SystemVerilog

实验4

组合逻辑电路的**行为级建模**

在功能和算法层次上描述电路

xgsun@fudan.edu.cn

孙晓光

2024-9-22

行为级建模

- 用于描述一个电路输入、输出的行为（逻辑功能），而不是怎样去实现硬件电路。
- 一般使用 always 结构
  - **过程赋值语句**
  - case 语句
  - if 语句
  - for 语句

```
always_comb
    if(En==1) y = 1;
    else      y = 0;
```

1为真；0、x、z 为假

if (条件表达式) 为真时执行的语句;

if (条件表达式) 为真时执行的语句;
else 为假时执行的语句;

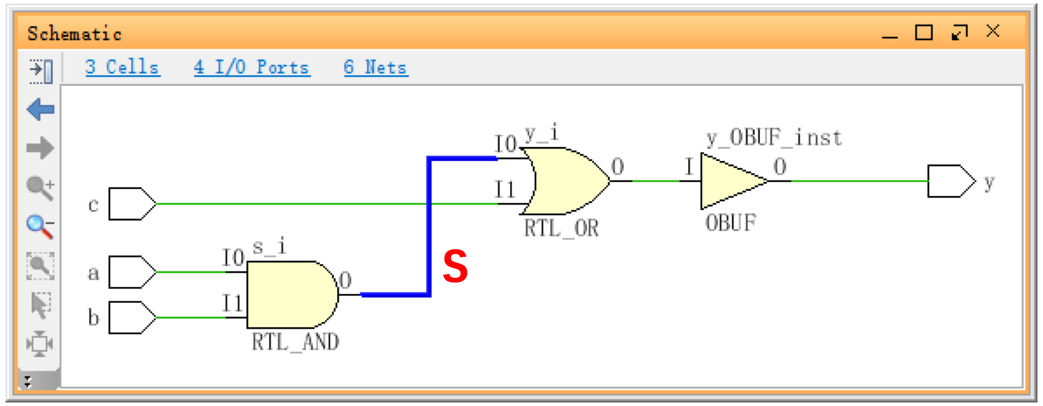
if (条件表达式1) 为真时执行的语句1;
else if (条件表达式2) 为真时执行的语句2;
……
else默认执行的语句;

隐含优先级

Example_assign

并行

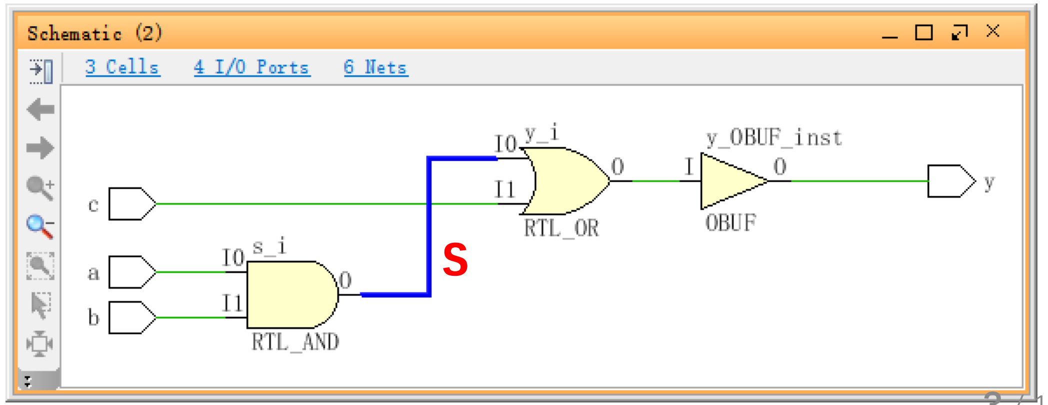
```
module Demo(
    input wire a, b, c,
    output wire y);

    wire s; // 内部连线
    // assign为并行语句,
    // 书写先后次序无关
    assign s = a & b;
    assign y = s | c;
endmodule
```



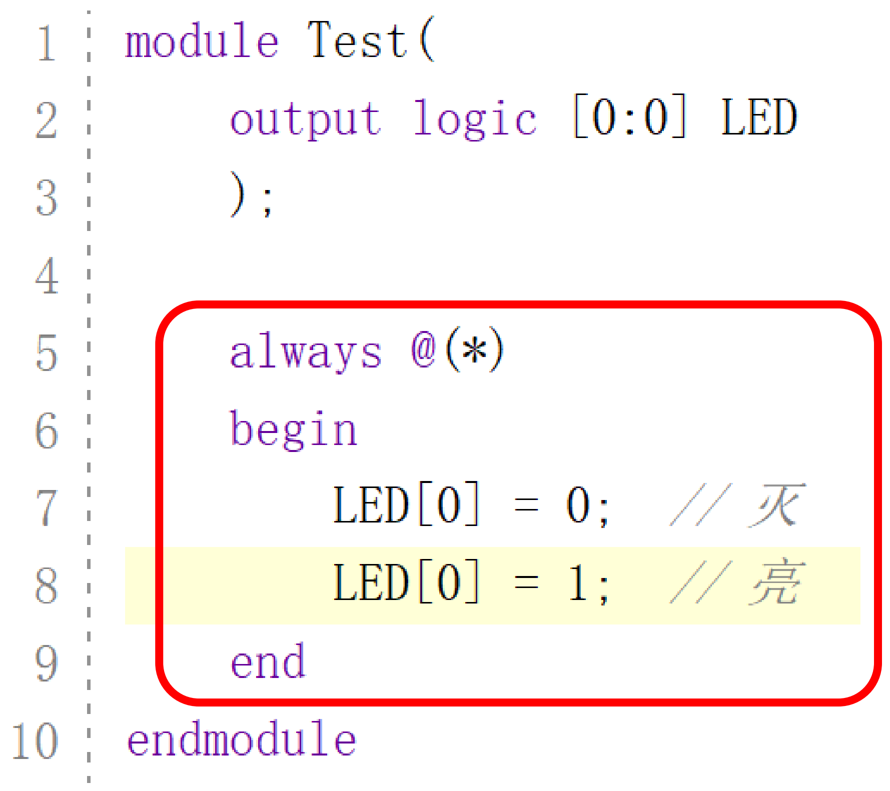
```
module Demo2(
    input wire a, b, c,
    output wire y);

    wire s; // 内部连线
    // assign为并行语句,
    // 书写先后次序无关
    assign y = s | c;
    assign s = a & b;
endmodule
```

||

实现的原理图一样



always 块



```
 1  module Test(
 2      output logic [0:0] LED
 3      );
 4
 5      always @(*)
 6      begin
 7          LED[0] = 0;   // 灭
 8          LED[0] = 1;   // 亮
 9      end
10  endmodule
```

- LED[0] 不断地重复：**灭**、**亮**?

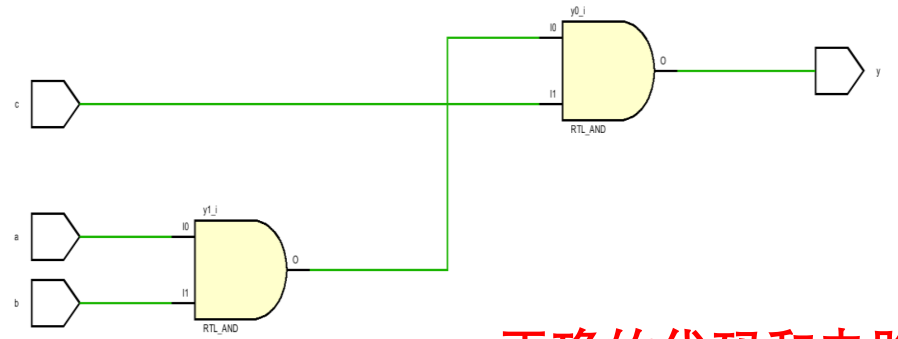
- 综合后，LED[0] 永远地**亮**，第7行将被完全忽略！

过程赋值语句 vs 连续赋值语句

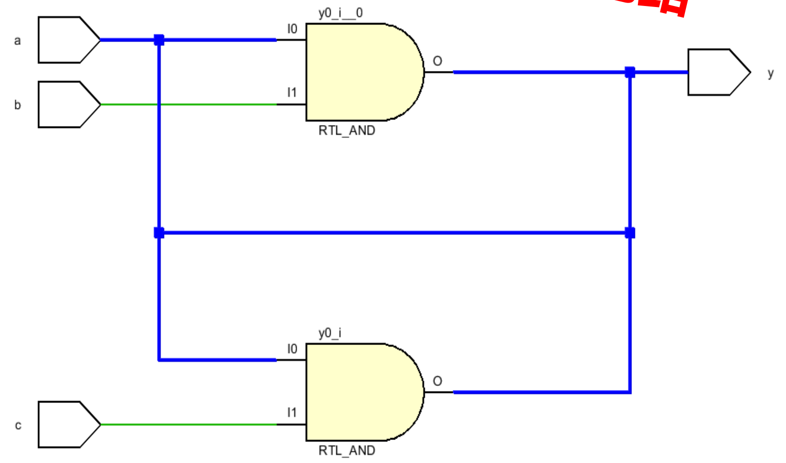
```
module and_block_assign(
    input a, b, c,
    output reg y);

    always @(*)
    begin
        y = a;
        y = y & b;
        y = y & c;
    end
endmodule
```

```
module and_continue_assign(
    input a, b, c,
    output y  );

    assign y = a;
    assign y = y & b;
    assign y = y & c;
endmodule
```

不正确的代码和电路



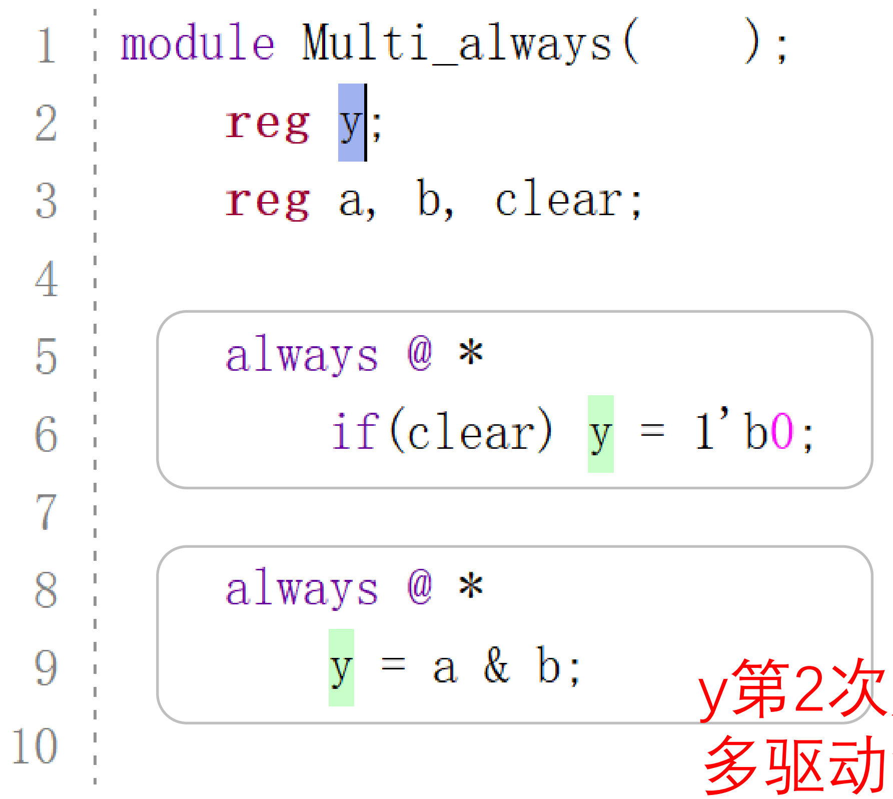
正确的代码和电路



```
Synthesis (3 critical warnings)
  [Synth 8-6859] multi-driven net on pin y_OBUF with 1st driver pin 'a_IBUF_inst/O' [and_continue_assign.sv:1]
```

组合电路代码中常见的错误

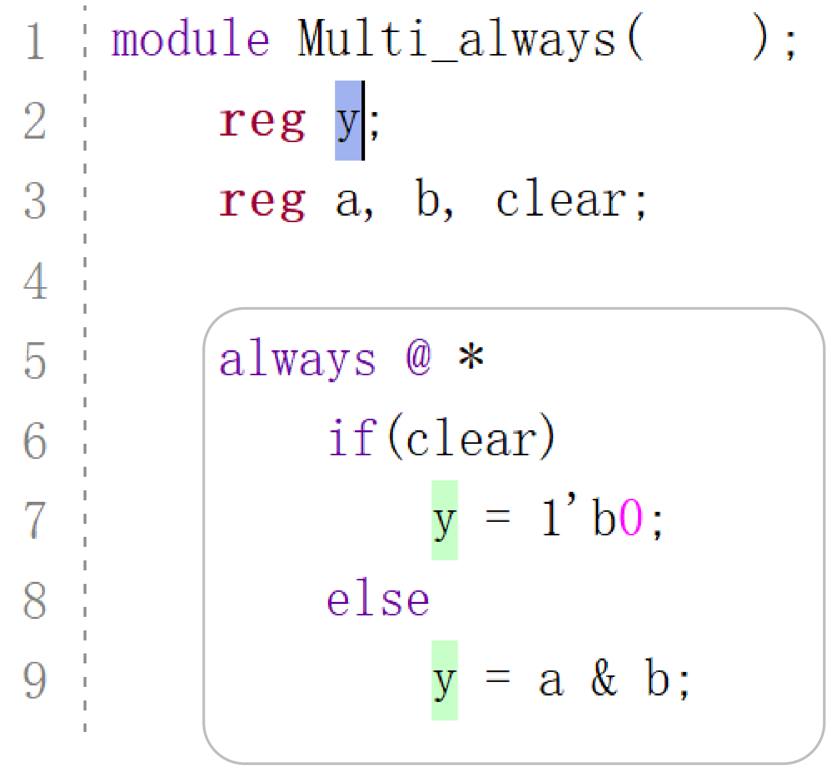
1）变量在多个always块中赋值



```
module Multi_always(    );
    reg y;
    reg a, b, clear;

    always @ *
        if(clear) y = 1'b0;

    always @ *
        y = a & b;

```

y第2次赋值，
多驱动错误

✗

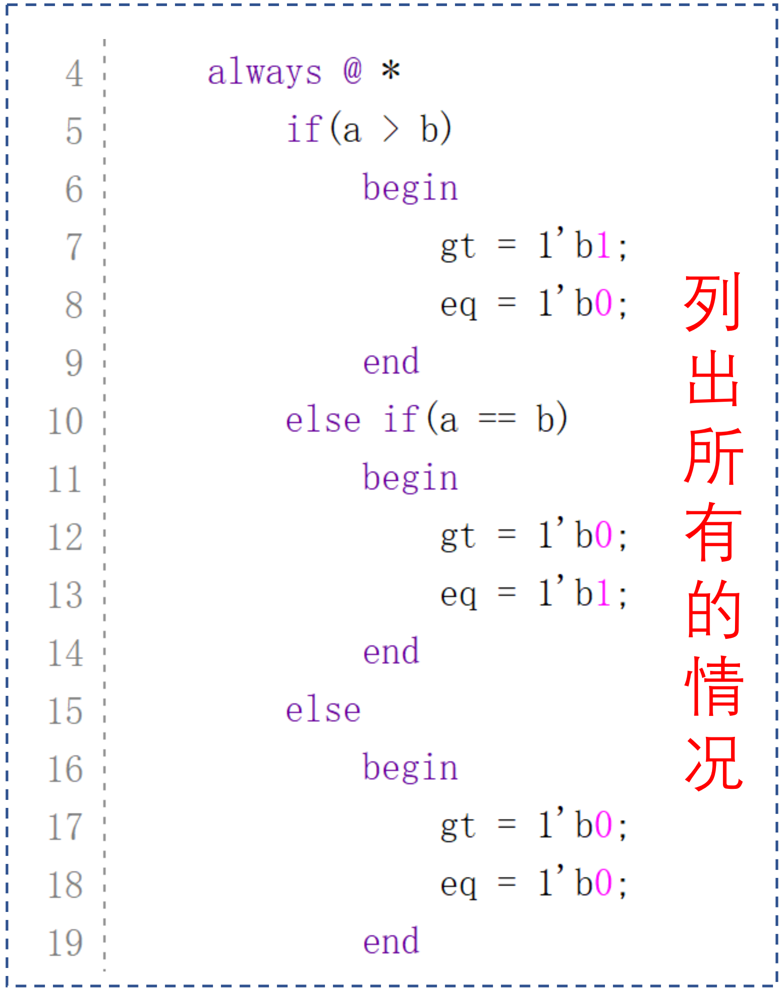


```
module Multi_always(    );
    reg y;
    reg a, b, clear;

    always @ *
        if(clear)
            y = 1'b0;
        else
            y = a & b;
```

✓

组合电路代码中常见的错误

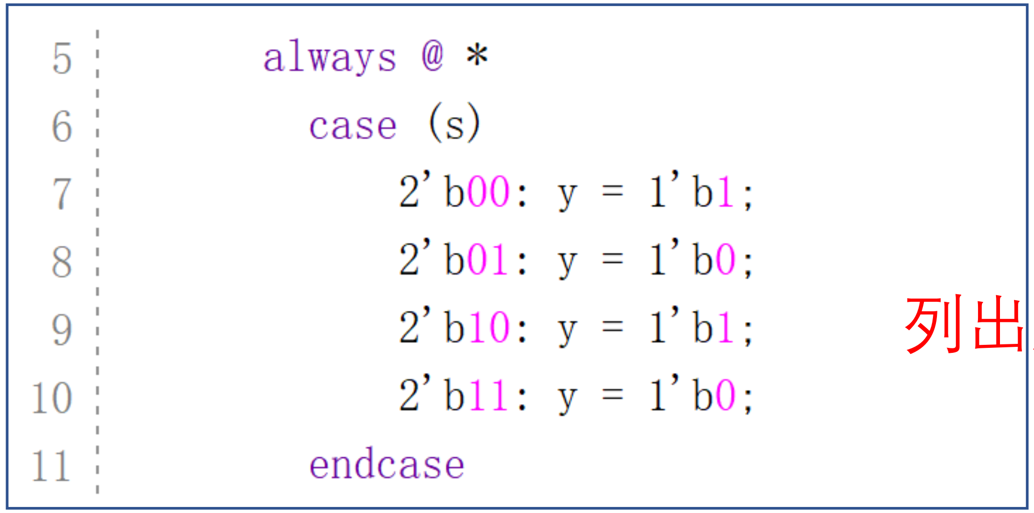
2）不完整的分支、不完整的输出赋值



```
always @ *
    if(a > b)
        begin
            gt = 1'b1;
            eq = 1'b0;
        end
    else if(a == b)
        begin
            gt = 1'b0;
            eq = 1'b1;
        end
    else
        begin
            gt = 1'b0;
            eq = 1'b0;
        end
```

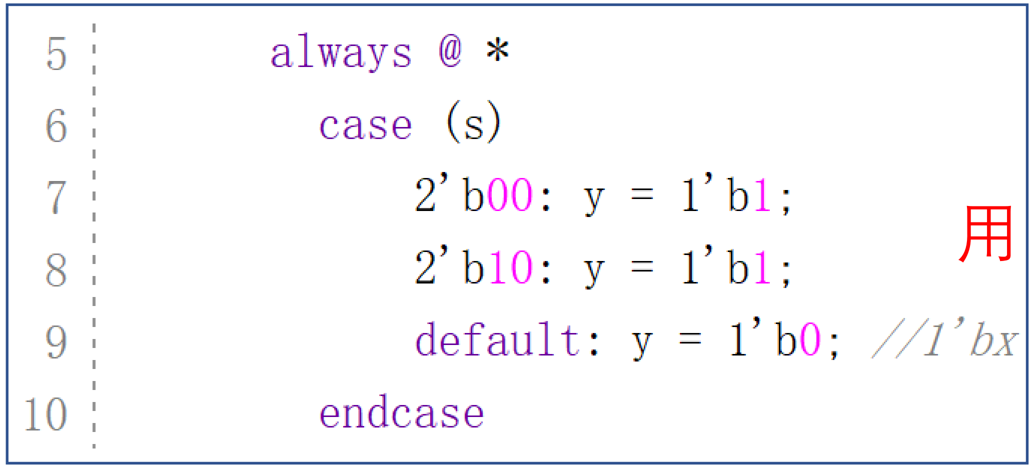
列出所有的情况

解决方案1：书写完整



```
always @ *
  case (s)
      2'b00: y = 1'b1;
      2'b01: y = 1'b0;
      2'b10: y = 1'b1;
      2'b11: y = 1'b0;
  endcase
```

列出所有的情况



```
always @ *
  case (s)
      2'b00: y = 1'b1;
      2'b10: y = 1'b1;
      default: y = 1'b0; //1'bx
  endcase
```

用 default 补全

组合电路代码中常见的错误

2）不完整的分支、不完整的输出赋值

变量如果没有赋值则保持原有的值，综合时将产生意外的存储器

解决方案2：在always起始部分为每个变量赋初值

```
always @ *
    begin
        gt = 1'b0;
        eq = 1'b0;
        if (a > b)
            gt = 1'b1;
        else if(a == b)
            eq = 1'b1;
    end
```

赋初值

```
always @ *
  begin
     y = 1'b0; // 赋初值
     case (s)
         2'b00: y = 1'b1;
         2'b10: y = 1'b1;
     endcase
  end
```

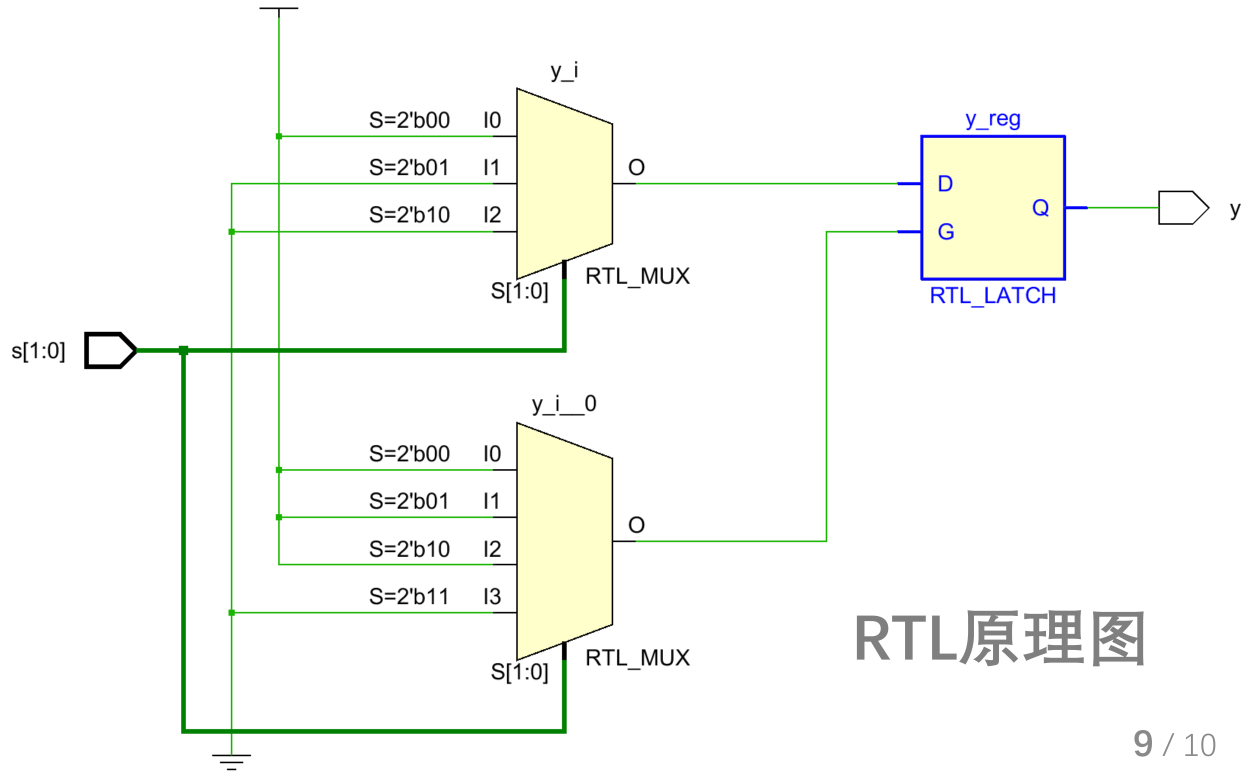
避免 **锁存器** (Latch) 的产生

一种在**异步**电路中存储信息的单元。

危害：对毛刺敏感、不能异步复位、上电后处于不定态、占用更多资源(FPGA)

产生：if、case不完整，输出变量赋值给自己。

```
// 输出变量赋值给自己
module ToMySelf(
    input  logic [1:0] s,
    output logic       y  );

    always_comb
        case (s)
            0: y = 1;
            1: y = 0;
            2: y = 0;
            3: y = y;
        endcase
endmodule
```



RTL原理图

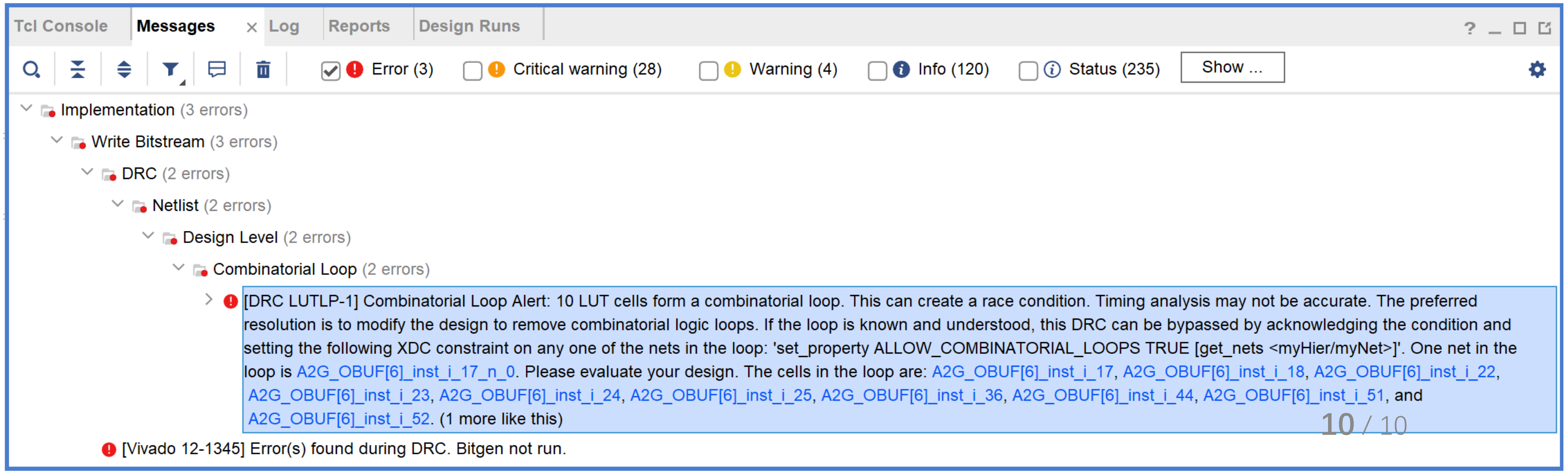
变量未初始化

```
// 8-bit binary-to-BCD converter: 移位加3算法
module Bin2BCD8bit(
    input  logic [7:0] b,
    output logic [9:0] p );

    // 中间变量
    logic [17:0] z;
    integer i;

    always_comb
    begin
//          for(i=0; i<=17; i=i+1)    z[i] = 0;
        z[10:3] = b; //左移3位

        repeat(5)    //重复5次
        begin
            // 如果个位大于4，则加3
            if(z[11:8] > 4) z[11:8]
            // 如果十位大于4，则加3
            if(z[15:12]> 4) z[15:12]
            // 左移1位
            z[17:1] = z[16:0];
        end
        p = z[17:8];
    end
endmodule
```

如果去掉for语句，

则没有给变量赋初值，

则产生错误信息！



```
Implementation (3 errors)
  Write Bitstream (3 errors)
    DRC (2 errors)
      Netlist (2 errors)
        Design Level (2 errors)
          Combinatorial Loop (2 errors)
            [DRC LUTLP-1] Combinatorial Loop Alert: 10 LUT cells form a combinatorial loop. This can create a race condition. Timing analysis may not be accurate. The preferred resolution is to modify the design to remove combinatorial logic loops. If the loop is known and understood, this DRC can be bypassed by acknowledging the condition and setting the following XDC constraint on any one of the nets in the loop: 'set_property ALLOW_COMBINATORIAL_LOOPS TRUE [get_nets <myHier/myNet>]'. One net in the loop is A2G_OBUF[6]_inst_i_17_n_0. Please evaluate your design. The cells in the loop are: A2G_OBUF[6]_inst_i_17, A2G_OBUF[6]_inst_i_18, A2G_OBUF[6]_inst_i_22, A2G_OBUF[6]_inst_i_23, A2G_OBUF[6]_inst_i_24, A2G_OBUF[6]_inst_i_25, A2G_OBUF[6]_inst_i_36, A2G_OBUF[6]_inst_i_44, A2G_OBUF[6]_inst_i_51, and A2G_OBUF[6]_inst_i_52. (1 more like this)
    [Vivado 12-1345] Error(s) found during DRC. Bitgen not run.
```