Pocket Board

FII-PRA006

Experiment Manual

Fraser Innovation Inc
June 25, 2019

Content

Project Files Content

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: vga

Experiment 1 LED_shifting

1.1 Experiment Objective

1. Practice to use development software Quartus II, including projects, system resources IP core;
2. Proficiency in the writing of Verilog HDL, develop a fine writing style;
3. Master the design of the frequency divider to realize the design of led shifting;
4. Mange FPGA pin assignment according to the hardware resources;
5. Observe the experiment result and summarize.

1.2 Experiment Requirement

1. Light all the LEDs when resetting;
2. After resetting, all the LEDs blink from right to left (low to high);
3. Each led is lit for 1 second;
4. After the last left led blinks, the most right led continue to blink, to create a blink loop.

1.3 Experiment

1.3.1 Project building

Take Quartus II 18.0 version as an example, the actual steps have been shown in figures.

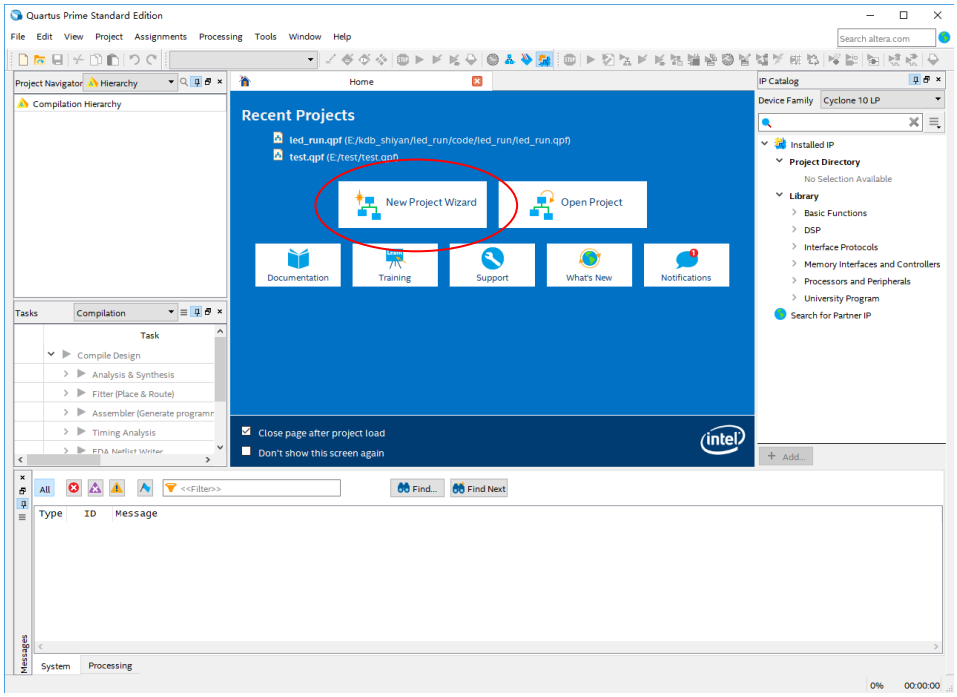

Fig 1.1 The main Quartus II display

A. In Fig 1.1, you could start a new project by clicking the **New Project Wizard** at the software

center, or going to **File > New Project Wizard**, or the shortcut **Ctrl + N**.

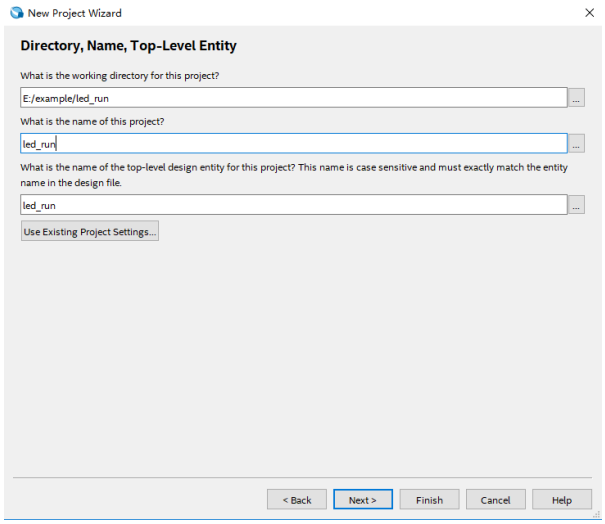

Fig 1.2 choose the directory

B. In Fig 1.2, set the working directory to be *example/led_run*. The project must have a name, a related and easy name is suggested for future use and invoke. Choose *led_run* for the name. Since we have not yet created the directory, Quartus II software displays the pop-up box in Fig 1.3 asking if it should create the desired directory. Click **Yes**, which leads to the window in Fig 1.4. Choose **Empty project** and click **Next**.

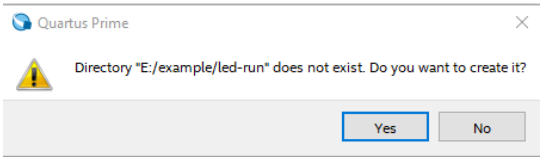

Fig 1.3 Quartus II can create a new directory for the project

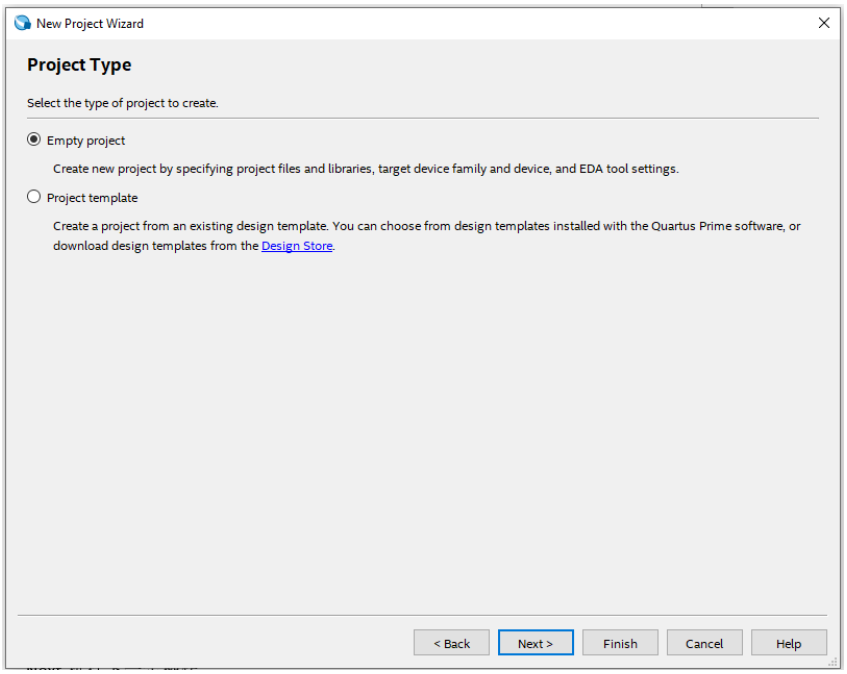

Fig 1.4 Select project type

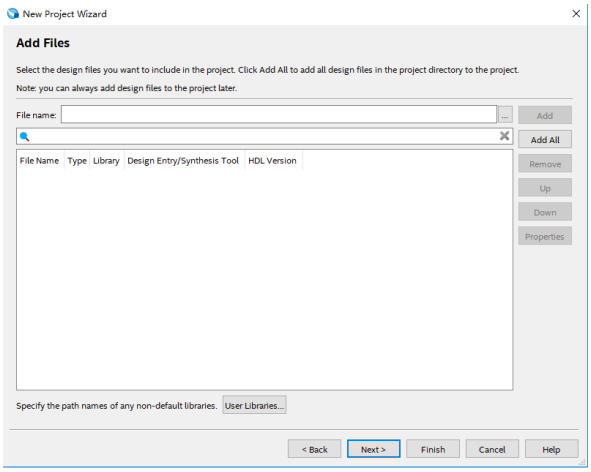

Fig 1.5 Add files

C. In Fig 1.5, you could add existing files (if any) to the project. Here, we click **Next.**

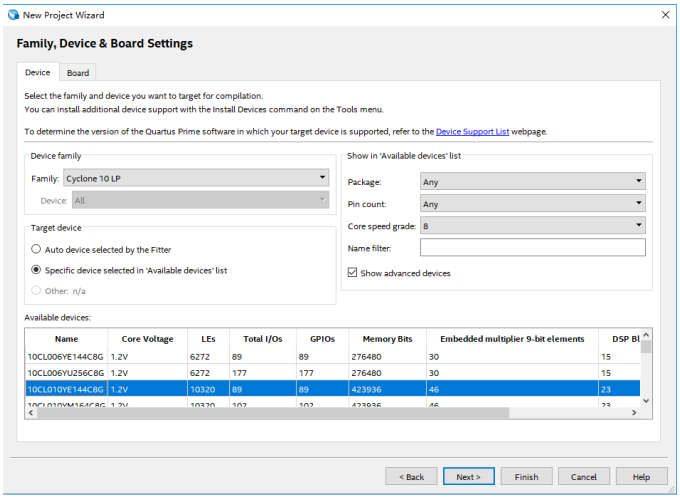

Fig 1.6 Choose the device family and a specific device

D. In Fig 1.6, choose **Cyclone 10 LP** for the family, **Specific device selected in 'Available devices' list** in target device and choose **10CL010YE144C8G** chip in available devices. Click **Next**.

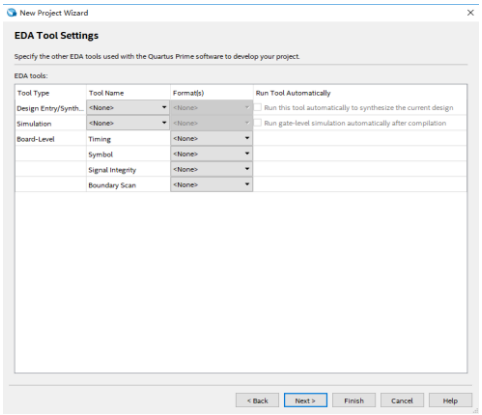

Fig 1.7 EDA tools selection

E. In Fig 1.7, some EDA tools are available. Here, we use default setting. Click **Next** and then

Finish to finish the project building.

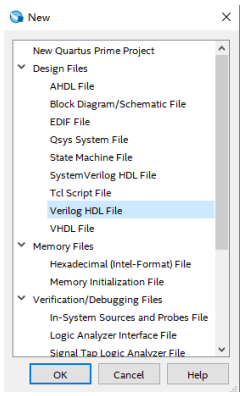

Fig 1.8 Select the new file type

F. In Fig 1.8, choose **File > New**, **Verilog HDL File** and then click **OK**. To make the file name and project name consistent, click **File > Save As**, for the file name, use *led_run*, the type should be **Verilog HDL Files**. Remember to save it under the right directory.

1.3.2 PCB Schematics

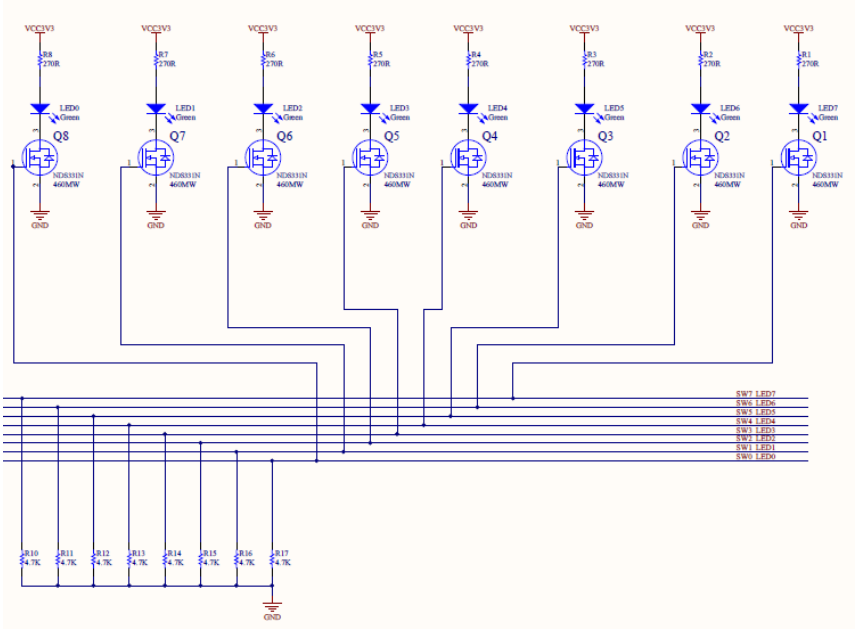

Fig 1.9 PCB schematics for the LEDs

In Fig 1.9, all the LEDs share the same high anodes, so when an external low voltage is given in cathodes, LEDs are lit up.

1.3.3 Experiment Procedure

Step 1: Main Verilog HDL code block

module LED_shifting (clk, rst, led);

```
    input    clk,  rst;
    output   [7:0]    led;
endmodule
```

The input has *clk* and *rst*. *clk* is 50 MHz in this case. *rst* is to reset (We use PB1 on board as our reset key). 8 LEDs are defined as a vector 7 downto 0, to save the pin resources. Enter the main code to the *led_run* Verilog HDL file we just made.

Step 2: Invocation for IP core, building and using PLL module

1. In Fig 1. 10, find **IP Catalog** in the right side of the main interface. Click **Library > Basic Functions > Clocks; PLLs and Resets > PLL > ALTPLL**

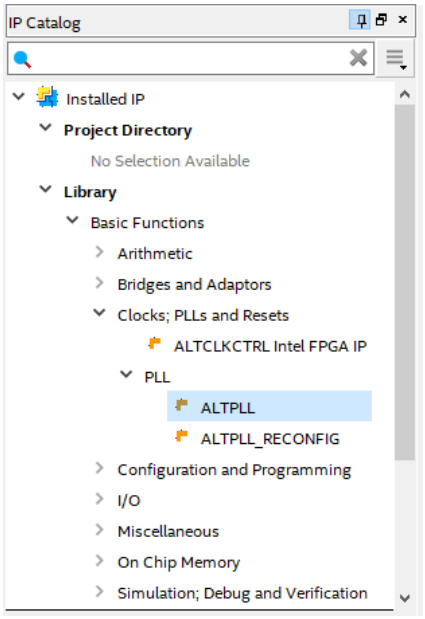

Fig 1. 10 IP Catalog

2. Double click **ALTPLL**, and name PLL module. Here, we use *PLL1*, and make sure file type is **Verilog**, click **OK**. See Fig 1. 11.

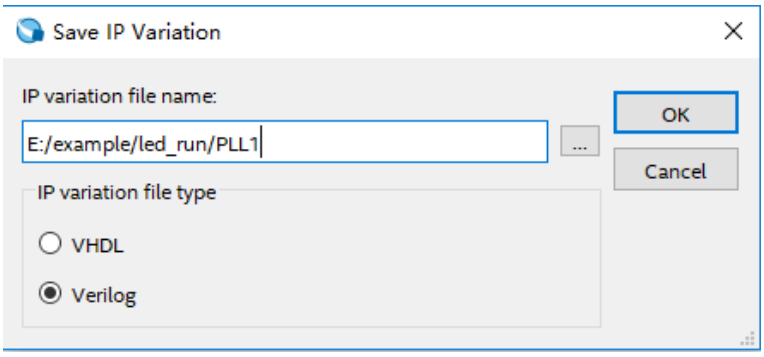

Fig 1.11 Name PLL module

3. In Fig 1. 12, PLL setting interface had popped up. **Inclk0** is the input clock of PLL, provided by the original board. It should be consistent with the system clock, to be **50 MHz.** Set **In normal mode** for the feedback path inside the PLL, and **c0** is the output clock. Click **Next**.

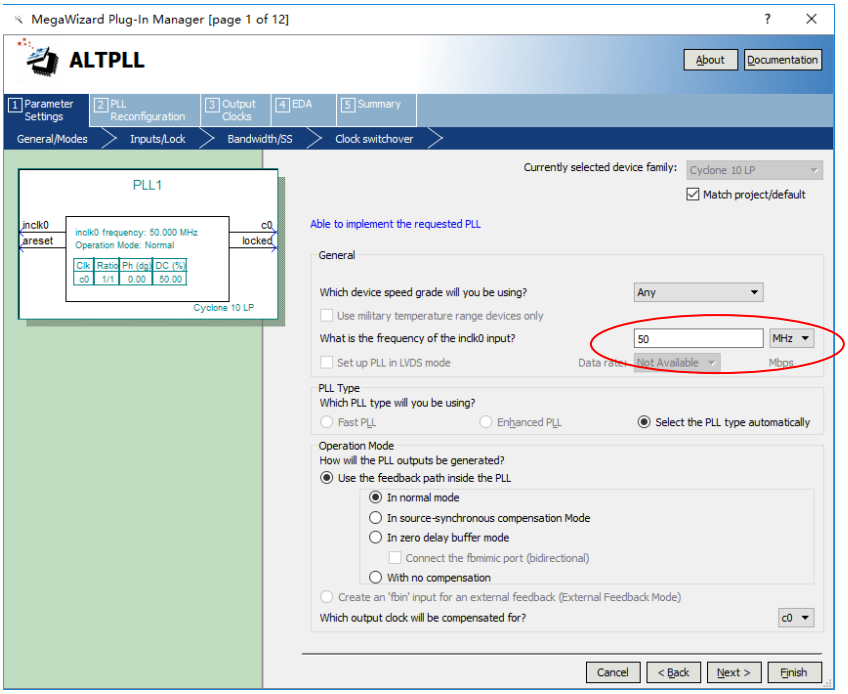

Fig 1. 12 PLL setting 1 input clock

4. In Fig 1. 13, optional inputs and lock output are for selecting. Here, we use the default setting.

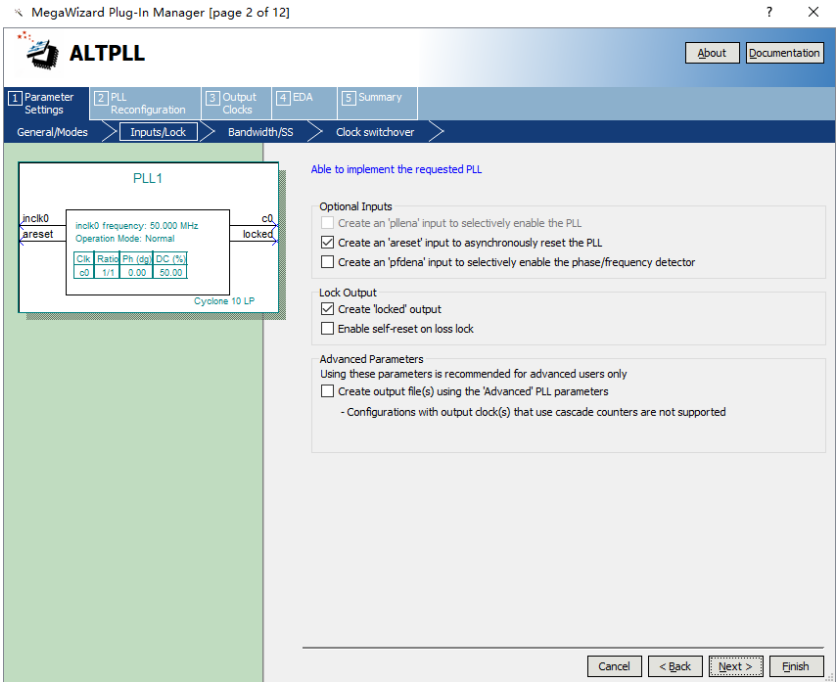

Fig 1. 13 PLL setting 2

5. Click **Next** in the next 3 steps. PLL Reconfiguration default setting are used.
6. In Fig 1. 14, Output Clocks are set. In total, 5 different clocks **clk c0** – **clk c4** are available. Here, only **clk c0** is needed. Click **Use this clock** only for c0. Check the box **Enter output clock frequency**, set the frequency to be **100 MHz**. Make sure the **Clock phase shift** is **0** degree, and the **Clock duty cycle** is **50** (%).

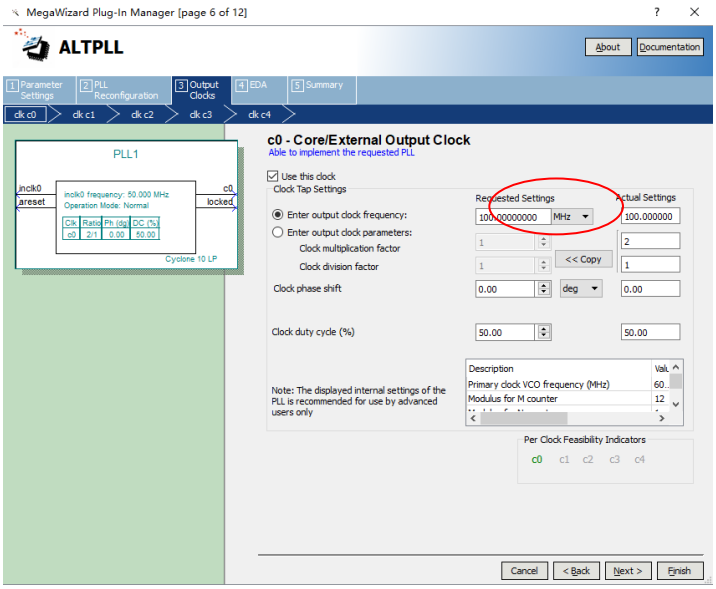

Fig 1.14 PLL setting 3, output clock

7. Use the default EDA setting. Click **Next**.
8. In Fig 1.15, select the output file type ***.bsf** (will be used in the future when designing the graphic symbol design), remain the other to be the same. Click **Finish**.

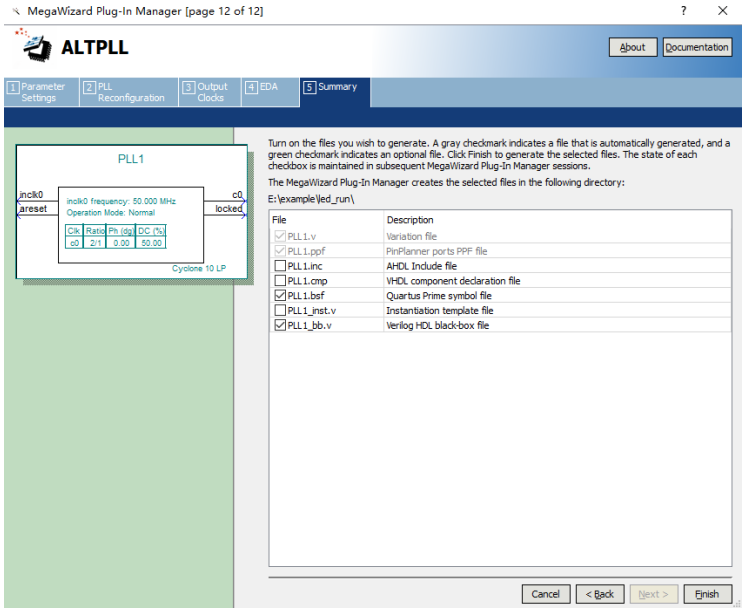

Fig 1. 15 Set the output file type

9. In Fig 1.16, choose **Files** in the drop-down menu of **Project Navigator** (by default is **Hierarchy**).

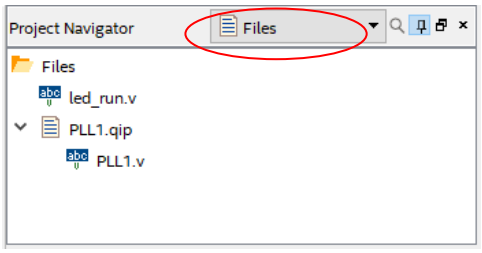

Fig 1.16 The location of PLL in Files

10. In Fig 1. 17, click **PLL1.v**, the main interface will display the code for PLL, find the module name and port list, copy them to the top level file (*led_run.v*), and instantiate it.

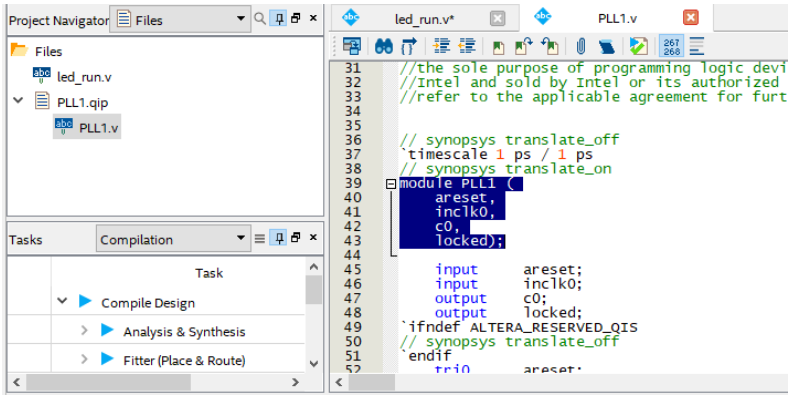

Fig 1.17 Module and port list of PLL1

11. Refer to the actual project files *LED_shifting* attached, adjust port allocation in top level file. Sys_rst is 1 before PLL, as the reset signal for the whole system. After the whole system gets locked (pll_locked == 1'b1), sys_rst is 0, and the register is driven by the rising edge of sys_clk, so it is the synchronous reset signal.

Step 3: The design of the frequency divider (code can be found in the attached project files)

100 MHz clock output by the PLL is used for the system clock. The LED light blinking period is 1 second after the frequency division.

1. Microsecond frequency division

   First period of 100 MHz clock is 10 ns, 1 us needs 100 clock cycle. A register *[7:0] us_reg* is defined.

2. Millisecond frequency division

   Since 1 ms = 1000 us, a *[9:0] ms_reg* is defined.

3. Second frequency division

   Since 1 s = 1000 ms, a *[9:0] s_reg* is defined, and a second pulse signal *s_f.* Only after these three registers are counted full at the same time, it is 1 s, and a second pulse signal is sent.

Step 4: Blinking led design

After pressing reset, all the LEDs are lit. The LED output is 8'hff, and then the LED will blink one by one from the right most (lowest). The LED output is 8'b0000_0001, after received the pulse signal, the LED output will become 8'b0000_0010. It seems like the high voltage logical shifts left. This could be implemented by bit splicing, that is, *led <= {led [6:0], led [7]}*.

Step 5: Verification

Fig 1. 18, Program simulation

In Fig 1.18, click the icon to compile the program, or use the shortcut **Cltr + K**. The third icon on the left is **Pin Planner**. The second one from the right is **Programmer**. A compilation report will be generated after finishing compilation, shown in Fig 1. 19.

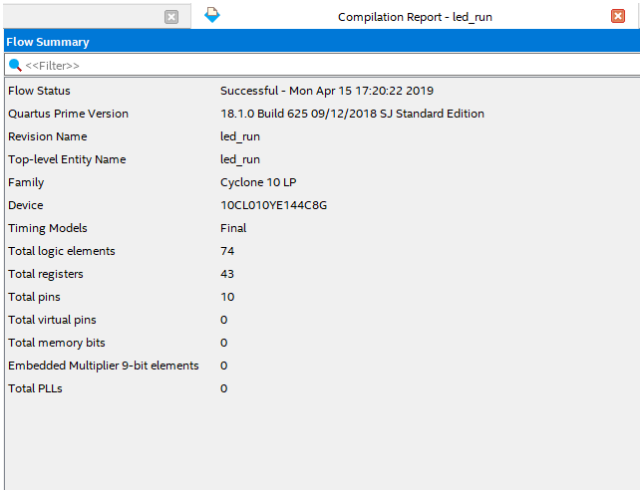

Fig 1. 19 Compilation report

After debugging, you could download the program to the board. But before that, remember to do the pin assignment in the **Pin Planner** by other clicking the icon stated above or go to **Assignments > Pin Planner**. See Fig 1. 20. More available for reference in attached project files.

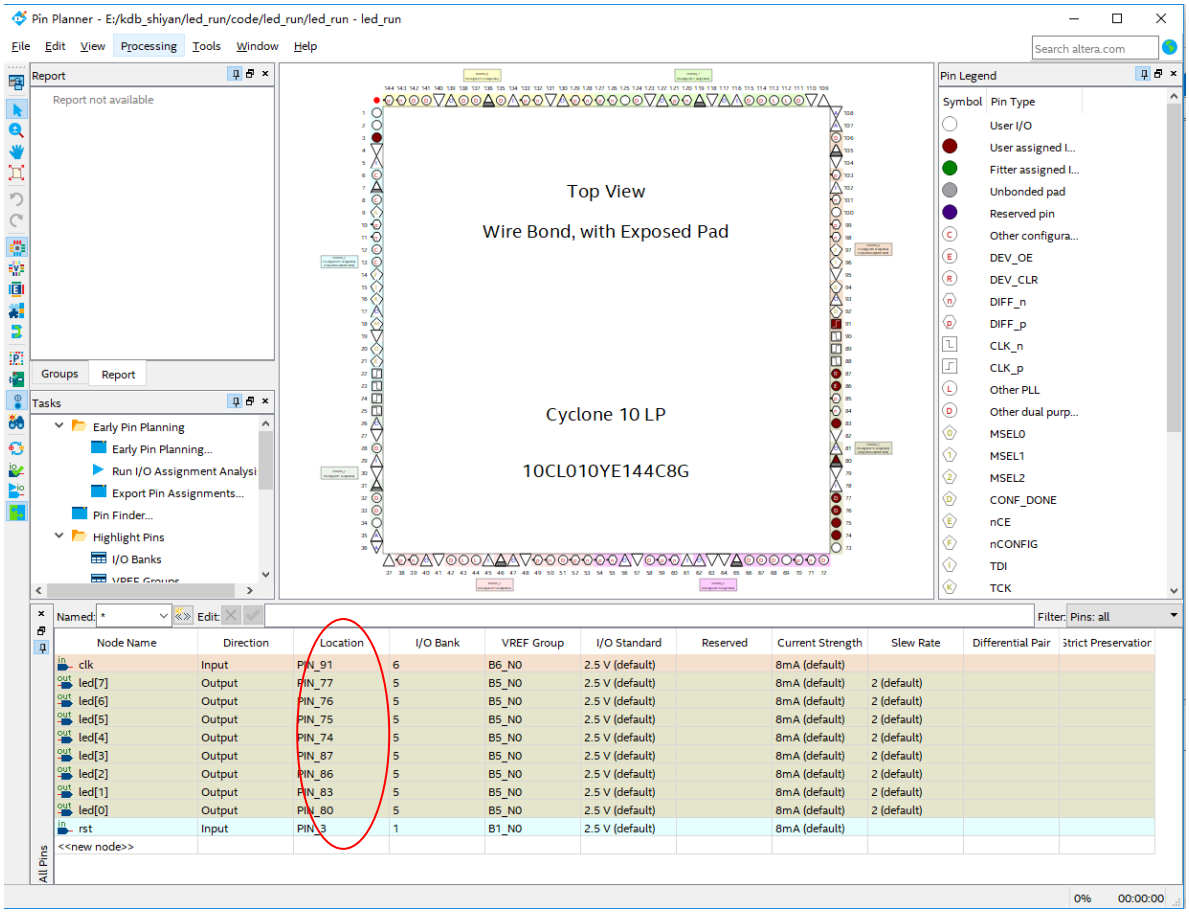

Fig 1. 20 Pin assignment

After successfully downloading the program to the board, when you press PB 1, you should see all the LEDs are lit, after releasing, the LEDs are blinking one after the other from low to high.

Experiment 2 Switch and Use SignalTap II

2.1 Experiment Objective

1. Continue to practice using the develop board
2. Use SignalTap II Logic Analyzer in Quartus II
3. Use FPGA configuration memory to program

2.2 Experiment Requirement

By using SignalTap II, learn to analyze and capture the experimental signals.

2.3 Experiment

2.3.1 Project Building

Refer to Experiment1, the following experiment project building steps will be eliminated.

2.3.2 PCB Schematics

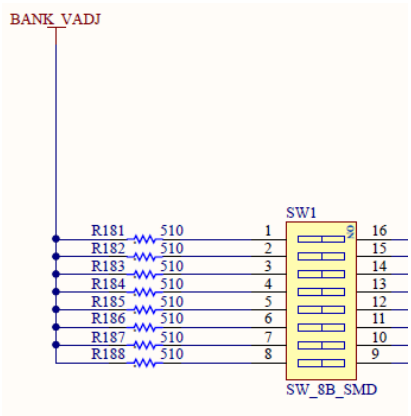

Fig 2. 1 Switch schematics

2.3.3 Experiment Procedure

We include the PLL1 generated in Experiment 1

Verilog HDL code is as follows:

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

wire    sys_clk;
wire    pll_locked;
reg     sys_rst;
```

```
always@(posedge sys_clk)
    sys_rst<=!pll_locked;

    always @(posedge inclk)
      if(sys_rst)
        led<=8'hff;
      else
        led<=~sw;


    PLL1 PLL1_INST
        (
          .areset    (1'b0),
          .inclk0    (inclk),
          .c0        (sys_clk),
          .locked    (pll_locked)
                     );
endmodule
```

2.3.4 SignalTap II Logic Analyzer

Step 1: SignalTap II startup and basic setup

Tools > Signal Tap Logic Analyzer,

1. In Fig 2.2, enter the **setup** interface
2. In **JTAG Chain Configuration**, click **setup** to set the same type as the downloader
3. Set the scan chain type
4. Set the **SOF Manager**, choose the *.sof file generated in Experiment 1

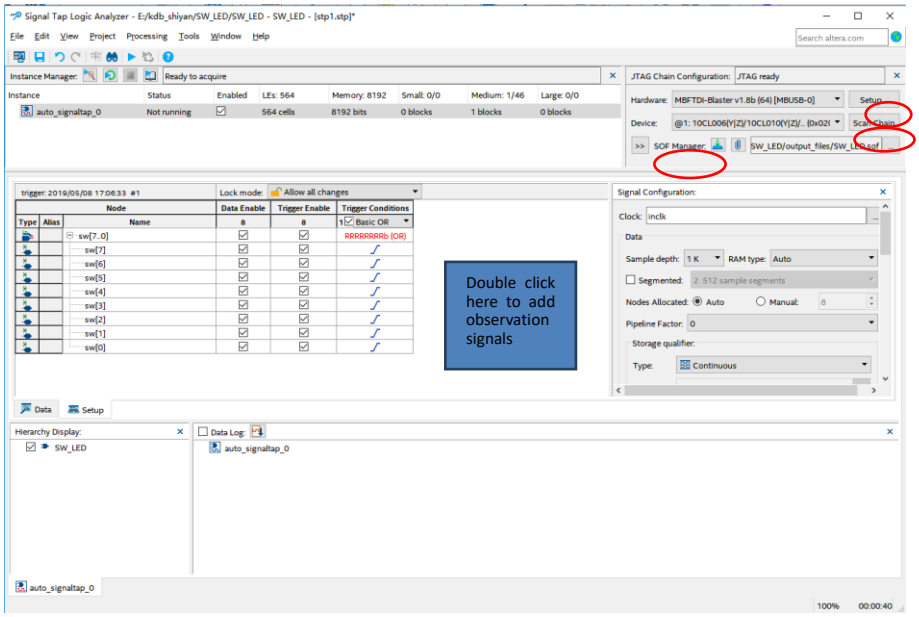

Fig 2. 2 SignalTap II setup interface

5. Clock setting. See Fig 2. 3

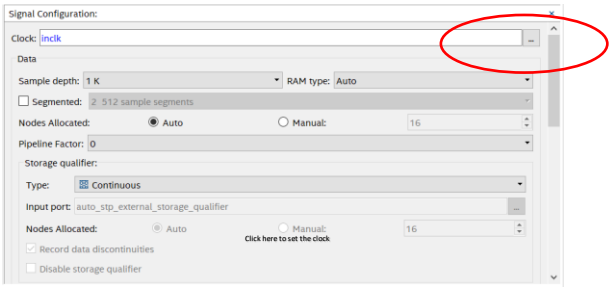

Fig 2.3 Clock, trigger and data depth setting

6. In Fig 2. 4, in the popup window, choose **SignalTap II: pre-synthesis** for **Filter**, in **Matching Nodes** column, go to **PLL1: PLL_INST**, select **c0**, and click **>** to move it to the right frame.

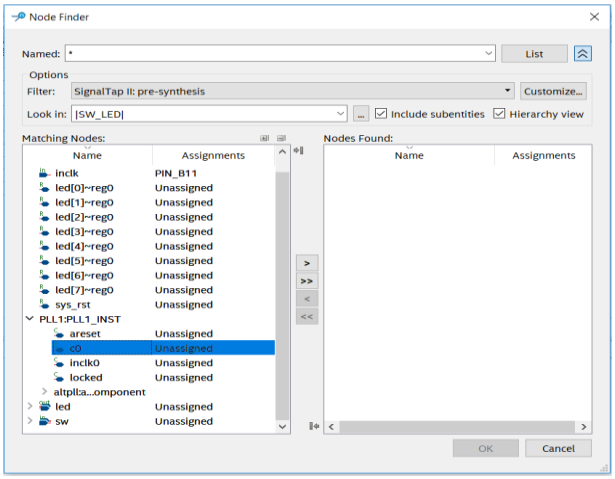

Fig 2. 4 Clock signal selection dialog boxes

Other settings are shown in Fig 2. 2. For furthermore reference, see the attached file for help.

Step 2: Add the observation signals

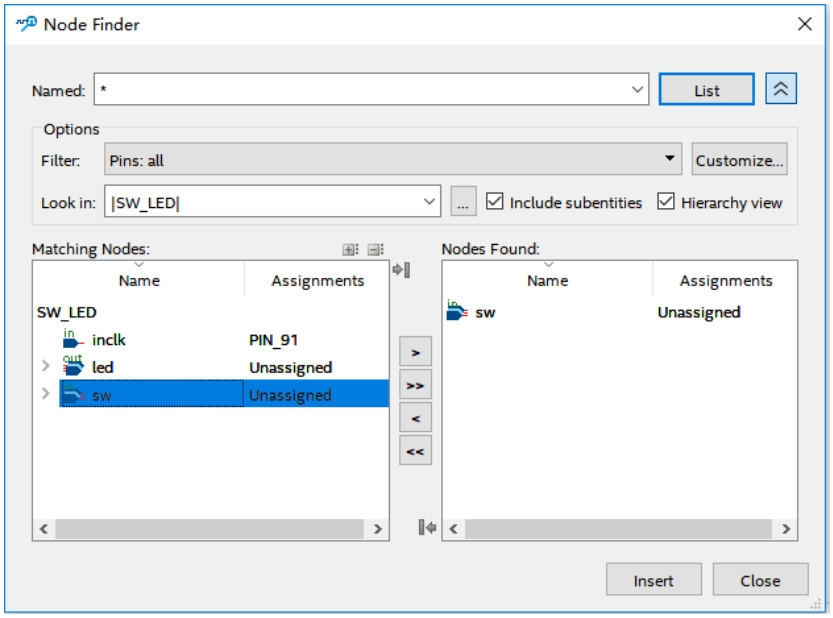

Fig 2. 5 Adding observation signal

In Fig 2. 2, double click any blank space to add the observation signals. The interface is shown in Fig 2. 5, choose the signal you want to observe on the left side, click **>** to add them to the right side, and then click **Insert**. **Save** it and recompile later.

Step 3: Set the observation signals

For the observation signals, some settings are still needed, such as whether it is a **Rising Edge** trigger, a **Falling Edge** trigger, or **Don't Care**, etc. They need to be adjusted manually. See Fig 2. 6.

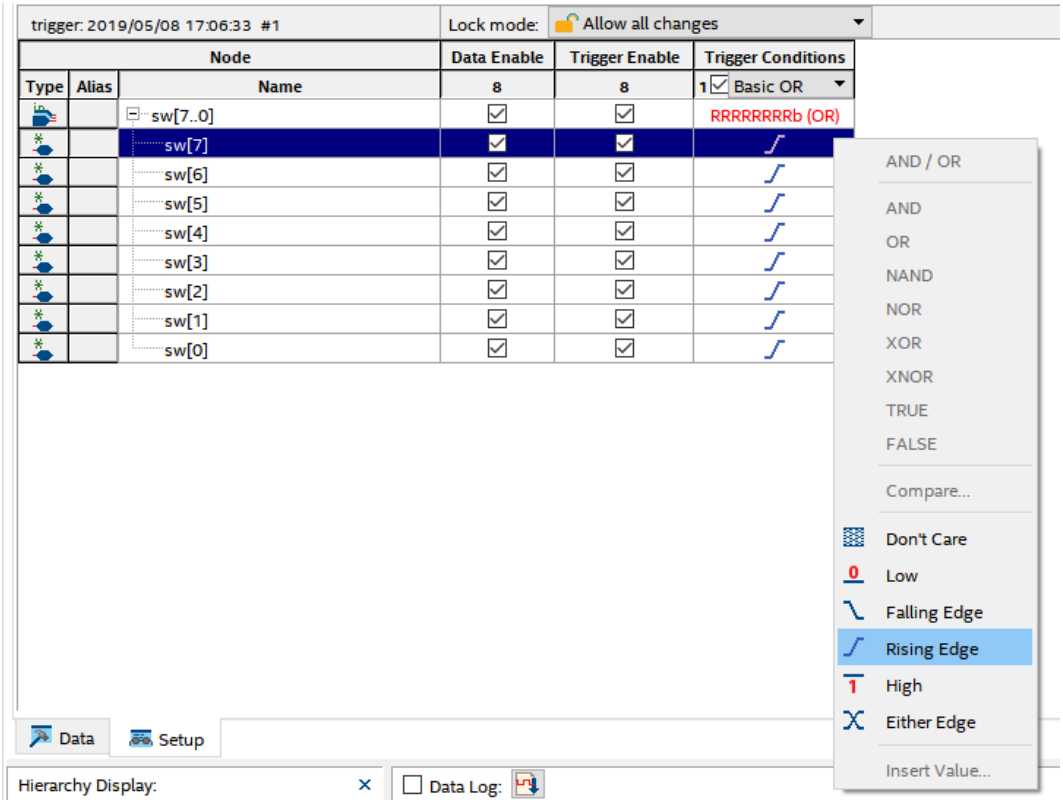

Fig 2. 6 Setting for the trigger signal

Step 4: Observe the result

In Fig 2. 7. Run the analysis and observe the SignalTap II output.

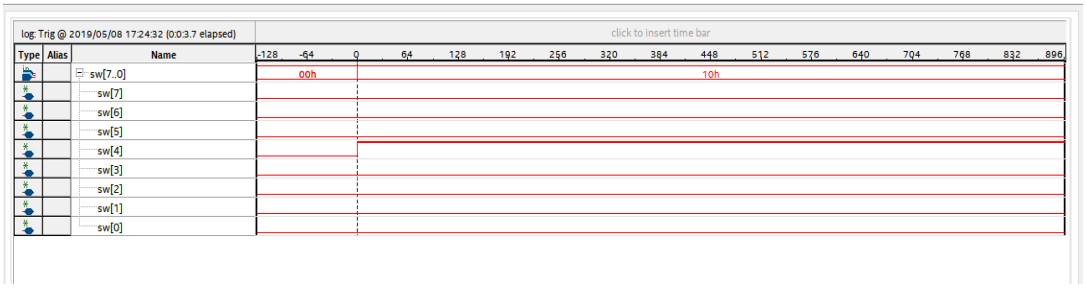

Fig 2. 7 Testing result

After analysis, when the switch SW [4] is on, the signal is high, and the corresponding LED will be lighted. You could change the trigger type and observe different outputs. Analyze the result and organize it.

Experiment 3 BCD_counter

3.1 Experiment Objective

1. Review Experiment 1, the setting for PLL, design of frequency division, and the compilation of the code
2. Study the BCD code counter
3. Design of 7 segment decoder
4. Download the program into the board flash memory

3.2 Experiment Requirement

1. The highest two segment decoders display hours, the middle two are for minutes, and the lowest two display the seconds.
2. The decimal point will be off all the time. It will not be considered in this case.

3.3 Experiment

3.3.1 Build New Project

See Experiment 1

3.3.2 PCB Schematics

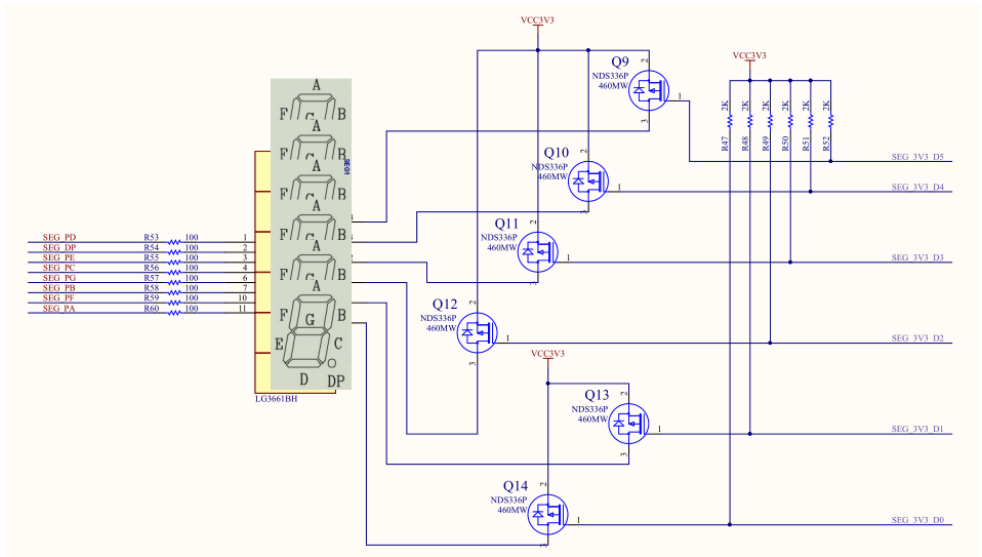

Fig 3. 1 PCB schematics for 7 segment decoders

In Fig 3. 1, six 7 segment decoders are used in this experiment. Some points need to be paid attention.

1. The segment names are shown above. A, B, C, D, E, F, and G correspond to the digital tube, while DP stands for the decimal point. D0, D1, D2, D3, D4, and D5 (on the rightest part) are for the current driver.
2. They are common anode segment decoders, D0- D5 are set high or low to control the segment decoders.
3. For the main segments, A- G are lit when the input is low, that is, '0' is high.
4. The segment decoder code is given as follows

```
always @ (*)
    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_0011;
        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

always @ (posedge sys_clk)
    seven_seg<={1'b1,seven_seg_r};
```

5. Dynamic Scanning for human eyes. Since human eyes have visual persistence characteristics, the speed to illuminate the segment decoders is fast enough that it cannot be distinguished by naked eyes. Therefore, it can be viewed as consistent lighting instead of blinking. Like the running LED, one decoder is lit at one time, implemented in the form of low-level loop left shift.

3.3.3 Experiment Procedure

1. For the frequency division design, see Experiment 1.
2. Dynamic scanning is implemented by the state machine. The relationship of conversion should be considered.
3. The code implementation of one-to-one segment should be precise.
4. Check the pin assignments before downloading the program to the board. Pin assignment file can be referred in the reference file.

3.3.4 Configuration Serial Flash Programming

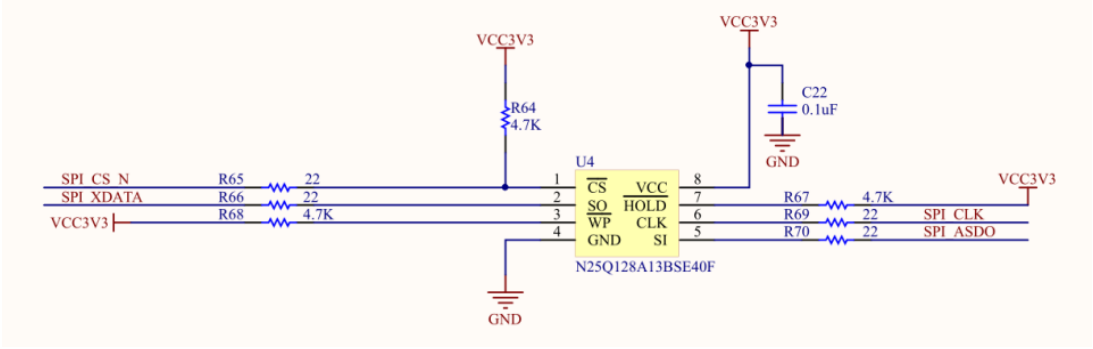

Fig 3. 2 Flash schematics

In Fig 3. 2, the functionality of Flash is to save the uploaded program even after the power is turned off. If the power is on next time, the program can be running on the board immediately. It is a very useful characteristic. The specific configuration process is as follows:

1. **File > Convert Programming Files**, as shown in Fig 3. 3
2. Option setting:
    a. In **Programming file type**, choose **JTAG Indirect Configuration File(.jic)**
    b. For **Configuration device**, choose **EPCQ 128A** (compatible development board N25Q128A)
    c. In **Mode**, choose **Active Serial**

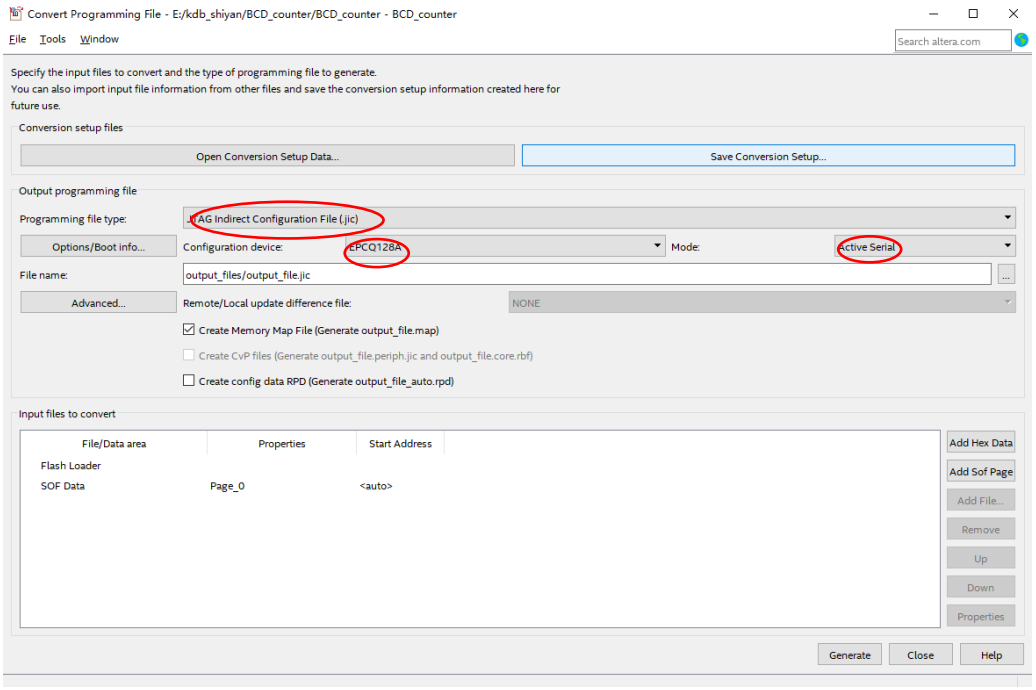

Fig 3. 3 Set .jic file

3. Click **Advanced** and set as in Fig 3. 4

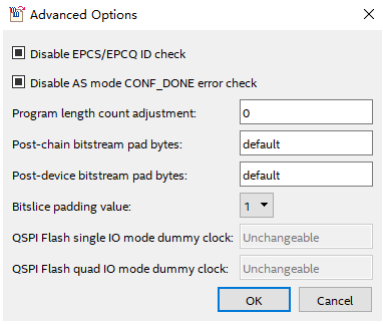

Fig 3. 4 Set the advanced option

4. Click **SDF Data**, and then **Add File**, find ***.sof** file in the **output_files**. See Fig 3. 5

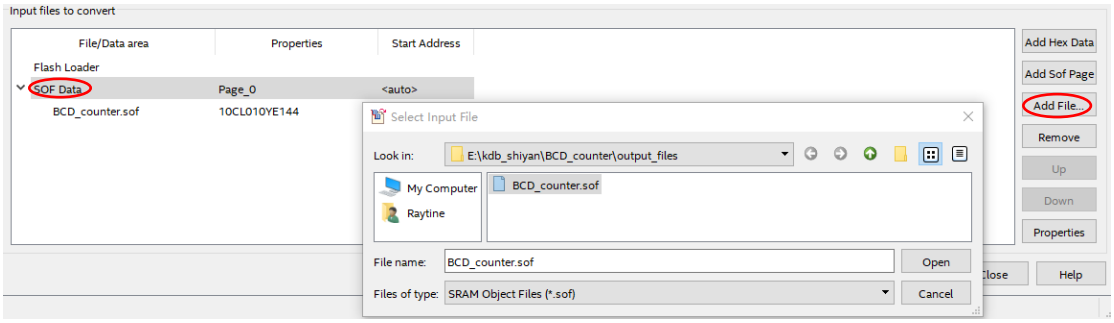

Fig 3. 5 Add a conversion file

5. Add device. See Fig 3. 6
6. Click **Generate**, and BCD_counter.jic file will be generated.

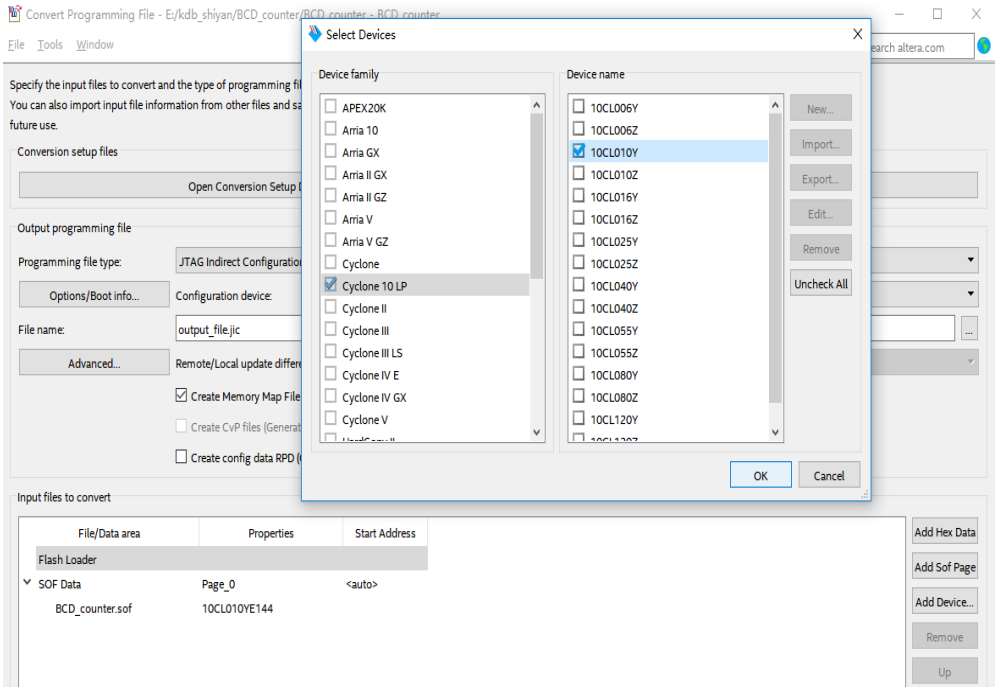

Fig 3. 6 Add device

7. Follow the same downloading procedure in previous experiments. Observe the segment decoder on develop board.

Experiment 4 Block/ Schematic Test

4.1 Experiment Objective

1. Review the new project building, PLL setting, Verilog HDL's tree hierarchy design, use of SignalTap
2. Use graphics method top-down design
3. Combine the BCD_counter design to realize the display of the digital clock
4. Observe the experiment result

4.2 Experiment

1. Build new project named **block_counter**

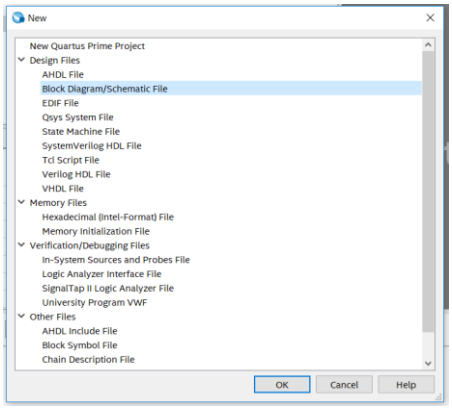

Fig 4. 1 New file selection

2. In Fig 4. 1, choose **Block Diagram/Schematic File** this time instead of Verilog HDL file.
3. In Fig 4. 2, the middle blank part is for designing block diagram or schematics.
   a. Save the file as *block_counter.bdf.*
   b. Double click the bank space to add the symbol.

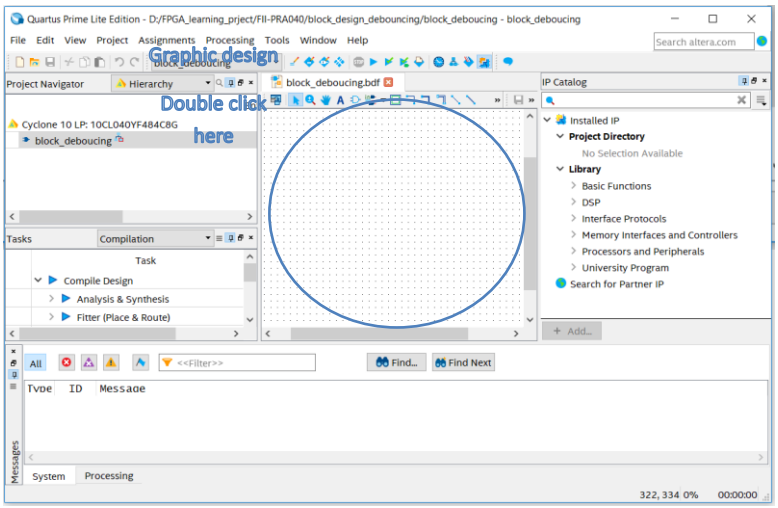

Fig 4. 2 Block diagram/ schematic design interface

4. In Library, find **c:/**, and the **primitives**, or just simply type the symbol name in the search

box.

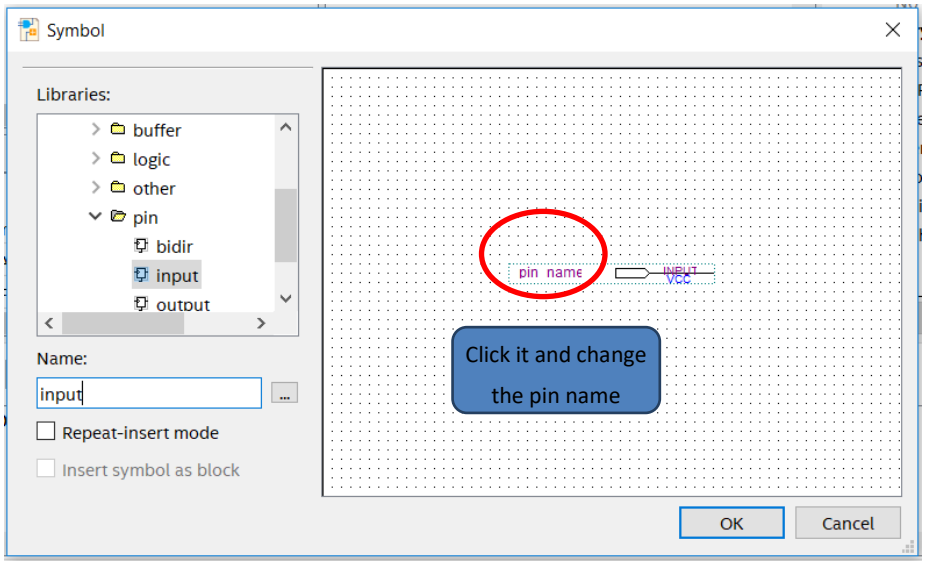

Fig 4. 3 Input symbols

5. Add **input** and **output** and modify their names. See Fig 4. 3
6. Add a customized symbol
   a. Add a new **Block Diagram/Schematic File**. Save the file as *PLL_sys.bdf*
   b. Create a new PLL I referring to Experiment 1.
   c. Select the new generated file to include *PLL1.bsf* file.
   d. Double click blank space in *PLL_sys.bdf*, and choose under **Project**, add **PLL1**. See Fig 4. 4
   e. Continue to add other symbols, such as **input**, **output**, **dff**, **GND** etc. Remember to modify their names. You could choose the **Orthogonal node tool** icon to wire them. See Fig 4. 5

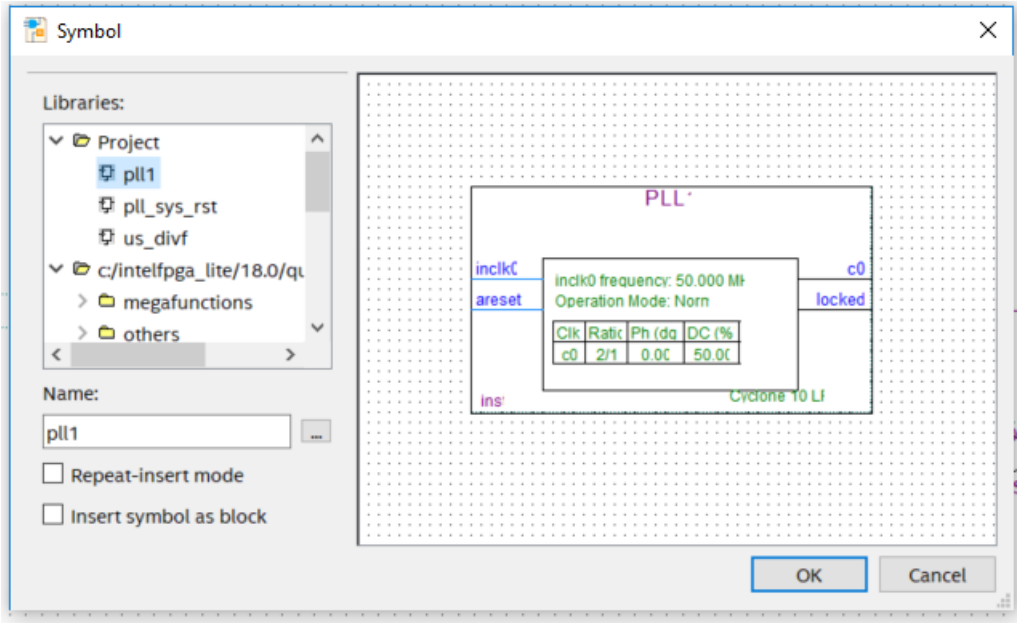

Fig 4. 4 Invoke the customized symbol

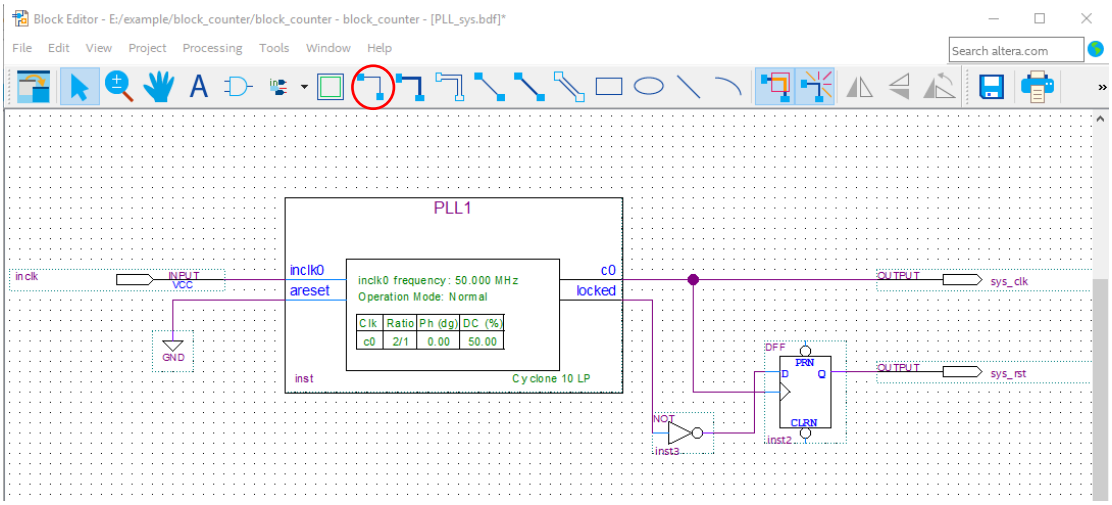

Fig 4. 5 Symbol wiring

7. Create the symbol for the new file
   a. **File > Create/ Update > Create Symbol Files for Current File**. See Fig 4. 6
   b. Save as *PLL_sys.bsf*

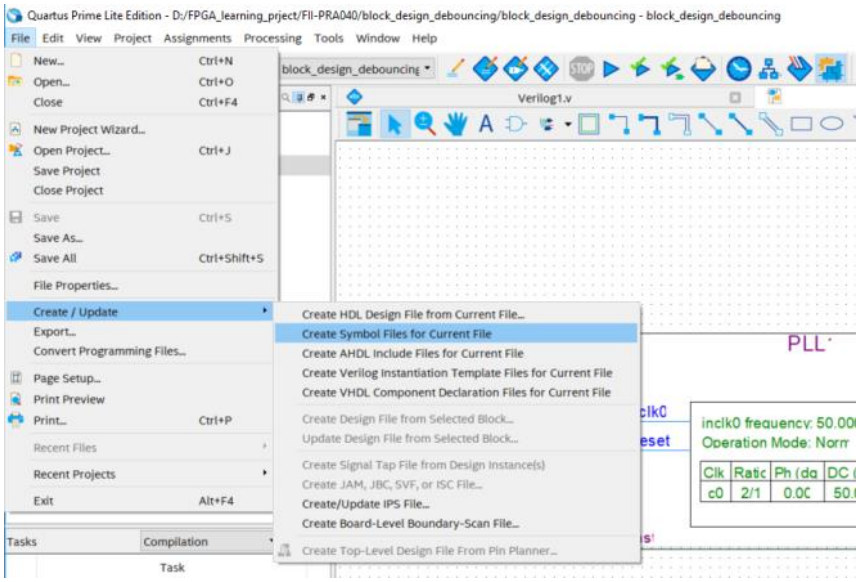

Fig 4. 6 Create symbol files for current file

8. Create a new **Verilog HDL** file for the frequency division (see reference project files)
   a. Create a new frequency divider Verilog HDL file named **div_us**
   b. Set PLL output clock as its own input clock. Divide the clock of **100 MHz** into a clock of **1 MHz**.
   c. **Repeat** step 7, create *div_us.bsf*
   d. Create a new Verilog HDL file with a frequency of 1000: *div_1000f.v*.
   e. Create a *div_1000f.bsf* symbol
9. Create a graph of us, ms, and second output pulse files for testing. See Fig 4. 7
   a. Create a new **Block Diagram/Schematic File**, and add this generated symbol to *block_div.bdf*.
   b. Repeat step 7 and create symbol file for *block_div.bdf* as well.

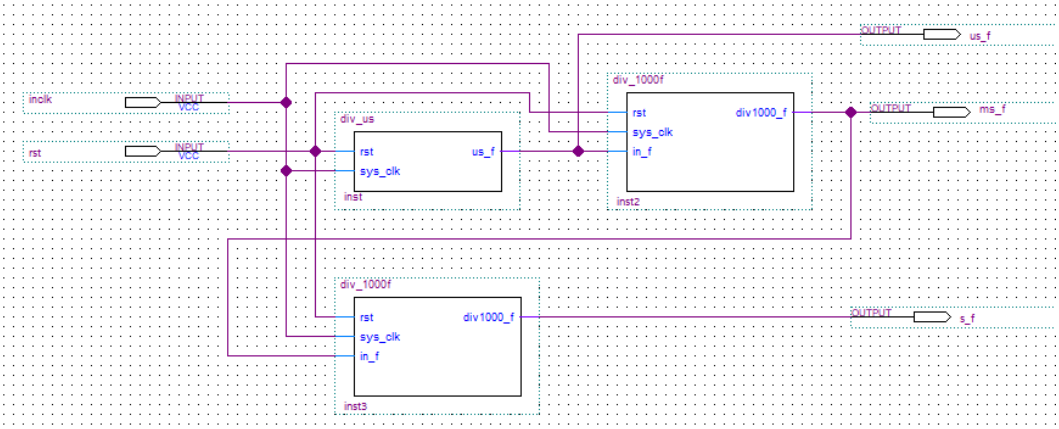

Fig 4. 7 us, ms, second pulse

10. Create a new Verilog HDL file named *bcd_counter.v*. Design time, minute counter and create bsf symbol. Refer Experiment 3 and implement some of the division using **block_div**.
11. Combine each *.bsf and complete the design of the digital clock (*block_counter.bdf*). For the output, use **Orthogonal Bus Tool** to wire. See Fig 4. 8

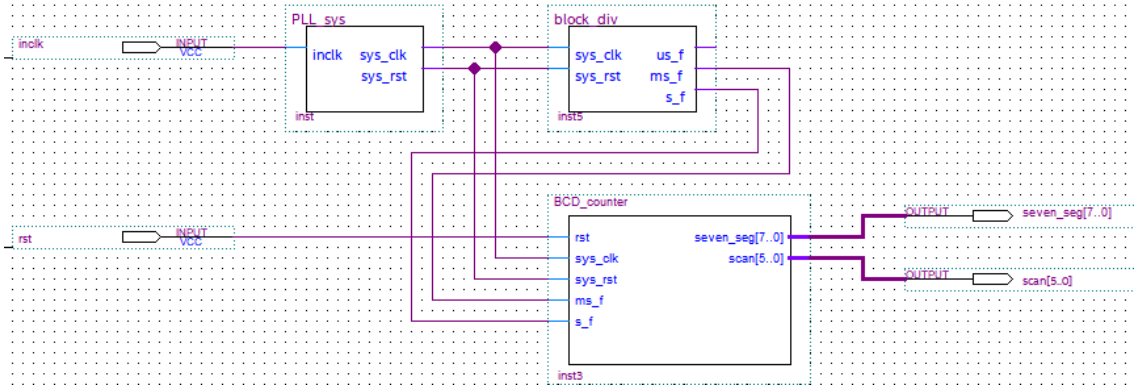

Fig 4. 8 BDF designed digital clock

12. Assign pins, and program it. For the board downloading, you can refer to Experiment 3.

Experiment 5 Block_debouncing

5.1 Experiment Objective

1. Review the design process of running LED
2. Learn the principle of button debounce and designing of adaptive programming
3. Learn the connection and used of the Fii-PRA010 button
4. Integrated application of button debounce, and furthermore development design

5.2 Experiment

1. Bouncing button principle

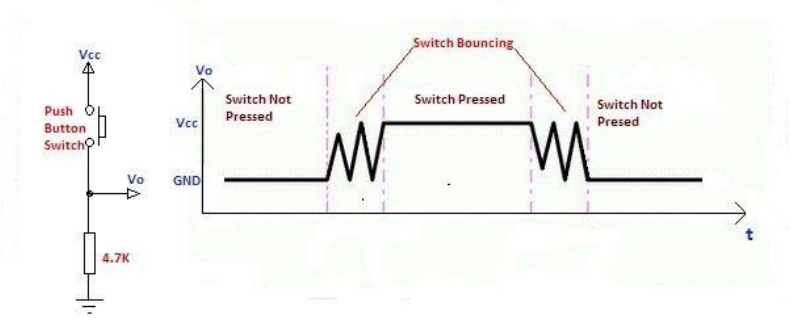

Fig 5. 1 Button bounce principle

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. 1

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.

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.

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. 2.

2. Code for button debouncing

Verilog code is as follows：

```
module pb_ve(
 input     sys_clk,   // 100Mhz
 input     sys_rst,   //
 input     ms_f,      //
 input     keyin,     // Input status of the key
 output    keyout     //Output status of key. Every time releasing the button, only one system
                      //clock pulse outputs
);

reg keyin_r;    //Input latch to eliminate metastable
reg keyout_r;//
//push_button vibrating elemination
   reg     [1:0]     ve_key_st;          //State machine status bit
   reg     [3:0]     ve_key_count;       //delay counter

            always@(posedge sys_clk)
            keyin_r<=keyin;              // Input latch to eliminate metastable

always@(posedge sys_clk)
      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) begin
```

```
                ve_key_st          <=2;
              end
            else if(ms_f)
              ve_key_count<=ve_key_count+1;
        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;        //After releasing debounce, output a
                                             //synchronized clock pulse
                     end
                    else if(ms_f)
                      ve_key_count<=ve_key_count+1;
              end
        end
default:;
      endcase
  assign keyout=keyout_r;
endmodule
```

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.

3. Button debounce flow chart

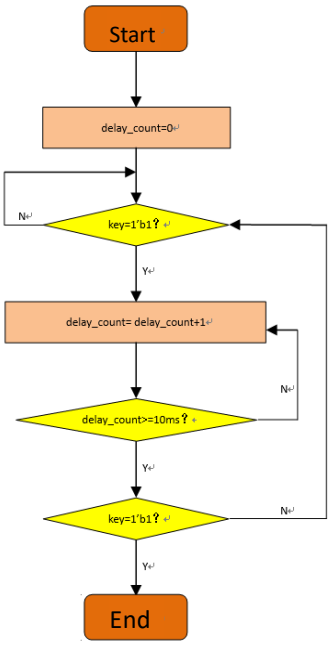

Fig 5. 2 Button debounce flow chart

4. Combine running LED design and modify the button debounce.
    a. Build new project
    b. Create a PLL symbol
    c. Create a button debounce symbol (See the Verilog HDL code in this experiment)
    d. Create a running LED symbol

```
//Verilog code
module Led_shifting(
    input               rst,
    input               sys_clk,
    input               key_left,
    input               key_right,
    input               s_f,
    output reg [7:0]    led
    );

  reg     ext_rst;

   always@(posedge sys_clk) begin
    ext_rst<=rst;
    end
always@(posedge sys_clk)
if(ext_rst)begin
led<=8'hff;
end
else begin
    if(key_left) begin
```

```
            if(led==8'hff)
                led<=8'b0000_0001;
            else
                led<={led[6:0],led[7]};
        end
        else if(key_right) begin
            if(led==8'hff)
                led<=8'b1000_0000;
            else
                led<={led[0],led[7:1]};
        end
    end
endmodule
```

e. Create top level file and combine each symbol referring to Experiment 4. See Fig 5. 3

f. Pin assignment

Signal Name	Port Description	Network Label	FPGA Pin
left	Left shift signal	KEY0	3
right	Right shift signal	KEY1	7
rst	Reset signal	KEY2	10

Table 5. 1 Pin assignment

One more thing to mention is that in the **I/O Standard** column, **select 3.3-V LVCMOS** instead of **2.5 V**

g. Compile

h. Download the program to the board

i. Observe the testing result, to see whether every time pressing a button, LED will move towards the corresponding direction. PB3 is reset, PB1 is to move to the left, and PB2 is to move to the right. (block_debouncing Quartus II project files can be referred).

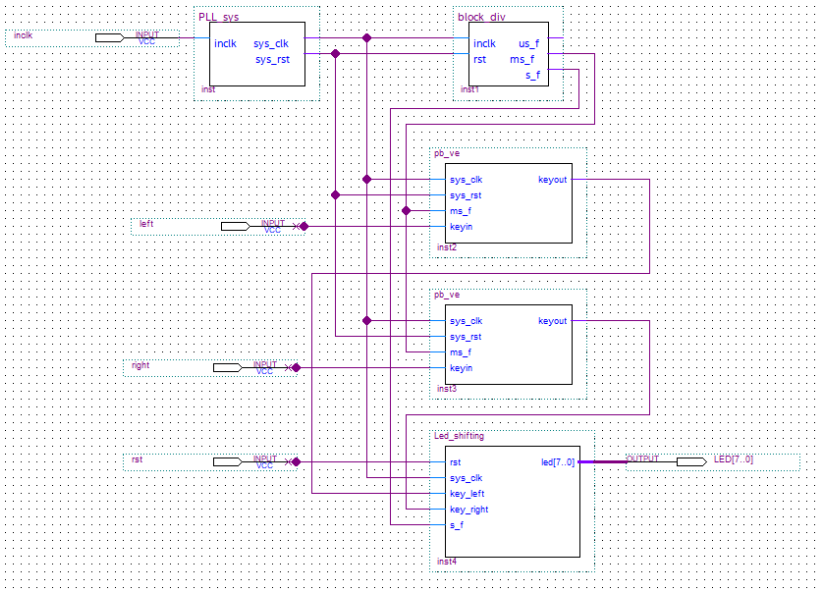

Fig 5. 3 Top level design

5. Button PCB schematics. See Fig 5. 4

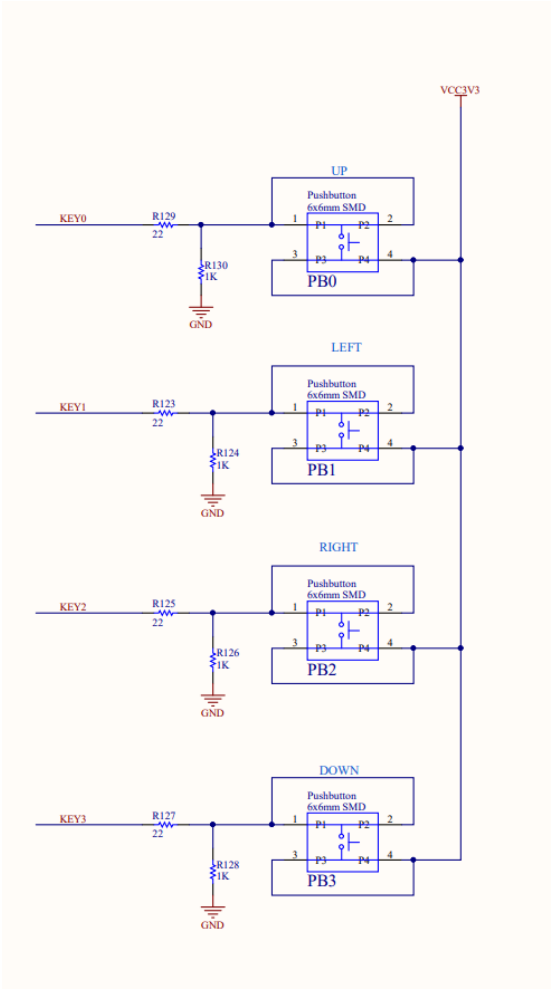

Fig 5. 4 PCB schematics

Experiment 6 Use Multiplier and ModelSim

6.1 Experiment Objective

1. Learn to use multiplier
2. Use ModelSim to output

6.2 Experiment Requirement

1. 8×8 multiplier. The first input is 8-bit switch, and the second input is the output of an 8-bit counter.
2. Observe the output in ModelSim.
3. Observe the calculation result on 4 segment decoders.

6.3 Experiment

1. Build a new project **mult_sim**
   Different from what we did before, we use EDA simulation. The actual setting is shown in Fig 6. 1

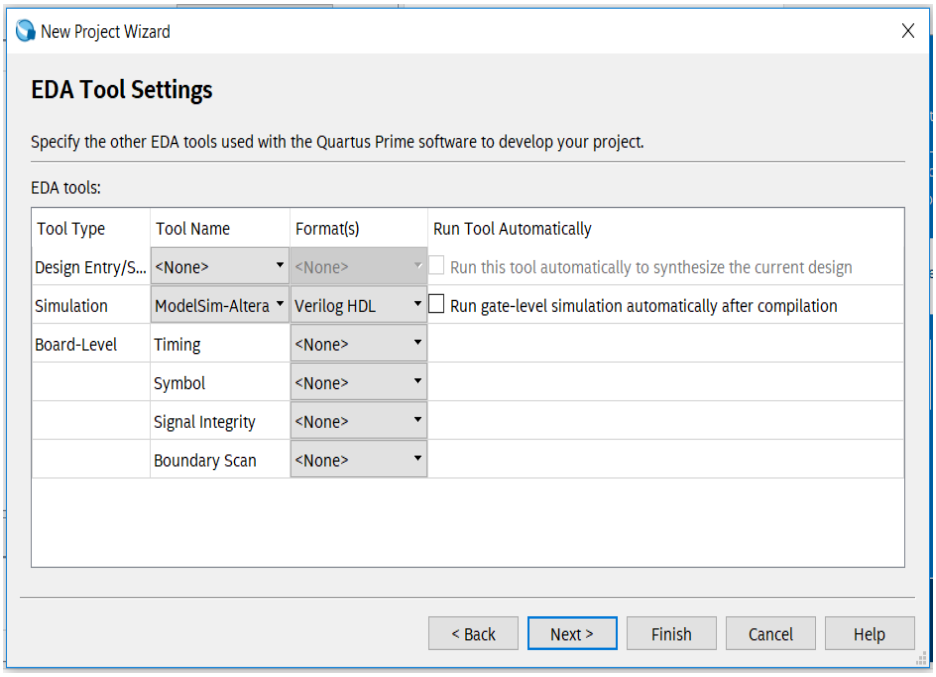

Fig 6. 1 Set EDA tool

2. Design procedure
   a. Create a new file named *mult_sim.v*
   b. Add PLL, set the clock input frequency is **50 MHz**, and the output is **100 MHz**
   c. In the right of the main interface, find **Installed IP > Library > Basic Functions >**

Arithmetic > LPM_MULT IP. An interface will pop up. See Fig 6. 2

d. Select the **Multiplication Type** to be **unsigned**. See Fig 6. 3

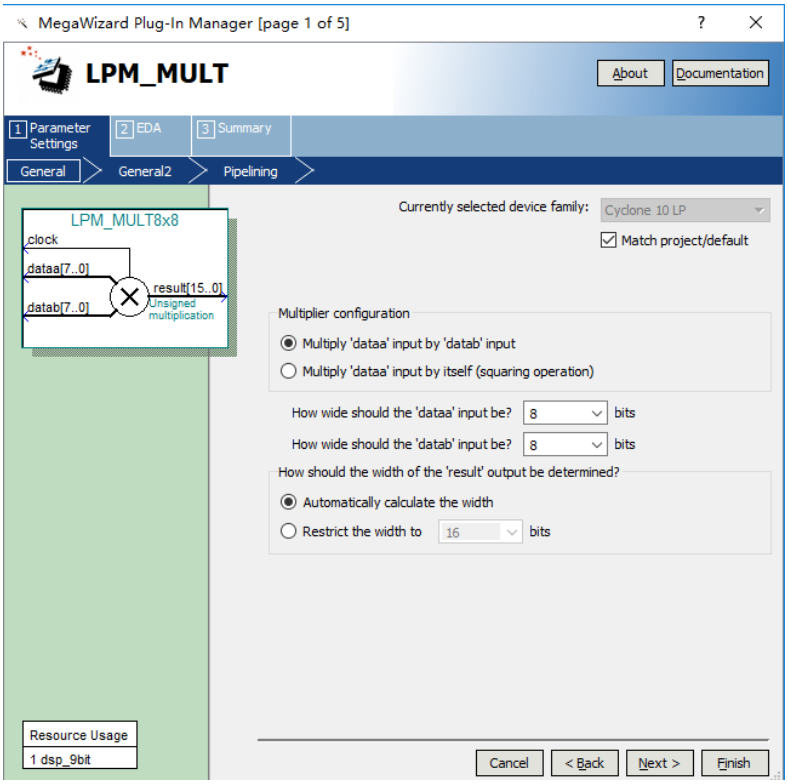

Fig 6. 2 LPM_MULT interface

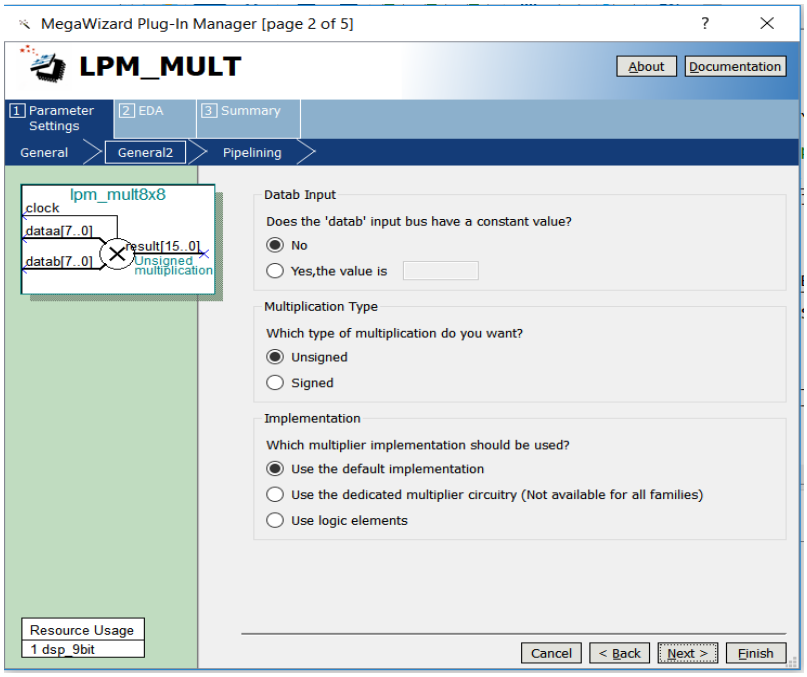

Fig 6. 3 Multiplication type selection

e. In **Pipelining**, select **Yes**, and set the output latency to be **1** clock cycle. Pipeline will speed up the execution speed. See Fig 6. 4

f. Make other settings default.

g. Instantiate it in the top level file.

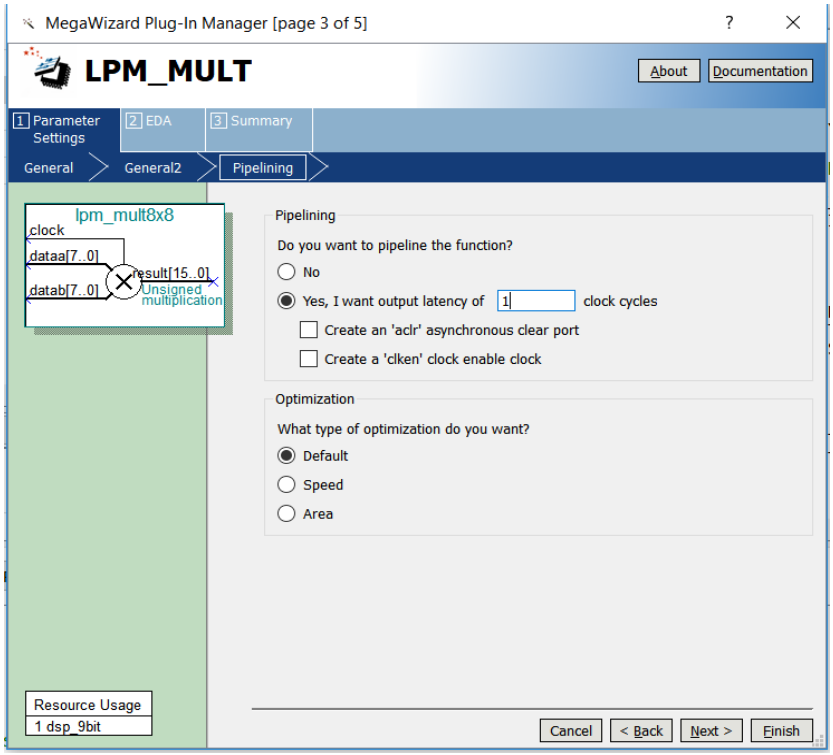

Fig 6. 4 Pipelining setting

3. The top level file is as follows

```
module mult_sim (
    input                   inclk,
    input                   rst,
    input       [7:0]sw,
    output      [6:0]seven_seg,
    output      [3:0]scan,
    output      [15:0]      mult_res,
    output  reg [7:0]count
);

    wire        sys_clk;
    wire        sys_rst;

    always @ (posedge sys_clk)
        if(sys_rst)
            count <= 0;
        else
            count <= count + 1;

    pll_sys_rst pll_sys_rst_inst(
            .inclk      (inclk),
            .sys_clk    (sys_clk),
            .sys_rst    (sys_rst)
        );
```

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

4. ModelSim simulation
   a. Simulation based on waveform inputs
   b. **Tool > Option**. In the popup window, under **General**, find **EDA Tool Options**. In **ModelSim-Altera**, find the correct path. See Fig 6. 5

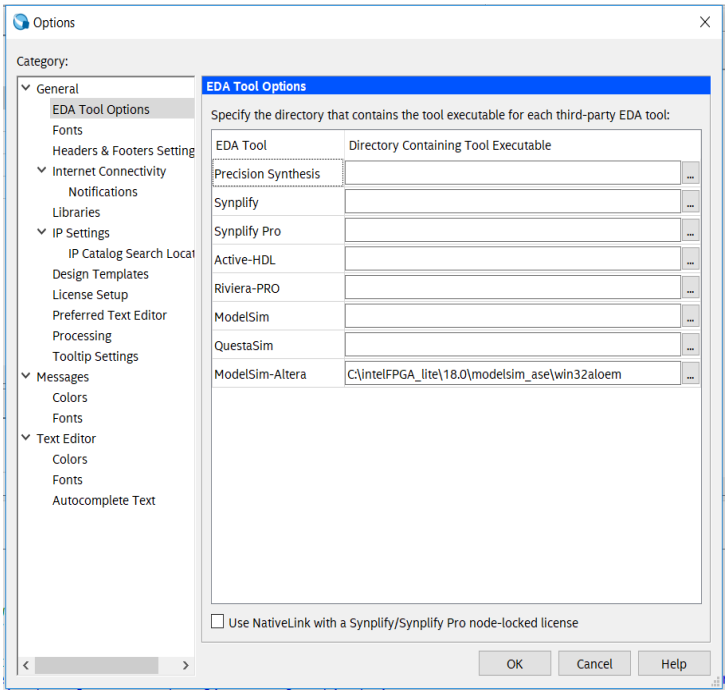

Fig 6. 5 Set the correct path for ModelSim-Altera

   c. **Tool > Run Simulation Tool > RTL Simulation**. See Fig 6. 6

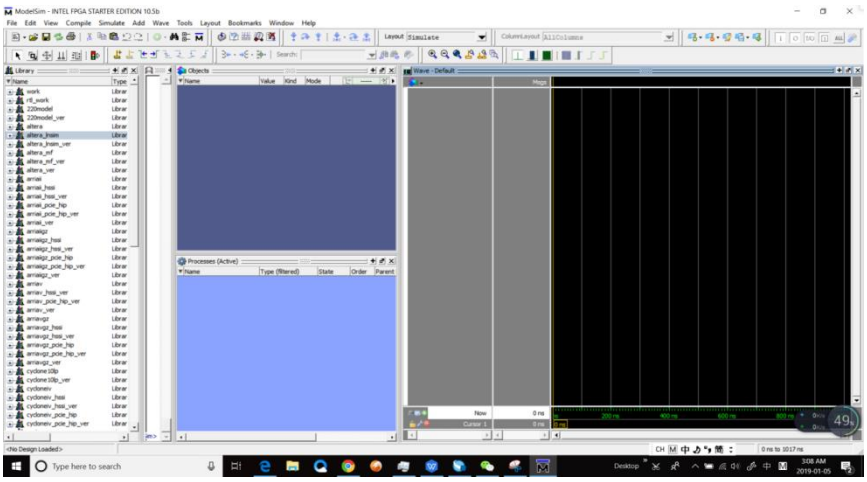

Fig 6. 6 Simulation interface

d. Set ModelSim

i. **Simulate > Start Simulation**

ii. In the popup window, add libraries under **Libraries** tag. See Fig 6. 7

iii. Under **Design** tag, choose simulation project *mult_sim* and click **OK**. See Fig 6. 8

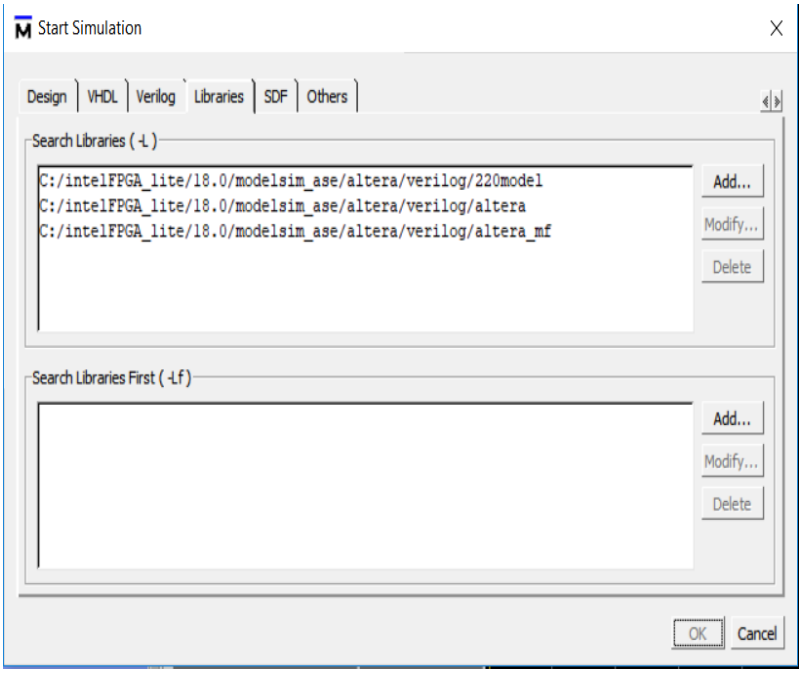

Fig 6. 7 Add simulation libraries

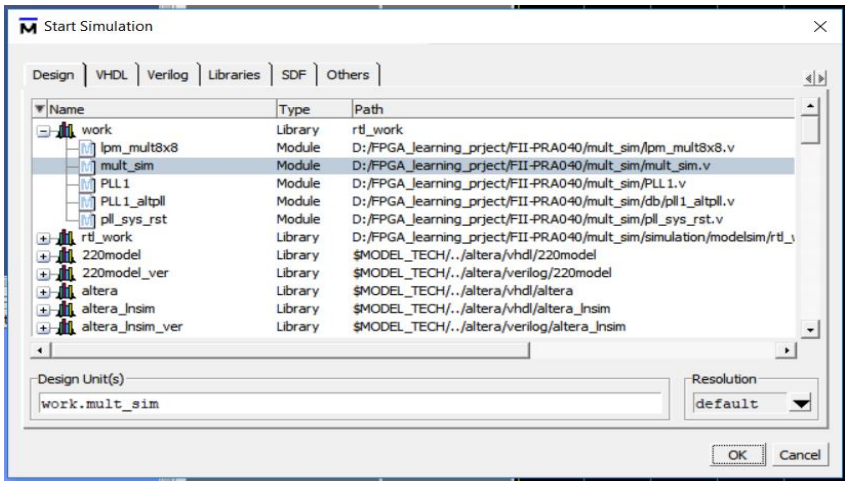

Fig 6. 8 Choose the project in simulation

iv. In the **Objects** window, choose all the signals and drag them to **Wave** window. See Fig 6. 9

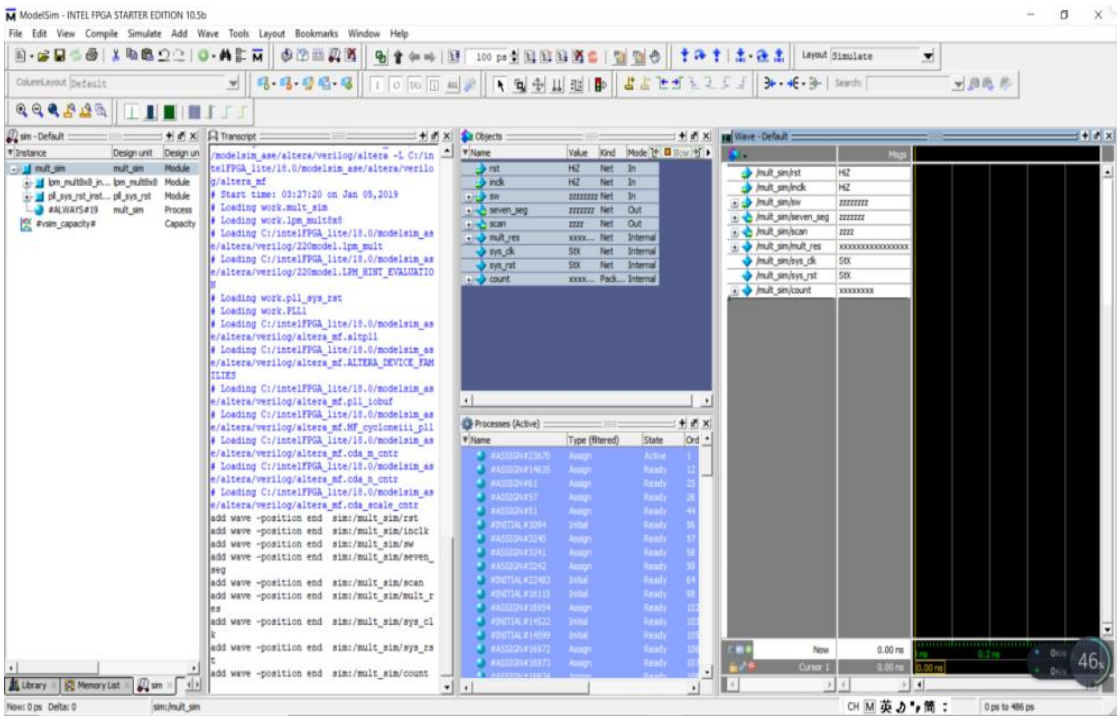

Fig 6. 9 Add observation signals

v. Set the signals in **Wave**, right click any signal and a selection window will occur. See Fig 6. 10

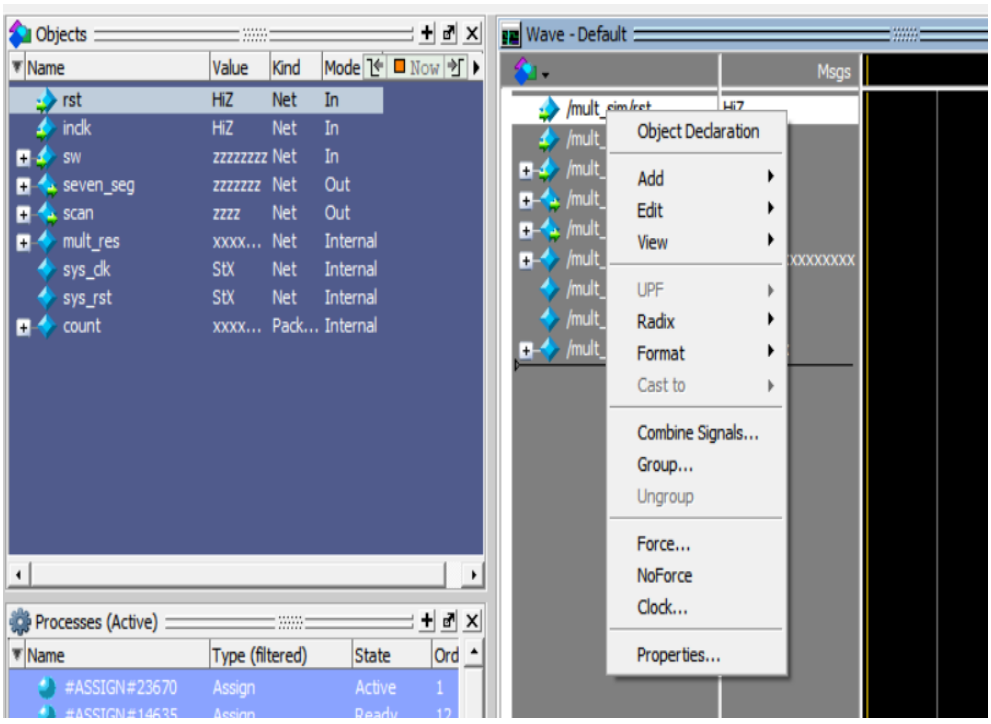

Fig 6. 10 Set the signals

vi. For logical signals select **Force** and select **Clock** for clock signals

1) Set *rst* signal. See Fig 6. 11

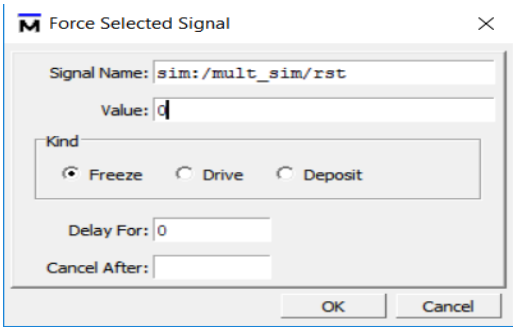

Fig 6. 11 Set *rst* signal

2) Set *Inclk* signal. See Fig 6. 12

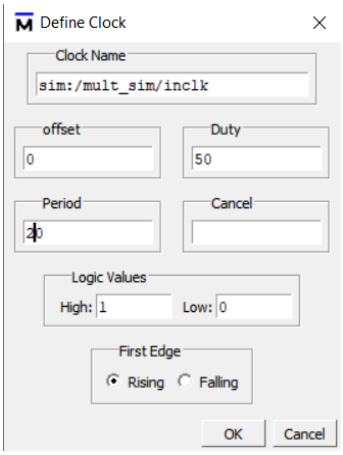

Fig 6. 12 Set *inclk* signal

3) Set *sw* signal. See Fig 6. 13

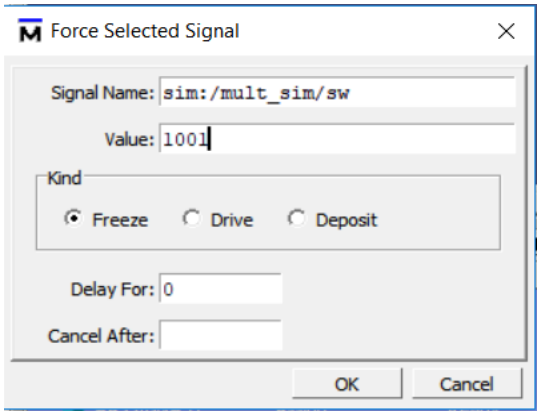

Fig 6. 13 Set *sw* signal

vii. Run simulation. In the tool bar, set the simulation time to be **100 ns**. Click the **Run** icon to run. See Fig 6. 14

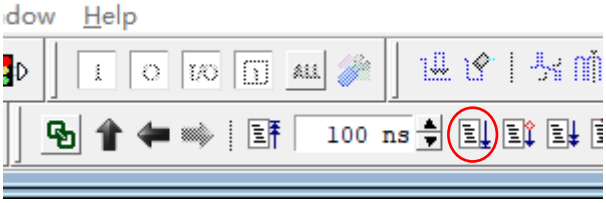

Fig 6. 14 Set the simulation time

viii. Observe the simulation result. See Fig 6. 15

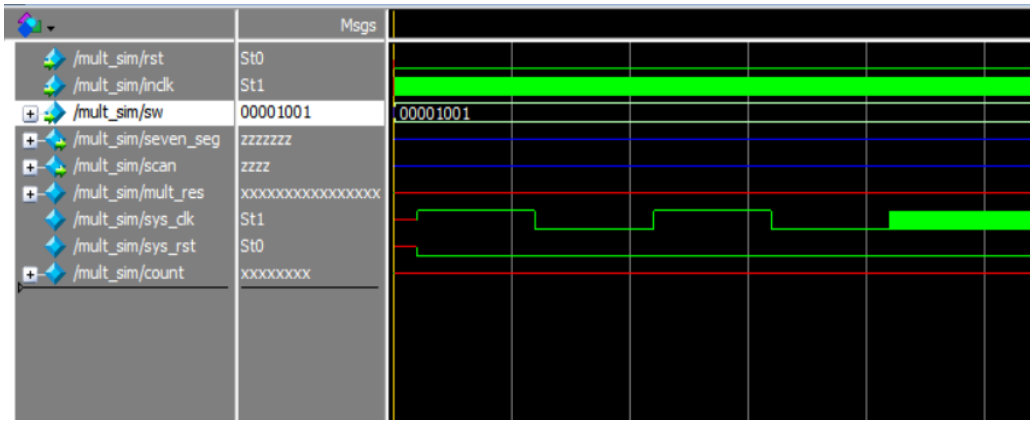

Fig 6. 15 Simulation result

ix. Result analysis

1) Counter **count** does not have a valid result, instead, unknow result XXXXXX is gotten.
2) **sys_rst** does not reset signals. It changes from X to 0
3) Add **pll_locked** signal to the wave, and re-simulate

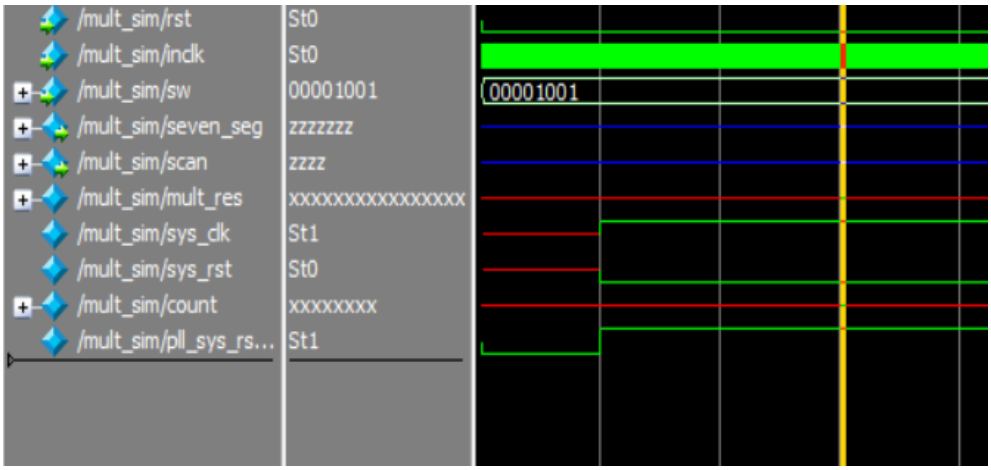

Fig 6. 16 Re-simulation result

4) In Fig 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.
5) Solution

Method 1: Define **sys_rst** to be 1'b0;

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

Method 2: Use external rst signal to procide reset

Here, method 1 is adopted

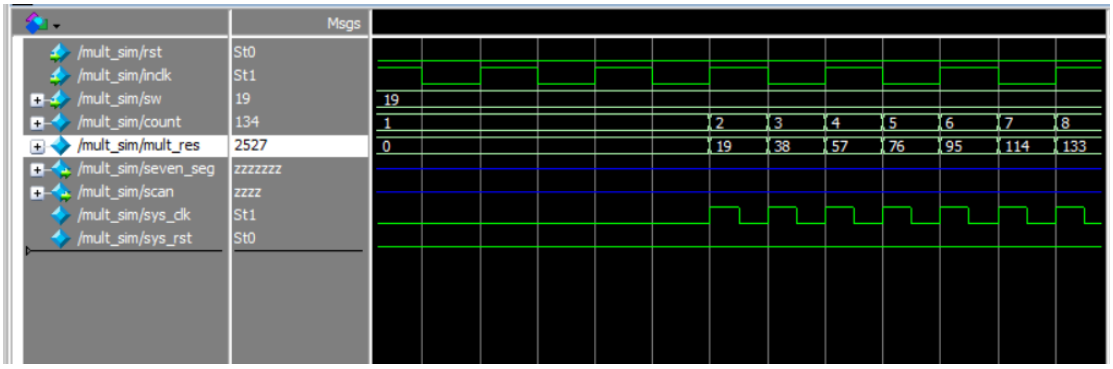

Fig 6. 17 Recompile the simulation

x. Recompile the simulation. See Fig 6. 17

xi. Since waveform editing efficiency is relatively low, the use of simulation testbench file is encouraged. Name a new Verilog HDL file *tb_mult.v*.

```
`timescale 10ns/1ns

module tb_mult;          //Define the simulation signal

    reg                  rst;
    reg                  clk;
    reg       [7:0]sw;
    wire [7:0]count;
    wire [7:0]seven_seg;
    wire [3:0]scan;
    wire [15:0]     mult_res;

    mult_sim S1(                        // S1 is the instance of simulation module
        .rst        (rst),
        .inclk      (clk),
        .sw         (sw),
        .seven_seg  (seven_seg),
        .scan       (scan),
        .count      (count),
        .mult_res (mult_res)
    );

    always    #5 clk = ~clk;

    initial                  //Initialize the simulation signals
    begin
        rst = 0;
        clk = 1;
        #5 sw = 20;
        #10 sw = 50;
        #10 sw = 100;
```

```
        #10 sw = 101;
        #10 sw = 102;
        #10 sw = 103;
        #10 sw = 104;
        #50 sw = 105;
        $monitor("%d * %d=%d", count, sw, mult_res);
        #1000 $stop;
    end
endmodule
```

xii. Compile and simulate

1) Only choose **Start Analysis & Elaboration**, do not choose either compilation or synthesis one. See Fig 6. 18

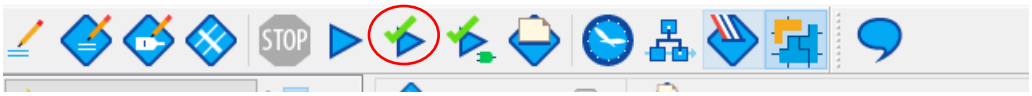

Fig 6. 18 TB file analyzing

2) Set the testbench file: **Assignments > Settings**. See Fig 6. 19

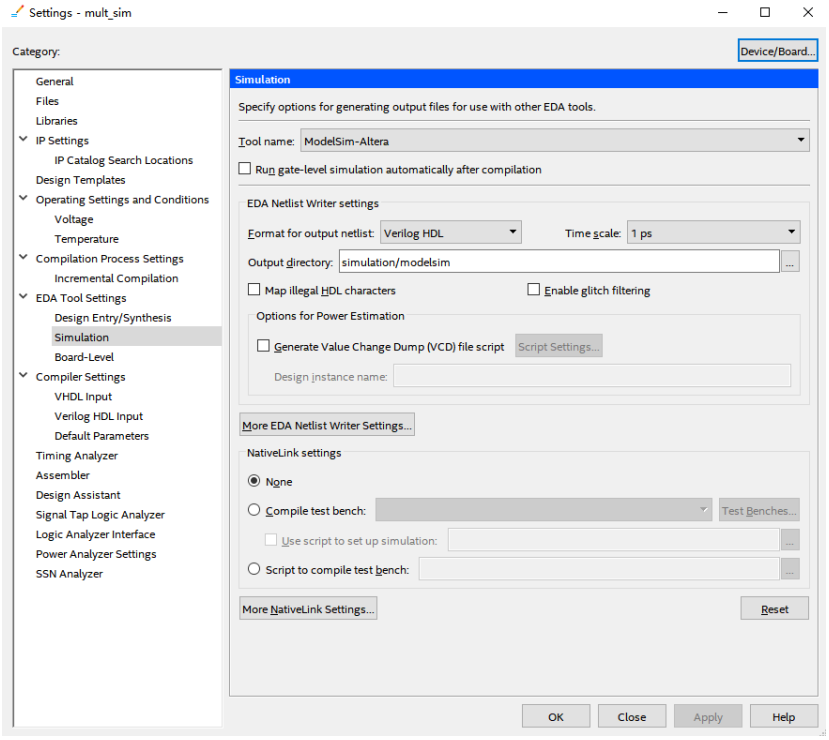

Fig 6. 19 Simulation setting 1

Go to **Simulation**, For **Tool name**, select **ModelSim-Altera**. In **Compile test bench**, click **Test Benches** to add tb simulation file. See Fig 6. 20

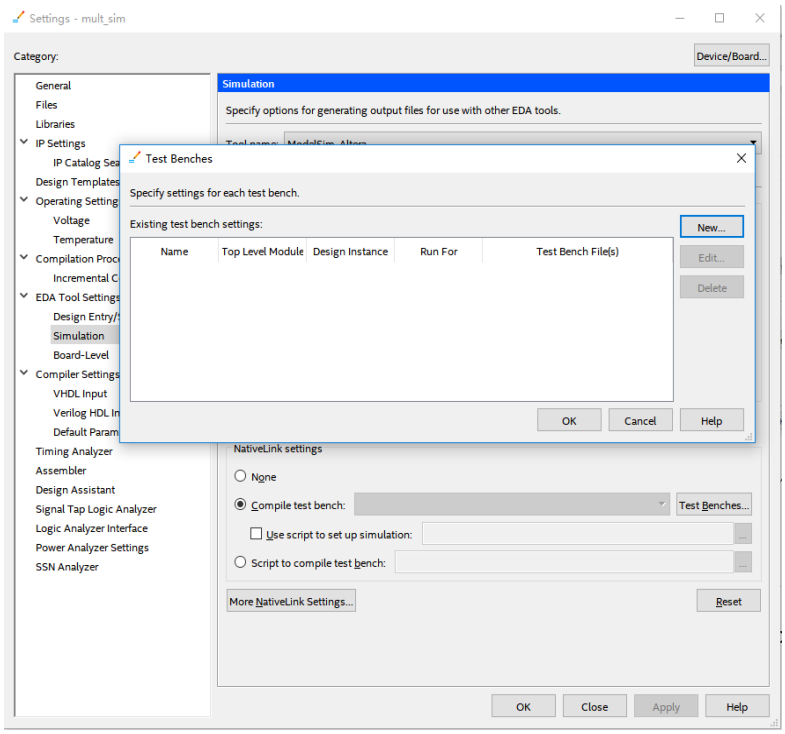

Fig 6. 20 Simulation setting 2

Click **New**, input the **Test bench name**. Make the name be consistent with tb file. See Fig 6. 21.

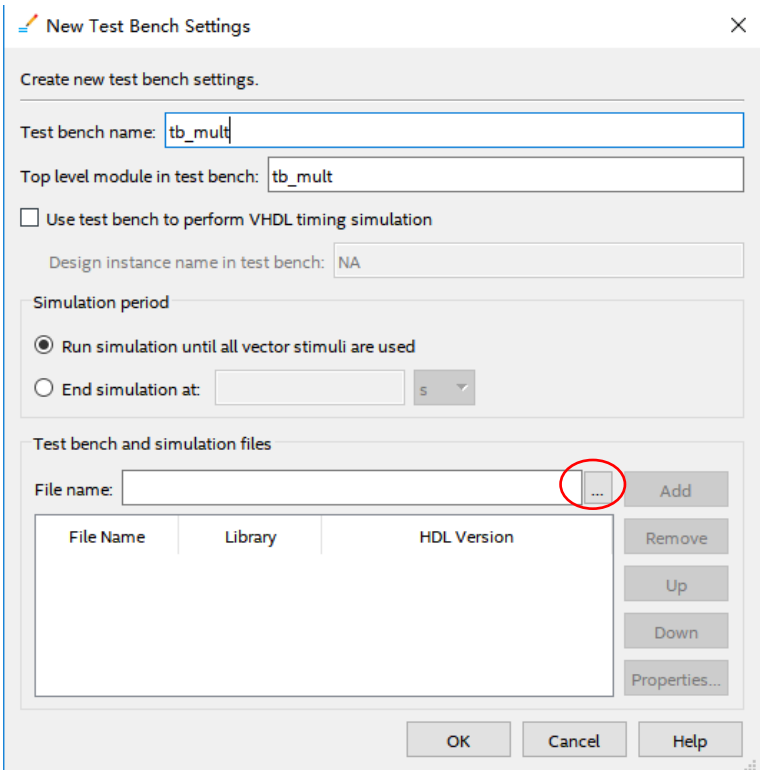

Fig 6. 21 Simulation setting 3

Click the red ellipse to add the test bench file. Find *tb_mult.v* file written before. Click **Add** to add. Click **OK** (three times) to finish the setting. See Fig 6. 22

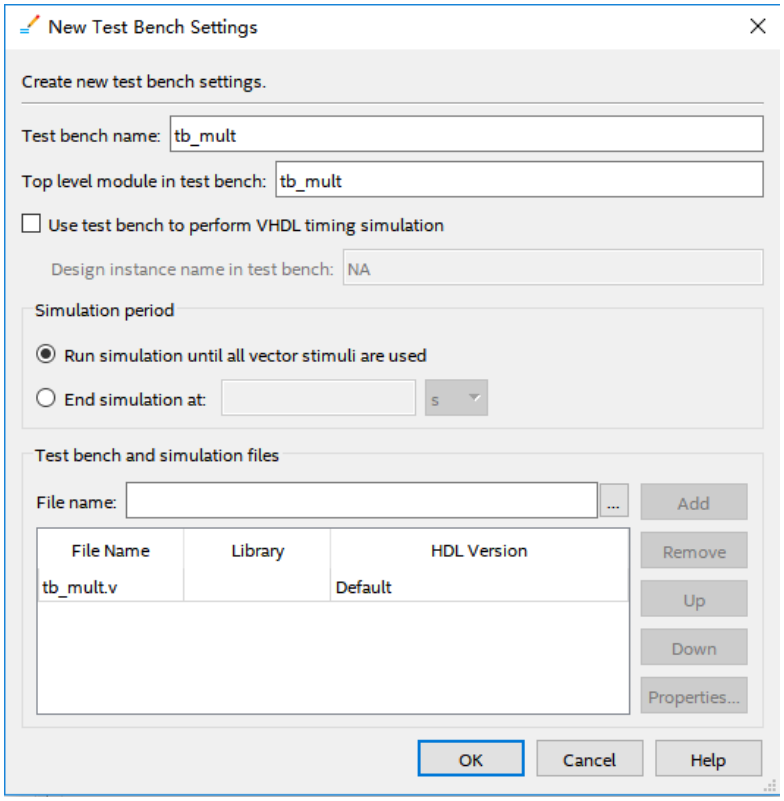

Fig 6. 22 Simulation setting 4

3) Repeat previous step, to start **ModelSim** to simulate. See Fig 6. 23

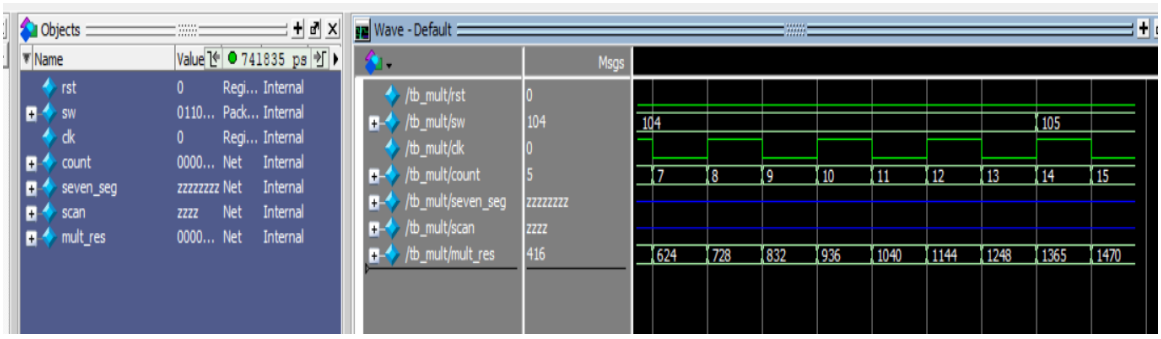

Fig 6. 23 Waveform output

4) After a certain delay, outputs will display in **Transcript**. See Fig 6. 24

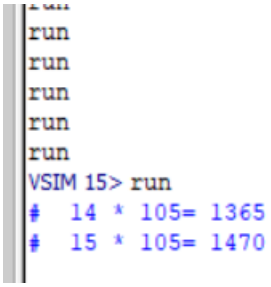

Fig 6. 24 Text displays operation result

Experiment 7 Hexadecimal Numbers to BCD Code Conversion and Application

7.1 Experiment Objective

1. Convert binary numbers to BCD
2. Convert hexadecimal numbers to BCD

7.2 Experiment Principle

1. Since the hexadecimal display is not intuitive, decimal display is more widely used in real life.
2. 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
   a. 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.

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

      iii. 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)

1) After shifting it becomes 1100 (12), greater than 1001 (decimal 9)
2) By plus 0110 (decimal 6), ABCD = 0010, carry position is 1, the result is expressed as decimal 12
3) Use pre-shift processing, ABCD = 0110 (6), greater than 5, plus 3
4) ABCD = 1001 (9), shift left by one
5) ABCD = 0010, the shifted shift is the lowest bit of the high four-bit BCD.
6) Since the shifted bit is 1, ABCD = 0010(2), the result is also 12 in decimal.
7) The two results are the same
8) 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

100's	10's	1's	Binary	Operation
			1010 0010	162
		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
1	6	2		

Fig 7. 1 Binary to decimal

Example 2: Hexadecimal to BCD

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		

Fig 7. 2 Hexadecimal to decimal

4. Write a Verilog HDL to convert 16-bit binary to BCD. (You can find reference in the project folder, *HEX_BCD.v*.
5. ModelSim simulation
   a. Refer to Experiment 6 to set the simulation
   b. Th simulation result is shown in Fig 7. 3

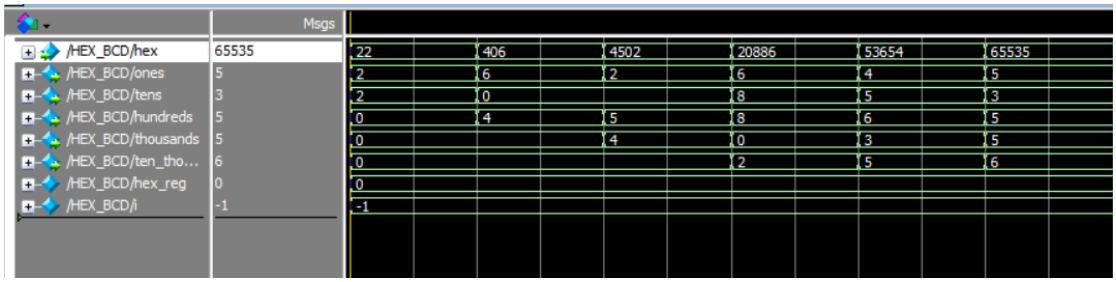

Fig 7. 3 Simulation for binary to decimal

6. Remark

   The assignment marks for the examples above are “=” instead of “<=”. Why?
   Since the whole program is designed to be combinational logic, when invoking the modules, the other modules should be synchronized the timing.

7.3 Application of Hexadecimal Number to BCD Number Conversion

1. Continue to complete the multiplier of Experiment 6 and display the result in digital form in decimal. Refer to the attached project file *HEX_BCD_mult.v*.
2. Compilation. Observe the **Timing Analyzer** in **Compilation Report**.
   a. **Slow 1200mV 85C Model > Fmax Summary** is 83. 71 MHz. See Fig 7. 4

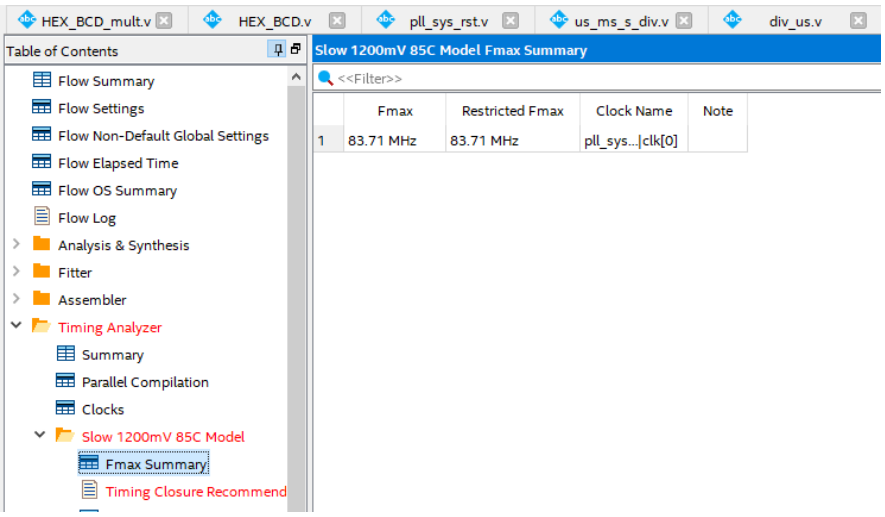

Fig 7. 4 Fmax Summary

b. **Setup Summary**

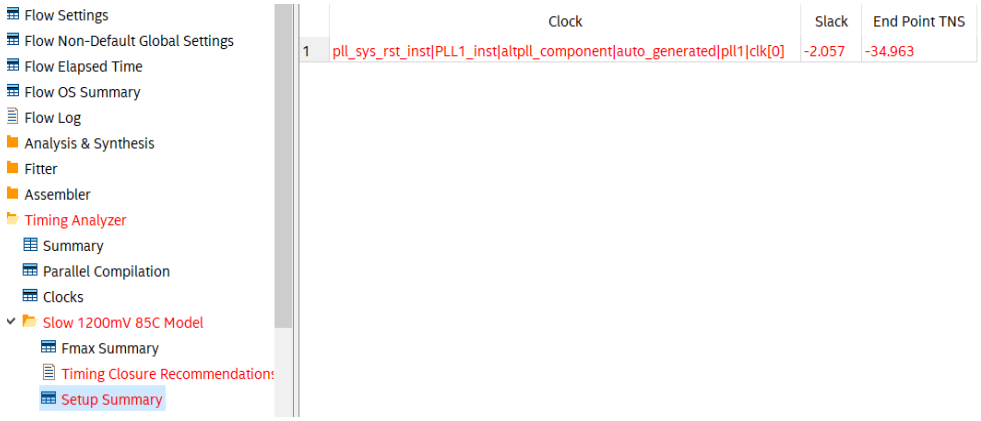

Fig 7. 5 Setup summary

c. **Timing Closure Recommendation**. See Fig 7. 6

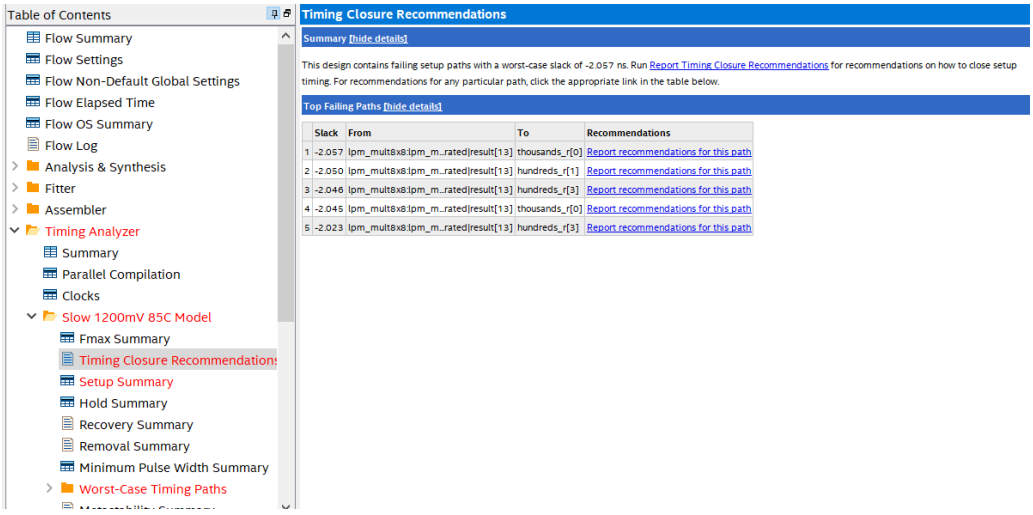

Fig 7. 6 Timing Analysis

d. 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:

i. Reduce the clock frequency
ii. Increase the timing of HEX_BCD and increase the pipeline
iii. 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.

```
module pll_sys_rst(

    input               inclk,
    output              sys_clk,
    output              BCD_clk,
    output    reg       sys_rst =1'b1
);

    wire pll_locked;

    always@(posedge sys_clk)
        sys_rst <= !pll_locked;

    PLL  PLL_inst (
        .areset (1'b0),
        .inclk0 (inclk),
        .c0   (sys_clk),
        .c1         (BCD_clk),          //20Mhz
        .locked (pll_locked)
    );
endmodule
```

4. New code added. Refer to the project files.

```
    reg     [15:0] mult_res_r;

    always @ (posedge BCD_clk)
        mult_res_r<=mult_res;
```

5. Recompile and observe the timing result.
6. Lock the pins and download the program to FII-PRA010 board. Test it.

7.4 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 (Read-only Memory)

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 Requirement

1. Design 16 outputs ROM, address ranging 0-255
2. Interface 8-bit switch input as ROM's address
3. Segment decoders display the contents of ROM and require conversion of hexadecimal to BCD output.

8.3 Experiment

8.3.1 Design Procedure

1. Build a new project named *memory_rom*
2. 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 Fig 8. 1

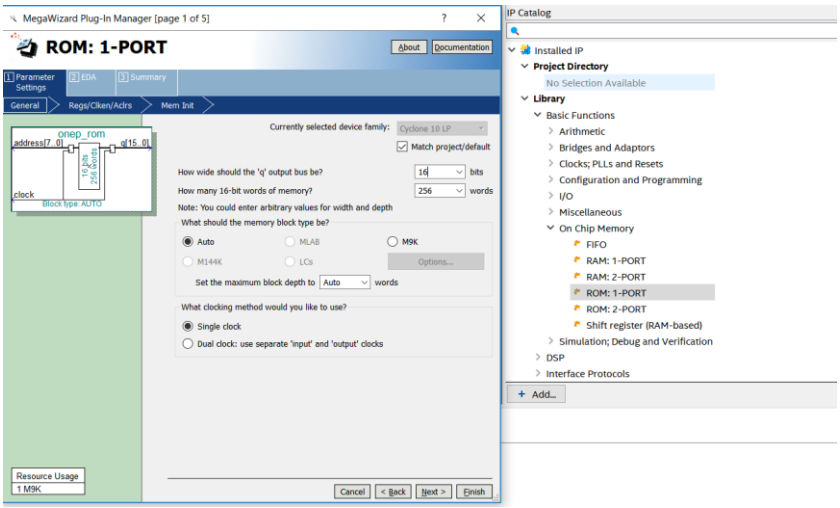

Fig 8. 1 RAM IP core invoking

3. According to the default setting, you need to add an initial ROM file in the location where red oval circles. See Fig 8. 2. In the figure, a *.mif file has already been added. (Refer to the project files)
4. Create a top level entity *rom.mif*
   a. Go to **File > New > Memory Files > Memory Initialization File**. See Fig 8. 3

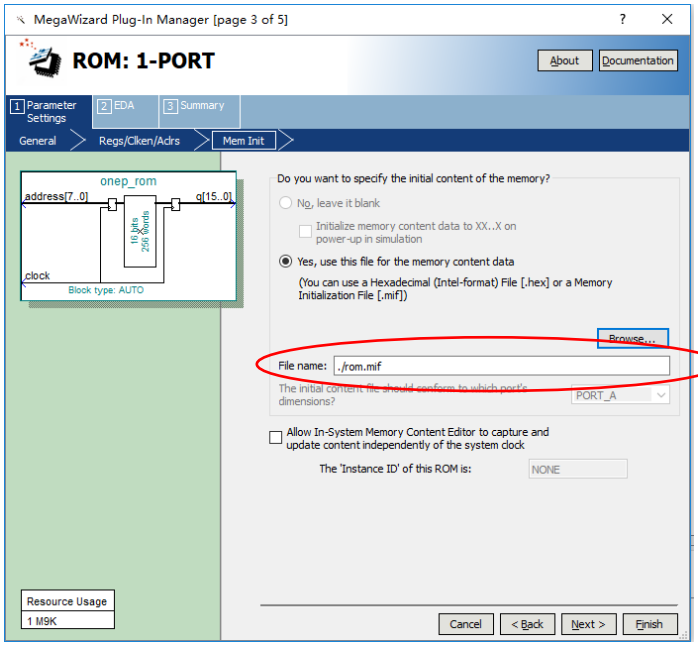

Fig 8. 2 ROM setting

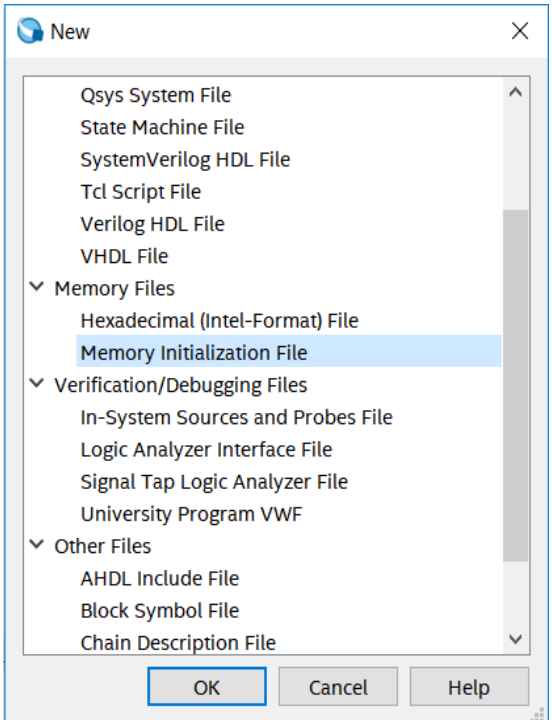

Fig 8. 3 New *.mif file

b. In Fig 8. 4, modify the **Number of words** and **Word size**.

c. In Fig 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.

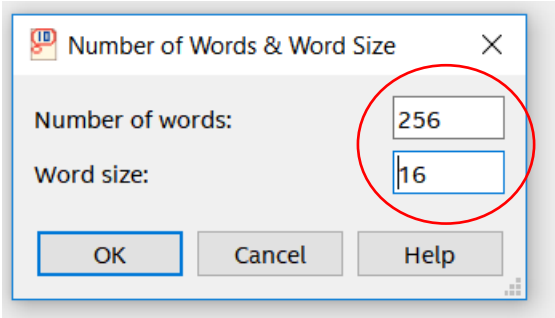

Fig 8. 4 *.mif file setting 1

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

Fig 8. 5 *.mif file setting 2

d. 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 Fig 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.

e. After the setup, the system will fill in the data automatically. See Fig 8. 7

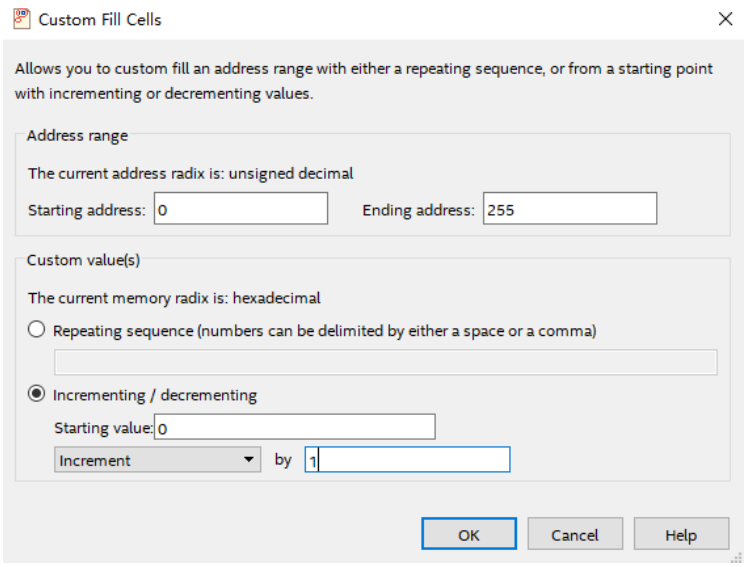

Fig 8. 6 Fill date for *rom.mif*

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

Fig 8. 7 Part of data after auto filling

5. Refer to the design of conversion from hexadecimal to BCD in Experiment 7, display the data in ROM to the segment decoders. (You can refer to the project files attached)

8.3.2 Board Verification

Compile, lock the pins, and verify the experiment downloading the program to the develop board.

Reflection:

1. How to use the initial file of ROM to realize the decoding, such as decoding and scanning the segment decoders.
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
4. Observing the synchronization structure of synchronous clock frames using SignalTap II
5. Extended the use of dual-port RAM
6. Design the use of three-stage state machine

9.2 Experiment Requirement

1. Generate dual-port RAM and PLL
   a. 16-bit width, 256-depth dual-port RAM
   b. 2 PLL, both 50 MHz input, different 100 MHz and 20 MHz outputs
2. Design a 16-bit data frame
   a. Data is generated by an 8-bit counter: Data={~counta,counta}
   b. The ID of the data frame inputted by the switch (7 bits express maximum of 128 different data frames)
   c. 16-bit *checksum* provides data verification
      i. 16-bit *checksum* accumulates, discarding the carry bit
      ii. 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

Wr_addr	Date/ 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

Table 9. 1 Memory structure

6. Read and write in an agreed order

   Firstly, write in the order

      i. Read the flag of the 8'hff address (control word). If *valid*=1'b0, the

program proceeds to the next step, otherwise waits

ii. Address plus 1, 8'hff+1 is exactly zero, write data from 0 address and calculate the *checksum*

iii. 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.

iv. *checksum* complements and write to memory

v. Write the control word in the address 8'hff, packet it

Secondly, read in the order

i. *Idle* is the state after reset

ii. *Init*: Initialization, set the address to 8'hff

iii. *Rd_pipe0*: Add a latency (since the read address and data are both latched). Address +1, forming a pipeline structure

iv. *Read0*: Set the address to 8'hff, read the control word and judge whether the valid bit is valid.

If *valid*=1'b1, address +1, proceeds to the next step

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.

v. *Read1*: Read the control word again

If *valid*=1'b1, address+1, ID and data length are assigned to the corresponding variables and proceeds to the next step

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.

vi. Rd_data:

Read data and pass to data variables

Calculate *checksum*, *data_len* - 1

Determine whether the *data_len* is 0, if so, all data has been read, proceeds to the next step, otherwise, continue the operation in current state

vii. *grd_chsum*: Read the value of *checksum* and calculate the last *checksum*. Correct the data and set the flag of *rd_err*

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

Thirdly, *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. See the project files for more details.

9.3 Experiment

1. Port

```
module frame_ram
#(parameter data_len=250)
(
```

```
input                       inclk,
input                       rst,          //external reset
input           [6:0]sw,          //used as input ID
output    reg[6:0] oID,         //used as output ID
output    reg           rd_done,    //frame read is done
output    reg       rd_err       //frame read has errors
);
```

2. Definition of 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;
```

3. Define clock parameter

```
wire              sys_clk;
wire              BCD_clk;
wire              sys_rst;
reg               ext_clk;
```

4. Define two-port RAM interface

```
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;
```

5. Write state machine partial variable definition
   a. Write state machine variables

```
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=~counta;

reg           ext_rst;
reg [2:0]     sta;
reg[2:0]      sta_nxt;
```

b. Read state machine variables

```
reg[15:0]     rd_chsum；
reg[7:0]      rd_len;
reg[15:0]     rd_data;

reg           ext_rst;
reg[2:0]      stb;
reg[2:0]      stb_nxt;
```

6. Data generation counter

```
always@(posedge BCD_clk)
ext_rst<=rst;

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

7. Write state machine

a. First and second stages

```
assign wr_done=(wr_len==data_len-1); //Think why to use wr_len==data_len-1
                                     //instead of wr_len==data_len
always@(posedge sys_clk)
if(sys_rst) begin
   sta=mema_idle;
end
else
    sta=sta_nxt;

always@(*)
```

```
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
```

b. Third stage

```
always@(posedge sys_clk)
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;
wren_a<=1'b1;
data_a<={countb,counta};
wr_len<=wr_len+1;

wr_chsum<=wr_chsum+{countb,counta};
end

mema_wr_chsum:begin
addr_a<=addr_a+1;
wr_len<=wr_len+1;
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
```

8. Read state machine
   a. First stage

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

   b. Second stage

```
always @ (*)
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
```

c. Third stage. The actual operation is driven by the edge of the clock

```
always@(posedge sys_clk)
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;
```

```
endmemb_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)//Determine if rd_chsum is not 0, error occurs when reading data
    rd_err<=1'b1;
```

```
        end

        memb_rd_done: begin
        addr_b<=8'hff;
        wren_b<=1'b1;
        end
        default:;
        endcase

        always@(*)begin
        if(stb==memb_rd_data)
        rd_done=(rd_len==0);
        else
        rd_done=1'b0;
        end
```

9. Instantiate dual-port RAM and PLL

```
//Instantiate dual-port RAM
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)
);

//Instantiate PLL
pll_sys_rst pll_sys_rst_inst
(
.inclk     (inclk),
.sys_clk (sys_clk),
.BCD_clk(BCD_clk),
.sys_rst (sys_rst)
);
endmodule
```

9.4 Lock the Pins, Compile, and Download to FII-PRA006 FPGA to Test

Signal Name	Port Description	Network Label	FPGA Pin

Inclk	Input clock	C10_50MCLK	91
rst	Reset signal	KEY2	10
sw[6]	Switch input 6	SW6_LED6	76
sw[5]	Switch input 5	SW5_LED5	75
sw[4]	Switch input 4	SW4_LED4	74
sw[3]	Switch input 3	SW3_LED3	87
sw[2]	Switch input 2	SW2_LED2	86
sw[1]	Switch input 1	SW1_LED1	83
sw[0]	Switch input 0	SW0_LED0	80

9.5 Use SignalTap II to Observe the Dual-port RAM Read and Write

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,          //external reset
input    [6:0]sw,          //used as input ID
output   reg[6:0] oID,          //used as output ID
output   reg    rd_done,    //frame read is done
output   reg      rd_err //frame read has errors
);
```

2. SignalTap II simulation result. See Fig 9. 1

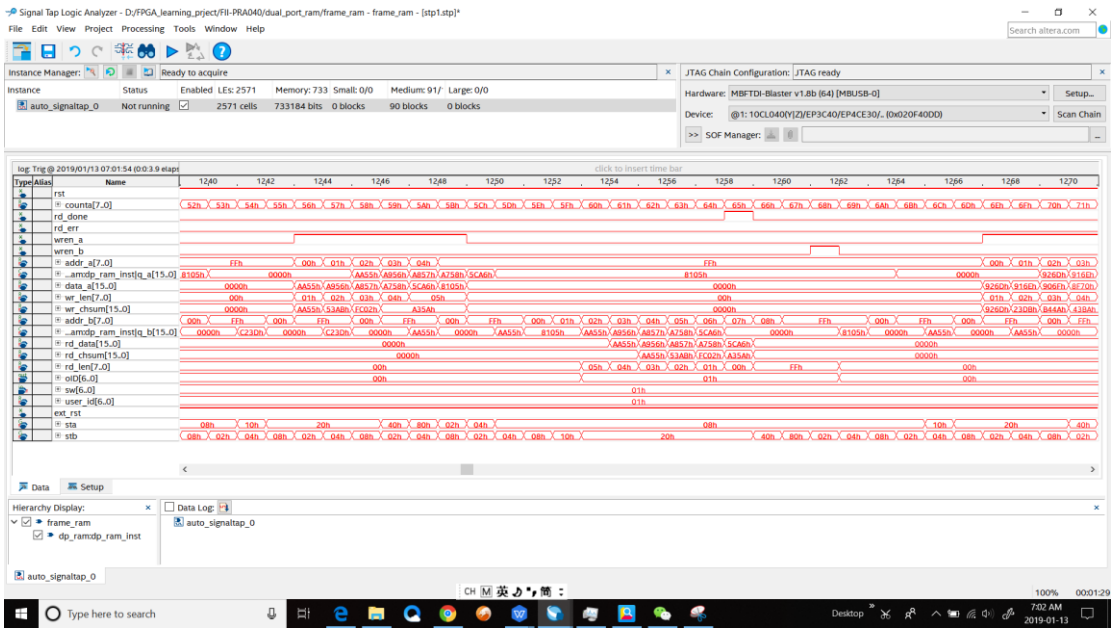

Fig 9. 1 SingalTap II simulation

3. Observe the simulation result
   a. Observe the handshake mechanism through dual-port RAM
      Determine whether the reading is started after the packet is written, whether the write packet is blocked before reading the entire packet is completed.
   b. Observe the external interface signal and status
      Rd_done, *rd_err*
      Set *rd_err* = 1, or the rising edge is the trigger signal to observe whether the error signal is captured.
      Observe whether *wren_a*, *wren_b* signal and the state machine jump are strictly matched to meet the design requirements.

9.6 Experiment Summary and Reflection

1. Review the design requirements. How to analyze an actual demand, 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, 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.

Experiment 10 Asynchronous Serial Port Design and Experiment

10.1 Experiment Objective

1. Because asynchronous serial ports are very common in industrial control, communication, and software debugging, they are also vital in FPGA development.
2. Learning the basic principles of asynchronous serial port communication, handshake mechanism, data frame
3. Master asynchronous sampling techniques
4. Review the frame structure of the data packet
5. Learning FIFO
6. Joint debugging with common debugging software of PC (SSCOM, teraterm, etc.)

10.2 Experiment Requirement

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

1. Build new project named *uart_frame*, select **10Cl010YF144C8G** for **device**.
2. Add new file named *uart_top*, add a PLL (can be copied from the previous experiment)

```
module uart_top
(
input       inclk,
input       rst,
Input       baud_sel,
input       rx,
output      intx
);

wire    sys_clk;
wire    uart_clk;
wire     sys_rst;
wire    uart_rst;

pll_sys_rst pll_sys_rst_inst
(
```

.inclk (inclk),
.sys_clk (sys_clk),
.uart_clk (uart_clk),
.sys_rst (sys_rst),
.uart_rst(uart_rst)
);

endmodule

3. New baud rate generator file
   a. 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 Fig 10. 1.

      Fine solution
      i. Implemented with a two-stage PLL for a finer frequency
      ii. 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.

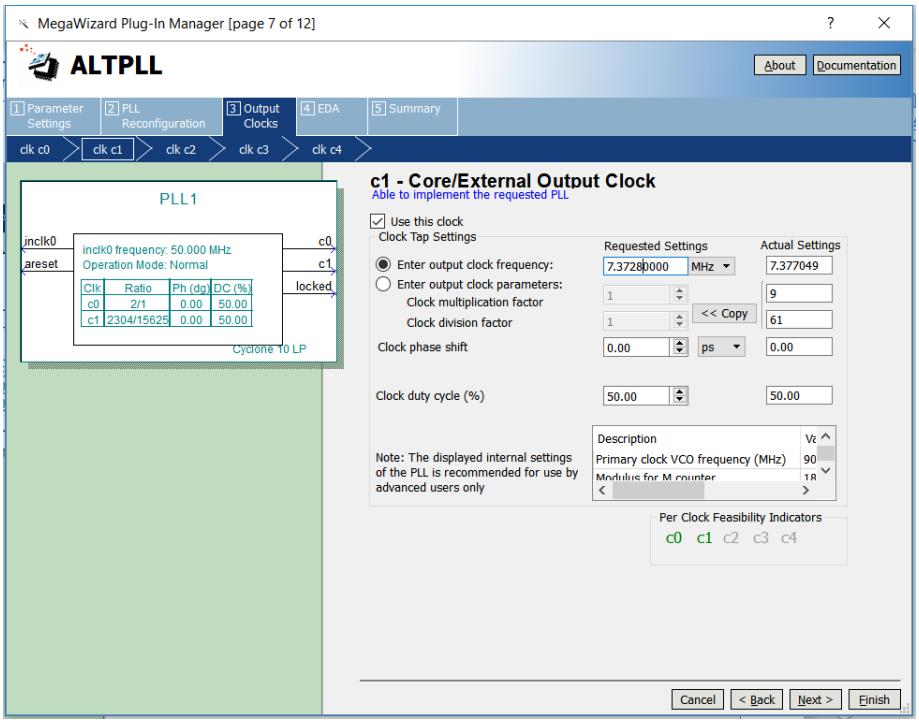

Fig 10. 1 PLL setting

   b. Supported baud rates are 115200，57600，38400，19200
   c. The default baud rate is 115200

4. Design of baud rate (Refer to the project files *UART_FRAME*)
   a. Instantiate and set it as top level

wire tx_band;
Wire tx_band;

```
baud_rate
#(.div(64))
  baud_rate_inst
(
.rst              (uart_rst),
.inclk            (uart_clk),
.baud_sel         (baud_sel),
.baud_tx          (baud_tx),
);
```

b. Baud rate design source file

```
module baud_rate
#(parameter div=64)
(
input                    rst,
input                    inclk,
input        [1:0] baud_sel,
output reg           baud_tx,
output reg          baud_rx
);

wire [8:0] frq_div_tx;          //Send baud rate, clock frequency division selection
assign frq_div_tx=(baud_sel==2'b0)?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
```

```
wire [6:0] frq_div_rx;                                  //Accept partial baud rate design
assign frq_div_rx=(baud_sel==2'b0)?7'd7:
                  (baud_sel==2'b01)?7'd15:
                  (baud_sel==2'b10)?7'd31:7'd63;

reg   [8:0] count_rx=9'd0;

always@(posedge inclk)
if(rst) begin
   count_rx    <=9'd0;
   baud_rx <=1'b0;
end
else begin
if(count_rx==frq_div_rx) begin
   count_rx <=9'd0;
   baud_rx<=1'b1;
      end
else begin
count_rx<=count_rx+1'b1;
baud_rx<=1'b0;
end
end

endmodule
```

5. Design the buffer file *tx_buf*
   a. 8-bit FIFO, depth is 256, read/write clock separation, full flag, read empty flag
   b. Interface and handshake
      i. *rst* reset signal
      ii. *wr_clk* write clock
      iii. *tx_clk* send clock
      iv. 8-bit write data *tx_data*
      v. *wr_en* write enable
      vi. *ctrl* writes whether the data is a data or a control word
      vii. *rdy* buffer ready, can accept the next data frame
   c. Send 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)
);
```

d. Send buffer source file (Refer to the project file)

6. Serial transmission, interface and handshake file design
   a. Interface design
      i. *tx_rdy*, send vacancy, can accept new 8-bit data
      ii. *tx_en*, send data enable, pass to the sending module 8-bit data enable signal
      iii. *tx_data*, 8-bit data to be sent
      iv. *tx_clk*, send clock
      v. *tx_baud*, send baud rate

   b. 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)
);
```

   c. Source file (Refer to the project files)
7. Send file *testbench.v* (Refer to the project file *tb_uart*)
8. Send ModelSim simulation. See Fig 10. 2 (Already saved as *wave.do* file)

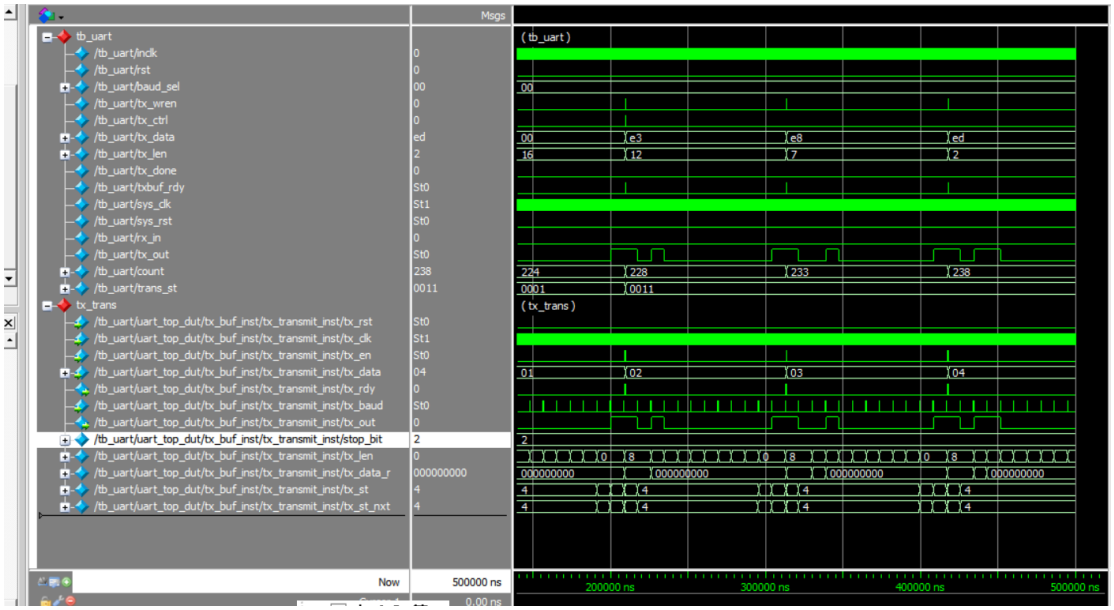

Fig 10. 2 ModelSim simulation waves sent by serial

9. Extended design (extended content is only reserved for users to think and practice,)
   a. Design the transmitter to support 5, 6, 7, 8-bit PHY (Port physical layer)
   b. Support parity check
10. The settings of the above steps involve FIFO, PLL, etc. (Refer to *uart_top* project file)

    UART accept file design

    a. Design of *rx_phy.v*

       Design strategies and steps

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

       ii. Adopting multiple judgments to realize the judgment of receiving data. Determine whether the data counter is greater than 4 after the sampling value is counted.

       iii. Steps to receive data:

            1) Synchronization: refers to how to find the start bit from the received 0101... *sync_dtc*
            2) Receive start bit (start)
            3) Cyclically receive 8-bit data
            4) Receive stop bit (determine whether it is one stop bit or two stop bits)

       Determine if the stop bit is correct

       Correct, jump to step ii

       Error, jump to step i, resynchronize

       Do not judge, jump directly ii, this design adopts the scheme of no judgment

    b. *rx_phy* source file (Refer to the project file)

    c. The design of *rx_buf*

       Design strategy and steps

       i. Add 256 depth, 8-bit fifo

1) Read and write clock separation
2) Asynchronous clear (internal synchronization)
3) Data appears before the *rdreq* in the read port

ii. Steps:

1) Initialization: *fifo*, *rx_phy*
2) Wait: FIFO full signal (*wrfull)* is 0
3) Write: Triggered by *rx_phy*: *rx_phy_byte*, *rx_phy_rdy*
4) End of writing
5) Back to ii and continue to wait

rx_buf.v source program (Reference to project files)

Receive simulation incentive

Content and steps

i. *tx*, *rx* loopback test (assign *rx_in = tx_out*)
ii. Continue to use the incentive file in the TX section
iii. Writing the incentive part of *rx*

Modelsim simulation, as shown in Fig 10. 3

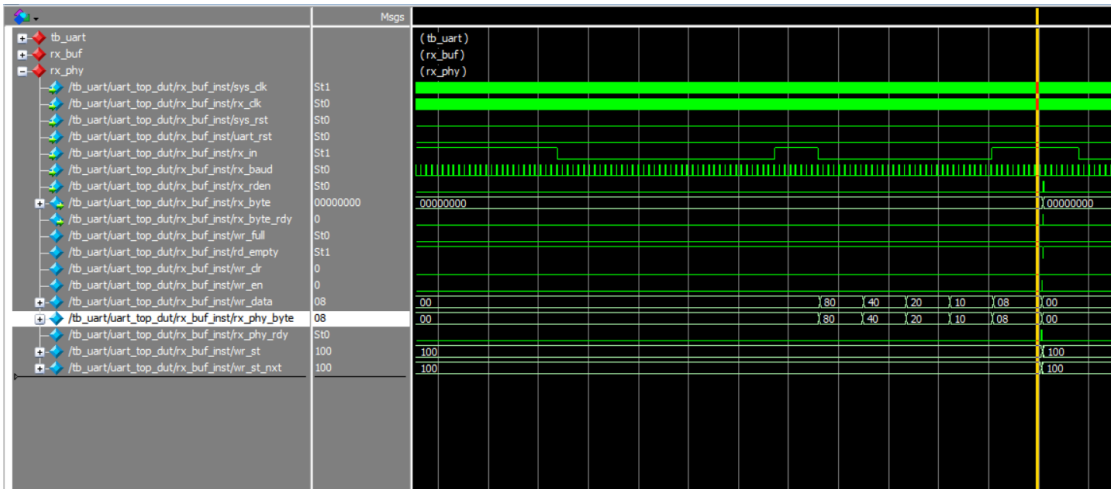

Fig 10. 3 *ry_phy* waveform

Reflection and expansion

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 receipt frame of *rx_buf*
iv. Using multi-sampling to design 180° alignment of data, compare with FPGA resources, timing and data recovery effects

Hardware test

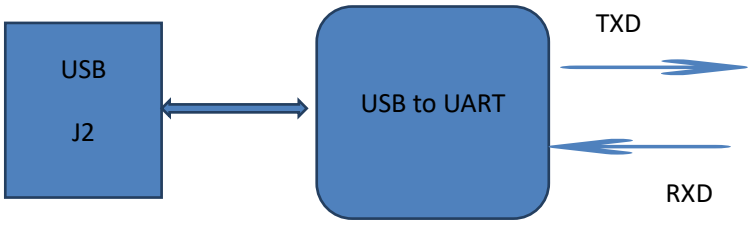

Fig 10. 4 USB to serial conversion

Firstly, Use FII-PR006 to test

Secondly, FPGA and UART pin mapping table. See Table 10. 1

UART	RXD	TXD
Schematic Name	JTAG_TXD_O	JTAG_RXD_I
FPGA Pin	144	143

Table 10. 1 FPGA and UART pin mapping table

Thirdly, lock the pins, and recompile

Fourthly, write a hardware test file

a. Development board J2 is connected to the host USB interface
b. Using test software such as teraterm, SSCOM3, etc. You can also write a serial communication program (C#, C++, JAVA, Python...).
c. PC sends data in a certain format
d. The test end uses a counter to generate data in a certain format.
e. The test procedure is as follows (*hw_tb_uart*)

```
module hw_tb_uart(
    input       inclk,
    input       rst,
    input       rx_in,
    output      tx_out
    );

wire [1:0]    baud_sel=2'b00;  //Default baud rate is 115200

reg             tx_wren=0;
reg             tx_ctrl=0;
reg     [7:0] tx_data=0;
reg     [7:0]   tx_len=0;
reg             tx_done;
wire            txbuf_rdy;

wire            sys_clk;
wire            sys_rst;

reg    [7:0] count=0;

reg    [3:0] trans_st;
always@(posedge sys_clk)
if(sys_rst)begin
    trans_st      <=0;
    tx_wren        <=1'b0;
    tx_ctrl       <=1'b0;
    tx_data        <=8'b0;
```

```
        tx_done         <=1'b0;
        tx_len          <=0;
        tx_len          <=0;
        count           <=8'd0;
end
else case(trans_st)
0:begin
        trans_st        <=1;
        tx_wren         <=1'b0;
        tx_ctrl         <=1'b0;
        tx_data         <=8'b0;
        tx_done         <=1'b0;
        tx_len          <=16;
        end
1:begin
        tx_wren         <=1'b0;
        tx_ctrl         <=1'b0;
        tx_data         <=8'b0;
        tx_done         <=1'b0;
        if(txbuf_rdy)
        trans_st        <=2;
        end
  2:begin
     tx_wren         <=1'b1;
     tx_ctrl         <=1'b1;
     tx_data         <=tx_len;
     trans_st        <=3;
      end
  3:begin
     tx_wren         <=1'b0;
     tx_ctrl         <=1'b0;
     if(tx_len==0)
           trans_st        <=4;
     else if(txbuf_rdy) begin
            tx_data         <=count;
            count           <=count+1;
            tx_wren         <=1'b1;
            tx_len          <=tx_len-1;
     end
     end
4:begin
        tx_wren         <=1'b0;
        tx_ctrl         <=1'b0;
```

```
        tx_data         <=0;
        tx_len          <=16;
        tx_done         <=1'b1;
        trans_st        <=5;
      end
5:begin
            tx_done         <=1'b0;
            trans_st        <=1;
          end
endcase

wire        [7:0] rx_byte;
wire                rx_byte_rdy;

reg [7:0]      rx_byte_r;
reg                rx_rden;

always@(posedge sys_clk)
  if(rx_byte_rdy)begin
              rx_rden <=1'b1;
              rx_byte_r<=rx_byte;
       end
else begin
rx_rden<=1'b0;
end

  uart_top uart_top_dut
(
.inclk          (inclk),
.rst            (rst),
.baud_sel       (baud_sel),
.tx_wren        (tx_wren),
.tx_ctrl        (tx_ctrl),
.tx_data        (tx_data),
.tx_done        (tx_done),
.txbuf_rdy      (txbuf_rdy),
.rx_rden        (rx_rden),
.rx_byte        (rx_byte),
.rx_byte_rdy(rx_byte_rdy),
.sys_clk        (sys_clk),
.sys_rst        (sys_rst),
.rx_in          (rx_in),
.tx_out         (tx_out)
```

);

endmodule

Set this file to be top level entity and instantiate the previous program. Download the program to the board and verify it.

Fifthly, lock the pins and test.

Signal Name	Port Description	Network Label	FPGA Pin
Inclk	Clock input	C10_50MCLK	91
rst	Reset signal	KEY2	10
rx_in	Serial input data	TAG_RXD_I	144
tx_out	Serial output data	TAG_TXD_O	143

Observe the data received by PC. See Fig 10. 5

Use SignalTap II to observe the data received by FPGA.

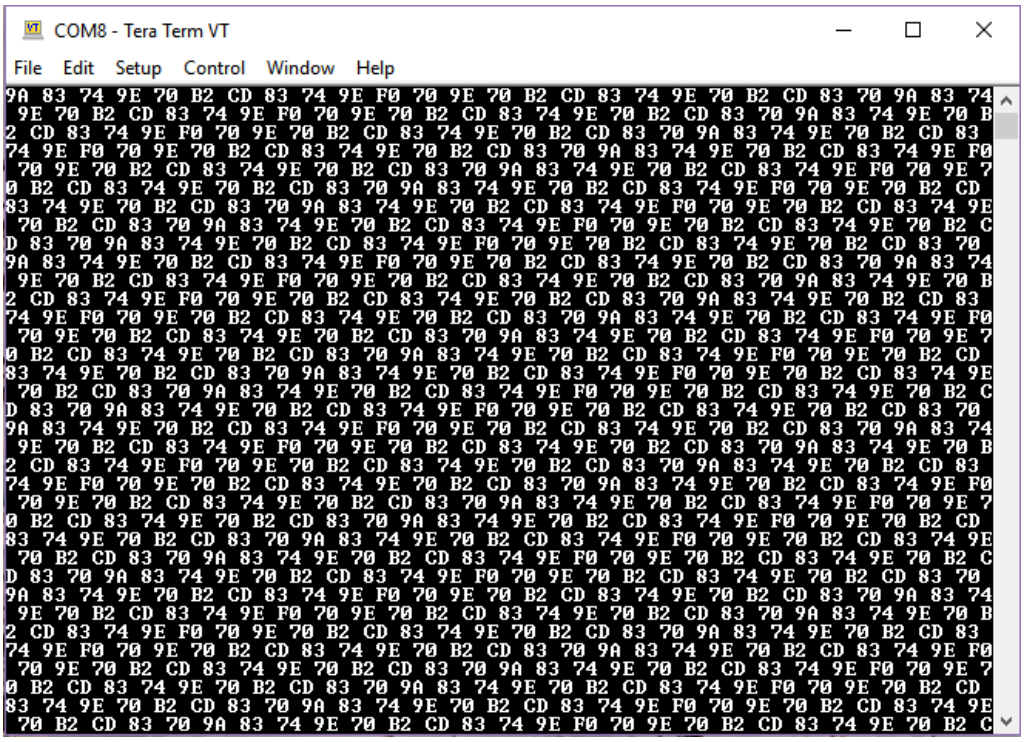

Fig 10. 5 Data sent displayed on the host computer

Sixthly, the receiving part has been eliminated here. You are encouraged to try it on your own.

Experiment 11 IIC Protocol Transmission

11.1 Experiment Objective

There is an IIC interface EEPROM chip 24LC02 in the test plate, capacity sized 2 kbit (256 bite). Due to the fact that the data is not lost after the EEPROM is powered down. Users can store some hardware setup data or user information.

1. Learning the basic principles of the different IIC bus, mastering the IIC communication protocol
2. Master the method of reading and writing EEPROM
3. Joint debugging using logic analyzer

11.2 Experiment Requirement

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 write in successfully, read the data as well. The read data is displayed directly on the segment decoders.
2. Download the program into the FPGA and press the left push button PB1 to execute the data write EEPROM operation. Press the right push button PB2 to read the data that was just written.
3. Determine whether the value read is correct or not by reading the value displayed on the segment decoders. If the segment decoders display 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 decoders.

11.3 Introduction to the IIC Agreement

11.3.1 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 phase: including serial input and output of data and response model issued by data receiver
4. IIC transmission end bit: SCL is high level, SDA jumps from low level to high level, and generates an end flag. See Fig 11. 1

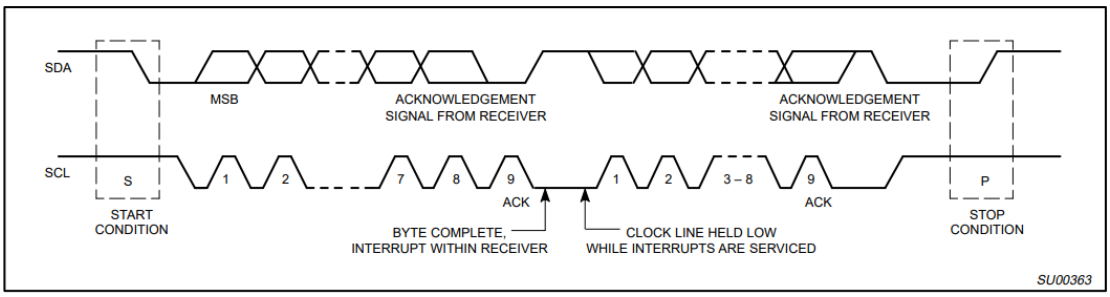

Figure 11. 1 Timing protocol of IIC

11.3.2 IIC Device Address

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 Fig 11. 2

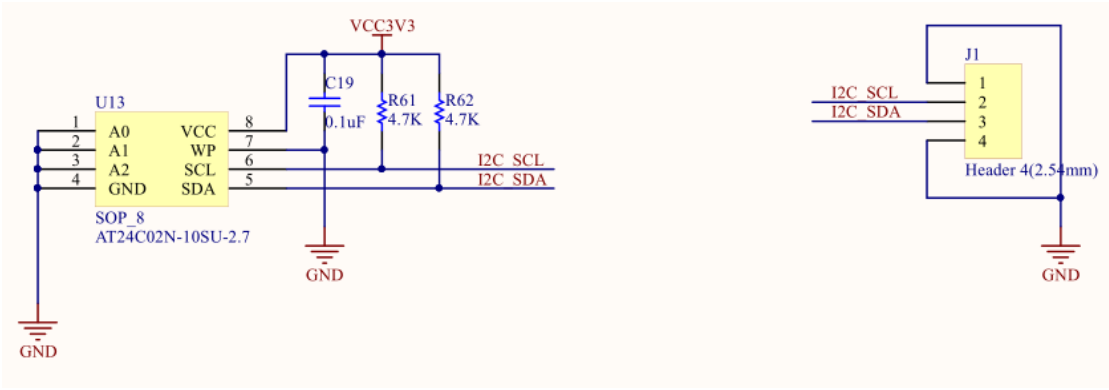

Fig 11. 2 Device schematics of IIC

11.4 The Key Code of Experiment, *IIC_COM.v*

```
module iic_com(
                clk,rst,
                data,

                sw1,sw2,
                scl,sda,
                iic_done,
                dis_data
            );

input clk;      // 50MHz
```

```
input rst;
input sw1,sw2;
inout scl;
inout sda;
output[7:0] dis_data;
input [7:0] data ;
output reg      iic_done =0 ;
reg    [7:0] data_tep;
reg    scl_link ;

reg      [19:0] cnt_5ms    ;
reg sw1_r,sw2_r;
reg[19:0] cnt_20ms;

always @ (posedge clk or posedge rst_n)
      if(rst) cnt_20ms <= 20'd0;
      else cnt_20ms <= cnt_20ms+1'b1;

always @ (posedge clk or posedge rst)
      if(rst) begin
                  sw1_r <= 1'b0;
                  sw2_r <= 1'b0;
            end
      else if(cnt_20ms == 20'hfffff) begin
                  sw1_r <= sw1;
                  sw2_r <= sw2;
            end

//-------------------------------------------

reg[2:0] cnt;
reg[8:0] cnt_delay;
reg scl_r;

always @ (posedge clk or posedge rst)
      if(rst) cnt_delay <= 9'd0;
      else if(cnt_delay == 9'd499) cnt_delay <= 9'd0;
      else cnt_delay <= cnt_delay+1'b1;

always @ (posedge clk or negedge rst) begin
      if(rst) 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 posedge rst)
      if(rst_n) data_tep <= 8'h00;
      else     data_tep<= data ;     //



always @ (posedge clk or negedge rst)
      if(rst) scl_r <= 1'b0;
      else if(cnt==3'd0) scl_r <= 1'b1;     //scl
      else if(cnt==3'd2) scl_r <= 1'b0;     //scl

assign scl = scl_link?scl_r: 1'bz ;
//----------------------------------------------


`define    DEVICE_READ       8'b1010_0001
`define DEVICE_WRITE   8'b1010_0000
`define    WRITE_DATA        8'b1000_0001
`define BYTE_ADDR       8'b0000_0011
reg[7:0] db_r;
reg[7:0] read_data;

//----------------------------------------------

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[3:0] cstate;
reg sda_r;
reg sda_link;
reg[3:0] num;


always @ (posedge clk or posedge rst) begin
    if(rst) 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;
```

```
                cstate <= ADD1;
                num <= 4'd0;
                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
                    //          sda_r <= db_r[4'd7-num];
                        end
                end
//          else if(`SCL_POS) db_r <= {db_r[6:0],1'b0};
            else cstate <= ADD1;
        end
ACK1:   begin
            if(/*!sda*/`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
            //          sda_r <= db_r[4'd7-num];
                    cstate <= ADD2;
                end
        end
//      else if(`SCL_POS) db_r <= {db_r[6:0],1'b0};
        else cstate <= ADD2;
    end
ACK2:   begin
        if(/*!sda*/`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

                        //  sda_r <= db_r[4'd7-num];
                            cstate <= ADD3;
                        end
                end
        //  else if(`SCL_POS) db_r <= {db_r[6:0],1'b0};
            else cstate <= ADD3;
        end
ACK3:   begin
            if(/*!sda*/`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

                //                  read_data[4'd7-num] <= sda;
                                    end
                    //              else        if(`SCL_NEG)        read_data        <=
{read_data[6:0],read_data[7]};
                            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

                                //    sda_r <= db_r[4'd7-num];
                                        end
                        //              else if(`SCL_POS) db_r <= {db_r[6:0],1'b0};
                                        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(/*!sda*/`SCL_NEG) begin
        //                  sda_r <= 1'b1;
                            cstate <= STOP1;
                            end
                            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  cstate  <=  IDLE;
cnt_5ms<=20'h00000;  iic_done<=1 ; end
```

```
                    else begin cstate <= STOP2      ;       cnt_5ms<=cnt_5ms+1 ; end
                end
            default: cstate <= IDLE;
            endcase
end

assign sda = sda_link ? sda_r:1'bz;
assign dis_data = read_data;

endmodule
```

11.5 Downloading to the Board

1. Lock the pins

Signal Name	Port Description	Network Label	FPGA Pin
clk	System clock 50 MHz	C10_50MCLK	91
rst	Reset, default value is low	KEY3	11
sm_db[0]	Segment decoder seg a	SEG_PA	132
sm_db [1]	Segment decoder seg b	SEG_PB	137
sm_db [2]	Segment decoder seg c	SEG_PC	133
sm_db [3]	Segment decoder seg d	SEG_PD	125
sm_db [4]	Segment decoder seg e	SEG_PE	126
sm_db [5]	Segment decoder seg f	SEG_PF	138
sm_db [6]	Segment decoder seg g	SEG_PG	135
sm_db [7]	Segment decoder seg h	SEG_DP	125
sm_cs1_n	Segment decoder seg 2	SEG_3V3_D1	142
sm_cs2_n	Segment decoder seg 1	SEG_3V3_D0	136
data[0]	DIP switch input	SW0	80
data[1]	DIP switch input	SW1	83
data[2]	DIP switch input	SW2	86
data[3]	DIP switch input	SW3	87
data[4]	DIP switch input	SW4	74
data[5]	DIP switch input	SW5	75
data[6]	DIP switch input	SW6	76
data[7]	DIP switch input	SW7	77
sw1	Write EEPROM button	KEY0	3
sw2	Read EEPROM button	KEY1	7
scl	EEPROM clock	I2C_SCL	R20
sda	EEPROM data line	I2C_SDA	R21

2. After the program is downloaded to the board, press the left push button PB1 to write

the 8-bit value represented by SW7~SW0 to EEPROM. Then press the right push button PB 2 to read the value from the written position. Observe the value displayed on the segment decoders on the develop board and the value written in the 8'h03 register of the EEPROM address (SW7~SW0) (Here, it writes to 8'h34 address). The read value is displayed on the segment decoders. See Fig 11. 3

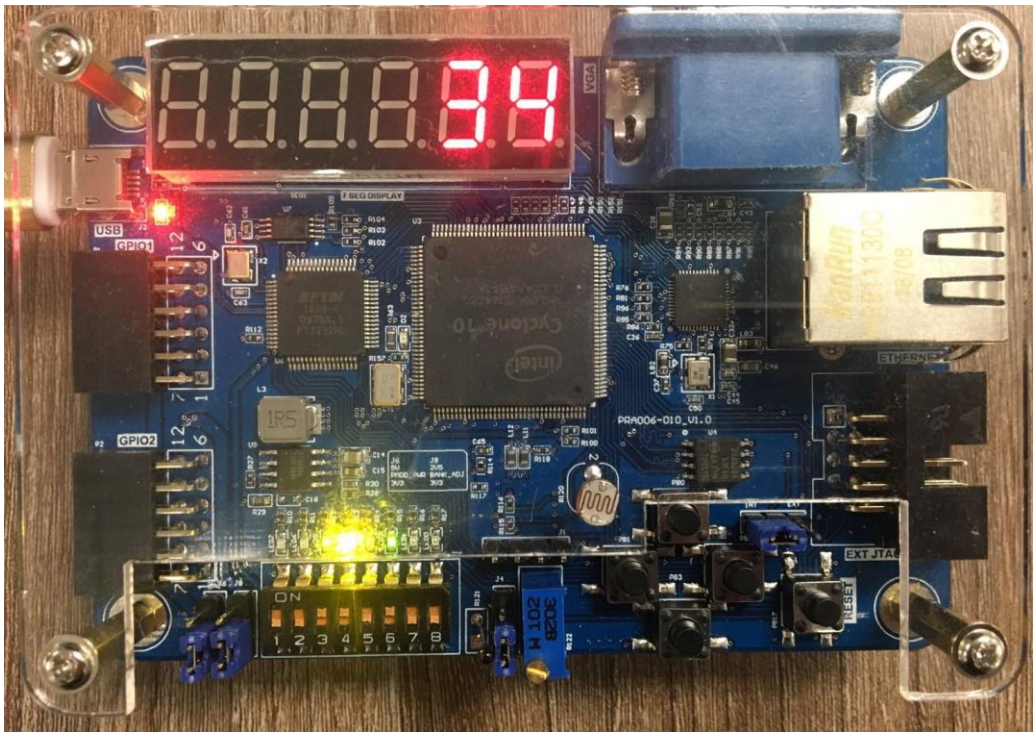

Fig 11. 3 Demonstration of develop board

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 twofold:

1. Learning the theory of AD conversion
2. Read the value of AD acquisition from PCF8591, and convert the value obtained into actual value, display it with segment decoders

12.2 Experiment Requirement

1. Perform analog-to-digital conversion using the ADC port of the chip and display the collected voltage value through the segment decoders.
2. Board downloading verification for comparison
3. Introduction to PCF8591: The PCF8591 uses the IIC bus protocol to communicate with the controller (FPGA). Please refer to the previous experiment for the contents of the IIC bus protocol. The first four bits of the device address are 1001, and the last three bits are determined by the actual circuit connection (here the circuit is grounded, so the device address is 7'b1001000). The LSB is the read/write control signal. After sending the device address information and the read/write control word are done, the control word information is sent. The specific control word information is shown in Figure 12. 1.

Bit	Slave address							
	7 MSB	6	5	4	3	2	1	0 LSB
slave address	1	0	0	1	A2	A1	A0	R/$\overline{W}$

Fig 12. 1 PCF8591 address byte

Here, the experiment uses the DIP switch (SW1, SW0) input channel as the AD acquisition input channel. Configure the control information as (8'h40). For more details, refer to the datasheet of PCF8591.

SW1，SW0	Channel Selection	Collection Object
00	0	Photosensitive Resistor Voltage Value
01	1	Thermistor Voltage Value
10	2	Adjustable Voltage Value

12.3 Experiment

1. Program design and review the top-down design method used before.

2. The top-level entity is divided into three parts: the segment decoder driver part, the AD sampling part of the PCF and the IIC serial port driver part.
3. Refer to the project file *adda_test* for the program part.

12.4 Downloading to the Board

Signal Name	Port Description	Network Label	FPGA Pin
clk	System clock 50MHz	C10_50MCLK	91
rst	Reset, default value is low	KEY3	11
sm_db[0]	Segment decoder seg a	SEG_PA	132
sm_db [1]	Segment decoder seg b	SEG_PB	137
sm_db [2]	Segment decoder seg c	SEG_PC	133
sm_db [3]	Segment decoder seg d	SEG_PD	125
sm_db [4]	Segment decoder seg e	SEG_PE	126
sm_db [5]	Segment decoder seg f	SEG_PF	138
sm_db [6]	Segment decoder seg g	SEG_PG	135
sm_db [7]	Segment decoder seg h	SEG_DP	128
data[0]	DIP switch input	SW0	80
data[1]	DIP switch input	SW1	83
data[2]	DIP switch input	SW2	86
data[3]	DIP switch input	SW3	87
data[4]	DIP switch input	SW4	74
data[5]	DIP switch input	SW5	75
data[6]	DIP switch input	SW6	76
data[7]	DIP switch input	SW7	77
scl	PCF8591 clock line	ADDA_I2C_SCL	53
sda	PCF8591 data line	ADDA_I2C_SDA	52
sel[0]	Segment decoder position selection	SEG_3V3_D0	124
sel[1]	Segment decoder position selection	SEG_3V3_D1	127
sel[2]	Segment decoder position selection	SEG_3V3_D2	129
sel[3]	Segment decoder position selection	SEG_3V3_D3	141
sel[4]	Segment decoder position selection	SEG_3V3_D4	142
sel[5]	Segment decoder position selection	SEG_3V3_D5	136

Note: The six segment decoders are reversed in order due to actual observations.

Experiment 13 VGA Experiment

13.1 Experiment Objective

1. Master the principle of VGA implementation
2. Design a simple VGA image display (The design here is a vertical color bar)

13.2 VGA Principle

VGA (Video Graphics Array) is a computer display standard that IBM introduced in 1987 using analog signals. VGA is a low standard that is supported by most manufacturers. PCs must support the VGA standard before loading their own unique drivers. The schematic diagram of the VGA PCB is shown in Figure 13. 1

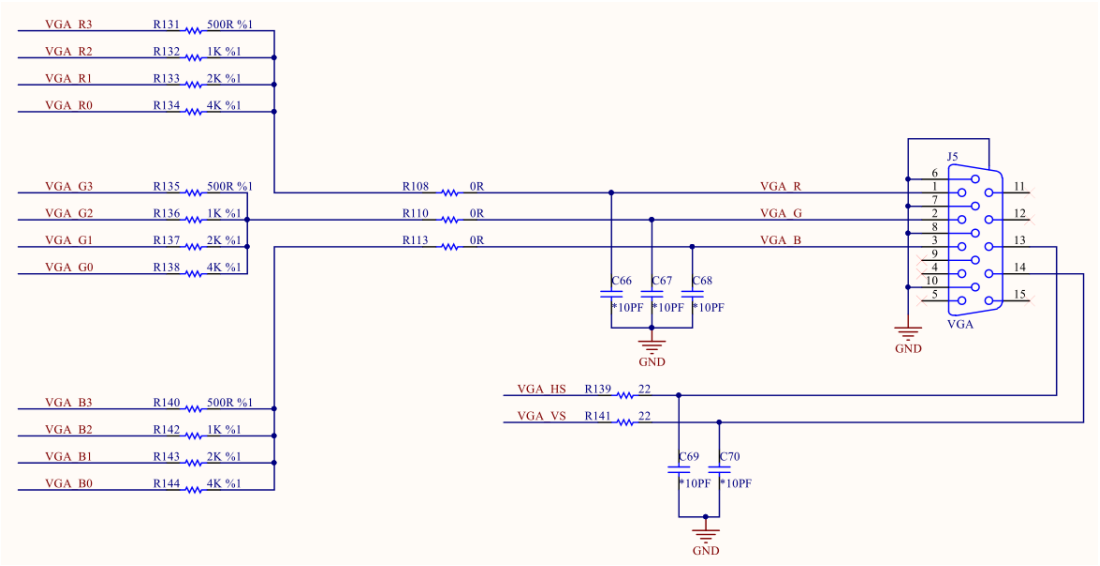

Fig 13. 1 PCB schematics

The VGA scanning mode on the display is divided into progressive scanning and interlaced scanning: progressive scanning is scanning from the top left corner of the screen, scanning from left to right point by point, each time a line is finished, the electron beam returns to the starting position of the next line on the left of the screen. During the process the CRT blanks the electron beam, and at the end of each line, synchronizes with the line sync signal; when all the lines are scanned, a frame is formed. The field sync signal is used for field synchronization and make the scan back to the top left of the screen, while performing field blanking, start the next frame. Interlaced scanning refers to scanning every other line in the scanning of electron beams. After scanning one screen and then returning to scan the remaining lines, the interlaced display flashes quickly, which may cause eye fatigue. (This experiment uses progressive scanning). See Fig 13. 2, 13. 3

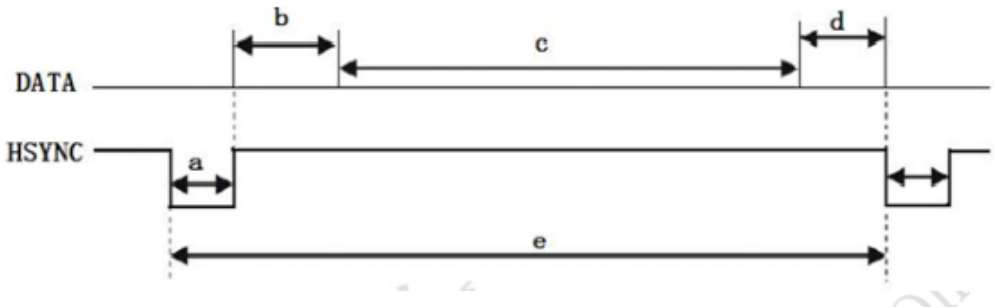

Fig 13. 2 Column synchronization timing

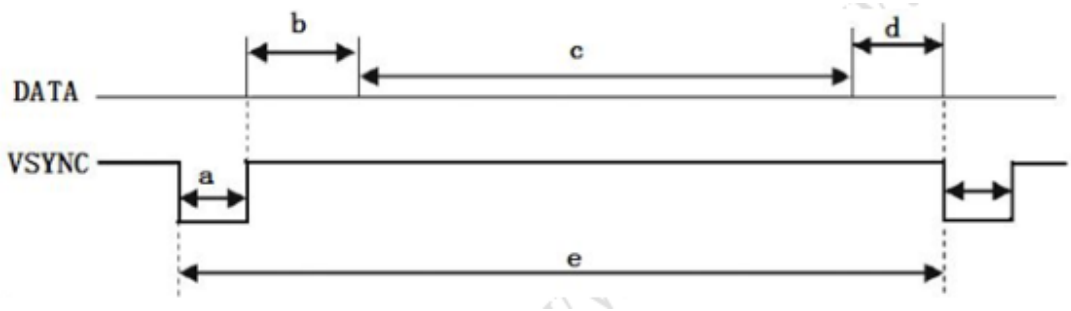

Fig 13. 3 Row synchronization timing

The definition of the row timing and column timing in the VGA requires a sync pulse (a segment), a display trailing edge (b segment), and a display timing segment (c segment) and a display leading edge (d segment). VGA industry standard display mode requirements: row synchronization, column synchronization is both negative, that is, the synchronization pulse is required to be a negative pulse. According to the VGA row timing, each row has a negative row sync pulse (a segment), which is the end mark of the data line and the start mark of the next row. After the sync pulse is the display trailing edge (b segment), during the display timing segment (c segment), the display is bright, and the RGB data drives each pixel on the row to display one row. At the end of a row is the display leading edge (d segment). No image is projected onto the screen outside the display timing period, but a blanking signal is inserted. The sync pulse, display trailing edge, and display leading edge are all within the line blanking interval. When the blanking is valid, the RGB signal is invalid and the screen does not display data.

The column timing of VGA is basically the same as the row timing analysis.

VGA also has many display standards. In this experiment, we use the standard of 640×480@60 Hz. The standard overview is shown in Fig 13. 4.

Display mode	Clock (MHz)	Columns					Rows				
		a	b	c	d	e	a	b	c	d	e
640*480*60	25. 175	96	48	640	16	800	2	33	480	10	525
640*480*75	31. 5	64	120	640	16	840	3	16	480	1	500
800*600*60	40. 0	128	88	800	40	1056	4	23	600	1	628
800*600*75	49. 5	80	160	800	16	1056	3	21	600	1	625
1024*768*60	65	136	160	1024	24	1344	6	29	768	3	806
1024*768*75	78. 8	176	176	1024	16	1312	3	28	768	1	800
1280*1024*60	108. 0	112	248	1280	48	1688	3	38	1024	1	1066
1280*800*60	83. 64	136	200	1280	64	1680	3	24	800	1	828
1440*900*60	106. 47	152	232	1440	80	1904	3	28	900	1	932

Fig 13. 4 VGA display standard

Take the display standard 640*480*60 Hz of this experiment as an example. (640 is the number of columns, 480 is the number of rows, 60 Hz is the frequency to refresh a screen). Line timing: The number of lines corresponding to the screen is 525 (a + b + c + d = e segments), of which 480 (c segment) is the display row; each row has a line synchronization signal (a segment), which is 2 row periods Low level. Column timing: Each display line consists of 800 columns (a + b + c + d = e segments), where 640 (c segment) is the valid display areas, and each row has a row sync signal (a segment) of 96 column periods Low level.

13.3 Experiment

1. Experimental design and module description
    a. Clock frequency division module (Refer to the previous experiment, call PLL)
    b. The main task of the control module (Refer to the project file *vga_driver* module) is to display the pixels to the active area.
    c. The main task of the display module (refer to the project file *vga_display* module) is to divide the display area and fill in the color as required in each block.
2. Board downloading verification
    As shown in Fig 13. 5, the screen is divided into five vertical bars, white, black, red, green, and blue.

Fig 13. 5 VGA verification

References

Quartus II Introduction Using VHDL Designs
https://www.ee.ryerson.ca/~courses/coe328/Quartus_II_Introduction-V13.pdf
https://electrosome.com/switch-debouncing/
https://www.nxp.com/docs/en/data-sheet/PCF8591.pdf