Z7-Lite Series

Development Tutorial

Xilinx ZYNQ7000 FPGA

Rev. 1.0

Version History

Version	time	describe
Rev. 1.0	2020.7.28	First version released

Copyright Notice:

Disclaimer:

Microphase Technology does not guarantee the source code, software, documents, etc. provided in the product.

any warranties of any kind, whether express or implied, including but not limited to fitness for a particular purpose.

There is no warranty for a particular purpose; all risks are borne by the user.

Table of contents

Chapter 1 Development Board Introduction and Unboxing Test

For more details, please refer to the test tutorial in the 01_Development Tutorial and Unboxing Test folder.

Chapter 2 ZYNQ Introduction

The Zynq-7000 series chips are based on Xilinx's scalable processor platform architecture.

Processing Platform (EPP), which combines a dual-core ARM Cortex-A9 processor and FPGA programmable logic

The elements are integrated into a single chip, thus forming a PS (Processing System) plus a PL (Programmable

Logic) single-chip SoC solution.

Beginners who use Zynq often have a question: Zynq is an FPGA with an embedded ARM core.

Is it an ARM embedded in FPGA logic? From its boot up, it is the ARM that dominates the entire system.

This is completely different from the original FPGA startup to boot the soft core MicroBlaze startup.

Most people think that Zynq is ARM embedded in FPGA logic; compared to Intel FPGA's Cyclone/Arria

The booting method of SoC (with integrated ARM) is slightly different from that of Zynq. The FPGA part in Intel SoC can be

It can be configured by itself or through ARM, so it is hard to say who is the master and who is the slave.

From the perspective of designers, they are more concerned about the ease of use of the chip and the flexibility of the chip to meet various application requirements.

From the user's perspective, they are more concerned about how to realize their own needs. As for who is the master and who is the follower, it is up to each person to decide.

If we must distinguish who is the master and who is the follower, I personally think it should be divided based on system application.

2.1 ZYNQ Structure Introduction

Zynq consists of two functional blocks, the PS part and the PL part. To put it simply, it is the ARM (PS) and

FPGA (PL) part. PS integrates two ARM Cortex-A9 processors, AMBA interconnect, internal

Memory, external memory interface and peripherals. These peripherals mainly include USB bus interface, Ethernet interface

port, SD/SDIO interface, I2C bus interface, CAN bus interface, UART interface, GPIO, etc.

The FPGA part of the 7 series is different. Different models have different resources, such as logic units.

Internal storage capacity, multiplier, high-speed interface, etc.

The following is the block diagram of Zynq.

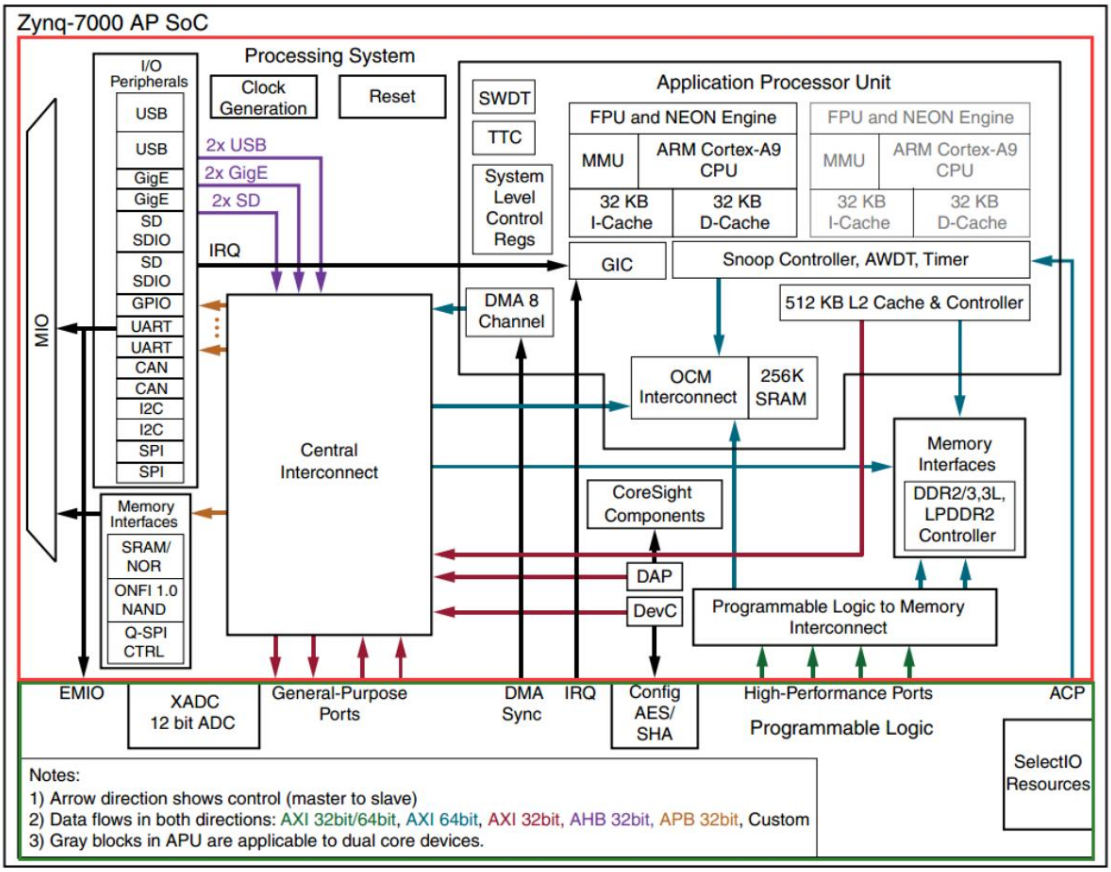


Figure 2-1

The red box on the block diagram is the PS part, that is, the ARM part

The blue box below the block diagram is the PL part, that is, the FPGA part

Here we need to introduce a few concepts, which will be used later.

MIO (Multiplexing IO, PS multiplexable IO, because the configuration of this IO pin is within a certain range flexibility);

EMIO (Extensible MIO, an extension of MIO, extends the IO port of PL to PS, directly used);

GIC (General Interrupt Controller);

IRQ (Interrupt Request);

OCM (On Chip Memory)

DMA (Direct Memory Access)

AXI (Advanced eXtensible Interface) is a bus protocol that interconnects the PS and PL of Zynq.
Using the AXI bus

2.2 PS-PL Interconnection: **AXI** Bus

The AXI bus is a general bus that connects the PS (programmable system) and the PL (programmable logic).
The final form is equivalent to the PL block being mounted on the AXI bus as an IP core, and then
PS call. Xilinx has supported the AXI bus since the 6 series FPGAs, and currently uses AXI4
Version.

ZYNQ has three **AXI** buses:

AXI4: (For high-performance memory-mapped requirements.) Mainly for high-performance
It is an address-mapped interface that allows a maximum of 256 rounds of data burst transmission.
lose;

AXI4-LiteÿÿFor simple, low-throughput memory-mapped communication ÿÿ
A lightweight address-mapped single-transfer interface that occupies very few logic cells.

AXI4-Stream: (For high-speed streaming data.) For high-speed streaming data transmission; removed
address items, allowing unlimited data burst size.

AXI has three types of interfaces:

AXI-GP interface (4): It is a general-purpose AXI interface, including two 32-bit master interfaces and two
32-bit slave device interface, which can be used to access on-chip peripherals in the PS.

AXI-HP interface (4): It is a high-performance/bandwidth standard interface, and the PL module is connected as a master device
(As can be seen from the arrow in the figure below). Mainly used for PL to access the memory on the PS (DDR and On-Chip
RAM

AXI-ACP interface (1): It is an interface defined in the ARM multi-core architecture.

The speed controller consistency port is used to manage non-cached AXI peripherals such as DMA. The PS end is the slave interface.

mouth.

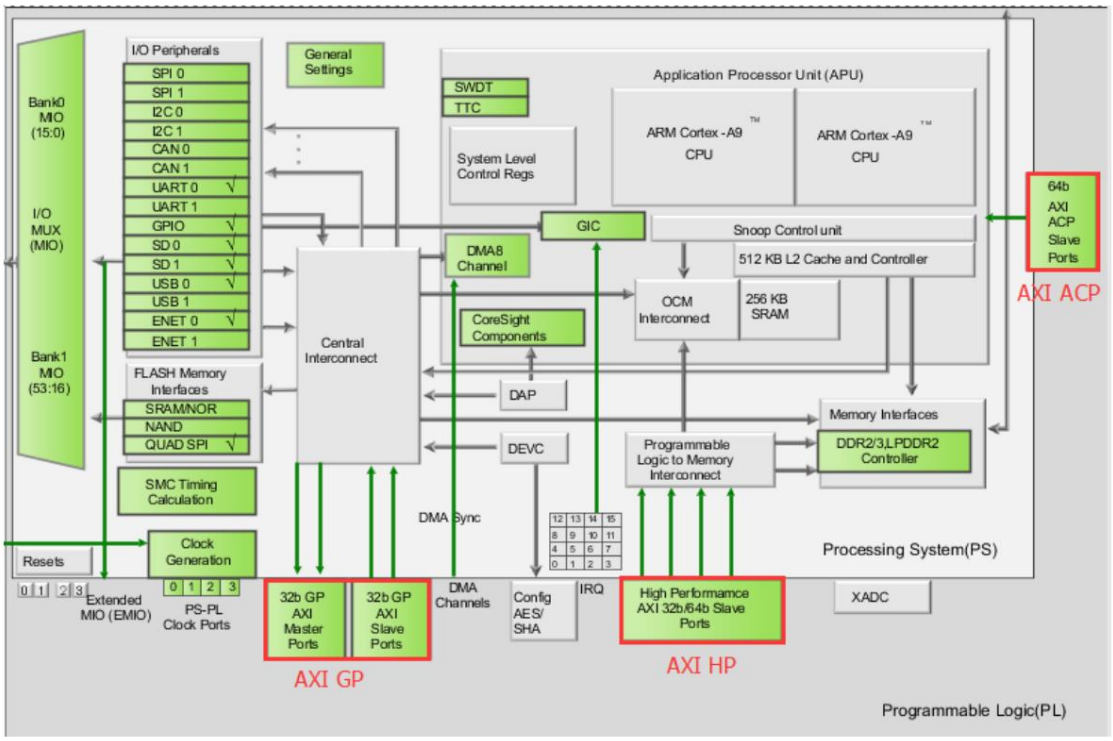


Figure 2- 2

The throughput comparison of the three is as follows (the figure below divides **AXI GP** into master and slave):

Table 22-2: **Theoretical Bandwidth of PS-PL and PS Memory Interfaces**

Interface	Type	Bus Width (bits)	IF Clock (MHz)	Read Bandwidth (MB/s)	Write Bandwidth (MB/s)	R+W Bandwidth (MB/s)	Number of Interfaces	Total Bandwidth (MB/s)
General Purpose AXI	PS Slave	32	150	600	600	1,200	2	2,400
General Purpose AXI	PS Master	32	150	600	600	1,200	2	2,400
High Performance (AFI) AXI_HP	PS Slave	64	150	1,200	1,200	2,400	4	9,600
AXI _ACP	PS Slave	64	150	1,200	1,200	2,400	1	2,400

Figure 2- 3

2.3 ZYNQ Development Process

ZYNQ integrates CPU and FPGA together, so ZYNQ developers need to have the following skills:

1. ARM OS Applications
2. Device Drivers
3. FPGA hardware logic design

That is to say, as a Zynq developer, you need to understand the Linux operating system, system architecture, FPGA and The hardware design platform between ARM systems, that is, the hardware and software collaborative design, can complete the above work A person can also be a team.

Tools needed during development:

1. **Xilinx Vivado: FPGA** design and development, pin and timing constraints, compilation and simulation True, implement RTL to bitstream design.
2. **Xilinx SDK:** Based on the Vivado hardware device, configure the tool and library path, compiler selection JTAG and Flash setup, debugger connection, bare board support package, support for all supported Xilinx IP hard cores The SDK supports co-debugging of IP hard core and processor software.
Develop and debug ARM and FPGA systems using C++ language.
3. Cross-compilation tool function: a comprehensive development environment consisting of compiler, connector, and parser. The cross-compilation tool of Xilinx ZYNQ platform is based on GNU tools.
The compilation and debugging of the tool are completed in the Ubuntu operating system environment of the virtual machine.
4. Software and hardware debugging tools: ChipScope Pro provided by FPGA-Xilinx Function: Online observation The signal operating system software debugging tools inside PFGA include the program debugging tool GDB under UNIX and the remote Debugging tool GDBserver.

The development process must also be a process of hard first and soft later:

1. Create a project in Vivado and add an embedded source file
2. Hardware system construction, including adding PS system and peripherals, adding PL interface, IP
3. Vivado creates the top-level HDL and writes the necessary logic program.
4. Add constraints, synthesize and compile to generate .bit files
5. Export hardware to SDK

6. Create FSBL file in SDK

7. Generate uboot.elf, bootloader image in the virtual machine

8. Generate boot.bin from FSBL, System.bit, uboot

9. Generate Ubuntu kernel image file Zimage and Unbuntu file system ramdisk,

Sometimes it is also necessary to write interface drivers.

10. Put Boot, device tree, kernel image file, and root file system into the SD card to start the board

The above is the general development process of Zynq. In actual use, some steps may not be used, such as

For pure PL development, PS and Linux are not involved. Pure ARM bare metal running does not involve PL and Linux.

For the Linux part, users can develop according to their actual situation.

Chapter 3 LED Experiment and Development Environment Experience

3.1 Experimental Introduction

In this chapter, we will introduce Vivado software development Verilog through a simple LED flashing experiment.

The basic process of the program, including device selection, code writing, synthesis and compilation, pin allocation, program

download.

3.2 Experimental Principle

There are 3 LED lights available for users on the Z7-Lite development board, one of which is the LED on the PS end.

There are two LED lights on the PL side, as shown below:

Figure 5- 1

As can be seen from the schematic diagram, the LED light on the bottom board is low level effective, and when the output to the LED light

When the level is low, the LED light is on, and when the level is high, the LED light is off;

3.3 Create a project

1) In the start menu, select Xilinx Design Tools-->Vivado2018.3-->Vivado 2018.3 or double-click

Click the Vivado 2018.3 icon on the desktop to open the Vivado development environment.

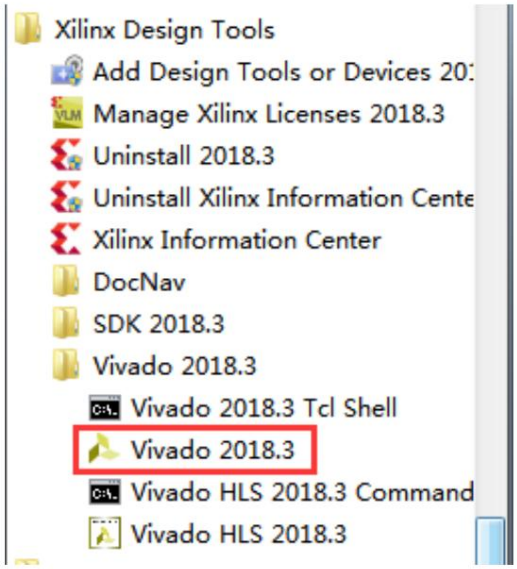


Figure 5- 2

2) Click Create New Project in the development environment

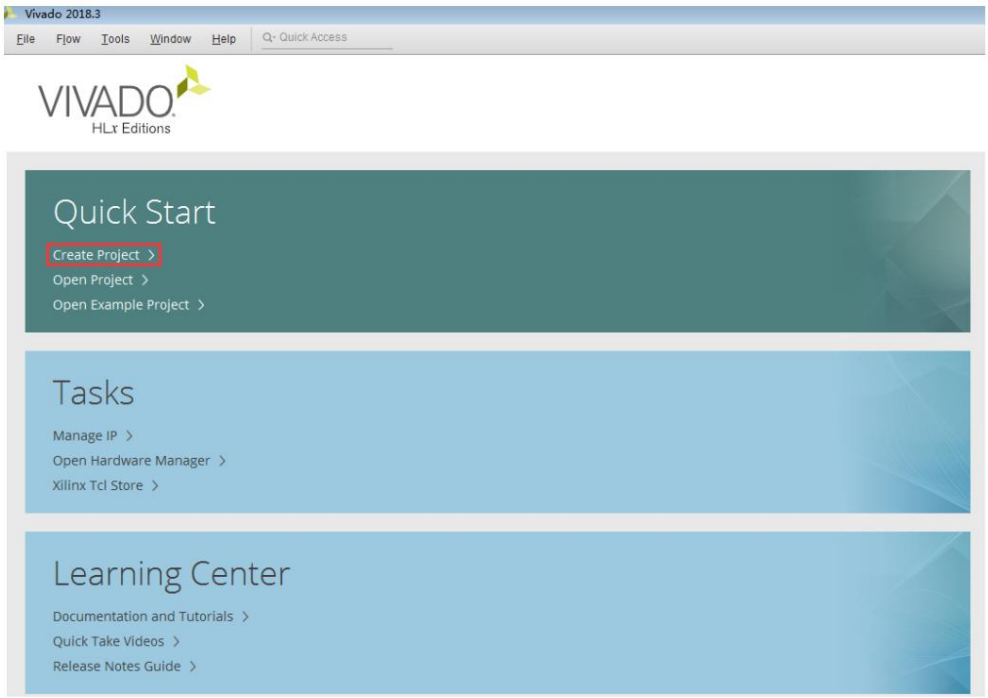


Figure 5- 3

3) Click Next

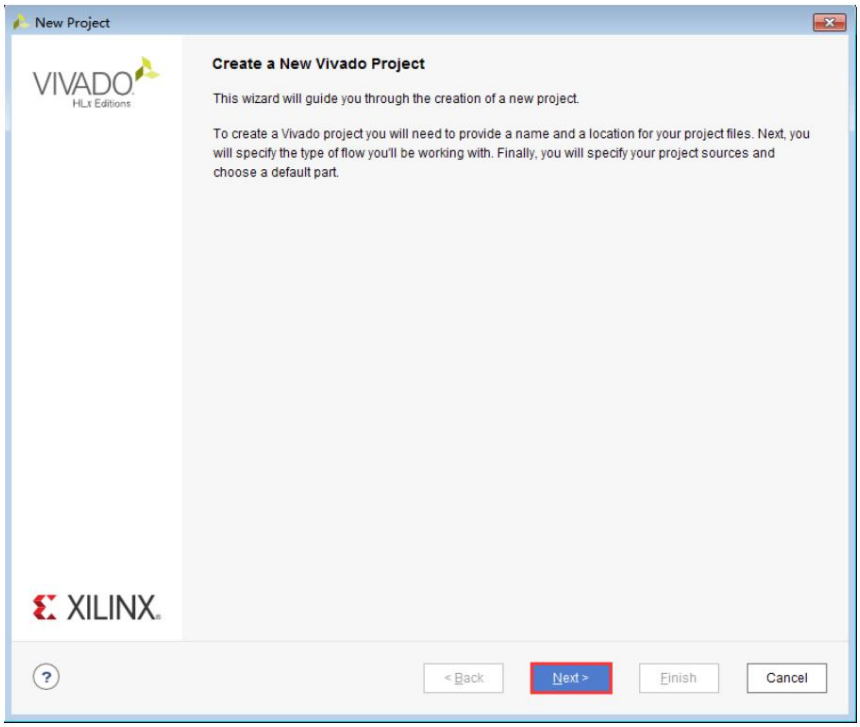


Figure 5- 4

4) Enter the project name, select the storage path, and click Next

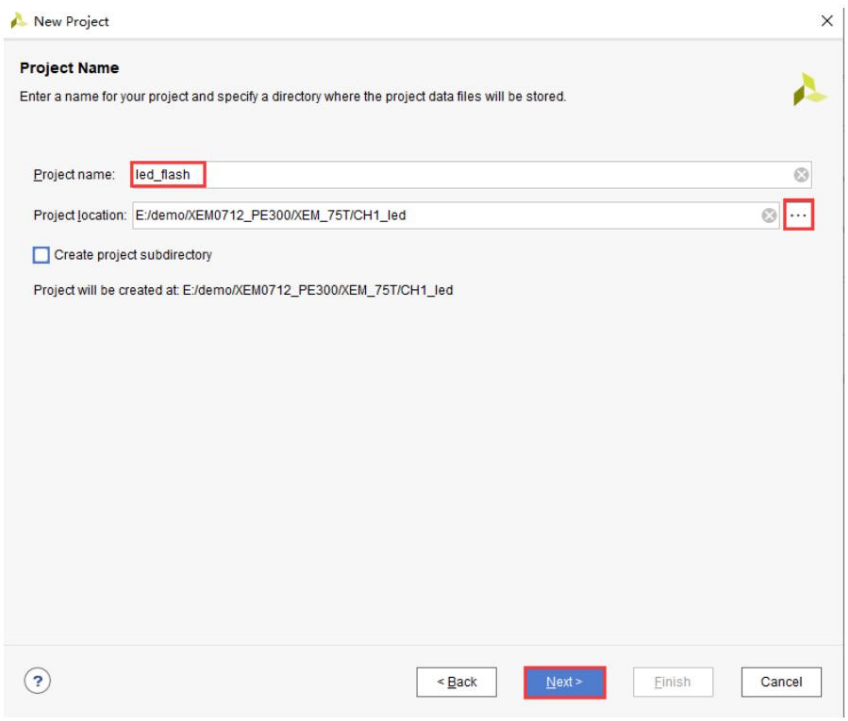


Figure 5- 5

5) Select RTL Project and click Next

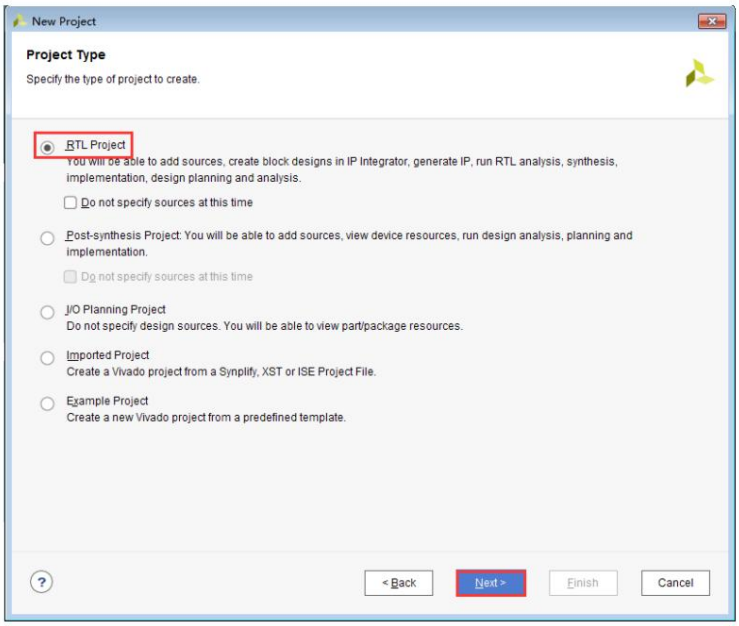


Figure 5-6

6) This step is to add the Source file. Don’t add it here yet. Click Next.

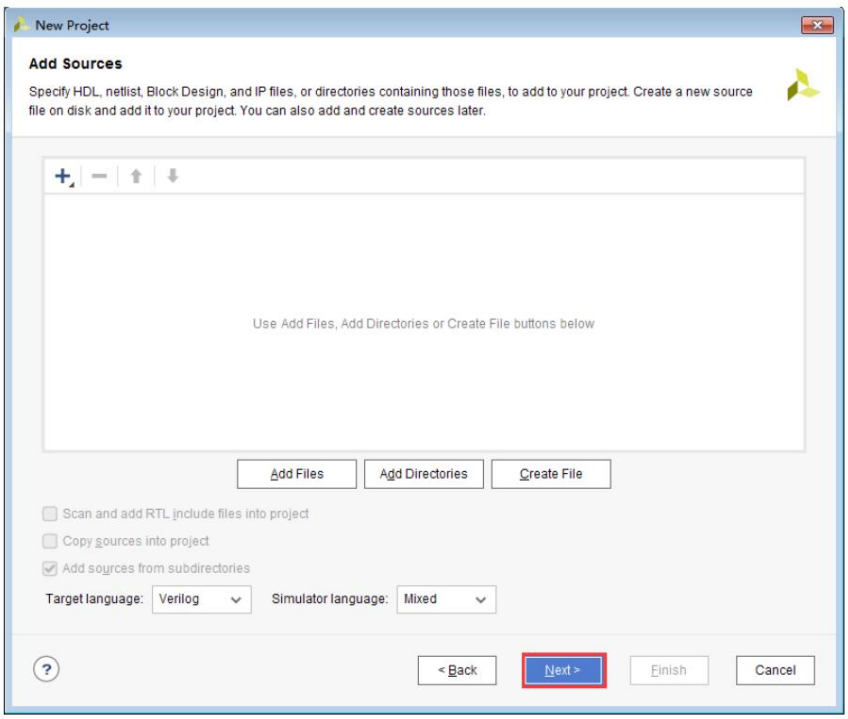


Figure 5- 7

7) Here is to add the constraint file. Again, do not add it for now. Click Next.

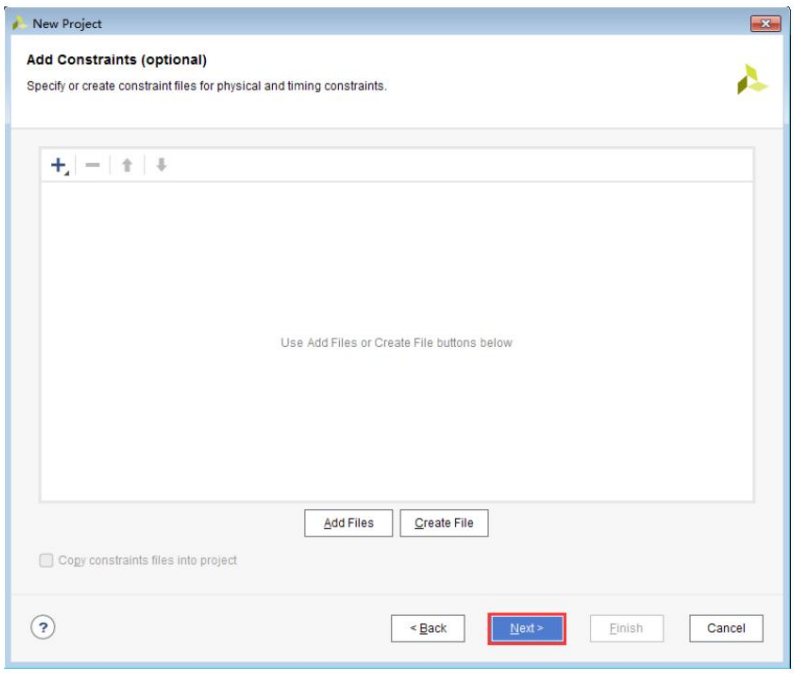


Figure 5- 8

8) Select the device. You can choose according to the chip model on the development board. Since there are many chip models,

Some screening can be done.

Users who purchase Z7-Lite7020 should select Zynq-7000 in the Family column and clg400 in the Package column.

The speed grade is 2, and then select the corresponding device in the table below.

Users who purchase Z7-Lite7010 should select Zynq-7000 in the Family column and clg400 in the Package column.

The speed grade is 1, and then select the corresponding device in the table below.

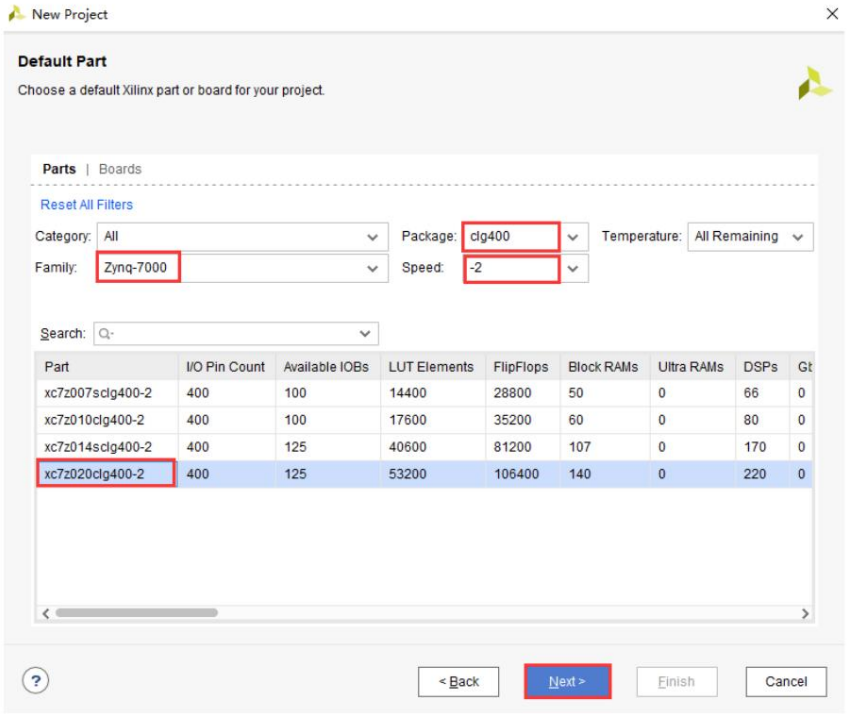


Figure 5- 9

Or users can directly search according to their FPGA chip model. Users who purchase Z7-Lite can select

xc7z020clg400-2 , users who purchased Z7-Lite selected Xc7z010clg400-1

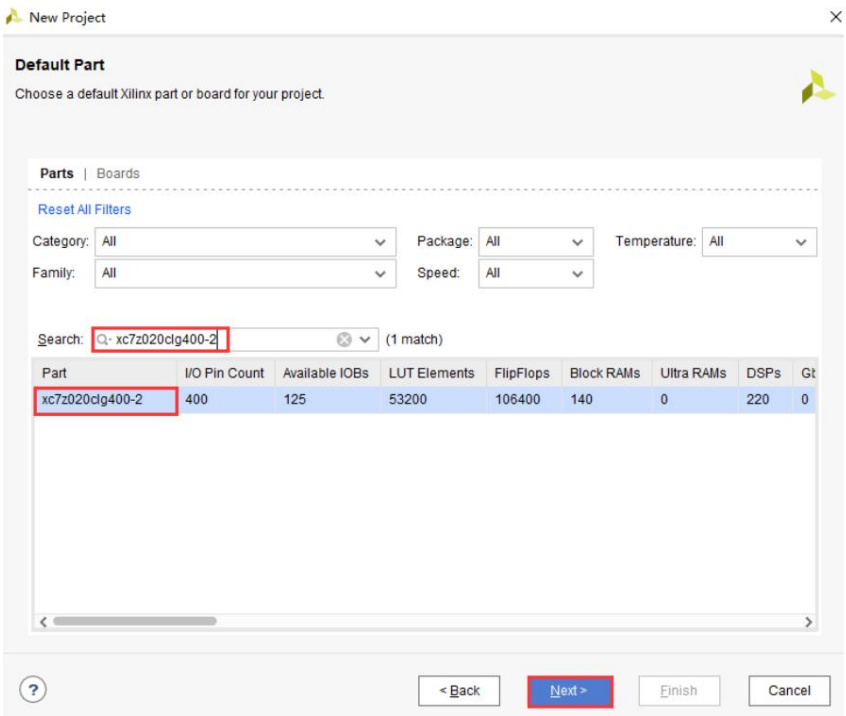


Figure 5- 10

9) Confirm again whether the model is selected correctly. If not, click Finish to complete the project creation.

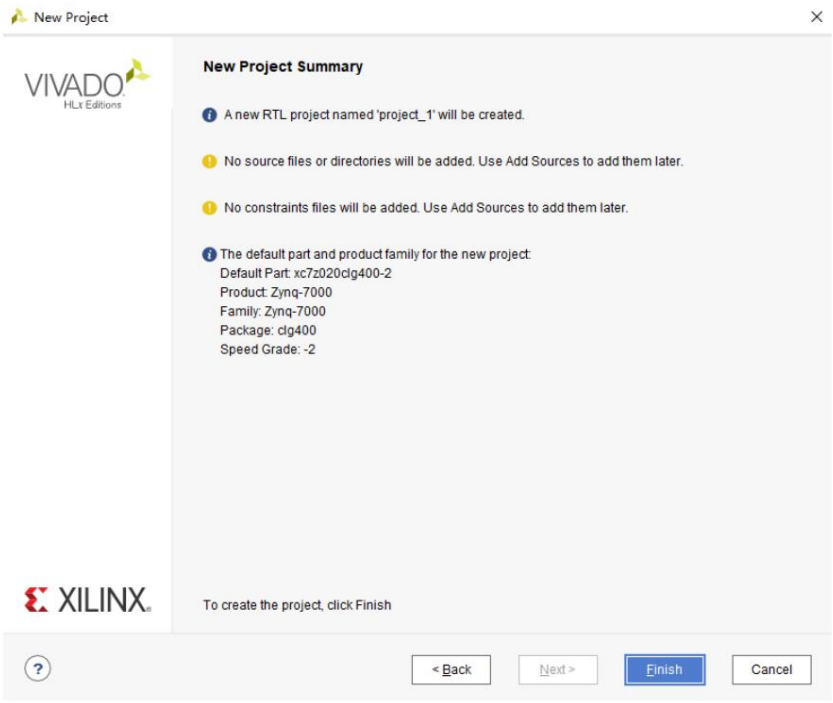


Figure 5- 11

10) Created project

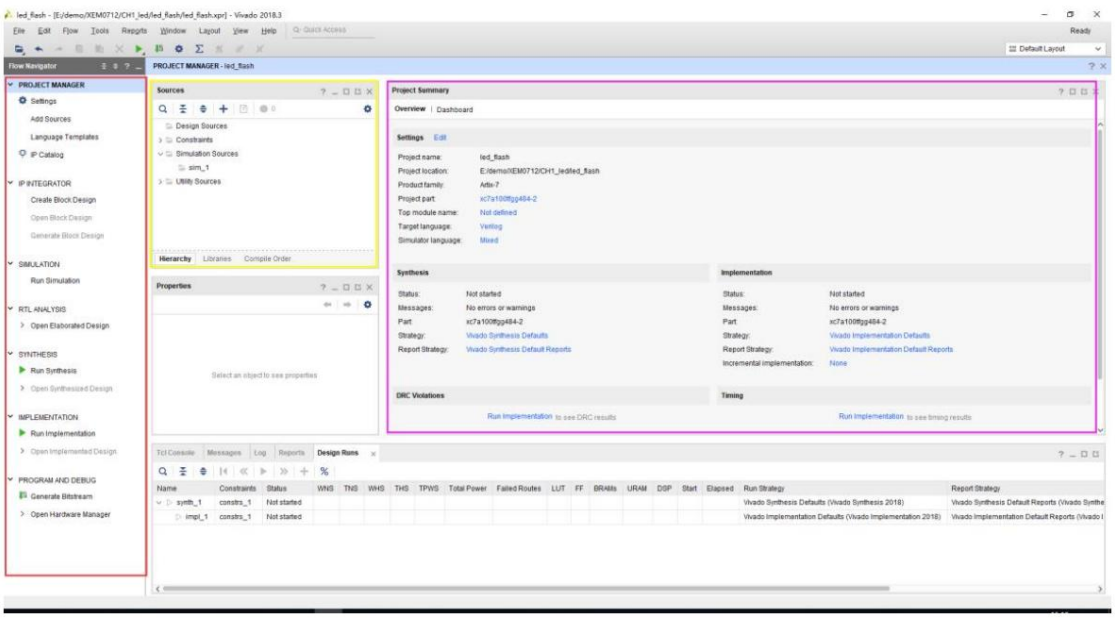


Figure 5- 12

The red box is the design process navigation window, which shows the various links of FPGA development from top to bottom.

The purple box is the project data file window, where you can view the project hierarchy and edit project file properties.

The main working window has different display contents according to different layouts. The blue box is the result display window, which displays

The execution results of each step. The black box can switch between different layouts.

3.4 Programming

1) Before programming, we need to create a .v program file.

Manager, click the Add Source icon

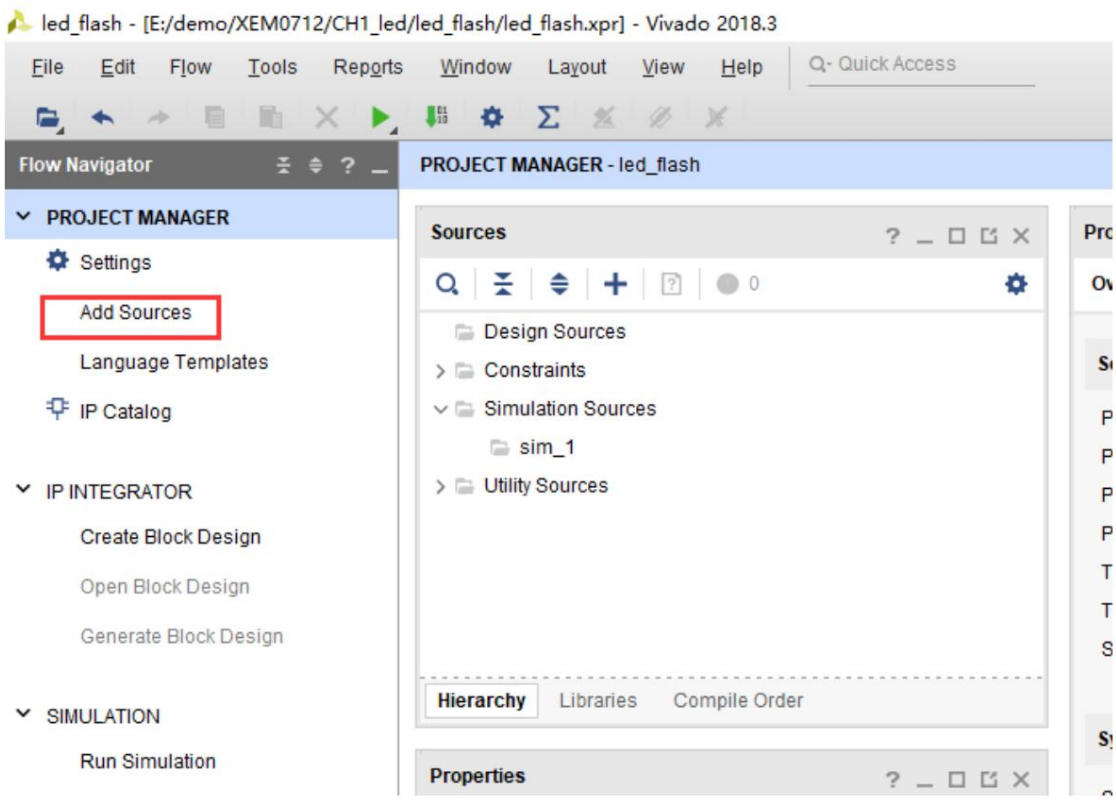


Figure 5- 13

2) Select Add or Create Design Source, then click Next

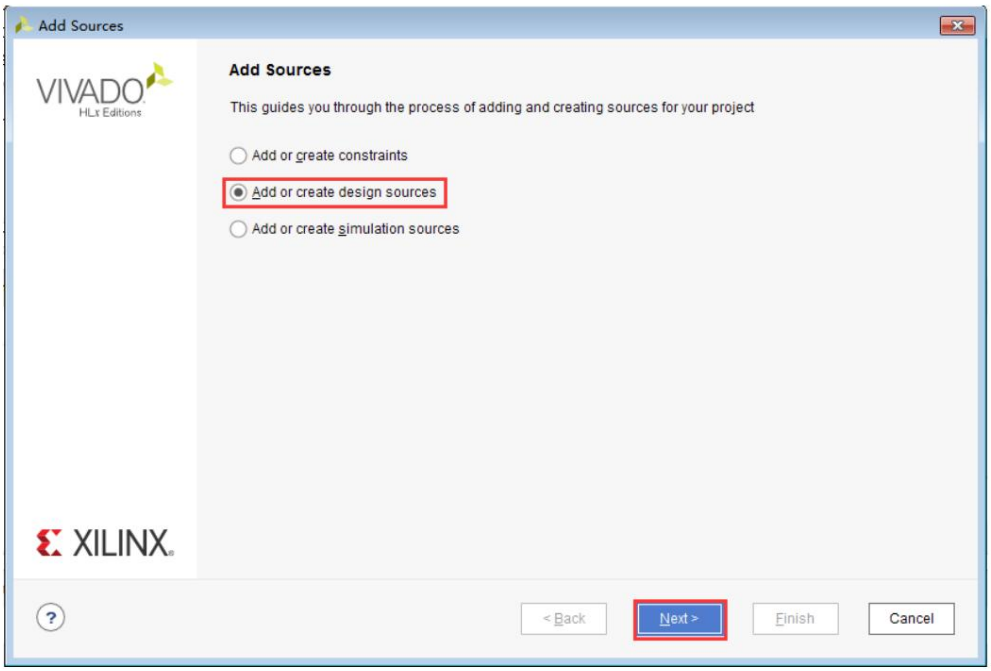


Figure 5- 14

3) Users can add existing programs. Here we have programs, so select Create File

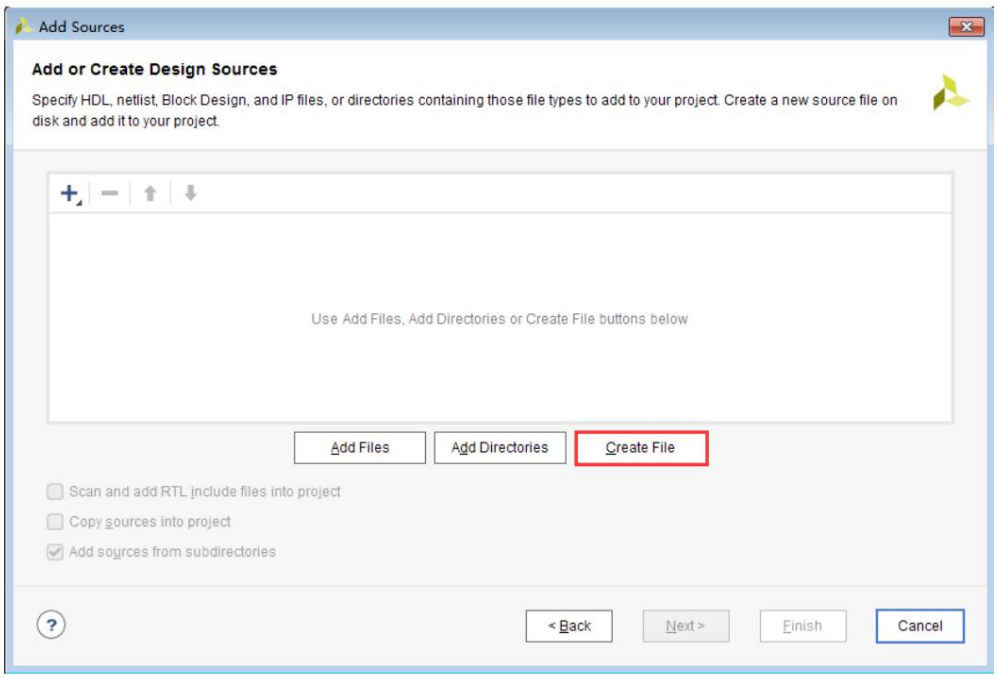


Figure 5- 15

A dialog box pops up, enter the file name led, and click OK

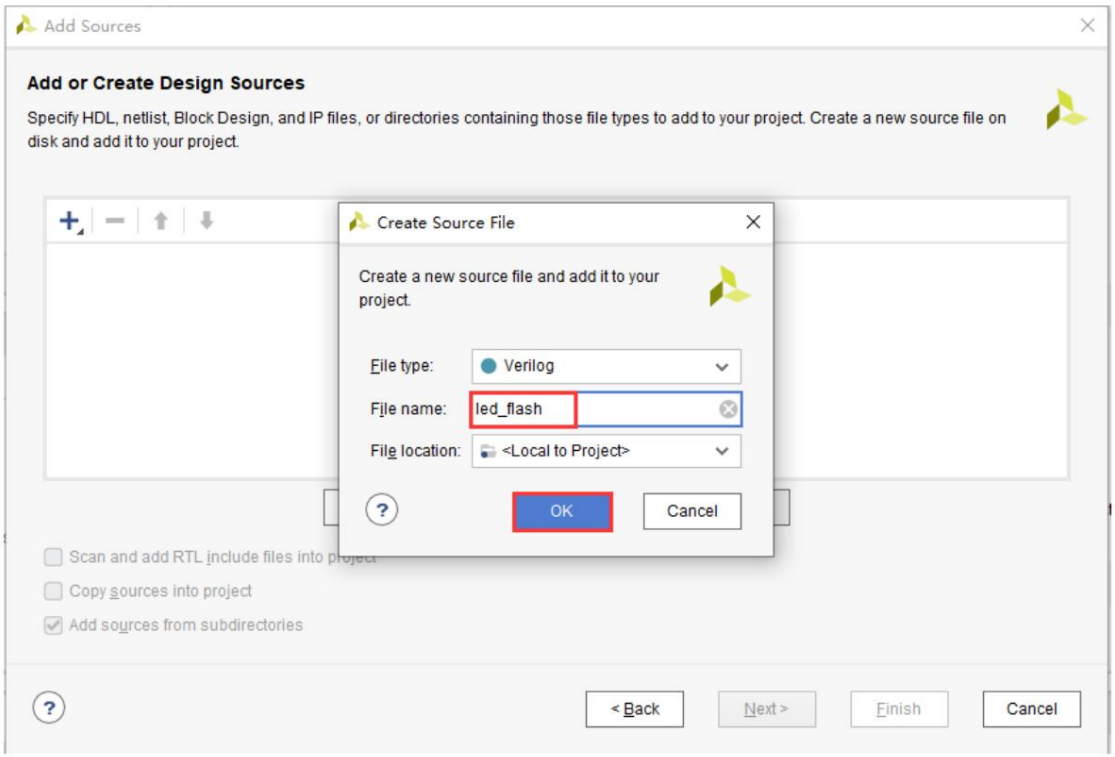


Figure 5- 16

4) Click Finish

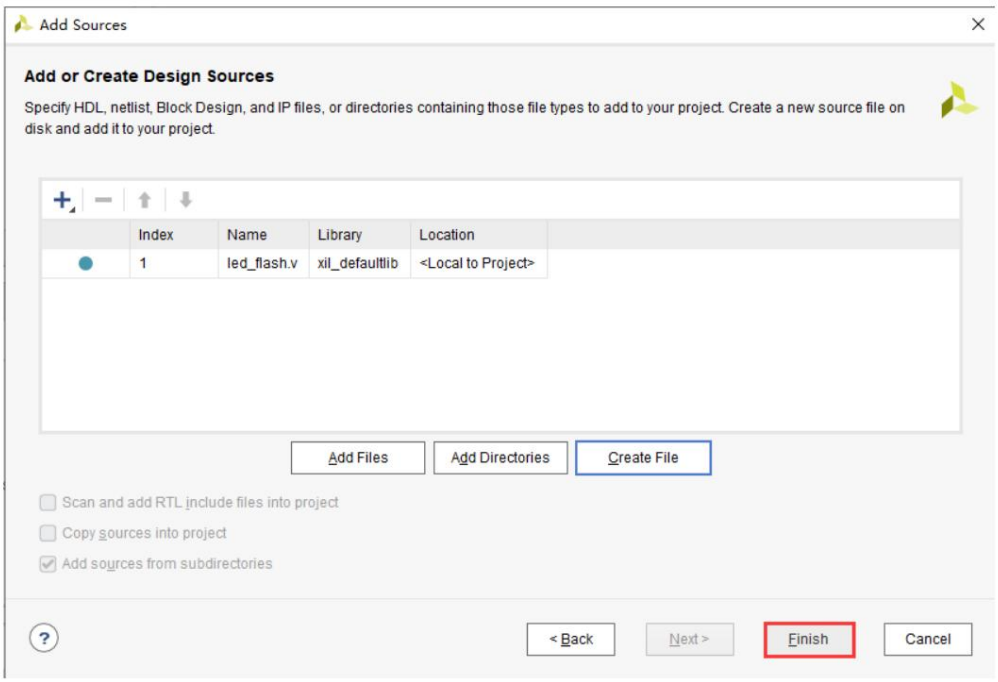


Figure 5- 17

5) The wizard will prompt you to define the I/O port. Do not define it here. You can write it in the program later.

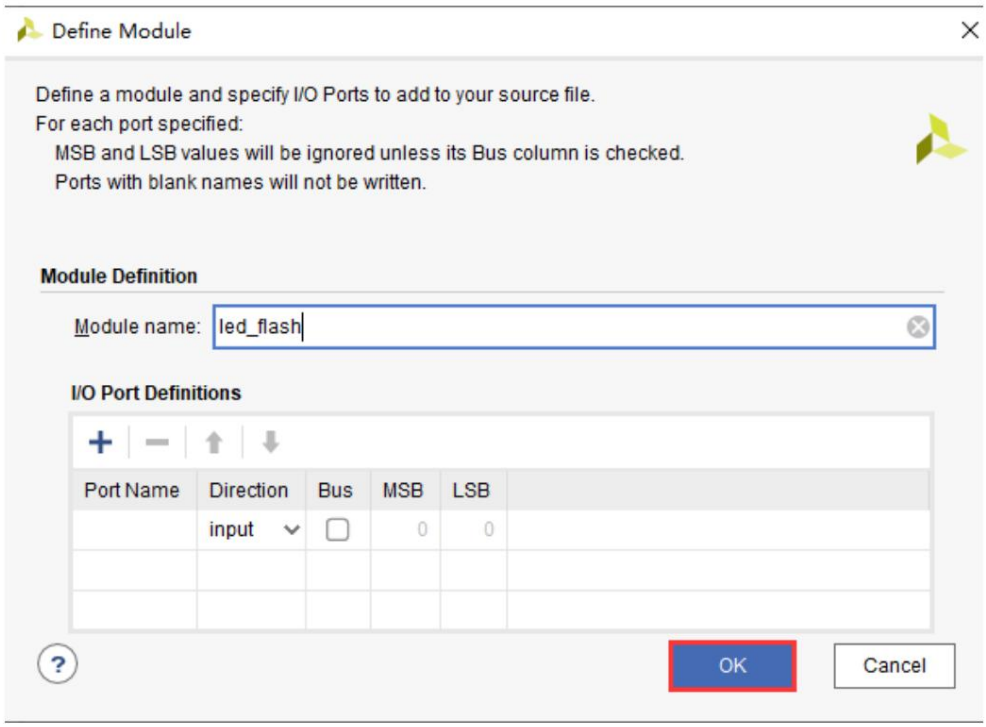


Figure 5- 18

Click Yes

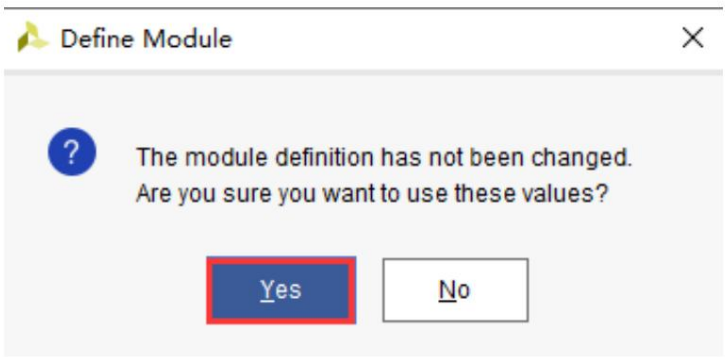


Figure 5- 19

6) Now under the Project Manager interface, under Design Source, there is the led_flash.v file, and

And automatically becomes the top module of the project

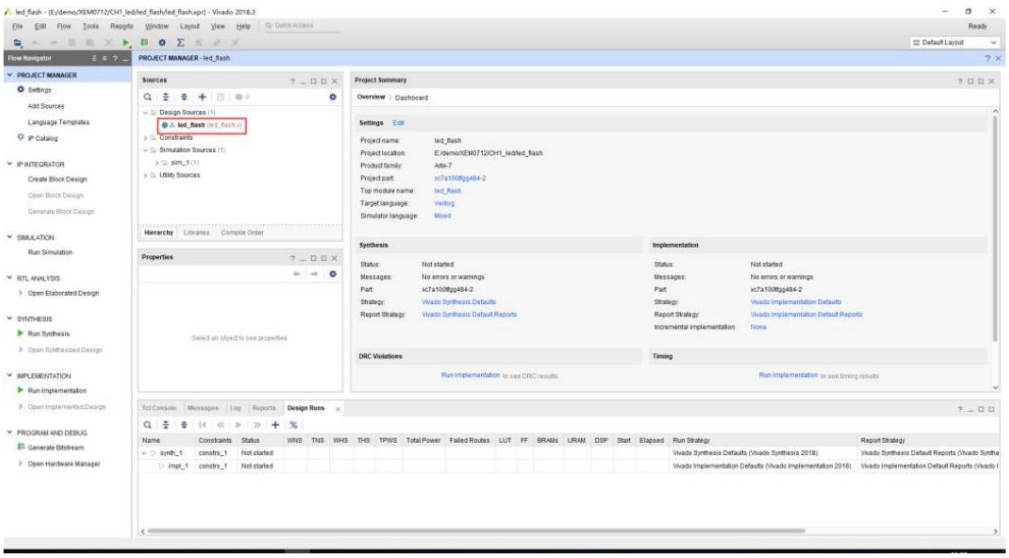


Figure 5- 20

7) The next step is to write the main part of this experiment and complete the part of the running light. The code is as follows:

The Z7-Lite series codes are as follows:

```
1 module led_flash(
2           input wire input                    clk         ,// System differential clock
3           wire                                rst_n             ,// system reset
4           output reg              [1:0]       led               // output led
5 );
6
7 //================================================
8 //internal singals
9 //================================================
10
11
12 reg          [27:0] cnt                  ;// 0.25s counter
13 wire                    add_cnt              ;
14 wire                      end_cnt            ;
15
16
17 //--------------------cnt--------------------
18 always @(posedge clk or negedge rst_n)begin
19          if(rst_n==1'b0)begin
20                  cnt <= 'd0;
21          end
22          else if(add_cnt)begin
```

```
23                  if(end_cnt)
24                      cnt <= 'd0;
25                  else
26                      cnt <= cnt + 1'b1;
27          end
28 end
29
30 assign add_cnt = 1; 31
assign end_cnt = add_cnt && cnt== 10_000_000 - 1;
32
33
34 //--------------------led--------------------
35 always @(posedge clk or negedge rst_n)begin
36          if(rst_n==1'b0)begin
37                  led <= 2'b10;// reset state
38          end
39          else if(end_cnt)begin
40                  led <= {led[0], led[1]};
41          end
42          else begin
43                  led <= led; //keep the state
44          end
45 end
46
47
48 endmodule
```

Here are some points to note in the code: Line 12 declares a counter to count the time.

The counter is used to control other logic. Lines 18 to 31 are the implementation codes of the counter.

The counter remains at 0. After the count is reset, the counter increases by one at each clock cycle. When the count reaches the maximum value,

Clear the counter and restart counting.

Lines 35 to 45 are used to control the LED light flow. When reset, the lowest bit of the LED is valid.

When the counter reaches the maximum value, the LED completes the shift, which is done by bit splicing.

The shift operation of LED. You can see that the code is very simple.

8) After writing, press Ctrl+S to save the code.

3.5 Add pin constraints

Unlike the previous ISE software, the constraint file format used by Vivado is the xdc file.

It mainly defines pin constraints, clock constraints, and other timing constraints. Here we need to define the led.v program

To assign the input and output ports in the FPGA to the pins, you need to add the pin assignment .xdc file

To the project

1) We need to create a .xdc constraint file first. Under Project Manager on the left sidebar, click

Add Source Icon

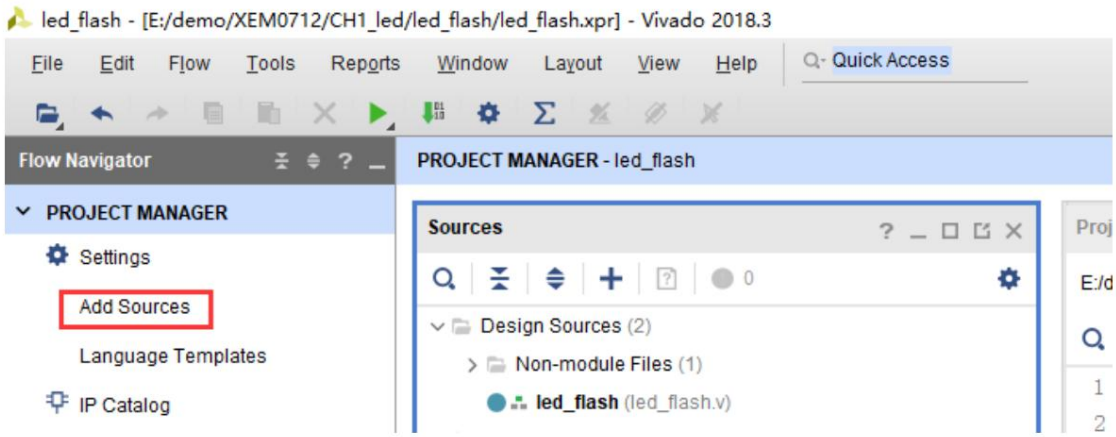


Figure 5- 21

2) Select Add or create constraints and click Next

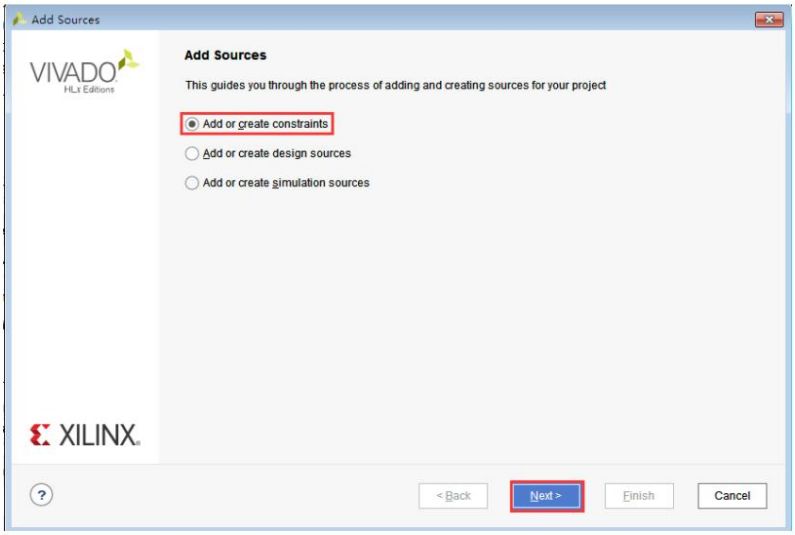


Figure 5- 22

3) Click the Create File button, enter the top_pin file name in the file name, and click OK

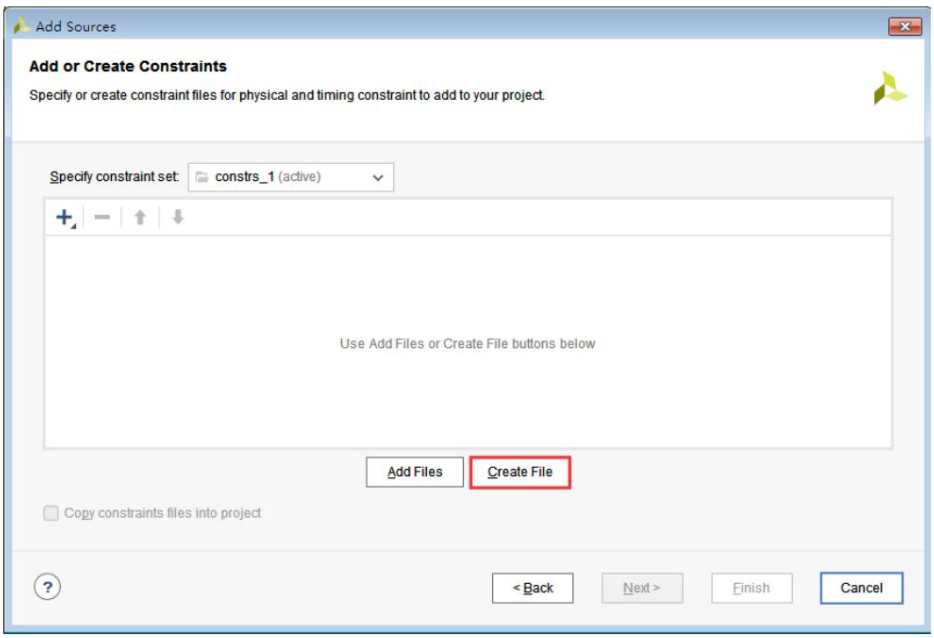


Figure 5- 23

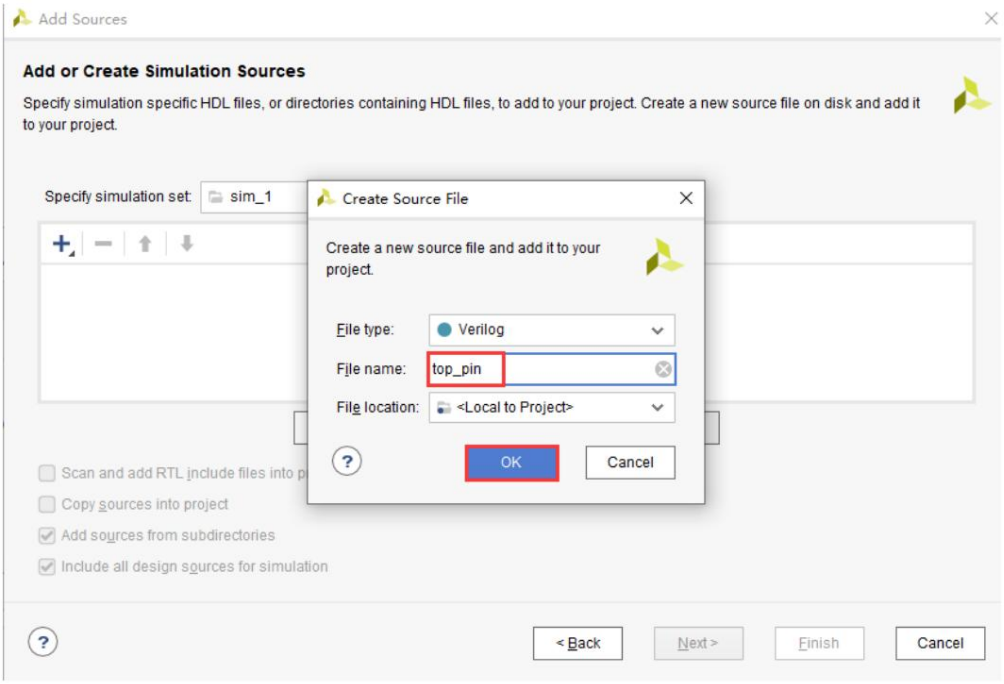


Figure 5- 24

4) Click the Finish button to complete the creation of the constraint file.

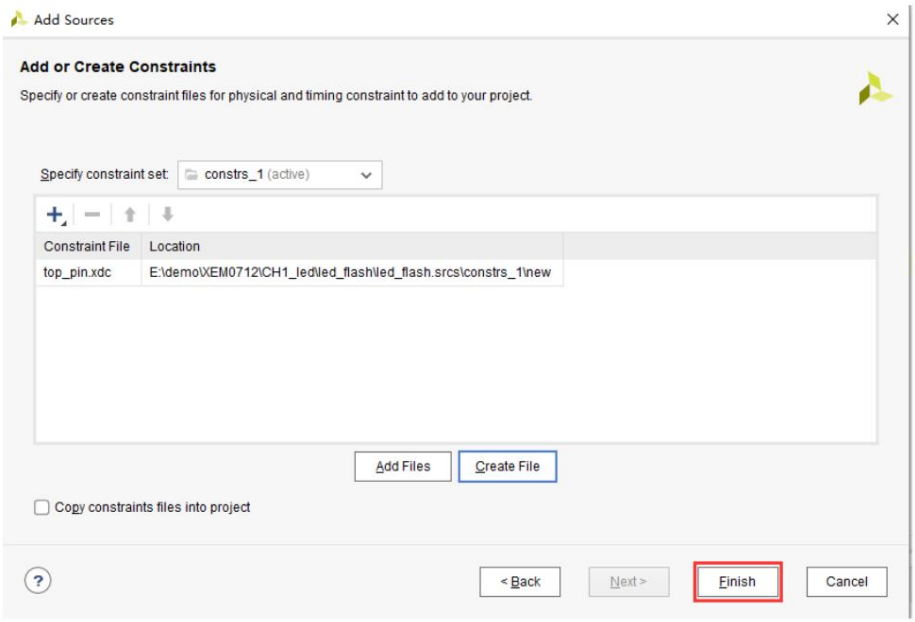


Figure 5- 25

5) After it is built, there is a top_pin.xdc file under the constraint file group.

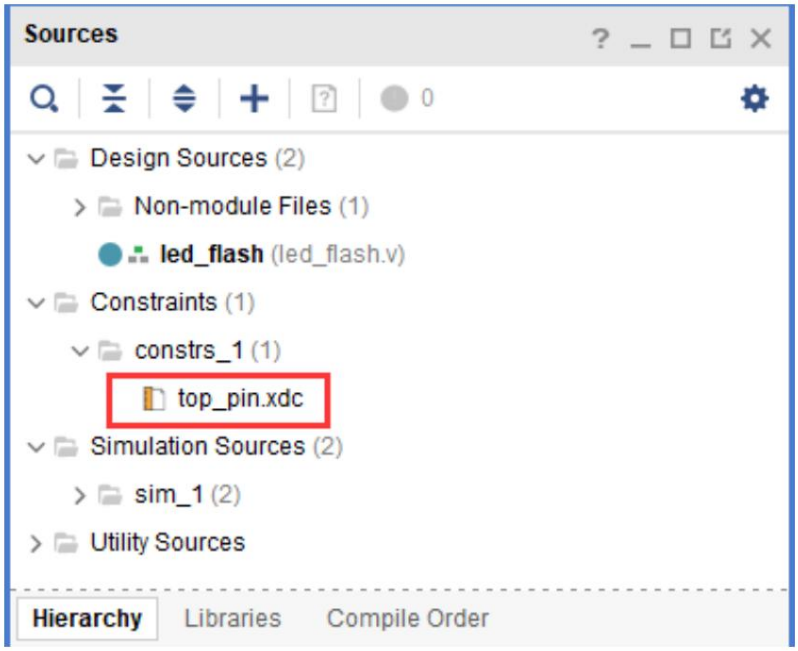


Figure 5- 26

6) Open the top_pin.xdc file and add the following constraints. For how to write the constraint file, you can refer to Xilinx's

Related documents, the following introduces the most basic XDC writing syntax, ordinary IO ports only need to constrain pins

The pin constraints are as follows:

set_property PACKAGE_PIN "Pin Number" **[get_ports** "Port Name"]

The constraints of the level signal are as follows:

set_property IOSTANDARD "voltage" **[get_ports** "port name"]

Please pay attention to the capitalization of the text here. If the port name is an array, use { } to enclose it. The port name must be consistent with the source code.

The port name must be consistent with the name in the table, and the port name cannot be the same as the keyword. Take the constraint of led[0] as an example:

set_property PACKAGE_PIN G14 [get_ports {led[0]}] is to bind the pin of led[0]

set_property IOSTANDARD LVCMOS33 [get_ports {led[0]}] is to set led[0]

Level constraints.

The constraint files for the Z7-Lite series are as follows:

```
################ clock define####################
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports clk]

############### key define###################
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
###############LED define###################

set_property PACKAGE_PIN P15 [get_ports {led[0]}]
set_property PACKAGE_PIN U12 [get_ports {led[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports {led[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
```

7) After writing, click Save.

8) Click 1 Synthesis, 2 Implementation, 3 Generate .bit file in sequence, or just double-click 3, the effect is the same

of.

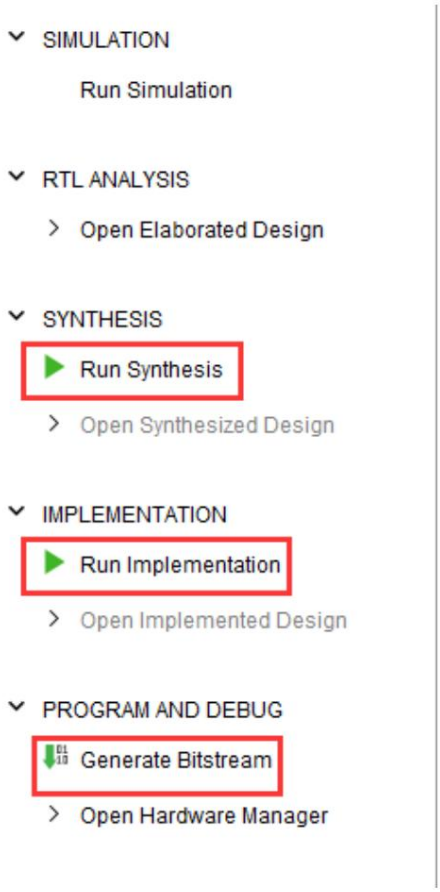


Figure 5- 27

After generating the .bit file, you can see relevant information in the Project summary, such as the entire project uses

Resource status, power consumption, timing and other information.

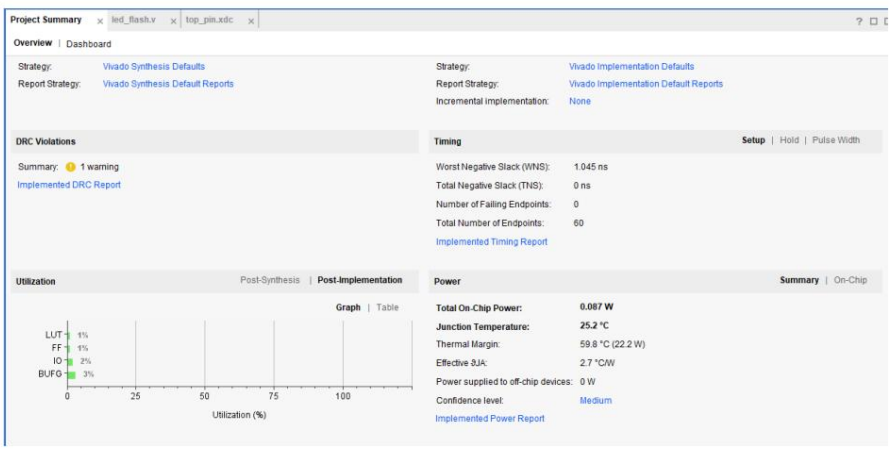


Figure 5- 28

3.6 Download Verification

1) After the .bit file is generated, it can be downloaded to the FPGA chip to see the operation results of the LED.

Connect the device to the JTAG port, power on the board, click Open Hardware Manager, and then

In the Manager window, click Open target, then click Auto Connect.

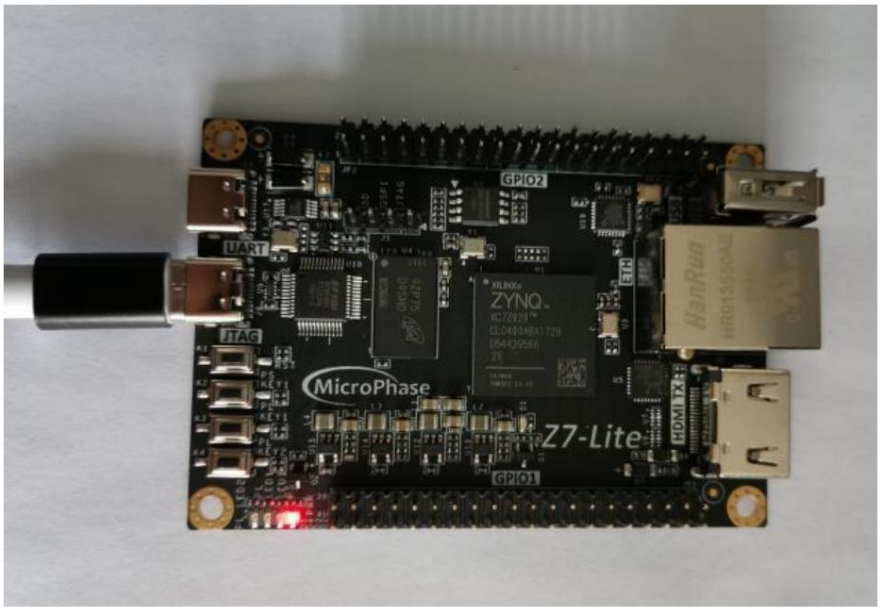


Figure 5-29 Z7-Lite connection method

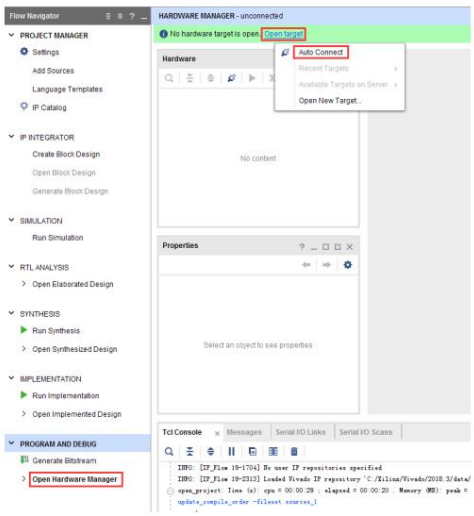


Figure 5- 30

2) After the connection is successful, the currently scanned device will be displayed in the Hardware window, as shown in the figure below

The device is xc7z020, which matches the device model we selected at the beginning.

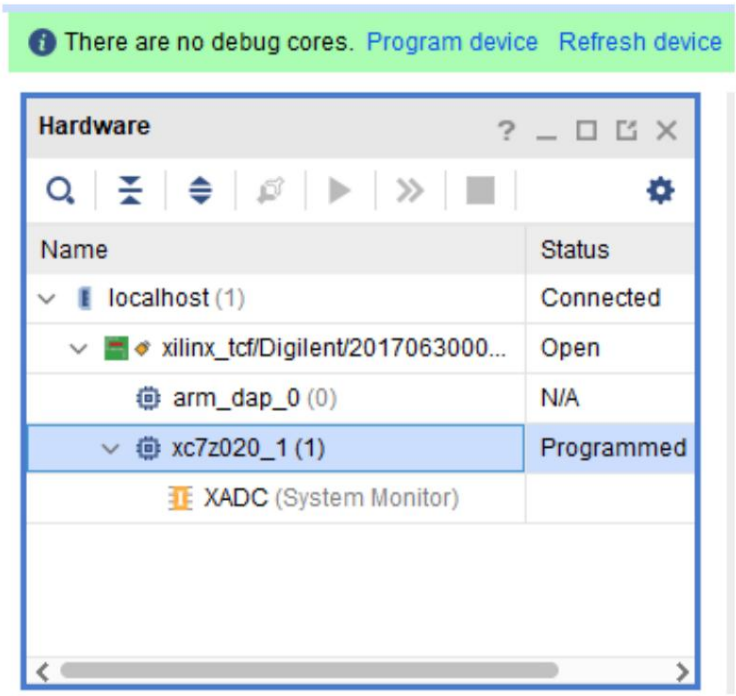


Figure 5- 31

3) Click Program device, download the program, a dialog box pops up, just click the Program button.

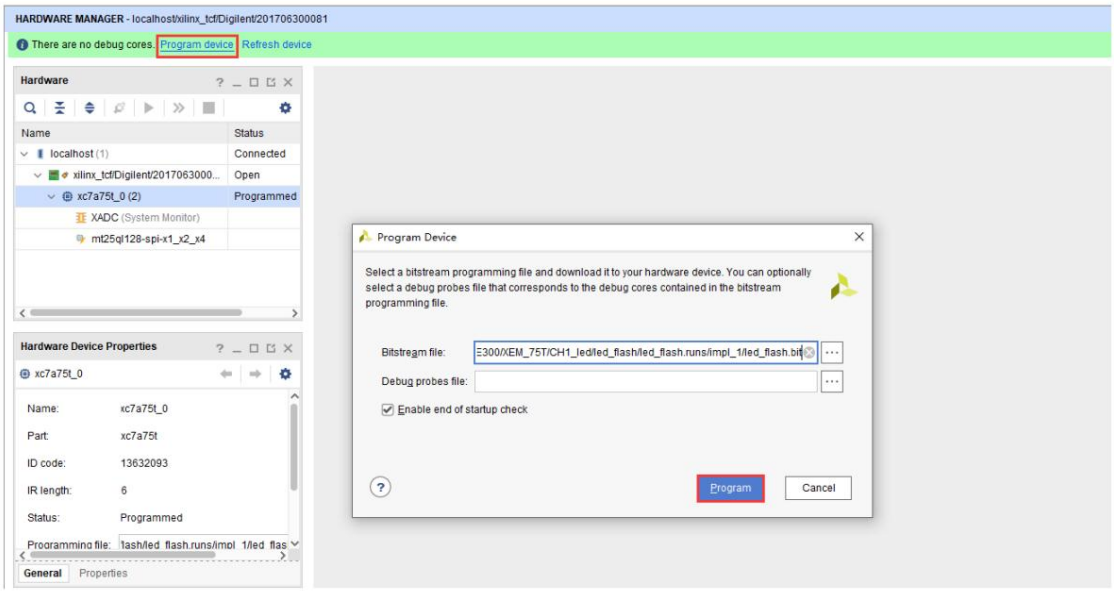


Figure 5- 32

4) After the download is complete, you can see that the two LED lights start to operate in a running manner.

3.7 Experimental Summary

This experiment used FPGA to complete a simple water lamp experiment for the first time.

After understanding how to develop in Vivado, there are still many things to pay attention to in the whole process.

During the experiment, after many times of practice, we were able to master the use of the software very skillfully.

Chapter 4 Verliog Grammar Basics

4.1 Module Declaration

In FPGA development, it is based on modules. Each synthesizable .v file is a module.

The module is declared by module——endmodule. Within these two keywords, the module functions are completed.

The following example uses a simple AND gate to demonstrate how to declare a module.

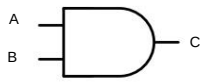


Figure 6- 1

```
1 module and_gate(
2           input wire              a       ,
3           input wire              b       ,
4           output wire             c
5 );
6
7 assign c = a & b;
8
9 endmodule
```

As you can see, an AND gate module is declared in the first line, and the module is implemented in module and

endmodule. Module is followed by the module name, followed by the module interface signals.

The input signal is indicated by input, and the output signal is indicated by output. The declared signals are separated by commas.

A semicolon is required after the module declaration brackets.

4.2 Variable Types

There are two data types in Verliog development, one is wire (wire) and the other is reg (register).

There are only two forms of signals in the circuit, one is transmission and the other is storage. Transmission is through connecting wires, and storage is through sending wires.

Therefore, it is clear that wire and reg variables are commonly used in Verilog HDL. Wire and reg variable models

As shown in the following figure:

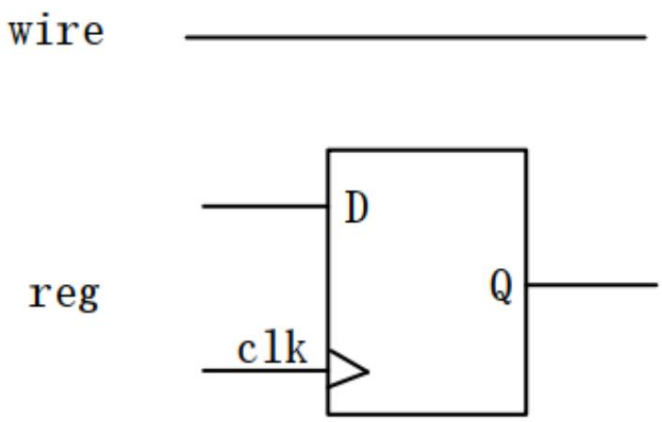


Figure 6- 2

A wire variable is just a wire in physical structure. When describing in Verilog HDL, the wire variable is assigned a value using

Just assign, it is relatively simple.

The reg type variable has an input port D on the left and an output port Q on the right, and the reg type storage data needs

This must be done under the control of the clk (clock) edge. clk is what we often call a square wave, which is generated by a crystal oscillator.

The most basic time unit in digital circuits, its period is fixed, and the duty cycle is generally 50% (i.e. the high level occupies the entire

The low level of clk is represented by the number 0, the high level is represented by 1, and the transition from low level to high level is

The process from high level to low level is called the falling edge.

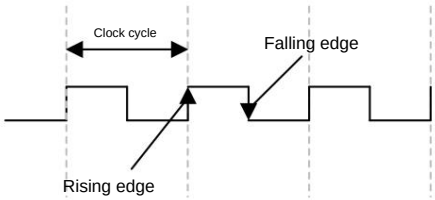


Figure 6- 3

When assigning a value to a reg type variable, it must be done in an always block. You can choose to use the rising edge of the clock.

You can choose the falling edge of the clock. Whether to use the rising edge or the falling edge can be determined according to your needs.

The output in the previous module declaration is changed to a reg type variable, and the result after the two inputs are ANDed is transferred to the register variable.

Quantity c, the RTL circuit diagram is shown in the figure below.

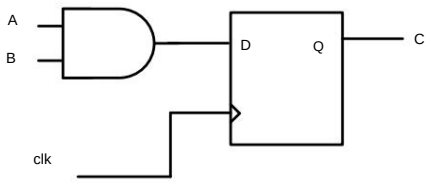


Figure 6- 4

The verliog code is as follows:

```
1 module and_gate(
2           input wire input wire      clk   ,
3           input wire                 a     ,
4                                      b     ,
5           output reg                 c
6           );
7
8 always @(posedge clk) begin
9           c <= a & b;
10 end
11
12 endmodule
```

The second line is the input clock, provided by the crystal oscillator

Since the output variable c is assigned in always, it must be of type reg

Lines 8 to 10 use the always block to assign values to reg type variables. @(posedge clk) indicates

Yes, the statements in the always block are executed whenever a rising edge of clk is encountered.

The assignment symbol (<=) in line 9 is a non-blocking assignment. There are two types of assignments in Veriog. One is

Non-blocking assignment (<=), the other is blocking assignment (=), non-blocking assignment is used in sequential logic, combinational logic

Use blocking assignment in .

In blocking assignment, the output changes when the input changes. In non-blocking assignment, the output changes only when the clock changes.

The output will change only when .

In digital circuits, all data is ultimately presented in binary form.

Like binary system, it is a form of data. Data can be expressed in both binary and decimal.

The same data is the same whether it is expressed in binary or decimal. The decimal data is

0 to 9, binary is 0, 1, each 0 or 1 in binary is called 1 bit.

How many bits does a piece of data have? First, the data needs to be converted into binary data.

How many bits are there to determine how many bits the data has.

In the previously defined data, since the bit width is not specified, each variable can only transmit or store

1-bit data. These variables can only express 0 or 1. If you want to express more than 0 or 1, you need to declare

Make the following changes to the previous code.

```
1 module and_gate(
 2          input wire                          clk   ,
 3          input wire input wire      [1:0]    a     ,
 4                                     [1:0]    b     ,
 5          output reg                 [1:0]    c
 6          );
 7
 8 always @(posedge clk) begin
 9          c <= a & b;
10 end
11
12 endmodule
```

Lines 3 to 5 define the format of wire and reg type variables with bit width. The middle text after the variable type keyword is

Write the highest and lowest bits of the data in brackets, separated by a colon. Suppose we define an eight-bit line

The data type variable a can be written as wire[7:0] a or wire[8:1] a. It is recommended that the lowest bit starts from 0.

4.3 Assignment Statements

There are three common assignment statements in Verliog HDL. We have already touched on two of them, assign and

Always These two assignment statements can eventually be synthesized to generate circuits.

Among them, assign is used for the assignment of combinational logic;

Always is used for sequential logic assignment;

There is another assignment statement called initial, which is often used to assign values to variables during simulation. Generally,

The initial assignment statement is only used in simulation files.

1) assign

When assigning values, the assign statement can only assign values to combinational logic.

For example, a clock can only be composed of wires and operators, and the variable assigned using assign must be of wire type.

```
1 wire        [2:0]    a,
2 reg         [2:0]    b,
3 wire        [2:0]    c
4
5 assign c = a | b ;
```

In the code shown above, lines 1 to 3 define three 3-bit variables using the assign method.

Assign a value to the wire variable c. The result of c is the sum of a and b. As you can see, the variable on the right side of the equal sign is

It can be either wire type or reg type.

2) Always assign values

Use always to assign values, which can implement combinational logic or sequential logic, but its

The object of assignment can only be register type data.

The always assignment statement implements sequential logic, which has been introduced before. Now let's introduce how to use

Use the always statement to implement sequential logic.

```
1 wire        [2:0]    a,
2 reg         [2:0]    b,
3 reg         [2:0]    c
4
5 always @(a or b) begin
6         c = a & b;
7 end
```

The third line defines a reg type variable c. c is assigned in the always block, so it can only be

Register type, but because the always block is assigned to combinational logic, it is not synthesized by the synthesizer.

It will be synthesized as wire type.

In the fifth line, always is followed by another way of writing in the form of @(). Here, there is no

posedge or negedge indicates that this statement is not triggered by edge, but by level, so the current

The path implements the combinational logic that is independent of the edge. The or in the brackets here can be changed to ",";

3) initial statement

The initial statement is an initialization statement. It will be executed once when the power is turned on (when the circuit is just running) and will not be repeated.

In a circuit, only registers that can store data need to be initialized, so the initial statement

The variable assigned in the statement must also be reg. Some third-party synthesis software considers initial to be unintegratable.

That is to say, it cannot be written into the function file, so for the compatibility of the code, we try to

Only write initial statements in test files. When initial and always are used together, they can well describe the situation.

The sample code is shown below.

```
1 reg          clk;
2 initial begin
3          clk = 0;
4          always #(5) clk = ~clk;
5 end
```

The above code will generate a clock with a period of 10ns. The first line defines a reg type

The third line initializes the variable, and the fourth line assigns a value to clk every 5ns.

4.4 Operators

The operators in Verilog HDL are basically the same as those in C language, so they will not be discussed in this summary.

There is not much to introduce here, so I will only introduce a few commonly used operators.

1) Relational operators (< , **<= , == , >= , > , !=)**

The relational operators are also similar to those in C. It should be noted that in Verliog,

There is still a slight difference in language.

For example, in C language, you can use if(2<a<6) to check whether a data is within a certain range.

But in Verilog, you need to use if(a>2 && a<6) to determine whether the data is within this range.

The analysis is as follows:

In Verilog, if we use if(2<a<6) to make a judgment,

When a=1, 2<1 is false and the result is 0. The result after operation is 0<6 is true. When a=8, 2<8

The result is 1. After the budget, 1<6 is true. You can see the condition after if. No matter what value a takes,

will remain true.

With the second way of writing, the condition after if is true only when both conditions are met.

2) Logical operators (&& , **||, !)**

Logical operators are consistent with those in C language: For &&, only when both data involved in the operation are true,

The result is true. For ||, as long as one of the two data involved in the operation is true, the result is true. (!)

The logic is negated when in progress.

3) Bitwise operators (& , **|,** ~)

The operation is performed according to each bit of the data involved in the operation. Bit operation is a binary operation.

When calculating, the data must be converted into binary before calculation can be performed.

```
1 wire[1:0] a=2'b10; 2
wire[2:0] b=3'd3;
3
4 assign c = a&b;
5 assign c = a|b;
6 assign c = ~a;
7 assign c = &a;
```

Lines 1 and 2 define two variables with initial values;

The fourth line implements a&b. At this time, both a and b need to be converted into binary, a=2'b10, b=3'b011.

When the bit widths of a and b are different, the one with smaller bit width will be padded with 0 in the high bit, and the final calculation is

3'b010 & 3'b011=3'b010ý

Line 5 implements a|b. At this time, both a and b need to be converted into binary: a=2'b10, b=3'b011.

At this time, the bit widths of a and b are different, and the one with smaller bit width is padded with 0 in the high bit, and the final calculation is

3'b010|3'b011=3'b011ÿ

Line 6 implements ~a, bitwise inversion, each bit is taken its opposite number, and the result of the operation is ~2'b10 = 2'b01;

Line 7 implements bitwise AND, and the result of the operation is a[1]&a[0] = 1&0 = 0;

4) Conditional Operator

Format: (x)?xxx:XXX; When using assign to assign values, sometimes the conditions are different for a certain

To assign a value to a variable, you can use the conditional operator:

For example: assign a = (b>6)?1'b1:1'b0;

When b is greater than 6, a is equal to 1, otherwise a is equal to 0;

5) Assignment Operator

We have already introduced the assignment operators in Verilog, <= and =, where <= is a non-blocking assignment.

Used for sequential logic, = is a blocking assignment, used for combinational logic.

6) Shift operators (<< , **>>)**

>> is right shift, one bit at a time, the high bit of the data is filled with zero, and the overflow data is discarded. << is the same as

The opposite of.

7) Bitwise concatenation operator ({})

The bit concatenation operator can concatenate bits of different data into a new data.

```
1 reg[3:0] a=4'b0110;
2 reg[4:0] b=5'b10110;
3
4 always@(posedge clk) 5
c <= {b[1],a[1:0],b[3],b[1]};
```

After splicing, c = 5'b11001;

4.5 Conditional Statements

The same variable can obtain different values in different situations, and different situations require judgment statements to Description. The judgment statements commonly used in Verilog HDL are if else statements and case endcase statements.

This judgment statement must be written in the always statement and cannot be written in the assign statement.

1) If-else

The if-else statement is similar to that used in C language.

```
always@(posedge clk)begin
        if(a==1)
                b <= 1;
        else if(a==2)
                b <= 2;
        else if(a==3)
                b <= 3;
        else if(a==4)
                b <= 4;
        else if(a==5)
                b <= 5;
end
```

It is not advisable to stack too many if else statements, otherwise it may cause excessive line delay.

A selector will be generated. When there are too many if else statements, the selector chain will be very long, and the number of selectors between each two levels will be very large.

There will be line delays. When there are too many links, the delays will be large.

The timing will be greatly affected. When the timing is wrong, even if the functional simulation is correct, the circuit will not be correct after the board is installed.

Yes.

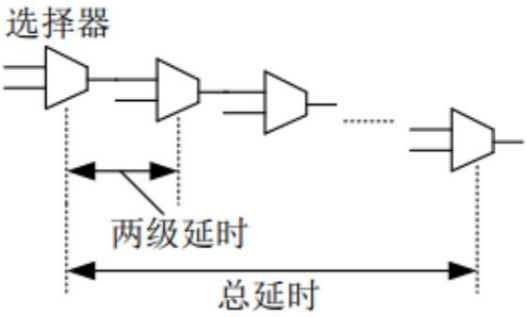


Figure 6- 5

When using if-else statements, you need to consider the order of priority. The conditions that are executed first are placed at the top, and the conditions that are executed first are placed at the bottom.

Lower level conditions are placed at the back.

2) Case Statement

When there are too many if-else series, you can also use case statements.

Path device.

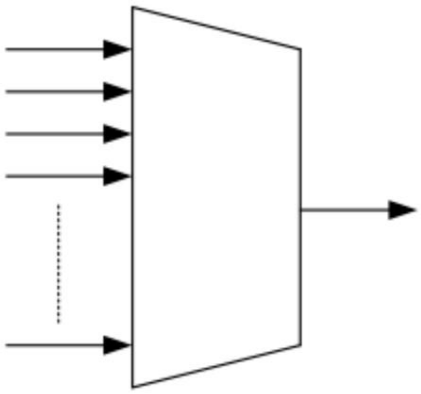


Figure 6- 6

```
1 always@(posedge clk) 2 case(a)


3           0: b <= 0;

4           1: b <= 1; 2: b

5           <= 2;

6           default:b <= 0;

7 endcase
```

The case statement starts with case and ends with endcase. In between, the conditions to be judged are listed.

Depending on the value of the condition, the corresponding code is executed.

4.6 Summary

In this chapter, the syntax of Verilog HDL is briefly introduced.

You must go deep into it, because in FPGA development, the essence is to develop circuits, and you don't have to stick to syntax.

In the subsequent contacts, you can find that there are only a few commonly used Verilog syntaxes. Mastering the above Verilog

The syntax is sufficient for most Verilog development. Verilog development focuses on timing and structure.

Code is just the implementation.

Chapter 5 Button Control **LED** Light

5.1 Experimental Introduction

In the previous chapter, the basic syntax of Verilog was introduced. In this chapter, a simple experiment is used to

Complete the experiment of lighting up the LED light by pressing a button, and further master the process of FPGA development.

There are two PL side buttons and LED lights on the Z7-Lite development board.

This experiment uses one of the LED lights and two buttons, PL_KEY1 is used to

System reset button.

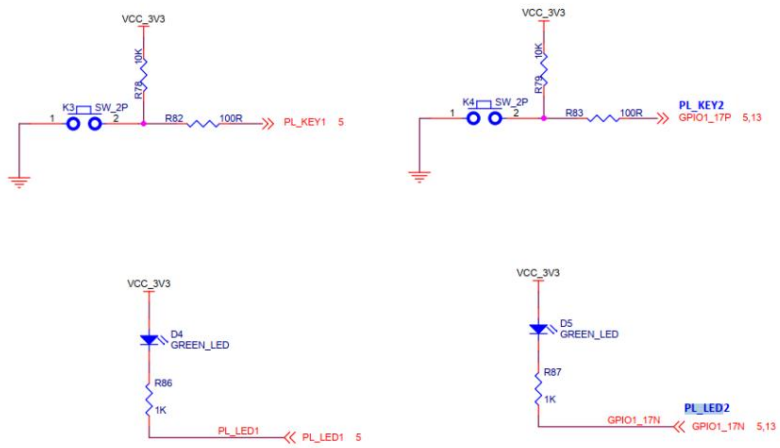


Figure 7- 1

5.2 Create a project

1) Open the Vivado development environment

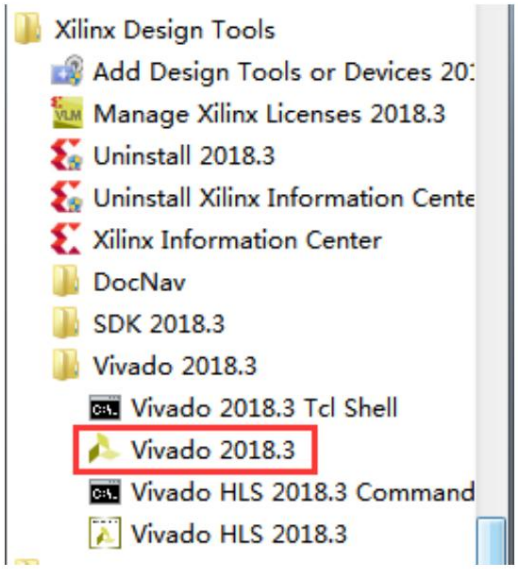


Figure 7- 2

2) Create a new project, enter the project name, and determine the project path

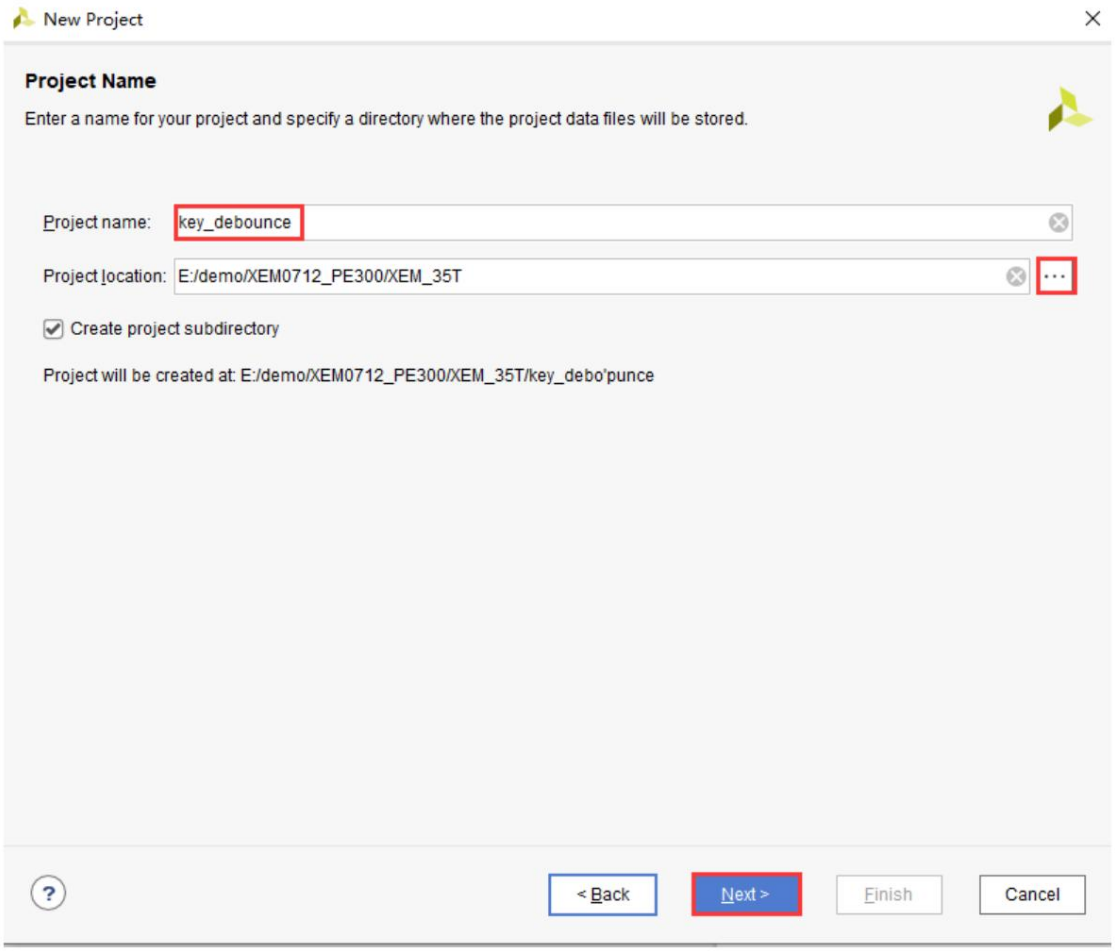


Figure 7- 3

Select the device. The device selection is the same as in the previous chapter. You can select it by filtering or by direct search.

Get the device, here we take xc7z020clg400-2 as an example

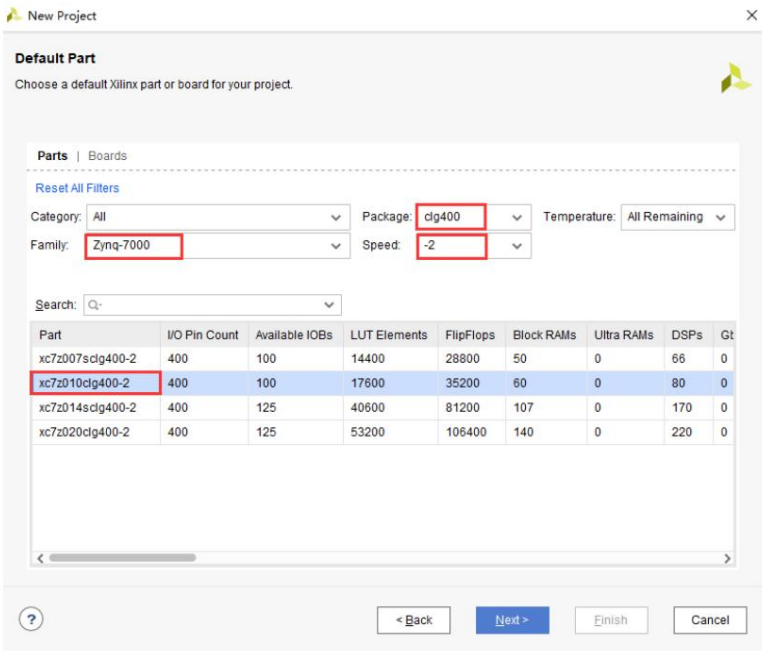


Figure 7- 4

Direct search yields:

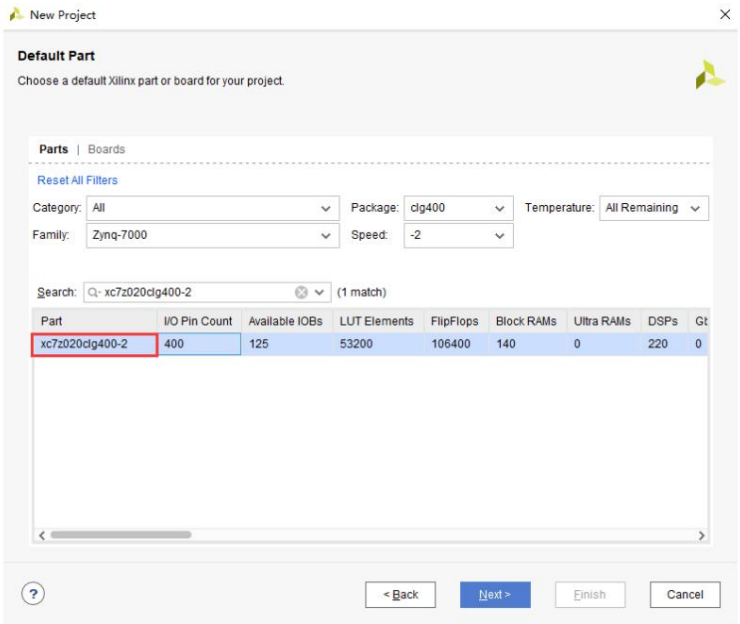


Figure 7- 5

3) Create a design file. You can click Add Source in the navigation bar or click the plus sign in the file management bar.

Add File

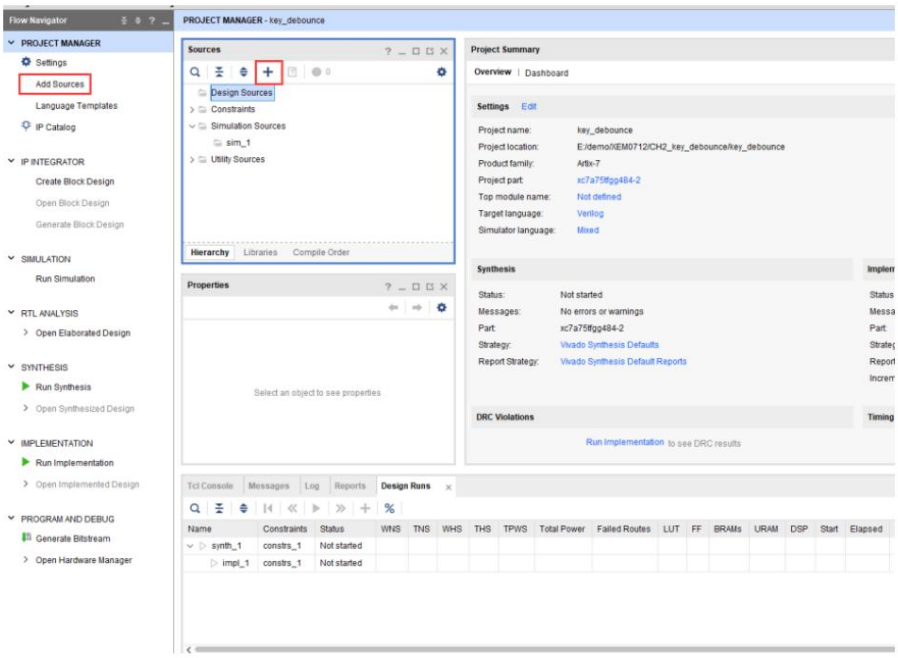


Figure 7- 6

First add a design file and name it key_led

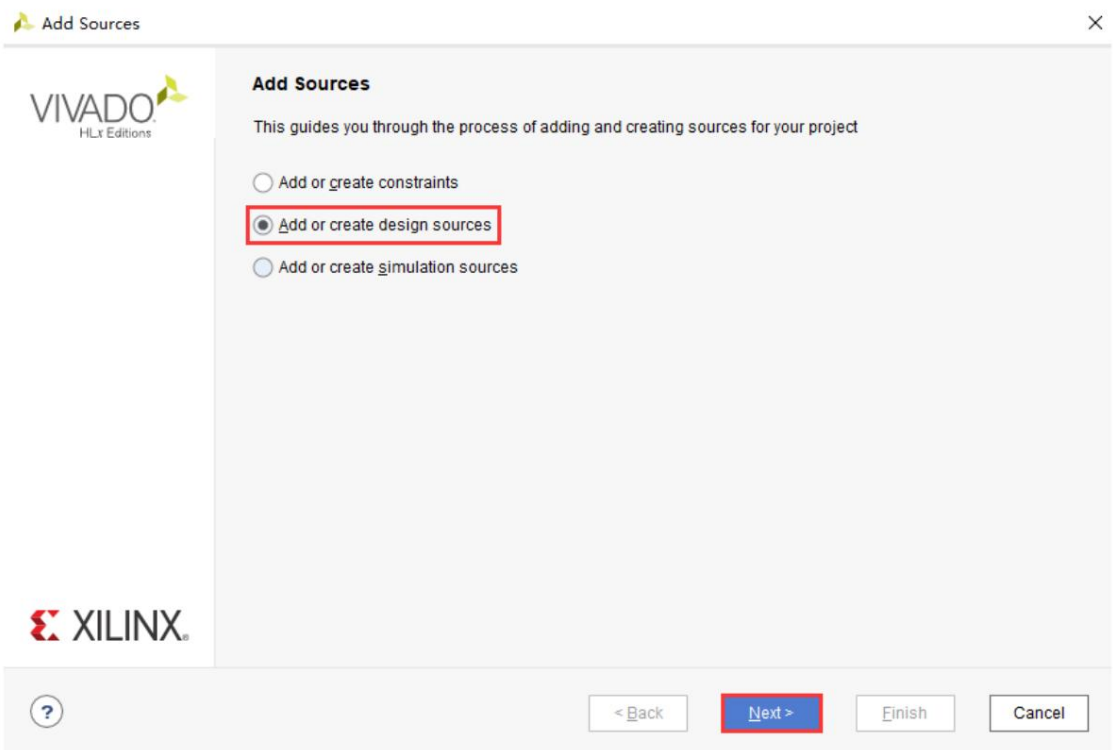


Figure 7- 7

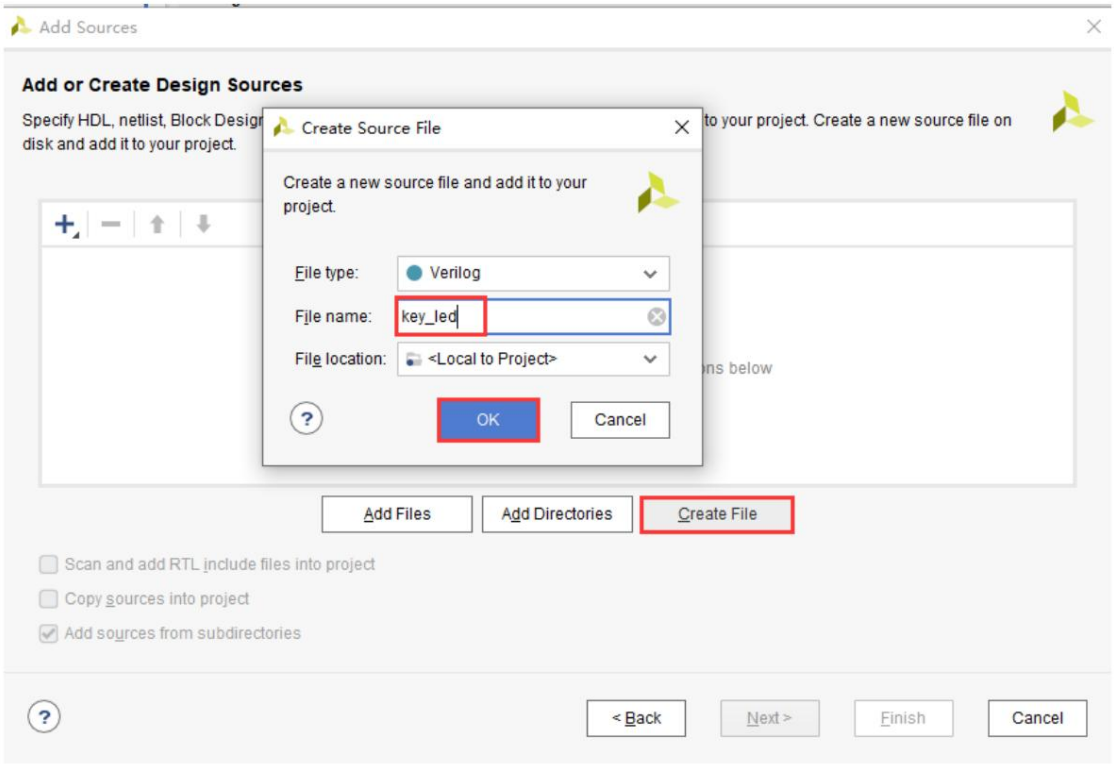


Figure 7- 8

5.3 Program Design

```
1 `timescale 1ns / 1ps
2 module key_led(
3 input wire 4 input                clk    ,
wire 5 input wire                          rst_n          ,
                                           key ,
6 output wire );                           led
7
8
9 //    Beat the input signal to eliminate metastable state
10 reg [1:0]key_dd ;
11
12 always @(posedge clk or negedge rst_n) begin
13 if (rst_n==1'b0) begin
14              key_dd <= 2'b11;
15 end
16 else begin
```

```
17              key_dd <= {key_dd[0],key};
18 end
19 end
20
21 //     Output    led The result after the beat
22 assign led = key_dd[1] ;
23
24 endmodule
```

Then click on Synthesis to view the results after synthesis:

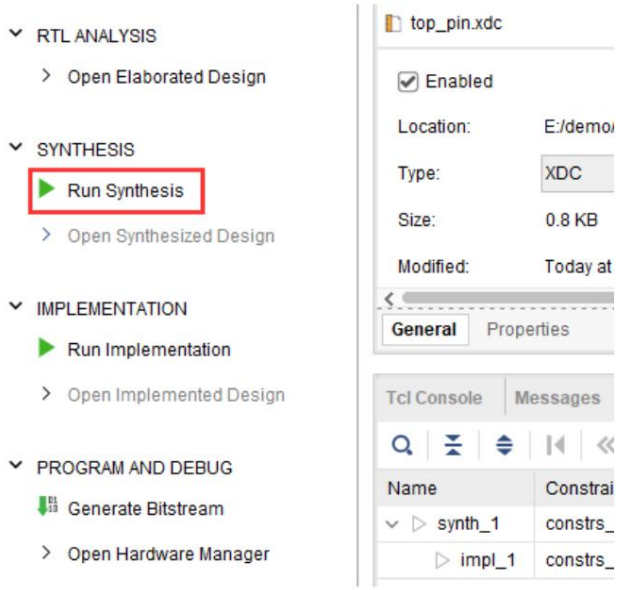


Figure 7- 9

After the neutralization is complete, you can open the synthesized results to see what the synthesizer has synthesized the code into.

Circuit structure:

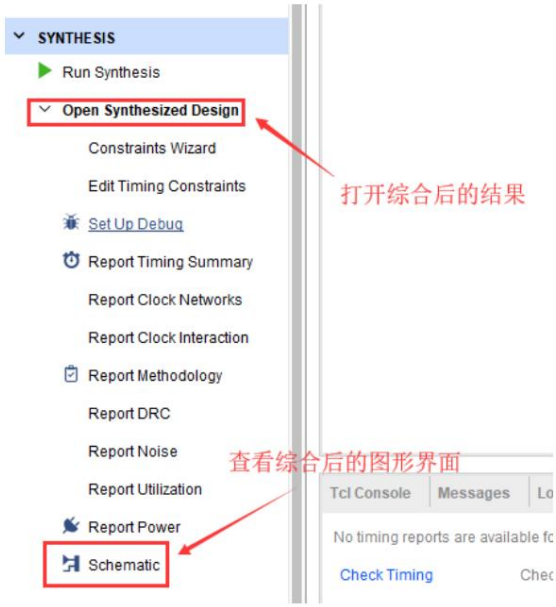


Figure 7- 10

It can be seen that after synthesis, the synthesizer generates 2 D flip-flops, which just corresponds to

The 2-bit width of key_dd uses bit splicing to complete the shift. By looking at the D flip-flop

The lines between them show the direction of the shifted data flow.

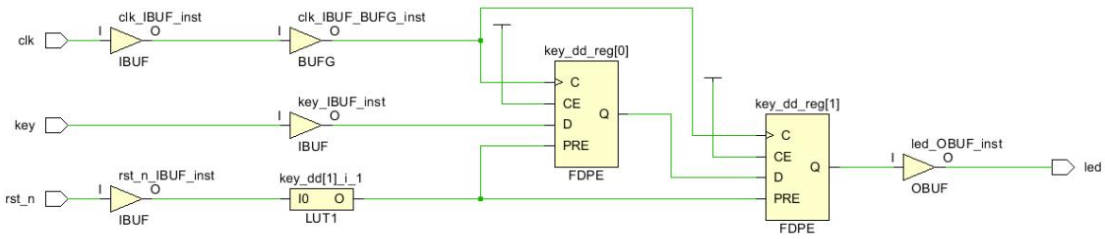


Figure 7-11 Z7-Lite series comprehensive results

You can also make pin assignments on this page, as shown below:

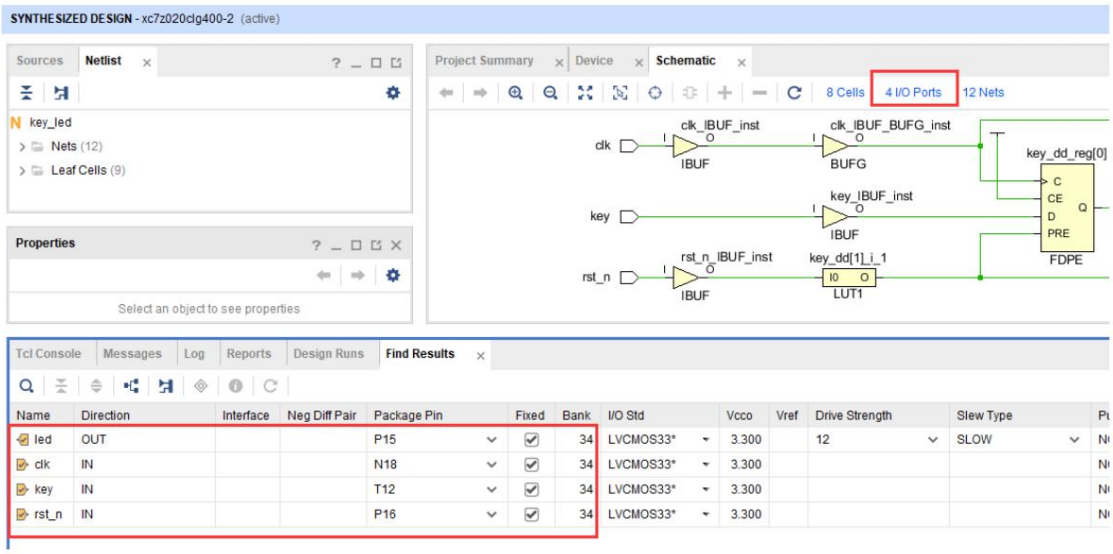


Figure 7- 12

After assigning the pins, press ctrl+s to save. Vivado will generate a constraint file. Click to view it.

Here's the constraints file.

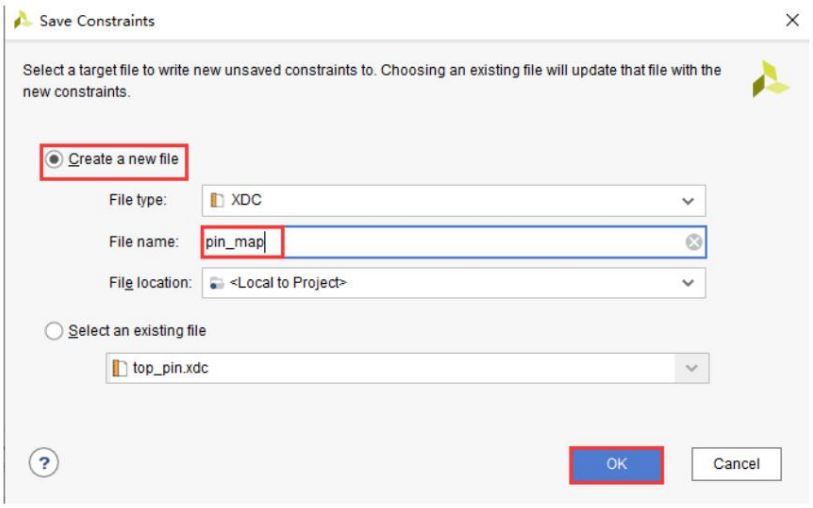


Figure 7- 13

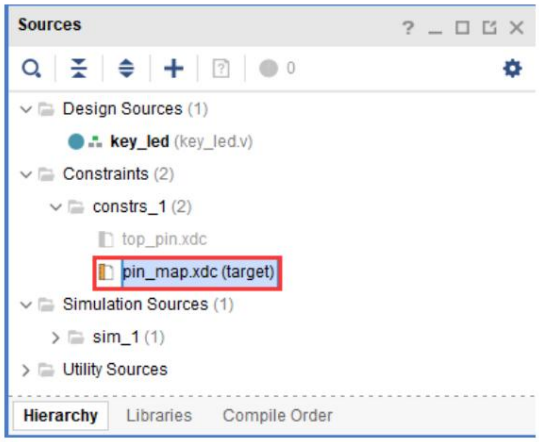


Figure 7- 14

After allocating the pins, you can proceed to the next steps, layout and routing, and generate the bit file.

5.4 Download Verification

After the bit file is generated, download the bit file into the FPGA, and then press the button to see the

Until the corresponding LED lights up.

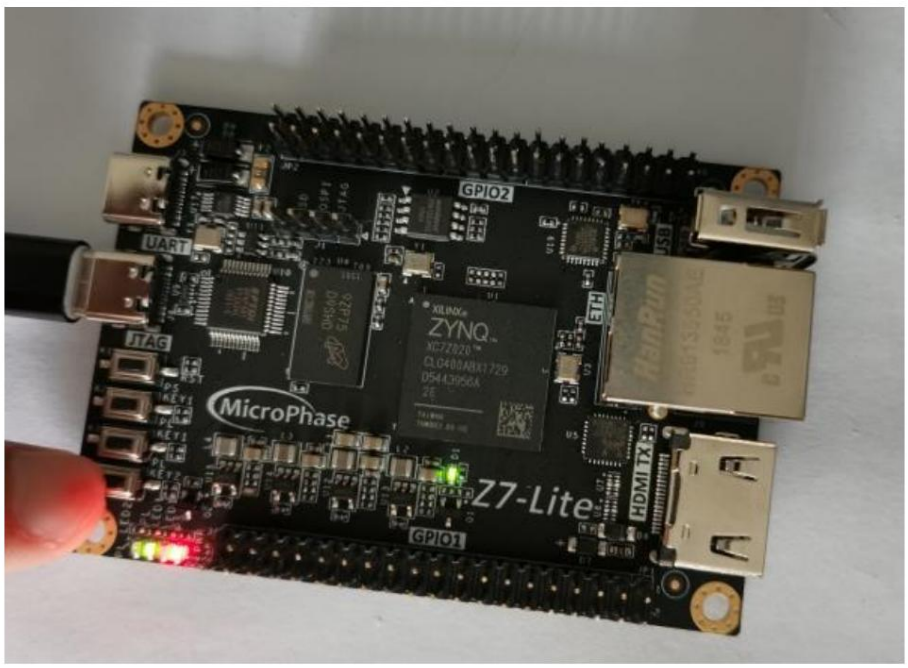


Figure 7-15 Z7-Lite experimental results

5.5 Experimental Summary

This experiment lights up the LED by pressing a button, which gives us a better understanding of the previous Verilog syntax.

Understand the role of synthesis tools in Vivado, and complete the pin assignment through the system pin assignment tool.

The creation of a constraint file.

6.1 Experimental Introduction

Mechanical buttons are commonly used devices, but they are always accompanied by jitter during use.

In this experiment, we will complete the debounce of the button and control the on and off of the LED light through the button.

6.2 Experimental Principle

The Z7-Lite development board has two PL-side buttons, as shown in the schematic diagram. One of them is a reset

This experiment uses PL_KEY to complete the debounce and control when PL_KEY is pressed.

LED lights.

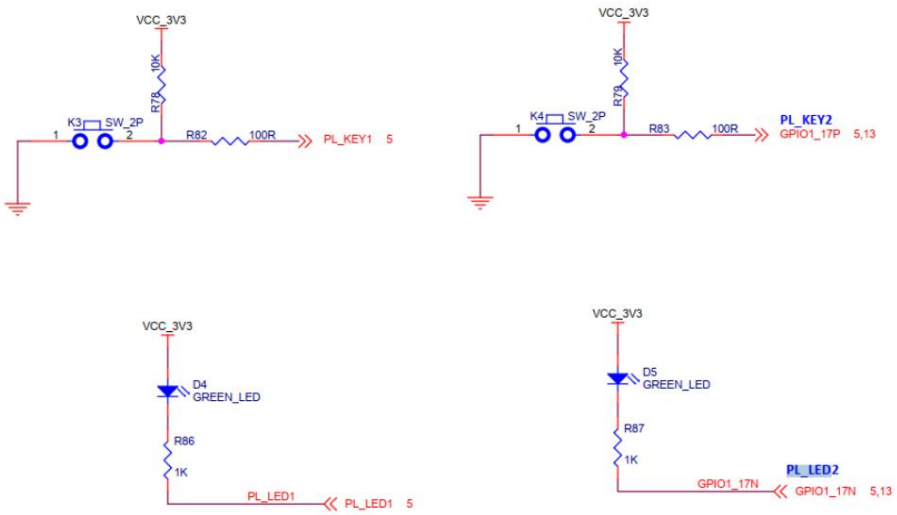


Figure 8- 1

The illustration of jitter generation:

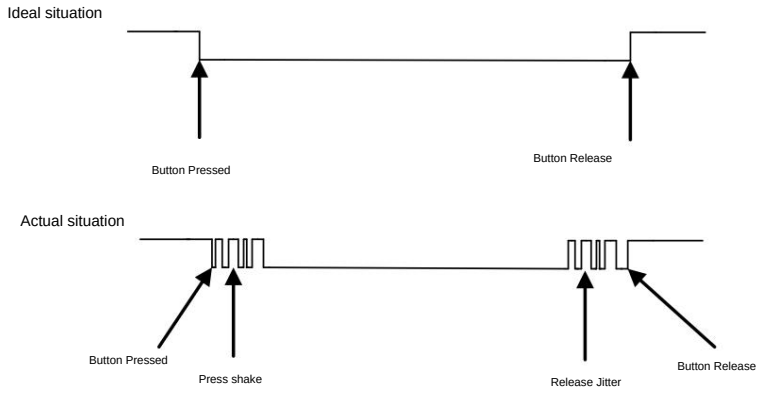


Figure 8- 2

Ideally, after a key is pressed, the key is at a low level, and when the key is released, it returns to a high level.

However, in actual situations, due to the mechanical structure of the key, the level cannot be instantly reset after the key is pressed.

That is, it remains low, and after a period of jump, it stabilizes and remains low. Similarly, when the key is released,

At times, the level cannot be maintained high immediately and is unstable for a period of time.

In this experiment, the key input level is kept low for 20ms to confirm that the key has been stable.

The counter counts the low-level time of the key input to eliminate key jitter.

6.3 Create a project

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is ram_pp, the project path is E:\demo\FPGA\ram_pp. Check Createproject subdirectoryÿNext
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device: Purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

6.4 Programming

Here is a method for key debounce. A counter is used in the program. When the counter detects

When the key input is low, it starts to accumulate, and when a high level is detected, it is cleared.

After a certain time, the key is recognized to be stable and a key stable signal key_flag is generated.

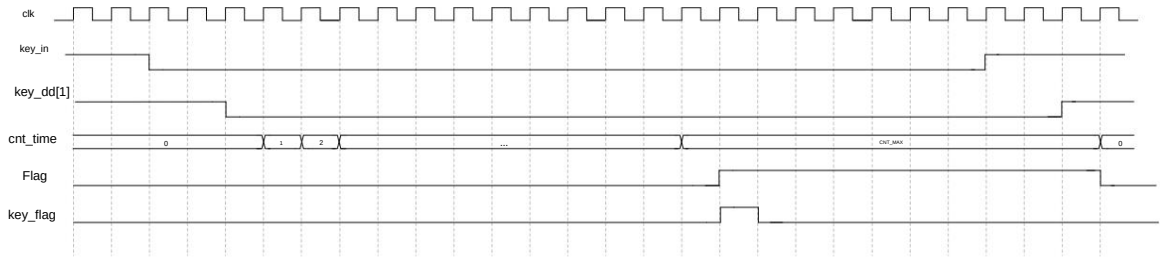


Figure 8- 3

```
1 module key_debounce(
2         input wire input                    clk         ,// system clock input 50M
3         wire                                rst_n           ,// system reset
4         input wire                          key_in          ,// key input
5         output reg              [1:0]       led
6 );
7
8 //================================================
9 //parameter define
10 //================================================
11 parameter CNT_MAX = 4000_000 - 1; //20ms counter max data
12
13 //================================================
14 //internal signals
15 //================================================
16 reg         [21:0] cnt_time                 ;// timer counter
17 wire                    add_cnt_time;//add condition
18 wire                    end_cnt_time;//end condition
19
20 reg         [1:0] key_dd                    ;// aim to Eliminate metastable state
21 reg                     key_flag            ;// key has been push down flag
22 reg                     flag                ;
23 reg         [1:0]       led_r               ;// led control
24
25
```

```
26 //--------------------key_dd--------------------
27 always @(posedge clk or negedge rst_n)begin
28         if(rst_n==1'b0)begin
29                 key_dd <= 'd0;
30         end
31         else begin
32                 key_dd <= {key_dd[0],key_in};//key_in delay 2 beats
33         end
34 end
35
36 //--------------------cnt_time--------------------
37 always @(posedge clk or negedge rst_n)begin
38         if(rst_n==1'b0)begin
39                 cnt_time <= 'd0;
40         end
41         else if(add_cnt_time)begin
42                 if(end_cnt_time)
43                         cnt_time <= cnt_time;
44                 else
45                         cnt_time <= cnt_time + 1'b1;
46         end
47         else begin
48                 cnt_time <= 'd0;
49         end
50 end
51
52 assign add_cnt_time = (key_dd[1]==1'b0);
53 assign end_cnt_time = add_cnt_time && cnt_time== CNT_MAX;
54
55 //--------------------flag--------------------
56 always @(posedge clk or negedge rst_n)begin
57         if(rst_n==1'b0)begin
58                 flag <= 1'b0;
59         end
60         else if(cnt_time==CNT_MAX)begin
61                 flag <= 1'b1;
62         end
63         else if(key_dd[1]==1'b1)begin
64                 flag <= 1'b0;
65         end
66         else begin
```

```
67                  flag <= flag;
68          end
69 end
70
71 //--------------------key_flag--------------------
72 always @(posedge clk or negedge rst_n)begin
73          if(rst_n==1'b0)begin
74                  key_flag <= 1'b0;
75          end
76          else if(cnt_time == CNT_MAX && flag==1'b0)begin
77                  key_flag <= 1'b1;
78          end
79          else begin
80                  key_flag <= 1'b0;
81          end
82 end
83
84 //--------------------led_r--------------------
85 always @(posedge clk or negedge rst_n)begin
86          if(rst_n==1'b0)begin
87                  led_r <= 2'b01;
88          end
89          else if(key_flag==1'b1)begin
90                  led_r <= {led_r[0],led_r[1]};
91          end
92          else begin
93                  led_r <= led_r;
94          end
95 end
96
97 //--------------------led--------------------
98 always @(posedge clk or negedge rst_n)begin
99          if(rst_n==1'b0)begin
100                 led <= 2'b11;
101         end
102         else begin
103                 led <= ~led_r;
104         end
105 end
106
107 endmodule
```

In this program design, a shift register key_dd is used to input key signals.

Performing metastable processing and delaying the input signal by two beats can effectively reduce the metastable state.

A counter cnt_time is used to count the time when the key input is low level.

When the input is high, the counter is cleared. When the count value required for 10ms is reached, the counter keeps

constant.

A count indication signal flag is used to locate and indicate whether the current cnt_time has reached the count value.

The maximum value required.

A key press indication signal key_flag is used to indicate that the key is pressed successfully.

Use led_r to control the movement of the LED light, and control the brightness of the LED light by detecting key_flag.

Destroy.

6.5 Add pin constraints

1) First create a pin constraint file, click Add Source in the navigation bar, and select the first item in the pop-up interface.

Add constraints file.

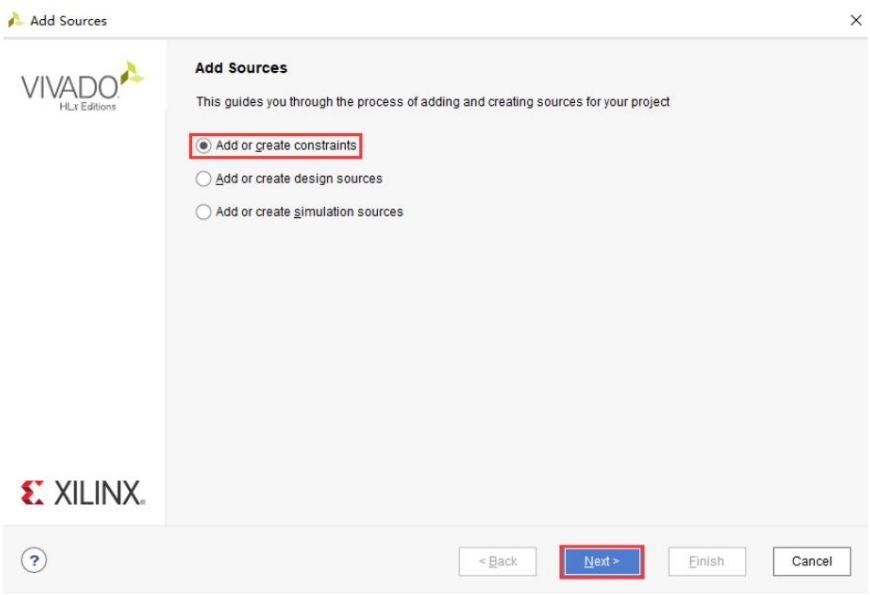


Figure 8- 4

2) In the pop-up interface, select Create File and name the file **top_pin**

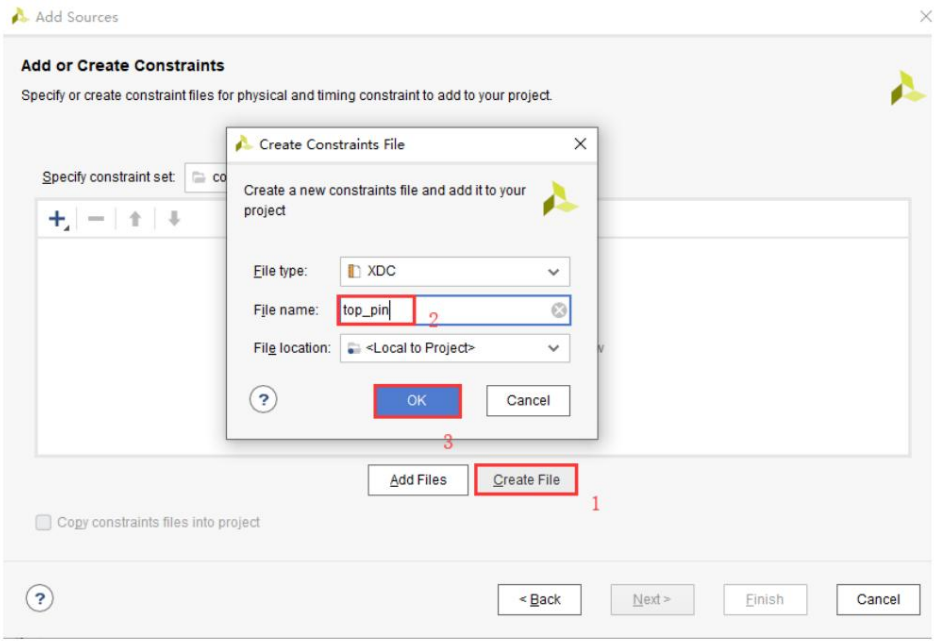


Figure 8- 5

3) Add constraint information

Make clock period, pin and level constraints for the input differential clock.

Make pin and level constraints for LED and buttons respectively.

Z7-Lite series constraint files:

```
############### clock define###################
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
############### key define###################
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property PACKAGE_PIN T12 [get_ports key_in]
###############LED define###################
set_property PACKAGE_PIN P15 [get_ports {led[0]}]
set_property PACKAGE_PIN U12 [get_ports {led[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports {led[*]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports key_in]
```

4) Save the pin constraints and click Synthesis, Implementation, and Generate Bit File in sequence. Or just click Generate Bit File.

Special file.

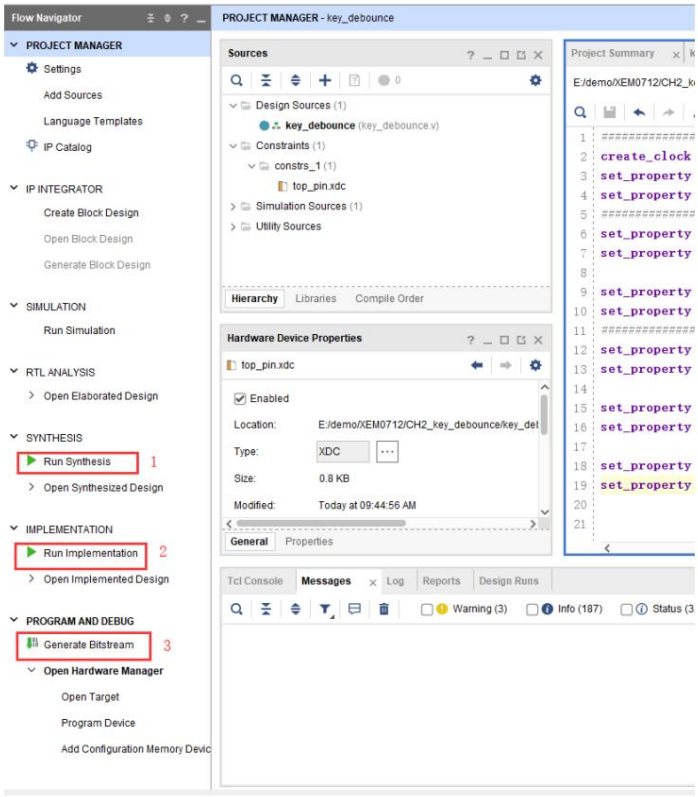


Figure 8- 6

6.6 Download Verification

1) After the bit file is generated, you can connect the development board to download the bit file. Connect the downloader to JTAG port, power on the board, click Open Hardware Manager, and in the Hardware Manager window

Click Open target, then click Auto Connect.

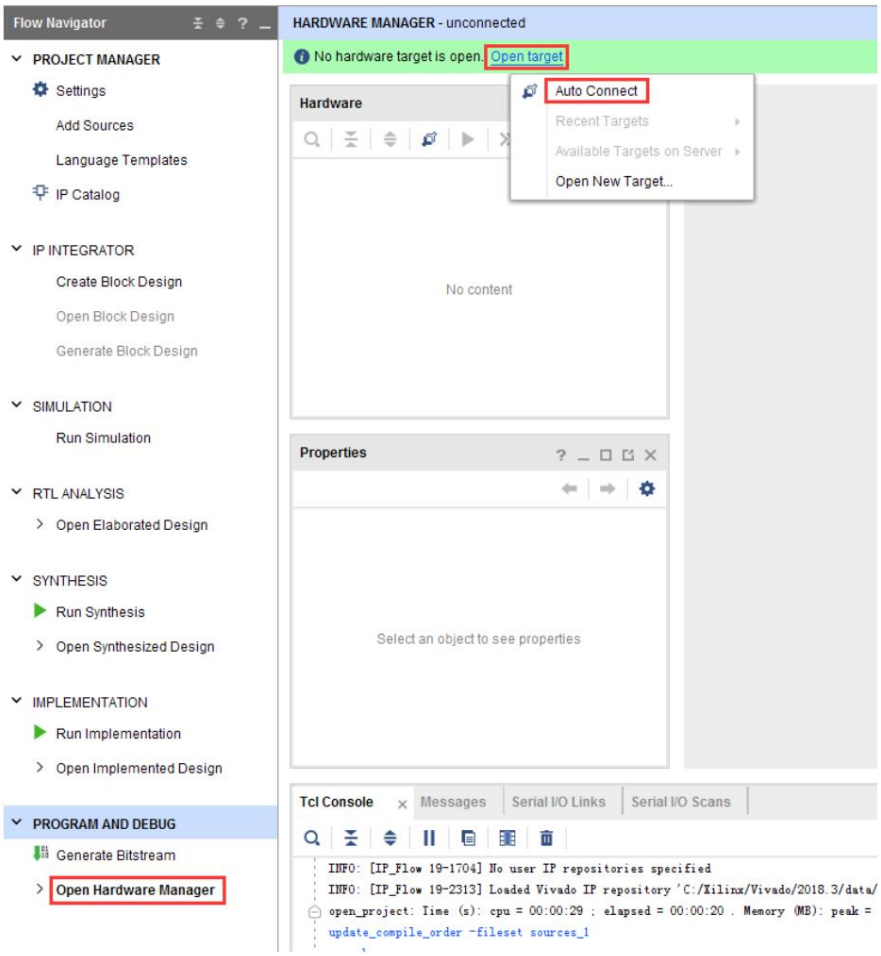


Figure 8- 7

2) Select program device, download the program to the development board, and observe the experimental phenomena.

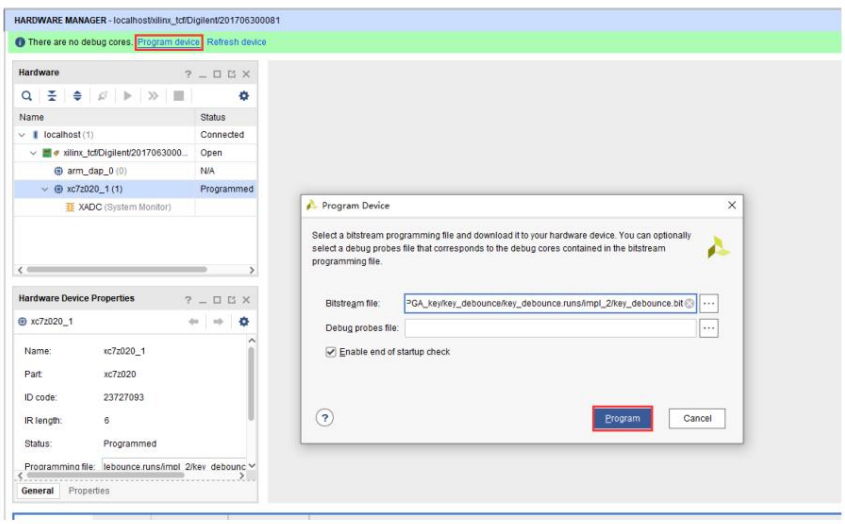


Figure 8- 8

3) Observe the phenomenon

After the Bit file is downloaded, press PL_KEY2 to observe the LED light turning on and off alternately.

The LED light moves one position each time. And the LED light will not jump multiple times when the button is pressed once.

6.7 Experimental Summary

Through this experiment, the debounce of the key is completed. This experiment is mainly to deepen the reader's understanding of the counter

This structure is often used in FPGA development, and it once again made me familiar with the FPGA development process.

Through this experiment, I believe readers will have a deeper understanding of Vivado's development of Verilog code.

Chapter 7 PWM Breathing Light

7.1 Experimental Introduction

Pulse Width Modulation (PWM) is a method of using the digital output of a microprocessor/FPGA to modulate analog

PWM is a very effective technology for controlling the circuit, which is widely used in many fields such as measurement, communication, power control and conversion.

Digital signals are in digital form from the processor to the controlled system, without the need for digital-to-analog conversion.

PWM is a method of digitally encoding analog signal levels. Through the use of high-resolution counters,

The duty cycle of the square wave is modulated to encode the level of a specific analog signal.

The dc supply is either fully present (ON) or fully absent (OFF) at any given moment.

The voltage or current source is applied to the simulated load in a repetitive sequence of pulses that are either ON or OFF.

When it is on, the DC power supply is added to the load, and when it is off, the power supply is disconnected.

Wide enough, any analog value can be encoded using PWM.

In layman's terms, PWM is a continuous pulse signal with a certain duty cycle.

Simply put, we can think of PWM as a square wave.

This experiment will use PWM to complete a breathing light. The breathing light will go from off to bright and then off within 2 seconds.

process.

7.2 Experimental Principle

The output level controls the LED light. When the output is low level, the light is on, and when the output is high level, the light is off.

If the output is low level, the light will be on; if the output is high level, the light will be off; if 50%

If the duty cycle of the LED is 50% low level and 50% high level, the light will be dimmer.

The brightness of the light.

Another factor that affects the brightness of the LED is the period of the PWM waveform.

If the duty cycle is 2 seconds and the duty cycle is 50%, then what we see is that the LED will be dark for 1 second and bright for 1 second, instead of half a second.

Only when we increase the PWM period can we see the half-bright state. According to experience, PWM

The cycle is preferably 10 milliseconds. In other words, the time to control the high and low levels, that is, the duty cycle (the high level takes

By controlling the PWM cycle, you can control the brightness of the light.

7.3 Create a project

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is pcie_test, the project path is E:\demo\pll, check Createproject subdirectoryÿNext
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device, purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

7.4 Programming

This experiment uses two counters, one for counting 10ms, which is the period of the PWM square wave.

One is used to count 2s, in order to determine how many PWM waves have been generated within 2s.

Use work_flag to indicate whether the duty cycle of the current square wave's high level is increasing or decreasing.

When work_flag == 1, the LED is turned off and the duty cycle of the square wave should be reduced.

When the LED is turned on, the duty cycle of the square wave should be increased, and duty_cycle is used to indicate when

In the previous PWM wave, the high level accounts for the proportion of the entire square wave.

The program design can refer to the following timing design diagram:

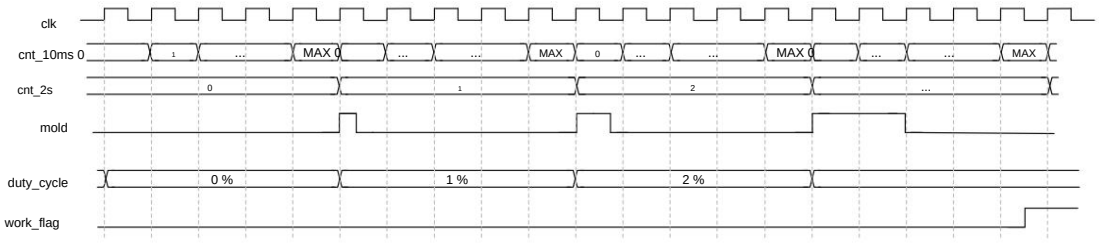


Figure 9-1

Below is the code for this experiment

```
1 `timescale 1ns / 1ps
2 module breath_led(
3 inputwire                clk     ,
4 input          wire                rst_n          ,
5 output         wire                led
6         );
7
8
9 //=======================================
10 //parameter define
11 //=======================================
12 parameter CNT_10MS = 500000 - 1;                    // 10ms     Maximum required count
13 parameter CNT_2S = 200 - 1;                    // 2s     Maximum required count
14 parameter CHANGE_TIME = 100 - 1;// 1s                Count maximum value
15 parameter PWM_OFFSET = 5000 ; //                The change value of each pulse modulation
16
17 reg [19:0]              cnt_10ms ;// 10ms          counter
18 reg [7:0] cnt_2s                        ;//    Two-second counter
19 reg                     mold                ;//    Modulated square wave
20 reg [19:0]              duty_cycle          ;//    Duty Cycle
21 reg                     work_flag ;//          Working status,  0  From off to bright,  1  From light to dark
22
23 assign led = ~pwm;
24
25
26 //----------cnt_10ms----------
27 always @(posedge clk or negedge rst_n) begin
28 if (rst_n==1'b0) begin
29          cnt_10ms <= 'd0;
30 end
31 // 10ms          Count has been reached, clear to zero
```

```
32 else if (cnt_10ms == CNT_10MS) begin
33              cnt_10ms <= 'd0;
34 end
35 //       Other times are carried out 10ms    count
36 else begin
37              cnt_10ms <= cnt_10ms + 1'b1;
38 end
39 end
40
41 //----------------cnt_2s------------------
42 always @(posedge clk or negedge rst_n) begin
43 if (rst_n==1'b0) begin
44              cnt_2s <= 'd0;
45 end
46 // 10ms          Add one when the count is complete
47 else if (cnt_10ms == CNT_10MS)begin
48             // 2s     Counting completed
49              if (cnt_2s == CNT_2S) begin
50                      cnt_2s <= 'd0;
51              end
52              else begin
53                      cnt_2s <= cnt_2s + 1'b1;
54              end
55 end
56 end
57
58 //----------------work_flag------------------
59 always @(posedge clk or negedge rst_n) begin
60 if (rst_n==1'b0) begin
61              work_flag <= 1'b0;
62 end
63 // 0~1          seconds, gradually brightens, counts to1s  After that, the status changes
64 else if (cnt_2s == CHANGE_TIME && cnt_10ms == CNT_10MS) begin
65              work_flag <= 1'b1;
66 end
67 // 1~2          seconds, gradually extinguished, count to2s  After that, the status changes
68 else if (cnt_2s == CNT_2S && cnt_10ms == CNT_10MS) begin
69              work_flag <= 1'b0;
70 end
71 end
72
```

```
73 //----------------duty_cycle------------------
74 always @(posedge clk or negedge rst_n) begin
75 if (rst_n==1'b0) begin
76             duty_cycle <= 'd0;
77 end
78 //       From off to on,      mold     The duty cycle increases
79 else if (work_flag == 1'b0) begin
80             if (cnt_10ms == CNT_10MS) begin
81                     duty_cycle <= duty_cycle + PWM_OFFSET;
82             end
83             else begin
84                     duty_cycle <= duty_cycle;
85             end
86 end
87 //       From on to off, the duty cycle decreases successively
88 else if (work_flag == 1'b1) begin
89             if (cnt_10ms == CNT_10MS) begin
90                     duty_cycle <= duty_cycle - PWM_OFFSET;
91             end
92             else begin
93                     duty_cycle <= duty_cycle;
94             end
95 end
96 end
97
98 //----------------mold--------------------
99 always @(posedge clk or negedge rst_n) begin
100 if (rst_n==1'b0) begin
101             pwm <= 1'b0;
102 end
103 else if (cnt_10ms < duty_cycle) begin
104             pwm <= 1'b1;
105 end
106 else begin
107             pwm <= 1'b0;
108 end
109 end
110
111 endmodule
```

7.5 Simulation Verification

After designing the source code, you can perform simulation verification. Simulation is a very important part in FPGA development.

Steps, through simulation, we can find out the problems in the code, and then we can go back and modify the code to make it

Be able to perform the correct function.

1) Create a new simulation file

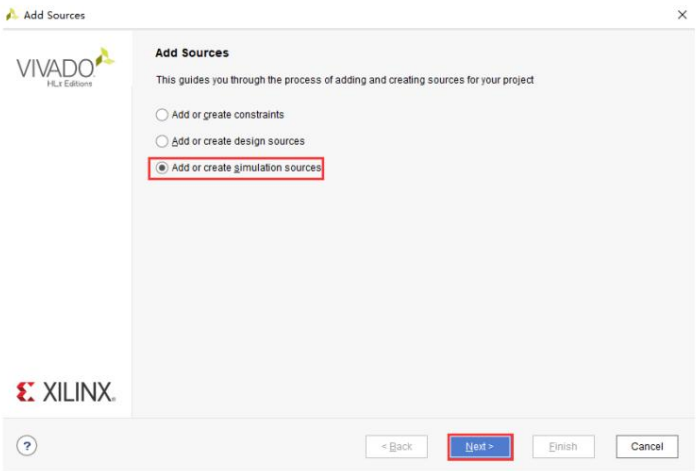


Figure 9- 2

The code design of the simulation file is as follows

```
1 `timescale 1ns/1ps
3 module tb_breath_led (); /* this is automatically generated */
4
5 reg rst_n;
6 reg clk;
7 wire led;
8
9 // (*NOTE*) replace reset, clock, others
10
11 parameter CNT_10MS = 500000 - 1;
12 parameter CNT_2S                = 200 - 1;
13 parameter CHANGE_TIME = 100 - 1;
14 parameter PWM_OFFSET = 5000;
15
16 breath_led #(
17                     .CNT_10MS(CNT_10MS),
```

```
18                      .CNT_2S(CNT_2S),
19                      .CHANGE_TIME(CHANGE_TIME),
20                      .PWM_OFFSET(PWM_OFFSET)
21              ) inst_breath_led (
22                      .clk        (clk),
23                      .rst_n (rst_n),
24                      .led        (led)
25              );
26
27 // clock
28 initial begin
29              clk = 0;
30              forever #(10) clk = ~clk;
31 end
32
33 initial begin
34              rst_n = 0;
35              repeat(10) @(posedge clk);
36              rst_n = 1;
37
38 end
39
40 endmodule
```

Lines 5 to 7 of the simulation file are the definitions of the input and output signals. In the access file, the stimulus signal is represented by

The reg type indicates that the signals between intermediate modules and the output signals use the wire type.

Lines 11 to 14 are the parameter definitions for this experiment. By modifying the parameters, the simulation efficiency can be improved.

Lines 1 to 25 are the instantiation of the module to be simulated, in which parameters are passed to the module to be simulated.

Lines 28 to 38 are the assignments to the input signals.

After the simulation file is designed, simulation can be performed. Vivado is an integrated development environment with

There are simulation tools. You can use the simulation tools provided by Vivado to complete the simulation.

Third-party simulation tools, such as Modelsim, are used to complete the simulation.

2) Use **Vivado** 's built-in simulation tool to complete the simulation

ÿÿ settings ÿ Simulation ÿ Vivado simulator ÿ Apply ÿ OK;

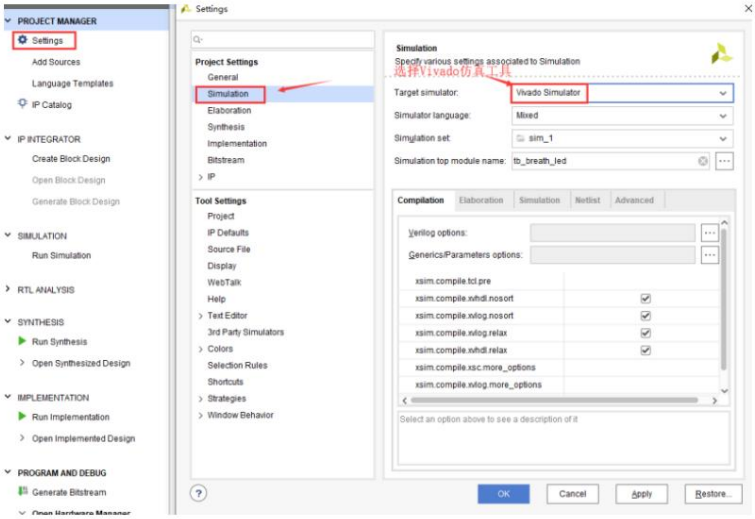


Figure 9- 3

After the settings are completed, click Run simulation to start the simulation, and then Vivado will pop up the following figure

The simulation interface shown in the figure shows that the red window can view each module and hierarchy, and the purple window can view

To see the signal inside the corresponding module, the cyan window can view the waveform of the corresponding signal;

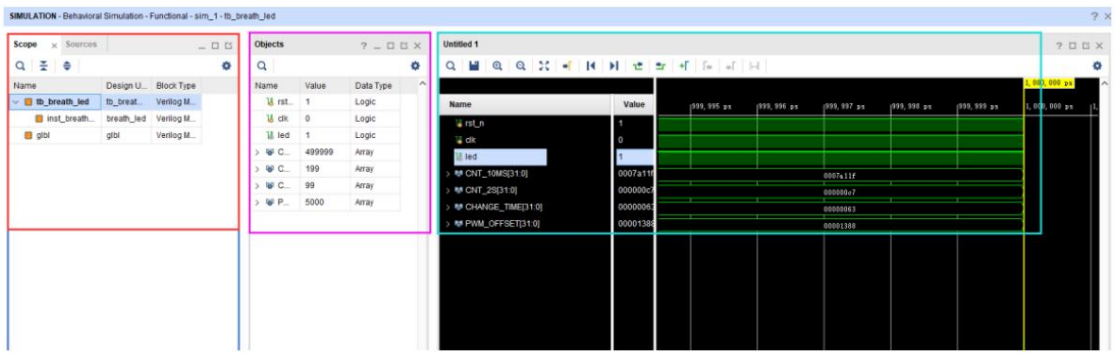


Figure 9- 4

If you want to observe a signal, just select it, right-click it, and add it to the waveform window.

Below:

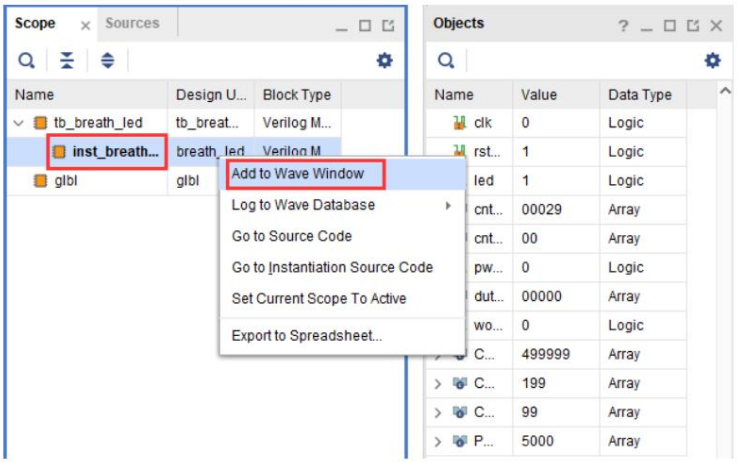


Figure 9-5

At the top of the simulation interface, there is a simulation start control option, which can be used to set the simulation, such as

Time, etc. Click here to run the simulation, and you can observe the waveform you want to display in the waveform window.

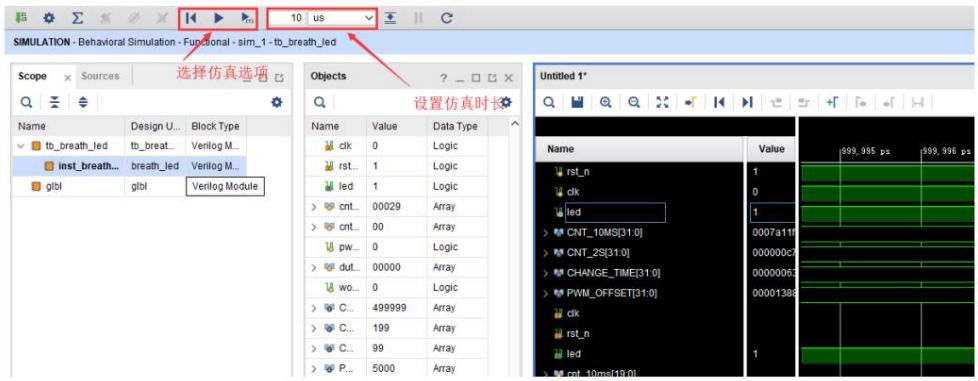


Figure 9-6

Through simulation, it can be found that the duty cycle of the PWM square wave increases from zero to 100% within two seconds, and then

Gradually decreases to 0. Through simulation, it can be judged that the function of the source program is correct. Next, bit generation can be performed.

File download verified.

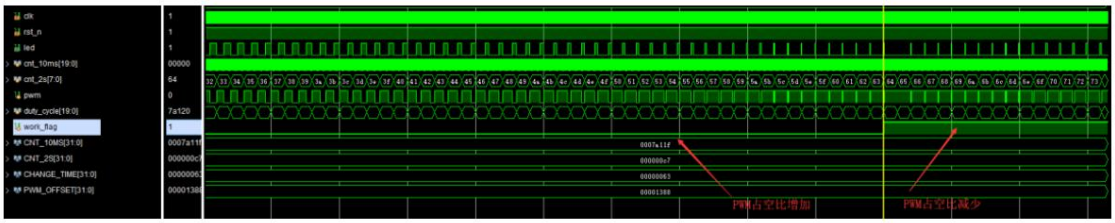


Figure 9-7

7.6 Download Verification

1) Add constraint file

The constraint files for this experiment are as follows:

Z7-Lite series constraint files:

```
############### clock define###################
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
############### key define###################
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
###############LED define###################
set_property PACKAGE_PIN P15 [get_ports led]

set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports led]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
```

2) Download verification

After generating the bit file, download the bit file into the FPGA, and then observe the phenomenon. You can see that LED1 is

From off to on and then off again, a breathing light experiment was completed:

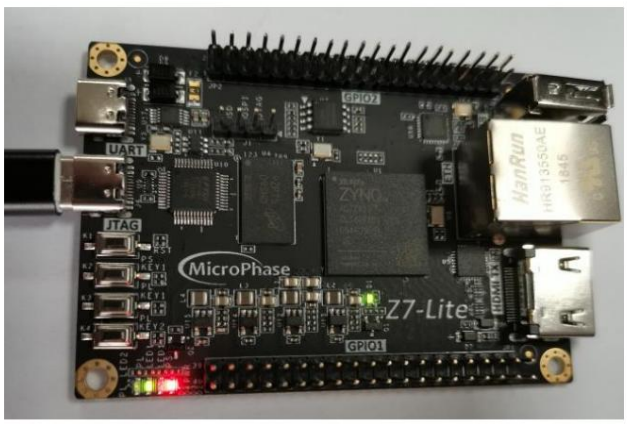


Figure 9- 8

7.7 Experimental Summary

Through this experiment, we completed a breathing light experiment. In this experiment, we learned how to use PWM to pulse

In addition, this experiment also designed a very important structure in FPGA development.

When developing FPGA, counters are often used to detect certain signals.

Counting, counters are often used in subsequent development, and may appear frequently in FPGA development.

The structure with the highest rate.

Regarding the counter, we need to clarify its increment condition, that is, under what circumstances the counter increments; and

Clearing conditions, that is, under what circumstances the counter needs to be cleared.

The structure is very simple.

Chapter 8 On-Chip Resources: **PLL**

In this chapter, readers will be introduced to Vivado development and design using IP provided by Xilinx.

This experiment will use modelsim for simulation. If the reader has not installed modelsim, it is recommended that the reader install it first

Of course, readers can also use the simulator that comes with Vivado to simulate this experiment.

real.

8.1 Experimental Introduction

The clock provided by the development board generates a fixed frequency signal after the crystal oscillates, which serves as our clock.

However, this clock is relatively fixed. However, in various experiments, clocks of different frequencies may be required.

At this time, we can use the counter to divide the incoming clock and the system clock, which can be compared with the system

If you need a clock with a lower clock frequency than the system clock frequency, then we users can use

It may not be so simple to implement it with logic code, or even if you get a clock with a fast frequency, it may not be able to

Use, as some burrs may occur.

At this point, we need to use the built-in functions of the Vivado tool to multiply and divide the clock.

This means that a clock with a lower frequency can generate a clock with a higher frequency. Here we need to use a new knowledge point.

That is, IP core, which means intellectual property, is actually a packaged module integrated into Vivado.

If you need to apply this module to an actual project, you need to call it, and you need to set the corresponding

The detailed steps for the parameters are described in screenshots below.

8.2 Create a project

Open Vivado and follow the steps in the table.

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is ram_pp, the project path is E:\demo\FPGA\pll, check Createproject subdirectory, Next
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device: Z7-Lite7020 User selects xc7z020clg400-2 and purchases Z7-Lite7020 User selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

8.3 IP core generation steps and parameter settings

1) Select IP Catalog in the navigation bar, and the IP Catalog interface will pop up.

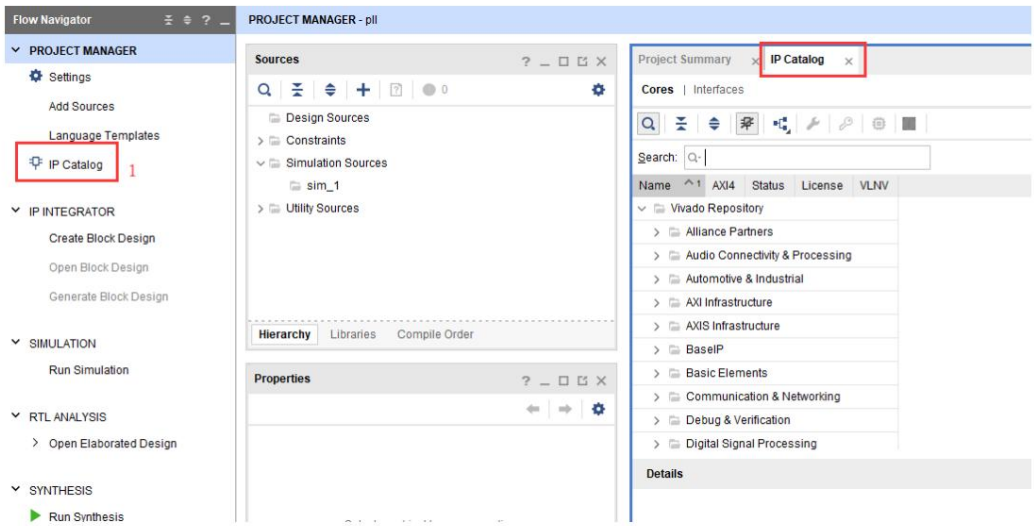


Figure 10- 1

2) Enter clock in the search field, select clocking wizard, and double-click to display the configuration interface of the clock IP.

noodle

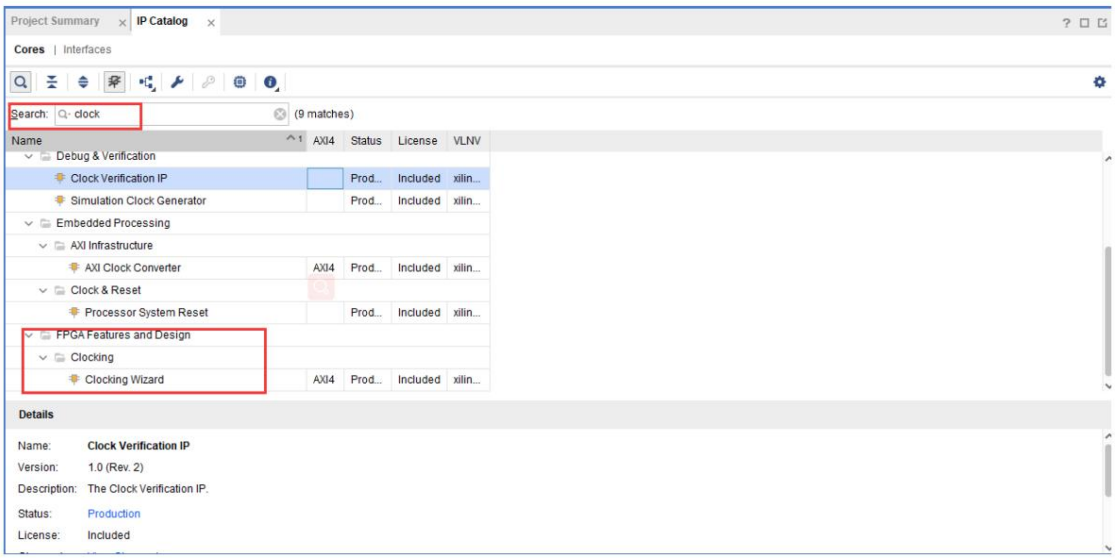


Figure 10- 2

3) On the first page of the configuration interface, you can modify the IP name and determine the frequency and Type, in this experiment, change the IP core name to clock, input clock to 200M, select Select differential clock. Because the output of the onboard clock management chip is 200M differential After setting, select the next page, output clocks.

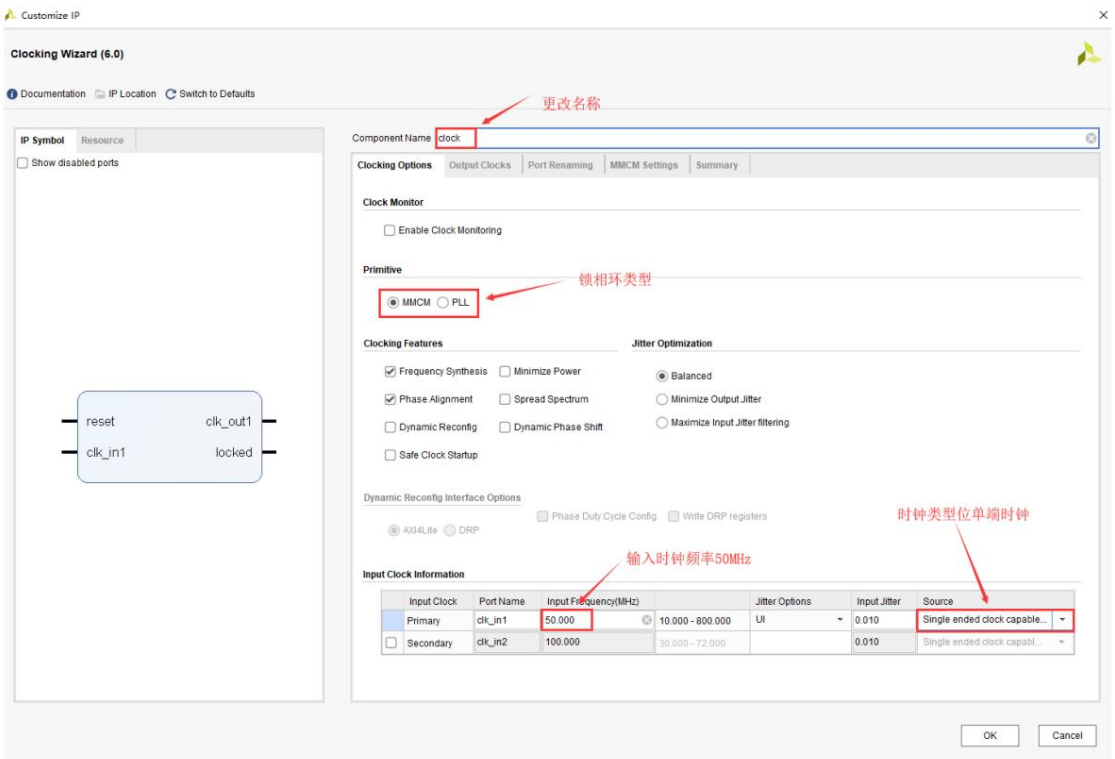


Figure 10- 3

4) In the output clocks interface, you can select the number of output clocks, the frequency, phase, and

The output clock waveform is controlled by the duty cycle, etc. In this experiment, three output clocks are used, with different frequencies.

The phases are 0 and 90 respectively. The duty cycle remains the default.

The reset signal of the IP core is the core lock signal. In this experiment, the reset type is high level reset. The Locked signal is

A signal that will be pulled high when all output clocks are stable, indicating that all signals are stable.

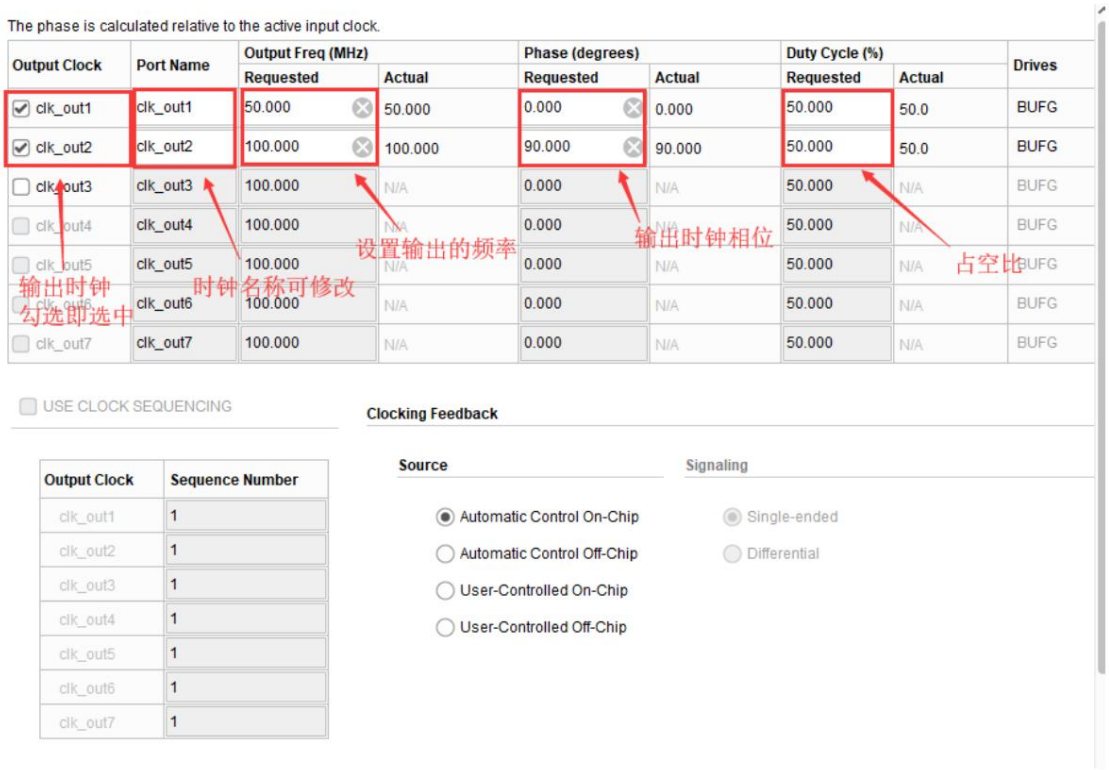


Figure 10- 4

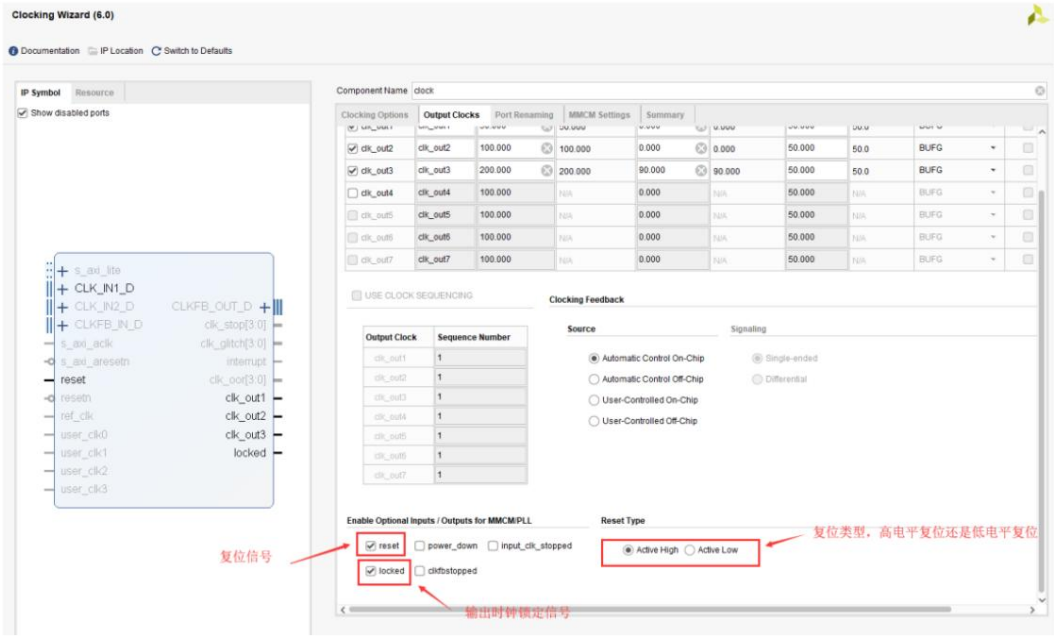


Figure 10- 5

5) Keep the default settings for other options. In the summary, you can view the IP core configuration information. Click OK.

Select generate in the pop-up interface and wait for a while to generate the IP core.

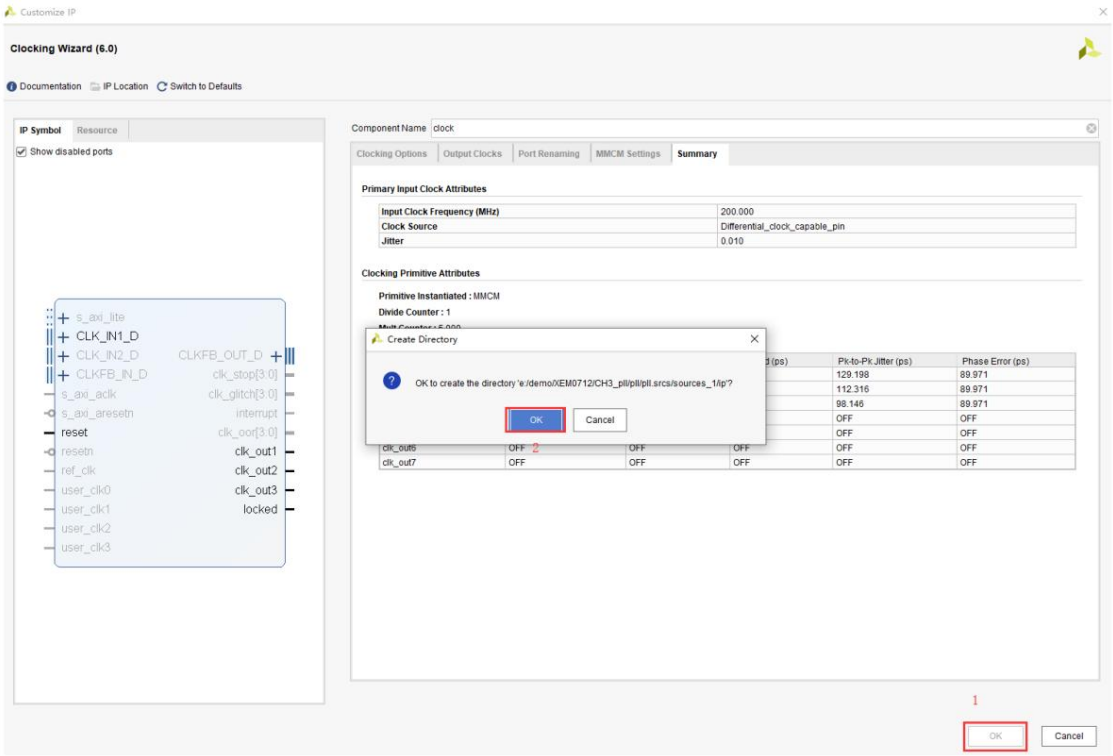


Figure 10- 6

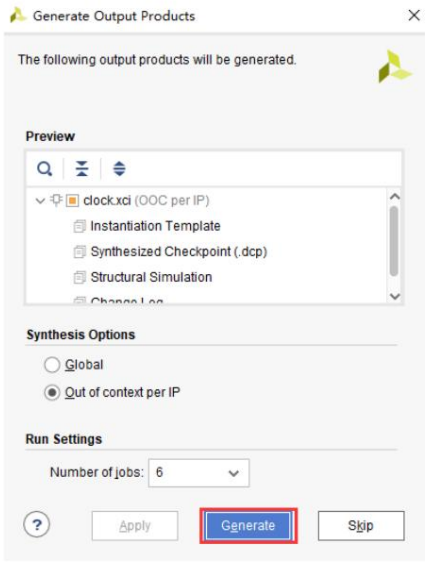


Figure 10- 7

6) After the IP is generated, you can see the generated IP in the resource box

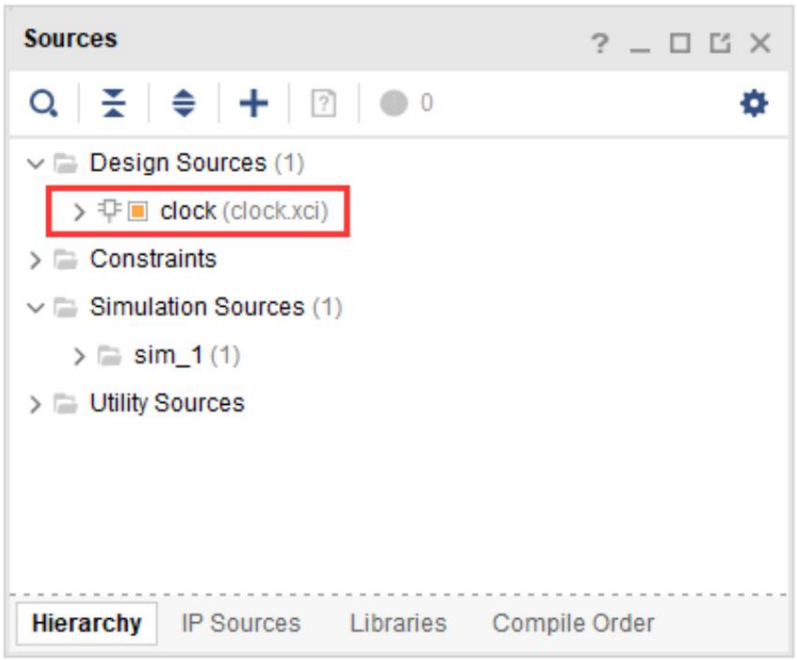


Figure 10- 8

8.4 Programming and calling **IP** core

After the IP core is generated, it needs to be called in other files to play the role of IP.

We will first simulate the IP core, and then use the clock generated by the IP core to drive the LED light.

1) Add a pll_led.v file and write a program. In this experiment, by calling the generated IP core,

To drive the LED light, two identical counters are used in the program. The maximum values of the three counters are the same.

The two counters count the two different clocks generated.

The two LED lights are on and off at different times, proving that two clocks with different frequencies are actually generated.

2) The programming process of this experiment involves the instantiation of IP cores. The instantiation of IP cores can be done using Vivado

Provided IP core instantiation template. Open the IP source under the source window and find the clock instantiation.

Double-click clock.veo under Template to open it and you will see the instantiation template of clock IP. Assign values to the template and add

Just go to the design program.

```
70  //------------ Begin Cut here for INSTANTIATION Template ---// INST_TAG
71
72    clock instance_name
73     (
74      // Clock out ports
75      .clk_out1(clk_out1),     // output clk_out1
76      .clk_out2(clk_out2),     // output clk_out2
77      // Status and control signals
78      .reset(reset), // input reset
79      .locked(locked),       // output locked
80     // Clock in ports
81      .clk_in1(clk_in1));      // input clk_in1
82  // INST_TAG_END ------ End INSTANTIATION Template ---------
83
```

Figure 10- 9

```
1 module pll_led(
2          input wire                          clk                 ,
3          input wire                          rst_n               ,
4          output reg                 [1:0]    led
5
6 );
7
8 //================================================
9 //parameter define
10 //================================================
11 parameter CNT_MAX                      = 50_000_000 - 1;
12
13 //================================================
```

```
//internal signals
//================================================
reg         [27:0] timer1;//counter for clk_out1
reg         [27:0] timer2;//counter for clk_out2
wire               locked;
wire               clk_out1;
wire               clk_out2;


//--------------------timer1--------------------
always @(posedge clk_out1 or negedge rst_n)begin
    if(rst_n==1'b0)begin
        timer1 <= 'd0;
    end
    else if(locked==1'b1)begin
        if(timer1==CNT_MAX)
            timer1 <= 'd0;
        else
            timer1 <= timer1 + 1'b1;
    end
    else begin
        timer1 <='d0;
    end
end

//--------------------timer2--------------------
always @(posedge clk_out2 or negedge rst_n)begin
    if(rst_n==1'b0)begin
        timer2 <= 'd0;
    end
    else if(locked==1'b1)begin
        if(timer2==CNT_MAX)
            timer2 <= 'd0;
        else
            timer2 <= timer2 + 1'b1;
    end
    else begin
        timer2 <='d0;
    end
end

```

```
55
56 //--------------------led1--------------------
57 always @(posedge clk_out1 or negedge rst_n)begin
58         if(rst_n==1'b0)begin
59                 led[0] <= 1'b1;
60         end
61         else if(timer1 == CNT_MAX)begin
62                 led[0] <= ~led[0];
63         end
64         else begin
65                 led[0] <= led[0];
66         end
67 end
68
69 //--------------------led2--------------------
70 always @(posedge clk_out2 or negedge rst_n)begin
71         if(rst_n==1'b0)begin
72                 led[1] <= 1'b1;
73         end
74         else if(timer2 == CNT_MAX)begin
75                 led[1] <= ~led[1];
76         end
77         else begin
78                 led[1] <= led[1];
79         end
80 end
81
82
83 clock clock_inst
84         (
85         // Clock out ports
86                                                 // output clk_out1
87                                                 // output clk_out2
88         .clk_out1(clk_out1), .clk_out2(clk_out2), // Status and control signals
89         .reset(~rst_n), // input reset
90         .locked(locked), // output locked
91        // Clock in ports
92         .clk_in1(clk)               // input clk_in1
93 );
94
95 endmodule
```

8.5 Simulate the **IP** core and top layer

When simulating a design, you need to first write a simulation file, which is also a A .v file is also written in Verilog.

1) First, click Add Source in the navigation bar and select Add Simulation Sources in the pop-up interface.

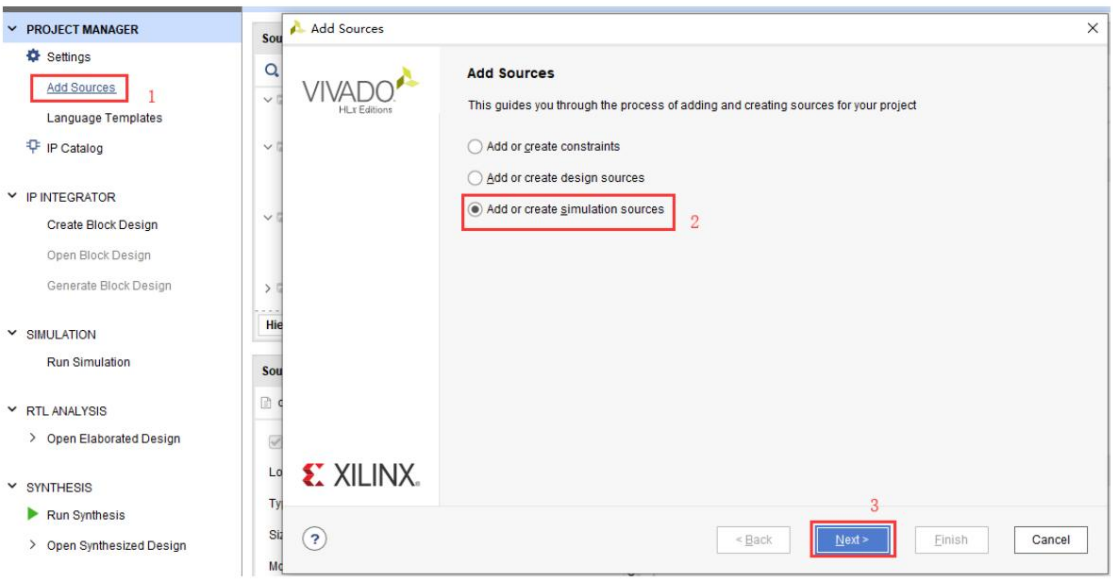


Figure 10- 10

2) Select create file, name the file tb_pll_led and click OK.

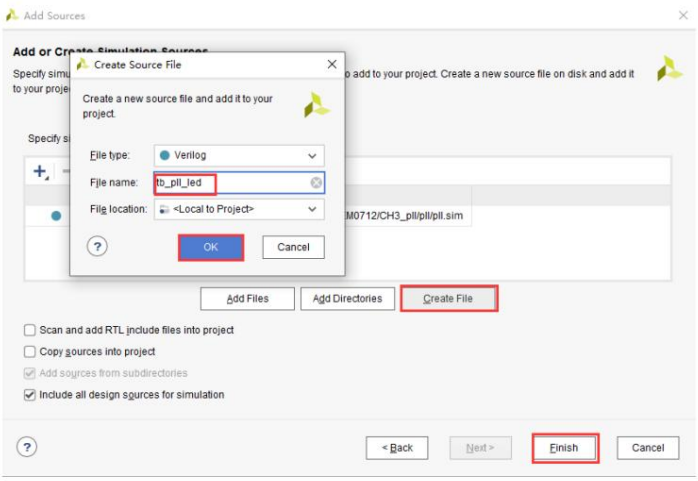


Figure 10-11

3) After adding the file, you can view the layers of the simulation file in the simulation source in the source column.

Substructure

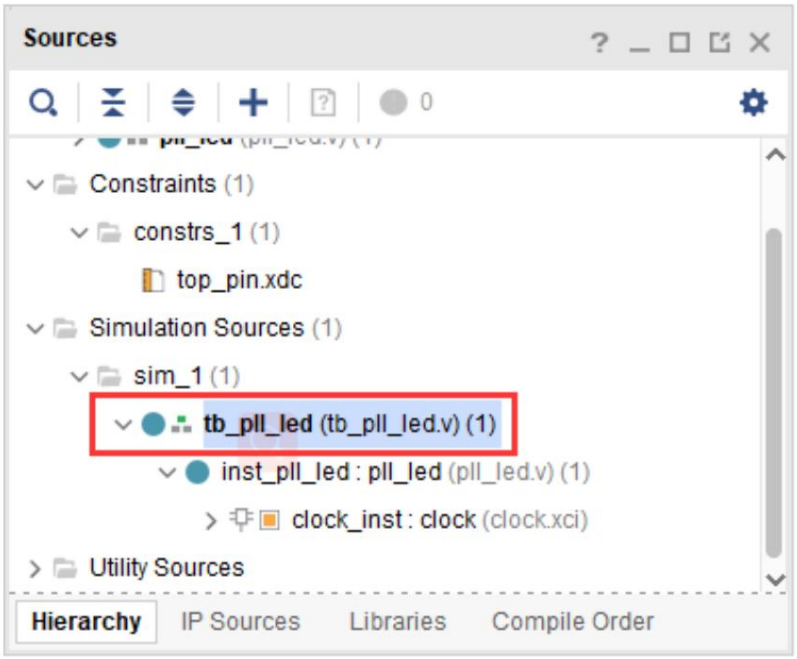


Figure 10-12

4) Next, we will introduce the writing of simulation files.

First, we need the simulation step size and simulation accuracy, that is, the first line of `timescale 1ns/1ps, where 1ns refers to

Simulation step size, 1ps refers to the simulation accuracy.

Then comes the definition of the simulation module, which is similar to the module definition of the design file.

Then comes the definition of the signal type. The variable of reg type is the stimulus signal, which is input from the outside or from the current module.

Generated inside the block. Wire type variables are used to observe output signals or signal connections between modules.

The next step is to instantiate the module to be simulated and instantiate the module we designed earlier into the simulation file.

The last step is to write the stimulus signal. Through the stimulus signal, we can simulate the operation we want to perform on the module.

To verify whether our design is correct and debug the design.

It is worth mentioning that using parametric design during simulation can speed up simulation and improve simulation efficiency.

For example, in this experiment, you can modify the CNT_MAX parameter to change the counter value.

The maximum value can reduce the simulation time.

```
`timescale 1ns/1ps

module tb_pll_led (); /* this is automatically generated */

    parameter CNT_MAX = 100_000 - 1;

    reg         clk;
    wire [1:0] led;
    reg         rst_n;

    pll_led #(
            .CNT_MAX(CNT_MAX)
        ) inst_pll_led (
            .clk        (clk),
            .rst_n      (rst_n),
            .led        (led)
        );

    initial begin
        clk = 0;
        forever #(10) clk = ~clk;
    end

    initial begin
        rst_n = 0;
        #500;
        rst_n = 1;
    end
endmodule
```

5) Simulation preparation

Here, the simulation files are simulated using Vivado's own simulator and a third-party simulator.

Users can choose the simulation method according to their preferences.

First, let's introduce the simulator that comes with Vivado. Vivado is a full-featured design platform that includes

The simulation platform is included, and we can directly see the simulation option from the navigation bar.

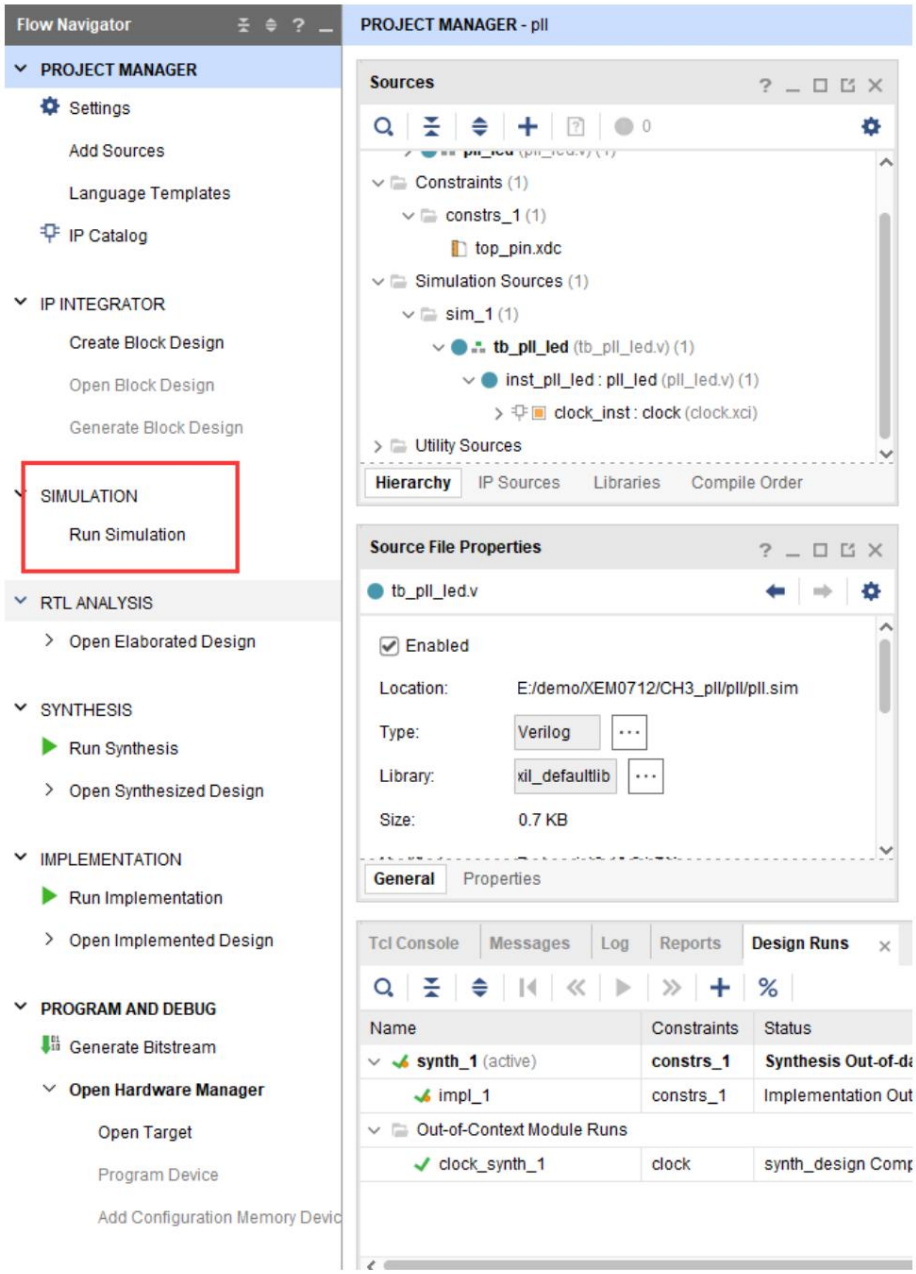


Figure 10- 13

We can directly click run simulation to simulate, select behavioral simulation here, wait for a while

After a while, Vivado will pop up the simulation interface.

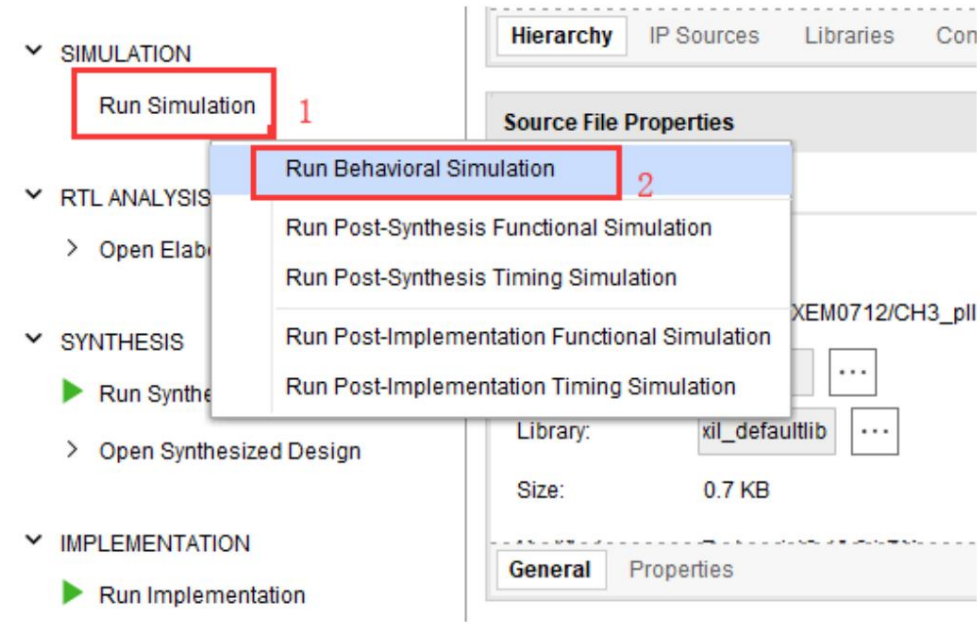


Figure 10- 14

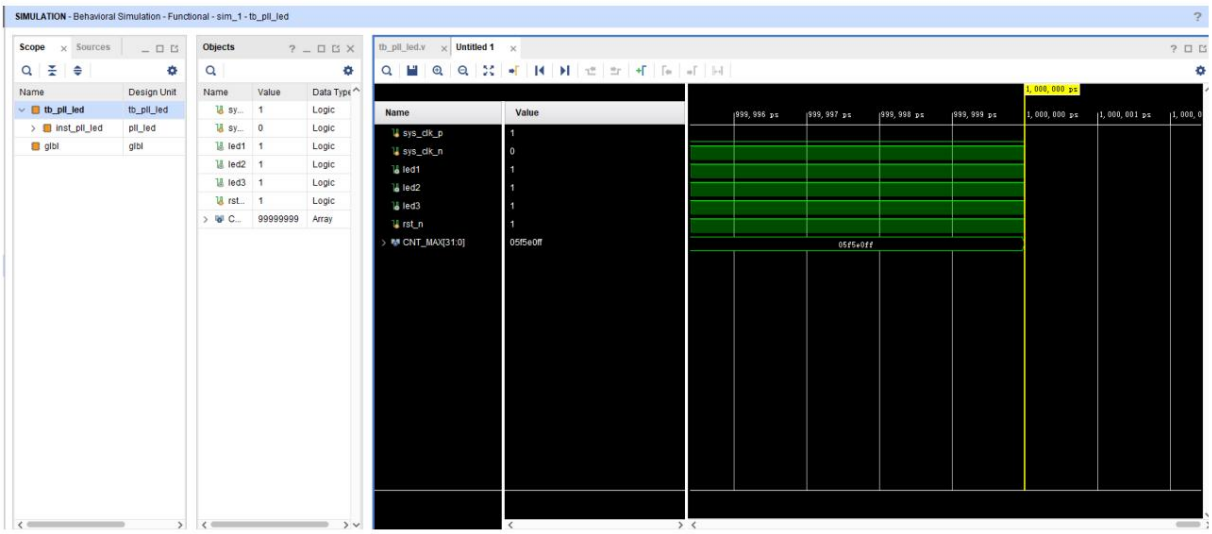


Figure 10- 15

We add other signals in the module to the waveform window, right-click inst_pll_led, and select Add to Wave window

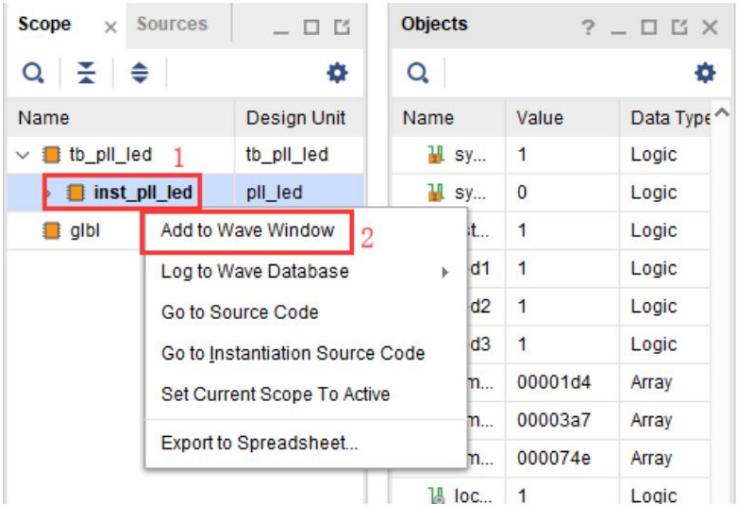


Figure 10- 16

Through the waveform window, we can observe the difference between clk_out1, clk_out2, clk_out3 and the input.

The relationship between the divided clock sys_clk. As can be seen from the waveform, clk_out1 is a 50MHz clock.

Clock, clk_out2 is a 100MHz clock

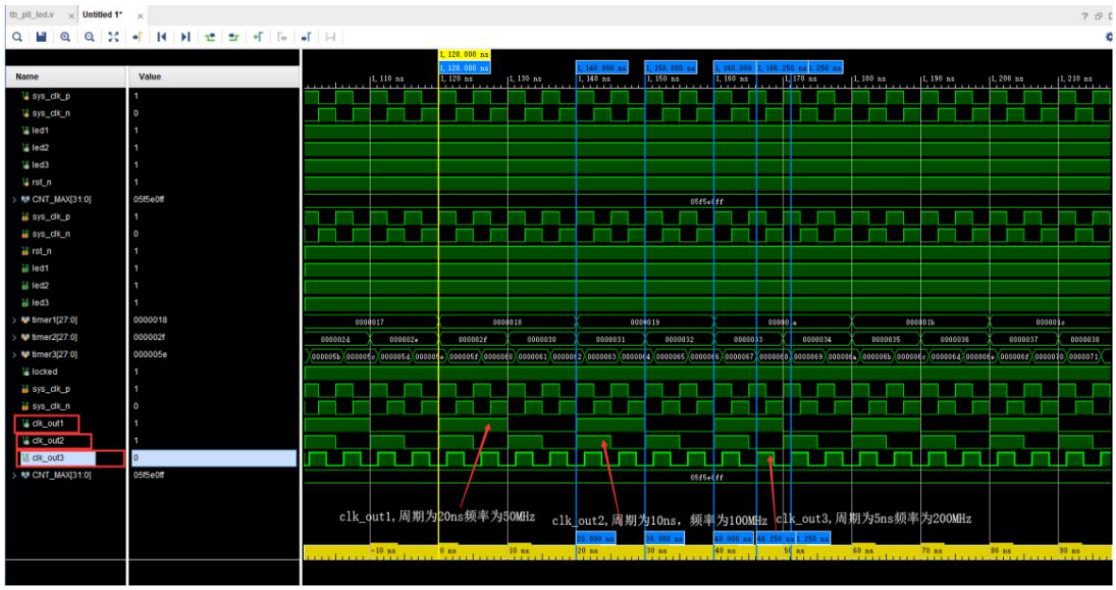


Figure 10- 17

6) Next, we will introduce how to use the third-party simulation software modelsim to simulate the design.

Click tools -> settings

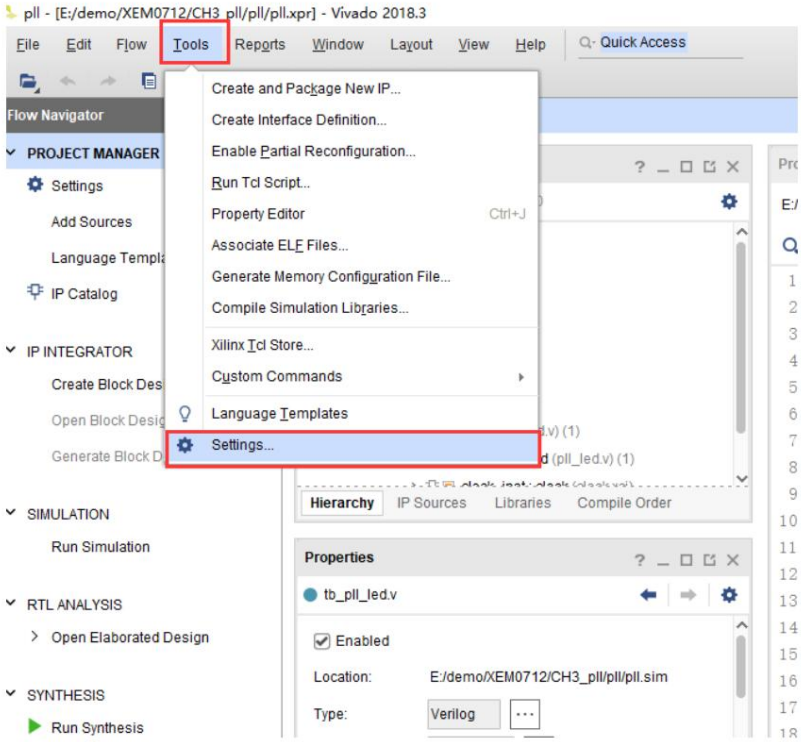


Figure 10- 18

Select Midelsim simulator in the simulation options

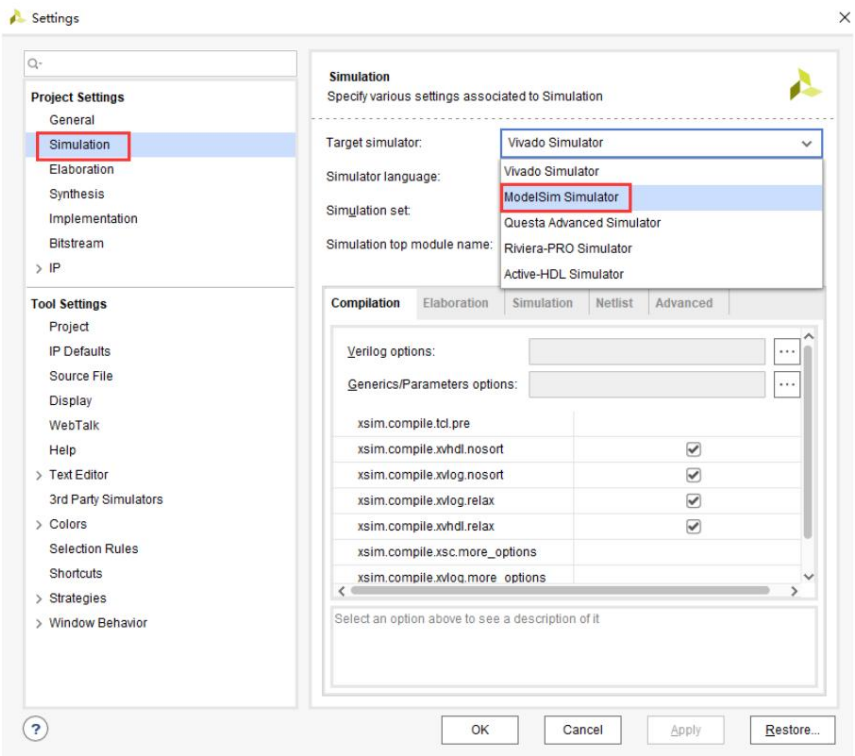


Figure 10- 19

Select yes in the pop-up interface

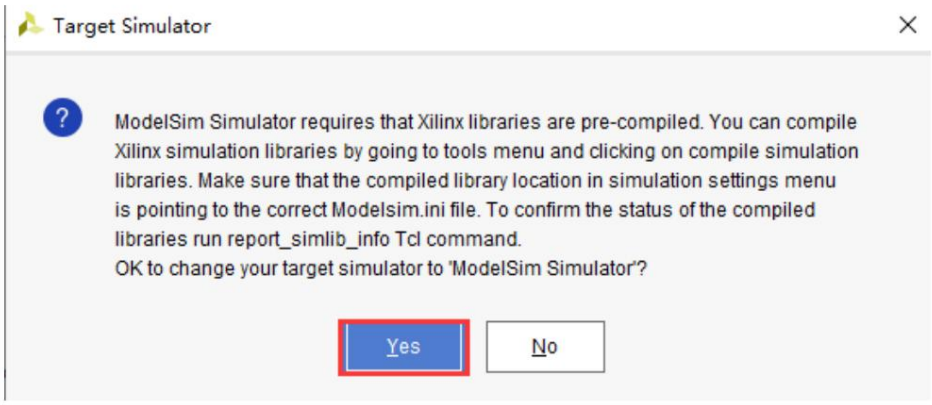


Figure 10- 20

Then click on the third-party simulation tool and select the installation path after Modelsim. Here is the user installation

The location of the modelsim software, pay attention to select the win64 directory. Then select the directory where the library files are stored.

Directory, users can create a library file directory to store library files in the simulation process.

Click Apply -> OK.

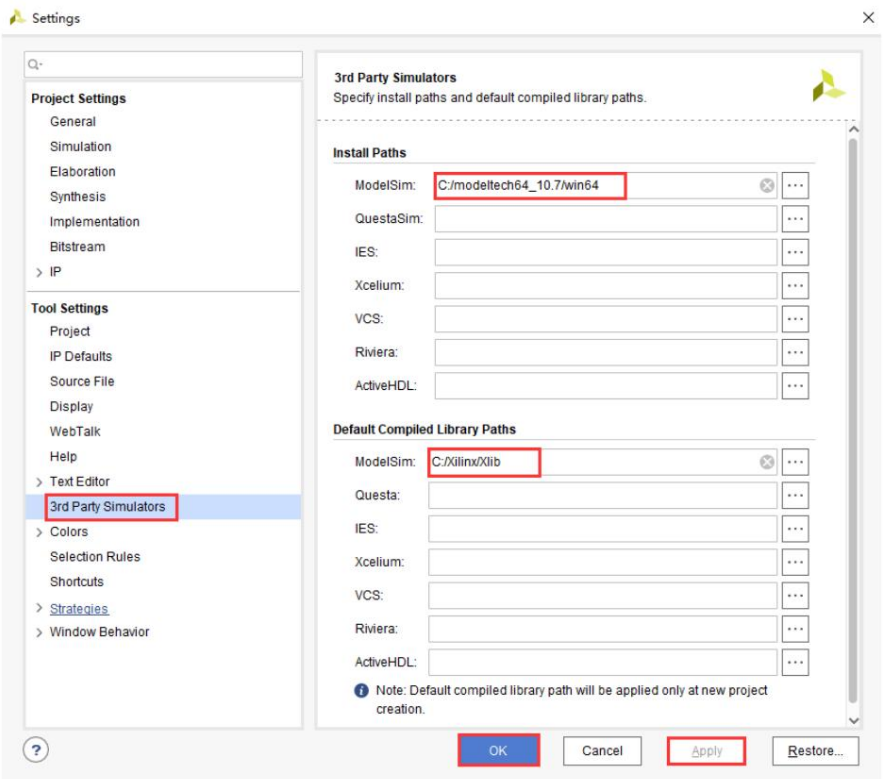


Figure 10- 21

After setting, click run simulation to open the modelsim software for simulation.

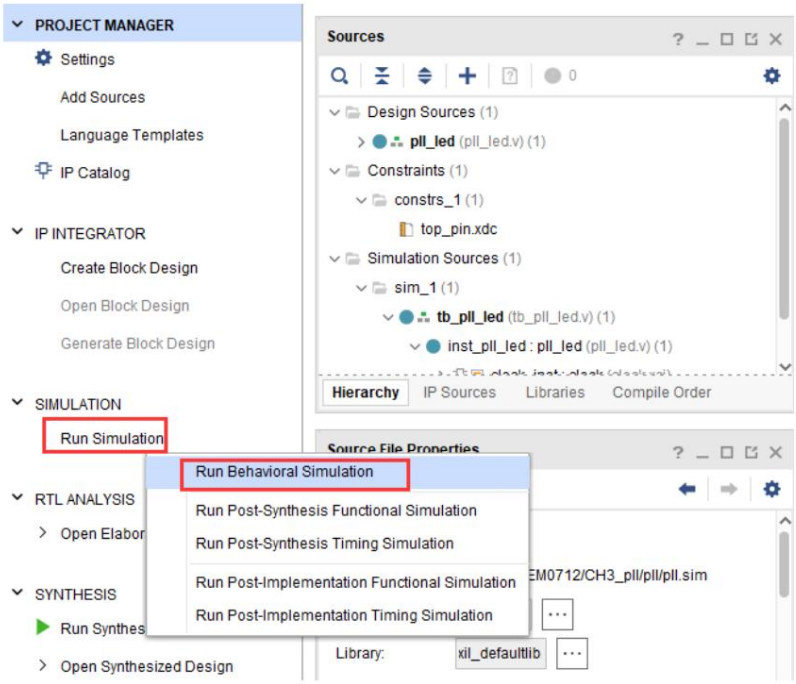


Figure 10- 22

After opening modelsim, the interface is as shown in the figure. The red part can view various files and compile files.

, and add the signals you want to observe.

The purple part is the printed script information. Users can also use the command line to operate ModelSim.

to realize simulation control.

The yellow part is the signal in the module and its status.

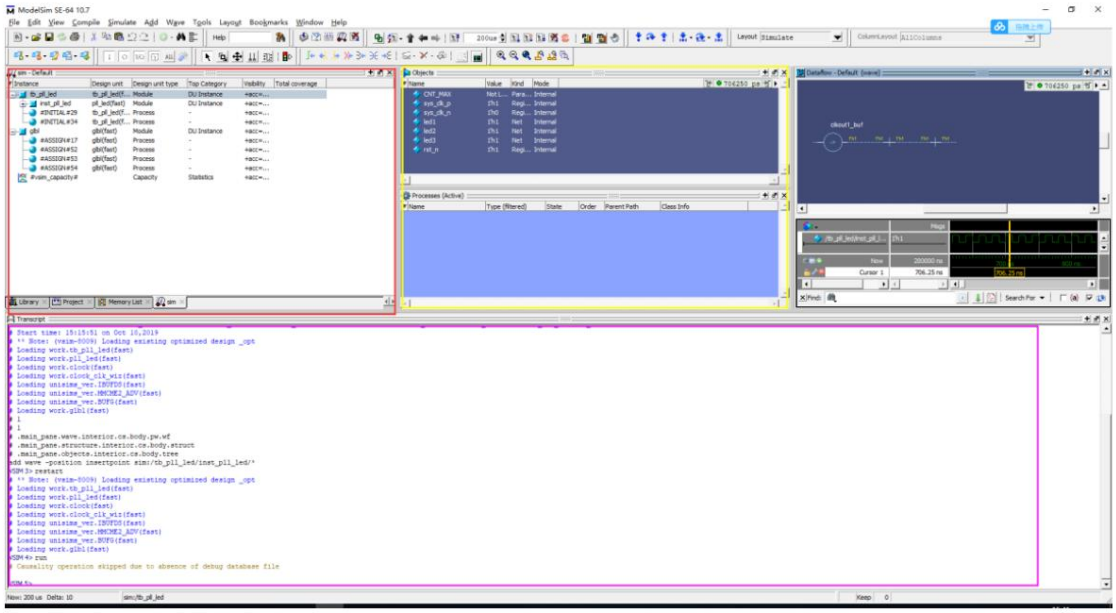


Figure 10- 23

Next, select the signal you want to observe and add it to the waveform window. In the red area, click the

For a signal, right-click -> add wave or use the shortcut keys Ctrl + w to add the waveform to the waveform window.

After entering the waveform window, you can use the shortcut keys Ctrl+A to select all signals, and Ctrl+G to select all signals.

Through this operation, the signals in different modules can be grouped for easy observation.

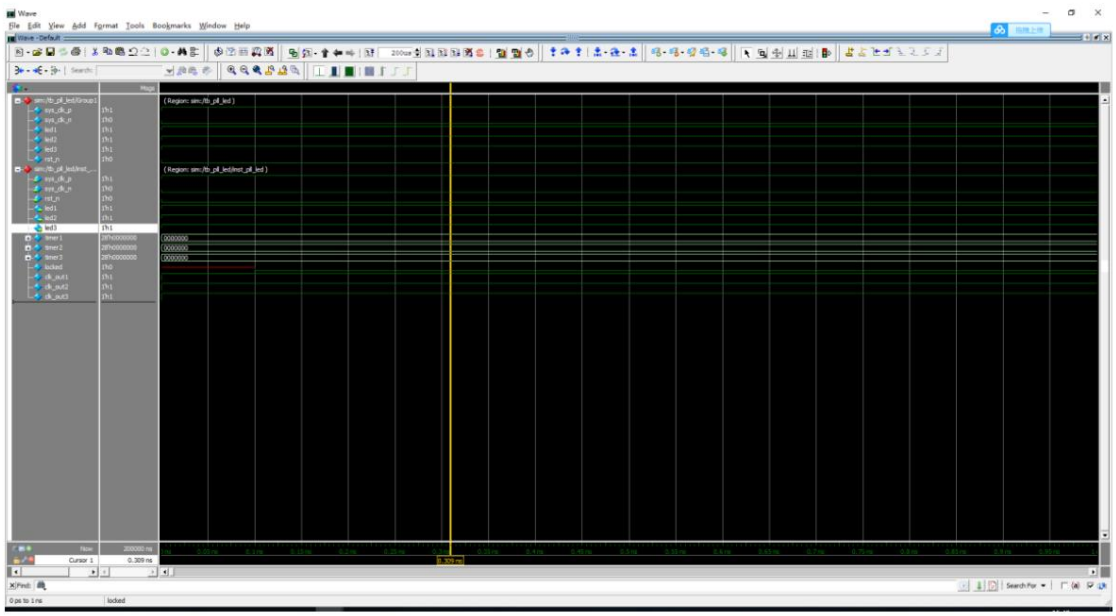


Figure 10- 24

It can be seen that the simulation time is too short and we cannot see the waveform we want. We can adjust

Simulation time, get more waveform information. Change the simulation time in the menu bar, and then click Run.

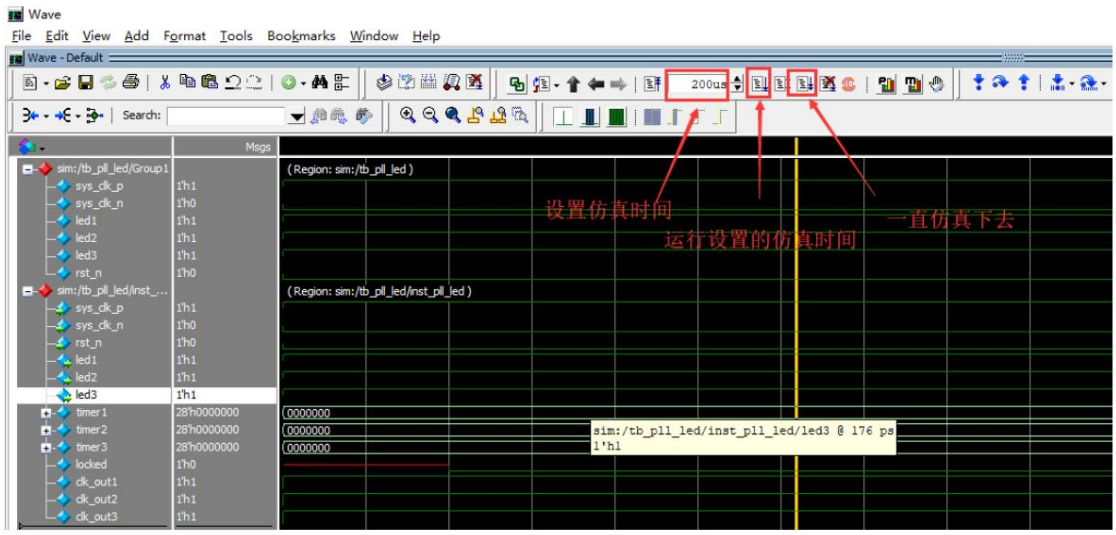


Figure 10- 25

In the menu bar, select the magnifying glass icon to view all waveform information.

Roll the mouse wheel to zoom in or out on a specific area, making it easier to observe the signal.

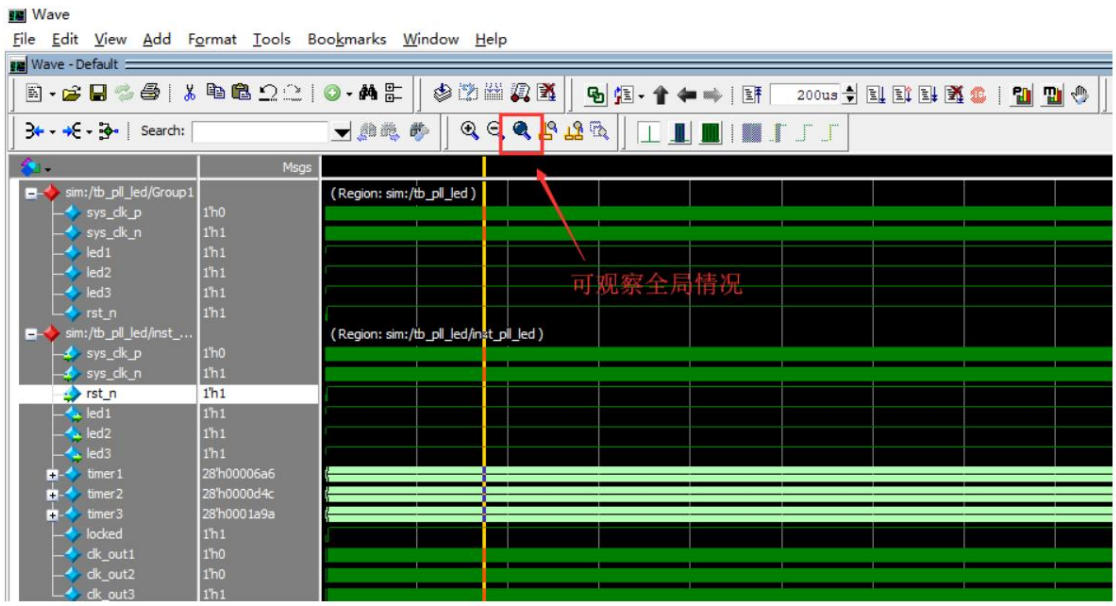


Figure 10- 26

Next, we focus on the relationship between the two clock signals output by the IP core and the input system clock.

As shown in the following figure:

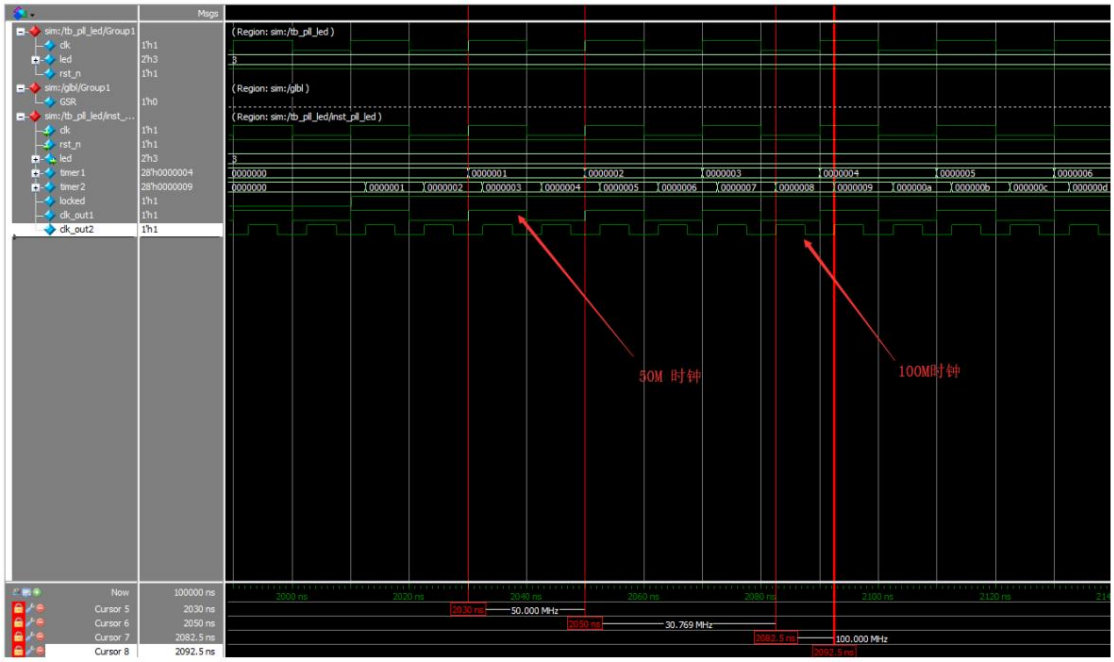


Figure 10- 27

By adding cursors, you can determine the period of each output clock. You can see that clk_out1,

The periods of clk_out2 are 20ns and 10ns respectively. This means that the frequencies of clk_out1 and clk_out2 are

50M, 100M. In addition, it can be found that clk_out2 is not aligned with the input clock, but is offset.

90°, which is the same as what we set when configuring the IP core, and is offset by 90° relative to the system clock.

7) At this point, the simulation of the test module is completed

8.6 Add pin constraints

1) First add a constraint file and click Add Source -> Add or Create Constraints –> ok;

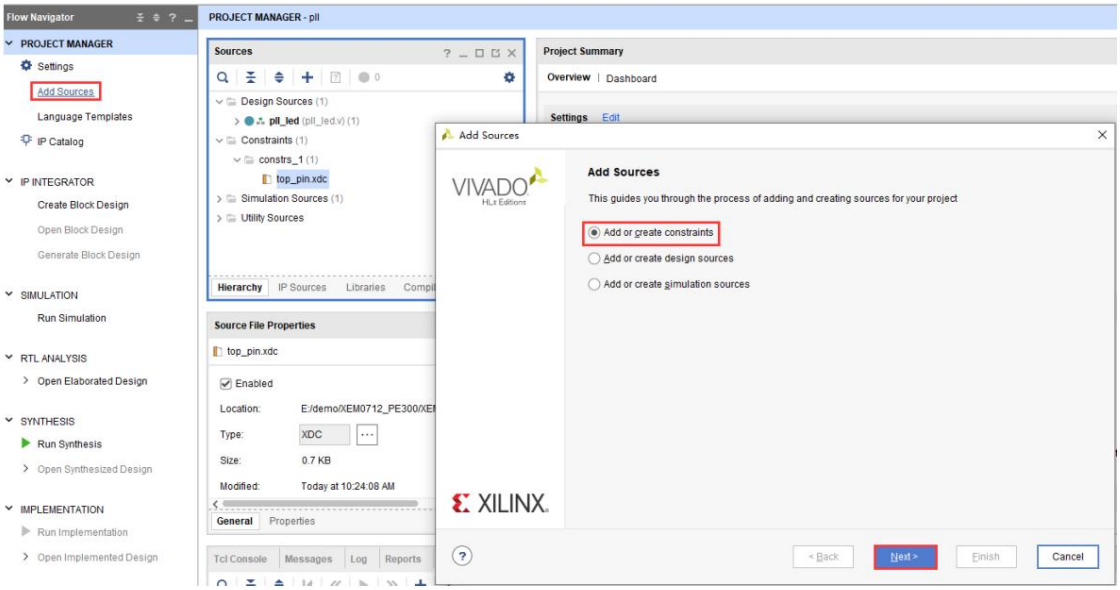


Figure 10- 28

2) Add a .xdc file named top_pind

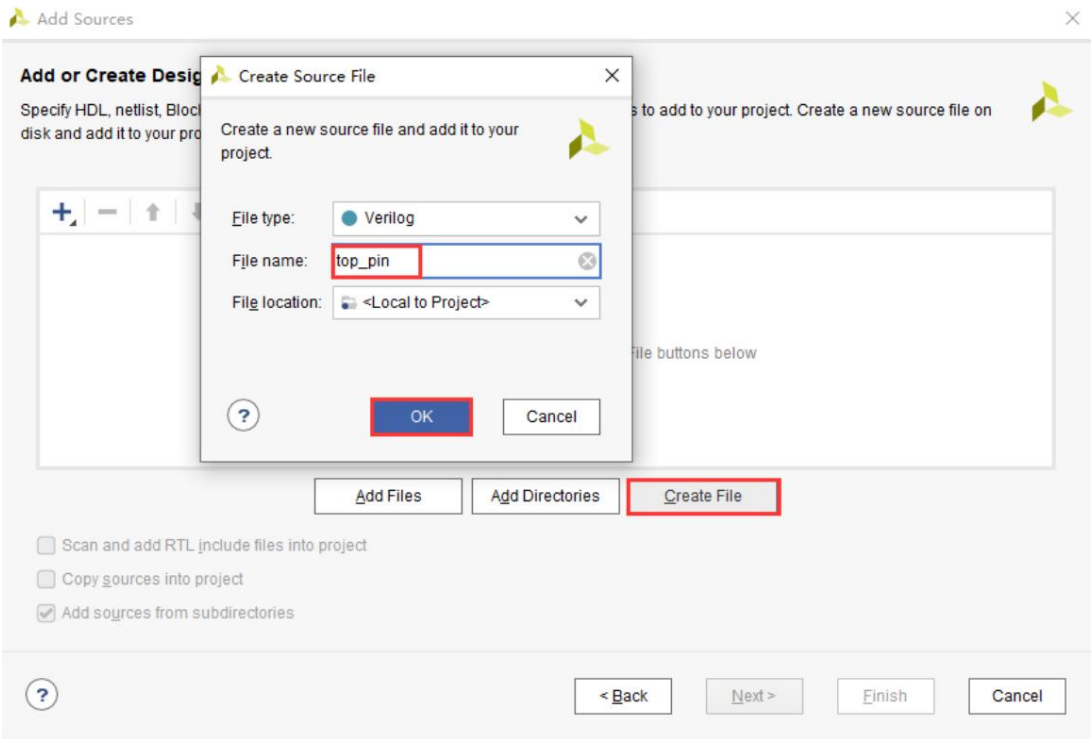


Figure 10- 29

3) Write the constraint file. The constraint file is basically the same as the previous experiments, so I will not explain it in detail here.

```
############### clock define###################
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
############### key define###################
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]

###############LED define###################

set_property PACKAGE_PIN P15 [get_ports {led[0]}]
set_property PACKAGE_PIN U12 [get_ports {led[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports {led[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
```

8.7 Download Verification

1) Next, we will verify the final results of this experiment on the development board. Click Generate Bit File and wait for the bit file to be

After the software is generated, connect the development board to the downloader.

2) Download the bit file to the development board and start observing the phenomenon

3) You can see that the two LEDs are flashing at different frequencies. And the flashing between the two lights can just form

It has become a two-level system plus one.

8.8 Experimental Summary

Through this experiment, we have a preliminary understanding of the steps of using IP cores for development under Vivado.

This step is used when developing FPGA programs using IP cores.

Other IP cores will also be developed using this method.

In addition, we also learned about the simulation process in FPGA development. It can be said that FPGA

The most important step in the development process is to quickly locate the problems in our design through simulation.

So here I hope everyone will pay attention to simulation.

We provide two simulation methods. One is to use the simulation tool provided by Vivado.

One is to use modelsim for simulation. Users can use the simulation tools they are used to.

We recommend using modelsim for simulation, which is a professional simulation tool in the industry.

Compared with the simulation tools that come with Vivado, it still has certain advantages.

Chapter 9 On-Chip Resources: **RAM**

9.1 Experimental Background

When it comes to FPGA, we have to mention memory. When we do related projects, we often encounter storage data. When the amount of data is too large, we can store it in the peripheral memory of the FPGA chip, such as sdram, ddr sdram, ddr3 sdram, etc. However, accessing peripheral memory is relatively troublesome, so when the amount of data is small, We can directly use the IP core of the RAM inside the FPGA chip.

RAM is the abbreviation of random access memory, which means random access memory. RAM can be used as needed. We can generate a RAM by calling the IP core inside the FPGA and write a random read/write function in Verilog. HDL code controls this ram.

This chapter familiarizes you with the generation and related settings of RAM IP, and writes a program to operate two simple dual-port RAMs to perform ping-pong operations. and simulate it.

9.2 Create a project

Double-click the Vivado 2018.3 icon on the desktop to open the development environment and follow the steps in the table below.

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is ram_pp, the project path is E:\demo\Z7-Lite\ram_pp Check Createproject subdirectoryÿNext
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device, purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

9.3 Generate **IP** core and parameter configuration

This experiment configures a simple dual-port RAM. The so-called dual-port RAM is a RAM that can both read and write.

Write RAM. In the write port, give the data you want to write and the address you want to write.

In this case, data can be written to RAM at the rising edge of the write clock; at the read port, the data to be read is given

The address obtained can be read from RAM if read enable is valid.

To configure the RAM IP core.

Click **IP Catalog,** enter **ram** in **the Search** field on the right, and then click **Memories & Storage Elements.**

You can see that there are two entries created with **RAM**

One is Distributed Memory Generator and the other is Block Memory Generator.

The main difference is that the generated Core occupies different FPGA resources, from Distributed Memory

The resources occupied by the ROM/RAM Core generated by the Generator are LUTs (lookup tables, which are essentially small

RAM); The resources occupied by the ROM/RAM Core generated from the Block Memory Generator are Block

Memory (embedded hardware RAM). Users can select the entry to create based on actual resource usage requirements.

Explain the creation process of Block RAM IP. Double-click Block Memory Generator to pop up the configuration interface.

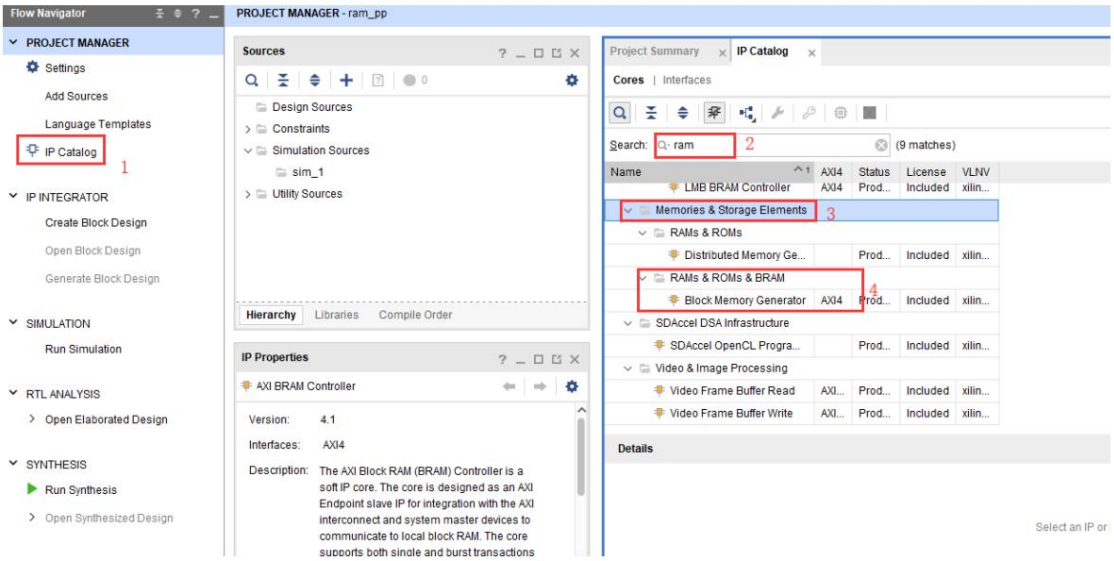


Figure 11- 1

1) Basic RAM configuration

In this configuration interface, you can modify the name of the generated IP and select the interface type.

Select Native type, and select the RAM type as Simple Dual Port RAM;

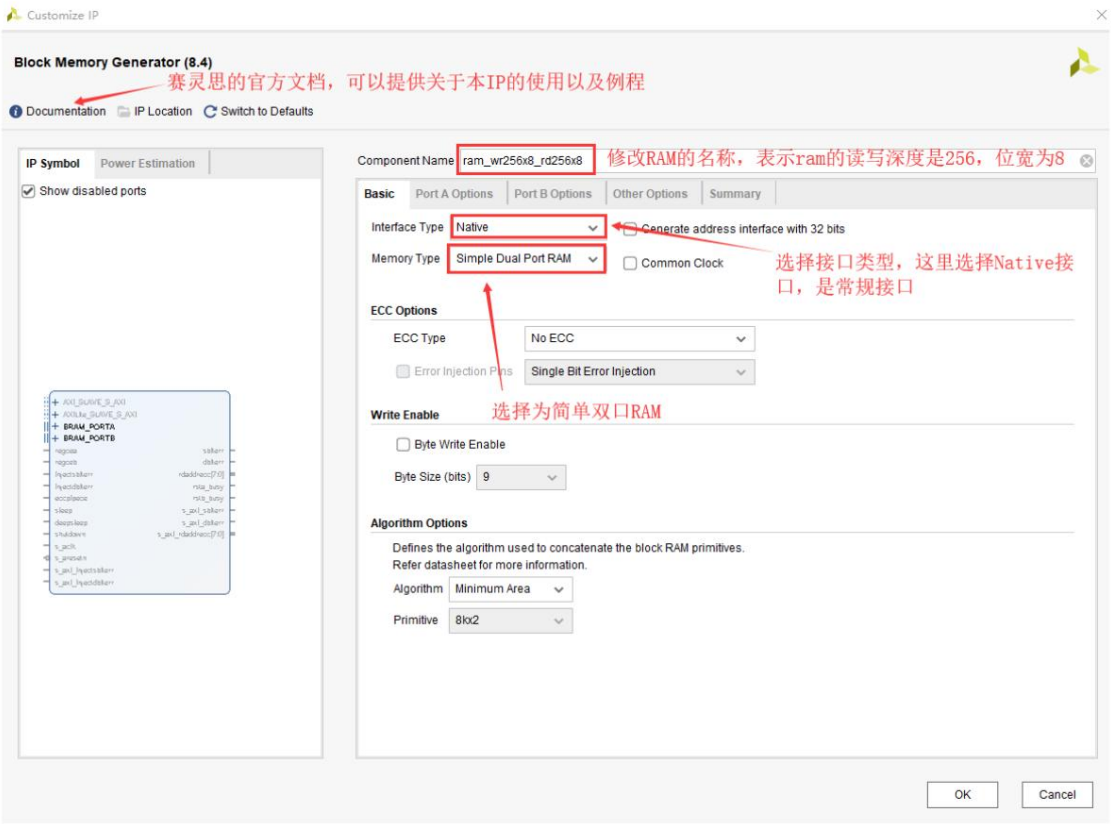


Figure 11- 2

2) Next, configure the input port

At the input port, you can choose to configure the bit width of the input port, and you can choose to configure the depth of the input port.

In this experiment, the bit width of the input port is selected as 8, the input depth is 256, Enable Port Type:

use ENA Pin (use enable control signal).

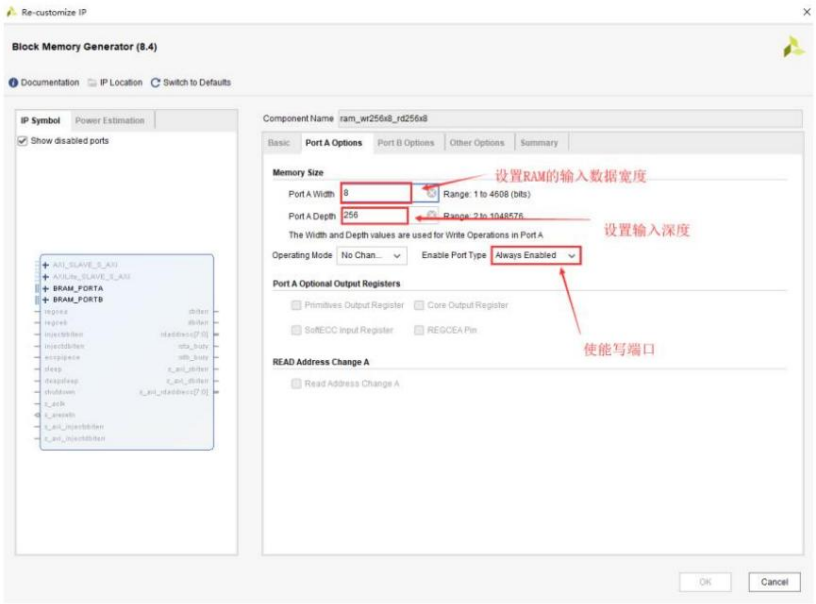


Figure 11- 3

3) Next is the output port configuration

The output port can still select the output bit width, which is kept at 8 bits here, and the depth of the output port

It is determined by the input depth, input bit width and output bit width.

No register is added to output port B. Using a register will cause an extra delay in the output data. Is this

The choice should be made according to the specific design and is not selected here.

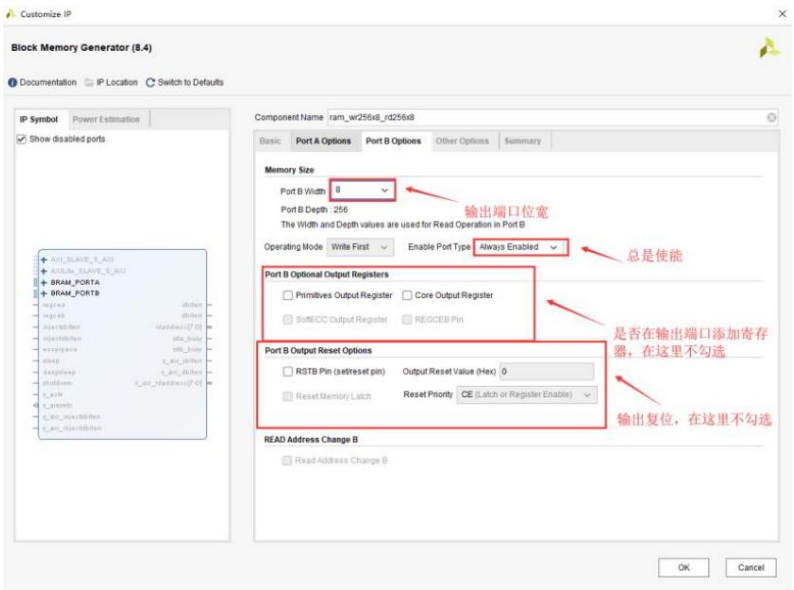


Figure 11- 4

4) In other options, you can choose to import the file to initialize RAM, but it will not be imported in this experiment.

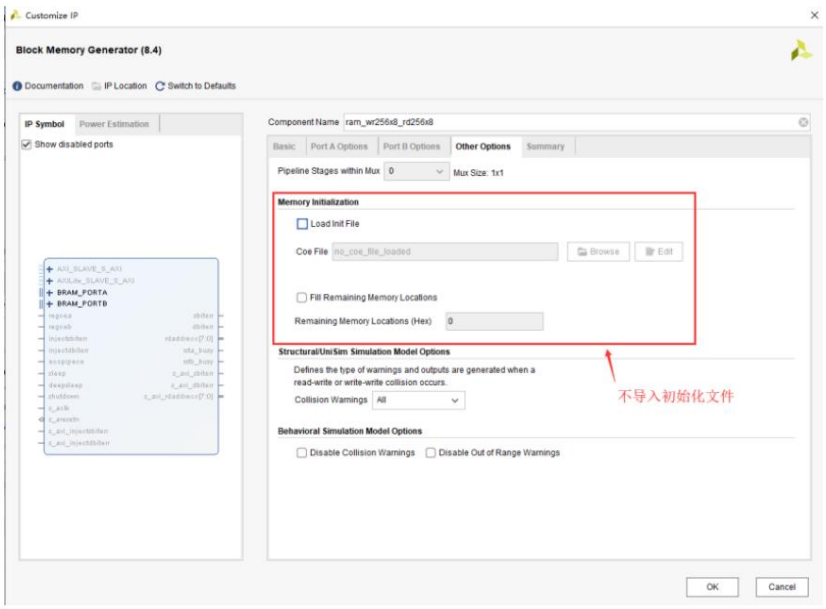


Figure 11- 5

5) In the final Summary, you can view detailed information about the configuration

Note that in this configuration, the data read from port B has a latency of 1 clock, i.e.

Delay in clock cycles.

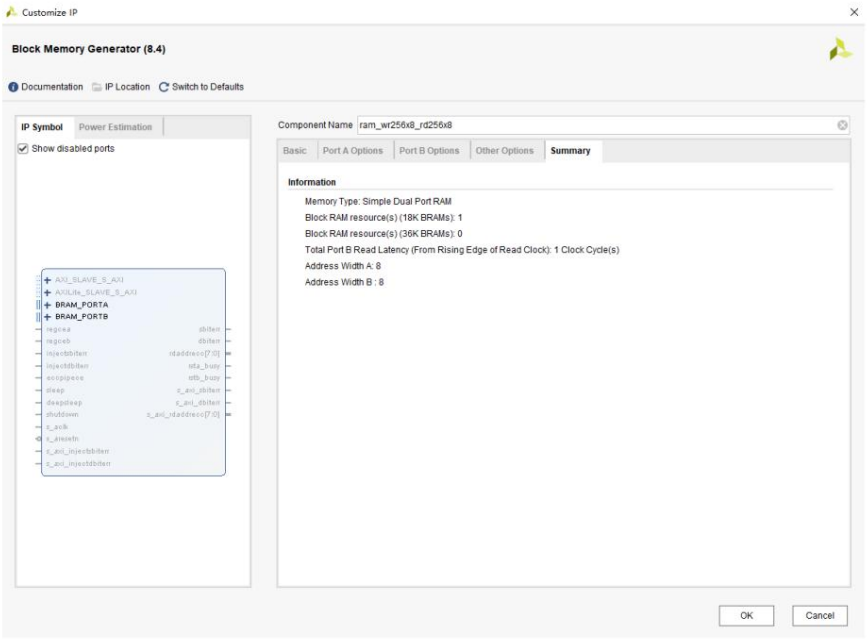


Figure 11- 6

6) Click OK to complete the configuration of the RAM IP.

9.4 Programming and **IP** Core Calling

This experiment requires the use of the generated IP to instantiate two RAMs to complete the ping-pong operation .

We call the alternating read and write operation ping-pong operation. To make the operation more obvious, we draw the following timing diagram:

This helps us understand the operation sequence of this experiment. The drawing is provided in the img directory under the project directory.

Clear graphics, can be opened with Visio.

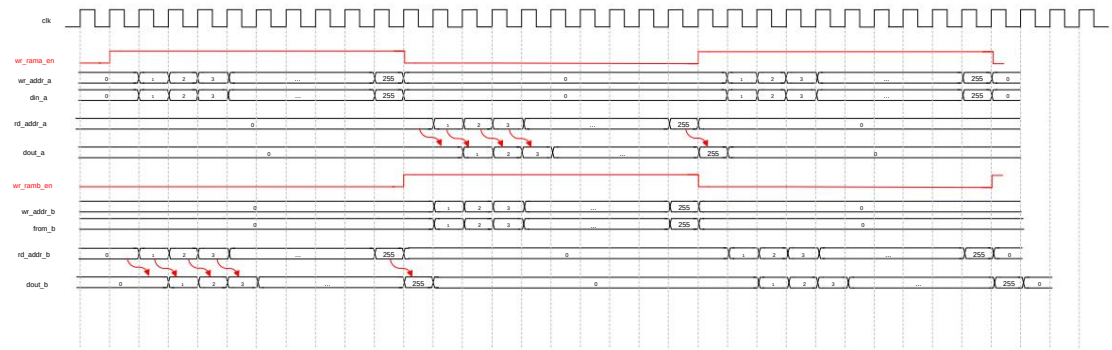


Figure 11- 7

In this experiment, two indication signals wr_rama_en and wr_ramb_en are used to indicate the current

Which RAM to read and write.

When wr_rama_en is high and wr_ramb_en is low, write operation is performed on ram_a, and the

Write data from 0 to 255 in sequence from address 0 to address 255. At the same time, read ram_b

The data in ram_b is read from address 0 to address 255 in sequence. According to the settings of the IP core, the read

The data output has a one-beat delay compared to the read address.

When wr_rama_en is low and wr_ramb_en is high, ram_a is read.

The data is read from address 0 to address 255 of ram_a. It can be seen that the read data is much faster than the read address.

At the same time, write operation is performed on ram_b, writing 0 to 255 in sequence.

ram_b has address 0 to address 255.

Ping-Pong operation is accomplished by reading and writing ram_a and ram_b at different times.

The program design of this experiment is mainly to realize the timing of the waveform diagram given above.

In development, drawing waveforms to design timing is a common method. This method can

To improve development efficiency, it is recommended to use this method more often.

1) Create a ram_pp.v file

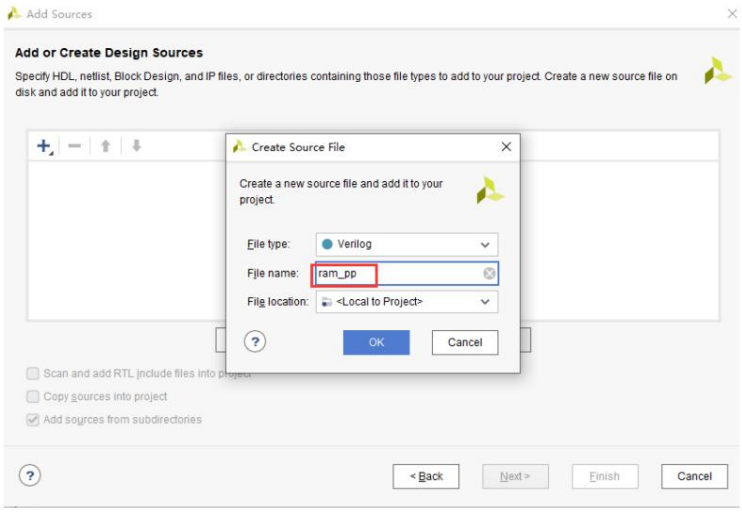


Figure 11- 8

2) Write the experimental code

The specific code is provided in the folder under the demo directory. Let's make a few comments on the key parts of the code.

bright:

Lines 44 to 60 control wr_rama_en

```
44 //--------------------wr_rama_en--------------------
45 always @(posedge clk or negedge rst_n)begin
46         if(rst_n==1'b0)begin
47                 wr_rama_en <= 1'b0;
48         end
49         //When ram_b writes to the last address, wr_rama_en is pulled high
50         else if((wr_addr_b==MAX) && (wr_rama_en==1'b0))begin
51                 wr_rama_en <= 1'b1;
52         end
53         //When ram_a writes the last address, wr_rama_en is pulled low
```

```
54          else if((wr_addr_a==MAX) && (wr_rama_en==1'b1))begin
55                  wr_rama_en <= 1'b0;
56          end
57          else begin
58                  wr_rama_en <= wr_rama_en;
59          end
```

Lines 62 to 65 use combinational logic to control wr_ramb_en

```
62 //------------------wr_ramb_en------------------
63 always @(*)begin
64          wr_ramb_en = ~wr_rama_en;
65 end
```

Lines 68 to 86 generate the address to write to ram_a and the data to write to ram_a;

```
68 //--------------------wr_addr_a--------------------
69 always @(posedge clk or negedge rst_n)begin
70          if(rst_n==1'b0)begin
71                  wr_addr_a <= 'd0;
72          end
73          //When wr_rama_en is high, ram_a write address increases
74          else if(wr_rama_en == 1'b1)begin
75                  if(wr_addr_a == MAX)
76                          wr_addr_a <= 'd0;
77                  else
78                          wr_addr_a <= wr_addr_a + 1'b1;
79          end
80          else begin
81                  wr_addr_a <= 'd0;
82          end
83 end
84
85 //--------------------from_a--------------------
86 assign din_a = wr_addr_a;
```

Lines 89 to 104 generate the address to read ram_a

```
89 //--------------------rd_addr_a--------------------
90 always @(posedge clk or negedge rst_n)begin
91          if(rst_n==1'b0)begin
92                  rd_addr_a <= 'd0;
93          end
94          //When wr_rama_en is low, ram_a read address increases
95          else if(wr_rama_en == 1'b0)begin
96                  if(rd_addr_a == MAX)
97                          rd_addr_a <= 'd0;
98                  else
99                          rd_addr_a <= rd_addr_a + 1'b1;
100         end
101         else begin
102                 rd_addr_a <= 'd0;
103         end
104 end
```

The same operation is used for ram_b, which will not be described here.

Lines 144 to 162 instantiate two RAM IPs, ram_a and ram_b. When instantiating IPs, you can use

Vivado provides instantiation templates. When selecting a template, please note that you should select a file with the suffix .veo.

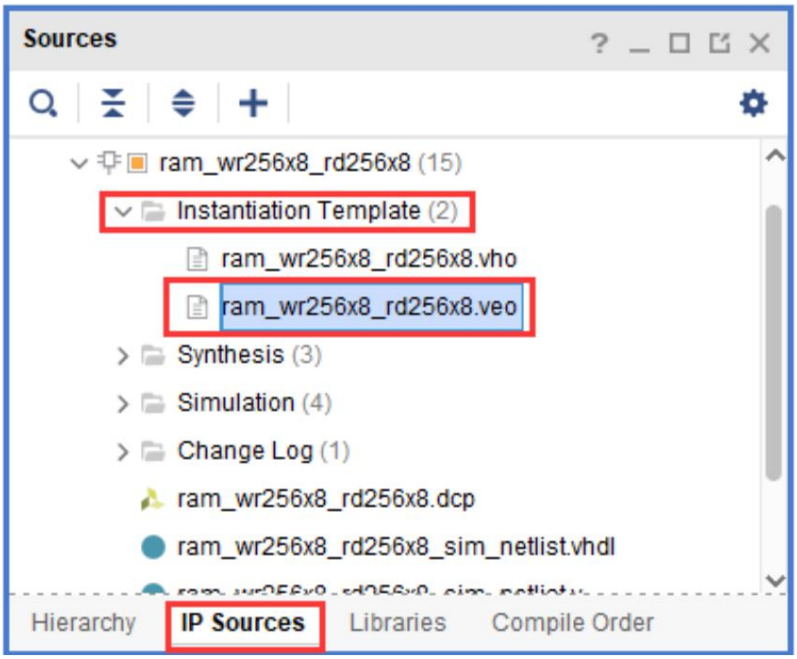


Figure 11- 9

```
144 ram_wr256x8_rd256x8 ram_a (
```

```
.clka (clk ), // input wire clka
.wea (wr_rama_en ), // input wire [0 : 0] wea
.addra(wr_addr_a ), // input wire [7 : 0] addra
.dina (din_a ), // input wire [7 : 0] on
.clkb (clk ), // input wire clkb
.addrb(rd_addr_a ), // input wire [7 : 0] addrb
.doutb(dout_a ) // output wire [7 : 0] doutb
);

ram_wr256x8_rd256x8 ram_b (
.clka (clk ), // input wire clka
.wea (wr_ramb_en ), // input wire [0 : 0] wea
.addra(wr_addr_b ), // input wire [7 : 0] addra
.dina (din_b ), // input wire [7 : 0] on
.clkb (clk ), // input wire clkb
.addrb(rd_addr_b ), // input wire [7 : 0] addrb
.doutb(dout_b ) // output wire [7 : 0] doutb
);
```

Lines 164 to 182 are for outputting the data read from RAM, where wr_rama_dd is wr_rama_en

The purpose is to align the timing with the data read from RAM.

The logical value is assigned to rd_data; when wr_rama_dd is 1, the data in ram_b is read out.

When wr_rama_dd is 0, the data in ram_a is read.

```
//----------wr_rama_dd----------
always @(posedge clk or negedge rst_n)begin
    if(rst_n==1'b0)begin
        wr_rama_dd <= 1'b0;
    end
    else begin
        wr_rama_dd <= wr_rama_en;
    end
end

//----------rd_data----------
always @(*)begin
//The data read out is the data in ram_a
    if(wr_rama_dd == 1'b1)
        rd_data = dout_b;
//The data read out is the data in ram_b
```

```
180         else
181                 rd_data = dout_a;
182 end
```

9.5 Simulation Verification

Next, we use simulation to verify the correctness of the design.

Simulate the design.

1) Add simulation file

Add Source => Add or Create Simulation Sources, add a simulation file named tb_ram_pp

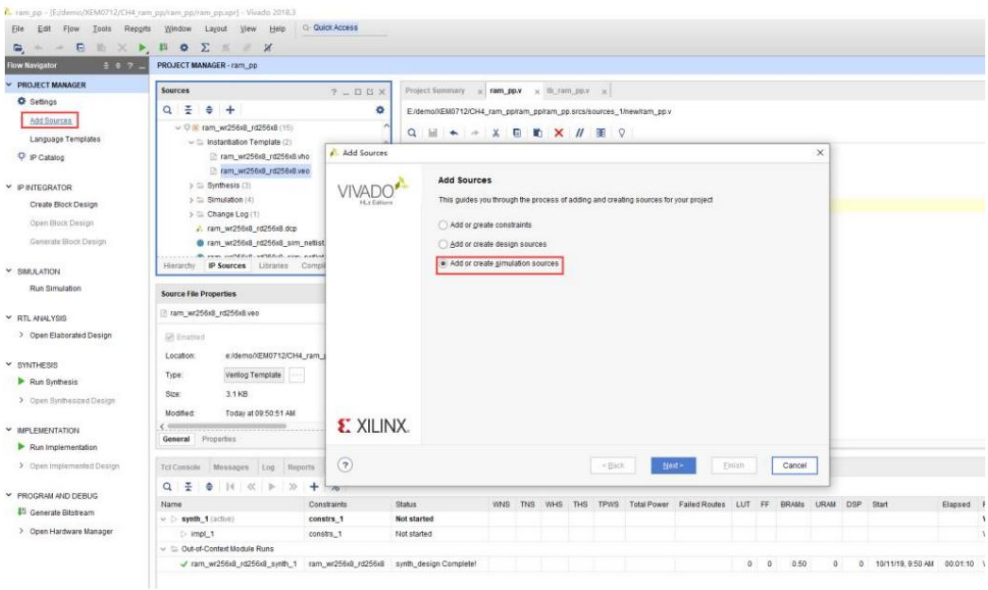


Figure 11- 10

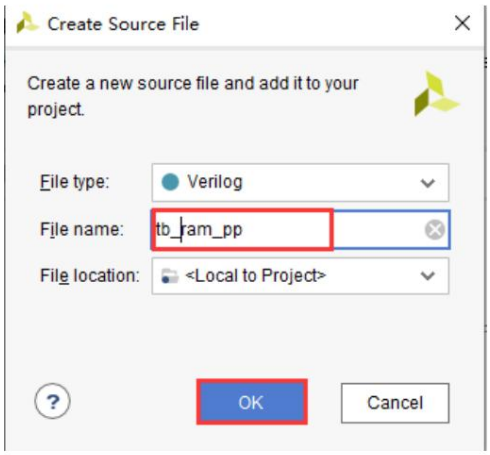


Figure 11- 11

2) Write simulation files

The simulation file can be found in the ram_pp.sim folder under the project directory.

A few points to explain:

Lines 6 to 8 define the stimulus and output signals.

Lines 10 to 16 are the instantiation of the module to be simulated

```
`timescale 1ns/1ps

module tb_ram_pp (); /* this is automatically generated */

    // (*NOTE*) replace reset, clock, others

    parameter MAX = 256 - 1;

    reg           clk ;
    reg           rst_n           ;
    wire [7:0] rd_data ;

    ram_pp
            #( .MAX(MAX)
        ) inst_ram_pp
            ( .clk (clk),
            .rst_n          (rst_n),
            .rd_data (rd_data)
```

```
        );

    // clock
    initial begin
        clk = 1;
        forever #(10) clk = ~clk;
    end

    // reset
    initial begin
        rst_n <= 0;
        #200;
        rst_n <= 1;
    end

endmodule
```

3) Save the simulation file and prepare to start the simulation. Follow the instructions in the previous chapter to close modelsim and Vivado.

Linking methods to associate modelsim and Vivado

Click Tools => Settings

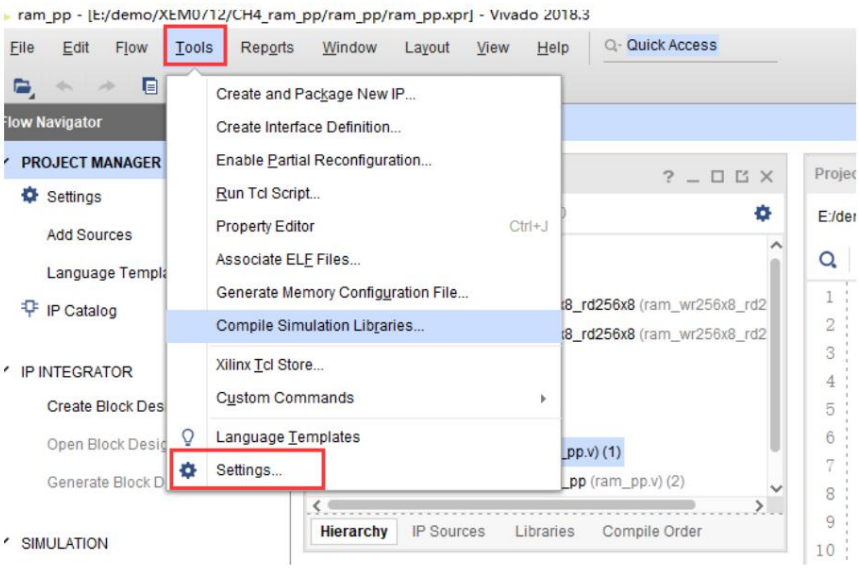


Figure 11- 12

In Project setting, select Simulation option, select Modelsim simulator, in Tools

In setting, select the path where Modelsim is installed and the location of the library files.

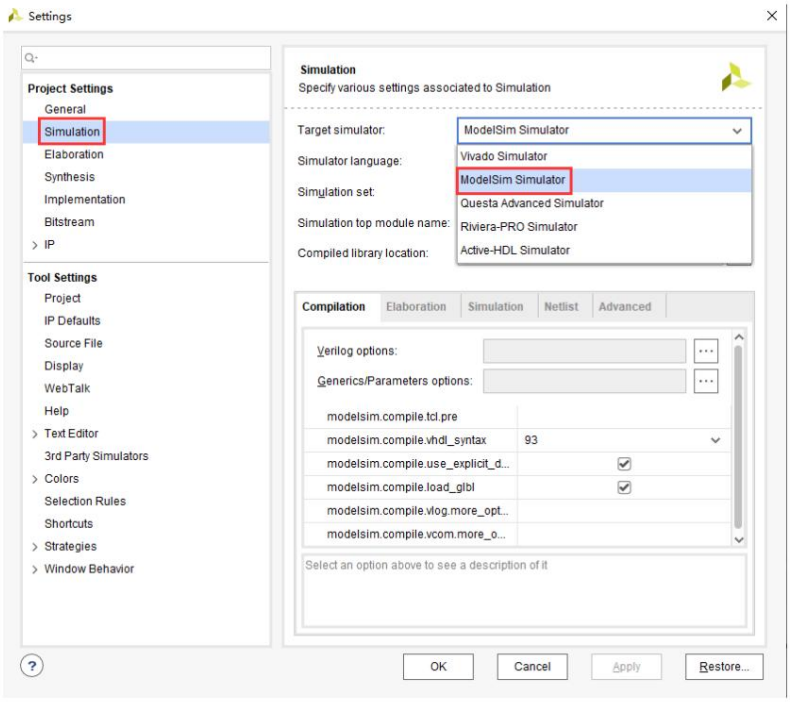


Figure 11- 13

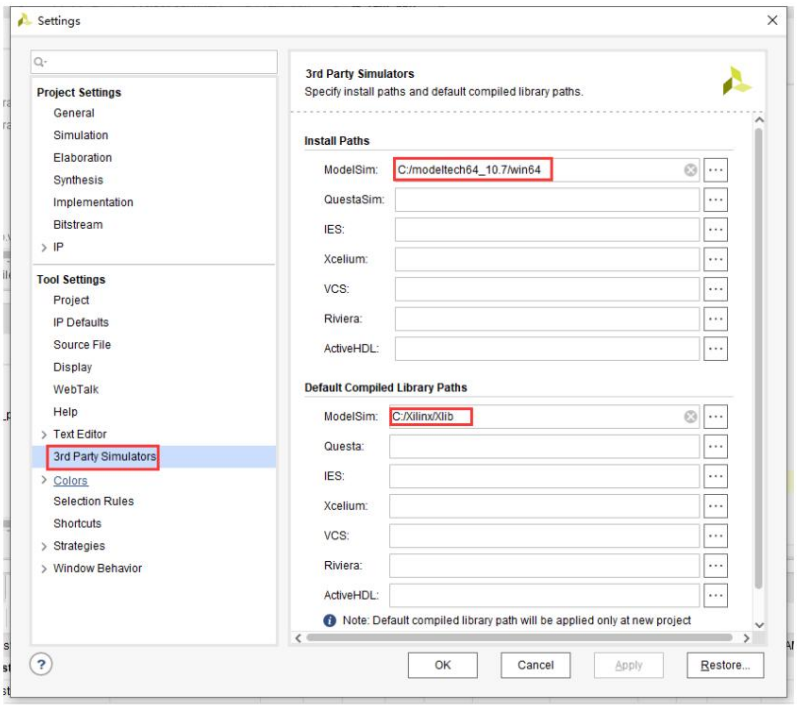


Figure 11- 14

4) Open the simulation interface

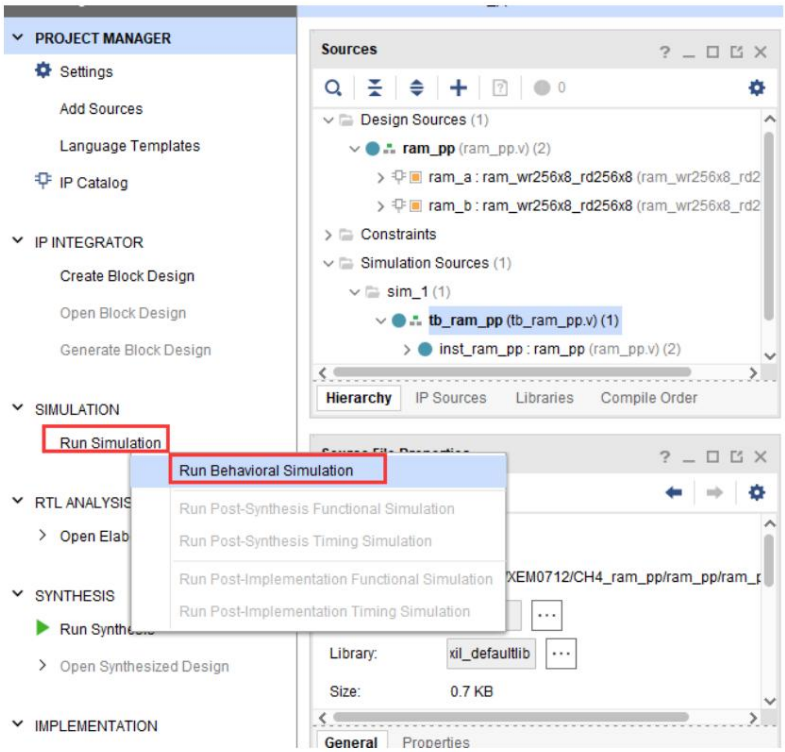


Figure 11- 15

5) Simulate and observe the waveform

Add inst_ram_pp to the waveform window

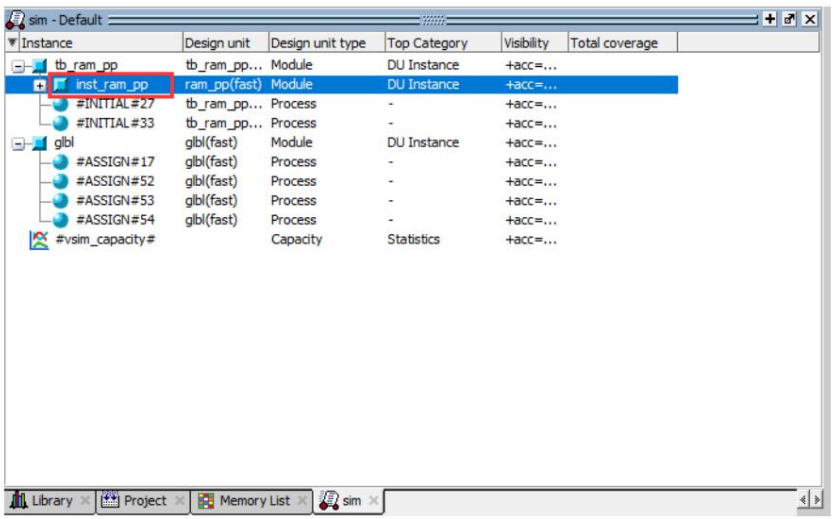


Figure 11- 16

Set the simulation time to 10us => restart => run to observe the waveform

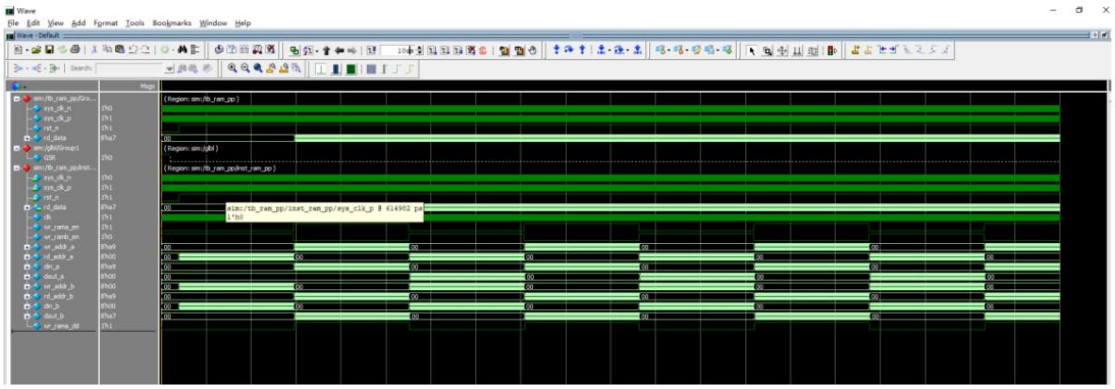


Figure 11- 17

By observing the waveform, we can see that the two RAMs performed a ping-pong operation.

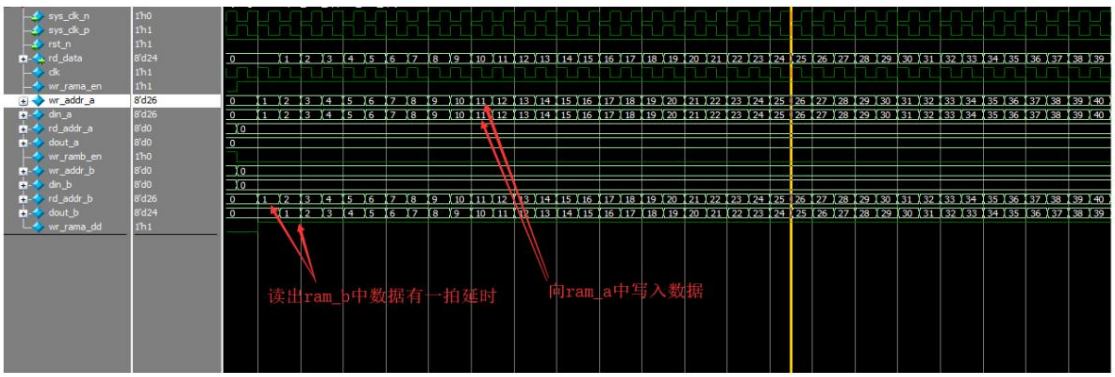


Figure 11- 18

From the above screenshot, we can see that when wr_rama_en is high, the address in ram is

At the same time, read the data from address 0 to address 255 from ram_b.

data.

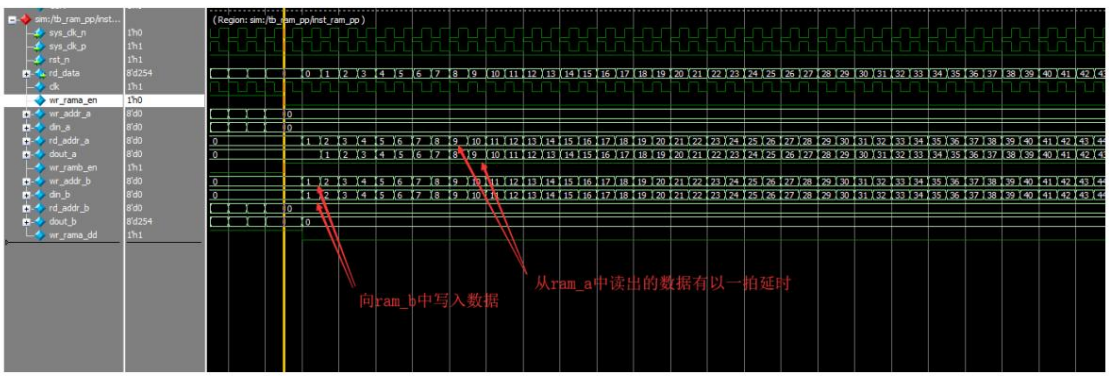


Figure 11- 19

When wr_rama_en is low, data is read from ram_a, while ram_b is reading data to address 0.

Write incrementing data to address 255.

In addition, we can see that the output signal rd_data outputs continuous data, and these numbers

The data we wrote into RAM is consistent.

Here we have completed the simulation of RAM ping-pong operation.

9.6 Experimental Summary

This experiment introduces the configuration of the Vivado Total RAM IP core and uses a simple and

The most commonly used IP core configuration completes the simulation of a RAM ping-pong operation.

In addition, a timing diagram is provided to be drawn before design.

The timing diagram is often used in actual development. Ping-pong operation is used to process high-speed data.

This is a frequently used method when streaming. By mastering the ping-pong operation of RAM in this chapter, you will be able to use it in the future.

Ping-Pong operations can also be performed on other types of on-chip storage resources.

Chapter 10 On-Chip Resources: **FIFO**

10.1 Experimental Introduction

This experiment will introduce the use of FIFO in Vivado, the configuration of FIFO IP core and the simulation.

10.2 Experimental Principle

1) The concept of FIFO

1. FIFO (First Input First Output), which means first in first out, is also a kind of memory, usually used for data buffer.

2. The similarities between FIFO and RAM are that they can store data and have signals to control writing and reading;

The similarity is that FIFO has no address, so you cannot arbitrarily specify to read a certain data, the data can only be read according to the number.

The data is output in the order of input, that is, first in first out, and reading and writing can be done at the same time.

3. If the data fills up the FIFO depth, no more data can be entered and the existing data will not be overwritten.

Data; the data in FIFO can only be read once. After reading once, the FIFO is empty and needs to be written to the FIFO again.

Only when the FIFO is read out can new data be read out, otherwise the data read out will always be the data when the FIFO was read for the last time .

The following figure shows a simple asynchronous fifo schematic diagram. The write port has write clock, write enable and

The data to be written. The read port has a read clock, read enable and read data. In addition, there is a fifo empty and full

Signal, when the fifo is full of data, the full signal is pulled high, and when all the data in the fifo is read out, the empty signal is pulled high.

The signal is pulled high.

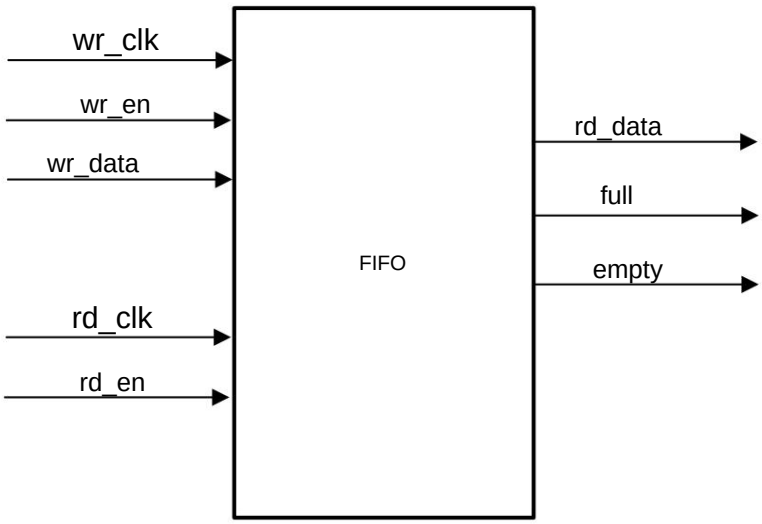


Figure 12- 1

4. Critical timing of FIFO processing

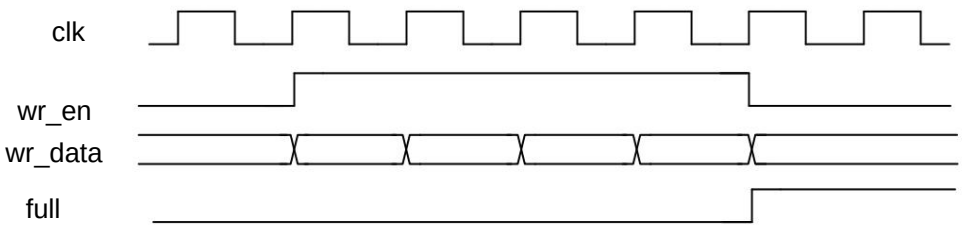


Figure 12-2 FIFO write timing

The FIFO write timing is shown in the figure above. When the write enable is pulled high, the write port data is

Write the data to be written into FIFO. When FIFO is full, the full signal is pulled high.

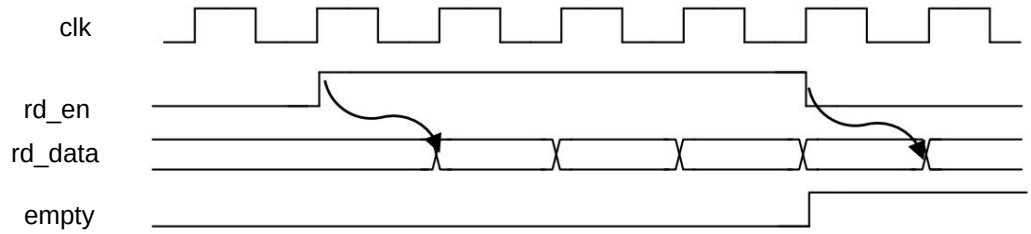


Figure 12-3 Standard read mode

The read timing of Standard is shown in the figure above. After rd_en is pulled high, data starts to be read from the fifo.

However, the data read out at this time has a one-beat delay. The Standard mode is used for comparison of the output timing characteristics.

In high-speed applications, you can also set multiple-beat delays to improve the timing characteristics between output registers through large register beats.

sex.

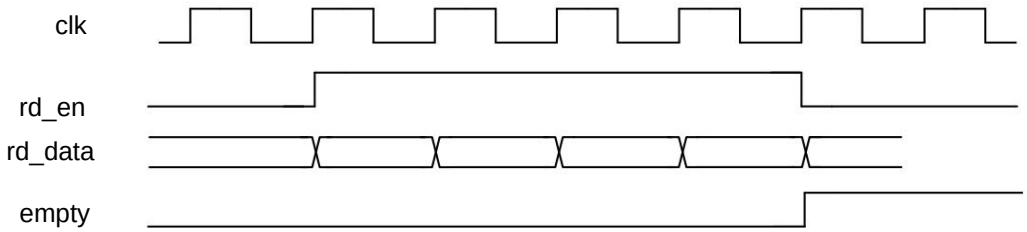


Figure 12-4 First-word Fall-Through mode

First-Word Fall-Through mode is widely used in data buffering applications. It features a delay output relative to read enable 0.

When reading out data, the principle is that the first data will automatically appear on the read port without the need for read enable.

10.3 Create a project

Double-click the Vivado2018.3 icon on the desktop and follow the steps below.

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is fifo_test, the project path is E:\demo\XEM0712\CH5_fifo. Check Createproject subdirectoryÿNext
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device, purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

10.4 Configuring **FIFO IP** Core

1) Add IP core

Click IP Catlog on the left, search for fifo in the search bar, and find Memories & storage Elements

Double-click FIFO Generate to pop up the configuration interface.

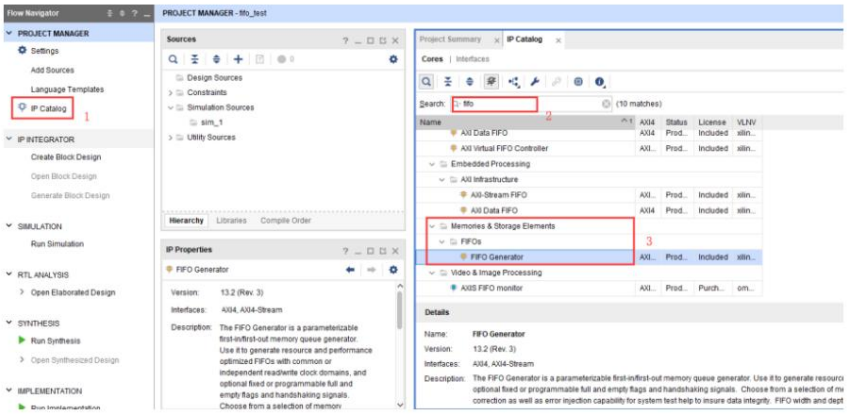


Figure 12- 5

2) IP core basic settings

On this page, you can first choose to modify the name of the fifo. Select Native type for the Fifo interface type.

implementation Select Independent clock Block RAM

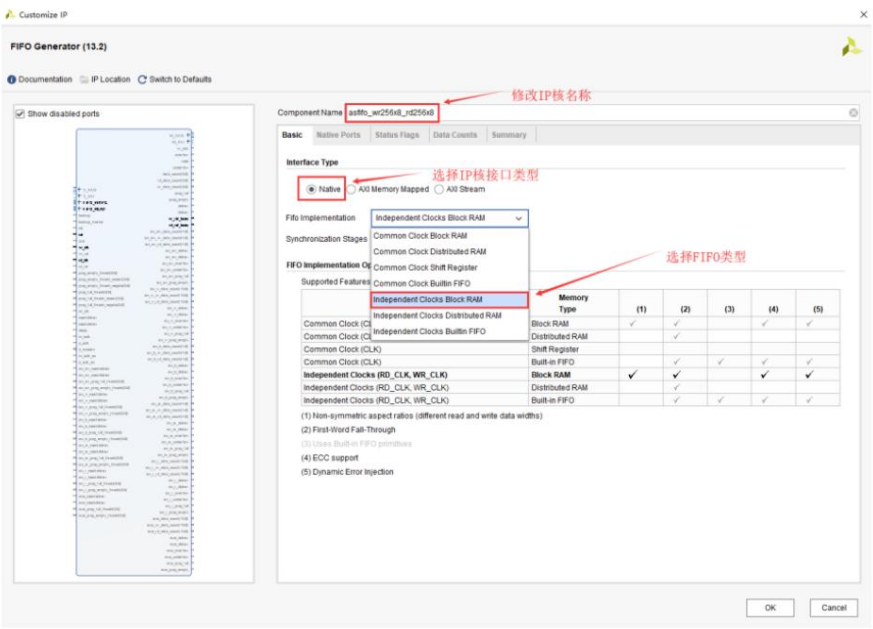


Figure 12- 6

3) FIFO input and output settings

Read Mode selects First Word Fall Throught mode;

The input data bit width is set to 8;

The Fifo depth is set to 256, where the FIFO depth must be a power of 2

The read FIFO bit width is set to 8;

Uncheck Reset Pin;

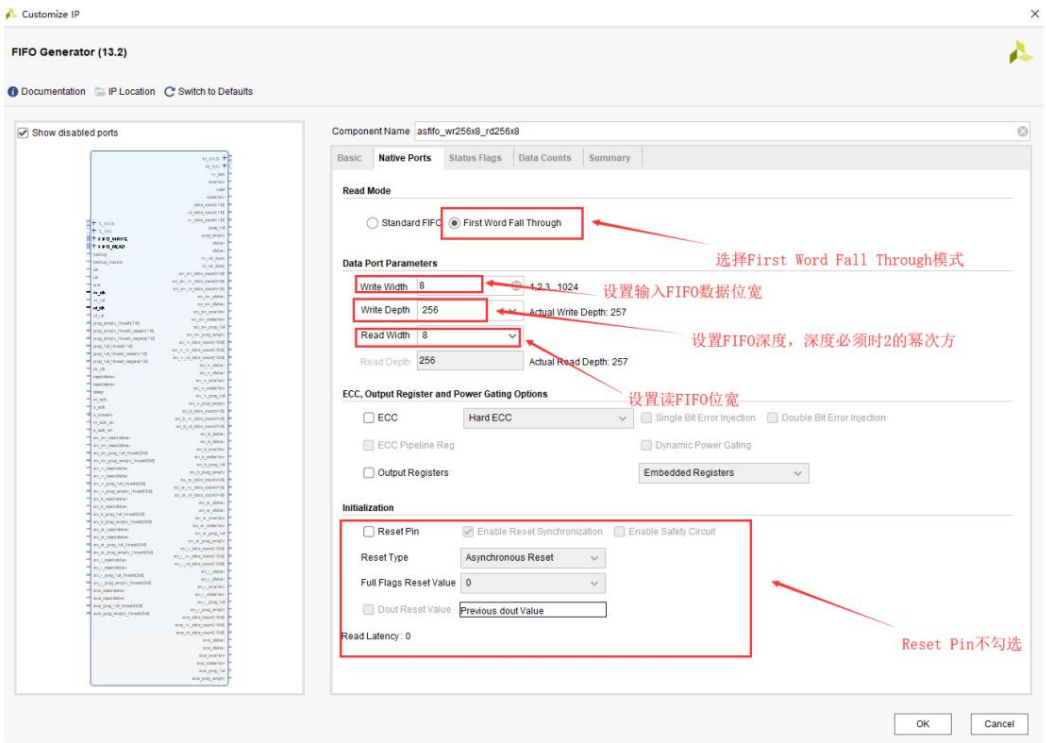


Figure 12- 7

4) Keep the Status Flag page as default; keep the Data count page as default.

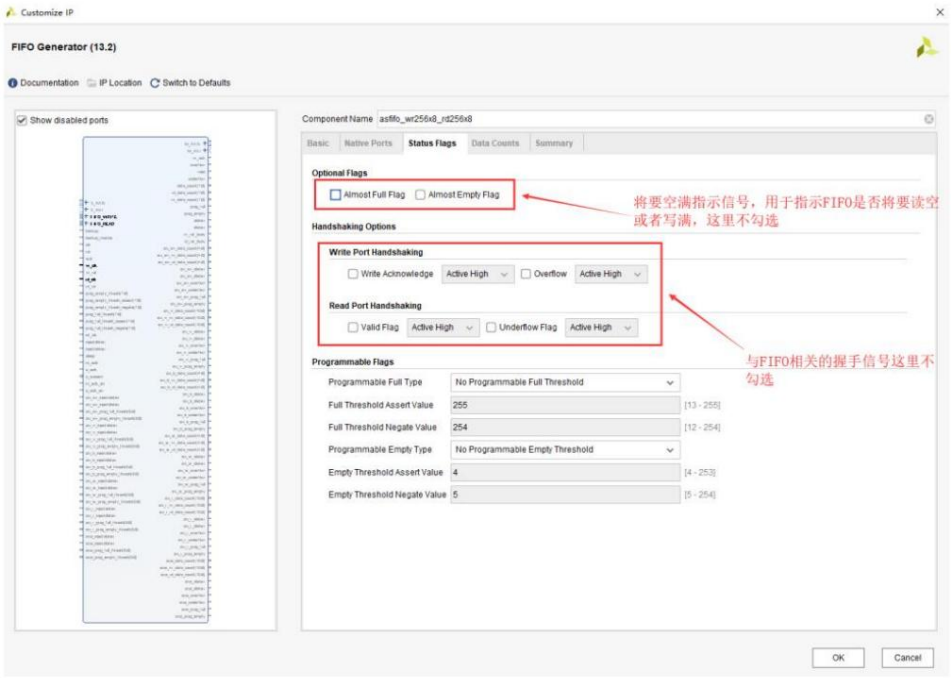


Figure 12- 8

5) On the Summary page, you can see the information of the FIFO we are going to generate. Click OK to generate the FIFO IP.

nuclear;

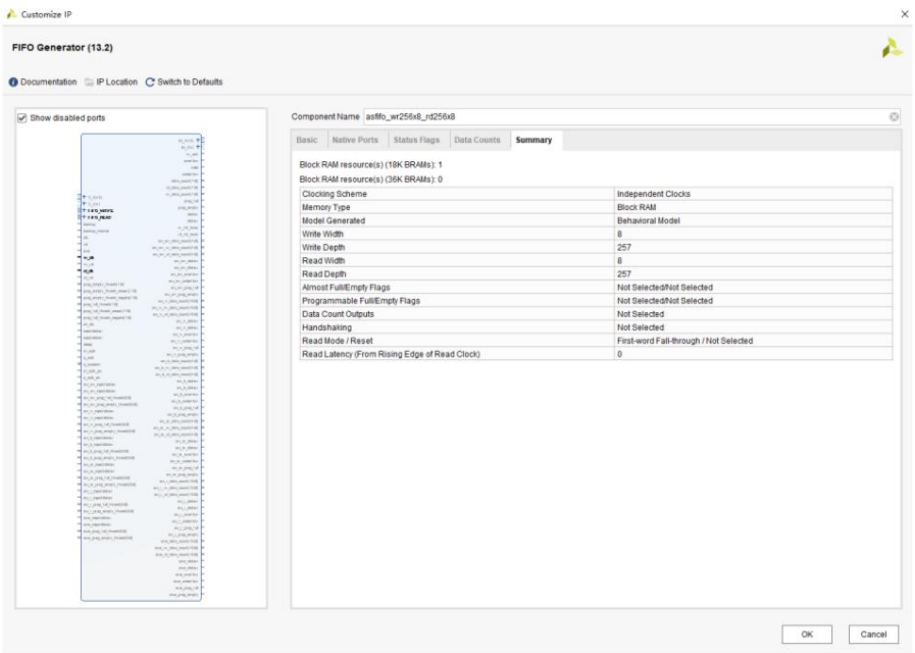


Figure 12- 9

10.5 Programming

1) Design timing

First, we design a timing diagram and write the corresponding code based on this timing diagram.

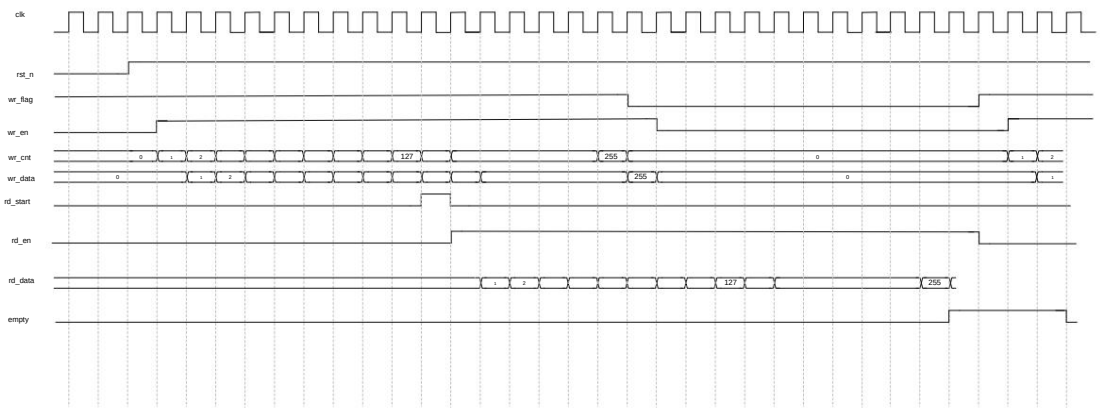


Figure 12- 10

From this timing diagram, we can see that wr_flag is used to indicate whether the current state is the write FIFO state.

wr_en is the write FIFO enable. When it is high, data is written to the FIFO. wr_data is the write

FIFO data; rd_start signal is the actual signal for reading FIFO. When this signal is pulled high, it starts to prepare to read FIFO;

The rd_en signal is the read FIFO enable signal. When this signal is pulled high, the FIFO sends the data to the output data bus.

Line; empty signal is pulled high when all data in FIFO are read out from FIFO.

The code can be easily designed based on this timing diagram.

2) Code design

According to the timing design diagram provided above, start designing the code for this experiment.

Simulation is done to verify the functionality of FIFO, so the code is relatively simple.

The source code for this experiment can be found in the corresponding project.

```
module fifo_test(
    input wire          clk         ,// System input differential clock
    input wire          rst_n
    );
```

```
parameter MAX                   = 256 - 1;
parameter RD_START = 128 - 1;
reg                             wr_en               ;//      FIFO Indicator signal
reg                                 wr_flag ;//         Write data indication signal
reg         [8:0]       wr_cnt              Write FIFO counter
reg         [7:0]       wr_data             ;//  Write FIFO data
wire                                full,empty ;//;//       Empty and full signal
reg                             rd_en               Read Enable FIFO
reg                             rd_start            ;//;//Read start signal FIFO
wire            [7:0]       rd_data             Read data


//--------------------wr_flag--------------------
//  Write FIFO Indicator signal
always @(posedge clk or negedge rst_n)begin
    if(rst_n==1'b0)begin
        wr_flag <= 1'b1;
    end
    else if(wr_cnt==MAX && wr_flag==1'b1)begin
        wr_flag <= 1'b0;
    end
    else if(empty==1'b1) begin
        wr_flag <= 1'b1;
    end
    else begin
        wr_flag <= wr_flag;
    end
end


//--------------------wr_en--------------------
always @(posedge clk or negedge rst_n)begin
    if(rst_n==1'b0)begin
        wr_en <= 1'b0;//     Low during reset
    end
    else begin
        wr_en <= wr_flag;
    end
end


//--------------------wr_cnt--------------------
```

```
always @(posedge clk or negedge rst_n)begin
    if(rst_n==1'b0)begin
        wr_cnt <= 'd0;
    end
    else if(wr_flag==1'b1)begin//    Count how much data is currently written
        if(wr_cnt == MAX)
            wr_cnt <= 'd0;
        else
            wr_cnt <= wr_cnt + 1'b1;
    end
    else begin
        wr_cnt <= 'd0;
    end
end

//--------------------wr_data--------------------
always @(posedge clk or negedge rst_n)begin
    if(rst_n==1'b0)begin
        wr_data <= 'd0;
    end
    else begin
        wr_data <= wr_cnt;//    Write FIFO Data
    end
end

//--------------------rd_start--------------------
// Start reading FIFO Signal
always @(posedge clk or negedge rst_n)begin
    if(rst_n==1'b0)begin
        rd_start <= 1'b0;
    end
    else if(wr_cnt == RD_START)begin//    When writing 128 After the data, start from FIFO Read data
        rd_start <= 1'b1;
    end
    else begin
        rd_start <= 1'b0;
    end
end

//--------------------rd_en--------------------
always @(posedge clk or negedge rst_n)begin
```

```
        if(rst_n==1'b0)begin
            rd_en <= 1'b0;
        end
        else if(rd_start==1'b1)begin//      After the read start signal is pulled high, the read enable signal is pulled high
            rd_en <= 1'b1;
        end
        else if(empty == 1'b1)begin//FIFO      Stop reading after reading empty
            rd_en <= 1'b0;
        end
        else begin
            rd_en <= rd_en;
        end
end



asfifo_wr256x8_rd256x8 fifo_inst (
    .wr_clk(clk), // input
    wr_clk .rd_clk(clk), // input
    rd_clk .din(wr_data), // input [7 : 0]
    din .wr_en(wr_en), // input
    wr_en .rd_en(rd_en), // input
    rd_en .dout(rd_data), // output [7 : 0]
    dout .full(full), // output
    full .empty(empty) // output empty
);
endmodule
```

10.6 Simulation Verification

Next, we simulate and verify the FIFO. This experiment still uses modelsim to simulate the FIFO.

real.

Before starting the simulation, you need to create and complete the testbench simulation file.

1) Design simulation files

It can be seen from the design documents of this experiment that there are only three inputs and no output in this experiment.

Therefore, when designing the simulation file, you only need to give a few simple stimuli and observe the internal signals of the module.

signal to determine whether the FIFO is working as expected.

The simulation files are as follows:

```
`timescale 1ns/1ps

module tb_fifo_test (); /* this is automatically generated */



    // (*NOTE*) replace reset, clock, others

    parameter MAX                = 256 - 1;
    parameter RD_START = 128 - 1;

    reg clk              ;
    reg rst_n                    ;

    fifo_test #(

                .MAX(MAX), .RD_START(RD_START)
        ) inst_fifo_test (
                .clk (clk),
                .rst_n          (rst_n)
        );


    initial begin
        clk = 0;
        forever #(10) clk = ~clk;
    end

    // reset
    initial begin
        rst_n <= 0;
        #200
        rst_n <= 1;
    end
endmodule
```

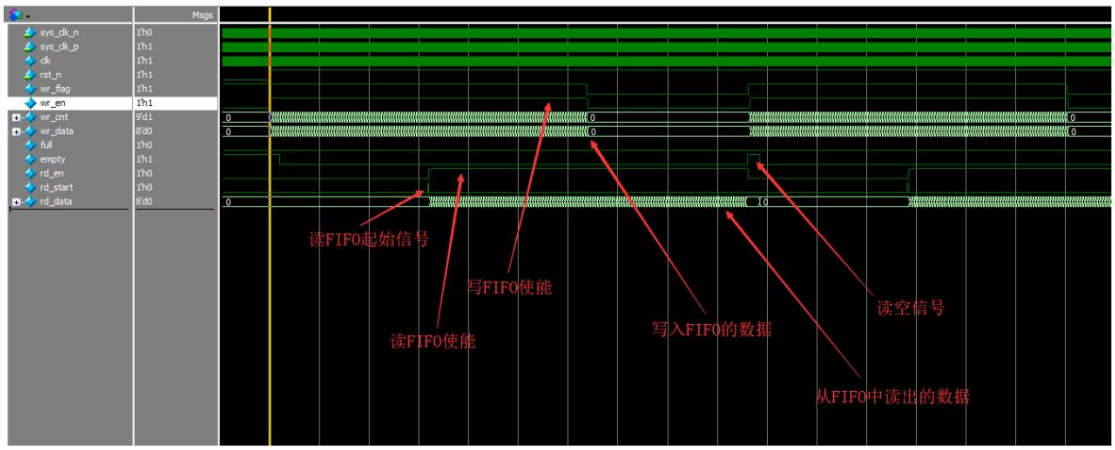
2) Vivado associates modelsim for simulation

The method of using Vivado associated with modelsim for simulation is the same as the above experiments.

I won't go into details here.

3) Observation results

Add the observed signal to the waveform window for observation

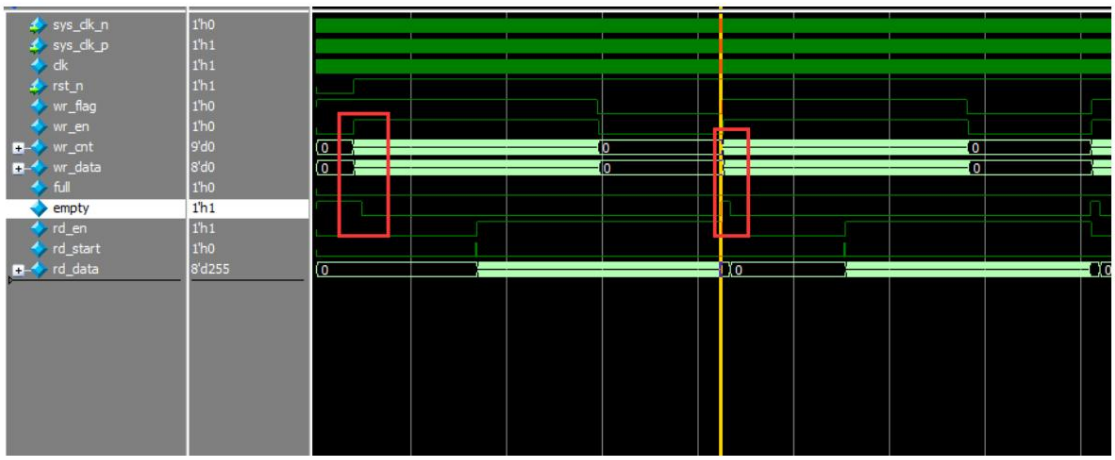


By observing the waveform, we can find that the waveform obtained by simulation is consistent with the timing diagram we designed in advance.

It is worth noting that the FIFO empty and full signals are not immediately generated when the FIFO is full or empty.

That is, pull high or pull low. For example, in this experiment, when a certain amount of data has been written into the FIFO, empty

The signal remains high.



This is related to the structure of FIFO. FIFO is essentially a special RAM.

Find information yourself to understand the structure of FIFO.

10.7 Experimental Summary

This experiment shows how to configure a simple asynchronous FIFO, which is the most commonly used one.
A single FIFO is simulated. FIFO plays a very important role in data buffering and clock domain switching.
This chapter is just a simple example. More usage methods and forms depend on the specific design.

Chapter 11 LCD Display Experiment

11.1 VGA Principle and Timing

There are many standards for displaying computer monitors, the most common ones are VGA, SVGA, etc.
VGA interface to control the display. VGA is the abbreviation of Video Graphics Array.
As a standard display interface, it is widely used.

Common color displays are generally composed of CRT (cathode ray tube), and colors are composed of R, G, B (red,
The display is solved by line-by-line scanning, and the cathode ray gun emits an electron beam to
On the fluorescent screen coated with phosphor, the three primary colors of RGB are generated to synthesize a color pixel.
Starting from the upper left corner, scanning from left to right and from top to bottom, the electron beam returns to the screen after each line is scanned.
The starting position of the next line on the left, during which the CRT blanks the electron beam.
After scanning all lines, synchronize with the field sync signal and return the scan to the screen.
At the upper left, field blanking is performed simultaneously to prepare for the next field scan.

For ordinary VGA monitors, there are 5 signals: R, G, B three primary colors; HS (horizontal synchronization
signal); VS (field synchronization signal). For timing drive, VGA monitors must strictly follow the "VGA" industrial standard.
We select the resolution of 1920x1080@60Hz mode. Usually the monitors we use meet the working
The VGA controller is an industry standard, so we need to refer to the technical specifications of the monitor when designing the VGA controller.
Field scan timing diagram:

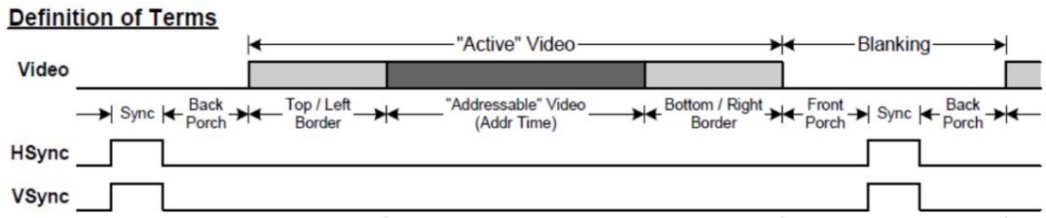
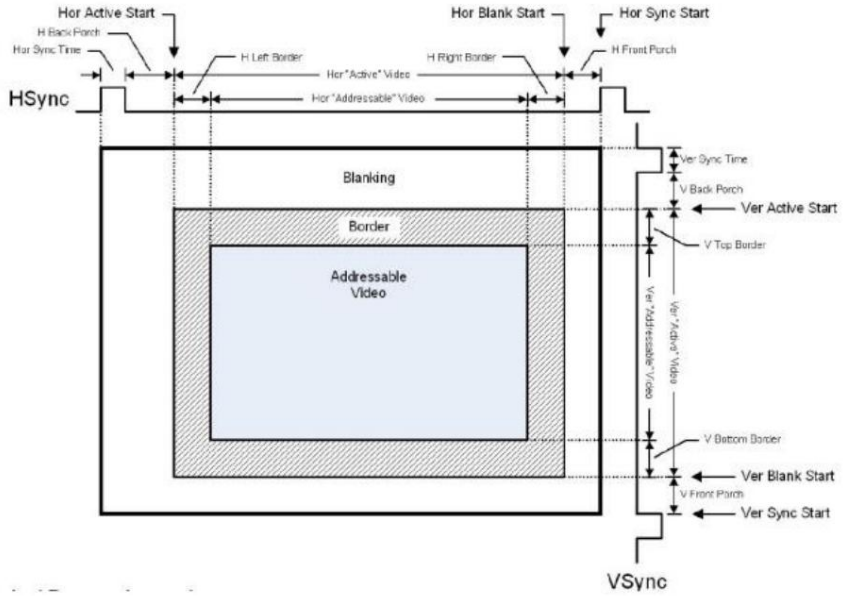


Figure 13- 1

The time to complete a line scan is called the horizontal scan time, and its reciprocal is called the line frequency; the time to complete a frame (whole screen) scan is called the horizontal scan time.

The time of scanning is called vertical scanning time, and its reciprocal is called field frequency, that is, the frequency of refreshing a screen, which is commonly 60Hz.

75Hz, etc. The standard display field frequency is 60Hz.



视频时序

Figure 13- 2

Clock frequency: Taking 1024x768@59.94Hz (60Hz) as an example, each field corresponds to 806 line cycles, of which 768 are

Each display line includes 1344 dots, of which 1024 dots are effective display areas.

Clock frequency: 806*1344*60 about 65MHz

The basic element of VGA scanning is line scanning. Multiple lines form a frame. The following figure shows the timing of a line, where "Active"

Video is the effective pixels of a line of video. In most resolution clocks, Top/Left Border and Bottom/Right

Border is 0. "Blanking" is the synchronization time of a line, and the "Blanking" time plus the "Active" Video time is

The time for one row. “Blanking” is divided into three sections: “Front Porch”, “Sync” and “Back Porch”.

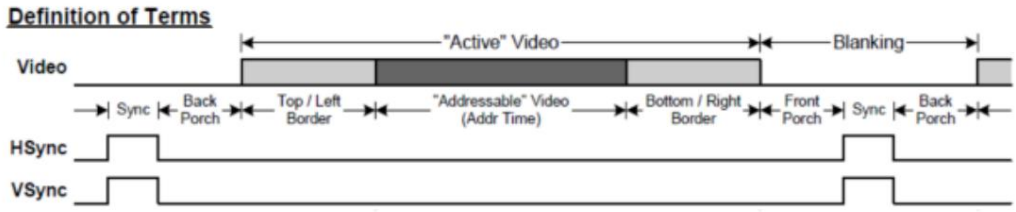


图 1 VGA 行扫描、场扫描时序示意图

Figure 13- 3

11.2 Create a project

After understanding the timing of VGA, you can start to create a project.

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is ram_pp, the project path is E:\demo\FPGA\lcdc. Check Createproject subdirectory, Next
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device: Purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

11.3 Programming

The overall results of this experiment are shown in the figure below:

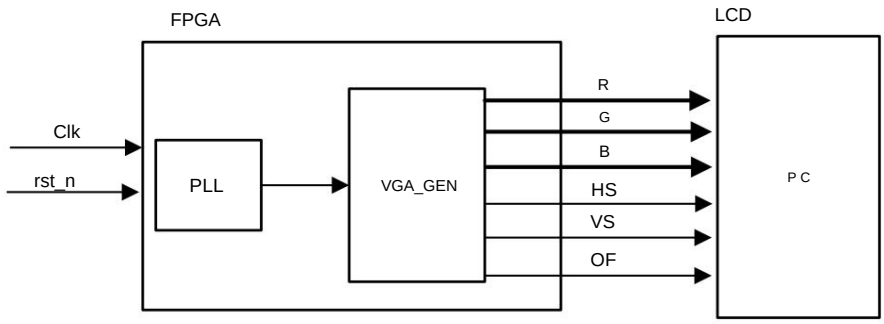


Figure 13- 4

External input clock and reset signal, the external input clock is generated by PLL for VGA image output

The required clock is generated by PLL and used to drive the VGA_GEN module to start working.

Generates horizontal and vertical synchronization signals, outputs image data, etc., and finally displays a moving square on the LCD display.

1) Graphics generation module **VGA_GEN**

In this module, the most important thing is to count the lines and fields. All other signals are based on the lines and fields.

The value of the signal counter changes.

The following is a brief explanation of the code in the program:

Lines 12 to 23 are parameter settings:

```
12 parameter      H_TOTAL = 525 - 1 ;//   The total number of values to be counted in a row
13 parameter 14   H_SYNC = 41 - 1    ;//  Line sync count
parameter         H_START = 43 - 1 ;//;//  Valid start count value of row image data
15 parameter      H_END = 523 - 1     Valid end count value of row image data
16 parameter 17   V_TOTAL = 286 - 1 ;//;//  The total number of values that need to be counted in the field
parameter 18      V_SYNC = 10 - 1     Field sync count
parameter         V_START = 12 - 1    ;//;//Field image data valid start count value
19 parameter      V_END = 284 - 1     Field image data valid end count value
20 parameter 21   SQUARE_X = 150;//      Width of the block
parameter         SQUARE_Y = 150;//     The length of the block
22 parameter      SCREEN_X = 480;//     Horizontal screen length
23 parameter      SCREEN_Y = 272;//     Vertical length of screen
```

Lines 42 to 49 call the IP core to generate the output pixel clock

```
42 clk_wiz_0 clock(
43      // Clock out ports
```

```
44          .clk_out1(vpg_pclk), // Status             // output clk_out1
45          and control signals
46          .reset(rst), // input reset
47          .locked(locked1),                        // output locked
48          // Clock in ports
49          .clk_in1(clk));                  // input clk_in1
```

Lines 52 to 76 are the line counter and field counter

```
52 //Line counter
53 always @(posedge vpg_pclk ) begin
54          if (rst==1'b1) begin
55                  cnt_h <= 'd0;
56          end
57          else if (cnt_h == H_TOTAL) begin//Count to maximum value, clear to zero
58                  cnt_h <= 'd0;
59          end
60          else if(cnt_h != H_TOTAL) begin//The maximum value has not been reached yet, so add one for each clock cycle
61                  cnt_h <= cnt_h + 1'b1;
62          end
63 end
64
65 //Field counter
66 always @(posedge vpg_pclk ) begin
67          if (rst==1'b1) begin
68                  cnt_v <='d0;
69          end
70          else if (cnt_v == V_TOTAL && cnt_h == H_TOTAL) begin//The field counter counts to the maximum value,
Clear (end of a frame)
71                  cnt_v <= 'd0;
72          end
73          else if(cnt_h == H_TOTAL) begin//One line scan ends, field counter increases by one
74                  cnt_v <= cnt_v + 1'b1;
75          end
76 end
```

Lines 156 to 176 determine the value of the output image based on the value of the counter. The moving square is gray.

In other cases, the background colors red, green and blue are displayed.

```
156 //rgb
```

```
157 always @(posedge vpg_pclk ) begin
158         if (rst==1'b1) begin
159                 rgb <='d0;
160         end
161         else if(cnt_h >=H_START+x && cnt_h <=H_START+SQUARE_X+x &&
cnt_v >=V_START+y && cnt_v <=V_START+SQUARE_Y+y)begin
162                 rgb <= 24'hFFB6C1; //output square image
163         end
164         else if (cnt_h >=H_START && cnt_h <H_END && cnt_v >=V_START && cnt_v <V_END
&& cnt_h[4:0]>='d20) begin
165                 rgb <=24'h00FF00;//green
166         end
167         else if (cnt_h >=H_START && cnt_h <H_END && cnt_v >=V_START && cnt_v <V_END
&& (cnt_h[4:0]>='d10 && cnt_h[2:0]<'d20)) begin
168                 rgb <=24'h0000FF;//bulue
169         end
170         else if (cnt_h >=H_START && cnt_h <H_END && cnt_v >=V_START && cnt_v <V_END
&& cnt_h[4:0]<'d10) begin
171                 rgb <=24'hFF0000;//red
172         end
173         else begin
174                 rgb <= 'd0;
175         end
176 end
```

11.4 Add constraint file

After the design is completed, you can add constraint files. For this experiment, you need to use the extension sub

In this experiment, insert it into the GPIO1 slot. The pin assignment is as follows:

```
############## clock define##################
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

############## key define##################
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
```

```
############### vga clock ,de,hs,vs###################
set_property PACKAGE_PIN T10 [get_ports vpg_pclk]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_pclk]

set_property PACKAGE_PIN W15 [get_ports vpg_hs]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_hs]

set_property PACKAGE_PIN V15 [get_ports vpg_vs]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_vs]

set_property PACKAGE_PIN R14 [get_ports vpg_de]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_de]

set_property PACKAGE_PIN T11 [get_ports vpg_disp]
set_property IOSTANDARD LVCMOS33 [get_ports vpg_disp]


set_property PACKAGE_PIN W16 [get_ports {rgb[0]}]
set_property PACKAGE_PIN V16 [get_ports {rgb[1]}]
set_property PACKAGE_PIN R18 [get_ports {rgb[2]}]
set_property PACKAGE_PIN T17 [get_ports {rgb[3]}]
set_property PACKAGE_PIN W13 [get_ports {rgb[4]}]
set_property PACKAGE_PIN V12 [get_ports {rgb[5]}]
set_property PACKAGE_PIN T15 [get_ports {rgb[6]}]
set_property PACKAGE_PIN T14 [get_ports {rgb[7]}]
set_property PACKAGE_PIN P18 [get_ports {rgb[8]}]
set_property PACKAGE_PIN Y17 [get_ports {rgb[9]}]
set_property PACKAGE_PIN Y16 [get_ports {rgb[10]}]
set_property PACKAGE_PIN V18 [get_ports {rgb[11]}]
set_property PACKAGE_PIN N17 [get_ports {rgb[12]}]
set_property PACKAGE_PIN V17 [get_ports {rgb[13]}]
set_property PACKAGE_PIN Y14 [get_ports {rgb[14]}]
set_property PACKAGE_PIN W14 [get_ports {rgb[15]}]
set_property PACKAGE_PIN Y19 [get_ports {rgb[16]}]
set_property PACKAGE_PIN Y18 [get_ports {rgb[17]}]
set_property PACKAGE_PIN W19 [get_ports {rgb[18]}]
set_property PACKAGE_PIN W18 [get_ports {rgb[19]}]
set_property PACKAGE_PIN U17 [get_ports {rgb[20]}]
set_property PACKAGE_PIN T16 [get_ports {rgb[21]}]
set_property PACKAGE_PIN R17 [get_ports {rgb[22]}]
```

```
set_property PACKAGE_PIN R16 [get_ports {rgb[23]}]
set_property IOSTANDARD LVCMOS33 [get_ports {rgb[*]}]
```

11.5 Download Verification

After the bit file is generated, connect the development board to the LCD expansion daughter card as follows.

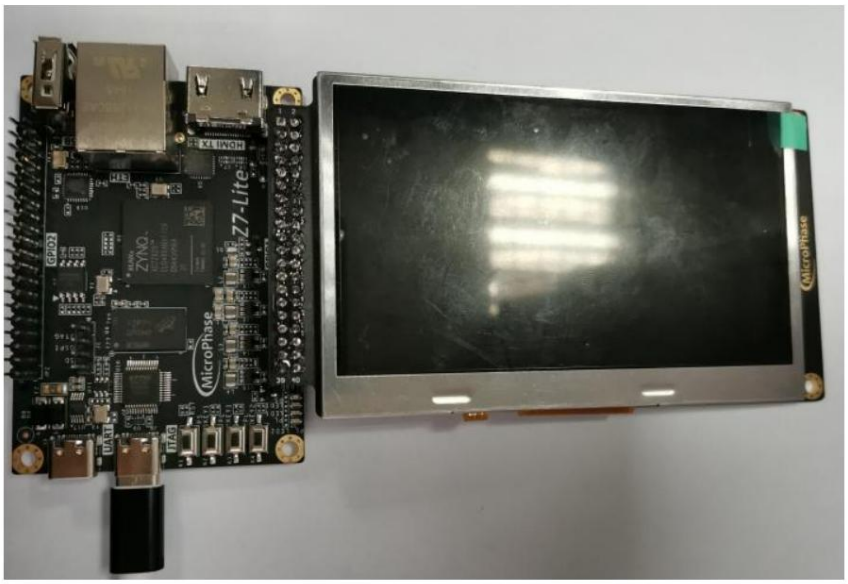


Figure 13- 5

Connect the LCD and download the bit file to the development board, and you can see the moving blocks appear on the screen.

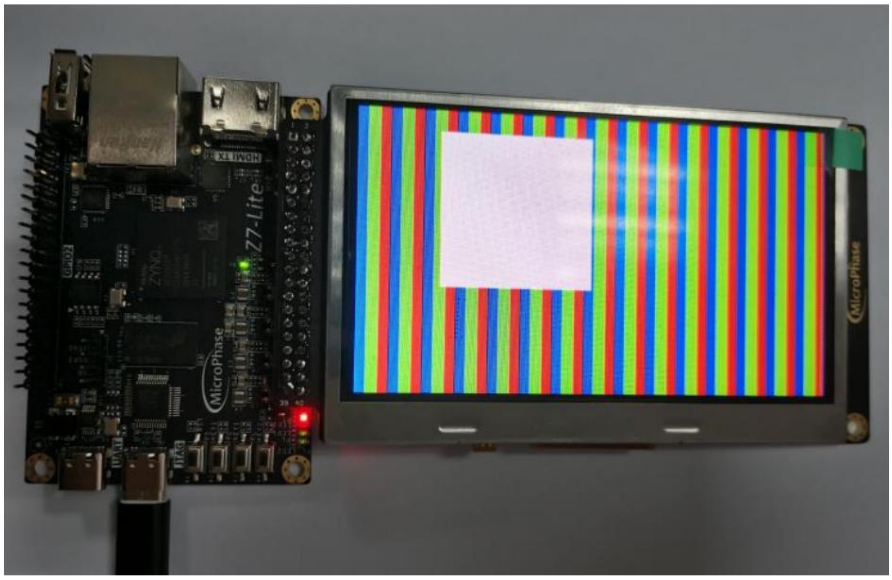


Figure 13- 6

11.6 Experimental Summary

In this experiment, we completed an example of using VGA timing to drive an LCD display.

The most important thing in the experience is to understand the VGA timing. After understanding the relevant timing, you can write the driver later.

It is very simple when you start the program.

In this experiment, since the timing parameters can be reconfigured, the VGA module will be transplanted later.

It will bring convenience.

12.1 Experimental Introduction

In the previous chapters, we have learned what VGA timing is and have successfully driven LCD The monitor shows a simple moving block. There are two HDMI ports on the Z7-Lite development board. One is for input and the other is for output. The audio and video signals can be directly transmitted through the FPGA pins. In this experiment, we will use FPGA to complete the HDMI output experiment.

Based on the HDMI interface, the image is output on a display that supports HDMI. First, you need to understand the following HDMI structure.

12.2 Experimental Principle

1) Overview

The HDMI system architecture consists of a source and a sink. A device may have one or more HDMI input, and one or more HDMI outputs. On these devices, each HDMI input should comply with the HDMI Each HDMI output should follow the HDMI source rules.

The cable and connector provide four differential pairs that make up the TMDS data and clock channels. In addition, HDMI provides a VESA DDC channel. DDC is used to

Configure and exchange states between a single source and a single sink.

Provides a high level of control over a wide range of audio and video products. Optional HDMI Ethernet and Audio Return (HEAC), providing Ethernet-compatible network data and an

TMDS audio return channel for opposite directions.

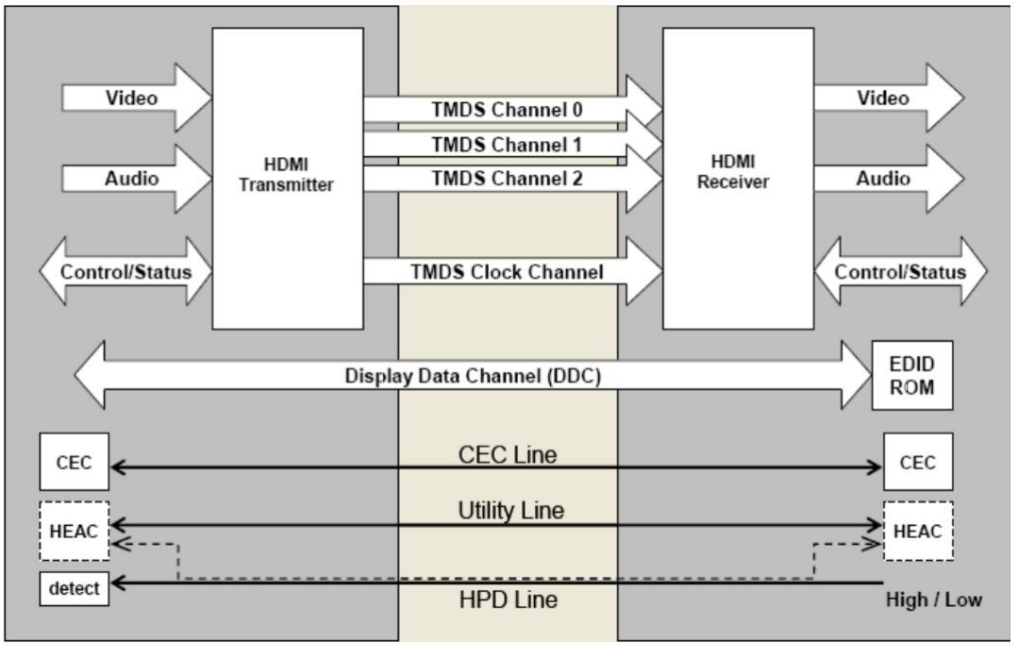


Figure 14- 1

Audio, video, and auxiliary data are transmitted in three TMDS data channels. One TMDS clock, typically

The video pixel rate is transmitted in the TMDS clock channel, which is used as a frequency reference by the receiver.

It is used to restore the data of three TMDS data channels. At the source end, TMDS encoding converts each TMDS

The 8 bits of data are converted into a 10-bit DC-balanced minimum transition sequence, serially, with each

TMDS clock cycle 10 bits, sent on the differential line pair. Video data, one pixel can be 24,

30, 36, 48 bits. The default 24-bit depth of video is uploaded at a TMDS clock equal to the pixel clock.

Higher color depths use correspondingly higher TMDS clock rates. Video formats with TMDS clock rates lower than

25M (such as 13.5M 480i/NTSC) can use the strategy of repeated pixel transmission. Video pixels can be

Encode in RGB, YCbCr4:4:4, YCbCr4:2:2 formats.

2) Signal and Coding

An HDMI connection consists of three TMDS data channels and one TMDS clock channel.

The clock channel runs at a constant rate that is proportional to the pixel rate of the video.

During the clock channel cycle, each of the three TMDS data channels sends 10 bits of data. This 10-bit word is

Encoding, using some different encoding techniques. The input stream to the source contains video pixels, data

packets, and control data. The data packets include audio data and auxiliary and associated error correction codes. These data items are

are handled differently and are represented as either 2 bits of control data in the TMDS encoder for each TMDS channel,

Or 4-bit message data, or 8-bit video data. The source end processes these

Encode the data type or encode a boundary character.

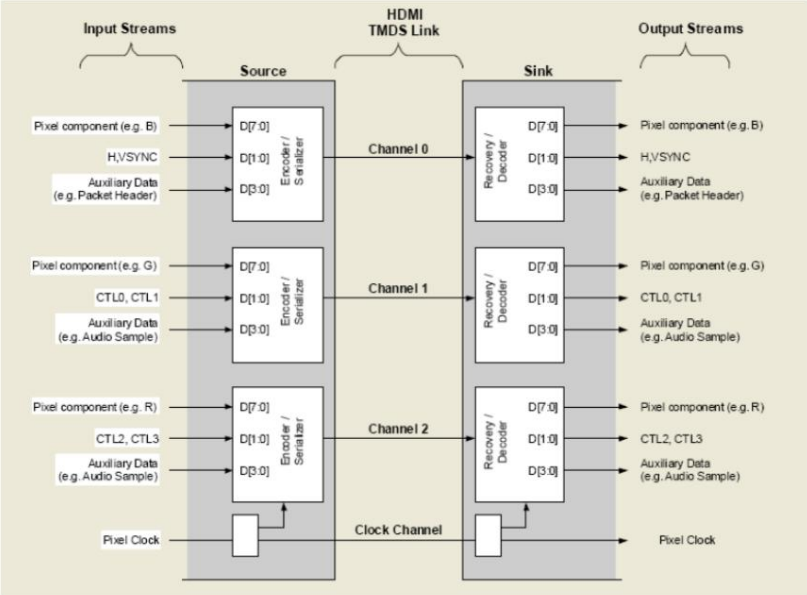


Figure 14- 2

12.3 Create a project

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is ram_pp, the project path is E:\demo\FPGA\hdmi, check Createproject subdirectory, Next
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device: Purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

12.4 Overall Architecture

The overall architecture of this experiment is shown in the figure below. There is a clock module instantiated inside the FPGA for Provides pixel clock and a clock with a frequency five times that of the pixel clock. A VGA driver module is instantiated for Generate output images. There are three important parts inside the HDMI transmission module, namely 8b/10b Encoding module, serial-to-parallel conversion module and single-ended to differential module. This experiment mainly focuses on several modules. to complete.

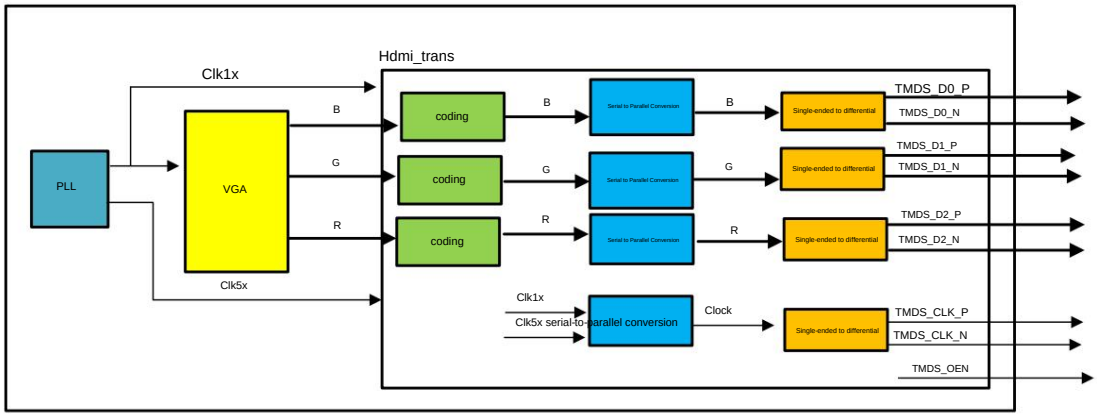


Figure 14- 3

1) Encoding module

The encoding module converts 8-bit data into 10-bit data, with each RGB color component occupying 8 bits.

bits, then the color data on each channel will be **converted into** a

10-bit pixel character.

Schematic diagram of a three-channel DVI encoder. For the three RGB color channels of pixel data, the encoder logic VDE is used for each channel to select whether to output video pixel data or control data. When VDE is valid When the current transmission is video data, HSYNC and VSYNC signals are encoded in the blue channel to obtain 10 bits. The green and red channel control signals C0 and C1 also need to be encoded.

However, the control signals of these two channels are reserved (not used) in the DVI specification, so Set it to 2'b00.

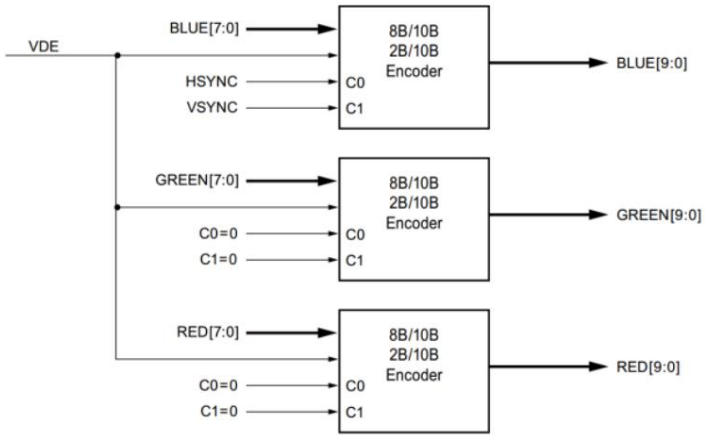


Figure 14- 4

The coding module for this experiment does not need to be written by yourself. You can download the 8b/10b code from the Xilinx official website.

Conversion modules are also provided in our project.

2) Serial-to-parallel conversion module

The 10-bit characters after the encoding module **are converted into serial data through the serializer.**

The serial data rate generated by this 10:1 parallel-to-serial process is actually

10 times the pixel clock rate. Since HDMI can transmit data on both the rising and falling edges of the clock,

A serial clock that is 5 times the pixel clock is required, which is clk5x in the overall architecture diagram.

This module will use the primitive OSERDESE2 provided by Xilinx for serial-to-parallel conversion. This primitive can be found in

Check **the UG768** and **UG471** manuals provided by **Xilinx .** The model of its primitive is as follows:

The primitive model only supports serial conversion of 8-bit data. In this experiment, 10-bit data conversion is required.

Therefore, two OSERDESE2 primitives need to be cascaded to perform serial conversion of 10-bit data.

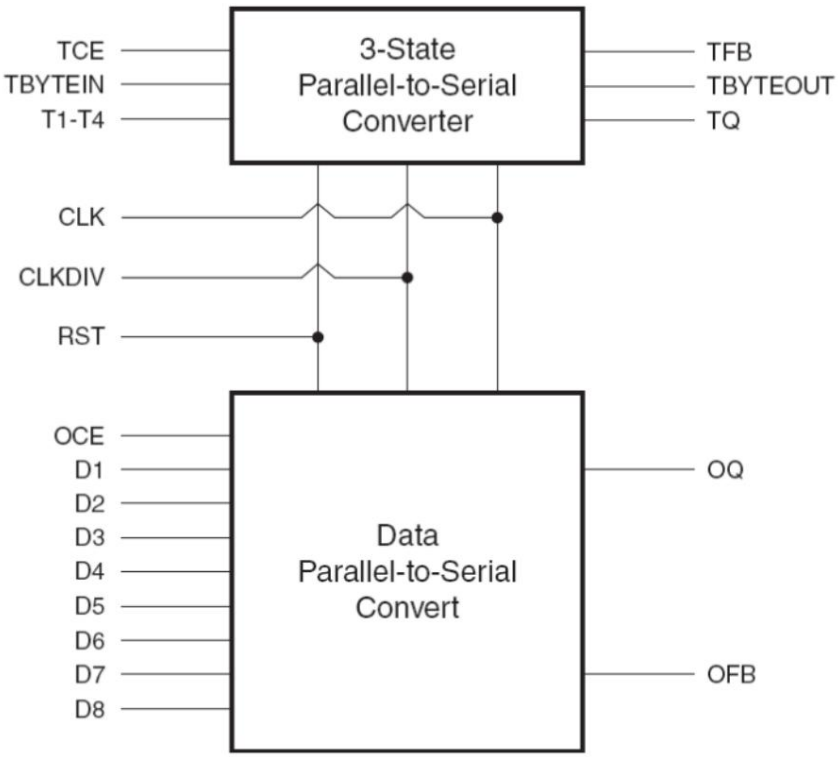


Figure 14- 5

The primitive model after cascading is as follows: The Master end of the OSERDESE2 primitive receives the low-level 10-bit data.

8bit, D3 and D4 of the slave end of the OSERDESE2 primitive receive the high 2 bits of the 10bit data and

The output SHIFTOUT1, 2 is connected to the SHIFTIN1, 2 of the Master, and the input 10-bit data will be

It will be output from the Data Out terminal in order from low to high.

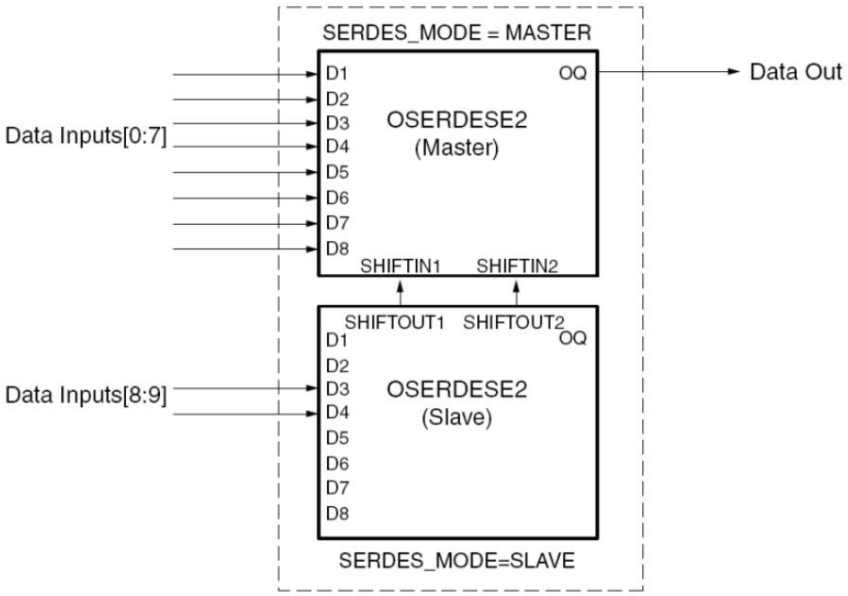


Figure 14- 6

3) Single-ended to differential module

After completing the serial-to-parallel conversion, the data is still single-ended data, so it needs to go through a single-ended-to-differential conversion.

The primitive of OBUFDS can convert the signal inside the FPGA into a differential signal and output it to the outside of the FPGA.

The model is as follows. The use of OBUFDS can also refer to **UG768** and **UG471.**

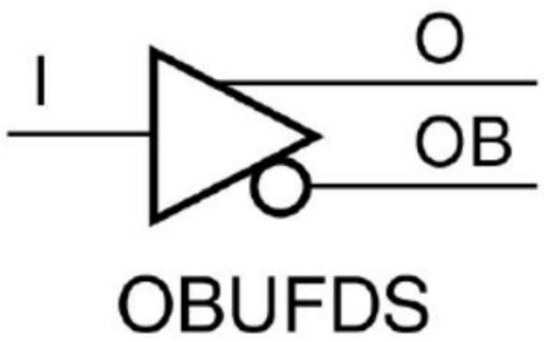


Figure 14- 7

The final completed serial-to-parallel conversion code is as follows:

```
module parallel_to_serial(
        input       wire                clk1x       ,
        input       wire                clk5x       ,
        input       wire                rst         ,
        input       wire    [9:0]from ,
        output wire                 dout_p ,
        output wire                 doubt_n
        );


wire                doubt       ;
wire                shift_in1 ;
wire                shift_in2   ;


OBUFDS #(
            .IOSTANDARD("DEFAULT"), // Specify the output I/O standard
            .SLEW("SLOW")                   // Specify the output slew rate
        ) OBUFDS_inst (
            .O(dout_p), // Diff_p output (connect directly to top-level port)
            .OB(dout_n), // Diff_n output (connect directly to top-level port)
            .I(dout)            // Buffer input
        );
OSERDESE2 #(
            .DATA_RATE_OQ("DDR"), // DDR, SDR
            .DATA_RATE_TQ("SDR"), // DDR, BUF, SDR
                                            // Parallel data width (2-8,10,14)
                                        // Initial value of OQ output (1'b0,1'b1)
                                       // Initial value of TQ output (1'b0,1'b1)
            .DATA_WIDTH(10), .INIT_OQ(1'b0), .INIT_TQ(1'b0), .SERDES_MODE("MASTER"), // MASTER, SLAVE
                                        // OQ output value when SR is used (1'b0,1'b1)
                                        // TQ output value when SR is used (1'b0,1'b1)
                                       // Enable tristate byte operation (FALSE, TRUE)
                                       // Tristate byte source (FALSE, TRUE)
            .SRVAL_OQ(1'b0), .SRVAL_TQ(1'b0), .TBYTE_CTL("FALSE"), .TBYTE_SRC("FALSE"), .TRISTATE_WIDTH(1)
        )
        OSERDESE2_inst_master (
                                       // 1-bit output: Feedback path for data
                                            // 1-bit output: Data path output
            .OFB(), .OQ(dout), // SHIFTOUT1 / SHIFTOUT2: 1-bit (each) output: Data output expansion (1-bit each)
```

```
        .SHIFTOUT1(),
        .SHIFTOUT2(),
        .TBYTEOUT(), // 1-bit output: Byte group tristate
        .TFB(),                         // 1-bit output: 3-state control
                                        // 1-bit output: 3-state control
                                            // 1-bit input: High speed clock
        .TQ(), .CLK(clk5x), .CLKDIV(clk1x), // 1-bit input: Divided clock
        // D1 - D8: 1-bit (each) input: Parallel data inputs (1-bit each)
        .D1(from[0]),
        .D2(your[1]),
        .D3(day[2]),
        .D4(your[3]),
        .D5(from[4]),
        .D6(from[5]),
        .D7(from[6]),
        .D8(from[7]),
                                            // 1-bit input: Output data clock enable
                                        // 1-bit input: Reset
        .OCE(1'b1), .RST(rst), // SHIFTIN1 / SHIFTIN2: 1-bit (each) input: Data input expansion (1-bit each)
        .SHIFTIN1(shift_in1),
        .SHIFTIN2(shift_in2),
        // T1 - T4: 1-bit (each) input: Parallel 3-state inputs
        .T1(1'b0),
        .T2(1'b0),
        .T3(1'b0),
        .T4(1'b0),
        .TBYTEIN(1'b0),             // 1-bit input: Byte group tristate
        .TCE(1'b0)                      // 1-bit input: 3-state clock enable
    );


    USERDESE2 #(
        .DATA_RATE_OQ("DDR"), // DDR, SDR
        .DATA_RATE_TQ("SDR"), // DDR, BUF, SDR
                                        // Parallel data width (2-8,10,14)
        .DATA_WIDTH(10), .INIT_OQ(1'b0), // Initial value of OQ output (1'b0,1'b1)
                                    // Initial value of TQ output (1'b0,1'b1)
        .INIT_TQ(1'b0), .SERDES_MODE("SLAVE"), // MASTER, SLAVE
                                    // OQ output value when SR is used (1'b0,1'b1)
                                    // TQ output value when SR is used (1'b0,1'b1)
        .SRVAL_OQ(1'b0), .SRVAL_TQ(1'b0), .TBYTE_CTL("FALSE"), // Enable tristate byte operation (FALSE, TRUE)
```

```
                                        // Tristate byte source (FALSE, TRUE)
        .TBYTE_SRC("FALSE"), .TRISTATE_WIDTH(1)      // 3-state converter width (1,4)
    )
    OSERDESE2_inst_slave (
        .OFB(), //                       // 1-bit output: Feedback path for data
        .OQ(), //  1-bit output: Data path output
        SHIFTOUT1 / SHIFTOUT2: 1-bit (each) output: Data output expansion (1-bit each)
        .SHIFTOUT1(shift_in1),
        .SHIFTOUT2(shift_in2),
        .TBYTEOUT(), // 1-bit output: Byte group tristate
        .TFB(),                         // 1-bit output: 3-state control
                                        // 1-bit output: 3-state control
                                            // 1-bit input: High speed clock
        .TQ(), .CLK(clk5x), .CLKDIV(clk1x), // 1-bit input: Divided clock
        // D1 - D8: 1-bit (each) input: Parallel data inputs (1-bit each)
        .D1(),
        .D2(),
        .D3(day[8]),
        .D4(your[9]),
        .D5(),
        .D6(),
        .D7(),
        .D8(),
        .OCE(1'b1),                     // 1-bit input: Output data clock enable
        .RST(rst), //                    // 1-bit input: Reset
        SHIFTIN1 / SHIFTIN2: 1-bit (each) input: Data input expansion (1-bit each)
        .SHIFTIN1(),
        .SHIFTIN2(),
        // T1 - T4: 1-bit (each) input: Parallel 3-state inputs
        .T1(1'b0),
        .T2(1'b0),
        .T3(1'b0),
        .T4(1'b0),
                                        // 1-bit input: Byte group tristate
        .TBYTEIN(1'b0), .TCE(1'b0)      // 1-bit input: 3-state clock enable
    );


endmodule
```

4) VGA timing generation module

In the previous chapter, we have introduced the VGA module. In this experiment, we also use the VGA module.

In this experiment, the VGA module is used and configured to 1920*1080@60Hz timing.

It can display high-definition images on HDMI monitors.

In this experiment, you only need to change its parameters to transplant the VGA module into this project.

The timing parameters of this experiment are as follows: Timing parameters at different resolutions, in the provided VESA timing

You can search in.

```
parameter       H_TOTAL = 2200 - 1 ; //Total value to be counted in one row
parameter       H_SYNC = 44 - 1 ; //Line synchronization count value
parameter       H_START = 190 - 1 ; // Row image data valid start count value
parameter       H_END = 2110 - 1 ; // Row image data valid end count value
parameter       V_TOTAL = 1125 - 1 ; //Total value to be counted
parameter       V_SYNC = 5 - 1 ; //Field synchronization count value
parameter       V_START = 41 - 1 ; // Field image data valid start count value
parameter       V_END = 1121 - 1 ; // Field image data valid end count value
parameter       SQUARE_X = 500 ; //Width of the square
parameter       SQUARE_Y = 500 ; //The length of the square
parameter       SCREEN_X        = 1920; // horizontal length of the screen
parameter       SCREEN_Y = 1080; // vertical length of the screen
```

5) Clock module

As mentioned above, there are two clocks needed for this experiment. One is the pixel clock.

In the experiment, the pixel clock in 1080P mode is 148.5MHz, and a serial clock 5 times the pixel clock is required.

The clock IP is used to generate the two clocks.

12.5 Add pin constraints

The pin constraints that need to be added in this experiment are as follows: They correspond to the various signals required by TMDS.

Where hdmi_oen is the output enable signal.

```
############## clock define##################
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
```

```
############### key define###################
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]

set_property PACKAGE_PIN U18 [get_ports hdmi_tx_clk_p]
set_property PACKAGE_PIN N20 [get_ports {hdmi_tx_chn_b_p}]
set_property PACKAGE_PIN T20 [get_ports {hdmi_tx_chn_g_p}]
set_property PACKAGE_PIN V20 [get_ports {hdmi_tx_chn_r_p}]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_r_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_g_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_b_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_clk_p]
```

12.6 Download Verification

After the bit file is generated, you can download and verify it. Finally, we can view it on the HDMI display.

Until a moving block has appeared.

First, connect the development board, download the bit file into the FPGA, then plug in the HDMI cable and observe

The screen that appears on the monitor.

Figure 14- 8

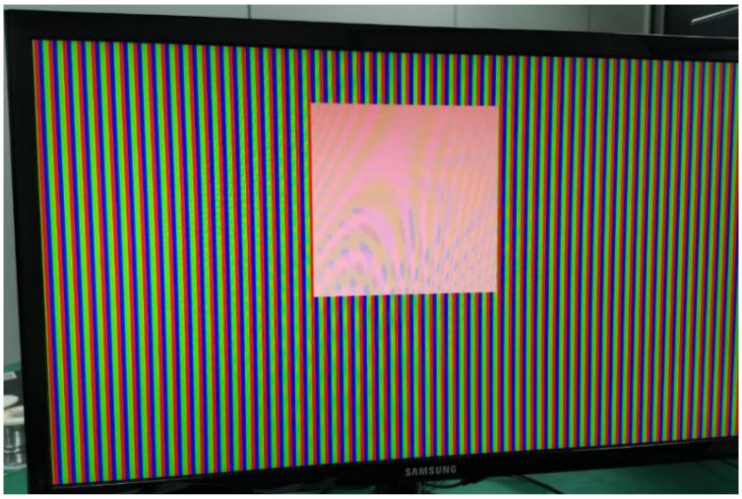


Figure 14- 9

12.7 Experimental Summary

In this experiment, we completed an example of displaying moving blocks on an HDMI monitor.

The content is about the principle of HDMI and the implementation of HDMI sending module. In the HDMI sending implementation module,

The focus is on the 8b/10b encoding module and the serial-to-parallel conversion module, in which the serial-to-parallel conversion module needs to use

The primitive is used to perform serial-to-parallel conversion and convert single-ended data into differential data.

information to use these two primitives.

Chapter 13 HDMI output image experiment

In the previous experiment, we have completed the task of displaying a moving square on the HDMI monitor.

In this experiment, a small image will be displayed on the monitor. Since the DDR chip on the development board is connected to

PS side is not convenient to access directly when developing PL side, so this experiment uses the on-chip storage space to store

It should be noted that due to the limited storage resources inside the FPGA, it is generally not possible to

Use internal storage resources to store large amounts of data. In this experiment, the image size to be displayed is

256*256ÿ

13.1 Generate image files

Before conducting this experiment, use MATLAB to generate an image of the data in hexadecimal format.

The image used is as follows, with a size of 256*256. Place the image and the matlab script in the same file

Folder, then create a new matlab script and enter the following content in the matlab script.

Figure 15- 1

```
% Convert png format images to the commonly used RGB format in digital image processing
clear
rgb1=imread('image.png');%Read image data
imtool(rgb1)
r=rgb1(1:256,1:256,1);%Take the R component
```

```
g=rgb1(1:256,1:256,2);
b=rgb1(1:256,1:256,3);
uint32 rgb2[256,256];%Define a 256*256 matrix with a depth of 32 bits
rgb2=bitor(bitshift(uint32(r),16),bitshift(uint32(g),8));
rgb2=bitor(rgb2,uint32(b));%Perform an OR operation to fill the pixel values into
the matrix fid0 = fopen('image.txt', 'wt');%Open a file
fprintf(fid0, '%06x,\n', rgb2');%Output the matrix as a 6-bit hexadecimal number and fill it with 0
fid0 = fclose(fid0);%Close the file

zero=zeros(256,256);% Clear the display area
R=cat(3,r,zero,zero);%R component image
G=cat(3,zero,g,zero);
B=cat(3,zero,zero,b);
RGB=cat(3,r,g,b);%Original image
%Display
imagesubplot (2,2,1),imshow(R),title('Red component');
subplot(2,2,2),imshow(G),title('Green component');
subplot(2,2,3),imshow(B),title('Blue component');
subplot(2,2,4),imshow(RGB);
```

After the run is completed, a text file image.txt will be generated. Add the following to the beginning of the file:

Two lines, so that Vivado can recognize the initialization file when initializing the ROM

```
MEMORY_INITIALIZATION_RADIX=16;
```

```
MEMORY_INITIALIZATION_VECTOR=
```

```
1  MEMORY_INITIALIZATION_RADIX=16;
2  MEMORY_INITIALIZATION_VECTOR=
3
4  ffffff,
5  ffffff,
6  ffffff,
7  ffffff,
8  ffffff,
9  ffffff,
10 ffffff,
11 ffffff,
12 ffffff,
13 ffffff,
14 ffffff,
15 ffffff,
16 ffffff,
17 ffffff,
18 ffffff,
19 ffffff,
20 ffffff,
21 ffffff,
```

After clicking Save, change the suffix of the txt text data to .coe, so that Vivado

The tool can initialize RAM and ROM correctly when initializing them.

Figure 15- 2

13.2 Create a project

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is ram_pp, the project path is E:\demo\FPGA\hdmi_img. Check Createproject subdirectoryÿNext
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device: Purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

13.3 Create **ROM IP**

In the previous chapter, we introduced how to use RAM. In this chapter, we will introduce another on-chip storage resource.

Source ROM. ROM literally means Read Only Memory.

During the process, ROM can be used to save some small amount of fixed content.

Click IP Catalog, enter rom, select Block Memory Generator, and enter the IP configuration interface.

noodle.

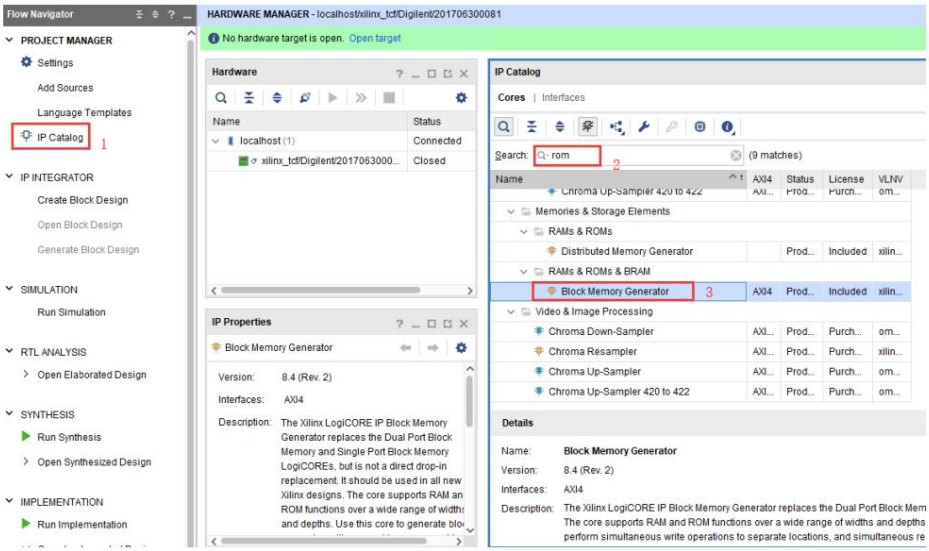


Figure 15- 3

On the Basic page, select Memory type as Single-port ROM;

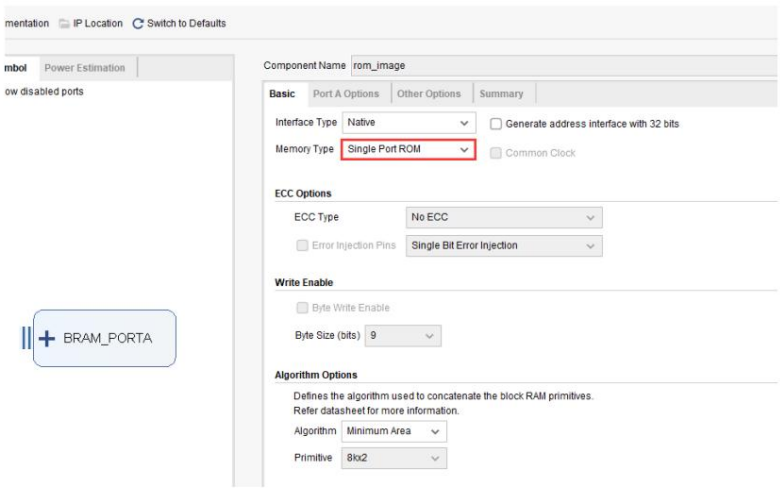


Figure 15- 4

Set the data width to 24 bits, the depth to 65536, and the port enable type to always be enabled;

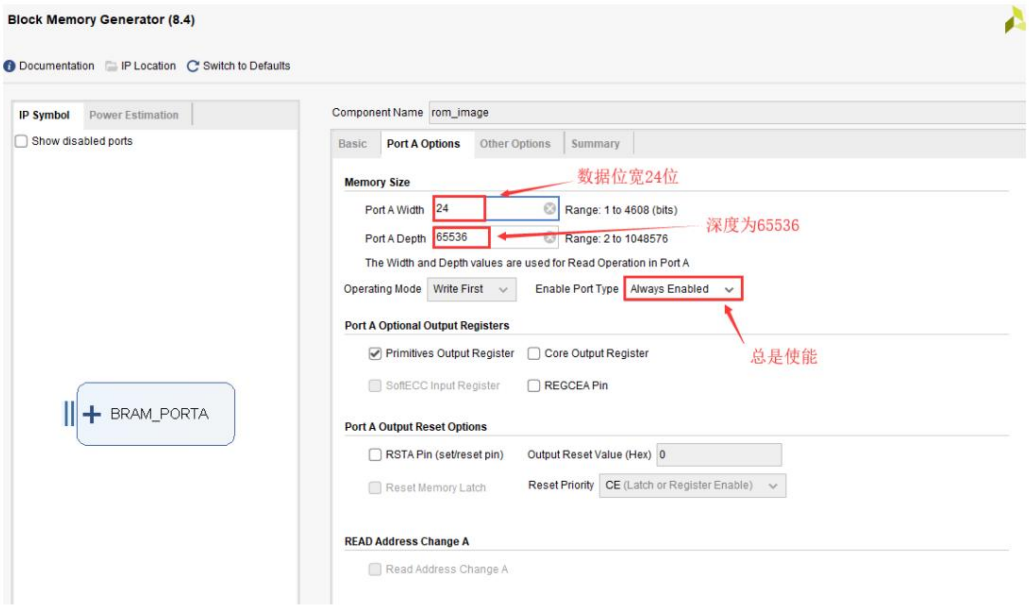


Figure 15- 5

In the other settings, select Load initialization file, and then select the coe file previously generated by MATLAB.

If everything is correct, it will be as shown in the figure below. If there is an error, it will be red.

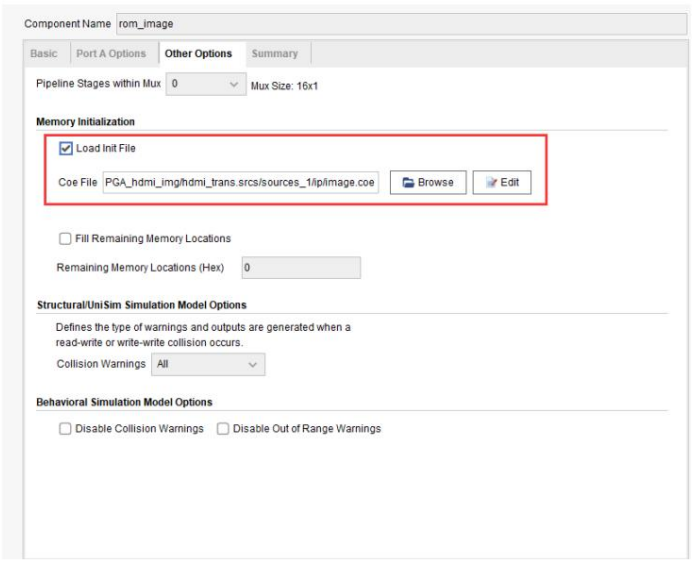


Figure 15- 6

After the configuration is complete, check the configuration information. From the summary, you can see that a total of 44 36K

BRAM, Latency is 2, level means that the data read from ROM is slower than the clock cycle of reading the data.

The rising edge of the period is delayed by two clock cycles.

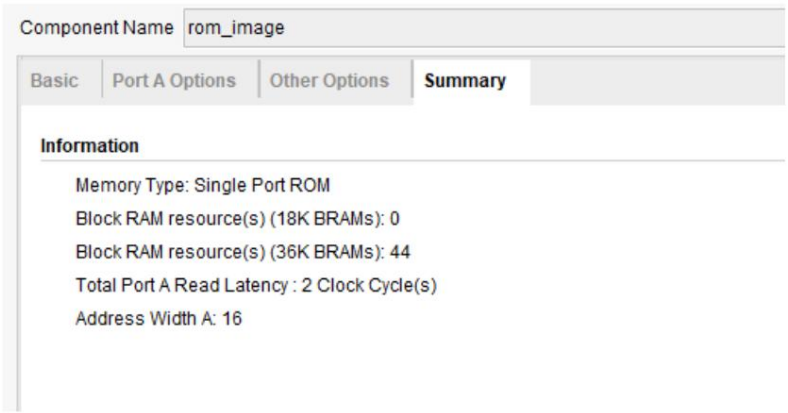


Figure 15- 7

13.4 Overall structural design

This experiment is based on the previous experiment, adding a ROM to provide output image data.

You only need to make simple additions, deletions, and modifications to the project from the last experiment. The overall structure is as follows:

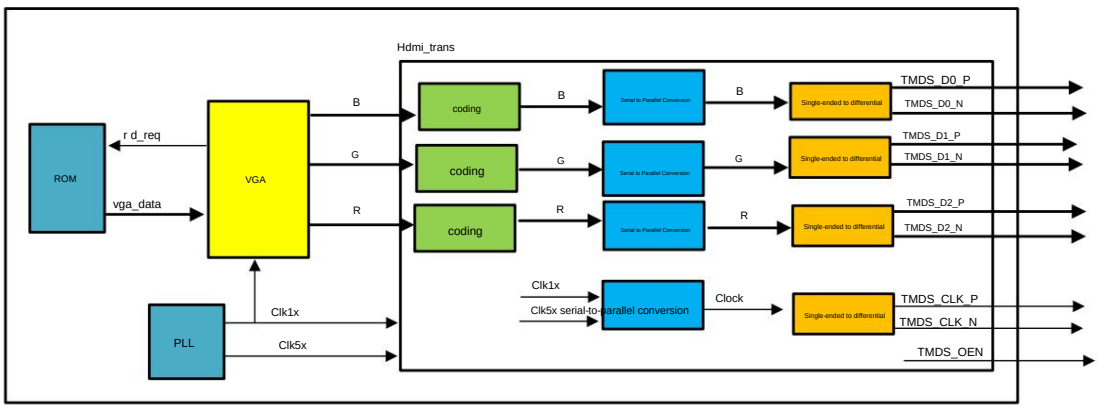


Figure 15- 8

1) ROM reading module

The task implemented by the ROM read module is to, upon receiving the read request passed by the VGA module,

Read the data from the ROM and send it to the VGA module to output it to the display.

The code is as follows:

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

```
module rd_image(
input wire input                    clk     ,
wire                                rst     ,
input wire                          rd_req ,
output wire [23:0] rd_data );


//=========================================
//parameter define
//=========================================
parameter           STOP_ADDR = 256*256 - 1;

reg [15:0]                  rd_addr ;
wire [23:0]                 doubt           ;

assign rd_data = dout;
//----------rd_addr----------
always @(posedge clk) begin
if (rst==1'b1) begin
        rd_addr <= 'd0;
end
else if(rd_req==1'b1 )begin //if(rd_addr == When a read data request comes, the read address is incremented by one.
        STOP_ADDR)begin
                rd_addr <= 'd0;
        end
        else begin
                rd_addr <= rd_addr + 1'b1;
        end
end
end


rom_image inst_rom (
.clka(clk),             // input wire clka
.addra(rd_addr), // input wire [15 : 0] addra
.douta(dout) // output wire [23 : 0] douta
);
endmodule
```

2) VGA module modification

In the VGA module, the original module needs to be modified, and a new read data

The signal is used to read data from the ROM.

Compared to the clock that reads the data, there is a delay of two clock cycles, so if you want to display the image in the square

When the read request signal is given two beats in advance.

It can be simply described by the following timing diagram. When the line counter counts to the position of the displayed image data,

At this time, the output data should be the data read from the ROM, but since the data in the ROM is read

There is a two-beat delay, so a read request needs to be issued two beats in advance to ensure data alignment.

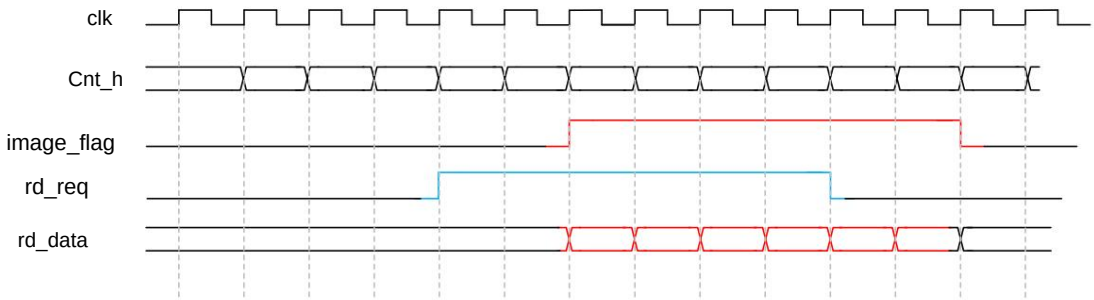


Figure 15- 9

13.5 Download Verification

After generating the bit file, download the bit file, connect the HDMI cable, and display it on the screen.

See that the content of the moving square in the final output becomes the output image.

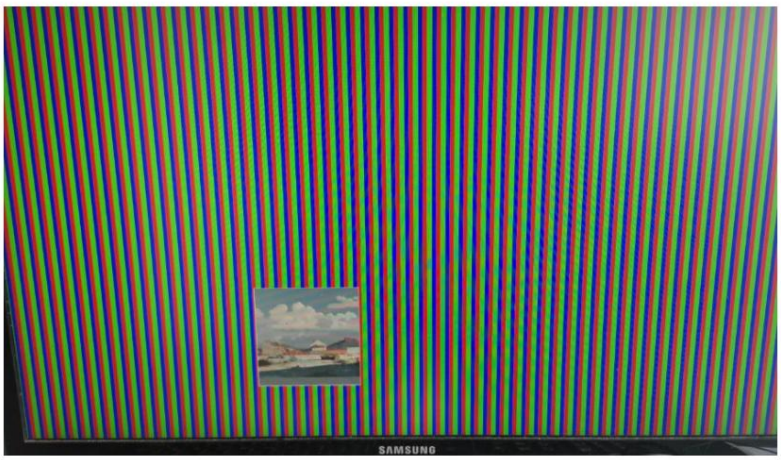


Figure 15- 10

13.6 Experimental Summary

This experiment is based on the previous experiment, adding a ROM to store the data of an image. Finally, the image data is output to the display.

For image storage, we use the on-chip storage resources, which are relatively scarce.

This experiment only stores images of size 256*256, which has occupied the ZYNQ7000 series.

1/3 of the on-chip storage resources of the 7020 chip.

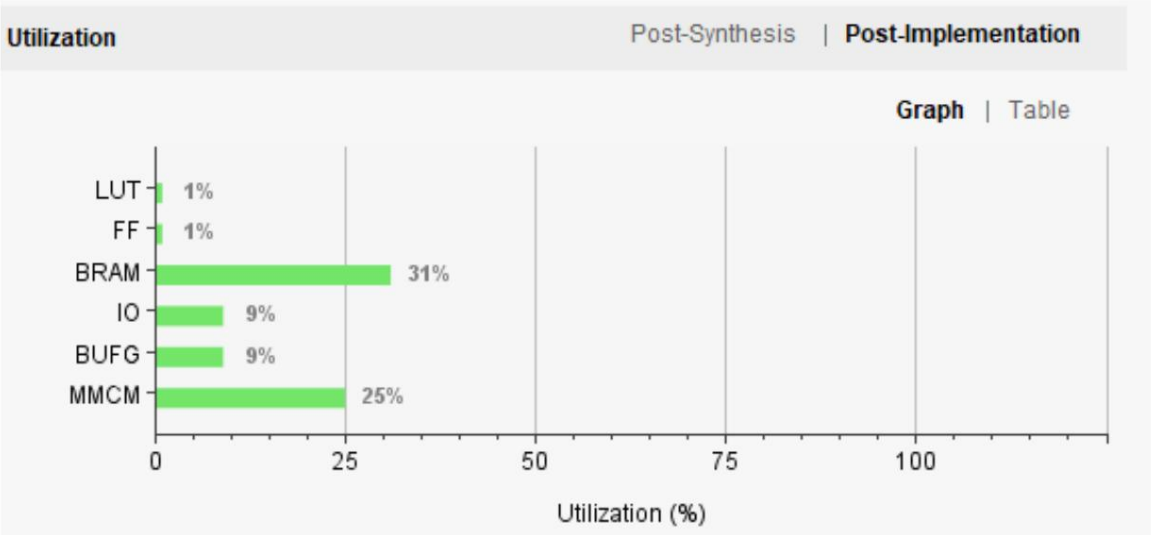


Figure 15- 11

In the following part, DDR will be used for data buffering. At that time, some simple Single image processing related content.

Chapter 14 AD_DA Conversion Module

14.1 Experimental Introduction

1) Board Introduction

In this chapter, we use the ADDA daughter card module of Microphase Technology to complete a simple AD_DA conversion. The high-speed ADDA daughter card is a high-speed ADDA data conversion card, which contains a high-speed ADC chip (ADC08060 produced by TI), one high-speed DA conversion chip (produced by TI DAC908E). This daughter card can be used for data acquisition, signal generation, DSP and other applications.

The connection diagram between the daughter card and GPIO is shown in the following figure:

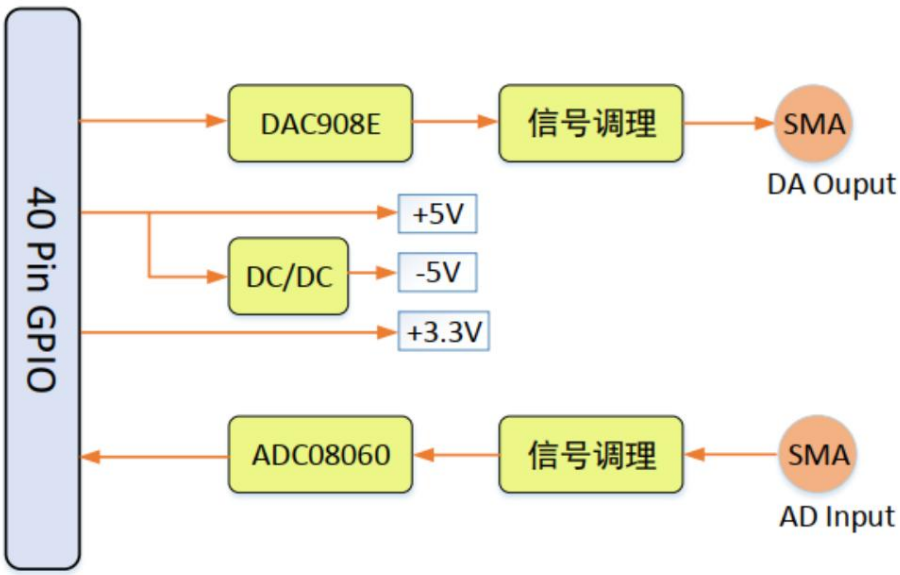


Figure 16- 1

2) High-speed **DA** conversion module

The structure of the DA conversion chip used by the high-speed ADDA daughter card is shown in the figure below:

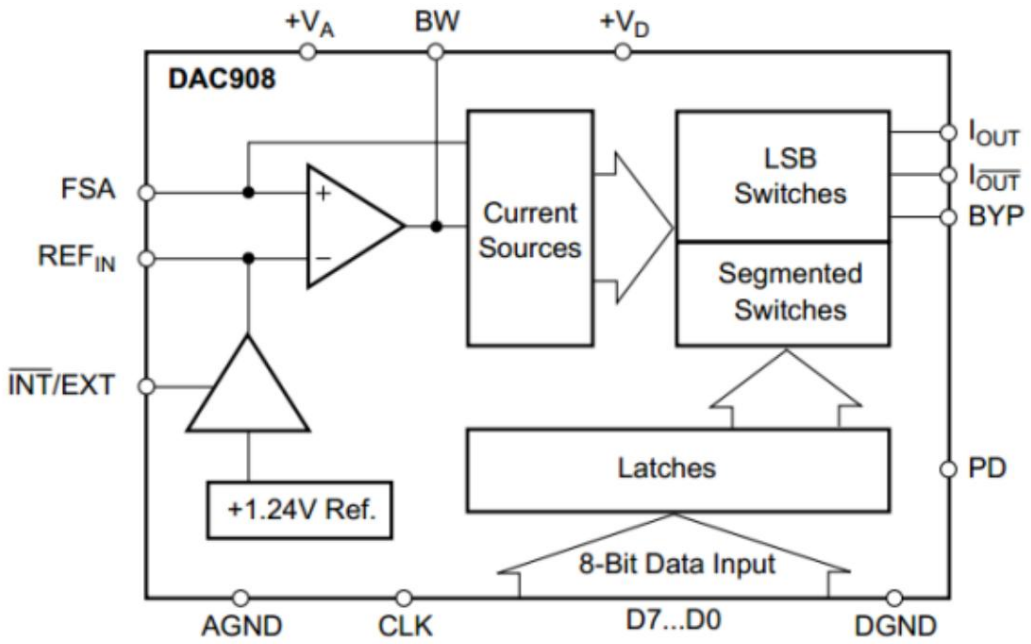


Figure 16- 2

From the dataset of this chip, we can see its reference design as shown below:

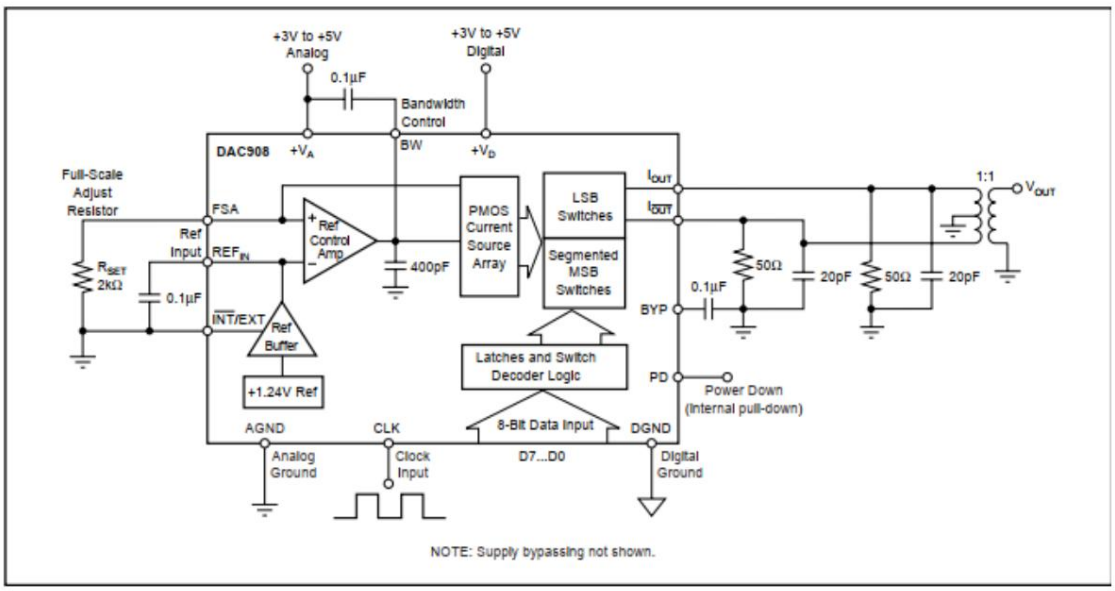


Figure 16- 3

From the reference design, we can see that the CLK signal is the input signal given to the chip, 8bit

The data is given to the chip for digital-to-analog conversion. In addition, we also need to pay attention to the operation of the chip.

The specific timings that need attention are as follows:

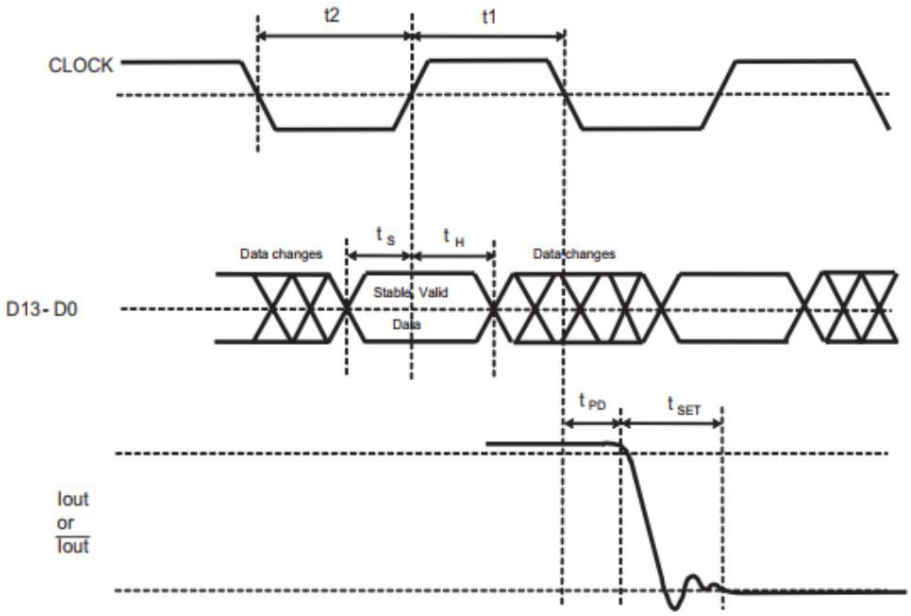


Figure 16- 4

The timing parameter list is shown in the figure below. When we design FPGA, we need to

These parameters are used to design the relationship between output data and clock.

SYMBOL	DESCRIPTION	MIN	TYP	MAX	UNITS
t_1	Clock Pulse HIGH Time		3		ns
t_2	Clock Pulse LOW Time		3		ns
t_S	Data Setup Time		1.0		ns
t_H	Data Hold Time		1.5		ns
t_{PD}	Propagation Delay Time		1		ns
t_{SET}	Output Settling Time to 0.1%		30		ns

Figure 16- 5

3) High-speed **AD** conversion module

The structure of the chip ADC08060 used in the high-speed AD conversion module is as follows:

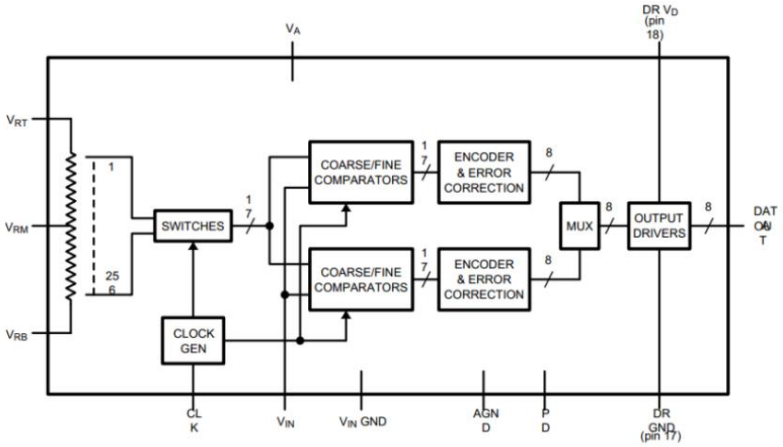


Figure 16- 6

From the structure, we can see that the working clock of the chip is also provided by the outside.

The clock is converted through a PLL-like structure for sampling, and finally an 8-bit digital signal is output.

Number.

The timing diagram is shown below:

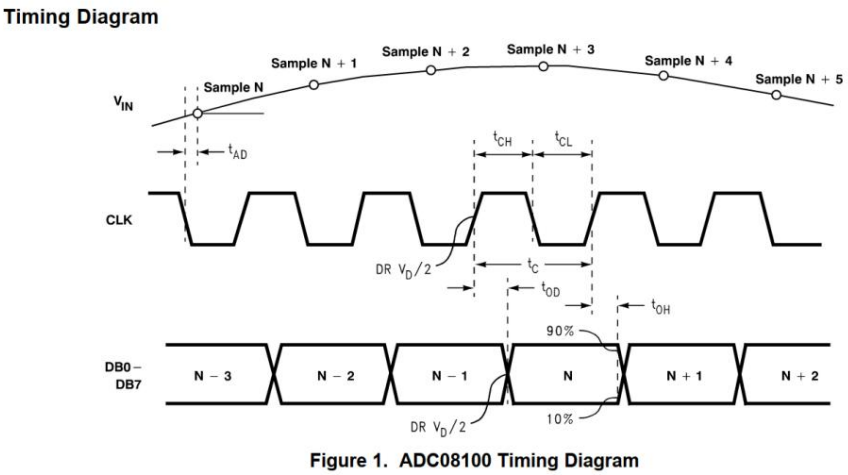


Figure 1. ADC08100 Timing Diagram

Figure 16- 7

The timing parameters are shown in the figure below. In actual design, if you want a higher speed, you can use the following

Timing parameters are used to design and modify timing.

AC ELECTRICAL CHARACTERISTICS					
f_{C1}	Maximum Conversion Rate		125	**100**	MHz (min)
f_{C2}	Minimum Conversion Rate		20		MHz
t_{CL}	Minimum Clock Low Time			**4.5**	ns (min)
t_{CH}	Minimum Clock High Time			**4.5**	ns (min)
t_{OH}	Output Hold Time	CLK Rise to Data Invalid	4.4		ns
t_{OD}	Output Delay	CLK Rise to Data Valid	5.9	**8.5**	ns (max)
	Pipeline Delay (Latency)		2.5		Clock Cycles
t_{AD}	Sampling (Aperture) Delay	CLK Fall to Acquisition of Data	1.5		ns
t_{AJ}	Aperture Jitter		2		ps rms

Figure 16- 8

14.2 Create a project

Follow the steps shown in the table to create a project:

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is pcie_test, the project path is E:\demo\adda_test, check Createproject subdirectory, Next
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device to purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

14.3 Generate **coe** file using **MATLAB**

Before conducting this experiment, you need to generate a rom initialization file. Execute in matlab

The following script.

```
X=linspace(0,2*pi,512);
Y=(255/2)*(sin(X)+1);
Y=uint8(Y);
plot(X,Y);
fid0 = fopen('sin512.txt', 'wt');%                Open a file
```

```
fprintf(fid0, '%02x,\n', Y);%fid0 = fclose(fid0);The matrix is sorted by    16    Binary number output, padding    0    Placeholder
```

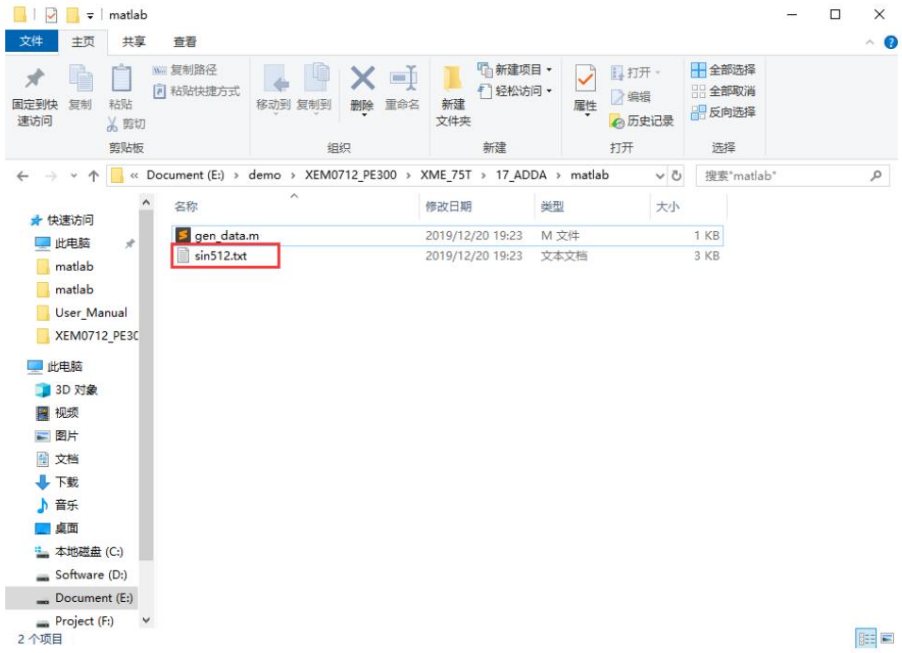


Figure 16- 9

After the execution is completed, a text data in TXT format will be generated. We need to add the following to the text:

The following content:

MEMORY_INITIALIZATION_RADIX=16ÿ

MEMORY_INITIALIZATION_VECTOR= As shown in the following figure:

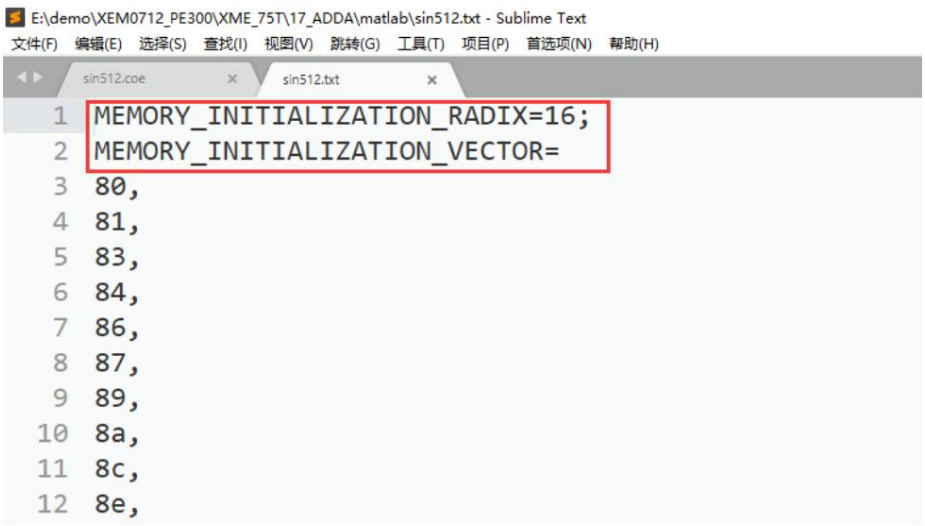


Figure 16- 10

After the change is completed, we need to change the suffix of the txt format file to .coe.

14.4 Initializing **RAM**

1) Initialize **ROM**

Next, we will use a ROM to store the waveform data, so that we can

Read to generate a waveform.

Click IP catalog to enter the IP generation page:

Type rom in the search bar and select block Memory Generator below.

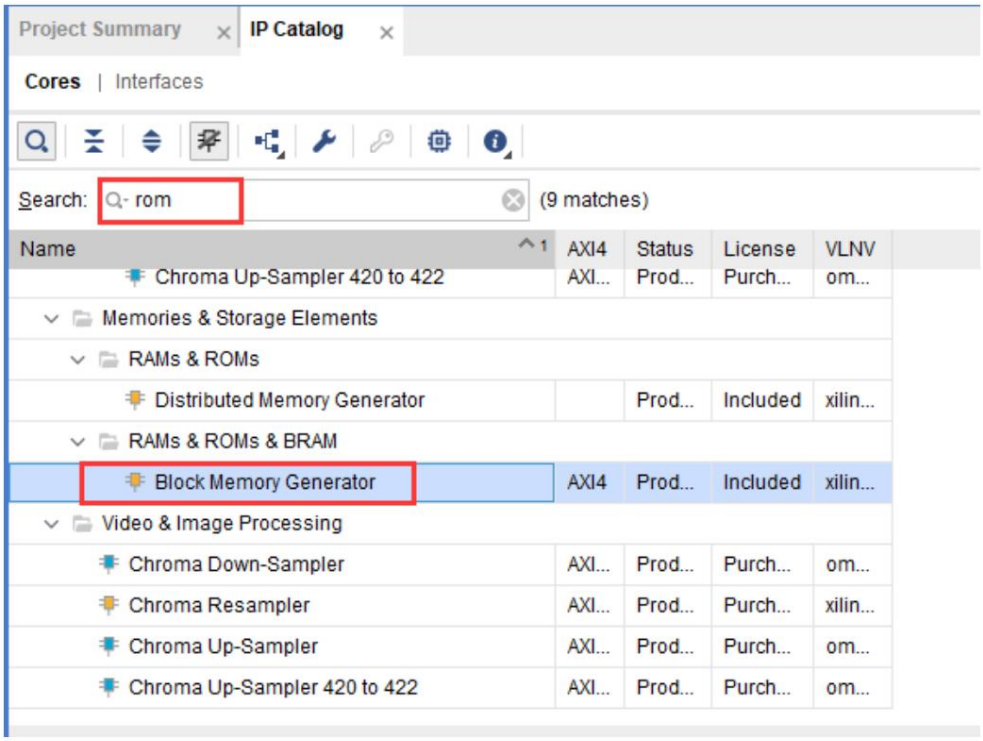


Figure 16- 11

On the basic configuration page, name the RAM to be generated and select single-port ROM;

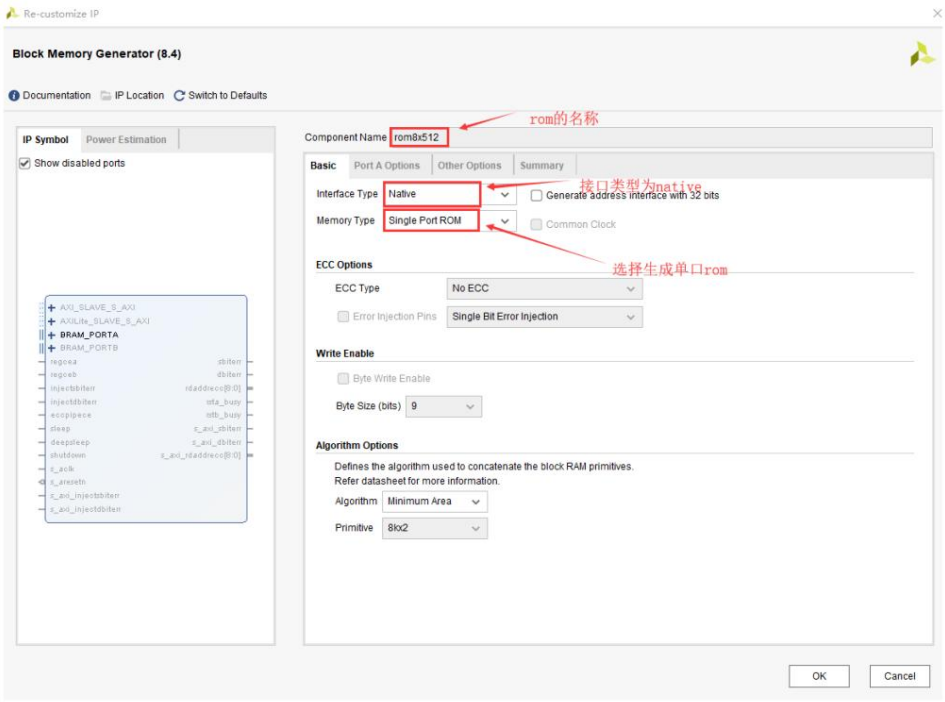


Figure 16- 12

On the configuration page of port A, you need to configure the bit width and depth of the ROM, and configure the depth to 512

Mode with a bit width of 8.

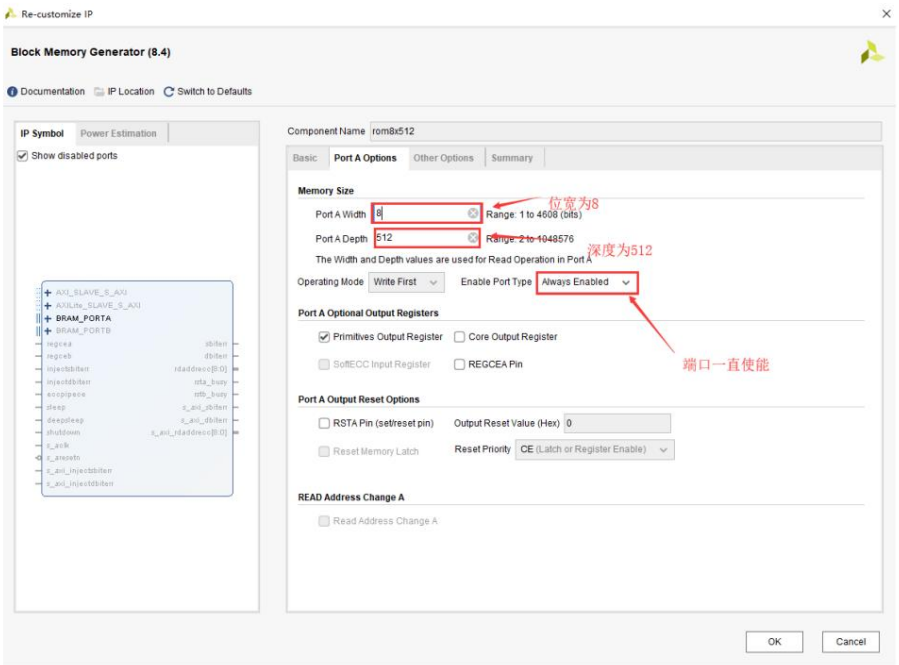


Figure 16- 13

Then add the import initialization file coe file

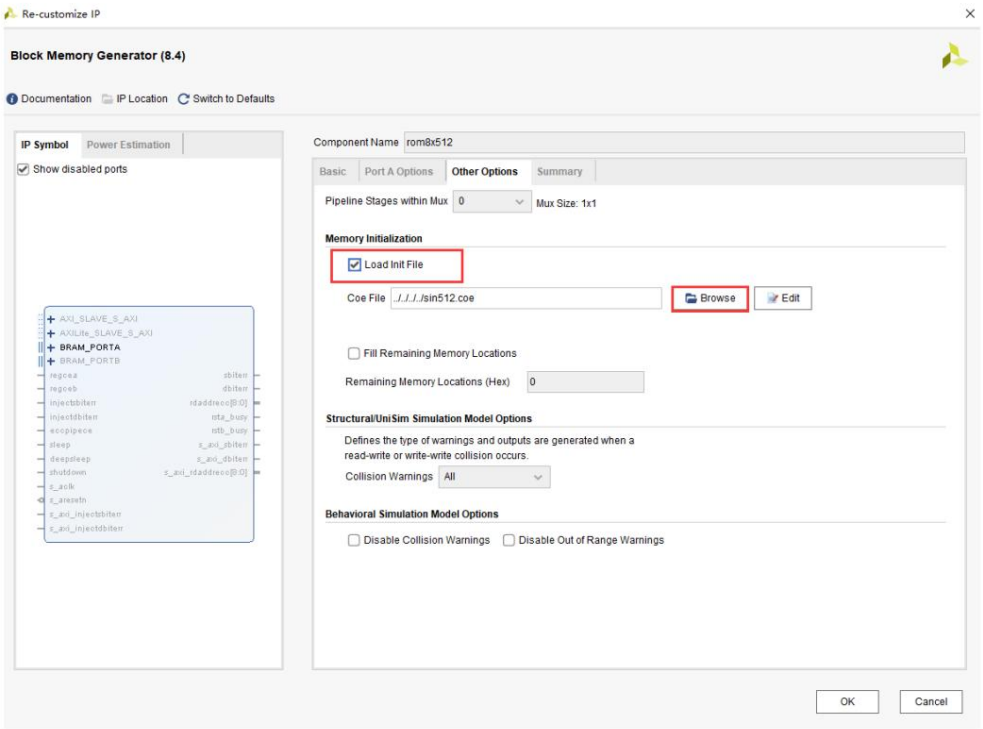


Figure 16- 14

Select the coe file just generated in the matlab folder.

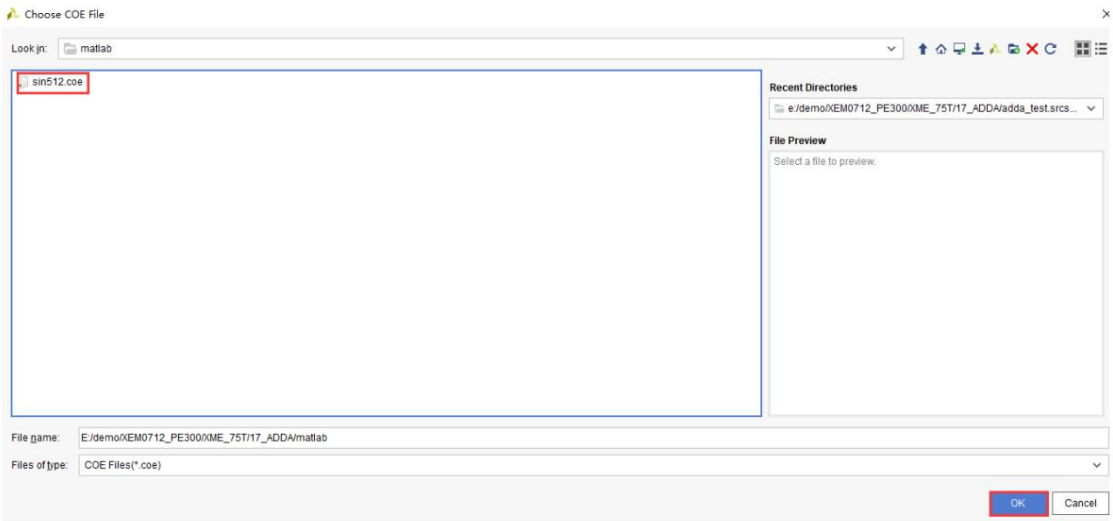


Figure 16- 15

Now the configuration of ROM is completed, and next we need to generate a PLL.

2) Configure the clock **IP**

One of the output clocks is configured to 50MHz and the other is configured to 20MHz. Follow the steps below

to perform the operation.

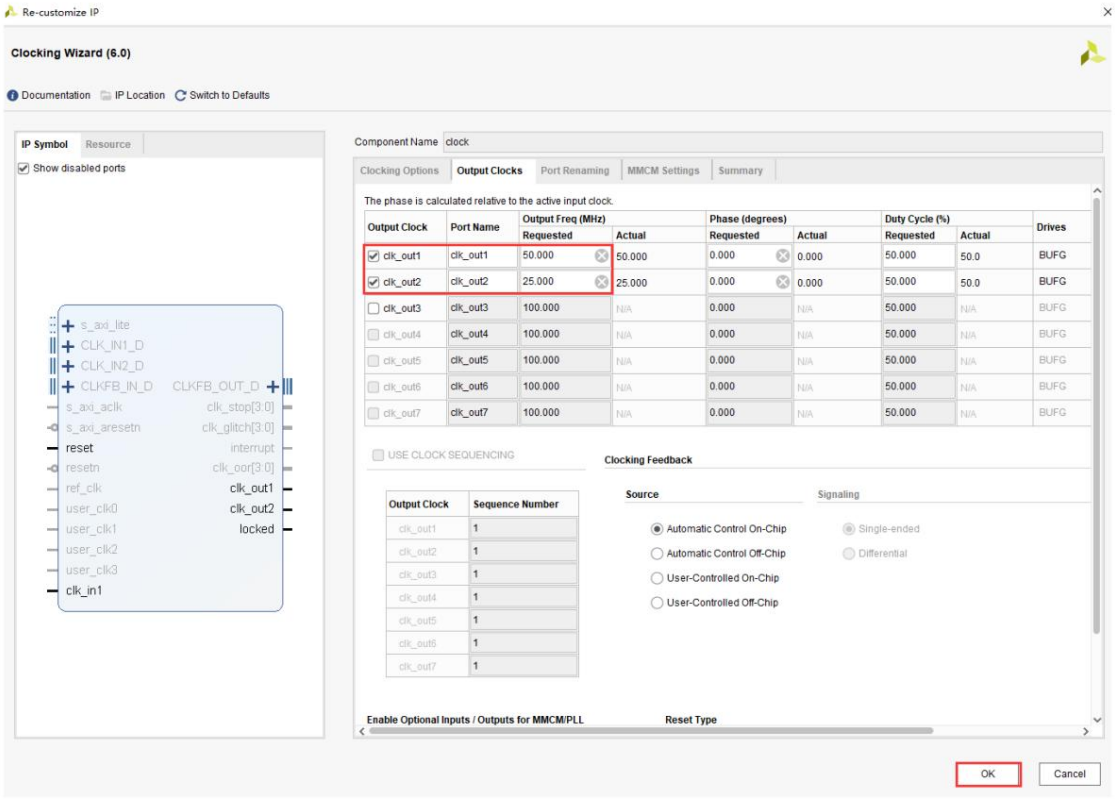


Figure 16- 16

3) Add **ILA** to observe the results

ILA is an embedded logic analyzer. It can use the internal resources of FPGA to observe

FPGA internal signals. This will bring great convenience during debugging.

This experiment uses two ILAs, one for observing the data output from the FPGA during DA conversion.

One is used for the data collected inside the FPGA after AD conversion.

Click IP Catalog ÿ Search for ila ÿ Select ila as shown in the figure to enter the configuration interface.

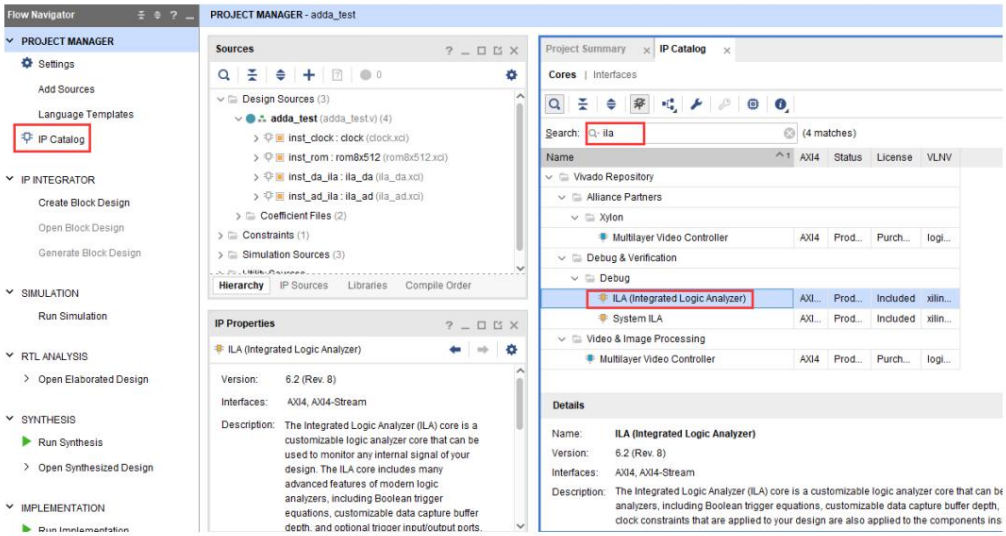


Figure 16-17

In the IP configuration interface, select the interface type as native, set the number of probes to 1, and the sampling The depth is set to 1024; then the probe bit width is set to 8, that is, the bit width of the signal to be observed is 8 bits of.

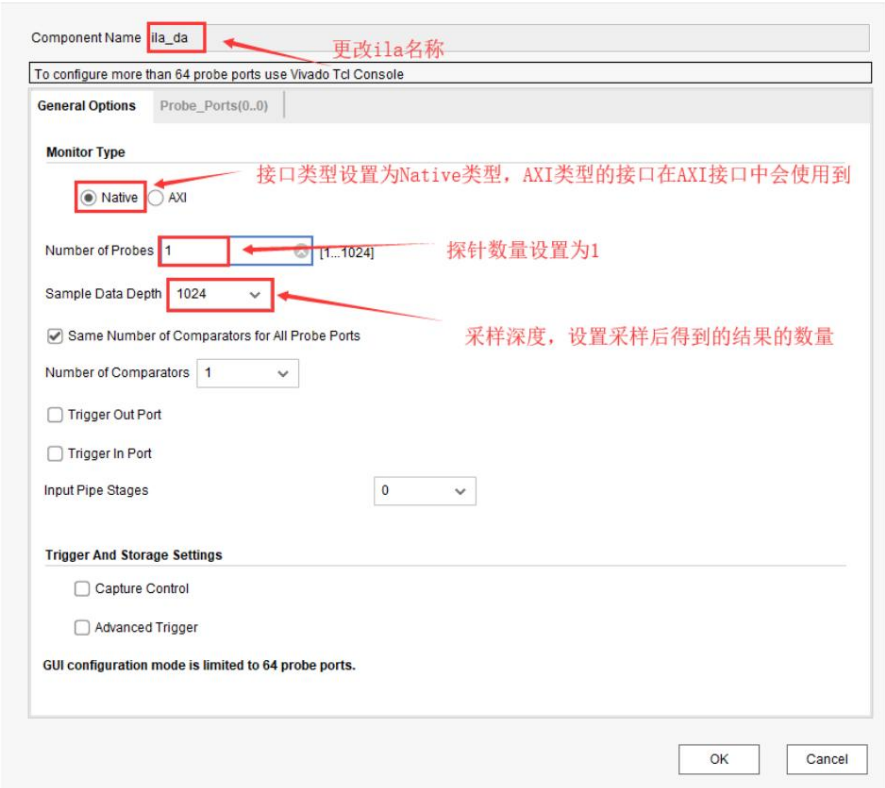


Figure 16-18

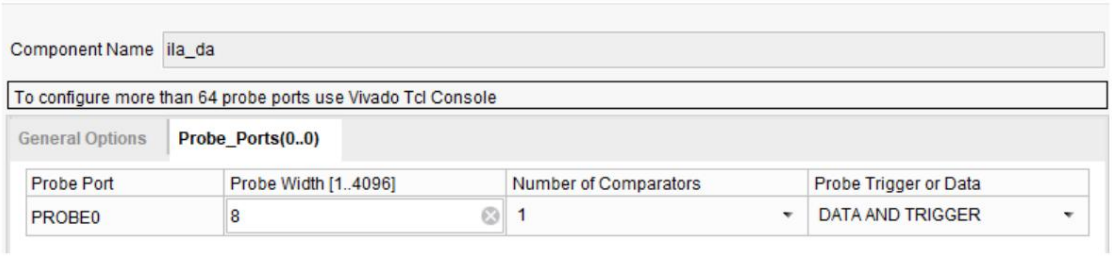


Figure 16-19

The two ila in this experiment can be configured in this way.

14.5 Programming

This experiment is relatively simple, mainly through the ila to observe the input and output waveforms, the process of this experiment

The sequence design is as follows:

```
1 `timescale 1ns / 1ps
 2 module add_test(
 3 input wire 4 input                clk          ,
 wire                                rst_n        ,
 5
 6 output wire 7 input               ad_clk       ,// ad   Module clock
 wire [7:0]ad_data ,//ad                    The converted data
 8
 9 output wire                       da_clk //    ,// and  Module output clock
10 output wire [7:0]da_data                 da     Module output data
11          );
12
13 wire              clk_50m ;
14 wire              clk_25m ;
15 wire              sys_clk ;
16 wire              locked      ;
17 reg [8:0]rom_addr             ;
18
19
20 assign          ad_clk = clk_25m ;
21 assign          da_clk = clk_50m ;
22
23 clock inst_clock(
```

```
24                                          // output clk_out1
25                                          // output clk_out2
26          .clk_out1(clk_50m), .clk_out2(clk_25m), .reset(~rst_n), // input reset
27          .locked(locked), // output locked
28          .clk_in1(clk) );            // input clk_in1
29
30 always @(posedge clk_50m or negedge rst_n) begin
31 if (rst_n==1'b0) begin
32          rom_addr <= 'd0;
33 end
34 else begin
35          rom_addr <= rom_addr + 1'b1;
36 end
37 end
38
39
40
41 rom8x512 inst_rom (
42 .clka(clk_50m), // input wire clka
43 .addra(rom_addr), // input wire [8 : 0] addra
44 .douta(da_data) // output wire [7 : 0] douta
45 );
46
47 ila_da inst_da_ila (
48 .clk(clk_50m), // input wire clk
49
50
51 .probe0(da_data) // input wire [7:0] probe0
52 );
53
54 ila_ad inst_ad_ila (
55 .clk(clk_25m), // input wire clk
56
57
58 .probe0(ad_data) // input wire [7:0] probe0
59 );
60 endmodule
```

14.6 Adding Constraint Files

The constraint files for this experiment are as follows:

```
############### clock define###################
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

############### key define###################
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]

set_property PACKAGE_PIN W19 [get_ports {da_data[0]}]
set_property PACKAGE_PIN W18 [get_ports {da_data[1]}]
set_property PACKAGE_PIN U17 [get_ports {da_data[2]}]
set_property PACKAGE_PIN T16 [get_ports {da_data[3]}]
set_property PACKAGE_PIN R17 [get_ports {da_data[4]}]
set_property PACKAGE_PIN R16 [get_ports {da_data[5]}]
set_property PACKAGE_PIN P18 [get_ports {da_data[6]}]
set_property PACKAGE_PIN N17 [get_ports {da_data[7]}]
set_property PACKAGE_PIN Y18 [get_ports da_clk]

set_property SLEW FAST [get_ports {da_data[*]}]
set_property SLEW FAST [get_ports da_clk]

set_property IOSTANDARD LVCMOS33 [get_ports {da_data[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports da_clk]

set_property PACKAGE_PIN U12 [get_ports ad_clk]
set_property PACKAGE_PIN V13 [get_ports {ad_data[0]}]
set_property PACKAGE_PIN U13 [get_ports {ad_data[1]}]
set_property PACKAGE_PIN U15 [get_ports {ad_data[2]}]
set_property PACKAGE_PIN U14 [get_ports {ad_data[3]}]
set_property PACKAGE_PIN R14 [get_ports {ad_data[4]}]
set_property PACKAGE_PIN P14 [get_ports {ad_data[5]}]
set_property PACKAGE_PIN W15 [get_ports {ad_data[6]}]
set_property PACKAGE_PIN V15 [get_ports {ad_data[7]}]

set_property SLEW FAST [get_ports ad_clk]

set_property IOSTANDARD LVCMOS33 [get_ports ad_clk]
set_property IOSTANDARD LVCMOS33 [get_ports {ad_data[*]}]
```

Next, you can generate the bit file.

14.7 Download Verification

After generating the bit file, we connect the ADDA daughter card to the Z7-Lite development board as follows:

Download the bit file to FPGA, grab the data through ila, and observe the output waveform.

Enter the waveform window, convert the data display format to analog form, and then observe the waveform, as shown below

picture:

Figure 16- 20

After the bit file is downloaded, a waveform window will pop up on the right, in which you can see

The signal you want to sample and observe. The red area is the area that can be used to collect when using ila for analysis.

And the waveform observation button, the yellow area is the current trigger status and the setting of the number of trigger point sampling windows.

The purple area can be used to set trigger conditions. The waveform will be sampled only when the conditions are met.

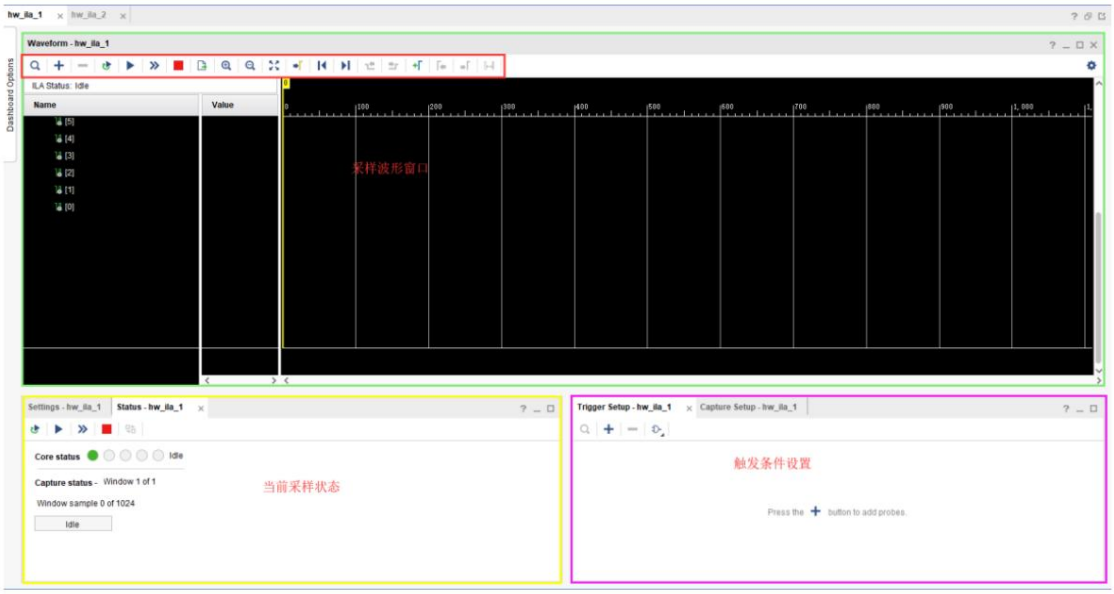


Figure 16- 21

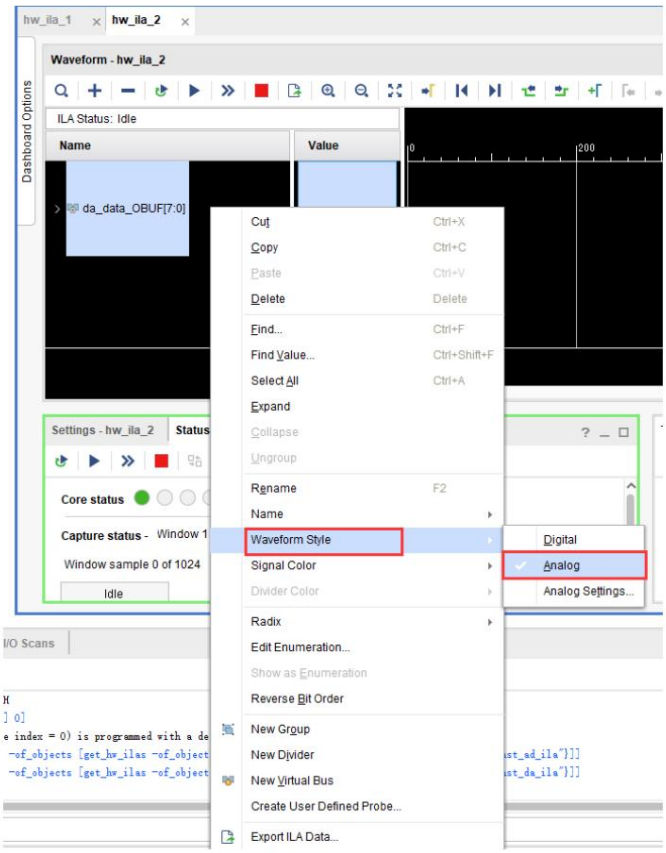
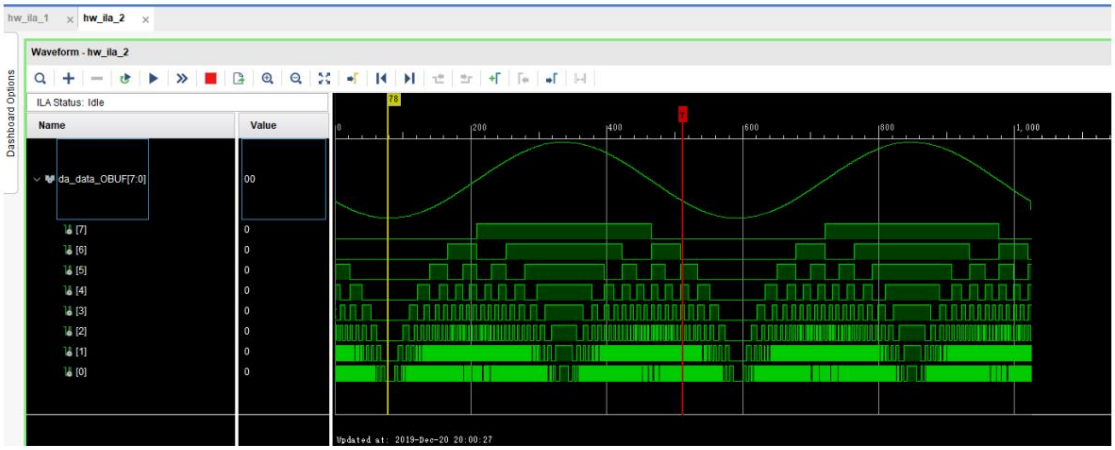


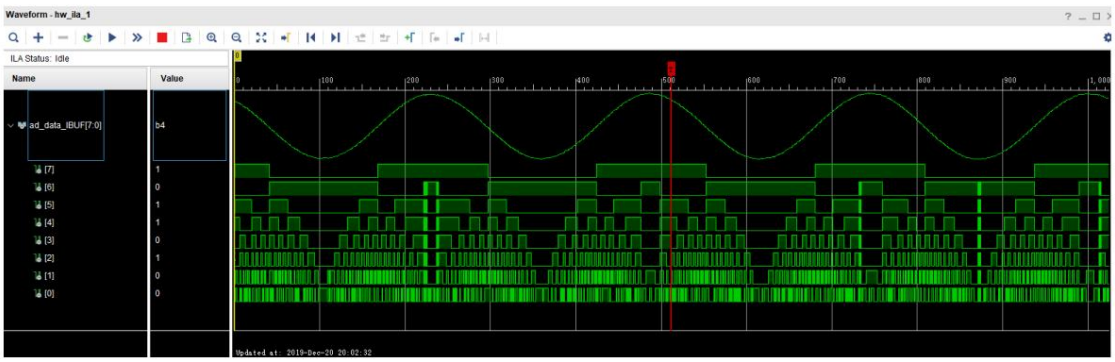
Figure 16- 22

Click the trigger button and you can see the waveform of da_data output from FPGA as shown in the figure below.

is a sine wave.



Then observe the ad_data after ad conversion, and you can see that the input analog quantity has been received correctly.



So far, a simple AD_DA conversion is completed.

14.8 Experimental Summary

In this experiment, the ADDA daughter card was used to complete the output of the sine wave and the sampling of the sine wave.

To digital quantity experiment.

Through this experiment, I have a rough understanding of how to use the following ADDA daughter cards, as well as the two digital

I have a certain understanding of analog conversion chips.

Chapter 15 ADDA Conversion and **HDMI** Display

15.1 Experimental Introduction

In the previous chapter, we introduced the use of the ADDA daughter card and observed the FPGA through ILA.

Internal output and sampled data. In this chapter, we will continue to use the ADDA daughter card to complete the ADDA

The data obtained by AD sampling is converted and displayed on the monitor through HDMI.

15.2 Create a project

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is pcie_test, the project path is E:\demo\adda_hdmi, check Createproject subdirectory, Next
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device, purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

15.3 Functional Design

This experiment wants to display the data collected by the AD conversion module through HDMI. The main task is

In fact, it is converted to how to achieve image superposition. First, let's look at the final display of this experiment.

Effect.

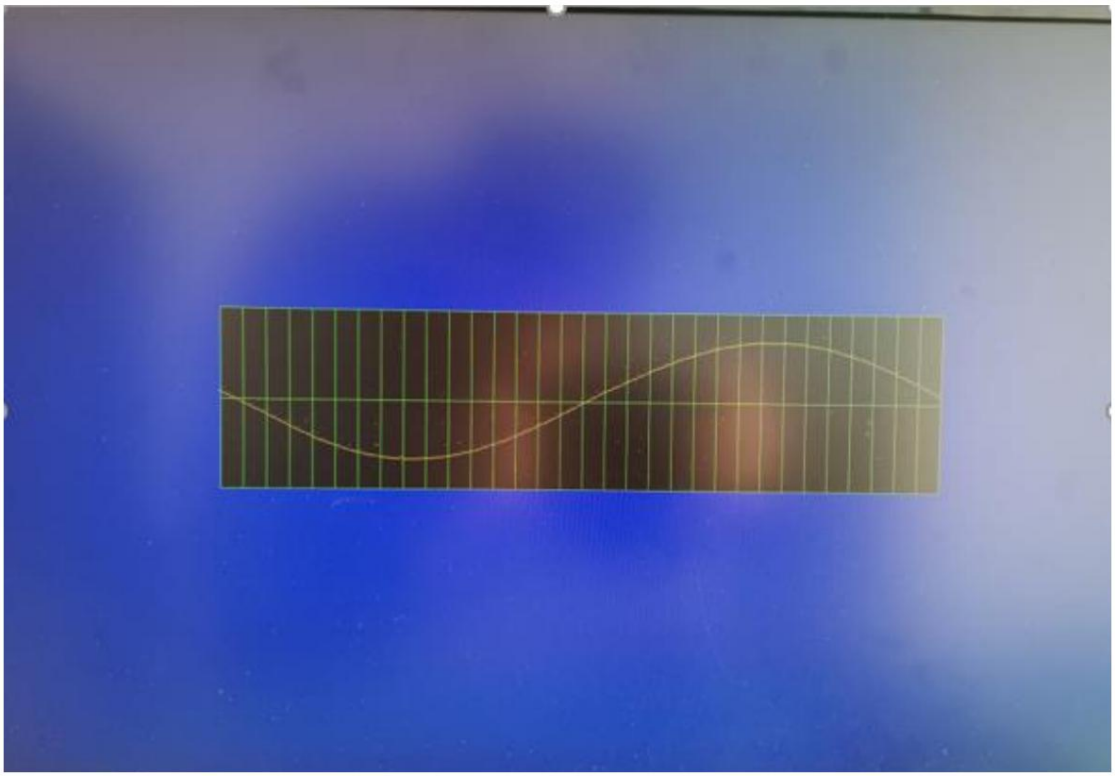


Figure 17- 1

This experiment will display a 1024x256 size waveform in the central area of the HDMI blue background.

The display window shows a grid and a black background, and finally the wave we collected is displayed.

The shape is displayed in this window.

Next, we will introduce what we need to do to achieve this function.

1) DA waveform generation module

The main function of this module is to read the data in the ROM inside the FPGA and transmit this data through the DA

The conversion chip outputs to the outside. The schematic diagram of this module is as follows:

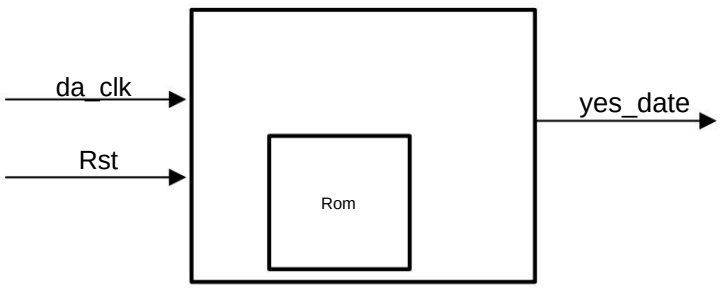


Figure 17- 2

Here, initialize the ROM as follows:

Click IP catalog to enter the IP generation page:

Type rom in the search bar and select block Memory Generator below.

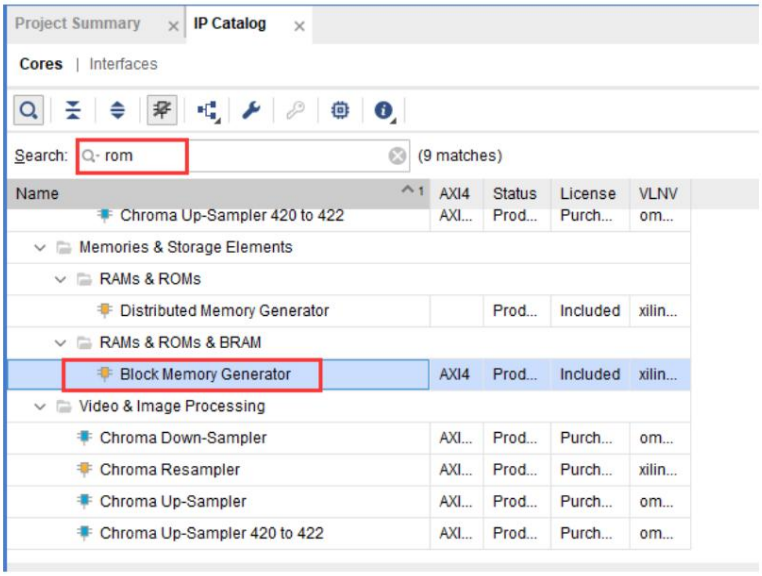


Figure 17- 3

On the basic configuration page, name the RAM to be generated and select single-port ROM;

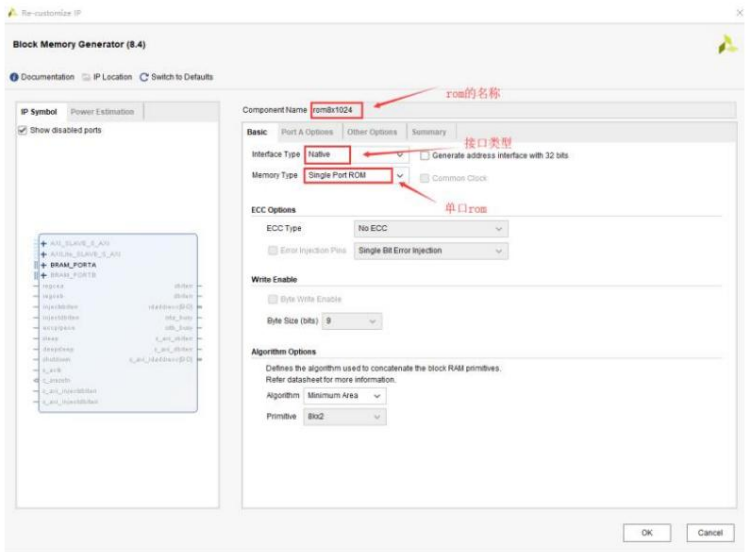


Figure 17- 4

On the configuration page of port A, you need to configure the bit width and depth of the ROM, and configure the depth to 1024.

Mode with a bit width of 8.

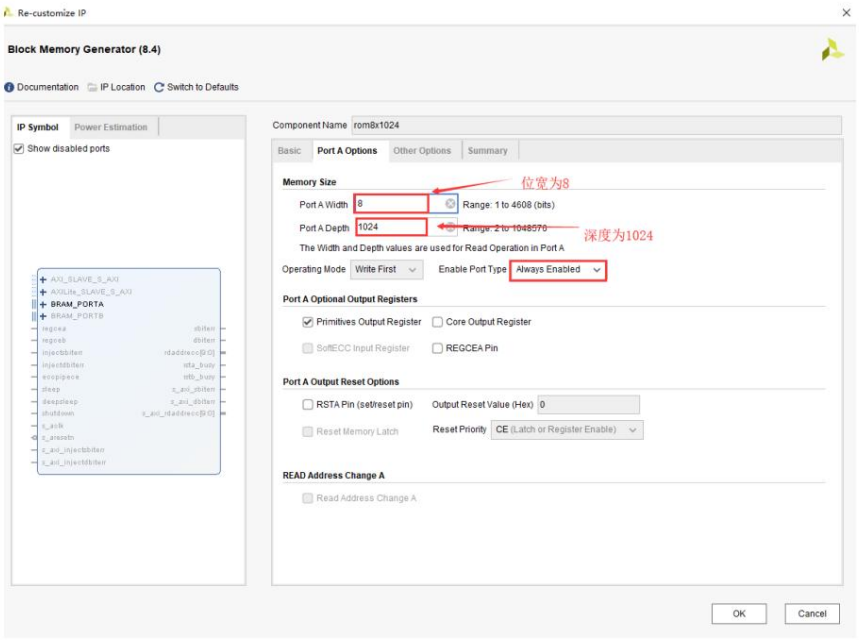


Figure 17- 5

Then add the import initialization file coe file

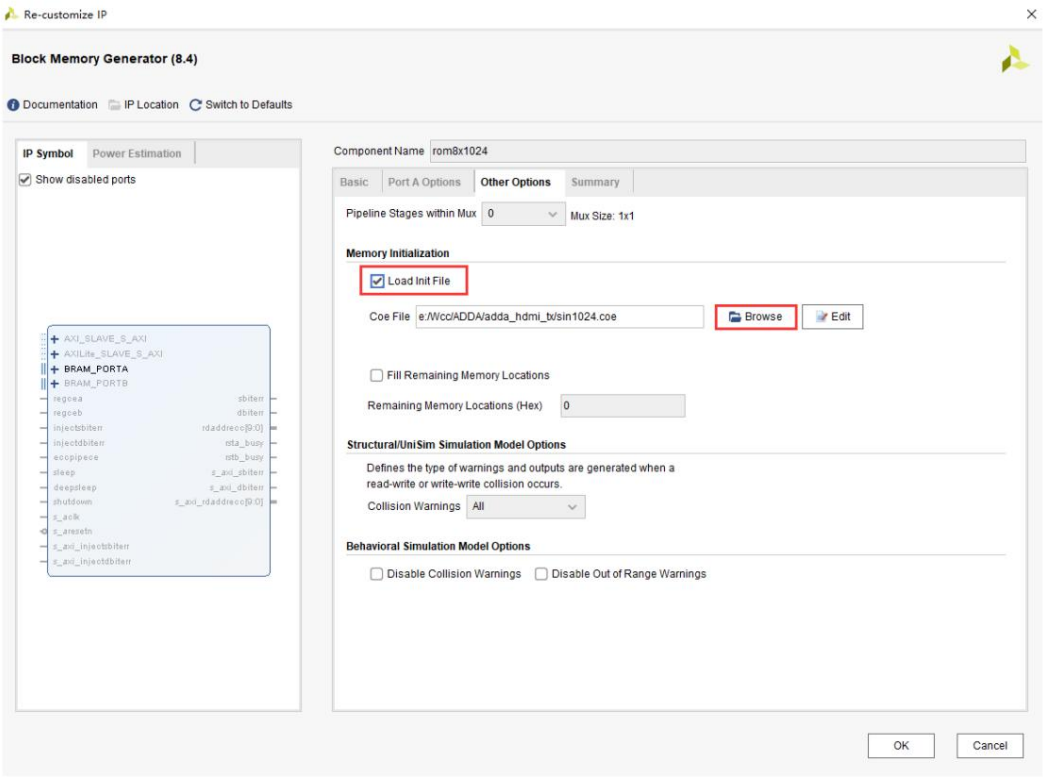


Figure 17- 6

Select the coe file in the project directory.

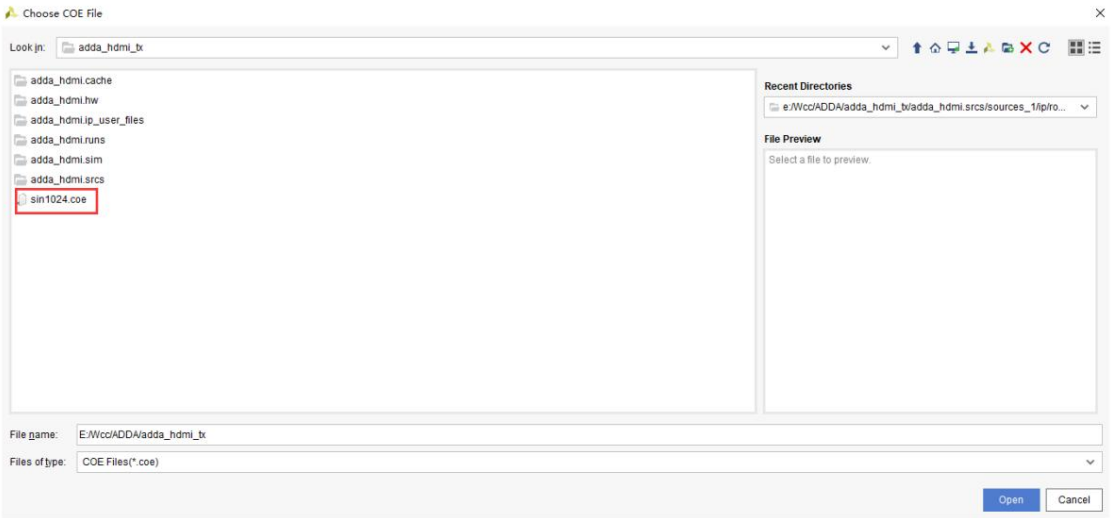


Figure 17- 7

2) Image background module

The main function of this module is to generate the image background of HDMI display, including blue background and waveform window.

The background, border and grid of the port. This module is similar to the content in the previous section on making the HDMI output block.

Very similar.

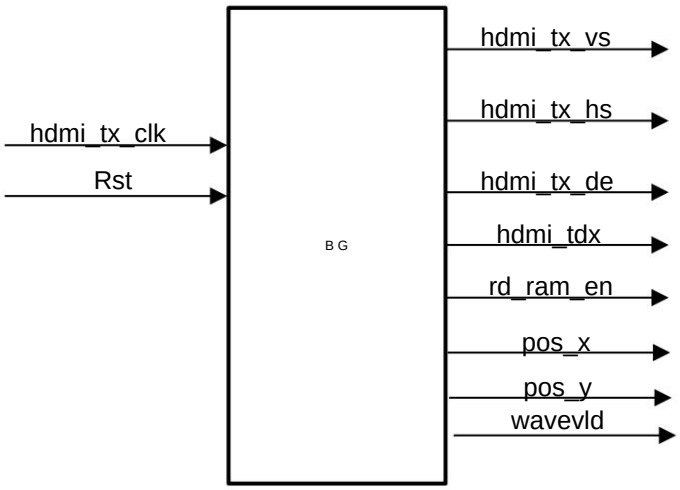


Figure 17- 8

Port Name	I/O Width	Notes	
hdmi_tx_clk		1	Pixel Clock
rst		1	Reset signal
hdmi_tx_vs	THE	1	Field sync signal
hdmi_tx_hs	THE	1	Line sync signal
hdmi_tx_de	THE	1	Output data valid signal
hdmi_td	THE	24	Output background image data
rd_ram_en	THE	1	Read AD data stored in RAM to enable
pos_x	THE	13	Waveform window horizontal axis
pos_y	THE	13	Waveform window vertical coordinate
wave_vld	THE	1	Waveform window valid signal

In this module, the window and window to display the image are mainly controlled by the line counter and field counter.

The grid, border, etc. to be displayed in the image. In simple terms, the time is to adjust the priority of image output so that the image

The image can appear above the background. As shown in the following code:

In the waveform window, when wave_vld is valid, if it is at the boundary position, a green boundary is output.

A black background is output. A blue background is displayed outside the valid area of wave_vld.

It can be seen that it is relatively simple to realize the image superposition of this part.

```
//hdmi_td
always @(posedge hdmi_tx_clk ) begin
    if (rst==1'b1) begin
        hdmi_td <='d0;
    end
    else if(wave_vld)begin
        if(x==0 || y==0 || x==WINDOW_X-1 || y==WINDOW_Y-1 ||(y==(WINDOW_Y>>1)) ||
x[4:0]==5'h1f)
            hdmi_td <= 24'h00ff00; //output image
        else
            hdmi_td <= 'd0;
    end
    else if((cnt_h >= H_START) && (cnt_h < H_END) && (cnt_v >=V_START) && (cnt_v < V_END))begin
        hdmi_td <= 24'h0000ff; //output image
    end
    else begin
        hdmi_td <= 'd0;
    end
end
```

3) Waveform superposition module

In this module, the main function is to collect AD data and superimpose the collected data on the waveform.

It is displayed in the graphical window. It can be seen that a part of this module is used to complete the collection of AD data.

The data received is stored in RAM. The instantiated background image module reads the data in RAM in this module.

The read data is compared with the point of the waveform window currently scanned to determine the waveform.

The results are displayed in the window.

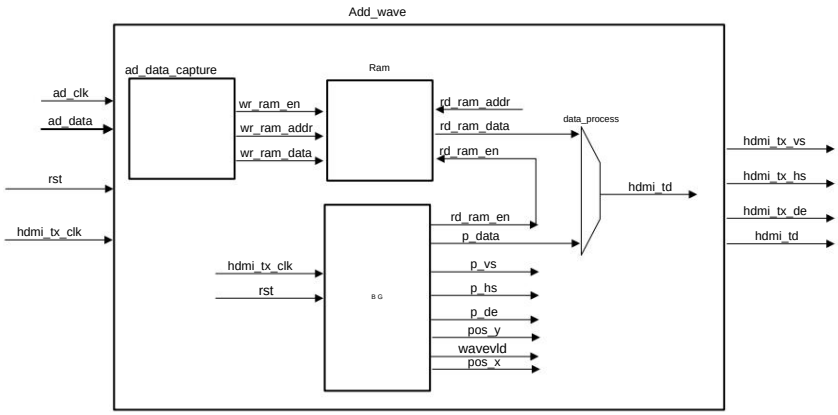


Figure 17- 9

The interface list is shown in the figure below:

Port Name	I/O Width Notes		
hdmi_tx_clk	'	1	Pixel Clock
rst	'	1	Reset signal
hdmi_tx_vs	THE	1	Field sync signal
hdmi_tx_hs	THE	1	Line sync signal
hdmi_tx_de	THE	1 Output data valid signal	
hdmi_td	THE	24 Output background image data	
ad_clk	'	1	AD sampling clock
ad_data	'	8	Data collected by AD

In this module, in order to enable the waveform to be displayed in the window, the image to be output is also prioritized.

The data output from the background image module is compared with whether the position of the waveform point is adjusted in this module.

The waveform is superimposed and finally displayed in the window set previously.

The code of this module is as follows:

```
1 `timescale 1ns / 1ps
2 module wave_mix(
3       input       wire            ad_clk          ,
4 input             wire [7:0] ad_data              ,
5       input wire                  hdmi_tx_clk     ,
6       input       wire            rst             ,
7       output      wire            hdmi_tx_hs      ,
8       output      wire            hdmi_tx_vs      ,
9           output wire                 hdmi_tx_de      ,
10 output           wire [23:0]         hdmi_td
```

```
11          );
12 //=============================================
13 //parameter define
14 //=============================================
15 parameter IDLE =           3'b001;
16 parameter CAPTURE =              3'b010;
17 parameter WAIT          =       3'b100;
18
19 parameter WAIT_MAX = 4000000;
20 parameter DATA_MAX = 1024 - 1;
21 parameter ADDR_MAX = 1024 - 1;
22 parameter AMP_DATA = 256 - 1 ;
23
24
25 //=============================================
26 //internal signals
27 //=============================================
28 reg [9:0] wr_ram_addr ;
29 wire                wr_ram_en                ;
30 wire                  rd_ram_en              ;
31 reg                   rd_ram_en_dd           ;
32 reg [9:0] rd_ram_addr ;
33 wire [12:0]          pos_x                    ;//horizontal axis
34 wire [12:0]          pos_y                    ;//vertical coordinate
35 wire                wave_vld           ;//Waveform window
36 reg [2:0] state                        ;
37
38 reg [11:0]          cnt_data           ;
39 wire                  add_cnt_data       ;
40 wire                  end_cnt_data       ;
41 wire          [7:0] doubt                ;
42
43 reg           [31:0] timer              ;
44 reg           [23:0] hdmi_td_r          ;
45 wire          [23:0] p_data             ;
46 wire                 p_de               ;
47 wire                 p_vs               ;
48 wire                 p_hs               ;
49
50
51
```

```
52 assign wr_ram_en = state==CAPTURE;
53 assign hdmi_td = hdmi_td_r ;
54 assign hdmi_tx_hs = p_hs;
55 assign hdmi_tx_de = p_de;
56 assign hdmi_tx_vs = p_vs;
57
58 //------------------------------------------------
59 //recove port
60 //------------------------------------------------
61
62 //-------------------state machine describe--------------------
63 always @(posedge ad_clk) begin
64 if (rst==1'b1) begin
65              state <= IDLE;
66 end
67 else begin
68              case(state)
69                        IDLE:begin
70                                  state <= CAPTURE;
71                        end
72
73                        CAPTURE:begin
74                                  if(end_cnt_data)
75                                            state <= WAIT;
76                                  else
77                                            state <= CAPTURE;
78                        end
79
80                        WAIT:begin
81                                  if(timer==WAIT_MAX)
82                                            state <= CAPTURE;
83                                  else
84                                            state <= WAIT;
85                        end
86
87                        default:begin
88                                  state <= IDLE;
89                        end
90              endcase
91 end
92 end
```

```
93
94 //----------cnt_data----------
95 always @(posedge ad_clk)begin
96         if(rst==1'b1)begin
97             cnt_data <= 0;
98         end
99         else if(add_cnt_data)begin
100             if(end_cnt_data)
101                 cnt_data <= 0;
102             else
103                 cnt_data <= cnt_data + 1'b1;
104         end
105         else begin
106             cnt_data <= 'd0;
107         end
108 end
109
110 assign add_cnt_data = state==CAPTURE;
111 assign end_cnt_data = add_cnt_data && cnt_data== DATA_MAX;
112
113 //----------timer----------
114 always @(posedge ad_clk)begin
115         if(rst == 1'b1)begin
116             hours <= 1'b0;
117         end
118         else if(state==WAIT)begin
119             if(timer==WAIT_MAX)
120                 timer <= 'd0;
121             else
122                 hours <= hours + 1'b1;
123         end
124         else begin
125             timer <= 'd0;
126         end
127 end
128
129 //----------wr_ram_addr----------
130 always @(posedge ad_clk)begin
131         if(rst == 1'b1)begin
132             wr_ram_addr <= 'd0;
133         end
```

```
134         else if(wr_ram_en==1'b1)begin
135             if(wr_ram_addr==ADDR_MAX)
136                 wr_ram_addr <= 'd0;
137             else
138                 wr_ram_addr <= wr_ram_addr + 1'b1;
139         end
140         else begin
141             wr_ram_addr <= 'd0;
142         end
143 end
144
145     dpram_8x1024 inst_ram (
146             .clka(ad_clk),                  // input wire clka
147             .wea(wr_ram_en),                // input wire [0 : 0] wea
148             .addra(wr_ram_addr),// input wire [9 : 0] addra
149             .dina(ad_data),                 // input wire [7 : 0] in
150             .clkb(hdmi_tx_clk), // input wire clkb
151             .addrb(rd_ram_addr),// input wire [9 : 0] addrb
152             .dead(dead)                     // output wire [7 : 0] doutb
153         );
154
155     hdmi_background inst_hdmi_background (
156             .rst                (rst),
157             .hdmi_tx_clk (hdmi_tx_clk),
158             .hdmi_tx_de (p_de),
159             .hdmi_tx_hs (p_hs),
160             .hdmi_tx_vs (p_vs),
161             .hdmi_td            (p_data),
162             .pos_x              (pos_x),
163             .pos_y              (pos_y),
164             .rd_ram_en (rd_ram_en),
165             .wave_vld           (wave_vld)
166         );
167
168 //----------------------------------------------------
169 //transfer port
170 //----------------------------------------------------
171 //------------------rd_ram_addr--------------------
172 always @(posedge hdmi_tx_clk)begin
173         if(rst == 1'b1)begin
174             rd_ram_addr <= 'd0;
```

```
175             end
176             else if(rd_ram_en==1'b1)begin
177                     if(rd_ram_addr==ADDR_MAX)
178                             rd_ram_addr <= 'd0;
179                     else
180                             rd_ram_addr <= rd_ram_addr + 1'b1;
181             end
182             else begin
183                     rd_ram_addr <= 'd0;
184             end
185 end
186
187 always @(posedge hdmi_tx_clk)begin
188             rd_ram_en_dd <= rd_ram_en;
189 end
190
191 always @(posedge hdmi_tx_clk)begin
192             if(rst == 1'b1)begin
193                     hdmi_td_r <= 'd0;
194             end
195             else if(rd_ram_en_dd==1'b1)begin
196                     if(AMP_DATA-pos_y == {4'd0,dout}) // Is the current position on the waveform?
197                             hdmi_td_r <= 24'hFFFF00;
198                     else
199                             hdmi_td_r <= p_data;
200             end
201             else begin
202                    hdmi_td_r <=p_data;
203             end
204 end
205
206 endmodule
```

The code snippet worth noting is from lines 191 to 203, which completes the process of adding the background image to the image.

The function of superimposing and displaying the module data and AD waveform.

For other modules in this project, please check the other source codes of this project.

This has been introduced in the chapter and will not be repeated in this chapter.

15.4 Adding Constraint Files

The constraint file of this experiment is based on the ADDA conversion experiment chapter, adding HDMI

Just display the constraints of the module. The constraint file is as follows:

```
############### clock define##################
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

############### key define##################
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]



set_property PACKAGE_PIN U18 [get_ports hdmi_tx_clk_p]
set_property PACKAGE_PIN N20 [get_ports {hdmi_tx_chn_b_p}]
set_property PACKAGE_PIN T20 [get_ports {hdmi_tx_chn_g_p}]
set_property PACKAGE_PIN V20 [get_ports {hdmi_tx_chn_r_p}]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_r_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_g_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_chn_b_p]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_clk_p]

set_property PACKAGE_PIN W19 [get_ports {da_data[0]}]
set_property PACKAGE_PIN W18 [get_ports {da_data[1]}]
set_property PACKAGE_PIN U17 [get_ports {da_data[2]}]
set_property PACKAGE_PIN T16 [get_ports {da_data[3]}]
set_property PACKAGE_PIN R17 [get_ports {da_data[4]}]
set_property PACKAGE_PIN R16 [get_ports {da_data[5]}]
set_property PACKAGE_PIN P18 [get_ports {da_data[6]}]
set_property PACKAGE_PIN N17 [get_ports {da_data[7]}]
set_property PACKAGE_PIN Y18 [get_ports da_clk]

set_property SLEW FAST [get_ports {da_data[*]}]
set_property SLEW FAST [get_ports da_clk]

set_property IOSTANDARD LVCMOS33 [get_ports {da_data[*]}]
set_property IOSTANDARD LVCMOS33 [get_ports da_clk]
```

```
set_property PACKAGE_PIN U12 [get_ports ad_clk]
set_property PACKAGE_PIN V13 [get_ports {ad_data[0]}]
set_property PACKAGE_PIN U13 [get_ports {ad_data[1]}]
set_property PACKAGE_PIN U15 [get_ports {ad_data[2]}]
set_property PACKAGE_PIN U14 [get_ports {ad_data[3]}]
set_property PACKAGE_PIN R14 [get_ports {ad_data[4]}]
set_property PACKAGE_PIN P14 [get_ports {ad_data[5]}]
set_property PACKAGE_PIN W15 [get_ports {ad_data[6]}]
set_property PACKAGE_PIN V15 [get_ports {ad_data[7]}]

set_property SLEW FAST [get_ports ad_clk]

set_property IOSTANDARD LVCMOS33 [get_ports ad_clk]
set_property IOSTANDARD LVCMOS33 [get_ports {ad_data[*]}]
```

15.5 Download Verification

Next, you can generate a bit file for download verification. After generating the bit file, follow the steps below:

Connect the development board.

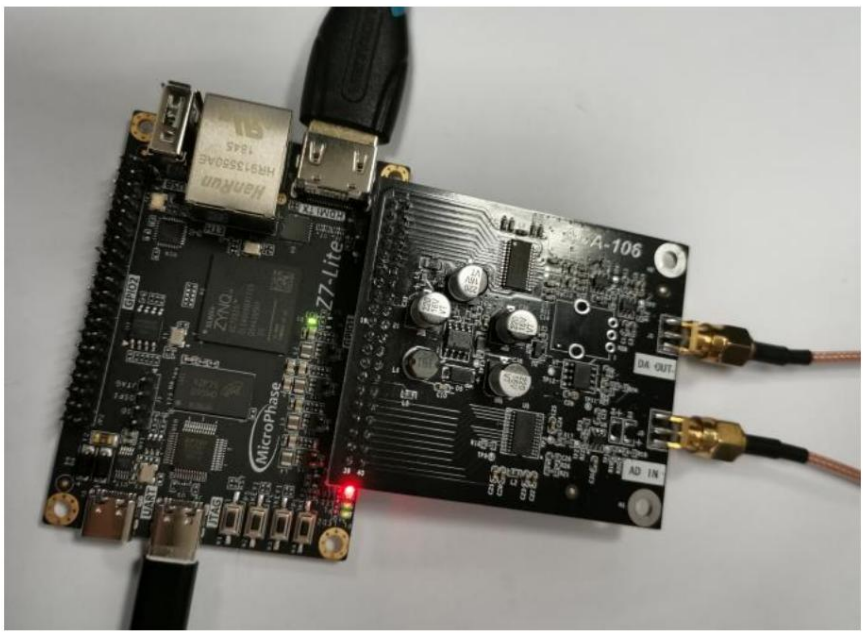


Figure 17- 10

Next, download the bit file to the FPGA. After a while, you can see the acquisition data appear on the display.

to the waveform.

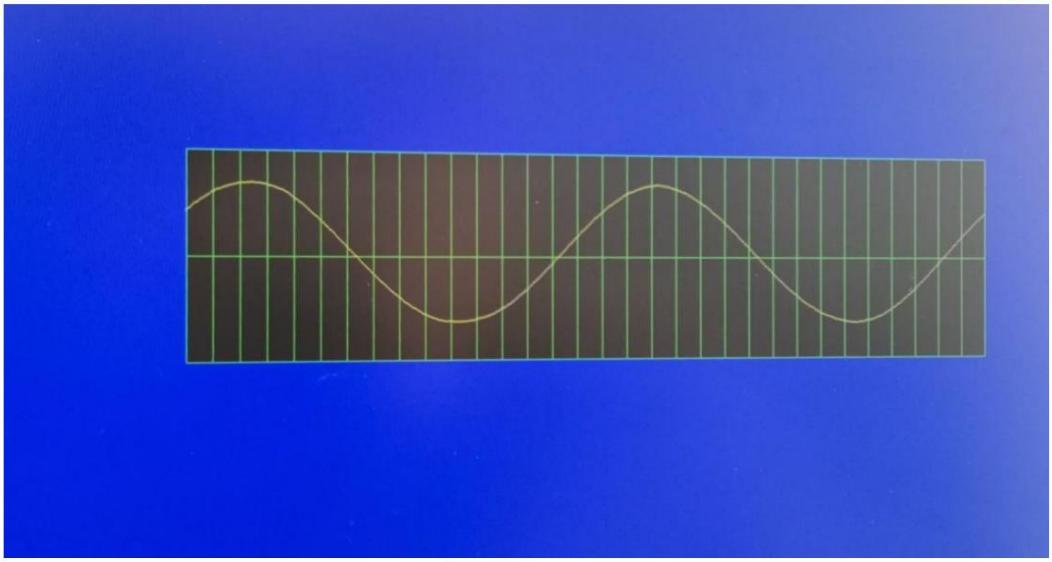


Figure 17- 11

So far, we have completed the experiment of displaying a sine wave on the screen using the ADDA daughter card.

In the same way, we only need to change the initialization file coe file to change the sampled waveform.

The same frequency can also produce different waveforms. We will not introduce it in this chapter. Those who are interested can set it up by themselves.

Design a few simple waveform displays.

15.6 Experimental Summary

In this experiment, the ADDA daughter card was used to display the collected waveform on the monitor.

The main purpose of the experiment is to complete the superposition of images. Simply put, the superposition of images is to change the image you want to display.

Priority, superimpose the image onto the background based on the priority.

Chapter 16 Using Third-Party **IP** to Complete **HDMI** Output

16.1 Experimental Background

There is one HDMI port on the Z7-Lite development board, which can be used as HDMI input or output.

This experiment will use a third-party IP to complete the HDMI output. This experiment only needs to use open source IP.

Can.

16.2 Download third-party **IP**

This experiment requires the use of a third-party IP, which can be found on github. The following is the github link

Go to: https://github.com/Digilent/vivado-library

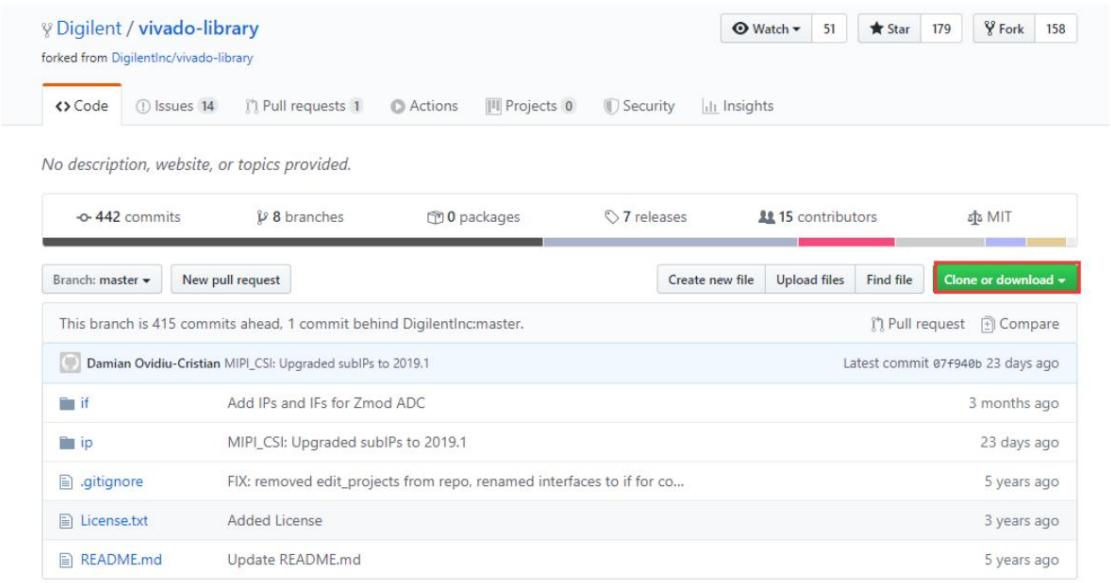


Figure 18- 1

Click clone to download files from other users' repositories to your local computer.

Add the third-party IP to the Vivado project, and then users can use the third-party IP.

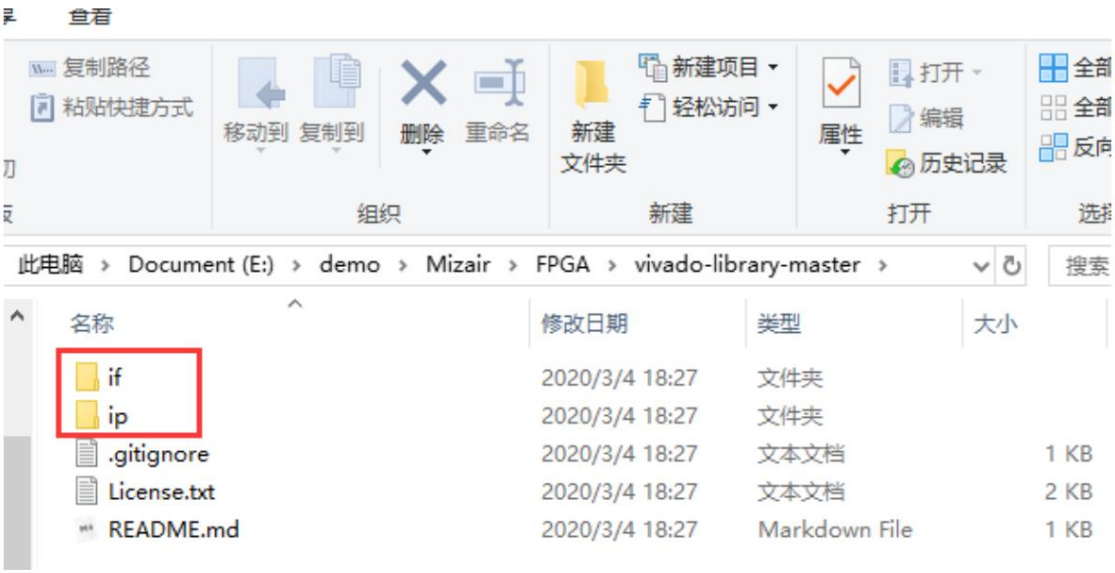


Figure 18- 2

After the download is complete, you can see that there are two folders, the ip folder is used to store IP, if the file The folder is used to store interfaces. This experiment requires the use of dvi2rgb and rgb2dvi in the ip folder. These two IPs also need to use the tmds interface definition in the if folder.

Create a new ip_repo folder in the project directory to store third-party IPs and Assign the required IP to this folder.

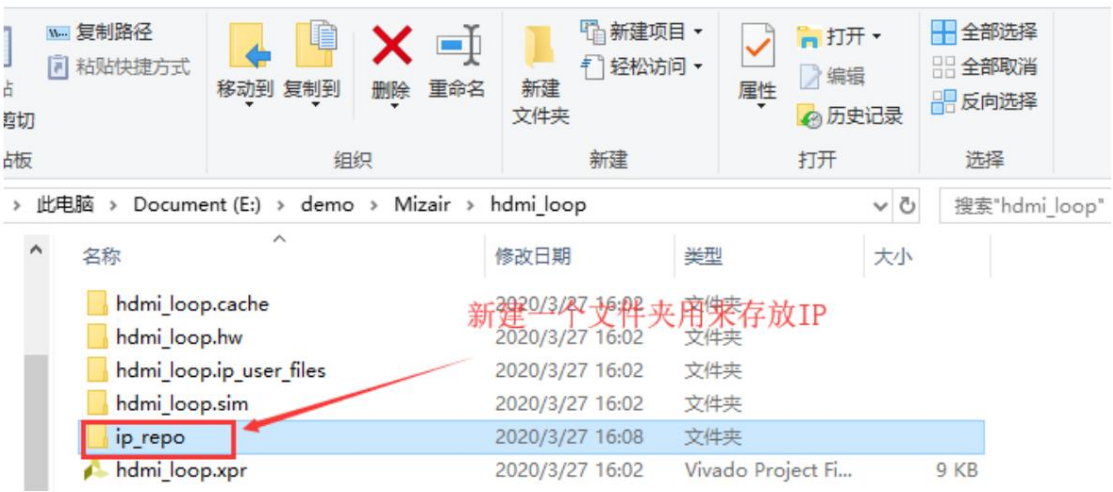


Figure 18- 3

Copy the required IP folder to ip_repo

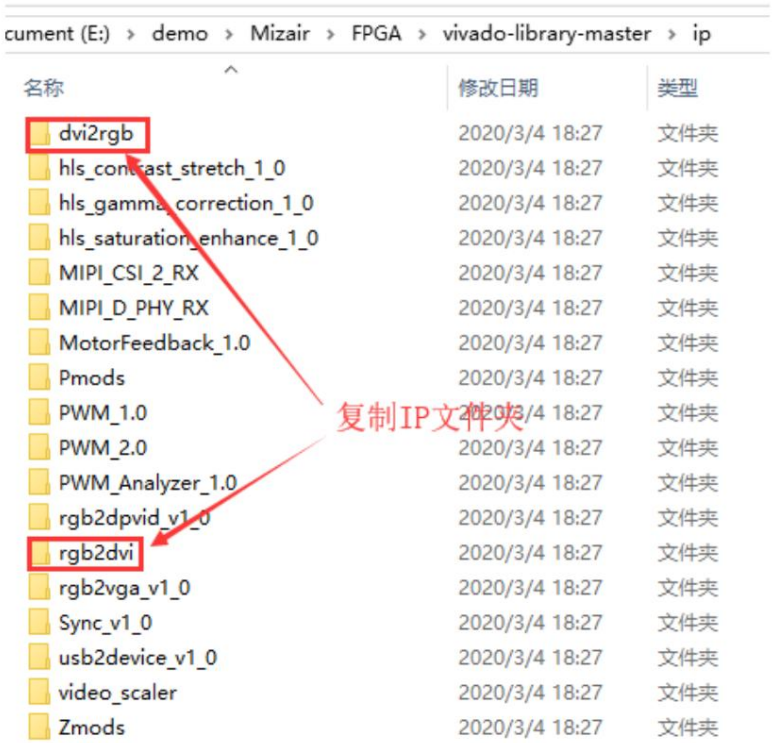


Figure 18- 4

Copy the required if folder to ip_repo

16.3 Create a Project

Start Page	Create New Project
Wizard Start Page	Next
Project Name	The project name is pcie_test, the project path is E:\demo\hdmi_loop, check Createproject subdirectory, Next
Project Type	RTL ProjectÿNext
Add Source	Next
Add Existing IP	Next
Add Constraints	Next
Default part	Select device, purchase Z7-Lite7020 User selected xc7z020clg400-2, purchase Z7-Lite7010 user selected xc7z010clg400-1
After confirming that the New Project summary is correct, click Confirm.	

16.4 Add third-party **IP** to the **IP** repository

After creating the project, you first need to add the third-party IP to the IP repository. Click settings ÿ

Select IP repository ÿ click the "+" sign to select the path of ip_repo.

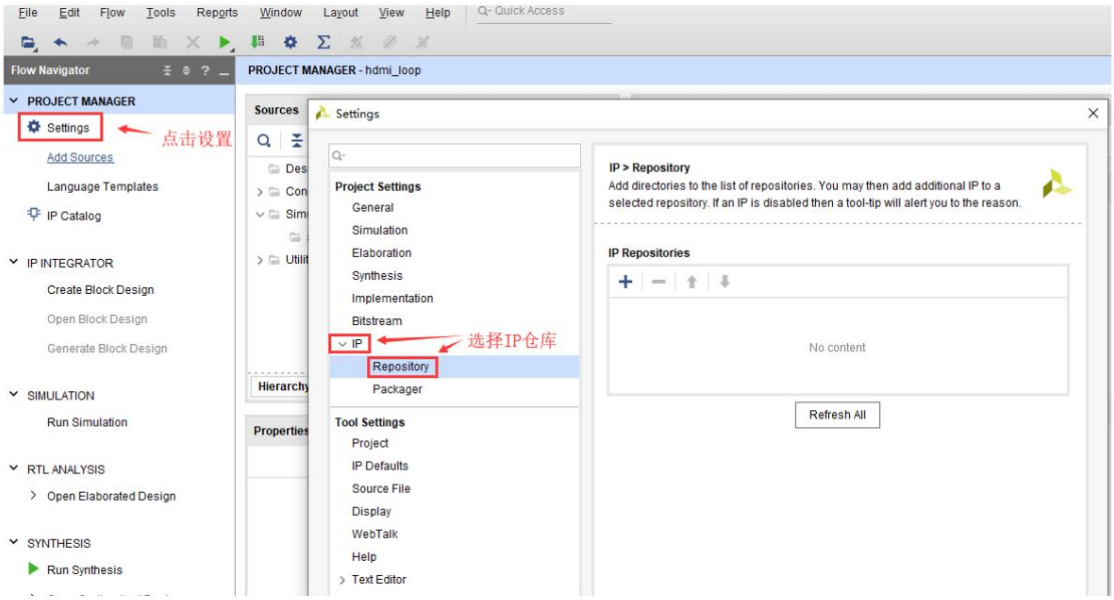


Figure 18- 5

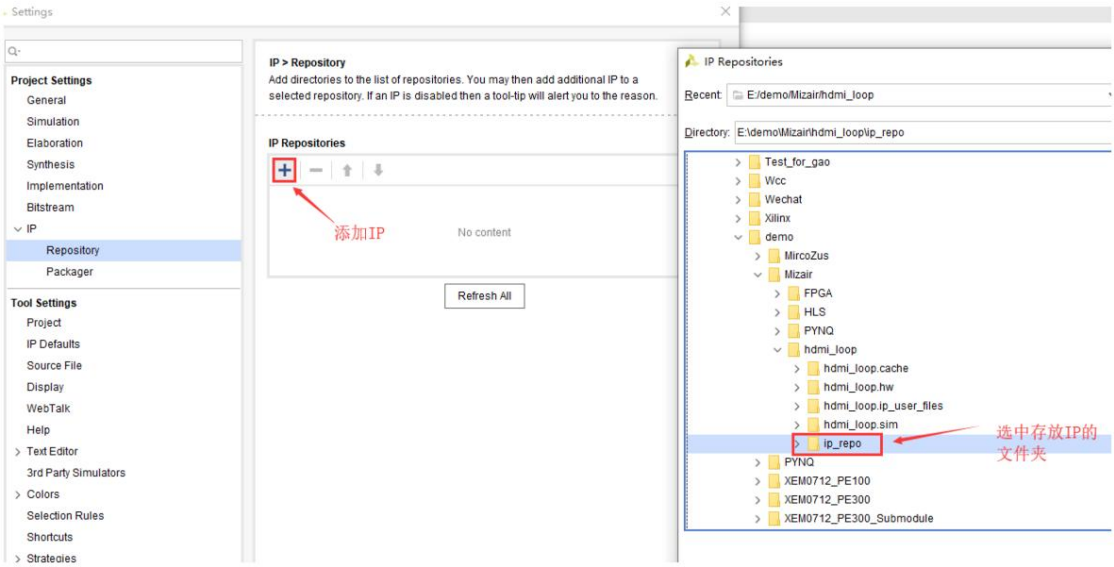


Figure 18- 6

After selecting the path where the IP is stored, the Vivado tool will identify the custom IP and custom interface in it.

And add it to the IP warehouse. Finally, click Apply ÿ OK to complete adding the IP to the IP warehouse.

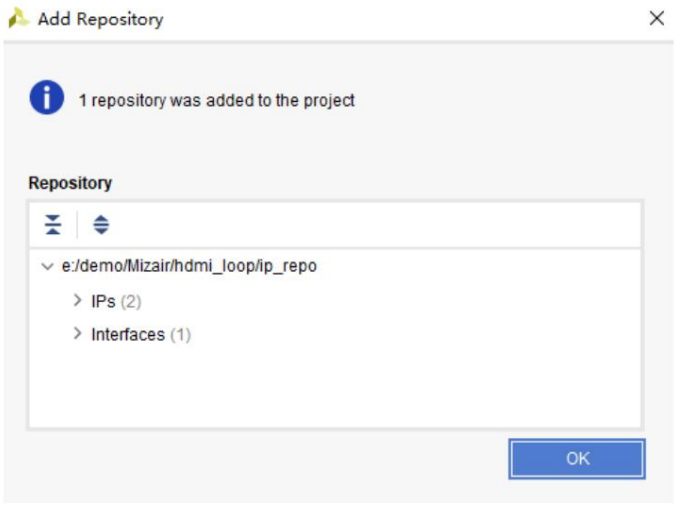


Figure 18- 7

16.5 Generating Third-Party **IP**

Click IP Catalog ÿ Select User Repository ÿ RGB to DVI Encoder ÿ Double-click to configure Set.

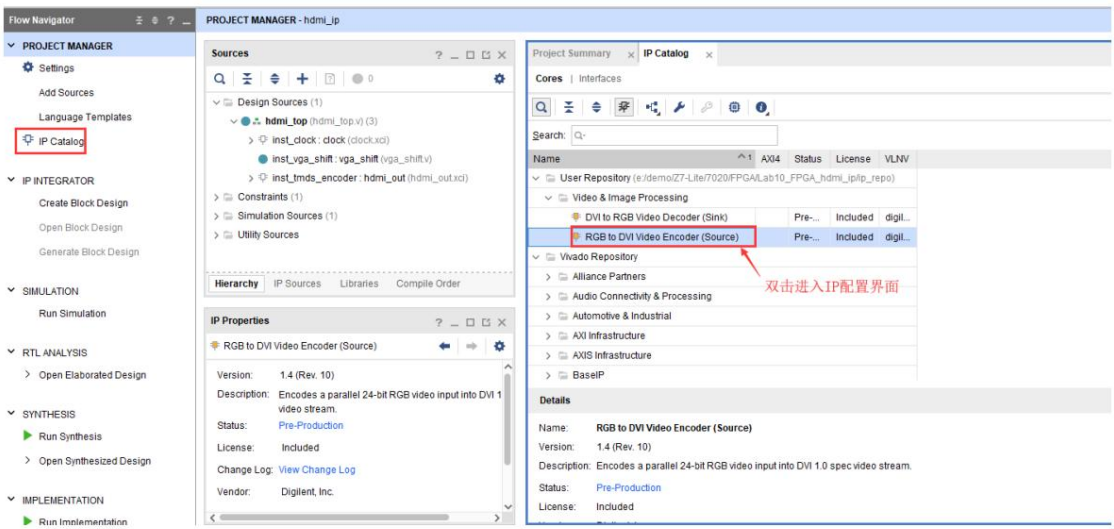


Figure 18- 8

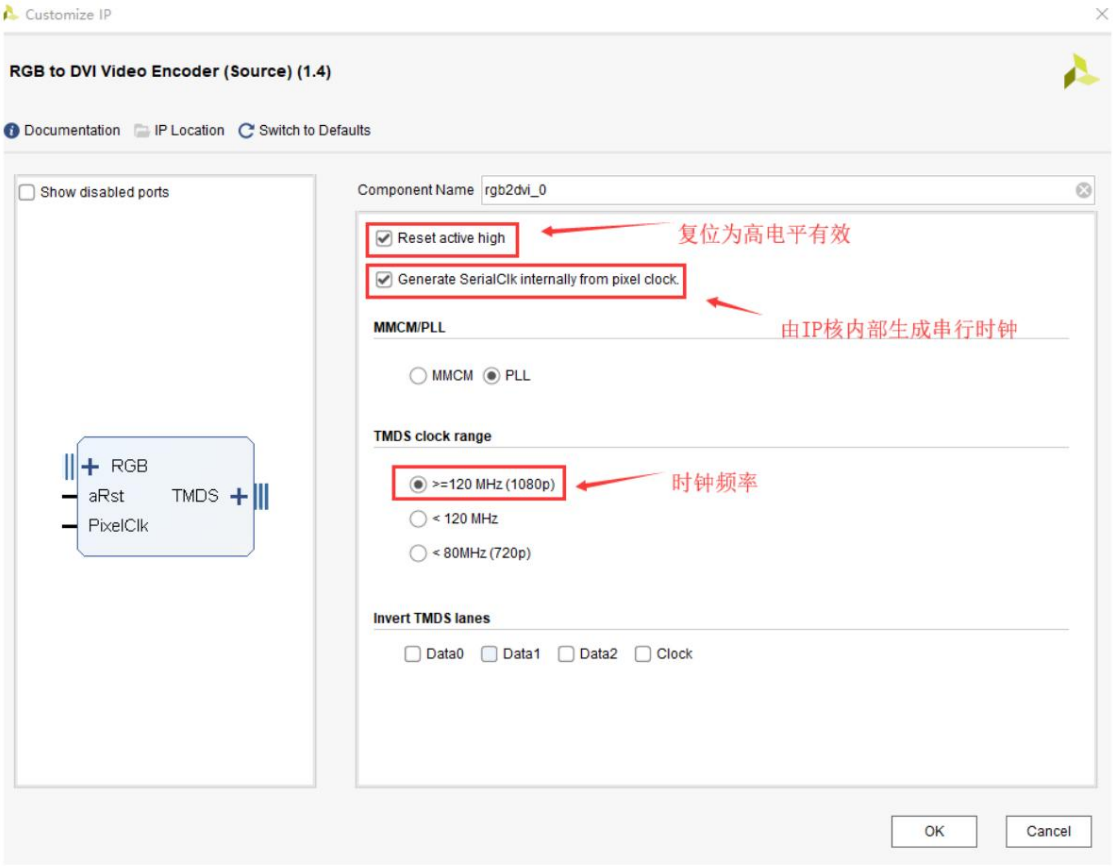


Figure 18-9 IP core configuration

```
// The following must be inserted into your Verilog file for this
// core to be instantiated. Change the instance name and port connections
// (in parentheses) to your own signal names.

//----------- Begin Cut here for INSTANTIATION Template ---// INST_TAG
hdmi_out your_instance_name (
  .TMDS_Clk_p(TMDS_Clk_p),    // output wire TMDS_Clk_p
  .TMDS_Clk_n(TMDS_Clk_n),    // output wire TMDS_Clk_n
  .TMDS_Data_p(TMDS_Data_p),  // output wire [2 : 0] TMDS_Data_p
  .TMDS_Data_n(TMDS_Data_n),  // output wire [2 : 0] TMDS_Data_n
  .aRst(aRst),                // input wire aRst
  .vid_pData(vid_pData),      // input wire [23 : 0] vid_pData
  .vid_pVDE(vid_pVDE),        // input wire vid_pVDE
  .vid_pHSync(vid_pHSync),    // input wire vid_pHSync
  .vid_pVSync(vid_pVSync),    // input wire vid_pVSync
  .PixelClk(PixelClk)        // input wire PixelClk
);
// INST_TAG_END ------ End INSTANTIATION Template ---------
```

Figure 18- 10 IP instantiation template

16.6 Programming

The program design of this experiment has been introduced in Chapters 14 and 15.

The TMDS encoding module designed by the user above needs to be replaced with the third-party IP in this experiment.

Can.

The underlying modules are as follows:

You only need to instantiate the IP into the project, replace the previous encoding implemented by the user, and serially convert

module.

```
1 `timescale 1ns / 1ps
2
3
4 module hdmi_top(
5 input 6          wire              clk                 ,
input              wire                    rst_n               ,
7 output wire                              hdmi_tx_clk_p       ,
8 output wire 9 output                     hdmi_tx_clk_n       ,
wire                          [2:0]hdmi_tx_data_p ,
10 output wire                [2:0]hdmi_tx_data_n
11          );
12
13 wire                  locked          ;
14 wire                  pixel_clk       ;
15 wire                  rst         ;
16
17 wire                  vpg_hs          ;
18 wire                  vpg_vs          ;
19 wire                  vpg_de          ;
20 wire [23:0]           rgb         ;
21
22 assign rst = ~locked;
23 clock inst_clock(
24
25              .clk_out1(pixel_clk), .reset(~rst_n),
26
27              .locked(locked), .clk_in1(clk)
28          );
```

```
29
30 vga_shift inst_vga_shift (
31             .rst                (rst),
32             .vpg_pclk (pixel_clk),
33             .vpg_de (vpg_de),
34             .vpg_hs (vpg_hs),
35             .vpg_vs (vpg_vs),
36             .rgb                (rgb)
37 );
38
39 hdmi_out inst_tmds_encoder (
40                                                         // output wire TMDS_Clk_p
41                                                         // output wire TMDS_Clk_n
42             .TMDS_Clk_p(hdmi_tx_clk_p), .TMDS_Clk_n(hdmi_tx_clk_n), .TMDS_Data_p(hdmi_tx_data_p), // output wire [2 : 0] TMDS_Data_p
43             .TMDS_Data_n(hdmi_tx_data_n), // output wire [2 : 0] TMDS_Data_n
44                                                     // input wire aRst
45                                                 // input wire [23 : 0] vid_pData
46                                                     // input wire vid_pVDE
47                                                 // input wire vid_pHSync
48                                                 // input wire vid_pVSync
49             .aRst(rst), .vid_pData(rgb), .vid_pVDE(vpg_de), .vid_pHSync(vpg_hs), .vid_pVSync(vpg_vs), .PixelClk(pixel_clk)
50 );
51
52
53 endmodule
```

16.7 Adding Constraint Files

```
############### clock define###################
create_clock -period 20.000 [get_ports clk]
set_property PACKAGE_PIN N18 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]


############### key define###################
set_property PACKAGE_PIN P16 [get_ports rst_n]
set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
set_property PACKAGE_PIN U18 [get_ports hdmi_tx_clk_p]
set_property PACKAGE_PIN N20 [get_ports {hdmi_tx_data_p[0]}]
set_property PACKAGE_PIN T20 [get_ports {hdmi_tx_data_p[1]}]
```

```
set_property PACKAGE_PIN V20 [get_ports {hdmi_tx_data_p[2]}]
set_property IOSTANDARD TMDS_33 [get_ports {hdmi_tx_data_p[*]}]
set_property IOSTANDARD TMDS_33 [get_ports hdmi_tx_clk_p]
set_property CLOCK_DEDICATED_ROUTE BACKBONE [get_nets inst_clock/inst/clk_in1_clock]
```

16.8 Experimental Phenomena

The experimental results of this experiment are consistent with the experimental results of Chapter 14, and will be displayed on the monitor.

Shows a moving block.

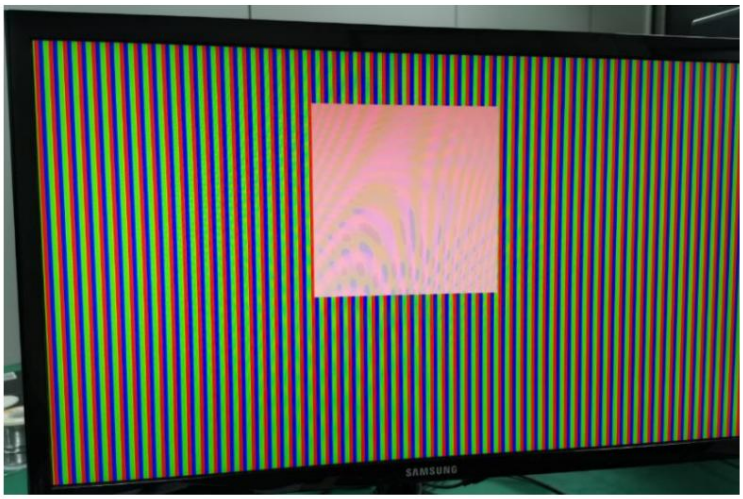


Figure 18- 11

16.9 Experimental Summary

In this experiment, the HDMI display is completed by using open source IP.

This chapter introduces how to add third-party IP in Vivado and instantiate it in the project.

Adding custom IP or third-party IP can speed up development.

Summary of the first version

Thank you for choosing Z7-Lite ZYNQ7000 series products from Microphase Technology (Shanghai) Co., Ltd.

Based on the above content, I believe you are already familiar with our development board and can skillfully use the PL

The resources of the (programmable logic) part, the development of the PL side is consistent with the traditional FPGA development process, and then

From the content, you will learn about the development of the PS side, and then you can truly reflect the advantages of the ZYNQ series.

Due to my limited level, I cannot guarantee that all the content in this article is correct. If you find any errors,

Please contact us and we will change it as soon as possible. The routines we can provide are very limited at the moment, but please give us your feedback.

If we give it more time, we will work hard to enrich and improve the relevant resources of this development board.

This document is my first written content, please forgive me for any errors that may occur.

We will revise it as soon as possible, and this document will be gradually enriched over time, rather than being static.

Please continue to pay attention.

MircoPhase