ZYNQ MPSoC Development Platform

Vitis Application Tutorial

ALINX

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

Preparation and precautions

Software Environment

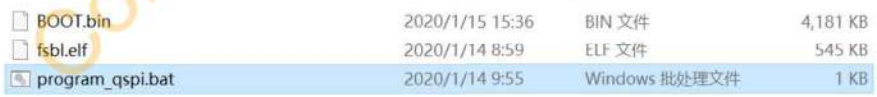
The software development environment is based on Vivado 2020.1

Hardware environment

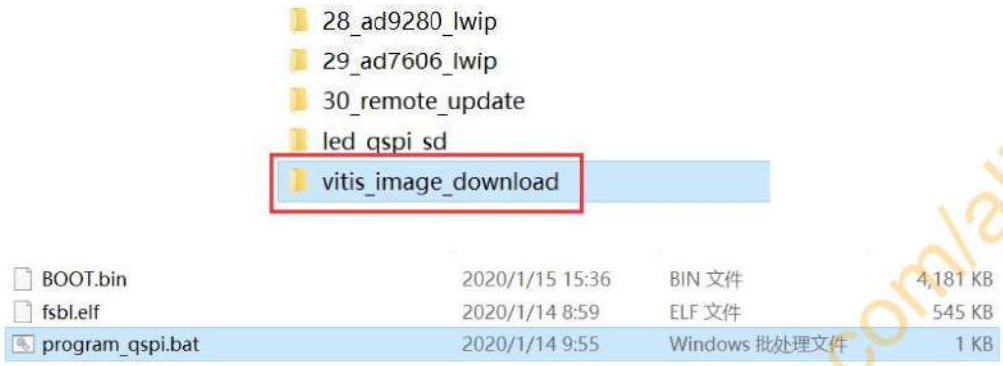
FPGA Development Board Model	FPGA Chip
AXU3EG	Xazu3eg-sfvc784-1-i
AXU4EV	Xczu4ev-sfvc784-1-i
AXU5EV	xazu5ev-sfvc784-1-i

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



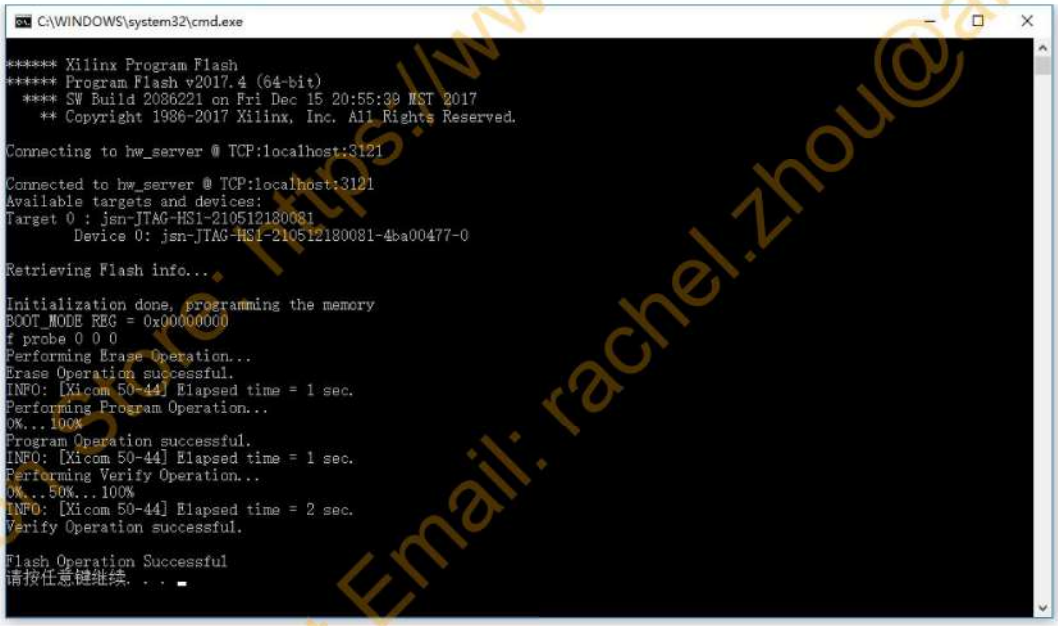
The **vitis_image_download** folder is under the **course_s2** directory, enter the folder, right-click **program_qspi.bat**, and open it for editing



Change the **program_flash** path to your own software installation path, save and close.

```
call C:\Xilinx\Vitis\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.

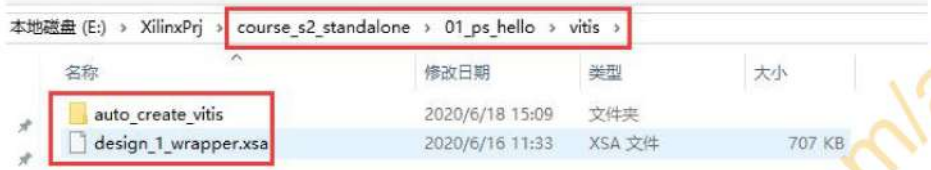


You can also use the SD card boot method to copy the **BOOT.bin** file to the SD to boot.

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

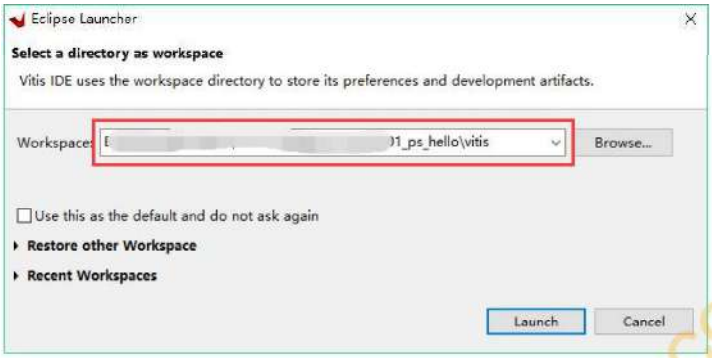
Replace the **xsct.bat** path in the yellow box with the path installed by yourself, the path is **xx\Vitis\2020.1\bin\xsct.bat**

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

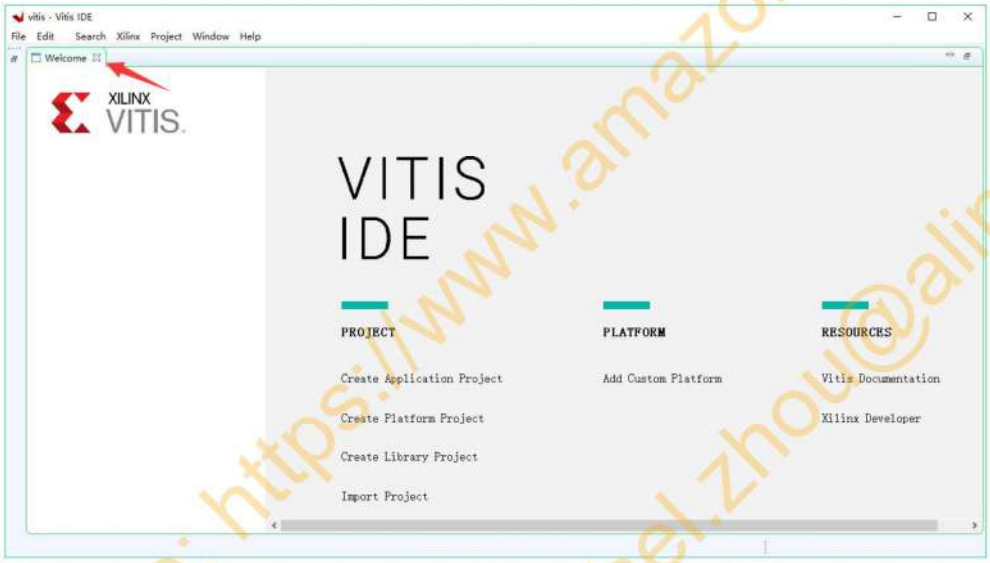
After saving, double-click **build_vitis.bat** to create the project

Compilation is over, press any key to exit

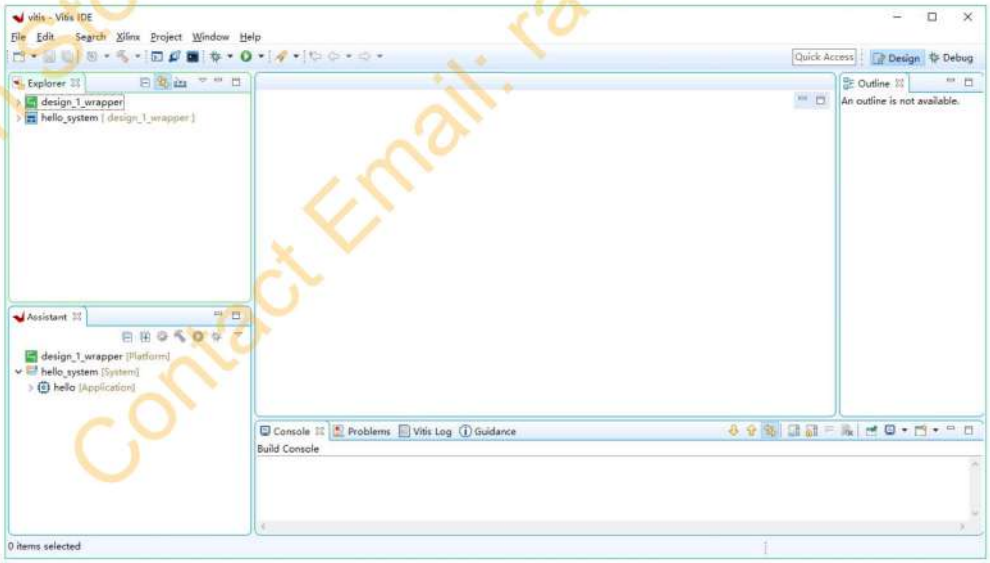
Open the **Vitis** software, select the project path, **Launch**



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

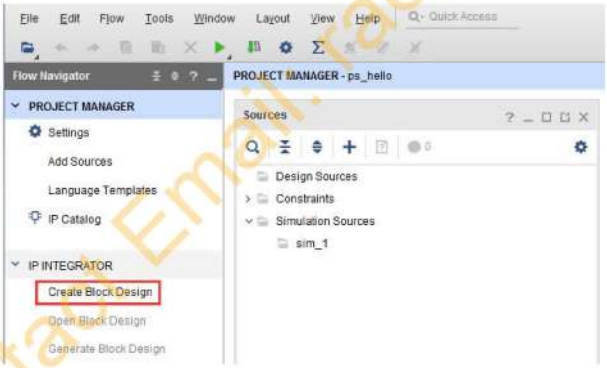
divided into PL and PS. The IO assignment on the PS side is relatively fixed and cannot be arbitrarily assigned, and there is no need to assign pins in the Vivado software. Although this experiment only uses PS, a Vivado project must also be established to configure the PS pins. Although the ARM on the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

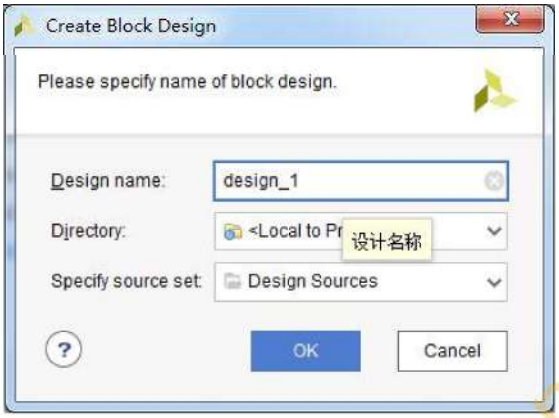
The following is the content that FPGA engineers are responsible for.

Part 1.2: Create a Vivado Project

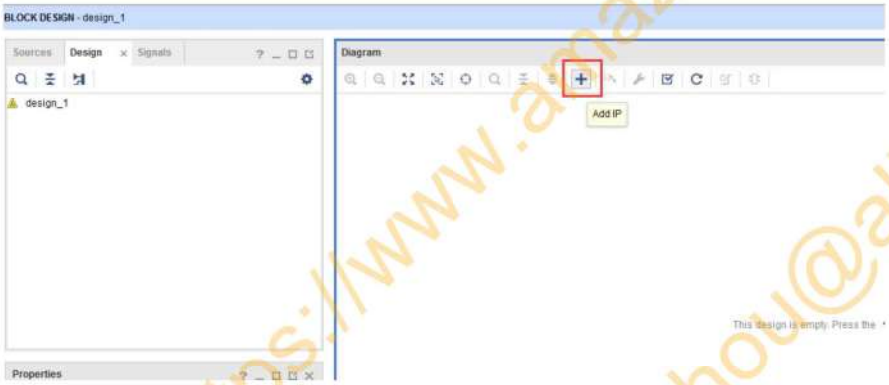
1) Create a project called "ps_hello". The creation process will not be repeated, please refer to "PL's "Hello World" LED Experiment".
2) Click on "Create Block Design" to create a block design, which is a graphical design



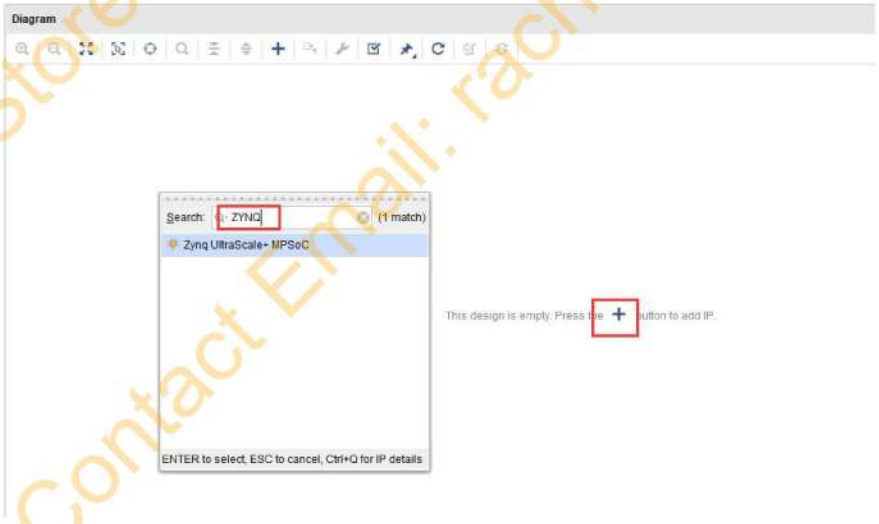
3) "Design name" is not modified here, keep the default "design_1", which can be modified as needed, but the name should be as short as possible, otherwise there will be problems compiling under Windows.



4) Click on the "Add IP" shortcut icon

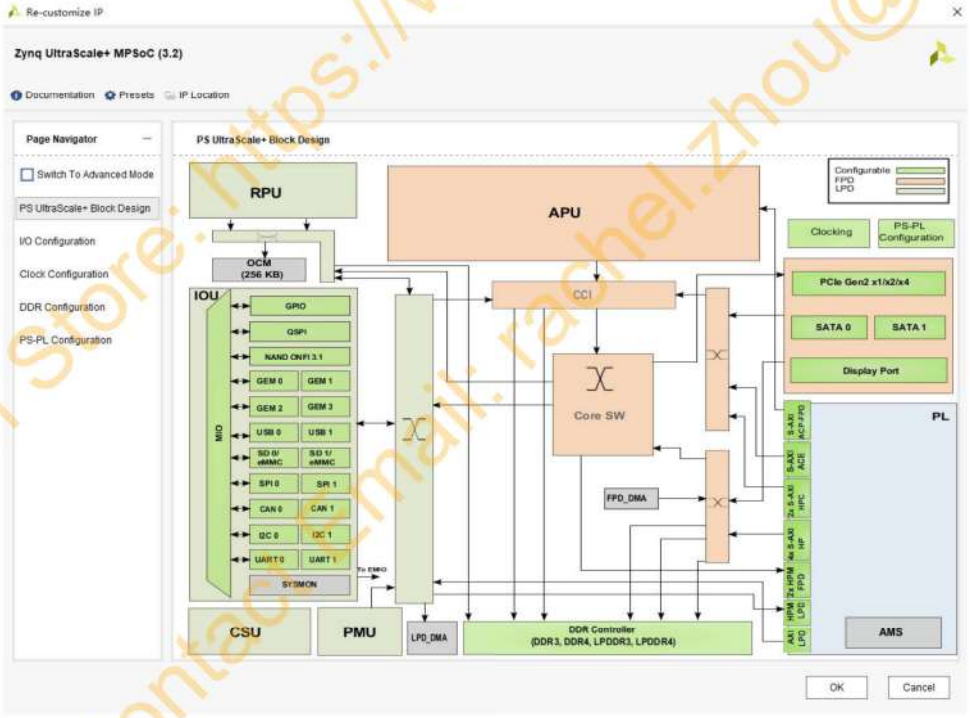


5) Search for "zynq" and double-click "ZYNQ UltraScale+ MPSoC" in the search results list



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

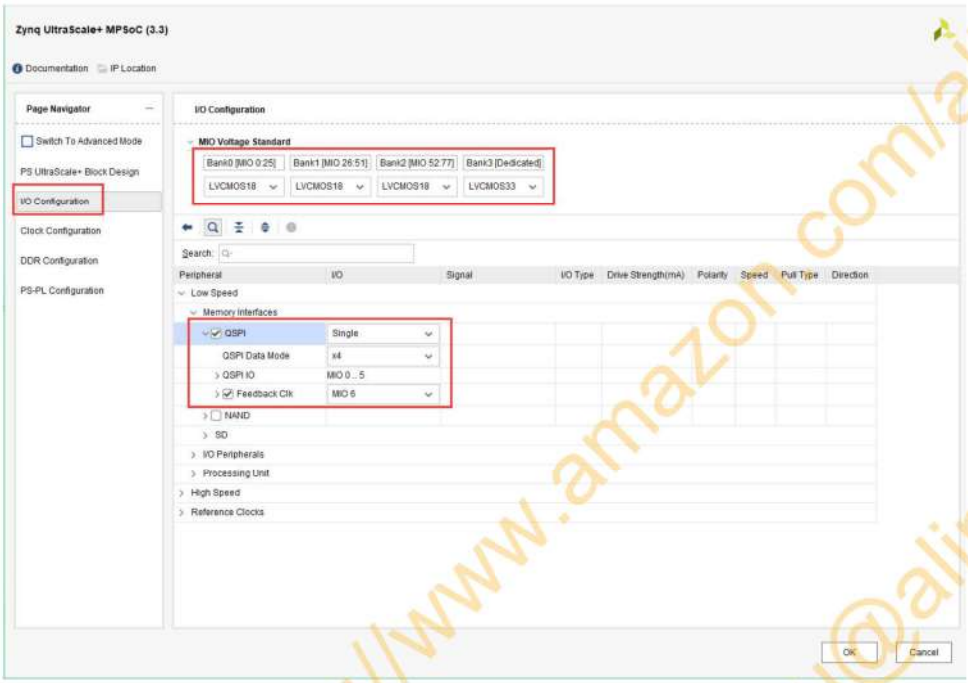
7) The first interface that appears is the ZYNQ hard core architecture diagram. You can clearly see its structure. You can refer to the ug585 document, which contains a detailed introduction to ZYNQ. The green part in the picture is the configurable module. You can click to enter the corresponding editing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



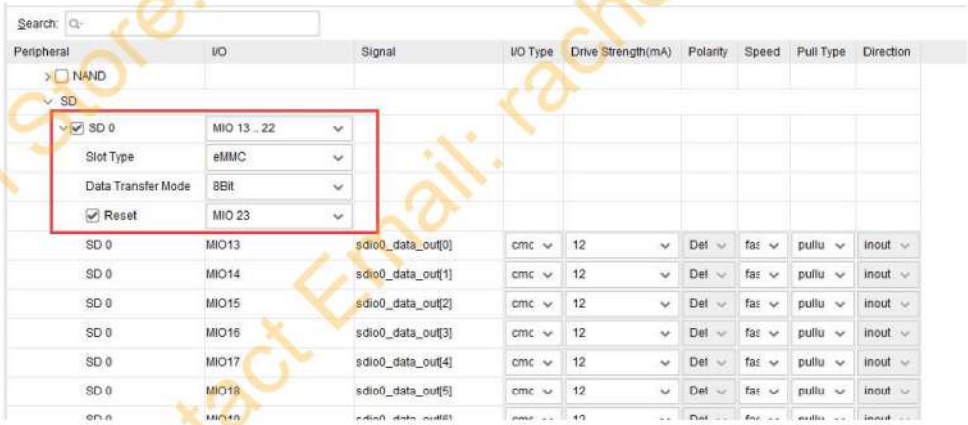
Part 1.2.1: Low Speed Configuration

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

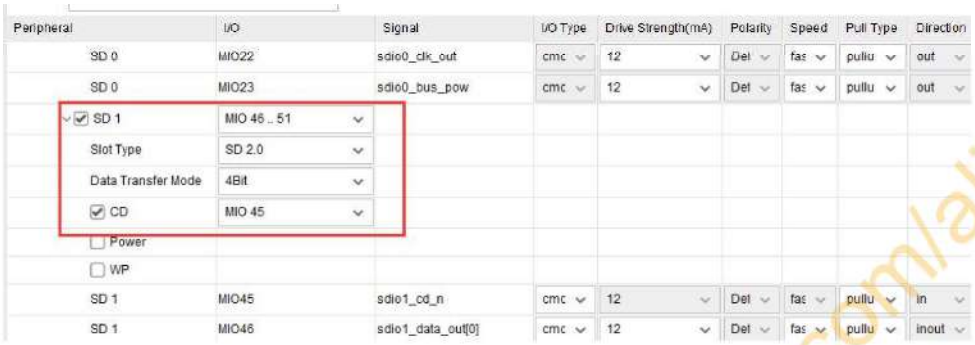
LVCMOS33. First configure Low Speed pin, check QSPI, and set it to "Single" mode, Data Mode is "x4", check Feedback Clk



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

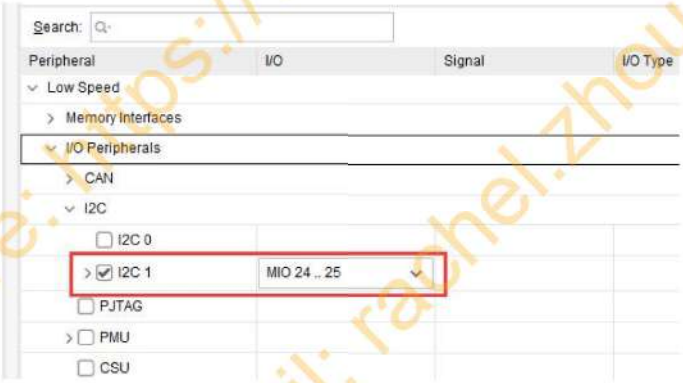


3) Check SD 1 to configure SD card. Select MIO 46..51, Slot Type select SD 2.0, Data Transfer Mode select 4Bit, check CD to detect SD card insertion, select MIO45



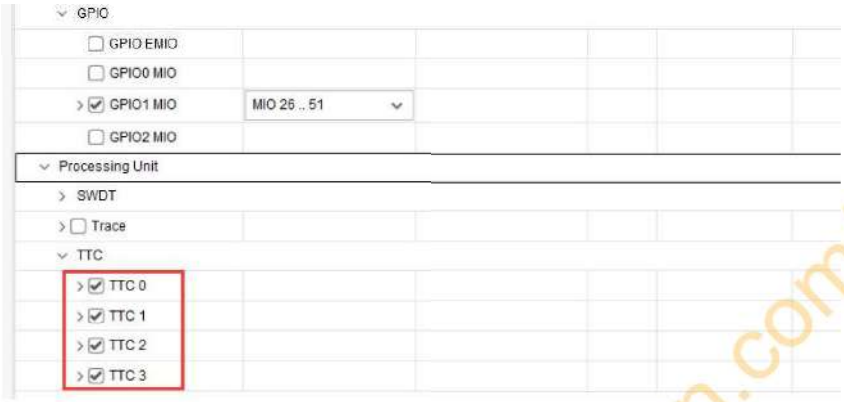
4) Check CAN 0, select MIO 38..39, check CAN 1, select MIO 32..33

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



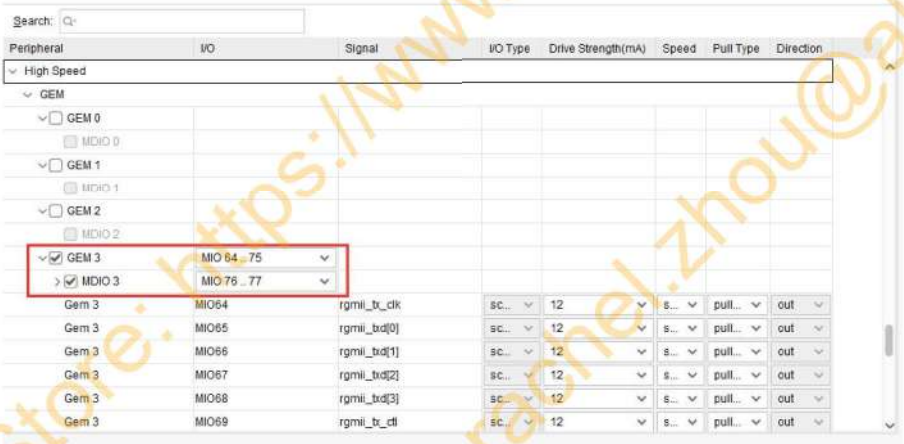
6) Check the serial port UART 0, select MIO 42..43, check GPIO1 MIO

7) Check TTC 0 ~ TTC 3

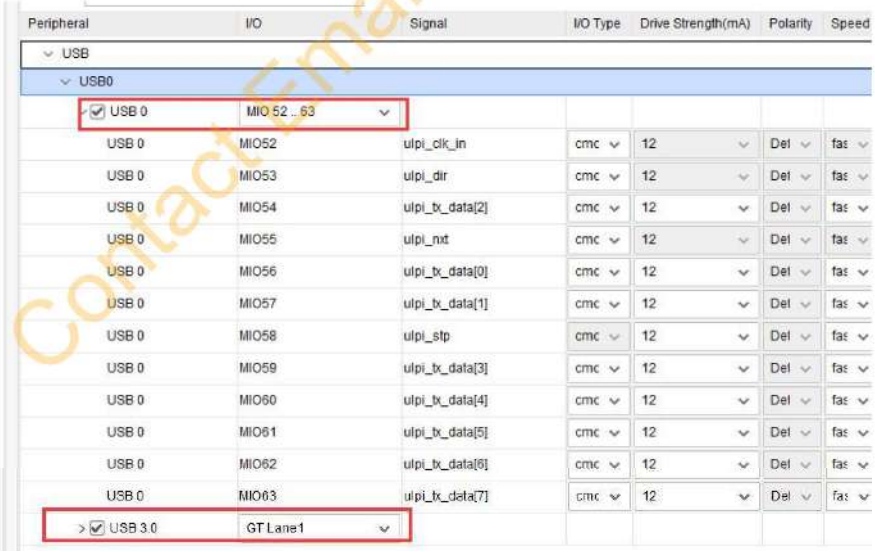


Part 1.2.2: High Speed Configuration

1) 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

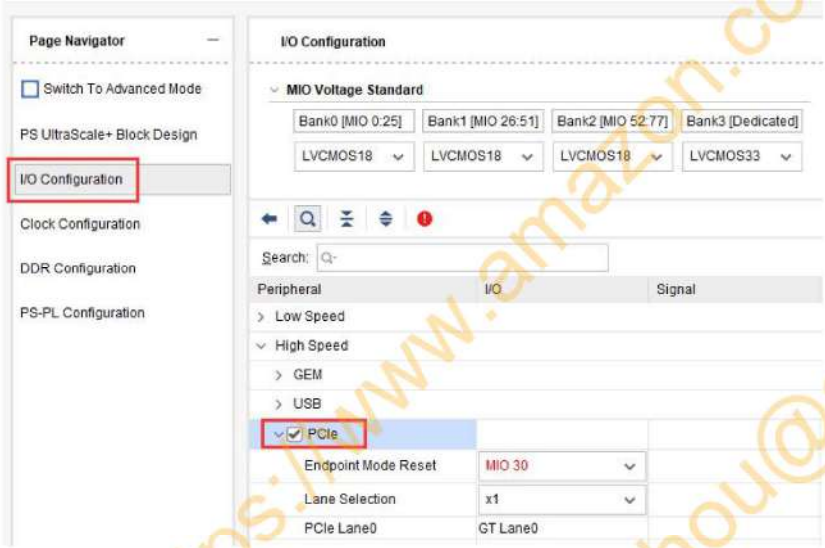


2) Check USB 0, select MIO 52..63, check USB 3.0, select GT Lane1

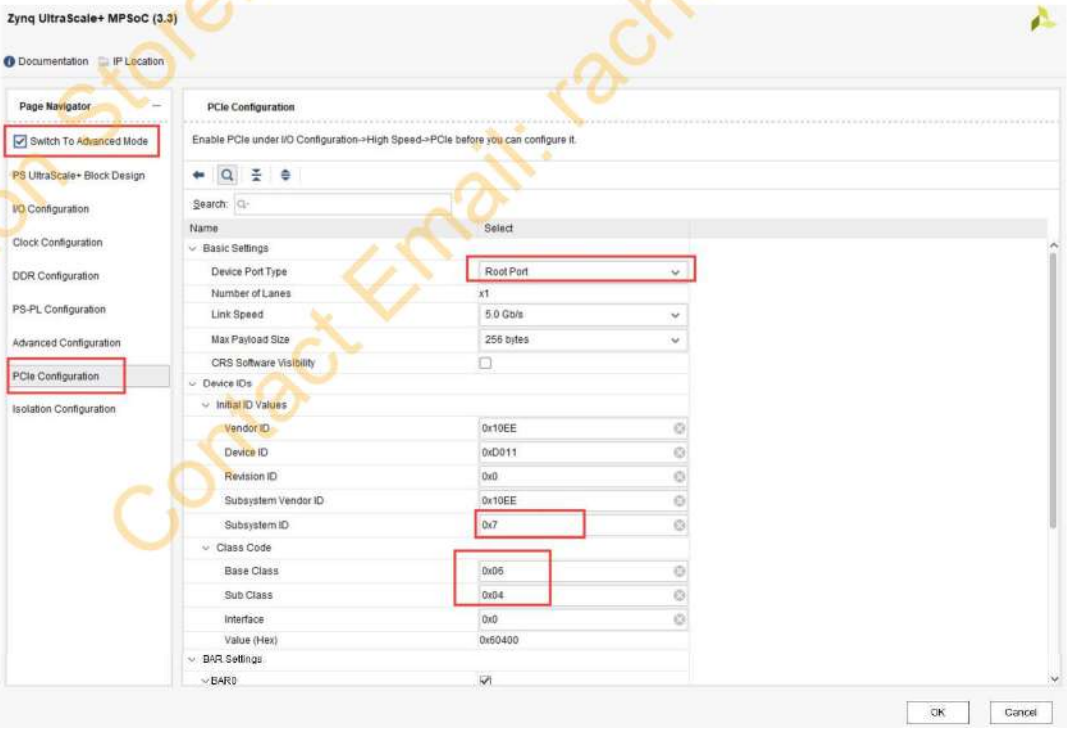


USB reset select MIO 31

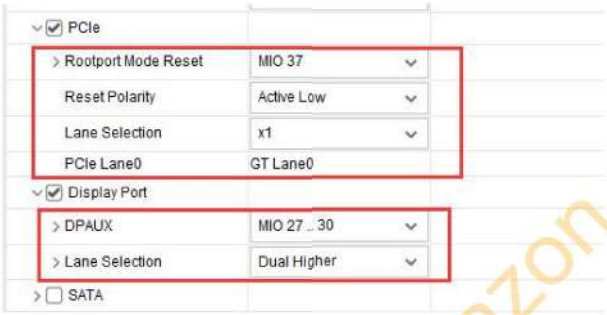
3) Check PCIe



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



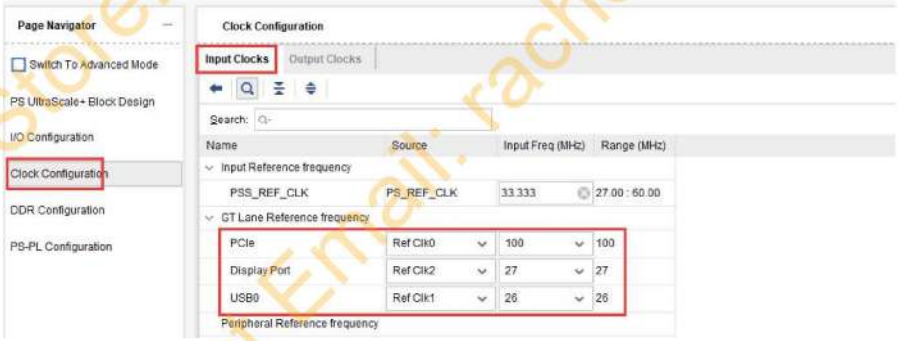
5) Go back to I/O Configuration, reset and select MIO 37; check Display Port, select MIO 27..30, and Lane Selection select Dual Higher



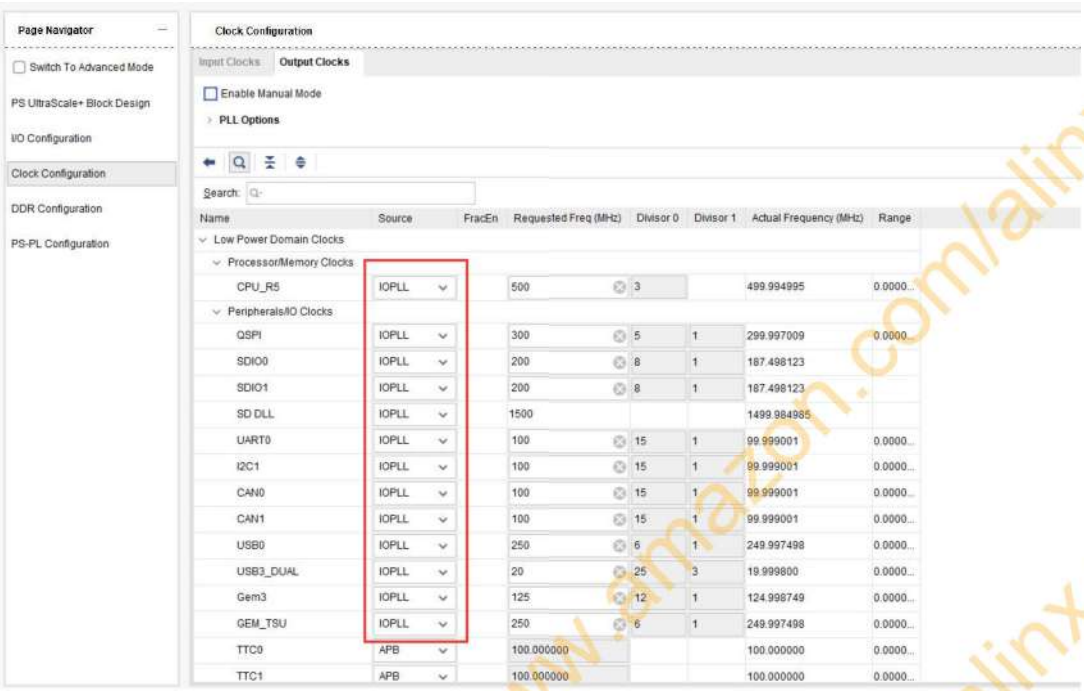
At this point, the I/O part is configured

Part 1.2.3: Clock Configuration

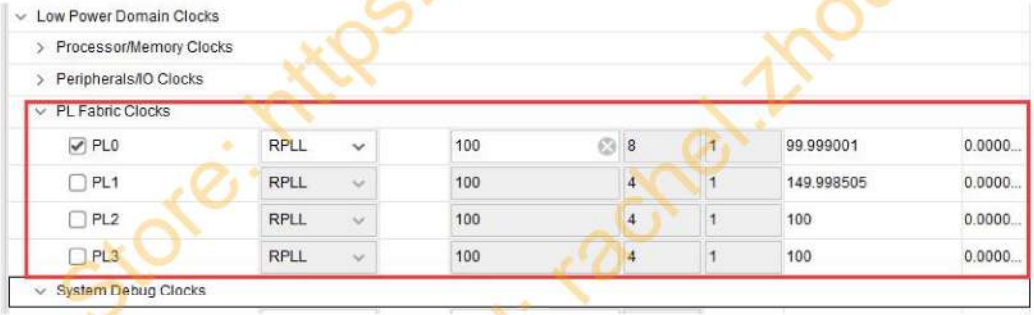
1) In the Clock Configuration interface, the Input Clocks window configures the reference clock, where PSS_REF_CLOCK is the ARM reference clock and the default is 33.333MHz; PCIe selects Ref Clk0, 100MHz; Display Port selects Ref Clk2, 27MHz; USB0 selects Ref Clk1, 26MHz.



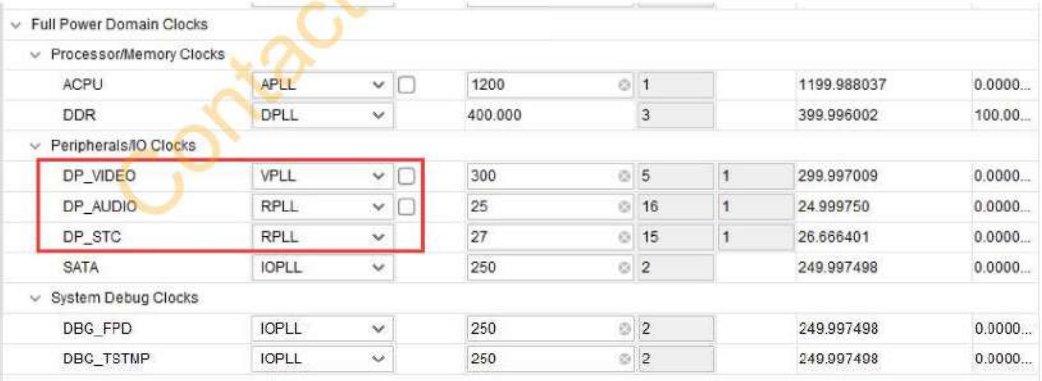
2) In the Output Clocks window, if it is not IOPLL, change to IOPLL, keep the same, use the same PLL



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



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.

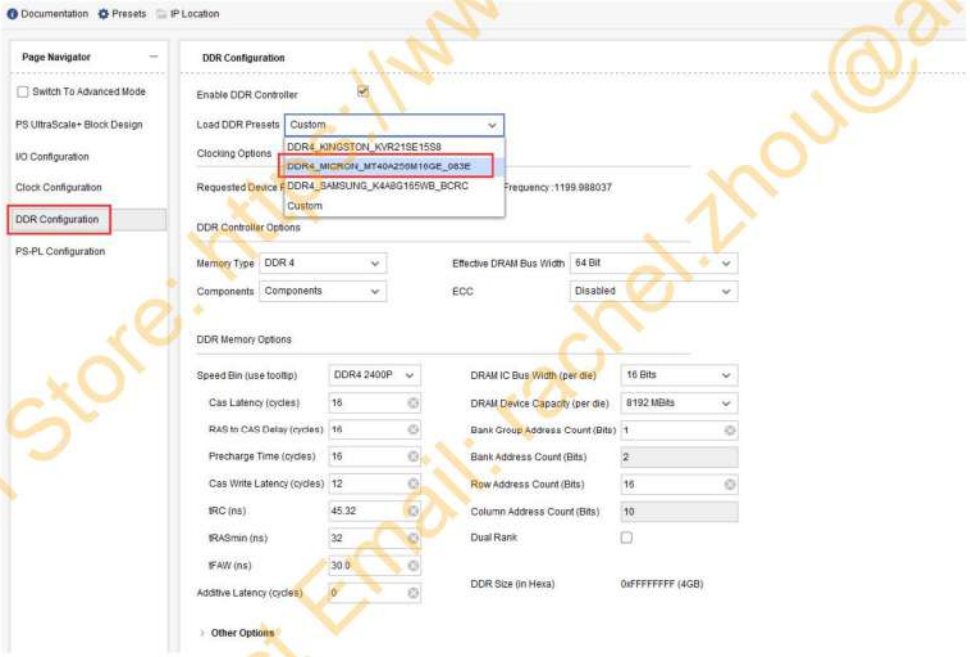


The bottom Interconnect is modified as follows

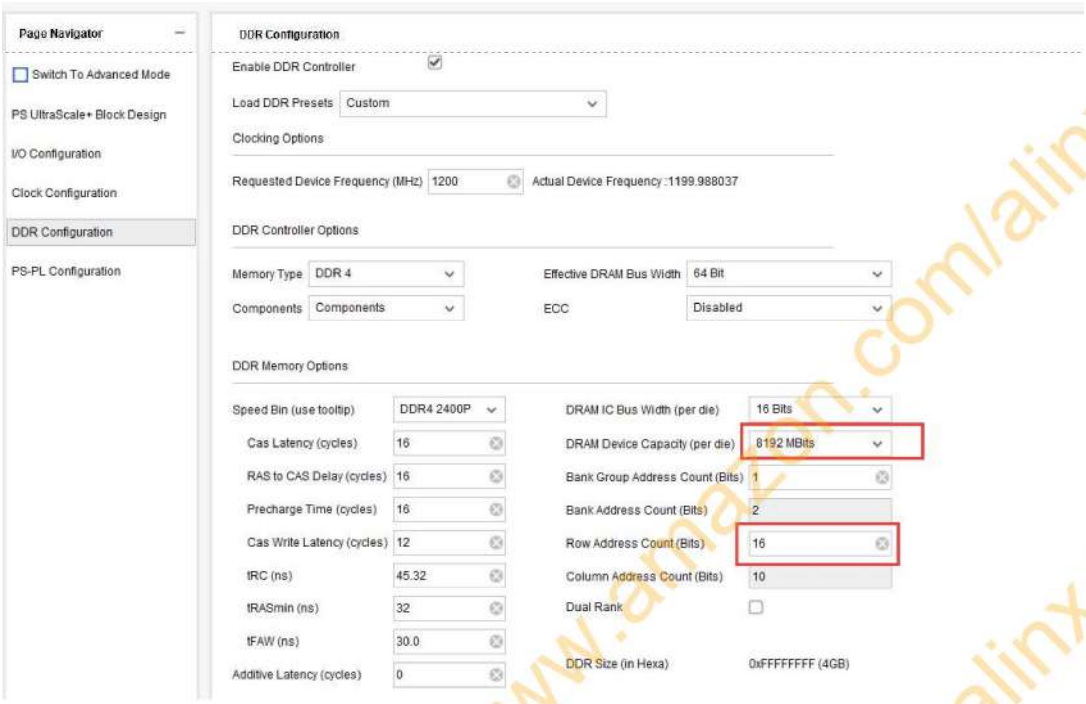
Keep the other parts as default, so far, the clock part configuration is complete.

Part 1.2.4: DDR Configuration

1) In the DDR Configuration window, select Load DDR Presets "DDR4_MICRON_MT40A256M16GE_083E"

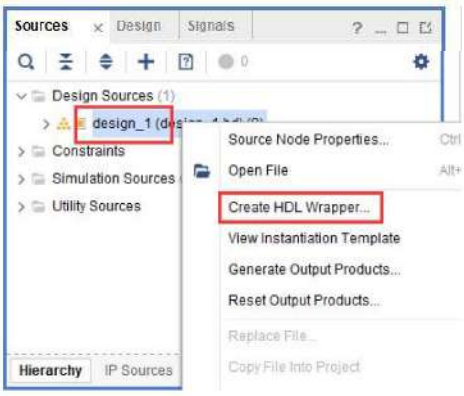


The parameters are modified as follows

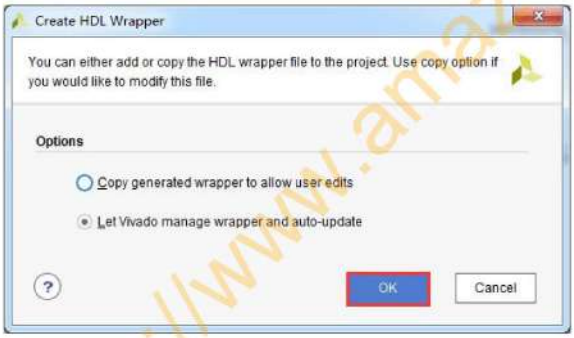


Keep the others as default, click OK, the configuration is complete, and connect the clock as follows:

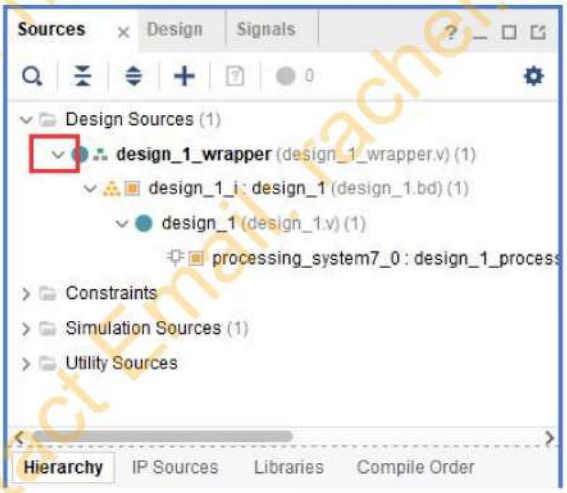
2) Select Block design, right click "Create HDL Wrapper...", create a Verilog or VHDL file, and generate HDL top-level file for block design.



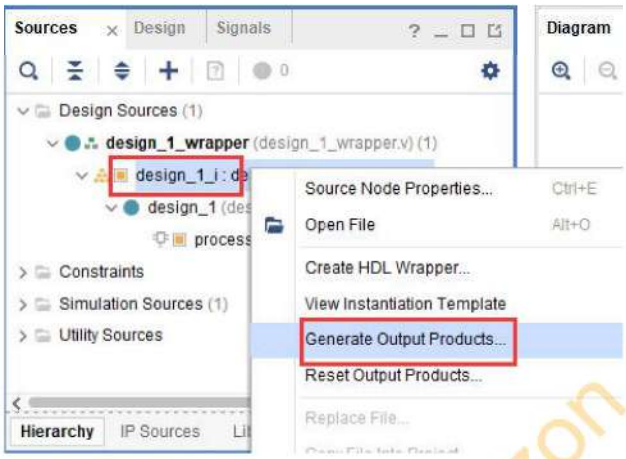
3) Keep the default options and click "OK"



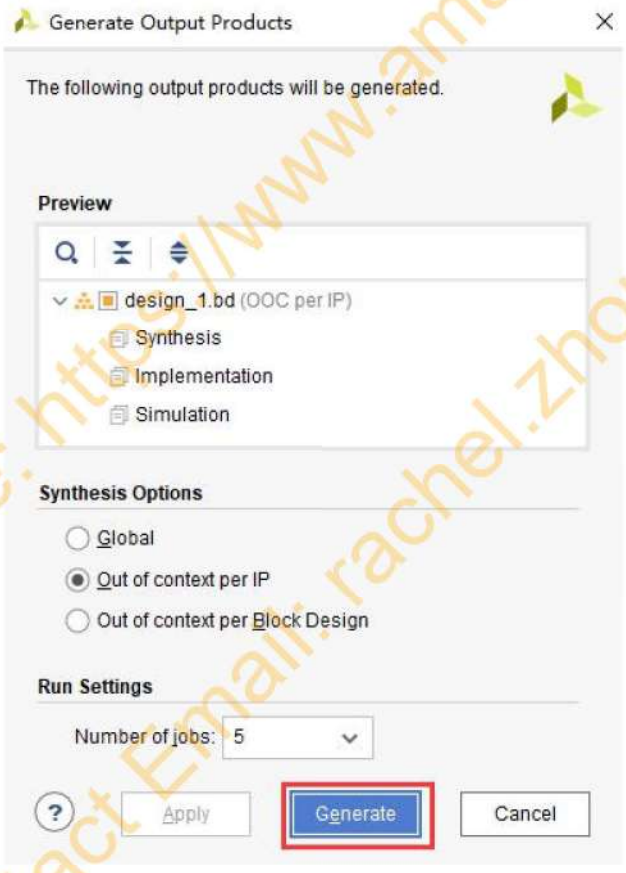
4) Expand the design and you can see that the PS is used as an ordinary IP.



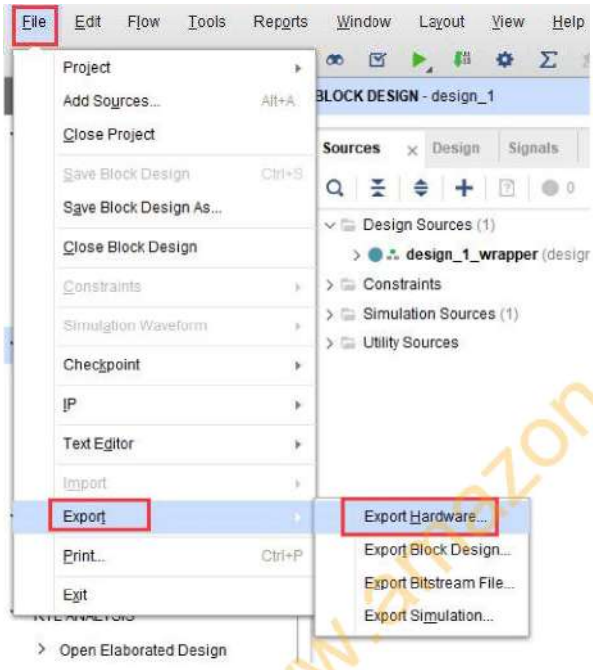
5) Select the block design and right click "Generate Output Products". This step will generate block output files, including IP, instantiation templates, RTL source files, XDC constraints, third-party integrated source files, and so on. For subsequent operations.



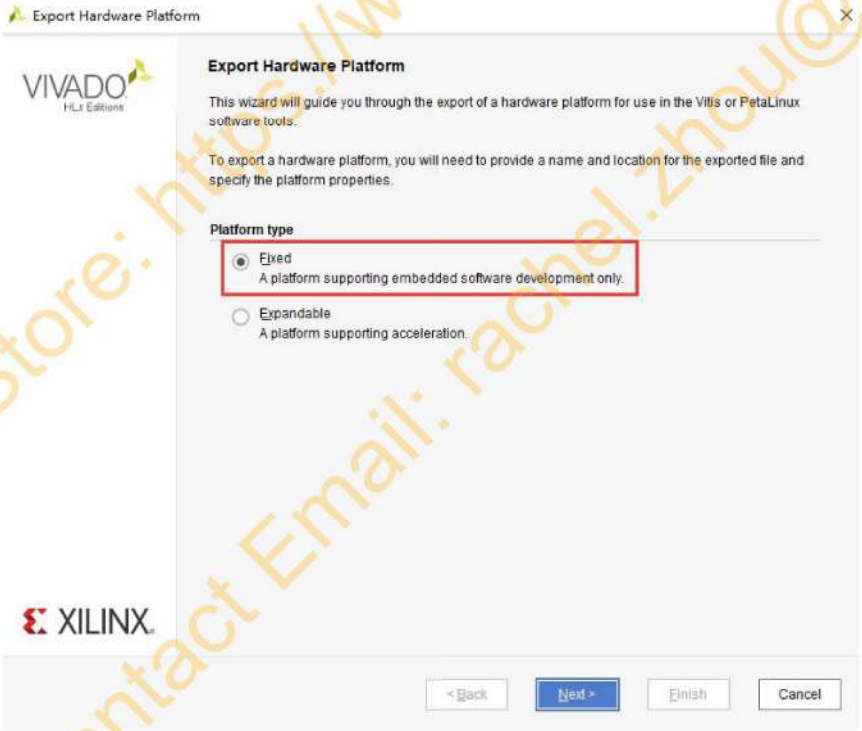
6) Click "Generate"



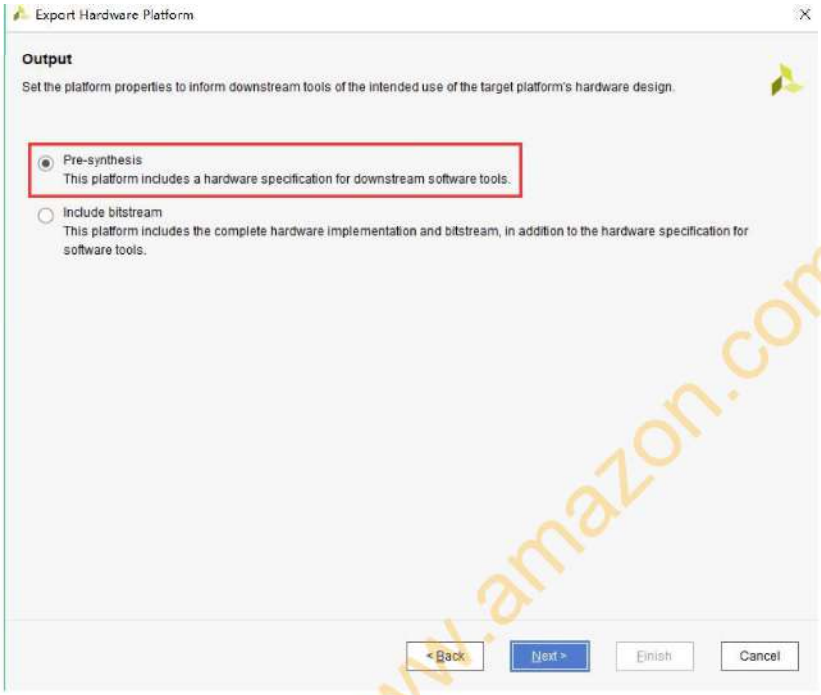
7) Export the hardware information in the menu bar "File -> Export -> Export Hardware...", here it contains the configuration information of the PS side.



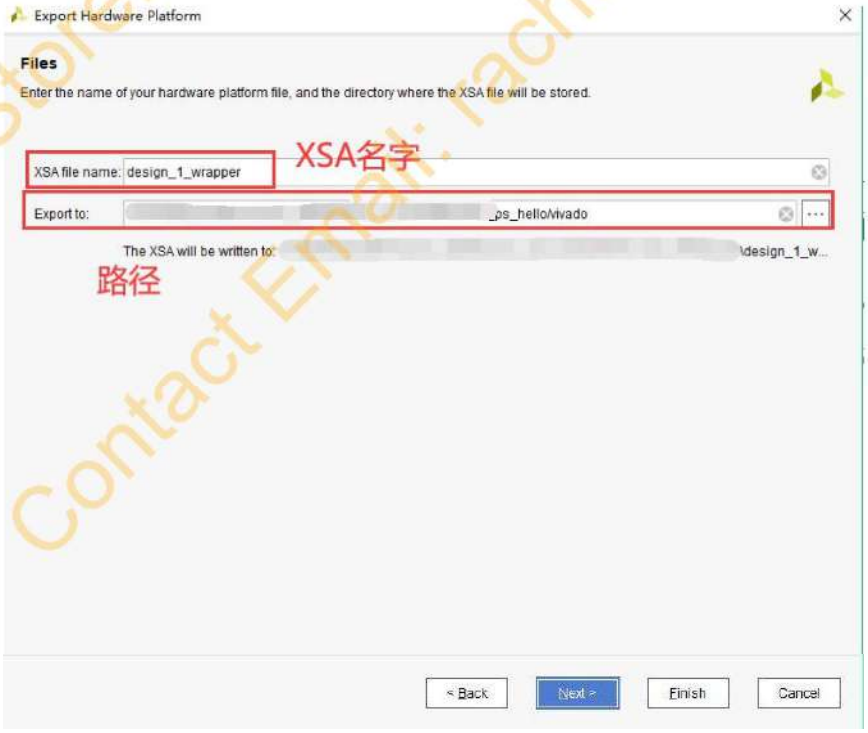
8) Select Fixed in the pop-up window, click Next



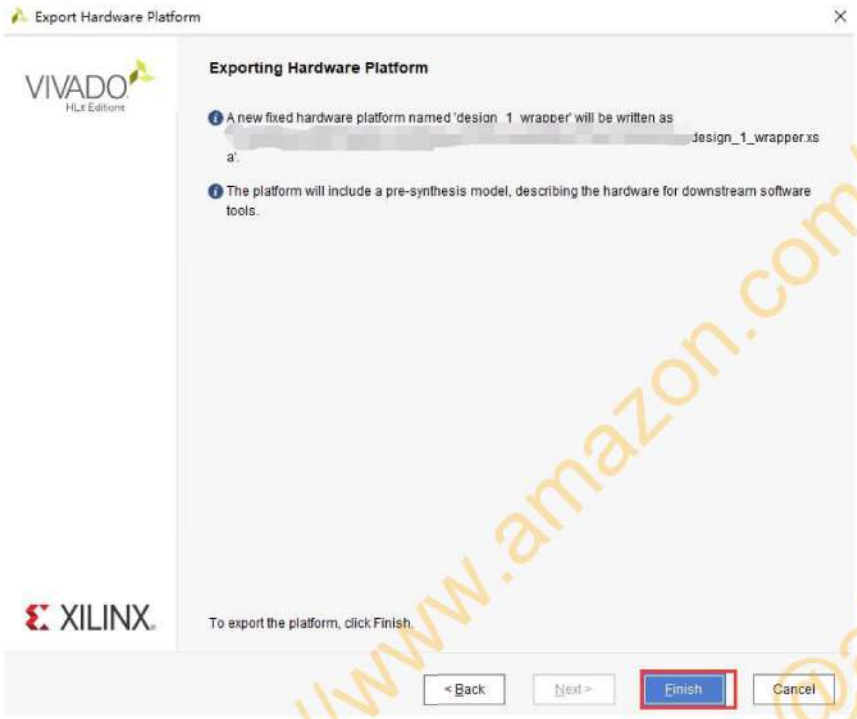
9) Click "OK" in the pop-up dialog box, because the experiment only uses the PS serial port and does not require PL to participate. There is no enable here. Do not select "Include bitstream", click Next



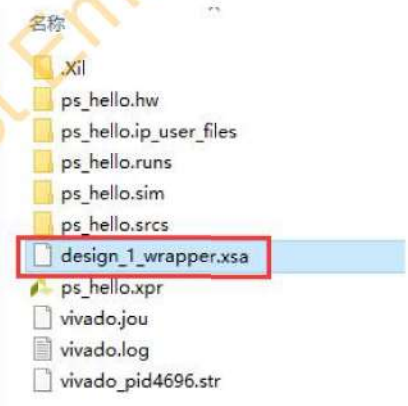
10)The export name and export path can be modified. The default is in the vivado project directory. This file can be placed in a suitable location according to your needs. It does not have to be placed under the vivado project. The vivado and vitis software are independent. Here we choose not to change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

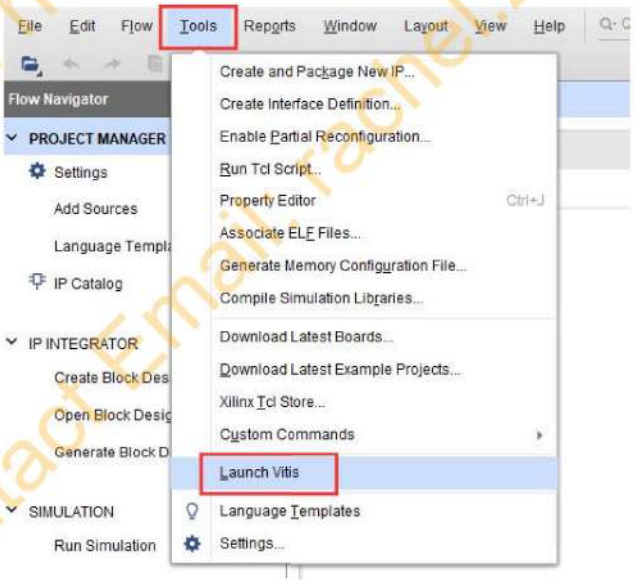
The following is the content that FPGA engineers are responsible for.

Part 1.3: Vitis Debugging

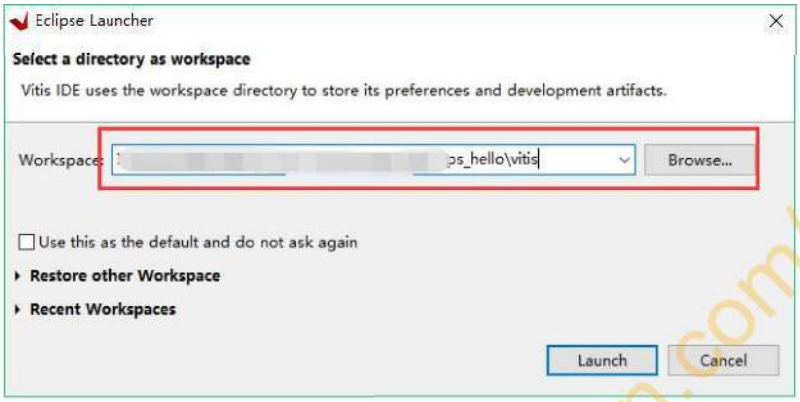
Part 1.3.1: Create Application Project

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

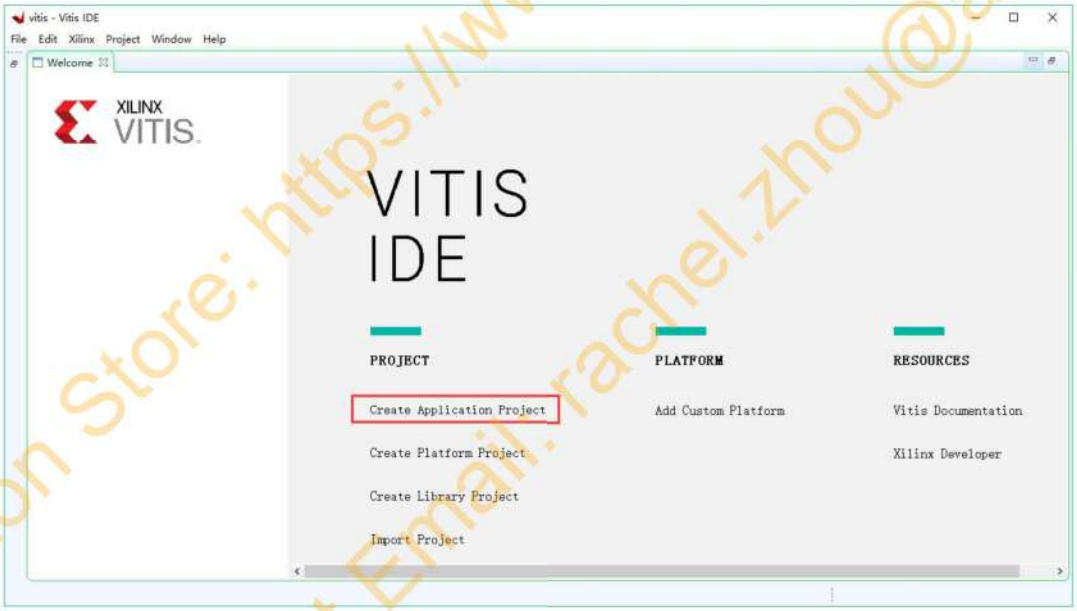
You can also open the Vitis software by selecting Tools → Launch Vitis in the Vivado software



Select the newly created folder and click "Launch"

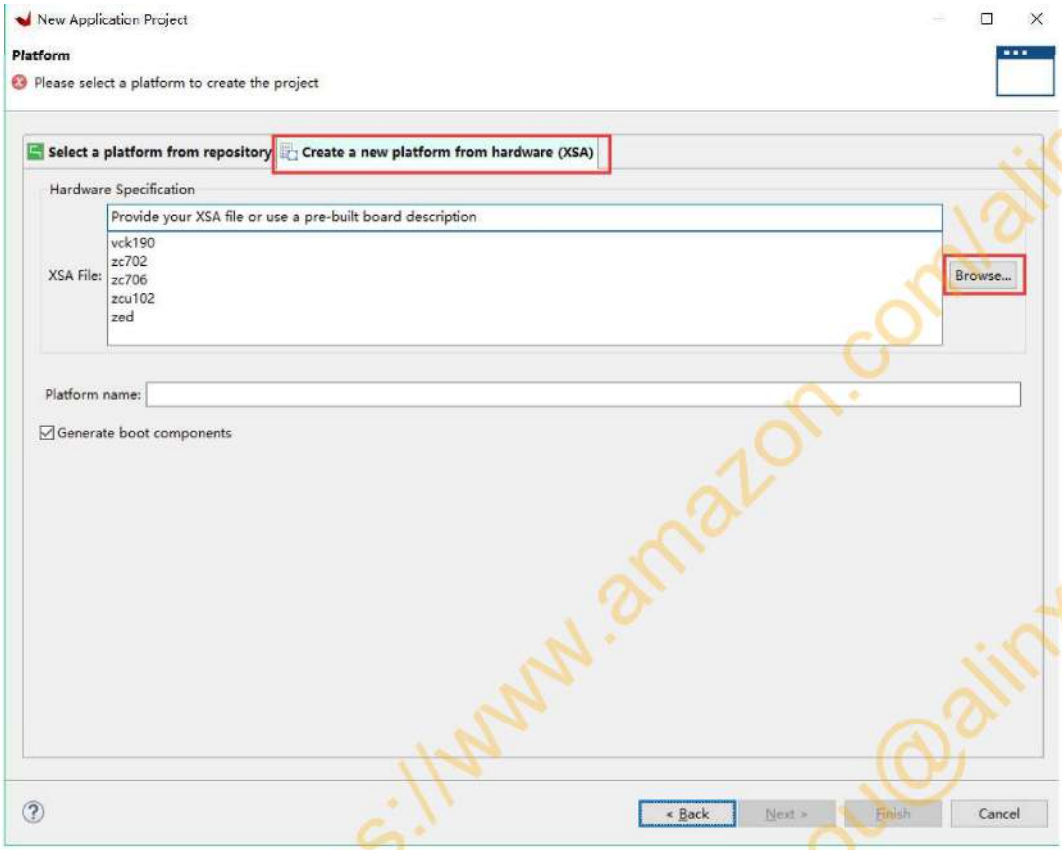


3) After starting Vitis, the interface is as follows, click "Create Application Project", this option will generate APP project and Platfrom project, Platform project is similar to the previous version of hardware platform, including hardware support related files and BSP.

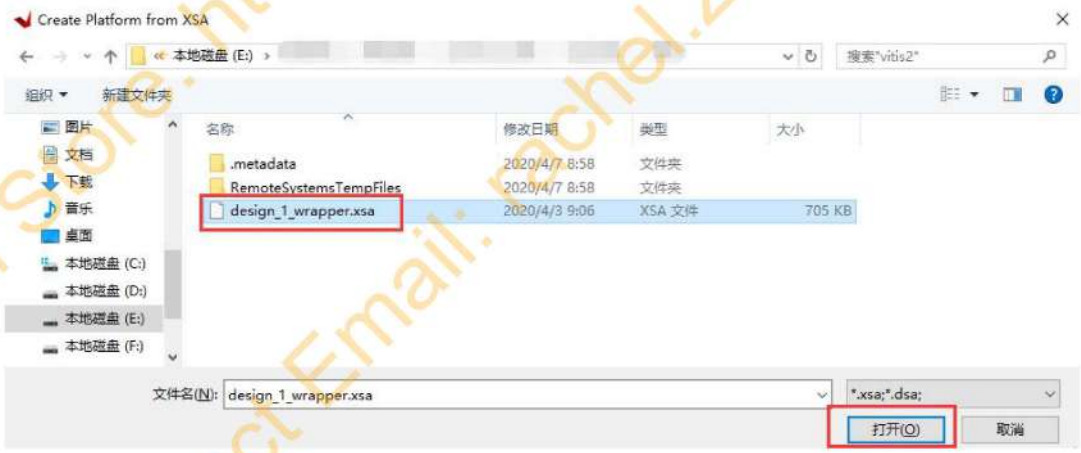


4) The first page is the introduction page, skip directly, click Next

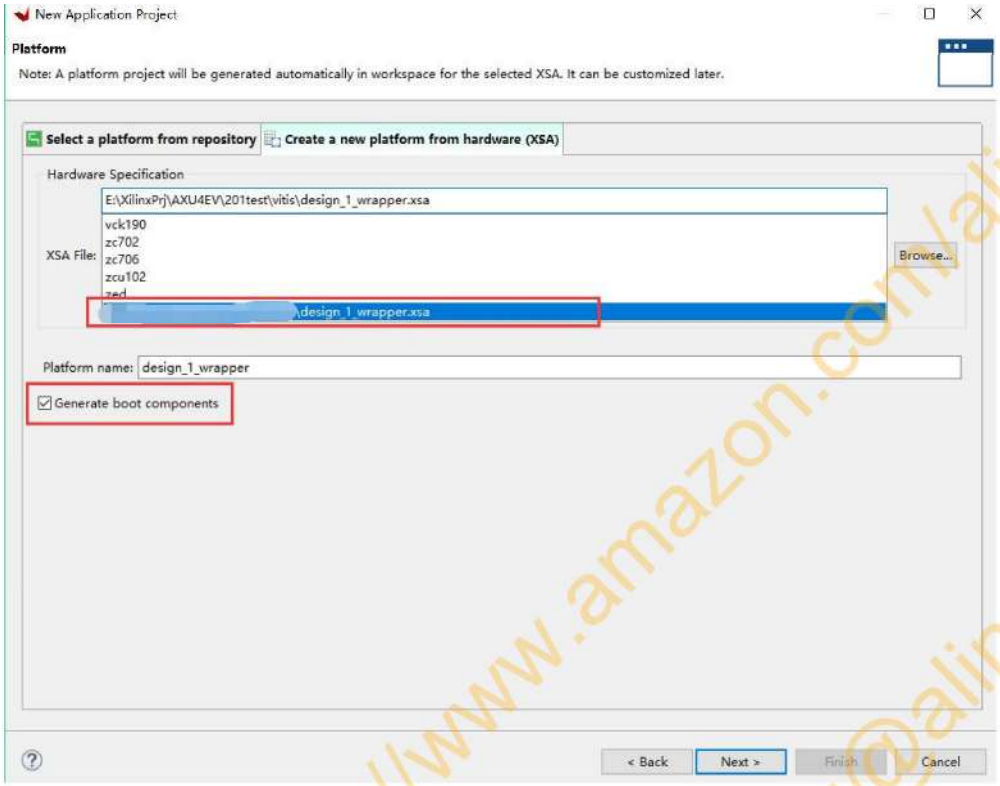
5) Select "Create a new platform from hardware(XSA)", select "Browse"



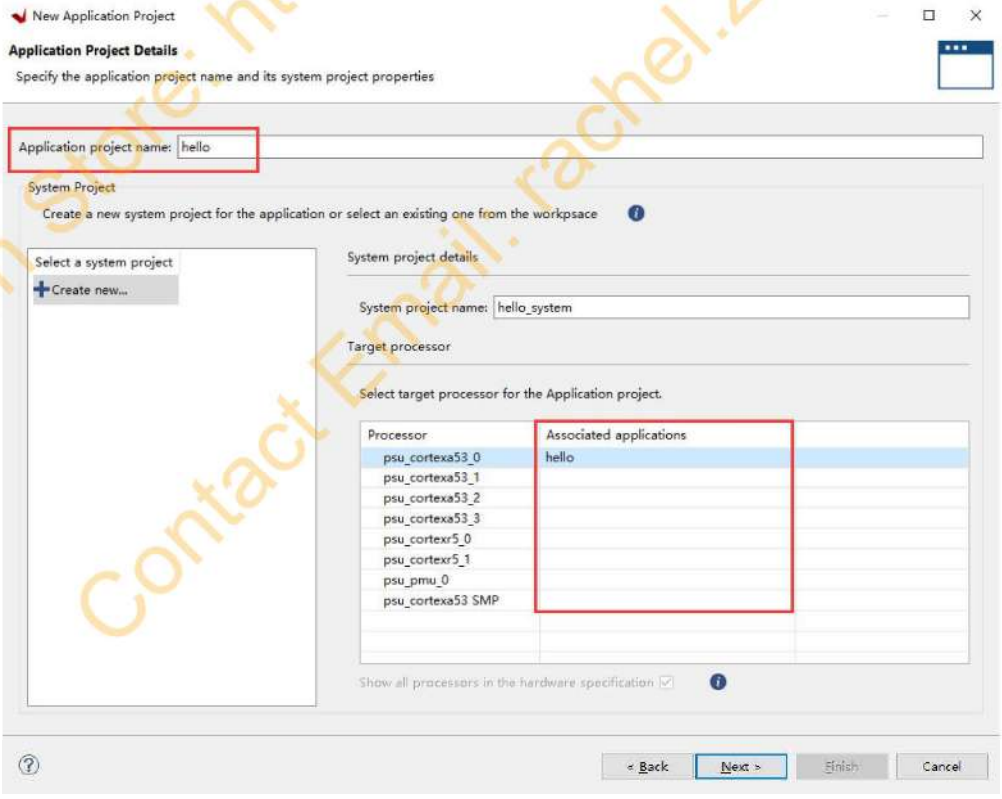
Select the previously generated xsa, click to open



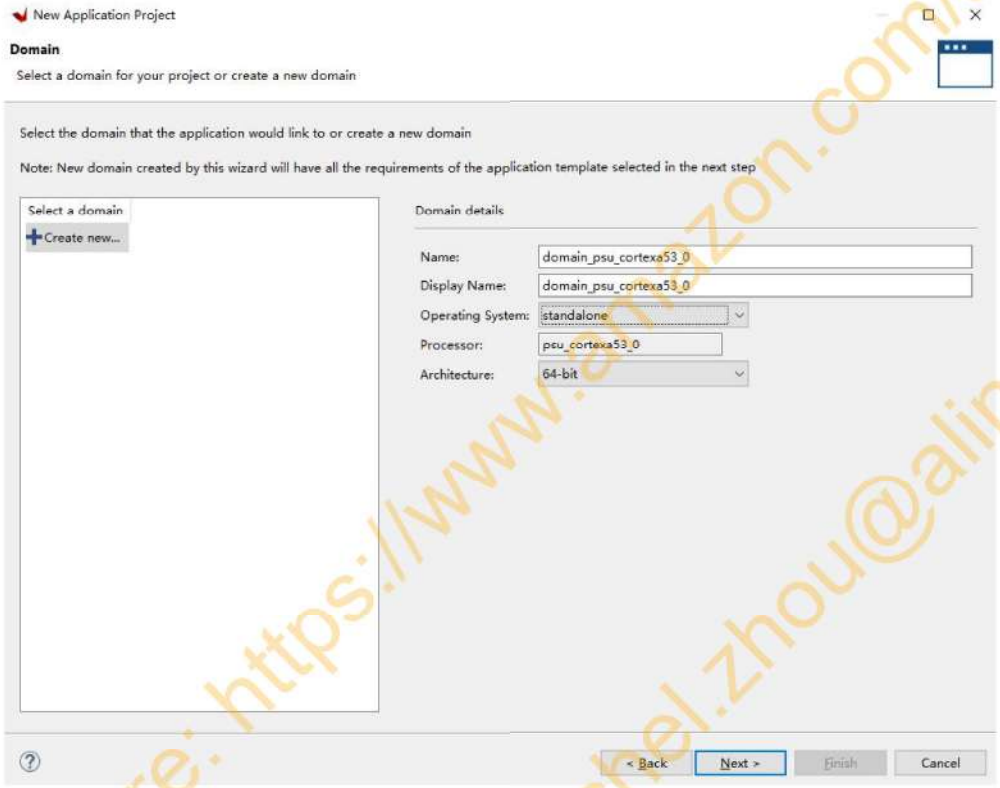
6) The Generate boot components option at the bottom, if checked, the software will automatically generate the fsbl project, we generally choose to check it by default



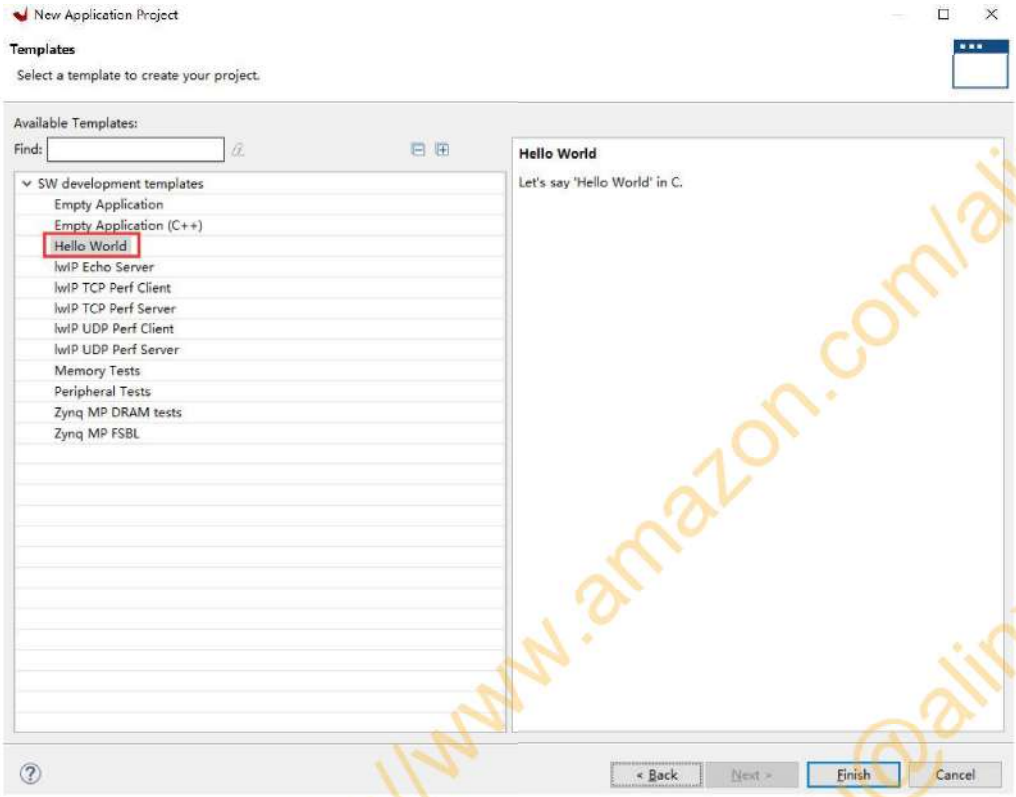
7) Fill in the APP project name, click in the box to select the corresponding processor, we keep the default here



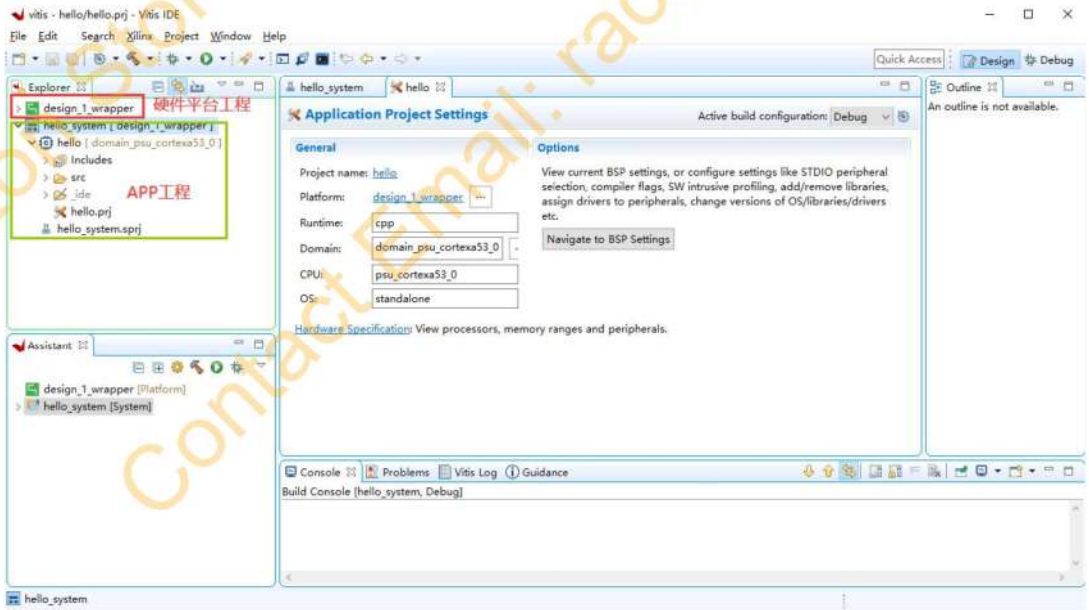
8) In this interface, you can modify the domain name, select the operating system, ARM architecture, etc., keep the default here, and select standalone for the operating system, which is bare metal.



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

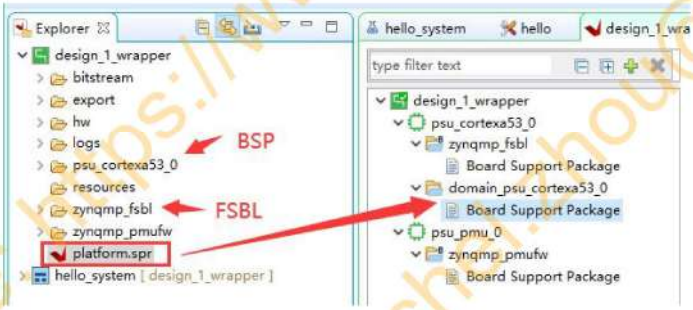


10)After completion, you can see that two projects have been generated, one is the hardware platform project, which is the Platform project mentioned before, and the other is the APP project.

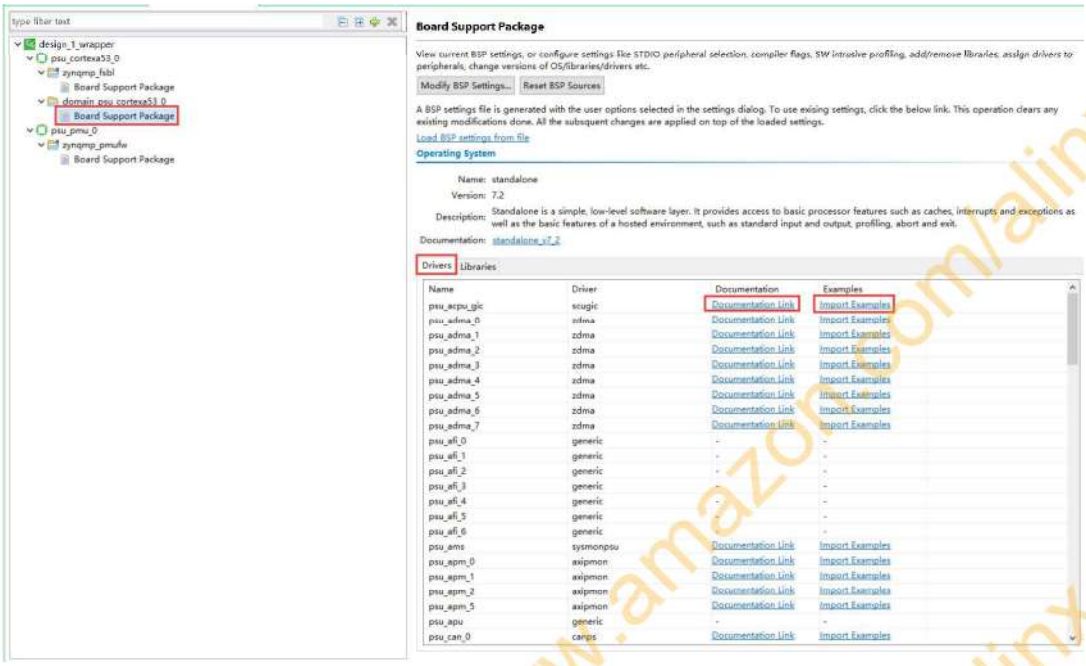


11)After expanding the Platform project, you can see that it contains

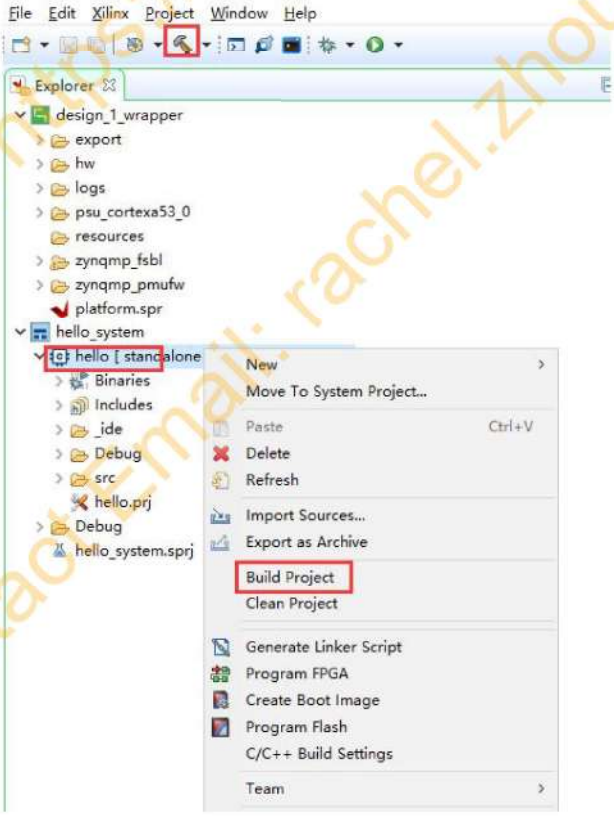
the BSP project and the zynq_fsbl project (this project is the result of selecting Generate boot components). Double-click platform.spr to see the BSP project generated by Platform, where you can configure the BSP. Software developers are more aware that BSP is also the meaning of Board Support Package,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



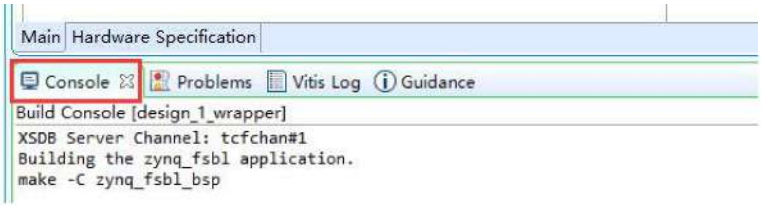
12)Click on the BSP to see the peripheral drivers of the project. Documentation is the driver documentation provided by xilinx, and Import Examples is the example project provided by xilinx to speed up learning.



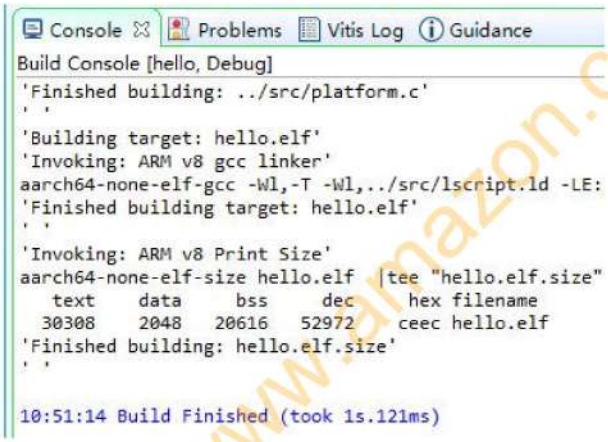
13)Select the APP project, right-click Build Project, or click the "hammer" button in the menu bar to compile the project



14)You can see the compilation process in the Console

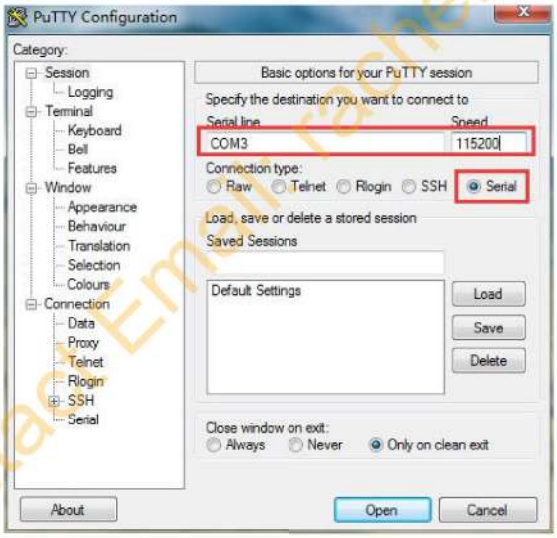


Compile is over, generate elf file

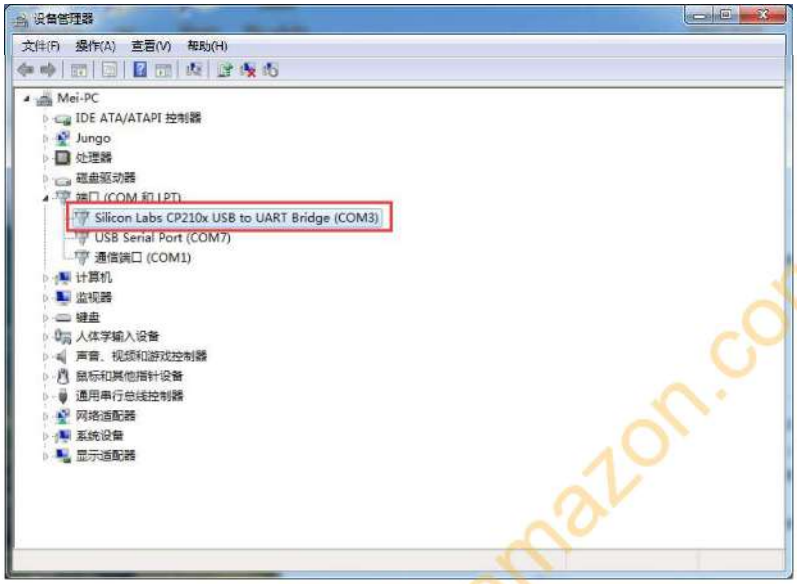


15)Connect the JTAG Cable to the development board, and the UART USB Cable to the PC

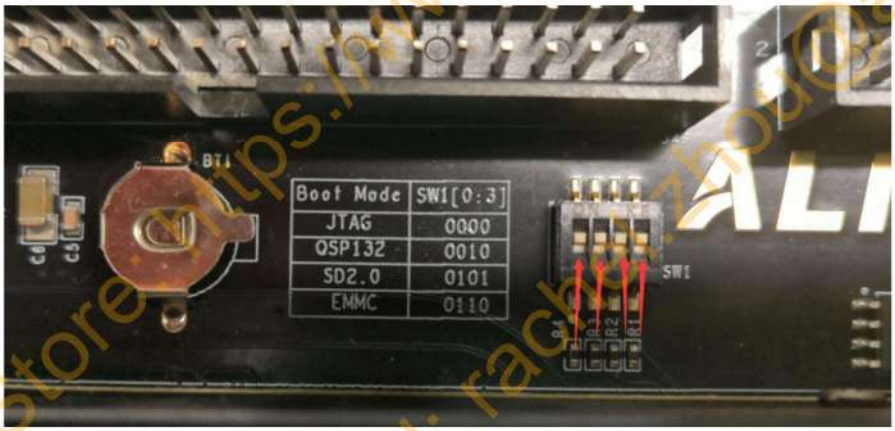
16)Use PuTTY software as a serial port terminal debugging tool, PuTTY is a small software that does not require installation



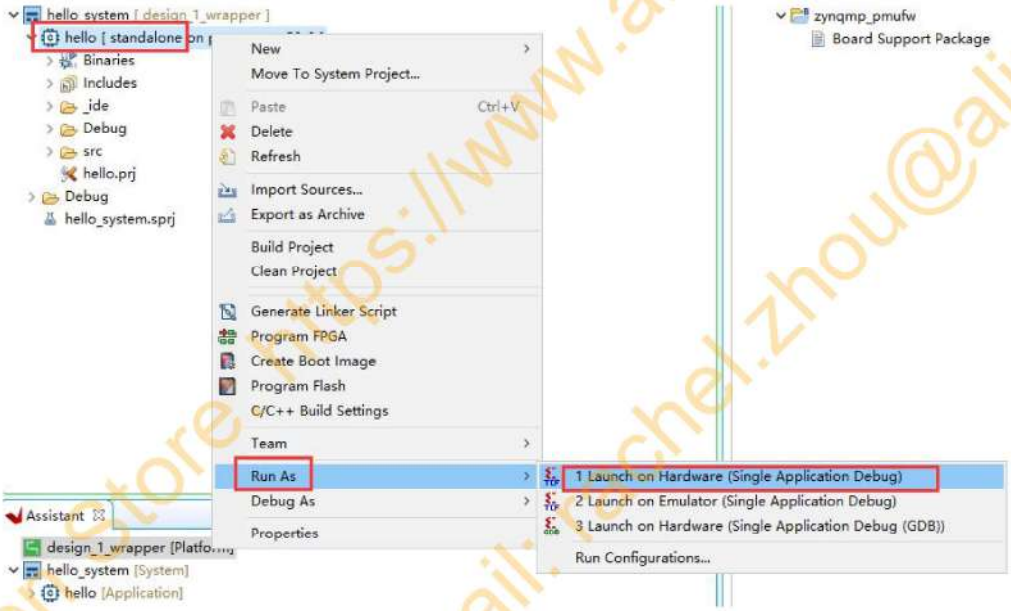
17)Select Serial, fill in COM3 for Serial line, 115200 for Speed, and fill in the COM3 serial port number as shown in the device manager, and click "Open"



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

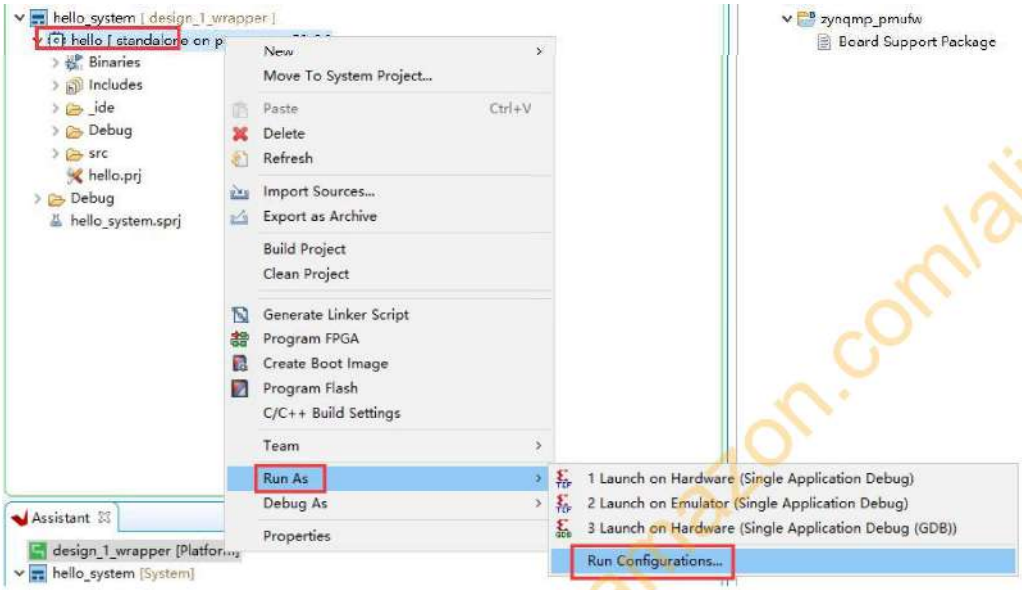


19)Power on the FPGA development board and prepare to run the program. The FPGA development board comes with a program when it leaves the factory. Here you can select the JTAG mode as the operating mode, and then power on again. Select "hello", right click, and you can see many options. The "Run as"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

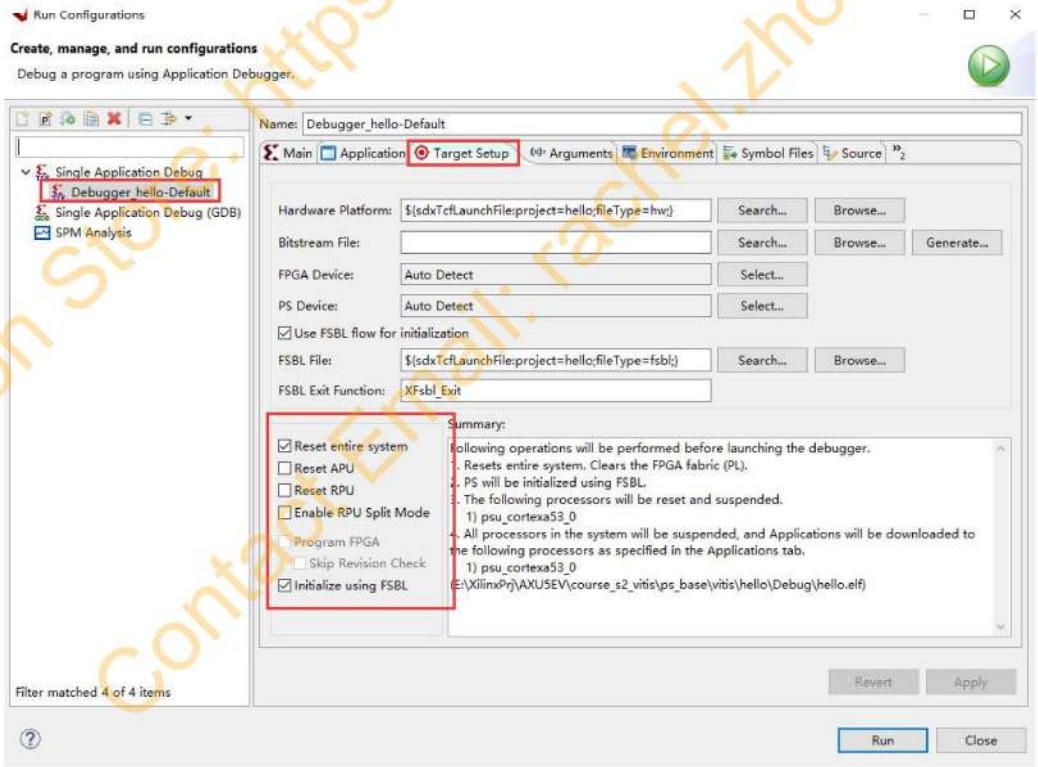


20)Observe the serial port software at this time, you can see the output "Hello World"

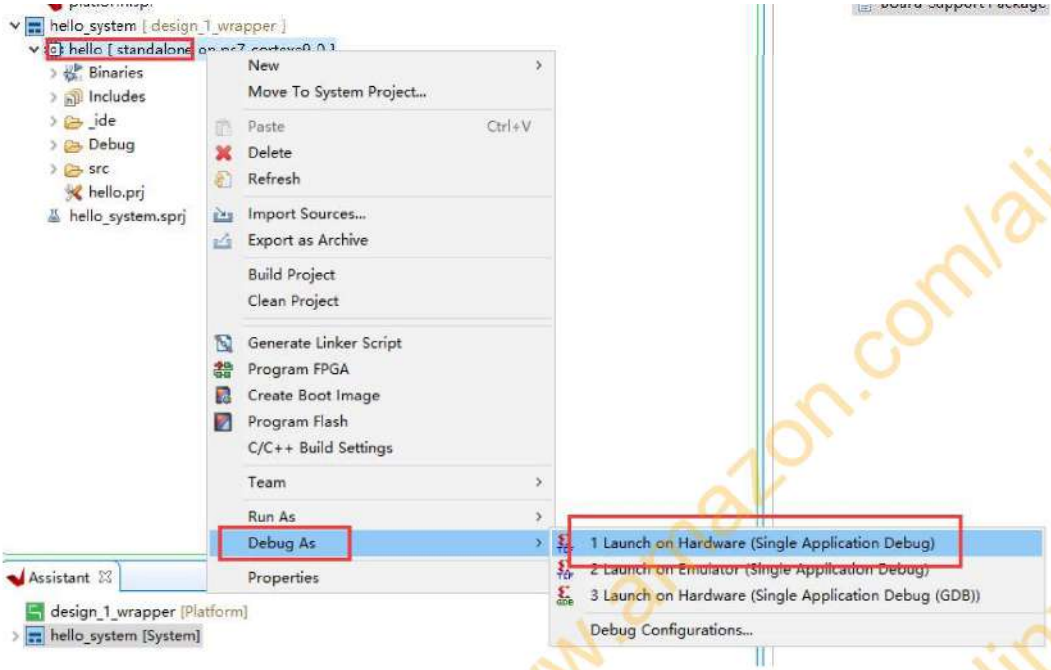
21)In order to ensure the reliable debugging of the system, it is best to right-click "Run As -> Run Configuration..."



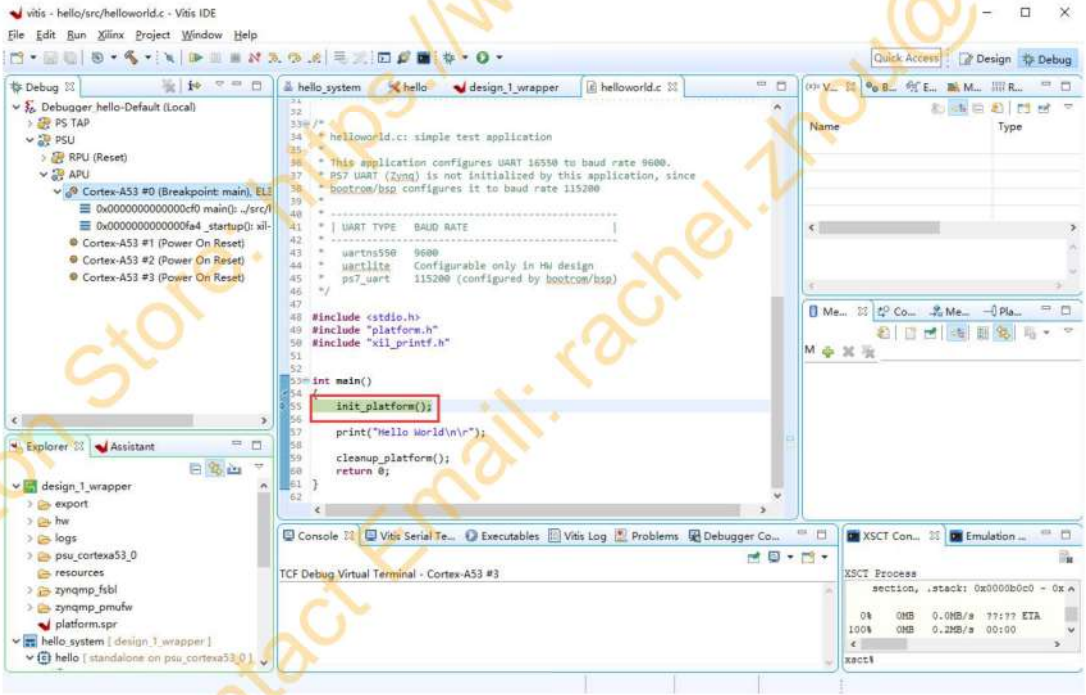
22)We can take a look at the configuration inside, where Reset entire system is selected by default, which is different from the previous SDK software. If there is a PL design in the system, you must also select "Program FPGA".



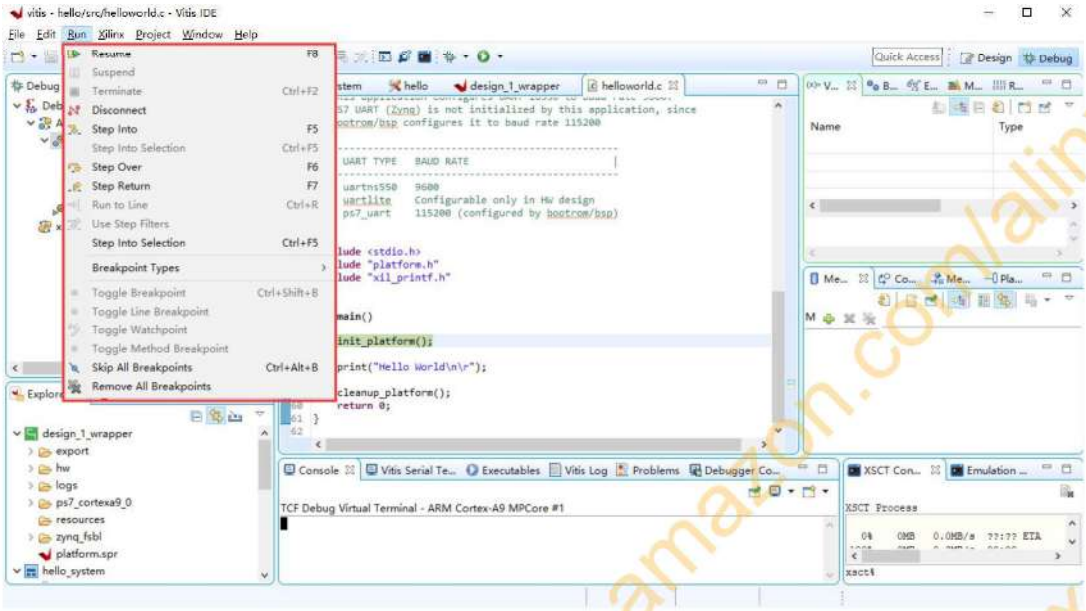
23)In addition to "Run As", you can also "Debug As" so that you can set breakpoints and run in a single step



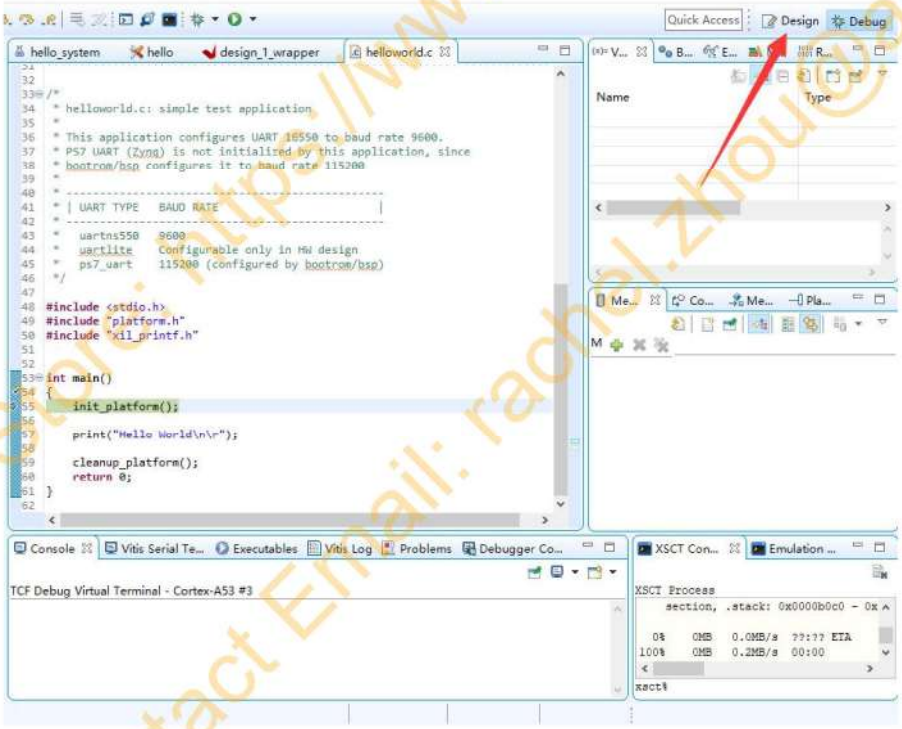
24)Enter Debug mode



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 satge: 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:

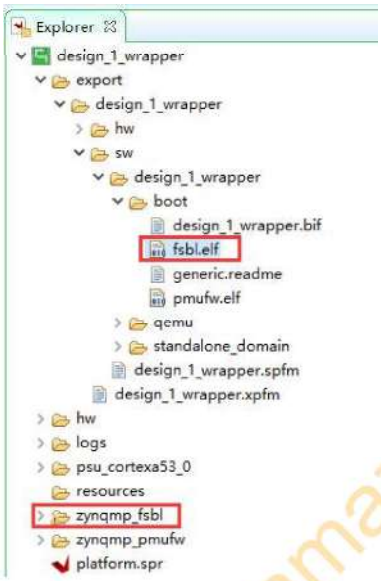
- Initialize PS configuration, MIO, PLL, DDR, QSPI, SD, etc.
- If there is a PL end program, load the PL end bitstream
- Transfer user program to DDR, and jump to execute

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. .

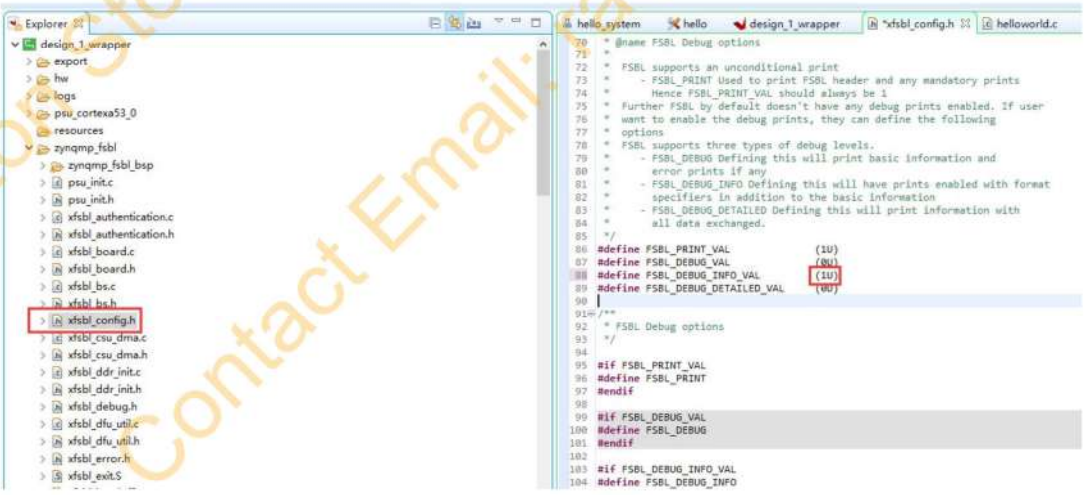
Part 1.4.1: Generate FSBL

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.

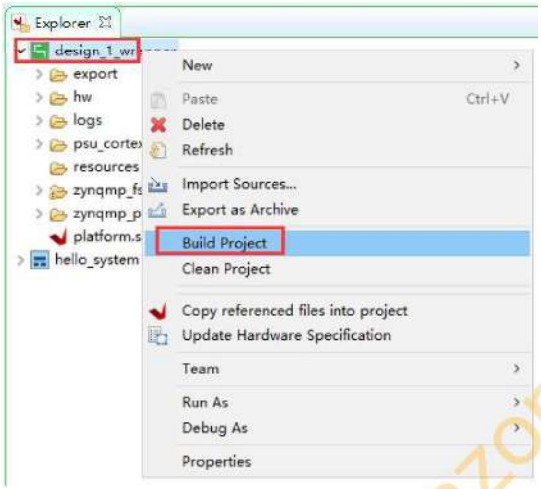
1) Since the Generate boot components option was selected when creating a new project, Platform has imported the fsbl project and generated the corresponding elf file.



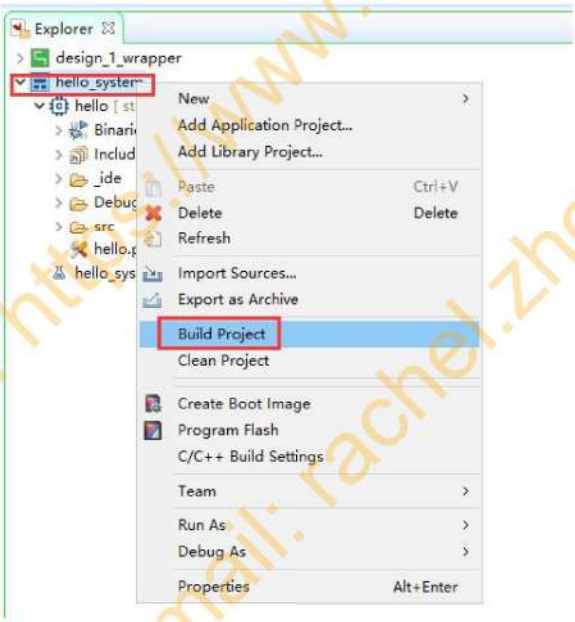
2) Modifying the debugging macro definition FSBL_DEBUG_INFO_VAL can output some status information of FSBL at startup, which is beneficial to debugging, but it will cause the startup time to become longer. save document. You can take a look at the files of many peripherals in fsbl, including psu_init.c, 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.



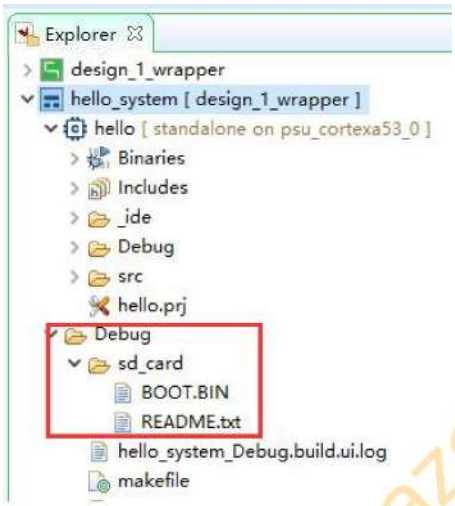
3) Build Project



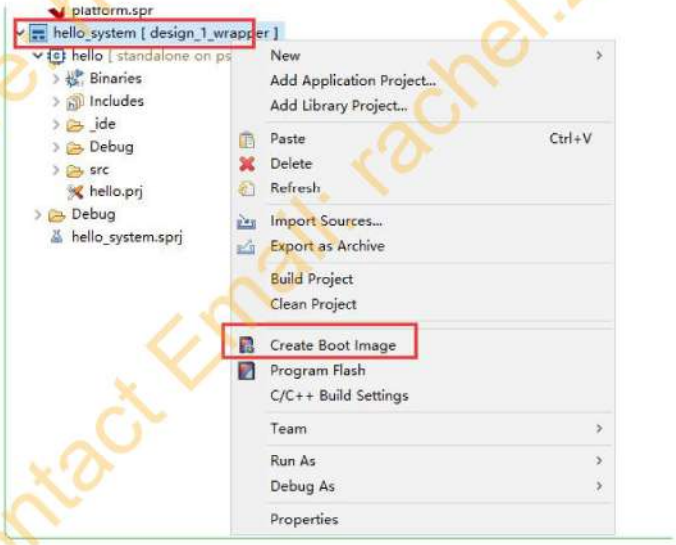
4) Next, we can click on the system of the APP project, right-click and select Build project

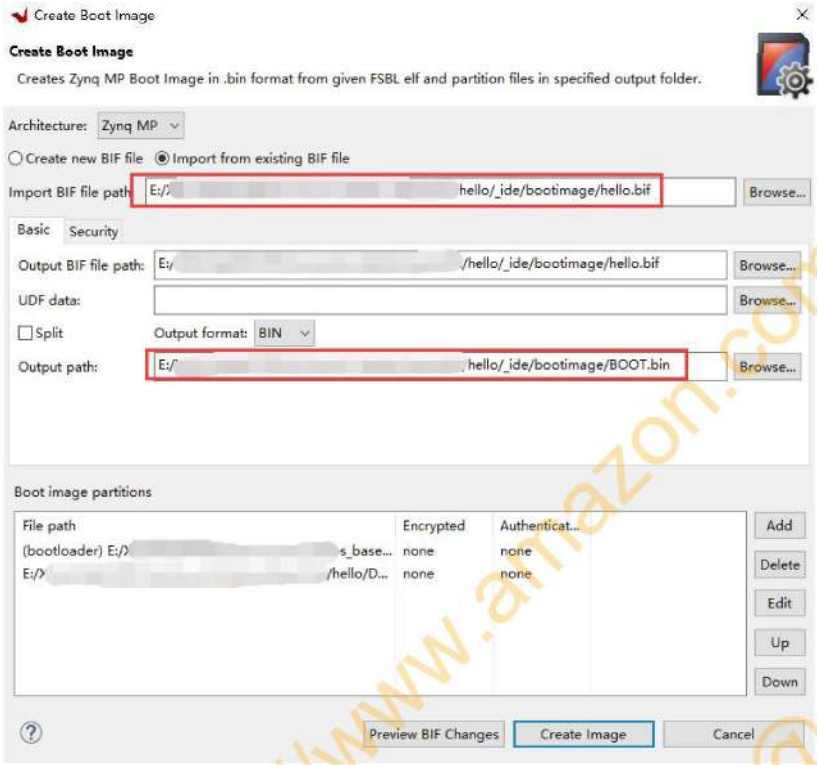


5) At this time, there will be an extra Debug folder, and the corresponding BOOT.BIN will be generated

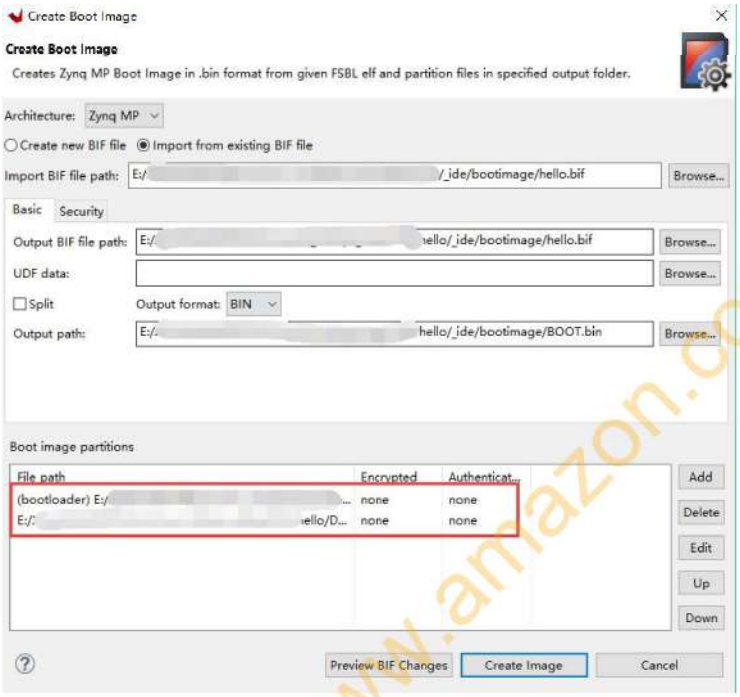


6) Another way is to click on the system of the APP project, right-click and select Creat Boot Image. In the pop-up window, you can see the path of the generated BIF file. The BIF file is the configuration file for generating the BOOT file, as well as the path of the generated BOOT.bin file. , The BOOT.bin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

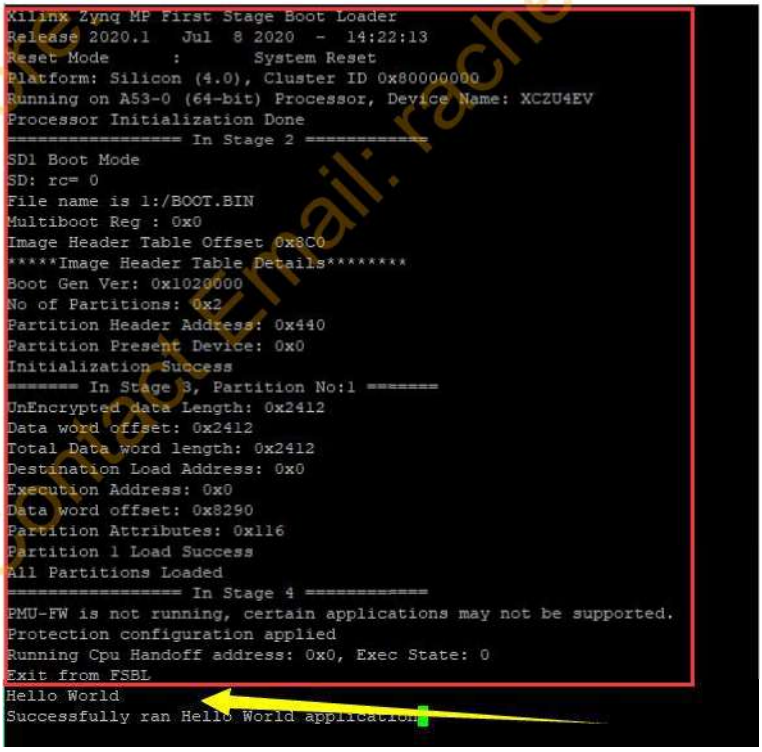
1) Format SD card, can only be formatted as FAT32, other formats cannot be started

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

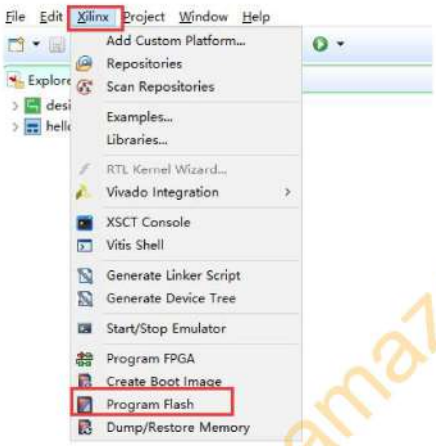
4) Adjust the boot mode to SD card boot

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”

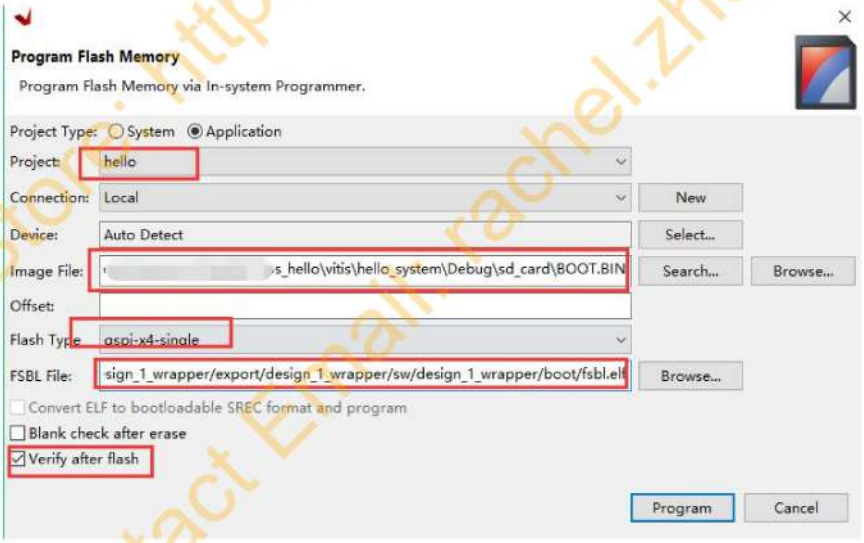


Part 1.4.3: QSPI Startup Test

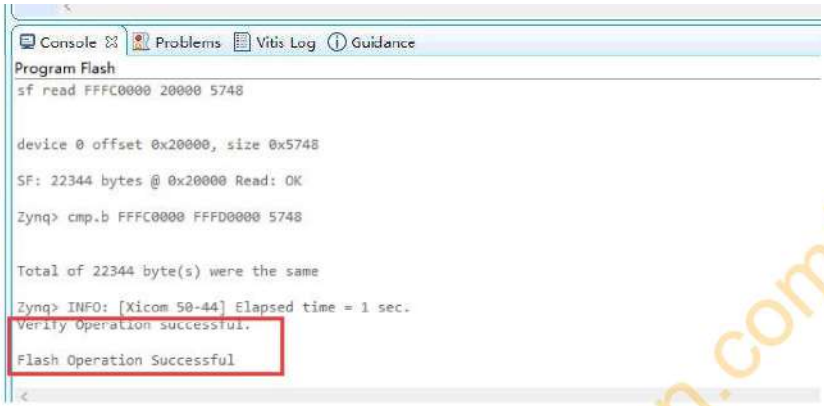
1) In the Vitis menu “Xilinx-> Program Flash”



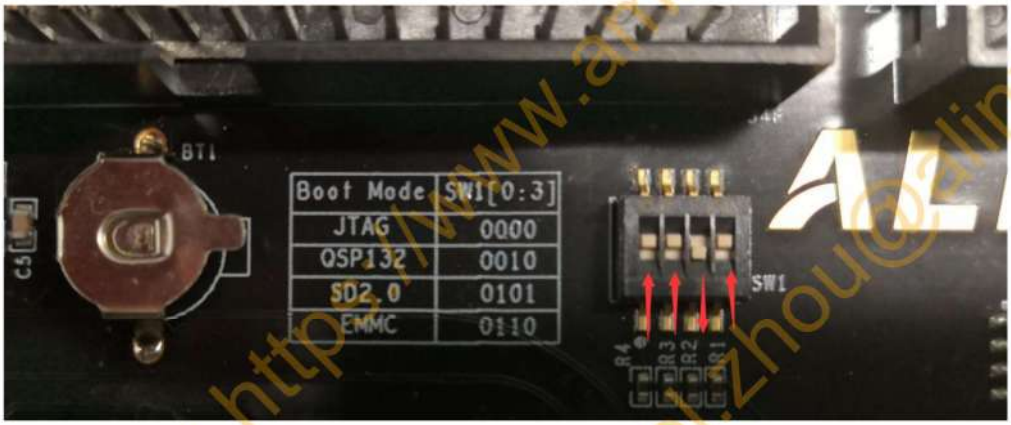
2) The “Hardware Platform” selects the latest, the “Image File” file selects the “BOOT.bin” to be flashed, and the “FSBL file” selects the “fsbl.elf”. Select “Verify after flash” to verify the “flash” after programming is complete.



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



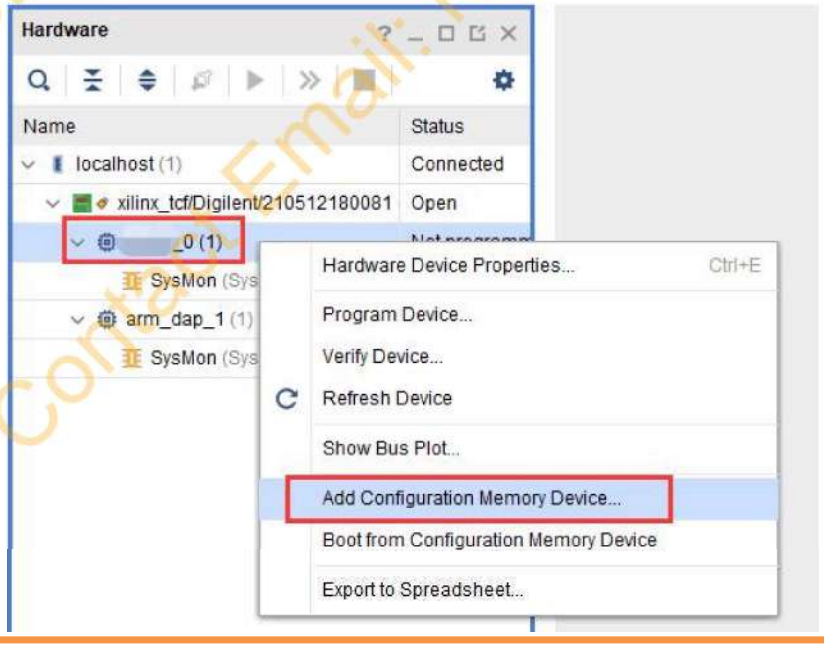
4) Set the startup mode to QSPI and start again, You can see the same startup effect as SD in the serial port software.



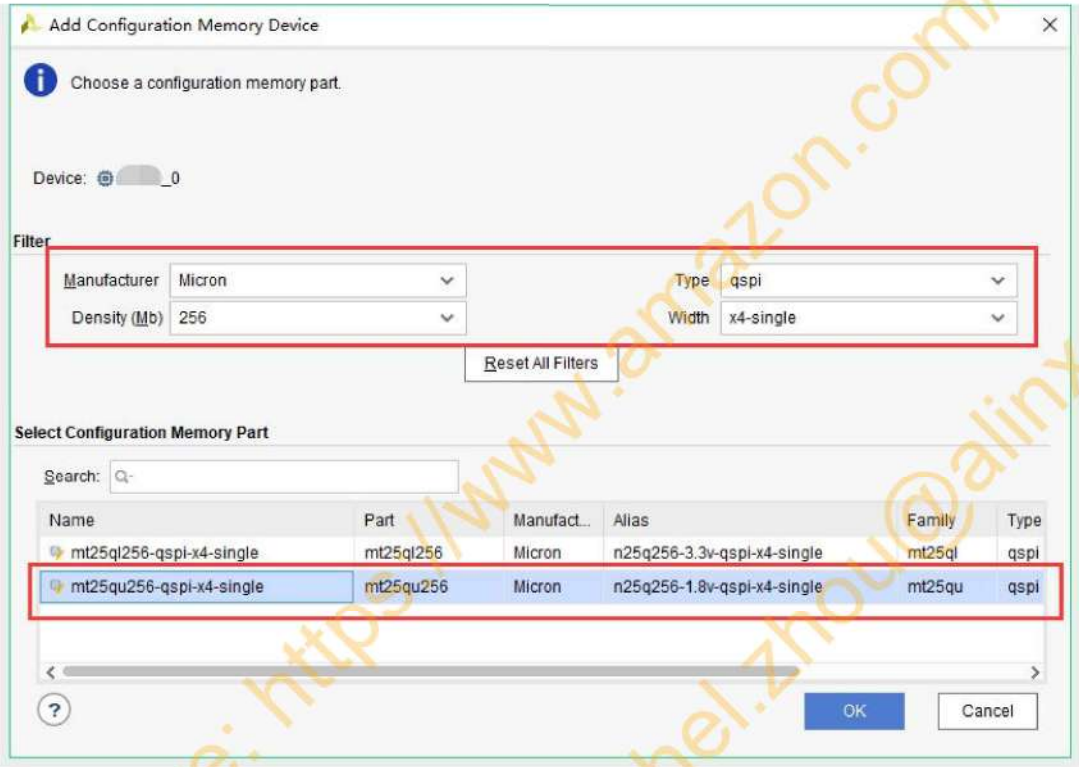
```
Xilinx Zynq MP First Stage Boot Loader
Release 2020.1   Jul  8 2020  -  14:22:13
Reset Mode      :       System Reset
Platform: Silicon (4.0), Cluster ID 0x80000000
Running on A53-0 (64-bit) Processor, Device Name: XCZU4EV
Processor Initialization Done
================== In Stage 2 ==============
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 EC0
.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 FSBL
Hello World
Successfully ran Hello World application
```

Part 1.4.4: Programming QSPI Under Vivado

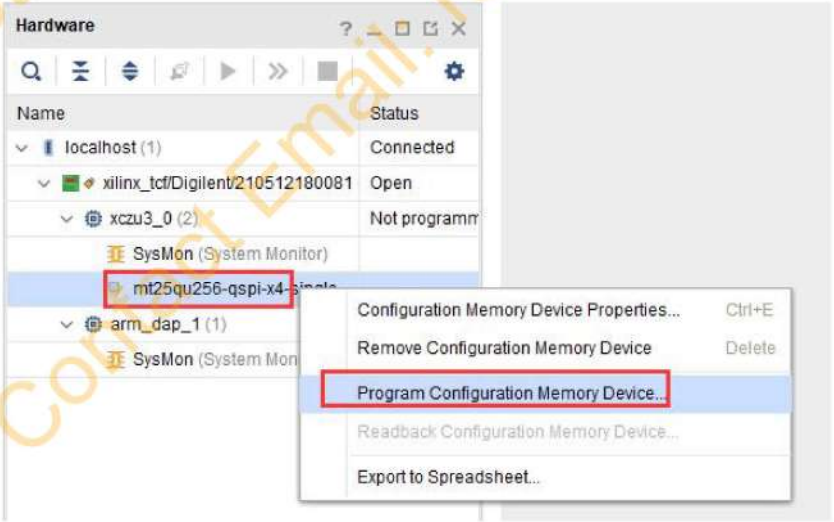
1) Select the device under “HARDWARE MANGER”, right-click “Add Configuration Memory Device”



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.

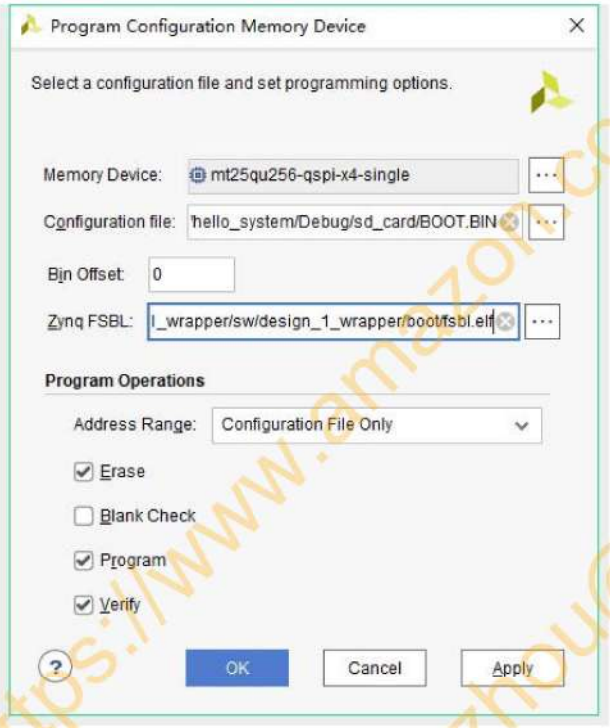


3) Right-click and select the programming file



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

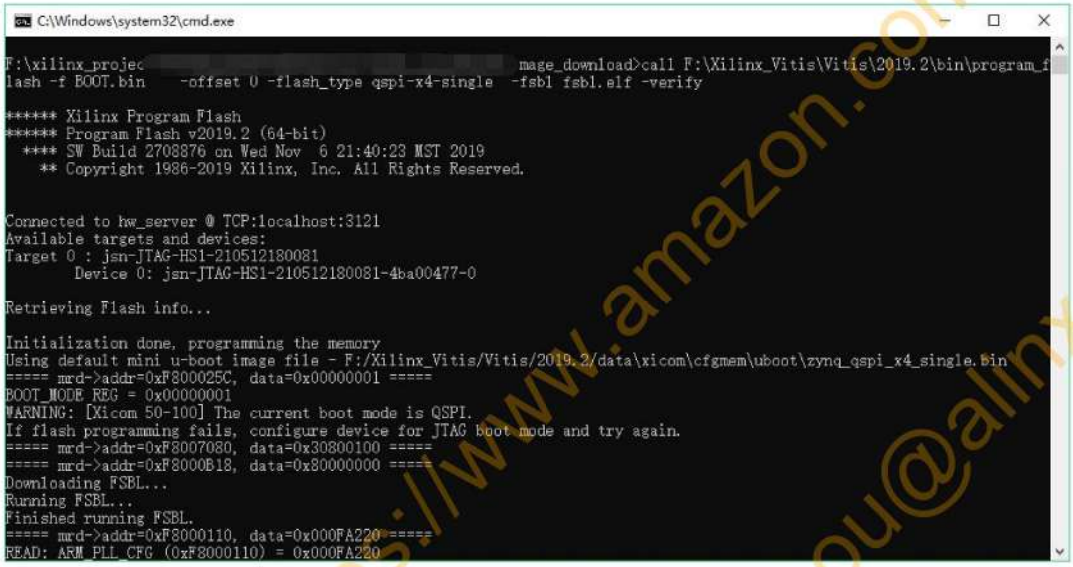
1) Create a new "program_qspi.txt" text file, change the extension to "bat", and fill in the following content, where "E:\XilinxVitis\Vitis\2020.1\bin\program_flash" is our tool path, modify it appropriately according to the installation path, "-f" is the file to be flashed, and "-fsbl" is the fsbl file (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.



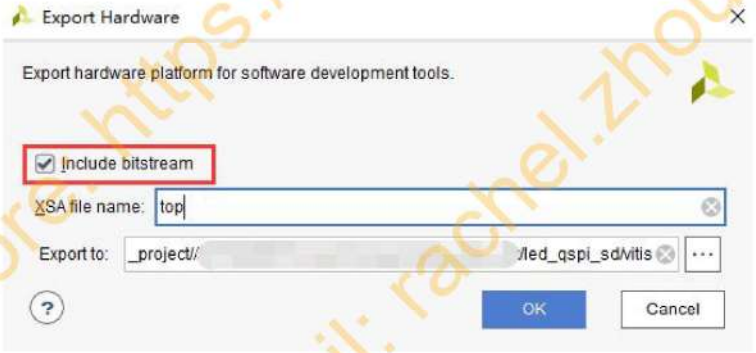
Part 1.5: Q&A

Part 1.5.1: Only PL side Logic Solidification

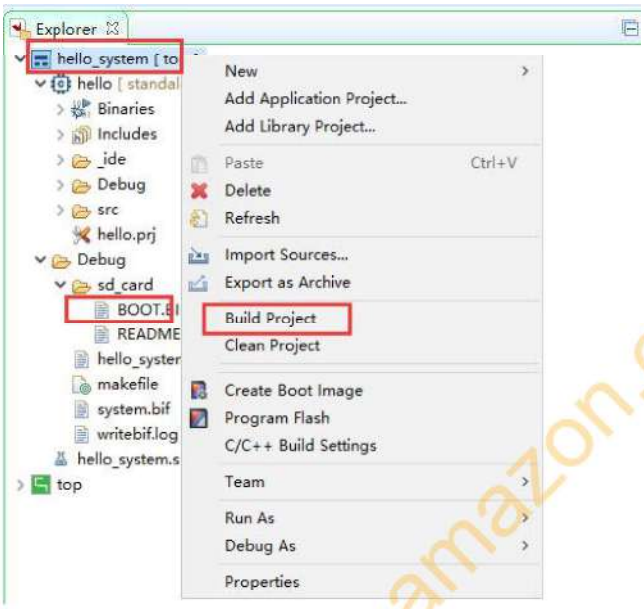
Many people will ask, if there is only the logic on the PL side, how to fix the program on the PS side? There is no problem in FPGA without ARM, but for ZYNQ, the cooperation of PS side is required to solidify the program. So how to fix the program for the previous "PL" Hello World "LED experiment"?

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.

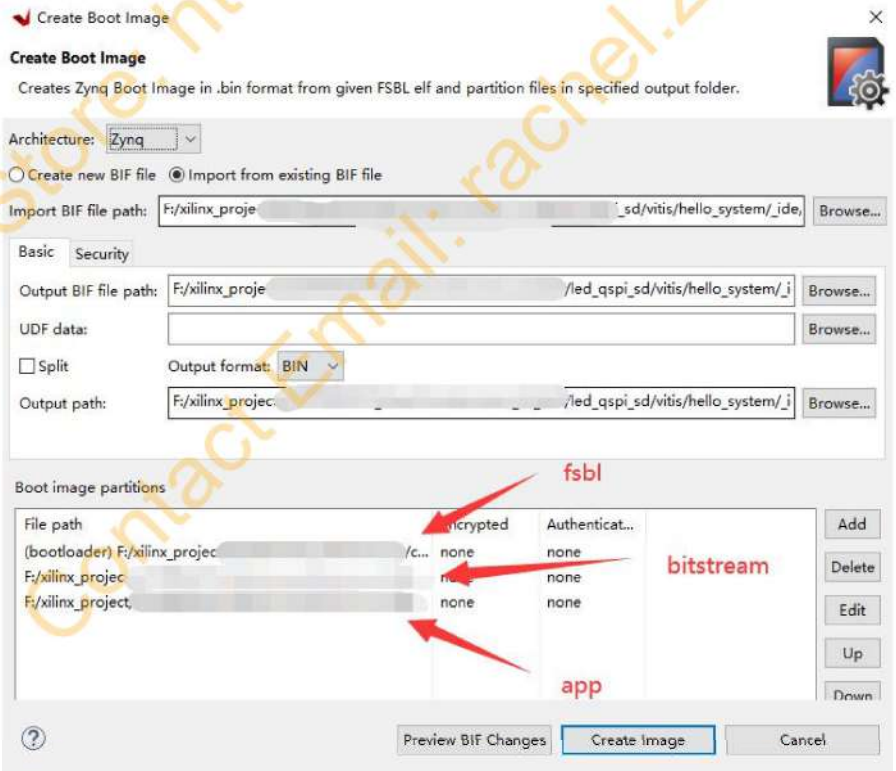
2) After generating the bitstream, export the hardware and select "include bitstream"



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.



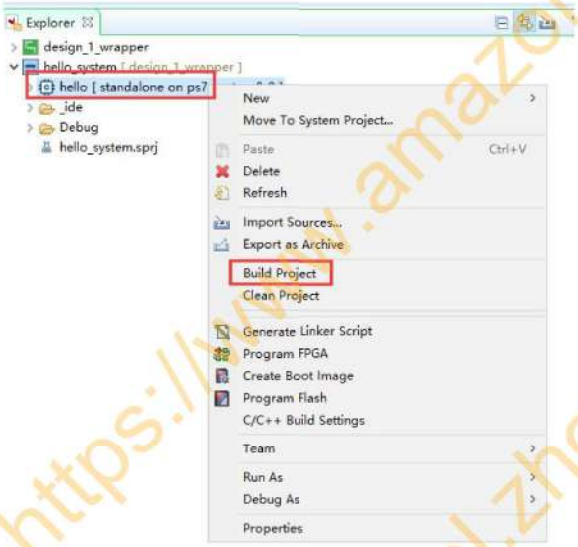
4) Open the Create Boot Image interface and you can see, the file order of the “BooImage Partitions” is “fsbl”, “bitstream”, “app”, pay attention not to reverse the order. Using the BOOT.BIN generated in this way, you can test and start according to the previous startup method.



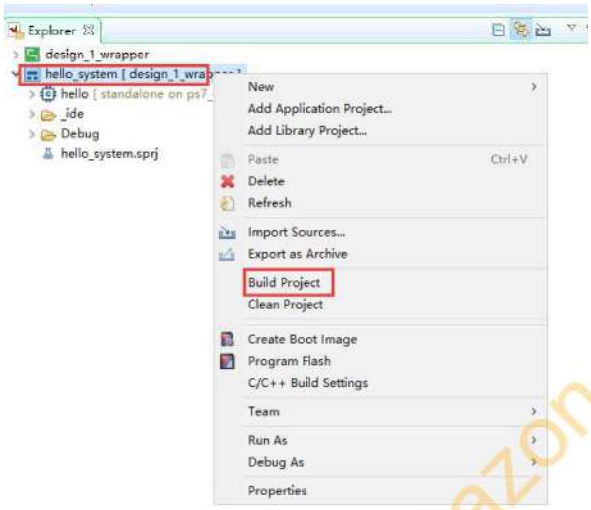
5) In the "course_s2" folder, a project named "led_qspi_sd" is provided for your reference.

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 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.

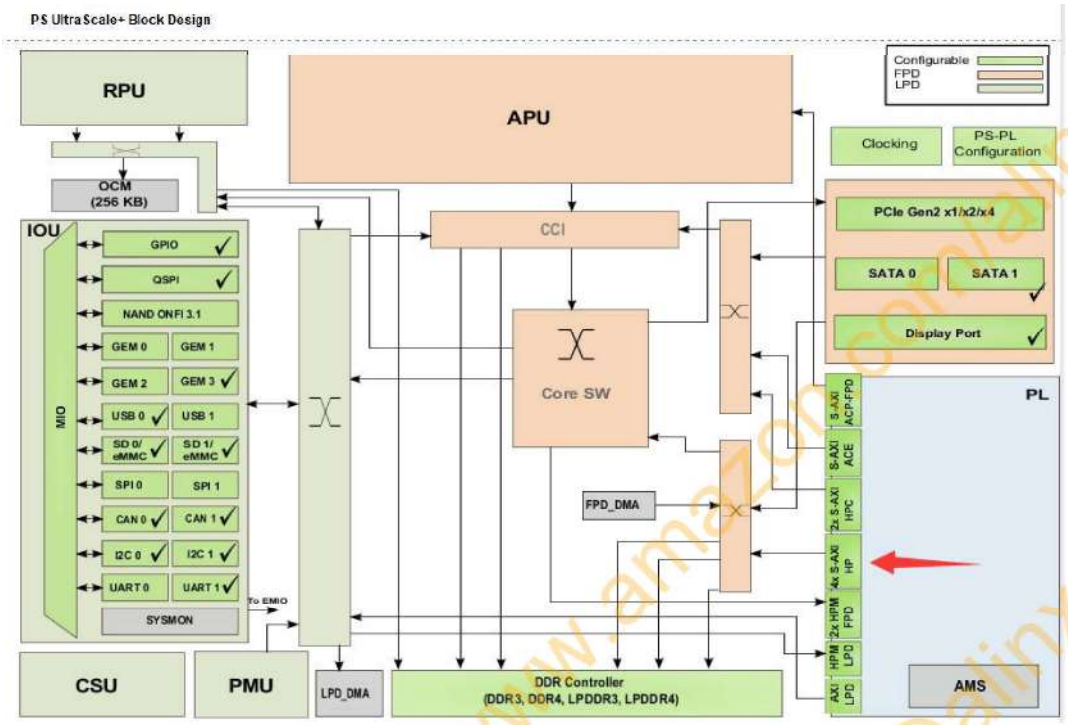
Normally we would think of using DMA to do this, but various protocols are very troublesome and flexibility is relatively poor. This section of the course explains how to directly read and write data on the PS side ddr through the AXI bus. This involves the AXI4 protocol, vivado FPGA debugging and so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

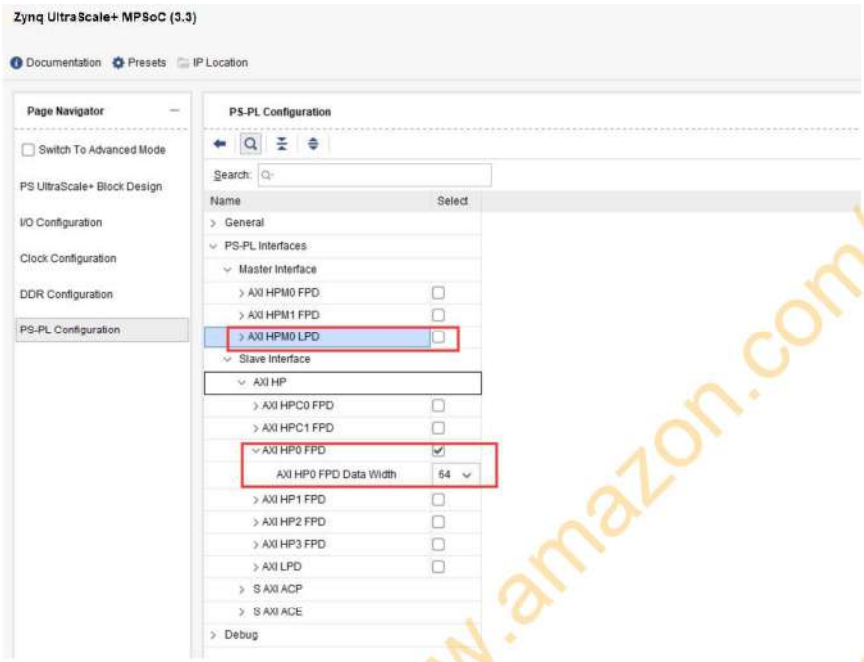


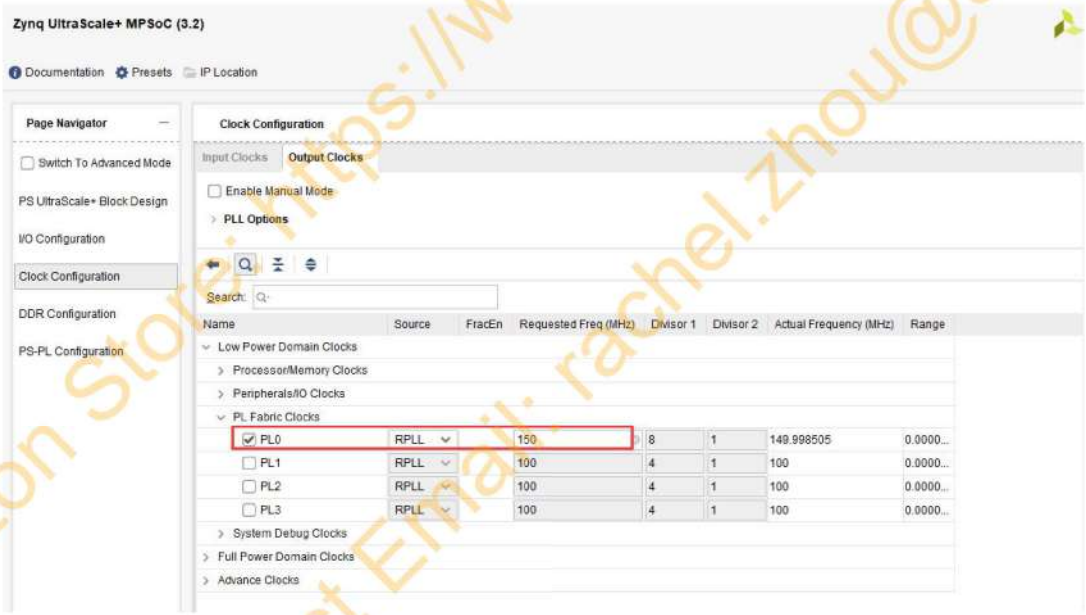
FPGA Engineer Job Content

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

Part 19.2: Hardware Environment

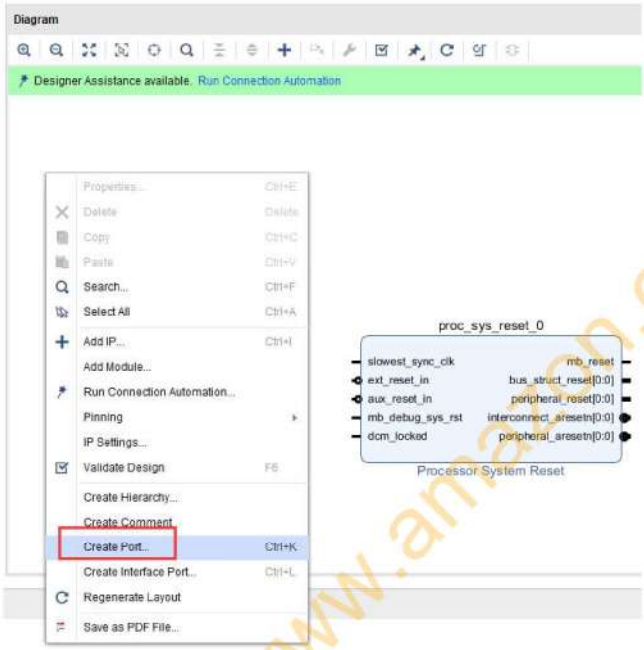
1) Based on the "ps_hello" project, the HP configuration in the vivado interface is as shown in the figure below (HP0~HP3). There are enable control and data bit width selection. You can choose 32bit, 64bit or 128bit. In our experiment, HP0 is configured to be 64bit wide, the clock used is 150Mhz, and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



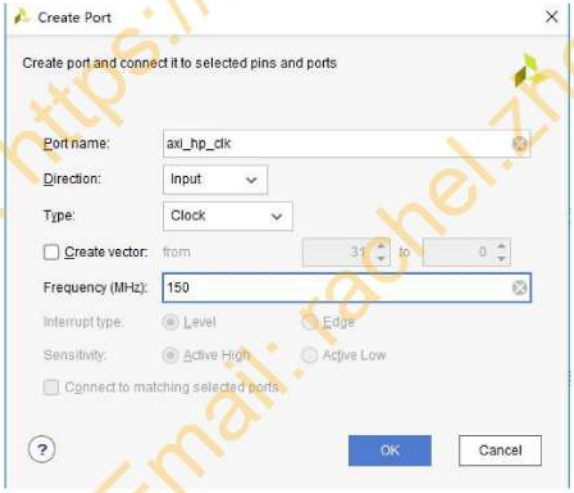


2) Add reset module for reset

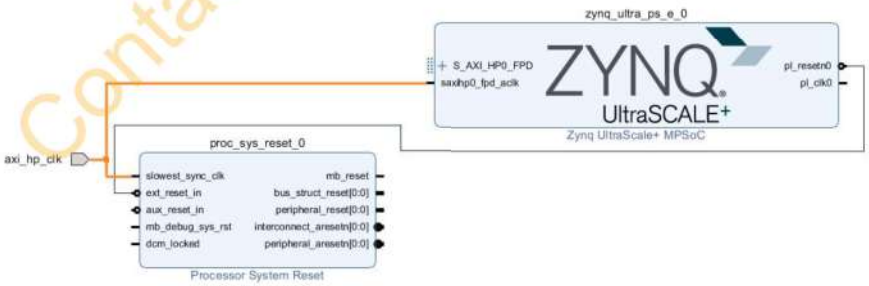
3) Right-click on the blank space and select "Create Port"



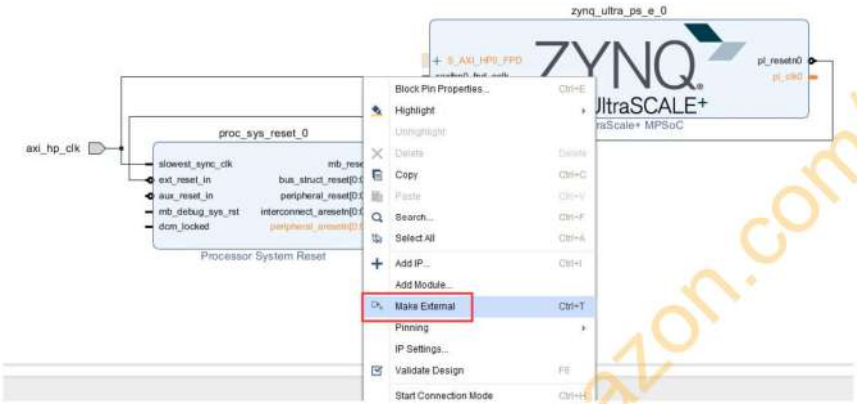
Configuration as shown



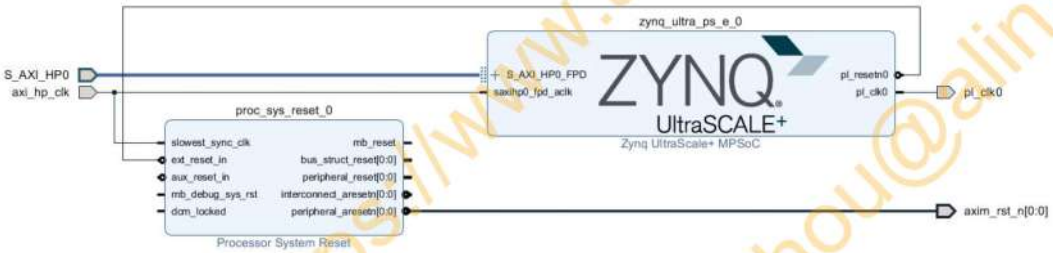
4) Connect the clock and reset



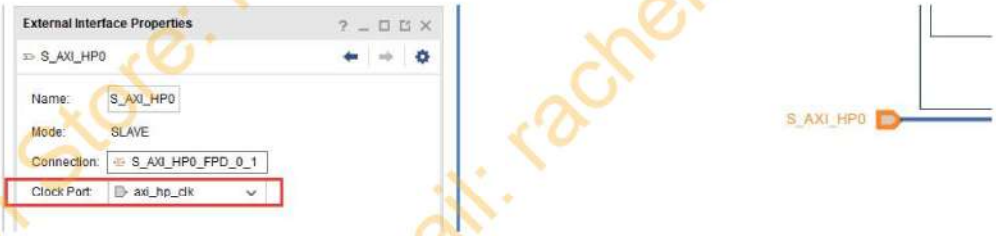
5) Select the pin and click Make External to export the signal



And modify the pin name as shown below

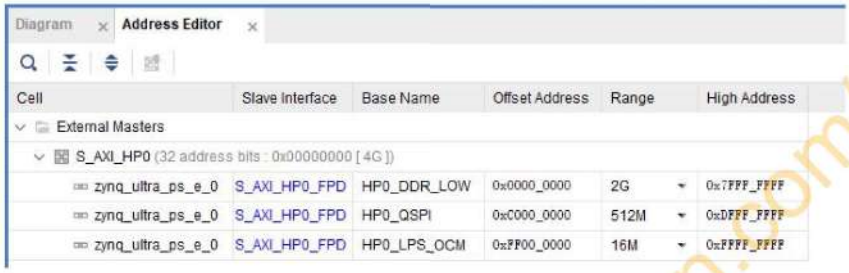


And select the bus synchronization clock as axi_hp_clk



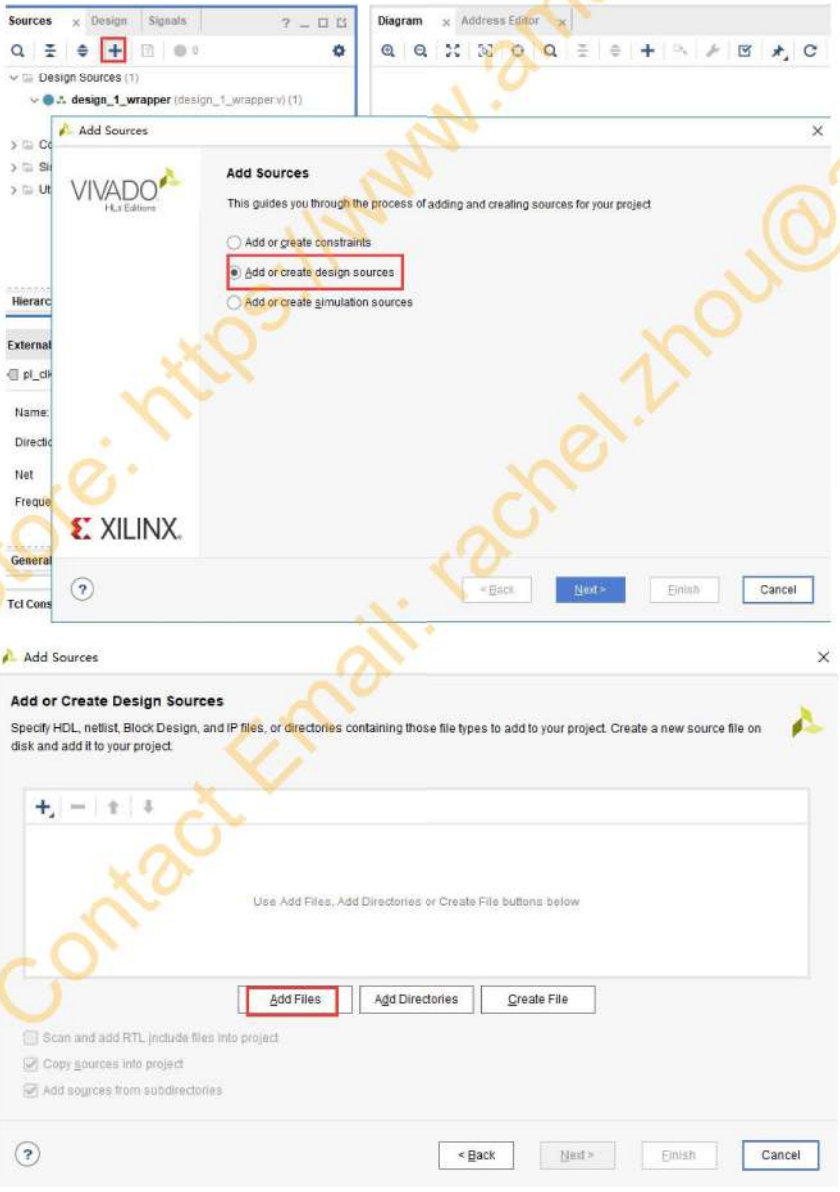
6) Click on the Address Editor, if you find that the address is not assigned, click the button to automatically assign an address

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

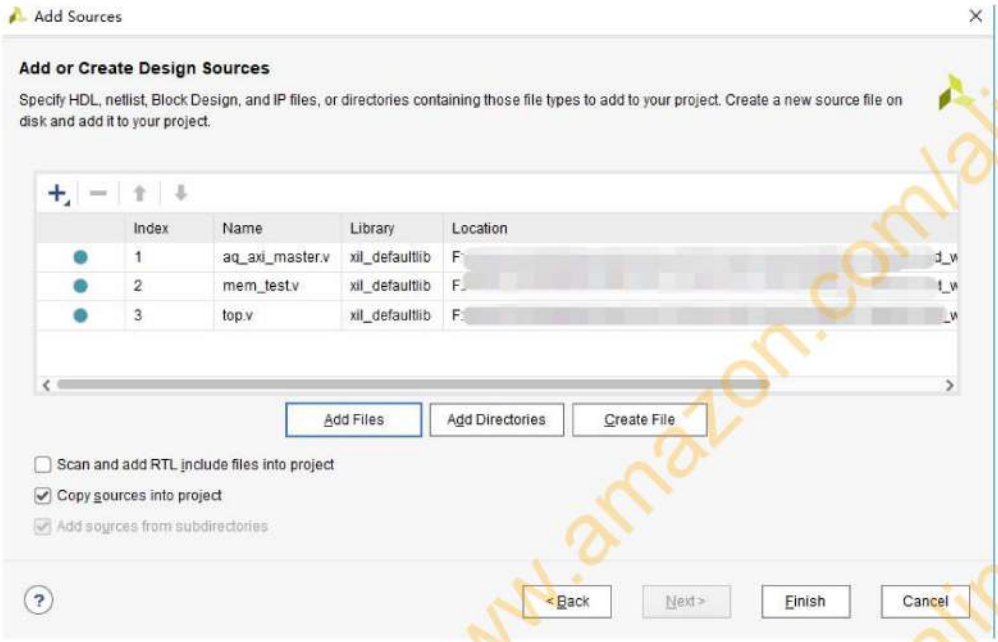


Save the design and regenerate Ouput Product

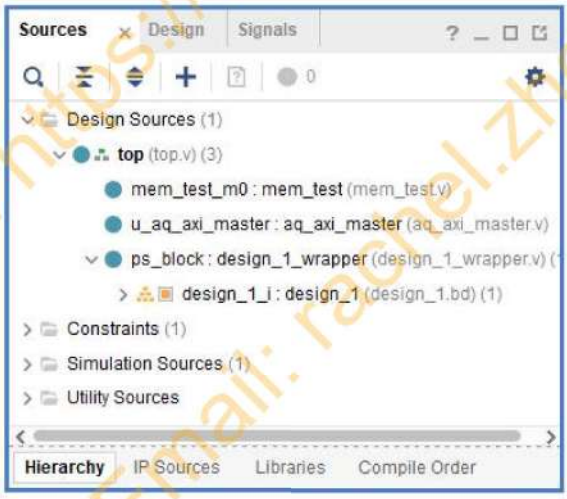
7) Add hdl files



8) Click Finish



HDL hierarchical relationship update results

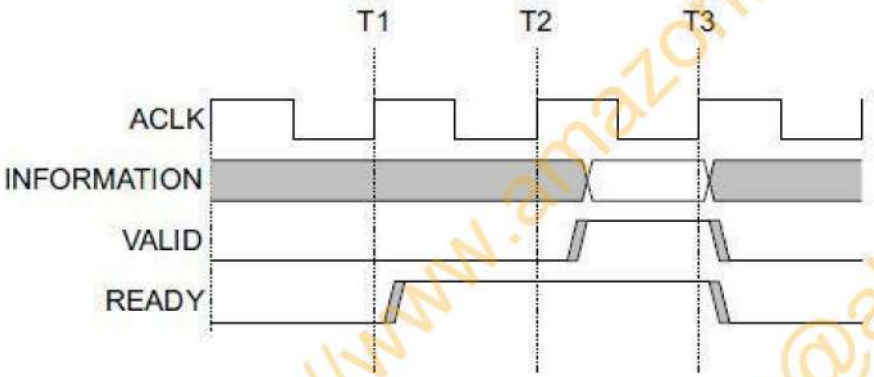


Part 19.3: PL Side AXI Master

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.



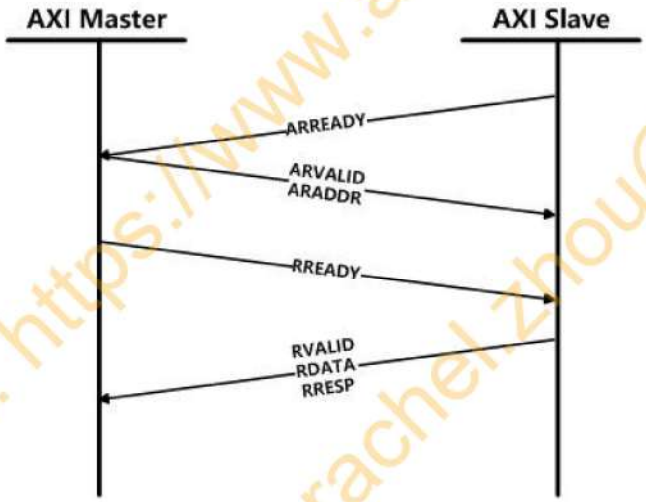
The AXI bus is divided into five channels:

- 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

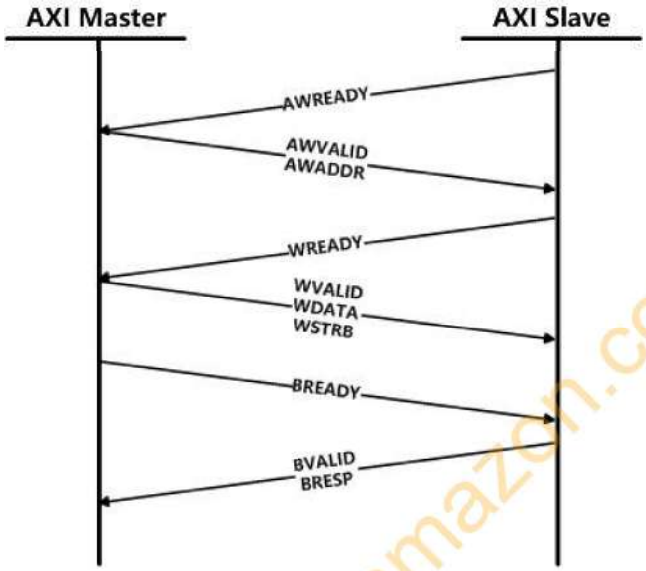
“ACLK” is “axi” bus clock, “ARESETN” is “axi” bus reset signal, active low; read/write data and read/write address class signal width

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.

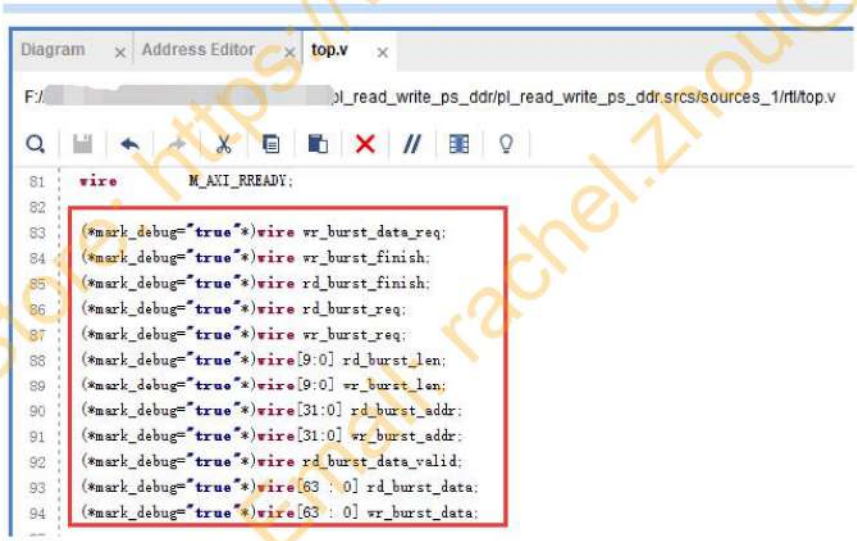
Part 19.4: Verification of ddr Read and Write Data

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:



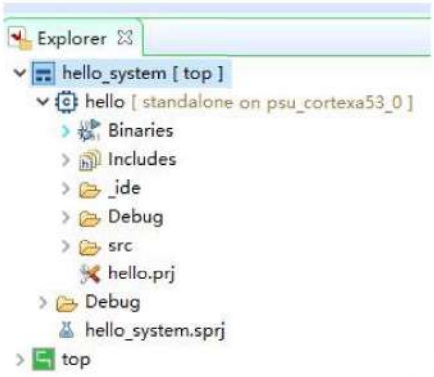
The specific adding method has been discussed in "PL's "Hello World" LED Experiment", you can refer to it.

And bind the error signal to the LED light on the PL side in the XDC file.

Part 19.6: Vitis Program Development

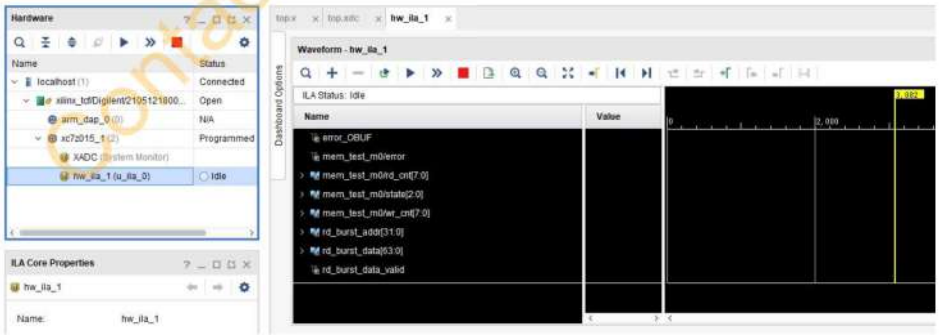
Use hello world as a template to create a new vitis project as

follows

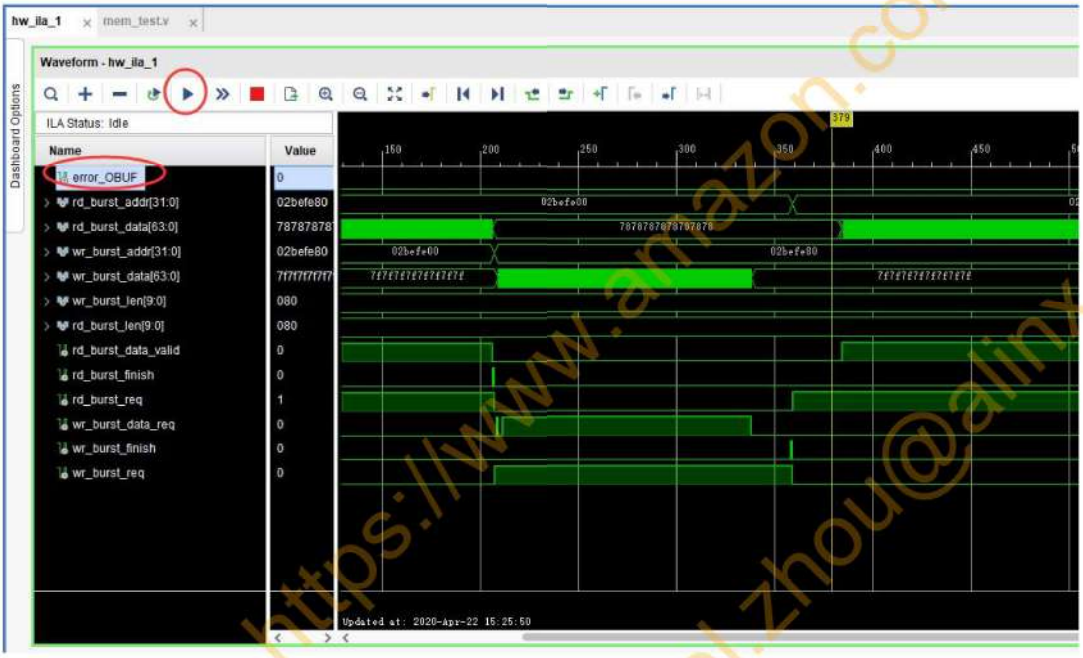


After downloading the program through vitis, the system will reset and download the bit file of FPGA. Then return to the vivado interface and click on the Program and Debug column to automatically connect to the target as shown in the figure below:

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