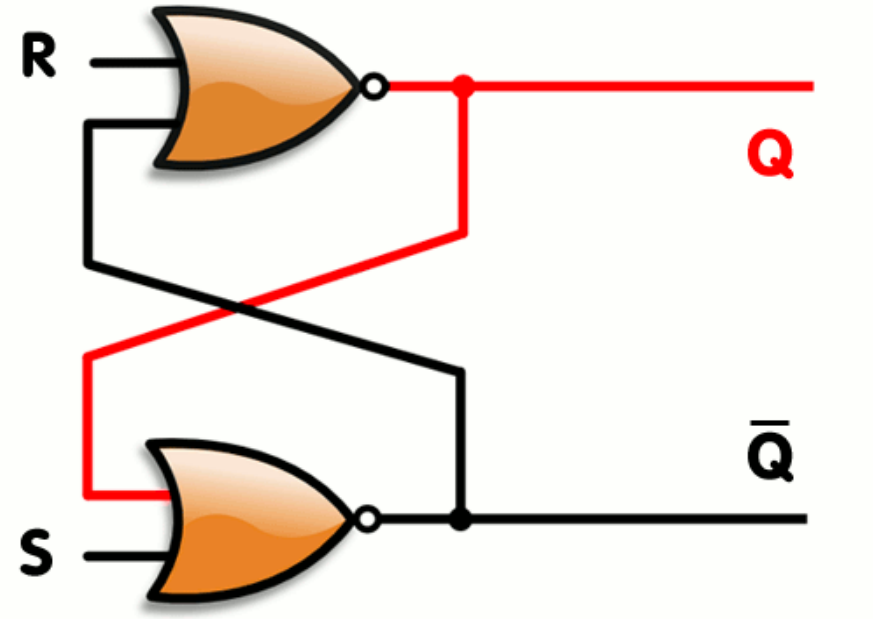
实验8：锁存器+触发器

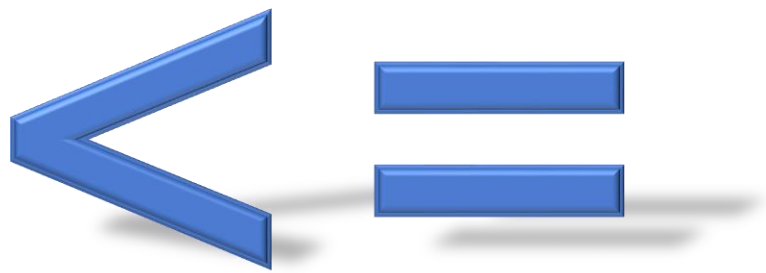


xgsun@fudan.edu.cn

孙晓光

2022-11-5

1



“=”阻塞赋值 vs 非阻塞赋值“<=”

组合逻辑

教材P121，例4.23

```
1  // 全加器：阻塞赋值
2  module FullAdder_Blocking(
3      input  logic a, b, Cin,
4      output logic S, Cout);
5
6      logic  p, g;
7
8      always_comb
9      begin
10         p = a ^ b;
11         g = a & b;
12         S = p ^ Cin;
13         Cout = g | (p & Cin);
14     end
15 endmodule
```

只有**阻塞**下一行的执行才能串行

教材P126，例4.28

```
1  // 全加器：非阻塞赋值(不推荐使用)
2  module FullAdder_Nonblocking(
3      input  logic a, b, Cin,
4      output logic S, Cout);
5
6      logic  p, g;
7
8      always_comb
9      begin
10         p <= a ^ b;          //nonblocking
11         g <= a & b;          //nonblocking
12         S <= p ^ Cin;        //nonblocking
13         Cout <= g | (p & Cin); //nonblocking
14     end
15 endmodule
```

因并行计算，
导致两次才计算出结果

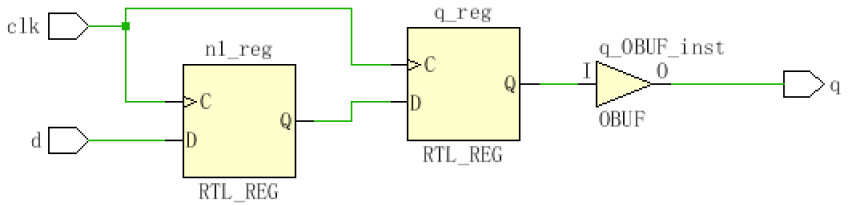
结论：在组合逻辑中，使用”=”赋值！

“=”阻塞赋值 vs 非阻塞赋值“<=”

时序逻辑

教材P118，例4.20

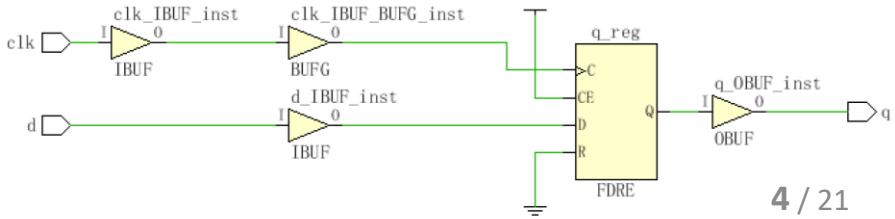
```
1  // 同步器：非阻塞赋值
2  module Sync_Nonblocking(
3      input  logic clk,
4      input  logic d,
5      output logic q );
6
7      logic n1;
8
9      always_ff @(posedge clk)
10     begin
11     并  n1 <= d;  //nonblocking
12     行  q  <= n1; //nonblocking
13     end
14 endmodule
```



教材P127，例4.29

```
1  // 同步器：阻塞赋值
2  module Sync_Blocking(
3      input  logic clk,
4      input  logic d,
5      output logic q );
6
7      logic n1;
8
9      always_ff @(posedge clk)
10     begin
11     串→n1 = d;    //Blocking
12     行→q  = n1;   //Blocking
13     end
14 endmodule
```

n1被优化没了



结论：在时序逻辑中，必须使用”<=”赋值！

“=”阻塞赋值 vs 非阻塞赋值“<=”

- =： **阻塞赋值**运算符。**顺序**执行。

  “右式计算”和“左式更新”完全完成之后，才开始执行下一条语句。

串行
$$\begin{cases} B = A & \text{// 将A送给B} \\ C = B + 1 & \text{// 将B加1传送给C，最后C的值等于A+1} \end{cases}$$

- <=： **非阻塞赋值**运算符。**并行**执行。 当前语句的执行不会阻塞下一语句的执行。

  1）在开始时，计算所有非阻塞赋值右侧表达式。

  2）在结束时，更新所有非阻塞赋值左侧表达式。

并行
$$\begin{cases} B <= A & \text{// 将A的值保存在一个存储区} \\ C <= B + 1 & \text{// 将B加1的值保存在另外一个存储区} \end{cases}$$

当所有的顺序表达式右侧都计算和保存后，赋值到左边的操作才会发生。

此时，**C等于B的起始值加1**，而不是A+1。

使用原则：“=” “<=”

① 在**assign**中，必须使用**阻塞赋值 (=)**。

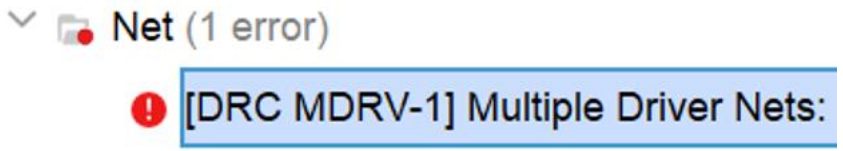
② 用**always**建立**组合电路**时，用 **阻塞赋值 (=)**。

③ 用**always**建立**时序电路**时，用**非阻塞赋值 (<=)**。

④ 在**同一个always块**中同时有**时序**和**组合**电路时，用**非阻塞赋值(<=)**。

⑤ 在**同一个always块**中**不要**既用非阻塞赋值(=)又用阻塞赋值(<=)。

⑥ **不要**为**同一个**变量赋值两次赋值。

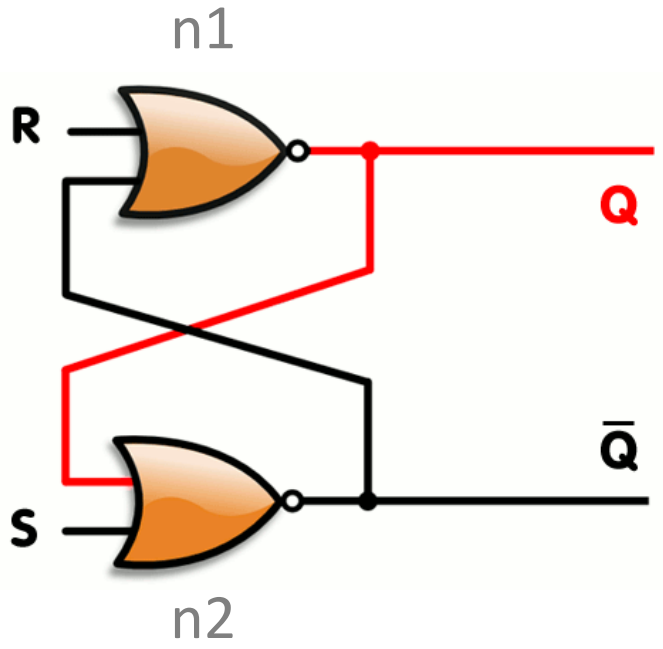


2

锁存器

RS 锁存器

```
1  // 用两个或非门首尾相连
2  module RS_latch(
3      input  logic R,
4      input  logic S,
5      output logic Q);
6
7      logic Qnot;
8
9      nor n1(Q   , R, Qnot);
10     nor n2(Qnot, S, Q);
11 endmodule
```



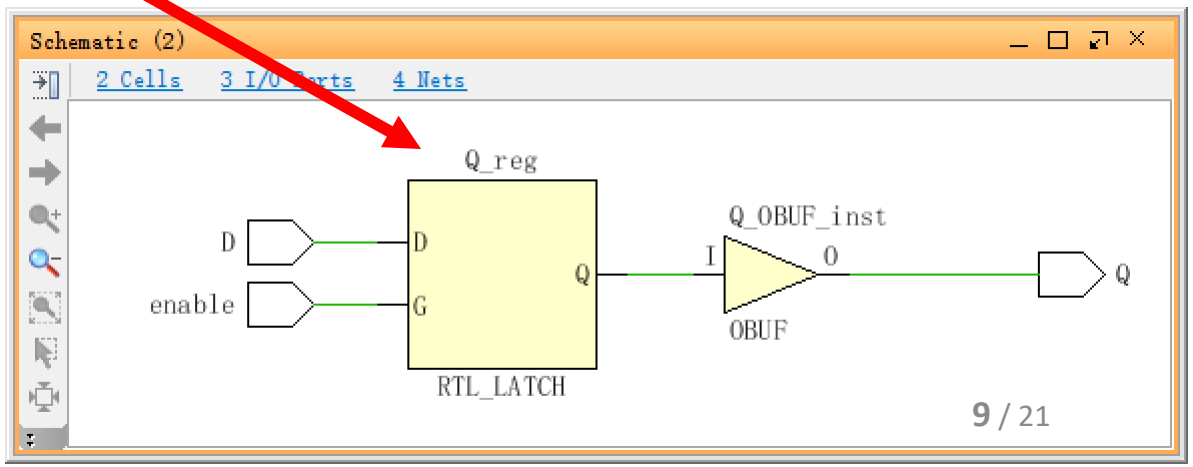
不是所有的综合工具都支持**锁存器**

D 锁存器

教材P119 HDL例4.21

```
1  // D锁存器
2  module D_Latch(
3      input  logic enable,
4      input  logic D,
5      output logic Q );  // output reg Q);
6
7      always_latch
8      // always @(enable, D)
9          if(enable) Q <= D;
10         //等同于if(enable ==1)
11 endmodule
```

不是所有的综合工具都支持锁存器



SystemVerilog

数据类型：**logic**。代替Verilog中的**wire**、**reg**

always语句细化为3种：

- **always_comb**：表示**组合逻辑**的过程
- **always_latch**：表示**锁存逻辑**的过程
- **always_ff**　　：表示**时序逻辑**的过程

Verilog

```
module flop (input          clk,
             input    [3:0] d,
             output reg [3:0] q);
  always @(posedge clk)
    q <= d;
endmodule
```

SystemVerilog

```
module flop (input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);
  always_ff @(posedge clk)
    q <= d;
endmodule
```

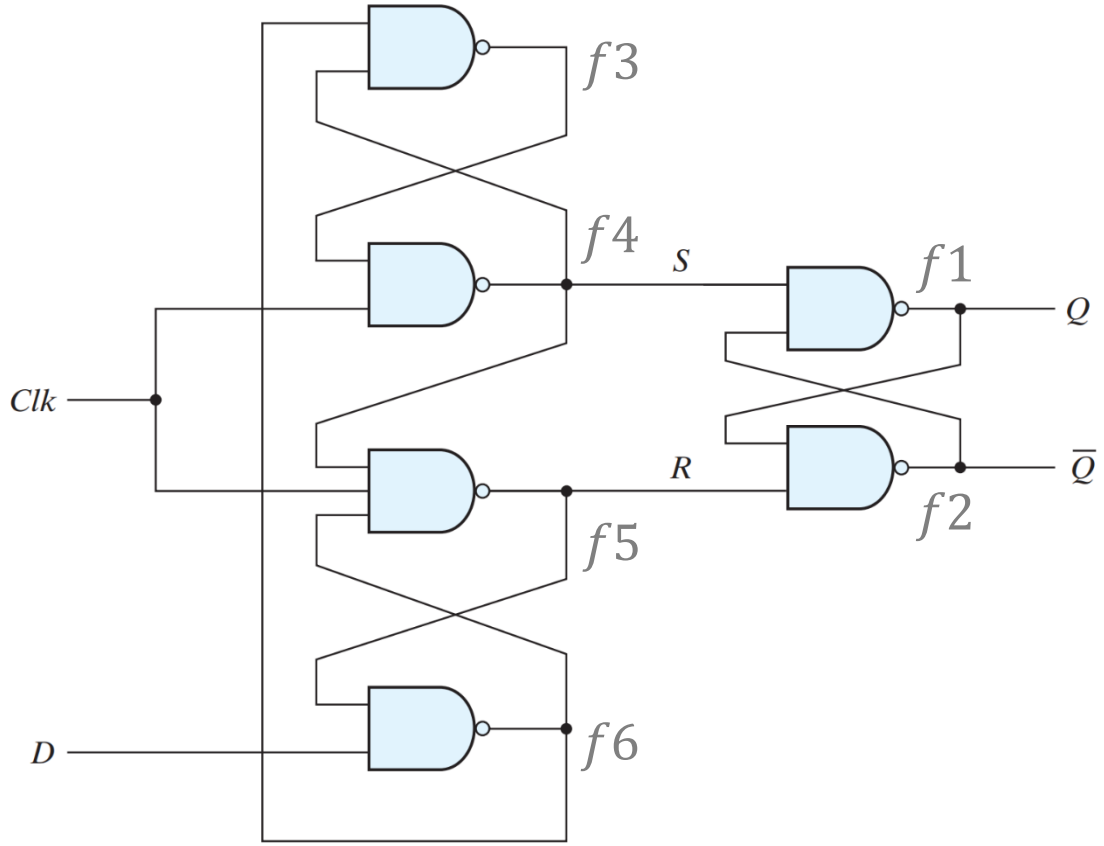
3

触发器

正边沿触发的 D 触发器

```
1  // 正边沿触发的D触发器
2  module D(
3      input  logic clk,
4      input  logic D,
5      output logic Q,
6      output logic notQ);
7
8      logic f1, f2, f3, f4, f5, f6;
9
10     assign f1 = ~(f4 & f2);
11     assign f2 = ~(f1 & f5);
12     assign f3 = ~(f6 & f4);
13     assign f4 = ~(f3 & clk);
14     assign f5 = ~(f4 & clk & f6);
15     assign f6 = ~(f5 & D);
16     assign    Q = f1;
17     assign notQ = f2;
18 endmodule
```

所有的CPLD和**FPGA**都包含有成千上万的D触发器，且用专门方法实现。

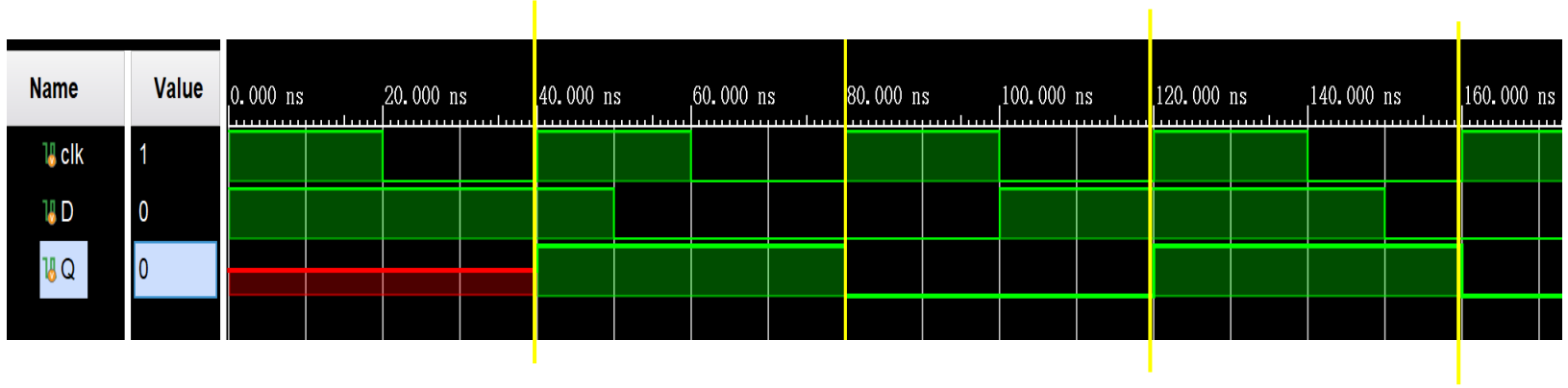


FlipFlop

正边沿触发的 D 触发器 仿真

```
1  // 正边沿触发的D触发器
2  module D(
3      input  logic clk,
4      input  logic D,
5      output logic Q,
6      output logic notQ);
7
8      logic f1, f2, f3, f4, f5, f6;
9
10     assign f1 = ~(f4 & f2);
11     assign f2 = ~(f1 & f5);
12     assign f3 = ~(f6 & f4);
13     assign f4 = ~(f3 & clk);
14     assign f5 = ~(f4 & clk & f6);
15     assign f6 = ~(f5 & D);
16     assign    Q = f1;
17     assign notQ = f2;
18 endmodule
```

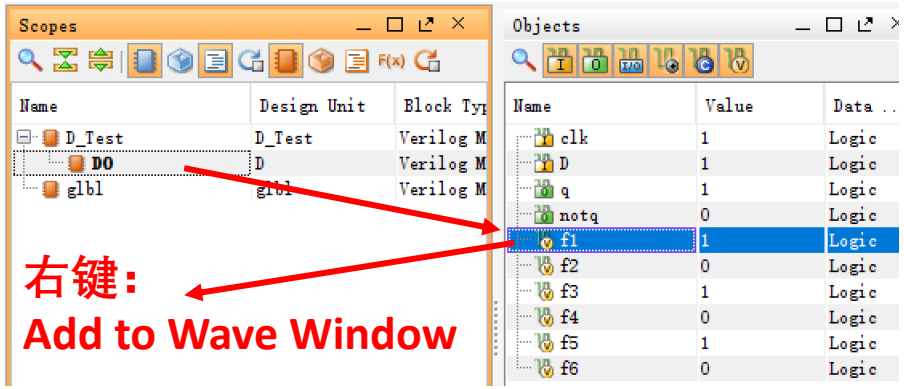
在每个时钟上升沿，Q被置为D的值

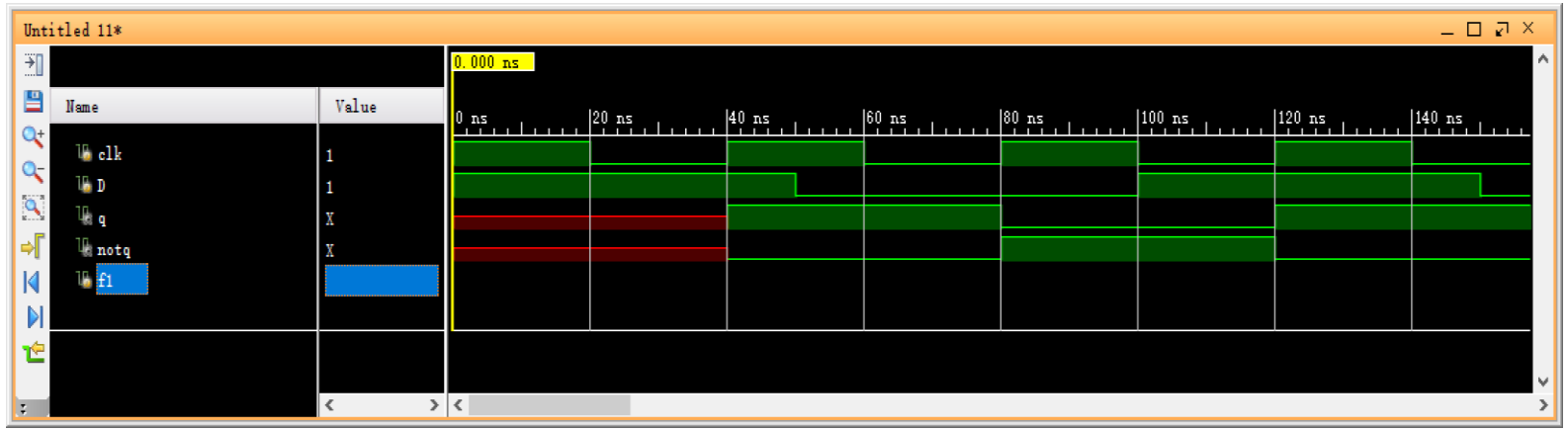



```
1  `timescale 1ns / 1ps
2  module D_Sim( );
3      logic clk , D;
4      logic Q, notQ;
5
6      //实例化
7      D D0(clk, D, Q, notQ);
8
9      //产生周期性时钟
10     always
11     begin
12         clk = 1; #20; clk = 0; #20;
13     end
14
15     //产生周期性D信号
16     always
17     begin
18         D = 1; #50; D = 0; #50;
19     end
20 endmodule
```

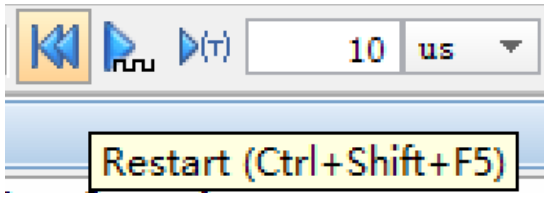
添加仿真变量

①

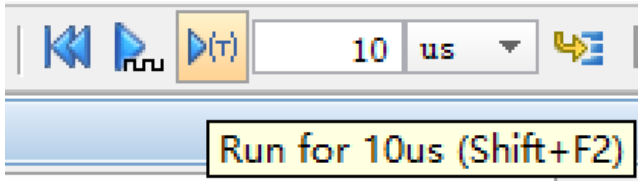

Scopes
Objects
Name
Design Unit
Block Typ
D_Test
D_Test
Verilog M
D0
D
Verilog M
glbl
glbl
Verilog M
Name
Value
Data
clk
D
q
notq
f1
f2
f3
f4
f5
f6
Logic
右键：
Add to Wave Window

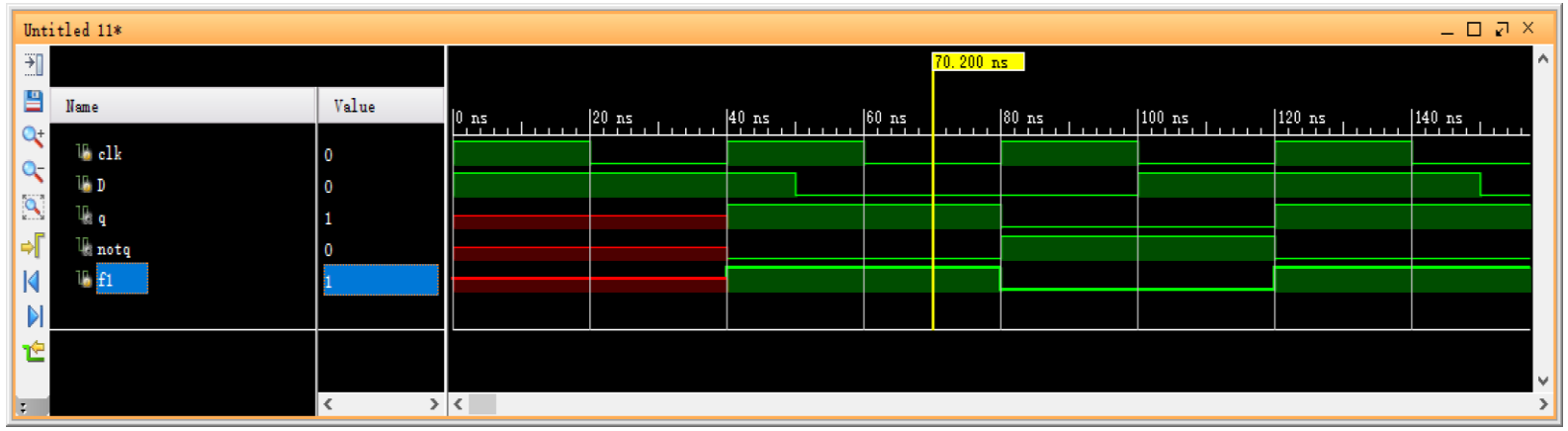

Untitled 11*
0.000 ns
Name
Value
clk
D
q
notq
f1
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns

②


10
us
Restart (Ctrl+Shift+F5)

③


10
us
Run for 10us (Shift+F2)


Untitled 11*
70.200 ns
Name
Value
clk
D
q
notq
f1
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns

Verilog 中的4种 **循环语句**

必须用在 **initial** 或 **always**语句中

```
initial                    产生8个时钟周期，
begin                      每个周期是10个单位时间
    clock = 1'b0;
    repeat (16)
      #5 clock = ~ clock;
end
```

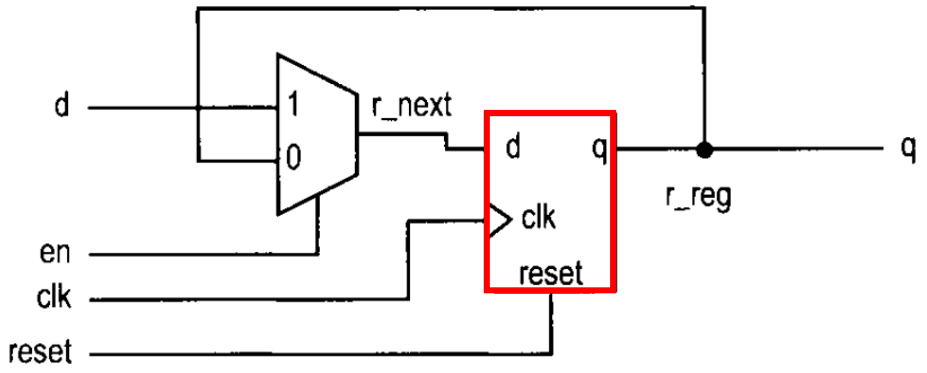
```
initial                    周期=20个单位时间
begin
    clock = 1'b0;
    forever
      #10 clock = ~ clock;
end
```

```
integer count;
initial
begin
    count = 0;
    while (count < 64)
      #5 count = count + 1;
end
```

```
initial
begin
    for (i = 0; i < 8; i = i + 1)
    begin
      // procedural statements
    end
end
```

D 触发器

```
1  module DFF( input  logic clk,
2              input  logic d,
3              output logic q );
4
5      always_ff @(posedge clk)
6          q <= d;
7  endmodule
```

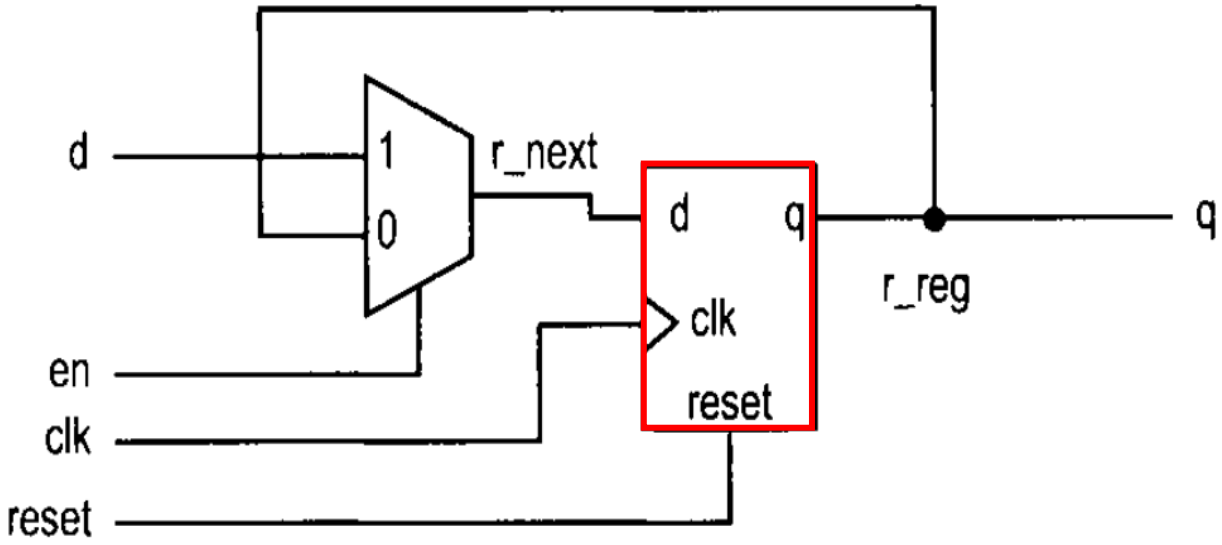



```
1  // DFF with asynchronous reset
2  module DFF_reset(
3      input  logic clk, reset,
4      input  logic d,
5      output logic q );
6
7      always_ff @(posedge clk, negedge reset)
8          if(reset) q <= 1'b0;
9          else      q <= d;
10 endmodule
```

```
1  // DFF with synchronous enable
2  module DFF_reset_enable(
3      input  logic clk, reset, enable,
4      input  logic D,
5      output logic Q  );
6
7      always @(posedge clk, posedge reset)
8          if (reset)
9              Q <= 1'b0;
10         else if (enable)
11             Q <= D;
12         //无需else,表示reset=0时，Q<=Q以前的值
13 endmodule
```

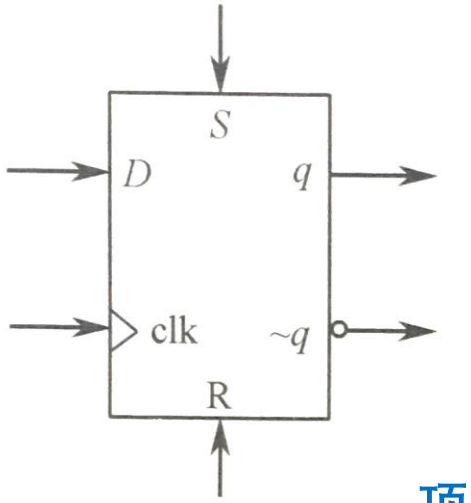
D触发器：有限状态机描述

```
1  module DFF_FSM(input  logic clk, reset, en,
2                 input  logic d,
3                 output logic q  );
4
5      logic r_reg, r_next;
6
7      // next-state logic                          ①
8      always @(*)
9          if (en) r_next = d;
10         else    r_next = r_reg;
11
12     // DFF                                       ②
13     always @(posedge clk , posedge reset)
14         if (reset) r_reg <= 1'b0;
15         else       r_reg <= r_next;
16
17     // output logic                              ③
18     always @(*)
19         q = r_reg;
20 endmodule
```



带有异步置位和清零端的正边沿触发 D 触发器

```
1  // 带有置位和清零端的正边沿触发 D 触发器
2  module D_cs(input  logic clk, clr, set,
3              input  logic D,
4              output logic q,  notq );
5
6      assign notq = ~q;
7
8      always_ff @(posedge clk, posedge clr, posedge set)
9      // always @(posedge clk, posedge clr, posedge set)
10     if(set == 1)
11         q <= 1;
12     else if(clr == 1)
13         q <= 0;
14     else
15         q <= D;
16 endmodule
```

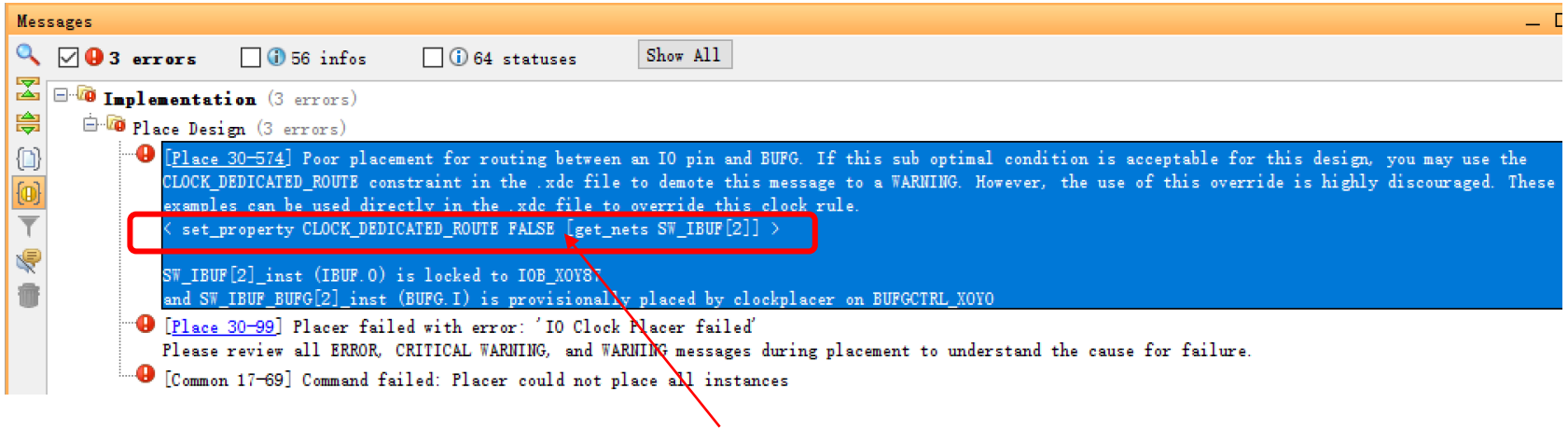


S	R	D	clk	q	~q
0	0	0	↑	0	1
0	0	1	↑	1	0
1	1	X	X	1	0
0	1	X	X	0	1
0	0	X	0	q_0	$\sim q_0$

顶层文件

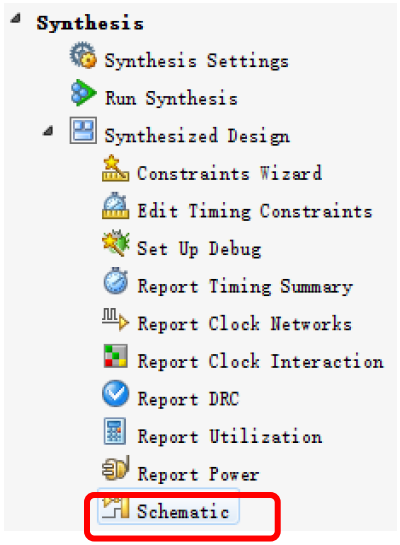
```
1  module D_cs2_Top(input  logic CLK100MHZ,
2                   input  logic [2:0] SW,
3                   output logic [1:0] LED);
4
5      D_cs2 D1(.clk(CLK100MHZ),
6               .D(SW[0]),
7               .clr(SW[1]),
8               .set(SW[2]),
9               .q(LED[0]),
10              .notq(LED[1]));
11 endmodule
```

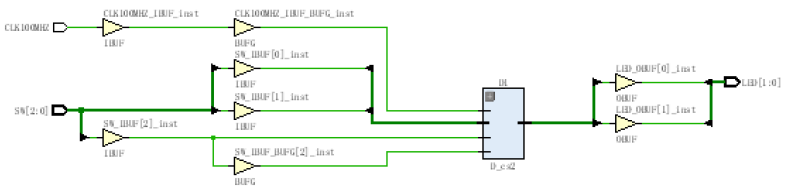
Run Implementation 时报错



解决方案1： 直接将 set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SW_IBUF[2]] 保存到约束文件中。

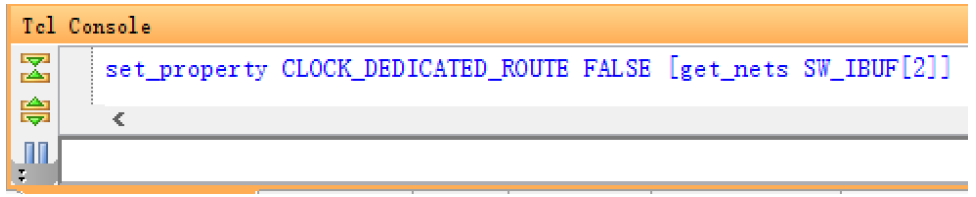
解决方案2： ① 打开综合的Schematic





② 在Tcl Console中输入 / 直接将之保存到约束文件中

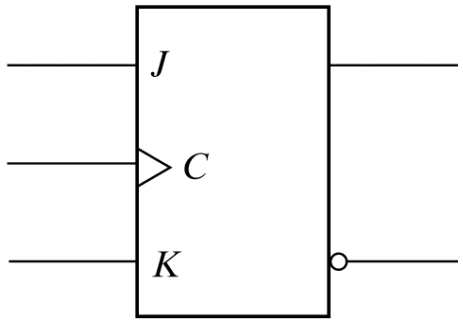
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SW_IBUF[2]]



③ 再次Run Implementation, 保存上述输入到引脚约束文件中(保存在最后一行)

上边沿触发的 JK 触发器

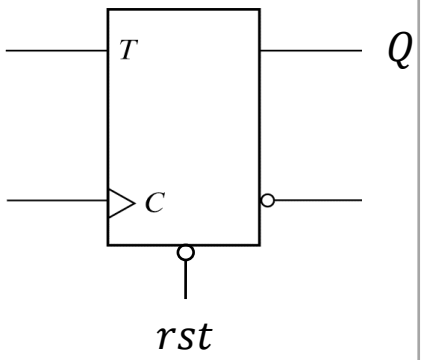
```
1  module JK(input  logic clk,
2            input  logic J, K,
3            output logic Q, notQ );
4
5      assign notQ = ~Q;
6
7      always_ff @(posedge clk)
8      // always @(posedge clk)
9          case ({J, K})
10             2'b00: Q <= Q;
11             2'b01: Q <= 1'b0;
12             2'b10: Q <= 1'b1;
13             2'b11: Q <= ~Q;
14         endcase
15 endmodule
```



J	K	$Q(t+1)$	
0	0	$Q(t)$	保持
0	1	**0**	置0
1	0	**1**	置1
1	1	$\overline{Q}(t)$	取反

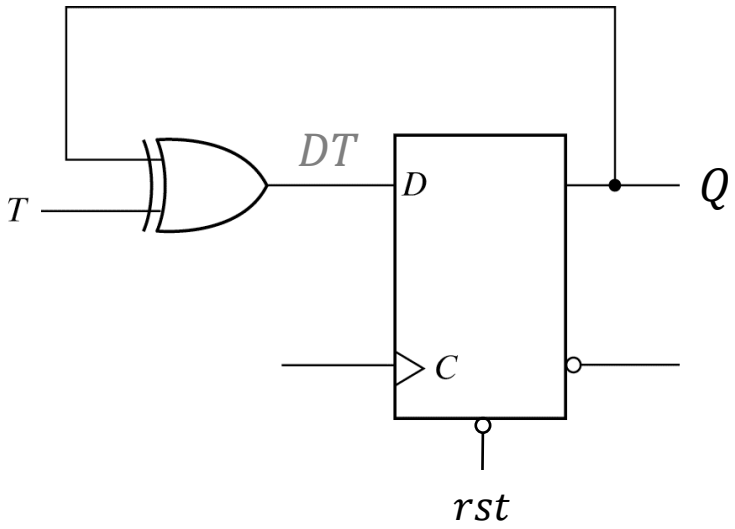
带异步reset的上边沿触发的T触发器

```
1  // T触发器
2  module T(
3      input  logic clk, rst, T,
4      output logic Q );// reg Q
5
6      logic DT;    //wire DT;
7
8      assign DT = Q ^ T;
9
10     // 实例化 D触发器
11     DFF D1(clk, rst, DT, Q);
12 endmodule
13
14 module DFF(
15     input  logic clk, rst, D,
16     output logic Q);//reg Q
17
18     always_ff @(posedge clk, negedge rst)
19         if(~rst) Q <= 1'b0;
20         else     Q <= D;
21 endmodule
```



下降沿触发

异步rst
放在敏感列表中



同步、异步参见教材P117 HDL例4.18

```
14 module DFF(
15     input  logic clk, rst, D,
16     output logic Q);//reg Q
17     always_ff @(posedge clk)
18         if(~rst) Q <= 1'b0;
19         else     Q <= D;
20 endmodule
```

同步rst,
与clk同步