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

1 锁存器

1 代码

```
22
23  module D_latch(Q,QN,D,EN,RST);
24      output reg Q, QN;
25      input D;
26      input EN,RST;
27      always @(EN,RST,D) begin
28          if(RST) begin
29              Q= 0;
30              QN= 1;
31          end
32          else if(EN) begin
33              Q<= D;
34              QN<=~D;
35          end
36      end
37  endmodule
```

这个模块描述了一个 D 锁存器，具有一个输入（D）和两个输出（Q 和 QN）。EN（使能信号）和 RST（复位信号）是输入信号。使用 always 块来定义组合逻辑，指定在 EN、RST 或 D 信号发生变化时执行的行为。如果 RST 为真（1），则 Q 被强制为 0，QN 被强制为 1，这是复位操作。如果 EN 为真（1），则 Q 被赋值为 D，QN 被赋值为 D 的反相值，这是使能操作。这里使用的是 <= 运算符，表示非阻塞赋值，可以在同一时间步骤内同时更新多个信号。

2 仿真代码

```
22
23  module sim_latch();
24      wire Q, QN;
25      reg D, EN, RST;
26      integer i;
27      D_latch D_latch1(.Q(Q), .QN(QN), .D(D), .EN(EN), .RST(RST));
28      initial begin
29          D = 0;
30          EN = 0;
31          RST = 0;
32          repeat (32) begin
33              #10 EN = EN + 1;
34              for (i = 0; i < 10; i = i + 1) begin
35                  #10 D = D + 1;
36                  #10 D = D - 1;
37              end
38              #500 RST = RST + 1;
39          end
40      end
41  endmodule
```

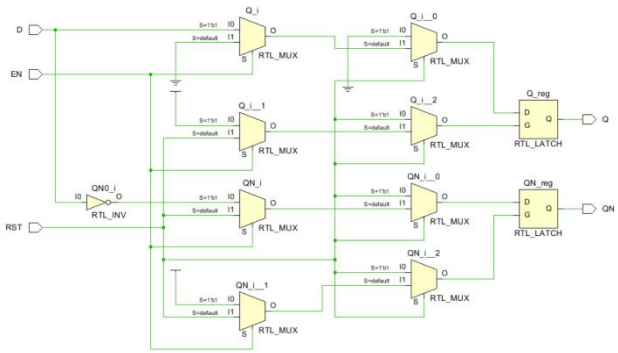
这个模块用于模拟 D_latch 模块的行为。定义了一些信号和一个循环，用于测试 D_latch 模块的行为。创建了一个 D_latch1 实例，将其输入和输出与 sim_latch 模块中的信号相关联。在 initial 块内定义了一系列行为：初始化了 D、EN、RST 为 0。使用 repeat 循环执行以下操作：

每隔 10ns，EN 递增 1。

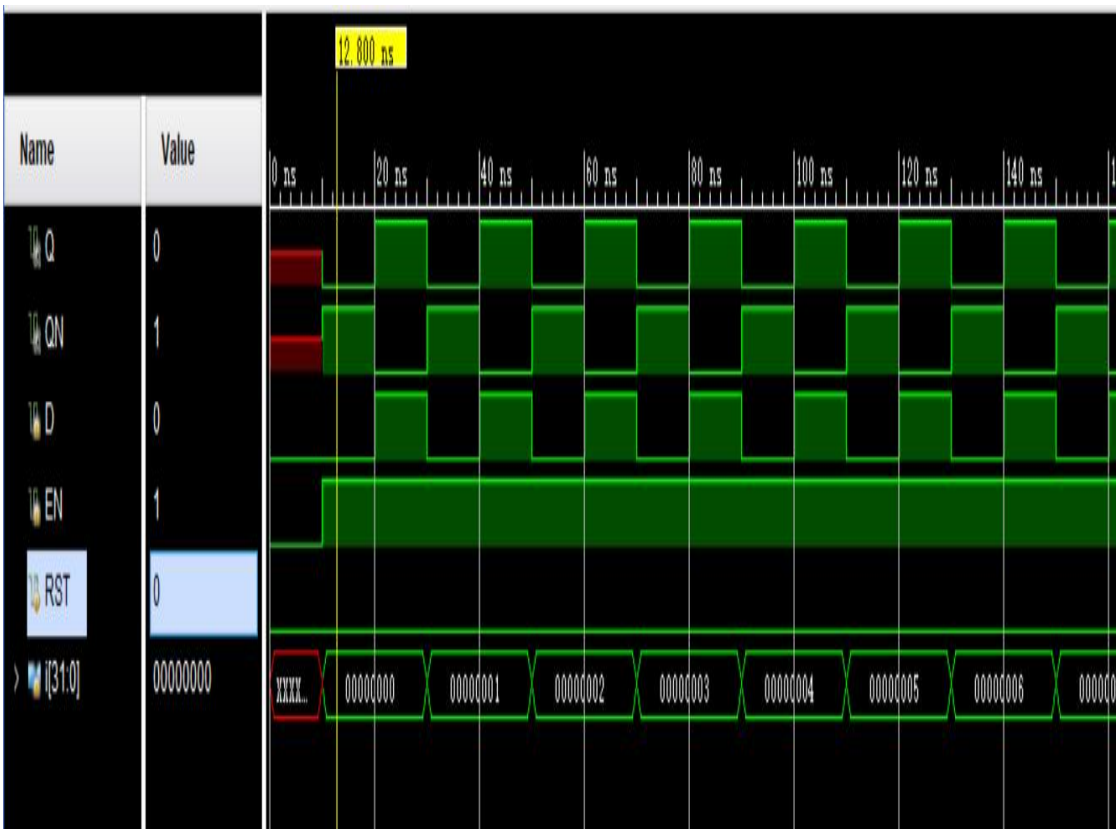
在每个 EN 周期内，使用 for 循环，每隔 10ns，D 递增 1，然后递减 1，模拟 D 信号的变化。

每个 EN 周期结束后，RST 递增 1，模拟 RST 信号的变化。

3 电路图



4 仿真结果



信号分析：

D 是数据输入信号，代表要存储在触发器中的数据值。EN 是使能信号，用于控制是否允许数据传输到 Q 和 QN。RST 是异步清零信号，用于强制 Q 和 QN 的值为 0 和 1。

Q 是触发器的输出信号，将根据 EN 和 RST 的状态来更新。QN 是 Q 的补码，即 Q 的反相信号。

波形分析：

当 RST 为高电平时，无论 EN 的状态如何，触发器都会被强制清零。这意味着 Q 将变为 0，QN 将变为 1，无论 D 的值如何。这是异步清零的作用，清除了触发器中的存储数据。

当 RST 为低电平，EN 为高电平时，触发器处于使能状态。在这种情况下，Q 将采用输入信号 D 的值，QN 将采用 D 的反相值。

当 RST 为低电平，EN 为低电平时，触发器保持其当前状态，不会更新 Q 和 QN 的值。它保持之前的数据。

2 触发器

1 代码

```
22
23  module D_ff(Q, QN,D, EN,RST,CLK);
24      output reg Q, QN;
25      input D;
26      input EN,RST,CLK;
27      always @ (posedge CLK) begin
28          if(RST) begin Q<= 1'b0;QN<=1'b1;end
29          else if(EN) begin Q<= D;QN<=~D; end
30      end
31  endmodule
```

output reg Q, QN; 定义了两个输出端口 Q 和 QN，它们是寄存器的输出信号。input D; 定义了输入端口 D，它是要存储的数据输入。input EN, RST, CLK; 定义了输入端口 EN、RST 和 CLK，

它们分别用于使能、复位和时钟信号。在 always @ (posedge CLK) 块中，使用 always 关键字表示此块内的行为在时钟信号的正边沿触发。在这个块中：

如果 RST 信号为 1，那么 Q 被置为 0，QN 被置为 1。

如果 EN 信号为 1，Q 被设置为输入信号 D，QN 被设置为 D 的反相。

2 仿真代码

```
22
23  module sim_D_ff;
24      reg CLK,D,EN,RST;
25      wire Q,QN;
26      D_ff u1(.Q(Q),.QN(QN),.D(D),.EN(EN),.RST(RST),.CLK(CLK));
27      always #10 CLK <=~CLK;
28      initial begin
29          CLK = 0;
30          #20 D=1;
31          #20 D=0;
32          RST=1;
33          #20 D=1;
34          #20 D=0;
35          RST=0;
36          #20 D=1;
37          EN=1;
38          #20 D=0;
39          #20 D=1;
40          EN=0;
41          #20 D=1;
42          EN=1;
43          #20 D=0;
44          #20 D=0;
45          #20 D=1;
46          EN=0;
47      end
48  endmodule
```

定义了输入信号 CLK, D, EN, 和 RST，以及输出信号 Q 和 QN。使用 D_ff 模块 u1 实例化了一个 D 触发器。u1 的端口连接到模块的信号。在 always #10 CLK <= ~CLK; 语句中，CLK 信号在每个时钟周期后翻转，以模拟时钟信号。在 initial 块中，模拟了一系列输入信号的变化：

初始化 CLK 为 0。

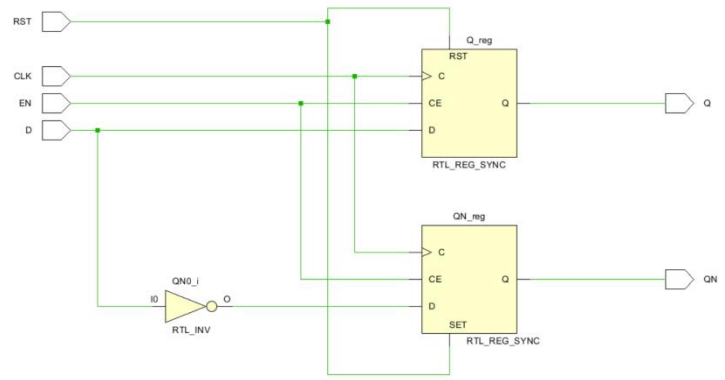
在一系列延迟后，D 信号的值发生了变化，模拟数据输入。

RST 被设置为 1，模拟了复位操作，将触发器的输出置为初始状态。

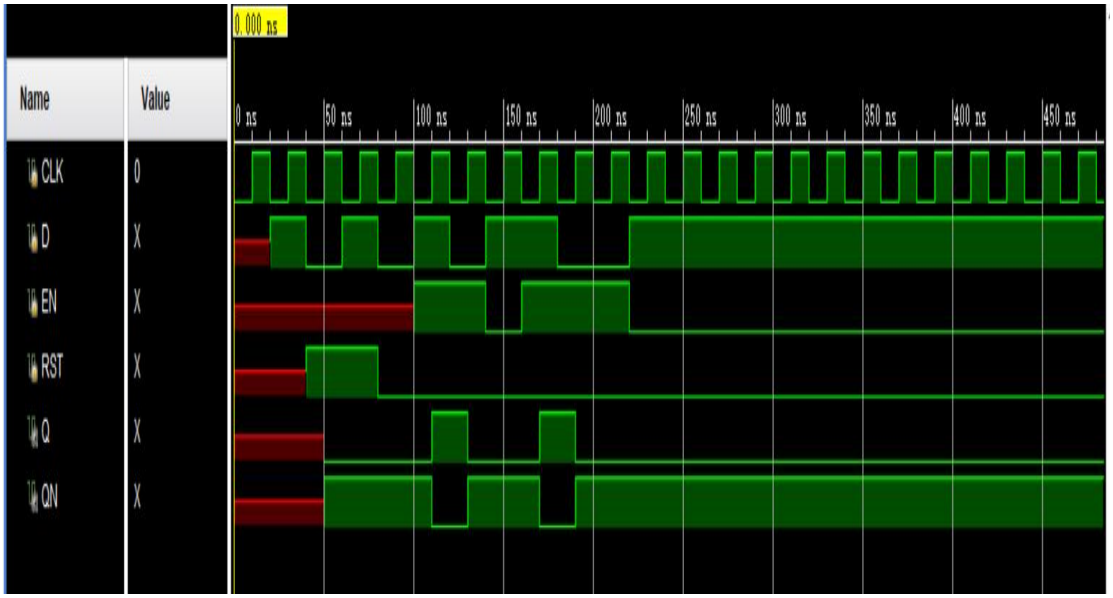
接下来的时钟周期内，RST 被设置为 0，触发器开始响应输入信号。

一系列 D 和 EN 的变化模拟了不同的数据输入和使能操作

3 电路图



4 仿真结果



信号分析：

D 是数据输入信号，代表要存储在触发器中的数据值。EN 是使能信号，用于控制是否允许数据传输到 Q 和 QN。RST 是异步清零信号，用于强制 Q 和 QN 的值为 0 和 1。CLK 是时钟信号，触发器在时钟的上升沿（posedge）时工作。

Q 是是触发器的输出信号，将根据 EN 和 RST 的状态以及时钟信号来更新。QN 是 Q 的补码，即 Q 的反相信号。

波形分析：

当 RST 为高电平时，无论 EN 的状态如何，触发器都会被强制清零。在时钟的下一个上升沿时，Q 将被设置为 0，QN 将被设置为 1，无论 D 的值如何。这是异步清零的作用，清除了触发器中的存储数据。

当 RST 为低电平，EN 为高电平，时钟信号（CLK）的上升沿到来时，触发器处于使能状态。在这种情况下，Q 将采用输入信号 D 的值，QN 将采用 D 的反相值。

当 RST 为低电平，EN 为低电平，时钟信号（CLK）的上升沿到来时，触发器保持其当前状态，不会更新 Q 和 QN 的值。它保持之前的数据。

3 寄存器

1 代码

```
22
23  module register(Q,D,OE,CLK);
24      parameter N=8;
25      output reg[N-1:0] Q;
26      input [N:1] D;
27      input OE,CLK;
28      always @(posedge CLK or posedge OE)
29          if (OE) Q <=8'bzzzz_zzzz;
30          else Q <= D;
31  endmodule
```

代码包括一个输出寄存器 Q，一个输入 D，一个输出使能信号 OE 和一个时钟信号 CLK。它有一个参数 N 设置为 8，并且使用一个 always 块来控制 Q 的赋值。当 OE 为高电平（1）时，Q 被赋值为全高阻态（8'bzzzz_zzzz），否则 Q 被赋值为输入 D。

2 仿真代码

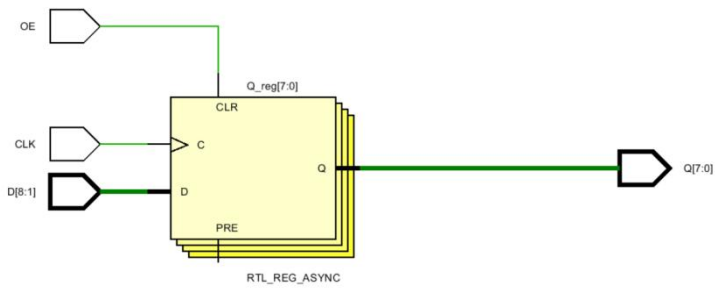
```
23  module sim_register;
24      reg CLK;
25      reg OE;
26      reg [3:0] D_in;
27      wire [3:0] D_out;
28      register reg0(D_out, D_in, OE, CLK);
29      initial CLK = 1'b1;
30      always #10 CLK = ~CLK;
31      initial begin
32          OE = 1'b0;
33          D_in = 4'b1111;
34          #100
35          OE = 1'b1;
36          #100
37          OE = 1'b0;
38          #50
39          D_in = 4'b0001;
40          #100
41          OE = 1'b1;
42          #100
43          D_in = 4'b0010;
44          #100
```

```
43          D_in = 4'b0010;
44          #100
45          D_in = 4'b0011;
46          #100
47          D_in = 4'b0011;
48          #100
49          OE=1'b0;
50          #35
51          D_in=4'b0101;
52      end
53  endmodule
```

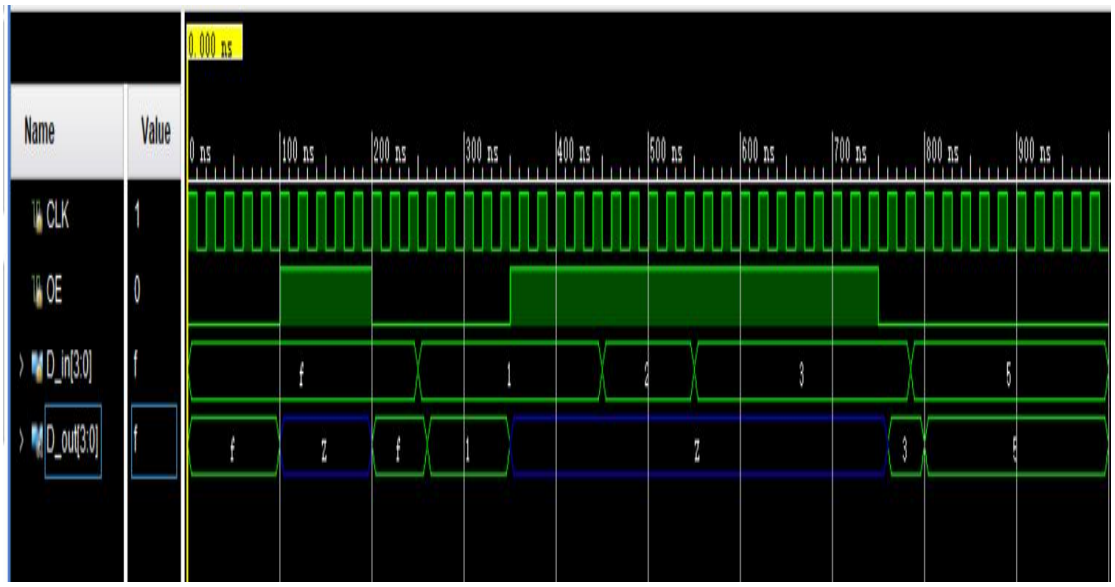
代码包括一个时钟信号 CLK，一个输出使能信号 OE，一个 4 位宽的输入 D_in，以及一个 4 位宽的输出 D_out。在模块内部，它实例化了一个 register 模块命名为 reg0，并将输入输出信号连接到 sim_register 模块的信号。

sim_register 模块在不同的时间点模拟了输出寄存器的输入信号 D_in 和输出使能信号 OE 的变化，以测试输出寄存器的行为。在输出使能信号为高电平时，输出寄存器的输出 D_out 会被设置为全高阻态，而在输出使能信号为低电平时，它会根据输入信号 D_in 的值来改变。同时，时钟信号 CLK 以每 10 个时间单位反转一次。

3 电路图



4 仿真结果



信号分析：

D 是一个 8 位的数据输入信号，代表要存储在寄存器中的数据值。OE 是输出使能信号，用于控制是否允许数据从寄存器输出到 Q。CLK 是时钟信号，寄存器在时钟的上升沿（posedge）时工作。

Q 是 8 位宽的寄存器的输出信号，存储了输入数据 D。Q 是一个寄存器的输出，当 OE 为高电平时，数据将被输出到 Q。当 OE 为低电平时，Q 保持其当前值。

波形分析：

当 OE 为高电平（有效）时，寄存器的输出 Q 会被输入信号 D 所驱动。在时钟信号（CLK）的上升沿到来时，输入信号 D 的值会被复制到 Q。换句话说，Q 将等于 D。这意味着寄存器会在时钟上升沿时根据输入信号 D 更新。

当 OE 为低电平时，寄存器的输出 Q 会保持其当前值，无论时钟信号如何变化。即使在时钟信号的上升沿，数据也不会被更新到 Q。这是因为 OE 信号禁用了寄存器的输出。

4 移位寄存器

1 代码

```
22
23  module shift_register(S1,S0,D,Dsl,Dsr,Q,CLK,CR);
24      parameter N=4;
25      input S1,S0;
26      input Dsl,Dsr;
27      input CLK,CR;
28      input [N-1:0] D;
29      output [N-1:0] Q;
30      reg [N-1:0] Q;
31      always @ (posedge CLK or posedge CR)
32          if(CR)
33              Q<=0;
34          else
35              case({S1,S0})
36                  2'b00:Q<=Q;
37                  2'b01:Q<={Dsr,Q[N-1:1]};
38                  2'b10:Q<={Q[N-2:0],Dsl};
39                  2'b11:Q<=D;
40              endcase
41  endmodule
```

定义了一个参数 N，默认值为 4，用于指定寄存器的位数。

输入信号包括 S1、S0、Dsl、Dsr、CLK 和 CR。

输入 D 是一个 N 位的向量，表示输入数据。

输出信号 Q 也是一个 N 位的向量，表示寄存器的内容。

定义了一个 reg 类型的寄存器 Q，用于存储移位寄存器的内容。

使用 always 块，在时钟信号 CLK 上升沿或复位信号 CR 上升沿触发。

如果 CR 为高电平，将 Q 清零。

否则，根据 S1 和 S0 的值，通过 case 语句执行不同的操作：

当 S1 和 S0 为 2'b00 时，保持 Q 不变。

当 S1 为 1，S0 为 0 时，将 Dsl 放入 Q 的最低位，其余位右移。

当 S1 为 0，S0 为 1 时，将 Dsr 放入 Q 的最高位，其余位左移。

当 S1 和 S0 均为 1 时，将 D 放入 Q。

2 仿真代码

```
21
22  module sim_shift_register;
23      parameter N=4;
24      reg s1,s0;
25      reg dsl,dsr;
26      reg clk,cr;
27      reg [N-1:0] d;
28      wire [N-1:0] q;
29      shift_register reg0( .S1(s1), .S0(s0), .D(d),.Dsl(dsl), .Dsr(dsr), .Q(q),  .CLK(clk), .CR(cr) );
30      initial clk =1'b1;
31      always #10 clk=~clk;
32      initial begin
33          dsl=0; dsr=1;cr=0;
34          s1=0;s0=0;
35          #100
36          cr=1;
37          d=4'b1100;
38          #100
39          s1=0;s0=1;
40          #100
41          s1=1;s0=0;
42          #100
43          s1=1;s0=1;
44          #100
45          s1=0;s0=0;
46          #50 cr=0;
47          #100
47          #100
48          cr=1;
49          d=4'b0011;
50          #100
51          s1=0;s0=1;
52          #100
53          s1=1;s0=0;
54          #100
55          s1=1;s0=1;
56          #100
57          s1=0;s0=0;
58          #100
59          $stop;
60      end
61  endmodule
```

定义了一个参数 N，默认值也为 4，以与 shift_register 保持一致。

定义了一些 reg 类型的信号，包括控制信号 s1、s0、dsl、dsr、clk、cr 以及 N 位的输入数据 d 和输出数据 q。

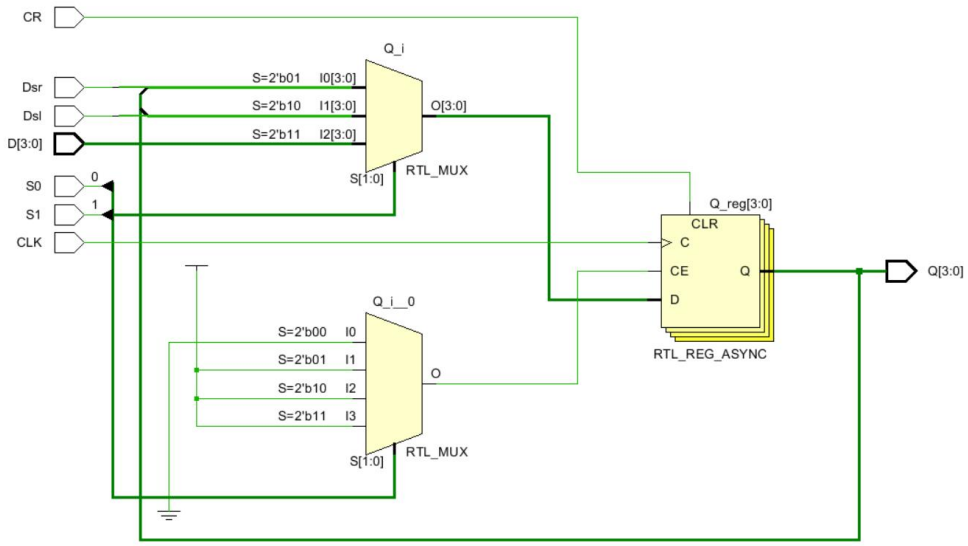
实例化了一个 shift_register 模块 reg0，并连接了输入和输出信号。

使用 initial 块初始化一些信号，并定义了时钟信号 clk 以及在每 10 个时间单位反转一次时钟信号的行为。

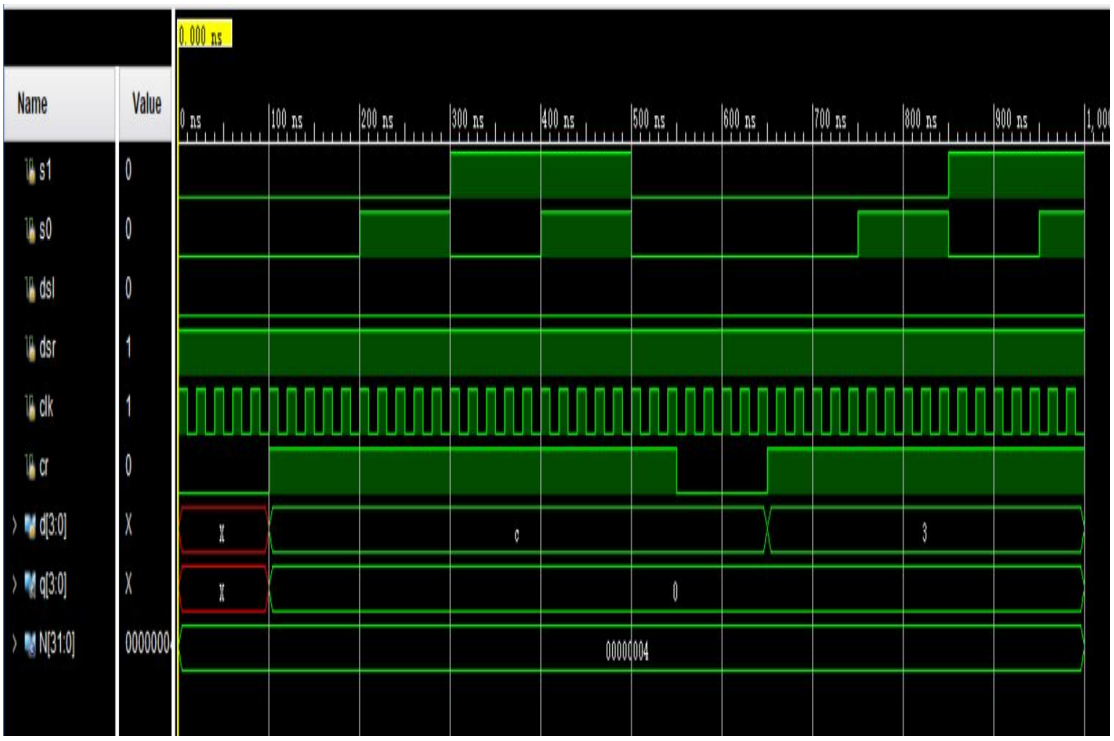
使用 initial 块进行寄存器测试操作。这些操作包括设置 dsl、dsr、cr，然后根据 s1 和 s0 的值以及 d 的不同值来测试移位寄存器的行为。

最后，通过 $stop 终止仿真。

3 电路图



4 仿真结果



信号分析：

S1,、S0 是控制信号，用于指定移位操作的方式。它们共同形成一个 2 位的选择器，决定了移位方向。Dsl、Dsr 是移位寄存器的左移位和右移位输入数据。CLK 是时钟信号，寄存器在时钟的上升沿（posedge）时工作。CR 是清零信号，当 CR 为高电平（posedge）时，寄存器的内容将被清零。D 是输入数据，要加载到寄存器中的数据。

Q 是 N 位宽的输出信号，代表移位寄存器的内容。

波形分析：

当 CR 为高电平时（清零操作），寄存器的内容 Q 会被清零为 0，无论 S1、S0 的状态如何。这是异步清零的作用。

当 CR 为低电平时，根据 S1 和 S0 的状态，寄存器的内容 Q 会发生不同的移位操作：

当 S1 和 S0 都为 00 时，寄存器保持不变，Q 保持不变。

当 S1 为 0，S0 为 1 时，寄存器向右移动，将 Dsr 的值移动到 Q 的最低位，同时将 Q 的其他位向右移动。

当 S1 为 1，S0 为 0 时，寄存器向左移动，将 Dsl 的值移动到 Q 的最高位，同时将 Q 的其他位向左移动。

当 S1 和 S0 都为 11 时，寄存器的内容将被加载为 D 的值，即 D 将替代 Q 的内容。

5 24 进制计数器(本人学号 2211410824，后两位为 24)

1 代码

```
22
23  module counter_24(CEP,CET,PE,CLK,CR,D,TC,Q);
24      parameter N=8;
25      parameter M=24;
26      input CEP,CET,PE,CLK,CR;
27      input [N-1:0] D;
28      output reg TC;
29      output reg [N-1:0] Q;
30
31      wire CE;
32      assign CE=CEP&CET;
33      always @(posedge CLK,negedge CR)
34          if(~CR) begin Q<=0;TC=0;end
35      else if(~PE) Q <=D;
36      else if(CE) begin
37          if(Q==M-1)begin
38              TC <=1;
39              Q<=0;
40          end
41          else    Q<=Q+1;
42      end
43      else Q<=Q;
44  endmodule
```

有以下输入和输出信号：

输入信号：CEP, CET, PE, CLK, CR, D

输出信号：TC, Q

CEP：计数器允许使能信号，当它为 1 时，计数器工作。

CET：计数器清零信号，当它为 1 时，计数器清零。

PE：计数器预置使能信号，当它为 0 时，计数器使用输入信号 D 的值来预置。

CLK：时钟信号，用于触发计数器的操作。

CR：复位信号，当它下降沿时，计数器复位。

D：输入信号，用于预置计数器的值。

TC：计数器满位输出，当计数器达到 M-1 时（M=24），TC 被设置为 1。

Q：计数器的输出值，一个 24 位的二进制计数值。

如果复位信号 CR 下降沿，则 Q 清零并将 TC 置为 0。

如果使能信号 PE 为 0，则使用输入信号 D 来预置计数器的值。

如果使能信号 CE 为 1（CE=CEP&CET），则计数器工作。如果计数器的值等于 M-1（M=24），则将 TC 置为 1，否则递增 Q 的值。

2 仿真代码

```
22
23  module sim_counter;
24      parameter N = 8;
25      parameter M = 30;
26      reg CEP, CET, PE, CLK, CR;
27      reg [N-1:0] D;
28      wire TC;
29      wire [N-1:0] Q;
30      counter_24 CNT_CASE (.CEP(CEP),.CET(CET),.PE(PE),.CLK(CLK),.CR(CR),.D(D),.TC(TC),.Q(Q));
31      initial begin
32          CLK = 1'b0;
33          CEP = 1;
34          CET = 1;
35          PE = 1;
36          CR = 1;
37          D = 8'b00000000;
38          #100;
39          CR = 0;
40          D = 8'b00000000;
41          #10;
42          CR = 1;
43          PE = 0;
44          CEP = 1;
45          CET = 1;
```

```
45          CET = 1;
46          D = 8'b00000000;
47          #100;
48          PE = 1;
49          D = 8'b00000000;
50          #100;
51          CR = 1;
52          #100;
53          $finish;
54      end
55      always begin
56          #5 CLK = ~CLK;
57      end
58  endmodule
```

在 initial 块中，对输入信号进行了初始化，然后进行了一系列的操作：

首先，时钟信号 CLK 被初始化为 0。

然后，使能信号 CEP，CET，PE，CR 被初始化。

输入信号 D 被初始化为全零。

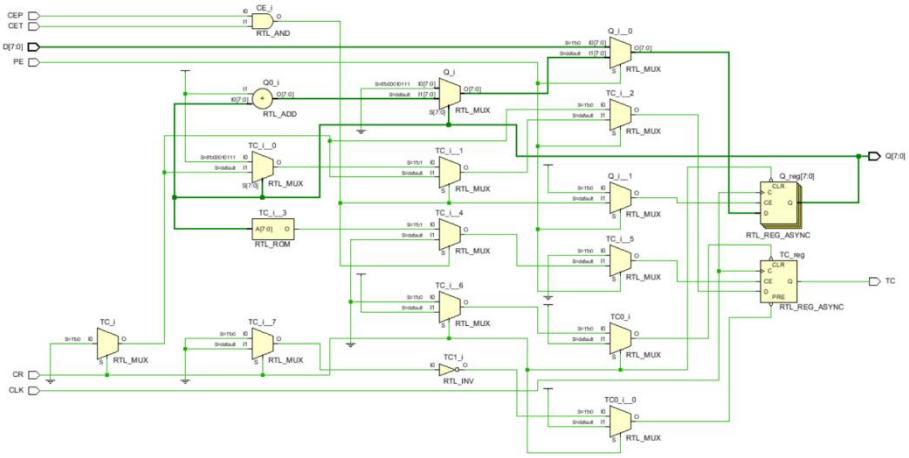
等待 100 个时间单位后，CR 被拉低，这会触发 counter_24 模块的复位逻辑。

再等待 10 个时间单位后，CR 被拉高，PE 被拉低，CEP，CET，D 被重新赋值，这将触发计数逻辑开始工作。

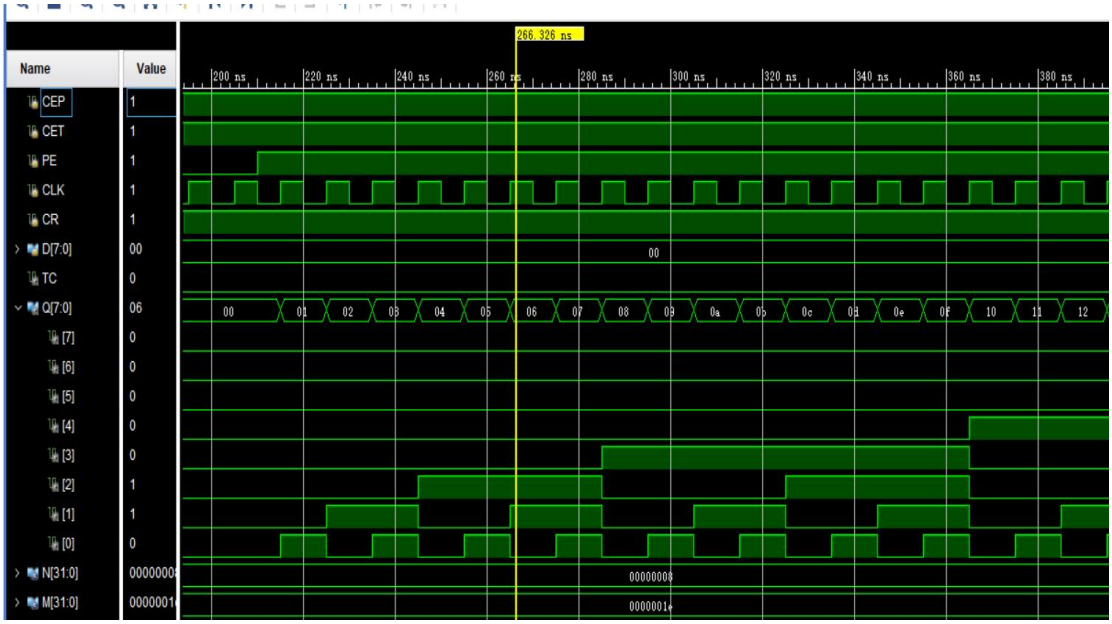
后续操作模拟了计数器的工作，包括清零、预置、计数等操作，最后在 100 个时间单位后结束仿真。

最后，在 always 块中，时钟信号 CLK 每 5 个时间单位取反，以模拟时钟的周期性。这确保了计数器在时钟信号的驱动下进行操作。

3 电路图



4 仿真结果



信号分析：

CEP 是计数使能信号（Count Enable Positive）。当 CEP 为高电平时，计数器开始计数。CET 是计数使能负边缘触发信号（Count Enable Toggle）。当 CET 在下降沿时，计数器的状态会切换，即开始/停止计数。PE 是并行加载使能信号（Parallel Enable）。当 PE 为低电平时，输入数据 D 会并行加载到计数器。CLK 是时钟信号，计数器在时钟的上升沿（posedge）时工作。CR 是清零信号，当 CR 在下降沿时，计数器的内容将被清零。D 是输入数据，用于并行加载到计数器。

TC：计数器溢出标志。当计数器达到 24（M-1）时，TC 将置位为 1，表示计数溢出。Q：这是一个 8 位宽的计数器的输出信号，代表计数器的当前计数值。

波形分析：

当 CR 在下降沿时，无论 CEP 和 CET 的状态如何，计数器的内容 Q 和溢出标志 TC 都会被清零，即 Q 和 TC 都被重置为 0。

如果 PE 为低电平，表示要并行加载数据，那么输入数据 D 将会被加载到计数器，即 Q 的值将等于 D。

如果 PE 为高电平，且 CE 为高电平时，计数器开始计数。当 Q 等于 M-1（24-1=23）时，表示计数器达到了最大值，此时 TC 被置位为 1，表示溢出。Q 将被重置为 0。否则，Q 会递增。

如果 PE 为高电平，但 CE 为低电平，计数器不会计数，Q 的值保持不变。

6 状态机(3 分频计数器)

1 代码

```
22
23  module FSM_case(input clk,input reset,output y);
24      reg[2:0] state, nextstate;
25      //state register
26      always @(posedge clk, posedge reset)
27      if(reset) state = 2'b001;
28      else state = nextstate ;//next state logic
29      always@(posedge clk)
30      case(state)
31          'b001 : nextstate = 'b010;
32          'b010: nextstate = 'b100;
33          'b100: nextstate = 'b001 ;
34          default: nextstate = 'b001;
35      endcase
36      //output state
37      assign y = (state== 'b001 );
38  endmodule
```

输入端口：clk 是时钟信号，reset 是复位信号。

输出端口：y 是一个输出信号。

寄存器：state 和 nextstate 都是 3 位寄存器，用于存储状态信息。

always 块 1：

在上升沿时钟 (posedge clk) 或复位信号上升沿 (posedge reset) 时，根据复位信号，更新当前状态 state。

always 块 2：

在时钟上升沿 (posedge clk) 时，根据当前状态 state 执行不同的操作。

如果当前状态是 3'b001，下一个状态是 3'b010。

如果当前状态是 3'b010，下一个状态是 3'b100。

如果当前状态是 3'b100，下一个状态是 3'b001。

否则，使用默认状态 nextstate = 3'b001。

assign 语句：

将输出信号 y 设置为当前状态是否等于 3'b001，如果相等，y 为 1，否则为 0。

2 仿真代码

```
22
23  module sim_FSM_case;
24      parameter N = 8;
25      reg clk;
26      reg reset;
27      wire y;
28      FSM_case CNT (.clk(clk),.reset(reset),.y(y));
29      initial begin
30          clk = 0;
31          reset = 0;
32          #10 reset = 1;
33          #20 reset = 0;
34          forever #5 clk = ~clk;
35          #200;
36          $finish;
37      end
38  endmodule
```

定义了一个名为 CNT 的 FSM_case 实例，将输入输出端口连接到此模块的端口。

定义了一个时钟信号 clk、一个复位信号 reset，以及一个输出信号 y。

在 initial 块中：

初始化 clk 和 reset 信号。

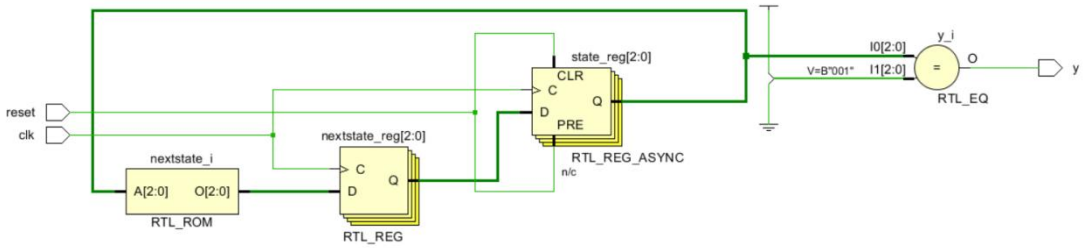
在模拟开始后的 10 个时间单位后，将 reset 信号置为 1，表示复位。

然后在 20 个时间单位后，将 reset 信号恢复为 0。

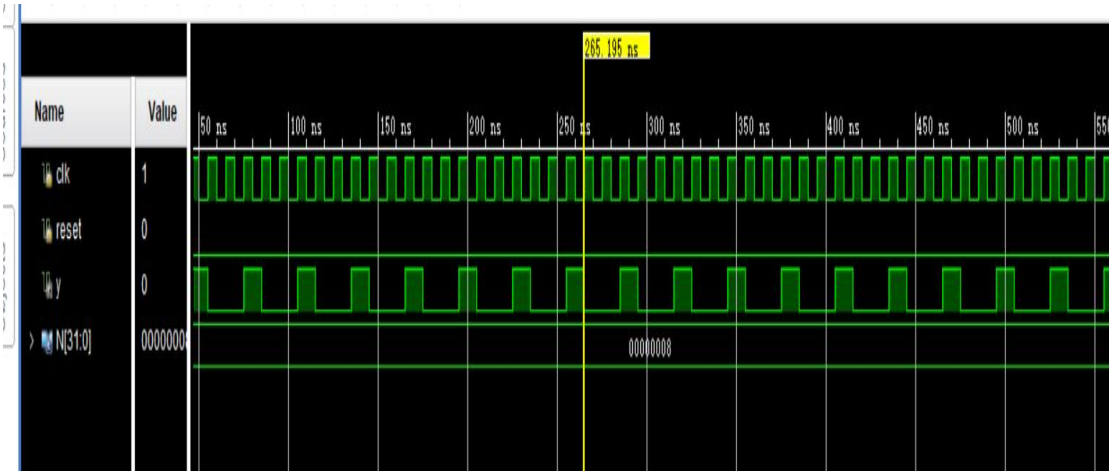
使用 forever 循环生成一个时钟脉冲，每 5 个时间单位切换一次时钟信号的值，模拟时钟的工作。

最后，在模拟运行了 200 个时间单位后，调用 $finish 终止模拟

3 电路图



4 仿真结果



信号分析：

clk 是时钟信号，状态机在时钟的上升沿（posedge）时工作。

reset 是重置信号，当 reset 为高电平（posedge）时，状态机的状态将被重置为初始状态。

y 是一个输出信号，表示状态机的当前状态是否为 'b001。如果当前状态为 'b001，y 将被置为高电平，否则为低电平。

波形分析：

在状态机的 always 块中，当 reset 为高电平时，状态机的状态 state 被重置为 'b001。这是异步重置操作，以确保状态机在初始状态开始。

在状态机的第二个 always 块中，当时钟信号 clk 的上升沿到来时，状态机的状态会根据 case 语句中的逻辑更新。根据当前状态 state 的值，会根据 case 语句中的逻辑来确定下一个状态 nextstate。具体来说：

当当前状态为 'b001 时，下一个状态为 'b010。

当当前状态为 'b010 时，下一个状态为 'b100。

当当前状态为 'b100 时，下一个状态为 'b001。

在其他情况下，下一个状态被设置为 'b001。

最后，通过 assign 语句，输出信号 y 被设置为当前状态是否等于 'b001。如果当前状态是

'b001，y 被置为高电平，表示状态机处于特定状态；否则 y 为低电平。

7 状态机(检测到 01 后输出 1)

1 代码

```
22
23  module FSM_cf(input clk,input reset,input a,output y);
24      reg[1:0] state,nextstate ;
25      //state register
26      always @(posedge clk, posedge reset)
27      if(reset) state = 2'b00;
28      else state = nextstate ;
29      //next stste logic
30      always@(posedge clk)case(state)
31          'b00: if(a) nextstate = 'b00;
32          else nextstate = 'b01;
33          'b01: if(a) nextstate = 'b10;
34          else nextstate = 'b01;
35          'b10: if(a) nextstate = 'b00;
36          else nextstate = 'b01 ;
37          default: nextstate = 'b00;
38      endcase
39      //output state
40      assign y = (state=='b10);
41  endmodule
```

代码有三个输入信号（clk、reset、a）和一个输出信号（y）。

reg[1:0] state, nextstate;

这里声明了两个寄存器，用于存储当前状态（state）和下一个状态（nextstate）的值。每个寄存器是 2 位宽度的。

always @(posedge clk, posedge reset)

这是一个组合块，它在时钟上升沿（posedge clk）和复位信号上升沿（posedge reset）时执行。根据 reset 的状态，它将当前状态(state)设置为 'b00 或者将其设置为下一个状态(nextstate)的值。

always @(posedge clk)

这也是一个组合块，它在时钟上升沿（posedge clk）时执行。它根据当前状态（state）的值，使用 case 语句来确定下一个状态（nextstate）的值。

assign y = (state == 'b10);

这里将输出 y 绑定到 FSM 的状态，如果当前状态为 'b10，则 y 为 1，否则为 0。

2 仿真代码

```
22
23  module sim_FSM_cf();
24      reg clk,reset,a;
25      wire[1:0] y;
26      initial begin
27          clk=0;
28          reset=0;
29          a=1;
30          fork
31
32          begin
33              repeat(30)begin
34              #10clk=1;
35              #10clk=0;
36          end
37      end
38
39      begin
40          repeat(30)begin
41          #50a=0;
42          #50a=1;
43      end
44    end
45    join
46  end
47   FSM_cf  FSM_cf1(.clk(clk),.reset(reset),.a(a),.y(y));
48  endmodule
```

reg clk, reset, a;声明了三个寄存器，分别用于模拟时钟信号（clk）、复位信号（reset）、和输入信号（a）。

wire[1:0] y;定义一个 wire，用于连接到 FSM 模块的输出信号 y。

在 initial begin 块中，设置了 clk、reset、和 a 的初值，然后通过 fork 创建两个并行执行的块。第一个块通过 repeat 循环产生一个时钟信号（clk）的脉冲，每个脉冲持续 10 个时间单位。

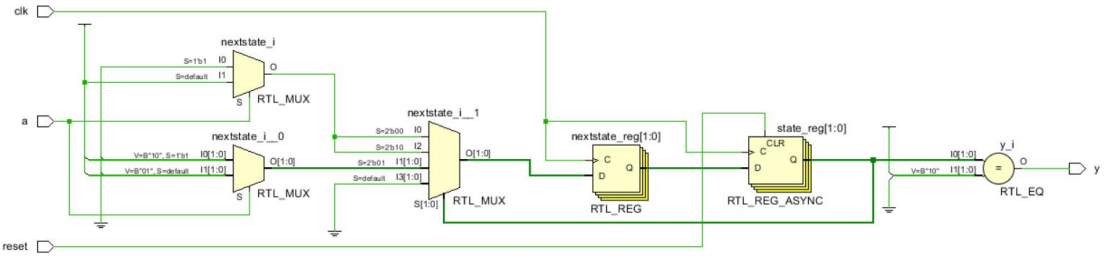
第二个块也使用 repeat 循环来产生输入信号 a 的变化，每个状态保持 50 个时间单位。

最后，使用 join 来等待所有并行执行的块完成。

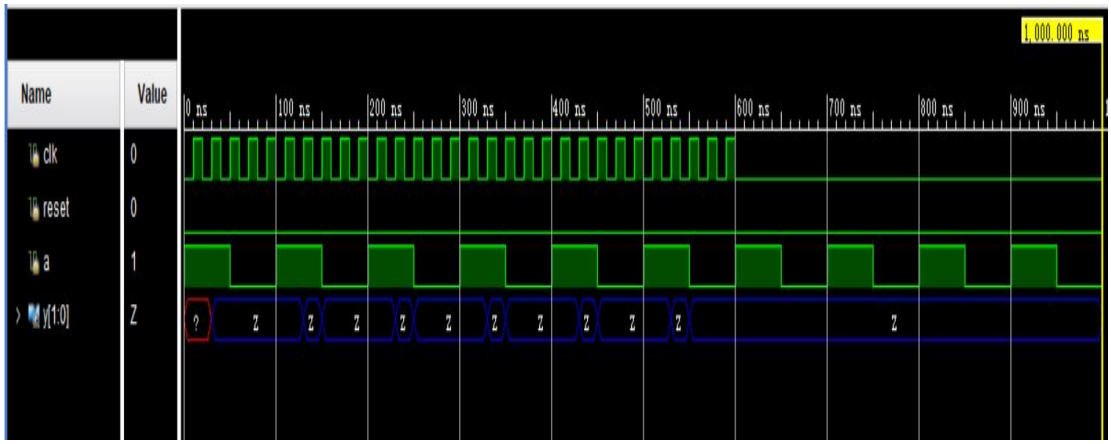
FSM_cf FSM_cf1(.clk(clk), .reset(reset), .a(a), .y(y));

实例化了 FSM 模块，连接了输入信号 clk、reset、a，以及输出信号 y

3 电路图



4 仿真结果



信号分析：

clk 是时钟信号，状态机在时钟的上升沿（posedge）时工作。reset 是重置信号，当 reset 为高电平（posedge）时，状态机的状态将被重置为初始状态。a 是输入信号，用于状态迁移逻辑。

y：这是一个输出信号，表示状态机的当前状态是否为 'b10。如果当前状态为 'b10，y 将被置为高电平，表示状态机处于特定状态；否则为低电平。

波形分析：

在状态机的第一个 always 块中，当 reset 为高电平时，状态机的状态 state 被重置为 'b00。这是异步重置操作，以确保状态机在初始状态开始。

在状态机的第二个 always 块中，当时钟信号 clk 的上升沿到来时，状态机的状态会根据 case 语句中的逻辑更新。根据当前状态 state 的值，会根据 case 语句中的逻辑来确定下一个状态 nextstate。具体来说：

当当前状态为 'b00 时，如果输入信号 a 为高电平，下一个状态仍为 'b00，否则下一个状态为 'b01。

当当前状态为 'b01 时，如果输入信号 a 为高电平，下一个状态为 'b10，否则下一个状态为 'b01。

当当前状态为 'b10 时，如果输入信号 a 为高电平，下一个状态为 'b00，否则下一个状态为 'b01。

在其他情况下，下一个状态被设置为 'b00。

最后，通过 assign 语句，输出信号 y 被设置为当前状态是否等于 'b10。如果当前状态是 'b10，y 被置为高电平，表示状态机处于特定状态；否则 y 为低电平。

五、调试和心得体会

本次实验的主要目的是利用 case 语句、 always 语句等完成触发器、锁存器等功能，掌握用 always、 assign、 case 等语句的详细用法。

用 Verilog 实现数字电路中所学的触发器、锁存器、寄存器等基本模块，通过这次实验，我重温并再次复习了这些电路的结构以及功能是如何实现的，并且使用 Verilog HDL 实现了这些功能。 通过这次实验，我基本上熟练掌握了 Verilog 的编程方法以及调试、 仿真的基本方法。