实验二 时序逻辑设计

一、实验目的

1 掌握 Verilog 语言和 Vivado、Logisim 开发平台的使用；

2 掌握基础时序逻辑电路的设计和测试方法。

二、实验内容（使用 Logisim 或 Vivado 实现）

1 锁存器、触发器的设计与测试

2 寄存器、计数器的设计与测试

3 状态机的设计与测试

三、实验要求

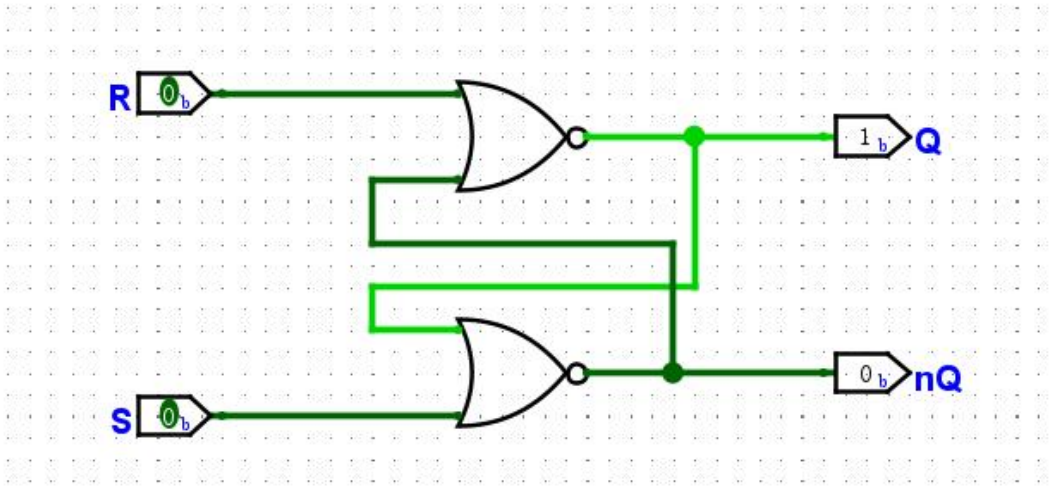
1 掌握 Vivado 或 Logisim 开发工具的使用，掌握以上电路的设计和测试方法；

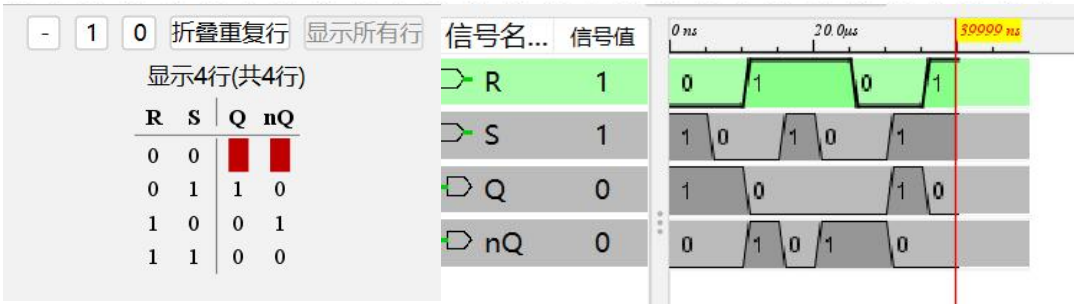
2 记录设计和调试过程（Verilog 代码/电路图/表达式/真值表，Vivado 仿真结果，Logisim 验证结果等）；

3 分析 Vivado 仿真波形/Logism 验证结果，注重输入输出之间的对应关系。

四、实验过程及分析

SR 锁存器：





SR 锁存器是一种基本的数字电路，用于存储一个比特的数据。它由两个输入端（S 和 R）和两个输出端（Q 和 Q'）组成，其中 Q 和 Q'是互补输出。SR 锁存器的原理是通过控制输入信号 S 和 R 的状态来控制输出信号 Q 和 Q'的状态。

SR 锁存器的逻辑表达式如下：

Q = S'Q' + R'Q

Q' = RQ + S'Q'

其中，S 和 R 分别代表设置和重置输入信号，Q 和 Q'分别代表输出信号。SR 锁存器可以通过两个 NOR 门实现。

SR 锁存器的工作原理如下：

当 S 和 R 均为 0 时，SR 锁存器处于保持状态，输出 Q 和 Q'保持不变。

当 S 为 1，R 为 0 时，SR 锁存器处于设置状态，输出 Q 为 1，Q'为 0。

当 S 为 0，R 为 1 时，SR 锁存器处于重置状态，输出 Q 为 0，Q'为 1。

当 S 和 R 均为 1 时，SR 锁存器处于不稳定状态，输出 Q 和 Q'的状态无法确定。

总之，SR 锁存器是一种基本的数字电路，用于存储一个比特的数据。它的原理是通过控制输入信号 S 和 R 的状态来控制输出信号 Q 和 Q'的状态。

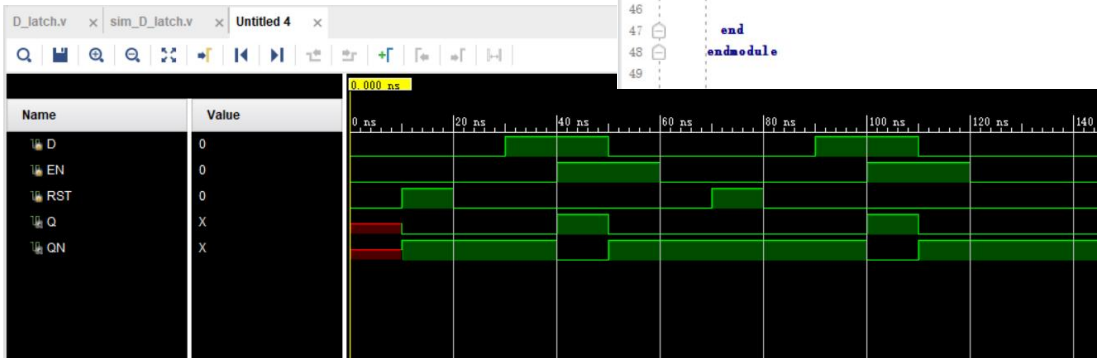
如上图，SR 锁存器可以通过两个 NOR 门来实现，真值表和时序图印证了上述原理逻辑。

锁存器：

```
module D_latch(Q, QN, D, EN, RST);
    output reg Q, QN;
    input D, EN, RST;
    always @(EN, RST, D) begin
        if(RST) begin
            Q = 0;
            QN = 1;
        end
        else if(EN) begin
            Q = D;
            QN <= ~D;
        end
    end
endmodule
```

C:/Users/Administrator/project_1/project_1.srcs/sim_1/new/sim_D_latch.v

```
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module sim_D_latch;
 reg D, EN, RST;
 wire Q, QN;

 D_latch D_latch_inst(Q, QN, D, EN, RST);

 initial begin
   D=0; EN=0; RST=0;
   #10 RST = 1;
   #10 RST = 0;

   #10 D = 1;
   #10 EN = 1;
   #10 D = 0;
   #10 EN = 0;

   #10 RST = 1;
   #10 RST = 0;

   #10 D = 1;
   #10 EN = 1;
   #10 D = 0;
   #10 EN = 0;

 end
endmodule
```



D 锁存器是一种基本的数字电路，用于存储一个比特的数据。它由一个输入端（D）和两个输出端（Q 和 QN）组成，其中 Q 和 QN 是互补输出。

程序的含义如下：

module D_latch(Q, QN, D, EN, RST);：定义了一个名为 D_latch 的模块，该模

块有四个输入输出端口，分别是 Q、QN、D、EN 和 RST。
output reg Q, QN;：定义了两个输出信号 Q 和 QN，使用 reg 类型表示，表示可以在 always 块中进行赋值。
input D, EN, RST;：定义了三个输入信号 D、EN 和 RST。
always @(EN, RST, D) begin：always 块表示在输入信号 EN、RST 和 D 变化时执行以下逻辑。
if(RST) begin：如果 RST 信号为 1，表示复位状态，则执行以下逻辑。
Q = 0;：将输出信号 Q 设置为 0。
QN = 1;：将输出信号 QN 设置为 1。
else if(EN) begin：如果 EN 信号为 1，表示使能状态，则执行以下逻辑。
Q = D;：将输出信号 Q 设置为输入信号 D 的值。
QN <= 'D;：将输出信号 QN 设置为输入信号 D 的互补值。
以上代码实现了 D 锁存器的功能。当输入信号 RST 为 1 时，D 锁存器处于复位状态，输出信号 Q 为 0，输出信号 QN 为 1。当输入信号 EN 为 1 时，D 锁存器处于使能状态，输出信号 Q 为输入信号 D 的值，输出信号 QN 为输入信号 D 的互补值。在其他情况下，D 锁存器保持上一状态不变。

module sim_D_latch;：定义了一个名为 sim_D_latch 的模块。
reg D, EN, RST;：定义了三个寄存器类型的变量 D、EN 和 RST，用于模拟输入信号。
wire Q, QN;：定义了两个线类型的变量 Q 和 QN，用于模拟输出信号。
D_latch D_latch_inst(Q, QN, D, EN, RST);：实例化了一个 D_latch 模块，将输入输出信号连接到对应的变量。
initial begin：initial 块表示仿真开始时执行以下逻辑。
#10 RST = 1;：在仿真时间 10 个时间单位后，将 RST 信号置为 1，表示进行复位。
#10 RST = 0;：在仿真时间 10 个时间单位后，将 RST 信号置为 0，表示结束复位。
#10 D = 1;：在仿真时间 10 个时间单位后，将输入信号 D 置为 1。
#10 EN = 1;：在仿真时间 10 个时间单位后，将输入信号 EN 置为 1。
#10 D = 0;：在仿真时间 10 个时间单位后，将输入信号 D 置为 0。
#10 EN = 0;：在仿真时间 10 个时间单位后，将输入信号 EN 置为 0。
#10 RST = 1;：在仿真时间 10 个时间单位后，将 RST 信号置为 1，再次进行复位。
#10 RST = 0;：在仿真时间 10 个时间单位后，将 RST 信号置为 0，表示结束复位。
#10 D = 1;：在仿真时间 10 个时间单位后，将输入信号 D 置为 1。
#10 EN = 1;：在仿真时间 10 个时间单位后，将输入信号 EN 置为 1。
#10 D = 0;：在仿真时间 10 个时间单位后，将输入信号 D 置为 0。
#10 EN = 0;：在仿真时间 10 个时间单位后，将输入信号 EN 置为 0。
以上仿真程序通过改变输入信号的值和时间来测试 D 锁存器的功能。

通过时序图可以看出 D，EN，RST 三个信号的变化随着程序的控制进行。导致了 Q 和 QN 的变化。时序图验证了 D 锁存器的以下功能：当输入信号 RST 为 1 时，D

锁存器处于复位状态，输出信号 Q 为 0，输出信号 QN 为 1。当输入信号 EN 为 1 时，D 锁存器处于使能状态，输出信号 Q 为输入信号 D 的值，输出信号 QN 为输入信号 D 的互补值。在其他情况下，D 锁存器保持上一状态不变。

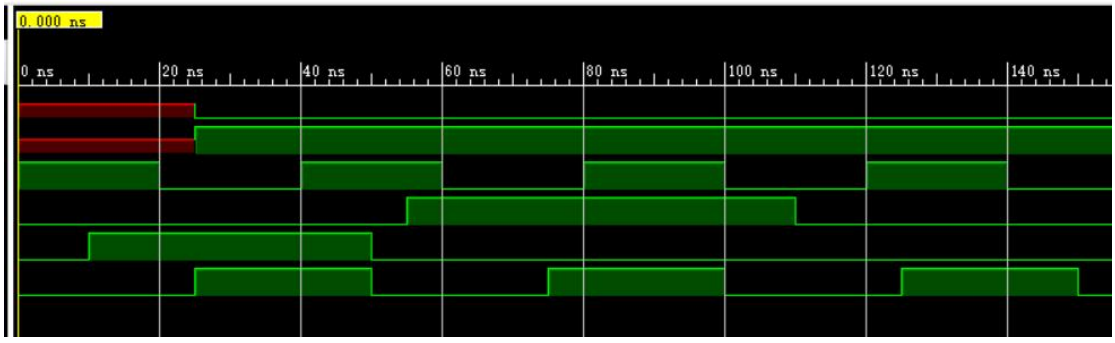
触发器：

Project Summary | D_ff.v | sim_D_ff.v

C:/Users/Administrator/project_2/project_2.srcs/sources_1/new/D_ff.v

```
// Company:
// Engineer:
//
// Create Date: 2023/10/22 14:43:08
// Design Name:
// Module Name: D_ff
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module D_ff(Q, QN, D, EN, RST, CLK);
    output reg Q, QN;
    input D, EN, RST, CLK;
    always @(posedge CLK) begin
        if(RST) begin Q <= 1'b0; QN <= 1'b1; end
        else if(EN) begin Q <= D; QN <= ~D; end
    end
endmodule
```

```
module sim_D_ff;
    wire Q,QN;
    reg D,EN,RST,CLK;
    D_ff ff_1(Q,QN,D,EN,RST,CLK);
    always #25 CLK=~CLK;
    always #20 D=~D;
    always #55 EN=~EN;
    initial begin
        D=1;CLK=0;EN=0;RST=0;
        #10 RST=1;
        #40 RST=0;
    end

endmodule
```



定义了一个 D 触发器模块（D_ff），它具有以下输入和输出：
output reg Q, QN;：输出信号 Q 和 QN，使用寄存器类型（reg）声明，表示 D 触发器的输出。

input D, EN, RST, CLK;：输入信号 D、使能信号 EN、复位信号 RST 和时钟信号 CLK，表示 D 触发器的输入。
always @(posedge CLK) begin：使用 always 块来定义时钟边沿触发的行为。
if(RST) begin Q <= 1'b0; QN <= 1'b1; end：如果复位信号 RST 为 1，则将输出信号 Q 设置为 0，输出信号 QN 设置为 1。
else if(EN) begin Q <= D; QN <= ~D; end：否则，如果使能信号 EN 为 1，则将输出信号 Q 设置为输入信号 D 的值，输出信号 QN 设置为输入信号 D 的互补值。
以上程序实现了 D 触发器的功能。在每个时钟上升沿时，根据输入信号的状态，更新输出信号的值。当复位信号 RST 为 1 时，D 触发器处于复位状态，输出信号 Q 为 0，输出信号 QN 为 1。当使能信号 EN 为 1 时，D 触发器处于使能状态，输出信号 Q 为输入信号 D 的值，输出信号 QN 为输入信号 D 的互补值。在其他情况下，D 触发器保持上一状态不变。

该模块包含了一个 D 触发器（D_ff）的实例化。
wire Q, QN;：定义了两个输出信号 Q 和 QN，使用 wire 类型声明。
reg D, EN, RST, CLK;：定义了四个输入信号 D、EN、RST 和 CLK，使用 reg 类型声明。
D_ff ff_1(Q, QN, D, EN, RST, CLK);：实例化了一个 D 触发器（D_ff）模块，将输入输出信号连接到实例化模块的对应端口。
always #25 CLK = ~CLK;：使用 always 块来定义一个时钟信号的周期性翻转。每隔 25 个时间单位，时钟信号 CLK 的值取反。
always #20 D = ~D;：使用 always 块来定义一个输入信号 D 的周期性翻转。每隔 20 个时间单位，输入信号 D 的值取反。
always #55 EN = ~EN;：使用 always 块来定义一个输入信号 EN 的周期性翻转。每隔 55 个时间单位，输入信号 EN 的值取反。
initial begin：initial 块表示仿真开始时执行以下逻辑。
D = 1; CLK = 0; EN = 0; RST = 0;：在仿真开始时，将输入信号 D 置为 1，时钟信号 CLK 置为 0，使能信号 EN 置为 0，复位信号 RST 置为 0。
#10 RST = 1;：在仿真时间 10 个时间单位后，将复位信号 RST 置为 1，表示进行复位操作。
#40 RST = 0;：在仿真时间 40 个时间单位后，将复位信号 RST 置为 0，表示结束复位操作。
以上仿真文件通过改变输入信号的值和时间来测试 D 触发器的功能。通过观察输出信号 Q 和 QN 的变化，可以验证 D 触发器是否按照预期工作。

时序图分析：
在仿真开始时，D 触发器处于复位状态，输出信号 Q 为 0，输出信号 QN 为 1。
在仿真时间 10 个时间单位后，复位信号 RST 被置为 1，D 触发器进入复位状态，输出信号 Q 为 0，输出信号 QN 为 1。
在仿真时间 40 个时间单位后，复位信号 RST 被置为 0，D 触发器退出复位状态。
在仿真时间 55 个时间单位后，使能信号 EN 被置为 1，D 触发器进入使能状态，根据输入信号 D 的值，输出信号 Q 和 QN 会相应改变。
在仿真时间 75 个时间单位后，输入信号 D 被翻转，D 触发器在下一个时钟上升沿时根据新的输入信号 D 的值更新输出信号 Q 和 QN。

在仿真时间 80 个时间单位后，时钟信号 CLK 被翻转，D 触发器在下一个时钟上升沿时根据当前输入信号 D 的值更新输出信号 Q 和 QN。

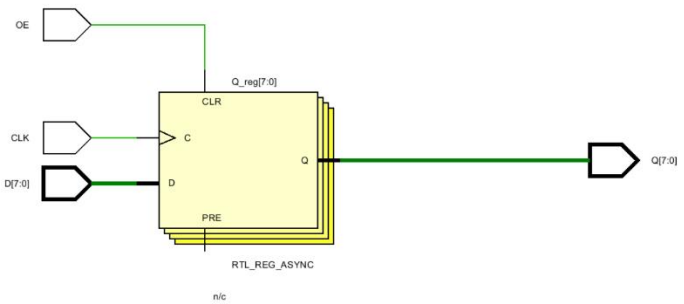
寄存器：

Project Summary | register.v * | C:/Users/Administrator/project_3/project_3.srcs/sources_1/new/register.v

```
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module register (
  Q, D, OE, CLK
);
    parameter I = 8;

    output reg [I-1:0] Q;
    input [I-1:0] D;
    input OE, CLK;

    always @(posedge CLK or posedge OE) begin
        if (OE)
          Q <= 8'bZZZZ_ZZZZ;
        else
          Q <= D;
    end
endmodule
```

register.v | sim_register.v | Untitled 22 | C:/Users/Administrator/project_3/project_3.srcs/sim_1/new/sim_registe

```
//////////////////////////////////////////////////////////////////////////////////


module sim_register( );
parameter I = 8;

  reg [I-1:0] D;
  reg OE, CLK;

  wire [I-1:0] Q;

  register u1 (
      .Q(Q),
      .D(D),
      .OE(OE),
      .CLK(CLK)
  );

  initial begin

      D = 8'b1010_1110;
      OE = 1;
      CLK = 0;
  end

  always #10 CLK=~CLK;
  always #20 OE=~OE;


endmodule
```



定义了一个名为 register 的模块，该模块实现了一个带有输出使能的寄存器。
parameter I = 8;：定义了一个参数 I，表示寄存器的位宽，默认为 8。
output reg [I-1:0] Q;：定义了一个输出信号 Q，使用 reg 类型声明，位宽为

I-1 到 0。
input [I-1:0] D;：定义了一个输入信号 D，位宽为 I-1 到 0。
input OE, CLK;：定义了两个输入信号 OE 和 CLK，分别表示输出使能和时钟。
always @(posedge CLK or posedge OE) begin：使用 always 块来定义一个时钟上升沿或输出使能上升沿触发的逻辑。
if (OE)：如果输出使能信号 OE 为 1，执行以下逻辑。
Q <= 8'bZZZZ_ZZZZ;：将输出信号 Q 的值置为 8 个高阻态（Z）。
else：否则，执行以下逻辑。
Q <= D;：将输出信号 Q 的值赋为输入信号 D 的值。
以上程序实现了一个带有输出使能的寄存器。当输出使能信号 OE 为 1 时，输出信号 Q 的值被置为高阻态（Z），表示输出无效。当输出使能信号 OE 为 0 时，输出信号 Q 的值被赋为输入信号 D 的值，表示输出有效。在时钟上升沿或输出使能上升沿触发时，根据输出使能信号 OE 的值来更新输出信号 Q 的值。

定义了一个名为 sim_register 的模块，该模块用于对寄存器进行仿真。
parameter I = 8;：定义了一个参数 I，表示寄存器的位宽，默认为 8。
reg [I-1:0] D;：定义了一个寄存器的输入信号 D，位宽为 I-1 到 0。
reg OE, CLK;：定义了两个寄存器的输入信号 OE 和 CLK，分别表示输出使能和时钟。
wire [I-1:0] Q;：定义了一个寄存器的输出信号 Q，位宽为 I-1 到 0。
register u1 (.Q(Q), .D(D), .OE(OE), .CLK(CLK));：实例化了一个名为 u1 的寄存器模块，并将输入输出信号连接到模块的对应端口。
initial begin：使用 initial 块来初始化寄存器的输入信号。
D = 8'b1010_1110;：将输入信号 D 的值初始化为 8 位二进制数 1010_1110。
OE = 1;：将输出使能信号 OE 的值初始化为 1。
CLK = 0;：将时钟信号 CLK 的值初始化为 0。
always #10 CLK=~CLK;：使用 always 块来定义一个每隔 10 个时间单位翻转一次时钟信号 CLK 的逻辑。
always #20 OE=~OE;：使用 always 块来定义一个每隔 20 个时间单位翻转一次输出使能信号 OE 的逻辑。
以上仿真程序实例化了一个寄存器模块，并通过初始化输入信号的值和周期性翻转时钟和输出使能信号的值来对寄存器进行仿真。

时序图：
初始状态下，输入信号 D 为 8'b1010_1110，输出使能信号 OE 为 1，时钟信号 CLK 为 0。
经过 10 个时间单位后，时钟信号 CLK 翻转为 1。
在时钟信号上升沿触发时，寄存器模块会根据输入信号 D 和输出使能信号 OE 的值来更新输出信号 Q 的值。由于输出使能信号 OE 为 1，输出信号 Q 应该被置为高阻态（Z）。
经过 20 个时间单位后，输出使能信号 OE 翻转为 0。
在输出使能信号 OE 为 0 时，寄存器模块会根据输入信号 D 的值来更新输出信号 Q 的值。根据输入信号 D 的值为 8'b1010_1110，输出信号 Q 应该被赋为

8’b1010_1110。
经过 10 个时间单位后，时钟信号 CLK 翻转为 0。
经过 10 个时间单位后，时钟信号 CLK 翻转为 1。
在时钟信号上升沿触发时，寄存器模块会根据输入信号 D 和输出使能信号 OE 的值来更新输出信号 Q 的值。由于输出使能信号 OE 为 0，输出信号 Q 应该保持为上一次更新的值，即 8’b1010_1110。

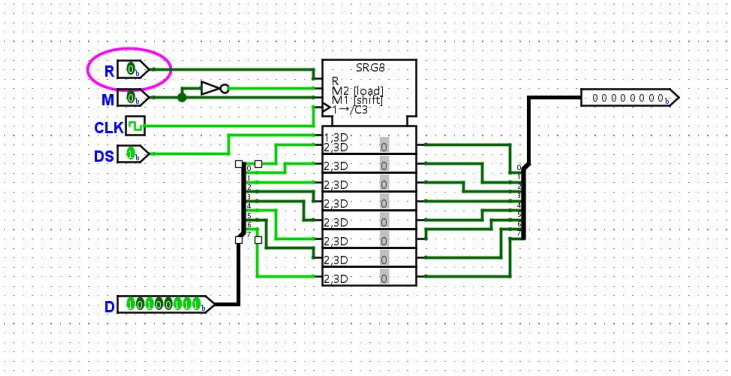
移位寄存器：

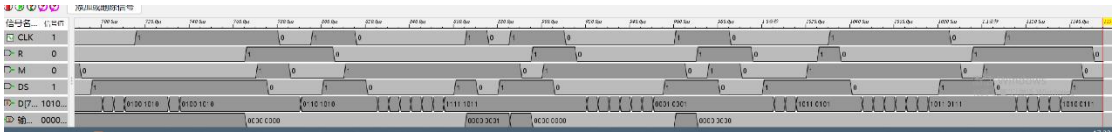
```
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module shift_register (S1, S0, D,  Dsl, Dsr, Q, CLK,CR);
    parameter  N= 4;
    input S1, S0;
    input Dsl, Dsr;
    input CLK, CR;
    input [N-1:0] D;
    output [N-1:0] Q;
     reg [N-1:0] Q;

always @ (posedge CLK or posedge CR)
 if (CR)
      Q <= 2'b00;
 else
      case ({S1, S0})
         2'b00: Q <= Q;
         2'b01: Q <= {Dsr, Q[N-1:1]};
         2'b10: Q <= {Q[N-2:0], Dsl};
         2'b11: Q <= D;
      endcase
endmodule
```

shift_register.v | sim_shift.v | Untitled 26

C:/Users/Administrator/project_4/project_4.srcs/sim_1/new/sim_shift.v

```
    wire [N-1:0] Q;

    shift_register u1 (
        .Q(Q),
        .D(D),
        .Dsl(Dsl),
        .Dsr(Dsr),
        .S1(S1),
        .S0(S0),
        .CLK(CLK),
        .CR(CR)
    );

    initial begin
        // 初始化输入
        D = 4'b0101;
        Dsl = 0;
        Dsr = 1;
        S1 = 1;
        S0 = 1;
        CLK = 0;
        CR = 0;#10;
        CR=1;#20;
        CR=0;
    end

    always #120 S1=~S1;
    always #14 CLK=~CLK;
    always #60 S0=~S0;

    endmodule
```





一个带有异步清零功能的移位寄存器模块。下面是对程序的详细分析：

module shift_register (S1, S0, D, Dsl, Dsr, Q, CLK,CR);：定义了一个名为 shift_register 的模块，该模块具有输入信号 S1、S0、D、Dsl、Dsr、CLK 和 CR，以及输出信号 Q。

parameter N= 4;：定义了一个参数 N，表示寄存器的位宽，默认为 4。

input S1, S0;：定义了两个输入信号 S1 和 S0，用于选择移位寄存器的操作模式。

input Dsl, Dsr;：定义了两个输入信号 Dsl 和 Dsr，用于输入数据时的移位方向。

input CLK, CR;：定义了两个输入信号 CLK 和 CR，分别表示时钟和异步清零信号。

input [N-1:0] D;：定义了一个输入信号 D，用于输入数据。

output [N-1:0] Q;：定义了一个输出信号 Q，表示寄存器的输出数据。

reg [N-1:0] Q;：定义了一个寄存器的内部变量 Q，用于存储寄存器的数据。

always @ (posedge CLK or posedge CR)：使用 always 块来定义一个时钟触发的逻辑，当时钟上升沿或异步清零信号 CR 上升沿时，执行下面的逻辑。

if (CR)：如果异步清零信号 CR 为 1，则执行下面的逻辑。

Q <= 2'b00;：将寄存器的数据 Q 置为 2'b00，即异步清零。

else：如果异步清零信号 CR 为 0，则执行下面的逻辑。

case ({S1, S0})：根据输入信号 S1 和 S0 的值进行选择。

2'b00: Q <= Q;：当 S1 和 S0 为 2'b00 时，寄存器的数据 Q 保持不变。

2'b01: Q <= {Dsr, Q[N-1:1]};：当 S1 和 S0 为 2'b01 时，将输入信号 Dsr 和寄存器的数据 Q[N-1:1]连接起来，作为新的寄存器数据 Q。

2'b10: Q <= {Q[N-2:0], Dsl};:当 S1 和 S0 为 2'b10 时,将寄存器的数据 Q[N-2:0]和输入信号 Dsl 连接起来，作为新的寄存器数据 Q。

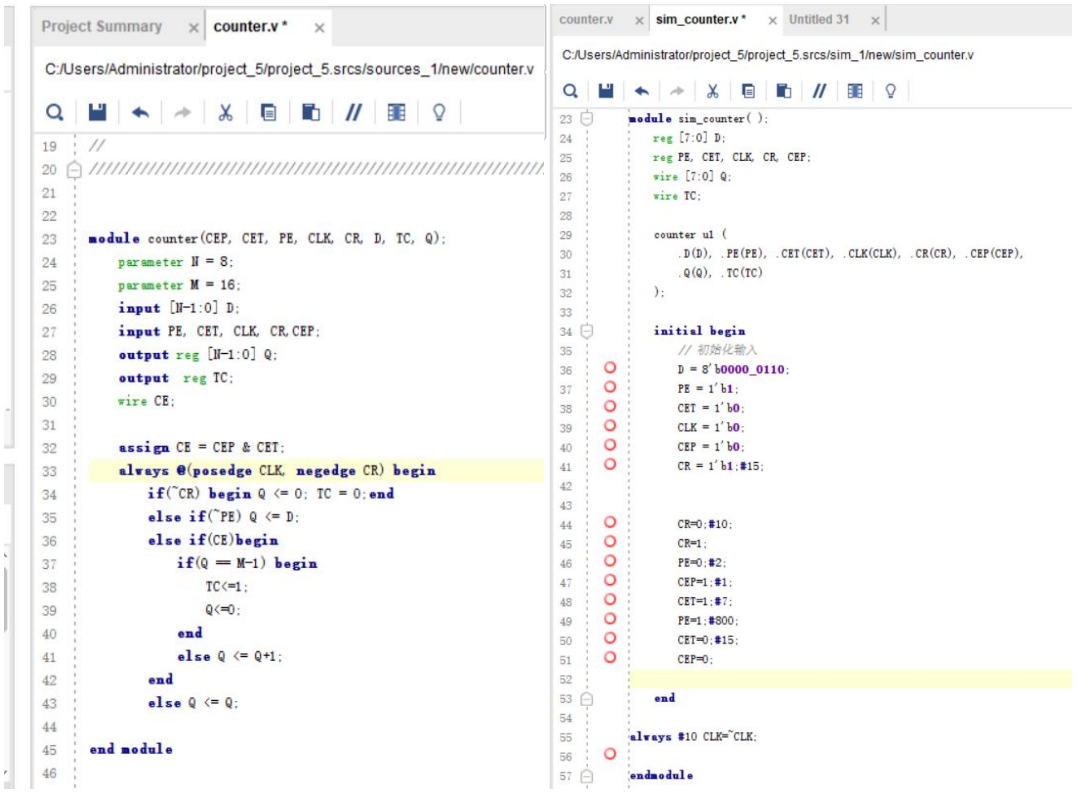
2'b11: Q <= D;：当 S1 和 S0 为 2'b11 时，将输入信号 D 作为新的寄存器数据 Q。

module sim_shift;：定义了一个名为 sim_shift 的模块。

parameter N = 4;：定义了一个参数 N，表示寄存器的位宽，默认为 4。

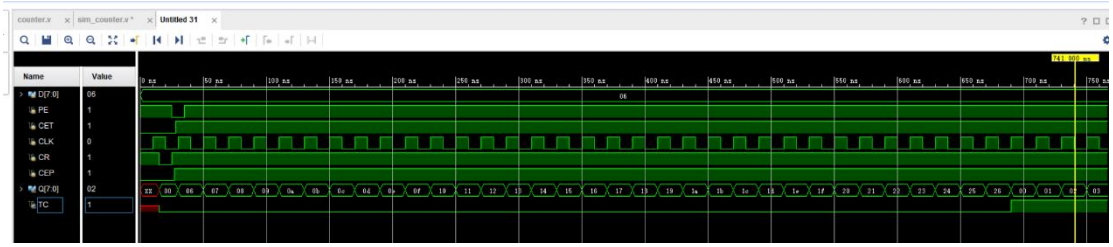
reg [N-1:0] D;：定义了一个寄存器变量 D，用于存储输入数据。
reg Dsl, Dsr, S1, S0, CLK, CR;：定义了一些寄存器变量，用于控制和输入。
wire [N-1:0] Q;：定义了一个连线 Q，用于连接到移位寄存器的输出。
shift_register u1 (.Q(Q), .D(D), .Dsl(Dsl), .Dsr(Dsr), .S1(S1), .S0(S0), .CLK(CLK), .CR(CR));：实例化了一个移位寄存器模块 u1，并将输入和输出信号连接到模块的对应端口。
initial begin：初始化块，用于初始化输入信号。D = 4'b0101;：将输入数据 D 初始化为 4'b0101。Dsl = 0;：将输入信号 Dsl 初始化为 0。Dsr = 1;：将输入信号 Dsr 初始化为 1。S1 = 1;：将输入信号 S1 初始化为 1。S0 = 1;：将输入信号 S0 初始化为 1。CLK = 0;：将输入信号 CLK 初始化为 0。CR = 0; #10;：将输入信号 CR 初始化为 0，并延迟 10 个时间单位。CR=1;#20;：将输入信号 CR 设置为 1，并延迟 20 个时间单位。CR=0;：将输入信号 CR 设置为 0。
always #120 S1=~S1;：每隔 120 个时间单位，取反一次 S1 的值。always #14 CLK=~CLK;：每隔 14 个时间单位，取反一次 CLK 的值。always #60 S0=~S0;：每隔 60 个时间单位，取反一次 S0 的值。
当 CR 信号为 1 时，移位寄存器的值将被异步清零，即 Q 的值被设置为 2'b00。
当 CR 信号为 0 时，根据 S1 和 S0 的值选择不同的操作模式：当 S1 和 S0 都为 0 时，Q 的值保持不变。当 S1 为 0，S0 为 1 时，将 Dsr 的值作为最低位，将 Q 的高位向右移动一位，得到新的 Q 的值。当 S1 为 1，S0 为 0 时，将 Dsl 的值作为最高位，将 Q 的低位向左移动一位，得到新的 Q 的值。当 S1 和 S0 都为 1 时，将 D 的值直接赋给 Q。

计数器：




```
//
//////////////////////////////////////////////////////////////////////////////////


module counter(CEP, CET, PE, CLK, CR, D, TC, Q);
    parameter N = 8;
    parameter M = 16;
    input [N-1:0] D;
    input PE, CET, CLK, CR,CEP;
    output reg [N-1:0] Q;
    output  reg TC;
    wire CE;

    assign CE = CEP & CET;
    always @(posedge CLK, negedge CR) begin
        if(~CR) begin Q <= 0; TC = 0;end
        else if(~PE) Q <= D;
        else if(CE)begin
            if(Q == M-1) begin
                TC<=1;
                Q<=0;
            end
            else Q <= Q+1;
        end
        else Q <= Q;

end module
```

```
module sim_counter( );
    reg [7:0] D;
    reg PE, CET, CLK, CR, CEP;
    wire [7:0] Q;
    wire TC;

    counter u1 (
        .D(D), .PE(PE), .CET(CET), .CLK(CLK), .CR(CR), .CEP(CEP),
        .Q(Q), .TC(TC)
    );

    initial begin
        // 初始化输入
        D = 8'b0000_0110;
        PE = 1'b1;
        CET = 1'b0;
        CLK = 1'b0;
        CEP = 1'b0;
        CR = 1'b1;#15;


        CR=0;#10;
        CR=1;
        PE=0;#2;
        CEP=1;#1;
        CET=1;#7;
        PE=1;#800;
        CET=0;#15;
        CEP=0;

    end

always #10 CLK=~CLK;

endmodule
```



一个 16 进制的 8 位计数器模块，具有异步清零功能和可编程上限值。它有以下输入和输出：
输入：
D：输入信号，用于在 PE 信号为 1 时，将 D 的值赋给计数器的当前值 Q。
PE：计数器使能信号，当 PE 为 1 时，将 D 的值赋给 Q。
CET：计数器计数使能信号，当 CET 和 CEP 同时为 1 时，计数器开始计数。
CEP：计数器计数使能信号，当 CET 和 CEP 同时为 1 时，计数器开始计数。
CLK：时钟信号，用于触发计数器的操作。
CR：异步清零信号，当 CR 为 1 时，计数器的值被清零。
输出：
TC：计数器的进位信号，当计数器从最大值 M-1 增加到 0 时，TC 被置为 1。
Q：计数器的输出信号，即当前计数器的值。
计数器的功能如下：
当 CR 信号为 1 时，计数器的值被清零，即 Q 的值被设置为 0，TC 被置为 0。
当 PE 信号为 1 时，将 D 的值赋给 Q，即计数器的值被设置为 D 的值。
当 CET 和 CEP 同时为 1 时，计数器开始计数。根据时钟信号的上升沿触发计数器的操作。
在计数器开始计数的情况下，当计数器的值 Q 等于 M-1 时，表示计数器已经达到了上限值，此时 TC 被置为 1，Q 被清零。
在计数器开始计数的情况下，当计数器的值 Q 小于 M-1 时，计数器的值 Q 被加 1。
当 PE、CET 和 CEP 都为 0 时，计数器的值 Q 保持不变。

将 M 改为 39 就变成了 39 进制的计数器：

输入信号：D、PE、CET、CLK、CR、CEP
输出信号：Q、TC
使用 counter 模块实例化一个计数器 u1，并将输入输出信号连接到该实例。
初始化块：
设置 D 的值为 8'b0000_0110，即 6 的二进制表示。设置 PE 为 1，表示使能计数器。设置 CET 为 0，表示不使能计数器的计数功能。设置 CLK 为 0，表示时钟信号初始值为 0。设置 CEP 为 0，表示不使能计数器的计数功能。设置 CR 为 1，表示清零计数器。延迟 15 个时间单位。设置 CR 为 0，表示停止清零计数器。延迟 10 个时间单位。设置 CR 为 1，表示再次清零计数器。设置 PE 为 0，表示禁用计数器。延迟 2 个时间单位。设置 CEP 为 1，表示使能计数器的计数功能。延迟 1 个时间单位。设置 CET 为 1，表示使能计数器的计数功能。延迟 7 个时间单位。设置 PE 为 1，表示使能计数器。延迟 800 个时间单位。设置 CET 为 0，表示禁用计数器的计数功能。延迟 15 个时间单位。设置 CEP 为 0，表示禁用计数器的计

数功能。
使用 always #10 CLK=~CLK 的语句，每隔 10 个时间单位，将时钟信号 CLK 取反。

最终可以通过时序图输出可以看到，在 CLK 时钟信号的脉冲下，Q 数组的数字在缓慢加 1，最终加到 16 进制的 26，即是代表了十进制的 38，说明计数器的正确性，39 进制的计数器的实现成功了，计数器的技术范围是十进制的 0-38,16 进制下的 00-26；

状态机：

```
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module FSM_case(input clk,
                input reset,
                output y);
    reg[2:0] state,nextstate;
    always @(posedge clk, posedge reset)
        if(reset) state = 2'b001;
        else state = nextstate;
        always@(posedge clk)
        case(state)
            'b001: nextstate = 'b010;
            'b010: nextstate = 'b100;
            'b100: nextstate = 'b001;
            default: nextstate = 'b001;
        endcase

        assign y = (state =='b001);
endmodule
```

```
module sim_case();
    reg clk;
    reg reset;
    wire y;

    // 实例化FSM_case模块
    FSM_case FSM_case_inst(
        .clk(clk),
        .reset(reset),
        .y(y)
    );
    initial begin
        reset = 0;
        clk=0;
    end
    // 时钟生成
    always #5 clk = ~clk;

endmodule
```

实现了一个有限状态机，根据输入的时钟信号和复位信号，以及当前状态 state，计算出下一个状态 nextstate，并根据当前状态判断输出信号 y 的值。
输入信号：clk、reset
输出信号：y
内部变量：state、nextstate
状态寄存器：

在 always 块中，使用 posedge clk 和 posedge reset 触发条件，根据 reset 信

号的值，将 state 的值初始化为 2'b001 或将 state 的值赋为 nextstate 的值。
状态转移逻辑：
在 always 块中，使用 posedge clk 触发条件，根据当前状态 state，使用 case 语句计算下一个状态 nextstate 的值：
当 state 为 2'b001 时，nextstate 被赋值为 2'b010。
当 state 为 2'b010 时，nextstate 被赋值为 2'b100。
当 state 为 2'b100 时，nextstate 被赋值为 2'b001。
当 state 为其他值时，nextstate 被赋值为 2'b001。
输出逻辑：
使用 assign 语句，根据当前状态 state 判断输出信号 y 的值：
当 state 为 2'b001 时，y 被赋值为 1'b1。
当 state 为其他值时，y 被赋值为 1'b0。

仿真程序实现了对 FSM_case 模块的仿真测试。
输入信号：clk、reset 输出信号：y
实例化 FSM_case 模块，并连接其输入输出信号。
使用 always #5 clk = ~clk 的语句，每隔 5 个时间单位，将时钟信号 clk 取反。
测试序列：
在 initial 块中，初始化 reset 为 0，clk 为 0。

时序分析：
根据状态机代码的逻辑，初始状态 state 为 2'b001，输出信号 y 为 1'b1。然后，每当时钟信号 clk 上升沿到来时，状态机会根据当前状态 state 计算下一个状态 nextstate。根据状态转移逻辑，状态会按照'b001 -> 'b010 -> 'b100 -> 'b001 -> 'b010 -> 'b100 -> ... 的顺序循环变化。
根据仿真文件中的时钟信号生成逻辑，时钟信号 clk 每隔 5 个时间单位取反一次。
因此，我们可以预测输出信号 y 的变化情况如下：
初始状态：state = 2'b001，y = 1'b1 时钟上升沿：state = 2'b010，y = 1'b0 时钟上升沿：state = 2'b100，y = 1'b0 时钟上升沿：state = 2'b001，y = 1'b1 时钟上升沿：state = 2'b010，y = 1'b0 时钟上升沿：state = 2'b100，y = 1'b0 ...

根据状态机的循环特性，输出信号 y 会周期性地在 0 和 1 之间变化。

```
module FSM_cf(input clk,
        input reset,
        input a,
        output y);
    reg[1:0] state, nextstate;
    always @(posedge clk, posedge reset)
        if(reset) state = 2'b00;
    else state = nextstate;
    always@(posedge clk)
    case(state)
        'b00:if(a) nextstate = 'b00;
            else nextstate = 'b01;
        'b01: if(a) nextstate = 'b10;
            else nextstate = 'b01;
        'b10: if(a) nextstate = 'b00;
            else nextstate = 'b01;
        default: nextstate = 'b00;
    endcase

    assign y = (state== 'b10);
endmodule
```

```
module sim_cf();
    reg clk;
    reg reset;
    reg a;
    wire y;

    FSM_cf FSM_cf_inst(
        .clk(clk),
        .reset(reset),
        .a(a),
        .y(y)
    );

    initial begin
        reset = 1; a = 0;
        clk = 0;
        #10 reset =0;
    end

    always #5 clk = ~clk;
    always #10 a=~a;

endmodule
```

它有一个时钟输入 clk，一个复位输入 reset，一个输入信号 a，一个输出信号 y。首先定义了两个寄存器 state 和 nextstate，用于存储当前状态和下一个状态。然后使用 always 块来描述状态转移逻辑。当时钟信号 clk 上升沿到来或复位信号 reset 上升沿到来时，根据 reset 的值来更新 state 的值。如果 reset 为 1，表示复位，将 state 设置为 2'b00；否则，将 state 设置为 nextstate 的值。
接着使用 always 块和 case 语句来描述状态转移逻辑。根据当前状态 state 的值，确定下一个状态 nextstate 的值。具体逻辑如下：
当 state 为 2'b00 时，如果输入信号 a 为 1，则下一个状态为 2'b00；否则，下一个状态为 2'b01。
当 state 为 2'b01 时，如果输入信号 a 为 1，则下一个状态为 2'b10；否则，下一个状态为 2'b01。
当 state 为 2'b10 时，如果输入信号 a 为 1，则下一个状态为 2'b00；否则，下一个状态为 2'b01。
其他情况下，下一个状态为 2'b00。
最后，使用 assign 语句将输出信号 y 赋值为(state == 2'b10)的结果。当且仅当当前状态为 2'b10 时，输出信号 y 为 1；否则，输出信号 y 为 0。
总结来说，以上程序实现了一个带有复位功能的有限状态机。根据输入信号 a 的值，状态在 0、1 和 2 之间循环切换，并且输出信号 y 在状态为 2 时为 1，其他状态为 0。

名为 sim_cf 的模块，首先定义了一个时钟输入 clk，一个复位输入 reset，一个

输入信号 a，一个输出信号 y。然后实例化了一个 FSM_cf 模块，并将 clk、reset、a、y 连接到对应的端口。接着使用两个 always 块来描述时钟和输入信号的行为。
第一个 always 块中，每隔 5 个时间单位，时钟信号 clk 取反一次，即实现了时钟的周期性变化。
第二个 always 块中，每隔 10 个时间单位，输入信号 a 取反一次，即实现了输入信号的周期性变化。
最后，在 initial 块中初始化了 reset 和 a 的值为 0，clk 的值为 0。然后等待 10 个时间单位后，将 reset 的值设为 0。

时序：
时钟信号 clk 每隔 5 个时间单位变化一次，即 0、5、10、15、20...依次循环。
输入信号 a 每隔 10 个时间单位变化一次，即 0、10、20、30...依次循环。
在仿真开始时，reset 和 a 的值都被初始化为 0，clk 的值也为 0。
在 10 个时间单位后，reset 的值被设为 0，状态机开始运行。
根据状态机的状态转移逻辑，当输入信号 a 为 0 时，状态机的状态会在 0 和 1 之间循环切换；当输入信号 a 为 1 时，状态机的状态会在 1 和 2 之间循环切换。
当状态机的状态为 2 时，输出信号 y 的值为 1；否则，输出信号 y 的值为 0。

五、调试和心得体会

首先，我使用 Logisim 实现了 SR 锁存器。SR 锁存器是一种基本的存储器元件，可以通过设置和复位输入来存储一个比特。在 Logisim 中，我通过使用 RS 触发器来实现 SR 锁存器，其中 R 和 S 输入分别对应设置和复位输入。通过这个实验，我更加深入地理解了 SR 锁存器的工作原理和应用场景。
接下来，我使用 Verilog 编写代码实现了 D 锁存器。D 锁存器是一种存储器元件，可以存储一个比特的数据。在 Verilog 中，我可以使用一个 D 触发器来实现 D 锁存器。通过将输入数据传递到 D 触发器的 D 输入，并使用时钟信号来控制触发器的工作，我可以实现 D 锁存器。这个实验让我更加熟悉了 Verilog 语言和 D 锁存器的原理。
然后，我使用 Verilog 实现了 D 触发器。D 触发器是一种存储器元件，可以存储一个比特的数据，并在时钟上升沿触发。通过使用 Verilog 中的 always 块和 posedge 关键字，我可以实现 D 触发器的行为。这个实验加深了我对 D 触发器的理解和 Verilog 语言的应用。
接着，我使用 Verilog 实现了寄存器。寄存器是一种存储多个比特的存储器元件。我可以使用 Verilog 中的 reg 类型和 always 块来实现寄存器。通过在 always 块中根据时钟信号和复位信号的状态来更新寄存器的值，我可以实现寄存器的功能。这个实验让我更加熟悉了寄存器的原理和 Verilog 语言的使用。
然后，我使用 Verilog 实现了移位寄存器。移位寄存器是一种特殊的寄存器，可以将存储的数据进行位移操作。我可以使用 Verilog 中的 shift 操作符和 for 循环来实现移位寄存器。通过这个实验，我更深入地理解了移位寄存器的原理和 Verilog 语言的应用。
接下来，我使用 Verilog 实现了 39 进制计数器。这是一种特殊的计数器，可以在每个时钟周期内按照 39 进制进行计数。为了实现 39 进制计数器，我设计了一

个 6 位的计数器，通过递增和重置操作来实现计数。这个实验让我更加深入地理解了 39 进制计数器的原理和 Verilog 语言的使用。
最后，我使用 Verilog 实现了一个检测二进制序列中的 01 序列的状态机。状态机是一种用于描述系统行为的模型。通过使用 Verilog 中的 always 块和 case 语句，我可以实现一个简单的状态机，用于检测二进制序列中的 01 序列。这个实验加深了我对状态机的理解和 Verilog 语言的应用。
通过这些实验，我不仅加深了对计算机组成原理的理论知识的理解，还学会了如何将这些理论应用到实际操作中。这些实验经验对我今后的学习和研究都将有很大的帮助。我相信，通过这些实验，我可以更加深入地理解计算机硬件的工作原理和实现方法。