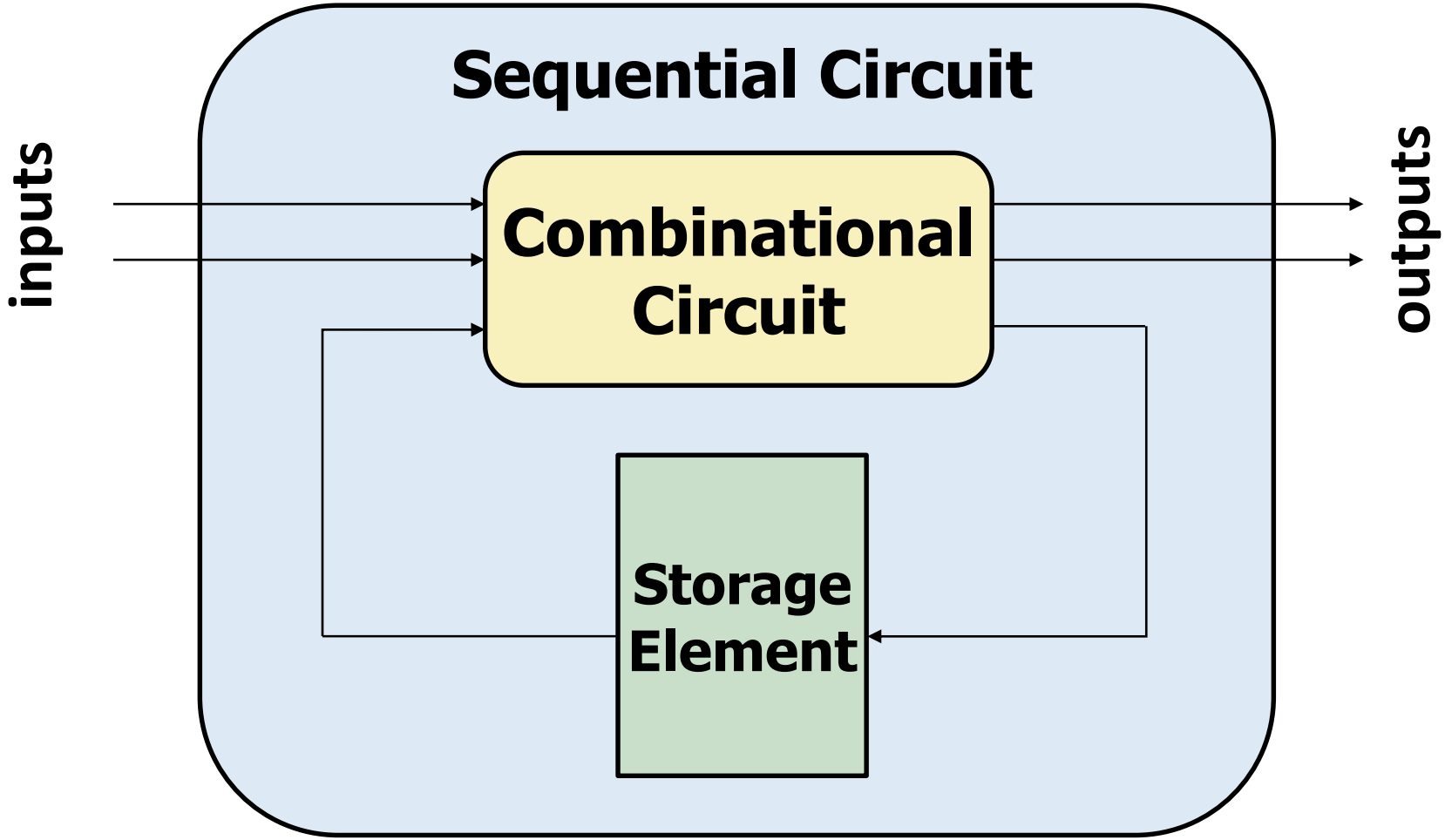
System Verilog入门（三）

实验目的

- 时序逻辑

时序逻辑电路描述


Sequential Circuit
inputs
Combinational Circuit
outputs
Storage Element

时序逻辑

- 定义具有记忆单元的模块
  - 锁存器latch，触发器flipflop，状态机FSM
- 时序逻辑通过时钟来“触发”
  - 锁存器电平触发
  - 触发器时钟边沿触发，我们仅使用边沿触发
- 时序逻辑编程需要用到
  - always过程语句
  - posedge/negedge触发条件

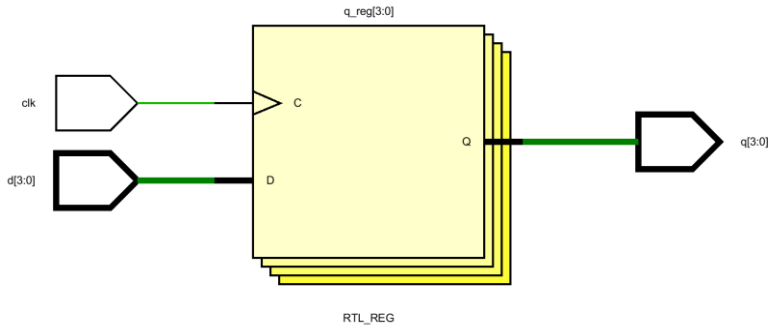
always过程块

```
always @ (sensitivity list)
      statement;
```

每当敏感度列表中的事件发生时。
语句就会被执行

基于触发器的寄存器

```
module flop(input              clk,
            input      [3:0] d,
            output reg [3:0] q);

  always_ff @ (posedge clk)
    q <= d;                    // pronounced “q gets d”

endmodule
```



- posedge 在时钟上升沿响应（处在敏感列表中）
- always过程块内部的语句会在时钟上升沿的时候得到执行
- 时钟上升沿来临时，d被拷贝入q

基于触发器的寄存器

```
module flop(input              clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                  // pronounced “q gets d”

endmodule
```

- assign 语句不会在always过程块内部使用
- <= 非阻塞赋值

基于触发器的寄存器

```
module flop(input             clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                  // pronounced “q gets d”

endmodule
```

- 赋值变量需要被声明为reg
- reg这个名字不一定意味着这个值是一个寄存器（可以是，但不一定是）。

过程赋值语句

过程赋值语句多用于对reg型变量进行赋值。

（1）非阻塞（non_blocking）赋值方式

赋值符号为”<=“，比如：b<=a; 非阻塞赋值在整个过程块结束时才完成赋值操作，即b的值并不是立刻就改变的。

（2）阻塞（blocking）赋值方式

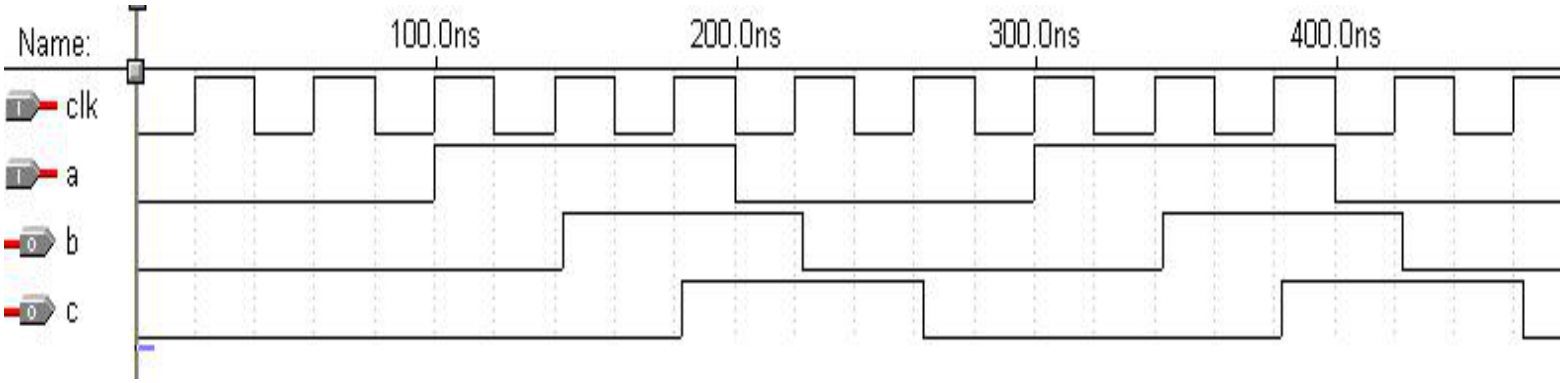
赋值符号为”=”，如：b=a; 阻塞赋值语句在该语句结束时就立即完成赋值操作，即b的值在该条语句结束后立刻改变。如果在一个块语句中，有多条阻塞赋值语句，那么在前面的赋值语句没有完成之前，后面的语句不能被执行，仿佛被阻塞了（blocking）一样，因此被称为是阻塞赋值方式。

阻塞赋值与非阻塞赋值

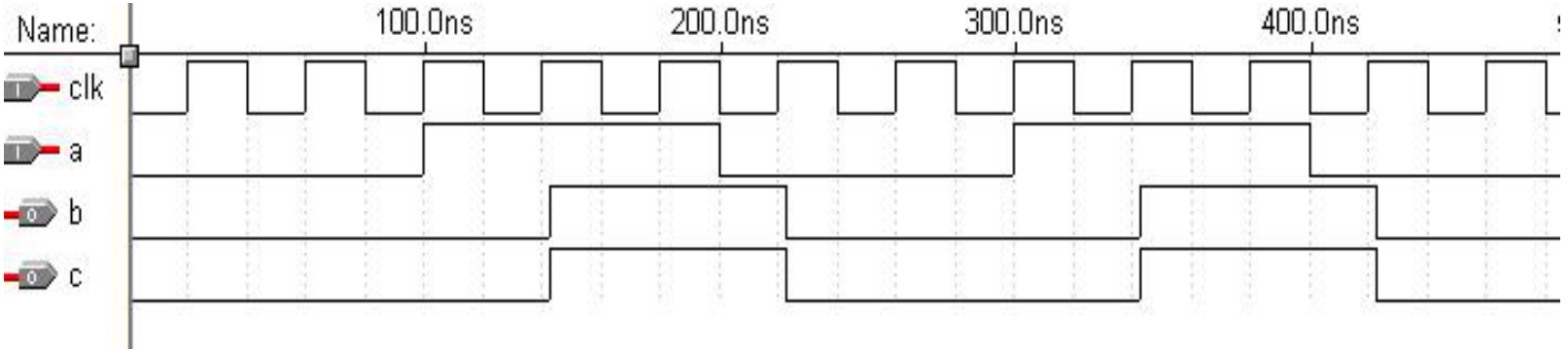
```
module non_block(c,b,a,clk);
  output c,b;
  input clk,a;
  reg c,b;

  always_ff @(posedge clk)
  begin
    b<=a;
    c<=b;
  end
endmodule
```

```
module block(c,b,a,clk);
  output c,b;
  input clk,a;
  reg c,b;

  always_ff @(posedge clk)
  begin
    b=a;
    c=b;
  end
endmodule
```

阻塞赋值与非阻塞赋值的仿真波形



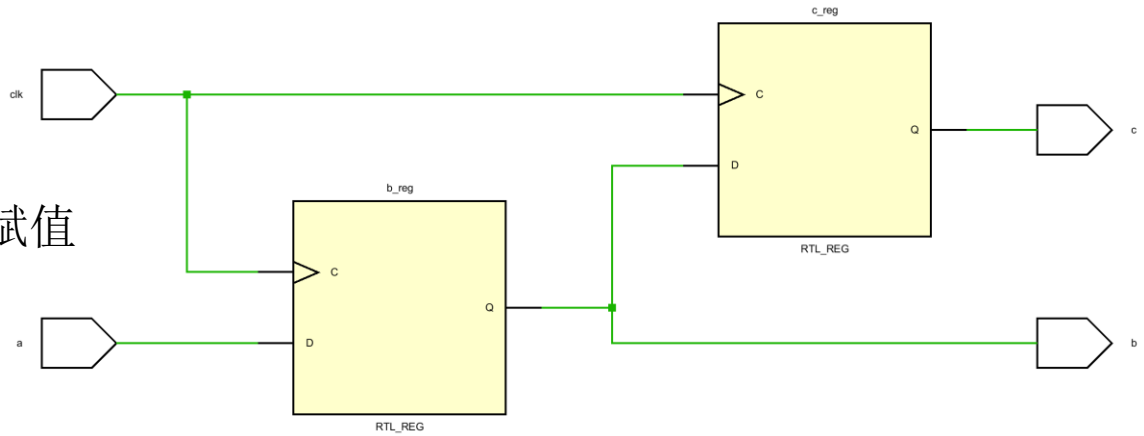
非阻塞赋值仿真波形图



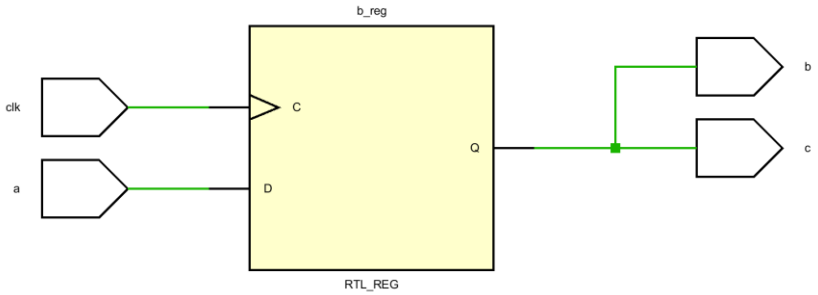
阻塞赋值仿真波形图

阻塞赋值和非阻塞赋值电路

非阻塞赋值



阻塞赋值综合结果



赋值语句的遵从规则

- 在时序逻辑上使用非阻塞赋值
- 在组合逻辑上使用阻塞赋值
- 在组合逻辑上也可以使用连续赋值语句assign

- 在always过程块中使用阻塞赋值和非阻塞赋值需要仔细考虑不同赋值的特征

顺序执行与并发执行

- 两个或者多个always过程块，assign持续赋值语句，实例元件调用等操作都是同时执行的。
- 在always模块内部，其语句如果是非阻塞赋值，也是并发执行的；而如果是阻塞赋值，则语句是按照指定的顺序执行的，语句的书写顺序对程序执行结果有着直接的影响。

顺序执行的例子

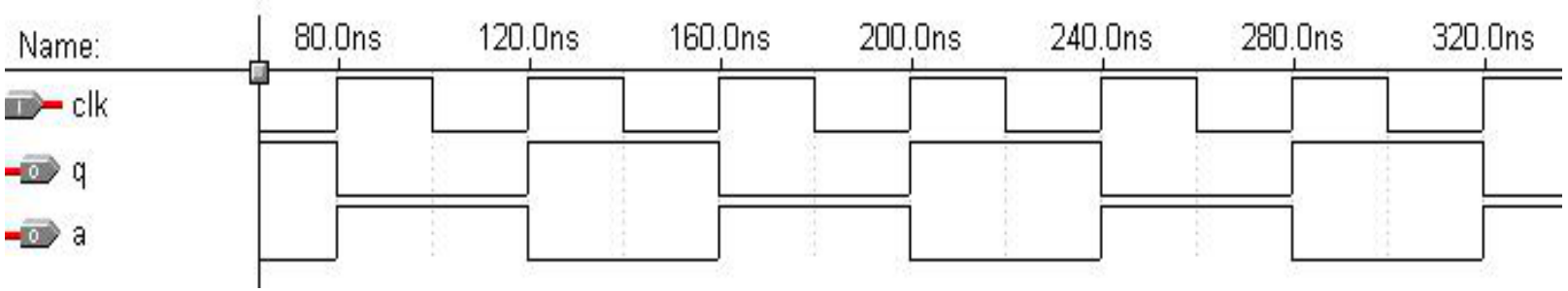
- 顺序执行模块1

```
module serial1(q,a,clk);
output q,a;
input clk;
reg q,a;
always_ff @(posedge clk)
   begin
      q=~q;
      a=~q;
   end
endmodule
```

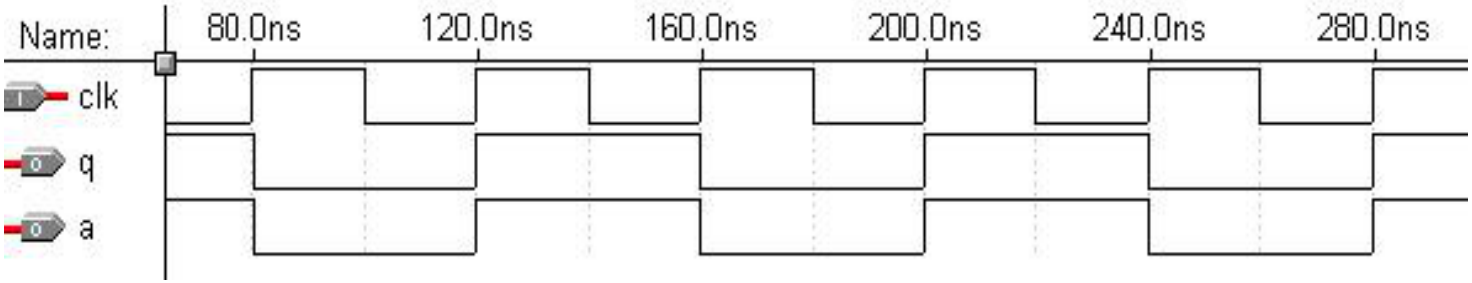
- 顺序执行模块2

```
module serial2(q,a,clk);
output q,a;
input clk;
reg q,a;
always_ff@(posedge clk)
   begin
      a=~q;
      q=~q;
   end
endmodule
```

顺序执行的时序效果

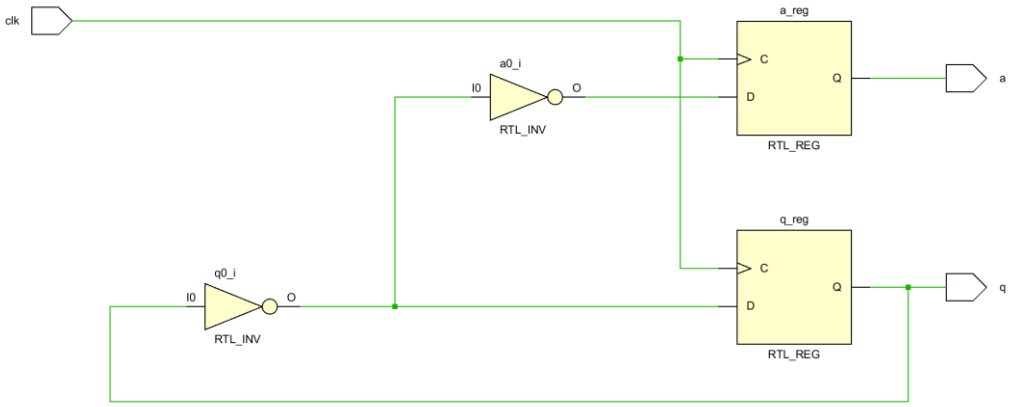


顺序执行模块1仿真波形图

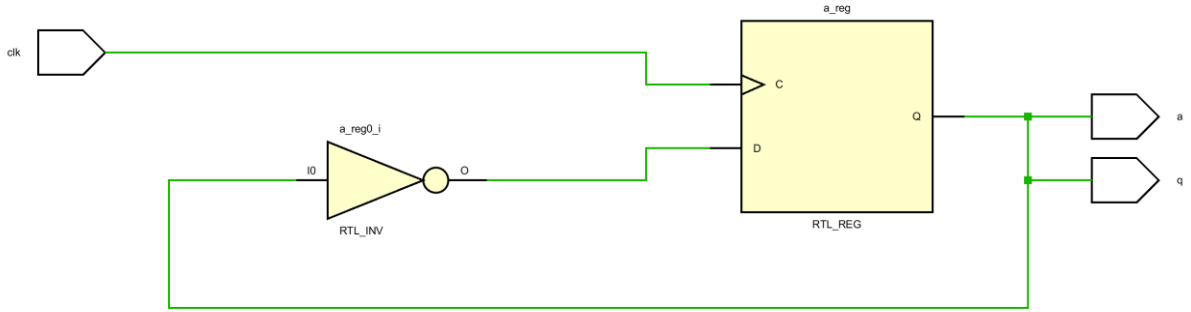


顺序执行模块2仿真波形图

顺序执行模块的综合结果



顺序执行模块1综合结果



顺序执行模块2综合结果

同步复位与异步复位

- 复位信号用于将硬件初始化到一个已知状态
  - 通常在系统启动时激活（上电时）
- **异步复位**
  - 复位信号的采样独立于时钟
  - 复位信号具有最高的优先权
  - 对毛刺敏感，可能会有亚稳态的问题
- **同步复位**
  - **复位信号是相对于时钟进行采样的**
  - 复位应该有足够长的激活时间，以便在时钟边沿采样成功。
  - 完全同步电路的结果

异步复位

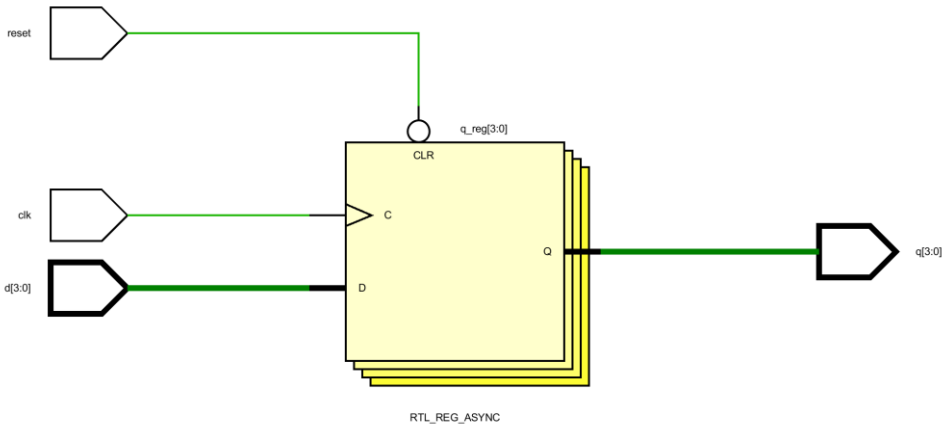
```
module flop_ar (input             clk,
                input             reset,
                input      [3:0] d,
                output reg [3:0] q);

  always_ff @ (posedge clk, negedge reset)
    begin
       if (reset == 0) q <= 0;   // when reset
       else            q <= d;   // when clk
    end
endmodule
```

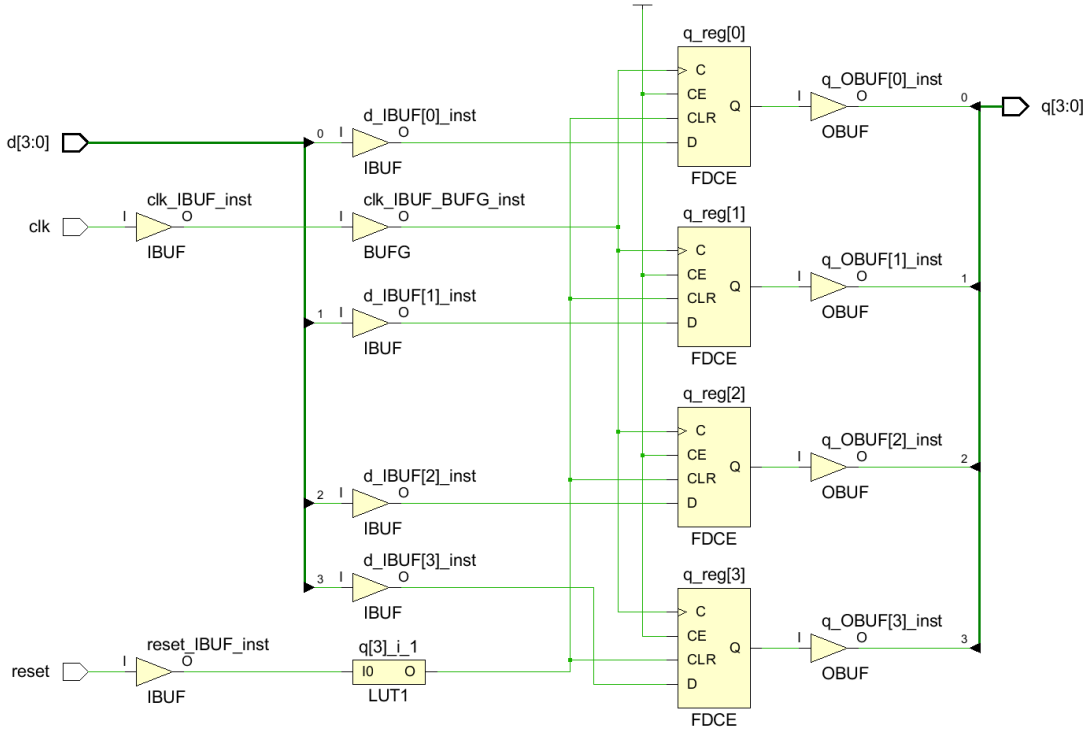
- 在这个例子中：有两个事件可以触发这个过程语句
  - 在clk上有一个上升沿
  - 复位时的下降沿

异步复位

```
module flop_ar (input             clk,
                input             reset,
                input      [3:0] d,
                output reg [3:0] q);

  always_ff @ (posedge clk, negedge reset)
    begin
      if (reset == 0) q <= 0;    // when reset
      else            q <= d;    // when clk
    end
endmodule
```

- 首先检查复位：如果复位为0，q被设置为0。
  - 这是一个异步复位，因为复位可以独立于时钟发生（在复位信号的负边沿）
- 如果没有复位，那么常规赋值就会生效

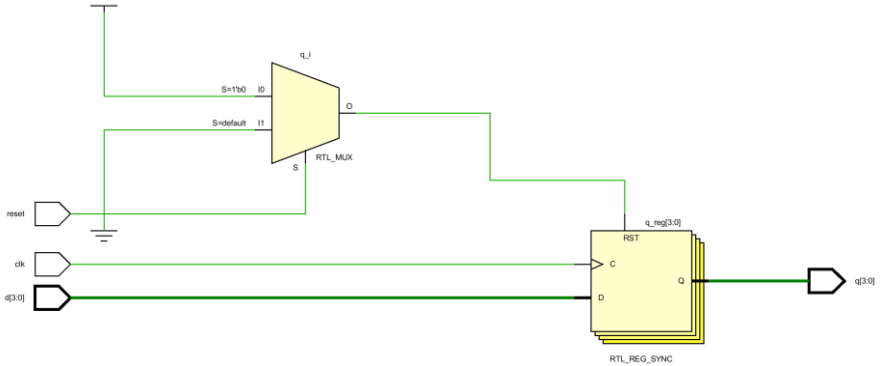

reset
q_reg[3:0]
CLR
clk
C
Q
q[3:0]
d[3:0]
D
RTL_REG_ASYNC

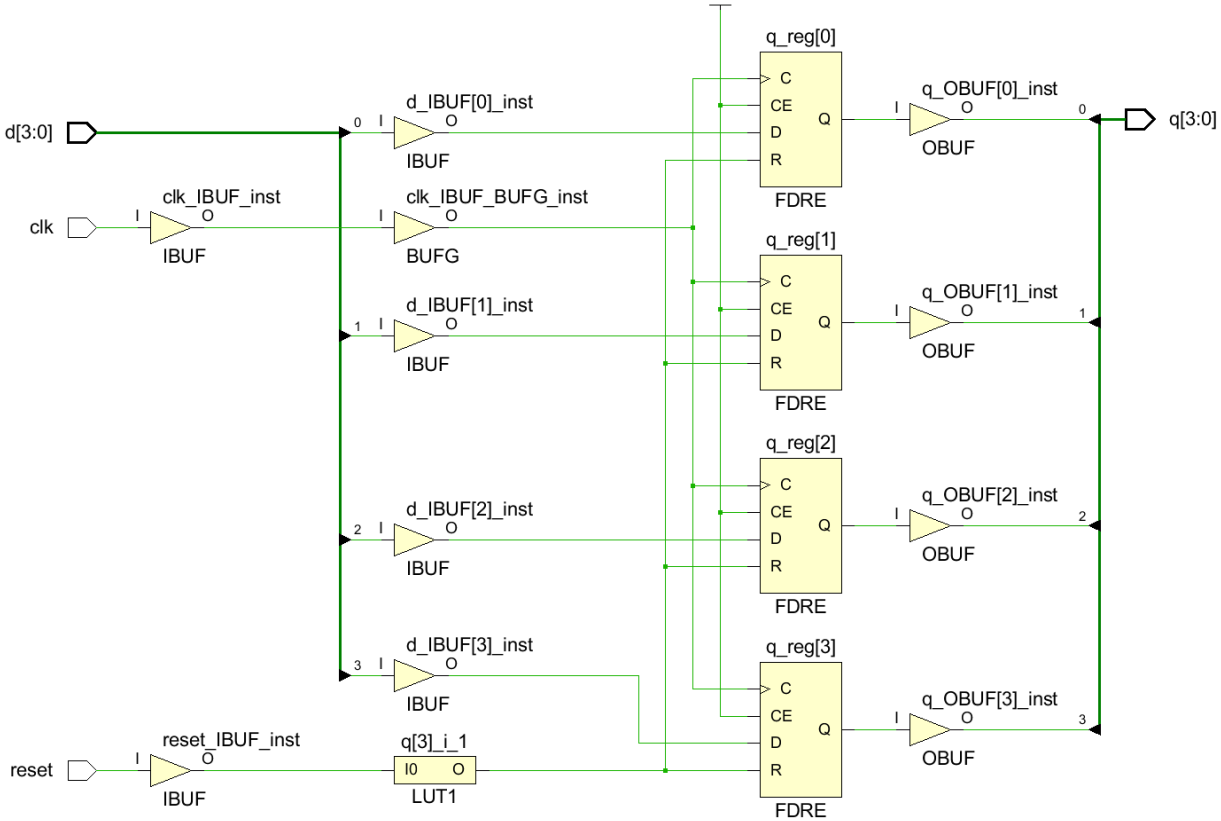

d[3:0]
clk
reset
d_IBUF[0]_inst
IBUF
clk_IBUF_inst
clk_IBUF_BUFG_inst
BUFG
d_IBUF[1]_inst
d_IBUF[2]_inst
d_IBUF[3]_inst
reset_IBUF_inst
q[3]_i_1
LUT1
q_reg[0]
q_reg[1]
q_reg[2]
q_reg[3]
C
CE
CLR
D
Q
FDCE
q_OBUF[0]_inst
q_OBUF[1]_inst
q_OBUF[2]_inst
q_OBUF[3]_inst
OBUF
q[3:0]

同步复位

```
module flop_sr (input             clk,
                input             reset,
                input      [3:0] d,
                output reg [3:0] q);

  always_ff @ (posedge clk)
    begin
       if (reset == 0) q <= 0;   // when reset
       else            q <= d;   // when clk
    end
endmodule
```

- 该过程只对时钟正边沿敏感
  - 复位只在时钟上升时发生，这是一个同步复位


q_i
S=1'b0
I0
S=default
I1
O
RTL_MUX
S
reset
clk
d[3:0]
q_reg[3:0]
RST
C
D
Q
q[3:0]
RTL_REG_SYNC

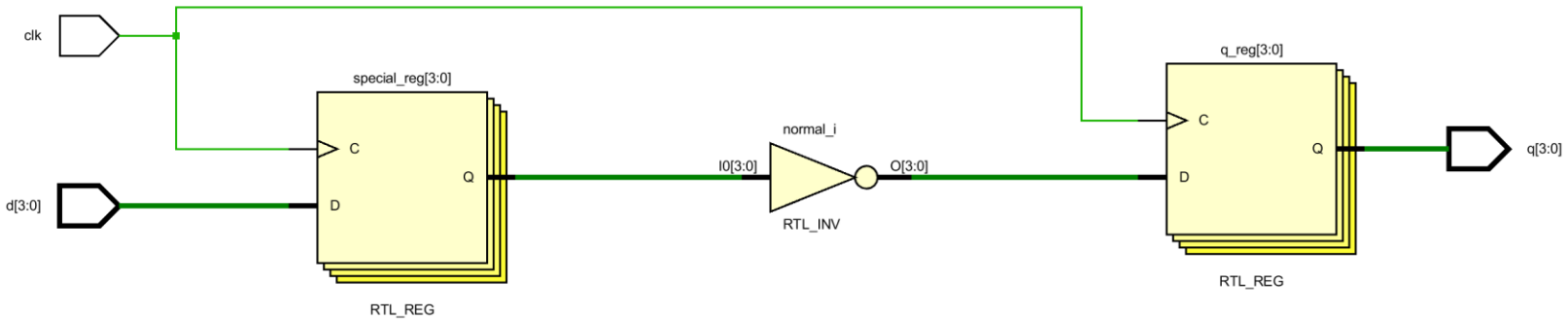

d[3:0]
clk
reset
d_IBUF[0]_inst
clk_IBUF_inst
clk_IBUF_BUFG_inst
d_IBUF[1]_inst
d_IBUF[2]_inst
d_IBUF[3]_inst
reset_IBUF_inst
q[3]_i_1
IBUF
BUFG
LUT1
I0
O
q_reg[0]
q_reg[1]
q_reg[2]
q_reg[3]
C
CE
D
R
Q
FDRE
q_OBUF[0]_inst
q_OBUF[1]_inst
q_OBUF[2]_inst
q_OBUF[3]_inst
OBUF
q[3:0]

always过程块

```
module example (input             clk,
                input      [3:0] d,
                output reg [3:0] q);

  wire [3:0] normal;          // standard wire
  reg  [3:0] special;         // assigned in always

  always_ff @ (posedge clk)
    special <= d;             // first FF array

  assign normal = ~special; // simple assignment

  always_ff @ (posedge clk)
    q <= normal;              // second FF array
endmodule
```

可以有多个always过程块

不允许在不同的always过程块中给相同的信号赋值
为什么？

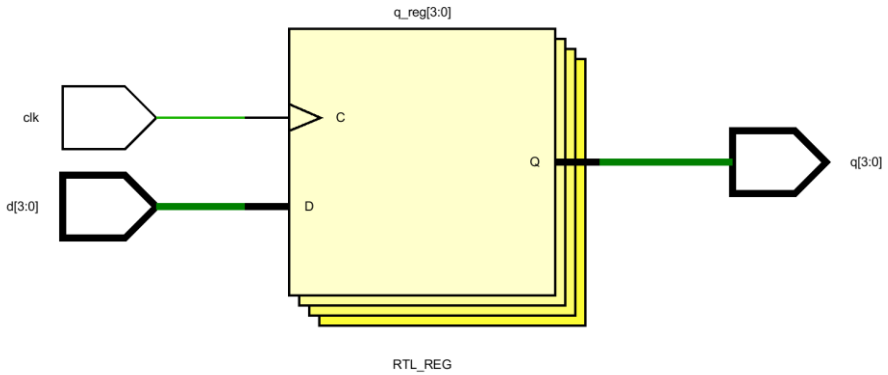

clk
d[3:0]
special_reg[3:0]
C
D
Q
RTL_REG
normal_i
I0[3:0]
O[3:0]
RTL_INV
q_reg[3:0]
C
D
Q
RTL_REG
q[3:0]

always过程块的记忆

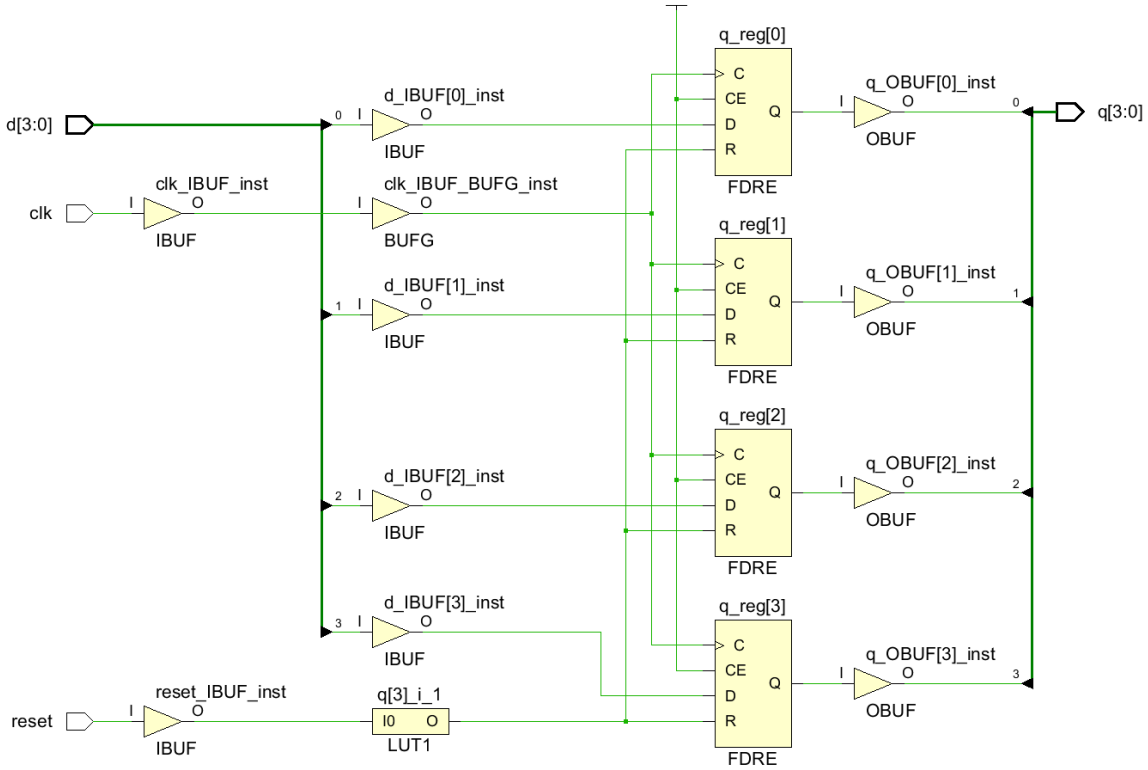
```
module flop (input             clk,
             input      [3:0] d,
             output reg [3:0] q);

  always @ (posedge clk)
    begin
        q <= d;   // when clk rises copy d to q
    end
endmodule
```

这个过程语句描述的是时钟正边沿时候的动作

如果时钟信号不是正边沿（高电平1，低电平0，或者负边沿）怎么办？
→ 保持不变：这就是过程块记忆实现时序逻辑



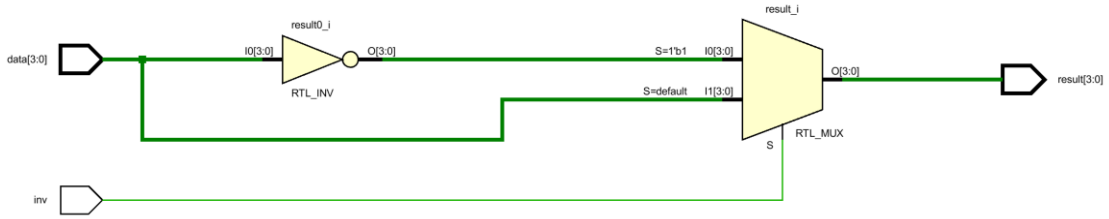

```
module flop (input            clk,
             input      [3:0] d,
             output reg [3:0] q);

  always_ff @ (posedge clk)
    begin
        q <= d    end
endmodule
```

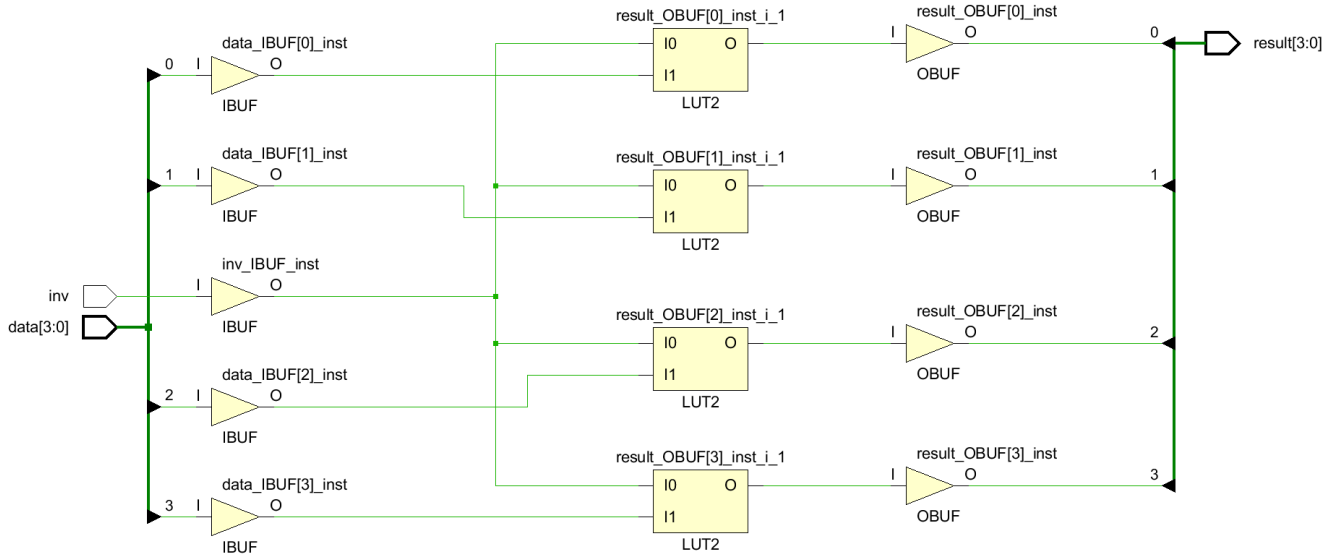


不带有记忆效应的always过程块

```
module comb (input             inv,
             input      [3:0] data,
             output reg [3:0] result);

  always @ (inv, data)       // trigger with inv, data
    if (inv) result <= ~data;// result is inverted data
    else     result <= data; // result is data

endmodule
```

上述过程块敏感信号是inv和data的电平，穷尽了所有的情况，没有记忆，是组合逻辑过程块



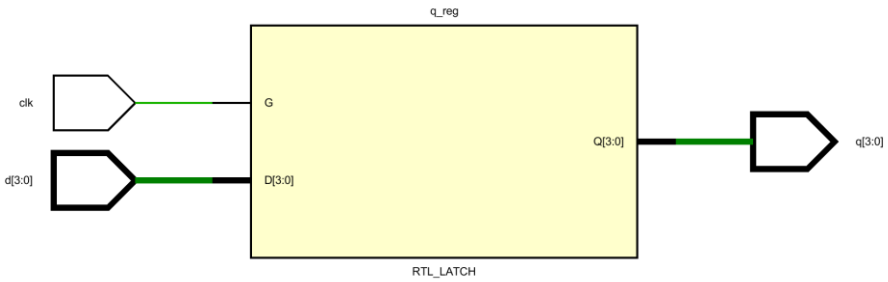


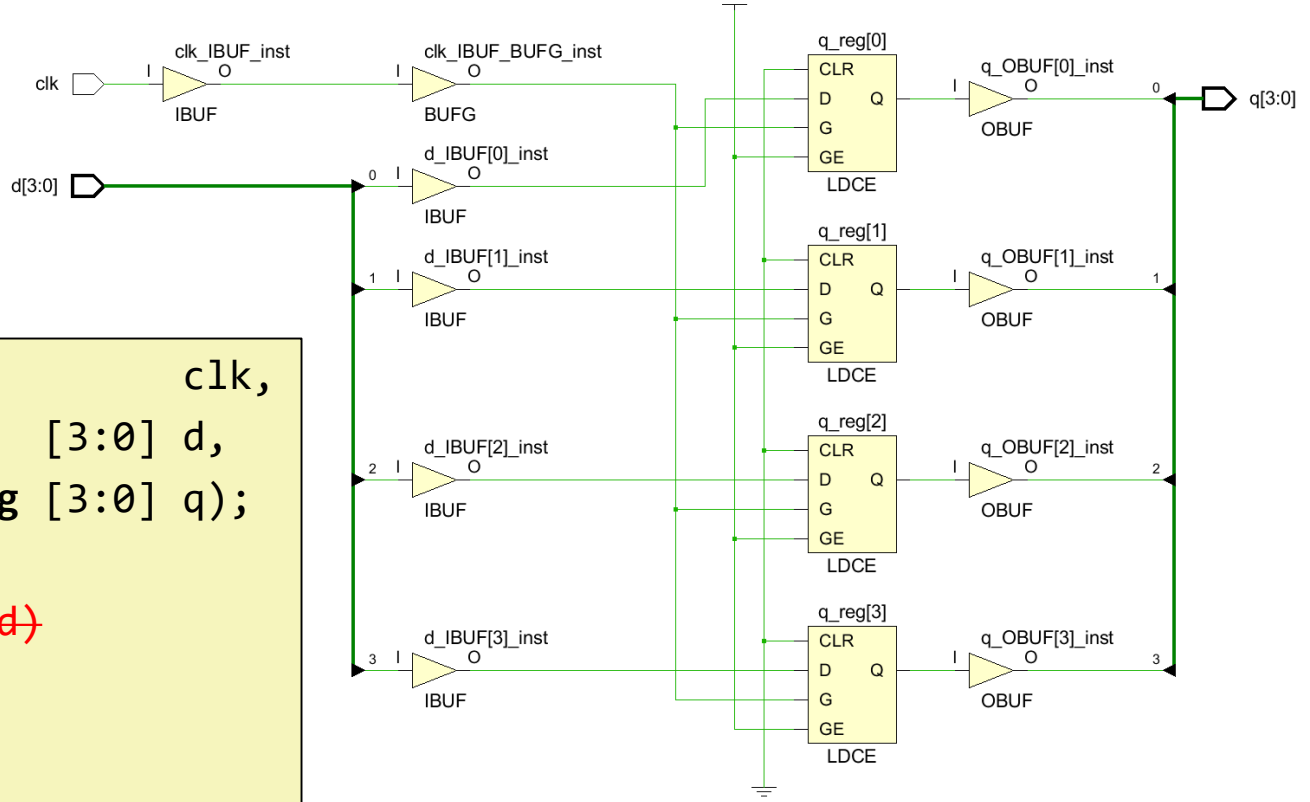

```
module comb (input             inv,
             input      [3:0] data,
             output reg [3:0] result);

  always_comb  @(inv, data)
    if (inv) result <= ~data
    else     result <= data;

endmodule
```

锁存器

```
module latch (input             clk,
              input      [3:0] d,
              output reg [3:0] q);

  always @ (clk, d)
    if (clk) q <= d;       // latch is transparent when
                           // clock is 1
endmodule
```

这是电平响应，只规定了clk高电平时候的动作
else部分缺失，代表这部分q值保持不变，被综合为锁存器






```
module latch (input             clk,
              input      [3:0] d,
              output reg [3:0] q);

  always_latch @ (clk, d)
    if (clk) q <= d;
endmodule
```

always的不同属性

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

```
module latch (input  clk,
              input [3:0] d,
          output reg[3:0] q);
  always_latch @ (clk, d)
    if (clk) q <= d;
endmodule
```

```
module comb (input             inv,
             input       [3:0] data,
             output reg [3:0] result);
  always_comb @ (inv, data)          // 没有敏感信号列表
    if (inv) result <= ~data;
    else     result <= data;
endmodule
```

- 在SystemVerilog中指定过程块的属性可以让系统帮助我们检查，预防出错
- 实验中不要出现latch，也不要同时响应时钟的上边沿和下边沿（FPGA中无此器件）

使用always会不小心生成锁存器

```
wire enable, data;
reg out_a, out_b;

always @ (*) begin
        out_a = 1'b0;
        if(enable) begin
                out_a = data;
                out_b = data;
        end
end
```

No assignment for ~enable

Sequential

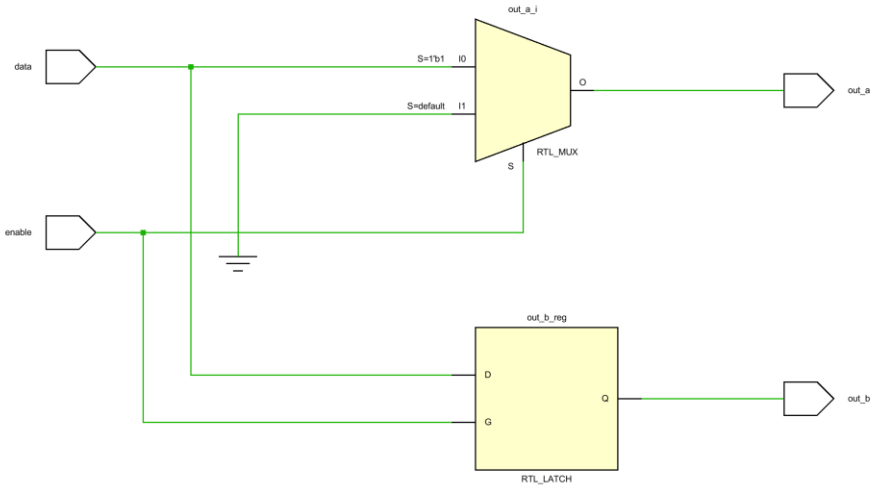
```
wire enable, data;
reg out_a, out_b;

always @ (data) begin
        out_a = 1'b0;
        out_b = 1'b0;
        if(enable) begin
                out_a = data;
                out_b = data;
        end
end
```

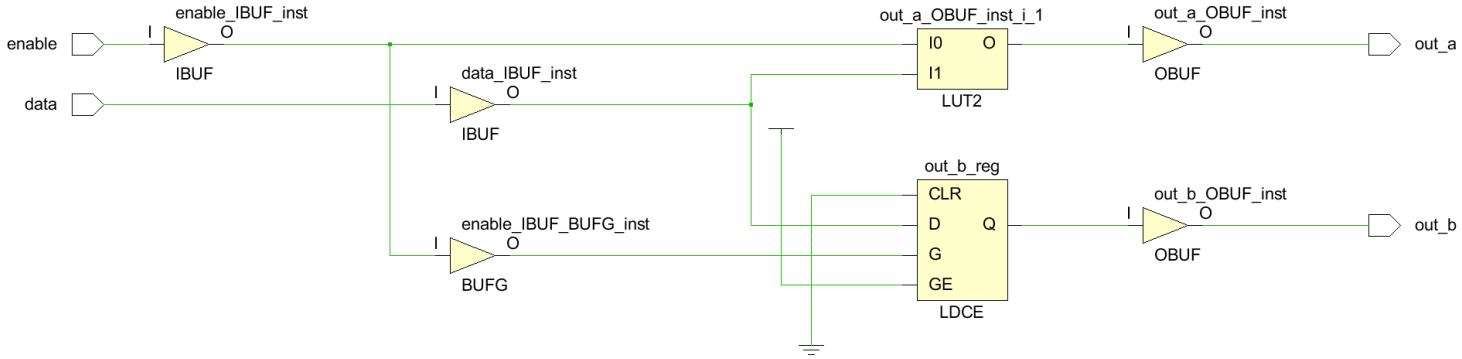
Not in the sensitivity list

Sequential

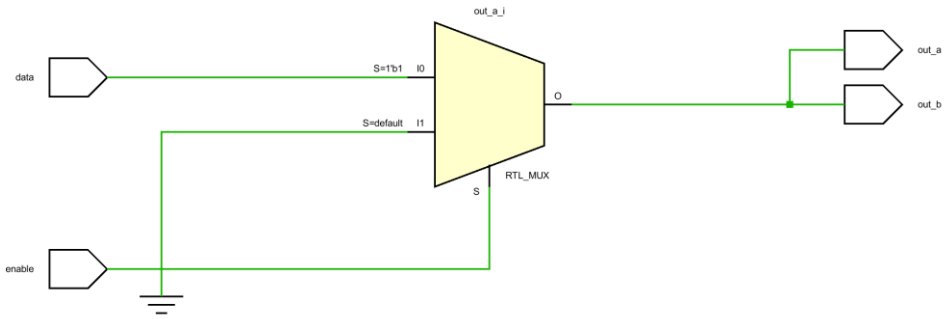
确定自己实现组合逻辑，请使用always_comb

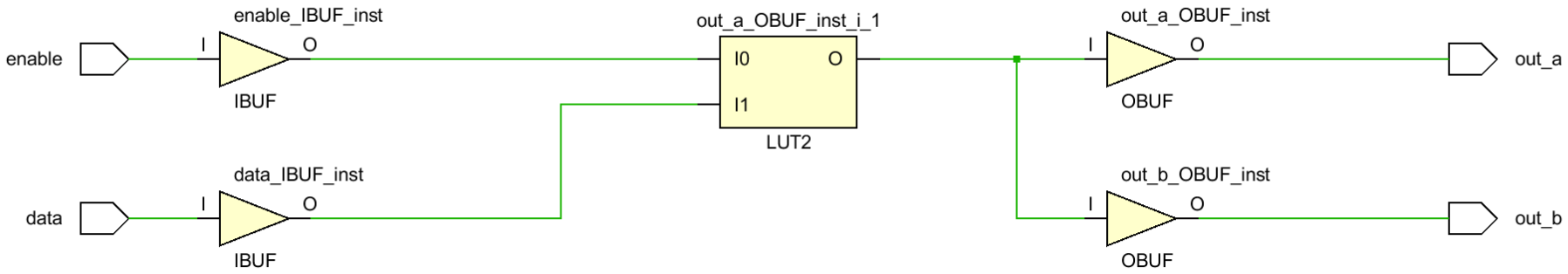
```
wire enable, data;
reg out_a, out_b;

always @ (*) begin
        out_a = 1'b0;
        if(enable) begin
                out_a = data;
                out_b = data;
        end
end
```





```
wire enable, data;
reg out_a, out_b;

always @ (data) begin
        out_a = 1'b0;
        out_b = 1'b0;
        if(enable) begin
                out_a = data;
                out_b = data;
        end
end
```



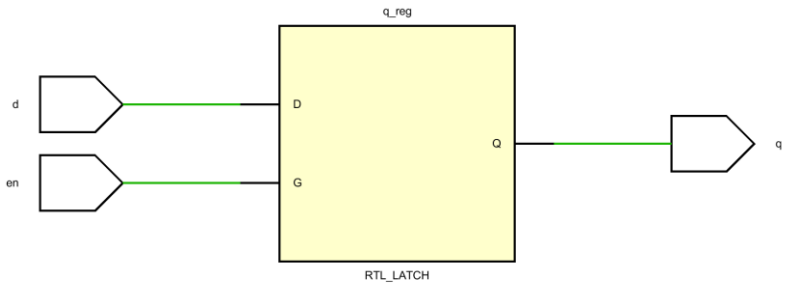


always_comb

- 没有敏感信号表
- 允许仿真工具检查正确的组合逻辑代码风格

```
always_comb
   q = d;

always_comb
   q <= d;
```

```
always_comb
    if(en) q<=d;

always_comb
    if(en) q=d;
```



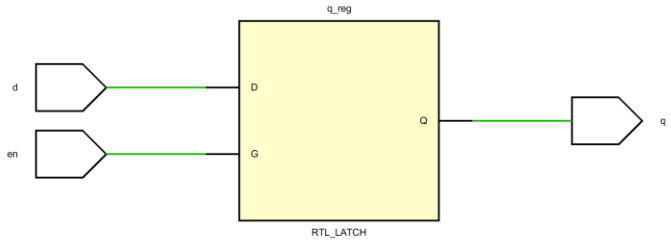
vivado编译不会报错！！

always_latch

- 没有敏感信号表
- 允许仿真工具检查正确的锁存器逻辑代码风格

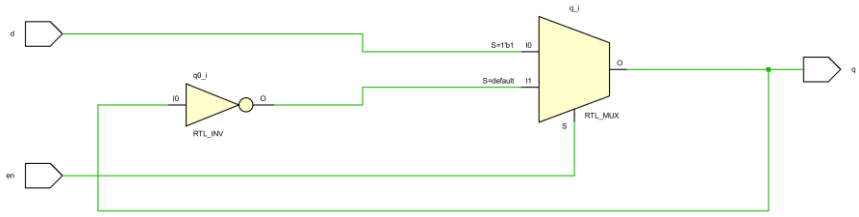
```
always_latch
   if(en) q<=0;
```

```
always_latch
   if(en) q=0;
```




```
always_latch
    if(en) q<=d;
    else q<=~q;
```

```
always_latch
    if(en) q=d;
    else q=~q;
```

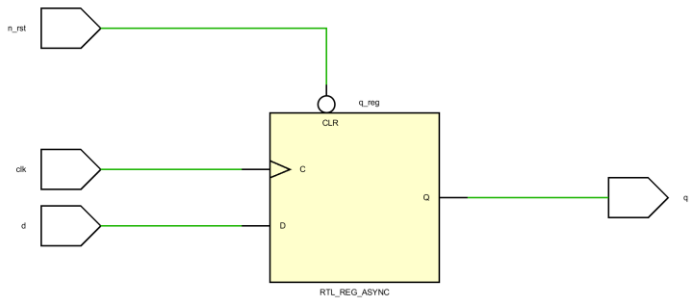


vivado编译不会报错！！

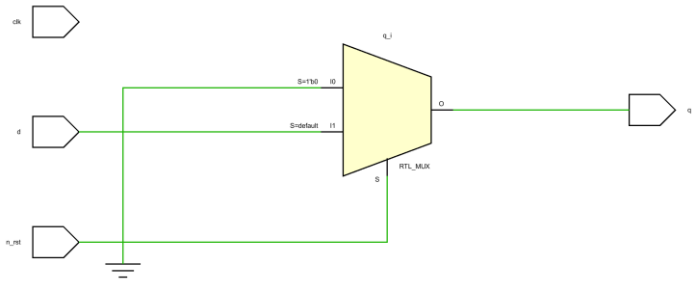
always_ff

- 允许仿真工具检查正确的寄存器逻辑代码风格

```
always_ff @(posedge clk,
negedge n_rst)
    if(~n_rst)
       q <= '0;
    else
       q<=d;
```




```
always_ff @(clk, n_rst)
    if(~n_rst)
       q <= '0;
    else
       q<=d;
```



可以编译，但不能生成触发器

case语句

```
module sevensegment (input      [3:0] data,
                     output reg [6:0] segments);

  always @ ( * )                    // * is short for all signals
  always_comb
    case (data)                     // case statement
      4'd0: segments = 7'b111_1110;  // when data is 0
      4'd1: segments = 7'b011_0000;  // when data is 1
      4'd2: segments = 7'b110_1101;
      4'd3: segments = 7'b111_1001;
      4'd4: segments = 7'b011_0011;
      4'd5: segments = 7'b101_1011;
      // etc etc
      default: segments = 7'b000_0000; // required
    endcase

endmodule
```

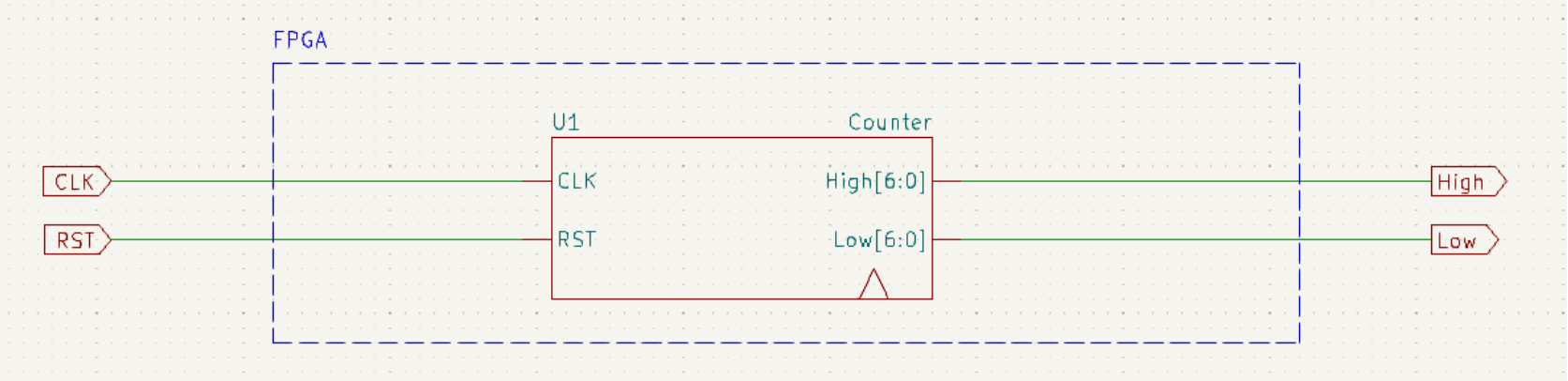
casex，casez使用通配符

```
always_comb
  begin
    {int2, int1, int0} = 3'b000;
    casez (irq)
      3'b1?? : int2 = 1'b1;
      3'b?1? : int1 = 1'b1;
      3'b??1 : int0 = 1'b1;
      default: {int2, int1, int0} = 3'b000;
    endcase
end
```

代码中禁止使用casex，通配的话用casez就好

实验内容

- 设计 00 ~ 59 的计数器，手动输入时钟，时钟上升沿计数一次，当计数到59后，计数回到 00， 重新开始计数，复位按键下降沿可以随时复位到00的状态。
- 使用时钟模块上的复位开关 CLK 作为手动时钟输入，复位开关 RST 作为复位输入。
- 使用不带译码的七段数码管模块的两个七段数码管分别作为计数器高低两位的输出 High[6:0] 和 Low[6:0]。



提高要求

- 使用时钟模块上的 1M 时钟，将计数器改成秒表；
- 使用开关模块上的拨动开光控制秒表启动/暂停。