实验三 可综合时序逻辑电路实验

学号 2023370018
姓名 梁桐
班级 10012209
课程代码 U10M21001.01

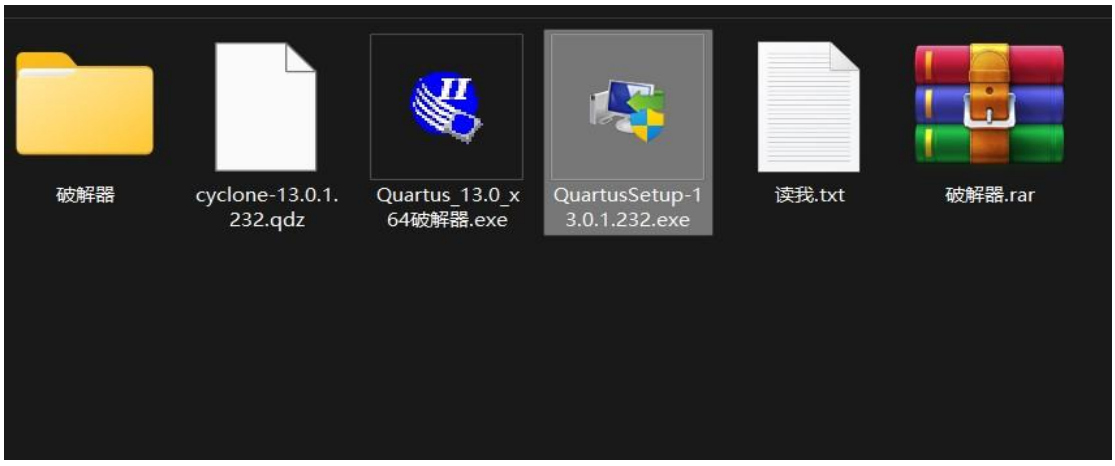
一．实验目的：

1． 掌握可综合 Verilog 语言进行时序逻辑设计的使用;
2． 学习测试模块的编写、综合和不同层次的仿真。

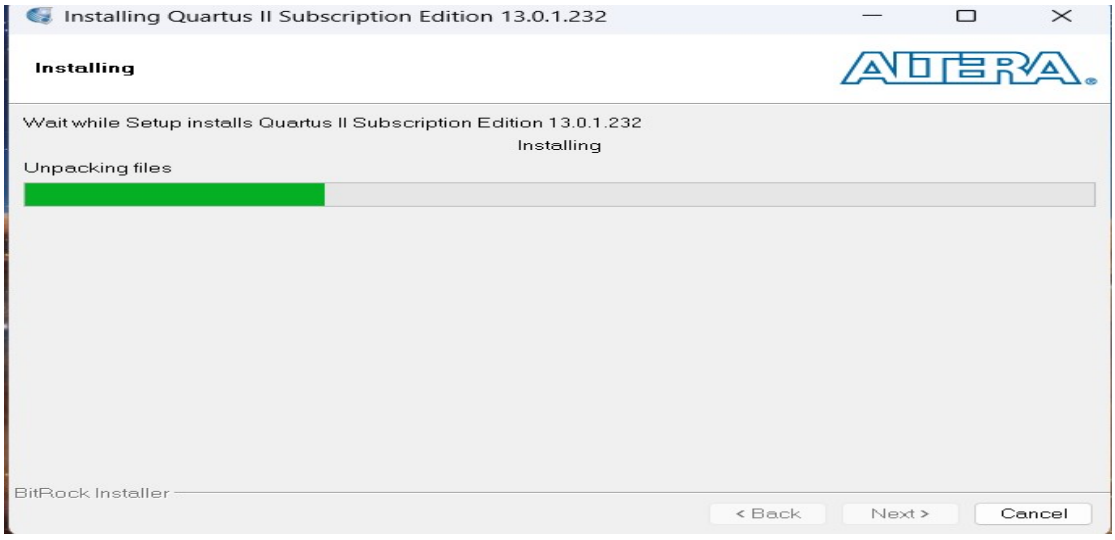
二．实验内容：

A. QuartusII 软件基本使用步骤

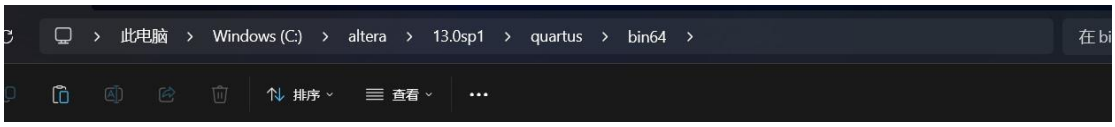
(1)点击运行 QuartusSetup-13.0.1.232.exe



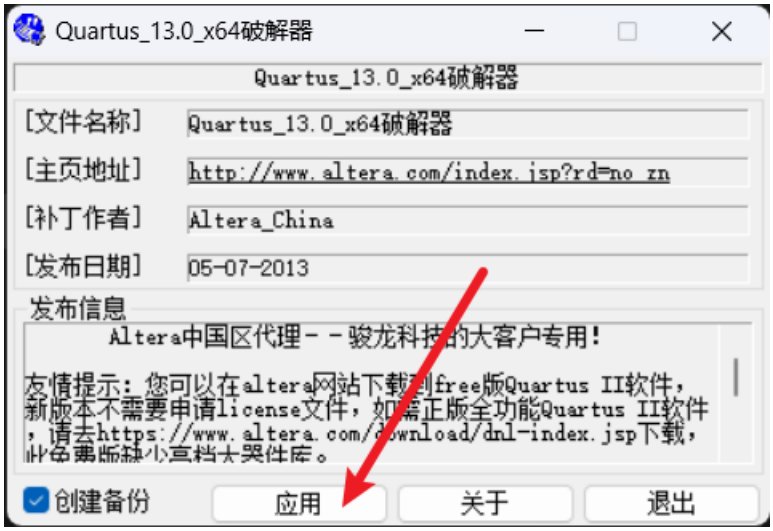
（2）按照默认步骤完成初步安装



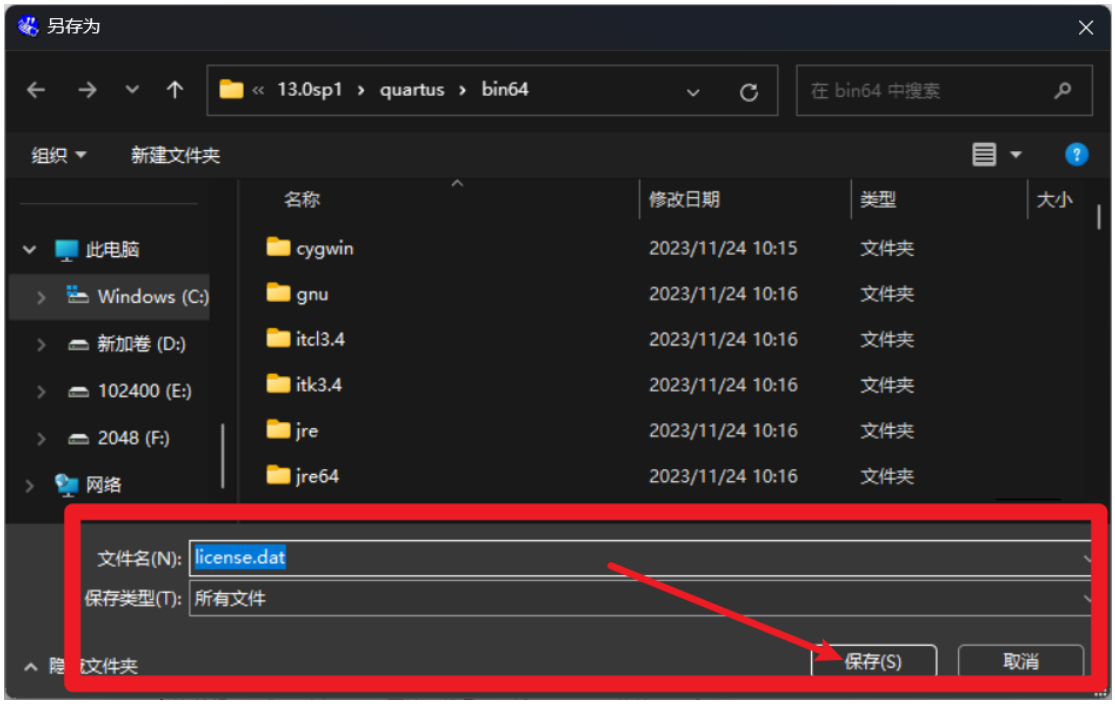
（3）将 Quartus_13.0_x64 破解器.exe 复制粘贴至 C:\altera\13.0sp1\quartus\bin64

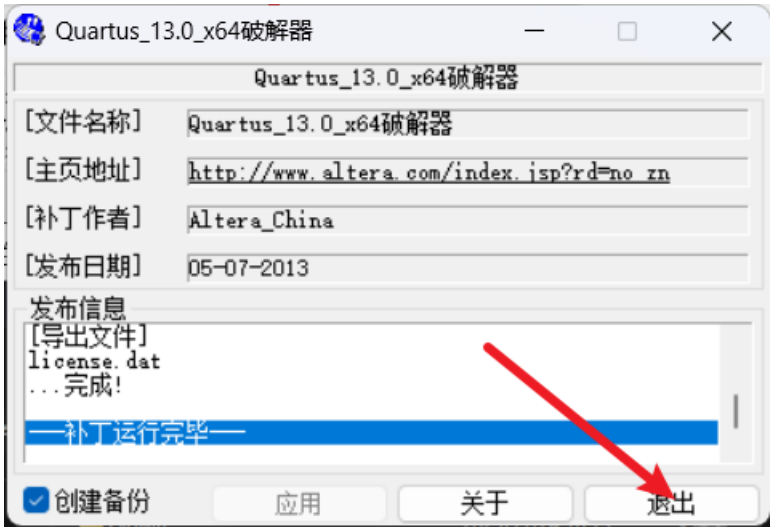


（4）双击运行粘贴后的 Quartus_13.0_x64 破解器.exe，点击应用。

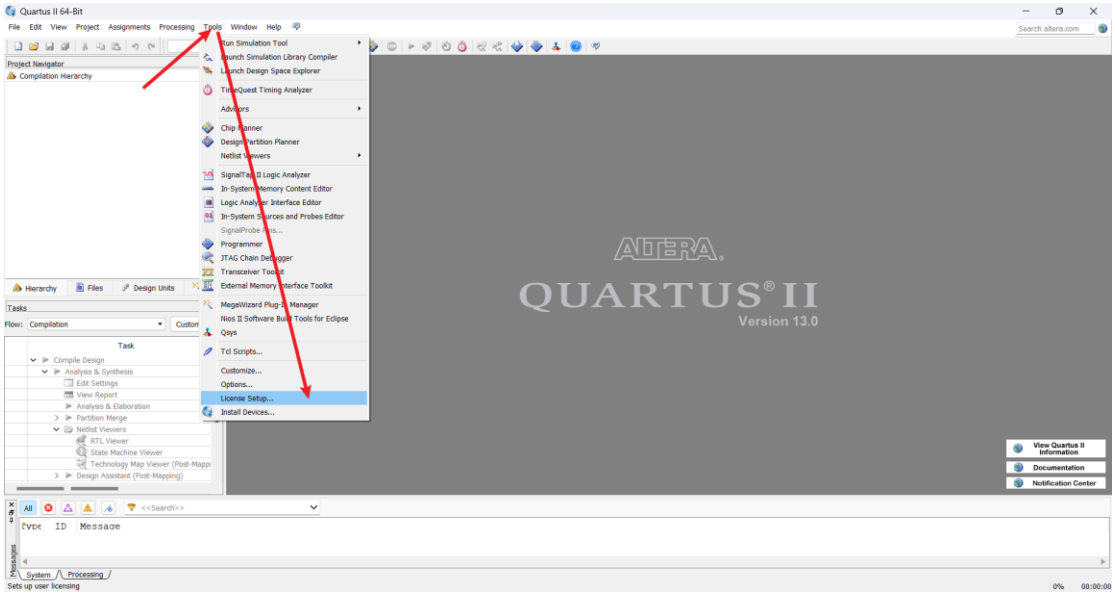


（5）保存生成的文件，点击退出。

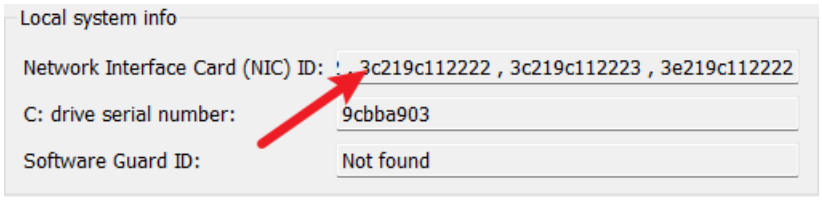


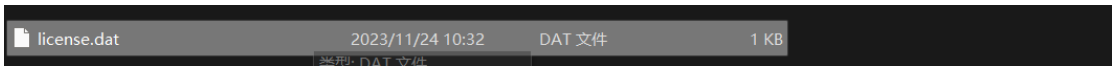


（7）双击桌面上生成的快捷方式 Quartus II 13.0sp1 (64-bit)，单击 Tools 选项，然后单击 License Setup 选项



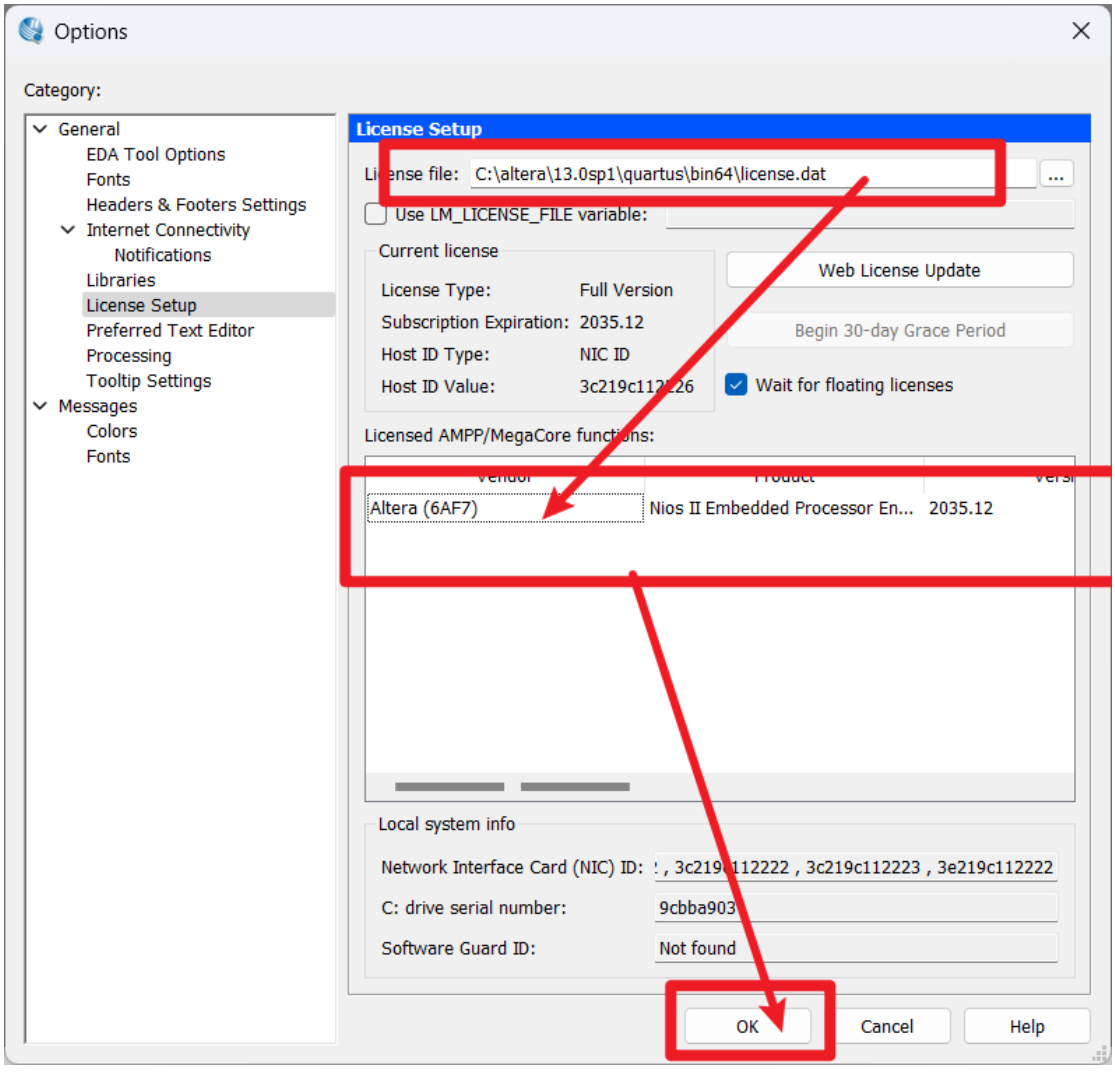
（8）复制箭头所标处的第一个 ID，然后在 C:\altera\13.0sp1\quartus\bin64 目录下找到 license.dat 文件打开，用刚刚复制的 ID 替换红圈处的 XXXXXXXXXXXX 并保存





```
FEATURE quartus alterad 2035.12 permanent uncounted 295142B536B3 \
        HOSTID=XXXXXXXXXXX SIGN="0C8D 31B5 AD64 E1C4 C6F9 1540 5072 \
        C53D 3860 7A5E 09F0 6FE0 EBAB A42C C139 015B 44B1 D3E6 8F4B \
        CD45 FAFF B30C 77BE FA54 955D 022F 0663 87C2 26B0 7305"


FEATURE 6AF7_00A2 alterad 2035.12 permanent uncounted E75BE809707E \
        VENDOR_STRING="iiiiiiiihdLkhIIIIIIIIUPDuiaaaaaaaa11X38DDDDDDDDpjz5cddddddddtmGzGJJJJJJJJbqIh0uuuuuuuugYYWiVVVVVVVVbp0FVHHHHHHHHBU
        HOSTID=XXXXXXXXXXX TS_OK SIGN="1E27 C980 33CD 38BC 5532 368B \
        116D C1F8 34E0 3436 99A0 5A2E 1C8C 8DD0 C9C6 011B A5A9 932B \
        08DE C5ED 9E62 2868 5A32 6397 D9B8 5C3A B8E8 4E4F CEC7 C836"


#license文件存放的路径名称不能包含汉字和空格，空格可以用下划线代替。
#把license.dat里的XXXXXXXXXXX用您老的网卡号替换(在Quartus II的Tools菜单下选择License Setup，下面就有NIC ID)。
```

（9）再次重复第（7）步的做法，将 License file 处的地址改为 license.dat 文件的地址，确认出现第二个红框内的信息后单击 OK，环境配置完成。



B．Quartusll 试运行

B1:在编写代码和测试代码，在 modelism 仿真出波形

代码

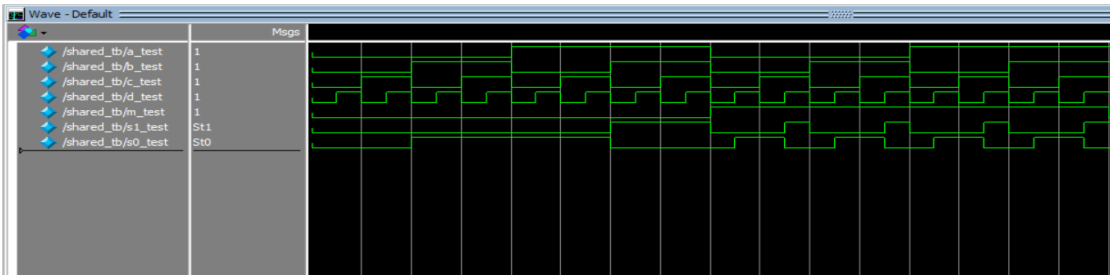
```
// `shared` 模块包含一个多路复用器（mux2to1）和一个加法器（adder）
module shared (a, b, c, d, m, s1, s0);
    input a, b, c, d, m;
    output s1, s0;
    wire w1, w2;

    // 实例化两个 mux2to1 模块，分别处理输入 a, c 和 b, d
    mux2to1 U1 (a, c, m, w1);
    mux2to1 U2 (b, d, m, w2);

    // 实例化加法器，将两个 mux2to1 模块的输出作为输入
    adder U3 (w1, w2, s1, s0);
endmodule

// `mux2to1` 模块是一个 2-to-1 多路复用器
module mux2to1 (x1, x2, s, f);
    input x1, x2, s;
    output f;

    // 多路复用器逻辑：根据选择信号 s，选择其中一个输入作为输出
    assign f = (~s & x1) | (s & x2);
endmodule

// `adder` 模块是一个加法器
module adder (a, b, s1, s0);
    input a, b;
    output s1, s0;

    // 加法器逻辑：计算输入 a 和 b 的和，s1 是和的高位，s0 是和的低位
    assign s1 = a & b;
    assign s0 = a ^ b;
endmodule
```

测试代码

```
`timescale 1ns/1ps

module shared_tb;
    reg a_test;             // 输入信号 a
    reg b_test;             // 输入信号 b
    reg c_test;             // 输入信号 c
    reg d_test;             // 输入信号 d
    reg m_test;              // 输入信号 m
```

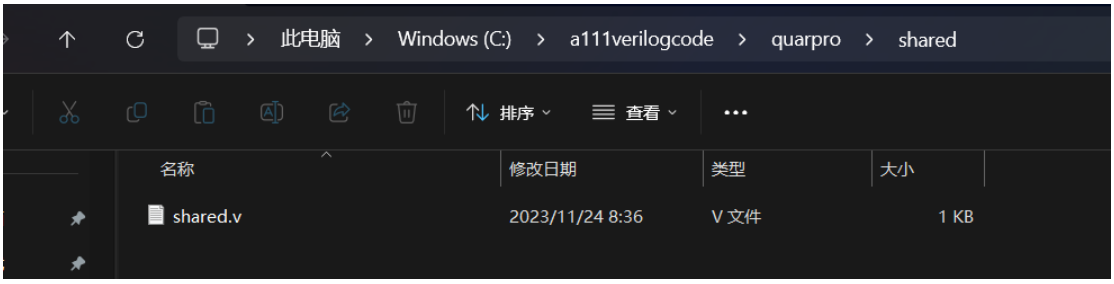
```
    wire s1_test;          // 输出信号 s1
    wire s0_test;          // 输出信号 s0

    initial
        m_test = 0;            // 初始化选择信号 m 为 0

    // 每 160 个时间单位翻转一次选择信号 m
    always #160 m_test = ~m_test;

    // 初始化输入信号 a, b, c, d
    initial begin
        a_test = 0;
        b_test = 0;
        c_test = 0;
        d_test = 0;

        // 一系列测试用例，每个测试用例间隔 10 个时间单位
        // 每个测试用例设置不同的输入信号值
        // 以测试 shared 模块的输出行为
        #10 a_test = 0; b_test = 0; c_test = 0; d_test = 1;
        #10 a_test = 0; b_test = 0; c_test = 1; d_test = 0;
        // ... （其他测试用例）

    end

    // 实例化 shared 模块
    shared UUT_shared_tb(
        .a(a_test),
        .b(b_test),
        .c(c_test),
        .d(d_test),
        .m(m_test),
        .s1(s1_test),
        .s0(s0_test)
    );

endmodule
```

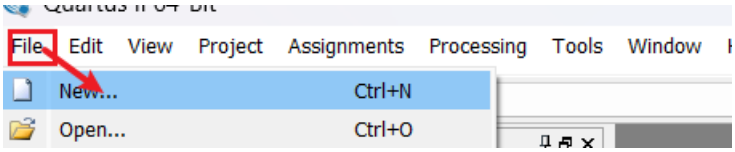
波形



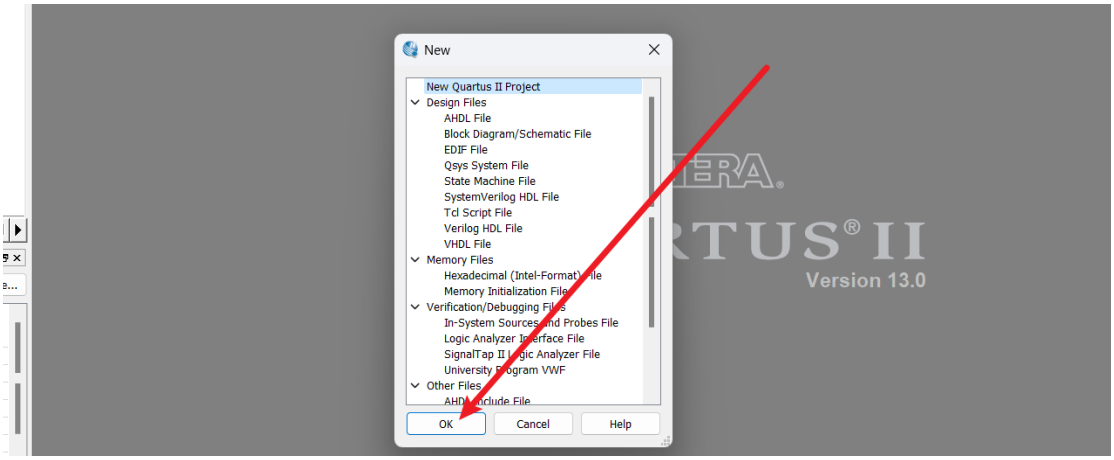
B2 仿真出电路

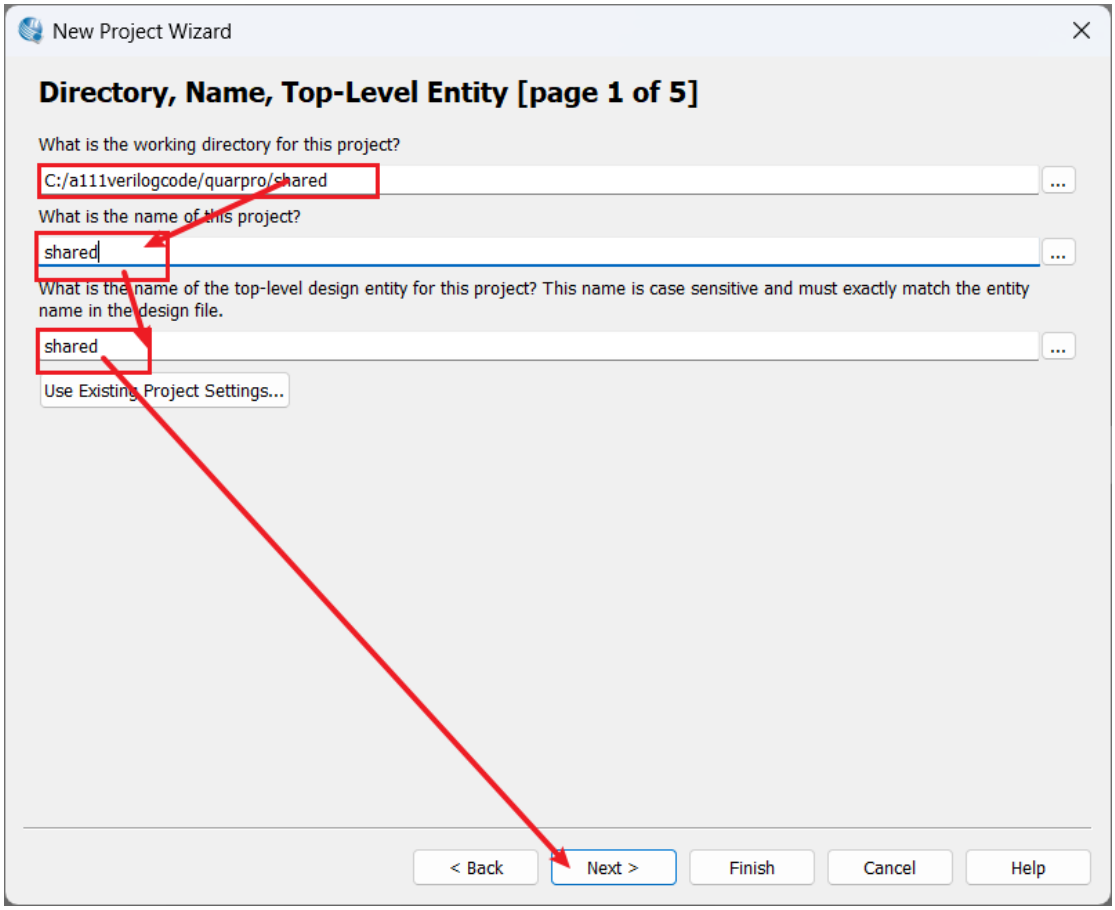
(1).新建一个文件夹，将 shared.v 文件放入文件夹中；双击运行 Quartus II 13.0sp1 (64-bit)，单击 Fire，单击 new



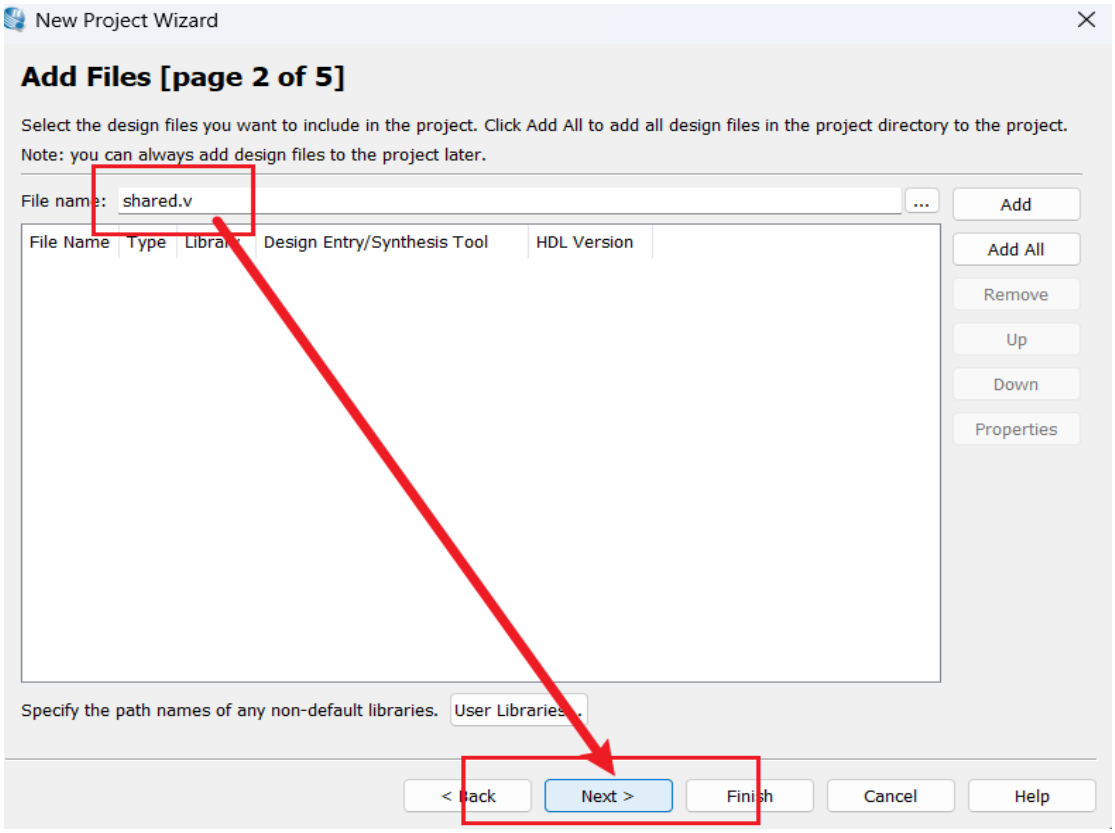


(2)单击 OK, 单击 Next, 输入刚才创建的文件的地址, 将下面两个红框中的名称改为 shared(与设计文件的 model 名称一致)，单击 Next

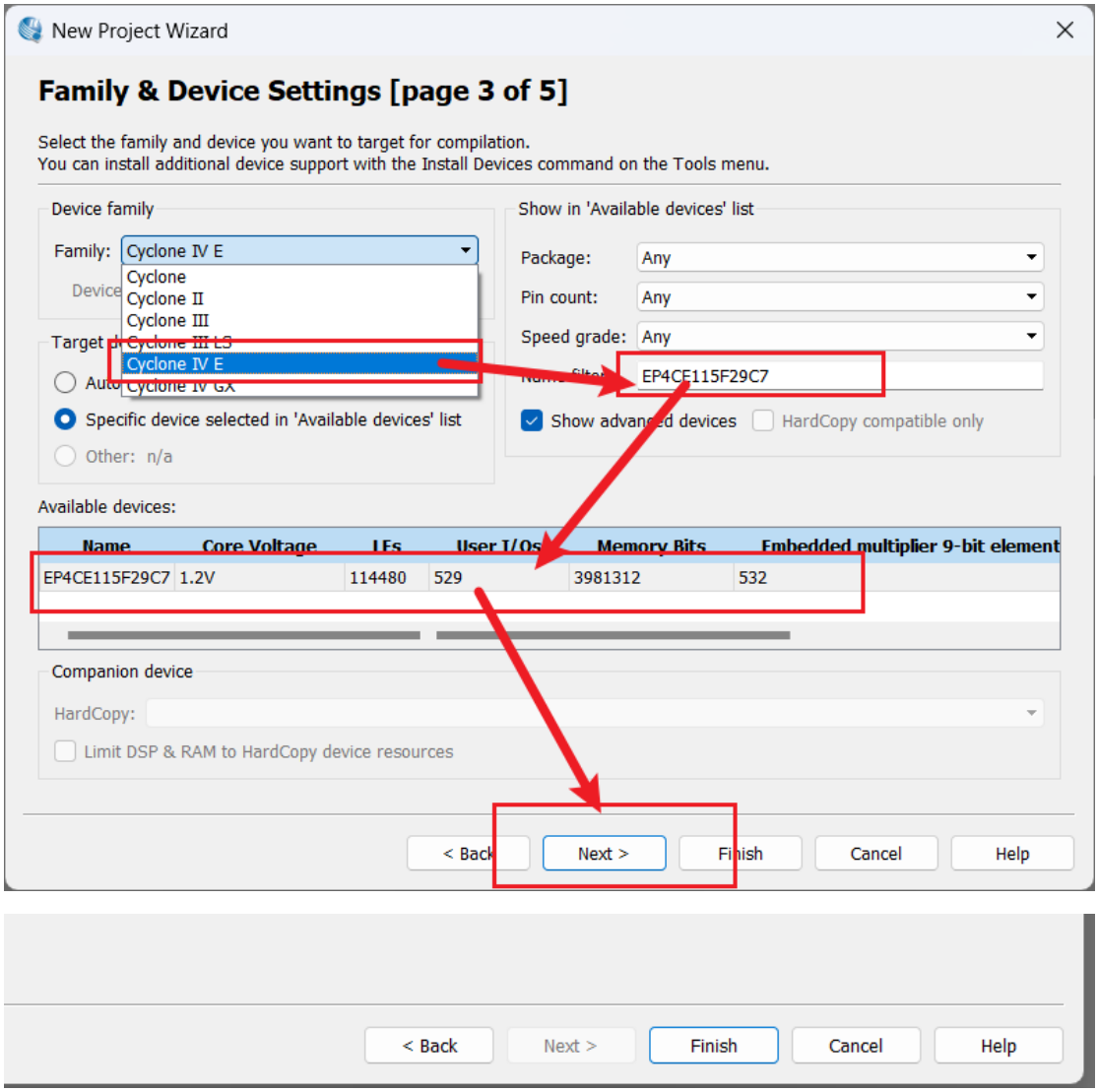




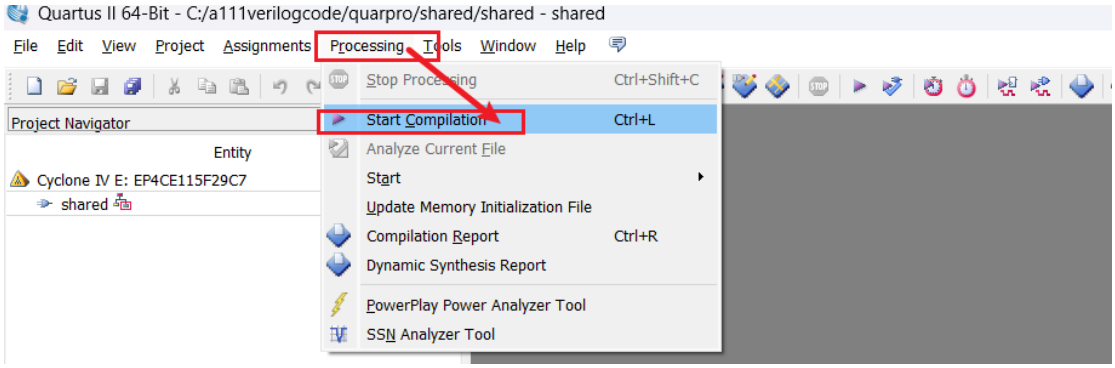
（3）File name 选项选中 share.v 文件，点击 Next，



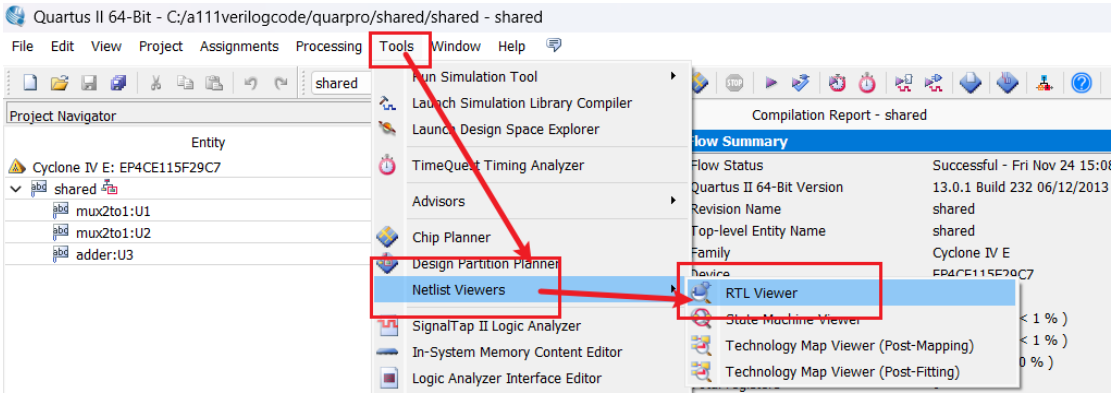
(4)Family 选项选择 Cyclone IV E,右侧输入 EP4CE115F29C7，选中之后点击 Next，Next，Finish



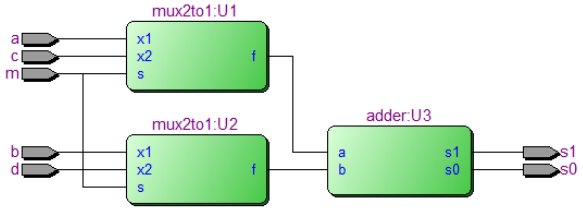
(5)单击 Processing,再单击 Start Compilation



(6)单击 Tools，再单击 Netlist Viewers,再单击 RTL Viewer



(7)生成电路图



1. 设计一款时钟上升沿触发的 D 寄存器，并编写 testbench.

代码

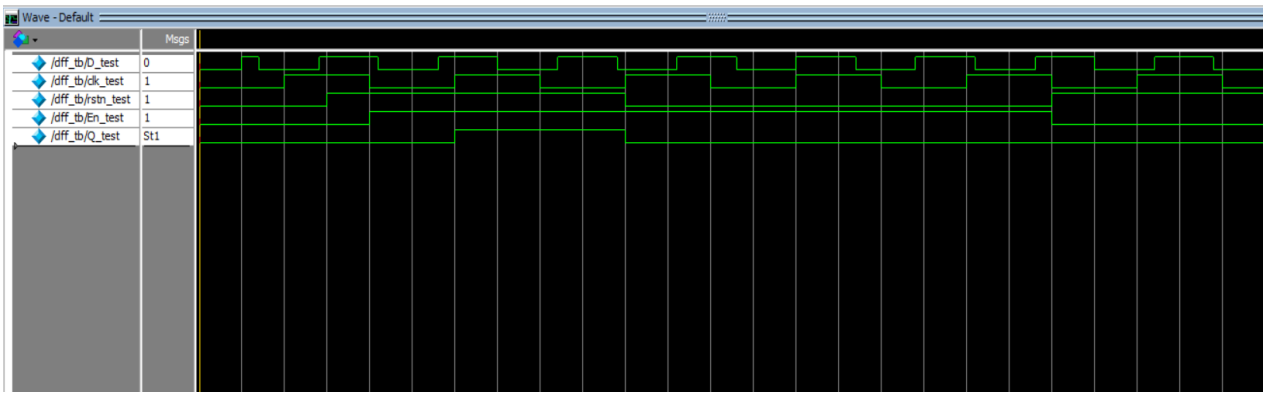
```
// DFlipFlop 模块是一个带有上升沿触发和异步复位功能的 D 触发器

module DFlipFlop (
    input D,              // 数据输入
    input clk,            // 时钟输入
    input rstn,           // 复位信号（低电平有效）
    input En,             // 使能信号
    output reg Q          // 输出信号
);

always @(posedge clk or negedge rstn)
    if (!rstn)
        Q <= 0;           // 复位时将输出 Q 置零
    else if (En)
        Q <= D;           // 在使能时，将输入数据 D 赋值给输出 Q

endmodule
```

测试代码

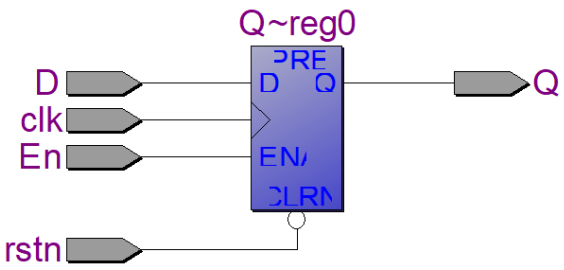
```
`timescale 1ns/1ps

module DFlipFlop _tb;

  // 输入信号
  reg D_test;          // 输入数据信号
  reg clk_test;       // 时钟信号
  reg rstn_test;     // 复位信号（低电平有效）
  reg En_test;        // 使能信号

  // 输出信号
  wire Q_test;        // 输出信号

  // 初始化
  initial begin
    D_test = 0;
    clk_test = 0;
    rstn_test = 0;
    En_test = 0;

    // Apply initial values
    #5 D_test = 1;
    #5 clk_test = 1;
    #5 rstn_test = 1;
    #5 En_test = 1;

    // Additional test cases
    #50 D_test = 0; // Test data change
    #50 En_test = 0; // Disable the flip-flop

    // End simulation
    #100 $finish;
  end

  // 时钟生成，每 10 个时间单位切换一次时钟边缘
  always #10 clk_test = ~clk_test;

  // 输入数据变化，每 7 个时间单位切换一次输入数据
  always #7 D_test = ~D_test;

  // 复位信号变化，每 50 个时间单位切换一次复位信号边沿
```

```
    always #50 rstn_test = ~rstn_test;

    // 使能信号变化，每 100 个时间单位切换一次使能信号边沿
    always #100 En_test = ~En_test;

    // 实例化 D 触发器模块
    DFlipFlop UUT_ DFlipFlop _tb (
        .D(D_test),
        .clk(clk_test),
        .rstn(rstn_test),
        .En(En_test),
        .Q(Q_test)
    );

endmodule
```

波形



电路



2．设计一款 4bBit 具有并行加载功能的移位寄存器，并编写 testbench

代码

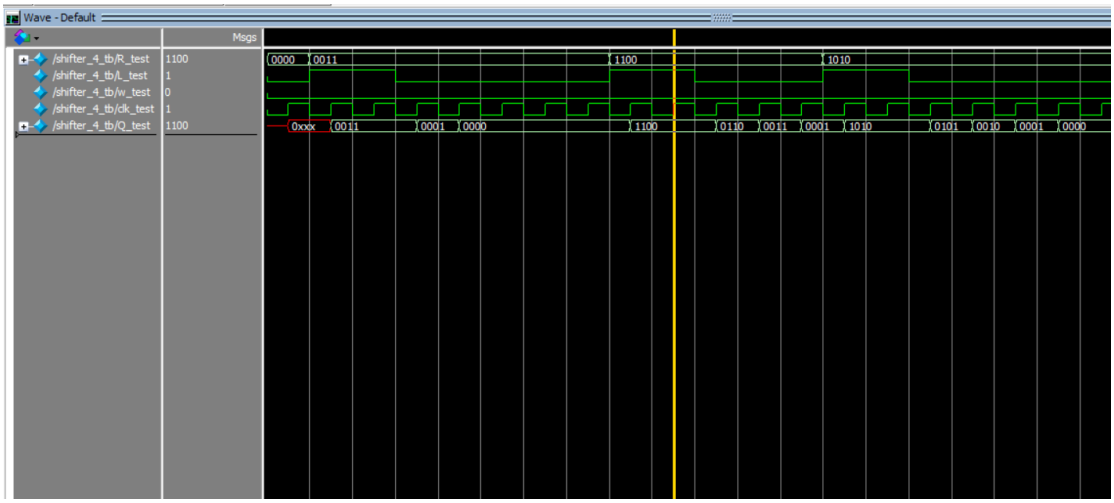
```
module 4bitSRPL (
    input [3:0] R,        // 输入数据（4 位）
    input L,              // 移位方向，1 表示左移，0 表示不移动
    input w,              // 输入的新数据
    input clk,            // 时钟信号
    output reg[3:0] Q     // 输出数据（4 位）
);
    integer k;

    always @(posedge clk) begin
        if (L) begin
            // 左移操作
            Q <= R;
        end else begin
            // 不移动，将每一位左移，最后一位用新数据 w 替代
            for (k = 0; k < 3; k = k + 1)
                Q[k] <= Q[k+1];
            Q[3] <= w;
        end
    end
endmodule
```

测试代码

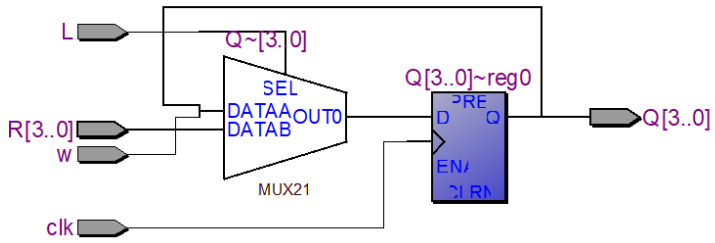
```
`timescale 1ns/1ps

module 4bitSRPL _tb;
    reg [3:0] R_test;        // 输入数据
    reg L_test;              // 移位方向，1 表示左移，0 表示不移动
    reg w_test;              // 输入的新数据
    reg clk_test;            // 时钟信号
    wire [3:0] Q_test;       // 输出数据

    initial begin
        R_test = 4'b0;       // 初始化输入数据
        L_test = 0;          // 初始化移位方向
        w_test = 0;          // 初始化新数据
        clk_test = 0;        // 初始化时钟信号

        #10
        L_test = 1;          // 设置移位方向为左移
```

```
`timescale 1ns/1ps
      R_test = 4'b0011;      // 设置输入数据

      #20
      L_test = 0;            // 设置移位方向为不移动

      #50
      L_test = 1;            // 设置移位方向为左移
      R_test = 4'b1100;      // 设置输入数据
      #20
      L_test = 0;            // 设置移位方向为不移动

      #30
      L_test = 1;            // 设置移位方向为左移
      R_test = 4'b1010;      // 设置输入数据
      #20
      L_test = 0;            // 设置移位方向为不移动

      #100
      L_test = 1;            // 设置移位方向为左移
      R_test = 4'b1111;      // 设置输入数据
      #20
      L_test = 0;            // 设置移位方向为不移动
   end

   always #5 clk_test = ~clk_test;        // 时钟信号翻转，每 5 个时间单位翻转一次

   // 实例化 4bitSRPL   模块
   4bitSRPL   UUT_4bitSRPL _tb (
      .R(R_test),
      .L(L_test),
      .w(w_test),
      .clk(clk_test),
      .Q(Q_test)
   );

endmodule
```

波形


Wave - Default
Msgs
/shifter_4_tb/R_test
/shifter_4_tb/L_test
/shifter_4_tb/w_test
/shifter_4_tb/clk_test
/shifter_4_tb/Q_test
1100
1
0
1
1100
0000
0011
1100
1010
0xxx
0011
0001
0000
1100
0110
0011
0001
1010
0101
0010
0001
0000

电路


L
Q~[3..0]
SEL
DATAA
DATAB
OUT0
MUX21
R[3..0]
w
clk
Q[3..0]~reg0
D
Q
Q[3..0]

3 .设计一款 4 bit 带复位功能的计数器，并编写 testbench 和仿真波形，请包括以下测试点：电路是否能复位？是否能按预期增加计数？是否按预定溢出？

代码

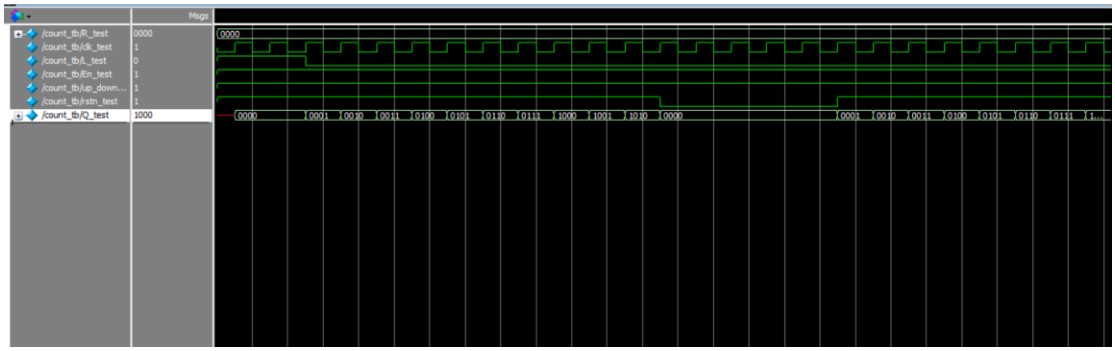
```
module Count4(
    input [3:0] R,          // 输入数据
    input clk,              // 时钟信号
    input L,                // 如果 L 为 1，Q 将被设置为输入数据 R
    input En,               // 使能信号，如果 En 为 1，Q 将递增或递减
    input up_down,          // 方向控制信号，如果 up_down 为 1，递增，否则递减
    input rstn,             // 复位信号，低电平有效
    output reg [3:0] Q      // 输出数据
);

always @(posedge clk) begin
    if (!rstn)              // 如果复位信号有效，将 Q 设置为 0
        Q <= 0;
    else if (L)             // 如果 L 为 1，将 Q 设置为输入数据 R
        Q <= R;
    else if (En)            // 如果使能信号为 1，执行递增或递减操作
        Q <= Q + (up_down ? 1 : -1);
end

endmodule
```

测试代码

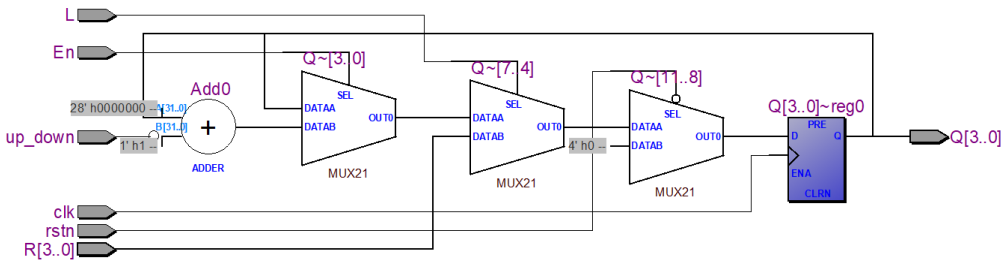
```
`timescale 1ns/1ps

module Count4_tb;
    reg [3:0] R_test;           // 输入数据
    reg clk_test;               // 时钟信号
    reg L_test;                 // 控制信号，如果 L 为 1，Q 将被设置为输入数据 R
    reg En_test;                // 使能信号，如果 En 为 1，Q 将递增或递减
    reg up_down_test;           // 方向控制信号，如果 up_down 为 1，递增，否则递减
    reg rstn_test;              // 复位信号，低电平有效
    wire [3:0] Q_test;          // 输出数据

    // 初始化
    initial begin
        R_test = 4'b0000;       // 初始化输入数据
        clk_test = 0;           // 初始化时钟信号
        L_test = 1;             // 初始化控制信号 L
        En_test = 1;            // 初始化使能信号 En
        up_down_test = 1;       // 初始化方向控制信号
```

```
        rstn_test = 1;              // 初始化复位信号

        // 在时钟的上升沿触发复位信号
        #50 L_test = 0;

        // 等待一段时间后，施加低电平有效的复位信号
        #200 rstn_test = 0;

        // 再等待一段时间后，恢复复位信号为高电平有效
        #100 rstn_test = 1;
    end

    // 每 10 个时间单位翻转一次时钟信号
    always #10 clk_test = ~clk_test;

    // 实例化 count 模块
    Count4 UUT_ Count4_tb (
        .R(R_test),
        .clk(clk_test),
        .L(L_test),
        .En(En_test),
        .up_down(up_down_test),
        .rstn(rstn_test),
        .Q(Q_test)
    );

endmodule
```

波形



电路



4．如何设计一个定时器，(如：时钟频率 20M，定时为 1 秒)并编写 testbench（在第 3 题的基础上完成）

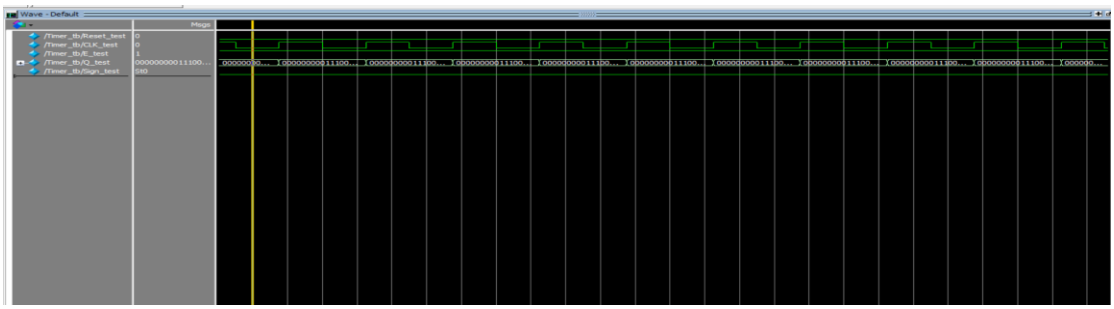
代码

```
module Timer(
    input Reset,              // 复位信号
    input CLK,                // 时钟信号
    input E,                  // 使能信号
    output reg [24:0] Q,      // 输出计时器值
    output reg Sign           // 输出标志位
);

    always @(posedge Reset or posedge CLK) begin
        if (Reset)                // 如果复位信号有效，将 Q 设置为 0
            Q <= 0;
        else if (E) begin         // 如果使能信号为 1
            if (Q < 20000000)     // 如果计时器值小于 20000000
                begin
                    Q <= Q + 1;       // 递增计时器值
                    Sign = 0;         // 标志位清零
                end
            else if (Q >= 20000000) begin
                Sign = 1;             // 如果计时器值达到 20000000，设置标志位为 1
                Q <= 0;               // 重置计时器值为 0
            end
        end
    end

endmodule
```

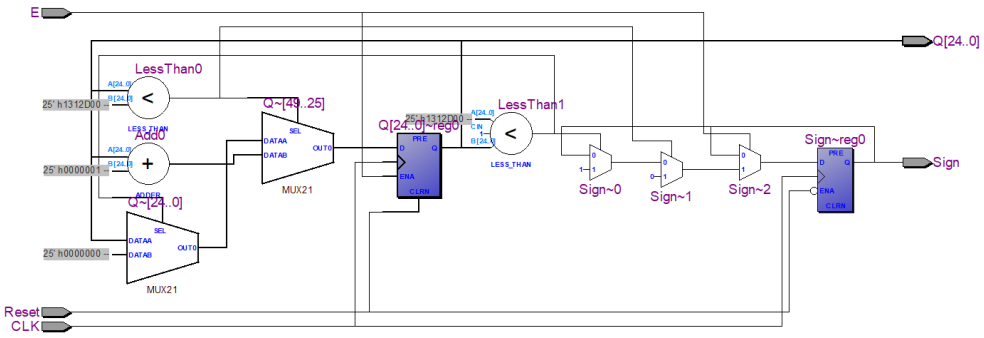
测试代码

```
`timescale 1ns/1ps

module Timer_tb;
    reg Reset_test;                 // 复位信号
    reg CLK_test;                   // 时钟信号
    reg E_test;                     // 使能信号
    wire [24:0] Q_test;             // 输出计时器值
    wire Sign_test;                 // 输出标志位

    initial
        CLK_test = 0;

    always #25 CLK_test = ~CLK_test;     // 时钟信号每 25 个时间单位翻转一次

    initial begin
        Reset_test = 1;             // 初始化复位信号为高电平
        E_test = 0;                 // 初始化使能信号为低电平

        #1
        Reset_test = 0;             // 施加低电平有效的复位信号
        E_test = 1;                 // 设置使能信号为高电平
    end

    // 实例化 Timer 模块
    Timer UUT_Timer_tb (
        .Reset(Reset_test),
        .CLK(CLK_test),
        .E(E_test),
        .Q(Q_test),
        .Sign(Sign_test)
    );

endmodule
```

波形



电路



三．实验收获与心得

本次实验中，我首次配置了 Quartus II 环境，完成了五个数字电路设计及相应的测试台。以下是我的实验心得：

在配置 Quartus II 环境的步骤中，我学会了基本的软件使用方法。通过创建项目、添加源文件、进行综合和仿真，我了解了 Quartus II 的工作流程，为我后续数字电路设计提供了一个良好的平台。

在第一项任务中，我设计了一个时钟上升沿触发的 D 寄存器。这个寄存器在时钟的上升沿对输入数据进行采样，并根据使能信号选择是否更新输出。编写了相应的 testbench，通过仿真验证了其正确性。

4 位带有并行加载功能的移位寄存器能够在并行加载模式下接收输入数据，通过控制信号实现左移或右移操作。我编写了相应的 testbench，验证了移位寄存器在不同输入条件下的正确运行。

4 位带复位功能的计数器在时钟的上升沿递增，具有复位功能，当复位信号有效时将计数器清零。编写了 testbench，包括了计数、复位、溢出等测试点，并通过仿真波形验证了计数器的功能。

基于第三项任务的计数器，定时器在 20MHz 时钟频率下实现 1 秒的定时。通过编写相应的 testbench，我验证了定时器的正确性。

本次实验让我初步了解了数字电路设计的基本流程和 Quartus II 的使用方法。通过设计不同功能的电路和编写相应的 testbench，我提升了对数字电路原理的理解，并在仿真中验证了设计的正确性。这次实验让我更加熟悉数字电路设计的流程，为今后的深入学习和应用打下了基础。