ZYNQ MPSoC Development Platform

FPGA Tutorial

ALINX

Preface

Some people want to ask if you can learn ZYNQ with zero foundation? It depends on where the zero is. If you can't understand the schematic, you don't know what an array is in the C language, and you have no concept of pointers. This is a negative foundation. Learning ZYNQ requires basic hardware knowledge and proficient C language skills.

This tutorial is part of the FPGA tutorial. Through continuous practice, you can master the basic process of FPGA development. Although it does not explain many principles, practice makes perfect, practice more, and gradually master the mysteries.

Version Record

Version	Date	Release By	Description
Rev1.0	2021-04-11	Rachel Zhou	First Release

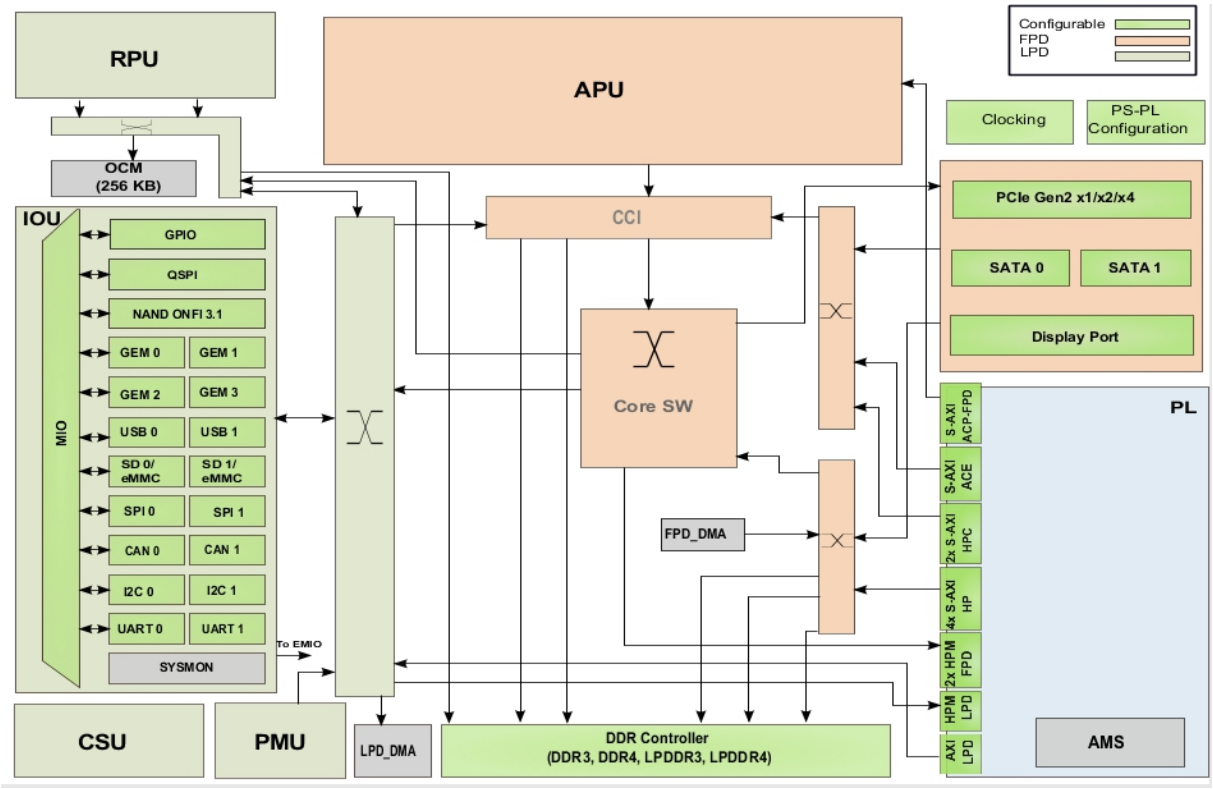
We promise that this tutorial is not a permanent, consistent document. We will continue to revise and optimize the tutorial based on the feedback of the forum and the actual development experience.

Content

Part 1: Introduction to Ultrascale+ MPSoC

The Zynq UltraScale+ MPSoC family is the second-generation Zynq platform of Xilinx. The highlight is that the FPGA contains a complete ARM processing subsystem (PS), including quad-core Cortex-A53 processors or dual-core Cortex-A53 plus dual-core Cortex-R5 processors. The entire processor is built around the processor. , And the processor subsystem integrates a memory controller and a large number of peripherals, so that the processor core is completely independent of the programmable logic unit in Zynq. That is to say, if the programmable logic unit (PL) is not used temporarily, The subsystem of the ARM processor can also work independently, which is essentially different from the previous FPGA, which is processor-centric.

Zynq is divided into two major functional blocks, the PS part and the PL part, which are the SOC part of ARM and the FPGA part. Among them, PS integrates APU ARM Cortex™-A53 processor, RPU Cortex-R5 processor, AMBA® interconnection, internal memory (OCM), external memory interface (DDR Controller) and peripherals (IOU). These peripherals (IOU) mainly include USB bus interface, Ethernet interface, SD/eMMC interface, I2C bus interface, CAN bus interface, UART interface, GPIO, etc. High-speed interfaces such as PCIe, SATA, Display Port.



Overall block diagram of ZYNQ MPSoC chip

PS: Processing System (Processing System) is the part of ARM SoC that has nothing to do with FPGA.

PL: Programmable Logic (Progarmmable Logic), which is the FPGA part.

Part 1.1: 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.

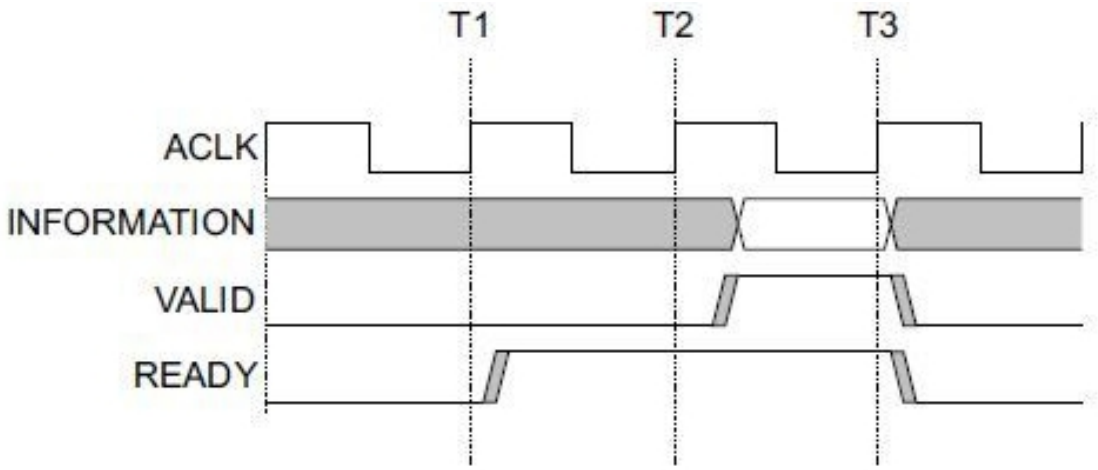
In fact, in the specific design, it isnot necessary to do too much work on the PS and PL connection technology, after joining the IP core, the system will automatically use the AXI interface to connect our IP core to the processor, we only You need to add a little more.

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 sent 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 1-1, 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：

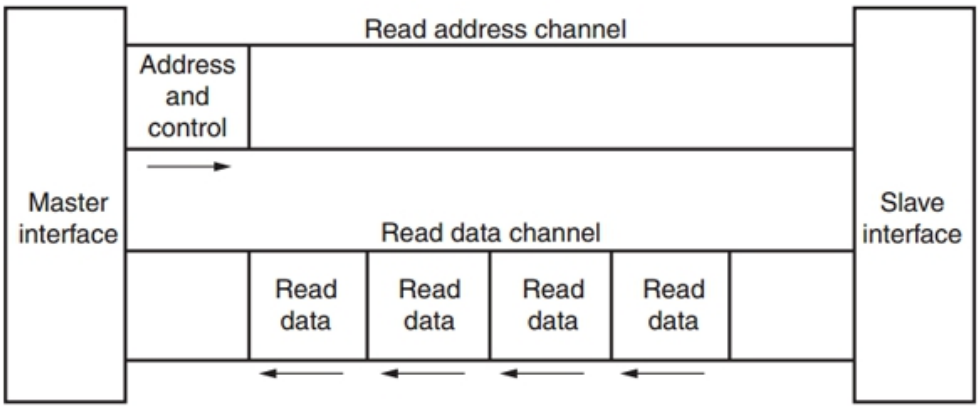
This is a continuous stream interface that does not require an address line (much like a FIFO, which is always read or written all the time). For this type of IP, ARM cannot be controlled by the above memory mapping method (the FIFO does not have the concept of an address at all), and there must be a 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:

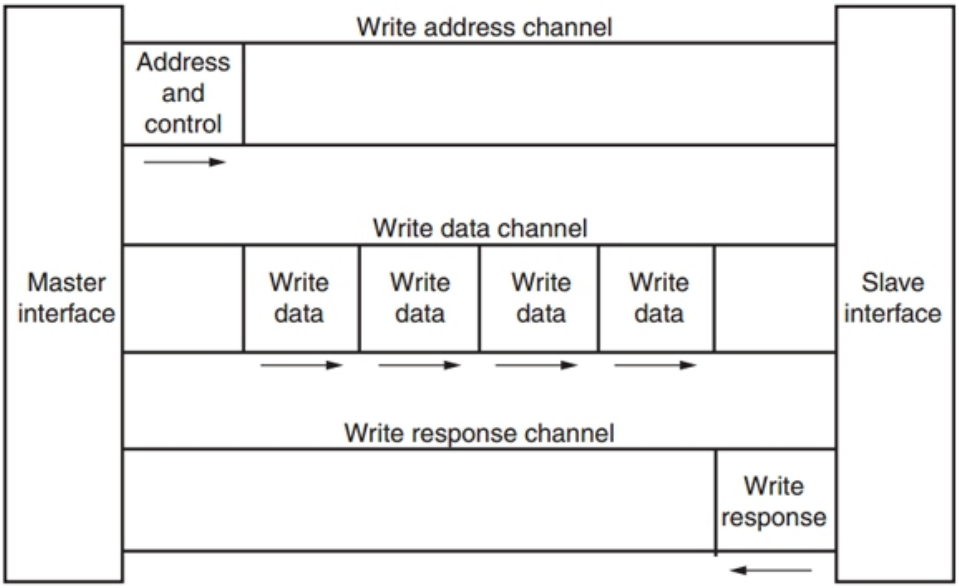
- Read Address Channel

- Write Address Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

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 with hardware inside the ZYNQ chip, including 12 physical interfaces, namely

S_AXI_HP{0:3}_FPD, S_AXI_LPD, S_AXI_ACE_FPD, S_AXI_ACP_FPD, S_AXI_HPC{0,1}_FPD, M_AXI_HPM{0,1}_FPD, M_AXI_HPM0_LPD interface.

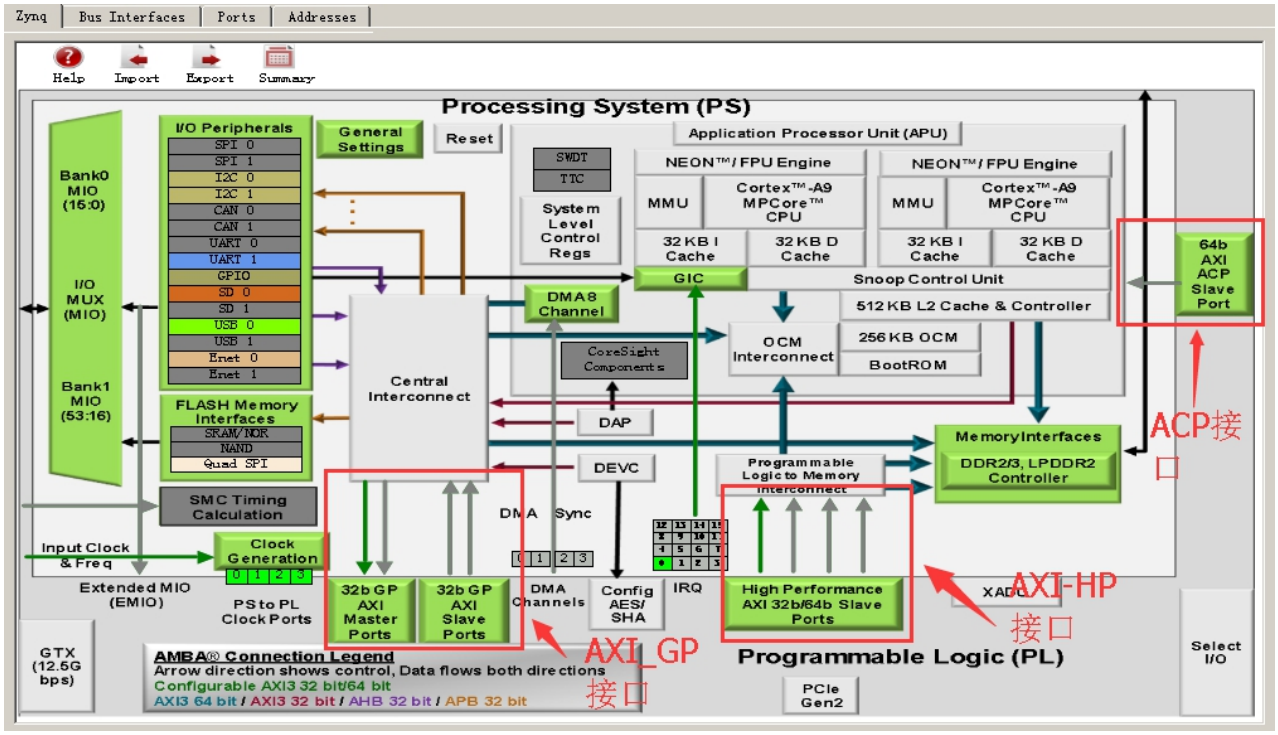
The S_AXI_HP{0:3}_FPD interface is a high-performance/bandwidth AXI4 standard interface, there are four in total, and the PL module is connected as the main device. Mainly used for PL to access memory on PS (DDR and FPD Main Switch)

S_AXI_LPD interface, high-performance port, connect PL to LPD. Low-latency access to OCM and TCM, access to PS side DDR.

S_AXI_HPC{0,1}_FPD interface, connect PL to FPD, connect to CCI, and access L1 and L2 Cache. Due to CCI, access to DDR controller will have a large delay.

M_AXI_HPM{0,1}_FPD interface, high-performance bus, PS is master, connects FPD to PL, and can be used for CPU, DMA, PCIe, etc. to push large amounts of data from PS to PL.

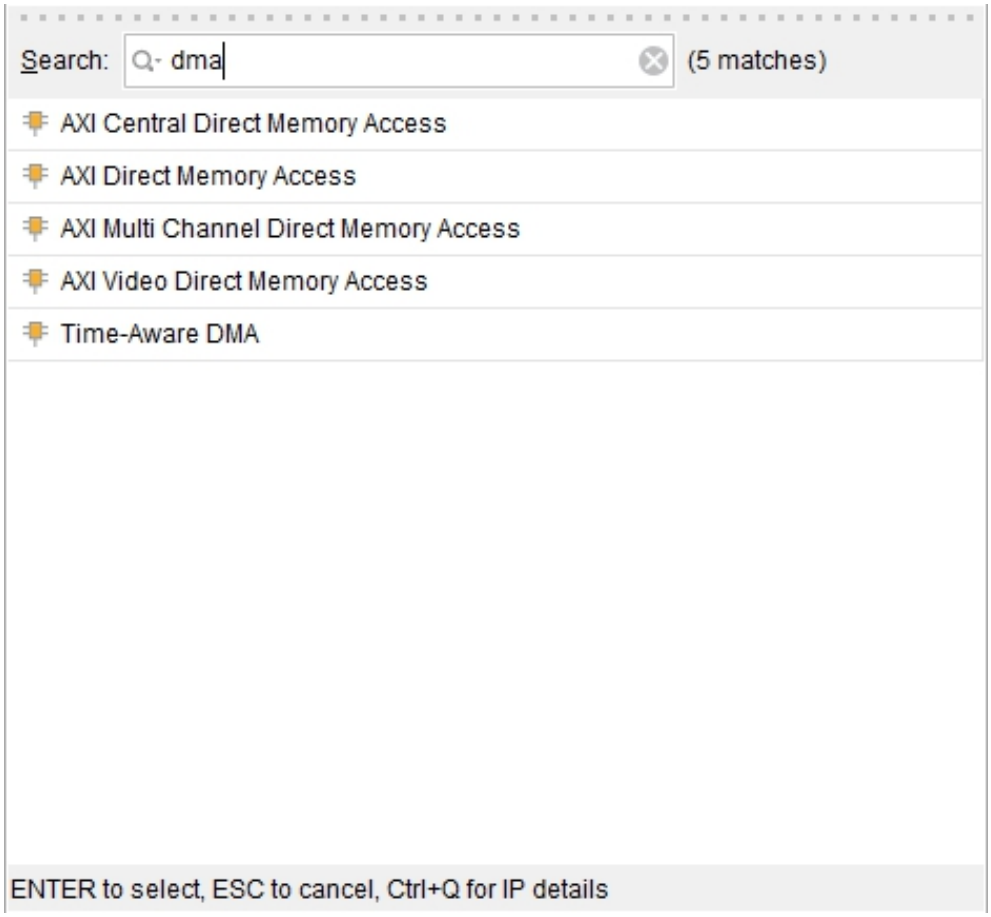
M_AXI_HPM0_LPD interface, low-latency interface bus, PS as master, connect LPD to PL, can directly access BRAM, DDR, etc. on the PL side, and is often used to configure the registers on the PL side.



Only M_AXI_HPM{0,1}_FPD and M_AXI_HPM0_LPD are Master Ports, namely host interfaces, and the rest are Slave Ports (slave interfaces). The host interface has the authority to initiate reading and writing. ARM can use the two host interfaces to actively access the PL logic. In fact, it maps the PL to a certain address, and reading and writing the PL register is like reading and writing its own memory. The other slave interfaces are passive interfaces, accepting reads and writes from the PL, and accept them in reverse. In PS and PL interconnection applications, the most used interfaces are S_AXI_HP{0:3}_FPD, M_AXI_HPM{0,1}_FPD and M_AXI_HPM0_LPD.

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 ready-made IP such as AXI-DMA, AXI-GPIO, AXI-Dataover, AXI-Stream in the Vivado development environment to implement the corresponding interfaces. When in use, you can directly add them from the Vivado IP list to

achieve the corresponding functions. The following picture shows 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.

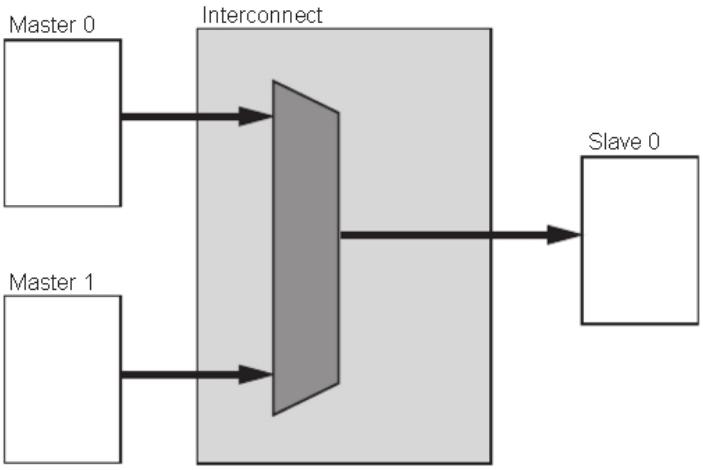
With the custom IP generated by these official IPs and wizards, users don't need to know too much about AXI timing (unless they do have problems), because Xilinx has encapsulated the details related to AXI timing, users only need Pay attention to your own logic implementation.

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.

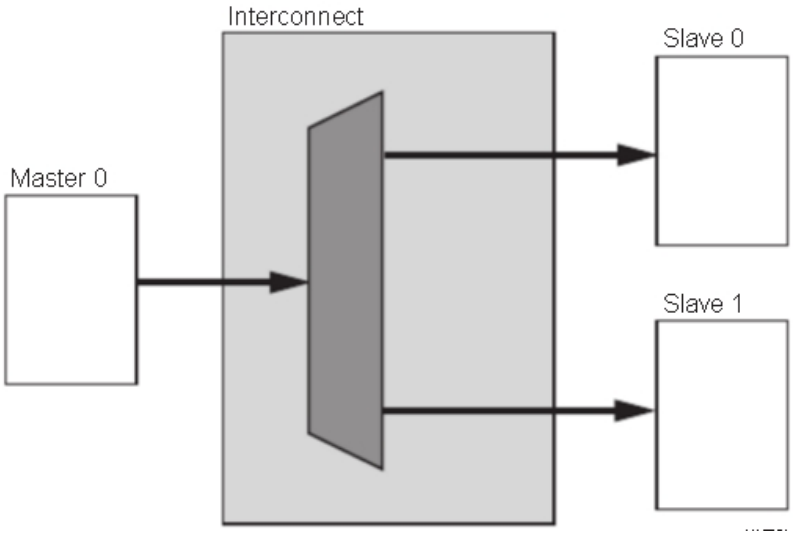
This AXI Interconnect IP core can support up to 16 master devices and 16 slave devices. If you need more interfaces, you can add more IP cores.

The AXI Interconnect basic connection modes are as follows:

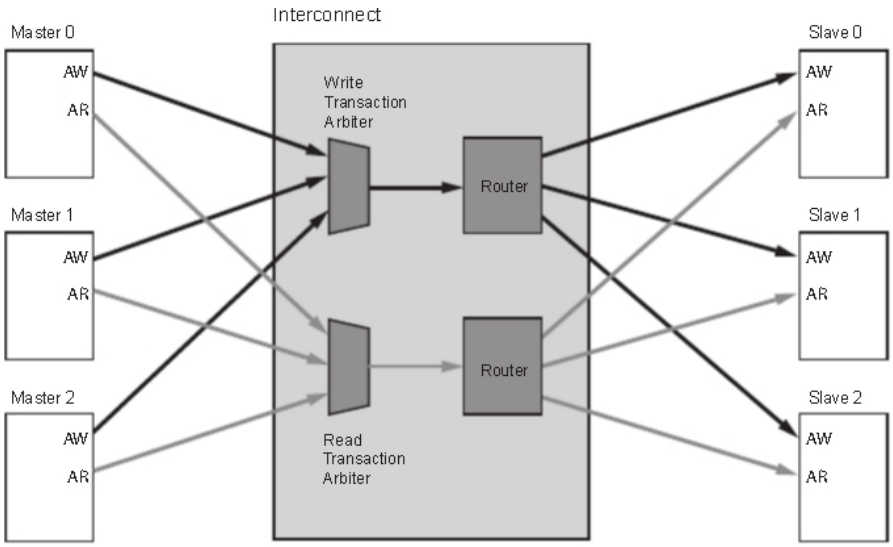
- N-to-1 Interconnect
- to-N Interconnect
- N-to-M Interconnect (Crossbar Mode)
- N-to-M Interconnect (Shared Access Mode)



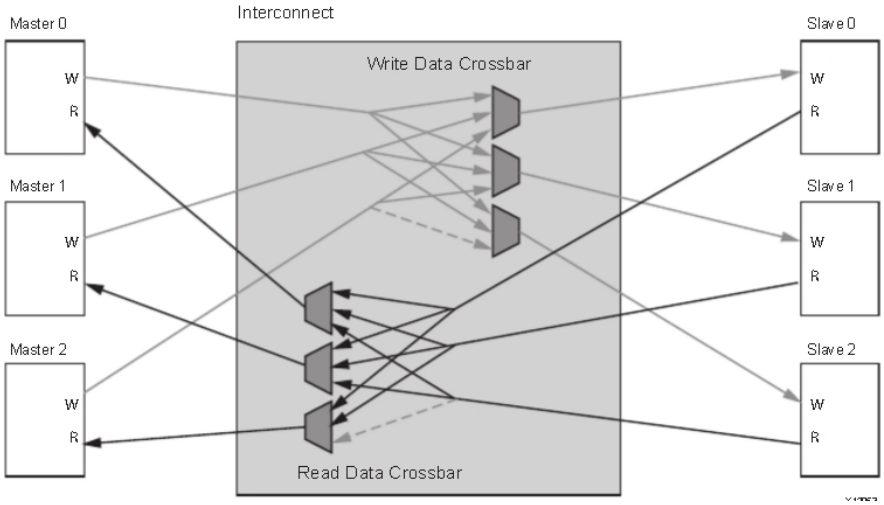
Many-to-one



One-to-many

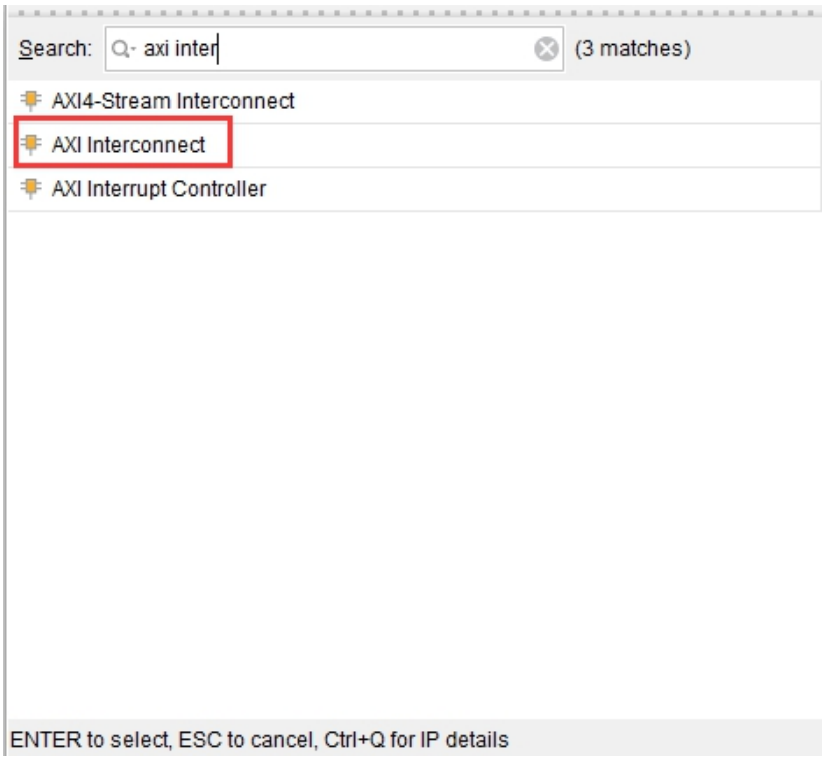


Many-to-many read and write address channels



Many-to-many read and write data channels

The AXI interface devices inside ZYNQ are interconnected by means of interconnection matrix, which not only ensures the efficiency of data transmission, but also ensures the flexibility of connection. In Xilinx Vivado, we provide the IP core axi_interconnect that realizes this interconnection matrix, and we just need to call it.



AXI Interconnect IP

Part 1.2: Introduction to the ZYNQ 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 1.3: 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 1.3.1: software developer

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

Part 1.3.2: Logic developer

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

Part 2: AXU2CGA/B Board Hardware Introduction

Refer to the AXU2CGA/B User Manual.pdf

Part 3: Introduction to Verilog basic modules

Part 3.1: Introduction to Verilog

This article mainly introduces the basic modules of Verilog, and consolidates the foundation, which will be very helpful for deep learning of FPGA.

Part 3.2: Data Types

Part 3.2.1: Constant

Integer: Integer can be represented by binary b or B, octal o or O, decimal d or D, hexadecimal h or H, for example, 8'b00001111 represents an 8-bit width binary integer, 4'ha represents 4 bits width hexadecimal integer.

X and Z values: X stands for unknown value, z stands for high impedance values, for example, 5'b00x11, the third one is unknown value, 3'b00z means the lowest bit is high impedance values.

Underscore: It can be used to divide the digits when the digits are too long to improve the readability of the program, such as 8'b0000_1111.

Parameters: you can use identifiers to define constants. Use only identifiers to improve readability and maintainability. For example, define parameter width = 8; define register reg [width 1:0] a; that is, define 8-bit width Register.

Transmission of parameters: If there are defined parameters in a module, parameters can be passed and modified when other modules call this module, as shown below, which is indicated by ##() after the

module.

For example, define the module as follows

Call module

```
module rom
#(
  parameter depth =15,
  parameter width = 8
  )
  (
   input [depth-1:0] addr ,
input [width-1:0] data ,
output result
) ;

endmodule
```

```
module top() ;

wire [31:0] addr ;
wire [15:0] data ;
wire result ;

rom
#(
  .depth(32),
  .width(16)
  )
r1
(
.addr(addr) ,
.data(data) ,
.result(result)
) ;
endmodule
```

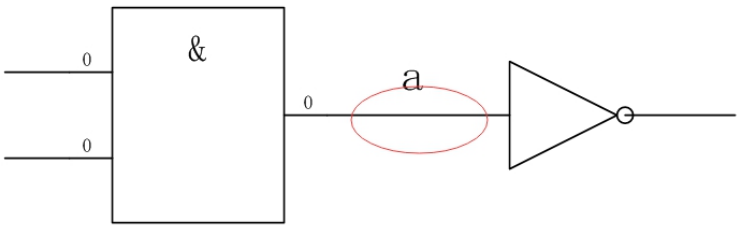
Parameter can be used for parameter transfer between modules, while localparam is only used within this module and cannot be used for parameter transfer. Localparam is mostly used to define the state of a state machine.

Part 3.3: Variable

Variables refer to the amount that can change its value when the program is running. The following mainly introduces several commonly used variable types.

Part 3.3.1: Wire

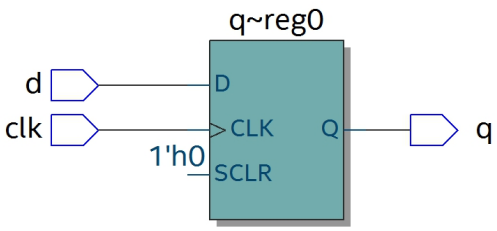
Wire type variables, also called nets type variables, are represent connection between hardware circuit, such as doors and doors, which cannot store values. Use continuous assignment statements assign to assign values, defined as wire [n-1:0] a; Where n represents the bit width, such as the definition of wire a; assign a = b; is to connect the node of b to the wire a. As shown in the figure below, the connection between the two entities is the wire type variable.



Part 3.3.2: Registers

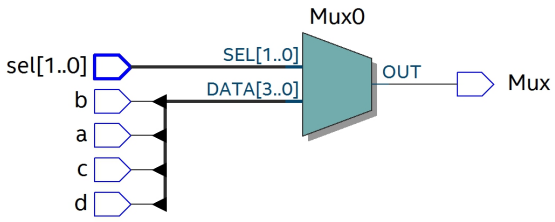
Reg type variables, also called register variables, can be used to store values and must be used in the always statement. Defined as reg [n-1:0] a, which means an n-bit width register, for example, reg[7:0] a, which means to define an 8-bit width register a. The register q is defined as shown below, the generated circuit is sequential logic, and the structure on the right is a D flip-flop.

```
module top(d, clk, q) ;
input  d  ;
input clk ;
output reg q ;

always @(posedge clk)
begin
  q <= d ;
end
endmodule
```



Combinational logic can also be generated, such as data selector, sensitive signals have no clock, reg Mux is defined, and the final generation circuit is combinational logic.

```
module top(a, b, c, d, sel, Mux) ;
input   a ;
input   b ;
input   c ;
input   d ;
input [1:0] sel ;
output reg Mux ;

always @(sel or a or b or c or d)
begin
  case(sel)
    2'b00 : Mux = a ;
    2'b01 : Mux = b ;
    2'b10 : Mux = c ;
    2'b11 : Mux = d ;
  endcase
end

endmodule
```



Part 3.3.3: Memory

The memory type can be used to define RAM, ROM and other memories. Its structure is reg [n-1:0] and memory name [m-1:0], meaning m registers with a width of n bits. For example, reg [7:0] ram [255:0] means that 256 8-bit registers are defined, 256 is the depth of the memory, and 8 is the data width.

Part 3.4: Operator Types

Operator symbols classified by the following categories:

1) Arithmetic Operators (+,—,*,/,%)
2) Assignment Operators （=，<=）
3) Relational Operators (>，<，>=，<=，==，！=)
4) Logical Operators (&&, II, !)
5) Conditional Operators (? :)
6) Bitwise Operators （~，|，^，&，^~）
7) Shift Operators (>>,<<)
8) Concatenation Operators （{}）

Part 3.4.1: Arithmetic Operators

"+" (add operator), "--" (subtract operator), "*" (multiply operator), "/" (divide operator, such as 7/3 = 2), "%" (modulus operator, that is to find the remainder, such as 7%3=1, the remainder is 1)

Part 3.4.2: Assignment Operators

"=" block assignment, "<=" non-blocking assignment. Blocking assignment means that one assignment statement is executed, and then the next one is executed. It can be understood as sequential execution, and the assignment is executed immediately; non-blocking assignment can be understood as parallel execution, regardless of order, and assignment is performed after the execution of the always block statement is completed. Such as the following blocking assignment:

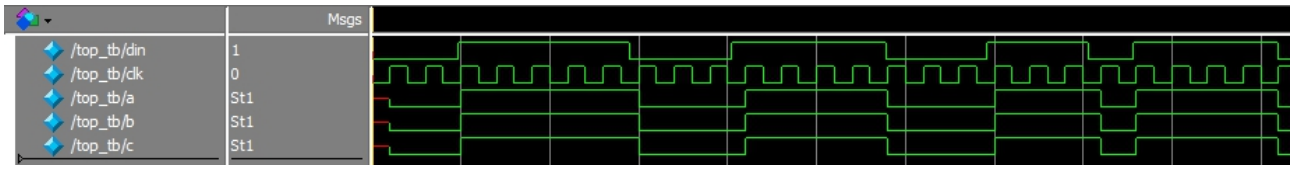
```
module top(din,a,b,c,clk);

input din;
input clk;
output reg a,b,c;

always @(posedge clk)
```

```
`timescale 1 ns/1 ns
module top_tb() ;
reg din ;
reg clk ;
wire a,b,c ;

initial
```

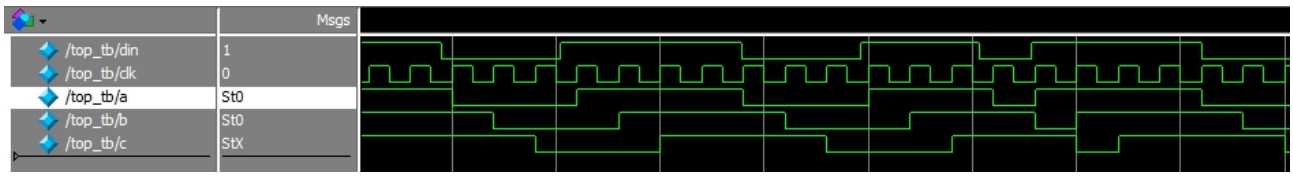
```
begin
        a = din;
        b = a;
        c = b;
end

endmodule
```

```
begin
  din =  0 ;
  clk = 0  ;
  forever
  begin
    #({$random}%100)
    din = ~din ;
  end
end

always #10 clk = ~clk ;

top
t0(.din(din),.a(a),.b(b),.c(c),.clk(clk)) ;
endmodule
```

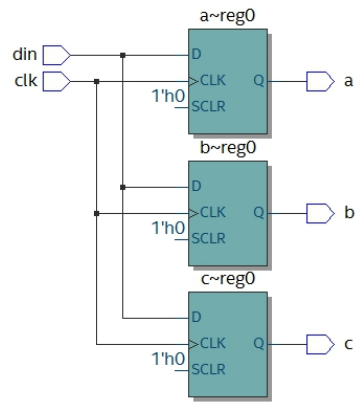
It can be seen from the simulation results that at the rising edge of clk, the value of a is equal to din, and the value of b is immediately assigned to b, and the value of b is assigned to c.



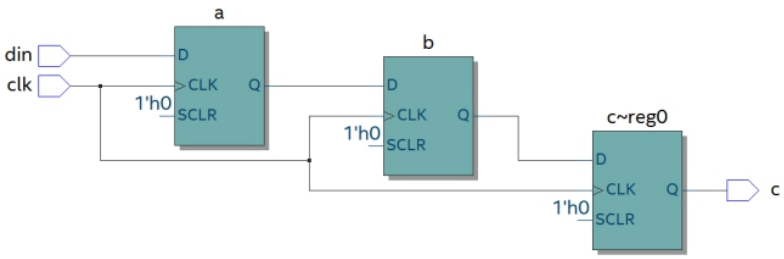
If it is changed to non-blocking assignment, the simulation result is as follows, at the rising edge of clk, the value of a is not immediately assigned to b, b is the original value of a, and similarly, c is the original value of b.



You can see the obvious difference from the RTL diagrams of the two:



Blocking Assignment RTL



Non-blocking Assignment RTL

Under normal circumstances, use non-blocking assignment in sequential logic circuits to avoid competition and risk in simulation; use blocking assignment in combinatorial logic, which changes immediately after the assignment statement is executed; blocking assignment must be used in assign statements.

Part 3.4.3: Relational Operators

Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to(>=),and less-than-or-equal-to (<=). if relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false. If there are any unknown or z bits in the operands, the expression takes a value x. These operators function exactly as the corresponding operators in the C programming language.

Part 3.4.4: Logical Operators

Logical operators are logical-and (& &), logical-or (II), and logical-not (!). Operators " & & " and "II" are binary operators. Operator (!) is a unary operators.

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B // Evaluates to a logical 0
A > B // Evaluates to a logical 1

Y >= X // Evaluates to a logical 1
Y < Z // Evaluates to an x
```

Part 3.4.5: Conditional Operators

The conditional operators (?:) takes three operands

Usage: *condition_expr ? true_expr : false_expr;*

The condition expression (*condition_expr*) is first evaluated. If the result is true (logical 1), then the true_expr is evaluated. If the result is false (logical 0), then the *false_expr* is evaluated

Part 3.4.6: Bitwise Operators

Bitwise operators are negation (~), and (&), or (I), xor (^), xnor(^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands. The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

Part 3.4.7: Shift Operators

Shift operators are *right shift (>>)* and *left shift (<<)*. These operators shift a vector operand to the right or the left by a specified number of bits. The operands are the vector and the number of bits to shift. When the bits are shifted,the vacant bit positions are filled with zeros. Shift operations do not wrap around.

```
// X = 4'b1100

Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.
Y = X << 2; //Y is 4'b0000. Shift left 2 bits.
```

Shift operators are useful because they allow the designer to model shift operations. *shift-and-add* algorithms for multiplication, and other useful operations.

Part 3.4.8: Concatenation Operators

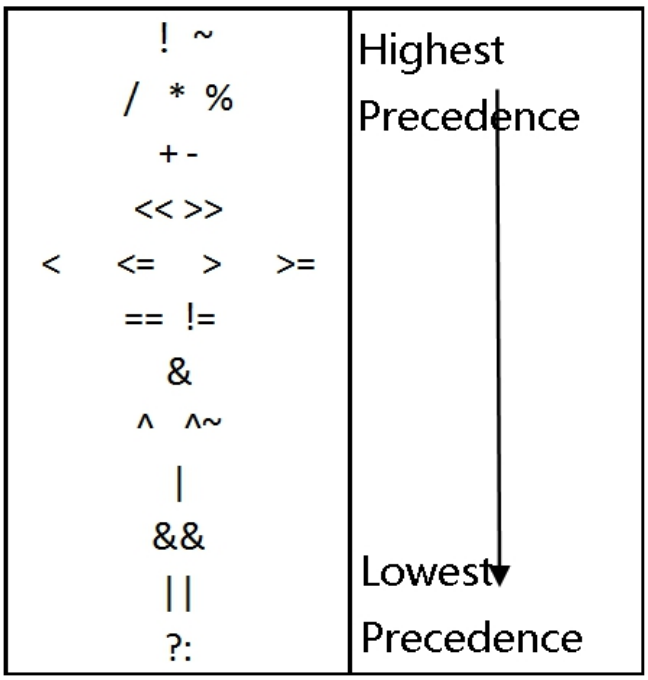
The concatenation operator ({ }) provides a mechanism to append multiple operands. The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.

Concatenations are expressed as operands within braces, with commas separating the operands. Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

Y = {B , C} // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001
Y = {A , B[0], C[1]} // Result Y is 3'b101
```

Part 3.4.9: Operator Procedure

Having discussed the operators, it is now important to discuss operator procedure. If no parentheses are used to separate parts of expressions, Verilog enforces the following procedure. Operators listed in below are in order from highest procedure to lowest procedure. It is recommended that parentheses be used the separate expression except in case of unary operator or when there is no ambiguity.



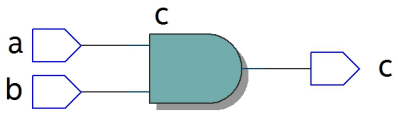
Part 3.5: Combinatorial logic

This section mainly introduces the combinational logic. The characteristic of the combinational logic circuit is that the output at any time depends only on the input signal. When the input signal changes, the output changes immediately and does not depend on the clock.

Part 3.5.1: Add Gates

In Verilog, "&" means bitwise and, such as c=a&b, the truth table is as follows, when a and b are both equal to 1, the result is 1, RTL means as shown as below:

Input		Output
a	b	c
0	0	0
0	1	0
1	0	0
1	1	1



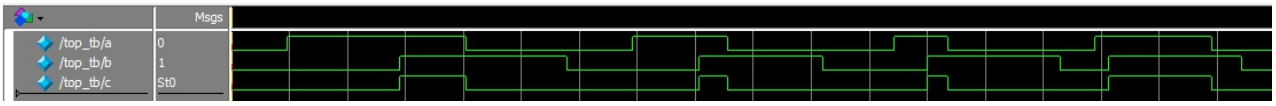
code as below:

```
module top(a, b, c) ;
input  a ;
input  b ;
output c ;

assign c = a & b ;
endmodule
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg a ;
reg b ;
wire c ;

initial
begin
  a = 0 ;
  b = 0 ;
  forever
  begin
    #({$random}%100)
    a = ~a ;
    #({$random}%100)
    b = ~b ;
  end
end

top  t0(.a(a), .b(b),.c(c)) ;

endmodule
```

The simulation results are as follows:



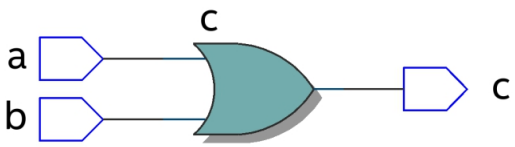
If the bit width of a and b is greater than 1, for example, define input [3:0] a, input [3:0]b, then a&b refers to the corresponding bitwise and of a and b. Such as a[0]&b[0],a[1]&b[1].

Part 3.5.2: Or Gates

In verilog, "|" means bitwise OR, such as c = a|b, the truth table is as follows, the result is 0 when both a and b are 0.

Input	Output

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1



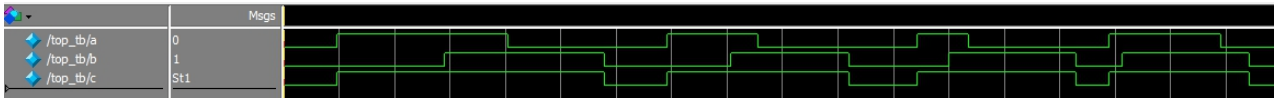
code as below:

```
module top(a, b, c) ;
input  a ;
input  b ;
output c ;

assign c = a | b ;
endmodule
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg a ;
reg b ;
wire c ;

initial
begin
  a = 0 ;
  b = 0 ;
  forever
  begin
    #({$random}%100)
    a = ~a ;
    #({$random}%100)
    b = ~b ;
  end
end

top  t0(.a(a), .b(b),.c(c)) ;

endmodule
```

The simulation results are as follows:

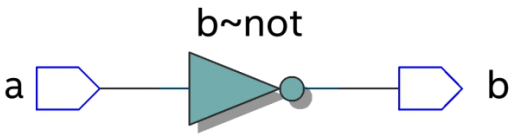


Similarly, if the bit width is greater than 1, it is a bitwise OR.

Part 3.5.3: Not Gates

In verilog, "~" means bitwise inversion, such as b=~a, the truth table is as follows: b is equal to the opposite of a.

Input	Output
a	b
0	1
1	0



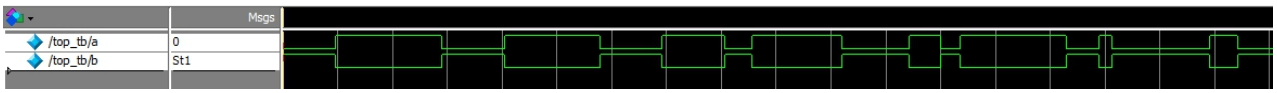
code as below:

```
module top(a, b) ;
input  a ;
output  b ;

assign b = ~a ;
endmodule
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg  a ;
wire b ;

initial
begin
  a = 0 ;
  forever
  begin
    #({$random}%100)
    a = ~a ;
  end
end

top  t0(.a(a), .b(b)) ;

endmodule
```

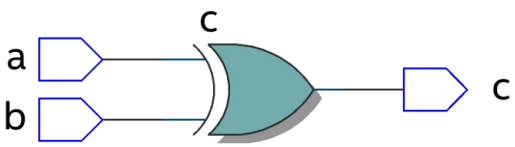
The simulation results are as follows:



Part 3.5.4: XOR Gates

In verilog, "^" means XOR, such as c= a^b, the truth table is as follows, when a and b are the same, the output is 0.

Input		Output
a	b	c
0	0	0
0	1	1
1	0	1
1	1	0



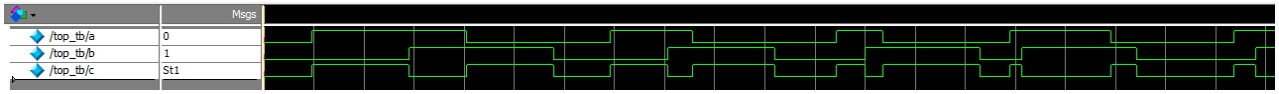
code as below:

```
module top(a, b, c) ;
input  a ;
input  b ;
output c ;

assign c = a ^ b ;
endmodule
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg a ;
reg b ;
wire c ;

initial
begin
  a = 0 ;
  b = 0 ;
  forever
  begin
    #({$random}%100)
    a = ~a ;
    #({$random}%100)
    b = ~b ;
  end
end

top  t0(.a(a), .b(b),.c(c)) ;

endmodule
```

The simulation results are as follows:

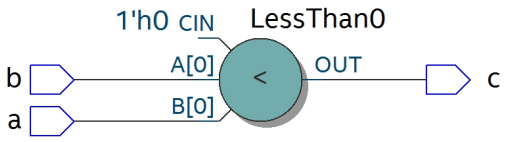


Part 3.5.5: Comparators

In verilog, use ">", "==", "<", ">=", "<=", "! =" means

Take ">" for example, such as c = a> b; means that if a is greater than b, then the value of c is 1, otherwise it is 0. The truth table is as follows:

Input		Output
a	b	c
0	0	0
0	1	0
1	0	0
1	1	1



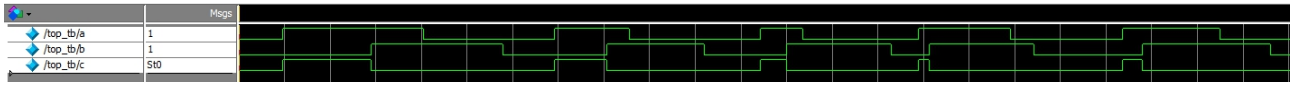
code as below:

```
module top(a, b, c) ;
input  a ;
input  b ;
output c ;

assign c = a > b ;
endmodule
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg a ;
reg b ;
wire c ;

initial
begin
  a = 0 ;
  b = 0 ;
  forever
  begin
    #({$random}%100)
    a = ~a ;
    #({$random}%100)
    b = ~b ;
  end
end

top  t0(.a(a), .b(b),.c(c)) ;

endmodule
```

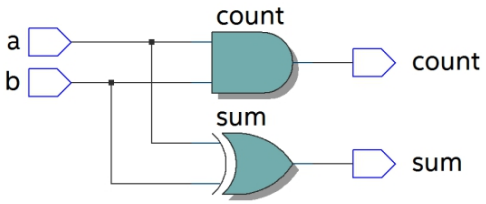
The simulation results are as follows:



Part 3.5.6: Half adder

Half adder and full adder are the basic units in arithmetic operation circuits. Since half adder does not consider carry from lower bits, it is called half adder, sum represents the addition result, count represents carry. The truth table can be expressed as follows:

Input		Output	
a	b	sum	count
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



code as below:

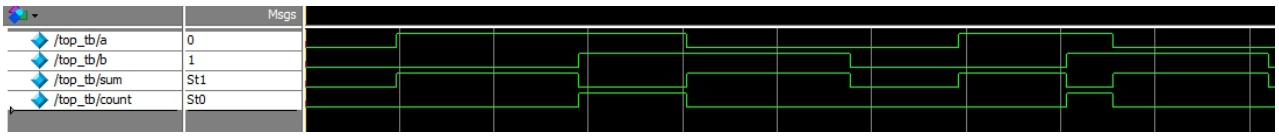
```
module top(a, b, sum, count) ;
input  a ;
input  b ;
output sum ;
output count ;

assign sum = a ^ b ;
assign count = a & b ;

endmodule
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg a ;
reg b ;
wire sum ;
wire count ;

initial
begin
  a = 0 ;
  b = 0 ;
  forever
  begin
    #({$random}%100)
    a = ~a ;
    #({$random}%100)
    b = ~b ;
  end
  end
```

```
top  t0(.a(a), .b(b),
.sum(sum), .count(count)) ;

endmodule
```

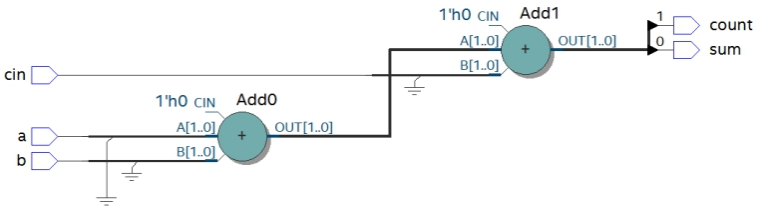
The simulation results are as follows:



Part 3.5.7: Full adder

The full adder needs to add the carry signal cin from the low bit. The truth table is as follows

Input			Output	
cin	a	b	sum	count
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



code as below:

```
module top(cin, a, b, sum, count) ;
input cin ;
input  a ;
input  b ;
output sum ;
output count ;

assign {count,sum} = a + b + cin ;
```

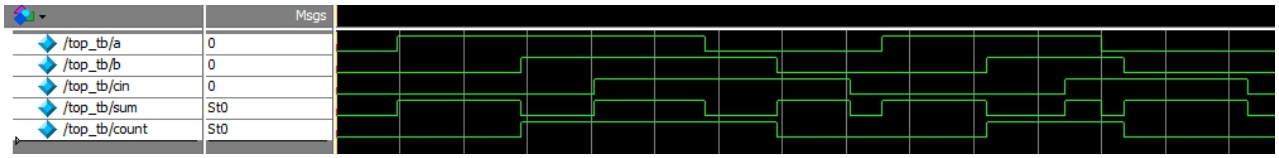
incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg a ;
reg b ;
reg cin ;
wire sum ;
wire count ;

initial
```

```
endmodule
```

```
begin
  a = 0 ;
  b = 0 ;
  cin = 0 ;
  forever
  begin
    #({$random}%100)
    a = ~a ;
    #({$random}%100)
 b = ~b ;
 #({$random}%100)
    cin = ~cin ;

  end
end

top  t0(.cin(cin),.a(a), .b(b),
.sum(sum), .count(count)) ;

endmodule
```

The simulation results are as follows:



Part 3.5.8:

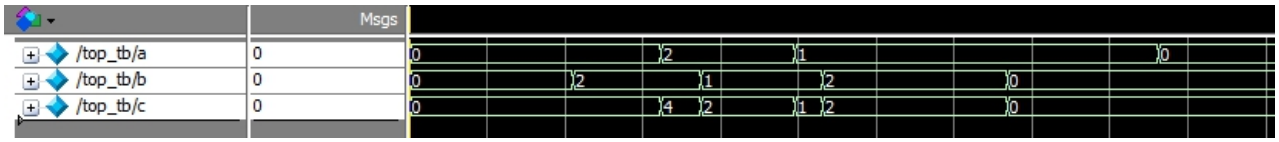
The representation of multiplication is also very simple, just use it, such as a*b, the example code is as follows

code as below: incentive file as follows:

```
module top(a, b, c) ;
input  [1:0] a ;
input  [1:0] b ;
output [3:0] c ;

assign c = a * b ;
endmodule
```

```
`timescale 1 ns/1 ns
module top_tb() ;
reg [1:0] a ;
reg [1:0] b ;
wire [3:0] c ;

initial
begin
  a = 0 ;
  b = 0 ;
  forever
  begin
    #({$random}%100)
    a = ~a ;
    #({$random}%100)
    b = ~b ;
  end
end

top  t0(.a(a), .b(b),.c(c)) ;

endmodule
```

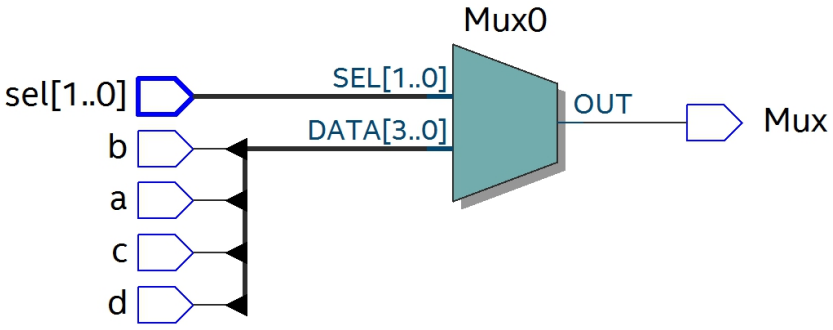
The simulation results are as follows:



Part 3.5.9: Data Selector

The data selector is often used in verilog. By selecting the signal, different input signals are selected and output to the output terminal. As shown in the truth table below, the data selector is selected from four. sel[1:0] is the selection signal, a, b, c, d are the input signals, and Mux is the output signal.

Select Signal		Input				Output
sel [0]	sel [1]	a	b	c	d	Mux
0	0	x	x	x	x	a
0	1	x	x	x	x	b
1	0	x	x	x	x	c
1	1	x	x	x	x	d



code as below:

```
module top(a, b, c, d, sel, Mux) ;
input  a ;
input  b ;
input  c ;
input  d ;

input [1:0] sel ;

output reg Mux ;

always @(sel or a or b or c or d)
begin
  case(sel)
    2'b00 : Mux = a ;
    2'b01 : Mux = b ;
    2'b10 : Mux = c ;
    2'b11 : Mux = d ;
  endcase
end
```

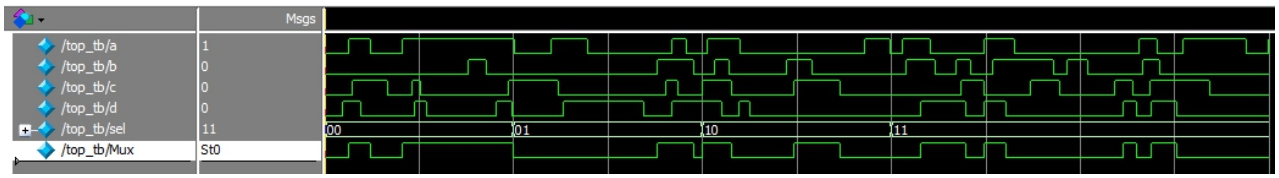
incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg  a ;
reg  b ;
reg  c ;
reg  d ;
reg [1:0] sel ;
wire  Mux ;

initial
begin
  a = 0 ;
  b = 0 ;
  c = 0 ;
  d = 0 ;
  forever
  begin
    #({$random}%100)
    a = {$random}%3 ;
    #({$random}%100)
```

```
endmodule
```

```
        b = {$random}%3 ;
        #({$random}%100)
        c = {$random}%3 ;
        #({$random}%100)
        d = {$random}%3 ;
    end
    end

    initial
    begin
      sel = 2'b00 ;
      #2000 sel =  2'b01 ;
      #2000 sel =  2'b10 ;
      #2000 sel =  2'b11 ;
    end

    top
    t0(.a(a), .b(b),.c(c),.d(d), .sel(sel
),
    .Mux(Mux)) ;

    endmodule
```

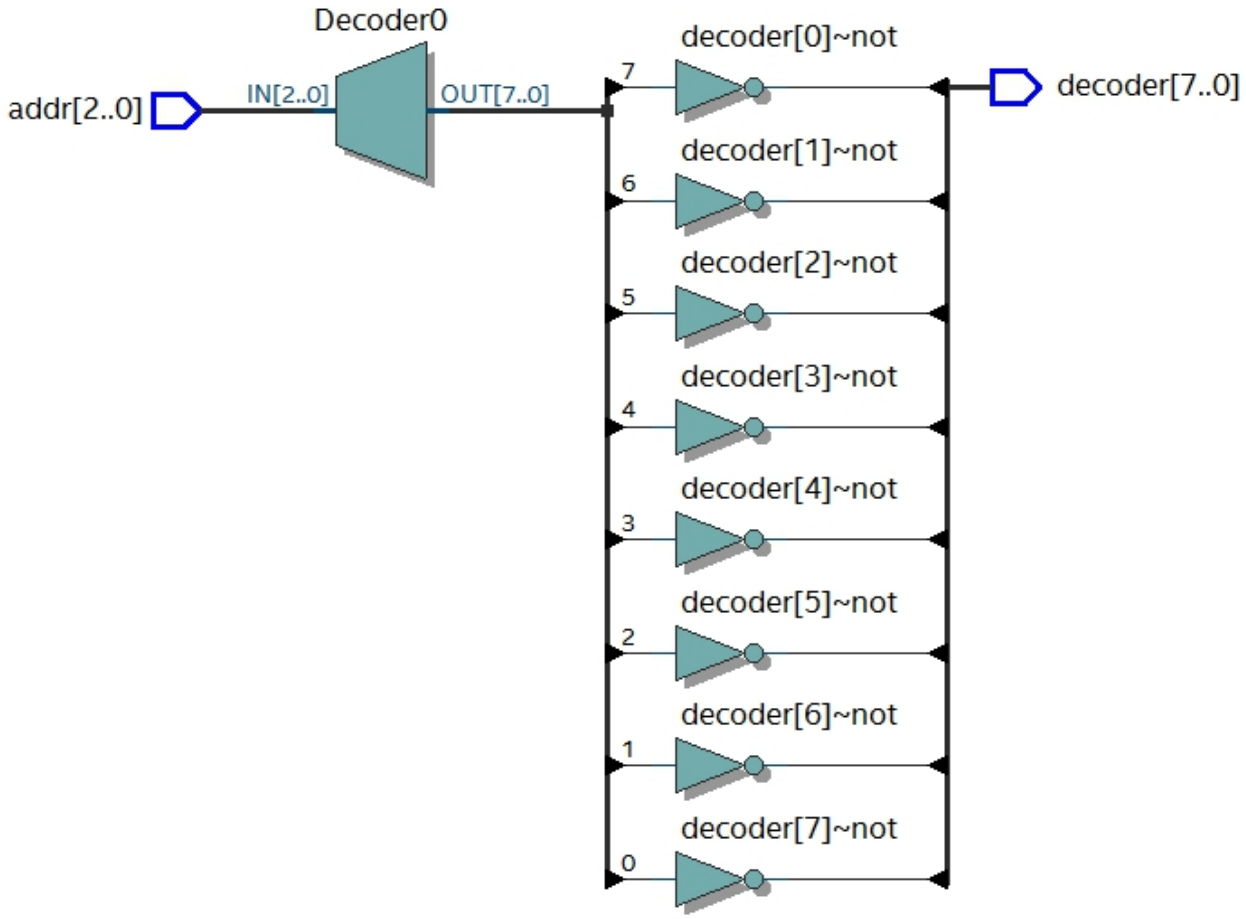
The simulation results are as follows:



Part 3.5.10: 3-8 decoder

The 3-8 decoder is a very commonly used device. Its truth table is shown below. According to the values of A2, A1, A0, different results can be obtained.

输入			输出							
A2	A1	A0	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
0	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0



code as below:

```
module top(addr, decoder) ;
input  [2:0] addr ;
output reg [7:0] decoder ;

always @(addr)
begin
  case(addr)
    3'b000 : decoder = 8'b1111_1110 ;
    3'b001 : decoder = 8'b1111_1101 ;
    3'b010 : decoder = 8'b1111_1011 ;
```

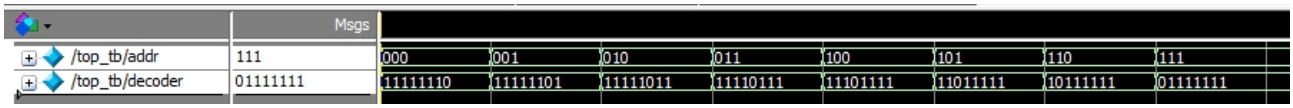
incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg  [2:0]  addr ;
wire  [7:0] decoder ;

initial
begin
  addr = 3'b000 ;
  #2000 addr =  3'b001 ;
  #2000 addr =  3'b010 ;
```

```
    3'b011 : decoder = 8'b1111_0111 ;
    3'b100 : decoder = 8'b1110_1111 ;
    3'b101 : decoder = 8'b1101_1111 ;
    3'b110 : decoder = 8'b1011_1111 ;
    3'b111 : decoder = 8'b0111_1111 ;
  endcase
end

endmodule
```

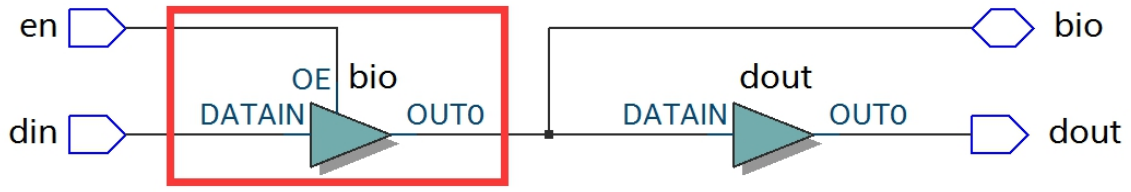
```
  #2000 addr =  3'b011 ;
  #2000 addr =  3'b100 ;
  #2000 addr =  3'b101 ;
  #2000 addr =  3'b110 ;
  #2000 addr =  3'b111 ;
end

top
t0(.addr(addr),.decoder(decoder)) ;

endmodule
```

The simulation results are as follows:



Part 3.5.11: Three-state gate

In FPGA use, bidirectional IO is often used, and a three-state gate is needed, such as bio = en? Din:1'bz; where en is the enable signal to open and close the three-state gate. The following RTL diagram is the realization of bidirectional IO, please refer to the code. The incentive file realizes the connection of two bidirectional IOs.



code as below:

```
module top(en, din, dout, bio) ;
input  din  ;
input  en ;
output dout ;
inout bio ;
```

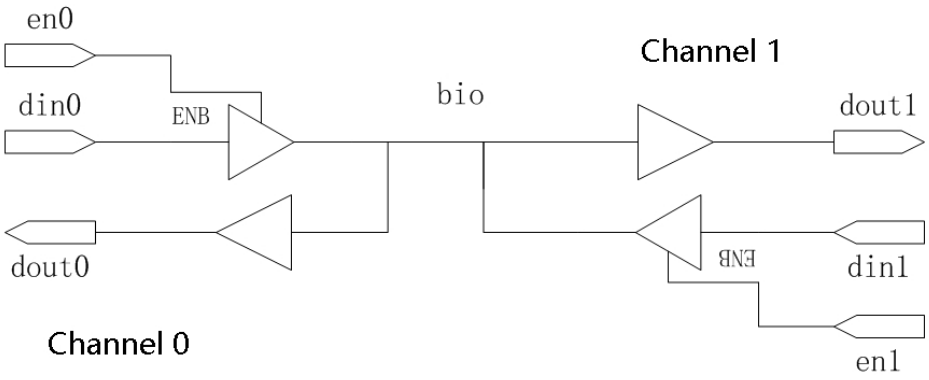
incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg en0 ;
reg din0 ;
wire dout0 ;
reg en1 ;
```

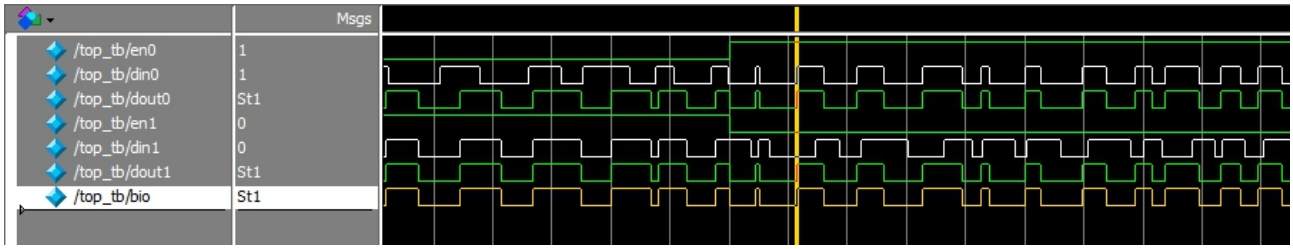
```
assign bio = en? din : 1'bz ;
assign dout = bio ;

endmodule
```

```
reg din1 ;
wire dout1 ;
wire bio ;

initial
begin
  din0 = 0 ;
  din1 = 0 ;
  forever
  begin
    #({$random}%100)
    din0 = ~din0 ;
    #({$random}%100)
 din1 = ~din1 ;
  end
end

initial
begin
  en0 = 0 ;
  en1 = 1 ;
  #100000
  en0 = 1 ;
  en1 = 0 ;
end

top
t0(.en(en0),.din(din0),.dout(dout0),
.bi
o(bio)) ;
top
t1(.en(en1),.din(din1),.dout(dout1),
.bi
o(bio)) ;

endmodule
```

The incentive file structure is shown below



The simulation results are as follows, when en0 is 0 and en1 is 1, channel 1 is opened, bidirectional IO bio is equal to din1 of channel 1, channel 1 transmits data out, channel 0 receives data, dout0 is equal to bio; When en0 is 1 and en1 is 0, channel 0 is open, bidirectional IO bio is equal to din0 of channel 0, channel 0 transmits data out, channel 1 receives data, dout1 is equal to bio.



Part 3.6: Sequential Logic

The characteristic of the logic function of the combinational logic circuit is that the output at any time depends only on the input at the current moment, and has nothing to do with the original state of the circuit. The characteristic of sequential logic in the logic function is that the output at any time depends not only on the current input signal, but also on the original state of the circuit. The following is a typical sequential logic analysis.

Part 3.6.1: D Flip-Flop

The D flip-flop stores data on the rising or falling edge of the clock,

and the output is the same as the state of the input signal before the clock jumps.

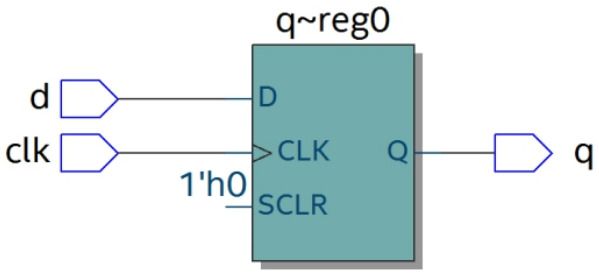
code as below:

```
module top(d, clk, q) ;
input  d  ;
input clk ;
output reg q ;
always @(posedge clk)
begin
  q <= d ;
end

endmodule
```

incentive file as follows:

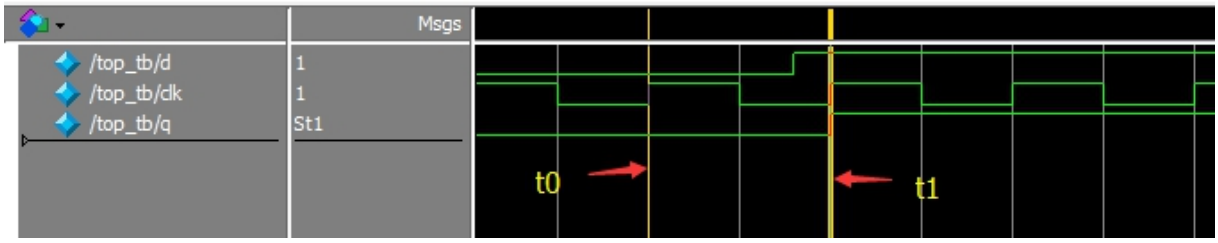
```
`timescale 1 ns/1 ns
module top_tb() ;
reg d ;
reg clk ;
wire q ;

initial
begin
  d = 0 ;
  clk = 0 ;
  forever
  begin
    #({$random}%100)
    d = ~d ;
  end
end

always #10 clk = ~clk ;

top  t0(.d(d),.clk(clk),.q(q)) ;

endmodule
```

The RTL diagram is expressed as follows



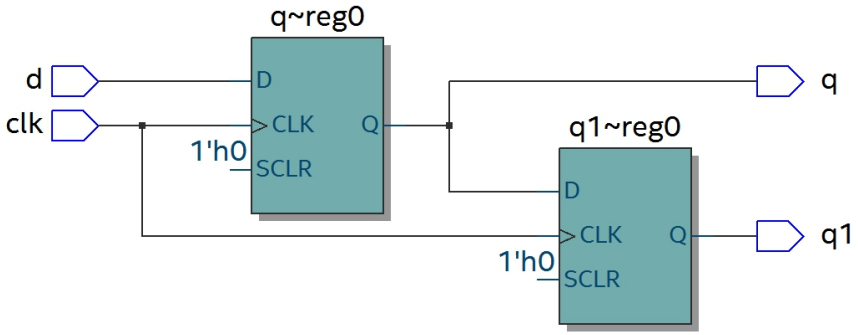
The simulation results are as follows. It can be seen that at time t0, the value of d is 0, and the value of q is also 0; when d changes at time t1, the value is 1, then q changes accordingly, and the value

becomes 1 . It can be seen that in a clock cycle between t0 and t1, no matter how the value of the input signal d changes, the value of q remains unchanged, that is, there is a storage function, and the saved value is d at the clock transition edge Value.



Part 3.6.2: Two-Level D Flip-Flop

The software performs timing analysis according to the two-level D flip-flop model. Specifically, it can analyze the difference between the data output by the two D flip-flops at the same time. The RTL diagram is as follows:



code as below:

```
module top(d, clk, q, q1) ;
input  d  ;
input clk ;
output reg q ;
output reg q1 ;

always @(posedge clk)
begin
  q <= d ;
end
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg d ;
reg clk ;
wire q ;
wire q1 ;

initial
begin
  d = 0 ;
```

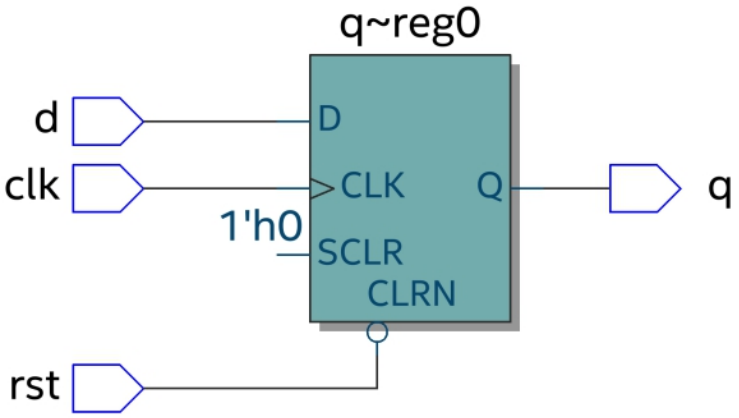
```
always @(posedge clk)
begin
  q1 <= q ;
end

endmodule
```

```
  clk = 0 ;
  forever
  begin
    #({$random}%100)
    d = ~d ;
  end
end

always #10 clk = ~clk ;

top
t0(.d(d),.clk(clk),.q(q),.q1(q1)) ;

endmodule
```

The simulation results are as follows. It can be seen that at t0, d is 0, q output is 0, and at t1, q changes with the data of d, and the value of q is still 0 before this clock jump. Then the value of q1 is still 0, and at t2, before the clock jumps, the value of q is 1, then the value of q1 is correspondingly 1, and q1 is one cycle behind q.

Part 3.6.3: D Flip-Flop with Asynchronous Reset

Asynchronous reset refers to being independent of the clock, once the asynchronous reset signal is valid, the reset operation is triggered. This function is often used when writing code to reset and initialize the signal. The RTL diagram is as follows:



The code is as follows, pay attention to the asynchronous reset signal in the sensitive list, if it is a low level reset, it is negedge, if it is a high level reset, it is posedge

code as below:

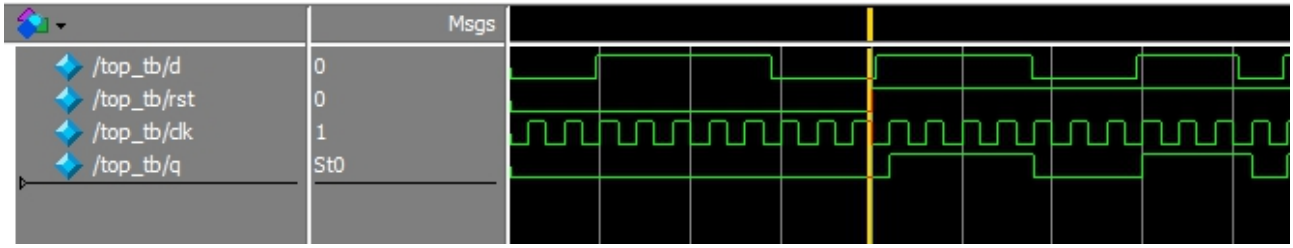
```
module top(d, rst, clk, q) ;
input  d  ;
input rst ;
input clk ;
output reg q ;

always @(posedge clk or negedge rst)
begin
  if (rst == 1'b0)
    q <= 0 ;
  else
    q <= d ;
end

endmodule
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg d ;

reg rst ;
reg clk ;
wire q ;

initial
begin
  d = 0 ;
  clk = 0 ;
  forever
  begin
    #({$random}%100)
    d = ~d ;
  end
end

initial
begin
  rst = 0 ;
```

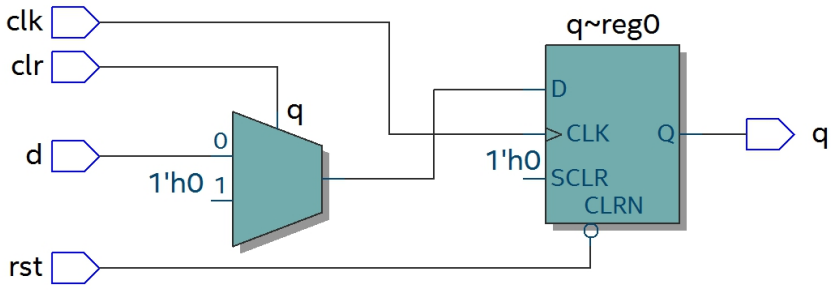
```
        #200 rst = 1 ;
    end

    always #10 clk = ~clk ;

    top
    t0(.d(d),.rst(rst),.clk(clk),.q(q))
;

    endmodule
```

The simulation results are as follows. It can be seen that before the reset signal, although the input signal d data has changed, because it is in the reset state, the input signal q is always 0 and the value of q is normal after reset.



Part 3.6.4: D flip-flop with Asynchronous Reset and Synchronous Clearing

As mentioned earlier, asynchronous reset is independent of clock operations, while synchronous clearing operates synchronously with the clock signal. Of course, it is not limited to synchronous clearing, but can also be other synchronous operations. The RTL diagram is as follows:



The code is as follows, unlike asynchronous reset, synchronous operation cannot put the signal in the sensitive list.

code as below:

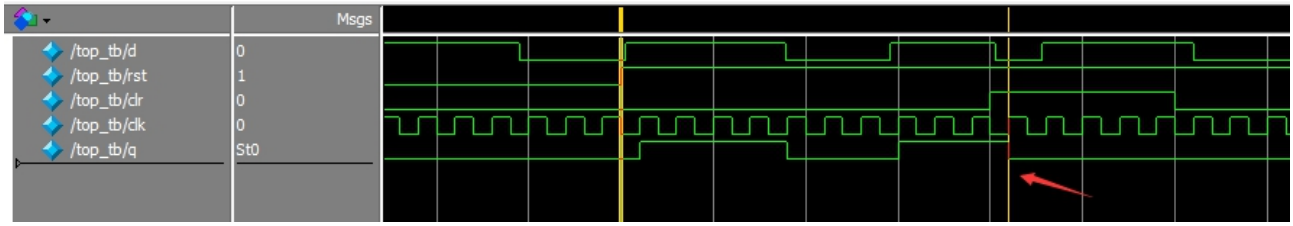
```
module top(d, rst, clr, clk, q) ;
input  d  ;
input rst ;
input clr ;
input clk ;
output reg q ;

always @(posedge clk or negedge rst)
begin
  if (rst == 1'b0)
    q <= 0 ;
  else if (clr == 1'b1)
    q <= 0 ;
  else
    q <= d ;
end

endmodule
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg d ;
reg rst ;
reg clr ;
reg clk ;
wire q ;

initial
begin
  d = 0 ;
  clk = 0 ;
  forever
  begin
    #({$random}%100)
    d = ~d ;
  end
end

initial
begin
  rst = 0 ;
  clr = 0 ;
  #200 rst = 1 ;
  #200 clr = 1 ;
  #100 clr = 0 ;
end

always #10 clk = ~clk ;
```

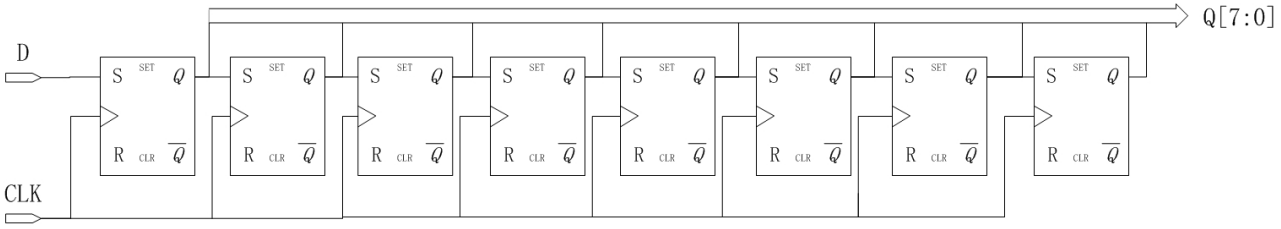
```
    top
    t0(.d(d),.rst(rst),.clr(clr),.clk(cl
k),
    .q(q)) ;

    endmodule
```

The simulation results are as follows. It can be seen that after the clr signal is pulled high, q is not cleared immediately, but is cleared after the next rising edge of clk, that is, clr is synchronized with clk.



Part 3.6.5: Shift Register

The shift register means that when each clock pulse comes, it moves one bit to the left or right. Due to the characteristics of the D flip-flop, the data output is synchronized with the clock edge. Its structure is as follows. When each clock comes, the output q of each D flip-flop is equal to the output value of the previous D flip-flop, so as to realize the function of shifting.



code as below:

```
module top(d, rst, clk, q) ;
input  d  ;
```

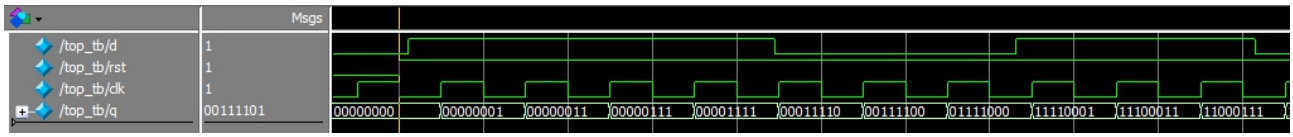
incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
```

```
input rst ;
input clk ;
output reg [7:0] q ;

always @(posedge clk or negedge rst)
begin
  if (rst == 1'b0)
    q <= 0 ;
  else
    q <= {q[6:0], d} ;  //Shift Left
  //q <= {d, q[7:1]} ;  //Shift Right
end

endmodule
```

```
reg d ;

reg rst ;
reg clk ;
wire [7:0] q ;

initial
begin
  d = 0 ;
  clk = 0 ;
  forever
  begin
    #({$random}%100)
    d = ~d ;
  end
end

initial
begin
  rst = 0 ;
  #200 rst = 1 ;
end

always #10 clk = ~clk ;

top
t0(.d(d),.rst(rst),.clk(clk),.q(q))
;

endmodule
```

The simulation results are as follows, you can see that after reset, each rising edge of clk shifts one bit to the left



Part 3.6.6: Single Port RAM

The write address and read address of single-port RAM share the same address, the code is as follows, where reg [7:0] ram [63:0] means that 64 8-bit width data are defined. Among them, addr_reg is defined, which can keep the read address and send out the data after a period of delay.

code as below:

```
module top
(
  input [7:0] data,
  input [5:0] addr,
  input wr,
  input clk,
  output [7:0] q
);

reg [7:0] ram[63:0];   //declare ram
reg [5:0] addr_reg;    //addr register

always @ (posedge clk)
begin
  if (wr)              //write
    ram[addr] <= data;

  addr_reg <= addr;
end

assign q = ram[addr_reg];  //read data
endmodule
```

incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg [7:0] data ;
reg [5:0] addr ;
reg wr ;
reg clk ;
wire [7:0] q ;

initial
begin
  data = 0 ;
  addr = 0 ;
  wr = 1 ;
  clk = 0 ;
  end

always #10 clk = ~clk ;

always @(posedge clk)
begin
  data <= data + 1'b1 ;
  addr <= addr + 1'b1 ;
end

top  t0(.data(data),
        .addr(addr),
        .clk(clk),
        .wr(wr),
        .q(q)) ;
endmodule
```

The simulation results are as follows, you can see that the output of q is consistent with the written data

Part 3.6.7: Pseudo Dual-Port RAM

The read and write addresses of the pseudo dual-port RAM are independent, and the write or read addresses can be randomly selected, and read and write operations are performed at the same time. The code is as follows. The en signal is defined in the excitation file, and the read address is sent when it is valid.

code as below:

```
module top
(
  input [7:0] data,
  input [5:0] write_addr,
  input [5:0] read_addr,
  input wr,
  input rd,
  input clk,
  output reg [7:0] q
);

reg [7:0] ram[63:0];   //declare ram
reg [5:0] addr_reg;    //addr register

always @ (posedge clk)
begin
  if (wr)              //write
    ram[write_addr] <= data;
  if (rd)              //read
     q <= ram[read_addr];
end
```

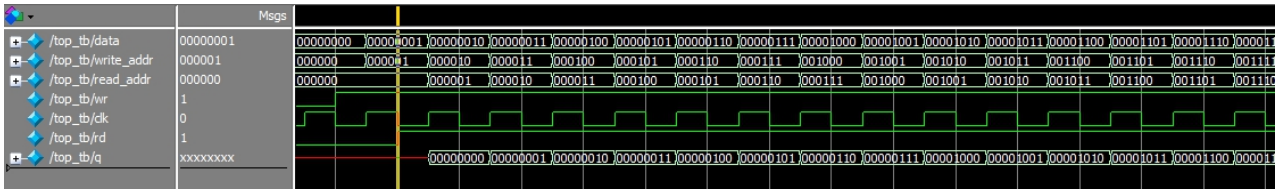
incentive file as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg [7:0] data ;
reg [5:0] write_addr ;
reg [5:0] read_addr ;
reg wr ;
reg clk ;
reg rd ;
wire [7:0] q ;

initial
begin
  data = 0 ;
  write_addr = 0 ;
  read_addr = 0 ;
  wr = 0 ;
  rd = 0 ;
  clk = 0 ;
  #100 wr = 1 ;
  #20 rd = 1 ;
end
```

```
endmodule
```

```
always #10 clk = ~clk ;

always @(posedge clk)
begin
  if (wr)
  begin
     data <= data + 1'b1 ;
     write_addr <= write_addr + 1'b1 ;
     if (rd)
       read_addr <= read_addr + 1'b1 ;
  end
end

top  t0(.data(data),
        .write_addr(write_addr),
        .read_addr(read_addr),
        .clk(clk),
        .wr(wr),
        .rd(rd),
        .q(q)) ;
endmodule
```

The simulation results are as follows. It can be seen that when rd is valid, the read address is operated and the data is read



Part 3.6.8: True Dual-Port RAM

True dual-port RAM has two sets of control lines and data lines, allowing two systems to read and write to it. The code is as follows:

code as below:

```
module top
(
  input [7:0] data_a, data_b,
  input [5:0] addr_a, addr_b,
```

incentive file as follows:

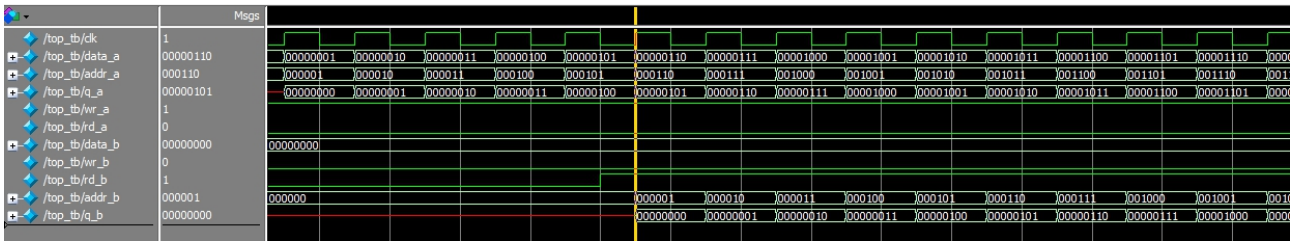
```
`timescale 1 ns/1 ns
module top_tb() ;
reg [7:0] data_a, data_b ;
reg [5:0] addr_a, addr_b ;
```

```
  input wr_a, wr_b,
  input rd_a, rd_b,
  input clk,
  output reg [7:0] q_a, q_b
);

reg [7:0] ram[63:0];   //declare ram

//Port A
always @ (posedge clk)
begin
  if (wr_a)              //write
    begin
     ram[addr_a] <= data_a;
     q_a <= data_a ;
    end
  if (rd_a)
//read
     q_a <= ram[addr_a];
end


//Port B
always @ (posedge clk)
begin
  if (wr_b)              //write
    begin
     ram[addr_b] <= data_b;
     q_b <= data_b ;
    end
  if (rd_b)
//read
     q_b <= ram[addr_b];
end

endmodule
```

```
reg wr_a, wr_b ;
reg rd_a, rd_b ;
reg clk ;
wire [7:0] q_a, q_b ;

initial
begin
  data_a = 0 ;
  data_b = 0 ;
  addr_a = 0 ;
  addr_b = 0 ;
  wr_a = 0 ;
  wr_b = 0 ;
  rd_a =  0 ;
  rd_b = 0 ;
  clk = 0 ;
  #100 wr_a = 1 ;
  #100 rd_b = 1 ;
end

always #10 clk = ~clk ;

always @(posedge clk)
begin
  if (wr_a)
  begin
    data_a <= data_a + 1'b1 ;
    addr_a <= addr_a + 1'b1 ;
  end
  else
begin
     data_a <= 0 ;
     addr_a <= 0 ;
  end
end

always @(posedge clk)
begin
  if (rd_b)
    begin
     addr_b <= addr_b + 1'b1 ;
```

```
        end
      else addr_b <= 0 ;

    end

    top
    t0(.data_a(data_a), .data_b(data_b),
           .addr_a(addr_a), .addr_b(addr_
b
    ),
           .wr_a(wr_a), .wr_b(wr_b),
           .rd_a(rd_a), .rd_b(rd_b),
           .clk(clk),
           .q_a(q_a), .q_b(q_b)) ;
    endmodule
```

The simulation results are as follows



Part 3.6.9: Single Port ROM

ROM is used to store data. You can initialize ROM according to the following code form, but this method is more troublesome to deal with large-capacity ROM. It is recommended to use the ROM IP core that comes with FPGA to implement it and add initialization files.

code as below:

```
module top
(
  input [3:0] addr,
  input clk,
  output reg [7:0] q
);
```

incentive file as follows:

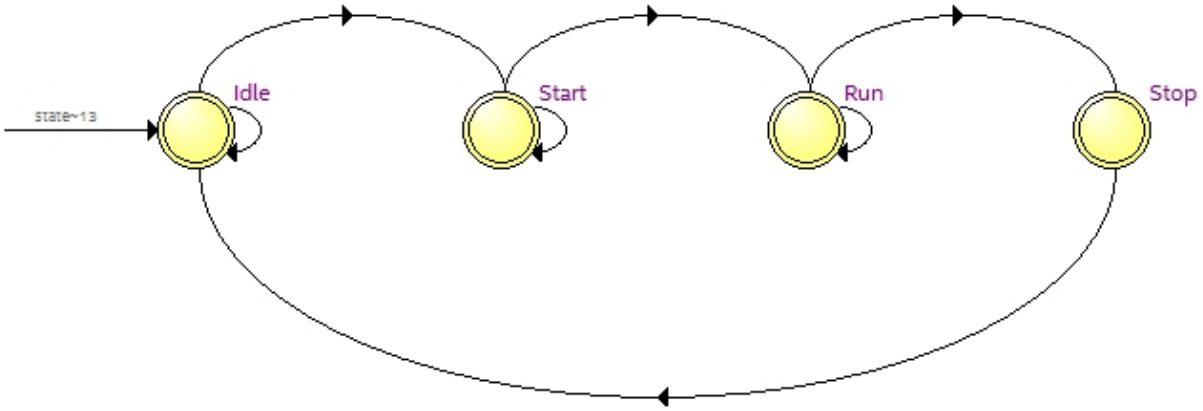
```
`timescale 1 ns/1 ns
module top_tb() ;
reg [3:0] addr ;
reg clk ;
wire [7:0] q ;
```

```
always @(posedge clk)
begin
  case(addr)
   4'd0  : q <= 8'd15  ;
   4'd1  : q <= 8'd24  ;
   4'd2  : q <= 8'd100 ;
   4'd3  : q <= 8'd78  ;
   4'd4  : q <= 8'd98  ;
   4'd5  : q <= 8'd105 ;
   4'd6  : q <= 8'd86  ;
   4'd7  : q <= 8'd254 ;
   4'd8  : q <= 8'd76  ;
   4'd9  : q <= 8'd35  ;
   4'd10 : q <= 8'd120 ;
   4'd11 : q <= 8'd85  ;
   4'd12 : q <= 8'd37  ;
   4'd13 : q <= 8'd19  ;
   4'd14 : q <= 8'd22  ;
   4'd15 : q <= 8'd67  ;
   default: q <= 8'd0 ;
  endcase
end

endmodule
```

```
initial
begin
  addr = 0 ;
  clk = 0 ;
end

always #10 clk = ~clk ;

always @(posedge clk)
begin
    addr <= addr + 1'b1 ;
end

top  t0(.addr(addr),
        .clk(clk),
        .q(q)) ;
endmodule
```

Part 3.6.10: Finite State Machine

Finite state machines are often used in verilog to process relatively complex logic, set different states, jump to the corresponding state according to the trigger condition, and do corresponding processing in different states. Finite state machines mainly use always and case statements. The following is an example of a four-state finite state machine.



The 8-bit shift register is designed in the program. In the “Idle” state, judge whether the “shift_start” signal is high, if it is high, enter the “Start” state, delay 100 cycles in the “Start” state, enter the “Run” state, and perform shift processing. The “shift_stop” signal is valid, enter the “Stop” state, in the “Stop” state, clear the value of “q”, and then jump to the “Idle” state.

Mealy finite state machine, the output is not only related to the current state, but also related to the input signal, which is connected to the input signal in RTL.

```
module top
(
  input shift_start,
  input shift_stop,
  input rst,
  input clk,
  input d,
  output reg [7:0] q
);

parameter Idle  = 2'd0 ;    //Idle state
parameter Start = 2'd1 ;    //Start state
parameter Run   = 2'd2 ;    //Run state
parameter Stop  = 2'd3 ;    //Stop state

reg [1:0] state ;          //statement
reg [4:0] delay_cnt ;      //delay counter
```

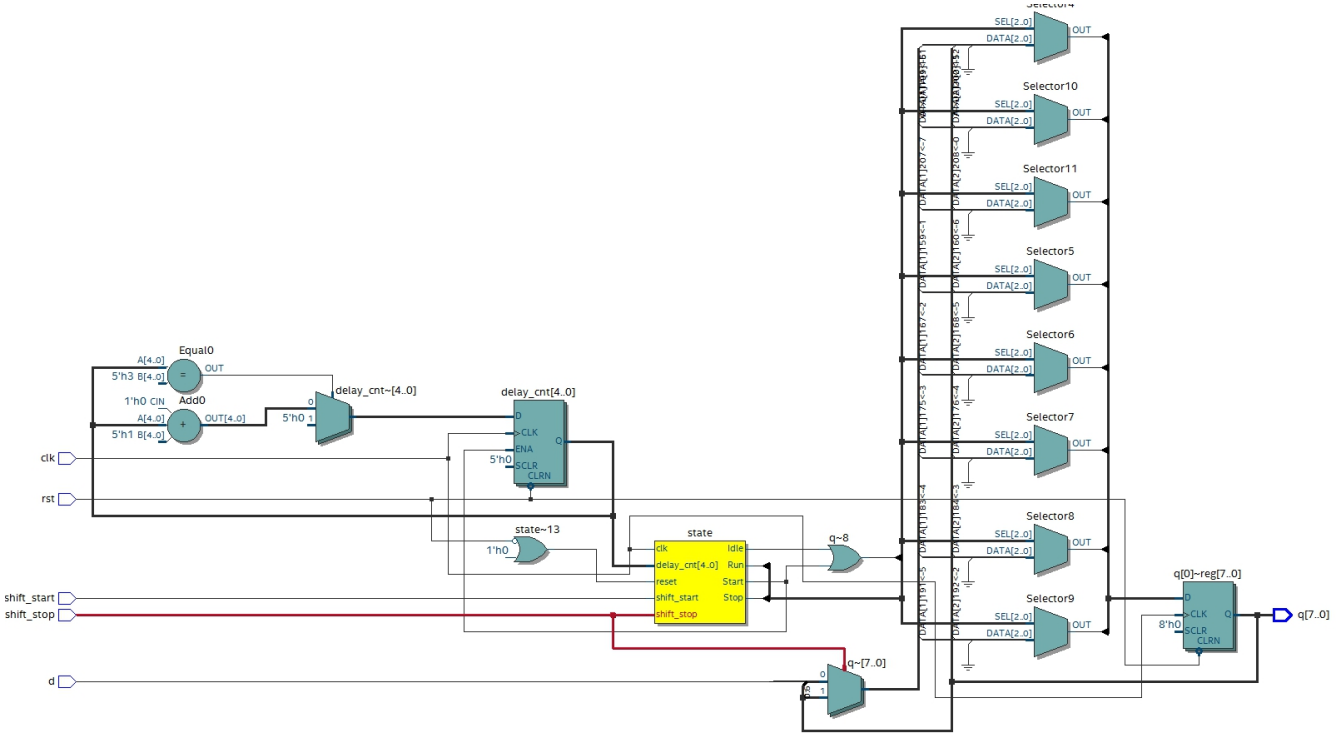
```
always @(posedge clk or negedge rst)
begin
  if (!rst)
  begin
   state <= Idle ;
   delay_cnt <= 0 ;
   q <= 0 ;
   end
  else
  case(state)
    Idle  : begin
            if (shift_start)
               state <= Start ;
    end
    Start : begin
             if (delay_cnt == 5'd99)
             begin
               delay_cnt <= 0 ;
               state <= Run ;
             end
             else
               delay_cnt <= delay_cnt + 1'b1 ;
           end
    Run   : begin
             if (shift_stop)
                state <= Stop ;
             else
                q <= {q[6:0], d} ;
           end
    Stop  : begin
             q <= 0 ;
             state <= Idle ;
          end
  default: state <= Idle ;
   endcase
end
endmodule
```

Moore finite state machine, the output is only related to the current state, has nothing to do with the input signal, the input signal only affects the change of the state, does not affect the output, such as the processing of delay_cnt and q, only related to the state.

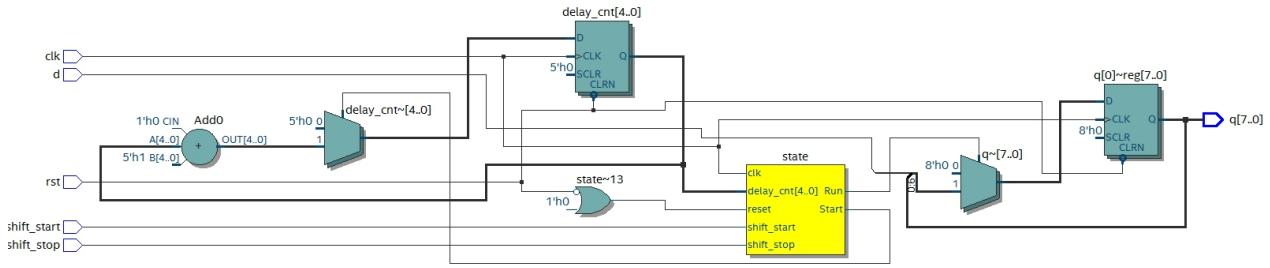
```
module top
(
  input shift_start,
  input shift_stop,
  input rst,
  input clk,
  input d,
  output reg [7:0] q
);

parameter Idle  = 2'd0 ;    //Idle state
parameter Start = 2'd1 ;    //Start state
parameter Run   = 2'd2 ;    //Run state
parameter Stop  = 2'd3 ;    //Stop state

reg [1:0] current_state ;          //statement
reg [1:0] next_state ;
reg [4:0] delay_cnt ;      //delay counter
//First part: statement transition
always @(posedge clk or negedge rst)
begin
  if (!rst)
   current_state <= Idle ;
  else
   current_state <= next_state ;
end
//Second part: combination logic, judge statement transition condition
always @(*)
begin
  case(current_state)
    Idle  : begin
              if (shift_start)
                 next_state <= Start ;
              else
                 next_state <= Idle ;
```

```
    end
    Start : begin
              if (delay_cnt == 5'd99)
                  next_state <= Run ;
              else
                  next_state <= Start ;
            end
    Run   : begin
              if (shift_stop)
                 next_state <= Stop ;
              else
                 next_state <= Run ;
            end
    Stop  :     next_state <= Idle ;
   default:     next_state <= Idle ;
  endcase
end
//Last part: output data
always @(posedge clk or negedge rst)
begin
  if (!rst)
    delay_cnt <= 0 ;
  else if (current_state == Start)
    delay_cnt <= delay_cnt + 1'b1 ;
  else
    delay_cnt <= 0 ;
end

always @(posedge clk or negedge rst)
begin
  if (!rst)
    q <= 0 ;
  else if (current_state == Run)
    q <= {q[6:0], d} ;
  else
    q <= 0 ;
end


endmodule
```

Two ways of writing are used in the above two programs. The first Mealy state machine uses a one-step writing method, using only one always statement. All state transitions, state transition conditions, and data output are all in In an always statement, the disadvantage is that if there are too many states, the entire program will become verbose. The second Moore state machine uses a three-stage writing method. The state transition uses an always statement. It is determined that the state transition condition is combinational logic. It uses an always statement and the data output is also a separate always statement. This makes it clearer to write, the state will not appear cumbersome in many situations.



Mealy finite state machine RTL diagram



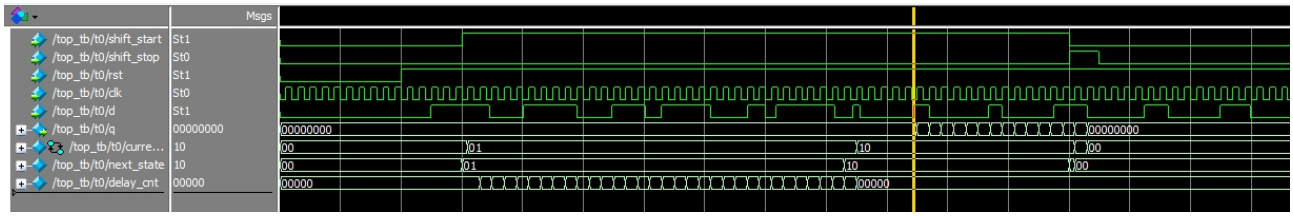
Moore finite state machine RTL diagram

The incentive documents are as follows:

```
`timescale 1 ns/1 ns
module top_tb() ;
reg shift_start ;
reg shift_stop ;
reg rst ;
reg clk ;
reg d ;
wire [7:0] q ;

initial
begin
  rst = 0 ;
  clk = 0 ;
  d = 0 ;
  #200 rst = 1 ;
  forever
  begin
    #({$random}%100)
    d = ~d ;
  end
end

initial
begin
  shift_start = 0 ;
  shift_stop = 0 ;
  #300 shift_start = 1 ;
  #1000 shift_start = 0 ;
```

```
        shift_stop  = 1 ;
    #50 shift_stop = 0 ;
end

always #10 clk = ~clk ;

top  t0
(
  .shift_start(shift_start),
  .shift_stop(shift_stop),
  .rst(rst),
  .clk(clk),
  .d(d),
  .q(q)
);
endmodule
```

The simulation results are as follows:



Part 3.7: Summary

This document introduces the commonly used modules in combinatorial logic and sequential logic. Among them, the finite state machine is more complicated, but it is often used. I hope everyone can understand it deeply, use it more in the code, and think more, which is conducive to rapid improvement.

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

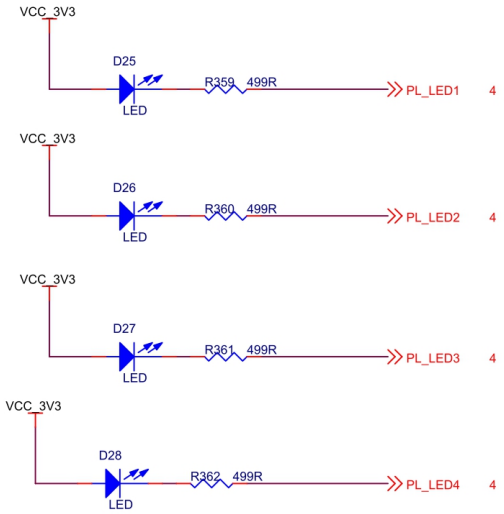
Experiment Vivado project "led"

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.

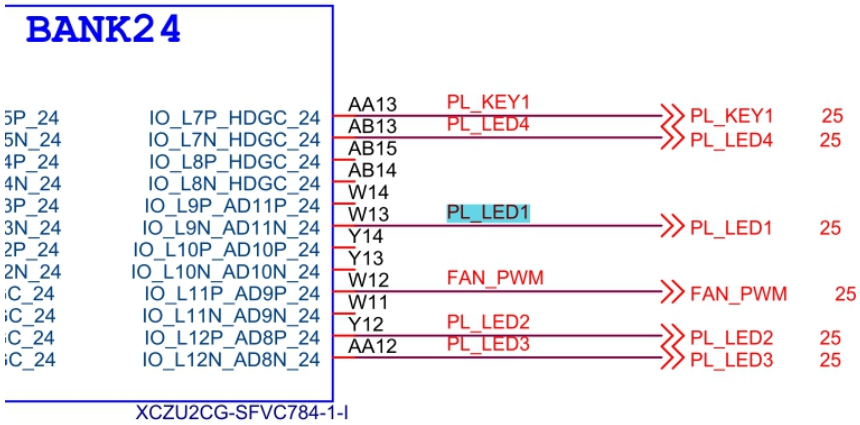
In this routine, what we need to do is the LED light control experiment. The LEDs on the development board are flipped once every second to achieve the control of on, off, on, off. Will control the LED lights, other peripherals will slow.

Part 4.1: LED Hardware Introduction

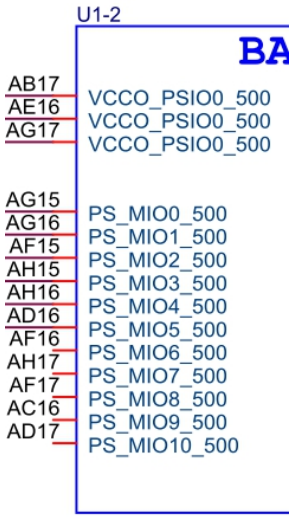
1) The 4 red user LEDs are connected to the PL part of the FPGA development board. This 1 LED is completely controlled by PL. If PL_LED1 is high level, the LED will be on, otherwise it will be off



2) We can determine the binding relationship between the LED and PL pins according to the wiring relationship of the schematic.



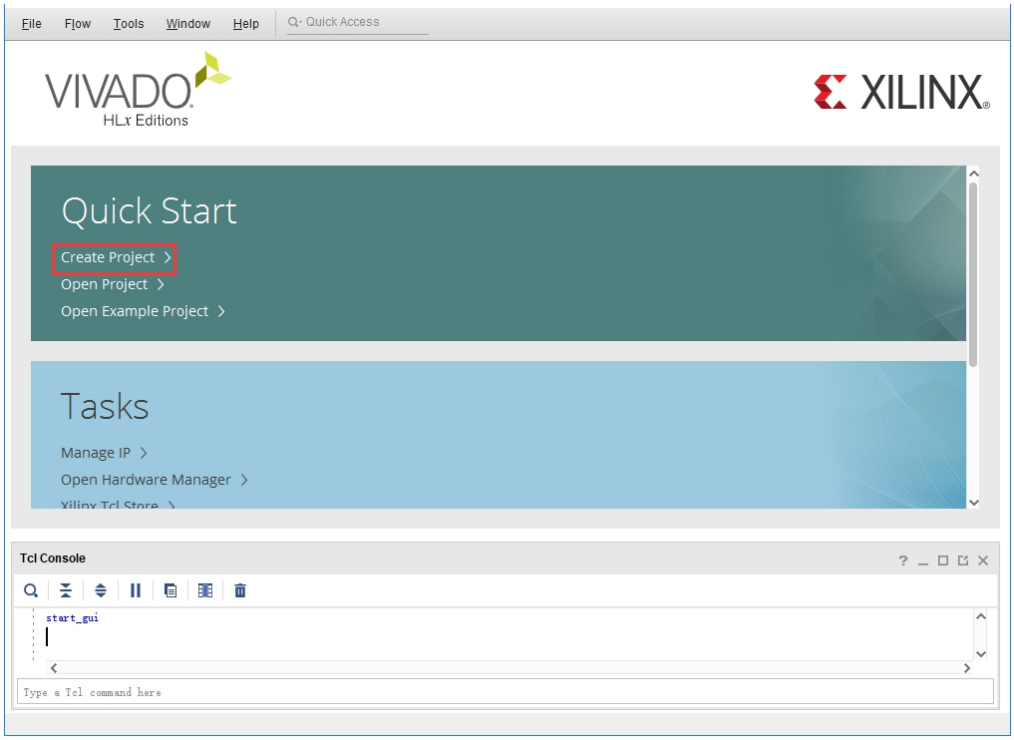
3) The IOs beginning with PS_MIO in the schematic are all PS-side IOs. They do not need to be bound or bound.



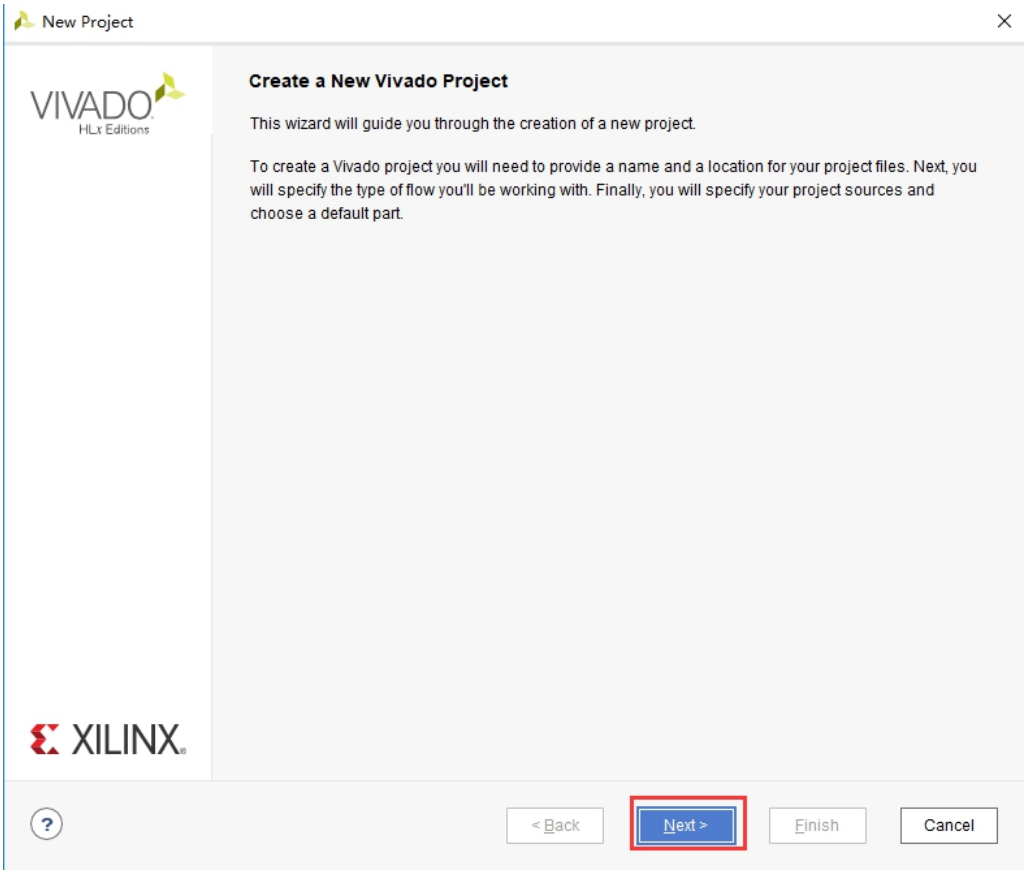
Part 4.2: Create a Vivado project

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

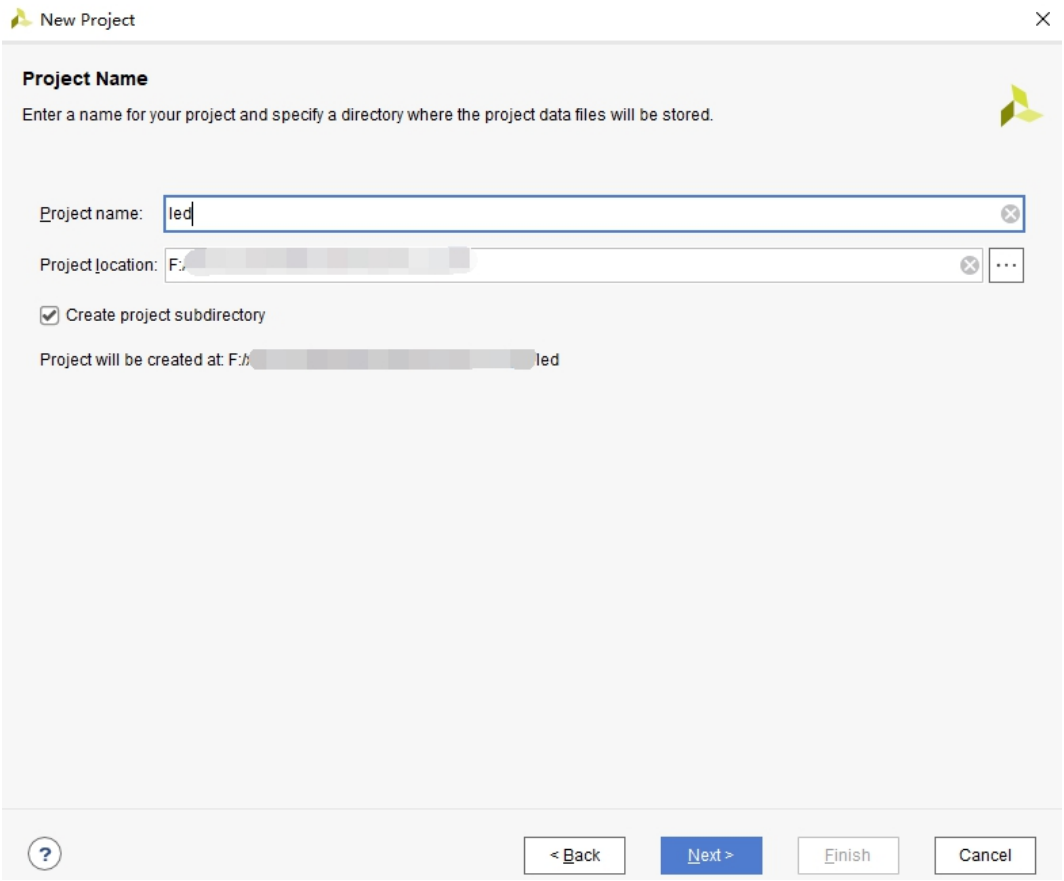
2) Create a new project by clicking on "Create New Project" in the Vivado development environment.



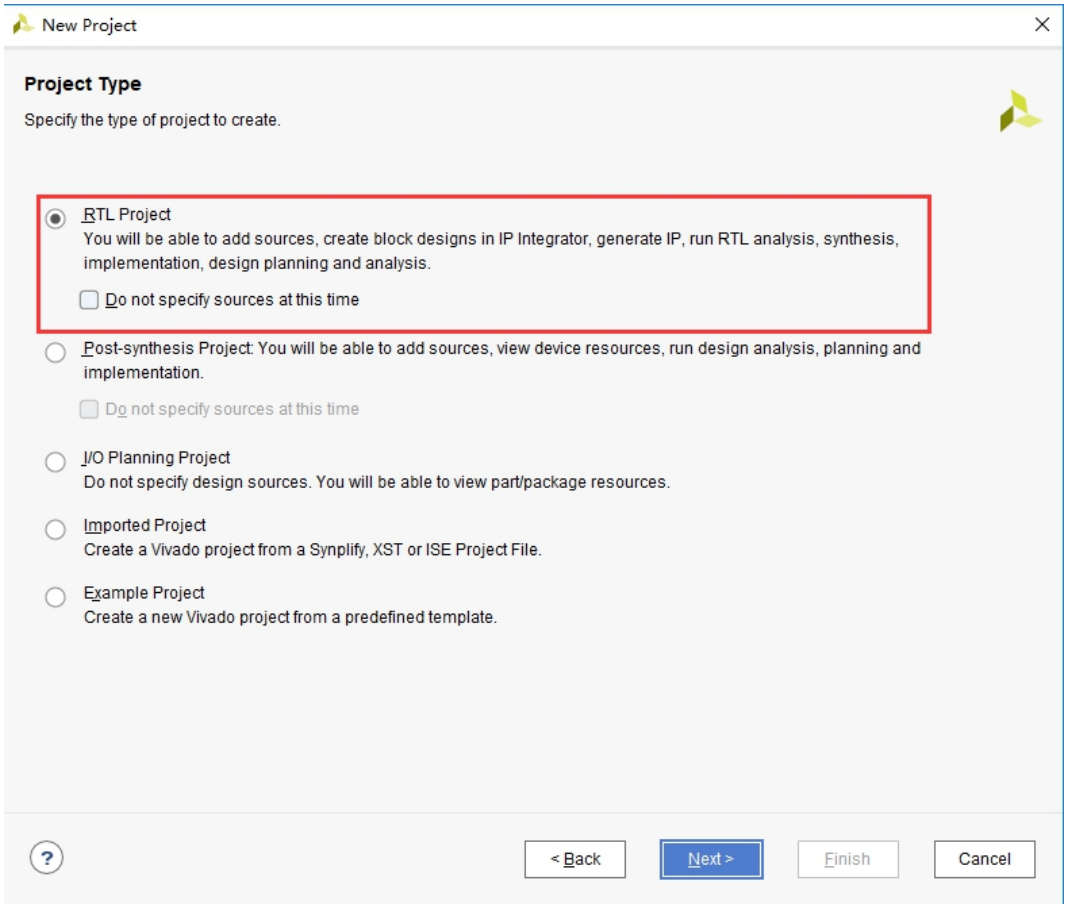
3) A wizard to create a new project pops up, click "Next"



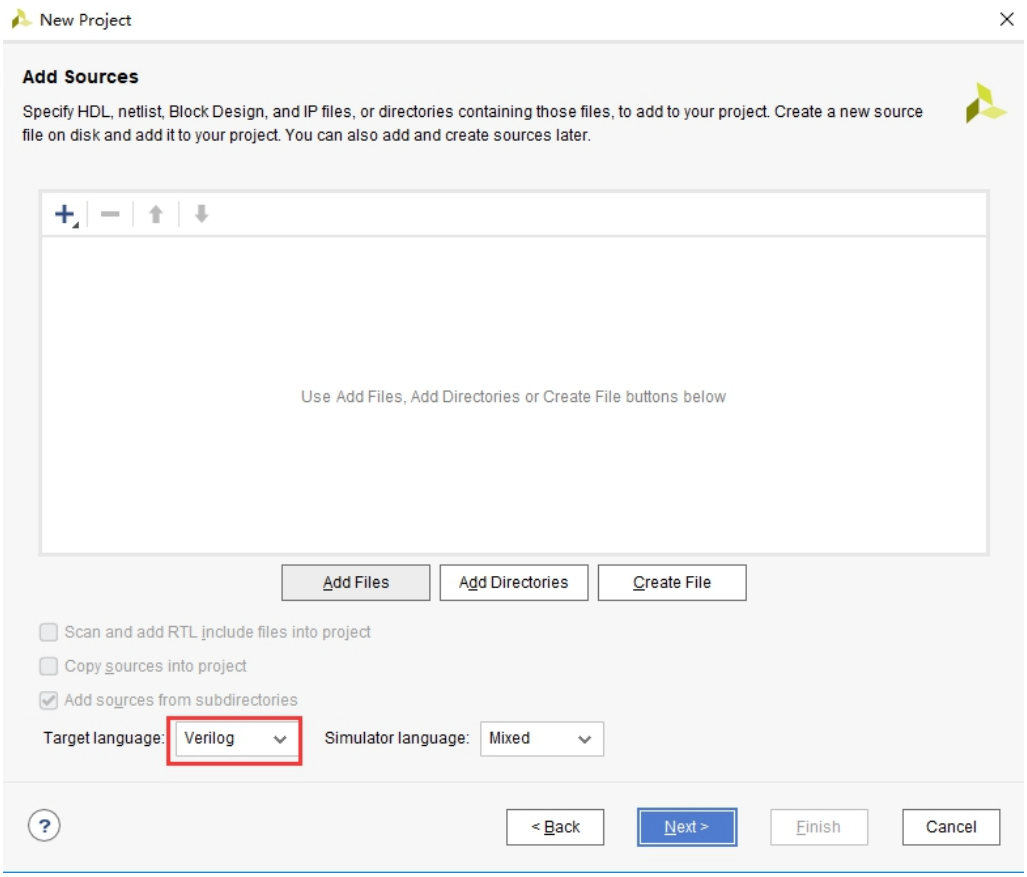
4) In the pop-up dialog box, enter the project name and the directory where the project is stored. We will take a led project name here. It should be noted that the project path "Project location" cannot have Chinese spaces, and the path name cannot be too long.



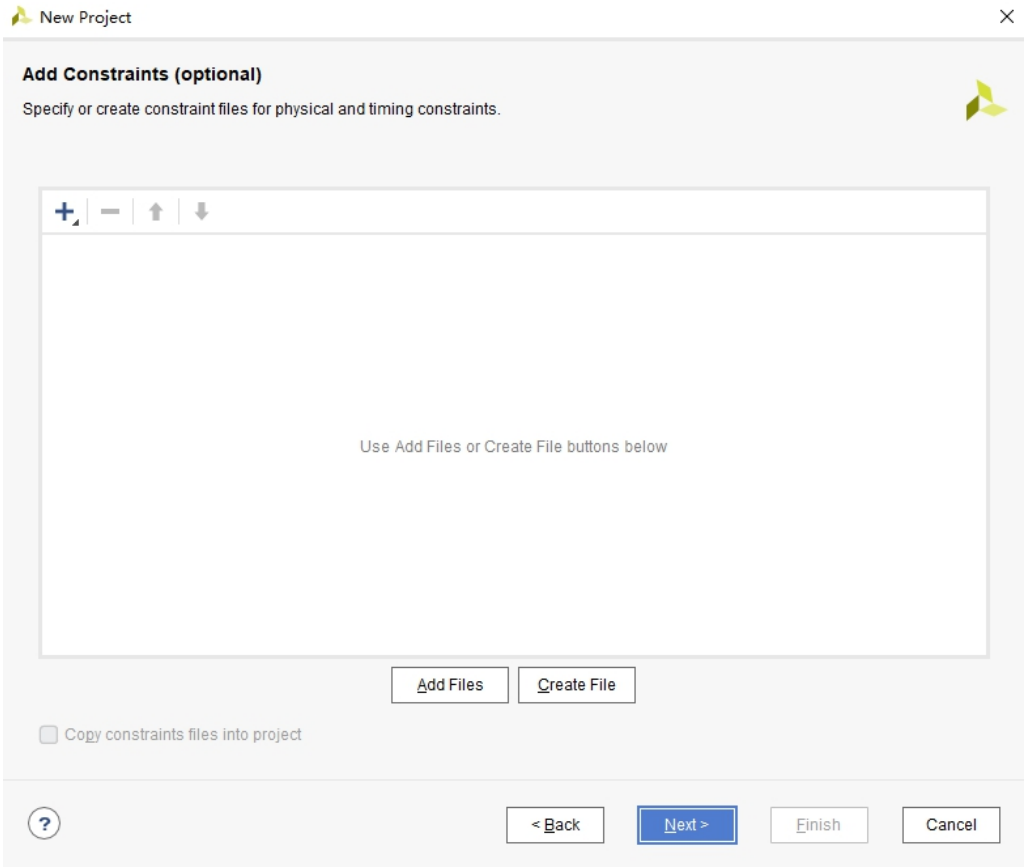
5) Select "RTL Project" in the project type



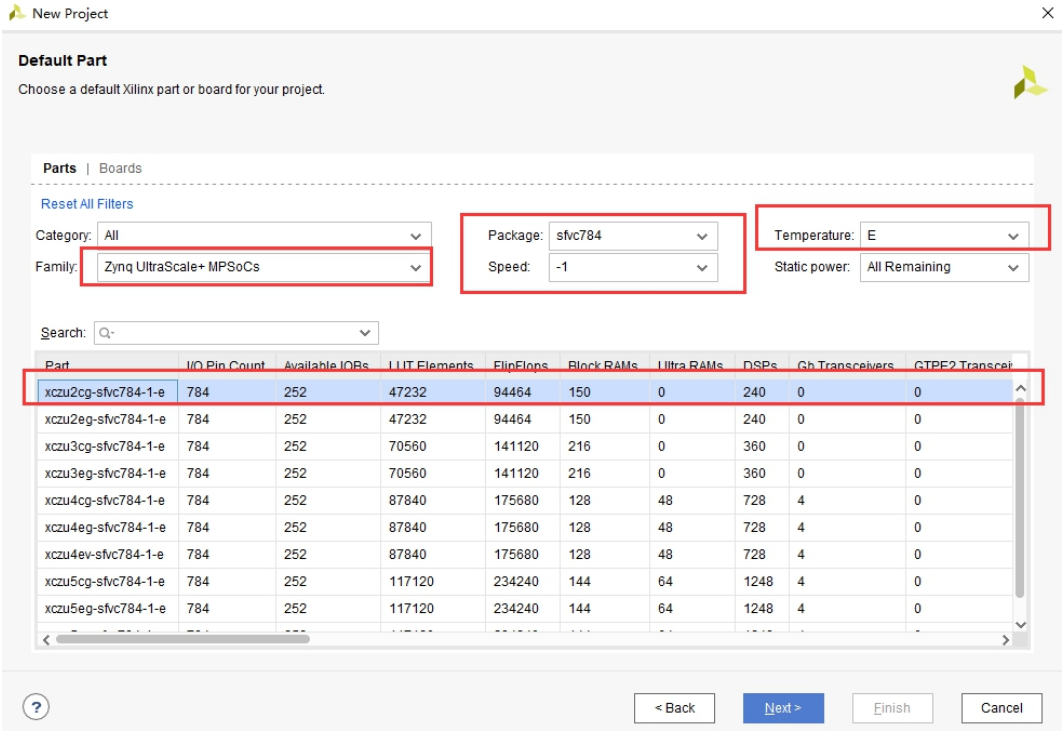
6) The target language "Target language" selects "Verilog". Although Verilog is selected, VHDL can also be used to support multi-language mixed programming.



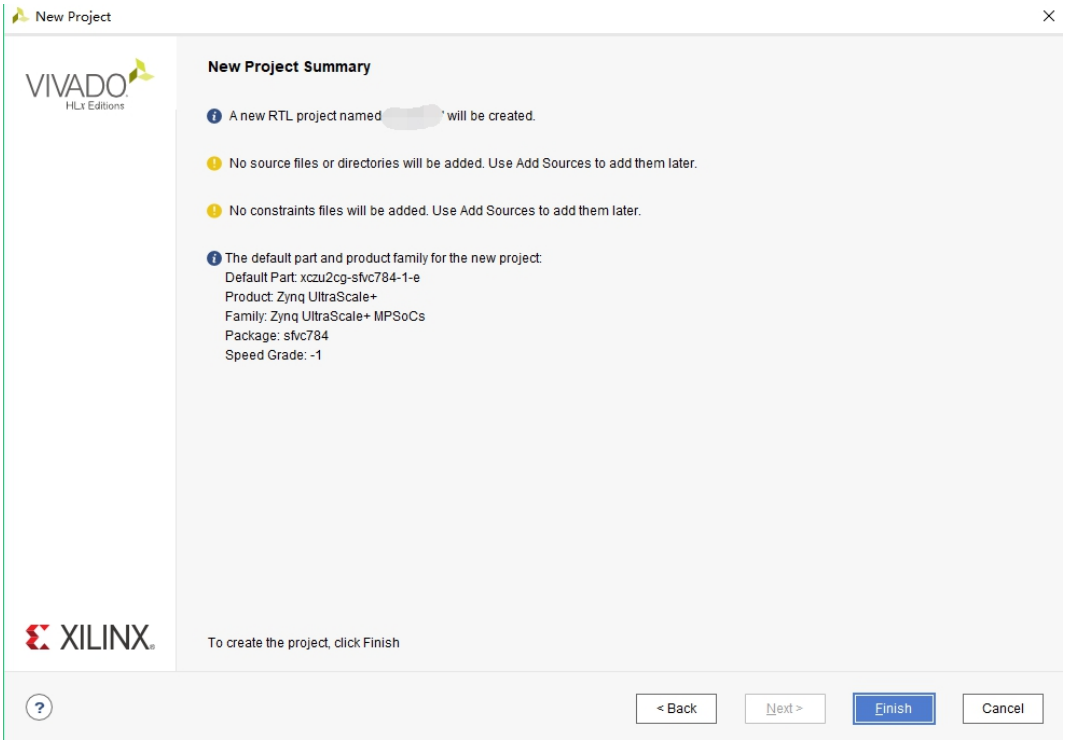
7) Click "Next" without adding any files



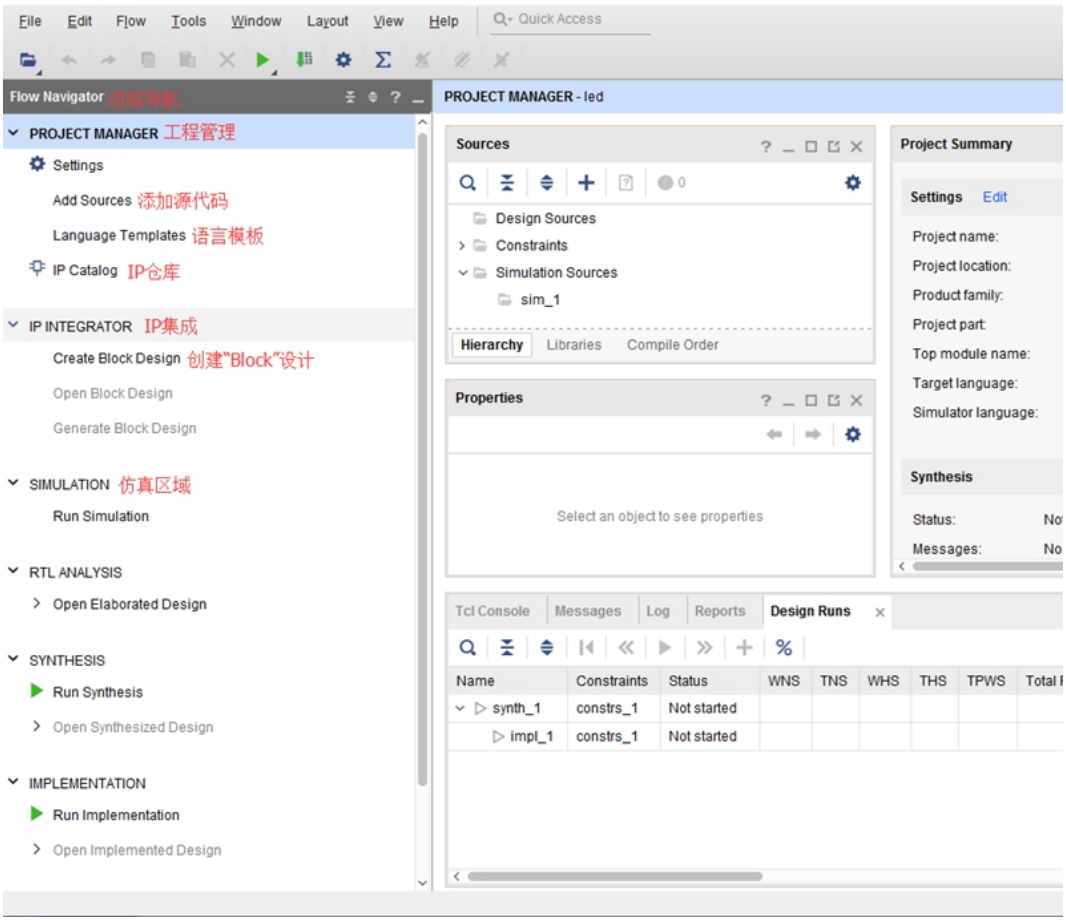
8) In the "Part" option, select "Zynq UltraScale+ MPSoCs" for the device "Family", select "sfvc784" for the package type "Package", select "Speed" -1", and select "E" for Temperature to reduce the selection range. In the drop-down list, select "xczu2cg-sfvc784-1-e", "-1" indicates the speed level, the larger the number, the better the performance, and the higher speed chip is backward compatible with the lower speed chip.



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

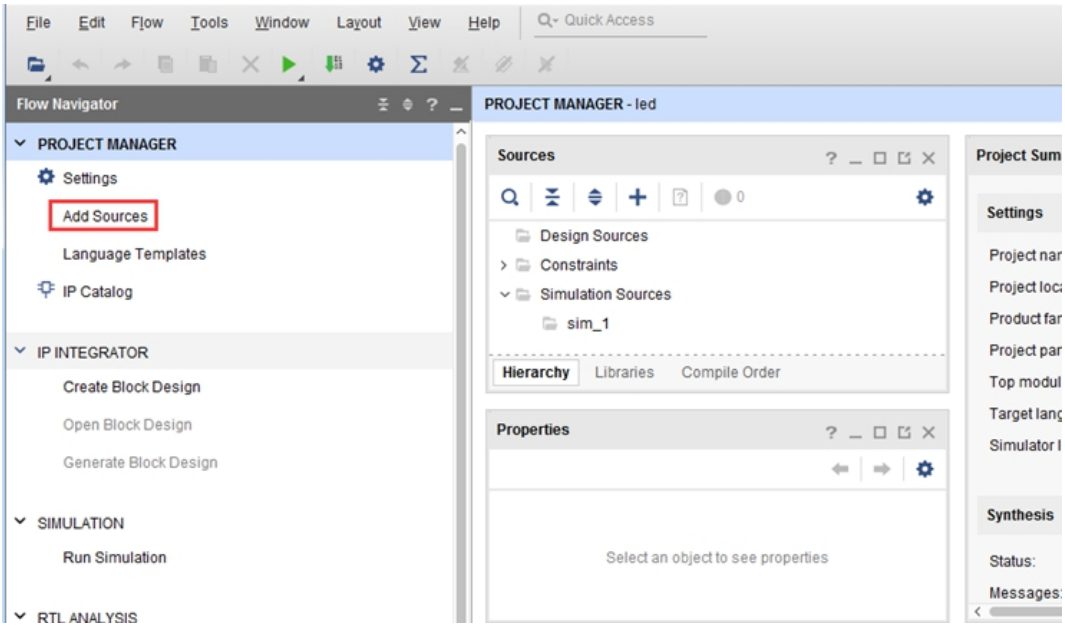


10)Vivado software interface

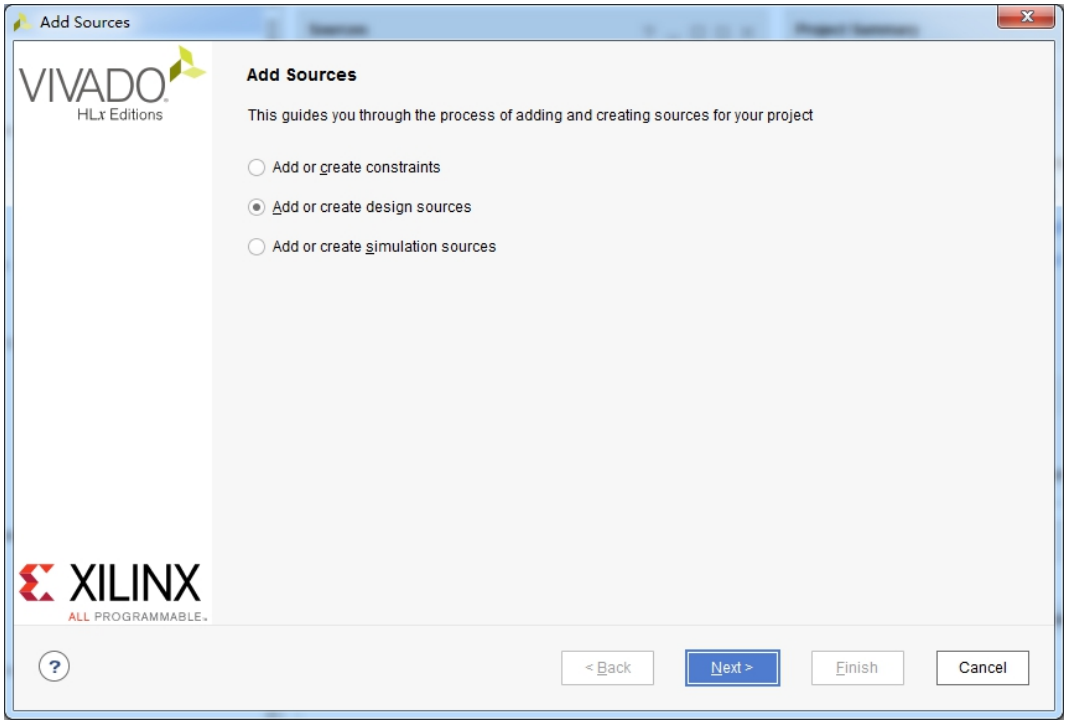


Part 4.3: Create a Verilog HDL file to illuminate the LED

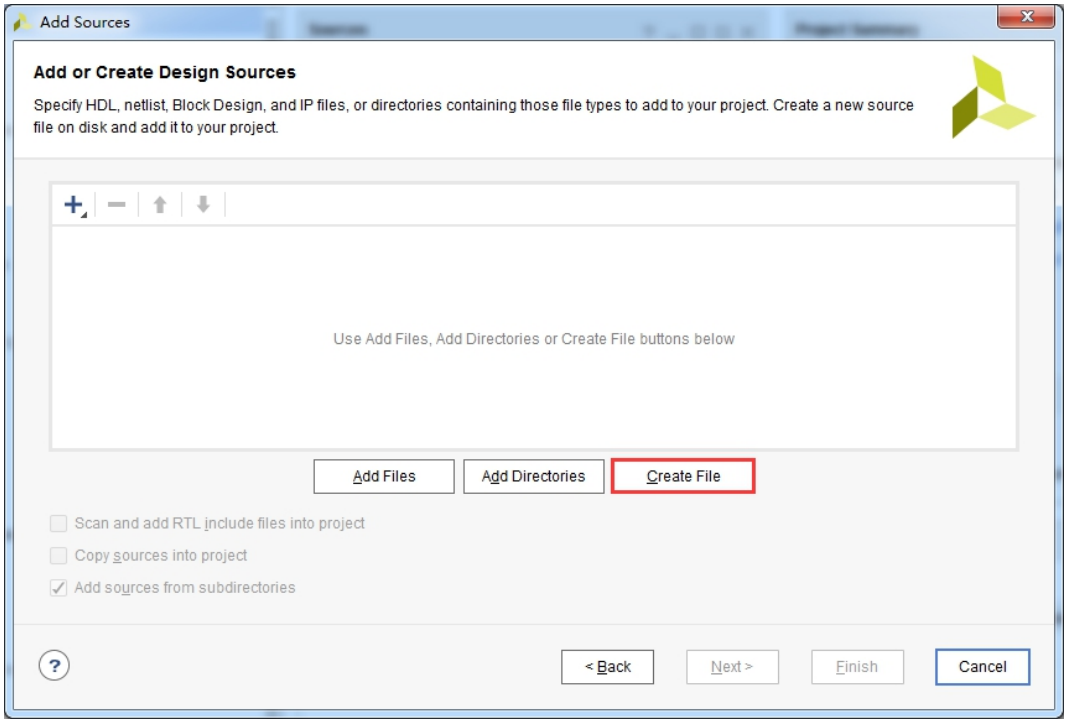
1) Click the Add Sources icon under Project Manager (or use the shortcut Alt+A)



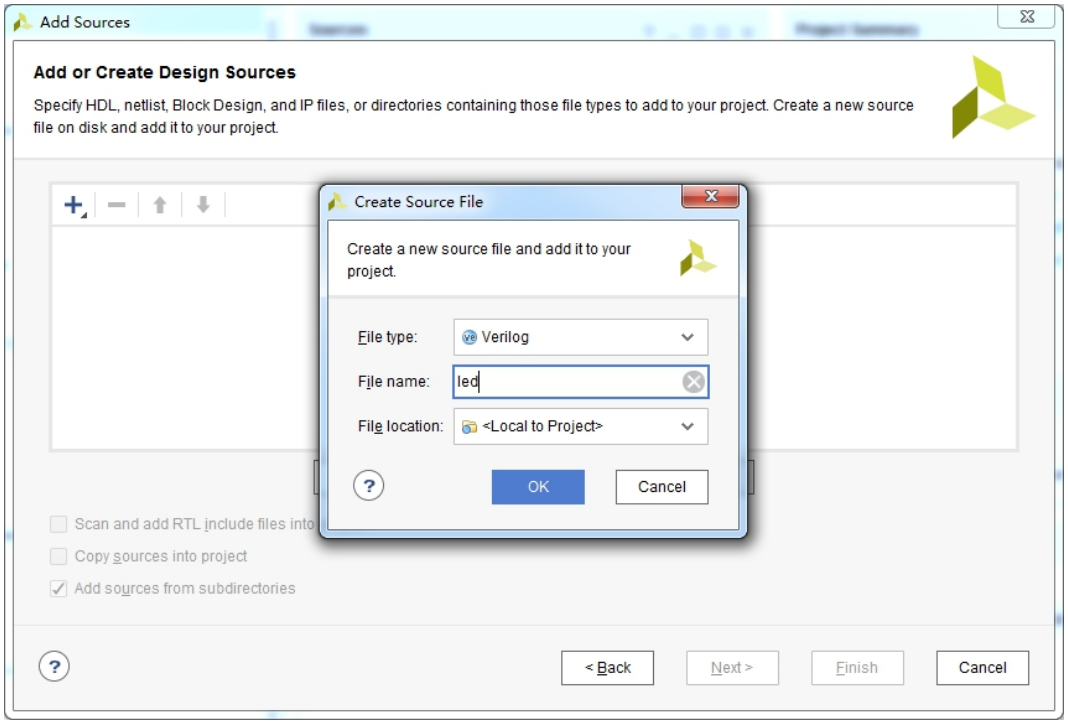
2) Add or create design source file "Add or create design sources", click "Next"



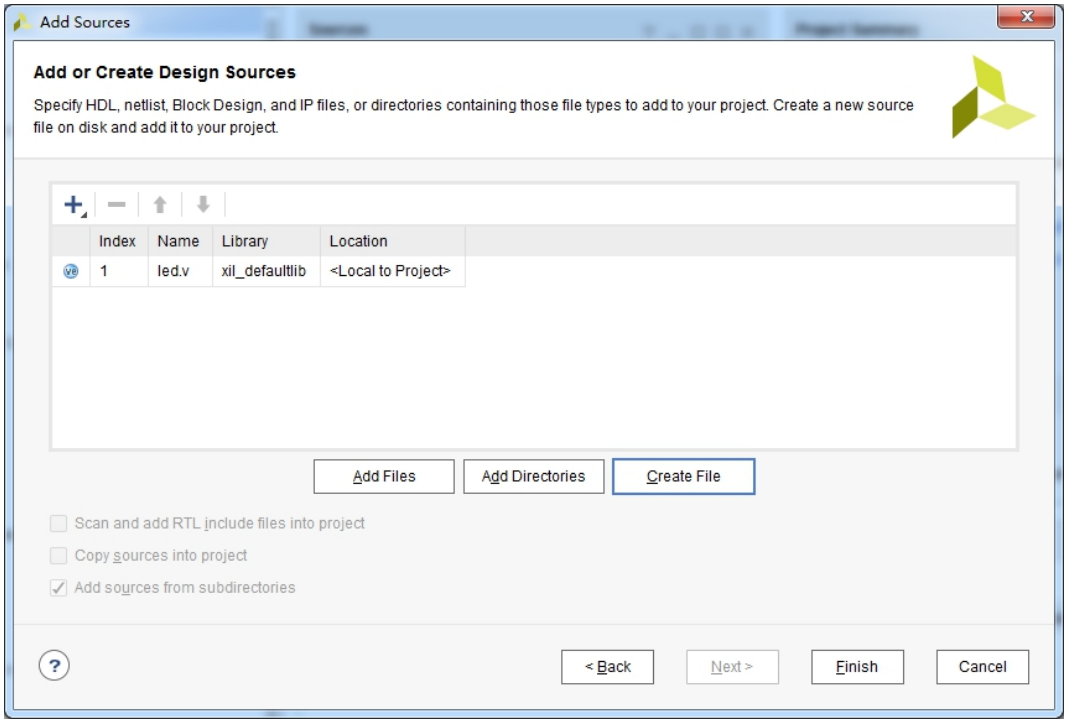
3) Choose to create the file "Create File"



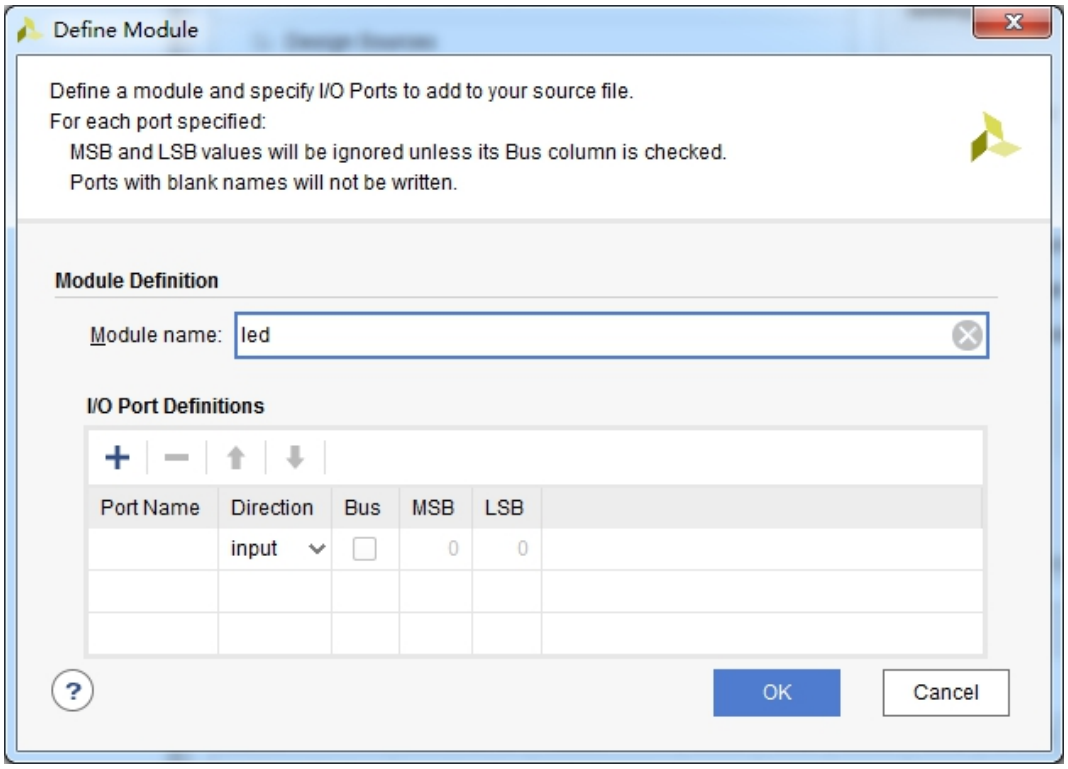
4) The file name "File name" is set to "led", click "OK"



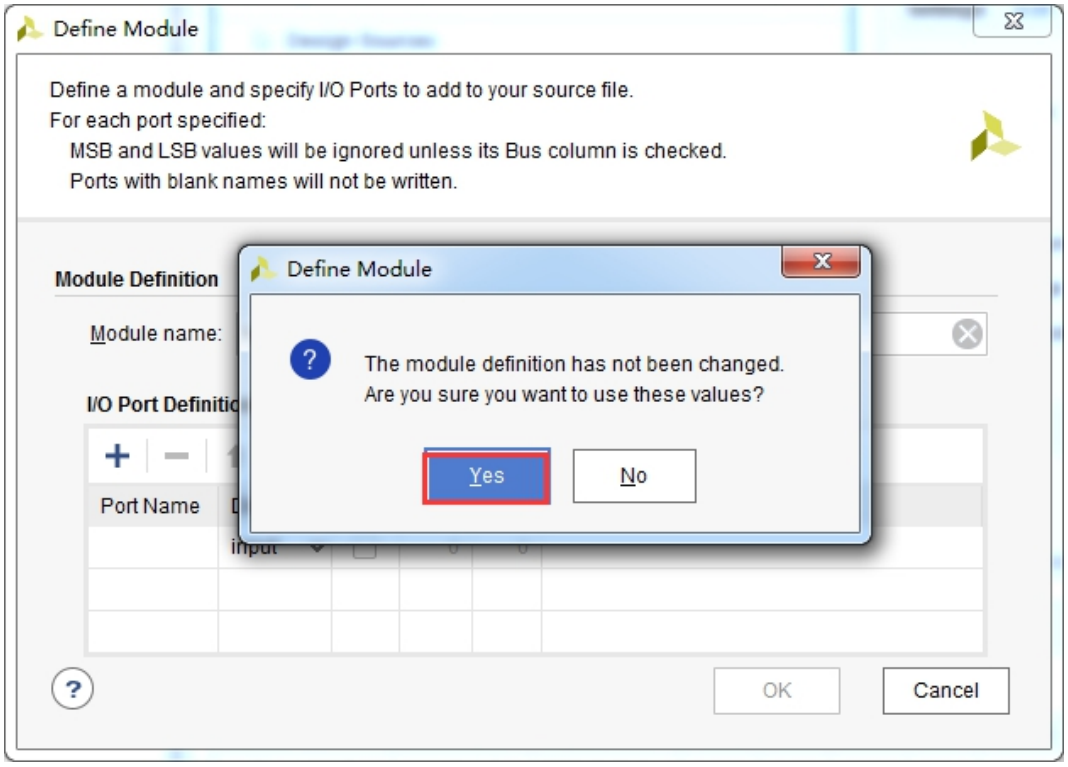
5) Click "Finish" to complete the "led.v" file addition.



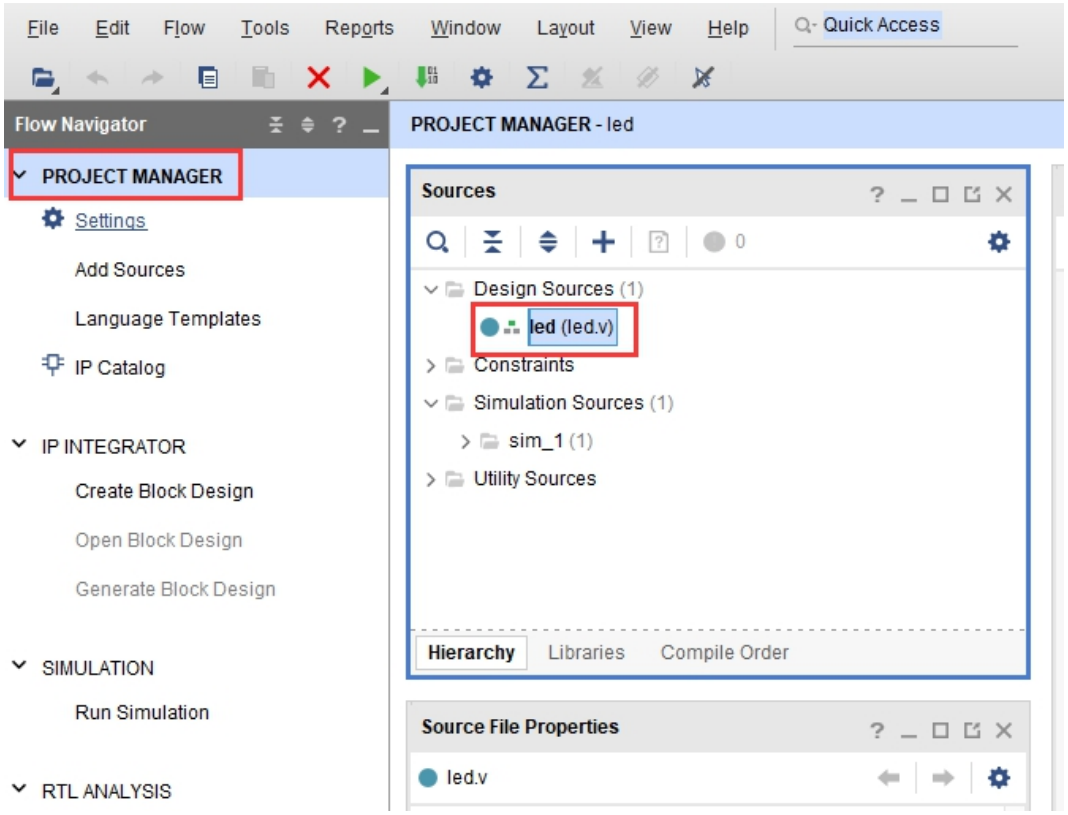
6) In the pop-up module definition "Define Module", you can specify the module name "Module name" of the "led.v" file, the default is not changed to "led", you can also specify some ports, not specified here, click "OK" ".



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. Since the input clock is a 200MHz differential clock, IBUFDS primitives need to be added to connect the differential signal. The compiled code is as follows:

```
module led(
    input sys_clk,
    input rst_n,
    output reg [3:0] led
    );
reg[31:0] timer_cnt;
always@(posedge sys_clk or negedge rst_n)
begin
    if (!rst_n)
    begin
        led <= 4'd0 ;
        timer_cnt <= 32'd0 ;
    end
    else if(timer_cnt >= 32'd24_999_999)
    begin
        led <= ~led;
        timer_cnt <= 32'd0;
    end
    else
    begin
        led <= led;
        timer_cnt <= timer_cnt + 32'd1;
    end

end
    endmodule
```

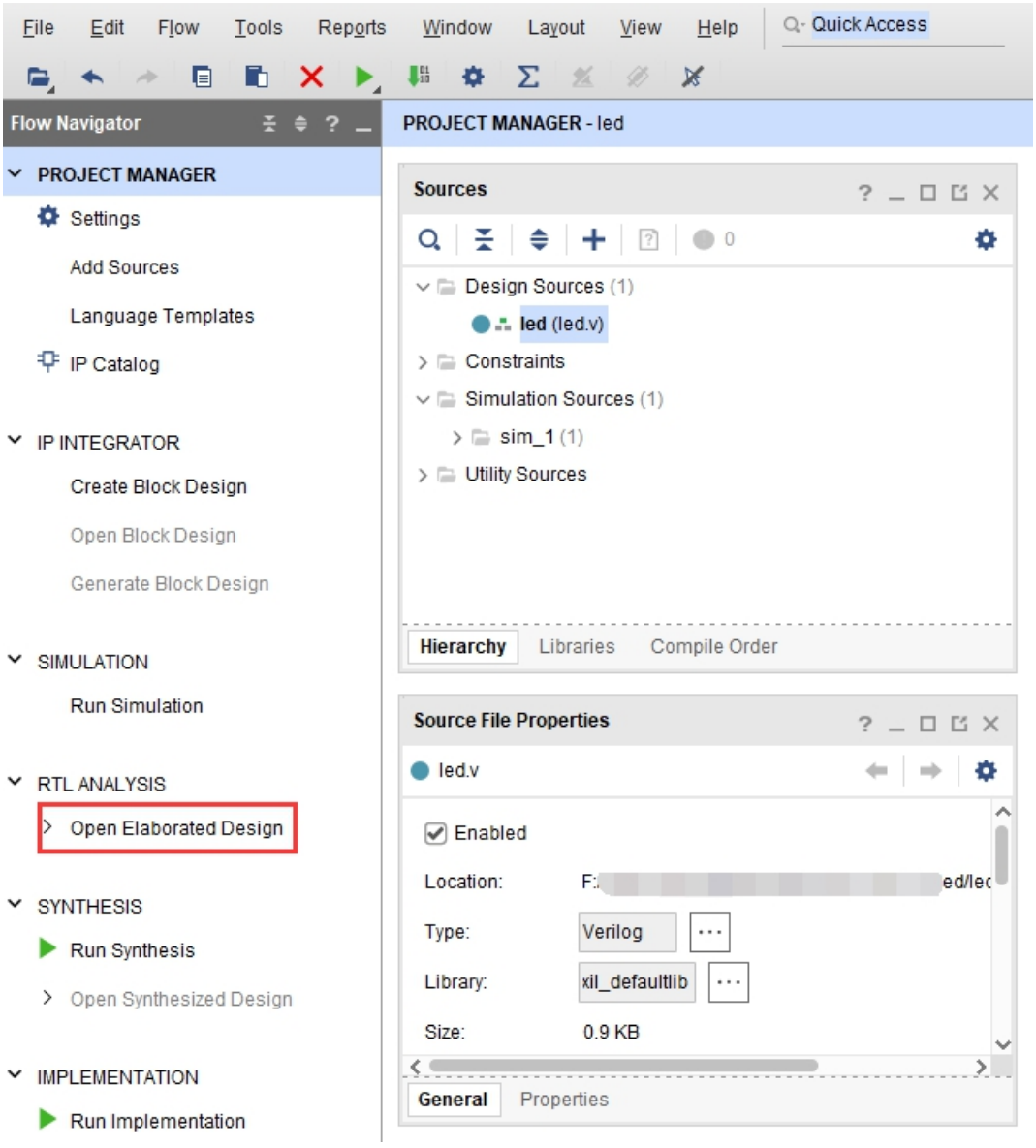
10) Write the code and save it, click on the menu "File -> Save All Files"

Part 4.4: Add Pin Constraint

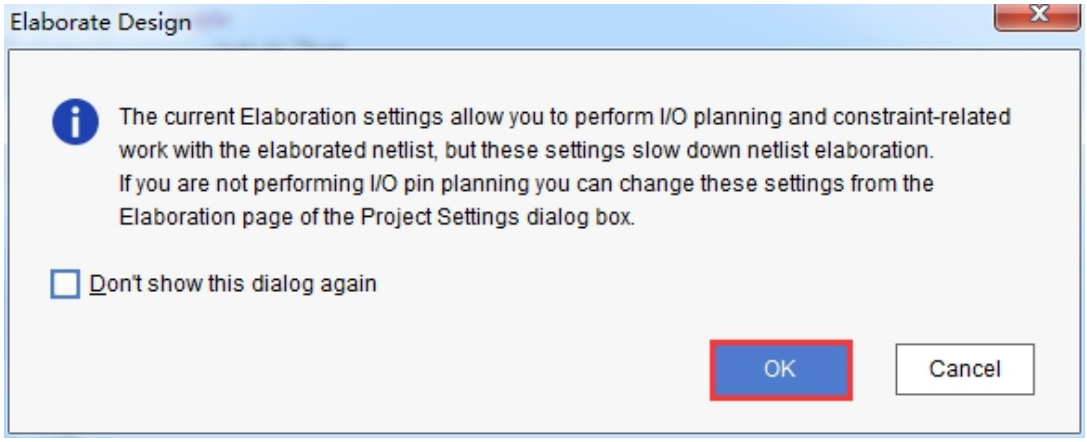
The constraint file format used by Vivado is xdc file. The xdc file

mainly completes 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 FPGA.

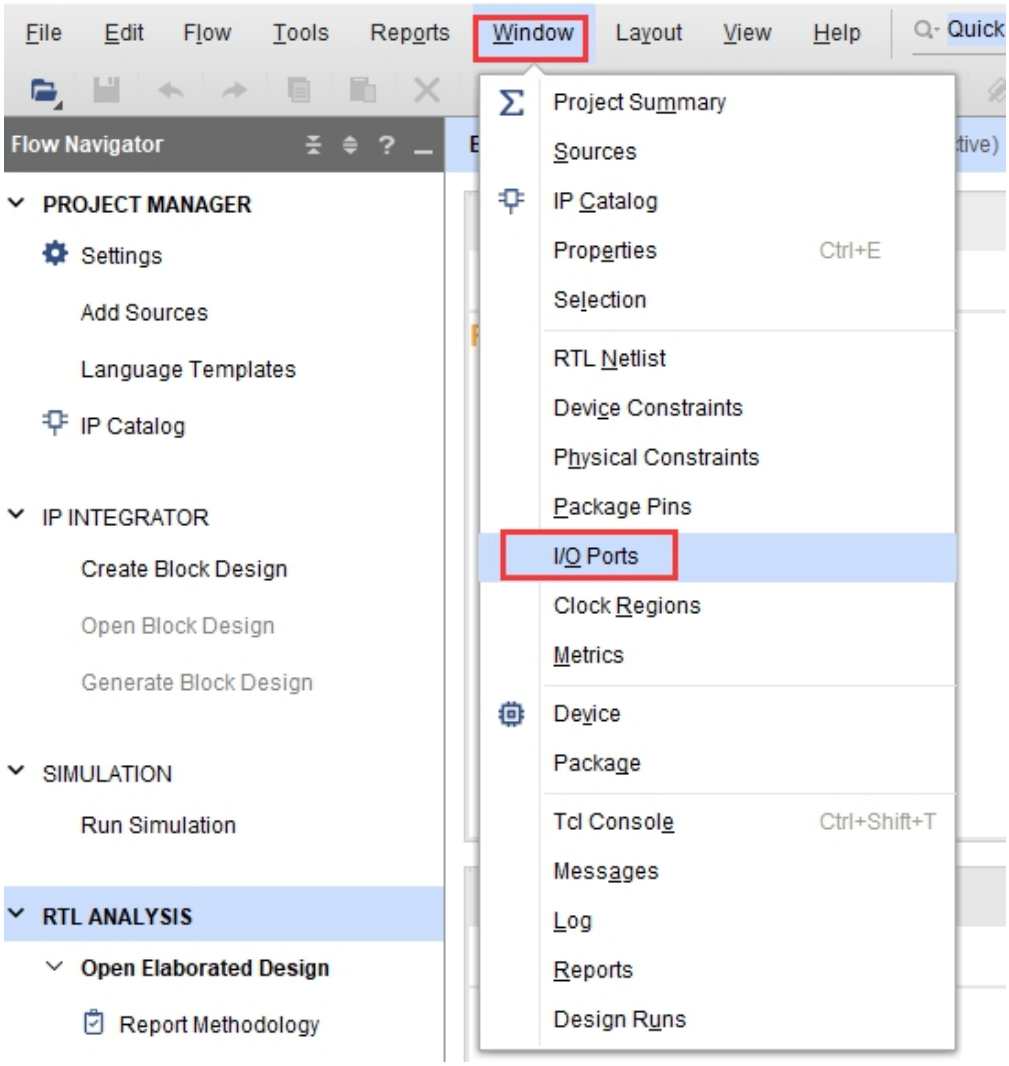
1) Click “Open Elaborated Design”



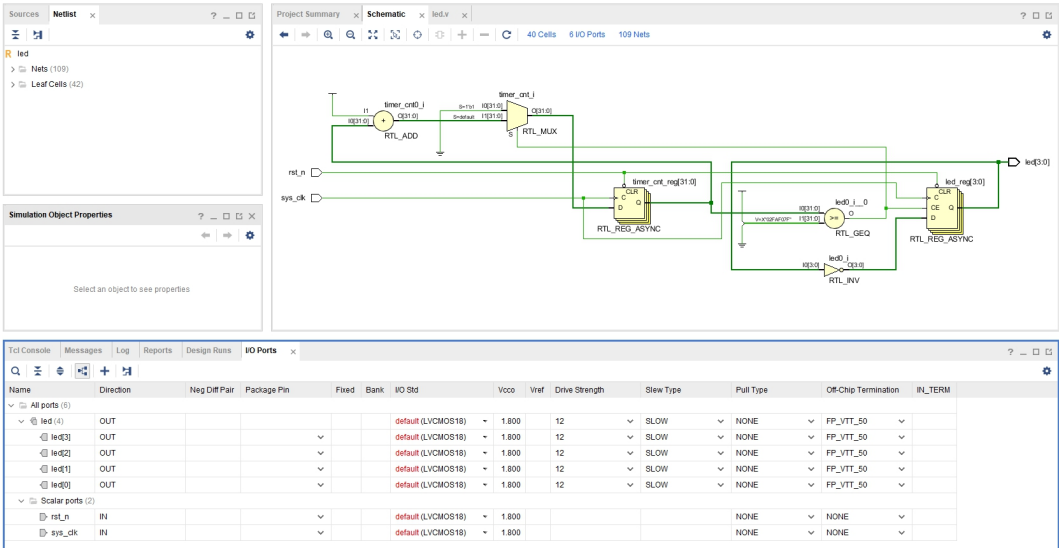
2) Click the "OK" button in the pop-up window



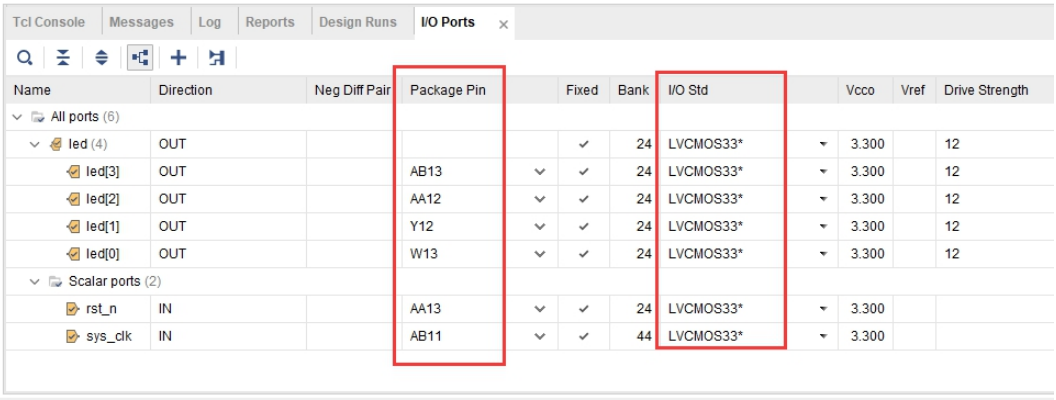
3) Select “Window -> I/O Ports”in the Menu



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

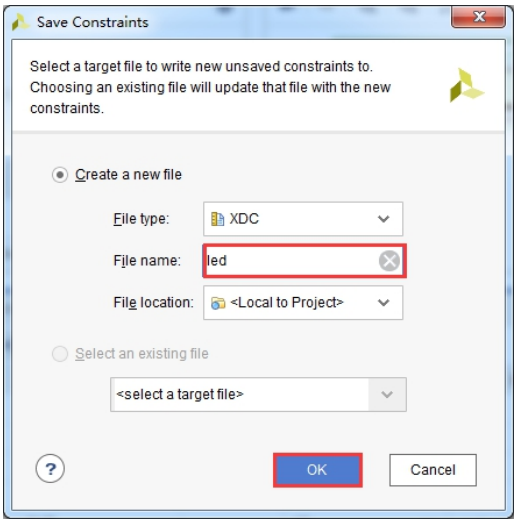


5) Bind the reset signal “rst_n” to the button on the “PL”side, assign the pin and level to the LED and clock, and click the save icon when finished.

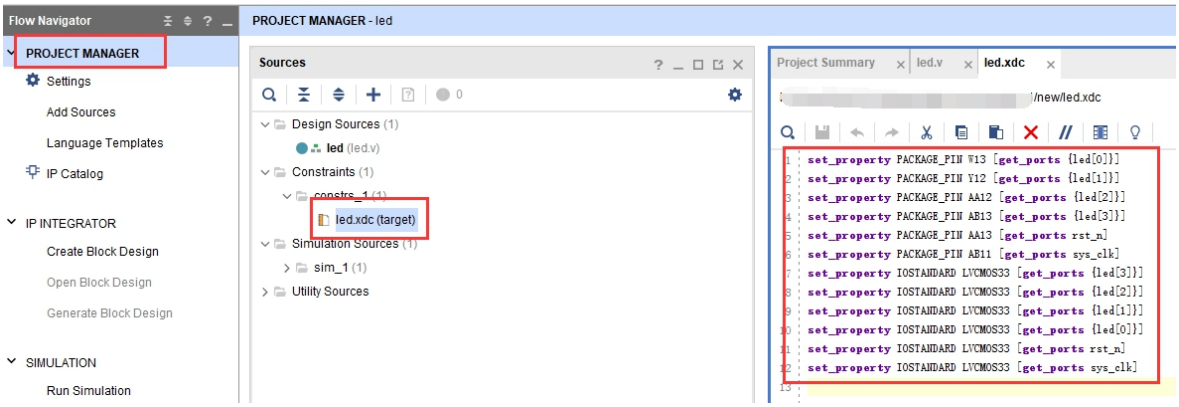


Name	Direction	Neg Diff Pair	Package Pin		Fixed	Bank	I/O Std		Vcco	Vref	Drive Strength
All ports (6)											
led (4)	OUT				✓	24	LVCMOS33*		3.300		12
led[3]	OUT		AB13		✓	24	LVCMOS33*		3.300		12
led[2]	OUT		AA12		✓	24	LVCMOS33*		3.300		12
led[1]	OUT		Y12		✓	24	LVCMOS33*		3.300		12
led[0]	OUT		W13		✓	24	LVCMOS33*		3.300		12
Scalar ports (2)											
rst_n	IN		AA13		✓	24	LVCMOS33*		3.300		
sys_clk	IN		AB11		✓	44	LVCMOS33*		3.300		

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”]

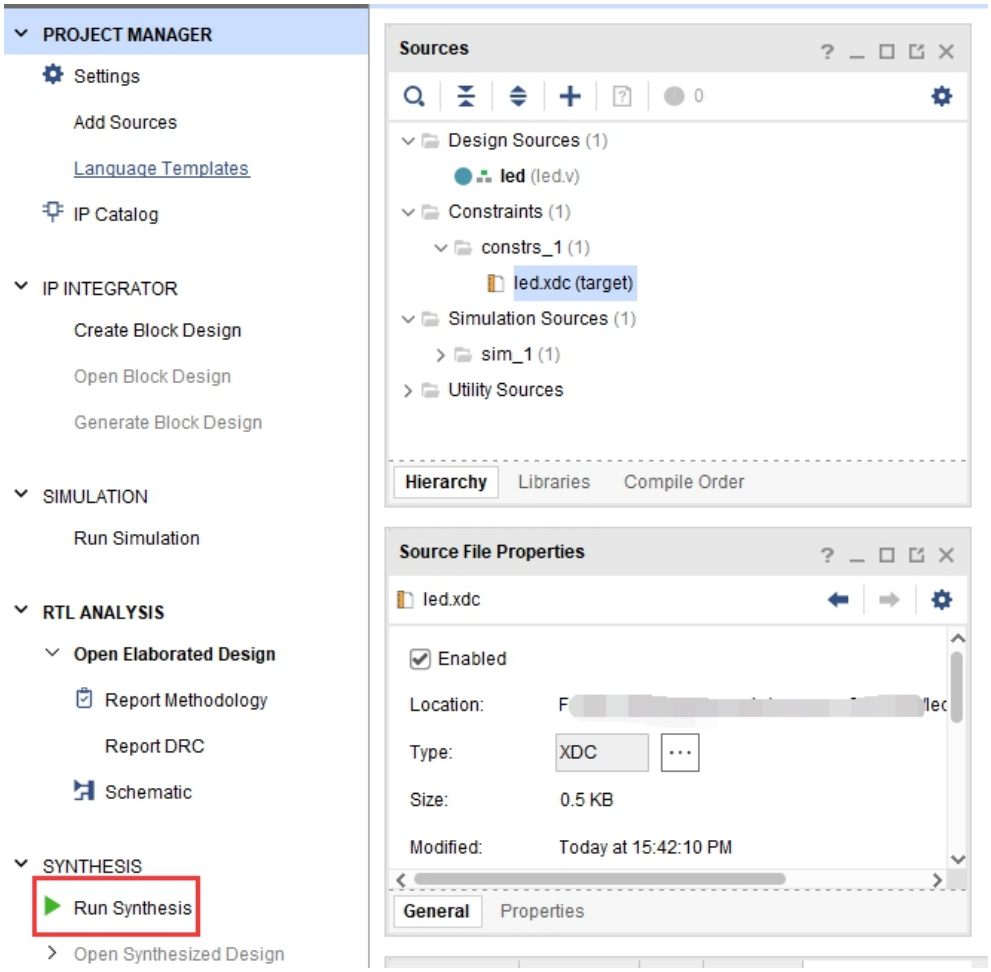
Pay attention to the case of the text. If the port name is an array, enclose it in { }. The port name must be the same as the name in the source code, and the port name cannot be the same as the keyword.

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***

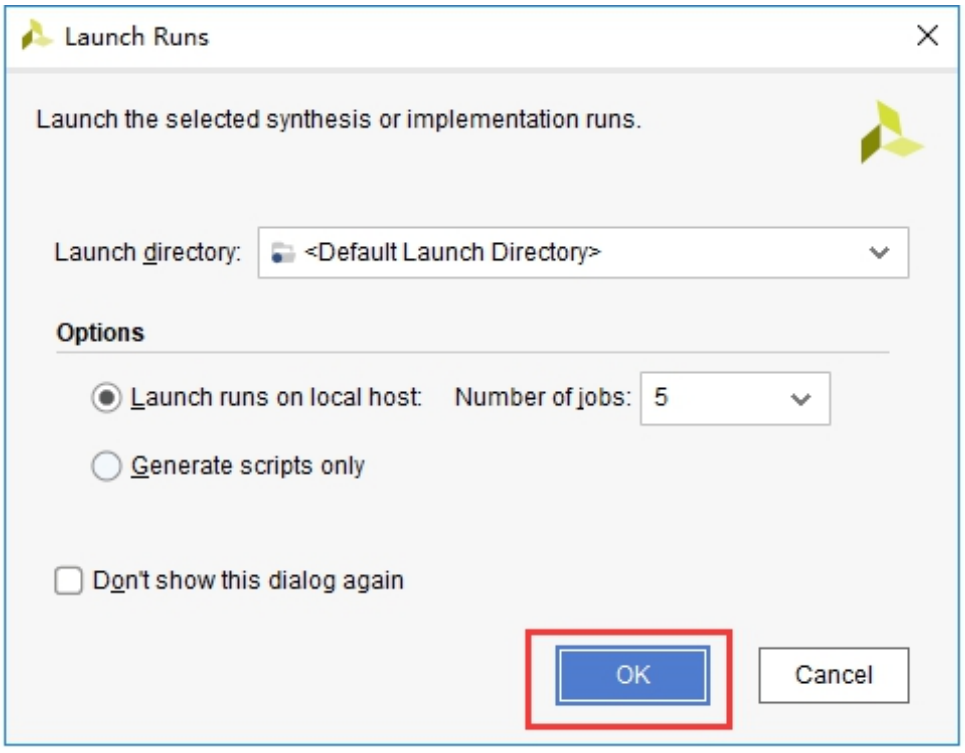
Part 4.5 Add timing constraints

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

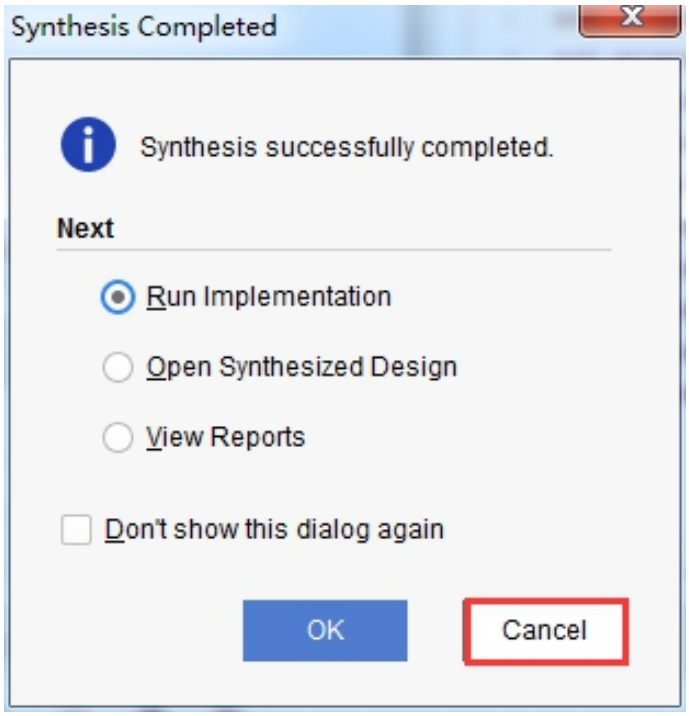
1) Click “Run Synthesis” to start to Synthesis



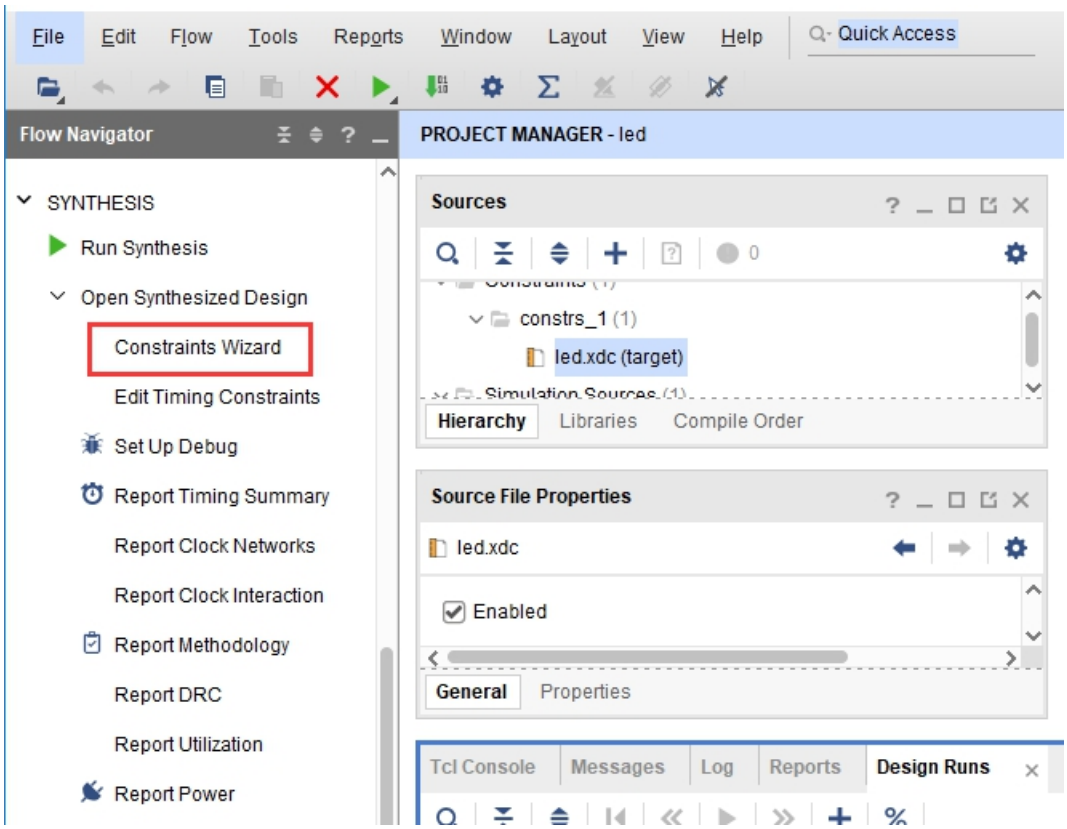
2) Pop up the dialog box and click "OK"



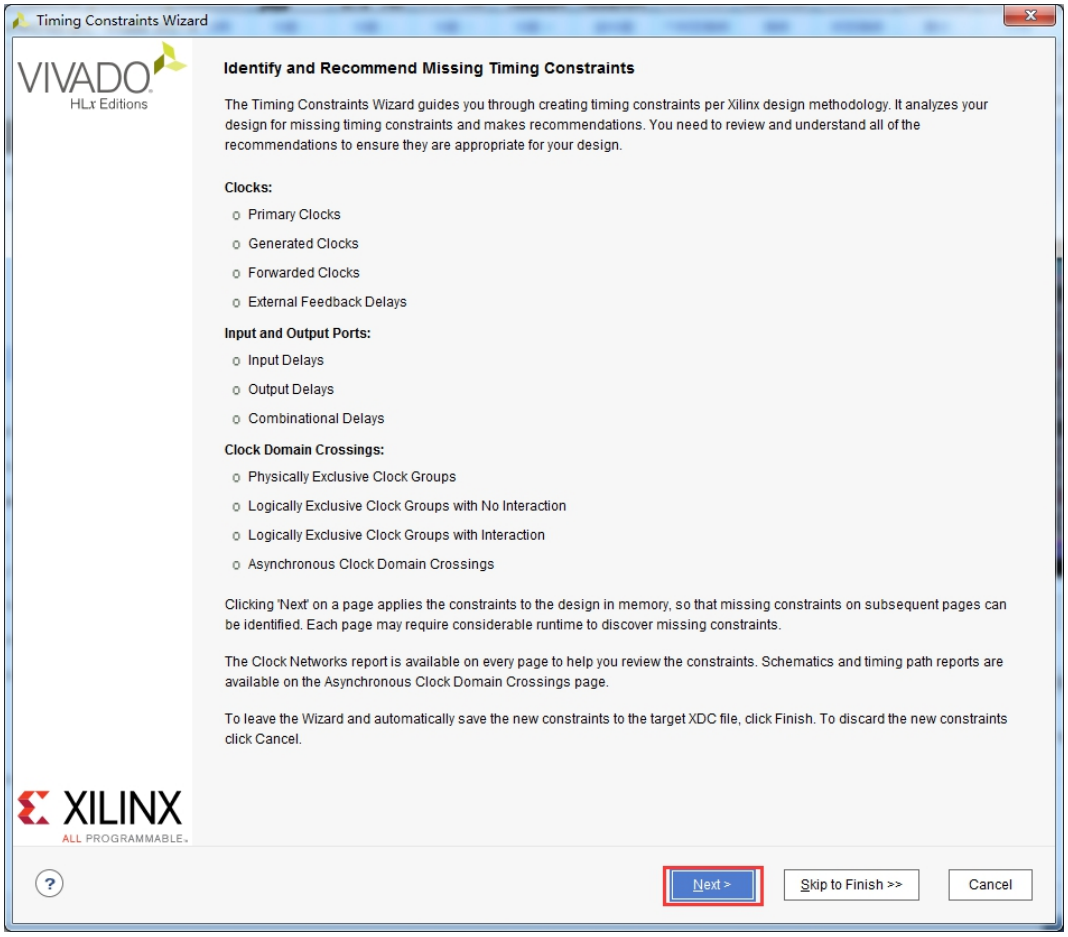
3) Click "Cancel" after the Synthesis successful Completed



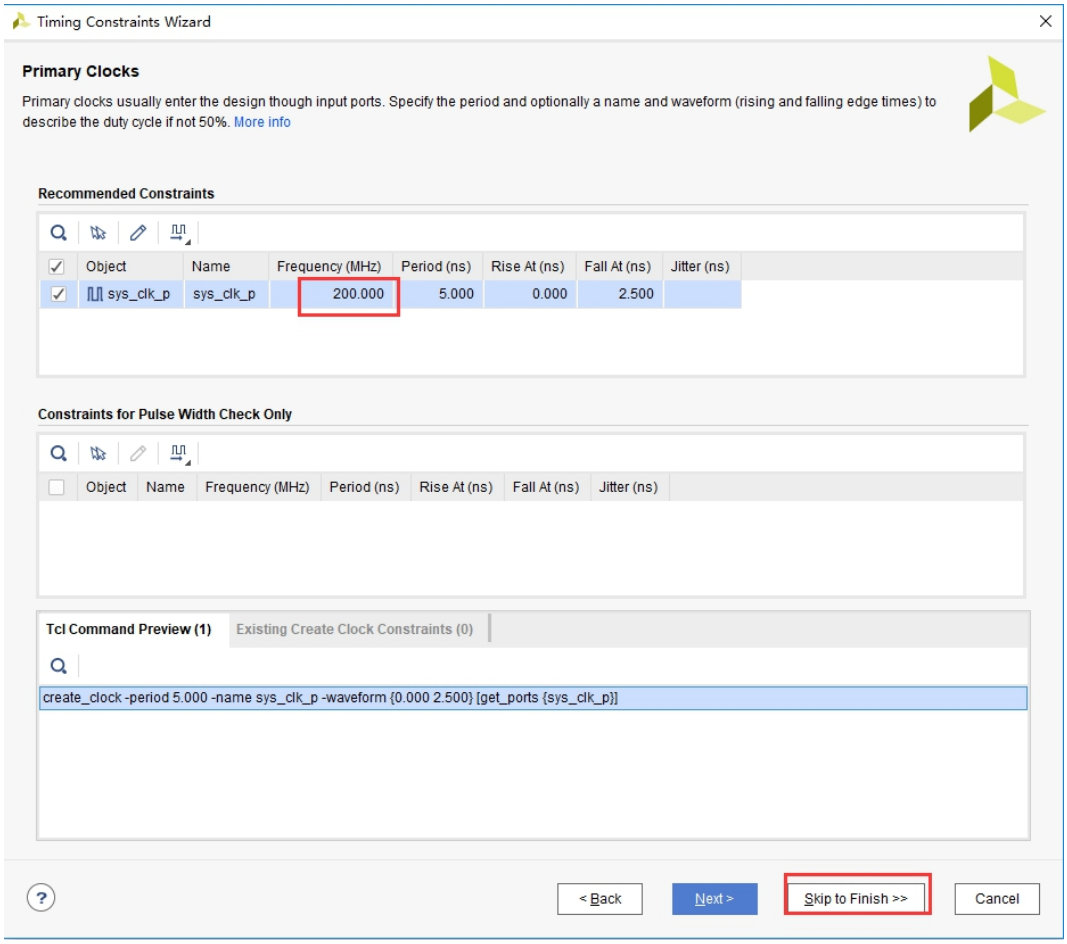
4) Click “Constraints Wizard”



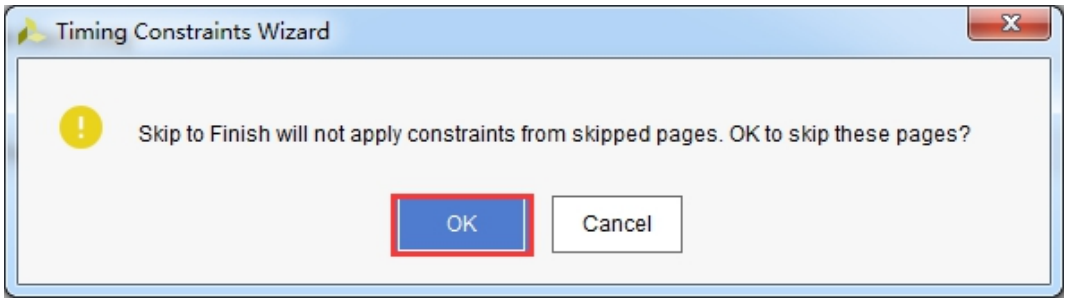
5) Click "Next" in the pop-up window.



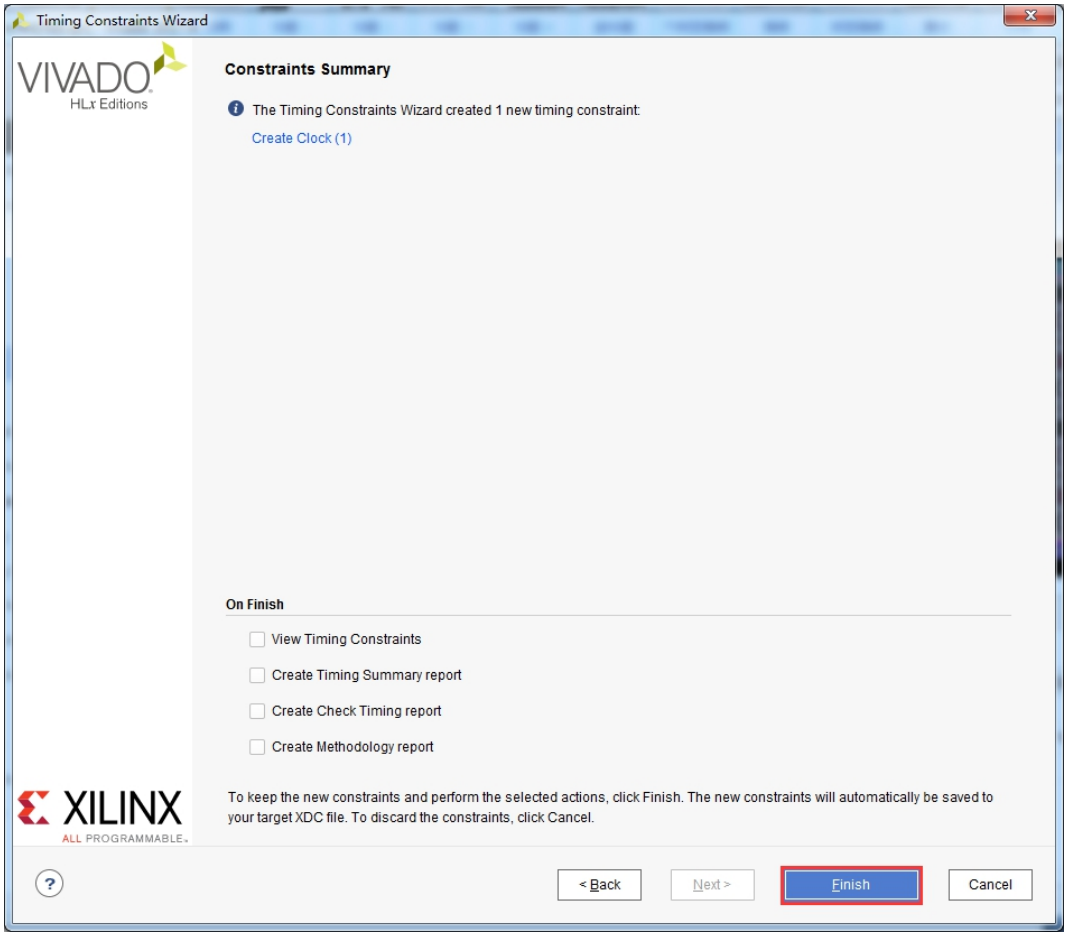
6) In the Timing Constrains Wizard, set the “sys_clk” frequency is “200Mhz”, then click “Skip to Finish"



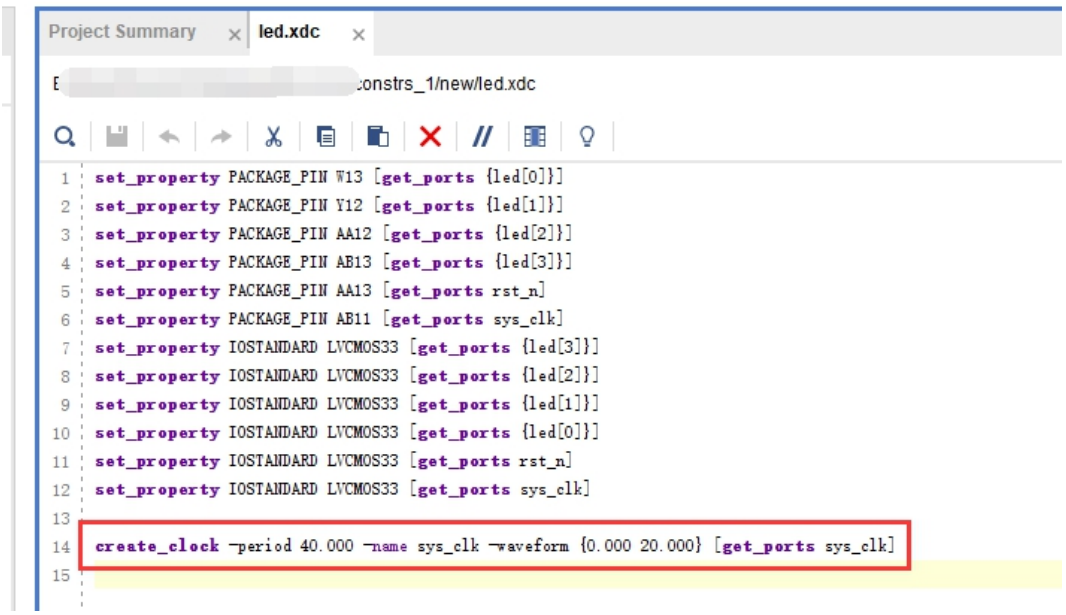
7) Click "OK" in the pop-up window



8) Click “Finish”

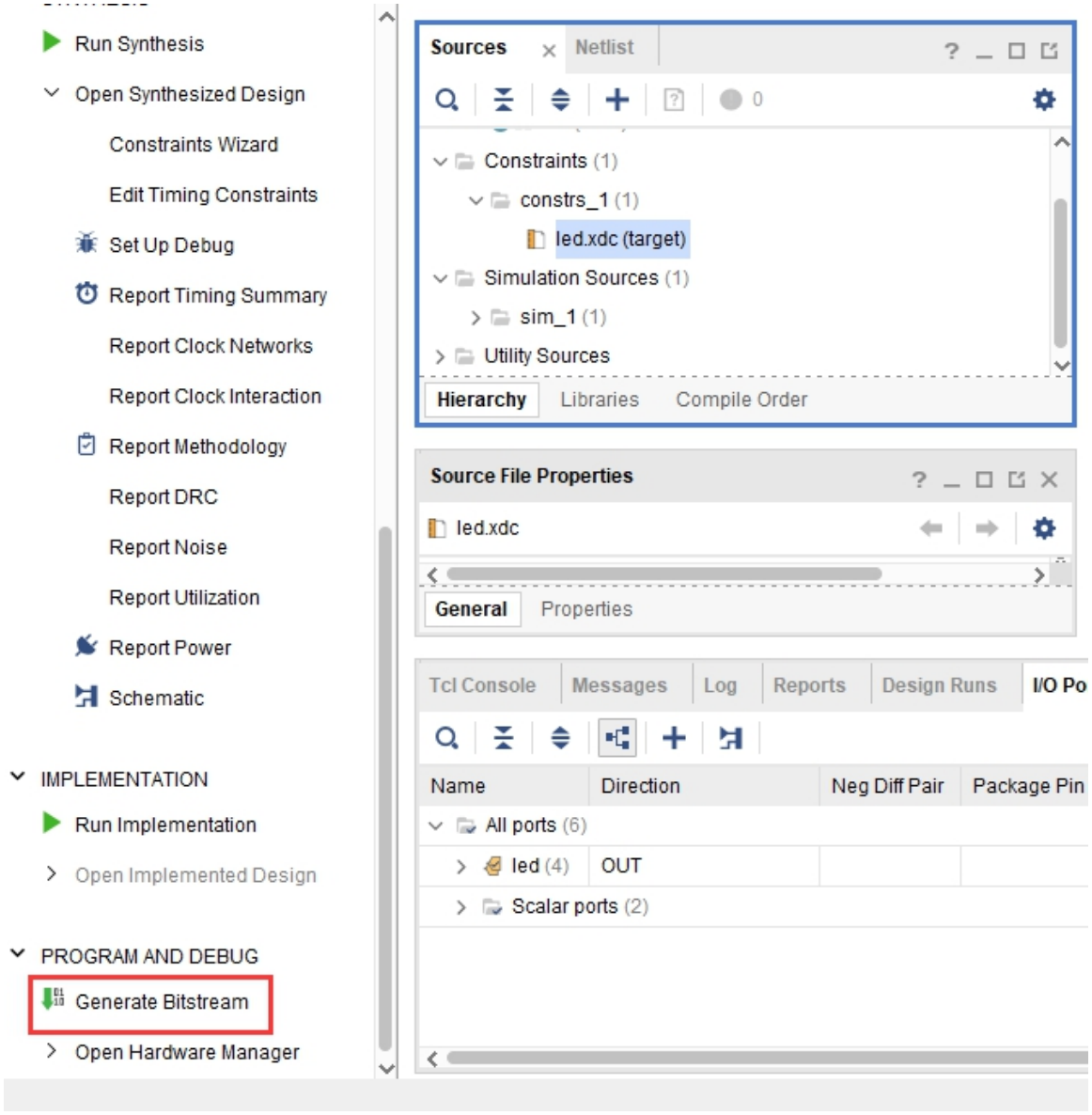


9) At this time, the “led.xdc” file has been updated. Click "Reload" to reload the file and save the file.

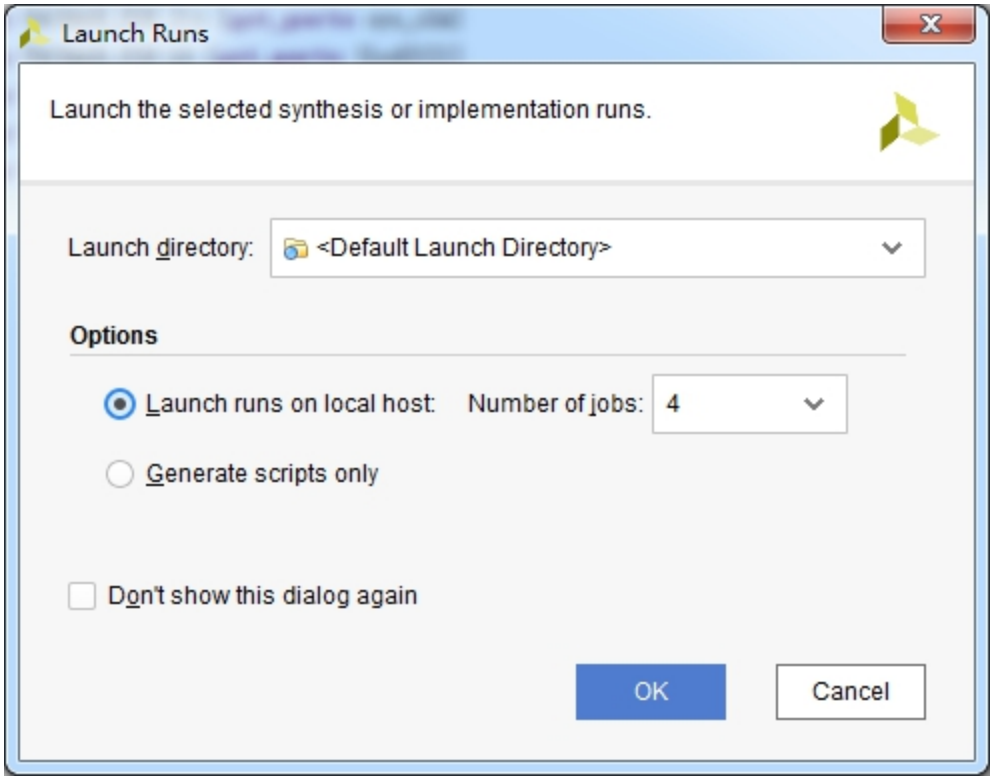


Part 4.6: Generate BIT File

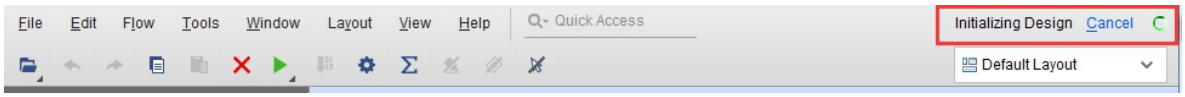
1) The compilation process can be subdivided into synthesis, place and route, generate bit files, etc. Here we directly click on "Generate Bitstream" to directly generate bit files.



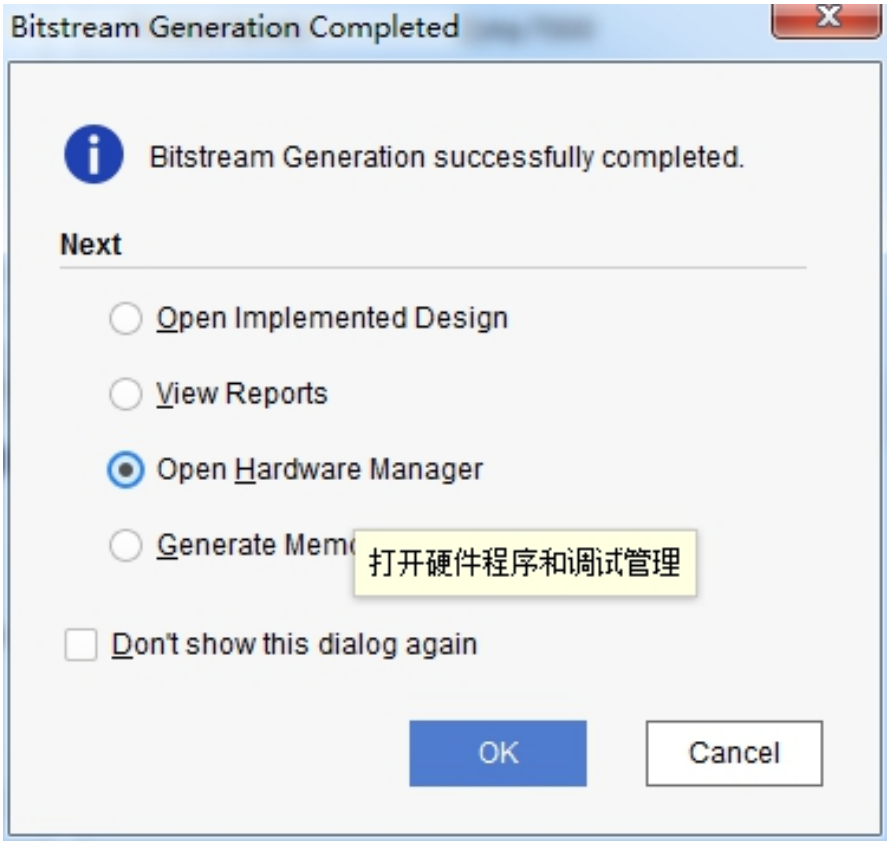
2) In the pop-up dialog box, you can select the number of tasks, which is related to the number of CPU cores. The larger the general number, the faster the compilation. Click “OK”.



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



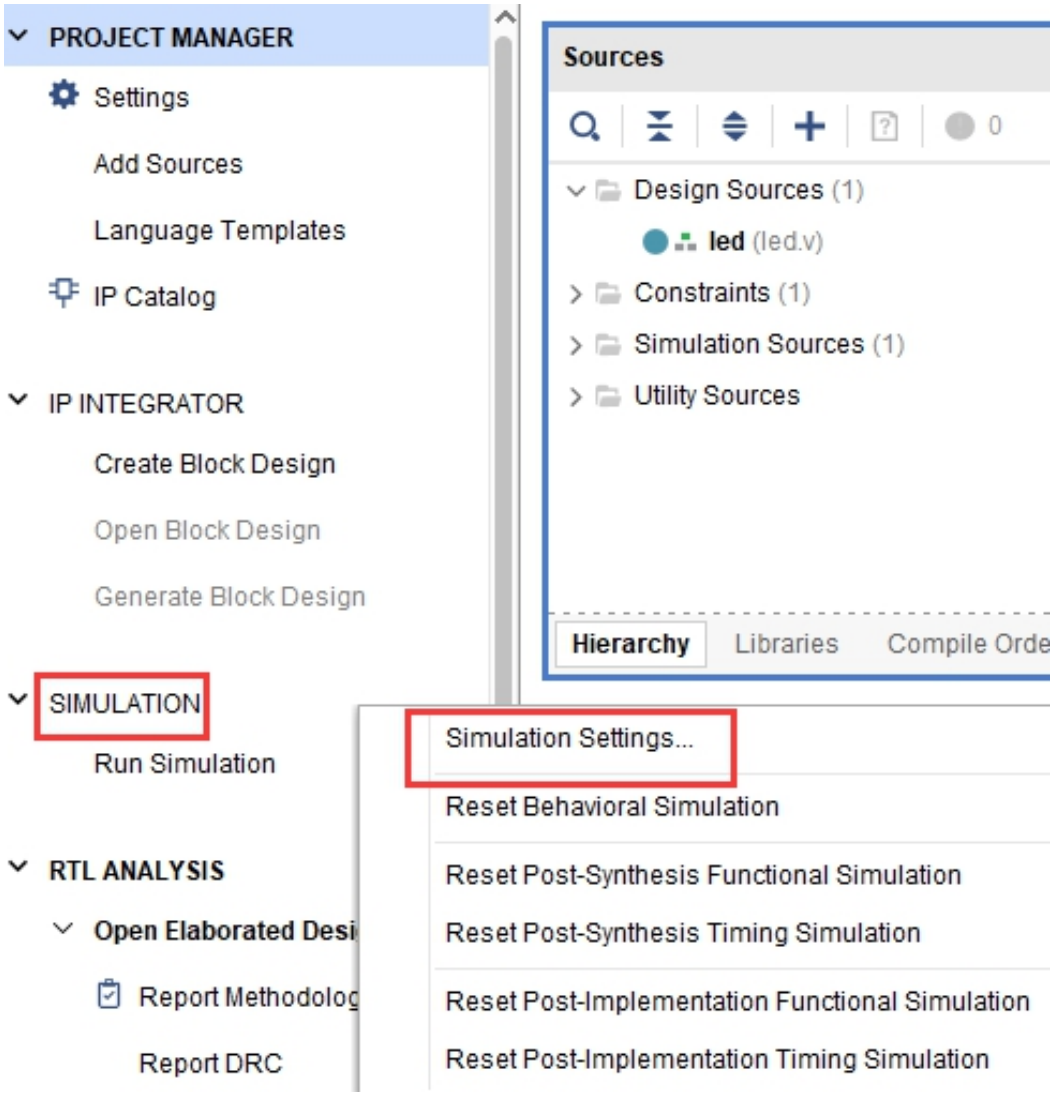
4) There is no error in the compilation, the compilation is completed, a dialog box pops up for us to select the subsequent operation, you can select "Open Hardware Manger", of course, you can also select "Cancel", we select "Cancel" here, do not download first



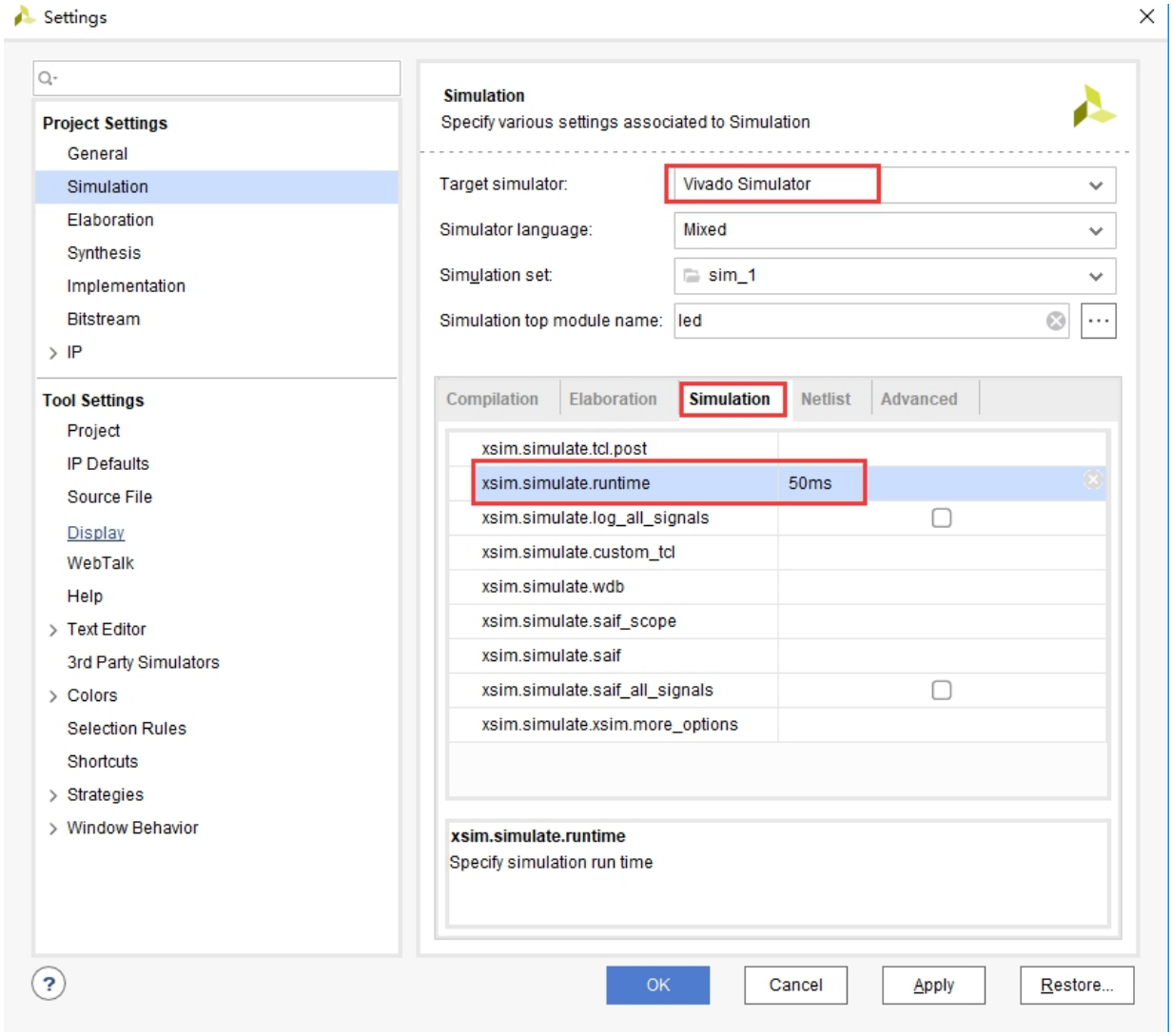
Part 4.7: Vivado Simulation

Next, use Vivado's own simulation tool to output the waveform. Verify that the flow light programming results are consistent with our expectations (Note: You can also simulate before generating the bit file) . Specific steps are as follows:

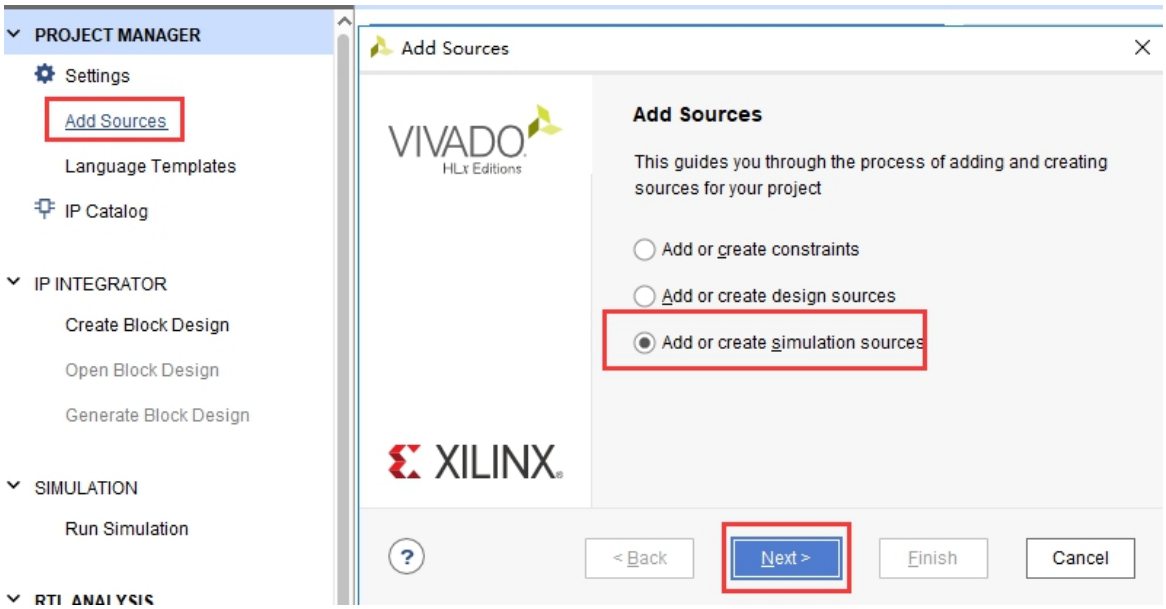
1) Set up the simulation configuration for Vivado and right click on "Simulation Settings" in "SIMULATION".



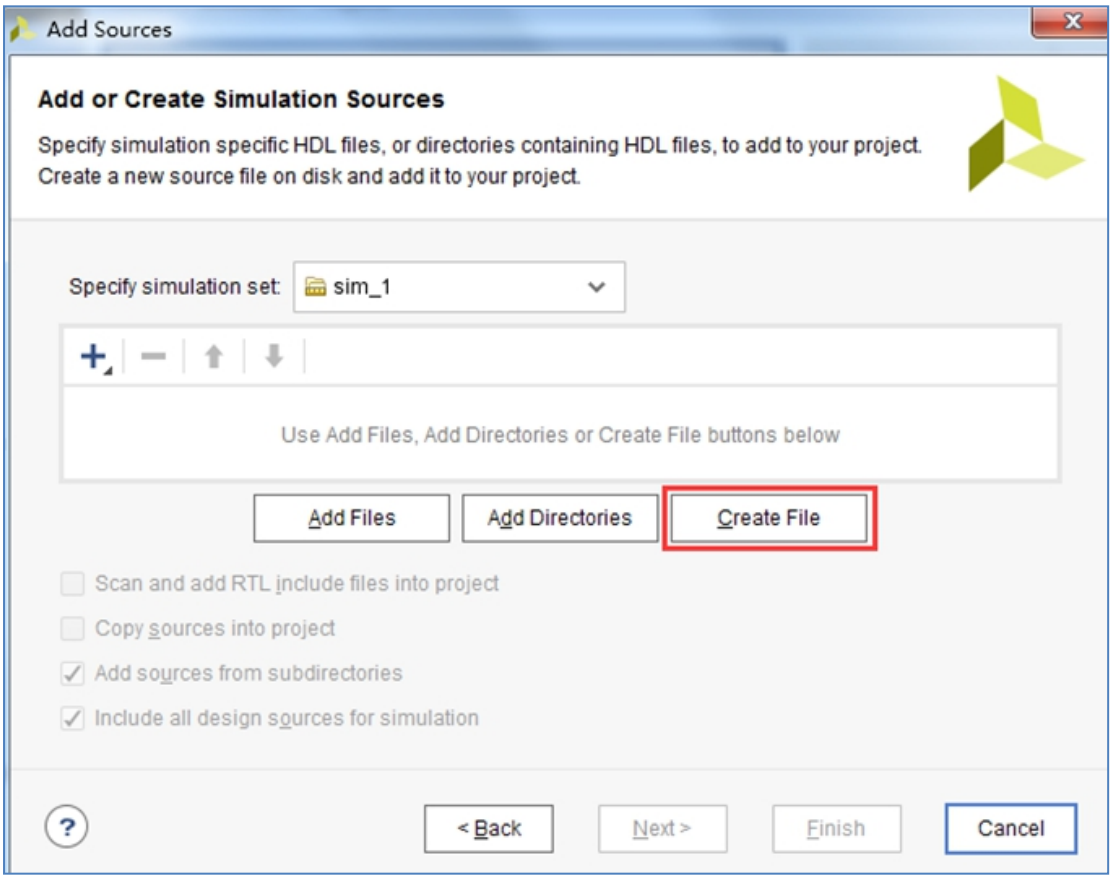
2) In the Simulation Settings window, perform the following configuration to configure it. Set it to 50ms (set it as needed), and press OK by default.



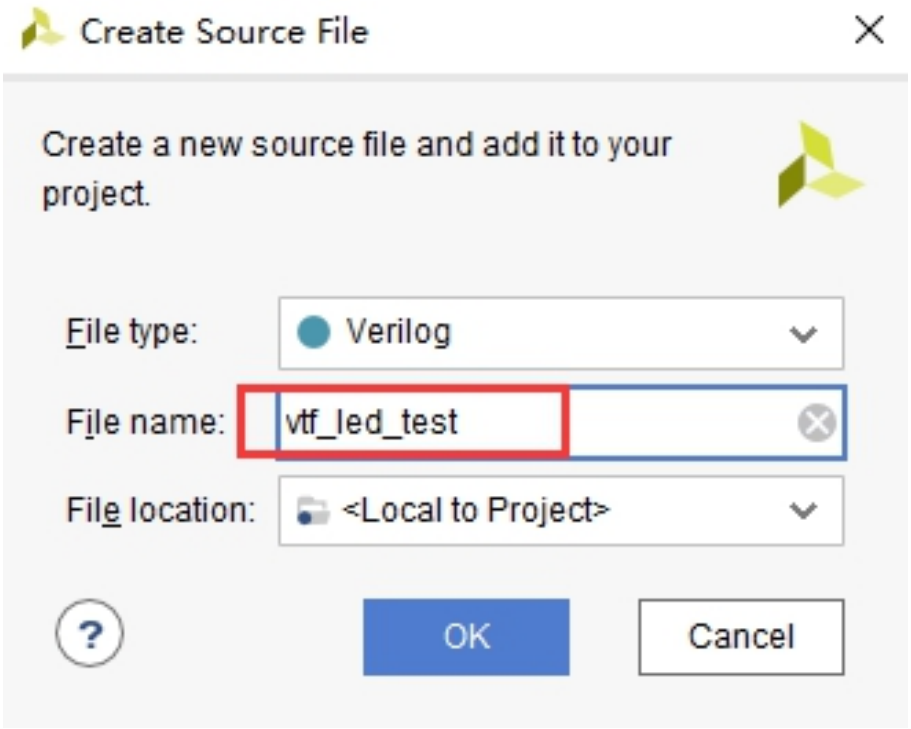
3) Add the stimulus test file, click the “Add Sources” icon under “Project Manager”, click the settings below and click “Next”.



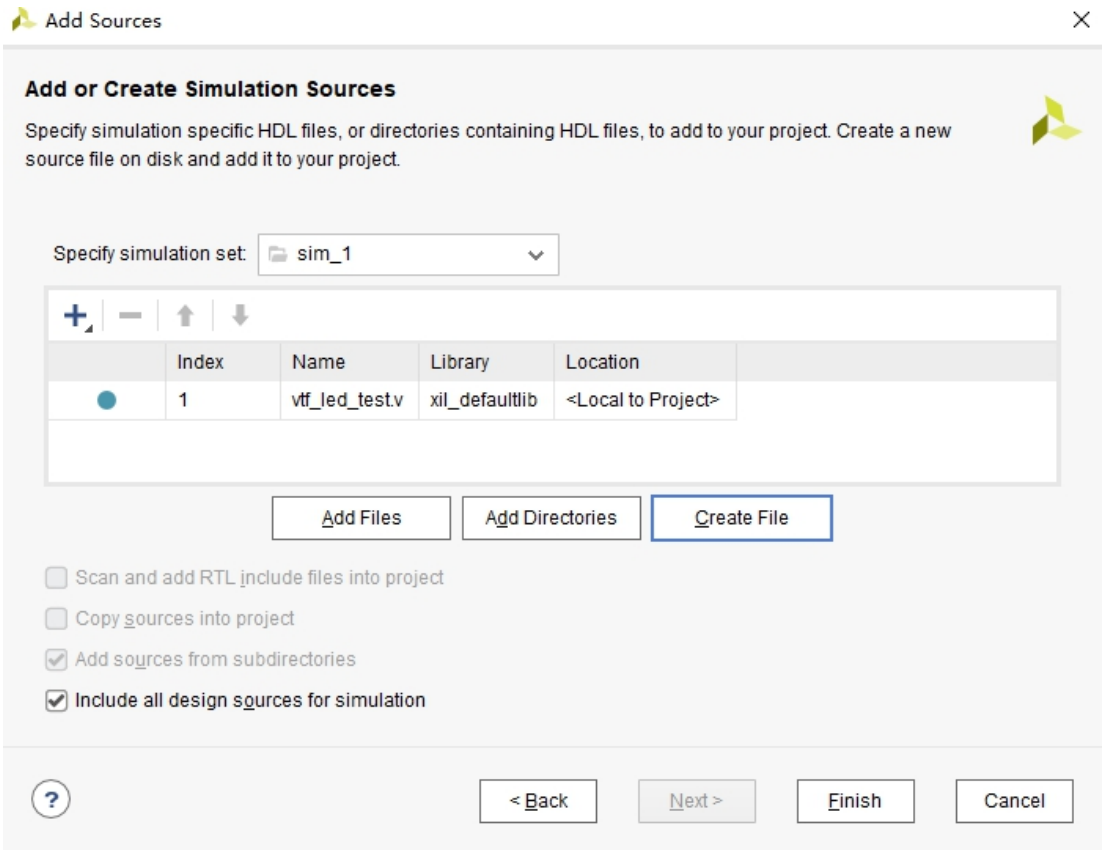
4) Click “Create File” to generate a simulation stimulus file.



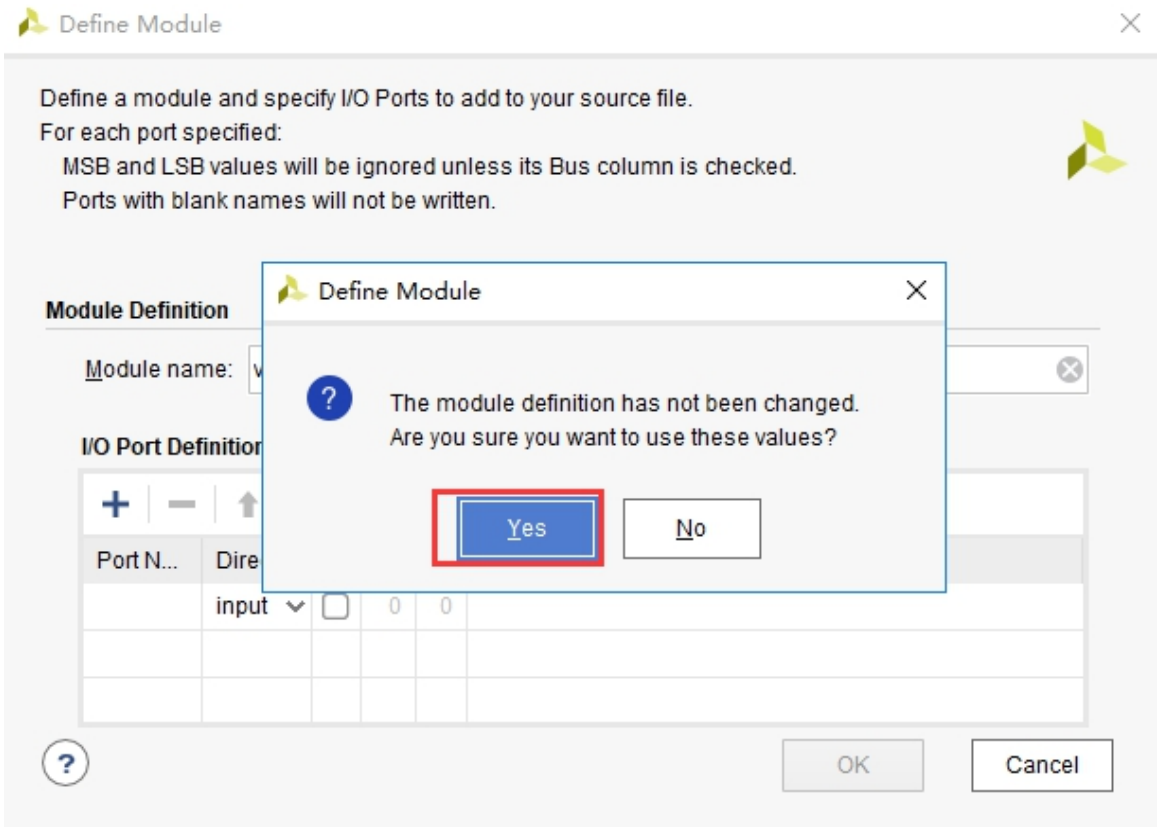
Enter the name of the stimulus file in the pop-up dialog box, here we enter the name “vtf_led_test”.



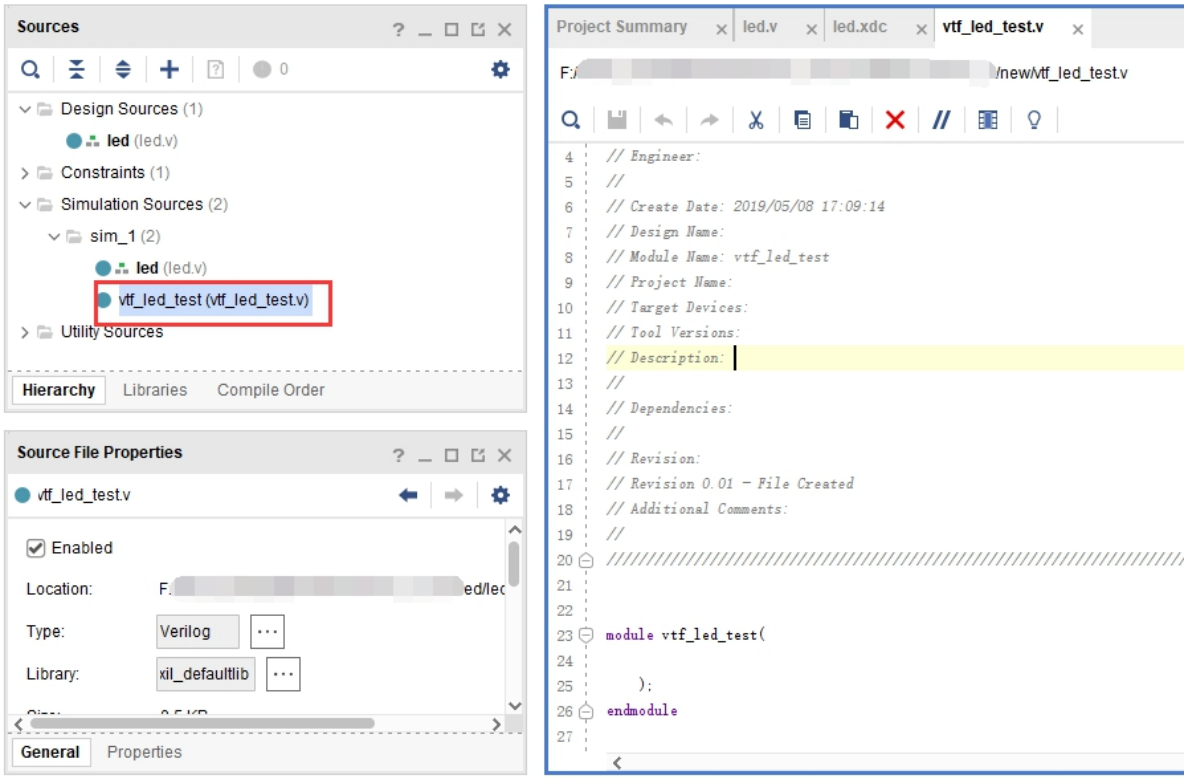
5) Click the Finish button to return



Here we do not add IO Ports first, click OK.



Add a “vtf_led_test” file just added in the “Simulation Sources” directory. Double-click to open this file, you can see that there is only the definition of the “module” name, nothing else.

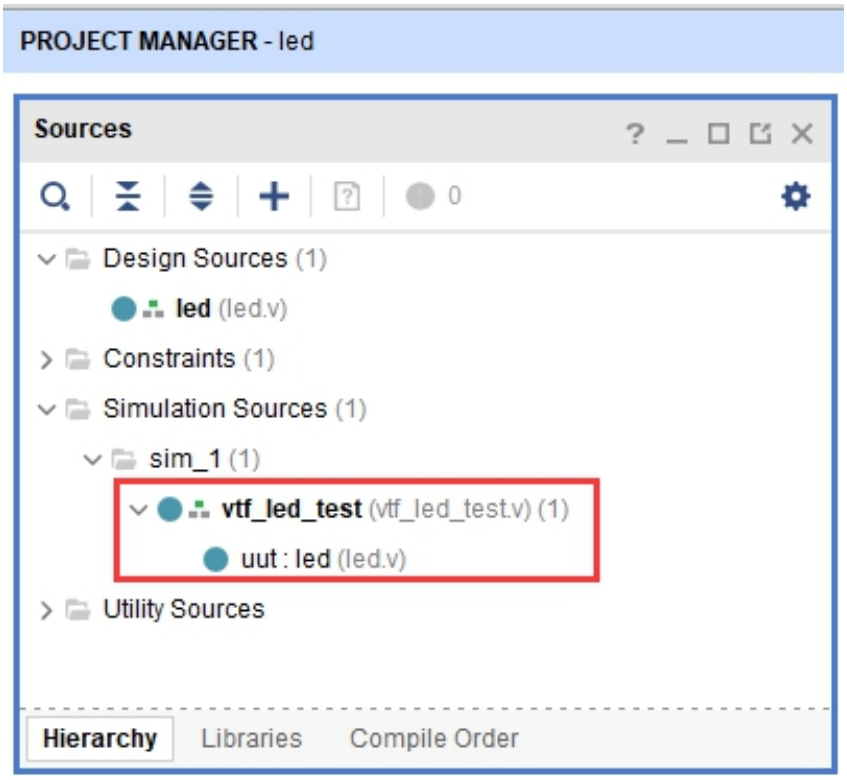


6) Next we need to write the contents of this “vtf_led_test.v” file. First define the input and output signals, then you need to instantiate the “led_test” module to make the “led_test” program part of the test program. Add the reset and clock excitation. The completed “vtf_led_test.v” file is as follows:

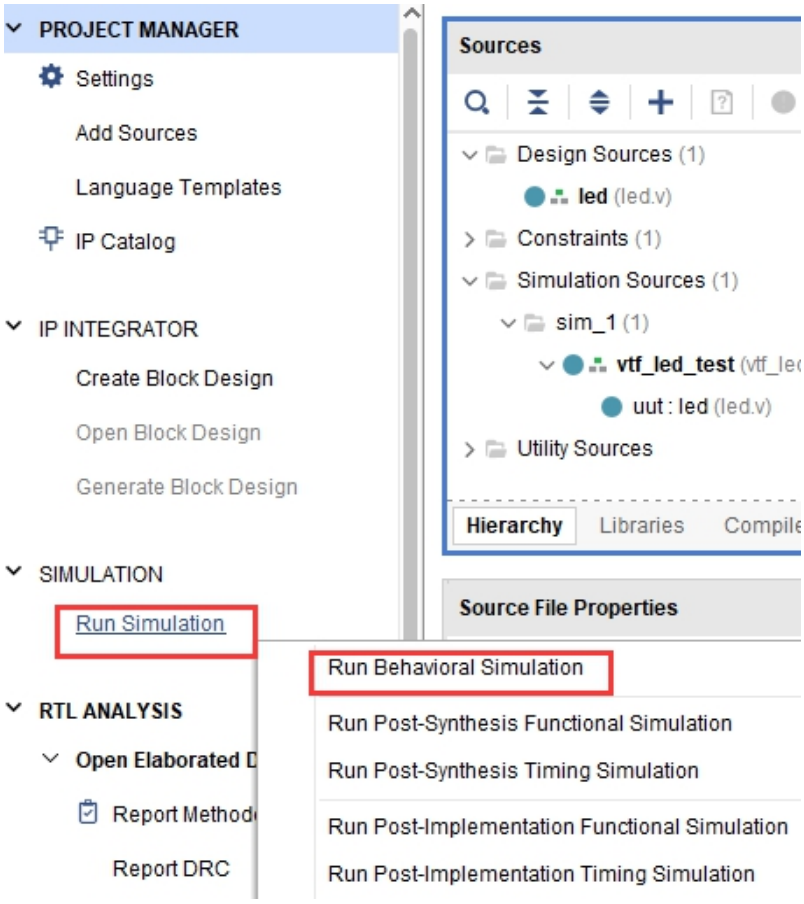
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Module Name: vtf_led_test
//////////////////////////////////////////////////////////////////////////////////

module vtf_led_test;
// Inputs
reg sys_clk_p;
reg rst_n ;
```

```
wire sys_clk_n;
// Outputs
wire led;

// Instantiate the Unit Under Test (UUT)
led uut (
    .sys_clk_p(sys_clk_p),
    .sys_clk_n(sys_clk_n),
    .rst_n(rst_n),
    .led(led)
);

initial
begin
// Initialize Inputs
    sys_clk_p = 0;
    rst_n = 0;
// Wait for global reset to finish
    #1000;
    rst_n = 1;
end
//Create clock
always #2.5 sys_clk_p = ~ sys_clk_p;
assign  sys_clk_n = ~sys_clk_p ;

endmodule
```

7) After writing and saving, “vtf_led_test.v” automatically becomes the top layer of this simulation “Hierarchy”. Below it is the design file “led_test.v”.

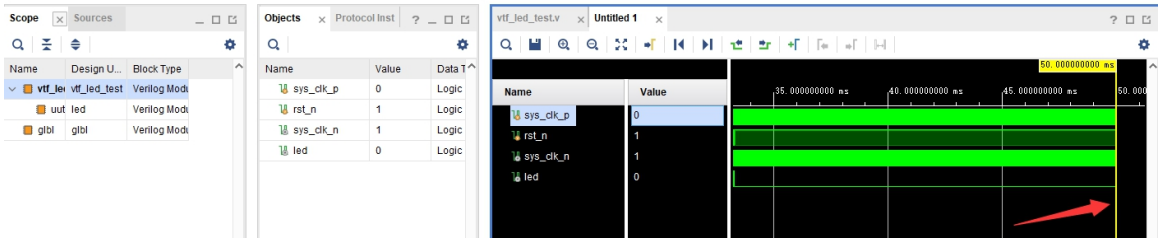


8) Click the “Run Simulation” button and select “Run Behavioral Simulation”. Here we can do a behavioral level simulation.

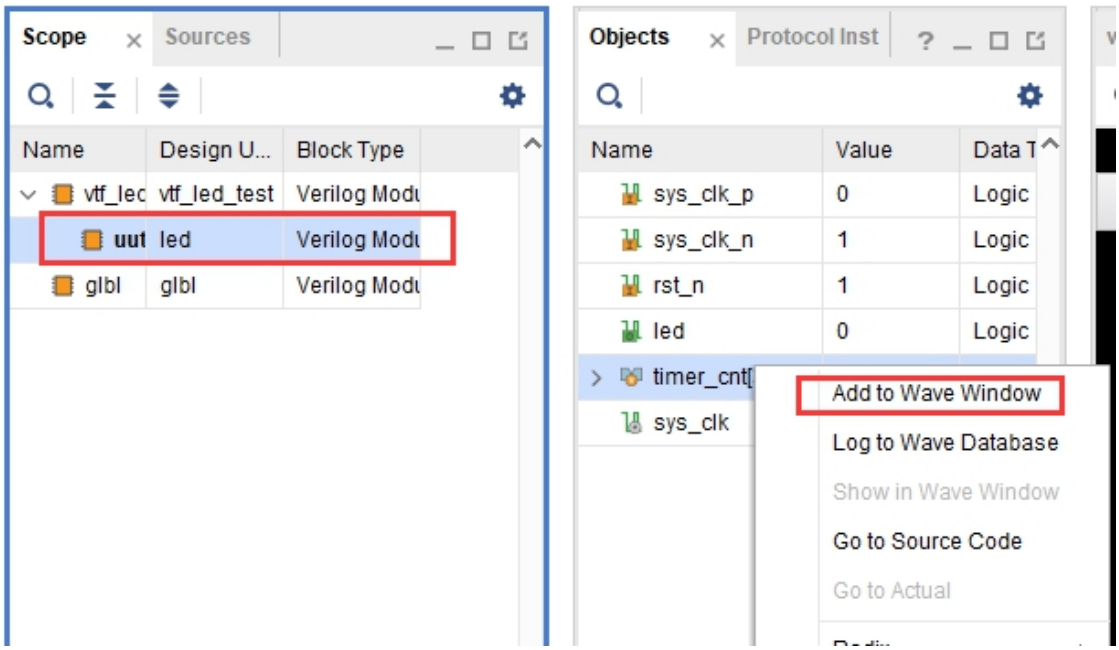


If there are no errors, the simulation software in Vivado is working.

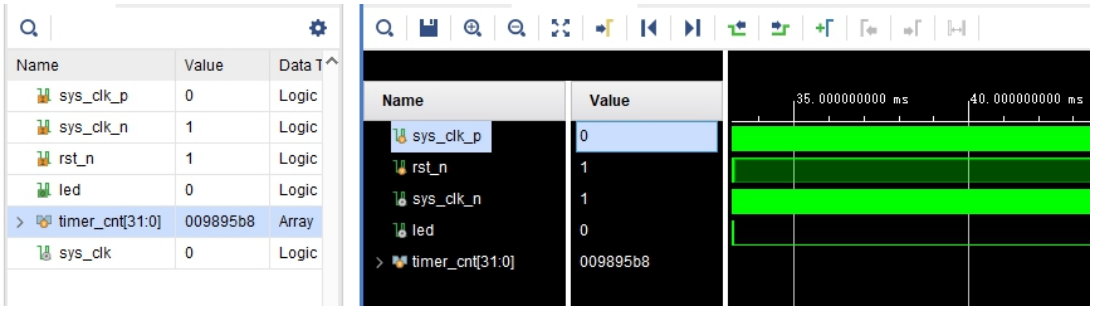
9) After popping up the simulation interface, as shown below, the interface is a 50ms waveform that the simulation software automatically runs to the simulation setup.



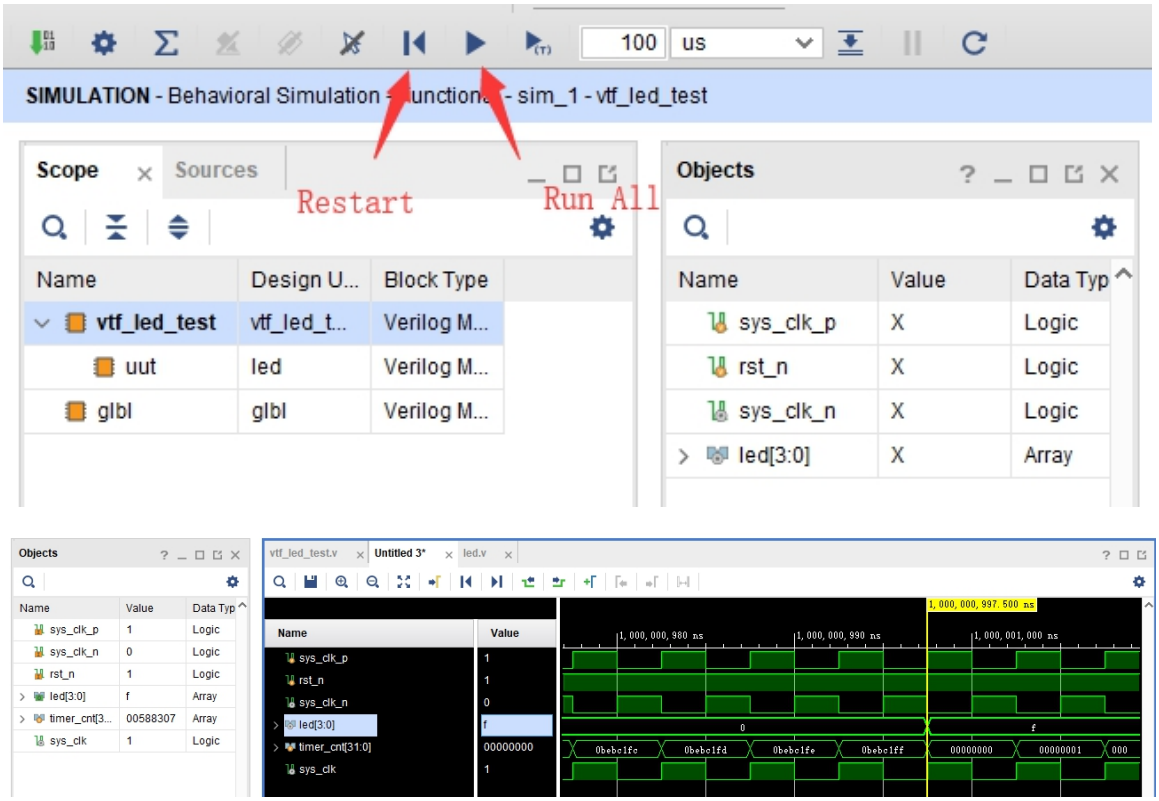
Since the state of “LED[3:0]” design in the program changes for a long time, and the simulation is time consuming, observe the “timer[31:0]” counter change here. Put it in the Wave to observe (click uut under the Scope interface, then right click on the timer under the “Objects” interface, select “Add Wave Window” in the pop-up drop-down menu).



After adding, the timer is displayed on the waveform interface of Wave, as shown in the figure below.



10) Click the “Restart” button labeled below to reset it, then click the “Run All” button. (Need patience!!!), you can see that the simulation waveform matches the design. (Note: The longer the simulation time, the larger the disk space occupied by the simulated waveform file, the waveform file is in the “xx.sim” folder of the project directory)

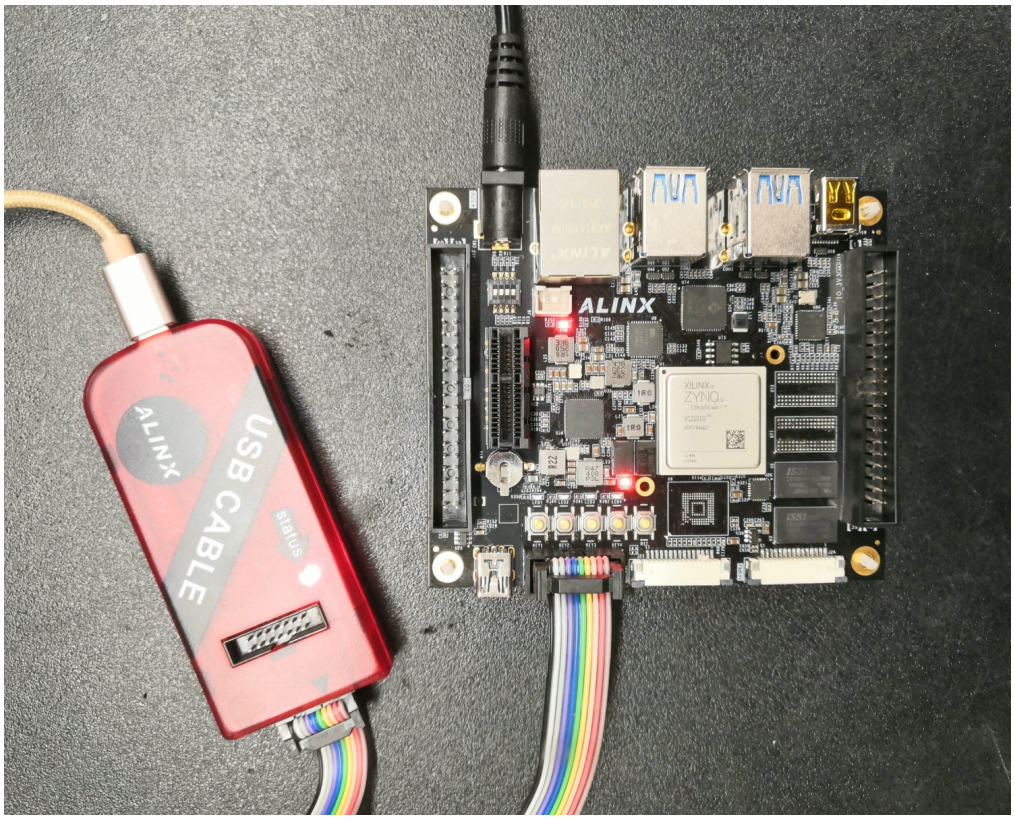


We can see that the led signal will change to 1, indicating that the LED light is on.

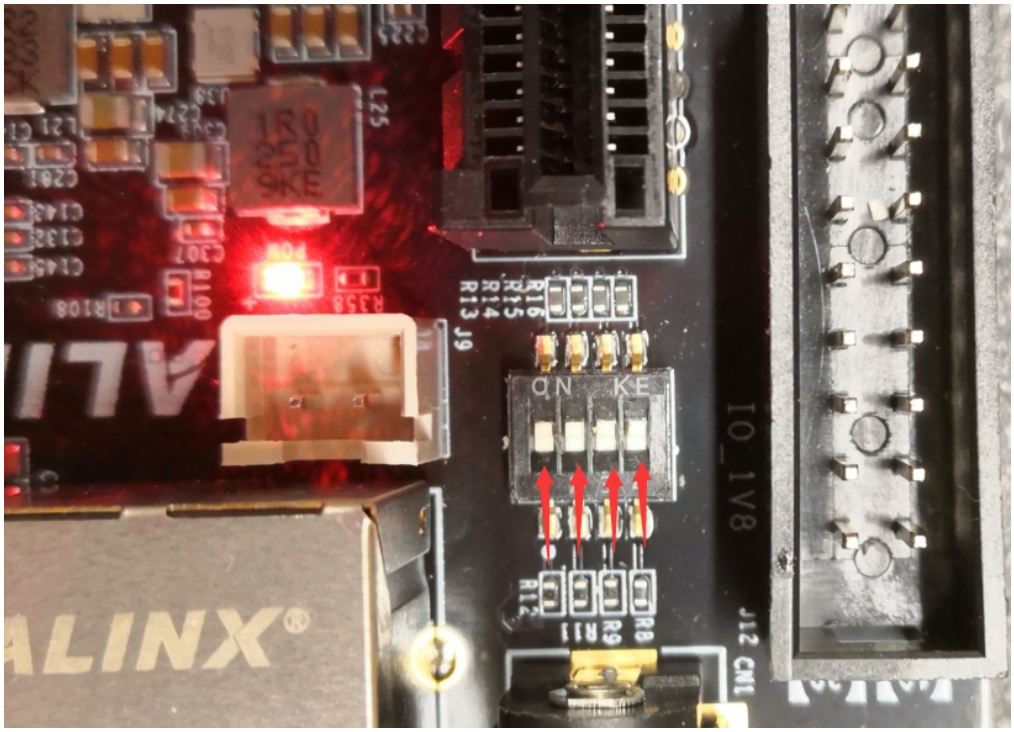
Part 4.8: Download

1) Connect the JTAG interface of the FPGA development board.

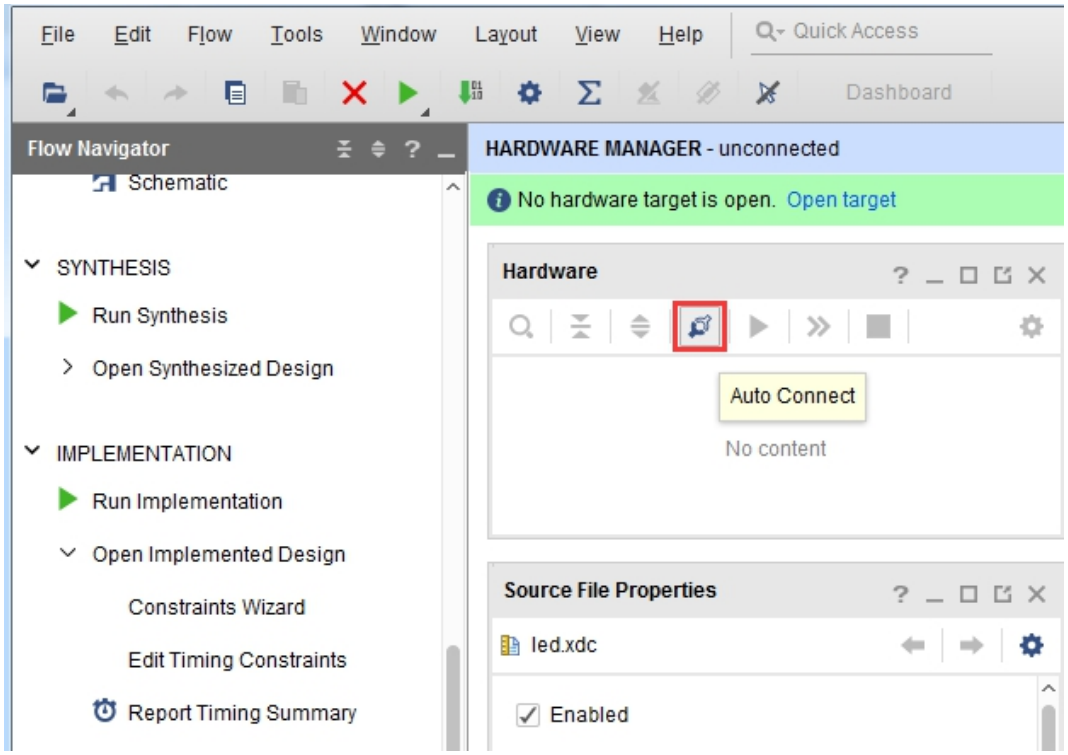
Power on the FPGA Development board.



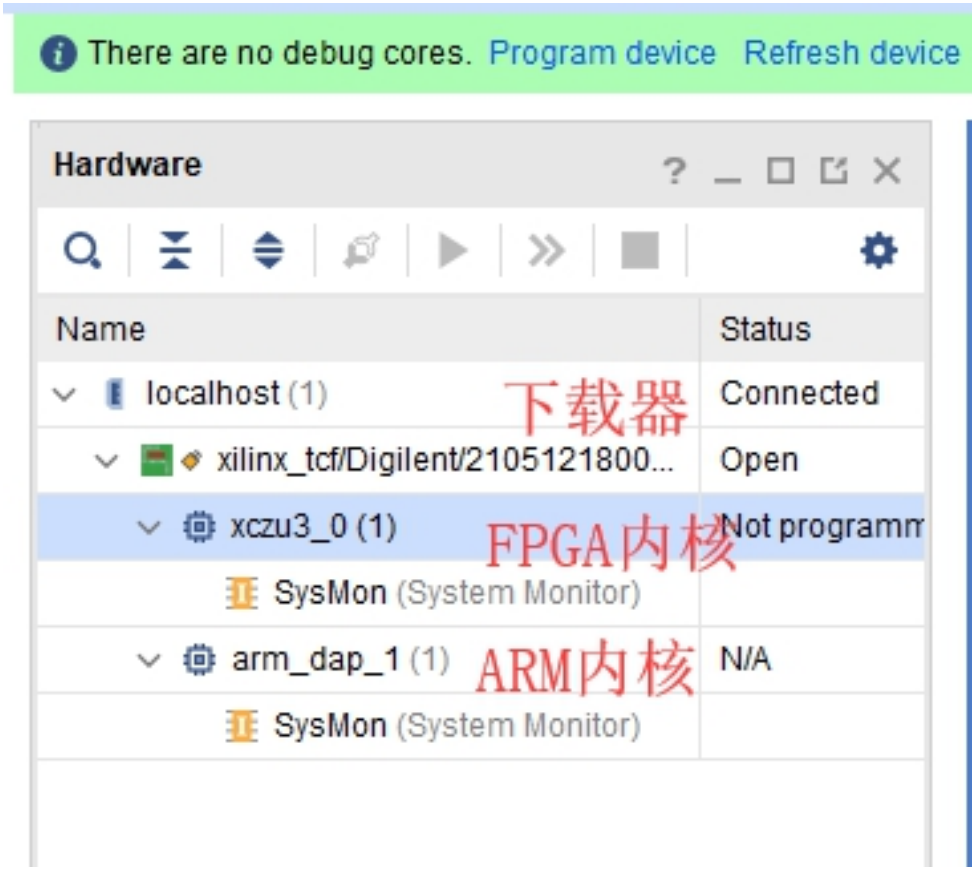
Note that the DIP switch should select the JTAG mode, that is, pull them all to "ON". The value of "ON" is 0. If the JTAG mode is not used, an error will be reported when downloading.



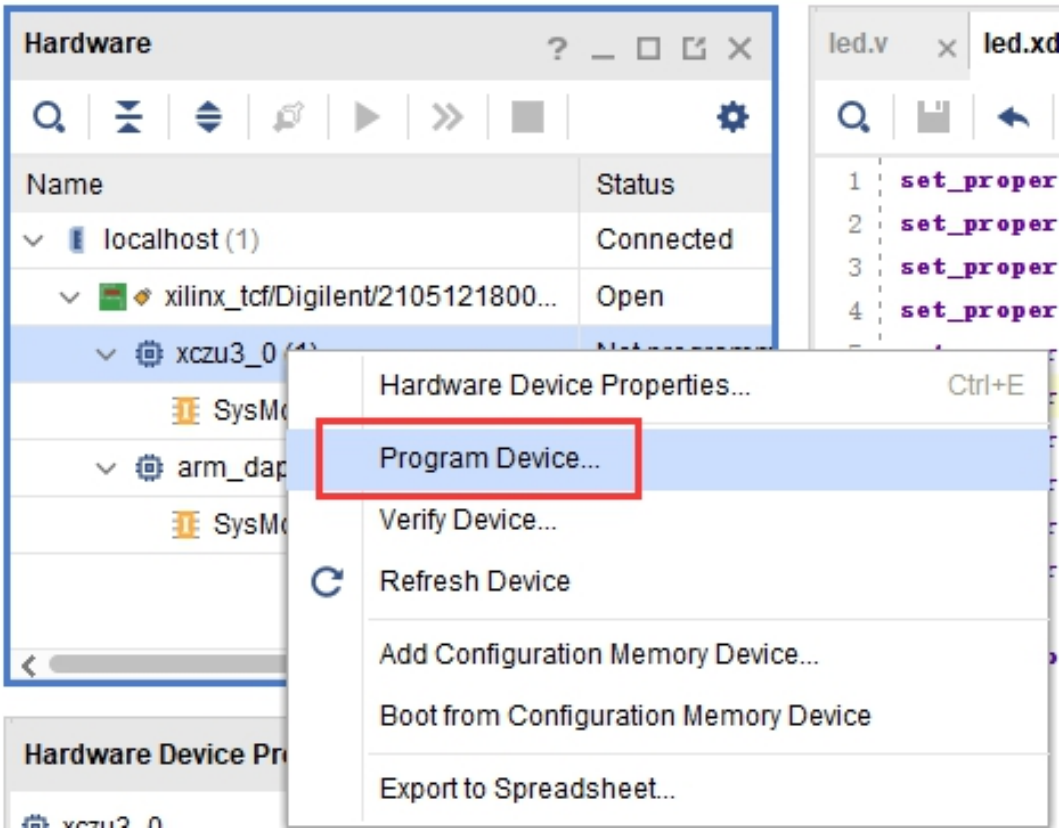
2) On the “HARDWARE MANAGER” interface, click “Auto Connect” to automatically connect the device.



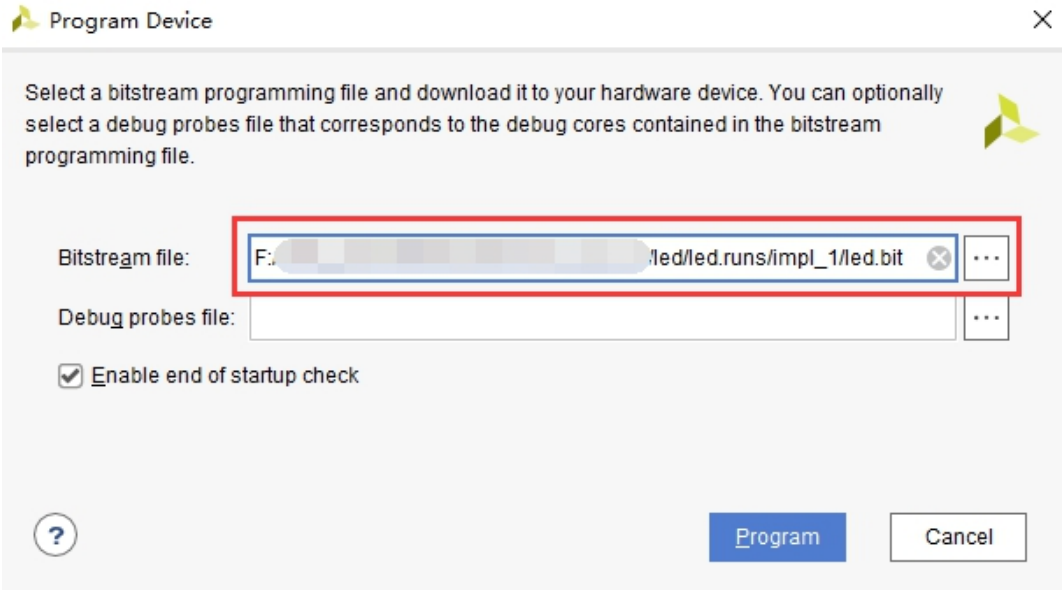
3) You can see JTAG scan to arm and FPGA core



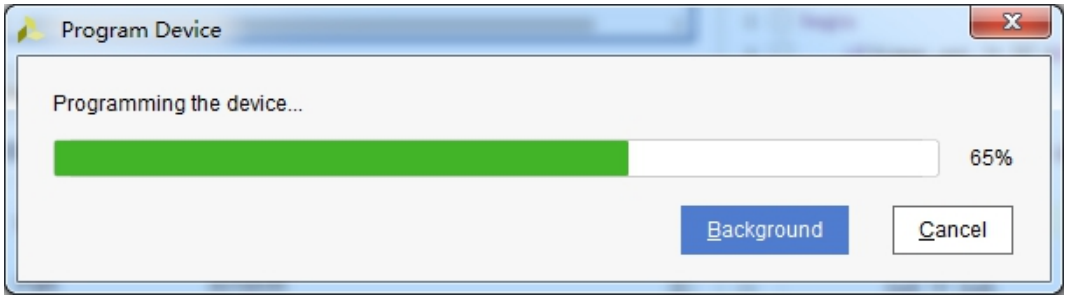
4) Select chip, right click "Program Device..."



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



6) Waiting for download



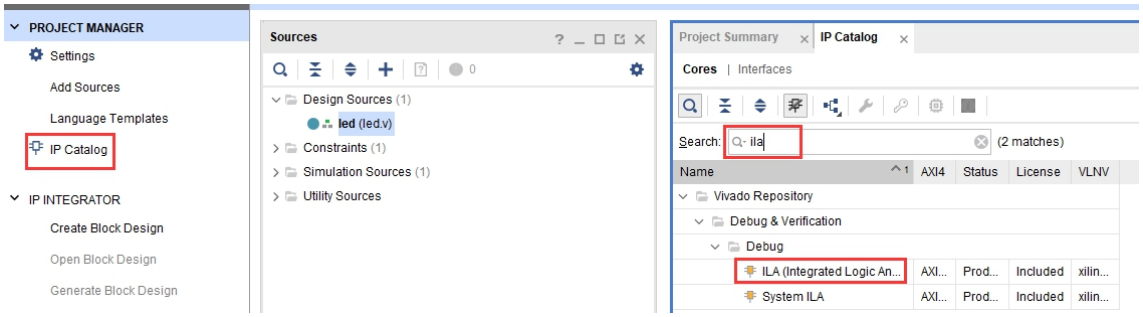
7) After the download is complete, we can see that PL LED start to change every second. The Vivado simple process experience is now complete. The following chapters will introduce how to burn the program to Flash. It needs the cooperation of the PS system to complete. Only the PL project can not directly write Flash. It is described in the FAQs in the chapter "Experiencing ARM, Bare Metal Output" "Hello World".

Part 4.9: Online debugging

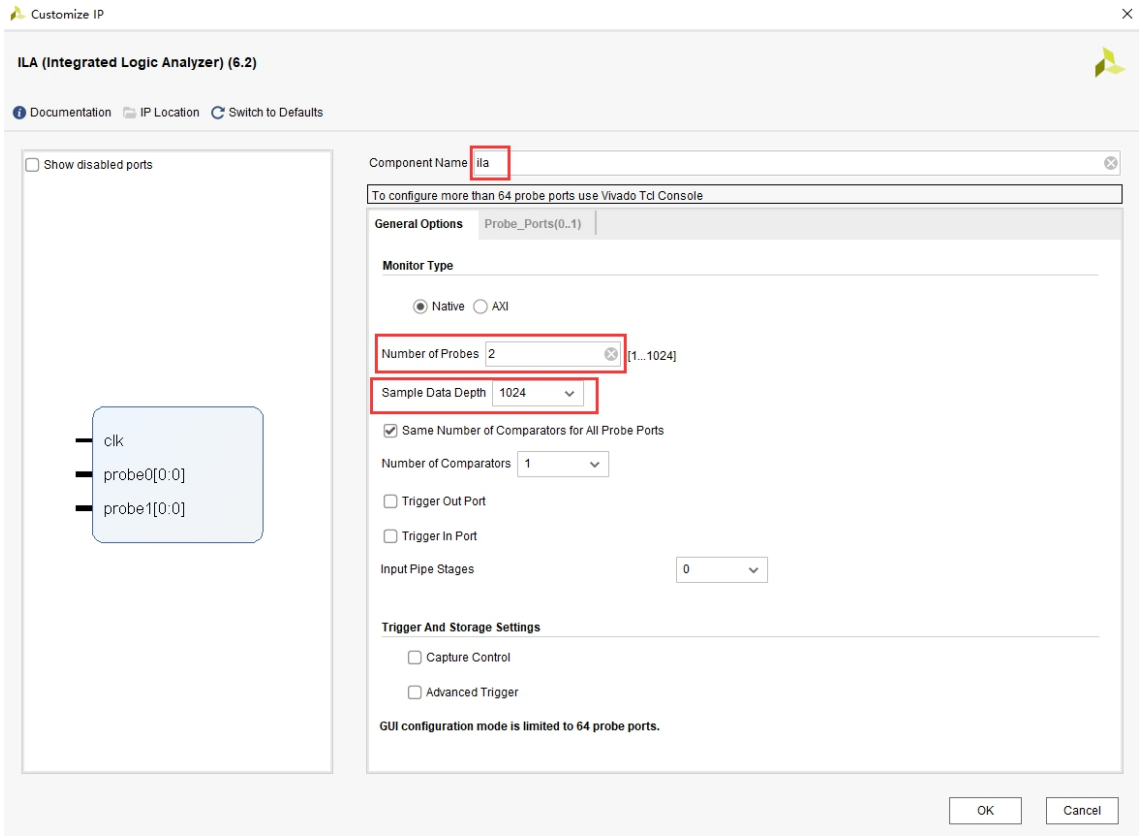
The simulation and download are introduced earlier, but the simulation does not require the program to be written to the board. It is an ideal result. The Vivado online debugging method is introduced below to observe the internal signal changes. Vivado has an embedded logic analyzer called ILA that can be used to observe changes in internal signals online, which is very helpful for debugging. In this experiment we observe the signal changes of “timer_cnt” and “led”.

Part 4.9.1: Add ILA IP Core

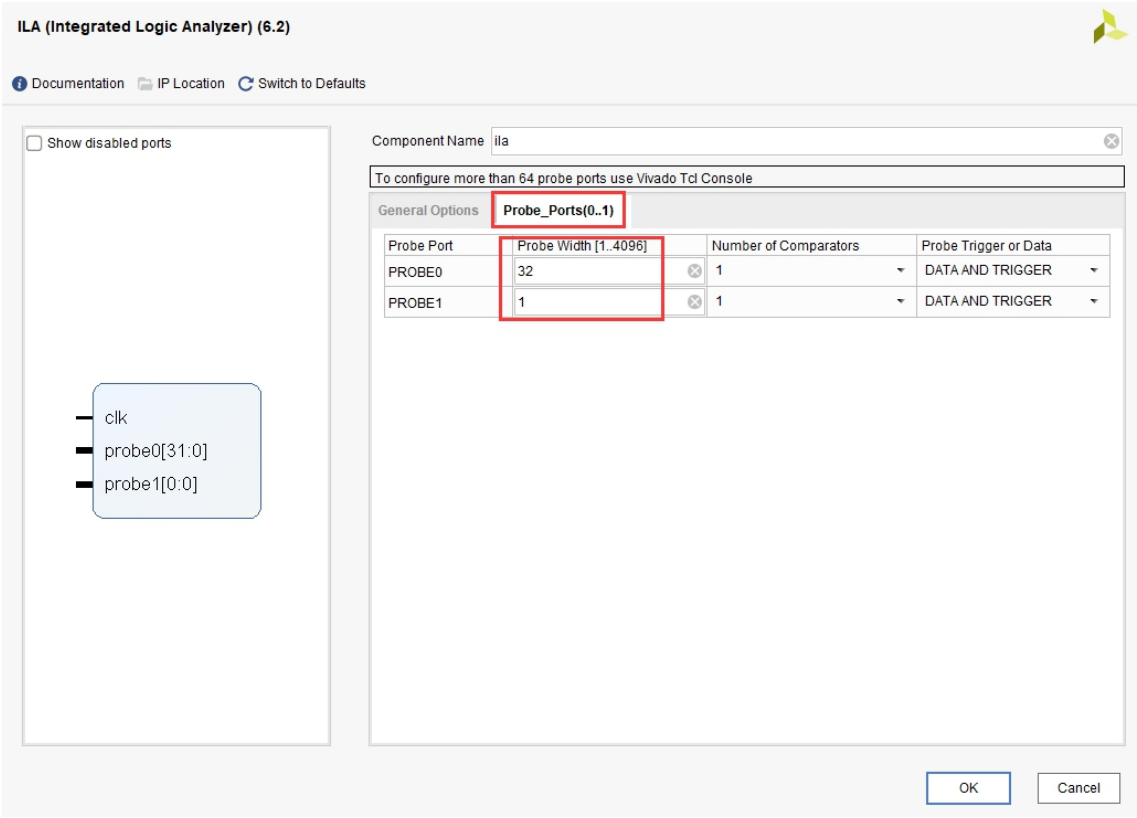
1) Click on “IP Catalog”, search for “ila” in the search box, double click on the “IP” of “ILA”



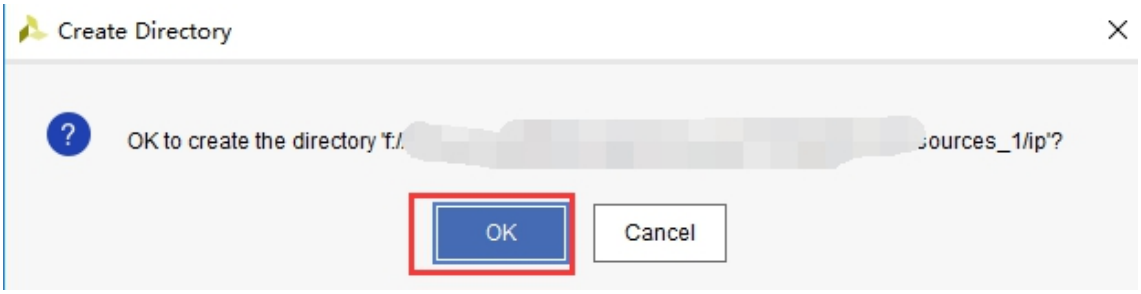
2) The name is modified to “ila”. Since two signals are to be sampled, the number of Probes is set to 2. “Sample Data Depth” refers to the sampling depth. The higher the setting, the more signals are collected, and the more resources are consumed.



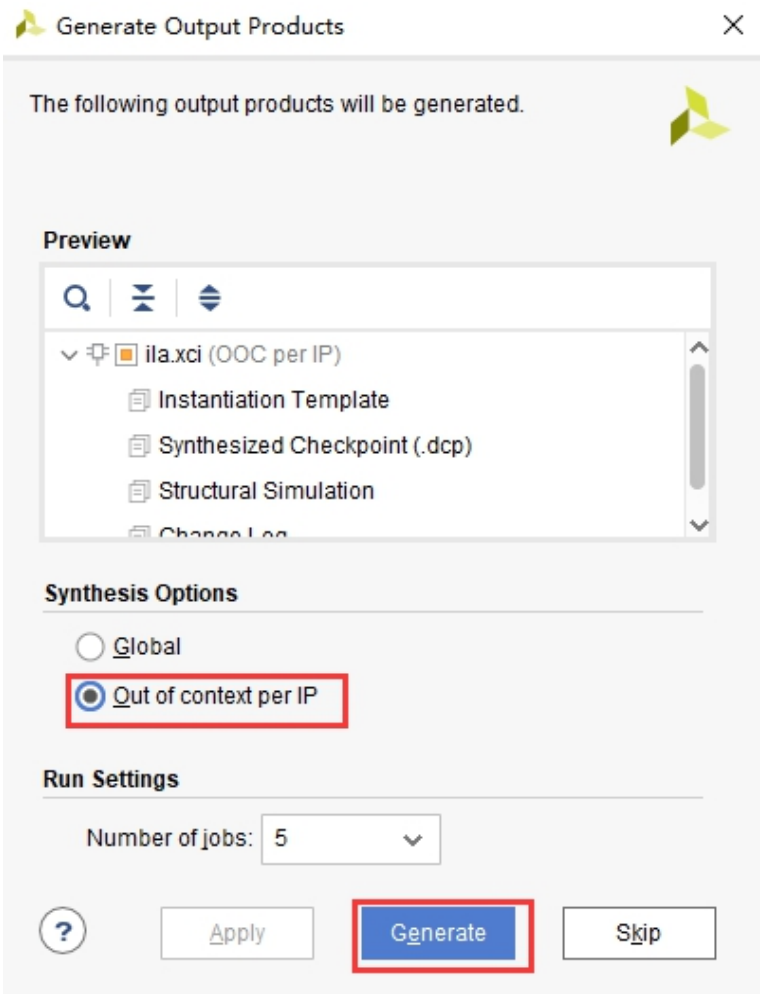
3) On the “Probe_Ports” page, set the width of the “Probe”, set the “PROBE0” bit width to “32”, use it to sample “timer_cnt”, and set the “PROBE1” bit width to “1” for sampling led. Click OK



Pop up the interface, select OK



Set it as below, click “Generate”

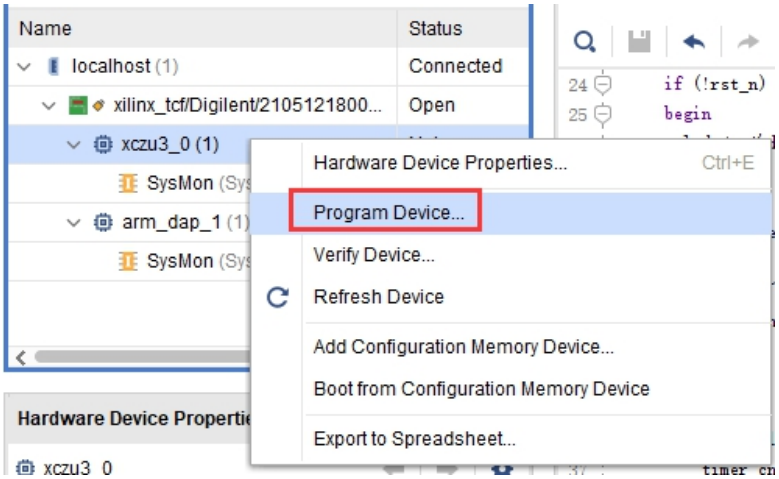


4) Instantiate “ila” in “led.v” and save

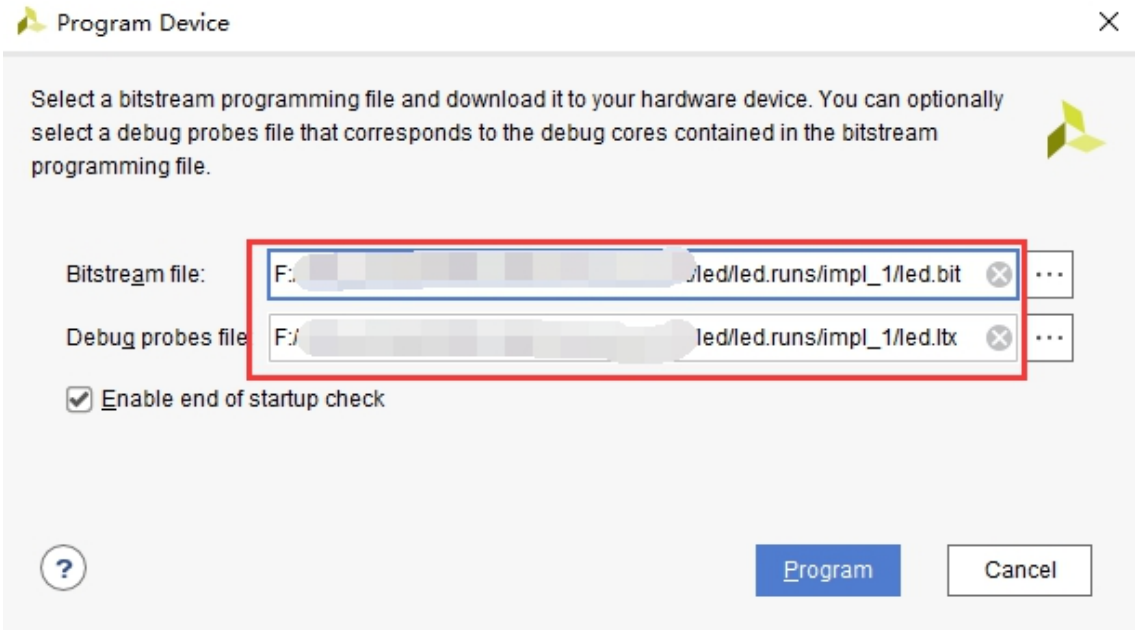
```
40      end
41      else
42      begin
43          led <= led;
44          timer_cnt <= timer_cnt + 32'd1;
45      end
46
47  end
48
49  //Instantiate ila in source file
50  ila ila_inst(
51    .clk(sys_clk),
52    .probe0(timer_cnt),
53    .probe1(led)
54    );
55
56  endmodule
57
```

5) Bitstream Regenerate “Bitstream”

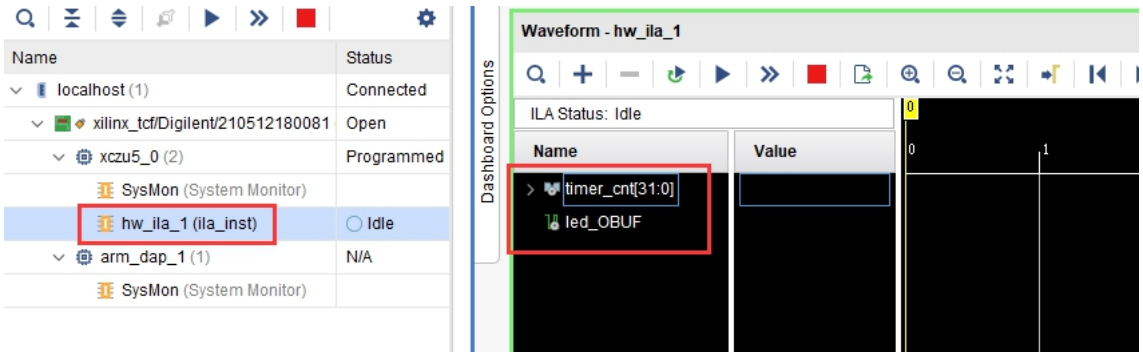
6) Download the program



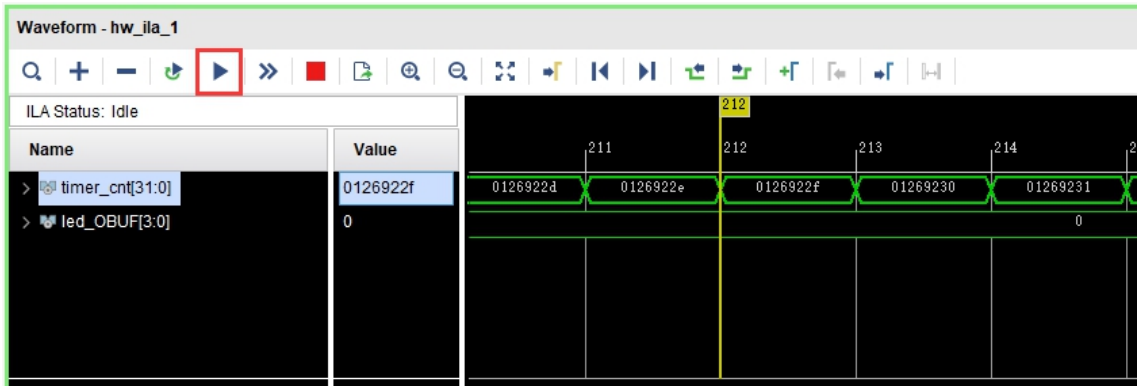
7) Then, you can find a “bit” and “ltx” file, click “program”



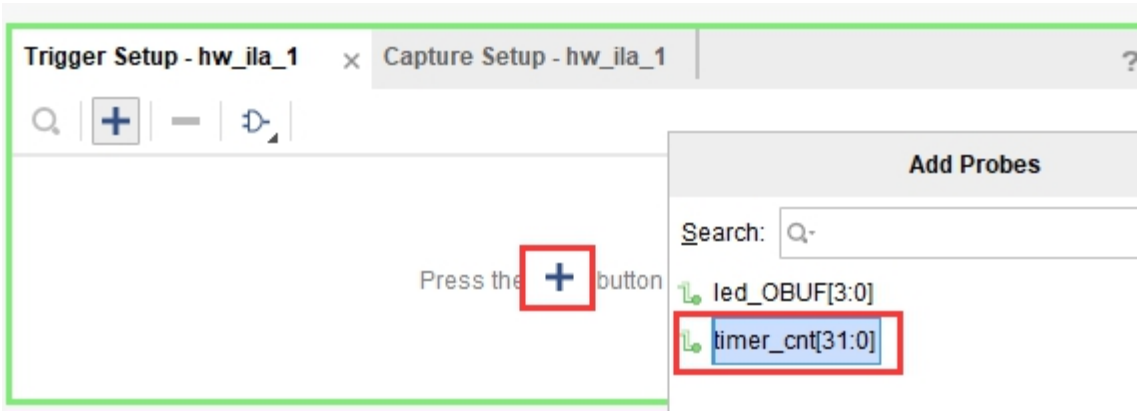
8) At this point the debug window pops up online, there have been signals we add



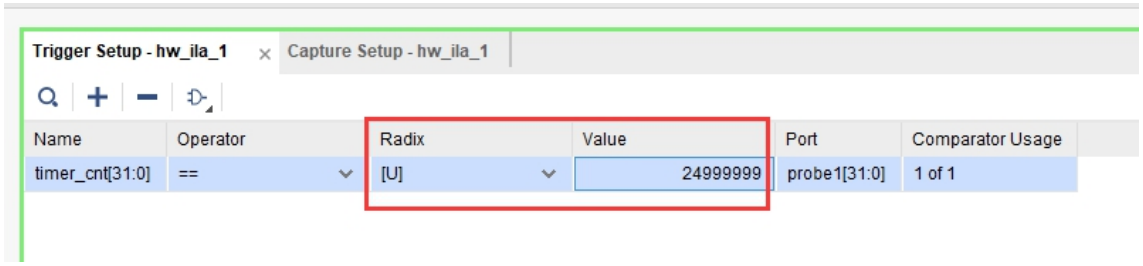
Click the Run button and the signal data appears.



It can also trigger the acquisition. Click “+” in the “Trigger Setup” window to select the “timer_cnt” signal in depth.

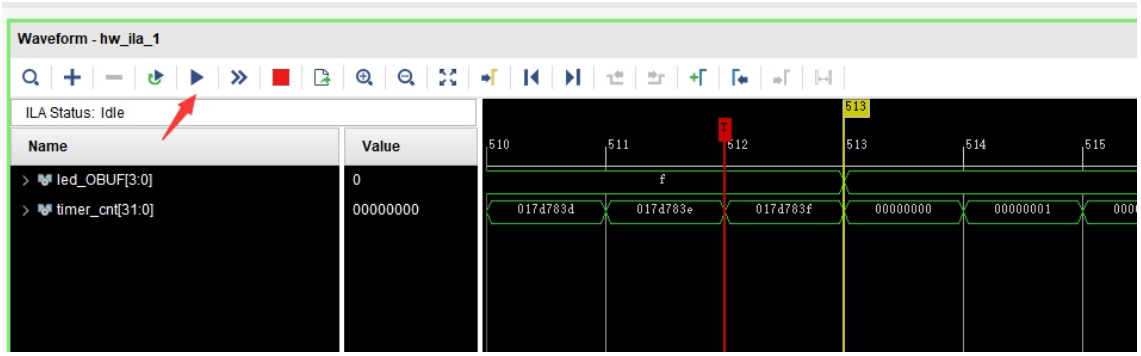


Change “Radix” to “U”, which is decimal, set to “49999999” in Value, which is the maximum value of “timer_cnt”.



Click Run again, you can see that the trigger is successful. At this

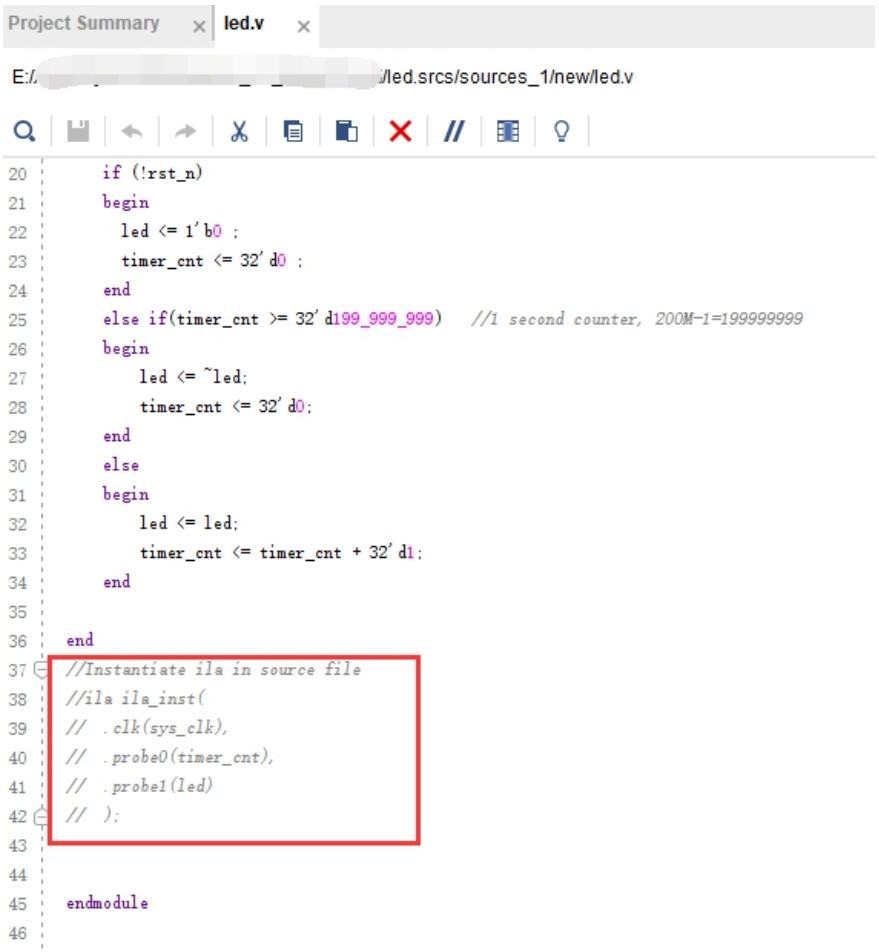
time, “timer_cnt” is displayed as hexadecimal, and led is also flipped at this time.



Part 4.9.2: MARK DEBUG

The above describes the way to add ILA IP online debugging. The following describes adding comprehensive attributes to the code to implement online debugging.

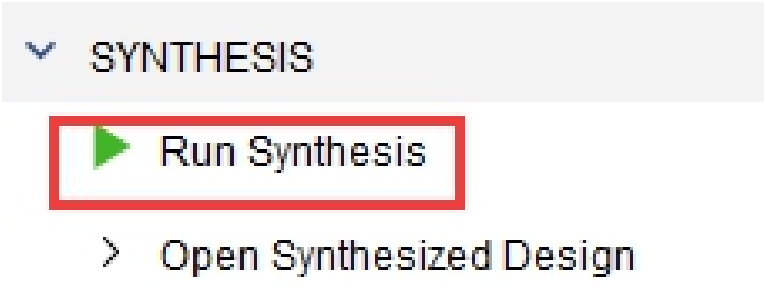
1) First open “led.v”, remove the annotations of the “ila” instant part.



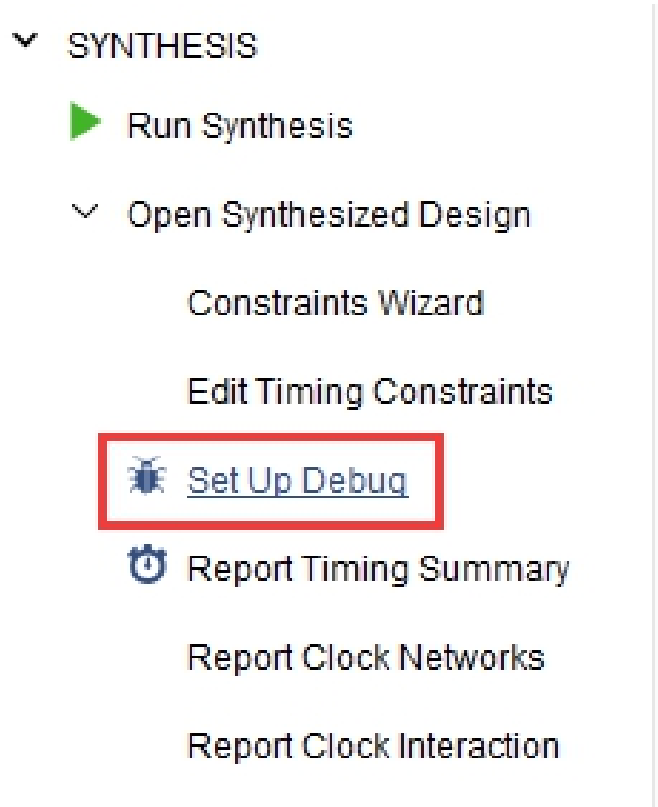
2) Add “(* MARK_DEBUG=”true” *)” in front of the definition of “led” and “timer_cnt”, save the file

```
module led(
    input sys_clk,
    input rst_n,
    (* MARK_DEBUG="true" *)output reg [3:0] led
    );
(* MARK_DEBUG="true" *)reg[31:0] timer_cnt;
always@(posedge sys_clk or negedge rst_n)
begin
```

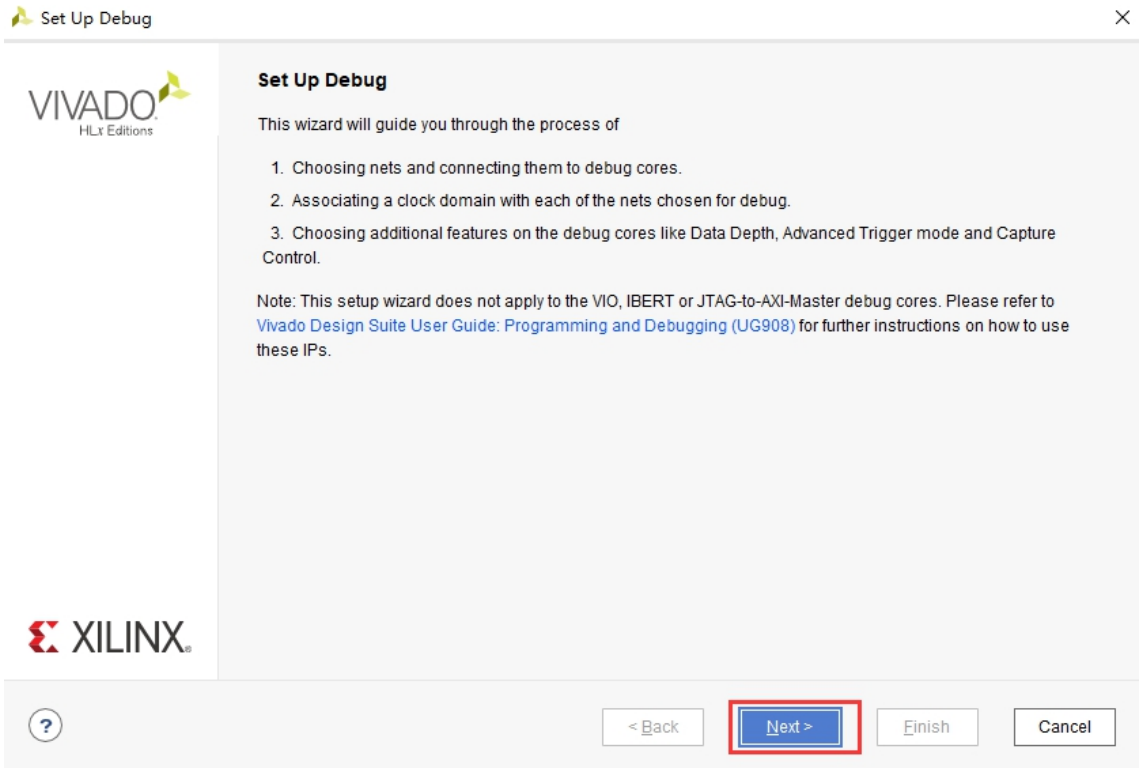
3) Click on Synthesis



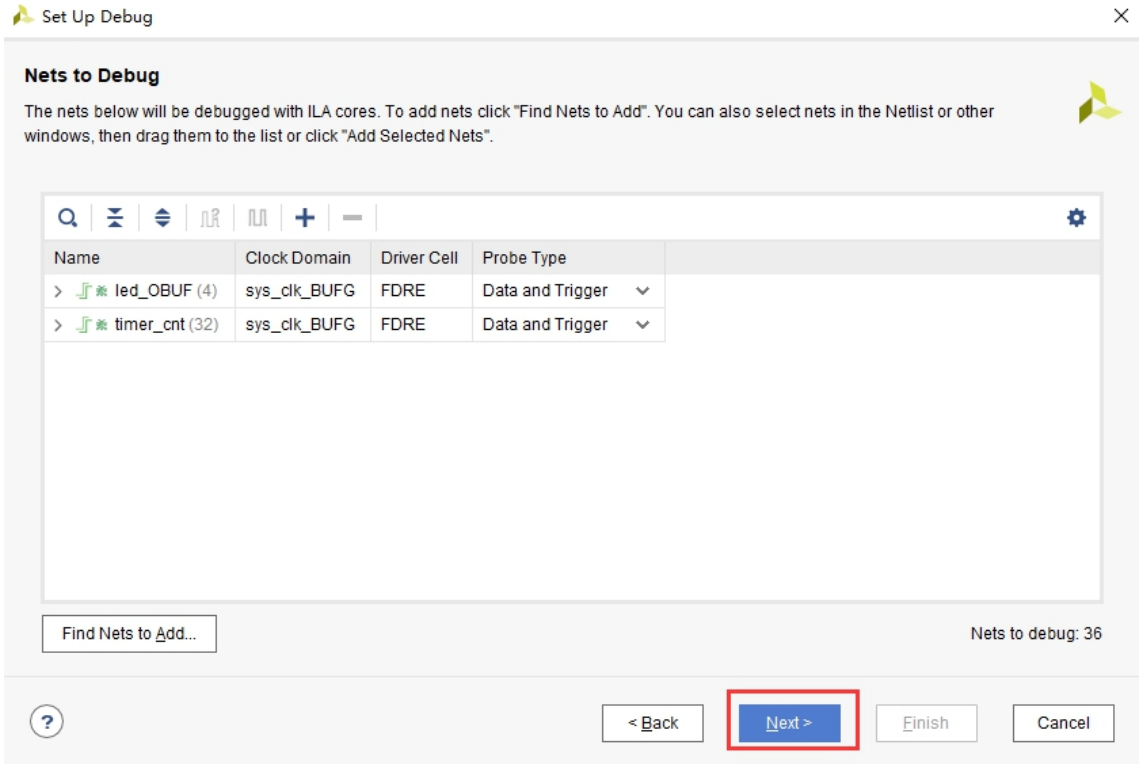
4) After the synthesis is finished, click “Set Up Debug”



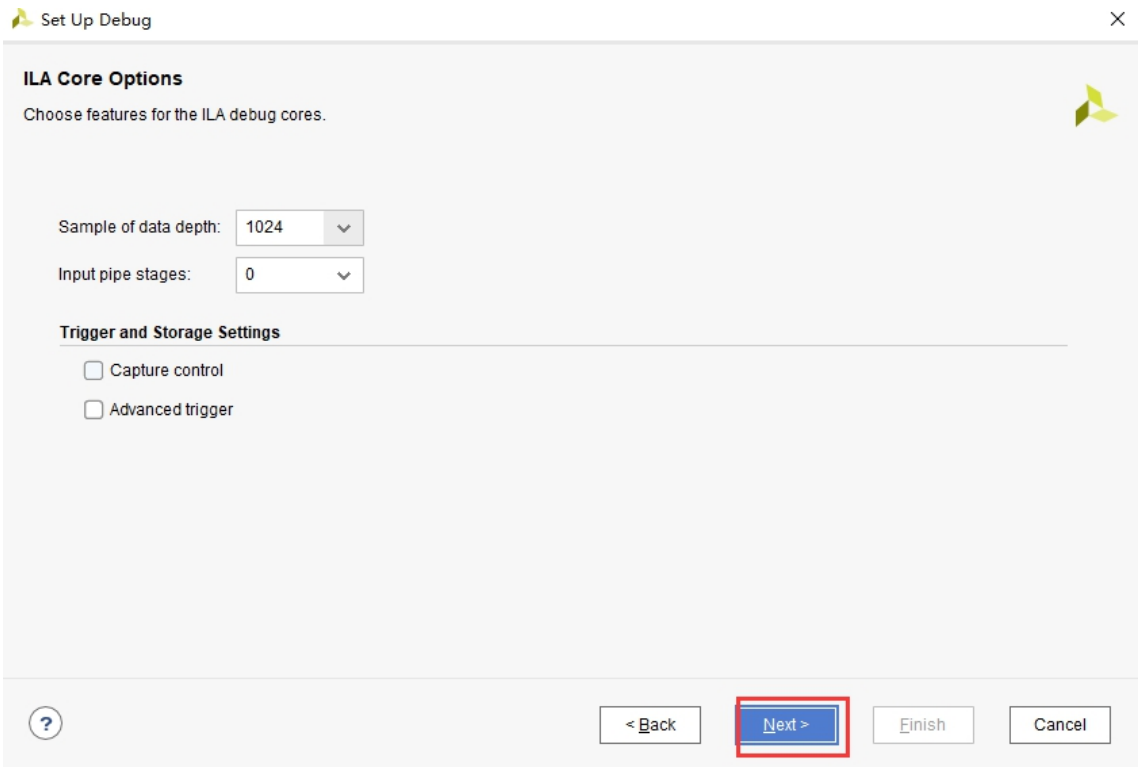
5) Pop up the window and click “Next”



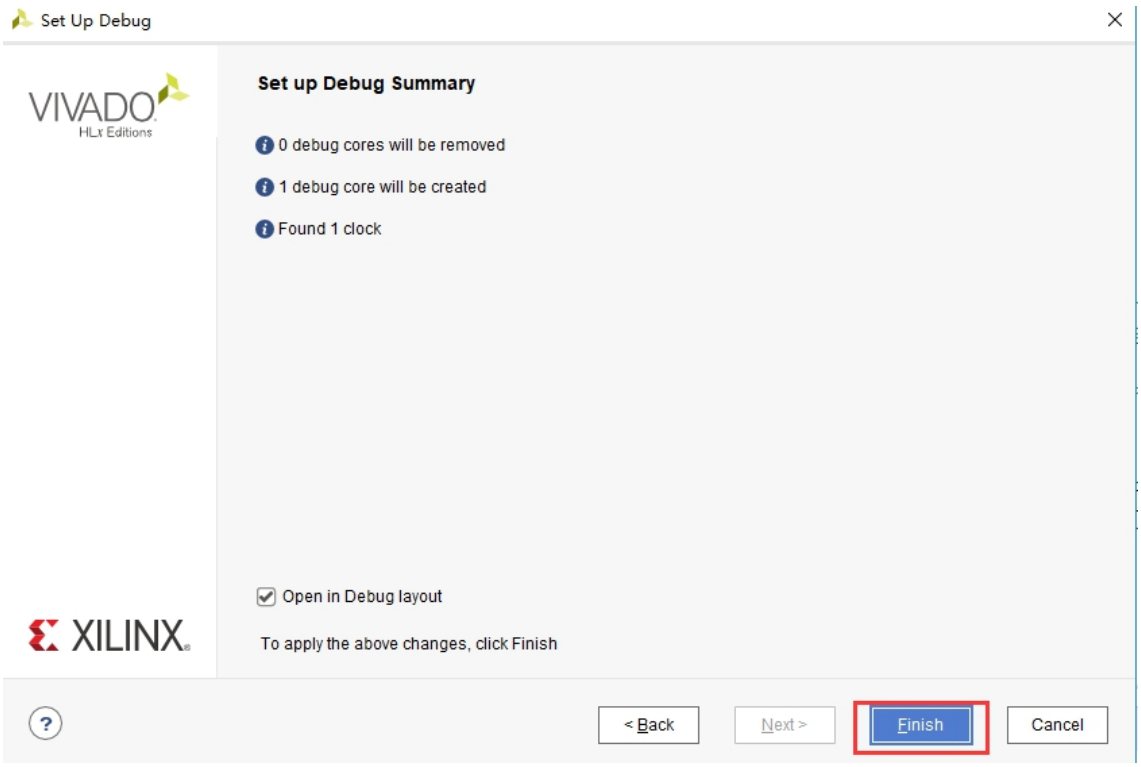
Click Next by default



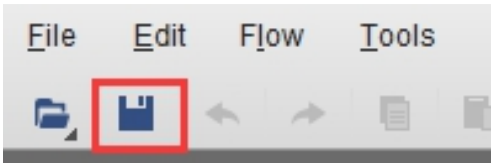
Sample depth window, select “Next”



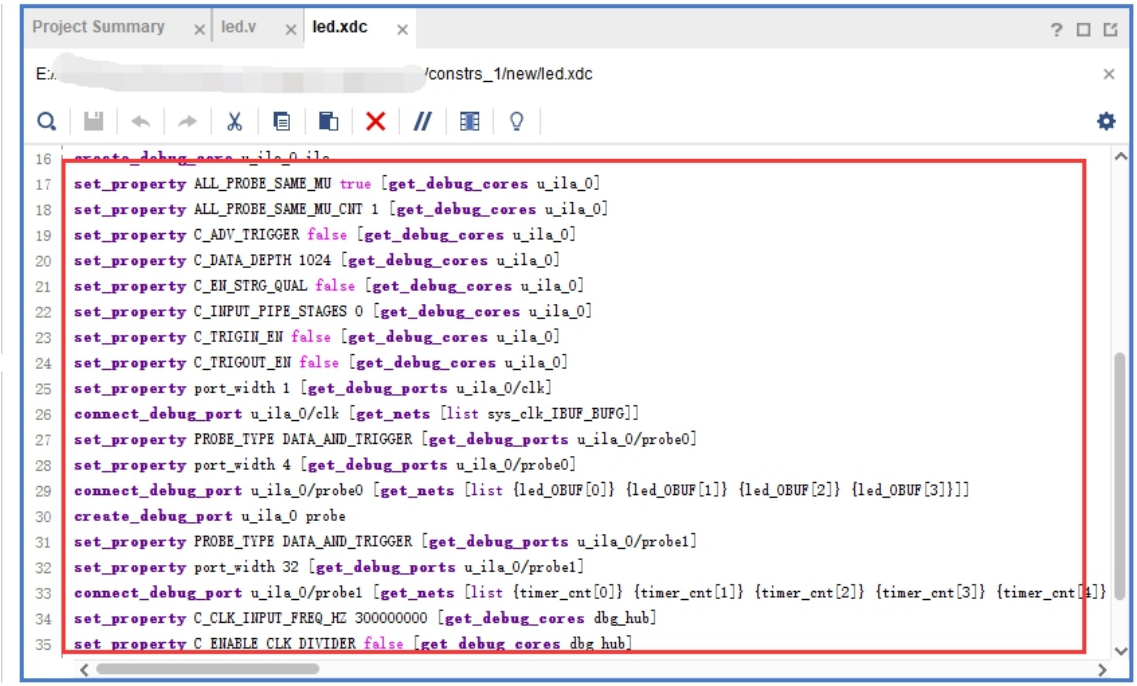
Click “Finish”



Click Save



In the “xdc” file, you can see the added “ila” core constraint.



6) Regeneration bitstream

7) The debugging method is the same as before and will not be described again.

Part 4.10: Experimental Summary

This chapter describes how to develop programs on the PL side, including create a project, constraints, simulation, online debugging, etc., in the subsequent code development methods can refer to this method.

Part 5: PLL Experiment Under Vivado

The experimental Vivado project is "pll_test".

Many beginners are confused when they see that there is only one 200Mhz clock input on the FPGA development board. How can the clock be 200Mhz? What if I want to work at 100Mhz or 150Mhz? In fact, many FPGA chips have integrated PLLs. Other manufacturers may not call them PLLs, but they also have similar functional modules. The PLL can multiply and divide the frequency to generate many other clocks. This experiment learns how to use PLL and vivado's IP core by calling PLL IP core.

Part 5.1: Experimental Principle

PLL, the phase locked loop. It is an important resource in FPGA. Because a complex FPGA system often requires multiple clock signals of different frequencies and phases. Therefore, the number of PLLs in an FPGA chip is an important indicator to measure the capabilities of the FPGA chip. In the design of FPGA, the high-speed design of the clock system FPGA is extremely important. A system clock with low jitter and low delay will increase the success rate of FPGA design.

This experiment will use the PLL to output a square wave to the expansion port on the FPGA development board to show you how to use the PLL in the Vivado software.

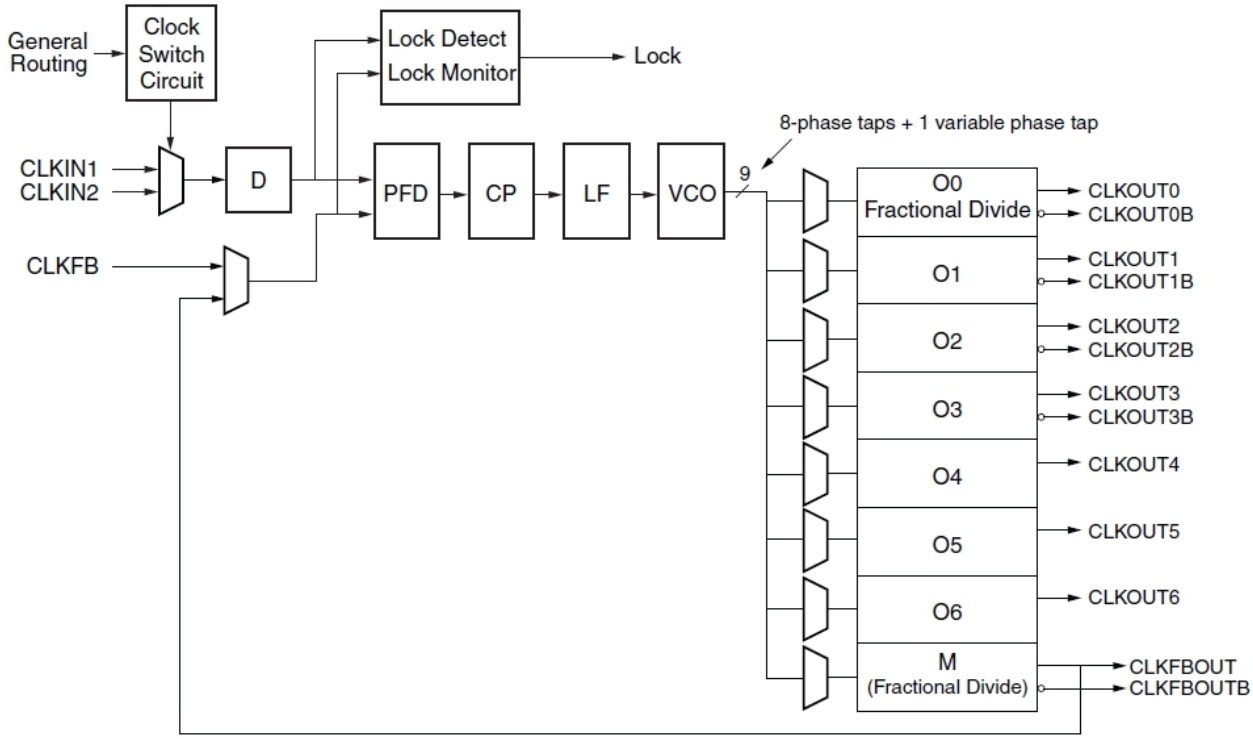
The Ultrascale+ series FPGAs use dedicated global and regional IO and clock resources to manage various clock requirements in the design. Clock Management Tiles (CMT) provide clock frequency synthesis, deskew, and jitter filtering.

Each CMTs contains a MMCM (mixed mode clock manager) and a PLL. As shown in the figure below, the input of CMT can be BUFR, IBUFG, BUFG, GT, BUFH, local wiring (not recommended), and the output needs to be connected to BUFG or BUFH before use.

➢ Mixed Mode Clock Manager (MMCM)

MMCM is used to generate different clock signals when there is a set phase and frequency relationship with a given input clock. MMCM provides extensive and powerful clock management functions.

The functional block diagram inside MMCM is shown below



➢ Digital Phase Locked Loop PLL

Phase-locked loop (PLL) is mainly used for frequency synthesis. Using one PLL can generate multiple clock signals from one input clock signal.

The functional block diagram inside the PLL is shown in the figure below

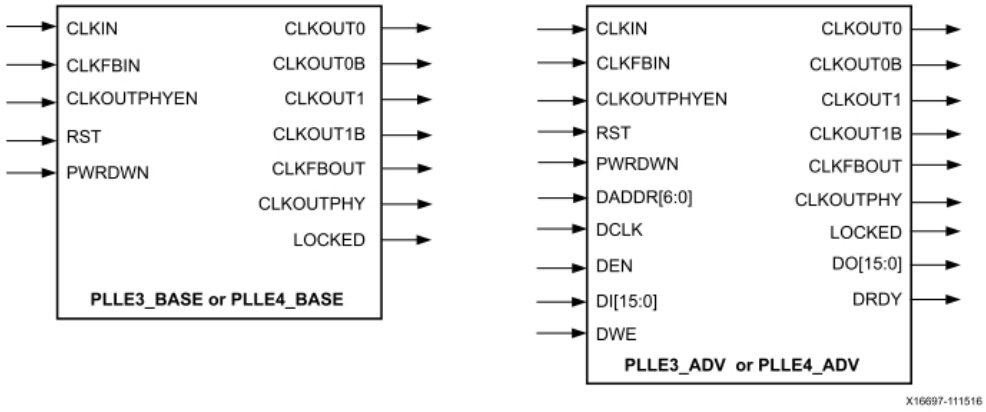
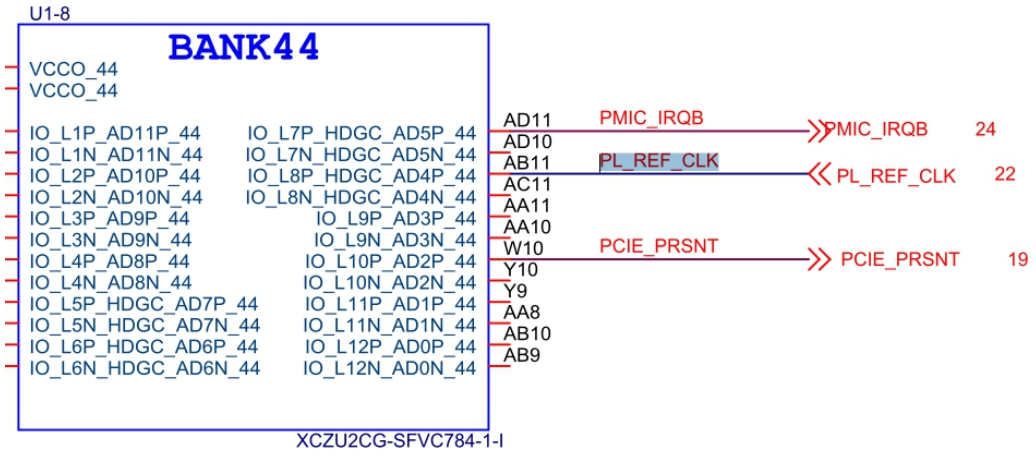


Figure 3-16: **PLL Primitives**

To learn more about clocking resources, I suggest you take a look at the "7 Series FPGAs Clocking Resources User Guide" provided by Xilinx.

Part 5.2: Create Vivado Project

In this experiment, I will show you how to use the PLL IP core provided by Xilinx to generate clocks of different frequencies, and output one of the clocks to the external IO of the FPGA. The following is the detailed procedure for program design. Before creating the PLL IP, there is one point that needs to be mentioned. In the schematic, you can see PL_REF_CLK, which is the 25MHz reference clock. It is in BANK44 and belongs to HDGC.

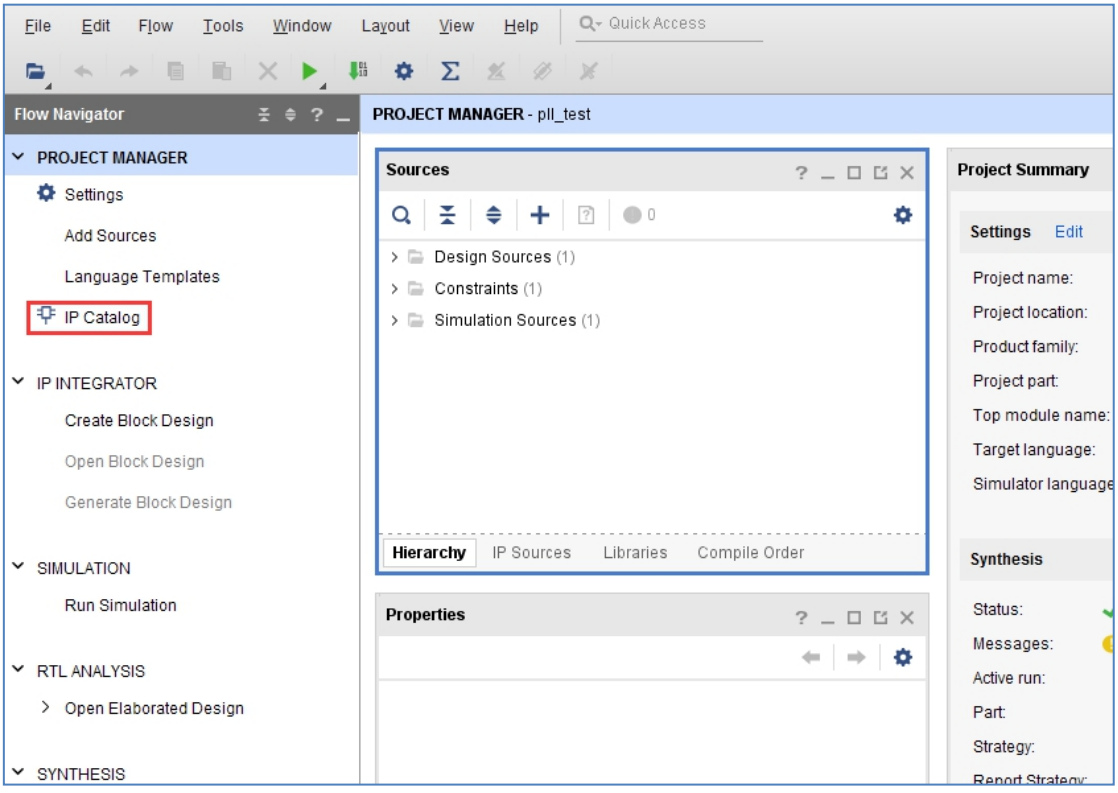


In the ug572 document, it is mentioned that the HDGC pin cannot be directly connected to MMCMs/PLLs. It needs to go through BUFG and then connect to MMCMs/PLLs. This need to pay attention.

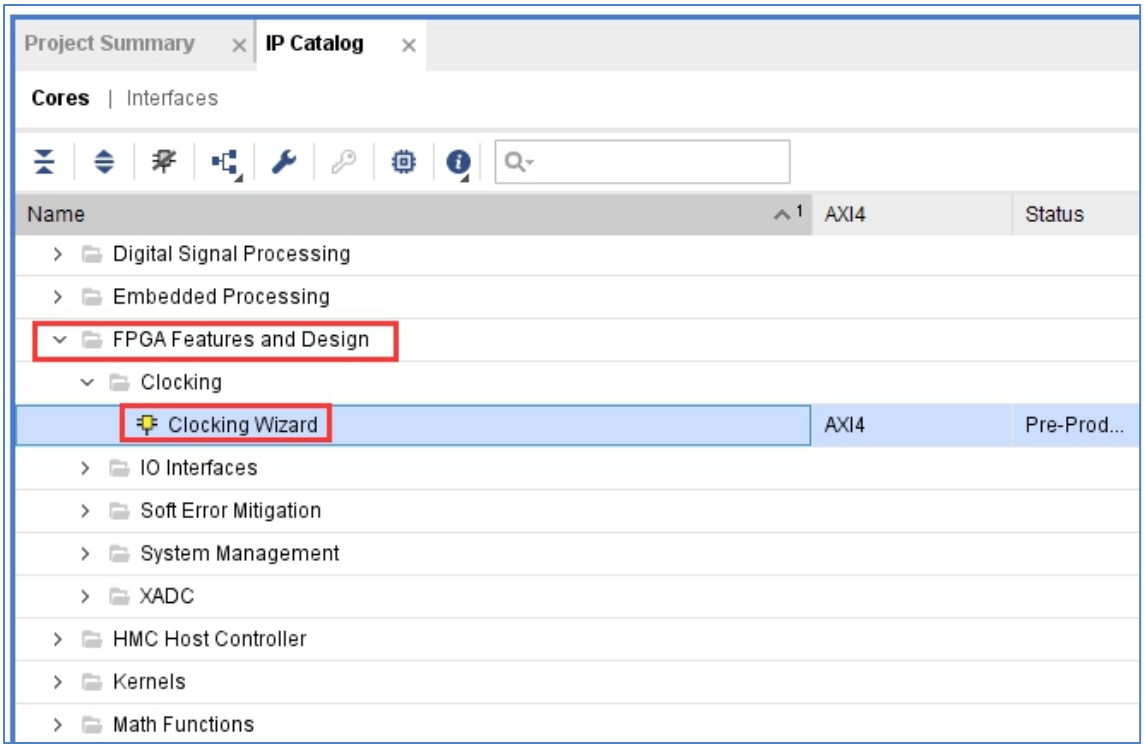
Global Clock Inputs

External global user clocks must be brought into the UltraScale device on differential clock pin pairs called global clock (GC) inputs. There are four GC pin pairs in each bank that have direct access to the global clock buffers, MMCMs, and PLLs that are in the CMT adjacent to the same I/O bank. The UltraScale+ architecture has four HDGC pins per HD I/O bank. HD I/O banks are only part of the UltraScale+ family. Since HD I/O banks do not have a XIPHY and CMT next to them, the HDGC pins can only directly drive BUFGCEs (BUFGs) and not MMCMs/PLLs. Therefore, clocks that are connected to an HDGC pin can only connect to MMCMs/PLLs through the BUFGCEs. To avoid a design rule check (DRC) error, set the property CLOCK_DEDICATED_ROUTE = FALSE. GC inputs provide dedicated, high-speed access to the internal global and regional clock resources. GC inputs use dedicated routing and must be used for clock inputs where the timing of various clocking features is imperative. General-purpose I/O with local interconnects should not be used for clock signals.

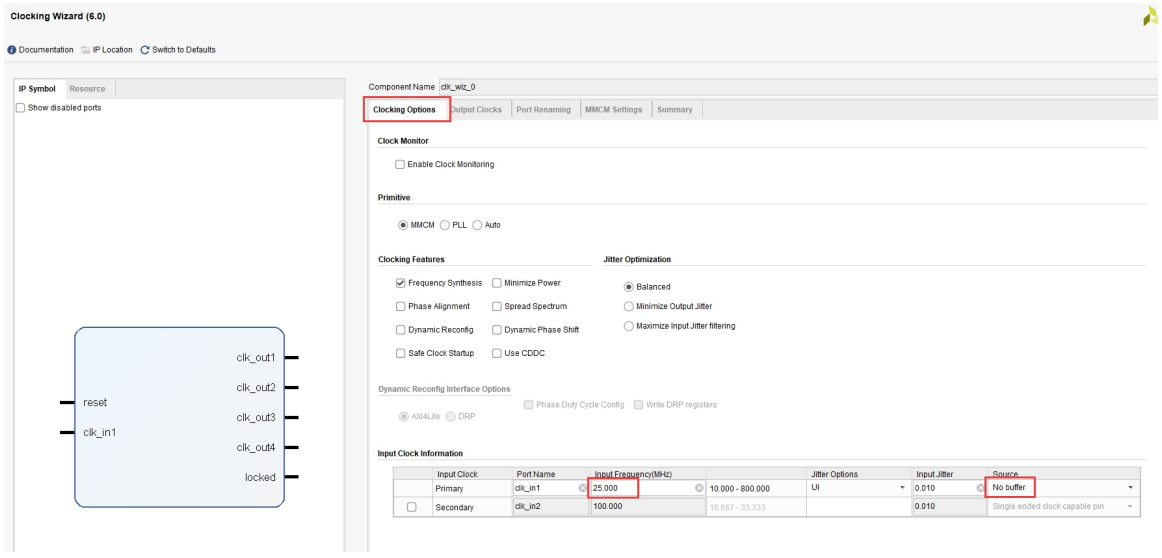
1) Create a new “pll_test” project and click “IP Catalog” under the “Project Manager” interface.



2) Then select “Clocking Wizard” under “FPGA Features and Design Clocking” in the “IP Catalog” interface, and double-click to open the configuration interface.

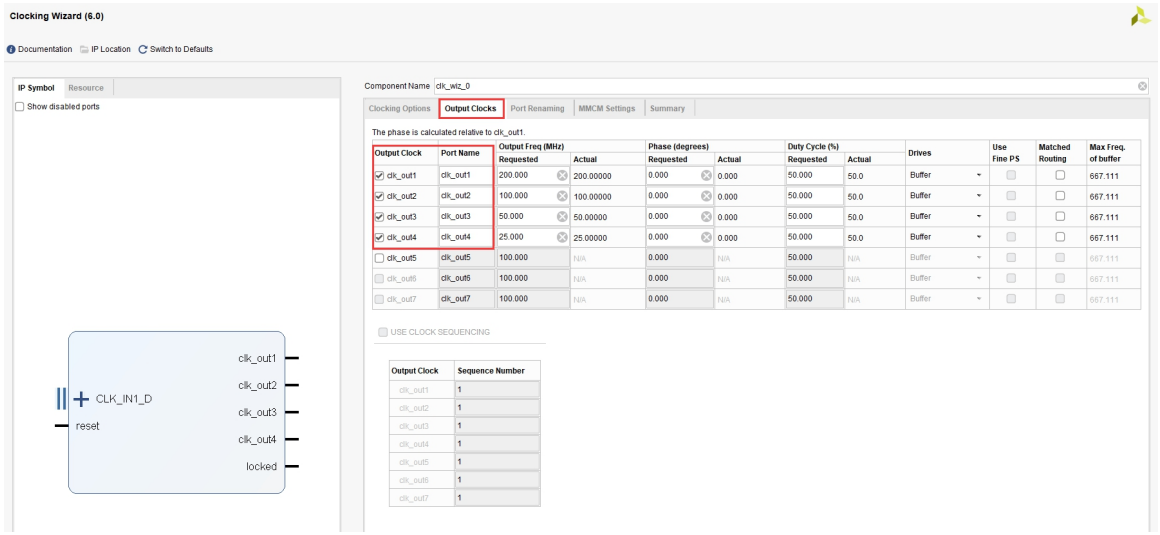


3) The default name of this “Clocking Wizard” is “clk_wiz_0”, we will not modify it here. In the first interface, “Clocking Options”, The input clock frequency is 25Mhz, and select "No buffer",that is to connect a BUFG before the PLL

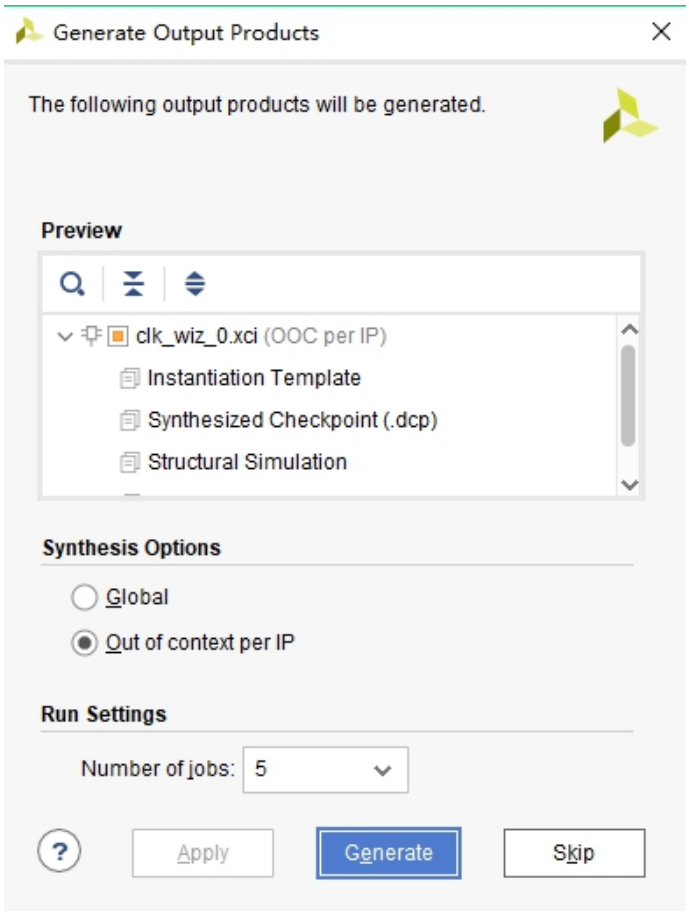


4) In the “Output Clocks” interface, select the output of the four clocks “clk_out1~clk_out4”, the frequencies are respectively “200Mhz”, “100Mhz”, “50Mhz”, “25Mhz”. Here you can also set the phase of the clock output, we will not do the setting, keep the default phase

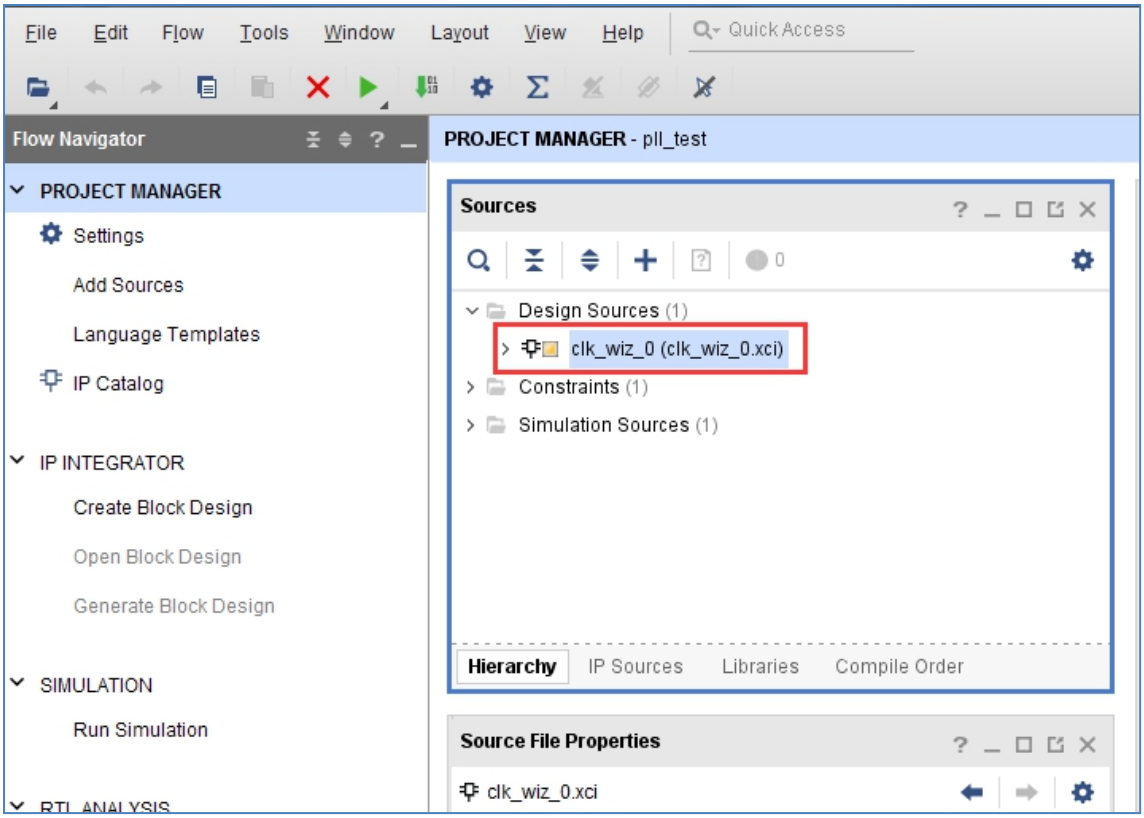
point, click OK to complete.



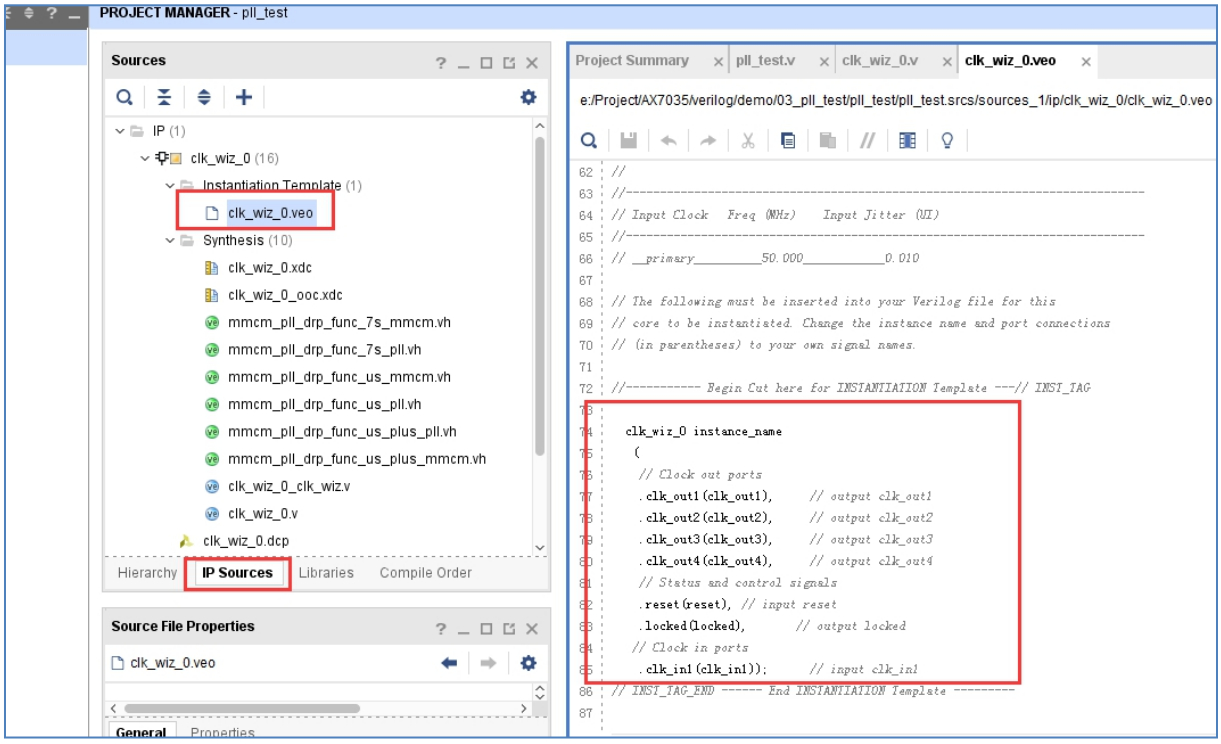
5) Click the Generate button in the pop-up dialog box to generate the "PLL IP" design file.



6) At this time, a "clk_wiz_0.xci" IP will be automatically added to our pll_test project, and the user can double-click it to modify the configuration of this IP.



Select the IP Sources page, and then double-click to open the "clk_wiz_0.veo" file, which provides an instantiation template for this IP. We only need to copy the contents of the box to our verilog program to instantiate the IP.



7) Let's write a top-level design file to instantiate this PLL IP, and write the "pll_test.v" code as follows. Note that the reset of the PLL is active at high level, that is, it is always in the reset state at high level, and the PLL will not work. Many novices will ignore this. Here we bind "rst_n" to a button, and the button is a low-level reset, so it needs to be reversely connected to the PLL reset.Insert a BUFG primitive in the program and connect it to the PLL.

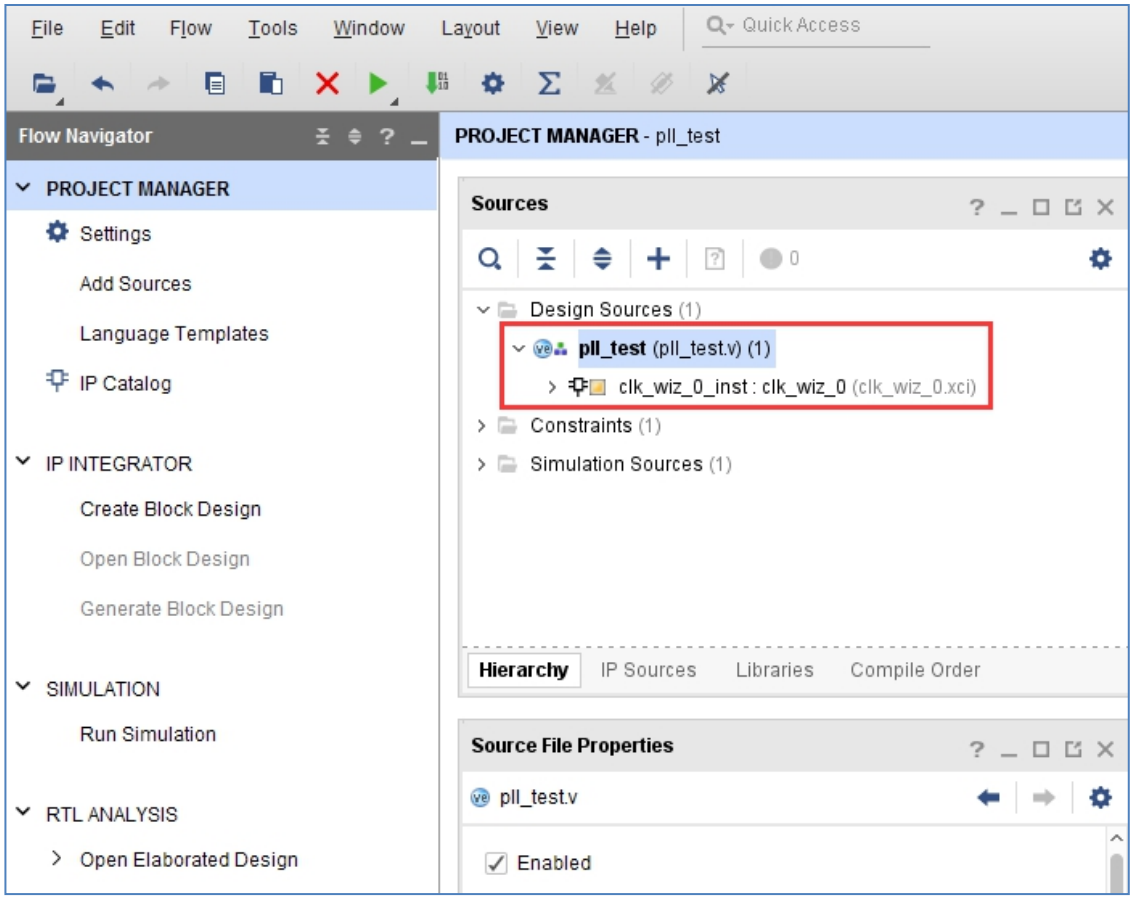
```
`timescale 1ns / 1ps
module pll_test(
 input      sys_clk,          //system clock 25Mhz on board
input       rst_n,           //reset ,low active
output      clk_out         //pll clock output

    );

wire        locked;

wire    sys_clkbuf ;

BUFG BUFG_inst (
   .O(sys_clkbuf), // 1-bit output: Clock output.
   .I(sys_clk)  // 1-bit input: Clock input.
);


//////////////////////PLL IP call//////////////////////////////
clk_wiz_0 clk_wiz_0_inst
   (// Clock in ports
    .clk_in1(sys_clkbuf),          // IN 25Mhz
    // Clock out ports
    .clk_out1(),              // OUT 200Mhz
    .clk_out2(),             // OUT 100Mhz
    .clk_out3(),            // OUT 50Mhz
    .clk_out4(clk_out),    // OUT 25Mhz
    // Status and control signals
    .reset(~rst_n),       // pll reset, high-active
    .locked(locked));     // OUT
endmodule
```

In the program, first instantiate clk_wiz_0, input the 25Mhz clock signal to clk_in1_p and clk_in1_n of clk_wiz_0, and assign the output of clk_out4 to clk_out.

Note: The purpose of instantiation is to call the instantiated module in the upper-level module to complete the code function. The format of the instantiation signal in Verilog is as follows: The module name must be the same as the module name to be instantiated, such as clk_wiz_0 in the program. Including module signal names must also be consistent, such as clk_in1, clk_out1, clk_out 2..... The connection signal is the signal transmitted between the TOP program and the module. The connection signal between the module and the module cannot conflict with each other, otherwise a compilation error will occur.

```
Module name                Extension name
(
.Module Signal 1,      (Connection Signal 1)
.Module Signal 2,      (Connection Signal 2)
.Module Signal 3,      (Connection Signal 3)
.....                      .....
.Module Signal N,      (Connection Signal N)
);
```

8) After saving the project, “pll_test” automatically becomes the top file, and “clk_wiz_0” becomes the submodule of the “Pll_test” file.



9) Then add the xdc pin constraint file “pll.xdc” to the project. Refer to PL's "Hello World" LED Experiment for the method to add, or you can directly copy the following content. And compile and generate bitstream.

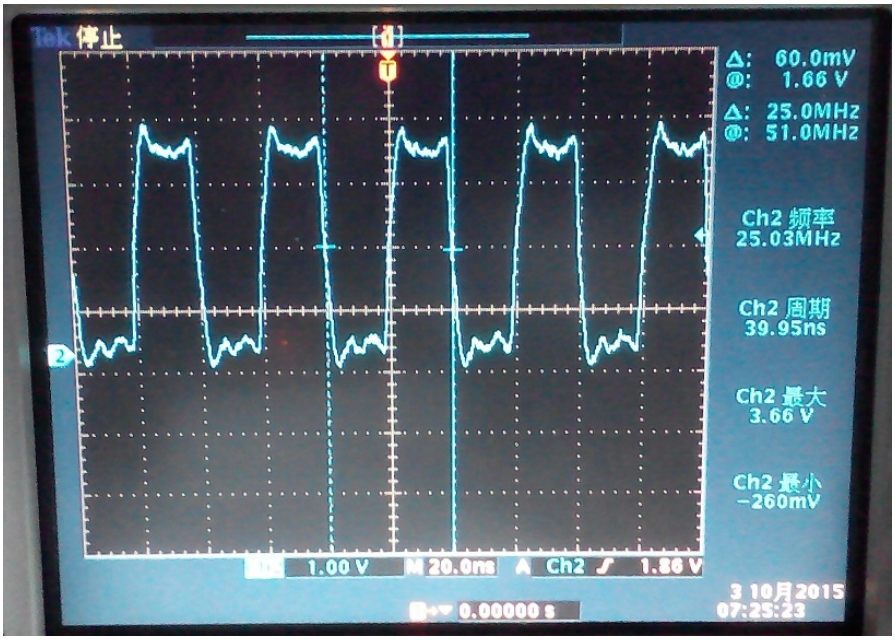
```
##################Compress Bitstream#################################
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]

set_property PACKAGE_PIN AE5 [get_ports sys_clk_p]
set_property IOSTANDARD DIFF_SSTL12 [get_ports sys_clk_p]

create_clock -period 5.000 -name sys_clk_p -waveform {0.000 2.500}
[get_ports sys_clk_p]

set_property PACKAGE_PIN AF12 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]

set_property PACKAGE_PIN AG11 [get_ports clk_out]
set_property IOSTANDARD LVCMOS33 [get_ports clk_out]
```

Part 5.3: On-board Verification

Compile the project and generate the pll_test.bit file, then download the bit file to the FPGA, and then we can use the oscilloscope to measure the output clock waveform.

Connect the ground wire of the oscilloscope probe to the ground of the FPGA development board (PIN1 of the expansion port J15 of the FPGA development board), and connect the signal terminal to the PIN3 pin of the expansion port 15 of the FPGA development board. It should be noted that when measuring, avoid the oscilloscope meter from touching other pins and causing a short circuit between the power supply and the ground.

At this time, we can see the 25Mhz clock waveform in the oscilloscope, the amplitude of the waveform is 3.3V, the duty cycle is 1:1, and the waveform display is as shown in the figure below:



If you want to output waveforms of other frequencies, you can modify the output of the clock to clk_wiz_0, clk_out2, clk_out3, clk_out4. You can also modify the clk_out4 of clk_wiz_0 to the frequency you want. You also need to pay attention here, because the

output of the clock is obtained by the PLL multiplying and dividing the input clock signal, so not all clock frequencies are available. The PLL can be accurately generated, but the PLL will also automatically calculate the actual output clock frequency for you.

In addition, it should be noted that the bandwidth and sampling rate of some users' oscilloscopes are too low, which will cause the high-frequency part attenuation to be too large when measuring high-frequency clock signals, and the amplitude of the measured waveform will be lower.

Part 6: FPGA on-chip RAM read and write test experiment

The experimental Vivado project is "ram _test".

RAM is a basic module commonly used in FPGAs and can be widely used for data buffering. It is also the basis of ROM and FIFO. This experiment will introduce how to use the RAM inside the FPGA and how to read and write data from the RAM by the program.

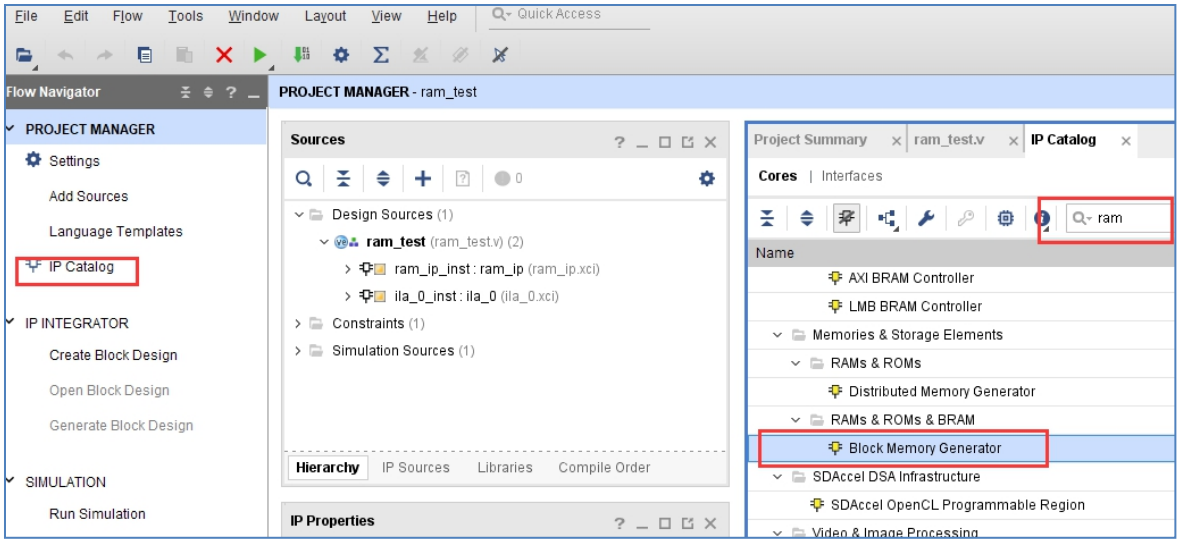
Part 6.1: Experimental Principle

Xilinx has already provided us with the IP core of RAM in Vivado, we only need to instantiate a RAM through the IP core to write and read the data stored in RAM according to the read and write timing of RAM. In the experiment, through the online logic analyzer ila integrated by Vivado, we can observe the read and write timing of RAM and the data read from RAM.

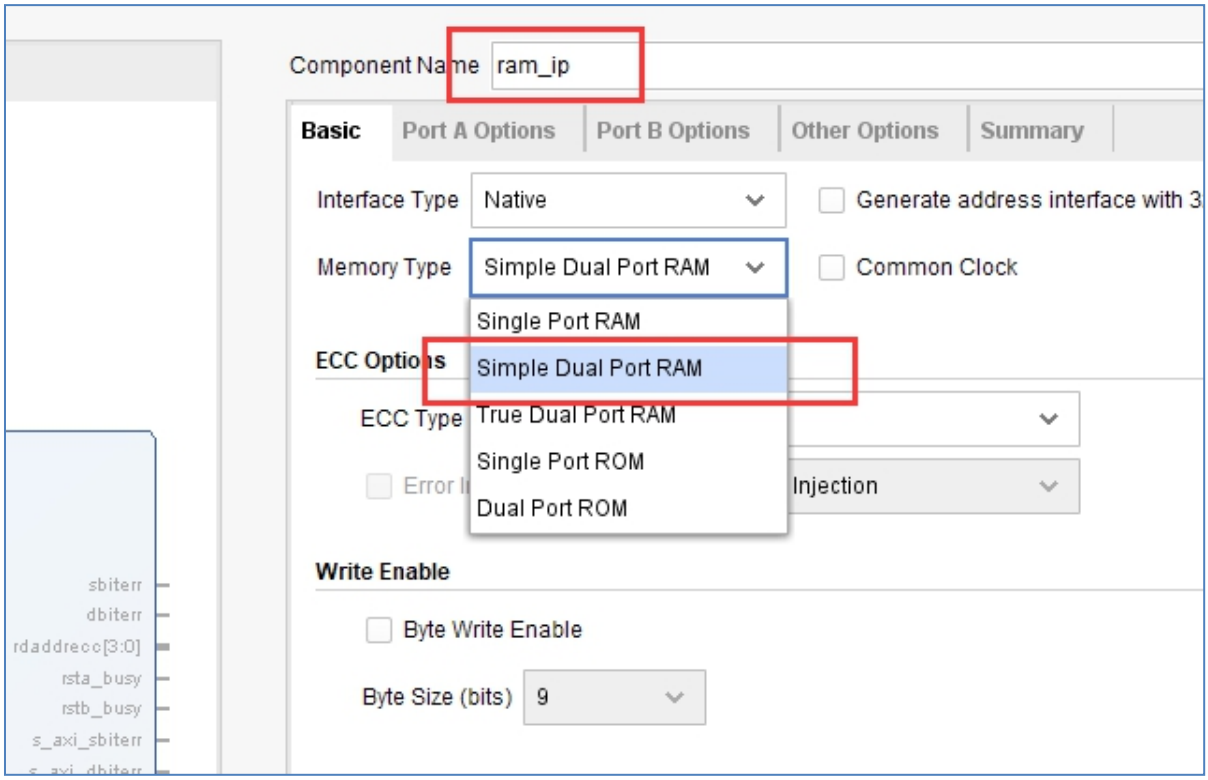
Part 6.2: Create Vivado project

Create a new “ram_test” project before adding “RAM IP”, and then add “RAM IP” to the project, the method is as follows

1) Click on the “IP Catalog” in the figure below, search for “ram” in the pop-up interface on the right, find “Block Memory Generator”, and double-click to open it.

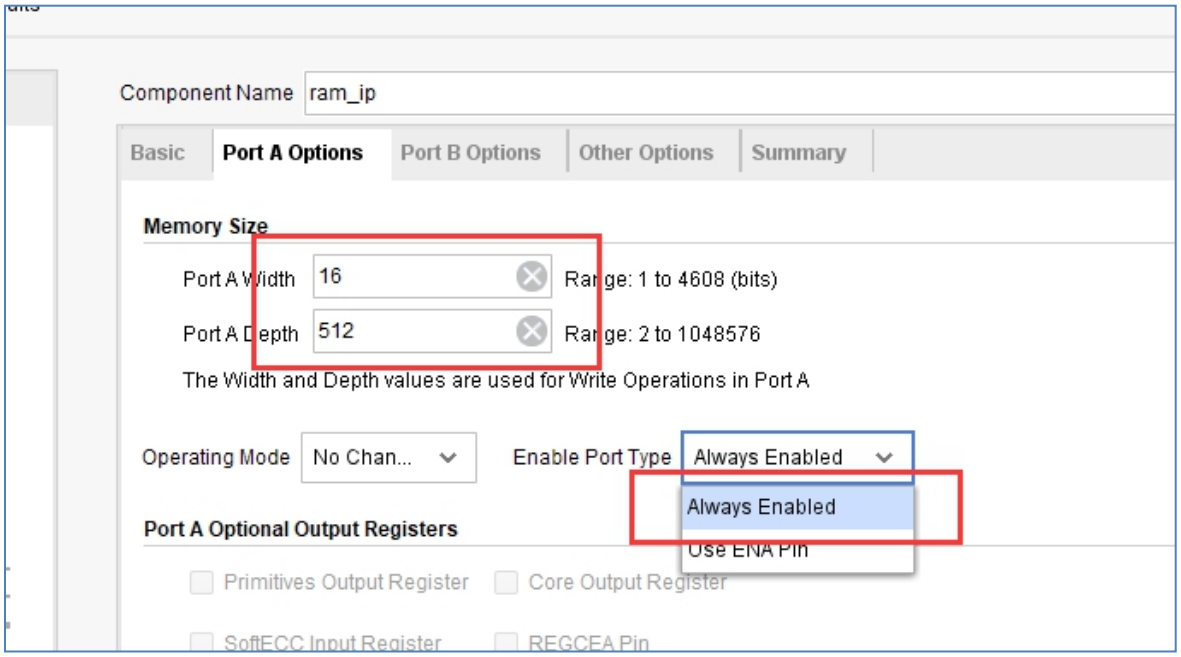


2) Change the “Component Name” to “ram_ip”, and under the “Basic” column, change the “Memory Type” to “Simple Dual Prot RAM”, which is pseudo dual-port RAM. Generally speaking, "Simple Dual Port RAM" is the most commonly used because it has two ports with independent input and output signals.

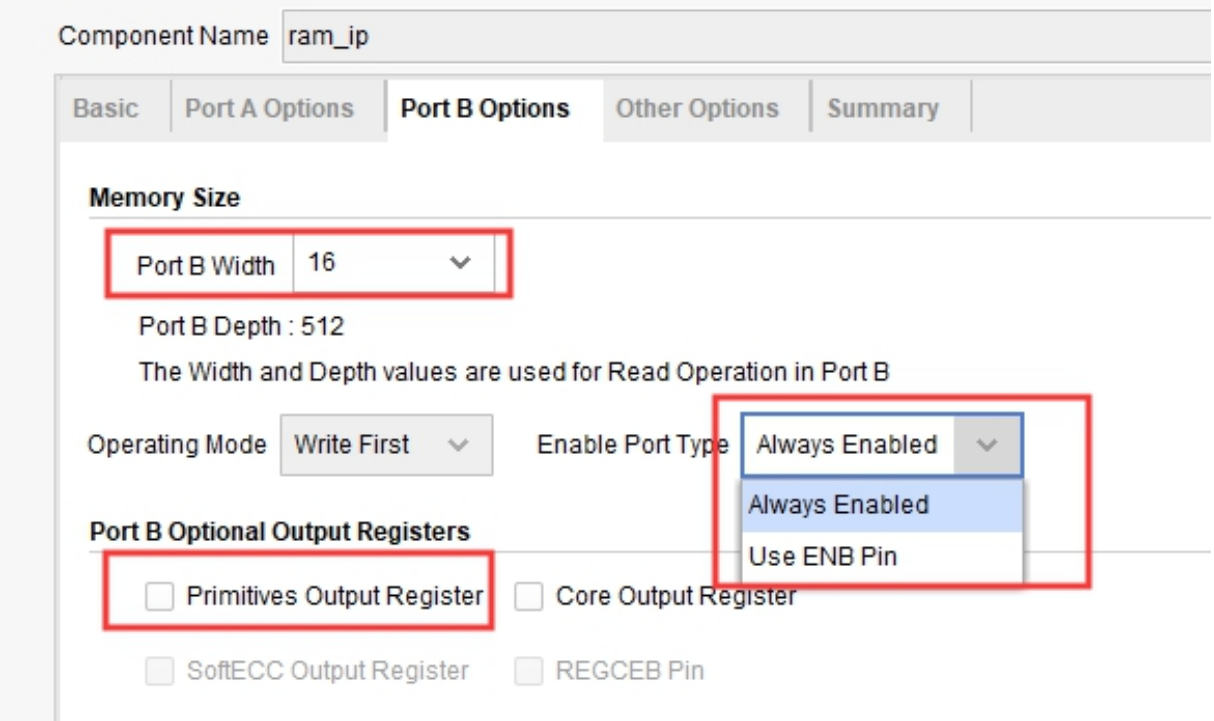


3) Switch to the “Port A Options” column, and change the RAM bit width “Port A Width” to 16, which is the data width. Change the RAM depth “Port A Depth” to 512. The depth refers to how many

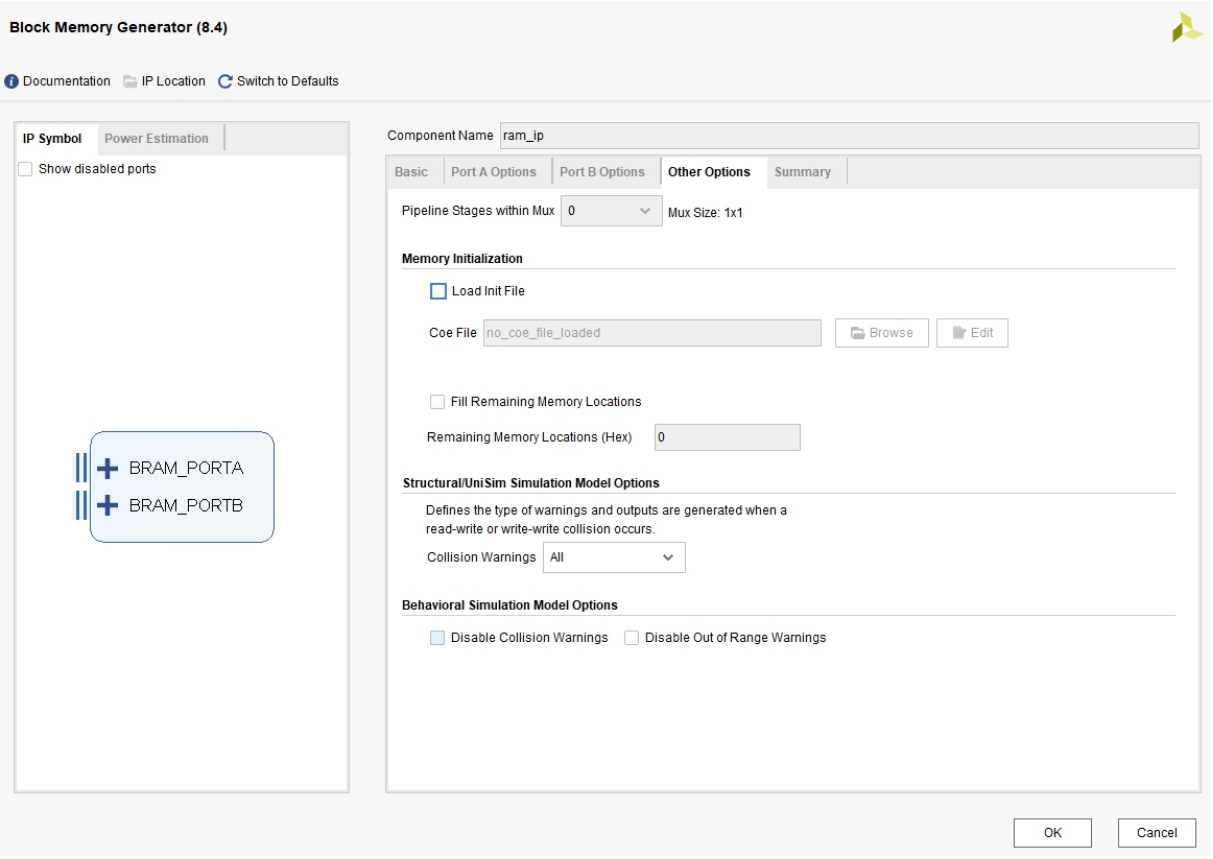
data can be stored in RAM. Change “Enable Port Type” to “Always Enable”.



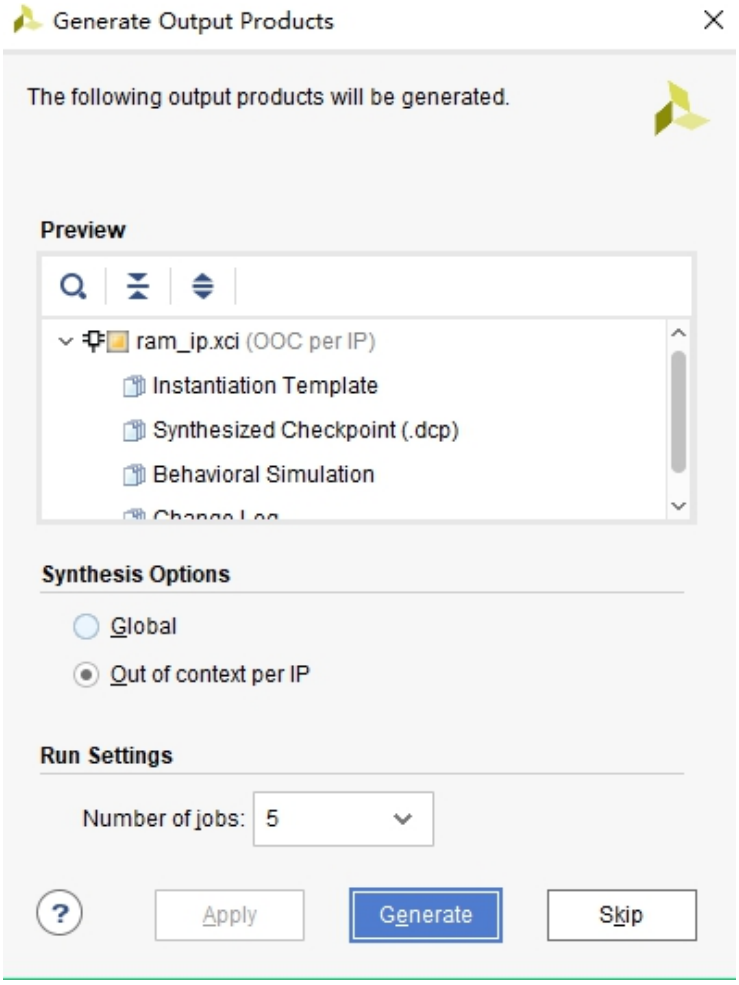
4) Switch to the “Port B Options” column, change the “Port B Width” to 16, and the “Enable Port Type” to “Always Enable”. Of course, you can also “Use ENB Pin”, which is equivalent to reading the enable signal. The “Primitives Output Register” is unchecked, and its function is to add registers to the output data, which can effectively improve the timing, but the read data will lag behind the address by two cycles. In many cases, this function is disabled and the data is kept behind the address by one cycle.



5) In the “Other Options” column, there is no need to initialize RAM data like ROM, we can write in the program, so the default configuration is enough, just click OK.



6) Click "Generate" to generate RAM IP



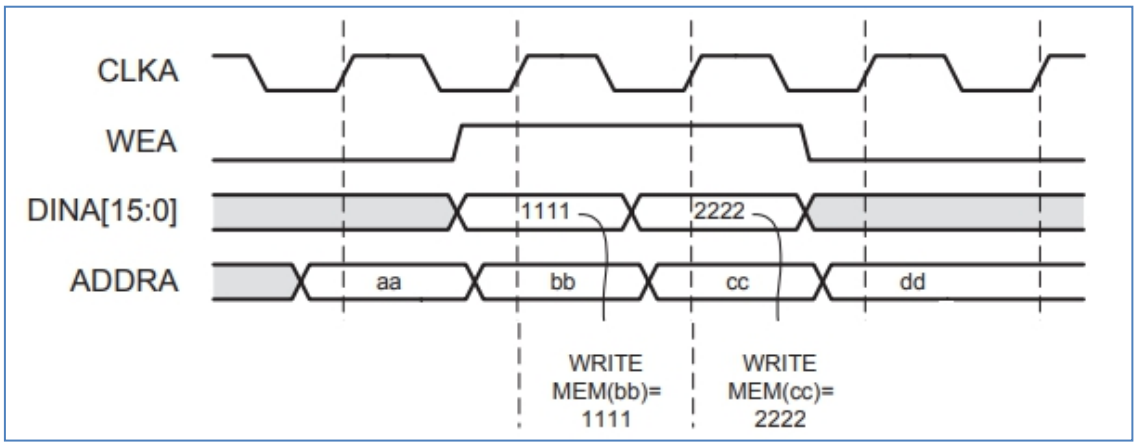
Part 6.3: RAM port definition and timing

The description of “Simple Dual Port RAM” module ports is as follows:

Signal Name	Direction	Description
clka	in	Port A clock input
wea	in	Port A enable
addra	in	Port A address input
dina	in	Port A data input
clkb	in	Port B clock input
addrb	in	Port B address input
doutb	out	Port B data output

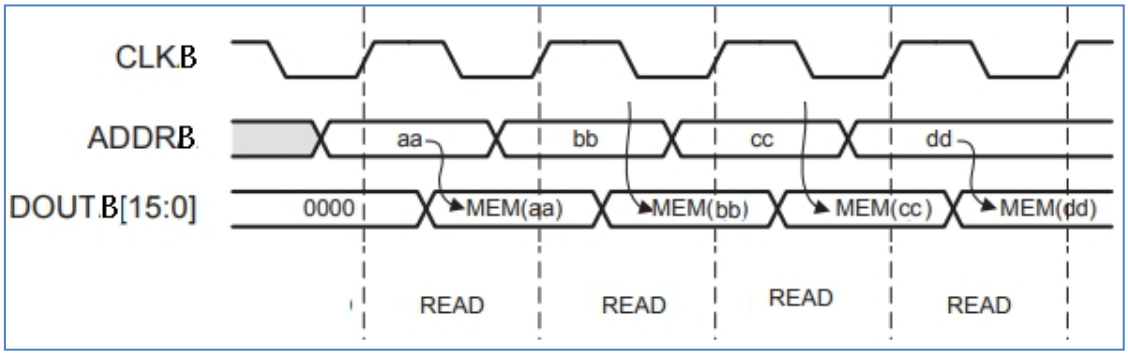
RAM data writing and reading are operated on the rising edge of

the clock. When port A data is written, the wea signal needs to be set high and the address and data to be written are provided at the same time. The following figure shows the timing chart of input writing to RAM.



RAM Write Timing

The port B cannot write data, it can only read data from RAM, as long as the address is provided, under normal circumstances, valid data can be collected in the next cycle.



RAM Read Timing

Part 6.4: Test program writing

Next, we will write the RAM test program. Due to the function of testing RAM, we write a series of continuous data to port A of RAM, write it only once, and read it from port B, and use a logic analyzer to view the data. The code as below:

```
`timescale 1ns / 1ps
```

```
//////////////////////////////////////////////////////////////////////////////////
//////////
module ram_test(
           input      sys_clk_p,          //system clock 200Mhz on board
           input      sys_clk_n,          //system clock 200Mhz on board
           input rst_n                   //Reset signal, active low
       );

//-----------------------------------------------------------
reg     [8:0]      w_addr;           //RAM PORTA Write Address
reg     [15:0]     w_data;           //RAM PORTA Write Data
reg                wea;       //RAM PORTA Enable
reg     [8:0]      r_addr;           //RAM PORTB Read Address
wire    [15:0]     r_data;           //RAM PORTB Read Data

wire clk ;

   IBUFDS IBUFDS_inst (
      .O(clk),  // Buffer output
      .I(sys_clk_p),  // Diff_p buffer input (connect directly to top-level
port)
      .IB(sys_clk_n) // Diff_n buffer input (connect directly to top-level
port)
   );

//Generate RAM PORTB read address
always @(posedge clk or negedge rst_n)
begin
  if(!rst_n)
    r_addr <= 9'd0;
  else if (|w_addr)          // w_addr bit or, not equal to 0
    r_addr <= r_addr+1'b1;
  else
    r_addr <= 9'd0;
end

//Genarate RAM PORTA Write Enable Signal
always@(posedge clk or negedge rst_n)
begin
  if(!rst_n)
      wea <= 1'b0;
```

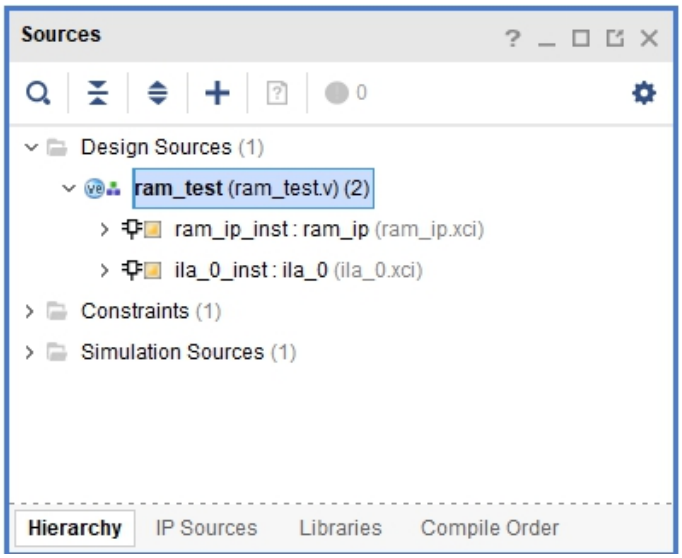
```
  else
  begin
     if(&w_addr)          //The bits of w_addr are all 1, a total of 512 data
are written, and the writing is complete
        wea <= 1'b0;
     else
        wea <= 1'b1;       //ram write enable
  end
end

//Generate RAM PORTA write address and data
always@(posedge clk or negedge rst_n)
begin
  if(!rst_n)
  begin
      w_addr <= 9'd0;
      w_data <= 16'd1;
  end
  else
  begin
     if(wea)                  //ram write enable valid
     begin
        if (&w_addr)          //The bits of w_addr are all 1, a total of 512
data are written, and the writing is complete
        begin
            w_addr <= w_addr ;   //Keep the address and data values, write
to RAM only once
            w_data <= w_data ;
        end
        else
        begin
            w_addr <= w_addr + 1'b1;
            w_data <= w_data + 1'b1;
        end
     end
  end
end

//------------------------------------------------------------
//Instantiate RAM
ram_ip ram_ip_inst (
```

```
  .clka      (clk          ),     // input clka
  .wea       (wea          ),     // input [0 : 0] wea
  .addra     (w_addr       ),     // input [8 : 0] addra
  .dina      (w_data       ),     // input [15 : 0] dina
  .clkb      (clk          ),     // input clkb
  .addrb     (r_addr       ),     // input [8 : 0] addrb
  .doutb     (r_data       )      // output [15 : 0] doutb
);
// Instantiate the ila logic analyzer
ila_0 ila_0_inst (
    .clk    (clk    ),
    .probe0    (r_data    ),
    .probe1    (r_addr    )
);

endmodule
```

In order to see the data value read in RAM in real time, we have added the ila tool here to observe the data signal and address signal of RAM PORTB. Please refer to PL's "Hello World" LED experiment on how to generate ila.

```
//实例化ila逻辑分析仪
ila_0 ila_0_inst (
    .clk    (clk    ),
    .probe0 (r_data ),
    .probe1 (r_addr )
);
```

The program structure is as follows:

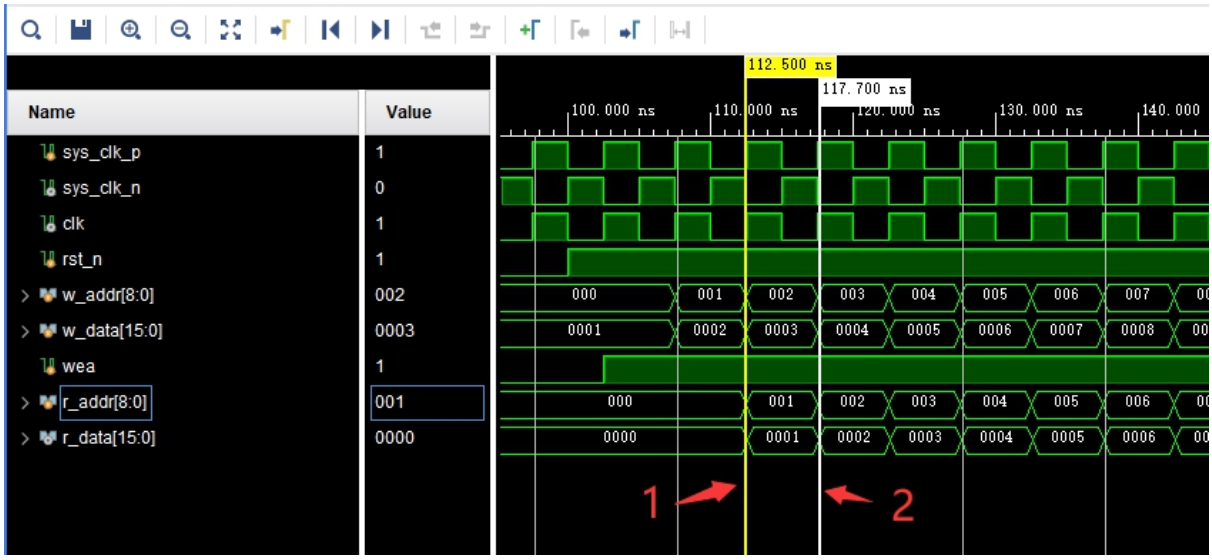


Binding pin

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

set_property PACKAGE_PIN AB11 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
create_clock -period 40.000 -name clk -waveform {0.000 20.000} [get_ports
clk]


set_property PACKAGE_PIN AA13 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
```

Part 6.5: Simulation

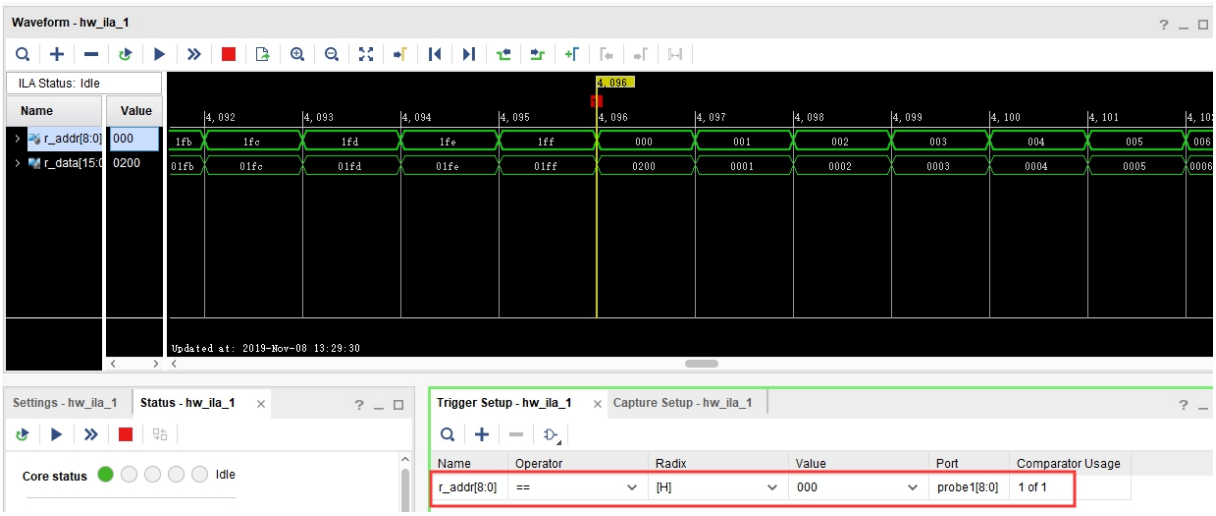
The simulation method refers to the PL's "Hello World" LED experiment. The simulation results are as follows. From the figure, it can be seen that the data written in address 1 is 0002. The valid data is read out in the next cycle, that is, time 2.



Part 6.6: On-board Verification

Generate bitstream and download bit file to FPGA. Next, we use ila to observe whether the data read from RAM is the data we initialized.

In the Waveform window, set the r_addr address to 0 as the trigger condition. We can see that r_addr is continuously accumulating from 0 to 1ff. As r_addr changes, r_data is also changing. The data of r_data is exactly the 512 data we write to RAM. It should be noted here that when a new address appears in r_addr, the data corresponding to r_data will only appear after a delay of two clock cycles. The data appears two clock cycles later than the address, which is consistent with the simulation results.


Waveform - hw_ila_1
ILA Status: Idle
Name
Value
r_addr[8:0]
000
r_data[15:0]
0200
4,096
4,092
4,093
4,094
4,095
4,096
4,097
4,098
4,099
4,100
4,101
1fb
1fc
1fd
1fe
1ff
000
001
002
003
004
005
01fb
01fc
01fd
01fe
01ff
0200
0001
0002
0003
0004
0005
Updated at: 2019-Nov-08 13:29:30
Settings - hw_ila_1
Status - hw_ila_1
Core status
Idle
Trigger Setup - hw_ila_1
Capture Setup - hw_ila_1
Name
Operator
Radix
Value
Port
Comparator Usage
r_addr[8:0]
==
[H]
000
probe1[8:0]
1 of 1

Part 7: FPGA on-chip ROM read and write test experiment

The experimental Vivado project is "rom _test".

FPGA itself is SRAM architecture. After power off, the program disappears. So how to use FPGA to realize a ROM? We can use the RAM resources inside FPGA to realize ROM, but it is not ROM in the true sense. Instead, the initialized value will be written into RAM every time it is powered on. This experiment will introduce how to use the ROM inside the FPGA and the program to read data from the ROM.

Part 7.1: Experimental Principle

Xilinx has provided us with an IP core of ROM in Vivado, we only need to instantiate a ROM through the IP core, and read the data stored in the ROM according to the read sequence of the ROM. In the experiment, we can observe the read timing of ROM and the data read from ROM through the online logic analyzer ila integrated by Vivado.

Part 7.2: Programming

Part 7.2.1: Create ROM initialization file

Since it is a ROM, we must prepare the data for it in advance, and then when the FPGA is actually running, we can directly read the pre-stored data in the ROM. The on-chip ROM of Xilinx FPGA supports initial data configuration. As shown in the figure below, we can create a file named rom_init.coe. Note that the suffix must be ". coe". Of course, the previous name can be changed at will.

The content format of the ROM initialization file is very simple, as shown in the figure below. The first line defines the data format, 16 represents the ROM data format is hexadecimal. From line 3 to line 34, it is the initialization data of this 32*8bit ROM. Use a comma after each line of numbers, and use a semicolon at the end of the last line of numbers.

rom_test.v rom_init.coe

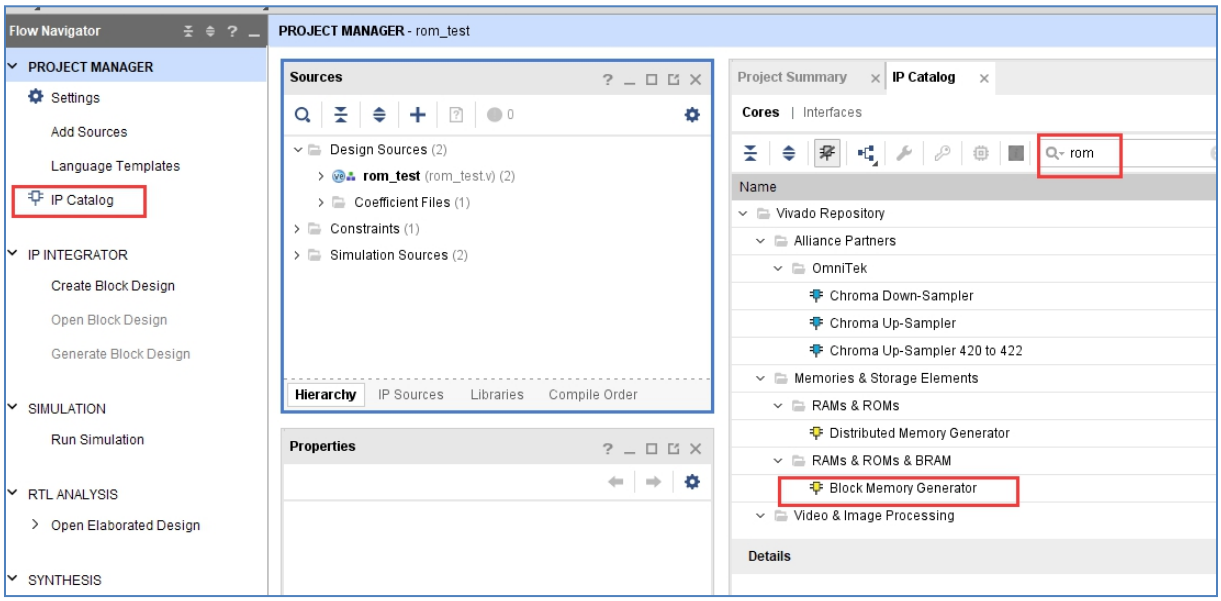
```
1  MEMORY_INITIALIZATION_RADIX=16;     //表示ROM内容的数据格式是16进制
2  MEMORY_INITIALIZATION_VECTOR=
3  11,
4  22,
5  33,
6  44,
7  55,
8  66,
9  77,
10 88,
11 99,
12 aa,
13 bb,
14 cc,
15 dd,
16 ee,
17 ff,
18 00,
19 a1,
20 a2,
21 a3,
22 a4,
23 a5,
24 a6,
25 a7,
26 a8,
27 b1,
28 b2,
29 b3,
30 b4,
31 b5,
32 b6,
33 b7,
34 b8;    //每个数据后面用逗号或者空格或者换行符隔开，最后一个数据后面加分号
35
```

After “rom_init.coe” is written, save it, and then we start to design and configure the ROM IP core.

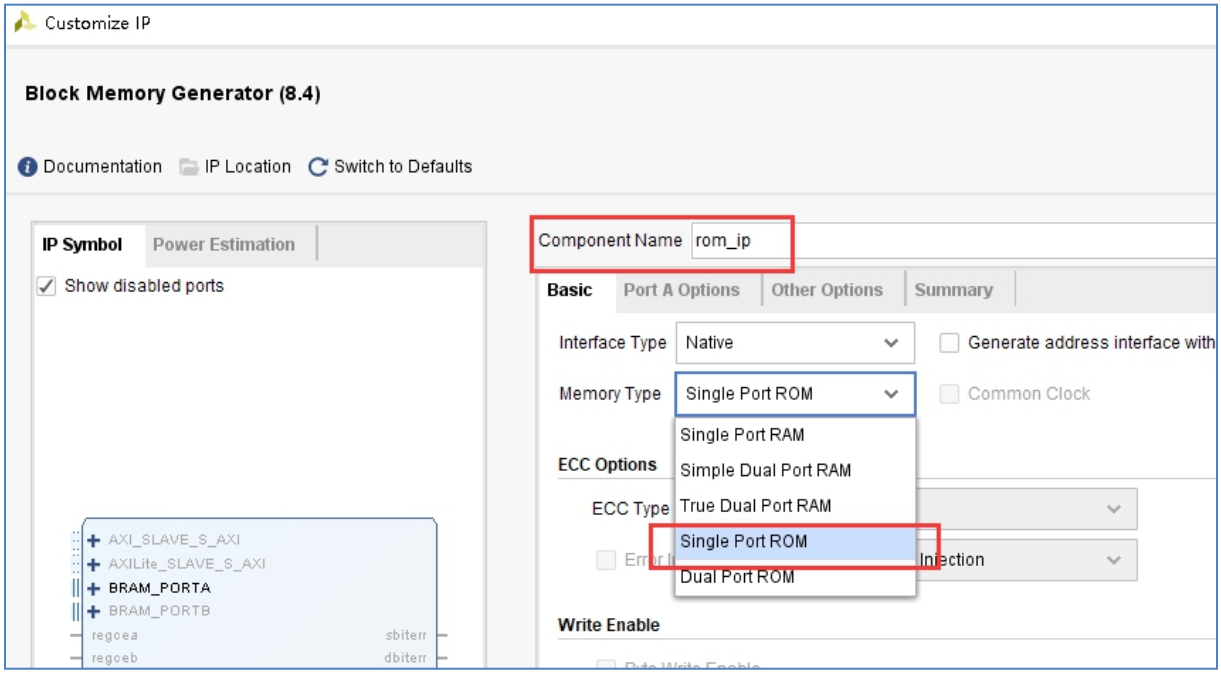
Part 7.2.2: Add ROM IP core

Create a new rom_test project before adding ROM IP, and then add ROM IP to the project, the method is as follows

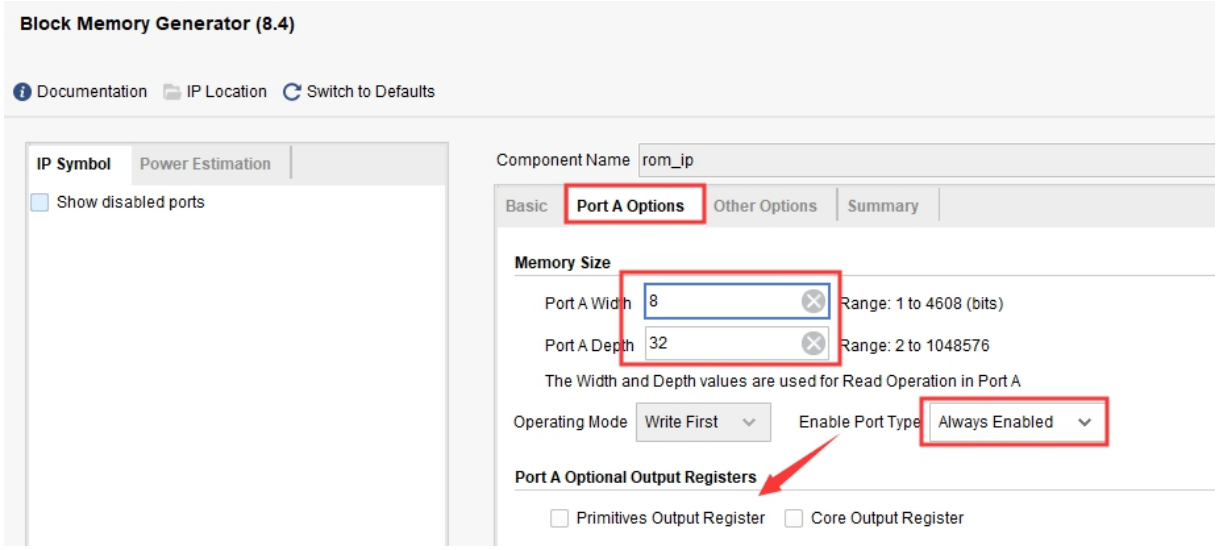
1) Click on “IP Catalog” in the figure below, search for “rom” in the pop-up interface on the right, find “Block Memory Generator”, and double-click to open it.



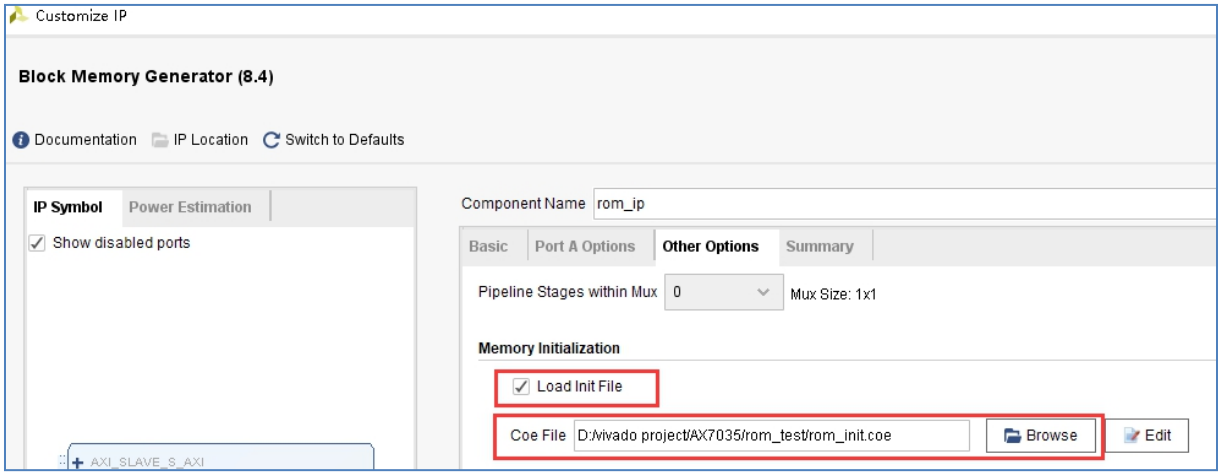
2) Change the “Component Name” to “rom_ip”, and under the Basic column, change the “Memory Type” to “Single Prot ROM”



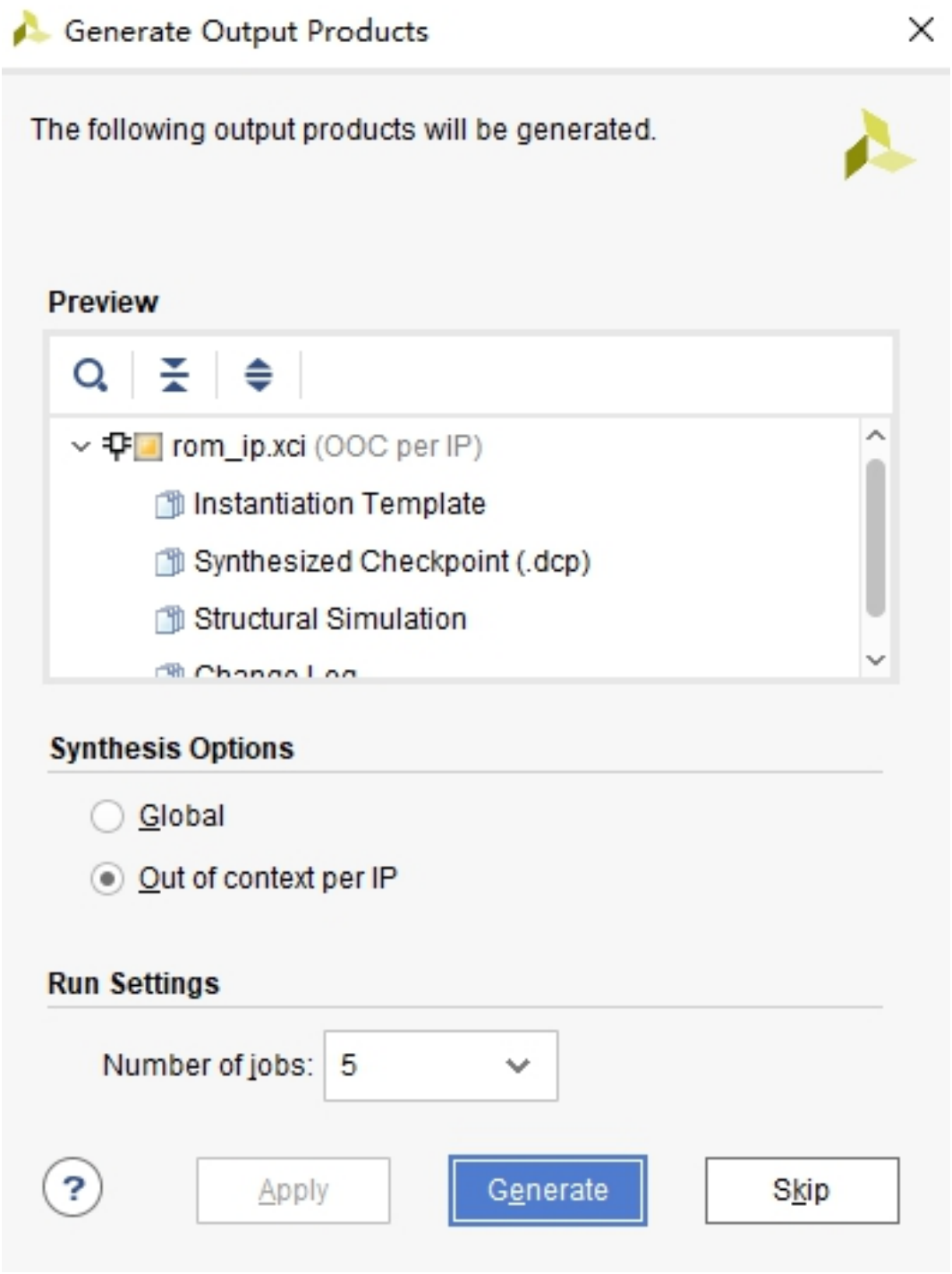
3) Switch to “Port A Options” column, change ROM bit width “Port A Width” to 8, change ROM depth “Port A Depth” to 32, “Enable Port Type” to “Always”, and cancel “Primitives Output Register”



4) Switch to the “Other Options” column, check “Load Init File”, click “Browse”, and select the “.coe” file that you have made before



5) Click ok and click “Generate” to generate the ip core.



Part 7.3: ROM Test Program Writing

The ROM programming is very simple. In the program, as long as we change the ROM address every clock, the ROM will output the internal storage data of the current address and instantiate ila to observe the changes in address and data. The instantiation and

programming of ROM IP are as follows

```
`timescale 1ns / 1ps

module rom_test(
    input      sys_clk_p,          //system clock 200Mhz positive pin
    input      sys_clk_n,          //system clock 200Mhz negative pin
    input rst_n       //Reset, active low
    );

wire [7:0] rom_data;   //ROM Read Data
reg  [4:0] rom_addr;     //ROM Input Address

wire sys_clk ;

IBUFDS IBUFDS_inst (
   .O(sys_clk),  // Buffer output
   .I(sys_clk_p),  // Diff_p buffer input (connect directly to top-level
port)
   .IB(sys_clk_n) // Diff_n buffer input (connect directly to top-level
port)
);

//Generate ROM address Read Data
always @ (posedge sys_clk or negedge rst_n)
begin
    if(!rst_n)
        rom_addr <= 10'd0;
    else
        rom_addr <= rom_addr+1'b1;
end
//Instantiate ROM
rom_ip rom_ip_inst
(
    .clka   (sys_clk    ),      //inoput clka
    .addra  (rom_addr   ),      //input [4:0] addra
    .douta  (rom_data   )       //output [7:0] douta
);
//Instantiate the logic analyzer
ila_0 ila_m0
```

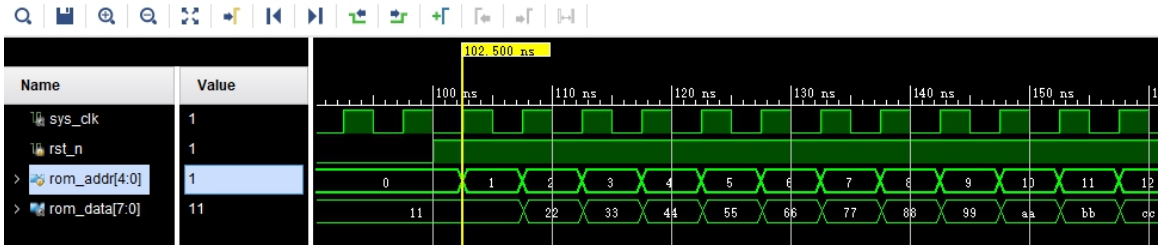
```
(
    .clk    (sys_clk),
    .probe0 (rom_addr),
    .probe1 (rom_data)
);

endmodule
```

Binding pin

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

set_property PACKAGE_PIN AE5 [get_ports sys_clk_p]
set_property IOSTANDARD DIFF_SSTL12 [get_ports sys_clk_p]

create_clock -period 5.000 -name sys_clk_p -waveform {0.000 2.500}
[get_ports sys_clk_p]



set_property PACKAGE_PIN AF12 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
```

Part 7.4: Simulation

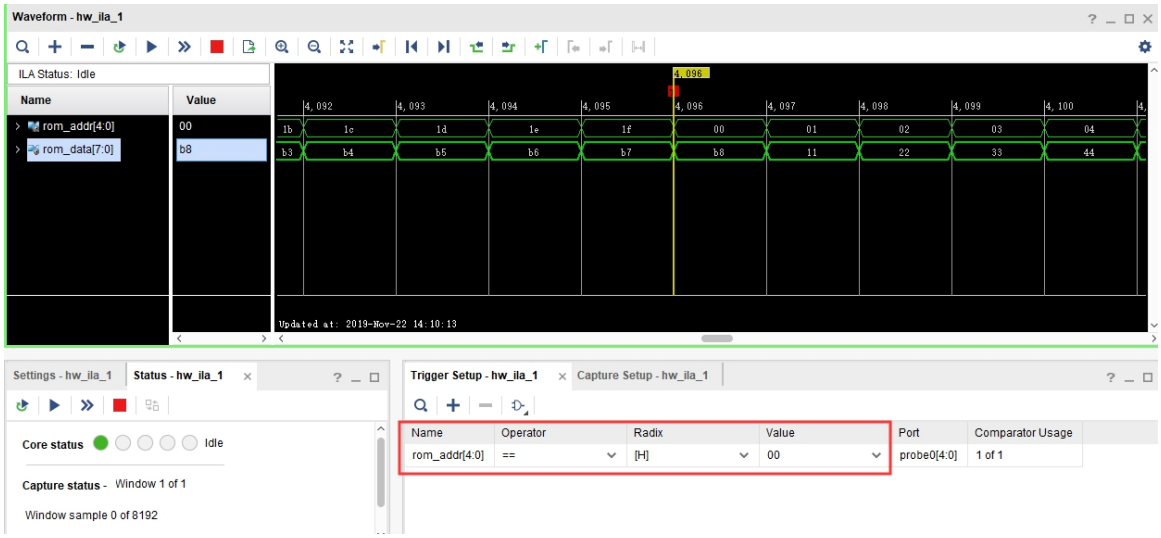
The simulation results are as follows, which is in line with expectations. Like the read data from RAM, the data lags behind the address by one cycle.



Part 7.5: On-board Verification

With address 0 as the trigger condition, you can see that the read

data is consistent with the simulation.



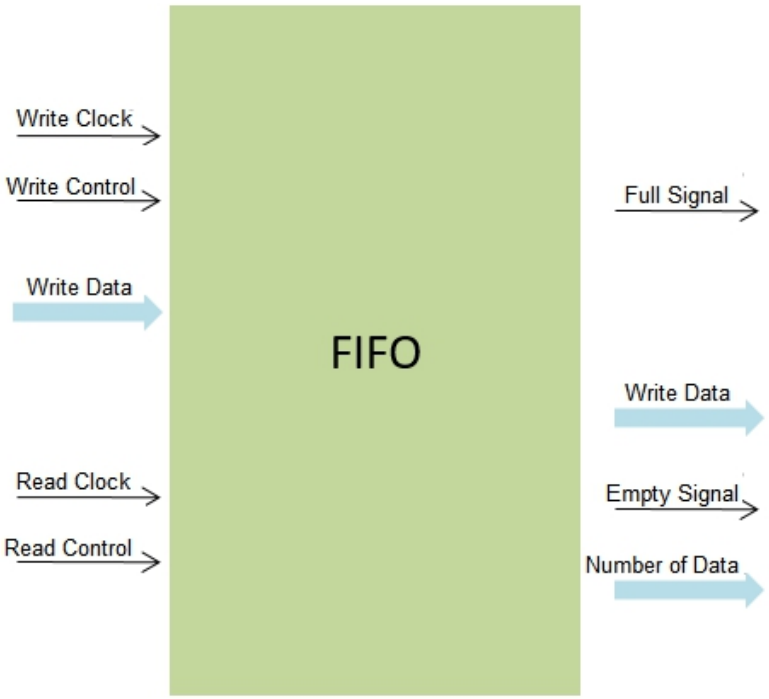
Part 8: FPGA on-chip FIFO read and write test experiment

The experimental Vivado project is "fifo _test."

FIFO is a very important module in FPGA applications. It is widely used for data buffering and data processing across clock domains. Learning FIFO is the key to FPGA, and flexibly using FIFO is an essential skill for FPGA engineers. This chapter mainly introduces the use of FIFO IP provided by XILINX for reading and writing tests.

Part 8.1: Experimental Principle

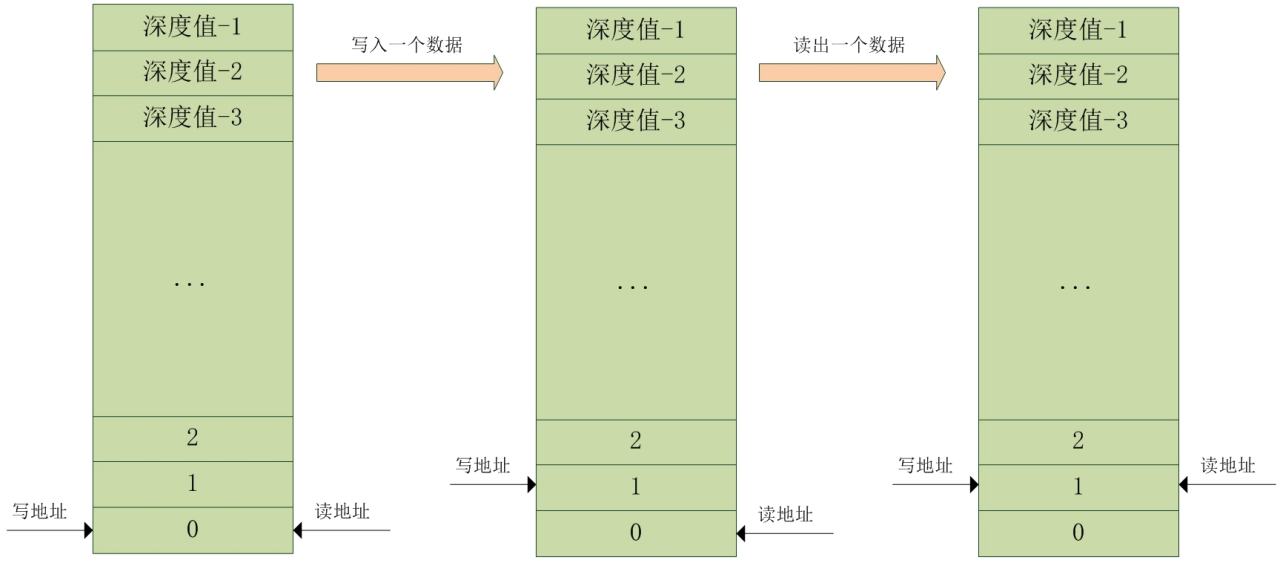
FIFO: First in, First out means that the advanced data comes out first, and the last data comes out later. Xilinx has provided us with an IP core of FIFO in VIVADO. We only need to instantiate a FIFO through the IP core, and write and read the data stored in the FIFO according to the read and write timing of the FIFO.



In fact, FIFO adds many functions on the basis of RAM. The typical structure of FIFO is as follows. It is mainly divided into two parts: read and write. The other is status signal, empty and full signal, and data quantity status signal. The biggest difference from RAM is that FIFO has no address line and cannot read data at random addresses. What is random read data, that is, data at a certain address can be read at will. The FIFO is different. Random reading is not possible. The advantage of this is that you don't need to control the address line frequently.

Although the user cannot see the address line, there are still address operations inside the FIFO to control the RAM read and write interface. The address is shown in the figure below during read and write operations, where the depth value is the maximum number of data that can be stored in a FIFO. In the initial state, the read and write addresses are all 0. After writing a data to the FIFO, the write address is increased by 1, and after reading a data from the FIFO, the read

address is increased by 1. At this time, the state of the FIFO is empty, because one data is written and another data is read.



You can think of the FIFO as a pool. The writing channel means adding water, and the reading channel means discharging water. If the water is added and discharged continuously, if the water filling speed is faster than the water discharging speed, then the FIFO will be full. If it is full and continue to add water, it will overflow. If the water release speed is faster than the water addition speed, then the FIFO will be free. Therefore, it is a difficult task to control the timing and speed of adding and releasing water to ensure that there is always water in the pool. That is, to determine the status of empty and full, choose an opportunity to write data or read data.

According to the read and write clock, it can be divided into synchronous FIFO (read and write clocks are the same) and asynchronous FIFO (read and write clocks are different). Synchronous FIFO control is relatively simple and will not be introduced. The experiment in this section mainly introduces the control of asynchronous FIFO, where the read clock is 75MHz and the write clock is 100MHz. In the experiment, through the online logic

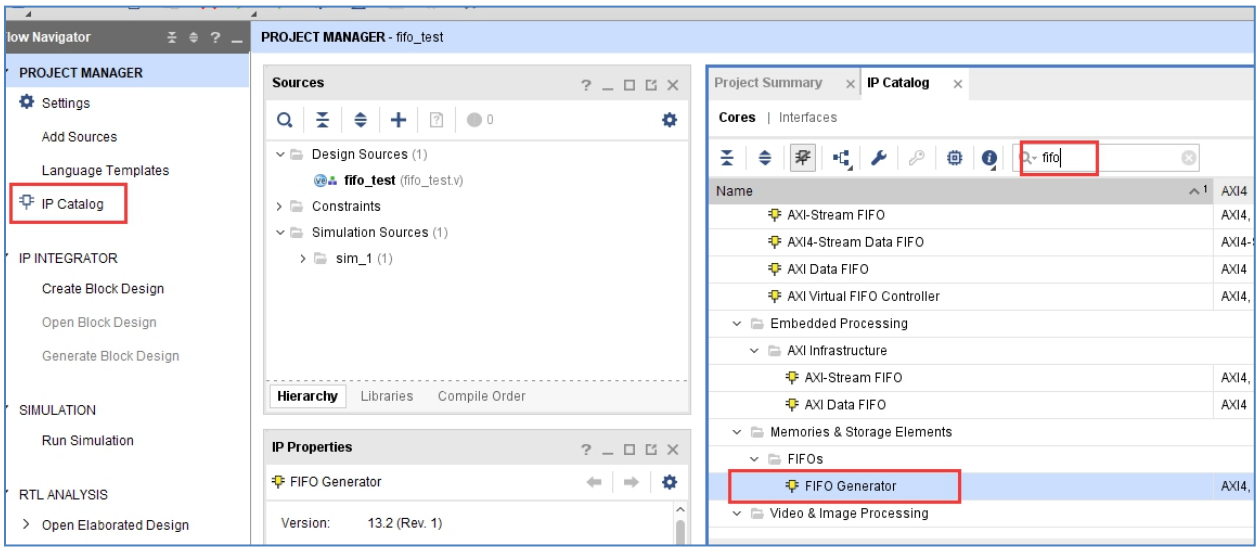
analyzer ila integrated by Vivado, we can observe the read and write timing of the FIFO and the data read from the FIFO.

Part 8.2: Create Vivado Project

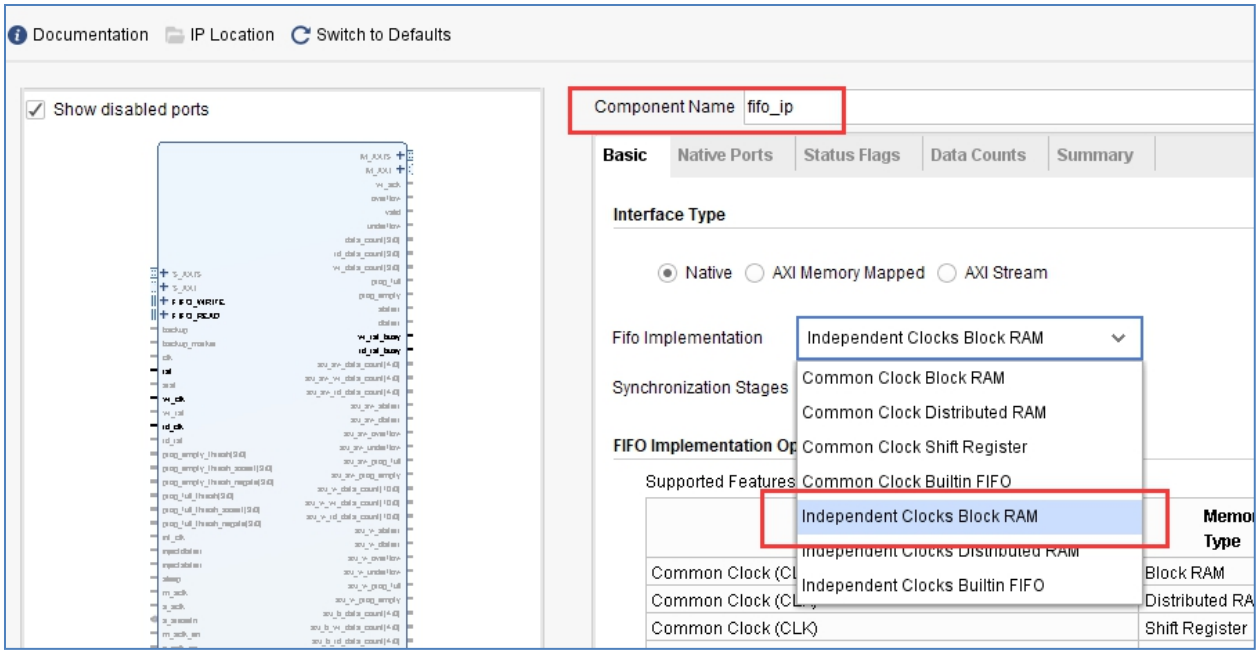
Part 8.2.1: Add FIFO IP Core

Create a new fifo_test project before adding FIFO IP, and then add FIFO IP to the project, the method is as follows

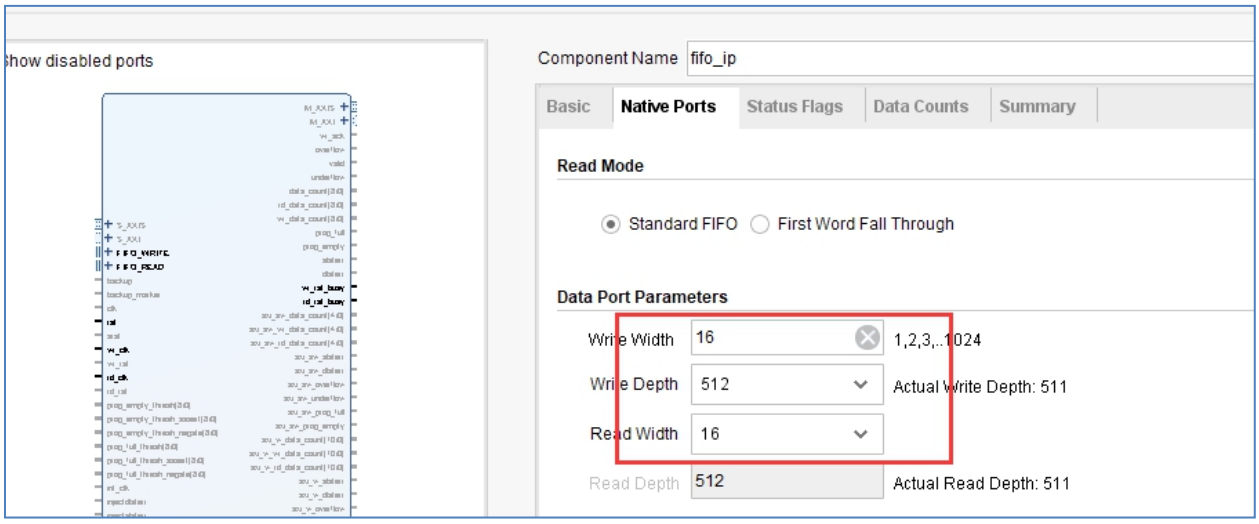
1) Click “IP Catalog” in the figure below, search for “fifo” in the pop-up interface on the right, find “FIFO Generator”, and double-click to open it.



2) In the configuration page that pops up, here you can choose to separate the read and write clocks or use the same one. Generally speaking, we use FIFO to buffer data. Usually, the clock speeds on both sides are different. So independent clock is the most commonly used, we choose "Independent Clocks Block RAM" here, and then click to the next configuration page.

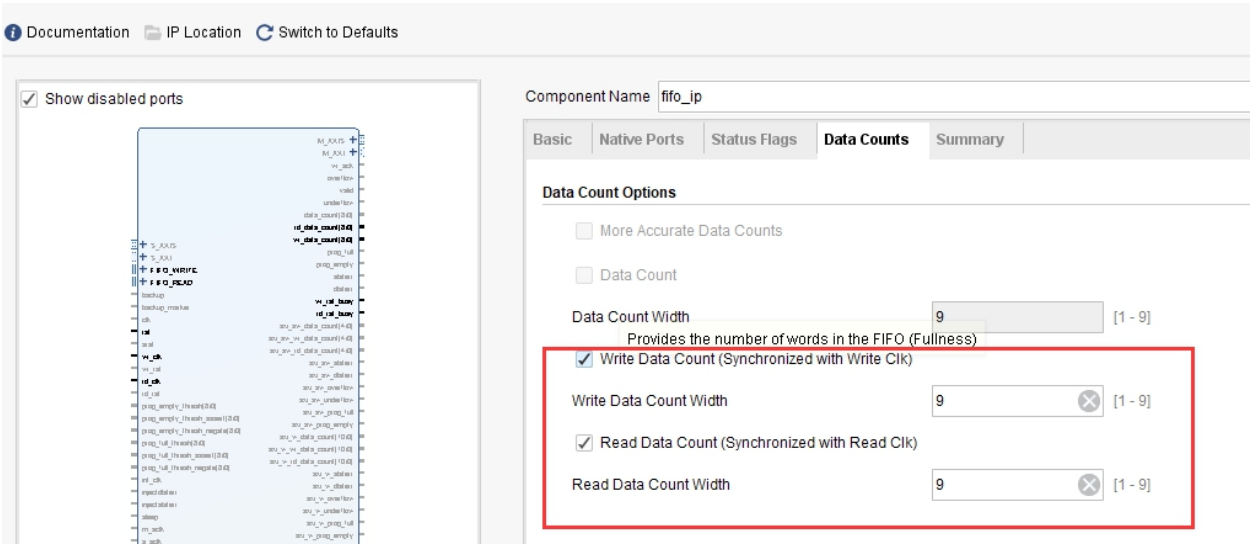


3) Switch to the Native Ports column, select the data bit width of 16, and the FIFO depth of 512. In actual use, you can set it according to your needs. There are two ways to read mode, one is Standard FIFO, that is, the usual FIFO data lags behind the read signal by one cycle, and the other is First Word Fall Through, the data prefetch mode is referred to as FWFT mode. That is, FIFO will fetch a piece of data in advance, when the read signal is valid, the corresponding data is also valid. We first do the standard FIFO experiment.



4) Switch to the “Data Counts” column and enable “Write Data Count”

(how much data has been written into the FIFO) and “Read Data Count” (how much data can be read in the FIFO), so that we can see how much data is inside the FIFO through these two values. Click “OK”, Generate “FIFO IP”.

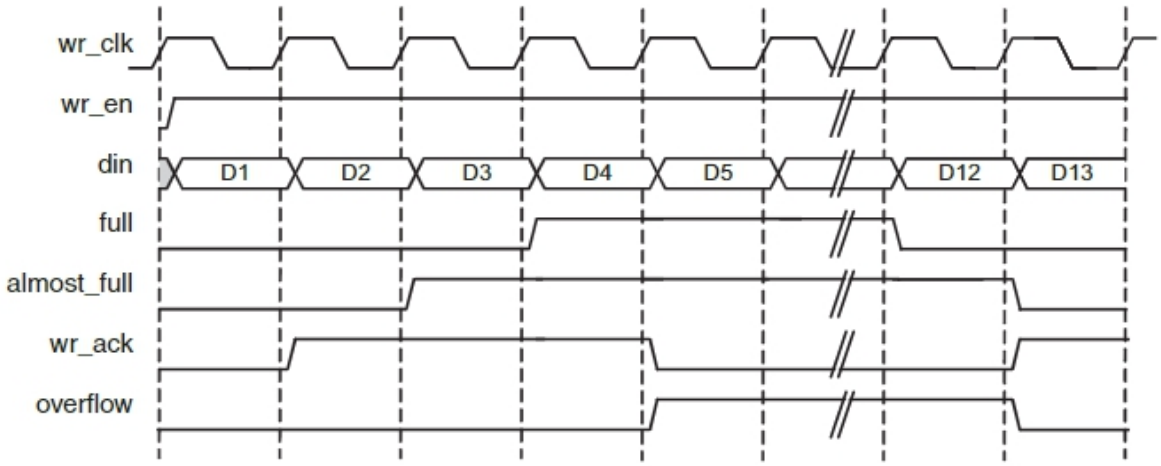


Part 8.2.2: FIFO Port Definition and Timing

Signal Name	Direction	Description
rst	in	reset signal, high valid
wr_clk	in	write clock input
rd_clk	in	read clock input
din	in	write data
wr_en	in	write enable, high effective
rd_en	in	read enable, high effective
dout	out	read data
full	out	full signal
empty	out	null signal
rd_data_count	out	number of readable data
wr_data_count	out	the amount of data written

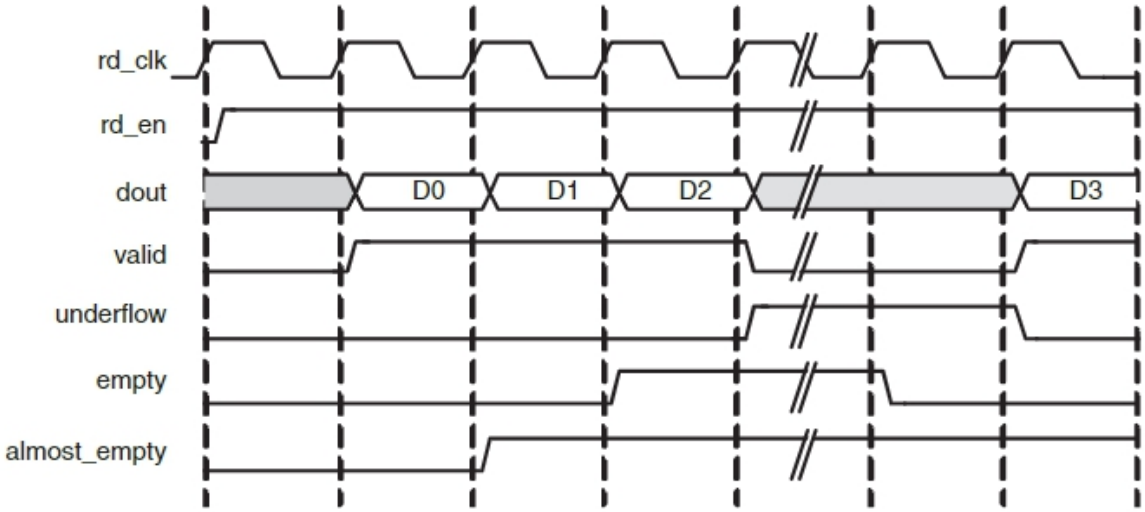
FIFO data writing and reading are operated by the rising edge of the clock. When the “wr_en” signal is high, the FIFO data is written. When the “almost_full” signal is valid, it means that the FIFO can only

write one more data. Once a data is written , the “full” signal will be pulled high, if “wr_en” is still valid in the case of full, that is, continue to write data to the FIFO, the “overflow” of the FIFO will be valid, indicating overflow.



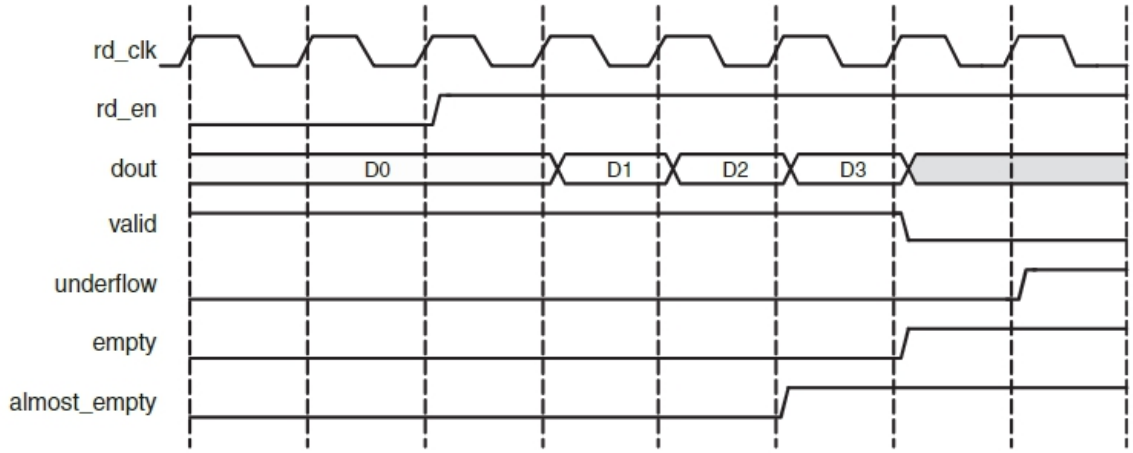
Standard FIFO Write Timing

When the “rd_en” signal is high, the FIFO data is read, and the data is valid in the next cycle. “Valid” is the data valid signal, “almost_empty” means there is another data read. When another data is read, the “empty” signal is valid. If you continue to read, the “underflow” is valid, which means underflow, and the data read is invalid at this time.



Standard FIFO Read Timing

It can be seen from the FWFT mode to read the data sequence diagram that when the rd_en signal is valid, the valid data D0 is already available on the data line and will not be delayed for another cycle. This is the difference from the standard FIFO.



FWFT FIFO Read Timing

For details about FIFO, please refer to the “pg057” document, which can be downloaded from the “xilinx” official website.

Part 8.3: FIFO Test Program Writing

We design according to asynchronous FIFO, and use PLL to generate two clocks, 100MHz and 75MHz respectively, used for write clock and read clock, that is, write clock frequency is higher than read clock frequency.

```
`timescale 1ns / 1ps
module fifo_test
    (
        input      sys_clk_p,           //system clock 200Mhz positive pin
        input      sys_clk_n,           //system clock 200Mhz negative pin
        input     rst_n               //Reset Signal, Negative Valid
    );

reg  [15:0]        w_data         ;          //FIFO Write Data
wire               wr_en          ;          //FIFO Write Enable
wire               rd_en          ;          //FIFO Read Enable
```

```
wire [15:0]       r_data         ;          //FIFO Read Data
wire              full           ;          //FIFO Full Signal
wire              empty          ;          //FIFO empty signal
wire [8:0]        rd_data_count  ;          //Number of readable data
wire [8:0]        wr_data_count  ;          //Number of Data Written
wire              clk_100M       ;          //PLL Generate 100MHz Clock
wire              clk_75M        ;          //PLL Generate 75MHz Clock
wire              locked         ;          //PLL lock Signal, used as
reset signal of the system,Positive mean Locked
wire              fifo_rst_n        ;          //fifo Reset
Signal,Negative Valid

wire              wr_clk         ;          //Write FIFO Clock
wire              rd_clk         ;          //Read FIFO Clock
reg [7:0]         wcnt           ;          //Waiting counter after
   writing FIFO reset
reg [7:0]         rcnt           ;          //Waiting counter after
   reading FIFO  reset

// Instantiate the PLL to generate 100MHz and 75MHz clocks
clk_wiz_0 fifo_pll
 (
  // Clock out ports
  .clk_out1(clk_100M),          // output clk_out1
  .clk_out2(clk_75M),       // output clk_out2
  // Status and control signals
  .reset(~rst_n),               // input reset
  .locked(locked),          // output locked
  // Clock in ports
  .clk_in1_p(sys_clk_p),               // input clk_in1
  .clk_in1_n(sys_clk_n)                // input clk_in1
  );

assign fifo_rst_n    = locked   ;   //Assign PLL LOCK signal to fifo reset
signal
assign wr_clk     = clk_100M    ;   //Assign 100MHz clock to write clock
assign rd_clk     = clk_75M  ;   //Assign 75MHz clock to read clock


/* Write FIFO state machine */
```

```
localparam      W_IDLE      = 1   ;
localparam      W_FIFO        = 2 ;

reg[2:0]  write_state;
reg[2:0]  next_write_state;

always@(posedge wr_clk or negedge fifo_rst_n)
begin
    if(!fifo_rst_n)
        write_state <= W_IDLE;
    else
        write_state <= next_write_state;
end

always@(*)
begin
    case(write_state)
        W_IDLE:
            begin
                if(wcnt == 8'd79)             //Wait for a certain time
after reset, the slowest clock in safety circuit mode is 60 cycles
                next_write_state <= W_FIFO;
                else
                    next_write_state <= W_IDLE;
            end
        W_FIFO:
            next_write_state <= W_FIFO;        //Always writing FIFO
status
        default:
            next_write_state <= W_IDLE;
    endcase
end
//In the IDLE state, that is, after reset, the counter counts
always@(posedge wr_clk or negedge fifo_rst_n)
begin
    if(!fifo_rst_n)
        wcnt <= 8'd0;
    else if (write_state == W_IDLE)
        wcnt <= wcnt + 1'b1 ;
    else
        wcnt <= 8'd0;
```

```
end
//In the state of writing FIFO, if not full, write data to the FIFO

assign wr_en = (write_state == W_FIFO) ? ~full : 1'b0;
//In the case of write enable valid, write data value plus 1

always@(posedge wr_clk or negedge fifo_rst_n)
begin
    if(!fifo_rst_n)
        w_data <= 16'd1;
    else if (wr_en)
        w_data <= w_data + 1'b1;
end

/* Read FIFO state machine  */

localparam      R_IDLE      = 1   ;
localparam      R_FIFO        = 2 ;
reg[2:0]  read_state;
reg[2:0]  next_read_state;

///Generate FIFO read data
always@(posedge rd_clk or negedge fifo_rst_n)
begin
    if(!fifo_rst_n)
        read_state <= R_IDLE;
    else
        read_state <= next_read_state;
end

always@(*)
begin
    case(read_state)
        R_IDLE:
            begin
                if (rcnt == 8'd59)              //Wait for a certain time
after reset, the slowest clock in safety circuit mode is 60 cycles
                    next_read_state <= R_FIFO;
                else
                    next_read_state <= R_IDLE;
            end
```

```
        R_FIFO:
            next_read_state <= R_FIFO ;        //Always read the FIFO
status
        default:
            next_read_state <= R_IDLE;
    endcase
end

//In the IDLE state, that is, after reset, the counter counts
always@(posedge rd_clk or negedge fifo_rst_n)
begin
    if(!fifo_rst_n)
        rcnt <= 8'd0;
    else if (write_state == W_IDLE)
        rcnt <= rcnt + 1'b1 ;
    else
        rcnt <= 8'd0;
end
//In the read FIFO state, if it is not empty, read data from the FIFO
assign rd_en = (read_state == R_FIFO) ? ~empty : 1'b0;

//------------------------------------------------------------
//Instantiate FIFO
fifo_ip fifo_ip_inst
(
  .rst           (~fifo_rst_n    ),   // input rst
  .wr_clk        (wr_clk         ),   // input wr_clk
  .rd_clk        (rd_clk         ),   // input rd_clk
  .din           (w_data         ),   // input [15 : 0] din
  .wr_en         (wr_en          ),   // input wr_en
  .rd_en         (rd_en          ),   // input rd_en
  .dout          (r_data         ),   // output [15 : 0] dout
  .full          (full           ),   // output full
  .empty         (empty          ),   // output empty
  .rd_data_count  (rd_data_count),   // output [8 : 0] rd_data_count
  .wr_data_count  (wr_data_count)    // output [8 : 0] wr_data_count
);

//Write channel logic analyzer 仪
ila_m0 ila_wfifo (
```

```
    .clk(wr_clk),
    .probe0(w_data),
    .probe1(wr_en),
    .probe2(full),
    .probe3(wr_data_count)
);
//Read channel logic analyzer
ila_m0 ila_rfifo (
    .clk(rd_clk),
    .probe0(r_data),
    .probe1(rd_en),
    .probe2(empty),
    .probe3(rd_data_count)
);

Endmodule
```

In the program, the lock signal of the PLL is used as the reset of the fifo, and the 100MHz clock is assigned to the write clock and the 75MHz clock is assigned to the read clock.

```
assign fifo_rst_n   = locked    ;   //将PLL的LOCK信号赋值给fifo的复位信号
assign wr_clk       = clk_100M  ;   //将100MHz时钟赋值给写时钟
assign rd_clk       = clk_75M   ;   //将75MHz时钟赋值给读时钟
```

One thing to note is that the FIFO setting uses the safety circuit by default. This function is to ensure that the input signal to the internal RAM is synchronized. In this case, if you reset asynchronously, you need to wait for 60 slowest clock cycles. In this experiment, it is 60 cycles of 75MHz, then the 100MHz clock needs about (100/75) x60=80 cycles.

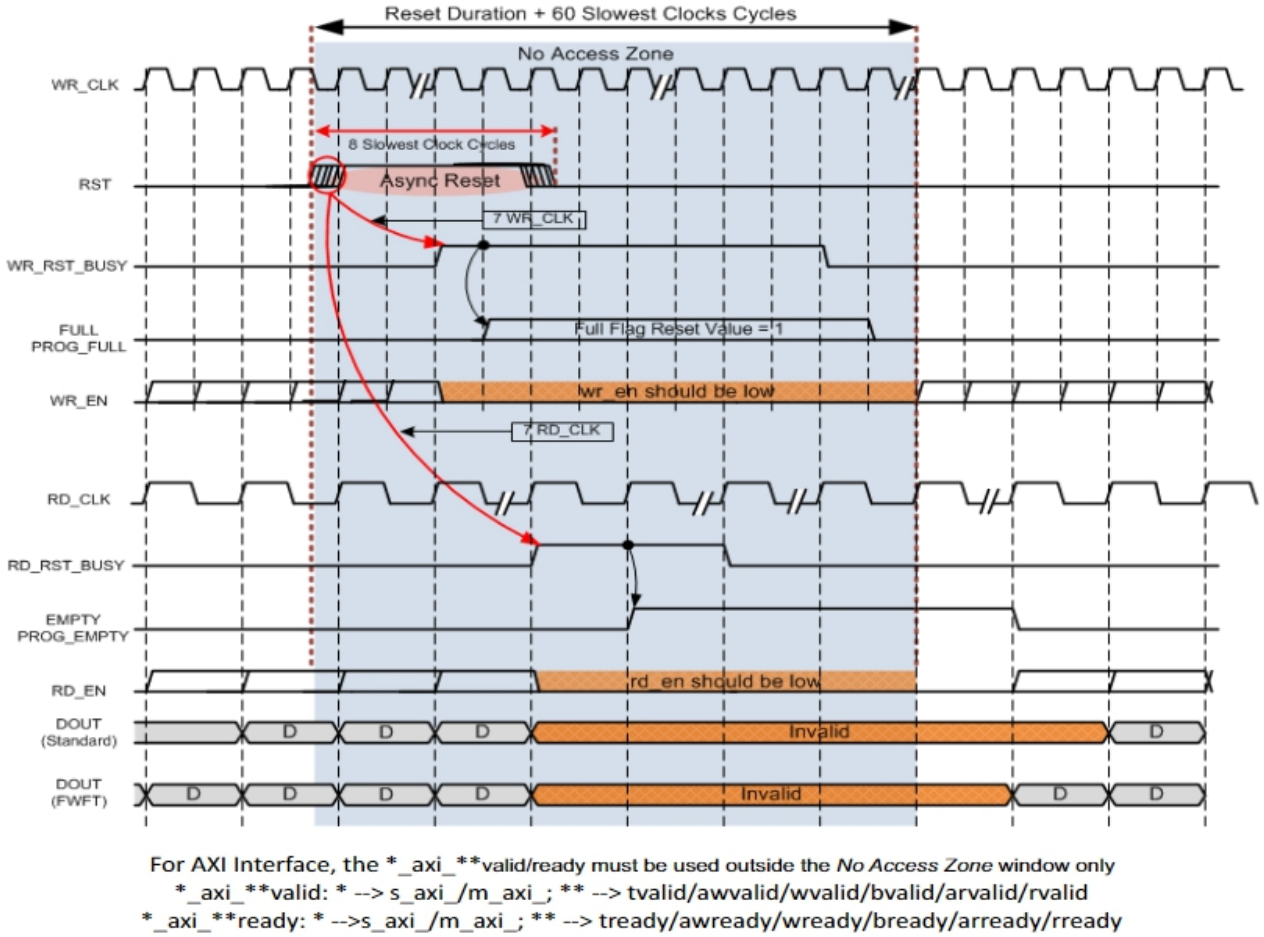


Figure 3-29: **FIFO Asynchronous Reset Timing With Safety Circuit**

Therefore, in the write state machine, wait for 80 cycles to enter the write FIFO state

```
always@(*)
begin
    case(write_state)
        W_IDLE:
            begin
                if(wcnt == 8'd79)                    //复位后等待一定时间，safety circuit模式下的最慢时钟60个周期
                    next_write_state <= W_FIFO;
                else
                    next_write_state <= W_IDLE;
            end
        W_FIFO:
            next_write_state <= W_FIFO;              //一直在写FIFO状态
        default:
            next_write_state <= W_IDLE;
    endcase
end
```

In the read state machine, wait for 60 cycles to enter the read state

```
always@(*)
begin
    case(read_state)
        R_IDLE:
            begin
                if (rcnt == 8'd59)              //复位后等待一定时间，safety circuit模式下的最慢时钟60个周期
                    next_read_state <= R_FIFO;
                else
                    next_read_state <= R_IDLE;
            end
        R_FIFO:
            next_read_state <= R_FIFO ;         //一直在读FIFO状态
        default:
            next_read_state <= R_IDLE;
    endcase
end
```

If the FIFO is not full, keep writing data to the FIFO

```
//在写FIFO状态下，如果不满就向FIFO中写数据
assign wr_en = (write_state == W_FIFO) ? ~full : 1'b0;
```

If the FIFO is not empty, always read data from the FIFO

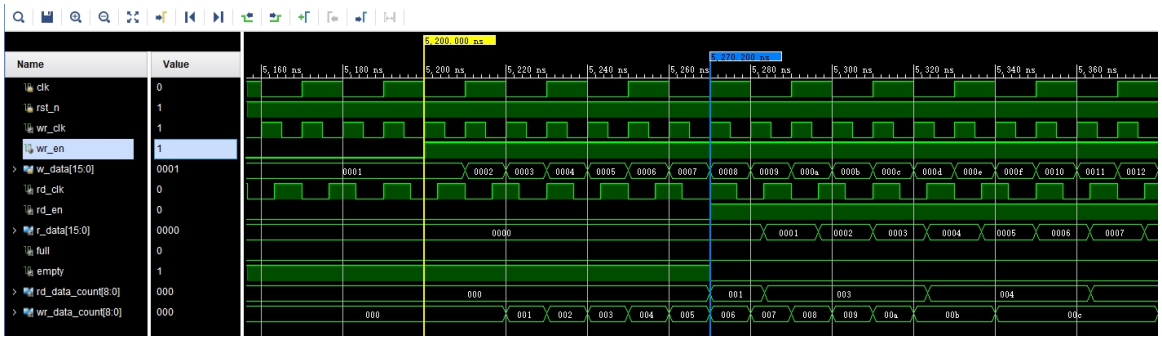
```
//在读FIFO状态下，如果不空就从FIFO中读数据
assign rd_en = (read_state == R_FIFO) ? ~empty : 1'b0;
```

Instantiate two logic analyzers and connect the signals of the write channel and the read channel respectively

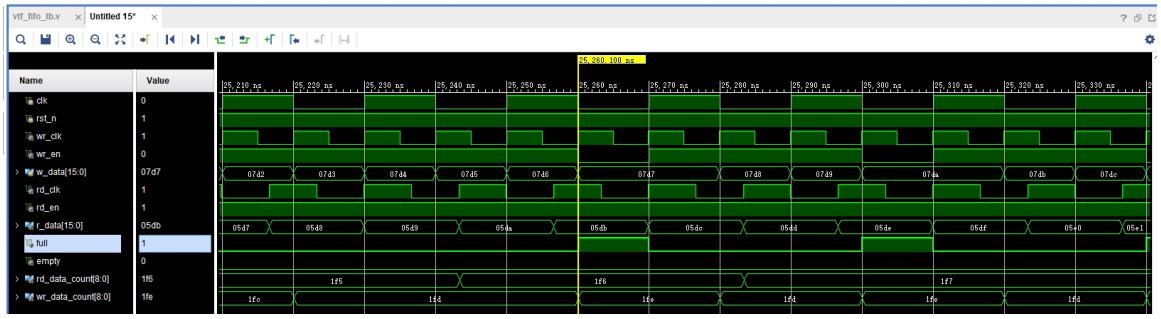
```
//写通道逻辑分析仪
ila_m0 ila_wfifo (
    .clk        (wr_clk         ),
    .probe0     (w_data         ),
    .probe1     (wr_en          ),
    .probe2     (full           ),
    .probe3     (wr_data_count  )
);
//读通道逻辑分析仪
ila_m0 ila_rfifo (
    .clk        (rd_clk         ),
    .probe0     (r_data         ),
    .probe1     (rd_en          ),
    .probe2     (empty          ),
    .probe3     (rd_data_count  )
);
```

Part 8.4: Simulation

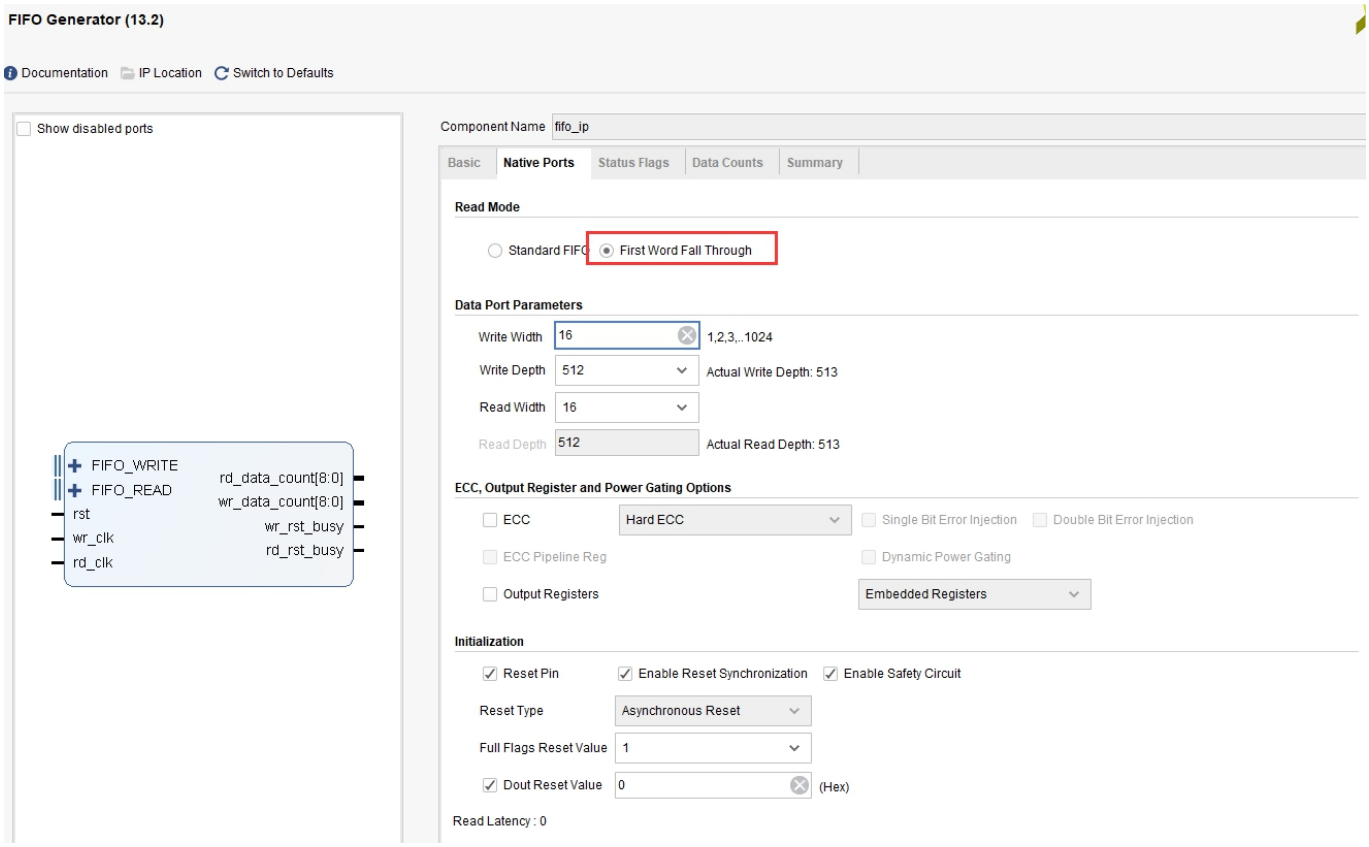
The following is the simulation result. It can be seen that data is written after the write enable “wr_en” is valid. The initial value is 0001. It takes a certain period of time to write from the beginning to when “empty” is not empty, because internal synchronization is still required. After it is not empty, start to read data, and the read data lags behind “rd_en” by one cycle.



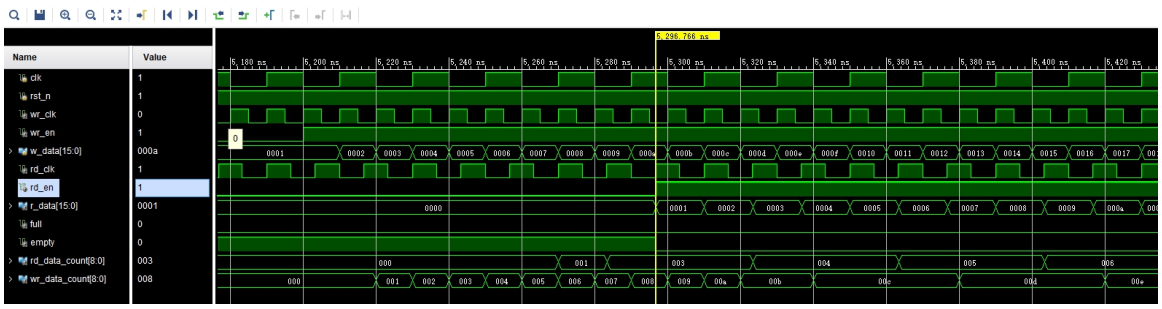
You can see later that if the FIFO is full, according to the design of the program, no data will be written to the FIFO when it is full, and wr_en will be pulled low. Why is it full? It is because the write clock is faster than the read clock. If the write clock and the read clock are swapped, that is, if the read clock is fast, there will be a read empty situation. You can try it



If you change the “Read Mode” of FIFO to “First Word Fall Through”

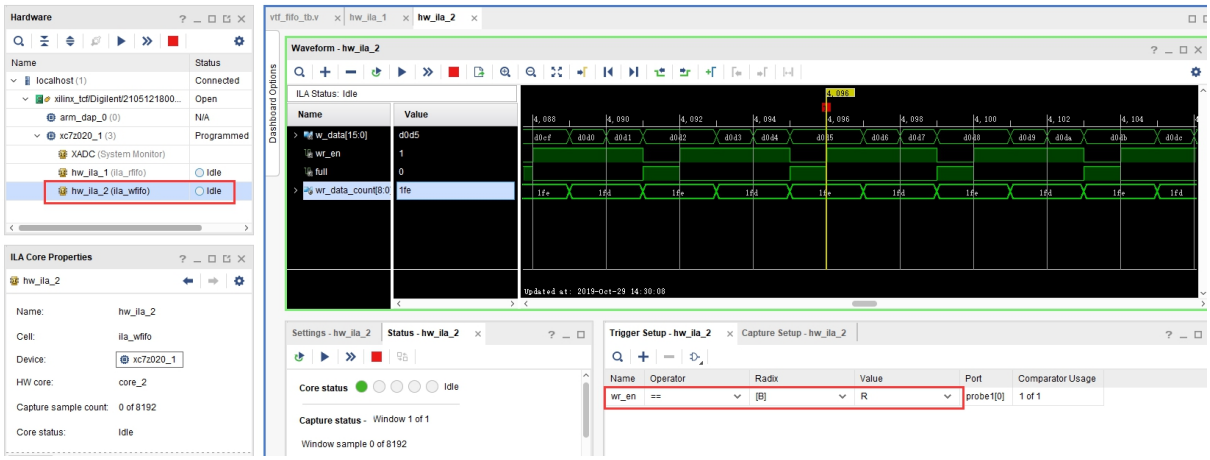


The simulation results are as follows, you can see that the data is also valid when “rd_en” is valid, there is no difference of one cycle.

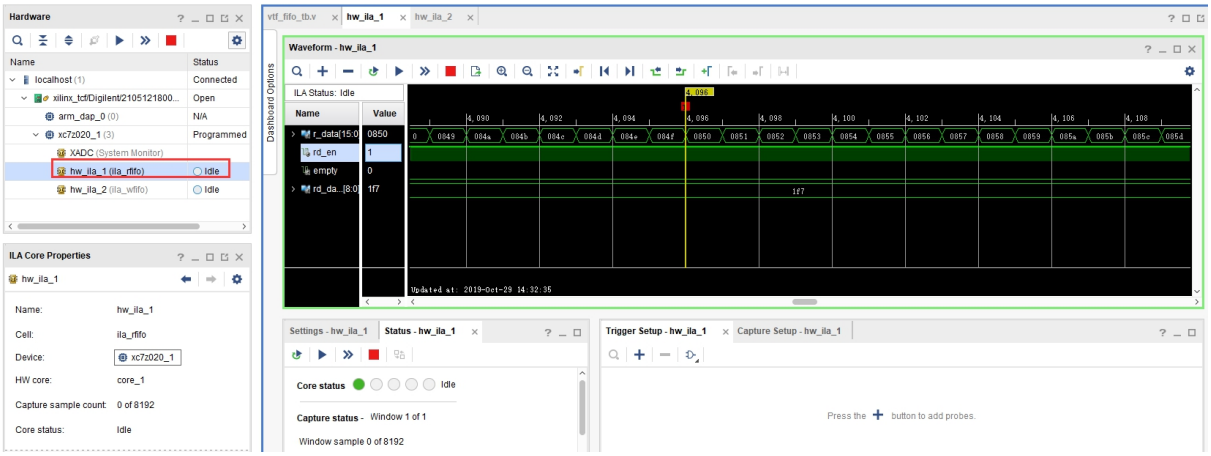


Part 8.5: On-board Verification

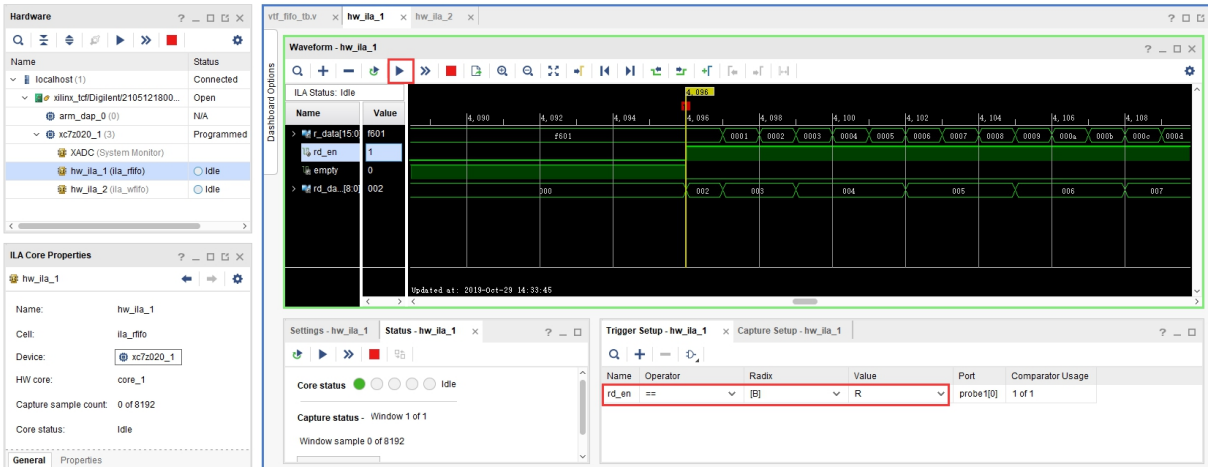
Generate the bit file, download the bit file, there will be two ila, first look at the write channel, you can see that when the “full” signal is high, “wr_en” is low, and no more data is written to it.



The read channel is also consistent with the simulation.



If you use the rising edge of rd_en as the trigger condition, click run, and then press reset, which is our bound PLKEY1, the following result will appear, which is consistent with the simulation. In the standard FIFO mode, the data lags “rd _en by” one cycle.

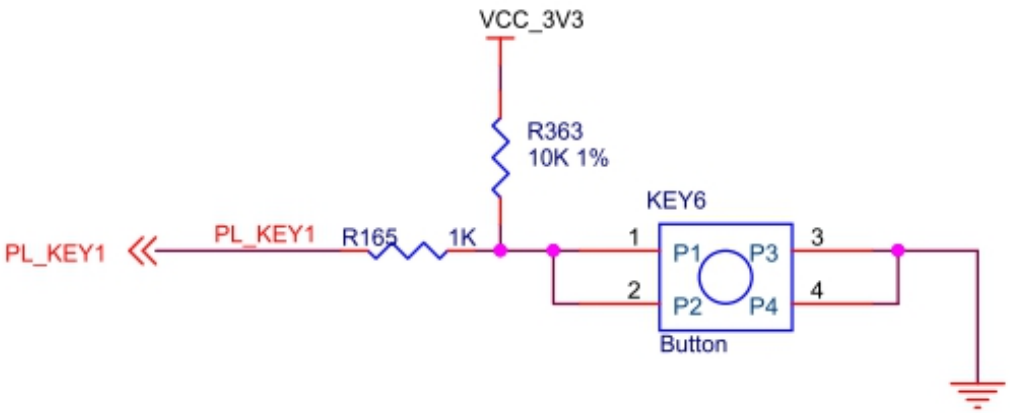


Part 9:Key Experiment in Vivado

The experimental Vivado project is "key_test".

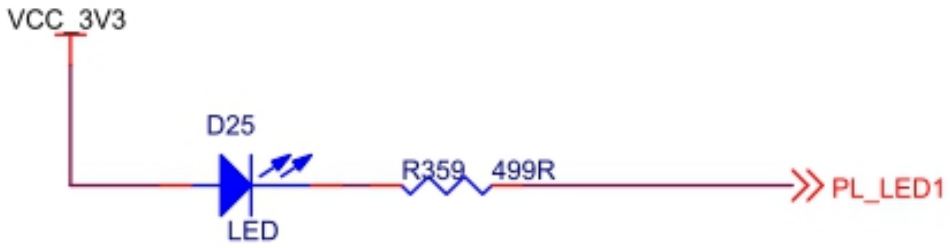
Keys are the most commonly used and simplest peripherals in FPGA design. In this chapter, we will use key detection experiments to detect whether the key functions of the FPPGA development board are normal, understand the specific relationship between the hardware description language and FPGA, and learn the use of Vivado RTL ANALYSIS.

Part 9.1: Key Hardware Circuit



Key Circuit Schematic

It can be seen from the circuit that the key is high (Positive) when it is released and low (Negative) when it is pressed.

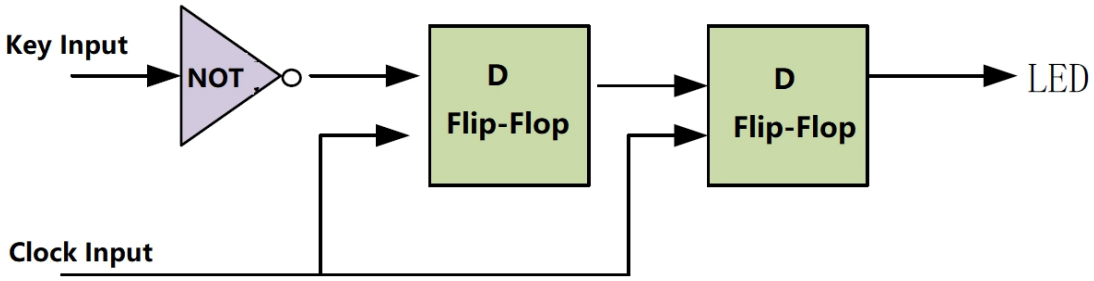


LED Circuit Schematic

And for the LED part, high level is on, low level is off

Part 9.2: Programming

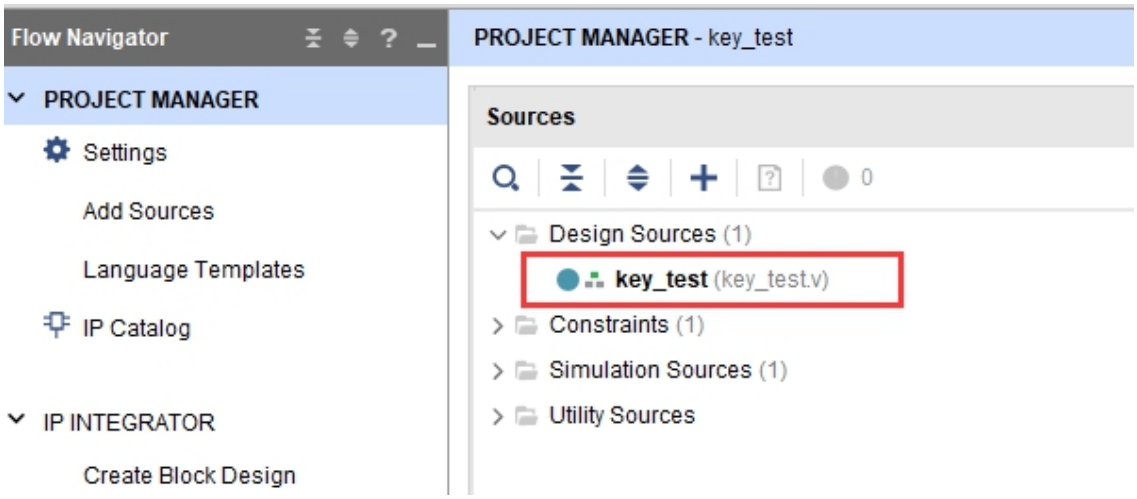
This program is not designed to be very complicated. Through a simple hardware description language, you can see through the connection between the hardware description language and the FPGA hardware. First, we pass the key input through a NOT gate and then through 2 sets of D flip-flops. The signal passing through the D flip-flop will be latched on the rising edge of the D flip-flop clock input and then sent to the output.



Before the hardware description language coding, we have completed the hardware construction, which is a normal development process. With the hardware design idea, the design can be completed either through drawing or through Verilog HDL or VHDL. The tool is selected according to the complex program designed and the program familiar with a certain language.

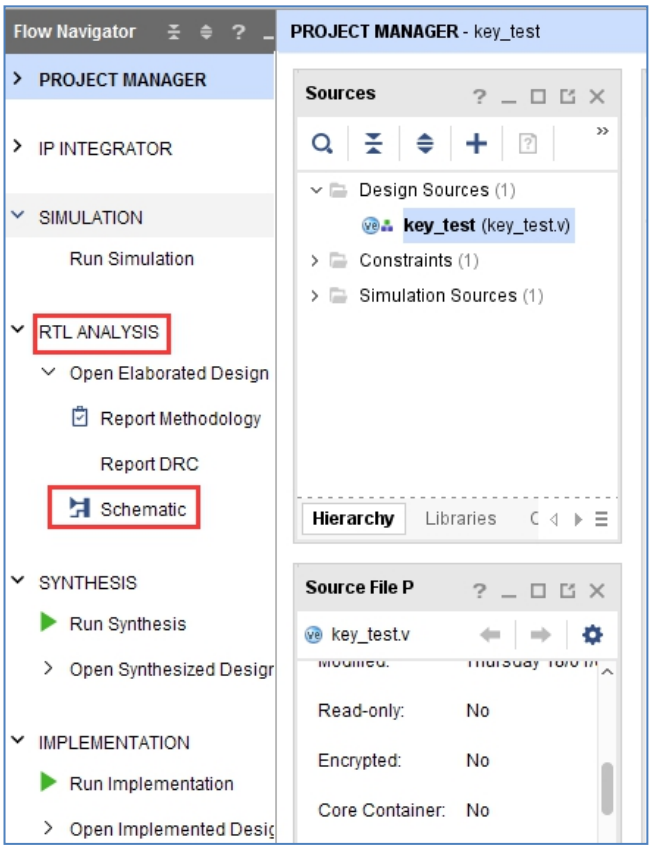
Part 9.3: Create Vivado project

1) First, create a test project for the button, add verilog test code, and complete the process of compiling and assigning pins.

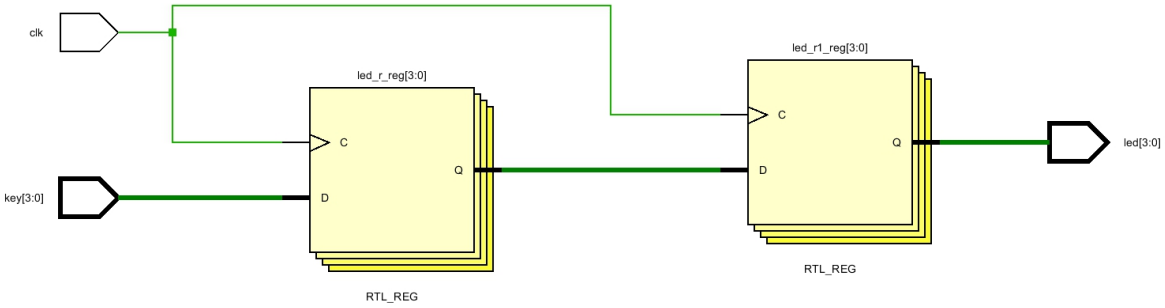



```
`timescale 1ns / 1ps
module key_test
(
    input               clk,      //system clock 25Mhz on board
    input [3:0]          key,      //input four key signal,when the keydown,the
value is 0
    output[3:0]          led       //LED display ,when the siganl low,LED lighten
);

reg[3:0] led_r;          //define the first stage register , generate four D
Flip-flop
reg[3:0] led_r1;         //define the second stage register ,generate four D
Flip-flop
always@(posedge clk)
begin
    led_r <=  key;       //first stage latched data
end

always@(posedge clk)
begin
    led_r1 <= led_r;     //second stage latched data
end

assign led = led_r1;

endmodule
```

2) We can use the RTL ANALYSIS tool to view the design



3) Analyzing the RTL diagram, it can be seen that the first-level D flip-flop is input after NOT Gate, and the second-level is directly input, which is consistent with the expected design.



Part 9.4: On-board Verification

After the Bit file is downloaded to the FPGA development board, the "PL LED" on the FPGA development board is on, and the button "PL KEY1" is pressed and the "PL LED1" is off.

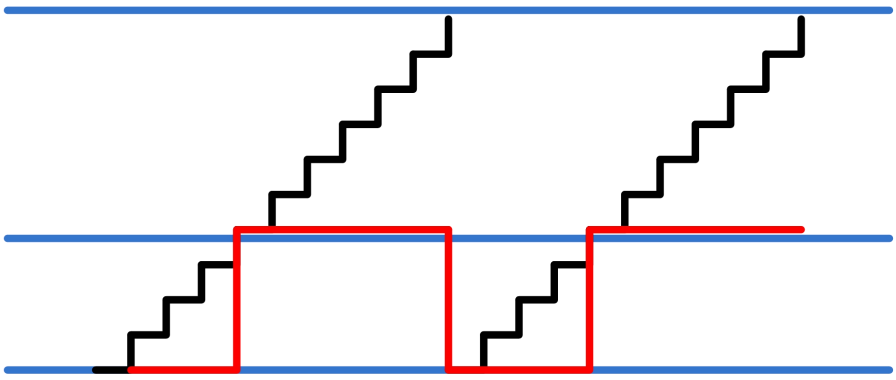
Part 10: PWM Breathing Light Experiment

The experimental Vivado project is "pwm_led".

This article mainly explains the use of PWM to control LED to achieve the effect of breathing light.

Part 10.1: Experimental Principle

As shown in the figure below, with an N-bit counter, the maximum value can be expressed as 2 to the Nth power, and the minimum value is 0. The counter accumulates with "period" as the step value, and when it reaches the maximum value, it will overflow and enter the next accumulation period. When the counter value is greater than "duty", the pulse output is high, otherwise the output is low. In this way, the pulse output with adjustable duty cycle shown in the red line in the figure can be completed. At the same time, the "period" can adjust the pulse frequency, it can be understood as the step value of the counter.



Schematic diagram of PWM pulse width modulation

After the square wave output with different pulse duty ratio is added to the LED, the LED light will display different brightness. By continuously adjusting the duty ratio of the square wave, the brightness of the LED light can be adjusted.

Part 10.2: Experimental Design

The design of the PWM module is very simple. It has been mentioned in the above principle, so the principle will not be discussed here.

Signal Name	Direction	Description
clk	in	system clock
rst	in	asynchronous reset, high (Positive) reset
period	in	PWM pulse width period (frequency) control. period = PWM output frequency * (2 to the Nth power) / system clock frequency. obviously, the larger the N, the higher the frequency accuracy.
duty	in	duty cycle control, duty cycle = duty / (2 to the Nth power) * 100%

PWM module (ax_pwm) port

```
`timescale 1ns / 1ps
module ax_pwm
#(
    parameter N = 16 //pwm bit width
)
(
    input        clk,
    input        rst,
    input[N - 1:0]period,//pwm step value
    input[N - 1:0]duty,      //duty value
    output       pwm_out  //pwm output
    );

reg[N - 1:0] period_r;      //period register
reg[N - 1:0] duty_r;     //duty register
reg[N - 1:0] period_cnt; //period counter
reg pwm_r;
```

```
assign pwm_out = pwm_r;
always@(posedge clk or posedge rst)
begin
    if(rst==1)
    begin
        period_r <= { N {1'b0} };
        duty_r <= { N {1'b0} };
    end
    else
    begin
        period_r <= period;
        duty_r   <= duty;
    end
end
//period counter, step is period value
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) //if period counter is bigger or equals to
duty value, then set pwm value to high
            pwm_r <= 1'b1;
        else
            pwm_r <= 1'b0;
    end
End
```

So how to achieve the effect of breathing light? We know that the effect of the breathing light is the process of changing from dark to bright, and then from bright to dark. The bright and dark effect is adjusted by the duty cycle, so we mainly control the duty cycle, that is, control the duty value.

In the following test code, by setting the value of period, set the PWM frequency to 200Hz, and the PWM_PLUS state is to increase the duty value. If it increases to the maximum value, set pwm_flag to 1, and start to reduce the duty value. When it is reduced to the minimum value, the duty value will be increased and the cycle will continue. The PWM_GAP state is the adjustment interval, and the time is 100us.

```
`timescale 1ns / 1ps
module pwm_test(
               input               sys_clk_p,          //system clock
200Mhz postive pin
                input                  sys_clk_n,          //system clock
200Mhz negetive pin
               input      rst_n,     //low active
               output     led        //high-on, low-off
              );
localparam CLK_FREQ = 200 ;             //200MHz
localparam US_COUNT = CLK_FREQ ;        //1 us counter
localparam MS_COUNT = CLK_FREQ*1000 ;  //1 ms counter

localparam DUTY_STEP   = 32'd100000 ;  //duty step
localparam DUTY_MIN_VALUE = 32'h6fffffff ;    //duty minimum value
localparam DUTY_MAX_VALUE = 32'hffffffff ;    //duty maximum value

localparam IDLE          = 0;   //IDLE state
localparam PWM_PLUS   = 1;    //PWM duty plus state
localparam PWM_MINUS     = 2;    //PWM duty minus state
localparam PWM_GAP    = 3;    //PWM duty adjustment gap

wire       pwm_out;   //pwm output
reg[31:0]  period;        //pwm step value
reg[31:0]  duty;      //duty value
```

```
reg        pwm_flag ; //duty value plus and minus flag, 0: plus; 1: minus

reg[3:0]   state;
reg[31:0]  timer;     //duty adjustment counter

assign led = pwm_out ; //led high active

wire clk ;

IBUFDS IBUFDS_inst (
   .O(clk),  // Buffer output
   .I(sys_clk_p),  // Diff_p buffer input (connect directly to top-level
port)
   .IB(sys_clk_n) // Diff_n buffer input (connect directly to top-level
port)
);

always@(posedge clk or negedge rst_n)
begin
    if(rst_n == 1'b0)
    begin
        period     <= 32'd0;
        timer      <= 32'd0;
        duty       <= 32'd0;
        pwm_flag   <= 1'b0 ;
        state      <= IDLE;
    end
    else
        case(state)
            IDLE:
            begin
                period     <= 32'd17179;   //The pwm step value, pwm
200Hz(period = 200*2^32/50000000)
                state      <= PWM_PLUS;
                duty       <= DUTY_MIN_VALUE;
            end
            PWM_PLUS :
            begin
                if (duty > DUTY_MAX_VALUE - DUTY_STEP) //if duty is bigger
than DUTY MAX VALUE minus DUTY_STEP , begin to minus duty value
                begin
```

```
                pwm_flag   <= 1'b1 ;
                duty       <= duty - DUTY_STEP ;
            end
            else
            begin
                pwm_flag   <= 1'b0 ;
                duty       <= duty + DUTY_STEP ;
            end

            state      <= PWM_GAP ;
        end
        PWM_MINUS :
        begin
            if (duty < DUTY_MIN_VALUE + DUTY_STEP) //if duty is little
than DUTY MIN VALUE plus duty step, begin to add duty value
            begin
                pwm_flag   <= 1'b0 ;
                duty       <= duty + DUTY_STEP ;
            end
            else
            begin
                pwm_flag   <= 1'b1 ;
                duty       <= duty - DUTY_STEP ;
            end
            state      <= PWM_GAP ;
        end
        PWM_GAP:
        begin
            if(timer >= US_COUNT*100)      //adjustment gap is 100us
            begin
                if (pwm_flag)
                    state <= PWM_MINUS ;
                else
                    state <= PWM_PLUS ;

                timer <= 32'd0;
            end
            else
            begin
                timer <= timer + 32'd1;
            end
```

```
            end
            default:
            begin
                state <= IDLE;
            end
        endcase
end

//Instantiate pwm module
ax_pwm
#(
  .N(32)
 )
ax_pwm_m0(
    .clk      (clk),
    .rst      (~rst_n),
    .period   (period),
    .duty     (duty),
    .pwm_out  (pwm_out)
    );
endmodule
```

Part 10.3: Download Verification

Generate bitstream and download the bit file, you can see that the PL LED1 light produces a breathing light effect. PWM is a commonly used module, such as fan speed control, motor speed control and so on.

Part 11: RS232 Experiment

The experimental Vivado project is "rs232_test".

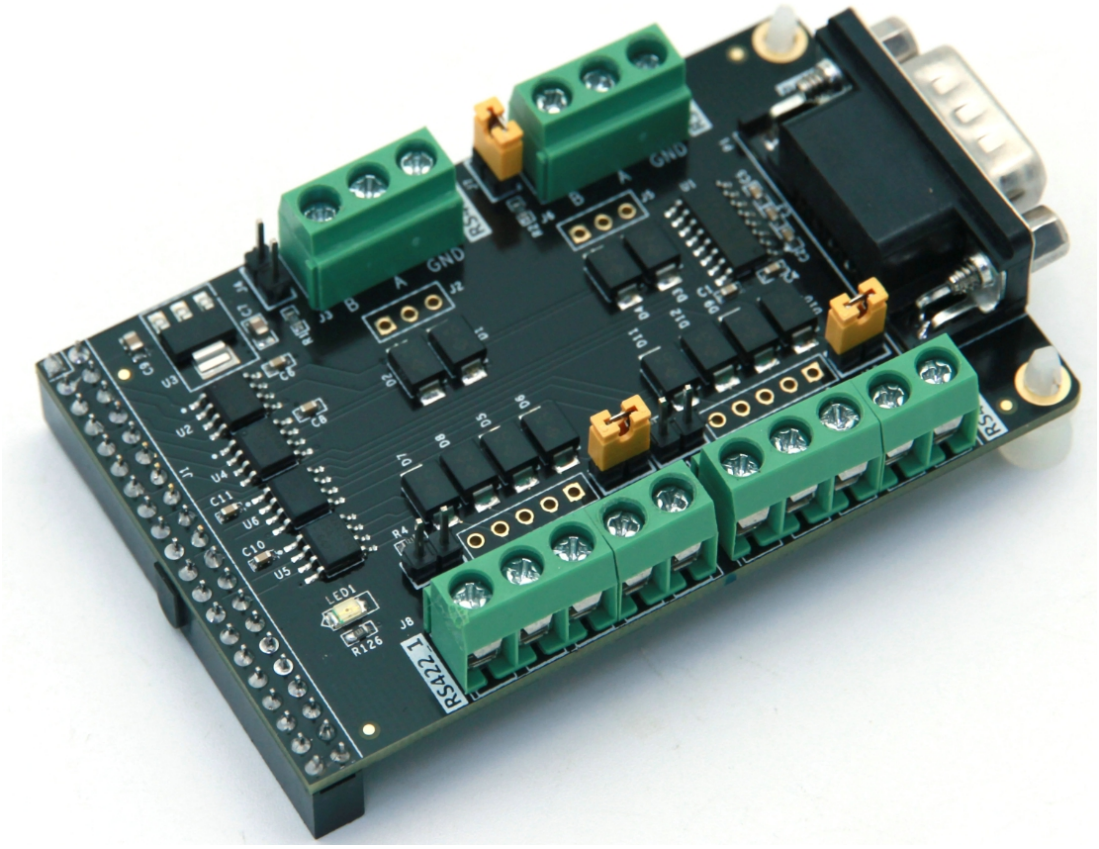
This chapter uses the RS232 circuit of the AN3485 module to realize UART data transmission.

Part 11.1: AN3485 Module Introduction

The ALINX AN3845 module, RS232/485/422 communication module designed for industrial field applications. It includes one RS232 interface, two RS485 and two RS422 communication interfaces. Cooperate with the FPGA development board to realize remote data transmission and communication of RS232, 485 and 422.

The RS232, 485, and 422 interfaces use the MAX3232, MAX3485, and MAX3490 chips as level-shifting chips. The AN3485 module has a 40-pin female header for connecting to the FPGA development board. The RS232 interface is a standard DB9 serial port, which can be directly connected to a computer or other device through a serial cable. The RS485 and RS422 interfaces are connected to the external terminals by terminals. The ultra-long-distance transmission can reach up to several kilometers. In addition, the RS485 and RS422 interface parts are equipped with ESD protection function of plus or minus 15KV

The AN3485 module product photo as below:



Part 11.1.1: AN3485 Communication Module Detail Parameter

The following are the detailed parameters of the AN3485 communication module:

- **RS232 Interface**
  - One-channel standard DB9 serial interface
  - Max3232 to convert between RS232 and TTL (Transistor-Transistor Logic) level signals
  - Transfer speed up to 120kbps
- **RS485 Interface**
  - 2-channel RS484, connected with 3 wire terminals
  - Max3485 to convert between RS485 and TTL (Transistor-Transistor Logic) level signals
  - Industrial design, Strong anti-interference ability, and effective

lightning protection

- With a 120 ohm matching resistor, a jumper cap can be inserted to enable matching resistors, and short-circuiting is recommended for long-distance transmission.
- Support multi machine communication, Max. allowed 128 devices in bus
- The transmission rate is up to 500Kbps data communication rate.

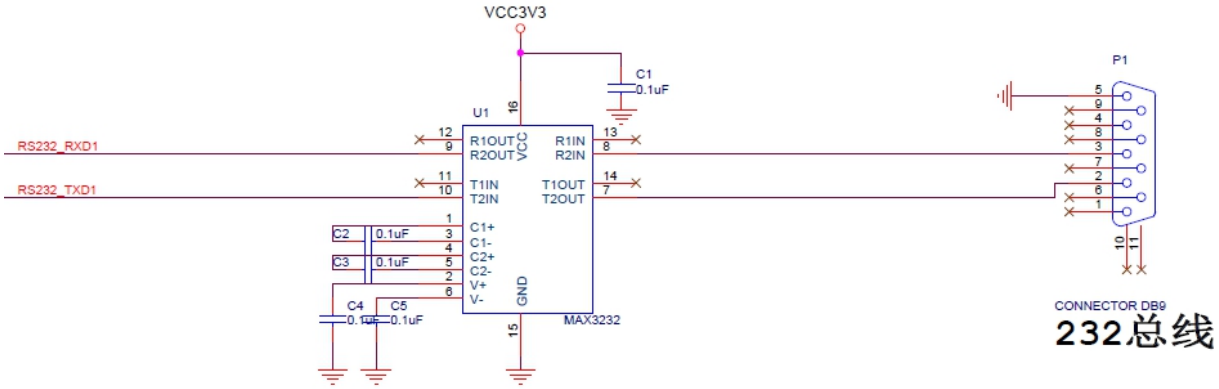
➢ **RS422 Interface**

- 2-channel RS422, connected with 5 wire terminals
- Max3490 to convert between RS422 and TTL (Transistor-Transistor Logic) level signals
- Industrial design, Strong anti-interference ability, and effective lightning protection
- With a 120 ohm matching resistor, a jumper cap can be inserted to enable matching resistors, and short-circuiting is recommended for long-distance transmission.
- Support multi machine communication, Max. allowed 128 devices in bus
- The transmission rate is up to 500Kbps data communication rate.

Part 11.1.2: AN3485 Module Function Description

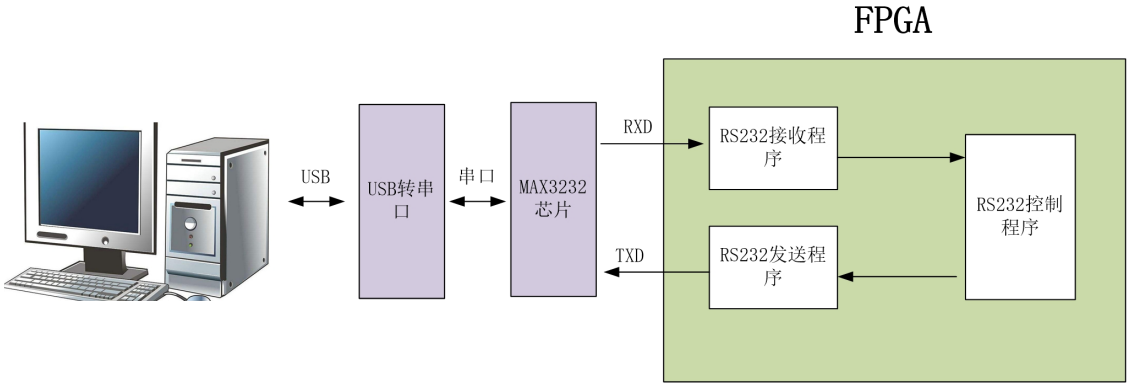
The RS232 interface of the AN3485 module uses the MAX3232 chip to convert RS232 and +3.3V TTL levels. The TTL level serial port receive and transmit signals (RXD, TXD) are connected to the 40-pin connector for serial communication with the external FPGA chip or ARM chip. The maximum speed of RS232 serial communication is 120kbps. The schematic diagram of the RS232 interface is shown as

below:



Part 11.2: Programming

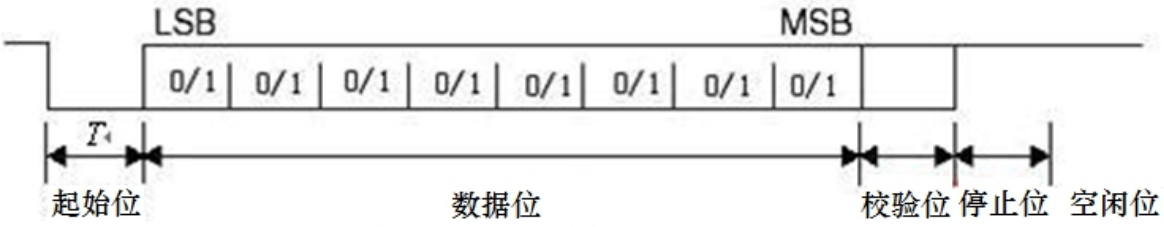
The serial port mentioned in this article refers to asynchronous serial communication, and asynchronous serial refers to UART (Universal Asynchronous Receiver/Transmitter), universal asynchronous receiving/transmitting. The experimental program is designed to send "HELLO ALINX" to the serial port every second. If the data received by RXD is received, then the received data will be sent out to realize the loopback function.



Part 11.2.1: Asynchronous serial communication protocol

The message frame starts with a low start bit, followed by 7 or 8 data bits, an available parity bit and one or several high stop bits.

When the receiver finds the start bit, it knows that the data is ready to be sent, and tries to synchronize with the transmitter clock frequency. If parity is selected, the UART adds parity bits after the data bits. The parity bit can be used to help error checking. In the receiving process, the UART removes the start bit and the end bit from the message frame, performs parity check on the incoming bytes, and converts the data bytes from serial to parallel. The UART transmission sequence is shown in the figure below:



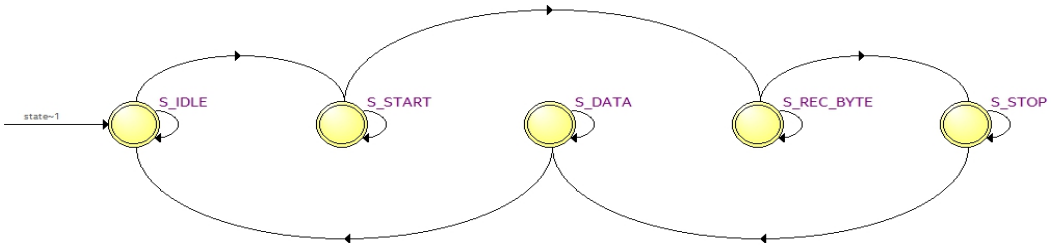
It can be seen from the waveform that the start bit is low, the stop bit and the idle bit are both high, which means it is high when there is no data transmission. Using this feature, we can accurately receive data when a falling edge occurs. When the incident occurs, we believe that there will be a data transfer.

Part 11.2.2: Baud rate

Common serial port communication baud rates are 2400, 9600, 115200, etc. The transmitting and receiving baud rates must be consistent to communicate correctly. Baud rate refers to the maximum number of data bits transmitted in 1 second, including start bit, data bit, parity bit, and stop bit. If the communication baud rate is set to 9600, then the time length of a data bit is 1/9600 second, and the baud rate in this experiment is generated by a 50MHz clock.

Part 11.2.3: Receiving module design

The serial port receiving module uart_rx is a parameterized configurable module, the parameter "CLK_FRE" defines the system clock frequency of the receiving module, the unit is Mhz, and the parameter "BAUD_RATE" is the baud rate. The state transition diagram of the receiving state machine is as follows:



"S_IDLE" state is idle state, enter "S_IDLE" after power-on, if the signal "rx_pin" has a falling edge, it is the start bit of the serial port, enter the state "S_START", wait a BIT time, after the start bit ends, enter Data bit receiving status "S_REC_BYTE". In this experiment, the data bit design is 8 bits. After the reception is completed, it enters the "S_STOP" state. There is no waiting for a BIT cycle in "S_STOP", but only half of the BIT time. This is because if you wait for a cycle, you may miss the start bit judgment of the next data, and finally enter the "S_DATA" state, and send the received data to other modules. In this module, we mention one point: In order to meet the sampling theorem, each data is sampled at the midpoint of the baud rate counter when receiving data to avoid data errors:

```
//receive serial data bit data
always@(posedge clk or negedge rst_n)
begin
    if(rst_n == 1'b0)
        rx_bits <= 8'd0;
```

```
    else if(state == S_REC_BYTE && cycle_cnt == CYCLE/2 - 1)
        rx_bits[bit_cnt] <= rx_pin;
    else
        rx_bits <= rx_bits;
end
```

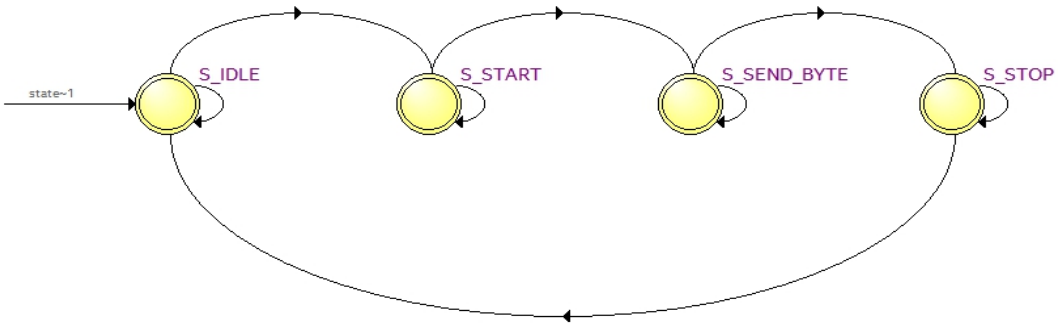
Note: No parity bit is designed in this experiment.

Signal Name	Direction	Width(bit)	Description
clk	in	1	System Clcok
rst_n	in	1	Asynchronous reset, low level reset
rx_data	out	8	Received serial port data (8-bit data)
rx_data_valid	out	1	The received serial port data is valid (high effective)
rx_data_ready	in	1	Indicates that the user can receive data from the receiving module, and the data will be sent when rx_data_ready and rx_data_valid are both high
rx_pin	in	1	Serial port receive data input

Table 11-1: Serial port receiving module uart_rx port

Part 11.2.4: Transmitting module design

The design of the transmitting module uart_tx is similar to that of the receiving module. It also uses a state machine. The state transition diagram is as follows:



After power-on, it enters the "S_IDLE" idle state. If there is a transmitting request, it enters the sending start bit state "S_START". After the start bit is sent, it enters the sending data bit state

"S_SEND_BYTE". After the data bit is sent, it enters the sending stop bit state. "S_STOP", it enters the idle state after the stop bit is sent. In the data transmitting module, the data written from the top module is directly transferred to the register 'tx_reg', and the data transmission is carried out under the condition of the state machine through the 'tx_reg' register to simulate the serial port transmission protocol:

```
always@(posedge clk or negedge rst_n)
begin
    if(rst_n == 1'b0)
        tx_reg <= 1'b1;
    else
        case(state)
            S_IDLE,S_STOP:
                tx_reg <= 1'b1;
            S_START:
                tx_reg <= 1'b0;
            S_SEND_BYTE:
                tx_reg <= tx_data_latch[bit_cnt];
            default:
                tx_reg <= 1'b1;
        endcase
end
```

Signal Name	Direction	Width(bit)	Description
clk	in	1	System Clock
rst_n	in	1	Asynchronous reset, low level (Negative) reset
tx_data	in	8	Serial data to be transmitted (8-bit data)
tx_data_valid	in	1	The serial port data sent is valid (high valid Positive)
tx_data_ready	out	1	The transmitting module is ready to transmit data, and the user can pull the tx_data_valid signal high to transmit data to the transmitting module. Data is transmitted when tx_data_ready and tx_data_valid are

			both high
tx_pin	out	1	Serial data transmission

Table 11-2: Serial port Transmitting module uart_tx port

Part 11.2.5: Baud rate generation

In the transmitting and receiving module, the parameter CYCLE is declared, which is a cycle count value of the UART. Of course, the count is performed under a 50MHz clock. The user only needs to set the two parameters CLK_FRE and BAUD_RATE.

```
module uart_rx
#(
    parameter CLK_FRE = 50,       //clock frequency(Mhz)
    parameter BAUD_RATE = 115200 //serial baud rate
)
(
    input                         clk,              //clock input
    input                         rst_n,            //asynchronous reset input, low a
    output reg[7:0]               rx_data,          //received serial data
    output reg                    rx_data_valid,    //received serial data is valid
    input                         rx_data_ready,    //data receiver module ready
    input                         rx_pin            //serial data input
);
//calculates the clock cycle for baud rate
localparam                        CYCLE = CLK_FRE * 1000000 / BAUD_RATE;
//state machine code
```

Part 11.2.6: Test Program

The test program is designed to transmit “HELLO ALINX\r\n” to the serial port once every second from the FPGA. During the period of non-sending, if the serial port data is received, the received data will be directly sent to the transmitting module and then returned. "\R\n", which is consistent with C language, is both carriage return and line feed.

The test program instantiated the transmitting module and the receiving module respectively, and passed the parameters in at the same time, and the baud rate was set to 115200.

```
always@(posedge sys_clk or negedge rst_n)
begin
```

```
    if(rst_n == 1'b0)
    begin
        wait_cnt <= 32'd0;
        tx_data <= 8'd0;
        state <= IDLE;
        tx_cnt <= 8'd0;
        tx_data_valid <= 1'b0;
    end
    else
    case(state)
        IDLE:
            state <= SEND;
        SEND:
        begin
            wait_cnt <= 32'd0;
            tx_data <= tx_str;

            if(tx_data_valid == 1'b1 && tx_data_ready == 1'b1 && tx_cnt < 8'd12)//Send 12
bytes data
            begin
                tx_cnt <= tx_cnt + 8'd1; //Send data counter
            end
            else if(tx_data_valid && tx_data_ready)//last byte sent is complete
            begin
                tx_cnt <= 8'd0;
                tx_data_valid <= 1'b0;
                state <= WAIT;
            end
            else if(~tx_data_valid)
            begin
                tx_data_valid <= 1'b1;
            end
        end
        WAIT:
        begin
            wait_cnt <= wait_cnt + 32'd1;
```

```
                if(rx_data_valid == 1'b1)
                begin
                    tx_data_valid <= 1'b1;
                    tx_data <= rx_data;      // send uart received data
                end
                else if(tx_data_valid && tx_data_ready)
                begin
                    tx_data_valid <= 1'b0;
                end
                else if(wait_cnt >= CLK_FRE * 1000000) // wait for 1 second
                    state <= SEND;
            end
            default:
                state <= IDLE;
        endcase
end

//combinational logic
//Send "HELLO ALINX\r\n"
always@(*)
begin
    case(tx_cnt)
        8'd0 :   tx_str <= "H";
        8'd1 :   tx_str <= "E";
        8'd2 :   tx_str <= "L";
        8'd3 :   tx_str <= "L";
        8'd4 :   tx_str <= "O";
        8'd5 :   tx_str <= " ";
        8'd6 :   tx_str <= "A";
        8'd7 :   tx_str <= "L";
        8'd8 :   tx_str <= "I";
        8'd9 :   tx_str <= "N";
        8'd10:   tx_str <= "X";
        8'd11:   tx_str <= "\r";
        8'd12:   tx_str <= "\n";
```

```
            default:tx_str <= 8'd0;
        endcase
end
uart_rx#
(
    .CLK_FRE(CLK_FRE),
    .BAUD_RATE(115200)
) uart_rx_inst
(
    .clk                    (sys_clk            ),
    .rst_n                  (rst_n              ),
    .rx_data                (rx_data            ),
    .rx_data_valid          (rx_data_valid      ),
    .rx_data_ready          (rx_data_ready      ),
    .rx_pin                 (uart_rx            )
);

uart_tx#
(
    .CLK_FRE(CLK_FRE),
    .BAUD_RATE(115200)
) uart_tx_inst
(
    .clk                    (sys_clk            ),
    .rst_n                  (rst_n              ),
    .tx_data                (tx_data            ),
    .tx_data_valid          (tx_data_valid      ),
    .tx_data_ready          (tx_data_ready      ),
    .tx_pin                 (uart_tx            )
);
```

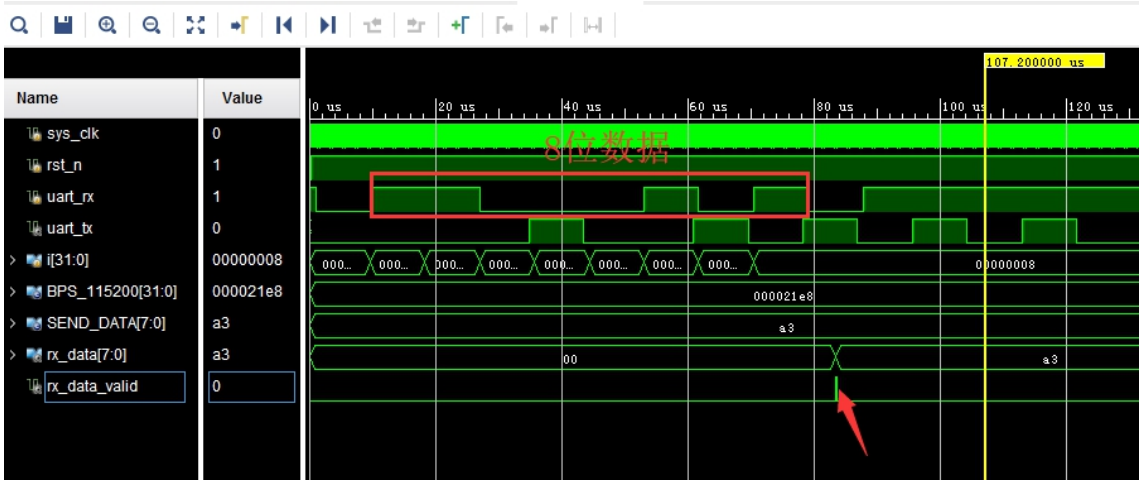
Part 11.3: Simulation

Here we have added a stimulus program **vtf_uart_test.v** file for serial port reception to simulate uart serial port reception. Here, the

data of 0xa3 is transmit to the uart_rx of the serial port module, Each bit of data is transmit at 115200 baud rate, with 1 start bit, 8 data bits and 1 stop bit.

```
    always #10 sys_clk = ~ sys_clk;    //20ns一个周期，产生50MHz时钟源

    parameter                          BPS_115200 = 8680;//每个比特的时间
    parameter                          SEND_DATA = 8'b1010_0011;//要发送的数据

    integer i = 0;

      initial begin
        uart_rx = 1'b1;     //bus idle
        #1000 uart_rx = 1'b0;      //transmit start bit

        for (i=0;i<8;i=i+1)
        #BPS_115200 uart_rx = SEND_DATA[i];      //transmit data bit

        #BPS_115200 uart_rx = 1'b0;      //transmit stop bit
        #BPS_115200 uart_rx = 1'b1;      //bus idle

       end

endmodule
```

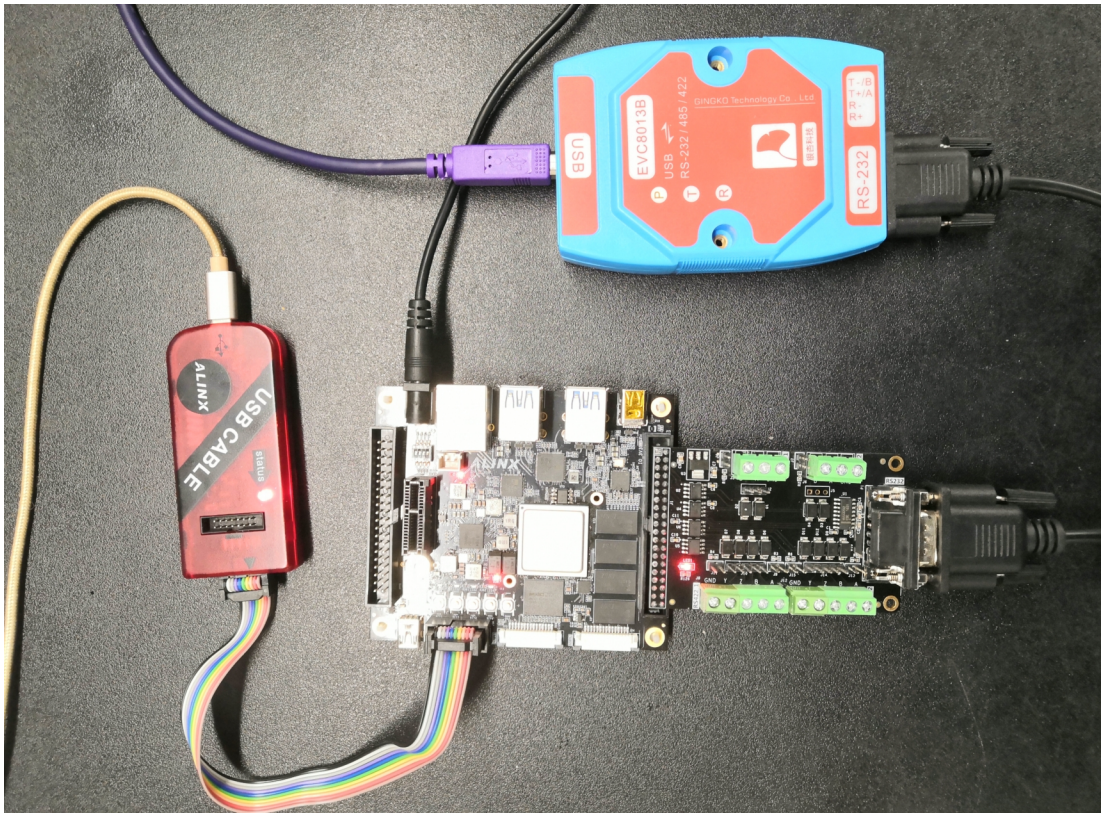
The simulation results are as follows. When the program receives 8-bit data, rx_data_valid is valid, and the data bit a3 of rx_data[7:0].



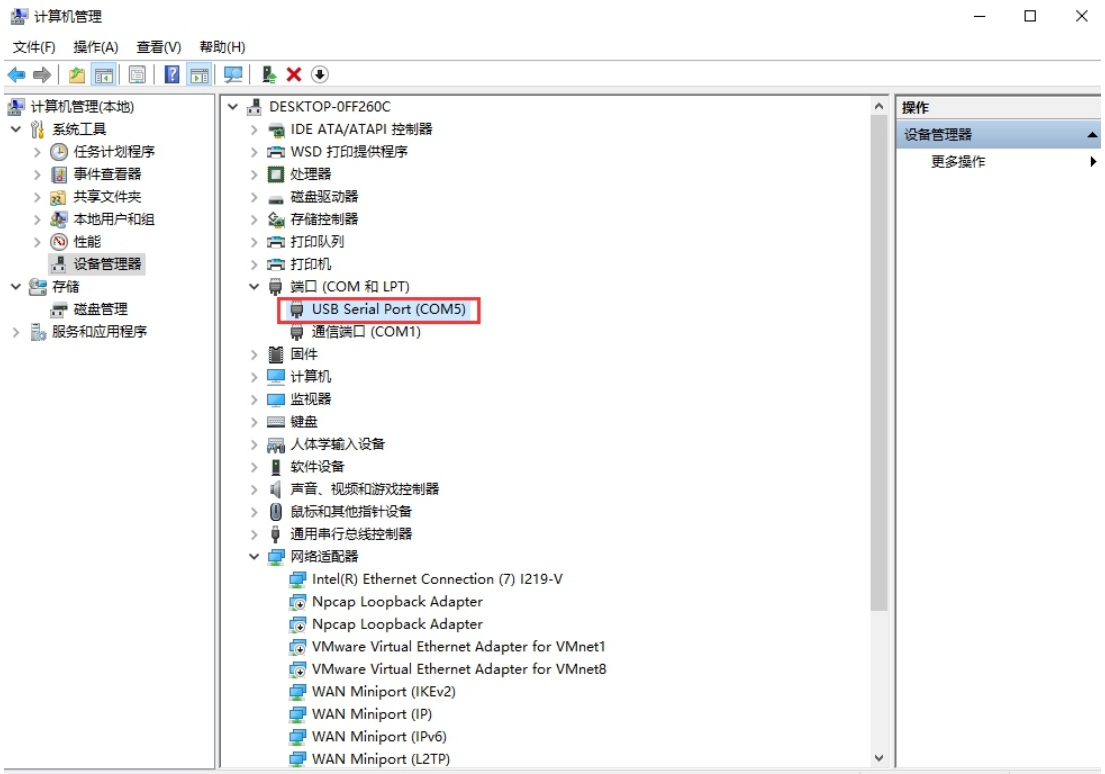
Part 11.4: On-board Verification

Insert the AN3485 module into the J11 expansion port, where the USB to RS232/RS485/RS422 device is used. Since many computers do not have a 9-pin serial interface, we connect to the USB-to-serial

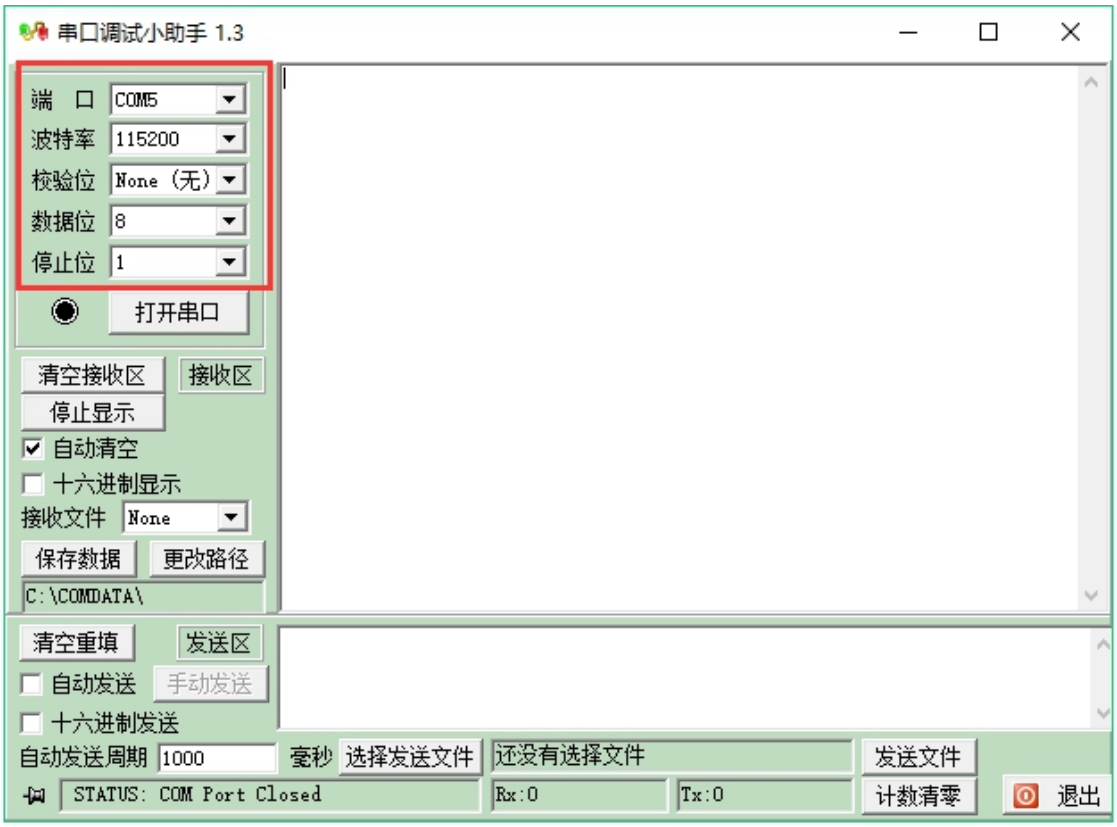
device through a serial cable, and then connect to the computer via USB. If the computer has a serial port, it can be directly connected to the serial port.



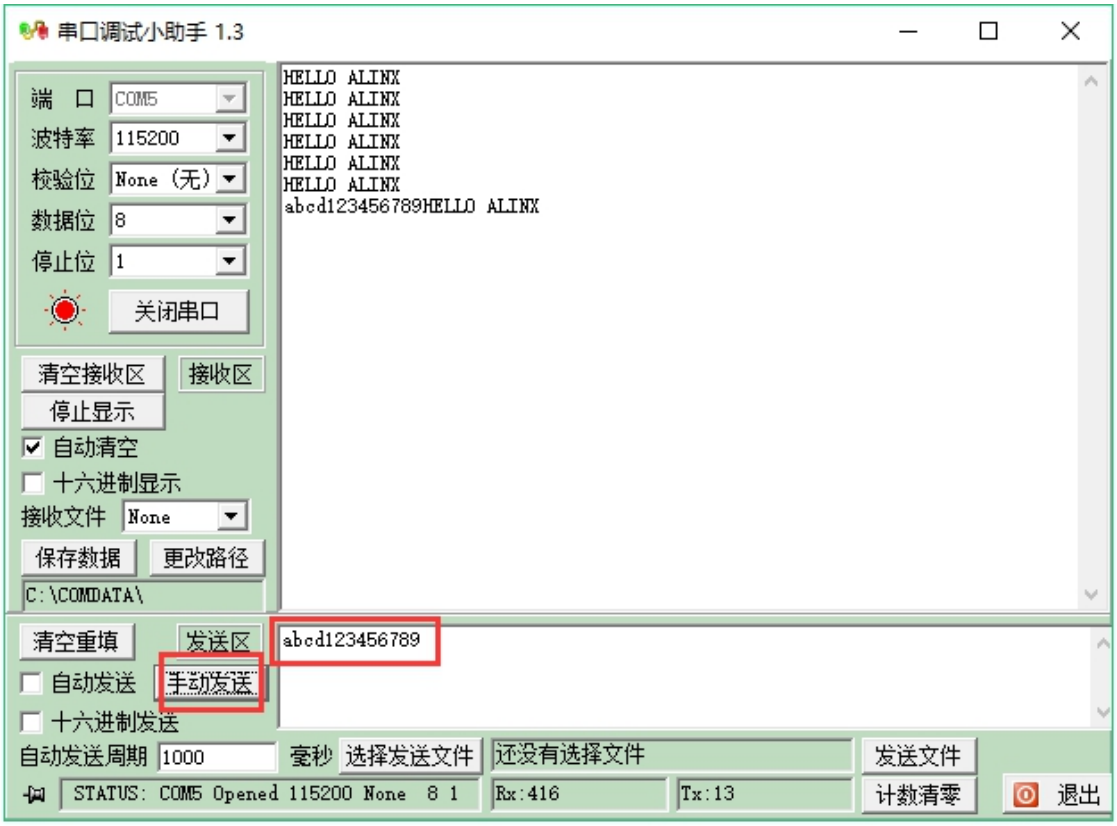
Find the serial port number "COM5" in the device manager



Open the serial port debugging, select "COM 5" for the port (choose according to your own situation), set the baud rate to 115200, select None for the check bit, 8 for the data bit, and 1 for the stop bit, and then click "Open serial port". This software is under the routine folder.



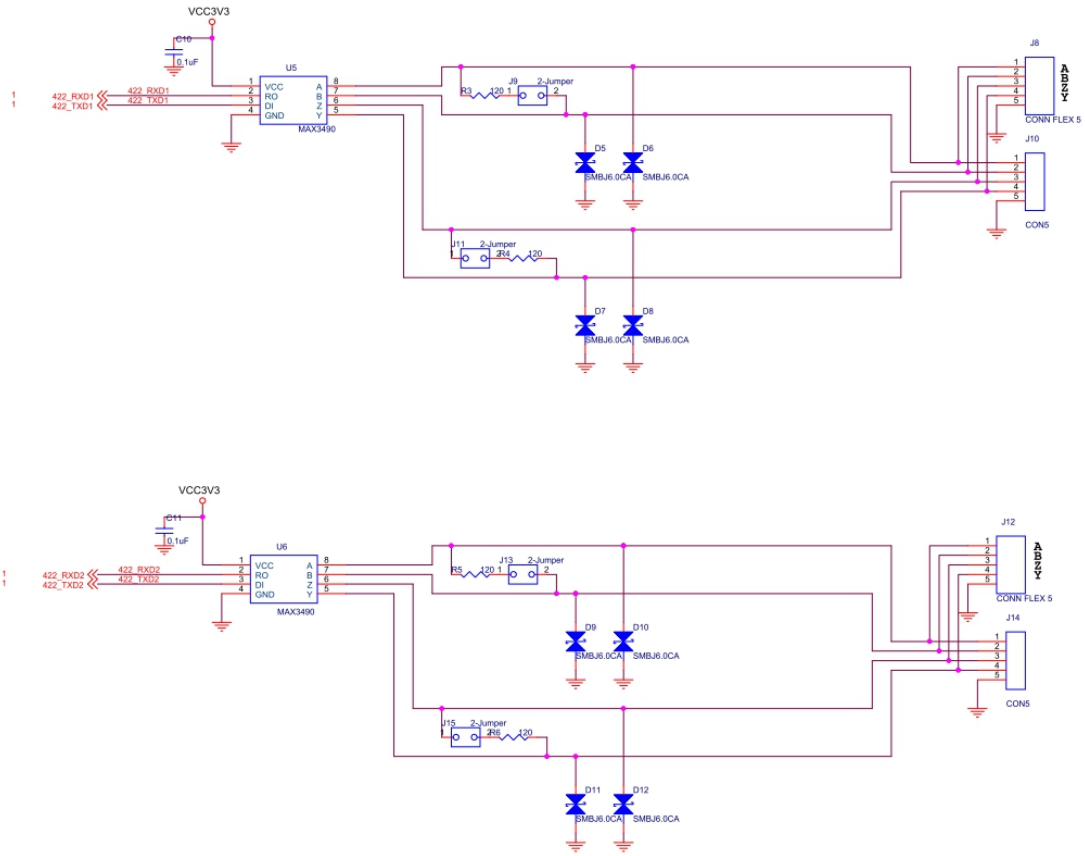
After opening the serial port, you can receive "HELLO ALINX" every second, enter the text you want to transmit in the input box of the transmitting area, click "manual send", you can see that you have received the characters you sent.


串口调试小助手 1.3
端 口
COM5
波特率
115200
校验位
None (无)
数据位
8
停止位
1
关闭串口
清空接收区
接收区
停止显示
自动清空
十六进制显示
接收文件
None
保存数据
更改路径
C:\COMDATA\
HELLO ALINX
HELLO ALINX
HELLO ALINX
HELLO ALINX
HELLO ALINX
HELLO ALINX
abcd123456789HELLO ALINX
清空重填
发送区
abcd123456789
自动发送
手动发送
十六进制发送
自动发送周期
1000
毫秒
选择发送文件
还没有选择文件
发送文件
STATUS: COM5 Opened 115200 None 8 1
Rx:416
Tx:13
计数清零
退出

Part 12: RS422 Experiment

The experimental Vivado project is "rs422_test"

This chapter uses the AN3485 module to realize the RS422 interface data transmission. Regarding the module, it has been introduced in the previous RS232 experiment, so I will not repeat it in this experiment. RS422 and RS232 are the same in the connection interface with FPGA, both are TXD and RXD. Therefore, this experiment is based on the RS232 experiment and instantiates two connections to the RS422 interface chip MAX3490.



RS422 Interface Schematic

Part 12.2: Programming

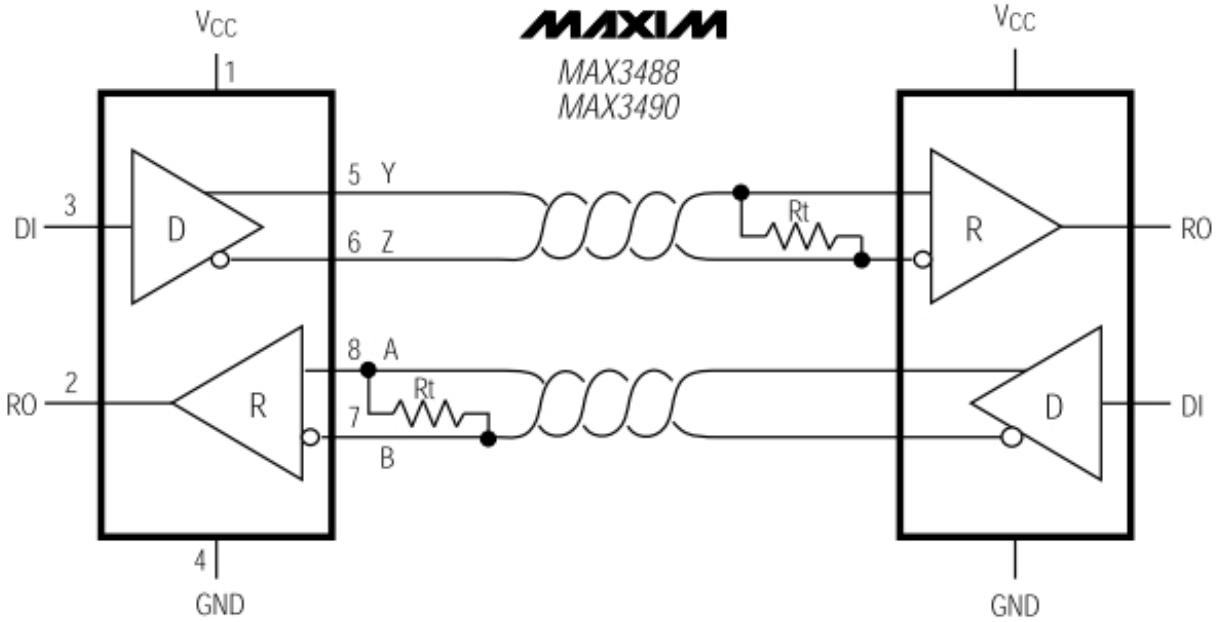
The program design is relatively simple, based on the RS232

experiment, instantiate two uart_tests.

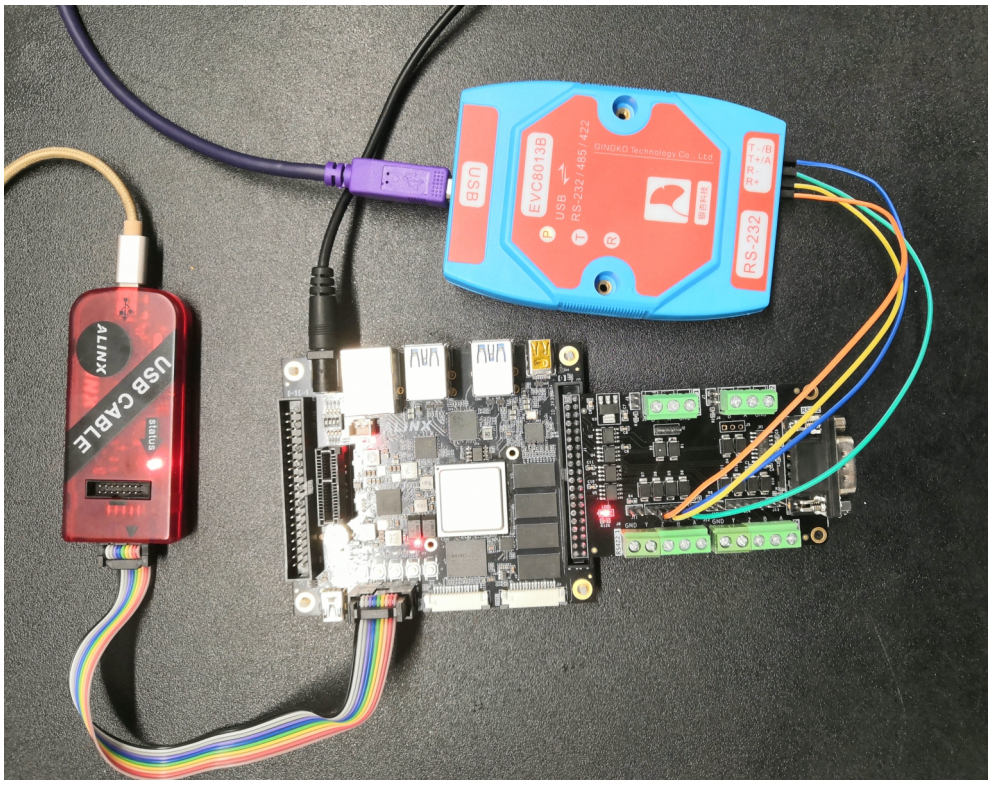
```
module rs422_top(
    input                       sys_clk,       //system clock 25Mhz on board
    input                       rst_n,        //reset ,low active
    input                       rs422_rx1,
    output                      rs422_tx1,
    input                       rs422_rx2,
    output                      rs422_tx2
    );


uart_test   rs422_inst1
(
    .sys_clk        (sys_clk),       //system clock 25Mhz on board
    .rst_n          (rst_n  ),        //reset ,low active
    .uart_rx        (rs422_rx1),
    .uart_tx        (rs422_tx1)
);


uart_test   rs422_inst2
(
    .sys_clk        (sys_clk),       //system clock 25Mhz on board
    .rst_n          (rst_n  ),        //reset ,low active
    .uart_rx        (rs422_rx2),
    .uart_tx        (rs422_tx2)
);

endmodule
```

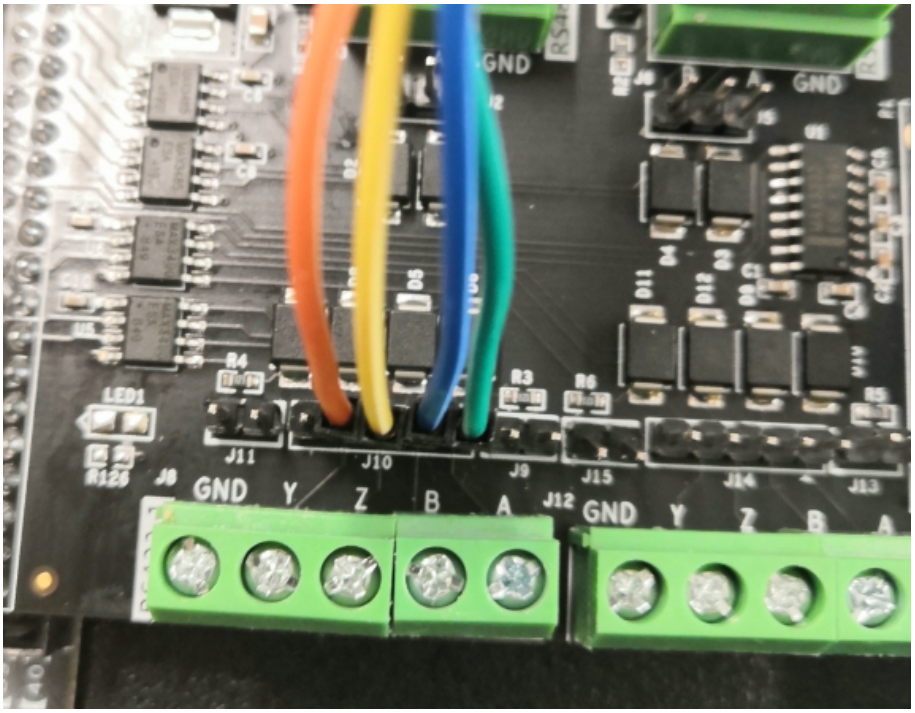
Part 12.3: Experimental Test

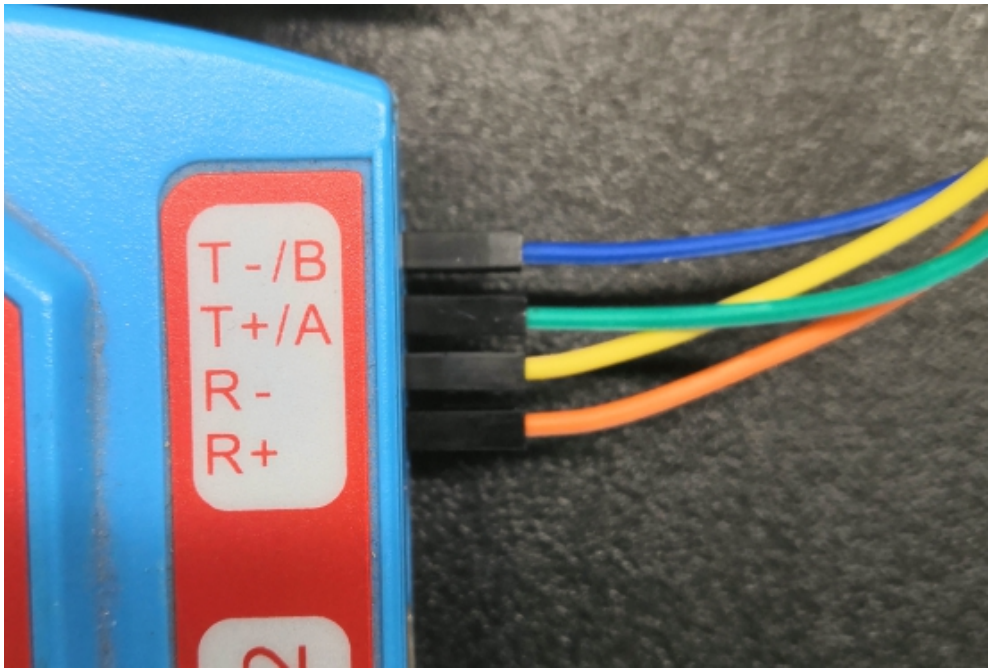
The interface part of RS422 is a differential signal, a total of four signal lines, two transmit TXD+ and TXD-, corresponding to Y and Z, and two receive RXD+ and RXD-, corresponding to A and B.



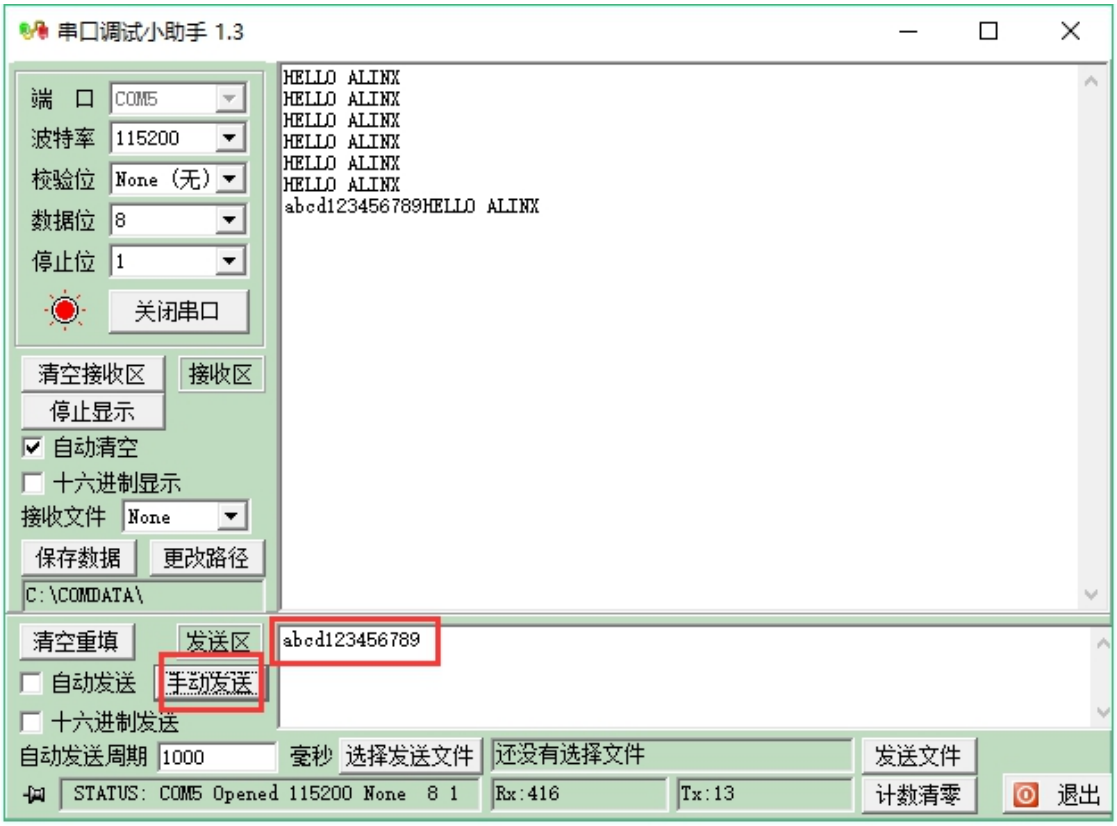
Same as the RS232 experiment, the device connection from USB to serial port is also used. Use a Du ties cable to connect Y and Z of RS422_1 of the module to R+ and R- of the device, and connect A and B of the module to T+ and T- of the device connection.







After downloading the program, you can see the same effect of the RS232 experiment in the serial port tool, and you can use the same method to test the RS422_2 interface.



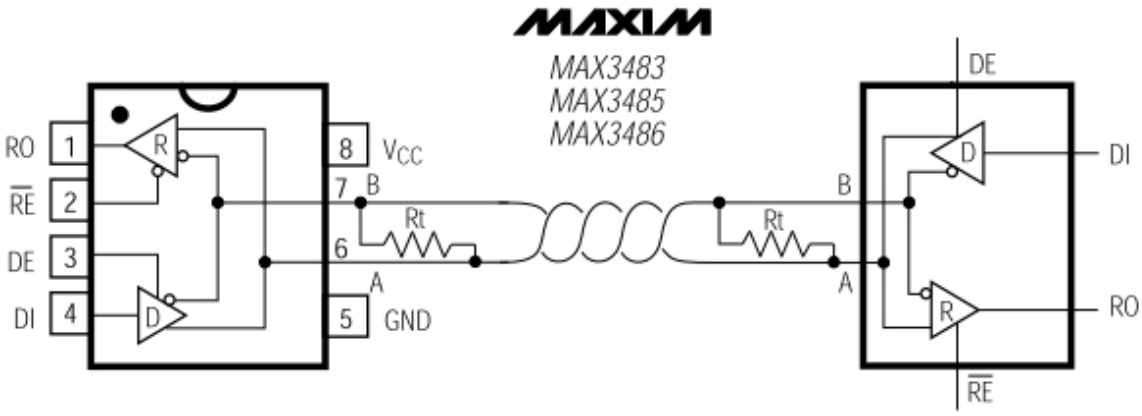
Part 13: RS485 Experiment

The experimental Vivado project is "rs485_test".

This chapter introduces RS485 data transmission with AN3485 module.

Part 13.1:Experimental principle

The experiments of RS232 and RS422 were introduced earlier, and RS485 is similar to RS422, and uses differential signal transmission, but RS485 is half-duplex transmission, that is, there can only be one direction of data transmission at the same time. And the interface is less than RS422, only differential signal A and B, and the signal connected with ARM or FPGA is DE (direction selection), DI (input signal TXD), RO (output signal RXD).



In the MAX3485 documentation, transmit direction. If DE is 1, that is, the output is enabled. When the DI value is 1, the corresponding differential signal A and B values are 1 and 0; otherwise, they are 0 and 1.

Table 1. Transmitting

INPUTS			OUTPUTS		MODE
$\overline{RE}$	DE	DI	B*	A*	
X	1	1	0	1	Normal
X	1	0	1	0	Normal
0	0	X	High-Z	High-Z	Normal
1	0	X	High-Z	High-Z	Shutdown

From the receiving point of view, if DE is 0 and the difference between A and B is greater than or equal to +0.2 V, the RO value is 1, otherwise it is 0

Table 2. Receiving

INPUTS			OUTPUTS	MODE
$\overline{RE}$	DE	A, B	RO	
0	0*	≥ +0.2V	1	Normal
0	0*	≤ -0.2V	0	Normal
0	0*	Inputs Open	1	Normal
1	0	X	High-Z	Shutdown

Part 13.2: Programming

Since RS485 is a half-duplex transmission, then we need to develop a transmission protocol for handshake. Set the first byte to 8'h55 to indicate the beginning of a frame of data. The next is the length of the transmitted data. Due to the FIFO size limit (256), the range is 1~255, and the next is data. The format is: starting 8 ' h55+data length+data.

Among them, uart_tx and uart_rx are the same as UART experiment, here only need to modify uart_test. The function we designed is to set DE to 0 in the initial state, that is, input, wait to receive the data from the host computer, and buffer it in the FIFO, set the FIFO size to 256, and then switch DE to 1, that is, output. The received data is read from the FIFO and sent out. Note that the

cached data is without the initial 8'h55 and the quantity information.

In RCV_HEAD state, judge whether the received data is "S"

```
RCV_HEAD:
begin
    if (rx_data_valid == 1'b1 && rx_data == "s")     //when
        state <= RCV_COUNT ;
end
```

In the RCV_COUNT state, if the data length is less than 0, it will jump to the IDLE state, if it is greater than 0, it will enter the receiving data state.

```
RCV_COUNT:
begin
    if (rx_data_valid == 1'b1)  //recor
    begin
        if (rx_data > 0)        //if da
            state <= RCV_DATA ;
        else
            state <= IDLE ;

        data_count <= rx_data ;
    end
end
```

In the RCV_DATA state, write data into the FIFO, check the data length, switch the direction of RS485 to output, and jump to the state.

```
RCV_DATA:
begin
    fifo_wren  <= rx_data_valid ;
    fifo_wdata <= rx_data ;

    if (rx_data_valid == 1'b1)
    begin
        if (rx_cnt == data_count - 1)   //the last received data
        begin
            rx_cnt      <= 8'd0 ;
            rs485_de    <= 1'b1 ;
            state       <= WAIT ;
        end
        else
            rx_cnt <= rx_cnt + 1'b1 ;
    end
end
```

When switching the bus state, in order to work reliably, in the WAIT state, delay 1ms to switch the direction.

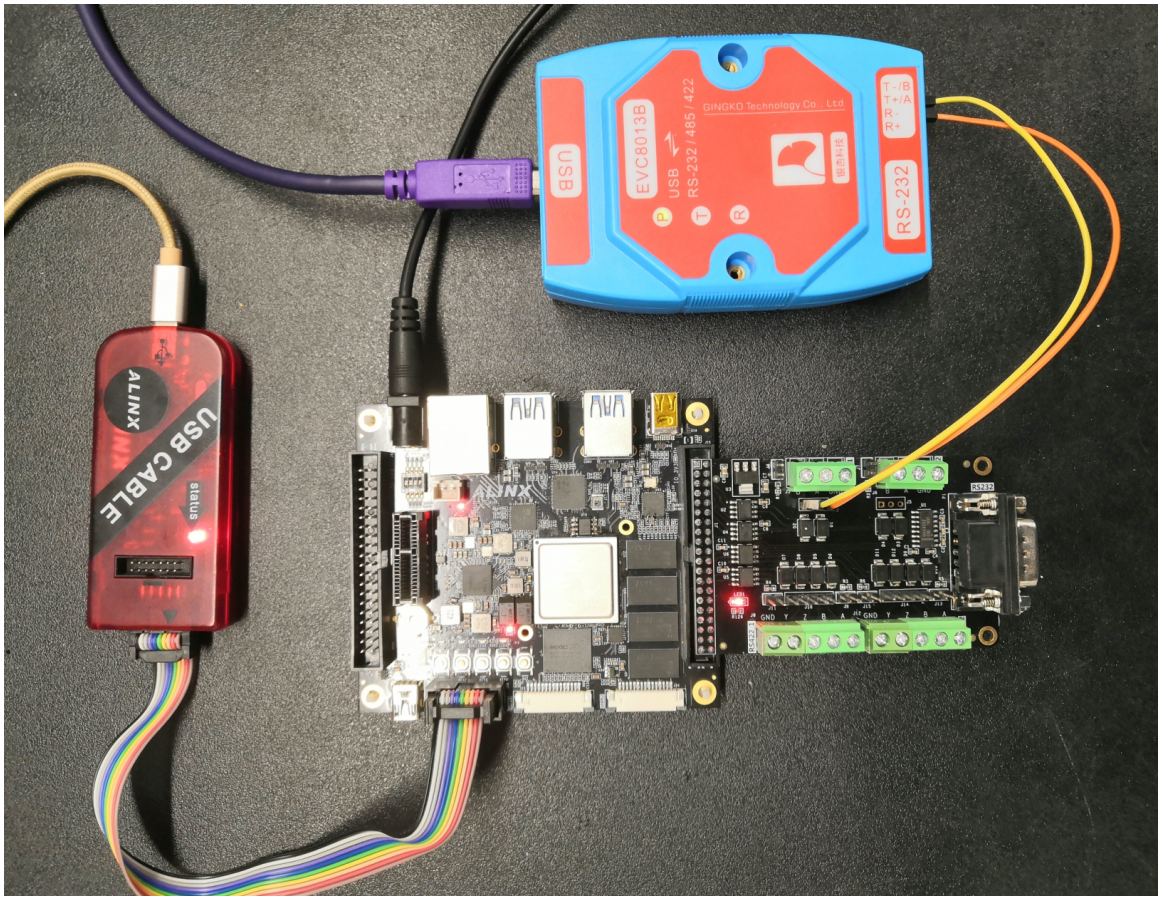
```
WAIT:
begin
    fifo_wren  <= 1'b0 ;
    if (wait_cnt >= CLK_FRE * 1000) // wait for 1 ms for direction change
    begin
        wait_cnt <= 32'd0 ;
        state <= SEND_WAIT;
    end
    else
    begin
        wait_cnt <= wait_cnt + 32'd1;
    end
end
```

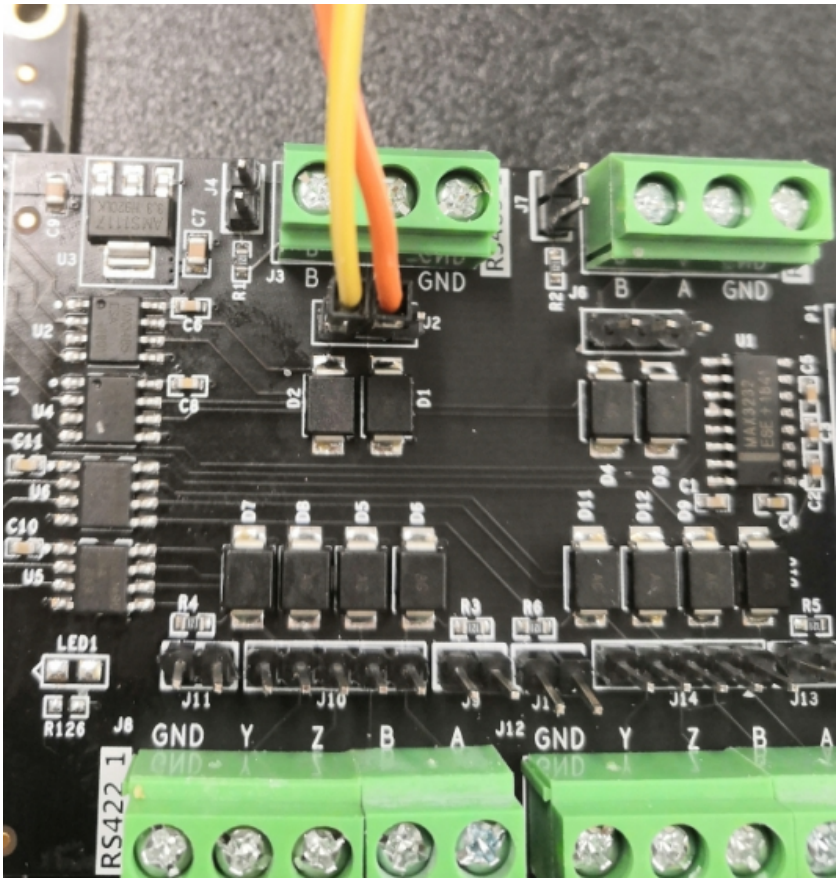
Then the data in the FIFO is transmitted. The SEND_WAIT state controls the read enable signal “fifo_rden” and judges whether the data is transmitted. After transmitting, it enters the IDLE state.

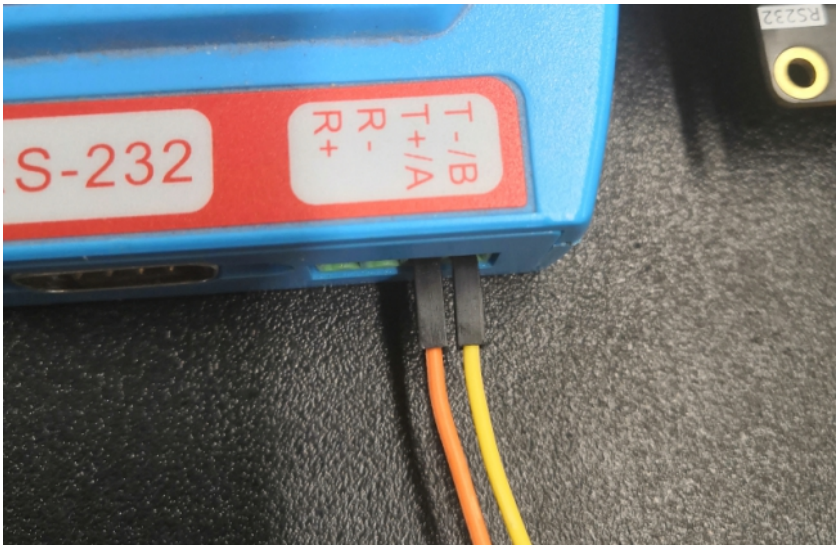
```
SEND_WAIT:
begin
    if (tx_data_ready == 1'b1)
    begin
        if (tx_cnt == data_count)           //the last data has transferred
        begin
            tx_cnt          <= 8'd0 ;
            fifo_rden       <= 1'b0 ;
            state           <= IDLE ;
        end
        else
        begin
            fifo_rden       <= 1'b1 ;       //read data from fifo
            state           <= SEND ;
        end
    end
    tx_data_valid   <= 1'b0 ;
end
SEND:
begin
    fifo_rden       <= 1'b0 ;
    tx_data_valid   <= 1'b1 ;

    if(tx_data_valid == 1'b1 && tx_data_ready == 1'b1 && tx_cnt < data_count)
    begin
        tx_cnt <= tx_cnt + 8'd1; //Send data counter
        state  <= SEND_WAIT ;
    end
end
```

Part 12.3: Experimental Results

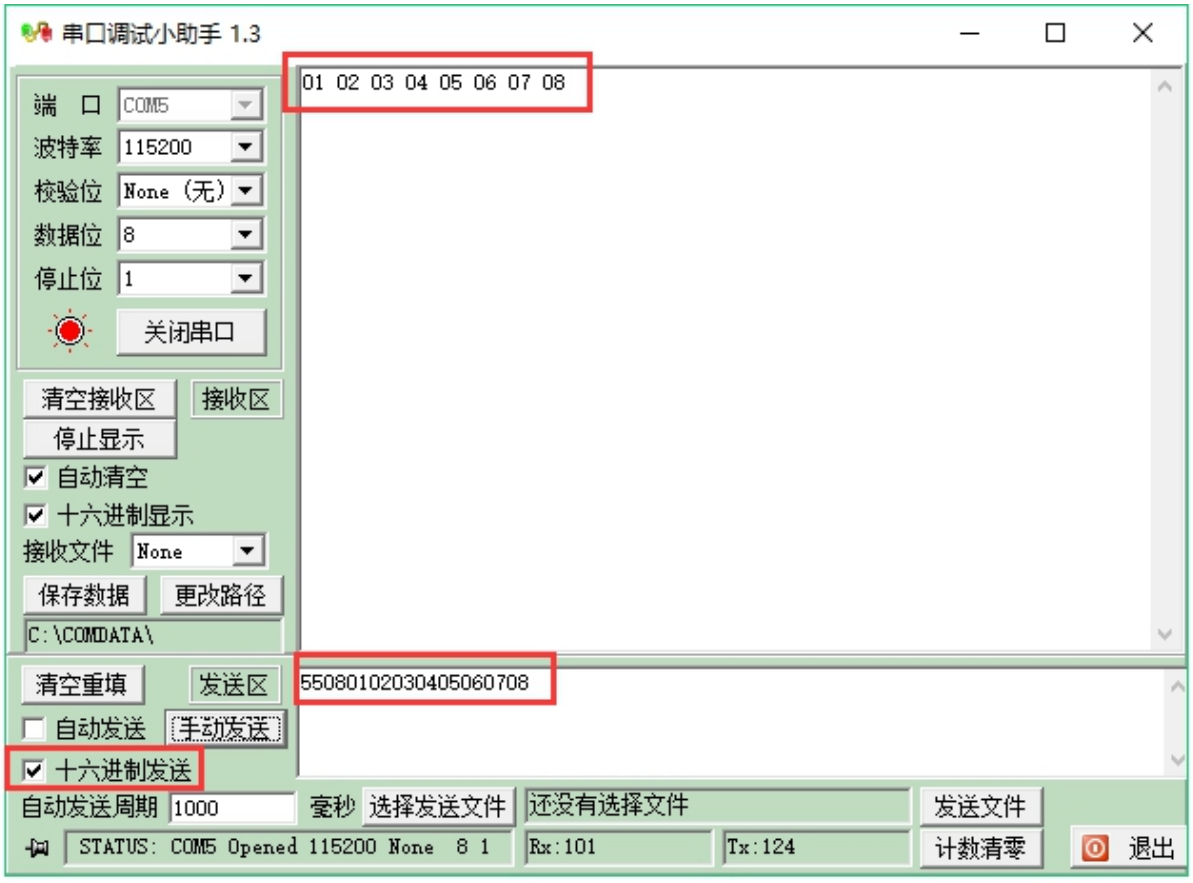
We still use USB Uart device, and connect “A” and “B” of “RS485_1” to “A” and “B” of the device through DuPont cable.


USB
EVC8013B
GINGKO Technology Co., Ltd
USB
RS-232 / 485 / 422
RS-232
ALINX
USB CABLE
status


GND
B
A
GND
LED1
R126
J8
GND
Y
Z
B
A
J12
GND
Y
Z
B
A
RS422_1



Open the serial port tool, set the baud rate of the serial port number, select hexadecimal to transmitted, and transmit data starting with 8'h55, click transmit, you can see the returned data in the receiving window.



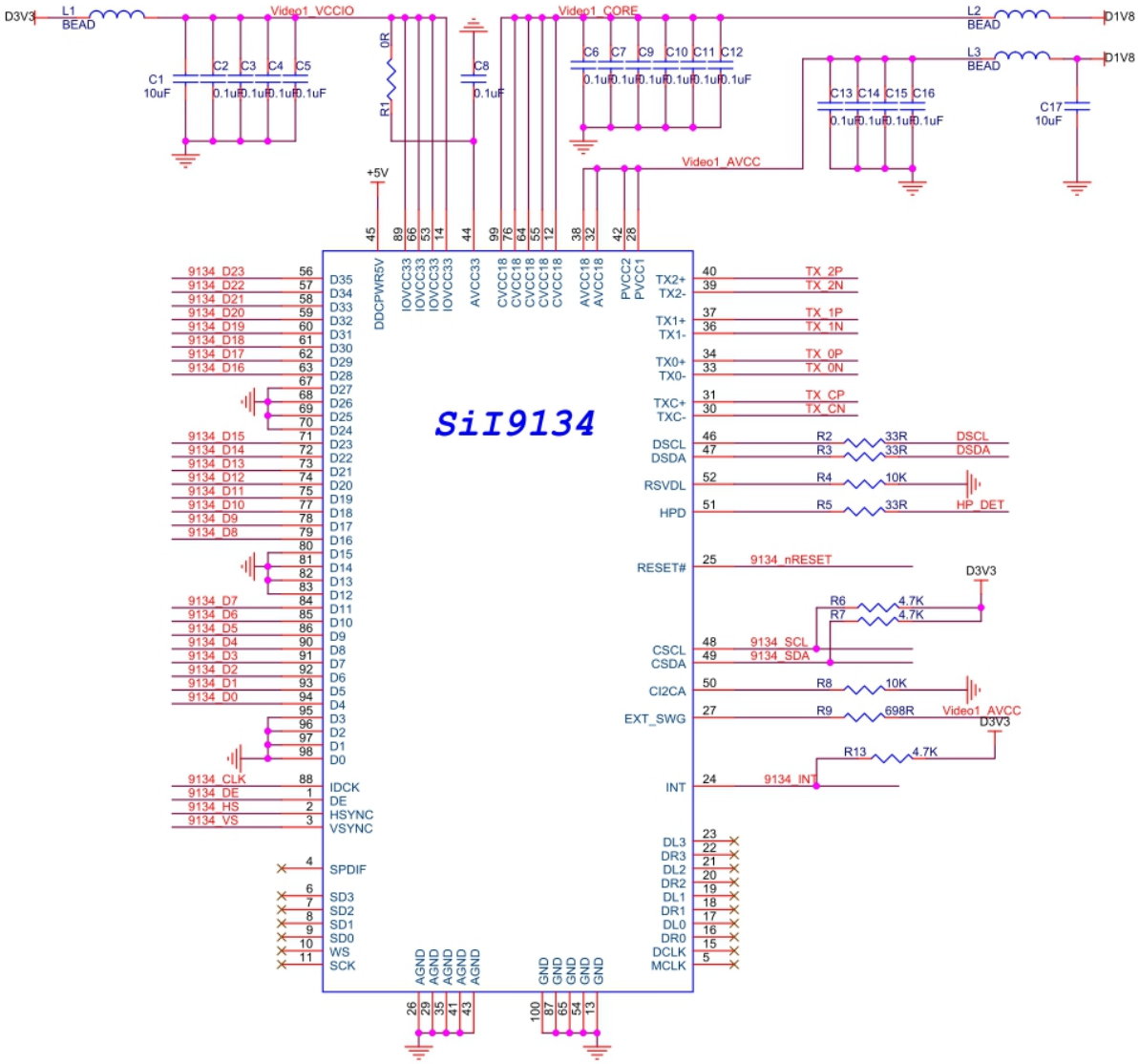
Part 14: HDMI Output Experiment (AN9134 Module)

Experiment Vivado project is "hdmi_output_test"

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.

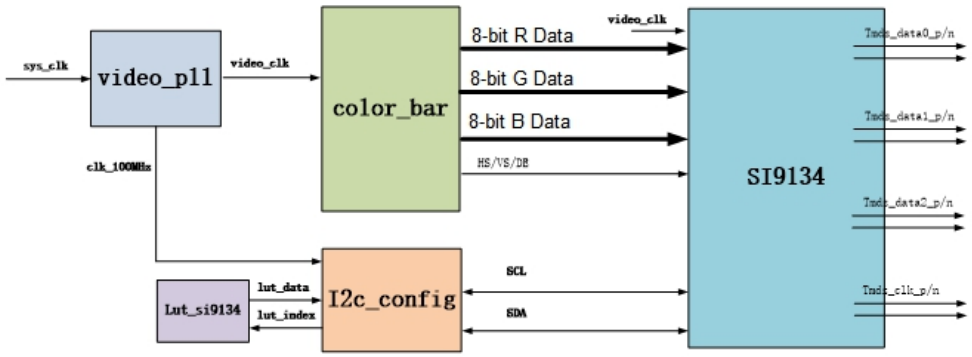
Part 14.1: Hardware introduction

Since only DP can be displayed on the development board, but it is on the PS side, and there is no HDMI interface on the PL side, we use the AN9134 HDMI expansion module to achieve HDMI display. The 24-bit RGB code output TMDS differential signal. SIL9134 is powerful, and only a small part of it is used in this experiment to display the RGB24 video data.



The SI9134 chip needs to configure the register through the I2C bus to work normally. From the schematic diagram, it can be seen that the I2C bus is connected to the IO on the PL side and can be directly configured through the PL.

Part 14.2: Programming



This experiment realizes the display of color bars through HDMI. In the experiment, the video timing generation and color bar generation module "color_bar.v", the I2C Master register configuration module "i2c_config.v", and the configuration data look-up table module "lut_si9134.v" are designed.

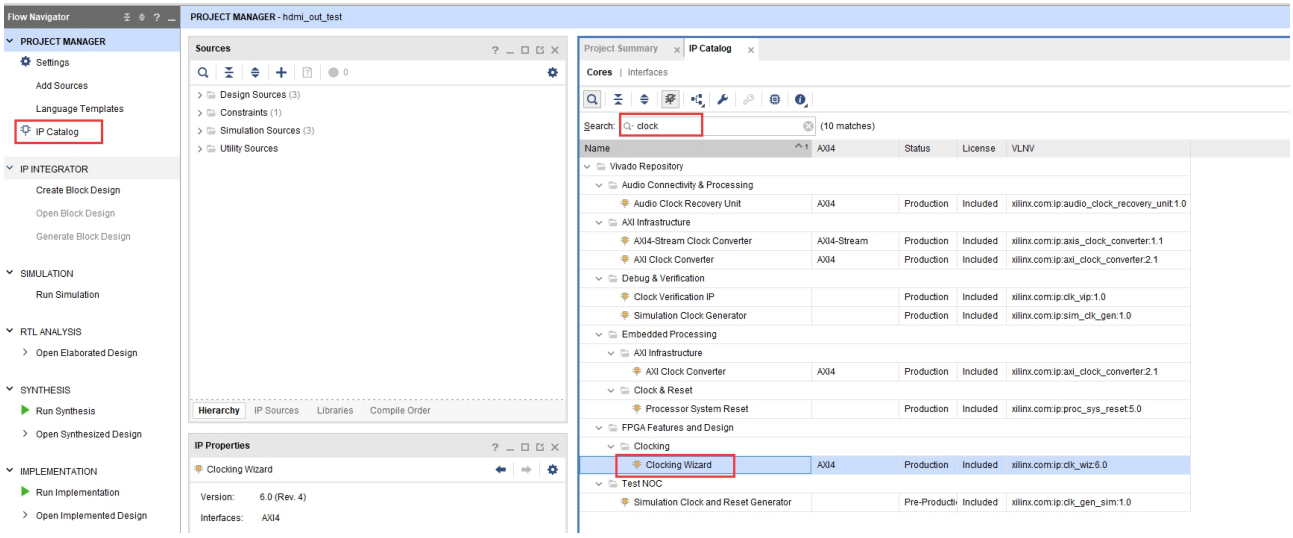
The specific code will not be introduced one by one here, let's go and see for yourself. Let me give you a brief introduction to the functions implemented by each module:

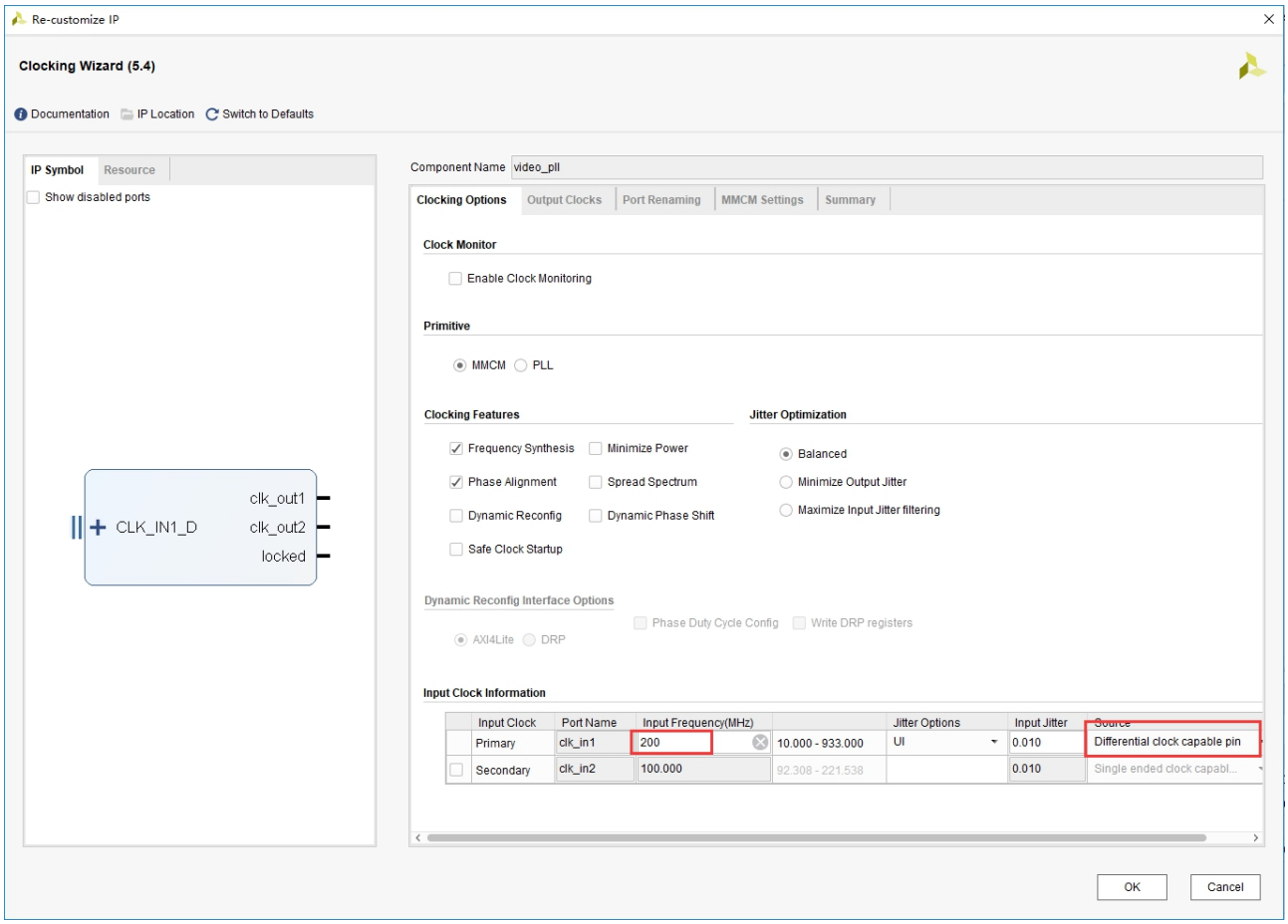
The top-level module top.v is the top-level file of the project, which mainly instantiates 4 sub-modules (clock module vidio_pll, color bar generation module color_bar, I2C configuration module i2c_config, and configuration lookup table module lut_si9134).

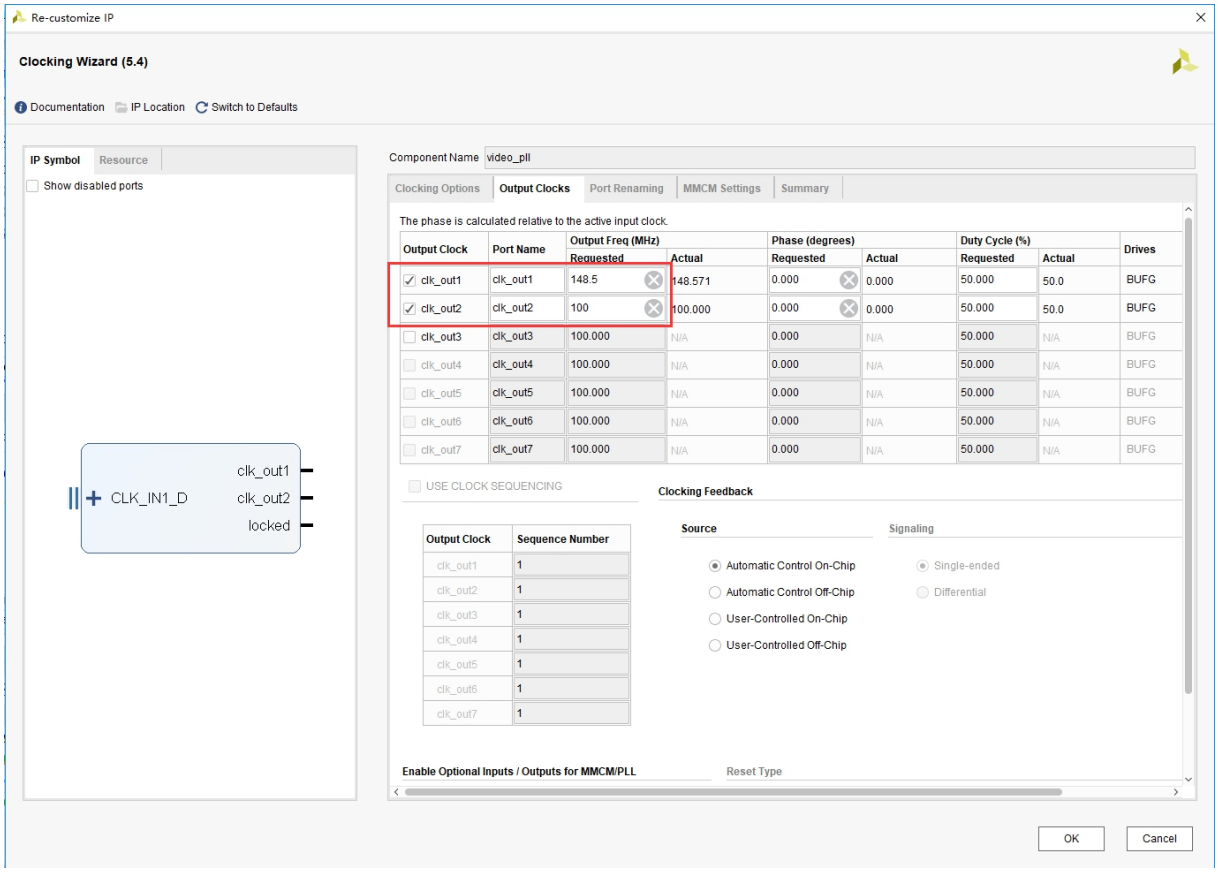
The color bar generation module color_bar.v is a VGA format color bar that generates 8 colors. The color bars are white, yellow, cyan, green, purple, red, blue, and black. Produces a color bar with a resolution of 1920x1080 and a refresh rate of 60Hz, which is the so-called 1080P high-definition video image. So this module will output R (8 bits) G (8 bits) B (8 bits) image signals, row synchronization, column synchronization and data valid signals.

The clock module video_pll calls a clock IP provided by Xilinx, and generates a 100Mhz clock and a 1080P pixel clock of 148.5Mhz

through the input system clock. The method to generate the clock IP is to click the IP Catalog under the Project Manager directory, and then select the FPGA Features and Design->Clocking->Clocking Wizard icon.

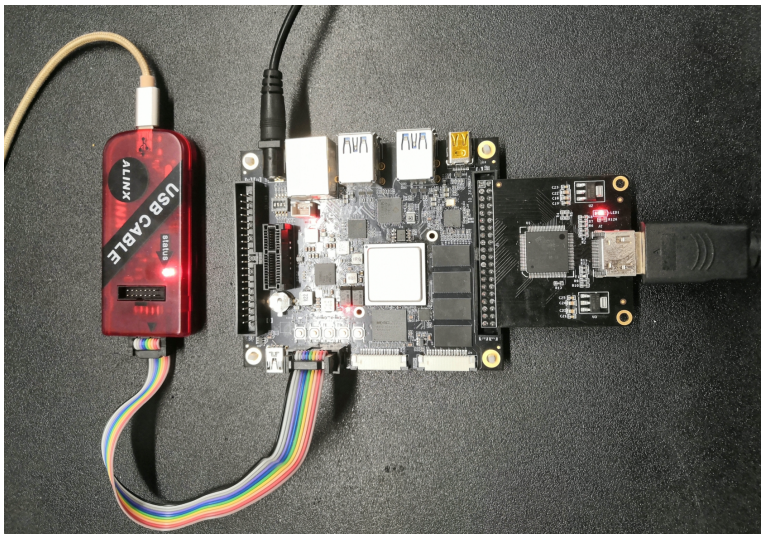






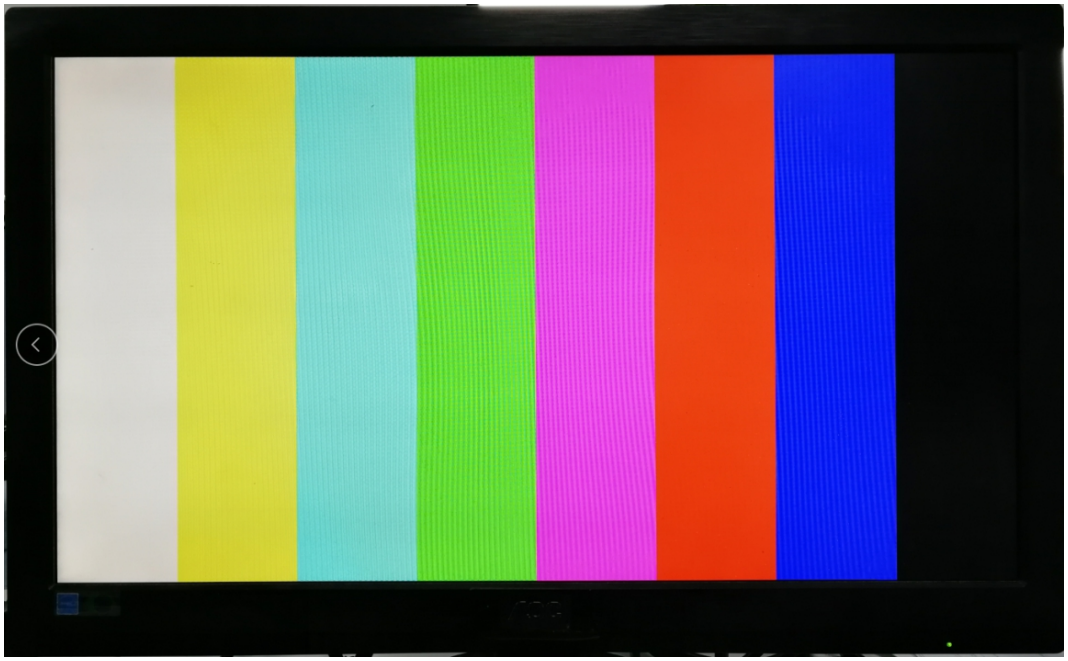
Part 14.3: Download Debugging

Save the project and compile to generate a bit file, connect the HDMI module to the J15 expansion port, and connect the HDMI interface to the HDMI display. Note that 1920x1080@60Hz is used here. Please make sure that your display supports this resolution.



Hardware Connection (Expansion Port J45)

After downloading, the display shows the following image



Part 14.4: Experimental Results

This experiment is initially uses 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 PL is used to drive the HDMI chip, including I2C register configuration. Of course, the configuration of I2C is better to use PS to configure.

Part 15: HDMI Character Display Experiment

The experimental Vivado project is "hdmi_char"

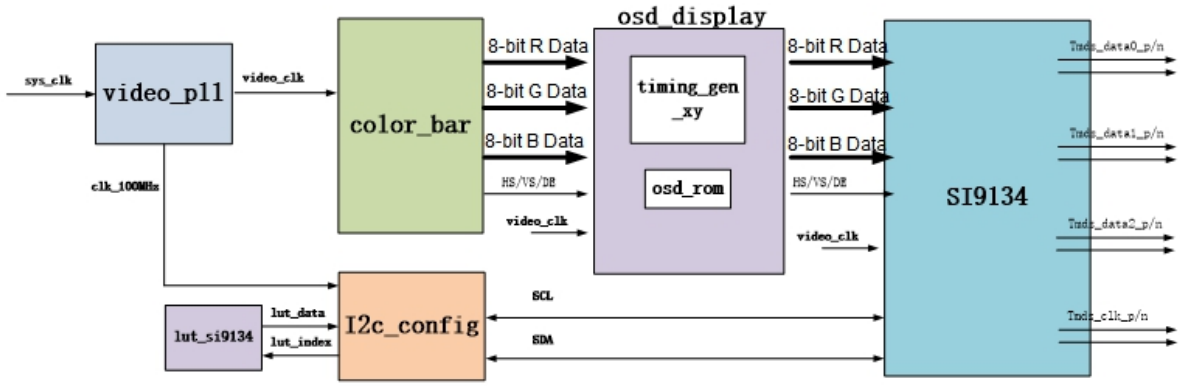
In the HDMI output experiment, the HDMI display principle and display method are explained. This experiment introduces how to use FPGA to realize character display. Through this experiment, you can have a deeper understanding of HDMI display methods.

Part 15.1: Experimental Principle

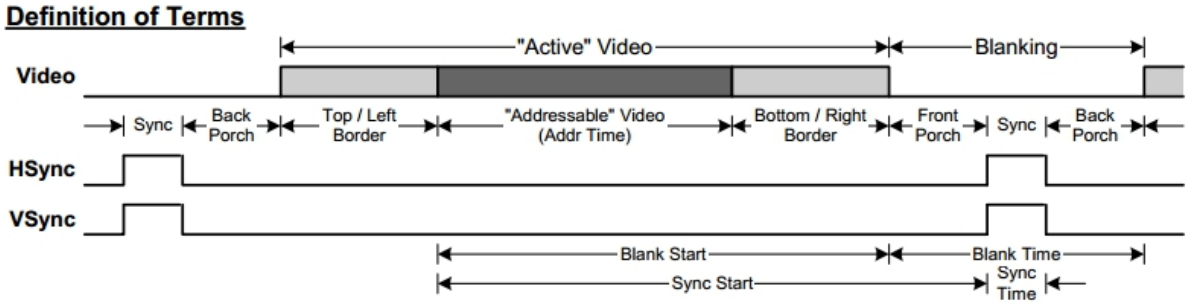
The experiment uses the character conversion tool to convert characters into hexadecimal coe files and store them in the single-port ROM IP core, and then read the converted data from the ROM and display it on HDMI.

Part 15.2: Programming

The character display routine adds an osd_display module to the HDMI display. The "osd_display" module is used to read the converted character information stored in the Rom ip core and display it in the designated area. The block diagram is shown in the figure below:



1) In the "timing_gen_xy" module, two counters "x_cnt" and "y_cnt" are set according to the HDMI timing standard, and these two counters generate the "x" and "y" coordinates displayed by HDMI. In the program, "vs_edge" and "de_falling" are used to represent the field sync start signal and the data valid end signal respectively. The principle is shown in the figure below:



Signal Name	Direction	Description
rst_n	in	Asynchronous reset input, low reset
clk	in	external clock input
i_hs	in	line sync input
i_vs	in	field sync input
i_de	in	Data valid signal
i_data	in	color_bar Data
o_hs	out	Output data valid signal
o_vs	out	Output field sync signal
o_de	out	Output data valid signal
o_data	out	output data
x	out	generate coordinate x
y	out	generate coordinate y

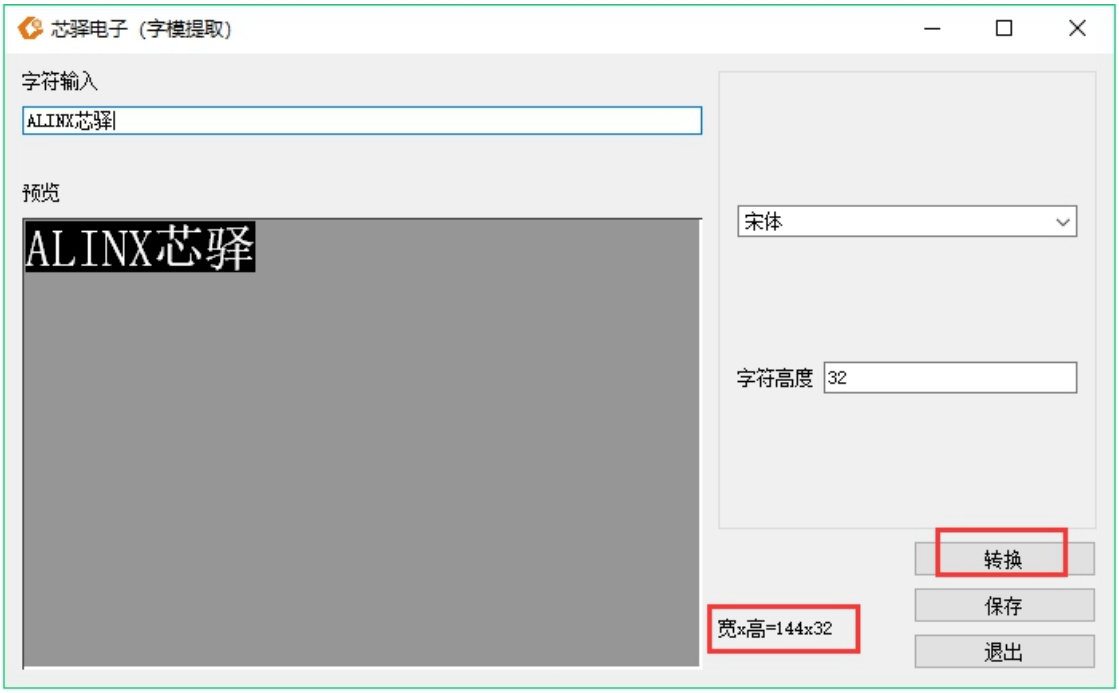
time_gen_xy module port

2) The following describes how to store the ROM IP of text information. First, you need to generate a .coe file that can be recognized by XILINX FPGA. First find the "FPGA font extraction" tool under the project folder.

FPGA字模提取.exe

Double-click the ".exe" file to open the tool

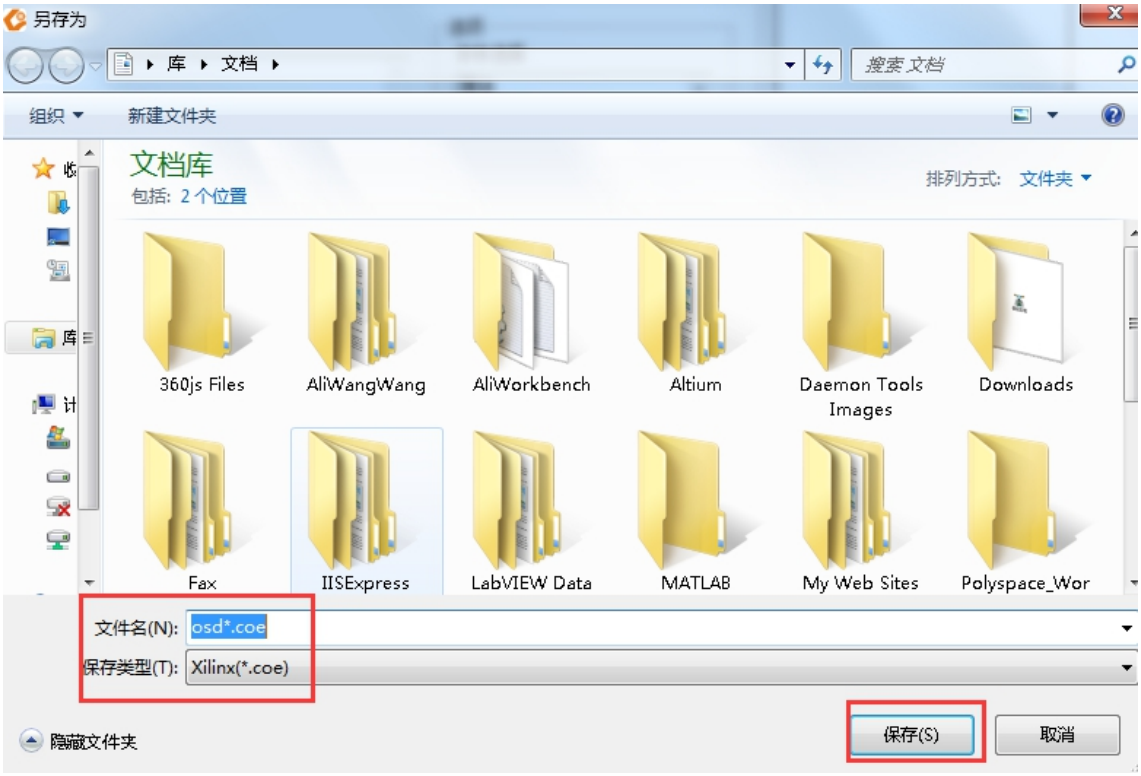
Enter the characters to be displayed in the "Character Input字符输入" box of the extraction tool. The font and character height can be customized. Click the "Conversion转换" button after setting. In the lower left corner of the interface, you can see the size of the converted character dot matrix. The width and height of the dot matrix are needed in the program.



The width and height of the dot matrix here are 144x32, which need to be consistent with those defined in the osd_display program:

```
parameter OSD_WIDTH  =  12'd144;
parameter OSD_HEGIHT =  12'd32;
```

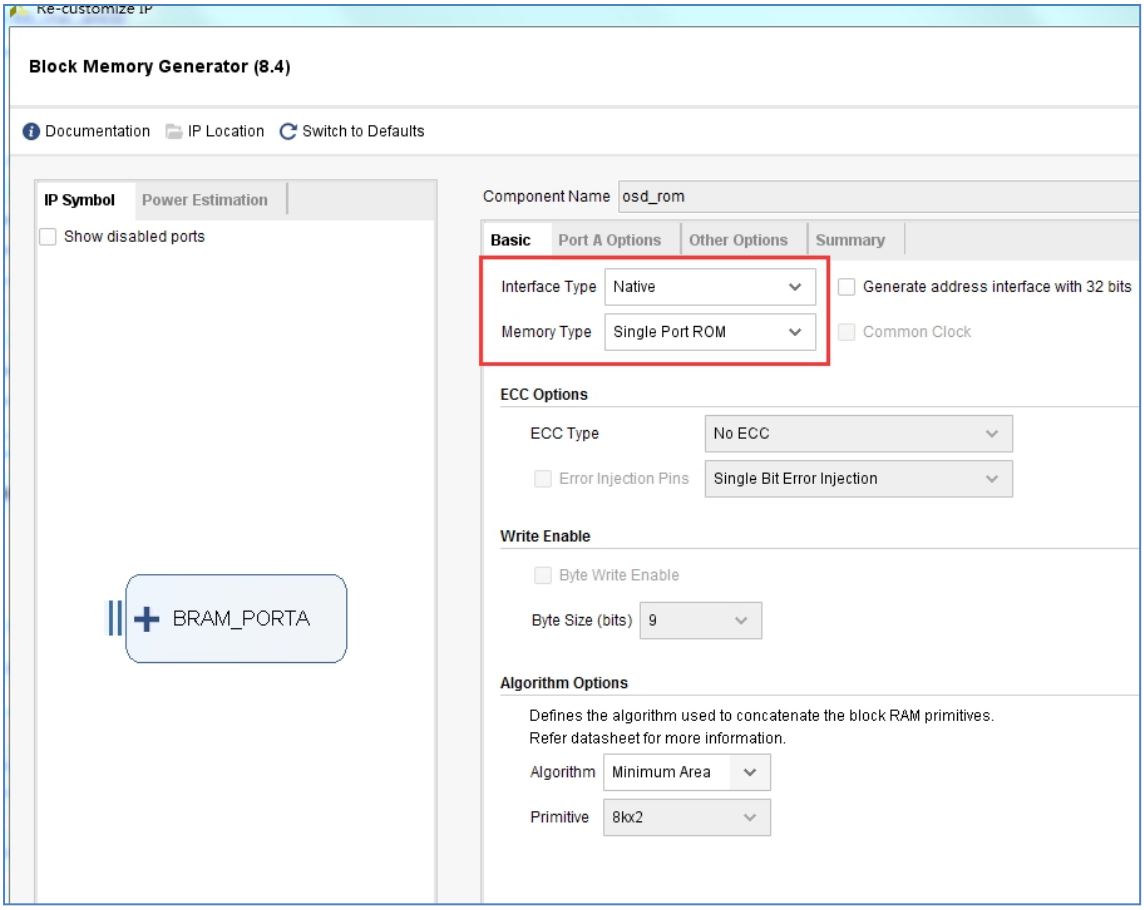
Click the "Save保存" button to save the file to the source file directory of this routine. It should be noted that Xilinx coe should be selected under the save type and click the "Save保存" button.



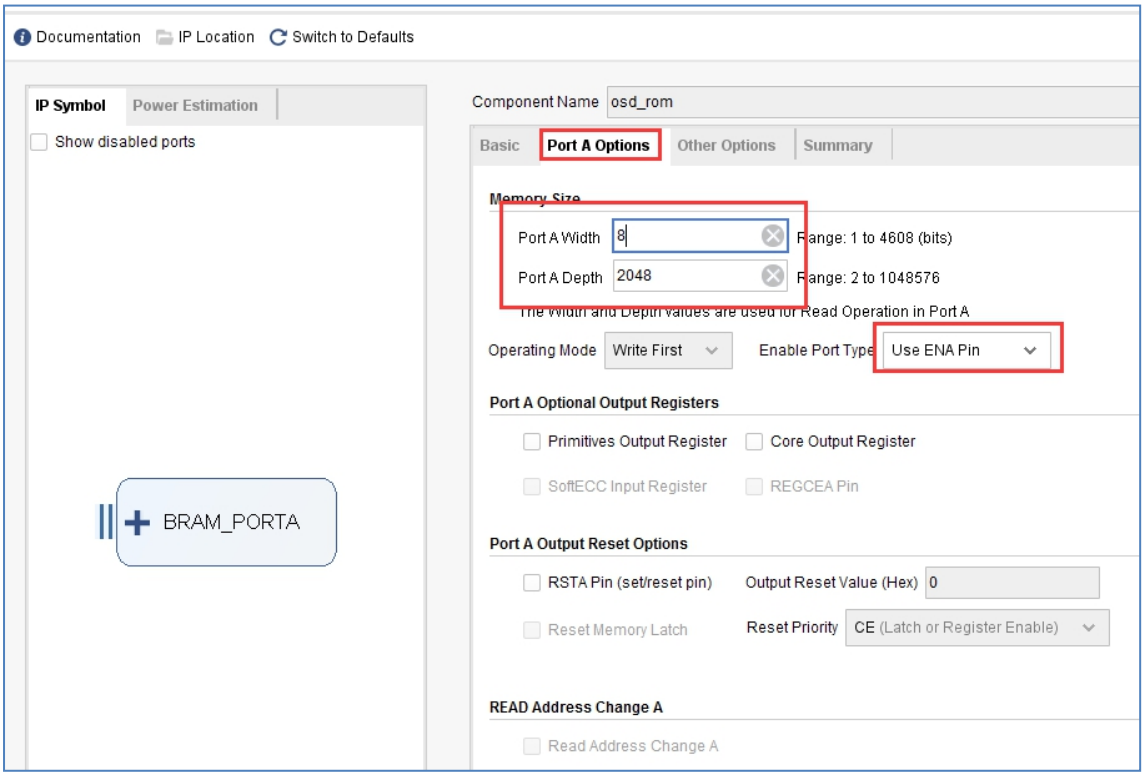
After you find the generated ".coe" file and open it, you can see the following:

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00,
00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00,
00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 04,
00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00,
00, 1C, 38, 00, 00, 00, 00, 03, 00, 00, 00, 00, 00, 00, 00, 00,
00, 00, 00, 0C, 38, 00, 00, E6, FF, 07, 80, 01, 00, 00, 00, 00,
00, 00, 00, 00, 00, 0C, 18, 08, FC, 0F, 01, 03, C0, 01, 7E, 00,
F8, 1F, 0F, FC, 3E, 7E, 00, 0C, 18, 1C, 00, 06, 82, 01, C0, 01,
18, 00, 80, 01, 1C, 10, 1C, 18, FC, FF, FF, 3F, 30, 06, C6, 00,
C0, 01, 18, 00, 80, 01, 1C, 10, 18, 08, 00, 0C, 18, 00, 30, 06,
EC, 00, C0, 03, 18, 00, 80, 01, 3C, 10, 18, 0C, 00, 0C, 18, 00,
10, 06, 78, 00, 60, 03, 18, 00, 80, 01, 34, 10, 30, 04, 00, 0C,
18, 00, 18, 02, 38, 00, 20, 03, 18, 00, 80, 01, 74, 10, 30, 06,
00, 04, 08, 00, 18, 03, FC, 00, 20, 03, 18, 00, 80, 01, 64, 10,
60, 02, 00, 60, 00, 00, 18, 03, C6, 03, 20, 03, 18, 00, 80, 01,
E4, 10, 60, 03, 00, C0, 00, 00, 18, 83, 01, 3F, 30, 06, 18, 00,
80, 01, C4, 10, C0, 01, 00, 82, 03, 00, 18, 63, 38, 0C, 10, 06,
```

The process of calling the single-port Rom IP core has been introduced in the previous ROM use, set to Single Port ROM

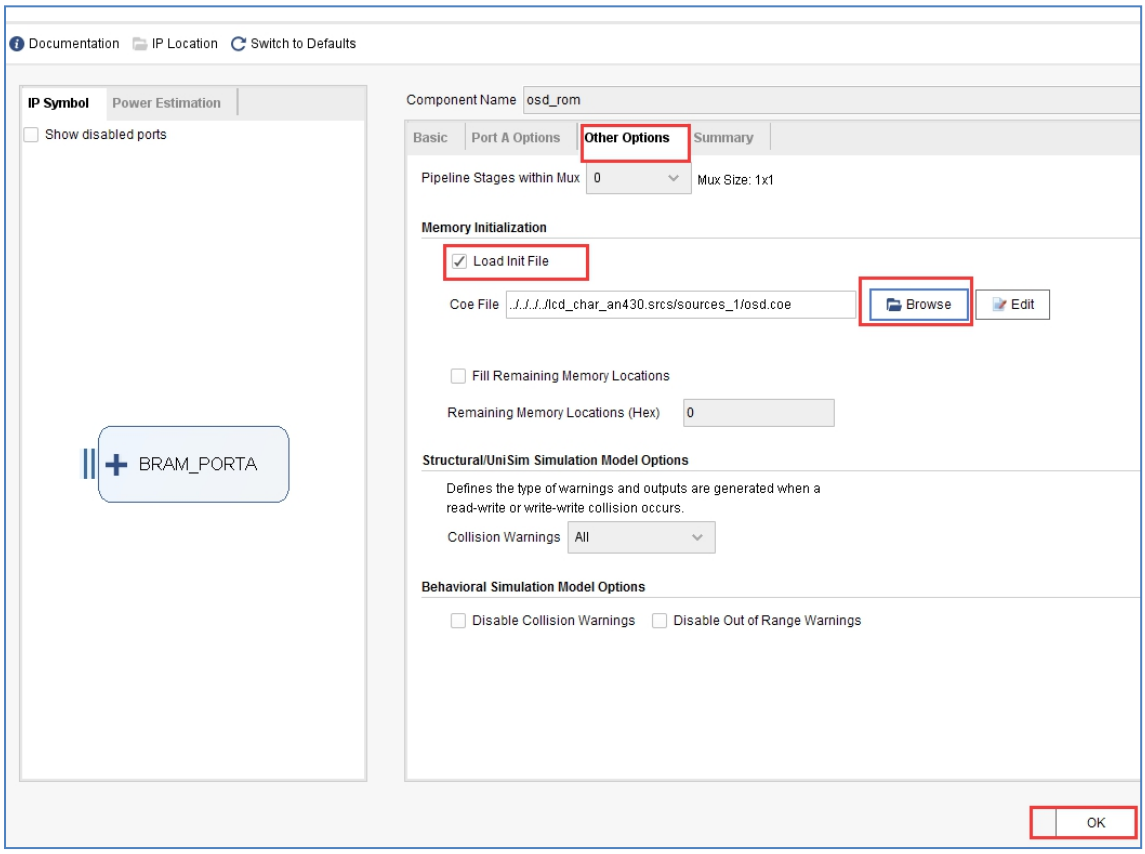


Set as follows in the “Port A Options” column:



Add the “osd.coe” file as shown below (find the previously

generated coe file), and click the "ok" button after completion:



3) The “osd_display” module includes “timing_gen_xy” module and “osd_rom module”. For the character data stored in “osd_rom”, if the data is 1, the OSD area displays the foreground red in the ROM (display “ALINX芯驿”), if the data is 0, the OSD area displays the data as the background color (color bar).

```
always@(posedge pclk)
begin
    if(region_active_d0 == 1'b1)
        if(q[osd_x[2:0]] == 1'b1)
            v_data <= 24'hff0000;
        else
            v_data <= pos_data;
    else
        v_data <= pos_data;
end
```

Set the region valid signal, that is, the characters are displayed in this area, and the starting coordinates are set to (9, 9), the area size

can be set according to the area set by the character generator.

```
always@(posedge pclk)
begin
    if(pos_y >= 12'd9 && pos_y <= 12'd9 + OSD_HEGIHT - 12'd1 && pos_x >= 12'd9 && pos_x  <= 12'd9 + OSD_WIDTH - 12'd1)
        region_active <= 1'b1;
    else
        region_active <= 1'b0;
end
```

Many people may not understand the read address part of ROM, why is it [15: 3]? That is to say, only one data is read out in eight clock cycles. This is because one point of a character only represents 1 bit, and the width of ROM storage data is 8 bits. Therefore, it takes eight cycles to fetch a piece of data, and compare the value of each bit to convert a character point into a pixel on the image.

```
osd_rom osd_rom_m0 (
    .clka                   (pclk                   ),
    .ena                    (1'b1                   ),
    .addra                  (osd_ram_addr[15:3]     ),
    .douta                  (q                      )
);
```

Signal Name	Direction	Description
rst_n	in	asynchronous reset input, low reset
pclk	in	external clock input
i_hs	in	line sync signal
i_vs	in	field sync signal
i_de	in	Data Signal Enable
i_data	in	color_bar Data
o_hs	out	output line sync signal
o_vs	out	output field sync signal
o_de	out	output data Enable
o_data	out	output data

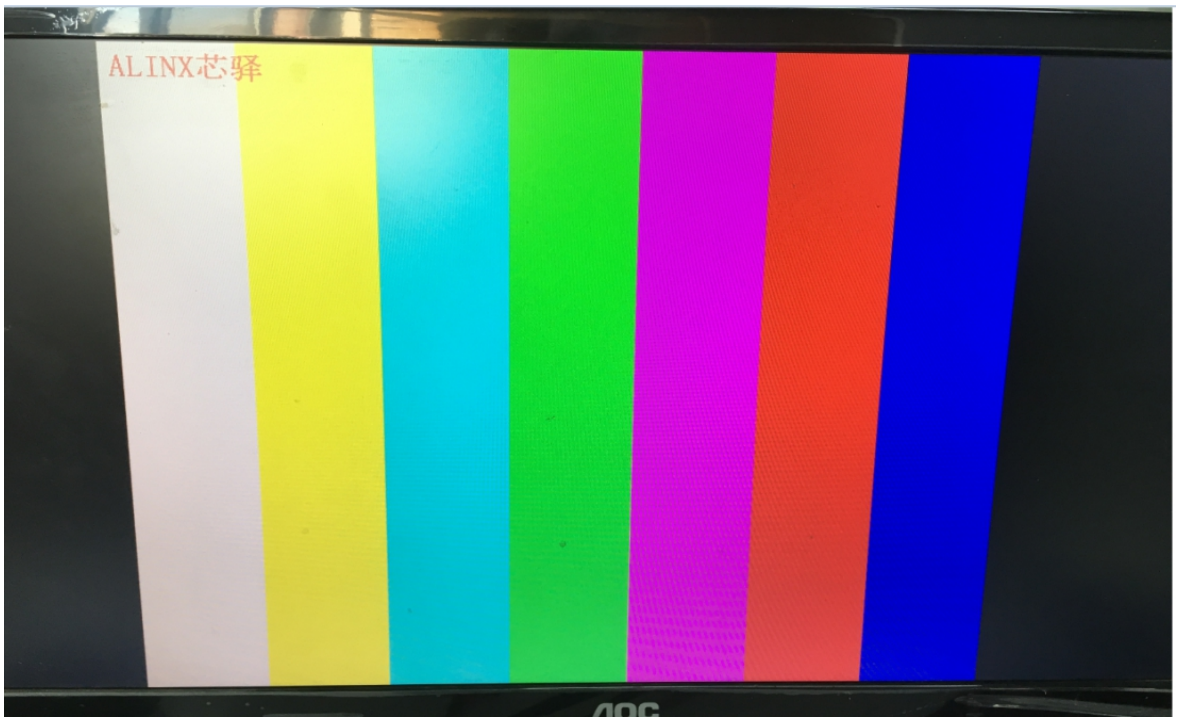
osd_display module port

Part 15.3: Experimental Results

Connect the FPGA development board and the monitor. Please refer to the "HDMI Output Experiment" tutorial for the connection

method. Please note that the connectors of the FPGA development board should not be hot plugged. After downloading the experiment program, you can see that the display shows a color bar as the background character.

As an HDMI output device, the FPGA development board can only be displayed through the HDMI display device. Don't try to display through the HDMI interface of the notebook computer, because the notebook is also an output device.



The default character display position is at the coordinates (9,9). In addition, the user can modify the following judgment conditions of pos_y and pos_x to display the character at any position on the screen:

```
73  always@(posedge pclk)
74  begin
75      if(pos_y >= 12'd9 && pos_y <= 12'd9 + OSD_HEGIHT - 12'd1 && pos_x >= 12'd9 && pos_x  <= 12'd9 + OSD_WIDTH - 12'd1)
76          region_active <= 1'b1;
77      else
78          region_active <= 1'b0;
79  end
```

Part 16: 7-inch LCD Screen Display Experiment (AN970 Module)

The experimental Vivado project is "lcd 7_test.

Based on the HDMI output experiment, this chapter introduces the display of the 7-inch LCD screen.

Part 16.1: Hardware Introduction

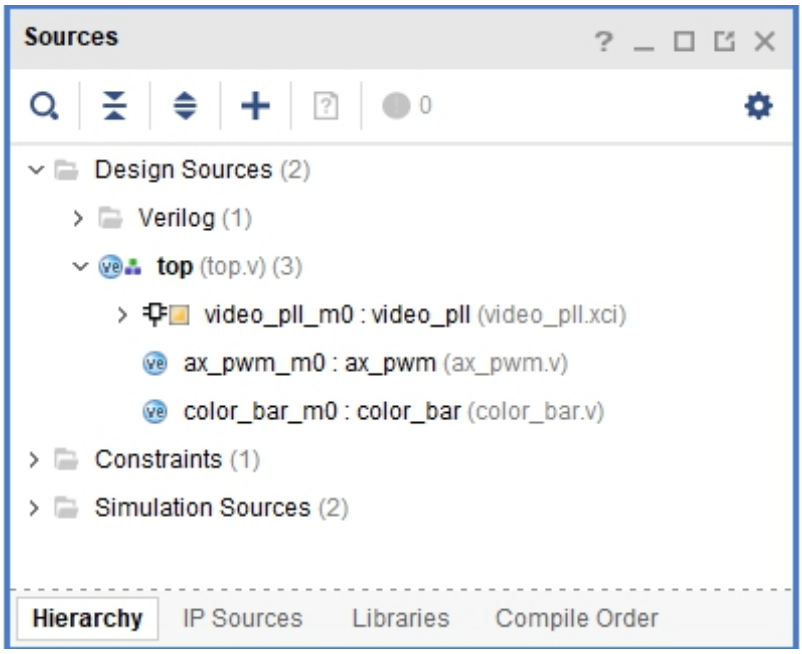
The AN970 LCD touch screen module consists of a TFT LCD screen, a capacitive touch screen and a driver board. Figure 16-1 is the AN970 module product photo as below:

Figure 16-1: AN970 Module Product Photo (Front side)

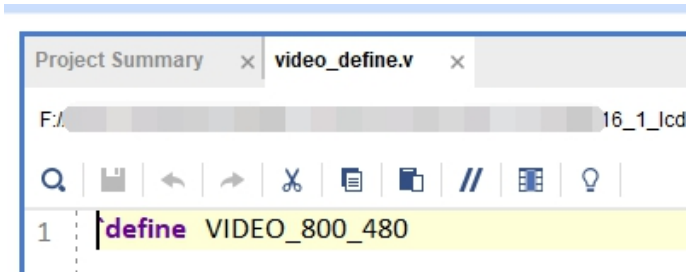
Part 16.2: Programming

The experiment in this chapter is actually very simple. The biggest difference from the HDMI display is that it does not require to configure i2c, and the output is just RGB. The following is the file

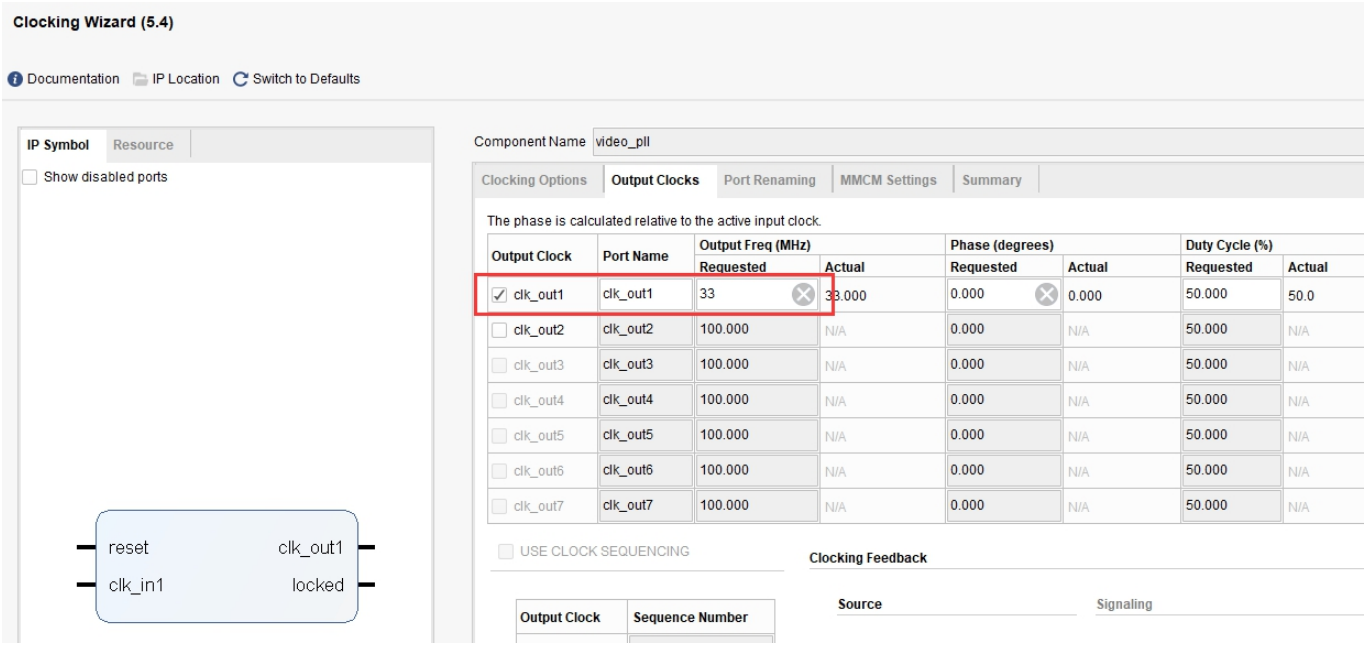
structure, with the rgb2dvi module removed.



At the same time, because the resolution of the LCD screen is 800x 480, the macro definition of video_define.v needs to be modified.



At the same time, modify the output clock frequency of the PLL to 33MHz, which is 800x480 pixel clock.

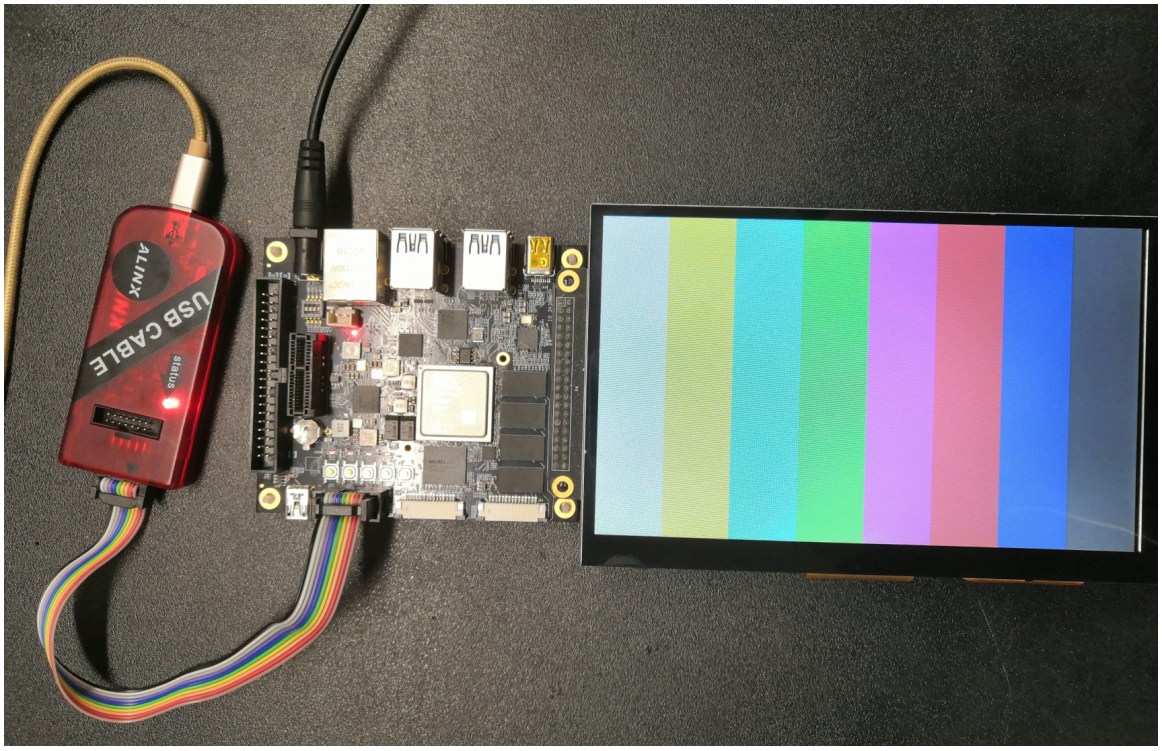


At the same time, ax_pwm is instantiated in top.v, which is used to adjust the brightness of the LCD screen and is set to 200Hz 30% duty cycle.

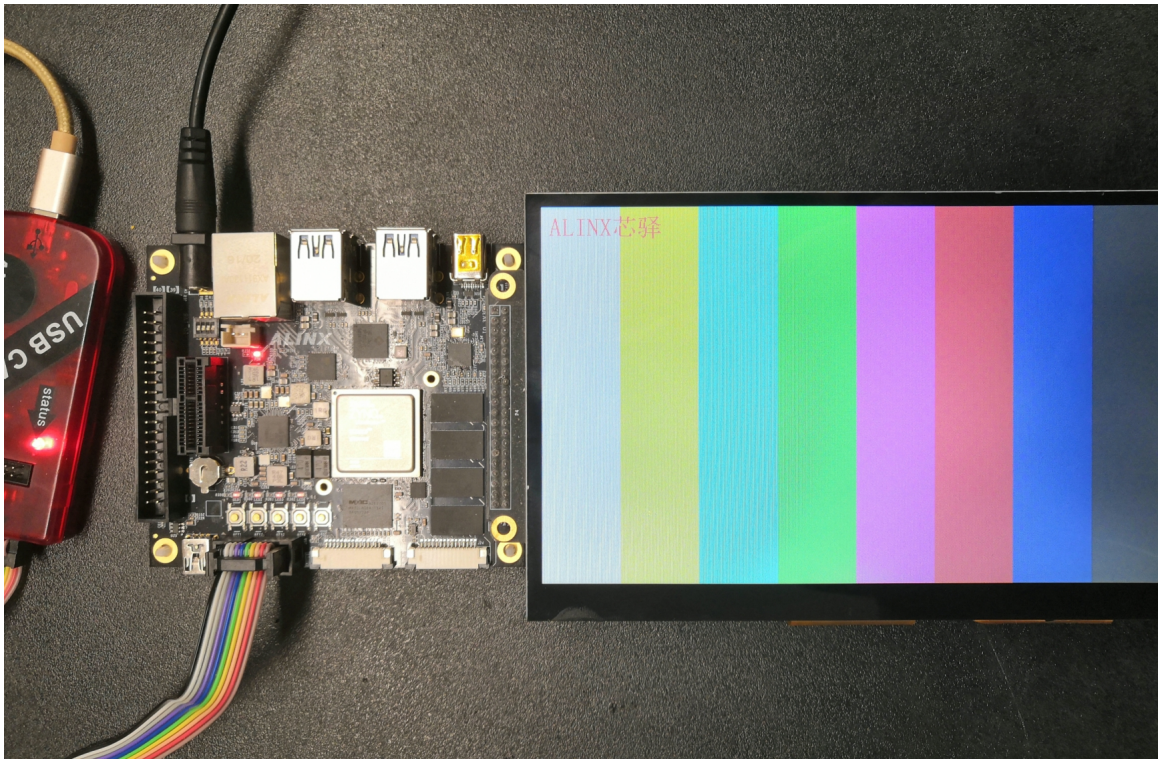
```
//200hz 30% duty
ax_pwm#(.N(24)) //pass new parameters
ax_pwm_m0(
    .clk                    (sys_clk            ),
    .rst                    (~rst_n             ),
    .period                 (24'd67             ),  //(2^24)*200Hz/50MHz
    .duty                   (24'd5033164        ),  //(2^24)* 30%
    .pwm_out                (lcd_pwm            )
);
```

Part 16.3: Experimental Results

Connect the LCD screen to the expansion port J15 of the FPGA FPGA development board, download the program, and you can see the color bar display.



At the same time, examples of character display and RTC display are prepared:



Character display