level boot program that completes “MIO” allocation, clock, PLL, DDR controller initialization, SD, QSPI controller initialization, finds “bitstream” configuration FPGA through startup mode, then searches for user program to load into DDR, and finally hand over to application carried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

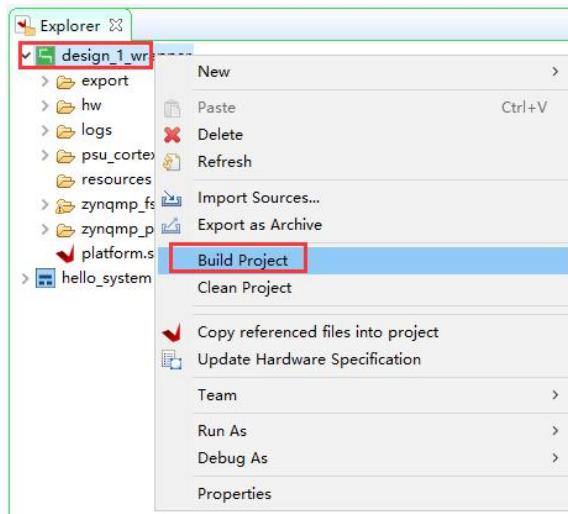


2) Modifying the debugging macro definition

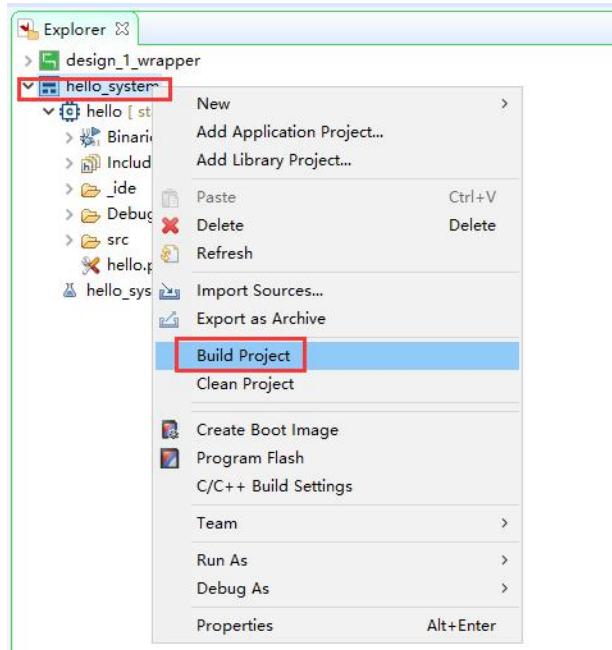
`FSBL_DEBUG_INFO_VAL` can output some status information of FSBL at startup, which is beneficial to debugging, but it will cause the startup time to become longer. save document. You can take a look at the files of many peripherals in fsbl, including `psu_init.c`, `qspi`, `sd`, etc. You can read the code carefully. Of course, this `fsbl` template can also be modified, as for how to modify it according to your own needs.



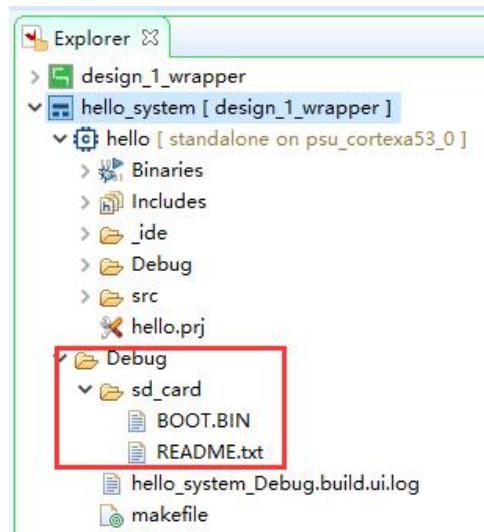
3) Build Project



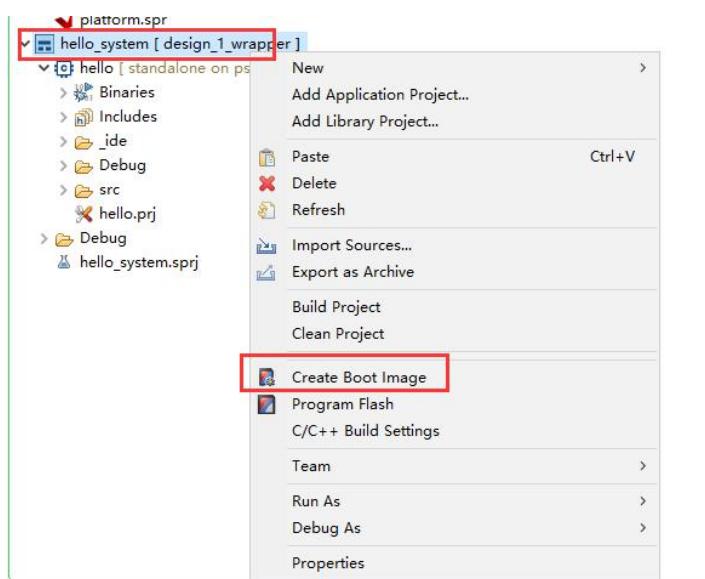
- 4) Next, we can click on the system of the APP project, right-click and select Build project

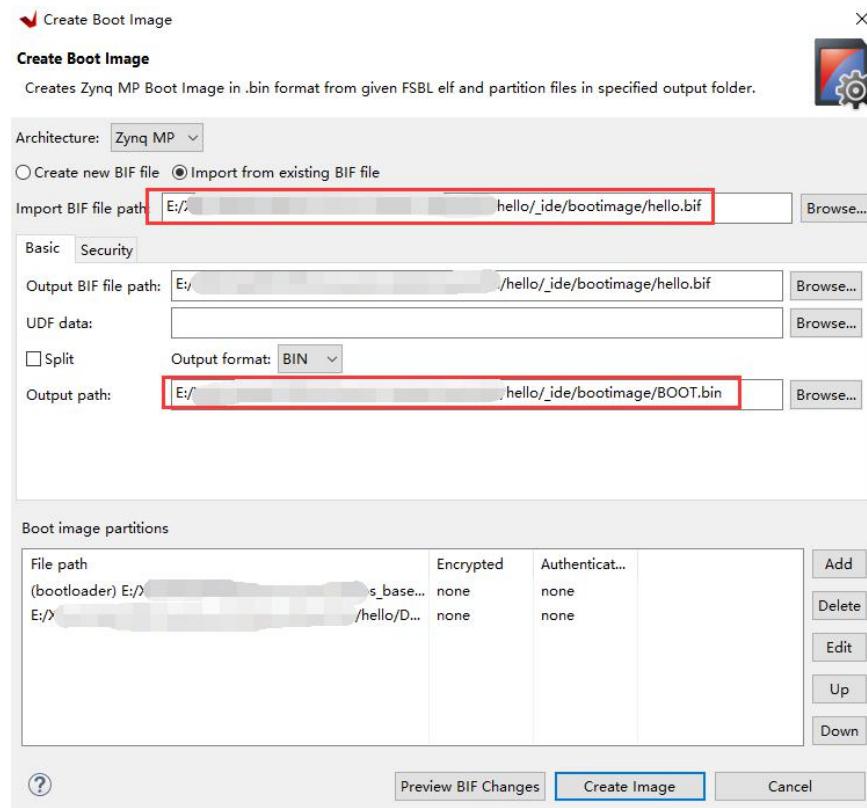


- 5) At this time, there will be an extra Debug folder, and the corresponding BOOT.BIN will be generated

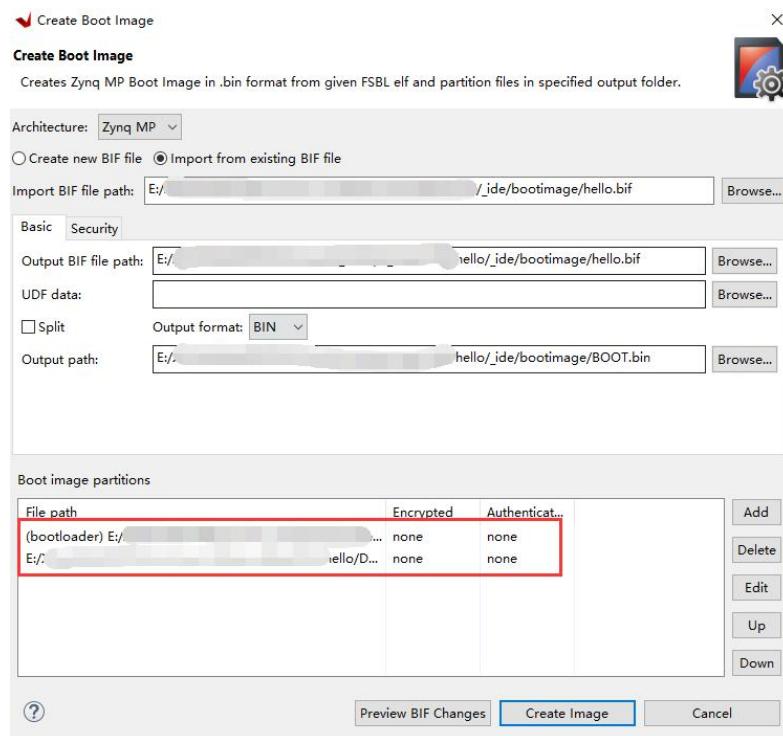


- 6) Another way is to click on the system of the APP project, right-click and select Creat Boot Image. In the pop-up window, you can see the path of the generated BIF file. The BIF file is the configuration file for generating the BOOT file, as well as the path of the generated BOOT.bin file. , The 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

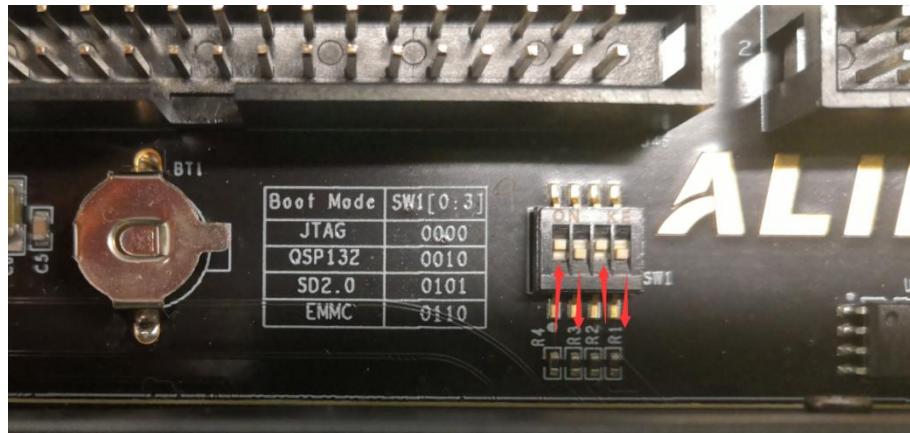
1) Format SD card, can only be formatted as FAT32, other formats cannot be started



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



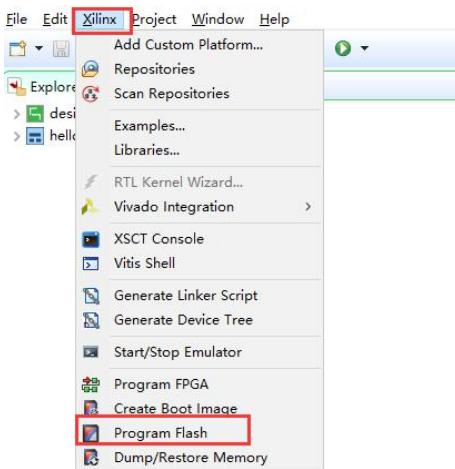
- 5) Open the serial port software, power on and start, you can see the print information, the red box is the FSBL startup information, the yellow arrow part is the executed application “helloworld”

```
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
```

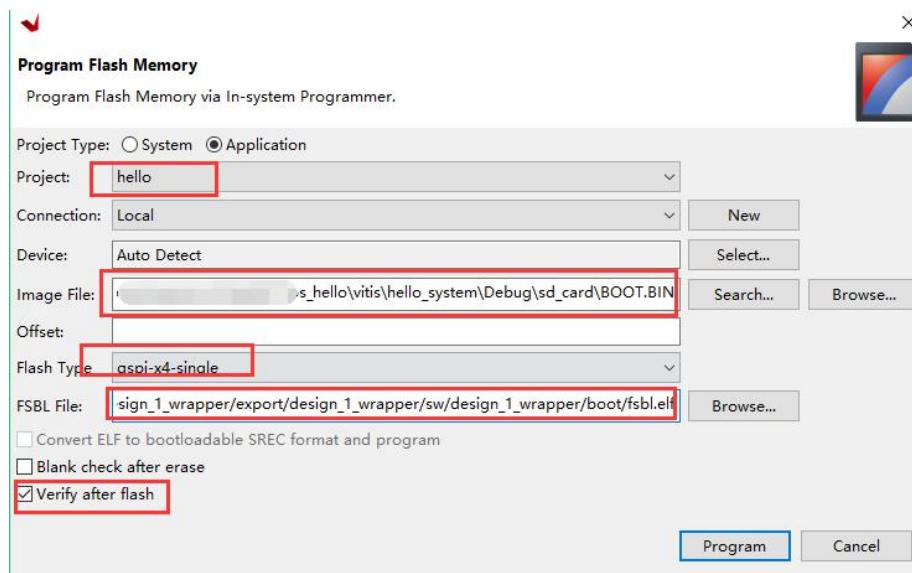
A screenshot of a terminal window showing the boot process of a Zynq MP system. The text is divided into several sections by horizontal lines. The first section shows the FSBL startup information, which is highlighted with a red rectangular box. The last section shows the execution of the "helloworld" application, which is highlighted with a yellow arrow pointing to the text "Successfully ran Hello World application".

Part 1.4.3: QSPI Startup Test

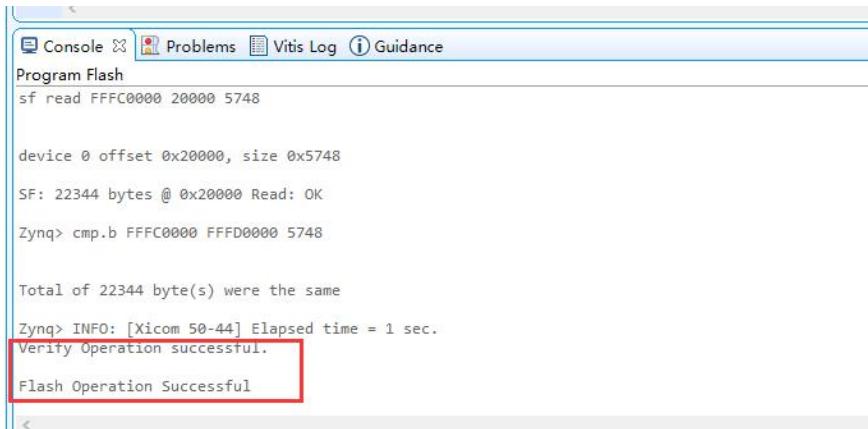
- 1) In the Vitis menu “Xilinx-> Program Flash”



- 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



The screenshot shows a terminal window with the following text:

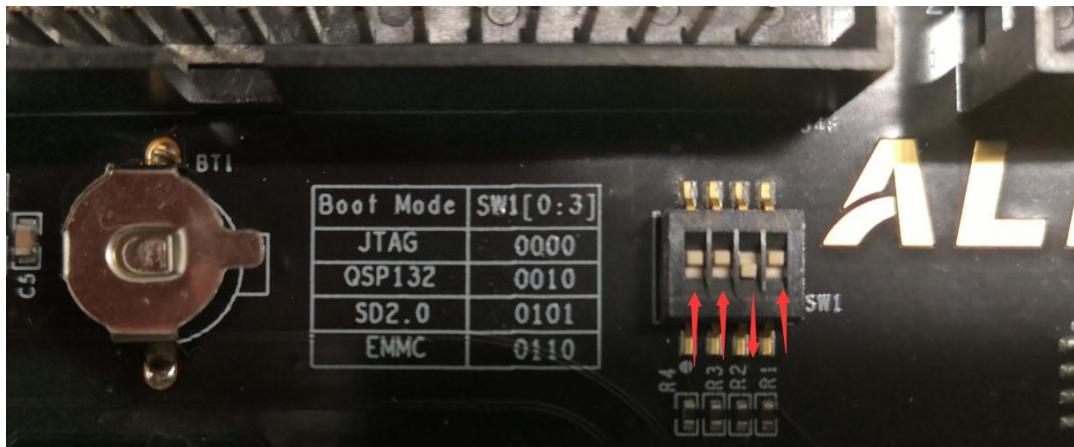
```
Console ✘ Problems Vitis Log Guidance
Program Flash
sf read FFFC0000 20000 5748

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
```

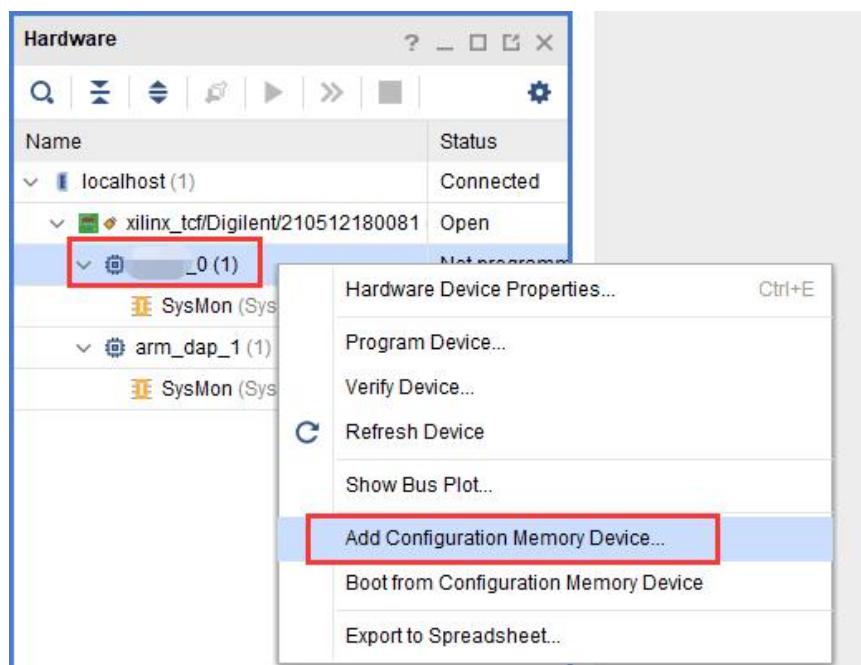
- 4) Set the startup mode to QSPI and start again, You can see the same startup effect as SD in the serial port software.



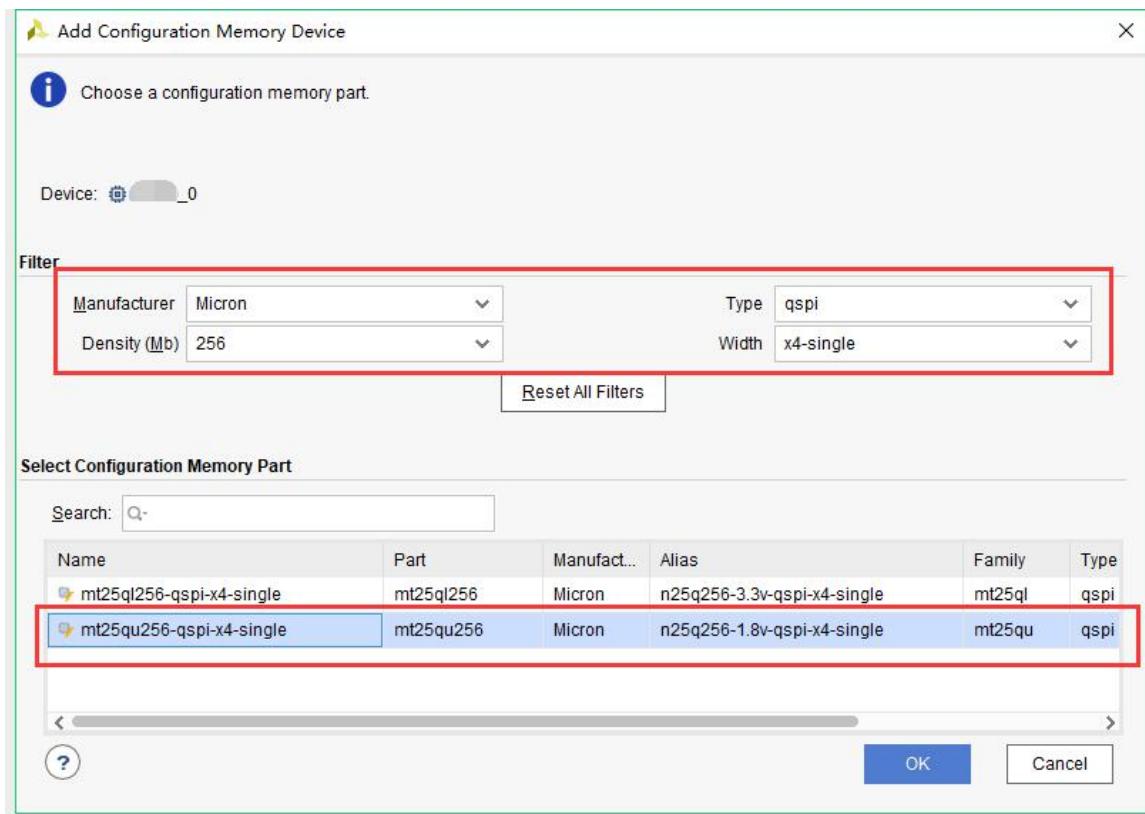
```
Xilinx Zynq MP First Stage Boot Loader
Release 2020.1 Jul 8 2020 - 14:22:13
Reset Mode : System Reset
Platform: Silicon (4.0), Cluster ID 0x80000000
Running on A53-0 (64-bit) Processor, Device Name: XCZU4EV
Processor Initialization Done
==== In Stage 2 ====
QSPI 32 bit Boot Mode
QSPI is in single 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

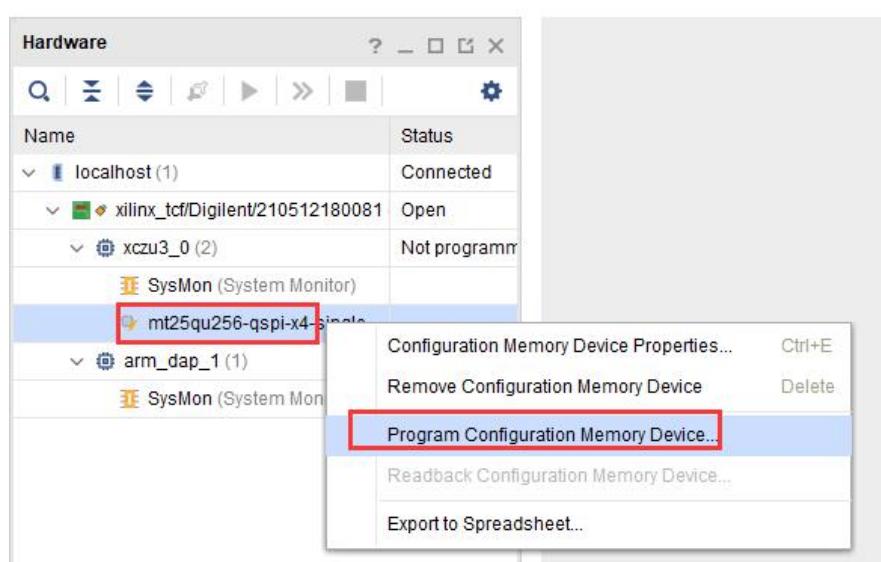
- 1) Select the device under “HARDWARE MANGER”, right-click “Add Configuration Memory Device”



- 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.

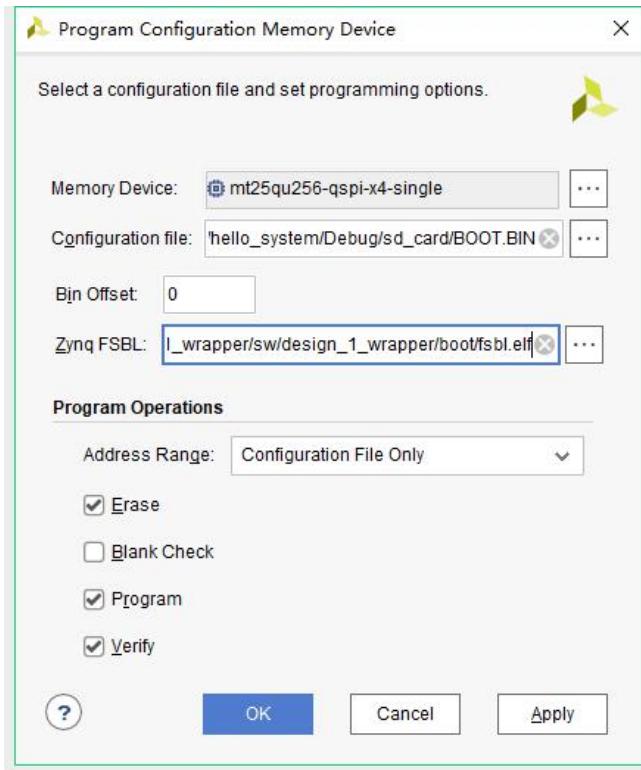


- 3) Right-click and select the programming file



- 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...

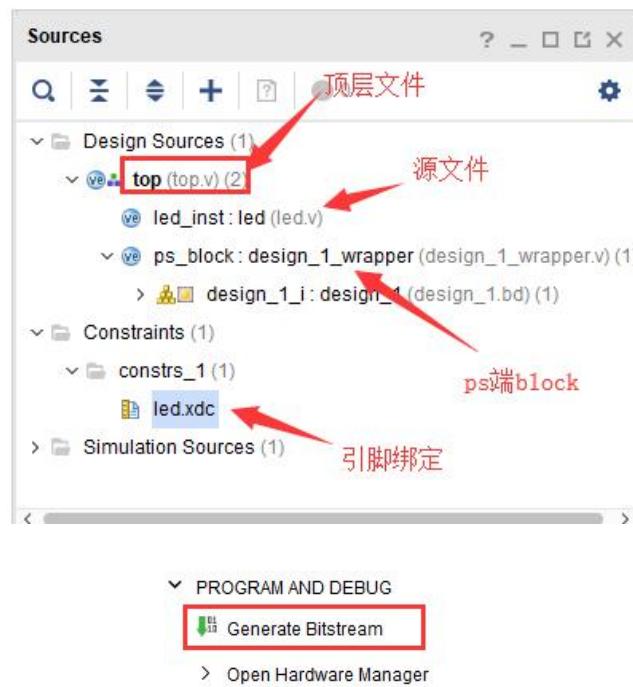
Initialization done, programming the memory
Using default mini u-boot image file - F:/Xilinx_Vitis/Vitis/2019.2/data\xicom\cfgmem\uboot\zynq_qspi_x4_single.bin
===== mrd->addr=0xF800025C, data=0x00000001 =====
BOOT_MODE REC = 0x00000001
WARNING: [Xicom 50-100] The current boot mode is QSPI.
If flash 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

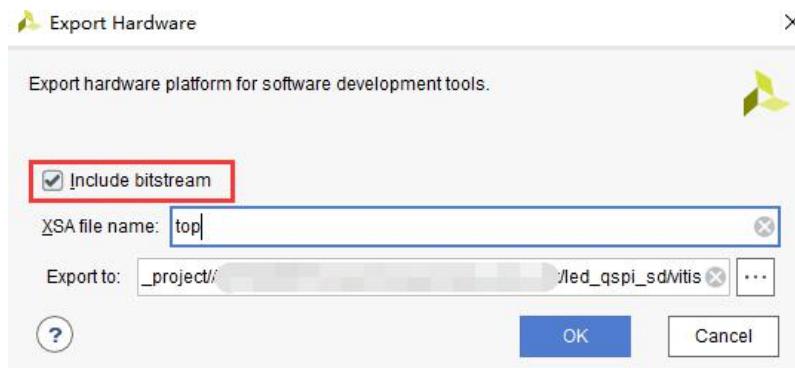
Part 1.5.1: Only PL side Logic Solidification

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"?

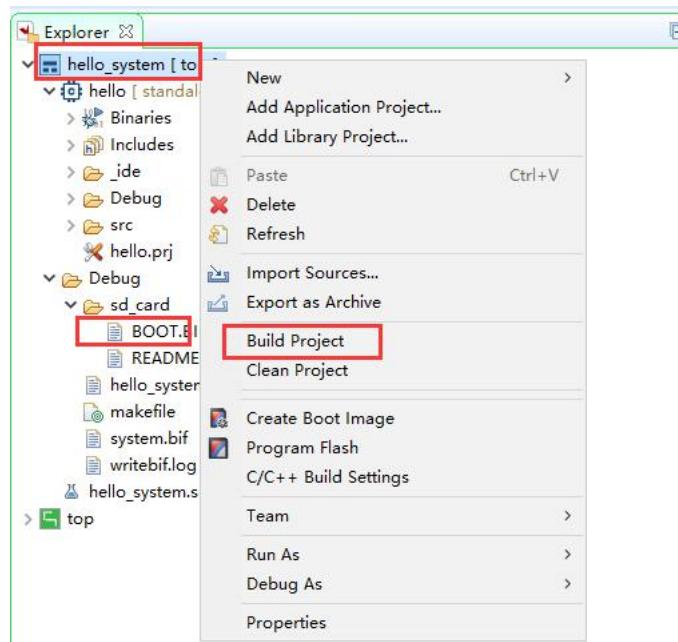
- 1) According to the PS side of this chapter, add the ZYNQ core and configure it. The easiest way is to add the verilog source file of the LED experiment on the basis of the project in this chapter and instantiate it to form a system, and the bitstream needs to be generated.



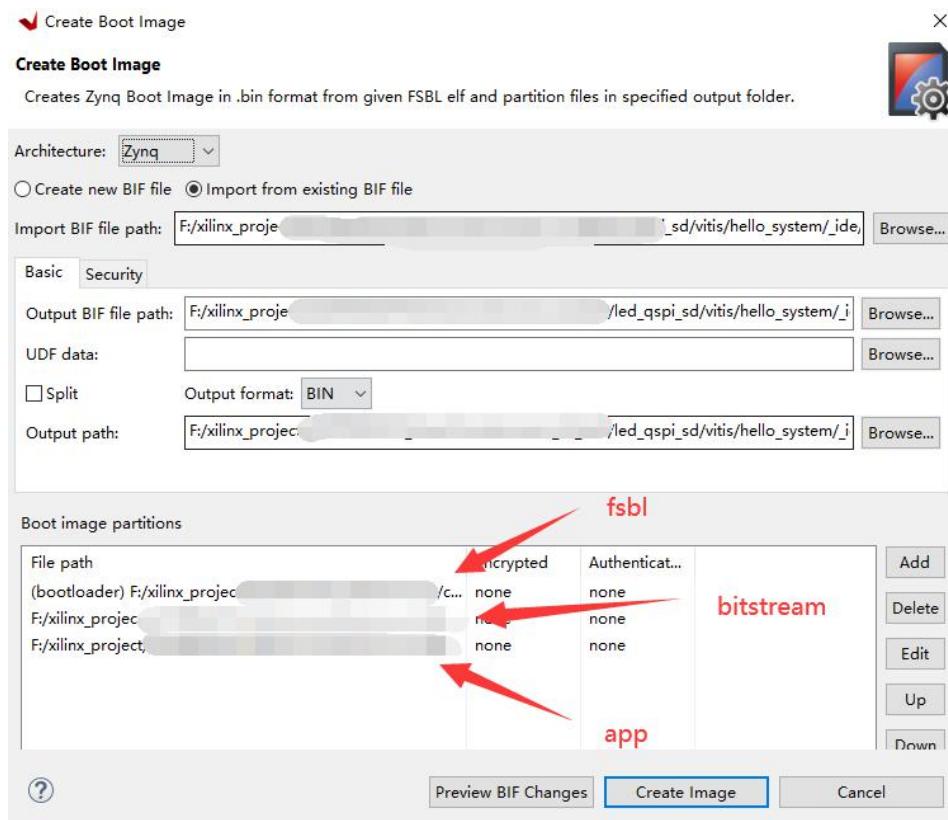
- 2) After generating the bitstream, export the hardware and select "include bitstream"



- 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.



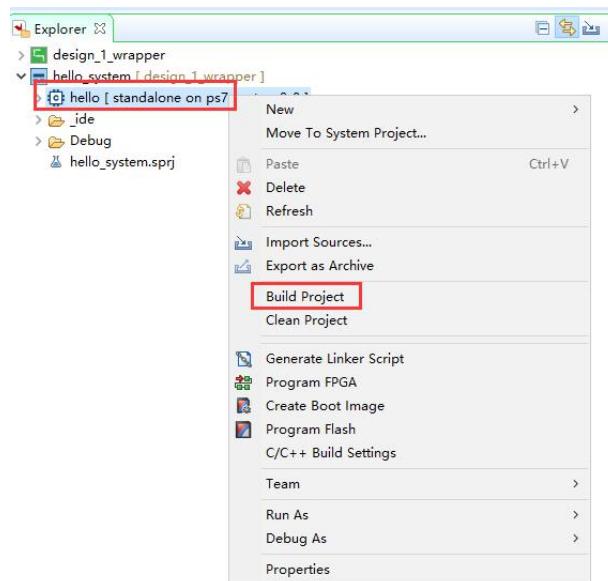
- 4) Open the Create Boot Image interface and you can see, the file order of the “Boolmage Partitions” is “fsbl”, “bitstream”, “app”, pay attention not to reverse the order. Using the BOOT.BIN generated in this way, you can test and start according to the previous startup method.



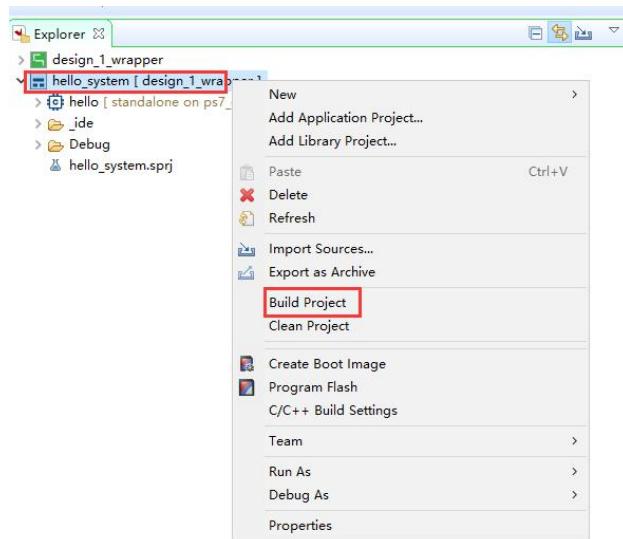
- 5) In the “course_s2” folder, a project named “led_qspi_sd” is provided for your reference.

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

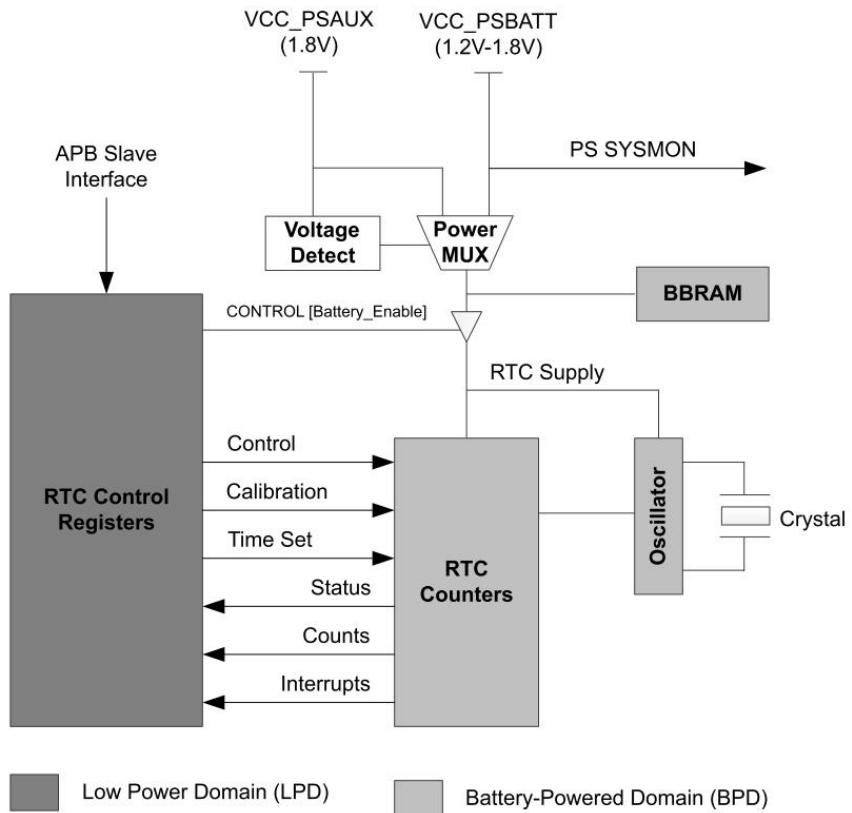
The real-time clock unit provides an accurate time reference for the system and application software. In order to meet the need for high precision, the real-time clock also includes a calibration circuit to compensate for temperature and voltage fluctuations. RTC is powered by VCC-PSAUX or VCC-PSBATT 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

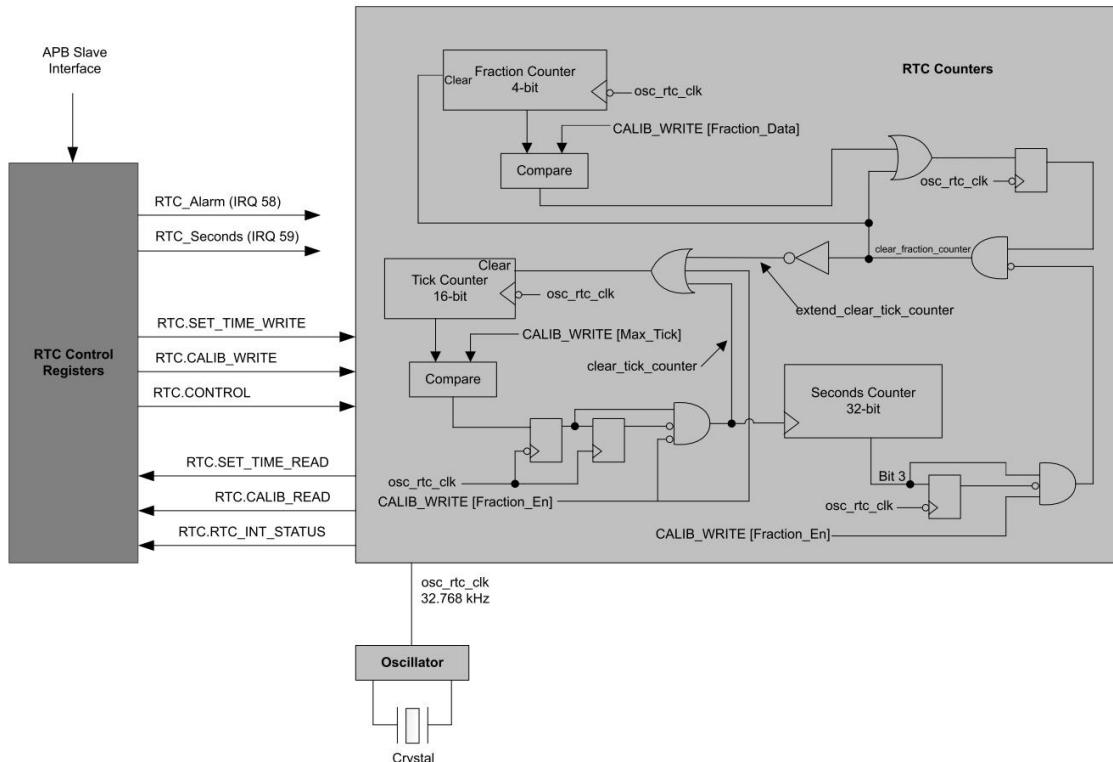
Time seconds counter, 32 bits, can count about 136 years

32 KHz reference clock counter, which means 1 second count

4-digit fraction counter for calibration



RTC Controller Block Diagram

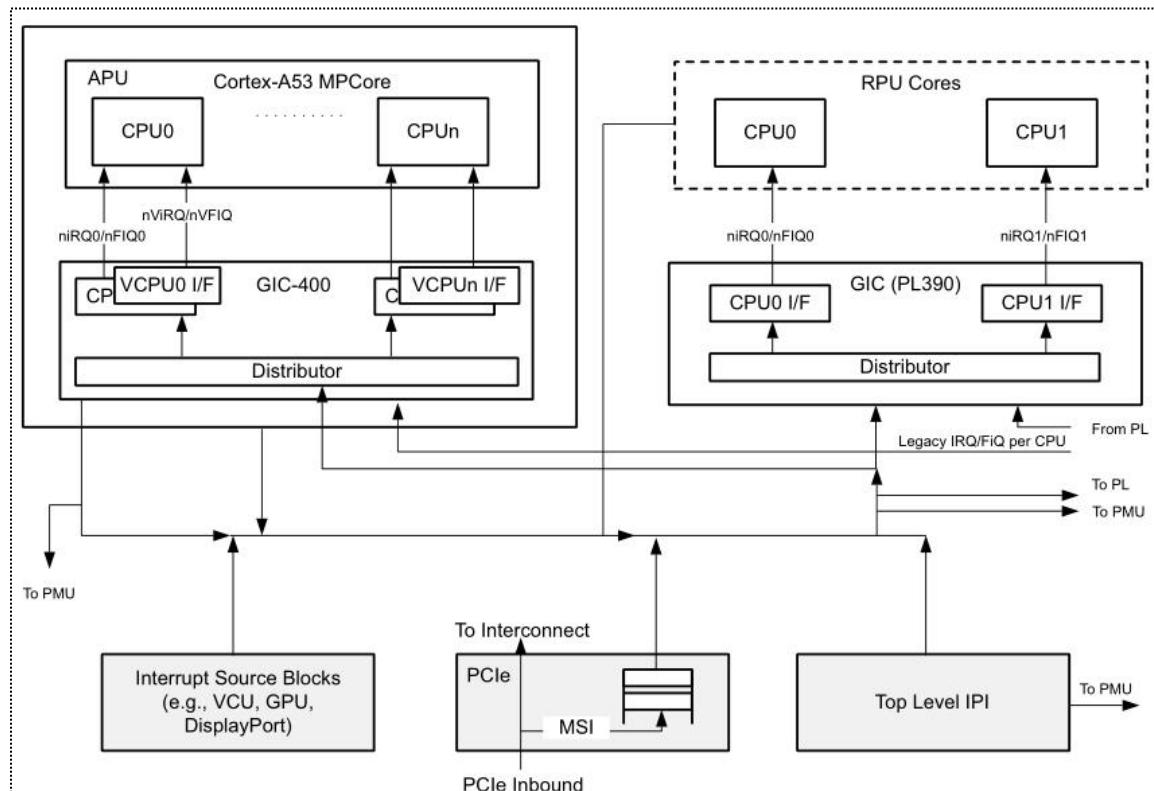


RTC Block Diagram

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

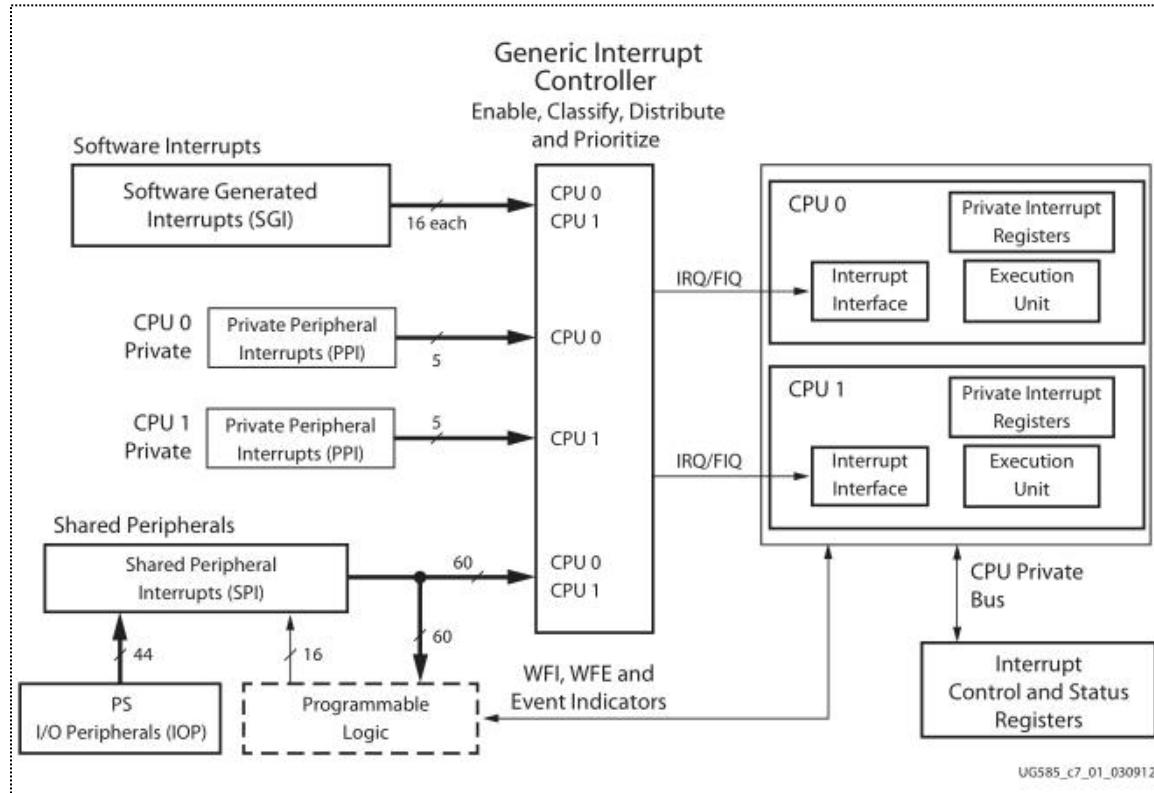
The figure contains two GICs:



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:



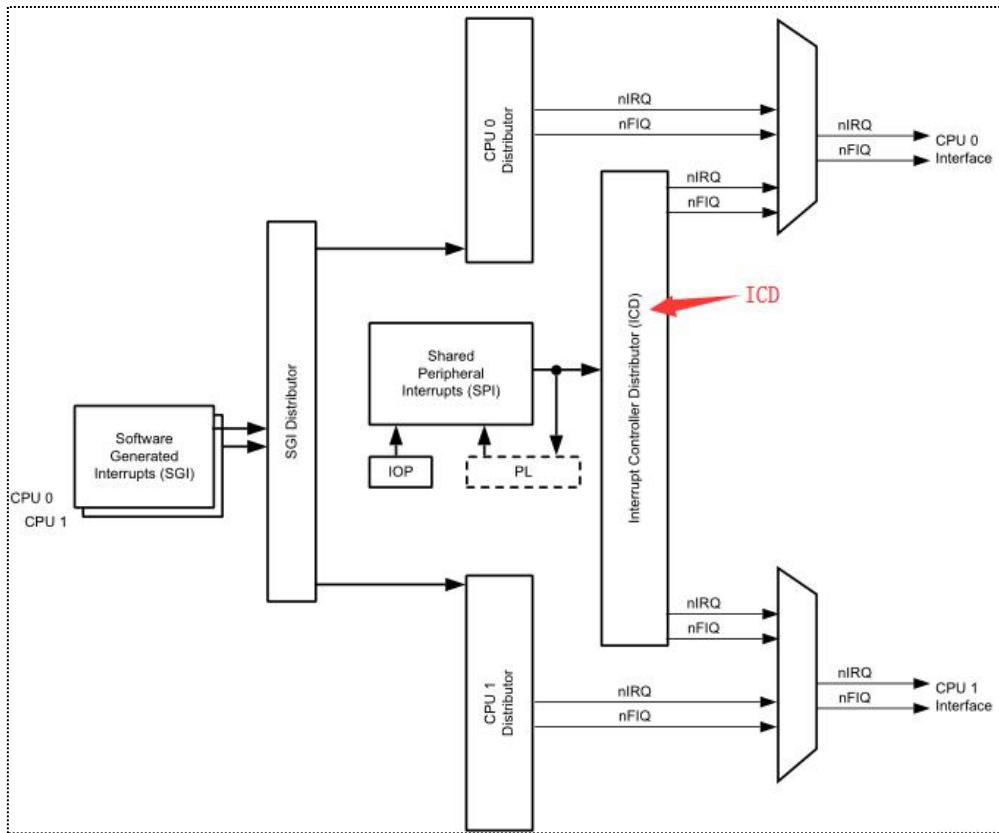
As can be seen from the figure, there are three main interrupt sources:

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

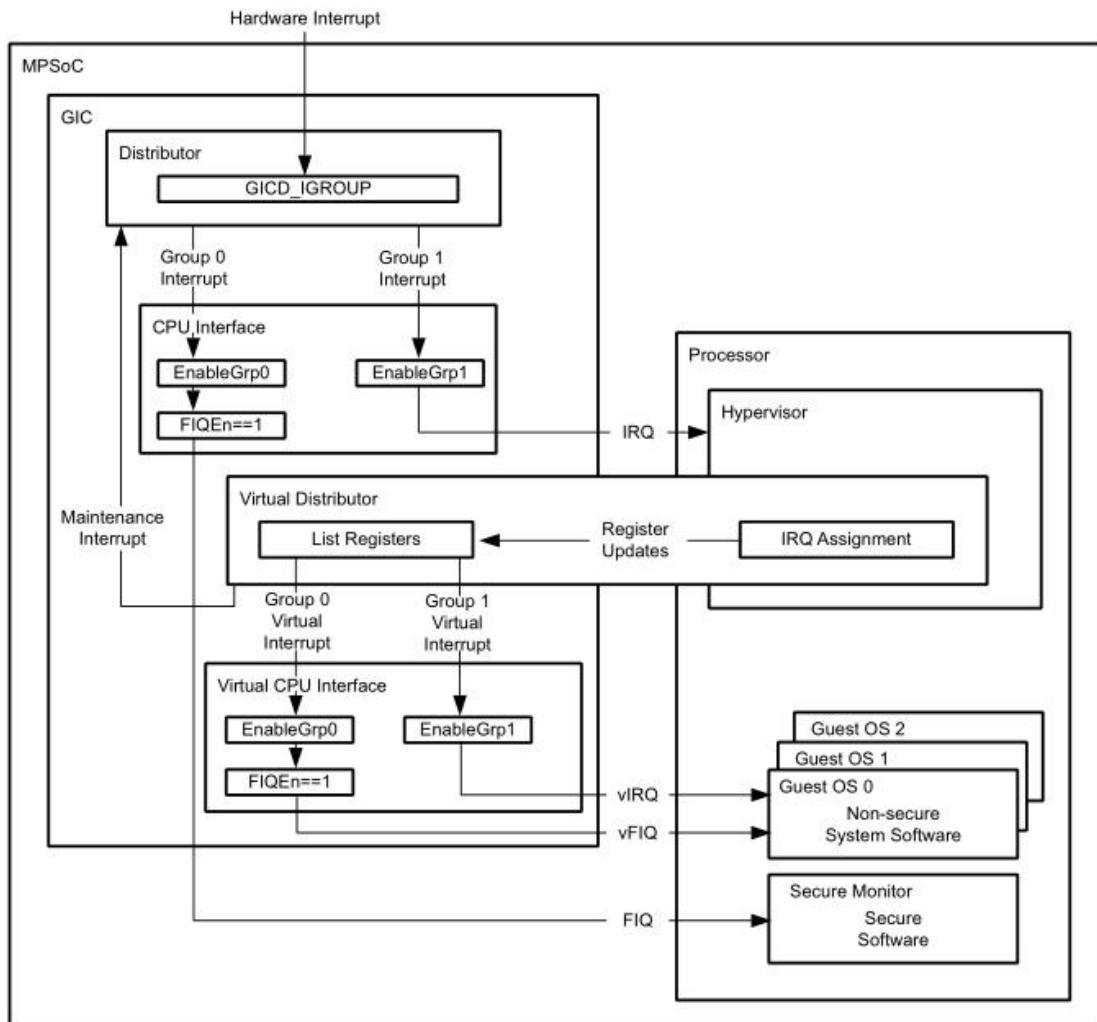
The functional block diagram of the controller is as follows:



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

Starting Address	Register Set	Count	Description
RPU - Private CPU Bus for RPU MPCore			
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.

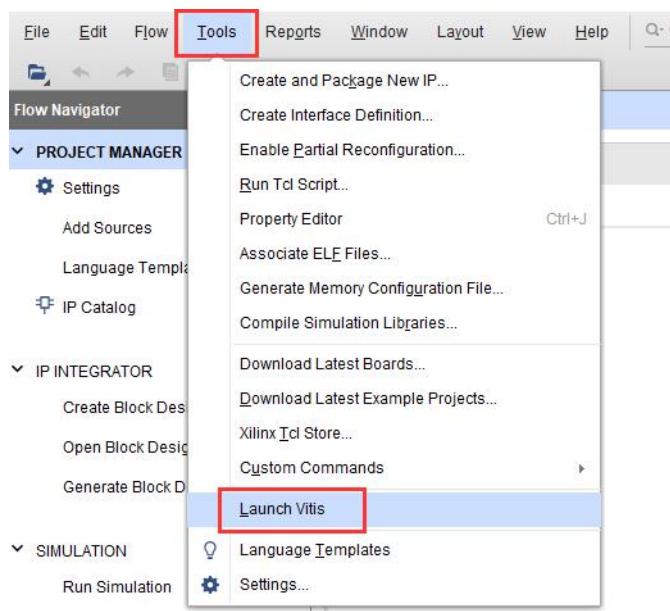
Software Engineer Job Content

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

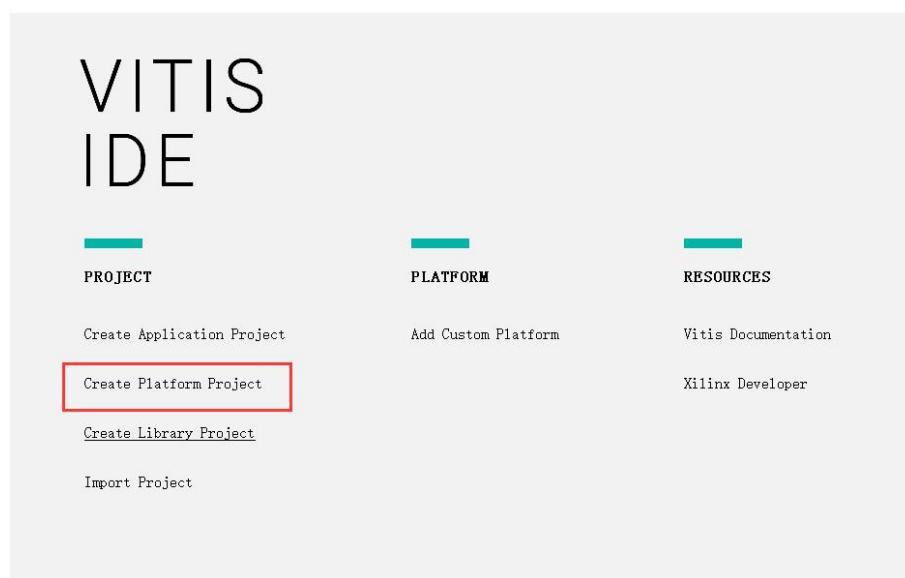
Part 2.3: Vitis Programming

Part 2.3.1: Create Platform Project

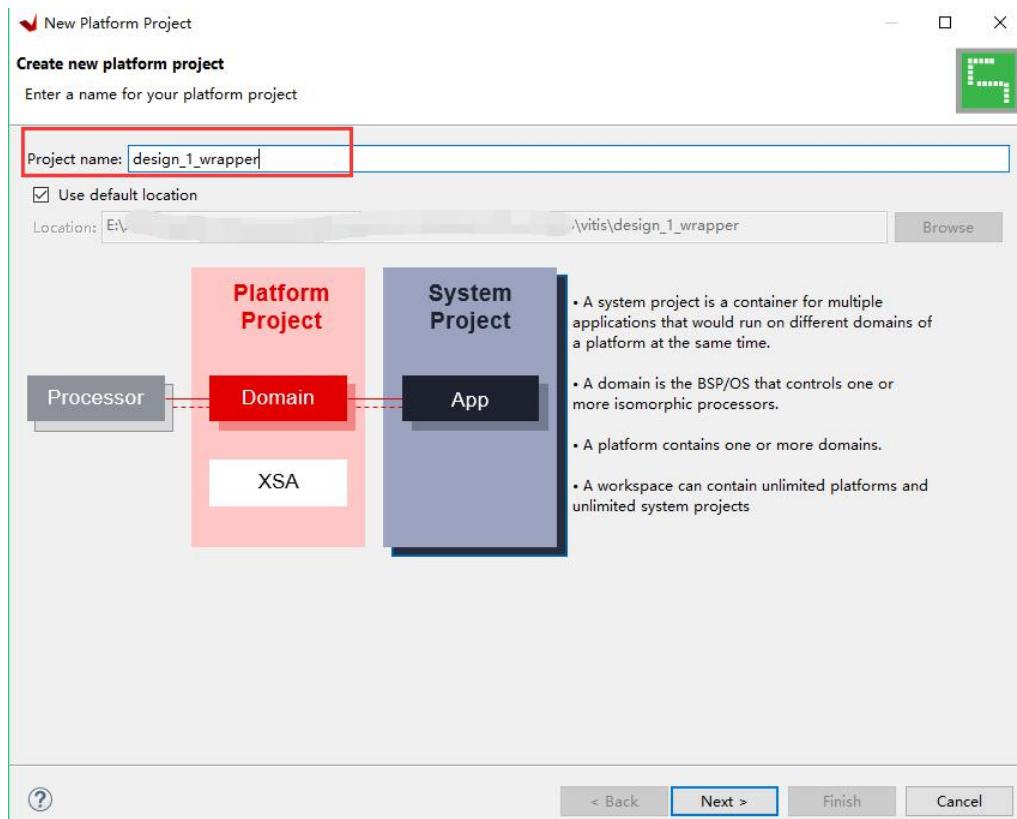
- 1) Click Tools→Launch Vitis



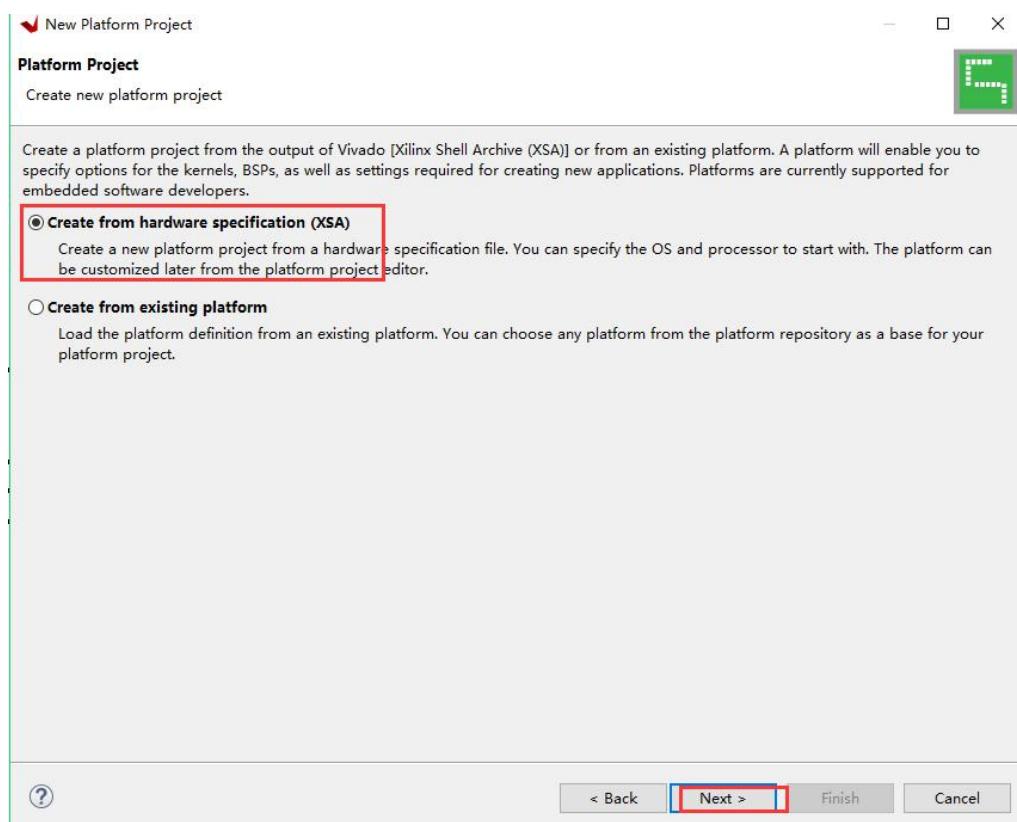
- 2) Unlike the previous Hello World experiment, we only build the Platform project



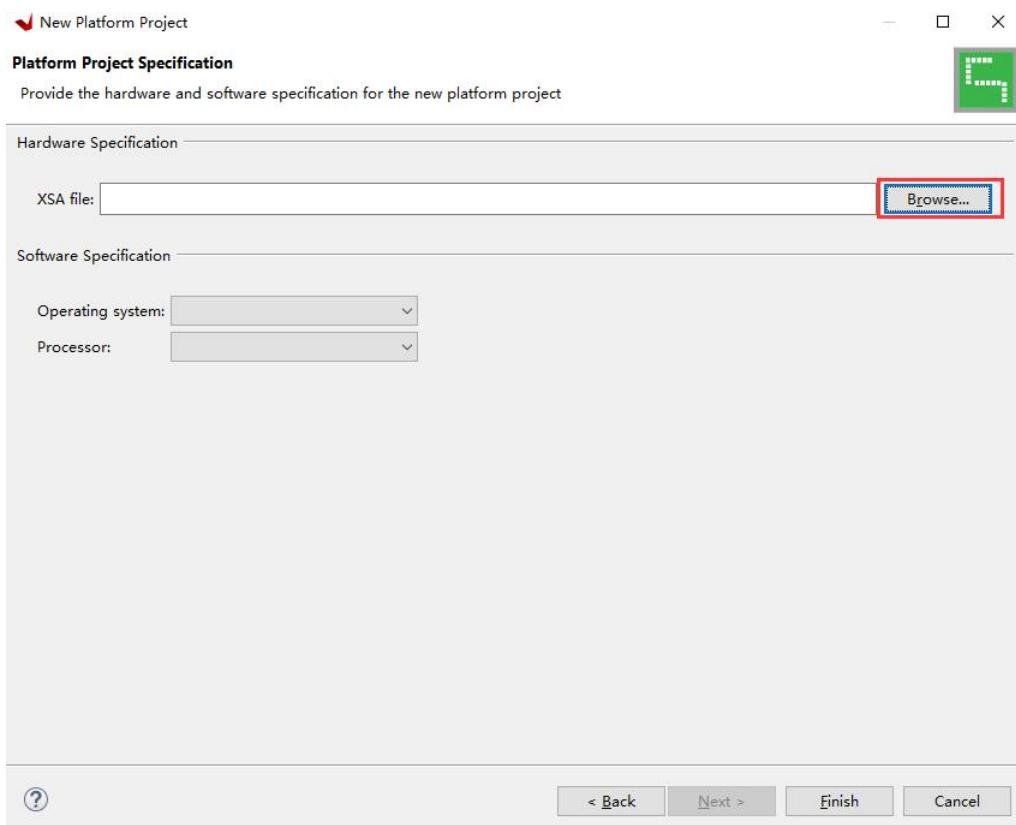
- 3) Fill in the project name, which should be the same as the name of the XSA file, click Next



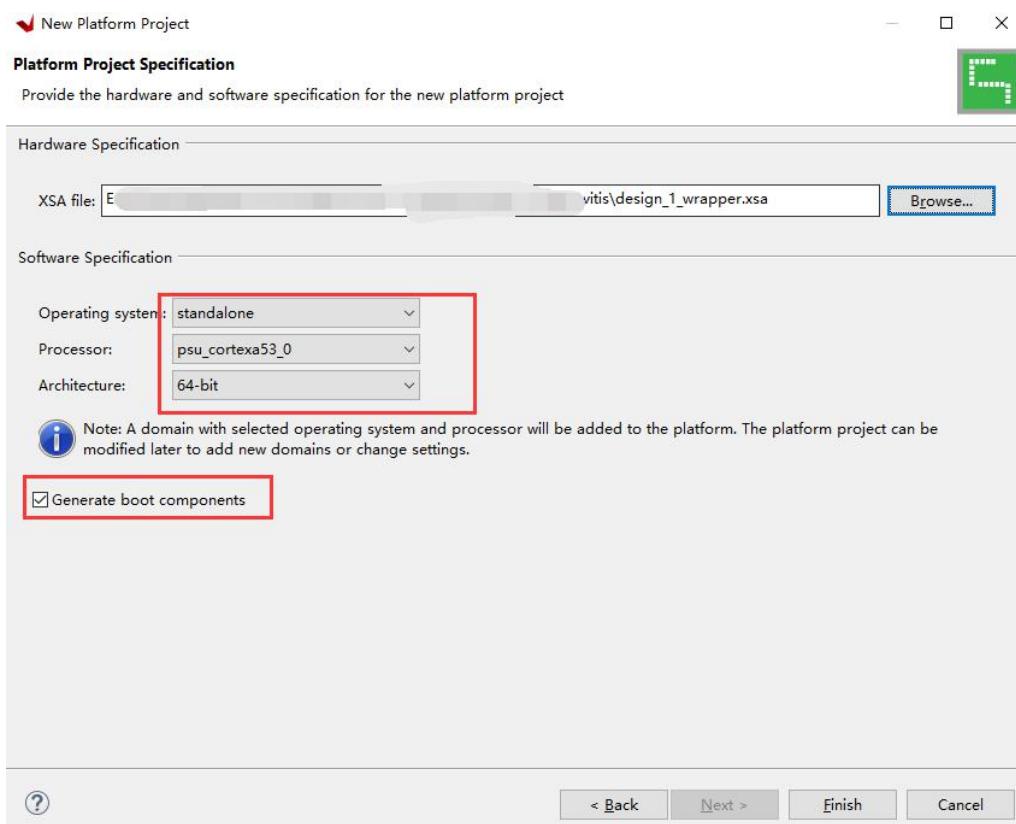
- 4) Click Next



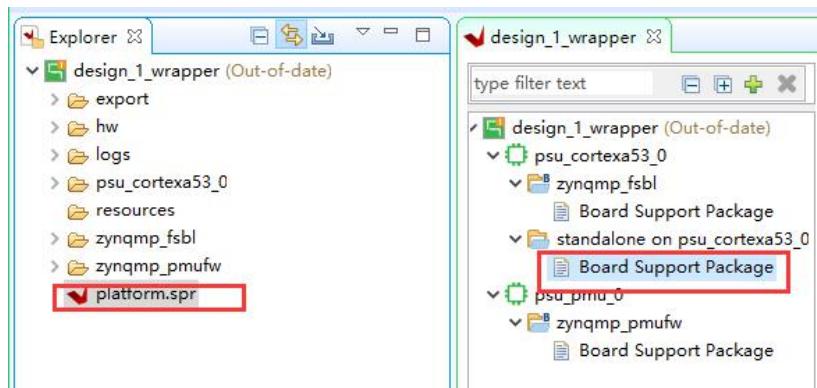
5) Select XSA file



Keep the default, click Finish



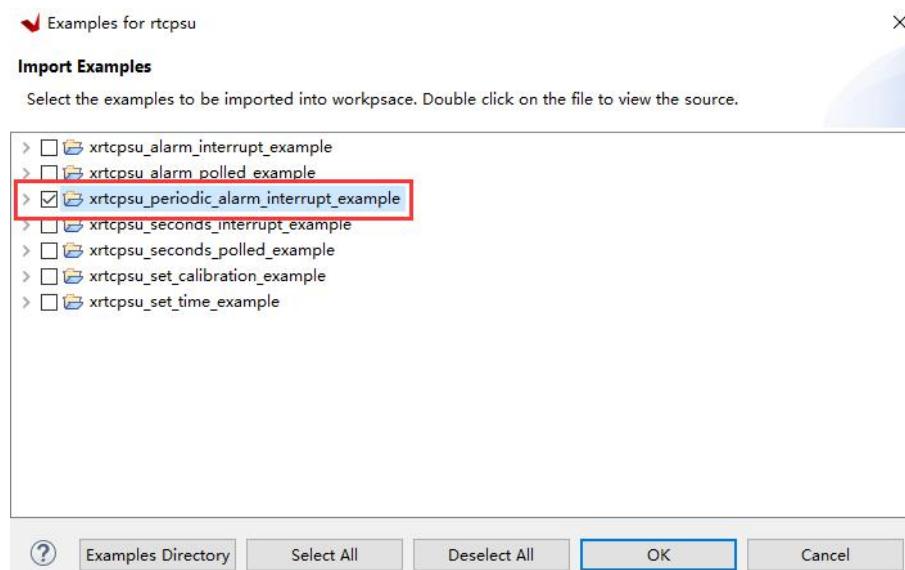
- 6) Click to open platform.spr, and click to open BSP



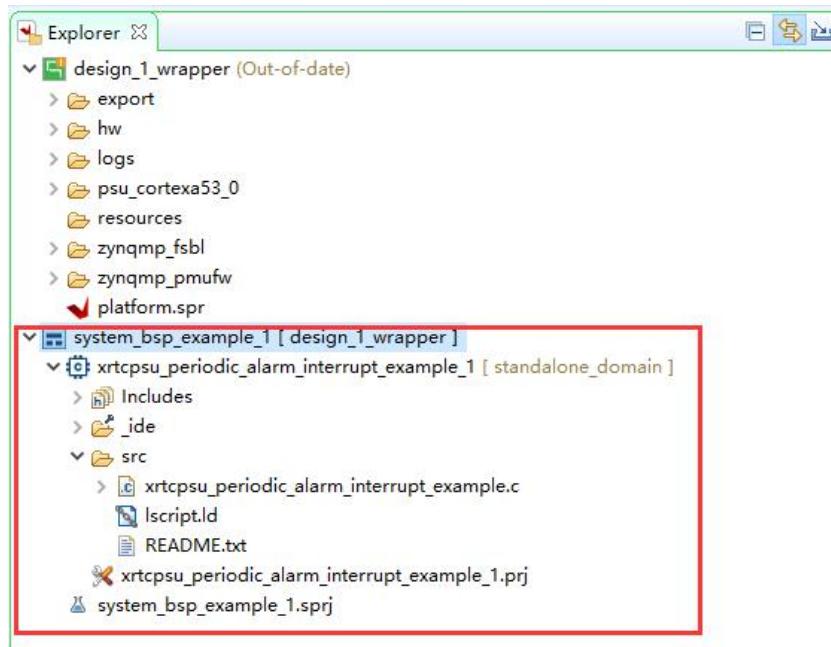
- 7) Find the RTC driver, and click Import Examples

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	Documentation Link	Import Examples
psu_rpu	generic	-	-
psu_rsa	generic	-	-
psu_RTC	rtcpsu	Documentation Link	Import Examples
psu_sd_0	sdps	Documentation Link	Import Examples
psu_sd_1	sdps	Documentation Link	Import Examples
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_SecToDateTIme"** 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\n\r");
xil_printf("Current RTC time is..\n\r");
CurrentTime = XRtcPsu_GetCurrentTime(RtcInstPtr);
KRtcPsu_SecToDateTIme(CurrentTime,&dt0);
xil_printf("YEAR:MM:DD HR:MM:SS \t %04d:%02d:%02d:%02d:%02d\t Day = %d\n\r",
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

```

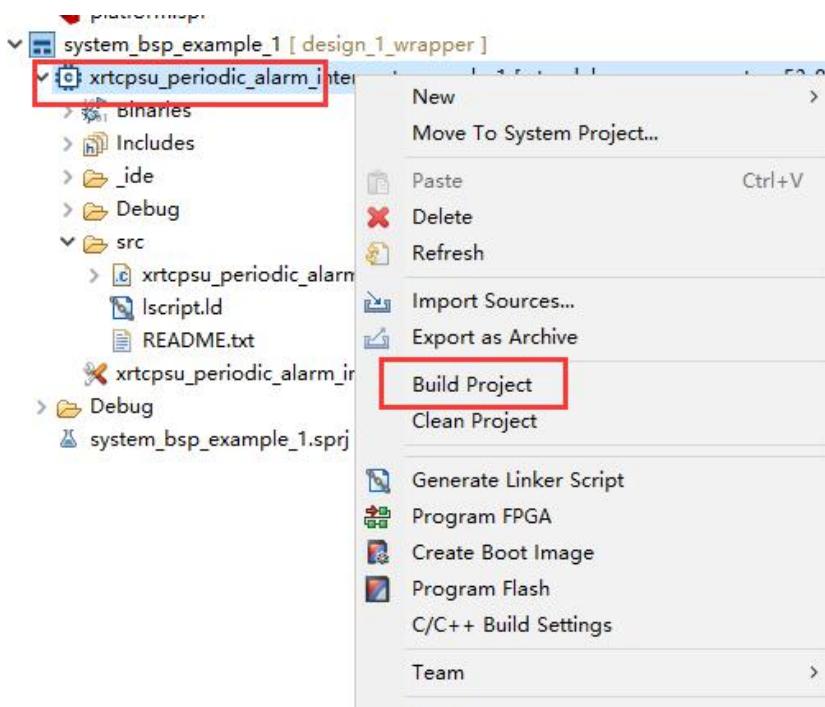
/*
 * Setup the handlers for the RTC that will be called from the
 * interrupt context when alarm and seconds interrupts are raised,
 * specify a pointer to the RTC driver instance as the callback reference
 * so the handlers are able to access the instance data
 */
XRtcPsu_SetHandler(RtcInstPtr, (XRtcPsu_Handler)Handler, 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;           // 中断处理函数
XRtcPsu_SetAlarm(RtcInstPtr, Alarm, 1U);              // 设置中断时间
while( PeriodicAlarms != REPETATIONS);                // 连续中断
{
    /*
     * 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);
```

```
#ifndef TESTAPP_GEN
    /* Enable interrupts */
    Xil_ExceptionEnable();
#endif
    Enable RTC Interrupt
    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

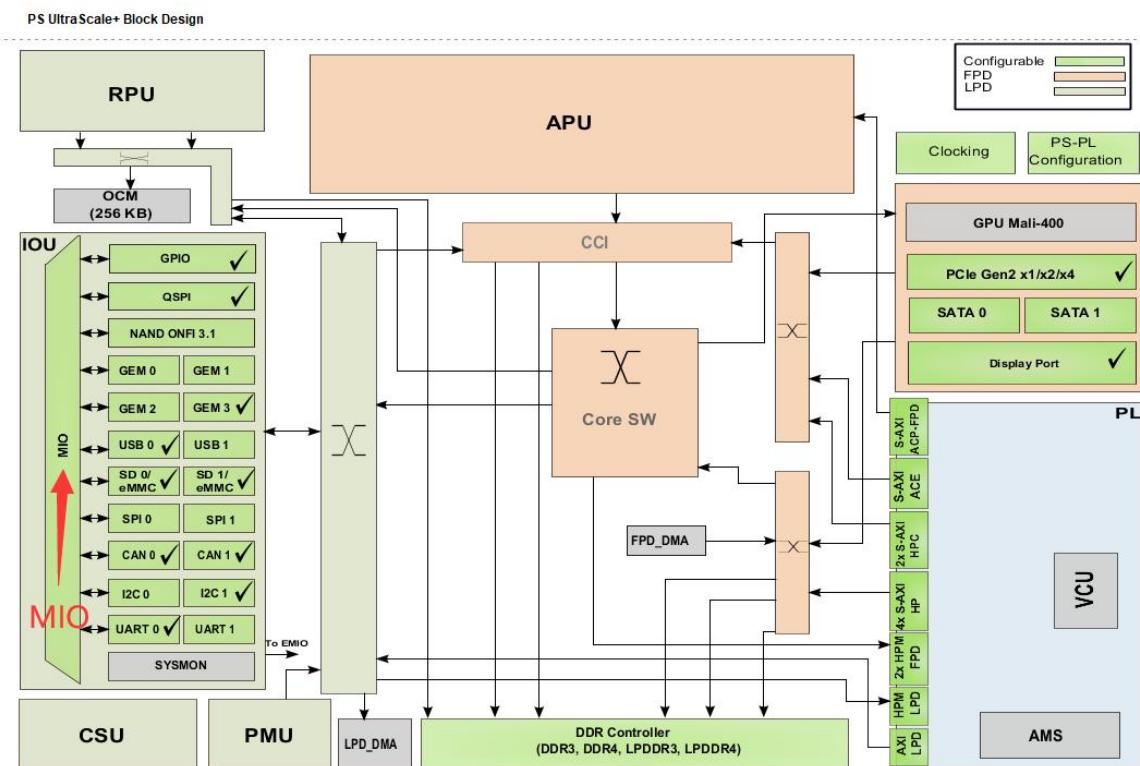
The vivado project directory is "ps_hello/vivado"

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.

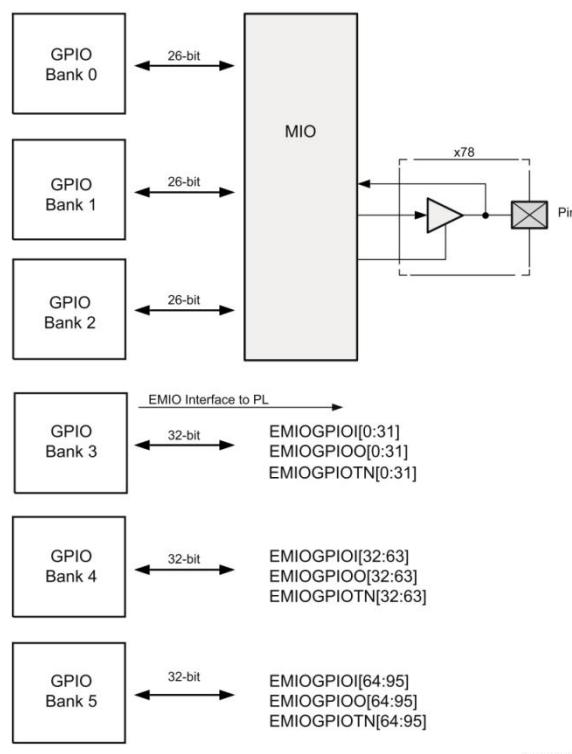
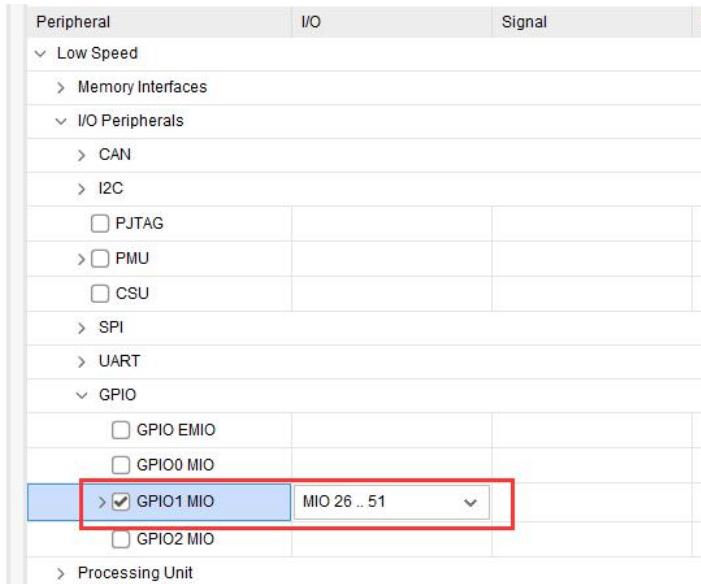


Figure 27-1: GPIO Block Diagram

Part 3.2: Create a “Vivado” Project

This experiment is based on the "ps_hello" project. Through the

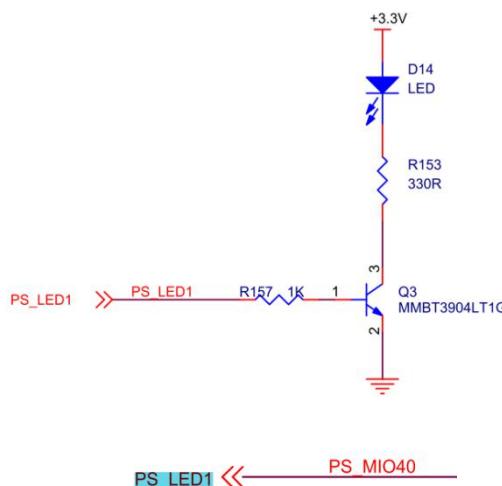
schematic diagram, we know that the LEDs and Keys on the PS side are on GPIO1, which is MIO BANK1, so we need to turn on the GPIO1 MIO, which has been configured in the "ps_hello" project.



Part 3.3: Vitis Program Development

Part 3.3.1: MIO Lights up PS LED

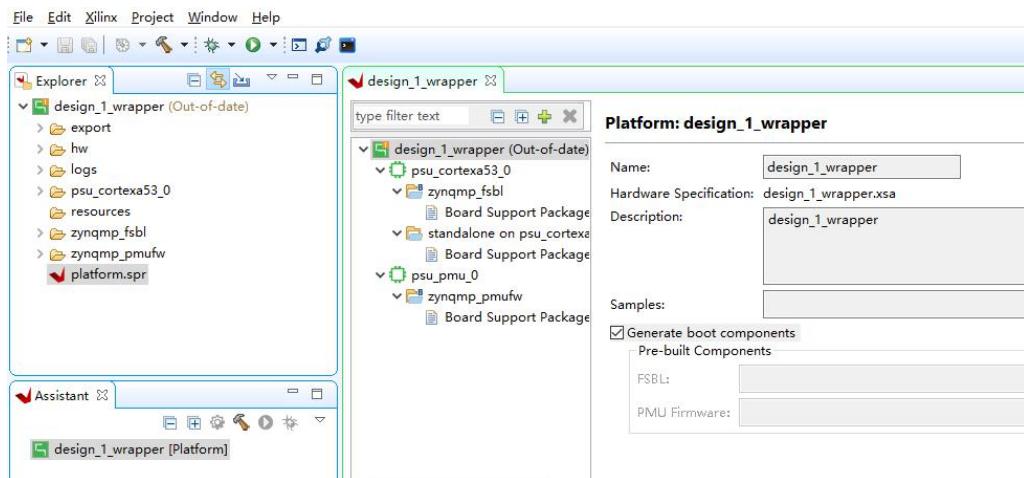
According to the schematic, the LED is connected to MIO40 on the PS side, and can be controlled according to the position of the corresponding FPGA development board MIO.



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.

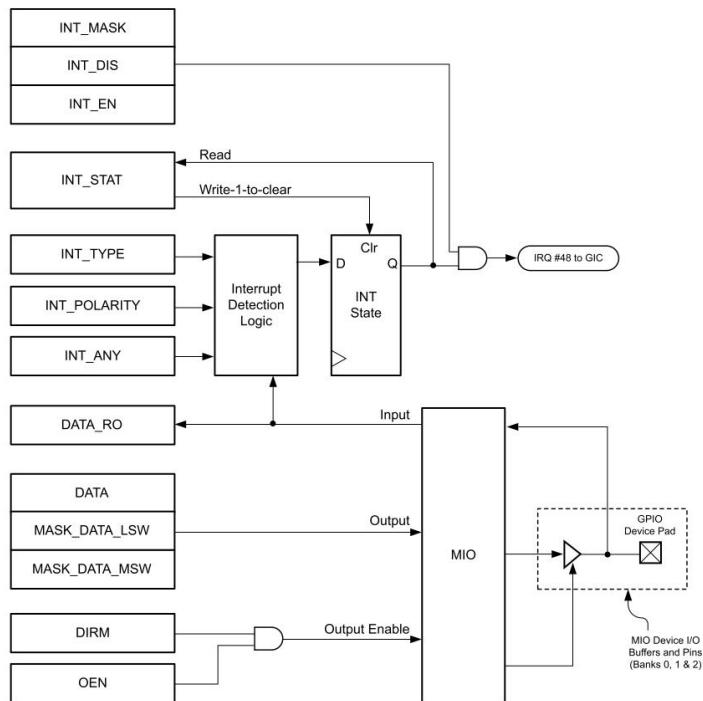
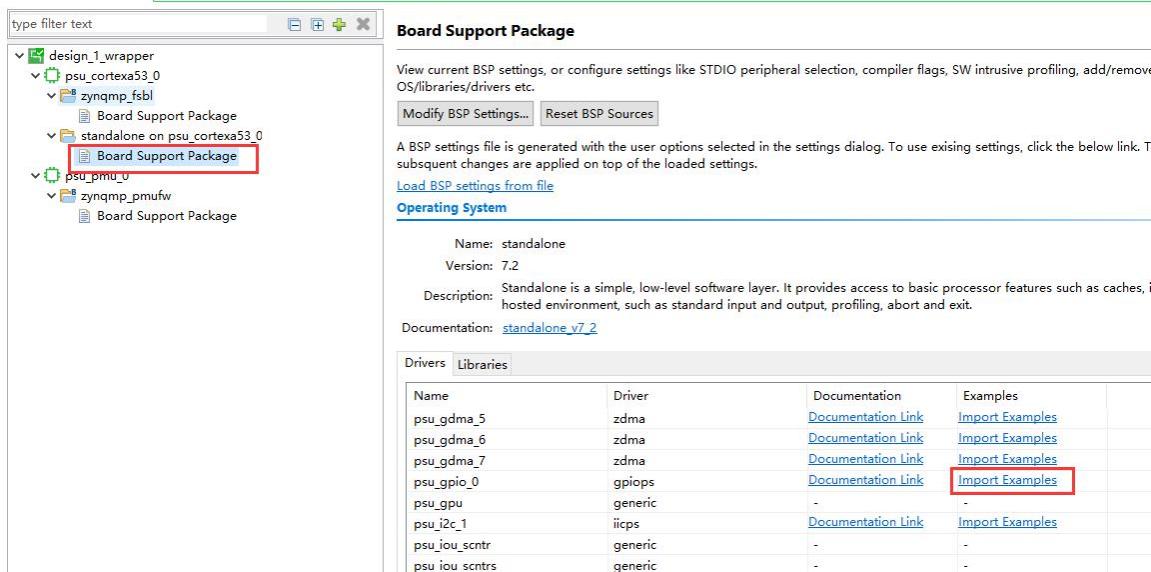


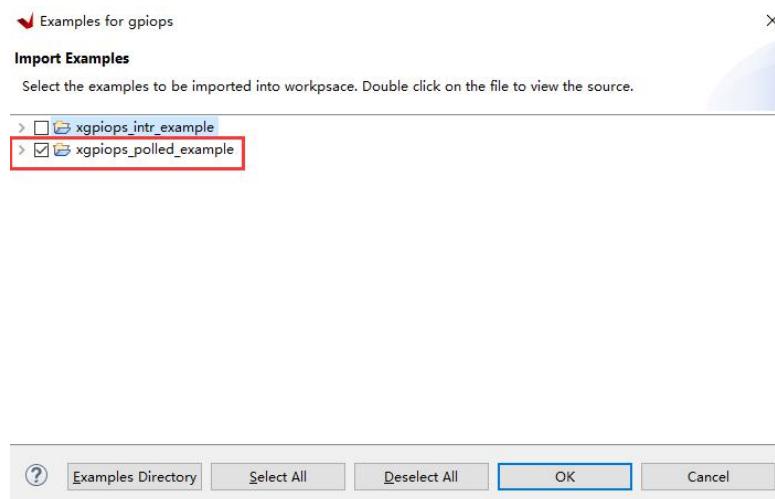
Figure 27-2: GPIO Channel

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

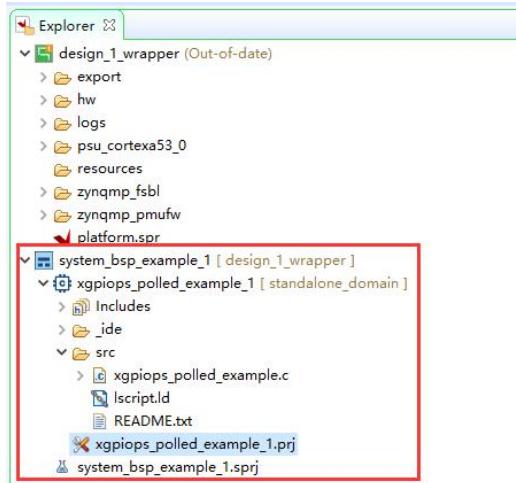
import the example project provided by Xilinx, find “ps7_gpio_0” in BSP, and click “Import Examples”



Select "xgpiops_polled_example" in the pop-up window and click OK



The new APP project will appear



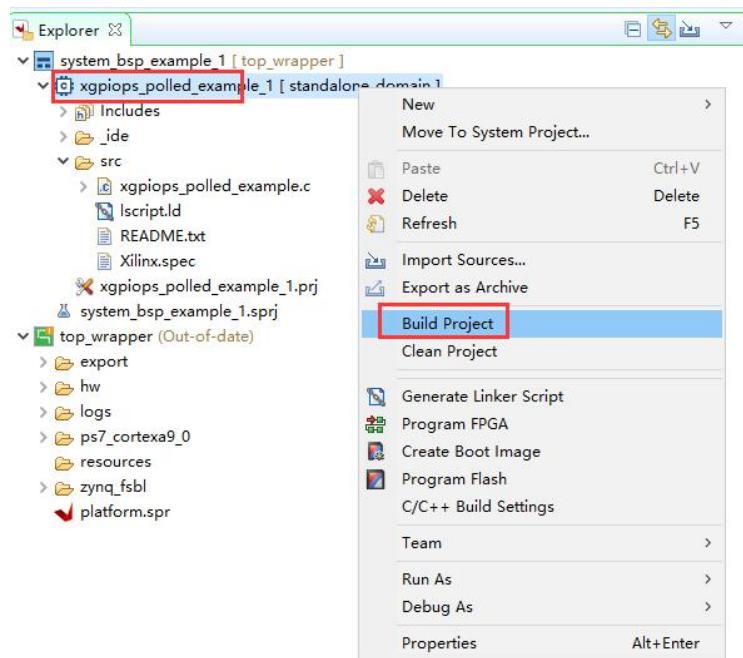
- 4) This example project tests the input and output of the MIO on the PS side. Since the LED on the PS side of the FPGA development board is MIO40, you need to modify the Output_pin to 40 in the file to test the MIO40 LED.

```
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; ← Red arrow points here
93             break;
94
95         case XPLAT_ZYNQ:
96             Input_Pin = 14;
97             Output_Pin = 10;
98             break;
99     }
```

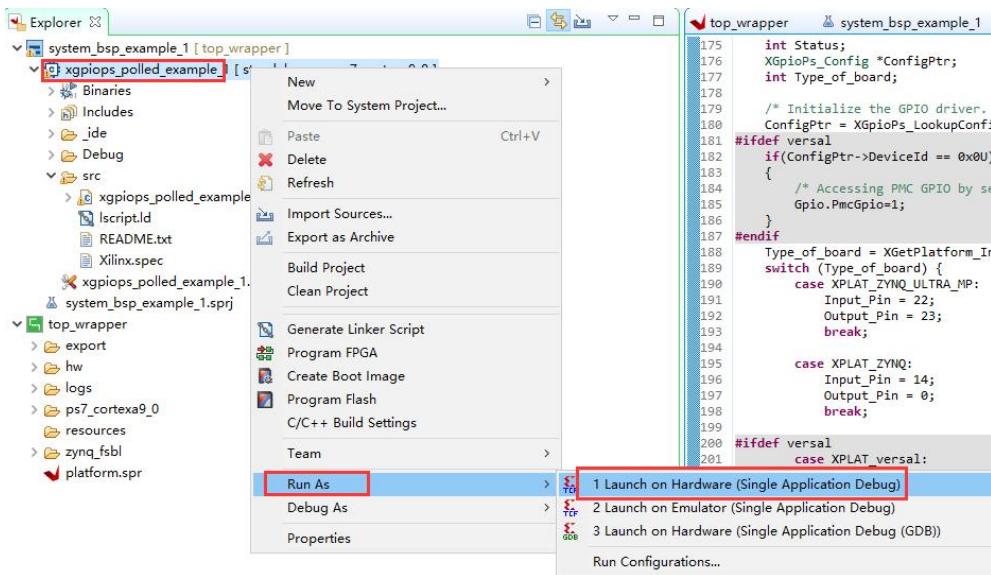
Since only the LED light is tested, that is, the output, comment out the input function and save the document.

```
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 }
```

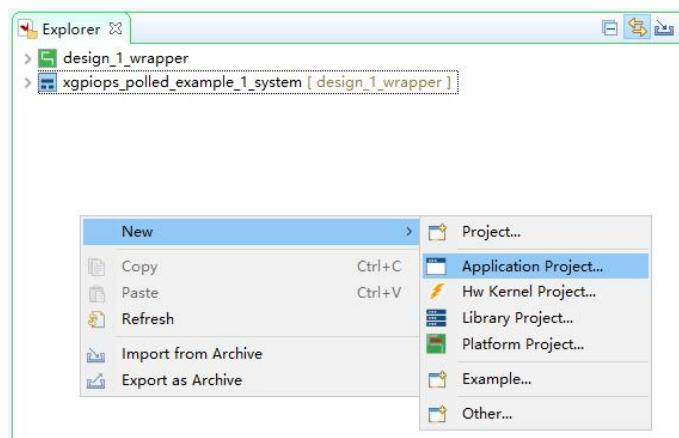
5) Build Project



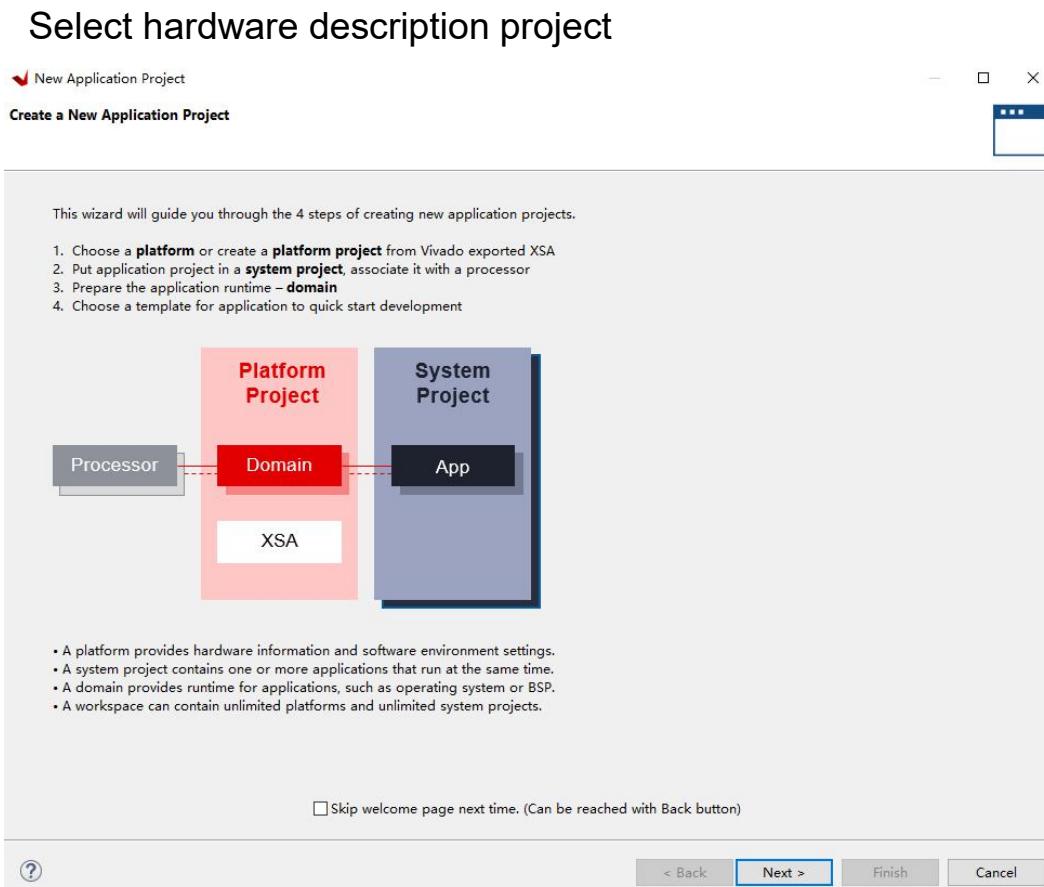
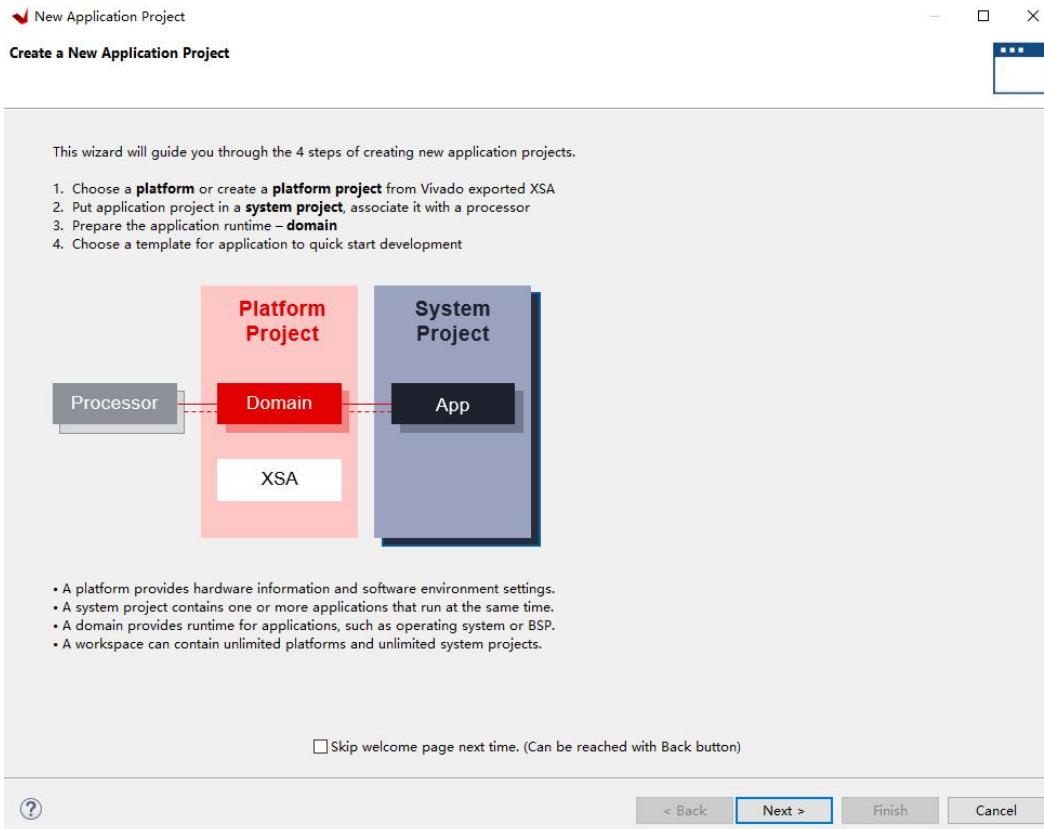
6) Run AsLaunch on Hardware (Single Application Debug), after downloading, you can see PS_LED flashing 16 times quickly



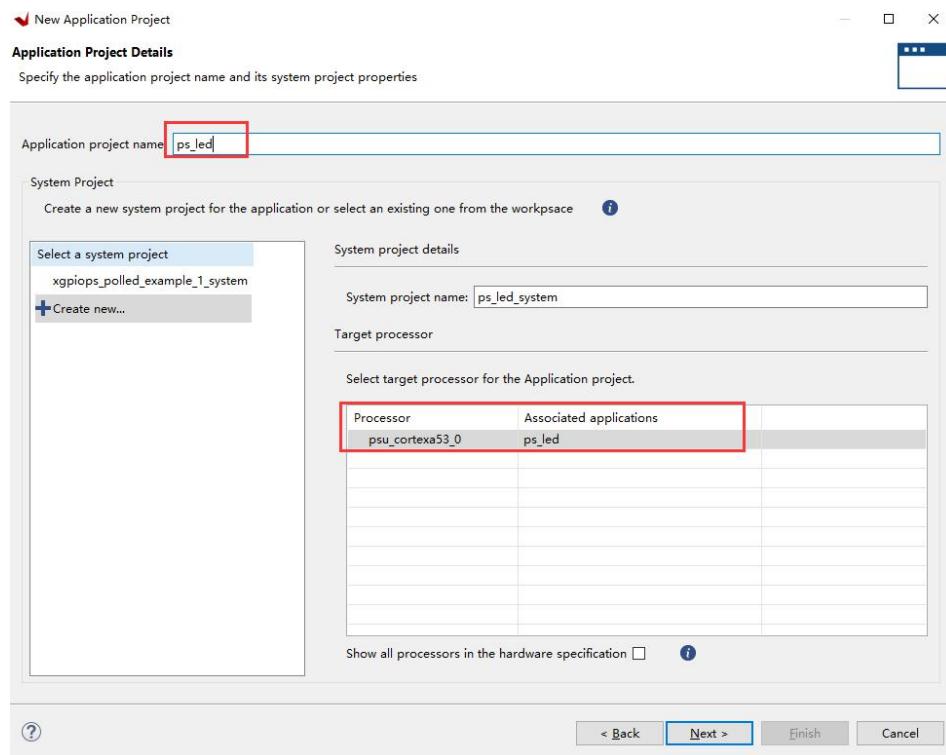
- 7) Although it is more convenient to use official examples, its code looks bloated. You can learn it and simplify it yourself. Modify it in "helloworld.c" of "ps_led_test". In fact, the procedure is very simple. "Initialize the GPIO→set the direction→output enable→control the GPIO output value". We 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



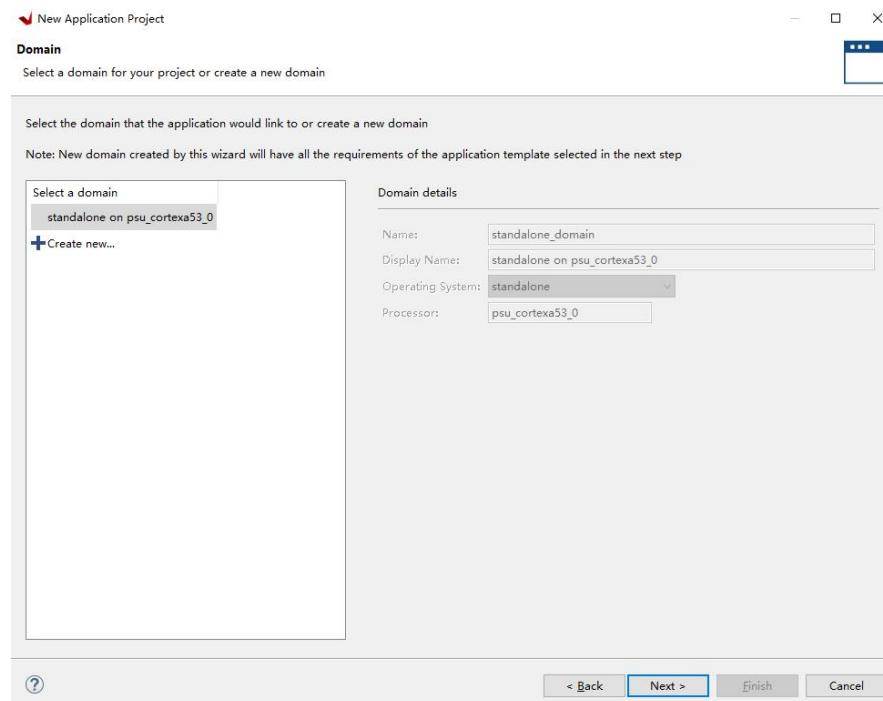
Skip the first page



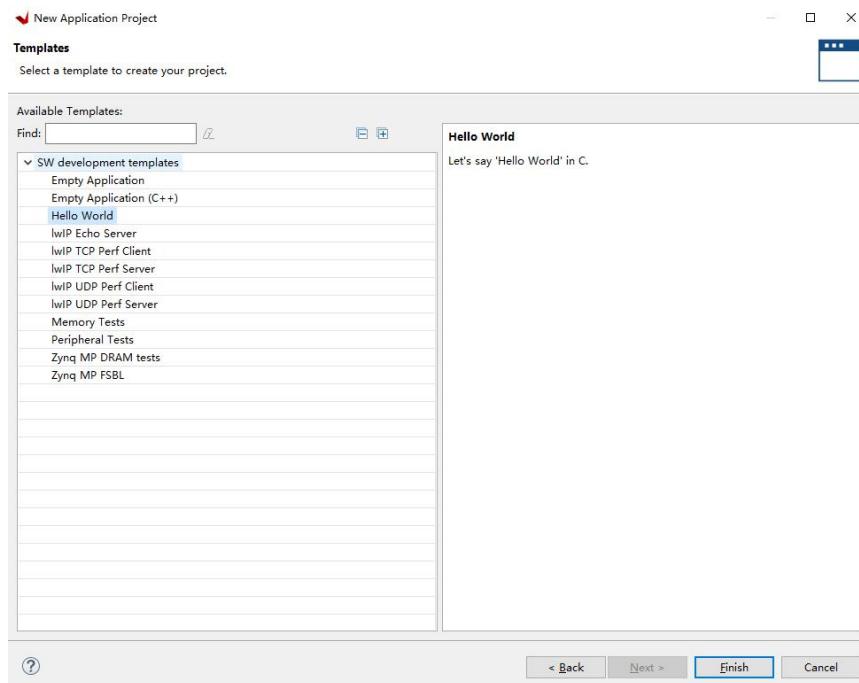
Fill in the project name and select the corresponding CPU



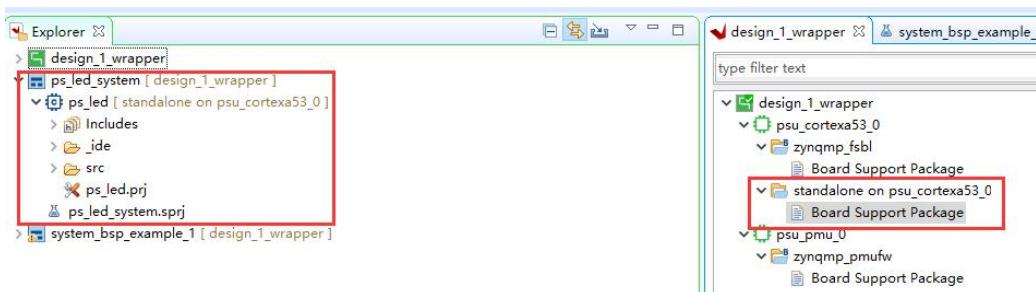
Next step



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

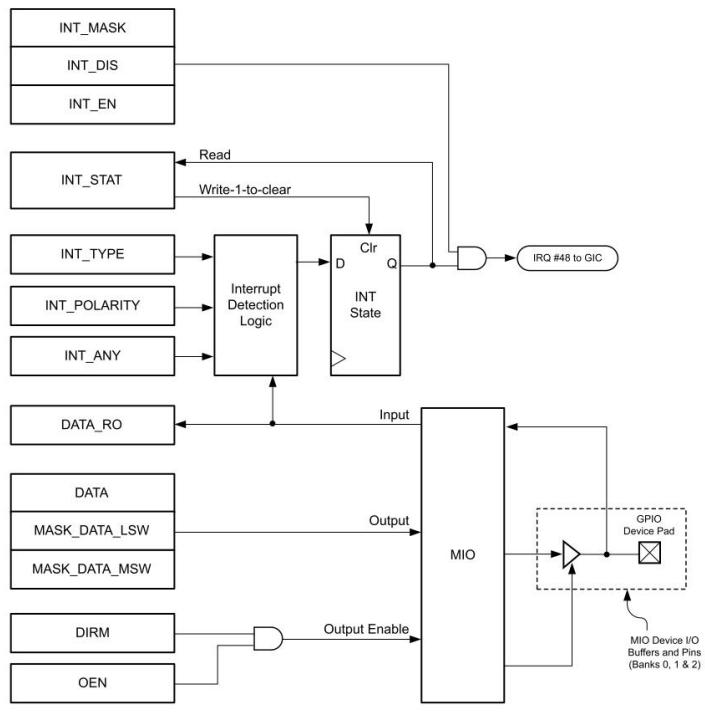
INT_TYPE: interrupt type, set level sensitive or edge sensitive

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

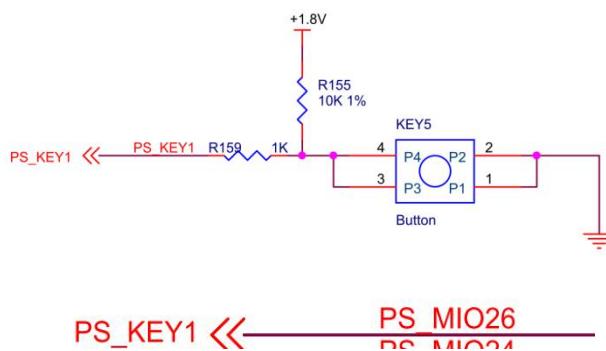
“ug1085 Register Details” for specific register meaning.



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 ;
}
```

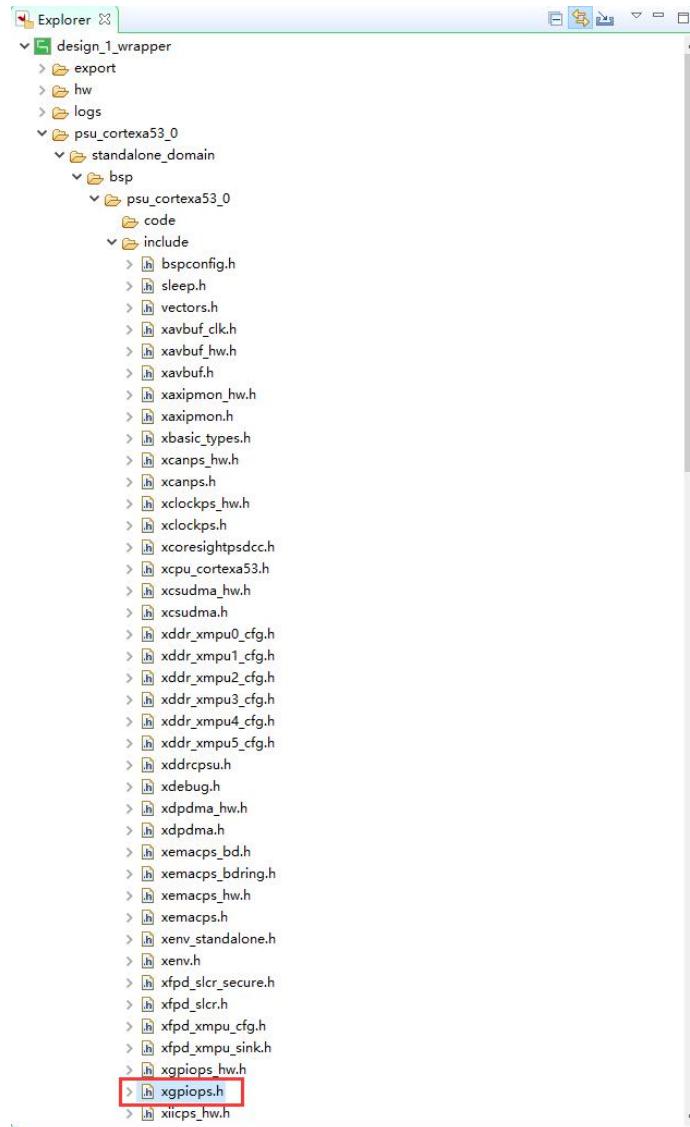
- 8) In the main function, judge the “key_flag” and write data to the “LED”.

```
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_GetOutputEnable(XGpioPs *InstancePtr, u8 Bank);
u32 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.

The screenshot shows the Vitis IDE interface. On the left, the Explorer view displays a tree of files under a project folder, including xil_util.h, xio_secure_slcr.h, xio_slcr.h, xipiops_hw.h, xipiopsu.h, xlpd_slcr_secure.h, xlpd_slcr.h, xlpd_xppu_sink.h, xlpd_xppu.h, xocm_xmpu_cfg.h, xparameters_ps.h, and xparameters.h. The xparameters.h file is selected and highlighted with a red box. On the right, the code editor shows the content of the xparameters.h file:

```

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.

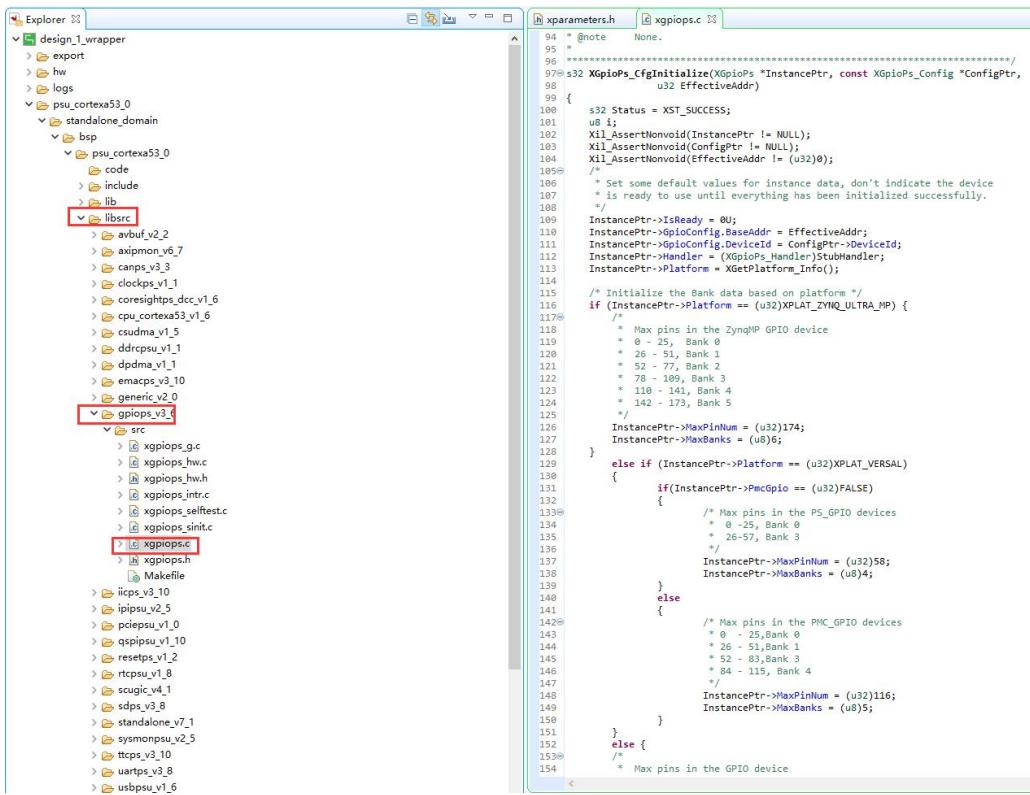
The screenshot shows the Vitis IDE code editor displaying the xparameters.h file. The DEVICE_ID macro definition is highlighted with a blue selection bar:

```

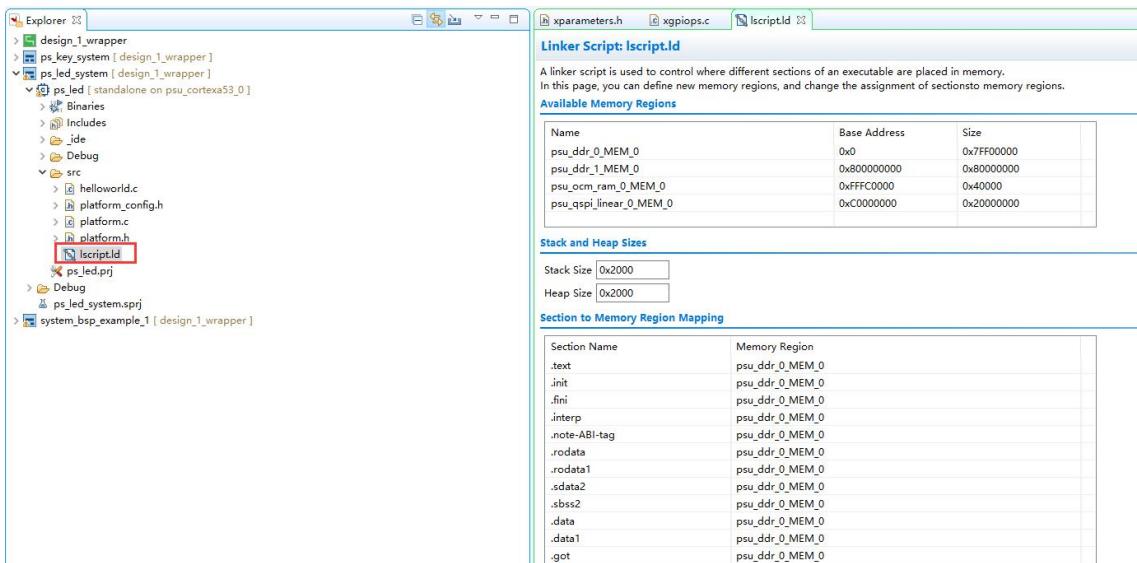
699 ****
700
701 /* Definitions for driver XGPIOPS */
702 #define XPAR_XGPIOPS_NUM_INSTANCES 1
703
704 /* Definitions for peripheral PSU_GPIO_0 */
705 #define XPAR_PSU_GPIO_0_DEVICE_ID 0
706 #define XPAR_PSU_GPIO_0_BASEADDR 0xFF0A0000
707 #define XPAR_PSU_GPIO_0_HIGHADDR 0xFF0AFFFF
708
709 ****
710
711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

```

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

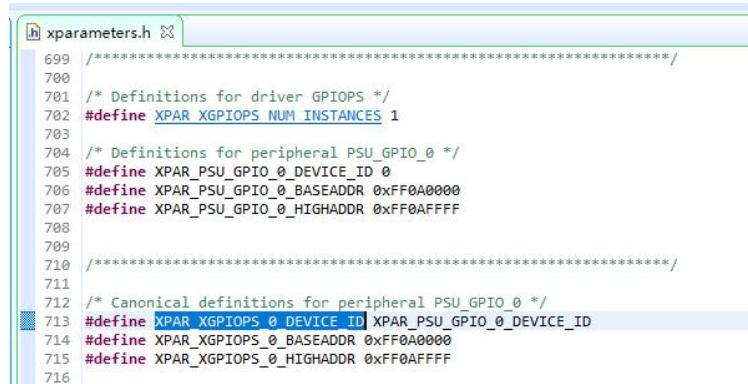


- 4) The “lscript.ld” under the “src” folder defines the available “memory” space, stack and heap space sizes, etc., which can be modified as needed.



- 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 XPAR_PSU_GPIO_0_BASEADDR
#define XPAR_XGPIOPS_0_HIGHADDR XPAR_PSU_GPIO_0_HIGHADDR
```

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.

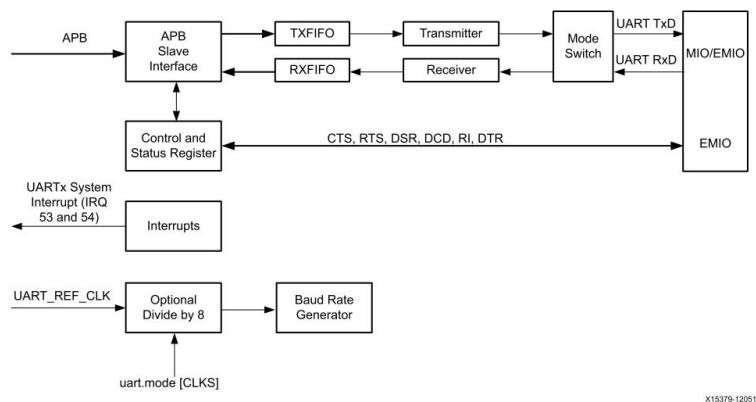


Figure 21-1: UART Controller

The following figure shows the four modes of the UART.

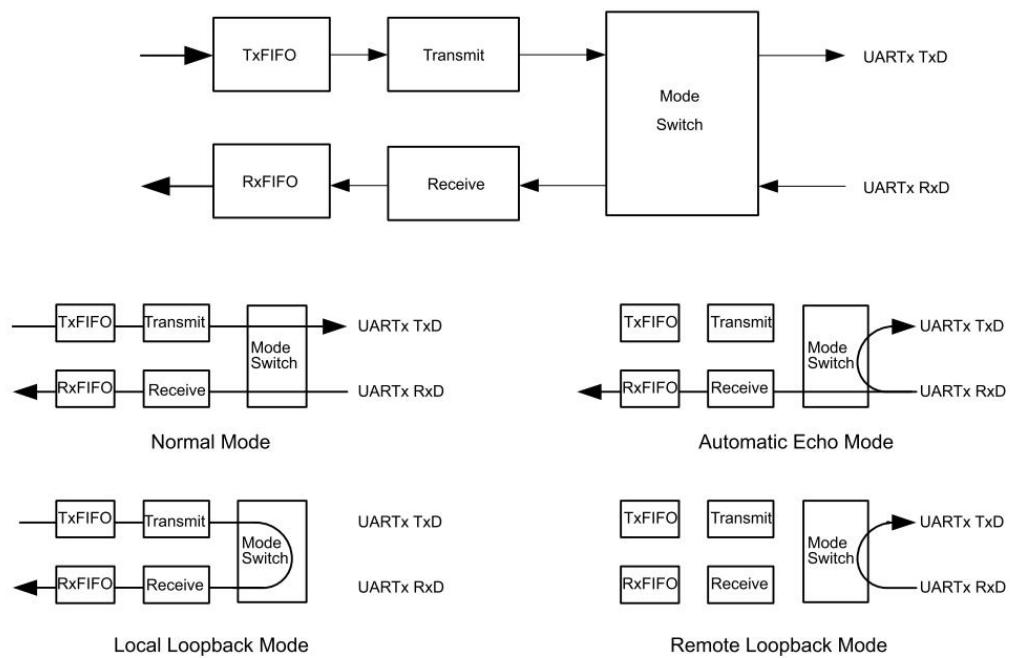


Figure 21-6: UART Mode Switch for TxD and RxD

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);
    }
}

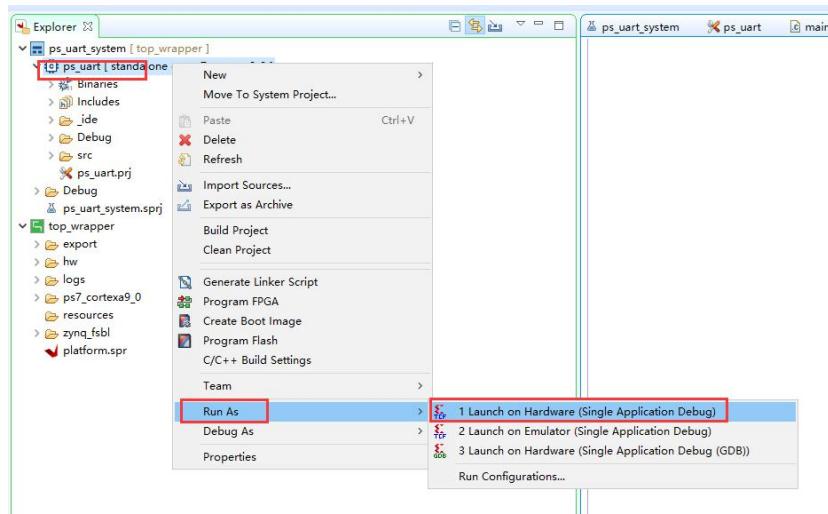
```

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.

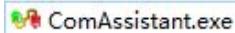
			Documentation Link	Import Examples
psu_sd_1	sdps	-	-	-
psu_serdes	generic	-	-	-
psu_siou	generic	-	-	-
psu_smmu_gpv	generic	-	-	-
psu_smmu_reg	generic	-	-	-
psu_ttc_0	ttcps	Documentation Link	Import Examples	
psu_ttc_1	ttcps	Documentation Link	Import Examples	
psu_ttc_2	ttcps	Documentation Link	Import Examples	
psu_ttc_3	ttcps	Documentation Link	Import Examples	
psu_uart_0	uartps	Documentation Link	Import Examples	
psu_usb_0	generic	-	-	
psu_usb_xhci_0	usbpsu	Documentation Link	Import Examples	

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"

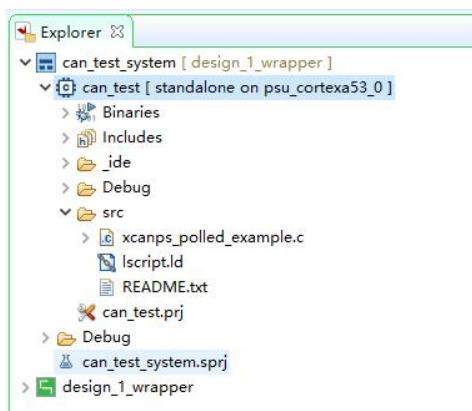
This chapter introduces the use of two CAN interfaces on the board for loopback testing.

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);

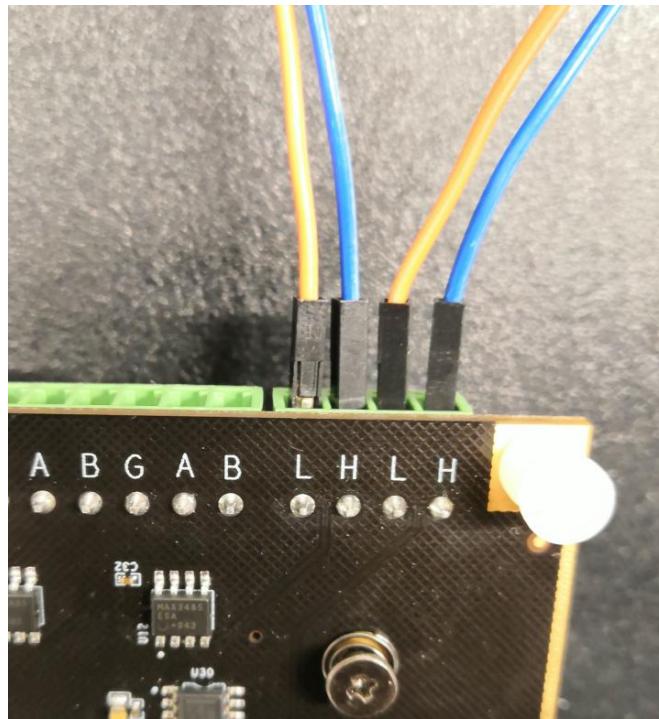
    return Status;
}

```

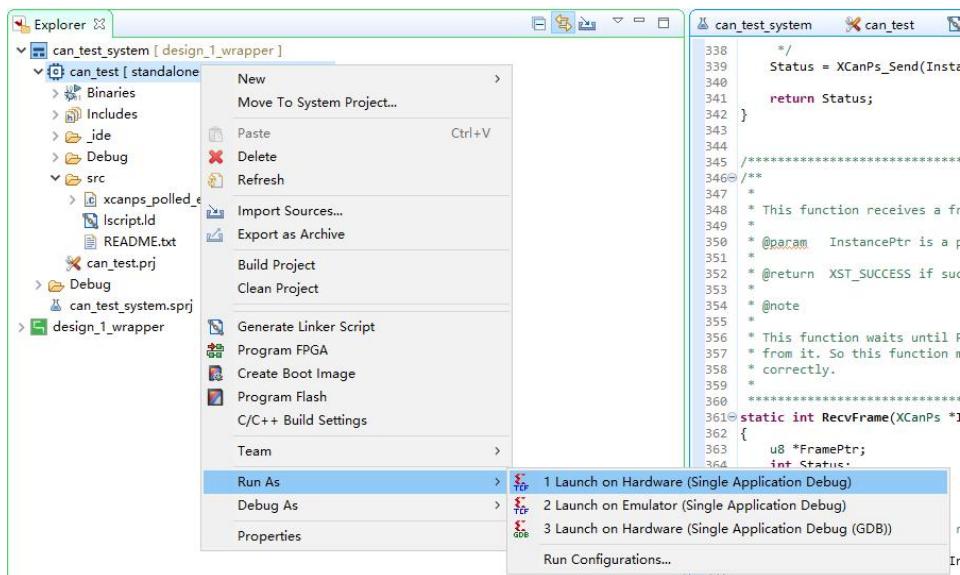
Part 5.2: Download and Test

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

wire



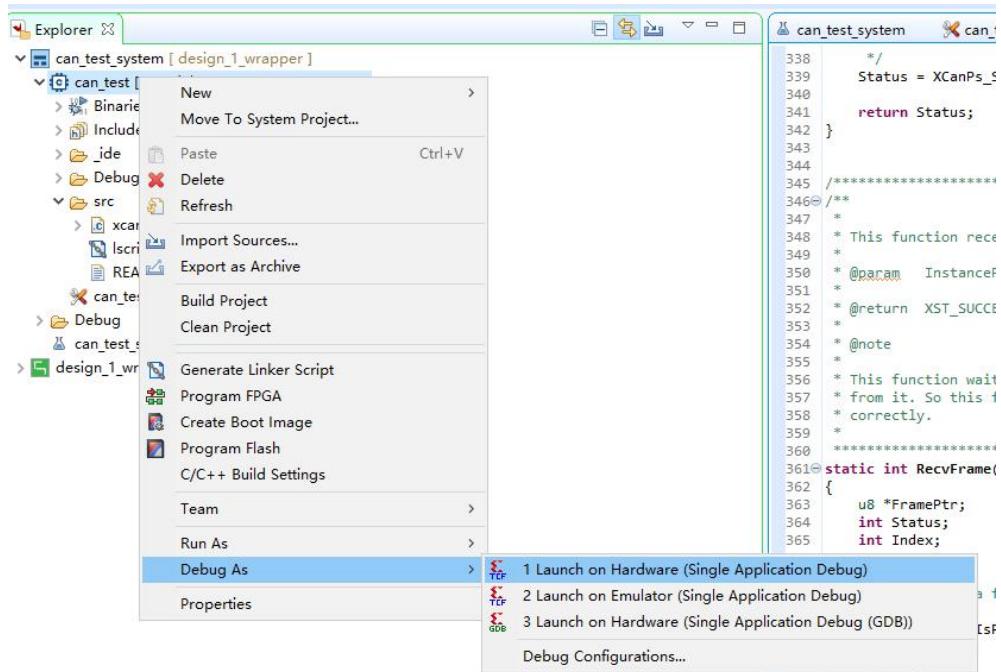
2) Download program



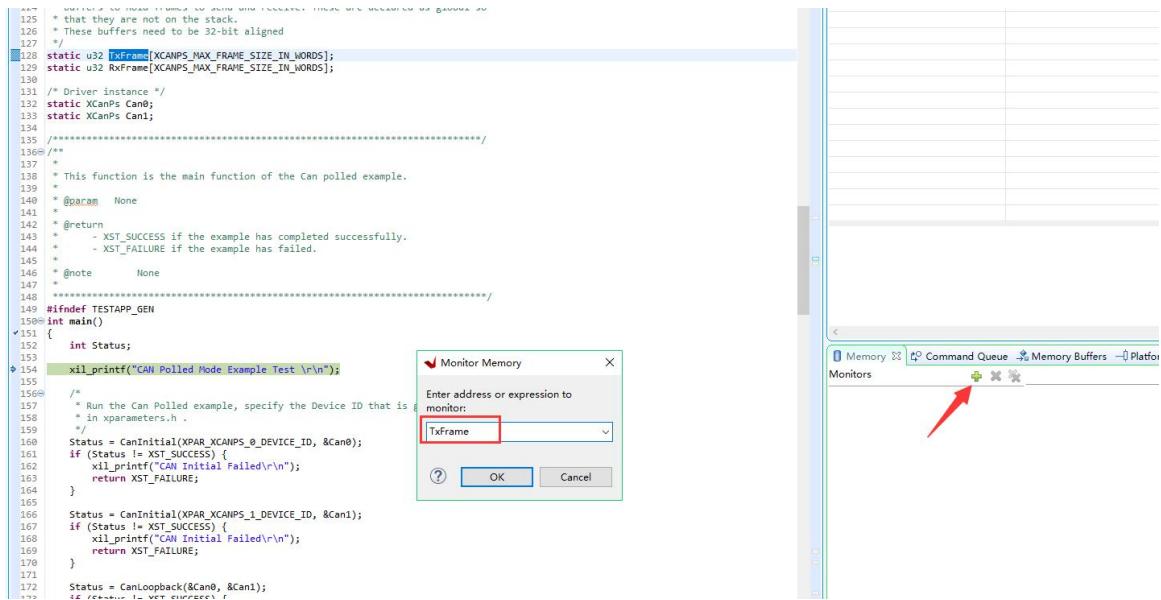
3) You can see the print information on the serial port

```
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

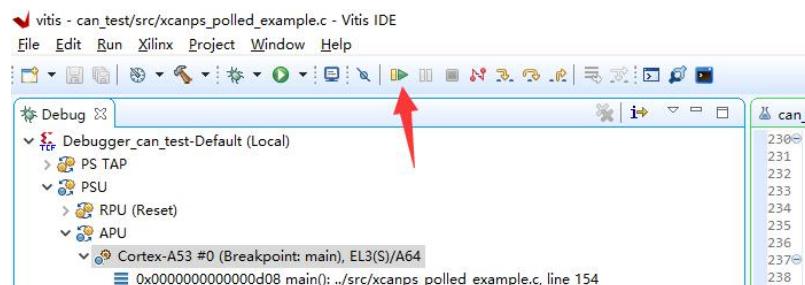
Monitors	RxFrame : 0xC0F8 <Hex Integer>	New Renderings...
TxFrame	000000000000C0F0	00000000
RxFrame	000000000000C100	00000000
	000000000000C110	00000000
	000000000000C120	00000000
	000000000000C130	00000000
	000000000000C140	00000000
	000000000000C150	00000000
	000000000000C160	00000000
	000000000000C170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

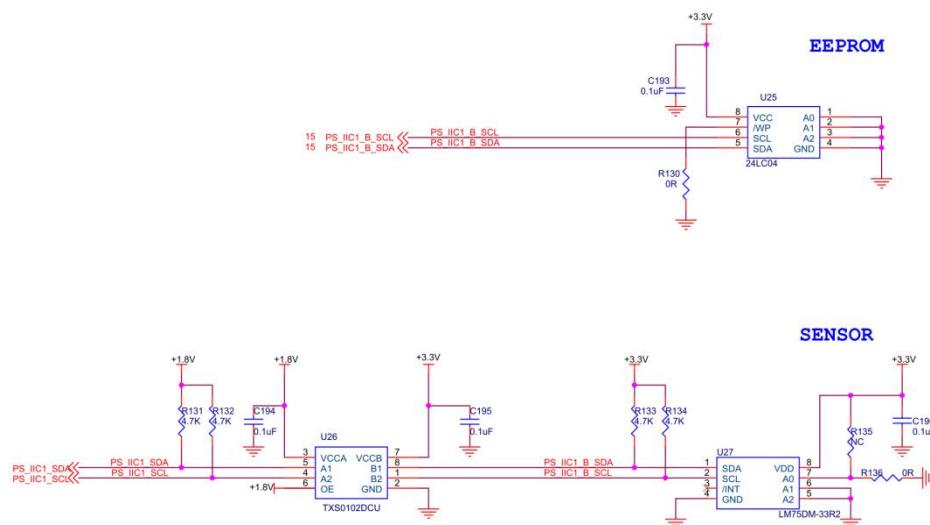
The vivado project directory is "ps_hello/vivado"

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 ($^{\circ}\text{C}$)	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.

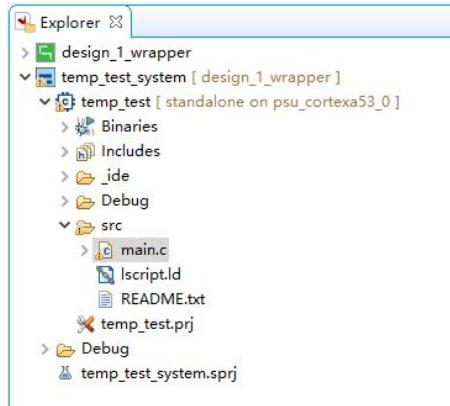
And its data has two bytes, the first byte is the integer bit, the second byte is the decimal place, the highest bit is 1, that is, 0.5 degrees Celsius, if it is 0, the decimal place is 0

Table 4. Temperature Data Output Format

TEMPERATURE ($^{\circ}\text{C}$)	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 ;

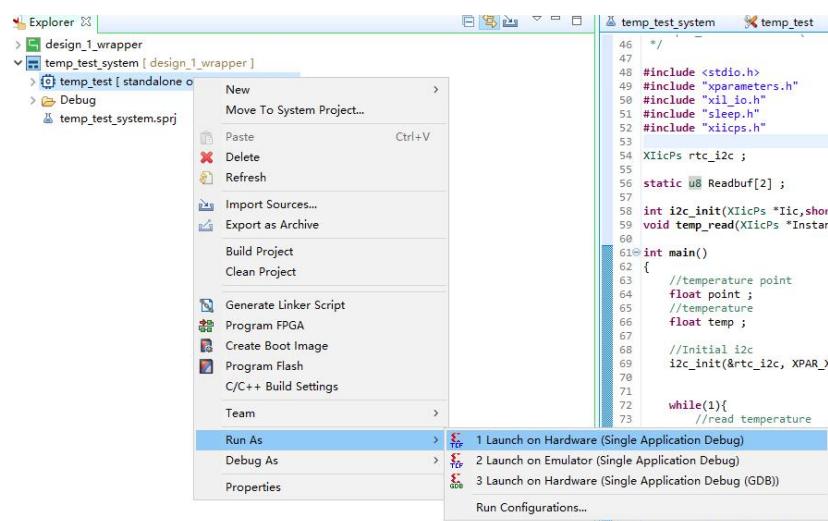
        //if bit7 equals to 1, then it is negative value
        if (Readbuf[0] & 0x80){
            temp = (float)(Readbuf[0]-256) + point ;
        }
        else{
            temp = (float)(Readbuf[0]) + point ;
        }

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

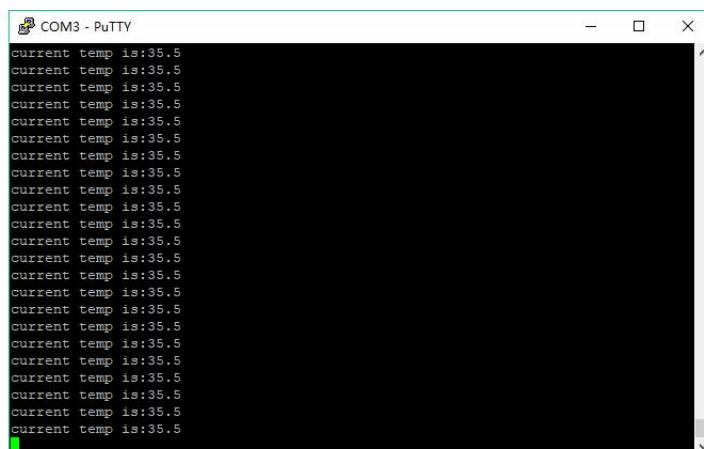
        sleep(1) ;
    }
    return 0;
}

```

5) Download the program

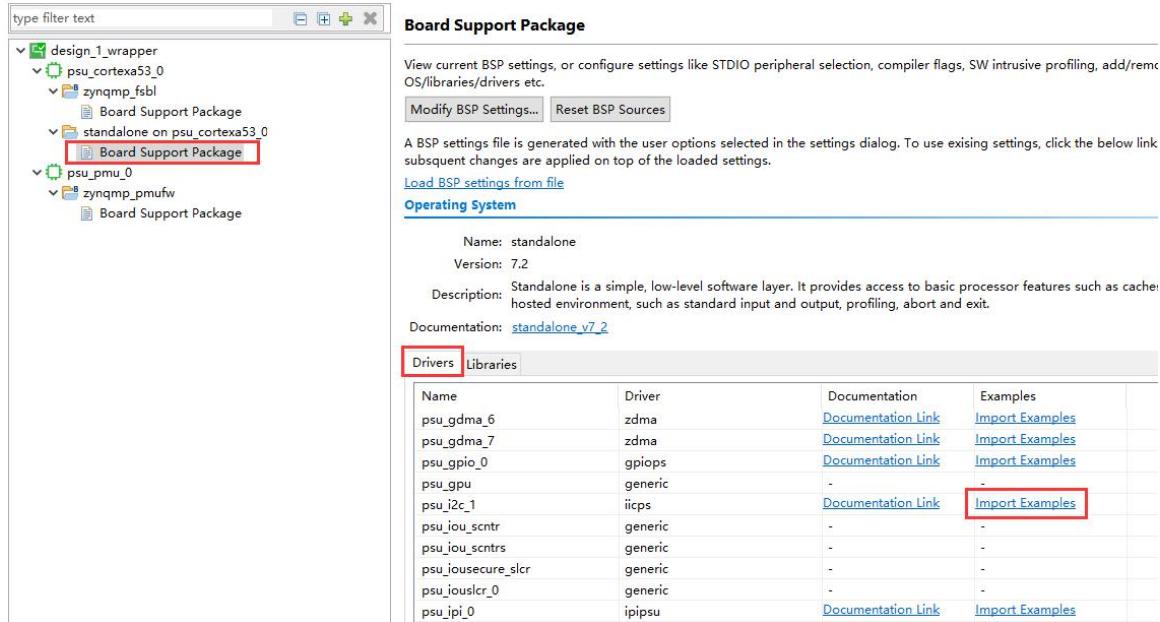


6) The serial port print information is as follows

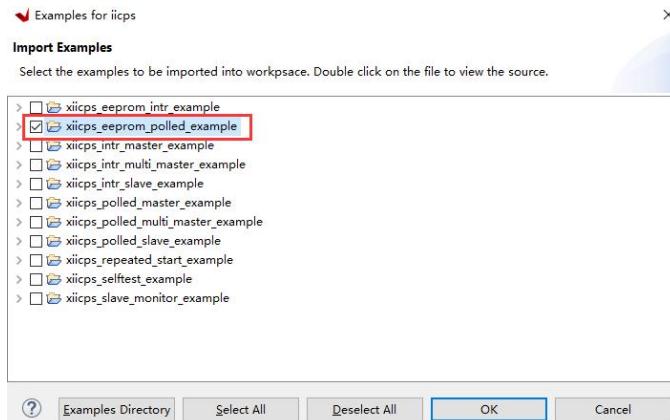


Part 6.1.2: EEPROM Read and Write

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

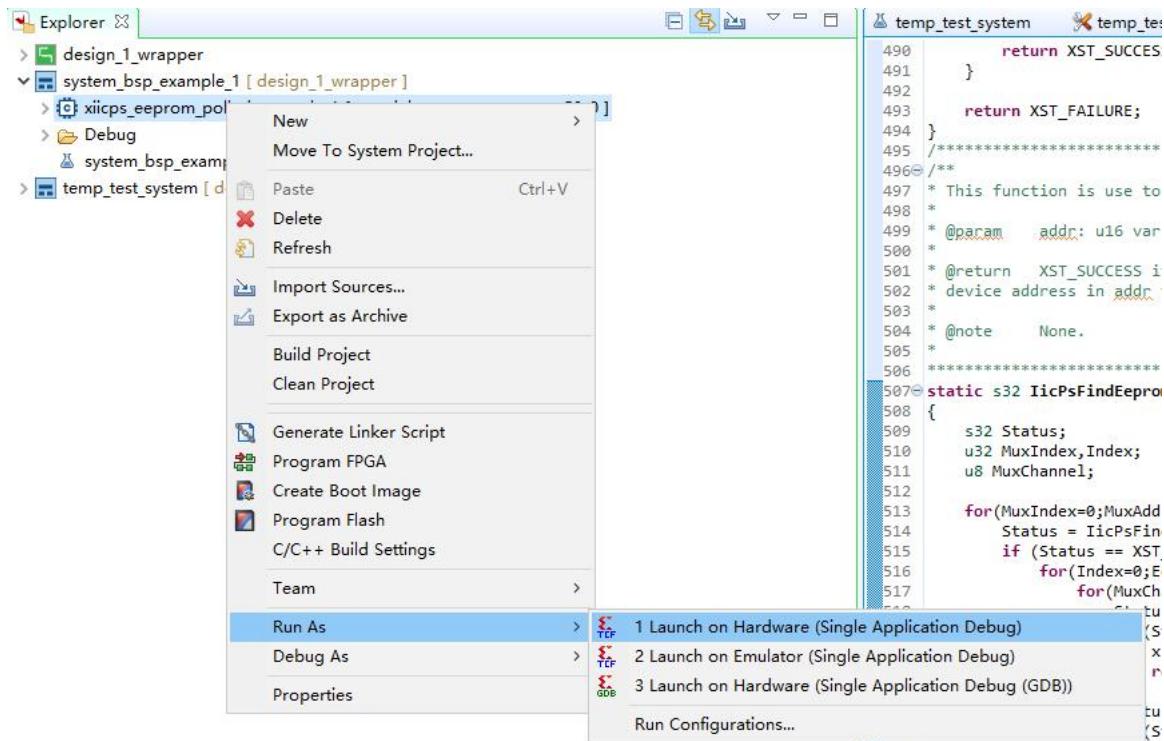
- 4) Since the address of EEPROM is 1 byte, modify it as follows in the program

```
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"

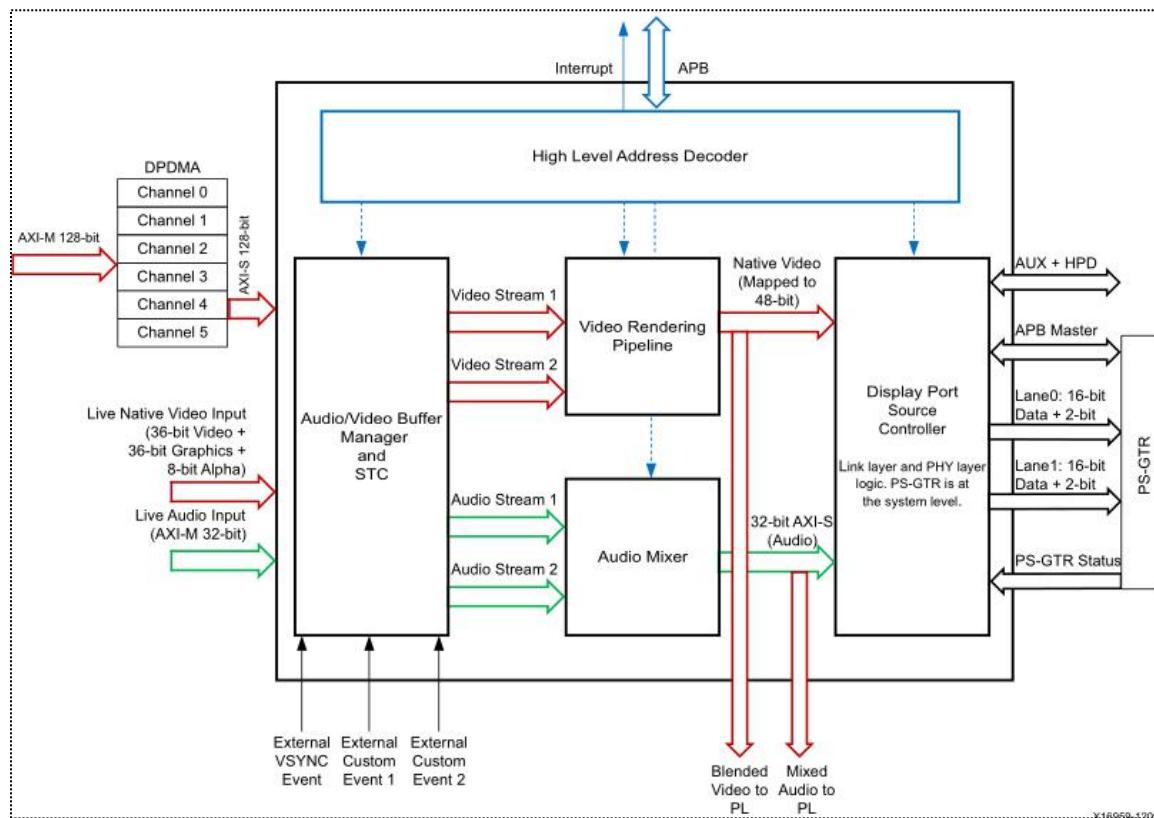
Software Engineer Job Content

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.

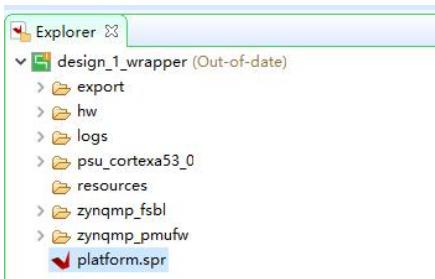
The controller data interface is as follows:



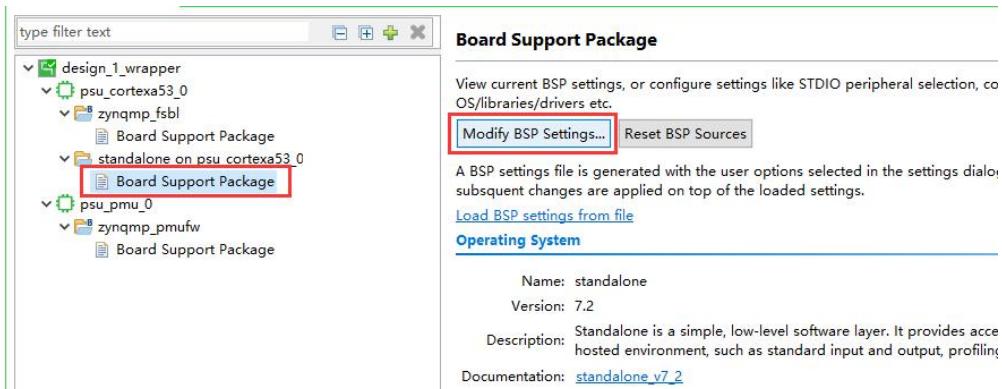
In the figure, AXI-M is used to read the video and audio data in the memory, here called non-real-time audio and video, DPDMA has six channels, of which 3 channels are used for video, 1 channel is used for graphics, and 2 channels are used for audio.

Part 7.2: Example Project Introduction

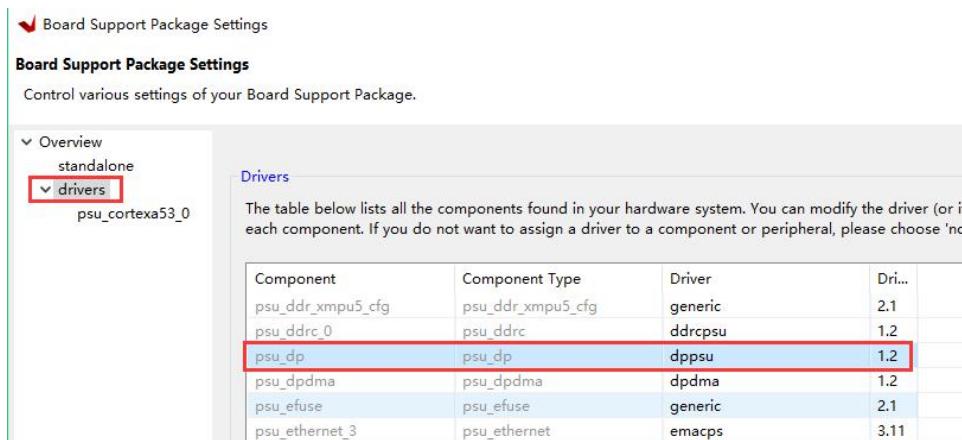
- 1) The process of creating a new platform will not be introduced. It has been introduced in "PS Side RTC Interrupt Experiment".



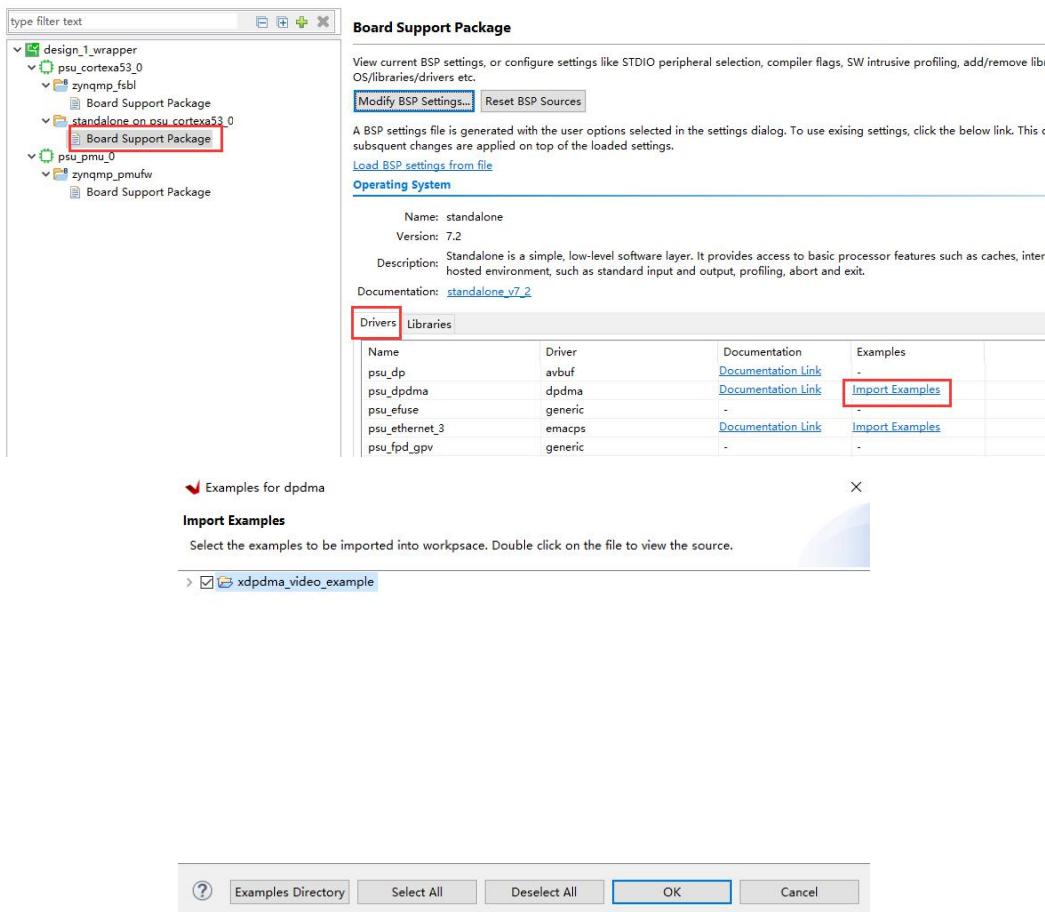
- 2) Configure BSP



And change the psu_dp driver to dppsu, then click OK



- 3) Import example project

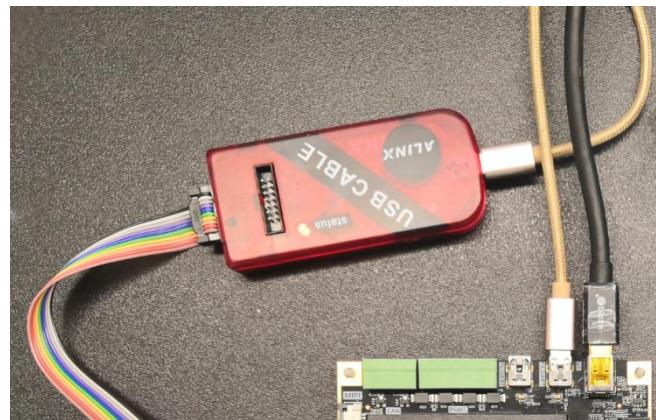


- 4) The example defaults to 1080P, RGBA display, you can change the Alpha value of RGBA to FF to make the display effect better, save it, and compile the project.

```
/*
*****
*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 = 0xF0
         */
        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"

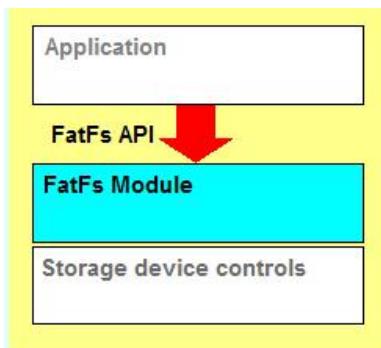
The vitis project directory is "ps_sd/vitis"

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

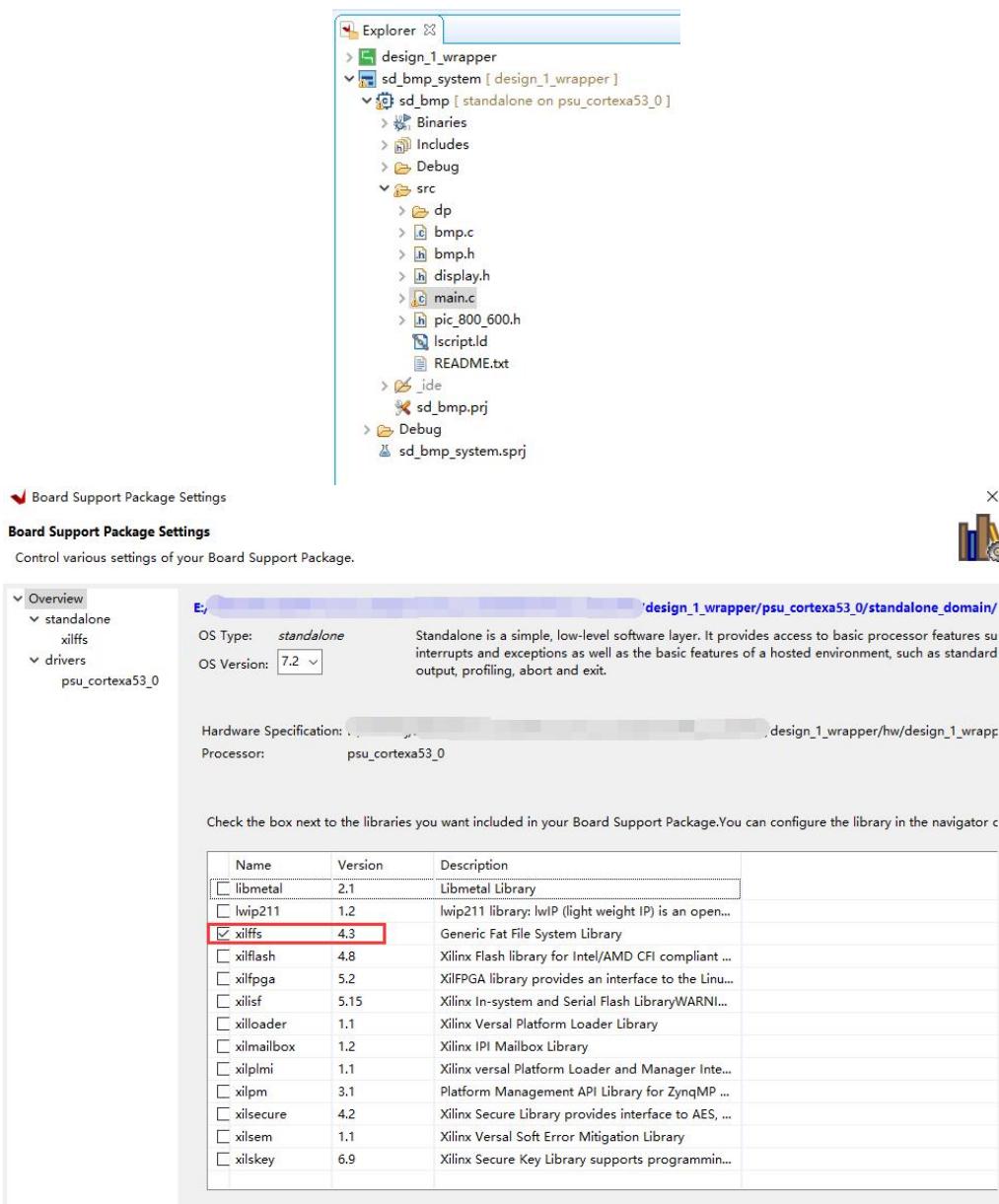
`F_write` - write file

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

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.

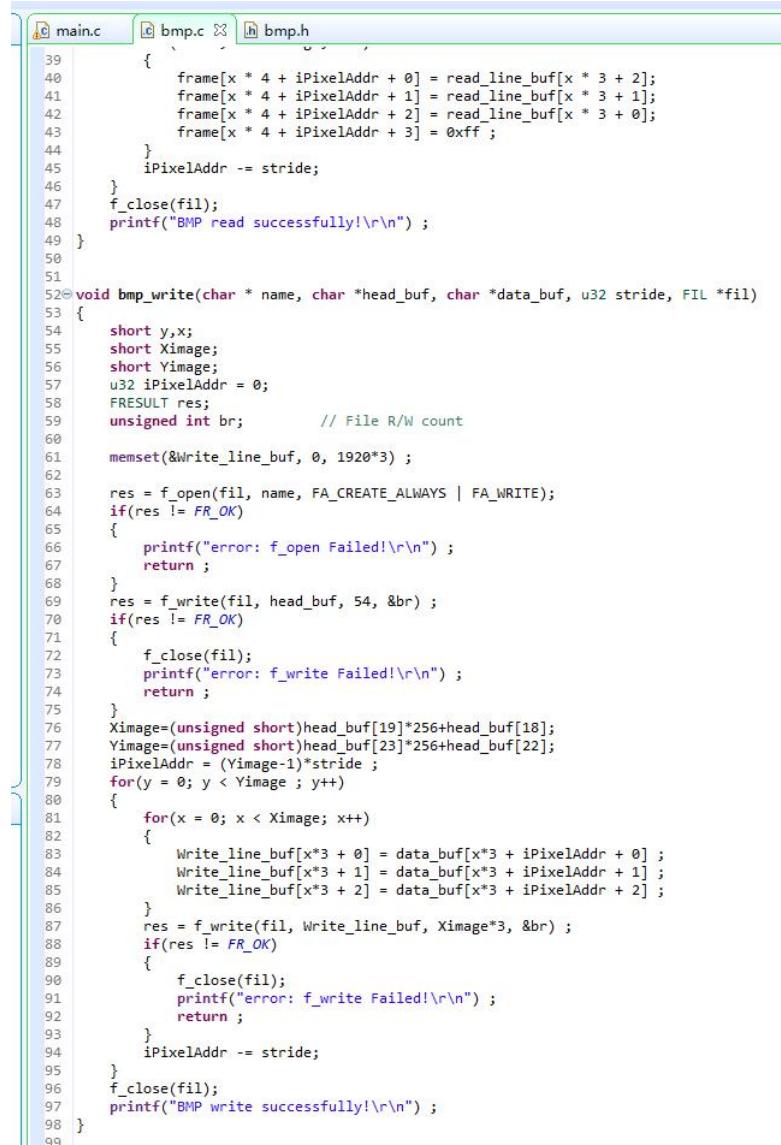
```

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.

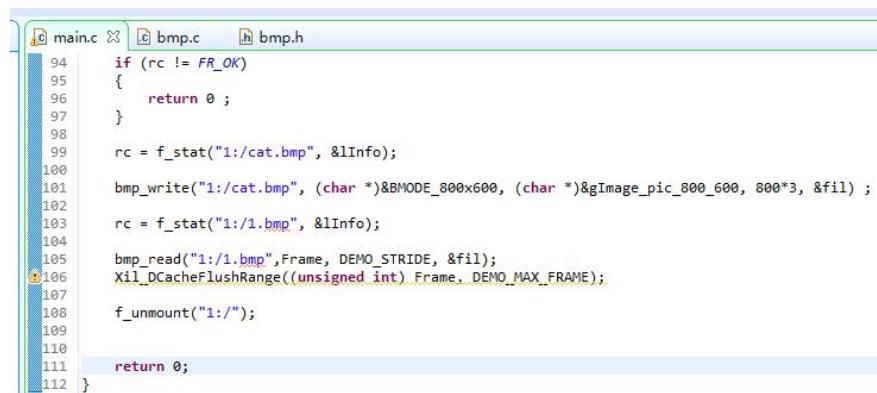


```

main.c
1 // main.c
2 // This file contains the main program logic for reading a BMP image
3 // from an SD card and writing it to memory.
4
5 #include "main.h"
6 #include "vitis_vdma.h"
7 #include "fsl_frc.h"
8 #include "fsl_sd.h"
9 #include "fsl_sdio.h"
10 #include "fsl_sdmmc.h"
11 #include "fsl_sdcc.h"
12 #include "fsl_sdcc.h"
13 #include "fsl_sdcc.h"
14 #include "fsl_sdcc.h"
15 #include "fsl_sdcc.h"
16 #include "fsl_sdcc.h"
17 #include "fsl_sdcc.h"
18 #include "fsl_sdcc.h"
19 #include "fsl_sdcc.h"
20 #include "fsl_sdcc.h"
21 #include "fsl_sdcc.h"
22 #include "fsl_sdcc.h"
23 #include "fsl_sdcc.h"
24 #include "fsl_sdcc.h"
25 #include "fsl_sdcc.h"
26 #include "fsl_sdcc.h"
27 #include "fsl_sdcc.h"
28 #include "fsl_sdcc.h"
29 #include "fsl_sdcc.h"
30 #include "fsl_sdcc.h"
31 #include "fsl_sdcc.h"
32 #include "fsl_sdcc.h"
33 #include "fsl_sdcc.h"
34 #include "fsl_sdcc.h"
35 #include "fsl_sdcc.h"
36 #include "fsl_sdcc.h"
37 #include "fsl_sdcc.h"
38 #include "fsl_sdcc.h"
39 #include "fsl_sdcc.h"
40 #include "fsl_sdcc.h"
41 #include "fsl_sdcc.h"
42 #include "fsl_sdcc.h"
43 #include "fsl_sdcc.h"
44 #include "fsl_sdcc.h"
45 #include "fsl_sdcc.h"
46 #include "fsl_sdcc.h"
47 #include "fsl_sdcc.h"
48 #include "fsl_sdcc.h"
49 #include "fsl_sdcc.h"
50 #include "fsl_sdcc.h"
51 #include "fsl_sdcc.h"
52 void bmp_read(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 read 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);
Xil_DCACHEflushRange((unsigned int)Frame, DEMO_MAX_FRAME);
107
108    f_unmount("1:/");
109
110
111    return 0;
112

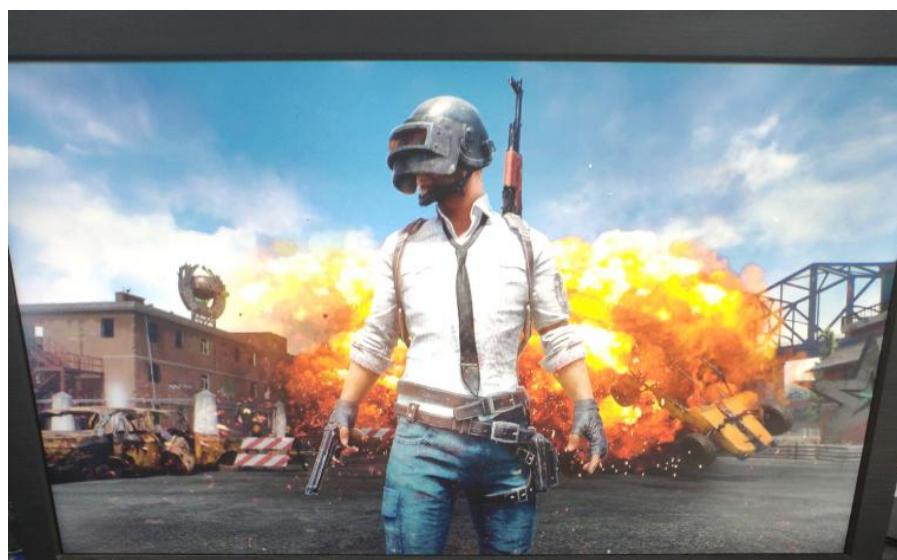
```

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_mcdma.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            0x3FO
#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;
}
```

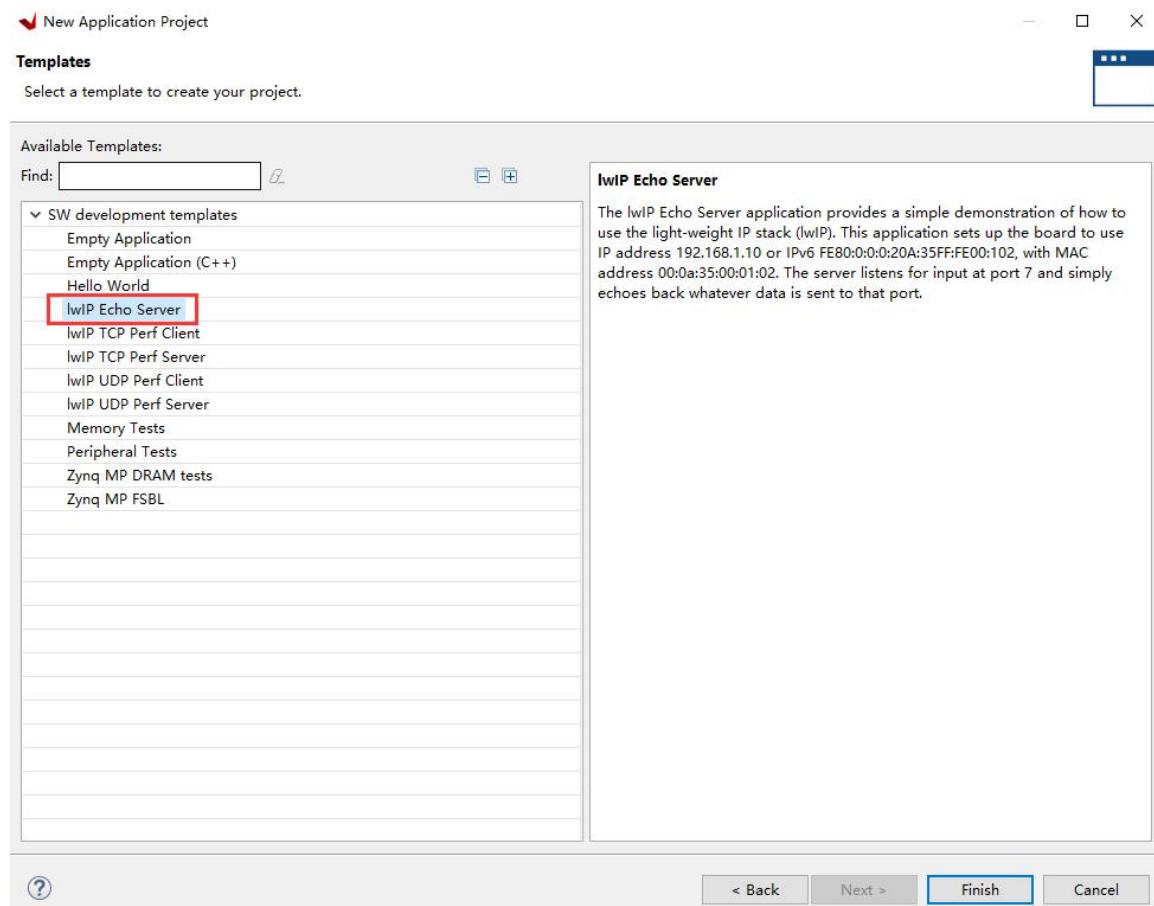
- 8) Modify the function "get_IEEE_phy_speed" to add support for KSZ9031

```
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

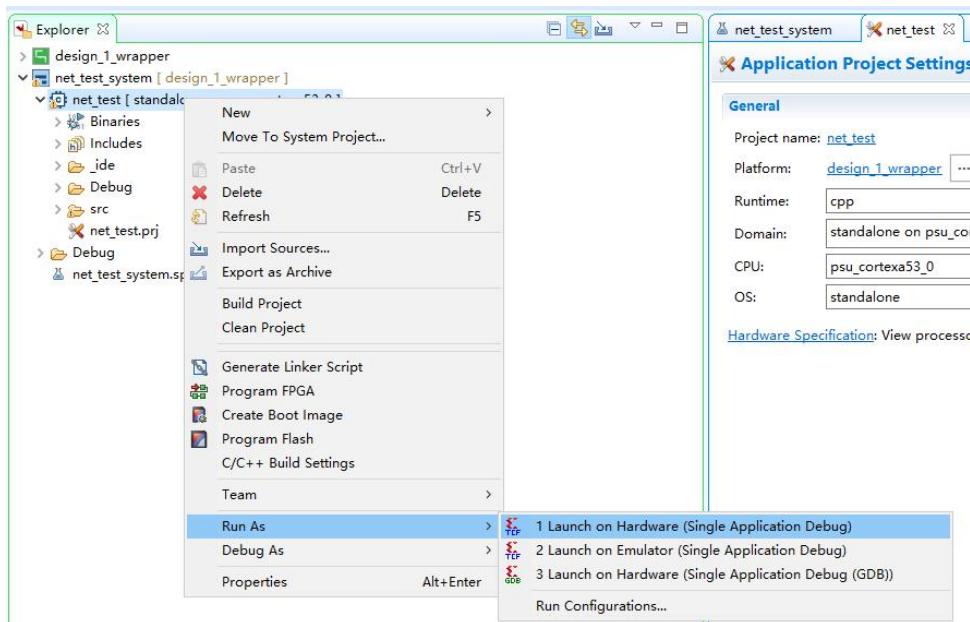


Part 9.2: Download Debugging

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

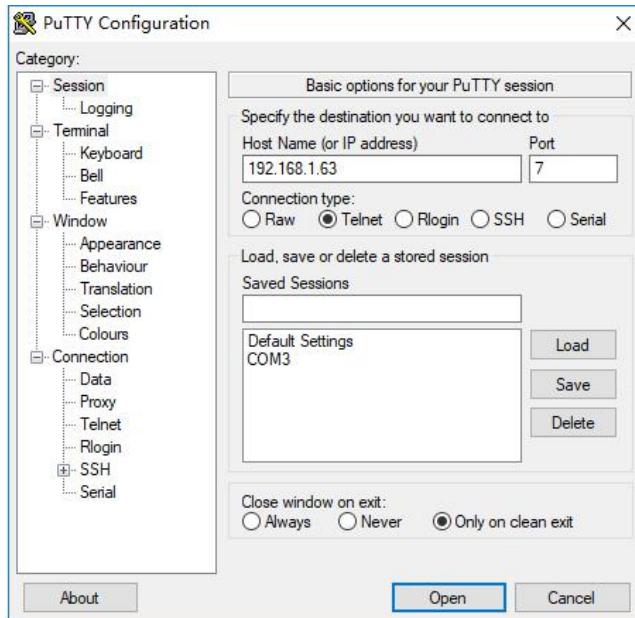
- 1) Connect the serial port to open the serial debugging terminal, connect the PS Ethernet cable to the router. Run the Vitis to Download the Programs



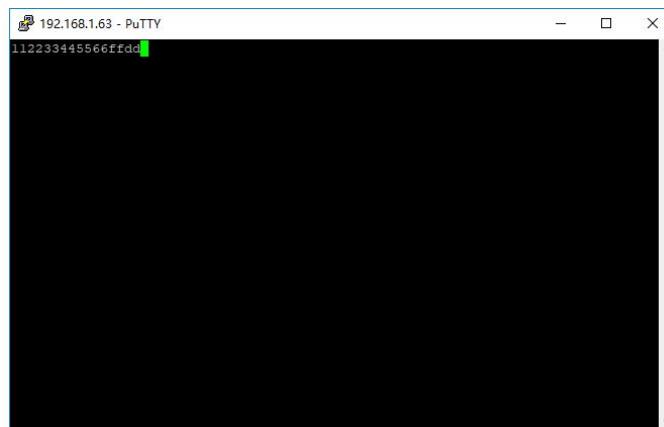
- 2) You can see that the serial port prints some information, you can see that the address is automatically obtained as "192.168.1.68", the connection speed is 1000Mbps, and the tcp port is 7

```
----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.

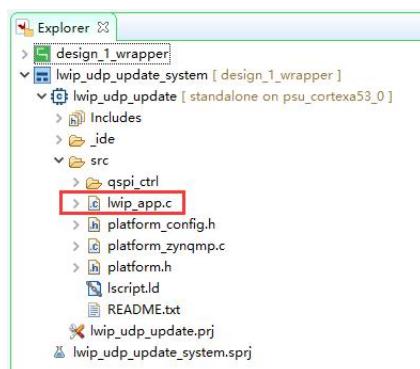
Software Engineer Job Content

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

Part 10.1: Vitis Program Development

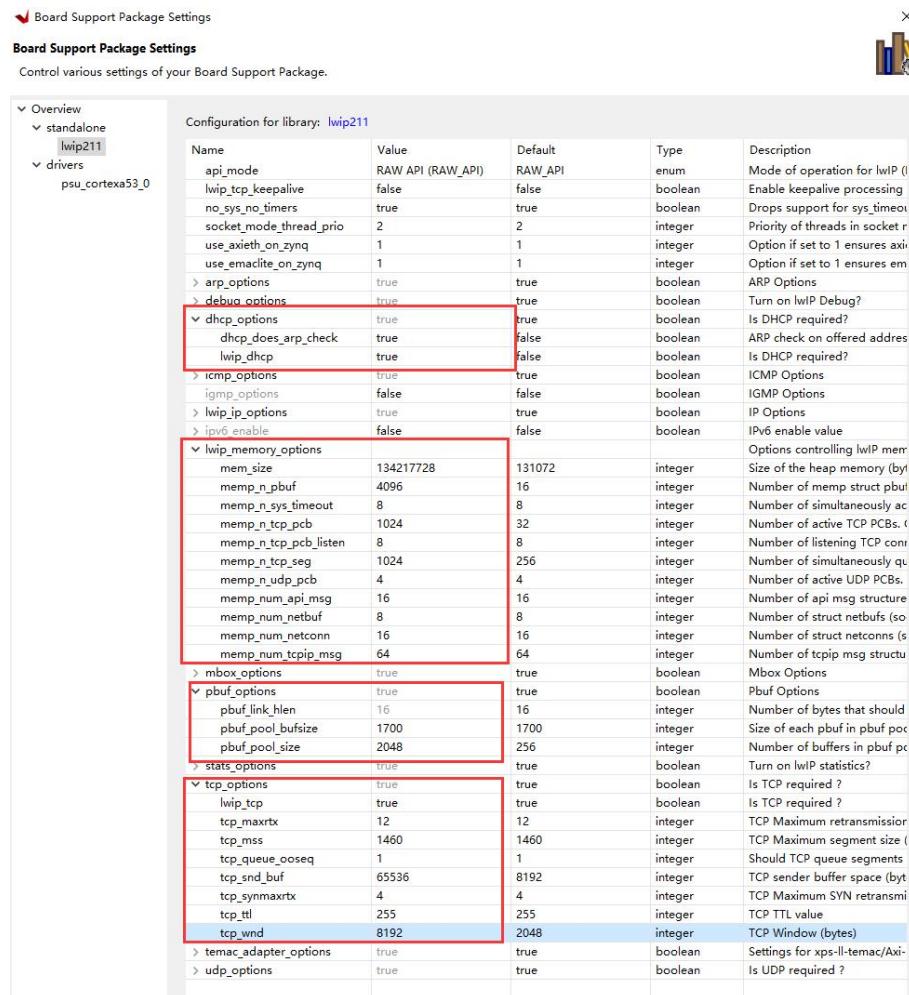
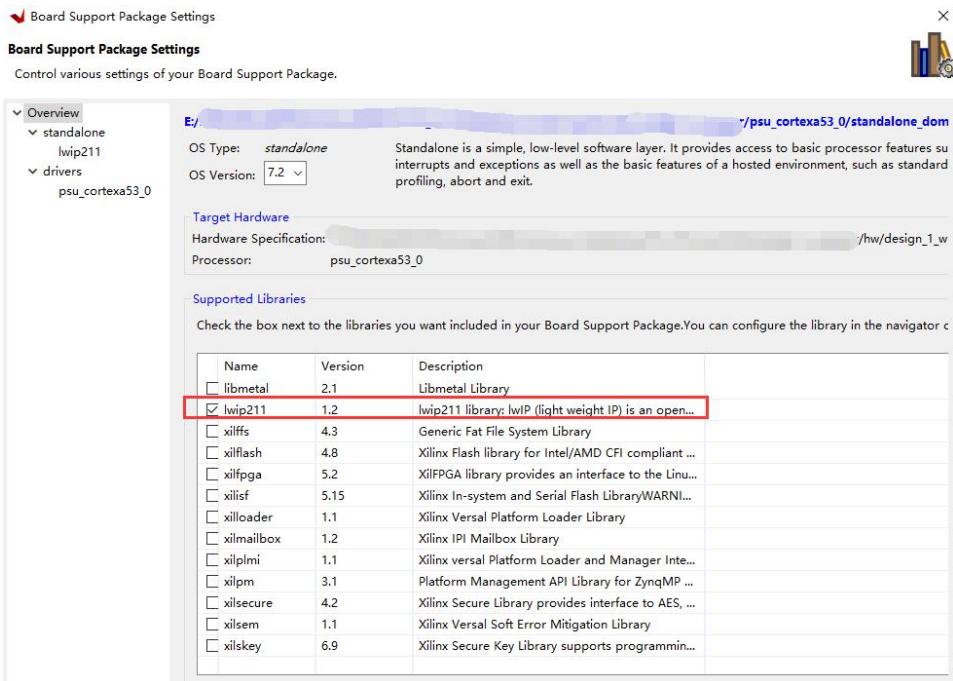
Part 10.1.1: UDP Transmission Mode

- 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.

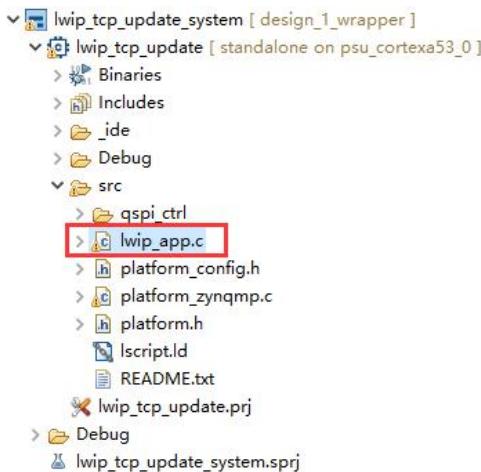


- 3) The “udp_receive” function is the set receive callback function. The main function is to receive the data, and buffer the received data into the “FlashRxBuffer” space, leaving it to be used for updating the “Flash”. After sending the data, send the "update" command to start updating the “flash”. 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

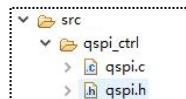
- 1) The “LWIP” part of “TCP” is also the “lwip_app.c” file. The control part refers to the “lwip echo” server routine to establish a “TCP Server”.



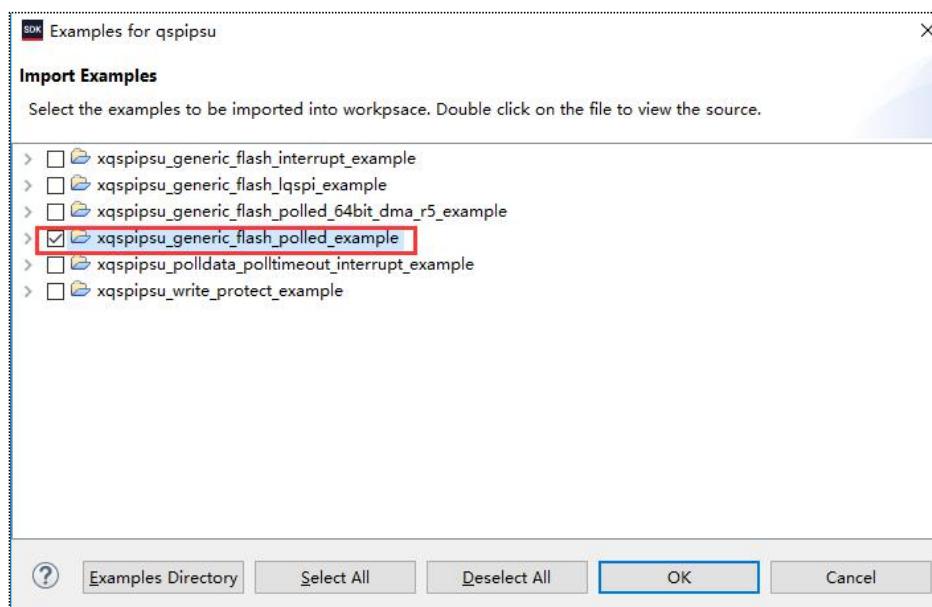
- 2) Similar to UDP, in the recv_callback receive callback function, the received BIN file is cached. The update command is also updated, and other parts are similar to UDP.

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 TotoalLen, char *FlashDataToSend) ;
int FlashReadID(XQspiPsu *QspiPsuPtr);
int FlashErase(XQspiPsu *QspiPsuPtr, u32 Address, u32 ByteCount, u8 *WriteBfrPtr);
int FlashWwrite(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

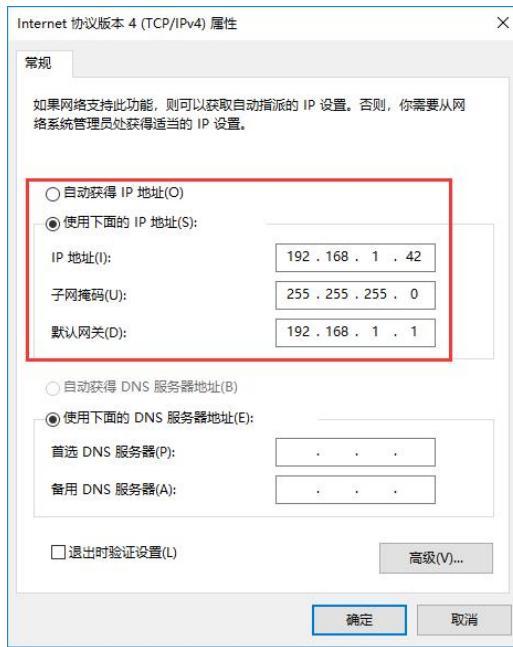
entire Flash. Instead, the Sector erase is done according to the TotalLen size, so the erased space will be slightly larger than the TotalLen; then the Flash is written, the FlashWrite function is used for writing; the final is the check, the data is read from the Flash, and written. The data is compared.

```
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.

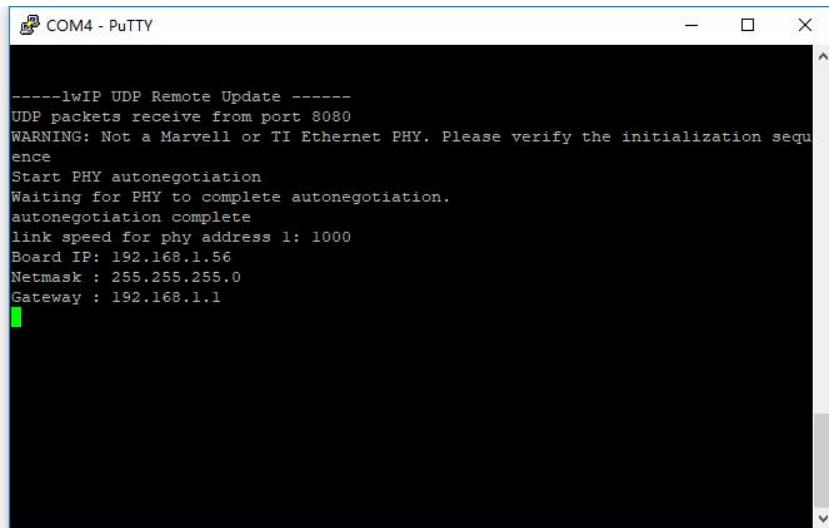
- 1) First connect the development board as follows, connect the network cable to port.
- 2) If there is a DHCP server, the IP will be automatically assigned to the development board; if there is no DHCP server, the default FPGA development board IP address is 192.168.1.10, and the IP address of the 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



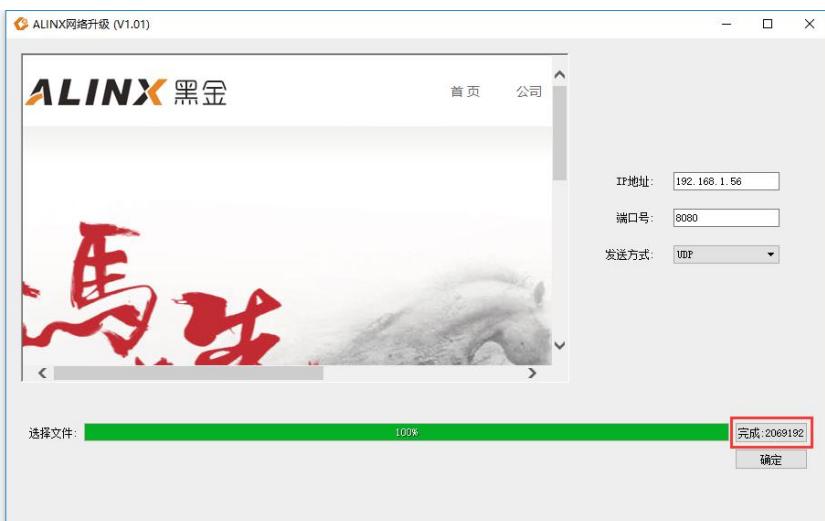
- 2) Open the board network upgrade software in the project directory.



- 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.



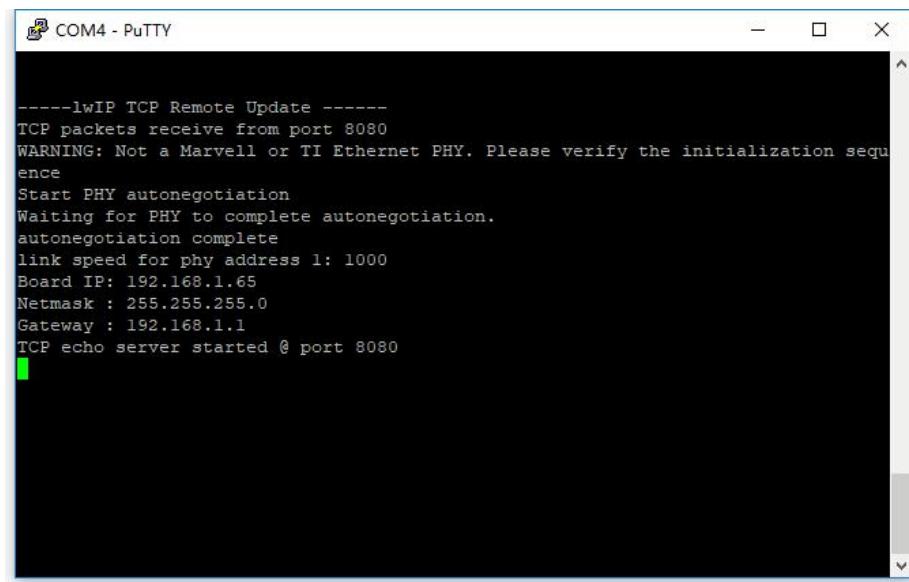
5) In the putty window you can see the number of bytes received by the board, as well as the erase, burn, and verify processes

The text in the terminal window is highlighted with a red box, focusing on the 'Received Size is 2069192 Bytes' line and the subsequent process logs.

- 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
```

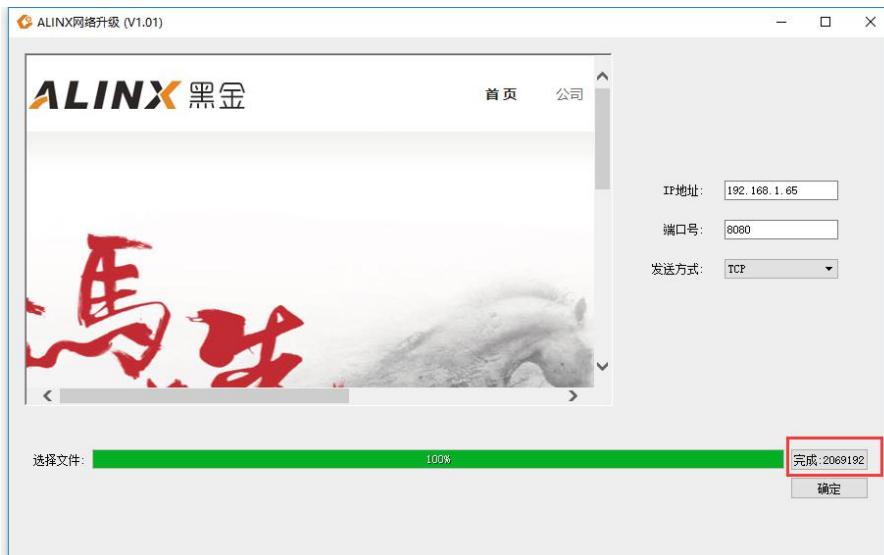
- 2) Open the board network upgrade software in the project directory.



- 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.
```

The terminal window shows a log of operations. A red box highlights the line 'Received Size is 2069192 Bytes', indicating the number of bytes received by the board. Other lines show the process of initialization, programming memory, performing an erase operation (size 2097152 Bytes), program operation (size 100%), verify operation (size 100%), and successful completion of each step.

- 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

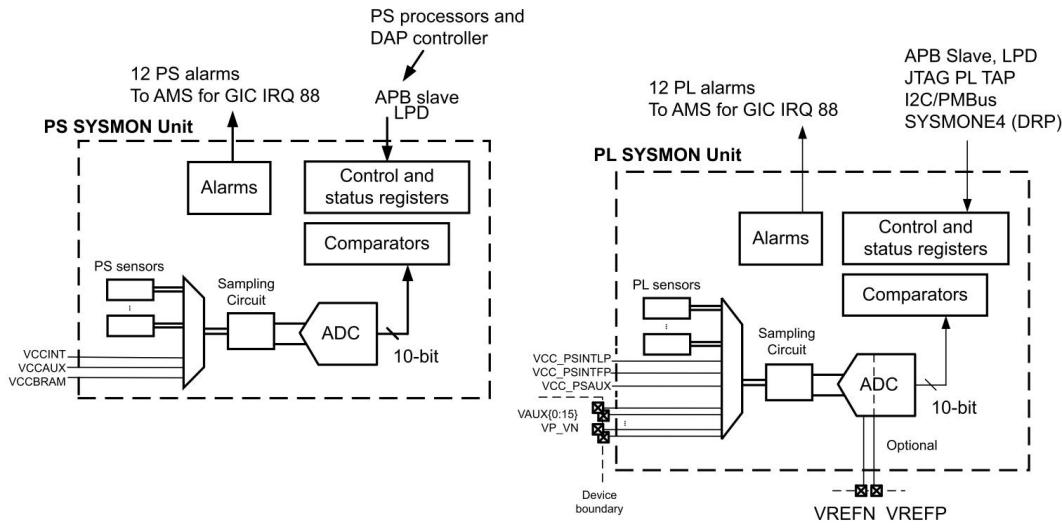
The vivado project directory is "ps_hello/vivado"

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

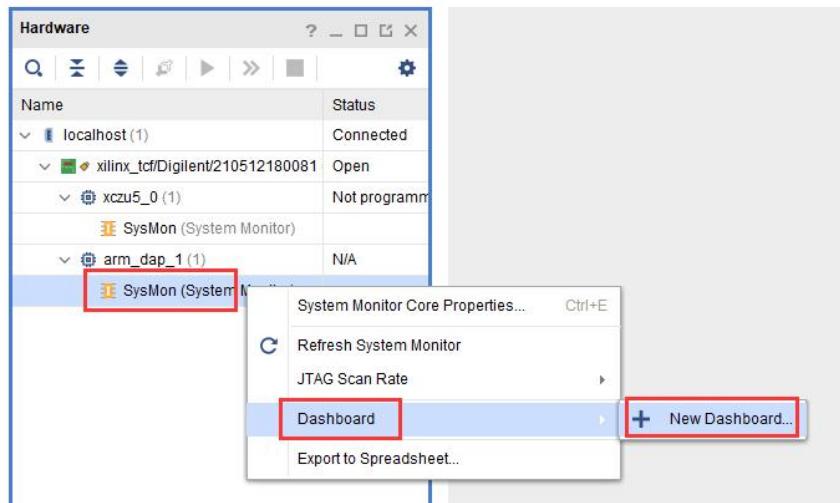
The Vivado project is also based on the "ps_hello" project.

Part 11.1: Hardware Read System Monitor

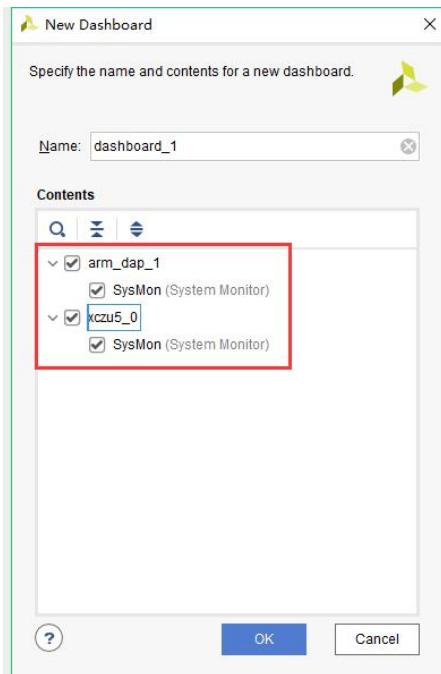
- 1) Open the project, connect the FPGA development board power supply, JTAG downloader, and adjust the development board to **JTAG mode**, power on the development board, click **Open Hardware Manager**, and then click **Auto Connect** to find the hardware.



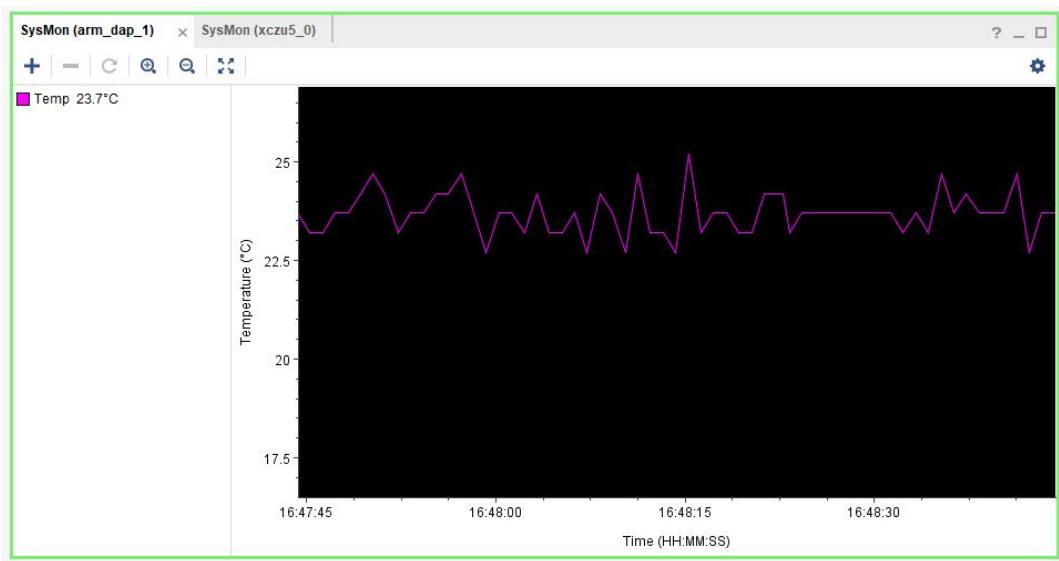
- 2) Right-click and select SysMon, create a new Dashboard



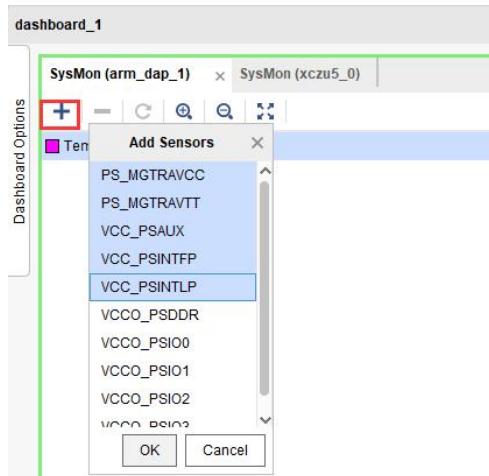
3) Select both PS and PL, click OK



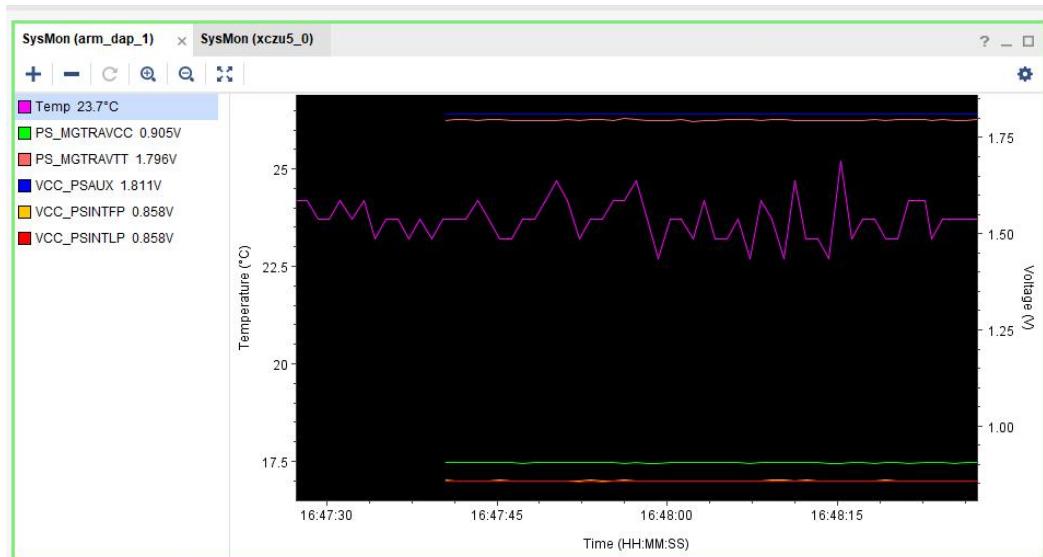
4) There will be temperature information by default



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

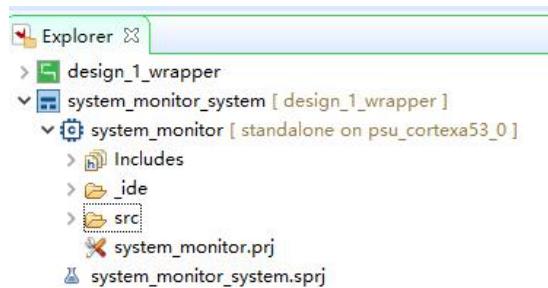
intuitive, but the disadvantage is that the data value cannot be obtained. The following describes how PS reads XADC information.

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	Documentation Link	Import Examples
psu_apm_0	axipmon	Documentation Link	Import Examples
psu_apm_1	axipmon	Documentation Link	Import Examples
psu_apm_2	axipmon	Documentation Link	Import Examples
psu_apm_5	axipmon	Documentation Link	Import Examples

- 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.

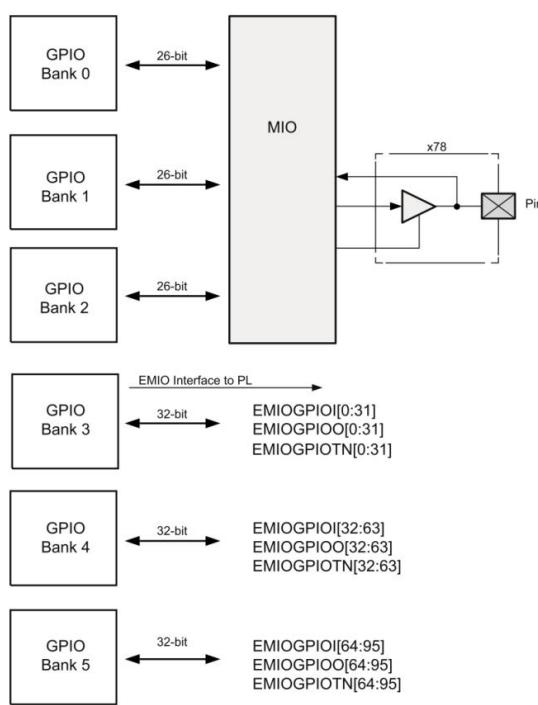


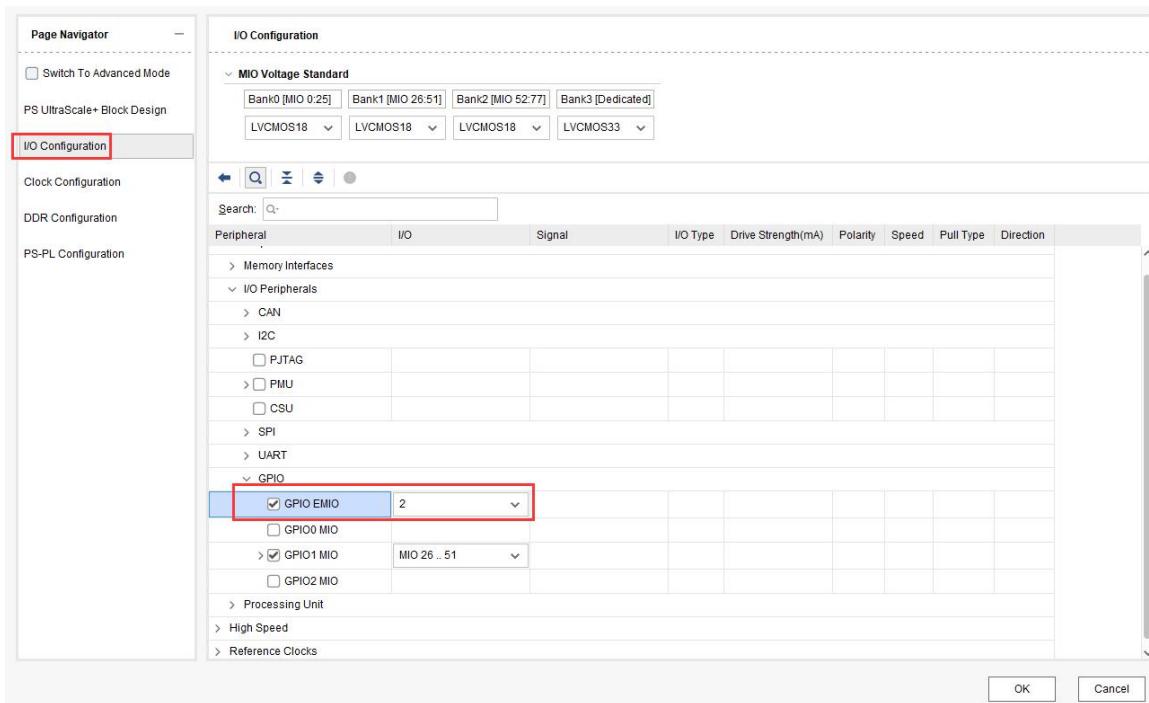
Figure 27-1: GPIO Block Diagram

FPGA Engineer Job Content

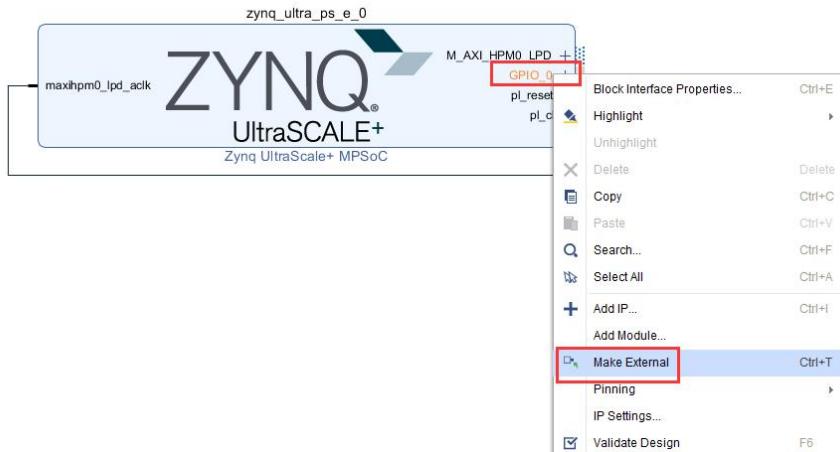
The following is the content that FPGA engineers are responsible for.

Part 12.2: Create a Vivado Project

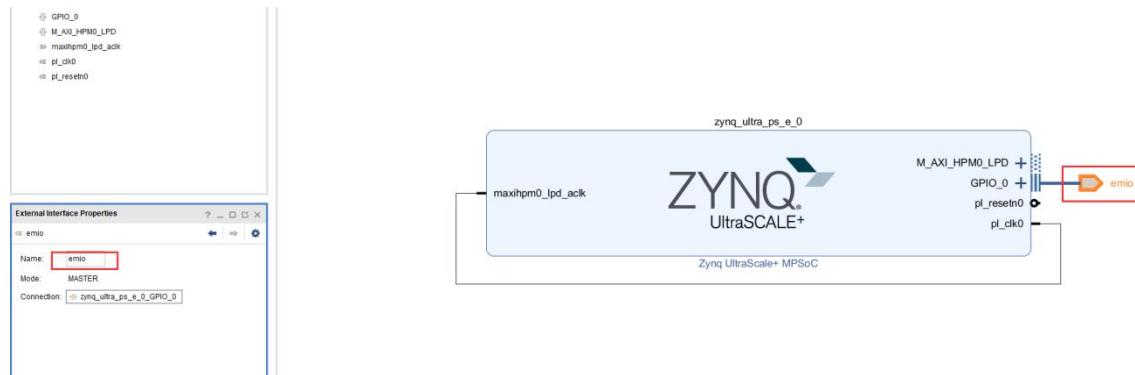
- 1) Based on the “ps_hello” project, save as a project named “ps_emio”, open the ZYNQ configuration, and check “GPIO EMIO”. Because there is 1 LED and 1 Key on the PL side, the bit width of EMIO is selected as 2 bits in the MIO configuration. After the configuration is completed, click OK.



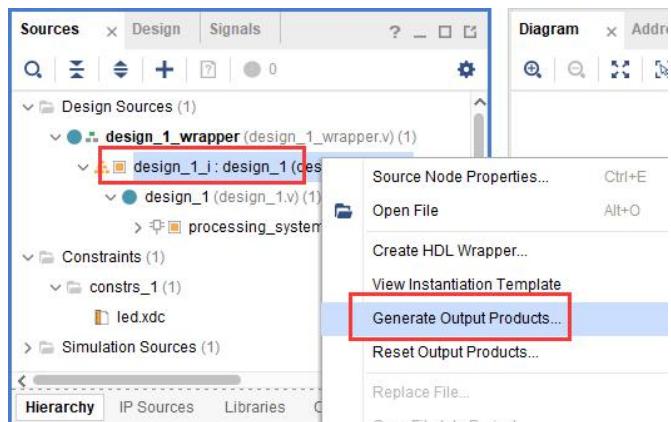
- 2) Click the extra “GPIO_0” port, right-click and select “Make External” to export the port signal



- 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

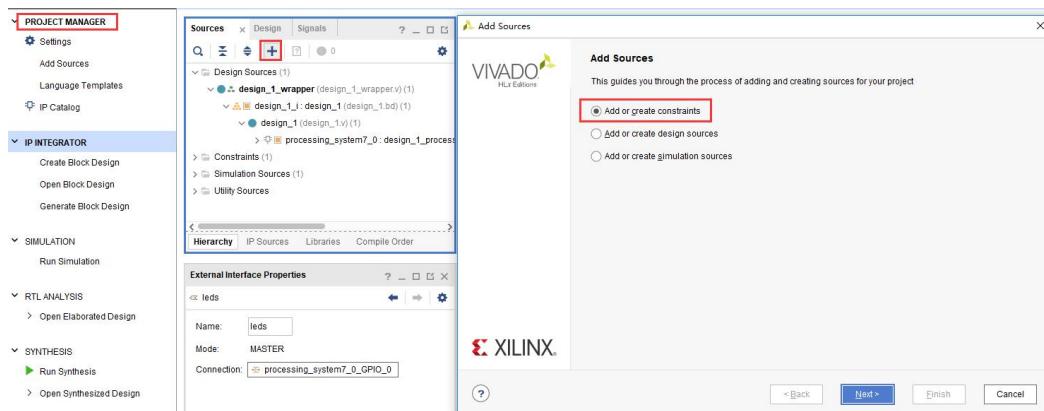
```

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;

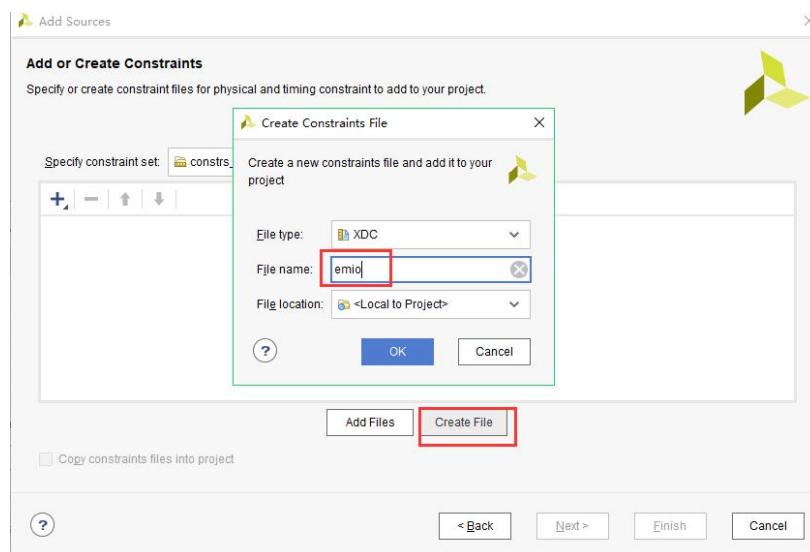
```

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

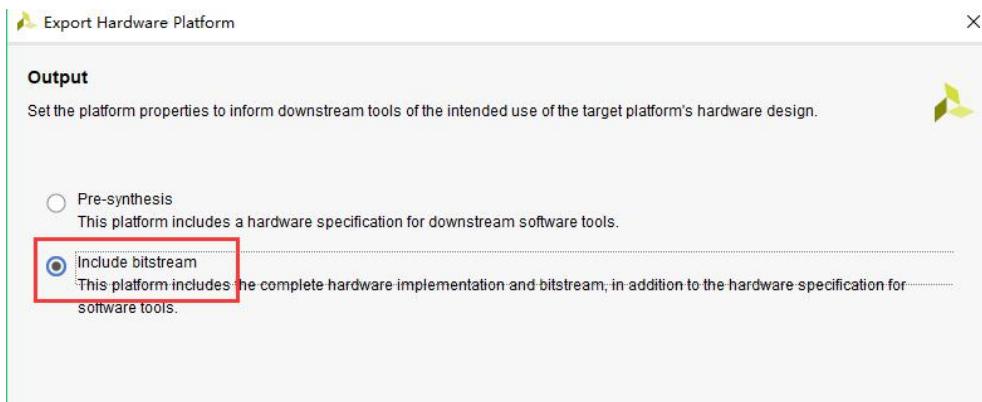
```
#####Compress Bitstream#####
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]

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"



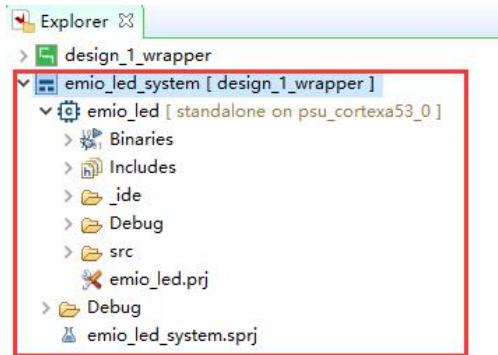
Software Engineer Job Content

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);

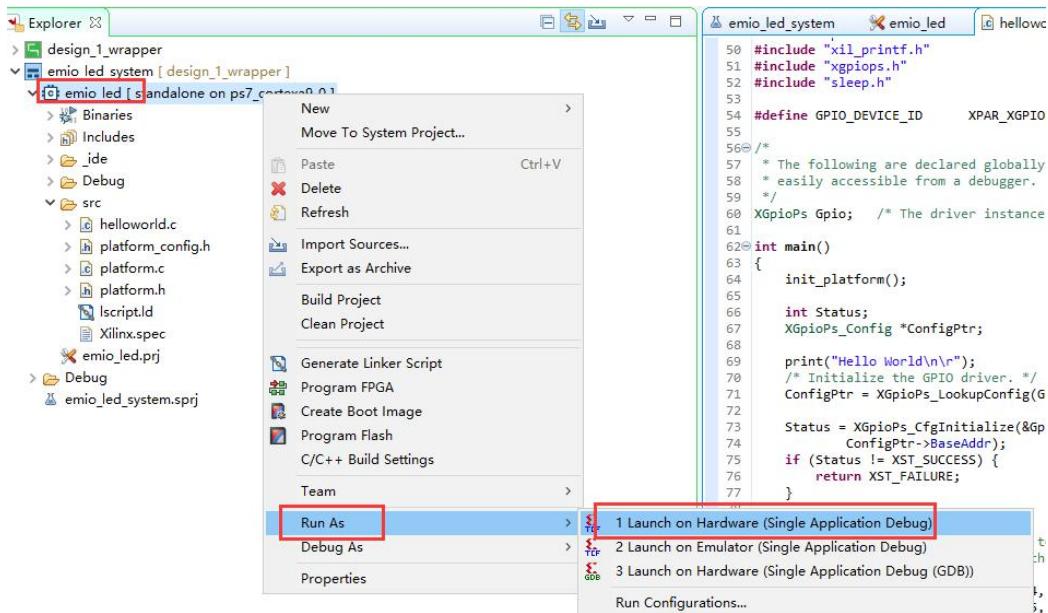
    Status = XGpioPs_CfgInitialize(&Gpio, ConfigPtr,
                                    ConfigPtr->BaseAddr);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * 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

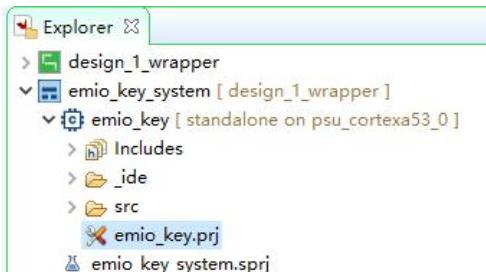


You can see the PL end LED Flashing.

Part 12.4.2: EMIO Implements PL Key Interrupt

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

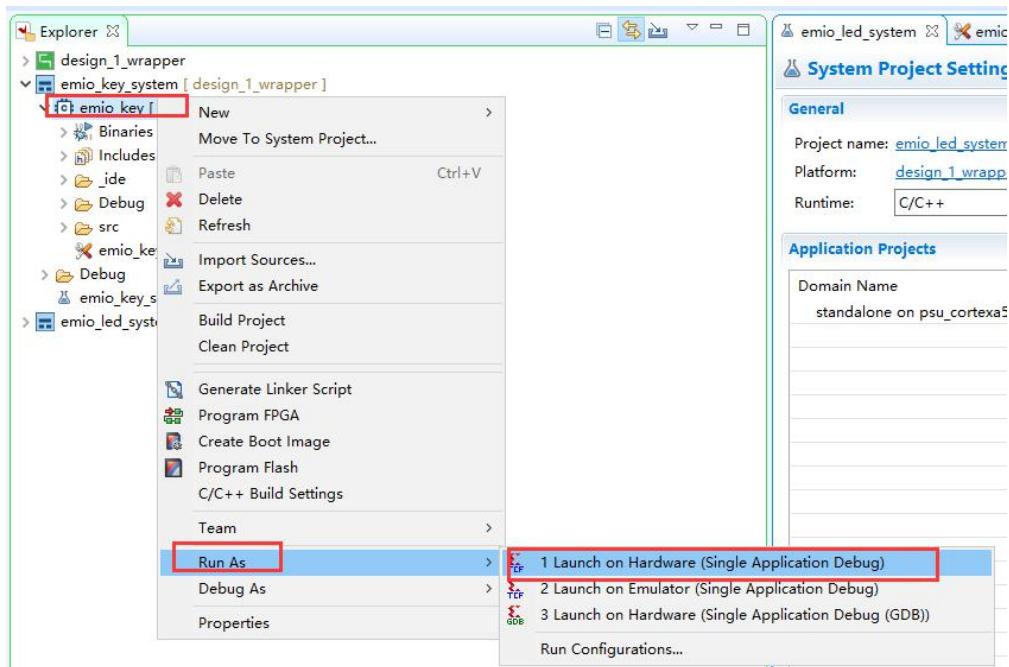
- 1) Create a new project named “emio_key” with a template of “hello world. Copy the routine program, save and compile



- 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”

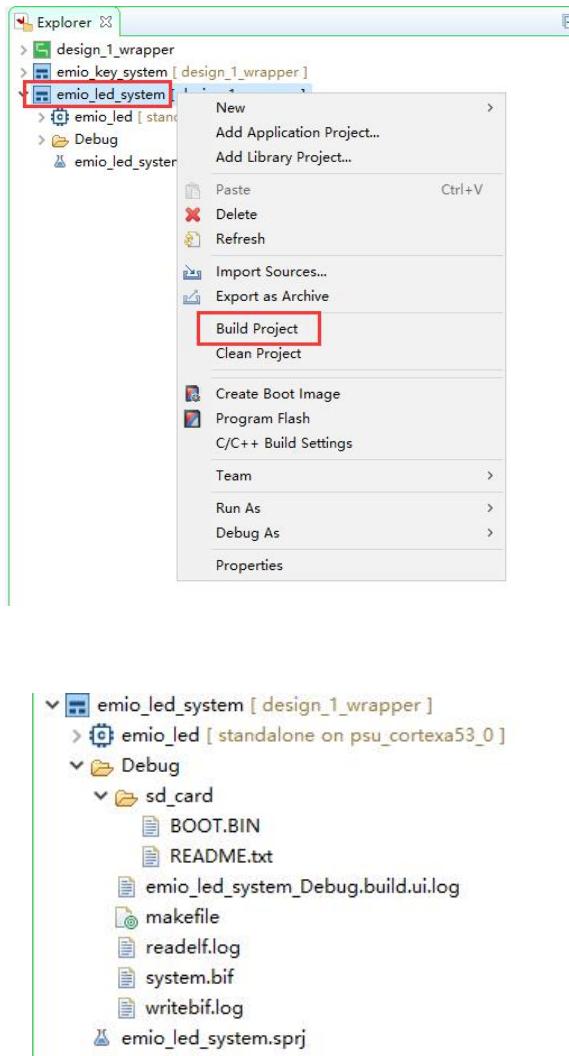


- 4) Observe the experimental phenomenon, press the PL end key PL_KEY, you can control the PL end LED PL-LED on and off.

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.

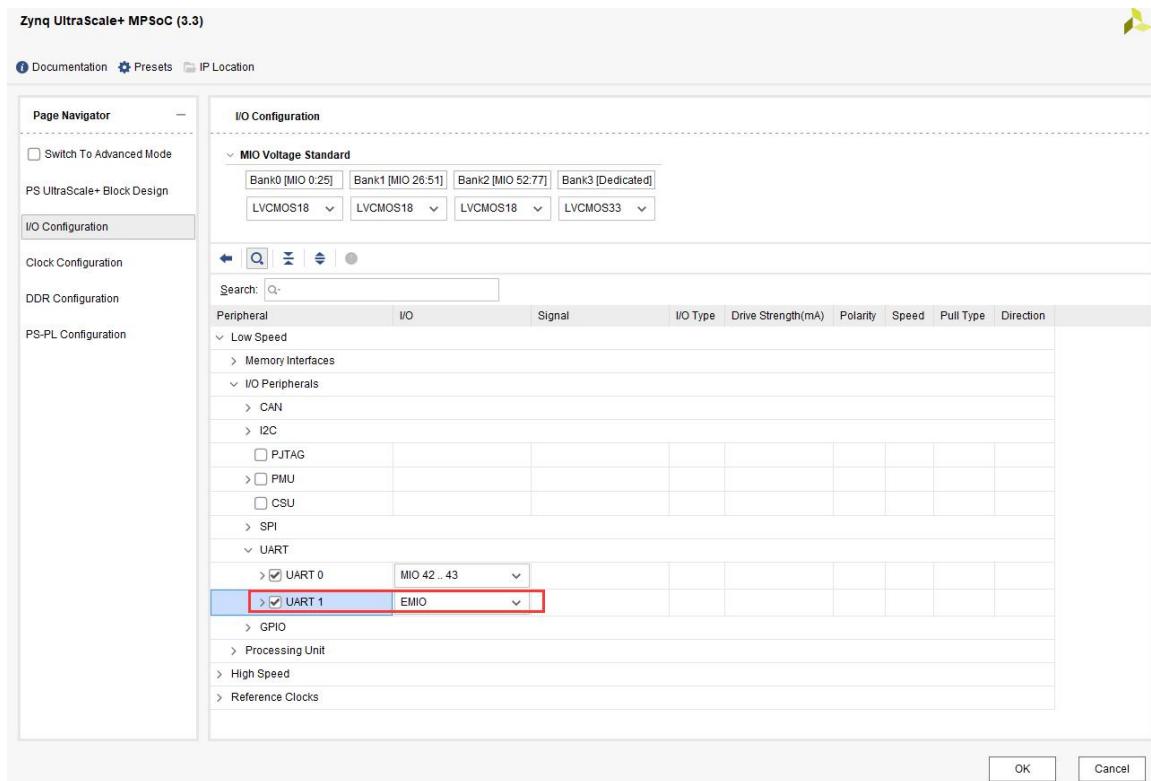


The software will automatically add three files, the first boot program is “fsbl.elf”, the second is the “bitstream” of FPGA, and the third is the application program “xx.elf”. Click “Create Image”, the download method is the same as above, and will not be repeated here.

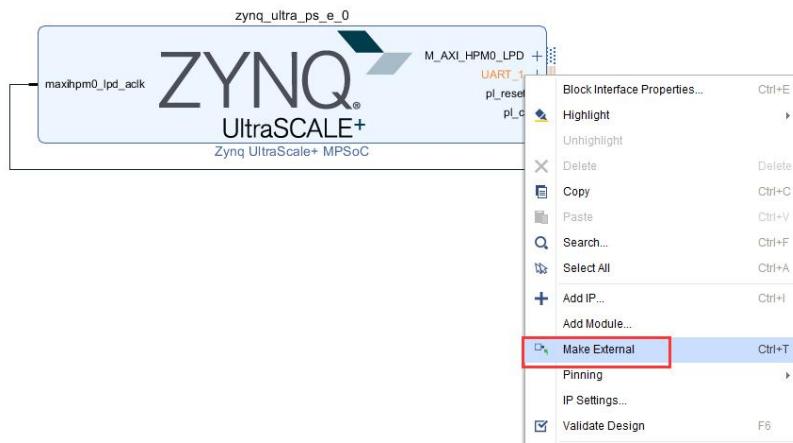
Part 12.6: EMIO Usage of UART Serial Port

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

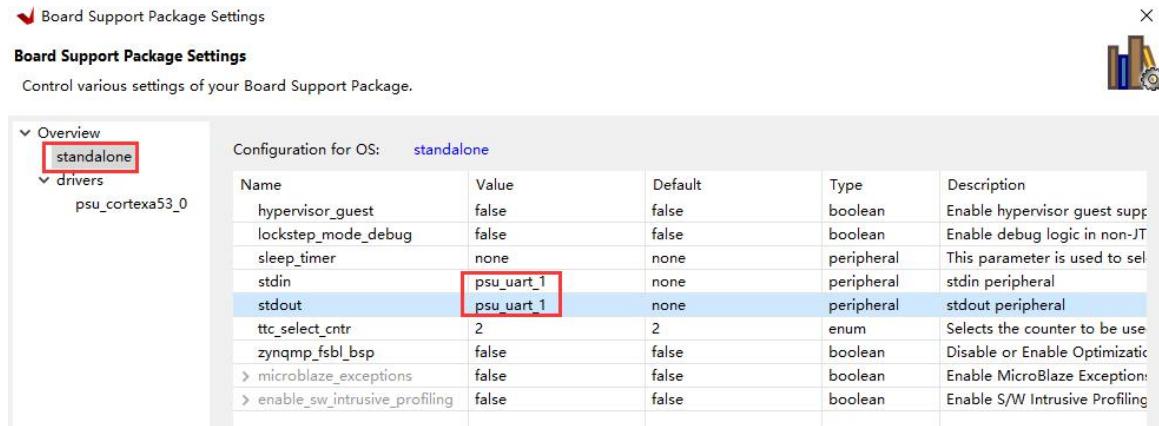


Bind the pins according to the schematic and recompile.

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

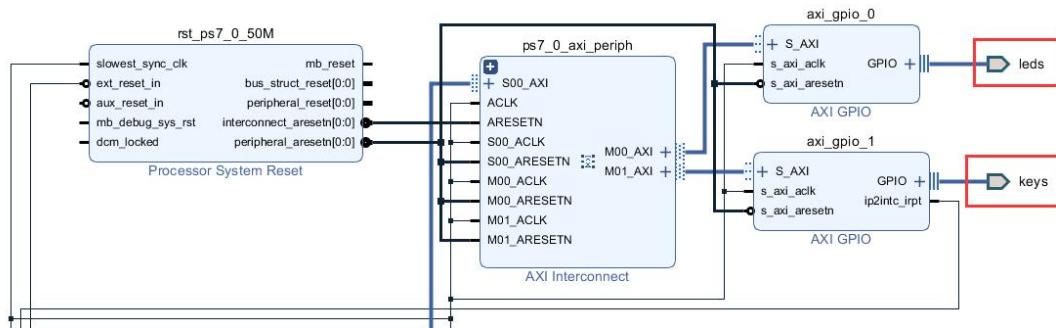
```
set_property PACKAGE_PIN AH12 [get_ports {uart_txd}]
set_property IOSTANDARD LVCMS33 [get_ports {uart_txd}]
```

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

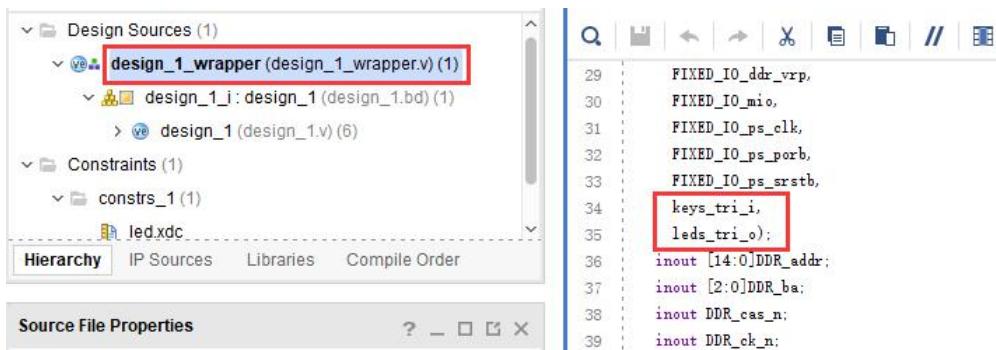


Part 12.7: Pin Binding Common Errors

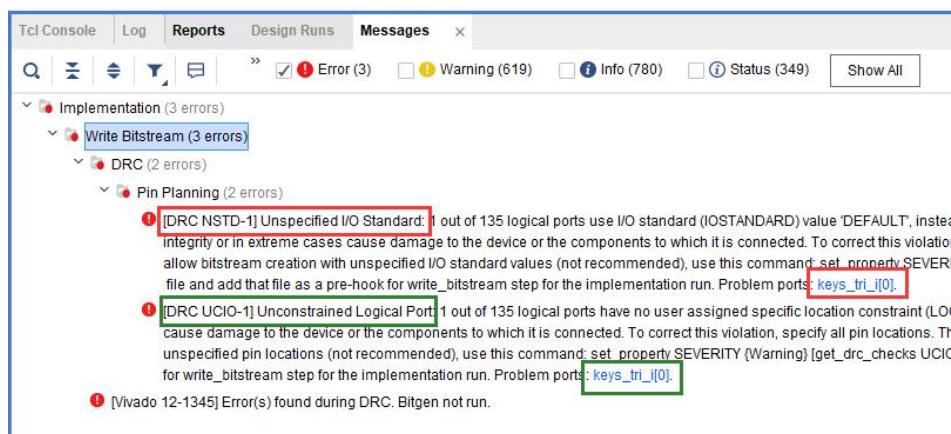
- 1) In the block design design, for example, the pin name of the GPIO module is set to leds and keys. Many people bind pins according to such names in XDC files.



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.

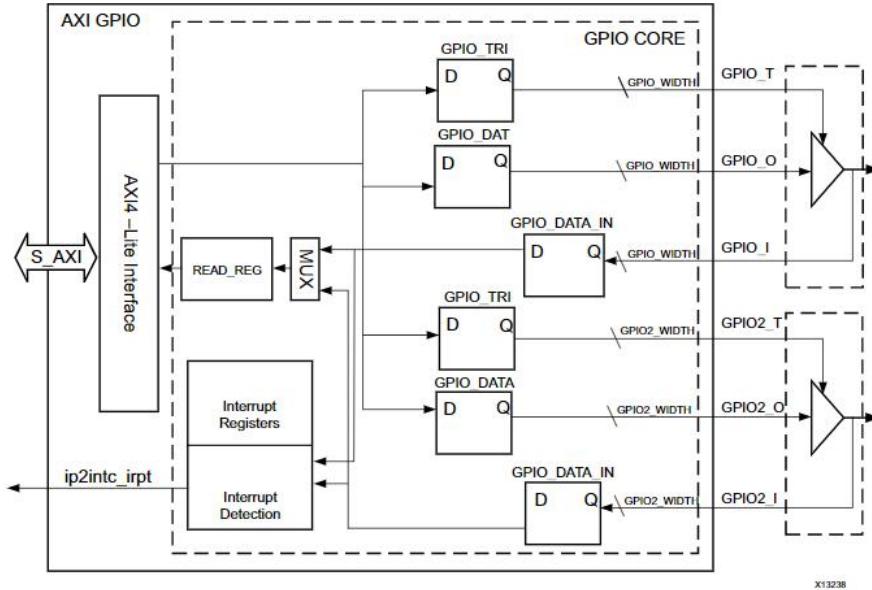


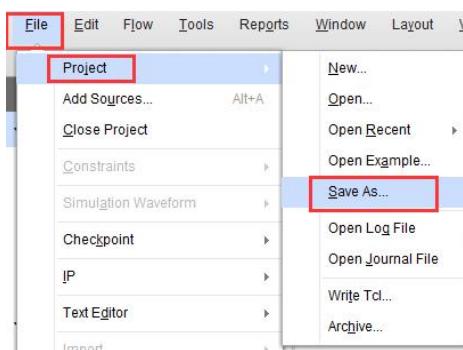
Figure 1-1: AXI GPIO Block Diagram

FPGA Engineer Job Content

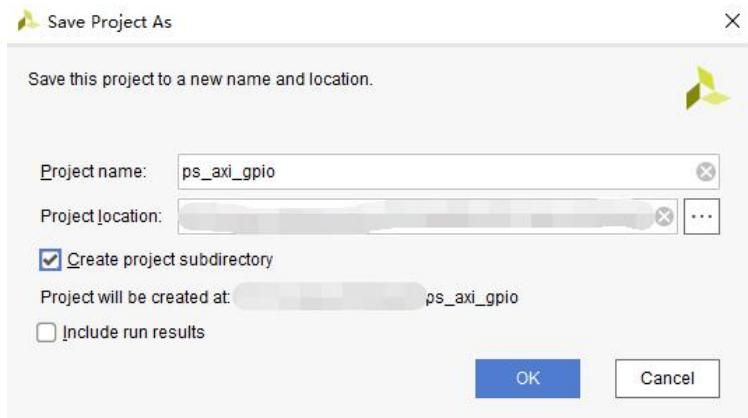
The following is the content that FPGA engineers are responsible for.

Part 13.2: Create a “Vivado” Project

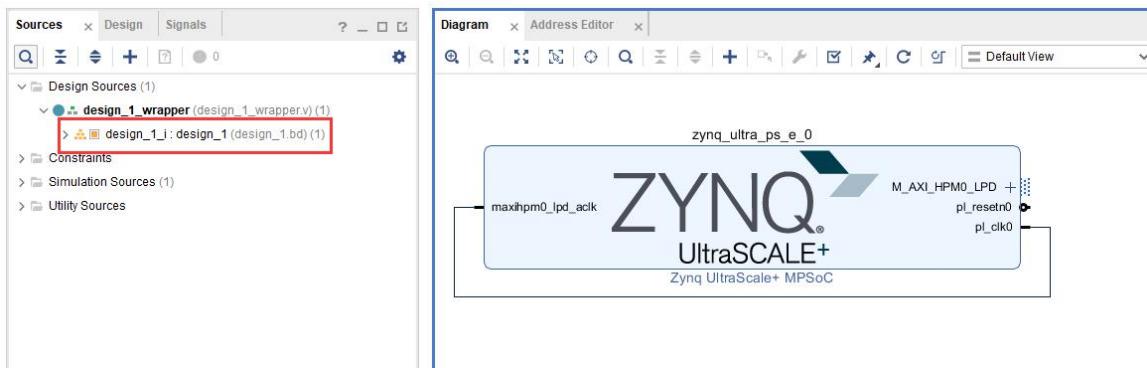
- 1) Open "ps_hello" and save as a "ps_axi_gpio" Vivado project, indicating that PS controls gpio through the AXI bus



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

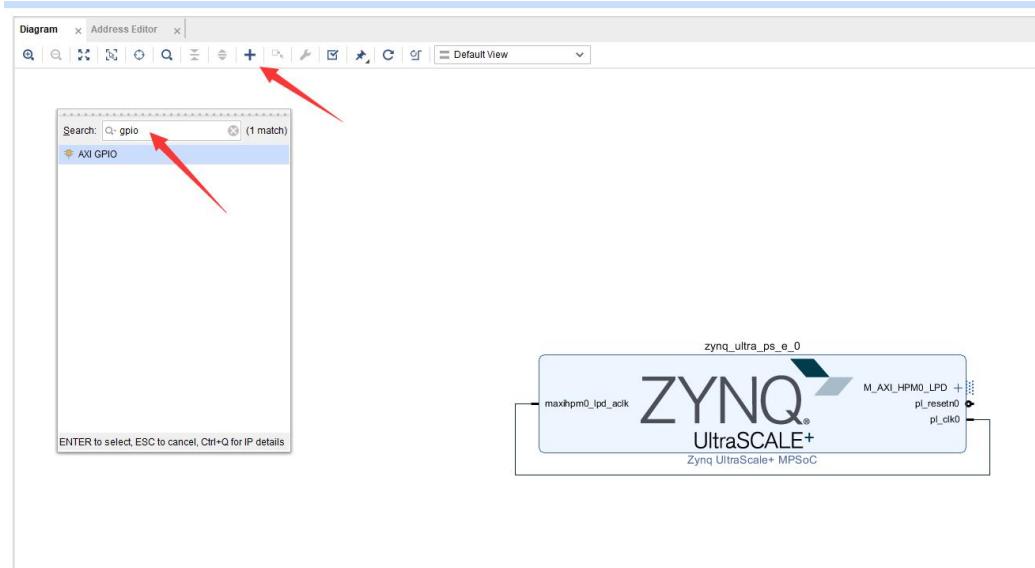


2) Double-click "xx.bd" to open the "block design"



Part 13.2.1: Add “AXI GPIO”

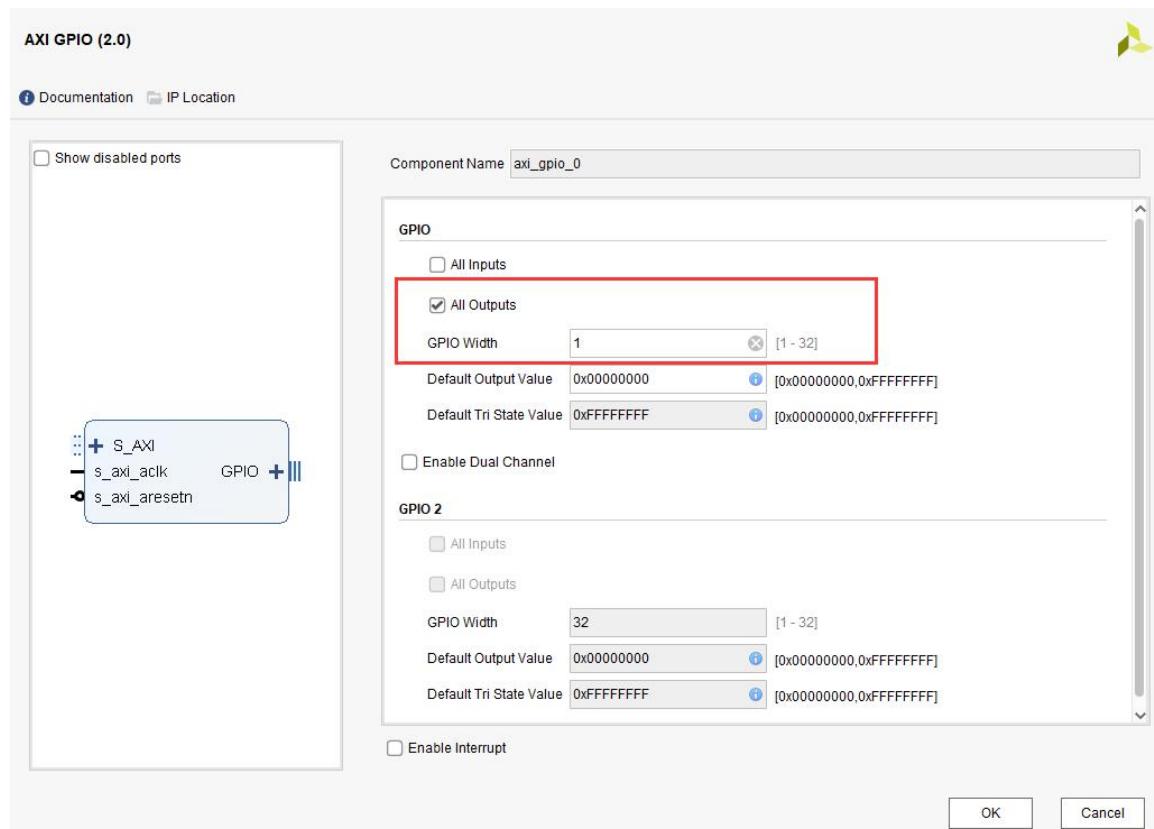
3) Add an AXI GPIO IP core



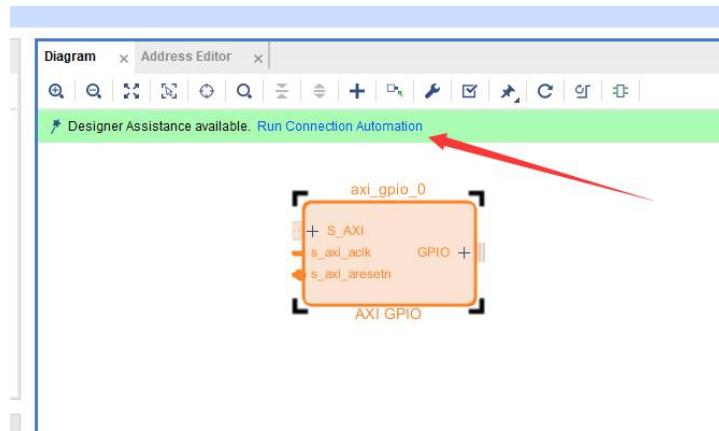
4) Double-click the "axi_gpio_0" just added, configure the parameters



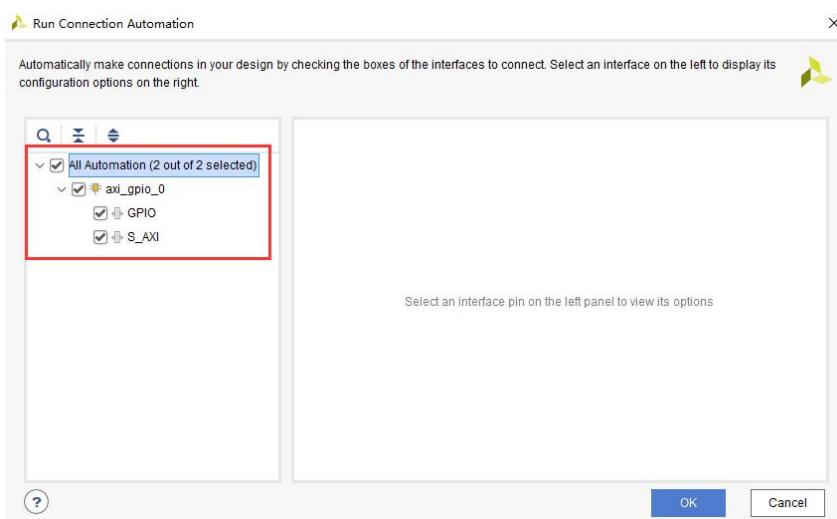
- 5) Select "All Outputs", because the LED is controlled here, as long as the output is OK. Fill in "GPIO Width" as 1, control 1 LED, and click OK. If you want to use "channel2", you need to turn on "Enable Dual Channel", which also enables GPIO2



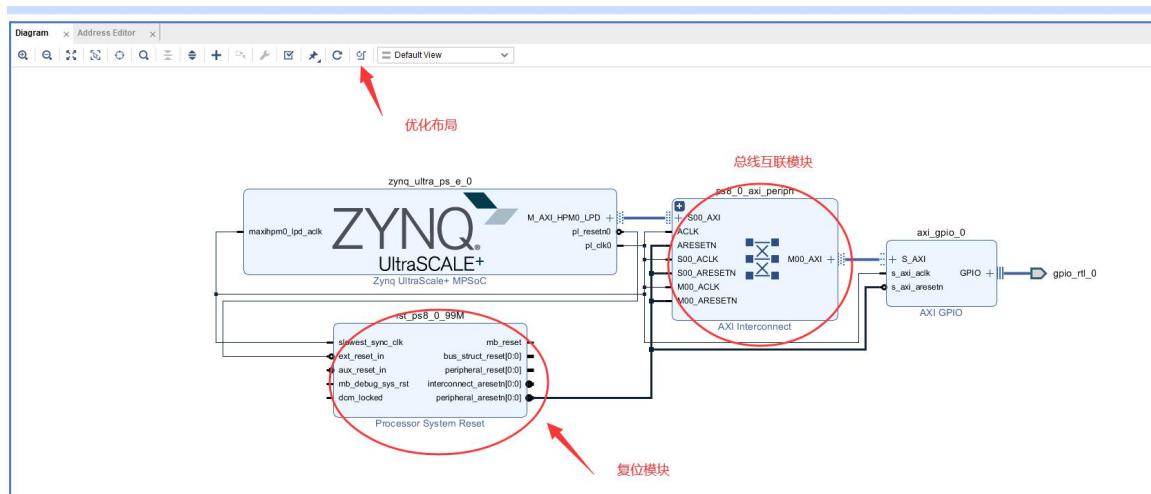
- 6) Click "Run Connection Automation" to complete some automatic connections



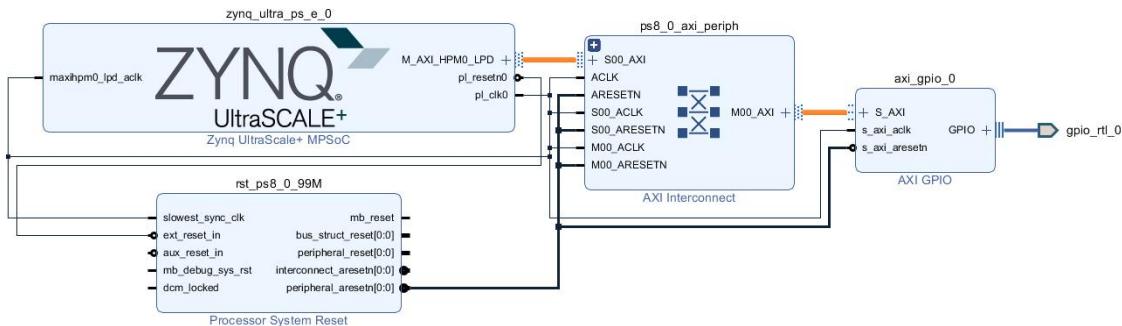
- 7) Select the port to be automatically connected, select all here, click OK



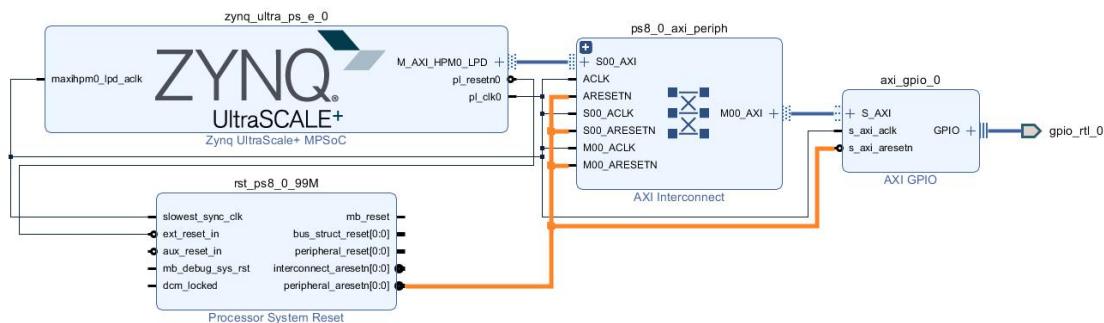
- 8) Click "Optimize Routing", you can optimize the layout, and you can see that there are two more modules. One is a “Processor System Reset” module, which provides a reset signal in the same clock domain as a synchronous reset module. The “AXI Interconnect module” is an AXI bus interconnection module,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



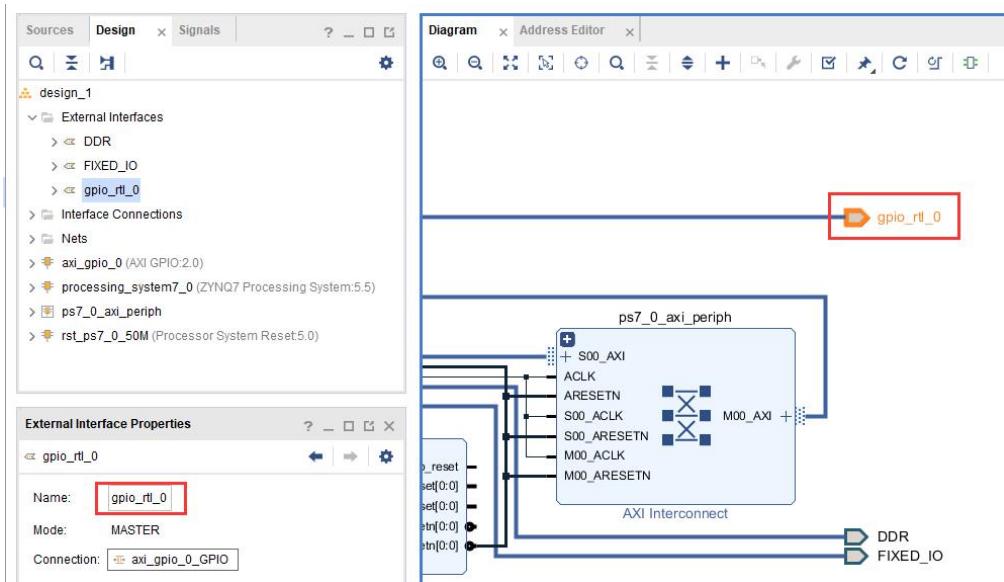
In this application, we can see that the HPM0_LPD port of ZYNQ is used. This interface is used to access PL data. In most applications, it is used to configure the registers of the PL module.



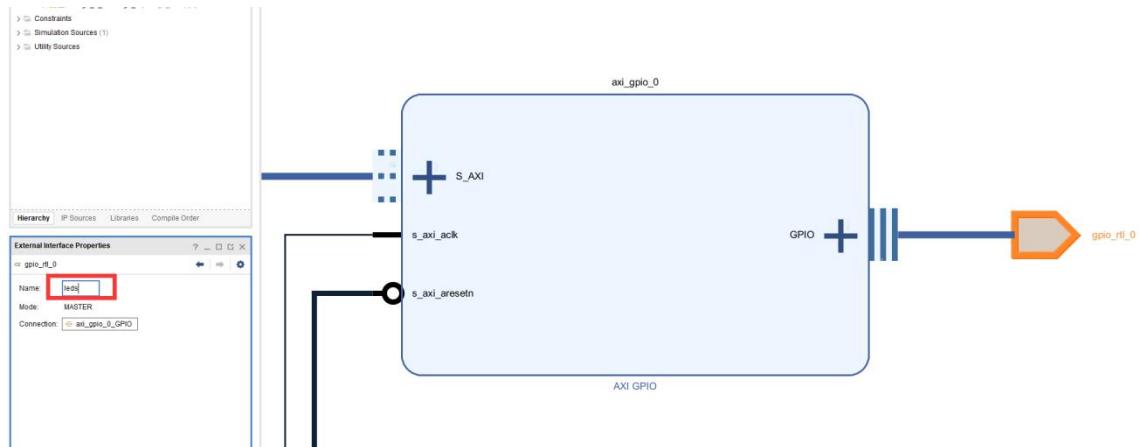
The reset signal is provided by the reset output of ZYNQ. It is best to add a reset module for each clock domain. You can search and add according to the name of the module.



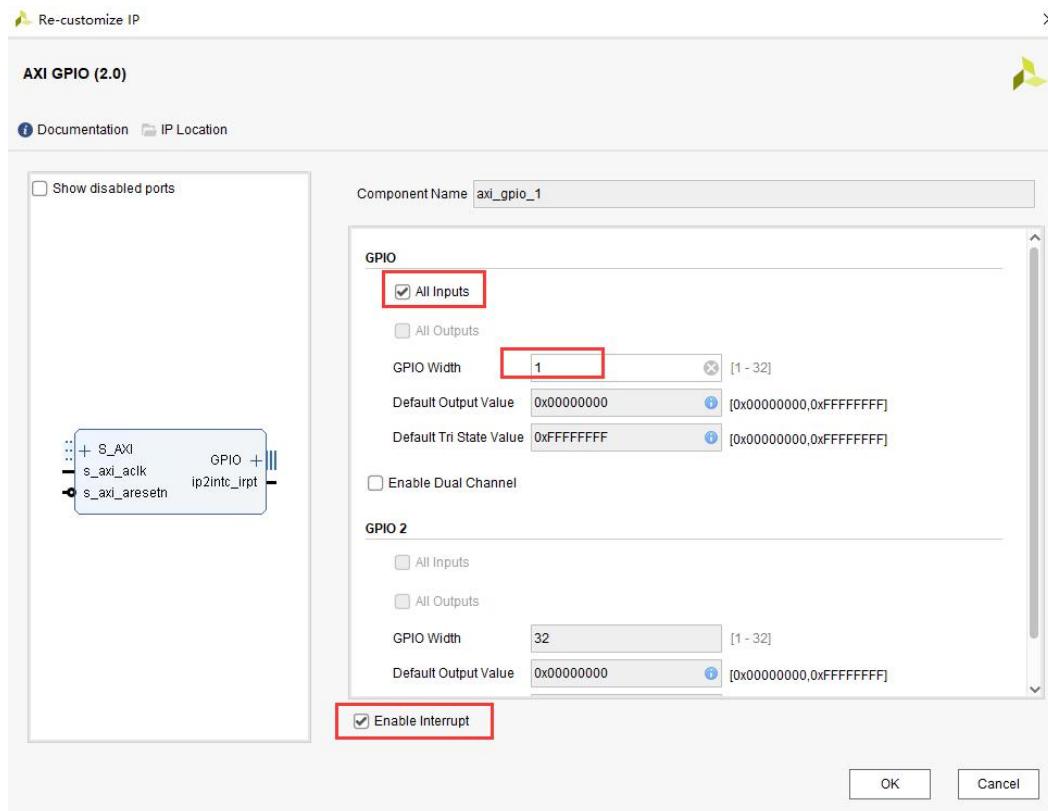
9) Modify the name of the GPIO port



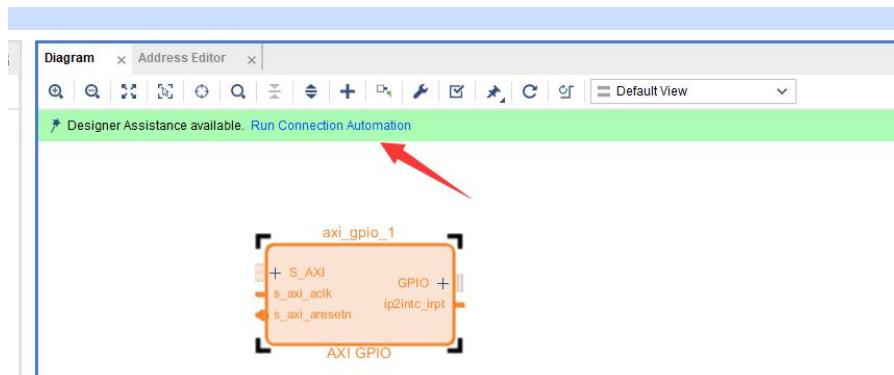
10) Name changed to leds



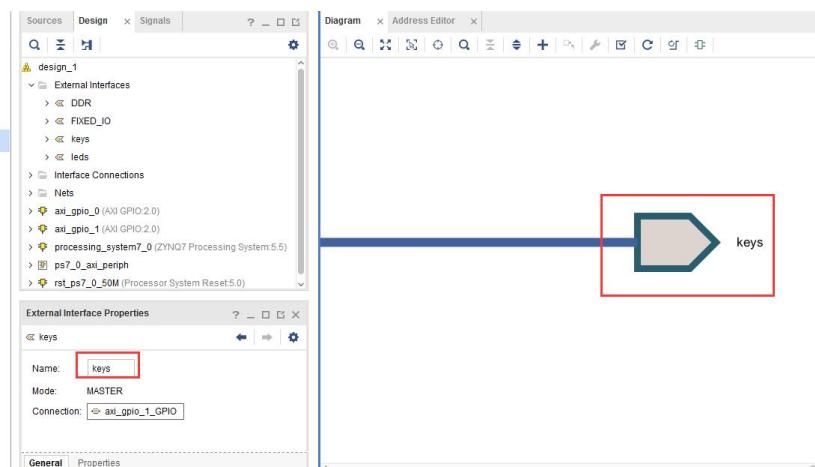
11) Add another “AXI GPIO”, connect the “PL” side key, configure “GPIO” parameters, all of which are input, width is 1, enable interrupt



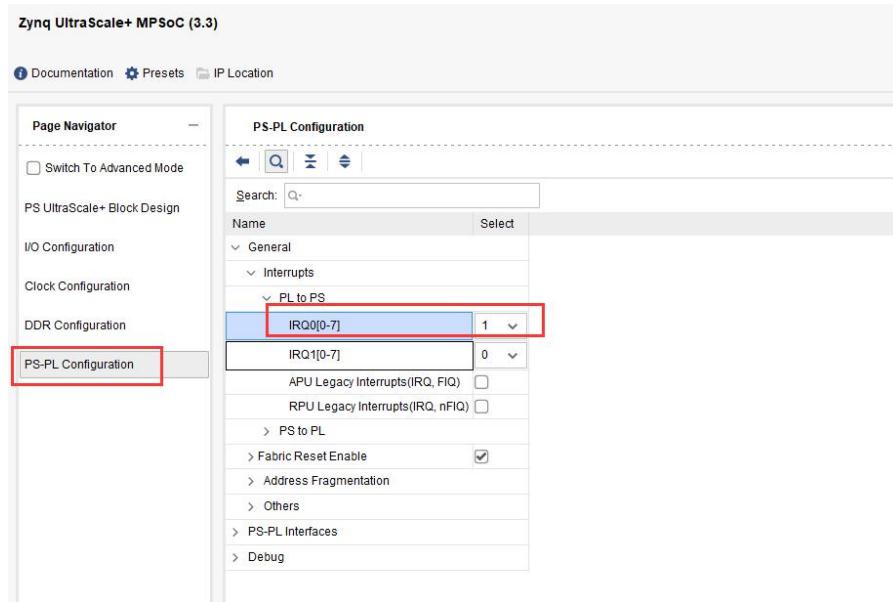
12) Use automatic connection



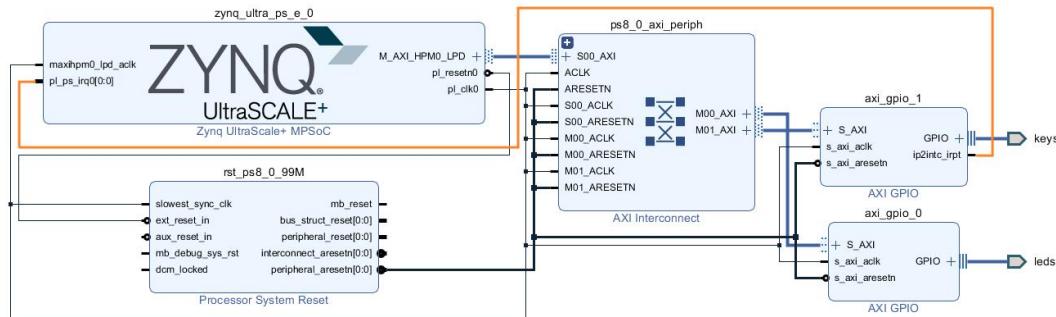
13) Change the port name to "keys"



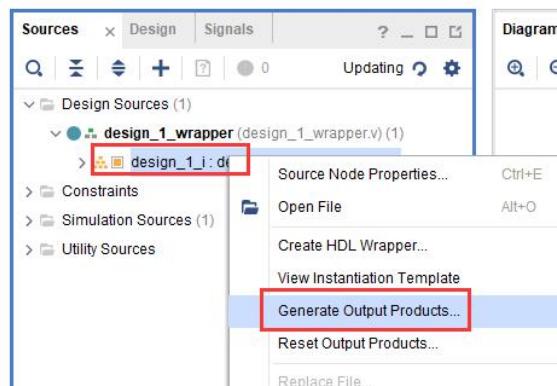
14) Since it is an interrupt from the PL side, here you need to configure the interrupt of the ZYNQ processor, Set IRQ0[0-7] to 1.



15) Connect “ip2intc_irpt” to “pl_ps_irq”

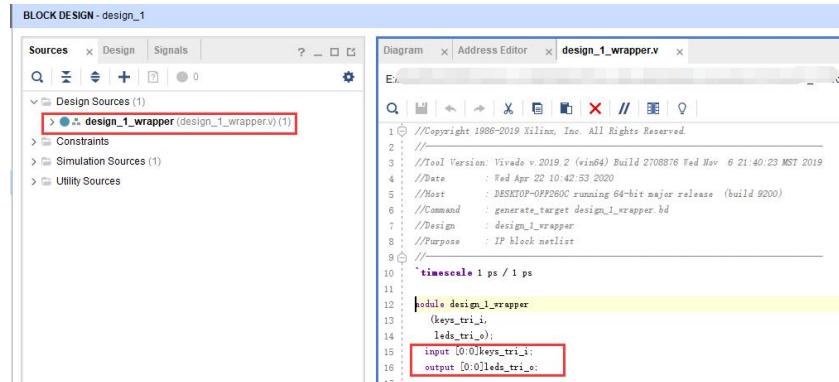


16) Save the design, click “xx.bd”, right-click “Generate Output Products”



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.



```

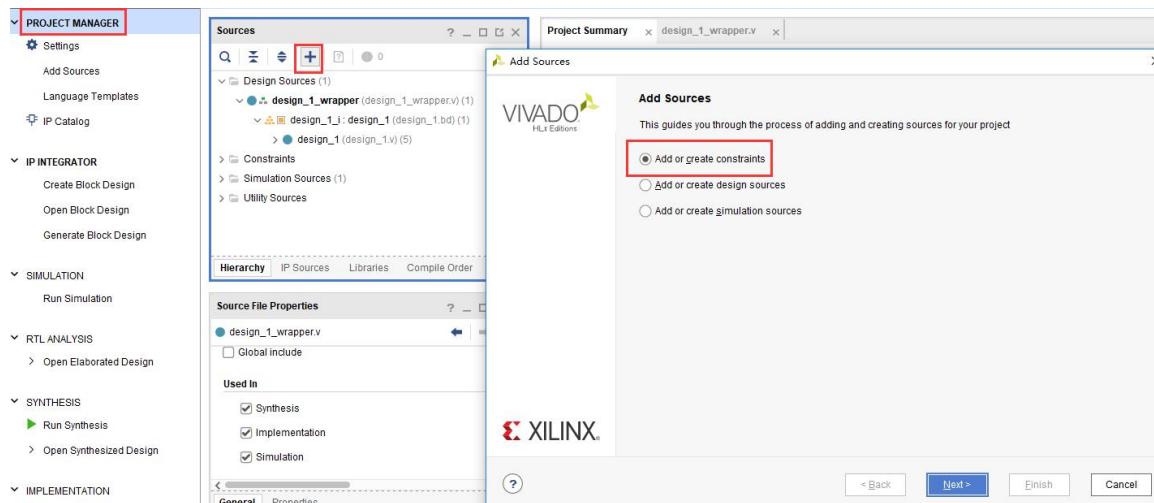
BLOCK DESIGN - design_1
Sources | Design | Signals | ? - □ 0
Design Sources (1)
> design_1_wrapper (design_1_wrapper.v)(1)
Constraints
Simulation Sources (1)
Utility Sources

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

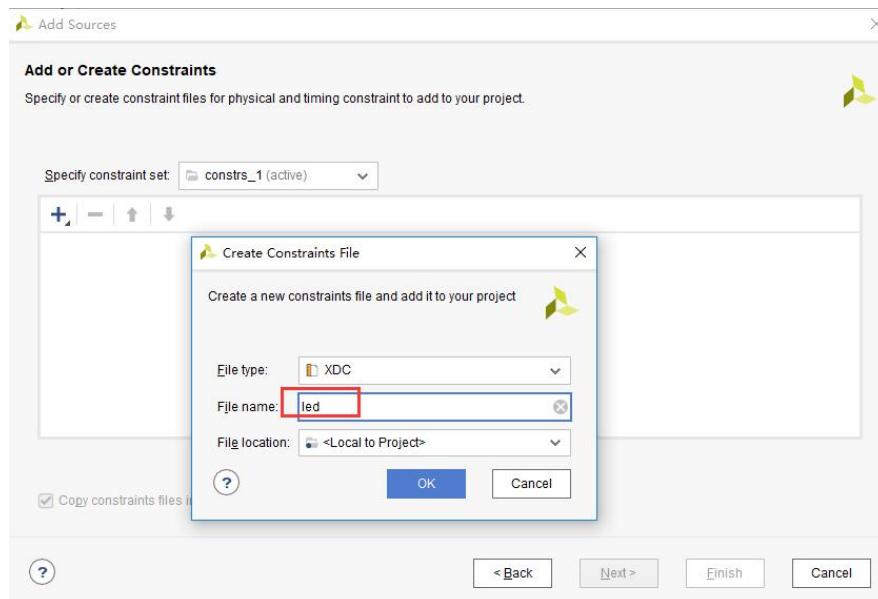
```

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]

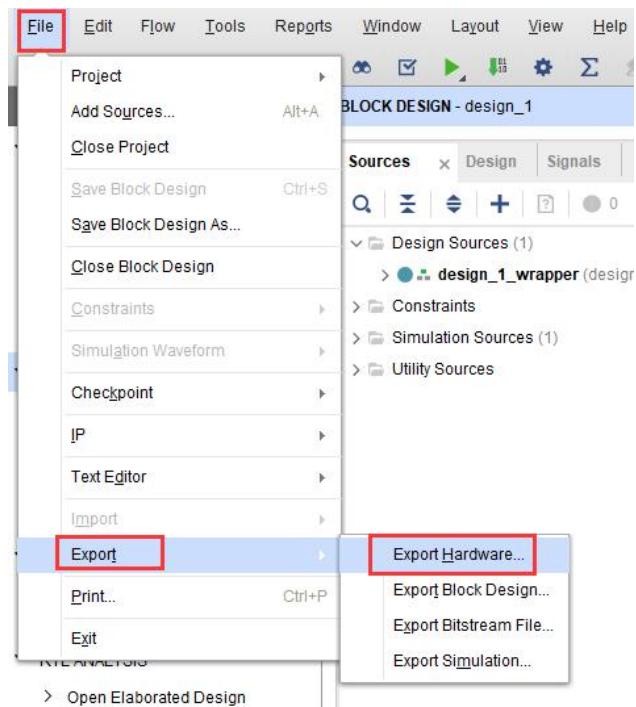
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[0]}]
set_property PACKAGE_PIN AE12 [get_ports {leds_tri_o[0]}]

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

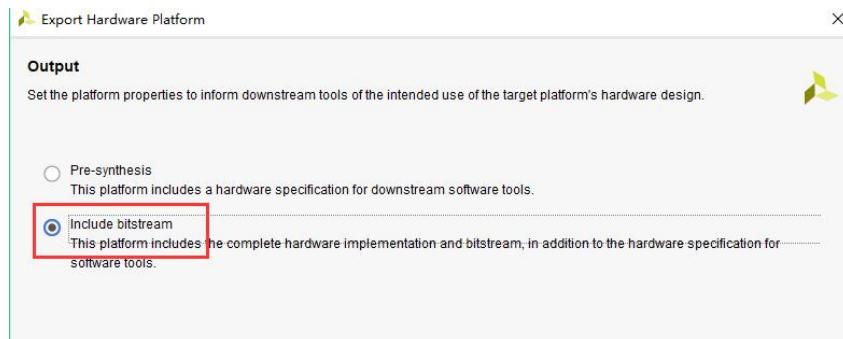
- 4) Generate bit file



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



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



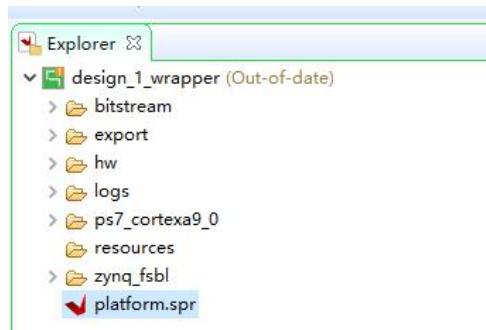
Software Engineer Job Content

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

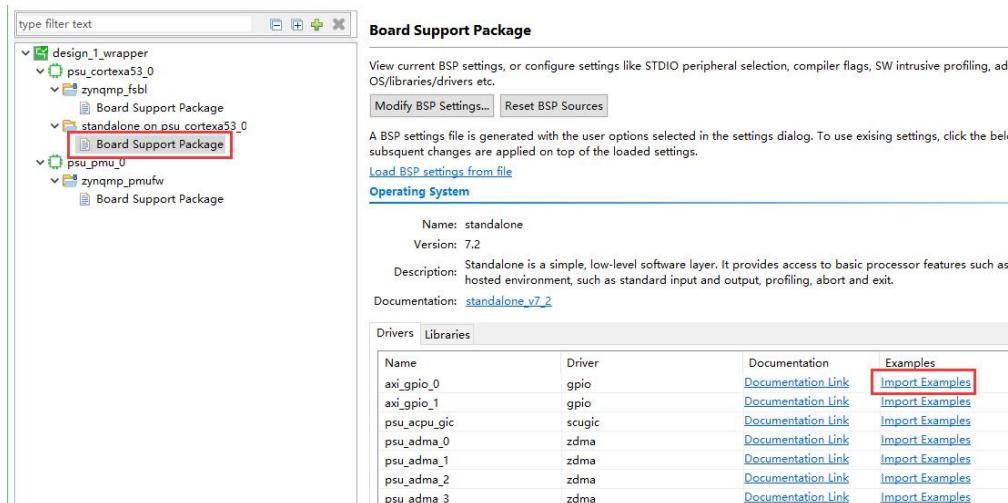
Part 13.4: Vitis Programming

Part 13.4.1: AXI GPIO lights PL LED

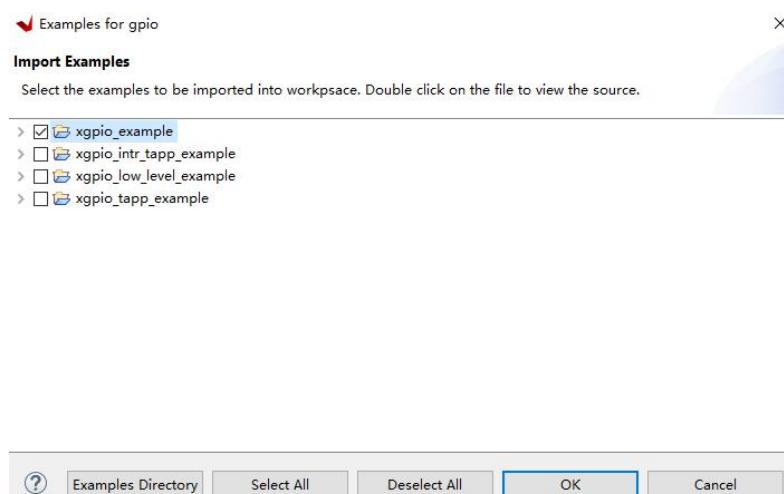
- 1) Create a platform, refer to the chapter "PS Side RTC Interrupt Experiment" for the creation process



- 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:

- Title Bar:** vitis - xgpio_example_1/src/xgpio_example.c - Vitis IDE
- File Menu:** File Edit Xilinx Project Window Help
- Toolbars:** Standard, Design, Debug
- Explorer View:** Shows the project structure:
 - design_1_wrapper (Out-of-date)
 - system_bsp_example_1 [design_1_wrapper]
 - xgpio_example_1 [standalone_domain]
 - Includes
 - jed
 - 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 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 }
```
- Assistant View:** Shows the design_1_wrapper [Platform] and system_bsp_example_1 [System]
- Console View:** Shows the Build Console [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 ⁽³⁾	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 ⁽¹⁾	R/W	0x0	Global Interrupt Enable Register.
0x0128	IP IER ⁽¹⁾	R/W	0x0	IP Interrupt Enable Register (IP IER).
0x0120	IP ISR ⁽¹⁾	R/TOW ⁽²⁾	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.

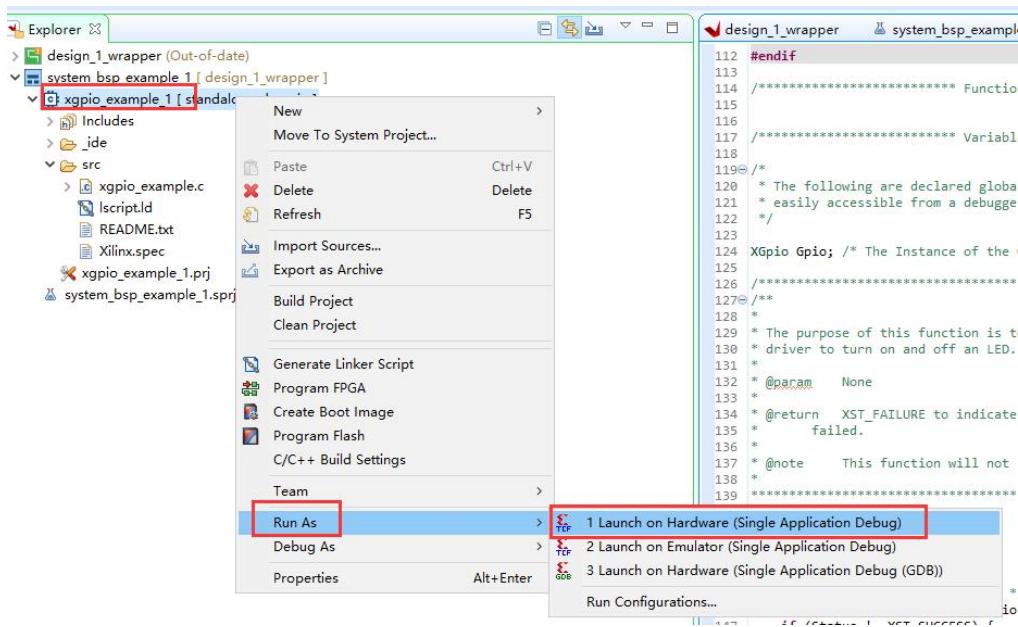
```
@ void XGpio_SetDataDirection(XGpio *InstancePtr, unsigned Channel,
                               u32 DirectionMask)
{
    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
    Xil_AssertVoid((Channel == 1) ||
                  ((Channel ==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

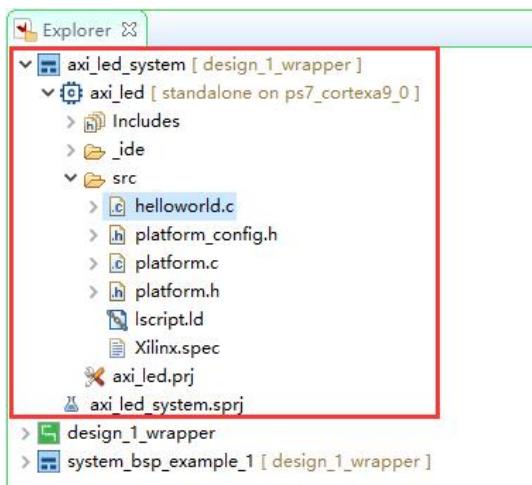
- 1) First compile the APP project, the compilation method has been introduced in the previous example. Although the Vitis can provide some routines, some of them need to be modified by yourself. This simple LED routine is not modified. Try to run it and find that it does not achieve the expected results, 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
#define TRI_OFFSET 0x4            偏移地址

#define LED_DELAY 10000000

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

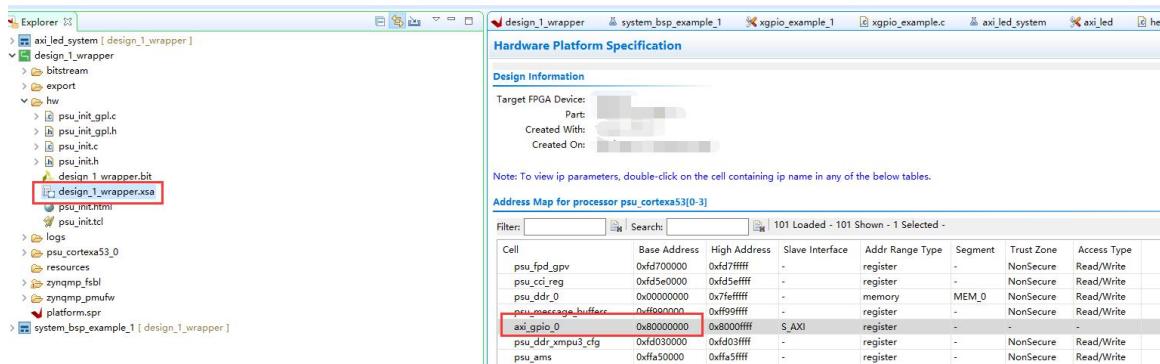
    while (1)
    {
        /* Set the LED to High */
        *(int*)(GPIO_BASEADDR + DATA_OFFSET) = 0xF ;    向数据寄存器
        /* Wait a small amount of time so the LED is visible */
        for (Delay = 0; Delay < LED_DELAY; Delay++);

        /* 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

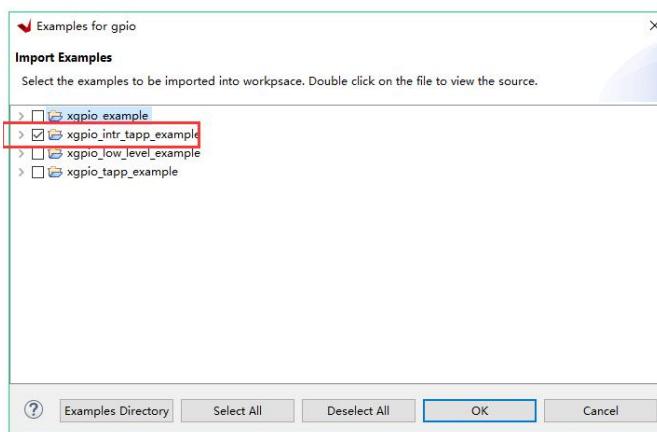
Address Space Offset ⁽³⁾	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register.
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register.

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 0x4121FFFF
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
```

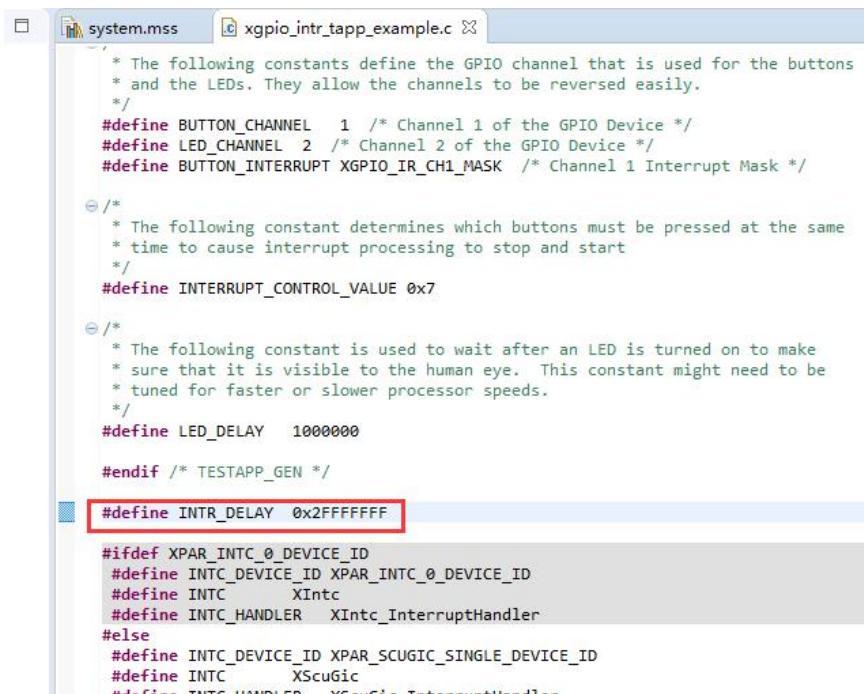
```
997
998
999 /* **** Definitions for Fabric interrupts connected to psu_acpu_gic ****/
1000
1001 /* Definitions for Fabric interrupts connected to psu_acpu_gic */
1002 #define XPAR_FABRIC_AXI_GPIO_1_IP2INTC_IRPT_INTR 121U
1003
```

- 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

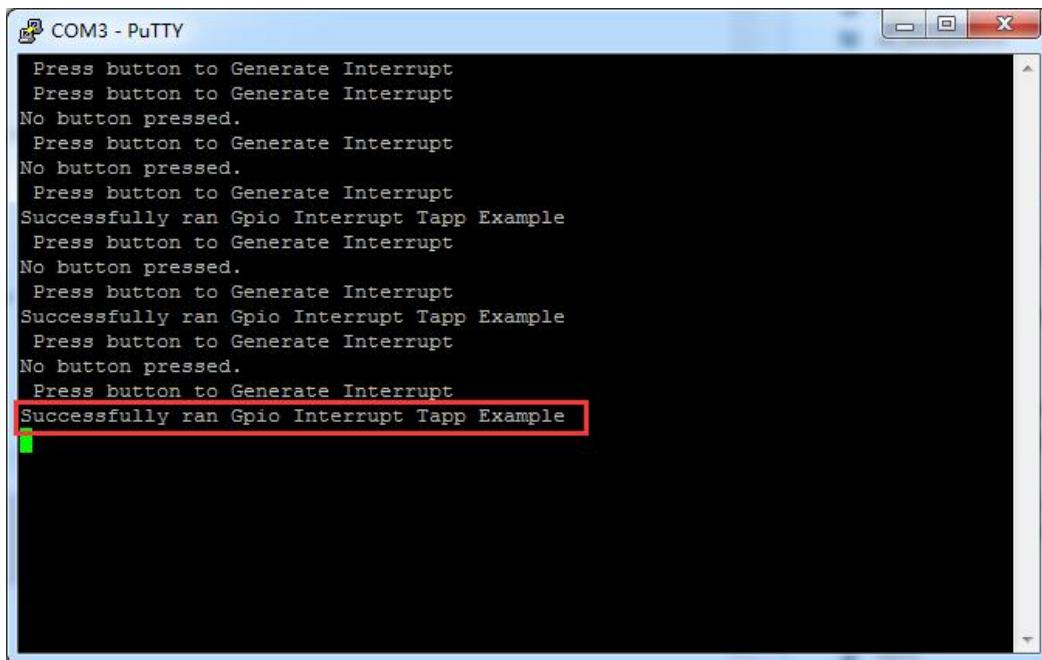
#endif /* TESTAPP_GEN */

#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.

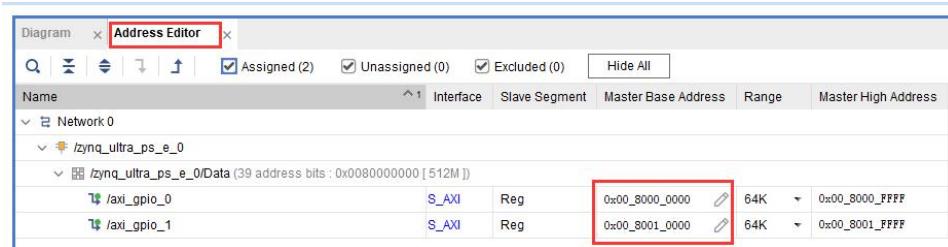
The PL end can send interrupt signals to the PS, which improves the efficiency of PL and PS data interaction. Interrupt processing is needed in applications that require large numbers and low latency.

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

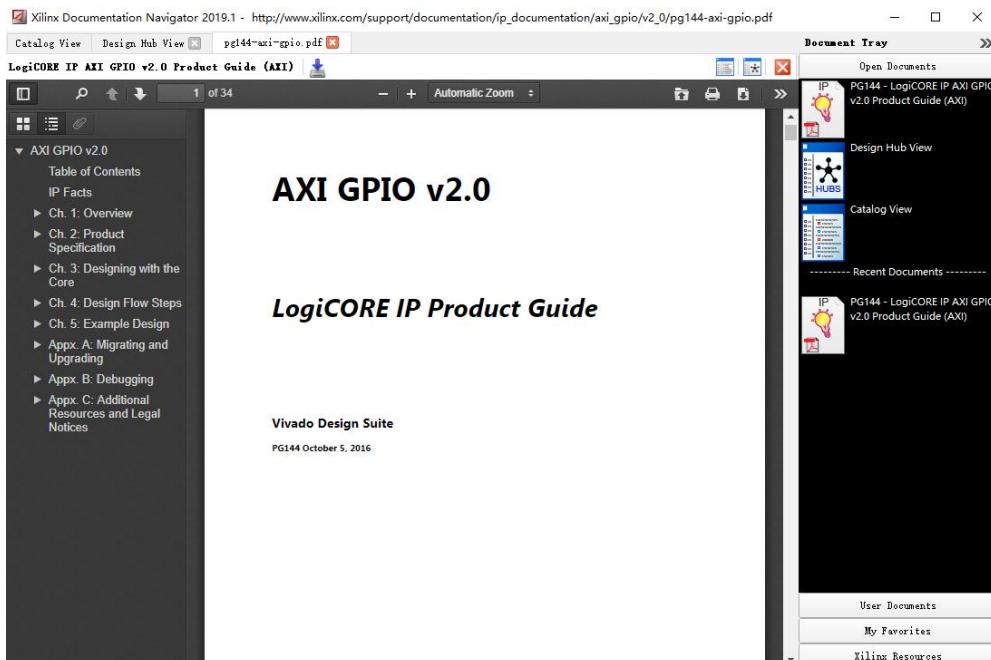
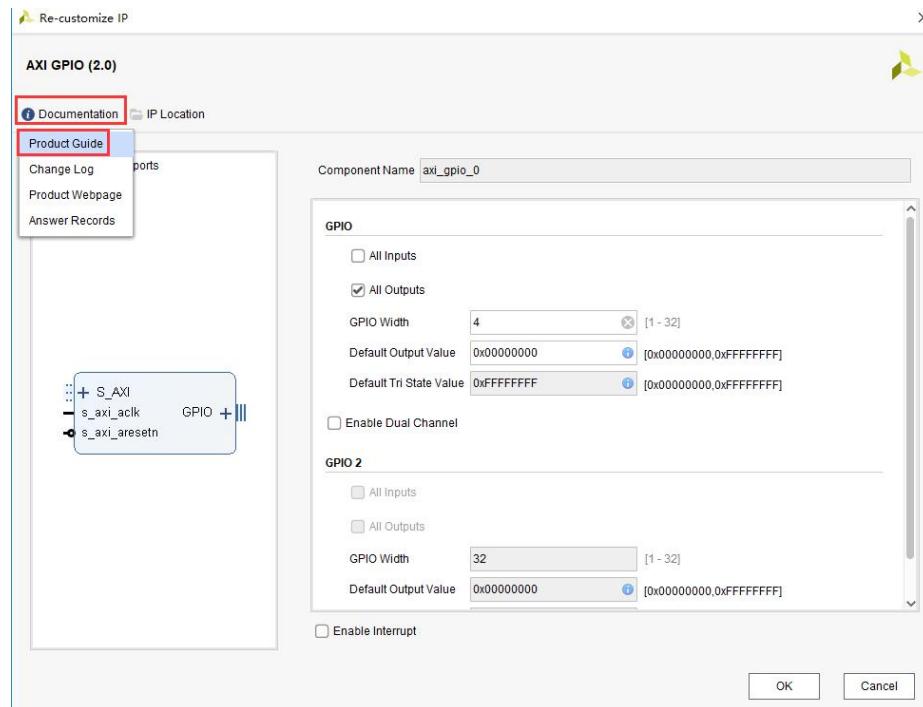
allocated for the AXI peripherals in the Address Editor, where the offset address and space size can be modified



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.

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 ⁽¹⁾	64 MB	0xA000_0000	0xA3FF_FFFF	
M_AXI_HPM0_FPD (HPM0) interface ⁽¹⁾	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	0xFDFF_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	0xFFFFC0_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'. A red box highlights the search input field. To the right of the search bar is a red search button with a magnifying glass icon. 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 result types 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 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.

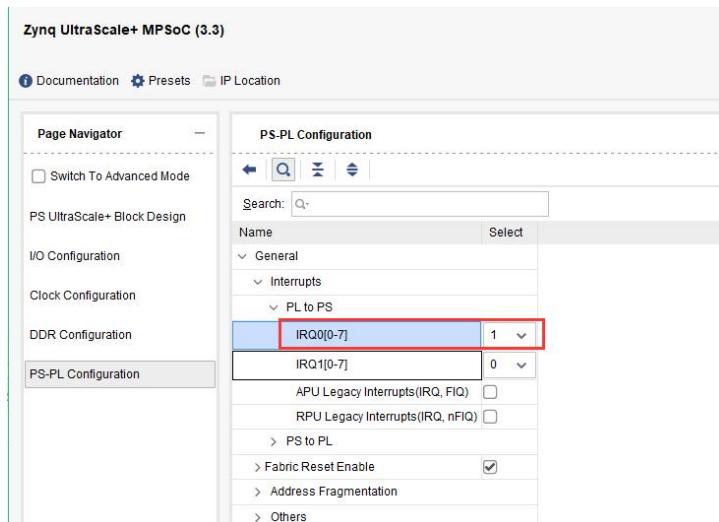
FPGA Engineer Job Content

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

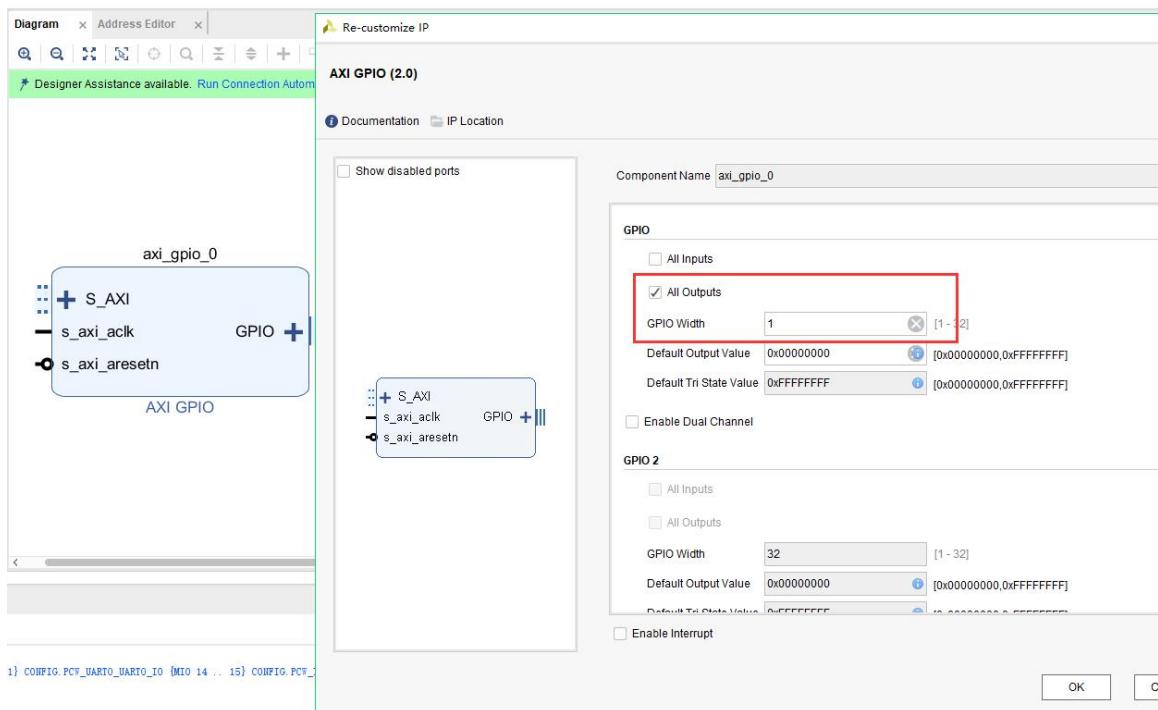
Part 14.1: Create a Hardware Project

The hardware project is based on "ps_ hello", save as a project "rs485_test".

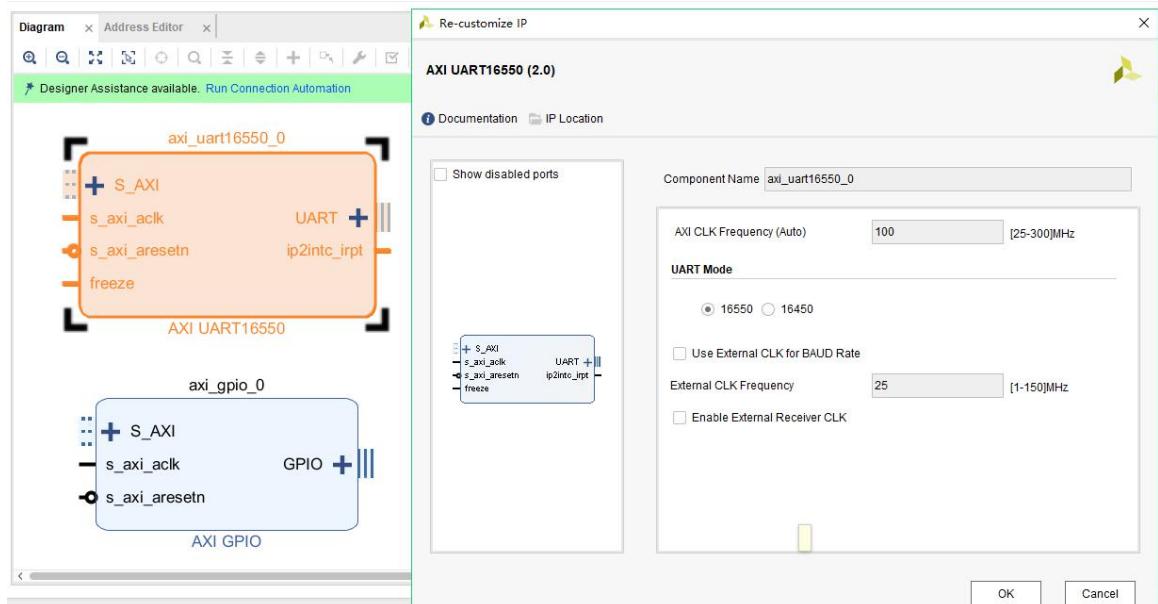
1) Open the PL end interrupt



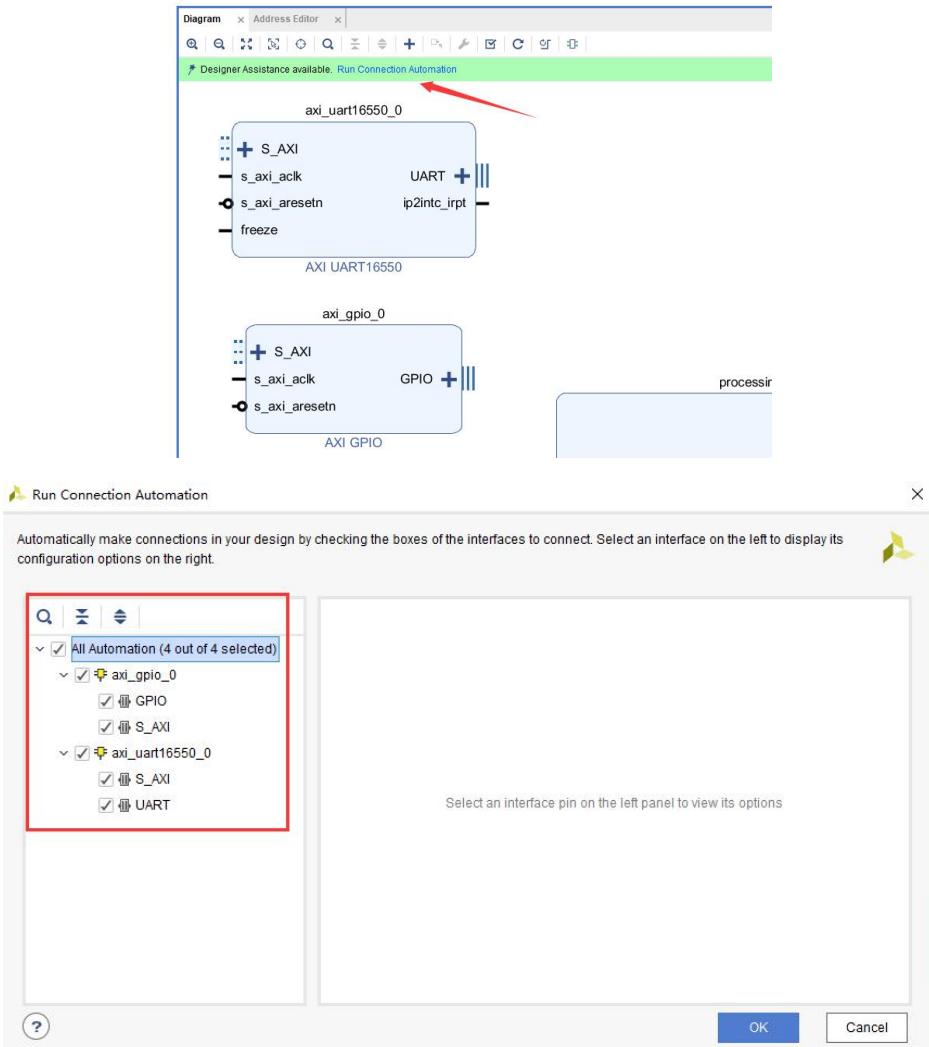
2) Add an AXI GPIO module and configure it as an output with a bit width of 1, used for DE control of the First RS485



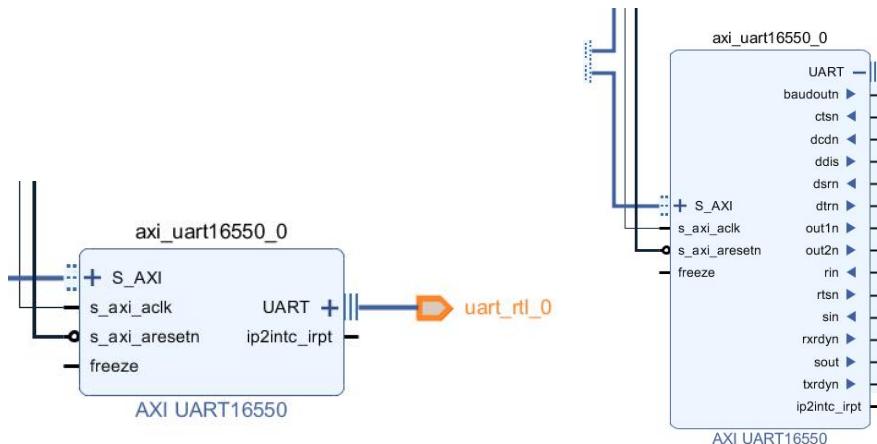
3) Add UART16550 module for the second RS485 data port



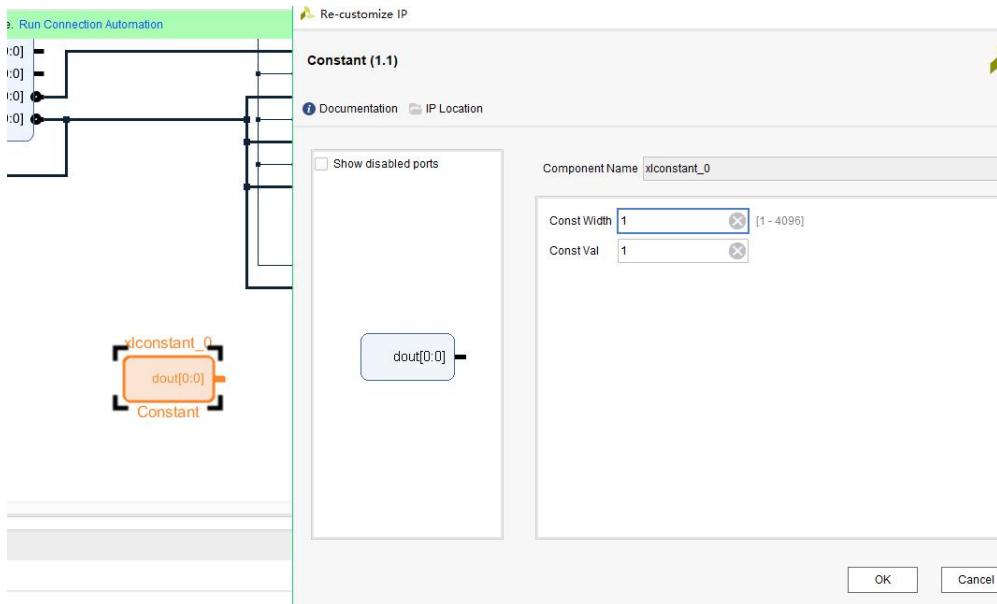
4) Auto Connect



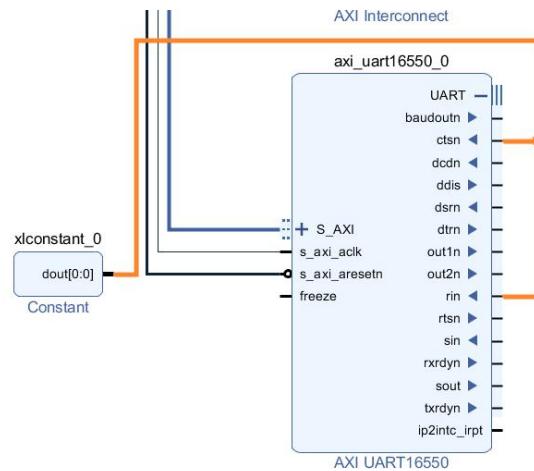
5) Delete the UART pin and expand the UART interface



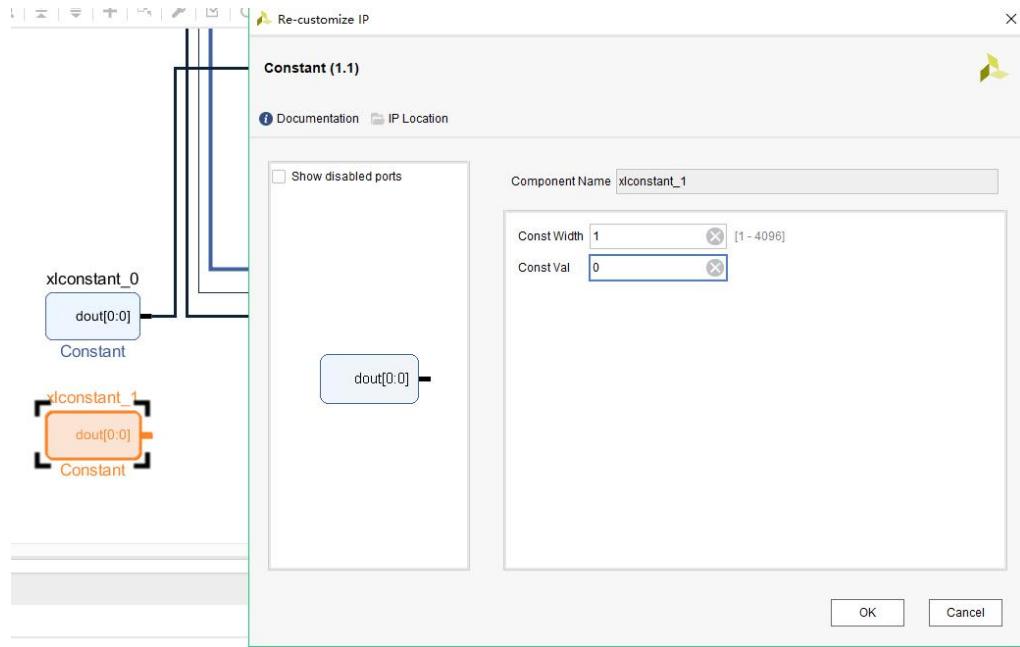
6) Add a constant module, and set the bit width to 1, the value is 1, which is constant



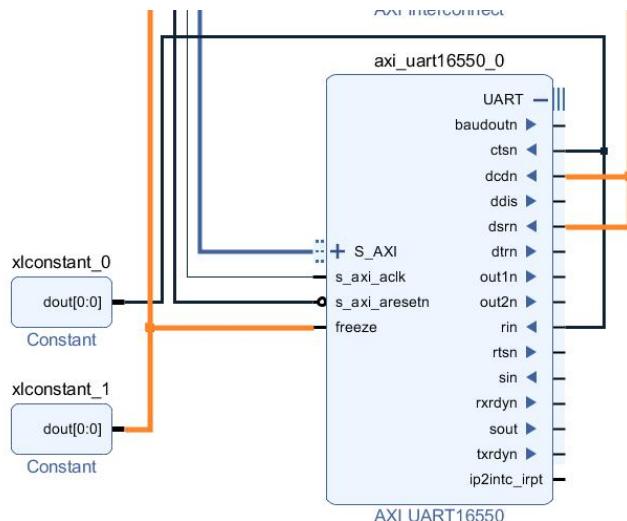
7) Connect ctsn, rin signal



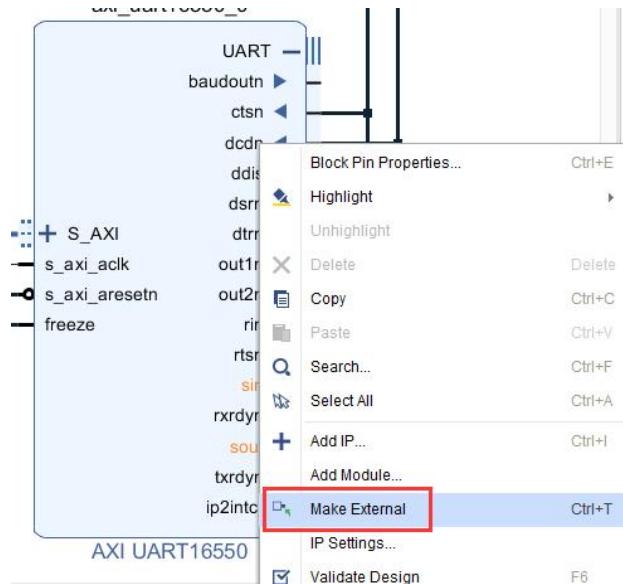
8) Add another constant module and set the value to 0



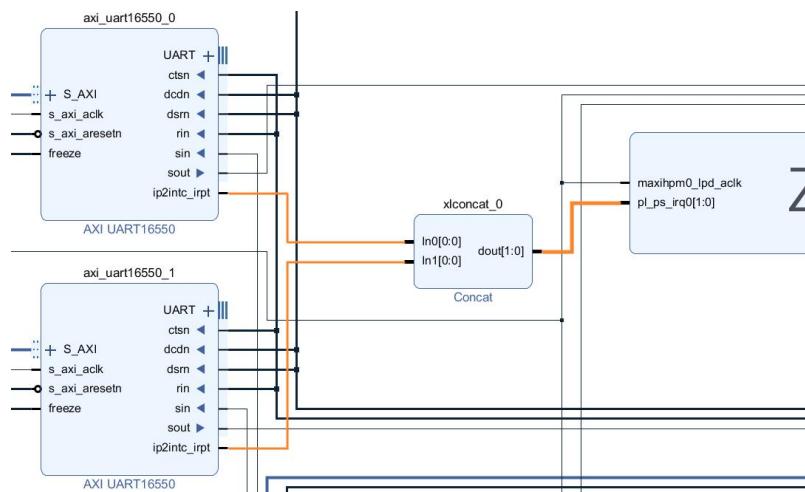
9) Connect "freeze", "dcdn", "dsrn" signals



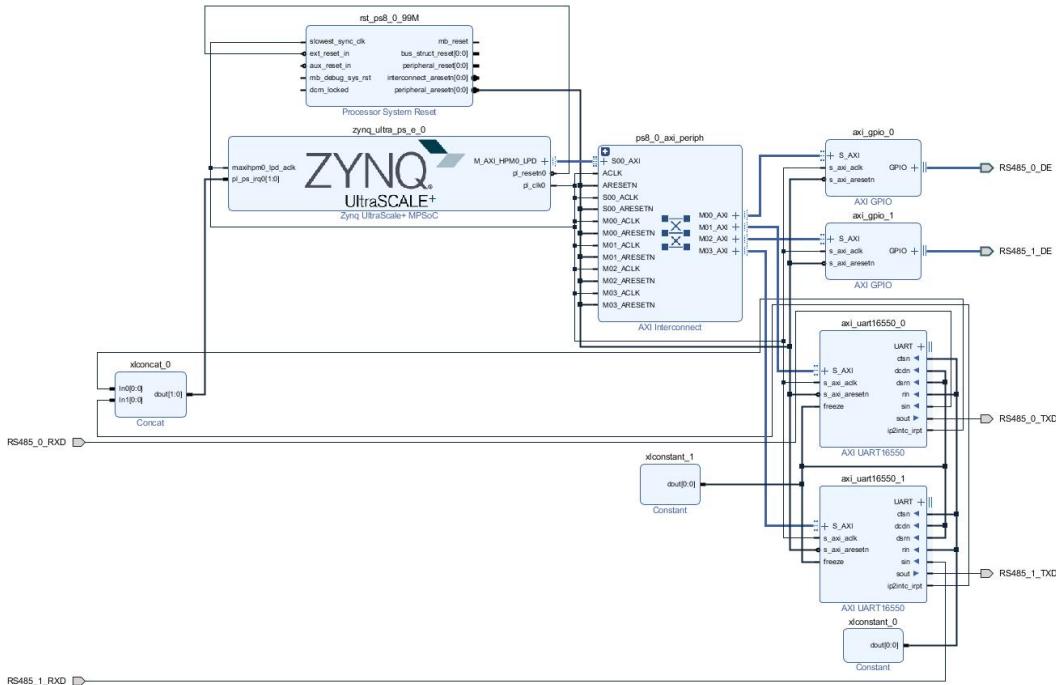
10) Configure the second 485 interface in the same way



- 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



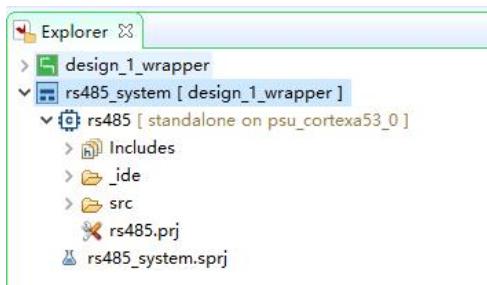
15) Export hardware platform information

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);

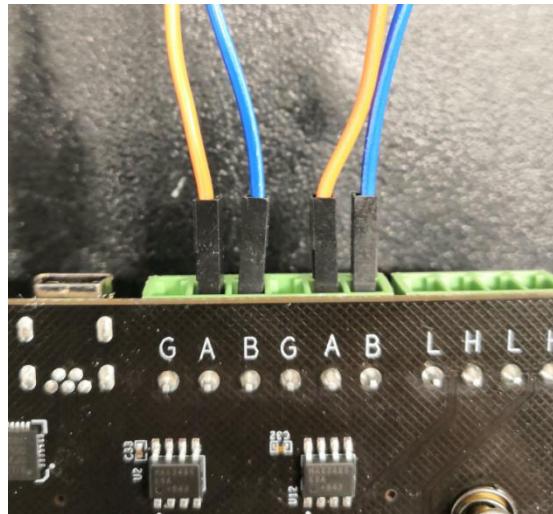
/* wait 1ms */
usleep(1000) ;

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;
}

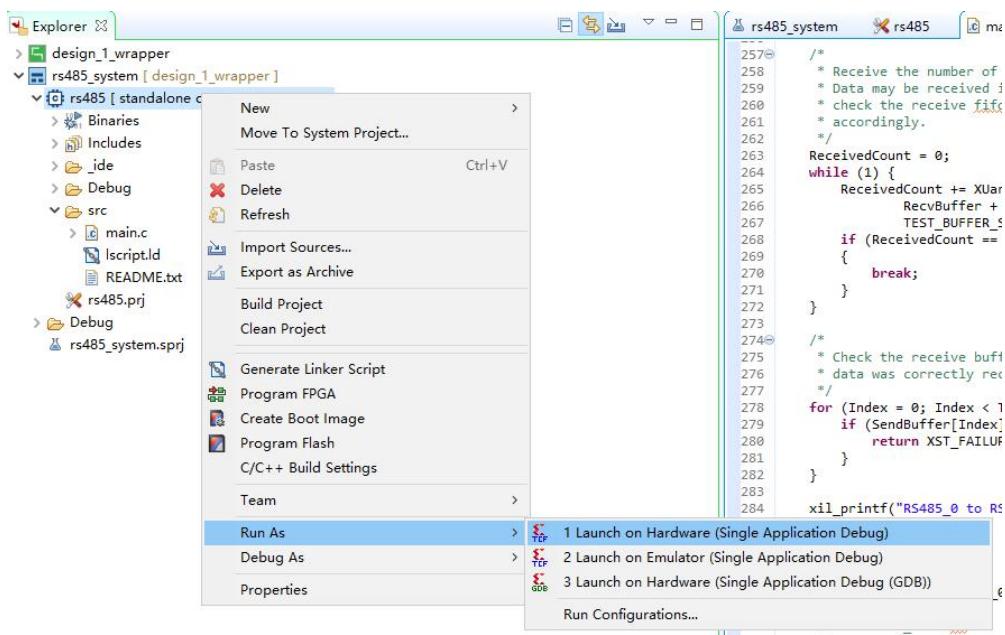
```

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".

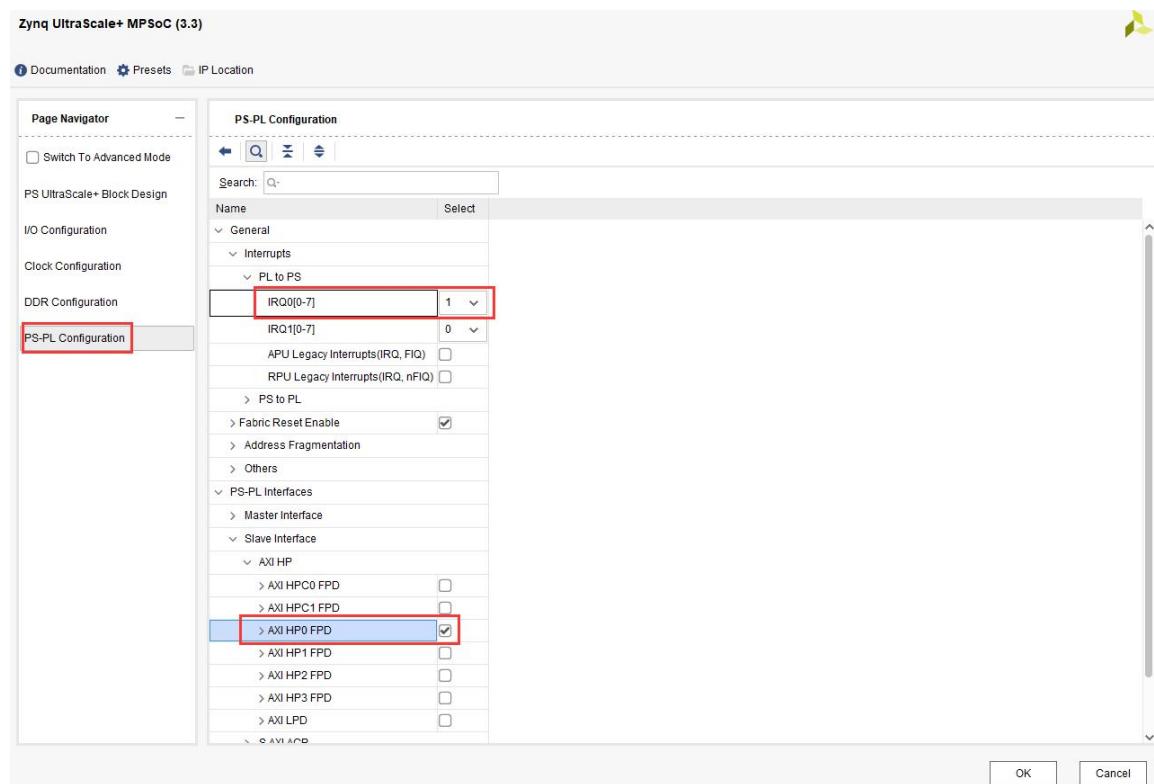
Previously introduced the PS Side use of Ethernet, this chapter introduces PL side use of Ethernet.

FPGA Engineer Job Content

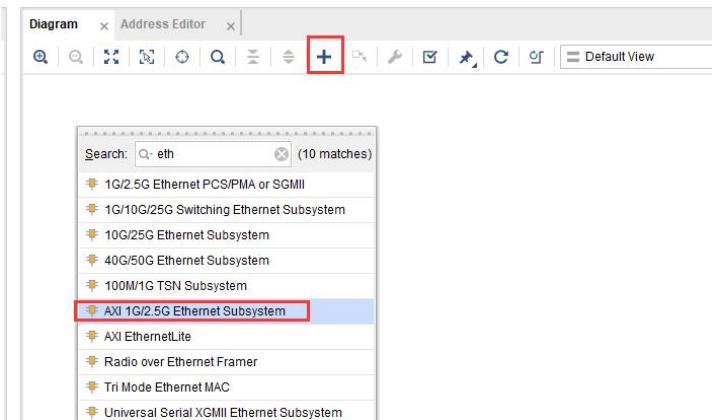
The following is the content that FPGA engineers are responsible for.

Part 15.1: Create a Hardware Project

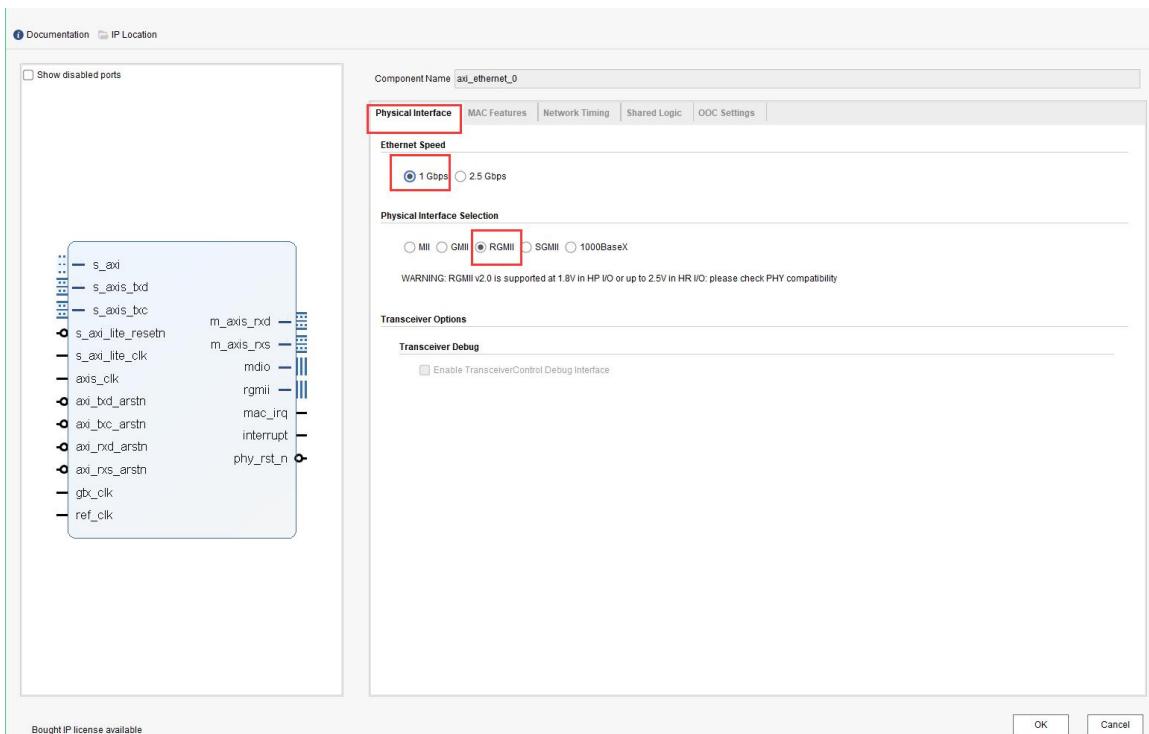
- 1) Based on "ps_hello", configure interrupts and AXI HP0 FPD



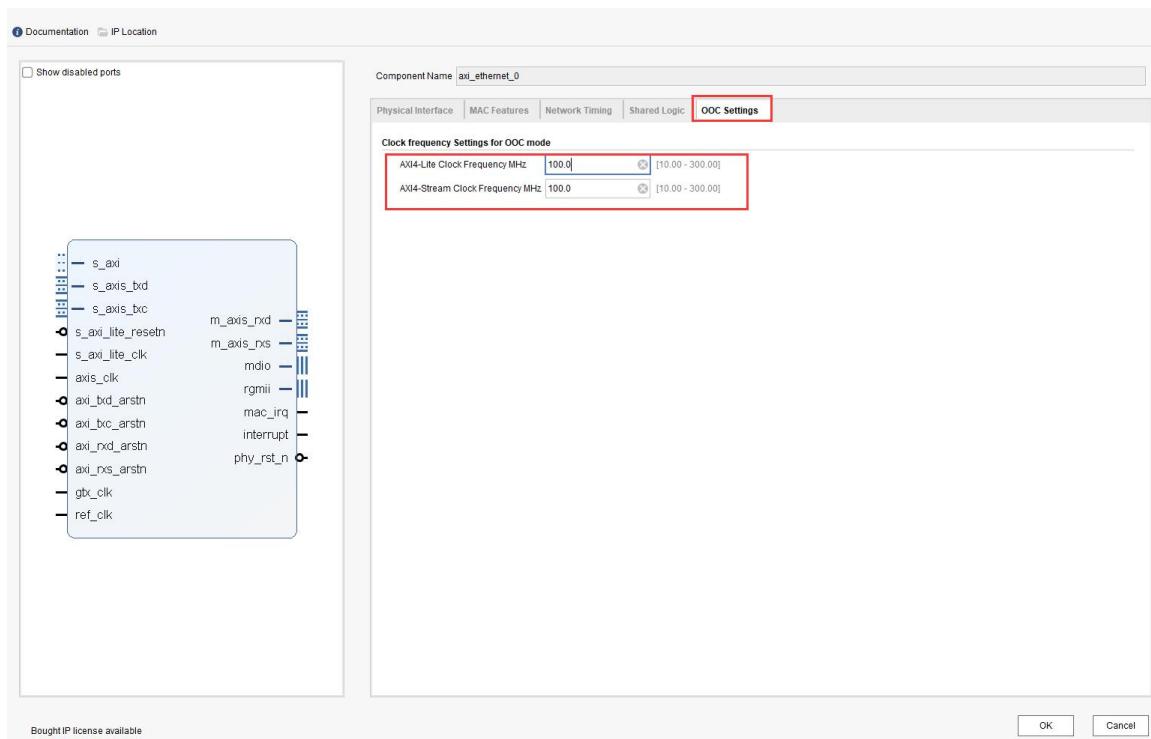
- 2) Add AXI 1G/2.5G Ethernet Subsystem module



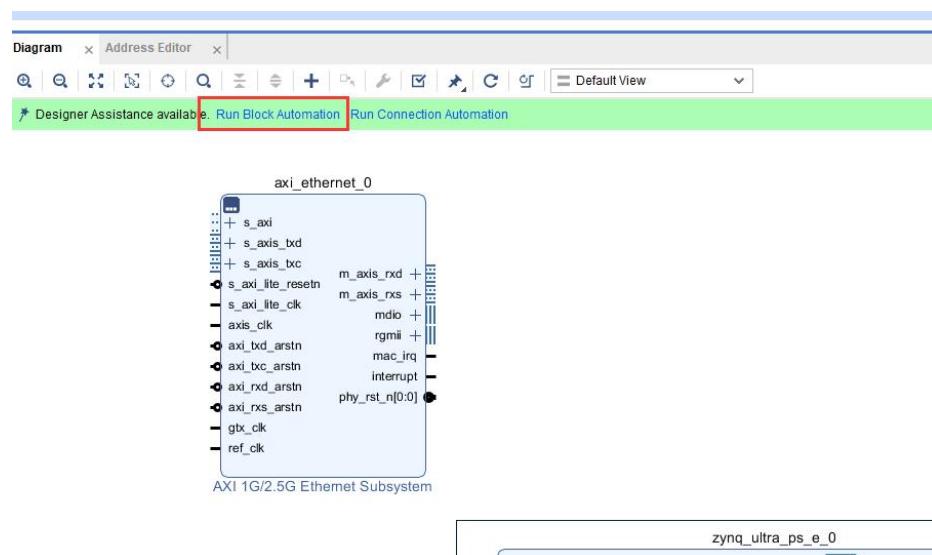
3) The configuration speed is 1Gbps, and the PHY interface is RGMII

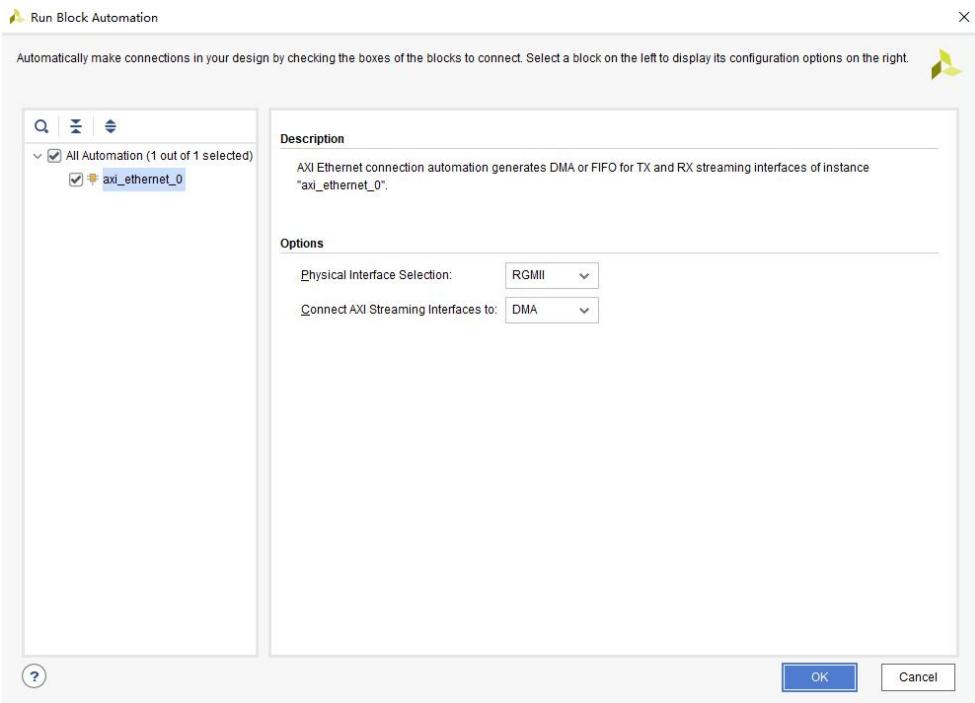


4) The clock defaults to 100MHz, other configurations keep the default, click OK

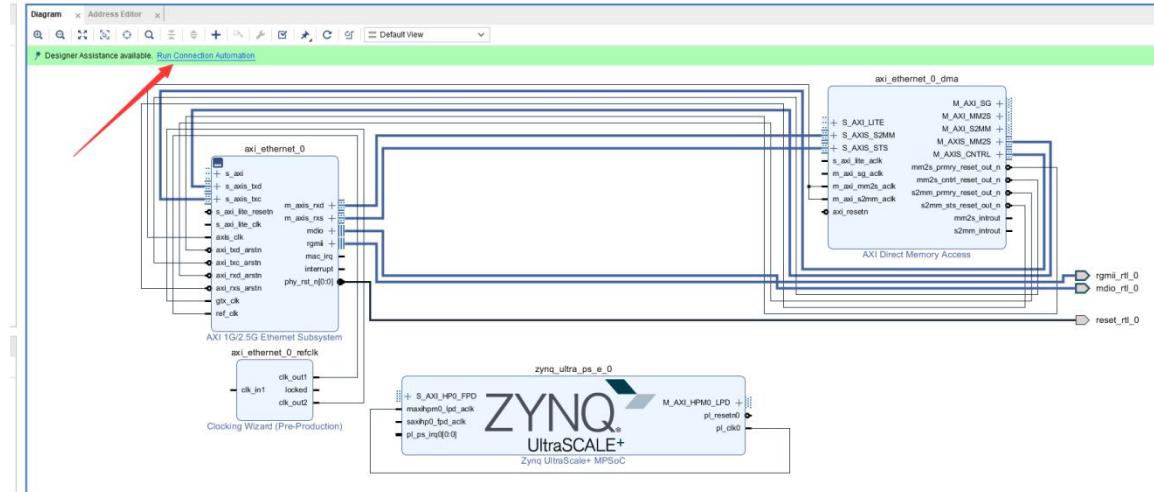


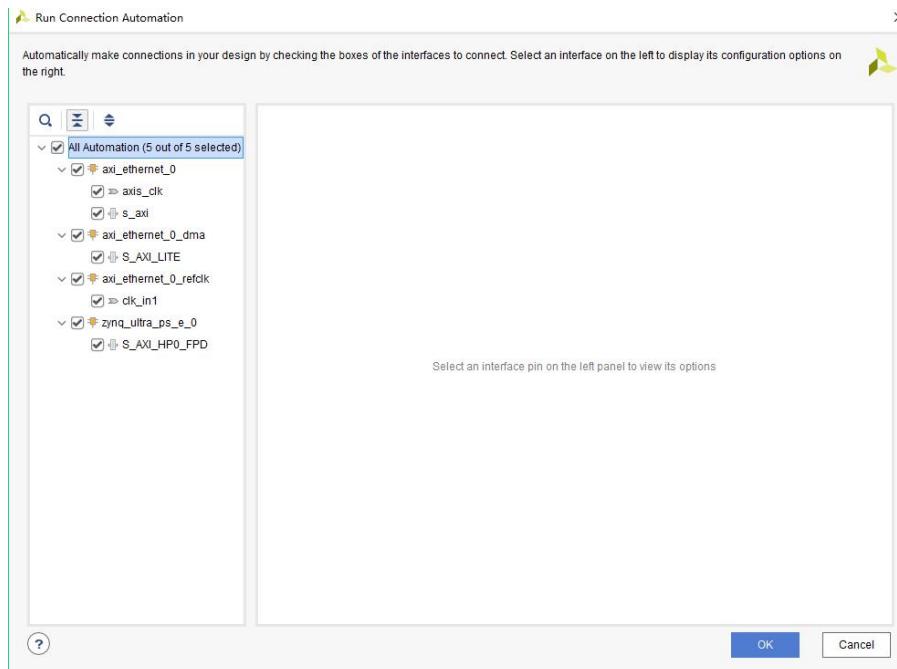
5) Click Run Block Automation



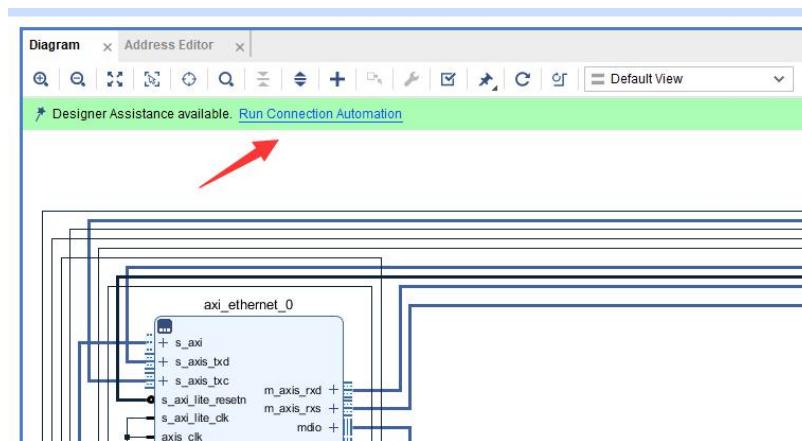


6) Click Run Connectino Automation to automatically connect

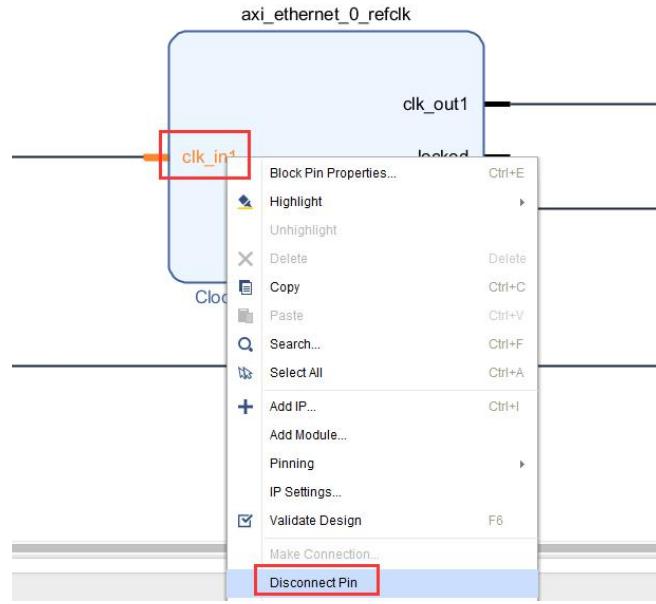




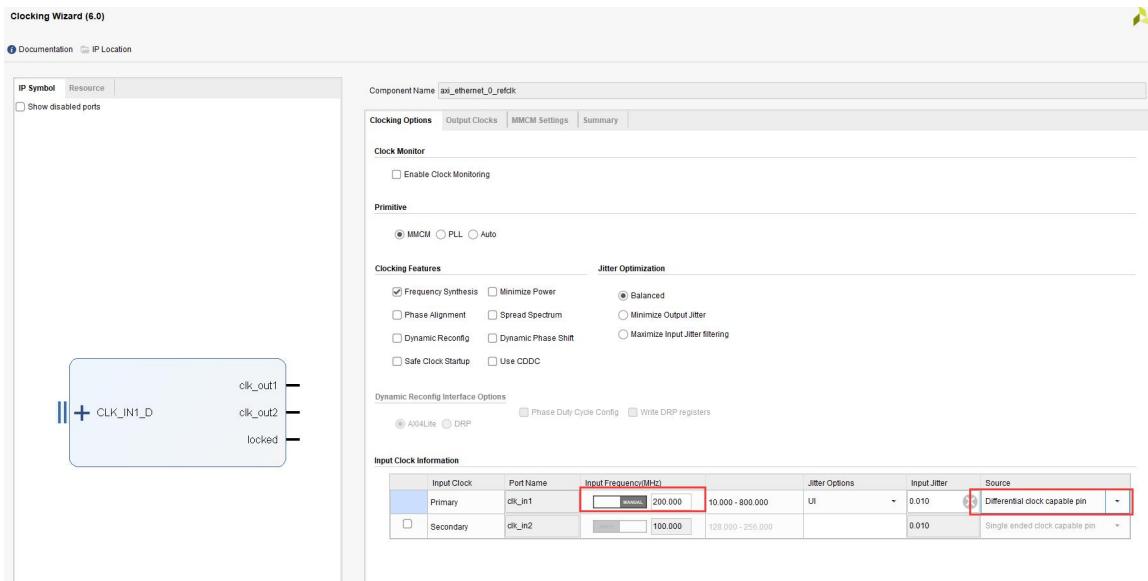
7) Continue to connect automatically



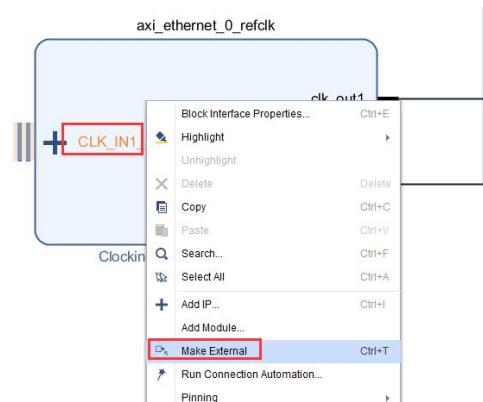
8) Select the input clock of PII, Disconnect Pin



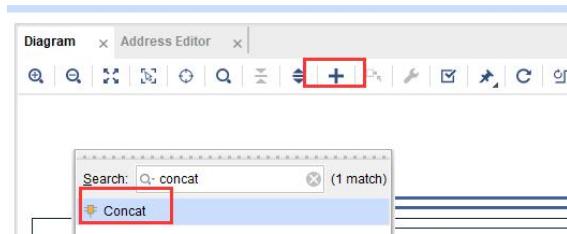
Click to open the PLL, configure the input clock to 200MHz, and set it as a differential input



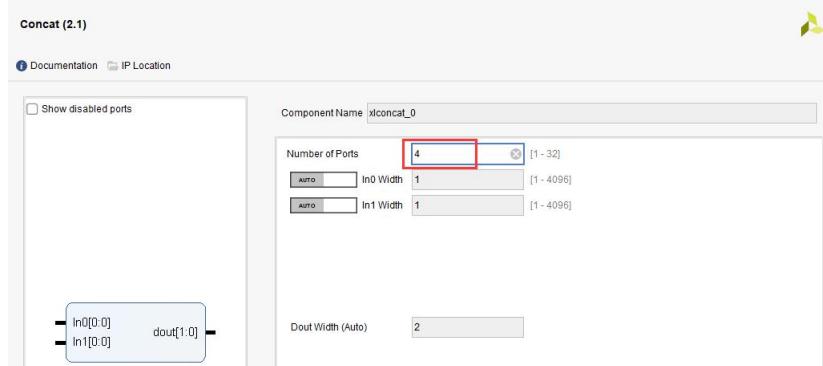
9) Export pin



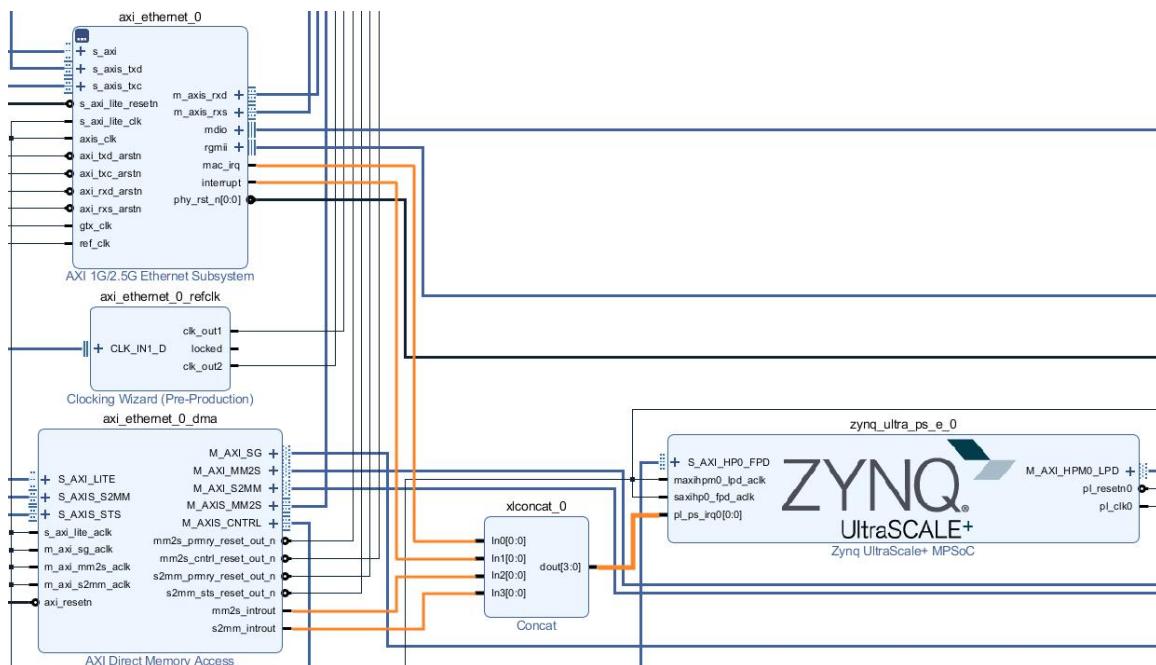
10)Add concat module



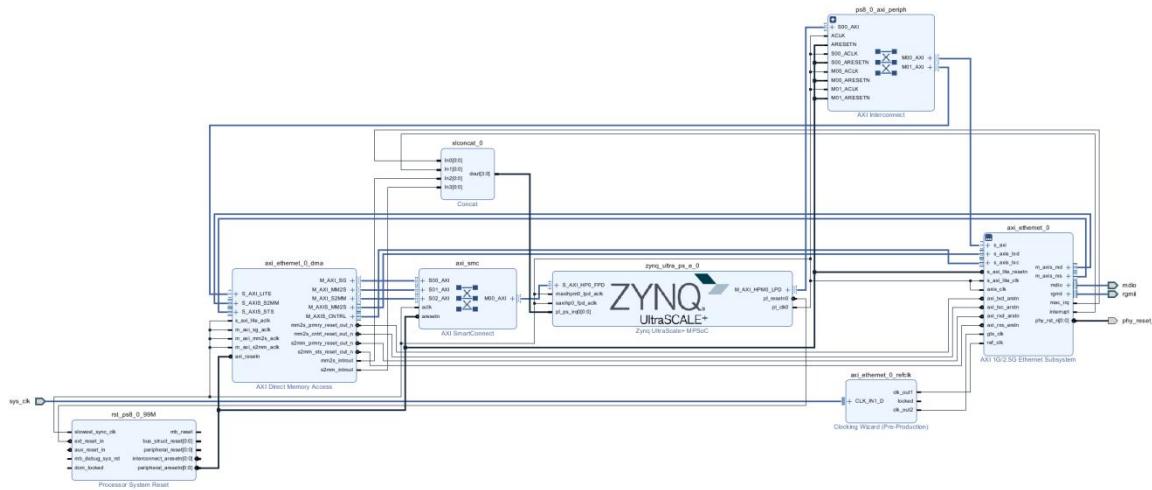
Configured to 4



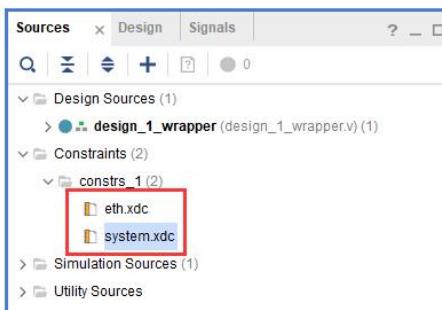
11)Connection interruption signal



12)Modify the pin name



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



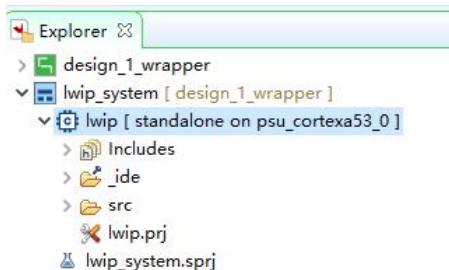
Software Engineer Job Content

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

Part 15.2: Vitis Program Development

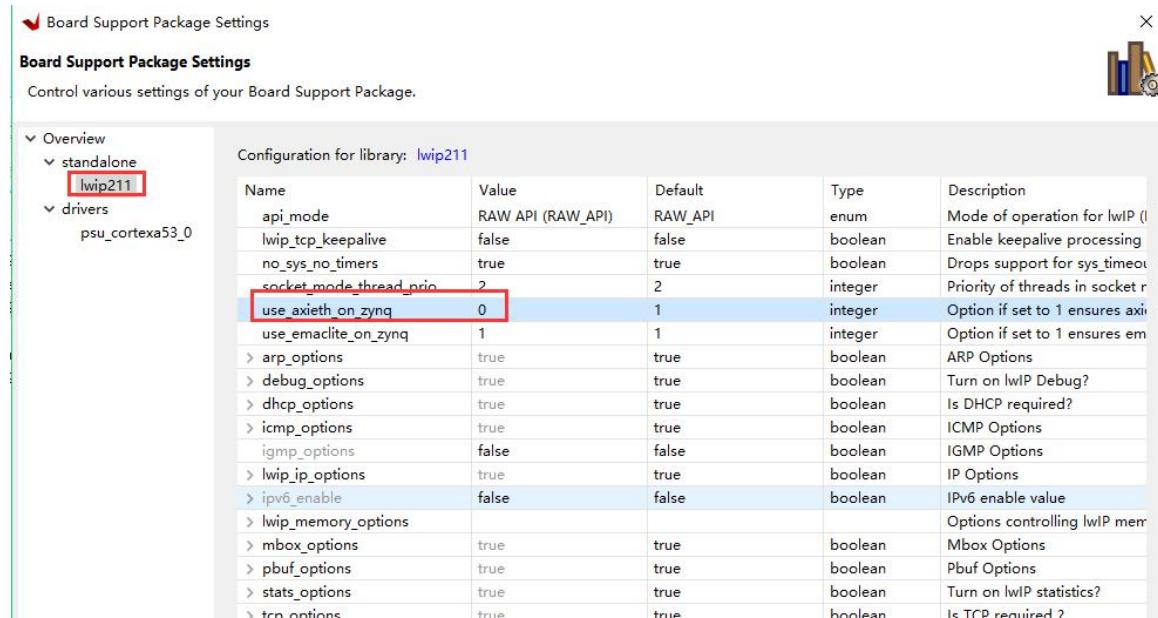
Part 15.2.1: PL Side Ethernet test

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.

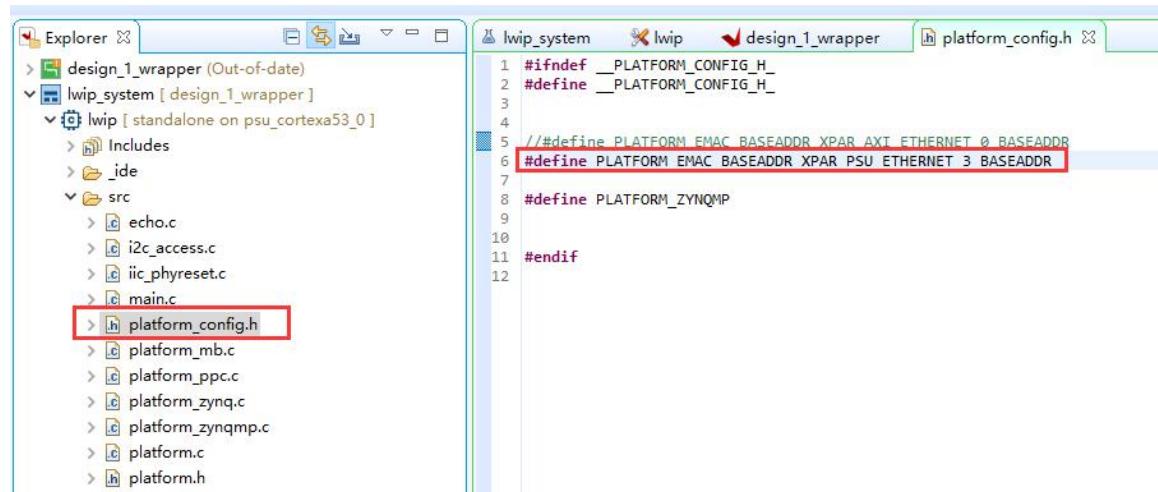


Part 15.2.2: PS Side Ethernet Test

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_on_zynq` 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.

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

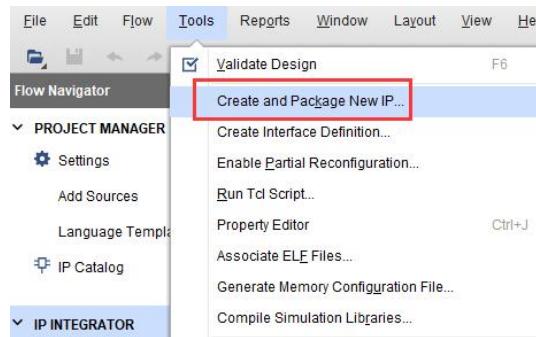
$$\text{PWM Duty Cycle} = 1 - \frac{\text{duty}+1}{2^N}$$

Part 16.2: Building a Vivado project

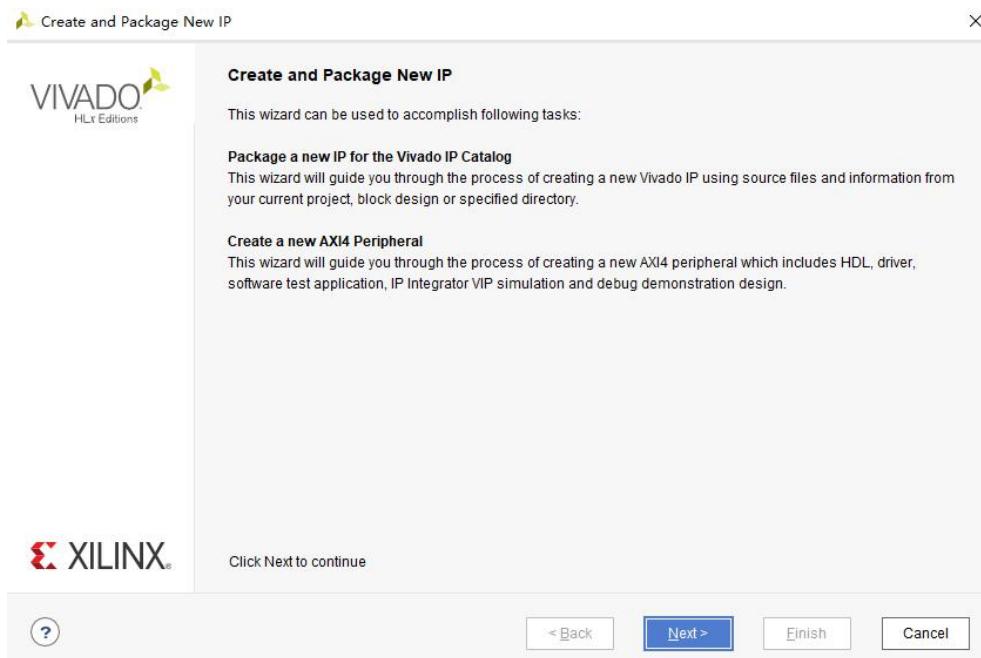
Save the project "ps_hello" as a project named "custom_pwm_ip"

Part 16.2.1: Create a custom IP

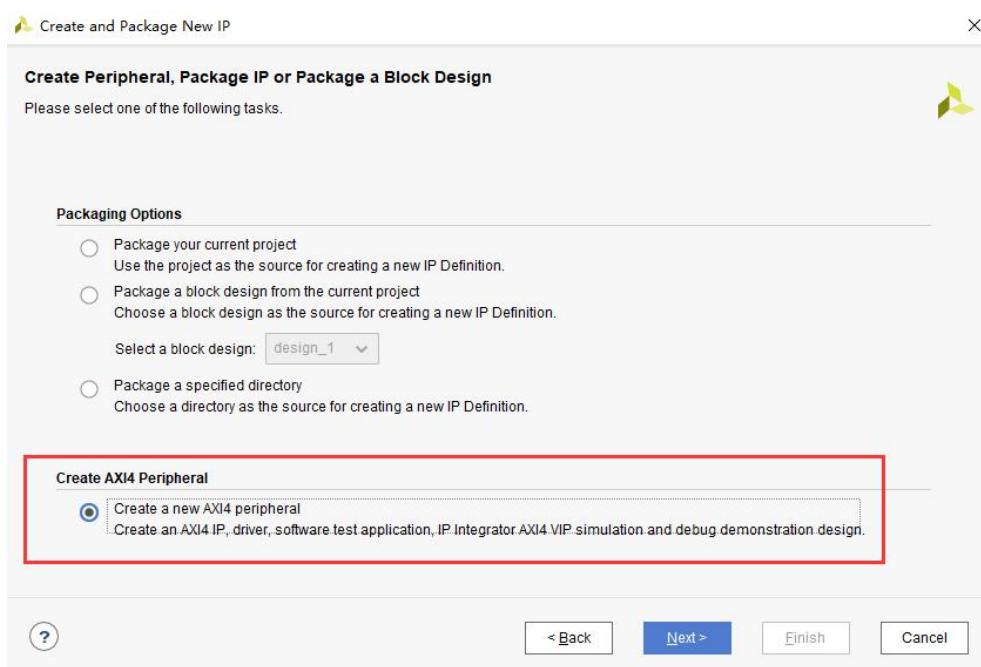
- 1) Click on the menu "Tools->Create and Package IP..."



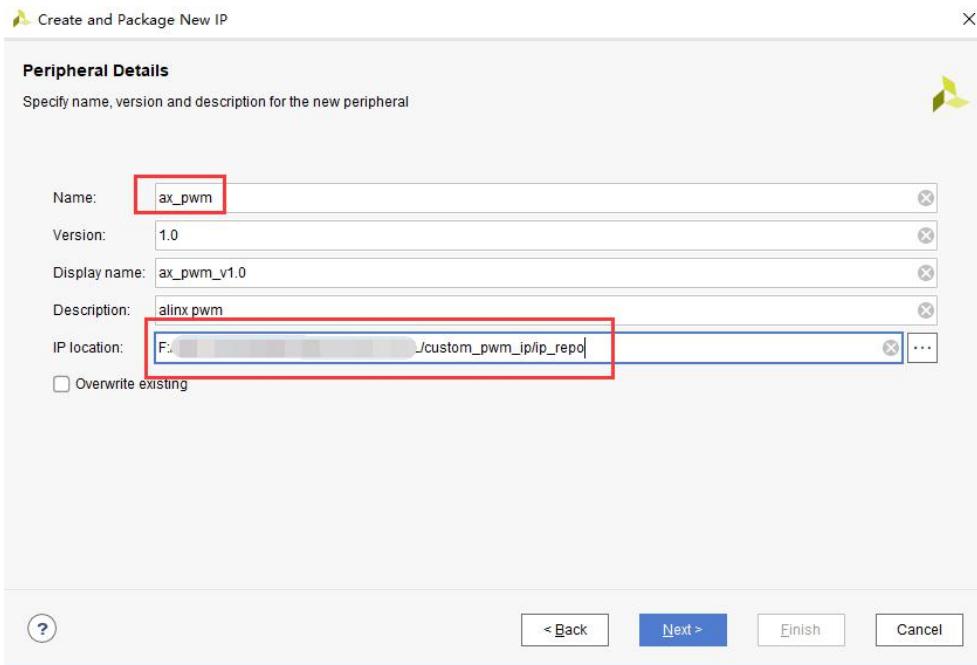
2) Select "Next"



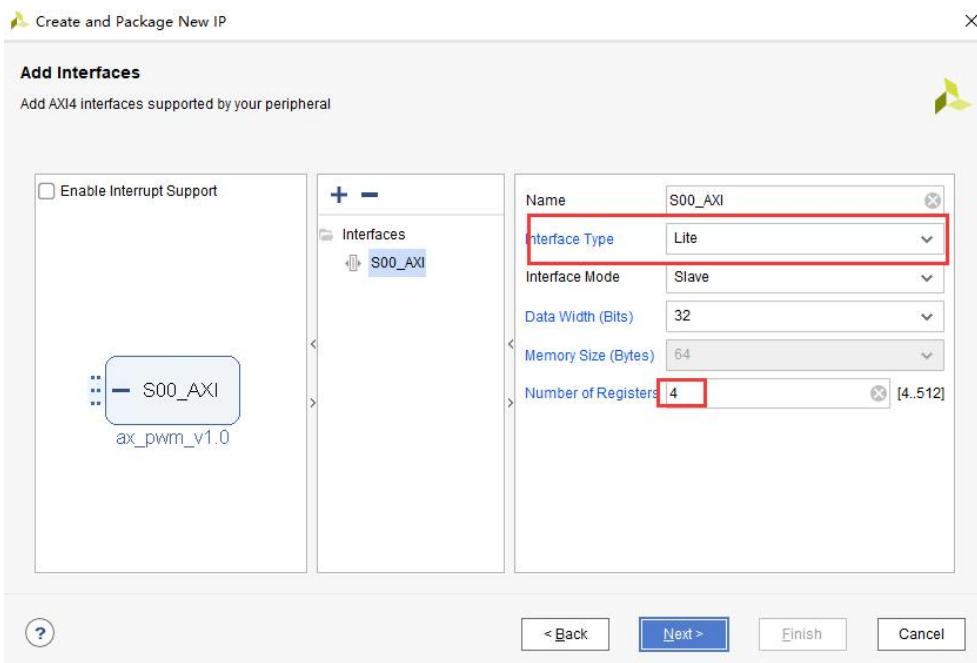
3) Choose to create a new AXI4 device



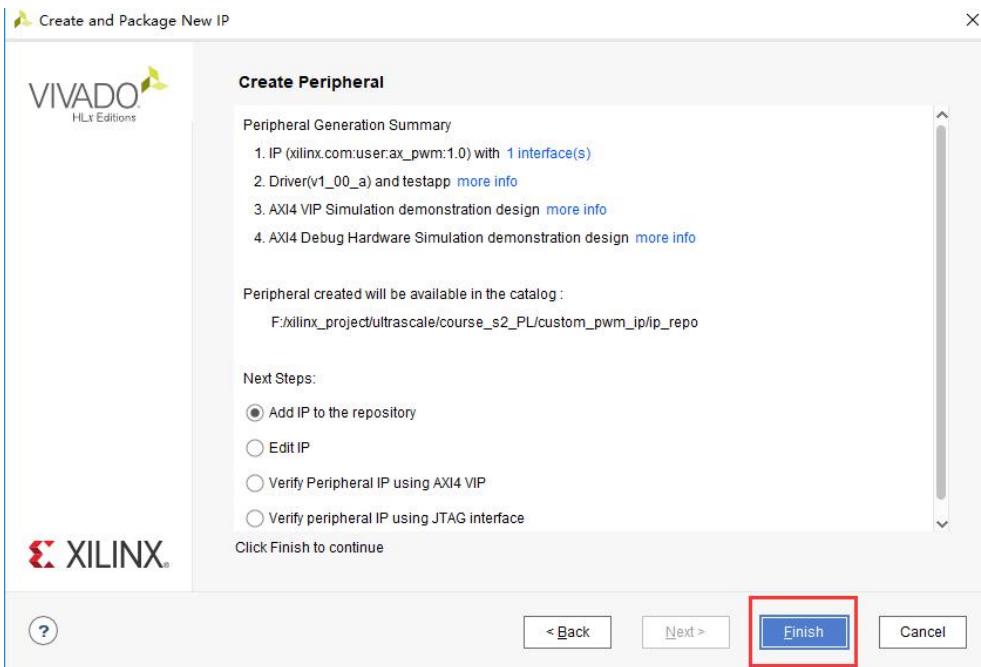
- 4) Fill in the name "ax_pwm", describe "alinx pwm", and then select a suitable location for IP



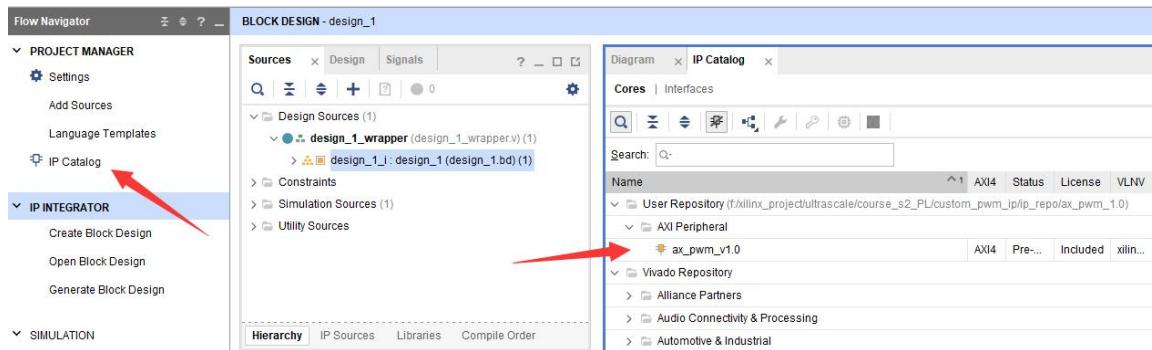
- 5) The following parameters can specify the interface type, the number of registers, etc., do not need to be modified here, use the AXI Lite Slave interface, 4 registers.



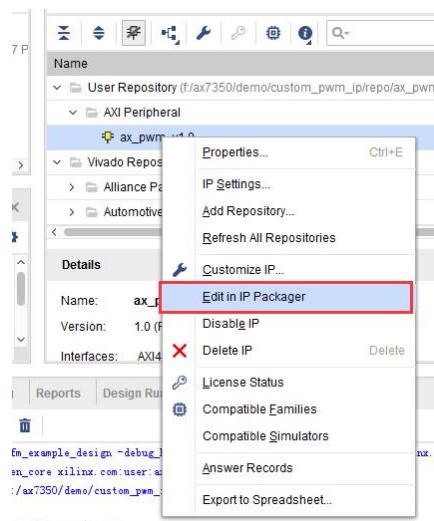
- 6) Click "Finish" to complete the creation of the IP.



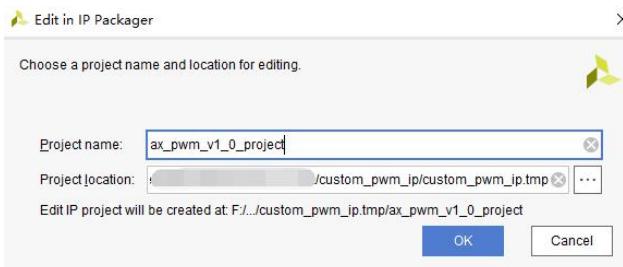
7) You can see the IP just created in "IP Catalog"



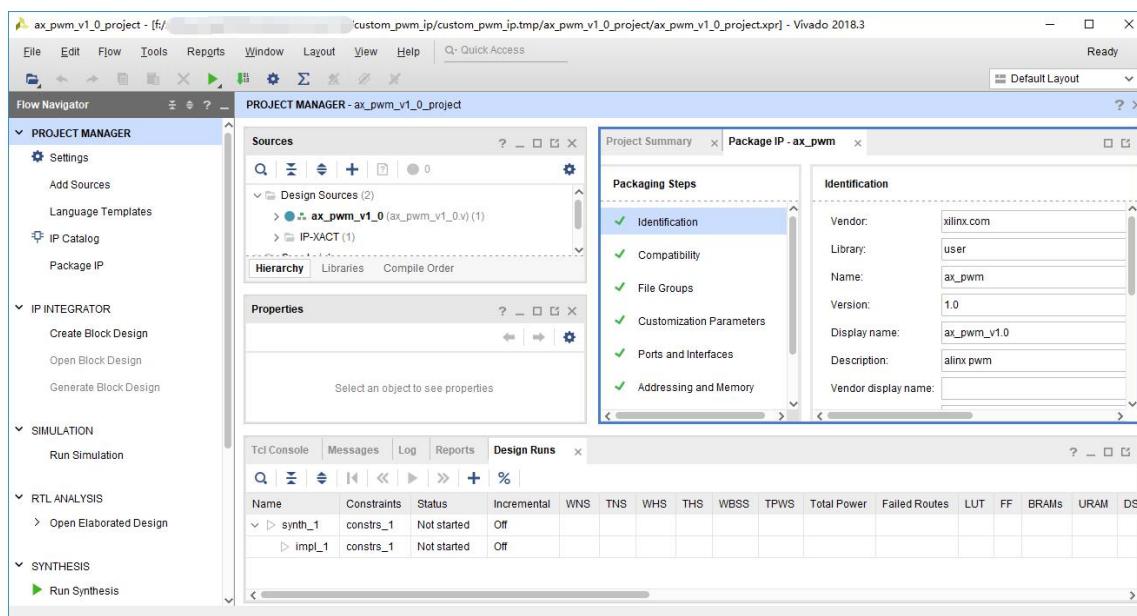
8) At this time, the IP only has a simple register read and write function. We need to modify the IP, select the IP, and right click "Edit in IP Package".



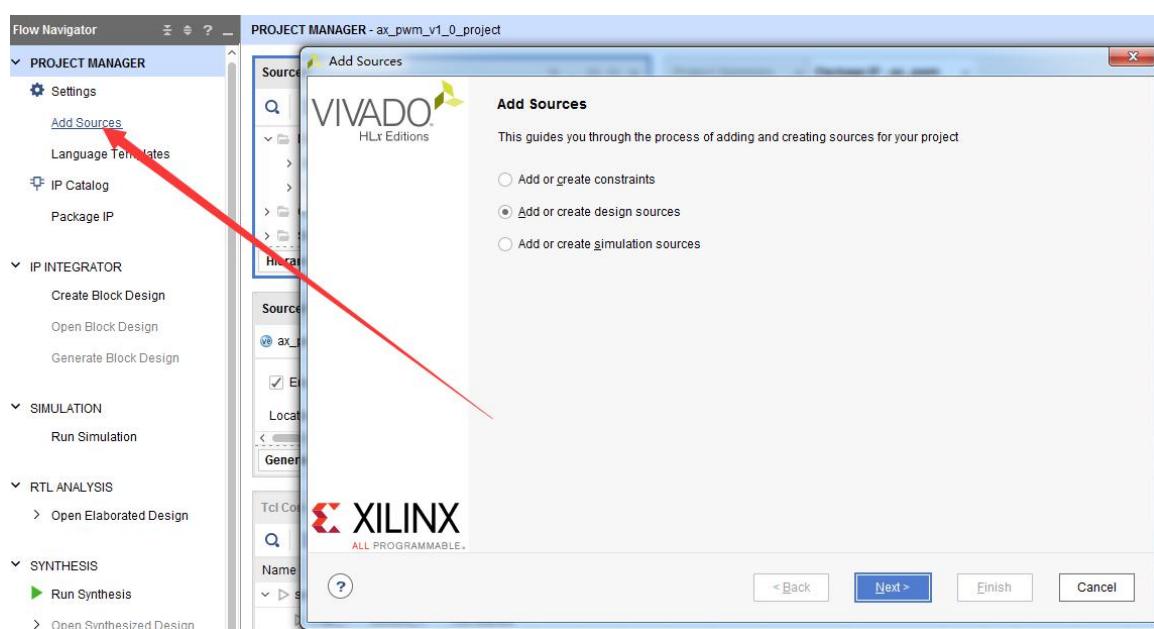
- 9) This is a pop-up dialog box where you can fill in the project name and path. By default, click "OK"



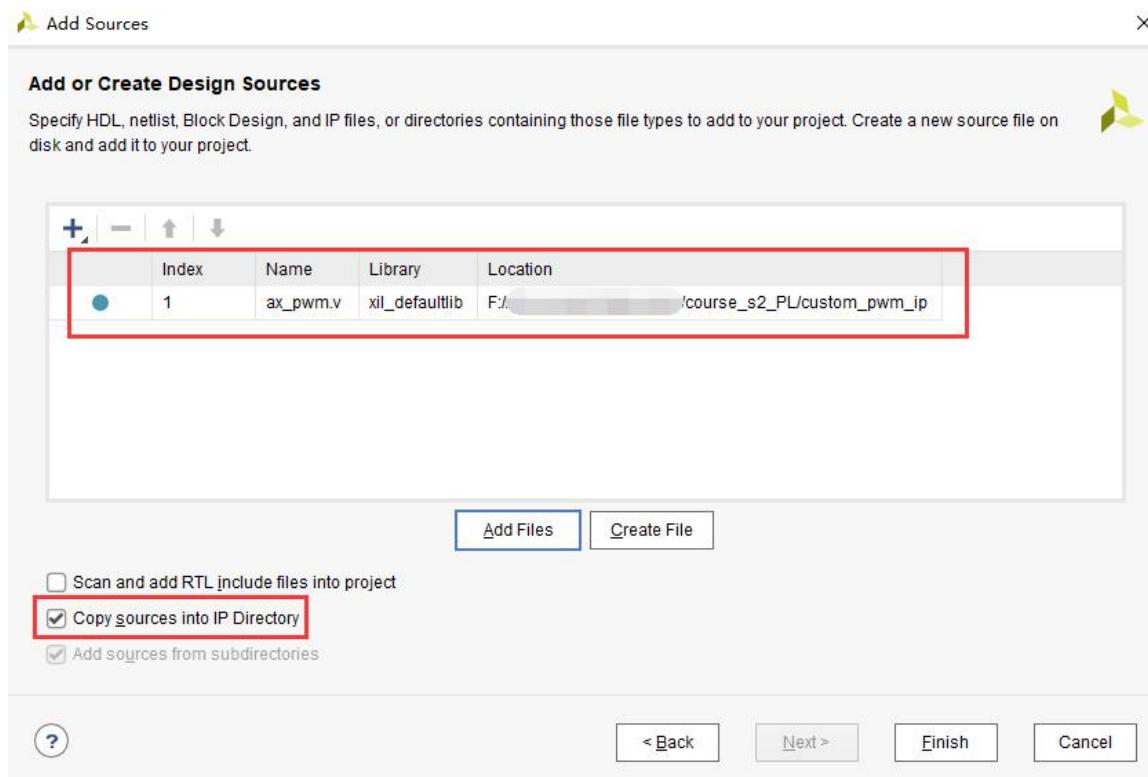
10)Vivado opens a new project



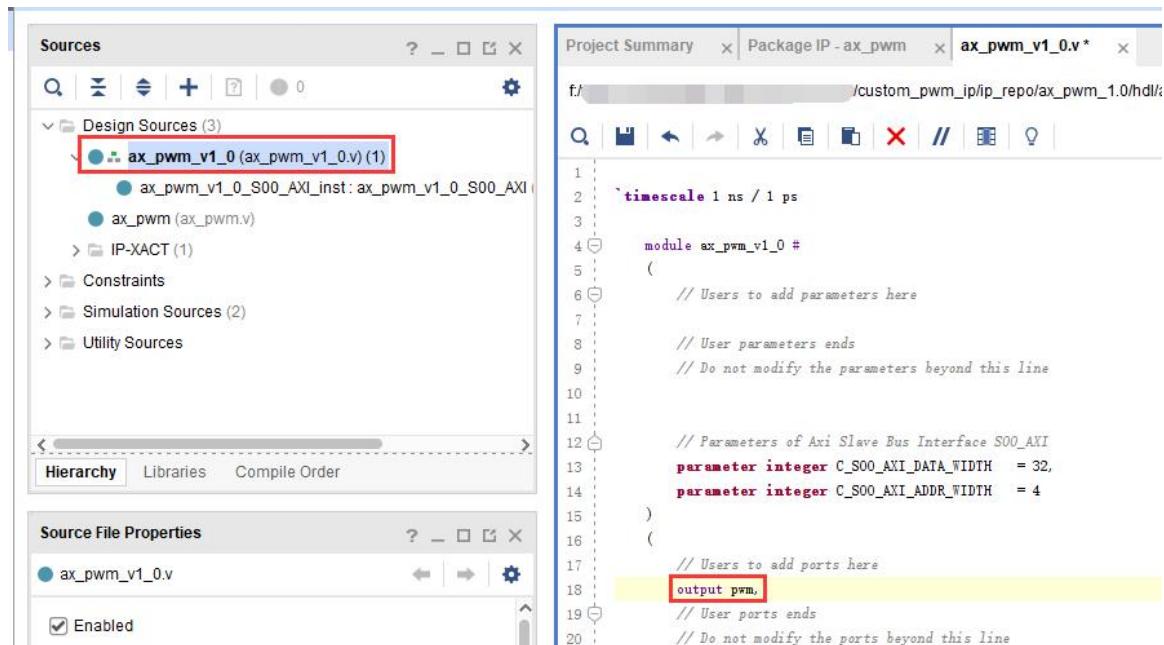
11)Add the core code of the PWM function



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_S00_AXI_DATA_WIDTH),
49     .C_S_AXI_ADDR_WIDTH(C_S00_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.

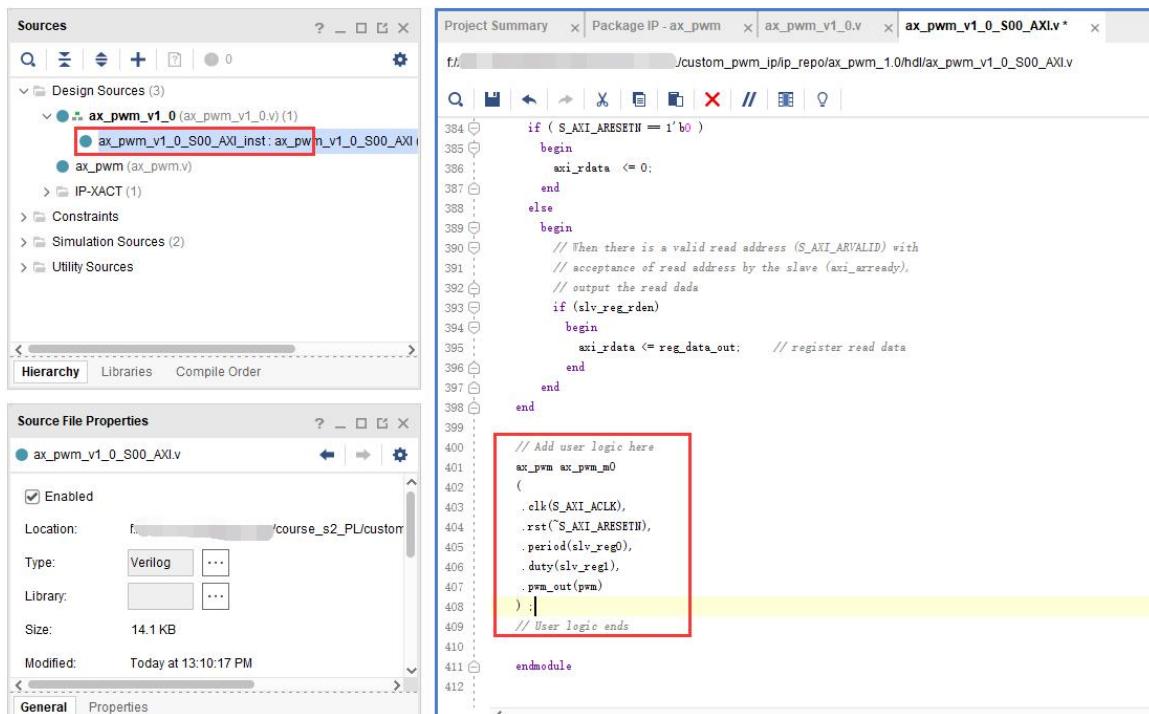
```

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

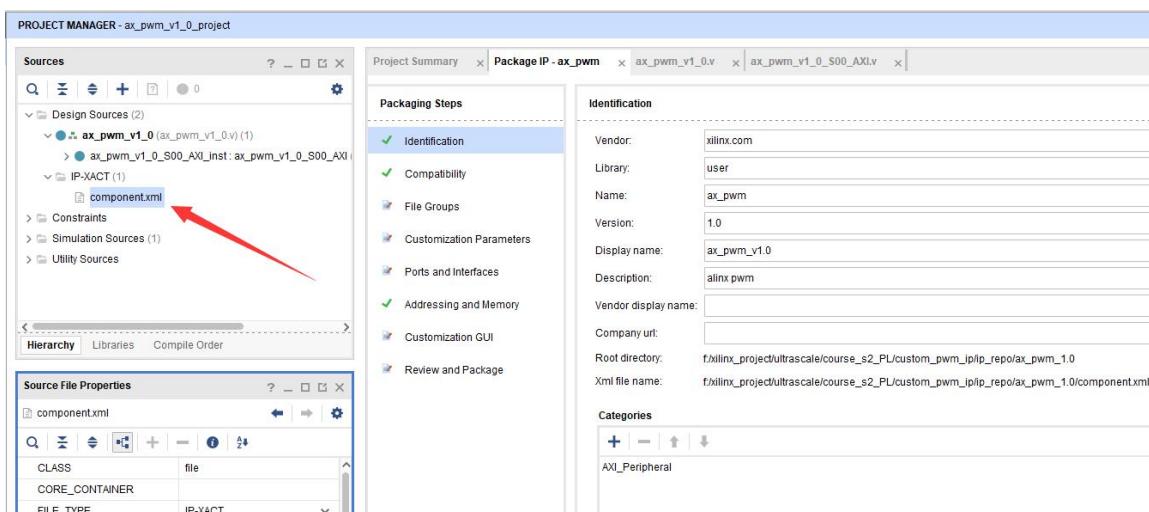
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

```

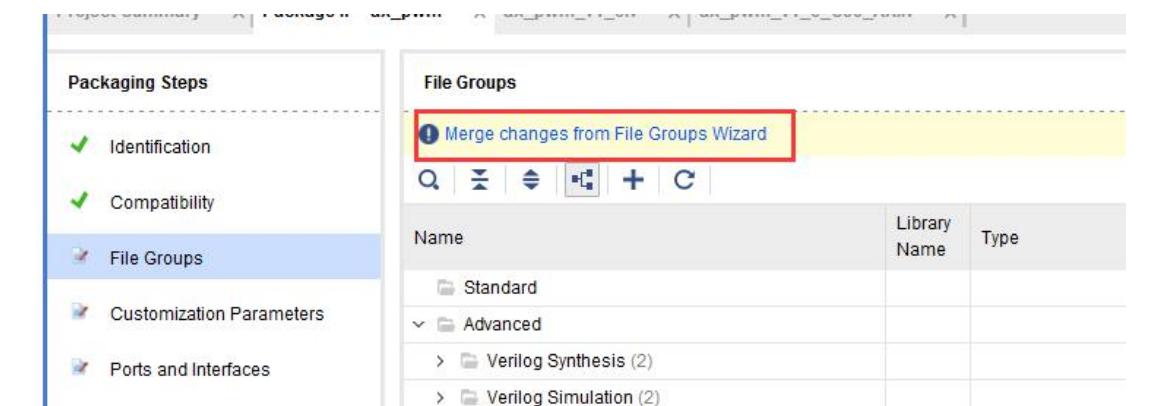
16) Modify the "ax_pwm_v1_0_s00_AXI.v" file, the routine pwm core function code, and use the registers slv_reg0 and slv_reg1 for parameter control of the pwm module.



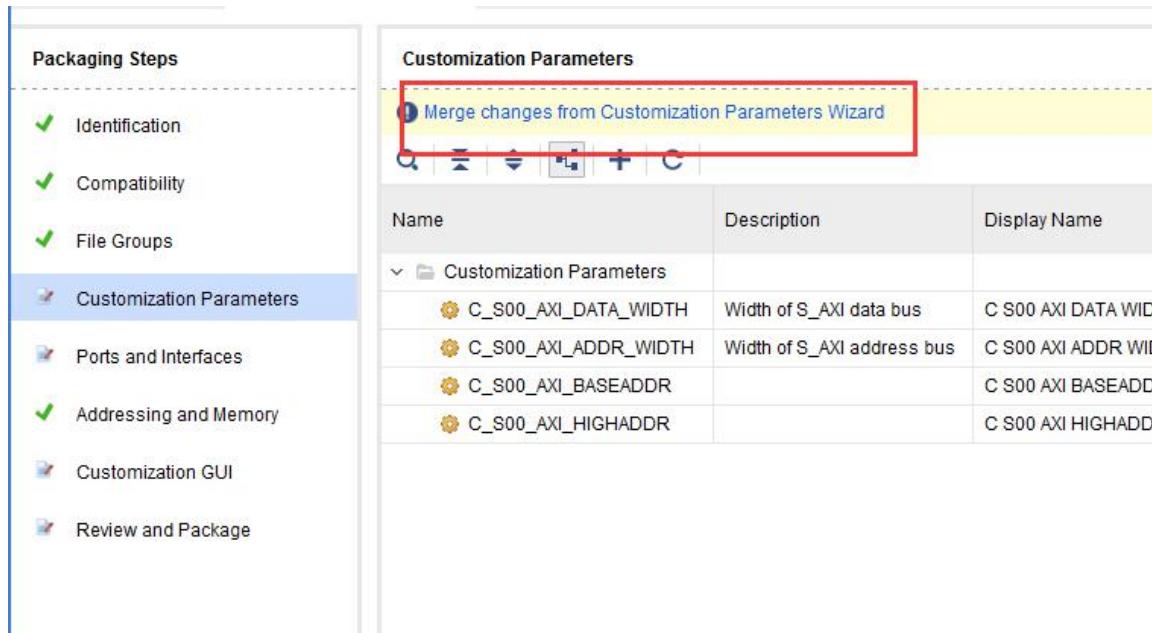
17) Double-click the "component.xml" file



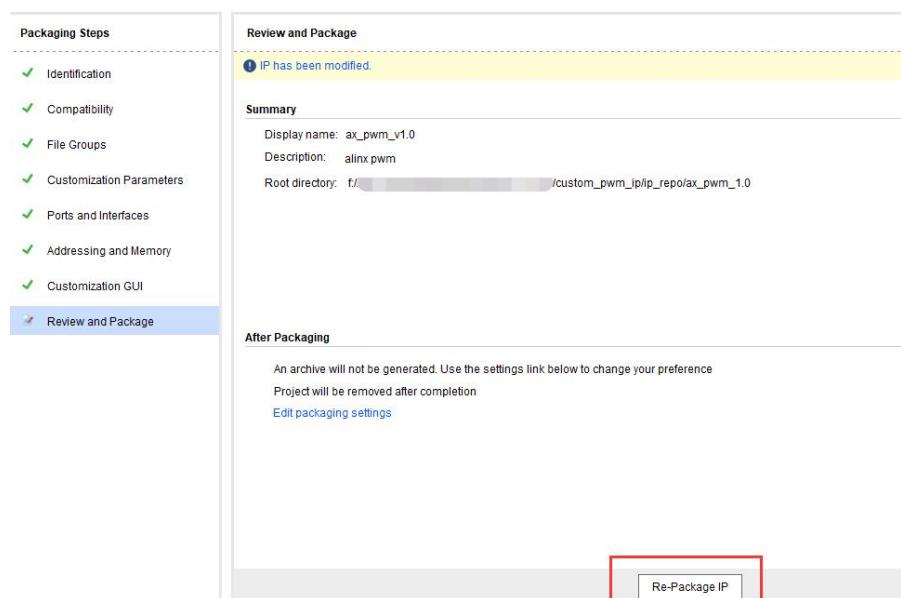
18) Click on "Merge changes from File Groups Wizard" in the "File Groups" option.



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

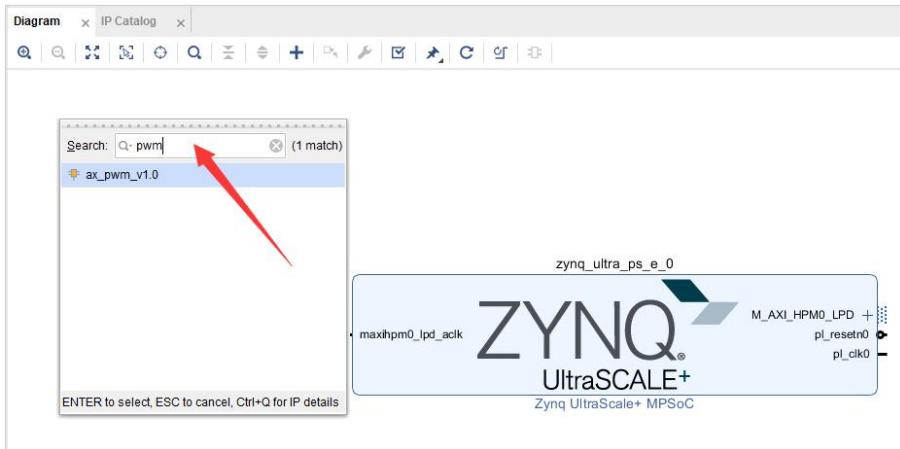


20) Click "Re-Package IP" to complete the IP modification.

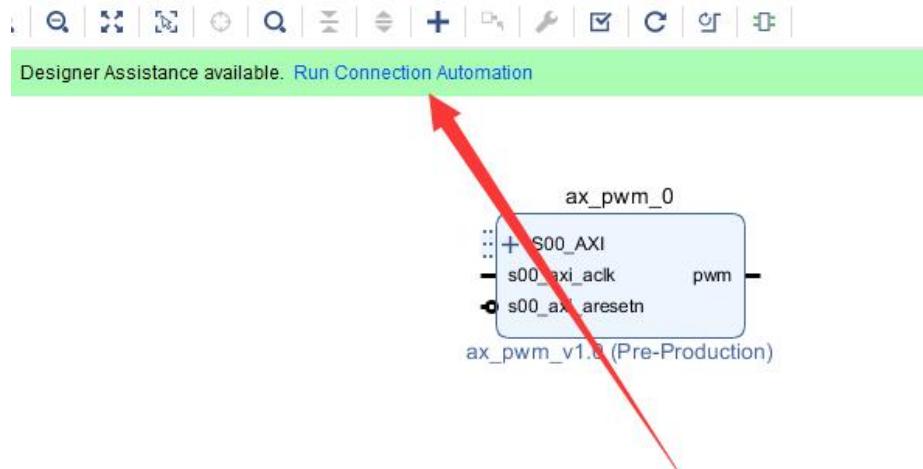


Part 16.2.2: Add a Custom IP to the Project

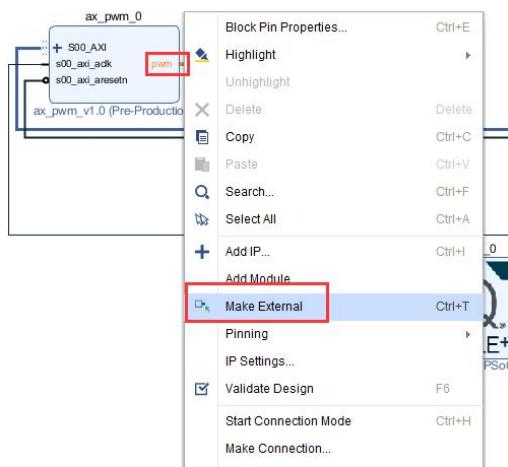
1) Search for "pwm" and add "ax_pwm_v1.0"

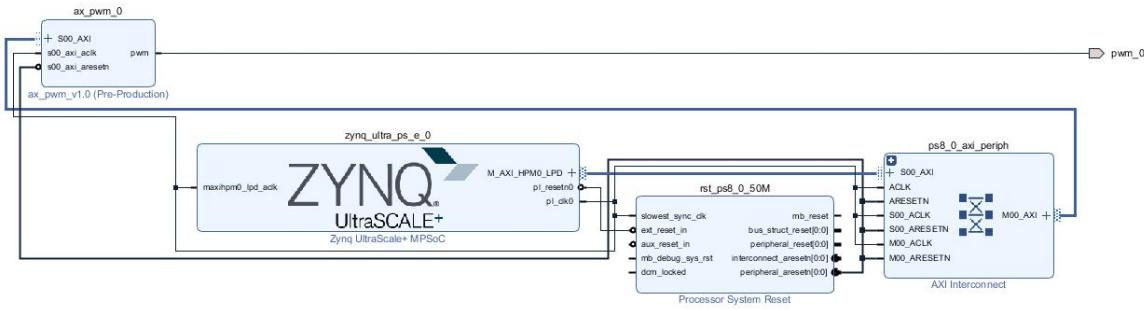


2) Click on "Run Connection Automation"

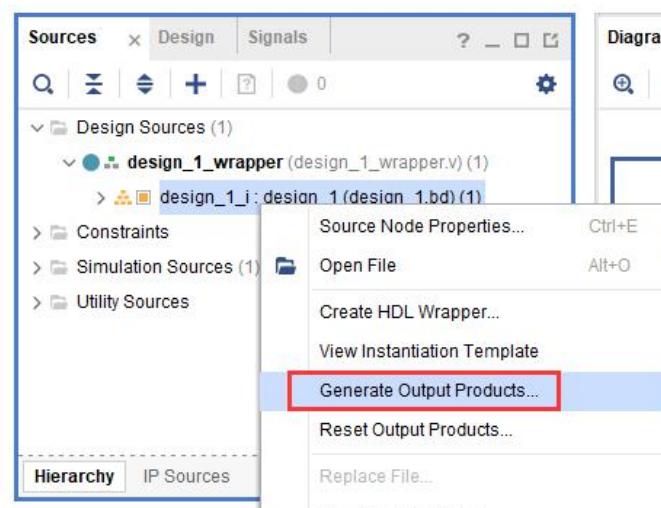


3) Export pwm port





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

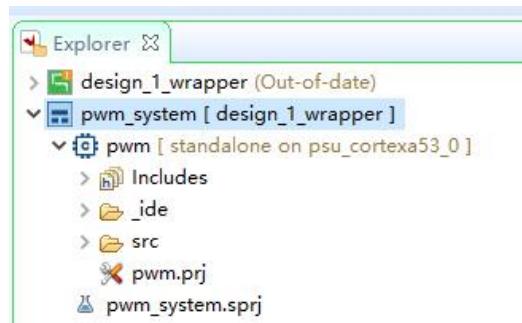
```
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
```

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

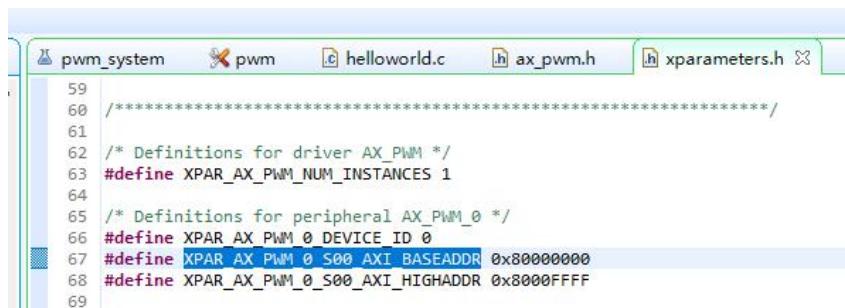
6) Compile and generate a bit file and export the hardware

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");

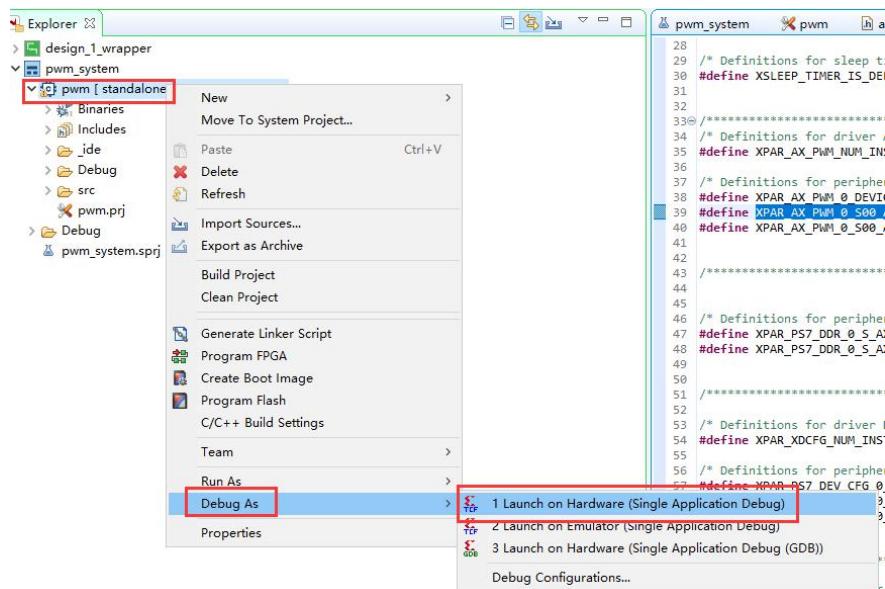
//pwm out period = frequency(pwm_out) * (2^N) / frequency(clk);
AX_PWM_mWriteReg(XPAR_AX_PWM_0_S00_AXI_BASEADDR,
AX_PWM_S00_AXI_SLV_REG0_OFFSET, 17179); //200hz
//duty = (2^N) * (1 - (duty cycle)) - 1
while (1) {
    for (duty = 0xffffffff; duty < 0xffffffff; duty = duty + 100000)
    {
        AX_PWM_mWriteReg(XPAR_AX_PWM_0_S00_AXI_BASEADDR,
AX_PWM_S00_AXI_SLV_REG1_OFFSET, duty);
        usleep(100);
    }
}

cleanup_platform();
return 0;
}

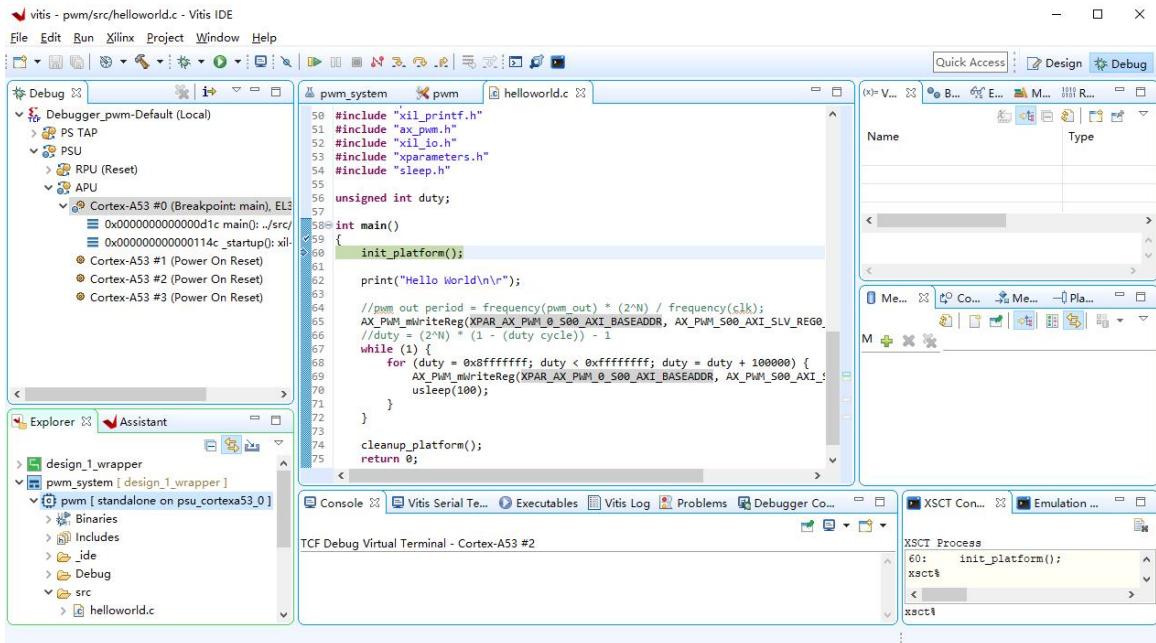
```

4) By running the code, we can see that PL LED shows the effect of a breathing light.

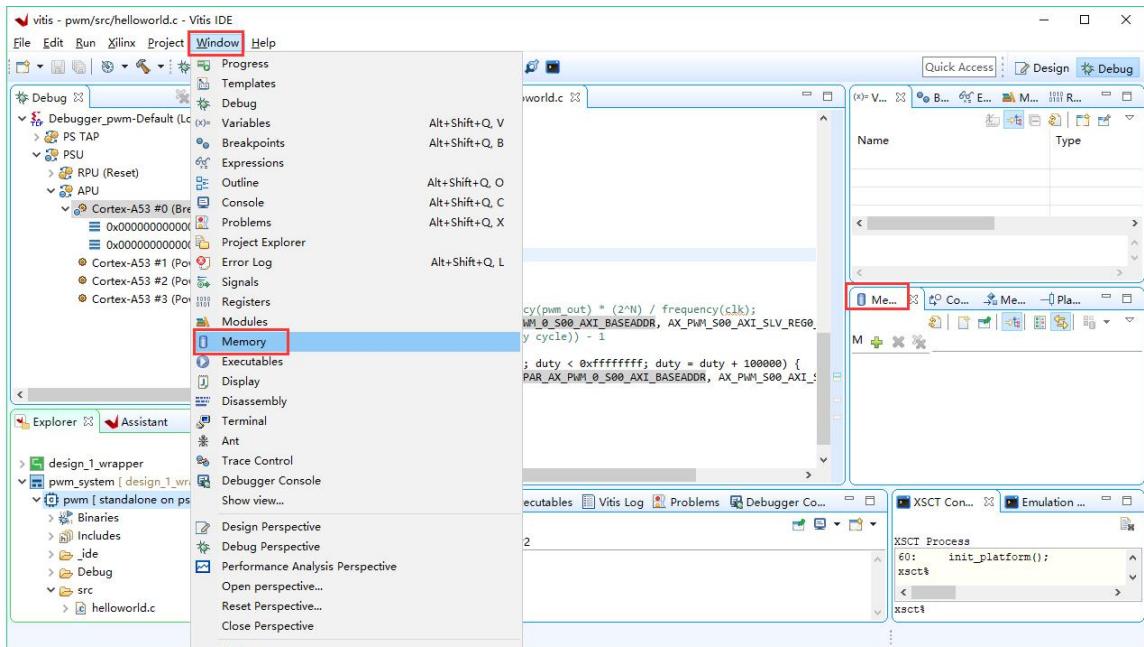
5) Through debug, let's look at the register



6) Enter the debug state and press "F6" to step through

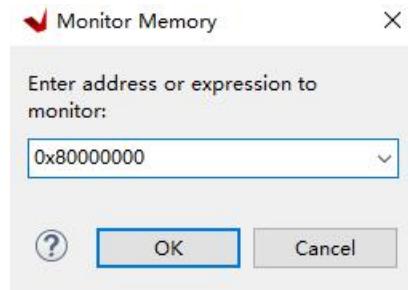


7) View the "Memory" window through the menu

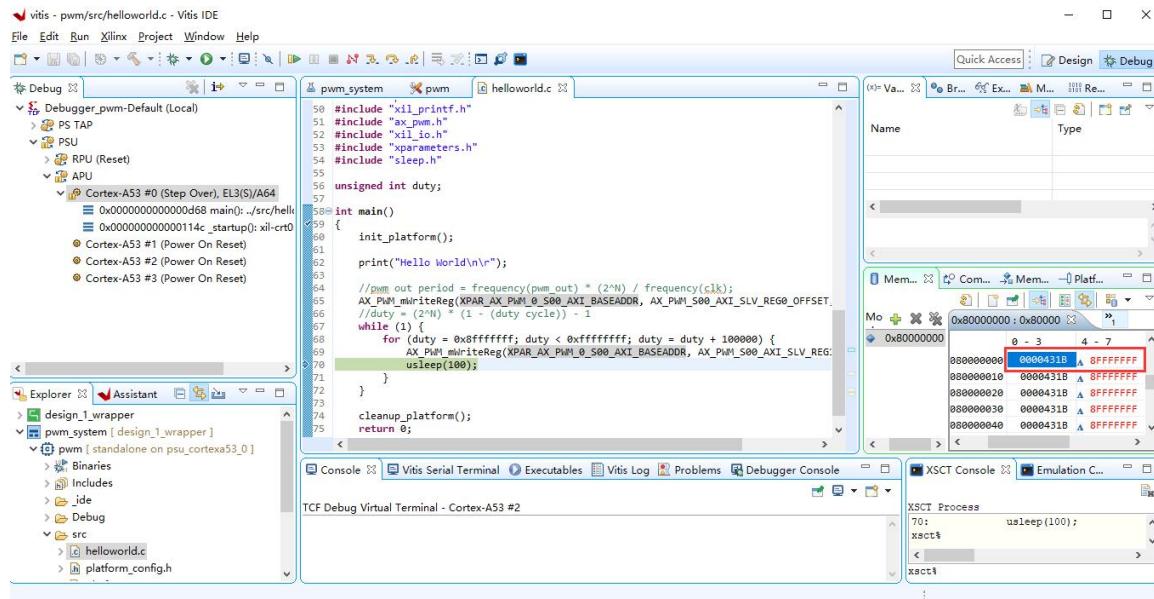


8) Add a monitoring address "0x80000000"





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"

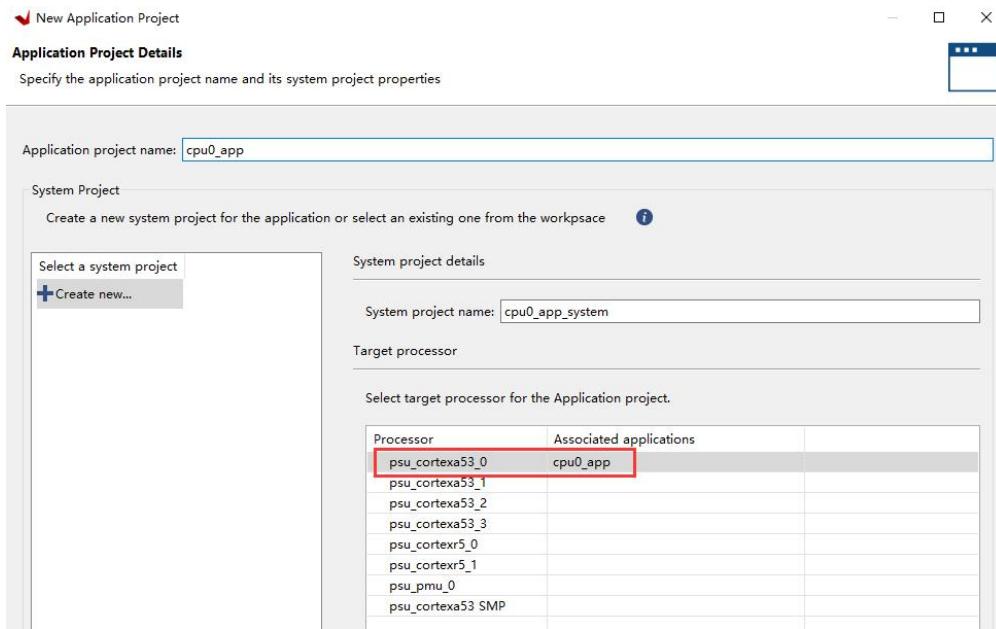
Software Engineer Job Content

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

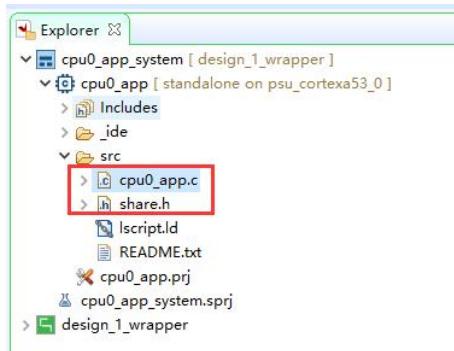
Part 17.1:Vitis Program Development

Part 17.1.1: Create CPU0 Vitis Project

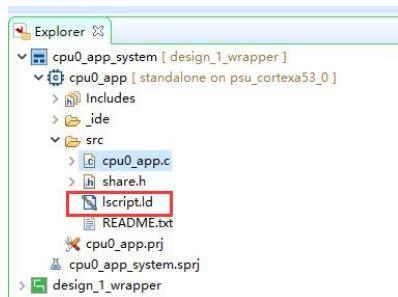
- 1) Create a new project, Note that the CPU selects "ps7_cortexa53_0" , which is "CPU0".



- 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.



Linker Script: lscript.ld

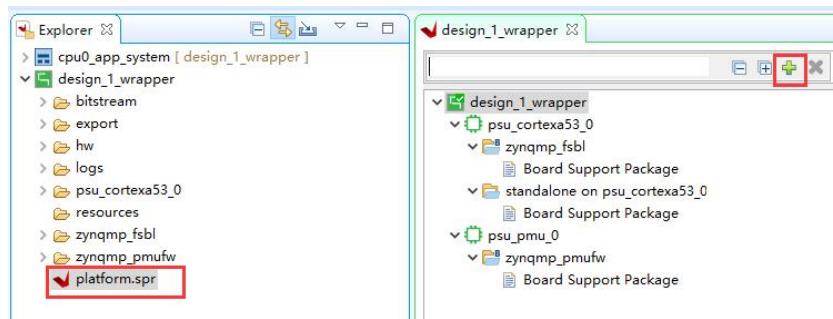
A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define new memory regions, and change the assignment of sections to memory regions.

Available Memory Regions

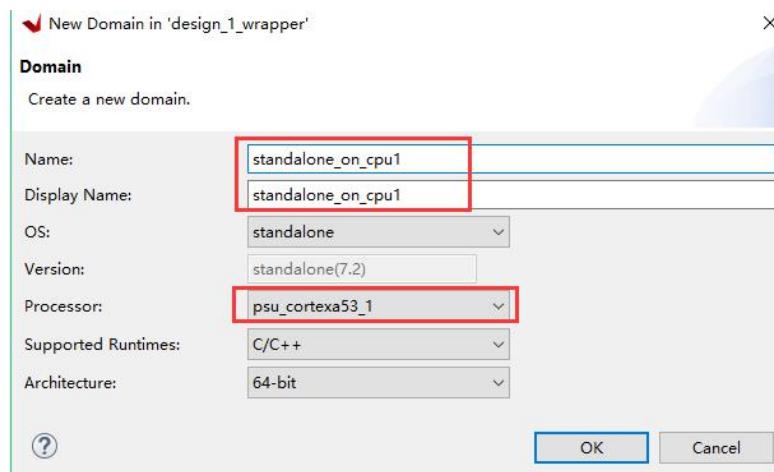
Name	Base Address	Size
psu_ddr_0_MEM_0	0x0	0x3FF00000
psu_ddr_1_MEM_0	0x800000000	0x800000000
psu_ocm_ram_0_MEM_0	0xFFFC0000	0x40000
psu_qspi_linear_0_MEM_0	0xC0000000	0x20000000

Part 17.1.2: Create a CPU1 Vitis project

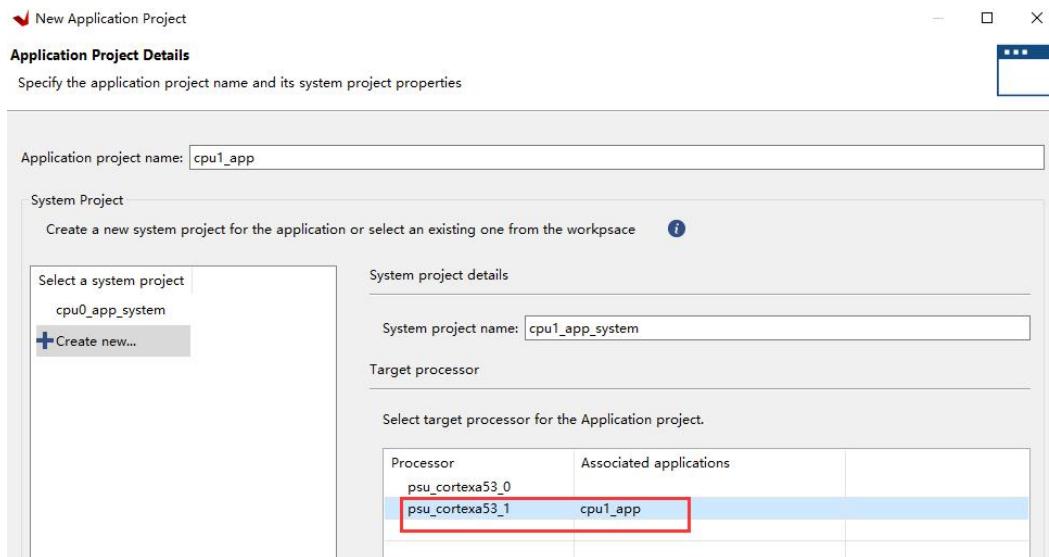
- Before creating a new CPU1 APP project, we can create a new domain based on CPU1, which is the so-called BSP, click "+" in platform.spr



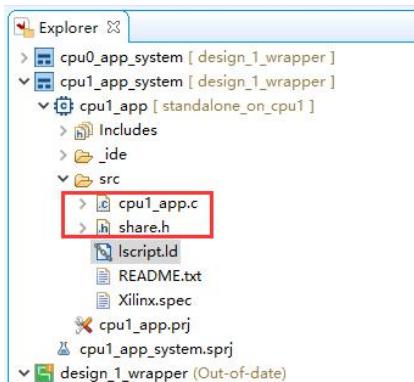
- Fill in the name, and the Processor selects ps7_cortexa9_1, which is CPU1, click OK



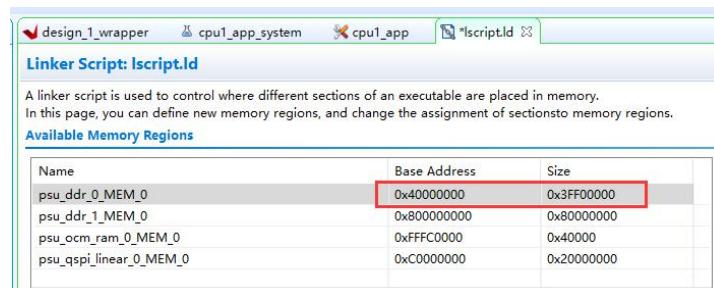
- When creating a new CPU1 project, select CPU1



- 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.

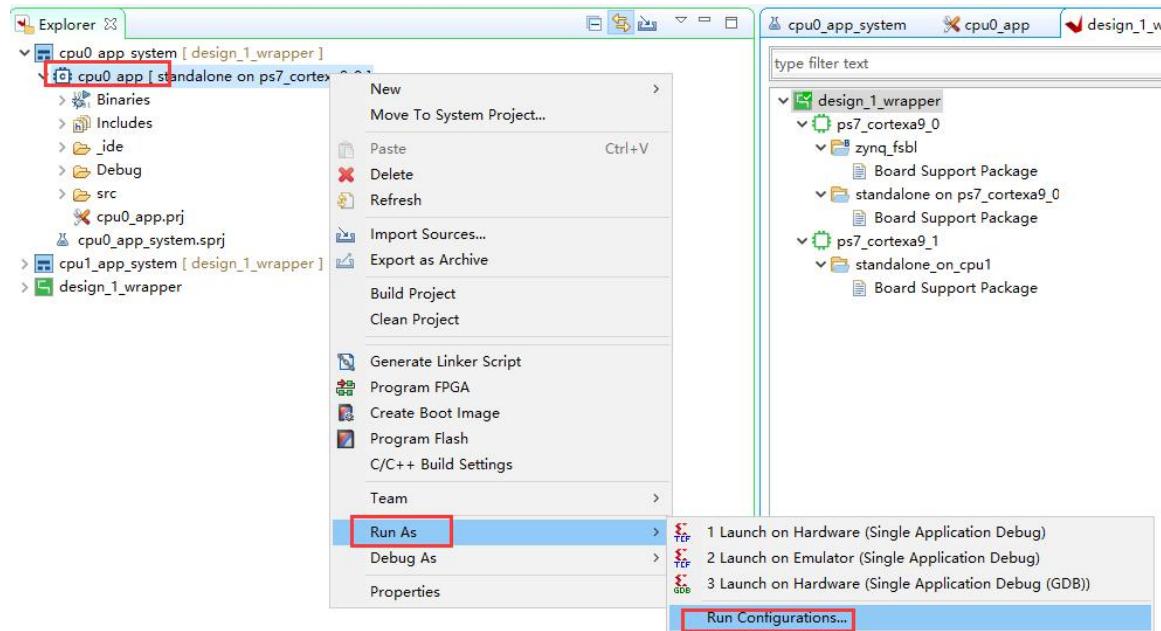
```
unsigned char Cpu1_Data[12] = "Hello Cpu0!" ;
unsigned char *Cpu0Data ;
```

- 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.

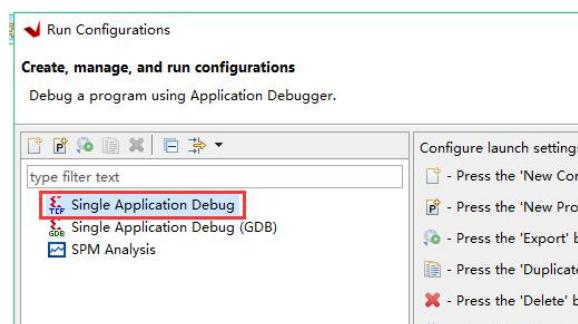
```
/* Connect key interrupt to CPU1 */
XScuGic InterruptMapToCpu(InstancePtr, XPAR_CPU_ID, KEY_INTR_ID) ;
```

Part 17.2: Onboard Verification

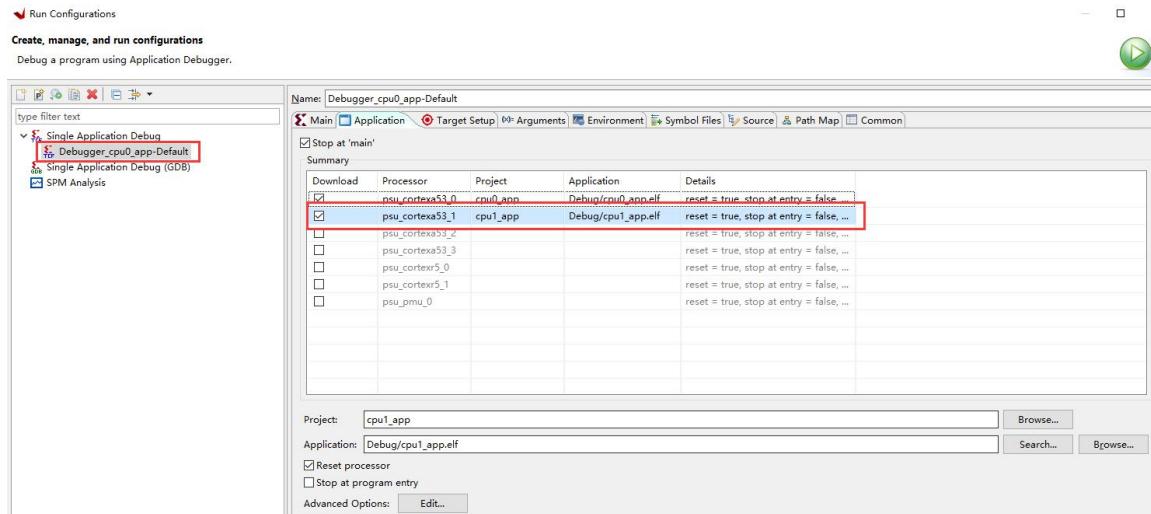
- 1) Run Configurations, and Configuration is as follows:



- 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.

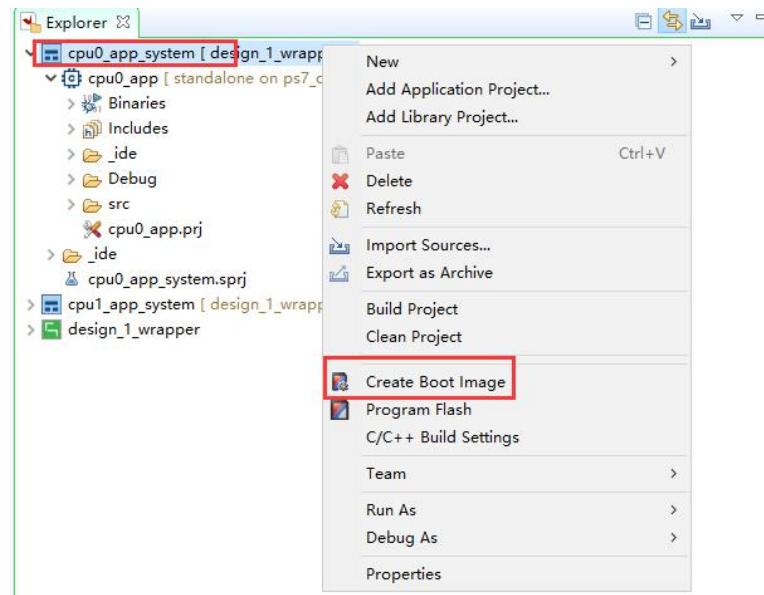
```
Soft Interrupt from CPU0
This is CPU1, Now Start to Print:
Hello Cpu1!
```

- 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.

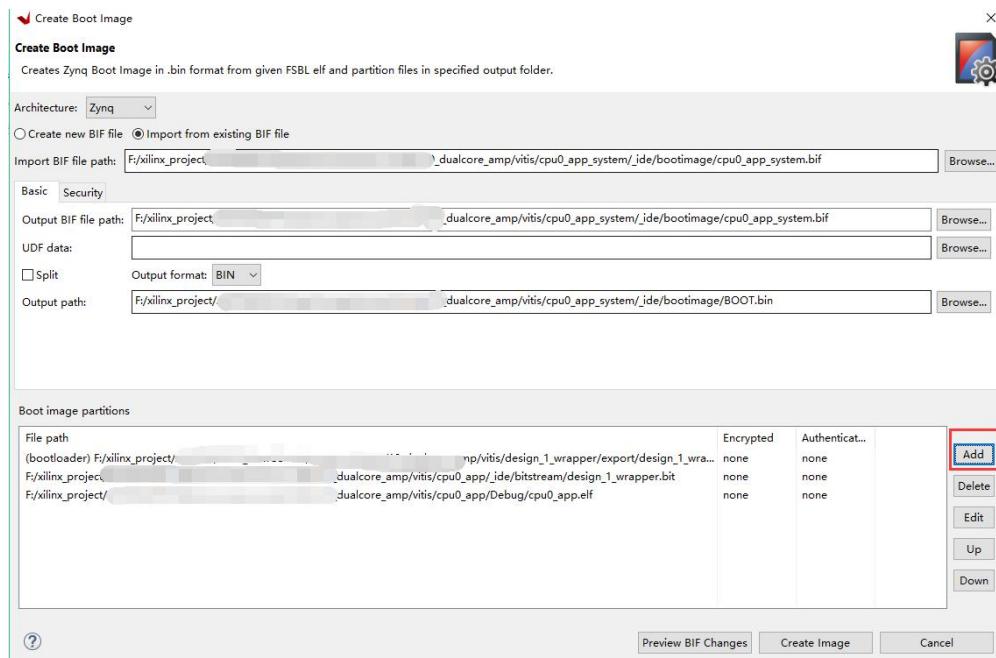
```
Soft Interrupt from CPU0
This is CPU1, Now Start to Print:
Hello Cpu1!
Soft Interrupt from CPU1
This is CPU0, Now Start to Print:
Hello Cpu0!
```

Part 17.3: QSPI Flash Startup

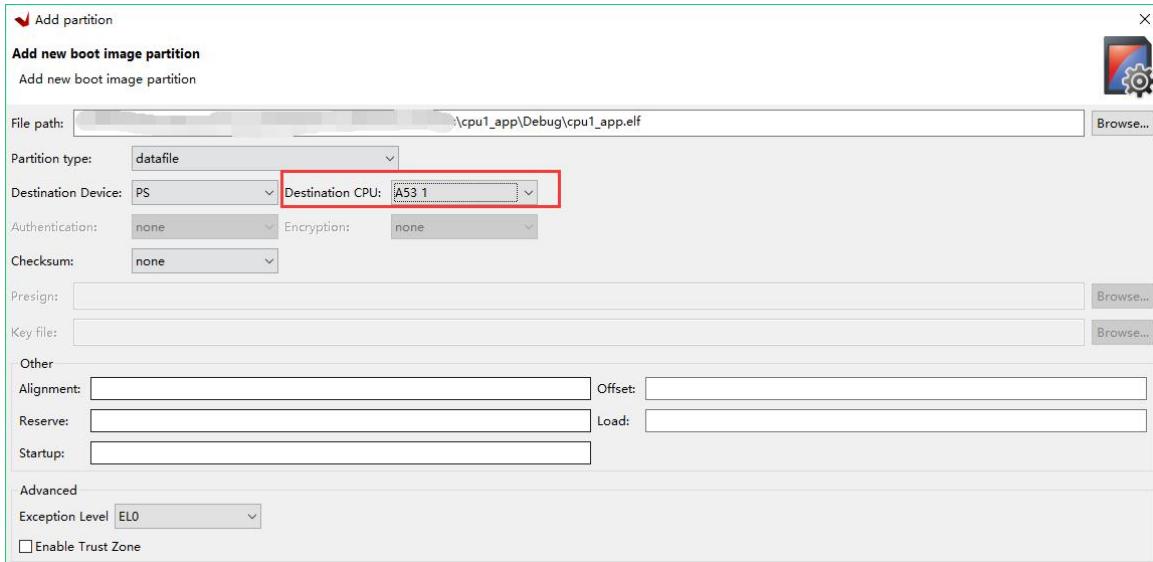
The way to generate BOOT.BIN is different from the previous Build Project generation, we need to configure it. Right-click on the system of CPU0 and select Create Boot Image



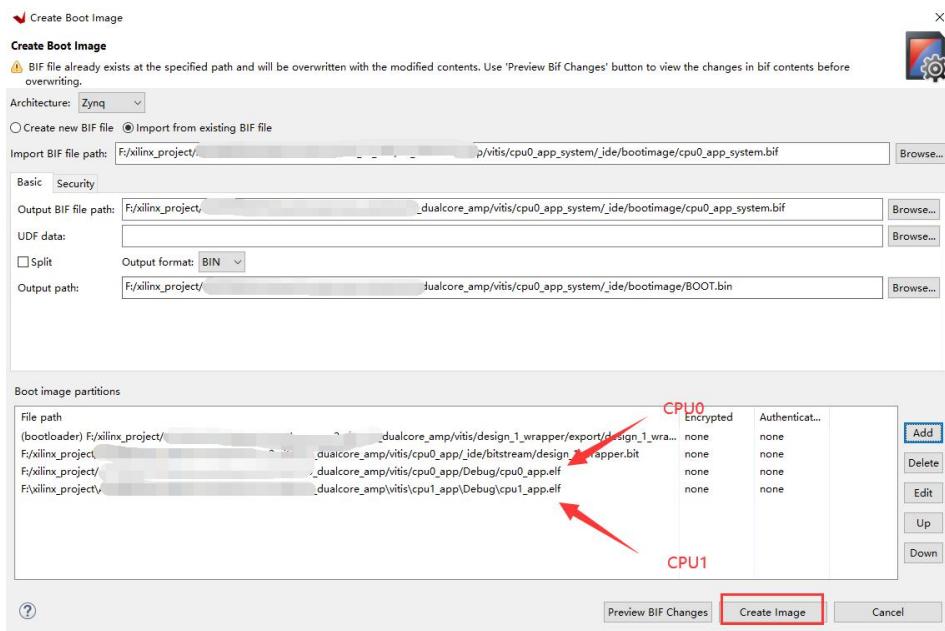
Click Add to add the elf file of CPU1



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.

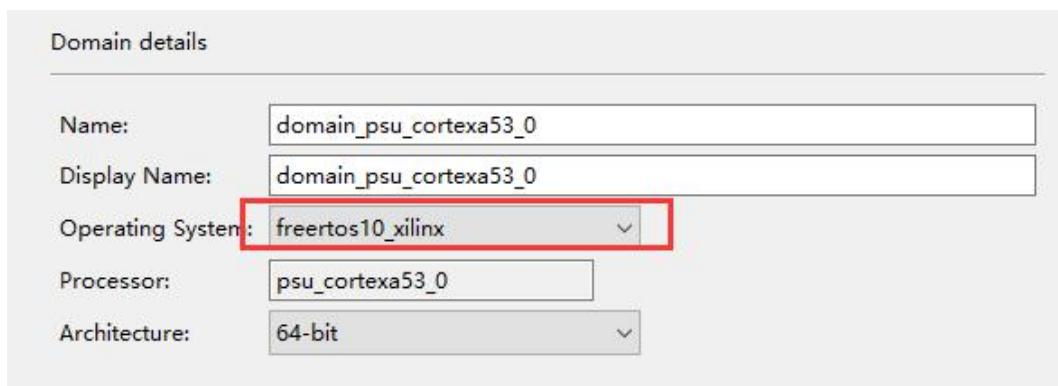
This experiment is based on the " PS Side Use of Dual-Core AXI GPIO" tutorial, the hardware environment does not need to be modified.

Software Engineer Job Content

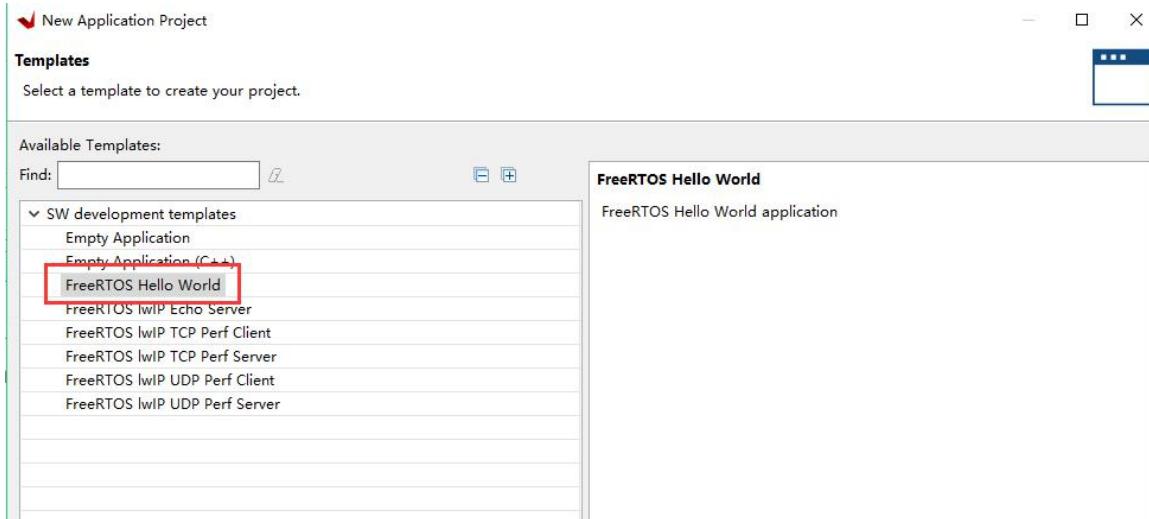
The following is the content that FPGA engineers are responsible for.

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 );

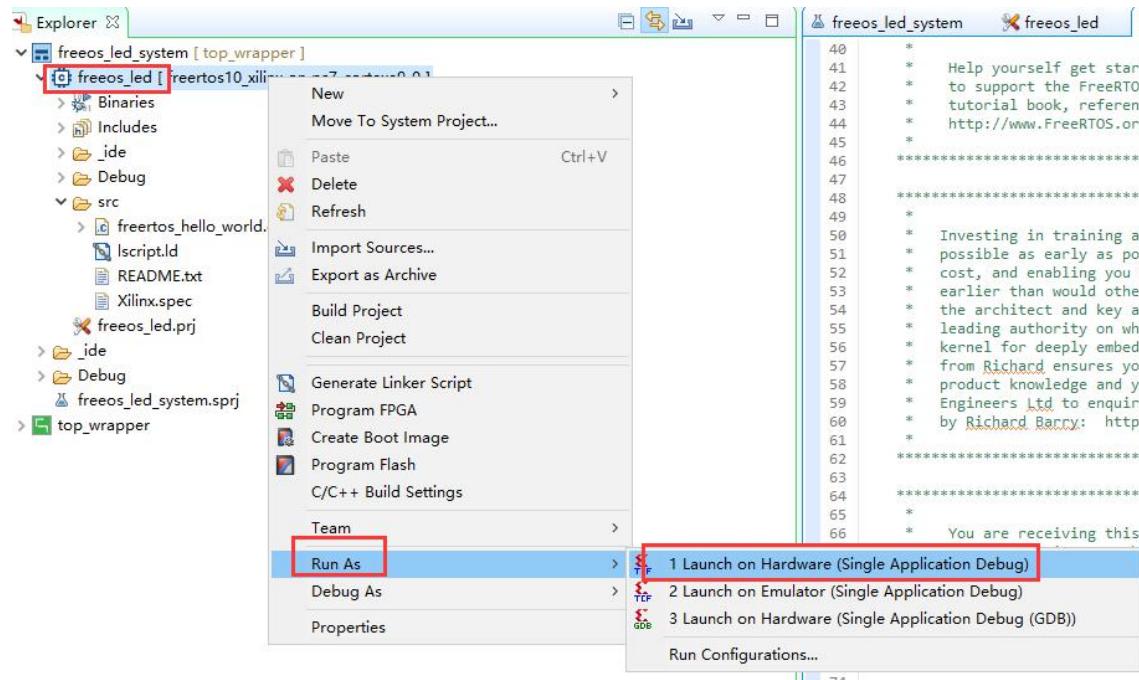
xTaskCreate( prvRxTask,
    ( const char * ) "Rx",
    configMINIMAL_STACK_SIZE,
    NULL,
    tskIDLE_PRIORITY + 1,
    &xRxTask );
```

- 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.

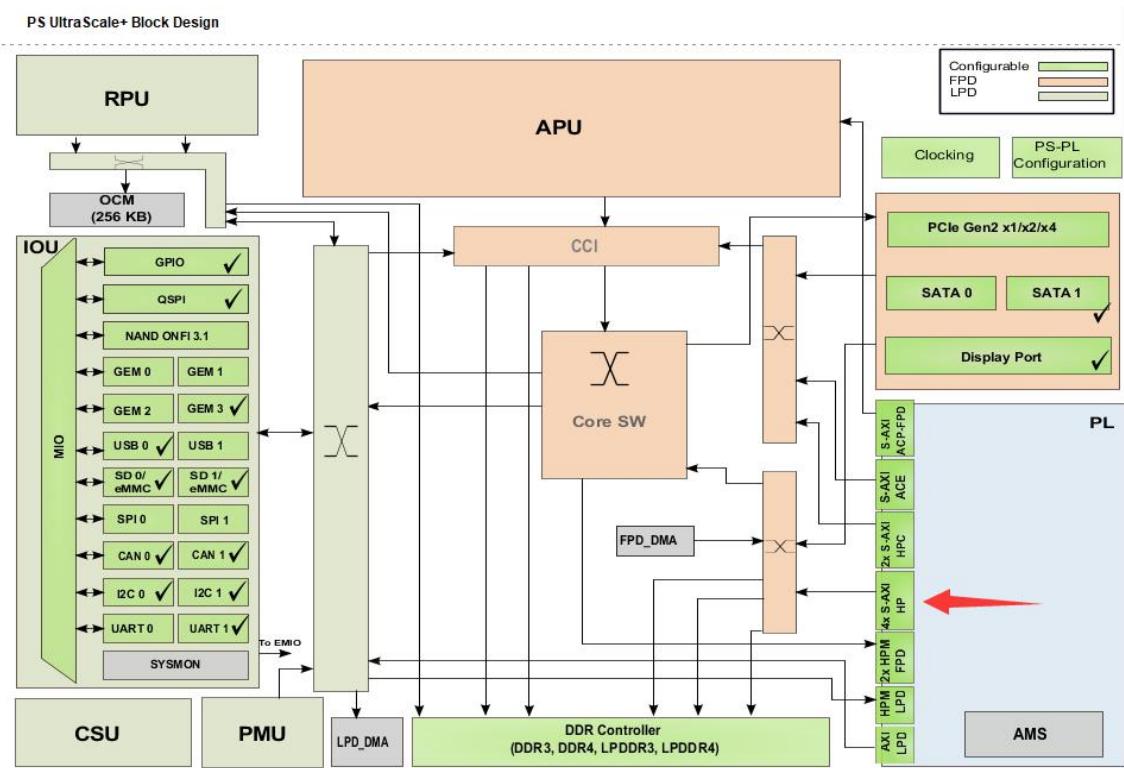
Normally we would think of using DMA to do this, but various protocols are very troublesome and flexibility is relatively poor. This section of the course explains how to directly read and write data on the PS side ddr through the AXI bus. This involves the AXI4 protocol, vivado FPGA debugging and so 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.

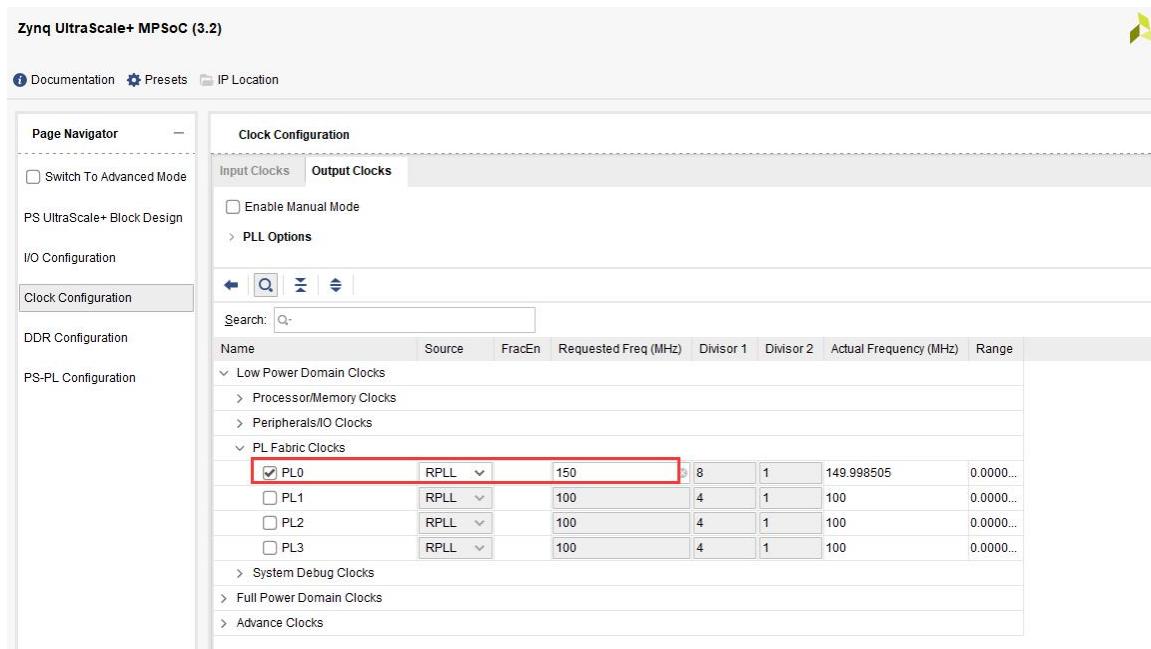
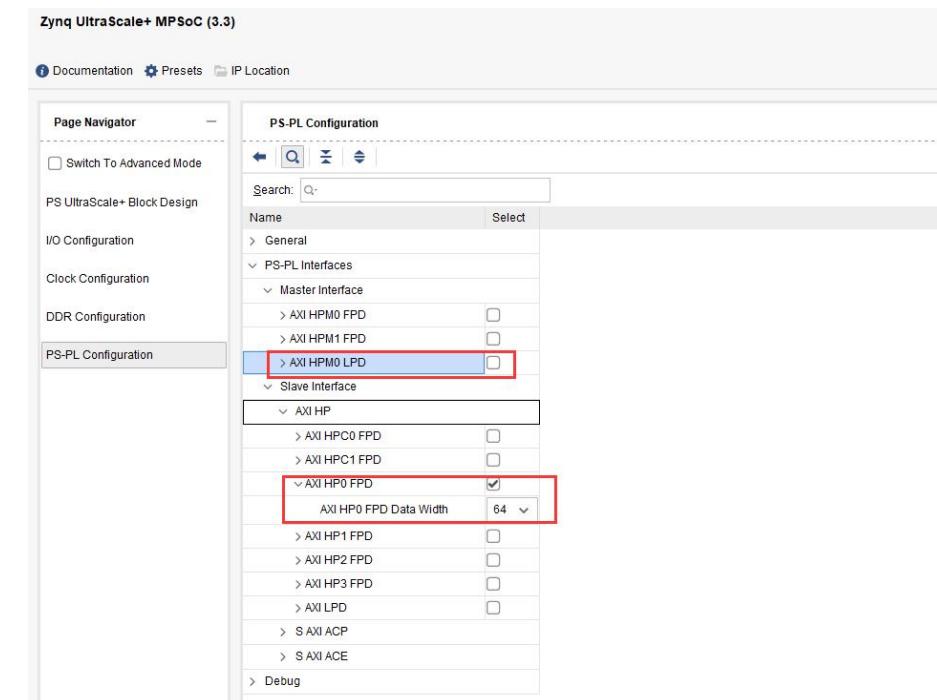


FPGA Engineer Job Content

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

Part 19.2: Hardware Environment

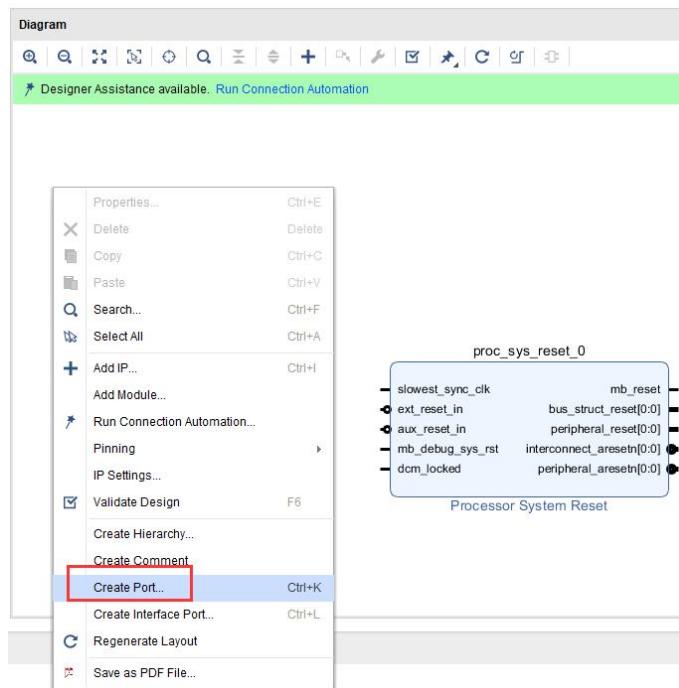
- 1) Based on the "ps_hello" project, the HP configuration in the vivado interface is as shown in the figure below (HP0~HP3). There are enable control and data bit width selection. You can choose 32bit, 64bit or 128bit. In our experiment, HP0 is configured to be 64bit wide, the clock used is 150Mhz, 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.



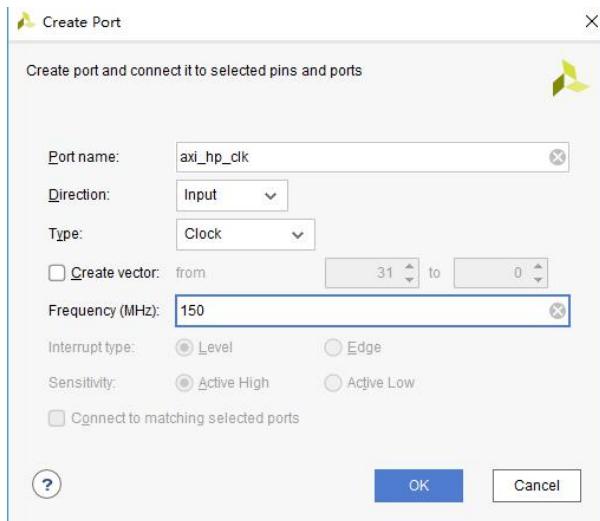
2) Add reset module for reset



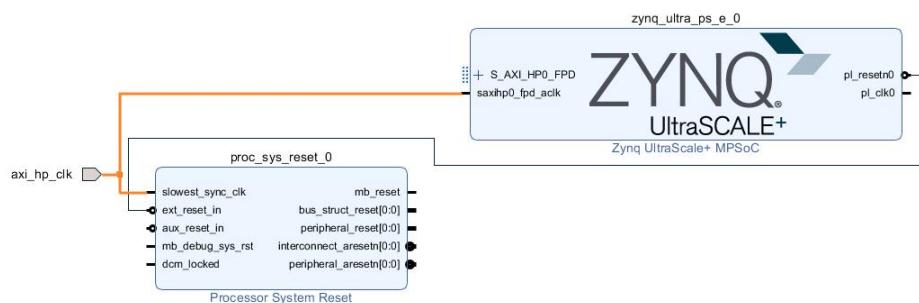
- 3) Right-click on the blank space and select "Create Port"



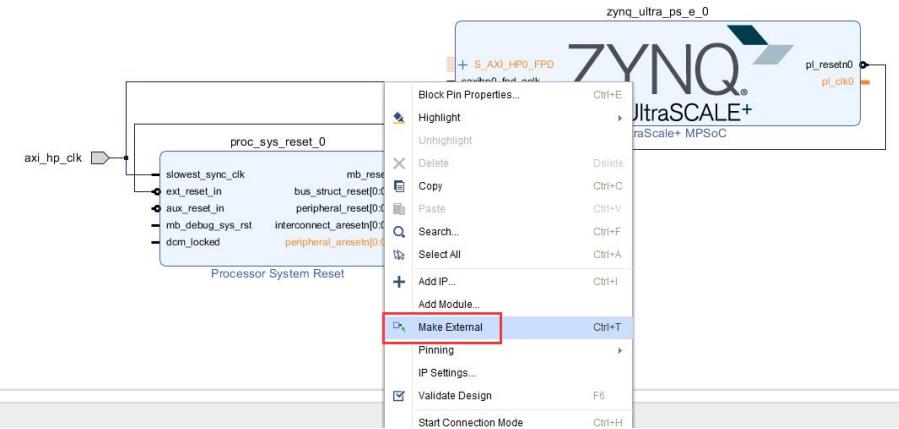
Configuration as shown



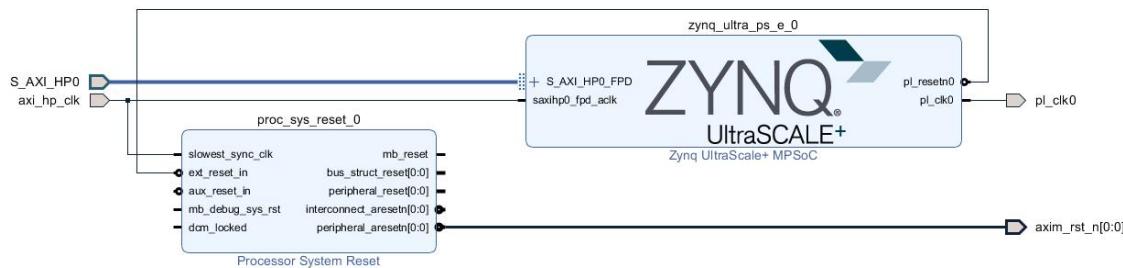
- 4) Connect the clock and reset



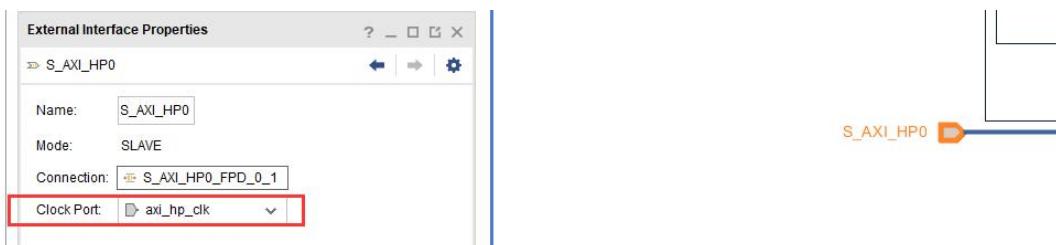
- 5) Select the pin and click Make External to export the signal



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

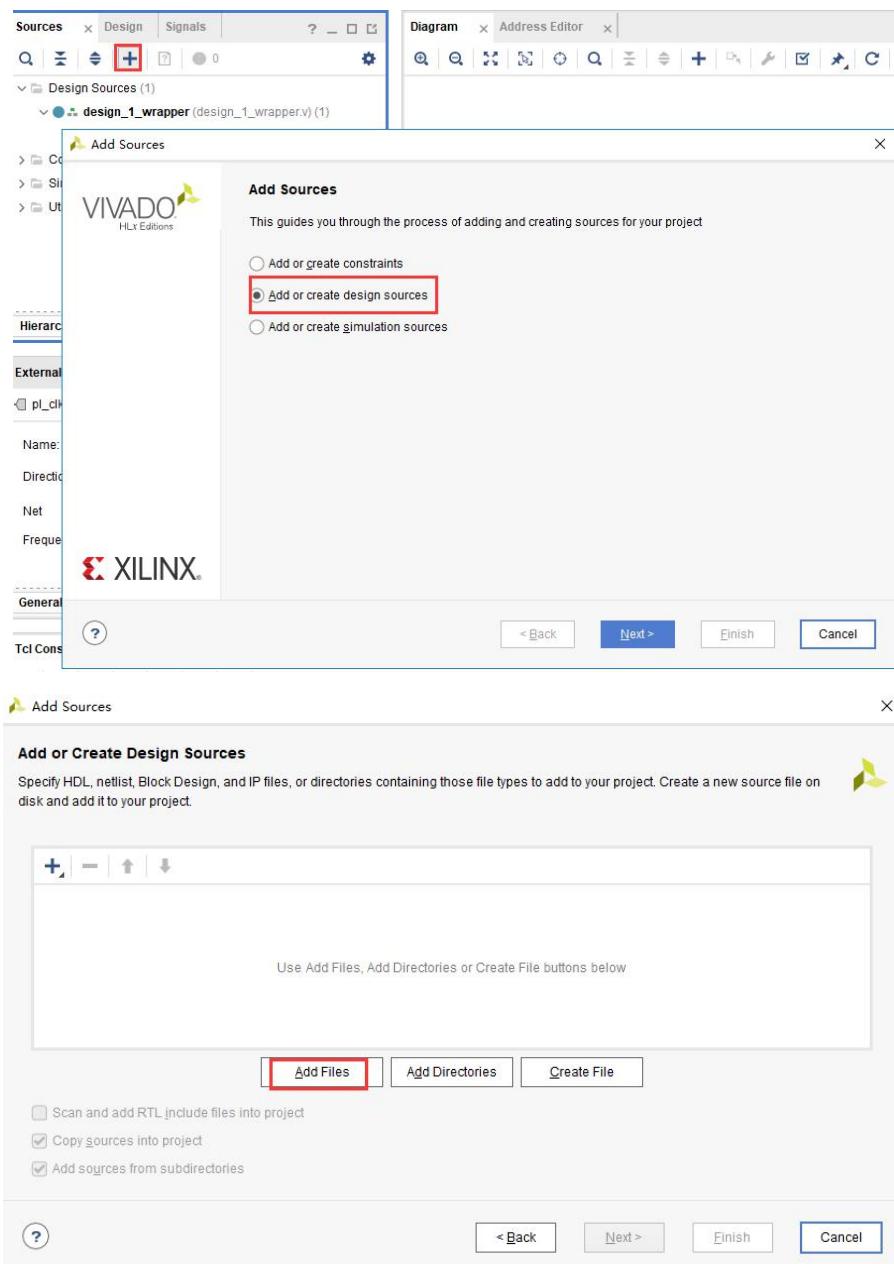


After allocation, you can see the address space for accessing DDR, QSPI, OCM

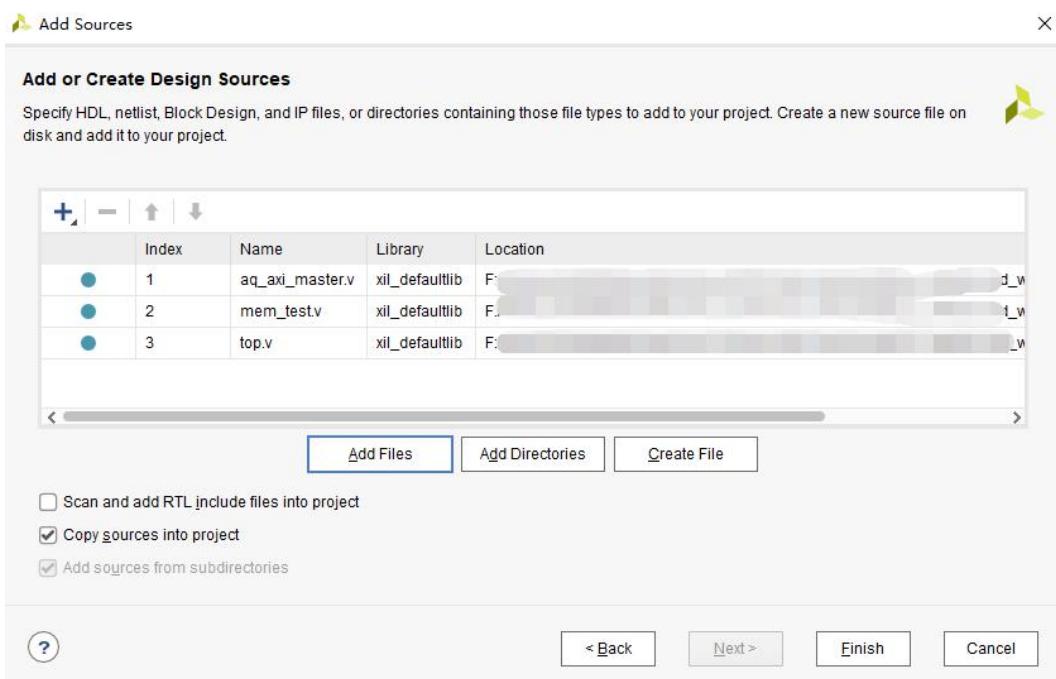
Cell	Slave Interface	Base Name	Offset Address	Range	High Address
External Masters					
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_DDR_LOW	0x0000_0000	2G	0x7FFF_FFFF
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_QSPI	0xC000_0000	512M	0xFFFF_FFFF
zynq_ultra_ps_e_0	S_AXI_HP0_FPD	HP0_LPS_OCM	0xFF00_0000	16M	0xFFFF_FFFF

Save the design and regenerate Output Product

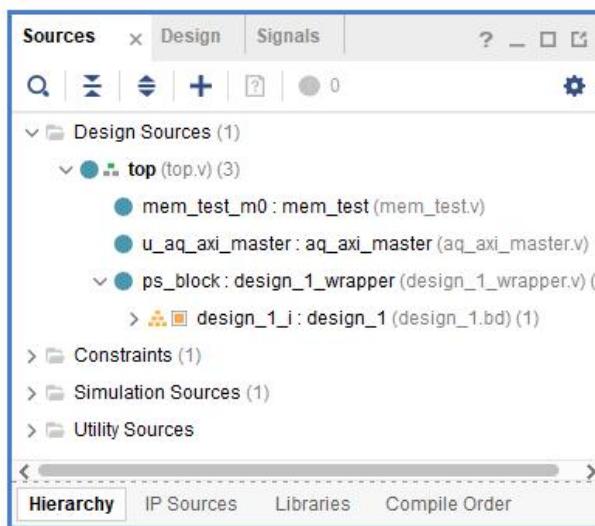
7) Add hdl files



8) Click Finish



HDL hierarchical relationship update results

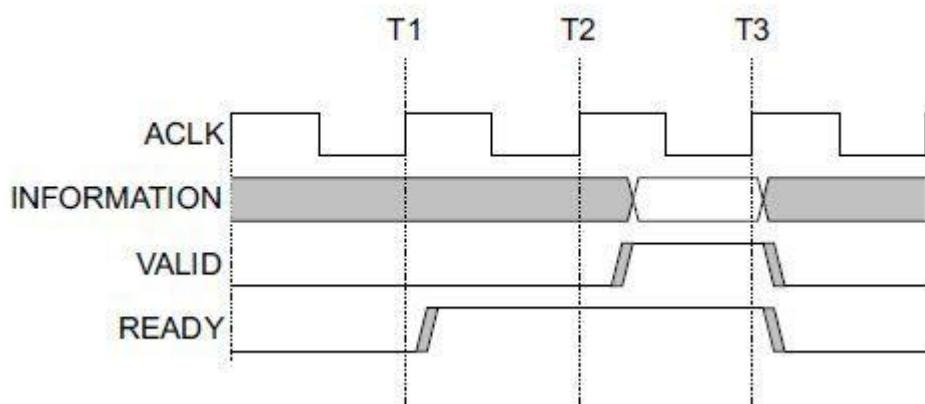


Part 19.3: PL Side AXI Master

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.



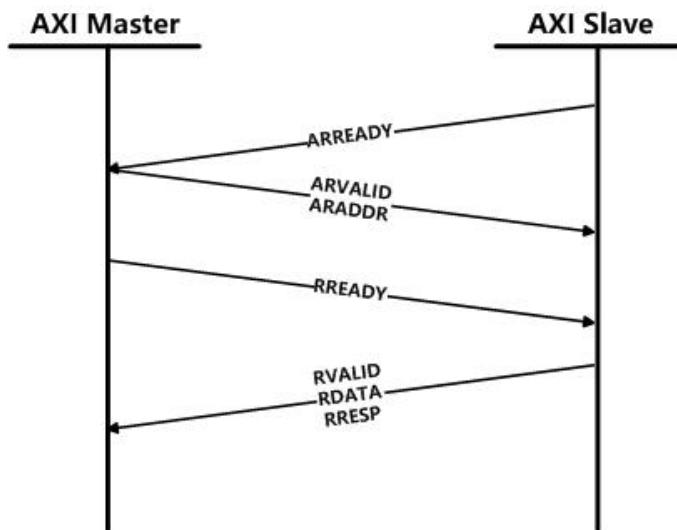
The AXI bus is divided into five channels:

- 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

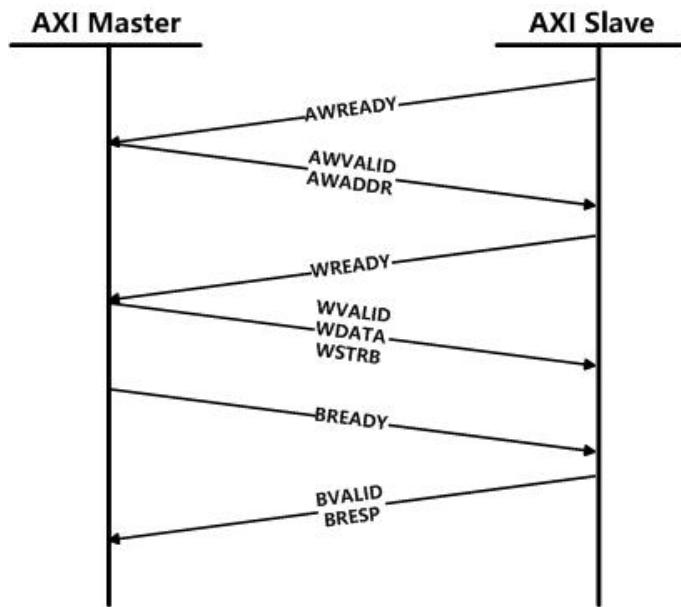
“ACLK” is “axi” bus clock, “ARESETN” is “axi” bus reset signal, active low; read/write data and read/write address class signal width

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

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.

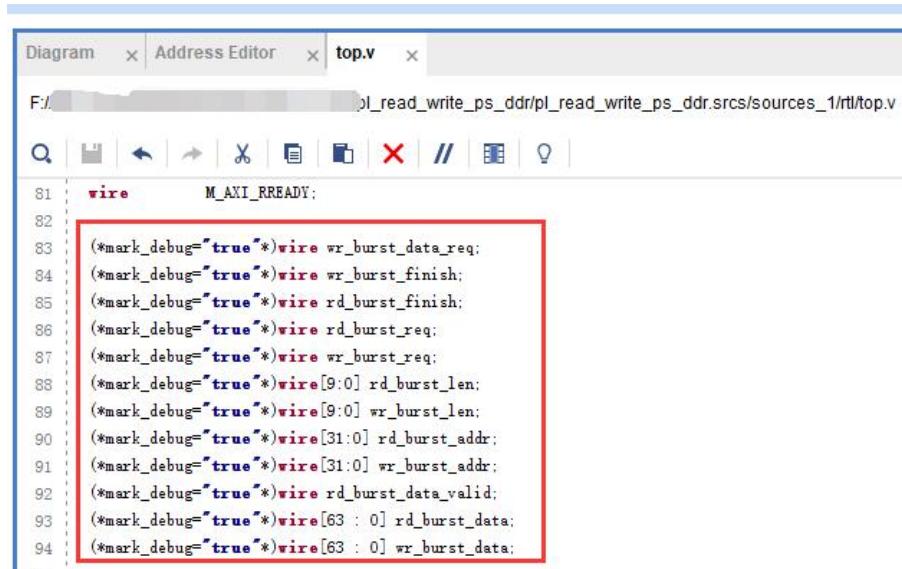
Part 19.4: Verification of ddr Read and Write Data

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.srscs/sources_1/rtl/top.v

Q | H | < | > | X | D | F | // | E | ? |
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;
```

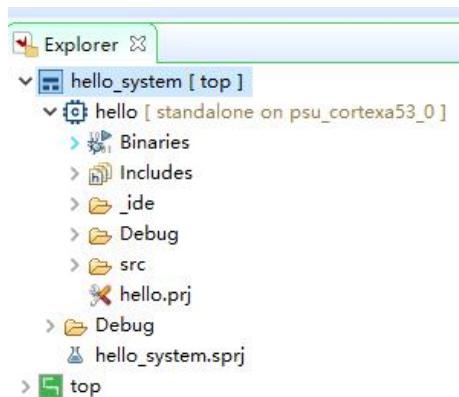
The specific adding method has been discussed in "PL's "Hello World" LED Experiment", you can refer to it.

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

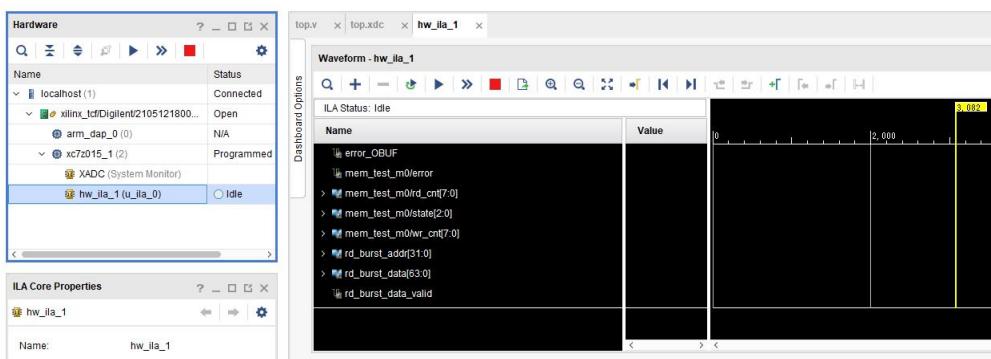
follows



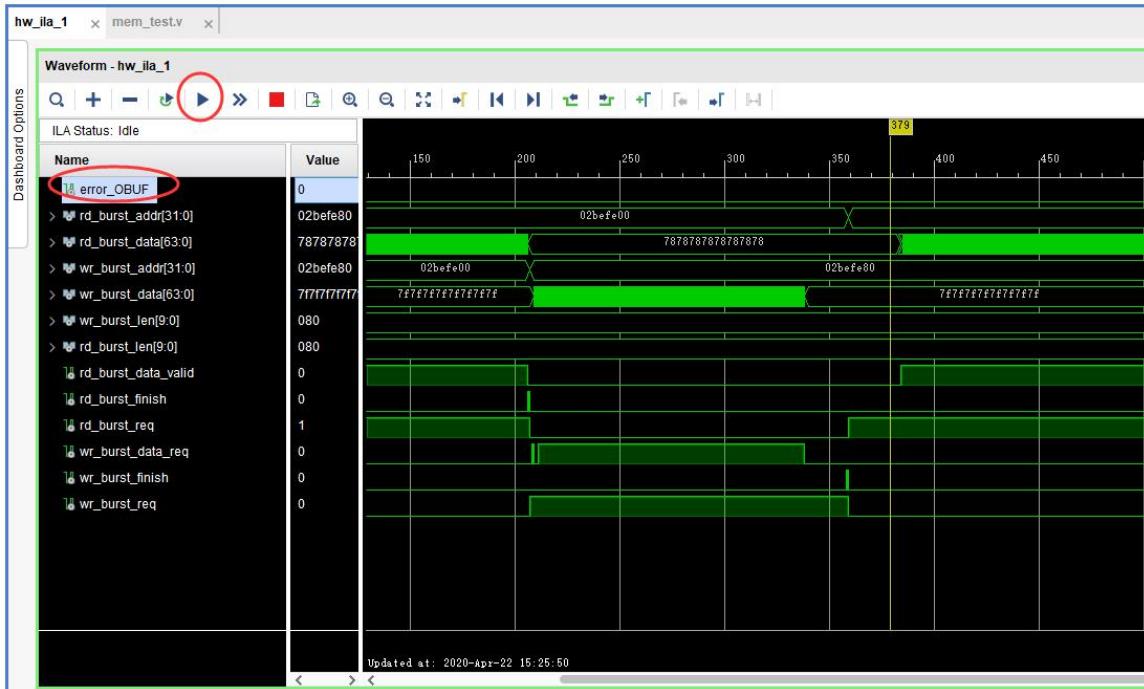
After downloading the program through vitis, the system will reset and download the bit file of FPGA. Then return to the vivado interface and click on the Program and Debug column to automatically connect to the target as shown in the figure below:



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

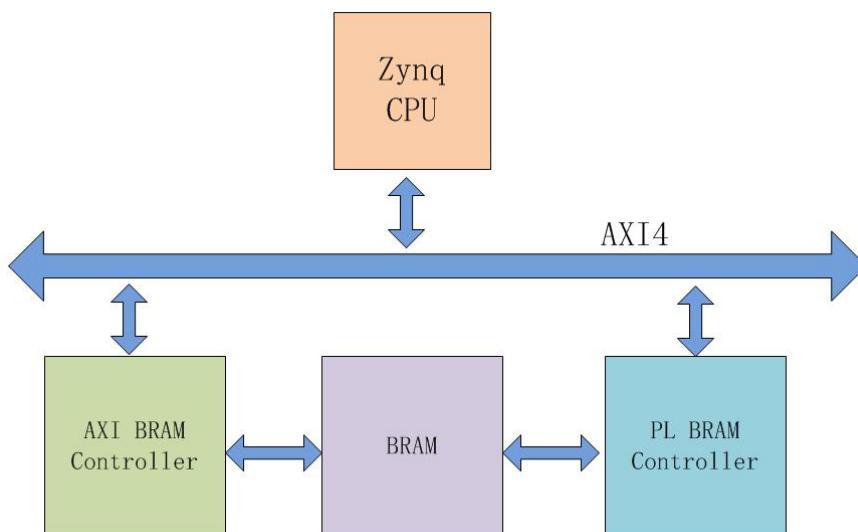
Part 20: Realize PS and PL Data Interaction through BRAM

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.



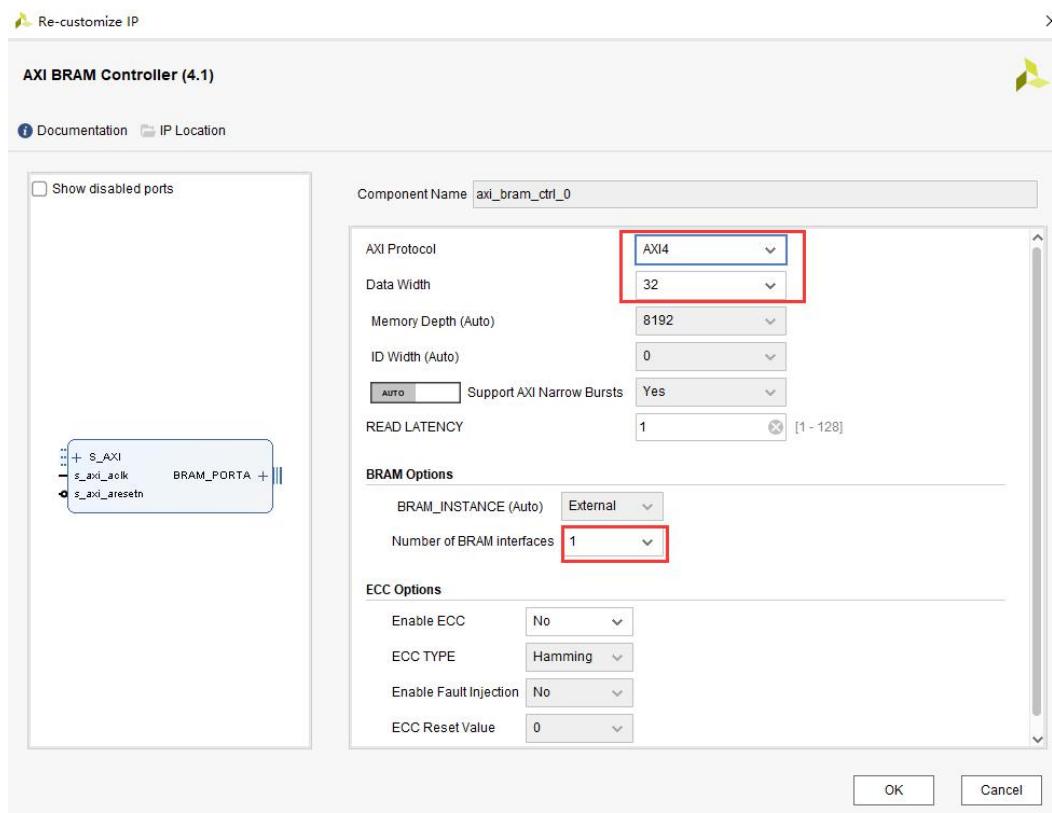
FPGA Engineer Job Content

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.



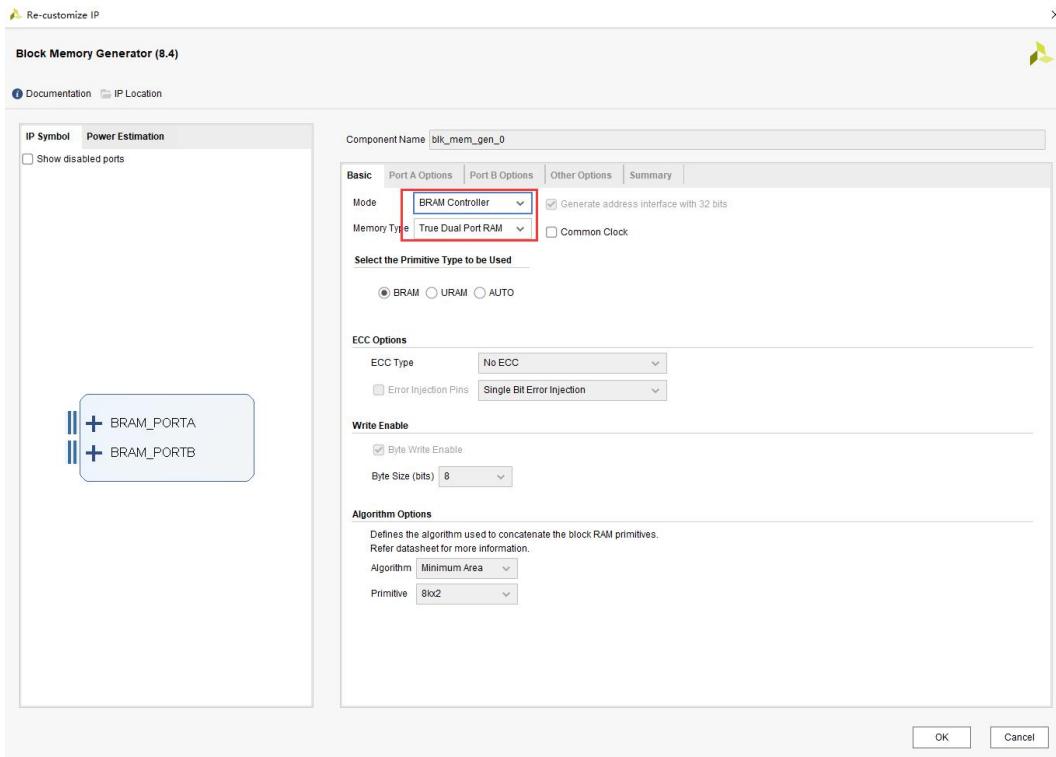
Since the "AXI4" bus is byte-addressed, the "BRAM" data width

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]

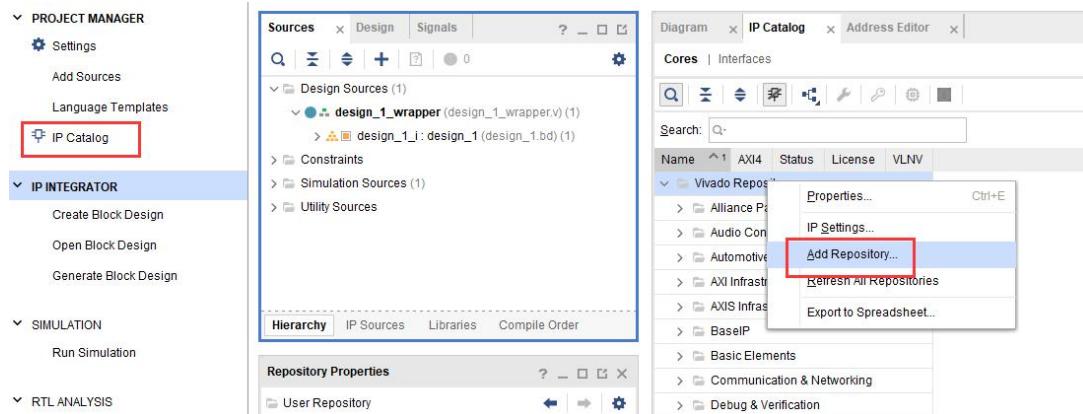
- 2) Add BRAM module, the BRAM settings are as follows, there are two mode options, “standalone” mode, which can freely configure the data width and depth of the RAM. “BRAM Controller” mode. In this mode, the address line and data port default to 32 bits. In this experiment, since the BRAM controller 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.



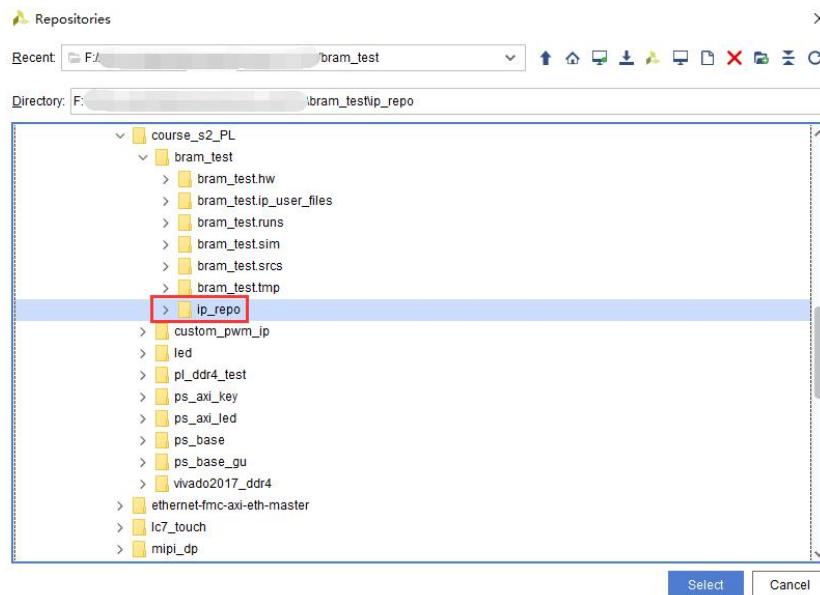
- 3) Add a custom “PL RAM” controller “pl_ram_ctrl”, the function is very simple, start the “BRAM” data after the “start” signal is valid, the read data can be observed by the “ILA” logic analyzer, and after the “PL RAM” controller reads the “BRAM”, it starts to write data to the BRAM. After writing 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

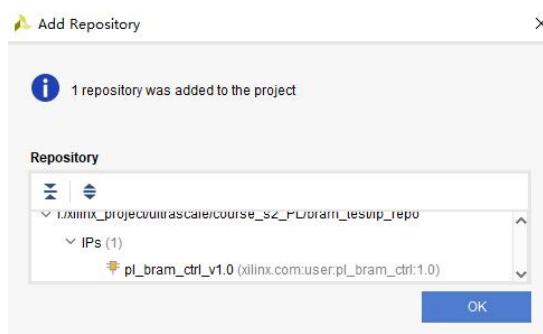
If you want to add a custom IP to IP library, click on the “IP Catalog”, right click the “Add Repository” in “Vivado Repository”



Find the folder where the custom IP is located and click Select



Pop up window, select IP, Click OK

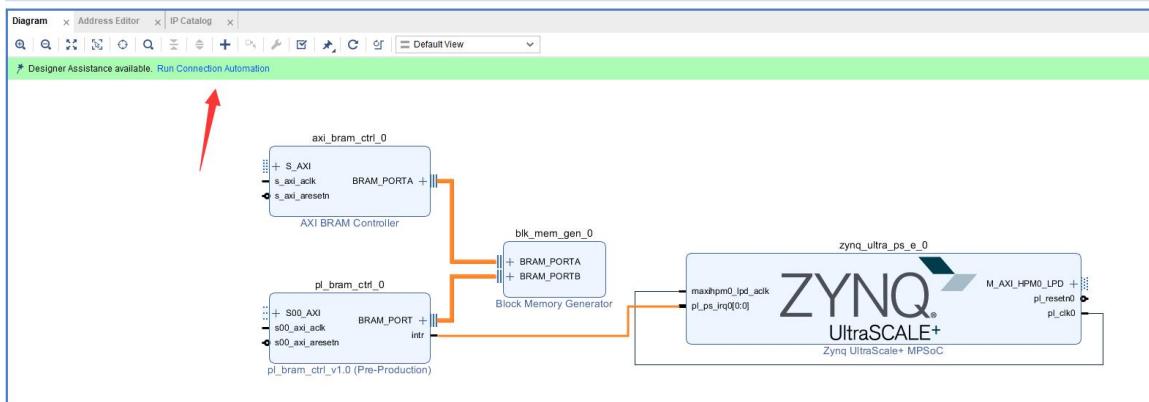


You can see that the newly added IP appears

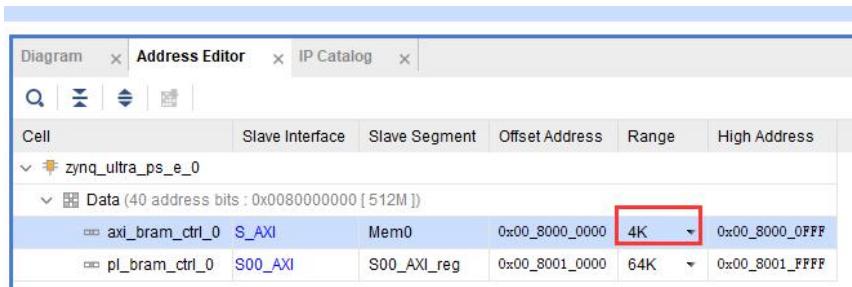


- 4) Connect BRAM_PORTA of AXI BRAM Controller to PORTA of BRAM, and connect BRAM_PORT of pl_bram_ctrl to PORTB of

BRAM. Connect the interrupt signal intr of the pl_bram_ctrl module to the interrupt port of ZYNQ. And click auto connect

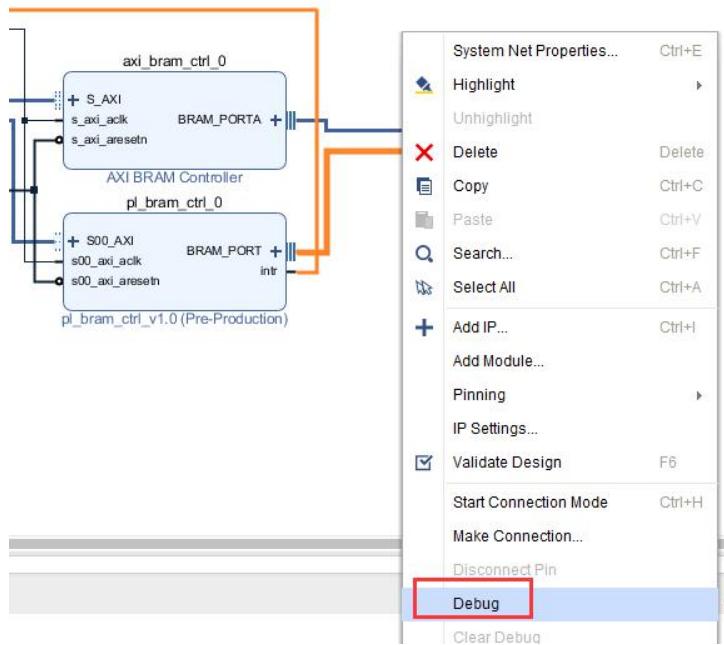


- 5) Select the BRAM address size in the Address Editor. If you set 4K, the address BRAM space is 1K deep.

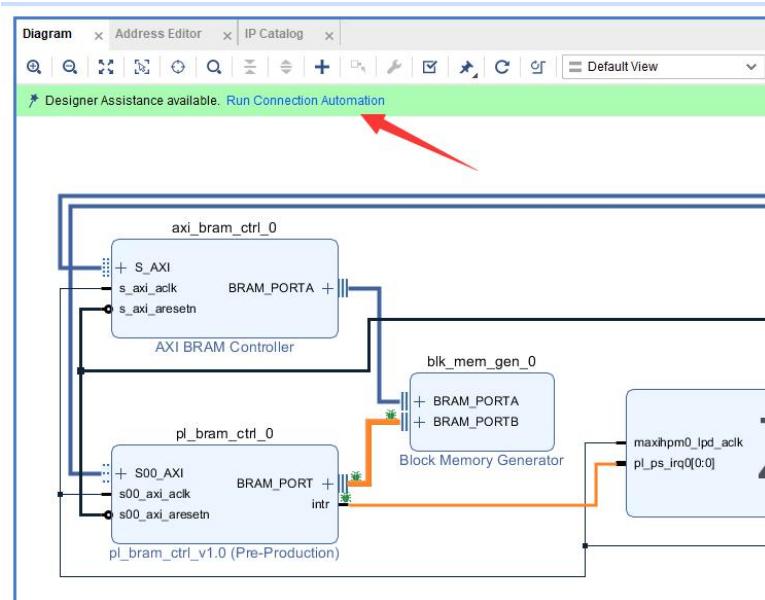


Part 20.1.1: Block Design adds logic analyzer method

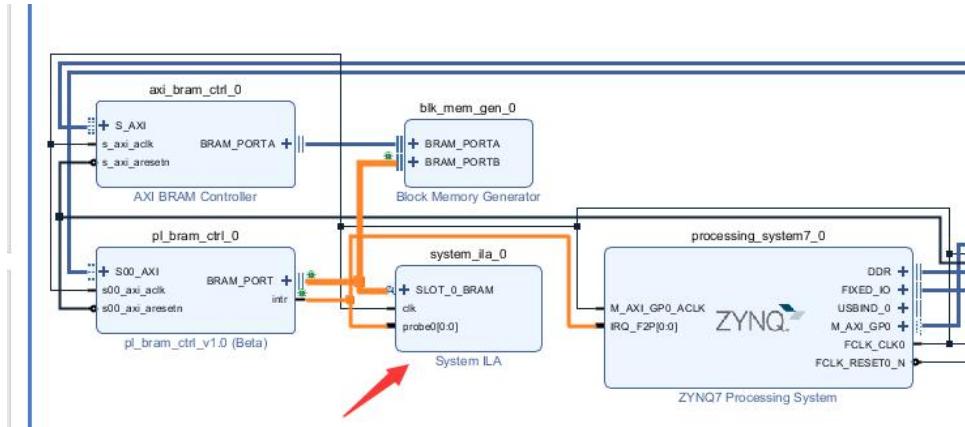
- 6) Then introduce a method to add a logic analyzer, select BRAM_PORT bus and intr interrupt, right click and select Debug



- 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) ;

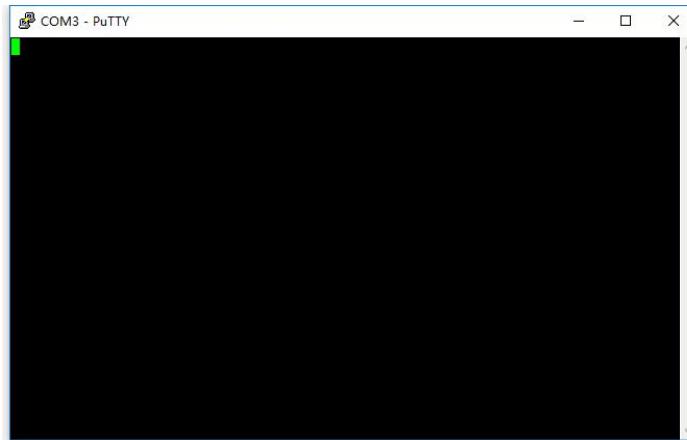
    return XST_SUCCESS ;
}

```

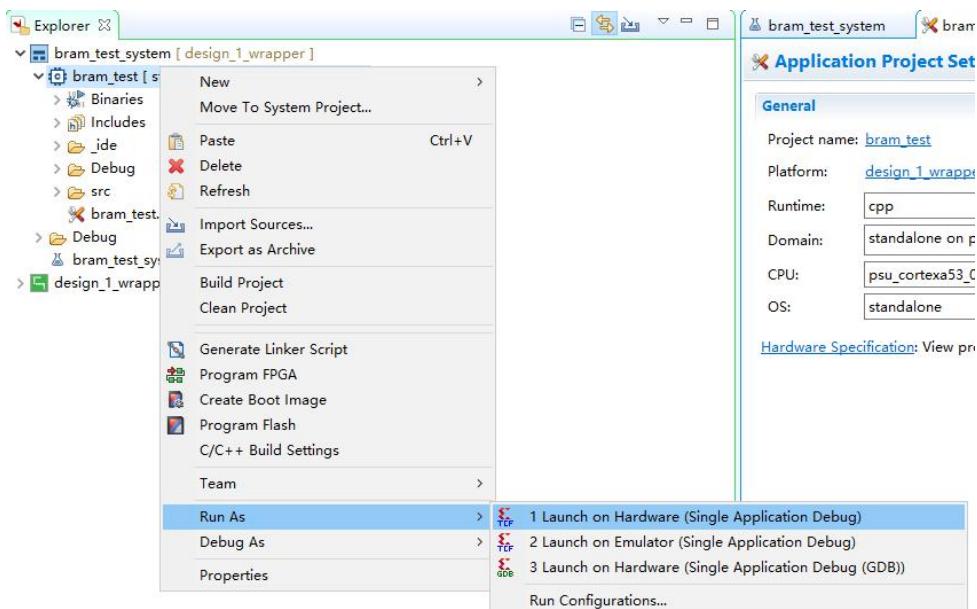
- 5) In the interrupt service routine, the “BRAM” controller reads the “BRAM” data and prints

Part 20.3: Experimental Result

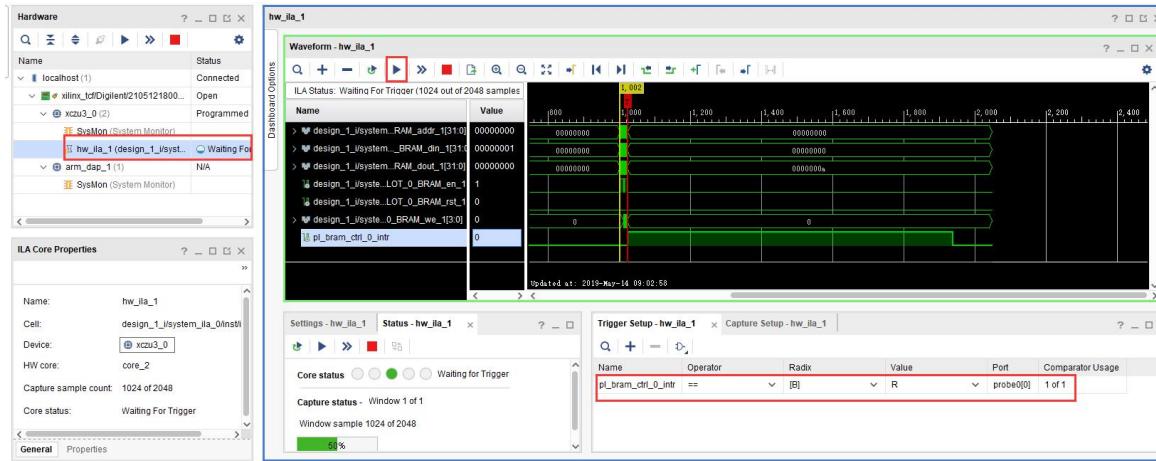
- 1) Open the serial port



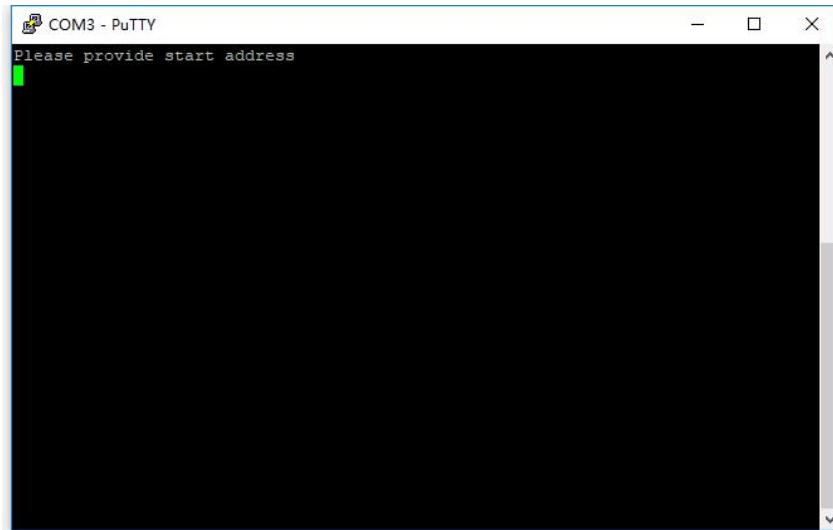
- 2) Download the program through Run Configurations



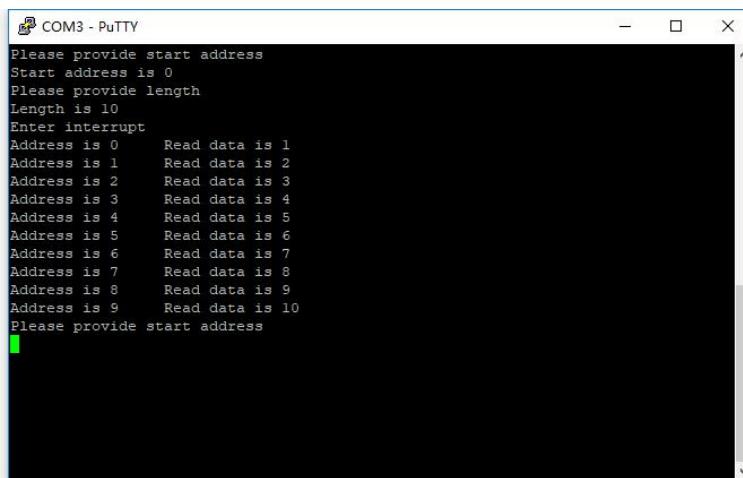
- 3) Open Vivado's Hardware Manager (joint debugging with PL), set the intr interrupt signal as the trigger signal, select the rising edge trigger, click the start button, you can see that hw ila_1 becomes the Waiting for trigger state



- 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.

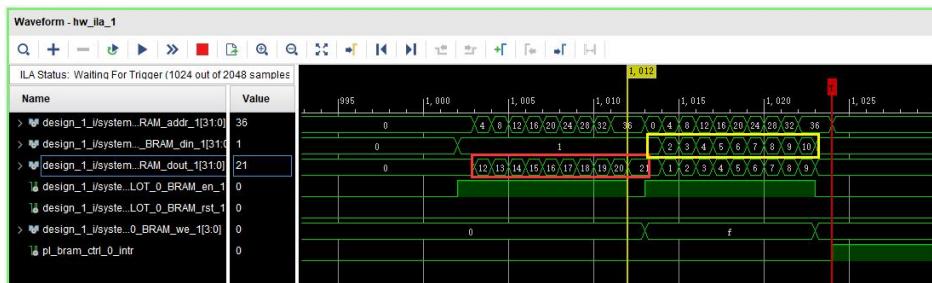


- 5) The input data is a decimal number, press Enter after the input

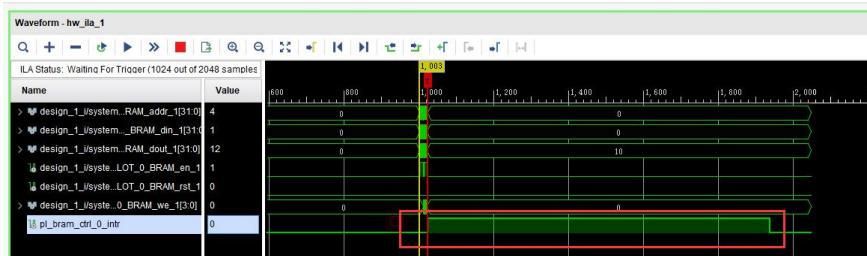


- 6) Open the ILA logic analyzer, you can see that it has been triggered.

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: Use VDMA to drive HDMI display

The experimental Vivado project directory is "vdma_hdmi_out /vivado".

The experiment vitis project directory is "vdma_hdmi_out /vitis".

The display control system can be realized with the help of PL. There are many implementation schemes, but they are all inseparable from the DMA system. The DMA system can read data from the ddr3 to the display and reduce the overhead of the CPU. VDMA is a special DMA developed by xilinx, dedicated to video input and output, and is an important part of learning xilinx FPGA video processing.

The previous HDMI display data is generated internally by the PL. In this experiment, the display data is generated by the PS, and then the PL is transmit to the HDMI interface through the VDMA.

FPGA Engineer Job Content

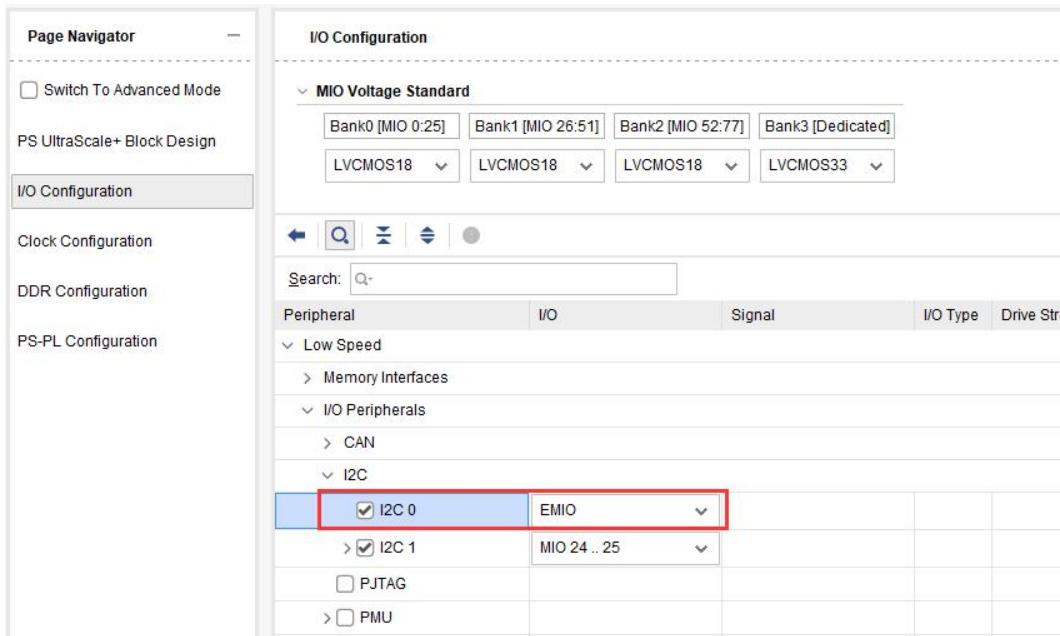
The following is the content that FPGA engineers are responsible for.

Part 21.1: Create a Vivado Project

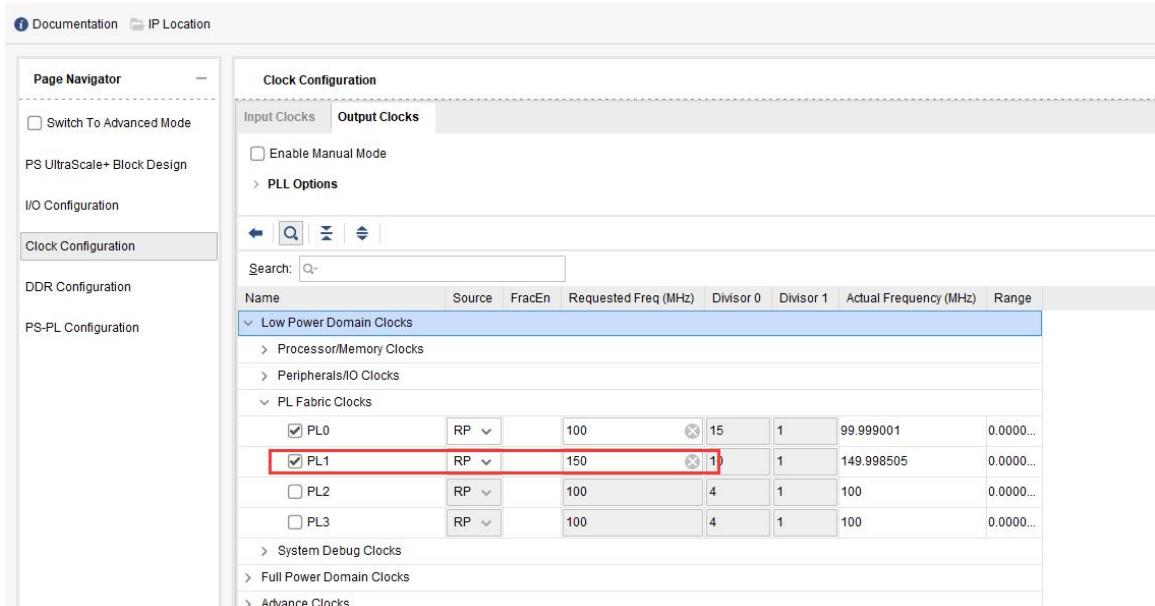
Since VDMA display is a very important content, this experiment will introduce the Vivado construction process in detail.

1) Use "ps_hello" to save as a project named "vdma_hdmi_out".

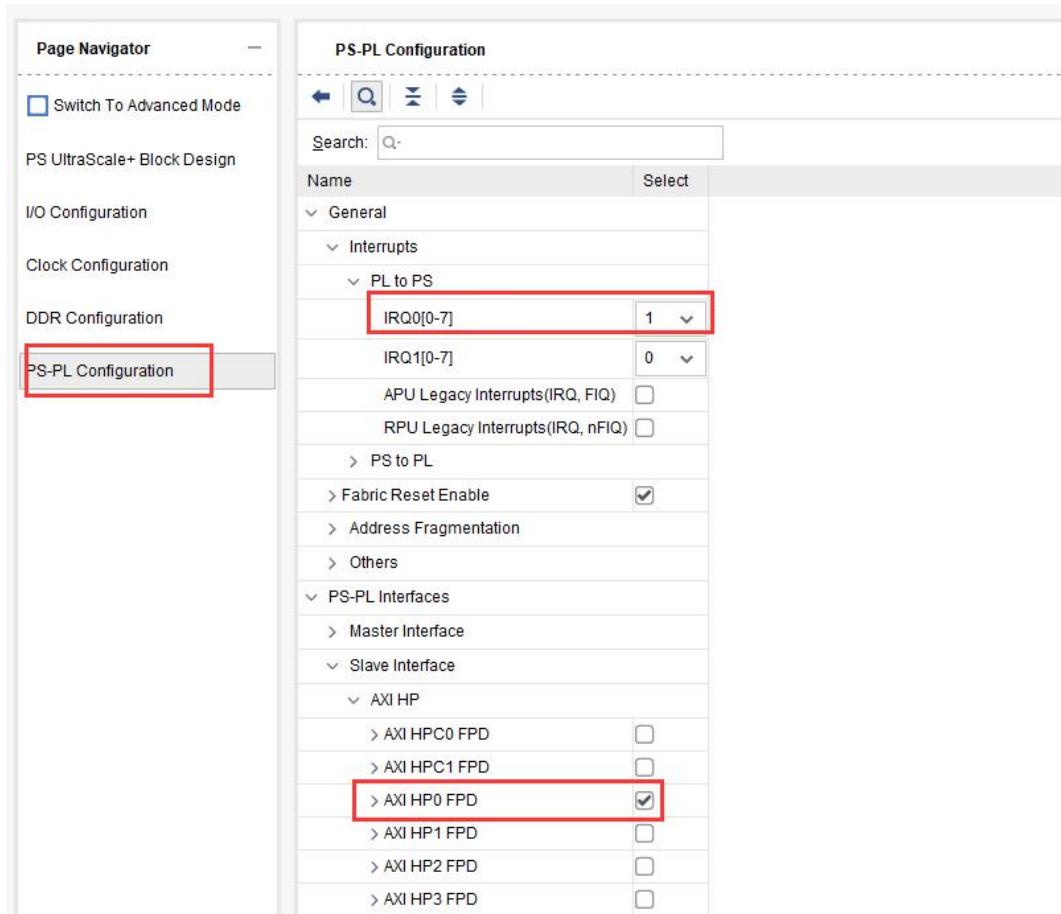
Configure ZYNQ parameters, configure I2C EMIO, enable I2C0, and select EMIO, so that I2C can be connected to the PL end for configuring HDMI chips.



- 2) Configure the clock, FCLK_CLK0 is configured as 100Mhz, FCLK_CLK1 is configured as 150Mhz, this clock is used for VDMA to read data. This is because the frequency of 1080p 60 frames is 148.5MHz, but includes synchronization and blanking time, and VDMA transmits all valid data, so setting it to 150MHz can meet the requirement.

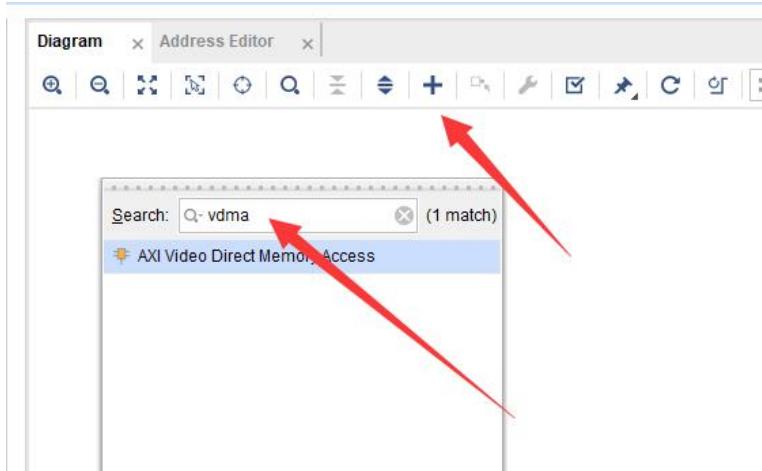


- 3) Configure interrupts, enable IRQ_F2P, receive interrupts from PL, and enable HP0 interface for VDMA to quickly read ddr.



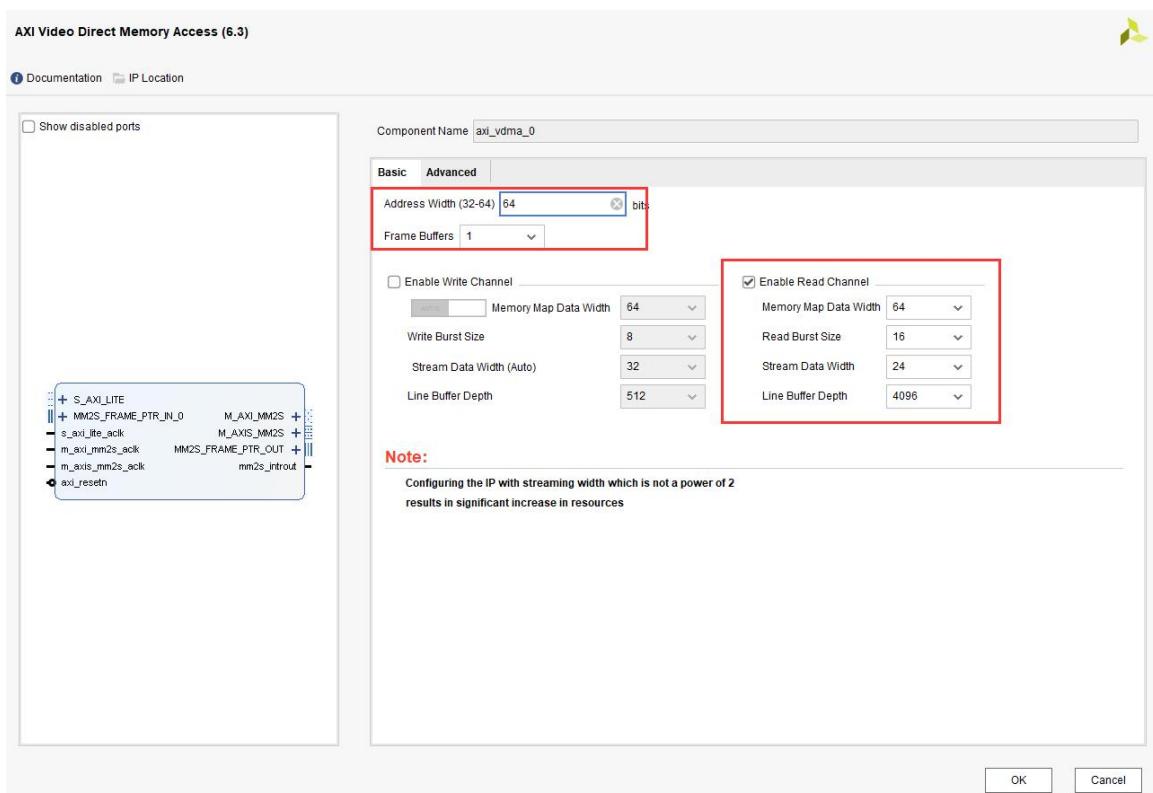
Part 21.1.1: Configure VDMA

4) Add VDMA IP



- 5) Configure the basic parameters of **VDMA** according to the following figure. There are mainly two interfaces involved, the **Memory Map** interface, which uses the **AXI4** interface to interact

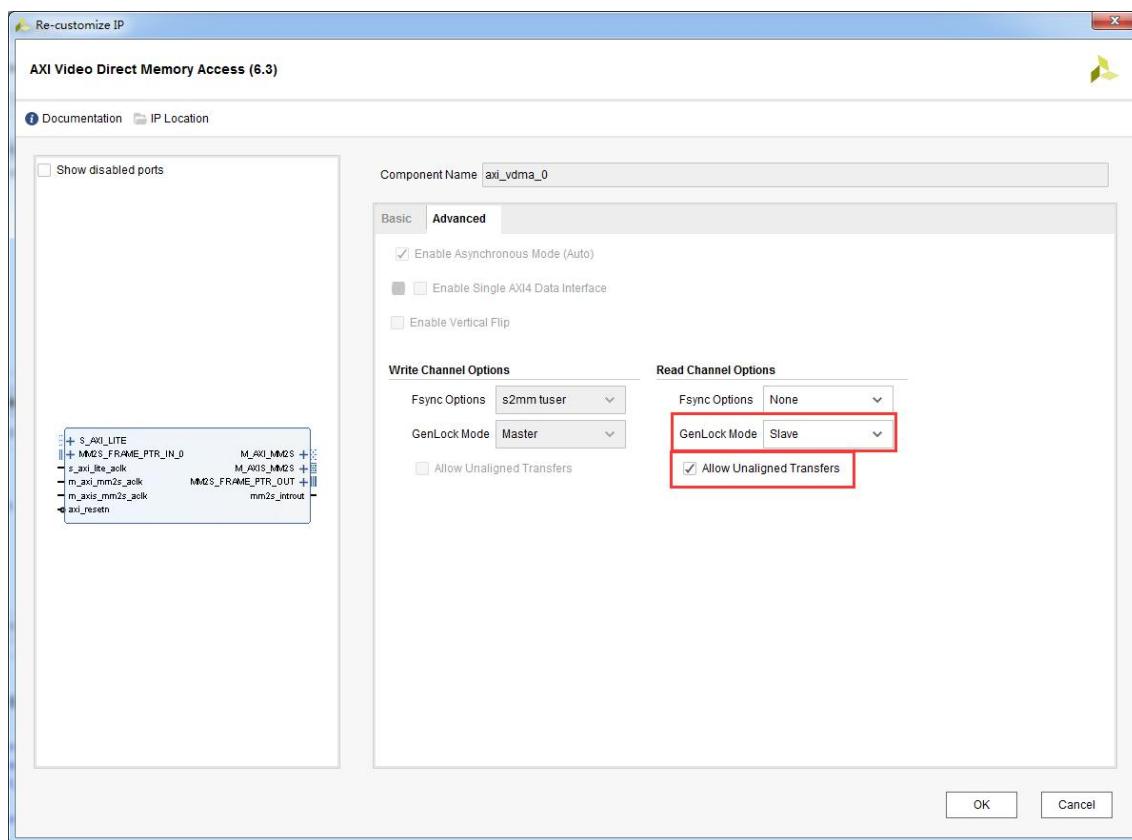
with the **ZYNQ HP** port and read the image data in the **DDR** on the **PS** side. The **ZYNQ HP** interface is a 64-bit interface. Here we also set it to a 64-bit interface. Of course, it can also be set larger. The data width can be automatically converted through the cross-interconnect module. The other interface is the **Stream** interface, which is the **AXI4 stream** interface. Here, it is mainly used to transmit image data to the **HDMI** interface. Since the **RGB** data is **24** bits, the **Stream Data Width** here is also set to **24**. **Frame Buffers** is the number of frame buffers, which can store multiple frames of images. In this experiment, only **1** frame of image buffer is enabled. The **Line Buffer Depth** is similar to the **fifo** cache, with **Stream Data Width** as the unit, the larger the setting, the more data can be cached.



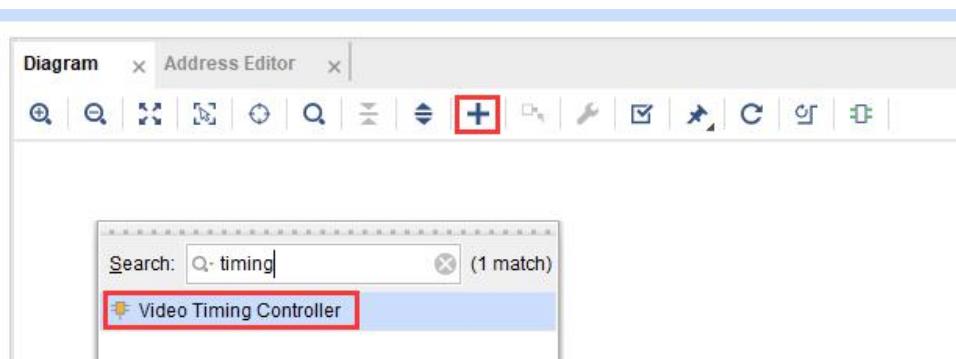
- 6) Configure the **VDMA** advanced parameters, enable **Allow Unaligned Transfers** here, if not enabled, the data must be aligned in the software according to the **Memory Map Data Width**. For

example, we set 64, which means 64-bit alignment. But if it is enabled here, unaligned data transmission can be carried out.

GenLock is used to prevent the read and write channels from accessing the same frame at the same time and make the image display abnormal. Since we only have one read channel, setting it does not make much sense. It is only useful if it needs to be configured with the write channel. There are many combinations. For details, please refer to the manual PG020 of VDMA.

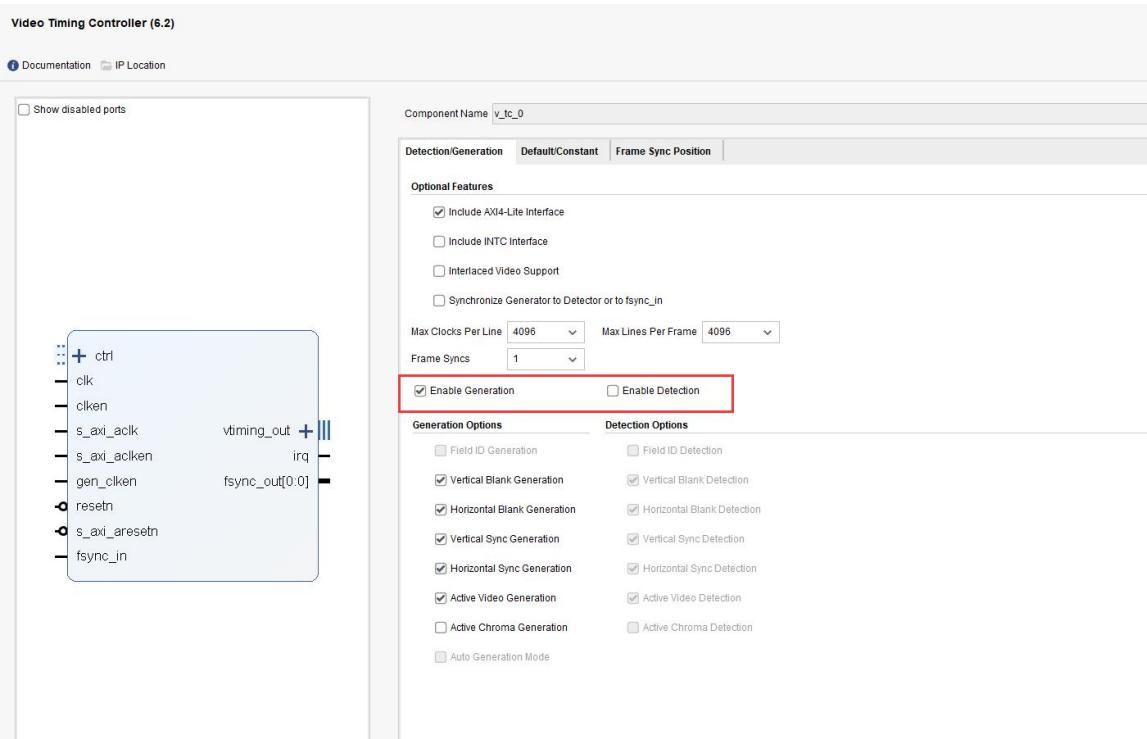


- 7) Add video timing controller, this module is mainly used to generate image timing.

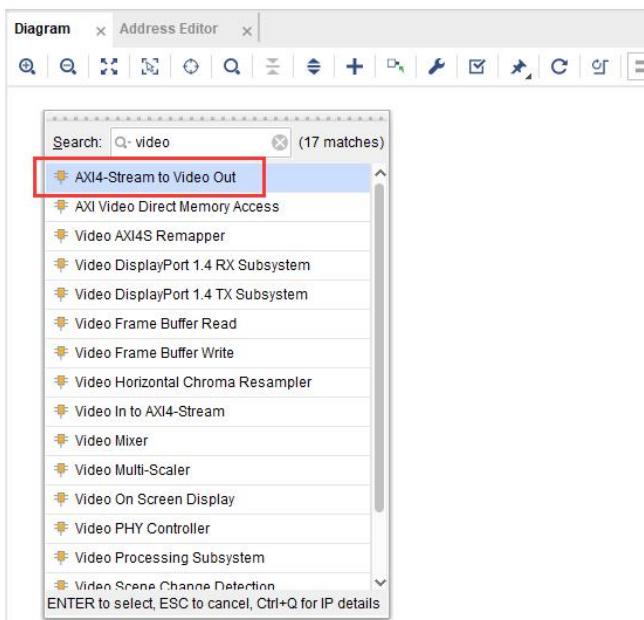


8) Configure the video timing controller parameters (VTC for short).

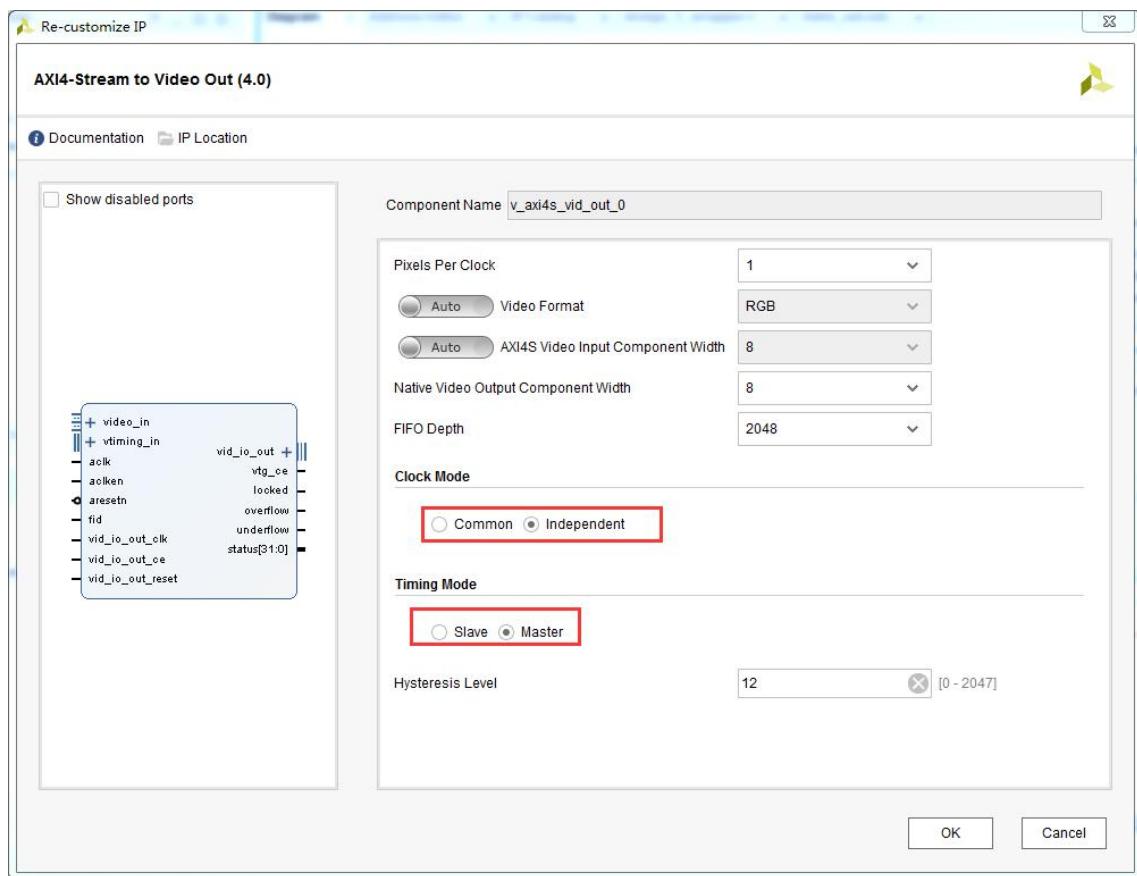
Enable Generation is the enable for generating output timing. After selection, the **vtiming_out** bus signal will appear. **Enable Detection** is used to detect the enable of the input timing signal. If it is enabled, the **vtiming_in** bus will appear. Since this experiment is for image output, it is not enabled.



9) Add AXI streaming video output controller



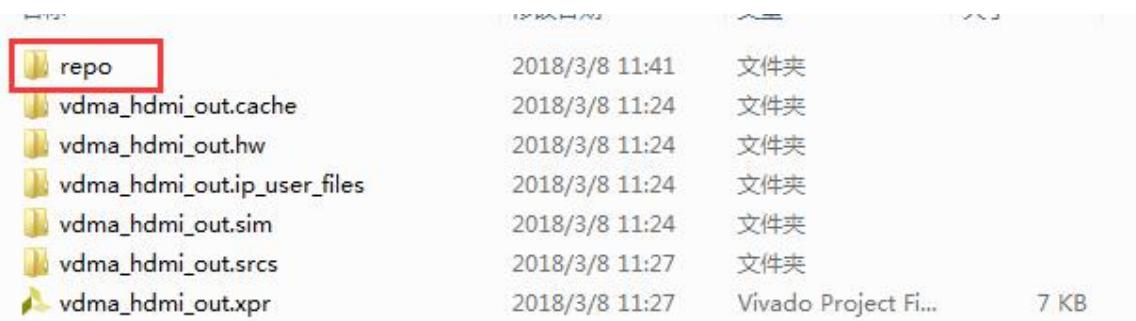
10) Configure the AXI streaming video output controller parameters, and select Independent for Clock Mode, which means that the clocks of AXI4-Stream and Video are independent and asynchronous, while common is synchronous. In this experiment, the two are asynchronous. The Slave mode of Timing Mode means that VTC is a time sequence Slave, and the Video Out module controls the output of the time sequence through clock enable. Master Mode means that VTC is the master of timing and is not controlled by Video Out. For details, refer to the module user manual pg044.



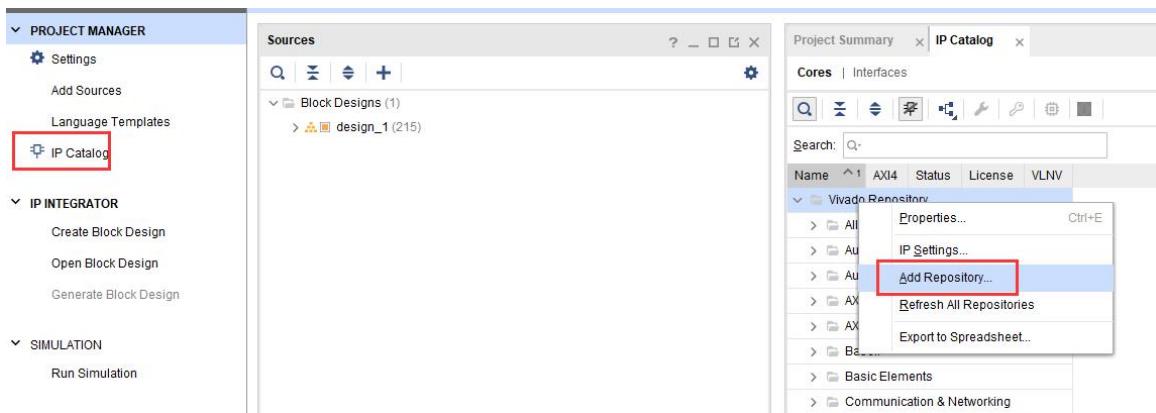
Part 21.1.2: Add custom IP

11) 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

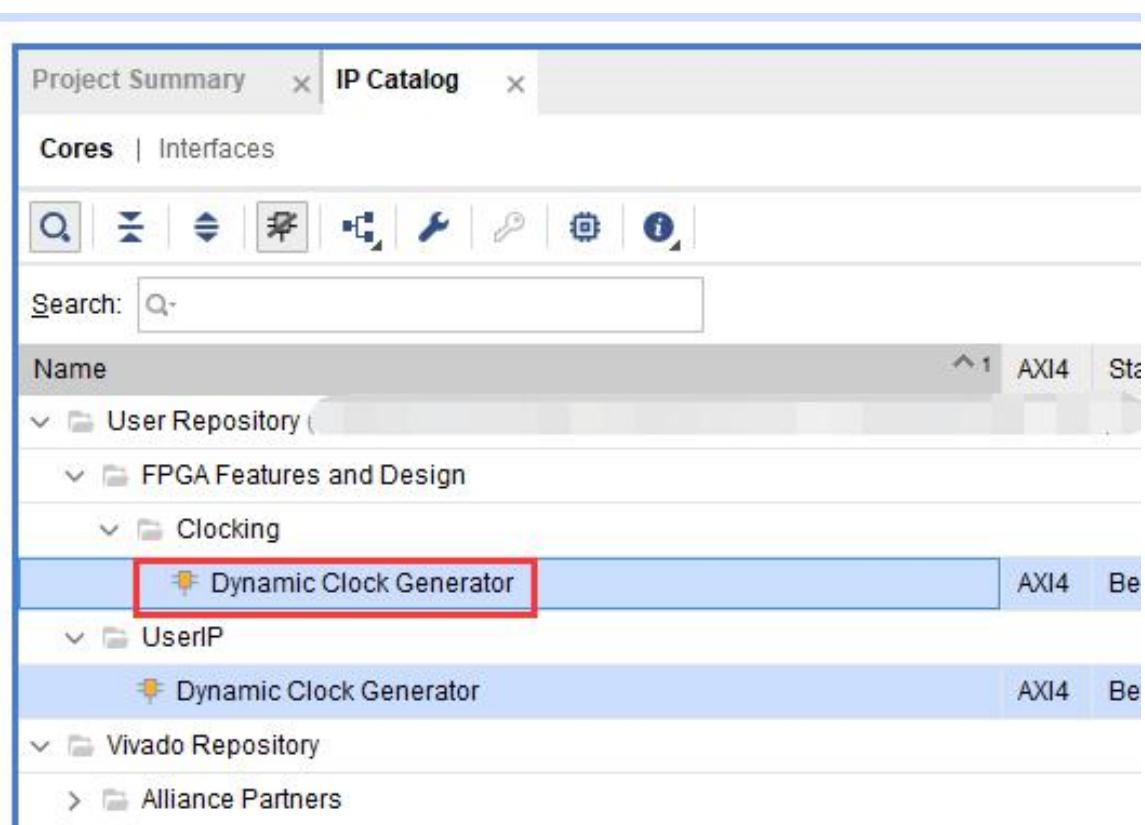
directory in the routine, and copy it to your own directory



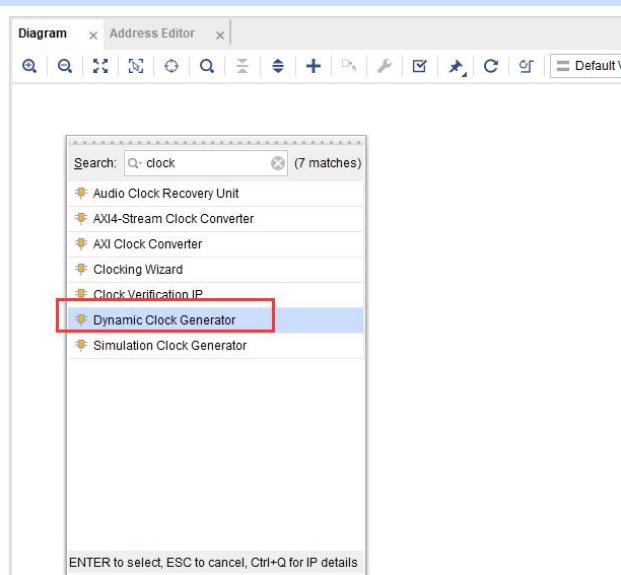
12) Add IP Library



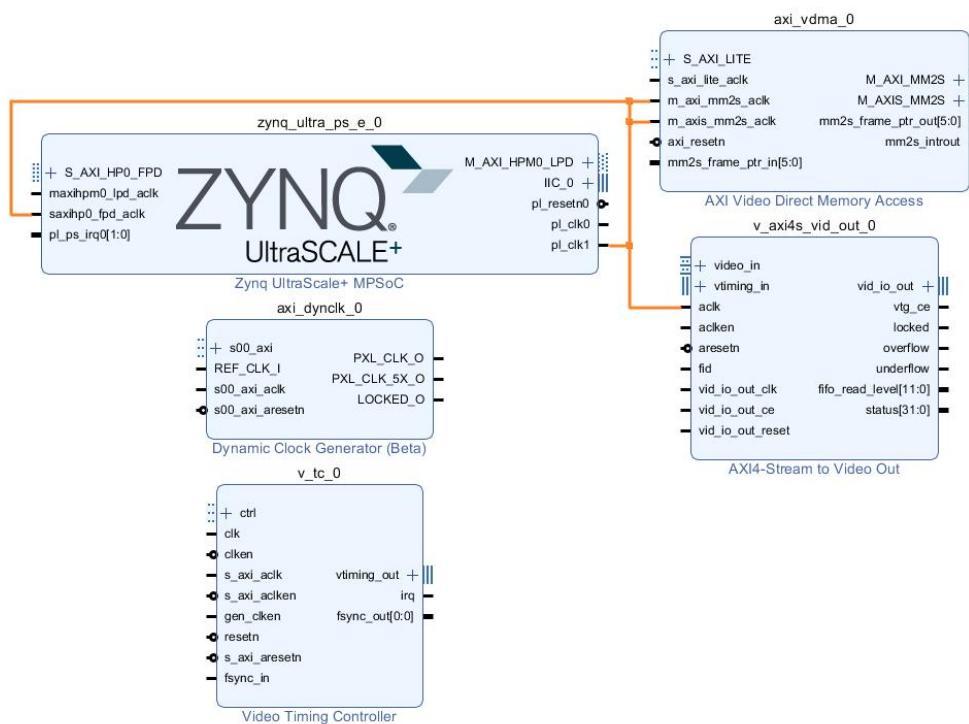
13) After the addition is complete, you can see the IP of the dynamic clock



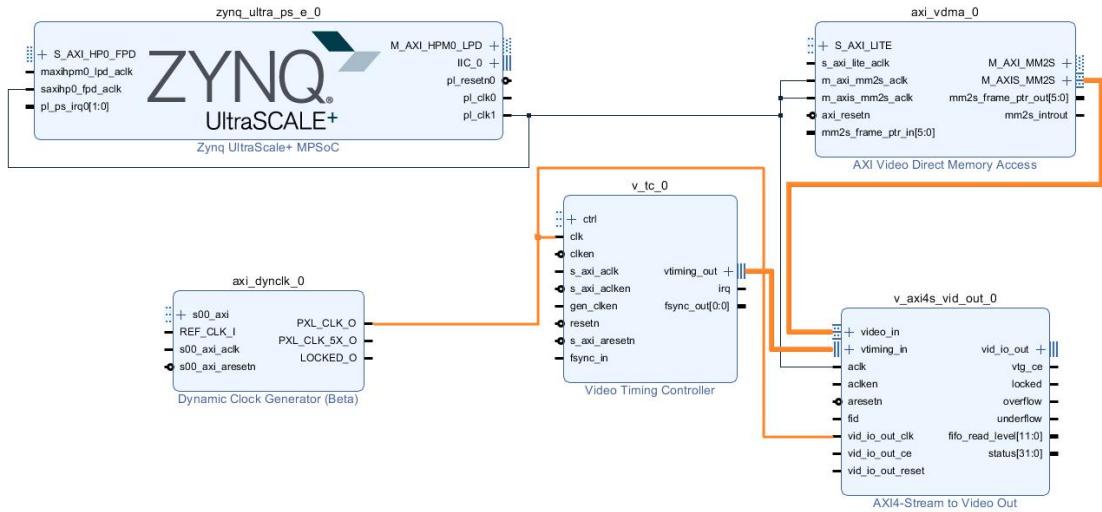
14) Add a dynamic clock controller. The main function of this module is to configure different clock outputs according to different resolutions. In essence, it calls a phase-locked loop, but it should be noted that the reference clock of this module must be set to 100MHz



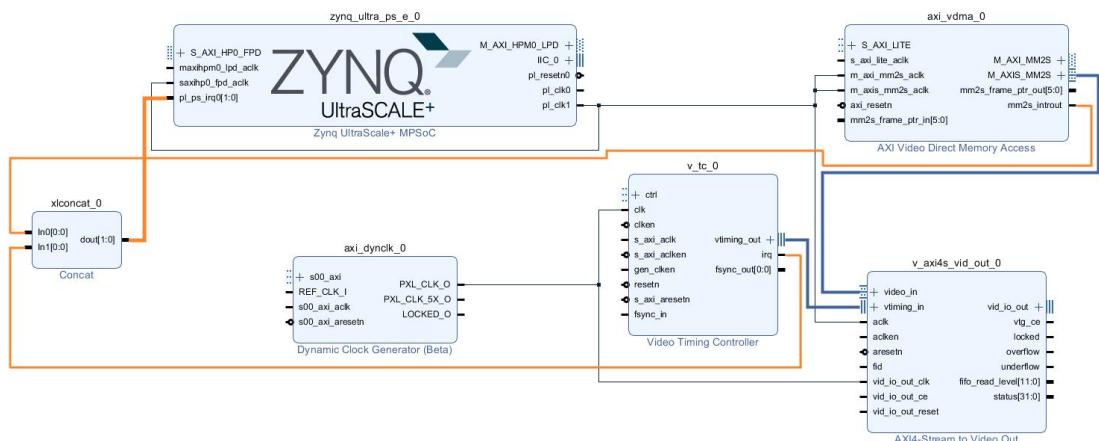
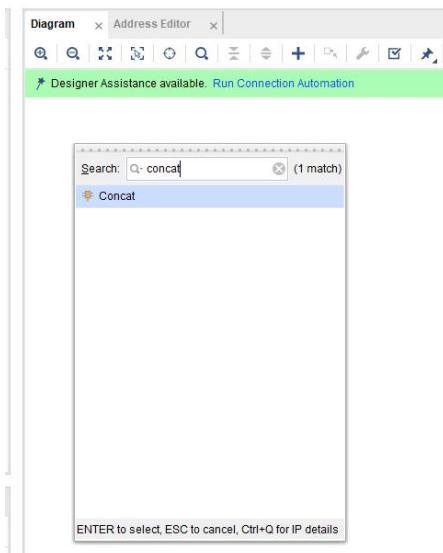
15) Connect to clock signals that Vivado may not be able to connect automatically



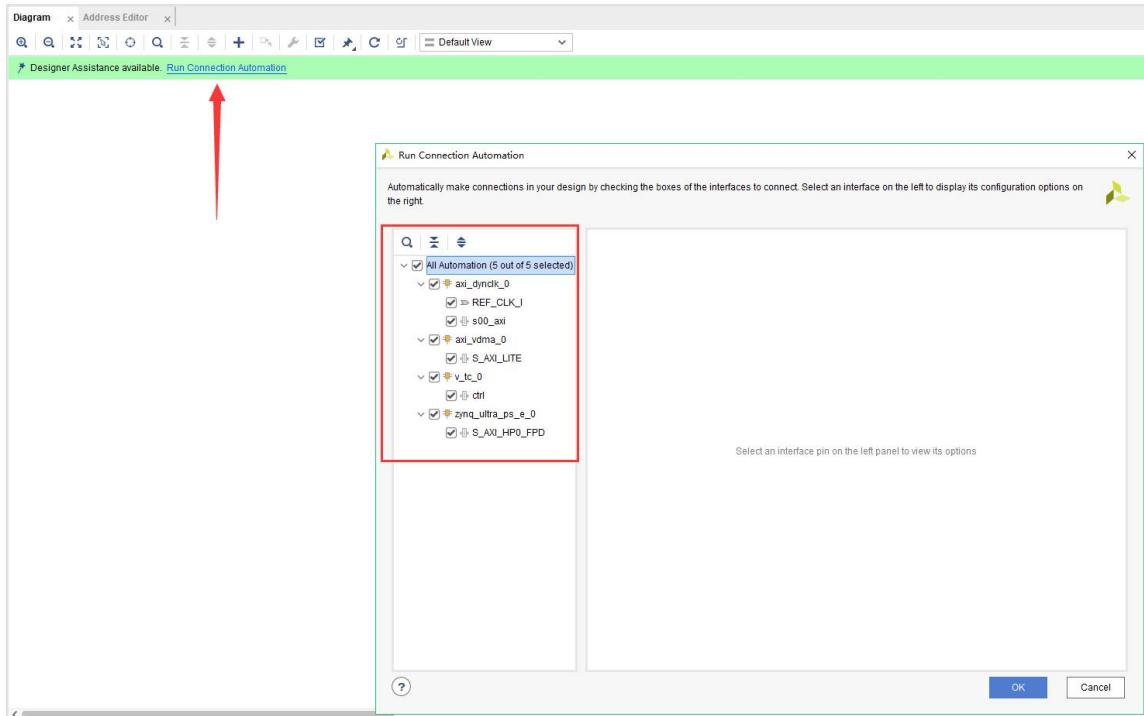
16) Connect some other key signals



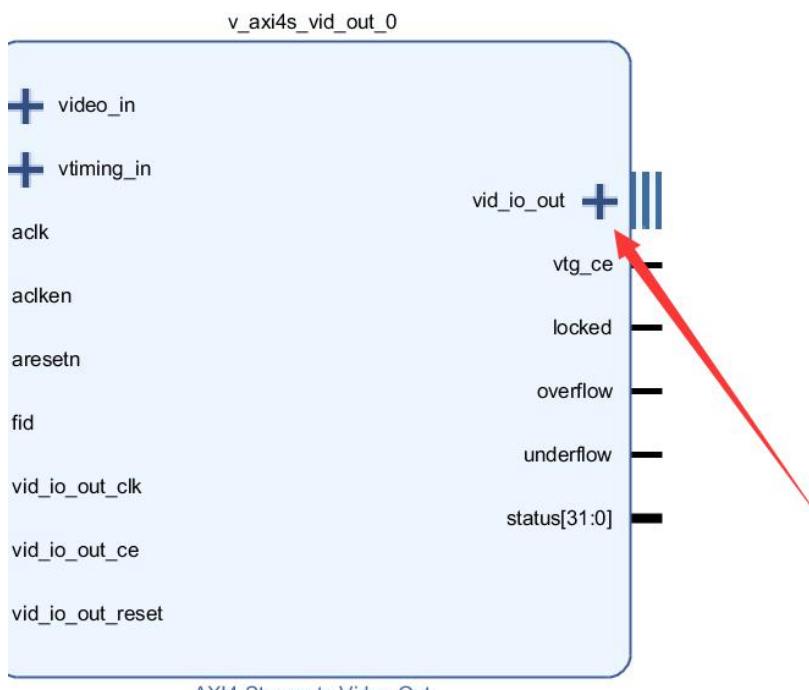
17) Connection interrupt signal, you need to add a **Concat IP** for signal connection



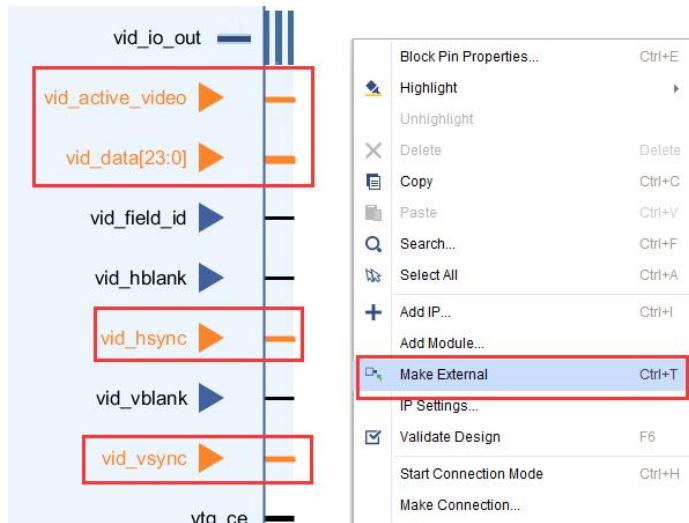
18) Use the Vivado automatic connection function to complete the remaining line connections and select all modules to automatically connect



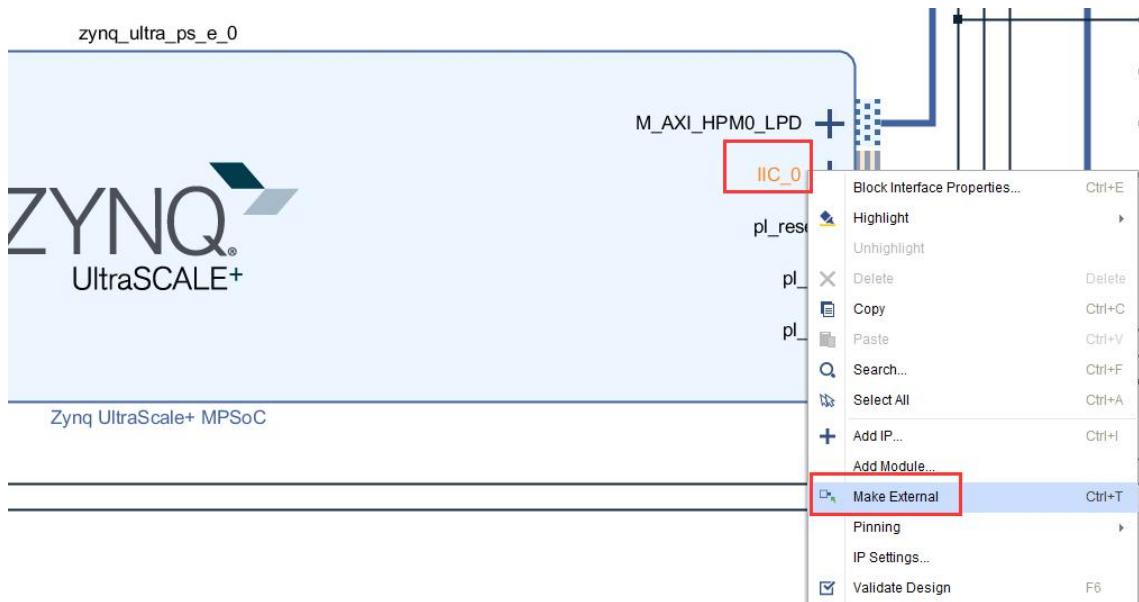
19) Expand `vid_io_out` port



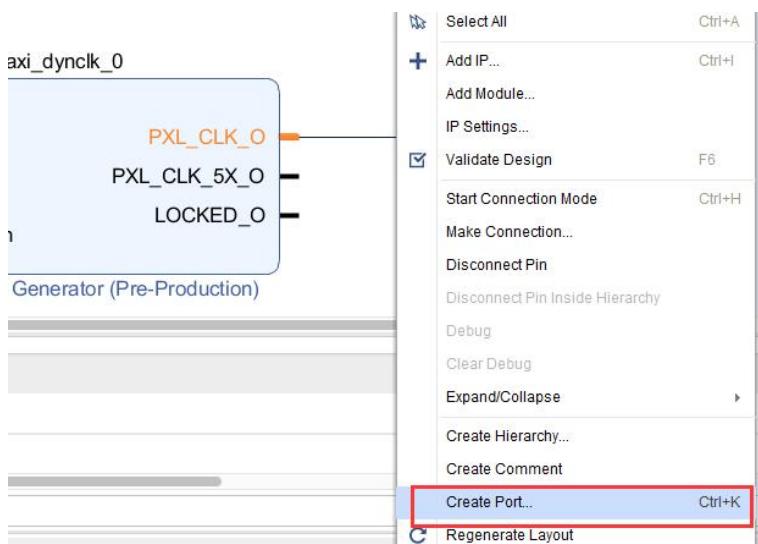
20) Choose the port we need to export



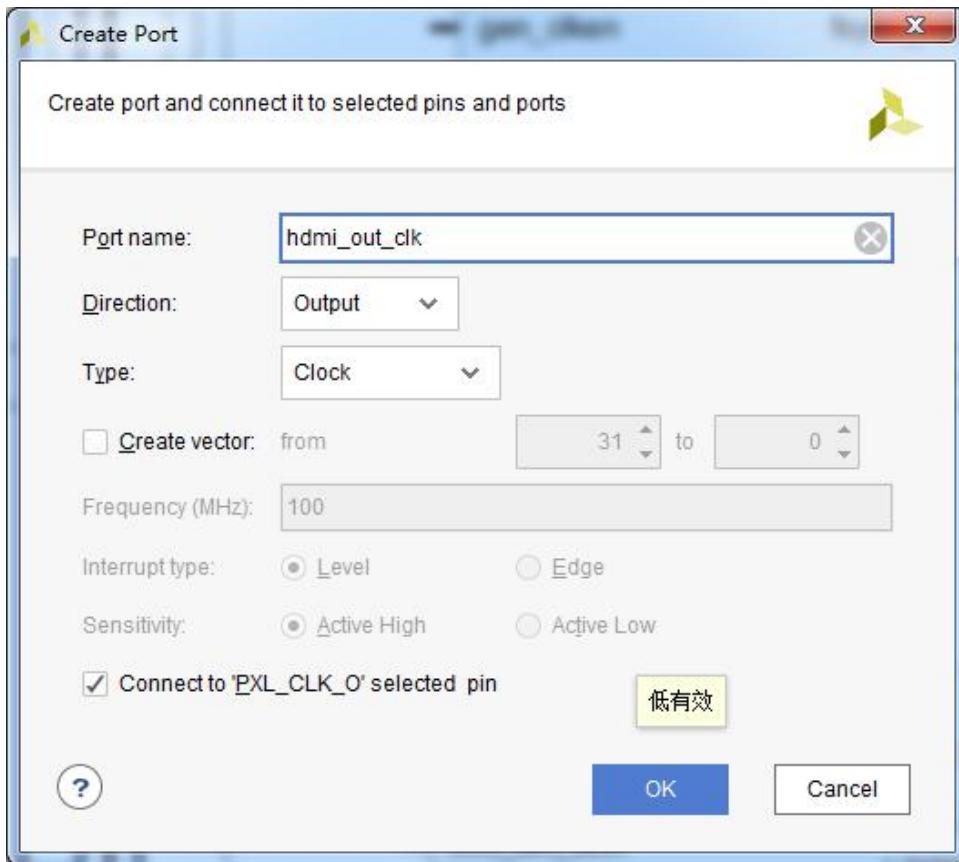
21) Export IIC_0 port



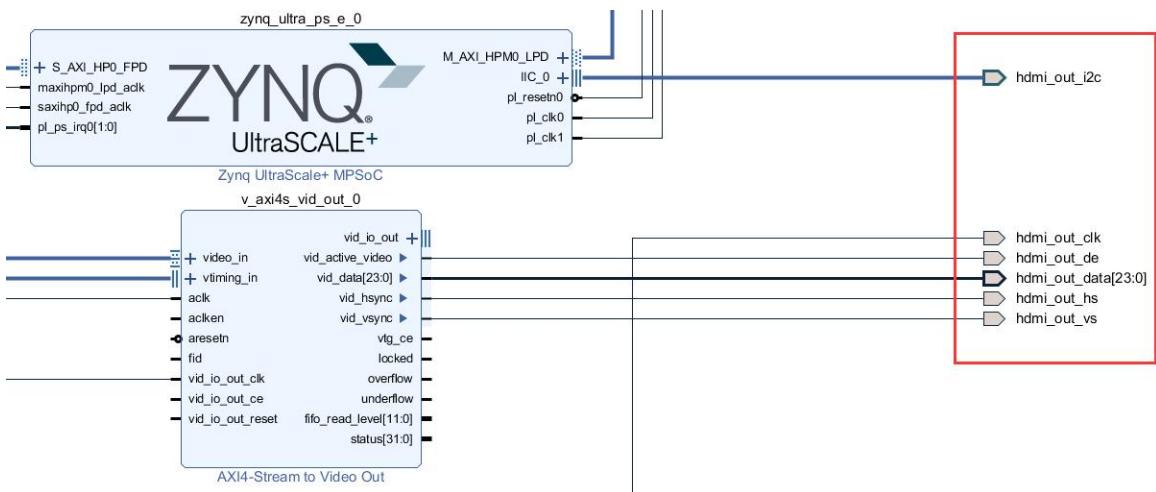
22) Export video clock port



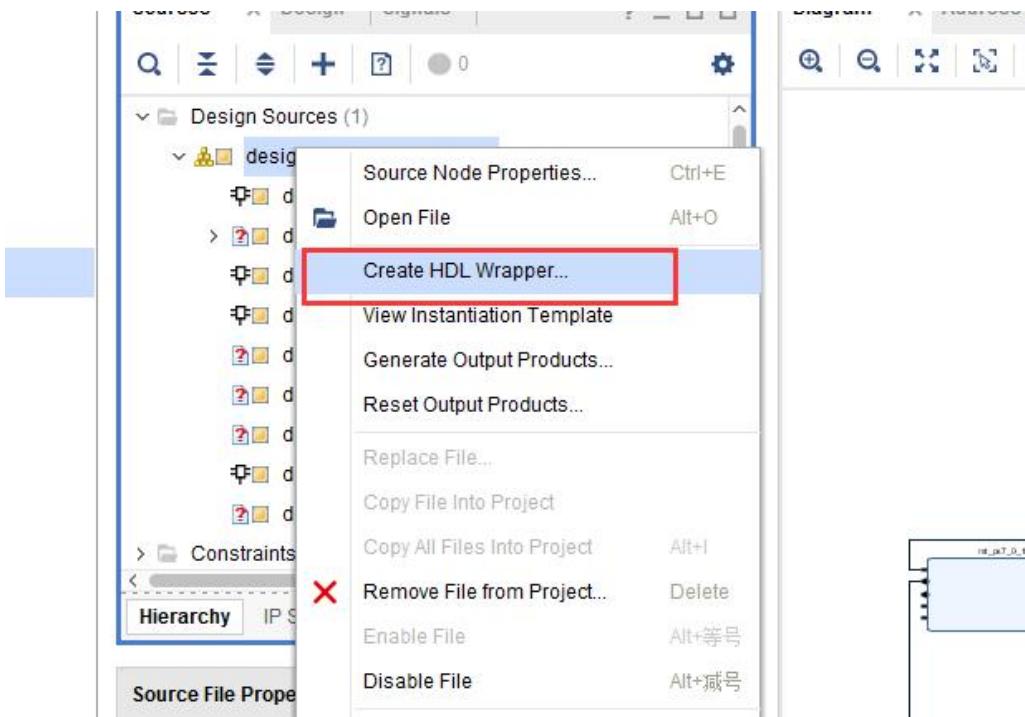
23) Change the name to `hdmi_out_clk`



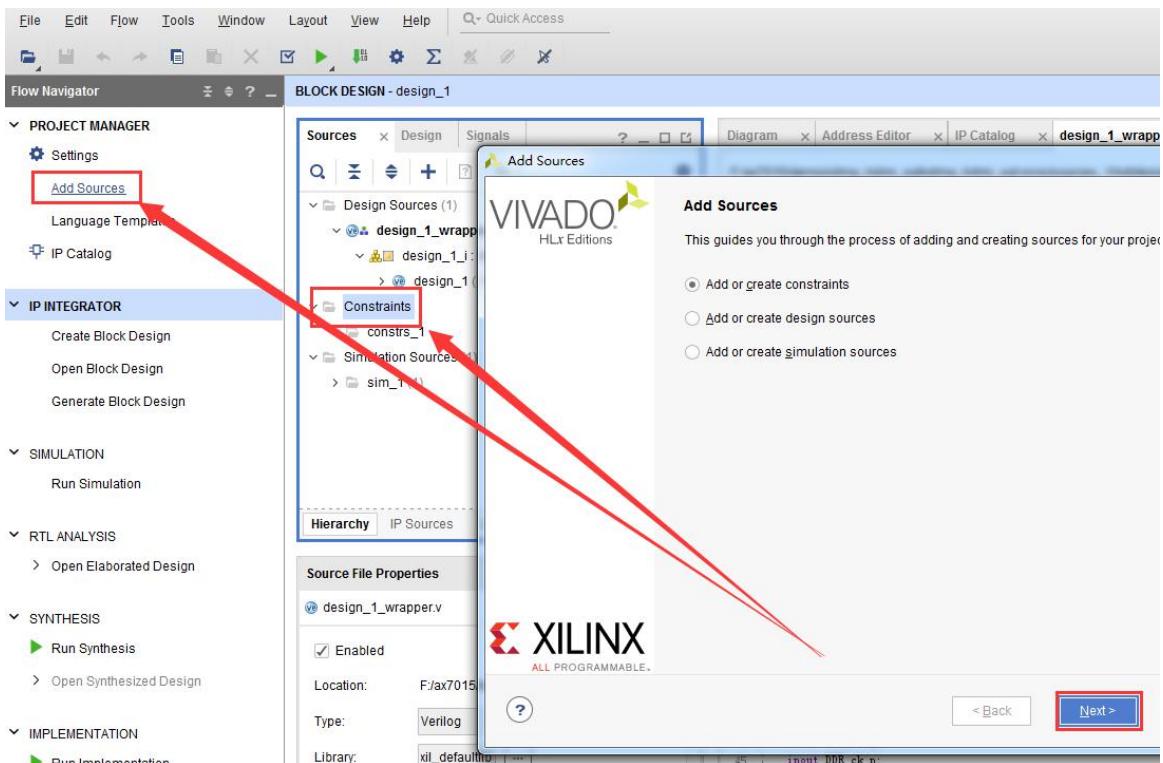
24) Modify the names of other ports



25) After saving the design, press F6 to check the design and create the HDL file if there is no problem



26) Add the `xdc` file of HDMI output and bind the pins



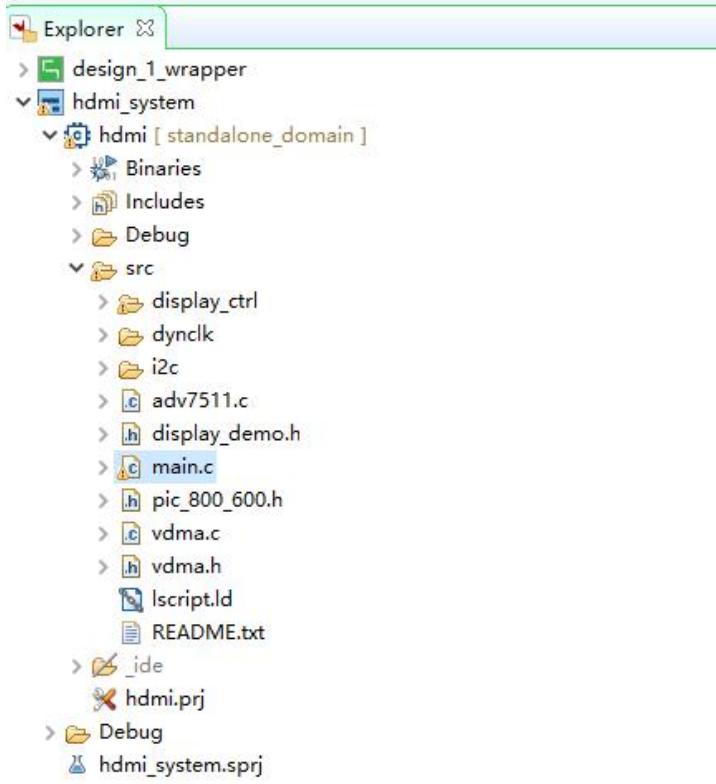
27) Compile and generate bit file, export hardware

Software Engineer Job Content

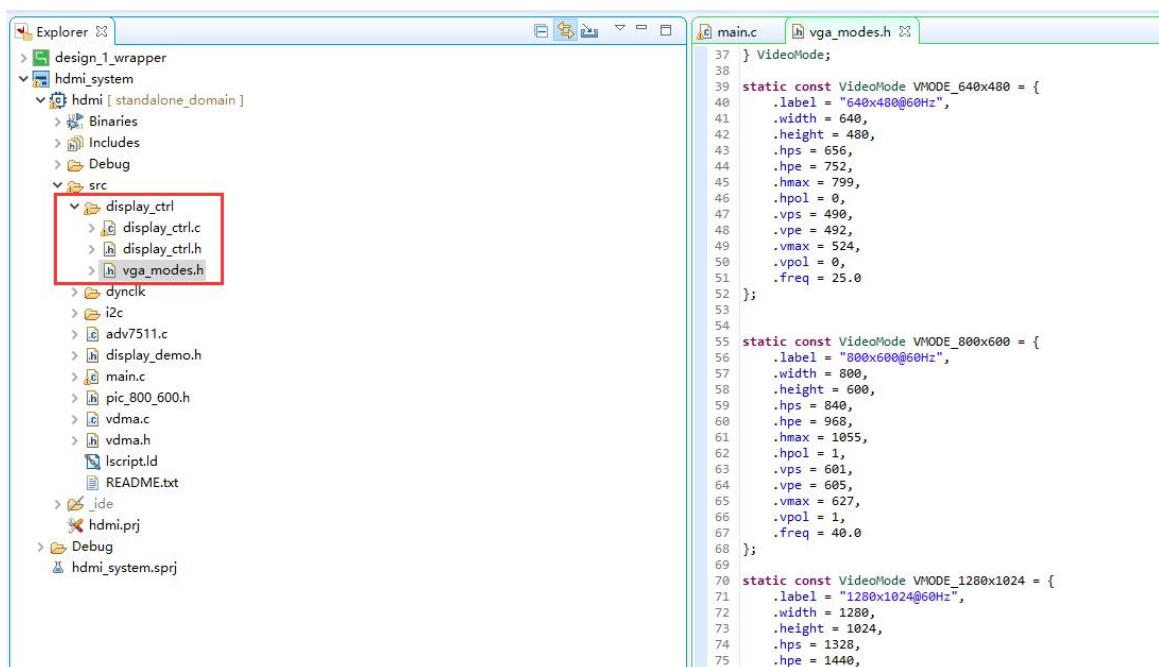
The following is the content that FPGA engineers are responsible for.

Part 21.2: Vitis Software Writing and Debugging

- 1) Create a new app named vdma_hdmi



- 2) In the **display_ctrl** folder, **display_ctrl.c** is mainly for display control, Some timing parameters for display resolution have been added to **vga_mode.h**.



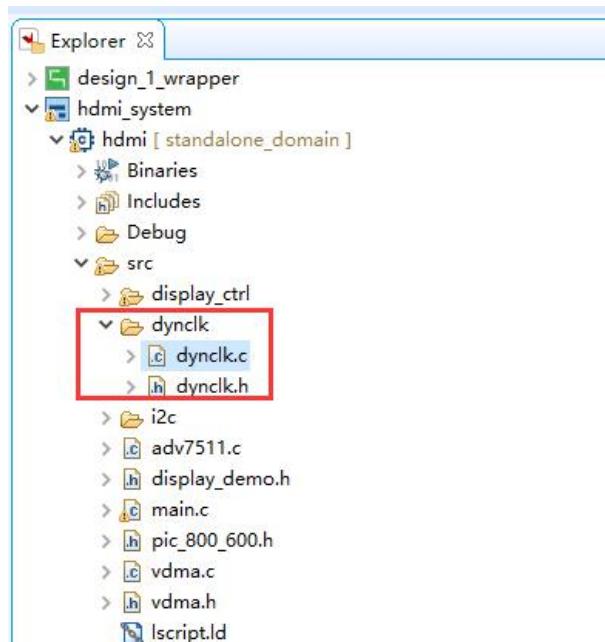
In `display_ctrl.c`, you can modify `displayPtr->vMode` to change the display resolution.

```

310 /**
311 */
312 int DisplayInitialize(DisplayCtrl *dispPtr, XAxiVdma *vdma, u16 vtcId, u32 dynClkAddr, u8
313 {
314     int Status;
315     int i;
316     XVtc_Config *vtcConfig;
317     ClkConfig clkReg;
318     ClkMode clkMode;
319
320     /*
321     * Initialize all the fields in the DisplayCtrl struct
322     */
323     dispPtr->curFrame = 0;
324     dispPtr->dynClkAddr = dynClkAddr;
325     for (i = 0; i < DISPLAY_NUM_FRAMES; i++)
326     {
327         dispPtr->framePtr[i] = framePtr[i];
328     }
329     dispPtr->state = DISPLAY_STOPPED;
330     dispPtr->stride = stride;
331     dispPtr->vMode = VMODE_800x600;
332
333     ClkFindParams(dispPtr->vMode.freq, &clkMode);
334
335 }

```

- 3) In the `Dynclk` file, the main function is to configure the clock output of the phase-locked loop according to different resolutions to generate the pixel clock.



- 4) There is a concept that needs to be clarified. Generally we know that images have the concept of rows and columns. In the VDMA registers, HSIZE and VSIZE, there is an extra STRIDE register, which can be understood as the maximum number of bytes stored

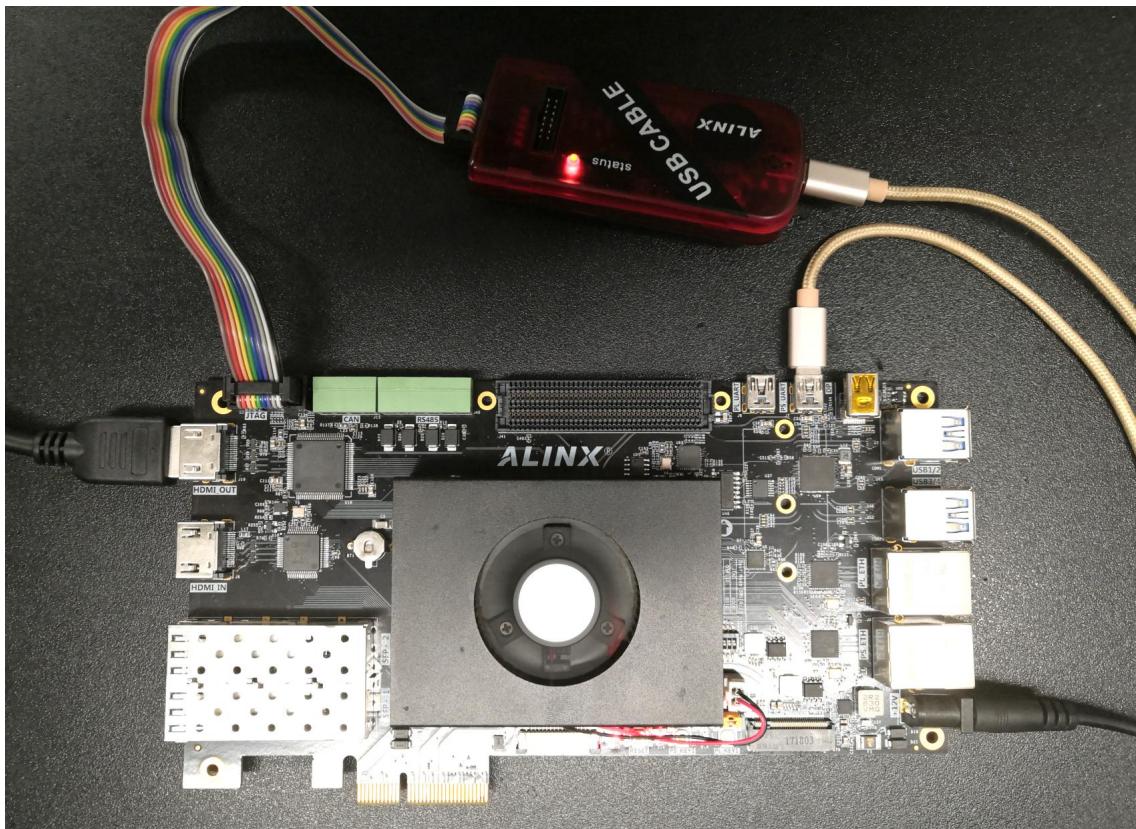
in a row, which is greater than Equal to HSIZE. Note that both HSIZE and STRIDE are in bytes.

50h	MM2S_VSIZE	MM2S Vertical Size Register
54h	MM2S_HSIZE	MM2S Horizontal Size Register
58h	MM2S_FRMDLY_STRIDE	MM2S Frame Delay and Stride Register

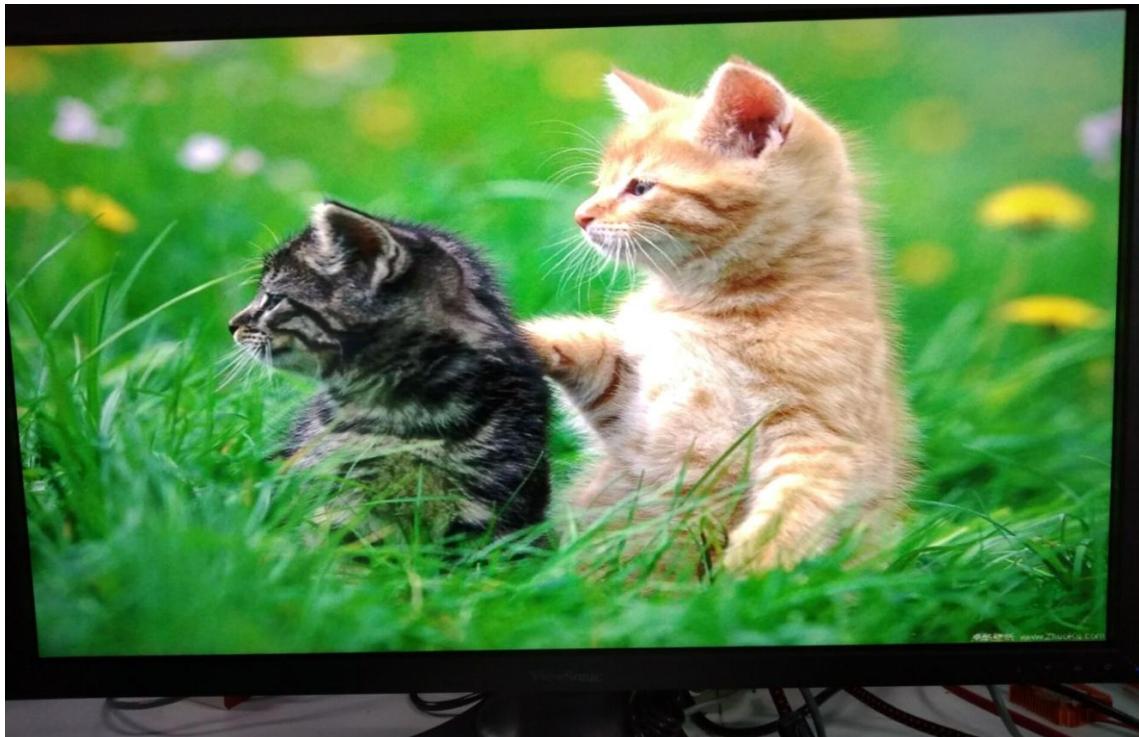
For example: if the display resolution is 1920*1080, 24-bit depth, which is 3 bytes, then HSIZE can be set to 1920*3, VSIZE is 1080, and STRIDE is 1920*3; if the display resolution is changed to 1280 *720, then HSIZE is set to 1280*3, VSIZE is 720, STRIDE does not need to be changed, it is still 1920*3.

Part 21.3: Onboard Verification

The connection board is as follows:



The monitor shows a picture



Part 21.4: Experimental Summary

This part introduces the use of VDMA to realize HDMI display. There are many modules used. You need to look at the documentation of each module, especially VDMA, VTC, Video Out, and digest slowly. This experiment is the basis of the experiment shown later, and it is necessary to spend more time learning.

Part 22: Use VDMA to Drive HDMI Acquisition and Display

The experimental Vivado project directory is "vdma_hdmi_inout /vivado".

The experimental vitis project directory is "vdma_hdmi_inout /vitis".

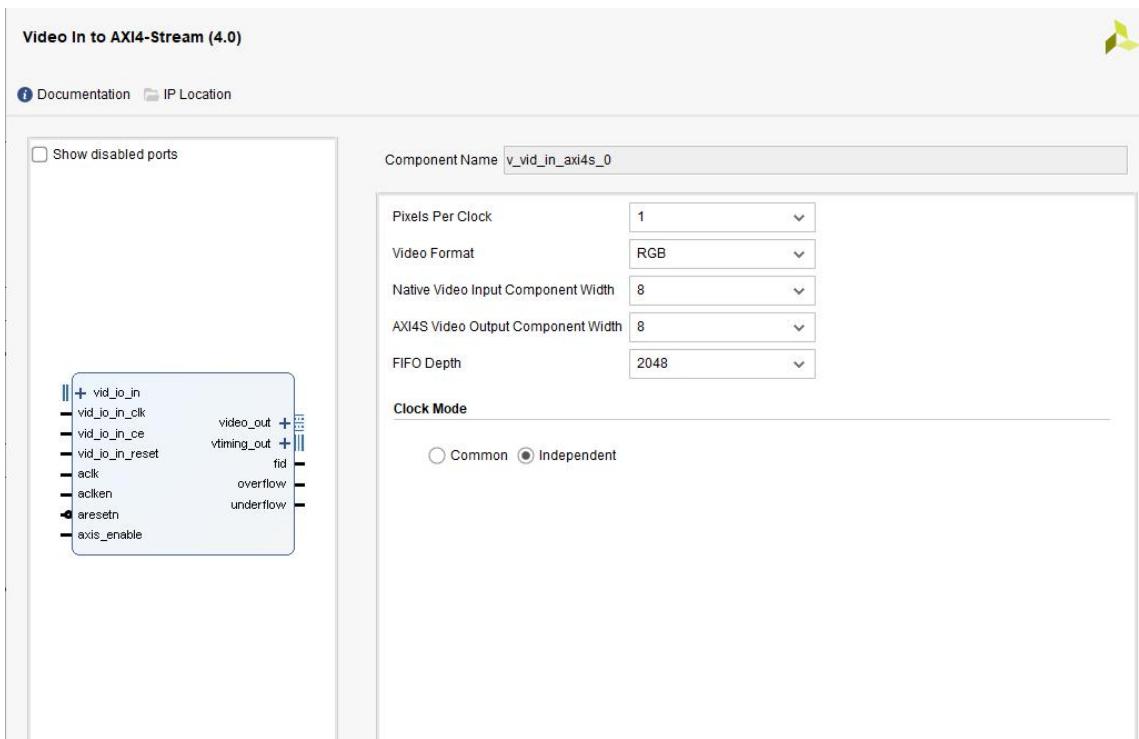
The previous chapter introduced the use of VDMA to drive the HDMI display. This chapter introduces the use of VDMA to drive the HDMI input and transmit the input image to the displayed VDMA for HDMI display.

FPGA Engineer Job Content

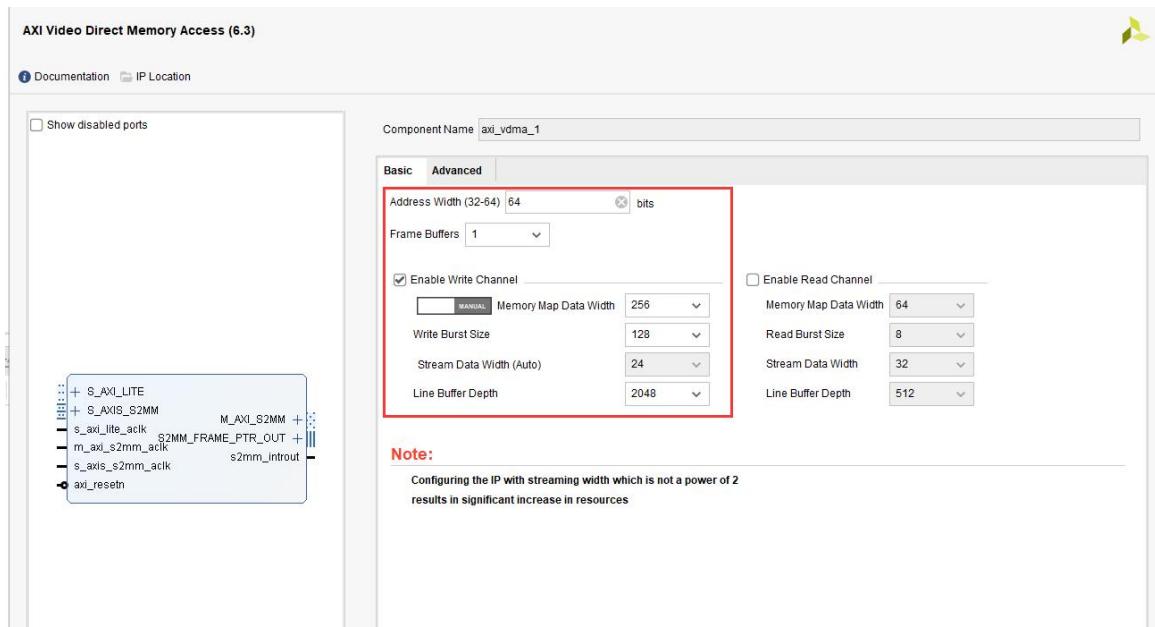
The following is the content that FPGA engineers are responsible for.

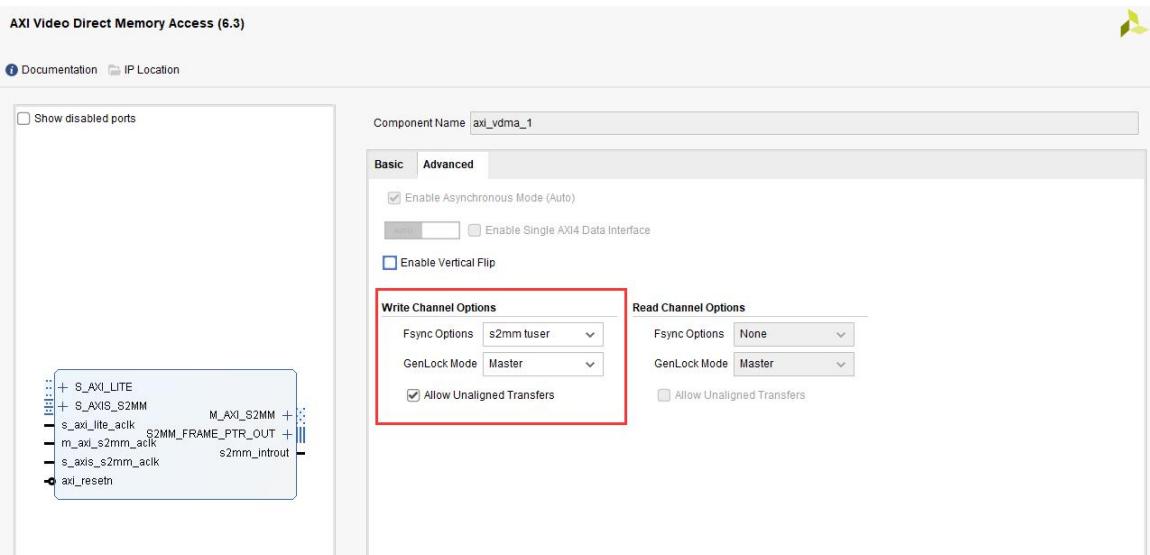
Part 22.1: Create a Vivado Project

- 1) On the basis of the project in the previous chapter, add the HDMI capture part, first add the video capture module [Video in to AXI4-Stream](#), the configuration is as follows

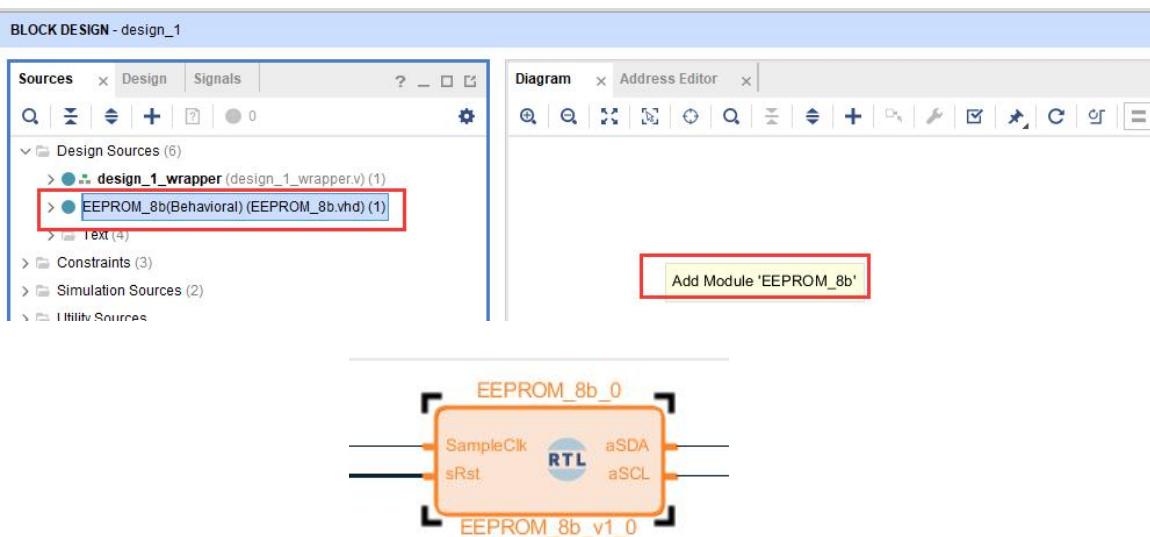


2) Add VDMA module, the configuration is as follows

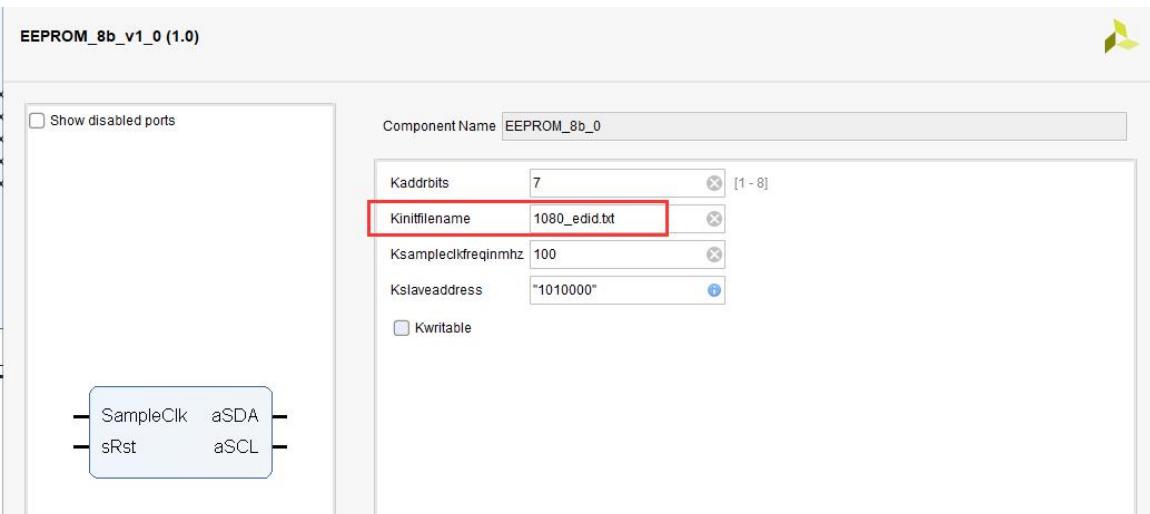




- 3) Add **EDID EEPROM** module, because it is an **RTL** file, you can drag it to **Block Design** with the left button



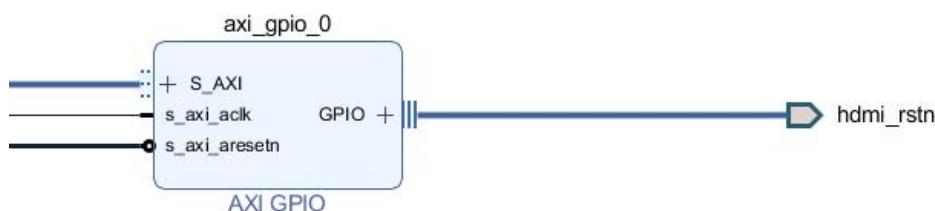
And the configuration is initialized to 1080p



- 4) Add a constant module, **constant**, and configure it as constant 1, which is high level, and connect to **hpd**



- 5) Add a 1-bit **gpio** module for resetting **HDMI** input



- 6) Export **IIC_1** as **EMIO** for the configuration of **HDMI** input chip



- 7) For how to connect the newly added modules, you can refer to the vivado project of the routine

Software Engineer Job Content

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

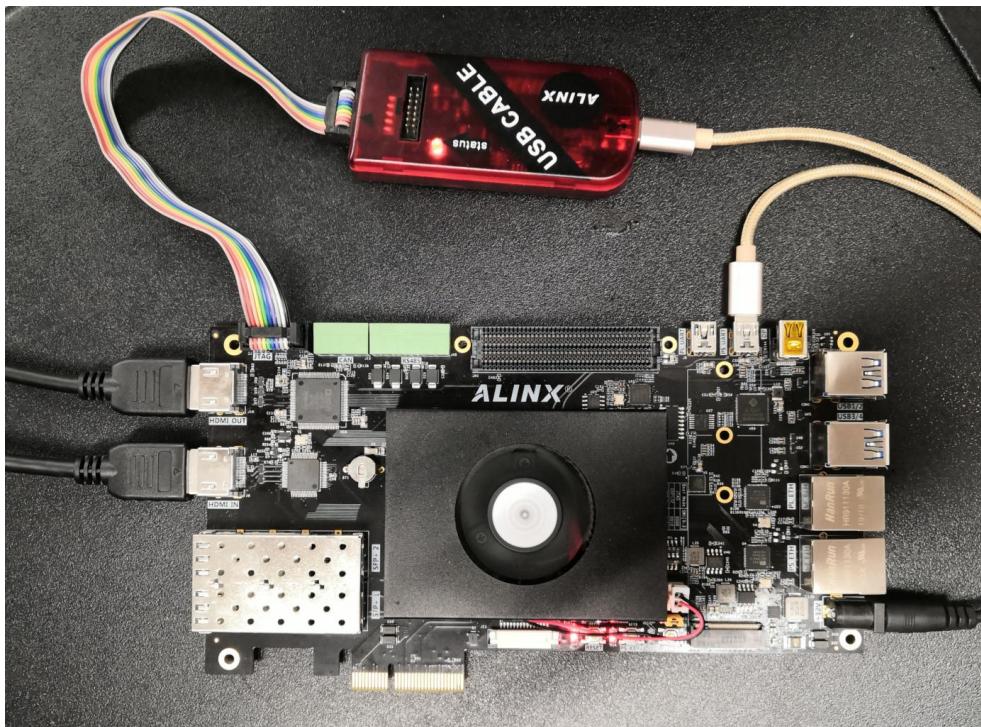
Part 22.2: Vitis Software Writing and Debugging

Create a new APP project, add the initialization of ADV7611 and the initialization of the acquisition VDMA on the basis of the previous chapter

```
main.c
87
88
89     adv7611_init(IicInstance1) ;
90
91
92     /*
93      * Initialize VDMA driver
94      */
95     vdmaConfig = XAxiVdma_LookupConfig(VGA_VDMA_ID);
96     if (!vdmaConfig) {
97         xil_printf("No video DMA found for ID %d\r\n", VGA_VDMA_ID);
98     }
99     Status = XAxiVdma_CfgInitialize(&vdma, vdmaConfig, vdmaConfig->BaseAddress);
100    if (Status != XST_SUCCESS) {
101        xil_printf("VDMA Configuration Initialization failed %d\r\n", Status);
102    }
103
104    /*
105     * Initialize the Display controller and start it
106     */
107    Status = DisplayInitialize(&dispCtrl, &vdma, DISP_VTC_ID, DYNCLK_BASEADDR,
108        pFrames, DEMO_STRIDE);
109    if (Status != XST_SUCCESS) {
110        xil_printf(
111            "Display Ctrl initialization failed during demo initialization%d\r\n",
112            Status);
113    }
114
115    Status = DisplayStart(&dispCtrl);
116    if (Status != XST_SUCCESS) {
117        xil_printf("Couldn't start display during demo initialization%d\r\n",
118            Status);
119    }
120
121
122
123    DemoPrintTest(dispCtrl.framePtr[dispCtrl.curFrame], dispCtrl.vMode.width,
124        dispCtrl.vMode.height, dispCtrl.stride, DEMO_PATTERN);
125    vdma_wrtie_init(XPAR_AXIVDMA_1_DEVICE_ID,1920 * 3,1080,1920 * 3,dispCtrl.framePtr[dispCtrl.curFrame]);
126
127
```

Part 22.3: Onboard Verification

Connect the board as follows. The HDMI input can be connected to the HDMI output interface of the PC, and the HDMI output interface of the board is connected to the monitor. After downloading the program, you can see the monitor displaying the PC screen.



Part 23: MIPI Acquisition and DP Display

Based on AN5641 Module

The experimental Vivado project directory is "an5641_mipi_dp /The previous chapter talked about the display of MIPI camera on DP, this chapter introduces the display of MIPI camera on HDMI."

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 23.1: 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 23.1.1: 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](#)

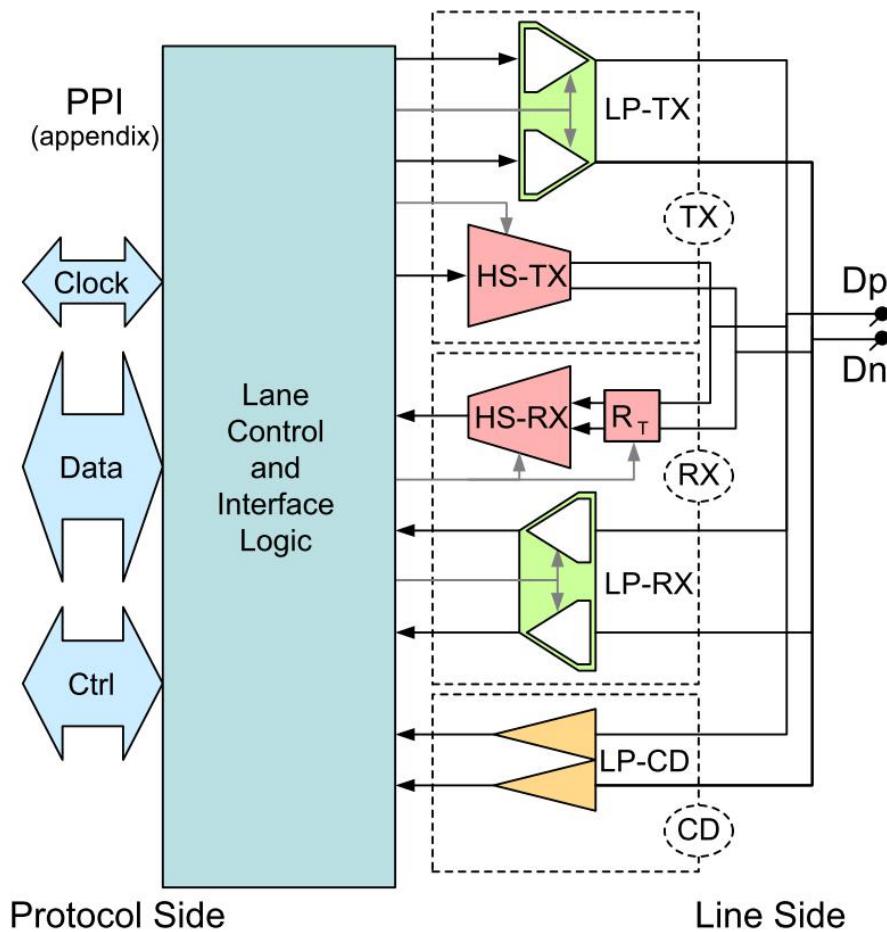


Figure 1 Universal Lane Module Functions

Part 23.1.2: 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.

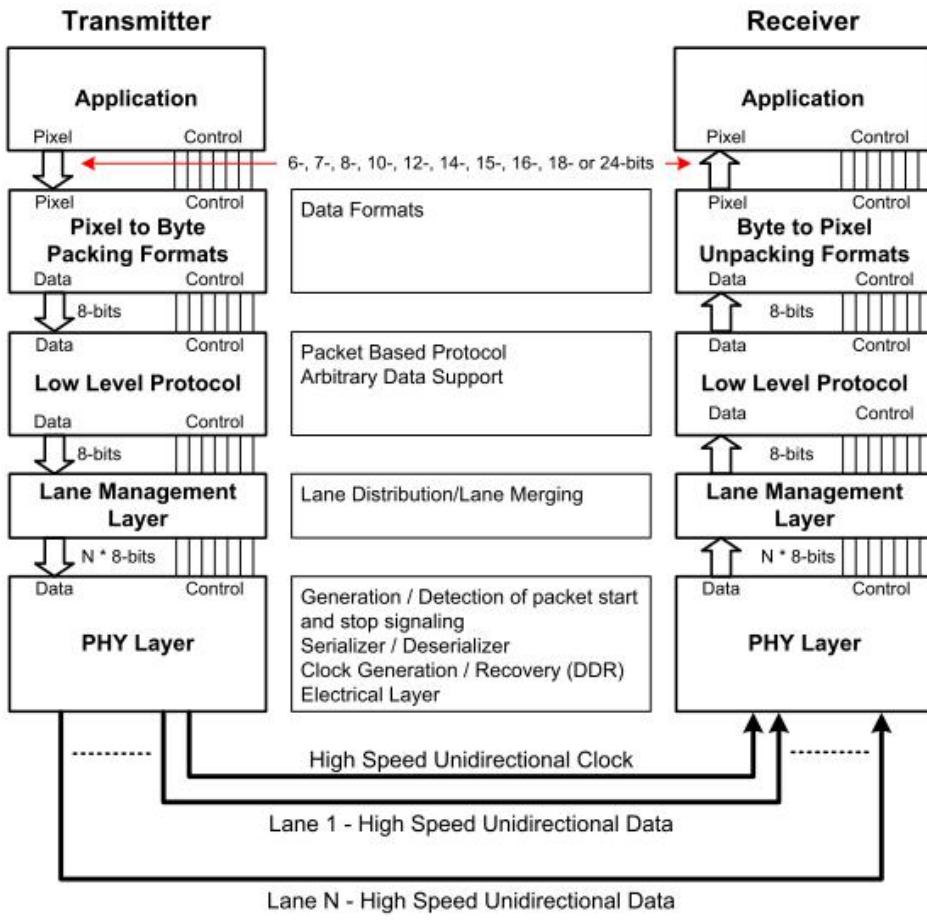


Figure 2 CSI-2 Layer Definitions

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:

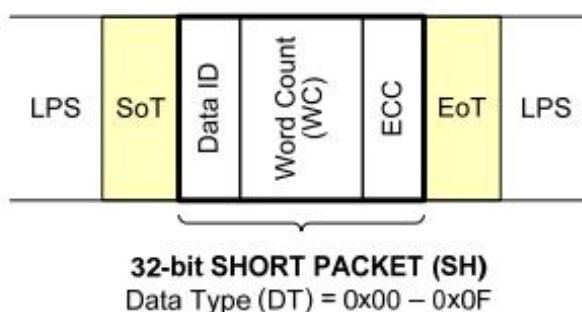


Figure 31 Short Packet Structure

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:

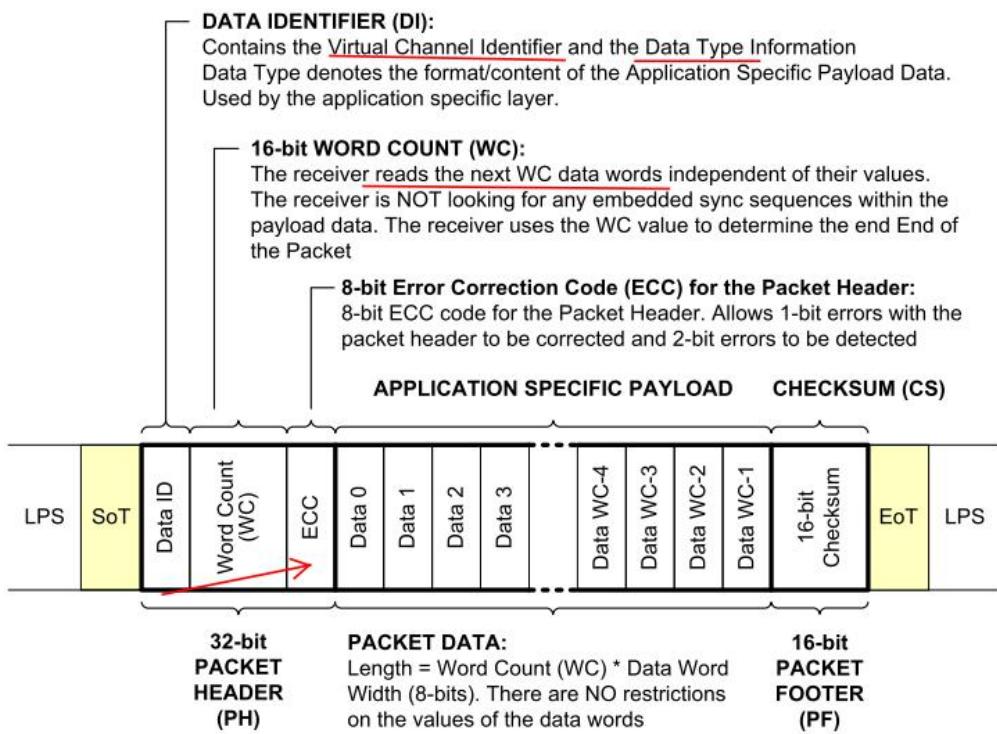


Figure 30 Long Packet Structure

For the specific content of MIPI CSI-2, please refer to

"MIPI Alliance Specification for Camera Serial Interface 2 (CSI-2).pdf"

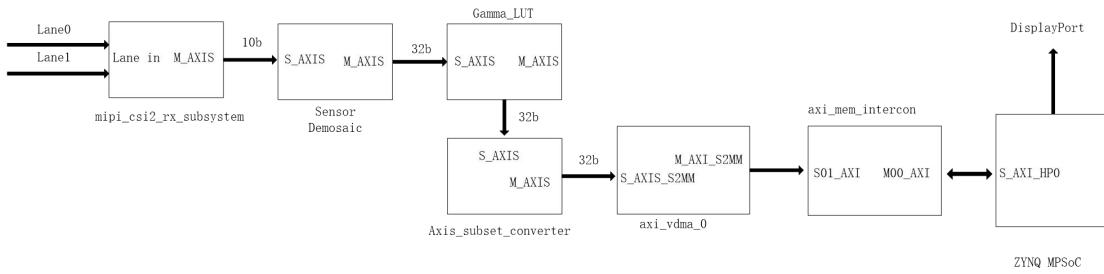
Saved in the the project catalog

FPGA Engineer Job Content

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

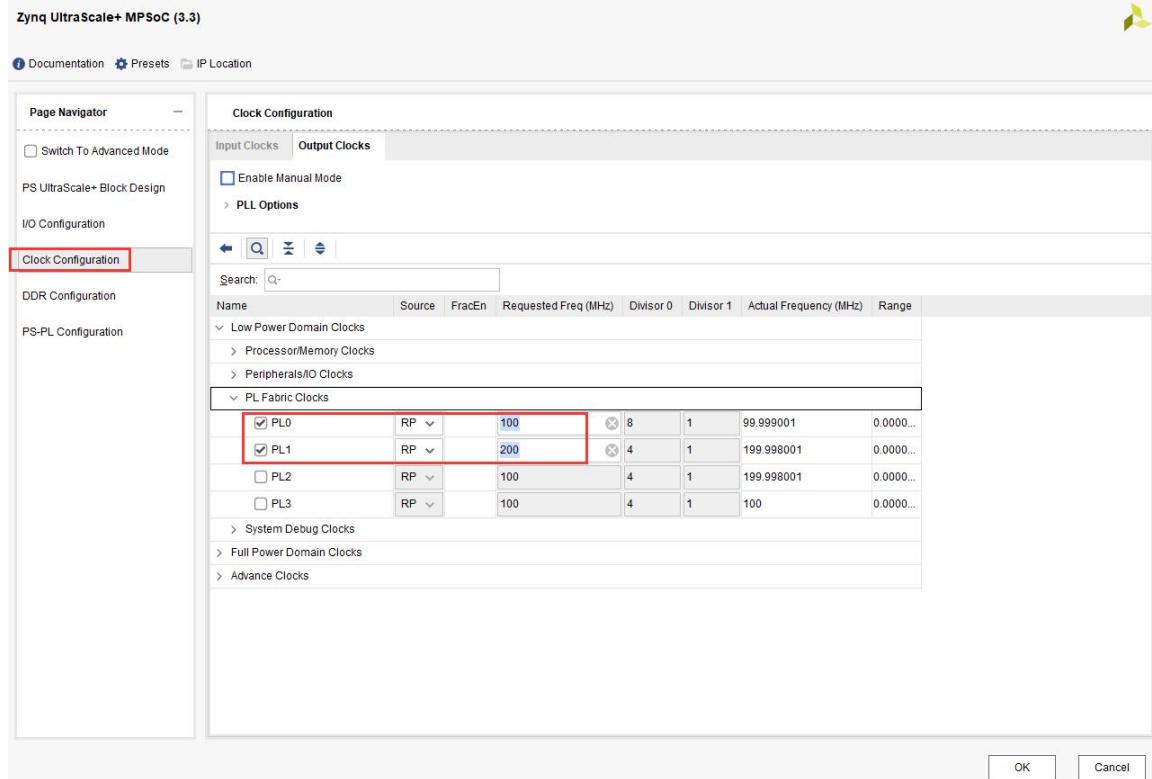
Part 23.2: Hardware Environment

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 [Gammer](#) calibration and other modules, enter the [VDMA](#), and then enter the [HP](#) port.

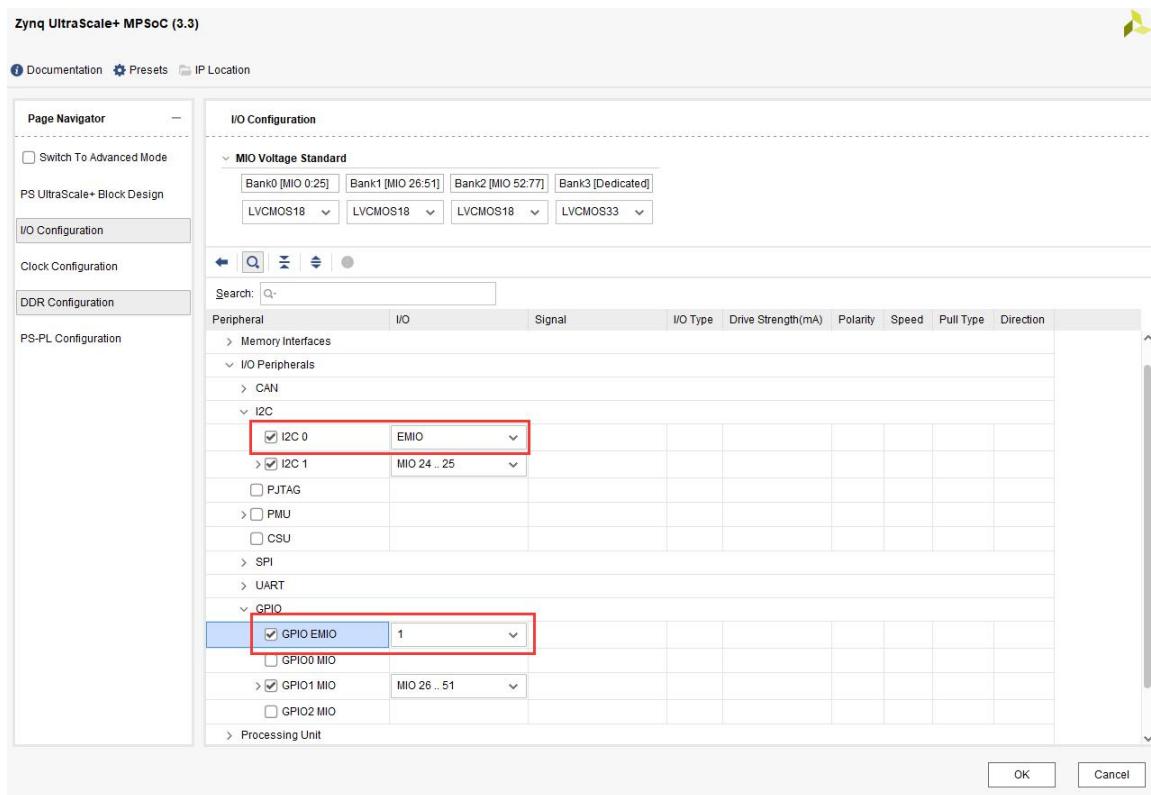


Based on the "ps_hello" project, we add a module for the MIPI acquisition part.

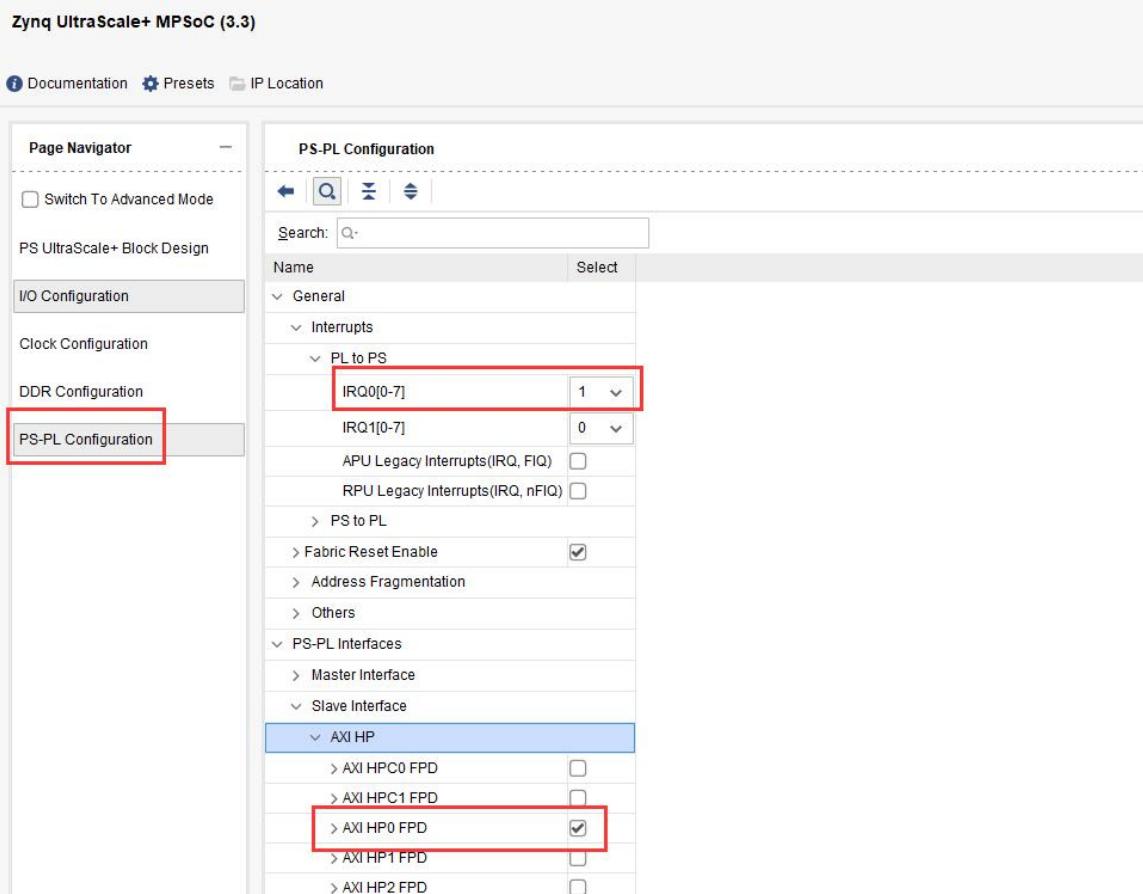
- 1) First configure the ZYNQ core, 100MHz is used for data transmission, and 200MHz is used for the MIPI module reference clock.



Configure i2c as EMIO, used to configure the camera register, GPIO EMIO set to 1, used to configure the camera enable



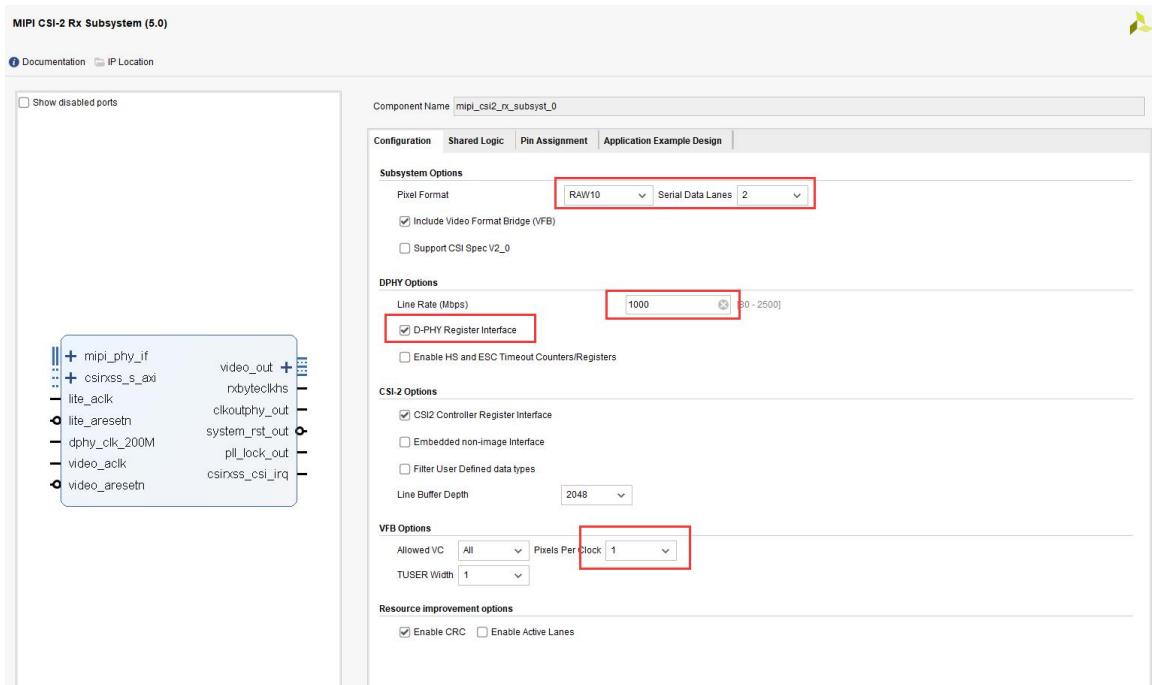
Configure interrupt and HP port

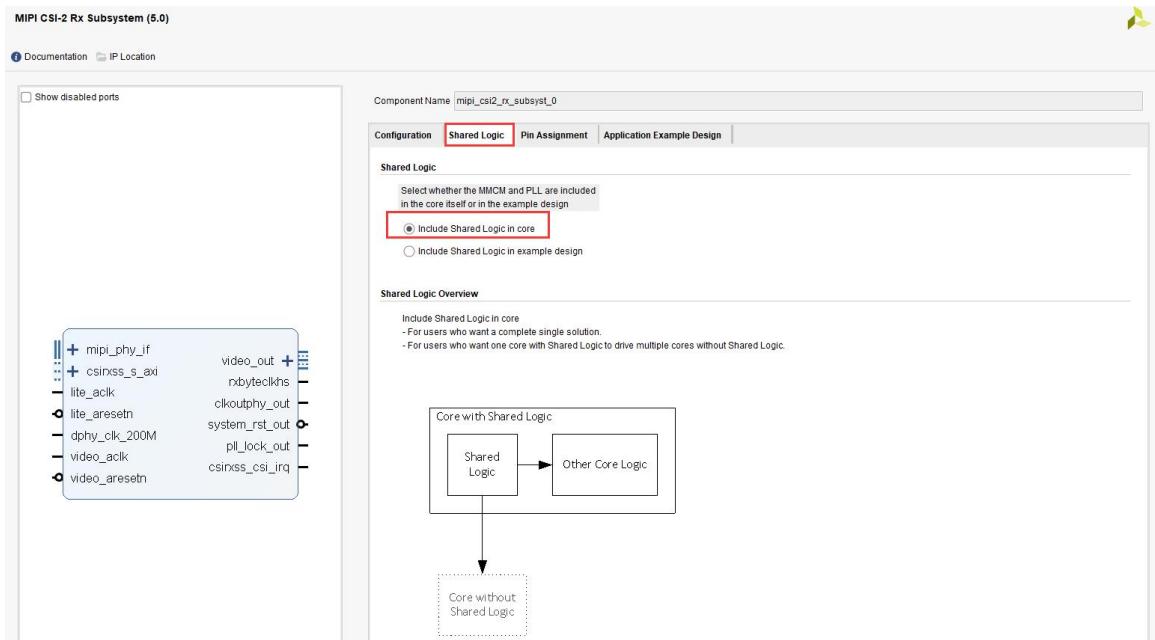


And connect the clock

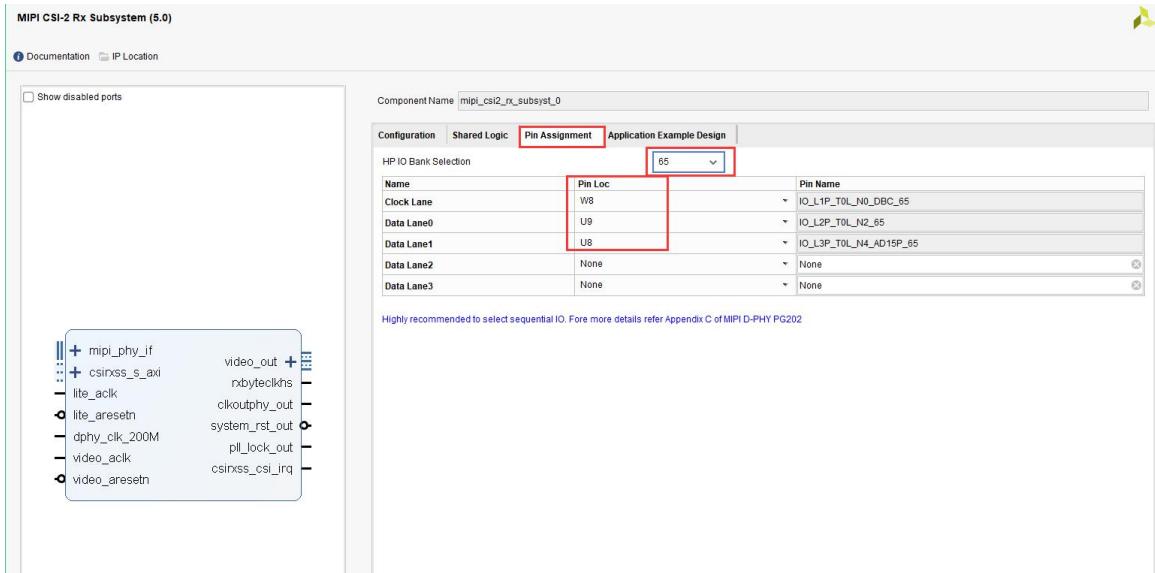


- 2) Add `mipi_csi2_rx_subsystem` module for receiving and parsing MIPI data and converting it to `axi-stream` interface. The configuration is as follows, the data format is **RAW10, 2 lanes** are selected, and the **line rate** is configured as **1000Mbps**, which refers to the maximum supported rate. 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;

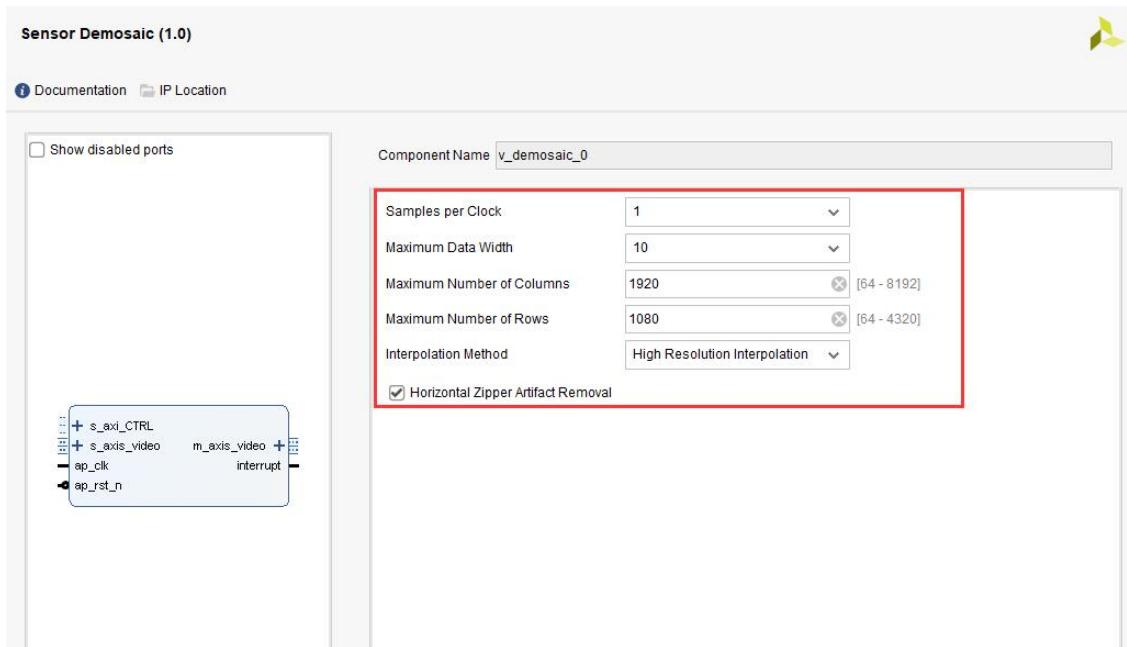




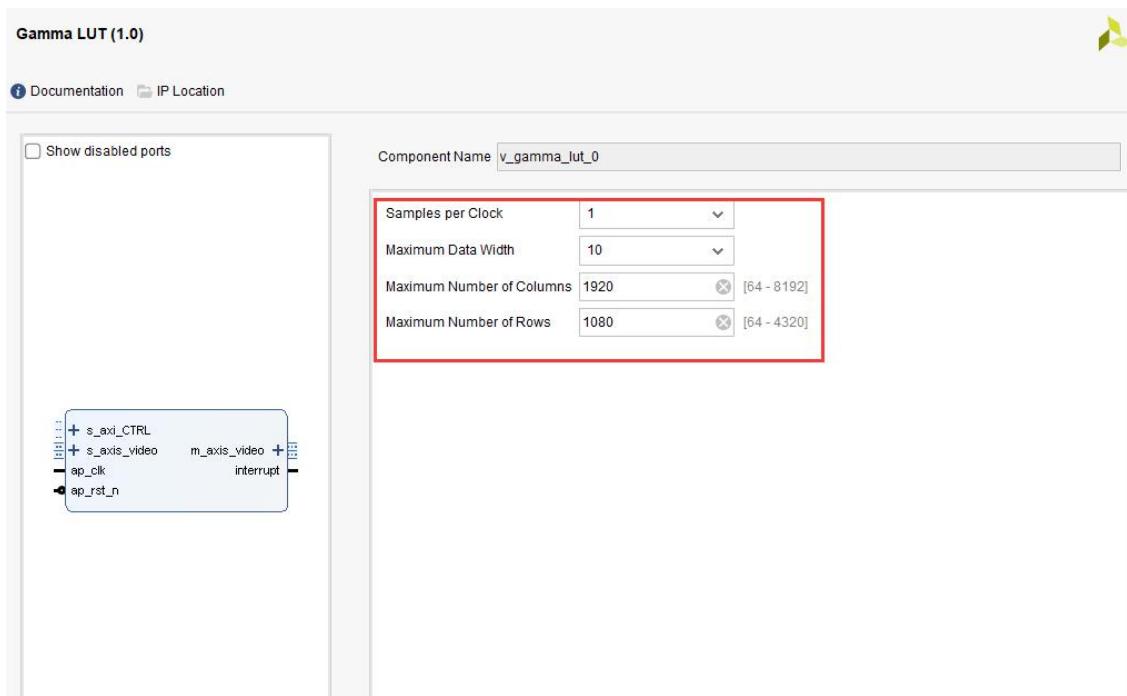
Pin Assignment is configured as follows according to the schematic



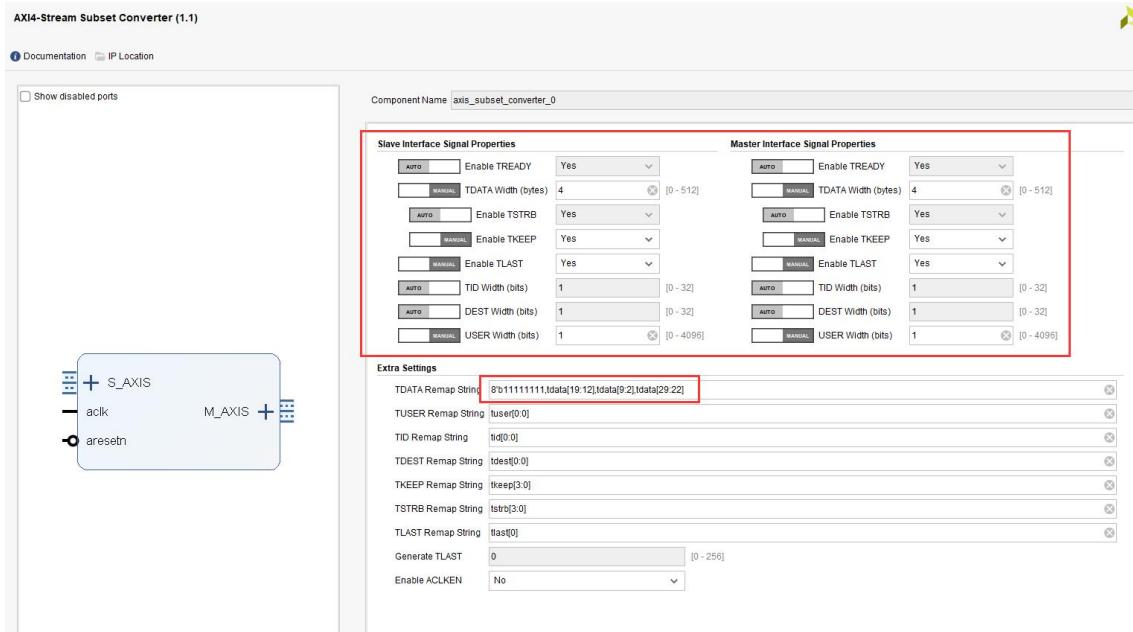
- 3) Add the Sensor Demosaic module to convert RAW10 to RGB, the configuration is as follows, the data bit width is set to 10, and the maximum resolution is 1920*1080



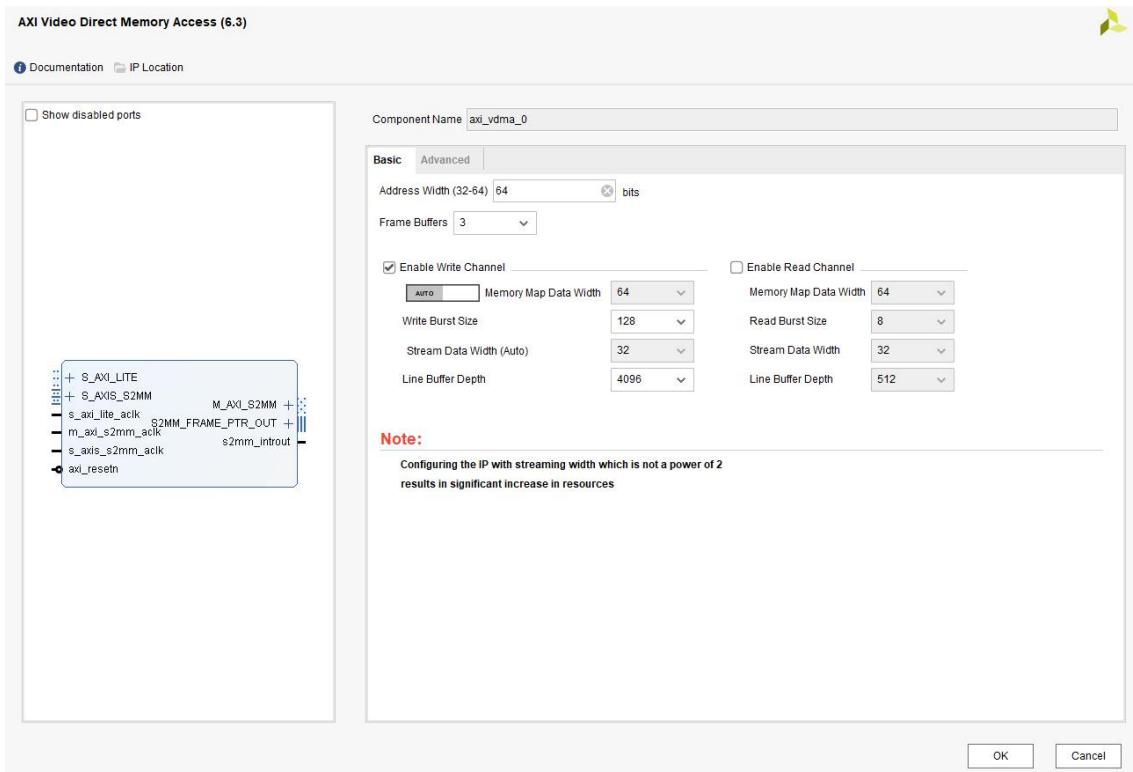
4) Add Gamma LUT module for Gamma correction

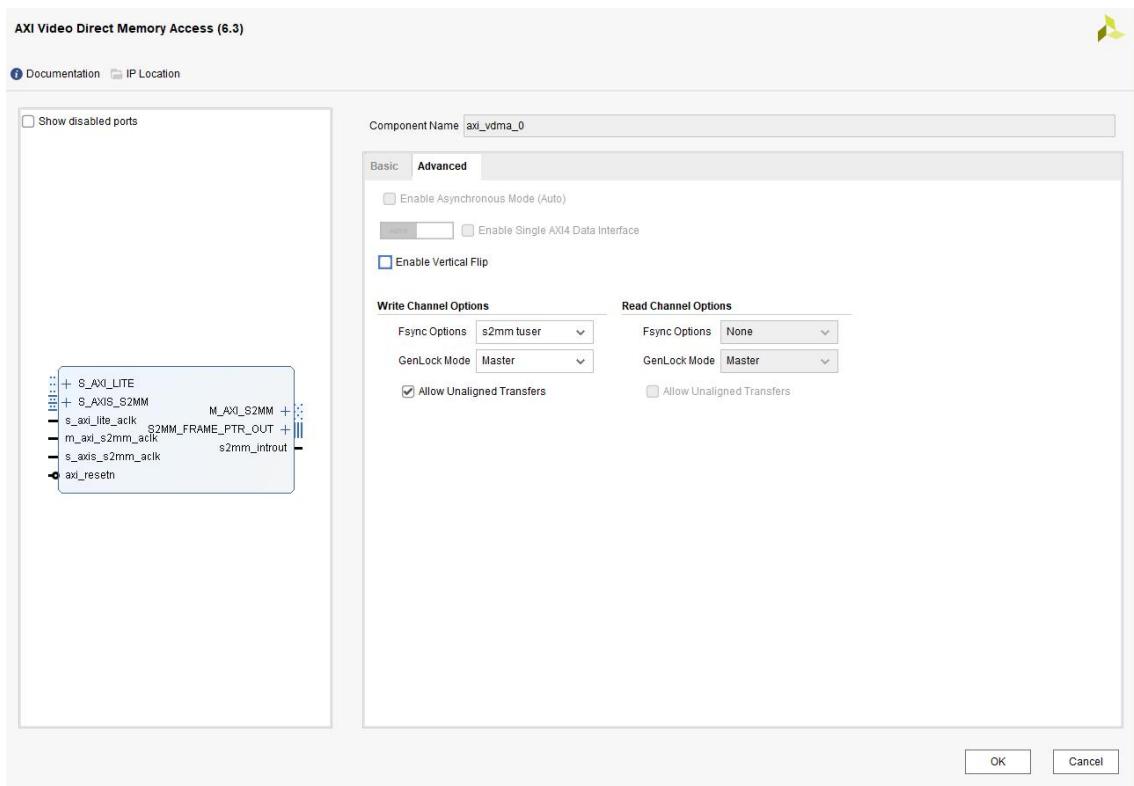


5) Add the **subset** module to adjust the order of the image data, because after actual operation, it is found that the order of the RGB data of the image needs to be adjusted.

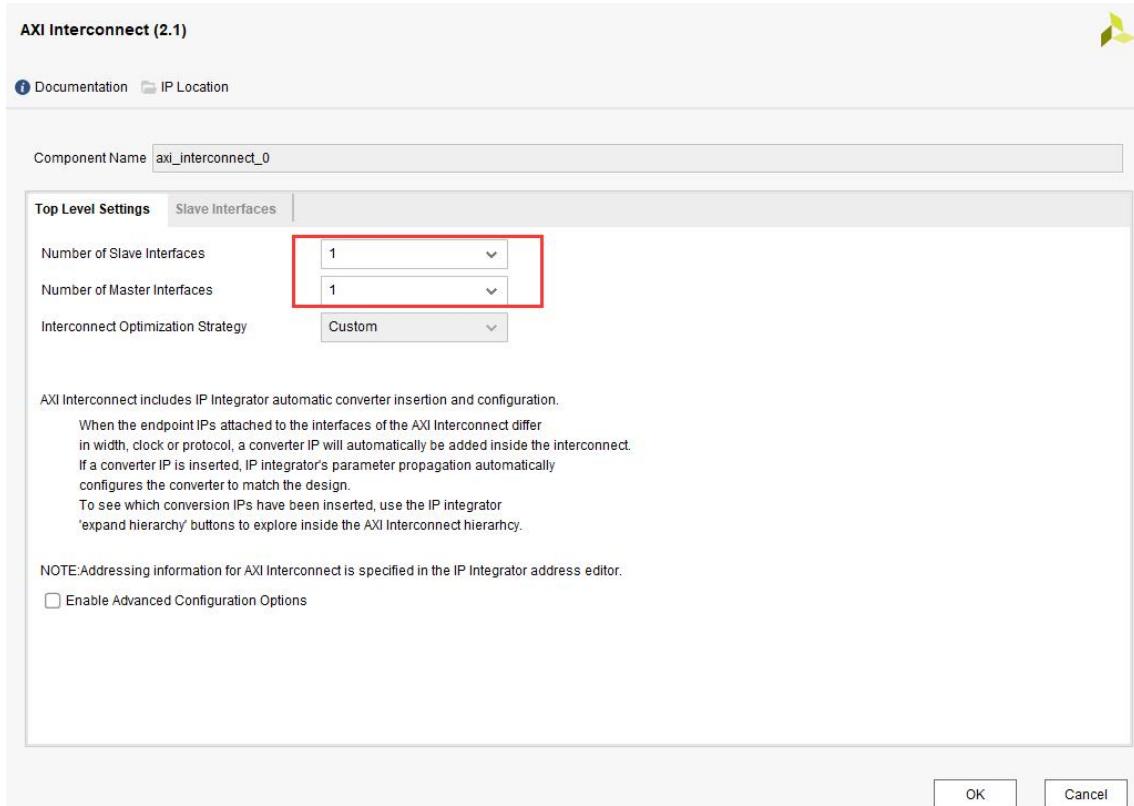


6) Add the VDMA configuration as follows

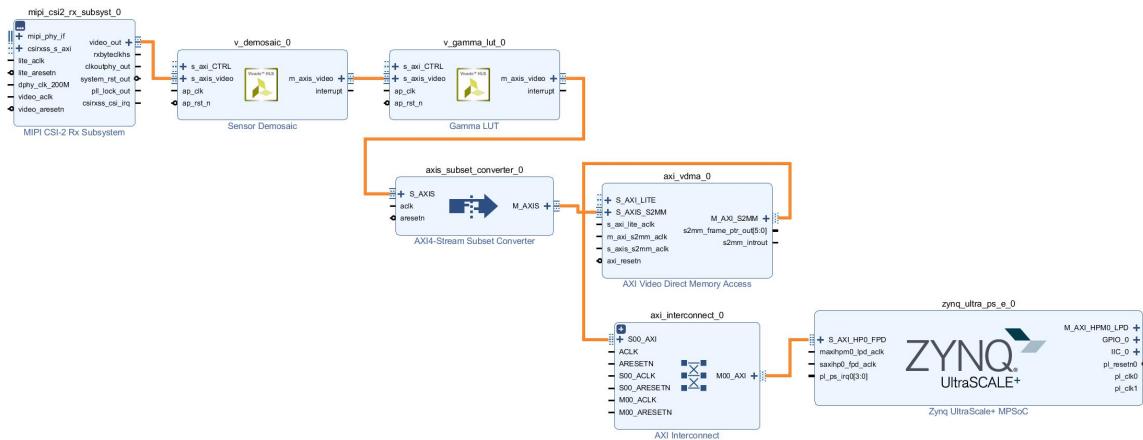




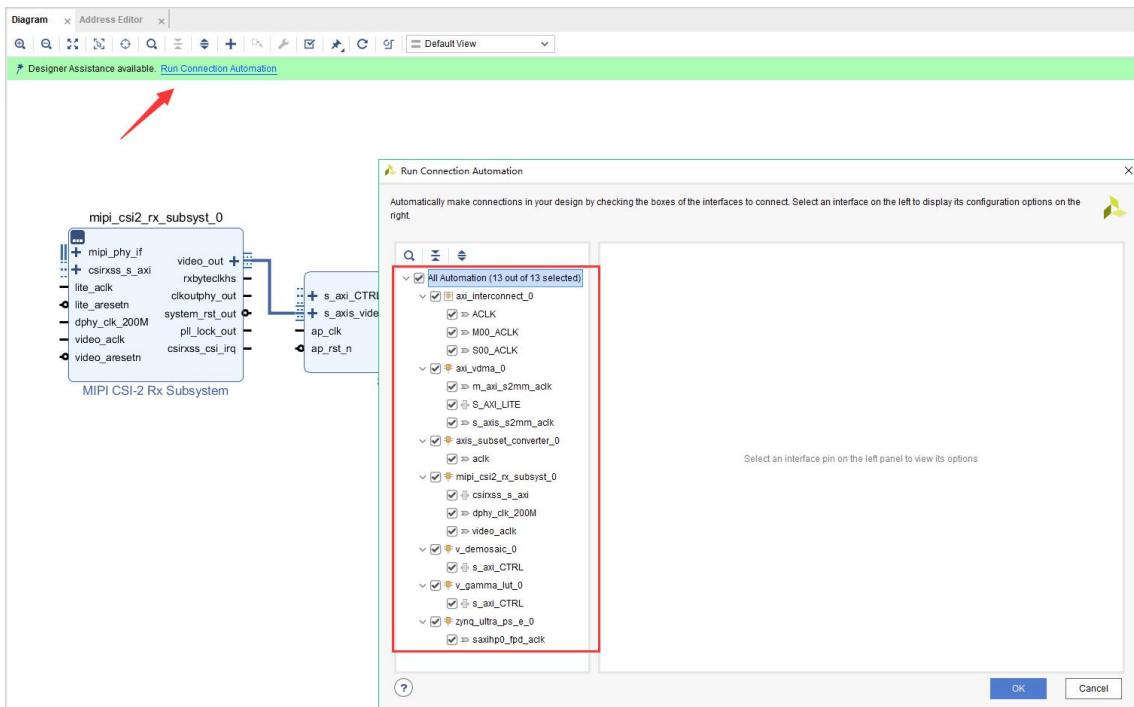
- 7) Add an **AXI Interconnect** module and configure it as 1 **master** and 1 **slave**



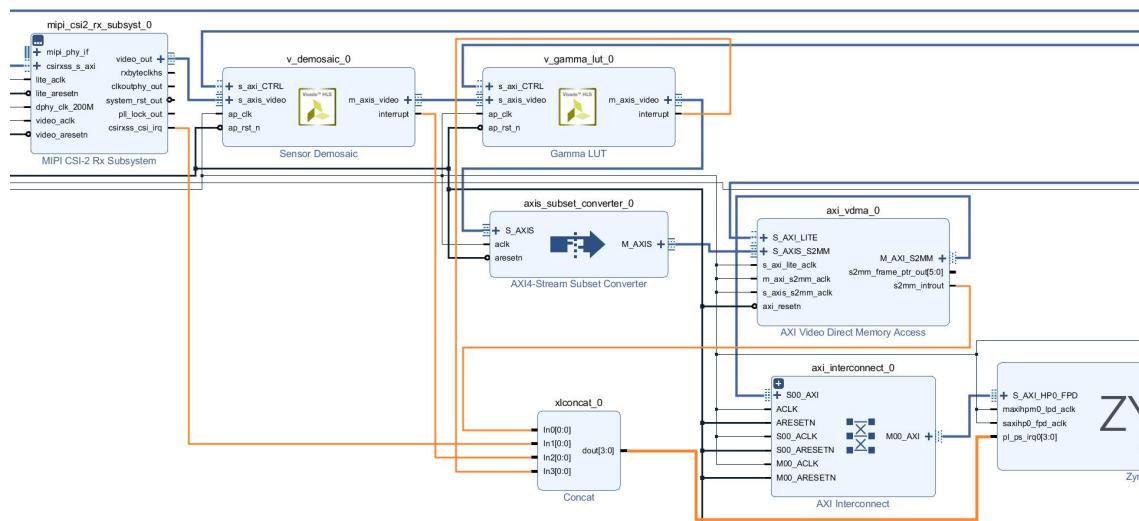
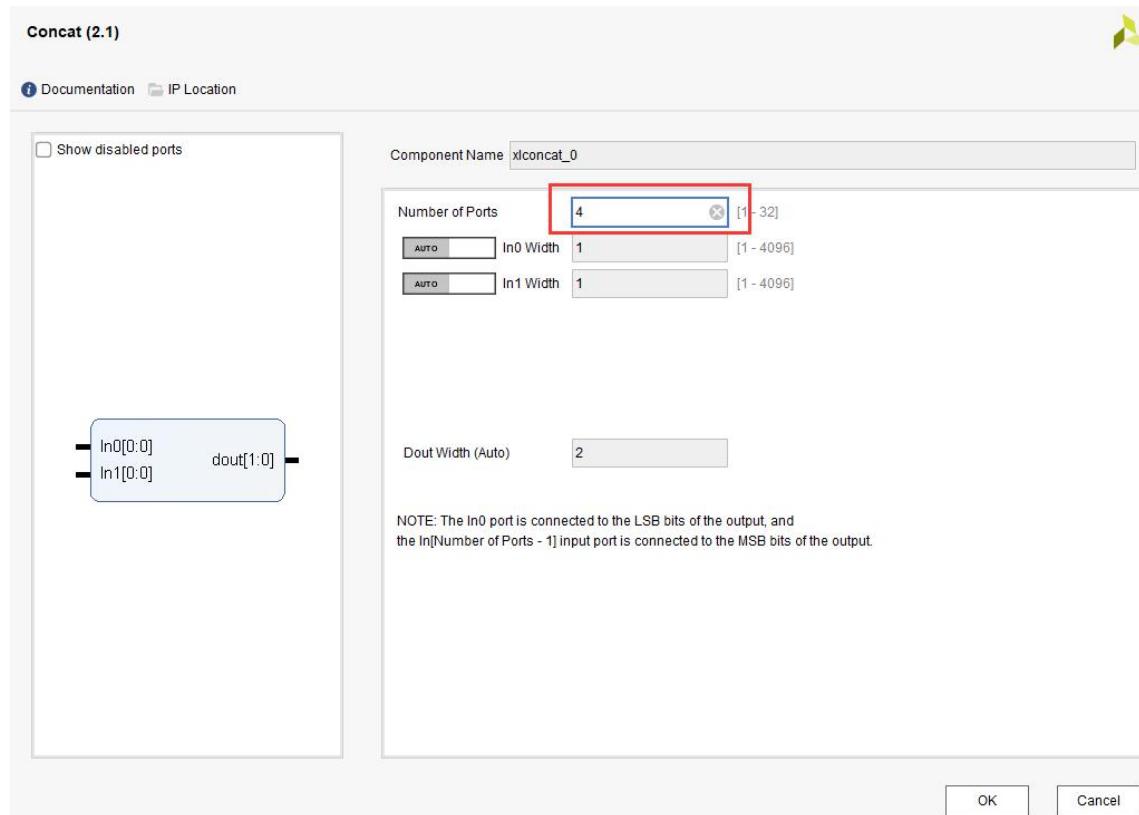
- 8) Connect key signals



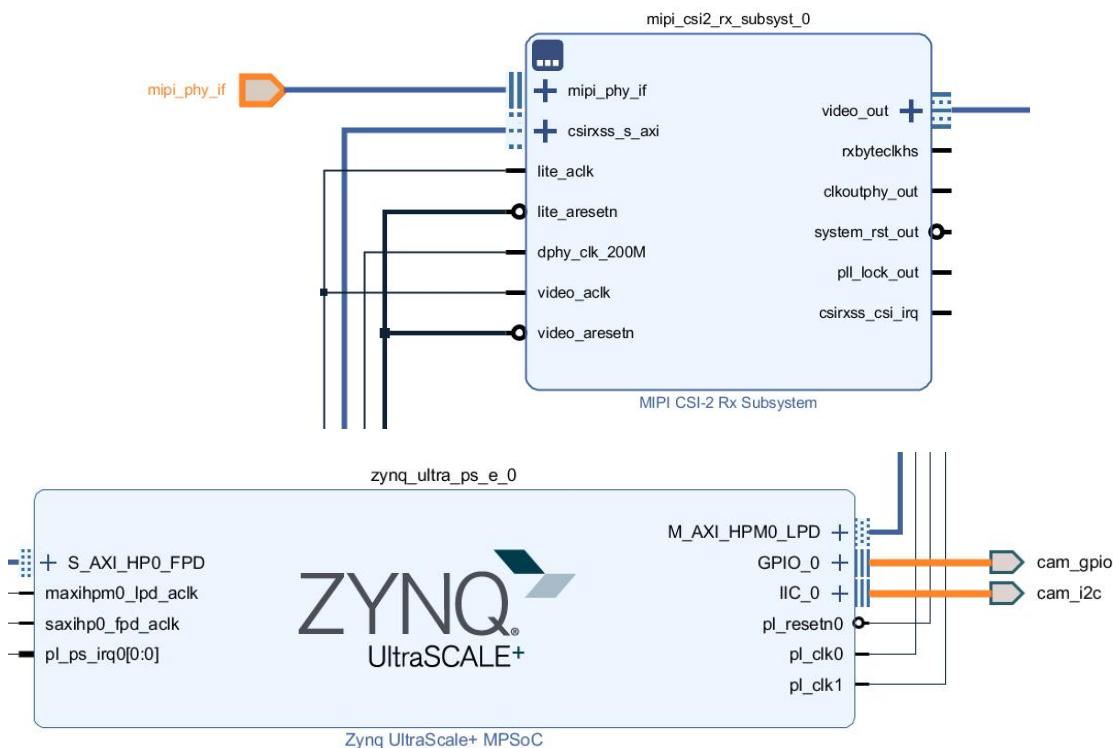
9) Automatic connection



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 23.3: 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,100000);

```

At the end, configure the MIPI camera and start the VDMA of the camera.

```

/*
 * 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();

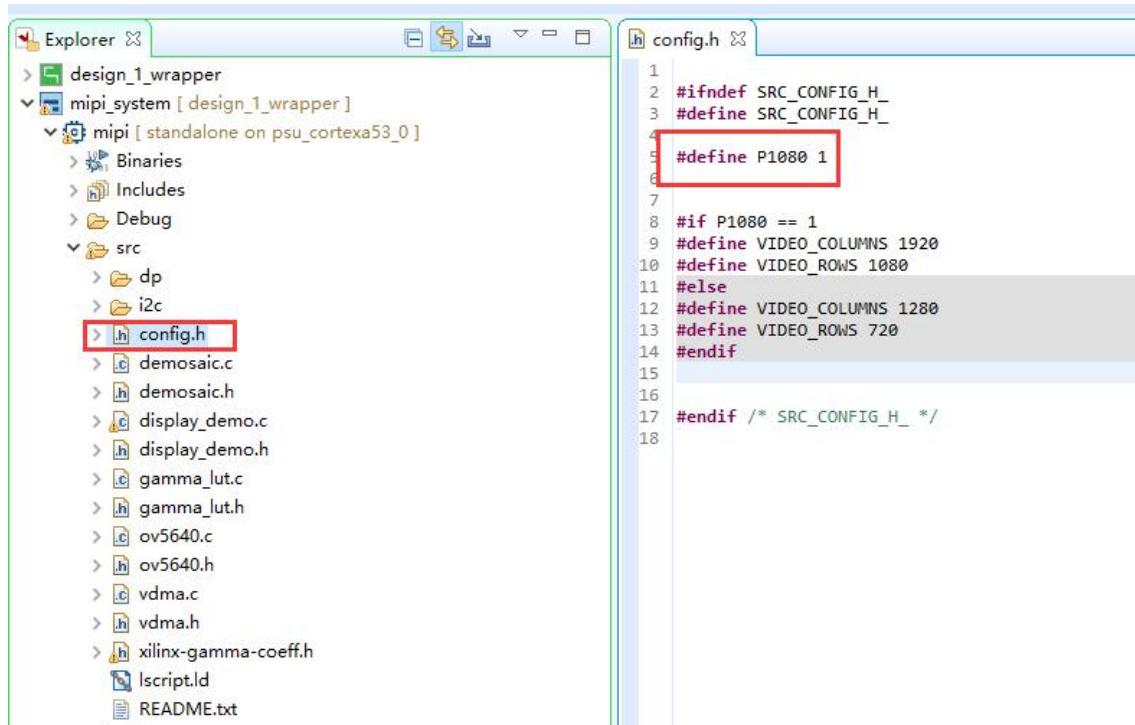
/* Start Sensor Vdma */
vdma_write_init(XPAR_AXIVDMA_0_DEVICE_ID,HORSIZE,VERSIZE,DEMO_STRIDE,(unsigned int)pFrames[0]);

/*
 * Initialize Sensor
 */
sensor_init(&ps_i2c0);

return 0;

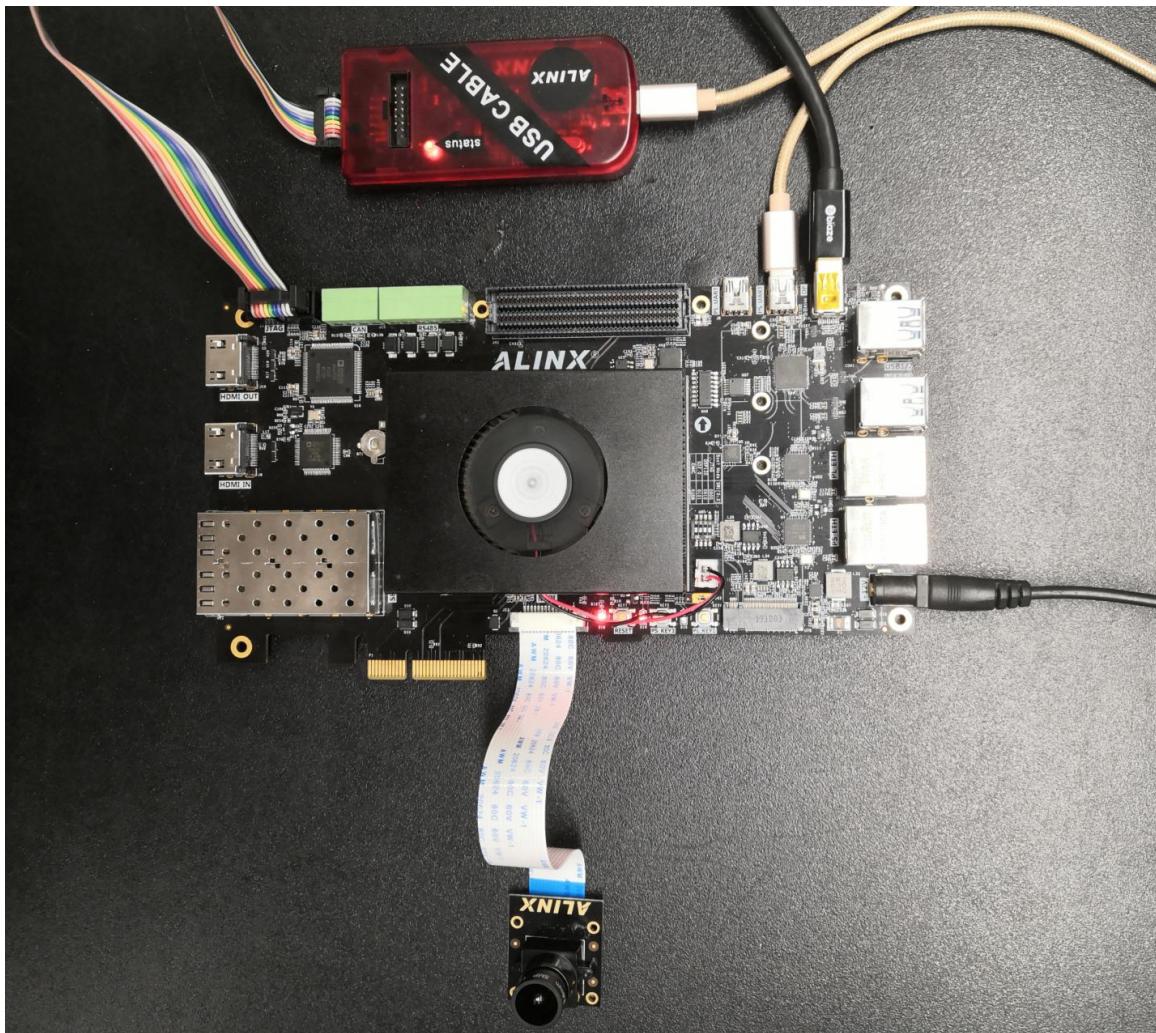
```

Currently the program supports two resolution configurations, 720p @60fps and 1080p@30fps. If you want to change to 1080p, Need to modify the macro definition in config.h and change the value of P1080 to 1.

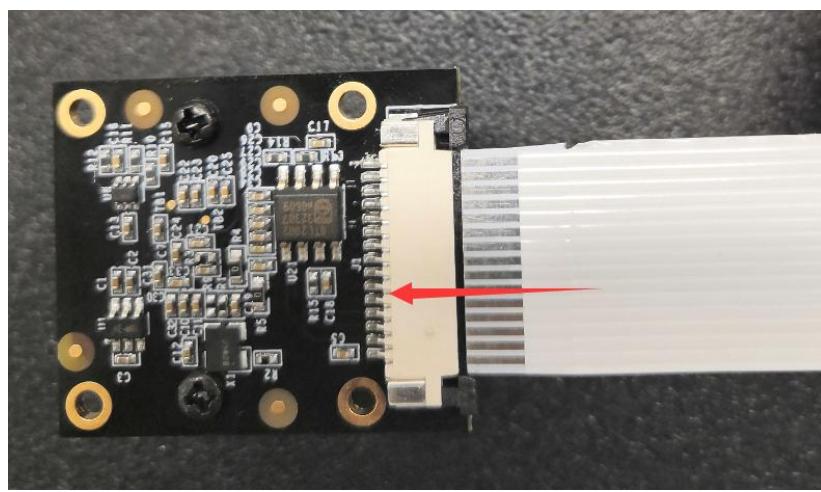


Part 33.4: Onboard Verification

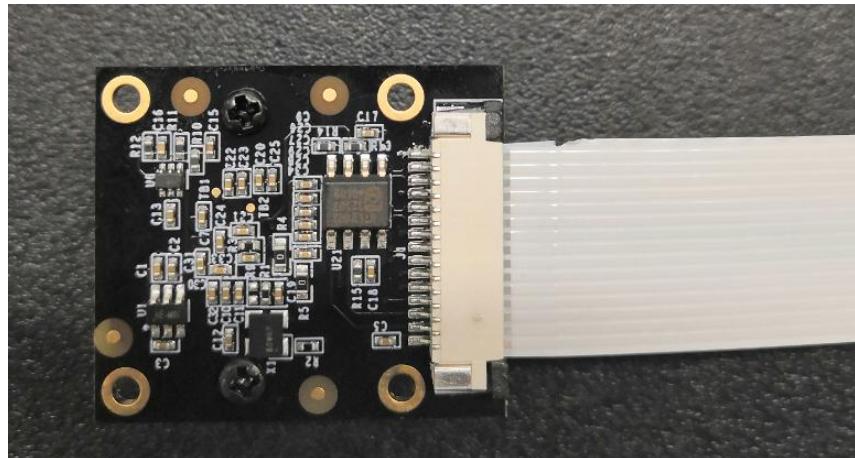
Connect the MIPI camera as shown in the figure below.



Hardware Connection

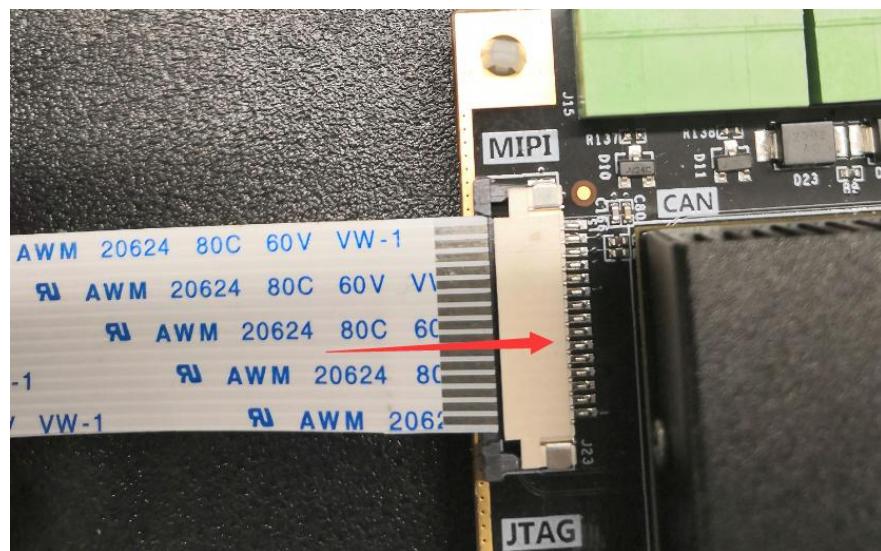


Camera Cable Connection (Not Connected)

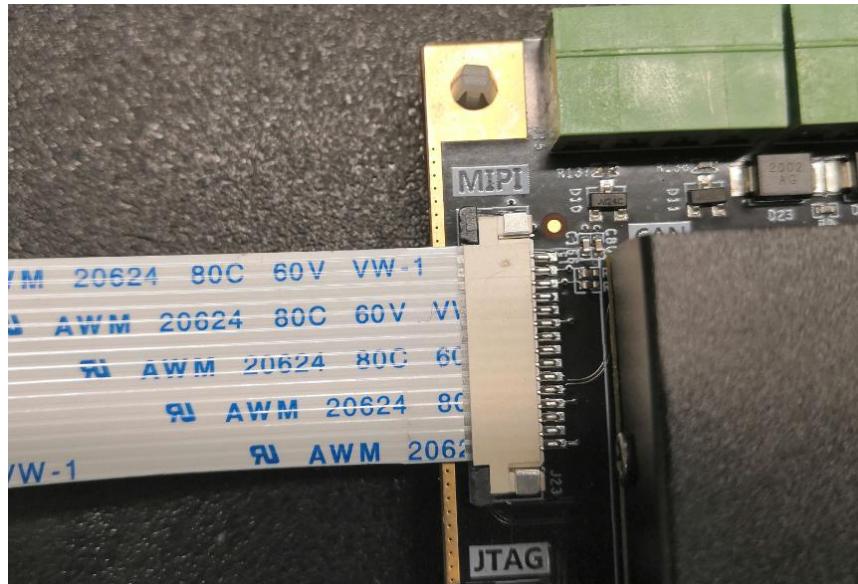


Camera Cable Connection (Connected)

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 24: MIPI Acquisition and HDMI Display Based on AN5641 Module

The experimental Vivado project directory is "an5641_mipi_hdmi/vivado".

The experiment vitis project directory is "an5641_mipi_hdmi/vitis".

The previous chapter talked about the display of MIPI camera on DP, this chapter introduces the display of MIPI camera on HDMI.

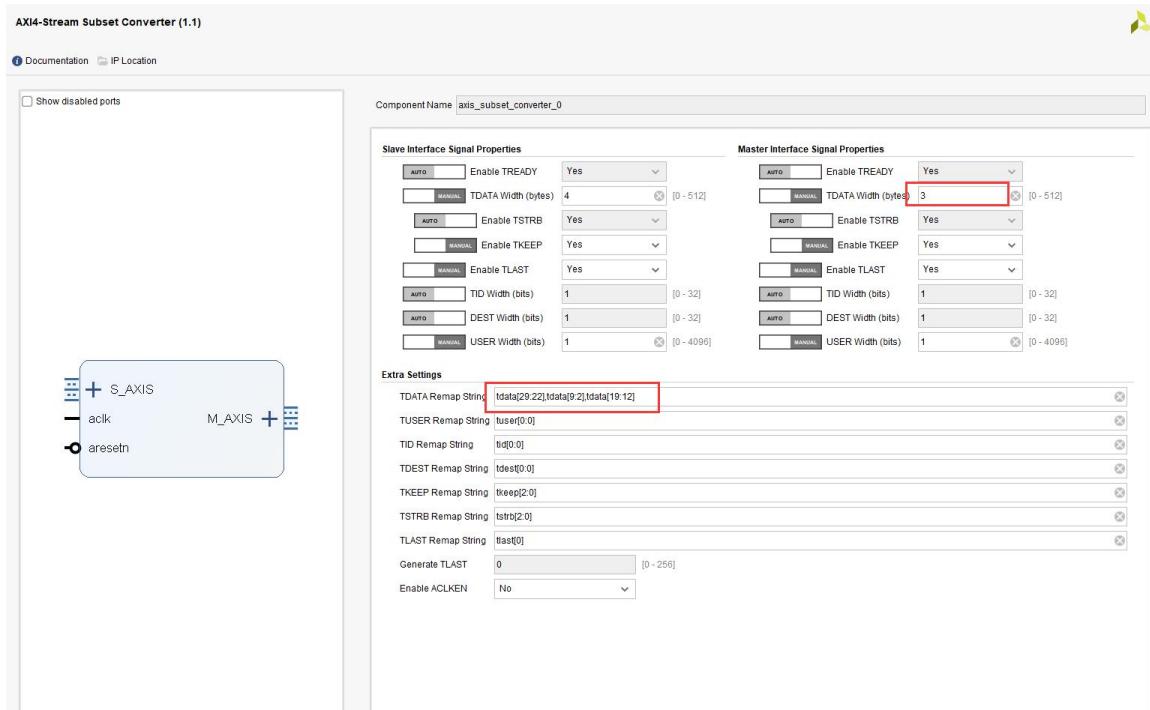
FPGA Engineer Job Content

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

Part 24.1: Create a Vivado Project

On the basis of the vivado project of MIPI acquisition and DP display, the HDMI output part can be added, and the MIPI part can also be added on the basis of the vivado project of VDMA driving HDMI display.

Since the HDMI display data format is RGB888, the subset converter needs to be modified



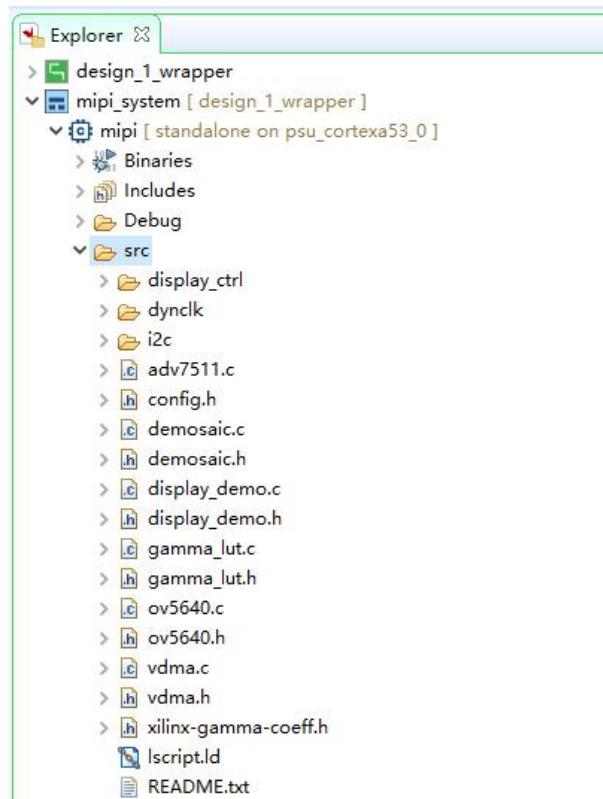
Other parts can refer to the project of the routine

Software Engineer Job Content

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

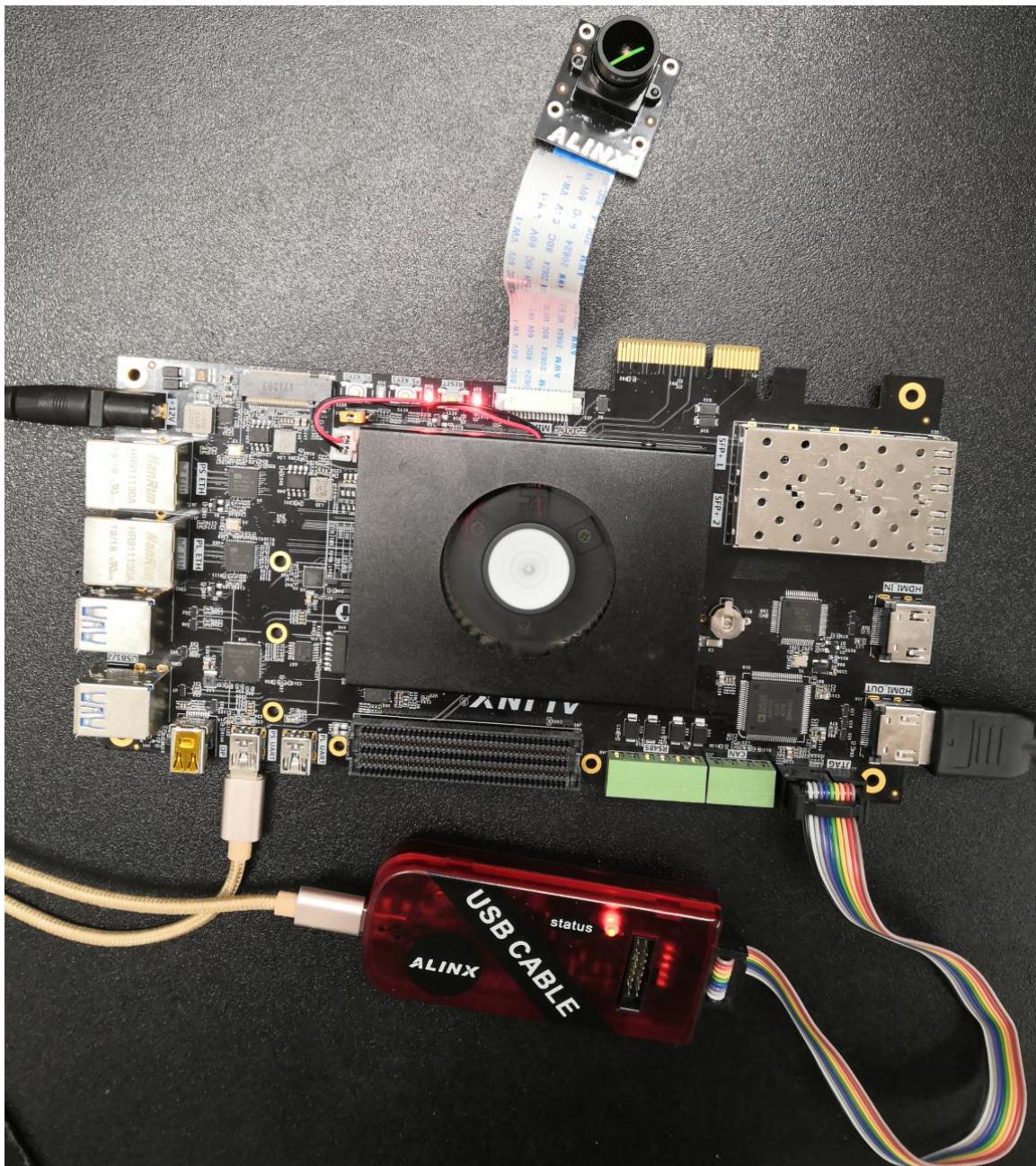
Part 24.2: Vitis Software Writing and Debugging

As for the writing of the program, it is also a combination of the two. For details, please refer to the routine code



Part 24.3: Onboard Verification

Connect the MIPI camera, connect the HDMI output interface to the monitor, download the program, you can see the captured camera image



Hardware Connection

Part 25: PCIe Test

The experimental Vivado project directory is "PCIe_test /vivado".

The experiment vitis project directory is "PCIe_test /vitis".

This experiment uses **PCIe XDMA** provided by Xilinx to complete a simple PCIe, read and write experiment to test the speed of PCIe. In **PCIe 3.0**, the speed of each **lane** is **8Gbit/s**, and the available bandwidth can reach **80%** due to overheads such as encoding and protocols. **ZYNQ's HP port has a maximum bit width of 128bit. If the clock is 200MHz**, the bandwidth can reach **25.6Gbit/s**, which can fully meet the bandwidth requirements of **PCIe 3.0 x2**.

FPGA development board provides Windows 7 64-bit and Windows 10 64-bit driver source code. The driver program is developed using VS2015, and the FPGA development board also provides the driver source code and test program of the Linux version. In order to test more intuitively under Windows, ALINX uses QT to develop some test programs with interfaces to facilitate testing.

FPGA Engineer Job Content

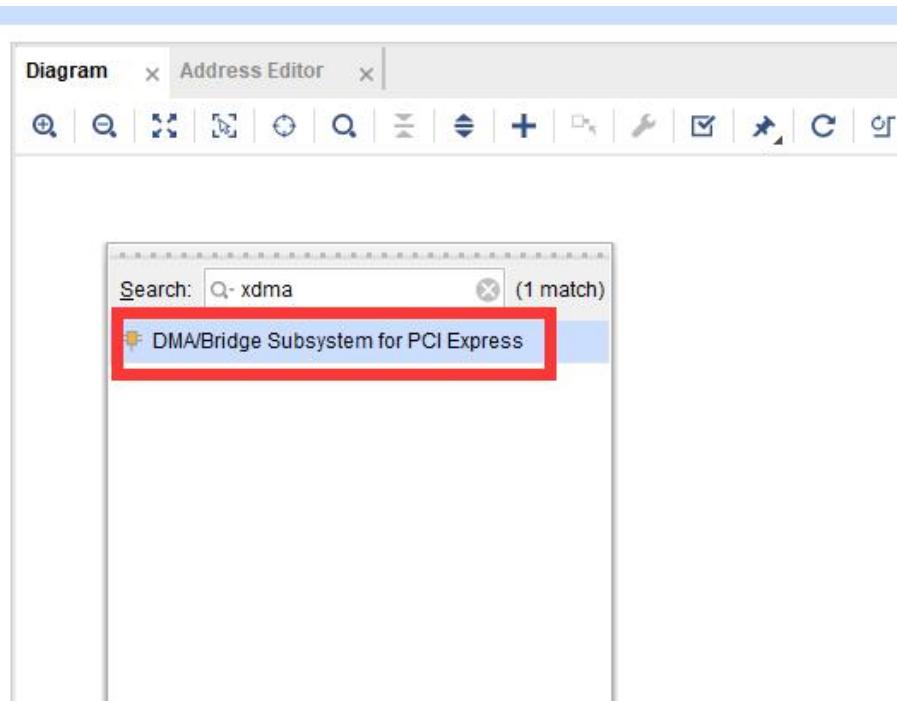
The following is the content that FPGA engineers are responsible for.

Part 25.1: Create a Vivado Project

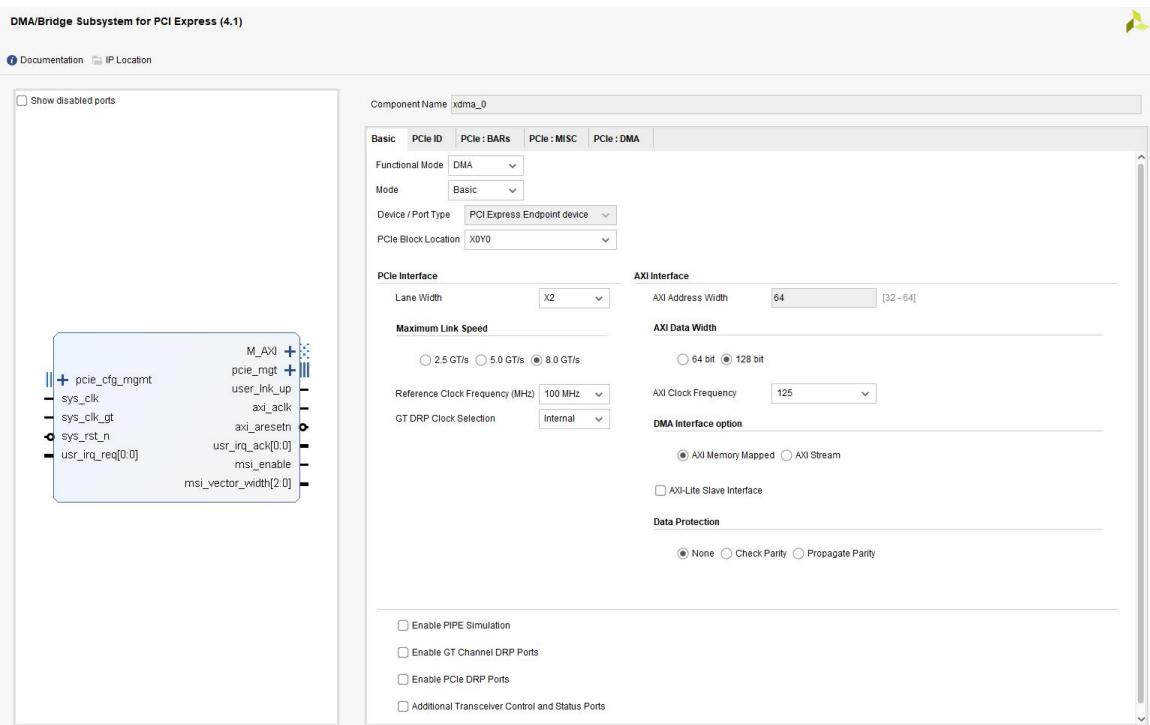
Based on the **ps_hello** project.

Part 25.1.1: PCIe xdma Configuration

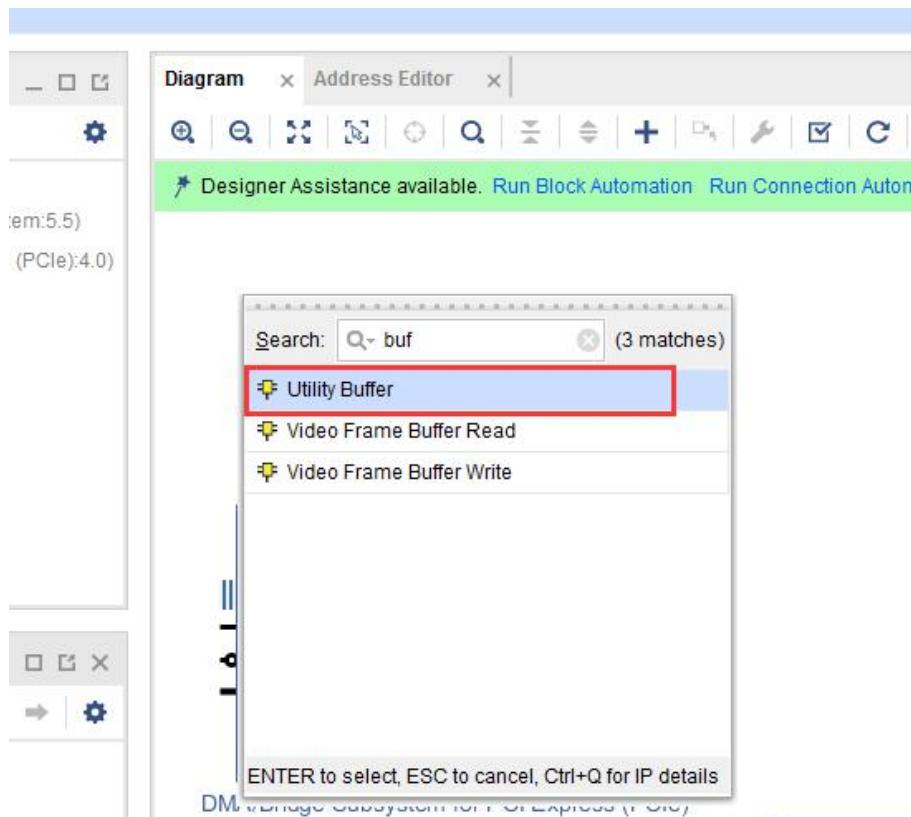
- 1) Add **PCIe DMA** subsystem



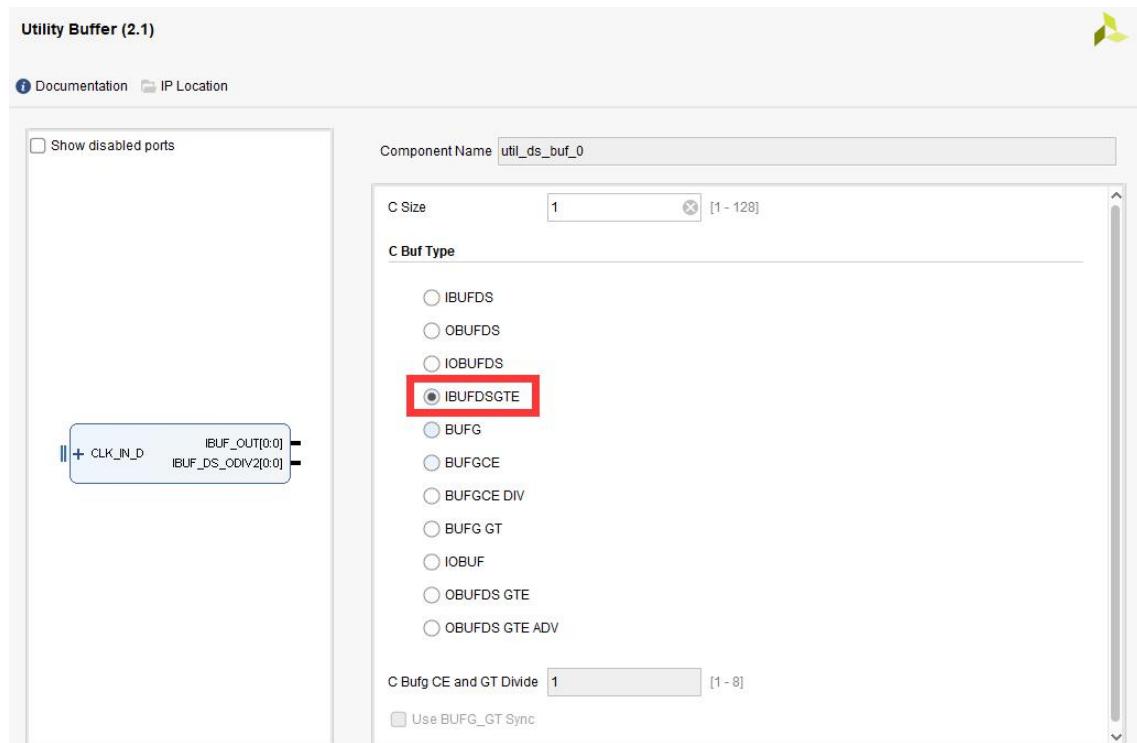
- 2) Configure PCIe xdma parameters, select X2 for Lane width, select 8.0 GT/s for speed, and keep other parameters as default.



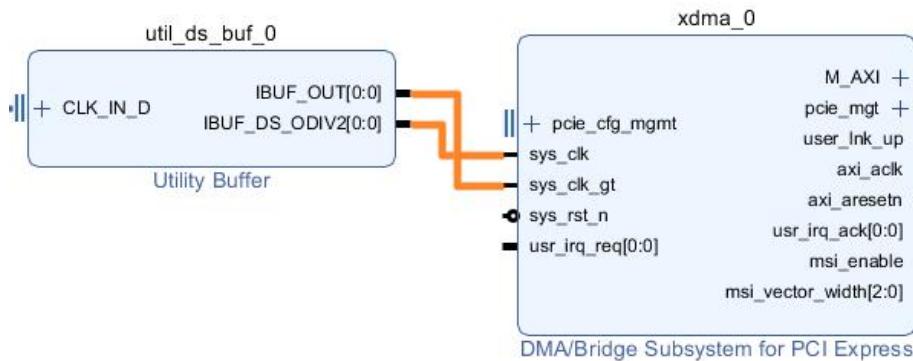
- 3) Add PCIe differential reference clock buffer



4) Configured as a transceiver clock differential buffer



5) Connect the PCIe reference clock

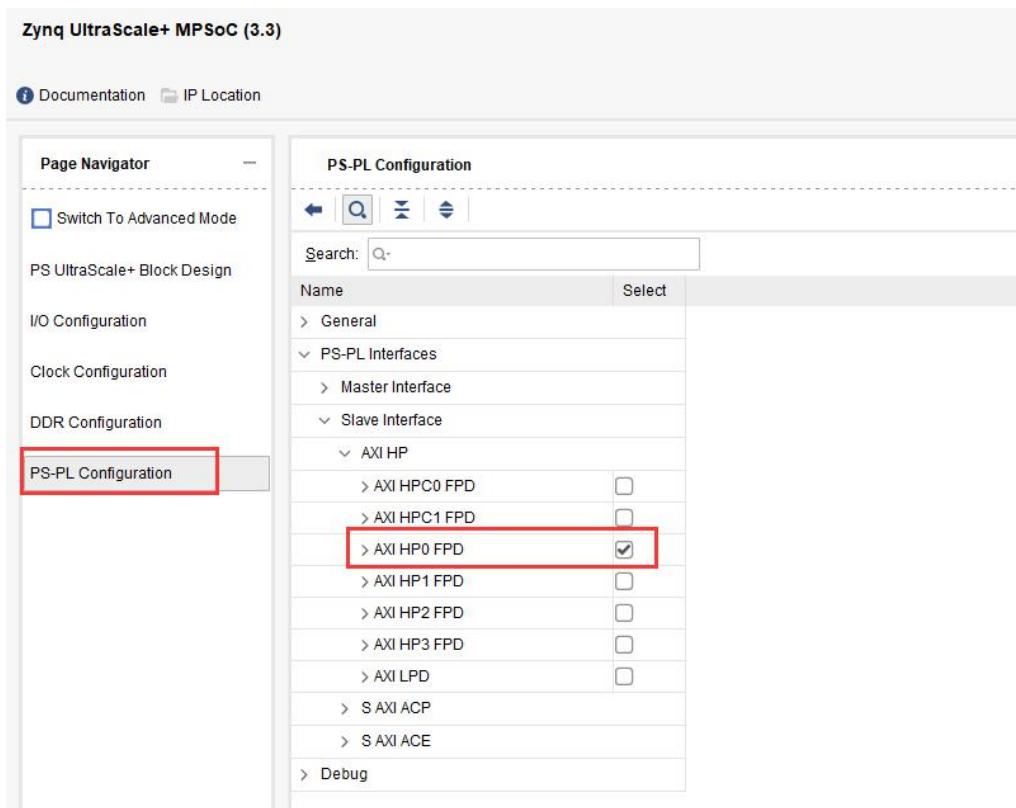


Part 25.1.2: ZNYQ Configuration

1) Configure PL end CLK1 to 200MHz

Name	Source	FracEn	Requested Freq (MHz)	Divisor 0	Divisor 1	Actual Frequency (MHz)	Range
PL0	RP		100	8	1	99.999001	0.000
PL1	RP		200	4	1	199.998001	0.000
PL2	RP		100	4	1	100	0.000
PL3	RP		100	4	1	100	0.000

2) Add HP port

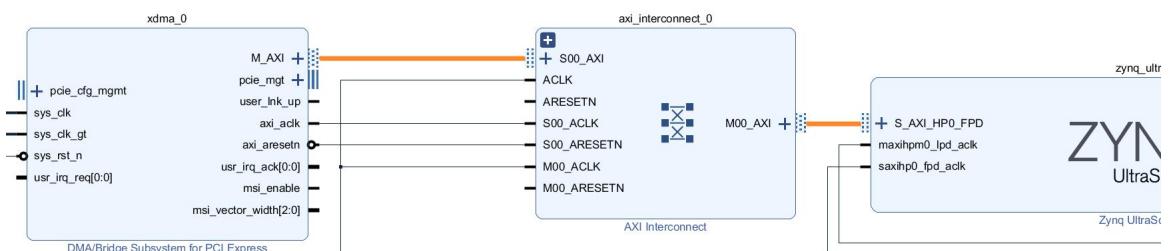


3) Connect the clock

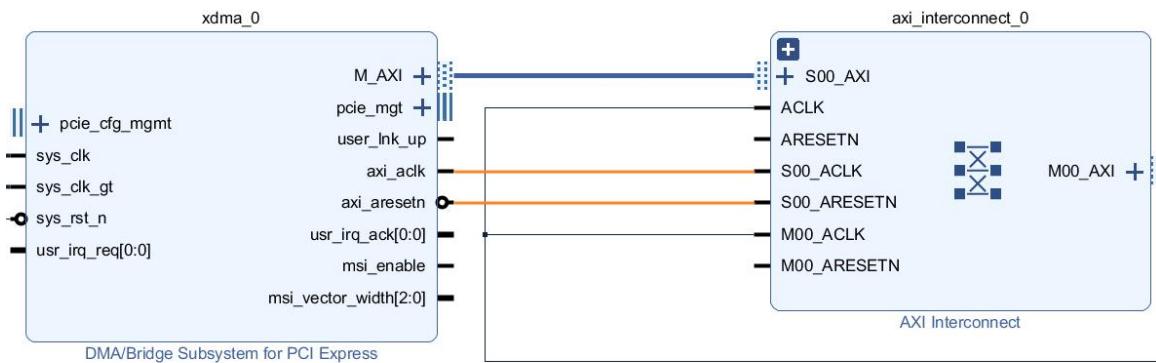


Part 25.1.3: Module Connection

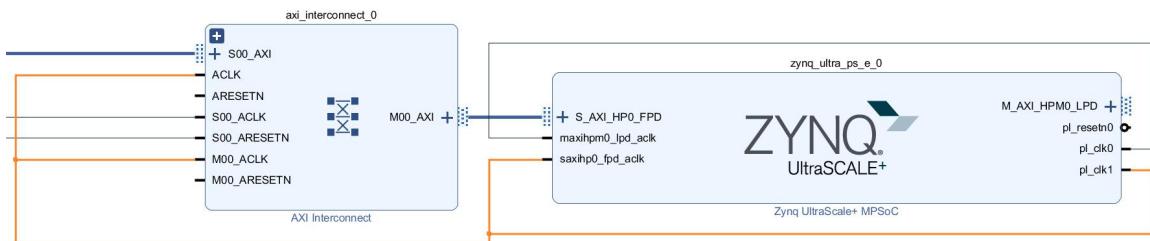
- 1) Add axi_interconnect, connect M00_AXI to S_AXI_HP0_FPD, connect M_AXI of xdma to S00_AX



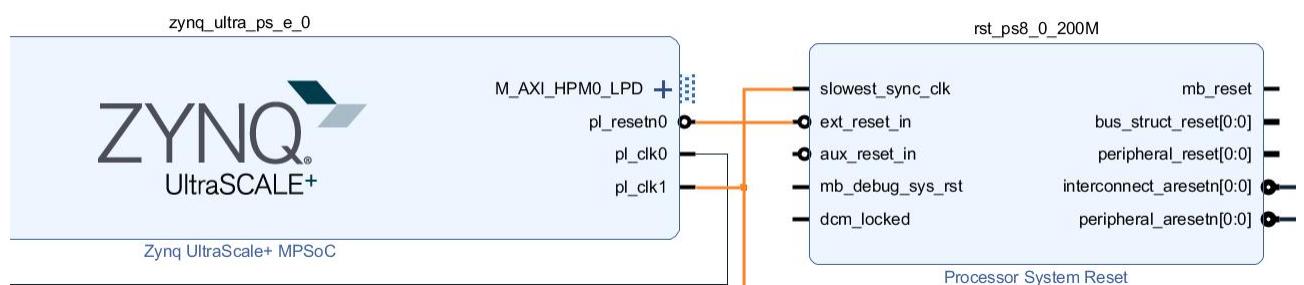
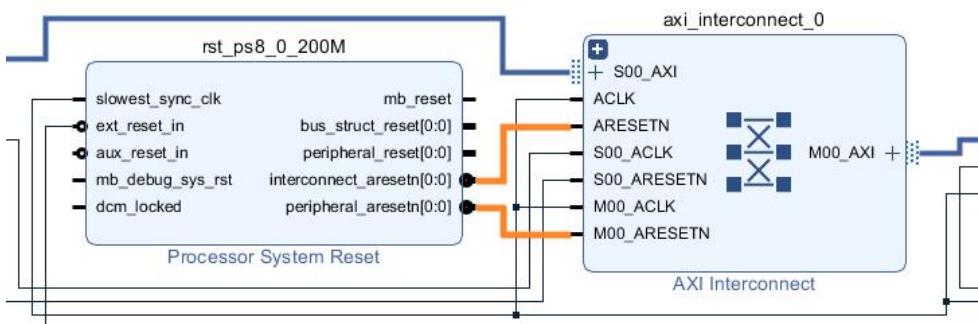
- 2) Connect `axi_aclk` of `xdma` to `S00_ACLK` and `axi_aresetn` to `S00_ARESETN`



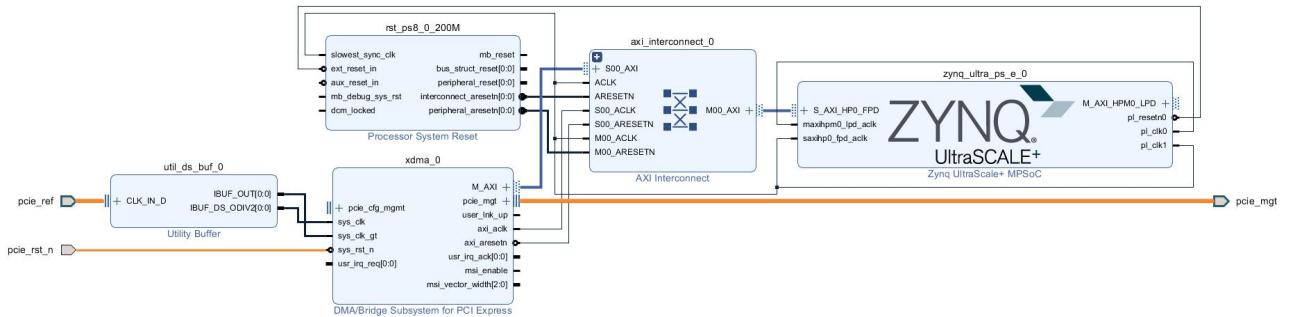
- 3) Connect other clocks



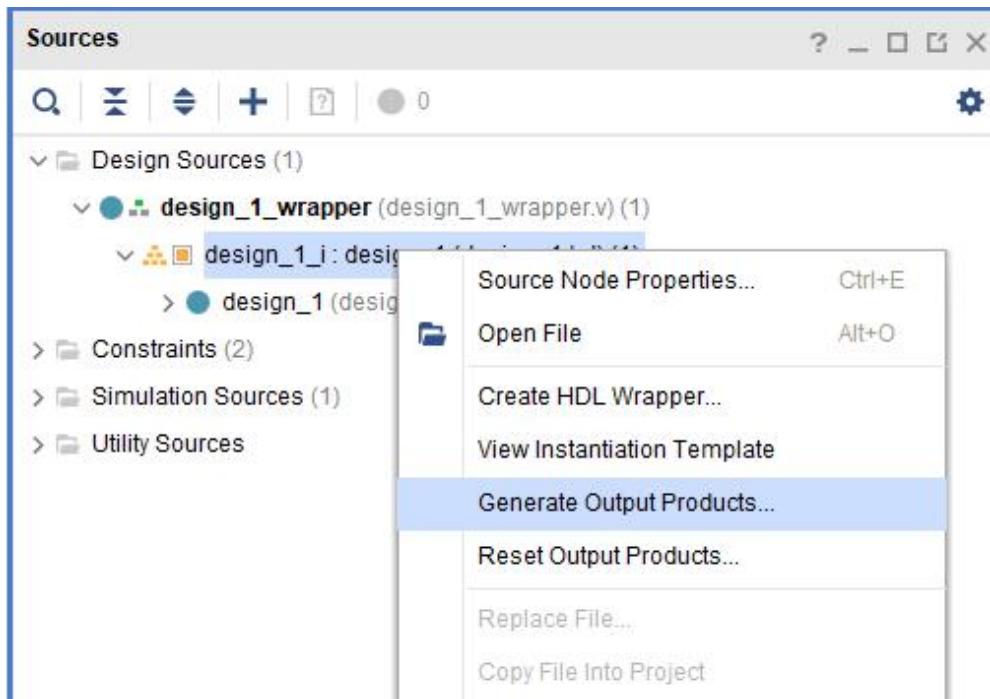
- 4) Add a reset module



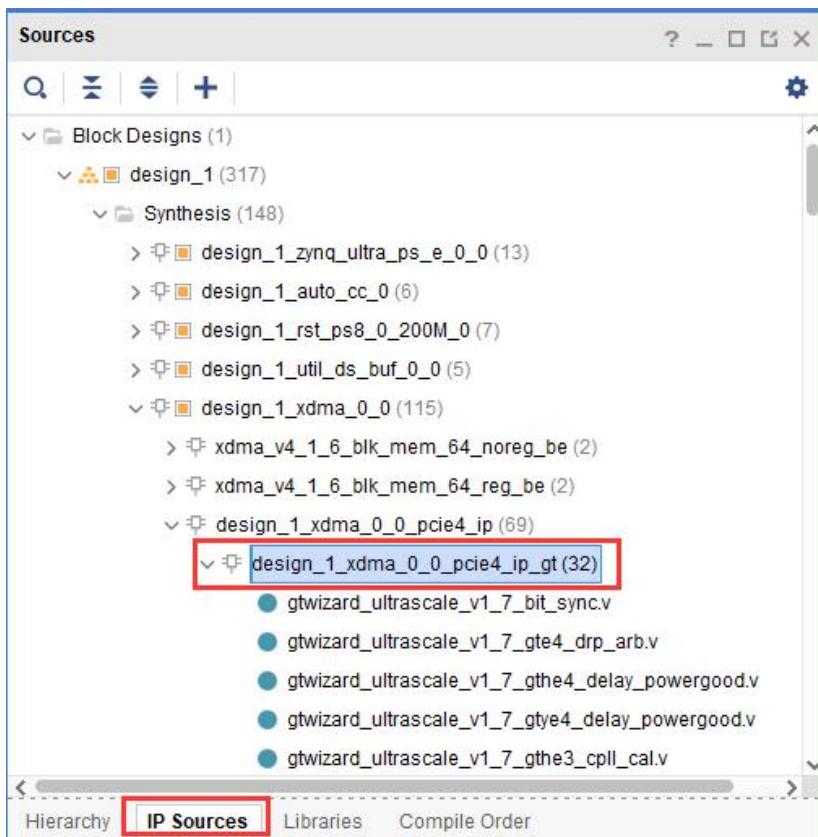
5) Export pin



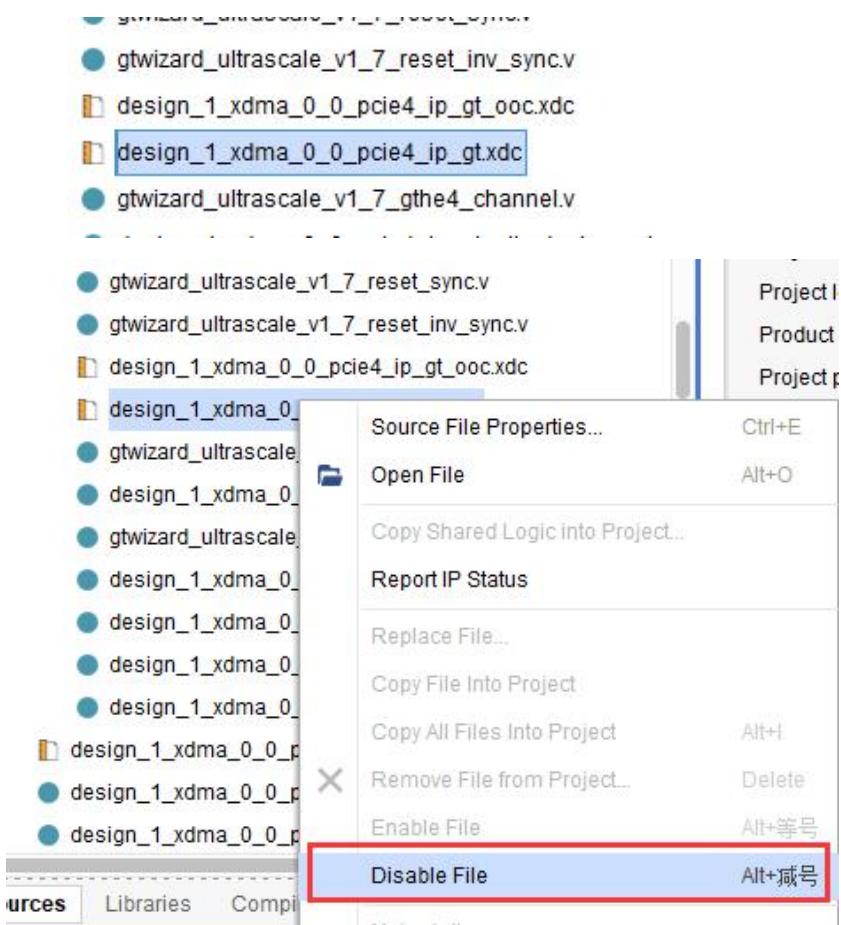
6) Save the design, press F6 to check the design, and [Generate Output Products...](#)



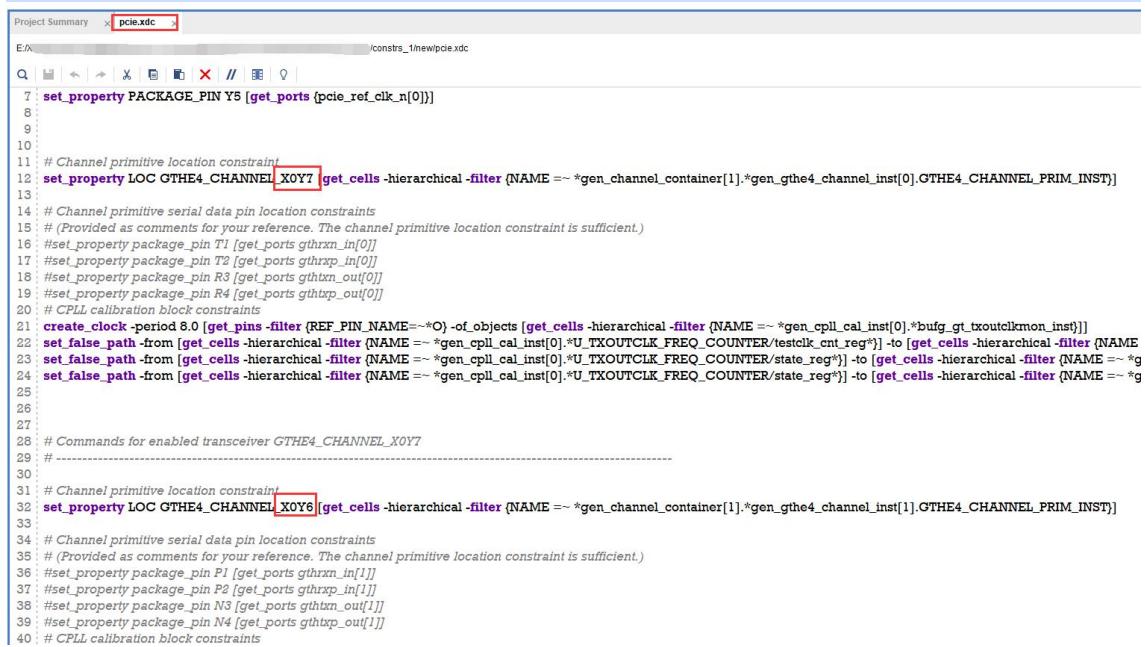
7) Since the pins automatically bound by the software are reversed from the hardware design, it is necessary to modify the pin binding, expand the red box part below in IP Sources



8) Find the file below, right click **Disable file**



- 9) Copy the content in the file, and copy the newly created xdc file, adjust the PCIe pin position as follows, save the file, generate a bit file, and export the hardware information



```

Project Summary  pcie.xdc
E:/.../constrs_1/newip/pcie.xdc

7 set_property PACKAGE_PIN Y5 [get_ports {pcie_ref_clk_n[0]}]
8
9
10
11 # Channel primitive location constraint
12 set_property LOC GTHE4_CHANNEL_XOY7 [get_cells -hierarchical -filter {NAME =~ *gen_channel_container[1].*gen_gthe4_channel_inst[0].GTHE4_CHANNEL_PRIM_INST}]
13
14 # Channel primitive serial data pin location constraints
15 # (Provided as comments for your reference. The channel primitive location constraint is sufficient.)
16 #set_property package_pin T1 [get_ports gthrxn_in[0]]
17 #set_property package_pin T2 [get_ports gthrxp_in[0]]
18 #set_property package_pin R3 [get_ports gthtxn_out[0]]
19 #set_property package_pin R4 [get_ports gthtxp_out[0]]
20
21 # CPLL calibration block constraints
22 create_clock -period 8.0 [get_pins -filter {REF_NAME=~"O"} -of_objects [get_cells -hierarchical -filter {NAME =~ *gen_cpll_cal_inst[0].*bufg_gt_txoutclkmon_inst}]]]
23 set_false_path -from [get_cells -hierarchical -filter {NAME =~ *gen_cpll_cal_inst[0].*U_TXOUTCLK_FREQ_COUNTER/testclk_cnt_reg}]-to [get_cells -hierarchical -filter {NAME =~ *gen_cpll_cal_inst[0].*U_TXOUTCLK_FREQ_COUNTER/state_reg}]-from [get_cells -hierarchical -filter {NAME =~ *gen_cpll_cal_inst[0].*U_TXOUTCLK_FREQ_COUNTER/state_reg}]-to [get_cells -hierarchical -filter {NAME =~ *gen_cpll_cal_inst[0].*U_TXOUTCLK_FREQ_COUNTER/testclk_cnt_reg}]
24 set_false_path -from [get_cells -hierarchical -filter {NAME =~ *gen_cpll_cal_inst[0].*U_TXOUTCLK_FREQ_COUNTER/state_reg}]-to [get_cells -hierarchical -filter {NAME =~ *gen_cpll_cal_inst[0].*U_TXOUTCLK_FREQ_COUNTER/testclk_cnt_reg}]
25
26
27
28 # Commands for enabled transceiver GTHE4_CHANNEL_XOY7
29 #
30
31 # Channel primitive location constraint
32 set_property LOC GTHE4_CHANNEL_XOY6 [get_cells -hierarchical -filter {NAME =~ *gen_channel_container[1].*gen_gthe4_channel_inst[1].GTHE4_CHANNEL_PRIM_INST}]
33
34 # Channel primitive serial data pin location constraints
35 # (Provided as comments for your reference. The channel primitive location constraint is sufficient.)
36 #set_property package_pin P1 [get_ports gthrxn_in[1]]
37 #set_property package_pin P2 [get_ports gthrxp_in[1]]
38 #set_property package_pin N3 [get_ports gthtxn_out[1]]
39 #set_property package_pin N4 [get_ports gthtxp_out[1]]
40 # CPLL calibration block constraints

```

Part 25.2: Generate and burn BOOT

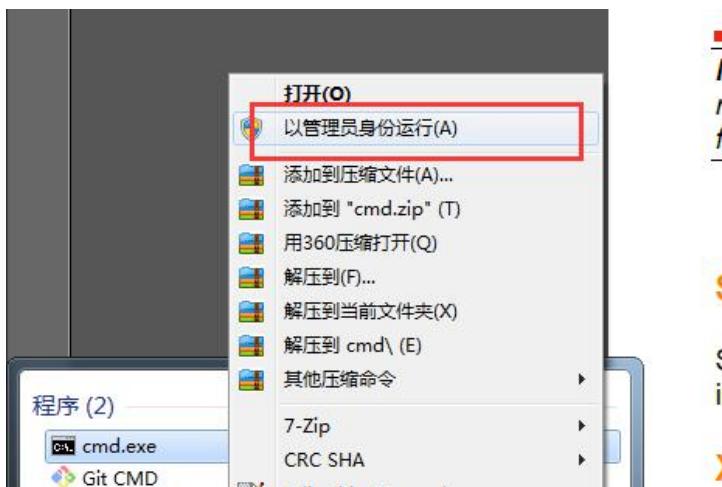
- 1) Export hardware and run vitis
- 2) To create a new app with the "Hello World" template, please refer to the chapter "[Experiencing ARM, outputting "Hello World" on bare metal](#)". PCIe has strict requirements on startup time. Booting using QSPI is faster than booting from SD card, so download [BOOT.bin](#) to QSPI
- 3) After the programming is completed, set the startup mode of the FPGA development board to [QSPI](#) and insert it into the computer's [PCIe](#) slot (power-off operation). At this time, the FPGA development board does not need an external power adapter to supply power, but is powered by the computer motherboard.

Part 25.3: Set the computer to enter the test mode

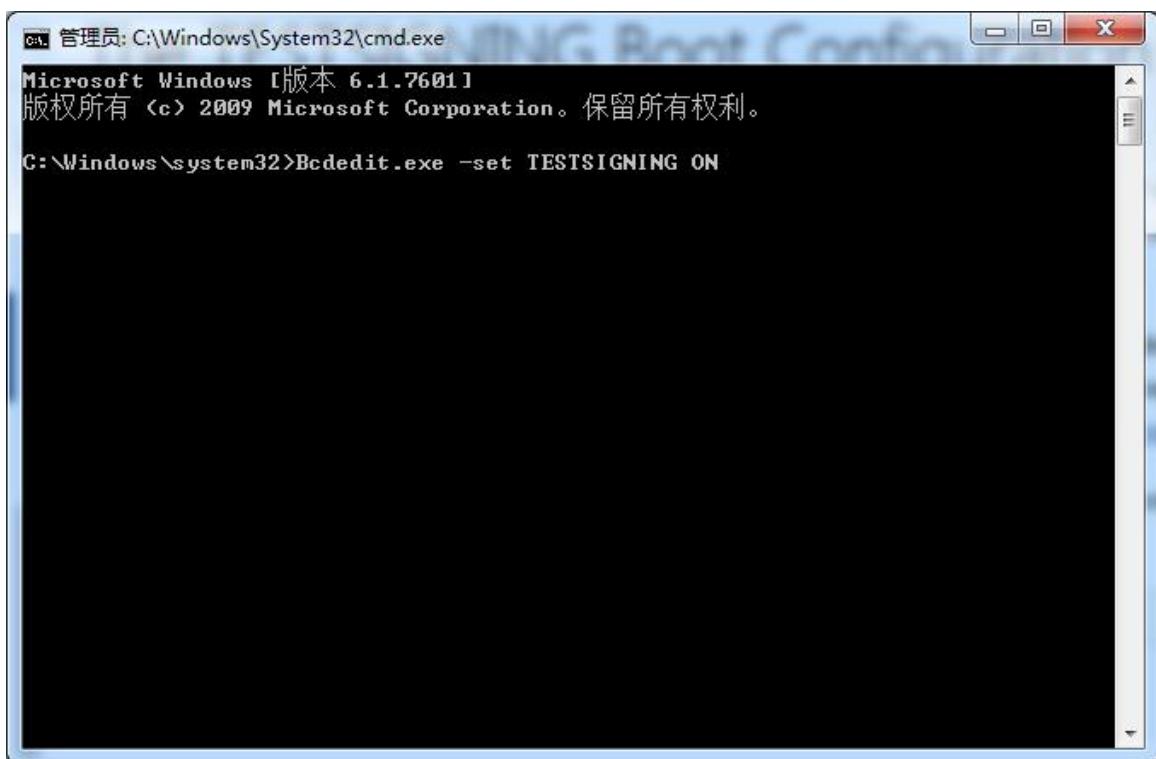
Since the PCIe driver has not been digitally signed by Microsoft, it

can only be used for testing, and the system needs to be set to test mode. Refer to the MSDN document for the detailed method of setting <https://msdn.microsoft.com/en-us/windows/hardware/drivers/install/test-signing-boot-configuration-option>

1) Run cmd as administrator



2) Enter the command `Bcdedit.exe -set TESTSIGNING ON` to open the test mode



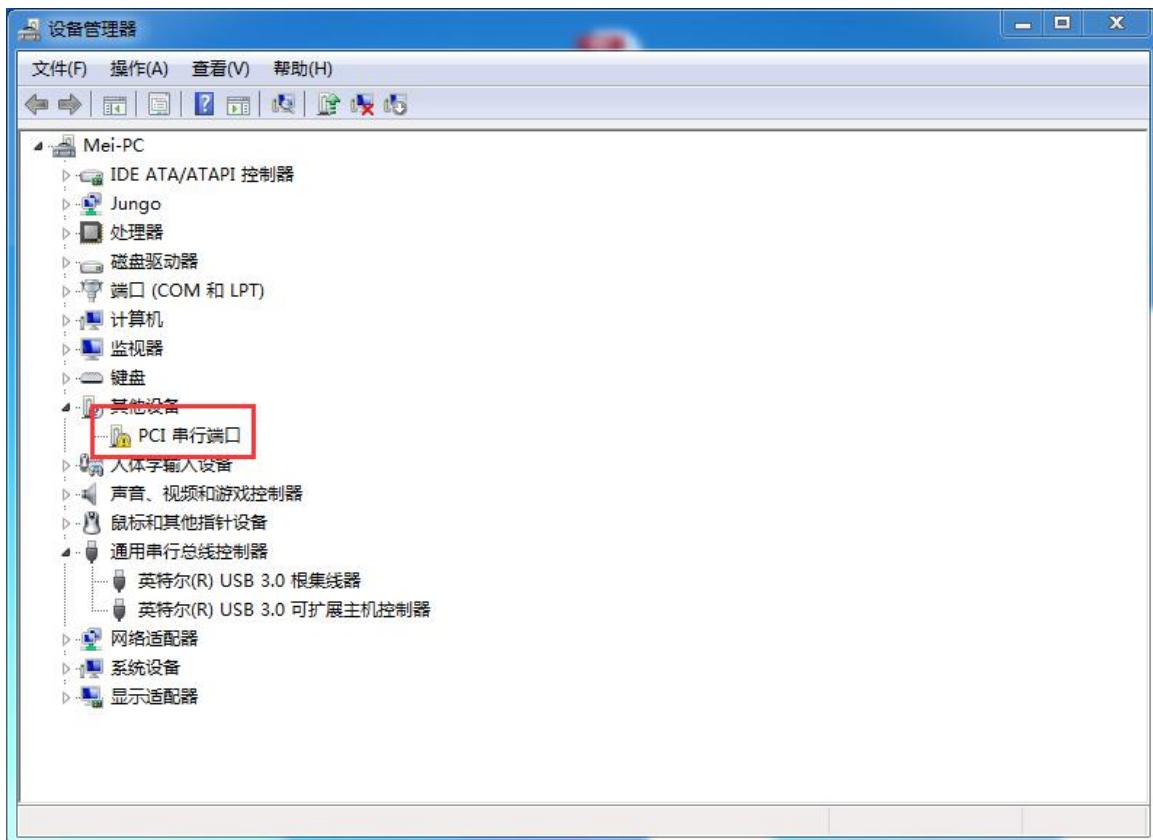
3) After restarting the computer, the desktop shows that the test mode is running



Part 25.4: Install PCIe Driver

The PCIe driver and PC test files are located in "[course_s2/PCIe](#)"

- 1) When the driver is not installed, the device manager is as follows, and a PCI serial port is found



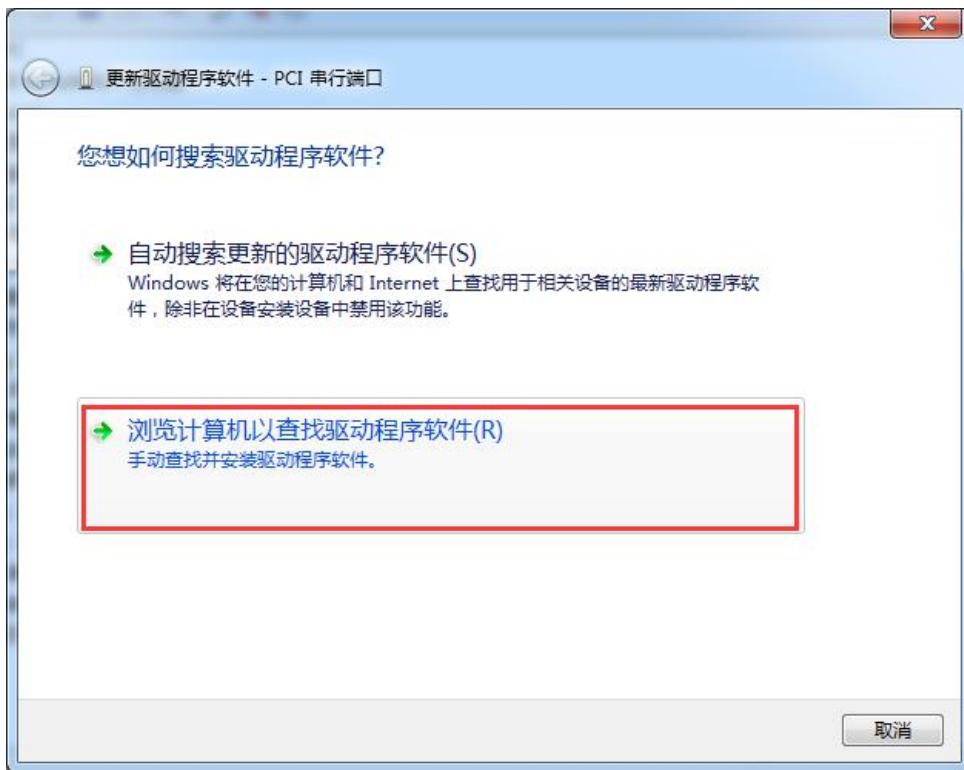
- 2) Compiled driver provided by the example



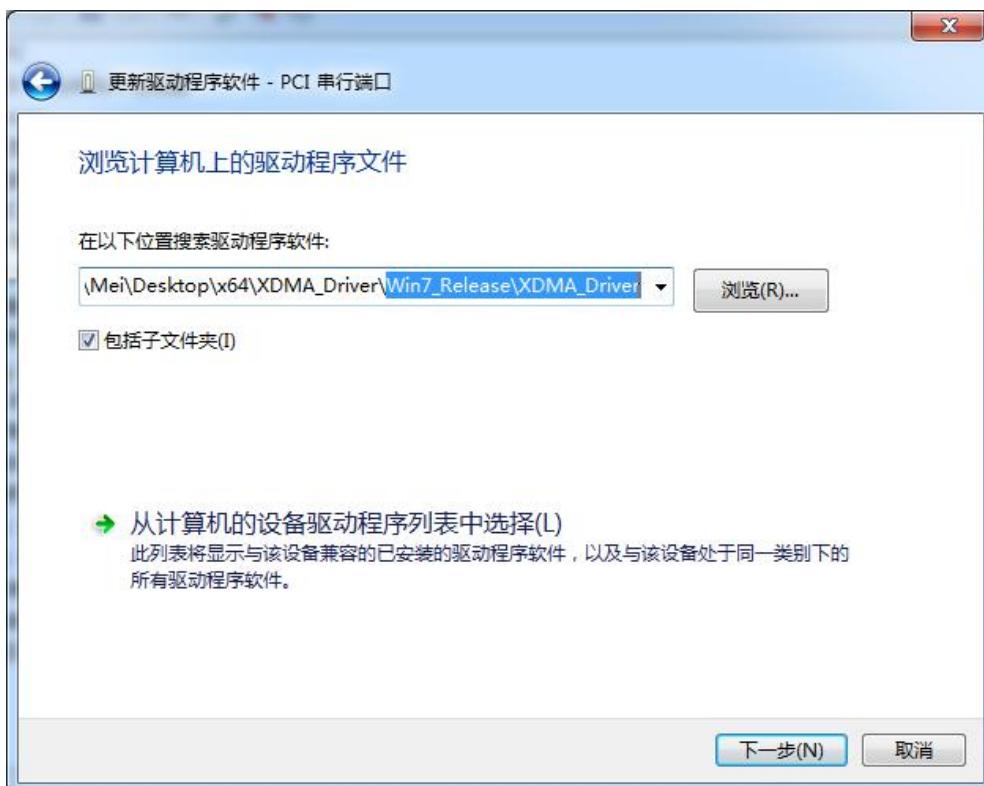
3) Select the device, right click to update the driver



4) Browse the computer for driver software



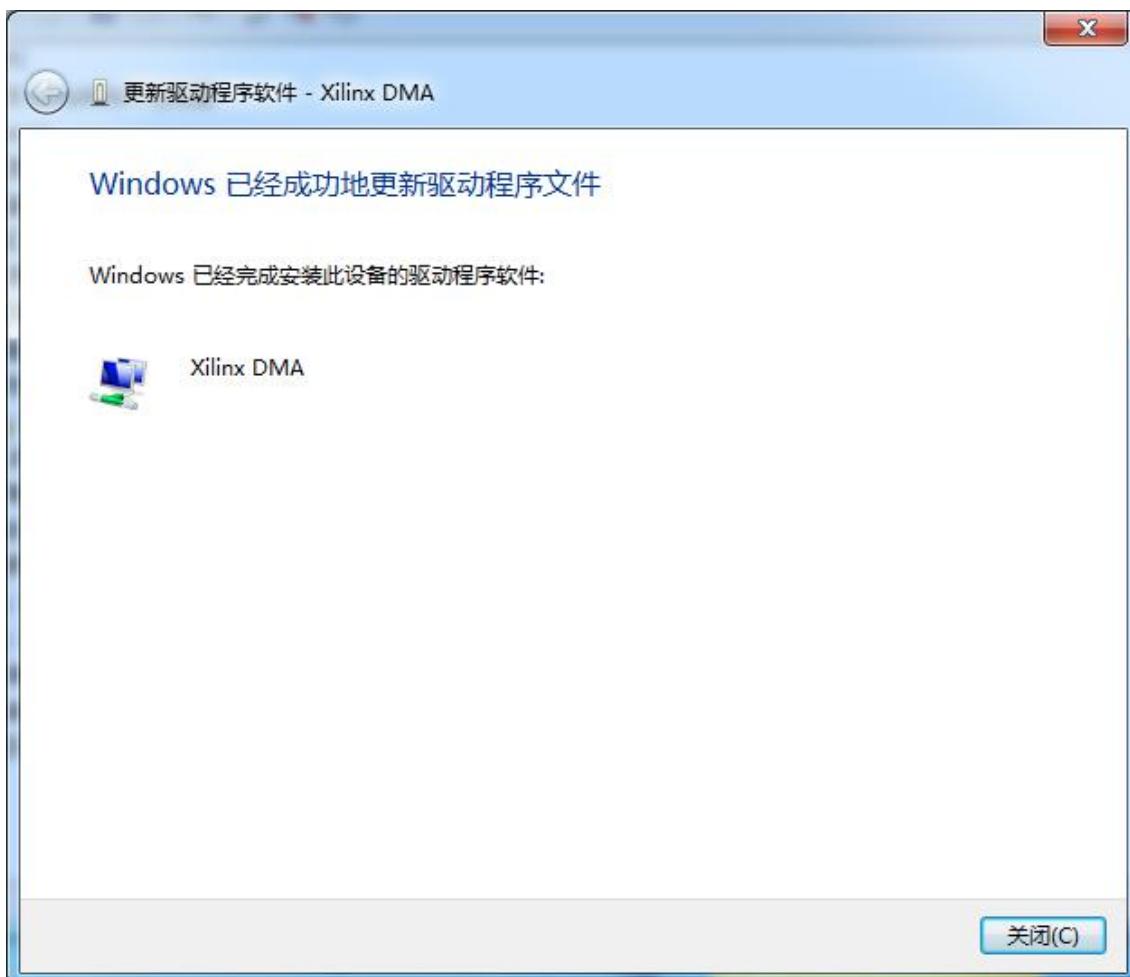
- 5) The test computer is installed with win7, here is the Win7_Release version



- 6) There is a safety warning, choose to install



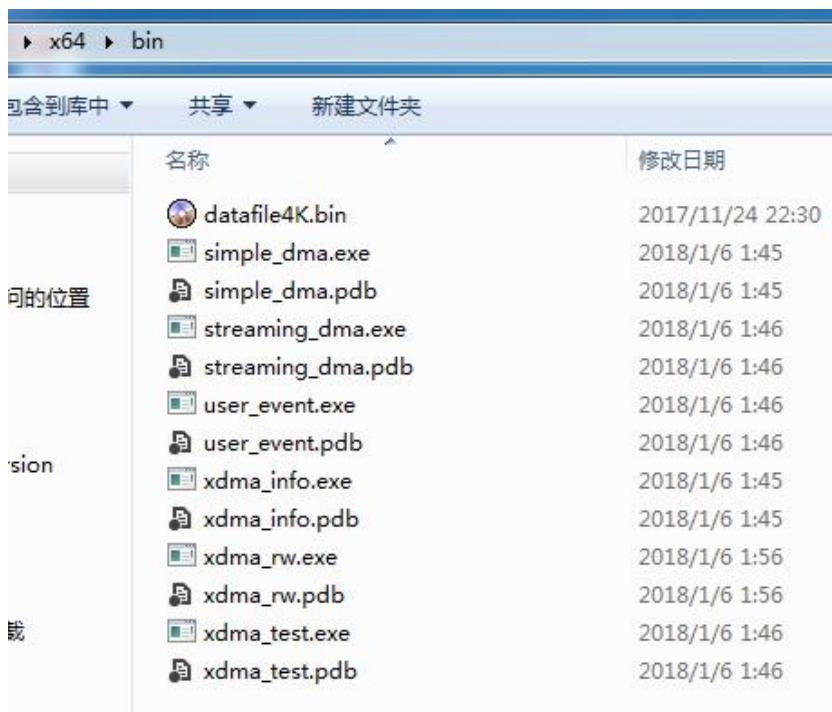
- 7) The device driver is installed correctly



Part 25.5: Testing PCIe

- 1) Xilinx provides some test programs, but they are all command line

programs



- 2) ALINX has developed some applications with interfaces. Use [pciespeed.exe](#) to test the PCIe read and write rate. This read and write test will write data to ZYNQ's ddr and then read it out.
- 3) Test while reading and writing



- 4) Read-only test



5) Write-only test



Part 25.6: Experiment Summary

This experiment is mainly to master the establishment of PCIe hardware engineering and experience the PCIe read and write speed initially. In subsequent experiments, we will do some more practical functions. The host computer software is developed using QT. If you need to modify the compilation yourself, please learn how to use QT first.