ZYNQ Development Platform Basic Tutorial

AX7010/AX7020

ALINX

Version Record

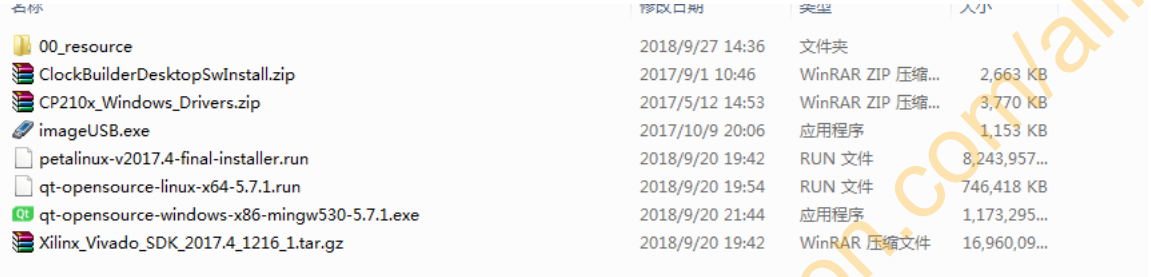
Version	Date	Release By	Description
Rev1.0	2020-09-06	Rachel Zhou	First Release

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.

Table of Content

Part 1: Software Package Introduction and FPGA Development board Inspection

Part 1.1: Software Package Introduction



名称	修改日期	类型	大小
00_resource	2018/9/27 14:36	文件夹	
ClockBuilderDesktopSwInstall.zip	2017/9/1 10:46	WinRAR ZIP 压缩...	2,663 KB
CP210x_Windows_Drivers.zip	2017/5/12 14:53	WinRAR ZIP 压缩...	3,770 KB
imageUSB.exe	2017/10/9 20:06	应用程序	1,153 KB
petalinux-v2017.4-final-installer.run	2018/9/20 19:42	RUN 文件	8,243,957...
qt-opensource-linux-x64-5.7.1.run	2018/9/20 19:54	RUN 文件	746,418 KB
qt-opensource-windows-x86-mingw530-5.7.1.exe	2018/9/20 21:44	应用程序	1,173,295...
Xilinx_Vivado_SDK_2017.4_1216_1.tar.gz	2018/9/20 19:42	WinRAR 压缩文件	16,960,09...

1) CP210x_Windows_Drivers.zip Serial port driver
2) Xilinx_Vivado_SDK_2017.4_1216_1.tar.gz Vivado 2017.4 Installation package; Windows and Linux Universal version. WinRAR decompression software is required under windows
3) Petalinux-v2017.4-final-installer.run Petalinux Installation package
4) qt-opensource-windows-x86-mingw530-5.7.1.exe **Windows Version QT**
5) qt-opensource-linux-x64-5.7.1.run **Linux Version QT**
6) ImageUSB.exe Mirror recovery tool
7) 00_resource contains Linux source code, root file system
8) VMware-workstation-full-12.1.1-3770994.exe Virtual machine installation package
9) Ubuntu-16.04.3-desktop-amd64.iso Ubuntu Installation package, Can only be installed on a PC, not on the FPGA development board

Part 1.2: FPGA Development Board Inspection

After receiving the development board, most people want to experience it immediately and see if the development board is working properly. Here we introduce how to perform a simple test on the development board.

Part 1.2.1: Detect tools that need to be prepared:

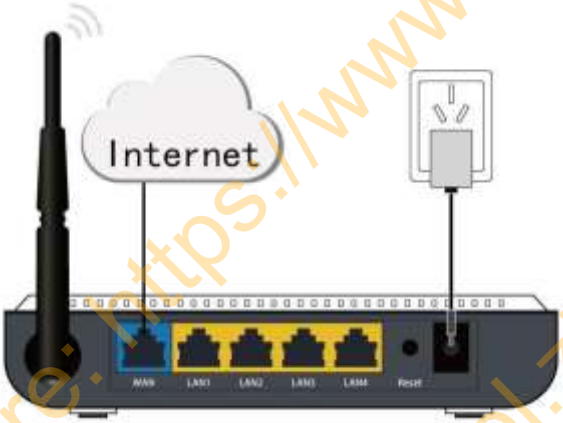
1) Computer

2) The monitor supports HDMI and requires a resolution of at least 1920x1080. The FPGA development board does not support a powerless HDMI to VGA converter, which requires a separate power supply converter

3) One HDMI cable

4) USB mouse and keyboard

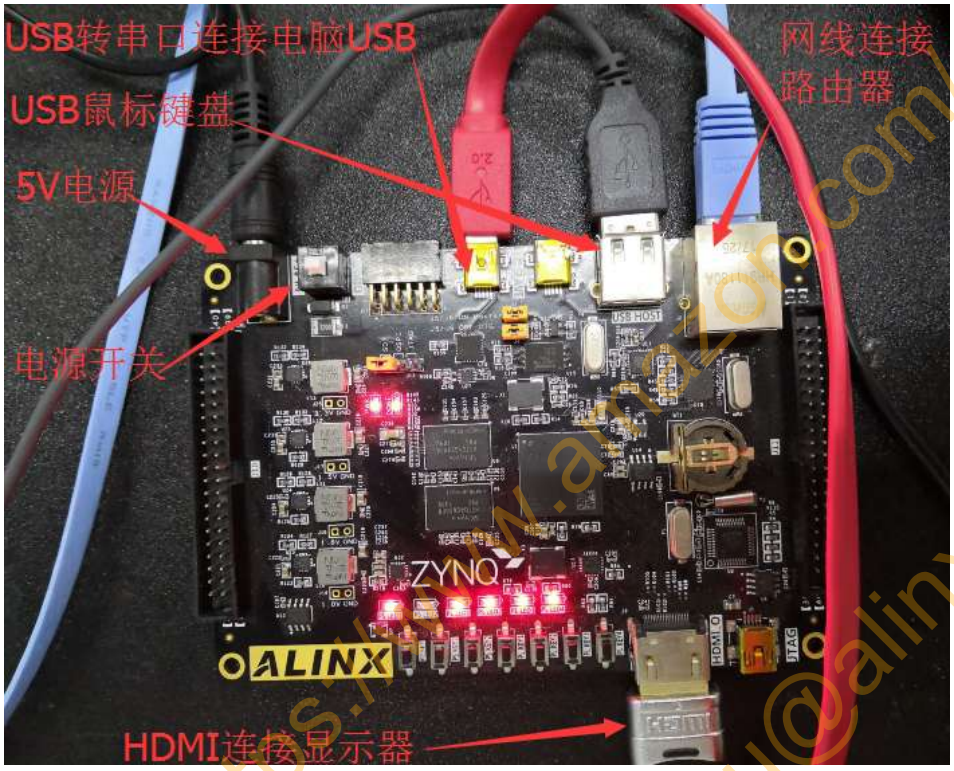
5) Routers, in order to test the network, it is best to connect to the Internet and support DHCP. This is very important, especially for developers who don't have the ability of configure the network. If it's a campus network, special carrier broadband, and a special router.



6) Network Cable

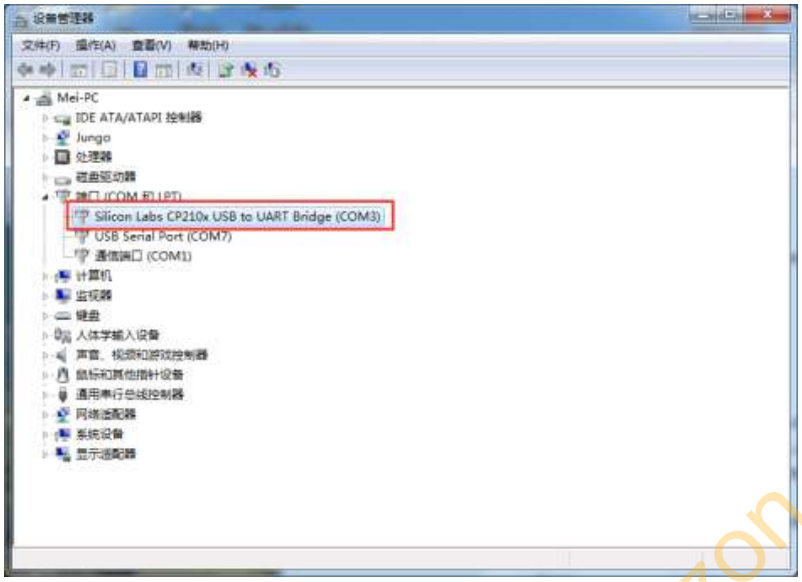
Part 1.2.2: FPGA Development Board Cable Connection

1) Connect HDMI display
2) Connect the network port to the router
3) Connect the power supply



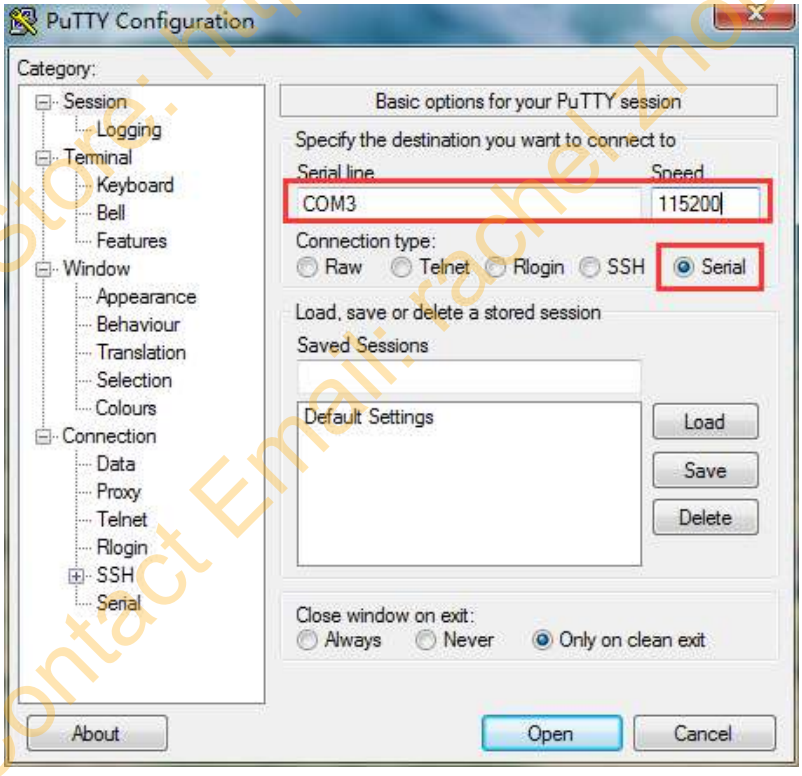
Part 1.2.3: Start Testing

1) Before testing, we need to install the USB to serial port driver software (software / CP210x_Windows_Drivers.zip), otherwise it can not do serial communication test. After the driver is installed, connect the USB port of the computer with the red USB cable and the UART port (J7) on the development board, then open the device manager of the computer. The device manager can find the serial port device CP210x. The mapping on the machine is COM3. . If you are unable to successfully install the driver, you can try to install it using the driver wizard.

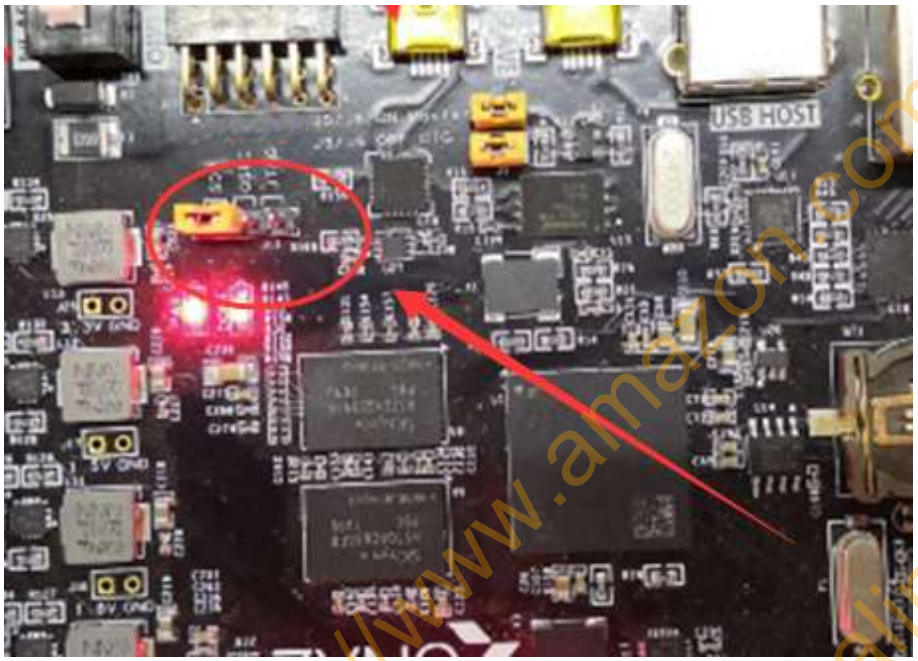


2) There are many terminal tools, such as putty, teraterm, Windows own terminal tools, SecureCRT, etc. Among many terminal tools, it is better to use putty, and the software package (software / putty.exe) has prepared free installable putty software for everyone.

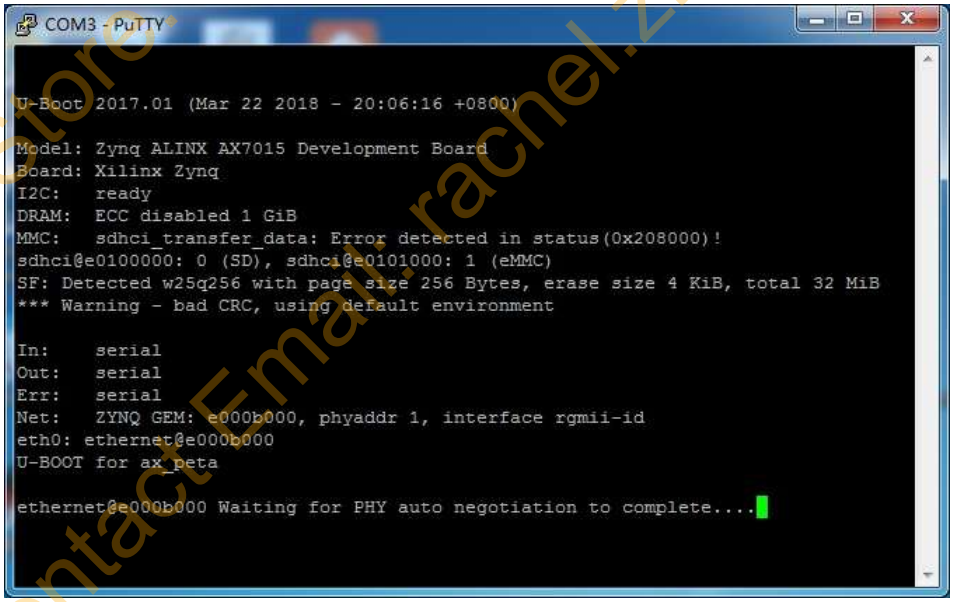
3) Select Serial, Serial line to fill in COM3, Speed fill in 115200, COM3 serial port number is filled in according to the display in the device manager, click "Open"



4) Determine whether the development board startup mode is SD startup mode (By default, there is a card in the SD card slot of the development board, and the startup mode is also the SD card by default). Modify the startup mode by the jump.

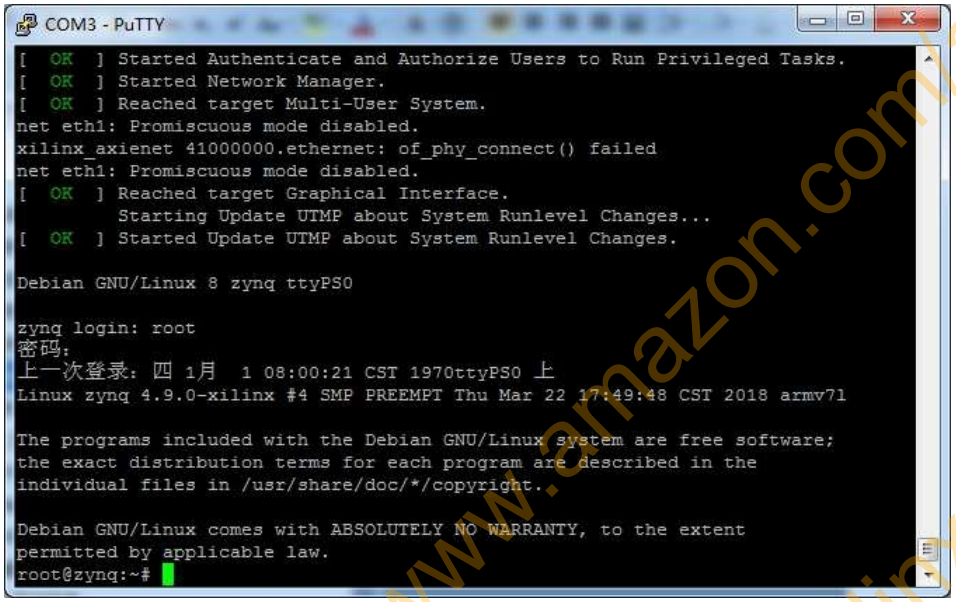


5) Turn on the power switch on the FPGA development board, and the PuTTY tool window will display the boot information of u boot and Linux system.

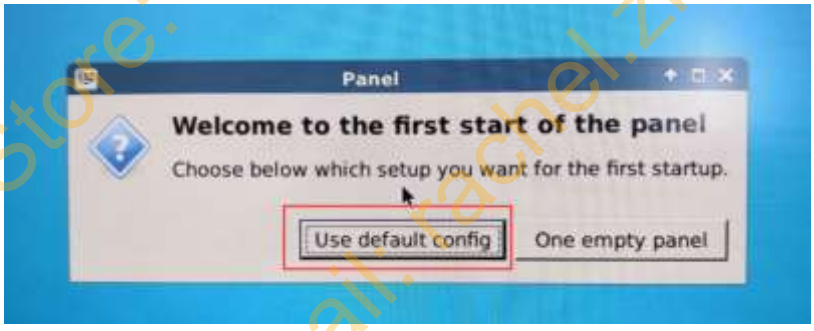


6) You can log in to the system at the serial terminal. User:root, password: root

Many people use Putty for the first time, or use the serial port for the first time. It needs to be explained that the Putty input command is input through the host keyboard, not through the keyboard input connected to the development board.

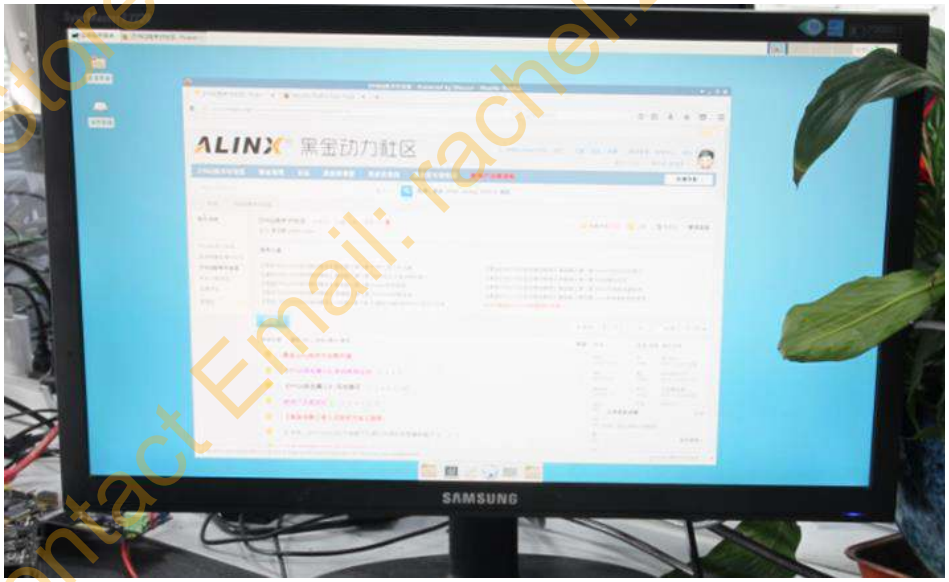


7) Once the boot is complete, connect the development board to the HDMI display to display the Debian desktop. You may be prompted to select the panel and select the default panel configuration.



8) At this time, you can use the mouse and keyboard to operate. Double-click the web browser with your mouse to start the browser for a long time. Please be patient.

9) Enter the website in the address bar. If it can be opened normally, the FPGA development board can be connected to the Internet.

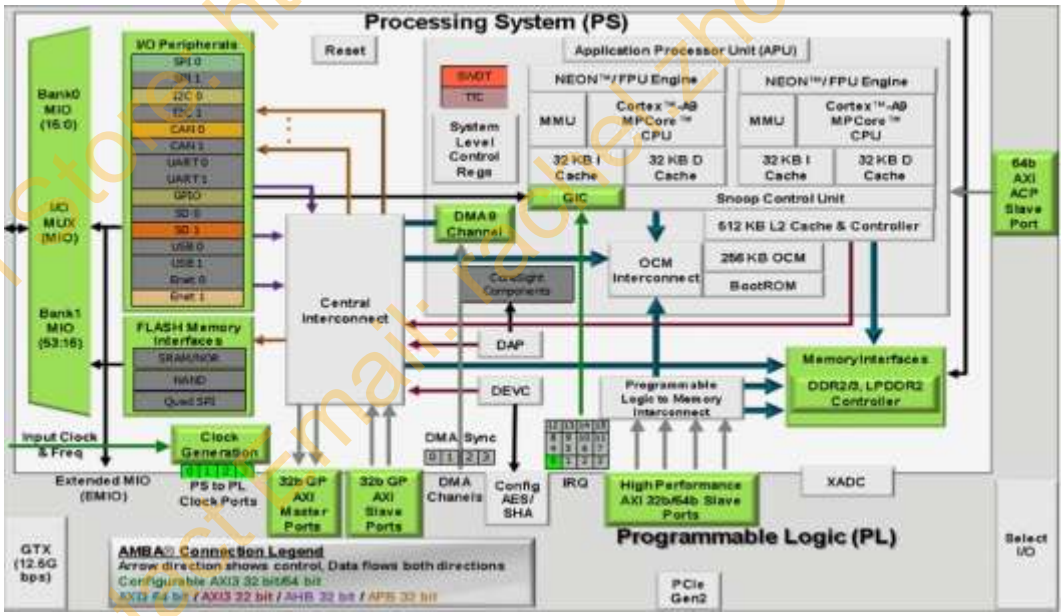


10) This is the end of the simple detection of the FPGA development board.

Part 2: Introduction to ZYNQ

The highlight of the Zynq family is that the complete ARM Processing Subsystem (PS) is included in the FPGA. Each Zynq family of processors includes the Cortex-A9 processor. The entire processor is processor-centric and processed. The memory controller and a large number of peripherals are integrated in the subsystem, making the Cortex-A9 core completely independent of the programmable logic unit in the Zynq-7000, which means that if the programmable logic unit (PL) is not used temporarily The subsystems of the ARM processor can also work independently, which is essentially different from previous FPGAs, which are processor-centric.

Zynq is the two major functional blocks of the PS part and the PL part, which is the SOC part of ARM and part of the FPGA. Among them, PS integrates two ARM CortexTM-A9 processors, AMBA® interconnect, internal memory, external memory interface and peripherals. These peripherals mainly include USB bus interface, Ethernet interface, SD/SDIO interface, I2C bus interface, CAN bus interface, UART interface, GPIO, etc.



Overall block diagram of the ZYNQ chip

PS: Processing System, FPGA-independent SOC portion of ARM

PL: Progarmmable Logic, is the FPGA part

Part 2.1: PS and PL interconnect technology

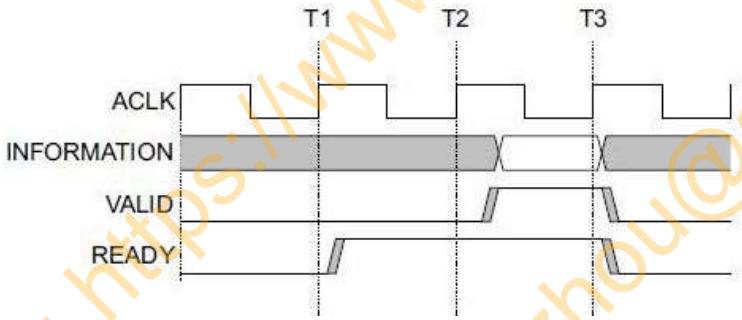
ZYNQ is the first product to combine high-performance ARM Cortex-A9 series processors with high-performance FPGAs in a single chip.In order to achieve high-speed communication and data interaction between ARM processor and FPGA, to take advantage of the performance of ARM processor and FPGA, it is necessary to design an efficient on-chip interconnection between high-performance processor and FPGA. Therefore, how to design efficient PL and PS data interaction channels is the top priority of ZYNQ chip design and one of the key to the success of product design. In this section, we will focus on the connection between PS and PL, and let users know the technology of the connection between PS and PL.

In fact, in the specific design, it isnot necessary to do too much work on the PS and PL connection technology, after joining the IP core, the system will automatically use the AXI interface to connect our IP core to the processor, we only You need to add a little more.

In fact, in the specific design, we often do not need to do too much work on the PS and PL. After we join the IP core, the system will automatically connect our IP core to the processor using the AXI interface.We only need to add a little more to it.

AXI full name Advanced eXtensible Interface, an interface protocol introduced by Xilinx from the 6 series FPGA, mainly describes the data transmission between the master device and the slave device. Continued to use in ZYNQ, the version is AXI4, so we often see AXI4.0, ZYNQ internal devices have AXI interface. In fact, AXI is part of the AMBA (Advanced Microcontroller Bus Architecture) proposed by ARM. It is a high-performance, high-bandwidth, low-latency on-chip bus that is also used to replace the previous AHB and APB buses. The first version of AXI (AXI3) was included in AMBA 3.0 released in 2003, and the second version of AXI (AXI4) was included in AMBA 4.0 released in 2010.

The AXI protocol mainly describes the data transmission mode between the master device and the slave device, and the master device and the slave device establish a connection through a handshake signal. The READY signal is issued when the slave is ready to receive data. When the master device's data is ready, the VALID signal is asserted and maintained to indicate that the data is valid. Data is only transmitted when both the VALID and READY signals are active. When these two signals remain active, the master will continue to transfer the next data. The master can either revoke the VALID signal or deactivate the READY signal from the device to terminate the transfer. The protocol of AXI is shown in the figure. When T2, the READY signal of the slave device is valid, When T3, the VILID signal of the master device is valid, and the data transmission starts.



AXI handshake timing diagram

In ZYNQ, AXI-Lite, AXI4 and AXI-Stream are supported. Through Table 5-1, we can see the characteristics of these three AXI interfaces.

Interface Protocol	Characteristic	Application
AXI4-Lite	Address/Single data transmission	Low-speed peripherals or Controls
AXI4	Address/burst data transfer	Bulk transfer of addresses
AXI4-Stream	Data transmission only, burst transmission	Data stream and media stream

AXI4-Lite:

It has the characteristics of lightweight and simple structure, suitable for small batch data and simple control occasions. Bulk transfer is not supported,

and only one word (32bit) can be read and written at a time. Mainly use to access and control ofsome low-speed peripherals.

AXI4：

The interface is similar to AXI-Lite, except that one feature is added for bulk transfer, which allows one-time read and write of an address. It means that there is a burst function for reading and writing data.

The above two methods use memory mapping control, that is, ARM encodes the user-defined IP into an address for access. When reading and writing, it is like reading and writing its own on-chip RAM. The programming is also very convenient, and the development is less difficult. The cost is that the resources are too much, and additional signal lines such as read address lines, write address lines, read data lines, write data lines, and write answer lines are required.

AXI4-Stream：

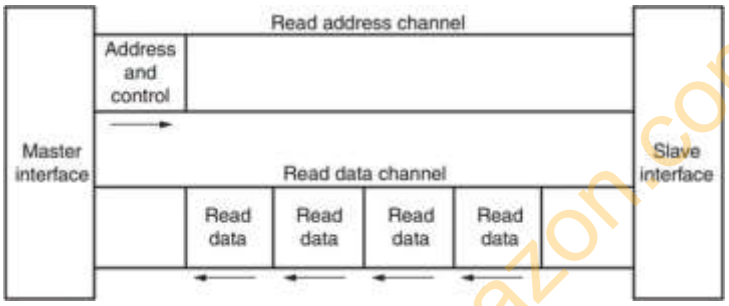
This is a continuous stream interface that does not require an address line (much like a FIFO, which is always read or written all the time). For this type of IP, ARM cannot be controlled by the above memory mapping method (the FIFO does not have the concept of an address at all), and there must be a conversion device, such as an AXI-DMA module, to implement memory-to-streaming conversion. There are many applications for AXI-Stream: video stream processing; communication protocol conversion; digital signal processing; wireless communication. The essence is a data path built for numerical flow, from a source (such as ARM memory, DMA, wireless receiving front-end, etc.) to a sink (such as HDMI display, high-speed AD audio output, etc.) to build a continuous stream of data. This interface is suitable for real-time signal processing.

The AXI4 and AXI4-Lite interfaces contain 5 different channels:

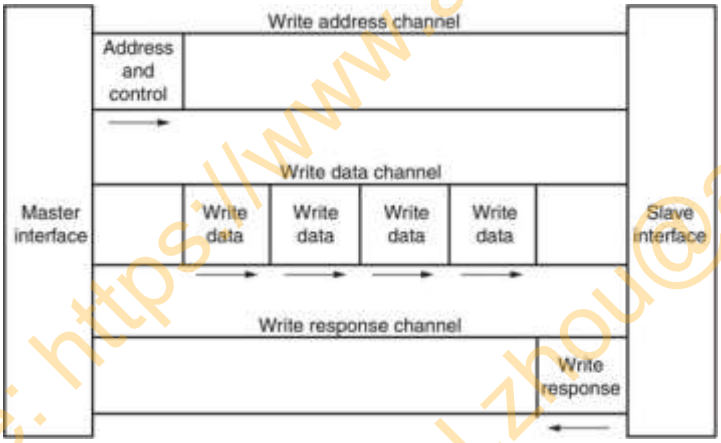
- Read AddressChannel
- Write AddressChannel

- Read DataChannel
- Write DataChannel
- Write ResponseChannel

Each of these channels is a separate AXI handshake protocol. The following two figures show the models for reading and writing:



AXI read data channel



AXI write data channel

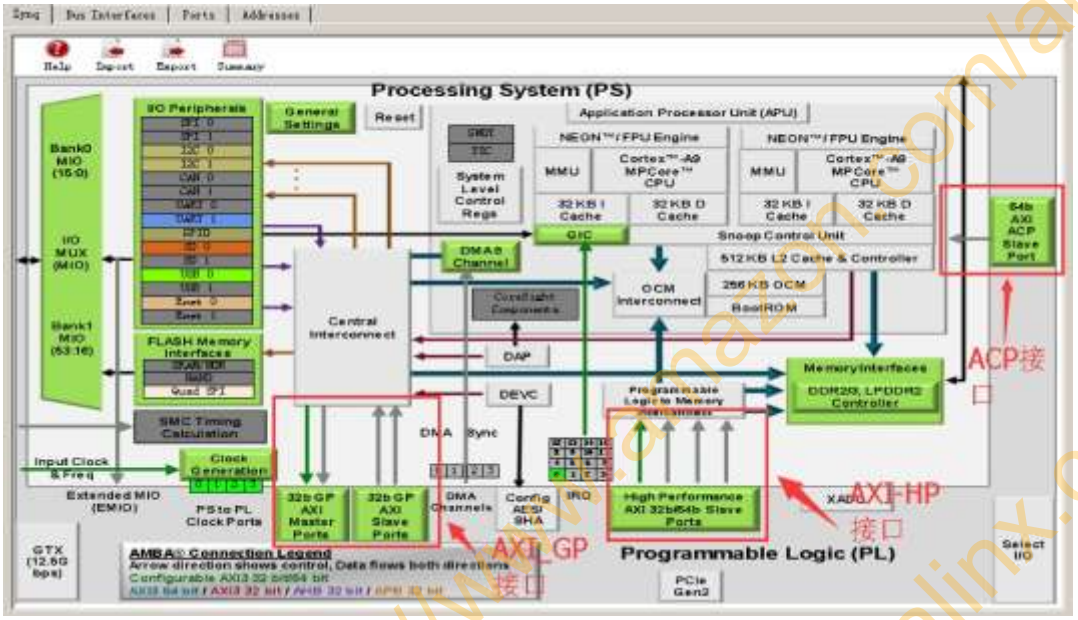
The AXI bus protocol is implemented in hardware inside the ZYNQ chip, including 9 physical interfaces, namely AXI-GP0~AXI-GP3, AXI-HP0~AXI-HP3, AXI-ACP interface.

The AXI_ACP interface is an interface defined by the ARM multi-core architecture. The Chinese translation is an accelerator coherent port for managing AXI peripherals without buffers such as DMA. The PS side is the Slave interface.

The AXI_HP interface is a high-performance/bandwidth AXI3.0 standard interface with a total of four, and the PL module is connected as a master.

Mainly used for PL access to the memory on the PS (DDR and On-Chip RAM)

The AXI_GP interface is a generic AXI interface with a total of four, including two 32-bit master interfaces and two 32-bit slave interfaces.

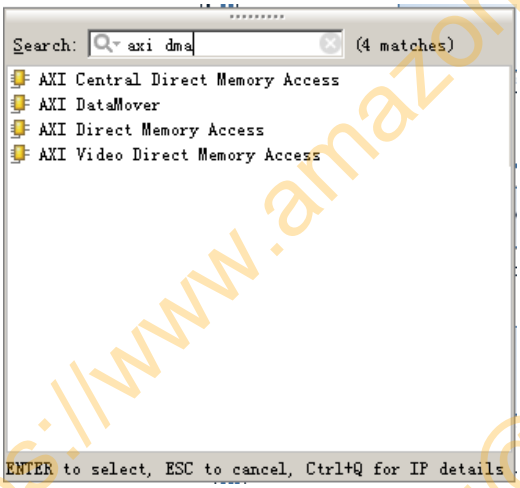


It can be seen that only two AXI-GPs are Master Ports, ie host interfaces, and the remaining seven ports are Slave Ports. The host interface has the right to initiate read and write. ARM can use the two AXI-GP host interfaces to actively access the PL logic. In fact, it maps the PL to an address, and reads and writes the PL register as if it is reading and writing its own memory. The remaining slave interfaces are passive interfaces, accepting reads and writes from the PL, and accepting them.

In addition, the performance of these 9 AXI interfaces is also different. The GP interface is a 32-bit low-performance interface with a theoretical bandwidth of 600MB/s, while the HP and ACP interfaces are 64-bit high-performance interfaces with a theoretical bandwidth of 1200MB/s. Some people willaskwhy is the high-performance interface not a host interface? This allows high speed data transfer by ARM. The answer is that the high-performance interface does not require an ARM CPU at all to be responsible

for data movement. The real porter is the DMA controller in the PL.

The ARM on the PS side has hardware directly supporting the AXI interface, while the PL requires logic to implement the corresponding AXI protocol. Xilinx provides off-the-shelf IP in the Vivado development environment, such as AXI-DMA, AXI-GPIO, AXI-Dataover, and AXI-Stream. The corresponding interfaces are implemented and added directly from Vivado's IP list to achieve the corresponding functions. The following picture detailed various DMA IPs under Vivado:



The following is a description of the functions of several commonly used AXI interface IPs:

AXI-DMA: Achieve conversion from PS memory to PL high-speed transmission AXI-HP<---->AXI-Stream

AXI-FIFO-MM2S: Implement conversion from PS memory to PL universal transfer channel AXI-GP<----->AXI-Stream

AXI-Datamover： Realize the conversion from PS memory to PL high-speed transmission high-speed channel AXI-HP<---->AXI-Stream, but this time it is completely controlled by PL, PS is completely passive.

XI-VDMA: Realize the conversion from PS memory to PL high-speed transmission AXI-HP<---->AXI-Stream, which is only for 2D data such as video and image.

AXI-CDMA: This is done by PL to move data from one location to

another in memory, without the need for a CPU to intervene.

We will talk about how to use these IPs in the following sections. Sometimes, users need to develop their own defined IP to communicate with PS. In this case, you can use the wizard to generate the corresponding IP. User-defined IP cores can have interfaces such as AXI4-Lite, AXI4, AXI-Stream, PLB and FSL. The latter two are not supported because ARM does not support this end.

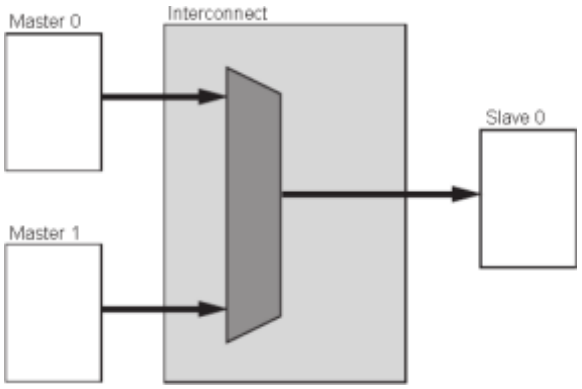
With the custom IP generated by these official IPs and wizards, users don't need to know too much about AXI timing (unless they do have problems), because Xilinx has encapsulated the details related to AXI timing, users only need Pay attention to your own logic implementation.

The AXI protocol is strictly a peer-to-peer master-slave interface protocol. When multiple peripherals need to interact with each other, we need to add an AXI Interconnect module, which is the AXI interconnect matrix, to provide one or more AXI masters. The role is to provide a switch mechanism (Somewhat similar to the switch matrix in the switch) that connects one or more AXI masters to one or more AXI slaves.

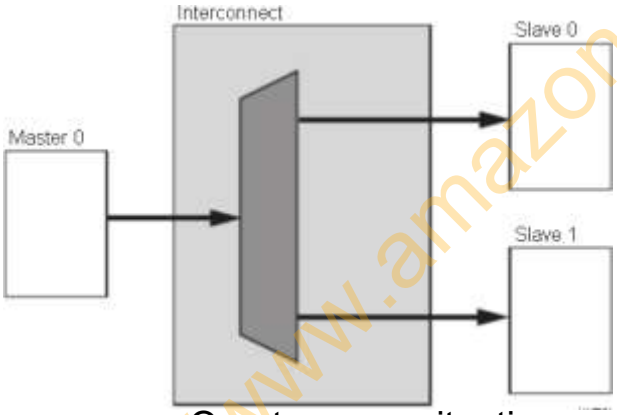
This AXI Interconnect IP core can support up to 16 master devices and 16 slave devices. If you need more interfaces, you can add more IP cores.

The AXI Interconnect basic connection modes are as follows:

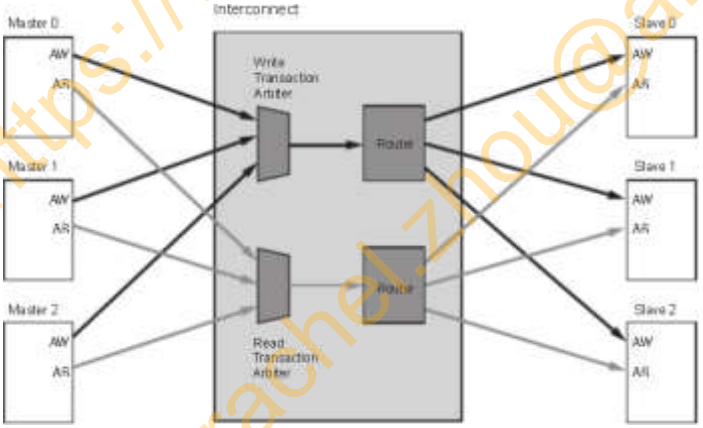
- N-to-1 Interconnect
- to-N Interconnect
- N-to-M Interconnect (Crossbar Mode)
- N-to-M Interconnect (Shared Access Mode)



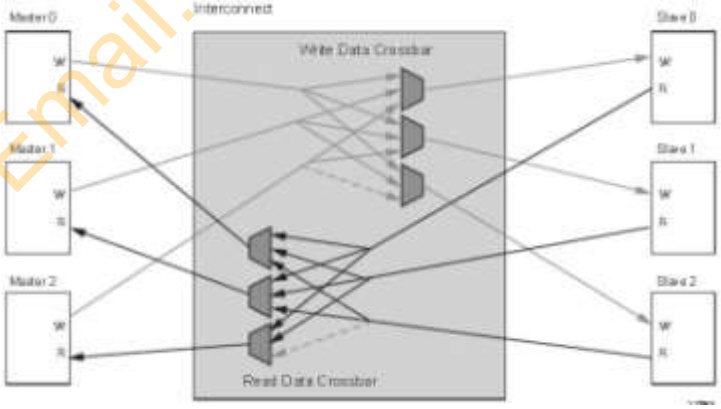
Many-to-one situation



One-to-many situation

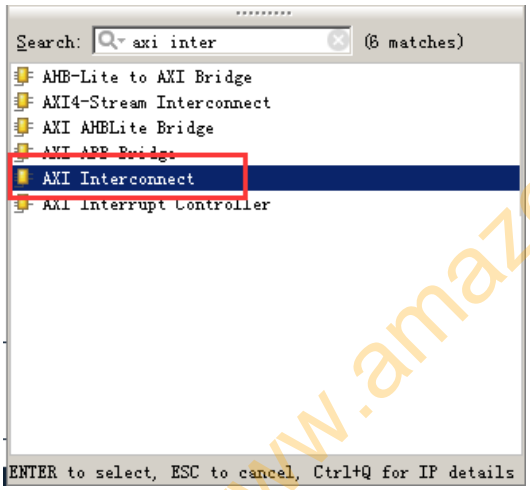


Many-to-many read and write address channels



Many-to-many read and write data channels

The AXI interface devices inside ZYNQ are interconnected by means of interconnecting matrices, which not only ensures the efficiency of data transmission, but also ensures the flexibility of connection. Xilinx In Vivado we provide the IP core axi_interconnect that implements this interconnect matrix, which we can just call.



AXI Interconnect IP

Part 2.2: Introduction to the ZYNQ chip development process

Because ZYNQ integrates the CPU with the FPGA, developers need to design ARM's operating system applications and device drivers as well as the hardware logic design of the FPGA part. In development, it is necessary to understand the Linux operating system, the architecture of the system, and the hardware design platform between the FPGA and the ARM system. Therefore, the development of ZYNQ requires the software personnel and hardware personnel to design and develop synergistically. This is both the so-called "software and hardware co-design" in ZYNQ development.

The design and development of the hardware and software systems of the ZYNQ system requires the development environment and debugging tools:Xilinx Vivado

The Vivado Design Suite implements the design and development of the FPGA section, pin and timing constraints, compilation and simulation, and implements the RTL to bitstream design flow. Vivado is not a simple upgrade

to the ISE Design Suite, but a completely new design suite. It replaces all the important tools of the ISE Design Suite, such as design tools of Project Navigator, Xilinx SynthesisDesign Technology, Implementation, CORE Generator, Constraint, Simulator, Chipscope Analyzer, FPGA Editoretc.

The SDK is the Xilinx Software Development Kit (SDK). Based on the Vivado hardware system, the system automatically configures some important parameters, including tool and library paths, compiler options, JTAG and flash settings. The Debugger connection already bare board support package (BSP). The SDK also provides drivers for all supported Xilinx IP hard cores. The SDK supports IP hard core (on FPGA) and processor software for collaborative debugging. We can use advanced C or C++ language to develop and debug ARM and FPGA systems to test whether the hardware system is working properly. The SDK software is also included with the Vivado software and does not need to be installed separately.

The development of ZYNQ is also the method of software after hardware. The specific process is as follows:

1) Create a new project on Vivado and add an embedded source file.
2) Add and configure basic PS and PL peripherals in Vivado, or add custom peripherals.
3) Generate a top-level HDL file in Vivado and add a constraint file. Recompile and generate a bitstream file (*.bit).
4) Export hardware information to the SDK software development environment. In the SDK environment, you can write some debugging software to verify the hardware and software, and debug the ZYNQ system separately with the bitstream file.
5) Generate FSBL files in the SDK
6) Generate a u-boot.elf, bootloader image in a VMware virtual machine.

7) A BOOT.bin file is generated in the SDK via the FSBL file, the bitstream files system.bit and the u-boot.elf file.
8) Generate Ubuntu kernel image files, Zimage and Ubuntu root file systems in VMware. You also need to write drivers for the FPGA's custom IP.
9) Put the BOOT, kernel, device tree, and root file system files into the SD card, start the development board power, and the Linux operating system will boot from the SD card.

The above is a typical ZYNQ development process, but ZYNQ can also be used as ARM alone, so there is no need to relate to PL resources, which is not much different from traditional ARM development. ZYNQ can also use only the PL part, but the configuration of the PL is still done by PS, that is, the firmware of the PL cannot be solidified by the conventional solidified flash method.

Part 2.3: What skills do you need to learn ZYNQ?

Learning ZYNQ is more difficult than learning traditional tools such as FPGA, MCU, and ARM. It is not a one-time thing to learn ZYNQ well.

Part 2.3.1: software developer

- ✓ Computer composition principle
- ✓ C, C++ language
- ✓ Computer operating system
- ✓ Tcl script

Part 2.3.2: Logic developer

- ✓ Computer composition principle
- ✓ C language
- ✓ Digital circuits
- ✓ Verilog, VHDL language

Part 3: Vivado development environment

Part 3.1: Vivado Software Introduction

When it comes to Xilinx's development environment, people always think of ISE first, but not Vivado. In fact, Vivado is the next-generation integrated design environment that Xilinx introduced in 2012. Although its popularity is currently low, it can be said that Vivado represents a trend in the future Xilinx FPGA development environment. Therefore, as a development user of Xilinx FPGA, it is inevitable to learn to master Vivado. As a developer, you must first have the following doubts: Since ISE already exists, why does Xilinx spend a lot of effort on Vivado? As mentioned in the Vivado Design Suite User Guide: Getting Started (UG910), Vivado was introduced to increase the efficiency of the designer, which significantly increases the design, synthesis, and implementation efficiency of Xilinx's 28nm process programmable logic devices. It can be speculated that as the FPGA enters the 28nm generation, the ISE tool seems to be somewhat "out of date", and the hardware is improved. If the software is not upgraded, the design efficiency will inevitably be affected. It is for this reason that Xilinx began planning to launch a new generation of software development environment in 2008, and it took 10 years to create the peak of Vivado tools.

Part 3.2: Vivado software version

The software version of Vivado is constantly being upgraded, and the latest software version so far is already 2017.4. Because all the routines and tutorials for the ZYNQ development board, done in the development environment of Vivado 2017.4. In order to avoid the cause of the software version and cause some unexplained problems, I hope that everyone will keep pace with us during the learning process. Users need to install Vivado 2017.4 software before using it. Because the Vivado software is relatively large, we do not provide the CD installation file, only the download link is

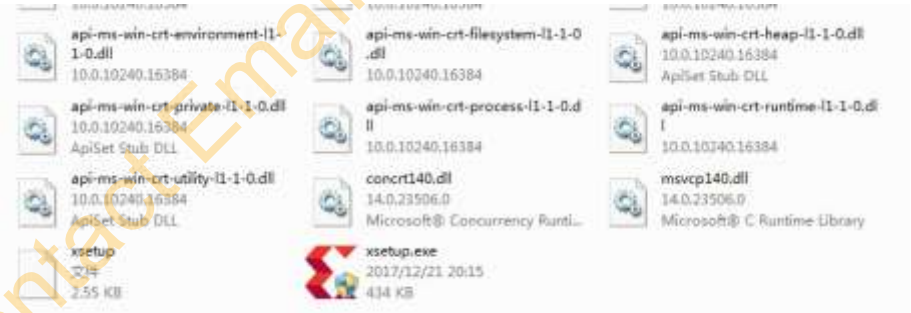
provided, and the user can also download it from the official website of Xilinx. The official website download needs to register the relevant account.

Xilinx official download address of Vivado software:
http://china.xilinx.com/support/download.html

Vivado is available in Linux and Windows versions. It is also available in 2-in-1 version. We use the 2-in-1 version to meet both Windows development and Linux development. Vivado requires the operating system to be 64-bit.

Part 3.3: Vivado software installed under Windows

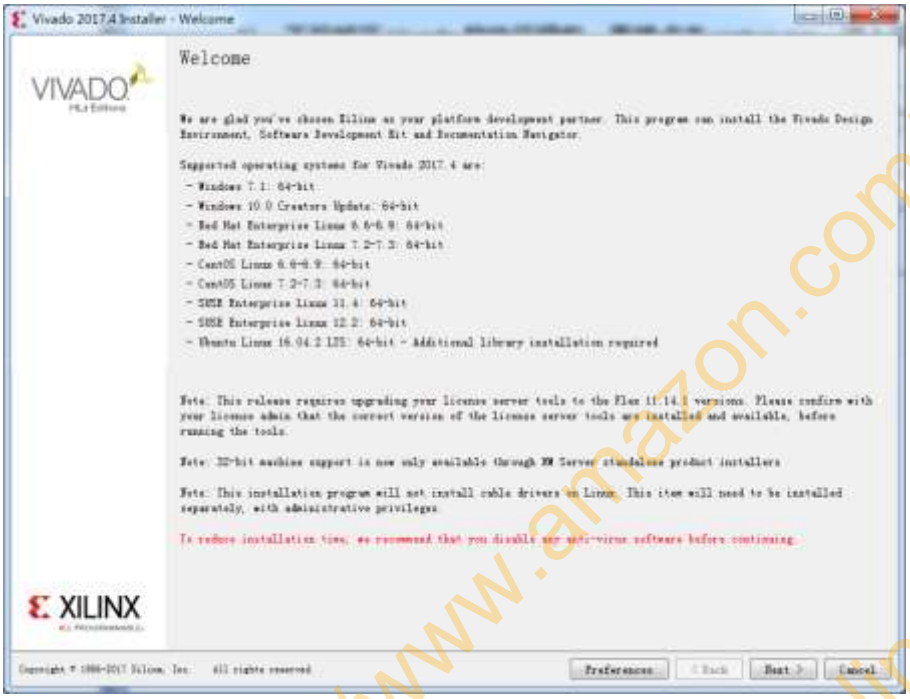
1) Download and unzip the Vivado software package, just click xsetup.exe to enter the installation, but for better installation, please



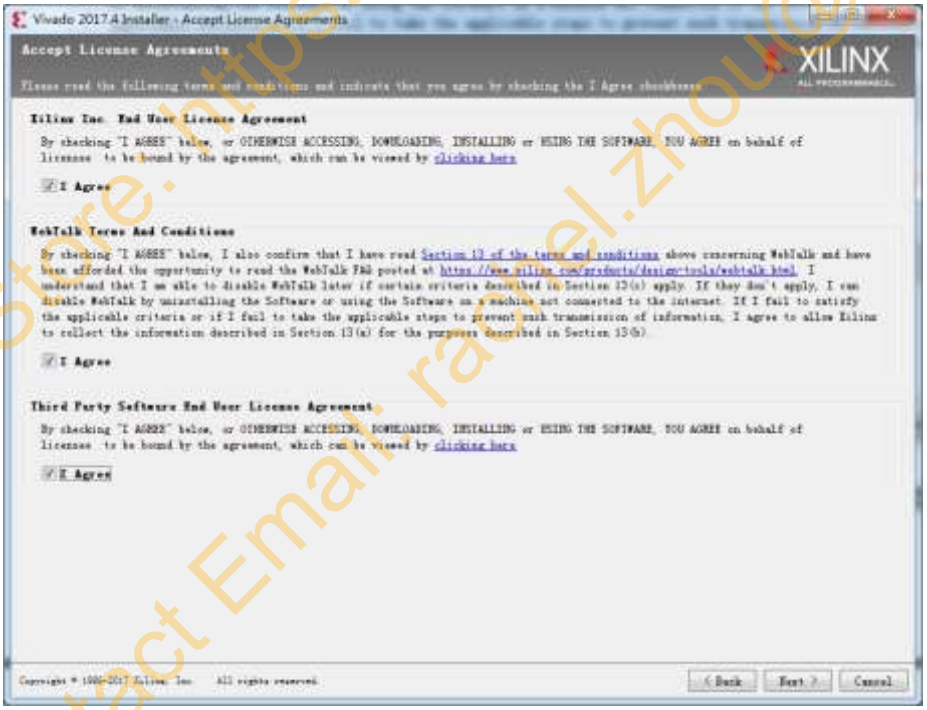
turn off the anti-virus software, all kinds of computer housekeeper, **the computer user name does not have Chinese, and spaces**

2) If the prompt version is updated, we ignore the update and click "Continue"

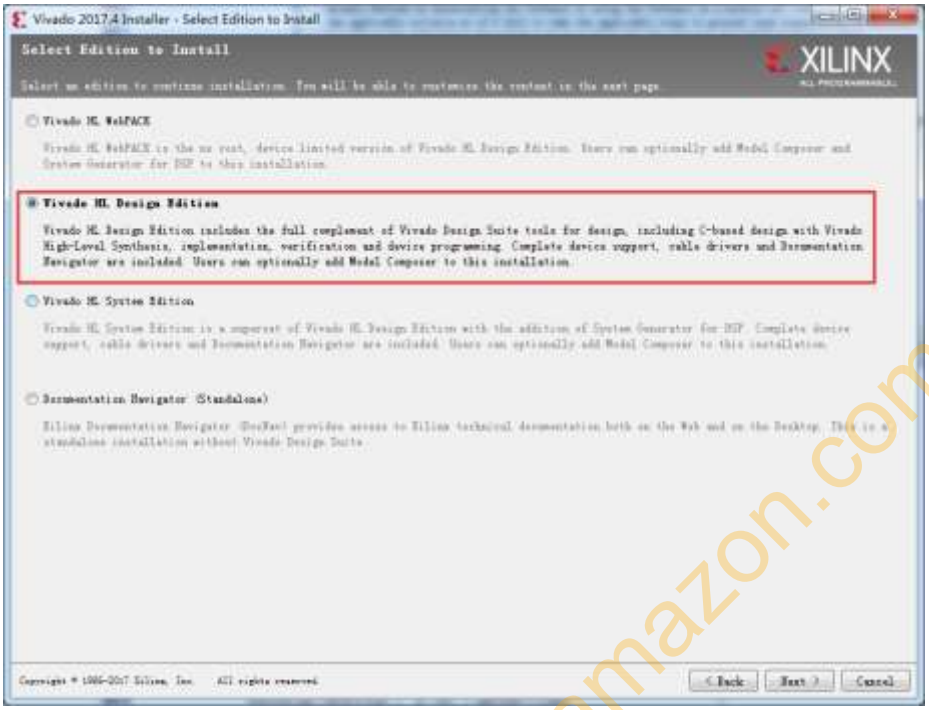
3) Click "next" to install and you can see Vivado's system requirements.



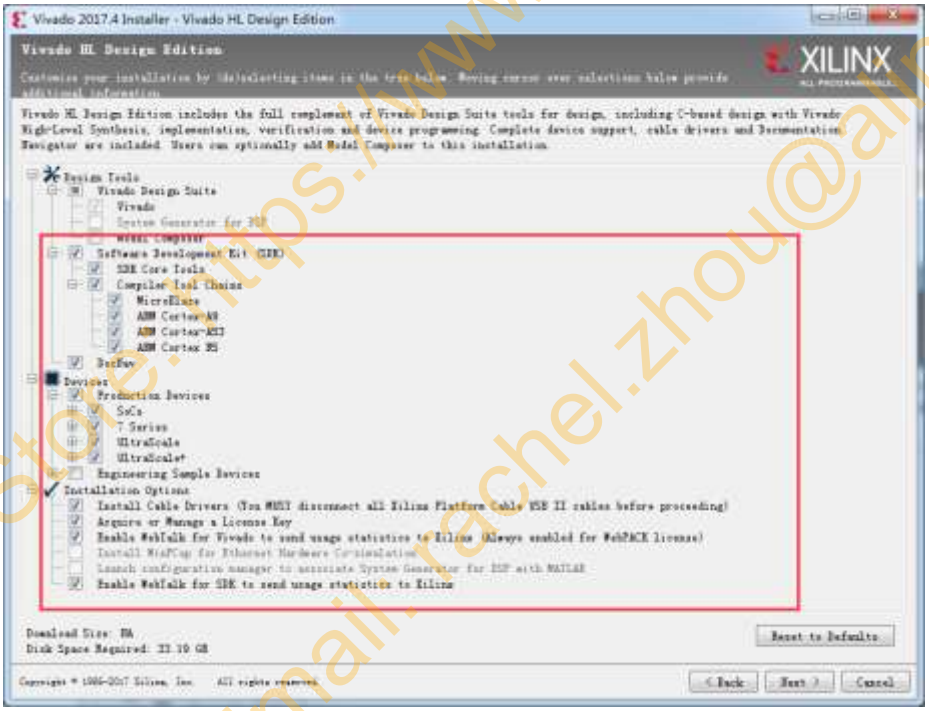
4) Click on "I Agree" to accept the terms



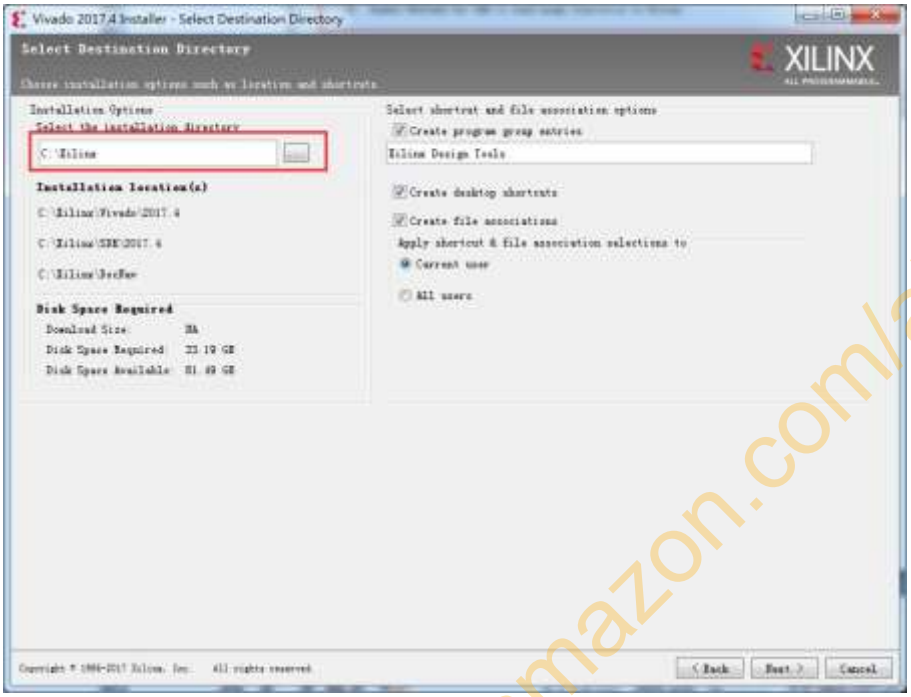
5) Select "Vivado HL Design Edition"



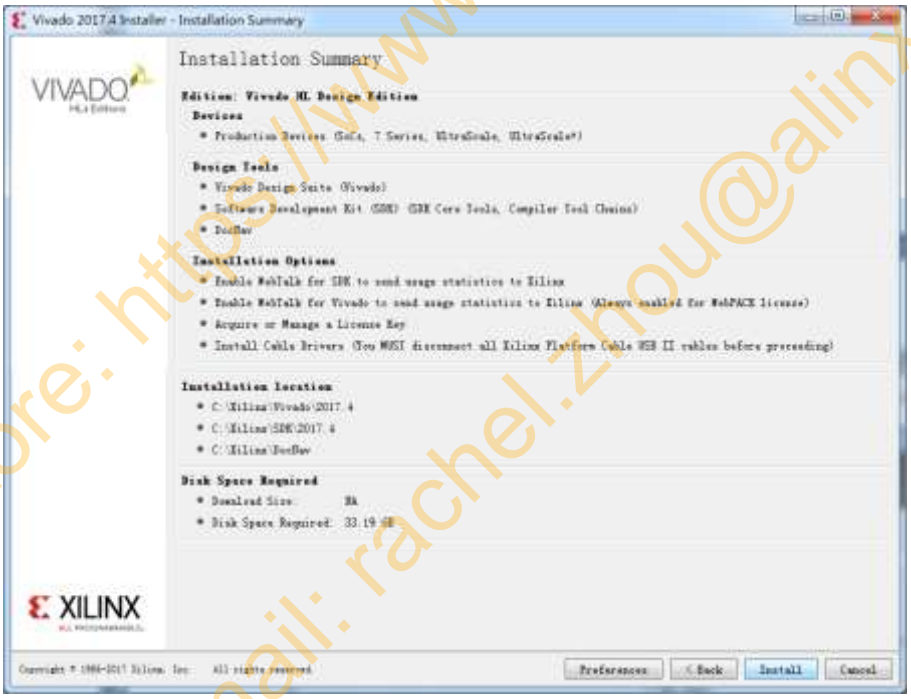
6) Use the default configuration here, click on "next"



7) The installation path is not modified here. The installation path cannot have special characters such as Chinese characters and spaces. The user name of the computer should not be Chinese or a space with a space. You can see Vivado's hard drive size requirement, about 33G.



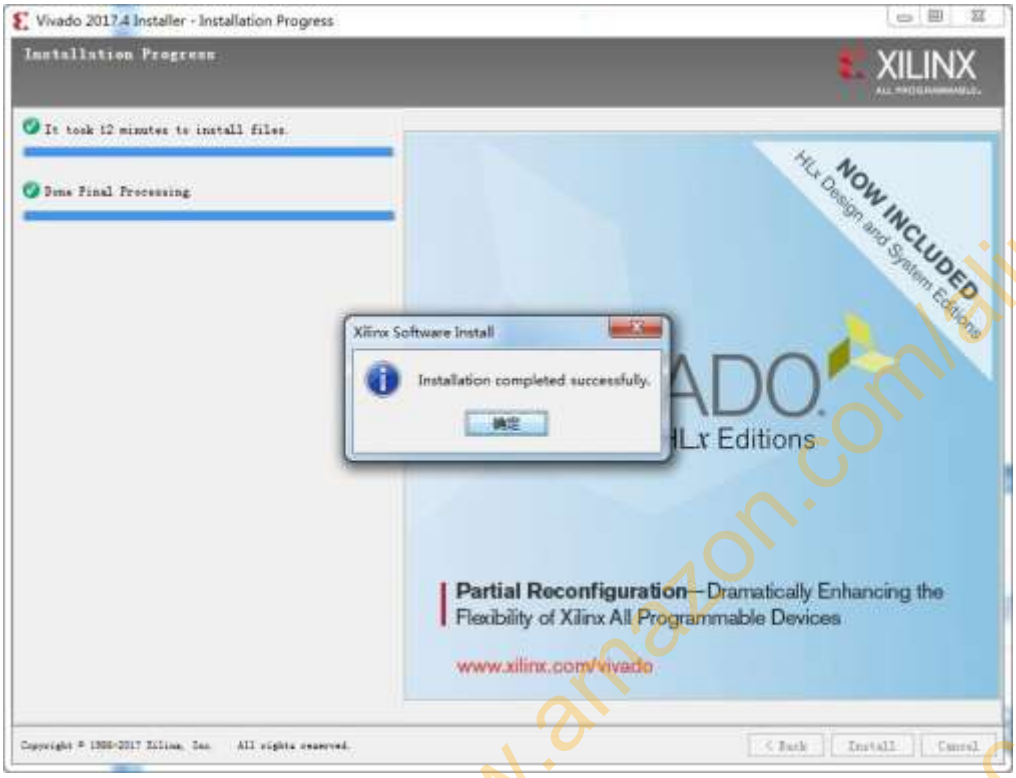
8) Click "Install" to install



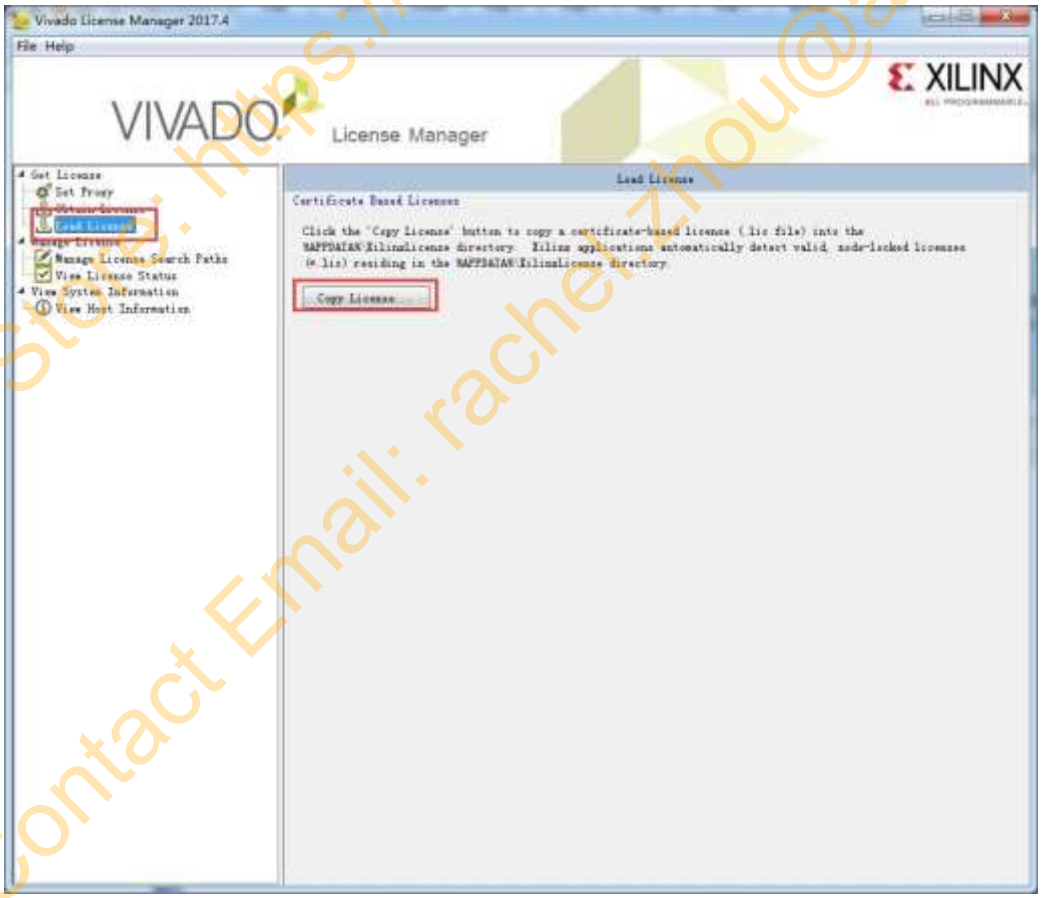
9) Waiting for installation, the time is long, if you do not close the anti-virus software and computer housekeeper, the installation process may be blocked, resulting in the installation of the software can not be used

10)At this time, we are prompted to disconnect the JTAG line of the downloader or development board and click "OK".

11)Prompt to install successfully



12) Install the license file, click "Copy License", and select the "xilinx_ise_vivado.lic" file.



13)Successful installation.Be reminded that computer usernames do not have Chinese and spaces.

14)If you need to install the downloader driver again, go to the vivado installationpath"X:\Xilinx\Vivado\2017.4\data\xicom\cable_drivers\nt64\digilent", double-click the "install_digilent.exe" file to install, close the vivado software before installation. If vivado does not recognize the downloader, try turning off the firewall, anti-virus software, and not open multiple versions of vivado, ise at the same time.

Part 4: PL's "Hello World" LED experiment

Experiment Vivado project "led"

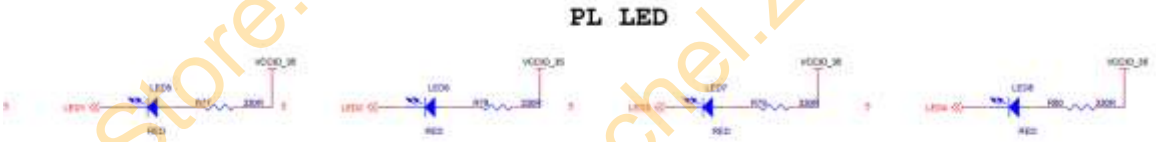
The contents of this chapter are mainly implemented by FPGA engineers.

For ZYNQ, PL (FPGA) development is crucial. This is where ZYNQ has advantages over other ARMs. You can customize many ARM peripherals. Let us pass an LED example before customizing the ARM peripherals. Cheng is familiar with the development process of PL (FPGA) and is familiar with the basic operation of Vivado software. This development process is completely consistent with the FPGA chip without ARM.

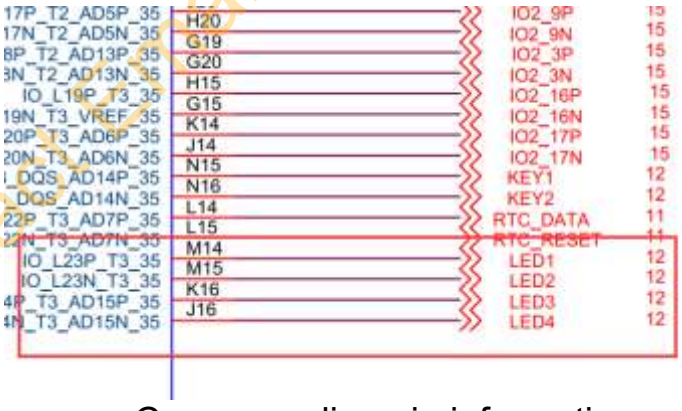
In this routine, what we need to do is the LED light control experiment. The LEDs on the development board are flipped once every second to achieve the control of on, off, on, off. Will control the LED lights, other peripherals will slow.

Part 4.1: LED hardware introduction

1) The PL part of the development board is connected to 4 red user LEDs. These 4 LEDs are completely controlled by the PL.

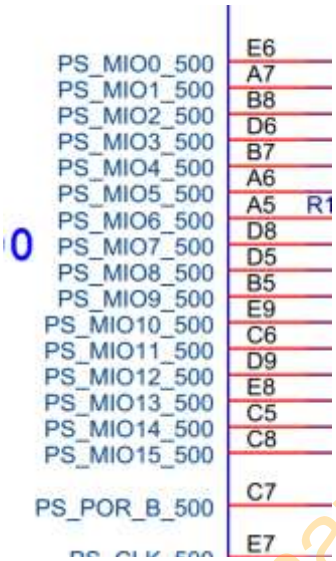


2) Determine the binding relationship between the LED and the PL pin according to the connection relationship of the schematic diagram.



Corresponding pin information

3) The IOs beginning with PS_MIO in the schematic are all PS-side IOs. They do not need to be bound or bound.

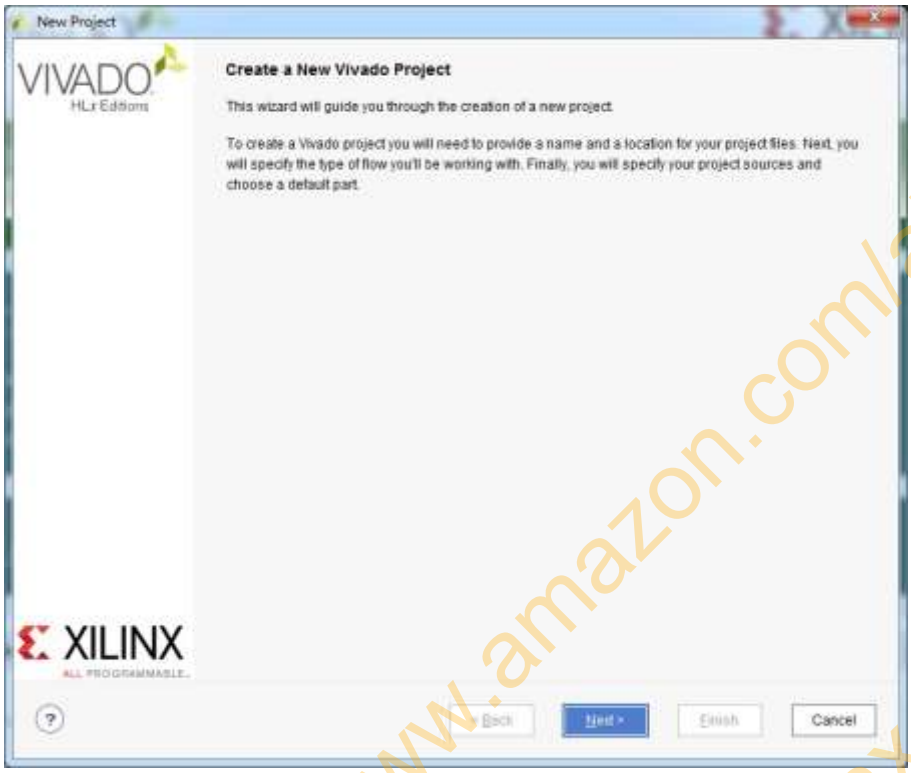


Part 4.2: Create a Vivado project

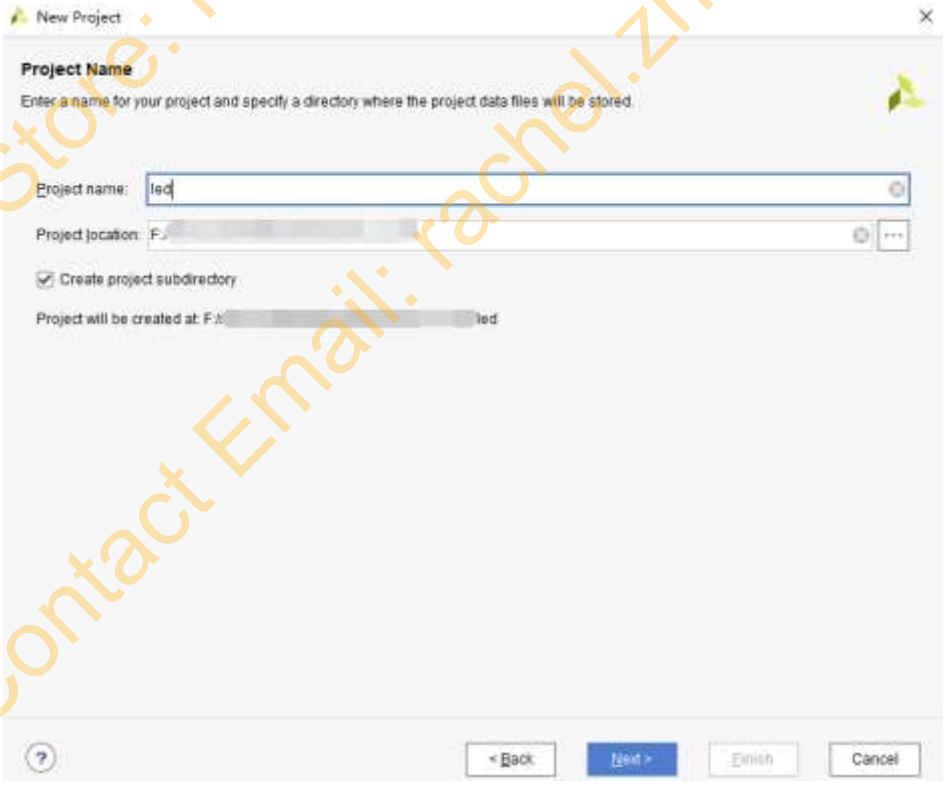
1) Start Vivado, which can be launched in Windows by double-clicking the Vivado shortcut

2) Create a new project by clicking on "Create New Project" in the Vivado 2017.4 development environment.

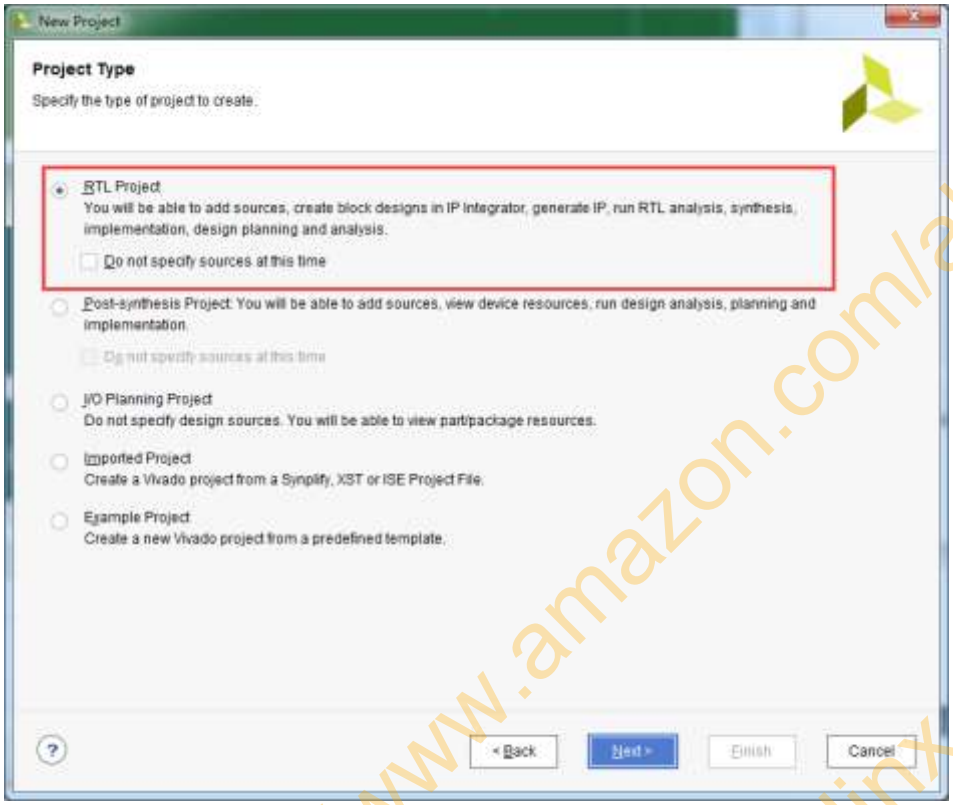
3) A wizard to create a new project pops up, click "Next"



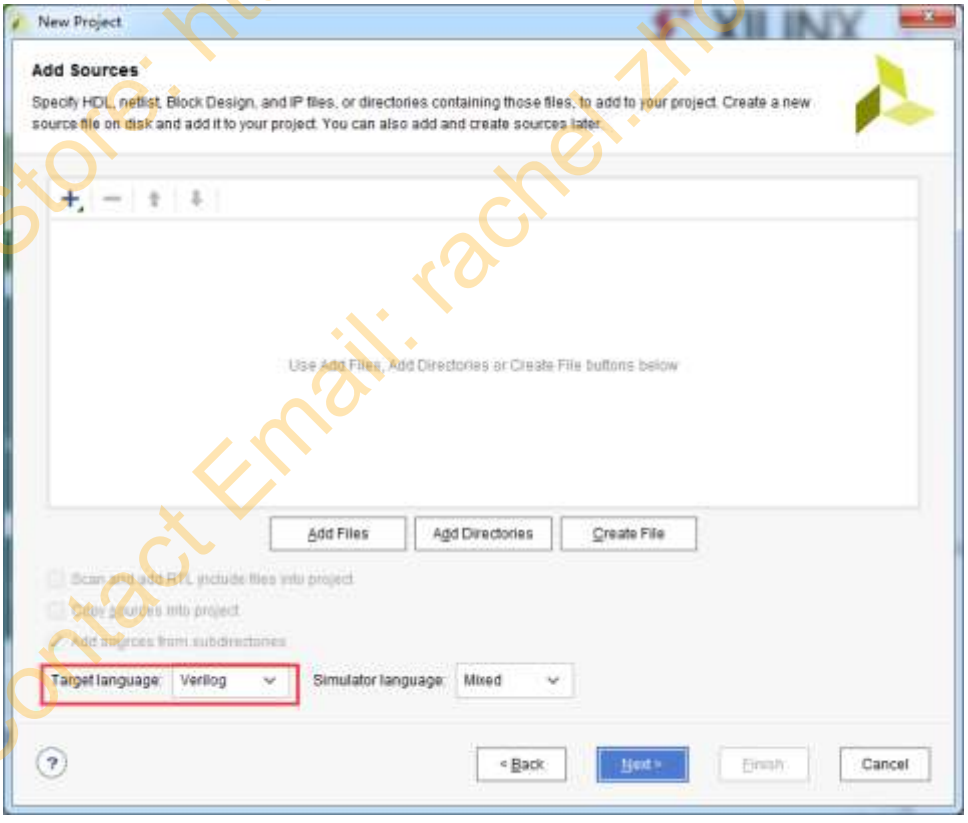
4) In the pop-up dialog box, enter the project name and the directory where the project is stored. We will take a led project name here. It should be noted that the project path "Project location" cannot have Chinese spaces, and the path name cannot be too long.



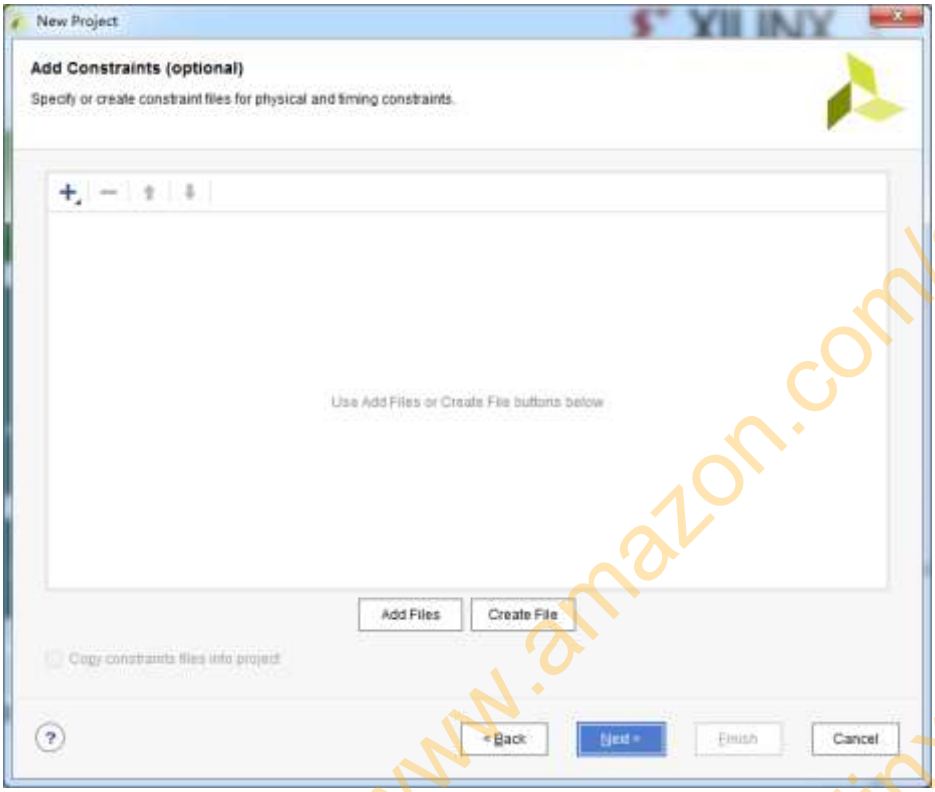
5) Select "RTL Project" in the project type



6) The target language "Target language" selects "Verilog". Although Verilog is selected, VHDL can also be used to support multi-language mixed programming.

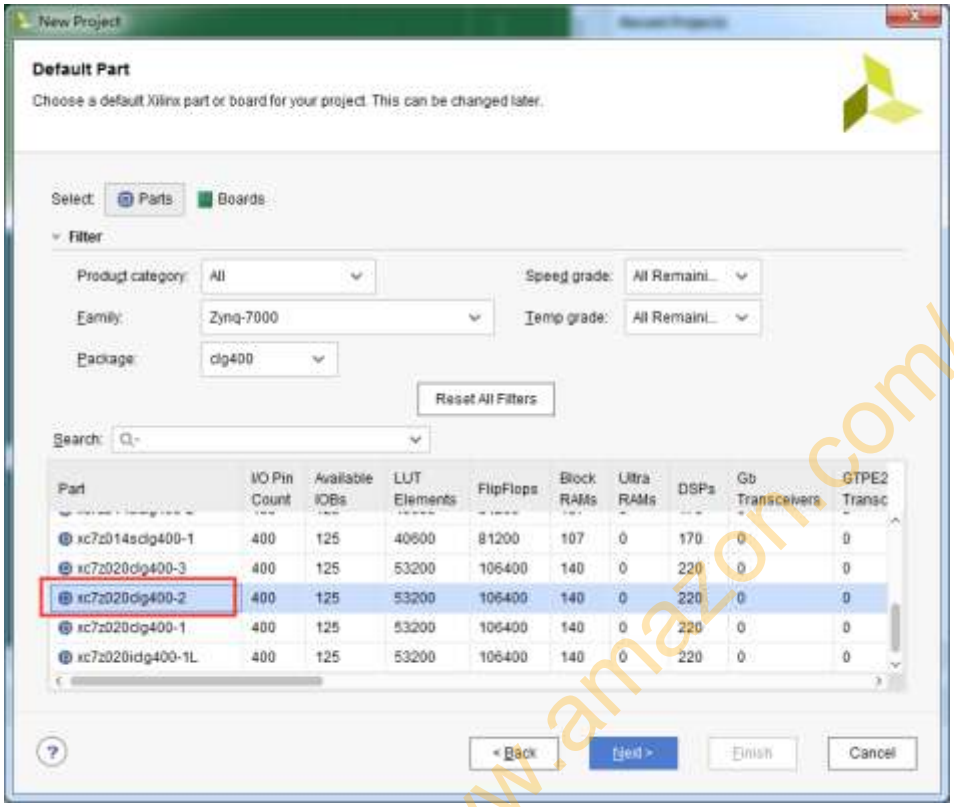


7) Click "Next" without adding any files

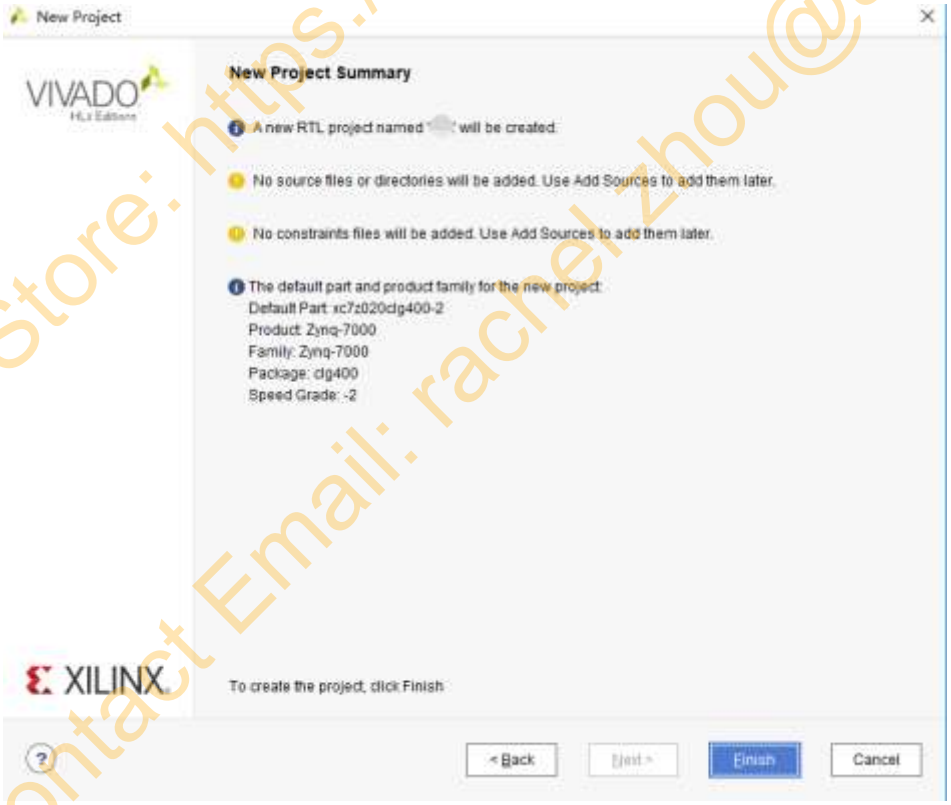


8) In the "Default Part" option, the device family "Family" selects "Zynq-7000", and the package type "Package" selects "clg400" to reduce our selection range.

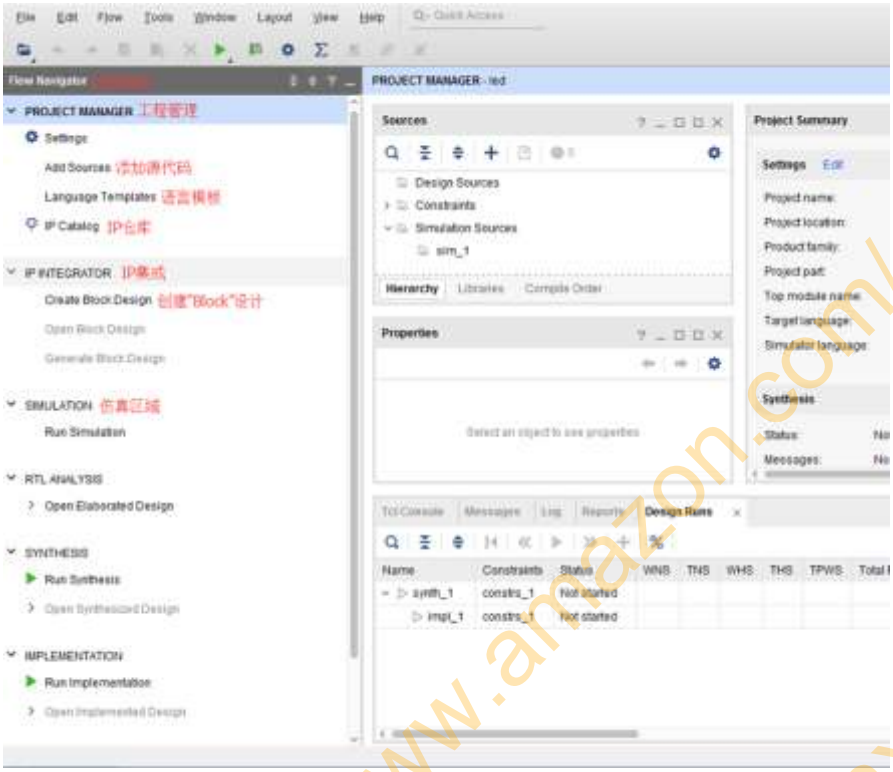
The AX7010 FPGA development board selects "xc7z010clg400-1" in the drop-down list. The AX7020 FPGA development board selects "xc7z020clg400-2" in the drop-down list, "2" indicates the speed level, the higher the number, the better the performance, and the higher the speed chip is down Compatible with low-speed chips.



9) Click "Finish" to complete the creation of the project named "led"

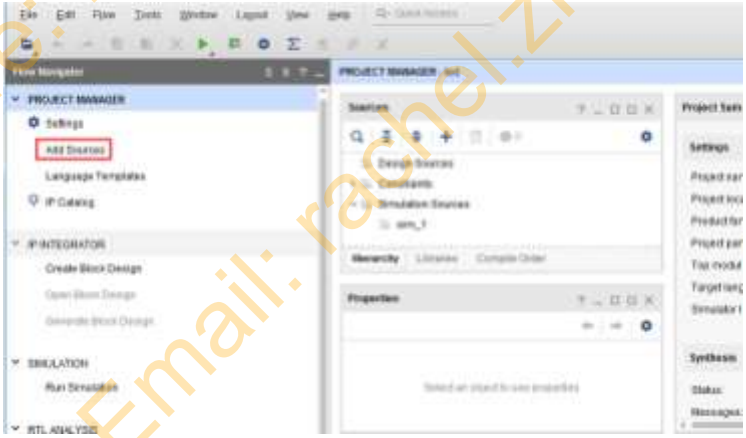


10)Vivado software interface

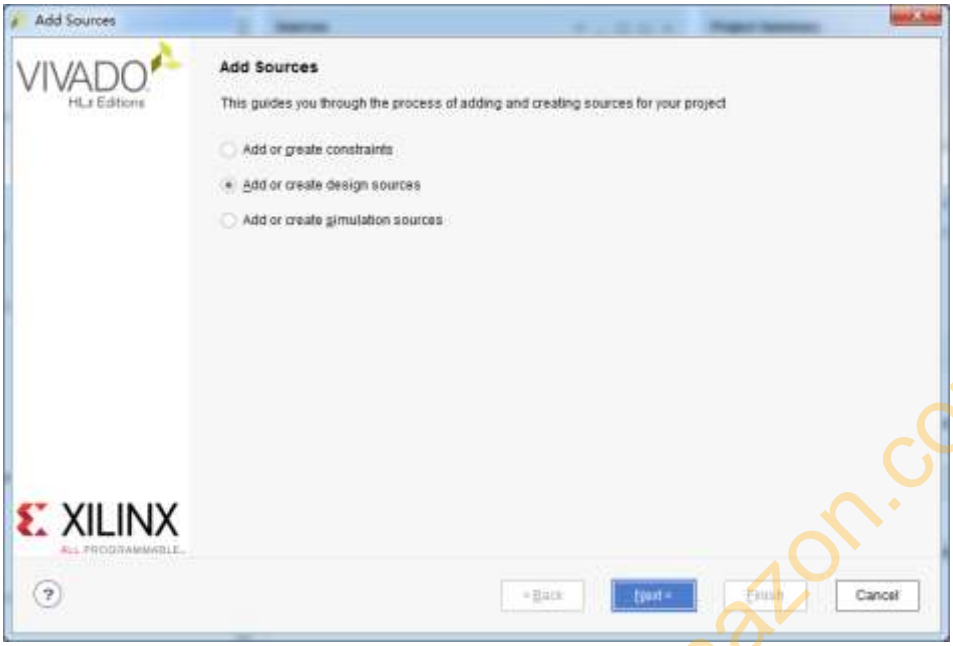


Part 4.3: Create a Verilog HDL file to illuminate the LED

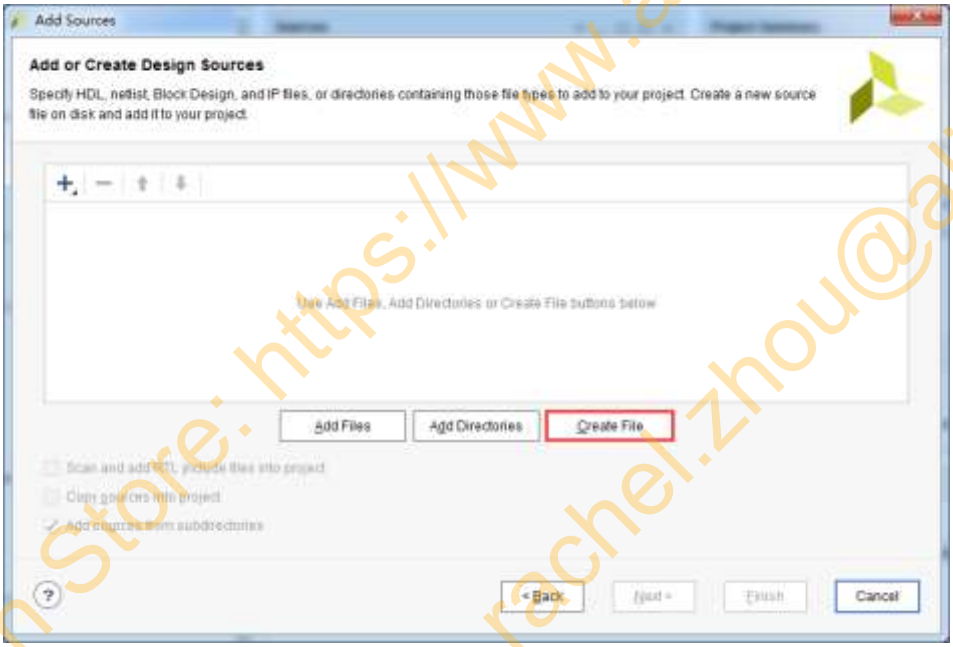
1) Click the Add Sources icon under Project Manager (or use the shortcut Alt+A)



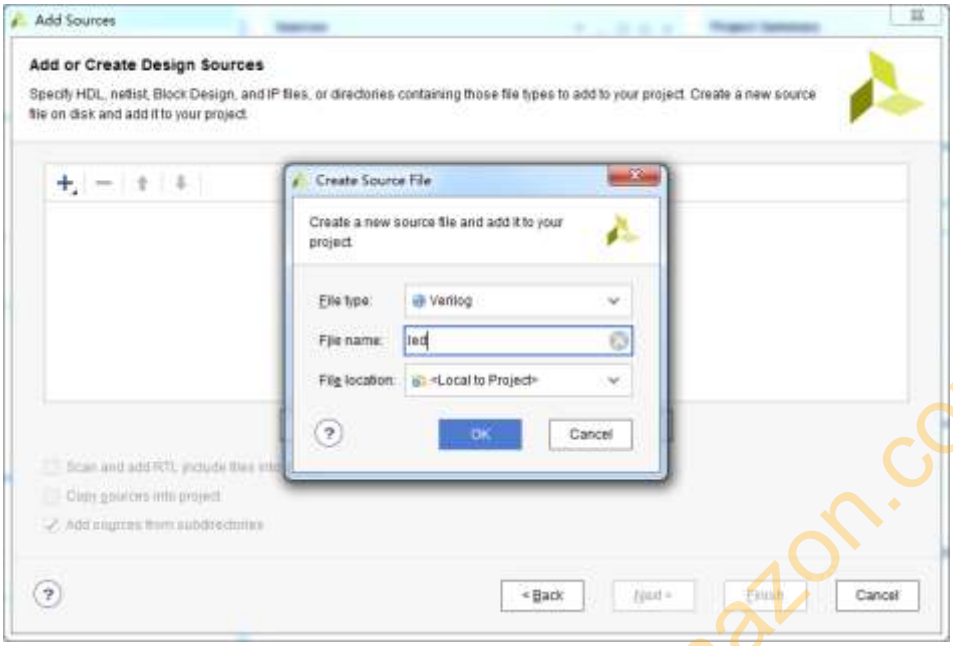
2) Add or create design source file "Add or create design sources", click "Next"



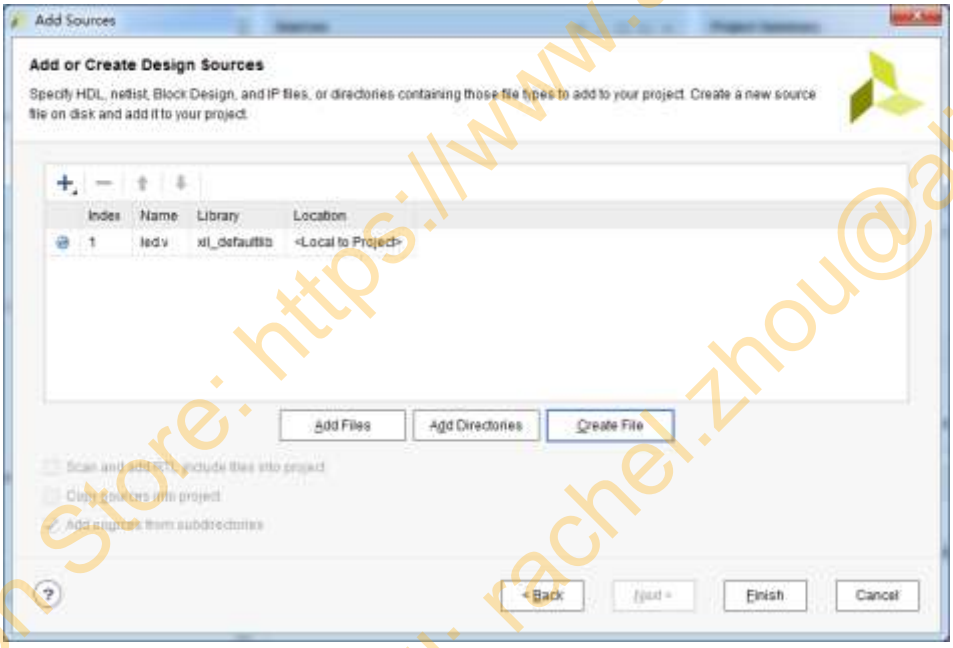
3) Choose to create the file "Create File"



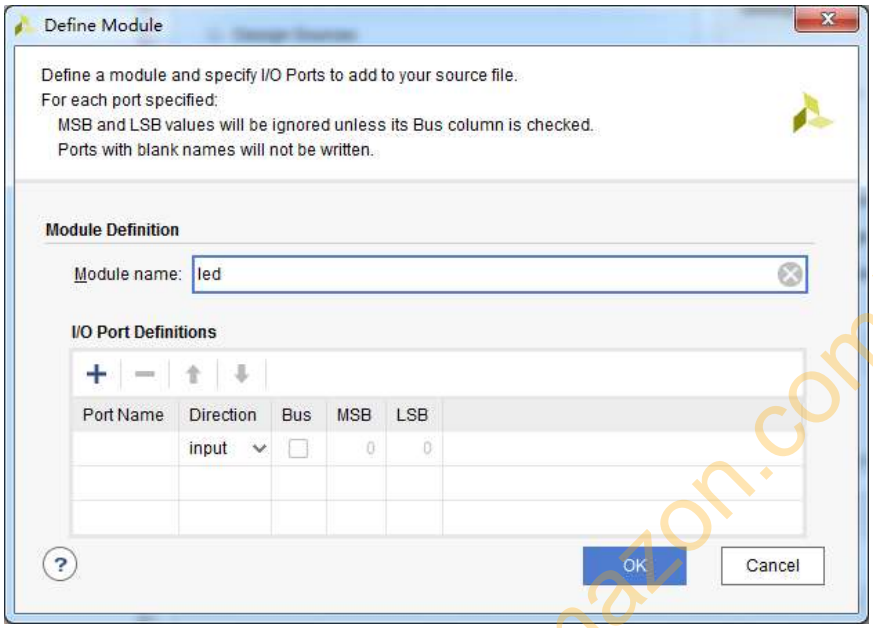
4) The file name "File name" is set to "led", click "OK"



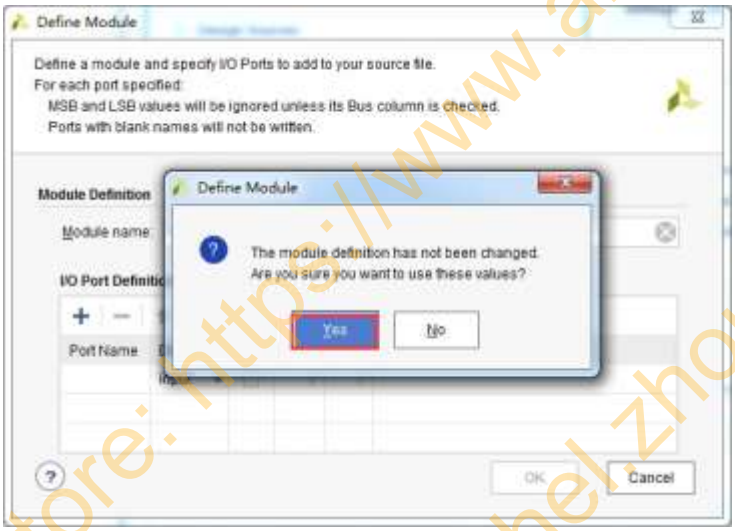
5) Click "Finish" to complete the "led.v" file addition



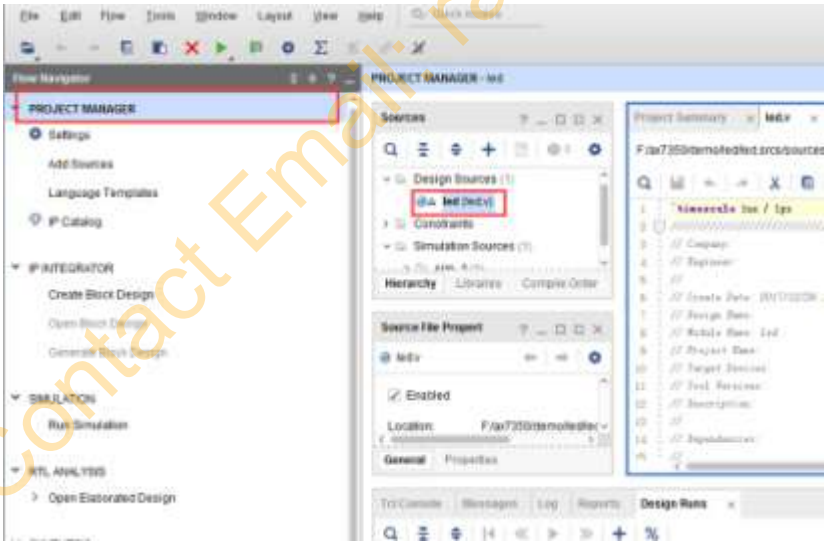
6) In the pop-up module definition "Define Module", you can specify the module name "Module name" of the "led.v" file, the default is not changed to "led", you can also specify some ports, not specified here, click "OK" ".



7) In the pop-up dialog box, select "Yes"



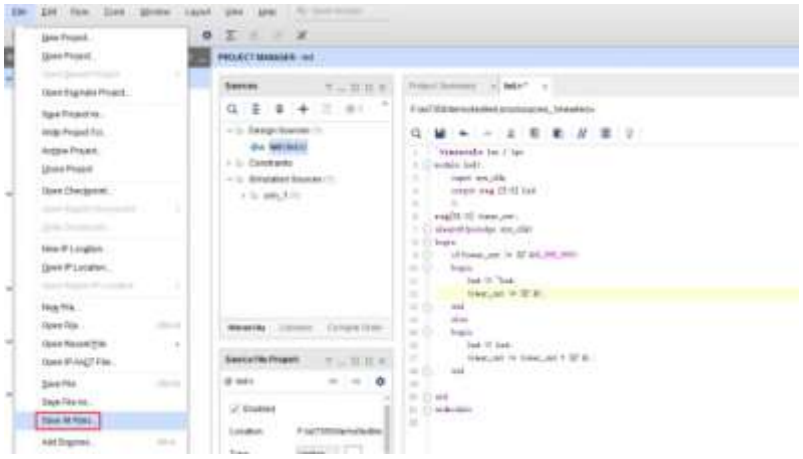
8) Double click on "led.v" to open the file and then edit



9) Write "led.v", here defines a 32-bit register timer for loop counting 0~49999999 (1 second), when counting to 49999999 (1 second), the register timer becomes 0, and flips four LED. If the original LED is off, it will light up. If the original LED is on, it will go out. The code after writing is as follows:

```
`timescale 1ns / 1ps
module led(
        input sys_clk,
        output reg [3:0] led
        );
reg[31:0] timer_cnt;
always@(posedge sys_clk)
begin
        if(timer_cnt >= 32'd49_999_999)
        begin
                led <= ~led;
                timer_cnt <= 32'd0;
        end
        else
        begin
                led <= led;
                timer_cnt <= timer_cnt + 32'd1;
        end

end
endmodule
```

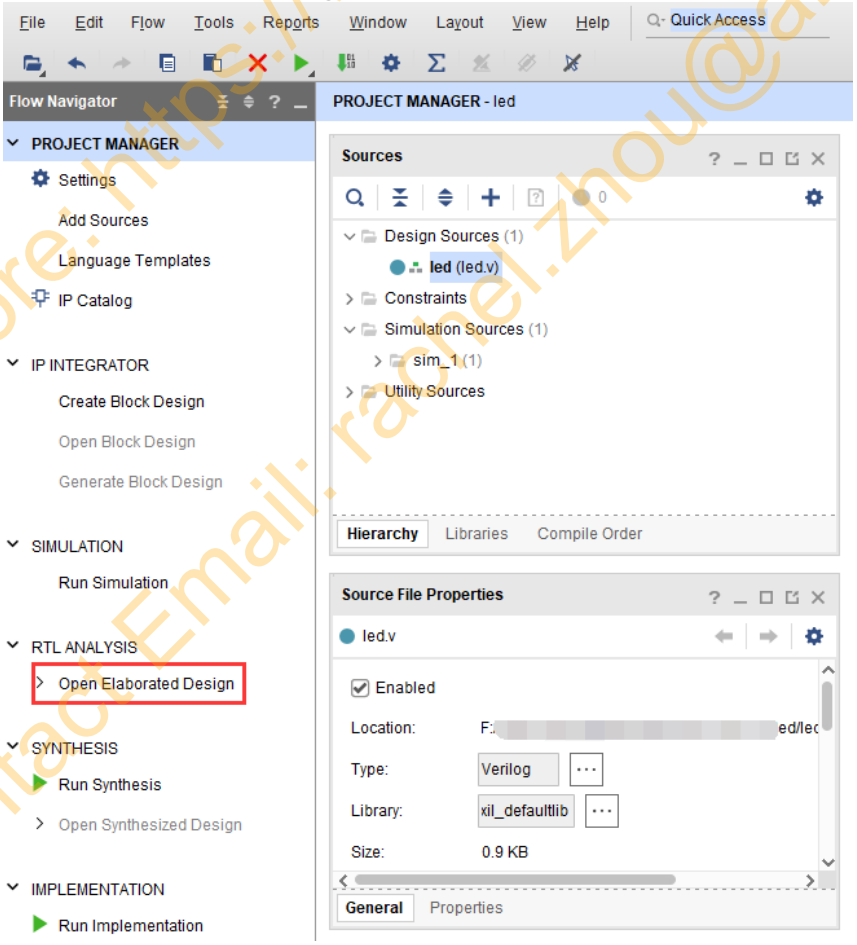
10) Write the code and save it, click on the menu "File -> Save All Files"



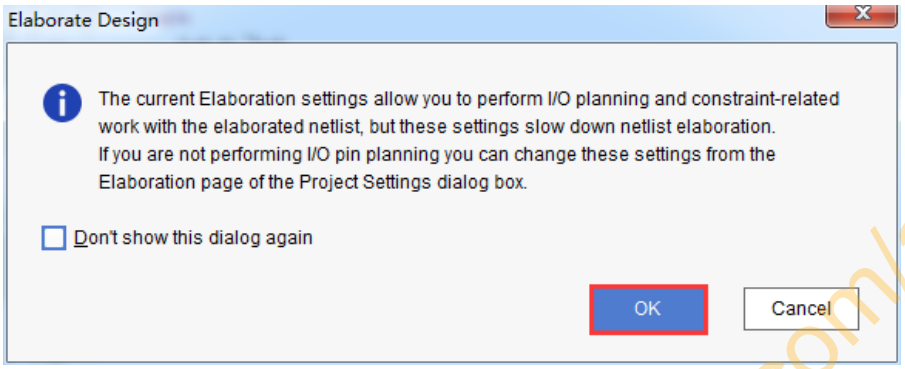
Part 4.4: Add XDC Pin Constraint

The constraint file format used by Vivado is an xdc file. The xdc file mainly consists of pin constraints, clock constraints, and group constraints. Here we need to assign the input and output ports in the led.v program to the real pins of the FPGA. This requires an FPGA pinout file, .xdc, to be added to the project.

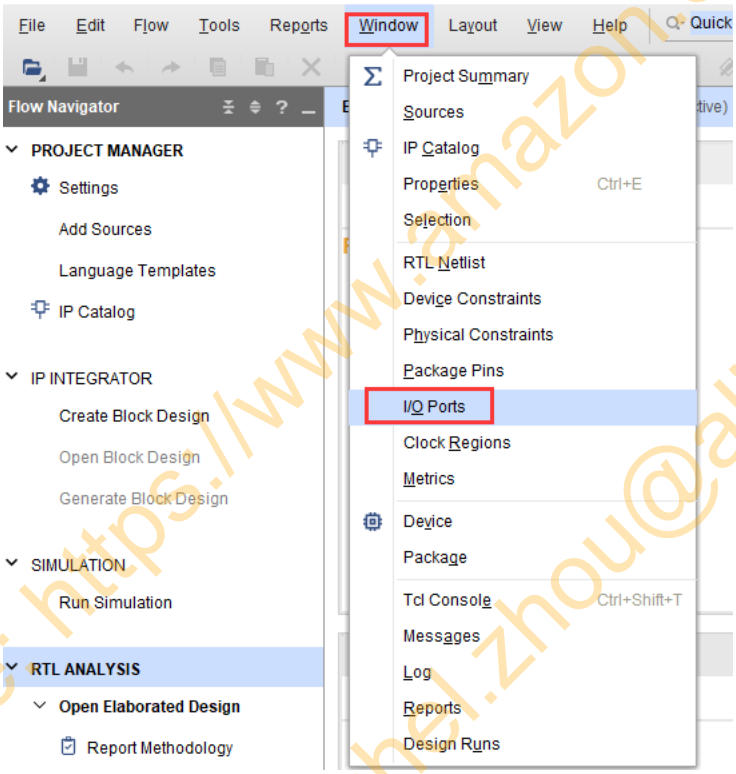
1) Click “Open Elaborated Design”



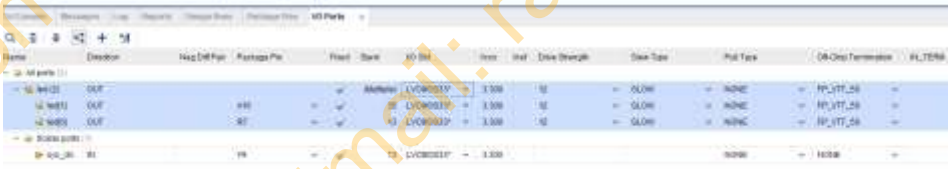
2) Click the "OK" button in the pop-up window



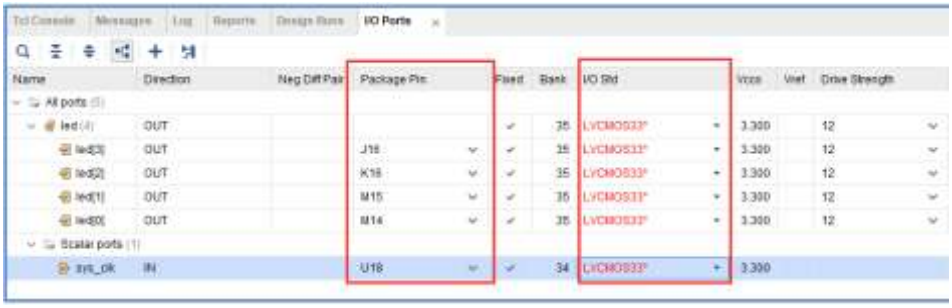
3) Select “Window -> I/O Ports” in the Menu



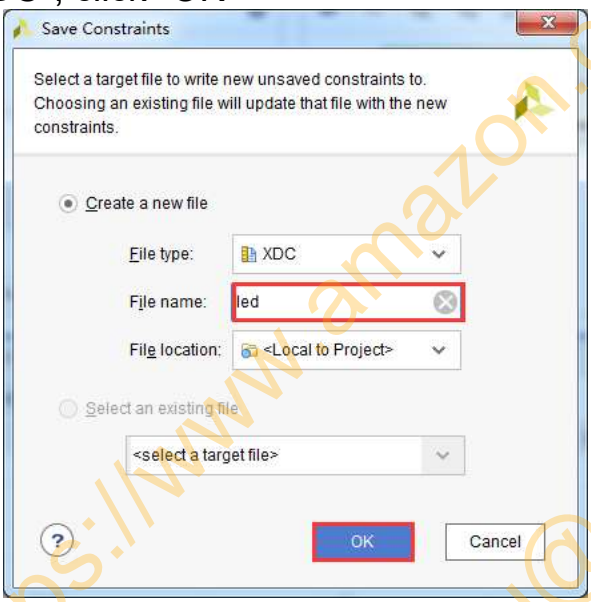
4) Pin assignments can be seen in the pop-up “I/O Ports”



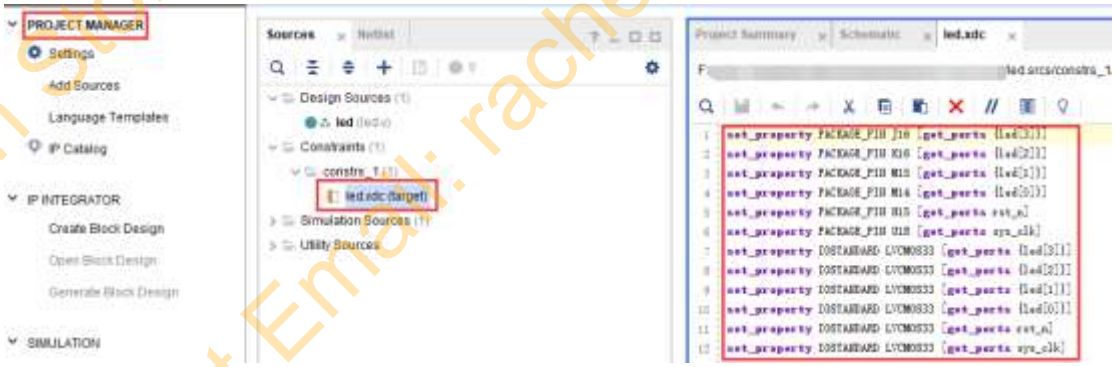
5) Assign pins and level standards to the LED and clock, click the save icon when finished



6) Pop-up window, ask to save the constraint file, file name we fill in "led", file type default "XDC", click "OK"



7) Open the "led.xdc" file just generated, we can see that it is a TCL script. If we understand these grammars, we can completely constrain the pins by writing the "led.xdc" file ourselves.



Let's introduce the most basic syntax written by XDC. The normal IO port only needs to constrain the pin number and voltage.

The pin constraints are as follows:

set_property PACKAGE_PIN "Pin Number" [get_ports "port name"]

The level signal constraints are as follows:

set_property IOSTANDARD "Level Standard" [get_ports"port name"]

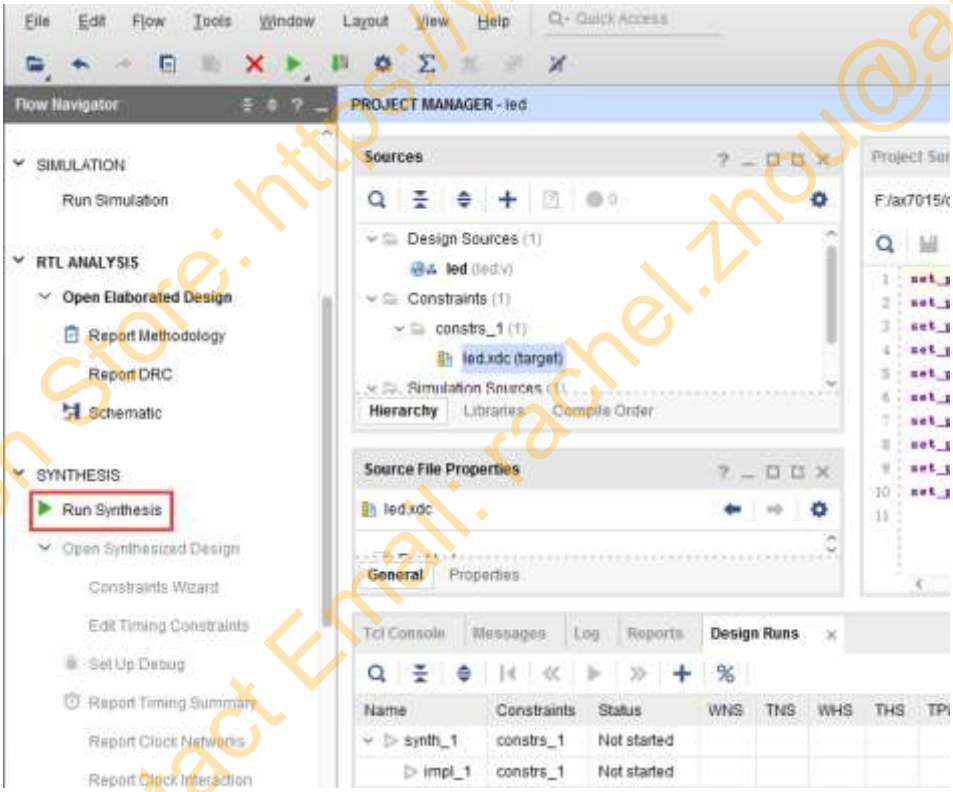
Pay attention to the case of the text. If the port name is an array, enclose it in { }. The port name must be the same as the name in the source code, and the port name cannot be the same as the keyword.

In the level standard, the number behind LVCMOS33 refers to the BANK voltage of the FPGA. The BANK voltage of the LED is 3.3 volts, so the level standard is "LVCMOS33". ***Vivado defaults to assigning the correct level standard and pin number to all IOs***

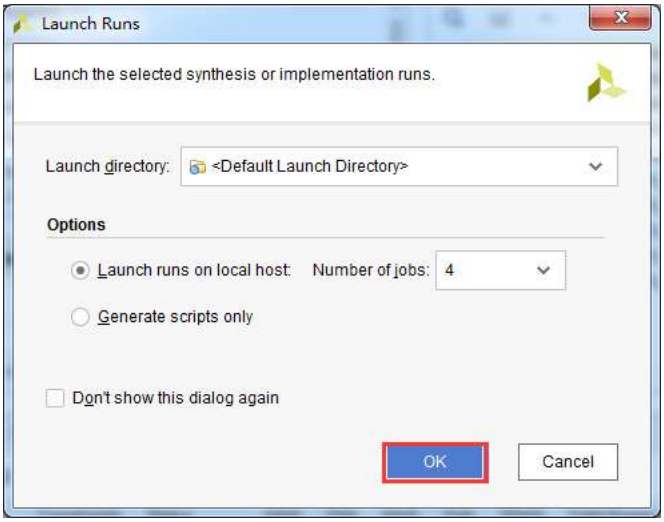
Part 4.5 Add timing constraints

In addition to the pin assignment, an FPGA design has timing constraints. Here, how to perform timing constraints is demonstrated through a wizard.

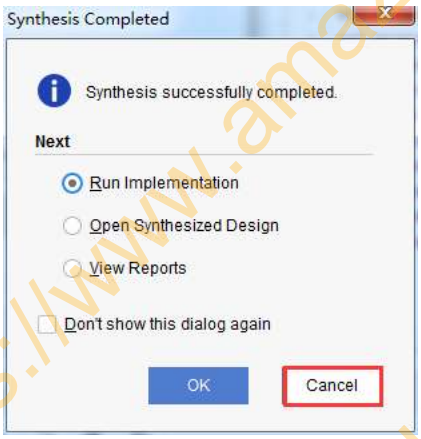
1) Click "Run Synthesis" to start to Synthesis



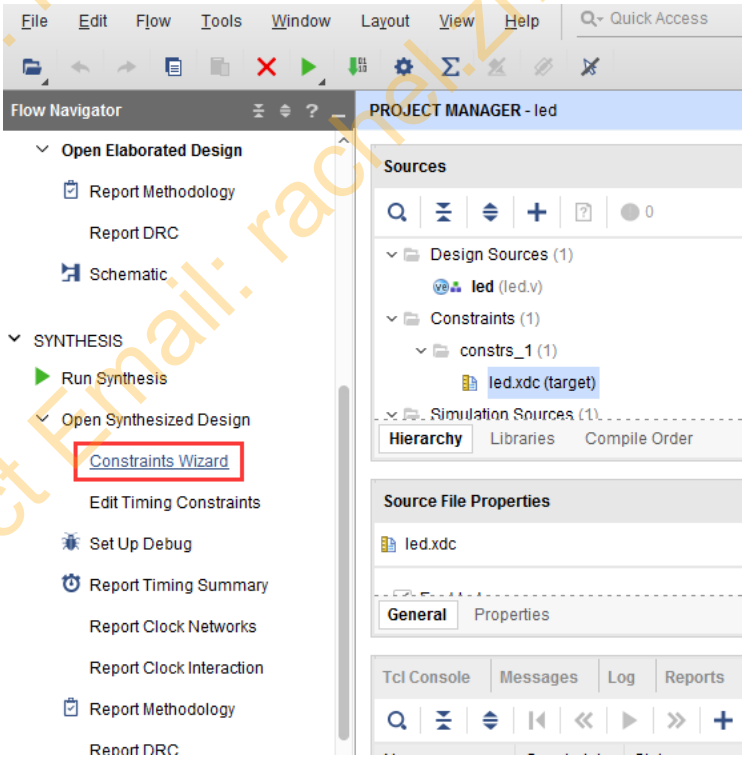
2) Pop up the dialog box and click "OK"



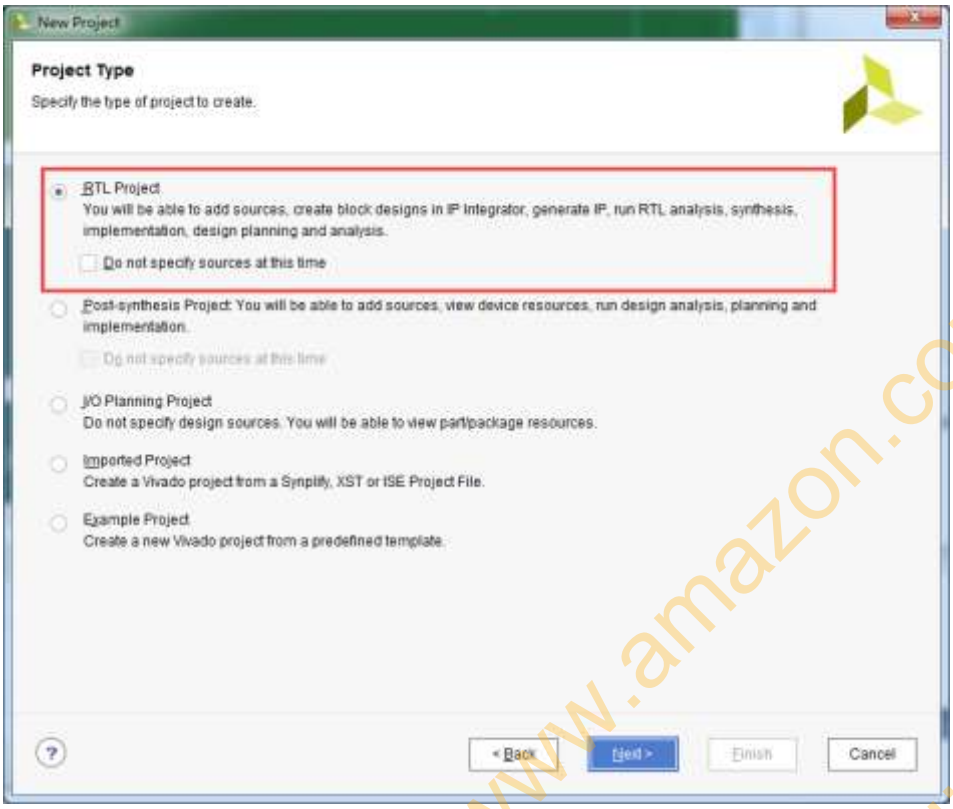
3) Click "Cancel" after the Synthesis successful Completed



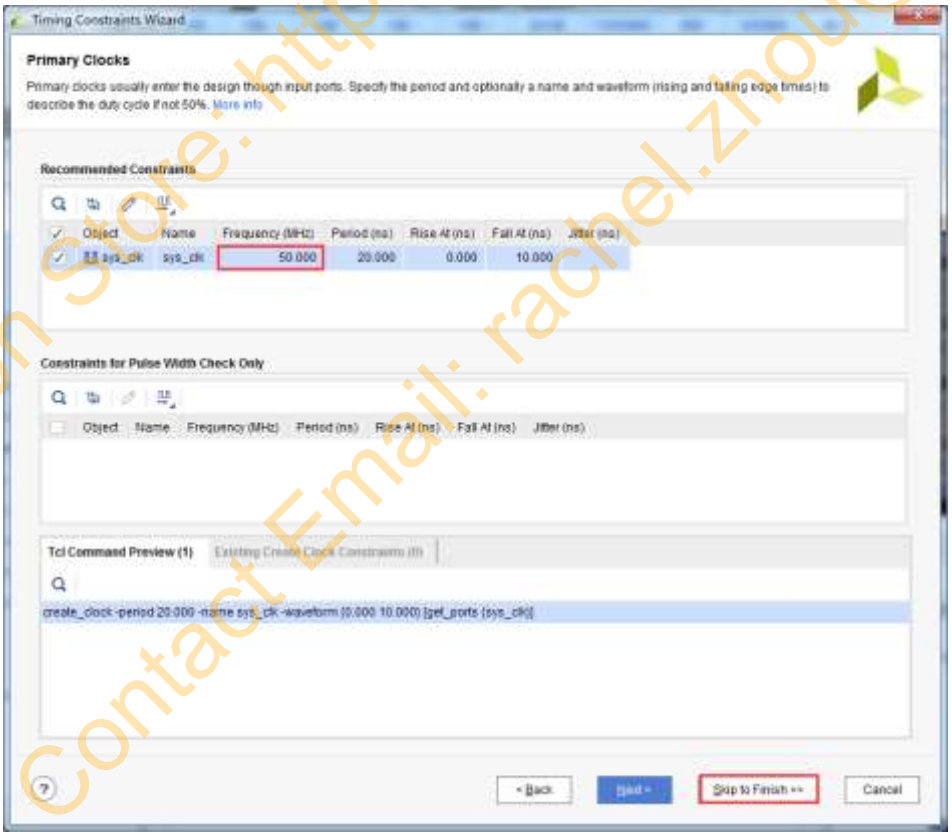
4) Click “Constraints Wizard”



5) Click "Next" in the pop-up window.

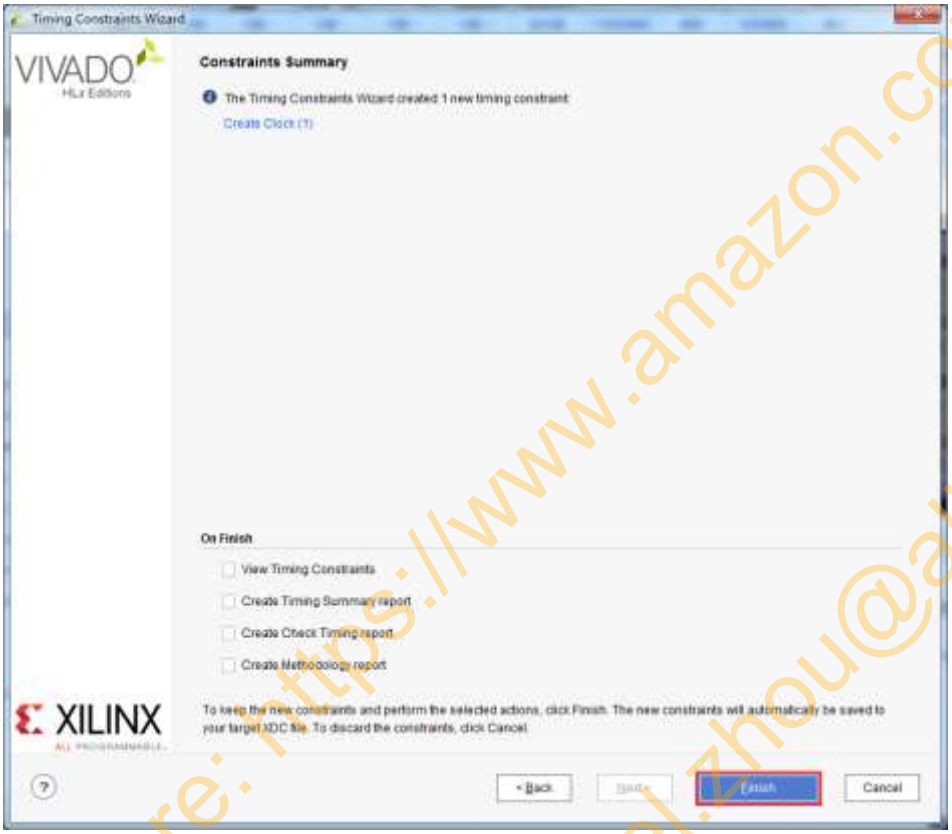


6) In the Timing Constrains Wizard, set the “sys_clk” frequency is “50Mhz”, then click “Skip to Finish"

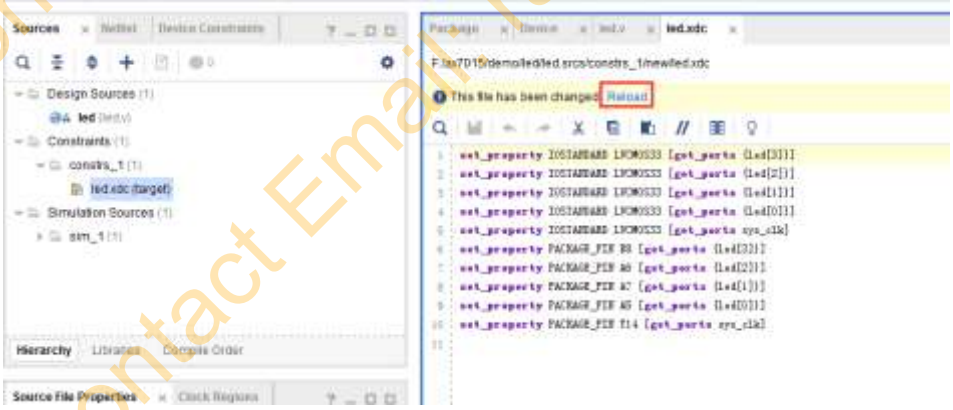


7) Click "OK" in the pop-up window.

8) Click “Finish”

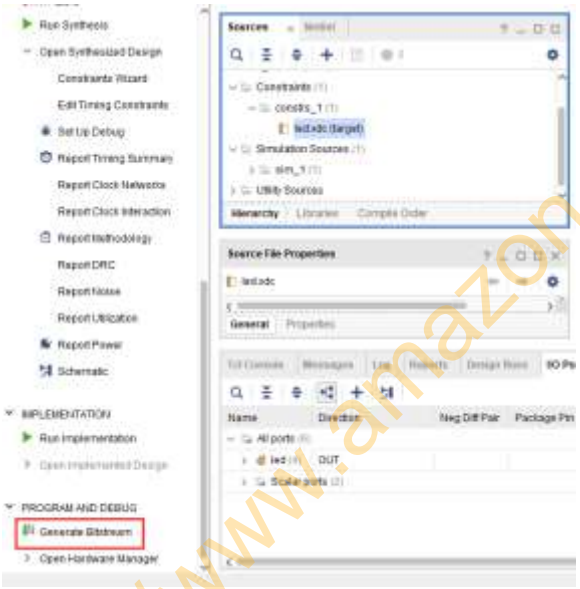


9) At this time, the “led.xdc” file has been updated. Click "Reload" to reload the file and save the file.

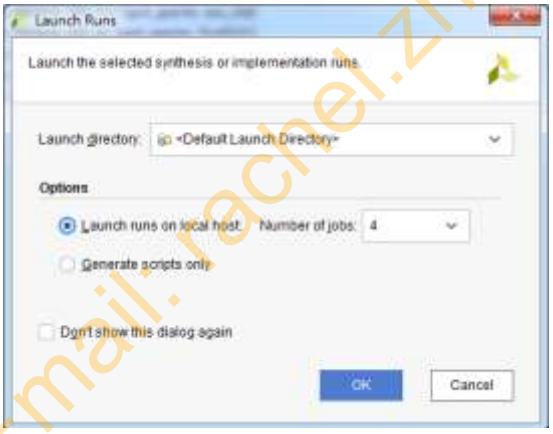


Part 4.6: Generate BIT File

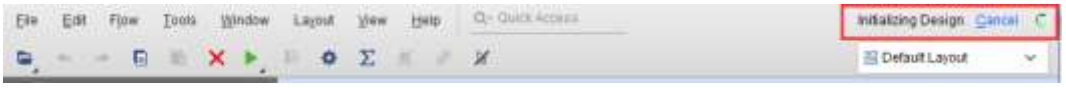
1) The compilation process can be subdivided into synthesis, place and route, generate bit files, etc. Here we directly click on "Generate Bitstream" to directly generate bit files.



2) In the pop-up dialog box, you can select the number of tasks, which is related to the number of CPU cores. The larger the general number, the faster the compilation. Click “OK”.



3) At this time, compile and start, you can see that there is a status information in the upper right corner. During the compilation process, it may be blocked by anti-virus software and computer housekeeper, resulting in failure to compile or not compiling successfully for a long time

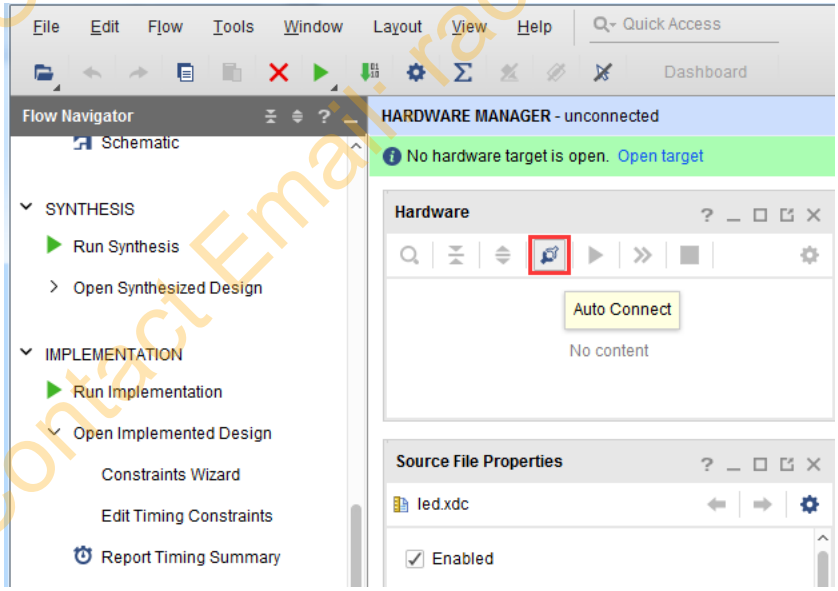


4) There is no error in the compilation, the compilation is completed, a dialog box pops up for us to select the subsequent operation, you can select "Open Hardware Manger", of course, you can also select "Cancel", we select "Cancel" here, do not download first

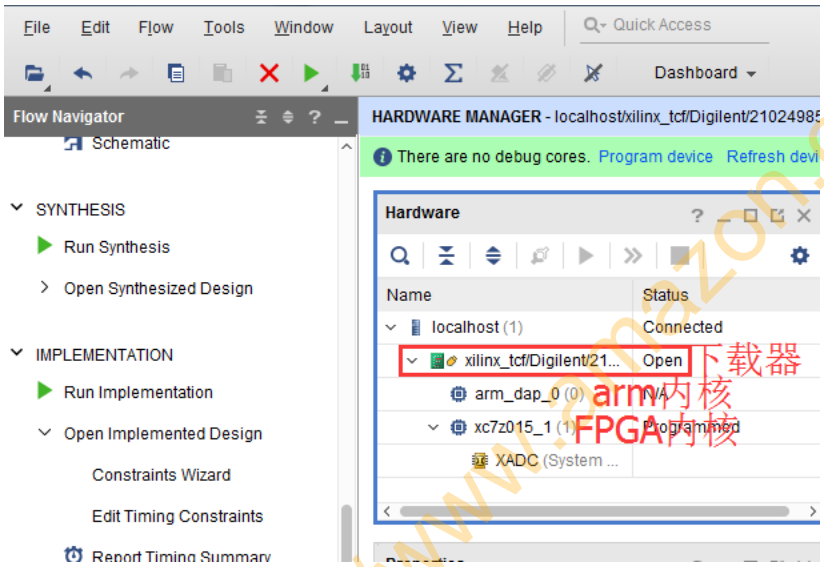
Part 4.7: Vivado Simulation

Next, use Vivado's own simulation tool to output the waveform.Verify that the flow light programming results are consistent with our expectations. Specific steps are as follows:

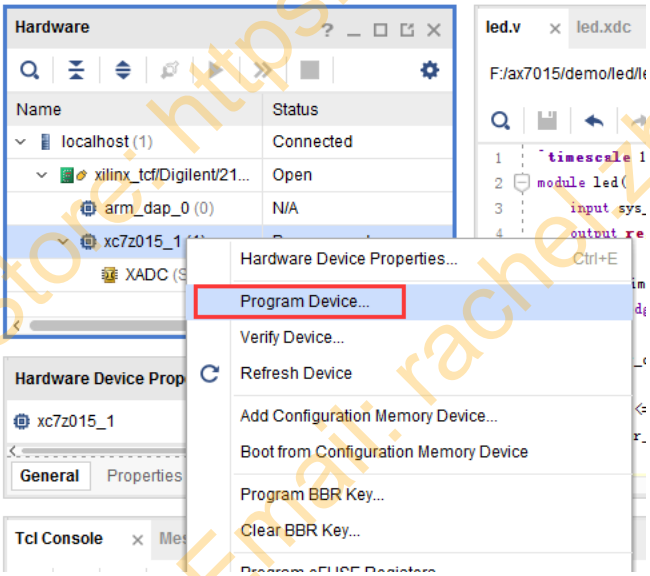
1) Connect the JTAG interface of the FPGA development board and power the development board
2) Click "Auto Connect" on the "Hardware Manager" interface to automatically connect to the device



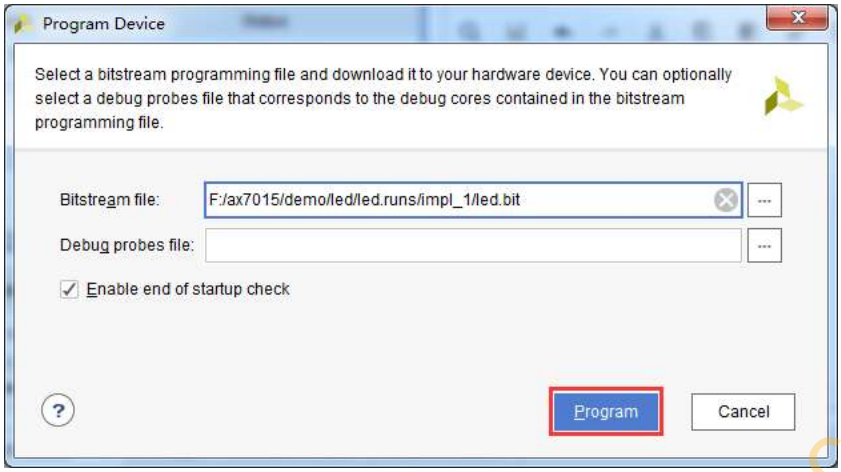
3) You can see that JTAG scans to the arm and FPGA core (the xc7z015 AX7010 FPGA development board in the picture is actually xc7z010_1, and the AX7020 FPGA development board is actually xc7z020_1), and there is also an XADC that can detect system voltage and temperature



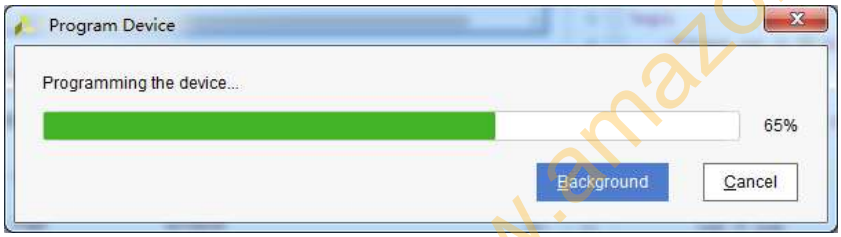
4) Select “xc7z0 1 0 _1” or “xc7z020_1”, right click “Program Device...”



5) Click "Program" in the pop-up window



6) Waiting to download



7) After the download is complete, we can see that 4 LEDs start to change every second. The Vivado simple process experience is now complete. The following chapters will introduce how to burn the program to Flash. It needs the cooperation of the PS system to complete. Only the PL project can not directly write Flash.

Part 5: HDMI output experiment

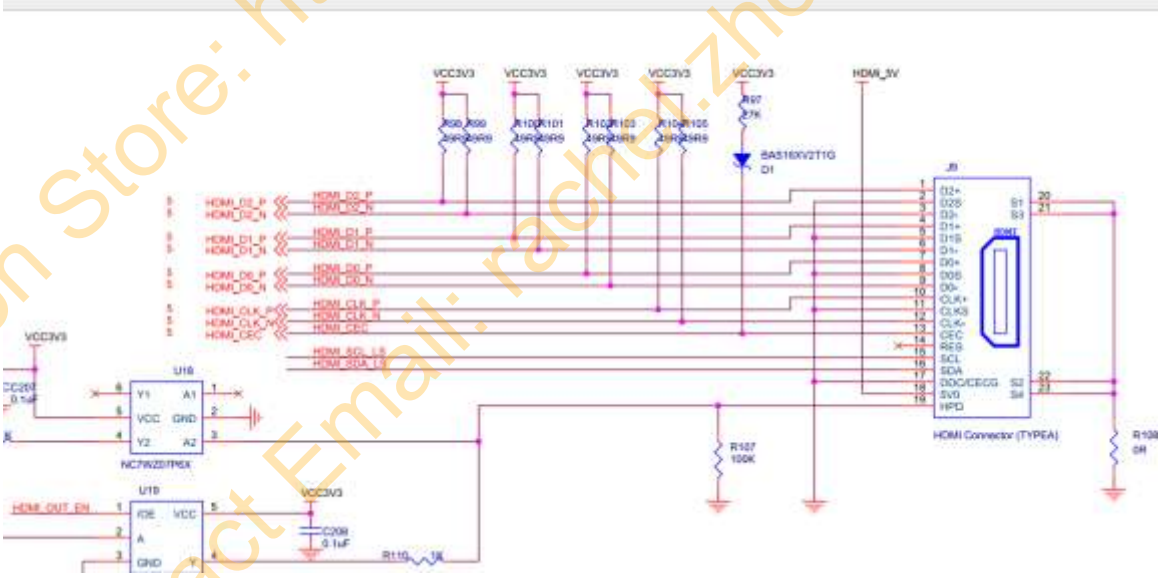
Experiment Vivado project is "hdmi_output_test"

The contents of this chapter are mainly implemented by FPGA engineers.

Earlier we introduced the led flashlight experiment, just to understand the basic development process of Vivado. This experiment is more complicated than the LED flash test. It is a color strip for HDMI output. This is the basis for learning display and video processing later. The experiment does not involve the PS system. It can be seen from the experimental design that if you want to use the ZYNQ chip very well, you need a good FPGA basis.

Part 5.1: Hardware introduction

Instead of using an HDMI encoding chip, the FPGA development board directly connects the 3.3V differential IO of the FPGA to the HDMI connector, and the FPGA completes the 24-bit RGB encoded output TMDS differential signal.



Before we do HDMI display, we need to think about a problem. What are the elements required to generate HDMI image data? If you know that the timing of the image should be clear, it is nothing more than the pixel

clock, image timing, and image data of the image. Of course, different resolution clocks and timings are also different. Specific online search for timing parameters of various resolutionsI will not repeat them here. This experiment only provides 720p timing parameters (in the color_bar.v file).

```
/*********视频时序参数定义**************************************/
parameter H_ACTIVE = 16'd1280;  //行有效长度（像素时钟周期个数）
parameter H_FP = 16'd110;       //行同步前肩长度
parameter H_SYNC = 16'd40;      //行同步长度
parameter H_BP = 16'd220;       //行同步后肩长度
parameter V_ACTIVE = 16'd720;   //场有效长度（行的个数）
parameter V_FP     = 16'd5;       //场同步前肩长度
parameter V_SYNC   = 16'd5;     //场同步长度
parameter V_BP = 16'd20;       //场同步后肩长度
```

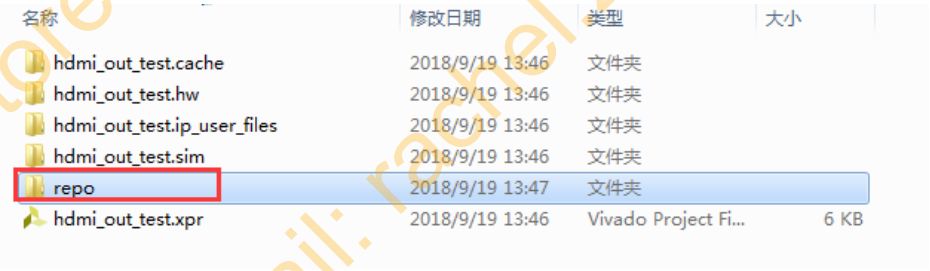
Part 5.2: Create a Vivado Project

1) Create a new project called "hdmi_output_test"

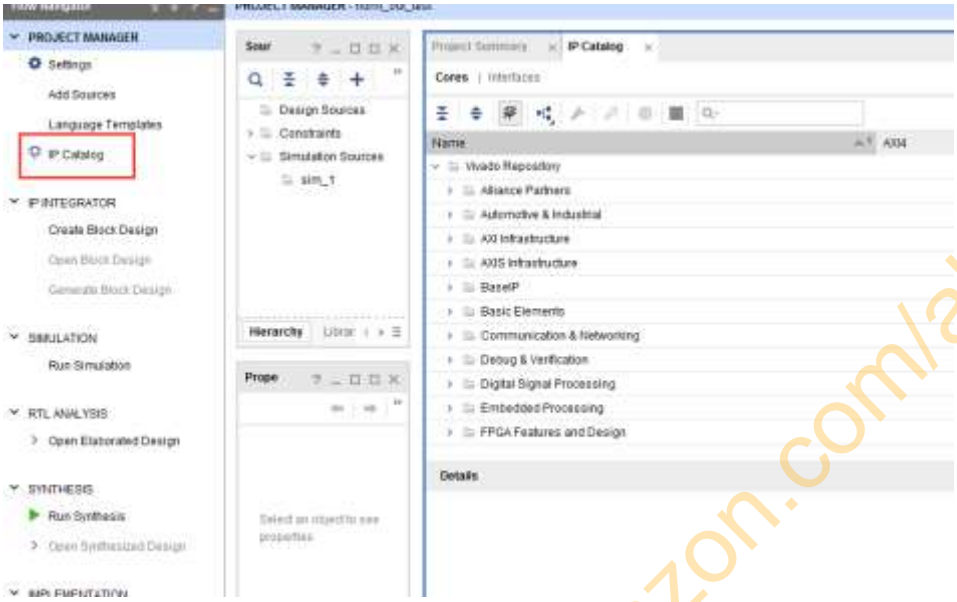
Part 5.2.1: Add HDMI Encoder IP Core

VGA data is clear to many people, for RGB data, and HDMI is TMDS differential signal, RGB data is relatively easy to operate in FPGA, then what we need to do is to convert RGB data into HDMI TMDS differential signal, so use the IP of RGB to DVI(DVI and HDMI are both TMDS signals).

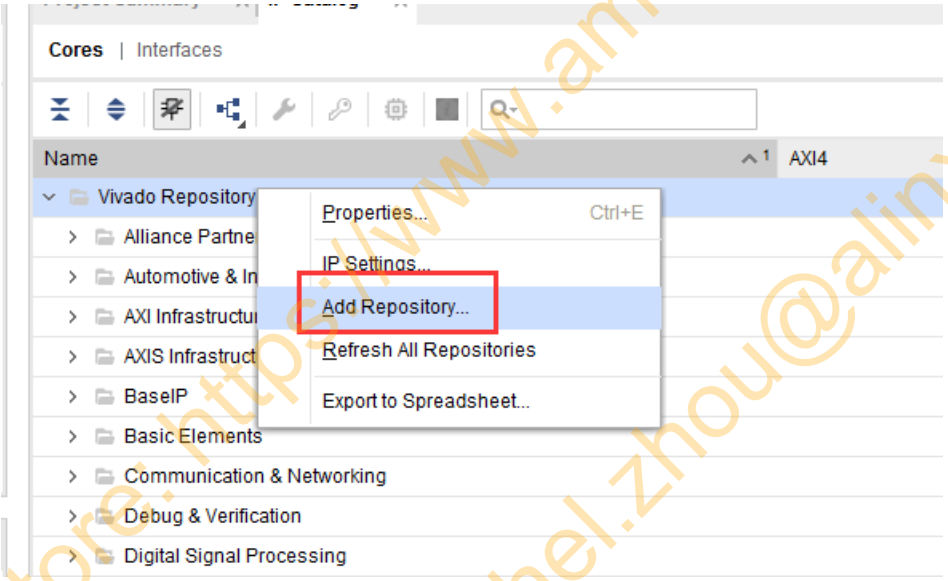
2) Copy the repo folder (this folder can be found in the given routine project) to the project directory, which contains the IP of the HDMI encoder, which is provided by other manufacturers



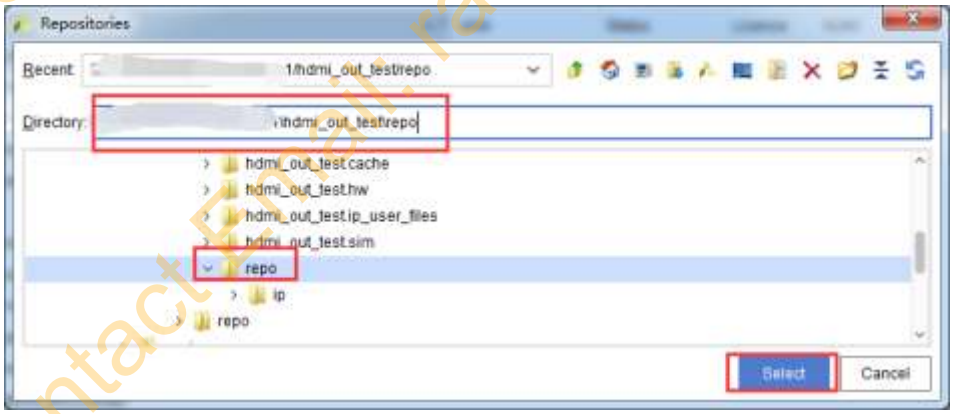
3) Click on "IP Catalog", the default IP and both are provided by Xilinx, now we have to add third-party IP, or our own IP



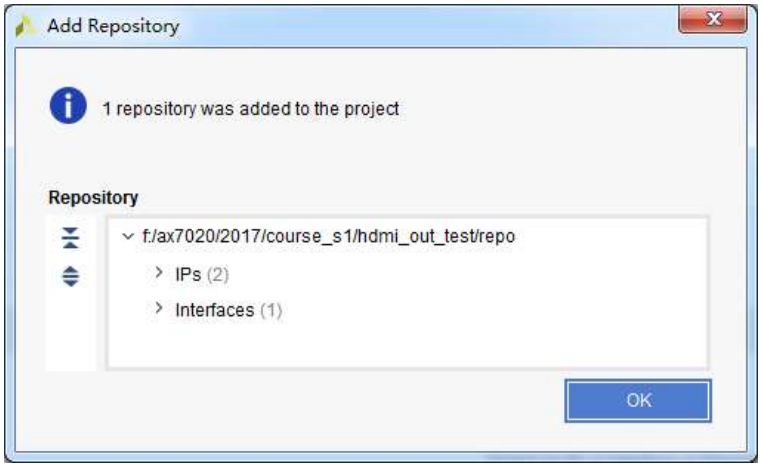
4) Right click "Add Repository"...



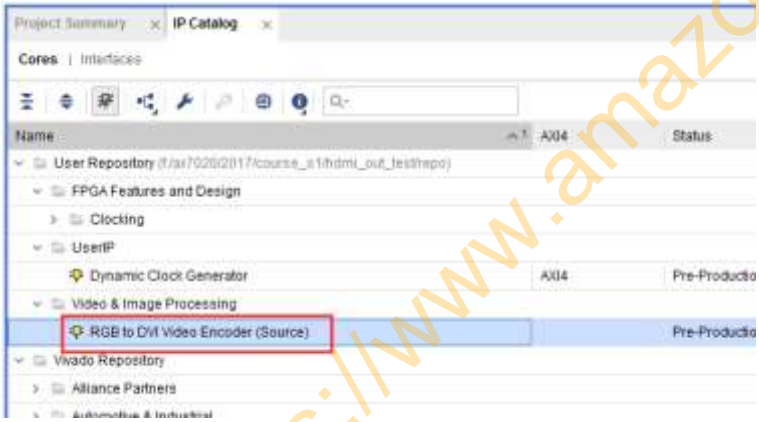
5) Path selects the “repo” folder just copied



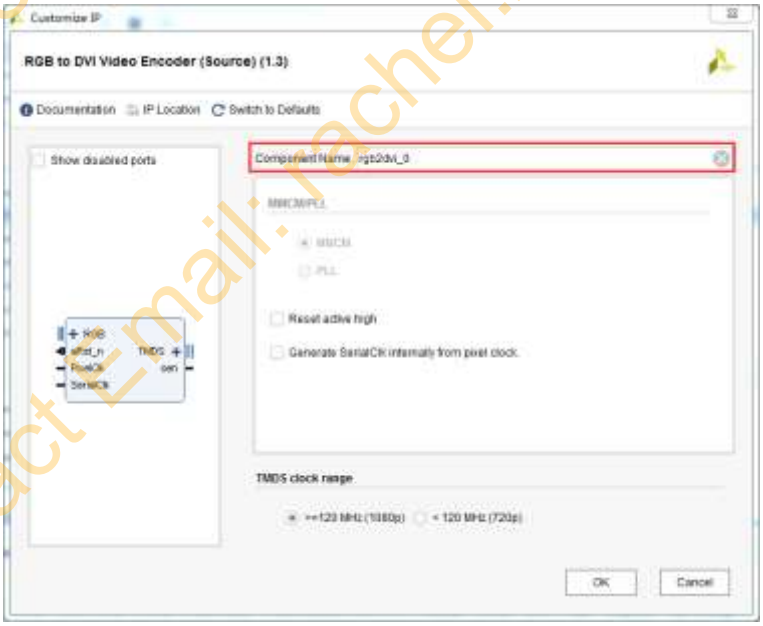
6) Add IP successfully, prompting the number of IP added



7) Find "RGB to DVI Video Encoder(Source)" and double click

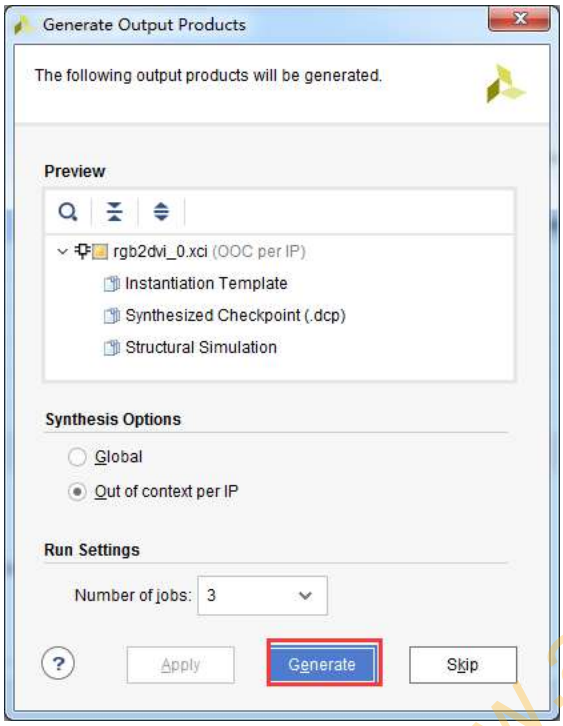


8) The following window pops up, the component name of "Component Name" remains unchanged, and other parameters are not changed. Click "OK"

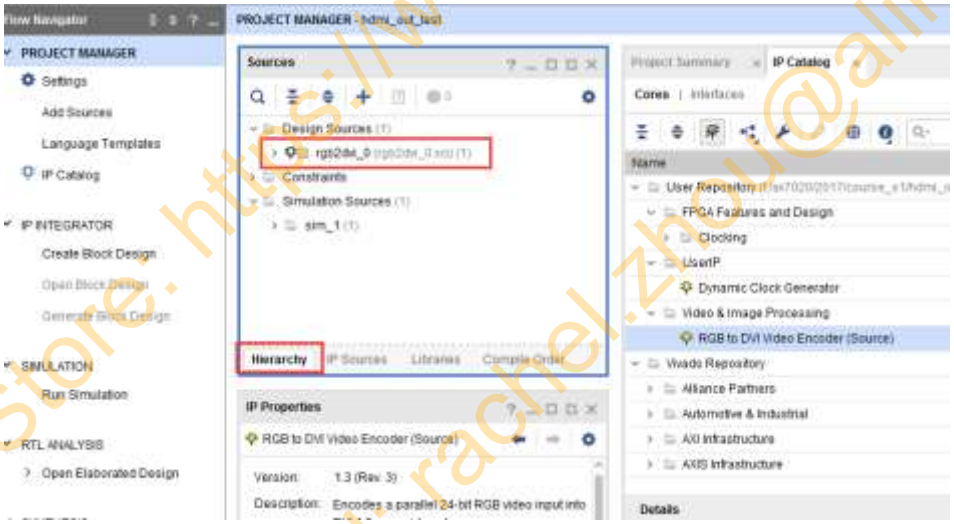


9) The “Generate Output Products” window pops up, where “Number of

jobs" refers to the number of threads, the higher the faster



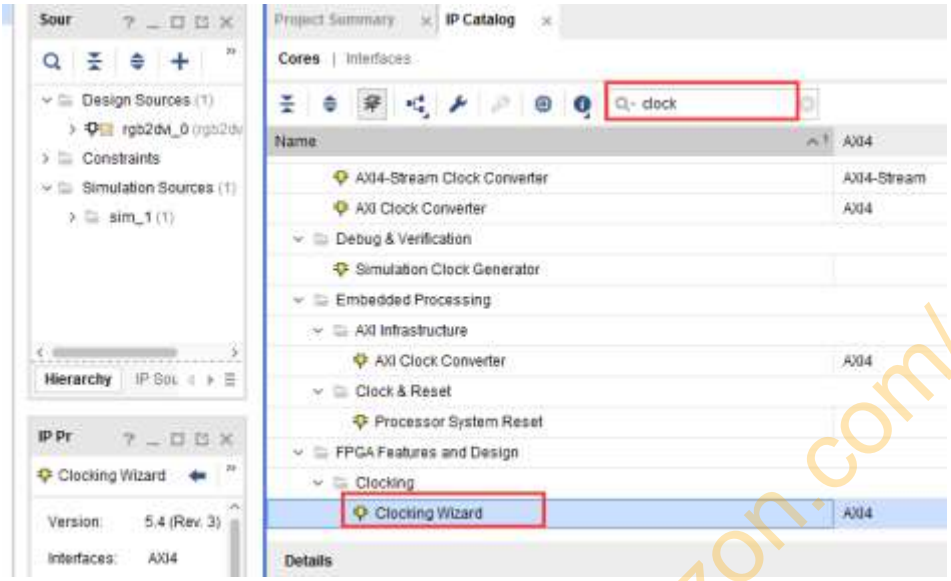
10) You can see a name called "rgb2dvi_0"



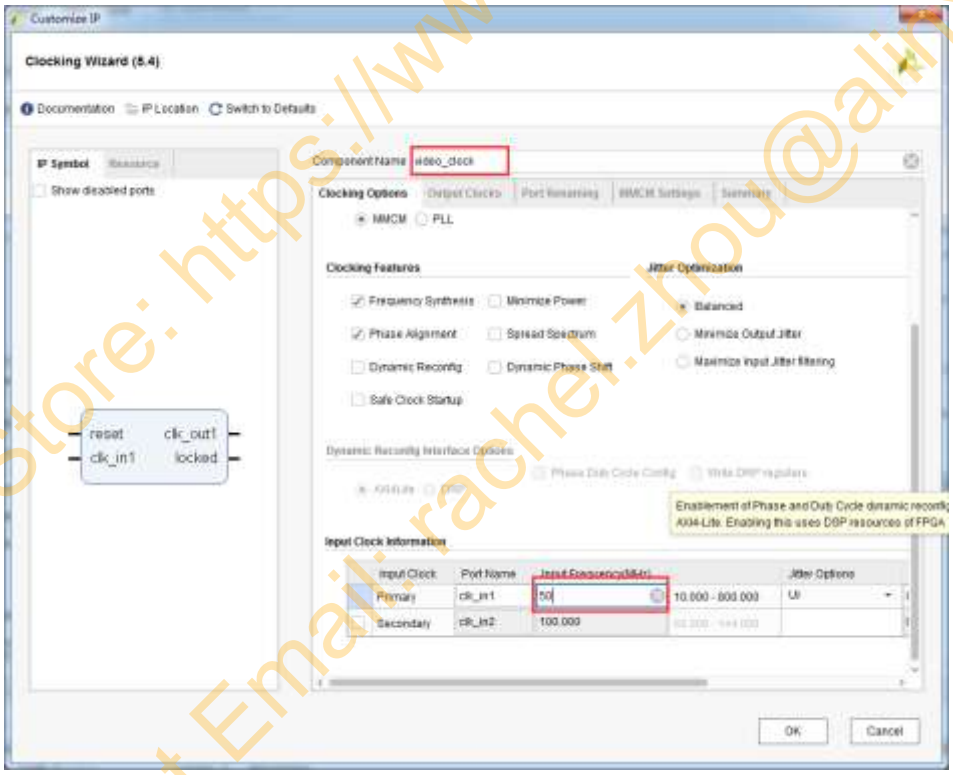
Part 5.2.2: Add pixel clock PLL module

In order to drive the HDMI encoder, a pixel clock and a 5x pixel clock are required, and a 5x pixel clock is used for 10:1 serialization.

11) Search for the keyword "clock" in the "IP Catlog" window and double-click "Clocking Wizard"

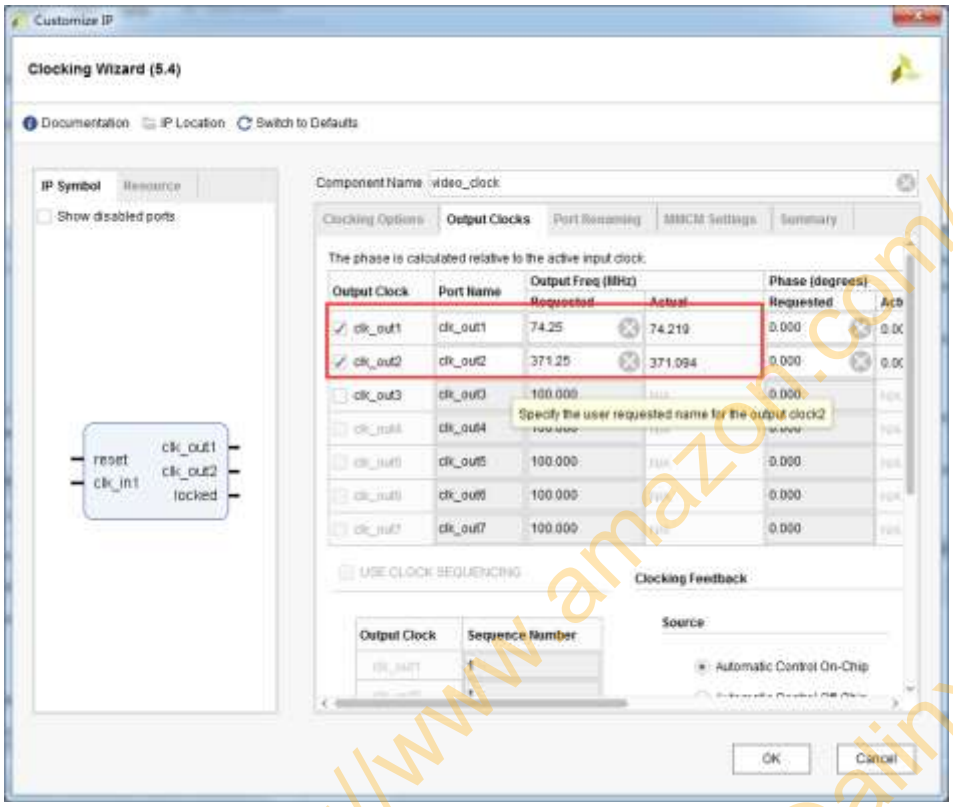


12) Name the component, fill in "video_clock" in "Component Name", and fill in 50 in "clk_in1". Here, 50Mhz and the PL crystal of the FPGA development board have the same frequency.



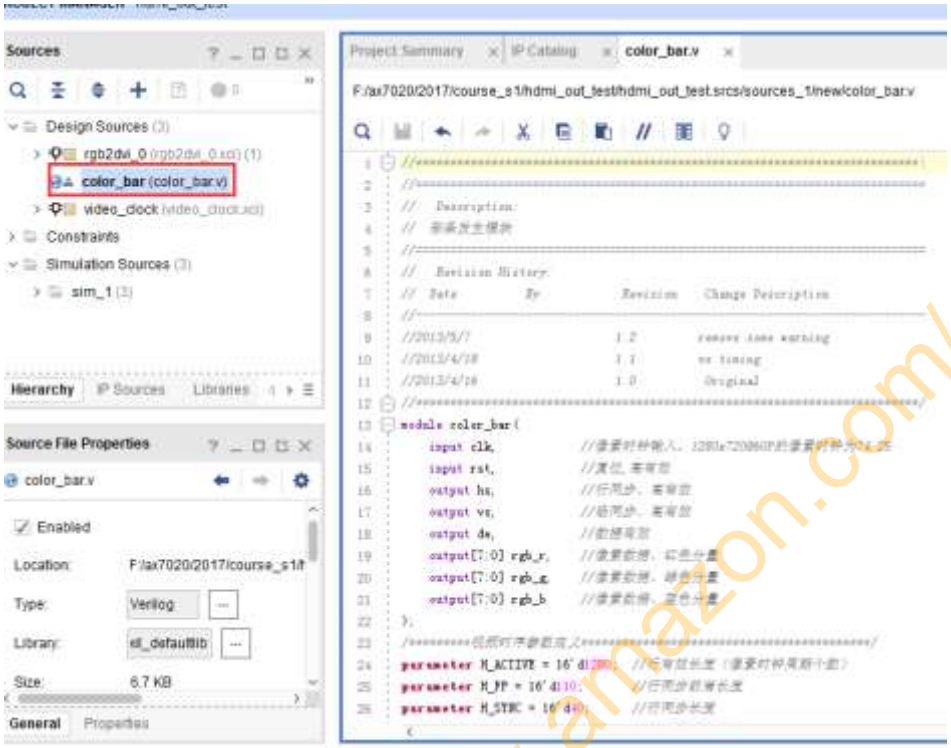
13) The output clock "clk_out1" is used for the video pixel clock. Here, fill in 74.25, which is a pixel clock of 1280x720@60 resolution. The pixel clock of each resolution is different. You need to know the video standard very well to know the resolution of each video. The pixel

clock, "clk_out2" is used for encoder serialization, 5 times the pixel clock, fill in 371.25 here, then click "OK" to generate the IP.



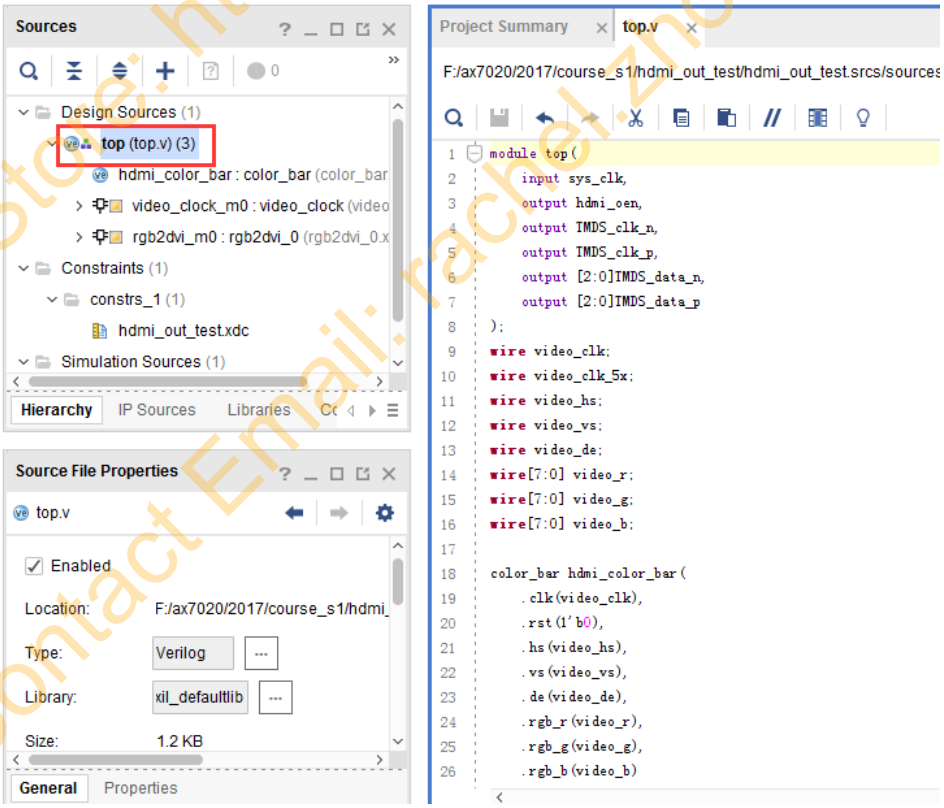
Part 5.2.3: Add a color bar generation module

14) The color bar generation module is a piece of Verilog code, which is used to generate 8 color bars in video timing and horizontal direction. FPGA is not the focus of this development. No more detailed explanation of the code, you can copy the existing code to the given routine.



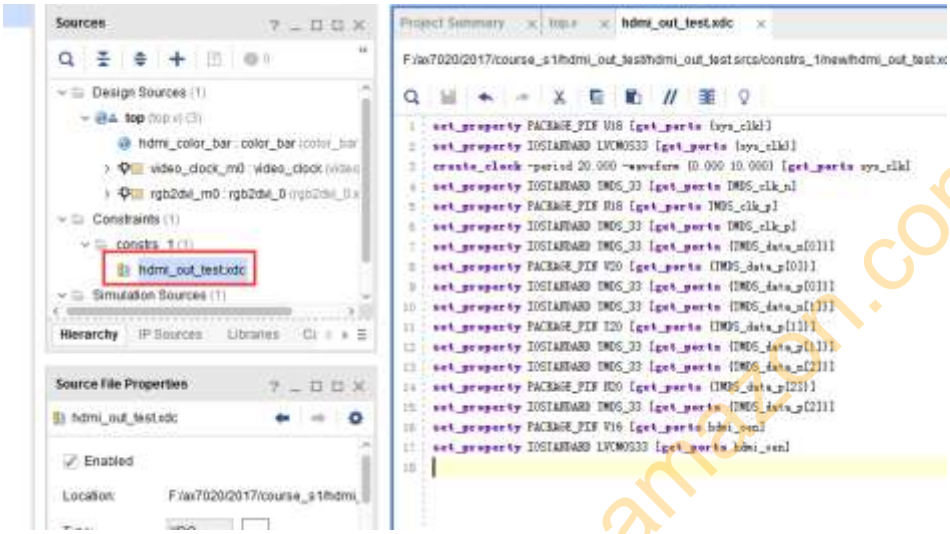
Part 5.2.4: Add top level module

15) The top module instantiates the color bar generation module, the HDMI encoding module, and the pixel clock generation module, and the code reference routine gives the project.



Part 5.3: Add an XDC constraint file

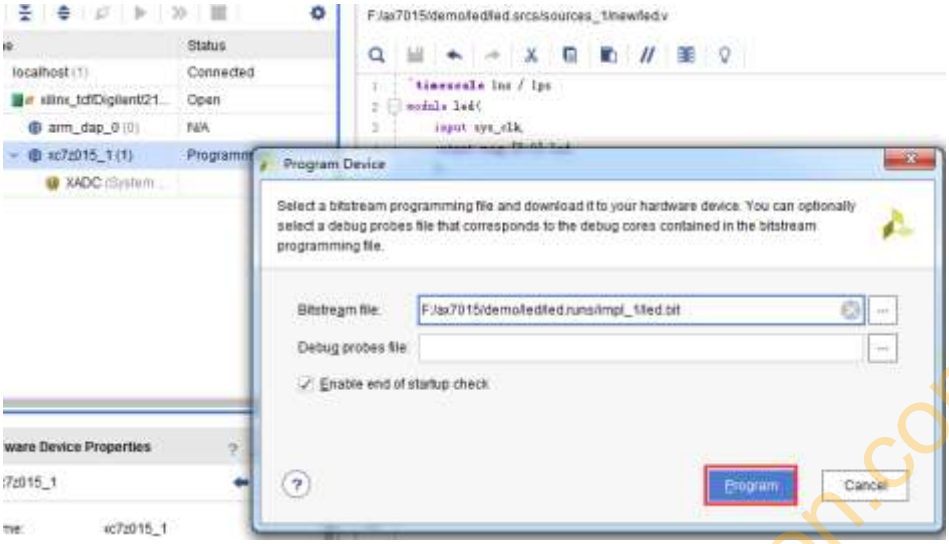
Add the following xdc constraint file to the project and add the clock and HDMI related pins to the constraint file.




```
set_property PACKAGE_PIN U18 [get_ports {sys_clk}]
set_property IOSTANDARD LVCMOS33 [get_ports {sys_clk}]
create_clock -period 20.000 -waveform {0.000 10.000} [get_ports sys_clk]
set_property IOSTANDARD TMDS_33 [get_ports TMDS_clk_n]
set_property PACKAGE_PIN N18 [get_ports TMDS_clk_p]
set_property IOSTANDARD TMDS_33 [get_ports TMDS_clk_p]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_n[0]}]
set_property PACKAGE_PIN V20 [get_ports {TMDS_data_p[0]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_p[0]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_n[1]}]
set_property PACKAGE_PIN T20 [get_ports {TMDS_data_p[1]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_p[1]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_n[2]}]
set_property PACKAGE_PIN N20 [get_ports {TMDS_data_p[2]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_p[2]}]
set_property PACKAGE_PIN V16 [get_ports hdmi_oen]
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_oen]
```

Part 5.4: Download debugging

Save the project and compile and generate the bit file, connect the HDMI interface to the HDMI display. Note that here 1290 x 720@60Hz, please make sure your monitor supports this resolution.



After downloading, the display shows the following image

Part 5.5: Experimental summary

This experiment is initially exposed to video display, which involves video knowledge. This is not the focus of zynq learning, so it is not detailed, but zynq is widely used in video processing and requires good basic knowledge for learners. In the experiment, only the PL is used to drive the HDMI chip, and the usage of the third-party custom IP is initially studied. Later we will learn how to customize the IP.

Part 6: Experience ARM, bare metal output "Hello World"

Experiment Vivado project is "ps_hello"

From the beginning of this chapter, FPGA engineers and software development engineers work together.

The previous experiments were carried out on the PL side. It can be seen that there is no difference between the ordinary FPGA development precess. The main advantage of ZYNQ is the reasonable combination of FPGA and ARM, which puts higher requirements 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 the features of the Vivado SDK and PS.

The previous experiments are all things that FPGA engineers should do. From the beginning of this chapter, there is a division of labor. The FPGA engineers are responsible for building the Vivado project and providing hardware to the software developers. The software developers can develop applications on this basis. .A good division of labor is conducive to the promotion of the project. If the software developer wants to do everything, it may take a lot of time and effort to learn the FPGA knowledge.

Part 6.1: Hardware introduction

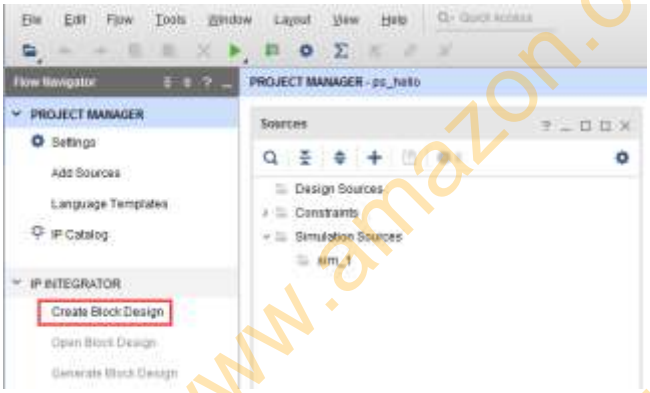
We can see from the schematic that the ZYNQ chip is divided into PL and PS. The IO allocation on the PS side is relatively fixed and cannot be arbitrarily allocated. It is not necessary to allocate pins in the Vivado software. Although this experiment only uses PS, it also to build a Vivado project, configure the PS pins. Although the ARM on the PS side is a hard core, in the ZYNQ, the ARM hard core must also be added to the project to be used. The previous chapter introduces the project in code form. This chapter begins to introduce the graphical creation method of ZYNQ.

FPGA engineer work content

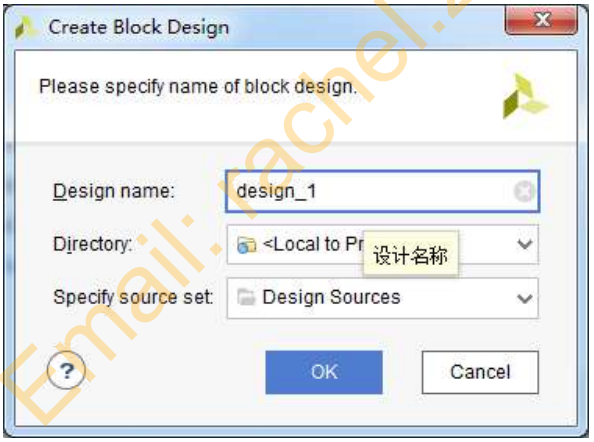
The following is an introduction to the contents of the FPGA engineer.

Part 6.2: Create a Vivado Project

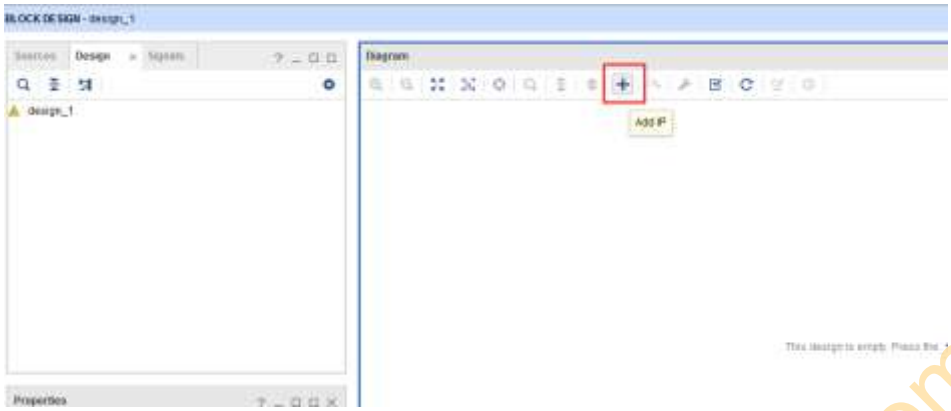
1) Create a project called "ps_hello". The process will not be repeated, refer to "PL" "Hello World" LED experiment"
2) Click on "Create Block Design" to create a block design, which is a graphical design.



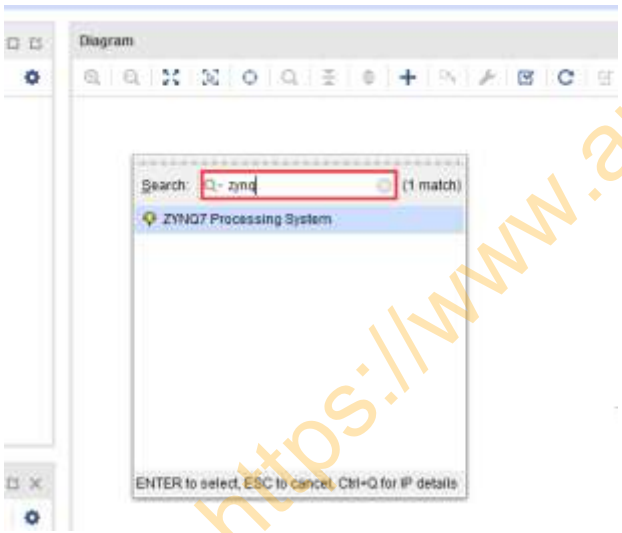
3) "Design name" is not modified here, keep the default "design_1", which can be modified as needed, but the name should be as short as possible, otherwise there will be problems compiling under Windows.



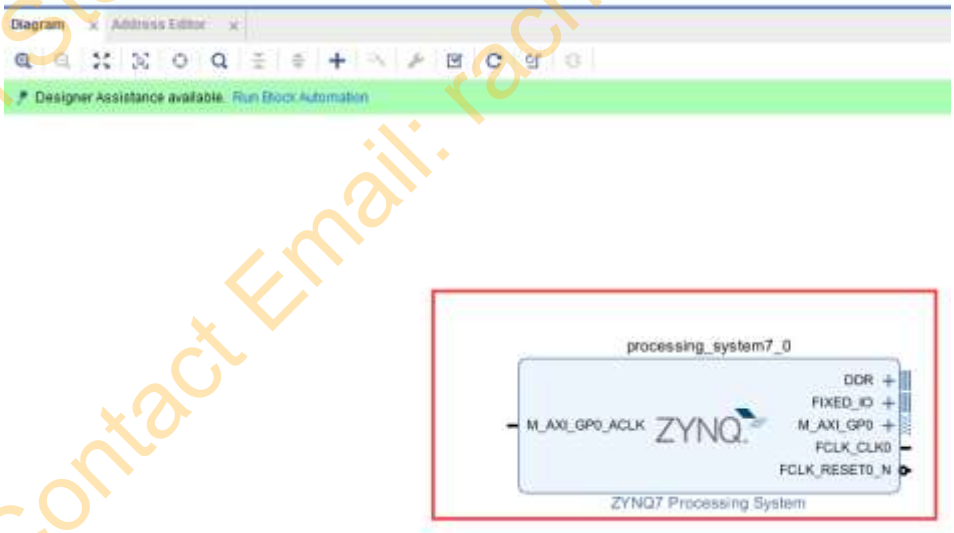
4) Click on the "Add IP" shortcut icon



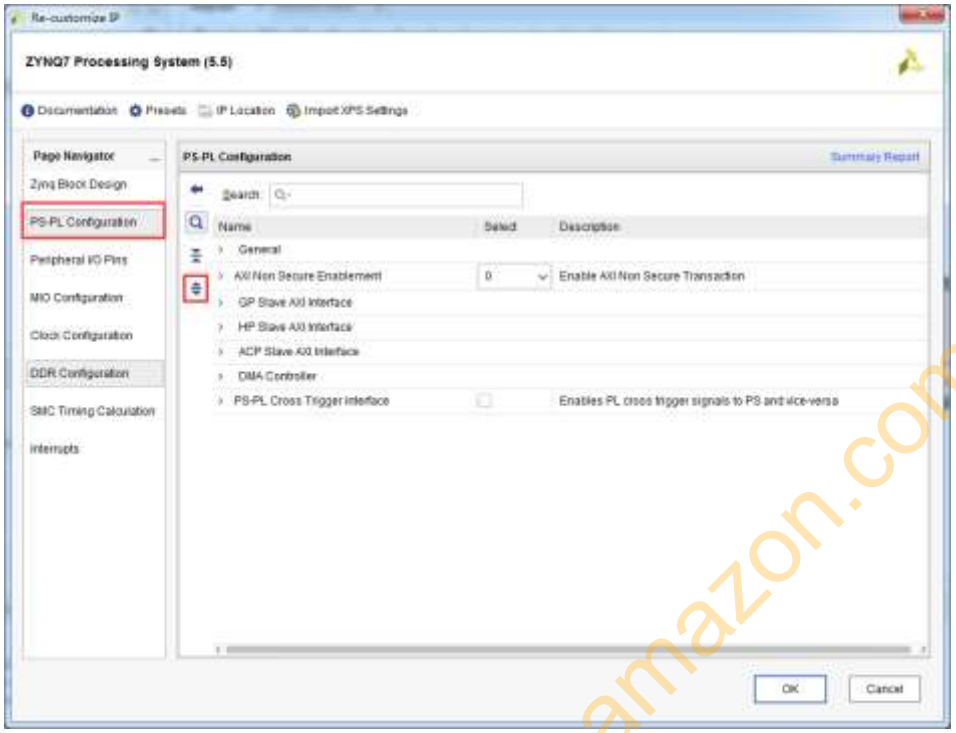
5) Search for "zynq" and double-click "ZYNQ7 Processing System" in the search results list



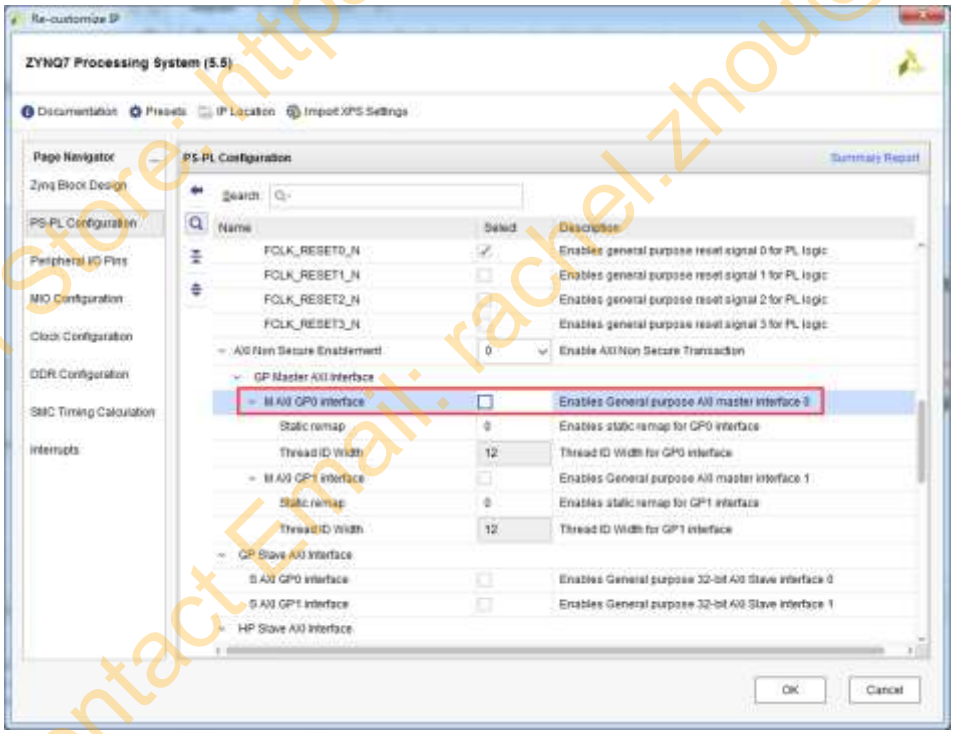
6) Double-click "processing_system7_0" in the block diagram to configure related parameters.



7) Expand all items in the "PS PL Configuration" option



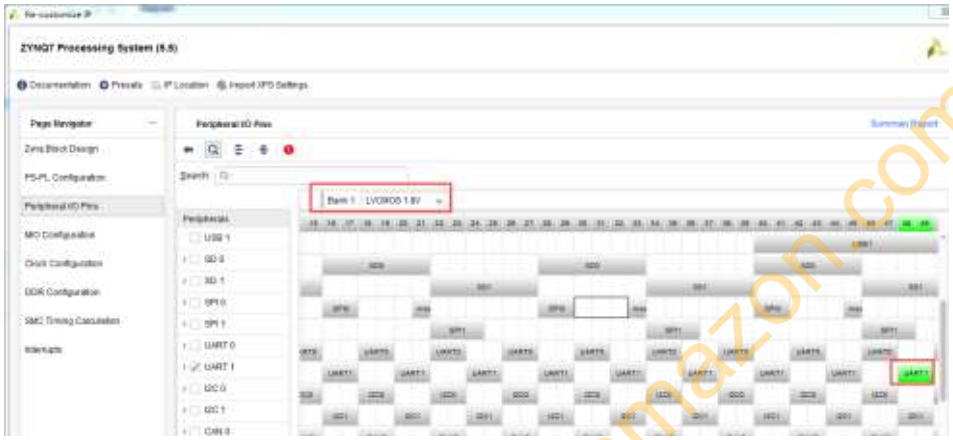
8) Cancel the "M AXI GP0 interface", this interface can expand the AXI interface peripherals on the PL side, so if the PL wants to interact with the PS, it must follow the AXI bus protocol. Xilinx provides us with a large number of AXI interface IP cores.



Part 6.2.1: Uart Configuration

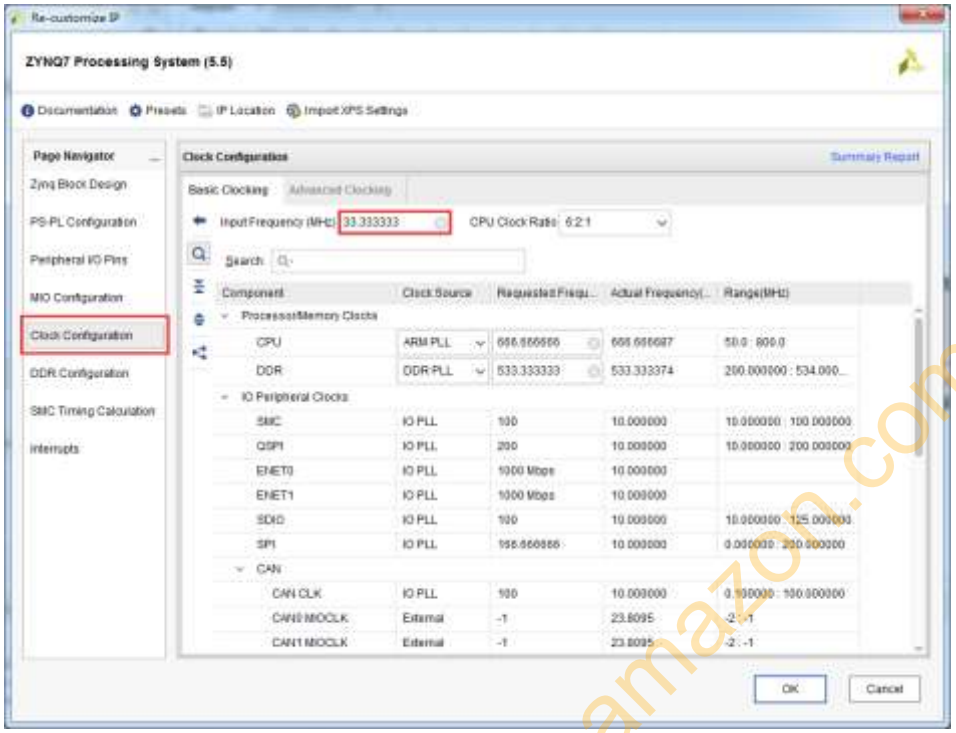
9) From the schematic, we can find the serial port connected to the

MIO48-MIO49 of the PS, so enable UART1 (MIO48 MIO49) in the "Peripheral I/O Pins" option. Select "LVCMOS 3.3V" for Bank 0 voltage, and "LVCOMS 1.8 V" for Bank 1 voltage. This experiment only uses a serial port function, and no other devices are used here.



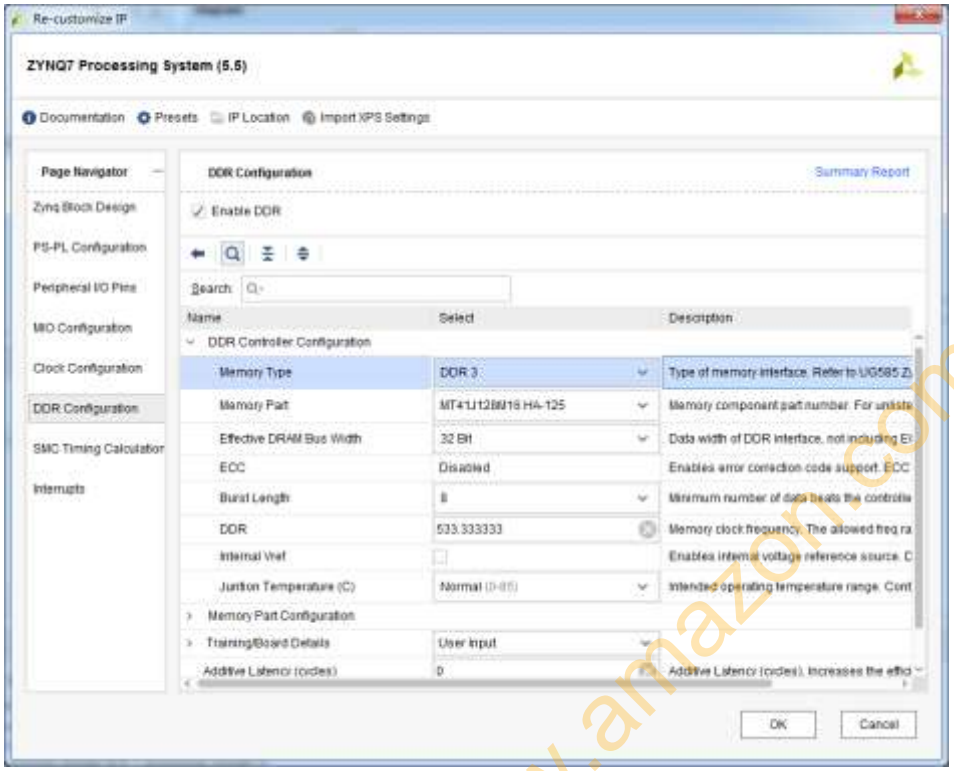
Part 6.2.2: Clock Configuration

10) In the "Clock Configuration" tab, we can configure the PS clock input frequency. The default here is 33.333333 Mhz, which is the same as on the FPGA board. There is no need to modify the CPU frequency. The default is 666.666666 Mhz, and there is no need to modify it here. At the same time, the PS can also provide 4 clocks to the PL side, and the frequency can be configured. It is not needed here, so keep the default.

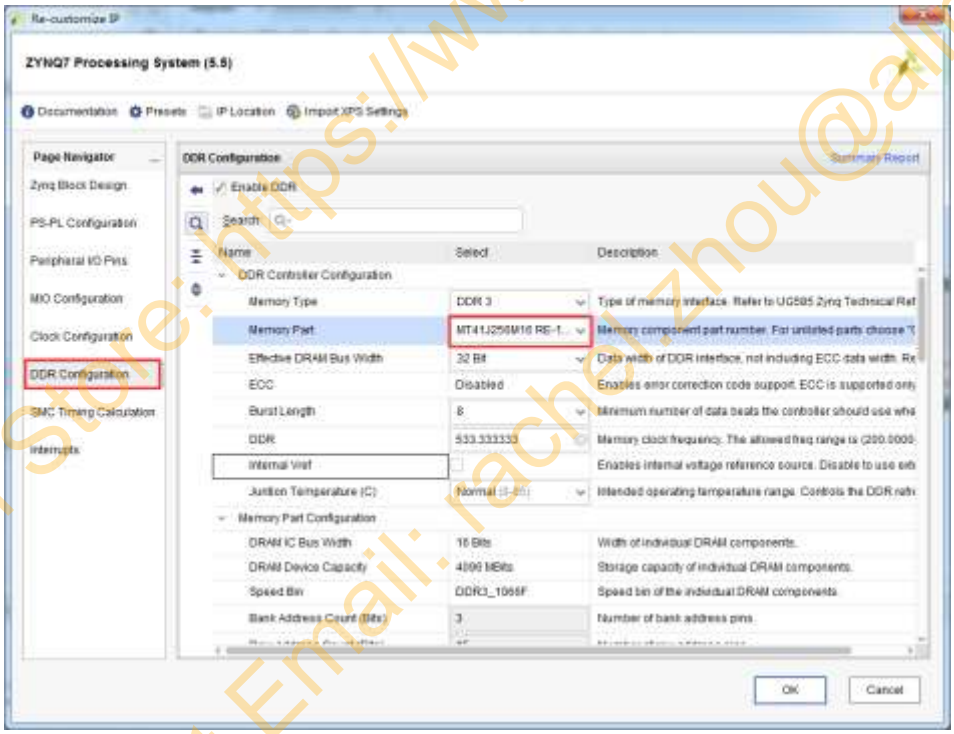


Part 6.2.3: DDR3 Configuration

11) In the "DDR Configuration" tab, you can configure the parameters of the PS terminal ddr. The AX7010 is configured with the DDR3 model as "MT41J128M16 HA-125", and the AX7020 is configured with the DDR3 model as "MT41J256M16 RE-125".***The ddr3 model here is not the ddr3 model on the board, but the model with the closest parameter.*** Effective DRAM Bus Width", select "32 Bit



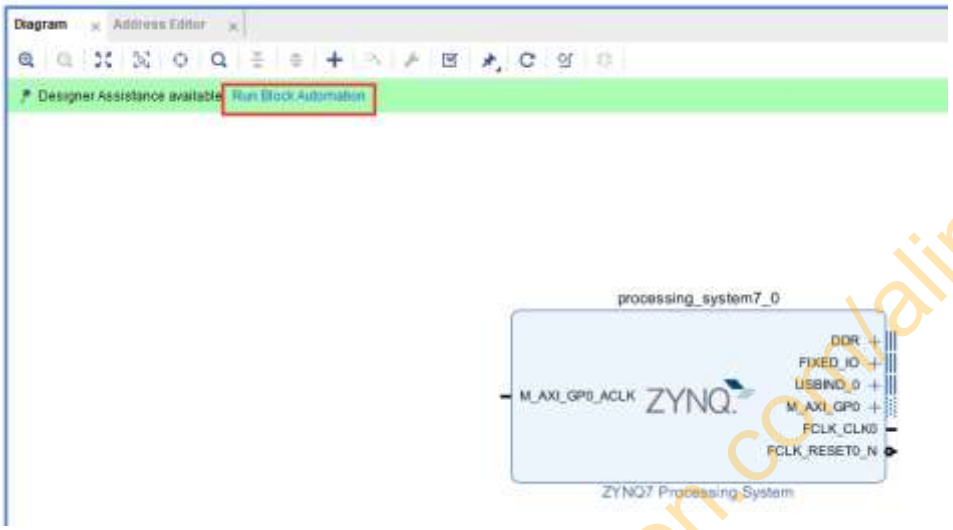
AX7010 DDR3 Configuration



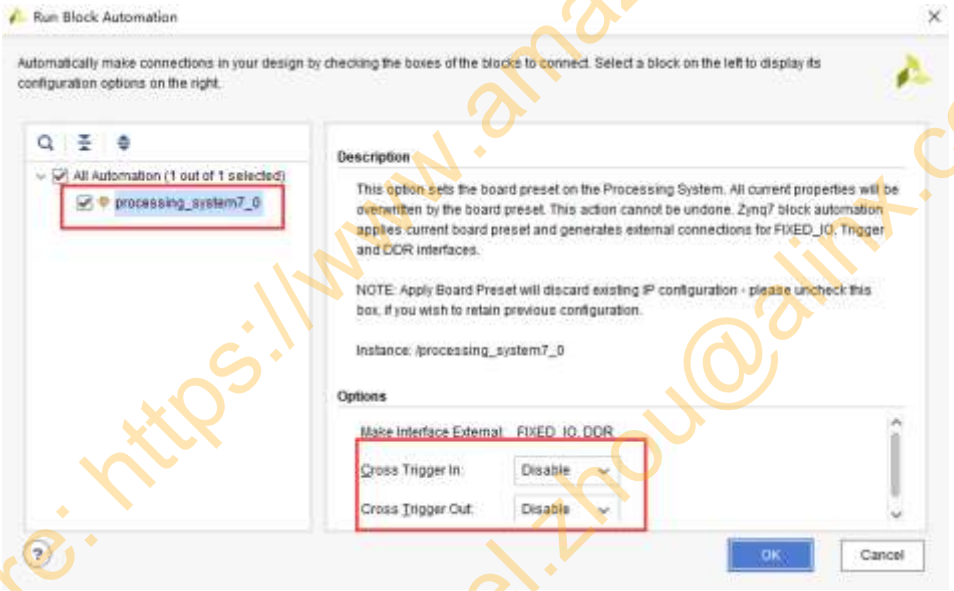
AX7020 DDR3 Configuration

Keep the rest of the defaults and click OK. At this point, the configuration of the ZYNQ core ends

12) Click on “Run Block Automation” and the vivado software will automatically complete some export port work.

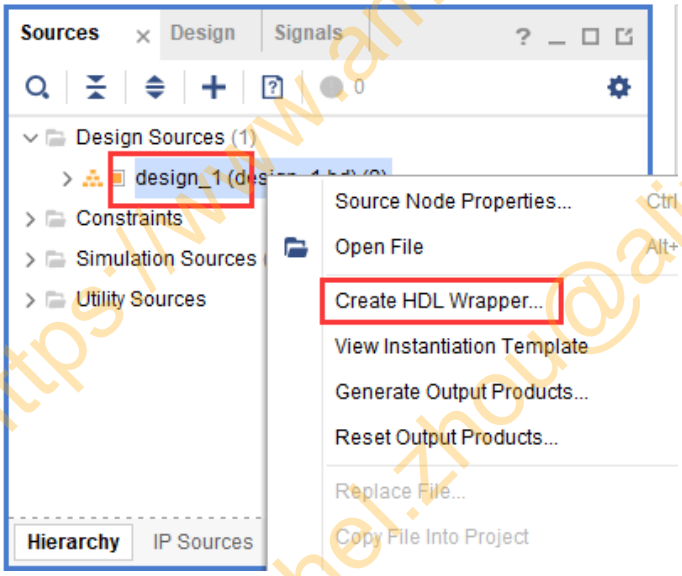


13) Click "OK" by default

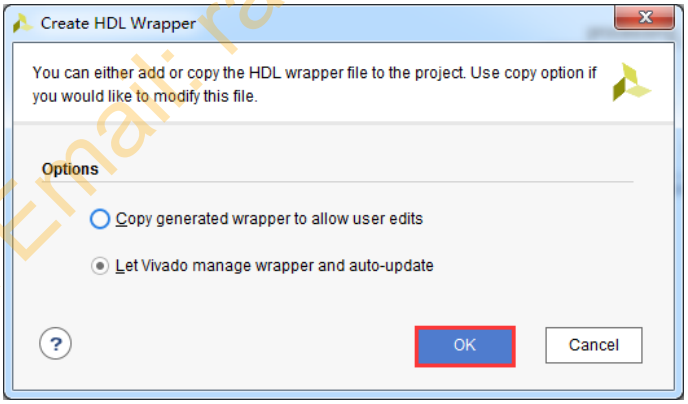


14) After clicking "OK", we can see that the PS side exports some pins, including “DDR” and “FIXED_IO”, DDR is the interface signal of DDR3, FIXED_IO is some fixed interface of PS terminal, such as input clock, PS side reset signal, MIO and so on. Press "Ctrl +s" to save the design.

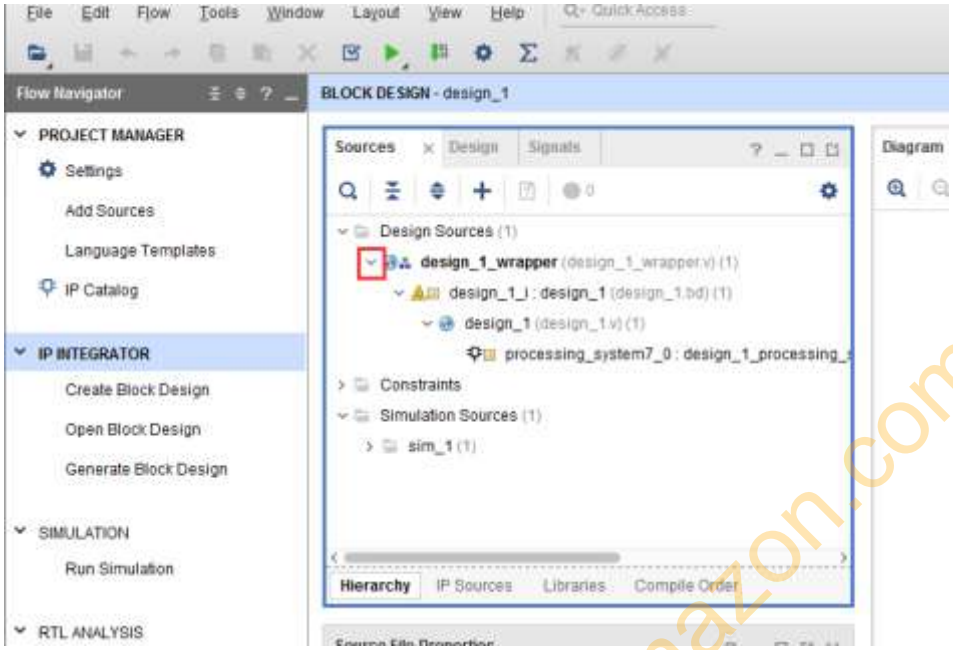
15) Select the Block design, right click on "Create HDL Wrapper...", create a Verilog or VHDL file, and generate the HDL top-level file for the “block design”



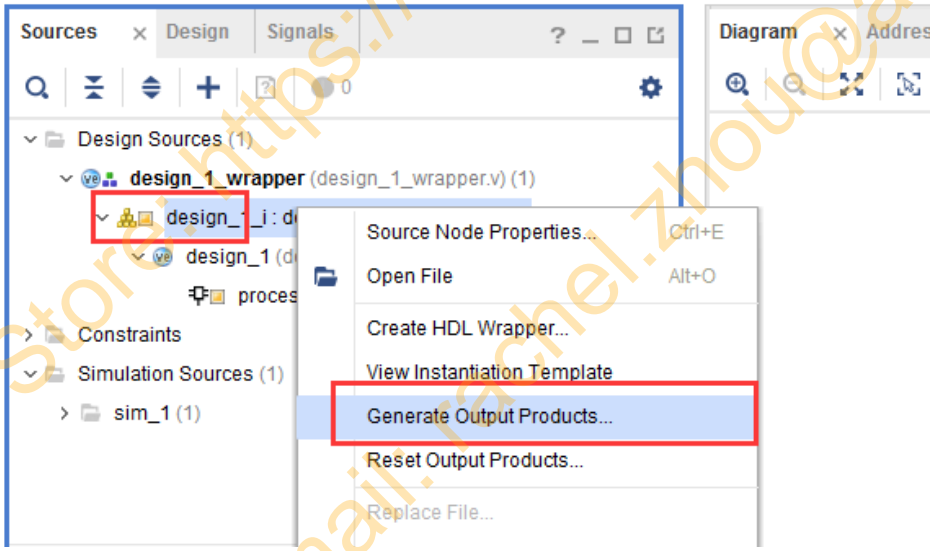
16) Keep the default options and click "OK"



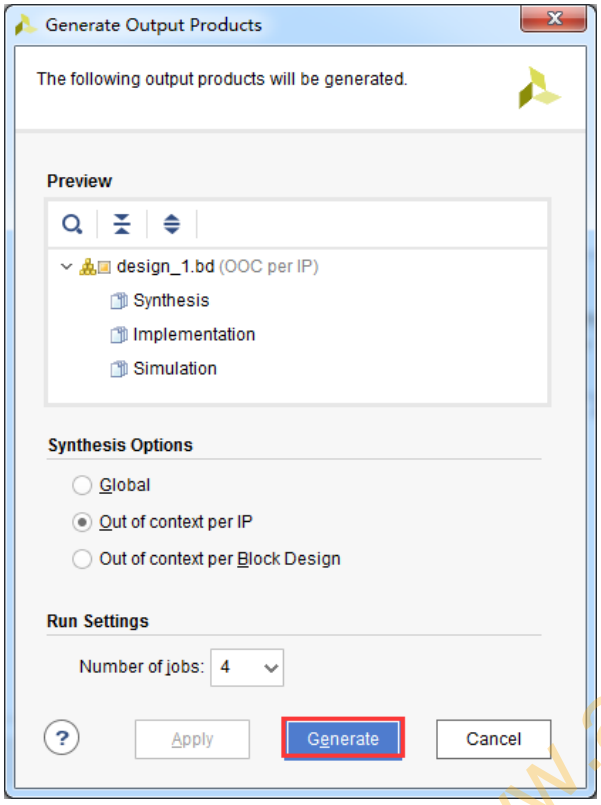
17) Expand the design to see that the PS is being used as a normal IP.



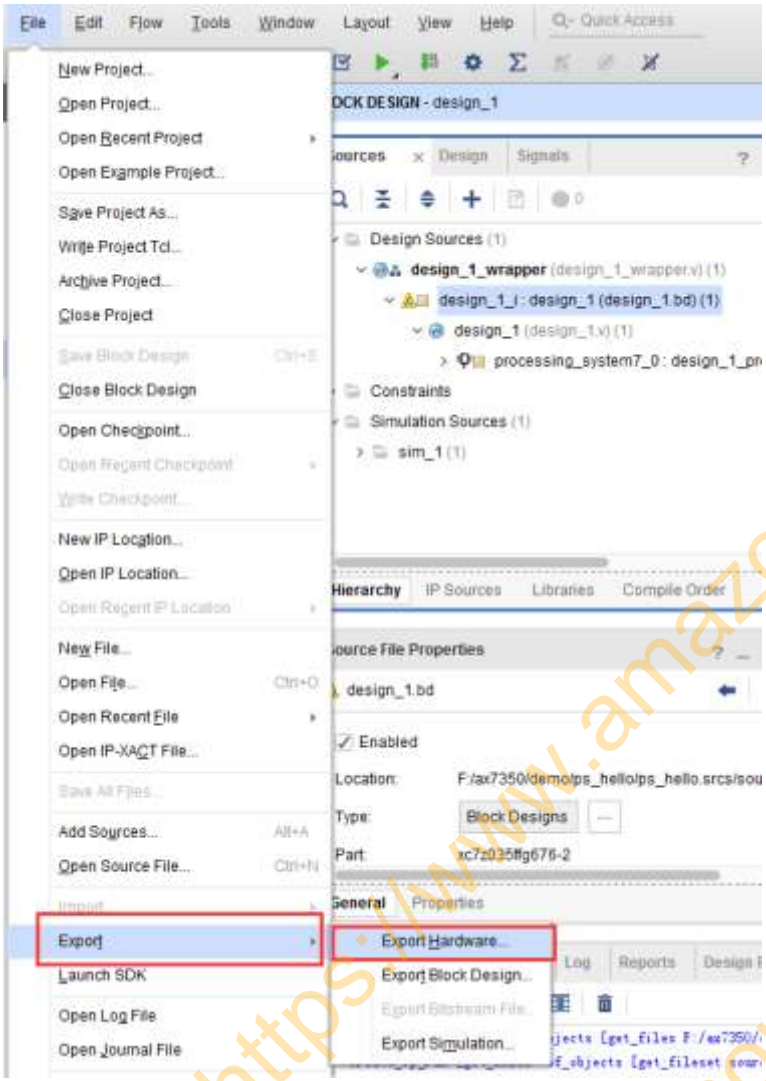
18) Select the block design, right click "Generate Output Products", this step will generate the block output file, including IP, instantiation template, RTL source file, XDC constraints, third-party integrated source files and so on. For subsequent operations.



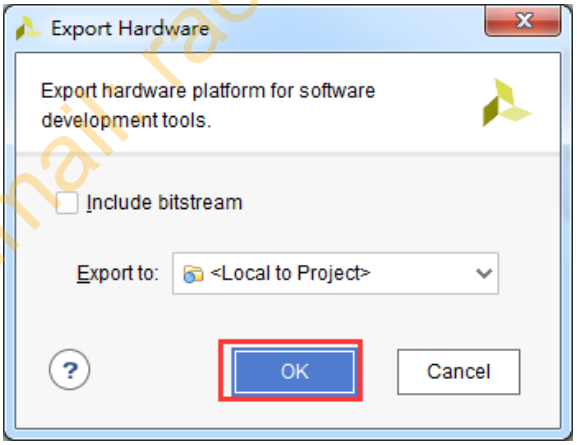
19) Click on "Generate"



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



21) In the pop-up dialog box, click "OK", because the experiment is only using the PS serial port, no PL participation, there is no "Include bitstream" enabled here.



At this point, there will be more “xx.sdk” folder, and there is a “hdf” file, this file is the file containing the Vivado hardware design information for

software developers to use.

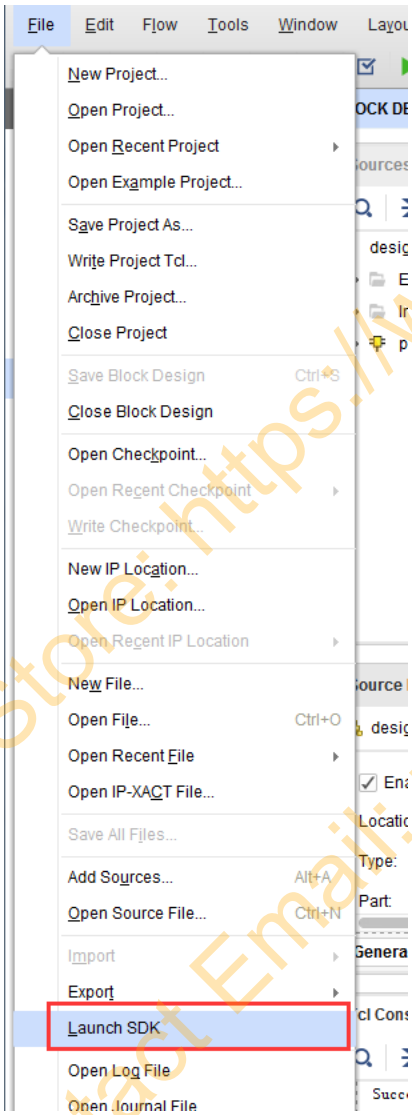
At this point, the FPGA engineer's work has come to an end.

Software engineer work content

The following is the software engineer responsible for the content

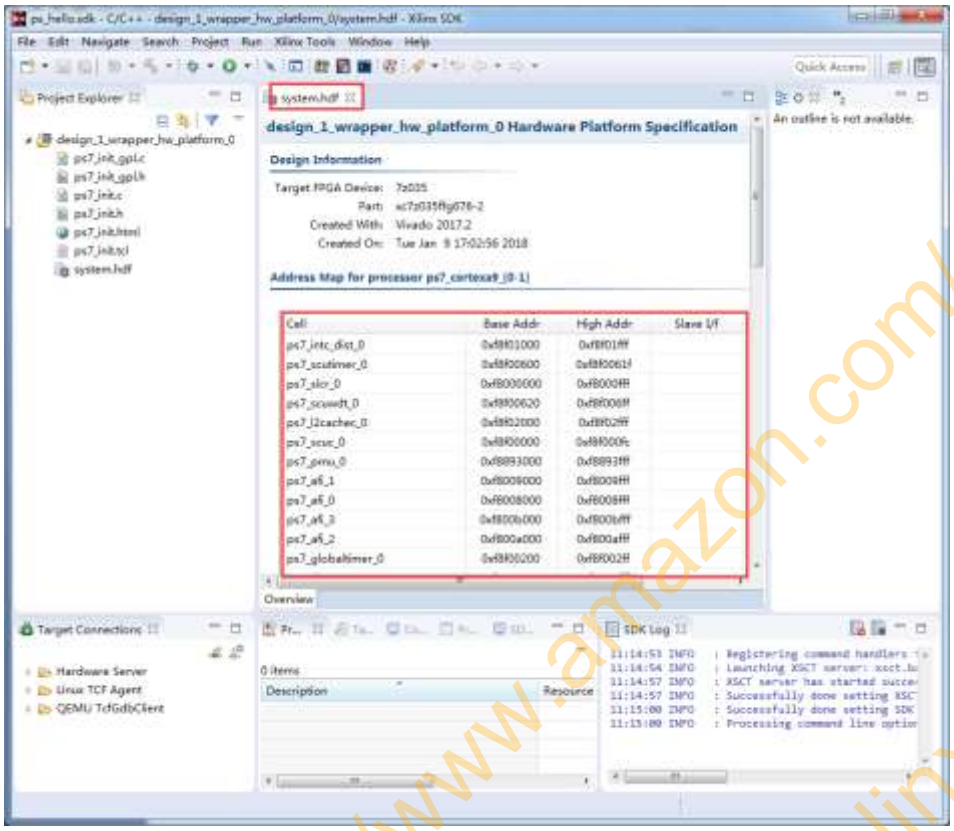
Part 6.3: SDK debugging

1) Click on the Vivado menu "File -> Launch SDK" to launch the SDK

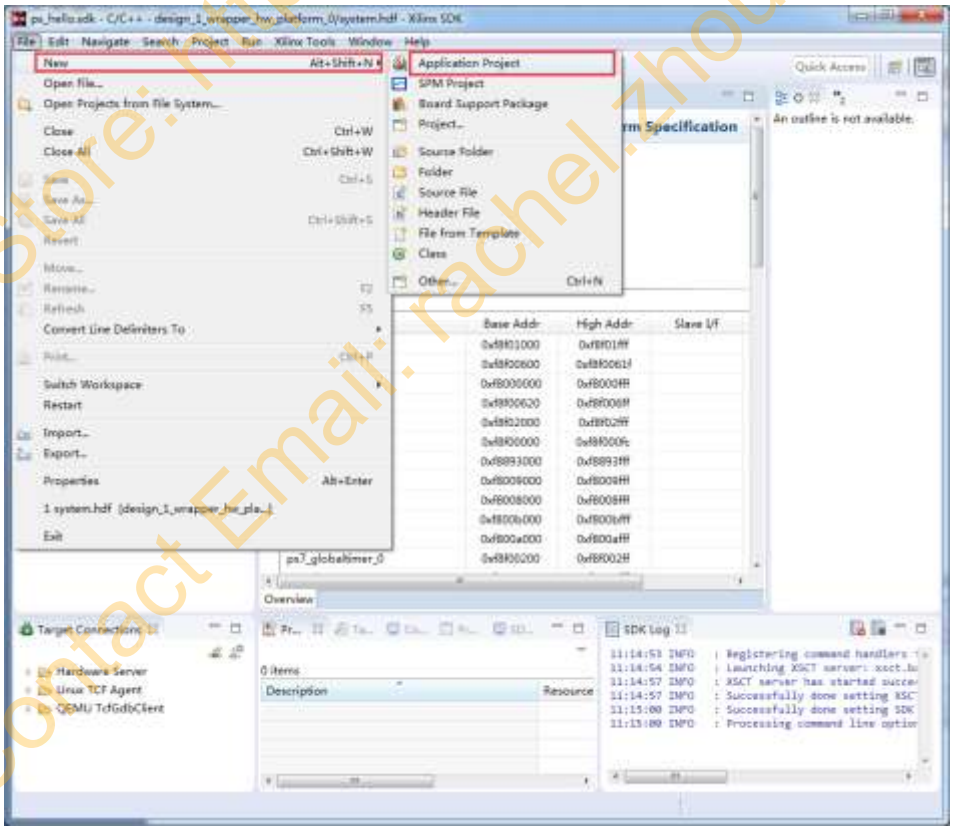


2) After starting the SDK, we will see a folder with a file named "system.hdf", which contains information about the Vivado hardware design, which can be used for software development, as well as a

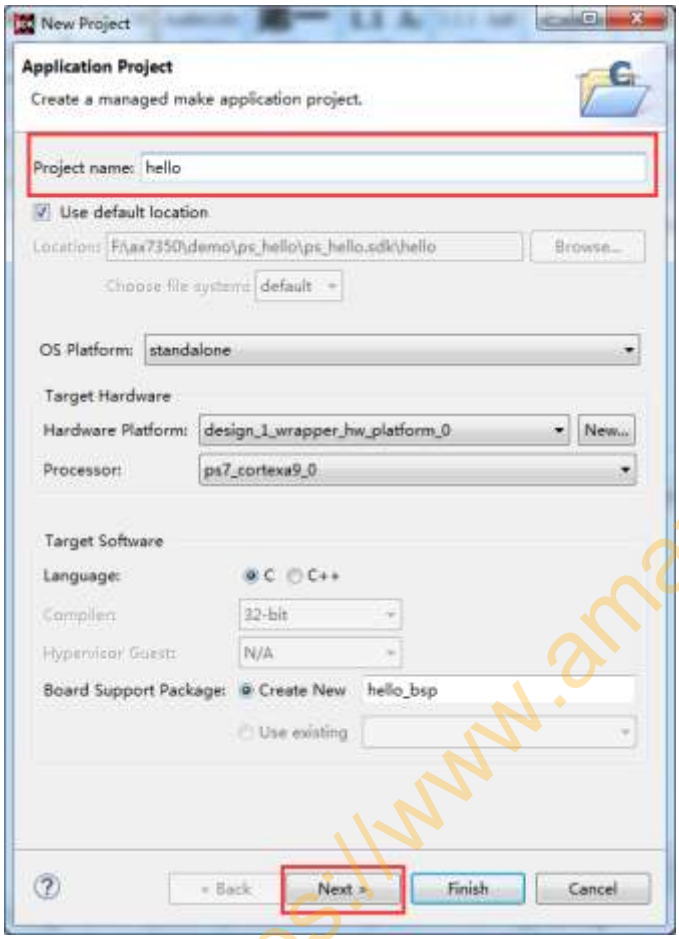
register list for the PS peripherals.



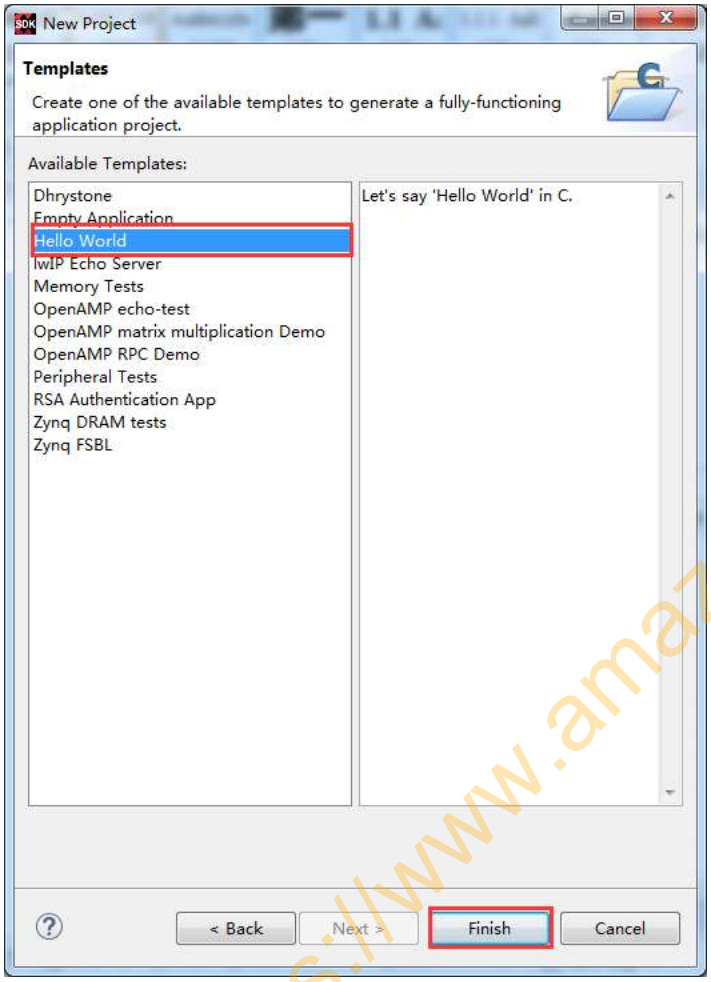
3) Create an APP project in the SDK's menu "New -> Application Project"



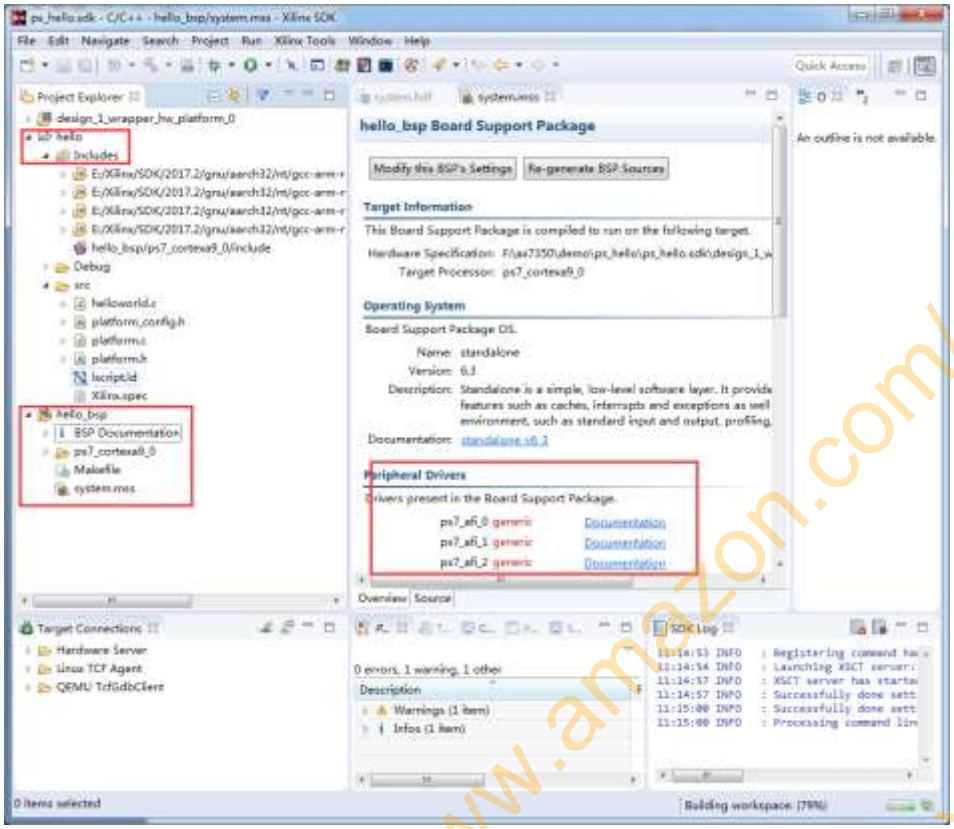
4) "Project name" fills in "hello", other defaults, click "Next"



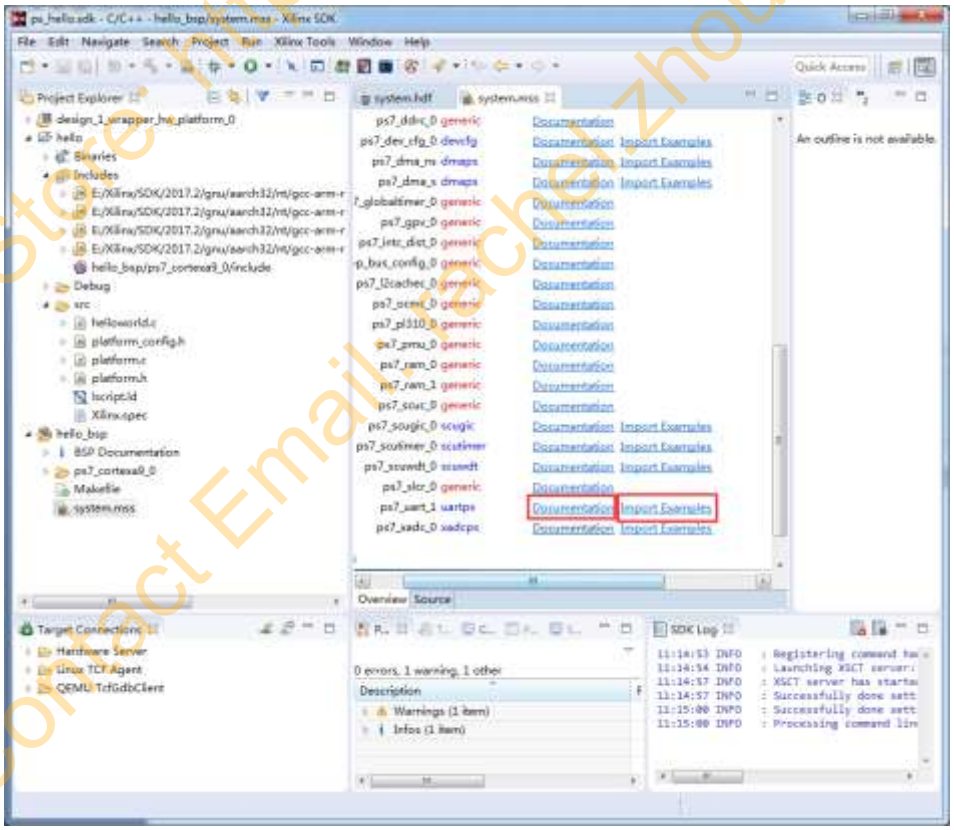
5) Select "Hello World" for the template and click "Finish"



6) You can see that the SDK creates a "hello" directory, and a "hello_bsp" directory. You can find a lot of useful information in the "hello_bsp" directory. The software developers know that the BSP is the Board Support Package. This means that the driver files needed for development are included for application development.

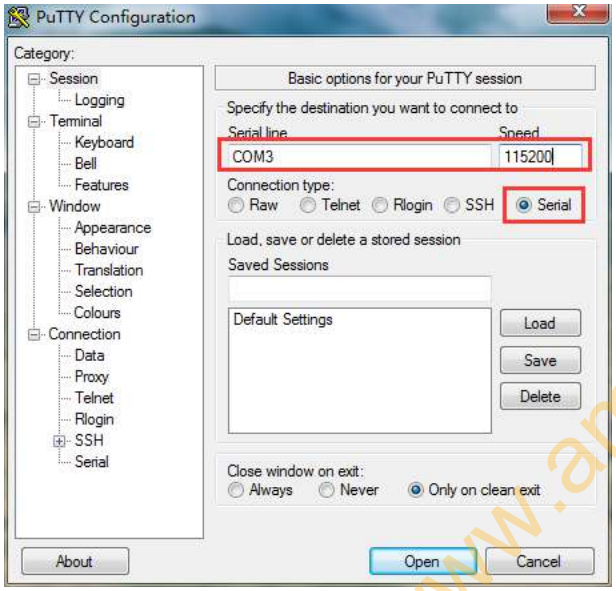


7) Double-click on "system.mss" and you can see that some PS peripherals also provide routines, which are used to learn first-hand information about learning these peripherals.

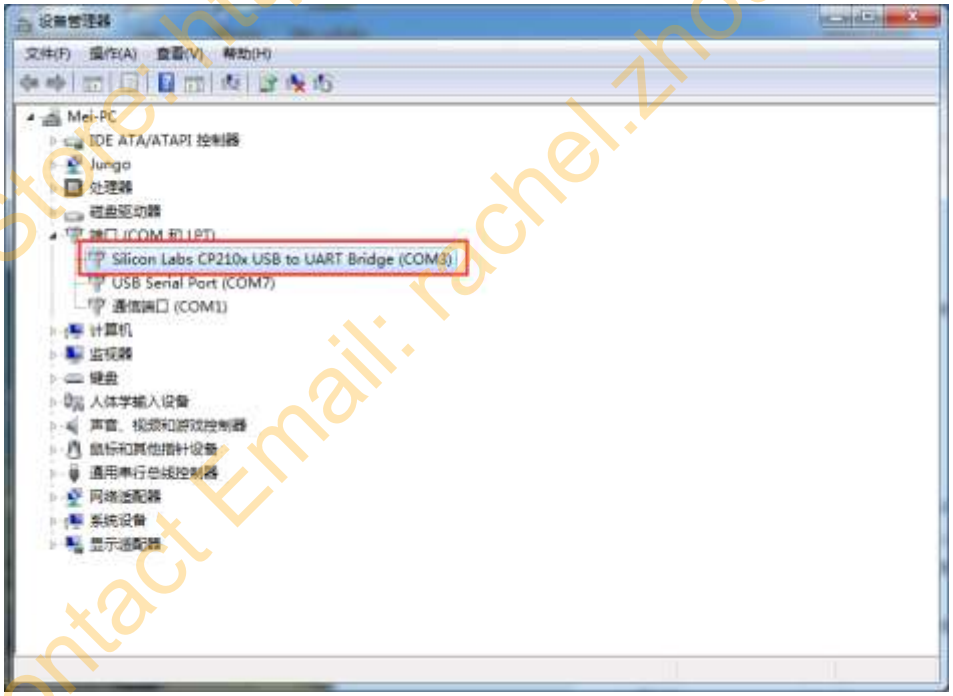


8) Connect the JTAG cable to the FPGA development board, UART USB cable to PC

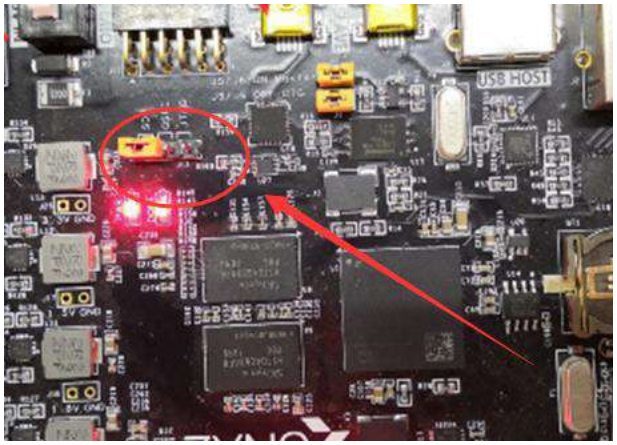
9) Using PuTTY software as a serial terminal debugging tool, PuTTY is the free installation software.



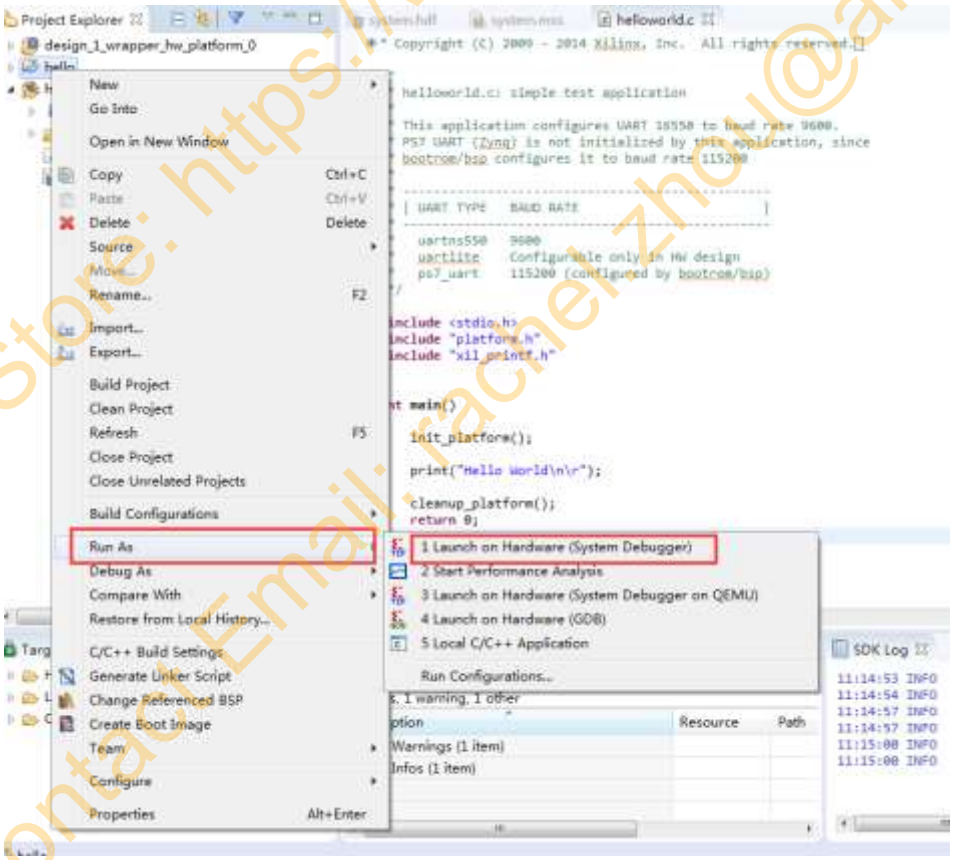
10) Select Serial, Serial line to fill in COM3, Speed fill in 115200, COM3 serial port number is filled in according to the display in the device manager, click "Open"



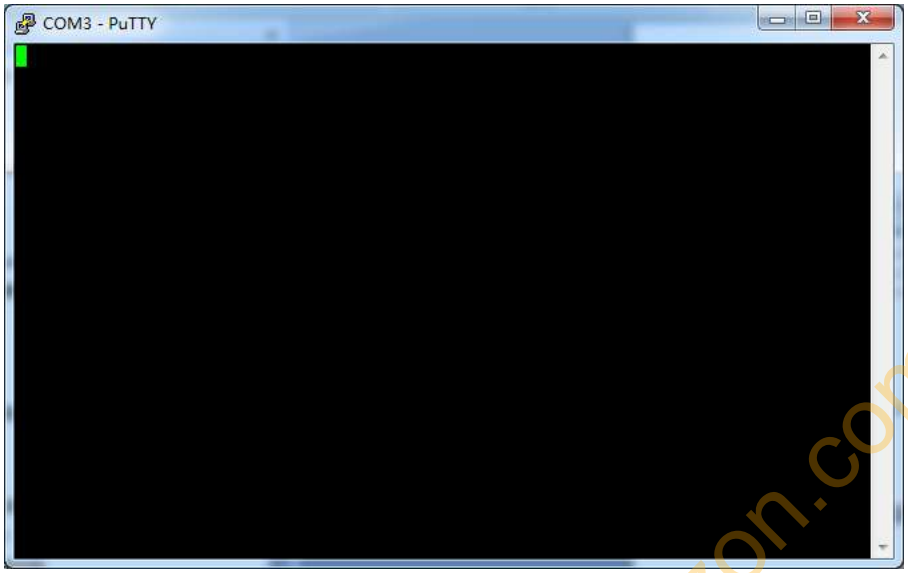
11) It is best to set the boot mode of the FPGA development board to JTAG mode before powering up.



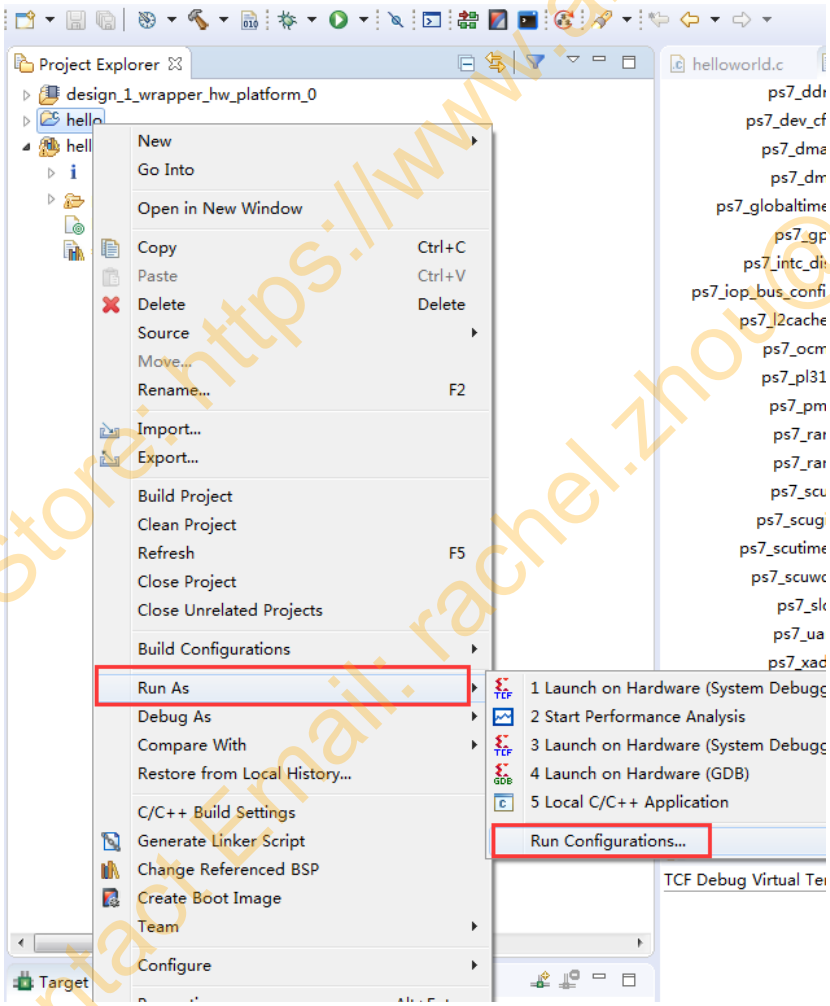
12)To power on the board, prepare to run the program. The board is shipped with the program. Here you can select the JTAG mode for the operating mode and then power it back on. Select "hello", right click, you can see a lot of options, this experiment uses "Run as" here, is to run the program, "Run as" has a very good option, choose the first "Launch on Hardware (System Debuger)", use system debugging to run the program directly



13) At this time, observe the PuTTY software, there may be output display, or there may be no output

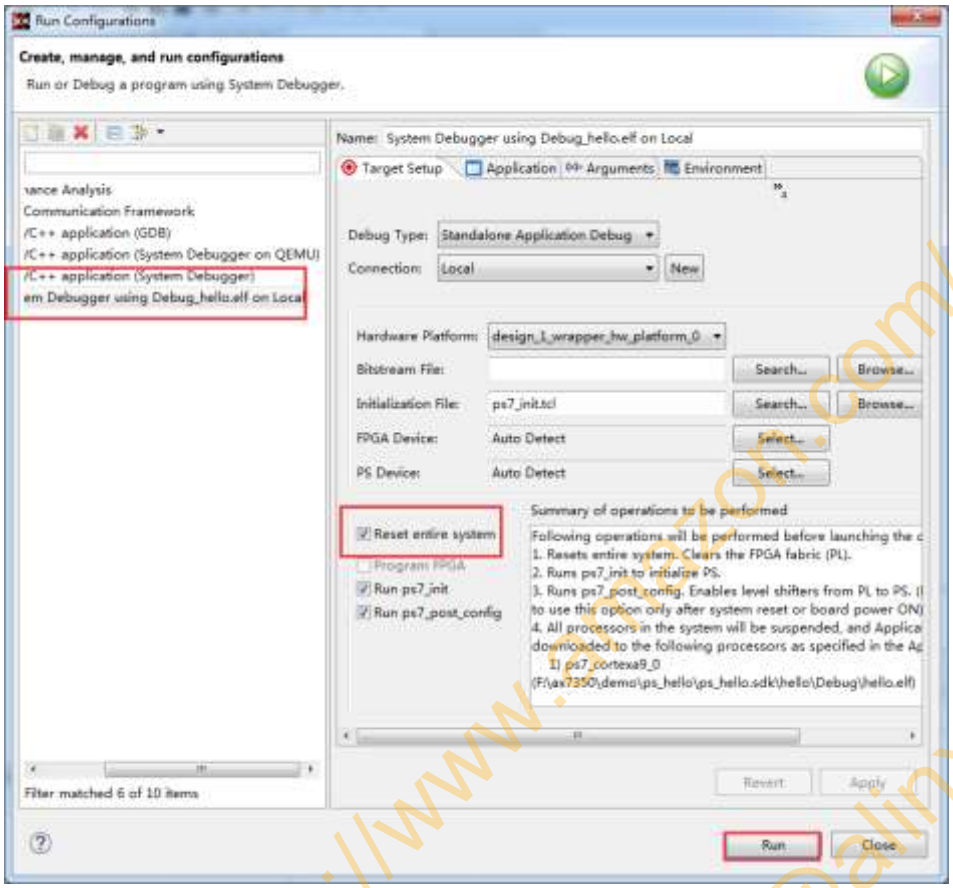


14) In order to ensure reliable debugging of the system, you need to add a configuration, right click "Run As -> Run Configuration..."

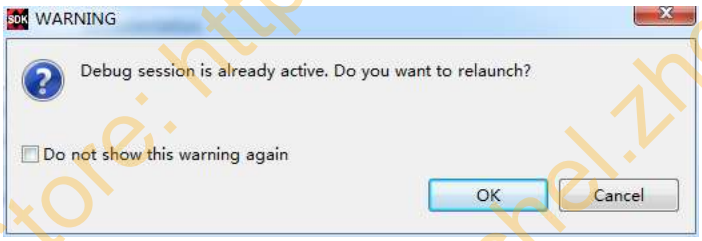


15) Select “Reset entire system” to reset the entire system. If there is a PL design in the system, you must also select “Program FPGA” and

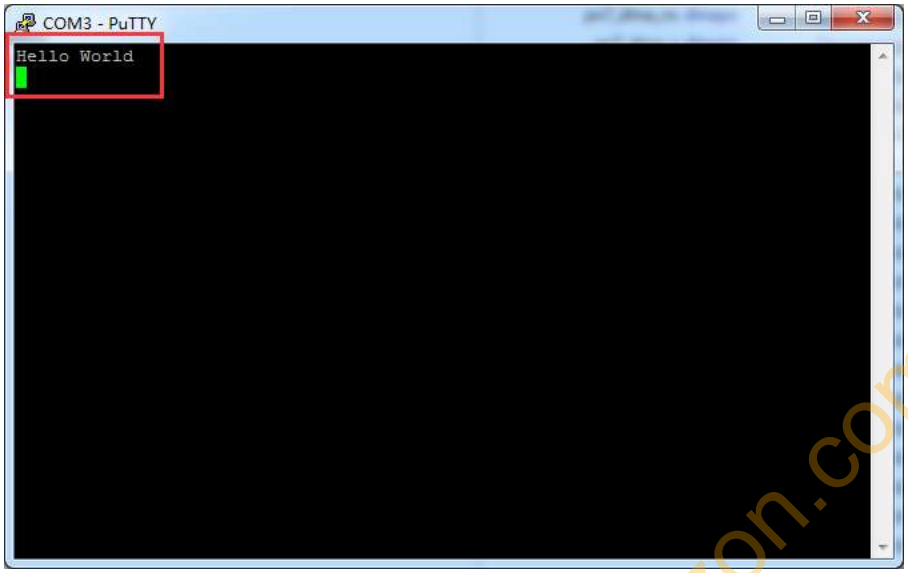
click “Run” again.



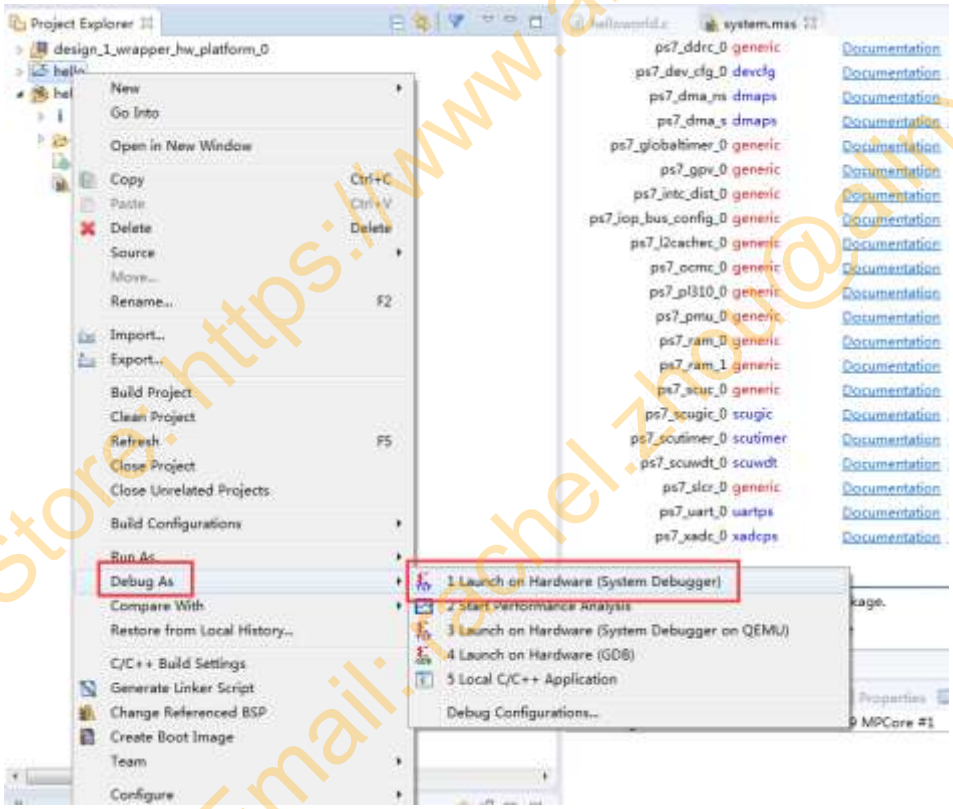
16) Click "OK" to confirm re-run



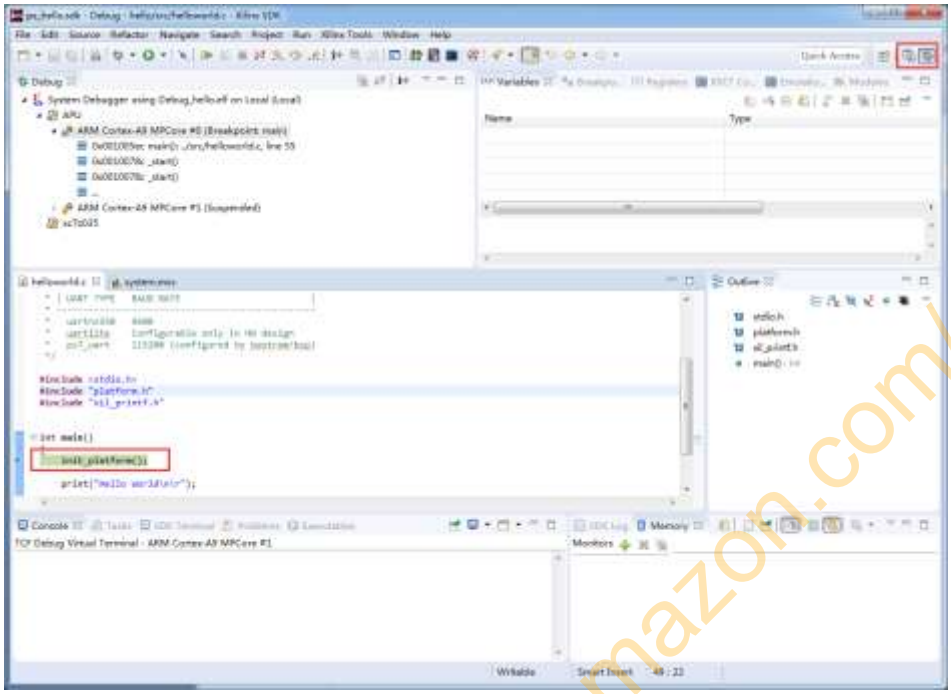
17)This time you can see that the familiar "Hello World" is displayed.



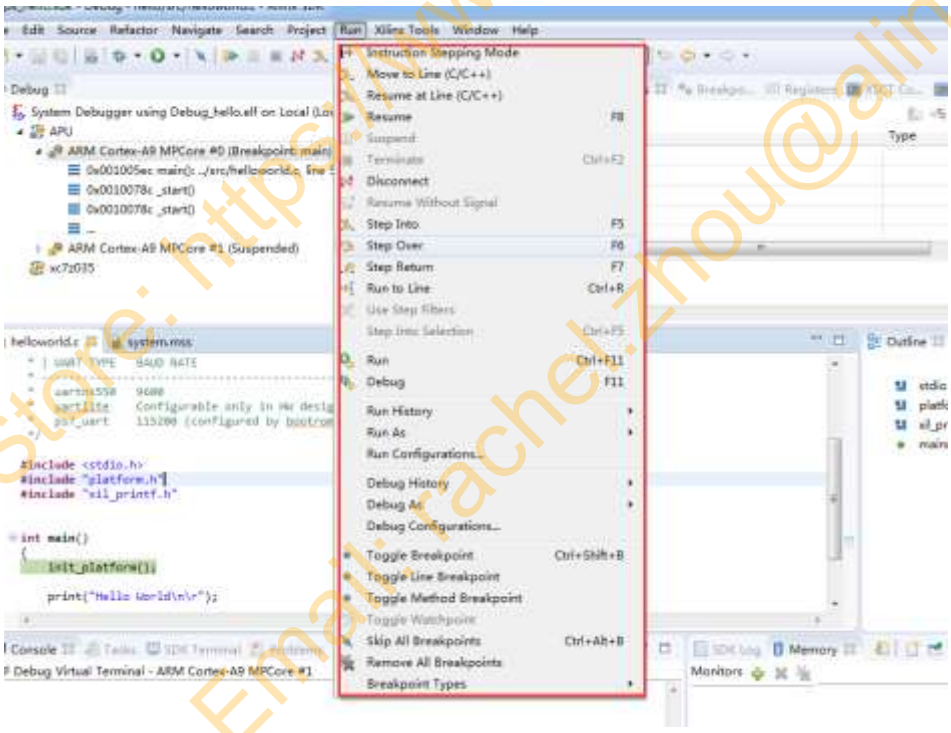
18) In addition to "Run As", you can also "Debug As", which can set breakpoints and single step operation.



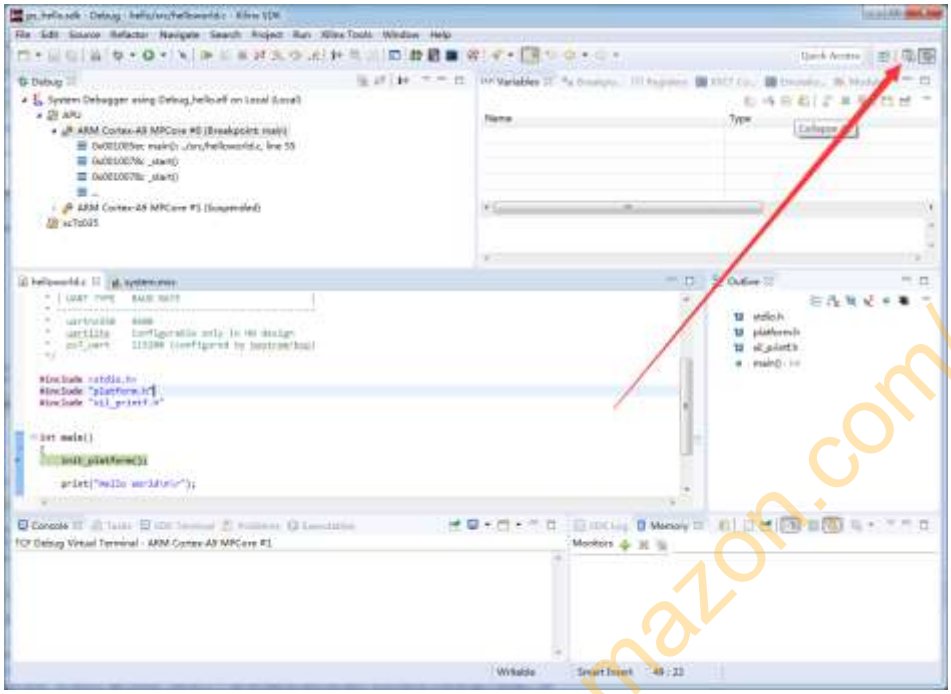
19) Enter Debug mode



20) Like other C language developments, the IDE can run step by step, set breakpoints, etc.



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



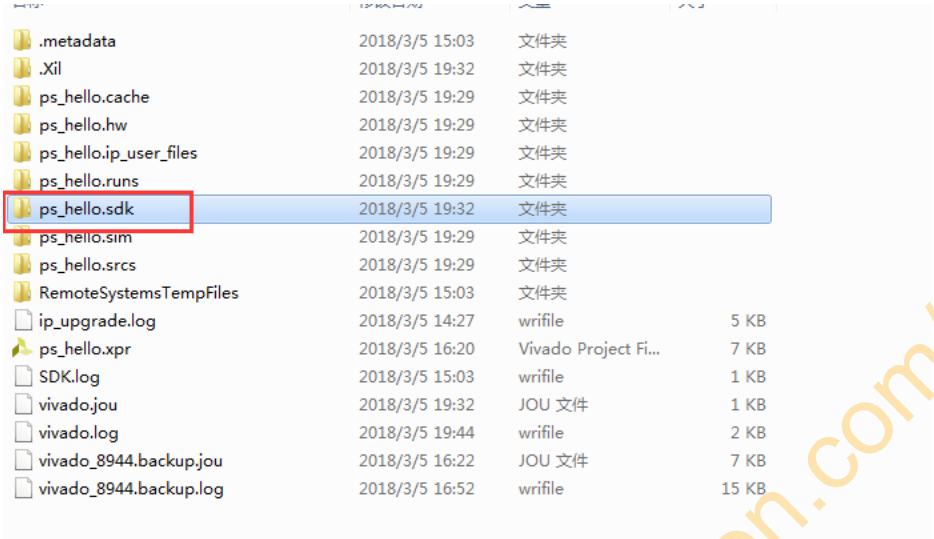
Part 6.4: Experiment summary

This experiment uses a simple Hello World to explain the use of the SDK. The SDK has very powerful functions and cannot be explained one by one. It requires engineers to gradually master it during continuous use.

Part 6.5: Q&As

Part 6.5.1: No window pops up after launching the SDK via vivado

1) Be sure to install the SDK when installing Vivado software.
2) There is a sdk directory before starting the SDK software, which may cause the SDK to fail to be started. Delete this directory and try again.



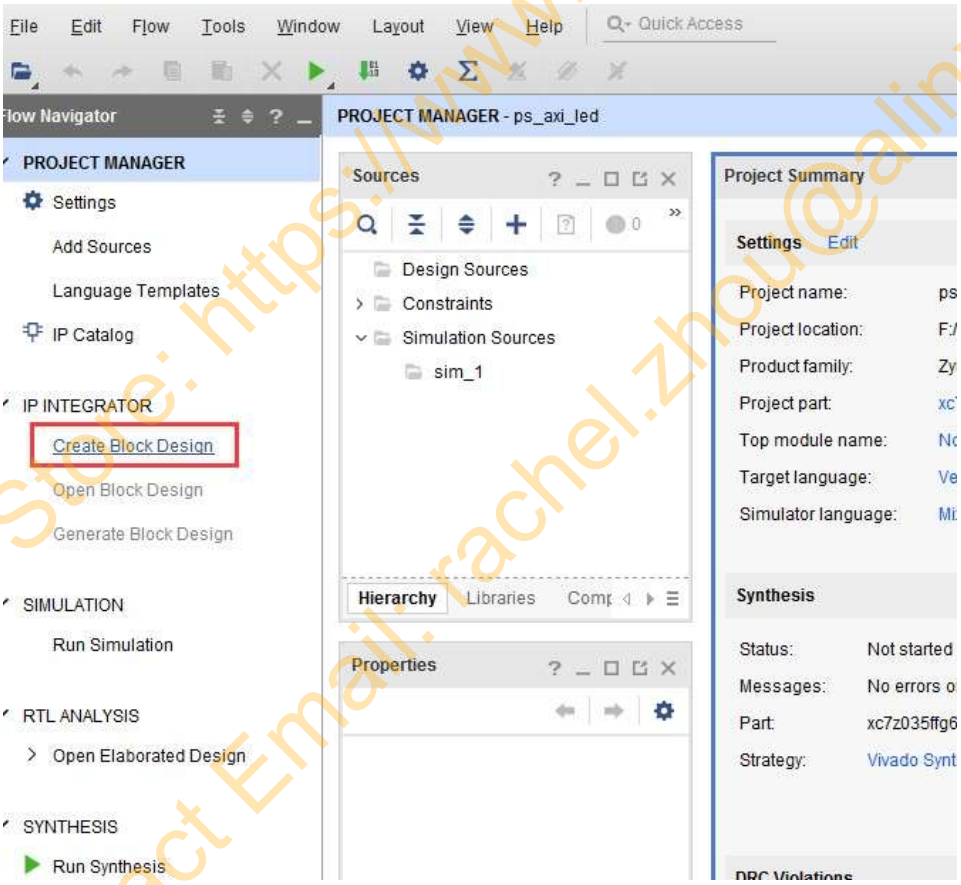
Part 7: PS lights up the LED lights of the PL

The experiment Vivado project is "ps_axi_led".

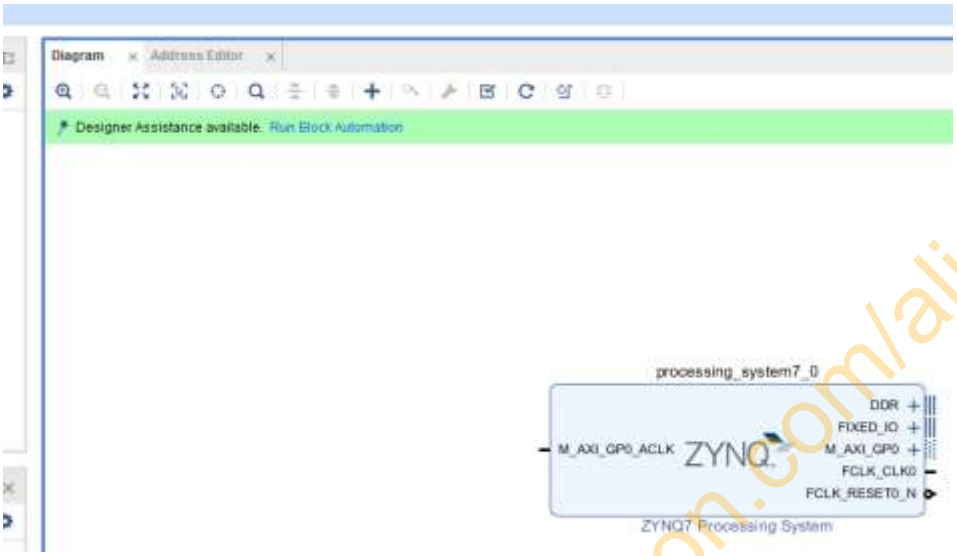
The biggest question about using zynq is how to use PS and PL together. In other SOC chips, there will be GPIO. In this experiment, an AXI GPIO IP core is used, Let the PS side control the LEDs on the PL side through the AXI bus. The experiment is simple, but demonstrated how PL and PS are combined.

Part 7.1: Building a Vivado project

1) Create a Vivado project called "ps_axi_led", which means that the PS controls the LEDs through the AXI bus.
2) Create a block design

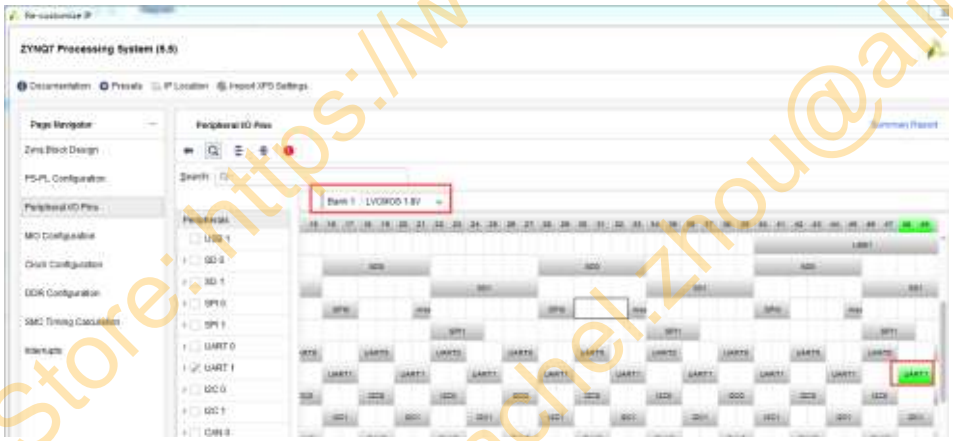


3) Add ZYNQ processor



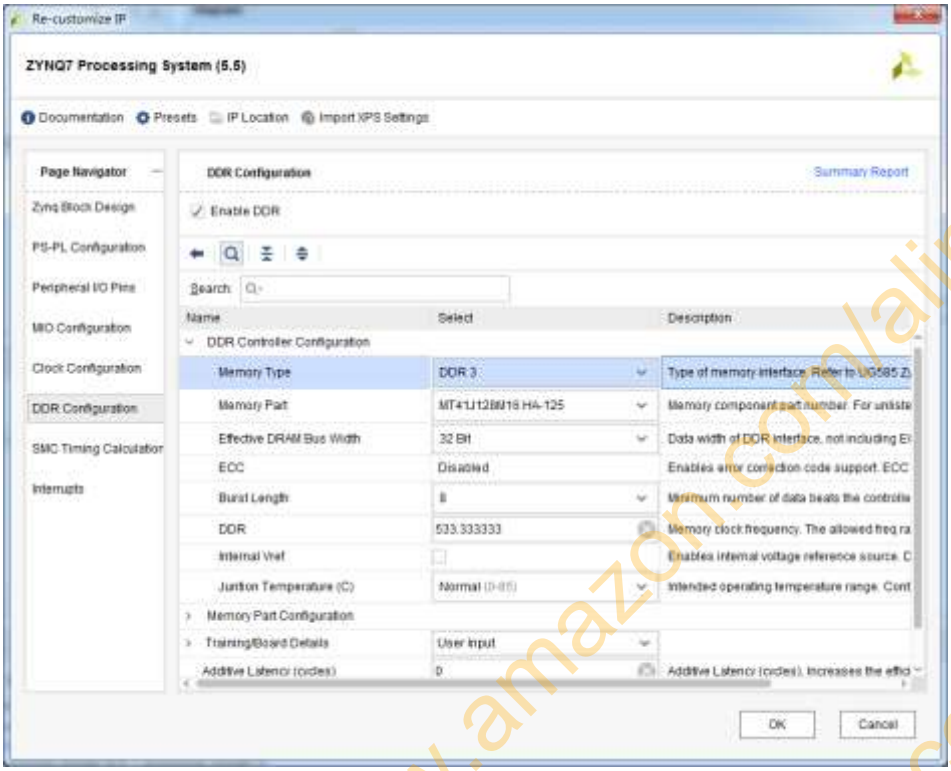
Part 7.1.1: Uart Configuration

4) Configure the Bank1 level standard to LVCMOS 1.8V. If you do not configure the Bank1 level standard, the serial port may not be able to receive. To enable the serial port, use MIO48 MIO49

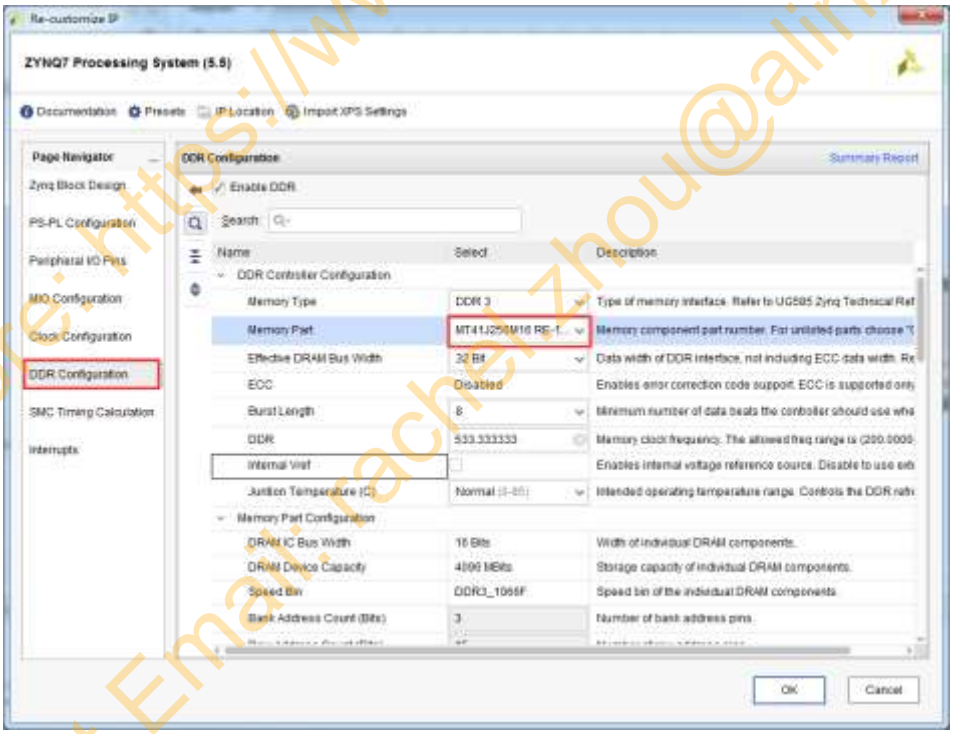


Part 7.1.2: DDR3 Configuration

5) The AX7010 FPGA development board configuration DDR3 model is "MT41J128M16 HA 125". The AX7020 FPGA development board configuration DDR3 model is "MT41J256M16 RE 125". Here the ddr3 model is not the ddr3 model on the board, but the model with the closest parameters.

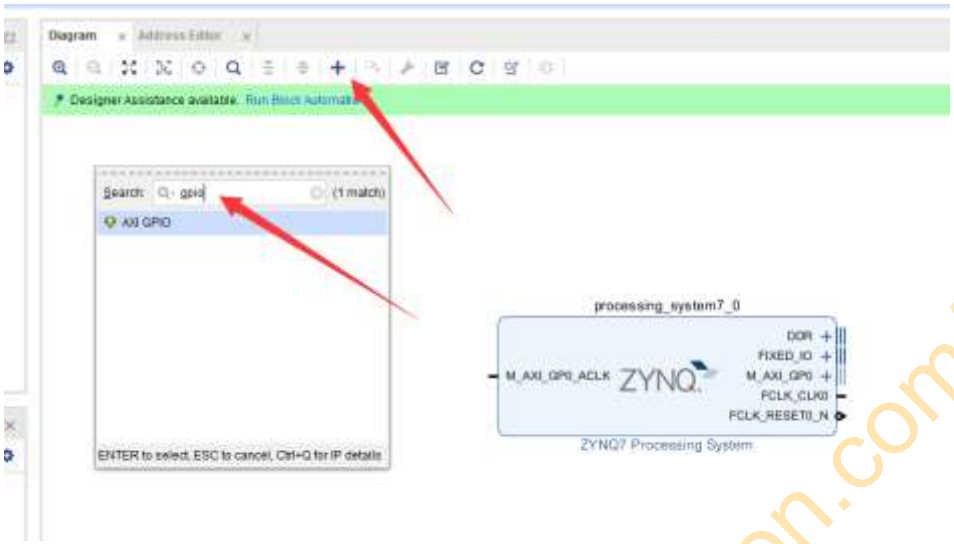


AX7010 DDR3 Configuration



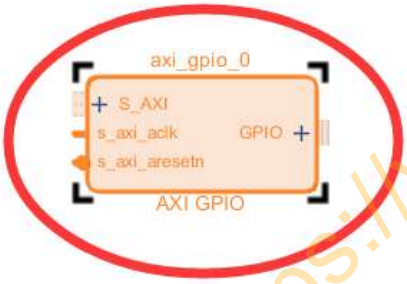
AX7020 DDR3 Configuration

6) Add an AXI GPIO IP core

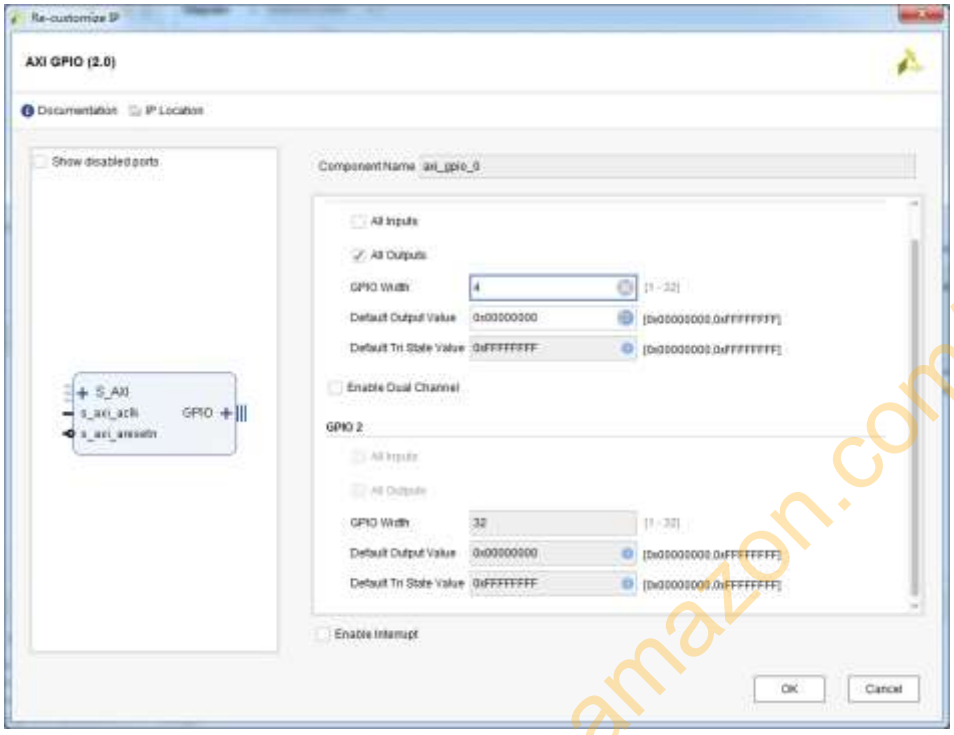


Part 7.1.3: Add AXI GPIO

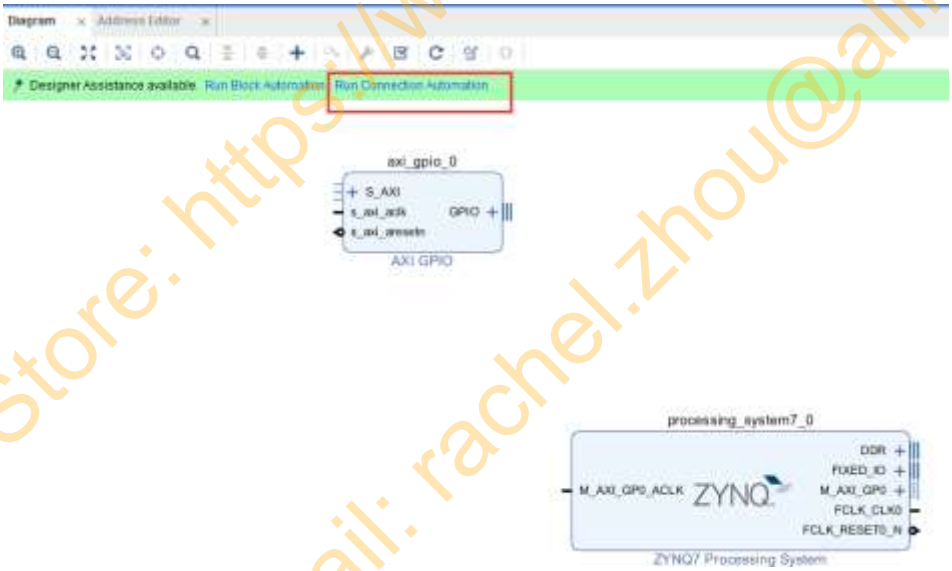
7) Double-click the "axi_gpio_0" configuration parameter you just added.



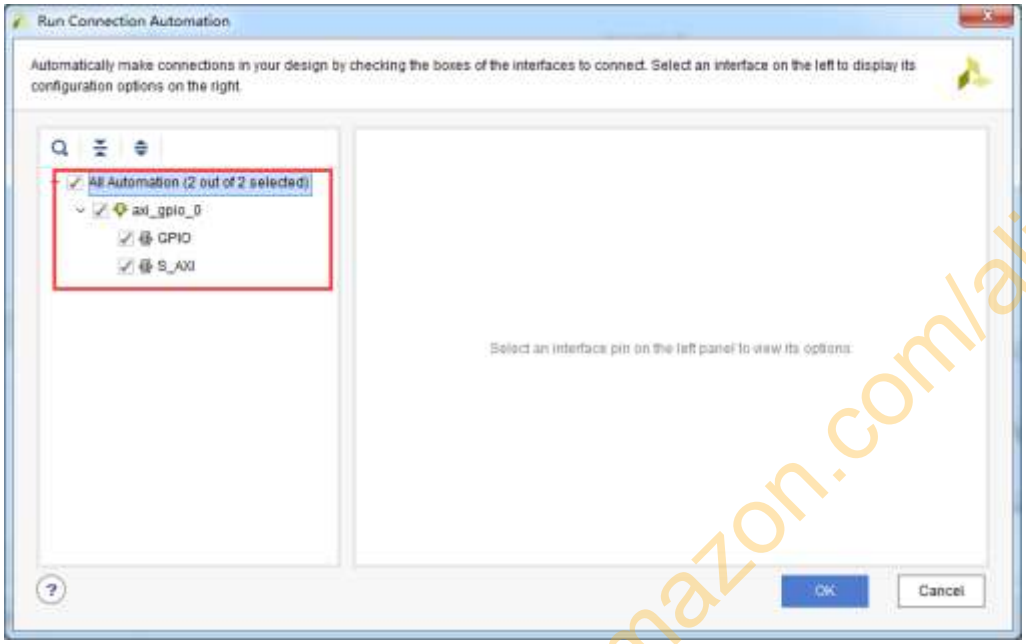
8) Select “All Outputs”, because here the LED is controlled, as long as the output is OK, “GPIO Width” is filled in 4, control 4 LEDs, click OK



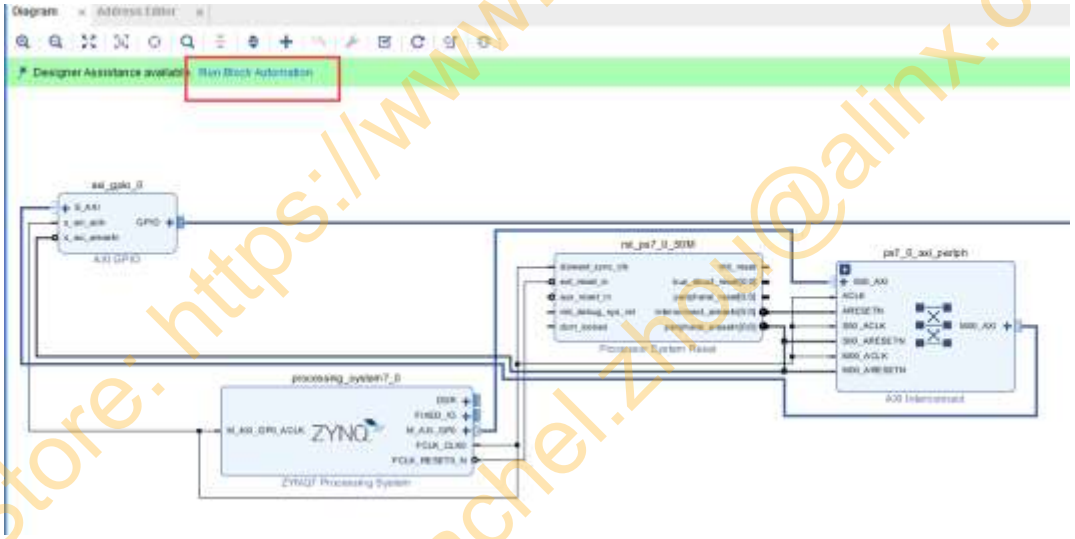
9) Click "Run Connection Automation" to complete some automatic connection



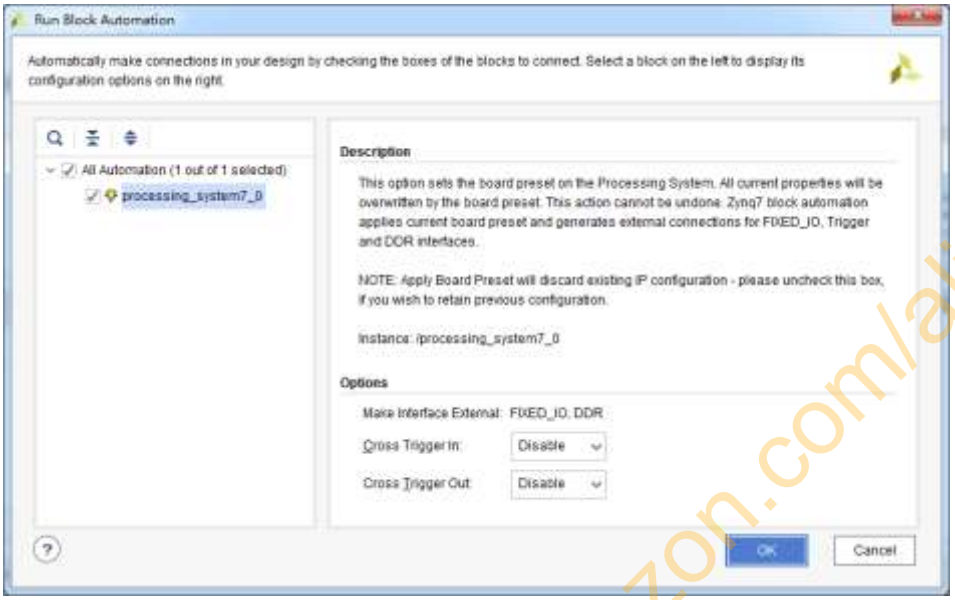
10) Select the port to connect automatically, select all here, click OK



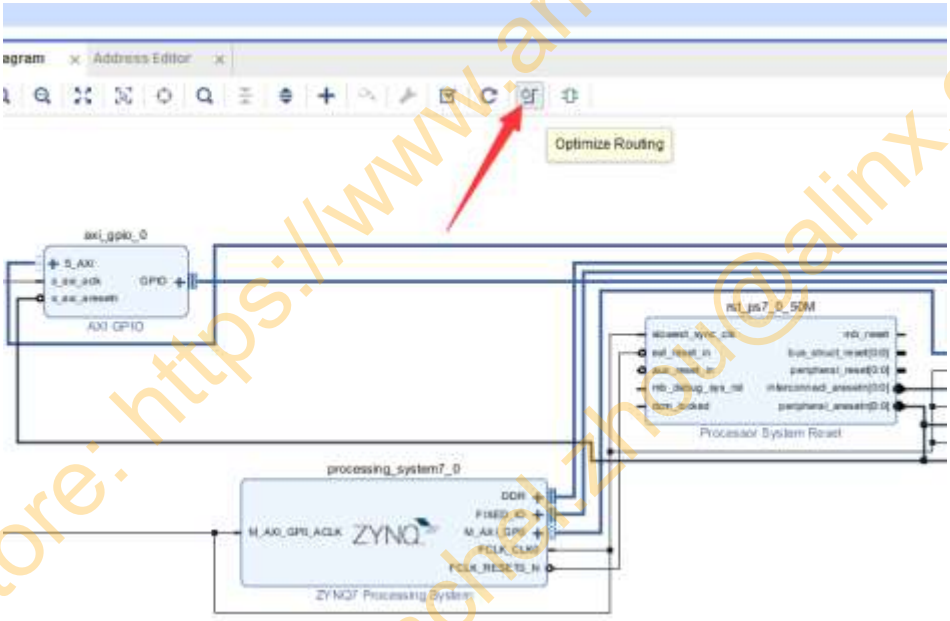
11) Click on "Run Block Automation"



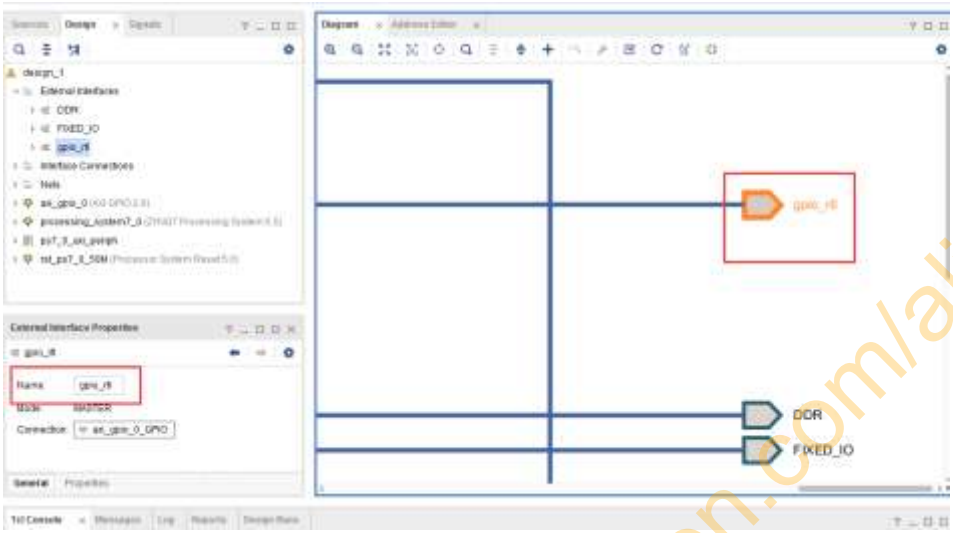
12) Click "OK"



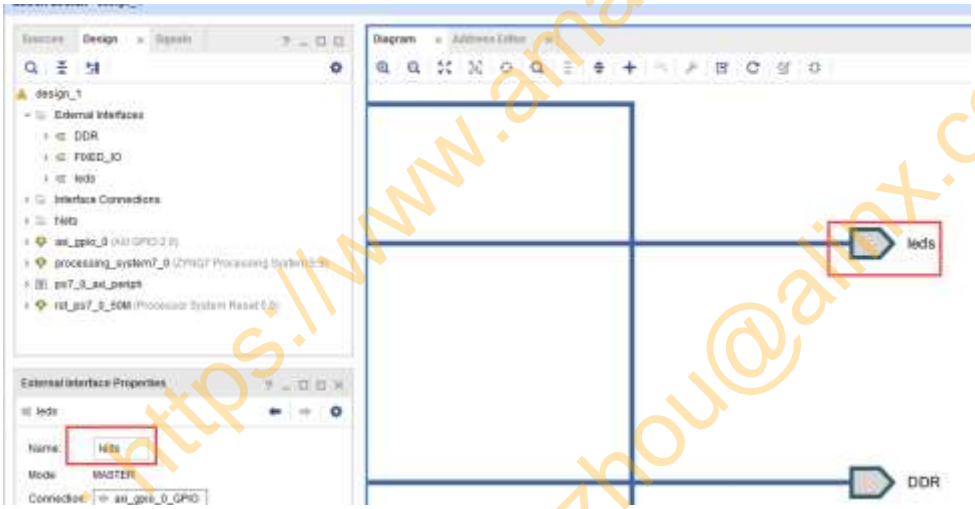
13) Click "Optimize Routing", you can optimize the layout



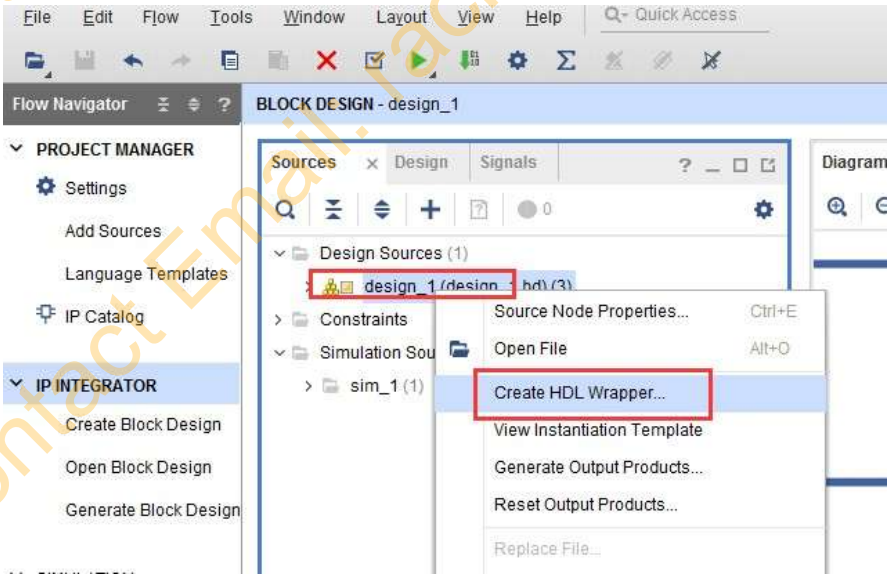
14) Modify the name of the GPIO port



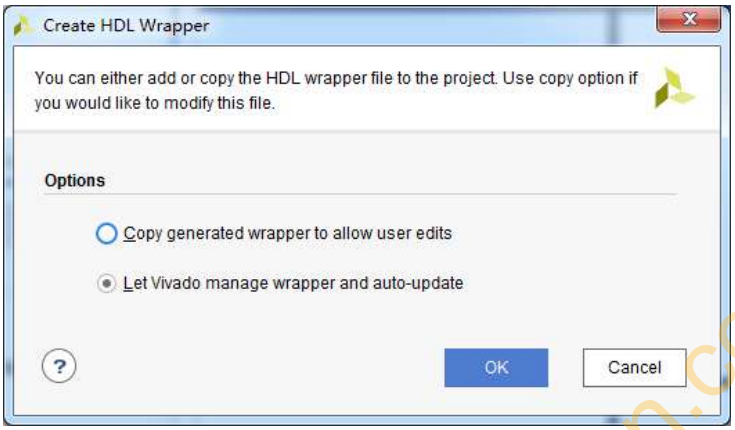
15) Change the name to leds



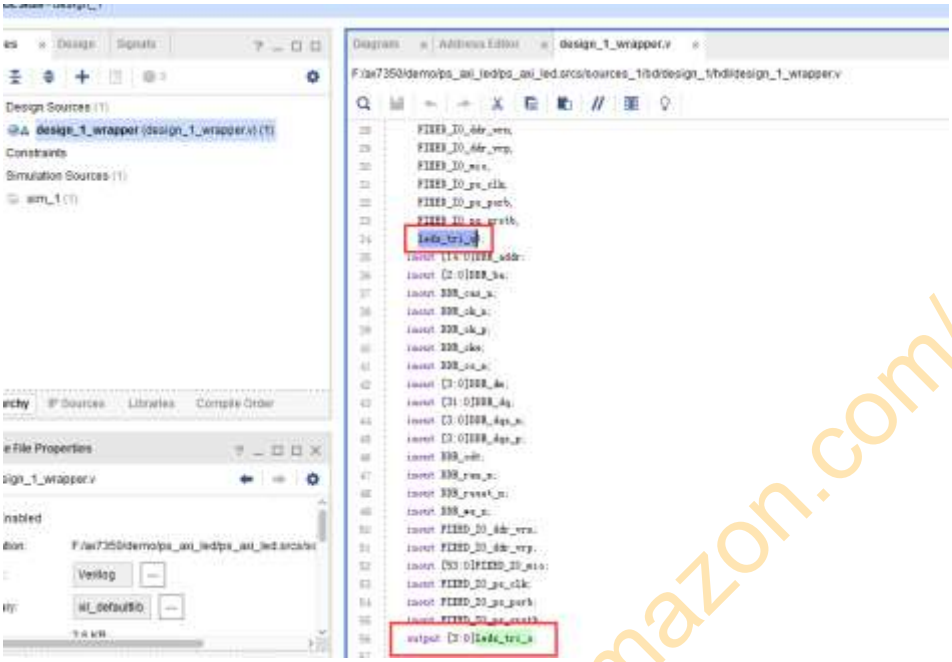
16) Create an HDL file



17) Click "OK"

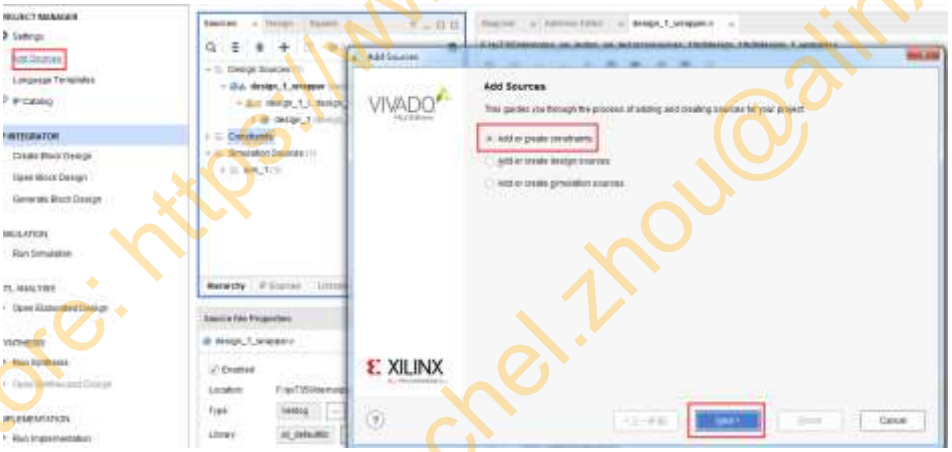


18) In the generated Verilog file, you can see that there is a "leds_tri_o" output port, you need to assign pins to them.

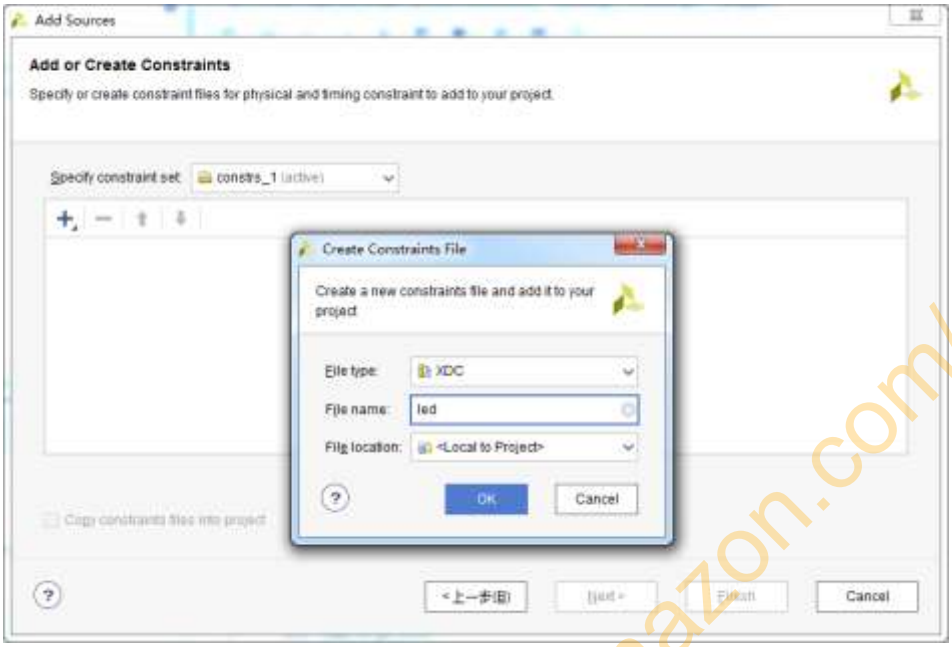


Part 7.2: XDC file constraint PL pin

1) Create a new “xdc” constraint file



2) The file name is led

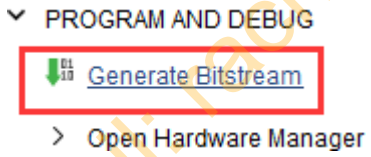


3) Led.xdc add the content, the port name must be consistent with the top file port

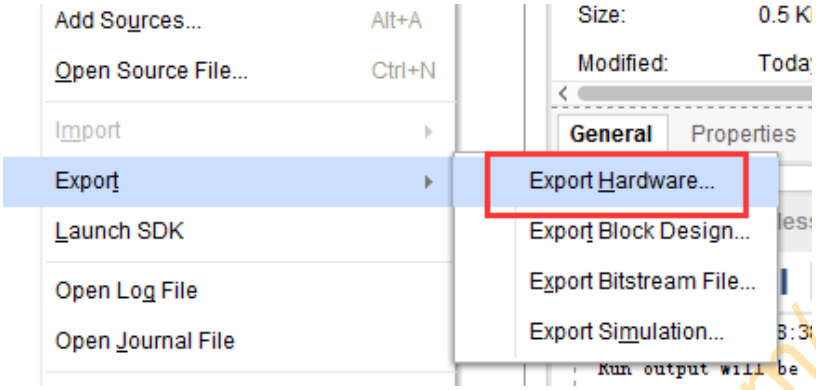
```
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[0]}]
set_property PACKAGE_PIN M14 [get_ports {leds_tri_o[0]}]
set_property PACKAGE_PIN M15 [get_ports {leds_tri_o[1]}]
set_property PACKAGE_PIN K16 [get_ports {leds_tri_o[2]}]
set_property PACKAGE_PIN J16 [get_ports {leds_tri_o[3]}]
```

Part 7.3: SDK programming

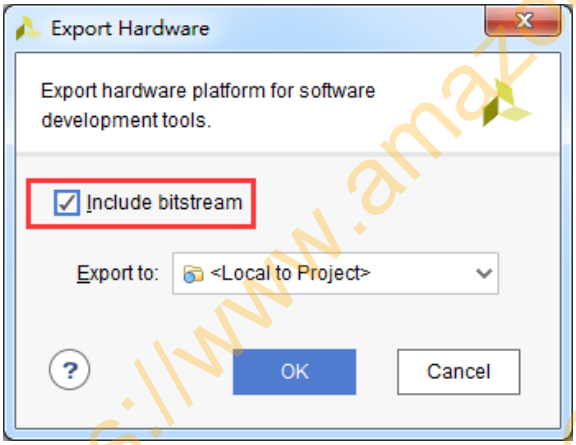
1) Generate bit file



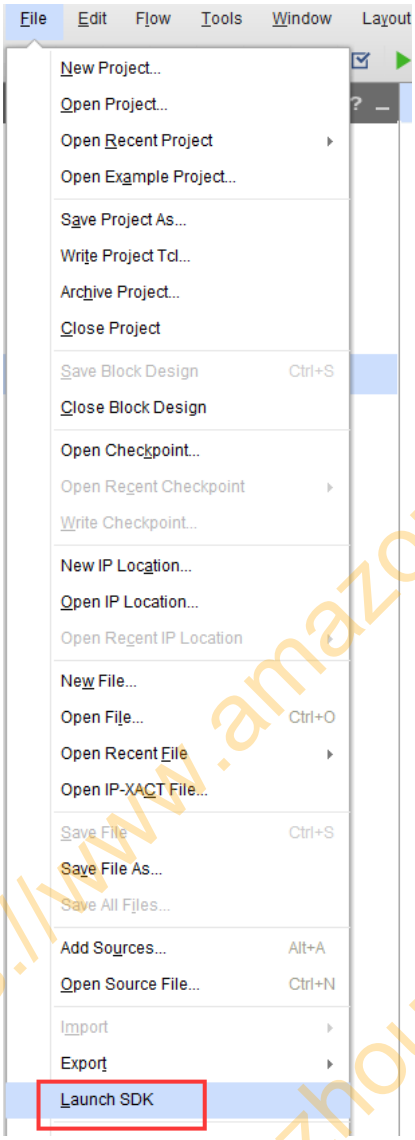
2) Export hardware



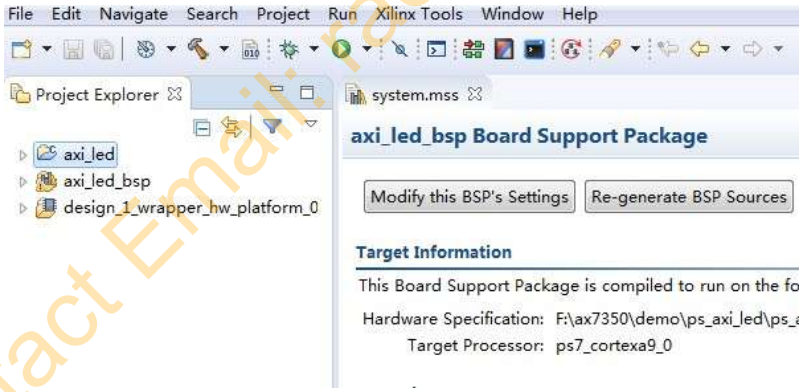
3) Since PL is used, select "Include bitstream" and click "OK"



4) Run the SDK



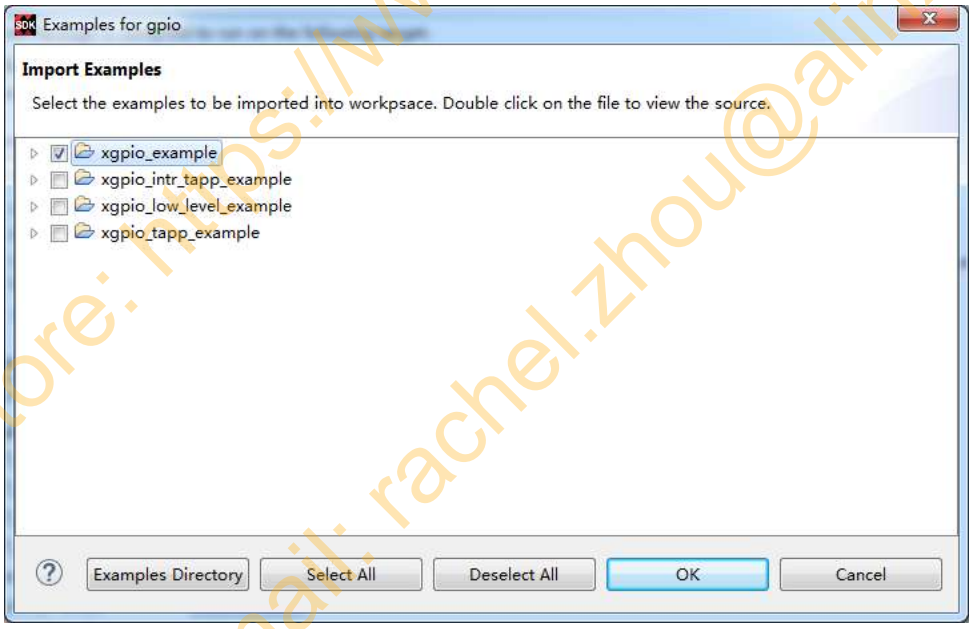
5) Create an app called "axi_led", Project template selection "Hello World"



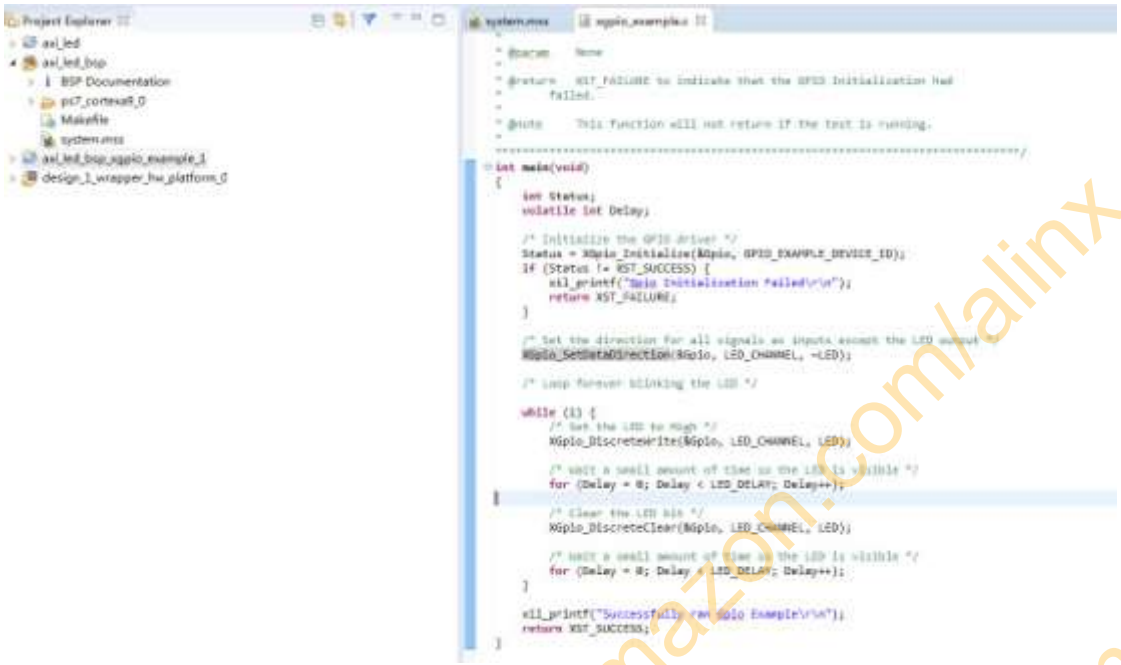
6) How do we control an unfamiliar AXI GPIO? Let's try the routines that come with the SDK.

7) Double-click "system.mss" and find "axi_gpio_0". Here you can click on "Documentation" to view related documents. Do not demonstrate here, click on "Import Examples"

8) There are multiple routines in the pop-up dialog box, you can guess from the name, here select the first "xgpio_example"



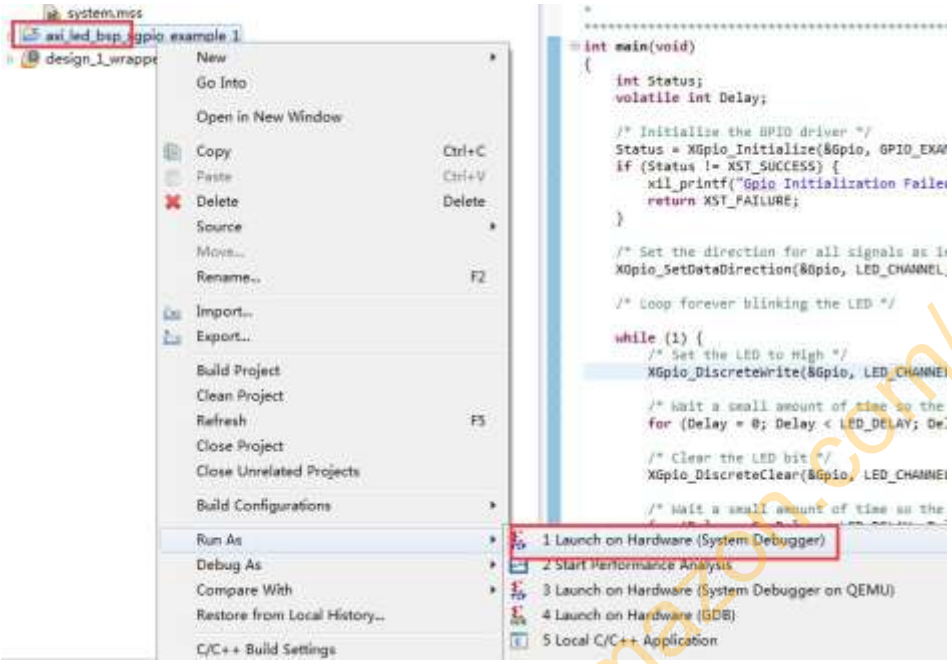
9) You can see that the routine is relatively simple, just a few lines of code, complete the operation of AXI GPIO



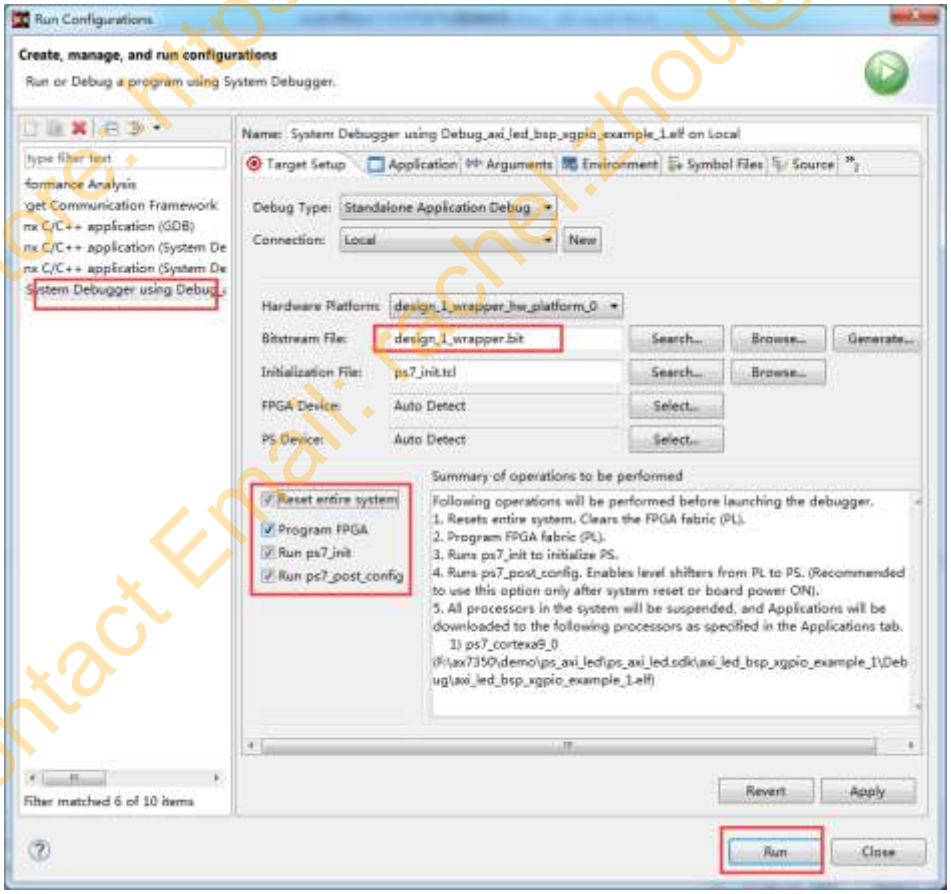
It uses a lot of GPIO related API functions. You can find out the details through the documentation. You can also select this function and press "F3" to view the specific definition. If you have this information, you still can't understand how to use AXI GPIO, indicating that you need to improve your C Language foundation.

Part 7.4: Download debugging

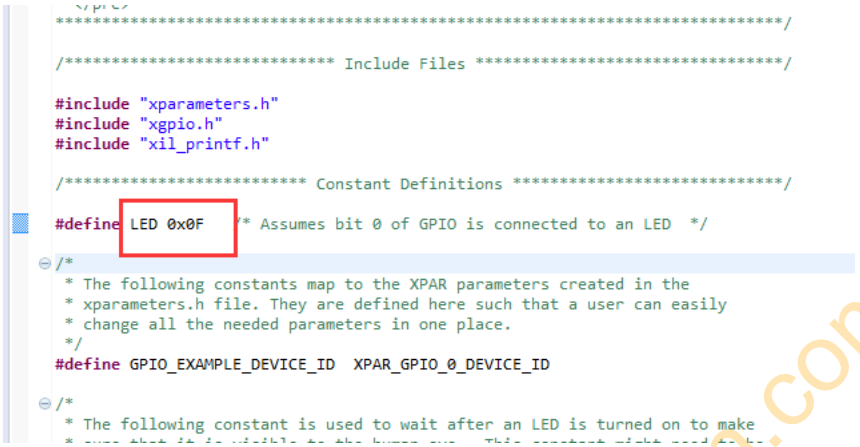
1) Although the SDK can provide some routines, some of the routines need to be modified by themselves. This simple LED routine is not modified. Try to run it and find that it can't achieve the expected results, and even prompt some errors.



2) As mentioned in the previous tutorial, "Run As" is best to reset the system. The design of PL is "Program FPGA". If your PL is modified many times, don't forget to re-export the hardware. Run again after configuring as shown below, you can see that the development board LED1 flashes quickly.



3) Modify the code to make 2 LEDs flash



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

#include "xparameters.h"
#include "xgpio.h"
#include "xil_printf.h"

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

#define LED 0x0F   /* Assumes bit 0 of GPIO is connected to an LED  */

/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are defined here such that a user can easily
 * change all the needed parameters in one place.
 */
#define GPIO_EXAMPLE_DEVICE_ID  XPAR_GPIO_0_DEVICE_ID

/*
 * The following constant is used to wait after an LED is turned on to make
```

Part 7.5: Experimental summary

Through experiments, we learned that PS can control PL through AXI bus, but it does not reflect the advantages of ZYNQ, because it can be easily done for controlling LED lights, whether ARM or FPGA, but if you replace LED with serial port, control 100 channels of serial communication, 8 channels of Ethernet and other applications, I think there is no SOC can complete this function, only ZYNQ can, this is the difference between ZYNQ and ordinary SOC.

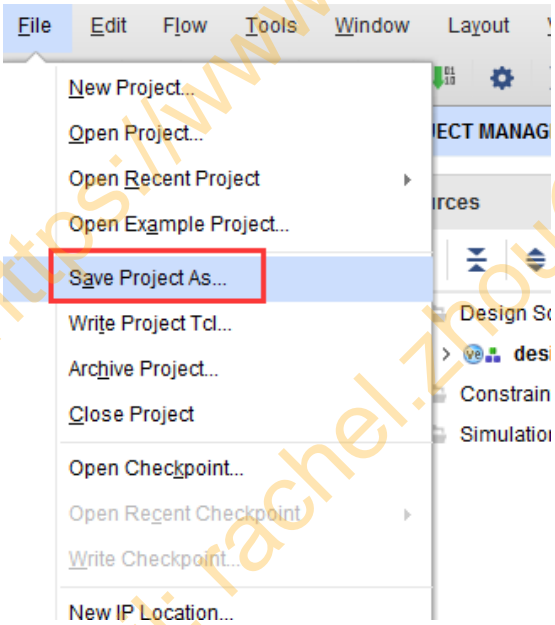
Part 8: PS timer interrupt experiment

Experiment Vivado project is "ps_timer"

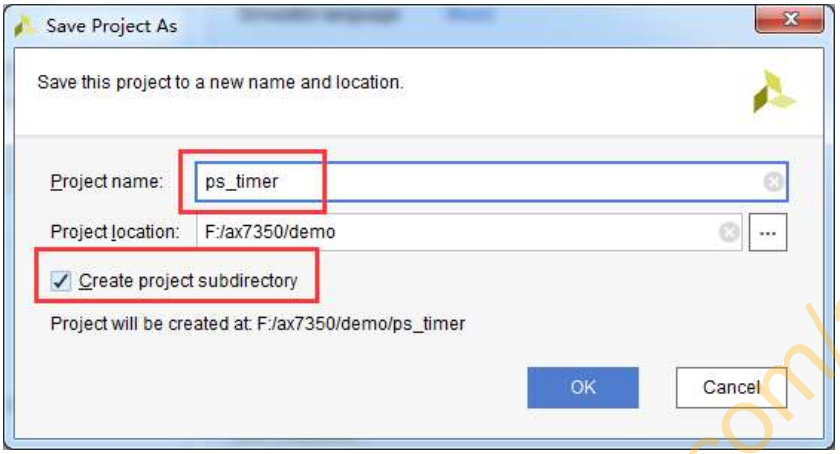
Many SOCs have timers inside, and ZYNQ's PS also has. What are the peripherals in ZYNQ? What are the characteristics of these? These are all developers must care about, so it is recommended to read the xilinx documentation UG585 frequently.

Part 8.1: Building a Vivado project

1) Repeatedly established Vivado project, which has a lot of repetitive work, the easiest solution is to copy the project, and then modify it, open the project "ps_hello"
2) Click on the menu "File -> Save Project As..."

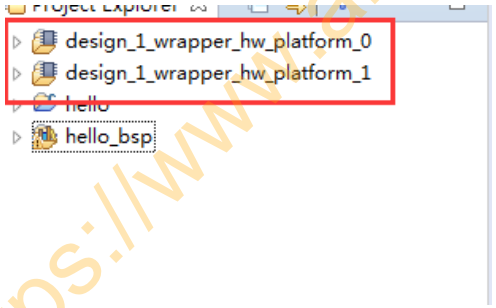


3) Fill in the new project name "ps_timer" in the pop-up dialog box, select the creation of the project subdirectory, the timer in the PS. Since the pin output is not needed, there is no need to configure the pin.

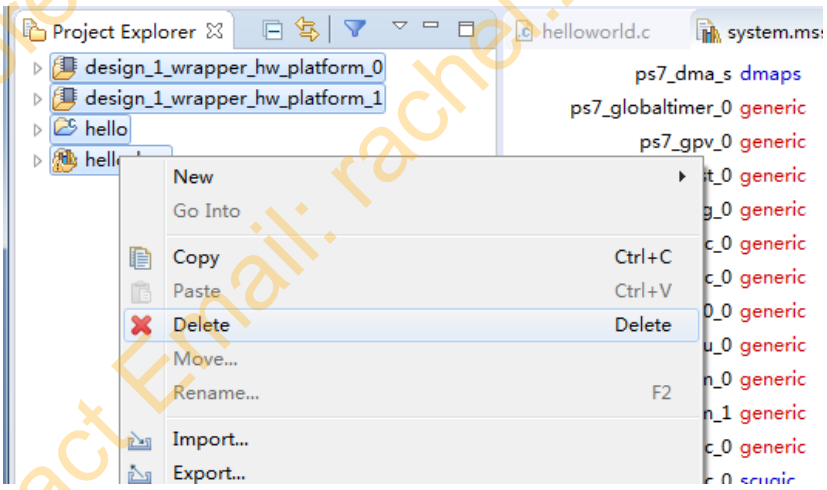


Part 8.2: SDK programming

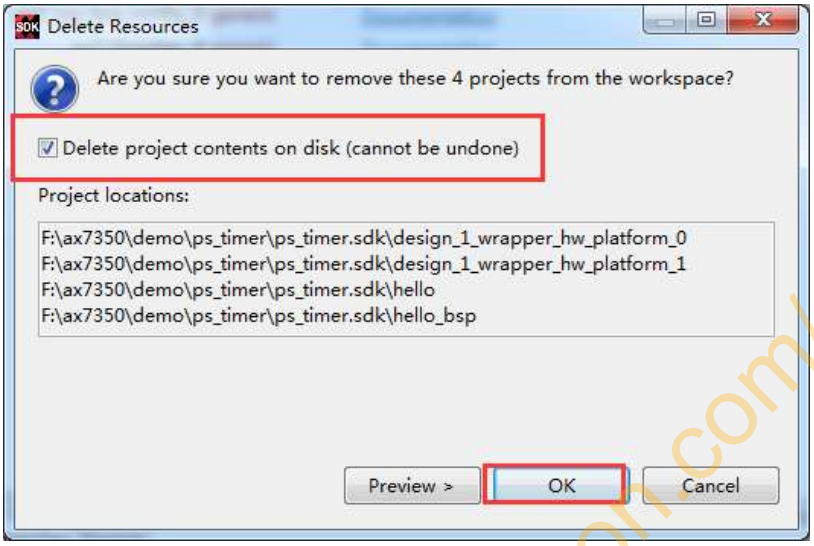
1) Run the SDK, you can see that, unlike the previous routine, there is another hardware platform information folder.



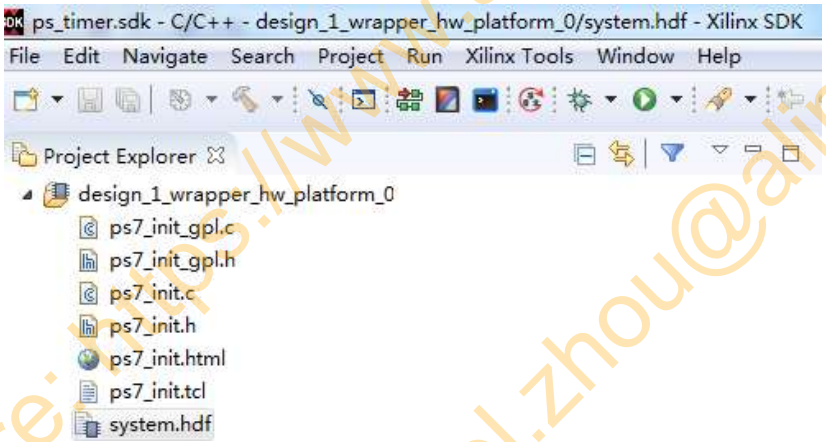
2) There will be a similar phenomenon when using someone else's SDK project, here we are all going to delete



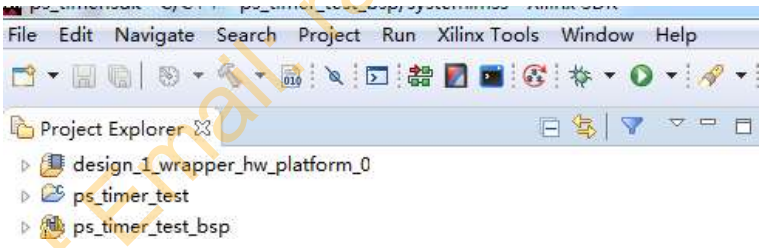
3) File is also deleted



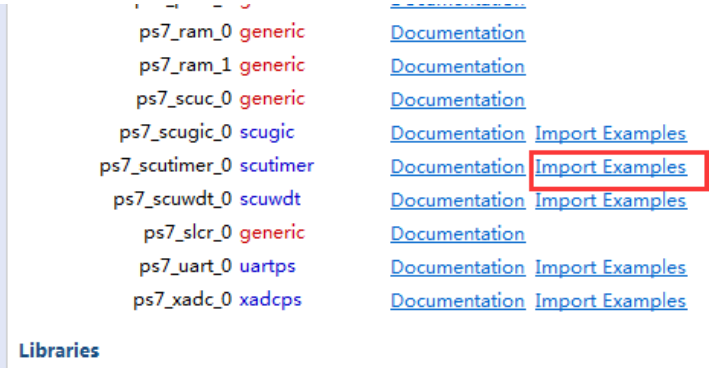
4) Re-run the SDK in Vivado and you will see a new hardware platform information.



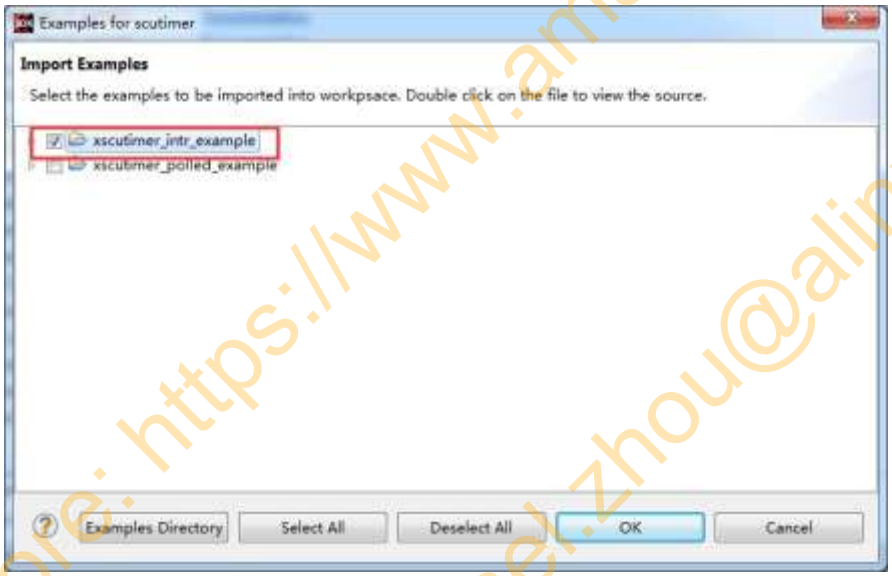
5) Re-establish a project, the name is called "ps_timer_test", the template is still “Hello World”



6) Now it's time to write the code. If you don't know how to use the timer and don't know how to use the interrupt, then use the old method to see the example.



7) Fortunately, there is an example of a timer interrupt. How do you know that this example is an example of an interrupt? It is guessed by "intr", so the basics are important, otherwise you won't even find the routine.



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

8) This experiment is designed to interrupt the timer for 1 second, then print out the information, and end after 30 seconds. First, modify the maximum value of the counter and change it to half of the CPU frequency, which is the clock frequency value of the counter, so that it will be interrupted once every 1 second.

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

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

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

/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are only defined here such that a user can easily
 * change all the needed parameters in one place.
 */
#ifndef TESTAPP_GEN
#define TIMER_DEVICE_ID		XPAR_XSCUTIMER_0_DEVICE_ID
#define INTC_DEVICE_ID		XPAR_SCUGIC_SINGLE_DEVICE_ID
#define TIMER_IRPT_INTR		XPAR_SCUTIMER_INTR
#endif

#define TIMER_LOAD_VALUE	(XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ/2 - 1)

/**************************** Type Definitions *****************************/

/***************** Macros (Inline Functions) Definitions ******************/

/************************** Function Prototypes ***************************/
```

9) Change the number of counts 3 to 30

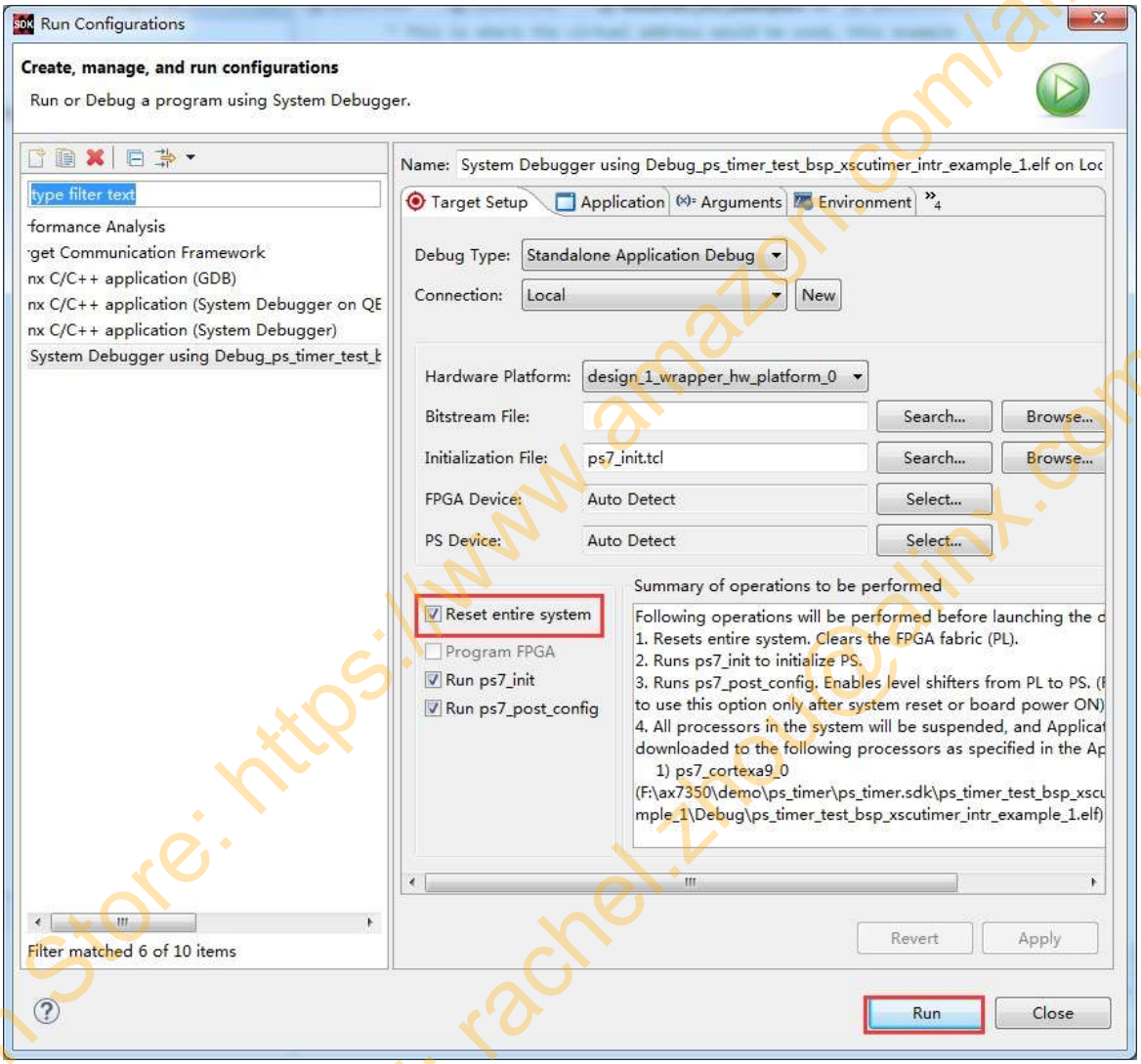
```
        LastTimerExpired = TimerExpired;
        /*
         * If it has expired a number of times, then stop the timer
         * counter and stop this example.
         */
        if (TimerExpired == 30) {
            XScuTimer_Stop(TimerInstancePtr);
            break;
        }
    }
    /*
     * Disable and disconnect the interrupt system.
     */
    TimerDisableIntrSystem(IntcInstancePtr, TimerIntrId);
```

10)Add print information

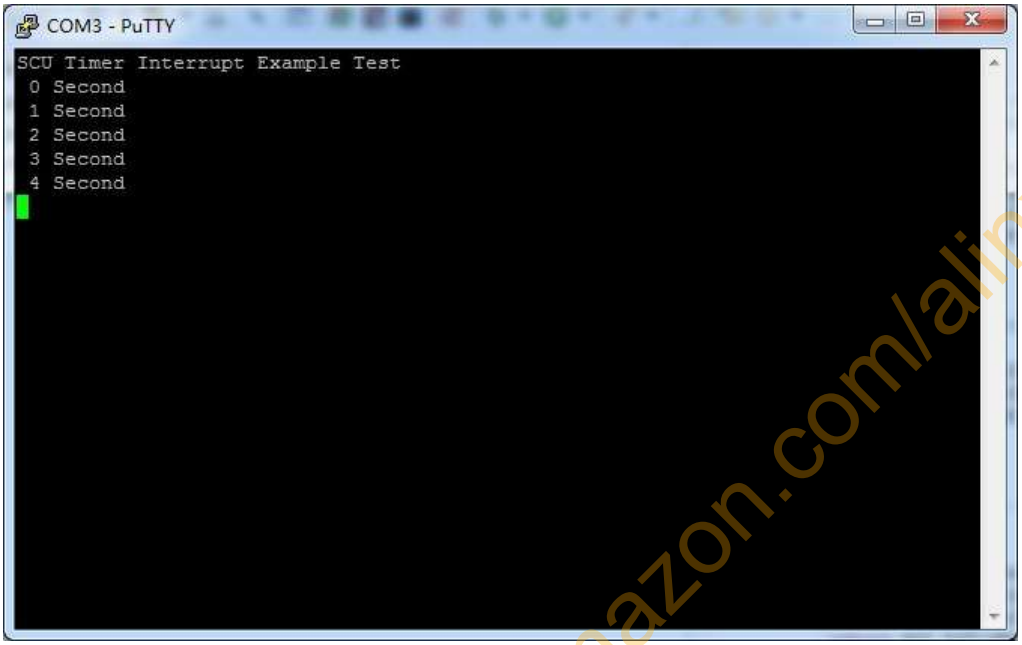
```
/*****************************************************************************/
static void TimerIntrHandler(void *CallBackRef)
{
    XScuTimer *TimerInstancePtr = (XScuTimer *) CallBackRef;

    /*
     * Check if the timer counter has expired, checking is not necessary
     * since that's the reason this function is executed, this just shows
     * how the callback reference can be used as a pointer to the instance
     * of the timer counter that expired, increment a shared variable so
     * the main thread of execution can see the timer expired.
     */
    if (XScuTimer_IsExpired(TimerInstancePtr)) {
        XScuTimer_ClearInterruptStatus(TimerInstancePtr);
        printf(" %d Second\n\r",TimerExpired);
        TimerExpired++;
        if (TimerExpired == 30) {
            XScuTimer_DisableAutoReload(TimerInstancePtr);
        }
    }
}
```

Part 8.3: Download debugging

1) Open PuTTY 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 output a message every second.



Part 8.4: Experimental Summary

In the experiment, the timer and interrupt application was completed by simply modifying the SDK routines. The simple operation that contains a wealth of knowledge, you need to understand the principle of the timer and the principle of the interrupt. This basic knowledge is a necessary condition to learn the ZYNQ

Part 9: PL key interrupt experiment

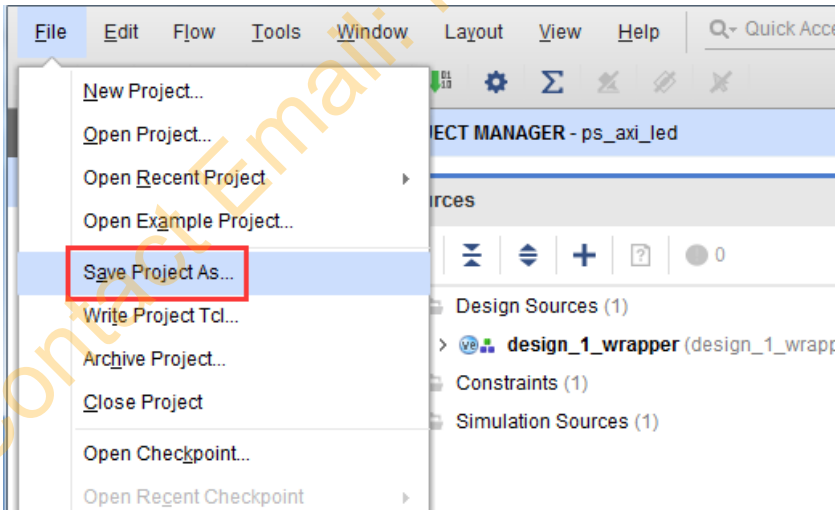
Experiment Vivado project is " ps_axi_key "

The interrupt of the previous timer interrupt experiment belongs to the internal interrupt of PS. The interrupt of this experiment comes from PL. PS can receive 16 interrupt signals from PL at the maximum, all of which are rising edge or high level trigger. This experiment uses a button interrupt to control the LED.

Source	Interrupt Name	IRQ ID#	Status Bits (mpcore Registers)	Required Type	PS-PL Signal Name	I/O
PL	PL [2:0]	63:61	spi_status_0[31:29]	Rising edge/ High level	IRQF2P[2:0]	Input
	PL [7:3]	68:64	spi_status_1[4:0]	Rising edge/ High level	IRQF2P[7:3]	Input
Timer	TTC 1	71:69	spi_status_1[7:5]	High level	-	-
DMAC	DMAC[7:4]	75:72	spi_status_1[11:8]	High level	IRQP2F[27:24]	Output
IOP	USB 1	76	spi_status_1[12]	High level	IRQP2F[7]	Output
	Ethernet 1	77	spi_status_1[13]	High level	IRQP2F[6]	Output
	Ethernet 1 Wake-up	78	spi_status_1[14]	Rising edge	IRQP2F[5]	Output
	SDIO 1	79	spi_status_1[15]	High level	IRQP2F[4]	Output
	I2C 1	80	spi_status_1[16]	High level	IRQP2F[3]	Output
	SPI 1	81	spi_status_1[17]	High level	IRQP2F[2]	Output
	UART 1	82	spi_status_1[18]	High level	IRQP2F[1]	Output
	CAN 1	83	spi_status_1[19]	High level	IRQP2F[0]	Output
PL	PL [15:8]	91:84	spi_status_1[27:20]	Rising edge/ High level	IRQF2P[15:8]	Input
SCU	Parity	92	spi_status_1[28]	Rising edge	-	-
Reserved	-	95:93	spi_status_1[31:29]	-	-	-

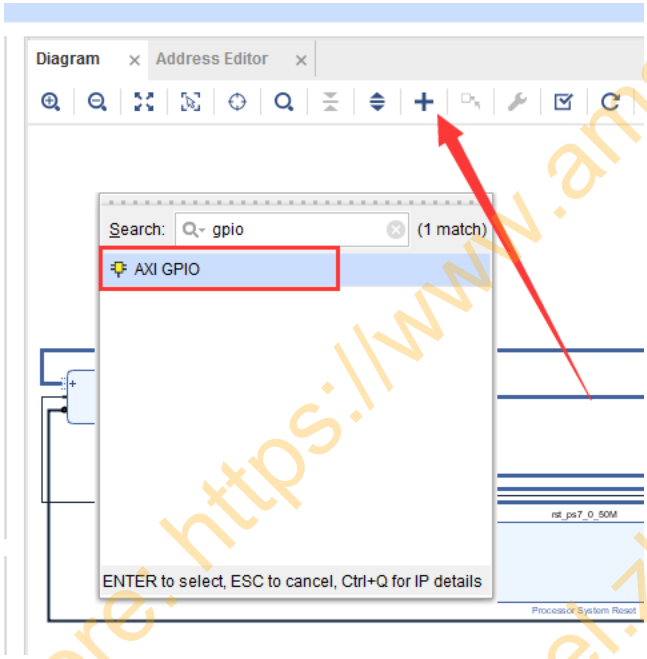
Part 9.1: Building a Vivado project

1) The Vivado project used in this experiment only needs to add AXI GPIO for key input in the project "ps_axi_led". Click the menu "File->SaveProjectAs..."

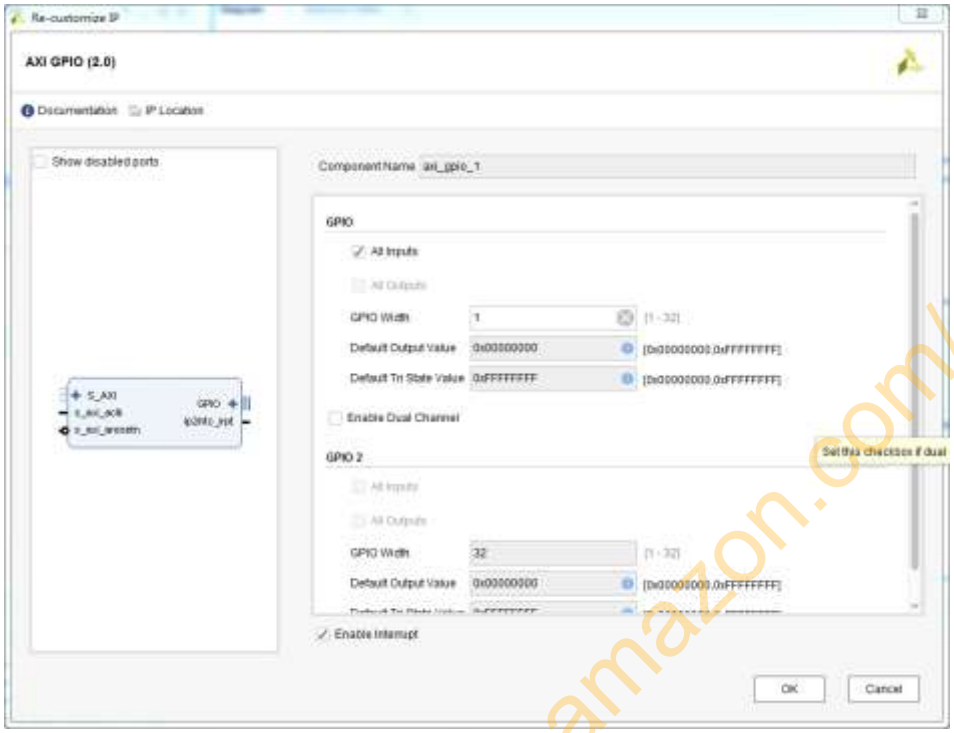


2) The new project is named "ps_axi_key"

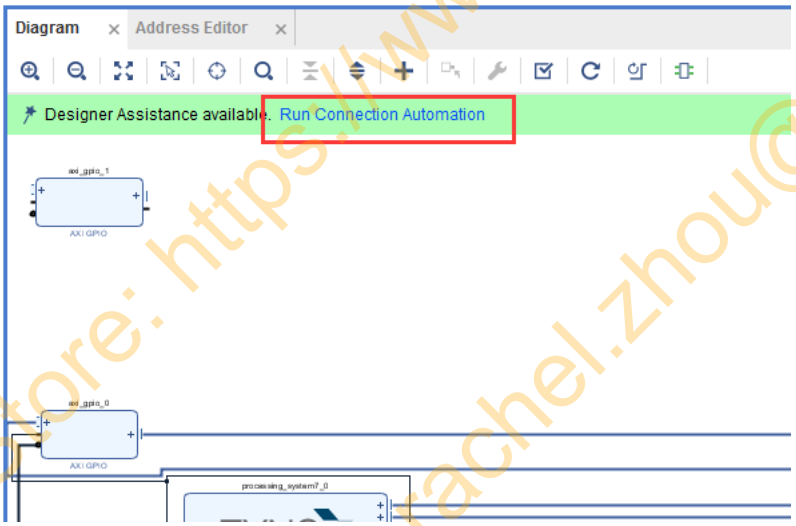
3) Add an AXIGPIO



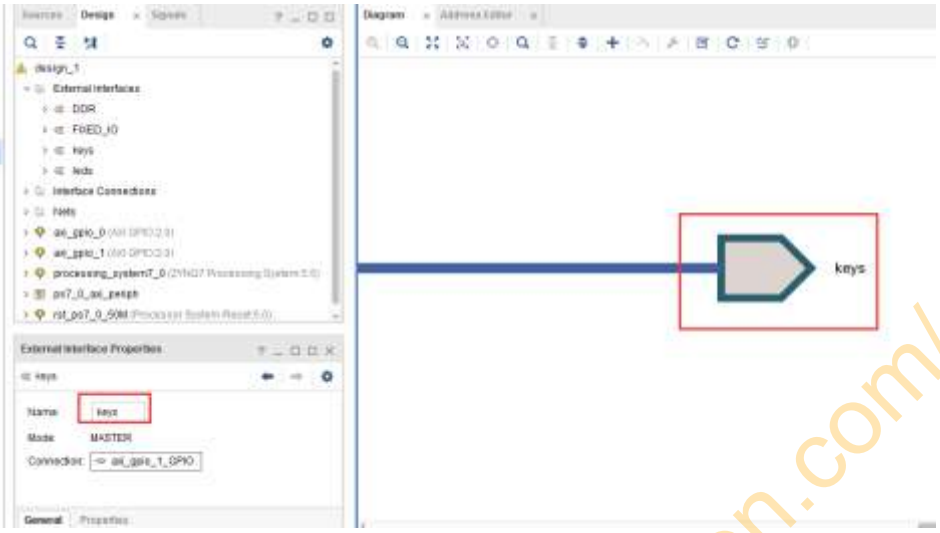
4) Configure GPIO parameters, both input, width 1, enable interrupt



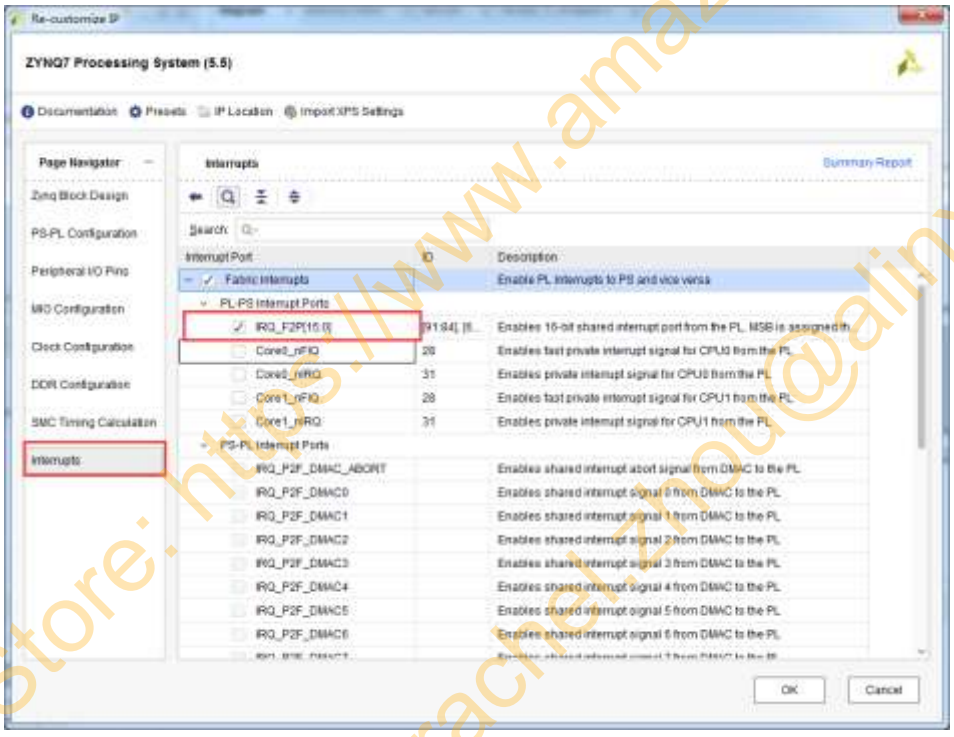
5) Use automatic connection



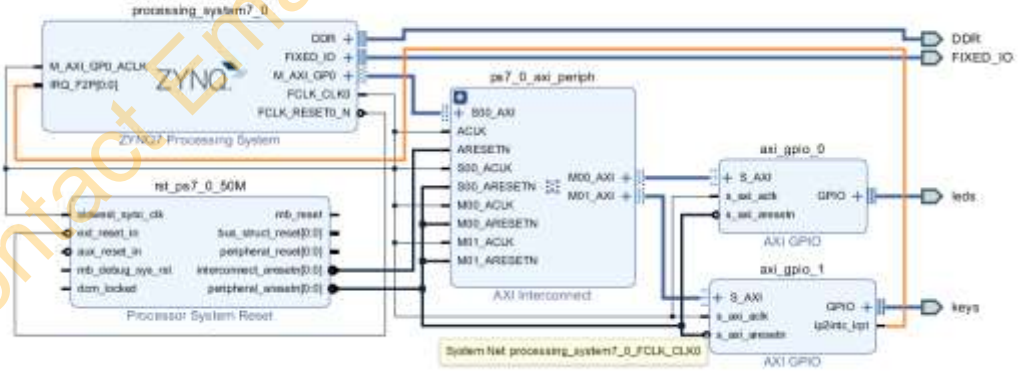
6) Change the port name to keys



7) Configure the interrupt of the ZYNQ processor, check “IRQ_F2P”



8) Connect “ip2intc_irpt” to “IRQ_F2Q”



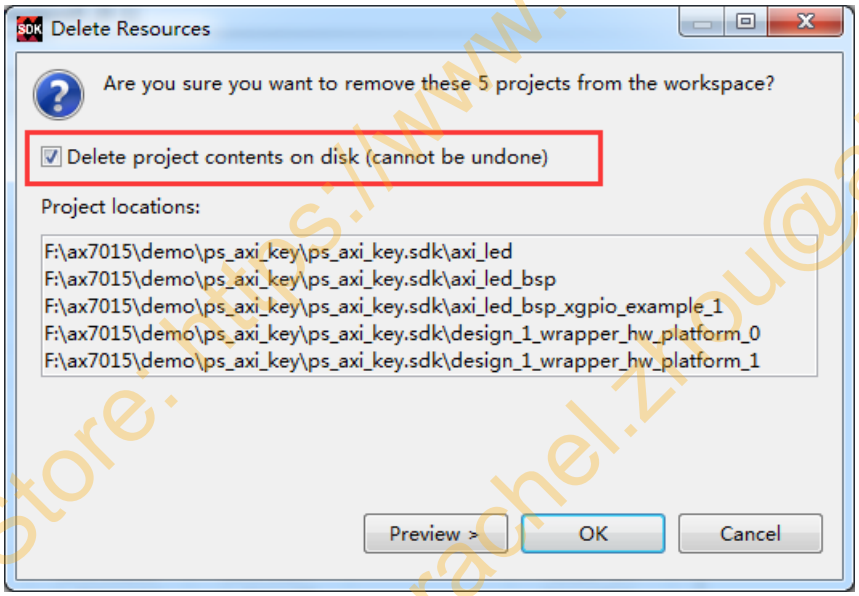
9) Modify the xdc constraint file

```
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_tri_o[0]}]
set_property PACKAGE_PIN M14 [get_ports {leds_tri_o[0]}]
set_property PACKAGE_PIN M15 [get_ports {leds_tri_o[1]}]
set_property PACKAGE_PIN K16 [get_ports {leds_tri_o[2]}]
set_property PACKAGE_PIN J16 [get_ports {leds_tri_o[3]}]

set_property IOSTANDARD LVCMOS33 [get_ports {keys_tri_i[0]}]
set_property PACKAGE_PIN N15 [get_ports {keys_tri_i[0]}]
```

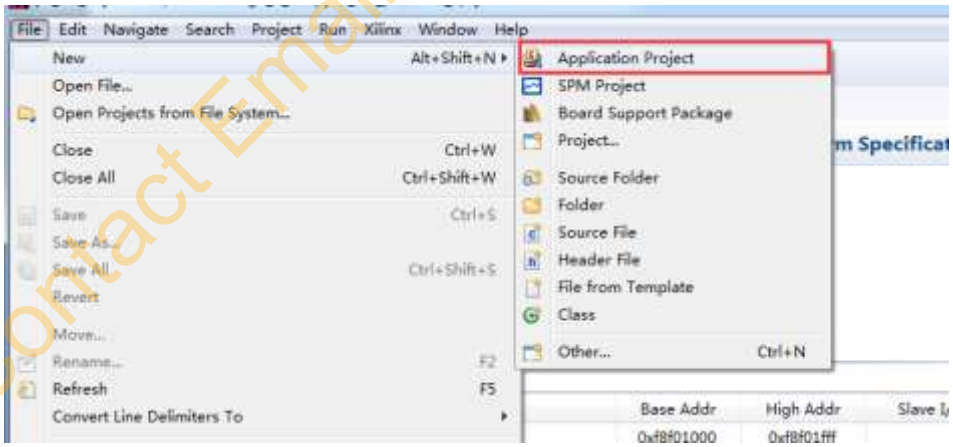
10) Save the design, compile and generate the bit file

Part 9.2: Download debugging

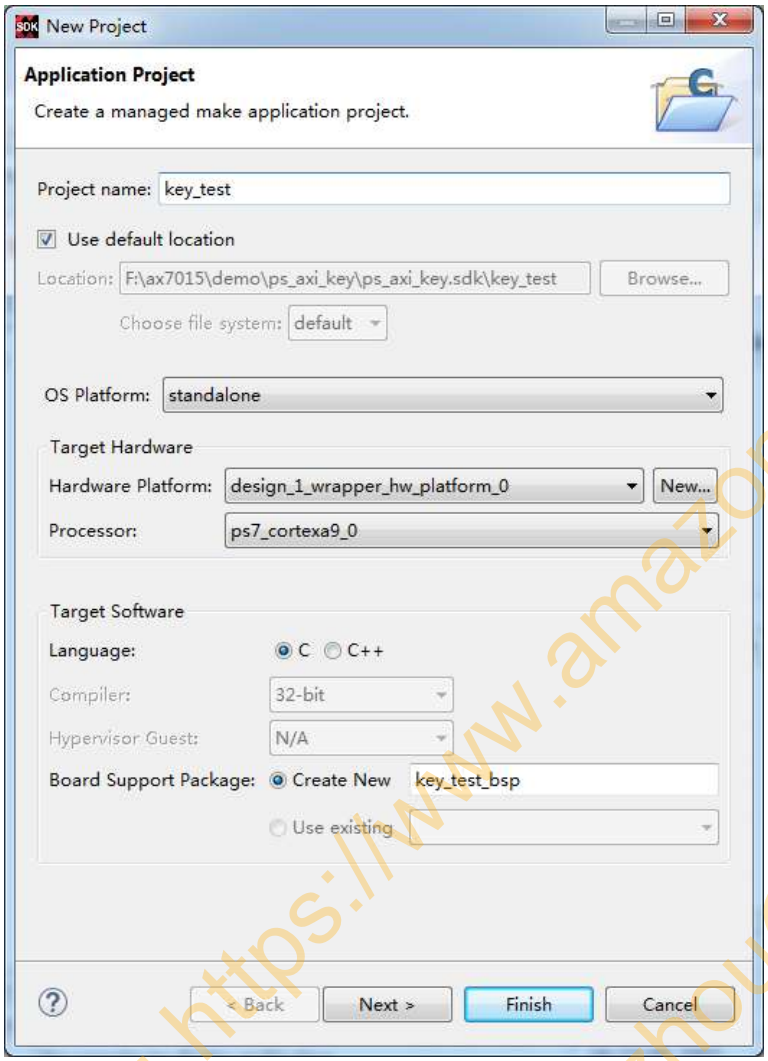
1) Run the SDK, because it is copied from other projects, there are a lot of files we don't need under the SDK, delete them all, then close the SDK and run again.



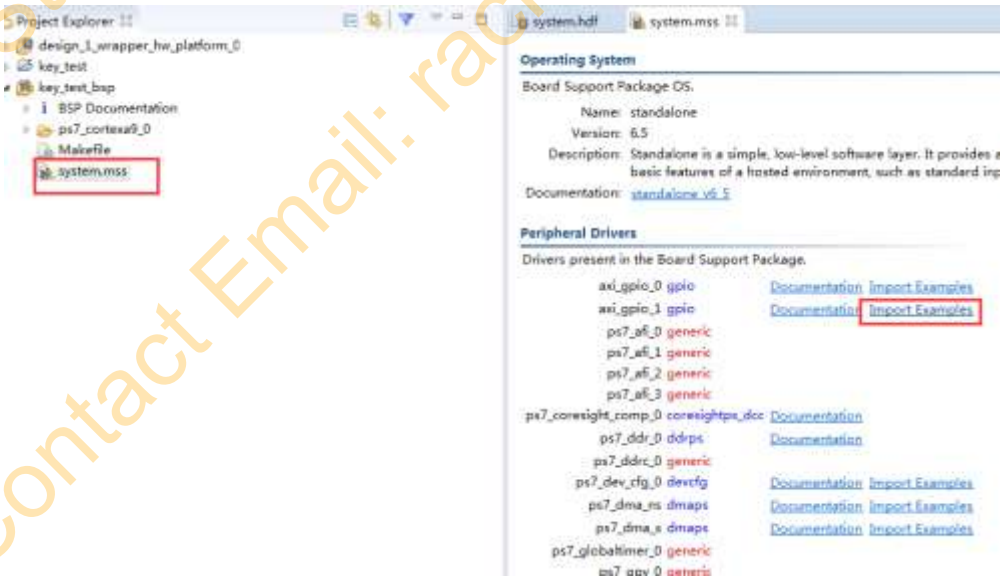
2) Create a new APP



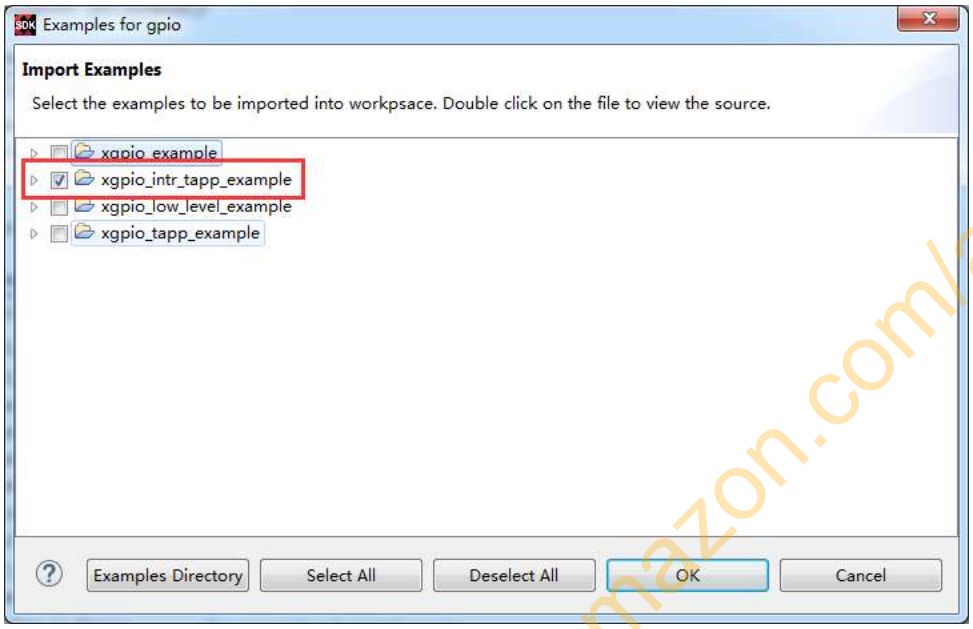
3) Set Project name to "key_test"



4) As in the previous tutorial, when you are not familiar with the SDK program, try to use the SDK's own routines to modify



5) Select "xgpio_intr_tapp_example"



6) There are undefined errors after the import routine, we need to modify some of the code

7) Modify the macro definition of GPIO and interrupt number as shown below

system.mss xgpio_intr_tapp_example.c

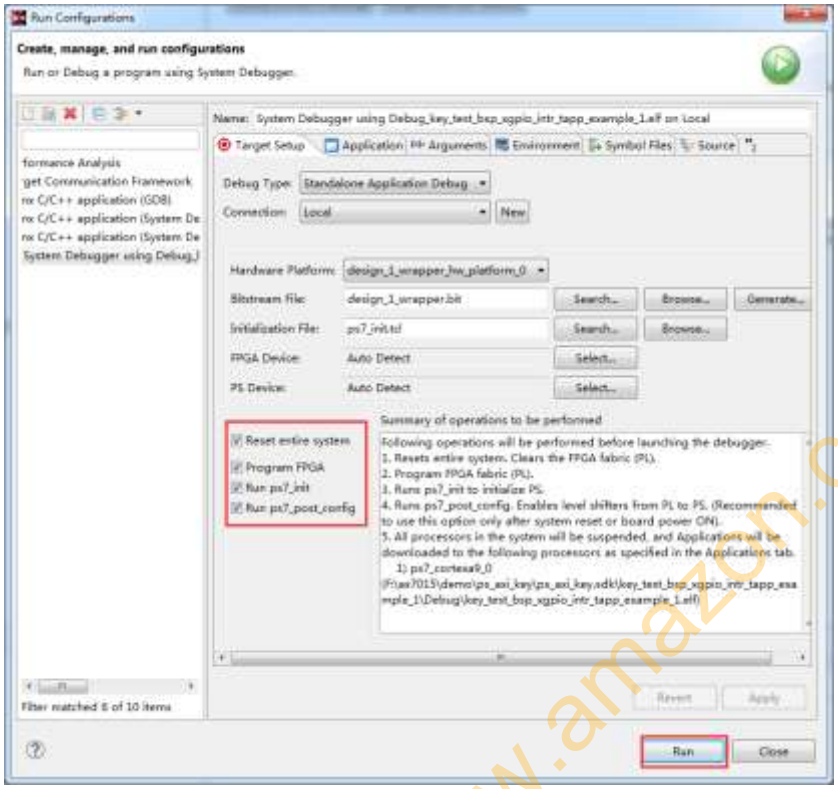
```
#include <stdio.h>
#else
 #include "xscugic.h"
 #include "xil_printf.h"
#endif

/************************** Constant Definitions *************************/
#ifndef TESTAPP_GEN
/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are defined here such that a user can easily
 * change all the needed parameters in one place.
 */
#define GPIO_DEVICE_ID      XPAR_GPIO_1_DEVICE_ID
#define GPIO_CHANNEL1       1

#ifdef 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 */

/*
 * The following constants define the positions of the buttons and LEDs each
 * channel of the GPIO
 */
#define GPIO_ALL_LEDS       0xFFFF
#define GPIO_ALL_BUTTONS    0xFFFF
```

8) Modify the test delay time so that we have enough time to press the key

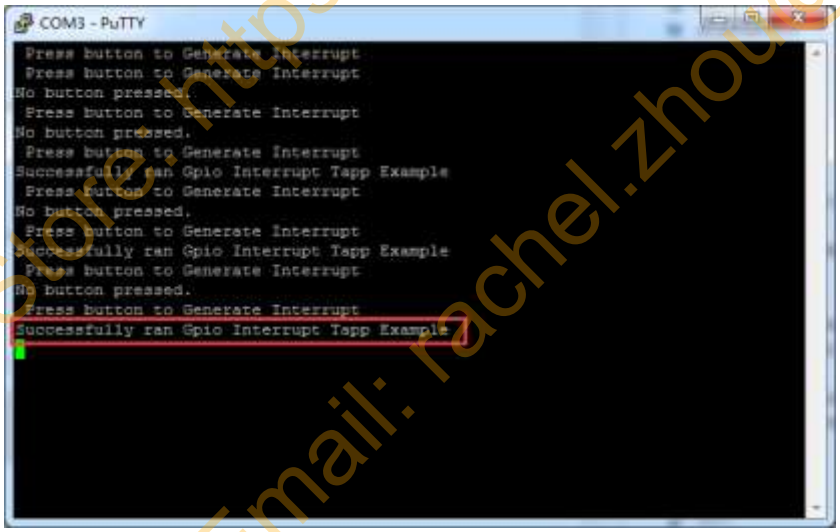
system.mss xgpio_intr_tapp_example.c

```
 * 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  0x2FFFFFFF

#ifdef 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
```

9) Open the serial terminal and run the program.



10) If you do not press the button, the serial port displays "No button pressed.", if you press "KEY2", it displays "Successfully ran Gpio Interrupt Tapp Example"



Part 9.3: Experimental summary

The PL side can send an interrupt signal to the PS, which improves the efficiency of PL and PS data interaction. Interrupt processing is required in application that required large amounts of low latency.

Part 10: Ethernet Experiment (LWIP)

Experiment Vivado project is " net_test "

The development board has 1 Gigabit Ethernet and is connected through the RGMII interface. This experiment demonstrates how to use the SDK's own LWIP template for Gigabit Ethernet TCP communication.

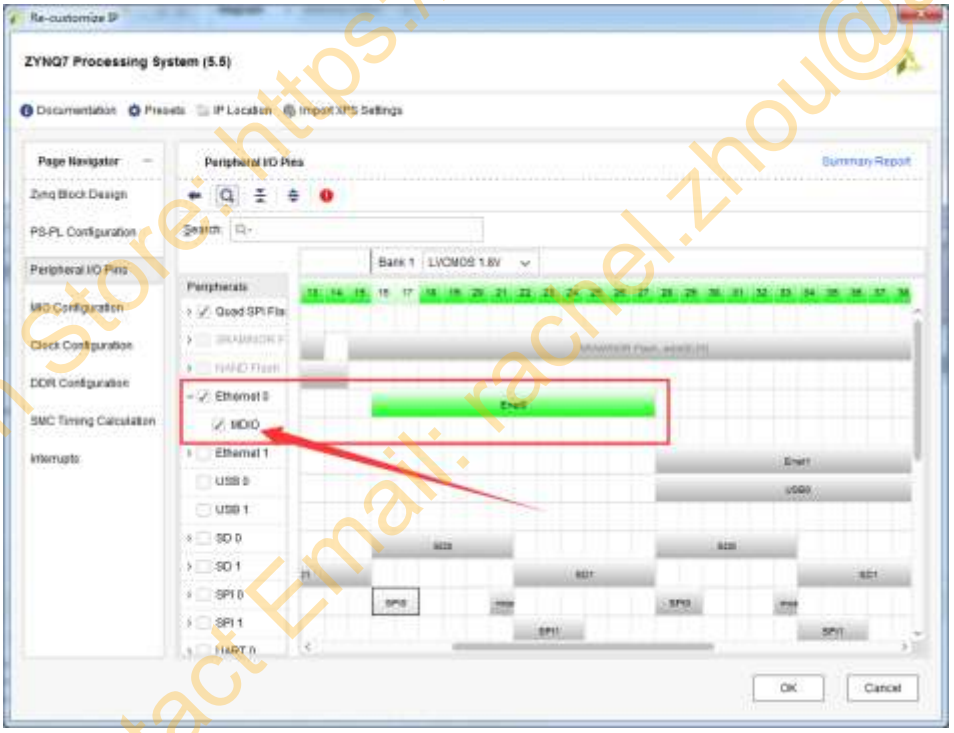
Although LWIP is a lightweight protocol stack, if you have never used it, it will be difficult to use. It is recommended to familiar with LWIP related knowledge firstly.

Part 10.1: Create a Vivado project

Create a new "net_test vivado project, add ZYNQ, configure serial port DDR3 etc. according to the previous tutorial. For detailed parameters, please refer to the vivado project in this tutorial."

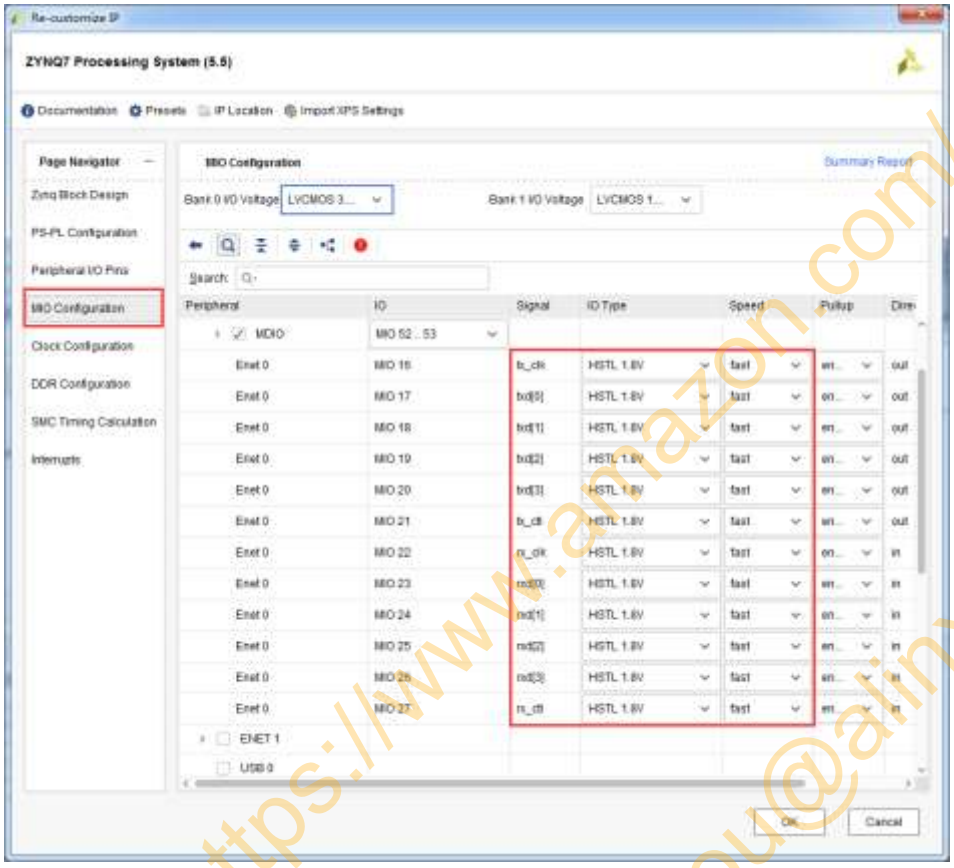
Part 10.1.1: Ethernet configuration on the PS side

1) Enable "Enet0 MIO16 MIO27 and" MDIO MIO52 MIO53

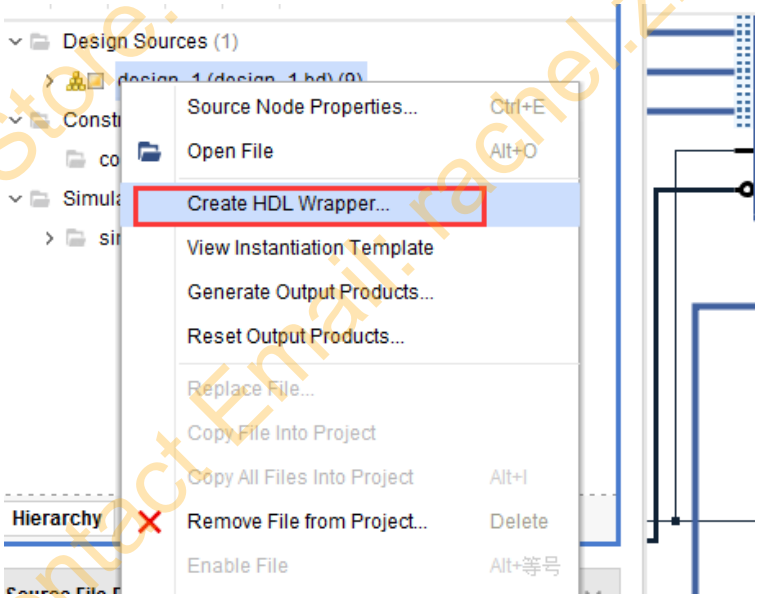


Select “MDIO”

2) Modify the level standard of Enet0 to HSTL "1.8 V" and Speed to fast. These parameters are very important. If they are not modified, the network may be blocked.



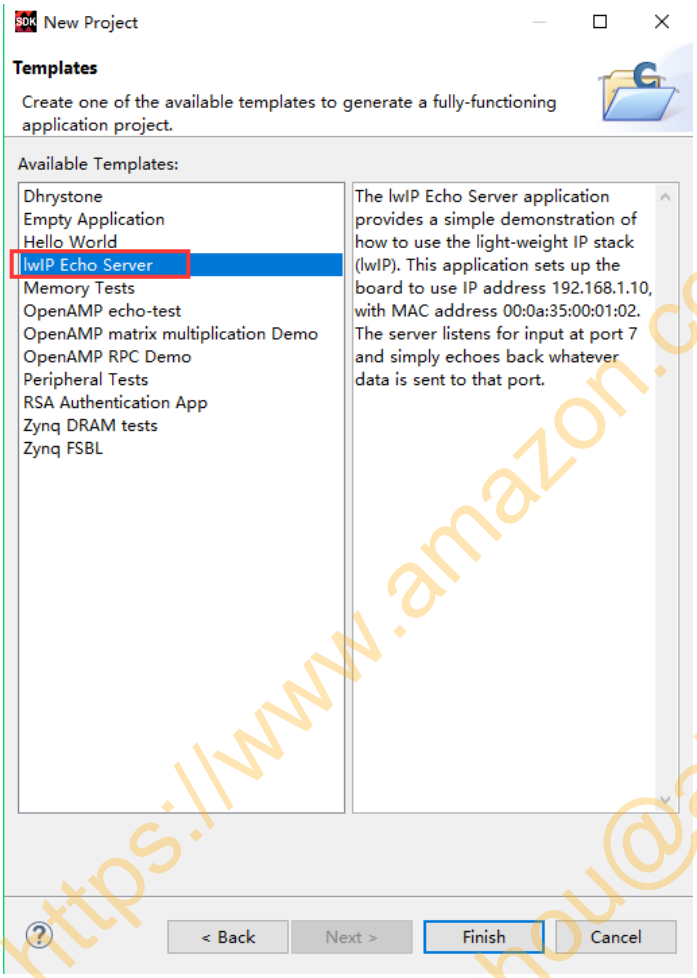
3) Create HDL file



4) Compile and generate bit file, then export hardware information, start SDK

Part 10.2: SDK Program Development

Part 10.2.1: Create an APP based on the LWIP template

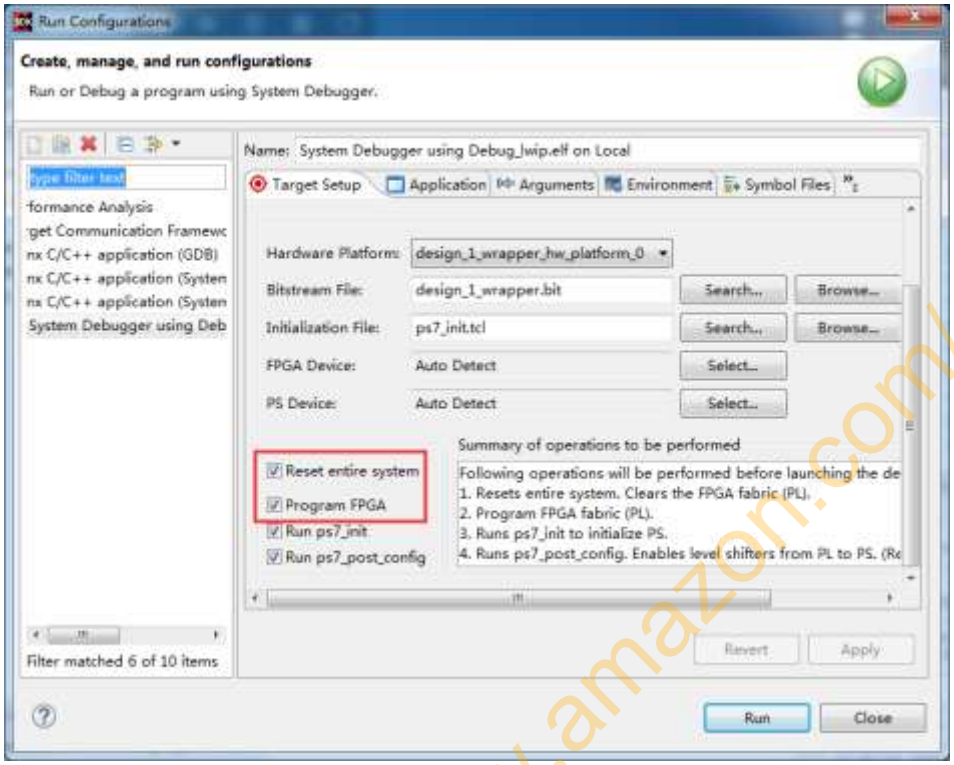


Part 10.3: Download debugging

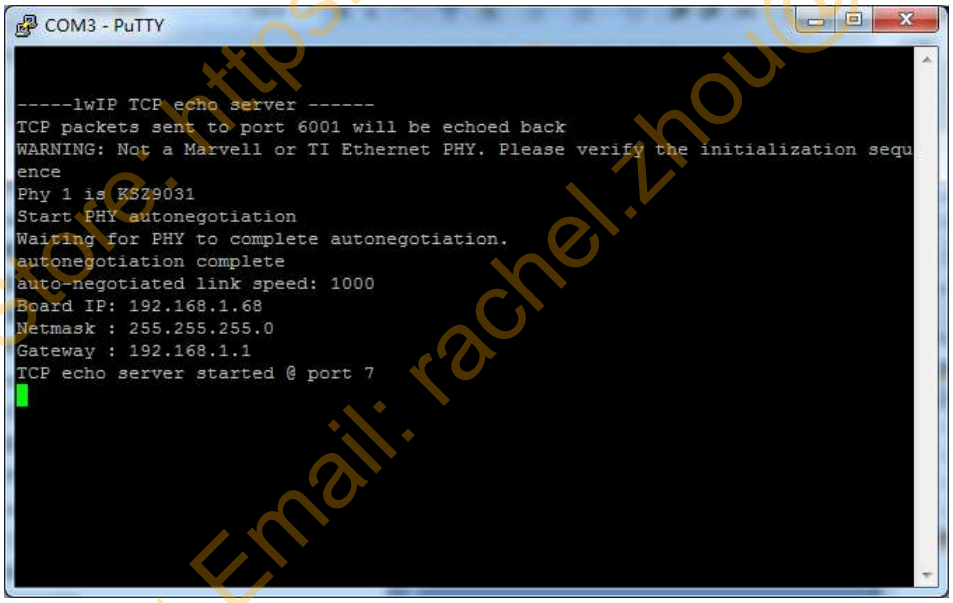
The test environment requires a router that supports dhcp. The development board connects to the router to automatically obtain the IP address. The experiment host and the FPGA development board are in a network and can communicate with each other.

Part 10.3.1: Ethernet Testing

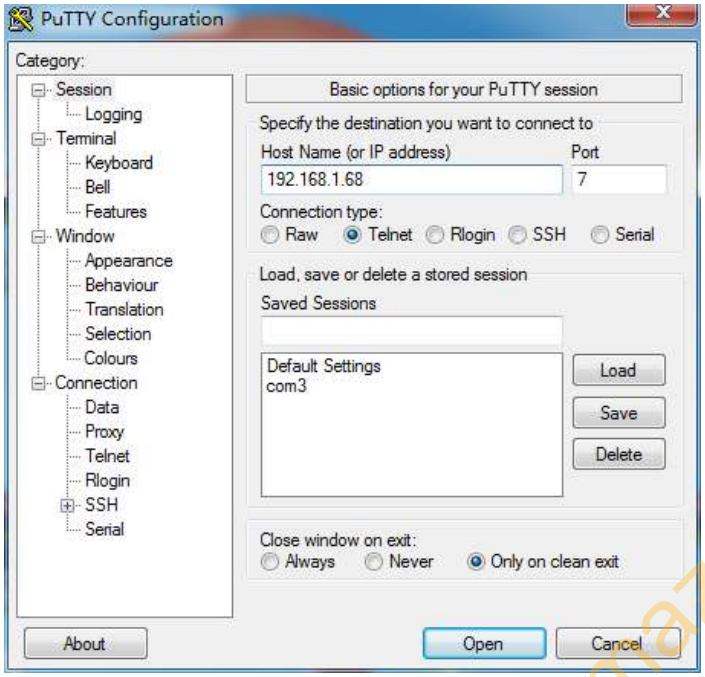
1) Connect the serial port to open the serial debugging terminal, connect the PS Ethernet cable to the router, and run the SDK.



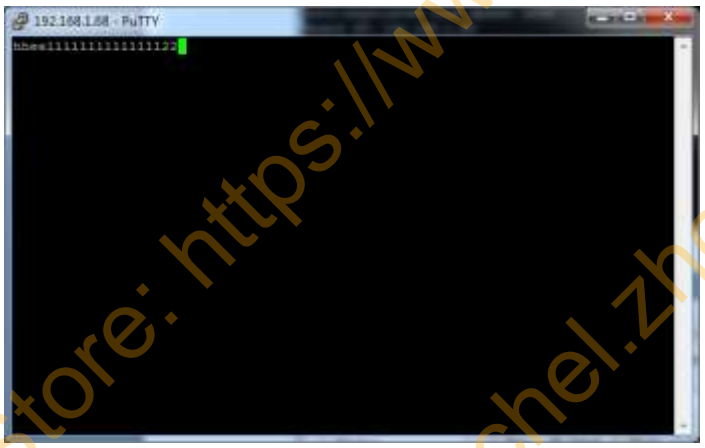
2) You can see that the serial port prints some information. You can see that the address is automatically obtained as "192.168.1.68", the connection speed is 1000Mbps, and the tcp port is 7



3) Connect using telnet



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



Part 10.4: Experimental summary

Through experiments, we have a deeper understanding of the development of the SDK program. This experiment is just a simple explanation of how to create a LWIP application. LWIP can complete protocols such as UDP and TCP. In the following tutorials, we will provide specific applications based on Ethernet, such as ADC. The collected data is sent via Ethernet, and the camera data is sent to the host computer via Ethernet.

Part 11: Custom IP experiment

Experiment Vivado project is " custom_pwm_ip "

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 experiment 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.

Part 11.1: 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:

```
//================================================================================
//  Description:  pwm model
//  pwm out period = frequency(pwm_out) * (2 ** N) / frequency(clk);
//================================================================================
    `timescale1ns / 1ps
    moduleax_pwm
     #(
    parameterN = 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;
regpwm_r;
assignpwm_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
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. The "period" is a step value, which is the value to be added to each counter of the counter. Duty is the value of the duty cycle. We need to design some registers to control these parameters, here we need to use the AXI bus, PS to read and write registers through the AXI bus.

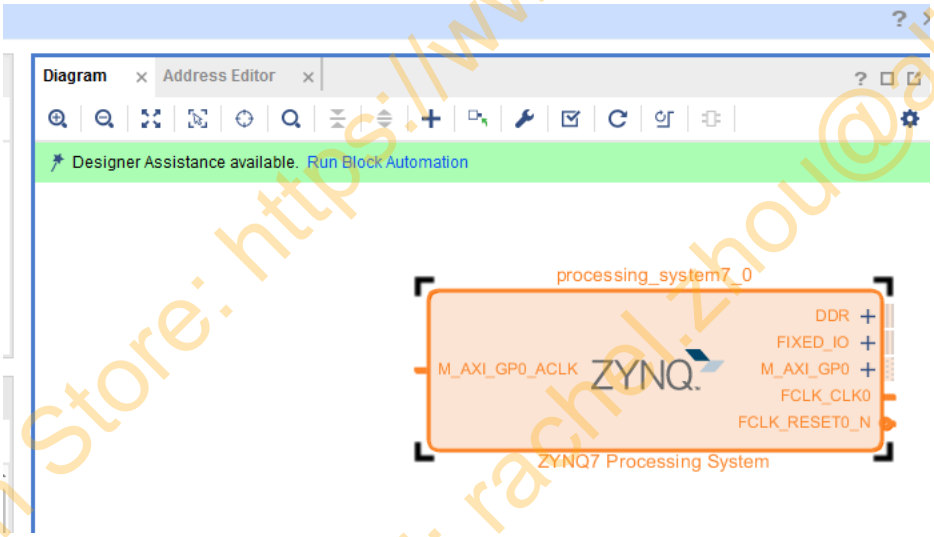
PWM frequency = period / (2 ^ N) × clk frequency (in Hz)

PWM duty cycle = 1-(duty+1)/(2^N)

Part 11.2: Create a Vivado project

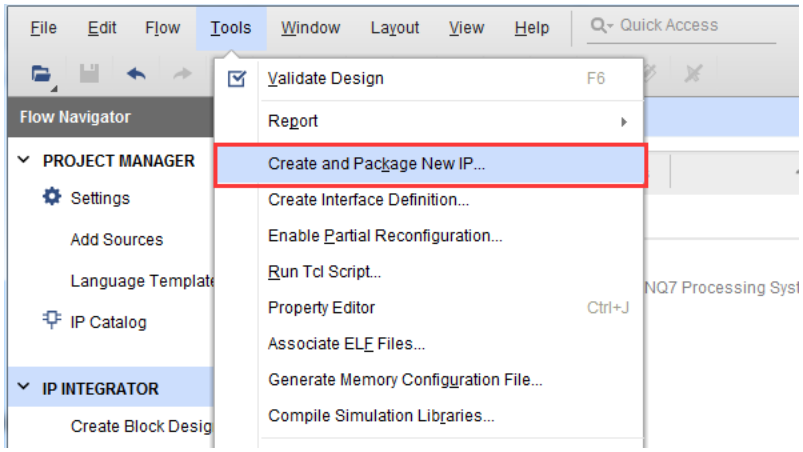
Part 11.2.1: Create a vivado project

Create a project named "custom_pwm_ip", add the zynq PS system, and configure the parameters, the specific method can refer to the previous method

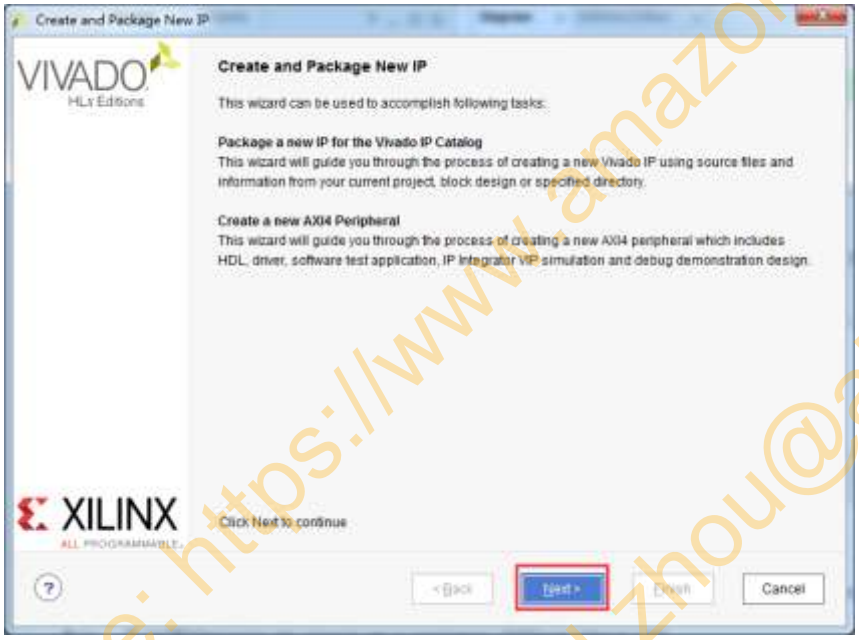


Part 11.2.2: Create a custom IP

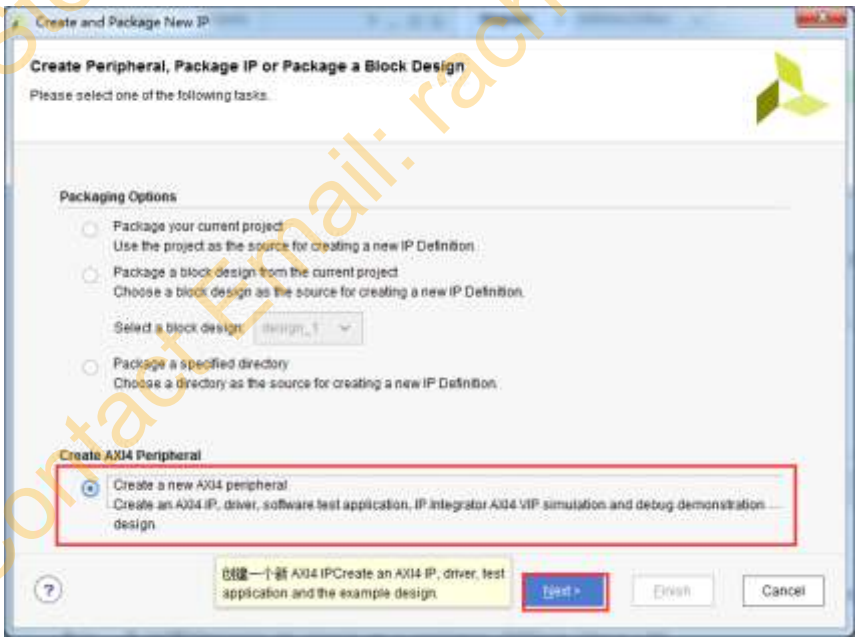
1) Click on the menu "Tools->Create and Package IP..."



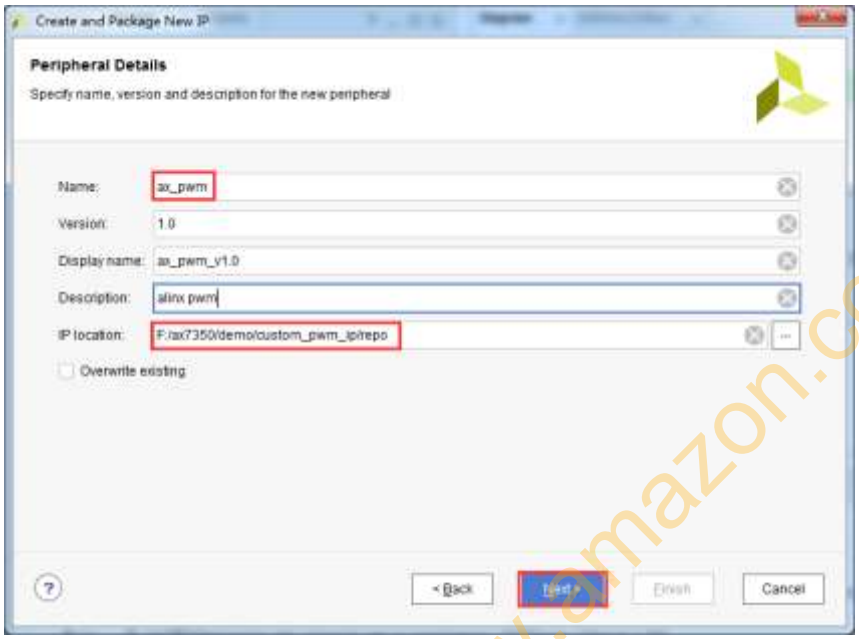
2) Select "Next"



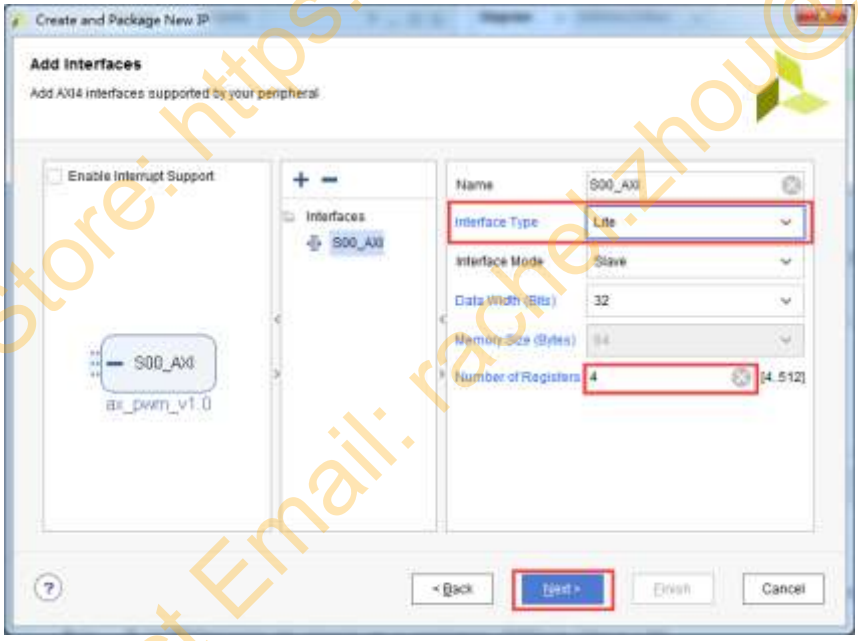
3) Choose to create a new AXI4 device



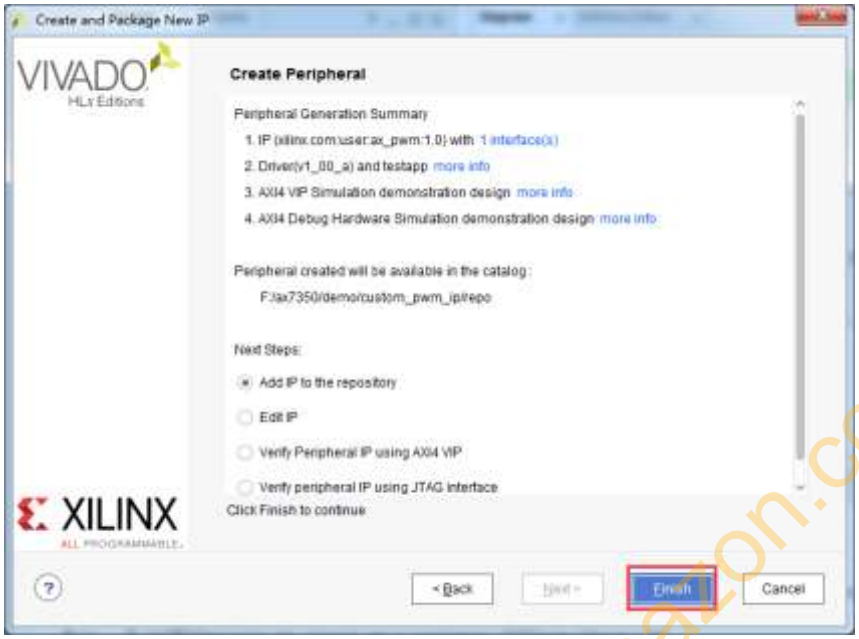
4) Fill in the name "ax_pwm", describe "alinx pwm", and then select a suitable location for IP



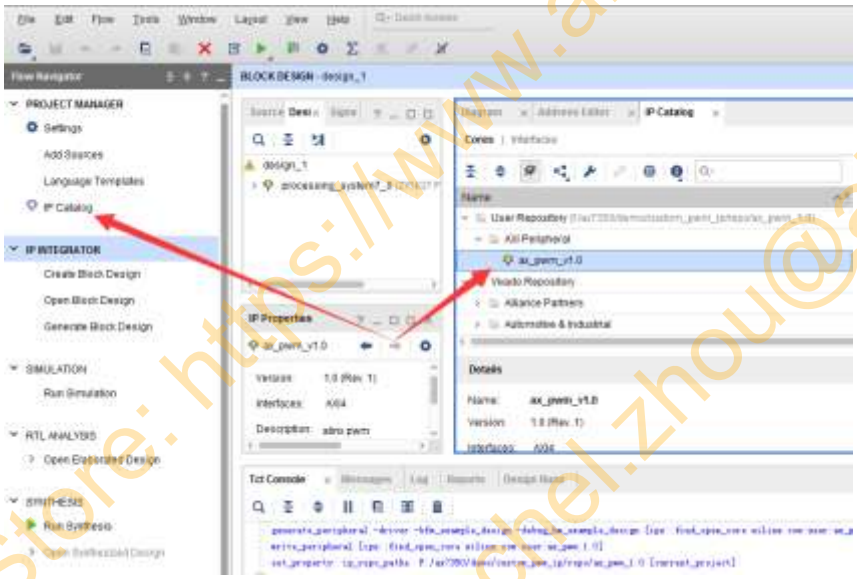
5) The following parameters can specify the interface type, the number of registers, etc., do not need to be modified here, use the AXILiteSlave interface, 4 registers.



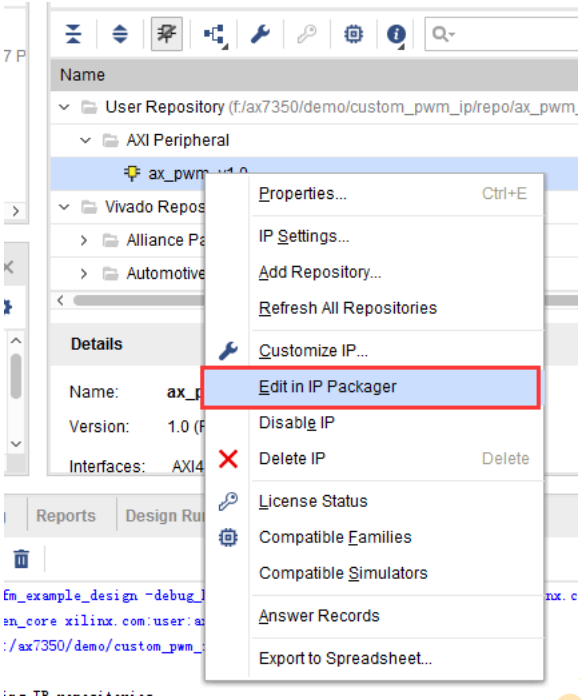
6) Click "Finish" to complete the creation of the IP.



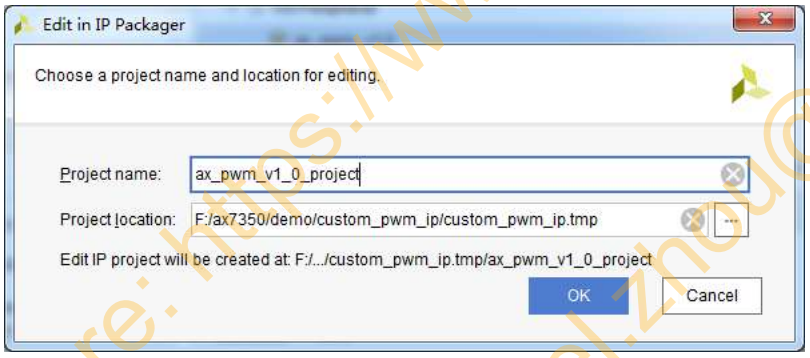
7) You can see the IP just created in "IP Catalog"



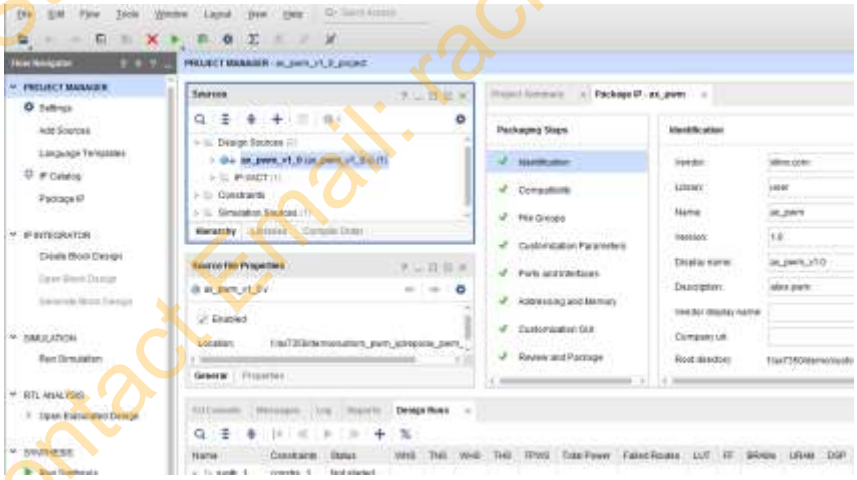
8) At this time, the IP only has a simple register read and write function. We need to modify the IP, select the IP, and right click "Edit in IP Packager"



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

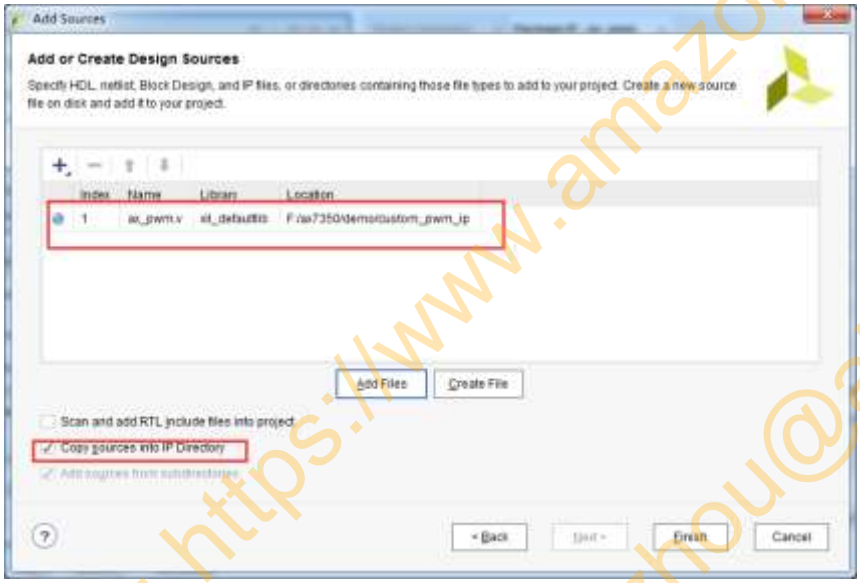


10) Vivado opens a new project

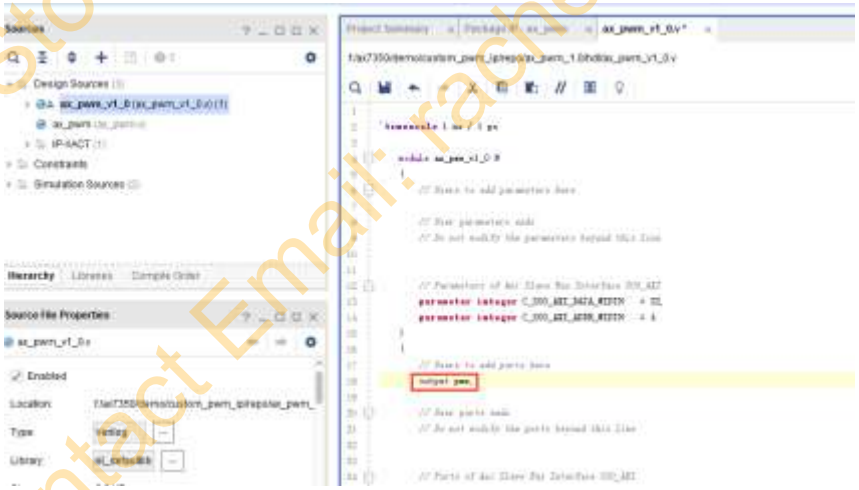


11)Add the core code of the PWM function

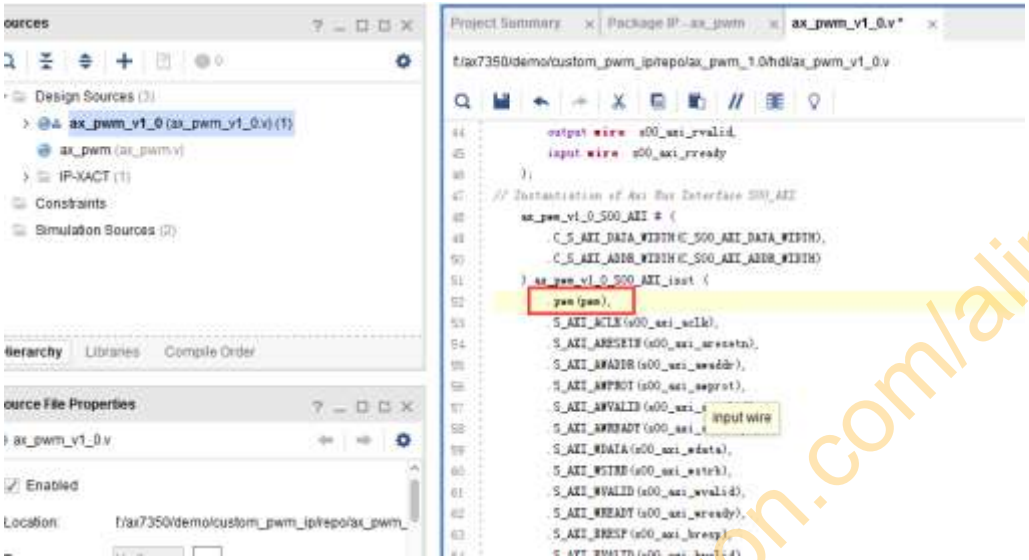
12) Select copy code to IP directory when adding code



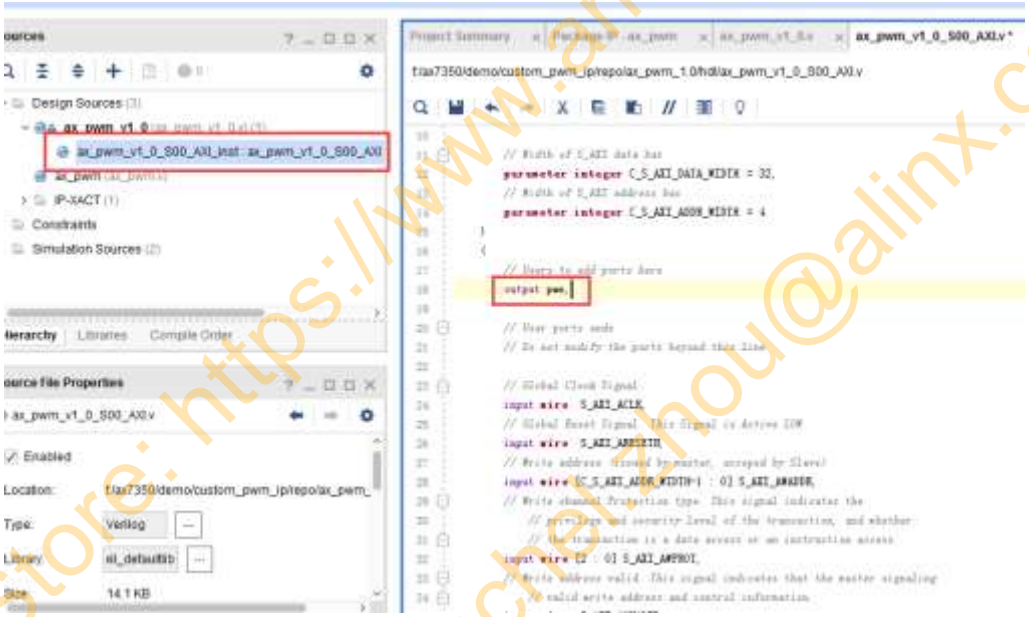
13) Modify "ax_pwm_v1_0.v" to add a pwm output port



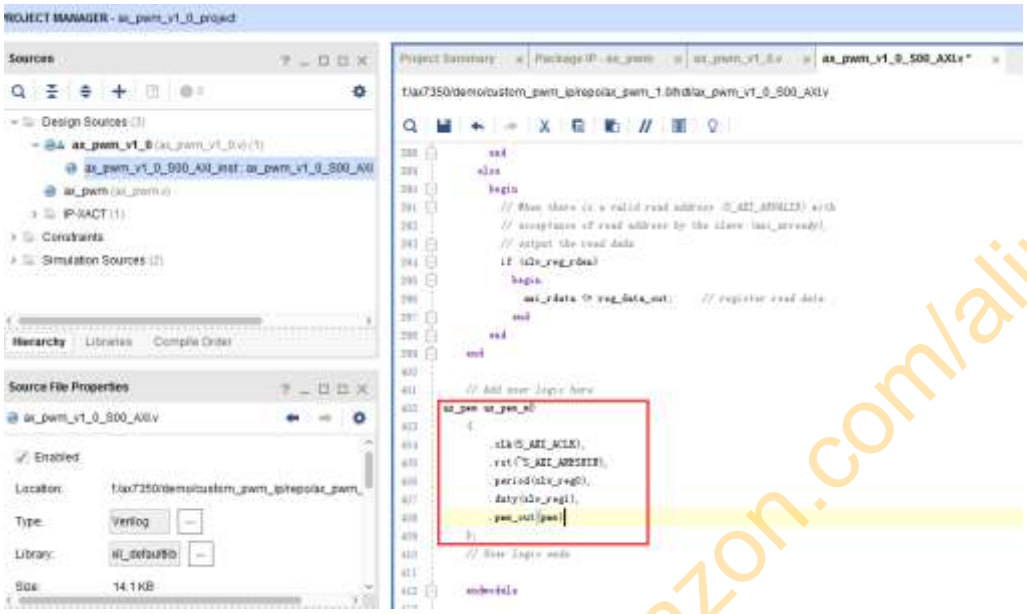
14) Modify "ax_pwm_v1_0.v" to add the pwm port routine to the routine "ax_pwm_V1_0_S00_AXI"



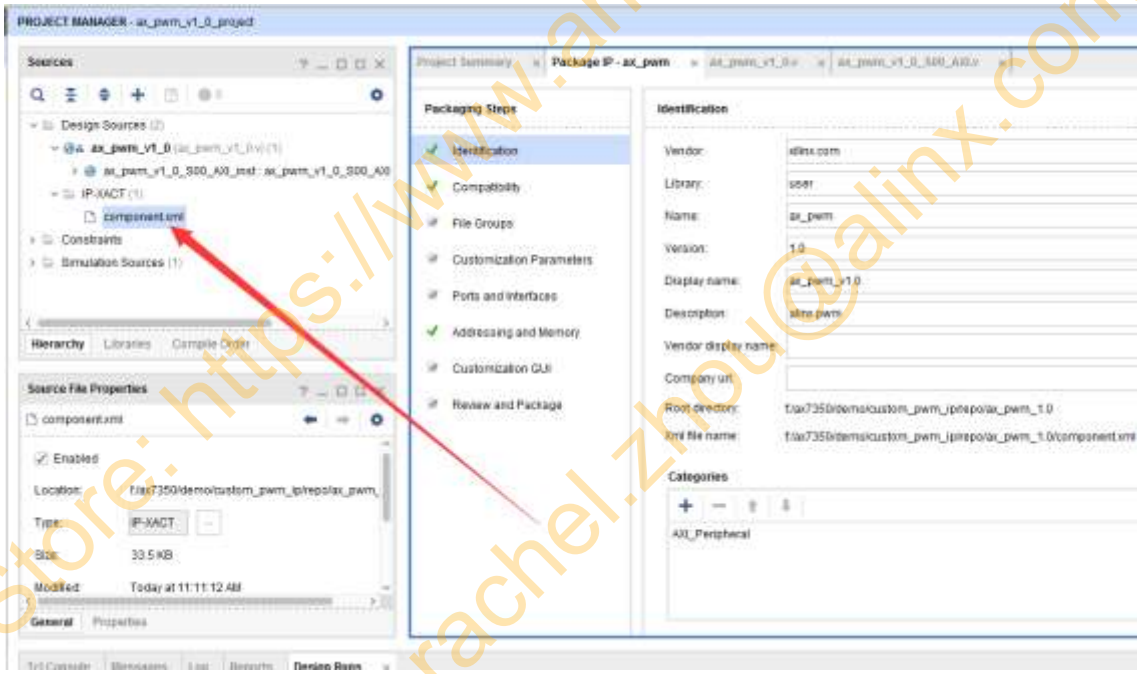
15) Modify the "ax_pwm_v1_0_s00_AXI.v" file and add the pwm port, which is the core code for implementing AXI4 Lite Slave.



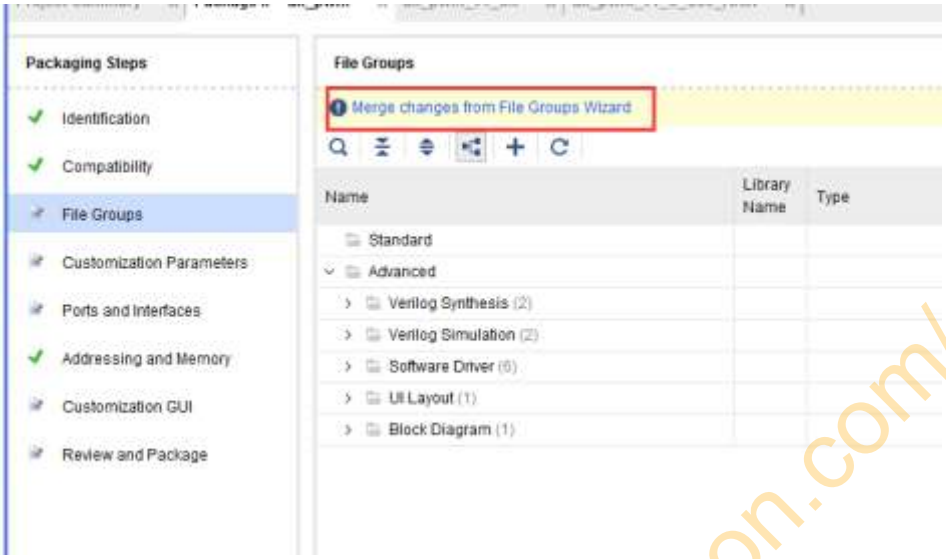
16) Modify the "ax_pwm_v1_0_s00_AXI.v" file, the routine pwm core function code, and use the registers slv_reg0 and slv_reg1 for parameter control of the pwm module.



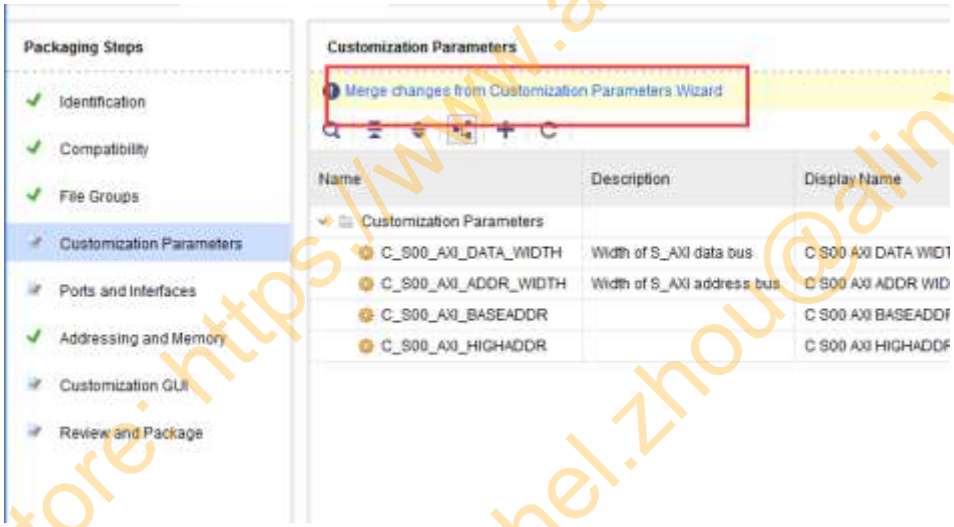
17)Double-click the "component.xml" file



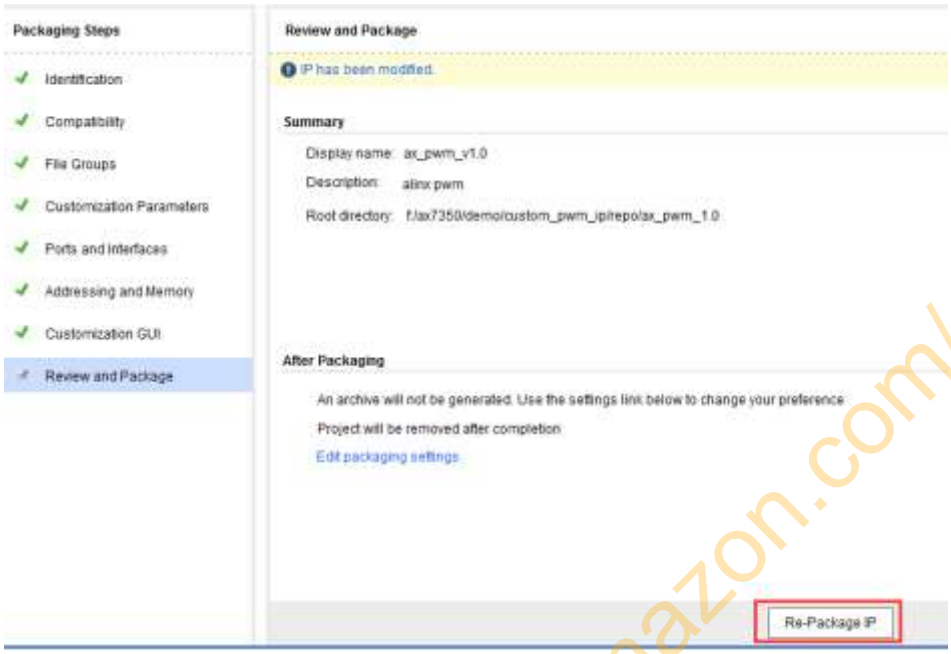
18)Click on "Merge changers from File Groups Wizard" in the "File Groups" option.



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

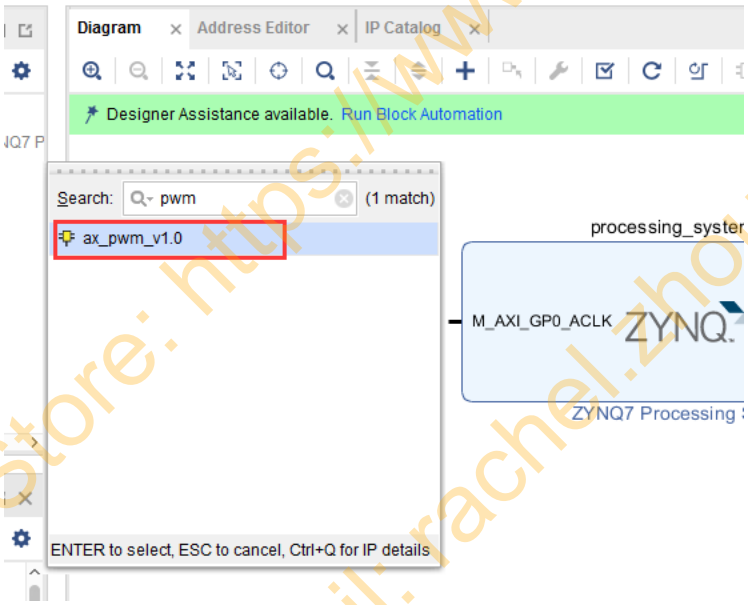


20)Click "Re-PackageIP" to complete the IP modification.

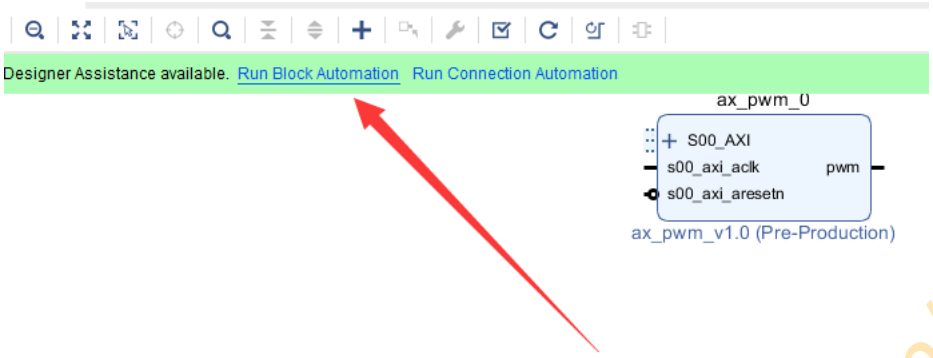


Part 11.2.3: Add a custom IP to the project

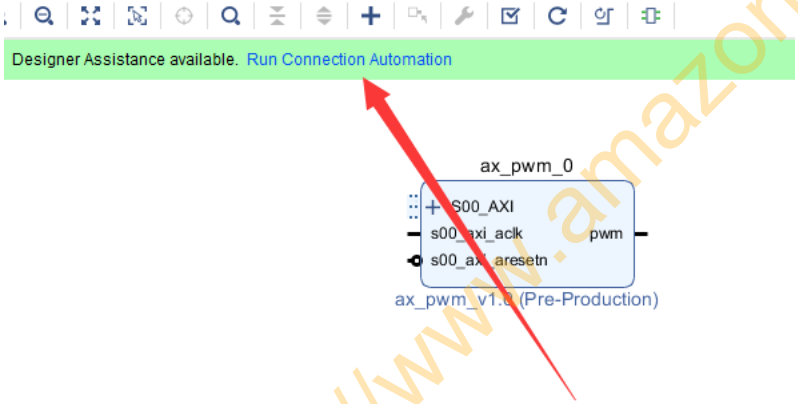
1) Search for "pwm" and add "ax_pwm_v1.0"



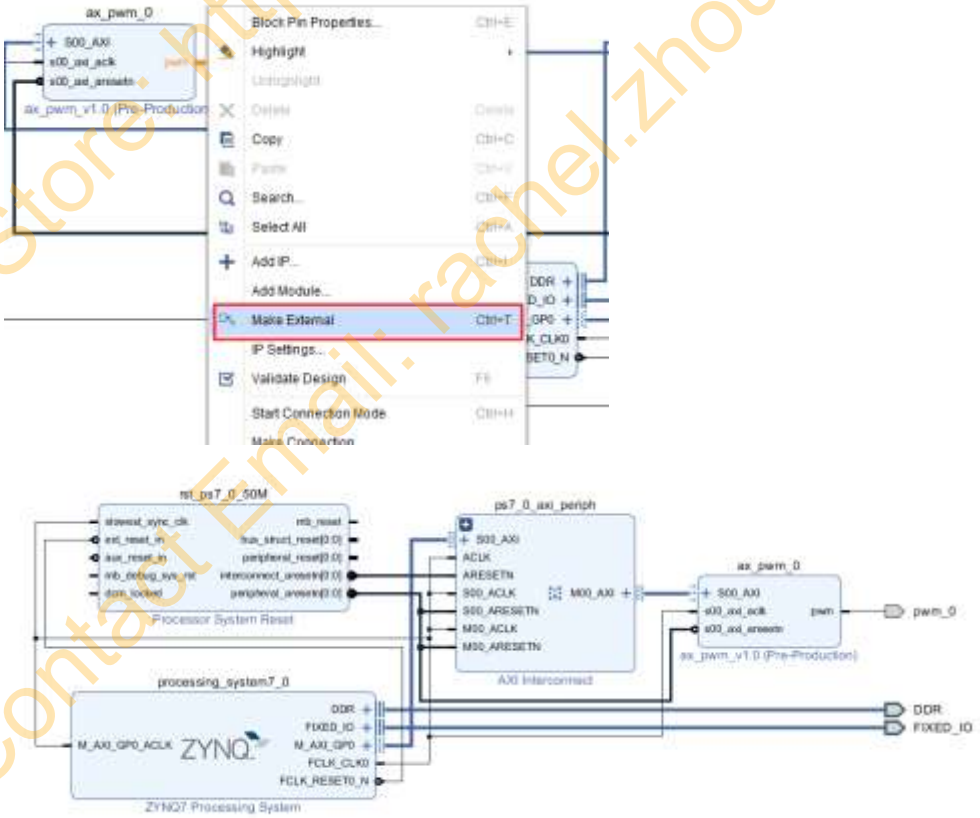
2) Click on " Run Block Automation "



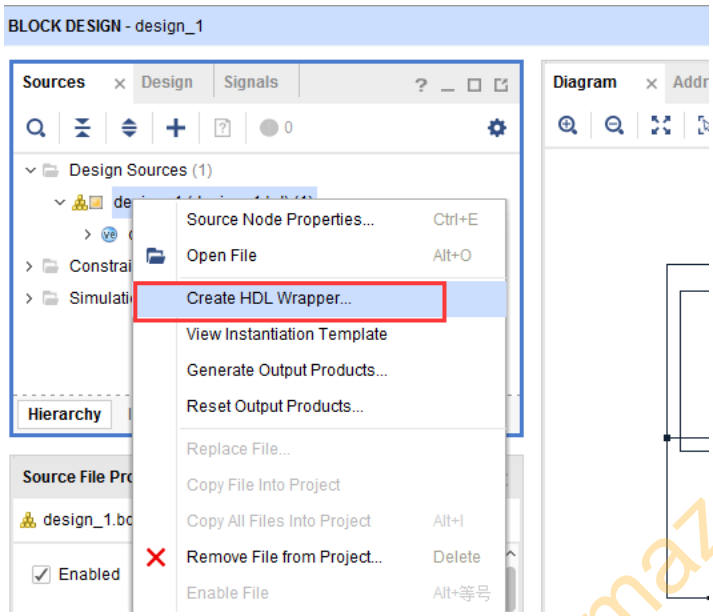
3) Click on " Run Connection Automation "



4) Export pwm port

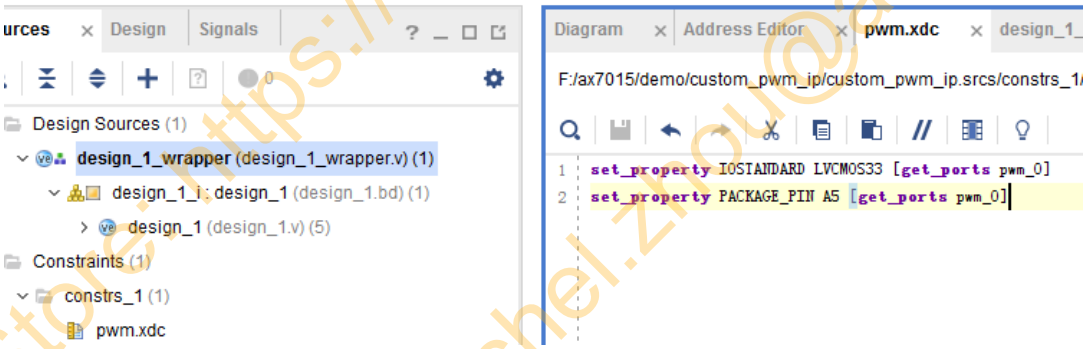


5) Create HDL file



6) Add the xdc file distribution pin, assign the pwm_0 output port to PLLED1, and make a breathing light

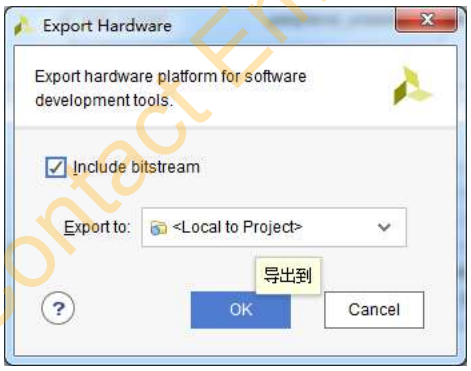
```
set_property IOSTANDARD LVCMOS33 [get_ports pwm_0]
set_property PACKAGE_PIN M14 [get_ports pwm_0]
```



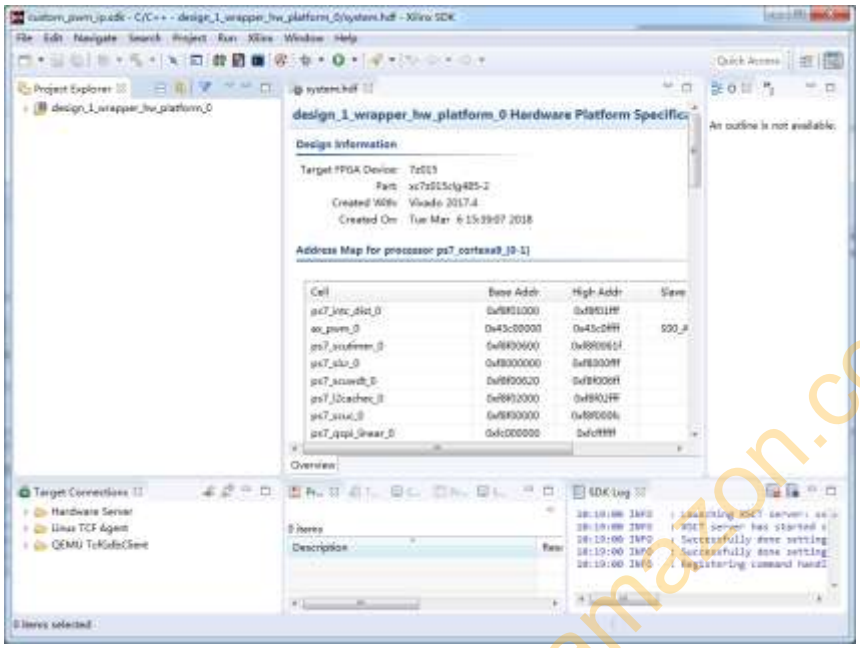
7) Compile and generate bit file, export hardware.

Part 11.3: SDK Software Writing and Debugging

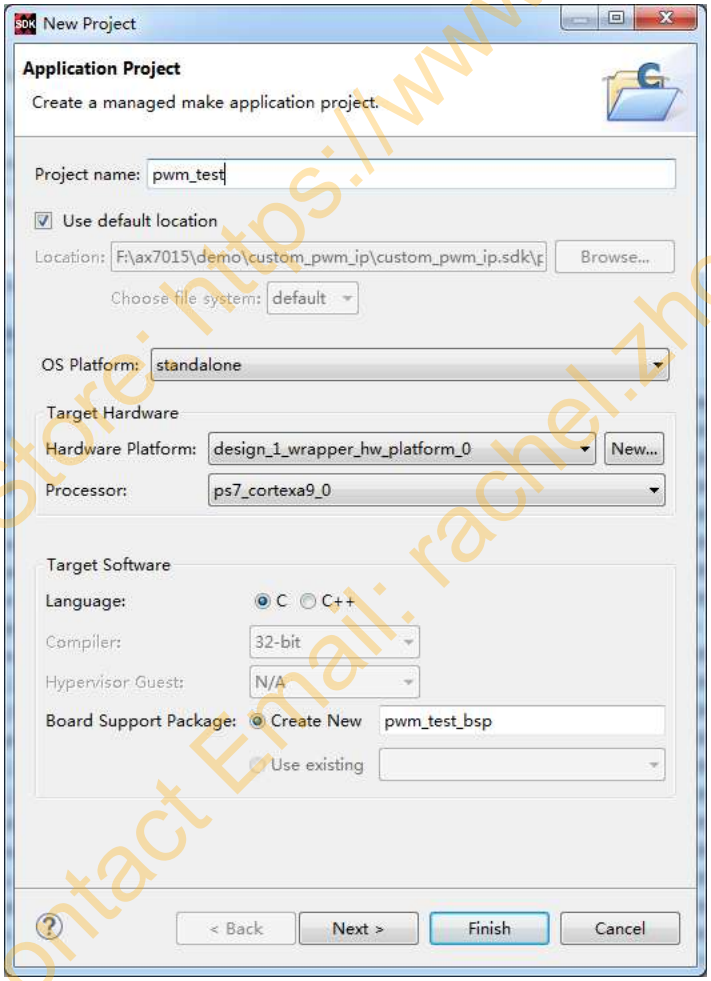
1) Export hardware



2) Start the SDK



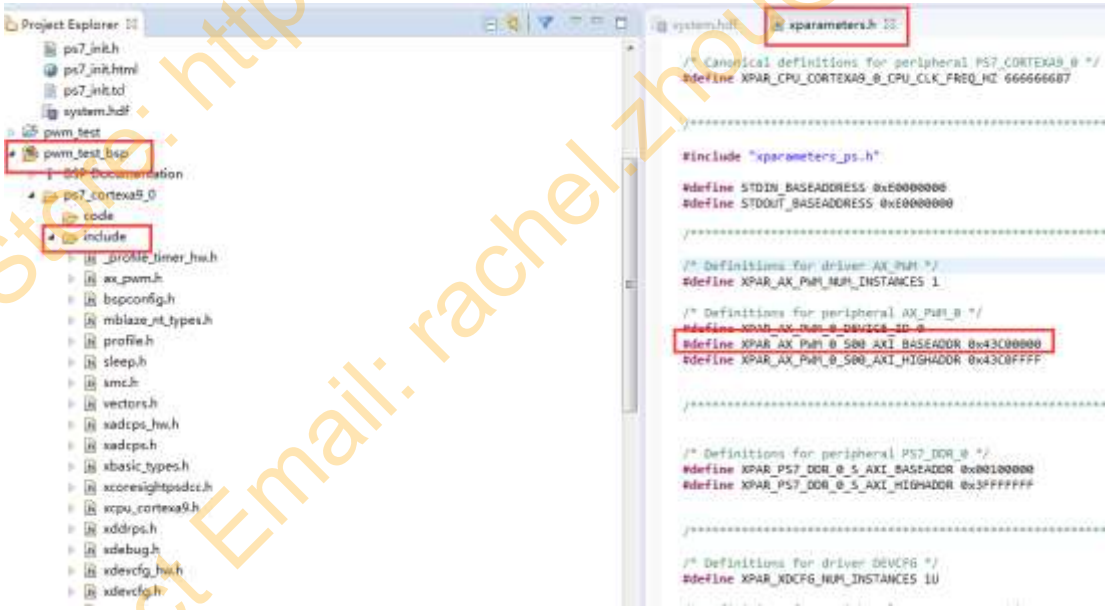
3) Create a new app, and select "Hello World" for the template.



4) The previous examples all use the IP of xilinx. xilinx provides a set of APIs. For this custom IP, we need to develop it ourselves. First look

at the resources in the APP directory, you can find a file ax_pwm.h, which contains Read and write macro definitions for custom IP registers

5) Find the "xparameters.h" file in bsp, this very important file, which finds the register base address of the custom IP, you can find the base address of the custom IP.



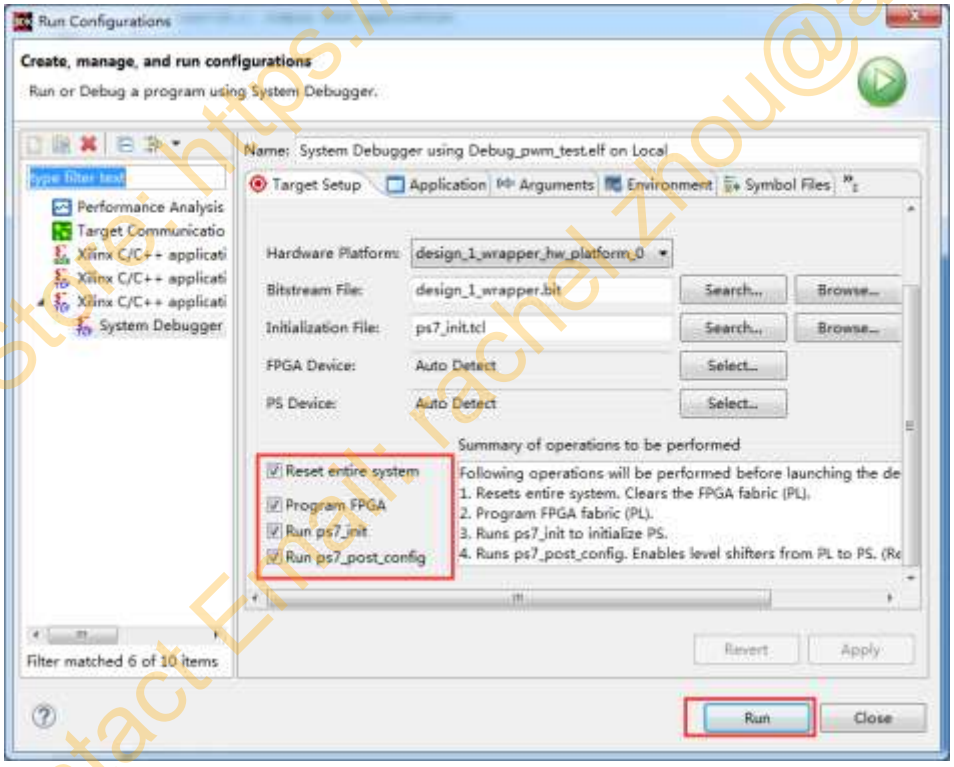
6) There is a register read and write macro and the base address of the custom IP. We start writing code to test the custom IP. We first control the PWM output frequency by writing to the register AX_PWM_S00_AXI_SLV_REG0_OFFSET, and then controlling the

duty cycle of the PWM output by writing to the register AX_PWM_S00_AXI_SLV_REG1_OFFSET

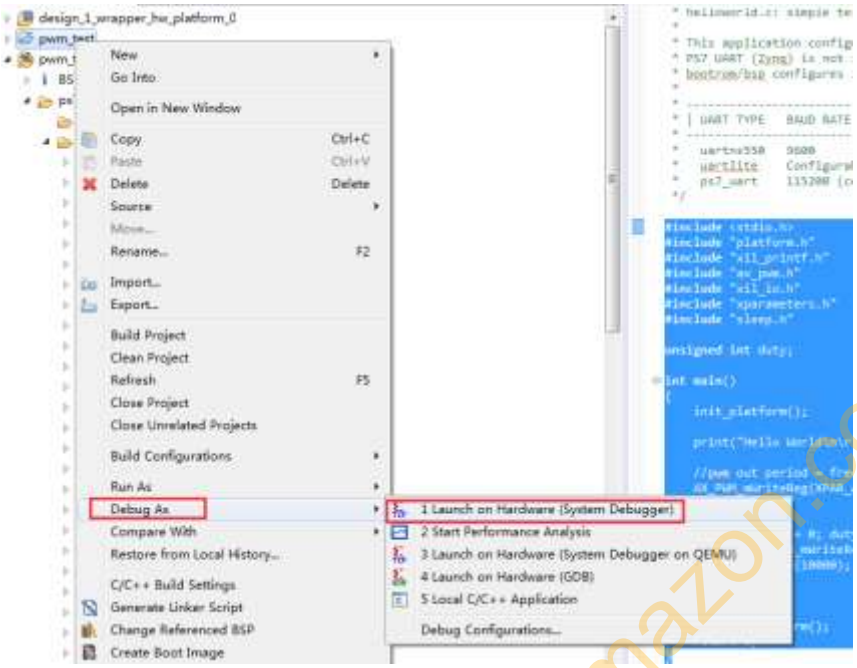
```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "ax_pwm.h"
#include "xil_io.h"
#include "xparameters.h"
#include "sleep.h"
```

```
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

     while (1) {
          for (duty = 0x8fffffff; 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;
}
```

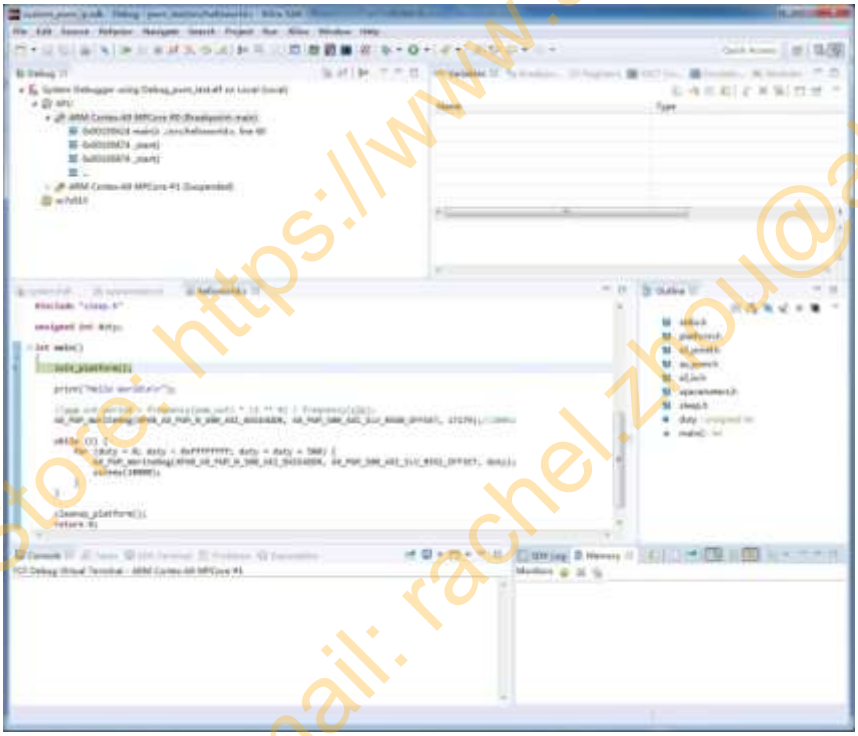
7) By running the code, we can see that LED2 shows the effect of a breath light.



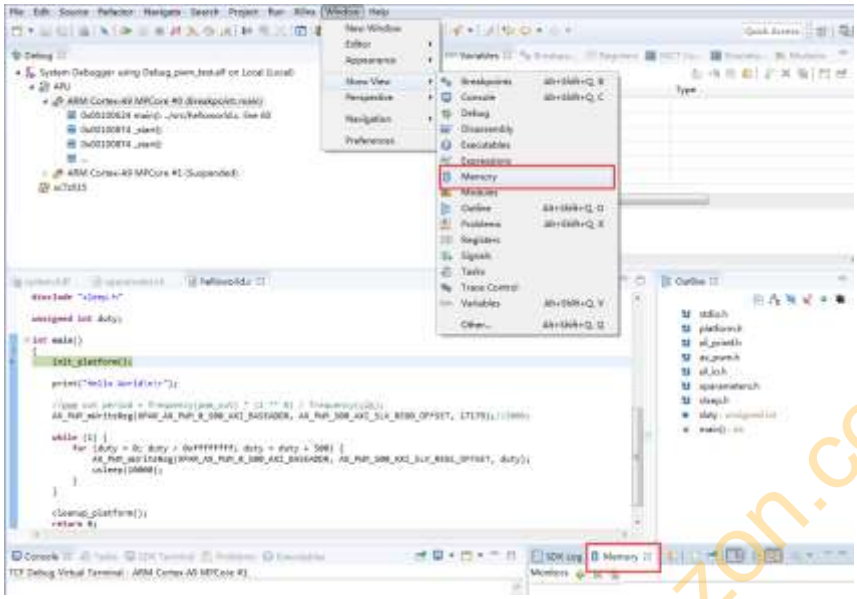
8) Through debug, let's look at the register



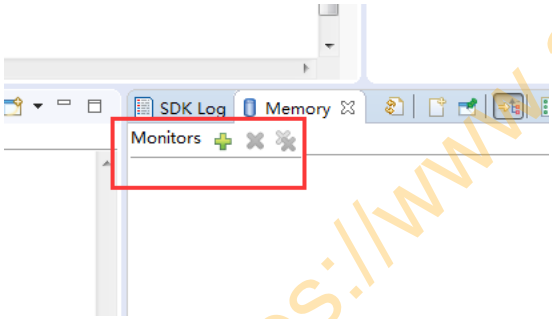
9) Enter the debug state and press "F6" to step through

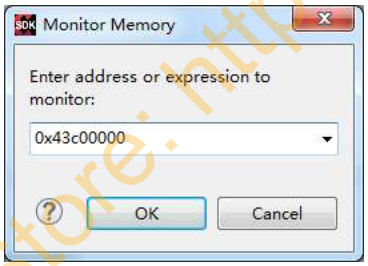


10) View the "Memory" window through the menu

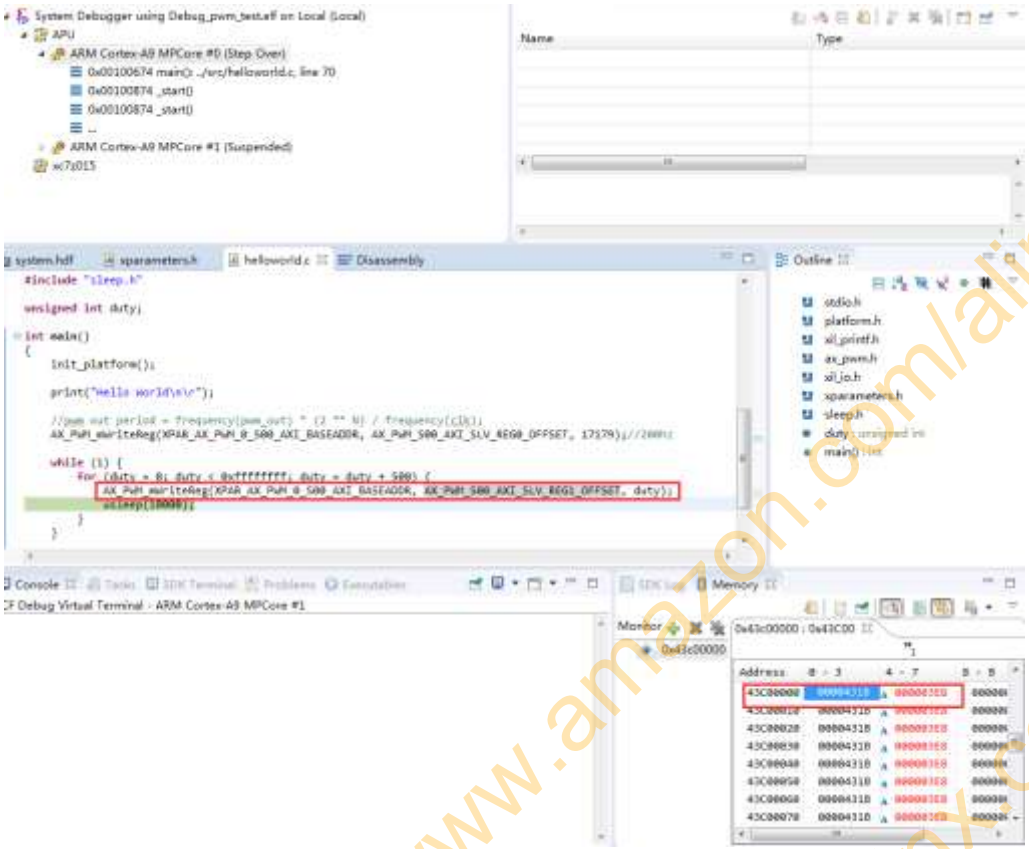


11) Add a monitoring address "0x43c00000"





12) Single step, observe changes



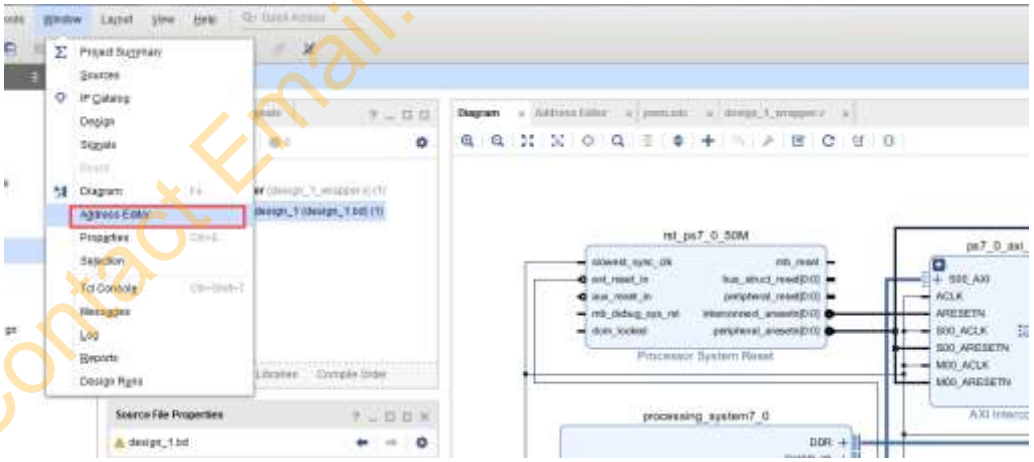
Part 11.4: Experimental summary

Through this experiment, we have mastered more SDK debugging skills, and mastered the core content of ARM + FPGA development, which is ARM and FPGA data interaction.

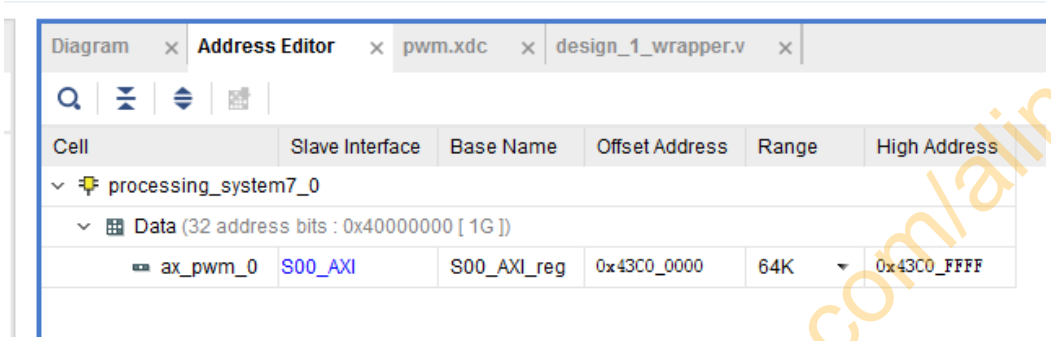
Part 11.5: Q&A

Part 11.5.1: How to know the base address of AXI IP ?

1) As shown in the figure below, open "Address Editor", you can see the address allocation



2) The address is usually assigned automatically by Vivado, we can also modify the address



Part 12: Use VDMA to drive HDMI display

Experiment Vivado project is " vdma_hdmi_out "

PS does not have an integrated display control system and needs to be implemented with the help of PL. There are many implementation schemes, but they are all inseparable from the DMA system. The DMA system can read the display data from ddr3 to the display, reducing CPU overhead. 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 transmits the HDMI interface through VDMA.

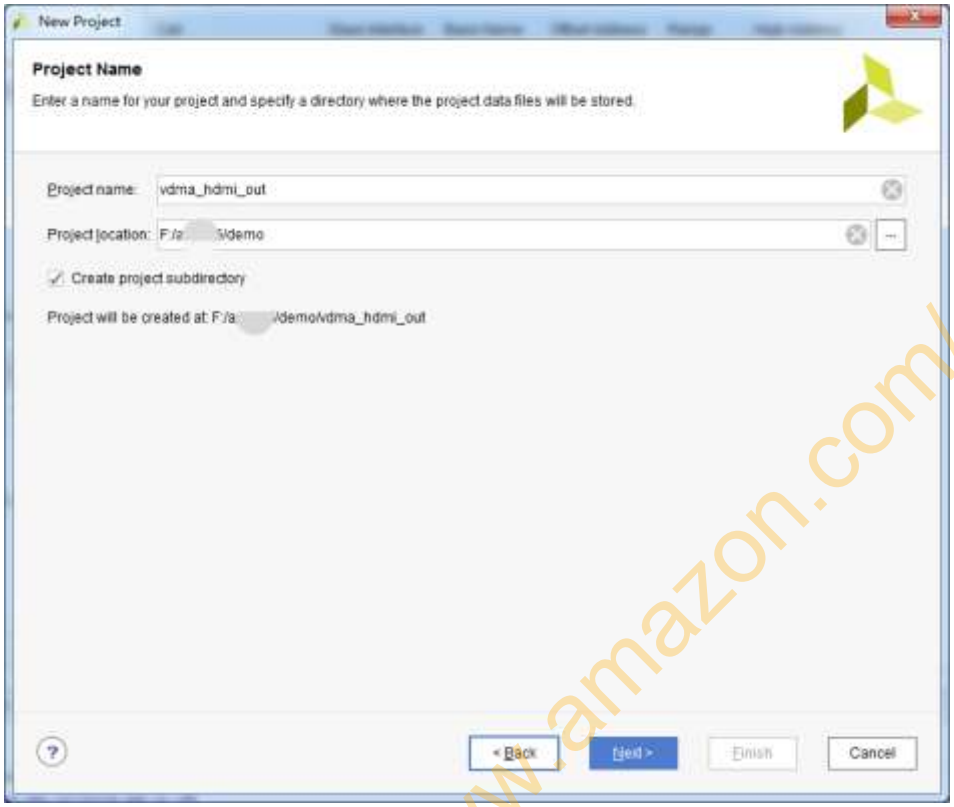
FPGA Engineer Work Content

The following is a description of what the FPGA engineer is responsible for.

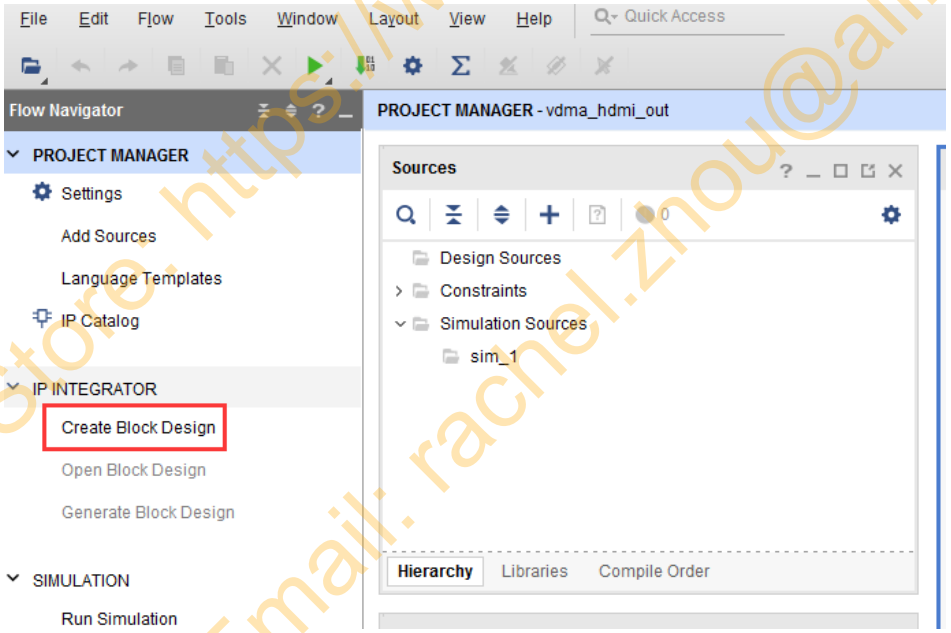
Part 12.1: Create a Vivado project

Since the VDMA display is a very important part, this experiment will detail the Vivado build process.

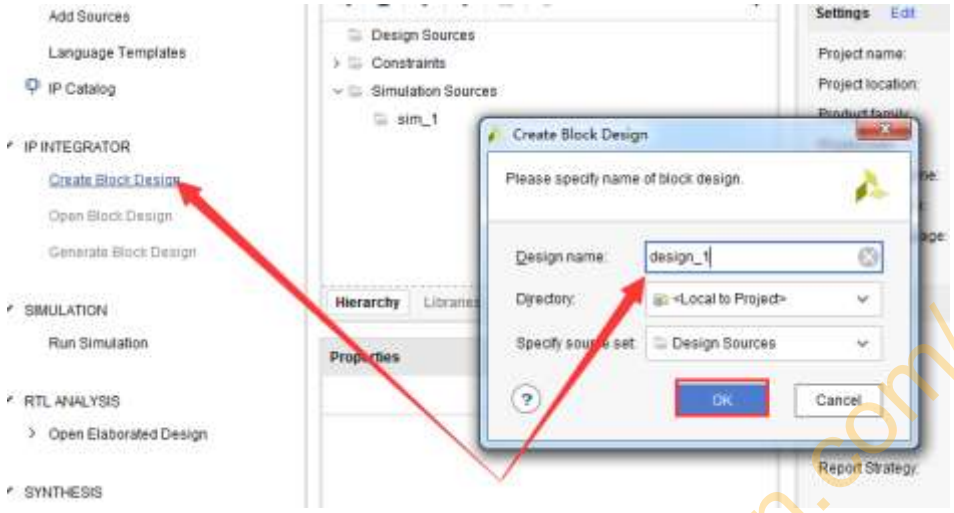
1) Create a new project called "vdma_hdmi_out"



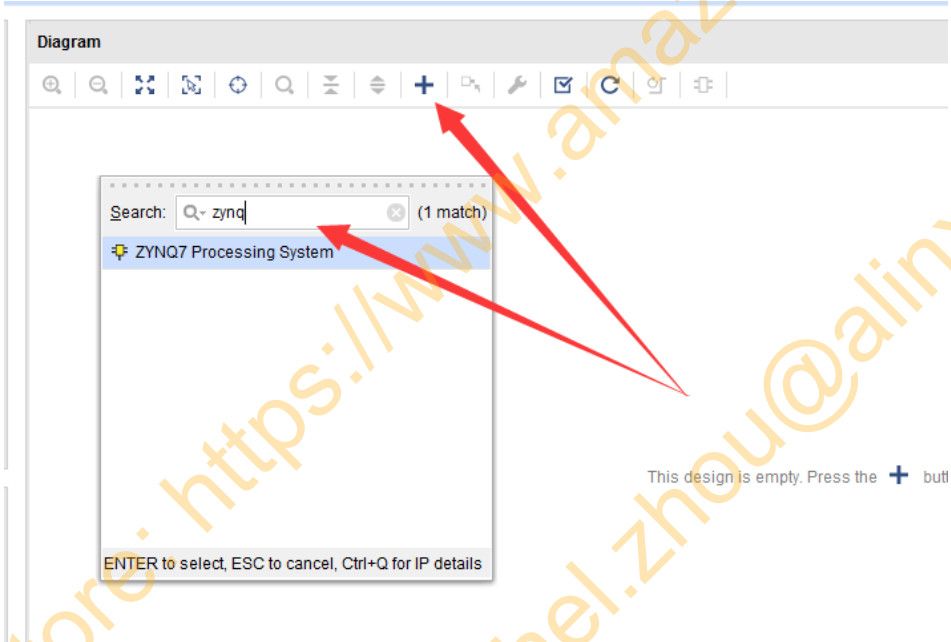
2) Create a block design



3) The design name remains the same by default



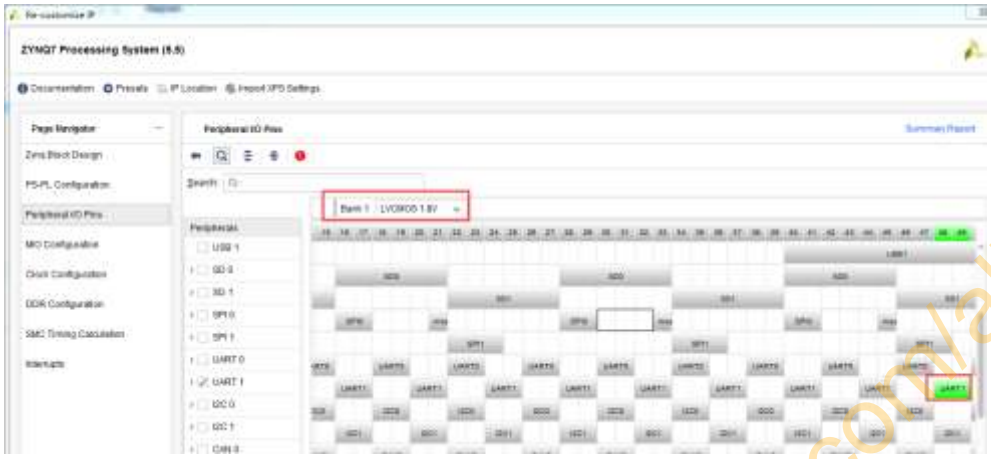
4) Add ZYNQ processor



5) Configure ZYNQ parameters and enable HP0 interface for VDMA to quickly read DDR3.

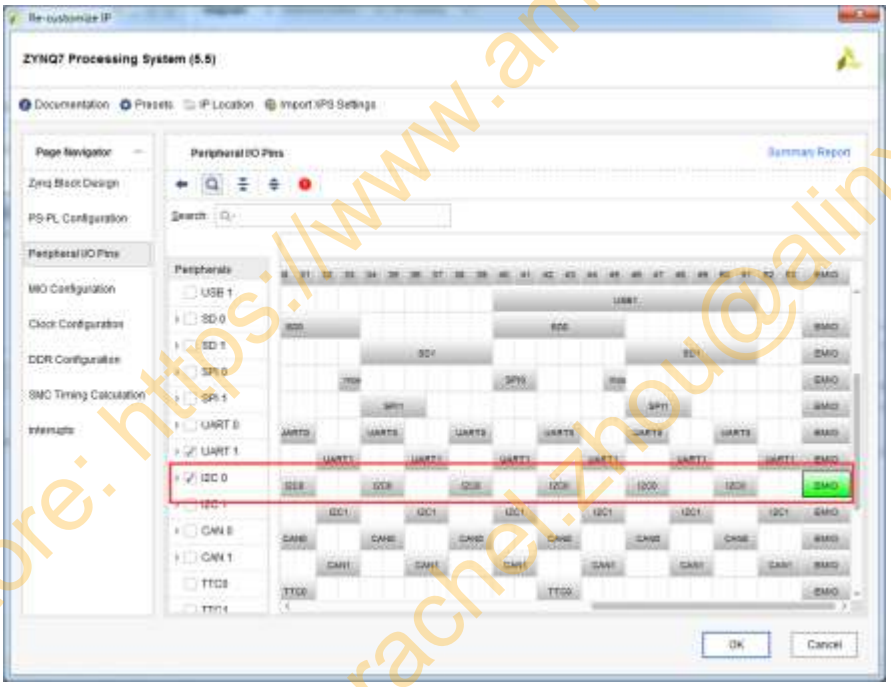
Part 12.1.1: Configuring UART

6) Configure Bank level standard, Bank0 is LVCMOS 3.3V, Bank1 is LVCMOS 1.8V, serial port is enabled

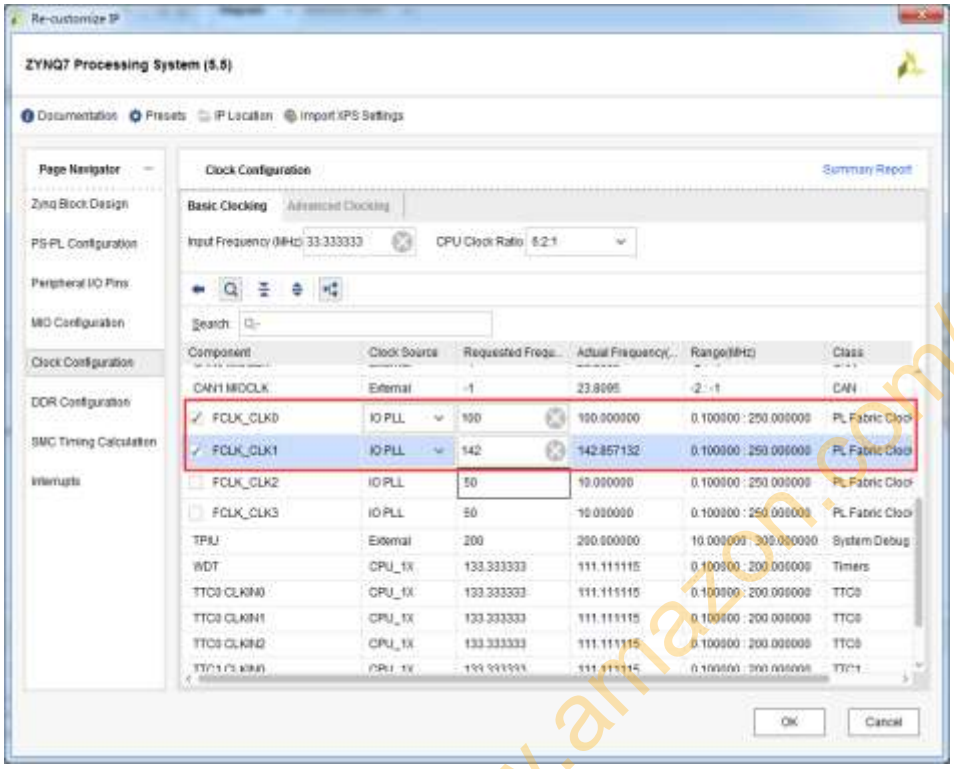


Part 12.1.2:

7) Enable I2C0 and select EMIO, so that I2C can be connected to the PL side for connection to the HDMI DDC

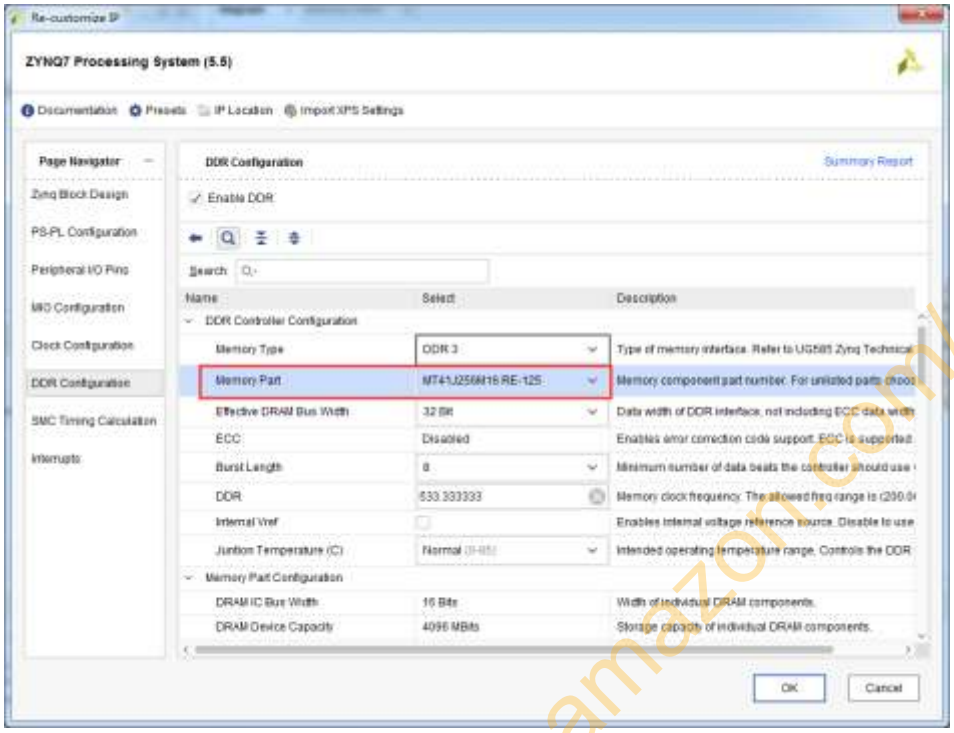


8) Configure the clock. FCLK_CLK0 is configured for 100Mhz and FCLK_CLK1 is configured for 142Mhz. This clock is used for VDMA to read data.



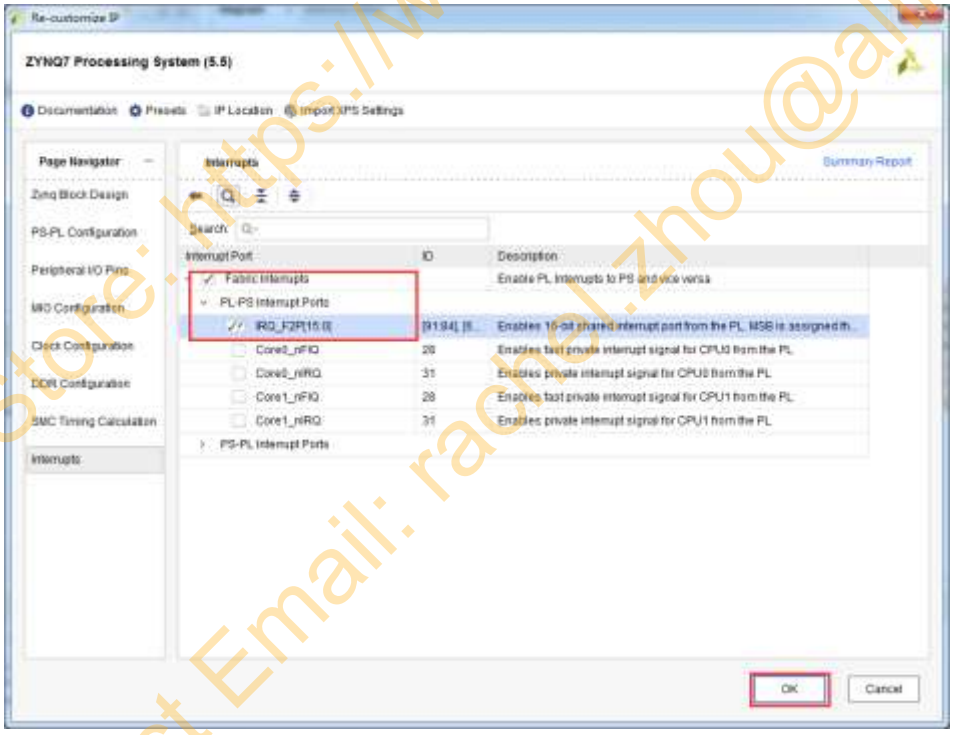
Part 12.1.3: Configuration DDR3

9) In the "DDR Configuration" tab, you can configure the PS side ddr parameters. AX7010 FPGA development board configuration DDR3 model is "MT41J128M16 HA 125". AX7020 FPGA development board configuration DDR3 model is "MT41J256M16 RE 125." Here ddr3 model is not developed The ddr3 model on the board is the model with the closest parameters. "Effective DRAM Bus Width", select 32 Bit



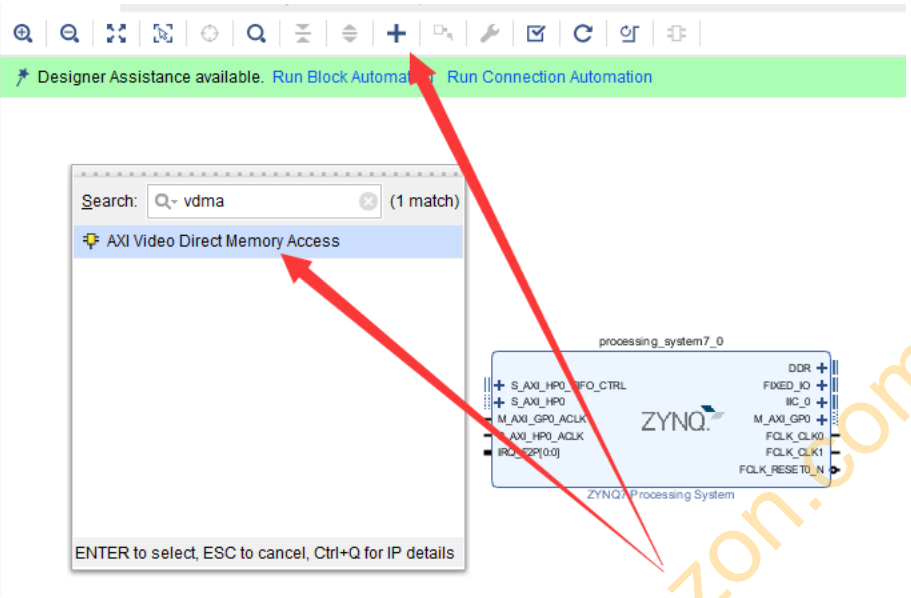
Part 12.1.4: Configuration Interrupt

10)Configure interrupts, enable IRQ_F2P, receive interrupts from PL

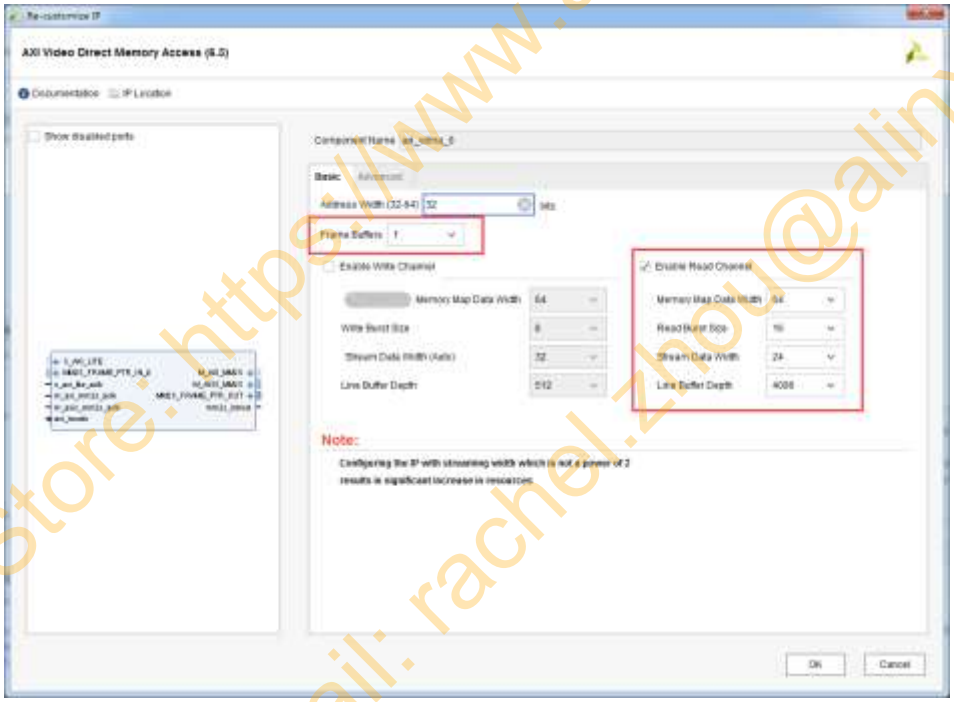


Part 12.1.5: Configuring VDMA

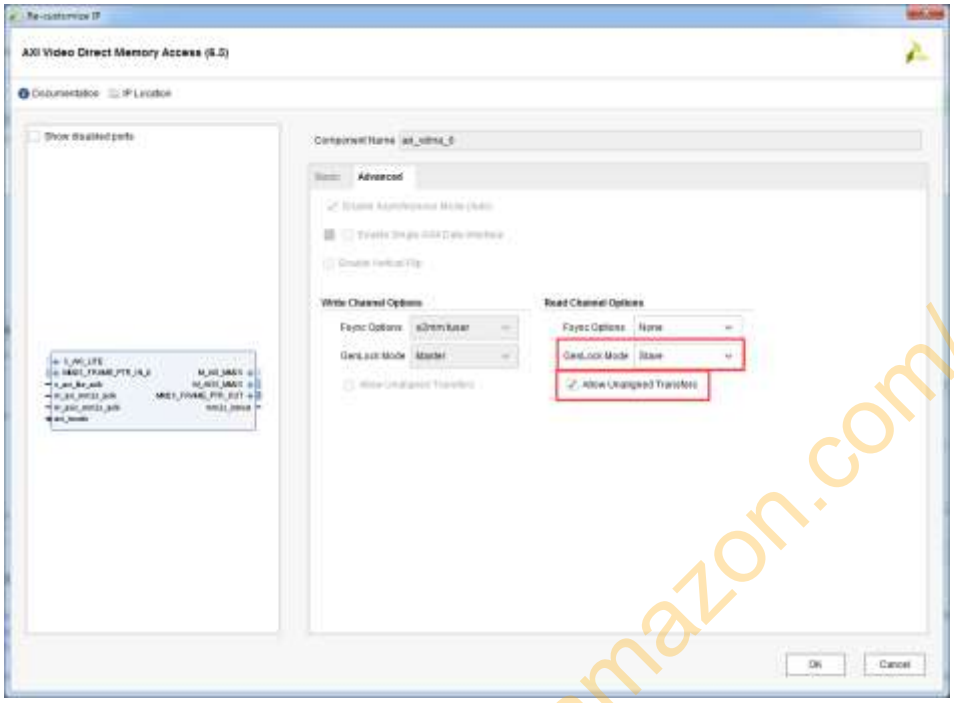
11)Add VDMA IP



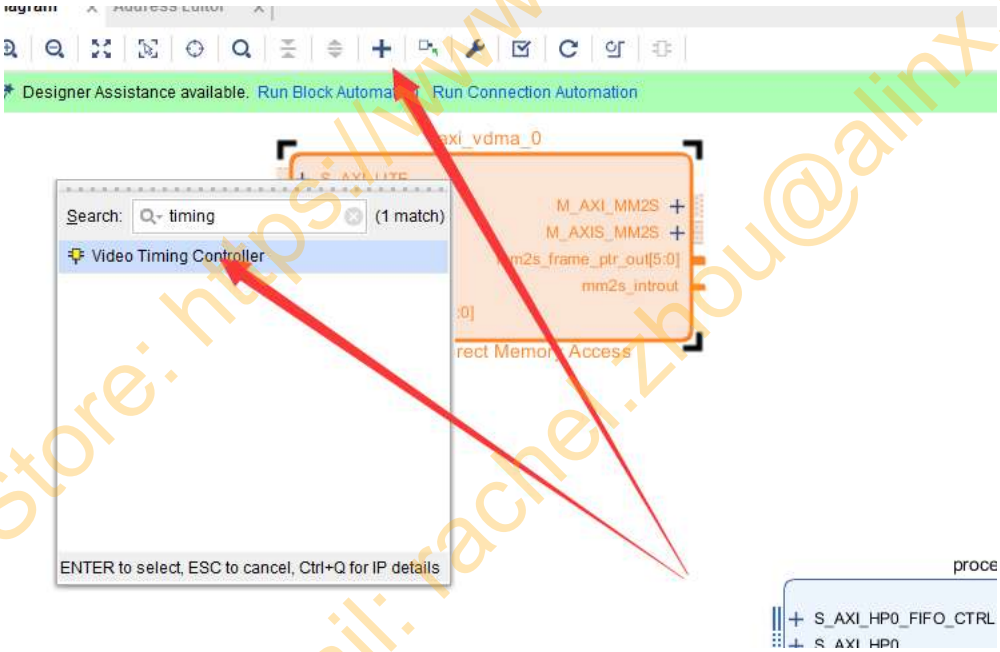
12)Configure the basic parameters of VDMA according to the figure below



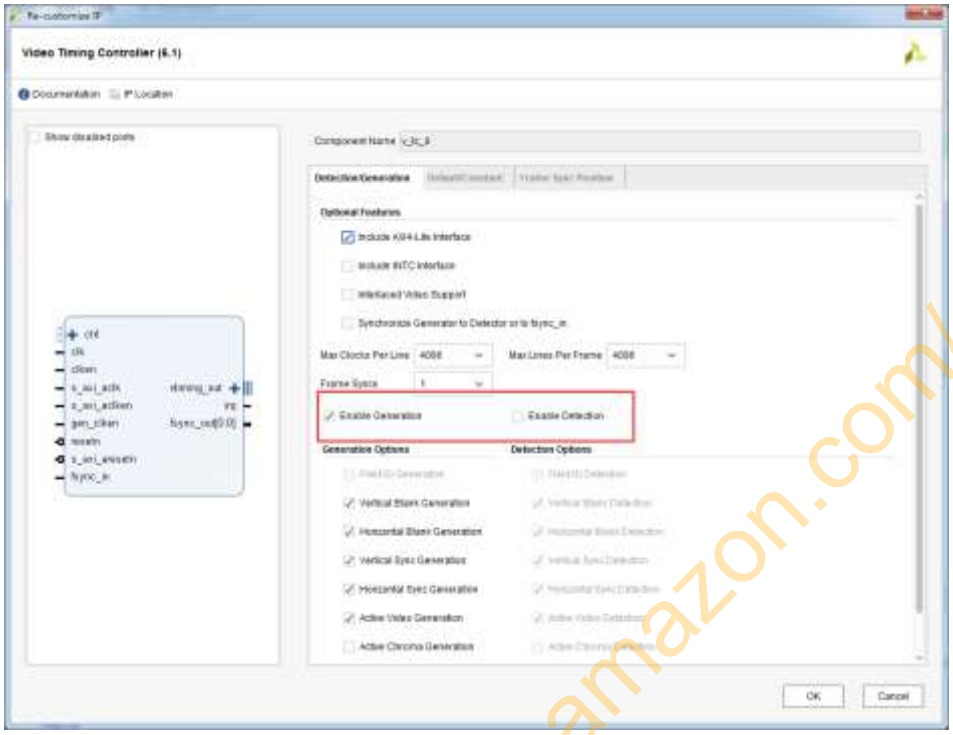
13)Configure VDMA advanced parameters



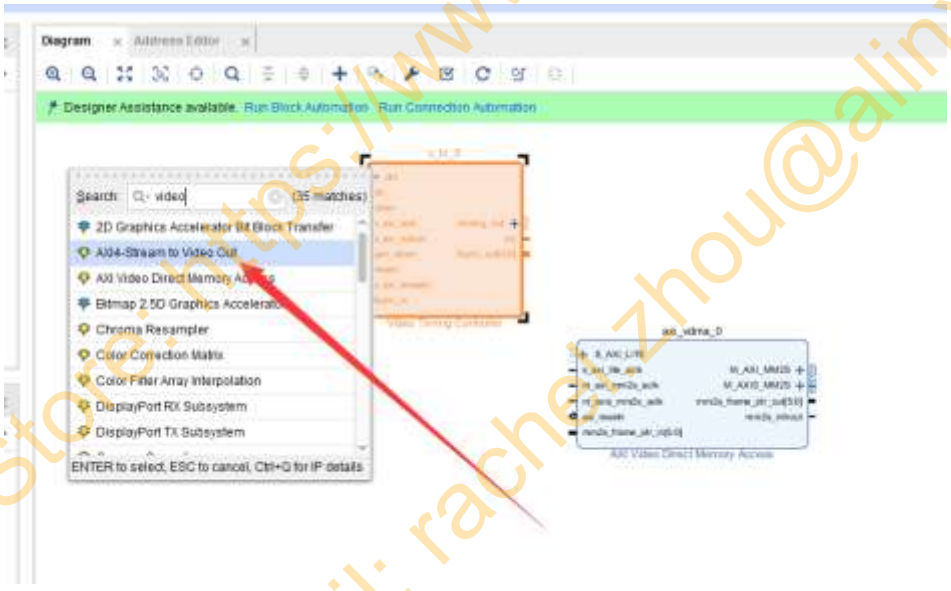
14)Add video timing controller



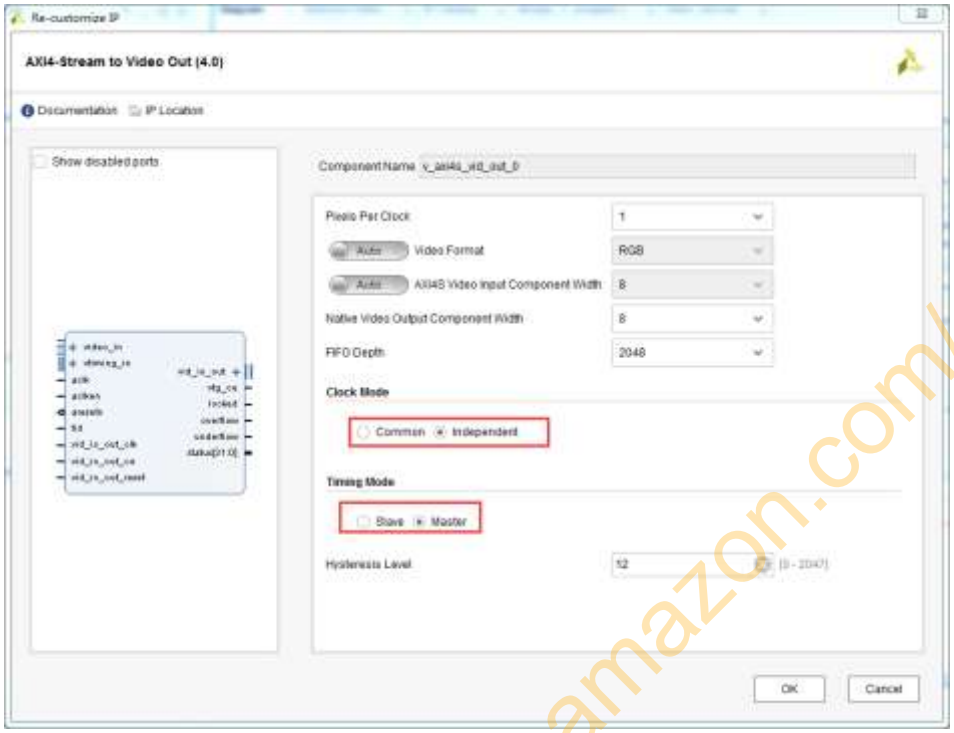
15)Configure video timing controller parameters



16)Add AXI streaming video output controller

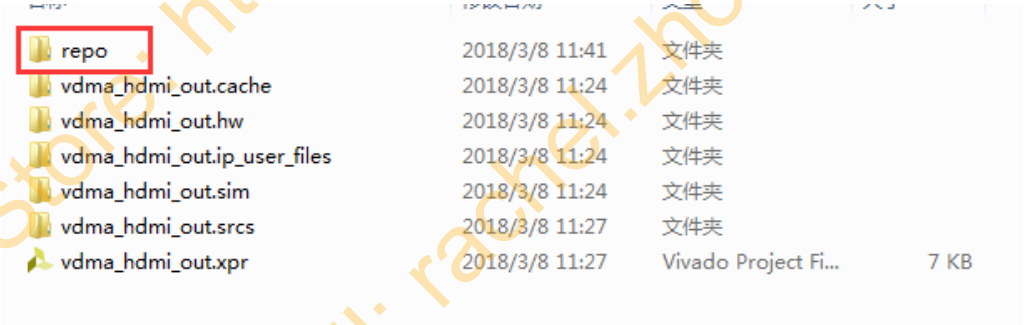


17)Configure AXI streaming video output controller parameters

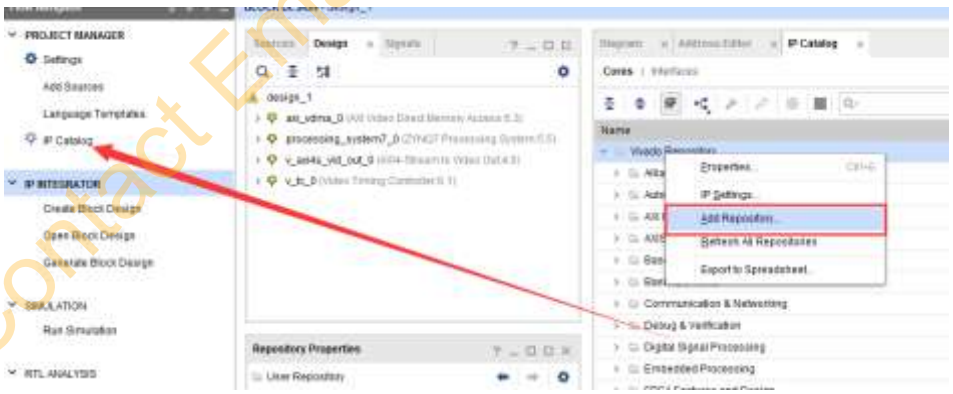


Part 12.1.6: Adding a Custom IP

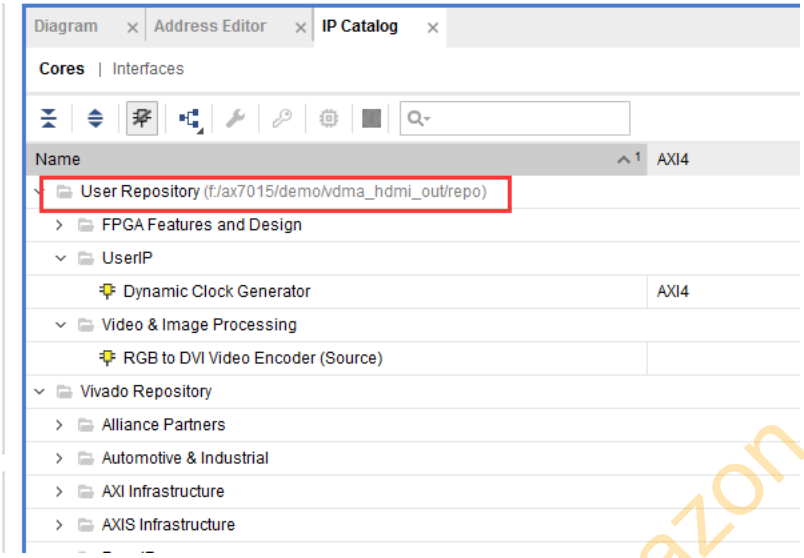
18) Since the video has a lot of resolution, and the various clock frequencies are different, you need to use a dynamic clock controller. This IP comes from open source software. Find the repo directory in the routine and copy it to your own directory.



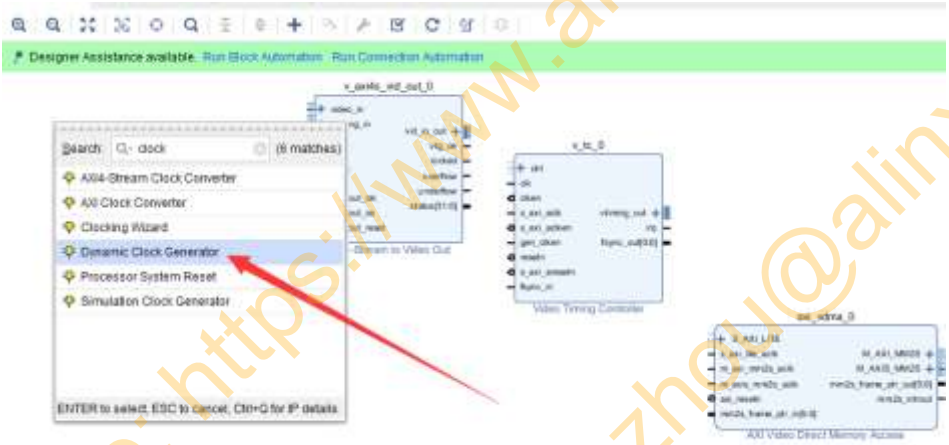
19) Add IP Library



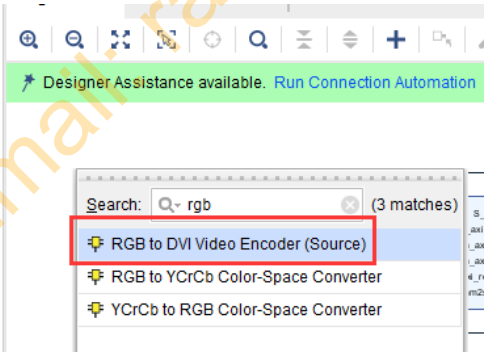
20) You can see a lot of IP after the addition is complete.

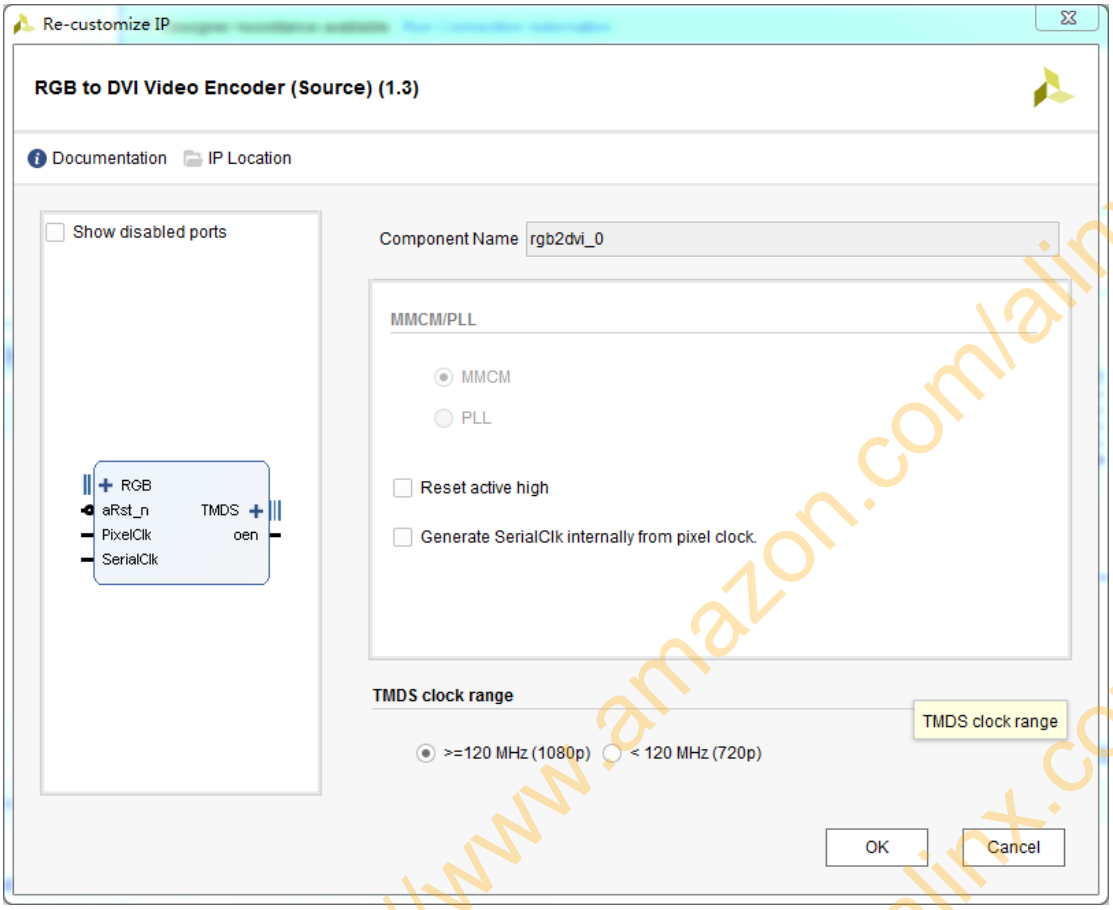


21)Add dynamic clock controller.

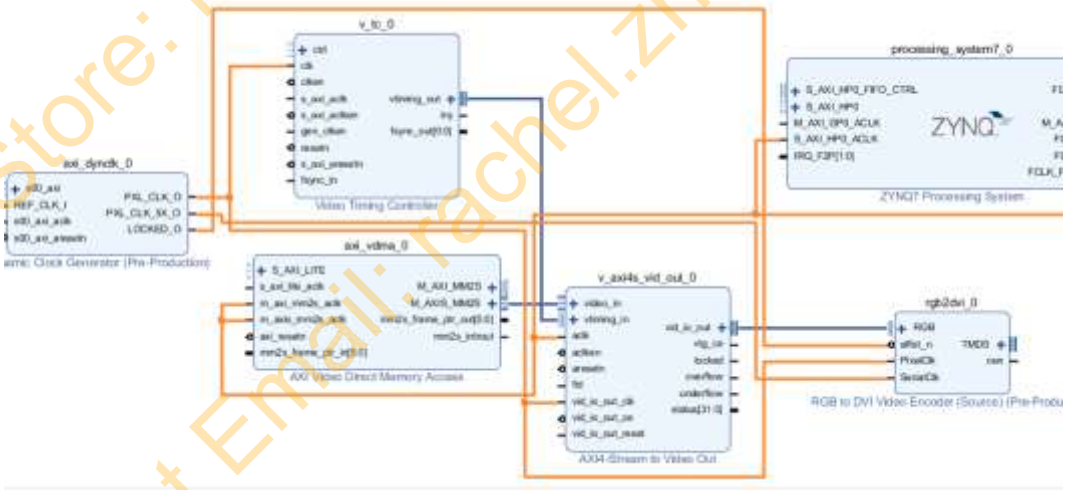


Part 12.1.7: Add an HDMI encoder for converting RGB data to TMDS signals.

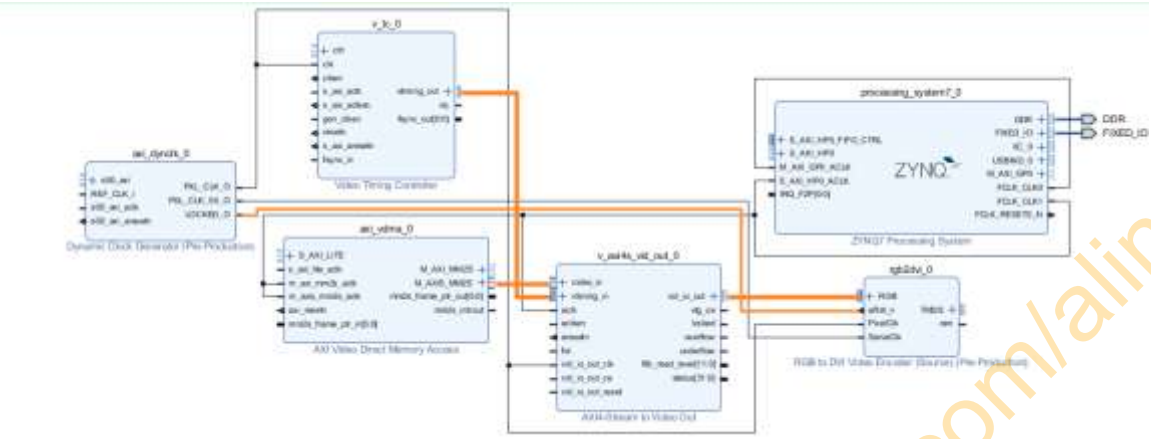




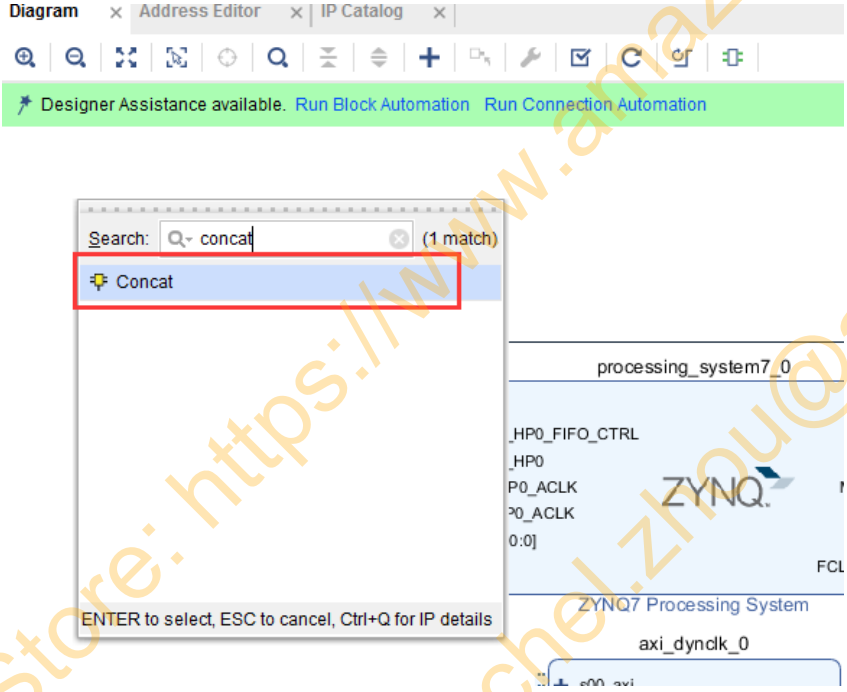
22)Connecting Vivado clock signals that may not be automatically connected

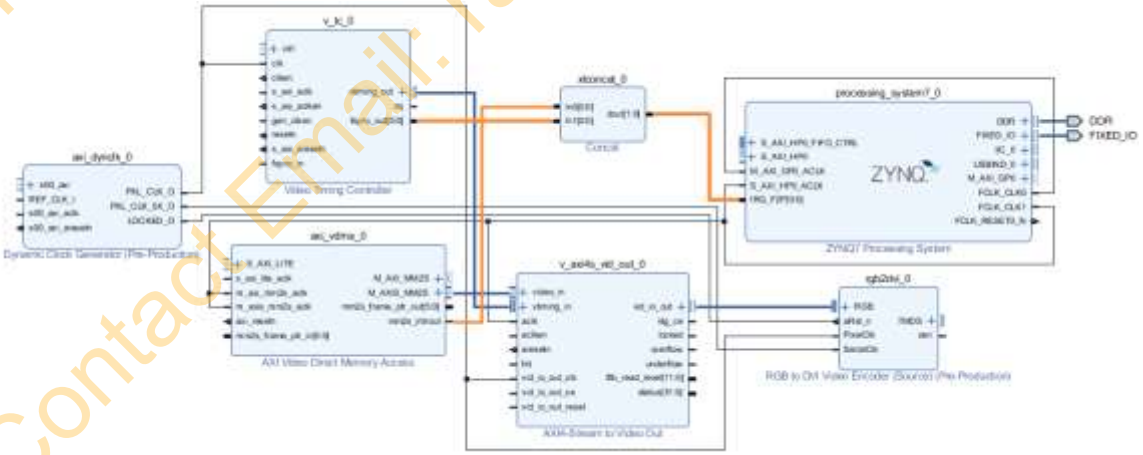


23) Connect some other key signals

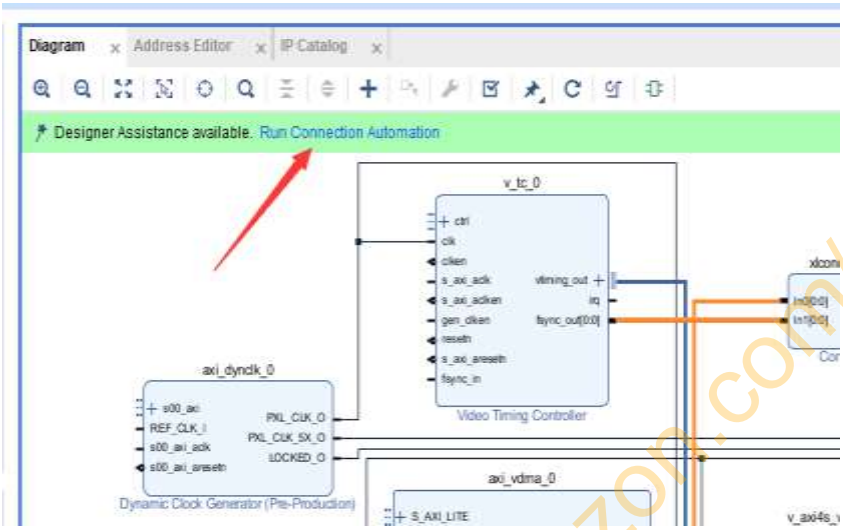


24)To connect to the interrupt signal, you need to add a Concat IP first for signal connection.

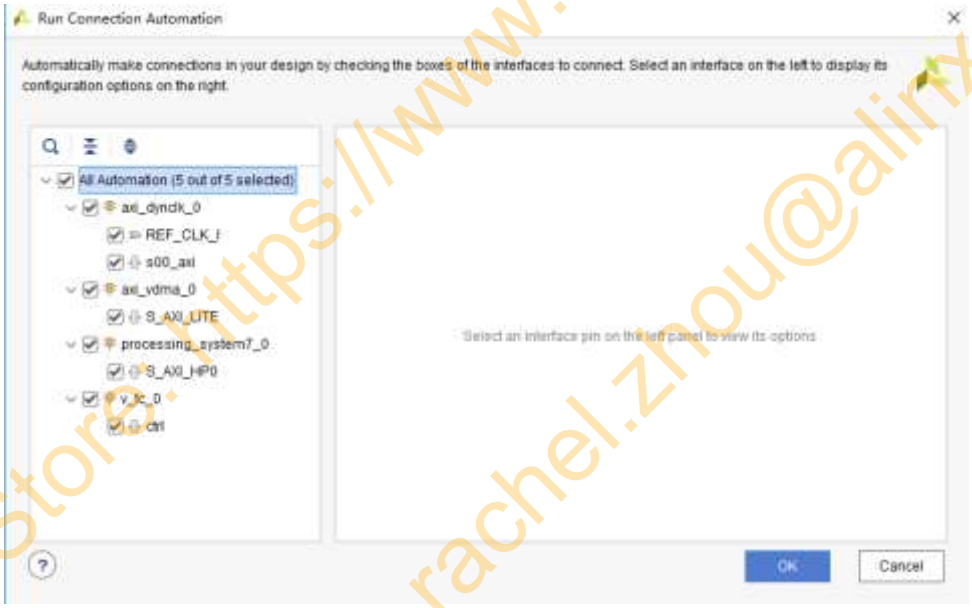




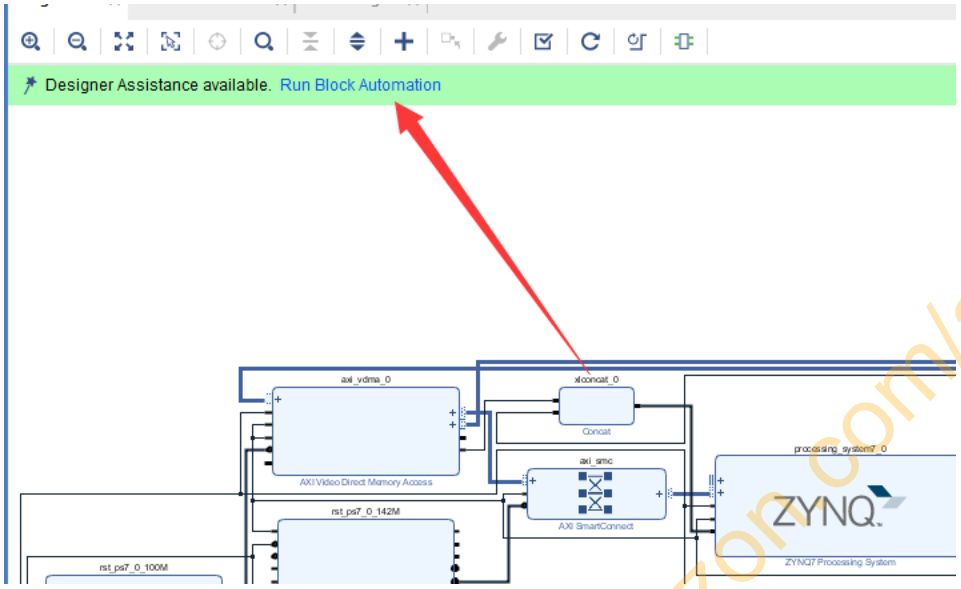
25)Complete the remaining wire connections using Vivado auto-connect



26) Select all modules to connect automatically

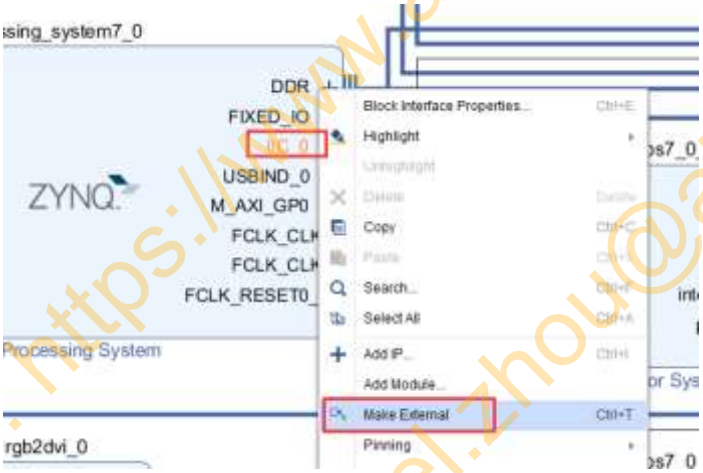


27) Run "Run Block Automation to complete some necessary port export

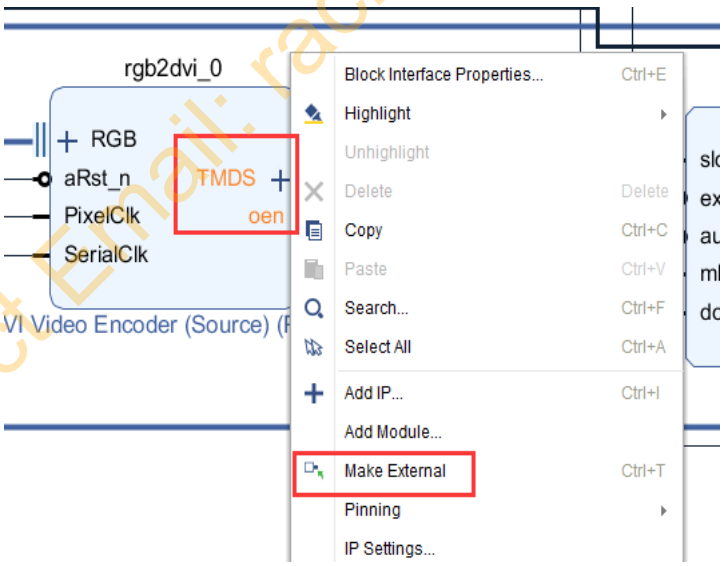


28) Choose the port we need to export

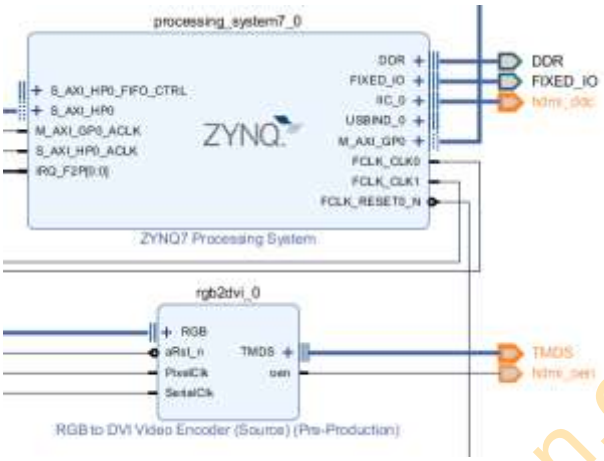
29) Export IIC_0 port



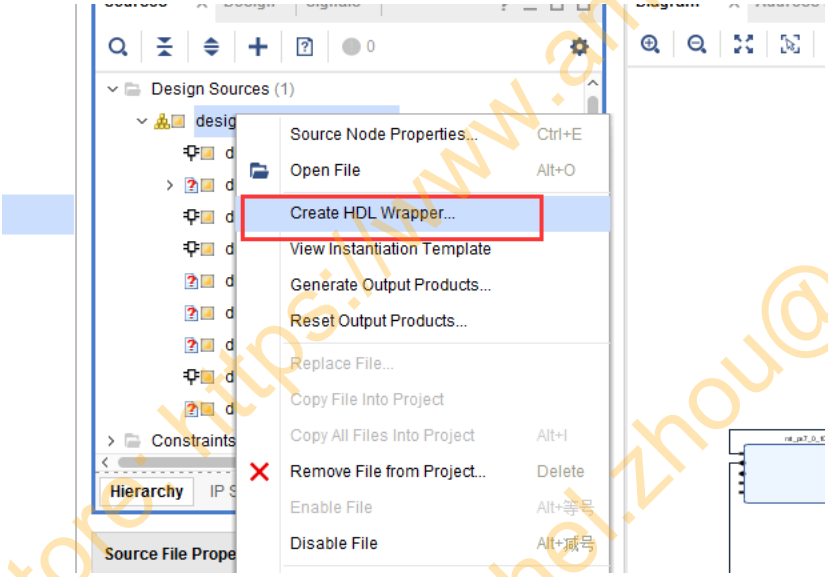
30) Export encoder port TMDS and oen



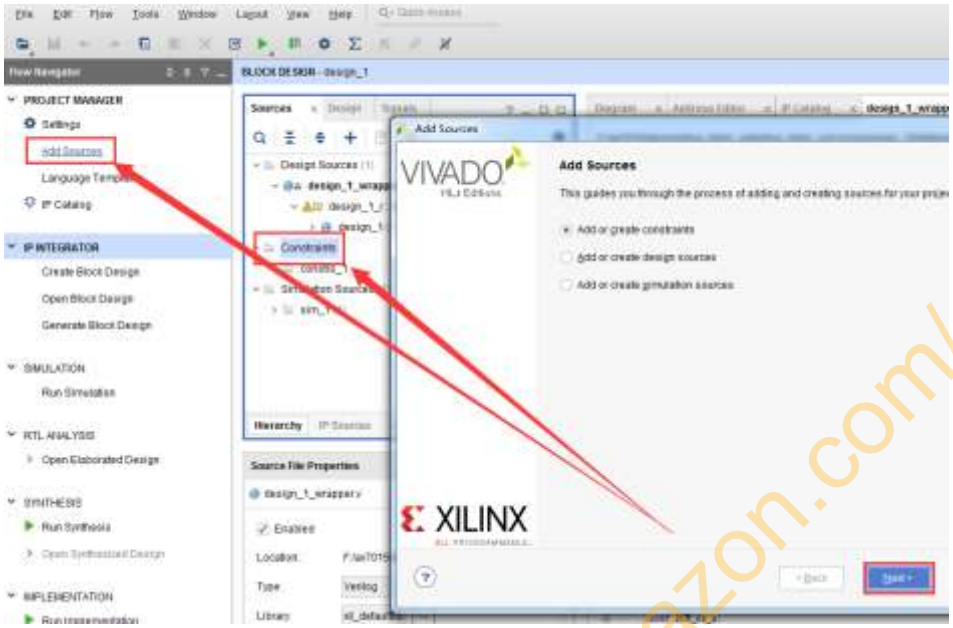
31) Modify the name of another port



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



33) Add the xdc file of HDMI output and bind the pins



34)The contents of the xdc file are as follows

```
set_property IOSTANDARD TMDS_33 [get_ports TMDS_clk_n]
set_property PACKAGE_PIN N18 [get_ports TMDS_clk_p]
set_property IOSTANDARD TMDS_33 [get_ports TMDS_clk_p]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_n[0]}]
set_property PACKAGE_PIN V20 [get_ports {TMDS_data_p[0]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_p[0]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_n[1]}]
set_property PACKAGE_PIN T20 [get_ports {TMDS_data_p[1]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_p[1]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_n[2]}]
set_property PACKAGE_PIN N20 [get_ports {TMDS_data_p[2]}]
set_property IOSTANDARD TMDS_33 [get_ports {TMDS_data_p[2]}]
#set_property PACKAGE_PIN Y19 [get_ports {hdmi_hpd_tri_i[0]}]
#set_property IOSTANDARD LVCMOS33 [get_ports {hdmi_hpd_tri_i[0]}]
set_property PACKAGE_PIN V16 [get_ports hdmi_oen]
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_oen]
set_property PACKAGE_PIN R18 [get_ports hdmi_ddc_scl_io]
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_ddc_scl_io]
set_property PACKAGE_PIN R16 [get_ports hdmi_ddc_sda_io]
set_property IOSTANDARD LVCMOS33 [get_ports hdmi_ddc_sda_io]
```

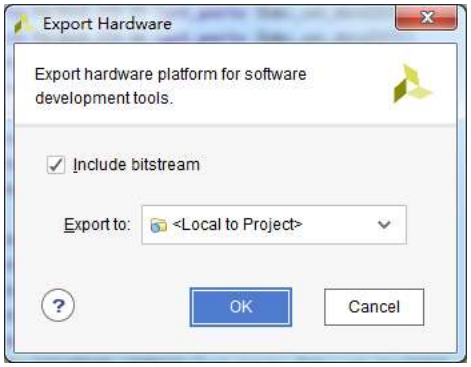
35) Compile and generate bit file

Software engineer work content

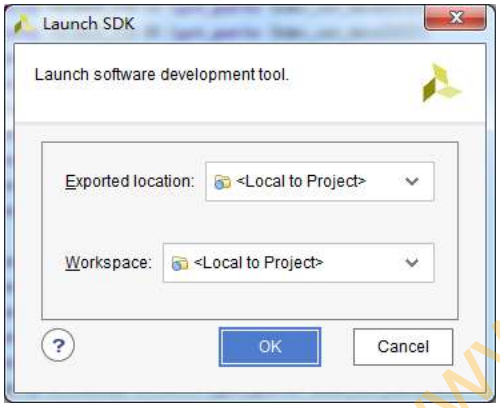
The following is the responsibility of the software engineer.

Part 12.2: SDK software writing and debugging

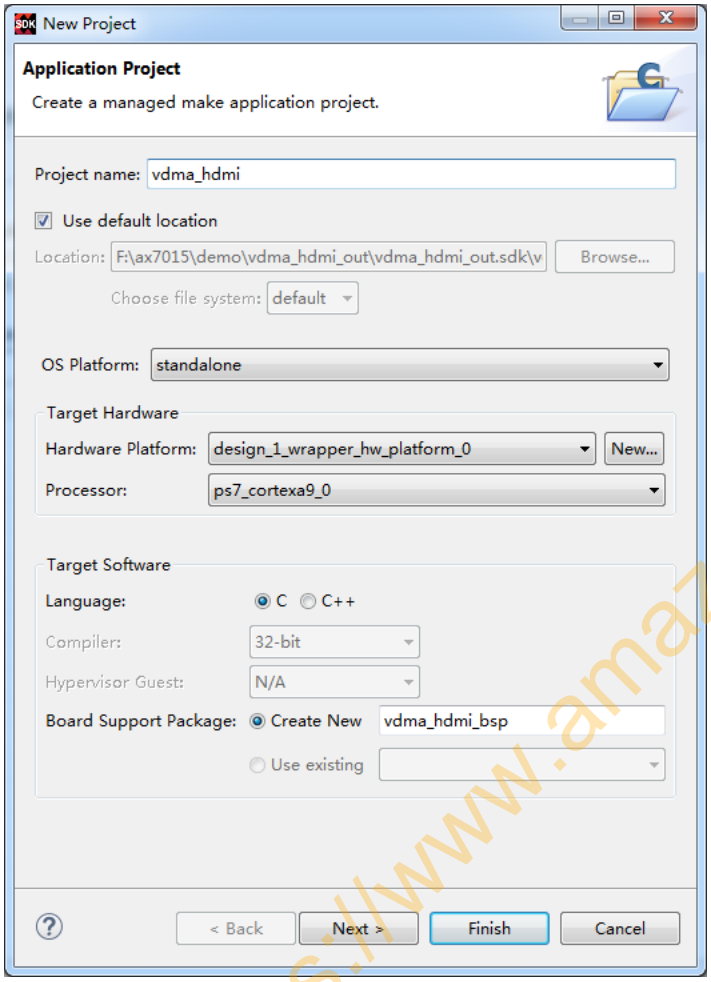
1) Export hardware



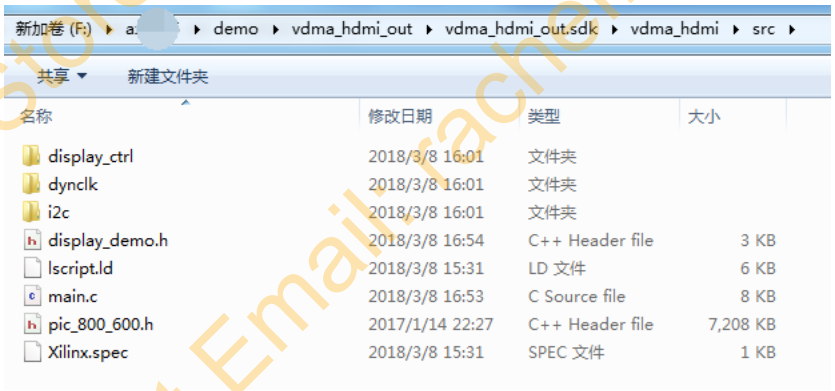
2) Run SDK



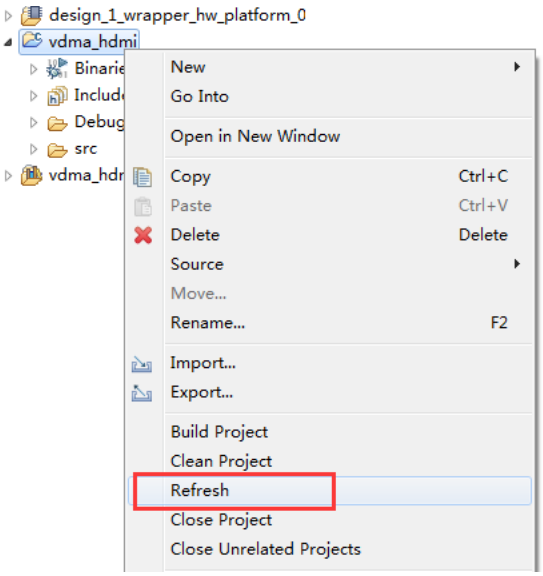
3) Create a APP named “vdma_hdmi”



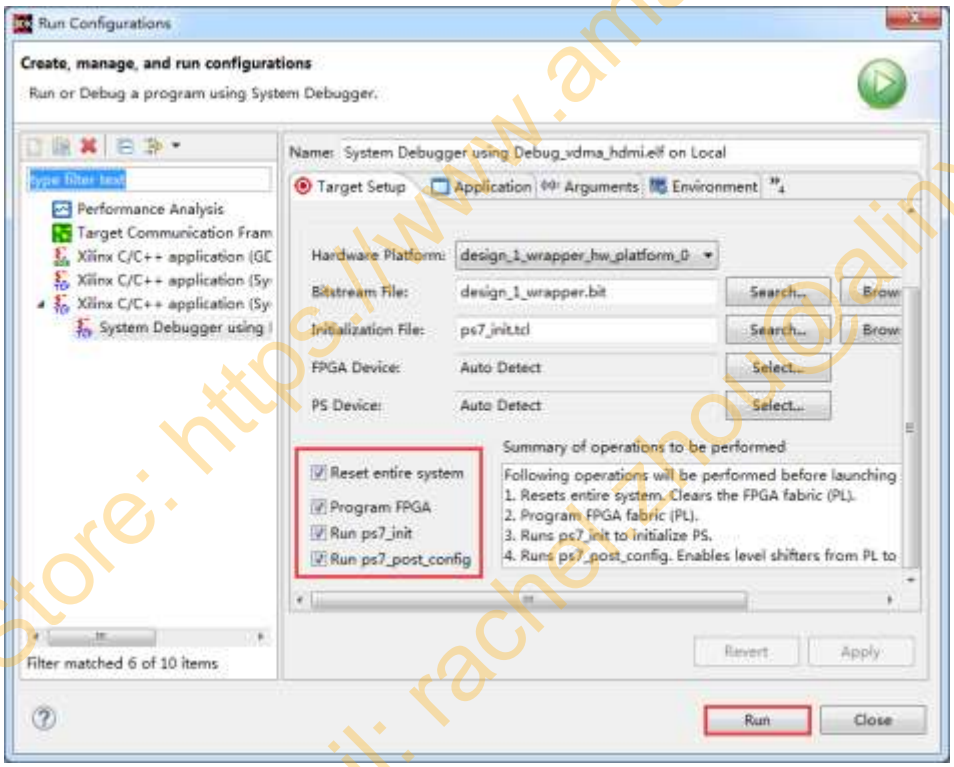
4) Since there are many program files, the source code of the routines is directly copied. Delete the files in the src directory and use the src directory file of the routine instead.



5) Refresh under the SDK



6) Connect the HDMI output port to the monitor, compile and run



7) The monitor shows a picture

8) The display shows a picture

Part 13: Curing Programes

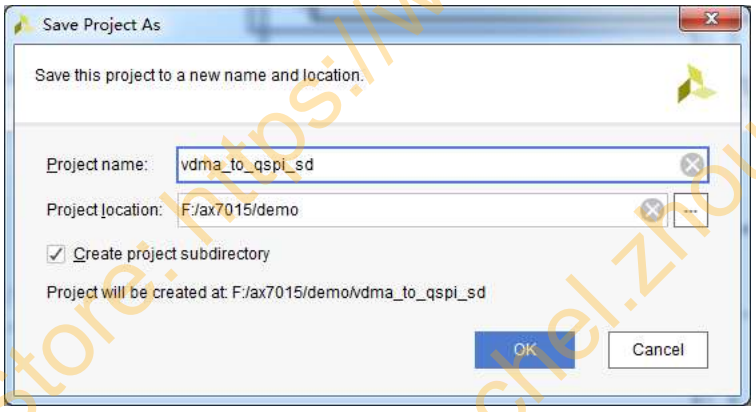
Experiment Vivado project is " vdma_to_qspi_sd "

After doing so many experiments, all are debugged by JTAG. How to put the program on the SD card or burn it into QSPI Flash? First, the PL must have a PS configuration, so it can't be directly burned to Flash like the previous FPGA burning method. This experiment explains how to cure the program.

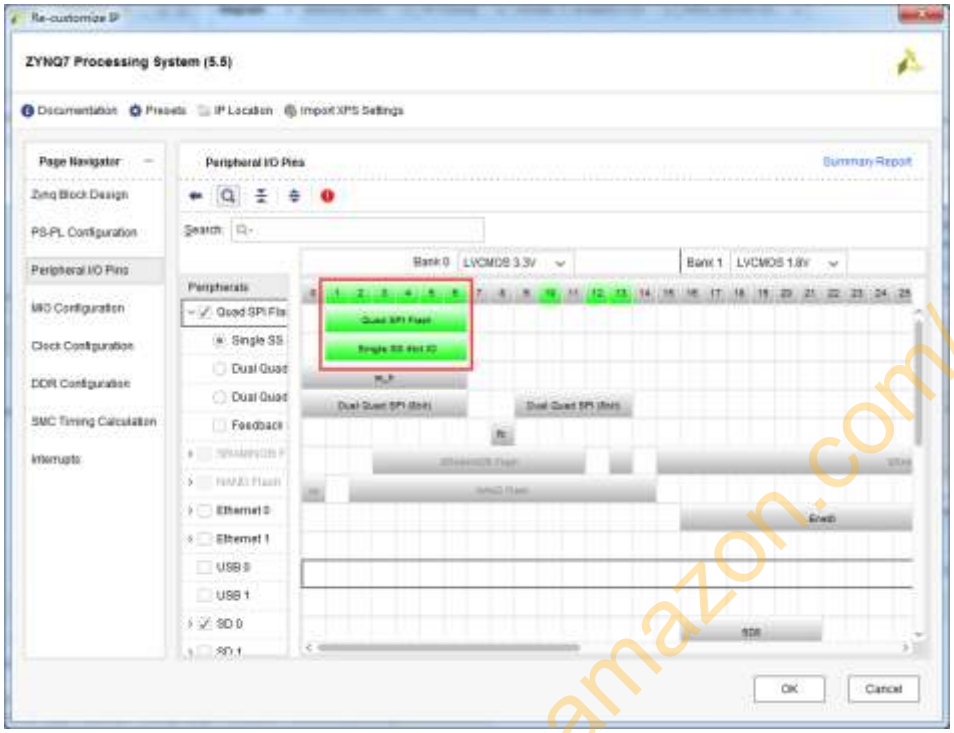
Part 13.1: Building a Vivado project

This experiment selected the VDMA test project to cure. When setting up the VDMA test project, we did not enable QSPI and SD card. The QSPI or SD card must be enabled to cure the program

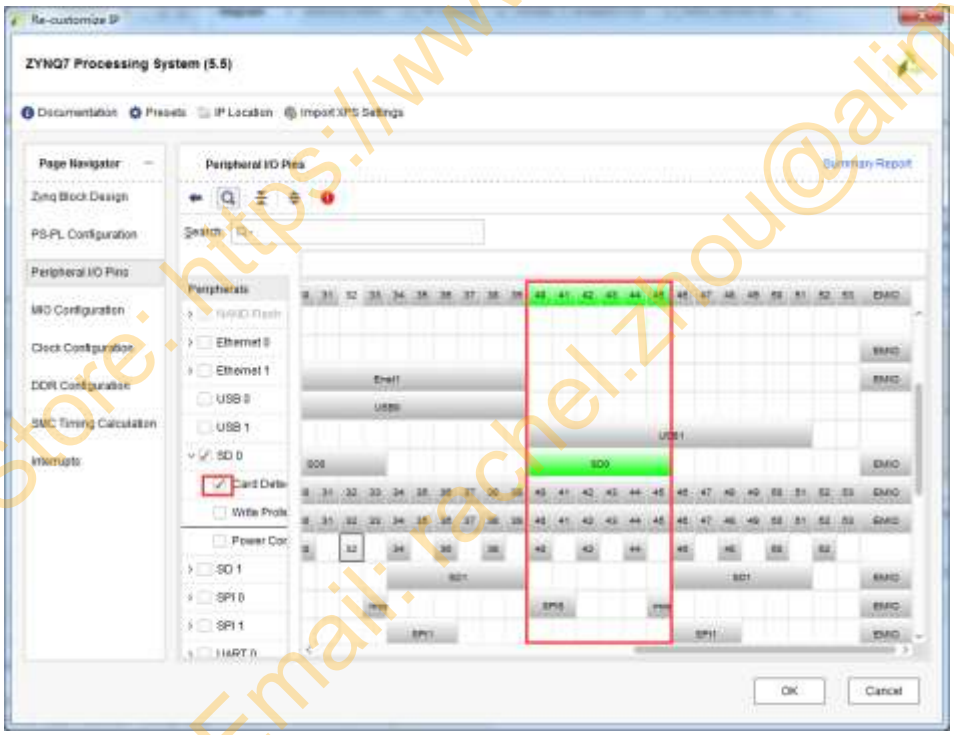
1) Save a copy of the VDMA test project for the curing program experiment and change the name to "vdma_to_qspi_sd"



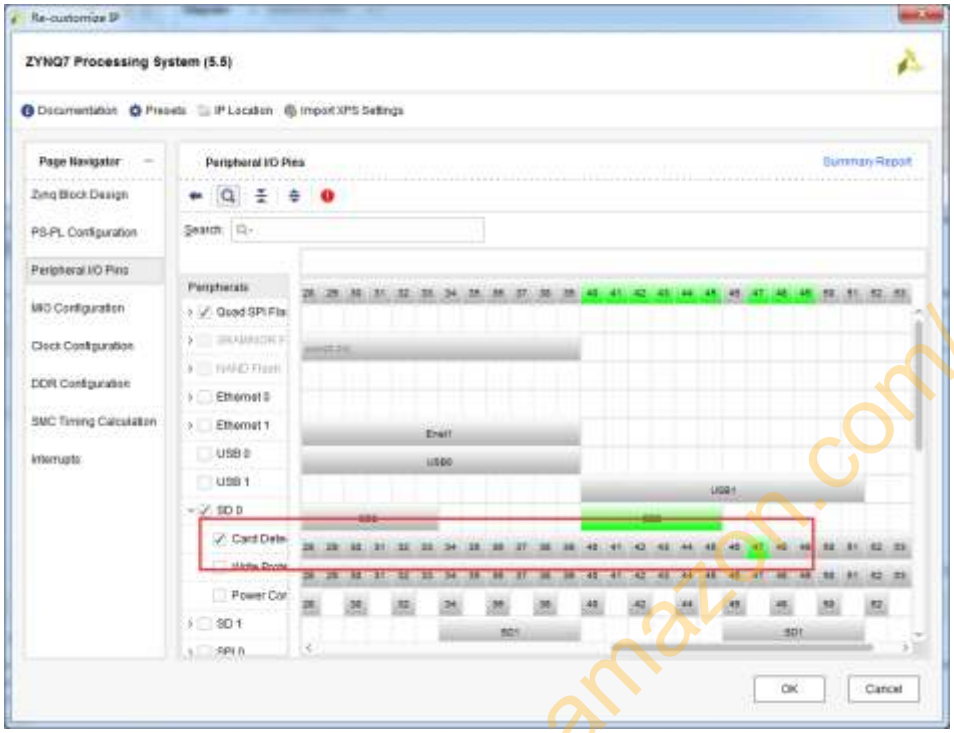
2) Add QSPI, use MIO1-6



3) Add SD0 controller, use MIO40-45, use TF card interface



4) Add card detection pin MIO47 for SD card

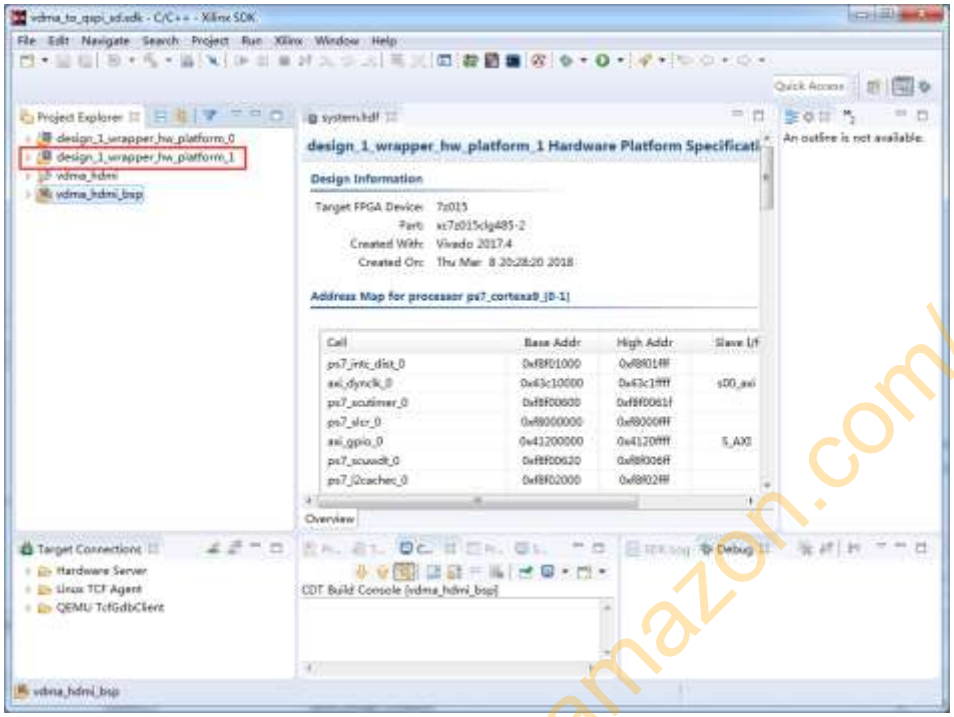


5) Save the design, compile and generate the bit file, and export the hardware again

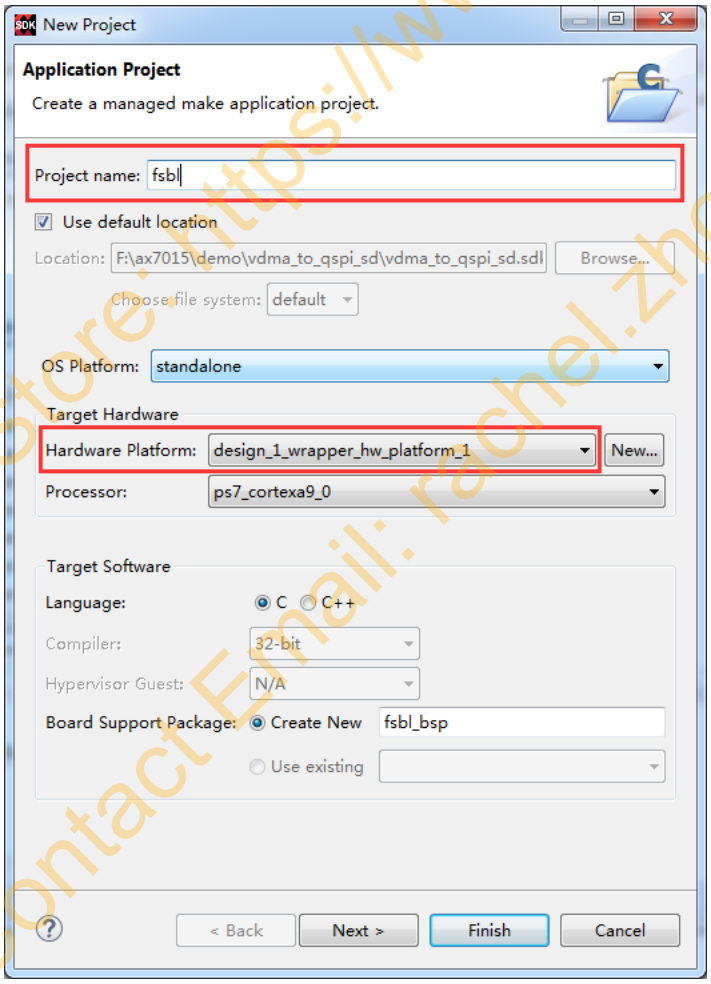
Part 15.2: Generate FSBL

The FSBL is a secondary boot loader that completes MIO allocation, DDR controller initialization, SD, QSPI controller initialization, configures the FPGA, and then loads the user program.

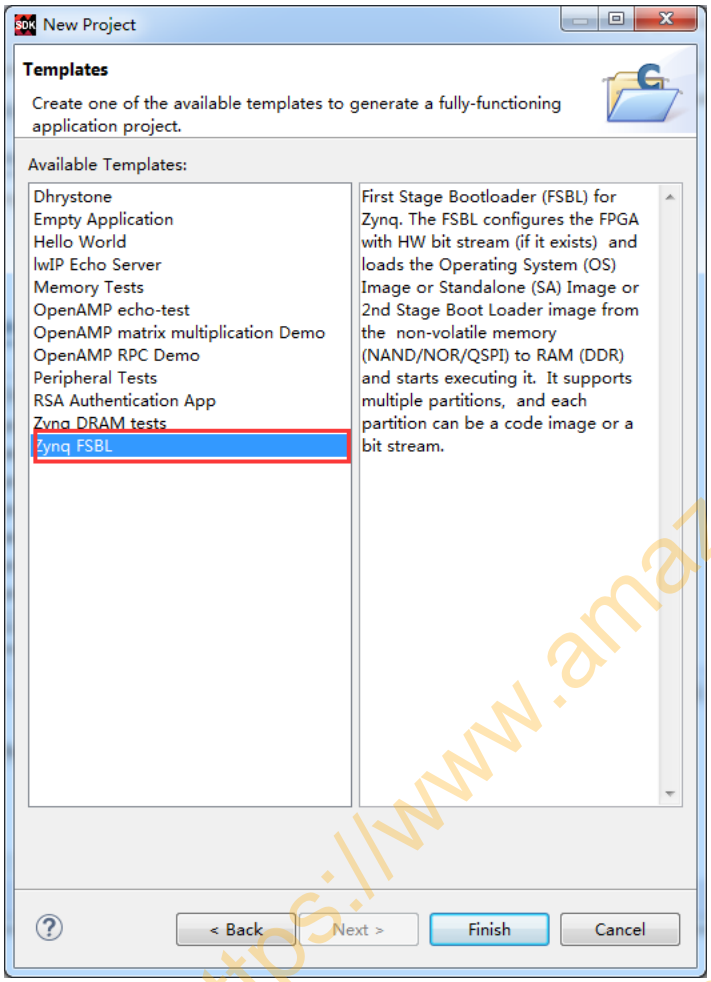
1) Start the SDK software, because it is copied from other projects, it used to be the SDK project. Because of the path change, there is another “hw_platform_1”



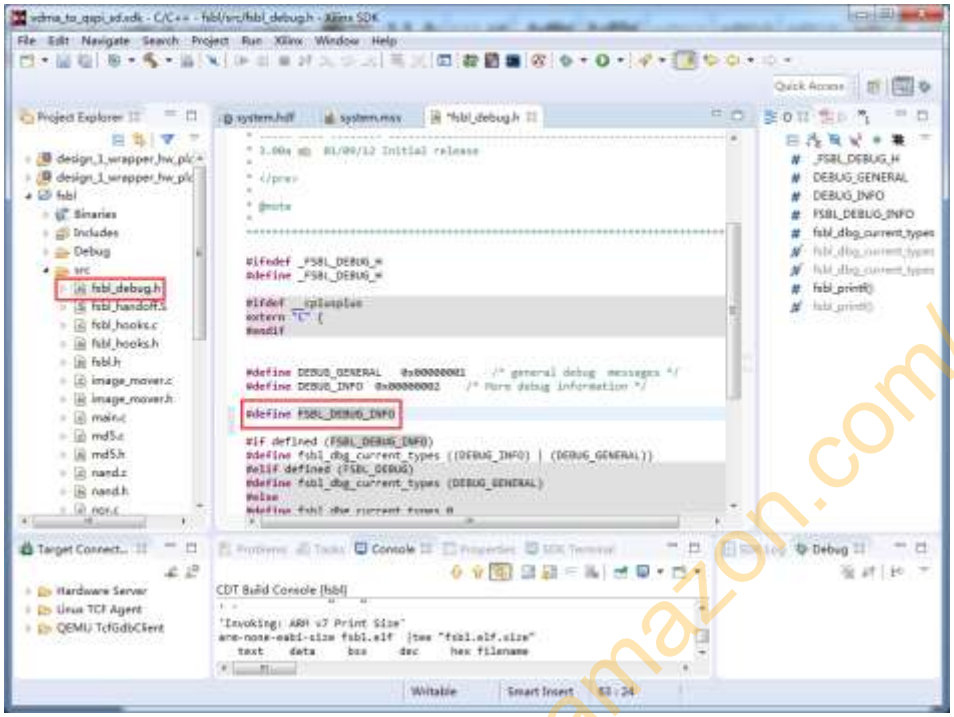
2) Create a new app called “fsbl”, pay special attention to the hardware platform to choose the latest one.



3) Template selection “ZynqFSBL”



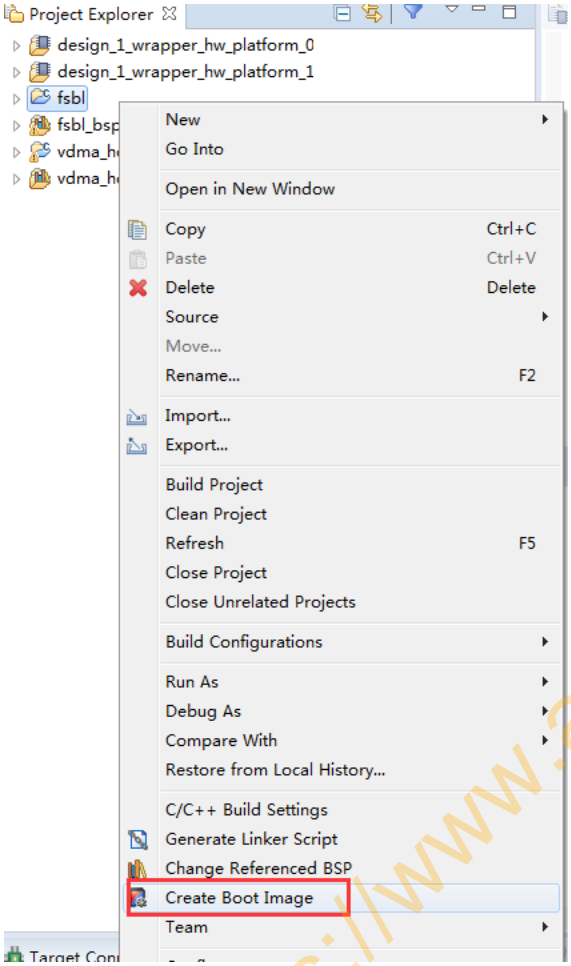
4) Add debug macro definition “FSBL_DEBUG_INFO”, you can start some output status information of “FSBL”, which is good for debugging, but it will lead to longer startup time.



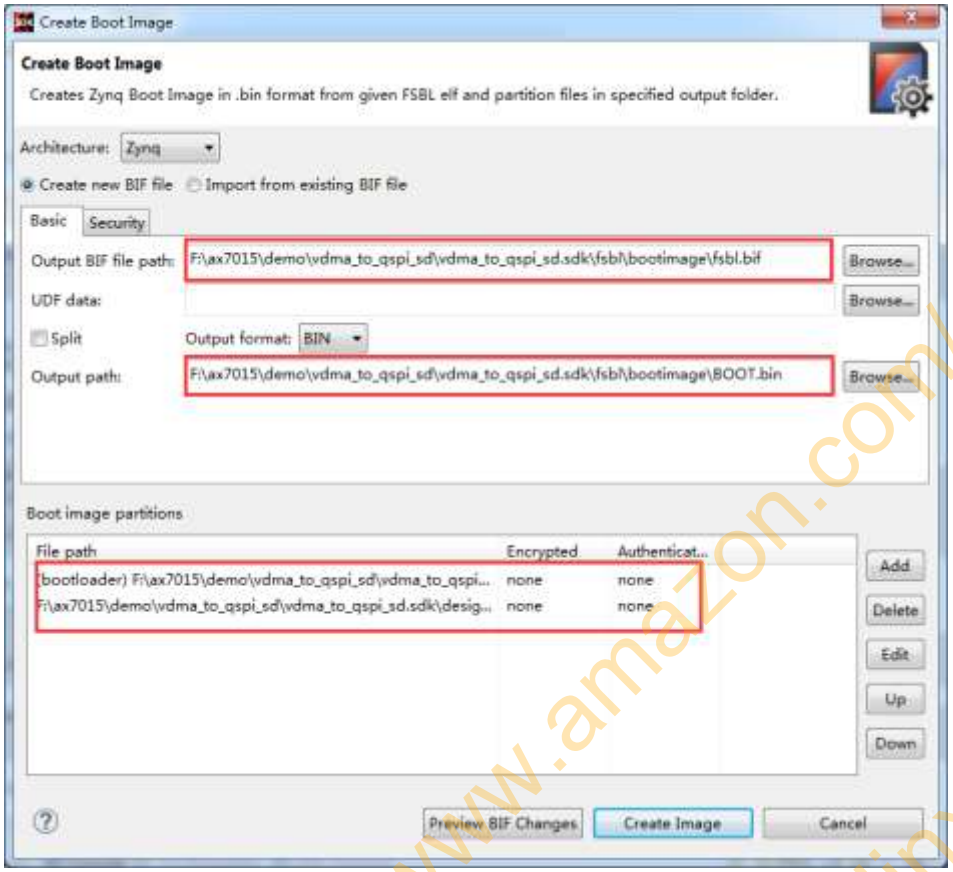
5) The SDK will compile automatically by default, generating the fsbl.elf file.

Part 13.3: Create a BOOT file

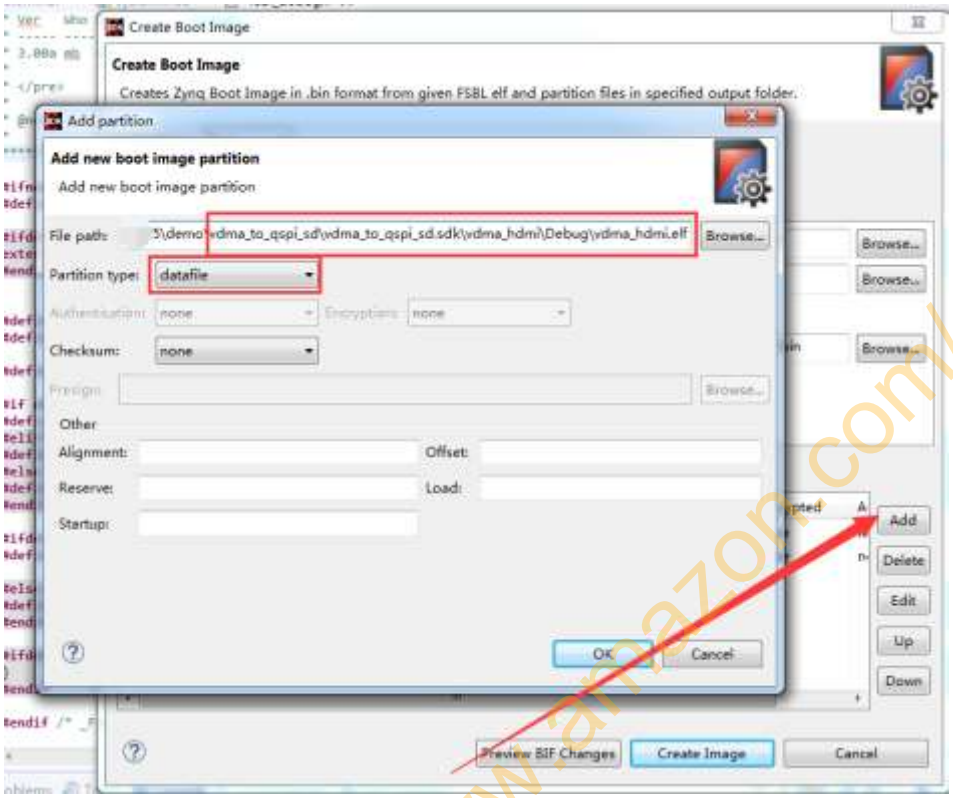
1) Select the “fsbl” project and right click on “Create Boot Image”



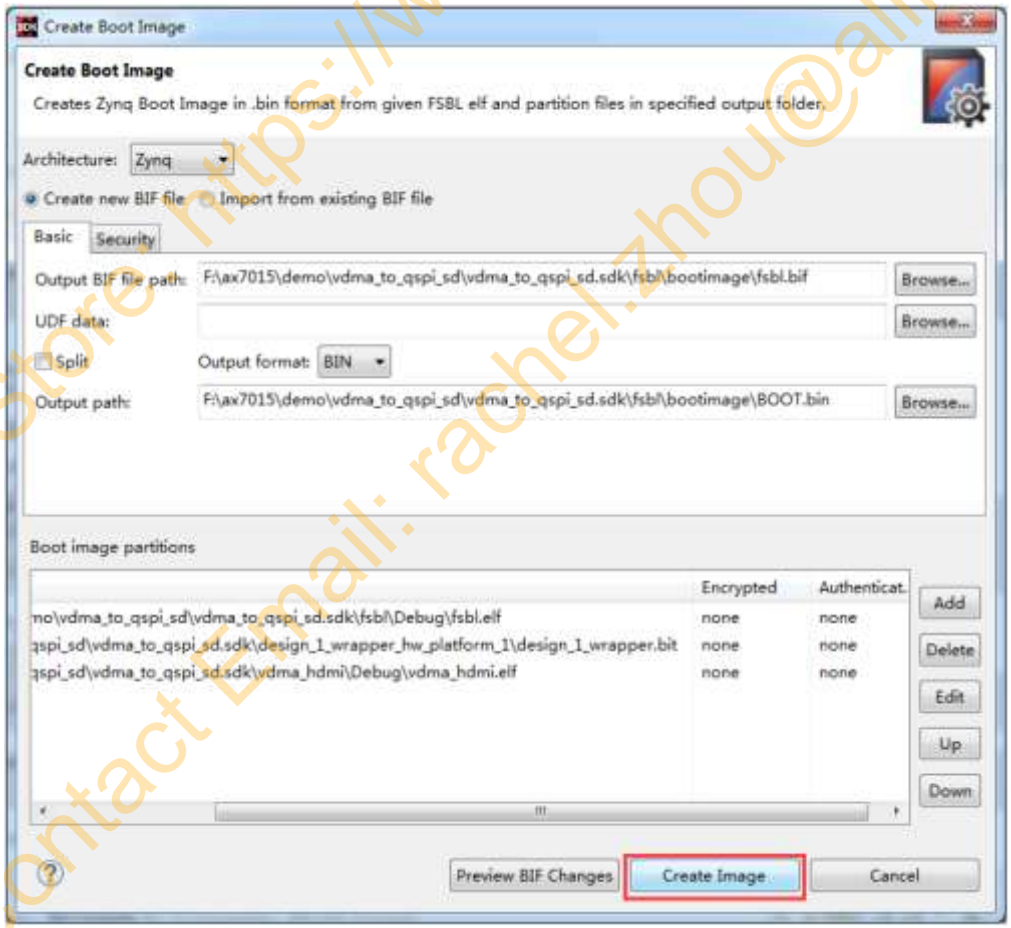
2) The generated BIF file path can be seen in the pop-up window. The BIF file is the configuration file for generating the BOOT file, and the generated BOOT.bin file path. The BOOT.bin file is the startup file we need and can be put into the SD card. It can also be written to QSPIFlash.



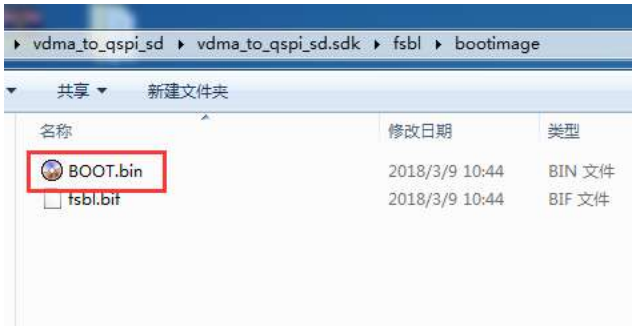
3) There is a file to be synthesized in the “Boot image partitions” list. The first file must be the boot loader file, which is the “fsbl.elf” file generated above, and the second file is the FPGA configuration file. Now click Add to add our VDMA test program “vdma_hdmi.elf”



4) Click “Create Image” to generate Image

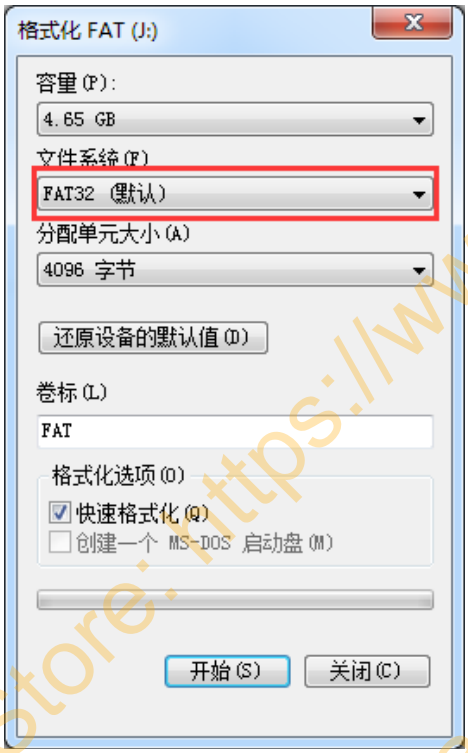


5) The BOOT.bin file can be found in the generated directory.



Part 13.4: SD card startup test

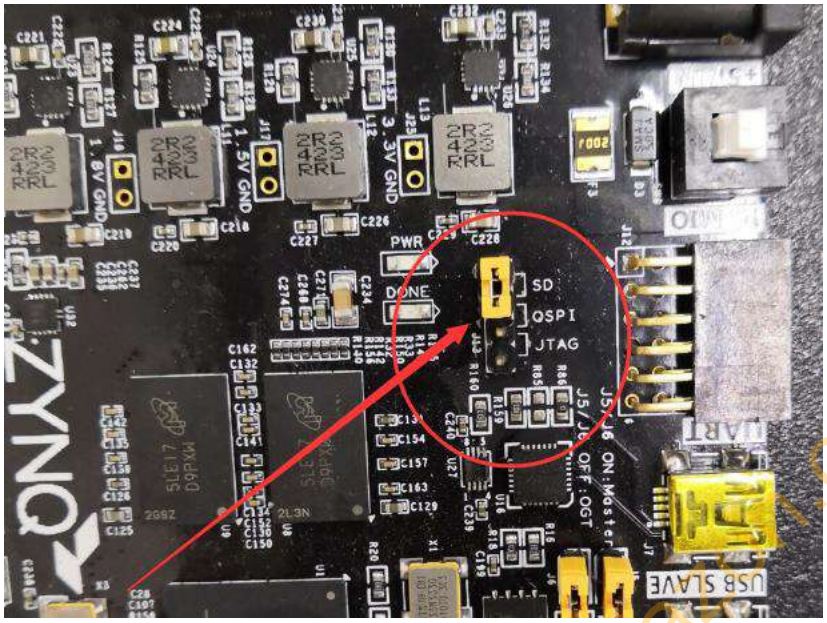
1) Format SD card, only formatted as FAT32 format, other formats cannot be started



2) Put the BOOT.bin file in the root directory

3) SD card inserted into the SD card slot of the FPGA development board

4) Start mode adjustment to SD card boot

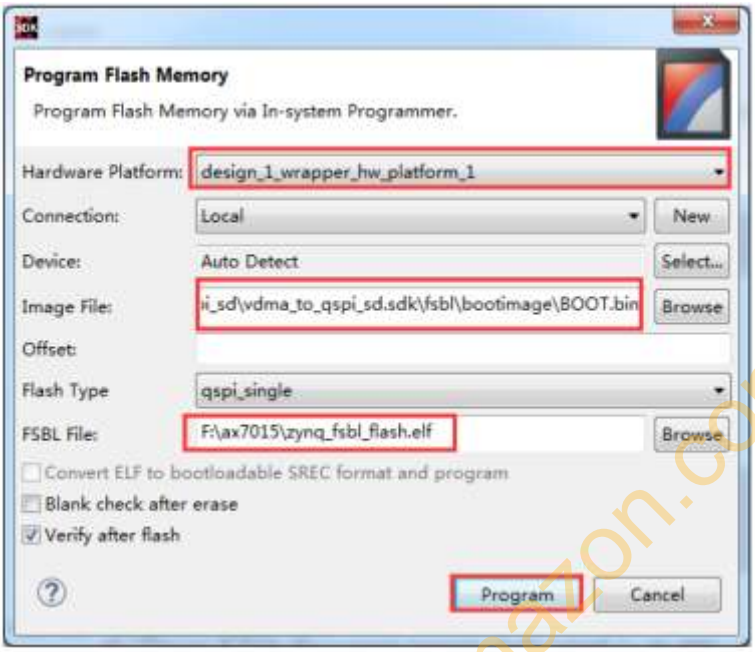


5) Plug in the HDMI display and power on the board. You can see that the display shows the picture of the kitten.

Part 13.5: QSPI startup test

1) In the SDK menu Xilinx -> Program Flash

2) Hardware Platform selects the latest version, “Image File” selects “BOOT.bin” to be programmed, “FSBL file” selects ALINX customized special version “fsbl.elff”, only use this “fsbl” to burn

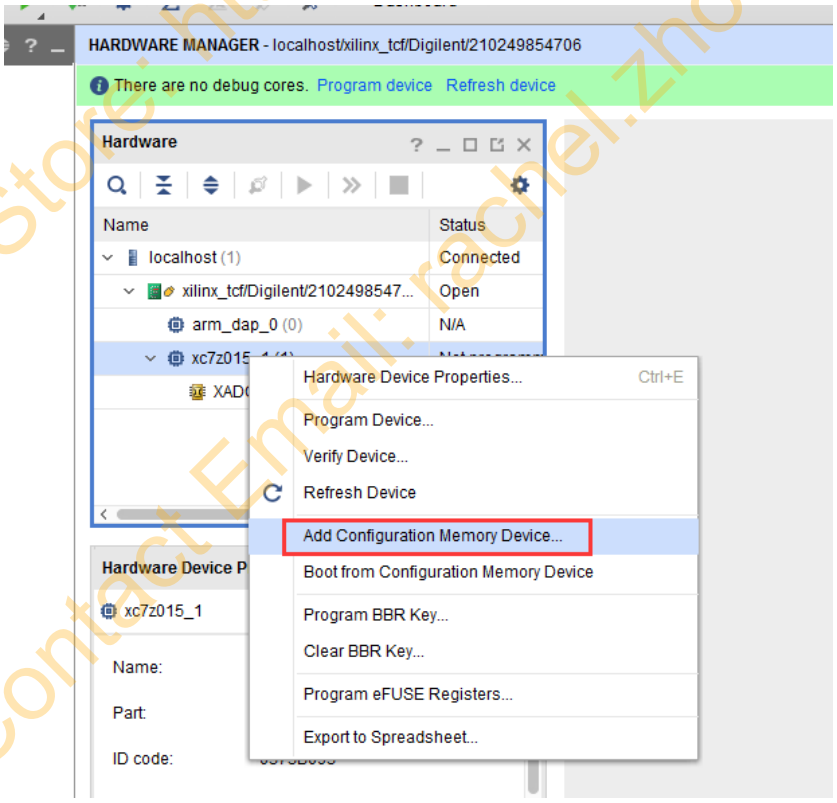


3) Click Program and wait for the programming to complete.

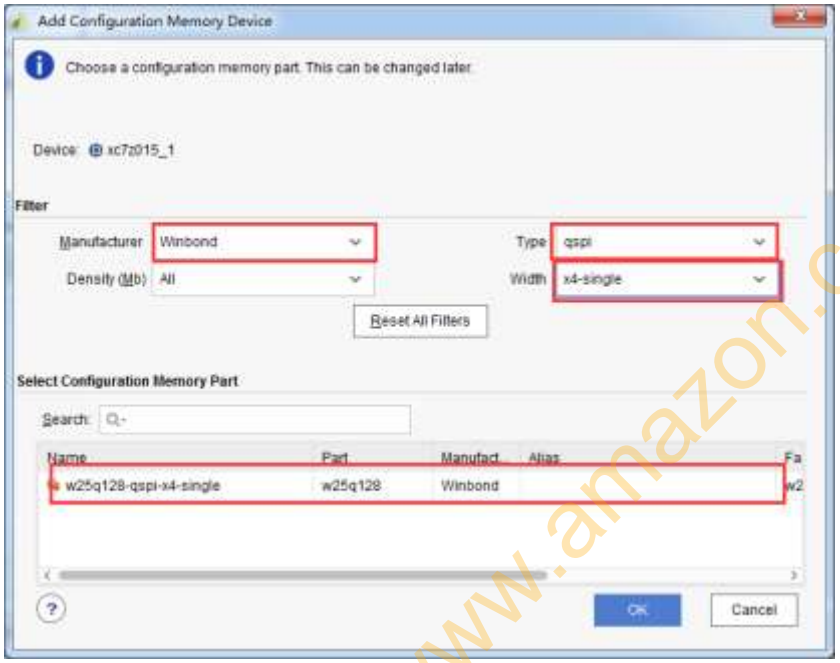
4) Set the startup mode to QSPI, start it again, you can see that the display has a display output.

Part 13.6: Vivado writes QSPI

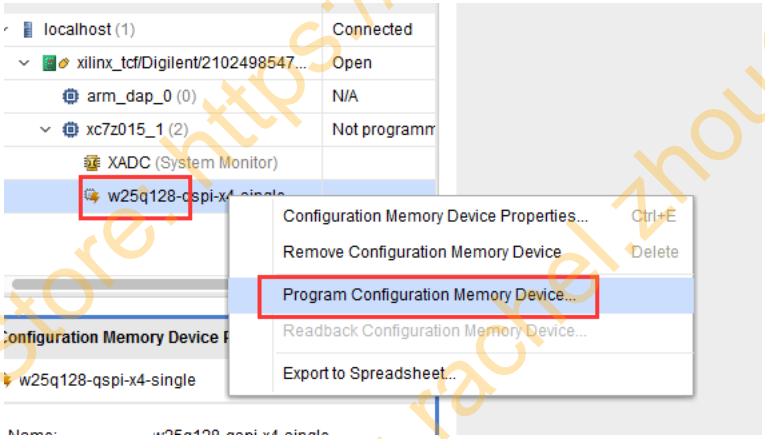
1) Select the device under “HARDWARE MANGER”, right click “Add Configuration Memory Device”



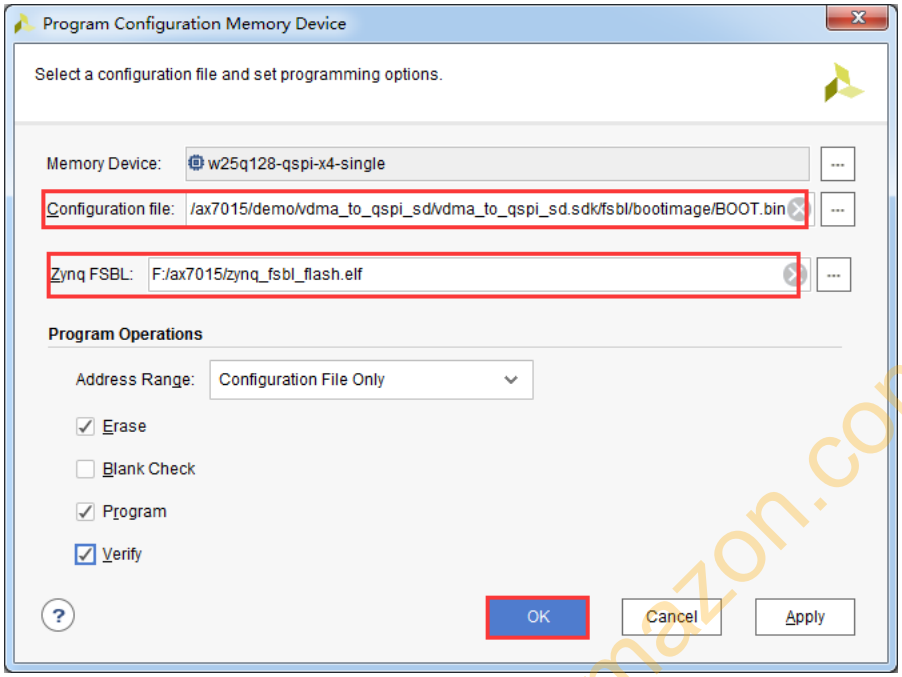
2) Choose “Winbond”, type select “qspi”, width select “x4-single”, this time a “w25q128” appears, the FPGA development board uses “w25q256”, but does not affect the burning



3) Right click to select the programming file



4) Select the file to be programmed and the ALINX customized fsbl file, you can program it. If it is not in JTAG startup mode, the software will give a warning, so it is recommended to set to JTAG startup mode, when programming QSPI.

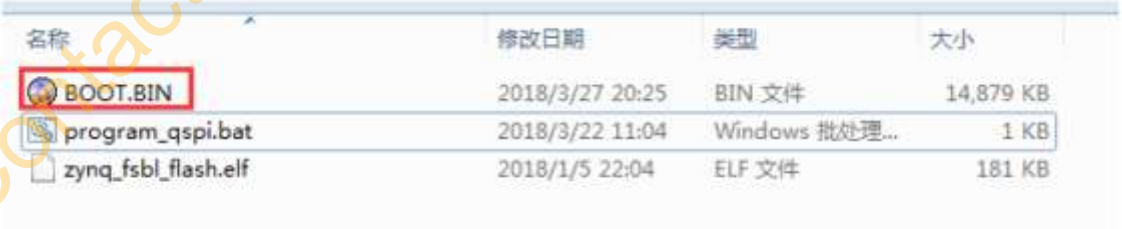


Part 13.7: Quickly program QSPI using batch files

1) Create a new “program_qspi.txt” text file, the extension is changed to bat, the content is filled in as follows, where “set XIL_CSE_ZYNQ_DISPLAY_UBOOT_MESSAGES=1” set to display the “uboot” print information during programming “C:\Xilinx\SDK\2017.4\bin\program_flash” for our tool path According to the installation path, “-f” is the file to be programmed, and “-fsbl” is the fsbl to be used for programming. The “-blank_check –verify” is the checksum option.

```
set XIL_CSE_ZYNQ_DISPLAY_UBOOT_MESSAGES=1
call C:\Xilinx\SDK\2017.4\bin\program_flash -f BOOT.bin  -fsbl
zynq_fsbl_flash.elf  -offset 0 -flash_type qspi_single  -blank_check
-verify
pause
```

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



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

Part 14: Install Virtual Machine and Ubuntu System

The latter tutorial involves embedded Linux development and generally requires a Linux operating system host to compile uboot or Linux-kernel. It is the easiest way to install a virtual machine on a Windows operating system and then install the Linux operating system on the virtual machine.

Part 14.1: Virtual machine software installation

The installation software version of the virtual machine we provide is VMware-worksation-full 12.1.1. Users can find it in the information we provide, double-click the “VMware-workstation-full-12.1.1-3770994.exe” icon to start the installation. Because the installation is relatively simple, the specific installation steps are not specifically introduced here. Users only need to click the "Next" button to install according to the default installation items. In the final interface of the installation, we need to select a license to enter a VMware12 serial number.

Once the installation is complete, there is an icon for VMware Workstation Pro on the desktop.

Part 14.2: Ubuntu installation

Part 14.2.1: install the system

After installing the virtual machine, you must install the Linux operating system on the virtual machine. Due to the simple installation and configuration of the ubuntu Linux desktop operating system, we chose the ubuntu desktop operating system. Due to the simple installation and configuration of the ubuntu Linux desktop operating system, we chose the ubuntu desktop operating system.

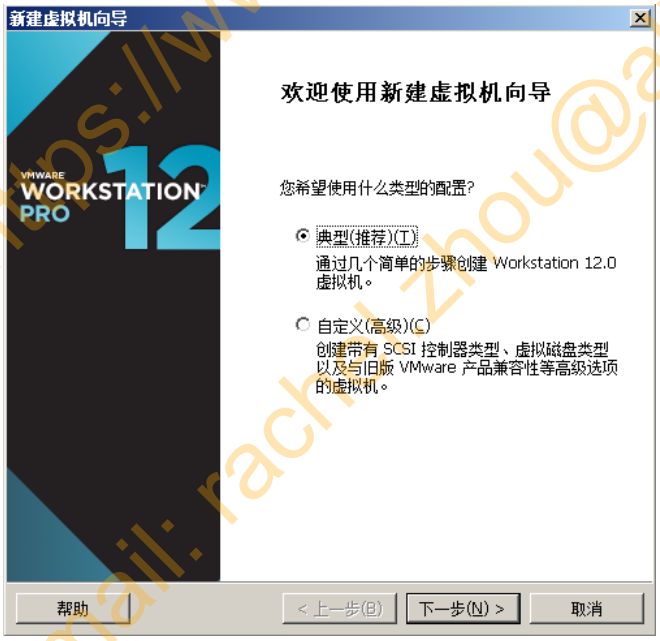
This tutorial uses Ubuntu 16.04.3 LTS 64-bit operating system

If you use other versions, there may be unpredictable errors, please keep the version consistent, please do not upgrade the system.

The ubuntu installation steps are as follows:

1) Double-click the icon for **VMwareWorkstationPro** on your desktop, and click the **Create New Virtual Machine** icon on the **VMware** work interface.

2) Select Typical and click Next

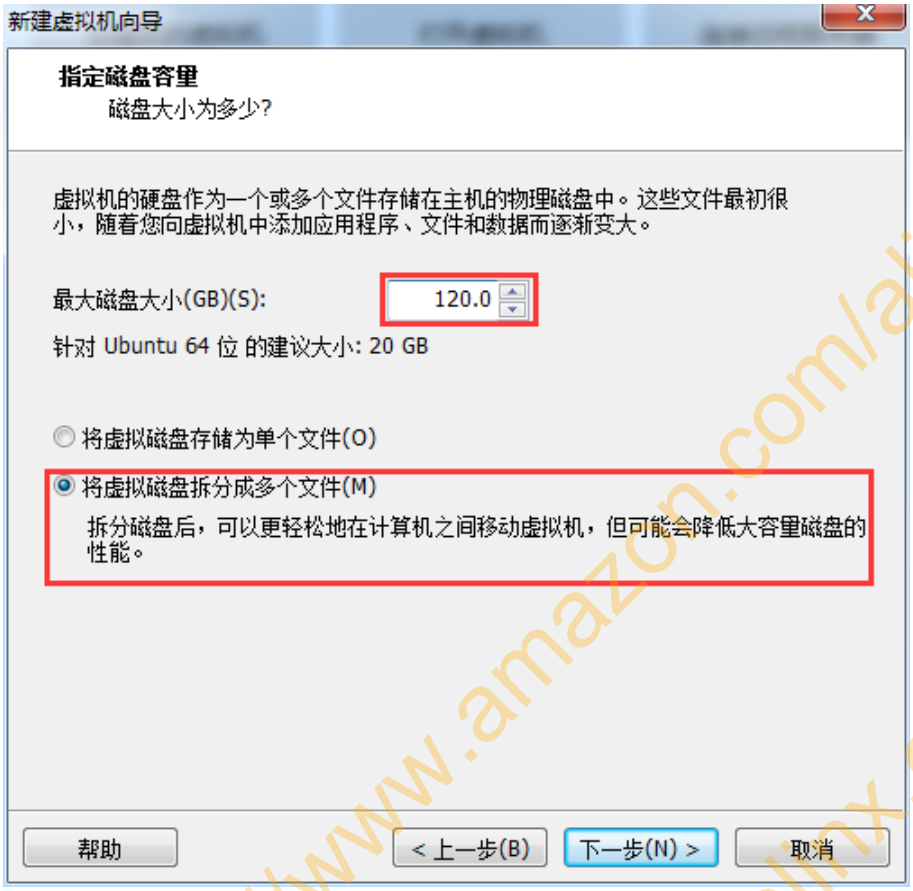


3) Select the "Installer CD Image (iso)" item and click Browse to locate the ubunt CD image file "ubuntu-16.04.3-desktop-amd64.iso".

4) Enter the full name, user and password of the virtual machine in the virtual machine wizard. The full name, username and password can be set by the user.

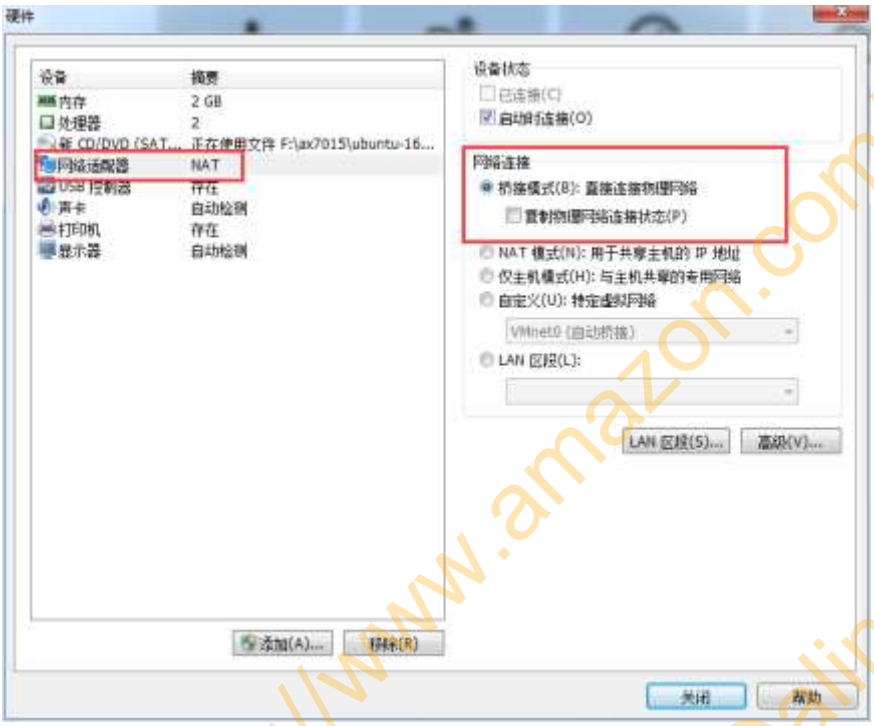
5) The virtual machine name can be modified by itself. The installation location needs to be selected to be installed on a disk with sufficient disk space.

6) To set the maximum disk size to 300G, we need to install the software in the virtual machine, where the reserved space is larger. Users can choose the appropriate space size according to their hard disk space. It is recommended to be 300G or more.

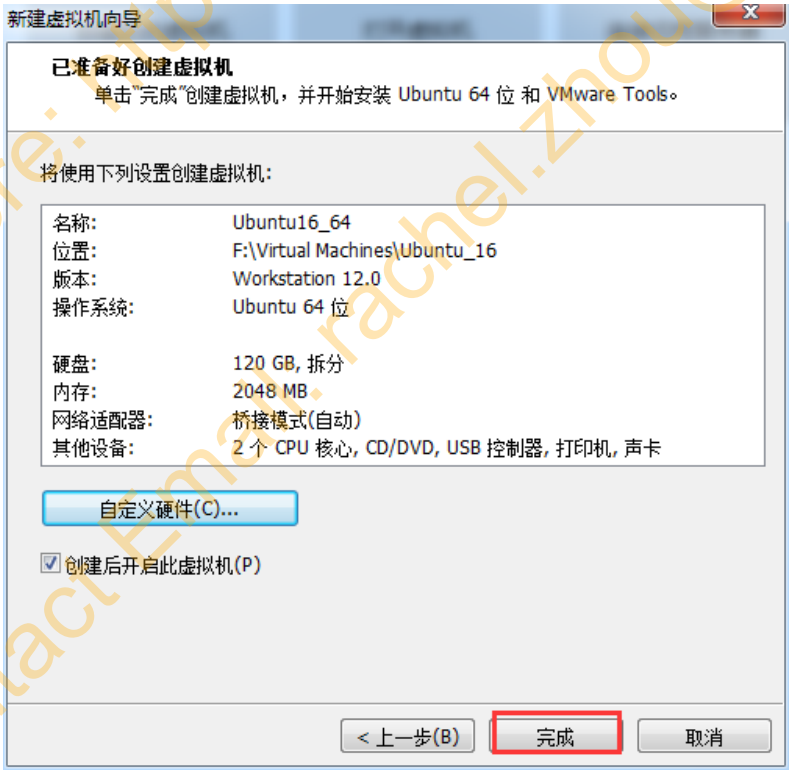


7) Choose custom hardware

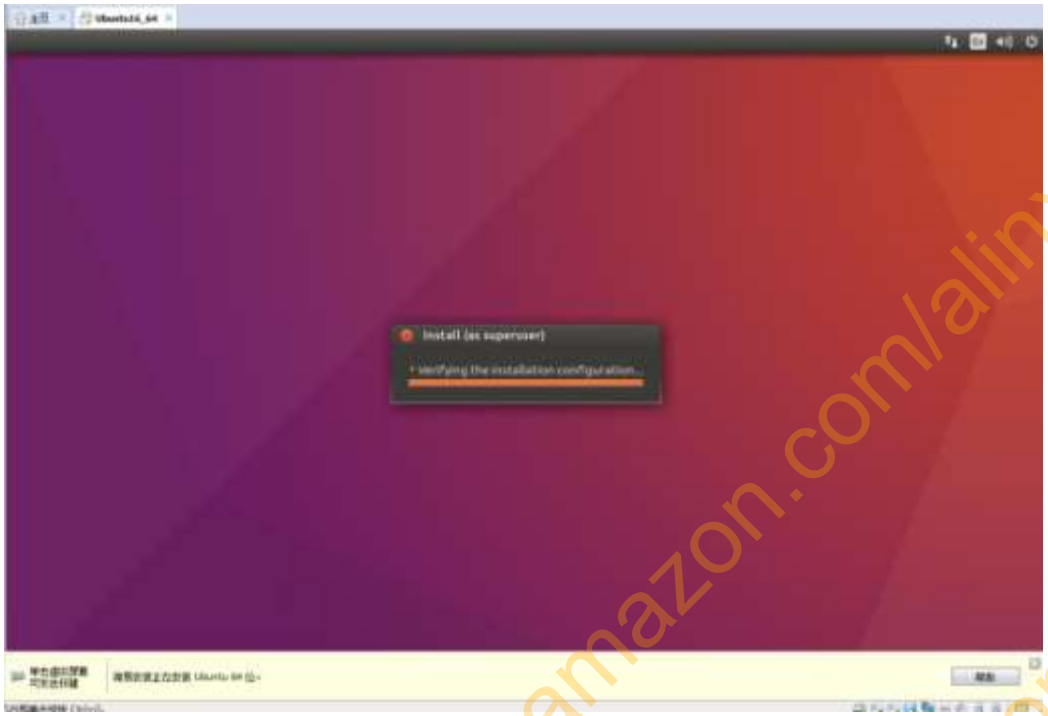
8) You can modify the memory size and processor core according to the modification, network adapter options, network connection to select the bridge mode



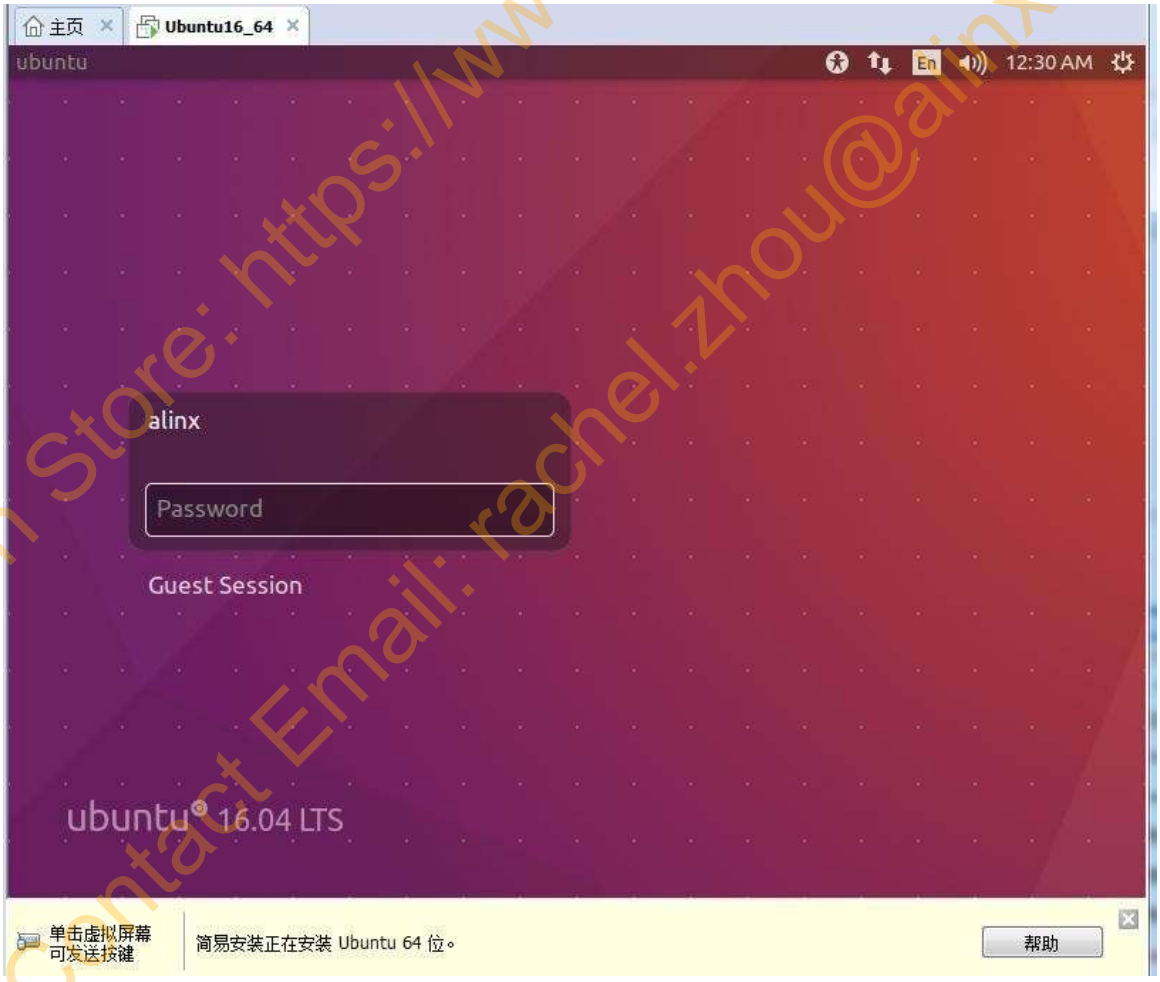
9) Click Finish to start installing Ubuntu.



10)The installation process is slow, waiting for a while



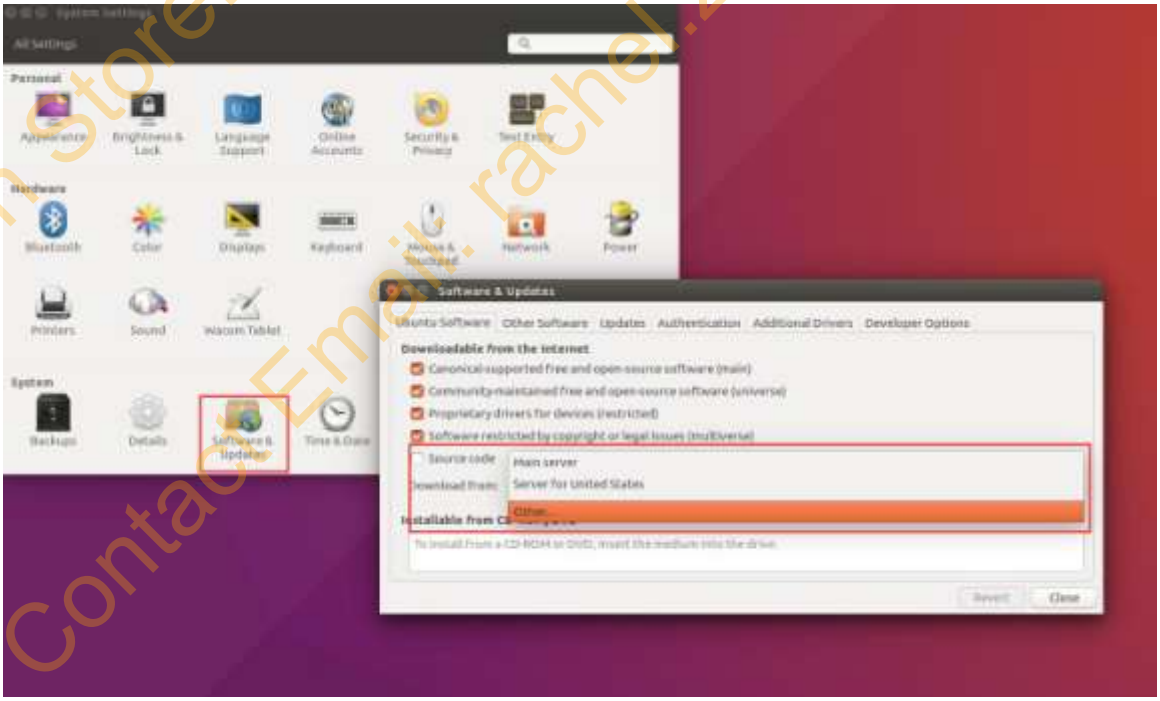
11)Enter the system after installation is complete



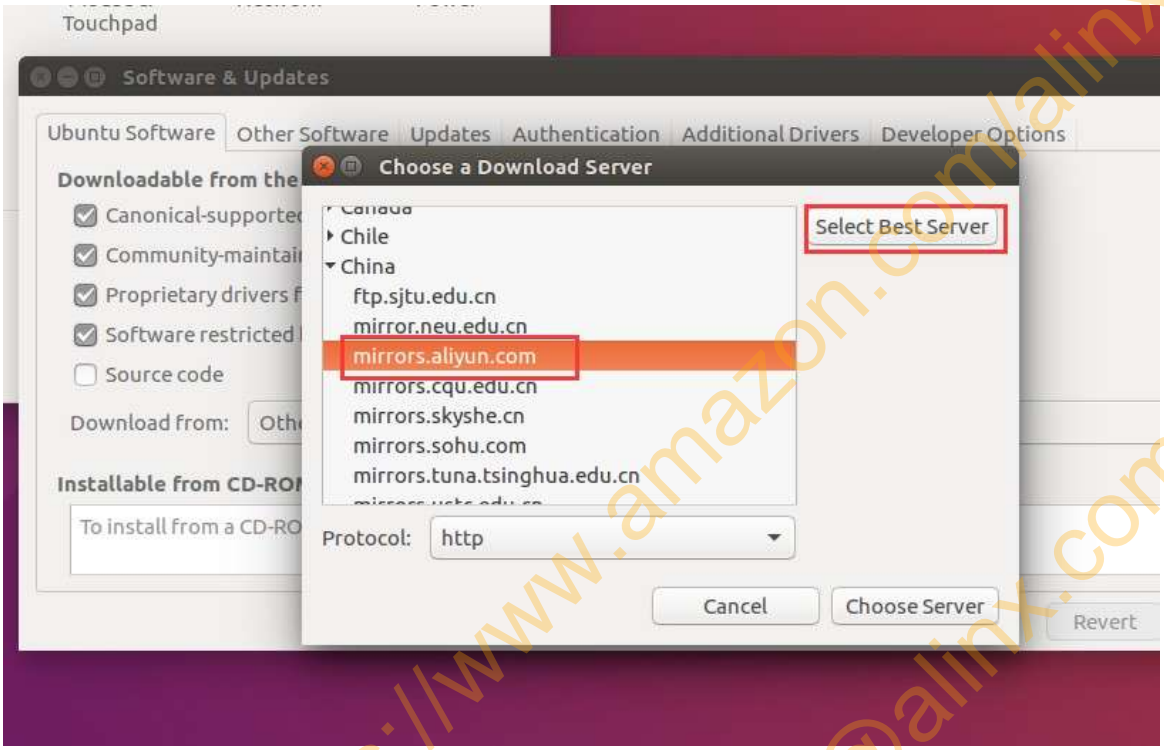
Part 14.2.2: Modify the Software Source Server

1) In order to install the software conveniently, we need to set the software source, click on the system settings.

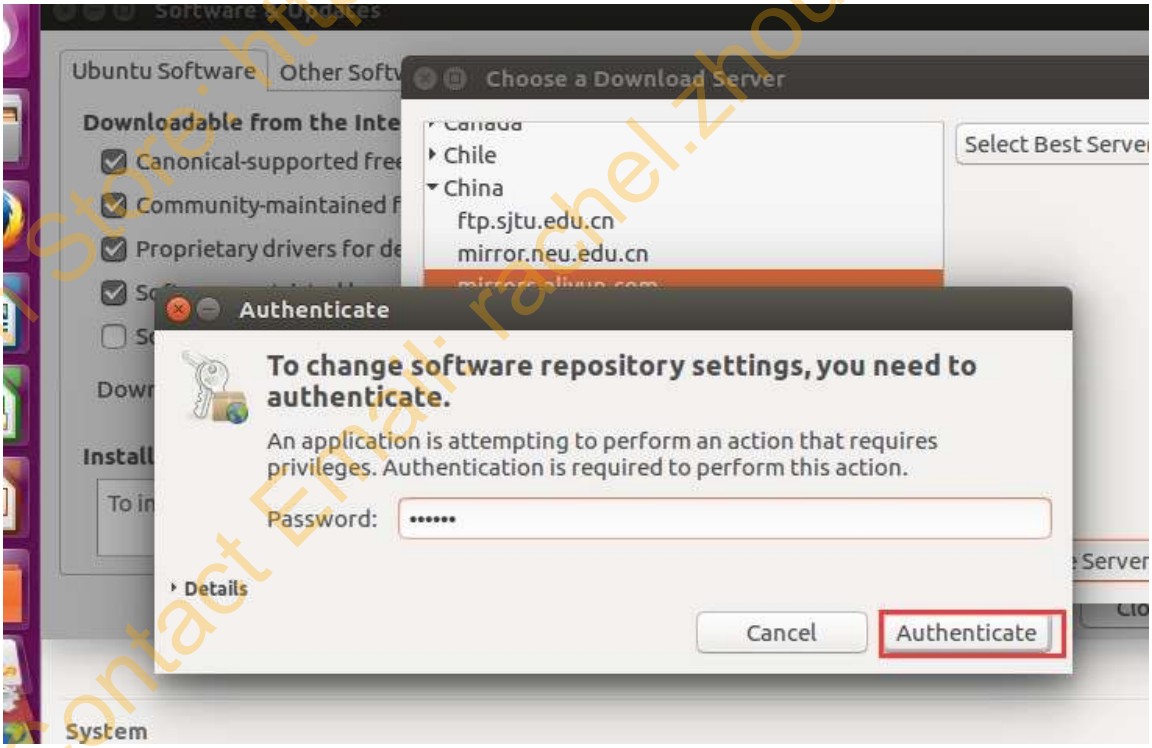
2) Select "Other..." in "Software&Updates"



3) Click on "SelectBestServer" to test out the fastest server and then select "Choose Server", which is based on the fact that the virtual machine can connect to the Internet.

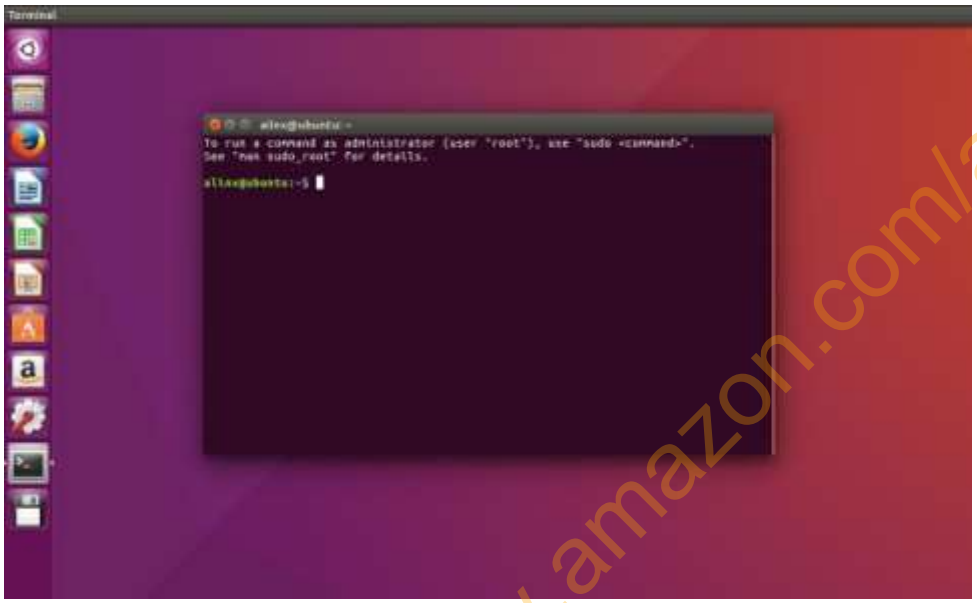


4) Enter the password to complete the software source modification.



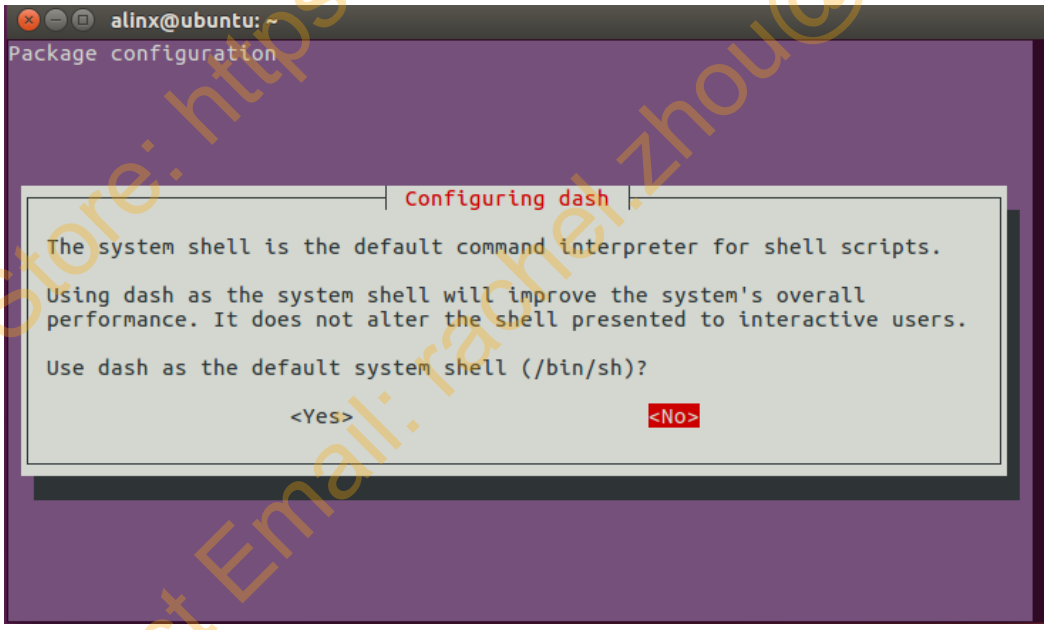
Part 14.2.3: Set bash to default sh

1) Open Ctrl+Alt+T terminal



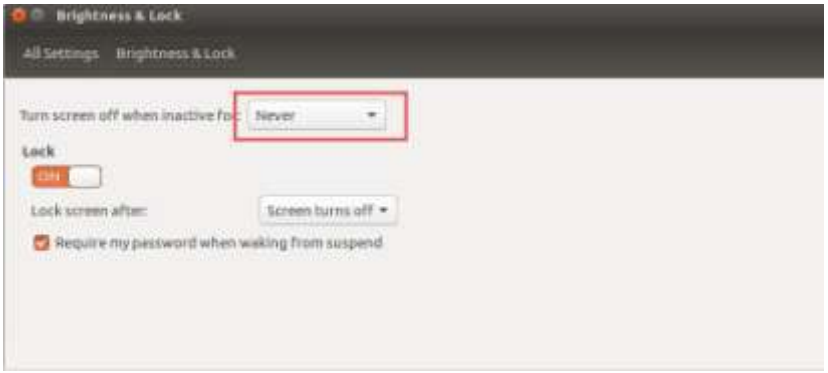
2) Enter the command, Configuringdash select "No", press Enter to confirm

```
sudo dpkg-reconfigure dash
```



Part 14.2.4: Set screen lock time

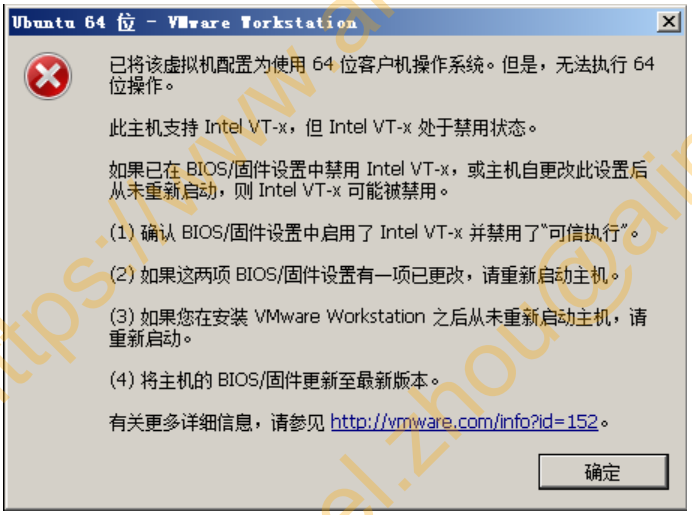
In order to copy large files to the Ubuntu system, we cancel the screen lock



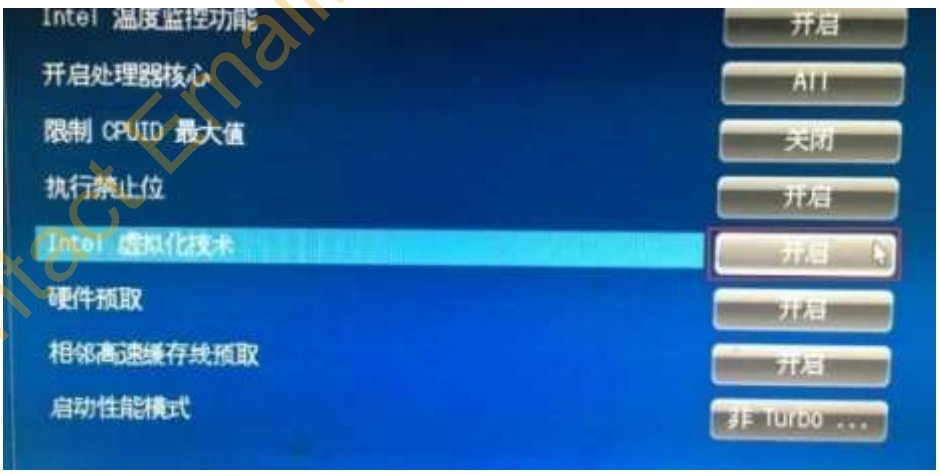
Part 14.3: Q&A

Part 14.3.1: Virtual machine requires virtualization support

1) If Ubuntu is installed, the following error message box will pop up, the user needs to restart the computer and enter the BIOS for settin



2) After restarting the computer, go to the BIOS, find the Intel virtualization technology and click Open. Different FPGA development boards, may have different names

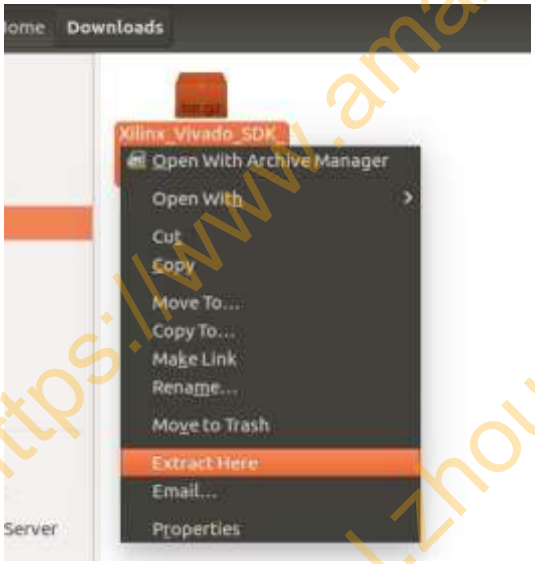


Part 15: Ubuntu installs the Vivado software for Linux

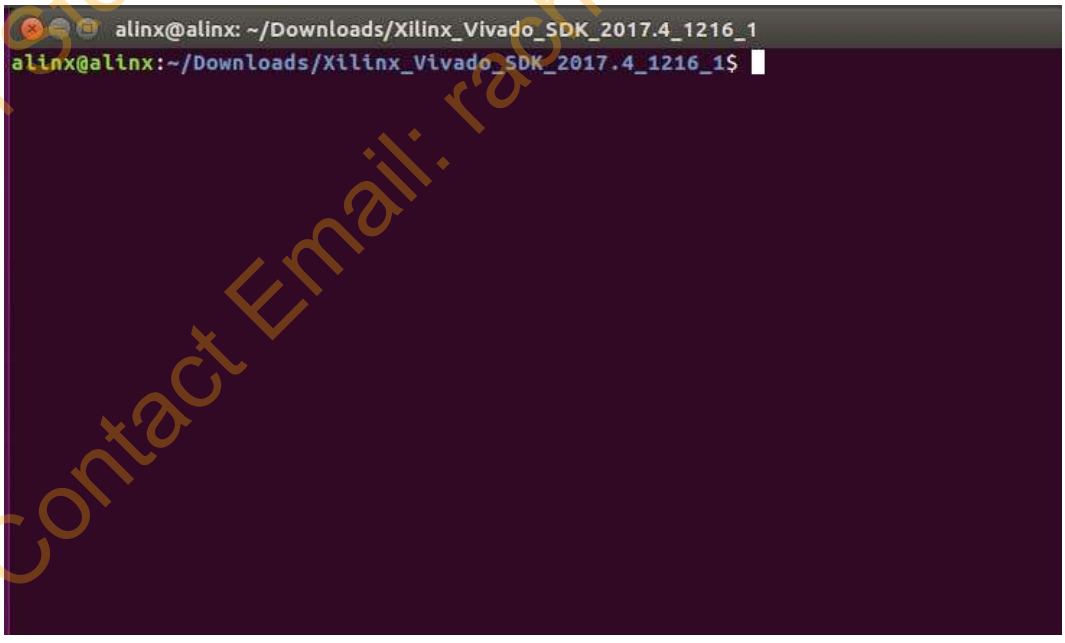
Although Vivado software under Windows can solve most problems, occasionally we will use the Linux version of Vivado, especially the SDK, we can cross-compile many applications.

Part 15.1: Install Linux version Vivado

1) Copy the installation file to the virtual machine Ubuntu, extract the file

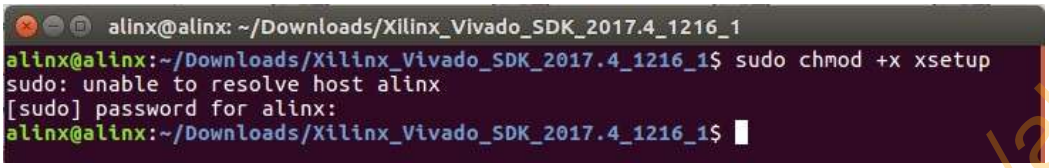


2) After using the terminal to enter the decompression



3) Run command

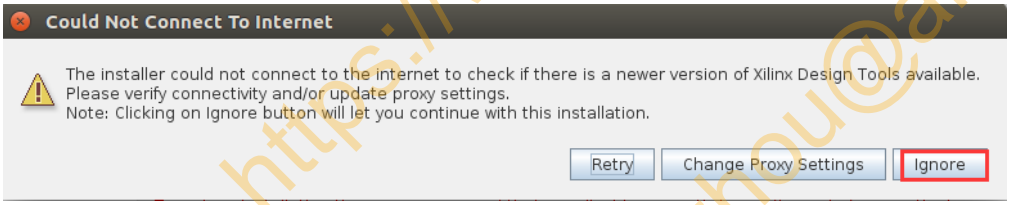
```
sudo chmod +x xsetup
```



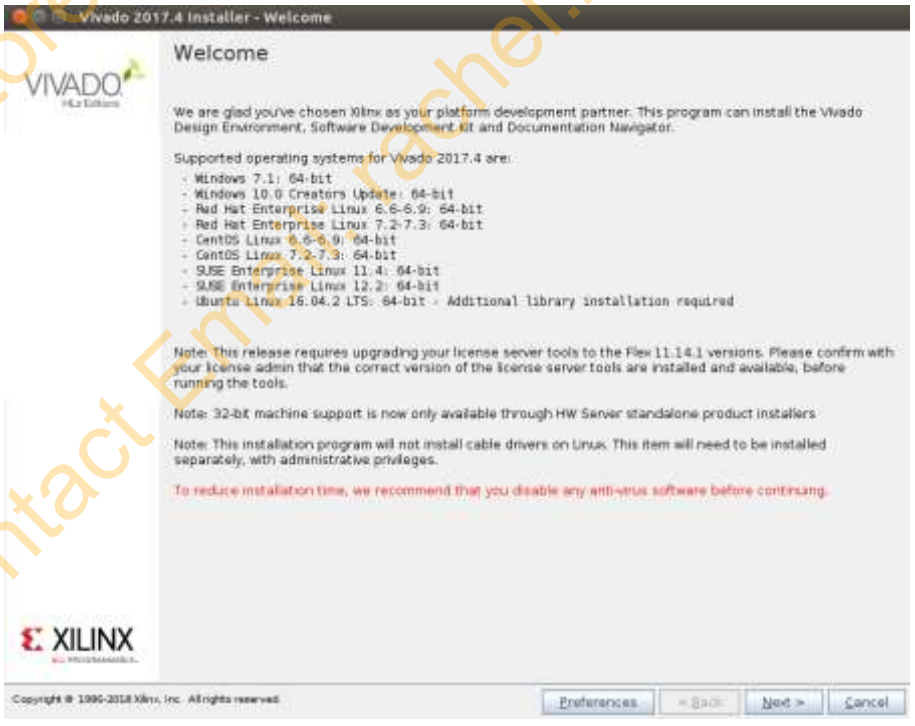
4) Run the command to start the installation, need to pay attention, here is the sudo installation

```
sudo ./xsetup
```

5) If these windows pop up, click on "Ignore"

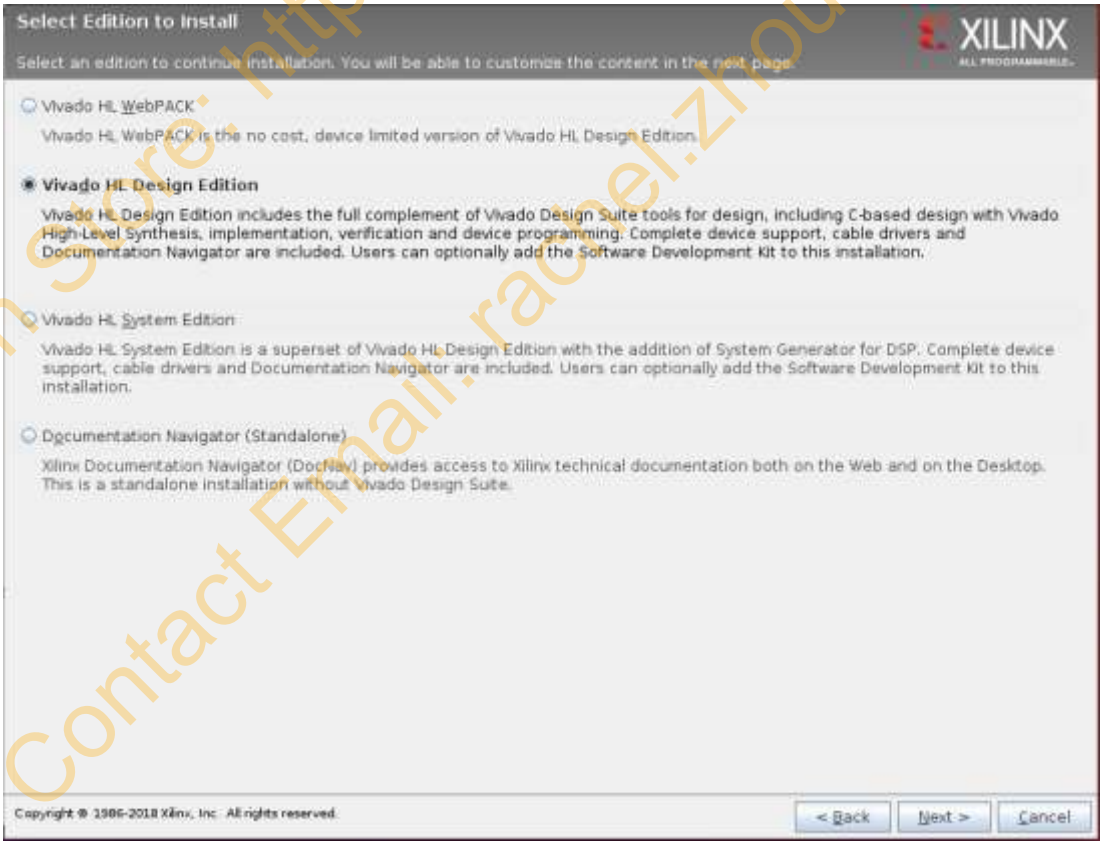


6) The installation process requires us to turn off anti-virus software

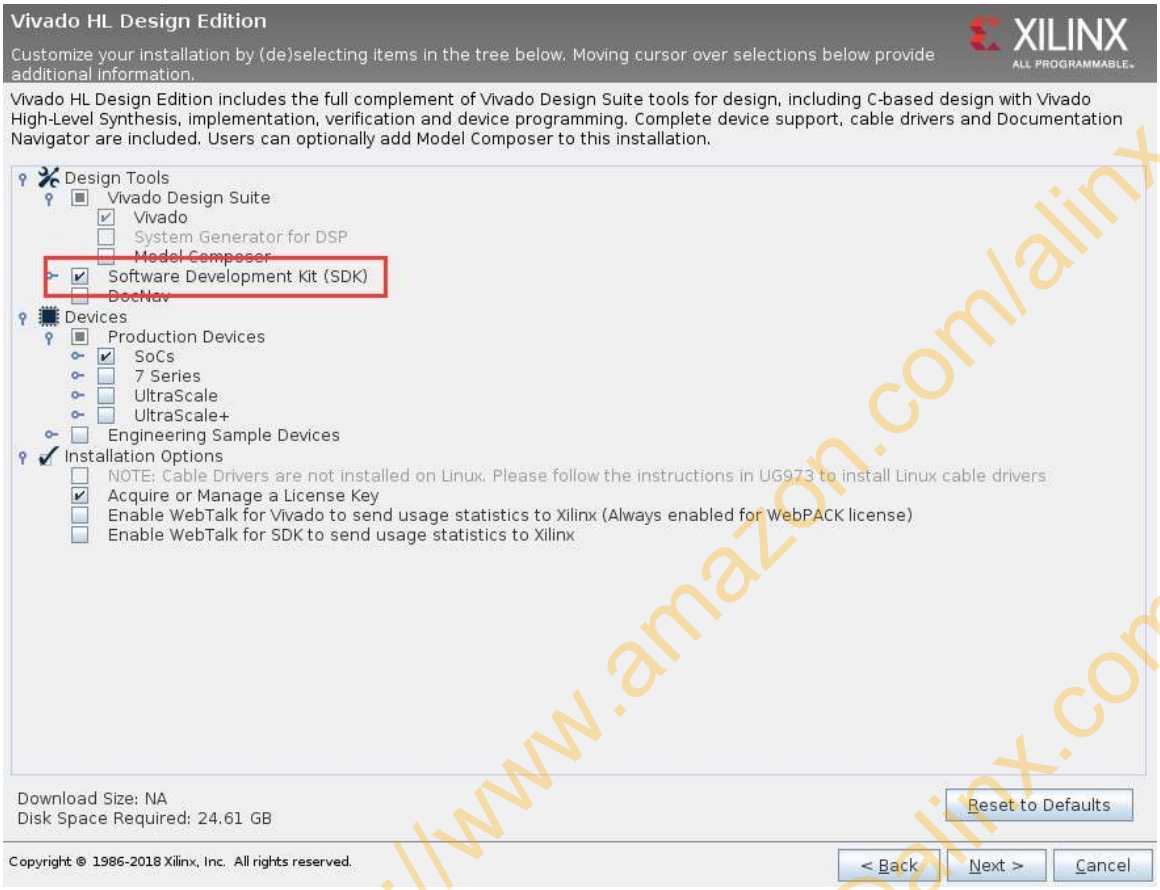


7) Agree to all terms

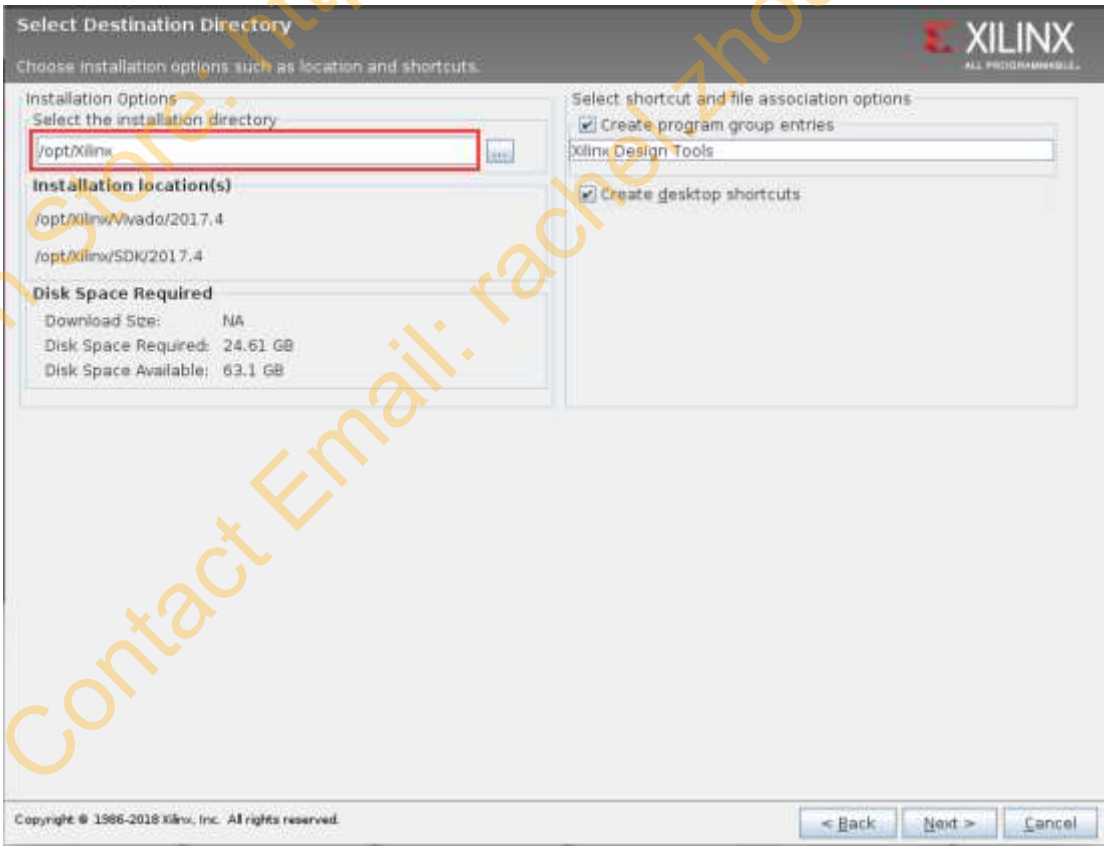
8) Select “VivadoHLDesignEdition”



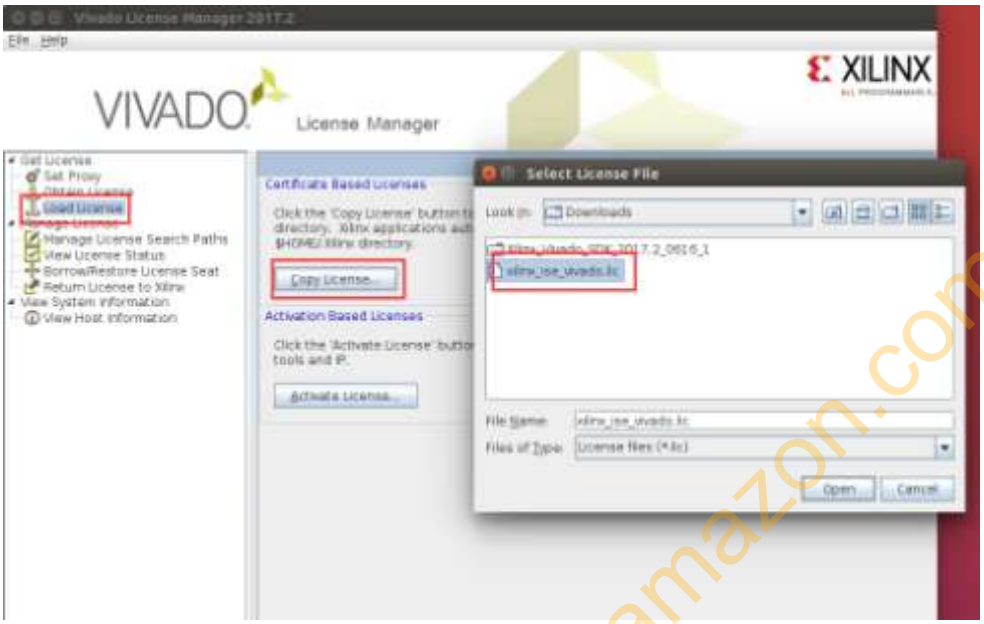
9) Be sure to choose "SoftwareDevelopmentKit(SDK)" here.



10) The installation path uses the default path



11)Click CopyLicense to install the "lic" file



Part 15.2: Permission settings

Run the command to add run permissions

```
sudo chmod 777 -R /opt/Xilinx/
sudo chmod 777 -R ~/.Xilinx/
```

Part 15.3: Install the downloader driver

Run the following command to install the downloader driver

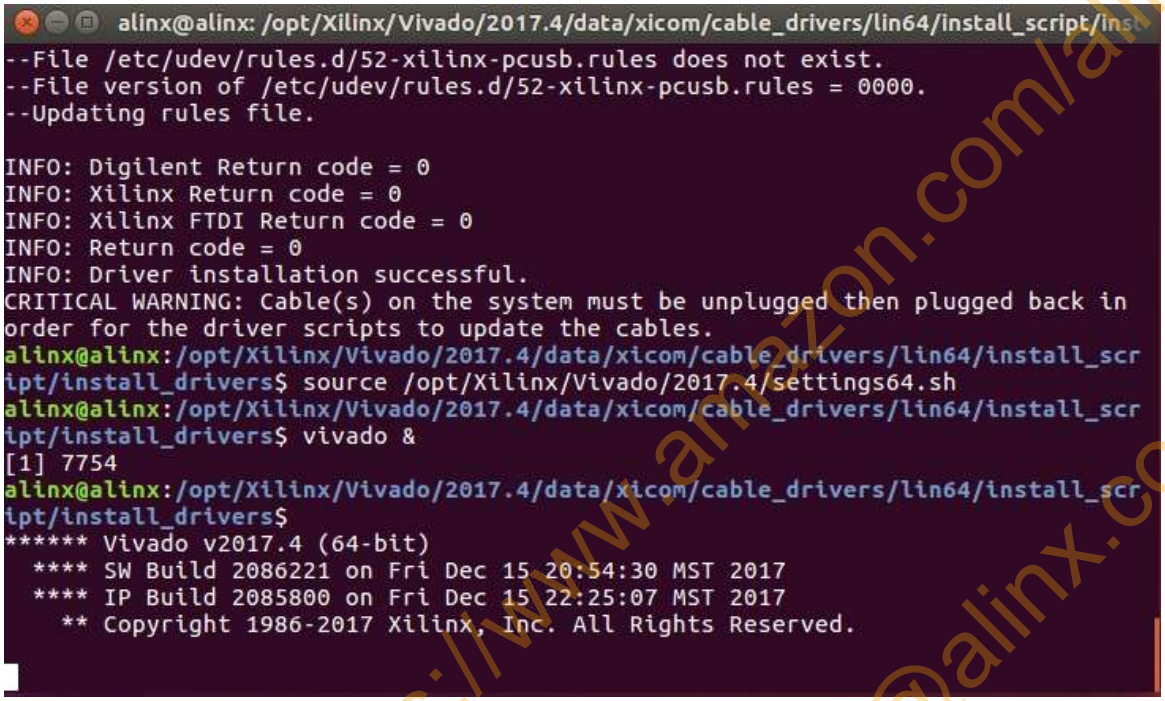
```
cd/opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/install_drivers/
sudo ./install_drivers
```

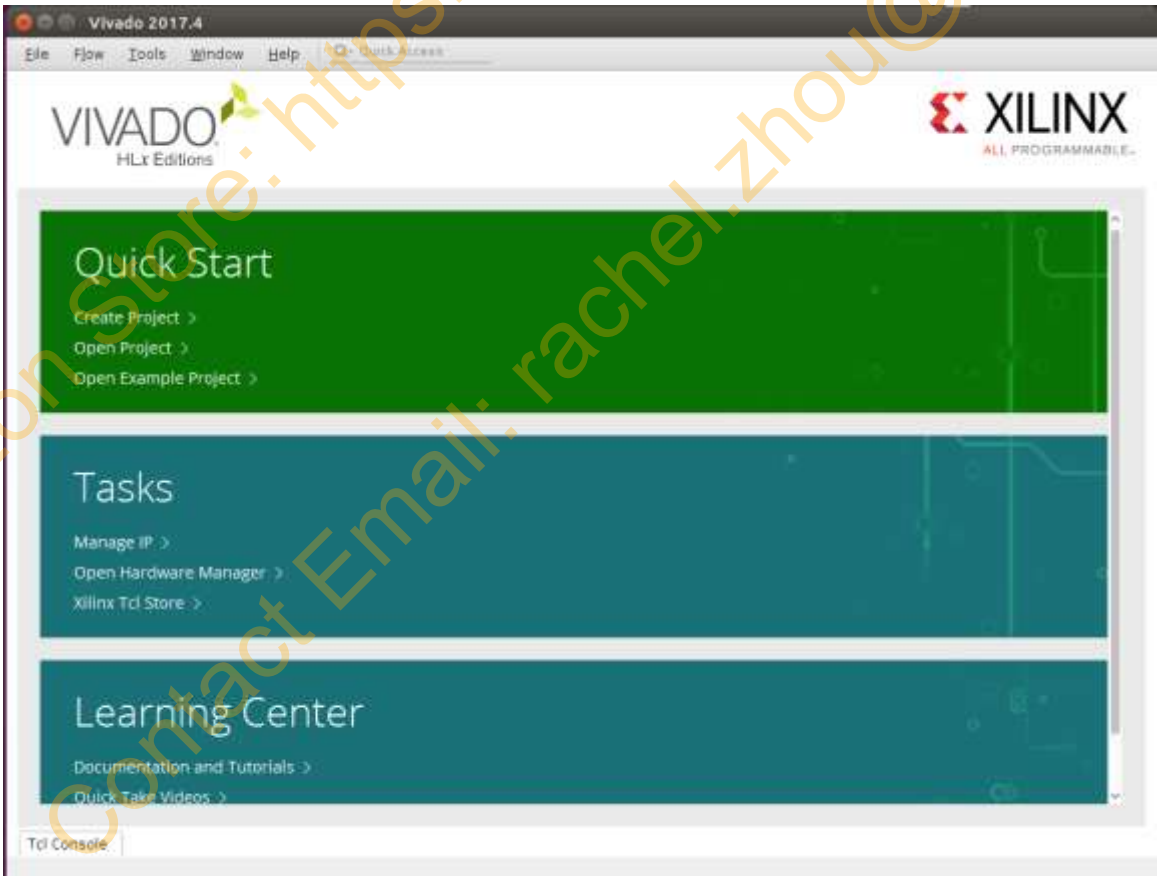
```
alinx@alinx: /opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_script/inst
INFO: Installing cable drivers.
INFO: Script name = ./install_drivers
INFO: HostName = alinx
INFO: Current working dir = /opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/l
in64/install_script/install_drivers
INFO: Kernel version = 4.10.0-28-generic.
INFO: Arch = x86_64.
Successfully installed Digilent Cable Drivers
--File /etc/udev/rules.d/52-xilinx-ftdi-usb.rules does not exist.
--File version of /etc/udev/rules.d/52-xilinx-ftdi-usb.rules = 0000.
--Updating rules file.
--File /etc/udev/rules.d/52-xilinx-pcusb.rules does not exist.
--File version of /etc/udev/rules.d/52-xilinx-pcusb.rules = 0000.
--Updating rules file.

INFO: Digilent Return code = 0
INFO: Xilinx Return code = 0
INFO: Xilinx FTDI Return code = 0
INFO: Return code = 0
INFO: Driver installation successful.
CRITICAL WARNING: Cable(s) on the system must be unplugged then plugged back in
order for the driver scripts to update the cables.
alinx@alinx:/opt/Xilinx/Vivado/2017.4/data/xicom/cable_drivers/lin64/install_scr
ipt/install_drivers$
```

Part 15.4: Testing Vivado

1) Start Vivado by running the following command

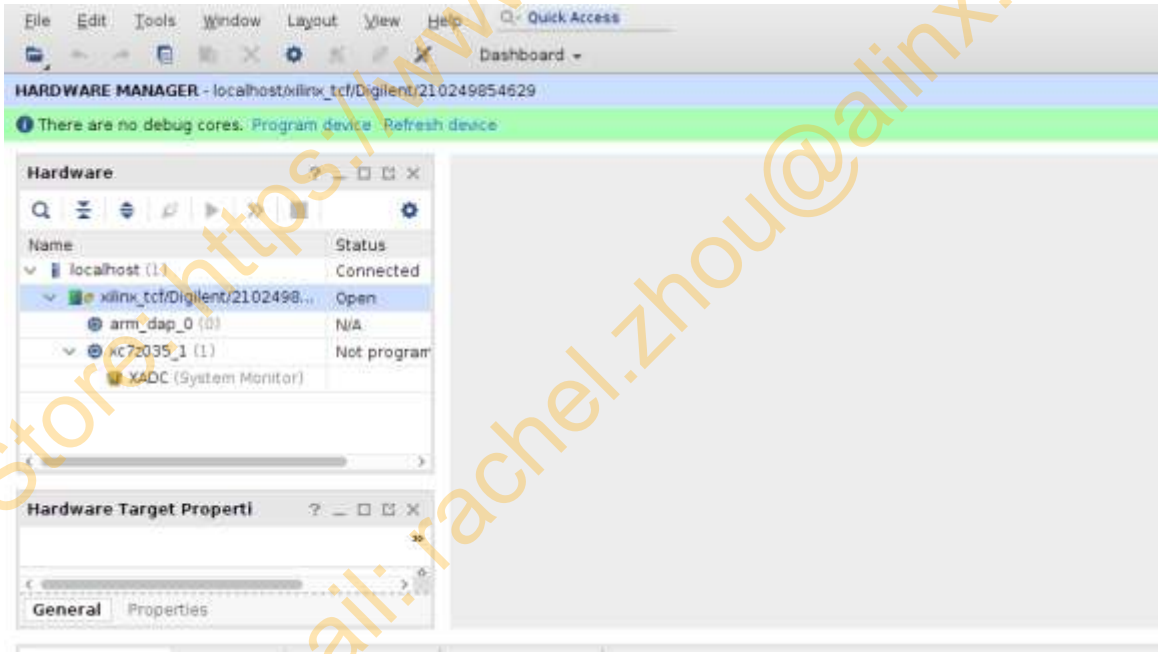
```
source    /opt/Xilinx/Vivado/2017.4/settings64.sh
vivado &
```





2) Connect the downloader to the virtual machine

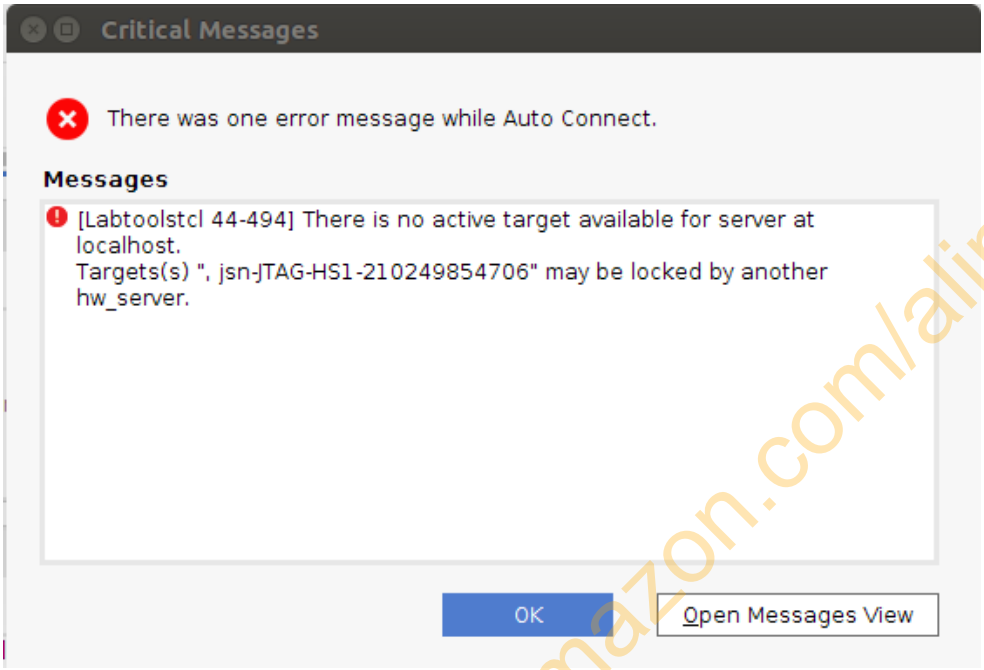
3) Connect the development board and downloader, use the "OpenHardware Manager" test, the chip can be found under normal circumstances, indicating that the Vivado and downloader drivers are successfully installed.



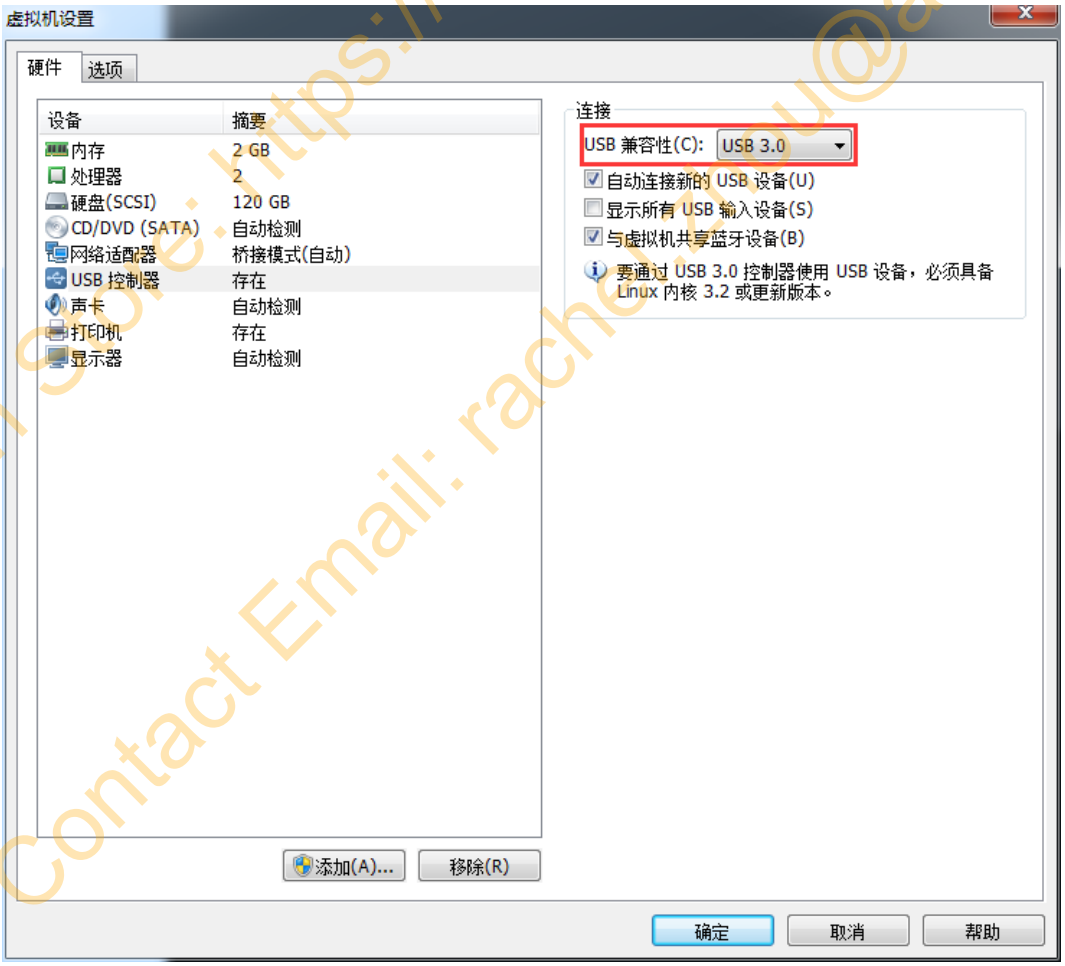
Part 15.5: Q&A

Part 15.5.1: Prompt is occupied when downloading Linux downloader

1) The downloader can be found when testing the hardware, but there is an error



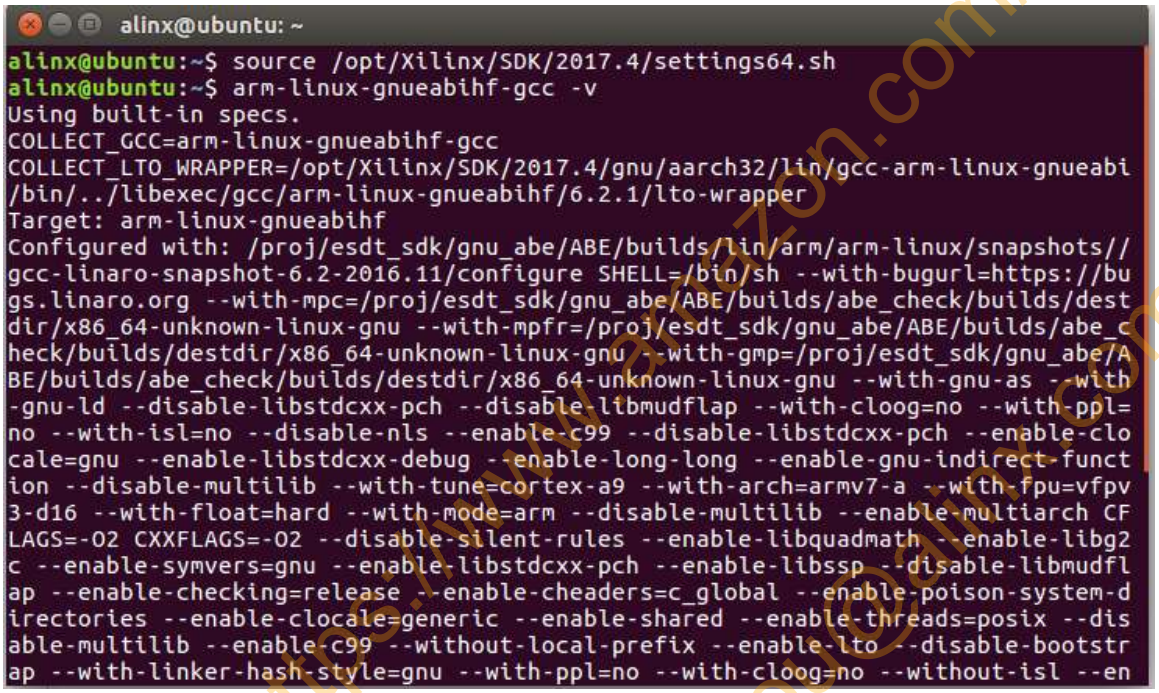
2) Some virtual machineneed to set USB compatibility, turn off the Ubuntu of the virtual machine, set the USB compatibility to "USB3.0", try again, if you can't use the downloader, you can only download it using the Windows version.



Part 15.5.2: Cross compiler for ZYNQ

When the installation Vivado have chosen to install the SDK, SDK contains the inside cross compiler “arm-linuxgnueabihf-gcc”.

```
source/opt/Xilinx/SDK/2017.4/settings64.sh
arm-linux-gnueabihf-gcc -v
```



Part 16: Petalinux tool installation

Part 16.1: Introduction to Petalinux

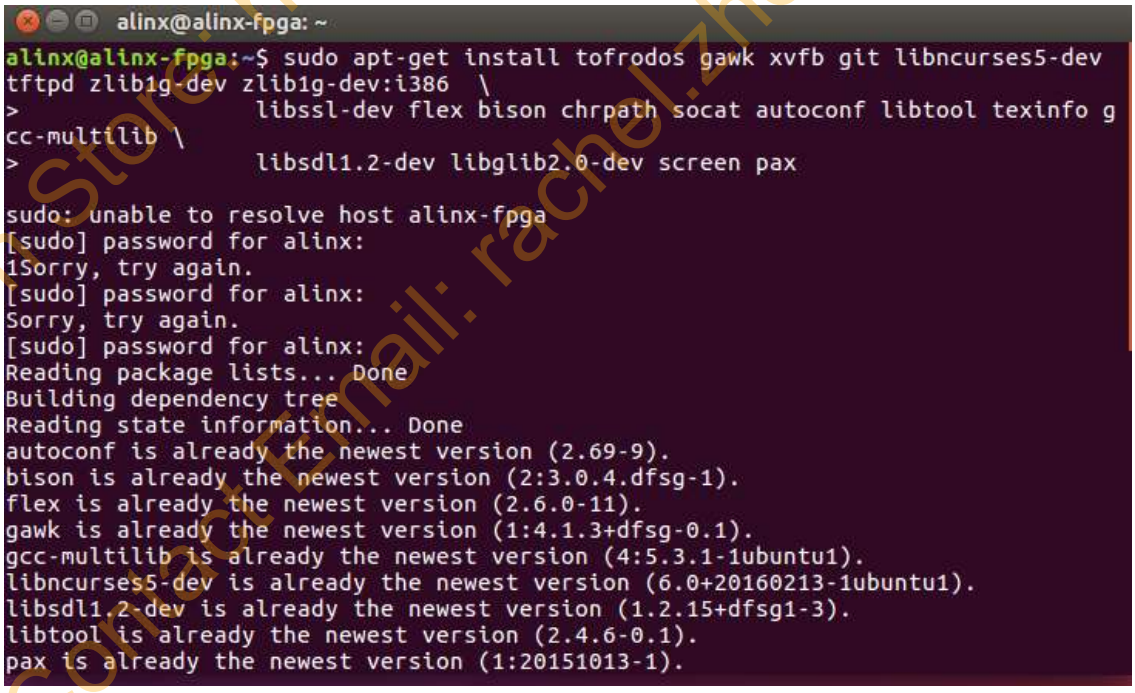
Petalinux is not a special Linux kernel, but a set of development environment configuration tools to reduce the workload of uboot, kernel, and root file system configuration. You can automatically configure related software from Vivado's exported hardware information.

Here to be particularly emphasized, petalinux has strict requirements on the system version and settings. If you use other versions of the operating system, please solve the problem yourself.

Part 16.2: Install the necessary libraries

1) Run the following command to install the library. Make sure the virtual machine cannot connect to the Internet before installation.

```
sudo apt-get install tofrodos gawk xvfb git libncurses5-dev tftpd zlib1g-dev zlib1g-dev:i386
\
                    libssl-devflexbisonchrpathsocatautoconflibtooltexinfogcc-multilib\
                    libsdl1.2-devlibglib2.0-devscreenpax
```



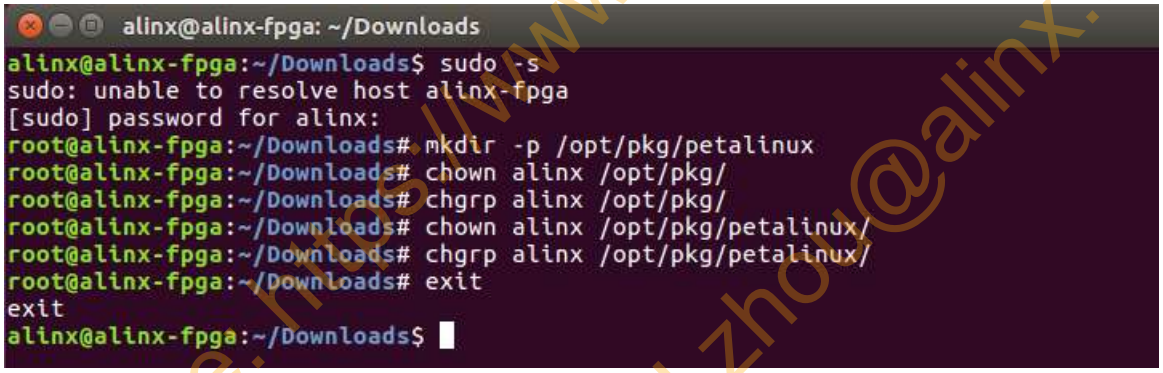
2) Configure tftpserver. If you do not need to boot from TFTP, this step is optional.

```
sudo-s
apt-get    install    tftpd-hpa
chmod   a+w/var/lib/tftpboot/
reboot
```

Part 16.3: Install Petalinux

1) Run the command to prepare for installation. <your_user_name> is your username, such as alinx in the figure.

```
sudo -s
mkdir -p /opt/pkg/petalinux
chown<your_user_name>/opt/pkg/
chgrp<your_user_name>/opt/pkg/
chgrp<your_user_name>/opt/pkg/petalinux/
chown<your_user_name>/opt/pkg/petalinux/
exit
```

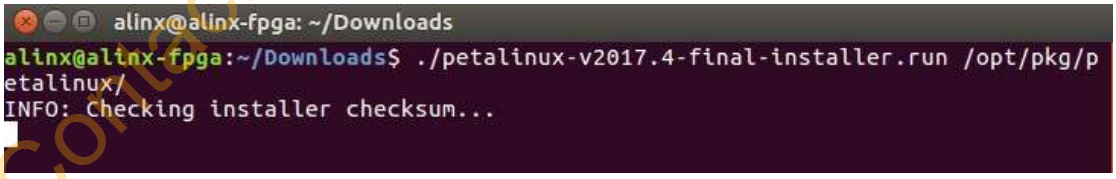


2) Add run permissions to the installation file, of course, copy the petalinux-v2017.4-final-installer.run file to Linux first.The working directory of the host, in the routine is ~/Downloads

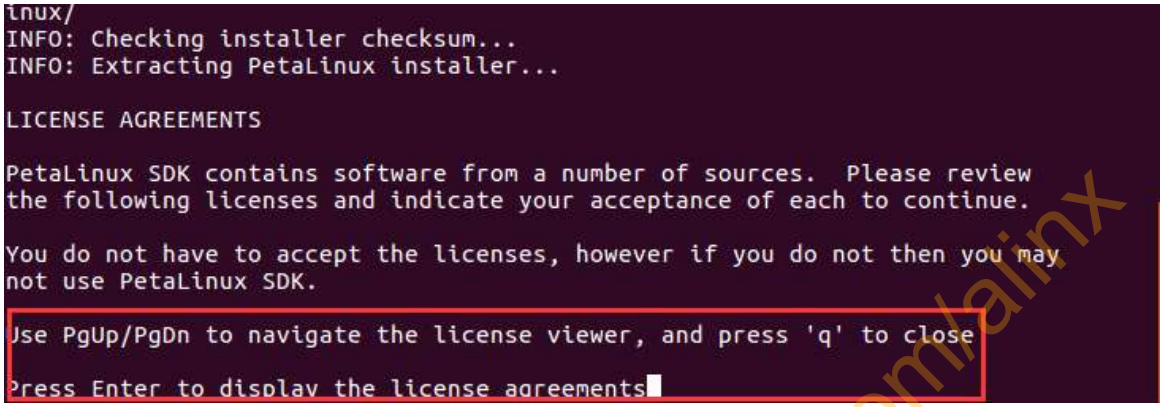
```
sudo chmod +x petalinux-v2017.4-final-installer.run
```

3) start installation

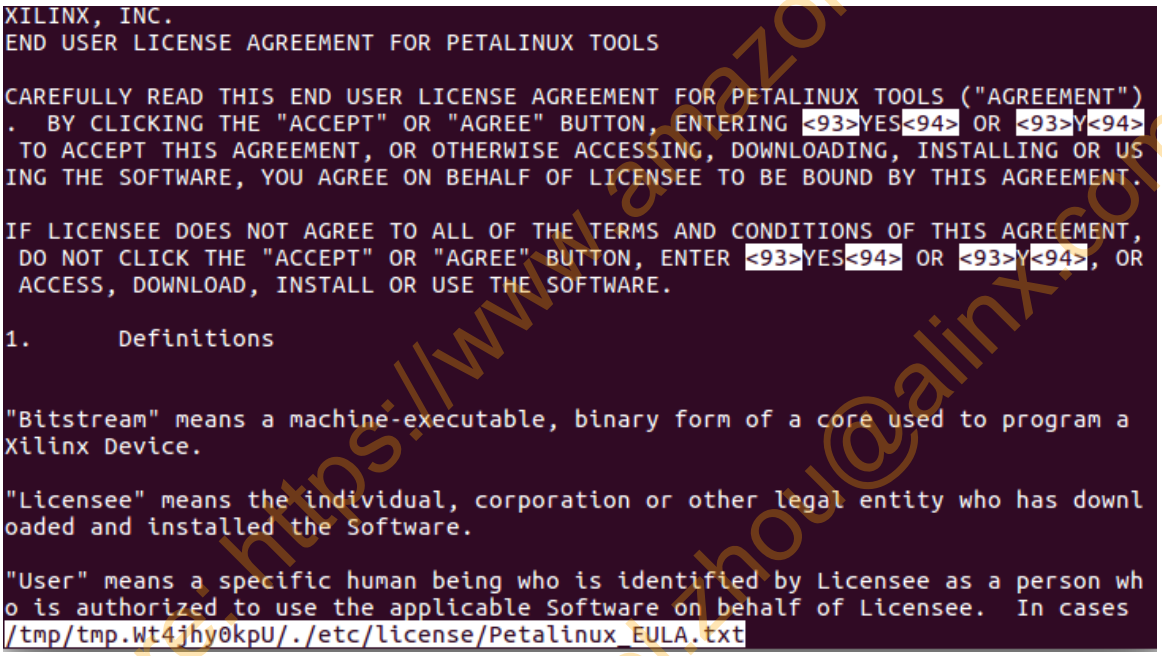
```
./petalinux-v2017.4-final-installer.run /opt/pkg/petalinux/
```



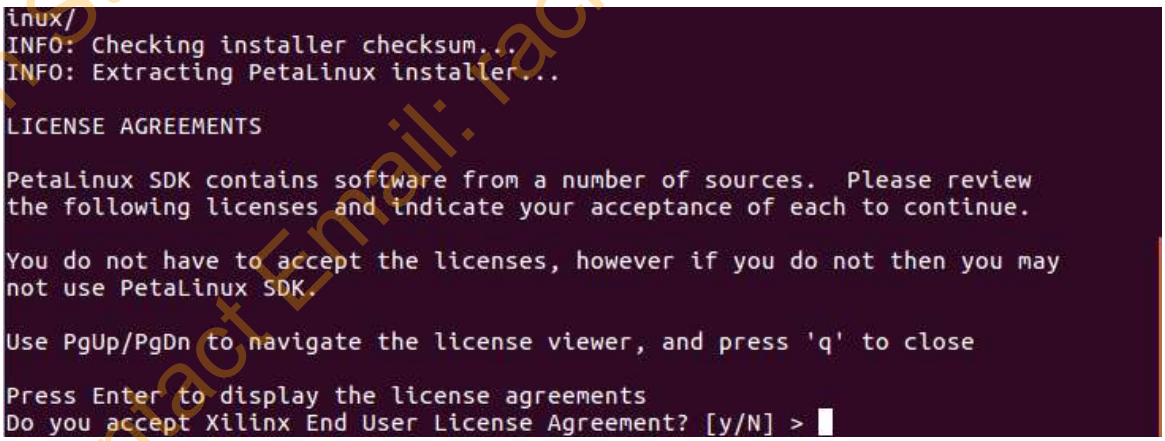
4) Press Enter to view the agreement content



5) Press “q” to quit the protocol content



6) Press y to agree to the agreement



7) During the installation process, the license will pop up, press “q” to exit, then press “y” to agree.

```
WebTalk Terms and Conditions

By indicating I accept this WebTalk notice, I also confirm that I have read Sect
ion 13 of the terms and conditions above concerning WebTalk and have been afford
ed the opportunity to read the WebTalk FAQ posted at http://www.xilinx.com/webta
lk.  I understand that I am able to disable WebTalk later if certain criteria de
scribed in Section 13(c) apply.  If they don't apply, I can disable WebTalk by u
ninstalling the Software or using the Software on a machine not connected to the
 internet.  If I fail to satisfy the applicable criteria or if I fail to take th
e applicable steps to prevent such transmission of information, I agree to allow
 Xilinx to collect the information described in Section 13(a) for the purposes d
escribed in Section 13(b).

/tmp/tmp.Wt4jhy0kpU/./etc/license/WebTalk_notice.txt (END)
```

```
INFO: Checking installer checksum...
INFO: Extracting PetaLinux installer...

LICENSE AGREEMENTS

PetaLinux SDK contains software from a number of sources.  Please review
the following licenses and indicate your acceptance of each to continue.

You do not have to accept the licenses, however if you do not then you may
not use PetaLinux SDK.

Use PgUp/PgDn to navigate the license viewer, and press 'q' to close

Press Enter to display the license agreementsq
Do you accept Xilinx End User License Agreement? [y/N] > y
Do you accept Webtalk Terms and Conditions? [y/N] > y
Do you accept Third Party End User License Agreement? [y/N] > y
INFO: Checking installation environment requirements...
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
INFO: Installing PetaLinux...
```

Part 17: NFS service software installation

NFS (Network File System) is a technology developed by Sun Corporation and introduced in 1984 to share files between different machines and different operating systems. NFS was originally designed to be used between different systems, so its communication protocol design is independent of the host and operating system.

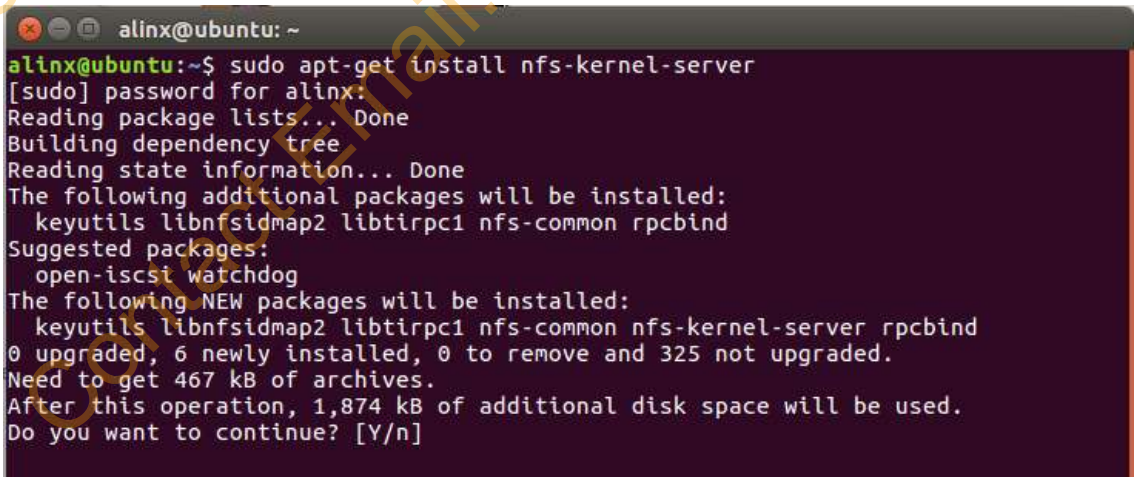
When using remote files, you can use the mount command to mount the file system on the remote NFS server under the local file system. Operating remote files is no different from operating local files. The shared file or directory of the NFS server is recorded in the /etc/exports file

In embedded Linux development, NFS is often used. The target system is usually used as an NFS client and the Linux host is used as an NFS server.On the target system, mount the NFS shared directory of the server to the local device through NFS, and directly run the files on the server.NFS is necessary to debug system driver modules and applications, and Linux also supports NFS root file system, which can boot the system directly from remote NFS root. This is also necessary for the cutting and integration of embedded Linux root file system.

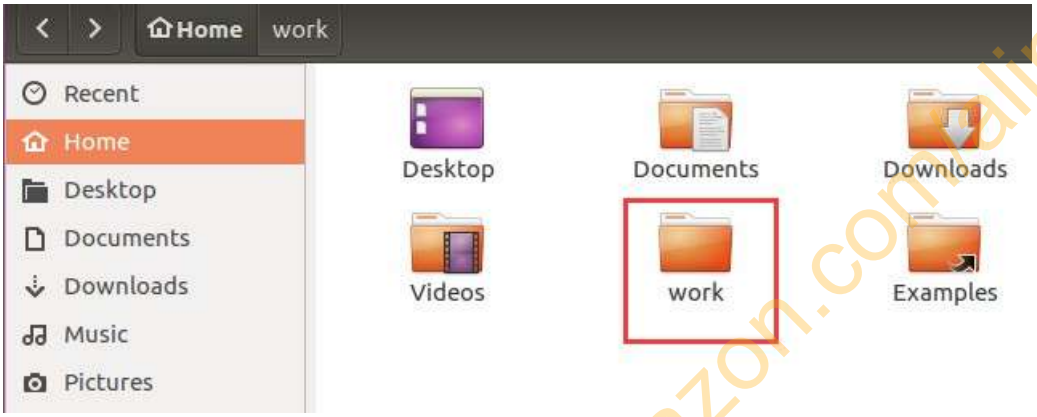
Part 17.1: Install the NFS service

1) Install the NFS server with the following command:

```
sudo apt-get install nfs-kernel-server
```



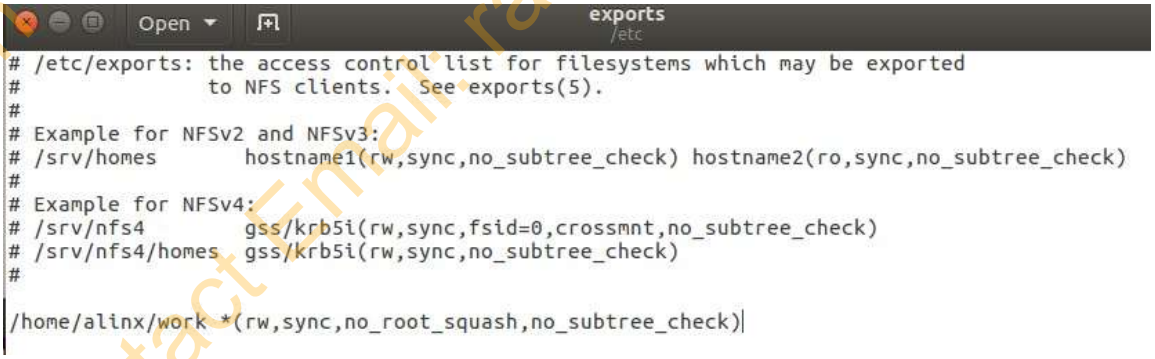
2) Create a new work directory as a working directory for NFS. In the future, we can put the cross-compiled program in this directory. The FPGA development board can easily share the files in this directory.



3) Use the following command to edit the /etc/exports file to configure the NFS service path.

```
sudo    gedit/etc/exports
```

4) Add /home/alinx/work *(rw,sync,no_root_squash,no_subtree_check) at the end to configure the /home/alinx/work directory to be a working directory for NFS.



5) Execute the following command to restart the rpcbind service. nfs is an RPC program. Before using it, you need to map the port and set it by rpcbind.

```
sudo /etc/init.d/rpcbind restart
```

6) Run the following command to restart the nfs service.

```
sudo /etc/init.d/nfs-kernel-server restart
```

Part 17.2: Testing NFS

1) Mount NFS by the following command, and mount the NFS working path in the /mnt directory locally.

```
mount -t nfs 127.0.0.1:/home/alinx/work /mnt
```

2) Enter /mnt, create a new test directory test, you can see the test folder in the /home/alinx/work directory.

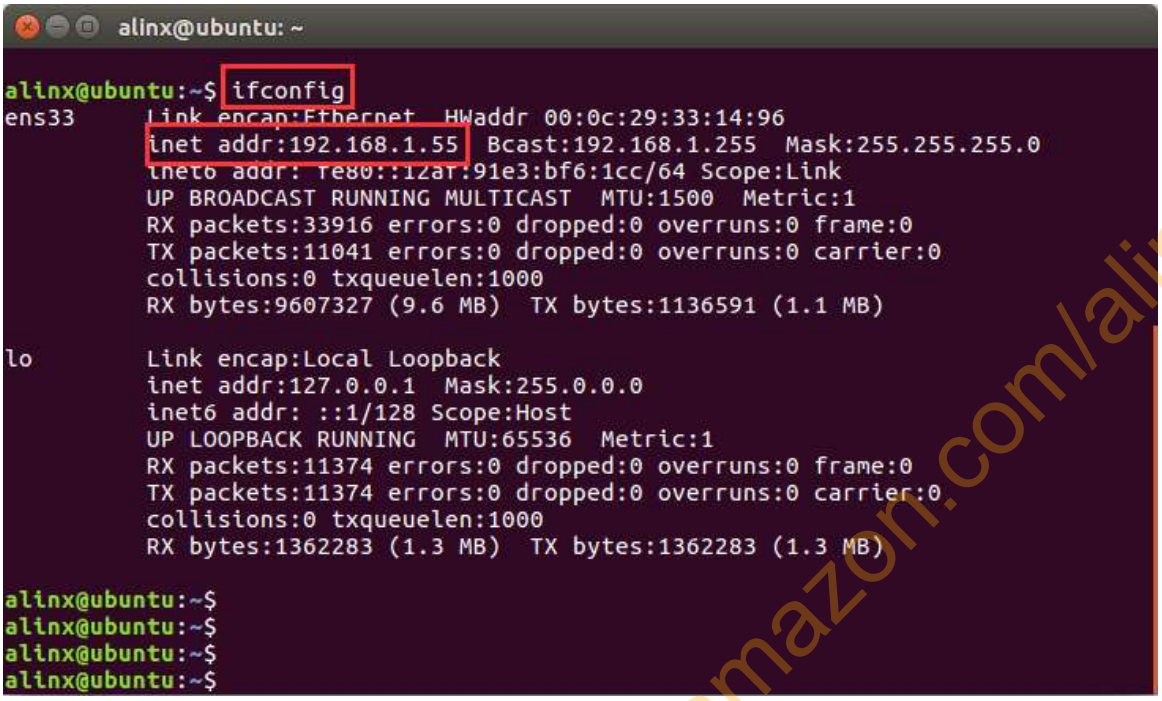
```
cd/mnt
mkdir test
```

Part 17.3: Q&A

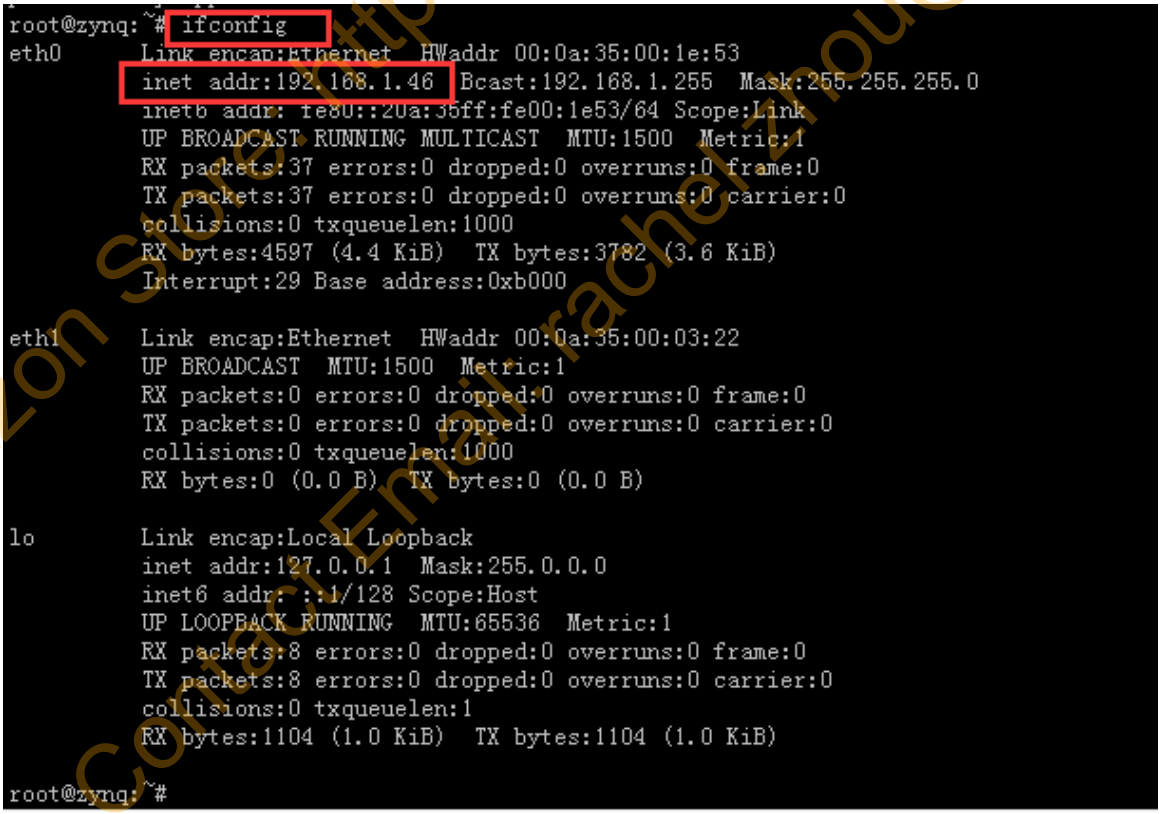
Part 17.3.1: NFS cannot be mounted

First confirm whether the virtual machine and FPGA development board are on the same network segment.

Use the “ifconfig” command to view the virtual machine IP address. The example in the following figure is **192.168.1.55**, which belongs to the **192.168.1** network segment. Because there is a “DHCP” server in the development environment, the virtual machine's IP address is automatically assigned because the network environment is different. This tutorial does not explain how to configure the network. If you do not configure the network, consult your network administrator.



Use the “ifconfig” command in the serial terminal to view the IP address of the development board. The example in the following figure is 192.168.1.46, which belongs to the 192.168.1 network segment. If there is no IP, or different network segment from the FPGA development board IP, please contact the network administrator



"Ping" the virtual machine in the serial terminal. The routine pings **192.168.1.55**. This is because the virtual machine IP is 192.168.1.55. It can be pinged to mount **NFS** normally.

```
root@zynq:~# ping 192.168.1.55
PING 192.168.1.55 (192.168.1.55) 56(84) bytes of data.
64 bytes from 192.168.1.55: icmp_seq=1 ttl=64 time=0.862 ms
64 bytes from 192.168.1.55: icmp_seq=2 ttl=64 time=0.489 ms
64 bytes from 192.168.1.55: icmp_seq=3 ttl=64 time=0.779 ms
64 bytes from 192.168.1.55: icmp_seq=4 ttl=64 time=0.604 ms
64 bytes from 192.168.1.55: icmp_seq=5 ttl=64 time=0.574 ms
^C
--- 192.168.1.55 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4131ms
rtt min/avg/max/mdev = 0.489/0.641/0.862/0.153 ms
root@zynq:~#
```

Part 18: Customizing Linux with Petalinux

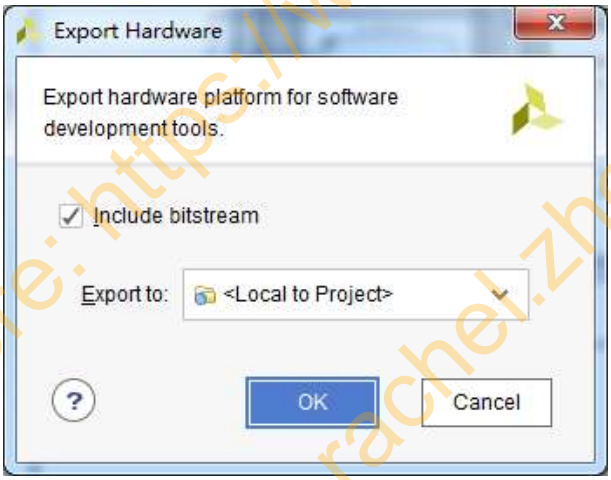
Experiment Vivado project is " linux_base "

In the previous tutorial we built the Petalinux environment, this tutorial mainly shows how to use Petalinux. It should be noted that this experiment can only be completed if the Linux host can connect to the Internet.

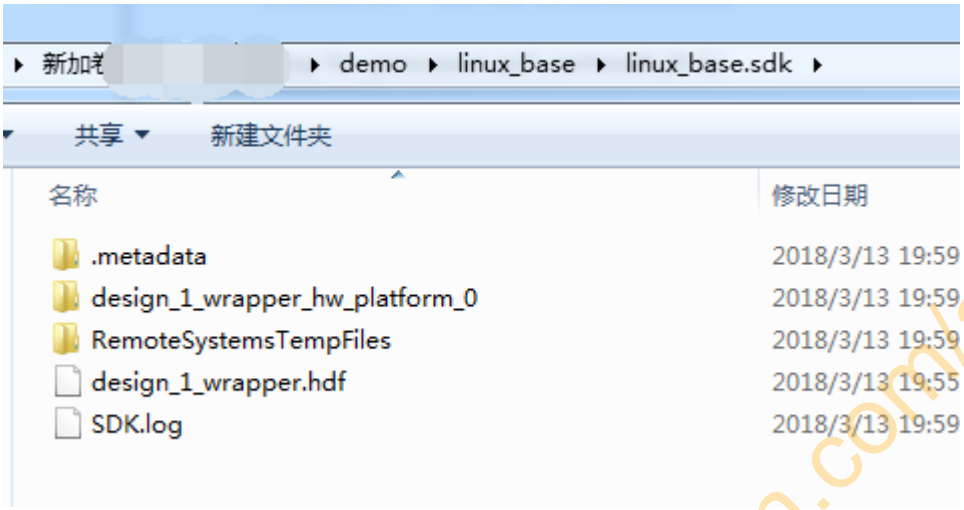
Part 18.1: Vivado Project

Using Petalinux, you can easily customize the embedded Linux system. You only need Vivado software to export the hardware information. Then Petalinux configures uboot, kernel, file system, etc. based on this information. The steps in the Vivado project were extensively explained in the previous experiments and will not be explained here.

1) Compile and generate bit file
2) Export hardware information

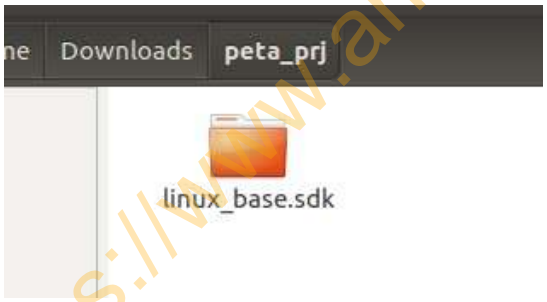


3) There will be a *.sdk directory in the vivado project directory, and a "*.hdf" file below, which contains the files used by petalinux.



Part 18.2: Create a project with “Petalinux”

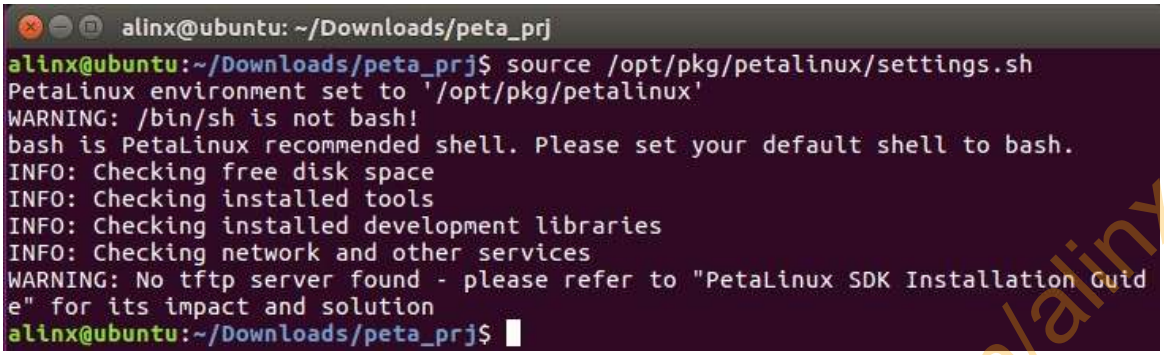
1) Copy the "linux_base.sdk" directory to the Linux host



2) Open the terminal and enter the working directory

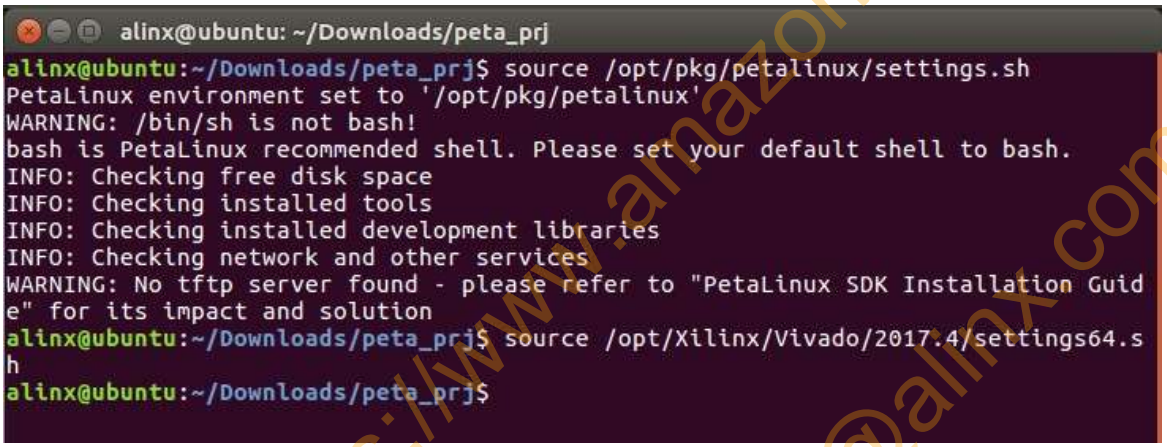
3) Set the petalinux environment variable, run the following command

```
source /opt/pkg/petalinux/settings.sh
```



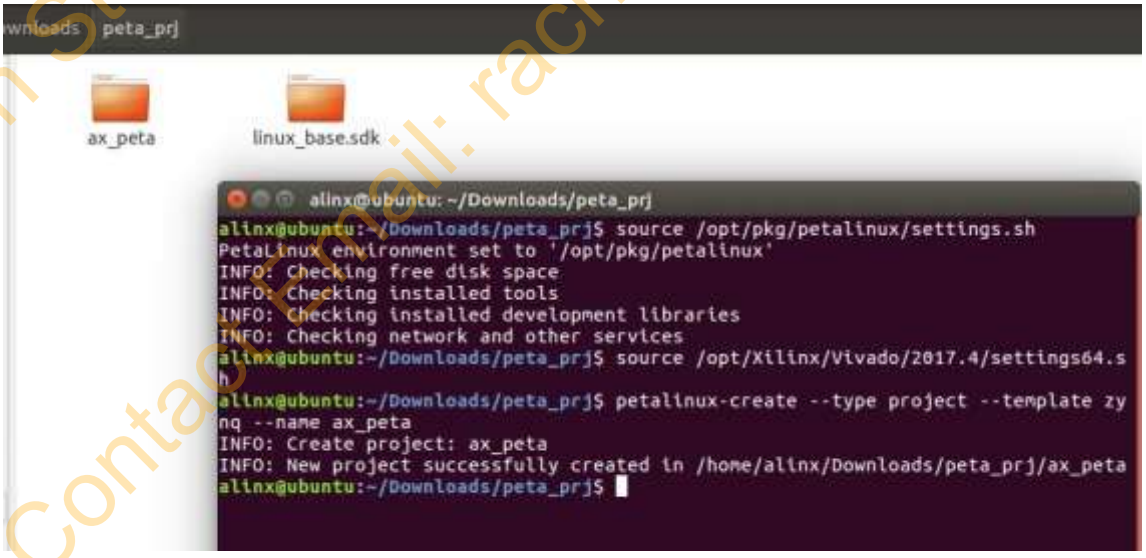
4) Run the following command to set the vivado environment variable

```
source /opt/Xilinx/Vivado/2017.4/settings64.sh
```



5) Use the following command to create a petalinux project named ax_peta. At this time, petalinux will automatically create a project named ax_peta. Pay attention to the double horizontal line

```
petalinux-create --type project --template zynq --name ax_peta
```

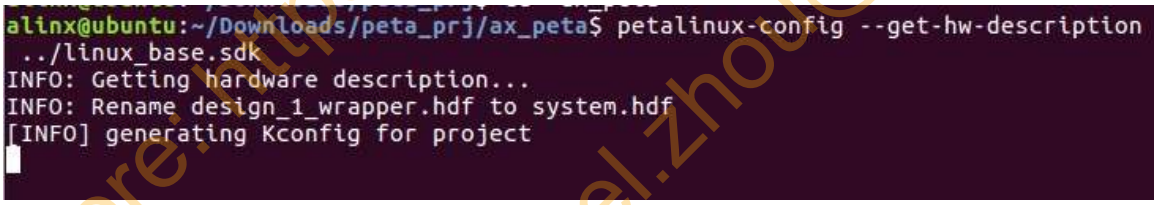


6) Use the following command to enter the petalinux working directory

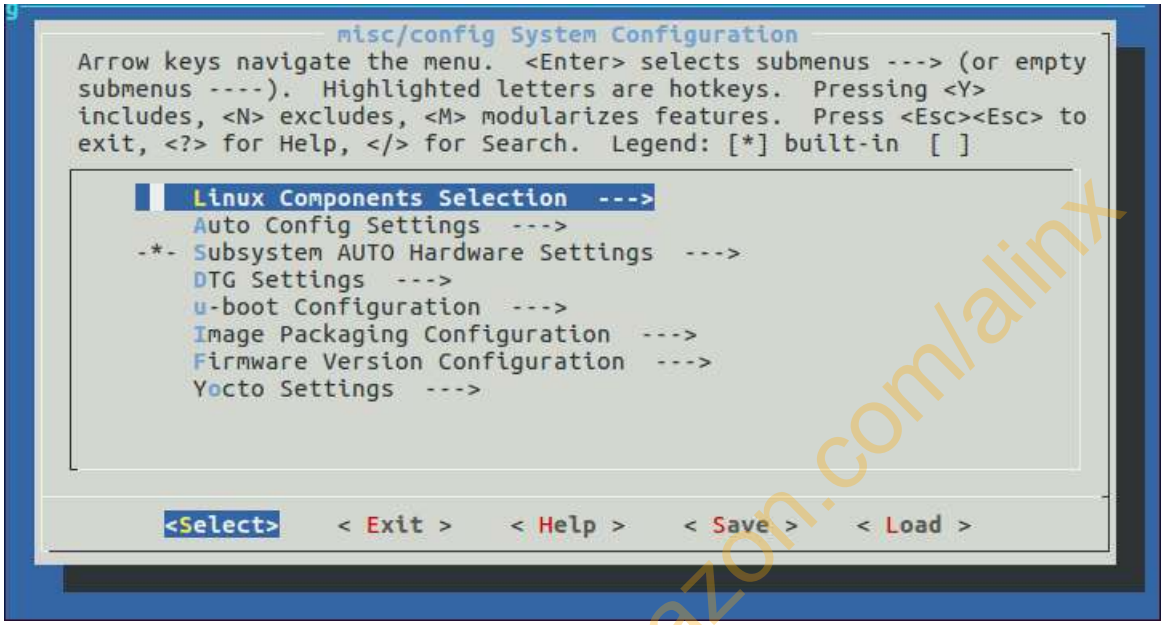
```
cd ax_peta
```

7) Use the following command to configure the hardware information of the Petalinux project. The "../linux_base.sdk" directory is the hardware information exported by vivado.

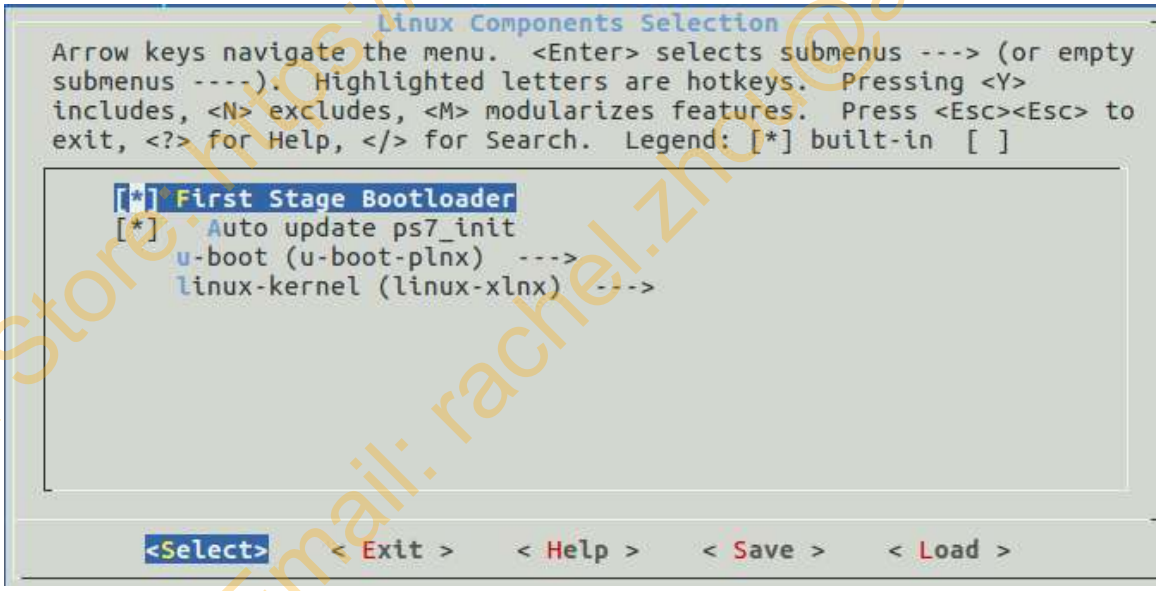
```
petalinux-config --get-hw-description ../linux_base.sdk
```



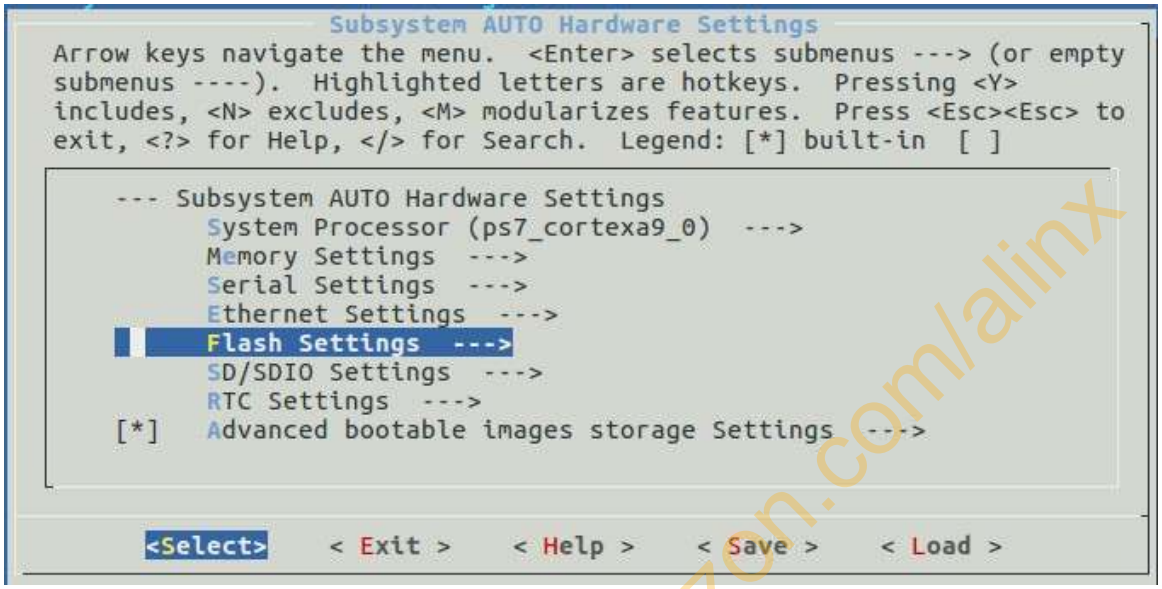
8) In the pop-up window, you can configure the petalinux project. If you want to configure it again after configuration, you can run the command "petalinux-config" to configure it.



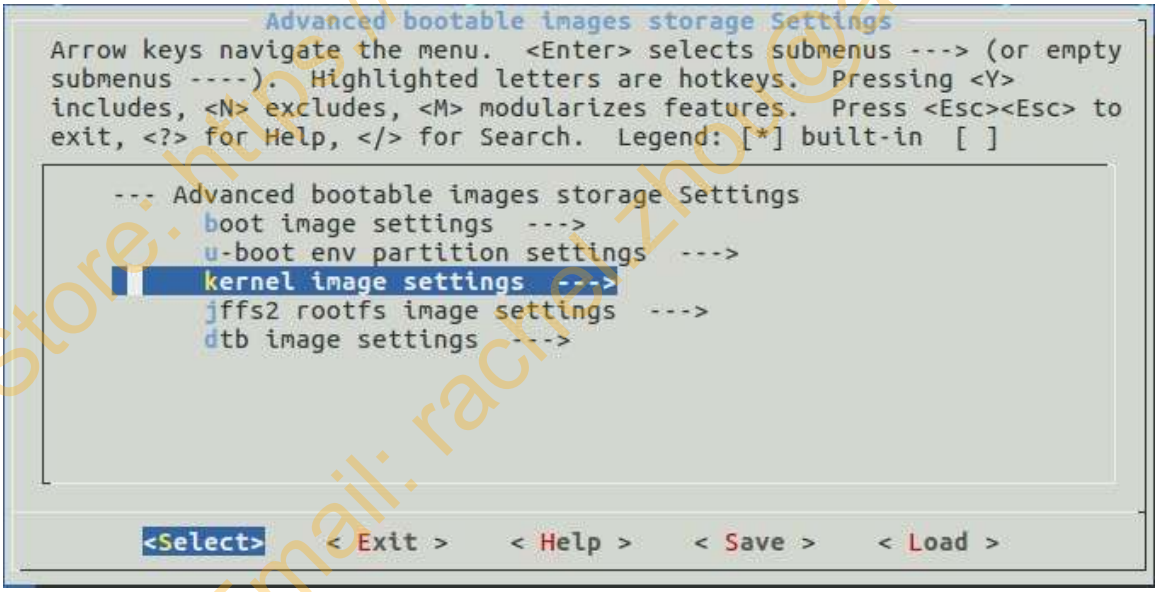
9) The source of uboot and Linux kernel can be configured in the option Linux Components Selection. The default is downloaded on github, which requires the Linux host to connect to the Internet to download. This experiment maintains the default configuration



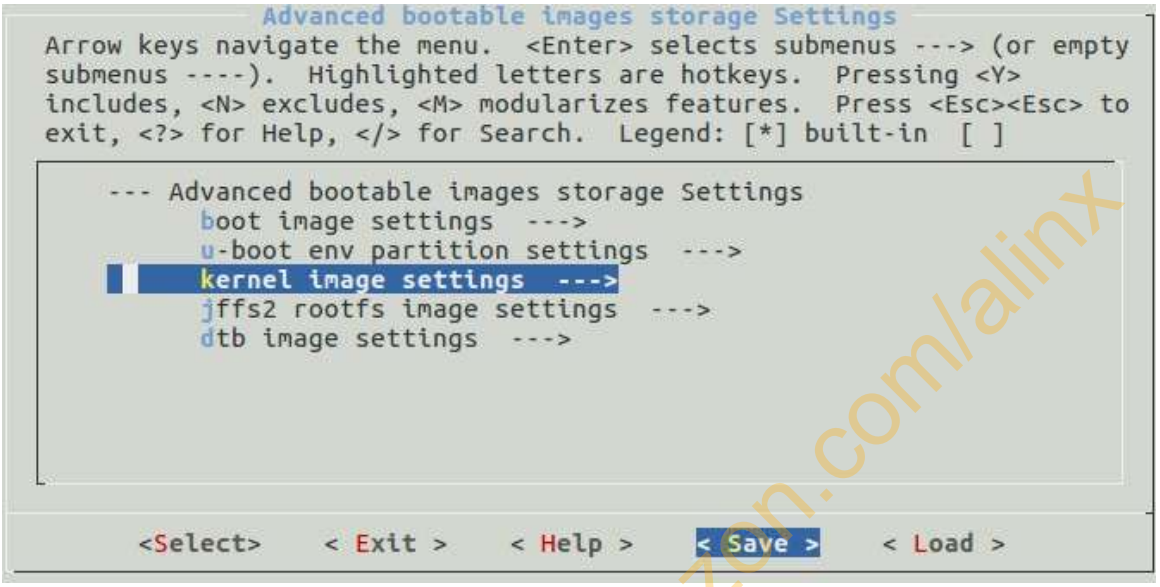
10) The peripherals and startup modes can be configured in the option "Subsystem AUTO Hardware Settings". This experiment keeps the mode.

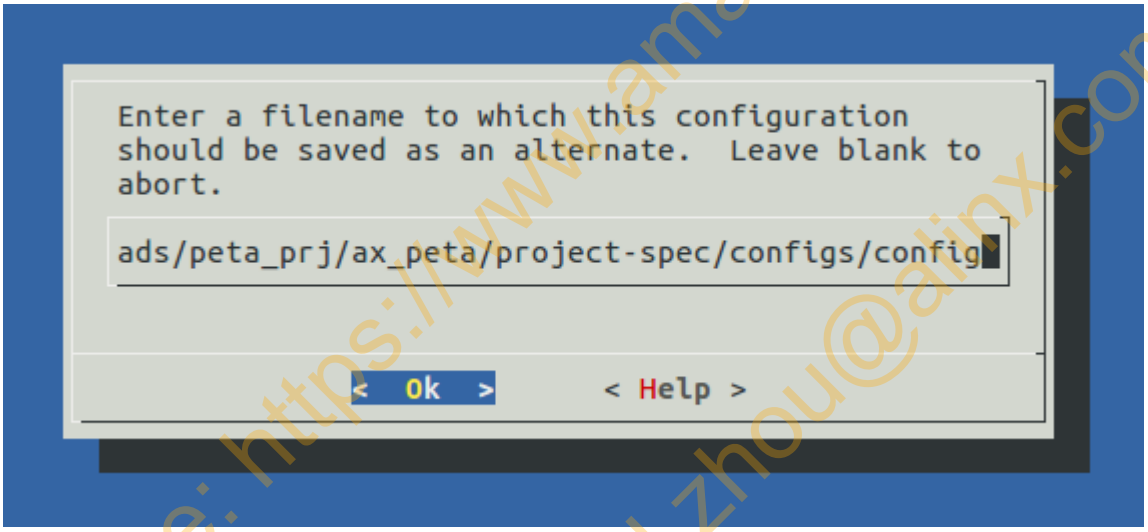


11)The boot mode is configured in the "Advanced bootable images storage Settings" option. The default is to boot from the sd card. For debugging, keep the default boot from the sd card. If you need to create an embedded "Linux" booted from "QSPI flash", you can configure it here.

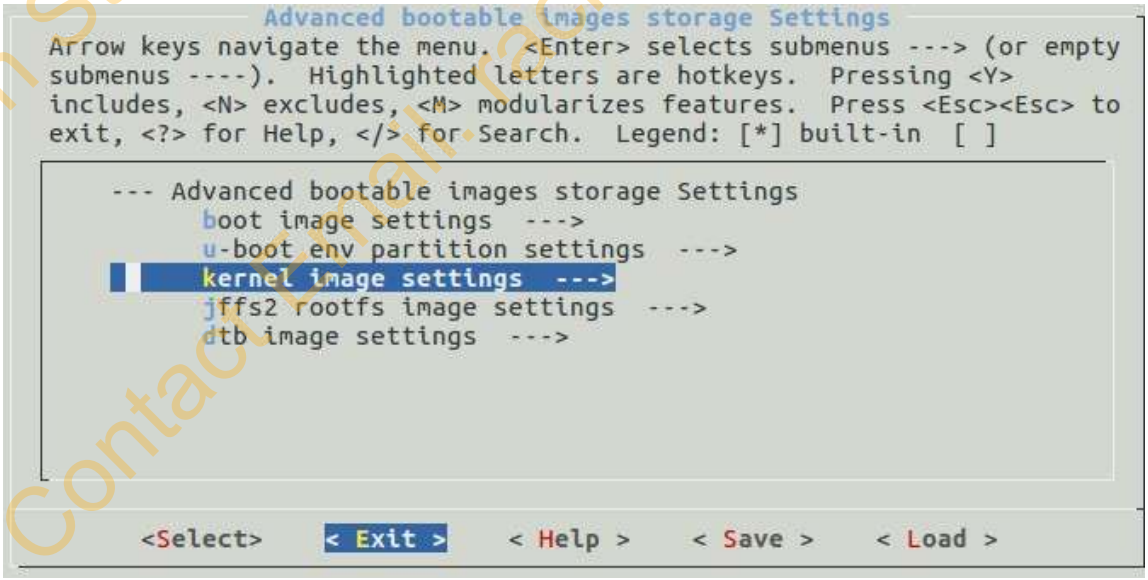


12)Keep the settings after the configuration is complete, this experiment is basically the default configuration

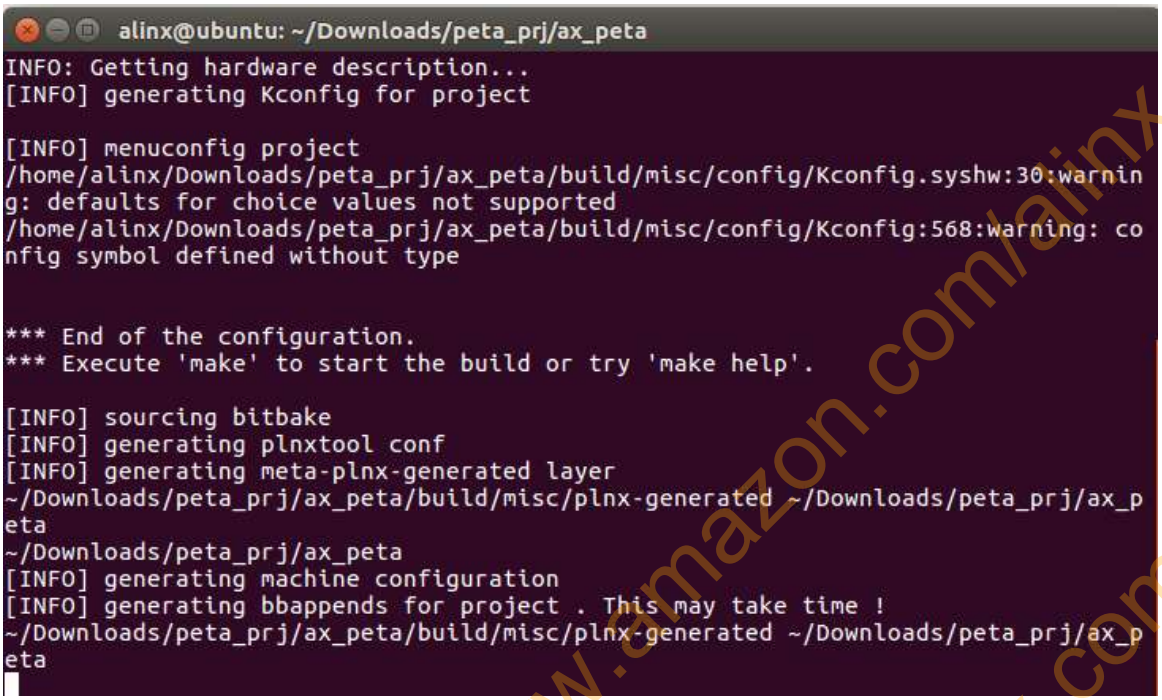




13)Exit the configuration interface



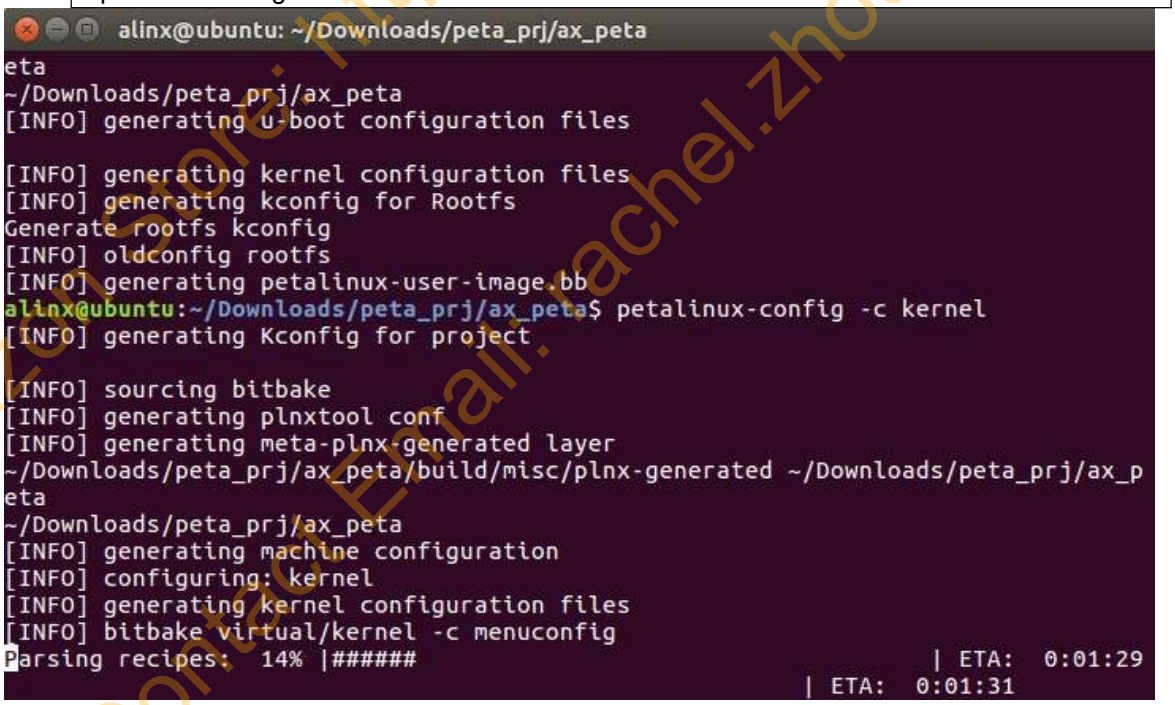
14)Start a long wait



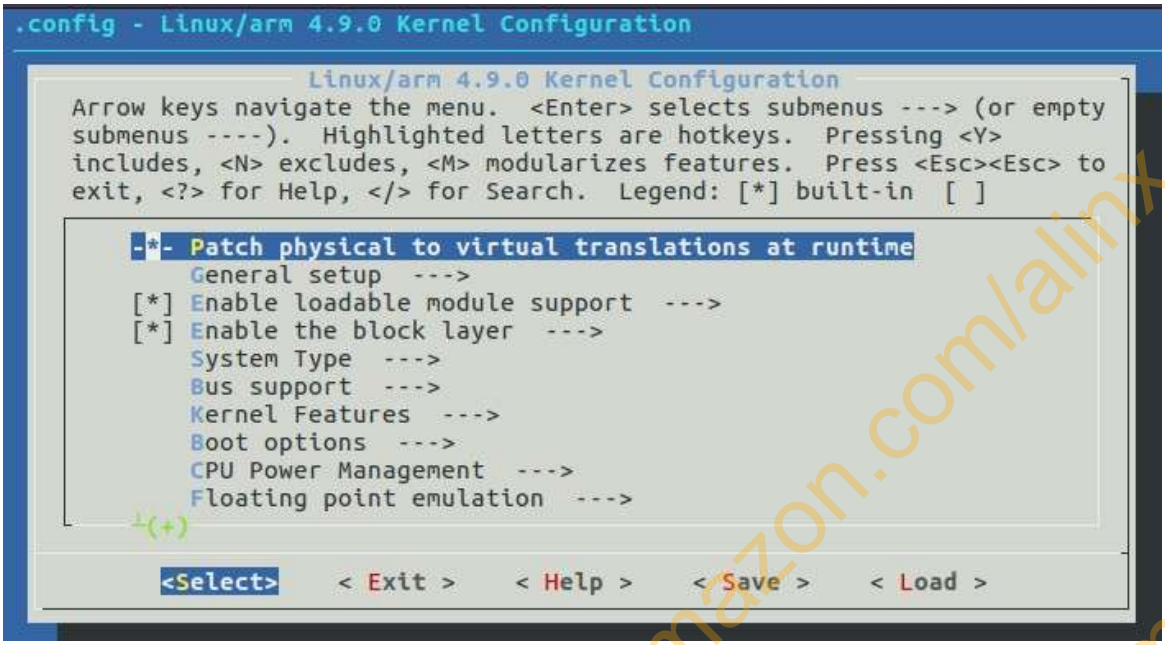
Part 18.3：Configuering the Linux kernel

1) Use the following command to configure the kernel. After running the command, wait for a long time.

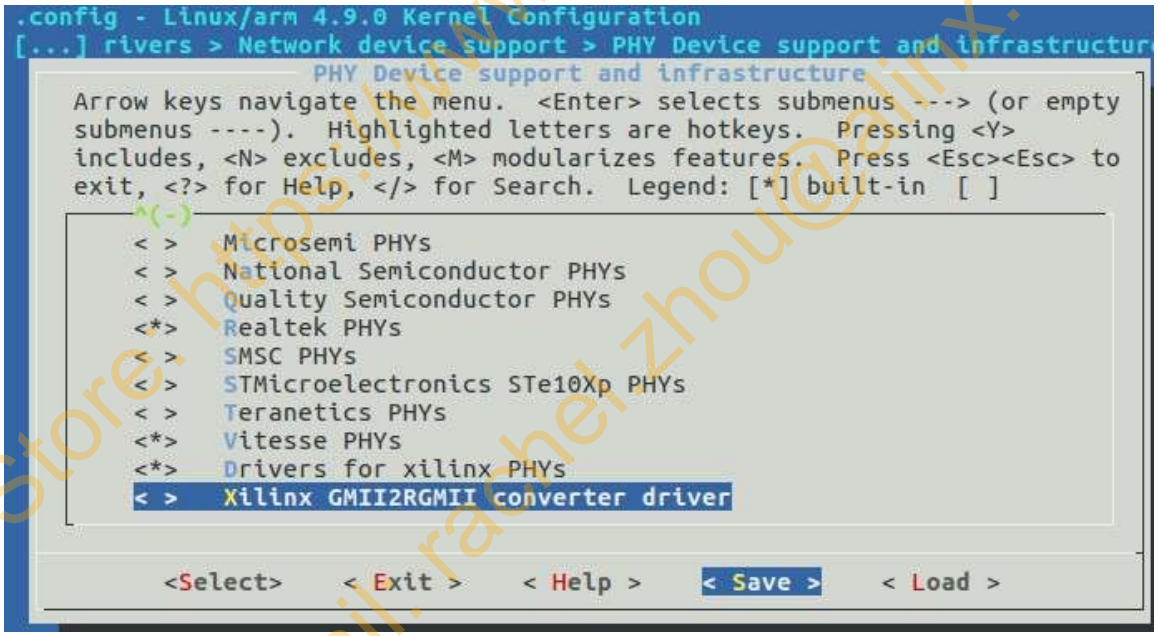
```
petalinux-config -c kernel
```



2) After waiting for a while, the configuration interface of the configuration kernel pops up.



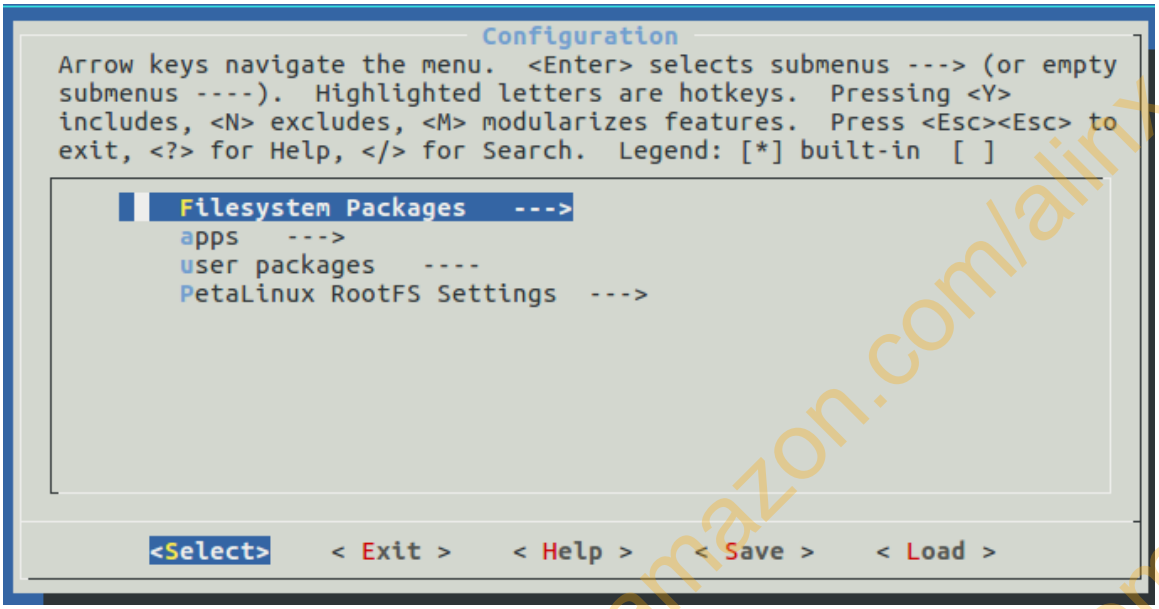
3) Others do not need to be configured again, save the configuration and exit



Part 18.4: Configuring the root file system

Run the following command to configure the root file system. You can configure the root file system according to your requirements. This experiment maintains the default configuration.

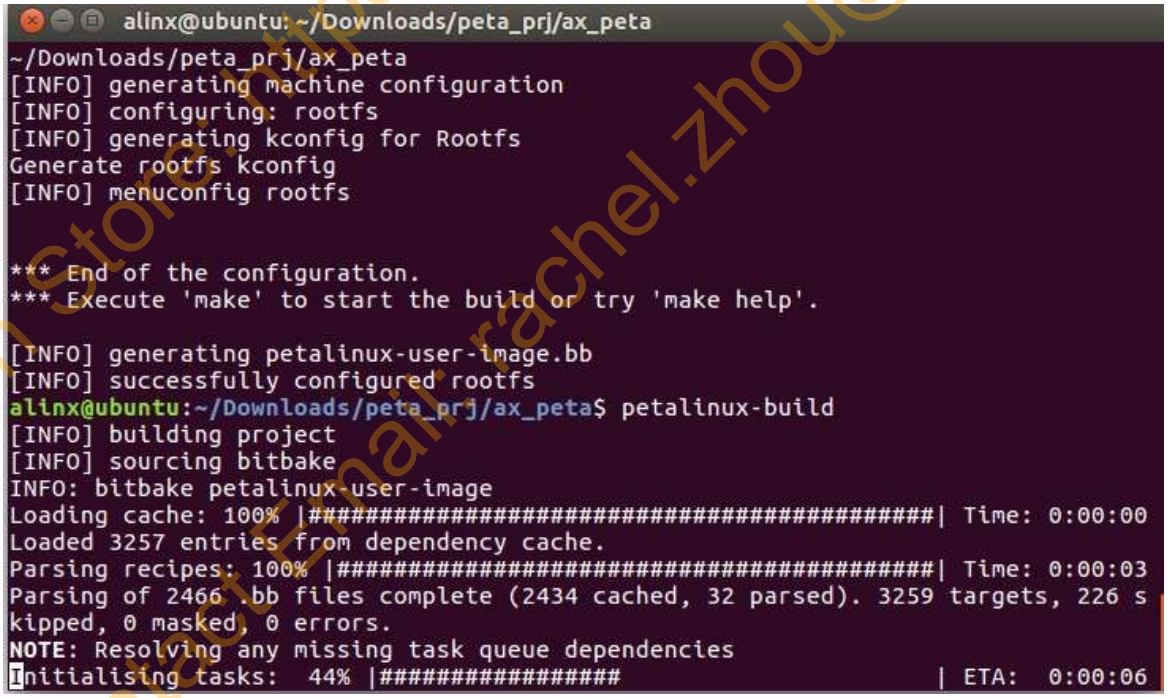
```
petalinux-config -c rootfs
```



Part18.5: Compile

1) Use the following command to configure uboot, kernel, root file system, device tree, and so on.

```
petalinux-build
```



2) Compiled completed

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
[INFO] successfully configured rootfs
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-build
[INFO] building project
[INFO] sourcing bitbake
INFO: bitbake petalinux-user-image
Loading cache: 100% |############################################| Time: 0:00:00
Loaded 3257 entries from dependency cache.
Parsing recipes: 100% |##########################################| Time: 0:00:03
Parsing of 2466 .bb files complete (2434 cached, 32 parsed). 3259 targets, 226 s
kipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |########################################| Time: 0:00:08
Checking sstate mirror object availability: 100% |###############| Time: 0:32:53
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
fsbl-2017.4+gitAUTOINC+77448ae629-r0 do_compile: NOTE: fsbl: compiling from exte
rnal source tree /opt/pkg/petalinux/tools/hsm/data/embeddedsw
NOTE: Tasks Summary: Attempted 2444 tasks of which 1884 didn't need to be rerun
and all succeeded.
INFO: Copying Images from deploy to images
INFO: Creating images/linux directory
NOTE: Failed to copy built images to tftp dir:  /tftpboot
[INFO] successfully built project
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

Part 18.6: Generate BOOT file

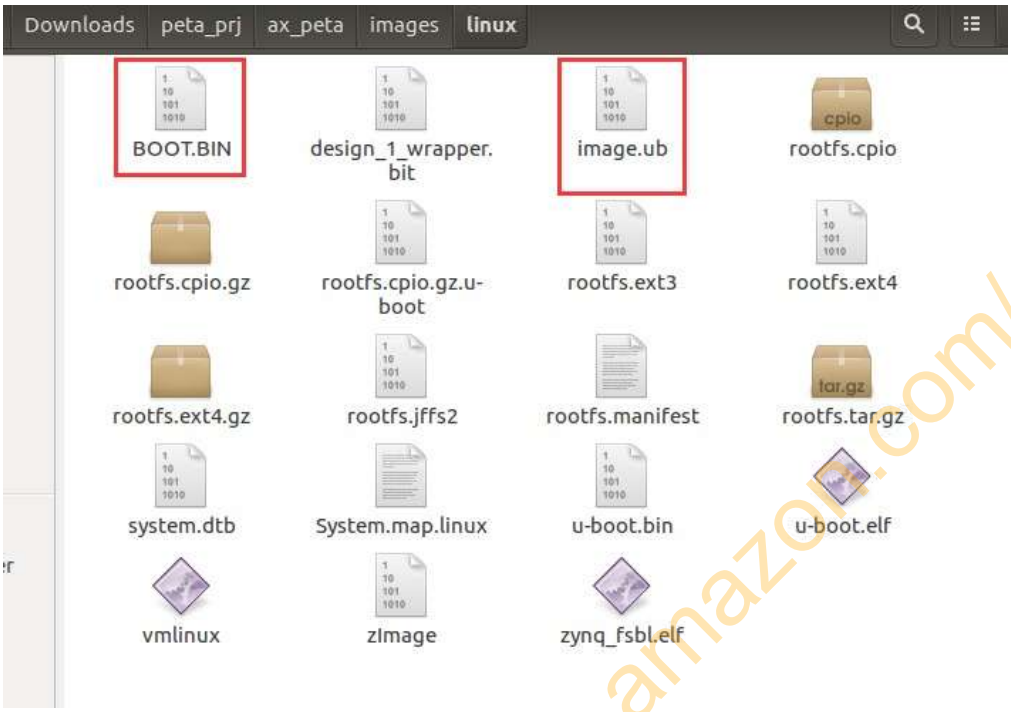
Run the following command to generate a BOOT file, paying attention to spaces and short lines.

```
petalinux-package--boot--fsbl./images/linux/zynq_fsbl.elf  --fpga  --u-boot  --
```

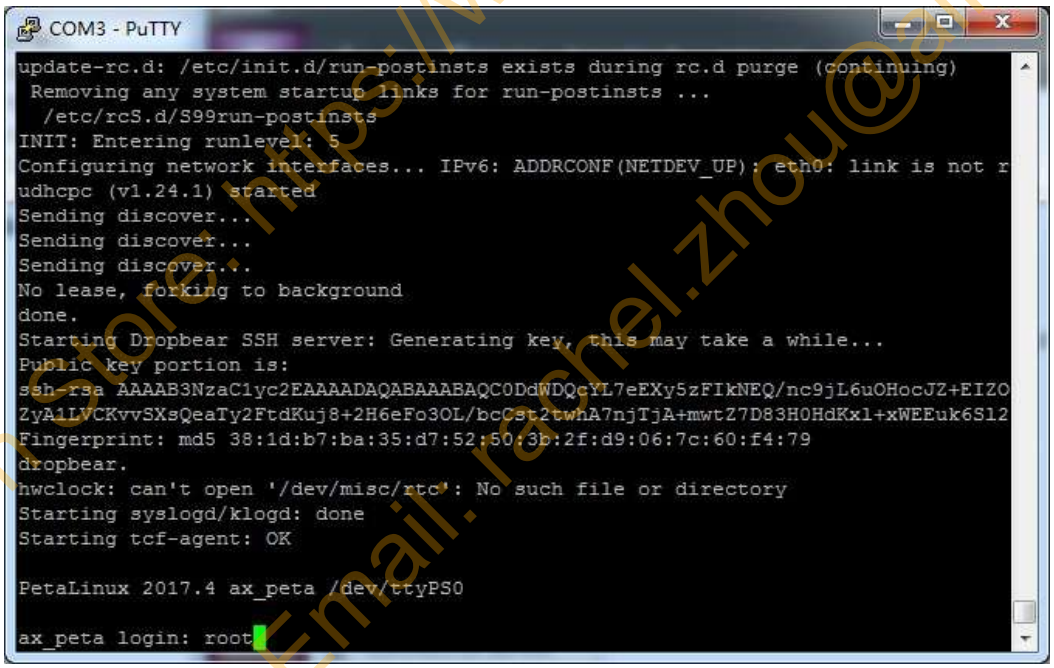
```
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-package --boot --fsbl ./ima
ges/linux/zynq_fsbl.elf  --fpga  --u-boot --force

INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/zyn
q_fsbl.elf"
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/des
ign_1_wrapper.bit"
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/u-b
oot.elf"
INFO: Generating zynq binary package BOOT.BIN...
INFO: Binary is ready.
WARNING: Unable to access the TFTPBOOT folder /tftpboot!!!
WARNING: Skip file copy to TFTPBOOT folder!!!
webtalk failed:Invalid tool in the statistics file:petalinux-yocto!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$
```

Part 18.7: Testing Linux

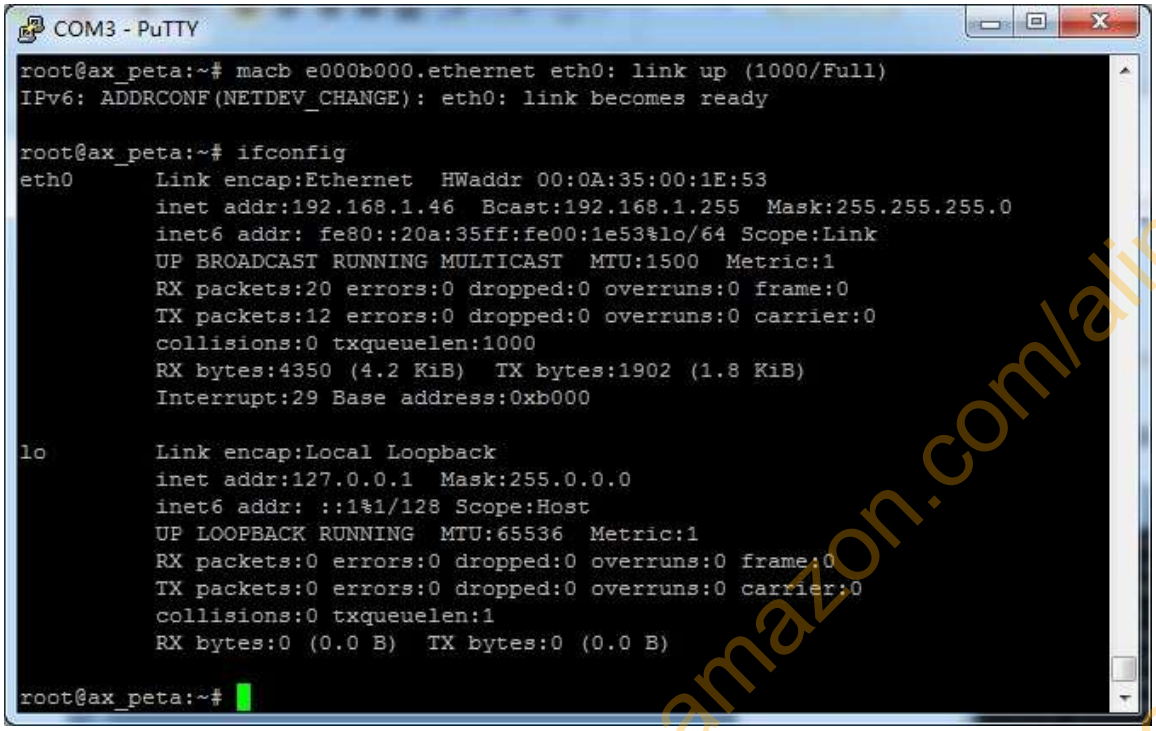
1) Copy BOOT.BIN and image.ub from the project directory images -> linux directory to the sd card. Before copying, it is best to format the sd card first, then plug it into the FPGA development board, and set the FPGA development boardto sd card boot



2) Open the serial terminal and power on the FPGA development board.

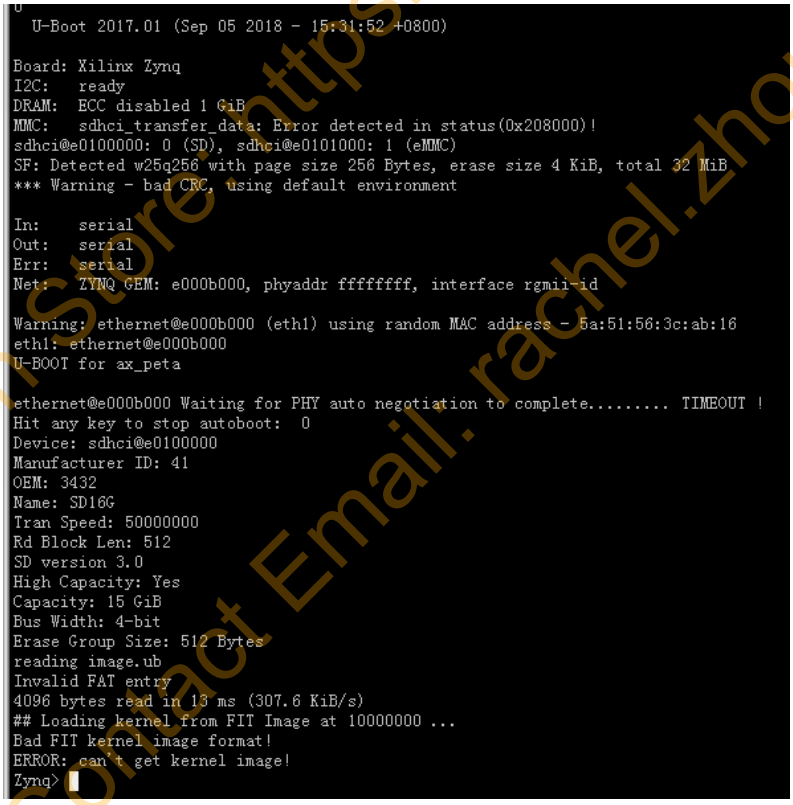


3) Use root login, the default password root, After plugging in the Internet cable (the router supports automatic IP acquisition), you can use the ifconfig command to see the network status



Part 18.8: Q&A

Part 18.8.1: Prompt "Bad FIT kernel image format!" cannot start the kernel



Solution:

Format the sd card fat32 partition again and reposition the startup file

Part 18.8.2: Unable to save file and configuration

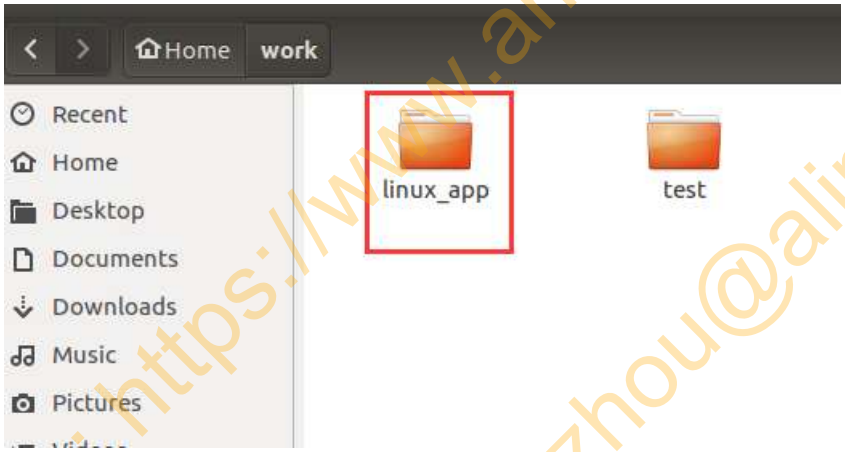
Because petalinux's default file system is a RAM type, it cannot be saved. In the subsequent tutorials, it can be set to the SD card type, and the data can be saved to the SD card.

Part 19: Develop Linux programs using the SDK

In the previous tutorial we used petalinux to create an embedded Linux system. This experiment will be a Linux application and then run on the FPGA development board. This experiment requires the use of the Linux operating environment in the above experiment.

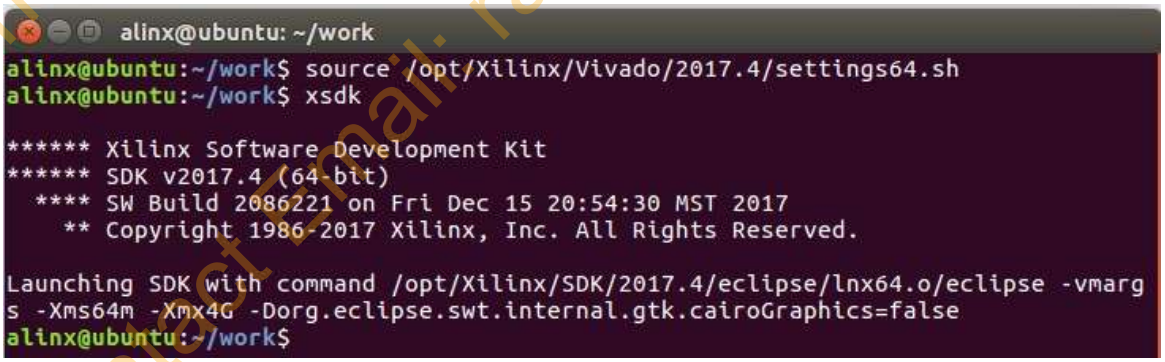
Part 19.1: Create a Linux application using the SDK

1) In /home/alinx/work, create the directory linux_app for the SDK workspace

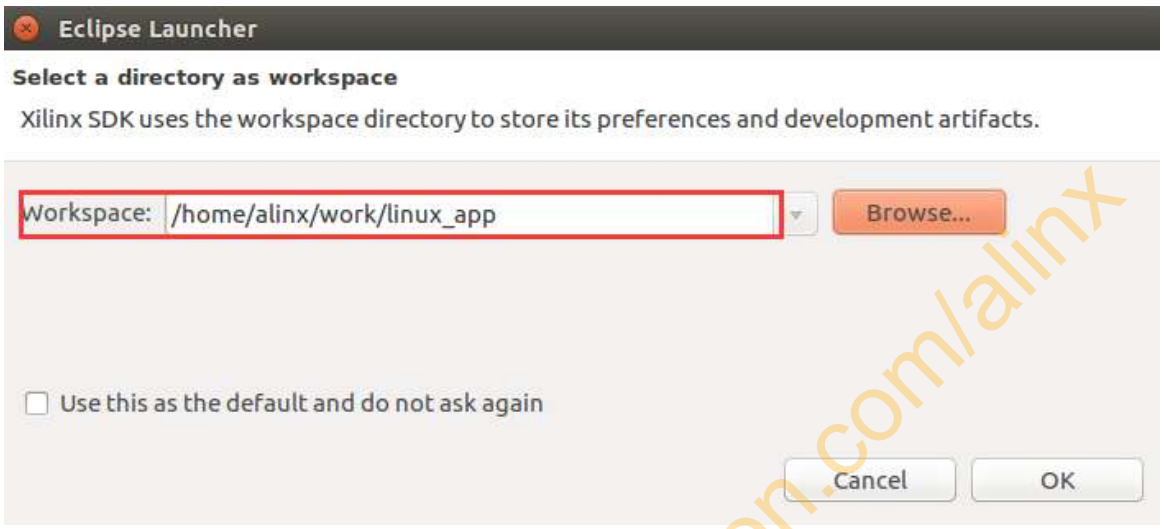


2) Set the Vivado environment variable to run the SDK

```
source /opt/Xilinx/Vivado/2017.4/settings64.sh
xsdk
```

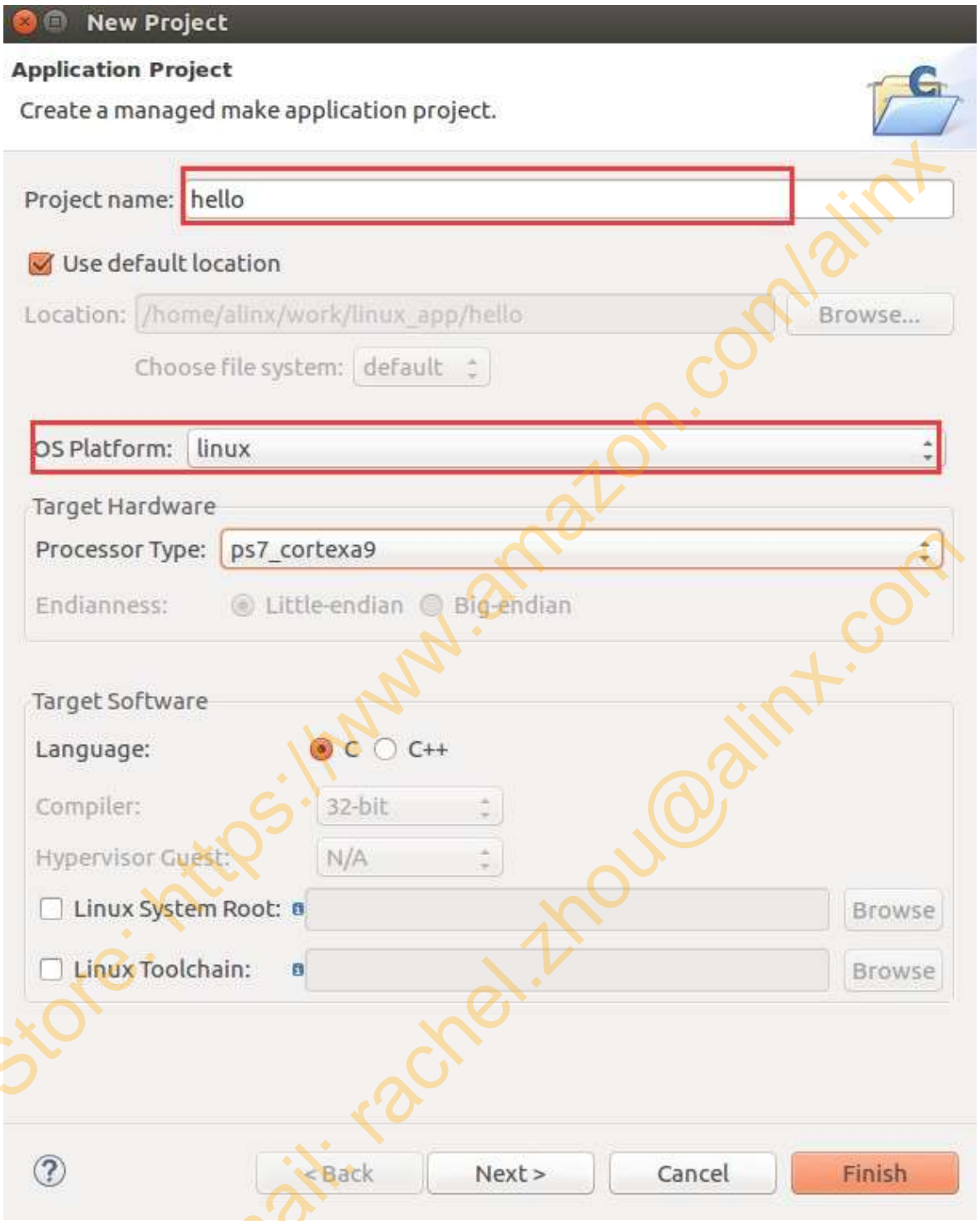


3) Workspace select /home/alinx/work/linux_app

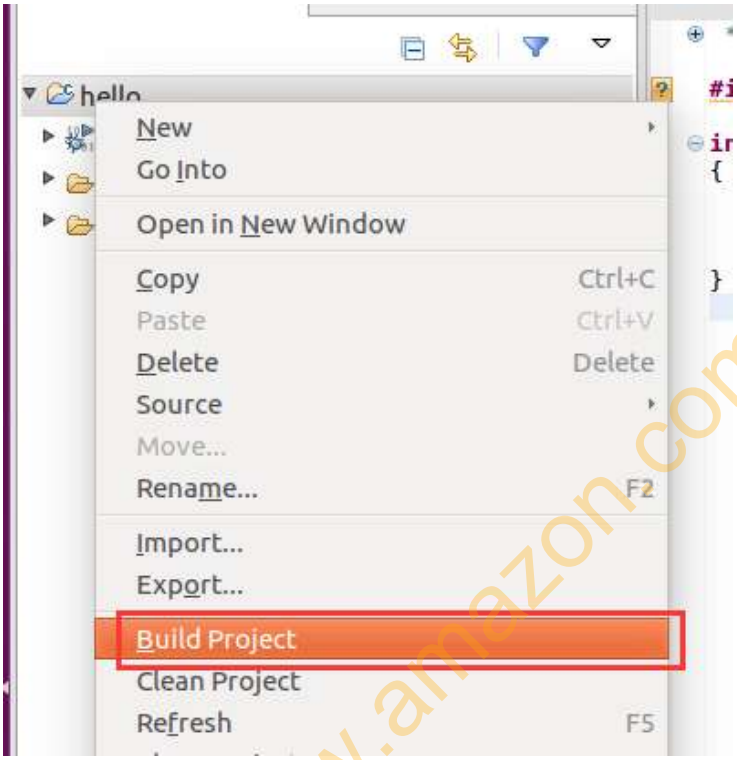


4) Select “Create Application Project”

5) Fill in the project name “hello”, “OSPlatform” select “linux”



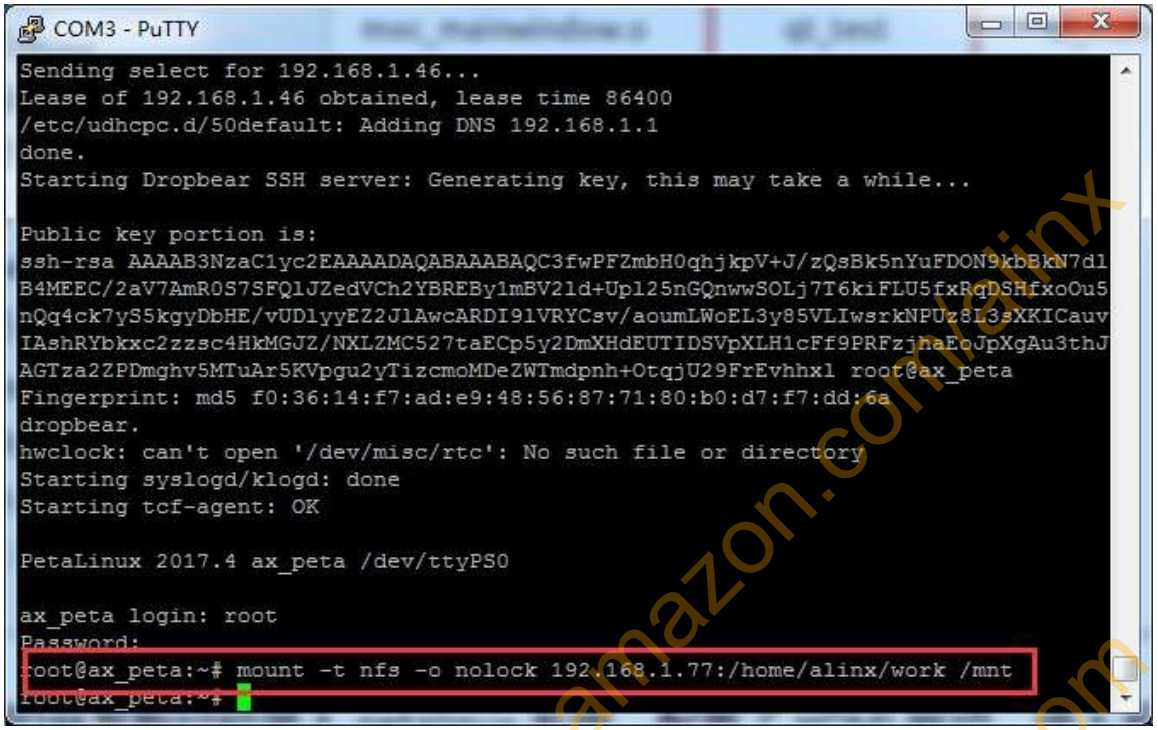
6) BuildProject



Part 19.2: Run through NFS share

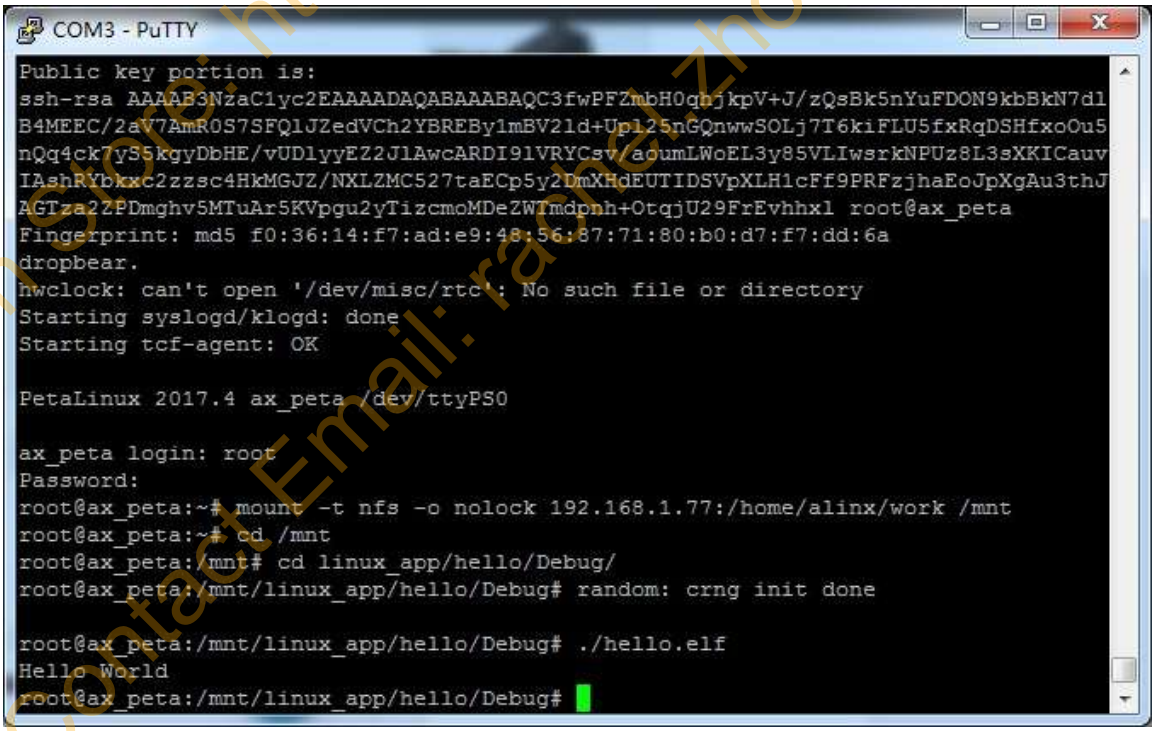
1) The FPGA development board plugs in the Internet cable (PS, ETH1, requires the router to support automatic IP acquisition), Power-on, mounts the Linux host NFS, where 192.168.1.77 is the host address, /home/alinx/work is the NFS directory, and /mnt is the FPGA Developemen Board directory. Here, the host and the FPGA development board are required to be on one network segment.

```
mount -t nfs -o nolock 192.168.1.77:/home/alinx/work /mnt
```



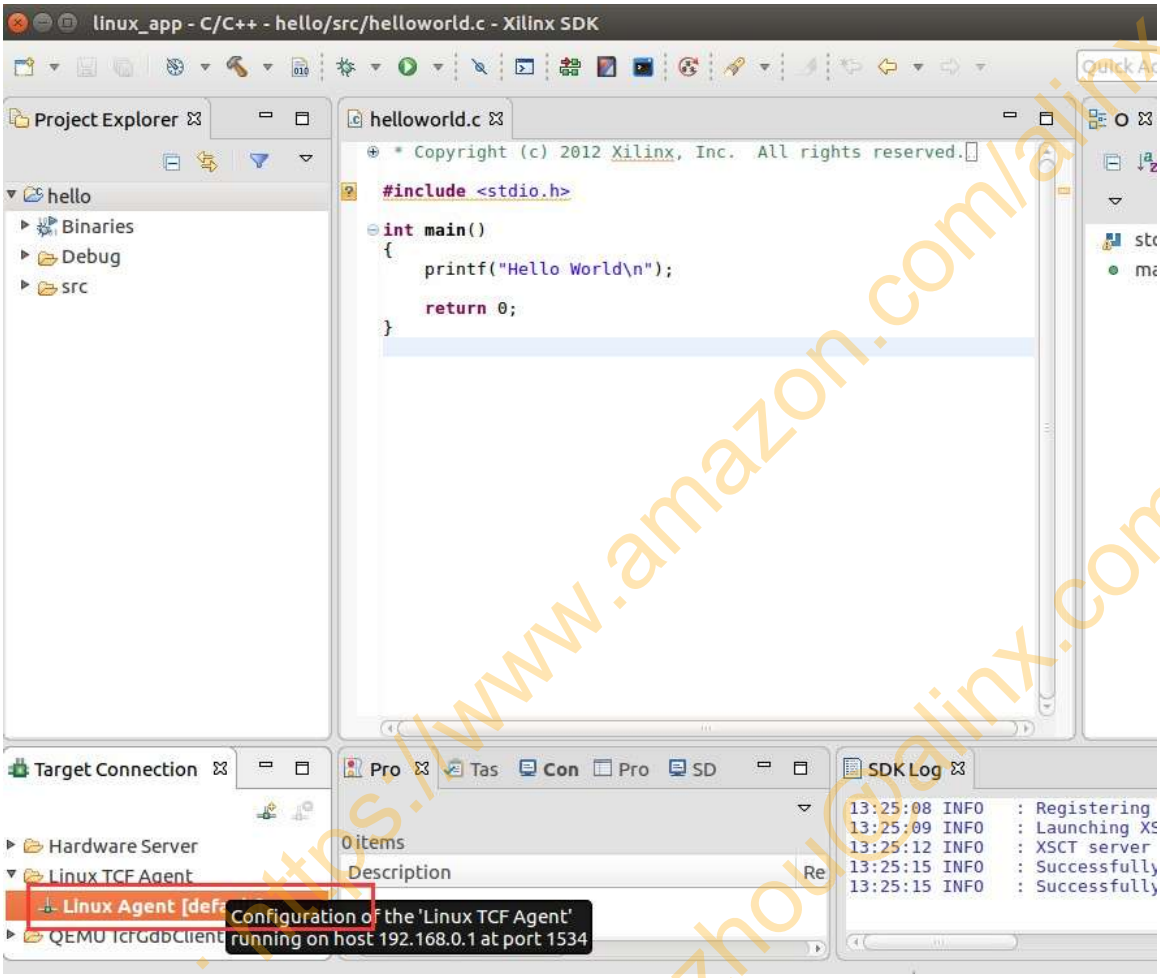
2) Go to the /mnt/linux_app/hello/Debug directory and run hello.elf to print out HelloWorld.

```
cd    /mnt/linux_app/hello/Debug
./ hello.elf
```

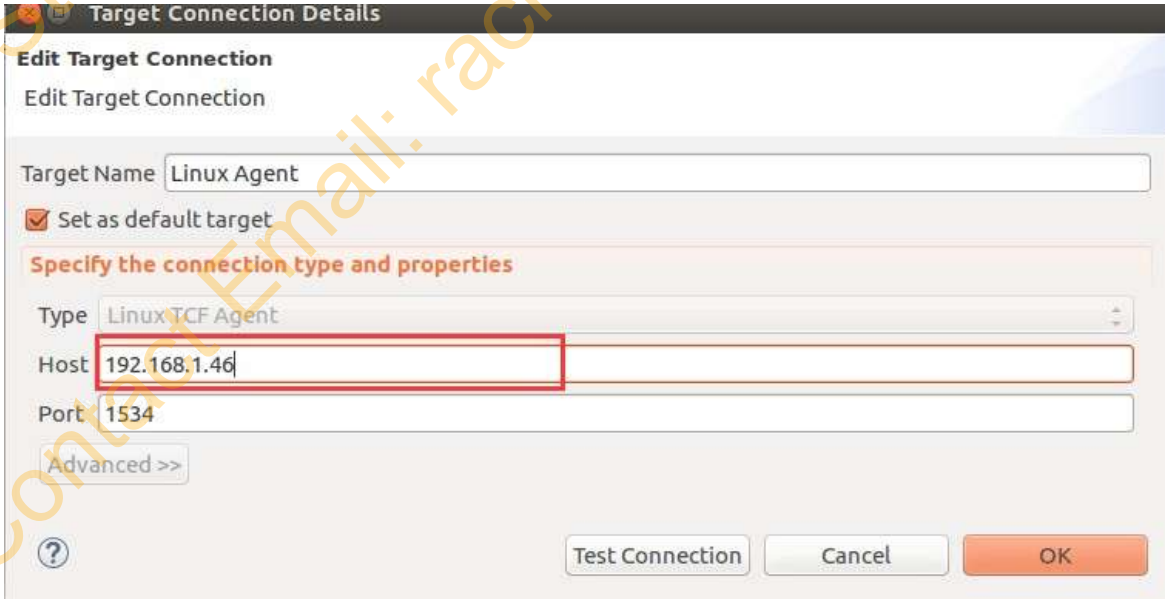


Part 19.3: Run debugging through TCF-Agent

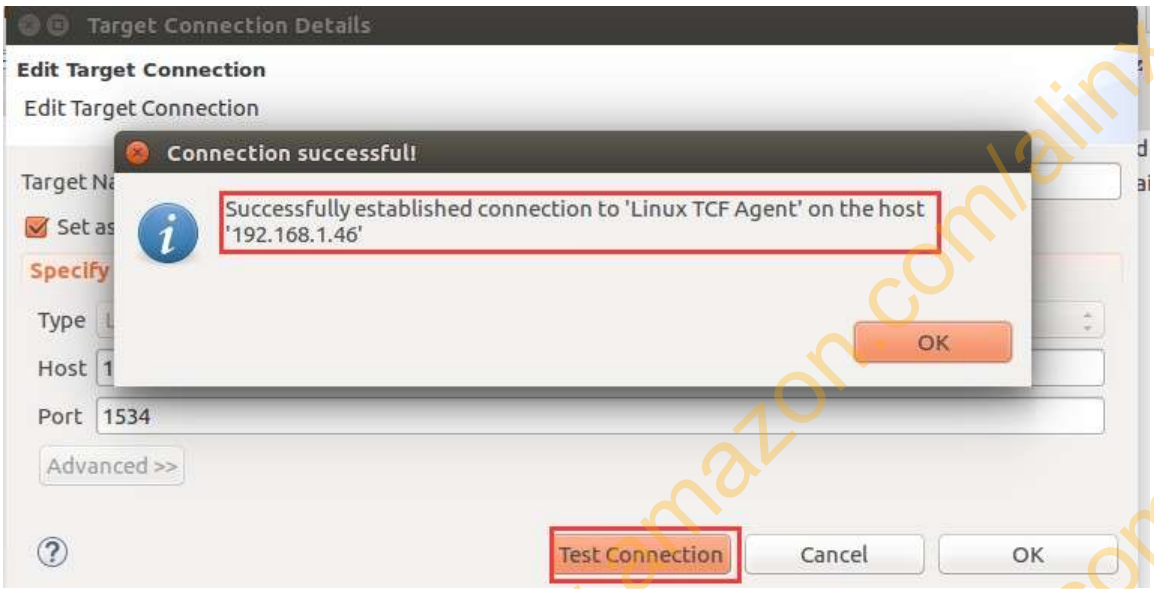
1) Double-click “LinuxAgent” to configure connection parameters.



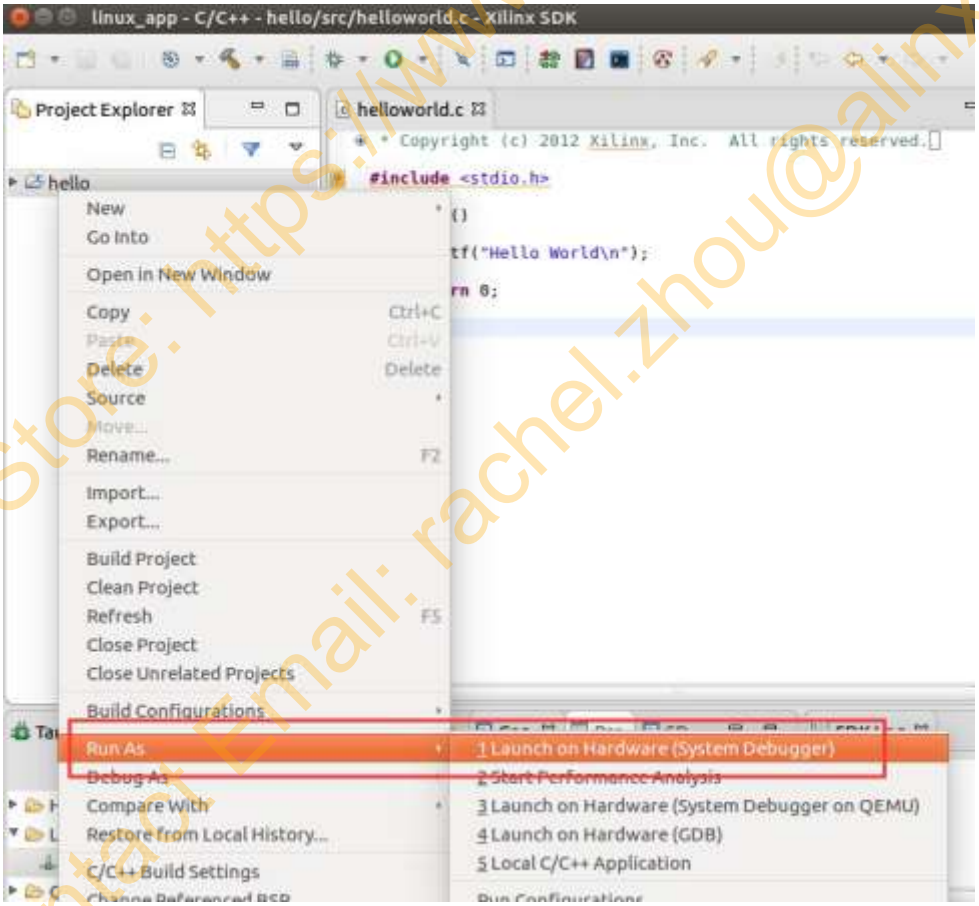
2) Fill in the IP address of the Host, here is the IP address of the FPGA development board.



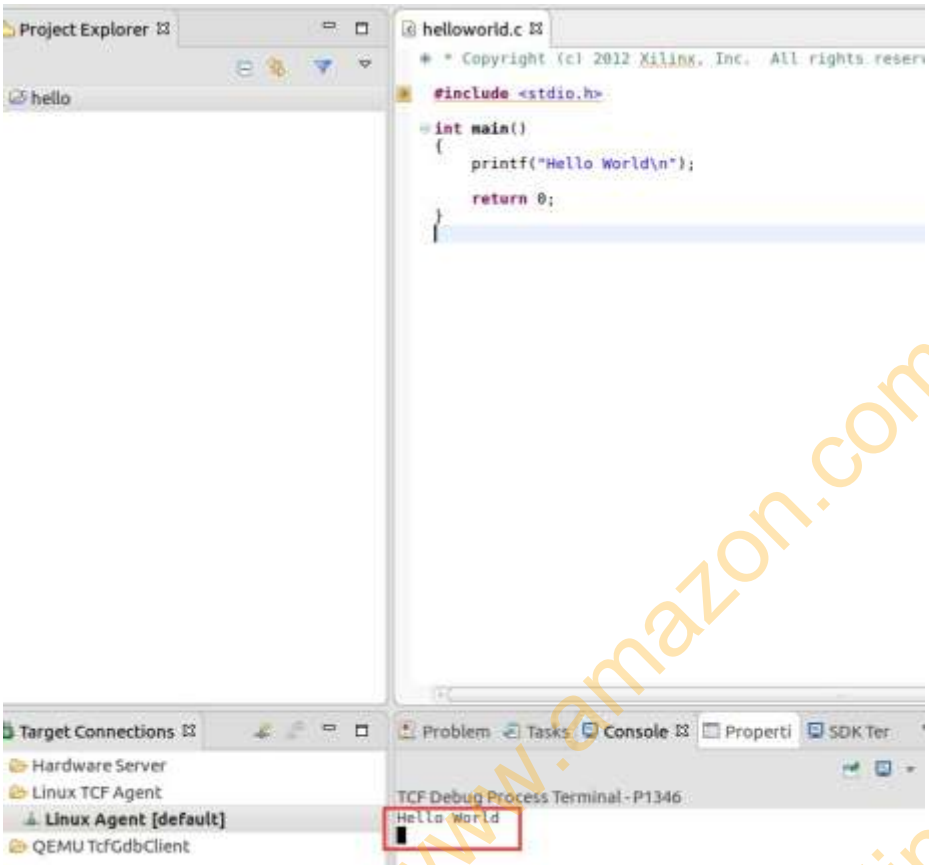
3) Click TestConnection. If the test connection is successful, the LinuxTCF Agent service is running and you can run the debug.



4) Select project, right click



5) You can see the HelloWorld output in the debug terminal.



Part 19.4: TCF-Agent problem

Although TCF-Agent can easily run and debug Linux applications and does not require NFS support, it is not very good for debugging multi-threaded programs. It cannot restore the debugging environment well when the application crashes. You need to restart the FPGA development board. So it has not been used often.

Part 20: GPIO experiment under Linux

The previous tutorial introduced how to use the SDK to write a zynq version of the helloworld experiment. This experiment describes how to control the zynq peripherals. Experiments use GPIO as an example. ZYNQ's GPIO can be divided into two types, one is the GPIO that comes with the PS. One is the GPIO implemented with PL. The GPIO IP of Xilinx is added when building the Vivado project. Most of the IP cores provided by Xilinx are already driven under Linux, and many default configurations are available, like AXI GPIO drivers. It does not need to be reconfigured in the kernel to be used.

On the http://www.wiki.xilinx.com/Linux+Drivers page we can find all Xilinx drivers for Linux. For example, the GPIO drivers are shown below, and some drivers give detailed usage

GPIO	Zynq and Zynq Ultrascale+ MPSoC	GPIO Driver	Yes	drivers/gpio/gpio-zynq.c
GPIO	axi_gpio	AXI GPIO Driver	Yes	drivers/gpio/gpio-xilinx.c
HDMI Clocks	SI5324 Clock Multiplier/Jitter Attenuator	CCF SI5324 Driver	No	hdmi-modules/clk/*

The GPIO driver usage scope, device tree examples, and how to write programs are described in the GPIO driver details page http://www.wiki.xilinx.com/Linux GPIO Driver.

Part 20.1: Use SHELL control

Linux provides powerful SHELL function, and it is also a skill that must be mastered when learning Linux. It is a headache for unfamiliar with Linux commands and SHELL, but in order to learn ZYNQ better, you must be familiar with SHELL. This tutorial will not explain Linux in detail. And the use of SHELL

The GPIO number can be viewed by the “ls /sys/class/gpio” command.

```
root@zynq:~# ls /sys/class/gpio
export   gpio900  gpio903  gpio906  gpio957       gpiochip898
gpio898  gpio901  gpio904  gpio919  gpiochip1016  unexport
gpio899  gpio902  gpio905  gpio956  gpiochip1020
```

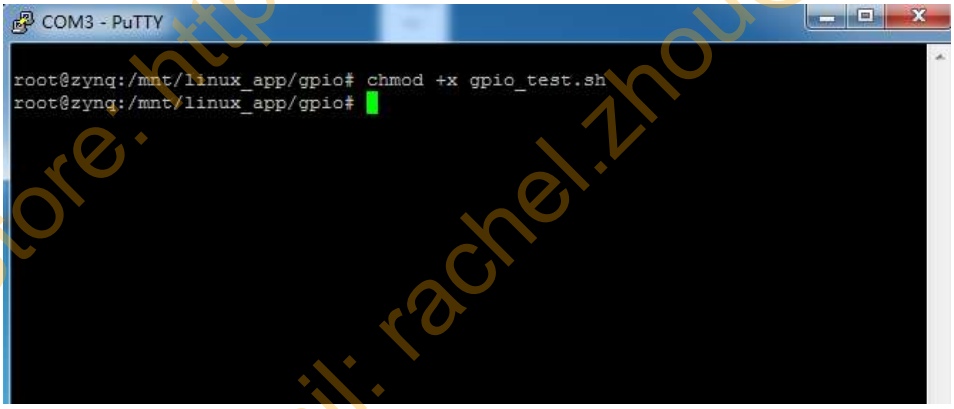
The contents of the gpio_test.sh file are as follows. The gpio_test function will export a GPIO according to the parameters, then a for loop 3 times, each time writing 0 first and then writing 1, calling 5 times gpio_test, lighting 5 LEDs in turn, 898 is PS End, the other is the PL side.

```
#!/bin/sh
gpio_test() {
    gpio=$1
    echo$gpio> /sys/class/gpio/export
    echoout > /sys/class/gpio/gpio${gpio}/direction
    for i in $(seq 1 3)
    do
    echo0 >/sys/class/gpio/gpio${gpio}/value
    sleep1
    echo1 >/sys/class/gpio/gpio${gpio}/value
    sleep1
    done
    echo$gpio> /sys/class/gpio/unexport
}
gpio_test 898
gpio_test1016
gpio_test1017
gpio_test1018
gpio_test1019
```

We can run this SHELL by mounting NFS

If SHELL can't run, you can add run permissions first, the command is as follows:

```
chmod +x gpio_test.sh
```



Part 20.2: Use C language to control peripherals

In most cases we need to use C to control peripherals on Xilinx's wiki page

http://www.wiki.xilinx.com/GPIO%20User%20Space%20App

We found a GPIO test code with the following contents:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>


// The specific GPIO being used must be setup and replaced thru
```

```
// this code.  The GPIO of 898 is in the path of most the sys dirs
// and in the export write.
//
// Figuring out the exact GPIO was not totally obvious when there
// were multiple GPIOs in the system. One way to do is to go into
// the gpiochips in /sys/class/gpio and view the label as it should
// reflect the address of the GPIO in the system. The name of the
// the chip appears to be the 1st GPIO of the controller.
//
// The export causes the gpio898 dir to appear in /sys/class/gpio.
// Then the direction and value can be changed by writing to them.

// The performance of this is pretty good, using a nfs mount,
// running on open source linux,
// the GPIO can be toggled about every 1sec.
// The following commands from the console setup the GPIO to be
// exported, set the direction of it to an output and write a 1
// to the GPIO.
//
// bash> echo 898 > /sys/class/gpio/export
// bash> echo out > /sys/class/gpio/gpio898/direction
// bash> echo 1 > /sys/class/gpio/gpio898/value

// if sysfs is not mounted on your system, the you need to mount it
// bash> mount -t sysfs sysfs /sys

// the following bash script to toggle the gpio is also handy for
// testing
//
// while [ 1 ]; do
//  echo 1 > /sys/class/gpio/gpio898/value
//  echo 0 > /sys/class/gpio/gpio898/value
// done

// to compile this, use the following command
// gcc gpio.c -o gpio

// The kernel needs the following configuration to make this work.
//
// CONFIG_GPIO_SYSFS=y
// CONFIG_SYSFS=y
// CONFIG_EXPERIMENTAL=y
// CONFIG_GPIO_XILINX=y

int main()
{
int valuefd, exportfd, directionfd;

    printf("GPIO test running...\n");

// The GPIO has to be exported to be able to see it
// in sysfs

    exportfd = open("/sys/class/gpio/export", O_WRONLY);
if(exportfd <0)
{
        printf("Cannot open GPIO to export it\n");
        exit(1);
}

    write(exportfd,"898",4);
    close(exportfd);

    printf("GPIO exported successfully\n");

// Update the direction of the GPIO to be an output

    directionfd = open("/sys/class/gpio/gpio898/direction", O_RDWR);
if(directionfd <0)
{
        printf("Cannot open GPIO direction it\n");
        exit(1);
}

    write(directionfd,"out",4);
    close(directionfd);

    printf("GPIO direction set as output successfully\n");

// Get the GPIO value ready to be toggled

    valuefd = open("/sys/class/gpio/gpio898/value", O_RDWR);
    if (valuefd < 0)
    {
        printf("Cannot open GPIO value\n");
        exit(1);
    }

    printf("GPIO value opened, now toggling...\n");

    // toggle the GPIO as fast a possible forever, a control c is needed
    // to stop it

    while (1)
    {
        write(valuefd,"1", 2);
```

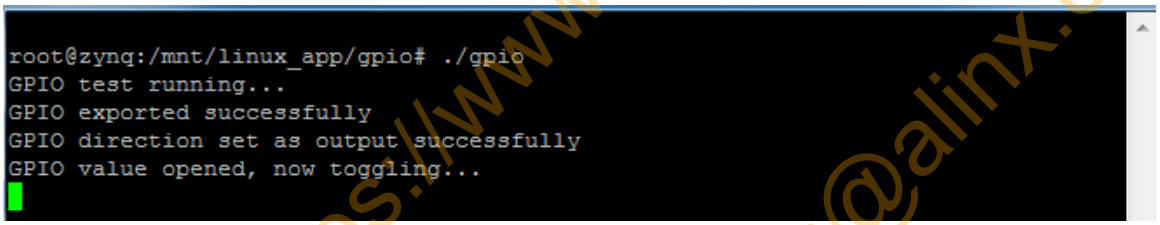
```
                sleep(1);
        write(valuefd,"0", 2);
                sleep(1);
    }
}
```

This time we no longer use the SDK to compile, the source code is named "gpio.c", run the following command to compile the code

```
source /opt/Xilinx/Vivado/2017.4/settings64.sh arm-
linux-gnueabihf-gcc      gpio.c -ogpio
```

After the compilation is completed, a gpio file will be generated. Unlike Windows, the extension name is not required so strict inLinux,the gpio file is an elf file.

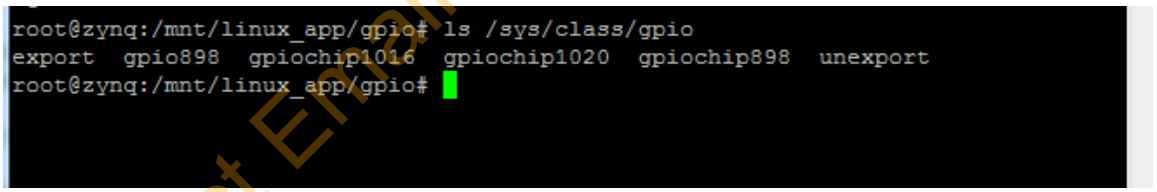
Running gpio, you can see that the PS LED flashes continuously, indicated that this 898 is the first LED on the PS side.



Part 20.2.1: How to determine this number for GPIO?

With the following command, we can see gpiochip898 gpiochip1016 gpiochip1020, indicating that there are three GPIO controller, the number is the controller GPIO base.

```
ls /sys/class/gpio
```



Part 20.2.2: How to determine the relationship with physical GPIO?

Use the following command to determine the relationship between GPIO1016 and physical GPIO. You can see that the gpio node in the device

tree is "/amba_pl/gpio@41210000". We can determine which physical GPIO is through the node of the device tree.

```
cat /sys/class/gpio/gpiochip1016/label
```

```
root@zynq:/sys/class/gpio/gpiochip1016# cat /sys/class/gpio/gpiochip1016/label
/amba_pl/gpio@41210000
root@zynq:/sys/class/gpio/gpiochip1016#
```

Part 20.3: Experimental summary

The focus of this experiment is on how to learn ZYNQ through the information given by Xilinx. The technical information is updated quickly. Only the latest information provided by the chip manufacturer can get the latest and best technology. In the follow-up tutorials, PCIe drivers and PL-side Ethernet drivers are provided by Xilinx. These materials are available on the wiki.

If you use a non-xilinx IP, or your own IP, you have to develop your own driver, which is a challenge for developers who have not done Linux drivers, so we use the IP of Xilinx to build the system. The advantage is that you don't need to develop a Linux driver. The disadvantage is that it is not flexible enough. If there is a problem with the IP or a problem with the drive, you cannot quickly locate the problem.

Part 21: HDMI display under Petalinux

In the previous tutorial, we have already experienced the development of embedded Linux system using Petalinux, but the function we use is only the tip of Petalinux iceberg. The powerful features of Petalinux need us to explore continuously. This experiment explains how to use a kernel to do Linux. So we can add a lot of drivers in the kernel, such as HDMI display.

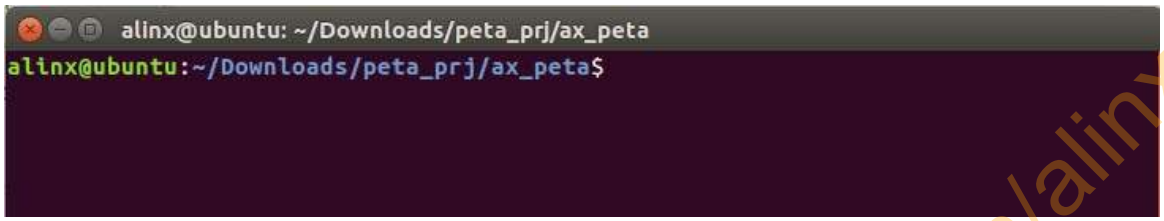
The HDMI interface chip used by the FPGA development board is the SIL9134. ALINX adds the driver for the HDMI interface chip to the kernel provided by Xilinx. Please use other versions of the software developer to note that this tutorial only provides a modified version of the linux-xlnx-xilinx-v2017.4 kernel, no other version of the modified version, and no modification instructions.

Part 21.1: Petalinux configuration

This experiment is still a modification of the Petalinux project in the previous experiment, you need to master the previous experimental content.

1) Copy the kernel source files to the Linux host, then decompress, the decompressed kernel directory is the kernel that Petalinux will use in this experiment.

2) Open the terminal and enter the Petalinux project directory in the previous experiment.

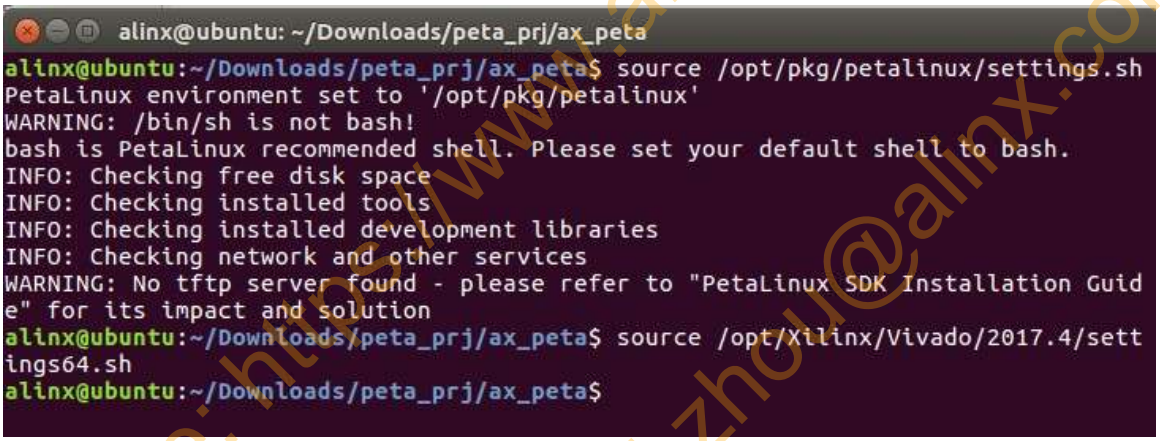


3) Set the petalinux environment variable, run the following command

```
source /opt/pkg/petalinux/settings.sh
```

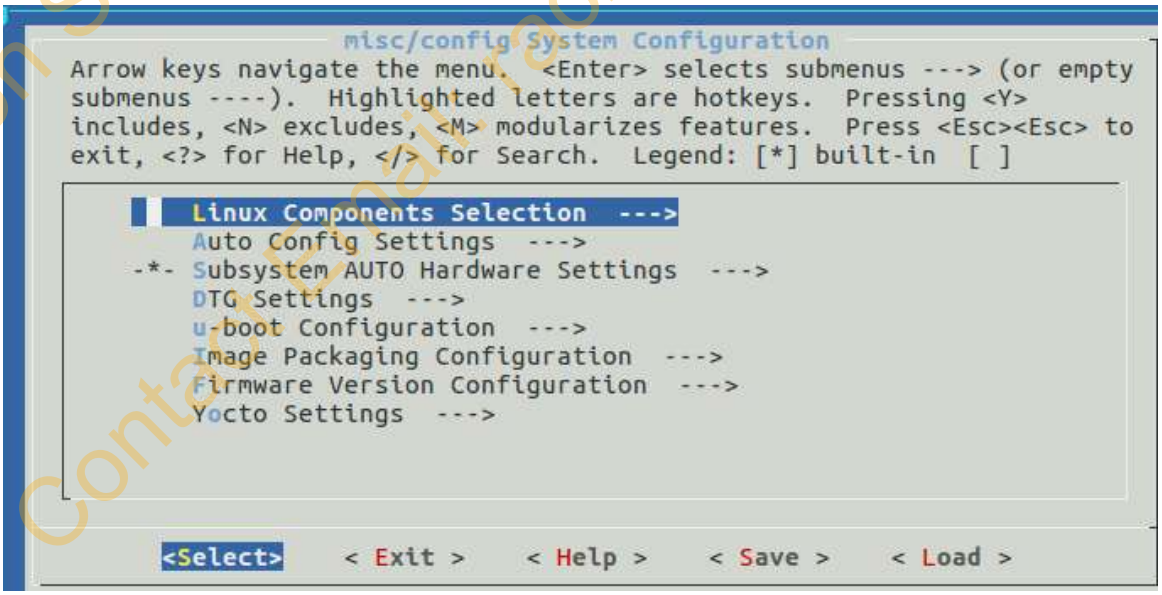
4) Run the following command to set the vivado environment variable

```
source /opt/Xilinx/Vivado/2017.4/settings64.sh
```

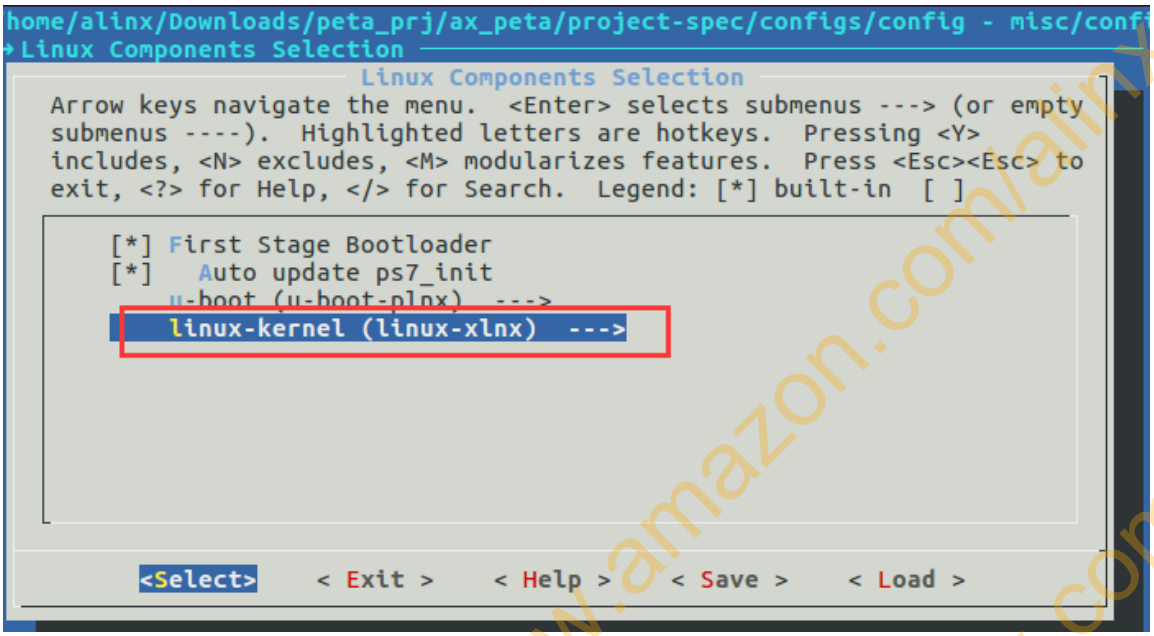


5) Use the following command to reconfigure petalinux

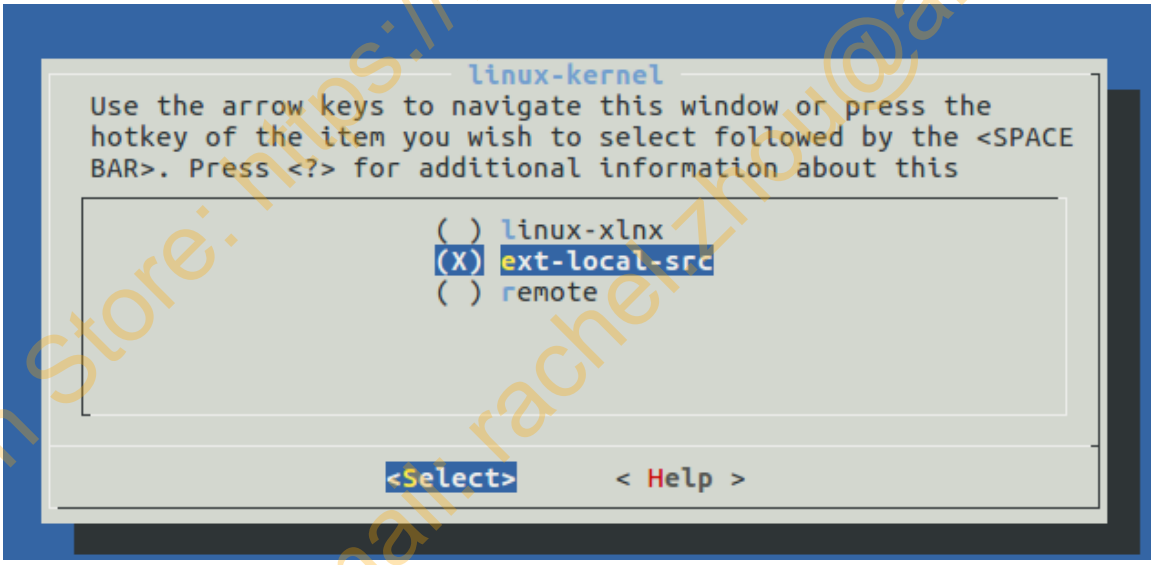
```
petalinux-config
```



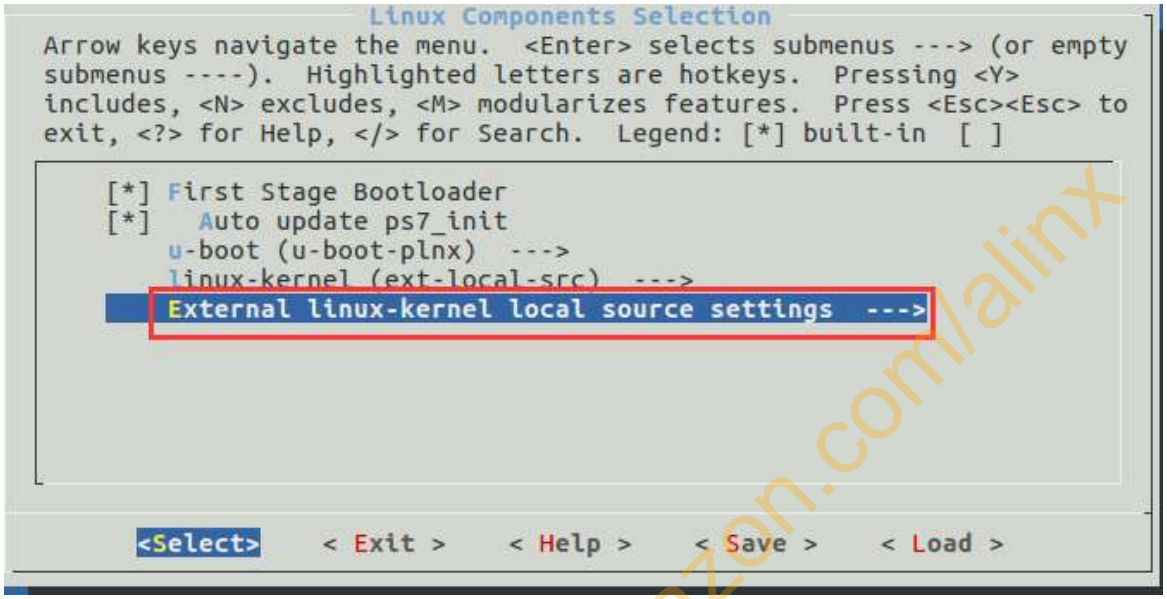
6) Select “Linux Components Selection--->linux-kernel(linux-xlnx) ---->”



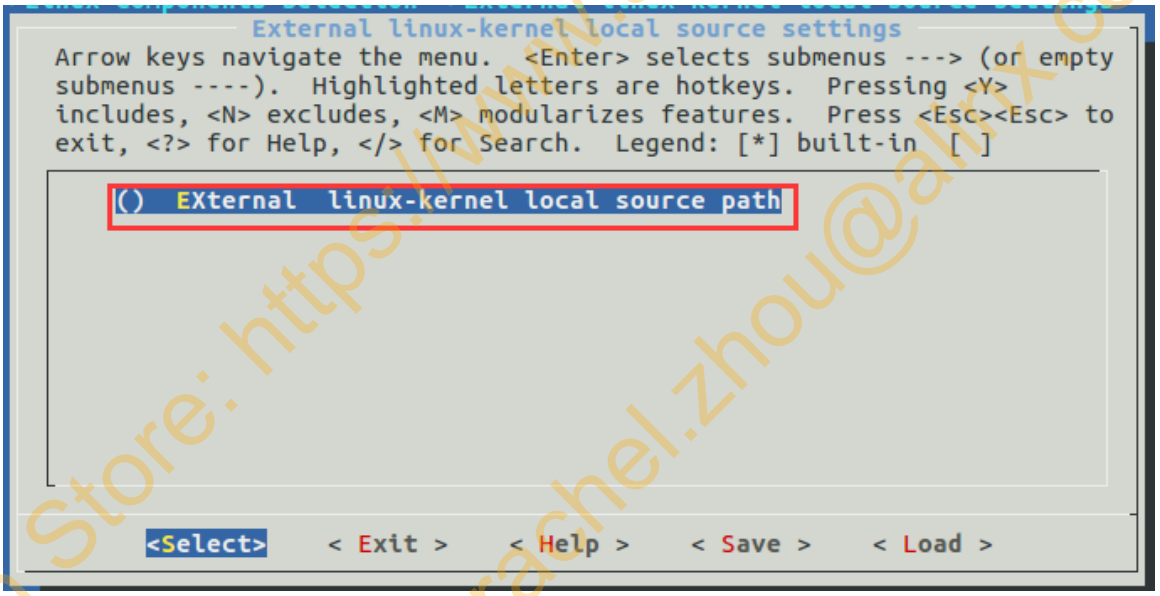
7) Select ext-local-src and press the space button



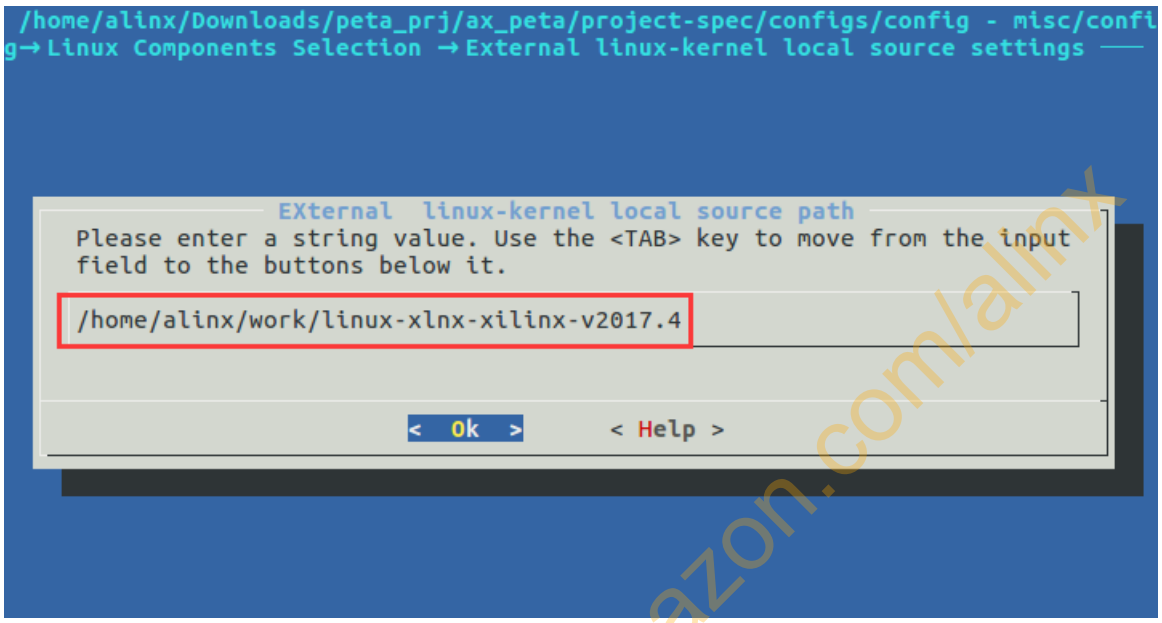
8) Select “External linux-kernel local source settings --->”



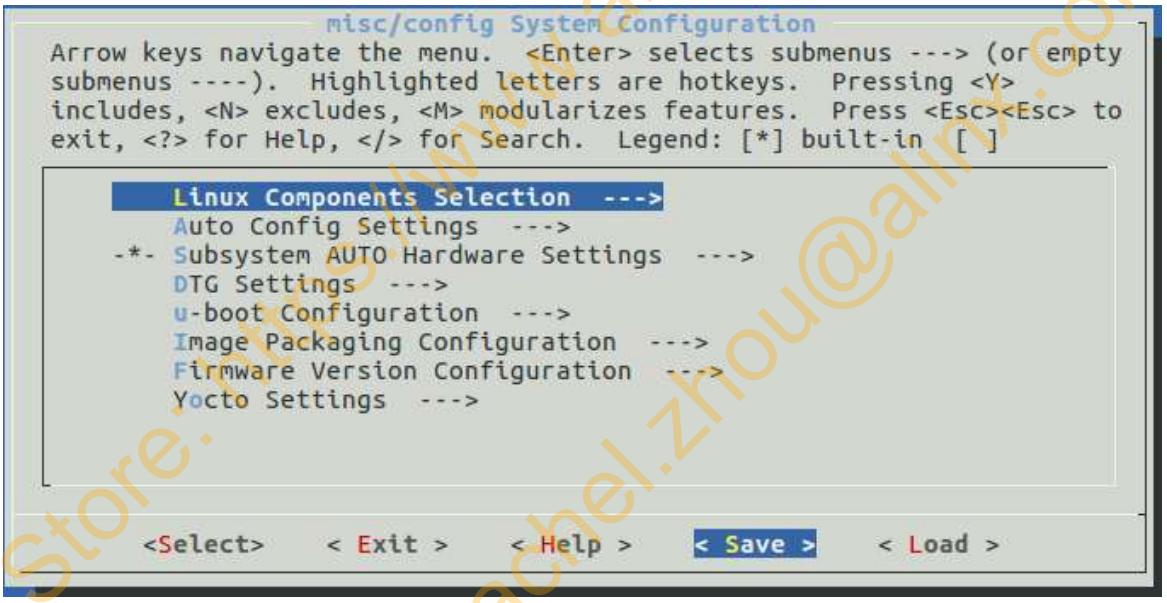
Select “External linux-kernel local source path”



9) Fill in the path to the Linux kernel source “/home/alinx/work/linux-xlnx-xilinx-v2017.4”. The actual path depends on the location of the kernel. Here is just an example.



10)Then save and exit



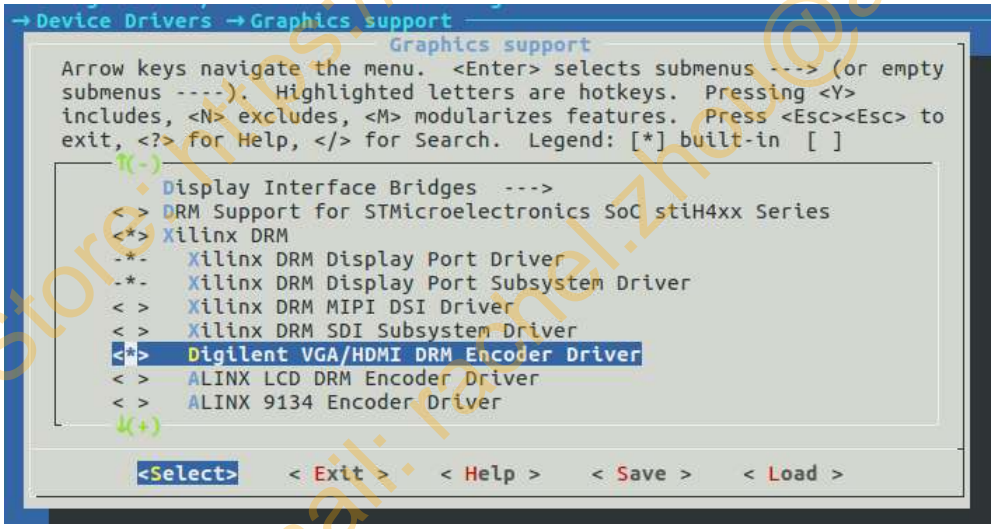
Part 21.2: Configuring the Linux kernel

1) Run the following command to configure the kernel

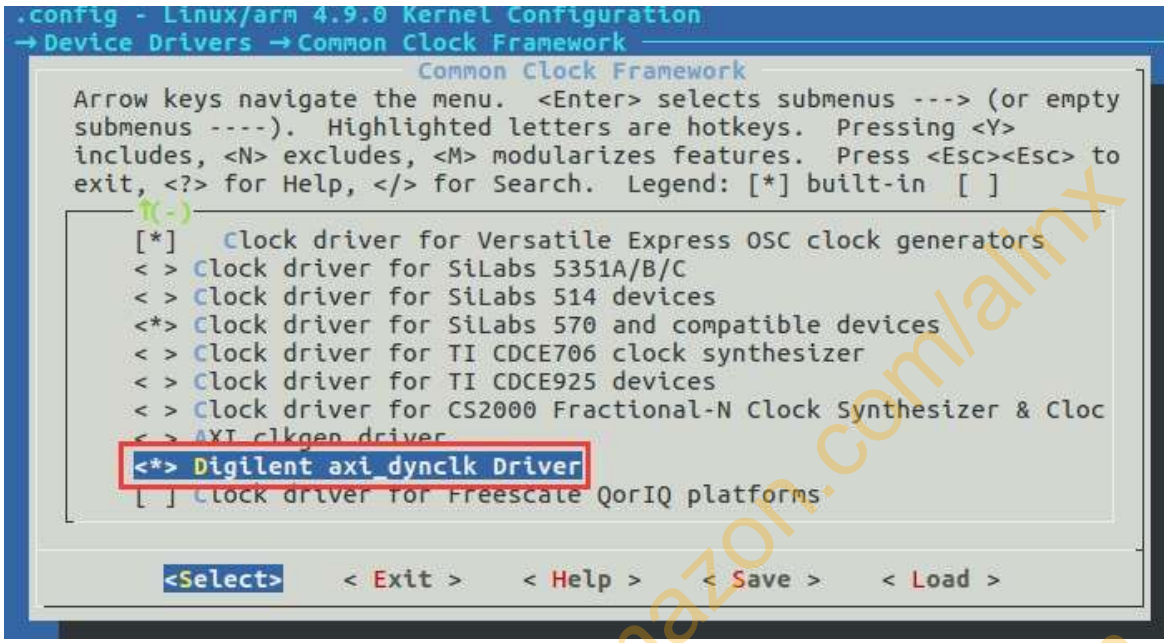
```
petalinux-config -c kernel
```

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
*** Execute 'make' to start the build or try 'make help'.

[INFO] sourcing bitbake
[INFO] generating plnxtool conf
[INFO] generating meta-plnx-generated layer
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_p
eta
~/Downloads/peta_prj/ax_peta
[INFO] generating machine configuration
[INFO] generating bbappends for project . This may take time !
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_p
eta
~/Downloads/peta_prj/ax_peta
[INFO] generating u-boot configuration files

[INFO] generating kernel configuration files
[INFO] generating kconfig for Rootfs
Generate rootfs kconfig
[INFO] oldconfig rootfs
[INFO] generating petalinux-user-image.bb
[INFO] successfully configured project
webtalk failed:PetaLinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-config  -c  kernel
```

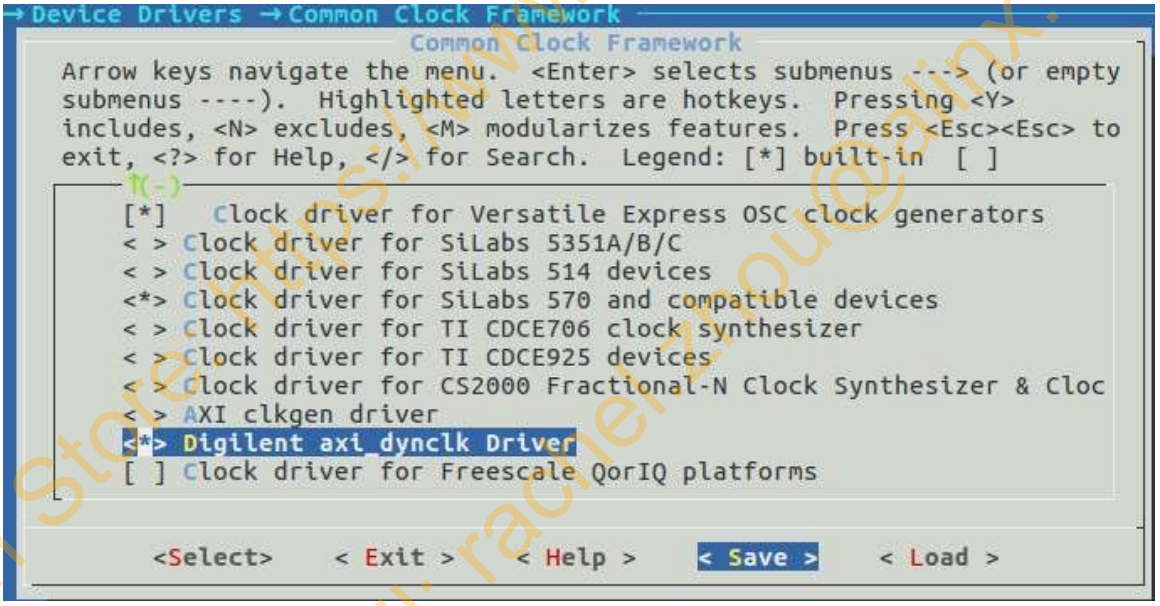
2) In the pop-up window, go to Device Drivers → Graphics support, select the “Digilent VGA/HDMI DRM Encoder Driver” item and press y.



3) Select “Digilent axi_dynclk Driver” in the option of “Device Drivers→ Common Clock Frame work” and press y



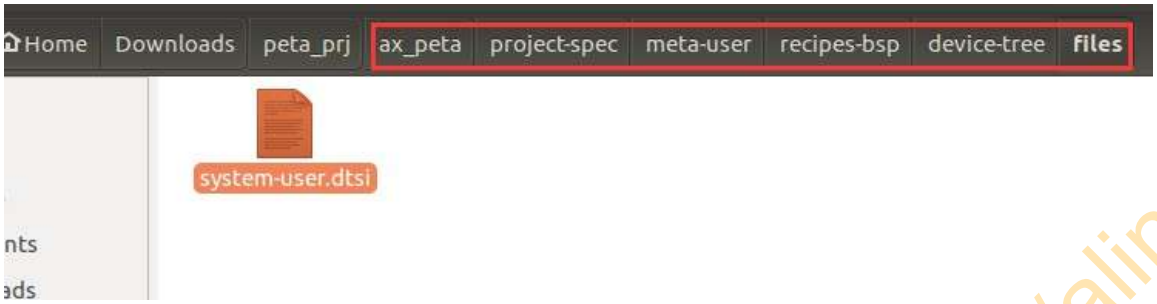
4) Save and exit



Part 21.3: Modify the device tree

The device tree is a kind of formatted text describing the device information. The device tree nodes required by each driver are also different. For developers who have not touched the device, they need to be familiar with them and use them slowly.

1) Open the file named system-user.dtsi in the petalinux project file.



2) Modify the device tree content as follows

```
/include/ "system-conf.dtsi"

/ {
      model = "Zynq ALINX Development Board";
    compatible = "alinx,zynq ", "xlnx,zynq-7000";
    aliases {
              ethernet0 = "&gem0";
               ethernet1 = "&axi_ethernet_0";
               serial0 = "&uart1";
  };

usb_phy0: usb_phy@0 {
          compatible = "ulpi-phy";
            #phy-cells = <0>;
          reg = <0xe0002000 0x1000>;
          view-port = <0x0170>;
          drv-vbus;
     };


};

&i2c0 {
        clock-frequency = <100000>;
 };
&usb0 {   usb-phy = <&usb_phy0>;
 };
&sdhci0 {
u-boot,dm-pre-reloc;
 };
```

```
  &uart1 {
  u-boot,dm-pre-reloc;
  };

  &flash0 {
  compatible = "micron,m25p80", "w25q256", "spi-flash";
   };
  &gem0 {
  phy-handle = <&ethernet_phy>;
    ethernet_phy: ethernet-phy@1 {
  reg = <1>;
  device_type = "ethernet-phy";
   };
};

  &axi_ethernet_0 {
  local-mac-address = [00 0a 35 00 01 22];
  phy-handle = <&phy1>;
  xlnx,has-mdio = <0x1>;
  phy-mode = "rgmii";
  mdio {
            phy1: phy@1 {
          device_type = "ethernet-phy";
               reg = <1>;
                  };
           };
 };

  &amba_pl {

   ax_9134_encoder_0:ax_9134_encoder {
  compatible = "ax_9134,drm-encoder";
        ax_9134,edid-i2c = <&i2c0>;
        ax_9134,reset-gpios = <&hdmi_rst 0 0>;
       };

  xilinx_drm {
  compatible = "xlnx,drm";
    xlnx,vtc = <&v_tc_0>;
```

```
    xlnx,connector-type = "HDMIA";
    xlnx,encoder-slave = <&ax_9134_encoder_0>;
    clocks = <&axi_dynclk_0>;
    planes {
            xlnx,pixel-format = "rgb888";
                plane0 {
                  dmas = <&axi_vdma_0 0>;
                 dma-names = "dma";
                   };
              };
          };
 };
&axi_dynclk_0 {
        compatible = "digilent,axi-dynclk";
         #clock-cells = <0>;
        clocks = <&clkc 15>;
         };
        &v_tc_0 {
        compatible = "xlnx,v-tc-5.01.a";
         };
```

Part 21.4: Compile and test Petalinux project

1) Use the following command to configure uboot, kernel, root file system, device tree etc.

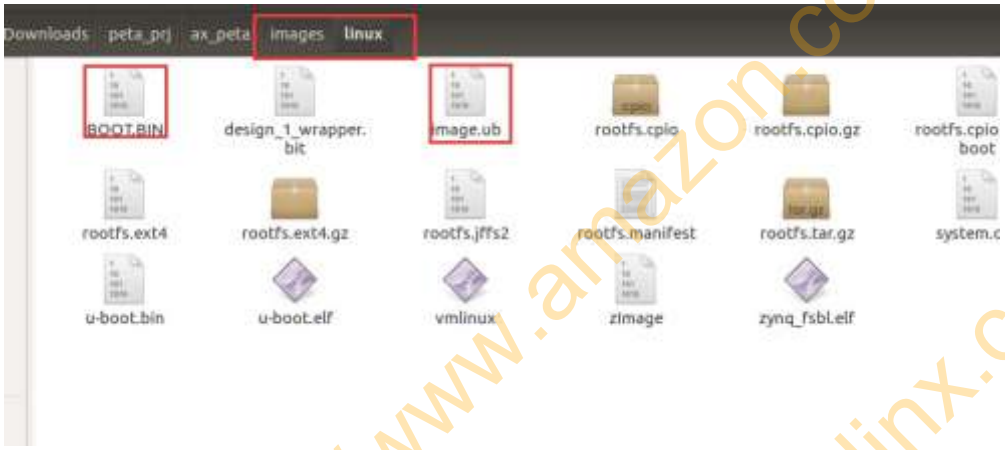
```
petalinux-build
```

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
Loading cache: 100% |############################################| Time: 0:00:00
Loaded 3256 entries from dependency cache.
Parsing recipes: 100% |##########################################| Time: 0:00:03
Parsing of 2466 .bb files complete (2433 cached, 33 parsed). 3259 targets, 226 s
kipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#######################################| Time: 0:00:04
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 2 tasks of which 0 didn't need to be rerun and al
l succeeded.
[INFO] successfully configured kernel
webtalk failed:PetaLinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-build
[INFO] building project
[INFO] sourcing bitbake
INFO: bitbake petalinux-user-image
Loading cache: 100% |############################################| Time: 0:00:01
Loaded 3256 entries from dependency cache.
Parsing recipes: 100% |##########################################| Time: 0:00:03
Parsing of 2466 .bb files complete (2433 cached, 33 parsed). 3259 targets, 226 s
kipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks:  18% |#######                                 | ETA:  0:00:08
```

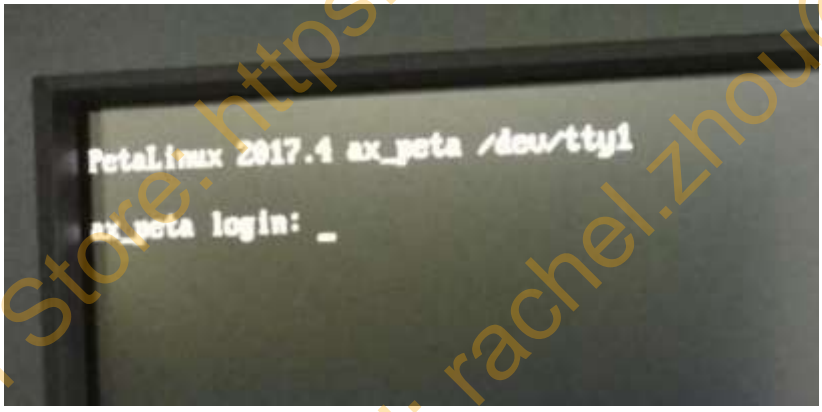
2) Run the following command to generate a BOOT file, paying attention to spaces and short lines.

```
petalinux-package--boot--fsbl./images/linux/zynq_fsbl.elf  --fpga  --u-boot--force
```

3) Copy “BOOT.bin” and “iamge.ub” to sd, set the FPGA development board to sd mode, plug in the HDMI display, and start the FPGA development board



4) The display will dispaly the following



Part 21.5: Q&A

Part 21.5.1: How to prevent system sleep

Run command before hibernation

```
echo  -e  " \033[9;0]\033[?33l\033[?25h\033[?1c"  >  /dev/tty0
echo  -e  " \033[9;0]\033[?33l\033[?25h\033[?1c"  >  /dev/tty1
echo  -e  " \033[9;0]\033[?33l\033[?25h\033[?1c"  >  /dev/tty
echo  -e  " \033[9;0]\033[?33l\033[?25h\033[?1c"  >  /dev/console
```

Part 22: Use the Debian desktop system

In the previous tutorial, we learned how to use Petalinux to customize an embedded Linux system and HDMI display. This experiment will be a root file system based on Debian. The root file system is more complicated. This chapter is no longer explained, directly using the created Debian root file system.

Part 22.1: Petalinux configuration

Since Debian root file system is relatively large, the QSPIflash capacity is not enough,can only be placed in sd card or emmc, so we have to configure Petalinux.

This experiment is still a modification of the Petalinux project in the previous experiment, you need to master the previous experimental content.

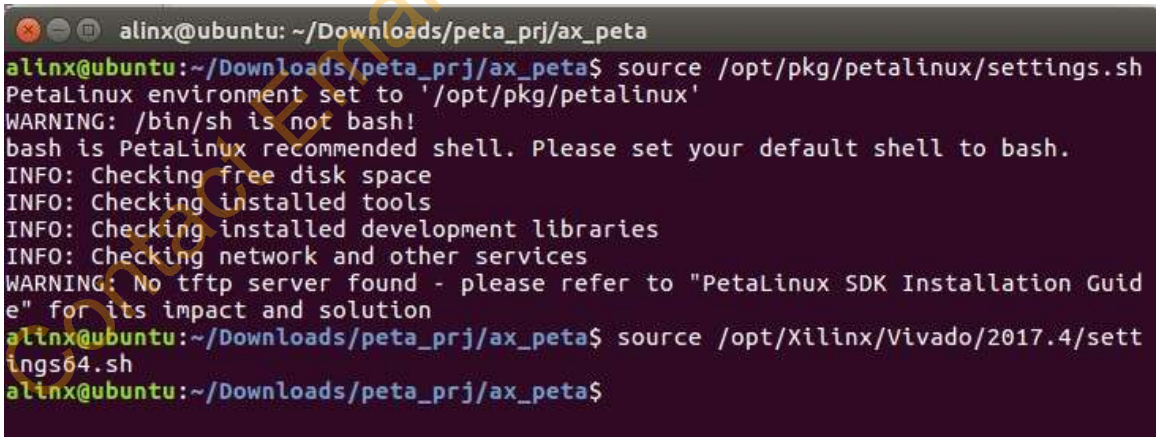
1) Open the terminal and enter the Petalinux project directory in the previous experiment.

2) Set the petalinux environment variable, run the following command

```
source /opt/pkg/petalinux/settings.sh
```

3) Run the following command to set the vivado environment variable

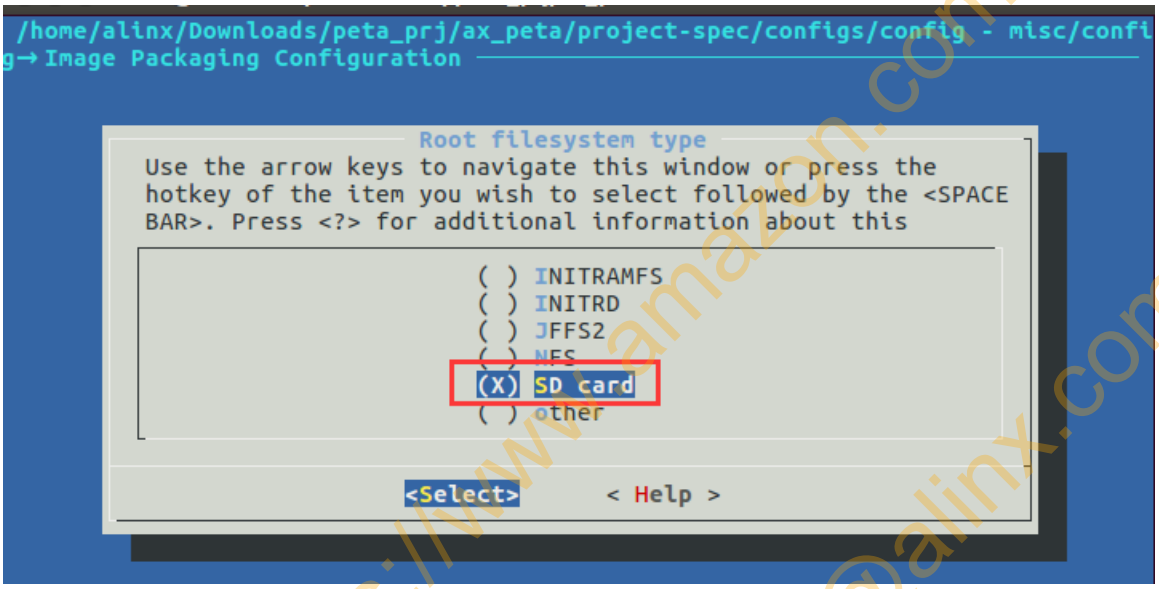
```
source /opt/Xilinx/Vivado/2017.4/settings64.sh
```



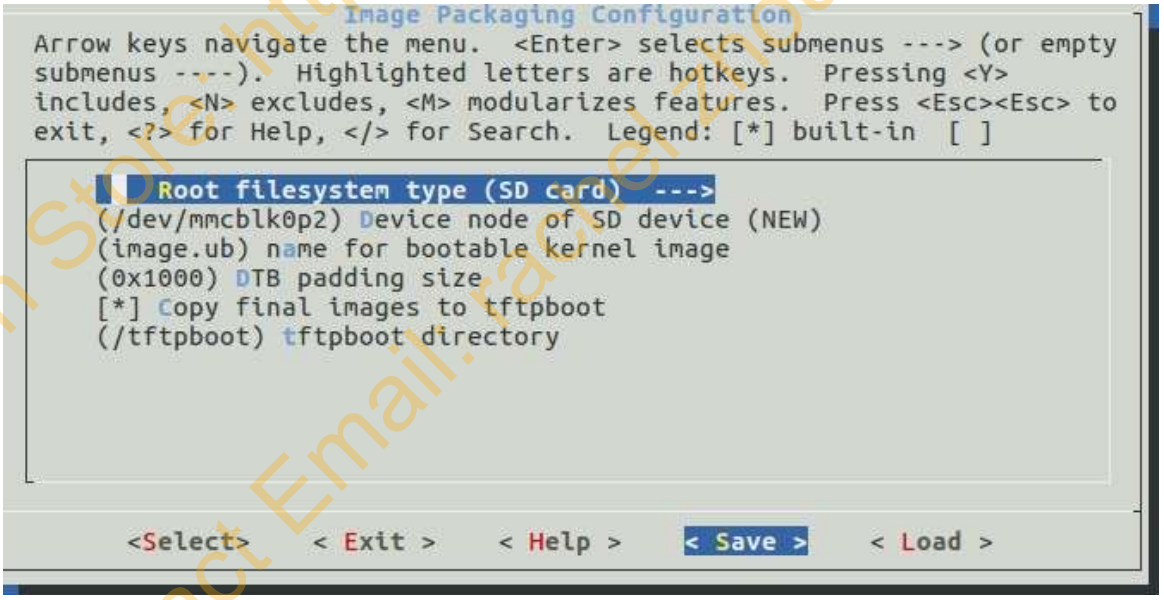
4) Use the following command to reconfigure petalinux

```
petalinux-config
```

5) Select SDcard in the ImagePackagingConfiguration-->Rootfilesystemtype option and put the root file system on the SD card.



6) Save and exit



Part 22.2: Configuring the Linux kernel

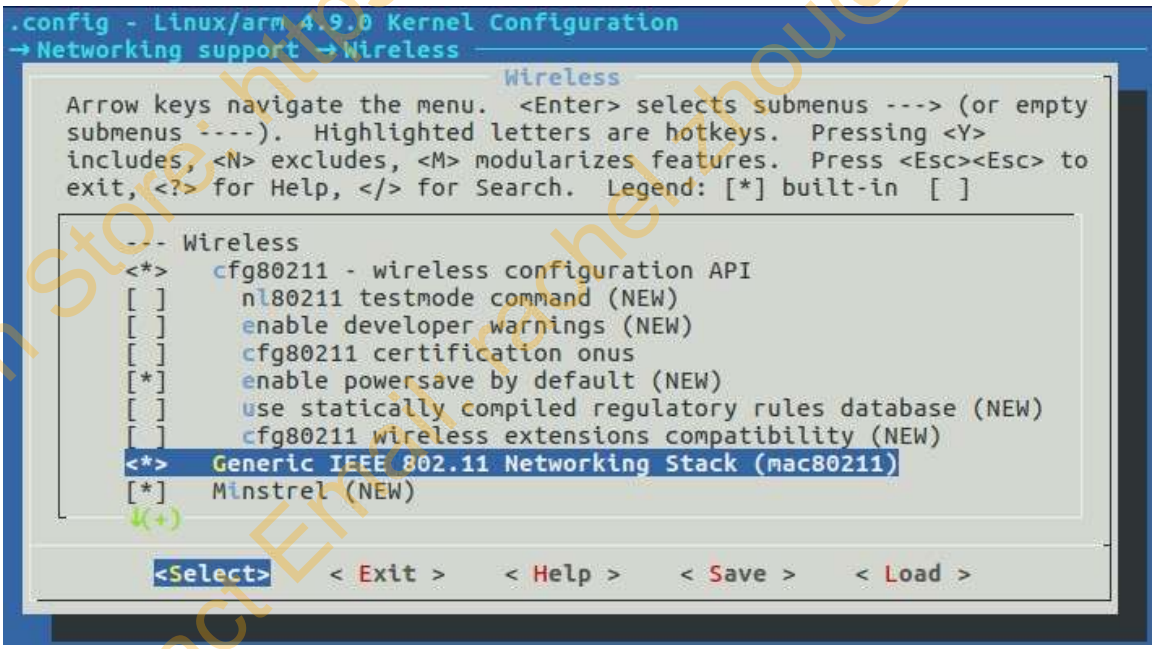
1) Run the following command to configure the kernel

```
petalinux-config -c kernel
```

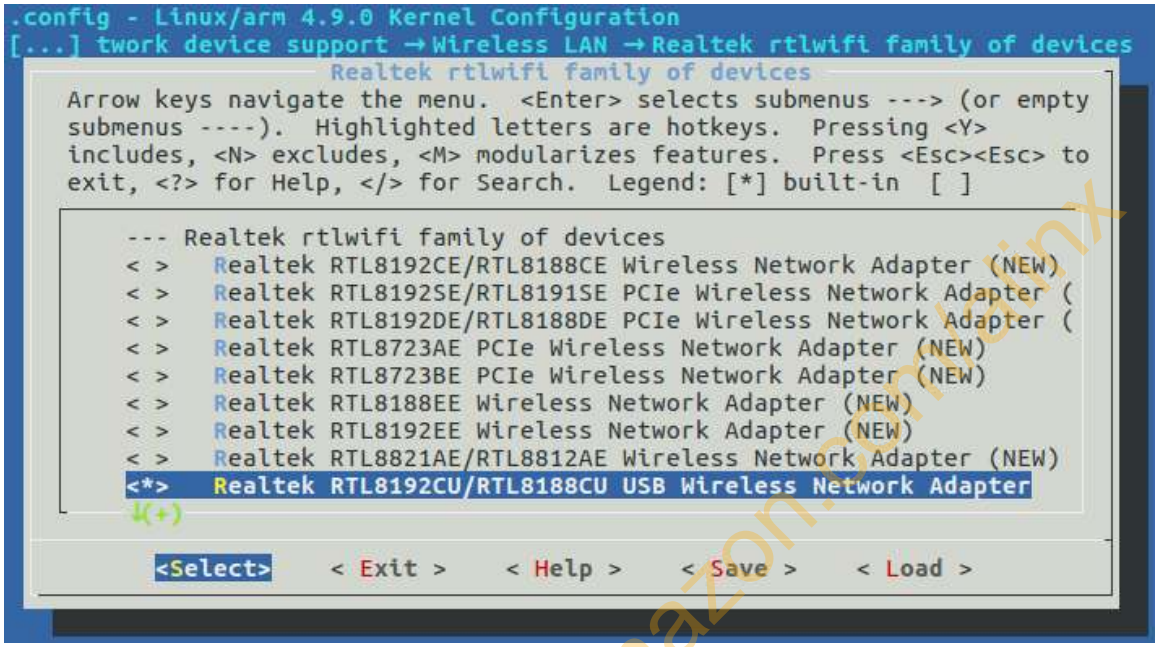
```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
*** Execute 'make' to start the build or try 'make help'.

[INFO] sourcing bitbake
[INFO] generating plnxtool conf
[INFO] generating meta-plnx-generated layer
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_p
eta
~/Downloads/peta_prj/ax_peta
[INFO] generating machine configuration
[INFO] generating bbappends for project . This may take time !
~/Downloads/peta_prj/ax_peta/build/misc/plnx-generated ~/Downloads/peta_prj/ax_p
eta
~/Downloads/peta_prj/ax_peta
[INFO] generating u-boot configuration files

[INFO] generating kernel configuration files
[INFO] generating kconfig for Rootfs
Generate rootfs kconfig
[INFO] oldconfig rootfs
[INFO] generating petalinux-user-image.bb
[INFO] successfully configured project
webtalk failed:PetaLinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-config  -c  kernel
```

Part 22.2.1: Configure USBWIFI module driver

1) Select cfg80211 - wireless configuration API in the option Networking Support -> Wireless -> Options and then select Generic IEEE 802.11 Networking Stack (mac80211)

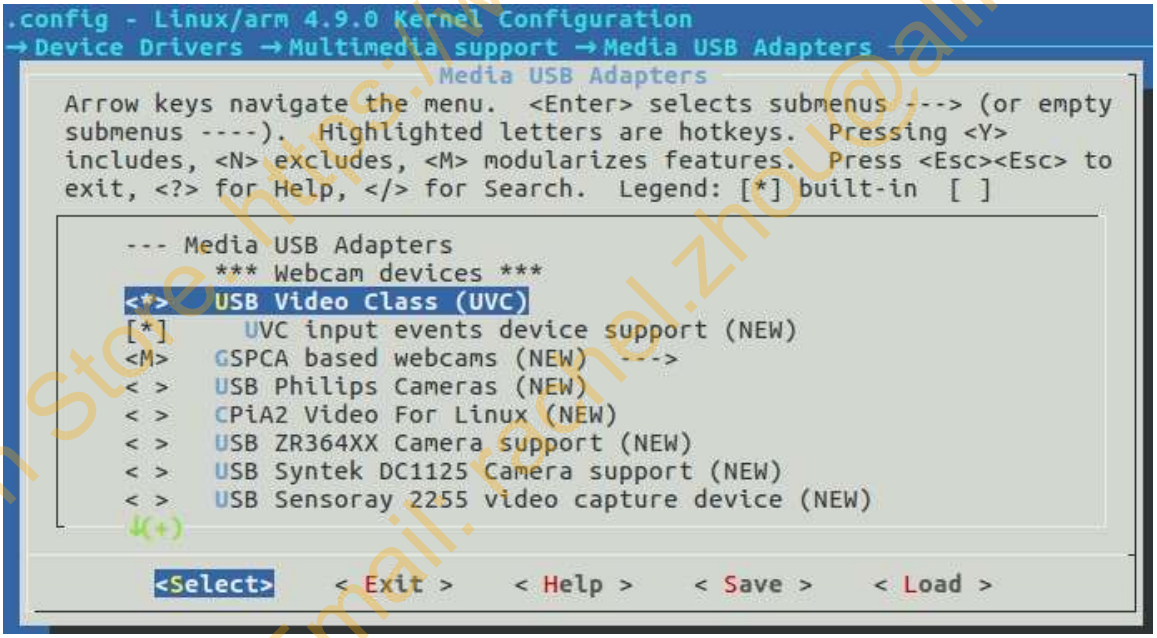


2) In the option Device Drivers -> Network device support -> Wireless LAN -> Realtek rtlwifi family of devices select Realtek RTL8192CU / RTL8188CU USB Wireless Network Adapter



Part 22.2.2: Configuring a USB camera driver

1) Enable USB Video Class (UVC) in the option Device Drivers ---> Multimedia support ---> Media USB Adapters --->



2) Save and exit

Part 22.3: Compile and test Petalinux project

1) Use the following command to configure uboot, kernel, root file system, device tree etc.

```
petalinux-build
```

```
alinx@ubuntu: ~/Downloads/peta_prj/ax_peta
Loading cache: 100% |#########################################| Time: 0:00:00
Loaded 3256 entries from dependency cache.
Parsing recipes: 100% |#######################################| Time: 0:00:03
Parsing of 2466 .bb files complete (2433 cached, 33 parsed). 3259 targets, 226 s
kipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |####################################| Time: 0:00:04
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 2 tasks of which 0 didn't need to be rerun and al
l succeeded.
[INFO] successfully configured kernel
webtalk failed:PetaLinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-build
[INFO] building project
[INFO] sourcing bitbake
INFO: bitbake petalinux-user-image
Loading cache: 100% |#########################################| Time: 0:00:01
Loaded 3256 entries from dependency cache.
Parsing recipes: 100% |#######################################| Time: 0:00:03
Parsing of 2466 .bb files complete (2433 cached, 33 parsed). 3259 targets, 226 s
kipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks:  18% |#######                              | ETA:  0:00:08
```

2) Run the following command to generate the BOOT.BIN file, pay attention to spaces and short lines

```
petalinux-package --boot --fsbl ./images/linux/zynq_fsbl.elf --fpga --u-boot --force
```

Part 22.4: Create SD card file system

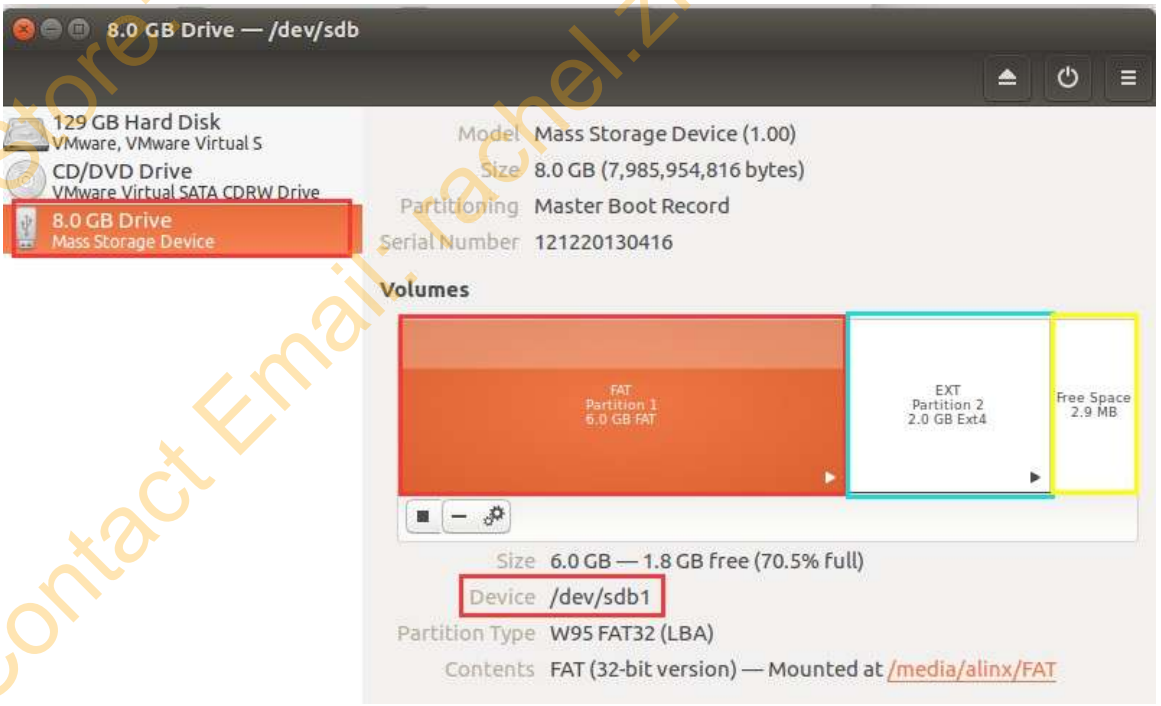
Creating an SD card file system will result in the loss of content on the SD card. Please make a copy first.

Part 22.4.1: Modify SD card partition

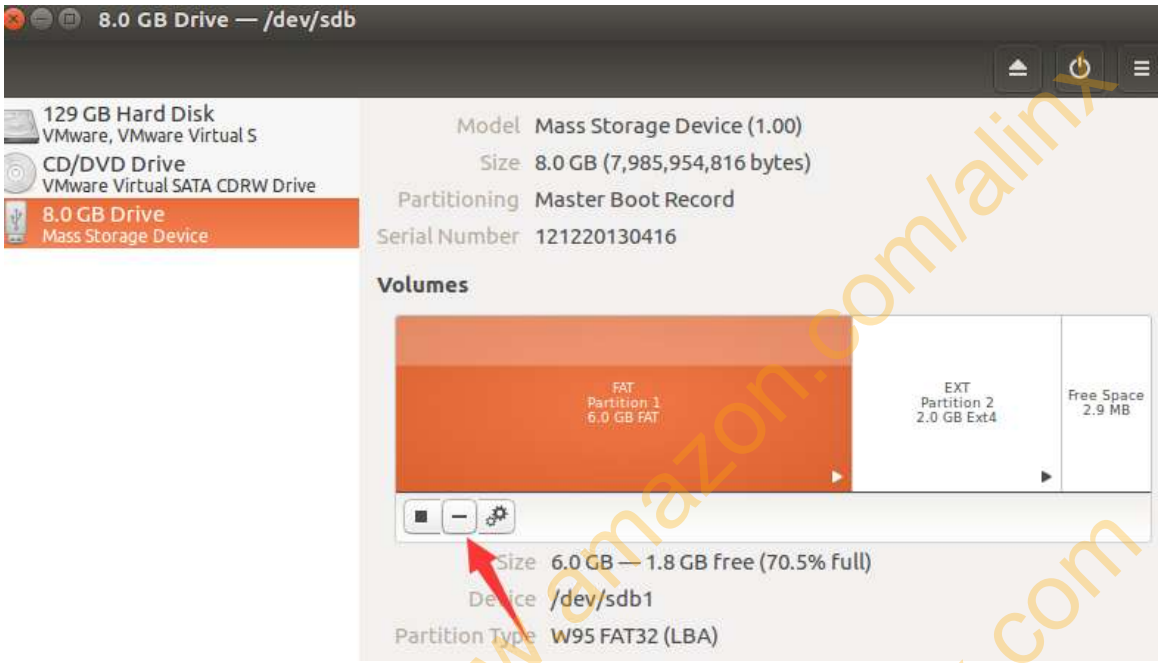
1) Insert the sd card of the FPGA development board into the card reader and plug it into the USB port of the computer.
2) Connect to a virtual machine in Linux

3) In the search path of ubuntu, enter disk and the icon of Disks will appear.

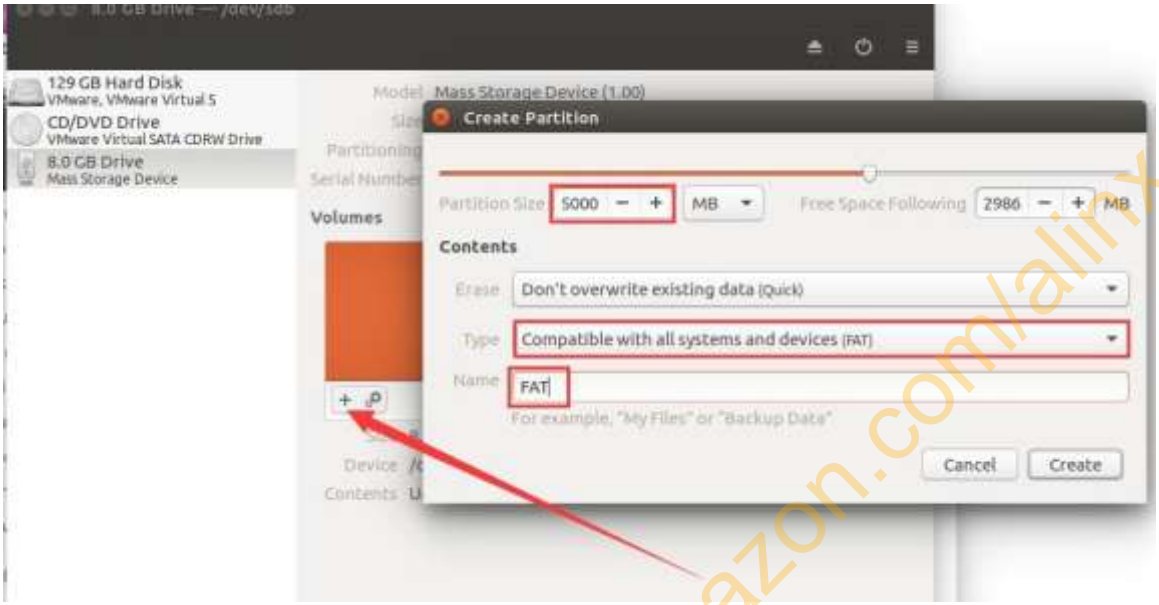
4) Click the Disks icon and the "Disks" dialog box appears. In this experiment, the SD card has been divided into 2 partitions, one named FAT and one named EXT, which is to be repartitioned here.



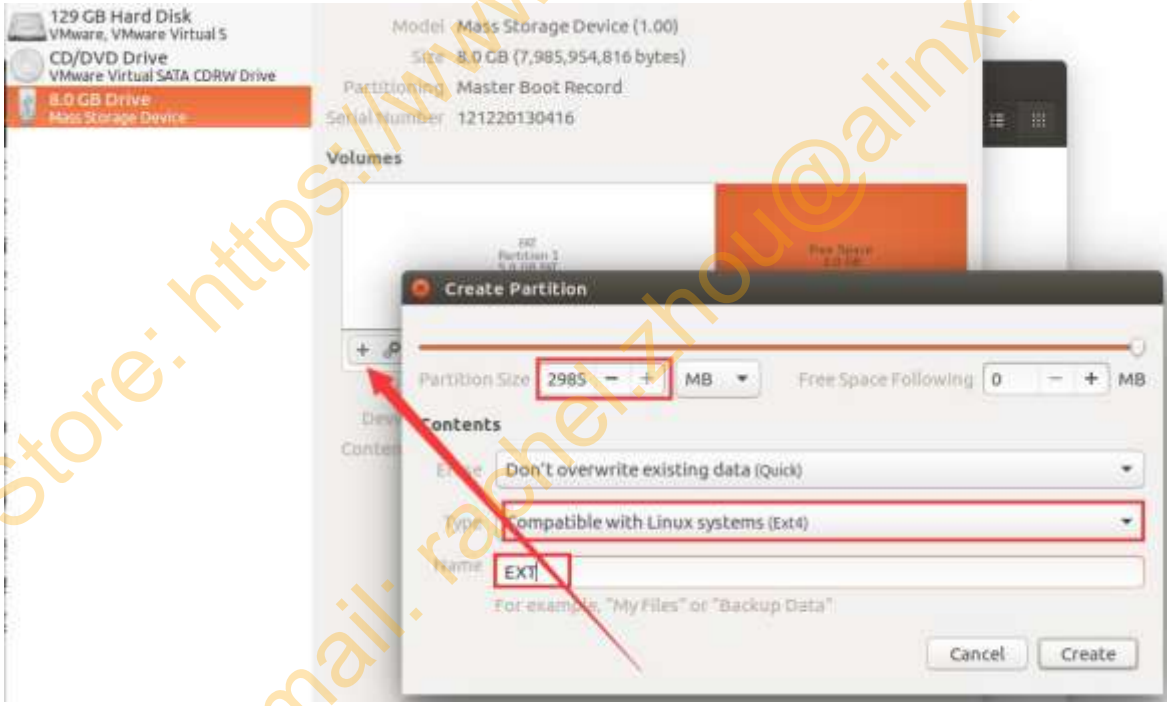
5) Select the delete partition icon under each partition to delete all partitions.



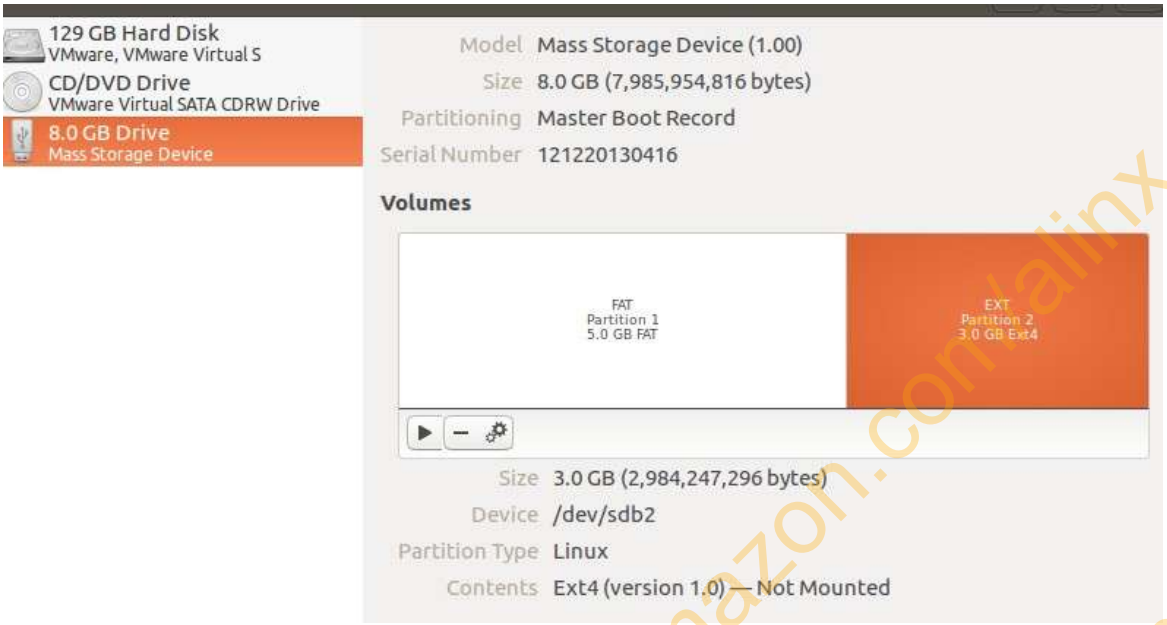
6) Click the icon to add a partition, add the first partition, this experiment fills in 5000MB, the format is FAT, used for storageZYNQ startup file BOOT.bin and kernel file, device tree, named FAT



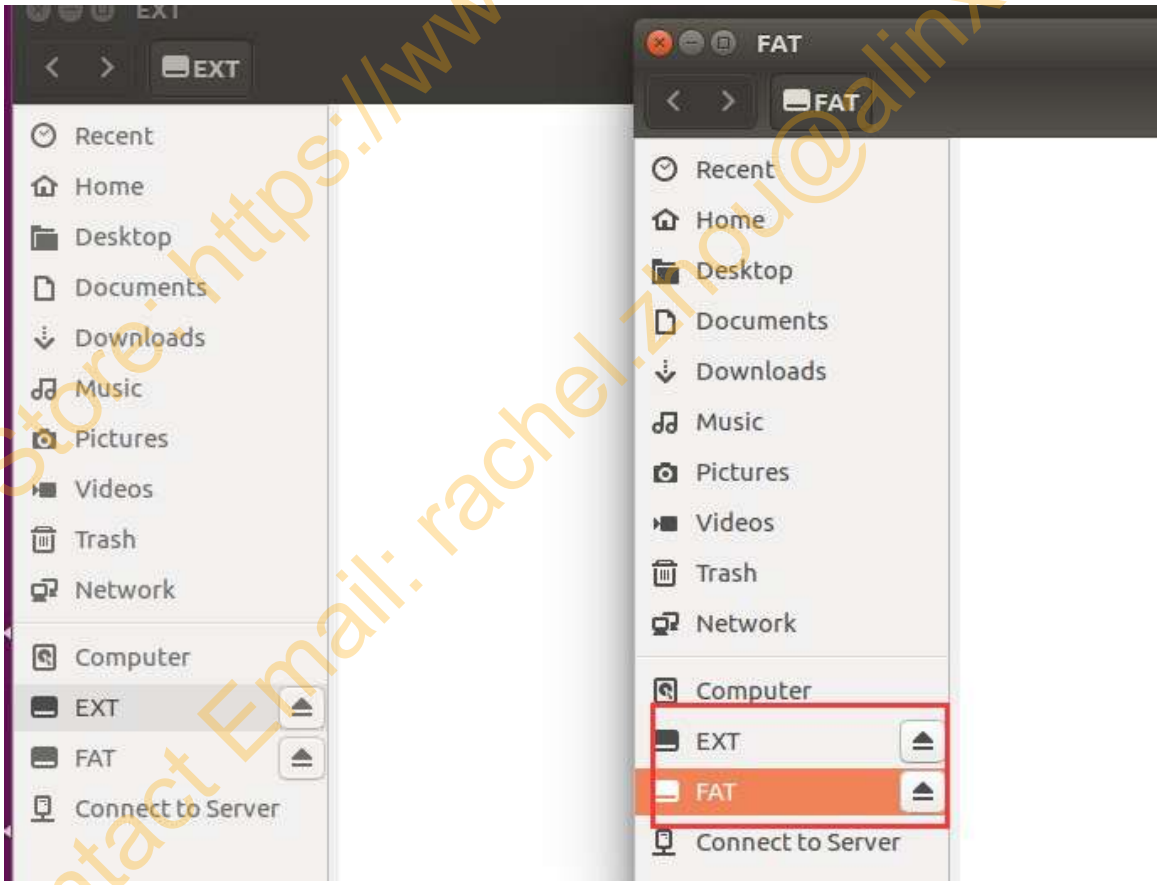
7) Create a second partition for the root file system in the format EXT4 with the name EXT



8) Re-insert the sd card after creating the partition



9) The system will automatically mount the partition and pop up the window.

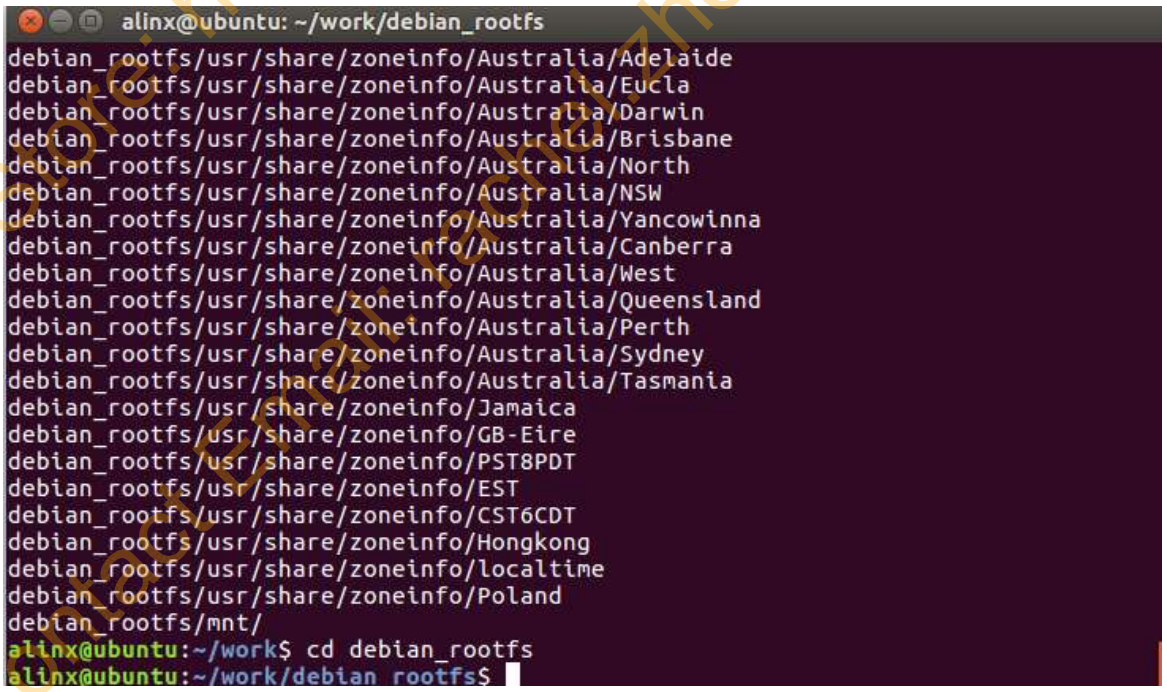


Part 22.4.2: Synchronize the root file system to the SD card EXT4 partition

1) Copy the compressed file of the root file system to the Linux host (experimental copy in the "/home/alinx/work" directory)

2) Unzip the file system by entering the unzip command "sudo tar zxvpf debian_rootfs.tar.gz". Decompression may take a few minutes.

3) Copy all the files of the file system to the root directory of the EXT partition of the SD card. Enter the command "cddebian_rootfs" in the command window to enter the directory of the root file system.

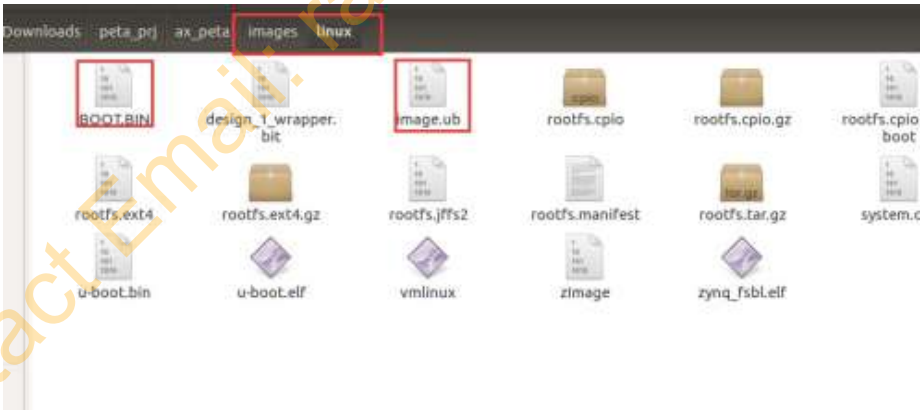


4) In the command window, type "sudorsync -av .//media/alinx/EXT" (/media/alinx/EXT is the path of the SD card EXT4 partition, which may be different, please modify it according to your actual situation), start synchronizing the current directory to The SD card's EXT partition root directory, synchronization may take ten minutes.

```
var/log/dpkg.log
var/log/faillog
var/log/fontconfig.log
var/log/lastlog
var/log/wtmp
var/log/apt/
var/log/apt/history.log
var/log/apt/term.log
var/log/fsck/
var/log/fsck/checkfs
var/log/fsck/checkroot
var/log/ntpstats/
var/mail/
var/opt/
var/spool/
var/spool/mail -> ../mail
var/spool/cron/
var/spool/cron/crontabs/
var/spool/rsyslog/
var/tmp/

sent 1,117,338,469 bytes  received 809,451 bytes  5,414,759.90 bytes/sec
total size is 1,114,115,109  speedup is 1.00
alinx@ubuntu:~/work/debian_rootfs$ sudo rsync -av ./ /media/alinx/EXT
```

5) When the command prompt reappears on the command line, the process that indicates synchronization ends

6) Copy BOOT.bin and iamge.ub to the FAT32 partition (first partition) of sd, set the FPGA development board to start in sd mode, plug in the HDMI display, and start the FPGA development board.



7) After the SD card is created, insert the finished SD card into the SD card slot of the FPGA development board. Connect the USB serial cable to the HDMI display. After the board is powered on, the interface of the Debian operating system will be displayed on the HDMI display. In addition, in the serial port tool, we can see the process of operating system startup. After running u-boot, start running Linux. Account: root, password: root

8) After entering the system, use the ifconfig command to view the network connection.

```
root@zynq:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0a:35:00:1e:53
          inet addr:192.168.1.46  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20a:35ff:fe00:1e53/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:119 errors:0 dropped:0 overruns:0 frame:0
          TX packets:86 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12231 (11.9 KiB)  TX bytes:7420 (7.2 KiB)
          Interrupt:29 Base address:0xb000

eth1      Link encap:Ethernet  HWaddr 00:0a:35:00:03:22
          inet addr:192.168.1.62  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20a:35ff:fe00:322/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:26 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2771 (2.7 KiB)  TX bytes:1184 (1.1 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:236 errors:0 dropped:0 overruns:0 frame:0
          TX packets:236 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:18960 (18.5 KiB)  TX bytes:18960 (18.5 KiB)
```

Part 23: Making QSPIFlash boot Linux

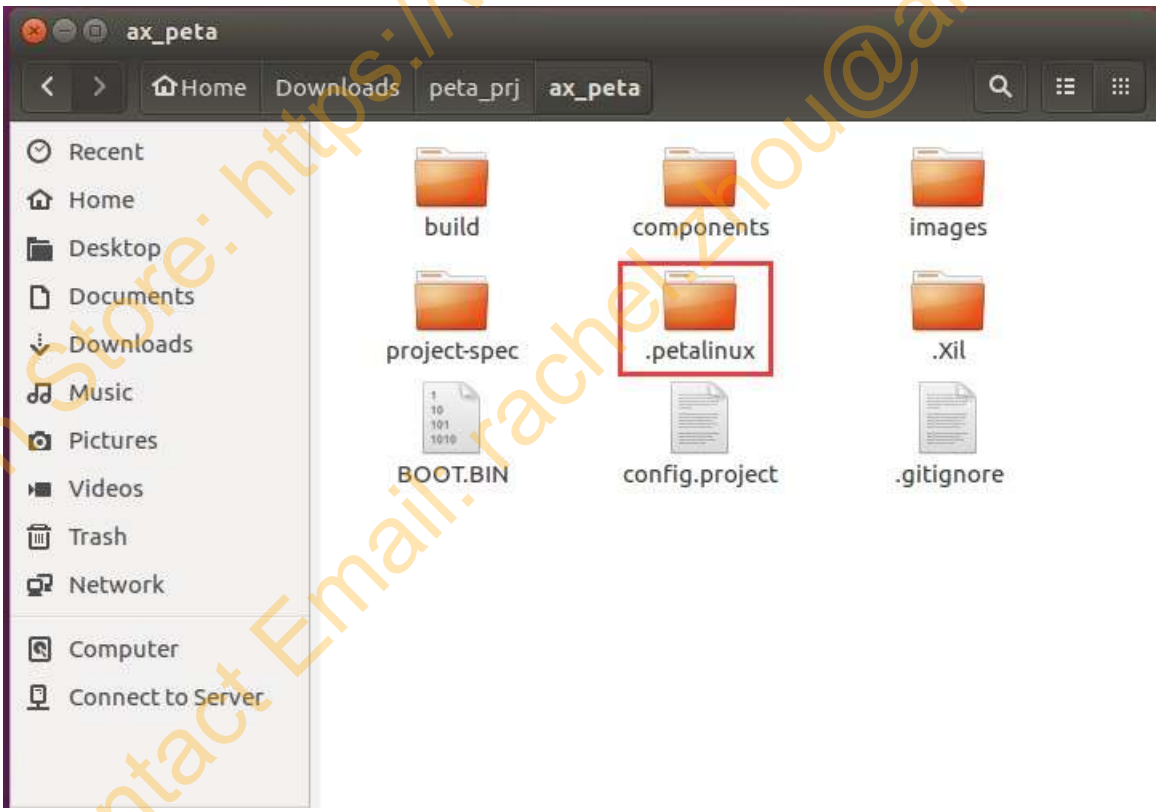
The Linux explained in the previous tutorial is started in SD. This experiment explains how to use Petalinux to make a Linux boot from QSPI Flash.

Here, it is emphasized that although the QSPI Flash size on the board is 32MByte, the ZYNQ chip can only use 16MByte, which is determined by ZYNQ itself. Therefore, the Linux system file cannot be too large, and it cannot be used beyond 16MB.

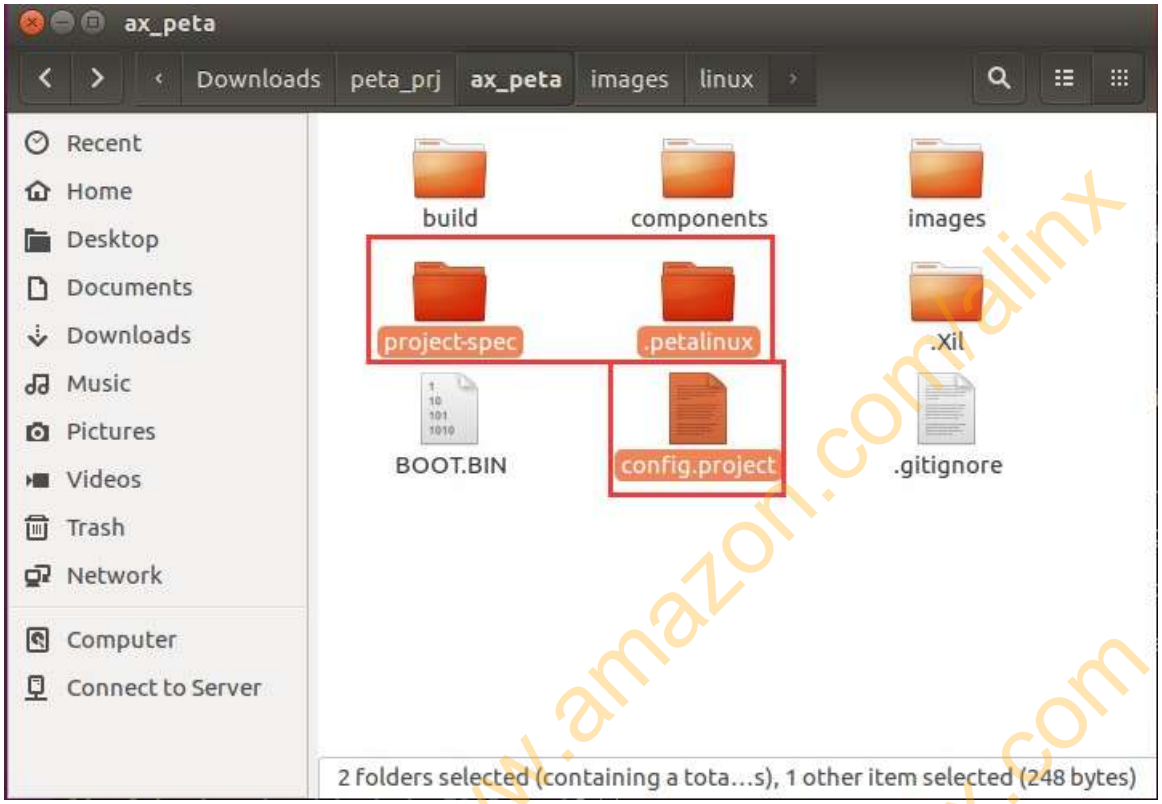
Part 23.1: Copy Petalinux Engineering

In the previous tutorial, we have used Petalinux to do various experiments of SD card startup. We want to keep the SD startup project, but we don't want to create a new project. We can copy the old project.

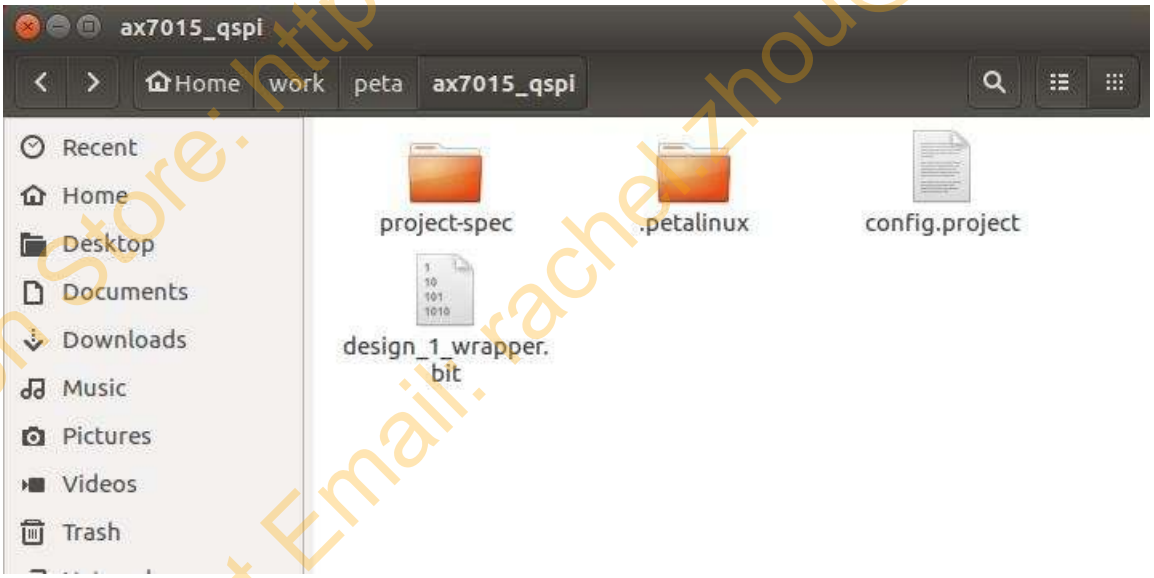
1) Press Ctrl+H in the project directory to display hidden files.



2) Copy project-spec, .petalinux, config.project to a new directory as a new Petalinux project



3) Then copy the bit file in the images/linux directory to the new project directory for synthesizing BOOT with PL configuration.



Part 23.2: Configuration Compilation Petalinux

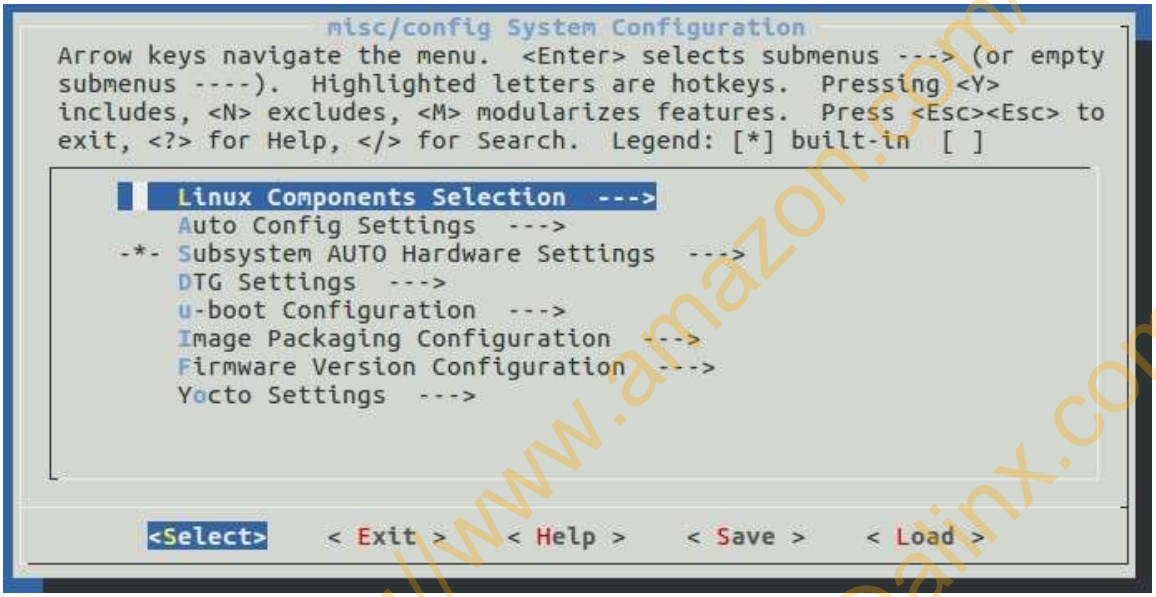
1) Use the following command to set environment variables

```
source /opt/pkg/petalinux/settings.sh
source /opt/Xilinx/Vivado/2017.4/settings64.sh
```

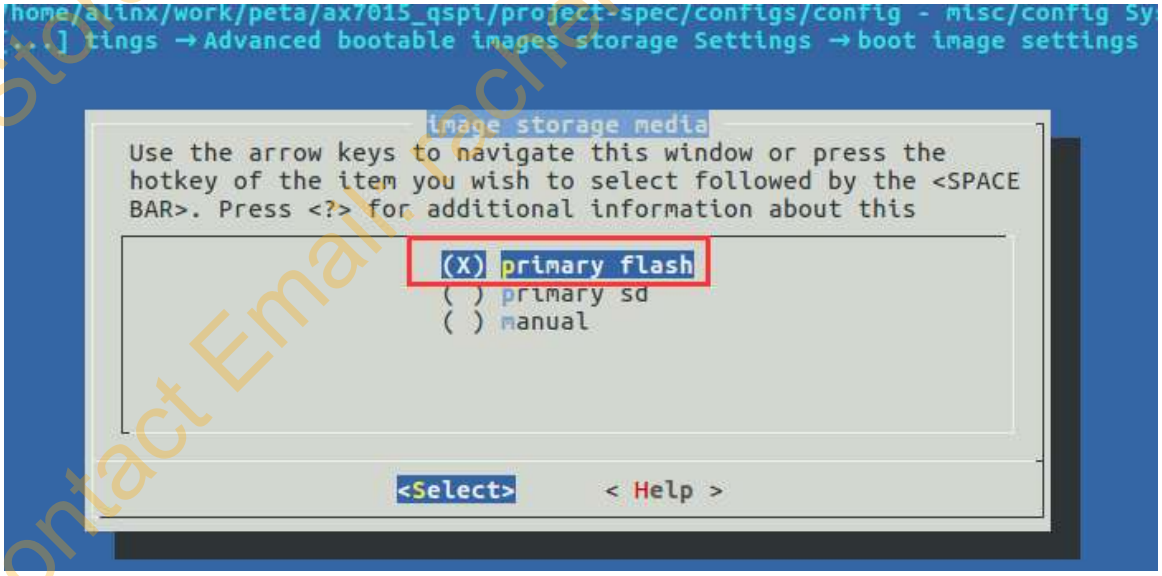
```
alinx@ubuntu:~/work/peta/ax7015_qspi$ source /opt/pkg/petalinux/settings.sh
```

```
alinx@ubuntu:~/work/peta/ax7015_qspi$ source /opt/Xilinx/Vivado/2017.4/settings64.sh
```

2) Configure Petalinux with the petalinux-config command

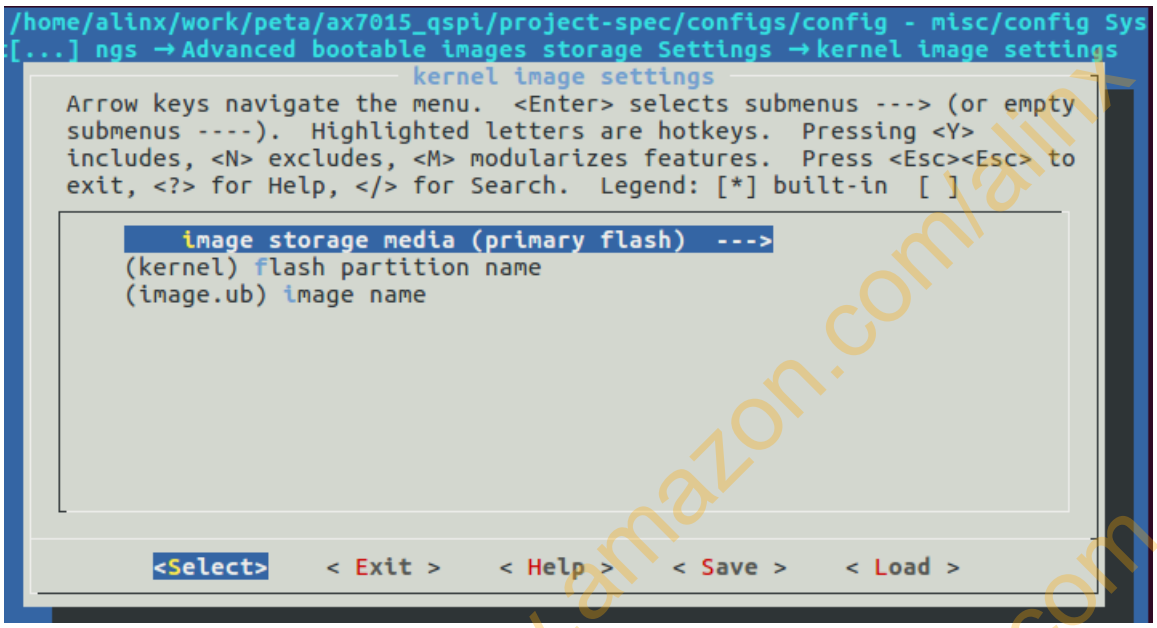


3) Select primaryflash in Subsystem AUTO Hardware Settings--->Advanced bootable images storage Settings--->boot image settings--->image storage media option

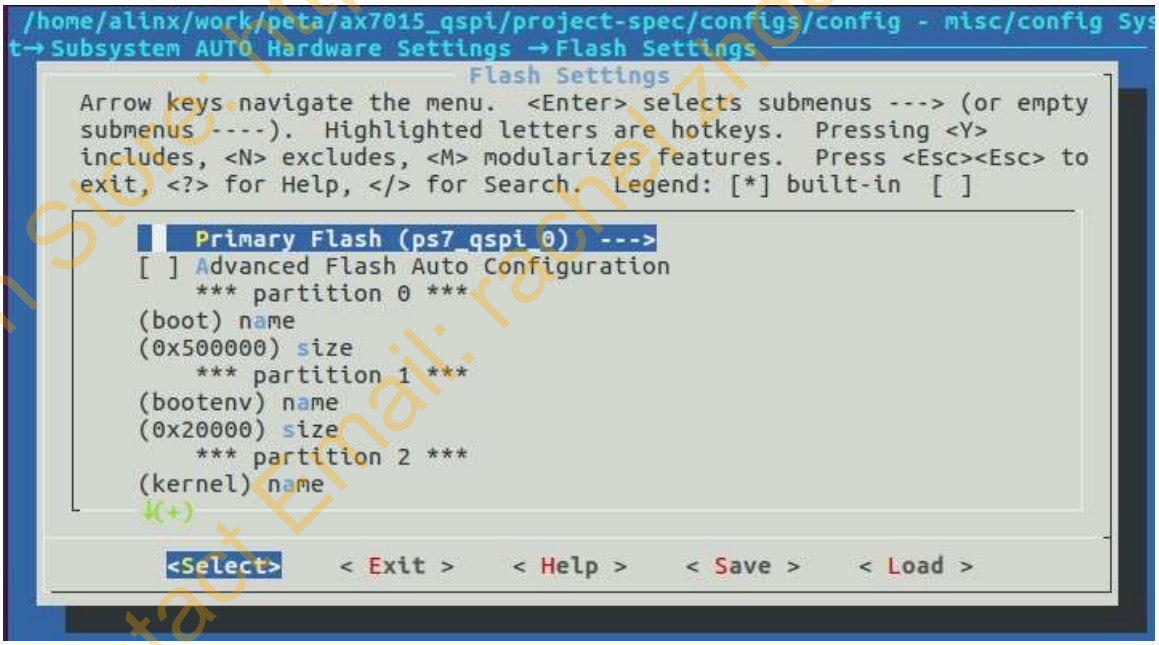


4) Select primary flash in the Subsystem AUTO Hardware Settings --->

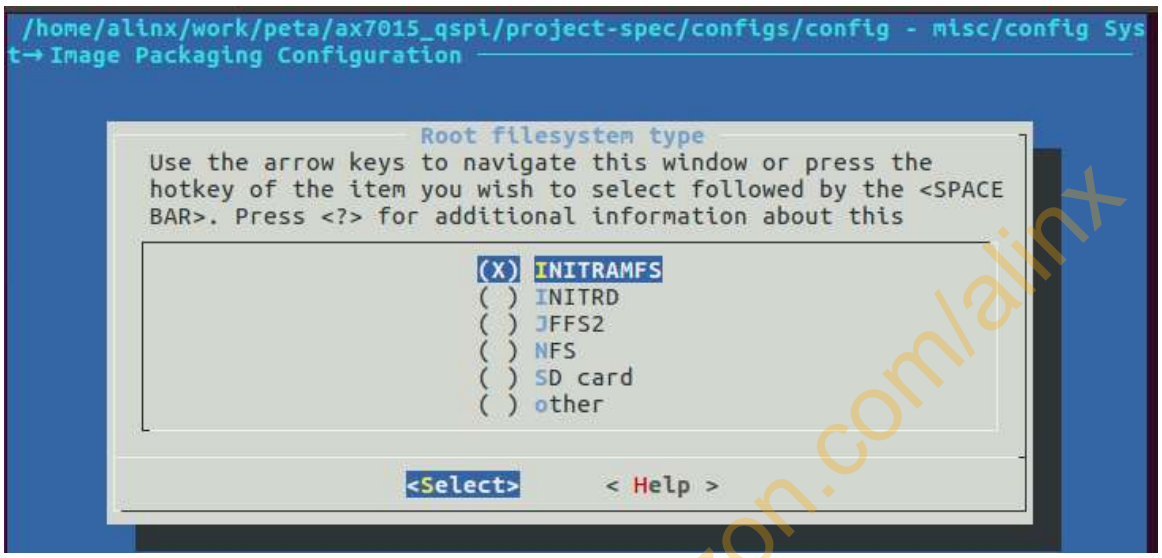
Advanced bootable images storage Settings ---> kernel image settings ---> image storage media option



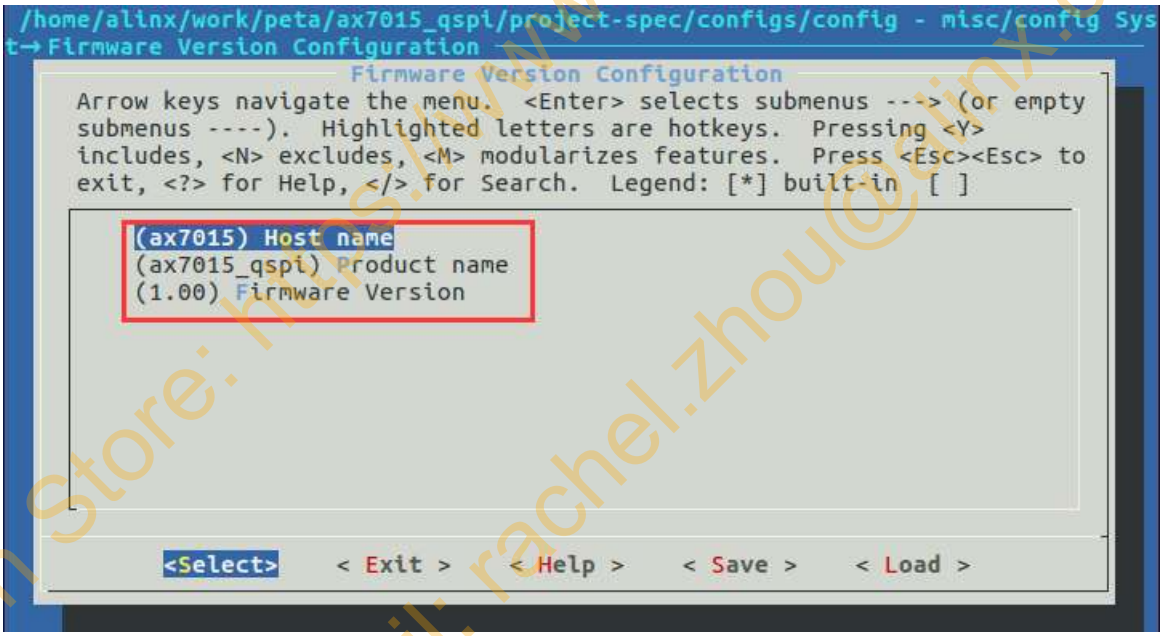
5) Subsystem AUTO Hardware Settings→ Flash Settings can modify the QSPI flash partition. The default is almost ready to use. If your file size exceeds the default partition size, you should adjust it yourself.



6) Select INITRAMFS in Image Packaging Configuration --->Root file system type, use the RAM type root file system, so that you can easily package and write to QSPI Flash.



7) You can modify the information such as Host name in Firmware Version Configuration --->



8) Save configuration

9) Compile

```
*** Execute 'make' to start the build or try 'make help'.

[INFO] sourcing bitbake
[INFO] generating plnxtool conf
[INFO] generating meta-plnx-generated layer
~/work/peta/ax7015_qspi/build/misc/plnx-generated ~/work/peta/ax7015_qspi
~/work/peta/ax7015_qspi
[INFO] generating machine configuration
[INFO] generating bbappends for project . This may take time !
~/work/peta/ax7015_qspi/build/misc/plnx-generated ~/work/peta/ax7015_qspi
~/work/peta/ax7015_qspi
[INFO] generating u-boot configuration files

[INFO] generating kernel configuration files
[INFO] generating kconfig for Rootfs
Generate rootfs kconfig
[INFO] oldconfig rootfs
[INFO] generating petalinux-user-image.bb
[INFO] successfully configured project
webtalk failed:PetaLinux statistics:extra lines detected:notsent_nofile!
webtalk failed:Failed to get PetaLinux usage statistics!
alinx@ubuntu:~/work/peta/ax7015_qspi$ petalinux-build
```

10)Use the following command to synthesize BOOT. The difference from the previous tutorial is to add the --kernel option and package the kernel into the BOOT.bin file.

```
petalinux-package--boot--fsbl./images/linux/zynq_fsbl.elf--fpga        --u-boot --kernel--force
```

```
alinx@ubuntu:~/Downloads/peta_prj/ax_peta$ petalinux-package  -boot --fsbl ./ima
ges/linux/zynq_fsbl.elf  --fpga  --u-boot --kernel  --force

INFO: Getting system flash information...
rlwrap: warning: your $TERM is 'xterm-256color' but rlwrap couldn't find it in t
he terminfo database. Expect some problems.: Inappropriate ioctl for device

INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/zyn
q_fsbl.elf"
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/des
ign_1_wrapper.bit"
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/u-b
oot.elf"
INFO: File in BOOT BIN: "/home/alinx/Downloads/peta_prj/ax_peta/images/linux/ima
ge.ub"
INFO: Generating zynq binary package BOOT.BIN...
INFO: Binary is ready.
WARNING: Unable to access the TFTPBOOT folder /tftpboot!!!
WARNING: Skip file copy to TFTPBOOT folder!!!
webtalk failed:Invalid tool in the statistics file:petalinux-yocto!
webtalk failed:Failed to get PetaLinux usage statistics!
```

11)You can burn BOOT.bin to QSPIFlash by referring to the “Curing Program” section.

12) Adjust the startup mode to QSPI by the DIP switch