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.SR锁存器

分析文件：

module SR\_latch(

input S,R,

output reg Q,QN

);

always@(S,R)

case({S, R})

2'b00: begin Q<=Q;QN<=QN; end

2'b01: begin Q<=0;QN<=1;end

2'b10:begin Q<=1;QN<=0;end

default: begin Q<=0;QN<=0;end

endcase

endmodule

仿真文件：

module sim;

reg S,R;

wire Q,QN;

SR\_latch s(S,R,Q,QN);

initial begin

S=0;R=0;

end

always begin

#10 case({S,R})

2'b00:{S,R}={S,R}+1'b1;

2'b01:{S,R}={S,R}+1'b1;

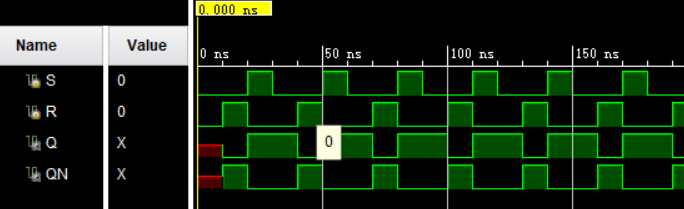
default:{S,R}=2'b00;

endcase

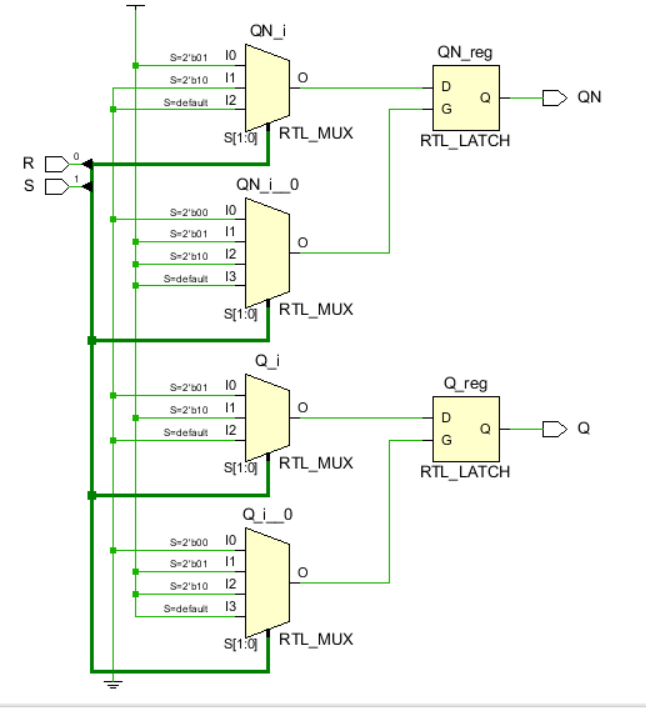
end

endmodule

仿真图形：



电路图：



分析：

整个代码使用case语句，以{S,R}为变量，当S和R全为0时，Q和QN都保持不变；当S为1，R为0的时候，将Q置1，QN置0；当S为0，R为1的时候，将Q置0，R置1；当S和R都是0的时候，将Q和QN同时置0，表示此时系统状态错误。

仿真图形中在10ns到20ns的过程中此时S为0，R为1，对应的Q就被置0，QN就被置1；在20ns到30ns的过程中此时S为1，R为0，对应的Q就被置1，QN就被置0；在30ns到40ns的过程中此时S为0，R为0，对应的Q和QN的值都保持不变，仿真图形全部符合预期

2.D锁存器

设计文件：

module D\_latch(

input D,CLK,

output Q,QN

);

SR\_latch s(CLK&D,CLK&~D,Q,QN);

Endmodule

仿真文件：

module sim;

reg D,CLK;

wire Q,QN;

D\_latch s(D,CLK,Q,QN);

initial begin

D=0;CLK=0;

end

always fork

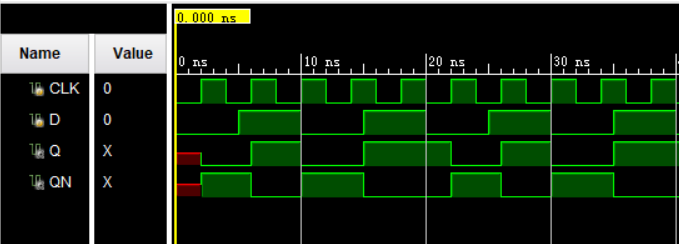
forever #2 CLK=CLK+1;

forever #5 D=D+1;

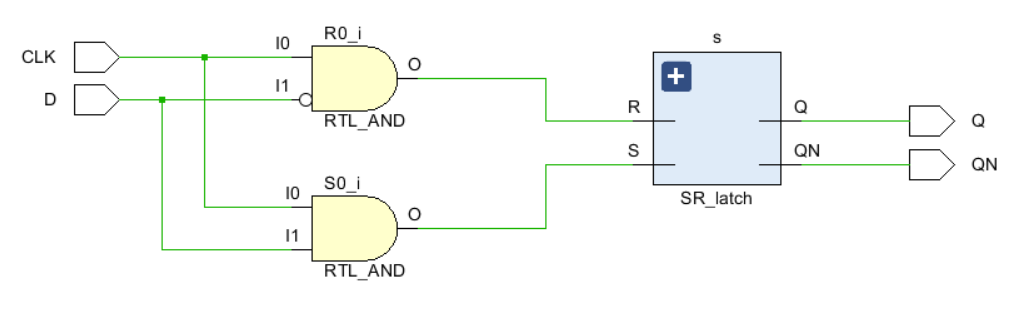
join

endmodule

仿真图形：



电路图：



分析：

D锁存器只有在CLK为1的时候才能发生改变，在实现过程中可以引用SR锁存器模块，将CLK&D作为S，将CLK&~D作为R，这样的话首先S和R不会同时为1，当CLK为0的时候，S和R都为0，Q和QN都保持不变，符合“只有在CLK为1的时候才能发生改变“这一特性;当CLK为1时候，若D为1，对应S为1，R为0，结果Q为1，QN也为0；若D为1，对应S为0，R为1，结果Q为0，QN为1，符合Q=D的预期要求。

在仿真图形中我们也能看到，当CLK的值为1的时候，Q的值永远和D的值相同，当CLK的值为0的时候，Q的值一直保持不变，这样的结果符合预期。

3.锁存器

设计文件：

module latch(

output reg Q,QN,

input D,

input 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

仿真文件：

module sim;

reg D,EN,RST;

wire Q,QN;

latch s(Q,QN,D,EN,RST);

initial begin

D=0;EN=1;RST=0;

end

always fork

forever #2 D=~D;

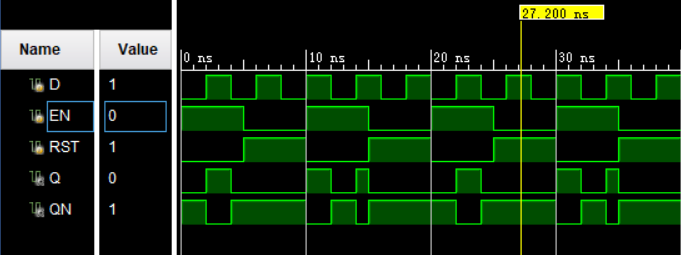
forever #5 EN=~EN;

forever #5 RST=~RST;

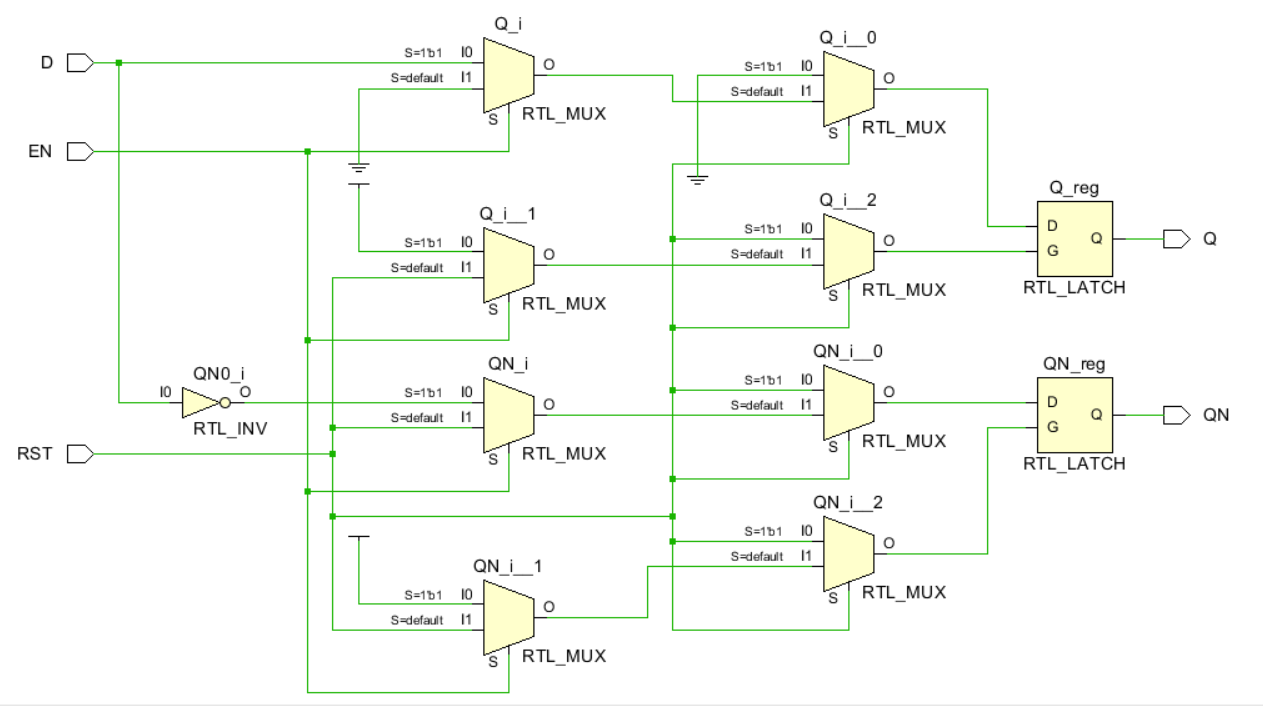
join

endmodule

仿真图象：



电路图：



分析：

带有复位功能和使能端的D锁存器，当EN，RST,D三者中有任意一个发生变化的时候就会进入always语句，在语句中首先对RST的值进行判断，如果RST有效，则直接将输出Q置0，QN置1；如果RST无效再来判断EN，如果EN有效，则将D的值赋给Q，~D的值赋给QN,若EN无效则保持不变。

在仿真图形中，在5ns到10ns的过程中RST为1，Q直接置0，QN直接置1；在10ns到15ns的过程中RST的值为0，EN为1，Q的值与D完全保持一致。符合预期结果

4.D触发器

设计文件：

module D\_Flip(

input D,CLK,

output Q,QN

);

wire Q1,Q1N;

D\_latch s(D,~CLK,Q1,Q1N),

s1(Q1,CLK,Q,QN);

endmodule

仿真文件：

module sim;

reg D,CLK;

wire Q,QN;

D\_Flip s(D,CLK,Q,QN);

initial begin

D=0;CLK=0;

end

always fork

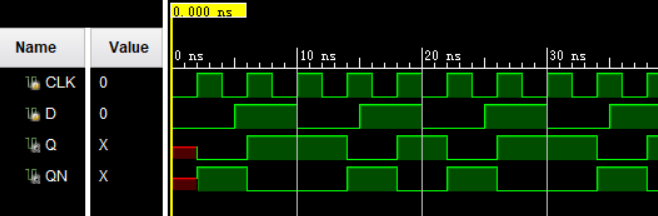
forever #2 CLK=CLK+1;

forever #5 D=D+1;

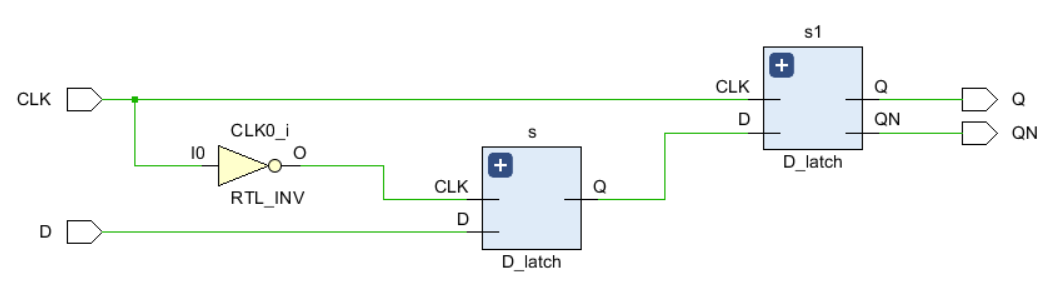
join

endmodule

仿真图象：



电路图：



分析：

该D触发器由两个D锁存器组成，当D为0时，s中的Q变为了D的值，但是s1中的Q保持不变，之后当D出于上升沿变为1的时候，s中的Q不再改变，s1中Q的值变为了s中Q的值，进而达到了“只有当CLK处于上升沿的时候，Q的值才会被改为D的值“这一特性。

仿真图形中，在2ns的时候CLK处于上升沿，Q的值变成了此时D的值0，然后在2ns到6ns的过程中，CLK都没有处于上升沿状态，尽管D的值发生了变化，但是Q的值在这段过程中始终保持不变。

5.带使能的触发器

设计文件：

module en\_flip(

input D,CLK,EN,

output Q,QN

);

D\_Flip s(D,CLK&EN,Q,QN);

Endmodule

仿真文件：

module sim;

reg D,CLK,EN;

wire Q,QN;

en\_flip s(D,CLK,EN,Q,QN);

initial begin

D=0;CLK=0;EN=0;

end

always fork

forever #2 CLK=CLK+1;

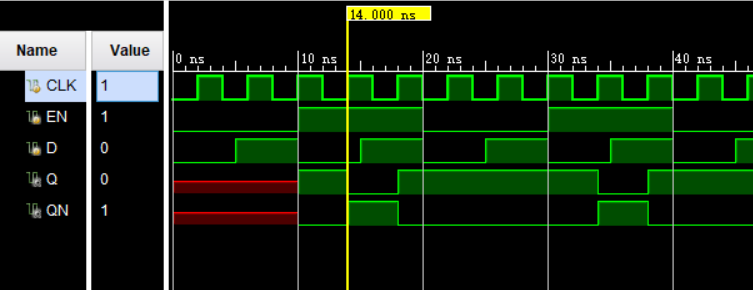
forever #5 D=D+1;

forever #10 EN=EN+1;

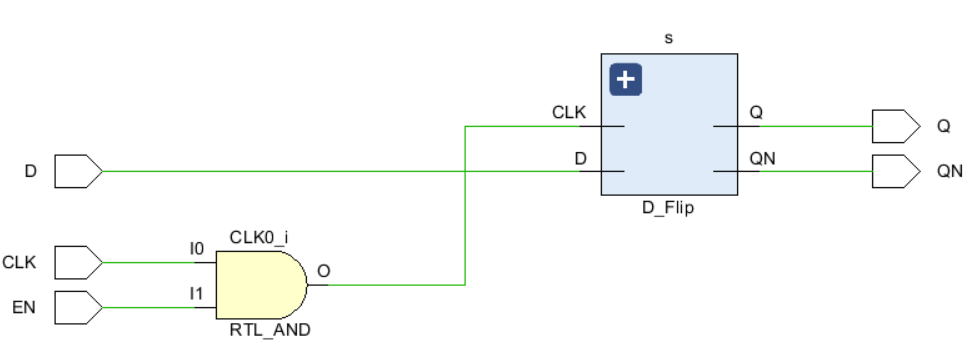
join

endmodule

仿真图形：



电路图：



分析：

在实现带使能端的D触发器时，我们调用了D触发器的模块，将CLK&EN的值作为CLK的值传入D触发器模块之中，这样就使得，当EN为0的时候，CLK&EN的值始终为0，不会处于上升沿，Q的值保持不变，D中的值无效；当EN为1的时候，CLK&EN的值就等于CLK的值，又回到了之前的D触发器，当CLK处于上升沿的时候，D的值就被赋给了Q。

可以看到，在仿真图形中，在0到10ns的过程中EN始终为0，本来Q和QN的值应该保持不变，但是由于Q和QN的初始值未知，所以导致Q和QN的值无法确定。在10到20ns的过程中，EN为1，当CLK处于上升沿时，Q和D的值相等，直到下一个CLK上升沿到来之前，Q的值都保持不变。在20到30ns的过程中，EN为0，D无效，Q保持不变。符合预期。

6.同步复位触发器

设计文件：

module reset\_flip(

input D,CLK,RESET,

output Q,QN

);

D\_Flip s(D&~RESET,CLK,Q,QN);

Endmodule

仿真文件：

module sim;

reg D,CLK,RESET;

wire Q,QN;

reset\_flip s(D,CLK,RESET,Q,QN);

initial begin

D=0;CLK=0;RESET=0;

end

always fork

forever #2 CLK=CLK+1;

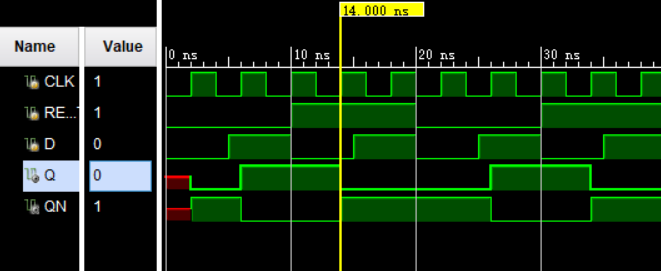
forever #5 D=D+1;

forever #10 RESET=RESET+1;

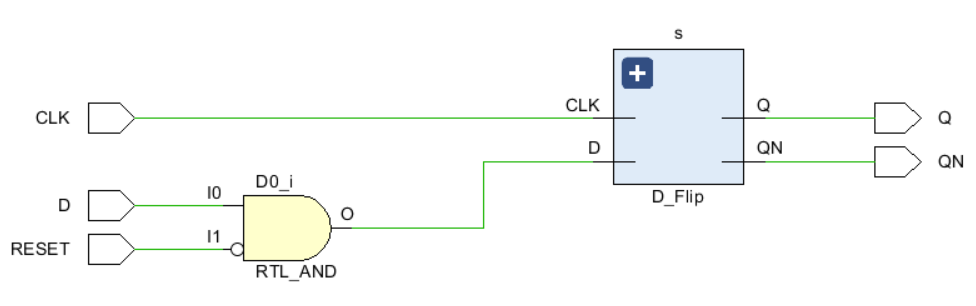
join

endmodule

仿真图形：



电路图：



分析：

在实现同步复位触发器时，我们调用了D触发器的模块，将D&~RESET的值作为D的值传入D触发器模块之中，这样就使得，当RESET为1的时候，D&~RESET的值始终为0，当CLK处于上升沿时候， D中的值无效，Q就被置0；当RESET为0的时候，D&~RESET的值就等于D的值，又回到了之前的D触发器，当CLK处于上升沿的时候，D的值就被赋给了Q。

可以看到，在仿真图形中，在0到10ns的过程中RESET始终为0，此时触发器和一般的D触发器完全一样，在 10ns刚到来的时候，RESET的值为0，D的值为1，此时CLK处于上升沿，Q仍被赋以D的值1，在14ns时，RESET的值为1，D的值为0，此时CLK处于上升沿，Q的值被置0。可以看到，在10ns到14ns的过程中，虽然RESET的值为1，但是CLK不处于上升沿，RESET无法将Q置0，只有等到14ns时CLK处于上升沿的时候，RESET才能将Q置0，满足同步的特性，符合预期结果。

7.异步复位触发器

设计文件：

module reset\_flip(

input D,CLK,RESET,

output Q,QN

);

wire Q1,Q1N;

D\_Flip s(D,CLK,Q1,Q1N);

assign Q=Q1&~RESET;

assign QN=~(Q1&~RESET);

endmodule

仿真文件：

module sim;

reg D,CLK,RESET;

wire Q,QN;

reset\_flip s(D,CLK,RESET,Q,QN);

initial begin

D=0;CLK=0;RESET=0;

end

always fork

forever #2 CLK=CLK+1;

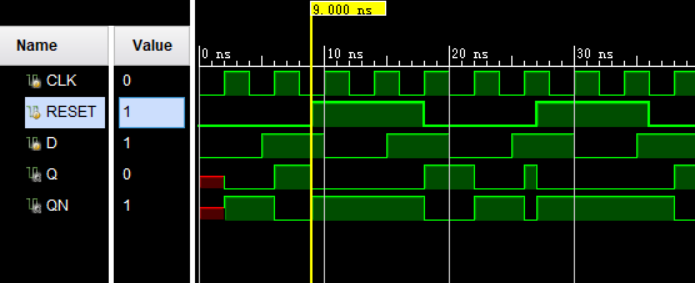
forever #5 D=D+1;

forever #9 RESET=RESET+1;

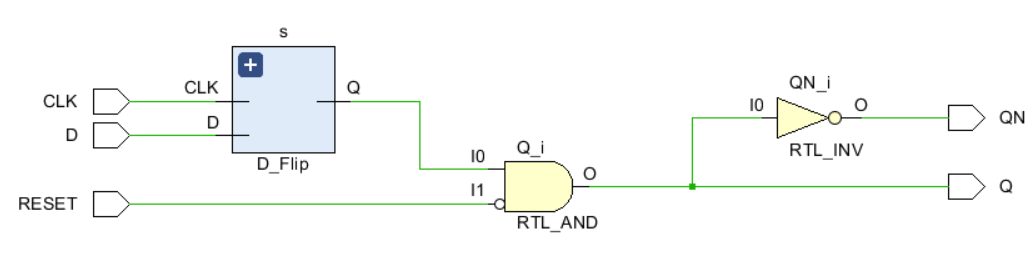
join

endmodule

仿真图形：



电路图：



分析：

在实现异步复位触发器的过程中，我首先调用D触发器的模块，将CLK和D进行代入后将D触发器的结果Q1和~RESET进行与操作，结果赋给Q。这样当RESET的值为1时，Q直接被置0；当RESET的值为0时，Q的值为Q1。

仿真图形中，在0到9ns的过程中，RESET的值为0，此时异步复位触发器跟普通的D触发器一样，在CLK处于上升沿的时候，D的值被赋给Q；在9ns到18ns的过程中此时RESET的值为1，不需要CLK处于上升沿状态，输出Q的值就被直接置0，这很好体现了异步的特性，符合预期结果。

8.触发器

设计文件：

module Flip(

input D,EN,RST,CLK,

output reg Q,QN

);

always@(posedge CLK)

if(RST) begin Q<=0;QN<=1; end

else if(EN) begin

Q<=D;QN<=~D;

end

endmodule

仿真文件：

module sim;

reg D,EN,CLK,RST;

wire Q,QN;

Flip s(D,EN,RST,CLK,Q,QN);

initial begin

D=0;CLK=0;RST=0;EN=0;

end

always fork

forever #2 CLK=CLK+1;

forever #5 D=D+1;

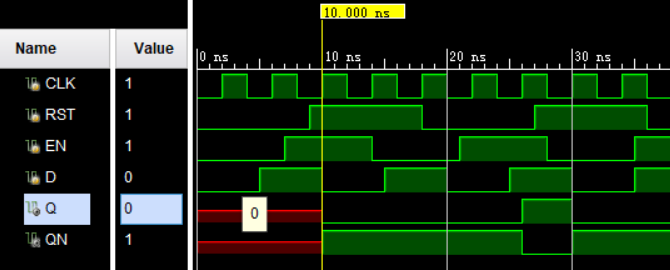
forever #7 EN=EN+1;

forever #9 RST=RST+1;

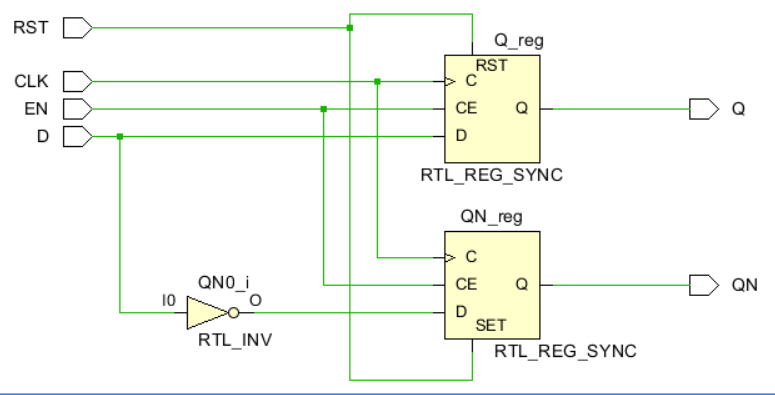
join

endmodule

仿真图形：



电路图：



分析：

在实现触发器的过程中，采用了always语句，只有当CLK发生变化的时候才能进入always语句，在语句内首先是对RST的判断，如果RST有效，则直接将Q置0，将QN置1；如果RST无效，再来对EN进行判断，如果EN有效，则将D的值赋给Q，将~D的值赋给QN,如果EN无效，则Q和QN保持原样。在这个触发器内RST复原信号的优先级是高于EN使能信号的优先级。

在仿真图形中，在2ns和6ns的时候CLK处于上升沿，但是此时RST和EN都无效，所以Q和QN保持原样，但是Q和QN初始值未知，所以在0到10ns的过程中，Q和QN都呈现红色状态。在10ns时CLK处于上升沿，RST为1，所以Q被置为0，QN被置1；在26ns时CLK处于上升沿，RST为0，EN为1有效，所以Q的值被修改为D的值1，QN的值对应的被修改为0。仿真结果符合预期

9.寄存器

设计文件：

module register(

Q,D,OE,CLK

);

parameter N=8;

output reg [N-1:0]Q;

input [N-1:0]D;

input OE,CLK;

always@(posedge CLK,posedge OE)

if(OE)Q<=8'bzzzz\_zzzz;

else Q<=D;

endmodule

仿真文件：

module sim;

parameter N=8;

reg [N-1:0]D;

reg OE,CLK;

wire [N-1:0]Q;

register s(Q,D,OE,CLK);

initial begin

D=0;OE=0;CLK=0;

end

always fork

forever #2 CLK=~CLK;

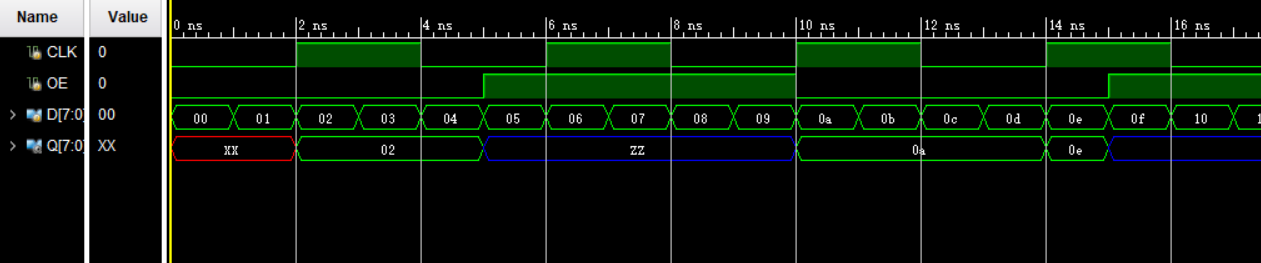
forever #5 OE=~OE;

forever #1 D=D+1;

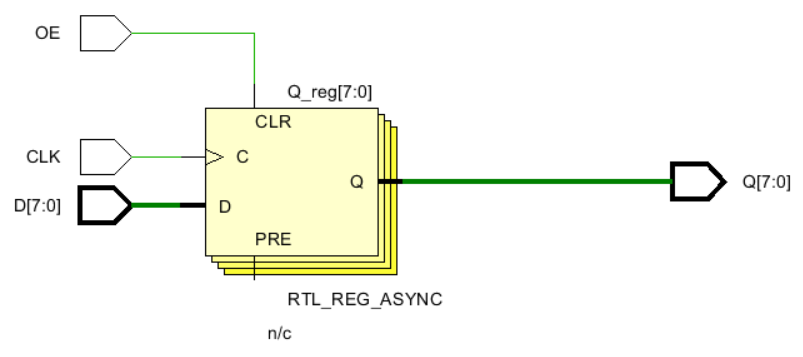
join

endmodule

仿真图形：



电路图：



分析：

在实现寄存器的过程中，我们使用了always语句，只有当CLK或者OE处于上升沿的时候才会进入always语句内，在语句中首先对OE进行判断，如果OE无效，则把Q置为高阻态z；如果OE有效，则把D的值全部赋给Q。

仿真图形中，在2ns时CLK处于上升沿，此时OE有效，D的值02被赋给了Q；在5ns的时候OE处于上升沿，但是此时CLK一直为0，这是Q就被置为了高阻态。符合预期

10.移位寄存器

设计文件：

module shift\_register(

S1,S0,D,Dsl,Dsr,Q,CLK,CR

);

parameter N=4;

input S1,S0,Dsl,Dsr,CLK,CR;

input [N-1:0]D;

output reg [N-1:0]Q;

always@(posedge CLK,posedge CR)

if(CR) Q<=0;

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

仿真文件：

module sim;

parameter N=4;

reg [N-1:0]D;

reg CR,CLK,S1,S0,Dsl,Dsr;

wire [N-1:0]Q;

shift\_register s( S1,S0,D,Dsl,Dsr,Q,CLK,CR);

initial begin

D=0;CR=1;CLK=0;S1=0;S0=0;Dsl=0;Dsr=0;

end

always fork

forever #2 CLK=~CLK;

repeat(1) #5 CR=~CR;

forever #1 D=D+1;

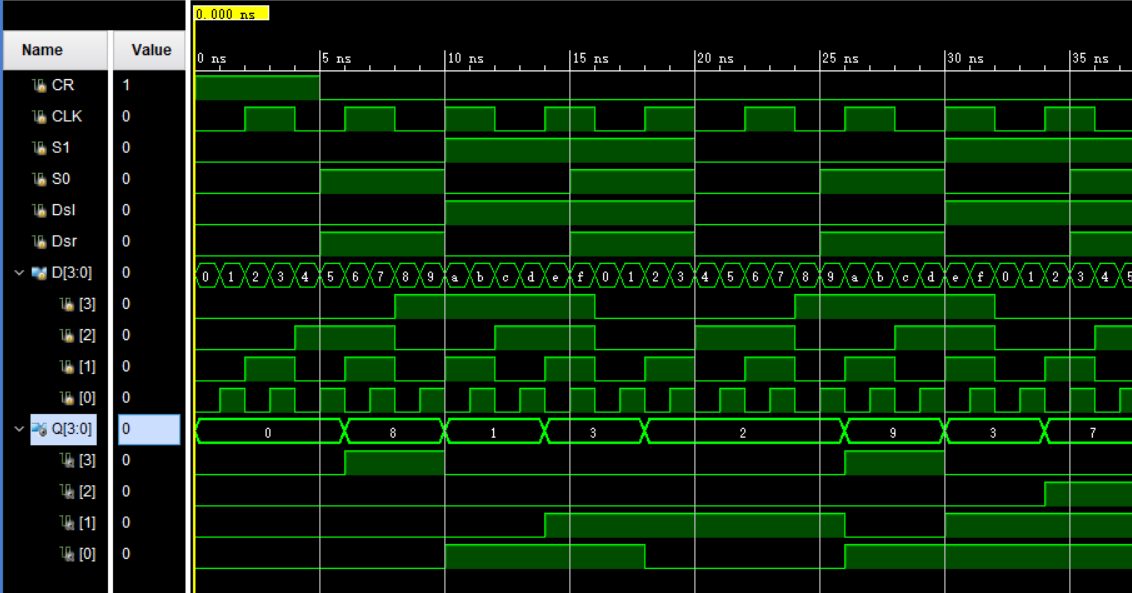
forever #5 {S1,S0}={S1,S0}+1;

forever #5 {Dsl,Dsr}={Dsl,Dsr}+1;

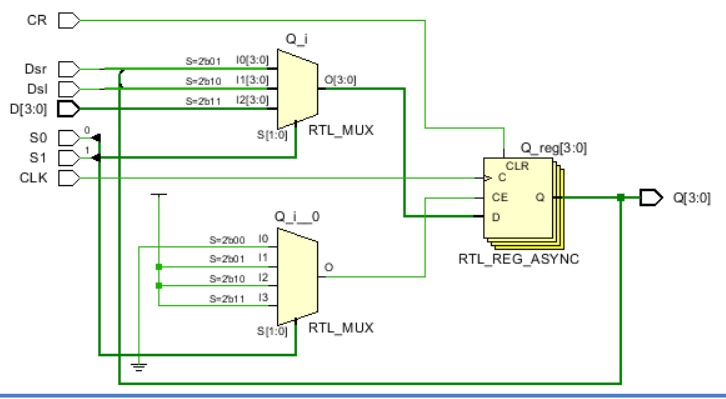
join

endmodule

仿真图形：



电路图：



分析：

由真值表可以直到，CR为异步清零信号，所以在always语句中@(posedge CLK or posedge CR),为的就是CR一旦处于有效状态，立即进入always语句内将Q置0。在语句内部，首先是对CR的if判断语句，如果CR有效，将Q置0，然后退出always语句；如果CR无效，使用case语句来判断S1和S0的值，在具体实现的过程中采用{S1,S0}这个变量作为case语句的判断内容，将S1和S0凑到一起视为一个变量，如果S1和S0都为0，则Q保持不变；如果S1为0，S0为1，则将Q[0]舍弃，并让Q[N-1:1]整体右移一位，最高位换成输入中的Dsr；如果S1为1，S0为0，则将Q[N-1]舍弃，并让Q[N-2:0]整体左移一位，最低位换成输入中的Dsl；如果S1为1，S0为1，则将D的值赋给Q。

在仿真图形中，首先在0ns的时候，CR的值为1，此时直接进入always语句内将Q置0。然后0ns到5ns的时间内CR的值都为1，进入always语句也会直接将Q置0。在第5ns时，CR处于下降沿，CR的值变为0，但是因为此时CLK并没有处于上升沿，所以无法进入always语句内，对应的Q值也保持不变。到了6ns的时候，此时CLK处于上升沿，CR为0，此时S1等于0，S0等于1，Q执行右移操作,原来的Q[3:1](000)变成了现在的Q[2:0](000),Q[3]的值被修改为Dsr对应的值1，所以此时Q的值是1000；到了第10ns的时候，CLK处于上升沿，此时CR仍然为0，S1为1，S0为0，Q执行左移操作，原来的Q[2:0](000)变成了现在的Q[3:1](000),Q[0]被修改成Dsl的值1，所以此时的Q的值是0001；在第18ns的时候，CLK处于上升沿，此时CR仍然为0，S1为1，S0为1，D的值2被赋给了Q，所以此时Q对应的值是2；在第22ns的时候，CLK处于上升沿，此时CR仍然为0，S1为0，S0为0，Q保持不变，仍为之前的值2。仿真结果符合预期。

11.计数器

设计文件：

module counter74x161(

CEP,CET,PE,CLK,CR,D,TC,Q

);

parameter N=8;

parameter M=49;

input CEP,CET,PE,CLK,CR;

input [N-1:0]D;

output reg TC;

output reg [N-1:0]Q;

wire CE;

assign CE=CEP&CET;

always@(posedge CLK,negedge CR)

if(~CR) begin Q<=0;TC=0;end

else if(~PE)Q<=D;

else if(CE)

if(Q==M-1) begin

TC<=1;

Q<=0;

end

else Q<=Q+1;

endmodule

仿真文件：

module sim;

parameter N=8;

reg [N-1:0]D;

reg CEP,CET,PE,CLK,CR;

wire [N-1:0]Q;

wire TC;

counter74x161 s(CEP,CET,PE,CLK,CR,D,TC,Q);

initial begin

{CEP,CET,PE,CLK,CR}=0;

D=0;

end

always fork

forever #2 CLK=~CLK;

repeat(1) #3 CR=~CR;

forever #2 D=D+1;

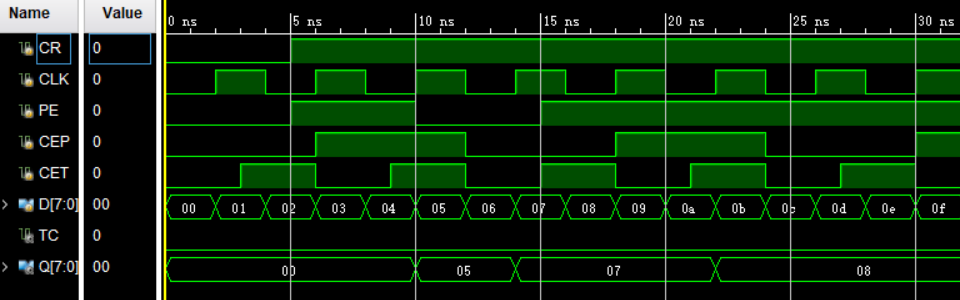
forever #3 {CEP,CET}={CEP,CET}+1;

repeat(1) #5 PE=PE+1;

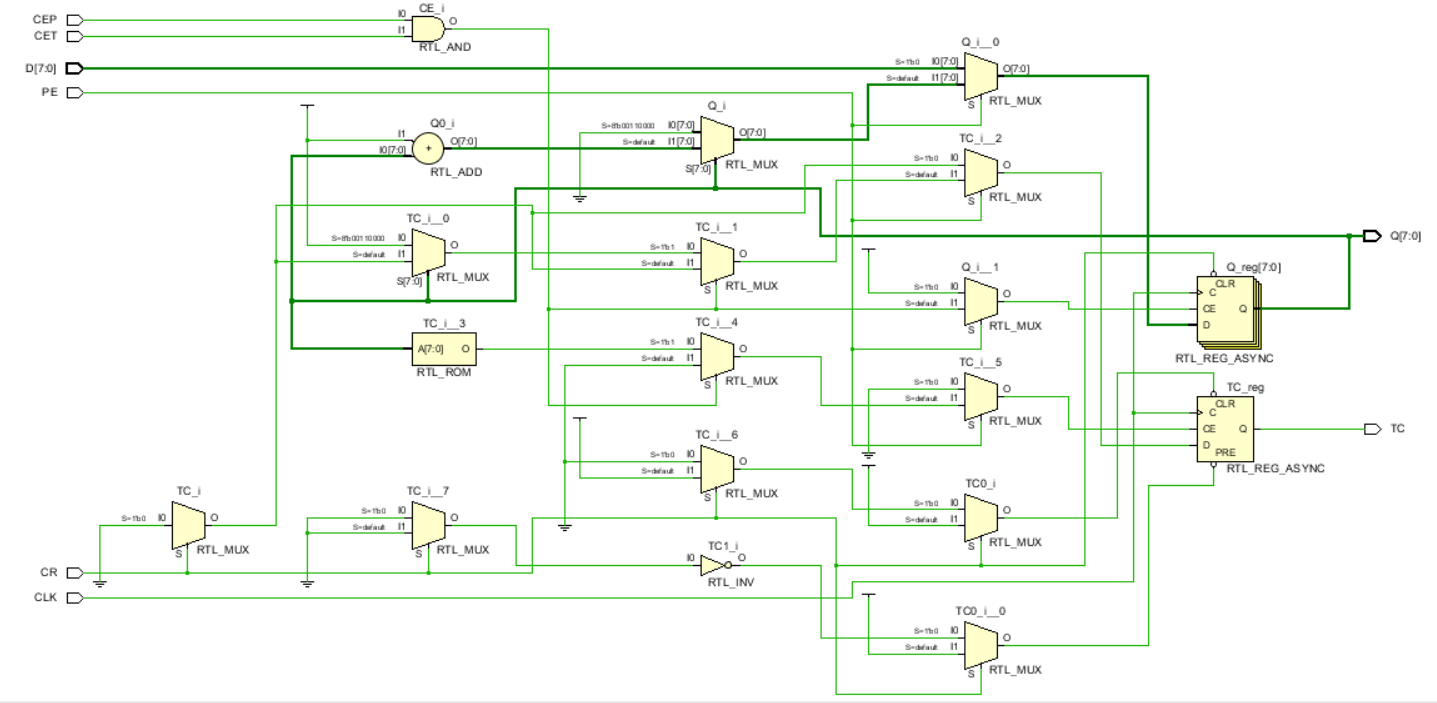
join

endmodule

仿真图形：



电路图：



分析：

在实现计数器的过程中，首先定义一个wire类型的中间变量，并使用数据流描述，令CE等于CER&CET，这样当CE为1的时候说明CER和CET同时为1，当CE为0的时候说明CER和CET两个变量之中至少有一个为0。低有效CR是实现异步清零的作用，CLK充当时钟信号，所以在always语句的判定条件应该为（posedge CLK or negedge CR），在语句内首先进行对CR的判断，如果CR为0，则立即对Q和TC置0，然后退出always语句；如果CR无效，再对PE进行判断，如果PE为0，则将D的值赋给Q,然后退出always语句；如果PE无效，再对CE进行判断，如果CE为1，则对输出进行+1操作，具体实现方法为：如果Q不等于最大值M-1，那么直接让Q+1即可；如果Q等于最大值M-1,那么让Q置0，同时让进位输出TC置1；如果CE无效，则让输出Q保持不变即可。

在仿真图形中，首先将CR初始化为0，所以输出Q和TC在一开始就被置0，然后在0到5ns的过程中，CR一直有效，所以Q和TC一直为0；在5ns的时候，CR被置1，但此时CLK并未处于上升沿，所以无法进入always语句，Q和TC依旧保持不变。在6ns的时候，CLK处于上升沿，程序进入always语句内，但是此时PE为1无效，CER为0进而导致CE为0无效，所以此时Q依旧保持不变；在10ns的时候，CLK处于上升沿，此时PE为0有效，Q的值被修改成D的值05；在14ns的时候同样也是这个情况，Q的值被修改成D的值07；在第22ns的时候，CLK处于上升沿，此时PE为1无效，CEP和CET都为1使得CE为1有效，此时输出Q执行+1操作，因为Q此时为07，并未达到最大值15，所以直接让Q+1即可，所以Q的值变成了8，TC依旧为0。

12.3分频计数器

设计文件：

module FSM\_case(

input clk,reset,

output y

);

reg[2:0] state,nextstate;

always@(posedge clk,posedge reset)

if(reset)state=1;

else state=nextstate;

always@(posedge clk)

case(state)

1:nextstate=2;

2:nextstate=4;

4:nextstate=1;

default:nextstate=1;

endcase

assign y=state[0];

endmodule

仿真文件：

module sim;

reg clk,reset;

wire y;

FSM\_case s(clk,reset,y);

initial begin

{clk,reset}=1;

end

always fork

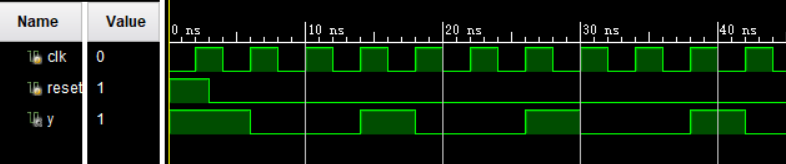
forever #2 clk=~clk;

repeat(1) #3 reset=~reset;

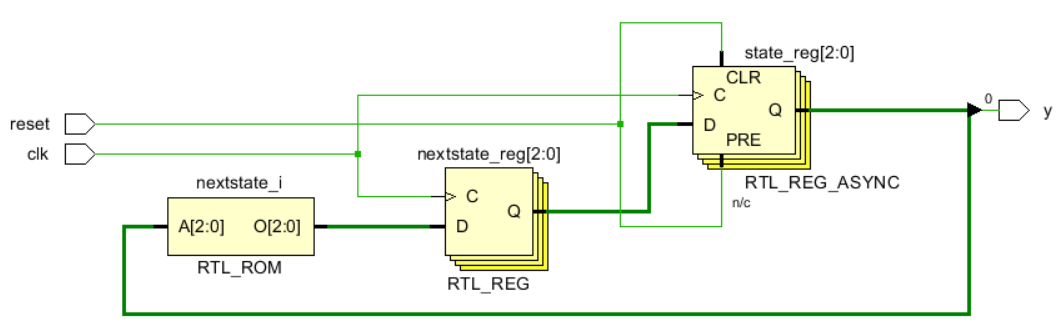
join

endmodule

仿真图形：



电路图：



分析：

在实现3分频计数器的过程中，首先定义了两个reg[2:0]类型的变量state和nextstate，然后第一个always语句用来实现clk控制下的state更新以及reset的异步清理功能，在always的判断条件中是（posedge clk, posedge reset）,在语句内是一个对reset进行判断的if语句，如果reset有效，则直接将state置为2’b001；如果reset无效，则将state置为nextstate的值。然后是第二个always语句，该语句是用来实现在clk控制下更新nextstate的值，敏感条件是(posedge clk),语句内是一个case语句，case语句针对state的值做了不同的操作，如果state为001，则nextstate置为010；如果state为010，则nextstate置为100；如果state为100，则nextstate置为001；state的值出现异常情况则将nextstate置为001。这样就保持了state从001状态开始，经过三个clk上升沿后，state的值以此变为010，100，010。同时用数据流形式来描述y，将y定义成assign y=(state==’b001),这样当经过三个clk上升沿之后y的值才能变成1，y的周期变成了clk周期的三倍，实现了3分频。

在仿真图形中，一开始reset为1，程序进入always语句将state修改为001，这时y的值就为1，然后到了2ns的时候，clk处于上升沿，在第二个always语句中将nextstate修改为010，此时reset仍有效，state保持不变，到了3ns的时候reset变为0，但是此时clk没有处在上升沿，无法出发always语句的敏感条件，所以此时state和nextstate都保持不变。此后reset一直无效，每经过以此clk上升沿，state和nextstate的状态都会发生改变。clk的周期是4ns，y的周期是12ns，实现了3分频。clk每隔4ns就会处于上升沿，y每隔三次clk上升沿就会变成1。符合预期。

13检测01子序列

设计文件：

module FSM\_cf(

input clk,reset,a,

output y

);

reg[1:0] state,nextstate;

always@(posedge clk,posedge reset)

if(reset) state=0;

else state=nextstate;

always@(posedge clk)

nextstate={state[0]&a,~a};

assign y=state[1];

endmodule

仿真文件：

module sim;

reg clk,reset,a;

wire y;

FSM\_cf s(clk,reset,a,y);

initial begin

{clk,reset,a}=2;

end

always fork

forever #2 clk=~clk;

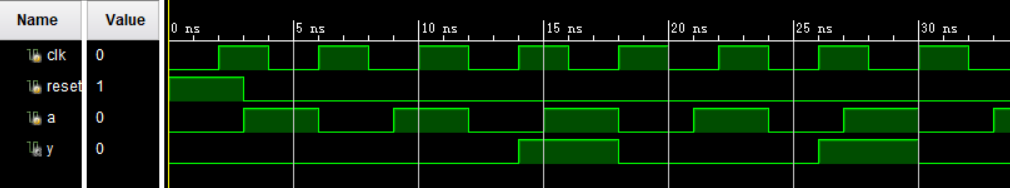
repeat(1) #3 reset=~reset;

forever #3 a=a+1;

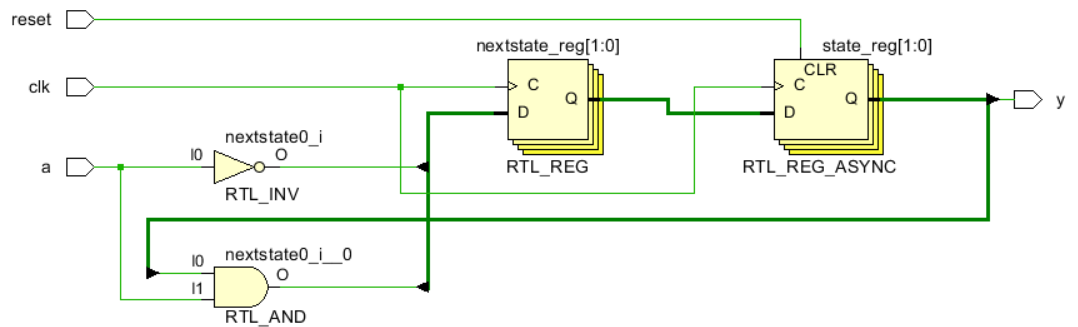
join

endmodule

仿真图形：



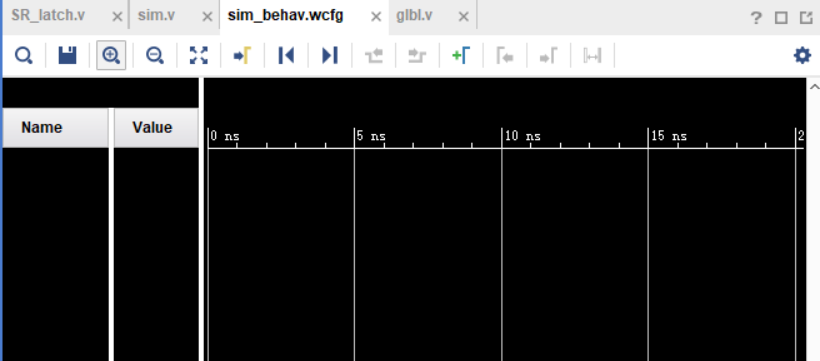
电路图：



分析：在实现检测01子序列的过程中，采用两个always语句和一个assign语句，第一个always语句用来实现clk控制下的state更新以及reset的异步清理功能，在always的判断条件中是（posedge clk, posedge reset）,在语句内是一个对reset进行判断的if语句，如果reset有效，则直接将state置为2’b00；如果reset无效，则将state置为nextstate的值。然后是第二个always语句，该语句是用来实现在clk控制下更新nextstate的值，敏感条件是(posedge clk),语句内是一个case语句，case语句针对state的值做了不同的操作，**➀**如果state为00，在这种情况下，如果a为0，符合01子序列的第一个值，那么nextstate就会变成下一个状态01，如果a为1，不符合01子序列的第一个值，那么nextstate还是00，state又会回到原状态;**➁**如果state为01，说明此时以及最近的输入是0，如果此时输入a为1，满足了01子序列的要求，那么可以将nextstate置为10，等到下一次clk上升沿到来，state被置为10，对应的y的值就被置1，如果此时输入a为0，说明仍然只满足了01子序列的第一个条件0，还需要下一次输入a为1，那么此时的nextstate应该被置为01，state的下一次状态就是其本身。**➂**如果state的值为10，说明此时01子序列已经得到了满足，我们需要寻找下一个01子序列，此时如果a是0,说明已经满足了01子序列的第一个条件0，此时将nextstate置为01即可，如果此时输入a为1，说明不满足01子序列的任何条件，需要将nextstate置为00，从原状态开始。所以在第二个always语句内可以统一写成nextstate={state[0]&a,~a}。采用数据流的方式对y进行描述，如果state的值为10，则y为1，反之y为0。

在仿真图形中，首先将reset置为1，这样就使得state初始化为00，在2ns时clk处于上升沿，进入第二个always语句，将nextstate置为01；在3ns时reset被置为0，但是clk没有处于上升沿；在6ns的时候，clk处于上升沿，此时a为0，state被置为01，nextstate被置为01；再到10ns时，clk处于上升沿，此时a为1，state被置为01，nextstate被置为10；在14ns时，clk处于上升沿，此时a为0，state被置为10，nextstate被置为00，对应的y等于1。符合预期结果

**五、调试和心得体会**



在打开verilog编写第一个SR锁存器的代码后运行仿真文件，在仿真图形中一个变量都没有，第一次遇到这种情况手足无措，不知道变量为什么突然都没了。后面问了老师之后才知道这可能是软件的问题，遇到这种情况需要手动添加变量。

设计与测试这些基本元件是深入理解数字电路原理的重要一步。通过实际操作，我更好地理解了逻辑门、时钟信号、数据传输等基本概念。这有助于将理论知识转化为实际的硬件设计。在设计这些元件时，我学会了制定清晰的设计规范和计划。正确的设计决策是确保电路功能正确性的关键。同时，我还学会了使用硬件描述语言Verilog来实现我的设计，这有助于更好地组织和管理复杂的电路。在设计计数器和状态机时，我深刻理解了时序和同步问题的重要性。时序问题可能导致电路功能不正确，因此需要特别注意时钟信号的管理和同步。这些基本元件是数字系统中的构建块，它们在计算机硬件、通信设备、嵌入式系统等领域中有广泛的应用。通过设计和测试这些元件，我更好地理解了它们在实际应用中的作用。