PRA040 USER EXPERIMENTAL MANUAL

PRA040 EXPERIMENTAL INSTRUCTIONS

FRASER INNOVATION INC
JULY 31, 2019

Version Control

version	Date	Description
1.0	07/20/2019	Initial Release
1.1	07/29/2019	Add Experiment 15
1.2	07/31/2019	Revised some description about HDMI

Contents

Project Files Appendix

Experiment 1: LED_shifting
Experiment 2: SW_LED
Experiment 3: BCD_counter
Experiment 4: block_counter
Experiment 5: block_debouncing
Experiment 6: mult_sim
Experiment 7: HEX_BCD，HEX_BCD_mult
Experiment 8: memory_rom
Experiment 9: dual_port_ram
Experiment 10: UART_FRAME
Experiment 11: eeprom_test
Experiment 12: adda_test
Experiment 13: hdmi
Experiment 14: Ethernet
Experiment 15: SRAM

Part One: Introduction of FII-PRA040 Development System

1、Design Objective of the System

The main purpose of this system design is to complete FPGA learning, development and experiment with Intel Quartus. The main device uses the Inte Cyclone10 10CL040YF484C8G and is currently the latest generation of FPGA devices from Intel. The major learning and development projects can be completed as follows:

(1) Basic FPGA design training
(2) Construction and training of the SOPC (NiosII) system
(3) IC design and verification, the system provides hardware design, simulation and verification of RISC-V CPU
(4) Development and application based on RISC-V
(5) The system is specifically optimized for hardware design for RISC-V system applications

2、System Resource

（1） Extended memory: Two Super Sram (IS61WV51216, 512K x 32bit) are connected in parallel to form a 32-bit data interface, and the maximum access space is up to 2M bytes.
（2） Serial flash: Spi interface serial flash (16M bytes)
（3） Serial EEPROM
（4） Gigabit Ethernet: 100/1000 Mbps
（5） USB to serial interface: USB-UART bridge

3、Human-computer Interaction Interface

（1） 8 DIP switches
（2） 8 push buttons, definition of 7 push buttons: MENU, UP, RETUN, LEFT, OK, RIGHT, DOWN, 1 for reset: RESET
（3） 8 LEDs
（4） 6 7-segment LED display
（5） I2C bus interface
（6） UART external interface
（7） Two JTAG programming interfaces: One is for downloading the FPGA debug interface, and the other is the JTAG debug interface for RISC-V CPU
（8） Built-in RISC-V CPU software debugger, no external RISC-V JTAG emulator required
（9） 4 12-pin GPIO connectors, in line with PMOD interface standards

4、Software Development System

（1） Quartus 18.0 and later version for FPGA development, Nios-II SOPC

（2） Freedom Studio-Win_x86_64 software development for RISC-V CPU

5、Supporting Resources

RISC-V	JTAG Debugger
Intel Altera	JTAG Download Debugger
FII-PRA040	User Experimental Manual
FII-PRA040	Hardware Reference Guide

6、Physical Picture

（1） FII-PRA040 system block diagram

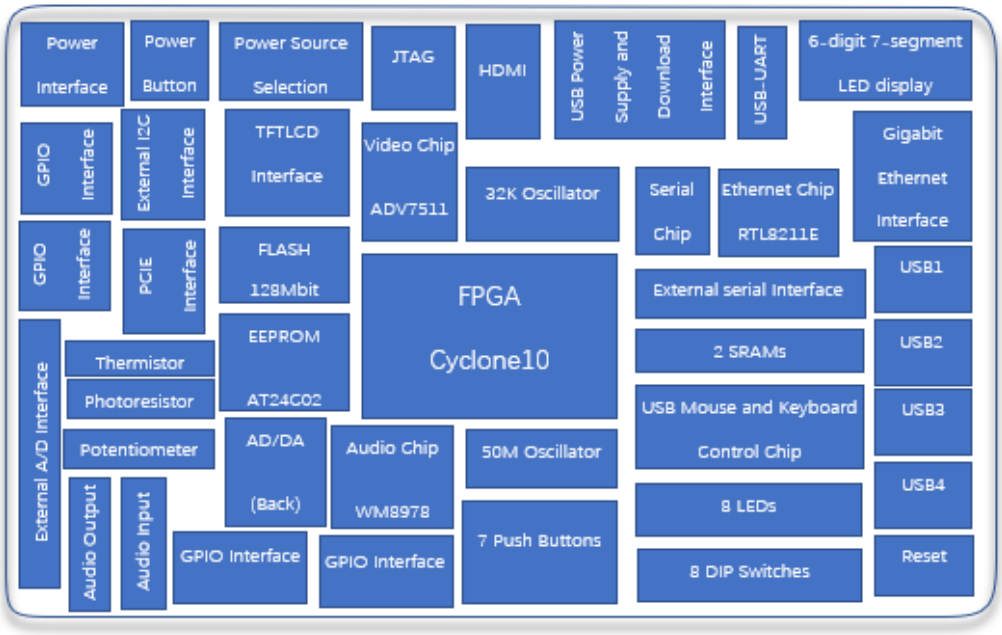


Figure 1 PRA040 system block diagram

（2） FII-PRA040 physical picture

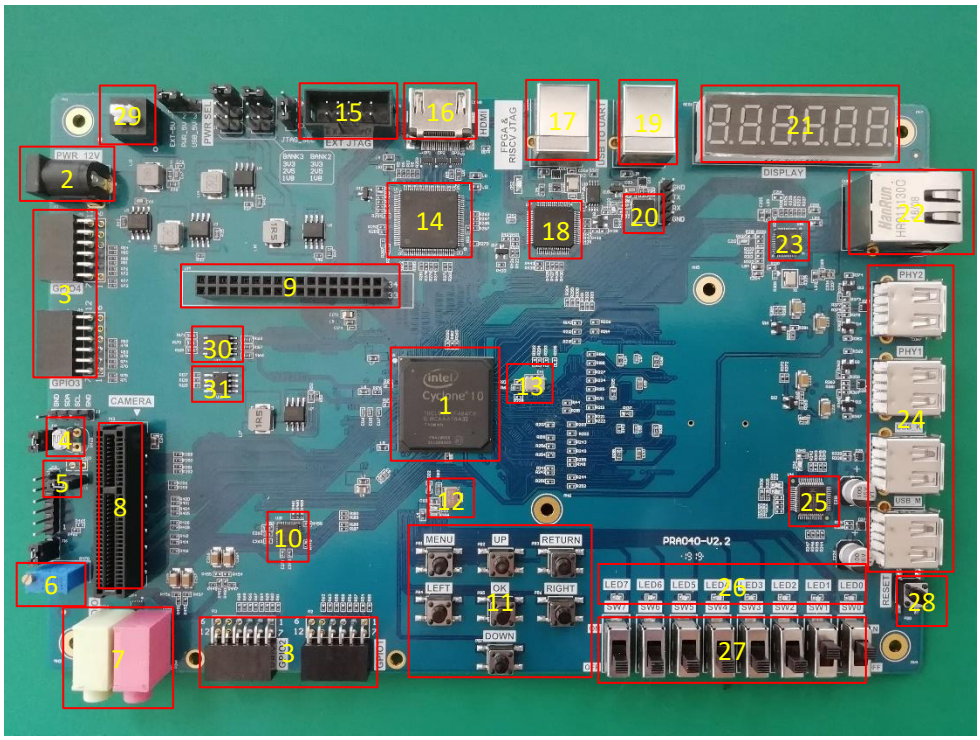


Figure 2 PRA040 physical front view

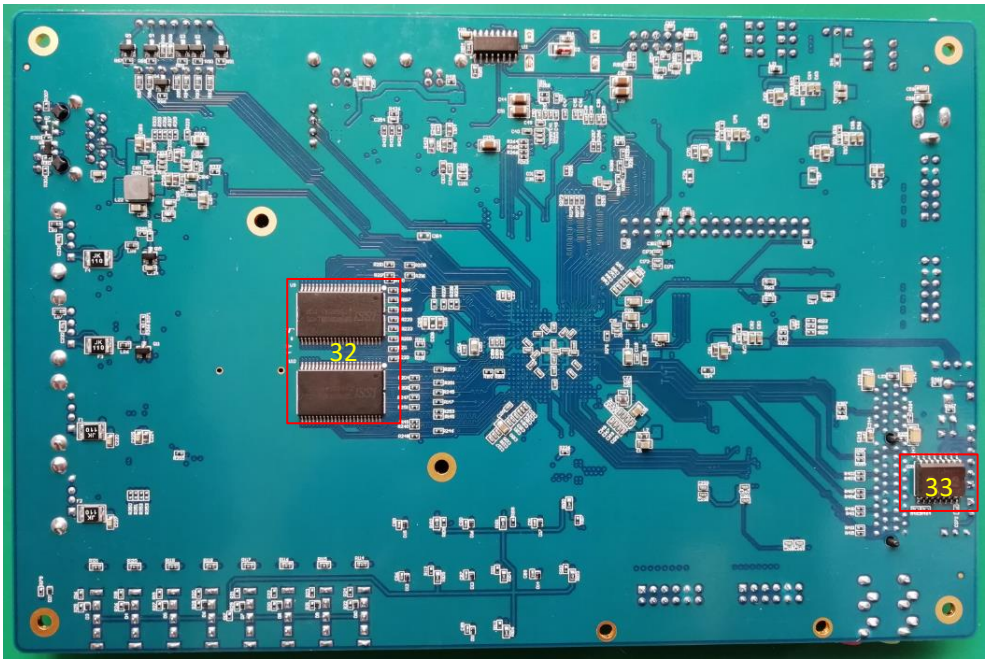


Figure 3 PRA040 physical back view

（3） Corresponding to the physical picture, the main devices on board are as follows:

1、10CL040YE484C8G chip

2、External 12V power interface

3、GPIO interface
4、Thermistor (NTC-MF52)
5、Photoresistor
6、Potentiometer
7、Audio output (green), audio input (red)
8、PCIE interface
9、TFTCLD interface
10、Audio chip (WM8978)
11、7 push buttons
12、50M system clock
14、Video chip(ADV7511）
15、External JTAG download interface
16、HDMI interface
17、USB power supply and download interface
18、FPGA and RISC_V JTAG download chips (FT2232)
19、USB_UART interface
20、Serial chip (CP2102)
21、6 7-segment LED display
22、Ethernet interface
23、Ethernet PHY chip (RTL8211E-VB)
24、4 USB interfaces
25、USB mouse and keyboard control chip
26、8 LEDs
27、8-bit DIP switch
28、Reset button
29、Power button
30、Flash (N25Q128A，128M bit/16M bytes)
31、EEPROM (AT24C02N)
32、Two SRAMs
33、AD/DA conversion chip (PCF8591)

Part Two: FII-PRA040 Main Hardware Resources Usage and FPGA Development Experiment

This part mainly guides the user to learn the development of FPGA program and the use of onboard hardware through the development example of FPGA. At the same time, the application system software Quartus is introduced from the elementary to the profound. The development exercises covered in this section are as follows:

Experiment 1: LED shifting design
Experiment 2: SignalTap experiment
Experiment 3: Segment display experiment
Experiment 4: Block/SCH experiment
Experiment 5: button debouncing experiment
Experiment 6: use of multiplier and ModelSim simulation
Experiment 7: hex to BCD conversion and application
Experiment 8: usage of ROM
Experiment 9: use dual-ROM to read and write frame data
Experiment 10: asynchrounous serial port design and experiment
Experiment 11: IIC transmission experiment
Experiment 12: AD/DA experiment
Experiment 13: HDMI experiment
Experiment 14: Ethernet experiment
Experiment 15: SRAM read and write

Learning exercises in the order of the experimental design, and successfully completing these basic experiments, we will be able to achieve the level and capabilities of the primary FPGA engineers.

Experiment 1　LED shifting

1.1　Experiment Objective

（1）　Practice to use Quartus II to create new projects and use system resources IP Core;
（2）　Proficiency in the writing of Verilog HDL programs to develop a good code writing style;
（3）　Master the design of the frequency divider to implemnet the runing LED;
（4）　Combine hardware resources to perform FPGA pin assignment and implement actual program downloading;
（5）　Observe the experiment result and summarize it.

1.2　Experiment Implement

（1）　Use all LEDS, all light up during reset;
（2）　End reset, LED lights from low to high (from right to left) in turn;
（3）　Each LED is lit for one second;
（4）　After the last (highest position) LED is lit, the next time it returns to the first (lowest position) LED, the loop is achieved;

1.3　Experiment

1.3.1　LED Introduction

LED (Light-Emitting Diode), is characterized by low operating current, high reliability and long life. Up to now, there are many types of LED lights, as shown in Figure 1.1. The FII-PRA040 uses the LED lights in the red circle.

Figure 1.1 Different kinds of LEDs

1.3.2　Hardware Design

The physical picture of the onboard 8-bit LED is shown in Figure 1.2. The schematics of LED is shown in Figure 1.3. The LED module of this experiment board adopts 8 common anode LEDs,

which are connected with Vcc 3.3V through 180 R resistors, and the cathodes are directly connected and controlled by the FPGA. When the FPGA outputs a low level of 0, a current flows through the LED, and it is turned on.

Figure 1.2 8-bit LED physical picture

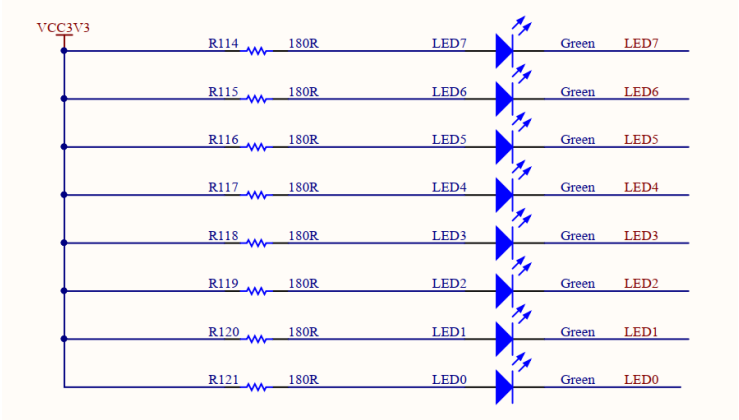


Figure 1.3 Schematics of LED

1.3.3 Program Design

1.3.3.1 Start Program

Before writing a program, let's briefly introduce the development environment we use and how to create a project. Take Quartus II 18.1 as an example. The specific project establishment steps are shown in Figure 1.4 to 1.9.

（1） As shown in Figure 1.4, after opening Quartus, you can directly click **New Project Wizard** in the middle of the screen to create a new project. You can also click **File** to create a new project in the toolbar, or press **Ctrl+N** to create a new project.

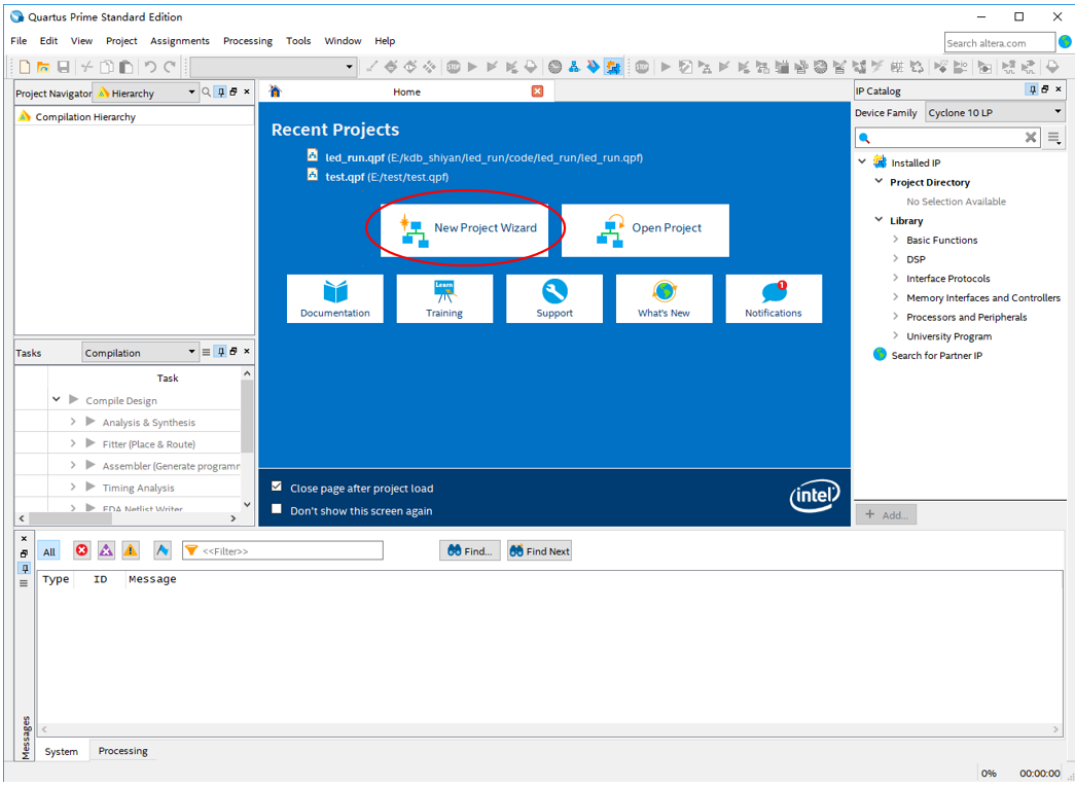


Figure 1.4 The main Quartus II interface

（2） As shown in Figure 1.5, select the correct project path. The project is named *LED_shifting*. It is recommended that the path is easy to find and convenient for later viewing and calling.

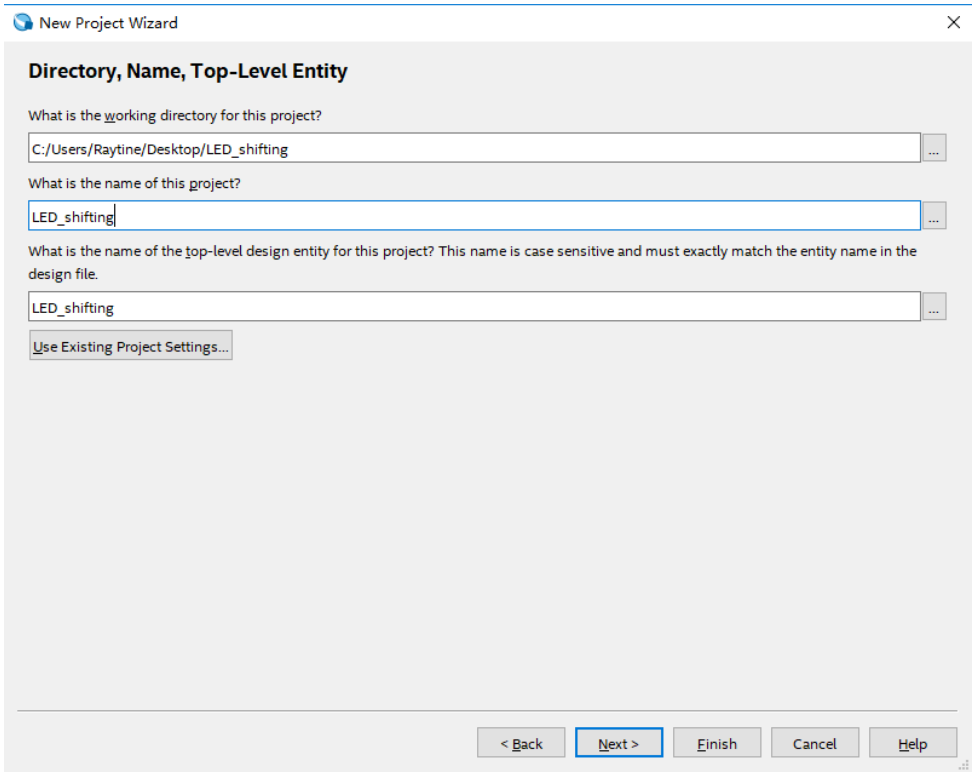


Figure 1.5 Name and define the path of the project file

（3） As shown in Figure 1.6, you can directly add some files written in advance. Since it is a new project, click **Next** to perform the next step.

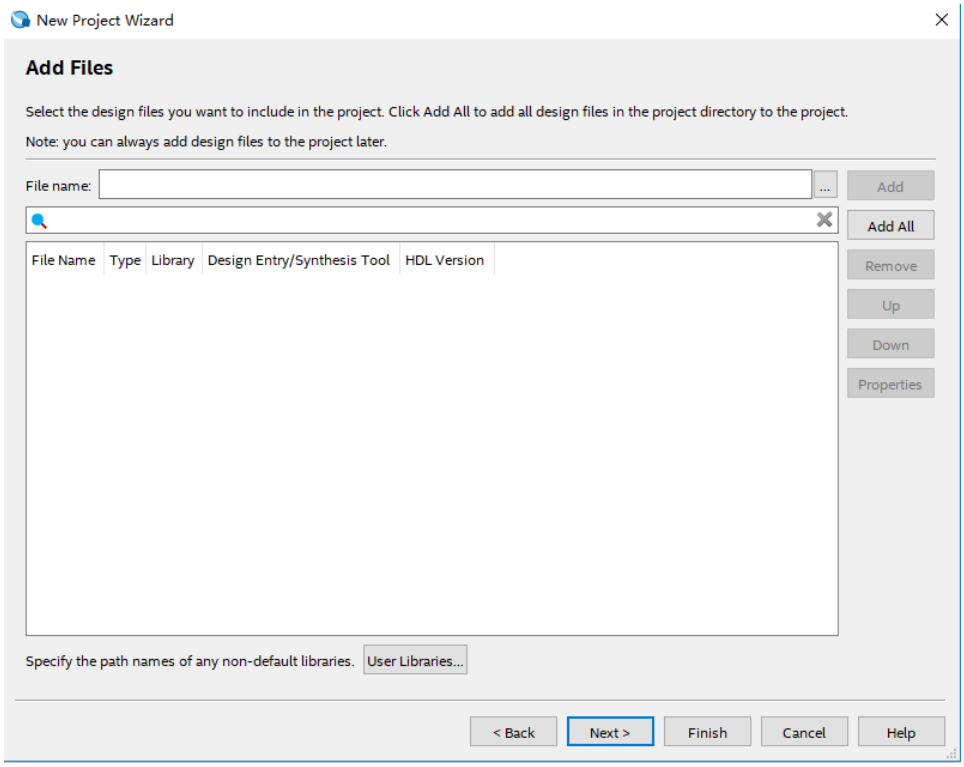


Figure 1.6 Add files

（4） As shown in Figure 1.7, select the correct FPGA chip model, the onboard chip model is 10CL040YF484C8G. Selecting **Cyclone 10 LP** in the Family, **FBGA** in the package, **484** in the Pin count, and **8** in the Core speed grage helps narrow down the selection and quickly find the target model.

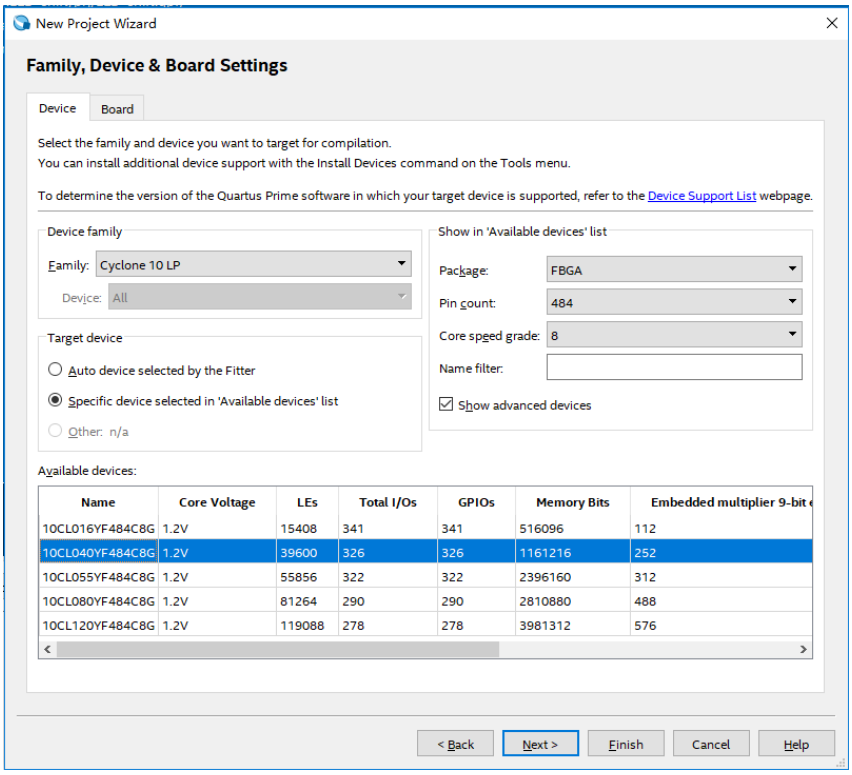


Figure 1.7 Device selection

（5） As shown in Figure 1.8, select the EDA tool. Here use the EDA tool that comes with Quartus.

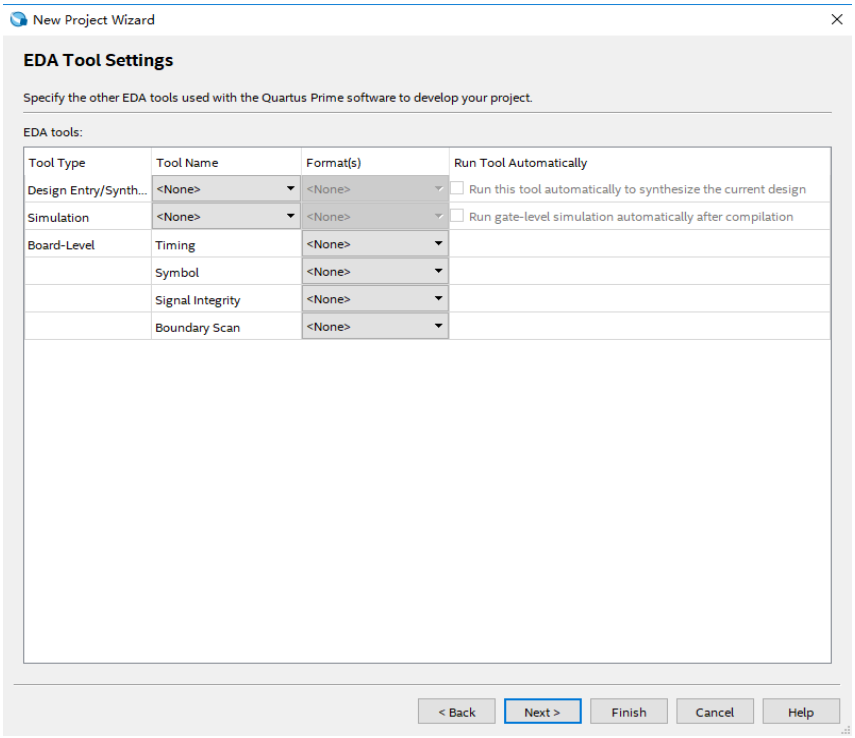


Figure 1.8 Selection of EDA tool

（6） Click **Next** to go to the next interface and select **Finish** to complete the project project.

（7） Click **File > New** or use the shortcut key **Ctrl+N** to pop up the dialog box shown in Figure 1.9, create a program file (Verilog HDL File) to write code. Pay attention to the consistency of the program name and project name, and save it in the correct path (folder).

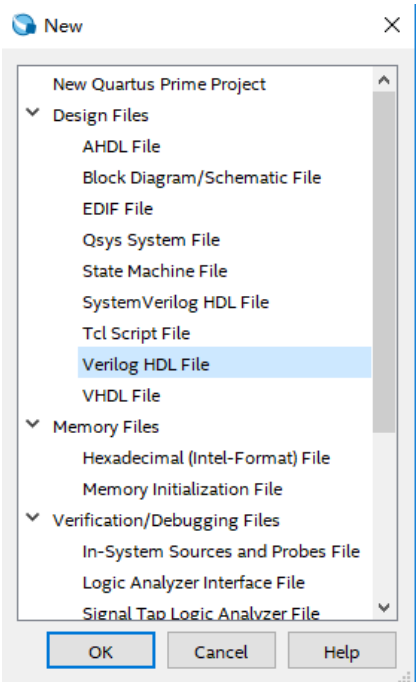


Figure 1.9 Create a new project file (LED_shifting.v)

Once the preparation is ready, start writing the program.

1.3.3.2 Program Introduction

The first step: the establishment of the main program framework (interface design)

```
module Led_shifting(
    input                   clk,
    input                   rst_n,
    output   reg  [7:0]     led
);
endmodule
```

The input signal of this experiment has 50 MHz system clock *clk* and reset signal *rst_n*. Output signal is *led*; 8 leds are defined by the multi-bit width form of *led [7:0]*.

The second step: the call of IP Core, the establishment and use of PLL module

（1） As shown in Figure 1.10, find the **ALTPLL** in the **IP catalog** option bar on the right side of the main interface.

（2） As shown in Figure 1.11, double-click **ALTPLL** and enter the name of the PLL module in the pop-up dialog box. The name given here is *PLL1*. Note that the selection type is **Verilog** language type.

（3） As shown in Figure 1.12, after completing the previous step, enter the detailed setting interface. *Inclk0* is the input clock of the PLL, provided by the development board, should be consistent with the system clock, set to **50MHz**; PLL feedback path is set to **normal mode**. For advanced features involved, please read the reference; The output clock of the PLL compensation is *C0*; after the setting is completed, click **Next** to proceed to the next step.

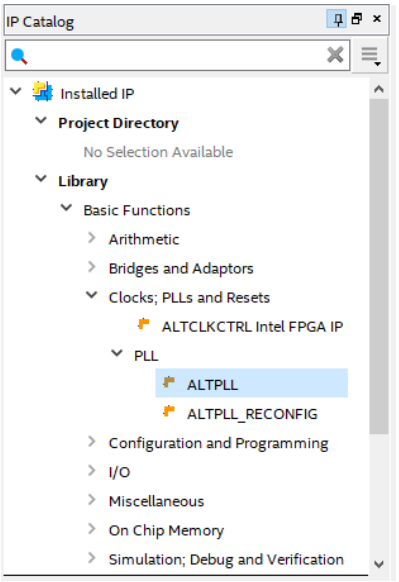


Figure 1.10 IP Catalog

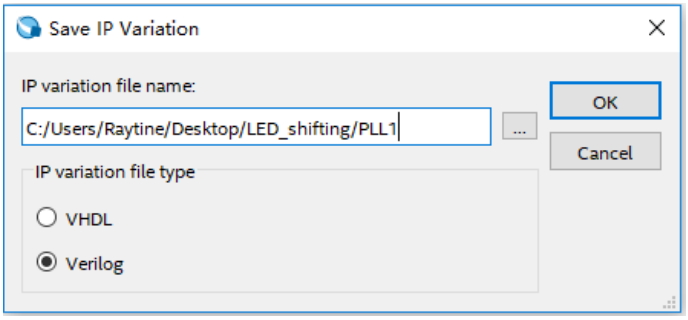


Figure 1.11 Name PLL

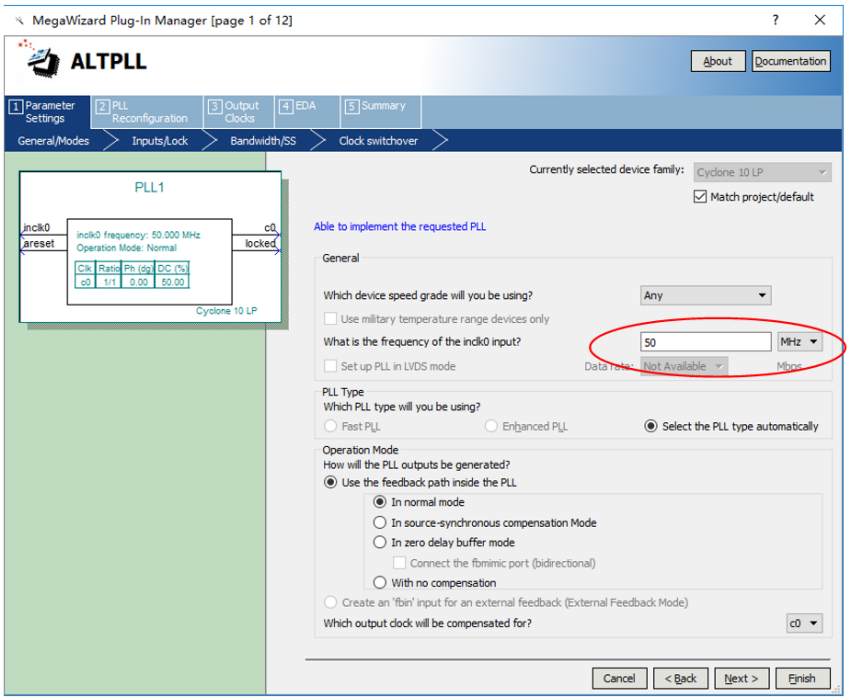


Figure 1.12 PLL setting1 (input clock setting)

（4） As shown in Figure 1.13, it is the setting of PLL asynchronous reset (areset) control and capture lock (locked) status. This experiment can be selected according to the default mode in the figure.

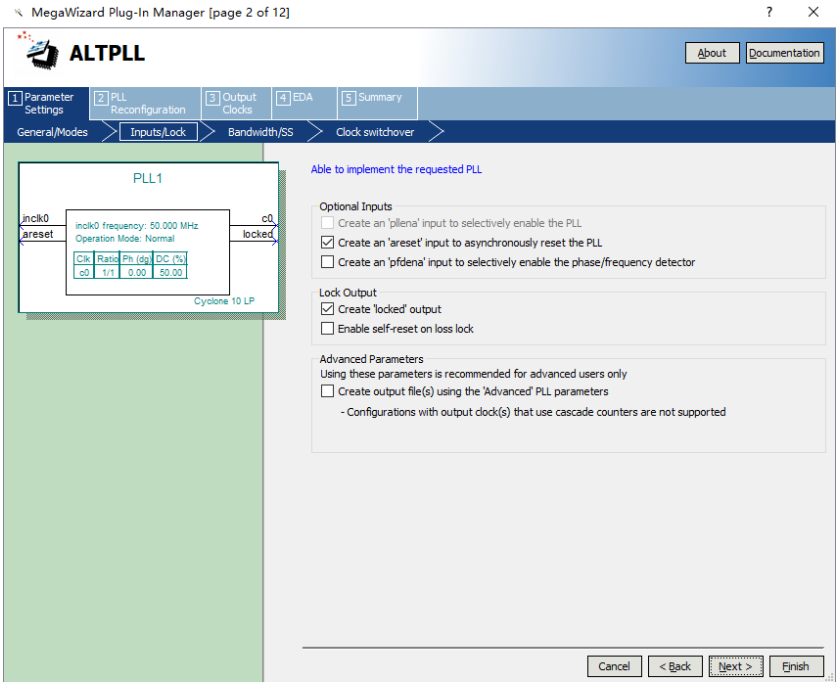


Figure 1.13 PLL setting2

（5） The contents of the next three settings pages are executed by default.

（6） As shown in Figure 1.14, it is the setting of the PLL **output clock.** It can output 5 different clocks clk c0~clk c4. This experiment only uses one, set clk *c0*, other defaults are not applicable. Set the output frequency to **100 MHz**, the **clock phase shift** to 0, and the clock duty cycle to **50%**.

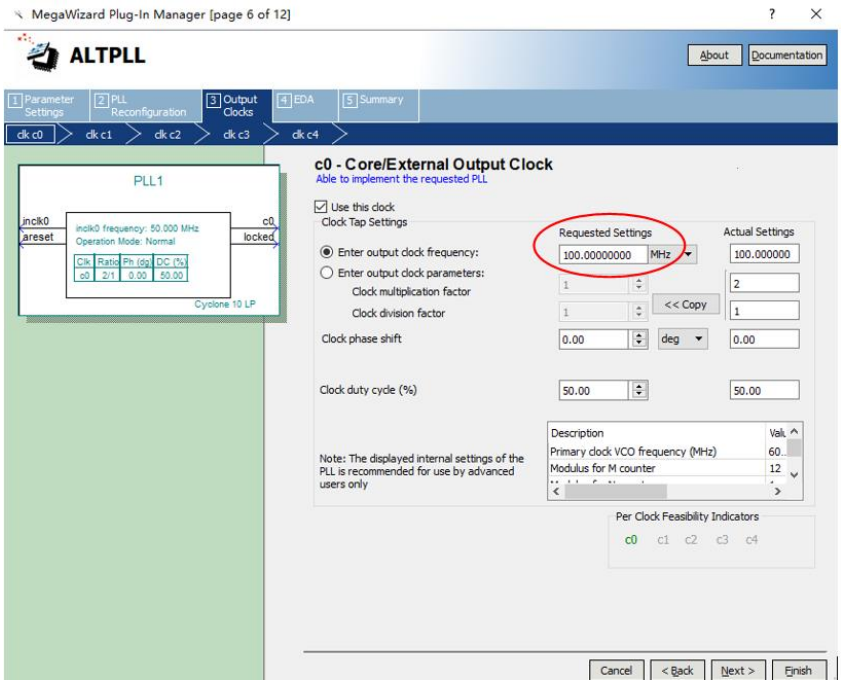


Figure 1.14 PLL setting3 (output clock setting)

（7） Keep the EDA setting to be default.

（8） As shown in Figure 1.15, the output file type setting selects *.bsf (used in the subsequent design of graphic symbols) files and *.v files. Others are set by default and click

Finish to complete the settings.

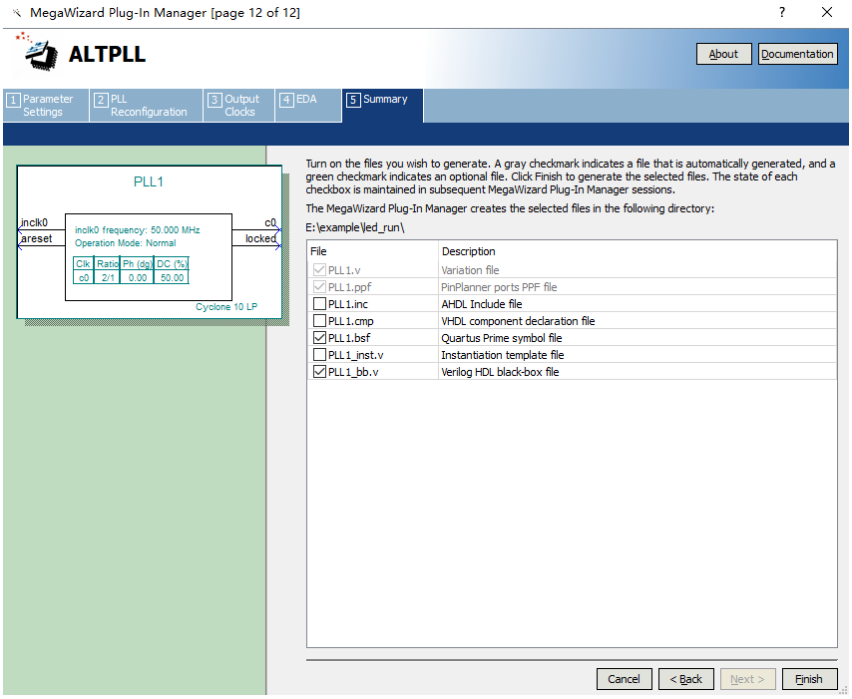


Figure 1.15 PLL settings 4 (Output File Type Settings)

（9） As shown in Figure 1.16, select file in the **Project Navigator** type box of the project interface (the default is the project hierarchy).

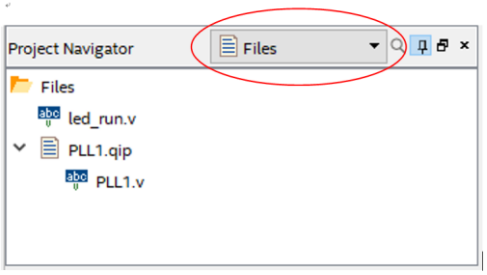


Figure 1.16 PLL1.v file setting

（10） As shown in Figure 1.17, click PLL1.v. The main window will display the contents of the PLL, find the module name and port list, copy it to the top level entity, and instantiate it.

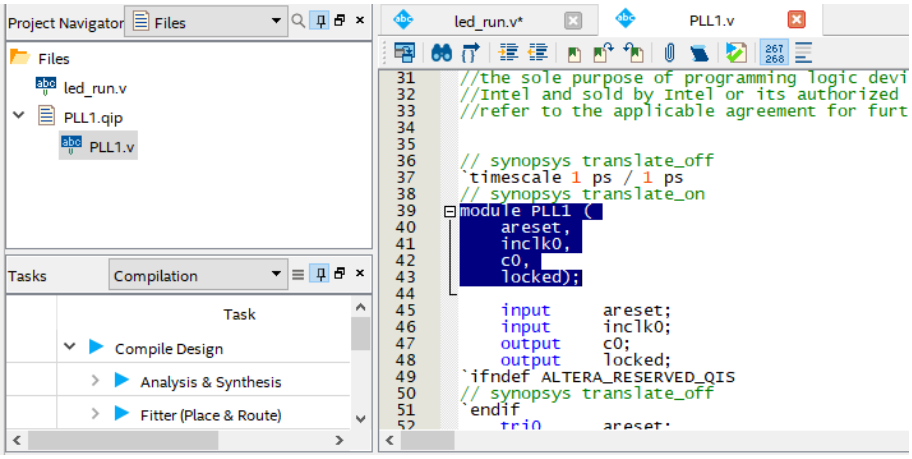


Figure 1.17 PLL1.v file

When the system is powered on, the *pll_locked* signal has a value of 0 before the PLL is locked

(stable operation), *pll_locked* is pulled high after the PLL is locked, and the clock signal *sys_clk* is output normally. The phase-locked loop is instantiated as follows:

```
wire            sys_clk;
wire            pll_locked;
PLL1 PLL1_inst
(
    .areset             (1'b0),
    .inclk0             (clk),
    .c0                 (sys_clk),
    .locked             (pll_locked)
)
```

（11） *Sys_rst* is used as the reset signal of the frequency division part, and *ext_rst* is used as the reset signal of the part of the running LED. Under the drive of the clock *sys_clk*, it is synchronously reset by the primary register.

```
reg             sys_rst;
reg             rxt_rst;
always @ (posedge sys_clk)
begin
    sys_rst <= !pll_locked;
    ext_rst <= rst;
end
```

The third step: the design of the frequency divider

We use the 100 MHz clock output by PLL as the system clock. The experiment requires the blinking speed of the running light to be 1 second. The design is firstly obtained 1us by microsecond frequency division , then dividing into milliseconds to get 1ms, and finally get 1s clock through second frequency division.

（1） Microsecond frequency division

```
reg     [6:0]   us_cnt;
reg             us_f;
always @ (posedge sys_clk)
begin
    if (sys_rst) begin
        us_cnt <= 0;
        us_f <= 1'b0;
    end
    else begin
        us_f <= 1'b0;
        if (us_cnt == 99) begin
            us_cnt <= 0;
            us_f <= 1'b1;
        end
        else
            us_cnt <= us_cnt + 1'b1;
```

```
    end
end
```

The 100 MHz clock has a period of 10ns, and 1us requires 100 clock cycles, that is, 100 10ns. Therefore, a microsecond counter *us_cnt [6:0]* and a microsecond pulse signal *us_f* are defined. The counter is cleared at reset. On each rising edge of the clock, the counter is incremented by one. When the counter is equal to 99, the period of 1us elapses, and the microsecond pulse signal *us_f* is pulled high. Thus, every 1us, this module will generate a pulse signal.

（2） Millisecond frequency divider

Similarly, 1ms is equal to 1000 1us, so a millisecond counter *ms_cnt [9:0]* is defined, a microsecond pulse signal *ms_f*.

```
reg     [9:0]   ms_cnt;
reg             ms_f;
always @ (posedge sys_clk)
begin
    if (sys_rst) begin
        ms_cnt <= 0;
        ms_f <= 1'b0;
    end
    else begin
        ms_f <= 1'b0;
        if (us_f) begin
            if (ms_cnt == 999) begin
                ms_cnt <= 0;
                ms_f <= 1'b1;
            end
            else
                ms_cnt <= ms_cnt + 1’b1;
        end
    end
end
```

（3） Second frequency divider

Similarly, 1s is equal to 1000 1ms, so a second counter *s_cnt [9:0]* is defined, one second pulse signal *s_f*. When the three counters are simultaneously full, the time passes for 1 s and the second pulse signal is issued.

```
reg     [9:0]   s_cnt;
reg             s_f;
always @ (posedge sys_clk) begin
    if (sys_rst) begin
        s_cnt <= 0;
        s_f <= 1'b0;
    end
    else begin
```

```
            s_f <= 1'b0;
            if (ms_f) begin
                if (s_cnt == 999) begin
                    s_cnt <=0;
                    s_f <= 1'b1;
            end
            else
                s_cnt <= s_cnt + 1'b1;
            end
        end
end
```

The fourth step: the design of the shifting LED

When resetting, 8 LEDs are all on, so the output *led* is 8'hff. The LEDs need to blink one by one, so the lowest LED is lit first. At this time, the *led* value is 8'b0000_0001. When the second pulse signal arrives, the next LED is illuminated, and the value of *led* is 8'b0000_0010.It can be seen that as long as the high level of "1" is shifted to the left, it can be realized by bit splicing, that is, led <= {led[6:0], led[7]}.

```
always @ (posedge sys_clk)
begin
    if (ext_rst)
        led <= 8'hff;
    else begin
        if (led == 8'hff)
            led <= 8'b0000_0001;
        else if (s_f)
            led <= {led[6:0], led[7]};
    end
end
```

1.4 Experiment Verification

1.4.1 Some Preparation Before Verification

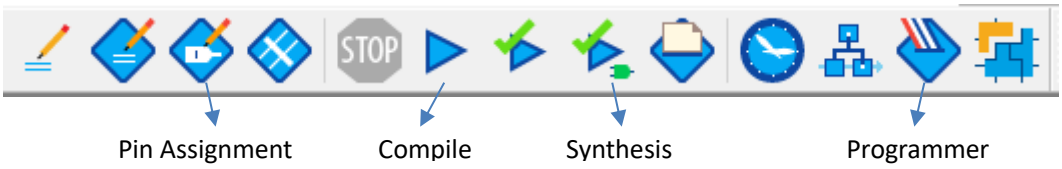


Figure 1.18 Introduction to some functions

As shown in Figure 1.18, after the program is written, analysis and synthesis is required to check for errors. Click the synthesis icon to complete, or use the shortcut key **Ctrl+K**, the pin assignment is to bind each signal to the FPGA pin, the compilation is to generate the

programming file for the development board and check the error again. Click the programmer icon, and follow the instructions to program the development board. Click on the synthesis icon, Quartus will automatically generate a report, as shown in Figure 1.19. The details of the report are not described in detail here.

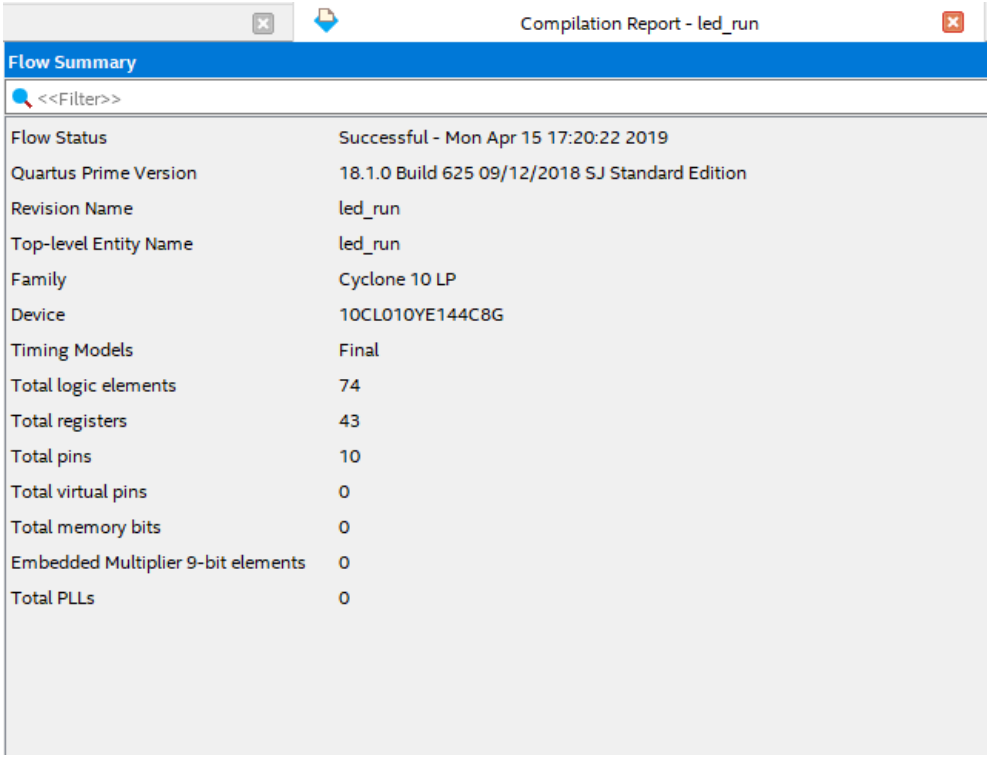


Figure 1.19 Compilation report

Check and modify to no error before board verification. Do the pin assignment before actually downloading the program to the board.

Table 1.1 Pin mapping

Signal Name	Network Label	FPGA Pin	Port Description
clk	CLK_50M	G21	Input clock
rst_n	PB3	Y6	Reset
led[7]	LED7	F2	LED 7
led[6]	LED6	F1	LED 6
led[5]	LED5	G5	LED 5
led[4]	LED4	H7	LED 4
led[3]	LED3	H6	LED 3
led[2]	LED2	H5	LED 2
led[1]	LED1	J6	LED 1
led[0]	LED0	J5	LED 0

Click the pin assignment icon to open the pin assignment window, as shown in Figure 1.20. Double-click the location bar corresponding to each pin, directly enter the corresponding pin number to confirm, or click the drop-down button to find the corresponding pin, but the latter is relatively slow. It should be noted that the I/O standard column in Figure 1.21, the content shown

is the voltage standard of each I/O port, determined by the BANK voltage in the schematics and the design requirements. In this experiment, the I/O voltage should be selected as 3.3V. Double-click the I/O standard column and click the pull-down button, as shown in Figure 1.22, select the right voltage standard. You can also set the default voltage standard in advance when selecting the chip model. Click **Device and Pin Options -> Voltage -> Default I/O standard** in Figure 1.7 to set it.

The pin assignment is complete, as shown in Figure 1.22. Then click on the compilation. After completion, program the development board.

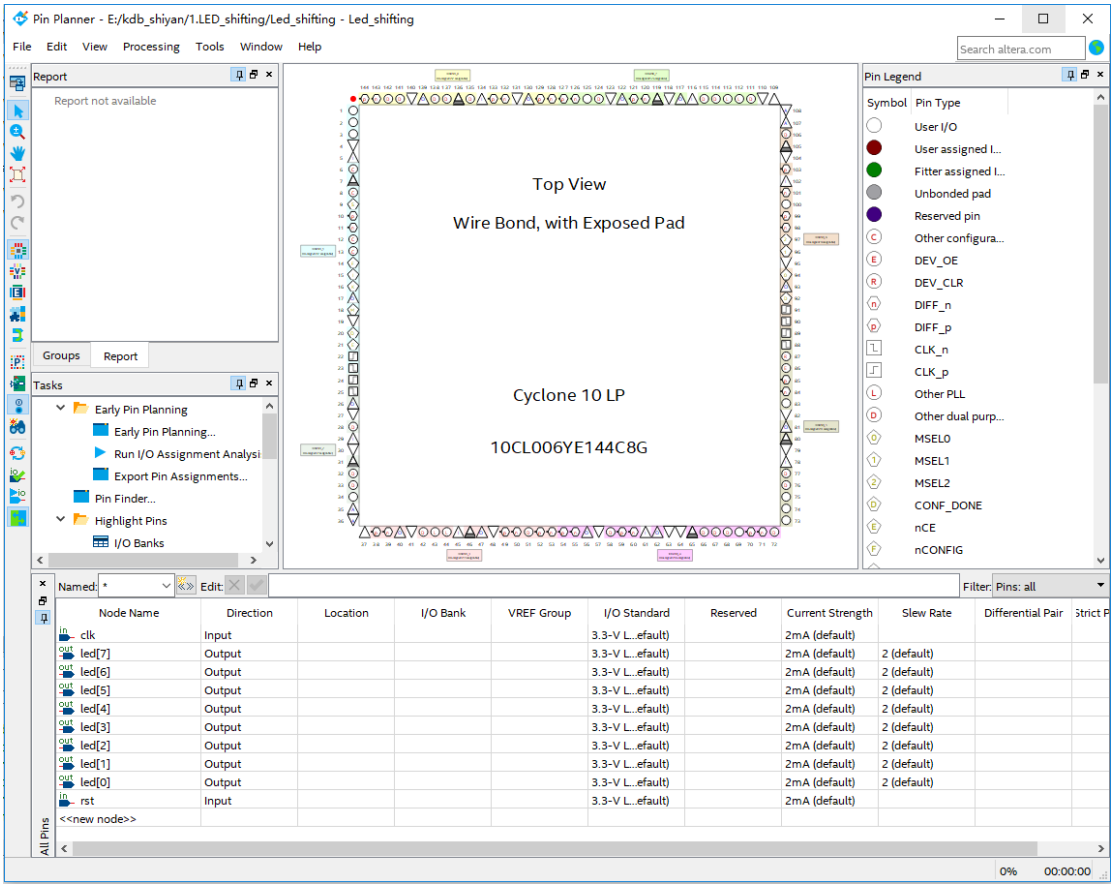


Figure 1.20 Pin assignment window

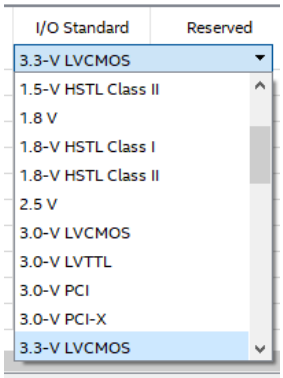


Figure 1.21 I/O voltage selection

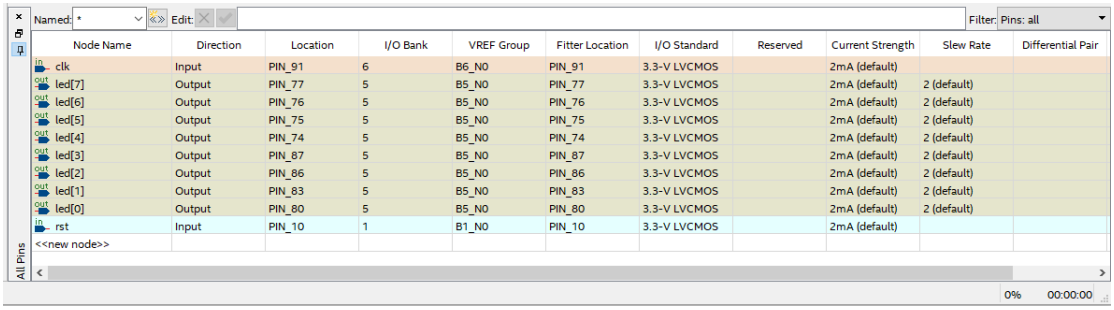


Figure 1.22 Pin assignment overview

1.4.2 Download the Program

Before programming the board, some settings should be made for the Quartus. For details, please refer to the “Intel FPGA Download Cable II User Guide” for reference. After the settings according to the instructions, click programmer icon to open the download window, as shown in Figure 1.23.

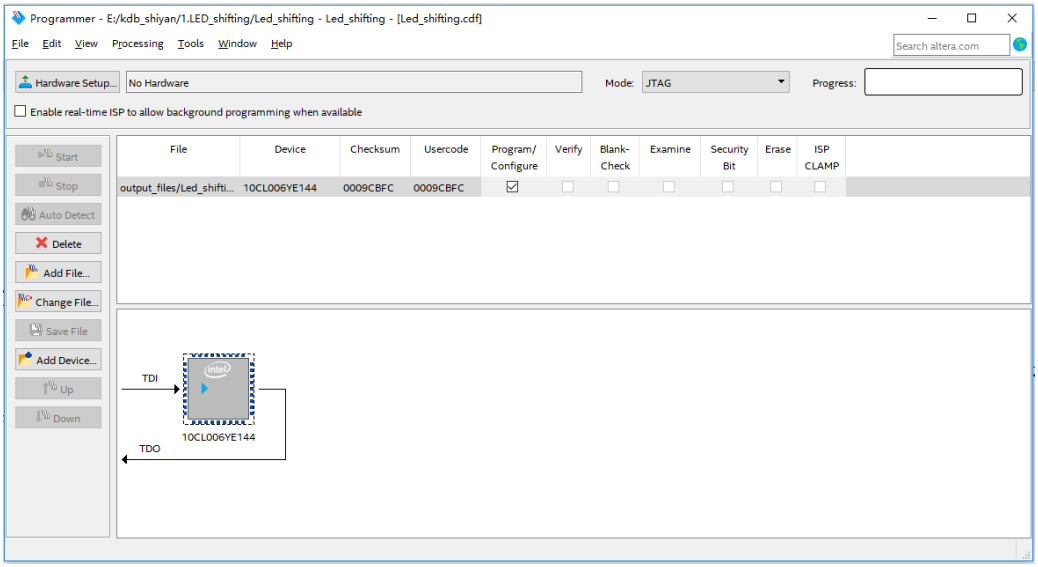


Figure 1.23 Programmer window

After connecting the development board to the host, click on **Hardware Setup** and select development board, as shown in Figure 1.24.

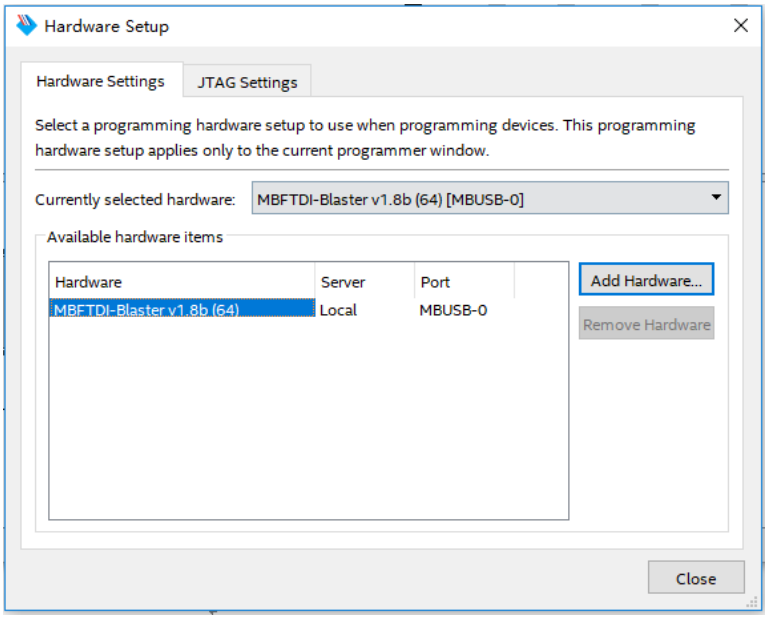


Figure 1.24 Hardwrae setup

Click **Start** to start the download, as shown in Figure 1.25, Progress shows 100% (Successful), that is, the download is completed.

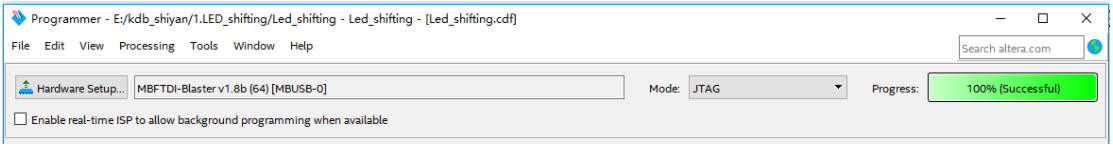


Figure 1.25 Program successfully

See Figure 1.26, the LEDs is lit from low to high, interval for one second.

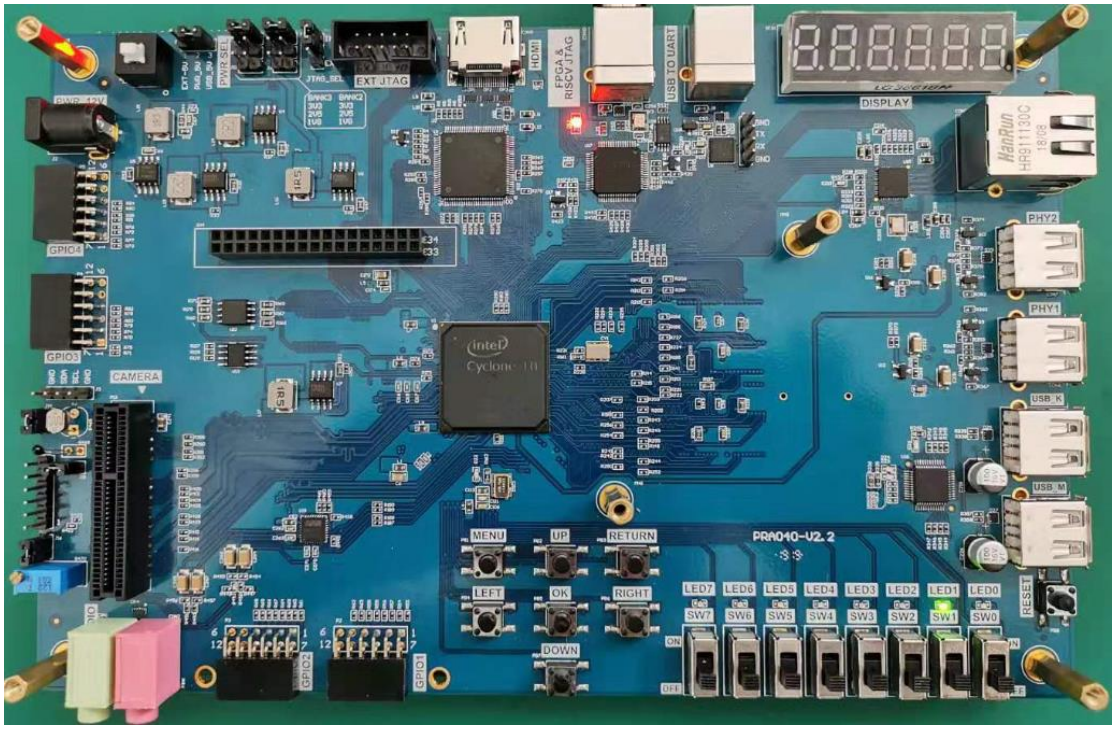


Figure 1.26 Experiment result

Experiment 2 SignalTap

2.1 Experiment Objective

（1） Continue to practice the use of the development board hardware;
（2） Practice the use of SignalTap Logic Analyzer in Quartus;
（3） Learn to analyze the captured signals.

2.2 Experiment Implement

（1） Use switches to control the LED light on and off
（2） and analyze the switching signals on the development board through the use of SignalTap.

2.3 Experiment

2.3.1 Introduction of DIP Switches and SignalTap

（1） Introduction of switches

The on-board switch is 8 DIP switches, as shown in Figure 2.1. The switch is used to switch the circuit by turning the switch handle.

Figure 2.1 Switch physical picture

（2） Introduction of SignalTap

SignalTap uses embedded logic analyzers to send signal data to SignalTap for real-time analysis of internal node signals and I/O pins when the system is operating normally.

2.3.2 Hardware Design

The schematics of the switch is shown in Figure 2.2. Port 2 of the 8 switches is connected to VCC, and port 3 is connected to the FPGA. Therefore, when the switch is toggled to port 3, the switch is turned on and input to the FPGA a high level signal.

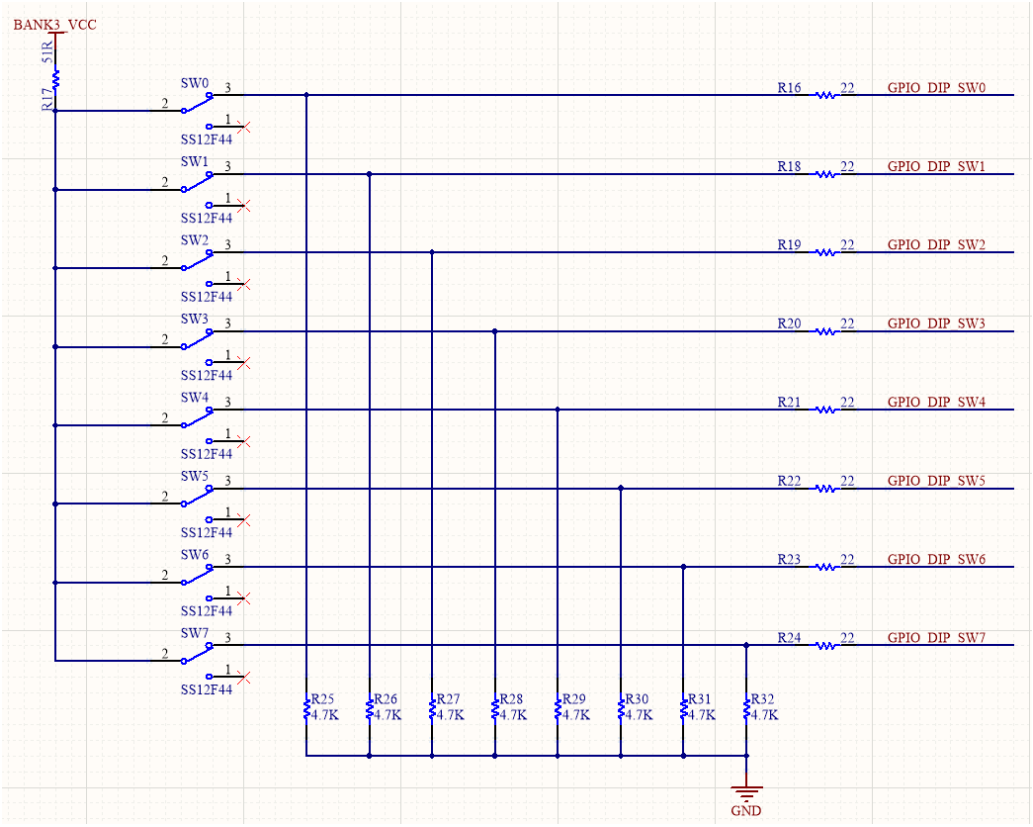


Figure 2.2 Schematics of the switches

2.3.3 Program Design

The first step: the establishment of the main program framework (interface design)

```
module SW_LED(
    input                clk,
    input        [7:0]   sw,
    output   reg [7:0]   led
);
endmodule
```

The experimental input signals have a system clock *clk* with frequency of 50 MHz, an high effective 8-bit switch *sw*, and an output 8-bit *led*.

The second step: realize the switch control LED

```
wire            sys_rst;
always @ (posedge inclk)
begin
    if (sys_rst)
        led <= 8'hff;           // All turned off
    else
        led <= ~sw;             // Determined by the state of the switch
end
```

When the reset signal is valid, all 8 LEDs are off. After the reset is completed, the LED light is turned on and off by the switch.

2.4 Use and Verification of SignalTap Logic Analyzer

The first step: pin assignment

Pin assignments are shown in Table 2.1.

Table 2.1 Pin Mapping

Signal Name	Network Label	FPGA Pin	Port Description
clk	C10_50M	G21	Input clock
SW[7]	PB7	W6	Switch 7
SW[6]	PB6	Y8	Switch 6
SW[5]	PB5	W8	Switch 5
SW[4]	PB4	V9	Switch 4
SW[3]	PB3	V10	Switch 3
SW[2]	PB2	U10	Switch 2
SW[1]	PB1	V11	Switch 1
SW[0]	PB0	U11	Switch 0
led[7]	LED7	F2	LED 7
led[6]	LED6	F1	LED 6
led[5]	LED5	G5	LED 5
led[4]	LED4	H7	LED 4
led[3]	LED3	H6	LED 3
led[2]	LED2	H5	LED 2
led[1]	LED1	J6	LED 1
led[0]	LED0	J5	LED 0

Step 2: SignalTap II startup and basic settings

Menu **Tools -> SignalTap II logic Analyzer**，

（1） In Figure 2.3, set the data under **Signal Configuration**

（2） Set the JTAG configuration and click on **Setup** to set the downloader.

（3） Set the device type by clicking **Scan Chain**

（4） Set up SOF Manager: set as *.SOF that is just compiled and generated before

（5） Clock and storage depth settings are shown in Figure 2.4.

Click the position shown in Figure 2.4 to add the clock, as shown in Figure 2.5. In the Clock Settings dialog box: Filter select **SignalTap: pre-synthesis -> List**, select the desired clock signal, select *c0* in PLL1: PLL1_INST, move to the box on the right.

Other settings can be set as shown in Figure 2.2. (for advanced use of SignalTap II, please read the reference)

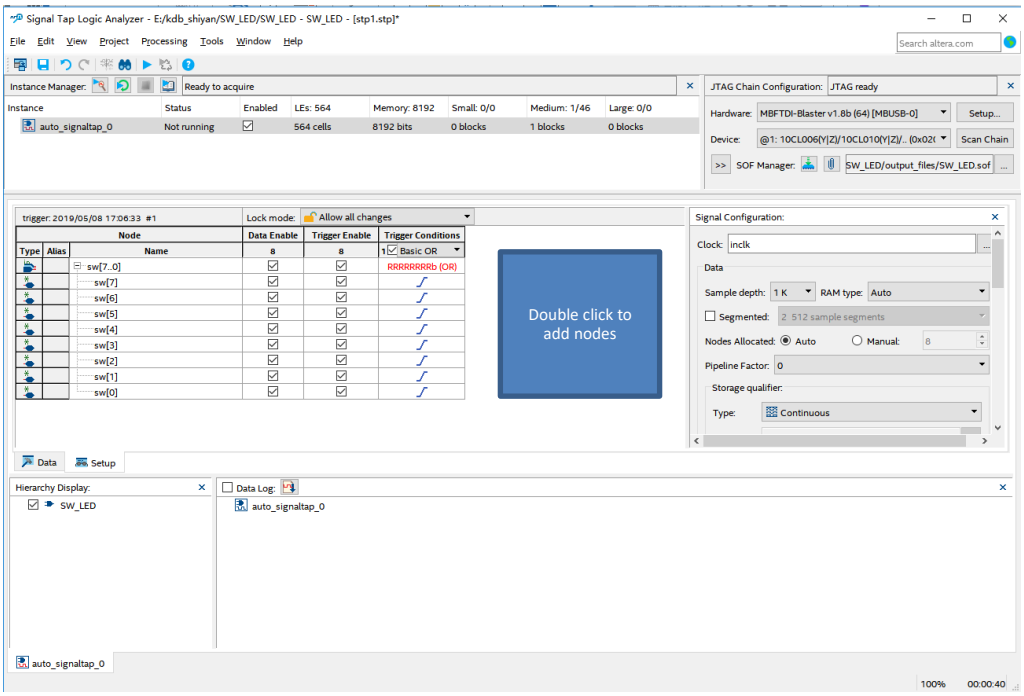


Figure 2.3 SignalTap setting interface

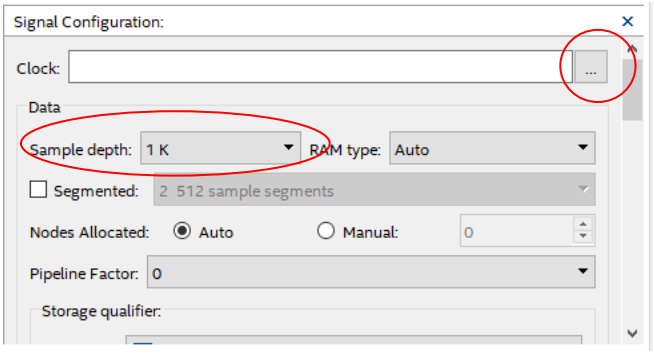


Figure 2.4 Clock signal and the sample depth

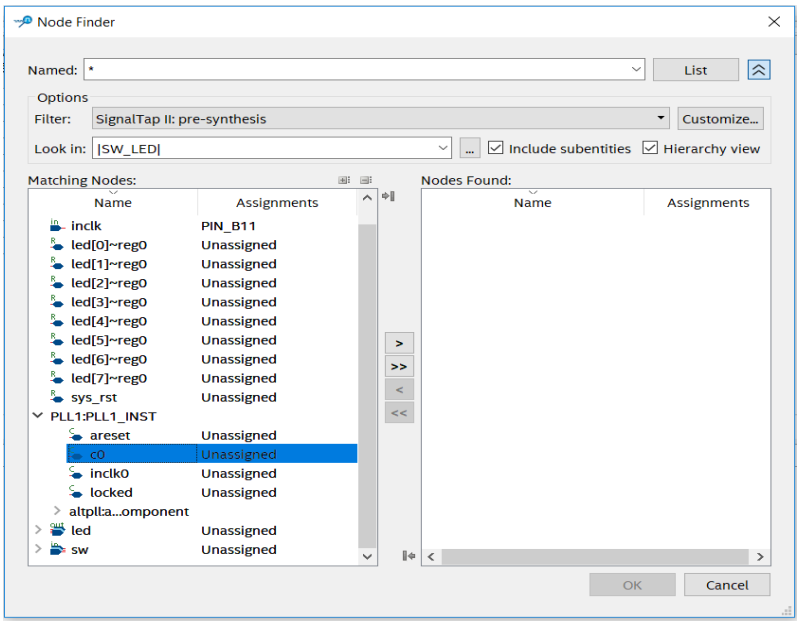


Figure 2.5 Clock signal selection dialogue

Step 3: Add observation signal

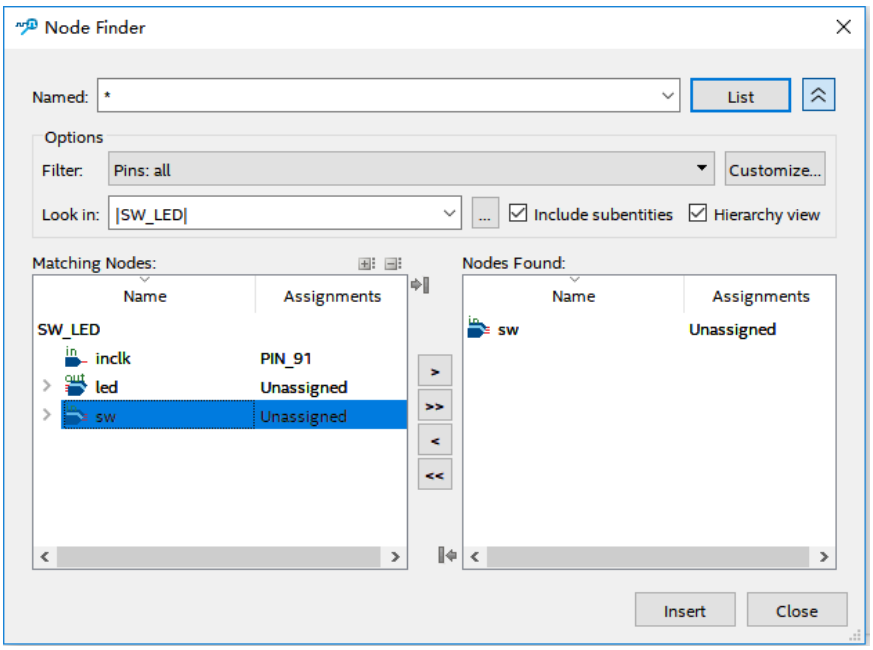


Figure 2.6 Adding interface for the observe signals

As shown in Figure 2.3, double-click the blank to add the observation signal. Add the interface as shown in Figure 2.6. Select the signal you want to observe on the left side, add it to the right side, click **Insert**. Save it and recompile.

Step 4: Settings of observe signals

For the signal to be observed, whether it is a rising edge trigger, a falling edge trigger, or not care, etc., need to be manually adjusted, as shown in Figure 2.7.

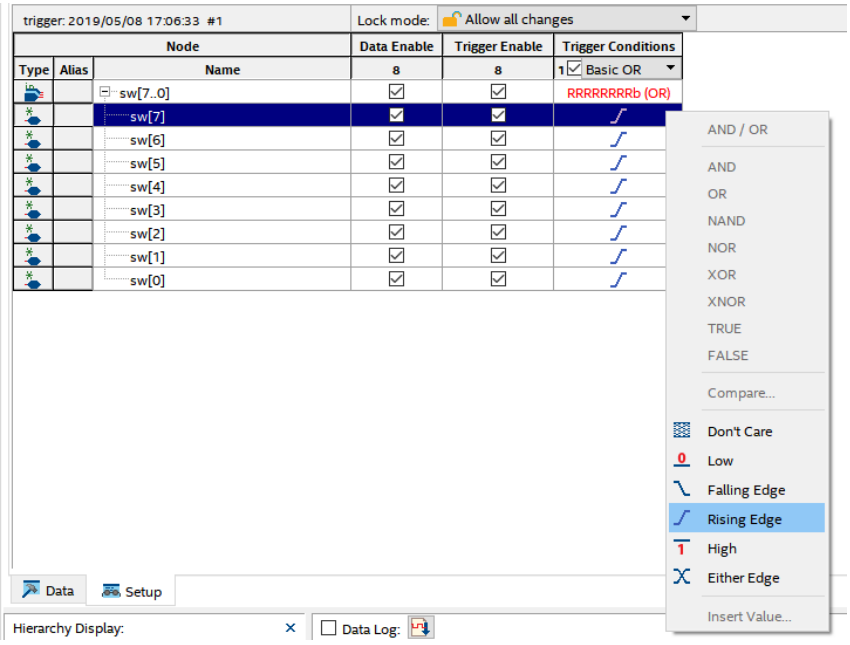


Figure 2.7 Trigger signal setting

Step 5: Observe the results

As shown in Figure 2.7, click **Run Analysis** to observe the output of SignalTap.

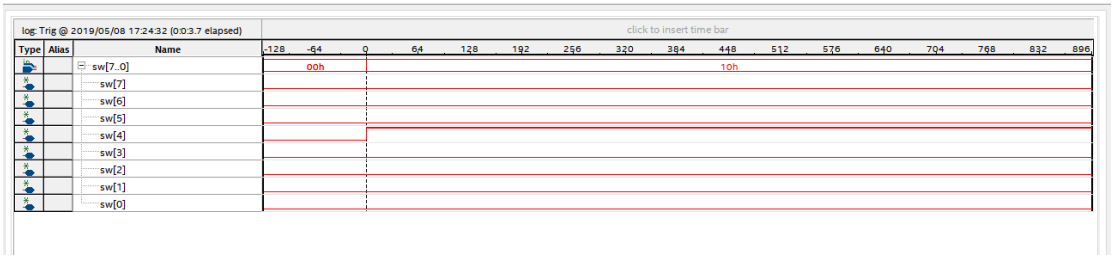


Figure 2.8 Test result

After the switch *sw[4]* is turned on, its signal is high, and the corresponding LED will be lit. Modify the Trigger condition, test the output results under different Trigger conditions, analyze and summarize.

The experimental phenomenon is shown in Figure 2.9. When the switches *sw5* and *sw1* are on, the corresponding LED5 and LED1 are illuminated, and the other LEDs remain off.

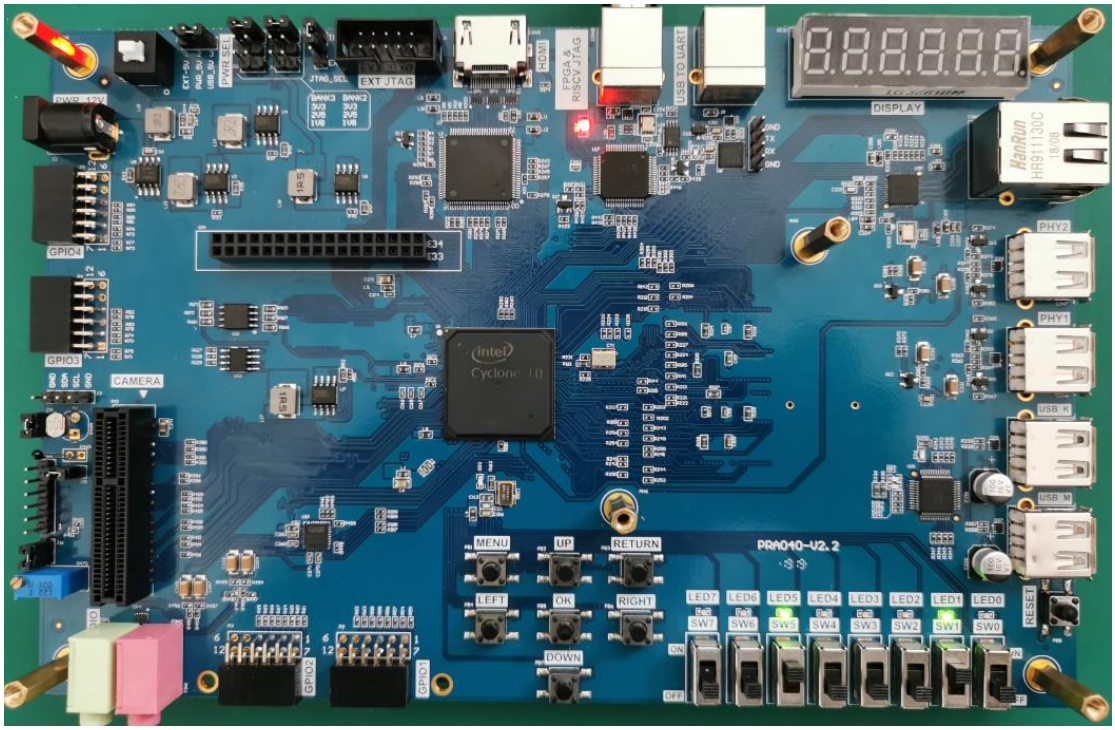


Figure 2.9 Experiment result

Experiment 3 Segment Display

3.1 Experiment Objective

（1） Review experiment 1, proficient in PLL configuration, frequency division design, and project verification;
（2） Learn the BCD code counter;
（3） Digital display decoding design;
（4） Learn to download the project into the serial FLASH of the development board;

3.2 Experiment Implement

（1） The segment display has two lower digits to display seconds, the middle two digits to display minutes, and the highest two digits to display hours.
（2） The decimal point remains off and will not be considered for the time being.

3.3 Experiment

3.3.1 Introduction to the Segment Display

One type of segment display is a semiconductor light-emitting device. The segment display can be divided into a seven-segment display and an eight-segment display The difference is that the eight-segment display has one more unit for displaying the decimal point, the basic unit is a light-emitting diode. The on-board segment display is a six-in-one eight-segment display as shown in Figure 3.1, and its structure is shown in Figure 3.2.

Figure 3.1 Segment display physcial picture

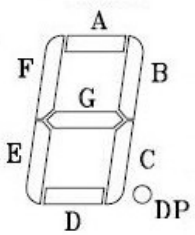


Figure 3.2 Single segment display structure

Common anode segment display are used here. That is, the anodes of the LEDs are connected. See Figure 3.3. Therefore, the FPGA is required to control the cathode of the LED to be low level, illuminate the diode, and display the corresponding information. The six-digit common anode eight-segment display refers to the signal that controls which one is lit, which is called the bit selection signal. The content displayed by each digital segment is called the segment selection signal. The corresponding truth table is shown in Table 3.1.

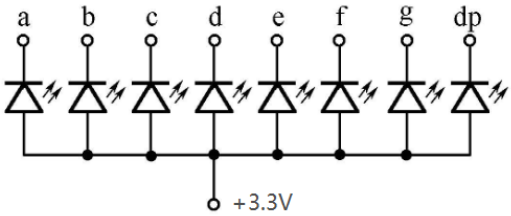


Figure 3.3 Schematics of common anode LEDs

Table 3.1 8-segment display truth table

Signal Segment	DP	G	F	E	D	C	B	A
•	0	1	1	1	1	1	1	1
0	1	1	0	0	0	0	0	0
1	1	1	1	1	1	0	0	1
2	1	0	1	0	0	1	0	0
3	1	0	1	1	0	0	0	0
4	1	0	0	1	1	0	0	1
5	1	0	0	1	0	0	1	0
6	1	0	0	0	0	0	1	0
7	1	1	1	1	1	0	0	0
8	1	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0
A	1	0	0	0	1	0	0	0
B	1	0	0	0	0	0	1	1
C	1	1	0	0	0	1	1	0
D	1	0	1	0	0	0	0	1
E	1	0	0	0	0	1	1	0
F	1	0	0	0	1	1	1	0

There are two ways to display the segment display, static display and dynamic display.

Static display: Each display segment is connected with an 8-bit data line to control and maintain the displayed glyph until the next segment selection signal arrives. The advantage is that the driver is simple, and the disadvantage is that it takes up too much I/O resources.

Dynamic display: Parallel the segment selection lines of all segment display, and the digit selection line controls which digit is valid and lights up. Through the afterglow effect of the LED and the persistence effect of the human eye, the segment display appears to be continuously lit at a certain frequency. The advantage is to save I / O resources, the disadvantage is that the driver is more complicated, the brightness is not static display high.

In this experiment, the digital tube was driven by dynamic scanning.

3.3.2 Hardware Design

The schematics of the digital tube is shown in Figure 3.4. The anode is connected to VCC through the P-channel field corresponding tube. Therefore, when the bit selection signal SEG_3V3_D[0:5] is low level 0, the FET is turned on, the anode of the segment display is high level; the cathode (segment selection signal) SEG_PA, SEG_PB, SEG_PC, SEG_PD, SEG_PE, SEG_PF, SEG_PG, SEG_DPZ are directly connected to the FPGA and directly controlled by the FPGA. Therefore, when the bit selection signal is 0, and the segment selection signal is also 0, the segment display is lit.

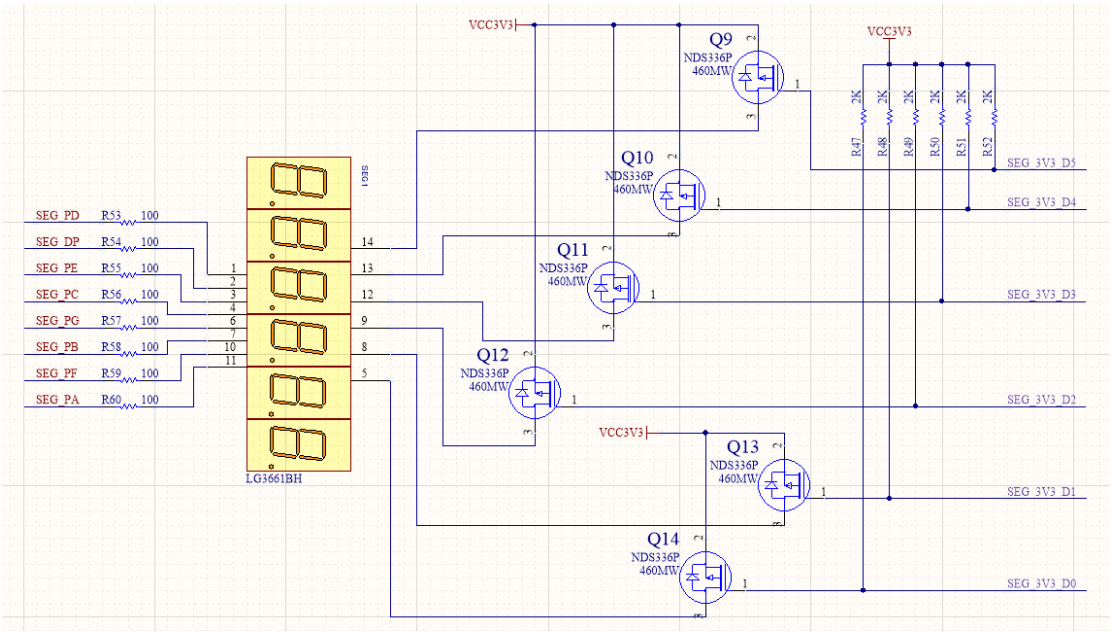


Figure 3.4 Schematics of the segment display

3.3.3 Program Design

3.3.3.1 Introduction of the Program

The first step: the establishment of the main program framework (interface design)

```
module BCD_counter (
    input           clk,
    input           rst_n,
    output  [7:0]   seven_seg,
    output  [5:0]   scan
);
endmodule
```

The input signal has a clock and a reset signal, and the output signal is a segment selection

signal *seven_seg* and a new signal *scan*.

Step 2: System Control Module

```
//Instantiate PLL
PLL  PLL_inst
(
      .areset                 (1'b0),
      .inclk0                 (clk),
      .c0                     (sys_clk),
      .locked                 (locked)
);

//Reset signal
always @ (posedge sys_clk)
begin
      sys_rst <= !locked;
      ext_rst <= rst_n;
end
```

In the first sub-module (system control module), the input clock is the system 50 MHz clock, and a 100MHz is output through the phase-locked loop as the working clock of the other sub-modules. The phase-locked loop lock signal is inverted as the system reset signal. The button is reset to be used as an external hardware reset signal.

The third step: the frequency division module

Referring to Experiment 1, a millisecond pulse signal and a second pulse signal are output as input signals of the segment display driving module.

The fourth step: segment display driver module

（1） Counting section

The counting part is similar to the frequency dividing module. It is timed by the second pulse signal for 60 seconds, 60 minutes, 24 hours, and when the time reaches 23 hours, 59 minutes and 59 seconds, the counters are all cleared, which is equivalent to one day.

（2） Segment display dynamic scanning part

```
reg      [3:0]      count_sel;
reg      [2:0]      scan_state;
always @ (posedge clk)
begin
      if (rst) begin
            scan            <= 6'b111_111;
            count_sel       <= 4'd0;
            scan_state      <= 0;
      end
      else case (scan_state)
```

```
        0    :
        begin
            scan       <= 6'b111_110;
            count_sel<= counta;
            if (ms_f)
                scan_state <= 1;
        end
        1    :
        begin
            scan       <= 6'b111_101;
            count_sel<= countb;
            if (ms_f)
                scan_state     <= 2;
        end
        2    :
        begin
            scan       <= 6'b111_011;
            count_sel<= countc;
            if (ms_f)
                scan_state <= 3;
        end
        3    :
        begin
            scan       <= 6'b110_111;
            count_sel<= countd;
            if (ms_f)
                scan_state <= 4;
        end
        4    :
        begin
            scan       <= 6'b101_111;
            count_sel<= counte;
            if (ms_f)
                scan_state <= 5;
        end
        5    :
        begin
            scan <= 6'b011_111;
            count_sel<= countf;
            if (ms_f)
                scan_state <= 0;
        end
        default    :     scan_state <= 0;
    endcase
```

```
end
```

The dynamic scanning of the segment display is realized by the state machine. A total of six segment display require six states. The state machine scan_state[2:0] is defined, and the corresponding content *count_sel* is displayed in different states. At reset, all six segment display are extinguished and jump to the 0 state. The segment display is dynamically scanned in 1 millisecond time driven by a millisecond pulse:

In the 0 state, the 0 segment display is lit, and the ones digit of the second is displayed;

In the 1 state, the first segment display is lit, and the tens digit of the second is displayed;

In the 2 state, the second segment display is lit, and the ones digit of the minute is displayed;

In the 3 state, the third segment display is lit, and the tens digit of the minute is displayed;

In the 4 state, the fourth segment display is lit, and the ones digit of the hour is displayed;

In the 5 state, the fifth segment display is lit, and the tens digit of the hour is displayed;

Part 5: segment code display section

```
always @ (*)
begin
    case (count_sel)
        0   :   seven_seg_r <= 7'b100_0000;
        1   :   seven_seg_r <= 7'b111_1001;
        2   :   seven_seg_r <= 7'b010_0100;
        3   :   seven_seg_r <= 7'b011_0000;
        4   :   seven_seg_r <= 7'b001_1001;
        5   :   seven_seg_r <= 7'b001_0010;
        6   :   seven_seg_r <= 7'b000_0010;
        7   :   seven_seg_r <= 7'b111_1000;
        8   :   seven_seg_r <= 7'b000_0000;
        9   :   seven_seg_r <= 7'b001_0000;
        default : seven_seg_r <= 7'b100_0000;
    endcase
end
```

Referring to Table 3.1, the characters to be displayed are associated with the segment code, the decimal point is set high, and then the final output segment selection signal is composed in a spliced form.

3.4 Flash Application and Experimental Verification

The first step: pin assignment

Pin assignments are shown in Table 3.1.

Table 3.1 Segment display pin mapping

Signal Name	Network Label	FPGA Pin	Port Description

clk	CLK_50M	G21	Input clock
rst_n	PB3	Y6	reset
scan[0]	SEG_3V3_D0	F14	Bit selection 0
scan[1]	SEG_3V3_D1	D19	Bit selection 1
scan[2]	SEG_3V3_D2	E15	Bit selection 2
scan[2]	SEG_3V3_D2	E13	Bit selection 3
scan[4]	SEG_3V3_D4	F11	Bit selection 4
scan[5]	SEG_3V3_D5	E12	Bit selection 5
seven_seg[0]	SEG_PA	B15	Segment a
seven_seg[1]	SEG_PB	E14	Segment b
seven_seg[2]	SEG_PC	D15	Segment c
seven_seg[3]	SEG_PD	C15	Segment d
seven_seg[4]	SEG_PE	F13	Segment e
seven_seg[5]	SEG_PF	E11	Segment f
seven_seg[6]	SEG_PG	B16	Segment g
seven_seg[7]	SEG_DP	A16	Segment h

The second step: compilation

The third step: solidify the program to Flash

Onboard Flash (N25Q128A) is a serial Flash chip that can store 128Mbit of content, which is more than enough for the engineering process in the learning process. The schematics of the Flash is shown in Figure 3.7.

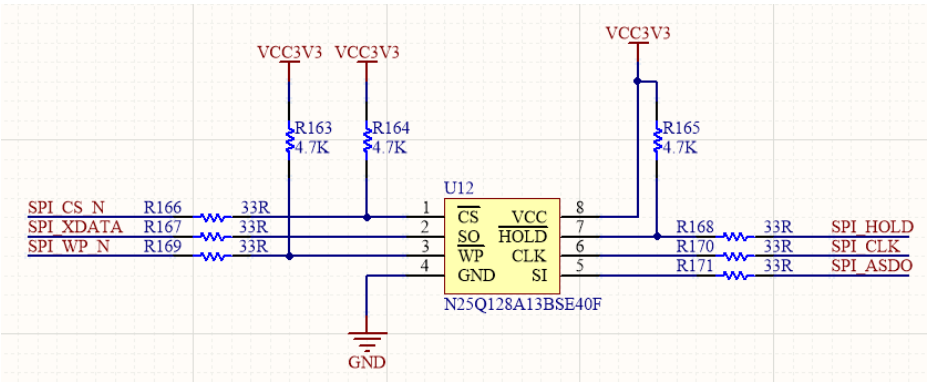


Figure 3.7 Schematics of FLASH

The function of Flash is to save the program on the development board. After the power is off, the program will not disappear. The next time the development board is powered on, it can be used directly. It is more practical in the actual learning life. Driven by the SPI_CLK clock, the FPGA downloads the program to Flash through the SPI_ASDO line. After power-on, the FPGA re-reads the program to the FPGA through SPI_XDATA for testing.

The specific configuration process of Flash is as follows:

（1） Menu File -> Convert programming files, as shown in Figure 3.8;

（2） Option settings

1) Select JTAG Indirect configuration File(*.Jic)

2) Configuration Device: EPCQ 128A (Compatible with development board N25Q128A)

3) Mode:Active serial

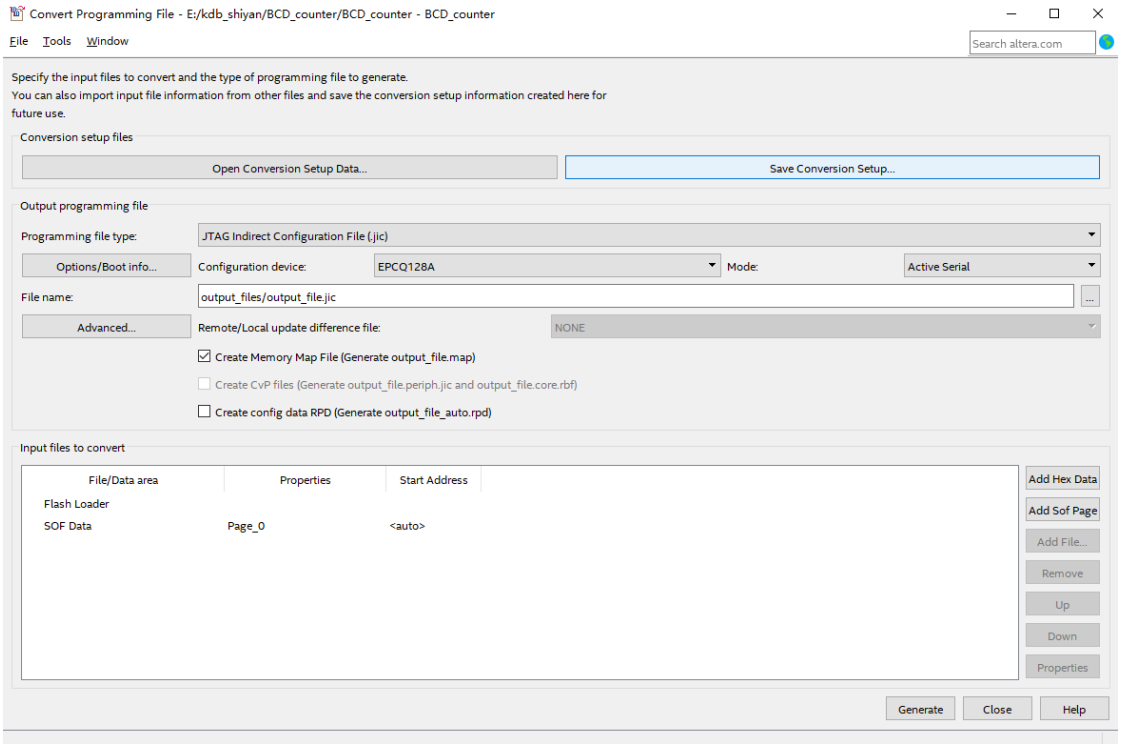


Figure 3.8 *.jic file setting

（3） Click the Advanced button and set it as shown in Figure 3.9.

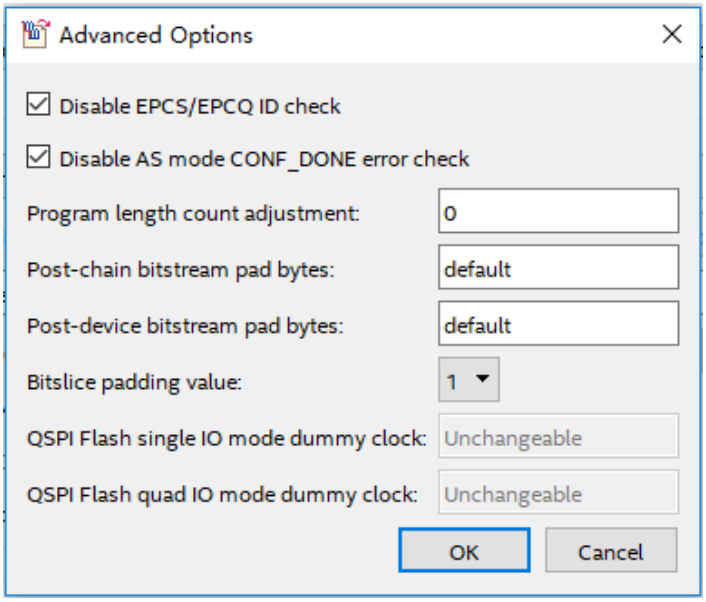


Figure 3.9 Advanced option setting

（4） Add a conversion file, as shown in Figure 3.10.

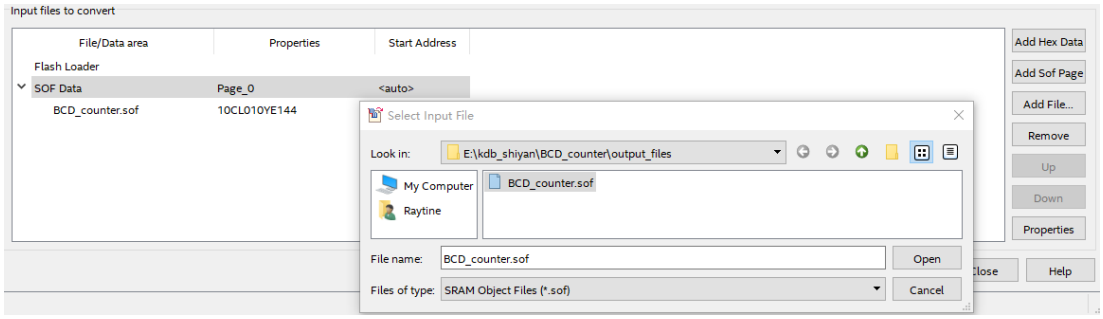


Figure 3.10 Add conversion file

（5） Add a device, as shown in Figure 3.11.

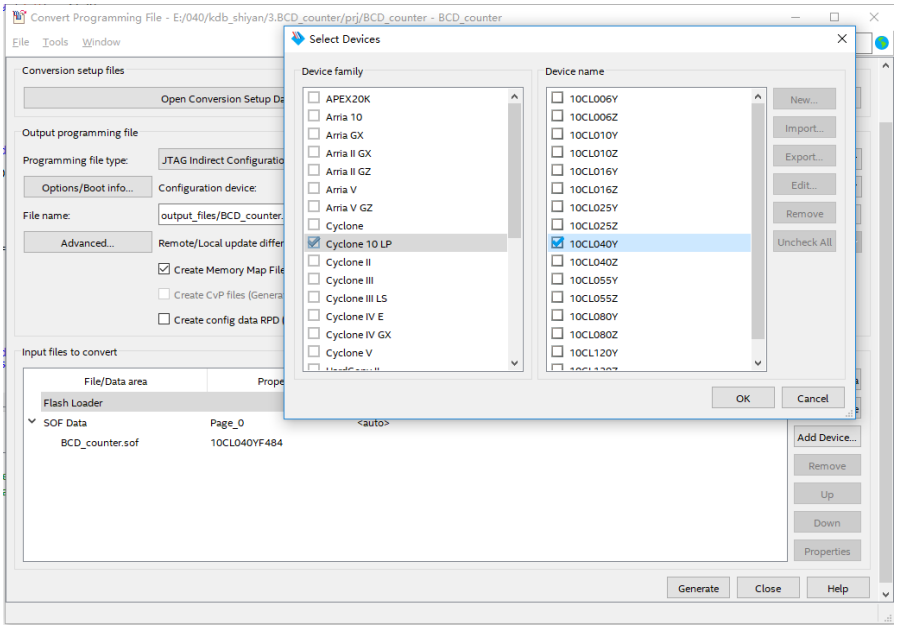


Figure 3.11 Add devices

（6） Click **Generate** to generate the *BCD_counter.jic* file

（7） Consistent with previous program verification operations, select the correct file (*.jic) to download

The fourth step: power up verification

shown in Figure 3.12, after power-on, the FPGA automatically reads the program in Flash into the FPGA and runs it.

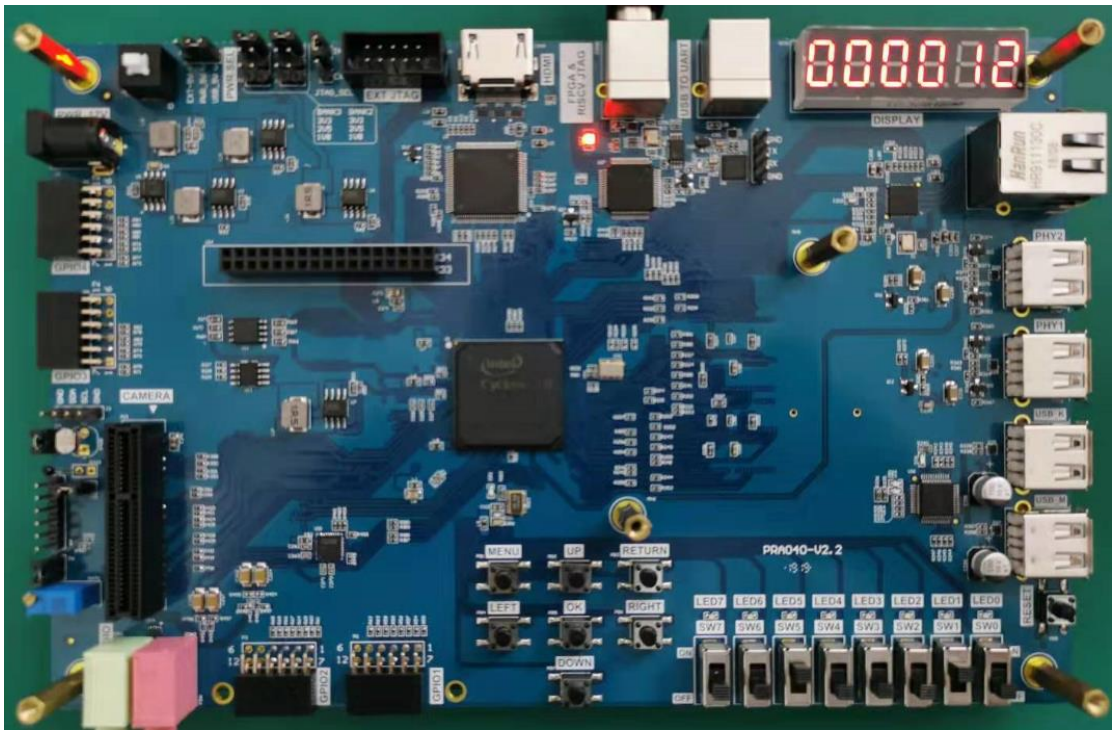


Figure 3.12 Experiment result

Experiment 4 Block/SCH

4.1 Experiment Objective

（1） Review building new FPGA projects in Quartus, device selection, PLL creation, PLL frequency setting, Verilog's tree hierarchy design, and the use of SignalTap II

（2） Master the design method of graphics from top to bottom

（3） Combined with the BCD_counter project to achieve the display of the digital clock

（4） Observe the experimental results

4.2 Experiment Implement

Use schematics design to build the project.

4.3 Experiment

This experiment is mainly to master another design method. The other design contents are basically the same as the experiment 3, and will not be introduced in detail. The modular design method is introduced below.

（1） New project: **File -> New Project Wizard**...

Project name: block_counter

Select device (10cl010YE144c8G)

（2） Create nre file; File -> New, select **Blcok Diagram/Schematic File**. See Figure 4.1.

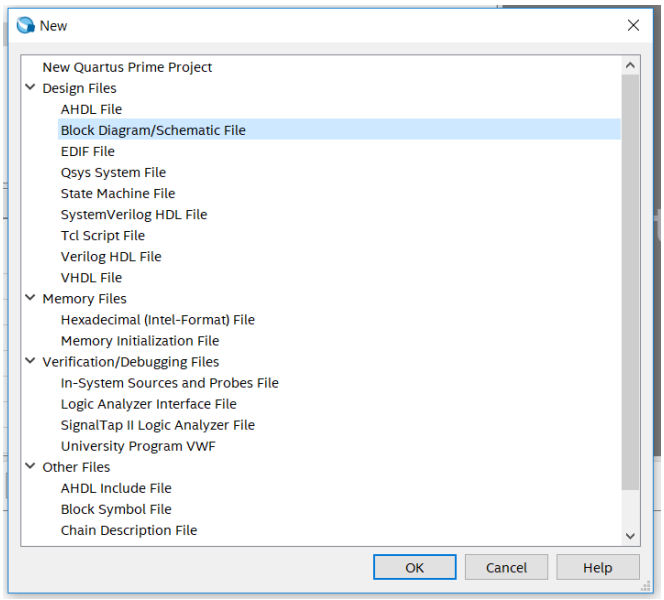


Figure 4.1 New file

（3） Create as shown in Figure 4.2, the middle part is the graphic design area, which can be used for Block/SCH design.

1) Save the file as *block_counter.bdf*

2) Double-click the blank space in the graphic design area to add a symbol

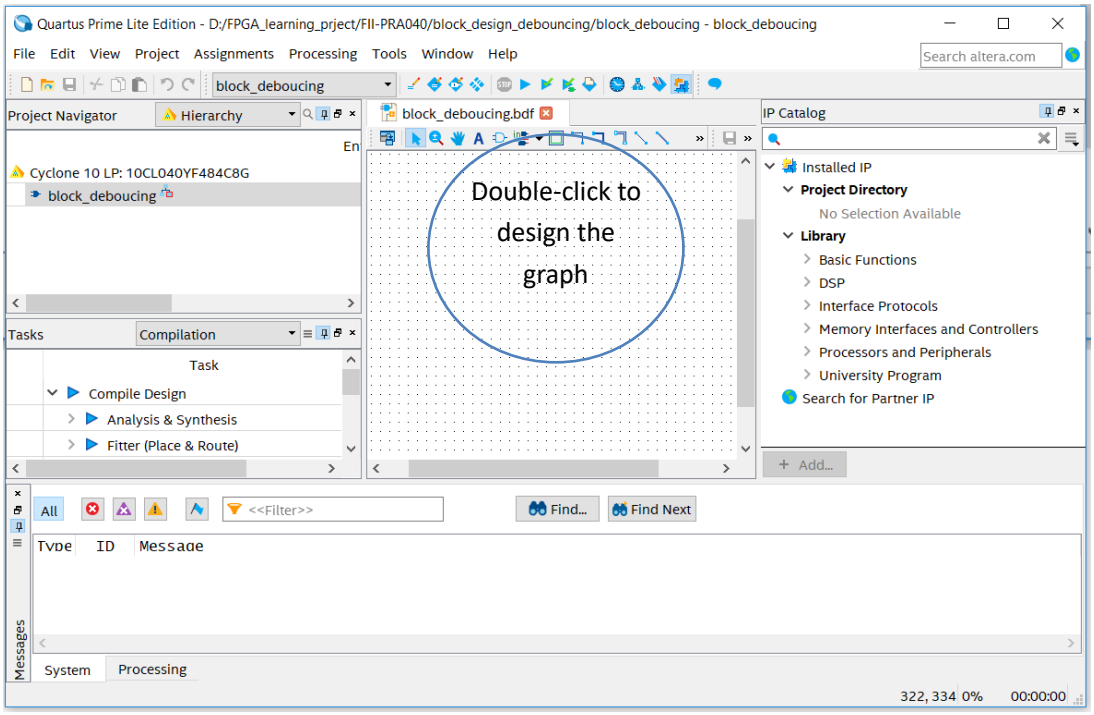


Figure 4.2 Graph design interface

（4） Graphic editing

-click on the graphic design area to select the appropriate library and device in the libraries

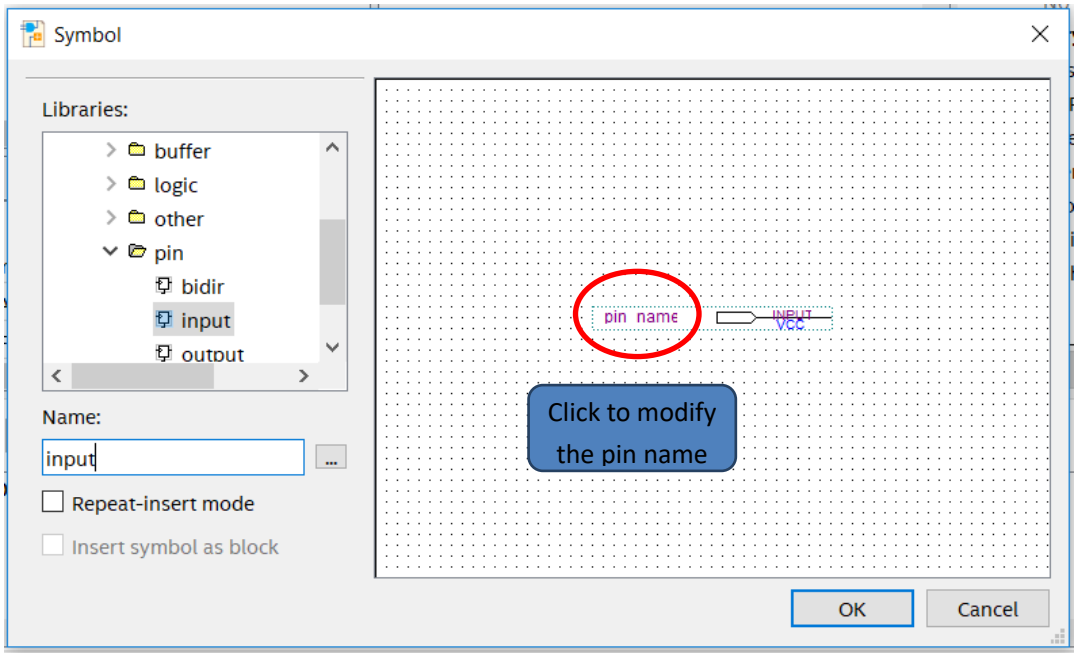


Figure 4.3 Input symbol

（5） Add input, output, and modify their names

（6） Add a custom symbol

1) Create a new block/sch file and store it as *PLL_sys.bdf*

2) Add PLL IP, refer to experiment 1

3) Select the generated file to include the *PLL1.bsf* file

4) Double-click in the blank area of the *PLL_sys.bdf* file to select the PLL1 symbol just generated and add it to the file, as shown in Figure 4.4.

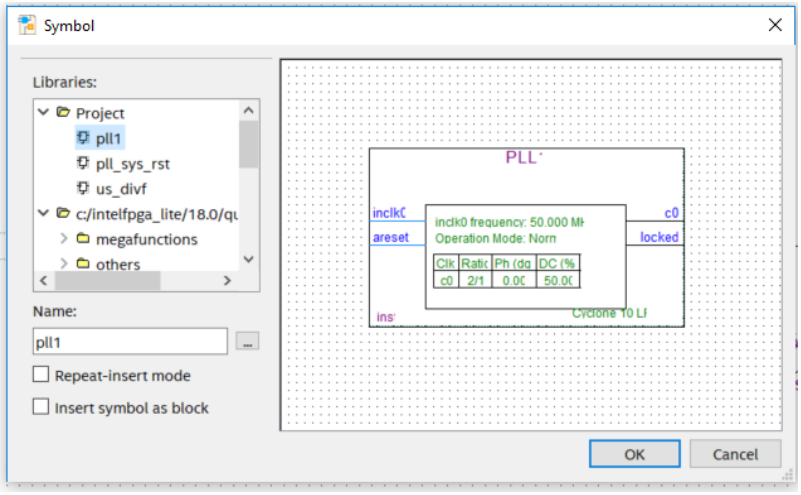


Figure 4.4 Invoke the symbols generated in the IP catalog in the graphical editing interface

5) Continue to add other symbols, input, output, dff, GND, etc. and connect them, as shown in Figure 4.5.

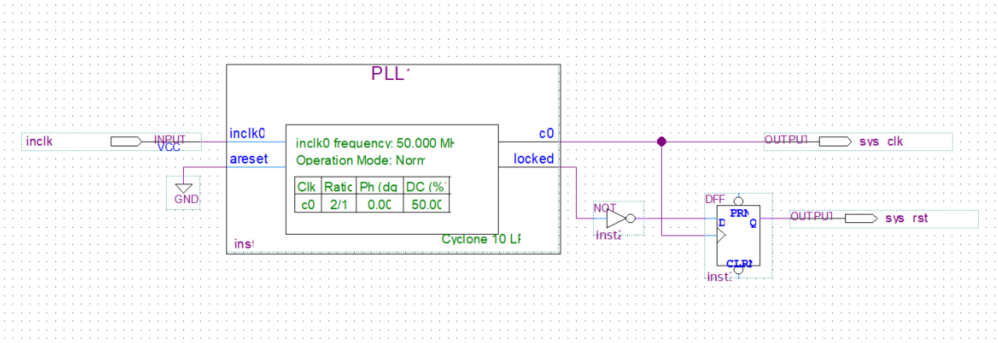


Figure 4.5 Connect the device

（7） Recreate the newly created file symbol for graphic editing to use in subsequent design

1) **File -> Create/Update -> Create Symbol file for Current File**. See Figure 4.6.

2) Generate *PLL_sys.bsf*

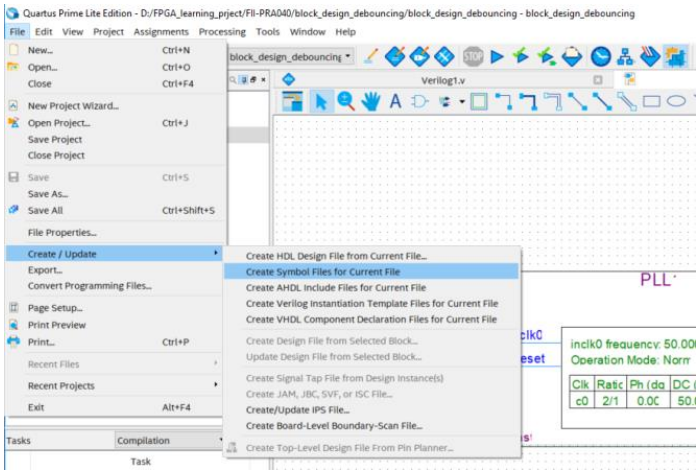


Figure 4.6 Creating a symbol file for the current file (symbol file *.bsf)

（8） Create a frequency division module

1) Create a new verilog file *div_us* for the divider (Refer project files for the code)

2) The PLL output clock is used as its own input clock, and the 100 MHz clock is divided into 1 MHz clocks.

3) Repeat (7) to create *div_us.bsf*

4) Create a new 1000 frequency division verilog file: *div_1000f.v*

5) Create *div_1000f.bsf* symbol

（9） Create the output pulse us, ms, second module, as shown in Figure 4.7. Refer the specific implementation to the reference code and the frequency division design of the experiment 1 and 3

1) Create a new block/sch file *block_div* and add the designed graphic symbol file to *block_div.bdf*

2) Repeat (7) to create the *block_div.bsf* symbol

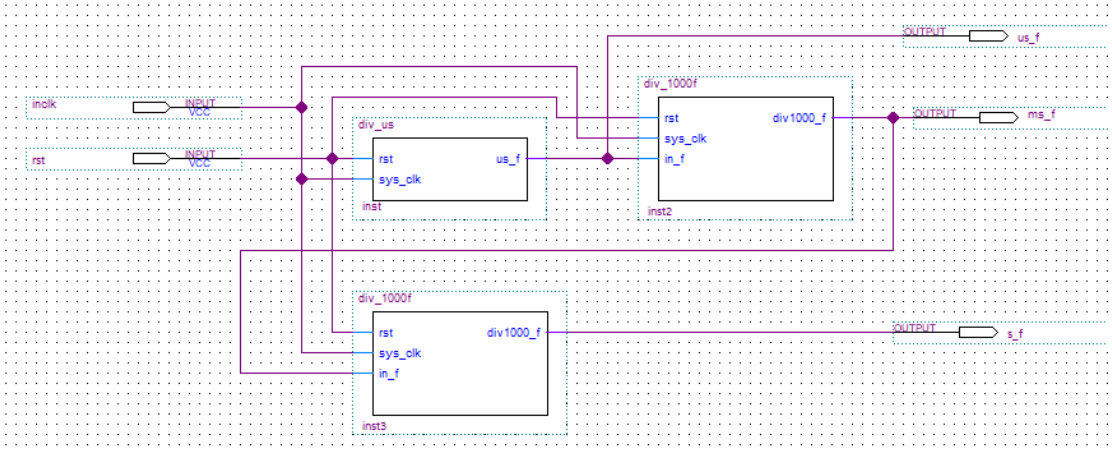


Figure 4.7 us, ms, second pulse of block/sch design

（10） Create a new verilog file *bcd_counter.v*, design the hour and minute counter, and create the bsf symbol. Refer to experiment 3, and implement part of the frequency division using *block_div* in (9).

（11） Combine each **.bsf* and complete the design of the digital clock (*block_counter.bdf*), as shown in Figure 4.8.

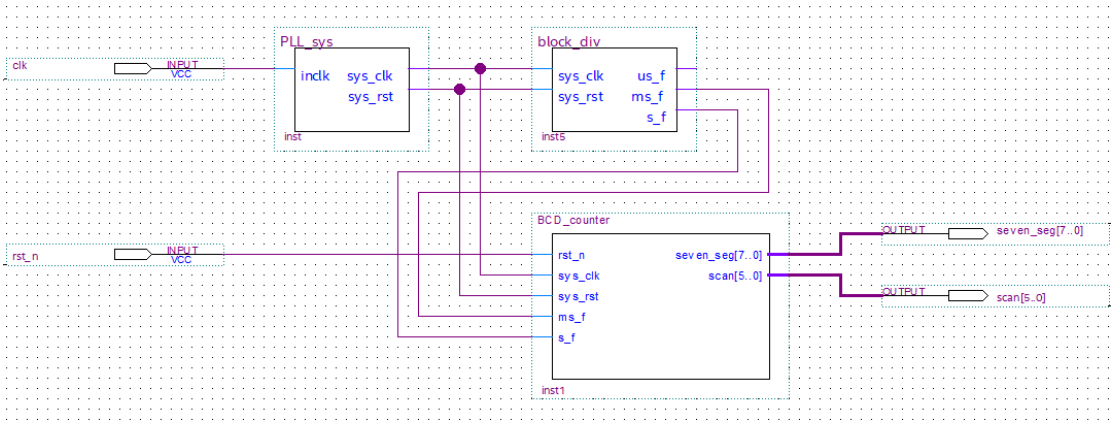


Figure 4.8 Digital clock for BDF design

4.4 Experiment Verification

Pin assignment, compilation, and program verification are consistent with Experiment 3. For reference, see Experiment 3, which is elimibated here.

Experiment 5 Button Debounce

5.1 Experiment Objective

（1） Review the design process of the shifting LED
（2） Learn button debounce principle and adaptive programming
（3） the connection and use of the Fii-PRA040 button hardware circuit
（4） Comprehensive application button debounce and other conforming programming

5.2 Experiment Implement

（1） Control the movement of the lit LED by pressing the button
（2） Each time the button is pressed, the lit LED moves one bit.
（3） When the left shift button is pressed, the water lamp moves to the left, presses the right button, and the water lamp moves to the right.

5.3 Experiment

5.3.1 Introduction to Button and Debounce Principle

（1） Introduction to button

The on-board button is a common push button, which is valid when pressed, and automatically pops up when released. A total of eight, respectively, PB1 (MENU), PB2 (UP), PB3 (RETURN), PB4 (LETF), PB5 (OK), PB6 (RIGHT), PB7 (DOWN) and a hardware reset button (RESET). As shown in Figure 5.1.

Figure 5.1 Button physical picture

（2） Introduction to button debounce

As long as mechanical buttons are used, instability should be considered. Usually, the switches used for the buttons are mechanical elastic switches. When the mechanical contacts are opened and closed, due to the elastic action of the mechanical contacts, a push button switch does not immediately turn on when closed, nor is it off when disconnected. Instead, there is some bouncing when connecting and disconnecting. See Fig 5. 2.

The length of the button's stable closing time is determined by the operator. It usually takes more than 100ms. If you press it quickly, it will reach 40-50ms. It is difficult to make it even shorter. The bouncing time is determined by the mechanical characteristics of the button. It is usually between a few milliseconds and tens of milliseconds. To ensure that the program responds to the button's every on and off, it must be debounced. When the change of the button state is detected, it should not be immediately responding to the action, but waiting for the closure or the disconnection to be stabilized before processing. Button debounce can be divided into hardware debounce and software debounce.

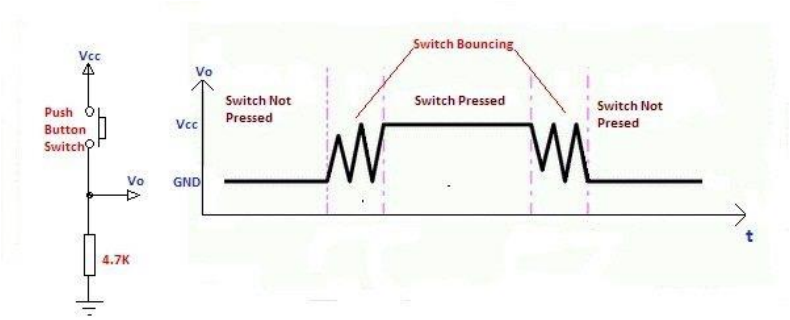


Fig 5. 2 Button bounce principle

In most of cases, we use software or programs to achieve debounce. The simplest debounce principle is to wait for a delay time of about 10ms after detecting the change of the button state, and then perform the button state detection again after the bounce disappears. If the state is the same as the previous state just detected, the button can be confirmed. The action has been stabilized. This type of detection is widely used in traditional software design. However, as the number of button usage increases, or the buttons of different qualities will react differently. If the delay is too short, the bounce cannot be filtered out. When the delay is too long, it affects the sensitivity of the button.

5.3.2 Hardware Design

The schematics is shown in Figure 5.3. One side of the button (P1, P2) is connected to GND, and the other side (P3, P4) is connected to the FPGA. At the same time, VCC is connected through a 10 kohm resistor. In the normal state, the button is left floating, thus the potential of the button P3 is 1, so the input value of the button to the FPGA is 1; when the button is pressed, the buttons are turned on both sides, and the potential of the button P3 is 0, so the input value of the button to the FPGA is 0. So the onboard switch is active low.

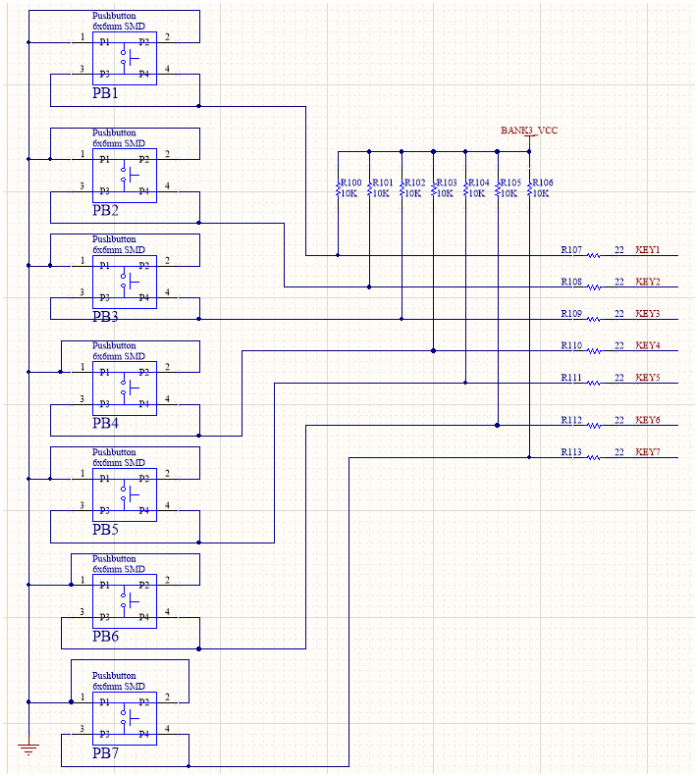


Figure 5.3 Schematics of the buttons

5.3.3 Program Design

5.3.3.1 Top Level Design

See Figure 5. 4.

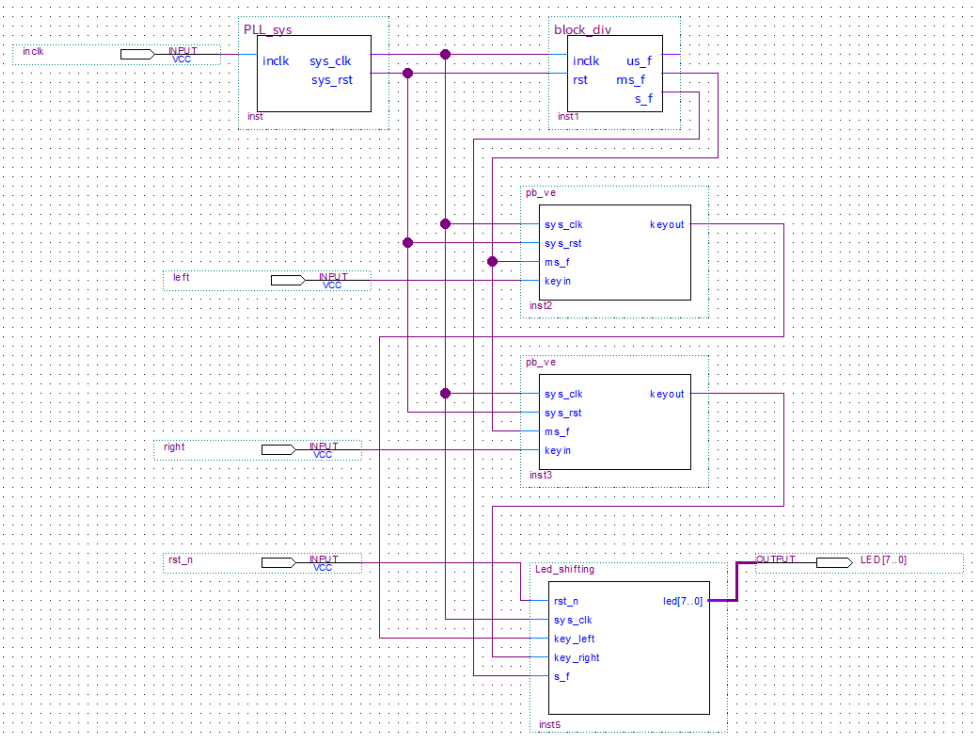


Figure 5.4 Top level design

5.3.3.2 Introduction to the program

Refer to the previous experiments for the frequency division module and the LED display module. Here, a new part of the button debounce module is introduced. This chapter introduces an adaptive button debounce method: starts timing when a change in the state of the button is detected. If the state changes within 10ms, the button bouncing exists. It returns to the initial state, clears the delay counter, and re-detects the button state until the delay counter counts to 10ms. The same debounce method is used for pressing and releasing the button. The flow chart is shown in Fig 5. 5.Case 0 and 1 debounce the button press state. Case 2 and 3 debounce the button release state. After finishing the whole debounce procedure, the program outputs a synchronized clock pulse.

```
module pb_ve (
    input                   sys_clk,
    input                   sys_rst,
    input                   ms_f,
    input                   keyin,
    output                  keyout
);
    reg                     keyin_r;
    reg                     keyout_r;
    reg             [1:0]   ve_key_st;
    reg             [3:0]   ve_key_count;
always @ (posedge sys_clk)
begin
    keyin_r <= keyin;
end
always @ (posedge sys_clk)
begin
    if (sys_rst) begin
        keyout_r        <= 1'b0;
        ve_key_count    <= 0;
        ve_key_st       <= 0;
    end
    else case (ve_key_st)
        0   :
        begin
            keyout_r        <= 1'b0;
            ve_key_count    <= 0;
            if (!keyin_r)
                ve_key_st <= 1;
        end
        1   :
        begin
```

```
                if (keyin_r)
                    ve_key_st <= 0;
                else begin
                    if (ve_key_count == 10)
                        ve_key_st <= 2;
                    else if (ms_f)
                        ve_key_count <= ve_key_count + 1'b1;
                end
            end
            2    :
            begin
                ve_key_count  <= 0;
                if (keyin_r)
                    ve_key_st <= 3;
            end
            3    :
            begin
                if (!keyin_r)
                    ve_key_st <= 2;
                else begin
                    if (ve_key_count == 10) begin
                        ve_key_st       <= 0;
                        keyout_r  <= 1'b1;
                    end
                    else if (ms_f)
                        ve_key_count <= ve_key_count + 1'b1;
                end
            end
            default    :    ;
        endcase
end
   assign keyout = keyout_r;
endmodule
```

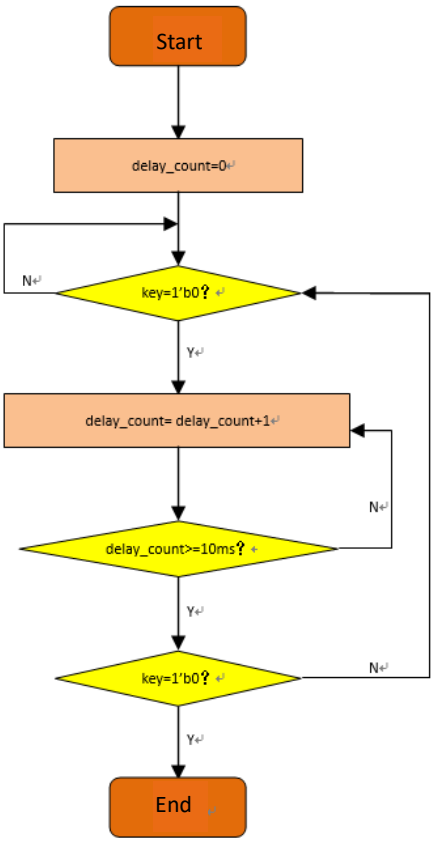


Figure 5.5 Button deboucne flow chart

5.4 Experiment Verification

The first step: pin assignment

Table 5.1 Pin mapping

Signal Name	Network Label	FPGA Pin	Port Description
left	PB44	AB4	Left shift signal
right	PB6	AA4	Right shift signal
clk	CLK_50M	G21	Input clock
rst_n	PB3	Y6	Reset
led[7]	LED7	F2	LED 7
led[6]	LED6	F1	LED 6
led[5]	LED5	G5	LED 5
led[4]	LED4	H7	LED 4
led[3]	LED3	H6	LED 3
led[2]	LED2	H5	LED 2
led[1]	LED1	J6	LED 1
led[0]	LED0	J5	LED 0

Step 2: download the program to verify

After the pin assignment is completed, the compilation is performed, and the programmer is

verified after passing. The experimental phenomenon is shown in Figures below.

All LEDs are lit after successfully programmed. See Figure 5.6.

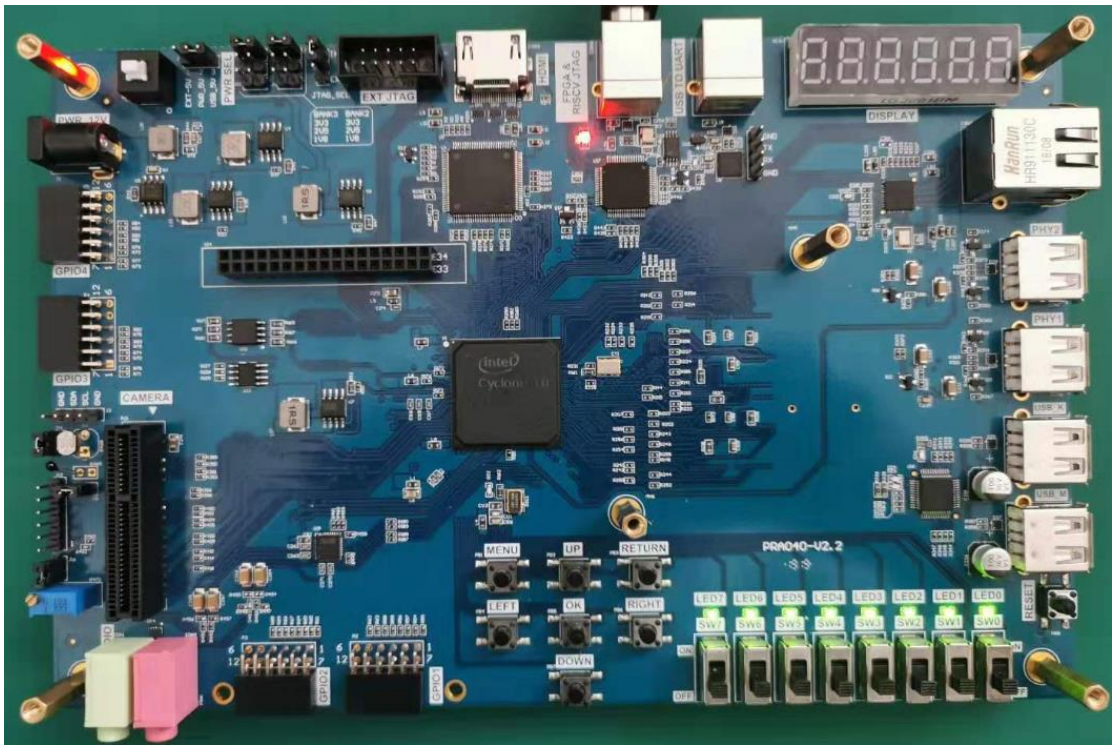


Figure 5.6 Experiment Result(reset)

When the right shift button is pressed, the highest LED lights up. See Figure 5.7.

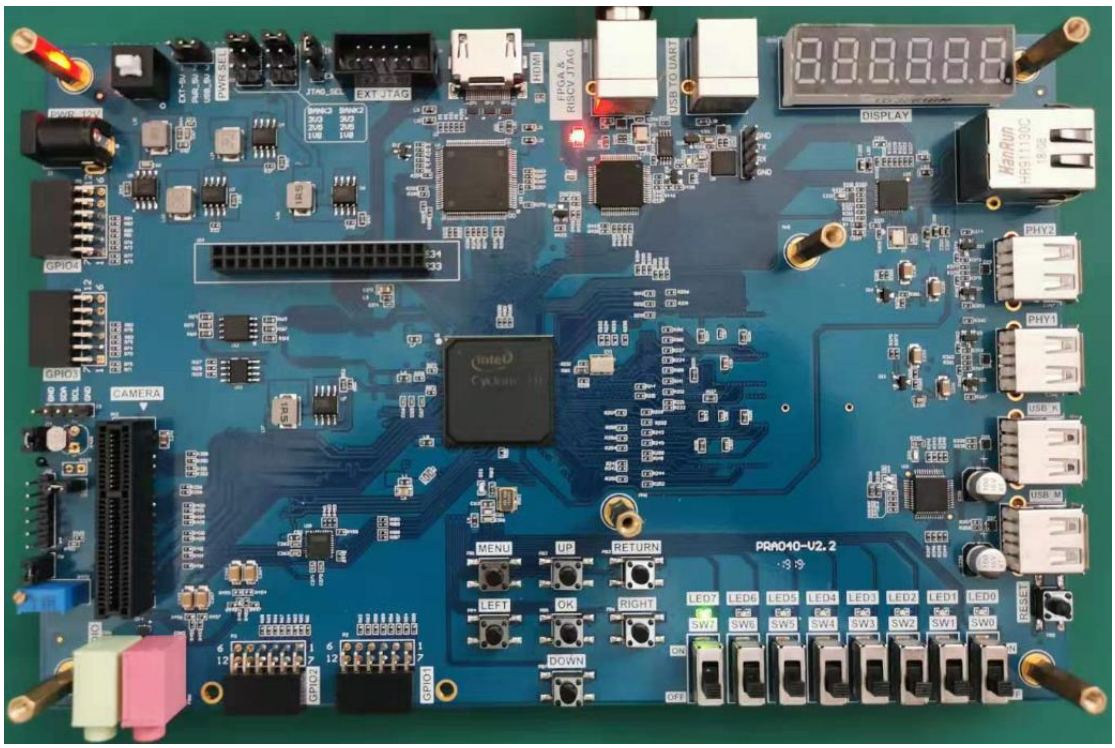


Figure 5.7 Experiment result(one right shift)

Press the right shift button again and the LED will move one bit to the right. See Figure 5.8.

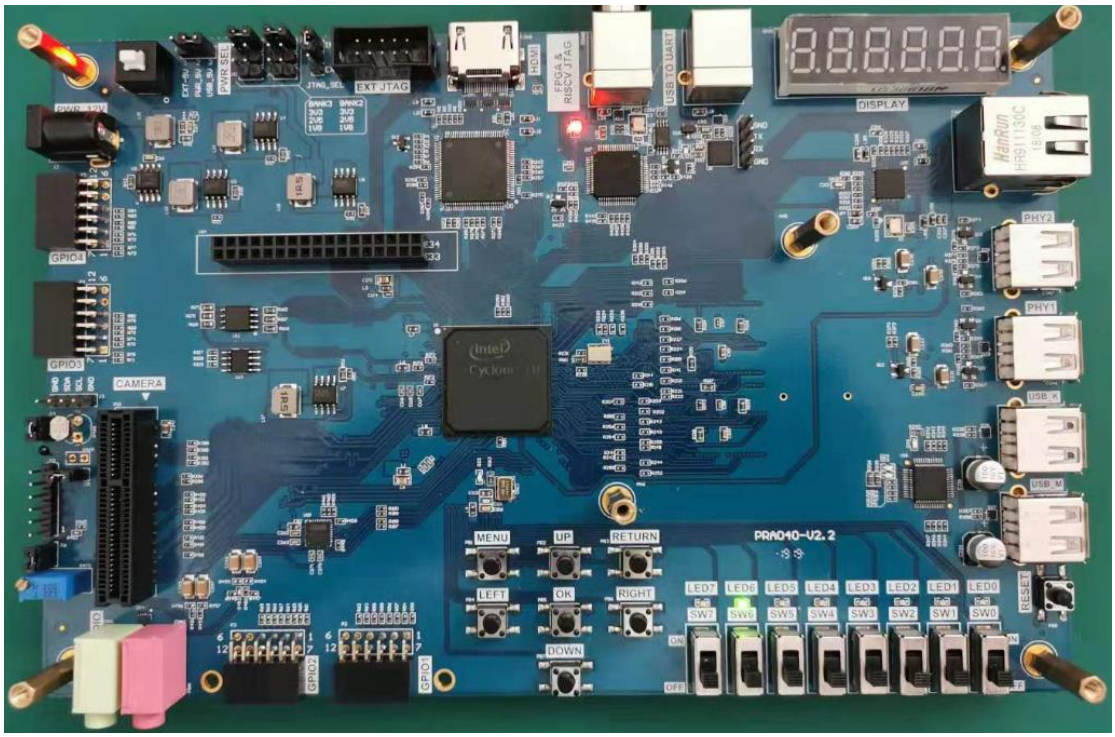


Figure 5.8 Experiment result(another right shift)

Experiment 6 Use of Multipliers and ModelSim

6.1 Experiment Objective

（1） Learn to use multiplier

（2） Use ModelSim to simulate design output

6.2 Experiment Implement

（1） 8x8 multiplier, the first input value is an 8-bit switch, and the second input value is the output of an 8-bit counter.

（2） Observe the output in ModelSim

（3） Observe the calculation results with a four-digit segment display

6.3 Experiment

Since learning uses of the simulation tools and the new IP core, there is no introduction and hardware design part.

6.3.1 Introduction of Program

ModelSim is an HDL language simulation software. Our program can be simulated to achieve inspection and error correction. ModelSim, different from the previous experiment, when building the project, you need to add the simulation tool to be used in the EDA tool selection window. See Figure 6.1.

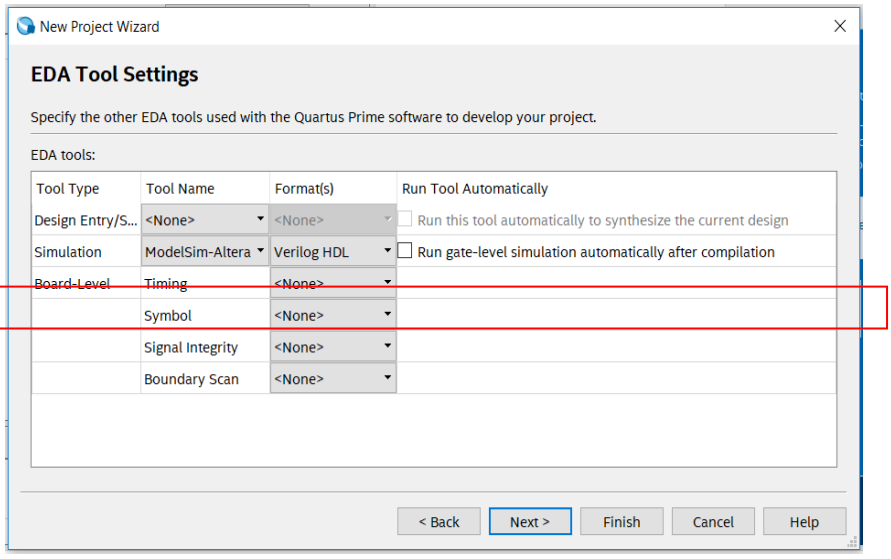


Figure 6.1 EDA tool setting

Only one counter, one PLL and one multiplier are used in the program. Only the multiplier is introduced here.

The first step: the establishment of the main program framework

```
module mult_sim (
    input           inclk,
    input           rst,
    input   [7:0]   sw,
    output  [15:0]  mult_res,
    output  reg [7:0]   count
);
```

The value of the switch is used as the first input of the multiplier, the value of the counter is the second input, and the result of the calculation is output.

Step 2: The multiplier IP core setting steps are as follows:

（1） After adding the LPM_MULT IP (**IP Catalog -> Library -> Basic Functions -> Arithmatic -> LPM_MULT**) and saving the path, the setting window of the multiplier is popped up, as shown in Figure 6.2, and the two input data is set to eight bits as required.

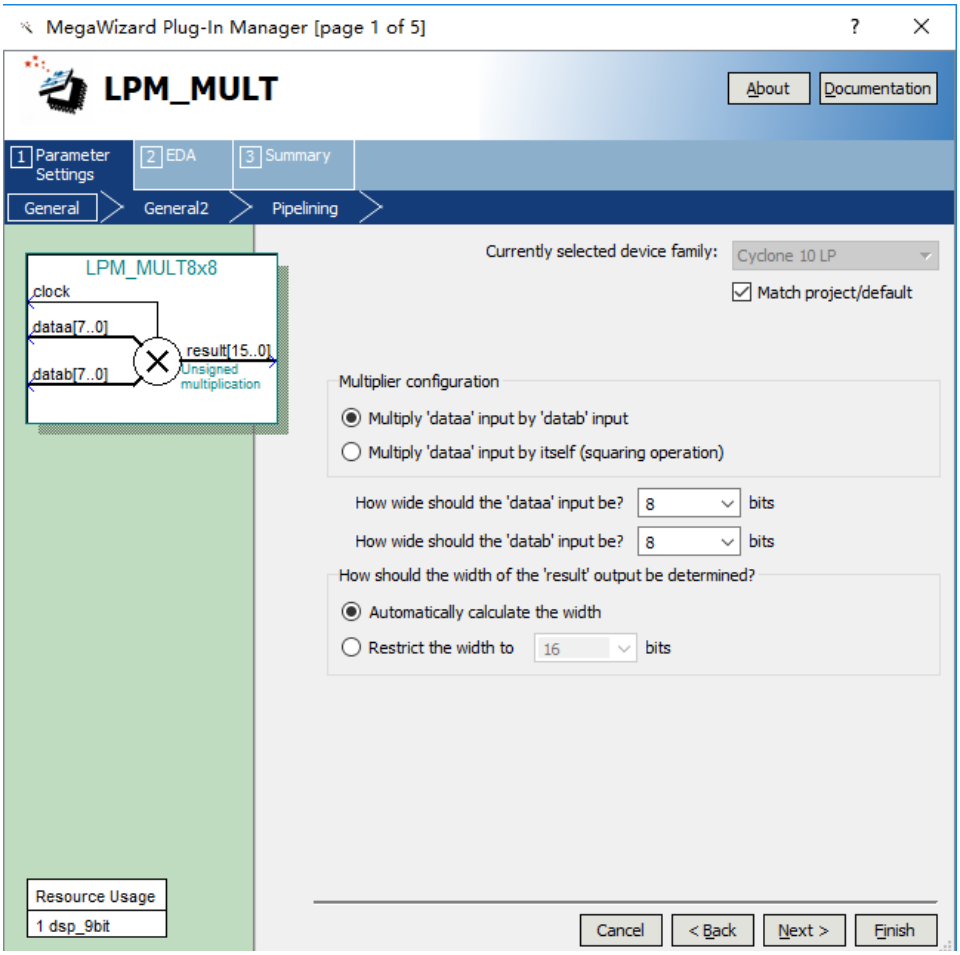


Figure 6.2 mult setting 1

（2） Select the multiplication type to be **Unsigned**. See Figure 6.3.

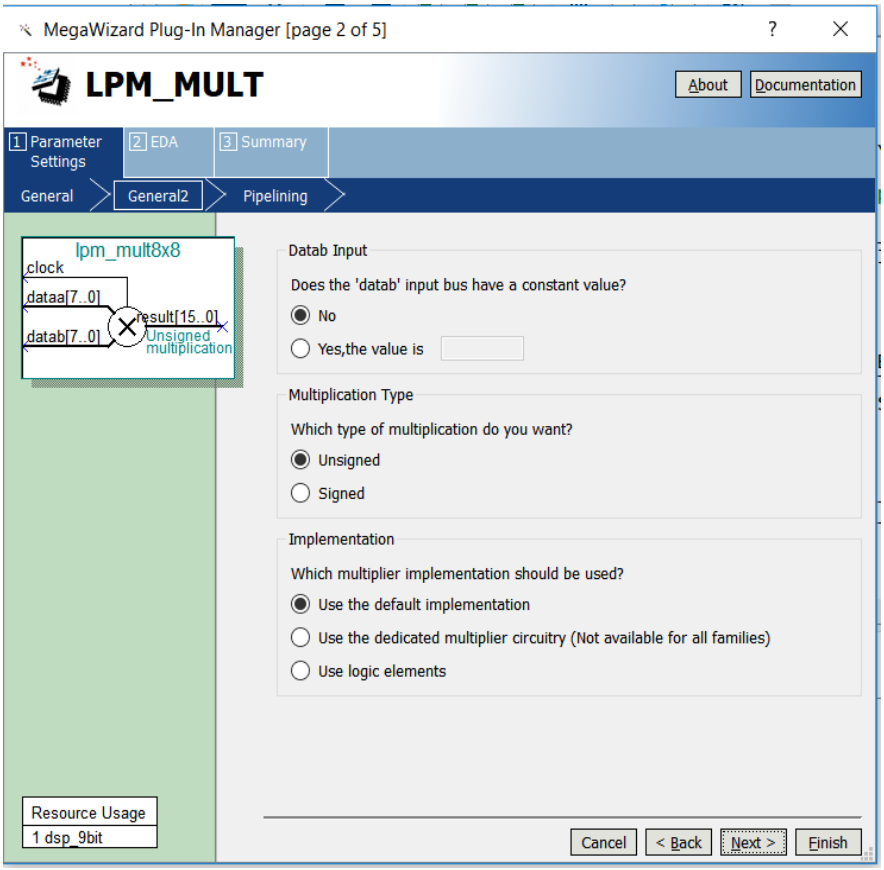


Figure 6.3 mult setting 2

（3） Select the pipeline to speed up the operation, as shown in Figure 6.4.

（4） Select others to be default

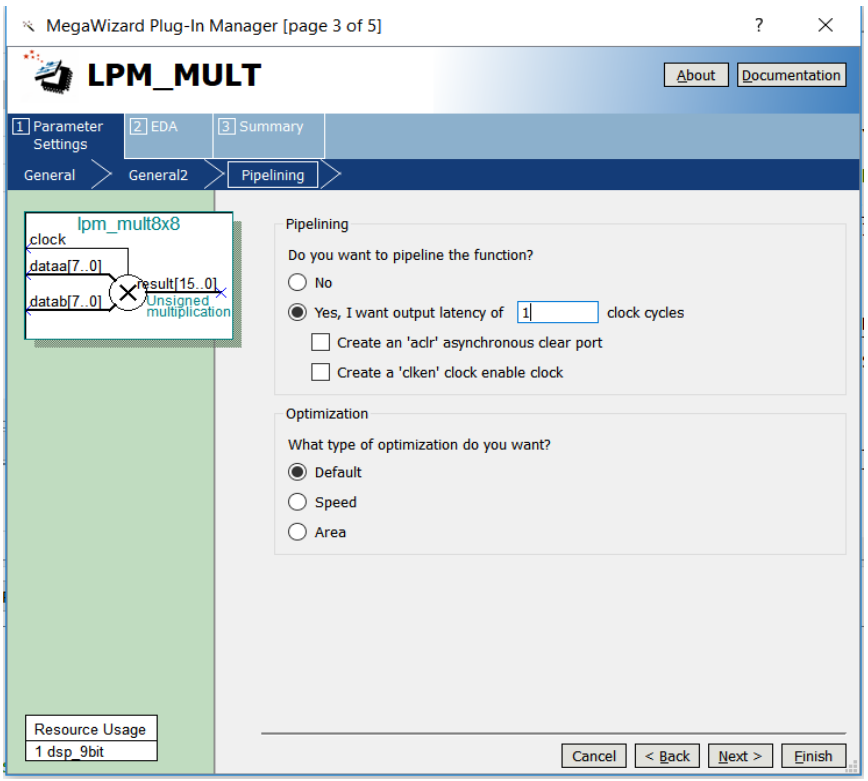


Figure 6.4 mult setting 3

8x8 multiplier instantiation:

```
reg             sys_clk;
mult_8x8 mult_8x8_inst (
    .clock          (sys_clk),
    .dataa          (sw),
    .datab          (count),
    .result         (mult_res)
);
```

6.4 Use of ModelSim and the Experiment Verification

Here, use ModelSim to simulate verifing the experiment.

Method 1: Simulation based on waveform input

（1） Click the menu bar **Tools -> Options**, as shown in Figure 6.5, click **OK**.

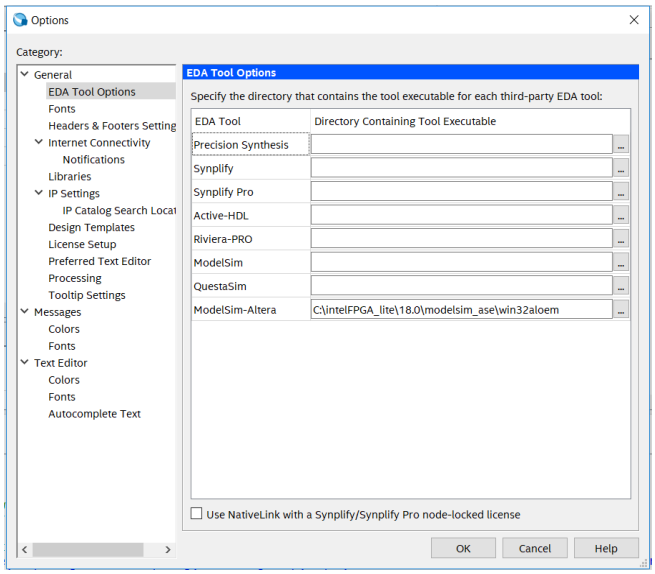


Figure 6.5 Set Modelsim-Altera path

（2） **Tool -> Run Simulation Tool -> RTL Simulation**. See Figure 6.6.

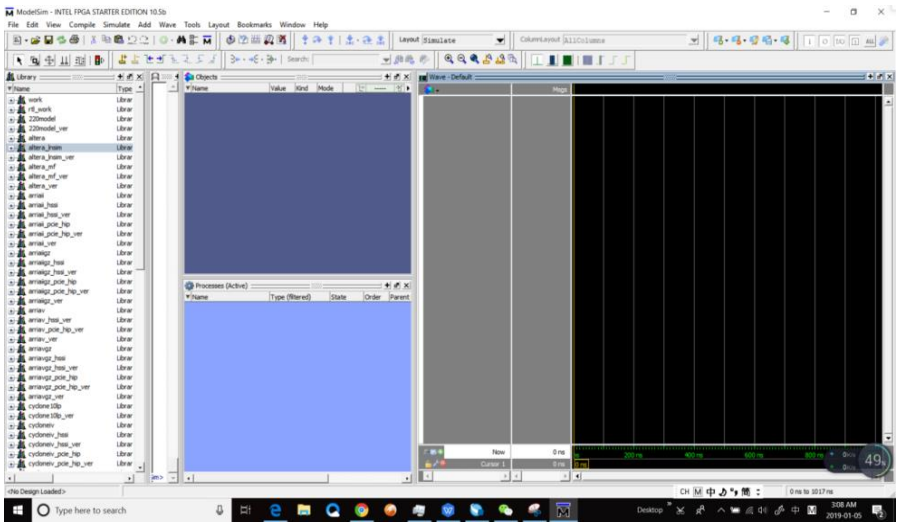


Figure 6.6 ModelSim interface

（3） Set ModelSim

1) Simulate -> Start Simulation

2) In the popup window, add libraries under Libraries tag. See Figure 6.7.

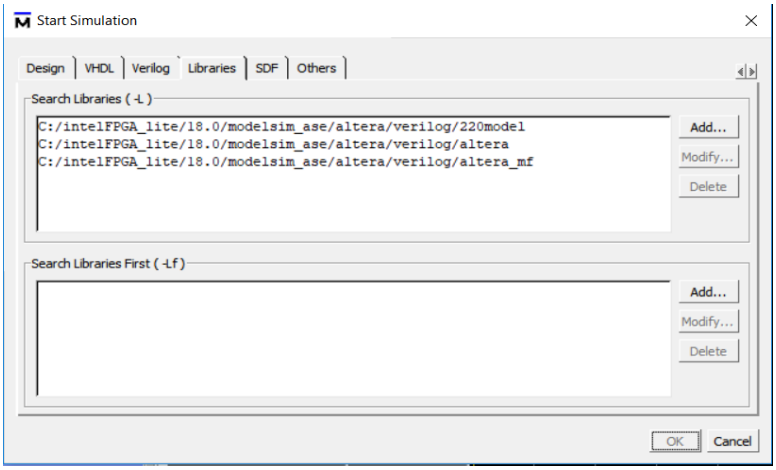


Figure 6.7 Add simulation libraries

3) Under **Design** tag, choose simulation project *mult_sim* and click **OK**. See Figure 6.8.

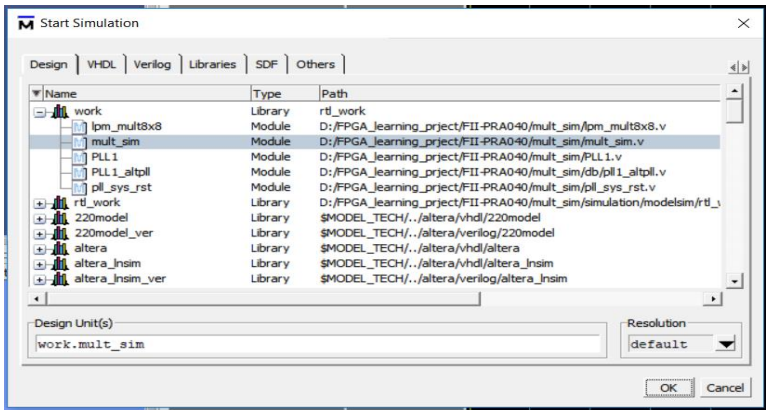


Figure 6.8 Choose the project in simulation

4) In the **Objects** window, choose all the signals and drag them to **Wave** window. See Figure 6.9.

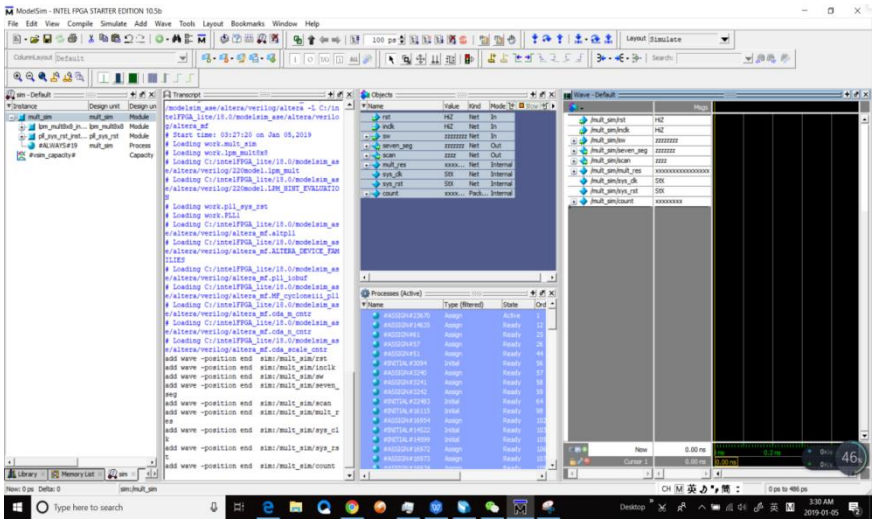


Figure 6.9 Add observation signals

5) Set the signals in **Wave**, right click any signal and a selection window will occur. See Figure 6.10.

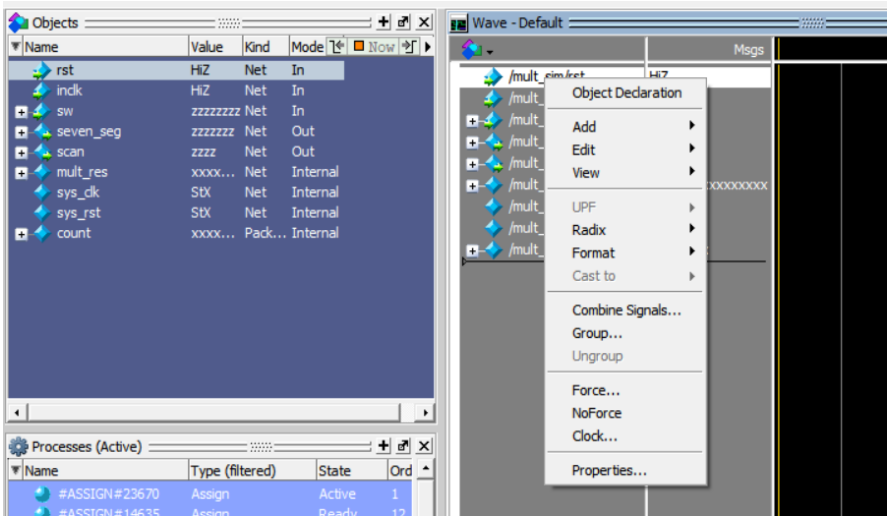


Figure 6.10 Set the signals

6) logical signals select **Force** and select **Clock** for clock signals

A. Set *rst* signal. See Figure 6.11.

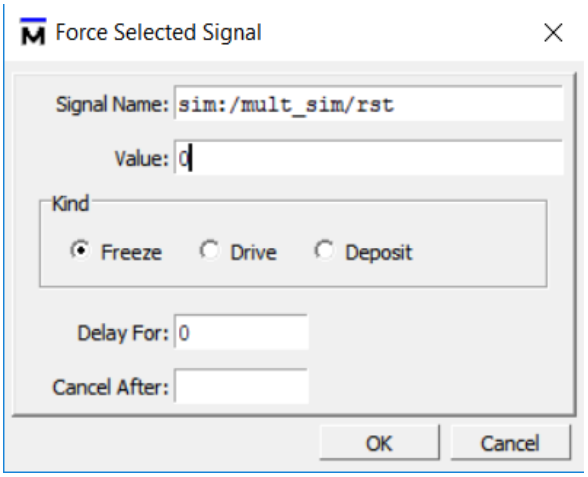


Figure 6.11 Set *rst* signal

B. Set *Inclk* signal. See Figure 6.12.

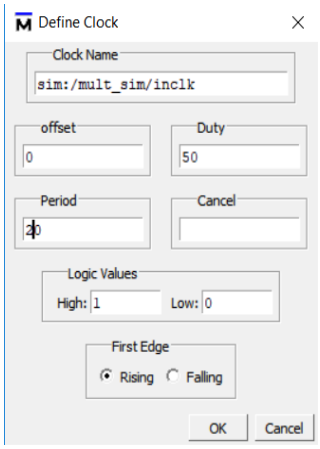


Figure 6.12 Set *inclk* signal

C. Set *sw* signal. See Figure 6.13.

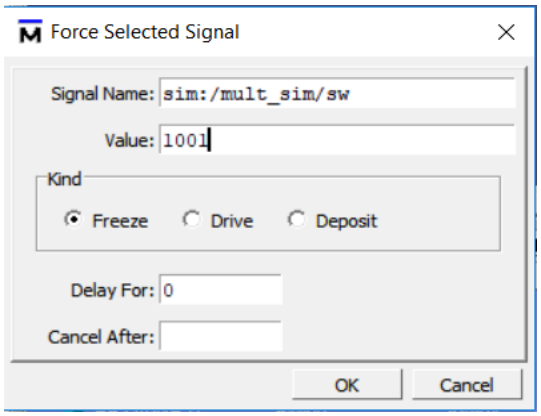


Figure 6.13 Set sw signal

7) Run simulation. In the tool bar, set the simulation time to be **100 ns**. Click the **Run** icon to run. See Figure 6.14.

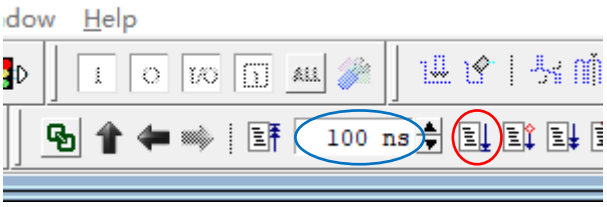


Figure 6.14 Set the simulation time

8) Observe the simulation result. See Figure 6.15.

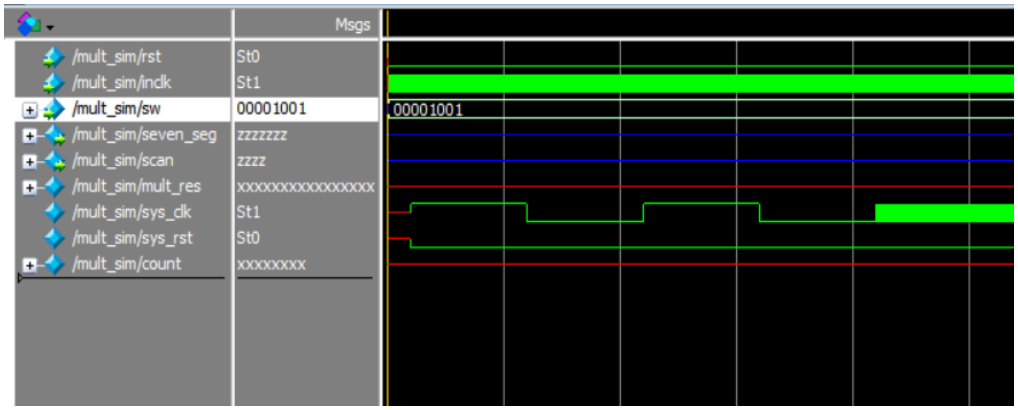


Figure 6.15 Simulation result

9) Result analysis
   A. Counter count does not have a valid result, instead, unknow result XXXXXX is gotten.
   B. *sys_rst* does not reset signals. It changes from X to 0
   C. Add *pll_locked* signal to the wave, and re-simulate
   D. In Figure 6.16, before PLL starts to lock, the *sys_clk* already has a rising edge, so *PLL_locked* signal is just converted from low to high. There is no reliable reset is formed.

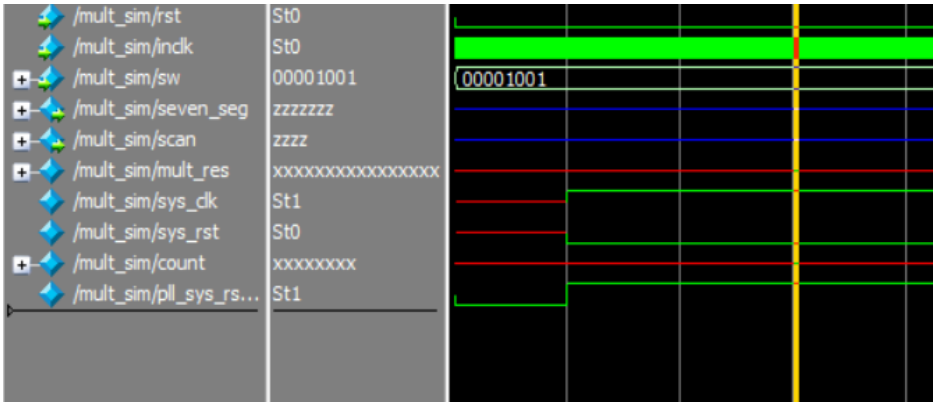


Figure 6.16 Re-simulation result

E. Solution

a. Define *sys_rst* to be 1'b0

b. Use external rst signal to procide reset

Here method **a** is adopted. The revised code is as follows：

```
module pll_sys_rst(
    input           inclk,
    output          sys_clk,
    output  reg     sys_rst = 1'b1
);
```

10) Recompile the simulation.

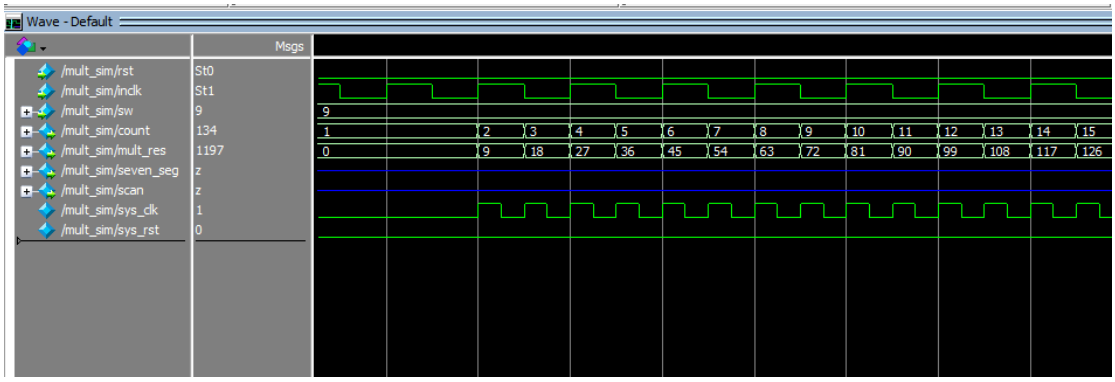


Figure 6.17 Recompile the simulation

Since waveform editing efficiency is relatively low, the use of simulation testbench file is encouraged.

Method 2: Write a testbench file for simulation

（1） Name a new Verilog HDL file *tb_mult.v*.

（2） The code is as follows:

```
`timescale 1ns/1ps
module tb_mult;
    reg             rst;
    reg             clk;
    reg     [7:0]   sw;
```

```
    wire    [7:0]    count;
    wire    [15:0]   mult_res;

// S1 is the instance of simulation module
mult_sim S1(
    .rst             (rst),
    .inclk           (clk),
    .sw              (sw),
    .count           (count),
    .mult_res        (mult_res)
);
// Define the clock required for the simulation and display the results in text form
always begin
    #10 clk = ~clk;
    $monitor ("%d * %d = %d", count, sw, mult_res);
end
//Set the simulation
initial begin
    rst = 0;
    clk = 1;

    #10 sw = 20;
    #10 sw = 50;
    #10 sw = 100;
    #10 sw = 101;
    #10 sw = 102;
    #10 sw = 103;
    #10 sw = 104;
    #50 sw = 105;
    //stop signal
    #1000 $stop;
end
endmodule
```

When writing the testbench file, first mark the time unit of the simulation at the beginning, this experiment is 1 ns, then instantiate the project that needs to be simulated into the testbench file, define the clock cycle and the simulation conditions, and stop the simulation after a certain time. This simulation stops after 1000 clock cycles.

After the compilation, the testbench file is added to the ModelSim for simulation, the specific steps are as follows:

A. Set the testbench file: **Assignments -> Settings**. See Figure 6.18.

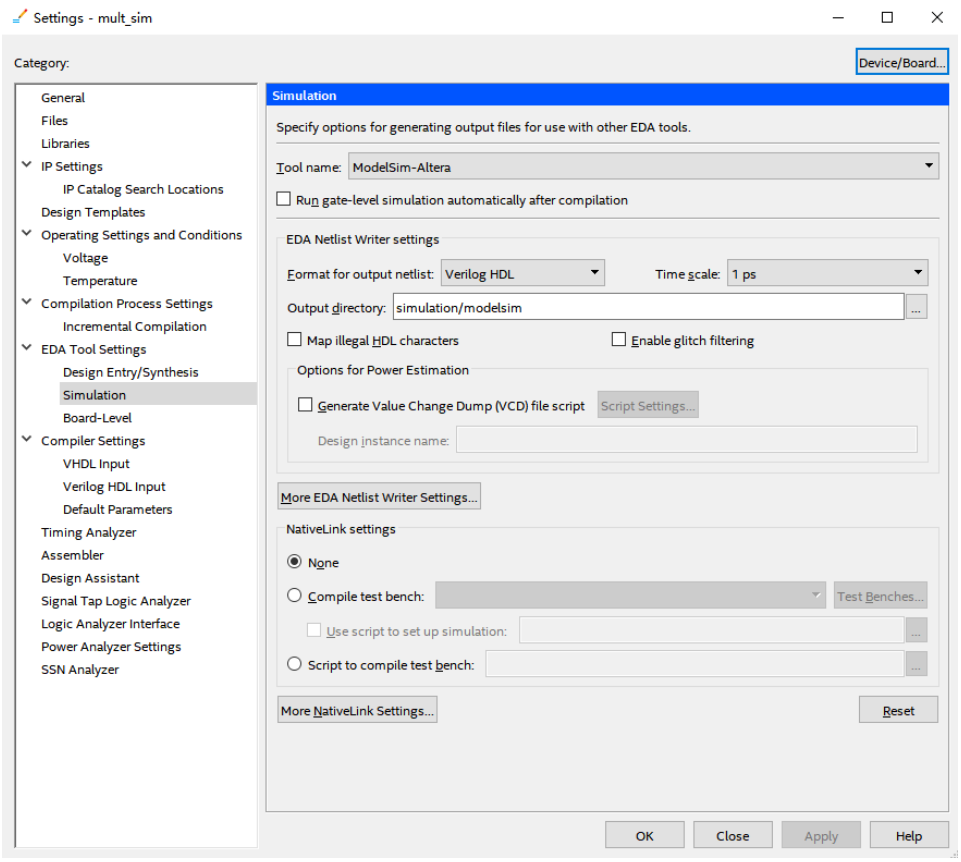


Figure 6.18 Simulation setting 1

B. In **Compile test bench**, click **Test Benches** to add tb simulation file. See Figure 6.19.

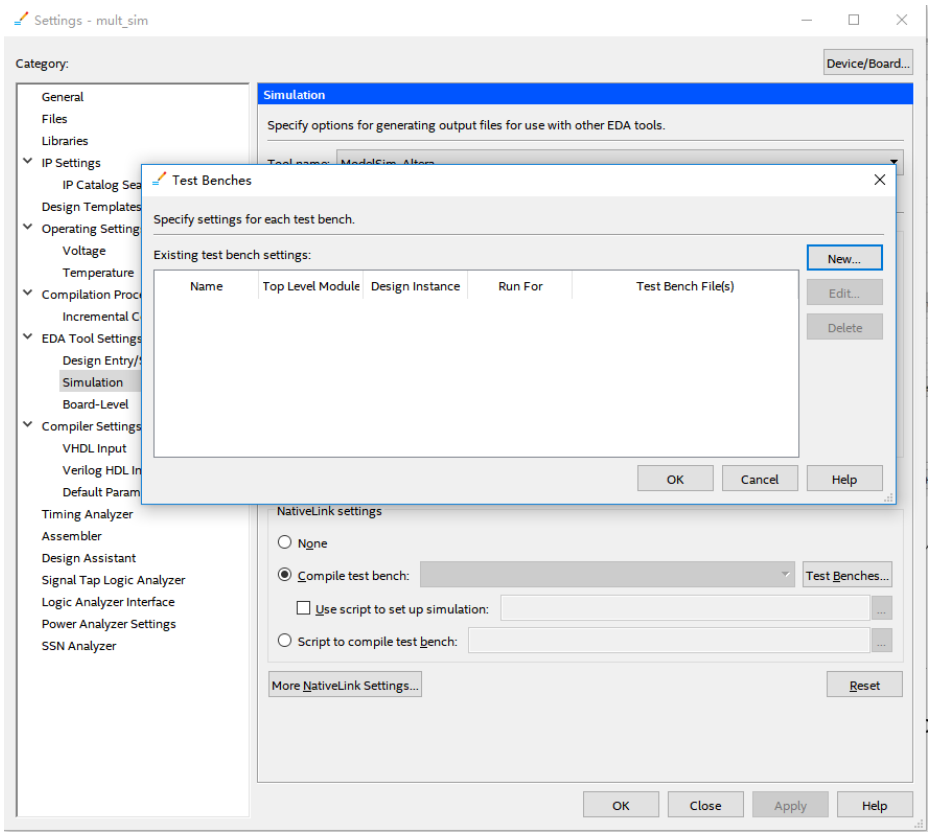


Figure 6.19 Simulation setting 2

C. Click **New,** input the **Test bench name**. Make the name be consistent with tb file. See Figure 6. 20.

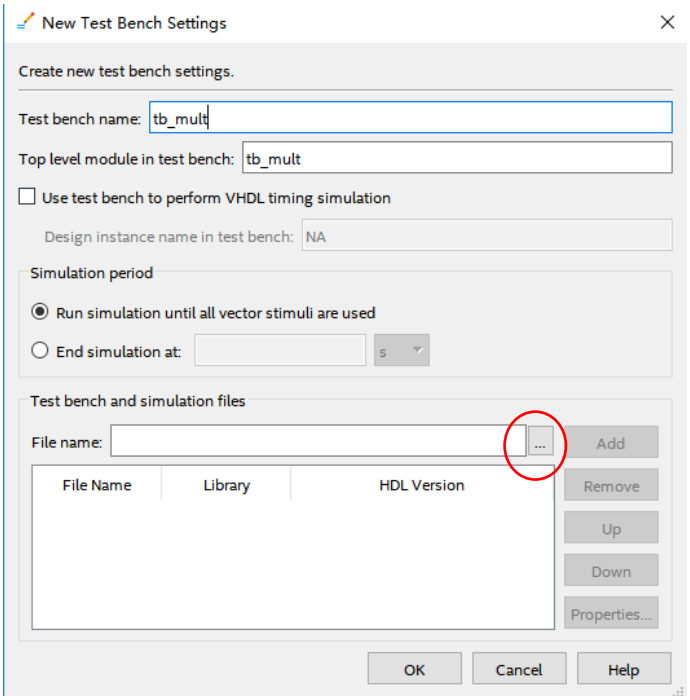


Figure 6.20 Simulation setting 3

D. the red ellipse to add the test bench file. Find *tb_mult.v* file written before.

E. Click **Add** to add. Click **OK** (three times) to finish the setting. See Figure 6.21.

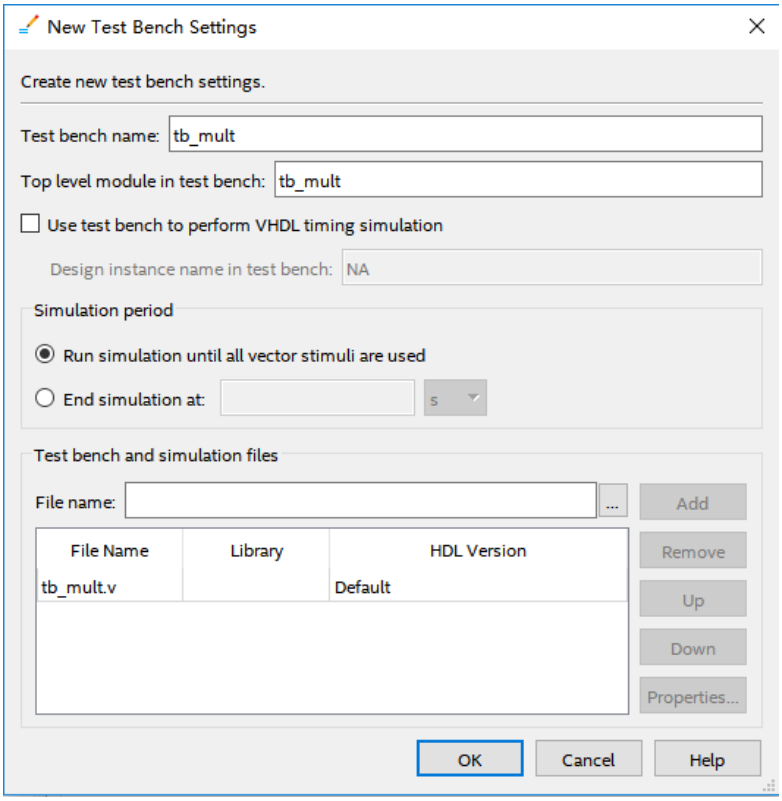


Figure 6.21 Simulation setting 4

（3） Repeat previous step, to start ModelSim to simulate. See Figure 6.22.

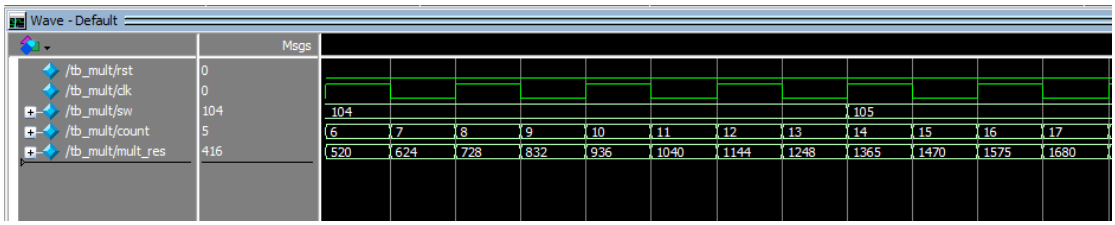


Figure 6.22 Waveform output

After a certain delay, outputs will display in Transcript. See Figure 6.23

Because the result of the operation will be one clock cycle later than the input, the multiplier and the result will differ by one line, which does not seem to match, but does not affect the analysis of the experimental results.

```
#    1 * 102 =     0
#    2 * 103 =   103
#    3 * 103 =   206
#    4 * 104 =   312
#    5 * 104 =   416
#    6 * 104 =   520
#    7 * 104 =   624
#    8 * 104 =   728
#    9 * 104 =   832
```

Figure 6.23 Text displays operation result

Summary and Reflection

Try to use the switch as the input to the multiplier. The upper four digits are one number, the lower fourth digit is a number, and the two numbers are multiplied to output the result.

Experiment 7 Hexadecimal Number to BCD Code Conversion and Application

7.1 Experiment Objective

（1） Learn to convert binary numbers to BCD code (bin_to_bcd)
（2） Learn to convert hexadecimal numbers to BCD code (hex_to_bcd)

7.2 Experimental Implement

Combined with experiment 6, display the results of the operation to the segment display.

7.3 Experiment

7.2.1 Introduction to the principle of hexadecimal number to BCD code

Since the hexadecimal display is not intuitive, decimal display is more widely used in real life.

Human eyes recognition is relatively slow, so the display from hexadecimal to decimal does not need to be too fast. Generally, there are two methods

（1） Countdown method:

Under the control of the synchronous clock, the hexadecimal number is decremented by 1 until it is reduced to 0. At the same time, the appropriate BCD code decimal counter is designed to increment. When the hexadecimal number is reduced to 0, the BCD counter Just gets with the same value to display.

（2） Bitwise operations (specifically, shift bits and plus 3 here). The implementation is as follows:

1) Set the maximum decimal value of the expression. Suppose you want to convert the 16-digit binary value (4-digit hexadecimal) to decimal. The maximum value can be expressed as 65535. First define five four-digit binary units: ten thousand, thousand, hundred, ten, and one to accommodate calculation results
2) Shift the hexadecimal number by one to the left, and put the removed part into the defined variable, and judge whether the units of ten thousand, thousand, hundred, ten, and one are greater than or equal to 5, and if so, add the corresponding bit to 3 until the 16-bit shift is completed, and the corresponding result is obtained.
   Note: Do not add 3 when moving to the last digit, put the operation result directly
3) The principle of hexadecimal number to BCD number conversion
   Suppose ABCD is a 4-digit binary number (possibly ones, 10 or 100 bits, etc.), adjusts it to BCD code. Since the entire calculation is implemented in successive shifts, ABCDE is obtained after shifting one bit (E is from low displacement and its value is

either 0 or 1). At this time, it should be judged whether the value is greater than or equal to 10. If so, the value is increased by 6 to adjust it to within 10, and the carry is shifted to the upper 4-bit BCD code. Here, the pre-movement adjustment is used to first determine whether ABCD is greater than or equal to 5 (half of 10), and if it is greater than 5, add 3 (half of 6) and then shift.

For example, ABCD = 0110 (decimal 6)

A. After shifting it becomes 1100 (12), greater than 1001 (decimal 9)
B. By plus 0110 (decimal 6), ABCD = 0010, carry position is 1, the result is expressed as decimal 12
C. Use pre-shift processing, ABCD = 0110 (6), greater than 5, plus 3
D. ABCD = 1001 (9), shift left by one
E. ABCD = 0010, the shifted bit is the lowest bit of the high four-bit BCD.
F. Since the shifted bit is 1, ABCD = 0010(2), the result is also 12 in decimal.
G. The two results are the same
H. Firstly, make a judgement, and then add 3 and shift. If there are multiple BCD codes at the same time, then multiple BCD numbers all must first determine whether need to add 2 and then shift.

（3） The first way is relatively easy. Here, the second method is mainly introduced.

Example 1: Binary to BCD. See Figure 7.1.

100's	10's	1's	Binary	Operation
			1010 0010	
		1	010 0010	<< #1
		10	10 0010	<< #2
		101	0 0010	<< #3
		1000		add 3
	1	0000	0010	<< #4
	10	0000	010	<< #5
	100	0000	10	<< #6
	1000	0001	0	<< #7
	1011			add 3
1	0110	0010		<< #8

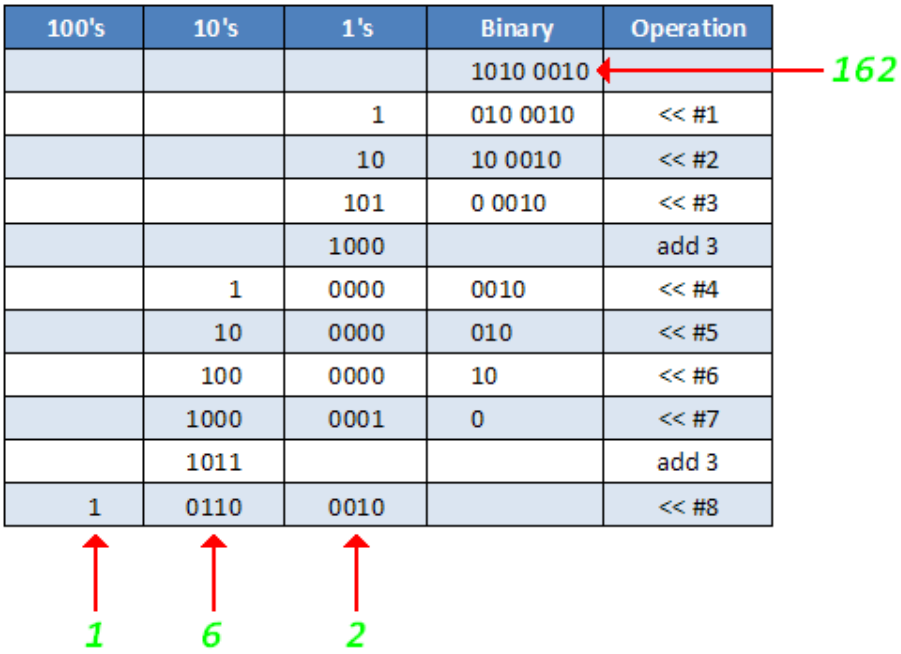


Figure 7.1 Example 1, bin_to_bcd

Example 2: Hexadecimal to BCD. See Figure 7.2.

Operation	Hundreds	Tens	Units	Binary	
HEX				F	F
Start				1 1 1 1	1 1 1 1
Shift 1			1	1 1 1 1	1 1 1
Shift 2			1 1	1 1 1 1	1 1
Shift 3			1 1 1	1 1 1 1	1
Add 3			1 0 1 0	1 1 1 1	1
Shift 4		1	0 1 0 1	1 1 1 1	
Add 3		1	1 0 0 0	1 1 1 1	
Shift 5		1 1	0 0 0 1	1 1 1	
Shift 6		1 1 0	0 0 1 1	1 1	
Add 3		1 0 0 1	0 0 1 1	1 1	
Shift 7	1	0 0 1 0	0 1 1 1	1	
Add 3	1	0 0 1 0	1 0 1 0	1	
Shift 8	1 0	0 1 0 1	0 1 0 1		
BCD	2	5	5		

Figure 7.2 hex_to_bcd

7.2.2 Introduction to the Program

The first step: the establishment of the main program framework

```
module HEX_BCD (

    input       [15:0] hex,
    output  reg [3:0]  ones,
    output  reg [3:0]  tens,
    output  reg [3:0]  hundreds,
    output  reg [3:0]  thousands,
    output  reg [3:0]  ten_thousands
);
```

Enter a 16-bit binary number *hex*, which can represent a maximum of 65535 decimal, so output *ones*, *tens*, *hundreds*, *thousands*, and *ten_thousands*.

The second step: the implementation of bit operation

```
reg       [15:0]    hex_reg;
integer             i;
always @ (*)
begin
    hex_reg = hex;
    ones = 0;
    tens = 0;
    hundreds = 0;
    thousands = 0;
    ten_thousands = 0;
```

```
    for (i = 15; i >= 0; i = i-1) begin
        if(ten_thousands >= 5)
        ten_thousands = ten_thousands + 3;
        if(thousands >= 5)
            thousands = thousands + 3;
        if(hundreds >= 5)
            hundreds = hundreds + 3;
        if(tens >= 5)
            tens = tens + 3;
        if(ones >= 5)
            ones = ones + 3;

        ten_thousands = ten_thousands << 1;
        ten_thousands[0] = thousands[3];
        thousands = thousands << 1;
        thousands[0] = hundreds[3];
        hundreds = hundreds << 1;
        hundreds[0] = tens[3];
        tens = tens << 1;
        tens[0]= ones[3];
        ones = ones << 1;
        ones[0] = hex_reg[15];
        hex_reg = {hex_reg[14:0], 1'b0};
    end
end
```

Referring to Figure 7.2, the former part of the program is the judgment calculation part, and if it is greater than or equal to 5, then adds 3. The latter part is the shift part.

The third step: verification

Referring to Experiment 6, simulation was performed using ModelSim, and the simulation conditions were set as follows:

```
initial   begin
    hex = 0 ;
    repeat (20) begin
    #10;
    hex = {$random}%20000;
    #10;
    end
end
```

At the beginning, the 16-bit binary number is equal to 0. The delay is 10 ns. The 16-bit binary number takes a random number less than 20,000. A delay of 10 ns is applied and the process is repeated 20 times.

After the ModelSim is set and the testbench file is added, perform the simulation. The result is shown in Figure 7.3.

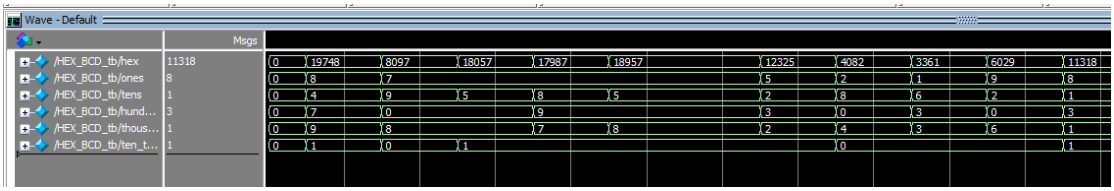


Figure 7.3 Simulation for binary to decimal

Remark and reflection:

A. The assignment marks for the examples above are "=" instead of "<=". Why?

B. Since the whole program is designed to be combinational logic, when invoking the modules, the other modules should be synchronized the timing.

7.4 Application of Hexadecimal Number to BCD Number Conversion

（1） Continue to complete the multiplier of Experiment 6 and display the result in segment display in decimal. Every 1 second, the calculation result on the segment display changes once. The experiment needs to use frequency division, segment display, multiplier and hexadecimal number to BCD code.

（2） Compilation. Observe the **Timing Analyzer** in **Compilation Report**.

1) Fmax Summary 83.71 MHz. See Figure 7.4.

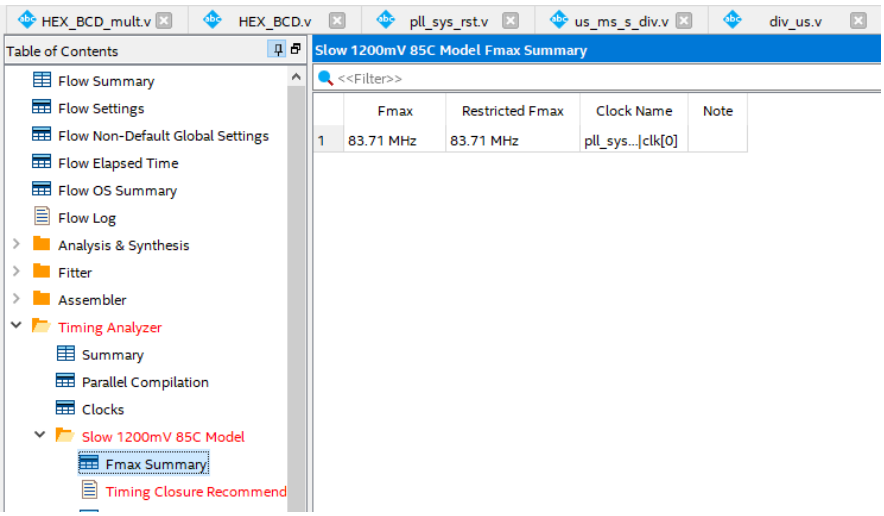


Figure 7.4 Fmax Summary

2) Setup Memory

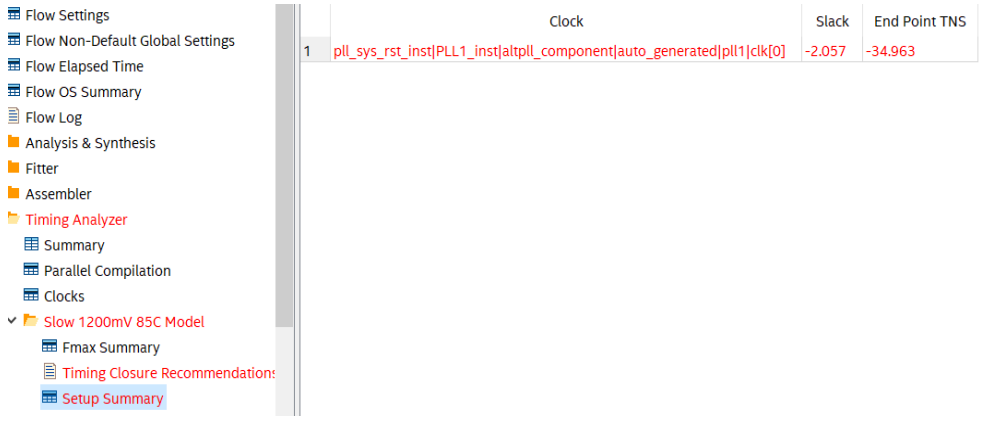


Figure 7.5 Setup time summary

3) Timing Closure Recommendation. See Figure 7.6.

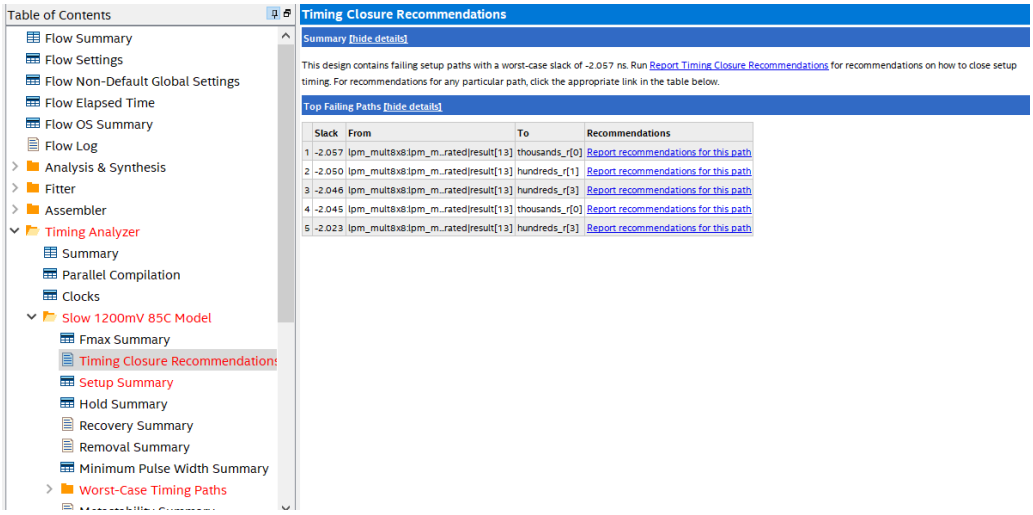


Figure 7.6 Timing Analysis

4) From the above three indicators, the above programming does not meet the timing requirements. It can also be seen that the maximum delay path is the delay of the output of the multiplier to *HEX_BCD*.

There are 3 solutions:

A. Reduce the clock frequency

B. Increase the timing of *HEX_BCD* and increase the pipeline

C. Insert pipeline isolation at the periphery (can reduce some delay)

The way to increase the pipeline, will be introduced in the follow-up experiment, because the function of *HEX_BCD* is mainly used to display the human-machine interface, the speed requirement is low, and the frequency reduction method is adopted here.

（3） Modify PLL to increase an output of 20 MHz frequency (BCD_clk)..

（4） Recompile and observe timing results

（5） Lock the pins, compile, and download the programto FII-PRA040 development board for testing

7.5 Experiment Verification

The first step: pin assignment

Table 7.1 Hexadecimal number to BCD pin mapping

Signal Name	Network Label	FPGA Pin	Description
clk	CLK_50M	G21	Input clock
rst_n	PB3	Y6	Reset
scan[0]	SEG_3V3_D5	F14	Bit selection 0
scan[1]	SEG_3V3_D4	D19	Bit selection 1
scan[2]	SEG_3V3_D3	E15	Bit selection 2
scan[3]	SEG_3V3_D2	E13	Bit selection 3
scan[4]	SEG_3V3_D1	F11	Bit selection 4
scan[5]	SEG_3V3_D0	E12	Bit selection 5
seven_seg[0]	SEG_PA	B15	Segment a
seven_seg[1]	SEG_PB	E14	Segment b
seven_seg[2]	SEG_PC	D15	Segment c
seven_seg[3]	SEG_PD	C15	Segment d
seven_seg[4]	SEG_PE	F13	Segment e
seven_seg[5]	SEG_PF	E11	Segment f
seven_seg[6]	SEG_PG	B16	Segment g
seven_seg[7]	SEG_DP	A16	Segment h
SW[7]	PB7	W6	Swicth 7
SW[6]	PB6	Y8	Swicth 6
SW[5]	PB5	W8	Swicth 5
SW[4]	PB4	V9	Swicth 4
SW[3]	PB3	V10	Swicth 3
SW[2]	PB2	U10	Swicth 2
SW[1]	PB1	V11	Swicth 1
SW[0]	PB0	U11	Swicth 0

Step 2: board downloading verification

After the pin assignment is completed, the compilation is performed. Downloading program to the board is verified after passing. The experimental result is shown in Figure 7.7. The value of the DIP switch input is 00001010, the decimal is 10, the counter is constantly accumulating, so the display result is always accumulatively changed by 10.

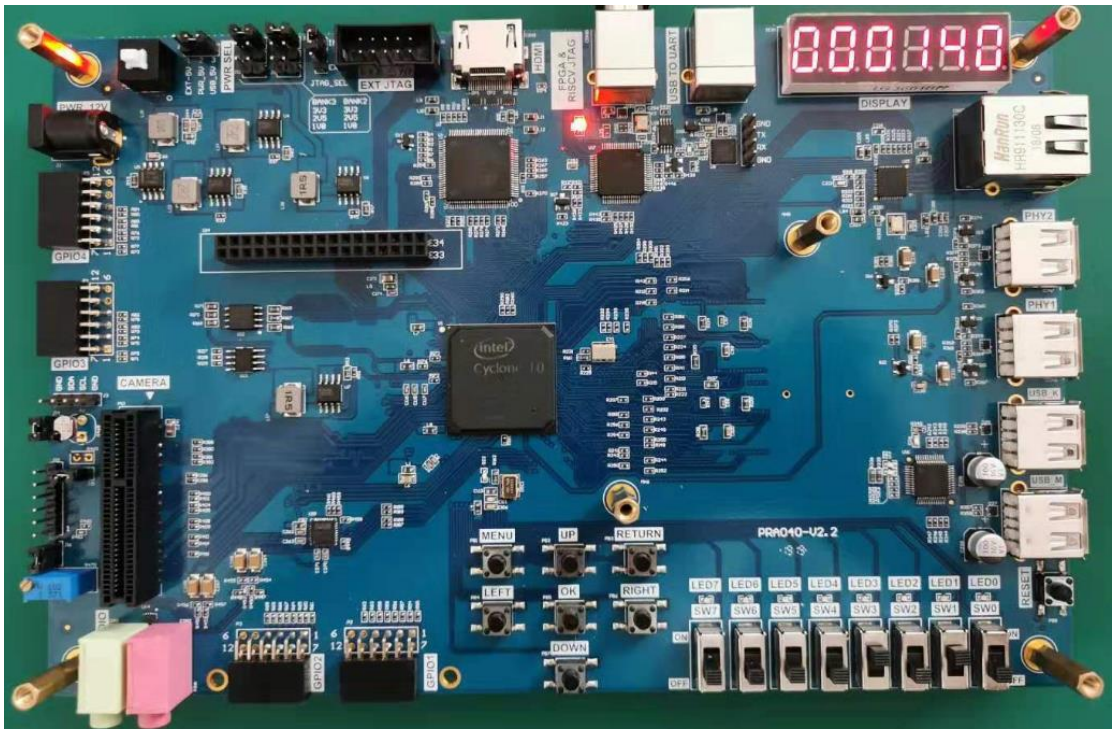


Figure 7.7 Experiment result

Experiment Summary and Reflection

（1） How to implement BCD using more than 16bits binary numbers

（2） What is a synchronous clock and how to handle an asynchronous clock

（3） Learn to design circuits meeting the requirement

Experiment 8 Use of ROM

8.1 Experiment Objective

（1） Study the internal memory block of FPGA

（2） Study the format of *.mif and how to edit *.mif file to configure the contents of ROM

（3） Learn to use RAM, read and write RAM

8.2 Experiment Implement

（1） Design 16 outputs ROM, address ranging 0-255

（2） Interface 8-bit switch input as ROM’s address

（3） Segment display the contents of ROM and require conversion of hexadecimal to BCD output.

8.3 Experiment

8.3.1 Introduction of the Program

This experiment was carried out on the basis of Experiment 7, and the contents of Experiment 7 were cited, so only the IP core ROM portion was introduced.

（1） In **Installed IP**, choose **Library -> Basic Function -> On Chip Memory -> ROM: 1-PORT**, file type to be Verilog HDL. Choose **16** bits and **256** words for output. See Figure 8.1.

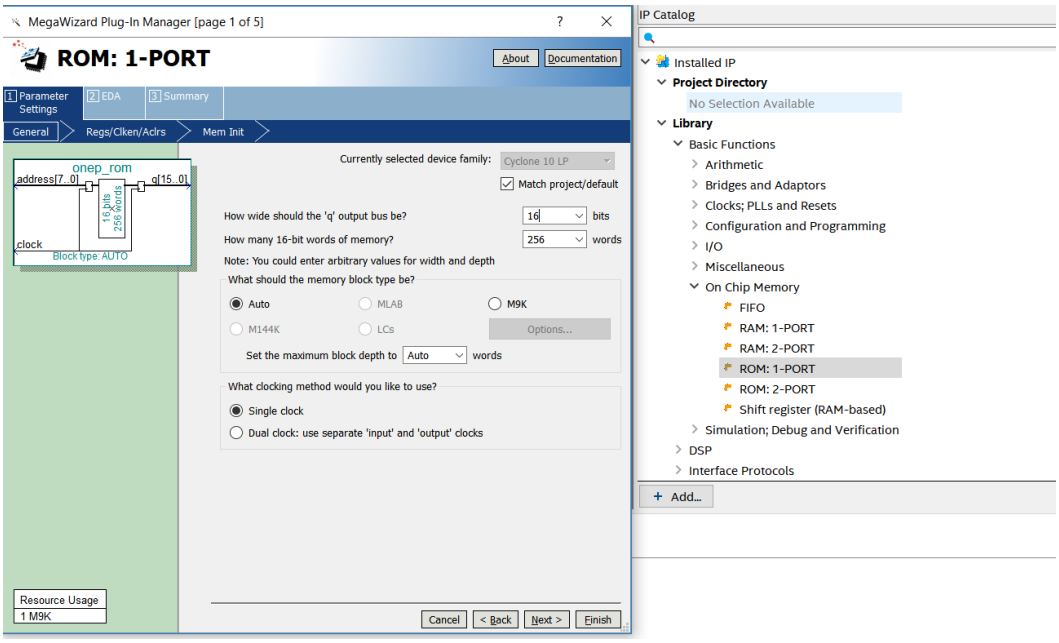


Figure 8.1 RAM IP core invoking

（2） According to the default setting, you need to add an initial ROM file in the location where red oval circles. See Figure 8.2. In the figure, a *.mif file has already been added.

（3） Create a top level entity *rom.mif*

1) Go to **File -> New -> Memory Files -> Memory Initialization File**. See Figure 8.3.
2) In Figure 8.4, modify the **Number of words** and **Word size**.
3) In Figure 8.5. In the address area, right click and you can input the data or change the display format, such as hexadecimal, octal, binary, unsigned, signed, etc.

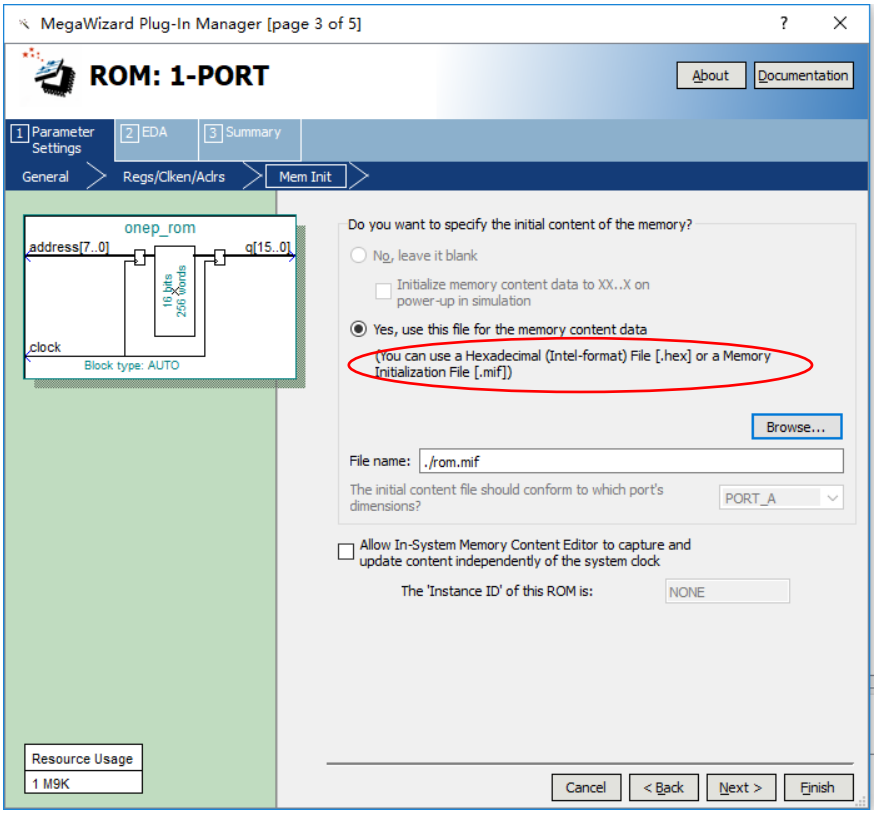


Figure 8.2 ROM setting

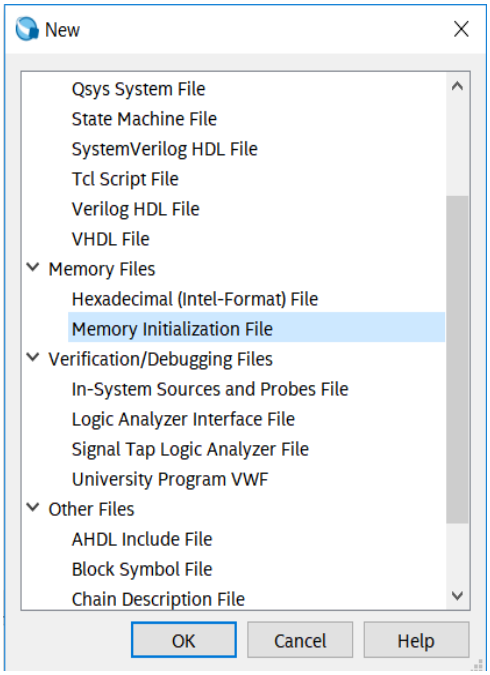


Figure 8.3 New *.mif file

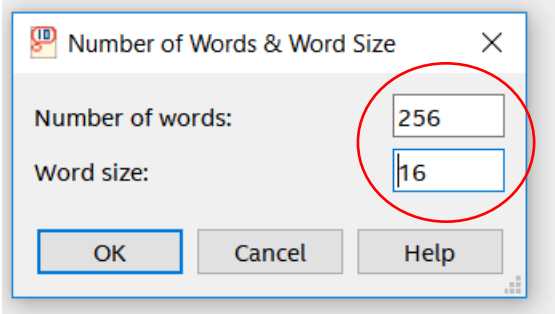


Figure 8.4 *.mif file setting 1

Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	0000	0900	FFFF	00FF	0200	0044	0027	04F6
008	0000	0000	0000	0000	0000	0000	0000	0000
010	0000	0000	0000	0000	0000	0000	0000	0000
018	0000	0000	0000	0000	0000	0000	0000	0000
020	0000	0000	0000	0000	0000	0000	0000	0000
028	0000	0000	0000	0000	0000	0000	0000	0000
030	0000	0000	0000	0000	0000	0000	0000	0000
038	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000
048	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000
058	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000
068	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000
078	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000

Figure 8.5 *.mif file setting 1

4) After completing the ROM and IP's setting, fill the data for rom.mif. For convenience of verification, store the same data as the address from the lower byte to higher byte in

ascending form. Right click to select **Custom Fill Cells**. See Figure 8.6. The starting address is 0, ending at 255 (previous address setting depth is 256). The initial value is 0 and the step is 1.

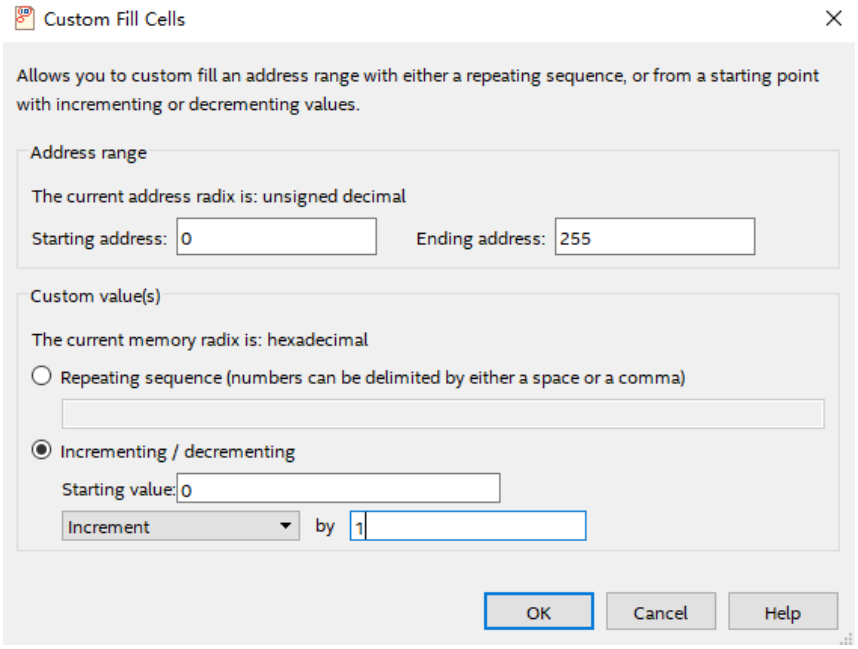
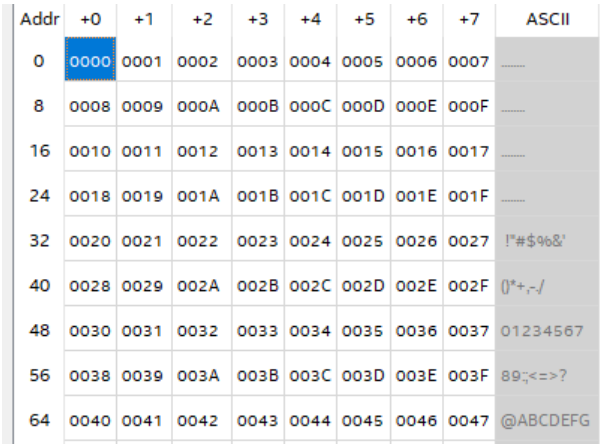


Figure 8.6 Fill date for rom.mif

5) After the setup, the system will fill in the data automatically. See Figure 8.7.



Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0000	0001	0002	0003	0004	0005	0006	0007	.....
8	0008	0009	000A	000B	000C	000D	000E	000F	.....
16	0010	0011	0012	0013	0014	0015	0016	0017	.....
24	0018	0019	001A	001B	001C	001D	001E	001F	.....
32	0020	0021	0022	0023	0024	0025	0026	0027	!"#$%&'
40	0028	0029	002A	002B	002C	002D	002E	002F	()*+,-./
48	0030	0031	0032	0033	0034	0035	0036	0037	01234567
56	0038	0039	003A	003B	003C	003D	003E	003F	89:;<=>?
64	0040	0041	0042	0043	0044	0045	0046	0047	@ABCDEFG

Figure 8.7 Part of data after auto filling

1. Refer to the design of conversion from hexadecimal to BCD in Experiment 7, display the data in ROM to the segment display.

（4）

ROM instantiation:

```
reg   [15:0] rom_q_r;

always @ (posedge BCD_clk)
      rom_q_r<=rom_q;

HEX_BCD HEX_BCD_inst(
      .hex                  (rom_q_r),
```

```
        .ones               (ones),
        .tens               (tens),
        .hundreds           (hundreds),
        .thousands          (thousands),
        .ten_thousands      (ten_thousands)
);
onep_rom    onep_rom_dut(
        .address            (sw),
        .clock              (sys_clk),
        .q                  (rom_q)
);
```

8.4 Experiment Verification

Pin assignments are consistent with Experiment 7. After the compilation is completed, the board is verified. As shown in Figure 8.8. When the DIP switch is 10010100, the decimal is 148, which means that we will read the contents of the 148th byte of the ROM, and the segment display will display 148, which is consistent with the data we have deposited.

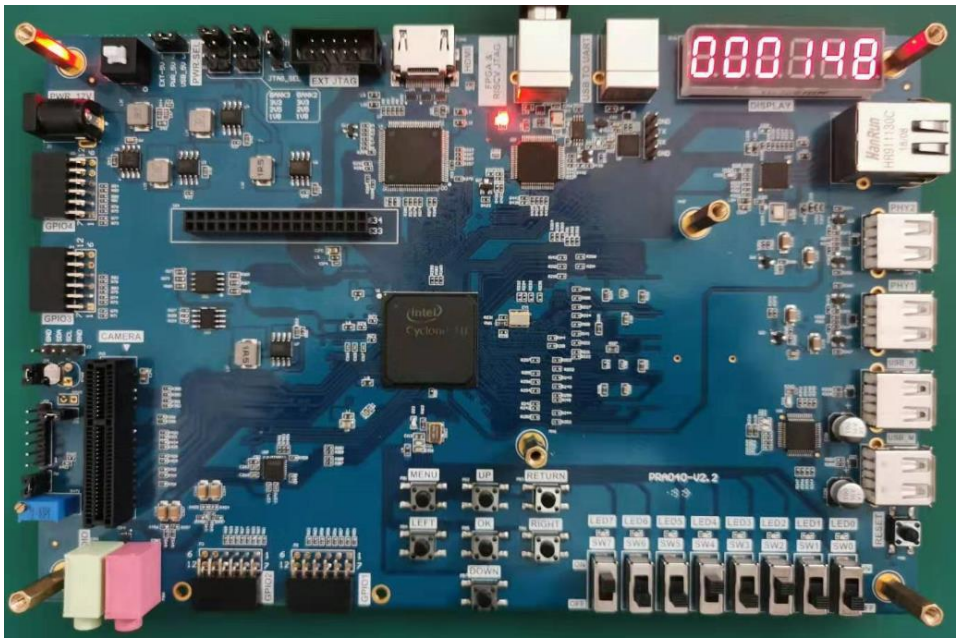


Figure 8.8 Experiment result

Experiment Summary and Reflection

1. How to use the initial file of ROM to realize the decoding, such as decoding and scanning the segment display.
2. Write a *.mif file to generate sine, cosine wave, and other function generators.
3. Comprehend application, combine the characteristic of ROM and PWM to form SPWM modulation waveform.

Experiment 9 Use Dual-port RAM to Read and Write Frame Data

9.1 Experiment Objective

（1） Learn to configure and use dual-port RAM
（2） Learn to use synchronous clock to control the synchronization of frame structure
（3） Learn to use asynchronous clock to control the synchronization of frame structure

9.2 Experiment Implement

（1） Observing the synchronization structure of synchronous clock frames using SignalTap II
（2） Extended the use of dual-port RAM
（3） Design the use of three-stage state machine
（4） Design a 16-bit data frame

1) Data is generated by an 8-bit counter: Data={~counta,counta}
2) The ID of the data frame inputted by the switch (7 bits express maximum of 128 different data frames)
3) 16-bit checksum provides data verification
   A. 16-bit checksum accumulates, discarding the carry bit
   B. After the checksum is complemented, append to the frame data
4) Provide configurable data length *data_len* by parameter
5) Packet: When the data and checksum package are written to the dual-port RAM, the userID, the frame length and the valid flag are written to the specific location of the dual-port RAM. The structure of the memory is shown in Table 9.1.

Table 9.1 Memory structure

Wr_addr	Data/Flag	Rd_addr
8'hff	{valid,ID,data_len}	8'hff
...	N/A	...
8'hnn+2	N/A	8'hnn+2
8'hnn+1	~checksum+1	8'hnn+1
8'hnn	datann	8'hnn
...	....	...
8'h01	Data1	8'h01
8'h00	Data0	8'h00

6) Read and write in an agreed order

Valid is the handshake signal. This flag provides the possibility of read and write synchronization, so the accuracy of this signal must be ensured in the program design.

9.3 Experiment

9.3.1 Introduction of the program

The first step: the establishment of the main program framework

```
module frame_ram
#(parameter data_len=250)
(
input                   inclk,
input                   rst,
input           [6:0]   sw,
output  reg     [6:0]   oID,
output  reg             rd_done,
output  reg             rd_err
);
```

The second step: the definition of the state machine

```
parameter       [2:0]       mema_idle=0,
                            mema_init=1,
                            mema_pipe0=2,
                            mema_read0=3,
                            mema_read1=4,
                            mema_wr_data=5,
                            mema_wr_chsum=6,
                            mema_wr_done=7;

parameter       [2:0]       memb_idle=0,
                            memb_init=1,
                            memb_pipe0=2,
                            memb_read0=3,
                            memb_read1=4,
                            memb_rd_data=5,
                            memb_rd_chsum=6,
                            memb_rd_done=7;
```

The third step: other definitions

```
Clock variable definition
wire                    sys_clk;
wire                    BCD_clk;
wire                    sys_rst;
reg                     ext_clk;
Dual-port RAM interface definition
```

```
reg         [7:0]       addr_a;
reg         [15:0]      data_a;
reg                     wren_a;
wire        [15:0]      q_a;
reg         [7:0]       addr_b;
reg                     wren_b;
wire        [15:0]      q_b;
Write state machine part variable definition
reg         [6:0]       user_id;
reg         [7:0]       wr_len;
reg         [15:0]      wr_chsum;
wire                    wr_done;
reg         [7:0]       counta;
wire        [7:0]       countb;
assign countb=~counta;
reg         [15:0]      rd_chsum;
reg         [7:0]       rd_len;
reg         [15:0]      rd_data;
reg                     ext_rst;
reg         [2:0]       sta;
reg         [2:0]       sta_nxt,;
Read state machine part variable definition
reg         [15:0]      rd_chsum;
reg         [7:0]       rd_len;
reg         [15:0]      rd_data;
reg         [2:0]       stb;
reg         [2:0]       stb_nxt;
```

Step 4: Generate dual-port RAM, PLL

```
dp_ram dp_ram_inst
(
    .address_a          (addr_a),
    .address_b          (addr_b),
    .clock              (sys_clk),
    .data_a             (data_a),
    .data_b             (16'b0),
    .wren_a             (wren_a),
    .wren_b             (wren_b),
    .q_a                (q_a),
    .q_b                (q_b)
);

pll_sys_rst pll_sys_rst_inst
(
```

```
    .inclk          (inclk),
    .sys_clk        (sys_clk),
    .BCD_clk        (BCD_clk),
    .sys_rst        (sys_rst)
);
```

The RAM is 16 bits wide and 256 depth. The PLL inputs a 50 MHz clock, outputs 100 MHz as the operating clock of other modules, and 20 MHz is used to drive the segment display.

Step 5: data generation counter

```
always @ (posedge sys_clk)
if(sys_rst) begin
    counta    <= 0;
    user_id    <= 0;
end
else begin
    counta <=counta + 1;
    user_id <= sw;
end
```

Step 6: write state machine

```
assign wr_done = (wr_len == (data_len - 1'b1));
// Think why using wr_len==data_len-1 instead of wr_len==data_len
//First stage
always @ (posedge sys_clk)
begin
    if (sys_rst) begin
        sta = mema_idle;
    end
    else
        sta = sta_nxt;
end
//Second stage
always @ (*)
begin
    case (sta)
        mema_idle    :    sta_nxt = mema_init;
        mema_init    :    sta_nxt = mema_pipe0;
        mema_pipe0   :    sta_nxt = mema_read0;
        mema_read0   :
        begin
            if (!q_a[15])
                sta_nxt = mema_read1;
            else
```

```
                    sta_nxt = sta;
            end
            mema_read1  :
            begin
                if (!q_a[15])
                    sta_nxt = mema_wr_data;
                else
                    sta_nxt = sta;
            end
            mema_wr_data    :
            begin
                if (wr_done)
                    sta_nxt = mema_wr_chsum;
                else
                    sta_nxt = sta;
            end
            mema_wr_chsum   :   sta_nxt = mema_wr_done;
            mema_wr_done    :   sta_nxt = mema_init;
            default   :   sta_nxt   =   mema_idle;
        endcase
end
```

```
//Third stage
always @ (posedge sys_clk)
begin
    case (sta)
        mema_idle    :
        begin
            addr_a          <= 8'hff;
            wren_a          <= 1'b0;
            data_a          <= 16'b0;
            wr_len          <= 8'b0;
            wr_chsum        <= 0;
        end
        mema_init, mema_pipe0, mema_read0, mema_read1       :
        begin
            addr_a          <= 8'hff;
            wren_a          <= 1'b0;
            data_a          <= 16'b0;
            wr_len          <= 8'b0;
            wr_chsum        <= 0;
        end
        mema_wr_data    :
        begin
```

```
                addr_a          <= addr_a + 1'b1;
                wren_a          <= 1'b1;
                data_a          <= {countb, counta};
                wr_len          <= wr_len + 1'b1;
                wr_chsum        <= wr_chsum + {countb, counta};
            end
            mema_wr_chsum   :
            begin
                addr_a    <= addr_a + 1'b1;
                wr_len    <= wr_len + 1'b1;
                wren_a    <= 1'b1;
                data_a    <= (~wr_chsum) + 1'b1;
            end
            mema_wr_done    :
            begin
                addr_a    <= 8'hff;
                wren_a    <= 1'b1;
                data_a    <= {1'b1, user_id, wr_len};
            end
            default    :      ;
        endcase
end
```

Write order:

1. Read the flag of the 8'hff address (control word). If valid=1'b0, the program proceeds to the next step, otherwise waits
2. Address plus 1, 8'hff+1 is exactly zero, write data from 0 address and calculate the checksum
3. Determine whether the interpretation reaches the predetermined data length. If so, proceeds to next step, otherwise the data is written, and the checksum is calculated.
4. checksum complements and write to memory
5. Write the control word in the address 8'hff, packet it

Step 7: Read state machine

```
//First stage
always @ (posedge sys_clk)
begin
    if (ext_rst) begin
        stb = memb_idle;
    end
    else
        stb = stb_nxt;
end
//Second stage
```

```
always @ (*)
begin
    case (stb)
        memb_idle    :    stb_nxt = memb_init;
        memb_init    :    stb_nxt = memb_pipe0;
        memb_pipe0   :    stb_nxt = memb_read0;
        memb_read0   :
        begin
            if (q_b[15])
                stb_nxt = memb_read1;
            else
                stb_nxt = memb_init;
        end
        memb_read1   :
        begin
            if (q_b[15])
                stb_nxt = memb_rd_data;
            else
                stb_nxt = memb_init;
        end
        memb_rd_data     :
        begin
            if(rd_done)
                stb_nxt = memb_rd_chsum;
            else
                stb_nxt = stb;
        end
        memb_rd_chsum   :    stb_nxt = memb_rd_done;

        memb_rd_done    :    stb_nxt = memb_init;
        default    :    stb_nxt = memb_idle;
    endcase
end
```

```
//Stage three, the actual operation needs to be driven by the edge of the clock.
always @ (posedge sys_clk)
begin
    case (stb)
        memb_idle    :
        begin
            addr_b          <= 8'hff;
            rd_data         <= 0;
            rd_chsum        <= 0;
            wren_b          <= 1'b0;
```

```
            rd_len          <= 8'b0;
            oID             <= 7'b0;
            rd_err          <= 1'b0;
        end
        memb_init     :
        begin
            addr_b          <= 8'hff;
            rd_data         <= 0;
            rd_chsum        <= 0;
            wren_b          <= 1'b0;
            rd_len          <= 8'b0;
            oID             <= 7'b0;
            rd_err          <= 1'b0;
        end
        memb_pipe0  :
        begin
            addr_b    <= 8'b0;
        end
        memb_read0  :
        begin
        if (q_b[15])
            addr_b          <= addr_b + 1'b1;
        else
            addr_b          <= 8'hff;
            rd_data         <= 0;
            rd_chsum        <= 0;
            wren_b          <= 1'b0;
            rd_len          <= 8'b0;
            oID             <= 7'b0;
        end
        memb_read1  :
        begin
        if(q_b[15])
            addr_b          <= addr_b + 1'b1;
        else
            addr_b          <= 8'hff;
            rd_data         <= 0;
            rd_chsum        <= 0;
            wren_b          <= 1'b0;
            rd_len          <= q_b[7:0];
            oID             <= q_b[14:8];
        end
        memb_rd_data      :
        begin
```

```
                addr_b          <= addr_b + 1'b1;
                rd_data         <= q_b;
                rd_chsum        <= rd_chsum + rd_data;
                wren_b          <= 1'b0;
                rd_len          <= rd_len - 1'b1;
            end
            memb_rd_chsum   :
            begin
                addr_b          <= 8'hff;
                wren_b          <= 1'b0;
                if (|rd_chsum)
                    rd_err      <= 1'b1;
            end
            memb_rd_done    :
            begin
                addr_b          <= 8'hff;
                wren_b          <= 1'b1;
            end
            default    :    ;
        endcase
end
```

Read order

1. *idle* is the state after reset
2. *Init*: Initialization, set the address to 8'hff
3. *Rd_pipe0*: Add a latency (since the read address and data are both latched). Address +1, forming a pipeline structure
4. *Read0*: Set the address to 8'hff, read the control word and judge whether the valid bit is valid.
    a. If *valid*=1'b1, address +1, proceeds to the next step
    b. If *valid*=1'b0, it means the packet is not ready yet, the address is set to be 8'hff and returns to the init state.
5. *Read1*: Read the control word again
    a. If *valid*=1'b1, address+1, ID and data length are assigned to the corresponding variables and proceeds to the next step
    b. If *valid*=1'b0, it means the packet is not ready yet, the address is set to 8'hff, and returns to the *init* state.
6. *Rd_data*:
    a. Read data and pass to data variables
    b. Calculate checksum, data_len - 1
    c. Determine whether the data_len is 0
        i. 0: all data has been read, proceeds to the next step
        ii. Not 0: continue the operation in current state
7. *rd_chsum*: Read the value of checksum and calculate the last checksum. Correct the

data and set the flag of *rd_err*

8. *rd_done*: The last step clears the valid flag in memory and opens the write enable for the next packet.

9.3 Experiment Verification

The first step: pin assignment

Table 9.2 Frame data read and write experiment pin mapping

Signal Name	Network Label	FPGA Pin	Port Description
Inclk	CLK_50M	G21	Input clock
rst	PB3	Y6	reset
SW[7]	SW7	W6	Switch inpuut 7
SW[6]	SW6	Y8	Switch inpuut 6
SW[5]	SW5	W8	Switch inpuut 5
SW[4]	SW4	V9	Switch inpuut 4
SW[3]	SW3	V10	Switch inpuut 3
SW[2]	SW2	U10	Switch inpuut 2
SW[1]	SW1	V11	Switch inpuut 1
SW[0]	SW0	U11	Switch inpuut 0

Step 2: Observe the read and write results of the dual-port RAM with SignalTap

（1） In order to facilitate the observation of the read and write state machine synergy results, the data length is changed to 4 here, recompile and download. Users can test themselves using long data.

```
module frame_ram
#(parameter data_len=4)
(
input                   inclk,
input                   rst,
input          [6:0]    sw,
output   reg   [6:0]    oID,
output   reg            rd_done,
output   reg            rd_err
);
```

（2） Observe the simulation result

1) Observe the handshake mechanism through dual-port RAM
   A. Determine whether the reading is started after the packet is writte
   B. Determine whether the write packet is blocked before reading the entire packet is completed.
2) Observe the external interface signal and status

A. *rd_done*, *rd_err*

Set *rd_err* = 1, or the rising edge is the trigger signal to observe whether the error signal is captured.

B. Observe whether *wren_a*, *wren_b* signal and the state machine jump are strictly matched to meet the design requirements.

（3） SignalTap result. See Figure 9.1.

Experiment Summary and Reflection

（1） Review the design requirements. How to analyze an actual demand, to gradually establish a model of digital control and state machine and finally design.

（2） Modify the third stage of the state machine into the if...else model and implement.

（3） Focus on thinking If the read and write clocks are different. After it becomes an asynchronous mechanism, how to control the handshake.

（4） According to the above example, consider how dual-port RAM can be used in data acquisition, asynchronous communication, embedded CPU interface, and DSP chip interface.

（5） How to build ITCM with dual-port RAM and DTCM preparing for future CPU design.

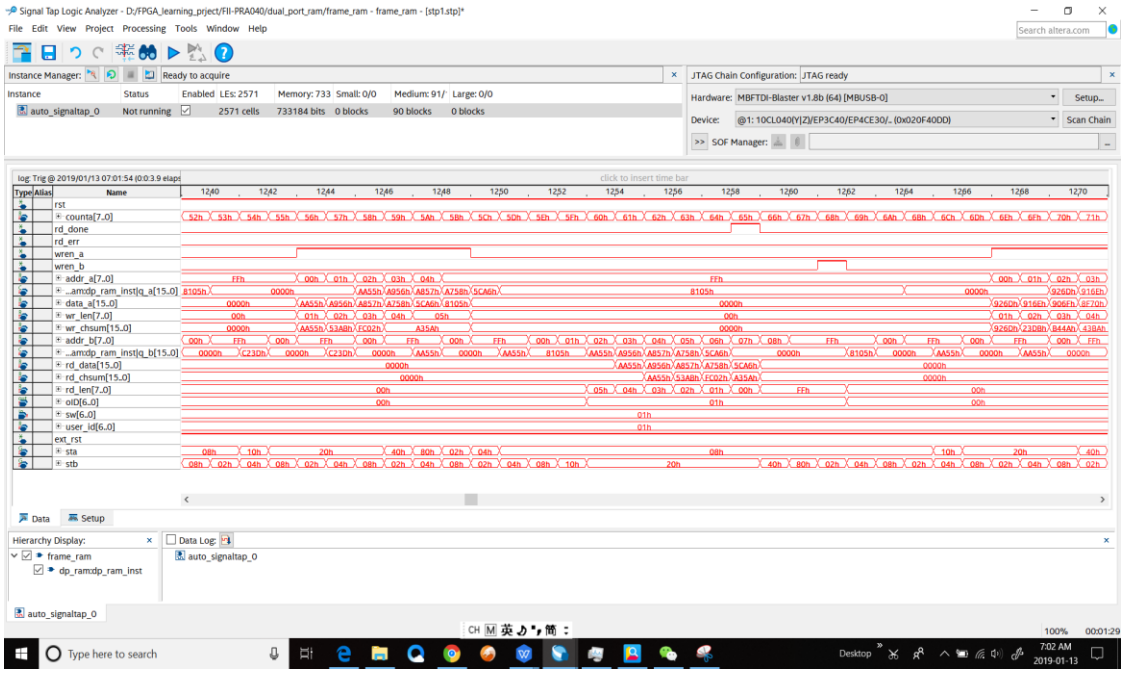


Figure 9.1 SingalTap II simulation

Experiment 10 Asynchronous Serial Port Design and Experiment

10.1 Experiment Objective

Because asynchronous serial ports are very common in industrial control, communication, and software debugging, they are also vital in FPGA development.

（1） the basic principles of asynchronous serial port communication, handshake mechanism, data frame

（2） Master asynchronous sampling techniques

（3） Review the frame structure of the data packet

（4） Learning FIFO

（5） Joint debugging with common debugging software of PC (SSCOM, teraterm, etc.)

10.2 Experiment Implement

（1） Design and transmit full-duplex asynchronous communication interface Tx, Rx

（2） Baud rate of 11520 bps, 8-bit data, 1 start bit, 1 or 2 stop bits

（3） Receive buffer (Rx FIFO), transmit buffer (Tx FIFO)

（4） Forming a data packet

（5） Packet parsing

10.3 Experiment

10.3.1 Introduction to the UART Interface

A USB-B interface and a CP2102 chip are onboard for serial data communication.

The CP2102 features a high level of integration with a USB 2.0 full-speed function controller, USB transceiver, oscillator, EEPROM, and asynchronous serial data bus (UART) to support modem full-featured signals without the need for any external USB devices. See Figure 10.1 for the pyhsical picture.

Figure 10.1 USB-B Interface and CP2102 Chip Physical Picture

10.3.2 Hardware Design

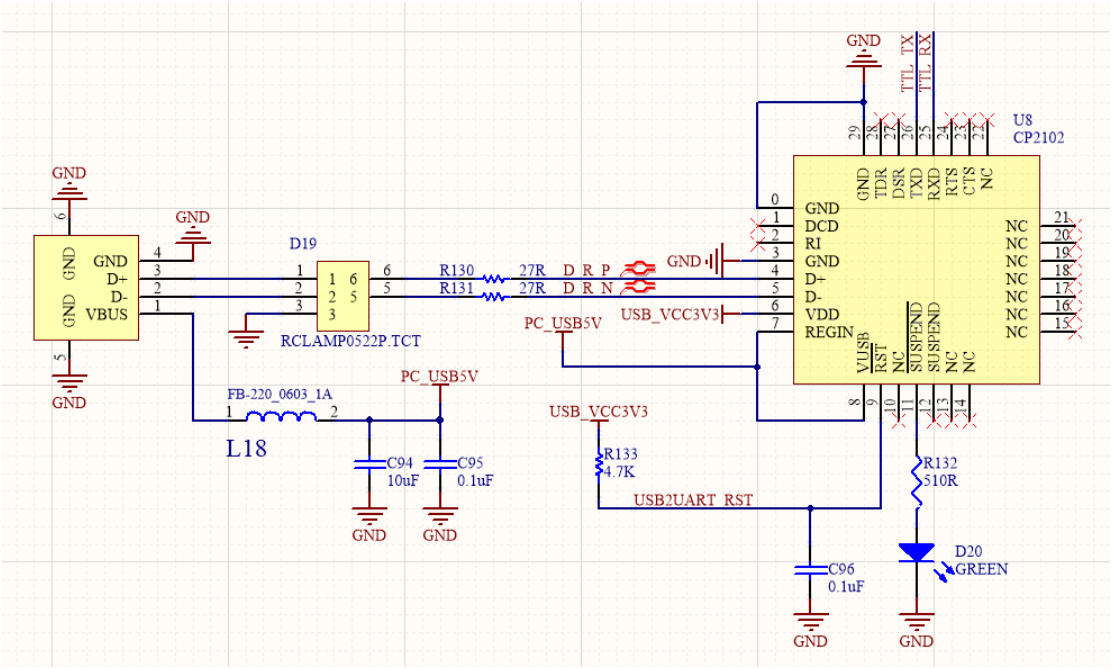


Figure 10.2 Schematics of the serial port

The principle of USB serial port conversion is shown in Figure 10.2. The TTL_TX and TTL_RX of the CP2102 are connected to the FPGA to transmit and receive data. After being processed internally by the chip, the D_R_P and D_R_N are connected to the USB interface through a protection chip, and the data is transmitted with the PC to implement serial communication.

10.3.3 Introduction of the Program

The first step: the main program architecture

```
module uart_top
(
    input           inclk,
    input           rst,
    input   [1:0]   baud_sel,
    input           tx_wren,
    input           tx_ctrl,
    input   [7:0]   tx_data,
    input           tx_done,
    output          txbuf_rdy,

    input           rx_rden,
    output  [7:0]   rx_byte,
    output          rx_byte_rdy,
    output          sys_rst,
    output          sys_clk,
    input           rx_in,
```

```
    output              tx_out
);
```

There are a lot of handshake signals here, with the tx prefix for the transmit part of the signal, and the rx prefix is for the receive part of the signal.

Step 2: create a new baud rate generator file

（1） Input clock 7.3728MHz (64 times 115200). The actual value is 7.377049MHz, which is because the coefficient of the PLL is an integer division, while the error caused by that is not large, and can be adjusted by the stop bit in asynchronous communication. See Figure 10.3.

Fine solution

A. Implemented with a two-stage PLL for a finer frequency

B. The stop bit is set to be 2 bits, which can effectively eliminate the error.

This experiment will not deal with the precision. The default input frequency is 7.3728 MHz.

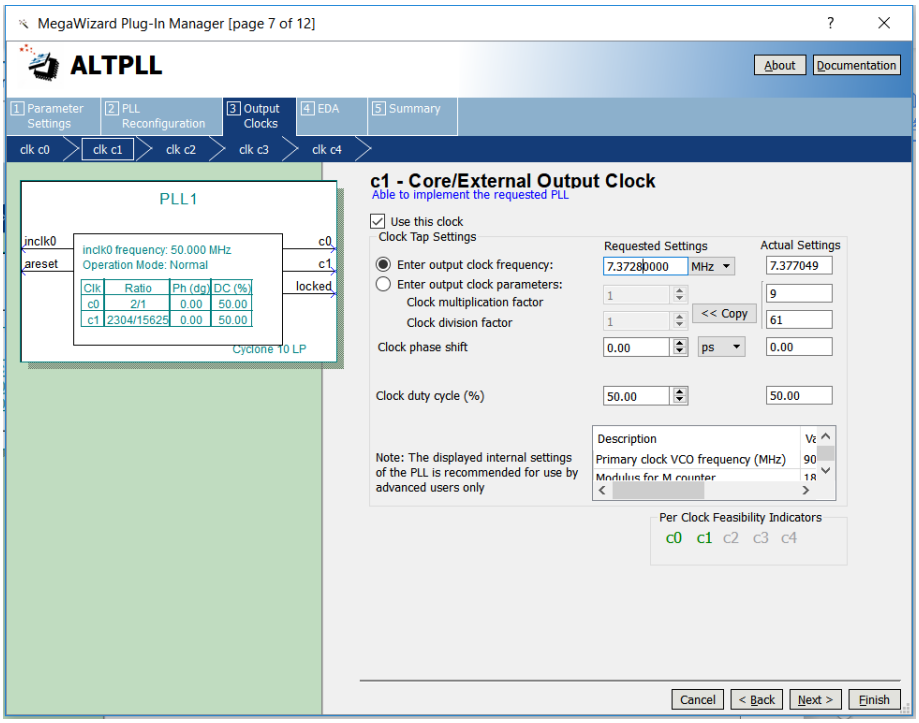


Figure 10.3 PLL setting

C. Supported baud rates are 115200，57600，38400，19200

D. The default baud rate is 115200

（2） Source file of designing baud rate

```
// Send baud rate, clock frequency division selection
wire [8:0] frq_div_tx;
assign frq_div_tx    =     (baud_sel == 2'b00) ? 9'd63:
                           (baud_sel == 2'b01) ? 9'd127:
                           (baud_sel == 2'b10) ? 9'd255:9'd511;
reg   [8:0] count_tx=9'd0;
always @ (posedge inclk)
```

```
if (rst) begin
      count_tx  <= 9'd0;
      baud_tx   <= 1'b0;
end
else begin
      if (count_tx == frq_div_tx) begin
            count_tx  <= 9'd0;
            baud_tx   <= 1'b1;
      end
      else begin
            count_tx  <= count_tx + 1'b1;
            baud_tx   <= 1'b0;
      end
end
```

Four different gear positions are set to select the baud rate, corresponding to the step 2, (1). The baud rate of the receiving part is similar to that of the transmitting part.

Step 3: Design the send buffer file *tx_buf*

（1） 8-bit FIFO, depth is 256, read/write clock separation, write full flag, read empty flag

（2） Interface and handshake

1) *rst* reset signal
2) *wr_clk* write clock
3) *tx_clk* send clock
4) 8-bit write data *tx_data*
5) *wr_en* write enable
6) *ctrl* writes whether the input is a data or a control word
7) *rdy* buffer ready, can accept the next data frame

Transmit buffer instantiation file

```
tx_buf
#(.TX_BIT_LEN(8),.STOP_BIT(2))
tx_buf_inst
(
.sys_rst          (sys_rst),
.uart_rst         (uart_rst),
.wr_clk           (sys_clk),
.tx_clk           (uart_clk),
.tx_baud          (tx_baud),
.tx_wren          (tx_wren),
.tx_ctrl          (tx_ctrl),
.tx_datain        (tx_data),
.tx_done          (tx_done),
.txbuf_rdy        (txbuf_rdy),
.tx_out           (tx_out)
```

```
);
```

（1） Serial transmission, interface and handshake file design

1) Interface design

A. tx_rdy, send vacancy, can accept new 8-bit data

B. tx_en, send data enable, pass to the sending module 8-bit data enable signal

C. tx_data, 8-bit data to be sent

D. tx_clk, send clock

E. tx_baud, send baud rate

2) Instantiation

```
tx_transmit
#(.DATA_LEN(TX_BIT_LEN),
   .STOP_BIT(STOP_BIT)
)
tx_transmit_inst
(
.tx_rst              (uart_rst),
.tx_clk              (tx_clk),
.tx_baud        (tx_baud),
.tx_en               (tx_en),
.tx_data        (tx_data),
.tx_rdy              (trans_rdy),
.tx_out              (tx_out)
);
```

（2） Write a testbench file to simulate the transmit module. (*tb_uart*)

（3） ModelSim simulation waveforms for transmit module. See Figure 10.4.

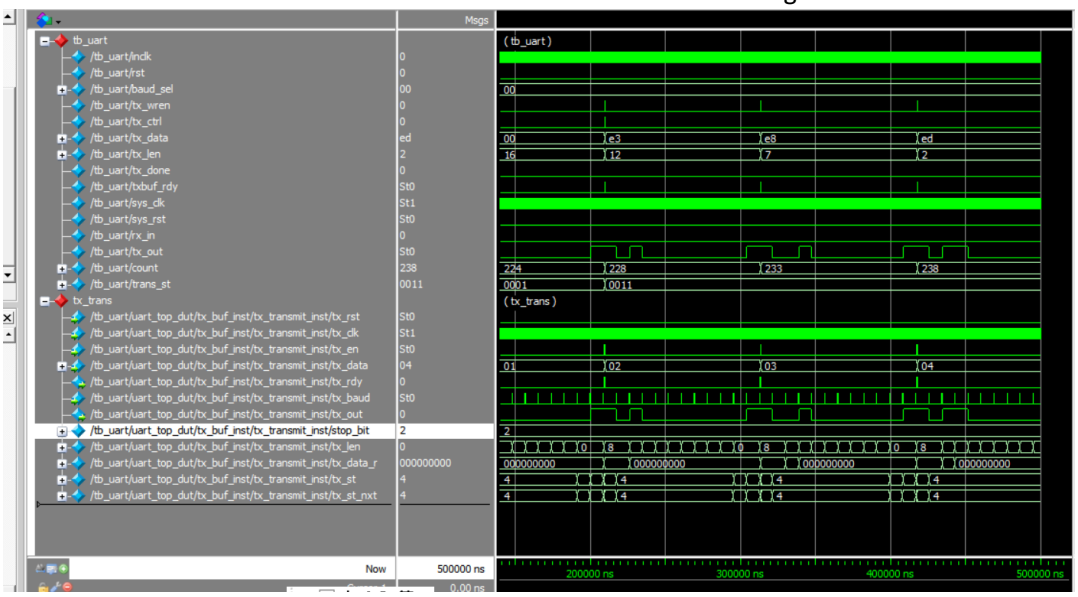


Figure 10.4 Serial port sending MoselSim simulation waveform

（4） Extended design (extended content is only reserved for users to think and practice)

1) Design the transmitter to support 5, 6, 7, 8-bit PHY (Port physical layer)

2) Support parity check

（5） The settings of the above steps involve FIFO, PLL, etc. (Refer to uart_top project file)

The fourth step: UART receiving module design

（1） Design of rx_phy.v

1) Design strategies and steps

A. Use 8 times sampling: so rx_baud is different from tx_baud, here sampling is rx_band = 8*tx_band

B. Adopting judgments to the receiving data

Determine whether the data counter is greater than 4 after the sampling value is counted.

2) Steps to receive data:

A. Synchronization: refers to how to find the start bit from the received 0101 (*sync_dtc*)

B. Receive start bit (*start*)

C. Cyclically receive 8-bit data

D. Receive stop bit (determine whether it is one stop bit or two stop bits)

a. Determine if the stop bit is correct

i. Correct, jump to step B

ii. Incorrect, jump to step A, resynchronize

b. Do not judge, jump directly to B, this design adopts the scheme of no judgment

（2） Design of rx_buf

1) Design strategies and steps

A. Add 256 depth, 8-bit fifo

a. Read and write clock separation

b. Asynchronous clear (internal synchronization)

c. Data appears before the *rdreq* in the read port

B. Steps:

a. Initialization: *fifo*, *rx_phy*

b. Wait: FIFO full signal (*wrfull*) is 0

c. Write: Triggered by *rx_phy_byte*, *rx_phy_rdy* of *rx_phy*:

d. End of writing

e. Back to step b and continue to wait

f. *rx_buf.v* source program (Reference to project files)

g. Receive module simulation

Content and steps

i. *tx, rx* loopback test (assign *rx_in* = *tx_out*)

ii. Continue to use the testbench file in the TX section

iii. Write the testbench of *rx*

h. ModelSim simulation. See Figure 10.5.

i. Reflection and development

I. Modify the program to complete the 5, 6, 7, 8-bit design

II. Completing the design of the resynchronization when the *start* and *stop*

have errors of the receiving end *rx_phy*

III. Complete the analysis and packaging of the receiving data frame of *rx_buf*

IV. Using multi-sampling to design 180° alignment data sampling method, compare FPGA resources, timing and data recovery effects

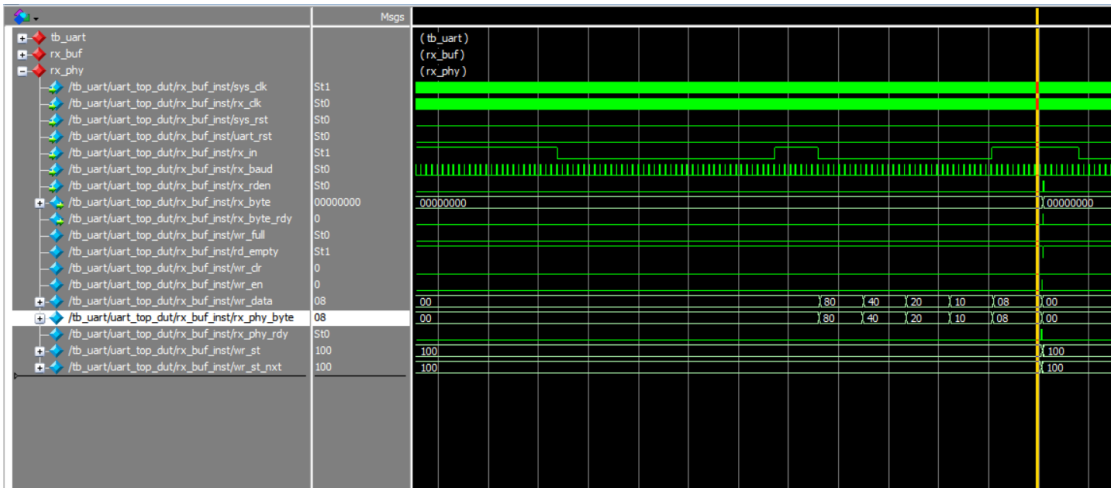
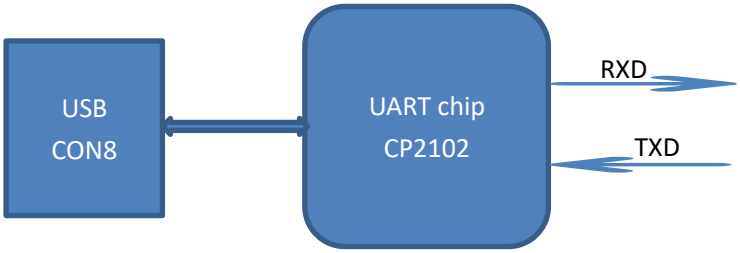


Figure 10.5 rx_phy wave form

10.4 Experiment Verification

（1） Hardware interface, FII-PRA040 development board has integrated USB to serial port conversion



（2） Write a hardware test file

1) Test plan: connect development board CON8 to host USB interface
2) Using test software such as teraterm, SSCOM3, etc., you can also write a serial communication program (C#, C++, JAVA, Python...)
3) PC sends data in a certain format
4) The test end uses a counter to generate data in a certain format.
5) Write the test program *hw_tb_uart* and instantiate *uart_top* in it.
6) Set *hw_tb_uart* to the top level, instantiate the previous program, and then verify it

（3） Pin assignmnets:

Table 10.1 Serial port experiment pin mapping

Signal Name	Network Label	FPGA Pin	Port Description
Inclk	CLK_50M	G21	Input clock
rst	KEY2	Y6	Reset signal
rx_in	TTL_RX	E16	Serial data received
tx_out	TTL_TX	F15	Serial data transmiteed

（4） Observe the experiment result

1) Observe the data received by the PC. See in Figure 10.6.

2) Observe the data received by the FPGA with SignalTap II

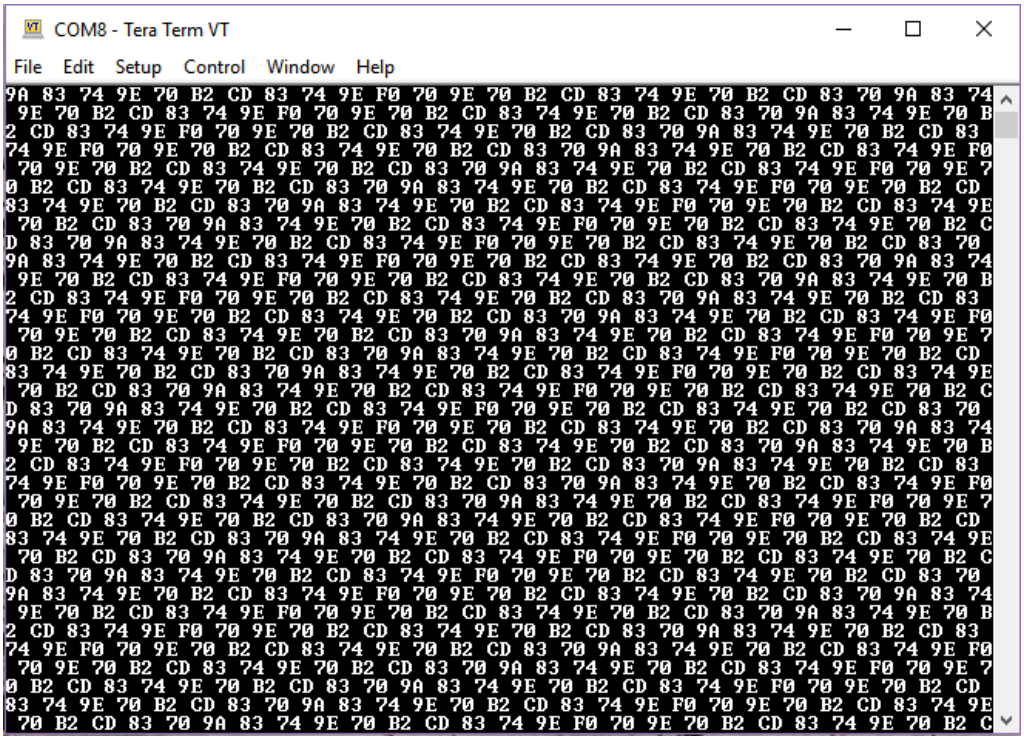


Figure 10.5 Data transmitted displayed on the host computer

（5） The receiving part has been skipped here. You are encouraged to try it on your own.

Experiment 11 IIC Protocol Transmission

11.1 Experiment Objective

（1） Learning the basic principles of asynchronous IIC bus, and the IIC communication protocol

（2） Master the method of reading and writing EEPROM

（3） Joint debugging using logic analyzer

11.2 Experiment Implement

（1） Correctly write a number to any address in the EEPROM (this experiment writes to the register of 8'h03 address) through the FPGA (here changes the written 8-bit data value by (SW7~SW0)). After writing in successfully, read the data as well. The read data is displayed directly on the segment display.

（2） Download the program into the FPGA and press the left push button to execute the data write into EEPROM operation. Press the right push button to read the data that was just written.

（3） Determine whether it is correct or not by reading the displayed number on the segment display. If the segment display has the same value as written value, the experiment is successful.

（4） Analyze the correctness of the internal data with SignalTap II and verify it with the display of the segment display.

11.3 Experiment

11.3.1 Introduction of EEPROM and IIC Protocol

（1） Introduction of EEPROM

EEPROM (Electrically Erasable Programmable Read Only M emory) refers to a charged erasable programmable read only register. It is a memory chip that does not lose data after turning off power.

On the experiment board, there is an IIC interface EEPROM chip 24LC02 with a capacity of 256 bytes. Users can store some hardware configuration data or user information due to the characteristics that the data is not lost after power-off.

（2） The overall timing protocol of IIC is as follows

1) Bus idle state: *SDA*, *SCL* are high
2) Start of IIC protocol: SCL stays high, SDA jumps from high level to low level, generating a start signal

3) IIC read and write data stage: including serial input and output of data and response signal issued by data receiver

4) IIC transmission end bit: *SCL* is in high level, *SDA* jumps from low level to high level, and generates an end flag. See Figure 11.1.

5) *SDA* must remain unchanged when *SCL* is high. It changes only when *SCL* is low

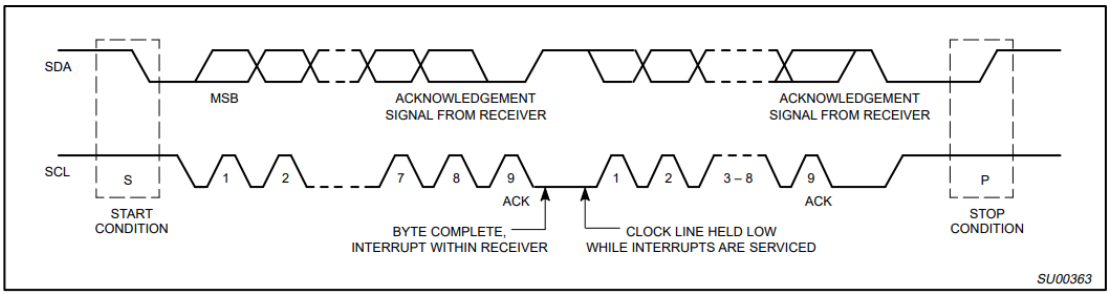


Figure 11.1 Timing protocol of IIC

11.3.2 Hardware Introduction

Each IIC device has a device address. When some device addresses are shipped from the factory, they are fixed by the manufacturer (the specific data can be found in the manufacturer's data sheet). Some of their higher bits are determined, and the lower bits can be configured by the user according to the requirement. The higher four-bit address of the EEPROM chip 24LC02 used by the develop board has been fixed to 1010 by the component manufacturer. The lower three bits are linked in the develop board as shown below, so the device address is 1010000. See Figure 11.2. EEPROM reads and writes data from the FPGA through the *I2C_SCL* clock line and the *I2C_SDA* data line.

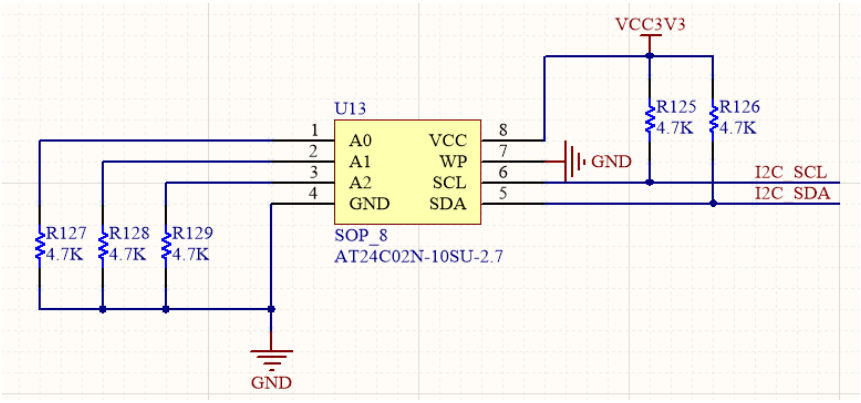


Figure 11.2 EEPROM schematics of IIC device

11.3.3 Introduction to the program

This experiment has two main modules, I2C reading and writing module and LED display module; The first module is mainly introduced here.

The first step: establishment of the main program framework

```
module iic_com(
    input                   clk,
```

```
    input                   rst_n,
    input       [7:0]       data,
    input                   sw1,sw2,
    inout                   scl,
    inout                   sda,
    output  reg             iic_done,
    output      [7:0]       dis_data
);
```

The input 8-bit data is needed to be written into the EEPROM, provided by an 8-bit DIP switch.

Step 2: Create clock *I2C_CLK*

```
reg     [2:0]   cnt;
reg     [8:0]   cnt_delay;
reg             scl_r;
reg             scl_link ;
always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        cnt_delay <= 9'd0;
    else if (cnt_delay == 9'd499)
        cnt_delay <= 9'd0;
    else
        cnt_delay <= cnt_delay + 1'b1;
end
always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        cnt <= 3'd5;
    else begin
        case (cnt_delay)
            9'd124:   cnt <= 3'd1;      //cnt=1:scl
            9'd249:   cnt <= 3'd2;      //cnt=2:scl
            9'd374:   cnt <= 3'd3;      //cnt=3:scl
            9'd499:   cnt <= 3'd0;      //cnt=0:scl
            default: cnt<=3'd5;
        endcase
    end
end

`define SCL_POS         (cnt==3'd0)         //cnt=0:scl
`define SCL_HIG         (cnt==3'd1)         //cnt=1:scl
`define SCL_NEG         (cnt==3'd2)         //cnt=2:scl
`define SCL_LOW         (cnt==3'd3)         //cnt=3:scl
```

```
always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        scl_r <= 1'b0;
    else if (cnt == 3'd0)
        scl_r <= 1'b1;
    else if (cnt == 3'd2)
        scl_r <= 1'b0;
end

assign scl = scl_link ? scl_r: 1'bz ;
```

First, use the system 50 MHz clock to get a 100 kHz clock with a period of 10us by frequency division as the transmission clock of the IIC protocol. Then, the rising edge, the high state, the falling edge and the low state of the clock are defined by the counter, prepared for the subsequent data reading and writing and the beginning of the IIC protocol. The last line of code means to define a data valid signal. Only when the signal is high, that is, when the data is valid, the IIC clock is valid again, otherwise it is in high impedance. This is also set according to the IIC transport protocol.

The third step: specific implementation of I2C transmission

```
`define    DEVICE_READ      8'b1010_0001
`define    DEVICE_WRITE     8'b1010_0000
`define    BYTE_ADDR        8'b0000_0011

parameter     IDLE      = 4'd0;
parameter     START1    = 4'd1;
parameter     ADD1      = 4'd2;
parameter     ACK1      = 4'd3;
parameter     ADD2      = 4'd4;
parameter     ACK2      = 4'd5;
parameter     START2    = 4'd6;
parameter     ADD3      = 4'd7;
parameter     ACK3      = 4'd8;
parameter     DATA      = 4'd9;
parameter     ACK4      = 4'd10;
parameter     STOP1     = 4'd11;
parameter     STOP2     = 4'd12;

reg       [7:0]     db_r;
reg       [7:0]     read_data;
reg       [3:0]     cstate;
reg                 sda_r;
```

```
reg                 sda_link;
reg     [3:0]       num;

always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n) begin
        cstate          <= IDLE;
        sda_r           <= 1'b1;
        scl_link        <= 1'b1;
        sda_link        <= 1'b1;
        num             <= 4'd0;
        read_data       <= 8'b0000_0000;
        cnt_5ms         <= 20'h00000;
        iic_done  <= 1'b0;
    end
    else case (cstate)
        IDLE :
        begin
            sda_link  <= 1'b1;
            scl_link  <= 1'b1;
            iic_done  <= 1'b0 ;
            if (sw1_r || sw2_r) begin
                db_r<= `DEVICE_WRITE;
                cstate     <= START1;
            end
            else cstate <= IDLE;
        end
        START1   :
        begin
            if (`SCL_HIG) begin
                sda_link  <= 1'b1;
                sda_r           <= 1'b0;
                num             <= 4'd0;
                cstate          <= ADD1;
            end
            else cstate <= START1;
        end
        ADD1     :
        begin
            if (`SCL_LOW) begin
                if (num == 4'd8) begin
                    num             <= 4'd0;
                    sda_r           <= 1'b1;
                    sda_link  <= 1'b0;
```

```
                cstate          <= ACK1;
            end
            else begin
                cstate          <= ADD1;
                num             <= num + 1'b1;
                case (num)
                    4'd0 : sda_r <= db_r[7];
                    4'd1 : sda_r <= db_r[6];
                    4'd2 : sda_r <= db_r[5];
                    4'd3 : sda_r <= db_r[4];
                    4'd4 : sda_r <= db_r[3];
                    4'd5 : sda_r <= db_r[2];
                    4'd6 : sda_r <= db_r[1];
                    4'd7 : sda_r <= db_r[0];
                    default : ;
                endcase
            end
        end
        else cstate <= ADD1;
    end
    ACK1    :
    begin
        if (`SCL_NEG) begin
            cstate    <= ADD2;
            db_r<= `BYTE_ADDR;
        end
        else cstate    <= ACK1;
    end
    ADD2    :
    begin
        if (`SCL_LOW) begin
            if (num == 4'd8) begin
                num             <= 4'd0;
                sda_r           <= 1'b1;
                sda_link   <= 1'b0;
                cstate          <= ACK2;
            end
            else begin
                sda_link <= 1'b1;
                num <= num+1'b1;
                case (num)
                    4'd0 : sda_r <= db_r[7];
                    4'd1 : sda_r <= db_r[6];
                    4'd2 : sda_r <= db_r[5];
```

```
                        4'd3 : sda_r <= db_r[4];
                        4'd4 : sda_r <= db_r[3];
                        4'd5 : sda_r <= db_r[2];
                        4'd6 : sda_r <= db_r[1];
                        4'd7 : sda_r <= db_r[0];
                        default : ;
                    endcase
                        cstate <= ADD2;
                    end
                end
        else cstate <= ADD2;
    end
    ACK2    :
    begin
        if (`SCL_NEG) begin
            if (sw1_r) begin
                cstate    <= DATA;
                db_r<= data_tep;
            end
            else if (sw2_r) begin
                db_r<= `DEVICE_READ;
                cstate    <= START2;
            end
        end
        else cstate <= ACK2;
    end
    START2  :
    begin
        if (`SCL_LOW) begin
            sda_link  <= 1'b1;
            sda_r         <= 1'b1;
            cstate        <= START2;
        end
        else if (`SCL_HIG) begin
            sda_r         <= 1'b0;
            cstate        <= ADD3;
        end
        else cstate       <= START2;
    end
    ADD3    :
    begin
        if (`SCL_LOW) begin
            if (num == 4'd8) begin
                num           <= 4'd0;
```

```
                    sda_r           <= 1'b1;
                    sda_link  <= 1'b0;
                    cstate          <= ACK3;
                end
                else begin
                    num <= num + 1'b1;
                    case (num)
                        4'd0 : sda_r <= db_r[7];
                        4'd1 : sda_r <= db_r[6];
                        4'd2 : sda_r <= db_r[5];
                        4'd3 : sda_r <= db_r[4];
                        4'd4 : sda_r <= db_r[3];
                        4'd5 : sda_r <= db_r[2];
                        4'd6 : sda_r <= db_r[1];
                        4'd7 : sda_r <= db_r[0];
                        default: ;
                    endcase

                end
            end
            else cstate <= ADD3;
        end
        ACK3    :
        begin
            if (`SCL_NEG) begin
                cstate          <= DATA;
                sda_link  <= 1'b0;
            end
            else cstate         <= ACK3;
        end
        DATA    :
        begin
            if (sw2_r) begin
                if (num <= 4'd7) begin
                    cstate <= DATA;
                    if(`SCL_HIG) begin
                        num <= num + 1'b1;
                        case (num)
                            4'd0 : read_data[7] <= sda;
                            4'd1 : read_data[6] <= sda;
                            4'd2 : read_data[5] <= sda;
                            4'd3 : read_data[4] <= sda;
                            4'd4 : read_data[3] <= sda;
                            4'd5 : read_data[2] <= sda;
```

```
                        4'd6 : read_data[1] <= sda;
                        4'd7 : read_data[0] <= sda;
                        default: ;
                    endcase

                end
            end
            else if((`SCL_LOW) && (num == 4'd8)) begin
                num     <= 4'd0;
                cstate  <= ACK4;
            end
            else cstate  <= DATA;
        end
        else if (sw1_r) begin
            sda_link <= 1'b1;
            if (num <= 4'd7) begin
                cstate <= DATA;
                if (`SCL_LOW) begin
                    sda_link  <= 1'b1;
                    num           <= num + 1'b1;
                    case (num)
                        4'd0 : sda_r <= db_r[7];
                        4'd1 : sda_r <= db_r[6];
                        4'd2 : sda_r <= db_r[5];
                        4'd3 : sda_r <= db_r[4];
                        4'd4 : sda_r <= db_r[3];
                        4'd5 : sda_r <= db_r[2];
                        4'd6 : sda_r <= db_r[1];
                        4'd7 : sda_r <= db_r[0];
                        default: ;
                    endcase
                end
            end
            else if ((`SCL_LOW) && (num==4'd8)) begin
                num           <= 4'd0;
                sda_r         <= 1'b1;
                sda_link  <= 1'b0;
                cstate        <= ACK4;
            end
            else cstate <= DATA;
        end
    end
    ACK4    :
    begin
```

```
                if (`SCL_NEG)
                    cstate <= STOP1;
                else
                    cstate <= ACK4;
            end
            STOP1    :
            begin
                if (`SCL_LOW) begin
                    sda_link  <= 1'b1;
                    sda_r     <= 1'b0;
                    cstate    <= STOP1;
                end
                else if (`SCL_HIG) begin
                    sda_r     <= 1'b1;
                    cstate    <= STOP2;
                end
                else cstate   <= STOP1;
            end
            STOP2    :
            begin
                if (`SCL_NEG) begin
                    sda_link  <= 1'b0;
                    scl_link  <= 1'b0;
                end
                else if (cnt_5ms==20'h3fffc) begin
                    cnt_5ms   <= 20'h00000;
                    iic_done  <= 1;
                    cstate    <= IDLE;
                end
                else begin
                    cstate    <= STOP2;
                    cnt_5ms   <= cnt_5ms + 1'b1;
                end
            end
            default: cstate <= IDLE;
        endcase
end
```

The entire process is implemented using a state machine. When reset, it is idle state, while data line *sda_r* is pulled high, clock and data are both valid, i.e. *scl_link*, *sda_link* are high; counter *num* is cleared and *read_data* is 0. 5ms delay counter is cleared, IIC transmission end signal *Iic_done* is low thus invalid.

（1） IDLE state: When receiving the read enable or write enable signal *sw1_r* || *sw2_r*, assign the write control word to the intermediate variable *db_r* <= `*DEVICE_WRITE*, and

jump to the start state START1;

（2） START1 state: pull the data line low when the clock signal is high, generating the start signal of IIC transmission, and jump to the device address state ADD1;

（3） Device address status ADD1: After the write control word (device address plus one '0' bit) is transmitted according to MSB (high order priority), the *sda_link* is pulled low causing data bus in a high impedance state, and jump to the first response state ACK1, waiting for the response signal from the slave (EEPROM).

（4） The first response status ACK1: If the data line is pulled low, it proves that the slave receives the data normally, otherwise the data is not written into EEPROM, and then the rewriting or stopping is decided by the user. There is no temporary judgment and processing here, jump directly to the write register address state ADD2, and assign the address *BYTE_ADDR* written to the intermediate variable (this experiment writes the data into the third register, i.e. *BYTE_ADDR* = 0000_0011)

（5） Register address status ADD2: Same as (3), it transfers register address to slave and jump to second response status ACK2

（6） The second response state ACK2: At this time, it is urgent to judge. If it is the write state *sw1*, it jumps to the data transfer state DATA, and at the same time assigns the written data to the intermediate variable. If it is the read state *sw2*, it jumps to the second start state START2 and assign the read control word to the intermediate variable.

（7） The second start state START2: it produces a start signal identical to (2) and jumps to the read register address state ADD3

（8） Read register address status ADD3: it jumps to the third response status ACK3S after the transfer of the register address that needs to be read out

（9） The third response state ACK3: it jumps directly to the data transfer state DATA. In the read state, the data to be read is directly read out immediately following the register address.

（10） Data transfer status DATA: it needs to be judged here. If it is the read status, the data will be directly output. If it is the write status, the data to be written will be transferred to the data line SDA. Both states need to jump to the fourth response state. ACK4

（11） The fourth response status ACK4: it direct jumps to stop transmission STOP1

（12） Stop transmission STOP1: it pulls up data line when the clock line is high, generating a stop signal, and jumps to the transfer completion status STOP2

（13） Transfer completion status STOP2: it releases all clock lines and data lines, and after a 5ms delay, returns to the IDLE state to wait for the next transfer instruction. This is because EEPROM stipulates that the interval between two consecutive read and write operations must not be less than 5ms.

11.4 Experiment Verification

The first step: pin assignments

Table 11.1 IIC protocol transmission experiment pin mapping

Signal Name	Network Label	FPGA Pin	Port Description
clk	CLK_50M	G21	System clock 50 MHz

rst_n	PB3	Y6	Reset
sm_db[0]	SEG_PA	B15	Segment a
sm_db [1]	SEG_PB	E14	Segment b
sm_db [2]	SEG_PC	D15	Segment c
sm_db [3]	SEG_PD	C15	Segment d
sm_db [4]	SEG_PE	F13	Segment e
sm_db [5]	SEG_PF	E11	Segment f
sm_db [6]	SEG_PG	B16	Segment g
sm_db [7]	SEG_DP	A16	Segment h
sm_cs1_n	SEG_3V3_D1	D19	Segment 1
sm_cs2_n	SEG_3V3_D0	F14	Segment 0
data[0]	SW0	U11	Switch input
data[1]	SW1	V11	Switch input
data[2]	SW2	U10	Switch input
data[3]	SW3	V10	Switch input
data[4]	SW4	V9	Switch input
data[5]	SW5	W8	Switch input
data[6]	SW6	Y8	Switch input
data[7]	SW7	W6	Switch input
sw1	PB4	AB4	Write EEPROM button
sw2	PB6	AA4	Read EEPROM button
scl	I2C_SCL	D13	EEPROM clock line
sda	I2C_SDA	C13	EEPROM data line

Step 2: board downloading verification

After the pin assignment is completed, the compilation is performed, and the board is verified after passing.

After the program is downloaded to the board, press the LEFT key to write the 8-bit value represented by SW7~SW0 to EEPROM. Then press the RIGHT key to read the value from the write position. Observe the consistency between the value displayed on the segment display on the experiment board and the value written in the 8'h03 register of the EEPROM address (SW7~SW0) (this experiment writes 8'h34). The read value is displayed on the segment display. The experimental phenomenon is shown in Figure 11.3.

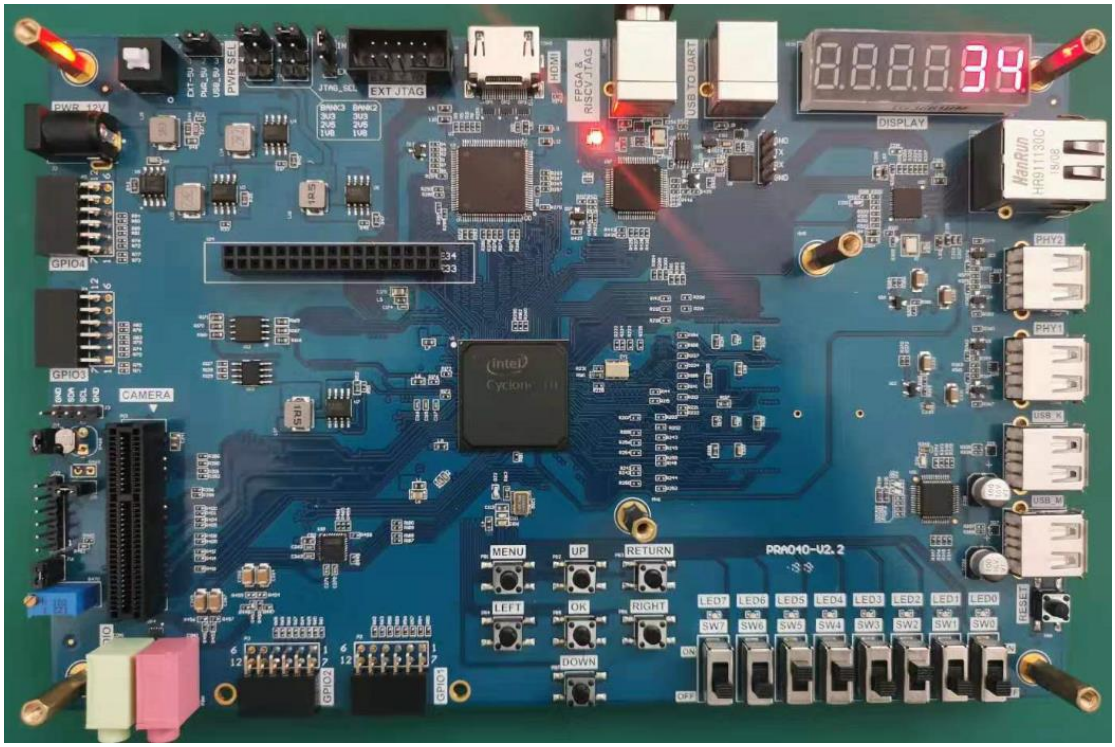


Figure 11.3 Observe experiment result

Experiment 12 AD,DA Experiment

12.1 Experiment Objective

Since in the real world, all naturally occurring signals are analog signals, and all that are read and processed in actual engineering are digital signals. There is a process of mutual conversion between natural and industrial signals (digital-to-analog conversion: DAC, analog-to-digital conversion: ADC). The purpose of this experiment is as follows:

（1） Learn about the theory of AD conversion
（2） Review the knowledge of the IIC protocol learned in the previous experiment and write the data into PCF8591 on the development board.
（3） Read the value of AD acquisition from PCF8591, and convert the value obtained into actual value, display it with segment display

12.2 Experiment Implement

（1） The ADC port of the chip is used for analog-to-digital conversion. The chip is correctly configured. Three variable (potentiometer, photoresistor, thermistor) voltages on the development board are collected, and the collected voltage value is displayed through the segment display.
（2） Board downloading verification, compared with resistance characteristics, verify the correctness of the results

12.3 Experiment

12.3.1 Introduction to AD Conversion Chip PCF8591

The PCF8591 is a monolithically integrated, individually powered, low power consuming, 8-bit CMOS data acquisition device. The PCF8591 has four analog inputs, one analog output, and one serial IIC bus interface. The three address pins A0, A1 and A2 of the PCF8591 can be used for hardware address programming. The address, control signals and data signals of the input and output on the PCF8591 device are transmitted serially via the two-wire bidirectional IIC bus. Please refer to the previous experiment 11 for the contents of the IIC bus protocol. After the device address information and the read/write control word are sent, the control word information is sent.

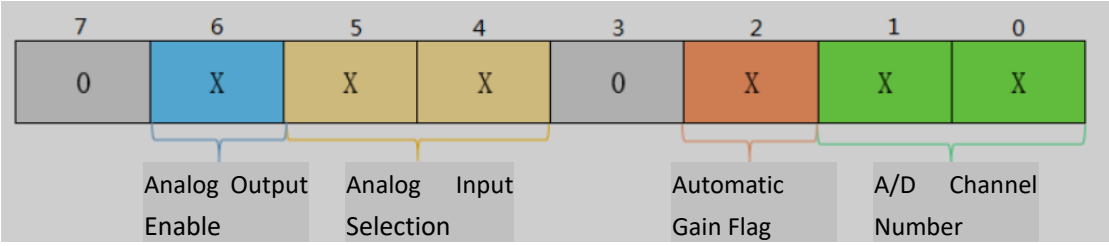


Figure 12.1 PCF8591 address

The specific control word information is shown in Figure 12.1. Digit 1 - digit 0 is used for four channel settings, digit 2 is for automatic gain selection, '1' is valid. Digit 5 - digit 4 determines analog input selection. Digit 6 is analog output enable. Digit 7 and digit 3 are reserved to be '0'. The second byte sent to PCF8951 is stored in the control register to control the device functionality. The upper nibble of the control register is used to allow the analog output to be programmed as a single-ended or differential input. The lower nibble selects an analog input channel defined by the high nibble. If the auto increment flag is set to 1, the channel number will be automatically incremented after each A/D conversion.

In this experiment, the input channel is selected as the AD acquisition input channel by using the DIP switch (SW1, SW0). The specific channel information is shown in Table 12.1. The control information is configured as 8'h40, which is the analog output, and defaults to "00" channels, which means that the photoresistor voltage value is displayed by default.

Table 12.1 Channel information

SW1，SW0	Channel Selection	Acquisition Object
00	0	Voltage of photoresistor
01	1	Voltage of thermistor
10	2	Voltage of potentiometer

12.3.2 Hardware Design

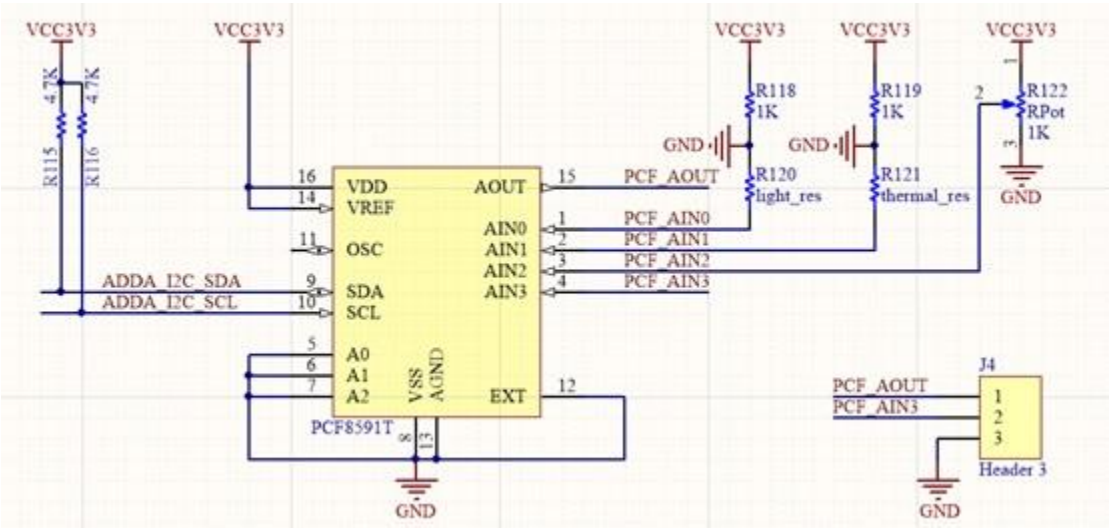


Figure 12.2 Schematics of the AD/DA converter

。The schematics of AD/DA conversion using PCF8591 is shown in Figure 12.2. The IIC bus goes through two pull-up resistors and pulls high when not working. *A0, A1, A2* are grounded, so the device address is 7'b1010000, the analog input channel *AIN0* is connected to the photoresistor, *AIN1* is connected to the thermistor, and *AIN3* is connected to the potentiometer.

When the channel is selected, FPGA will read the value in PCF8591 through the data bus *ADDA_I2C_SLC* for processing.

Introduction to the Program

This experiment also uses the IIC bus to control the PCF8951 chip, so the program is basically the same as Experiment 11. Only parts difference from Experiemnt 11 are indicated here.

```
reg         [7:0]       read_data_temp [15:0];
reg         [11:0]      dis_data_temp;
always @ (posedge clk)
   dis_data_temp <= read_data_temp[0] + read_data_temp[1] + read_data_temp[2]
                        + read_data_temp[3] + read_data_temp[4] + read_data_temp[5]
                        + read_data_temp[6] + read_data_temp[7] + read_data_temp[8]
                        + read_data_temp[9] + read_data_temp[10] + read_data_temp[11]
                        + read_data_temp[12] + read_data_temp[13] + read_data_temp[14]
                        + read_data_temp[15];
always @ (posedge clk )
      dis_data <= dis_data_temp >> 4;
integer     i;
always @ (posedge clk or negedge rst_n)
begin
      if (!rst_n) begin
            for (i=0;i<16;i=i+1)
                  read_data_temp[i] <= 8'h00;
      end
      else if (iic_done) begin
            for (i=0;i<15;i=i+1)
                  read_data_temp[i+1] <= read_data_temp[i];
                  read_data_temp[0] <= read_data ;
            end
      else begin
            for (i=0;i<16;i=i+1)
                  read_data_temp[i] <= read_data_temp[i];
      end
end
```

The role of this part is that when the chip continuously collects the voltage value across the resistor, due to a series of unstable factors, the voltage value will be unstable, so the output value will have a large error, so 16 sets of data is collected each time, then gets averaged, and the result is output as the voltage value across the resistor at this time. Then, by the change of the voltage value, it is possible to judge the regular pattern. Such as photoresistor, the greater the light intensity, the smaller the voltage value, the smaller the resistance value, satisfying the photoresistor characteristics; the higher the thermistor temperature, the smaller the voltage

value, the smaller the resistance, satisfying the photoresistor characteristics; the potentiometer rotates clockwise, and the voltage increases, the resistance increases; counterclockwise rotating decrases the voltage, and the resistance decreases.

The maximum output of the AD chip is an 8-bit digital quantity, but in fact it is not the required voltage value. It quantifies the voltage value of the range into 256 portions (8-bit binary number can represent 256 decimal numbers), so further calculations and conversions needs to be applide when displaying on the segment display.

```
    parameter       V_REF = 12'd3300;
    reg             [19:0]      num_t;
    reg             [19:0]      num1;
    wire            [3:0]       data0;
    wire            [3:0]       data1;
    wire            [3:0]       data2;
    wire            [3:0]       data3;
    wire            [3:0]       data4;
    wire            [3:0]       data5;
    assign    data5 = num1 / 17'd100000;
    assign    data4 = num1 / 14'd10000 % 4'd10;
    assign    data3 = num1 / 10'd1000 % 4'd10 ;
    assign    data2 = num1 / 7'd100 % 4'd10 ;
    assign    data1 = num1 / 4'd10 % 4'd10 ;
    assign    data0 = num1 % 4'd10;

always @ (posedge clk)
   num_t <= V_REF * dis_data;

always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) begin
        num1 <= 20'd0;
    end
    else
        num1 <= num_t >> 4'd8;
end
```

VCC is 3.3V, so the maximum resistance voltage is 3.3V. The 8-bit data *dis_data* is multiplied by 3300 and assigned to *numt* by 1000 times, which is convenient for display and observation. The *numt* is further reduced by 256 times (left shifting 8 bits) to *num1*, corresponding to 256 quantitized portions of PCF8951. *num1* at this time is 1000 times the voltage value of two ends of the resistor. Display each digit on the segment display, in the order of high to low (data5 to data0) and add the decimal point (data3) to digit of thousands . At this time, the value displayed by the segment display is the voltage across the resistor value, and correct to 3 decimal places.

12.4 Experiment Verification

The first step: assign the pin

Table 12.2 AD conversion experiment pin mapping

Signal Name	Network Label	FPGA Pin	Port Description
clk	CLK_50M	G21	System clock 50 MHz
rst_n	PB3	Y6	Reset
sm_db[0]	SEG_PA	B15	Segment a
sm_db [1]	SEG_PB	E14	Segment b
sm_db [2]	SEG_PC	D15	Segment c
sm_db [3]	SEG_PD	C15	Segment d
sm_db [4]	SEG_PE	F13	Segment e
sm_db [5]	SEG_PF	E11	Segment f
sm_db [6]	SEG_PG	B16	Segment g
sm_db [7]	SEG_DP	A16	Segment h
sel[0]	SEG_3V3_D0	F14	Bit selection 0
sel[1]	SEG_3V3_D1	D19	Bit selection 1
sel[2]	SEG_3V3_D2	E15	Bit selection 2
sel[3]	SEG_3V3_D3	E13	Bit selection 3
sel[4]	SEG_3V3_D4	F11	Bit selection 4
sel[5]	SEG_3V3_D5	R12	Bit selection 5
data[0]	SW0	U11	Swicth input
data[1]	SW1	V11	Swicth input
data[2]	SW2	U10	Swicth input
data[3]	SW3	V10	Swicth input
data[4]	SW4	V9	Swicth input
data[5]	SW5	W8	Swicth input
data[6]	SW6	Y8	Swicth input
data[7]	SW7	W6	Swicth input
scl	ADDA_I2C_SCL	C20	PCF8591 clock line
sda	ADDA_I2C_SDA	D20	PCF8591 data line

Step 2: board verification

After the pin assignment is completed, the compilation is performed, and board downloading is verified after passing.

Under the default state, that is, the channel selection is “00”, the segment display shows the current ambient brightness state, the voltage value across the photoresistor is 2.010V, as shown in Figure 12.3.

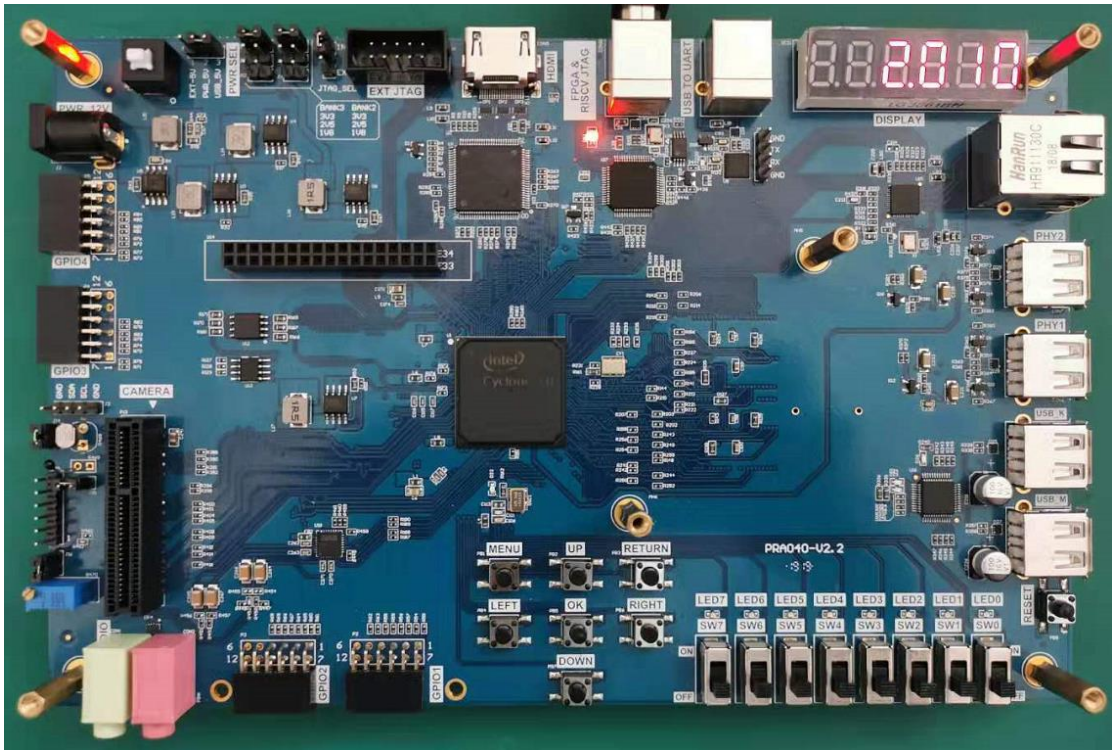


Figure 12.3 Photoresist test phenomenon

When the channel selection is “01”, the segment display shows the current ambient temperature, the voltage across the thermistor is 2.926V, as shown in Figure 12.4.

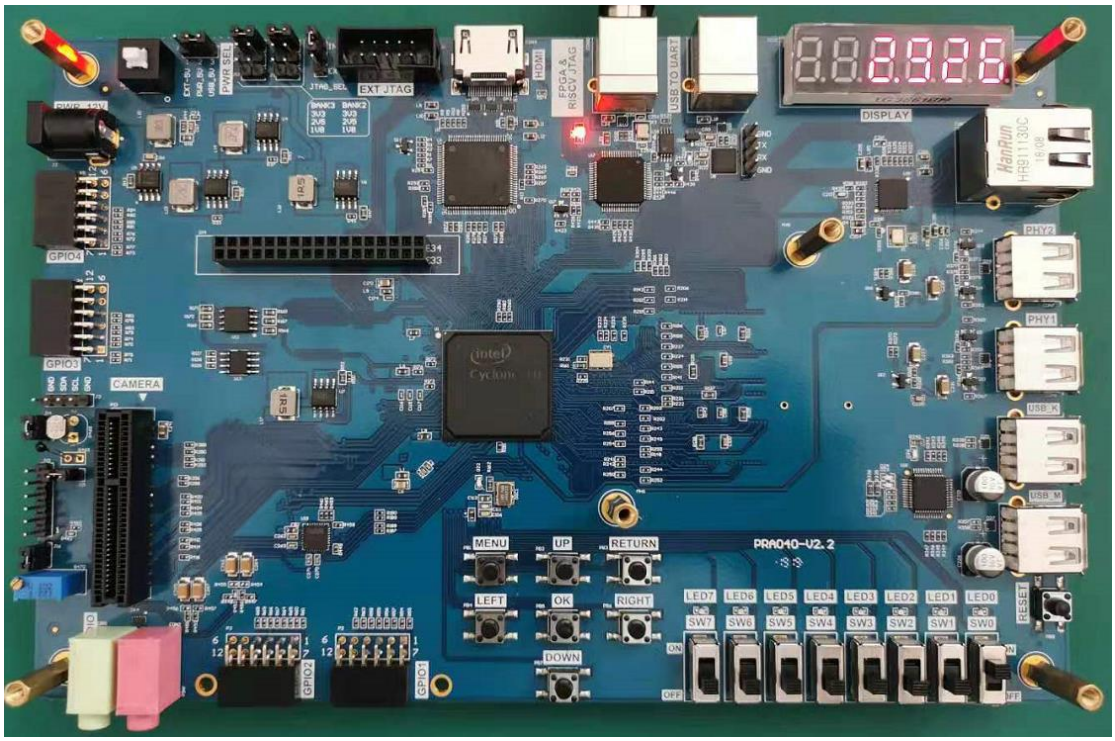


Figure 12.4 Thermistor experiment phenomenon

When the channel is selected as “10”, the segment display shows the current resistance value, and the voltage across the potentiometer is 1.456 V, as shown in Figure 12.5.

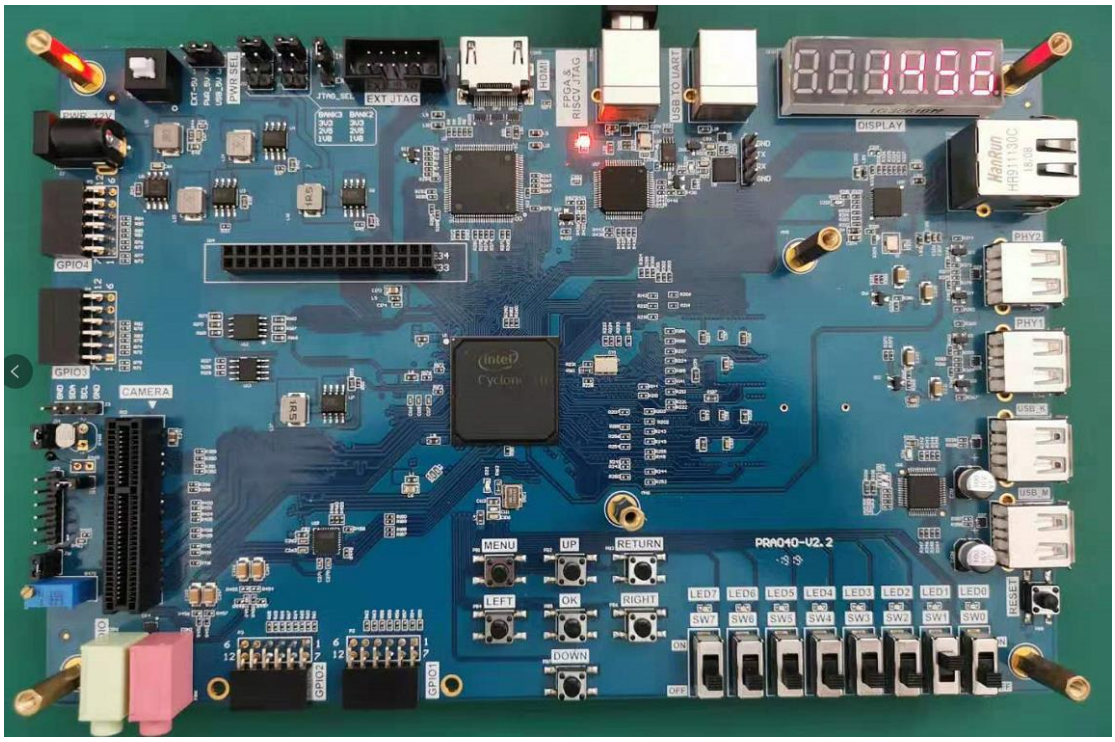


Figure 12.5 Potentiometer experiment phenomenon

Experiment 13 HDMI Display

13.1 Experiment Objetive

（1） Review IIC protocol
（2） Review EEPROM read and write
（3） Learn HDMI principle

13.2 Experiment Implement

Display different image content on the screen through the HDMI.

13.3 Experiment

13.3.1 Introduction to HDMI and ADV7511 Chip

Image display processing has always been the focus of FPGA research. At present, the image display mode is also developing. The image display interface is also gradually transitioning from the old VGA interface to the new DVI or HDMI interface. HDMI (High Definition Multimedia Interface) is a digital video/audio interface technology. It is a dedicated digital interface for image transmission. It can transmit audio and video signals at the same time.

The ADV7511 is a chip that converts FPGA digital signal to HDMI signal following VESA standard.For more details, see the related chip manual. Among them, “ADV7511 Programming Guide” and “ADV7511 Hardware Users Guide” are the most important. The registers of the ADV7511 can be configured by referring those documents.

ADV7511 Register Configuration Description: The bus inputs D0-D3, D12-D15, and D24-D27 of the ADV7511 have no input, and each bit of data is in 8-bit mode. Directly set 0x15 [3:0]) 0x0 data, 0x16 [3:2] data does not need to be set for its mode. Set [5:4] of 0X16 to 11for 8-bit data and keep the default values for the other digits. 0x17[1] refers to the ratio of the length to the width of the image. It can be set to 0 or 1. The actual LCD screen will not change according to the data, but will automatically stretch the full screen mode according to the LCD's own settings. 0x18[7] is the way to start the color range stretching. The design is that RGB maps directly to RGB, so it can be disabled directly. 0X18[6:5] is also invalid at this time. 0XAF[1] is to set HDMI or DVI mode, the most direct point of HDMI than DVI is that HDMI can send digital audio data and encrypted data content. This experiment only needs to Display the picture, and it can be set directly to DVI mode. Set 0XAF[7] to 0 to turn off HDMI encryption. Due to GCCD, deep color encryption data is not applicable, so the GC option is turned off. 0x4c register does not need to be set as well. Other sound data setting can be ignored here for DVI output mode. After writing these registers, the image can be displayed successfully.

13.3.2 Hardware Design

The onboard HDMI module consists of an HDMI interface and an ADV7511 chip. The physical photo is shown in Figure 13.1. The schematics is shown in Figure 13.2.

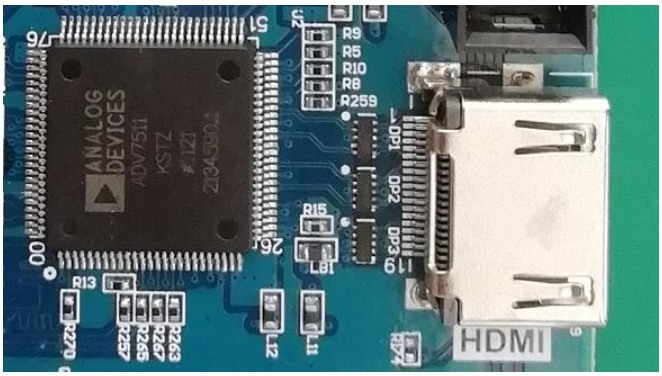


Figure 13.1 HDMI interface and ADV7511 chip physical photo

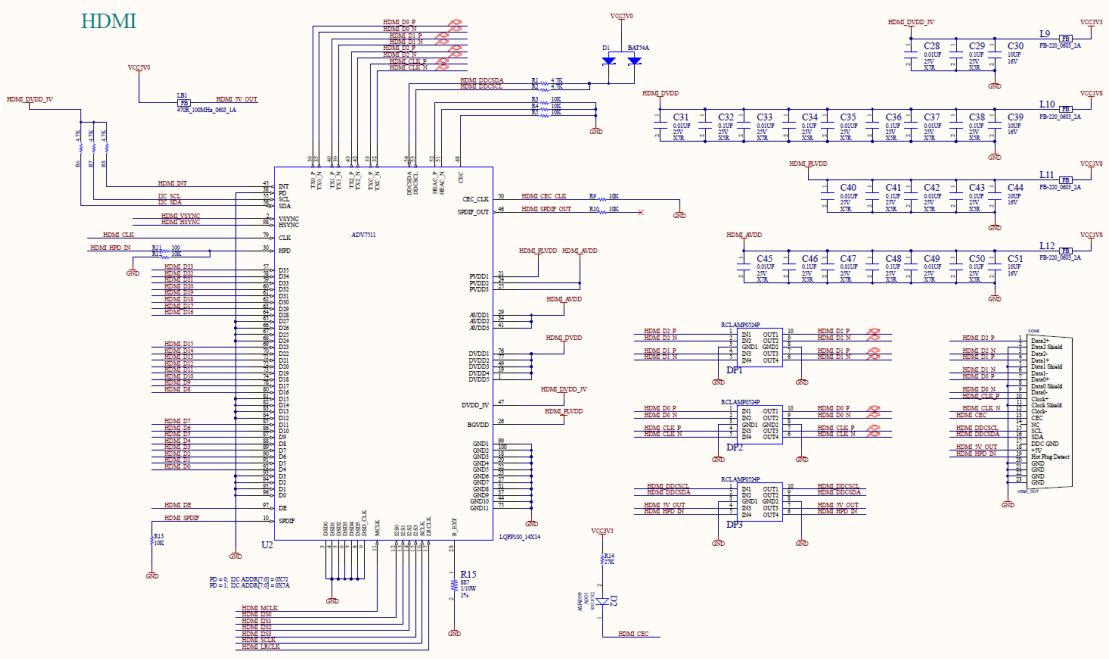


Figure 13.2 Schematics of HDMI

ADV7511 chip is set through the IIC bus, and send the picture information to be displayed to the chip through HDMI_D0 to HDMI_D23, and control signals HDMI_HSYNC and HDMI_VSYNC and the clock signal HDMI_CLK, which are transmitted to the PC through the HDMI interface after being processed internally by the chip.

13.3.3 Introduction to the Program

The configuration part of the ADV7511 chip is carried out using the IIC protocol, with reference to Experiment 11 and Experiment 12. A brief introduction to the data processing section is now

available.

```
module hdmi_test (
    input               rst_n,
    input               clk_in,
    input               key1,
    output              vga_hs,
    output              vga_vs,
    output      [7:0]   vga_r,
    output      [7:0]   vga_g,
    output      [7:0]   vga_b,
    output              vga_clk,
    inout               scl,
    inout               sda,
    output              en
);
```

The FPGA configures the ADV7511 chip through the IIC bus (clock line *scl*, data line *sda*). After the configuration is completed, the output image information needs to be determined. Taking the 1080P (1920*1080) image format as an example, it outputs data signal *rgb_r* (red component), *rgb_g* (green component), *rgb_b* (blue component), a line sync signal *rgb_hs*, a field sync signal *rgb_vs*, and a clock *rgb_clk* signal. Each pixel is formed by a combination of three color components. Each row of 1920 pixels is filled with color information in a certain order (from left to right), and begin to fill the next line after completing one line, in a certain order (from top to bottom)to finish 1080 lines, so that one frame of image information is completed. The image information of each frame is determined by this horizontal and vertical scanning, and then transmitted to the ADV7511 for processing. The timing diagram of the horizontal and vertical scan is shown in Figure 13.3,Figure 13.4.

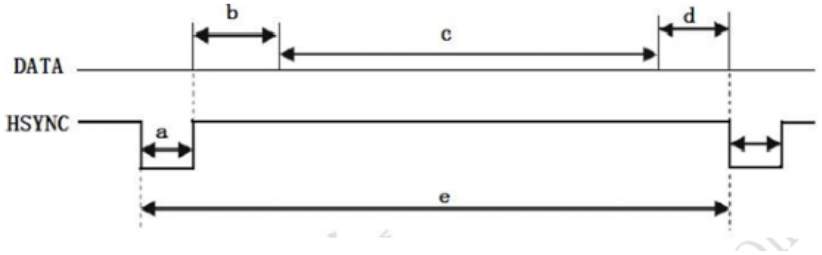


Figure 13.3 Horizontal synchronization

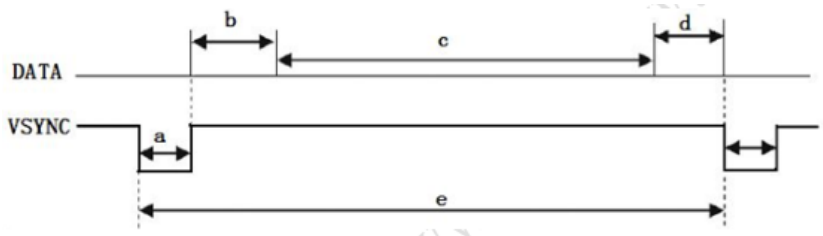


Figure 13.4 Vertical synchronization

The second step: data definition of 1080p image timing generation

```
Horizontal line scan parameter setting 1920*1080 60 Hz      clock 130 MHz
parameter LinePeriod    = 2000;              // Line period
parameter H_SyncPulse   = 12;                // Line sync pulse (Sync a)
parameter H_BackPorch   = 40;                // Display back edge (Back porch b)
parameter H_ActivePix   = 1920;             // Display interval c
parameter H_FrontPorch= 28;                  // Display front edge (Front porch d)
parameter Hde_start     = 52;
parameter Hde_end       = 1972;

Vertical scan parameter setting 1920*1080 60Hz
parameter FramePeriod = 1105;              //Frame period
parameter V_SyncPulse  = 4;                // Vertical sync pulse (Sync o)
parameter V_BackPorch = 18;                // Display trailing edge (Back porch p)
parameter V_ActivePix   = 1080;            //Display interval q
parameter V_FrontPorch= 3;                 // Display front edge (Front porch r)
parameter Vde_start     = 22;
parameter Vde_end       = 1102;

reg             [12:0]    x_cnt;
reg             [10:0]    y_cnt;
reg             [23:0]    grid_data_1;
reg             [23:0]    grid_data_2;
reg             [23:0]    bar_data;
reg             [3:0]     rgb_dis_mode;
reg             [7:0 ]    rgb_r_reg;
reg             [7:0]     rgb_g_reg;
reg             [7:0]     rgb_b_reg;
reg                       hsync_r;
reg                       vsync_r;
reg                       hsync_de;
reg                       vsync_de;
reg             [15:0]    key1_counter;            //Button

wire                      locked;
reg                       rst;
wire            [12:0]    bar_interval;

assign bar_interval = H_ActivePix[15: 3];          //Color strip width
```

The third step: Generate display content

```
always @ (posedge rgb_clk)
begin
    if (rst)
```

```
            hsync_r   <= 1'b1;
      else if (x_cnt == LinePeriod)
            hsync_r <= 1'b0;
      else if (x_cnt == H_SyncPulse)
            hsync_r <= 1'b1;
      if (rst)
            hsync_de <= 1'b0;
      else if (x_cnt == Hde_start)
            hsync_de <= 1'b1;
      else if (x_cnt == Hde_end)
            hsync_de <= 1'b0;
end

always @ (posedge vga_clk)
begin
      if (rst)
            y_cnt <= 1'b1;
      else if (x_cnt == LinePeriod) begin
            if (y_cnt == FramePeriod)
                  y_cnt <= 1'b1;
            else
                  y_cnt <= y_cnt + 1'b1;
      end
end

always @ (posedge rgb_clk)
begin
      if (rst)
            vsync_r <= 1'b1;
      else if ((y_cnt == FramePeriod) &(x_cnt == LinePeriod))
            vsync_r <= 1'b0;
      else if ((y_cnt == V_SyncPulse) &(x_cnt == LinePeriod))
            vsync_r <= 1'b1;
      if (rst)
            vsync_de <= 1'b0;
      else if ((y_cnt == Vde_start) & (x_cnt == LinePeriod))
            vsync_de <= 1'b1;
      else if ((y_cnt == Vde_end) & (x_cnt == LinePeriod))
            vsync_de <= 1'b0;
end

      assign en = hsync_de & vsync_de;

always @(posedge rgb_clk)
```

```
begin
      if ((x_cnt[4]==1'b1) ^ (y_cnt[4]==1'b1))
            grid_data_1 <= 24'h000000;
      else
            grid_data_1<= 24'hffffff;
      if ((x_cnt[6] == 1'b1) ^ (y_cnt[6] == 1'b1))
            grid_data_2 <=24'h000000;
      else
            grid_data_2 <=24'hffffff;
end

always @ (posedge    rgb _clk)
begin
      if (x_cnt==Hde_start)
            bar_data <= 24'hff0000;                          // Red strip
      else if (x_cnt == Hde_start + bar_interval)
            bar_data <= 24'h00ff00;                          // Green strip
      else if (x_cnt == Hde_start + bar_interval*2)
            bar_data <= 24'h0000ff;                          // Blue strip
      else if (x_cnt == Hde_start + bar_interval*3)
            bar_data <= 24'hff00ff;                          // Purple strip
      else if (x_cnt == Hde_start + bar_interval*4)
            bar_data <= 24'hffff00;                          // Yellow strip
      else if (x_cnt == Hde_start + bar_interval*5)
            bar_data <= 24'h00ffff;                          // Light blue strip
      else if (x_cnt == Hde_start + bar_interval*6)
            bar_data <= 24'hffffff;                          // White strip
      else if (x_cnt == Hde_start + bar_interval*7)
            bar_data <= 24'hff8000;                          // Orange strip
      else if (x_cnt == Hde_start + bar_interval*8)
            bar_data <= 24'h000000;                          //Black strip
end

always @ (posedge vga_clk)
begin
      if (rst) begin
            rgb_r_reg <= 0;
            rgb_g_reg <= 0;
            rgb_b_reg <= 0;
      end
      else case (vga_dis_mode)
            4'b0000   :                 // Display all black
            begin
                  rgb_r_reg <= 0;
```

```
            rgb_g_reg <= 0;
            rgb_b_reg <= 0;
        end
        4'b0001   :                 // Display all white
        begin
            rgb_r_reg <= 8'hff;
            rgb_g_reg <= 8'hff;
            rgb_b_reg <= 8'hff;
        end
        4'b0010 :                   // Display all red
        begin
            rgb_r_reg <= 8'hff;
            rgb_g_reg <= 0;
            rgb_b_reg <= 0;
        end
        4'b0011   :                 // Display all green
        begin
            rgb_r_reg <= 0;
            rgb_g_reg <= 8'hff;
            rgb_b_reg <= 0;
        end
        4'b0100   :                 // Display all blue
        begin
            rgb_r_reg <= 0;
            rgb_g_reg <= 0;
            rgb_b_reg <= 8'hff;
        end
        4'b0101   :                 // Display square 1
        begin
            rgb_r_reg <= grid_data_1[23:16];
            rgb_g_reg <= grid_data_1[15:8];
            rgb_b_reg <= grid_data_1[7:0];
        end
        4'b0110   :                 // Display square 2
        begin
            rgb_r_reg <= grid_data_2[23:16];
            rgb_g_reg <= grid_data_2[15:8];
            rgb_b_reg <= grid_data_2[7:0];
        end
        4'b0111 :                   // Display horizontal gradient
        begin
            rgb_r_reg <= x_cnt[10:3];
            rgb_g_reg <= x_cnt[10:3];
            rgb_b_reg <= x_cnt[10:3];
```

```
            end
            4'b1000  :                  // Display vertical gradient
            begin
                rgb_r_reg <= y_cnt[10:3];
                rgb_g_reg <= y_cnt[10:3];
                rgb_b_reg <= y_cnt[10:3];
            end
            4'b1001 :                   // Display red horizontal gradient
            begin
                rgb_r_reg <= x_cnt[10:3];
                rgb_g_reg <= 0;
                rgb_b_reg <= 0;
            end
            4'b1010  :                  // Display green horizontal gradient
            begin
                rgb_r_reg <= 0;
                rgb_g_reg <= x_cnt[10:3];
                rgb_b_reg <= 0;
            end
            4'b1011  :                  // Display blue horizontal gradient
            begin
                rgb_r_reg <= 0;
                rgb_g_reg <= 0;
                rgb_b_reg <= x_cnt[10:3];
            end
            4'b1100  :                  // Display colorful strips
            begin
                rgb_r_reg <= bar_data[23:16];
                rgb_g_reg <= bar_data[15:8];
                rgb_b_reg <= bar_data[7:0];
            end
            default   :                 // Display all white
            begin
                rgb_r_reg <= 8'hff;
                rgb_g_reg <= 8'hff;
                rgb_b_reg <= 8'hff;
            end
        endcase
end
        assign rgb_hs  = hsync_r;
        assign rgb_vs  = vsync_r;
        assign rgb_r   = (hsync_de & vsync_de) ? rgb_r_reg : 8'h00;
        assign rgb_g   = (hsync_de & vsync_de) ? rgb_g_reg : 8'b00;
        assign rgb_b   = (hsync_de & vsync_de) ? rgb_b_reg : 8'h00;
```

```
always @(posedge rgb_clk)
begin
    if (key1 == 1'b1)
        key1_counter <= 0;
    else if ((key1 == 1'b0) & (key1_counter <= 16'd130000))
        key1_counter <= key1_counter + 1'b1;
        if (key1_counter == 16'h129999) begin
            if(rgb_dis_mode == 4'b1100)
                rgb_dis_mode <= 4'b0000;
            else
                rgb_dis_mode <= rgb_dis_mode + 1'b1;
        end
end
```

When the button is pressed, a key1 signal will be input, and the content displayed on the screen will change according to the change of *vga_dis_mode*, and the corresponding picture content will be displayed.

13.4 Experiment Verification

The first step: pin assignment

Table 13.1 HDMI Experiment Pin Mapping

Signal Name	Network Label	FPGA Pin	Port Description
clk	CLK_50M	G21	System clock 50 MHz
rst_n	PB3	Y6	Reset
en	HDMI_R_DE	A8	Enable
scl	I2C_SCL	D13	IIC clock line
sda	I2C_SDA	C13	IIC data line
key1	PB2	V5	Switch display content
vga_clk	HDMI_R_CLK	E5	HDMI clock
vga_hs	HDMI_R_HS	B9	Horizontal sync signal
vg_vs	HDMI_R_VS	A9	Vertical sync signal
vga_b[0]	HDMI_R_D0	A7	Blue output
vga_b[1]	HDMI_R_D1	B8	
vga_b[2]	HDMI_R_D2	E9	
vga_b[3]	HDMI_R_D3	B7	
vga_b[4]	HDMI_R_D4	C8	
vga_b[5]	HDMI_R_D5	C6	
vga_b[6]	HDMI_R_D6	F8	
vga_b[7]	HDMI_R_D7	B6	
vga_g[0]	HDMI_R_D8	A5	

vga_g[1]	HDMI_R_D9	C7	Green output
vga_g[2]	HDMI_R_D10	D7	
vga_g[3]	HDMI_R_D11	B5	
vga_g[4]	HDMI_R_D12	C6	
vga_g[5]	HDMI_R_D13	A4	
vga_g[6]	HDMI_R_D14	D6	
vga_g[7]	HDMI_R_D15	B4	
vga_r[0]	HDMI_R_D16	E7	Red output
vga_r[1]	HDMI_R_D17	A3	
vga_r[2]	HDMI_R_D18	C4	
vga_r[3]	HDMI_R_D19	B3	
vga_r[4]	HDMI_R_D20	C3	
vga_r[5]	HDMI_R_D21	F7	
vga_r[6]	HDMI_R_D22	F9	
vga_r[7]	HDMI_R_D23	G7	

The second step: board verification

After the pin assignment is completed, the compilation is performed, and the development board is programmed.

Press the push button and the display content changes accordingly. The experimental phenomenon is shown in the figure below (only a few are listed).

Figure 13.5 HDMI display (all white)

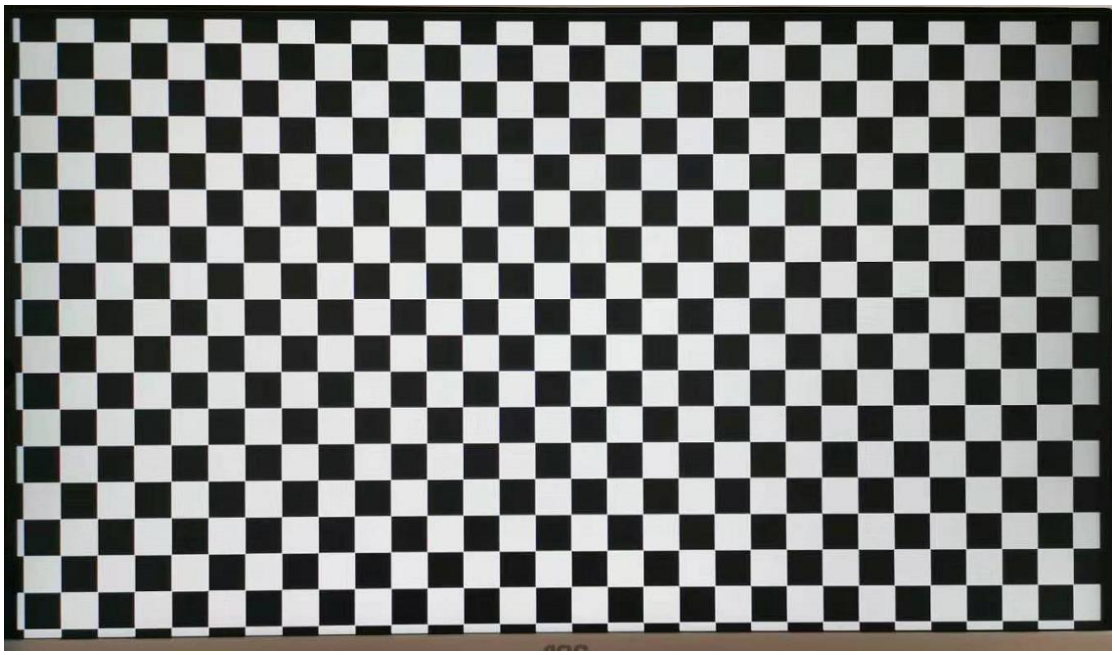


Figure 13.6 HDMI display (square)

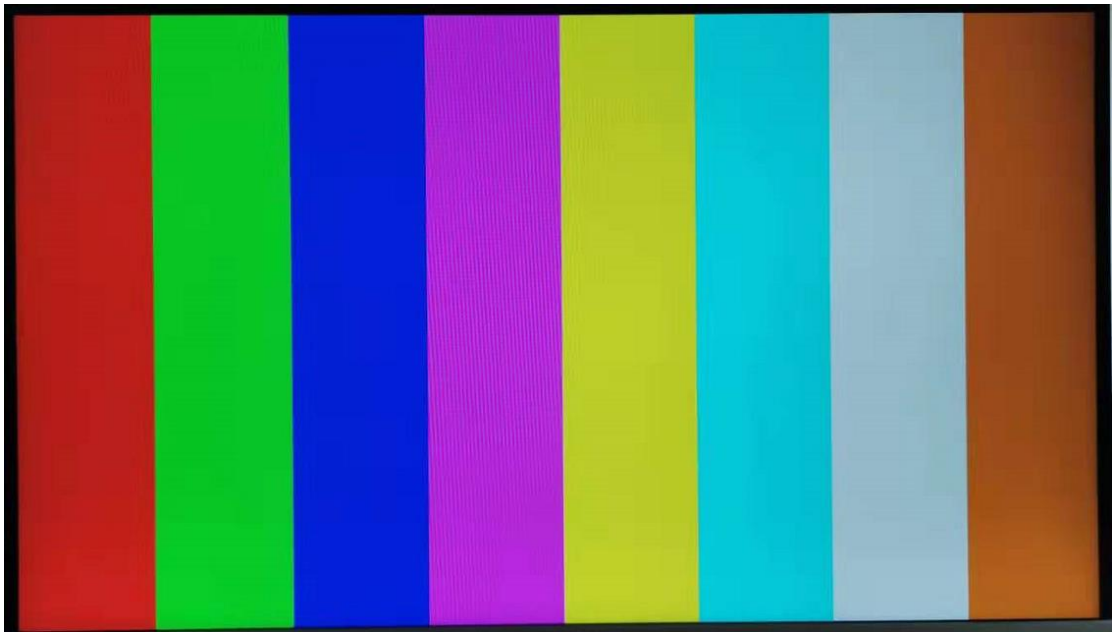


Figure 13.7 HDMI display (color strip)

Experiment 14 Ethernet

14.1 Experiment Objective

（1） Understand what Ethernet is and how it works

（2） Familiar with the relationship between different interface types (MII, GMII, RGMII) and their advantages and disadvantages (FII-PRA040 uses RGMII)

（3） Combine the development board to complete the transmission and reception of data and verify it

14.2 Experiment Implement

（1） Perform a loopback test to check if the hardware is working properly.

（2） Perform data receiving verification

（3） Perform data transmission verification

14.3 Experiment

14.3.1 Introduction to Experiment Principle

Ethernet is a baseband LAN technology. Ethernet communication is a communication method that uses coaxial cable as a network media and uses carrier multi-access and collision detection mechanisms. The data transmission rate reaches 1 Gbit/s, which can satisfy the need for data transfer of non-persistent networks. As an interconnected interface, the Ethernet interface is very widely used. There are many types of Gigabit Ethernet MII interfaces, and GMII and RGMII are commonly used.

MII interface has a total of 16 lines. See Fig 14. 1.

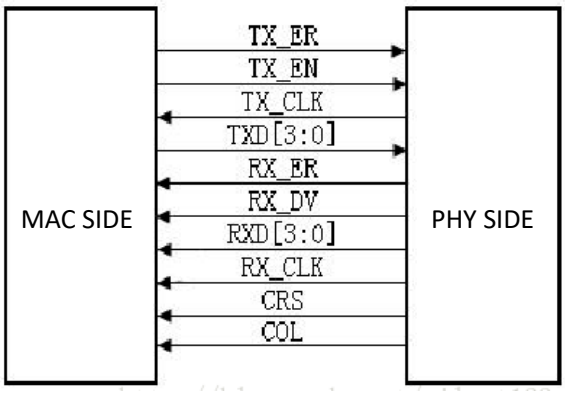


Figure 14.1 MII interface

RXD(Receive Data)[3:0]: data reception signal, a total of 4 signal lines;

TX_ER(Transmit Error): Send data error prompt signal, synchronized to *TX_CLK*, active high,

indicating that the data transmitted during *TX_ER* validity period is invalid. For 10Mbps rate, *TX_ER* does not work;

RX_ER(Receive Error): Receive data error prompt signal, synchronized to *RX_CLK*, active high, indicating that the data transmitted during the valid period of *RX_ER* is invalid. For 10 Mbps speed, *RX_ER* does not work;

TX_EN(Transmit Enable): Send enable signal, only the data transmitted during the valid period of *TX_EN* is valid；

RX_DV(Reveive Data Valid): Receive data valid signal, the action type is *TX_EN* of the transmission channel;

TX_CLK: Transmit reference clock, the clock frequency is 25 MHz at 100 Mbps, and the clock frequency is 2.5 MHz at 10 Mbps. Note that the direction of *TX_CLK* clock is from the PHY side to the MAC side, so this clock is provided by the PHY;

RX_CLK: Receive data reference clock, the clock frequency is 25 MHz at 100 Mbps, and the clock frequency is 2.5 MHz at 10 Mbps. *RX_CLK* is also provided by the PHY side;

CRS: Carrier Sense, carrier detect signal, does not need to synchronize with the reference clock. As long as there is data transmission, *CRS* is valid. In addition, *CRS* is effective only if PHY is in half-duplex mode;

COL: Collision detection signal, does not need to be synchronized to the reference clock, is valid only if PHY is in half-duplex mode.

GMII interface is shown in Fig 14. 2.

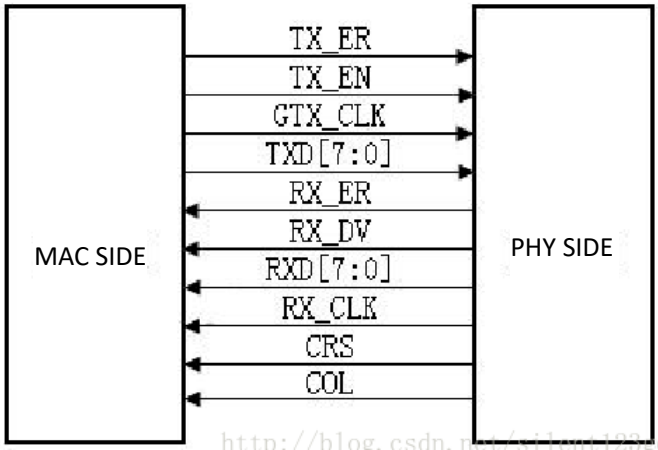


Figure 14.2 GMI Interface

Compared with the MII interface, the data width of the GMII is changed from 4 bits to 8 bits. The control signals in the GMII interface such as *TX_ER*, *TX_EN*, *RX_ER*, *RX_DV*, *CRS*, and *COL* function the same as those in the MII interface. The frequencies of transmitting reference clock *GTX_CLK* and the receiving reference clock *RX_CLK* are both 125 MHz (1000 Mbps / 8 = 125 MHz).

There is one point that needs special explanation here, that is, the transmitting reference clock *GTX_CLK* is different from the *TX_CLK* in the MII interface. The *TX_CLK* in the MII interface is provided by the PHY chip to the MAC chip, and the *GTX_CLK* in the GMII interface is provided to the PHY chip by the MAC chip. The directions are different.

In practical applications, most GMII interfaces are compatible with MII interfaces. Therefore, the general GMII interface has two transmitting reference clocks: *TX_CLK* and *GTX_CLK* (the

directions of the two are different, as mentioned above). When used as the MII mode, *TX_CLK* and 4 of the 8 data lines are used.

See Figure 14.3 for RGMII interface.

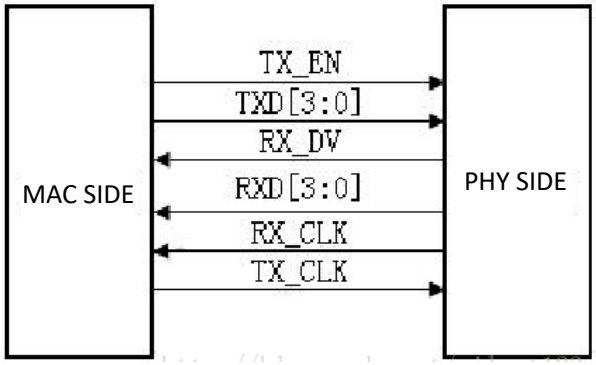


Figure 14.3 RGMII interface

RGMII, or reduced GMII, is a simplified version of GMII, which reduces the number of interface signal lines from 24 to 14 (COL/CRS port status indication signals, not shown here), the clock frequency is still 125 MHz, and the TX/RX data width is changed from 8 to 4 bits. To keep the transmission rate of 1000 Mbps unchanged, the RGMII interface samples data on both the rising and falling edges of the clock. *TXD*[3:0]/*RXD*[3:0] in the GMII interface is transmitted on the rising edge of the reference clock, and *TXD*[7:4]/*RXD*[7:4] in the GMII interface is transmitted on the falling edge of the reference clock. RGMI is also compatible with both 100 Mbps and 10 Mbps rates, with reference clock rates of 25 MHz and 2.5 MHz, respectively.

The *TX_EN* signal line transmits *TX_EN* and *TX_ER* information, *TX_EN* is transmitted on the rising edge of *TX_CLK*, and *TX_ER* is transmitted on the falling edge. Similarly, *RX_DV* and *RX_ER* are transmitted on the *RX_DV* signal line, and *RX_DV* is transmitted on the rising edge of *RX_CLK*, and *RX_ER* is transmitted on the falling edge.

14.3.2 Hardware Design

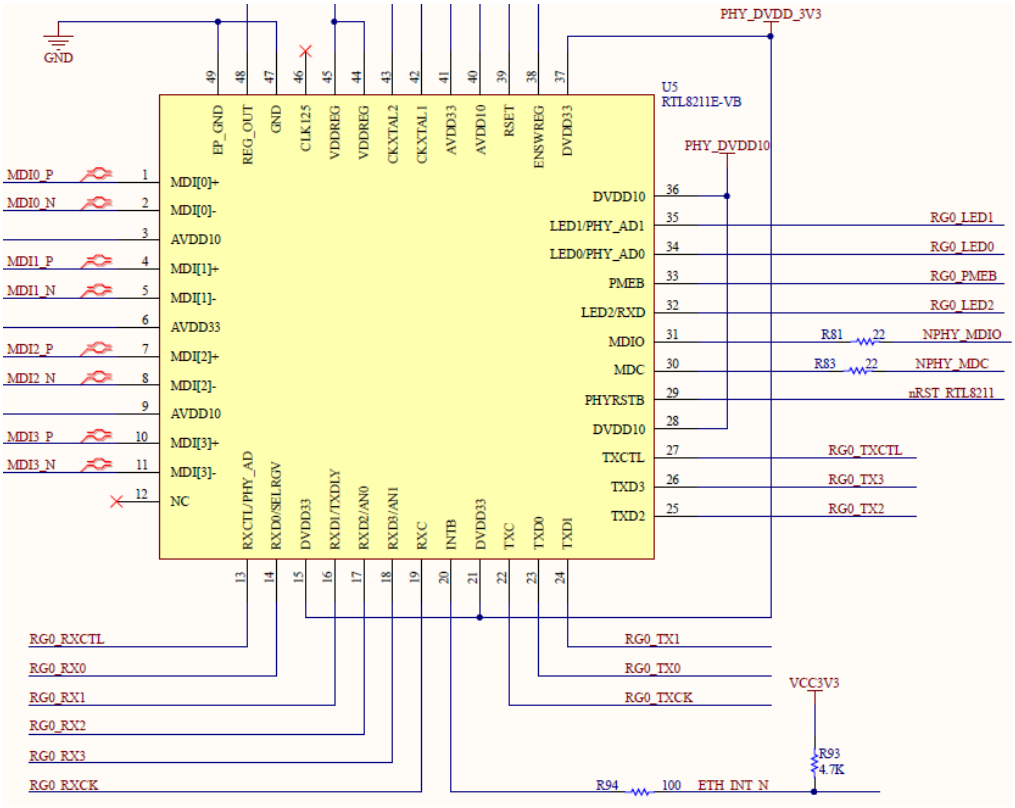


Figure 14.4 Schematics of RTL8211E-VB

The RTL8211E-VB chip is used to form a Gigabit Ethernet module on the experiment board. The schematics is shown in Figure 14.4. The PHY chip is connected to the FPGA by receiving and transmitting two sets of signals. The receiving group signal prefix is RG0_RX, and the transmitting group signal prefix is RG0TX, which is composed of a control signal CTL, a clock signal CK and four data signals 3-0. RG0_LED0 and RG0_LED1 are respectively connected to the network port yellow signal light and green signal light. At the same time, the FPGA can configure the PHY chip through the clock line NPHY_MDC and the data line NPHY_MDIO.

14.3.3 Design of the Program

（1） Loopback test design (test1)

The first step: introduction to the program

The loopback test is very simple, which just needs to output the input data directly.

```
module test1 (
    input           rst,
    input           rxc,
    input           rxdv,
    input   [3:0]   rxd,
    output          txc,
    output          txen,
    output  [3:0]   txd,
);
```

```
    assign      txd   = rxd;
    assign      txen = rxdv;
    assign      txc   = rxc;

endmodule
```

(Note: Each program in this experiment contains a *smi_ctrl* module. In the **config** folder, it is a setting module for the PHY chip, so as to solve the problem that some computers cannot connect to the network port normally, and will not explain in detail)

The second step: pin assignment

Table 14.1 Ethernet Experiment Pin Mapping

Signal Name	Network Label	FPGA Pin	Port Description
rxc	RGMII_RXCK	B12	Input data clock
rxdv	RGMII_RXCTL	A13	Input data control signal
rxd[3]	RGMII_RX3	A15	Input data bit 3
rxd[2]	RGMII_RX2	B14	Input data bit 2
rxd[1]	RGMII_RX1	A14	Input data bit 1
rxd[0]	RGMII_RX0	B13	Input data bit 0
txc	RGMII_TXCK	B20	Output data clock
txen	RGMII_TXCTL	A19	Output data control signal
txd[3]	RGMII_TX3	B18	Output data bit 3
txd[2]	RGMII_TX2	A18	Output data bit 2
txd[1]	RGMII_TX1	B17	Output data bit 1
txd[0]	RGMII_TX0	A17	Output data bit 0
e_mdc	NPHY_MDC	C17	Configuration clock
e_mdio	NPHY_MDIO	B19	Configuration data

Before verification (the default PC NIC is a Gigabit NIC, otherwise it needed to be replaced). PC IP address needs to be confirmed first. In the DOS command window, type **ipconfig -all** command to check it. Example is shown in Fig 14. 5.

```
C:\Users\HW-PC>ipconfig -all

Windows IP Configuration

   Host Name . . . . . . . . . . . . : yvonne
   Primary Dns Suffix  . . . . . . . :
   Node Type . . . . . . . . . . . . : Hybrid
   IP Routing Enabled. . . . . . . . : No
   WINS Proxy Enabled. . . . . . . . : No
   DNS Suffix Search List. . . . . . : hitronhub.home

Ethernet adapter Ethernet 2:

   Connection-specific DNS Suffix  . :
   Description . . . . . . . . . . . : Intel(R) Ethernet Connection (7) I219-V
   Physical Address. . . . . . . . . : 00-D8-61-19-A9-D0
   DHCP Enabled. . . . . . . . . . . : Yes
   Autoconfiguration Enabled . . . . : Yes
   Link-local IPv6 Address . . . . . : fe80::4d10:8892:fa1:9d08%23(Preferred)
   Autoconfiguration IPv4 Address. . : 169.254.157.8(Preferred)
   Subnet Mask . . . . . . . . . . . : 255.255.0.0
   Default Gateway . . . . . . . . . :
   DHCPv6 IAID . . . . . . . . . . . : 385931361
   DHCPv6 Client DUID. . . . . . . . : 00-01-00-01-24-77-1A-25-4C-ED-FB-68-AA-FD
   DNS Servers . . . . . . . . . . . : fec0:0:0:ffff::1%1
                                       fec0:0:0:ffff::2%1
                                       fec0:0:0:ffff::3%1
   NetBIOS over Tcpip. . . . . . . . : Enabled
```

Figure 14.5 PC end IP information

To facilitate subsequent experiments, PC is provided a fixed IP address. Take this experiment as an example, IP configuration is **169.254.157.8** (could be revised, but needs to be consistent to the IP address of target sending module). Find Internet Protocol Version 4(TCP/IPv4) in **Network and Sharing center**. See Fig 14. 6.

Figure 14.6 Configure PC end IP address

Since there is no ARP protocol content (binding IP address and MAC address of the develop board) in this experiment, it needs to be bound manually through the DOS command window. Here, the IP is set to **192.168.0.2** and the MAC address is set to **00-0A-35-01-FE-C0**, (can be replaced by yourself) as shown in Fig 14. 7, the method is as follows: (Note: Run the DOS command window as an administrator)

Run the command: **ARP -s 192.168.0.2 00-0A-35-01-FE-C0**

View binding results: **ARP -a**

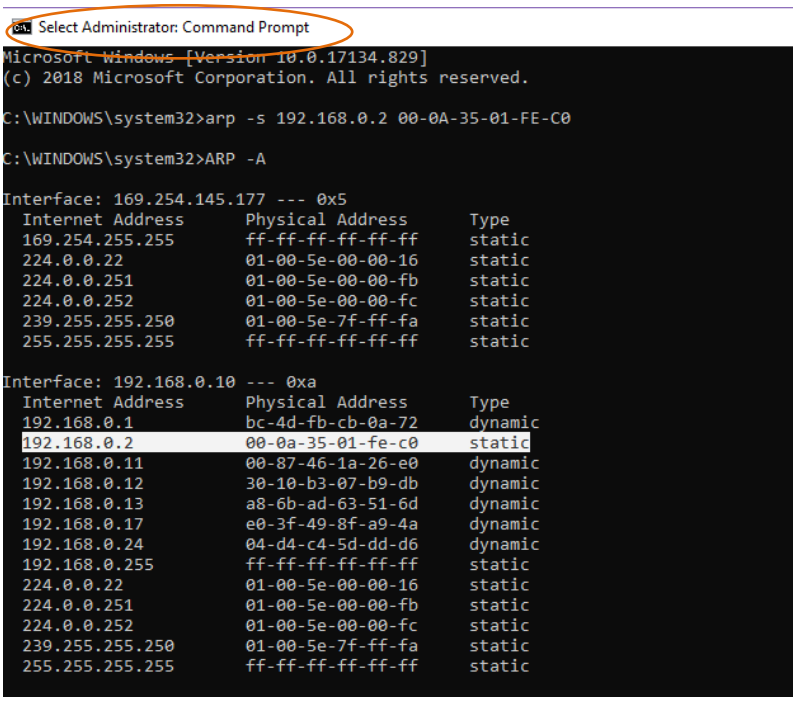


Figure 14.7 Address binding method 1

If a failure occurs while running the ARP command, another way is available , as shown in Figure 14.8:

1) Enter the **netsh i i show** in command to view the number of the local connection, such as the "23" of the computer used this time.
2) Enter **netsh -c “i i” add neighbors 23 (number) “192.168.0.2” “00-0A-35-01-FE-C0”**
3) Enter **arp -a** to view the binding result

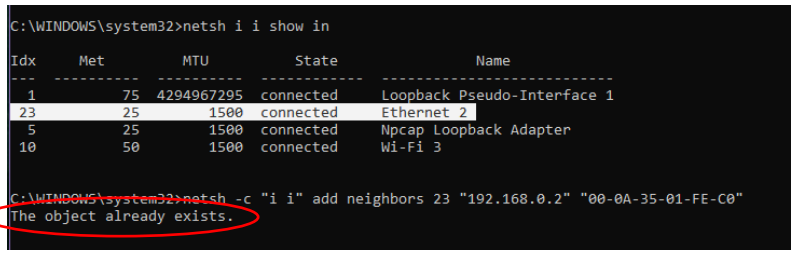


Figure 14.8 Address binding method 2

Next, we also use the DOS command window for connectivity detection, as shown in Fig 14.9. **Ping** is an executable command that comes with the Windows family. Use it to check if the network can be connected. It can help us analyze and determine network faults. Application format: Ping IP address (not host computer IP).

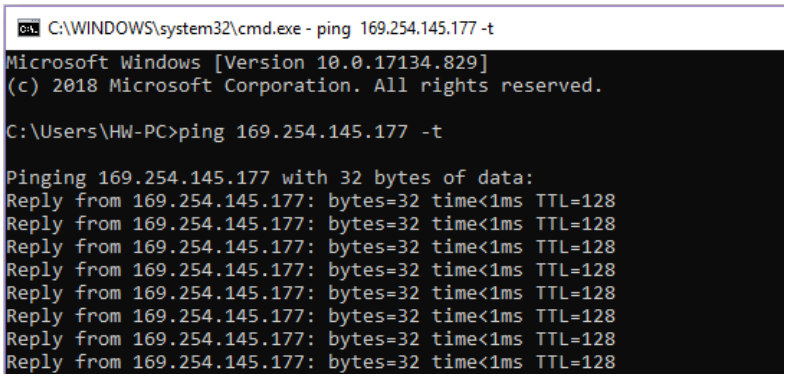


Figure 14.9 Send data

Start SignalTap II, after sending the command, as shown in Fig 14. 10. The data is ordinary and the hardware is intact seen from the screenshot.

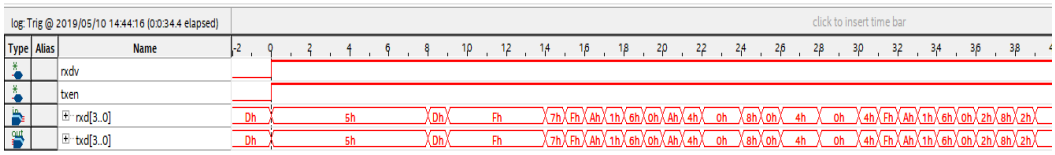


Figure 14.10 SignalTap II data capture

（2） Special IP core configuration (test2)

Because it is the RGMII interface, the data is bilateral along 4-bit data. Therefore, when data processing is performed inside the FPGA, it needs to be converted into 8-bit data. Go to **Installed IP > Library > Basic Functions > I/O** to find **ALTDDIO_IN** and **ALTDDIO_OUT**. To implement it, IP core (*ddio_in*) is called, and after internal data processing, IP core is passed (*ddio_out*) to convert 8-bit data into bilateral edge 4-bit data transfer. It should be noted that, considering the enable signal and data signal synchronization, the enable signal is entered to *ddio* for conversion at the same time. The specific settings are shown in Fig 14. 11 and Fig 14. 12.

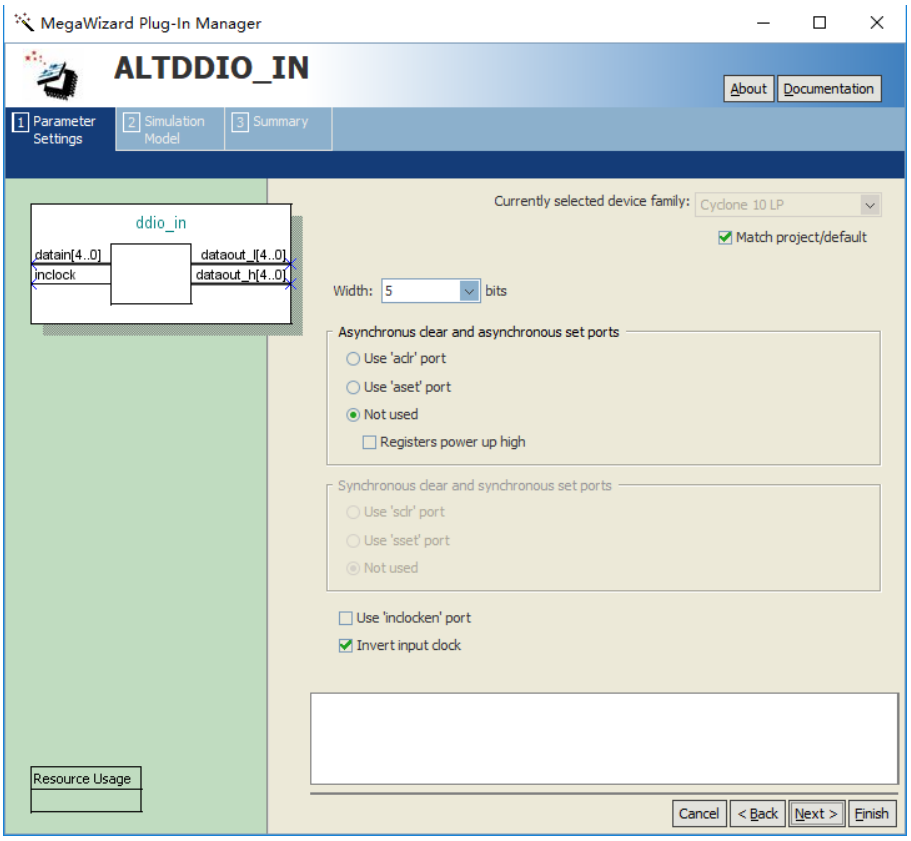


Figure 14.11 ddio_in setting

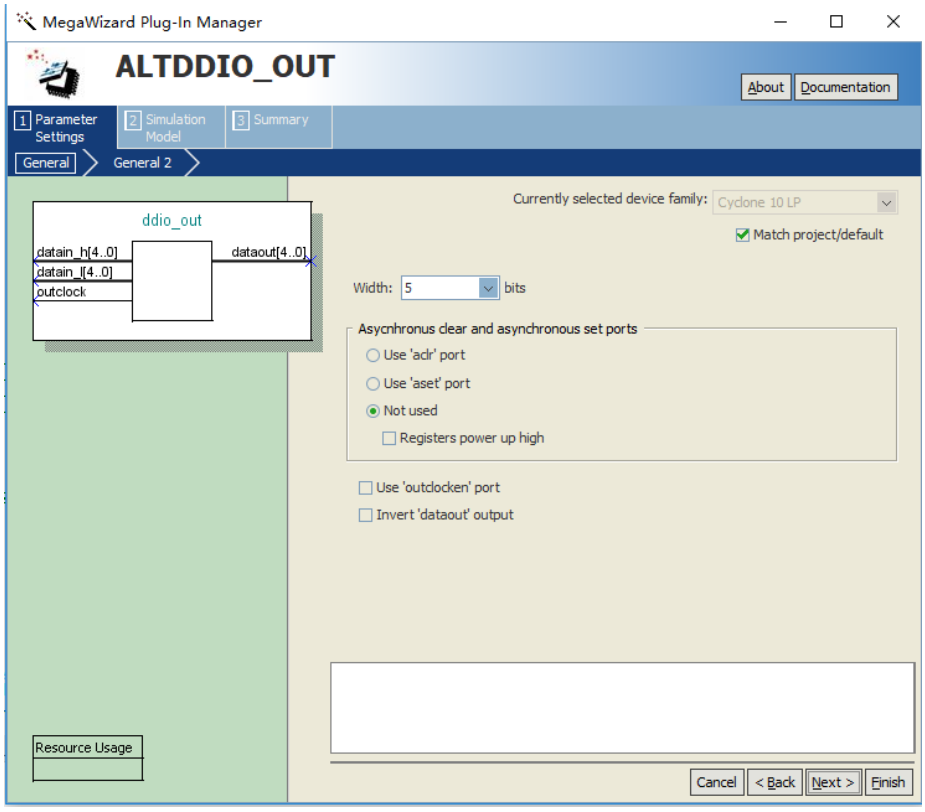


Figure 14.12 ddio_out setting

Considering that the driving ability of the clock provided by the PHY chip is relatively poor, after the phase-locked loop processing, unlike the prior part, the input clock *rxc* selects the

homologous input, as shown in Fig 14. 13, and outputs C0 clock *ddio_clk* as the driving clock of two *ddio* IP cores. As shown in Fig 14. 14, outputs the *C1* clock *txc* as the data transmission clock (note that due to hardware circuit and timing reasons, *txc* needs to be 90° phase difference). See Fig 14. 15.

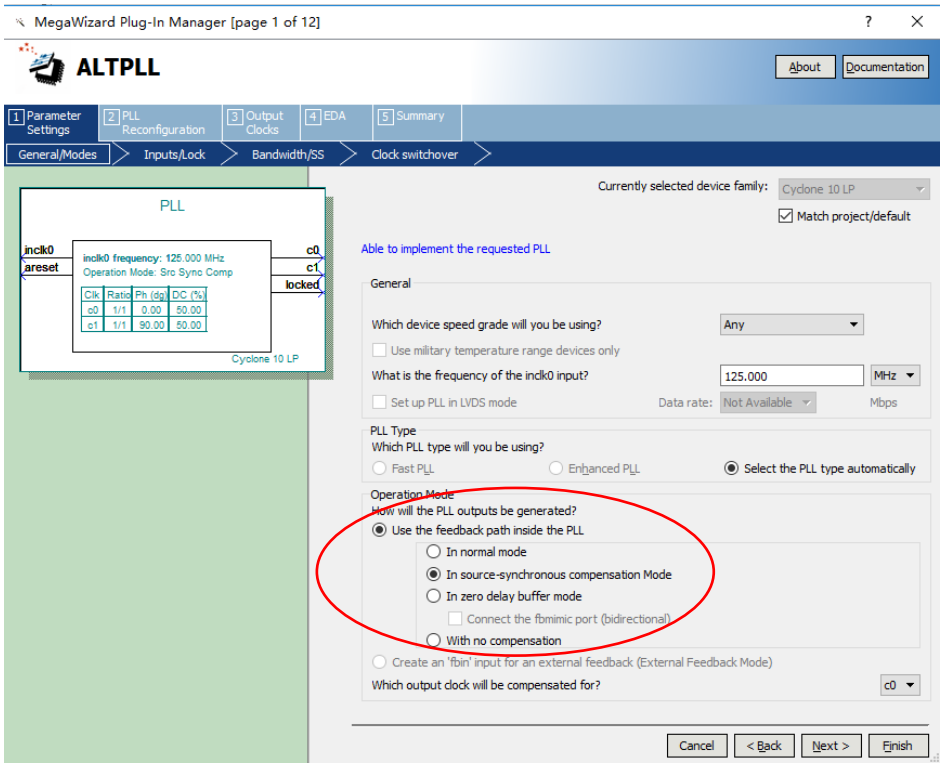


Figure 14.13 PLL input clock setting

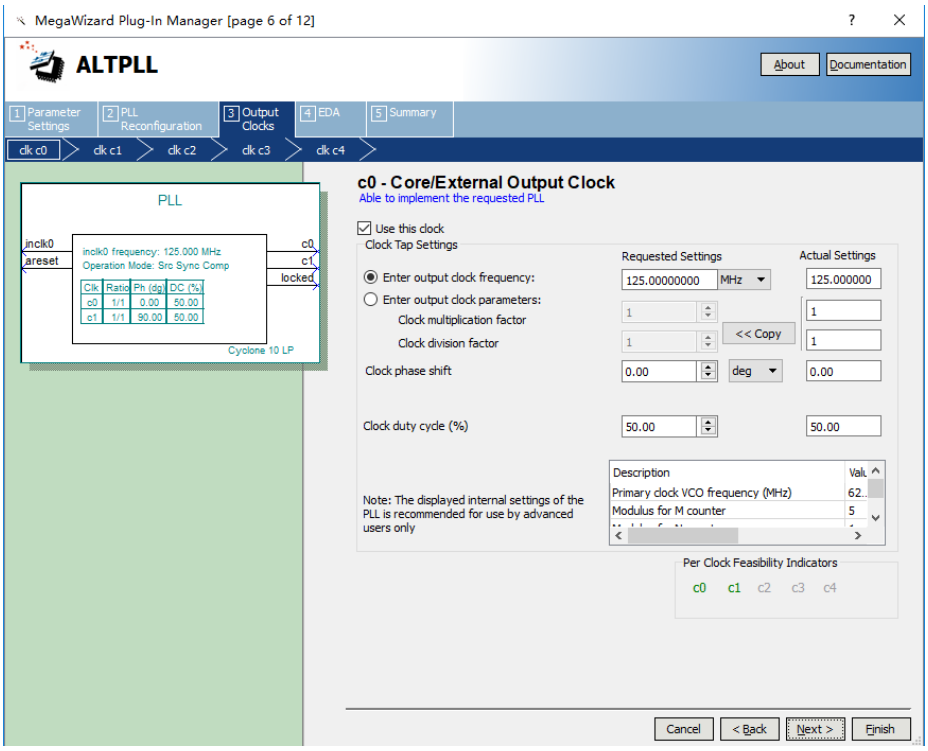


Figure 14.14 PLL output clcok(c0) setting

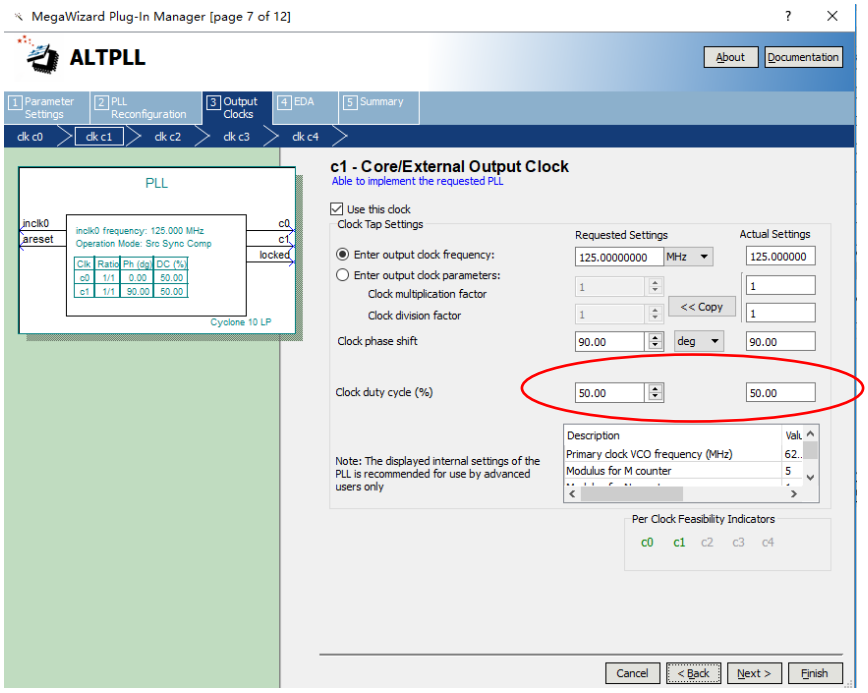


Figure 14.15 PLL output clcok(c1) setting

The three IP cores are instantiated into the previous loopback test, and the data transmission correctness test is performed. (It is necessary to notice the ordered timing. The *ddio_out* input data needs to be reversed. For details, refer to the project file (test2)). This time a network debugging assistant applet is uaed as an auxiliary testing tool. Program the board and verify it.

As shown in Figure 14.16, after setting the correct address and data type, we send the detection information (*love you!*) through the host computer. The data packet is captured by Wireshark, as shown in Figure 14.17. The data is correctly transmitted back to the PC.

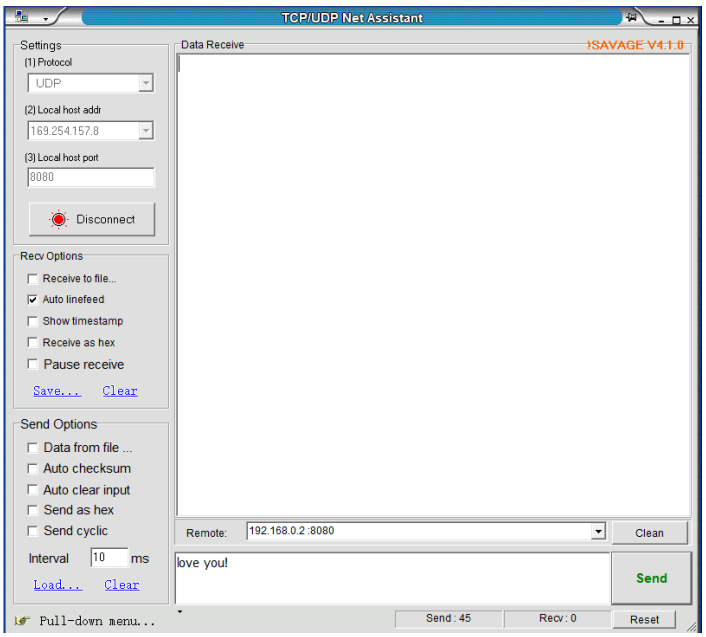


Figure 14.16 Host computer sends the test data

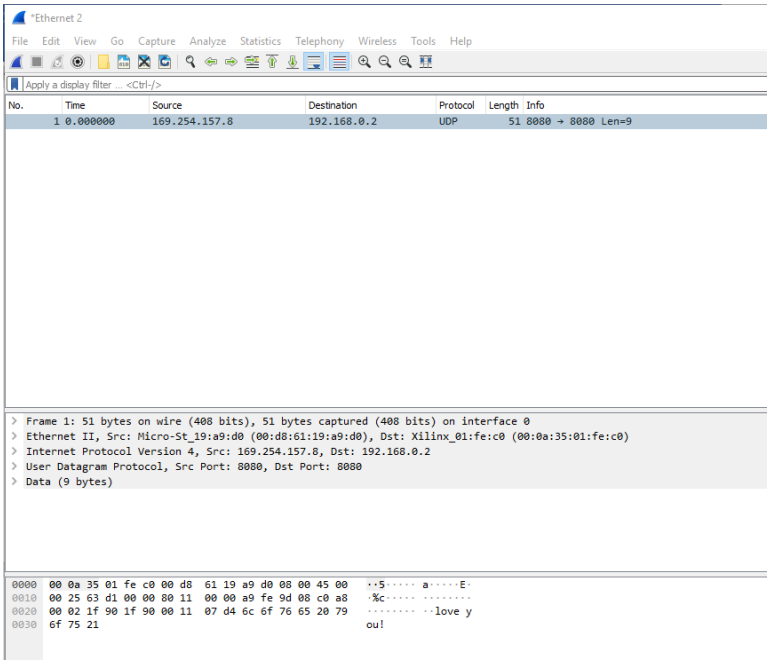


Figure 14.17 Corrrect reception of data on the PC side

（3） Complete Ethernet data transmission design

For complete Ethernet data transmission, it is necessary to have the receiving part of the data and the transmitting part of the data. For the convenience of experiment, we store the data transmitted by the PC first in the RAM. After reading via the transmitting end, send it to the PC. For a series of data unpacking and packaging, refer to the project file "ethernet". A brief introduction to each module follows.

1) Data receiving module (ip_receive)

The problem to be solved by this module is to detect and identify the data frame, unpack the valid data frame, and store the real data in the ram.

```
always @ (posedge clk) begin
    if (clr) begin
        rx_state <= idle;
        data_receive <= 1'b0;
    end
    else
    case (rx_state)

        idle  :
        begin
            valid_ip_P <= 1'b0;
            byte_counter <= 3'd0;
            data_counter <= 10'd0;
            mydata <= 32'd0;
            state_counter <= 5'd0;
            data_o_valid <= 1'b0;
            ram_wr_addr <= 0;
```

```
        if (e_rxdv == 1'b1) begin
            if (datain[7:0] == 8'h55) begin                    //First 55 received
                rx_state <= six_55;
                mydata <= {mydata[23:0], datain[7:0]};
            end
            else
                rx_state <= idle;
        end
    end

    six_55    :
    begin                                                      // 6 0x55 received
        if ((datain[7:0] == 8'h55) && (e_rxdv == 1'b1)) begin
            if (state_counter == 5) begin
                state_counter <= 0;
                rx_state <= spd_d5;
            end
            else
                state_counter <= state_counter + 1'b1;
        end
        else
            rx_state <= idle;
    end

    spd_d5    :
    begin                                                      //A 0xd5 received
        if ((datain[7:0] == 8'hd5) && (e_rxdv == 1'b1))
            rx_state <= rx_mac;
        else
            rx_state <= idle;
    end

    rx_mac    :
    begin                          // Receive target mac address and source mac address
        if (e_rxdv == 1'b1) begin
            if (state_counter < 5'd11) begin
                mymac <= {mymac[87:0], datain};
                state_counter <= state_counter + 1'b1;
            end
            else begin
                board_mac <= mymac[87:40];
                pc_mac <= {mymac[39:0], datain};
                state_counter <= 5'd0;
                if((mymac[87:72] == 16'h000a) && (mymac[71:56] == 16'h3501) &&
```

```
(mymac[55:40] == 16'hfec0))     // Determine if the target MAC Address is the current FPGA
                                rx_state <= rx_IP_Protocol;
                            else
                                rx_state <= idle;
                        end
                    end
                    else
                        rx_state <= idle;
                end

                rx_IP_Protocol  :
                begin                                           // Receive 2 bytes of IP TYPE
                    if (e_rxdv == 1'b1) begin
                        if (state_counter < 5'd1) begin
                            myIP_Prtcl <= {myIP_Prtcl[7:0], datain[7:0]};
                            state_counter <= state_counter+1'b1;
                        end
                        else begin
                            IP_Prtcl <= {myIP_Prtcl[7:0],datain[7:0]};
                            valid_ip_P <= 1'b1;
                            state_counter <= 5'd0;
                            rx_state <= rx_IP_layer;
                        end
                    end
                    else
                        rx_state <= idle;
                end

                rx_IP_layer     :
                begin                               // Receive 20 bytes of udp virtual header, ip address
                    valid_ip_P <= 1'b0;
                    if (e_rxdv == 1'b1) begin
                        if (state_counter < 5'd19) begin
                            myIP_layer <= {myIP_layer[151:0], datain[7:0]};
                            state_counter <= state_counter + 1'b1;
                        end
                        else begin
                            IP_layer <= {myIP_layer[151:0], datain[7:0]};
                            state_counter <= 5'd0;
                            rx_state <= rx_UDP_layer;
                        end
                    end
                    else
                        rx_state <= idle;
```

```
            end

            rx_UDP_layer  :
            begin                   // Accept 8-byte UDP port number and UDP packet length
                rx_total_length <= IP_layer[143:128];
                pc_IP <= IP_layer[63:32];
                board_IP <= IP_layer[31:0];
                if (e_rxdv == 1'b1) begin
                    if (state_counter < 5'd7) begin
                        myUDP_layer <= {myUDP_layer[55:0], datain[7:0]};
                        state_counter <= state_counter + 1'b1;
                    end
                    else begin
                        UDP_layer <= {myUDP_layer[55:0], datain[7:0]};
                        rx_data_length <= myUDP_layer[23:8]; //length of UDP data package
                        state_counter <= 5'd0;
                        rx_state <= rx_data;
                    end
                end
                else
                    rx_state <= idle;
            end

            rx_data   :
            begin                                           //Receive UDP data
                if (e_rxdv == 1'b1) begin
                    if (data_counter == rx_data_length-9) begin             //Save last data
                        data_counter <= 0;
                        rx_state <= rx_finish;
                        ram_wr_addr <= ram_wr_addr + 1'b1;
                        data_o_valid <= 1'b1;                                   // Write RAM

                        if (byte_counter == 3'd3) begin
                            data_o <= {mydata[23:0], datain[7:0]};
                            byte_counter <= 0;
                        end
                        else if (byte_counter==3'd2) begin
                            data_o <= {mydata[15:0], datain[7:0],8'h00}; //Less than 32-bit,
//add '0'
                            byte_counter <= 0;
                        end
                        else if (byte_counter==3'd1) begin
                            data_o <= {mydata[7:0], datain[7:0], 16'h0000};   //Less than
//32-bit , add '0'
```

```
                                        byte_counter <= 0;
                        end
                        else if (byte_counter==3'd0) begin
                            data_o <= {datain[7:0], 24'h000000};          //Less than 32-bit,
//add '0'
                            byte_counter <= 0;
                        end
                    end
                    else begin
                        data_counter <= data_counter + 1'b1;
                        if (byte_counter < 3'd3) begin
                            mydata <= {mydata[23:0], datain[7:0]};
                            byte_counter <= byte_counter + 1'b1;
                            data_o_valid <= 1'b0;
                        end
                        else begin
                            data_o <= {mydata[23:0], datain[7:0]};
                            byte_counter <= 3'd0;
                            data_o_valid <= 1'b1;          // Accept 4bytes of data, write
//RAM request
                            ram_wr_addr <= ram_wr_addr+1'b1;

                        end
                    end
                end
                else
                    rx_state <= idle;
            end

            rx_finish  :
            begin
                data_o_valid <= 1'b0;                          //added for receive test
                data_receive <= 1'b1;
                rx_state <= idle;
            end

            default    :    rx_state <= idle;
        endcase
end
```

The receiving module is to perform step by step analysis on the received data.

Idle state: If '55' is received, it jumps to the *six_55* state.

Six_55 state: If it continues to receive six consecutive 55s, it will jump to the *spd_d5* state, otherwise it will return the *idle* state.

Spd_d5 state: If 'd5' is countinuing received, it proves that the complete packet preamble "55_55_55_55_55_55_55_d5" has been received, and jumps to *rx_mac*, otherwise it returns the *idle* transition.

rx_mac state: This part is the judgment of the target MAC address and the source MAC address. If it matches, it will jump to the *rx_IP_Protocol* state, otherwise it will return the *idle* state and resend.

rx_IP_Protocol state: Determine the type and length of the packet and jump to the *rx_IP_layer* state.

rx_IP_layer state: Receive 20 bytes of UDP virtual header and IP address, jump to *rx_UDP_layer* state

rx_UDP_layer state: Receive 8-byte UDP port number and UDP packet length, jump to *rx_data* state

Rx_data state: Receive UDP data, jump to *rx_finish* state

Rx_finish state: A packet of data is received, and jumps to the *idle* state to wait for the arrival of the next packet of data.

2) Data sending module (ip_send)

The main content of this module is to read out the data in the RAM, package and transmit the data with the correct packet protocol type (UDP). Before transmitting, the data is also checked by CRC.

```
initial begin
    tx_state <= idle;
    //Define IP header
    preamble[0] <= 8'h55;                    //7 preambles "55", one frame start
//character "d5"
    preamble[1] <= 8'h55;
    preamble[2] <= 8'h55;
    preamble[3] <= 8'h55;
    preamble[4] <= 8'h55;
    preamble[5] <= 8'h55;
    preamble[6] <= 8'h55;
    preamble[7] <= 8'hD5;

    mac_addr[0] <= 8'hB4;                    //Target MAC address "ff-ff-ff-ff-ff-ff", full ff is
//broadcast package
    mac_addr[1] <= 8'h2E;                    //Target MAC address "B4-2E-99-20-C4-61",
// For the PC-side address used for this experiment, change the content according to the actual
//PC in the debugging phase.
    mac_addr[2] <= 8'h99;
    mac_addr[3] <= 8'h20;
    mac_addr[4] <= 8'hC4;
    mac_addr[5] <= 8'h61;

    mac_addr[6] <= 8'h00;                    //Source MAC address "00-0A-35-01-FE-C0"
```

```
        mac_addr[7] <= 8'h0A;                    //Modify it according to the actual needs
        mac_addr[8] <= 8'h35;
        mac_addr[9] <= 8'h01;
        mac_addr[10]<= 8'hFE;
        mac_addr[11]<= 8'hC0;

        mac_addr[12]<= 8'h08;                    //0800: IP package type
        mac_addr[13]<= 8'h00;

        i<=0;
end
```

This part defines the preamble of the data packet, the MAC address of the PC, the MAC address of the development board, and the IP packet type. It should be noted that in the actual experiment, the MAC address of the PC needs to be modified. Keep the MAC address consistent along the project, otherwise the subsequent experiments will not receive data.

```
always @ (posedge clk) begin
case (tx_state)
        idle  :
        begin
                e_txen <= 1'b0;
                crcen <= 1'b0;
                crcre <= 1;
                j <= 0;
                dataout <= 0;
                ram_rd_addr <= 1;
                tx_data_counter <= 0;
                if (time_counter == 32'h04000000) begin          //Wait for the delay, send a data
//package regularly
                tx_state <= start;
                time_counter <= 0;
        end
        else
                time_counter <= time_counter + 1'b1;
end

start :
begin                //IP header
        ip_header[0] <= {16'h4500, tx_total_length};          //Version: 4; IP header length: 20;
//IP total length
        ip_header[1][31:16] <= ip_header[1][31:16]+1'b1;      // Package serial number
        ip_header[1][15:0] <= 16'h4000;                       //Fragment offset
        ip_header[2] <= 32'h80110000;                      //mema[2][15:0] protocol: 17(UDP)
```

```
    ip_header[3] <= 32'hc0a80002;                    //Source MAC address
    ip_header[4] <= 32'hc0a80003;                    //Target MAC address
    ip_header[5] <= 32'h1f901f90;                    // 2-byte source port number and
//2-byte target port number
    ip_header[6] <= {tx_data_length, 16'h0000};      //2 bytes of data length and 2
//bytes of checksum (none)
    tx_state <= make;
end

make    :
begin                          // Generate a checksum of the header
    if (i == 0) begin
        check_buffer <= ip_header[0][15:0] + ip_header[0][31:16] +
                        ip_header[1][15:0] + ip_header[1][31:16] +
                        ip_header[2][15:0] + ip_header[2][31:16] +
                        ip_header[3][15:0] + ip_header[3][31:16] +
                        ip_header[4][15:0] + ip_header[4][31:16];
        i <= i + 1'b1;
    end
    else if(i == 1) begin
        check_buffer[15:0] <= check_buffer[31:16] + check_buffer[15:0];
        i <= i+1'b1;
    end
    else begin
        ip_header[2][15:0] <= ~check_buffer[15:0];    //header checksum
        i <= 0;
        tx_state <= send55;
    end
end

send55  :
begin                                  // Send 8 IP preambles: 7 “55”, 1 “d5”
    e_txen <= 1'b1;                                  //GMII transmitted valid data
    crcre <= 1'b1;                                   //reset CRC
    if(i == 7) begin
        dataout[7:0] <= preamble[i][7:0];
        i <= 0;
        tx_state <= sendmac;
    end
    else begin
        dataout[7:0] <= preamble[i][7:0];
        i <= i + 1'b1;
    end
end
```

```
sendmac :
begin                          // Send target MAC address, source MAC address and IP packet type
      crcen <= 1'b1;               // CRC check enable, crc32 data check starts from the target MAC

      crcre <= 1'b0;
      if (i == 13) begin
            dataout[7:0] <= mac_addr[i][7:0];
            i <= 0;
            tx_state <= sendheader;
      end
      else begin
            dataout[7:0] <= mac_addr[i][7:0];
            i <= i + 1'b1;
      end
end

sendheader    :
begin                                                    // Send 7 32-bit IP headers
      datain_reg <= datain;                              //Prepare the data to be transmitted
      if(j == 6) begin
            if(i == 0) begin
                  dataout[7:0] <= ip_header[j][31:24];
                  i <= i + 1'b1;
            end
            else if(i == 1) begin
                  dataout[7:0] <= ip_header[j][23:16];
                  i <= i + 1'b1;
            end
            else if(i == 2) begin
                  dataout[7:0] <= ip_header[j][15:8];
                  i <= i + 1'b1;
            end
            else if(i == 3) begin
                  dataout[7:0] <= ip_header[j][7:0];
                  i <= 0;
                  j <= 0;
                  tx_state <= senddata;
            end
      end
      else begin
            if(i == 0) begin
                  dataout[7:0] <= ip_header[j][31:24];
                  i <= i + 1'b1;
```

```
            end
            else if(i == 1) begin
                dataout[7:0] <= ip_header[j][23:16];
                i <= i + 1'b1;
            end
            else if(i == 2) begin
                dataout[7:0] <= ip_header[j][15:8];
                i <= i + 1'b1;
            end
            else if(i == 3) begin
                dataout[7:0] <= ip_header[j][7:0];
                i <= 0;
                j <= j + 1'b1;
            end
        end
end

senddata :
begin                                                   //Transmit UDP packets
        if(tx_data_counter == tx_data_length - 9) begin //Trasnmit last data
            tx_state <= sendcrc;
                if (i == 0) begin
                    dataout[7:0] <= datain_reg[31:24];
                    i <= 0;
                end
                else if (i == 1) begin
                    dataout[7:0] <= datain_reg[23:16];
                    i <= 0;
                end
                else if (i == 2) begin
                    dataout[7:0] <= datain_reg[15:8];
                    i <= 0;
                end
                else if (i == 3) begin
                    dataout[7:0] <= datain_reg[7:0];
                    datain_reg <= datain;                   //Prapare the data
                    i <= 0;
                end
        end
        else begin                                          //Send other data package
            tx_data_counter <= tx_data_counter+1'b1;
                if (i == 0) begin
                    dataout[7:0] <= datain_reg[31:24];
                    i <= i + 1'b1;
```

```
                    ram_rd_addr <= ram_rd_addr + 1'b1;          // Add 1 to the RAM address,
//let the RAM output data in advance.
                end
                else if (i == 1) begin
                    dataout[7:0] <= datain_reg[23:16];
                    i <= i + 1'b1;
                end
                else if (i == 2) begin
                    dataout[7:0] <= datain_reg[15:8];
                    i <= i + 1'b1;
                end
                else if (i == 3) begin
                    dataout[7:0] <= datain_reg[7:0];
                    datain_reg <= datain;                       //Prepare data

                    i <= 0;
                end
    end
end

sendcrc   :
begin                                                   //Send 32-bit CRC checksum
    crcen <= 1'b0;
    if (i == 0) begin
        dataout[7:0] <= {~crc[24], ~crc[25], ~crc[26], ~crc[27], ~crc[28], ~crc[29], ~crc[30],
~crc[31]};
        i <= i + 1'b1;
    end
    else begin
        if (i == 1) begin
            dataout[7:0]  <=  {~crc[16],  ~crc[17],  ~crc[18],  ~crc[19],  ~crc[20],  ~crc[21],
~crc[22], ~crc[23]};
                    i <= i + 1'b1;
        end
        else if (i == 2) begin
            dataout[7:0] <= {~crc[8], ~crc[9], ~crc[10], ~crc[11], ~crc[12], ~crc[13], ~crc[14],
~crc[15]};
                    i <= i + 1'b1;
        end
        else if (i == 3) begin
            dataout[7:0]  <=  {~crc[0],  ~crc[1],  ~crc[2],  ~crc[3],  ~crc[4],  ~crc[5],  ~crc[6],
~crc[7]};
            i <= 0;
            tx_state <= idle;
```

```
            end
        end
end

default    :      tx_state <= idle;
endcase
end
```

Idle state: Waiting for delay, sending a packet at regular intervals and jumping to the *start* state.

Start state: Send the packet header and jump to the *make* state.

make state: Generates the checksum of the header and jumps to the *send55* state.

Send55 status: Send 8 preambles and jump to the *sendmac* state.

sendmac state: Send the target MAC address, source MAC address and IP packet type, and jump to the *sendheader* state.

sendheader state: Sends 7 32-bit IP headers and jumps to the *senddata* state.

senddata state: Send UDP packets and jump to the *sendcrc* state.

sendcrc state: Sends a 32-bit CRC check and returns the *idle* state.

Following the above procedure, the entire packet of data is transmitted, and the *idle* state is returned to wait for the transmission of the next packet of data.

3) CRC check module (crc)

   The CRC32 check of an IP packet is calculated at the destination MAC Address and until the last data of a packet. The CRC32 verilog algorithm and polynomial of Ethernet can be generated directly at the following website:
   http://www.easics.com/webtools/crctool

4) UDP data test module (UDP)

   This module only needs to instantiate the first three sub-modules together. Check the correctness of each connection.

5) Top level module settings (ethernet)

   The PLL, *ddio_in*, *ddio_out*, *ram*, and *UDP* modules are instantiated to the top level entity, and specific information is stored in advance in the RAM (Welcome To ZGZNXP World!). When there is no data input, the FPGA always sends this information. With data input, the received data is sent. Refer to the project files for more information.

14.4 Experiment Verification

The pin assignment of this test procedure is identical to that in Test 1.

Before the promming the development board, it is necessary to note that the IP address of the PC and the MAC address of the development board must be determined and matched, otherwise the data will not be received.

Download the compiled project to the development board. As shown in Figure 14.18, the FPGA is keeping sending information to the PC. The entire transmitted packet can also be seen in Wireshark, as shown in Figure 14.19.

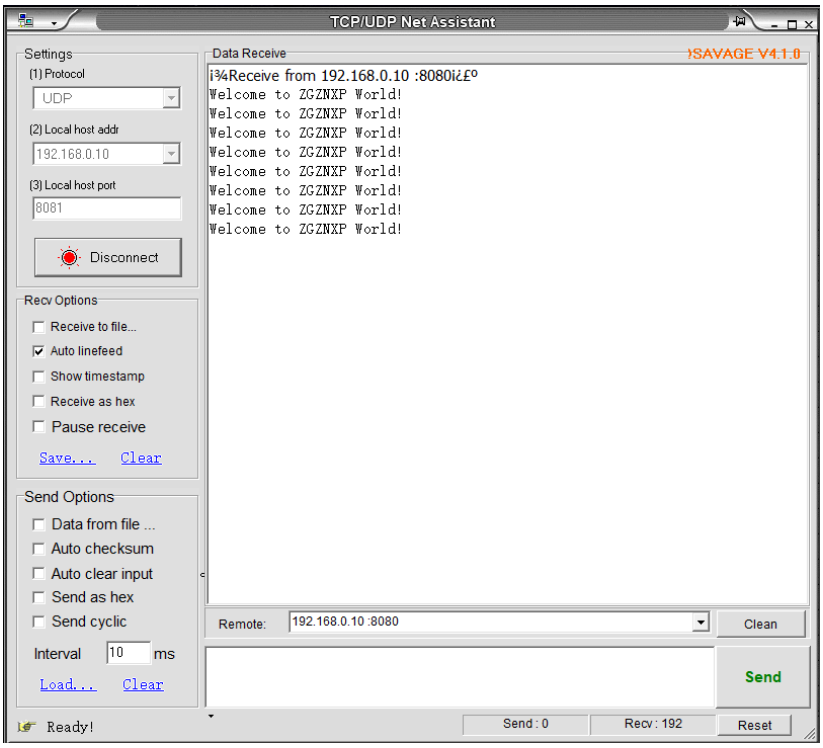


Figure 14.18 Send specific information

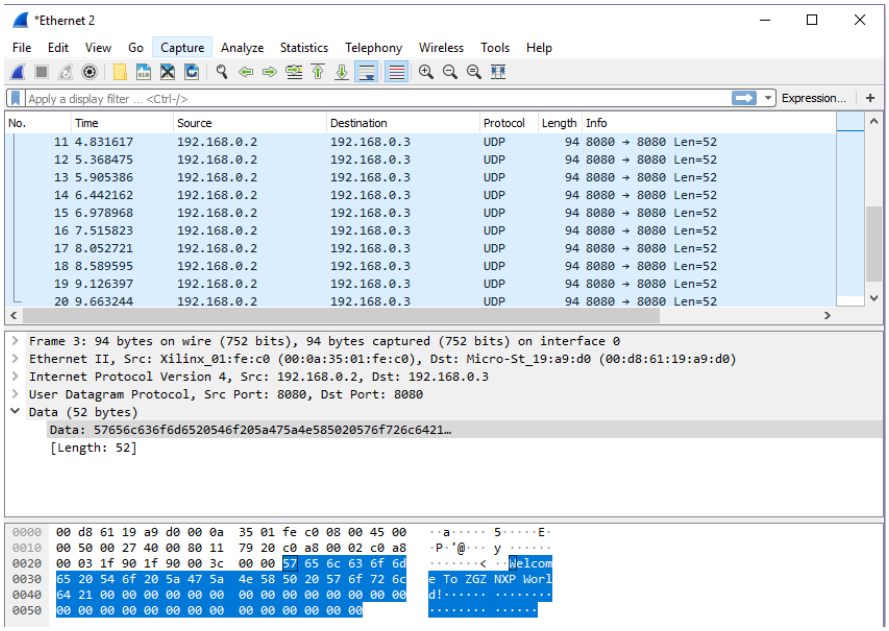


Figure 14.19 Specific information package

When the PC sends data to the FPGA, as shown in Fig 14. 20, the entire packet arrives at the FPGA, and then the FPGA repackages the received data and sends it to the PC. See Fig 14. 21, the network assistant also receives the transmitted data information accurately, as shown in Fig 14. 22. Similarly, through SignalTap we can see the process of writing the received data, as shown in Fig 14. 23.

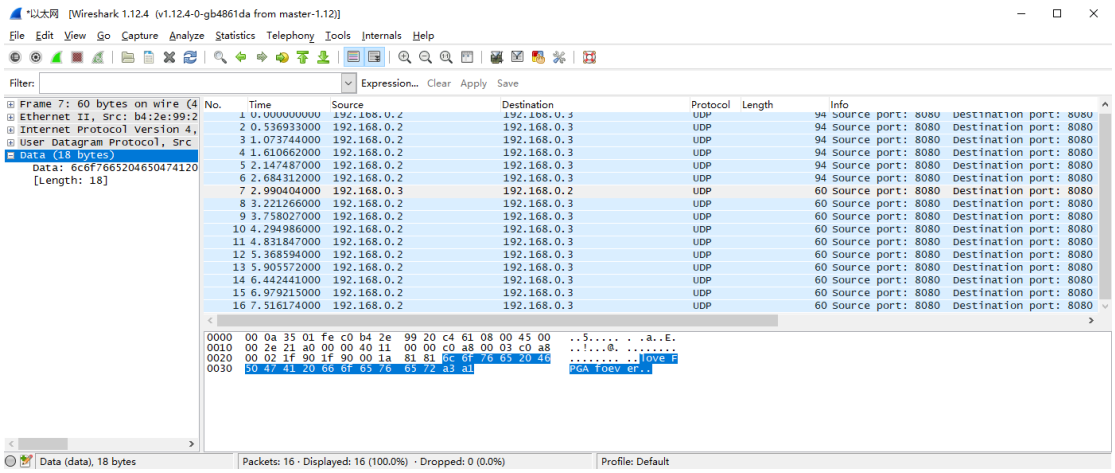


Figure 14. 20 PC send data package

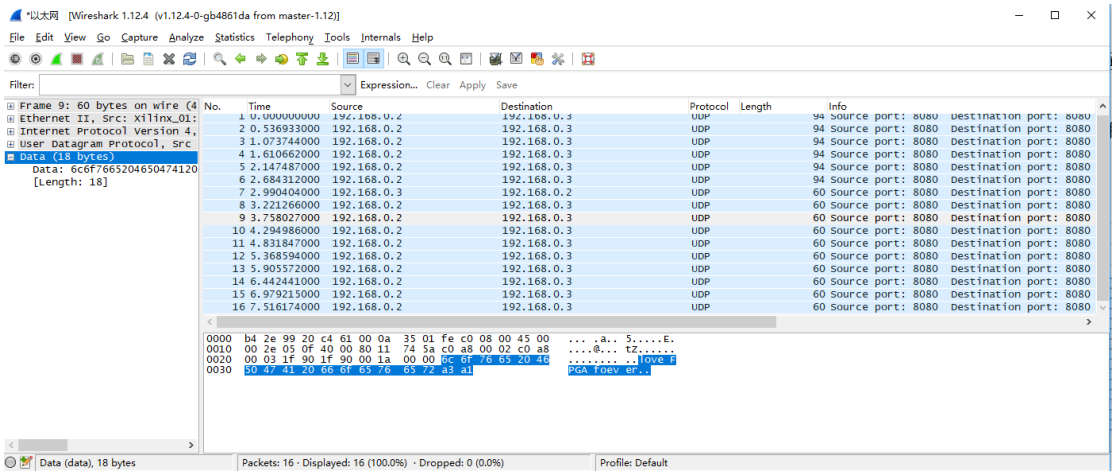


Figure 14.21 The FPGA repackages the received data and sends it to the PC

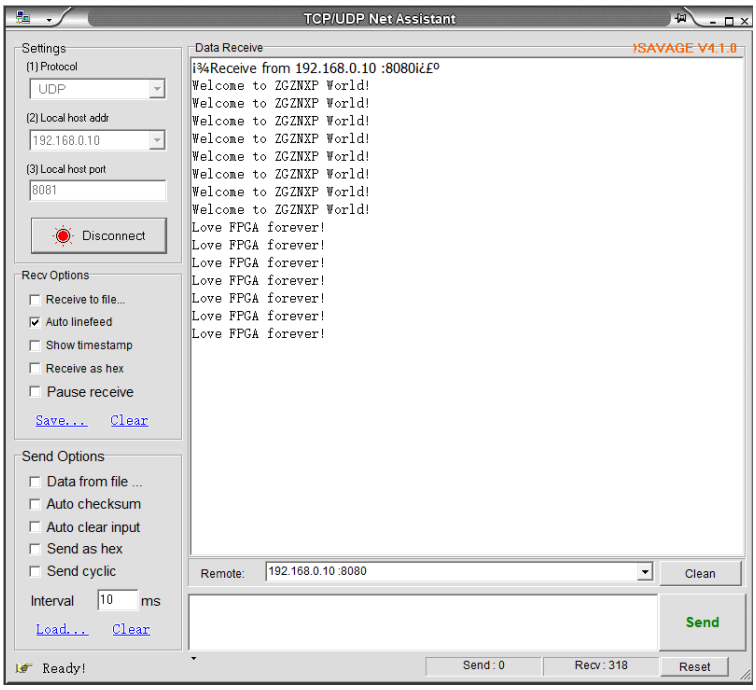


Figure 14.22 Information received by PC from FPGA

Figure 14.23 FPGA end data and stored in the RAM process

It should be noted that Ethernet II specifies the Ethernet frame data field is a minimum of 46 bytes, that is, the minimum Ethernet frame is 6+6+2+46+4=64. The 4-byte FCS is removed, so the packet capture is 60 bytes. When the length of the data field is less than 46 bytes, the MAC sublayer is padded after the data field to satisfy the data frame length of not less than 64 bytes. When communicating over a UDP LAN, "Hello World" often occurs for testing, but "Hello World" does not meet the minimum valid data (64-46) requirements. It is less than 18 bytes but the other party is still available for receiving, because data is complemented in the MAC sublayer of the link layer, less than 18 bytes are padded with ‘0’s. However, when the server is on the public network and the client is on the internal network, if less than 18 bytes of data is transmitted, the receiving end cannot receive the data. Therefore, if there is no data received, the information to be sent should be increased to more than 18 bytes.

Experiment 15 SRAM Read and Write

15.1 Experiment Objective

（1） Learn the read and write of SRAM

（2） Review frequency division, button debounce, and hex conversion experiment content

15.2 Experiment Implement

（1） Control the read and write function of SRAM by controling the button

（2） The data written to the SRAM is read out again and displayed on the segment display

（3） In the process of reading data, it is required to have a certain time interval for each read operation.

15.3 Experiment

15.3.1 Introduction to SRAM

SRAM (Static Random-Access Memory) is a type of random access memory. The “static” means that as long as the power is on, the data in the SRAM will remain unchanged. However, the data will still be lost after power turned off, which is the characteristics of the RAM.

Two SRAMs (IS61WV25616BLL) are on the development board, each SRAM has 256 * 16 words of storage space. Each word is 16-bit. The maximum read and write speed can reach 100 MHz. The physical picture is shown in Figure 15.1.

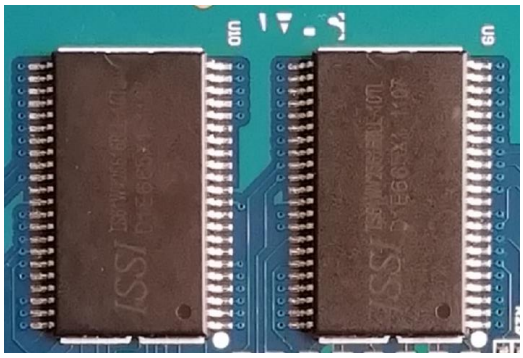


Figure 15.1 SRAM physical picture

15.3.2 Hardware Design

As shown in Figure 15.2, a set of control signals (low signal is valid): chip selection signal CE, read control signal OE, write enable control number WE, and two byte control signals UB and LB, through CE_N_SRAM, OE_N_SRAM, WE_N_SRAM, UB_N_SRAM, LB_N_SRAM, connect to the FPGA, and the read and write status is controlled by the FPGA. The address is sent to the SRAM

through the address line A[17:0]. In the write state, the data to be written is sent to the SRAM through the data line D[15:0], and can be written into the register of the corresponding address; In the read state, the data in the corresponding address register can be directly read into the FPGA by the data line.

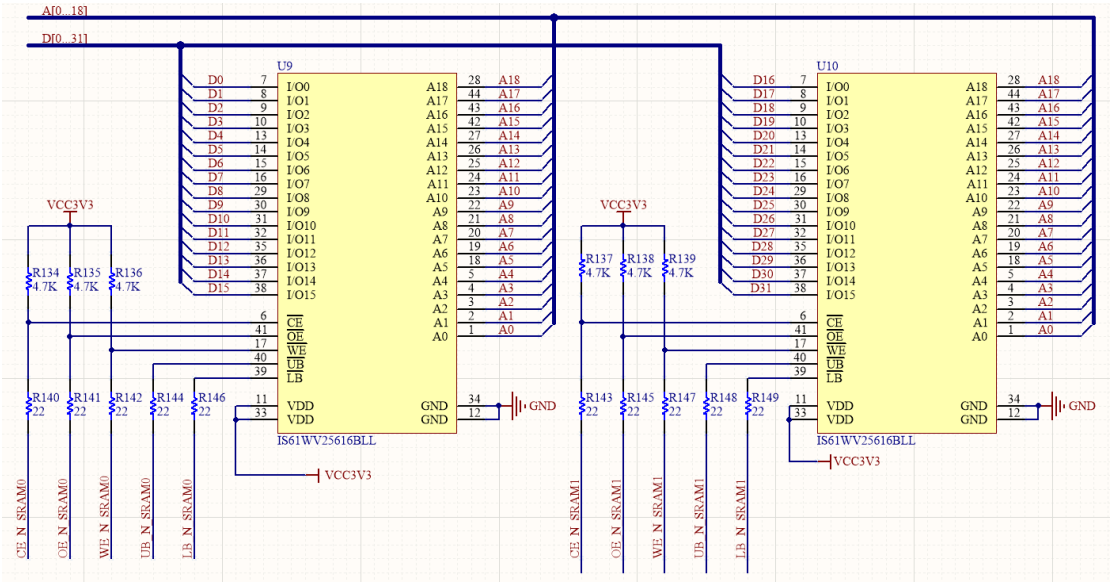


Figure 15.2 Schematics of SRAM

15.3.3 Introduction to the Program

This experiment will use the frequency division, button debounce, hex conversion and segment display module. (Refer to the previous experiment for more information) Here SRAM read and write module is mainly introduced.

The first step: the establishment of the main program framework

```
module sram (
    input           IN_CLK_50M,             //System clock on board
    input   [7:1]   PB,                     //Push buttons

    output          sram0_cs_n,             //First SRAM control signal group
    output          sram0_we_n,
    output          sram0_oe_n,
    output          sram0_ub_n,
    output          sram0_lb_n,

    output          sram1_cs_n,             //Second SRAM control signal group
    output          sram1_we_n,
    output          sram1_oe_n,
    output          sram1_ub_n,
    output          sram1_lb_n,

    output  [17:0]  sram_addr,              //sram adress signal
    inout   [31:0]  sram_data,              //sram data signal
```

```
    output          [5:0]       tube_sel,                   // Segment display control signal
    output          [7:0]       tube_seg
);
endmodule
```

The inputs are 50 MHz system clock *IN_CLK_50M*, button module PB[7:1], PB[3] (RETURN) as external hardware reset, PB[2] (UP) as write control, PB[7] (DOWN) as read control. The output has two sets of control signals to control two srams respectively, specifically chip selection signal sram_cs_n, write control signal sram_we_n, read control signal sram_oe_n, and byte control signals sram_ub_n and sram_lb_n, address bus sram_daddr[17:0], data bus Sram_data[31:0], and the segment display bit selection signal tube_sel[5:0] and the segment selection signal tube_seg[7:0].

The second step: SRAM read and write module

In this experiment, two SRAMs are used simultaneously and are expanded into a 32-bit wide data memory.

```
reg             [31:0]      wr_data;
reg                         wr_en;
reg             [3:0]       state;
reg             [7:1]       PB_flag;
reg                         wr_done;
reg                         rd_done;
reg                         s_flag;

assign      sram_data = wr_en ? wr_data : 32'hz;

always @ (posedge clk)
begin
    if (!rst_n)
        begin
            wr_done <= 1'b0;
            rd_done  <= 1'b0;
            rd_data   <= 32'd0;
            wr_data  <= 32'd0;

            sram0_cs_n <= 1'b1;
            sram0_we_n <= 1'b1;
            sram0_oe_n <= 1'b1;
            sram0_ub_n <= 1'b1;
            sram0_lb_n <= 1'b1;

            sram1_cs_n <= 1'b1;
            sram1_we_n <= 1'b1;
            sram1_oe_n <= 1'b1;
```

```
            sram1_ub_n <= 1'b1;
            sram1_lb_n <= 1'b1;
            sram_addr    <= 18'd0;

            wr_en <= 1'b0;
            state <= 4'd0;
        end
    else
        case(state)
            0    :
            begin
                wr_done <= 1'b0;
                rd_done <= 1'b0;

                sram_addr      <= 18'd511;
                wr_data        <= 32'd123456;

                if (PB_flag[2])
                    begin
                        wr_en <= 1'b1;
                        state <= 4'd1;

                        sram0_cs_n <= 1'b0;
                        sram0_we_n <= 1'b0;
                        sram0_oe_n <= 1'b1;
                        sram0_ub_n <= 1'b0;
                        sram0_lb_n <= 1'b0;

                        sram1_cs_n <= 1'b0;
                        sram1_we_n <= 1'b0;
                        sram1_oe_n <= 1'b1;
                        sram1_ub_n <= 1'b0;
                        sram1_lb_n <= 1'b0;
                    end

                else if (PB_flag[7])
                    begin
                        wr_en <= 1'b0;
                        state <= 4'd2;

                        sram0_cs_n <= 1'b0;
                        sram0_we_n <= 1'b1;
                        sram0_oe_n <= 1'b0;
                        sram0_ub_n <= 1'b0;
```

```
                    sram0_lb_n <= 1'b0;

                    sram1_cs_n <= 1'b0;
                    sram1_we_n <= 1'b1;
                    sram1_oe_n <= 1'b0;
                    sram1_ub_n <= 1'b0;
                    sram1_lb_n <= 1'b0;
                end

            else
                state<= 4'd0;
        end

        1    :
        begin
            if (sram_addr == 18'd0)
                begin
                    state<= 4'd4;
                    wr_done <= 1'b1;
                    wr_en    <= 1'b0;
                end
            else
                begin
                    state      <= 4'd1;
                    sram_addr     <= sram_addr - 1'b1;
                    wr_data       <= wr_data - 1'b1;
                end
        end

        2    :
        begin
            if (sram_addr == 18'd0)
                state<= 4'd4;
            else
                state<= 4'd2;
                if (s_flag)
                    begin
                        sram_addr     <= sram_addr - 1'b1;
                        rd_data       <= sram_data;
                        rd_done       <= 1'b1;
                    end
                else
                    rd_done  <= 1'b0;
        end
```

```
            4    :
            begin
                sram0_cs_n <= 1'b1;
                sram0_we_n <= 1'b1;
                sram0_oe_n <= 1'b1;
                sram0_ub_n <= 1'b1;
                sram0_lb_n <= 1'b1;

                sram1_cs_n <= 1'b1;
                sram1_we_n <= 1'b1;
                sram1_oe_n <= 1'b1;
                sram1_ub_n <= 1'b1;
                sram1_lb_n <= 1'b1;

                wr_done <= 1'b0;
                rd_done <= 1'b0;

                state <= 0;
            end

            default    :    state <= 0;
        endcase
end
```

In the write state, the write enable *wr_en* is pulled high. At this time, *sram_data* is the data *wr_data* to be written. In other cases, the write enable is pulled low. In the read state, the data is directly read into the FPGA by *sram_data*.

At reset, the SRAM control signals are all pulled high, then jumps to the 0 state, and the data is read and written by the state machine.

0 state: an initial address “511” is given, and an initial data “123456”, when the write enable signal *PB_flag[2]* is valid, the chip selection signal pulls down the selected SRAM. The write control signal is pulled low to prepare for write operation, and the read control signal remains pulled up. Meanwhile, the byte control signal is pulled low, indicating that the high and low two bytes of data are simultaneously written and then jump to the 1 state. When the read enable signal *PB_flag[7]* is active, in contrast to the write enable, the write control signal is held high, the read control signal is pulled low to prepare for the read operation, and jumps to the 2 state.

1 state: starting from the initial address “511”, writing initial data “123456”, each clock cycle address and data are simultaneously decremented by one, performing 512 data continuous write operations. When the register address bit is ‘0’, end the write operation and jum to the 4 state .

2 state: Starting from the initial address “511”, the address is decremented by 1 every 1 second under the control of the second pulse *s_flag*, and a continuous read operation of 512 data is performed. When the data is completely read, the address jumps to the 4 state when the address is ‘0’.

4 state: The control signals are all pulled high, deactivate the control of the SRAM, jumping to the ‘0’ state, an waiting for the next operation.

15.4 Experiment Verification

The first step: pin assignment

Table 15.1 SRAM read and write experiment pin mapping

Signal Name	Network Label	FPGA Pin	Port Description
IN_CLK_50M	CLK_50M	G21	System clock 50 MHz
PB[1]	PB[1]	Y4	7 push buttons on board
PB[2]	PB[2]	V5	
PB[3]	PB[3]	Y6	
PB[4]	PB[4]	AB4	
PB[5]	PB[5]	Y3	
PB[6]	PB[6]	AA4	
PB[7]	PB[7]	AB3	
sram0_cs_n	CE_N_SRAM0	F21	First SRAM control signal
sram0_we_n	WE_N_SRAM0	B22	
sram0_oe_n	OE_N_SRAM0	F17	
sram0_ub_n	UB_N_SRAM0	K22	
sram0_lb_n	LB_N_SRAM0	K21	
sram1_cs_n	CE_N_SRAM1	N22	Second SRAM control signal
sram1_we_n	WE_N_SRAM1	R19	
sram1_oe_n	OE_N_SRAM1	Y21	
sram1_ub_n	UB_N_SRAM1	Y22	
sram1_lb_n	LB_N_SRAM1	T18	
sram_addr[0]	A_R_0	J21	SRAM address line
sram_addr[1]	A_R_1	H22	
sram_addr[2]	A_R_2	H19	
sram_addr[3]	A_R_3	G18	
sram_addr[4]	A_R_4	H17	
sram_addr[5]	A_R_5	H21	
sram_addr[6]	A_R_6	H20	
sram_addr[7]	A_R_7	F19	
sram_addr[8]	A_R_8	H18	
sram_addr[9]	A_R_9	F20	
sram_addr[10]	A_R_10	W21	
sram_addr[11]	A_R_11	W22	
sram_addr[12]	A_R_12	V21	
sram_addr[13]	A_R_13	U20	
sram_addr[14]	A_R_14	V22	
sram_addr[15]	A_R_15	R21	

sram_addr[16]	A_R_16	U21	
sram_addr[17]	A_R_17	R22	
sram_addr[18]	A_R_18	U22	
sram_data[0]	D_R_0	F22	SRAM data line
sram_data[1]	D_R_1	E21	
sram_data[2]	D_R_2	D21	
sram_data[3]	D_R_3	E22	
sram_data[4]	D_R_4	D22	
sram_data[5]	D_R_5	C21	
sram_data[6]	D_R_6	B21	
sram_data[7]	D_R_7	C22	
sram_data[8]	D_R_8	M16	
sram_data[9]	D_R_9	K19	
sram_data[10]	D_R_10	M20	
sram_data[11]	D_R_11	M19	
sram_data[12]	D_R_12	L22	
sram_data[13]	D_R_13	L21	
sram_data[14]	D_R_14	J22	
sram_data[15]	D_R_15	J18	
sram_data[16]	D_R_16	M21	
sram_data[17]	D_R_17	K18	
sram_data[18]	D_R_18	N21	
sram_data[19]	D_R_19	M22	
sram_data[20]	D_R_20	P22	
sram_data[21]	D_R_21	P20	
sram_data[22]	D_R_22	R20	
sram_data[23]	D_R_23	P21	
sram_data[24]	D_R_24	W19	
sram_data[25]	D_R_25	W20	
sram_data[26]	D_R_26	R17	
sram_data[27]	D_R_27	T17	
sram_data[28]	D_R_28	U19	
sram_data[29]	D_R_29	AA21	
sram_data[30]	D_R_30	AA22	
sram_data[31]	D_R_31	R18	
tube_sel[0]	SEG_3V3_D0	F14	Segment display bit selection signal
tube_sel[1]	SEG_3V3_D1	D19	
tube_sel[2]	SEG_3V3_D2	E15	
tube_sel[3]	SEG_3V3_D3	E13	
tube_sel[4]	SEG_3V3_D4	F11	
tube_sel[5]	SEG_3V3_D5	E12	
tube_seg[0]	SEG_PA	B15	
tube_seg[1]	SEG_PB	E14	

tube_seg[2]	SEG_PC	D15	Segment display segment selection signal
tube_seg[3]	SEG_PD	C15	
tube_seg[4]	SEG_PE	F13	
tube_seg[5]	SEG_PF	E11	
tube_seg[6]	SEG_PG	B16	
tube_seg[7]	SEG_DP	A16	

The second step: board verification

After the pin assignment is completed, the compilation is performed, and the board is verified after passing.

After the development board is programmed, the segment display will all light up, but because no data is read, the segment display will display all '0's, as shown in Figure 15.3. Press the PB[2] (UP) button to write the data to the SRAM, and then press the PB[7] (DOWN) button to read the written data. At this time, it displays "123456" and decrement by one every second. See Figure 15.4. From this it is verified that the specified data is written into the SRAM and is read correctly.

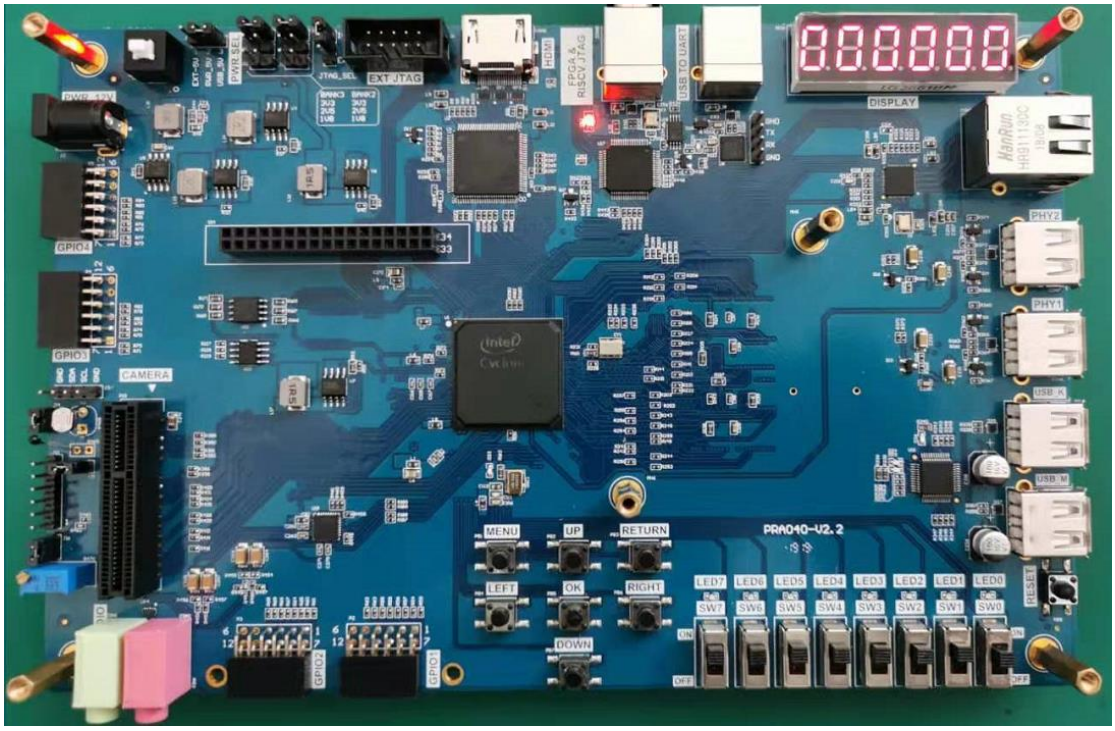


Figure 15.3 SRAM write and read 1

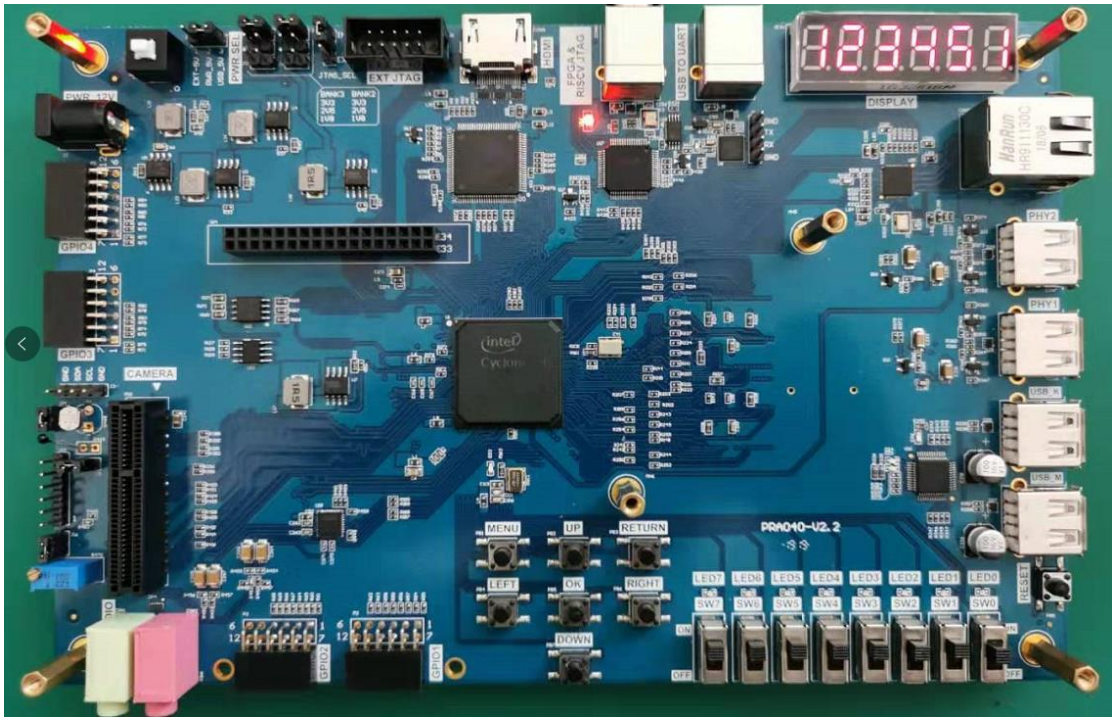


Figure 15.4 SRAM write and read 2

References:

1. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_usb_blstr_ii_cable.pdf
2. https://www.nxp.com/docs/en/data-sheet/PCF8591.pdf
3. https://www.analog.com/media/en/technical-documentation/user-guides/ADV7511_Hardware_Users_Guide.pdf
4. https://www.mouser.com/ds/2/268/atmel_doc0180-1065439.pdf
5. https://www.verical.com/datasheet/realtek-semiconductor-phy-rtl8211e-vb-cg-2635459.pdf'
6. https://www.mouser.com/ds/2/76/WM8978_v4.5-1141768.pdf