

Verilog组合逻辑电路与 时序逻辑电路

组合逻辑电路设计

- 数字逻辑中，组合逻辑是指在任意时刻的输出信号，仅与当时的输入信号有关。常用的组合逻辑电路包括编码器、译码器、数据选择器、数据分配器、数值比较器。

组合逻辑电路设计

- 数字系统中，常常需要将某一信息变换为某一特定的代码。把二进制代码按一定规律编码，如：8421、格雷码，使每组代码有特定的含义，称为编码，具有编码功能的逻辑电路成为编码器。

```
module bianma8_3(i, y);  
    input[7:0] i;  
    output[2:0] y;  
    reg[2:0] y;
```

```
always @(i)  
begin  
    case(i[7:0])  
        8'b00000001: y[2:0] = 3'b000;  
        8'b00000010: y[2:0] = 3'b001;  
        8'b00000100: y[2:0] = 3'b010;  
        8'b00001000: y[2:0] = 3'b011;  
        8'b00010000: y[2:0] = 3'b100;  
        8'b00100000: y[2:0] = 3'b101;  
        8'b01000000: y[2:0] = 3'b110;  
        8'b10000000: y[2:0] = 3'b111;  
        default: y[2:0] = 3'b000;  
    endcase  
end  
endmodule
```

❖ **8-3 编码器**是
将2的 n 次方个
分离的信息以 n
个二进制代码来
表示。

译码器

❖ 译码器：译码是编码的逆过程，它的功能是将具有特定含义的二进制编码进行辨别，并转换成控制信号，具有译码功能的逻辑电路称为译码器。

```

always @(a, y, g1, g2, g3)
begin
  if(g1 == 0) y = 8'b1111_1111;
  else if(g2 == 1) y = 8'b1111_1111;
  else if(g3 == 1) y = 8'b1111_1111;
  else
    case(a[2:0])
      3'b000: y[7:0] = 8'b1111_1110;
      3'b001: y[7:0] = 8'b1111_1101;
      3'b010: y[7:0] = 8'b1111_1011;
      3'b011: y[7:0] = 8'b1111_0111;
      3'b100: y[7:0] = 8'b1110_1111;
      3'b101: y[7:0] = 8'b1101_1111;
      3'b110: y[7:0] = 8'b1011_1111;
      3'b111: y[7:0] = 8'b0111_1111;
      default: y[7:0] = 8'b1111_1111;
    endcase
  end
endmodule

```

```

module decoder3_8
  (y, a, g1, g2, g3);
  output[7:0] y;
  input[2:0] a;
  input g1, g2, g3;
  reg[7:0] y;

```

3 – 8译码器是将 **n** 个二进制选择线，最多译码成 **2 的 n 次方** 个分离的信息以来表示。

```
module decoder3_8(y, a, g1, g2, g3);  
    output[2:0] y;  
    input[2:0] a;  
    input g1, g2, g3;  
    reg[2:0] y;  
    always @(a, g1, g2, g3)  
    begin  
        if(g1 ==0) y = 8'b1111_1111;  
        else if(g2 ==1) y = 8'b1111_1111;  
        else if(g3 ==1) y = 8'b1111_1111;  
        else  
            begin  
                y = 8'b0000_0001<<a;  
                y = ~y;  
            end  
        end  
    end  
endmodule
```

组合逻辑电路设计

- ❖ **数据选择器**：是指经过选择，把多个通道的数据传到唯一的公共数据通道上。实现数据选择功能的逻辑电路称为数据选择器。
- ❖ **四选一数据选择器**：对四个数据源进行选择，使用两位地址码**A1A0**产生地址信号来选择输出。


```
module mux41(y, g, d0, d1, d2, d3, a);  
    output y;  
    input[1:0] a;  
    input g;  
    input d0, d1, d2, d3;  
    reg y;  
    always @ (d0, d1, d2, d3, a, g)  
    begin  
        if(g ==0) y = 0;  
        else begin  
            case(a[1:0])  
                2'b00: y = d0;  
                2'b01: y = d1;  
                2'b10: y = d2;  
                2'b11: y = d3;  
            end  
        end  
    end  
end  
endmodule
```

case语句实现

```
module mux41(y, g, d0, d1, d2, d3, a);  
  output[2:0] y;  
  input[1:0] a;  
  input g;  
  input d0, d1, d2, d3;  
  reg[2:0] y;  
  wire nota1, nota2, x1, x2, x3, x4;  
  not (nota1, a[1]),  
      (nota2, a[2]);  
  and (x1, d0, nota1, nota[0]);  
      (x2, d1, nota1, a[0]);  
      (x3, d2, a[1], nota[0]);  
      (x4, d3, a[1], a[0]);  
  or (y, x1, x2, x3, x4);  
endmodule
```

门元件实现

```
module mux4_1a(y, g, d0, d1, d2, d3, a);  
    output y;  
    input[1:0] a;  
    input g;  
    input d0, d1, d2, d3;  
    reg y;  
    assign y =  
        ((d0&~a[1]&~a[0])|(d1&~a[1]&a[0])  
  
        |(d2&a[1]&~a[0])|(d3&a[1]&a[0]))&g;  
endmodule
```

数据流方式实现

```
module mux4_1a(y, g, d0, d1, d2, d3, a);  
    output y;  
    input[1:0] a;  
    input g;  
    input d0, d1, d2, d3;  
    reg y;  
    assign y =  
    g?(a[1]?(a[0]?d3:d2):(a[0]?d1:d0)):0;  
endmodule
```

条件运算符描述实现

数据分配器

数据分配器实现的功能与数据选择器相反。数据分配器是将一个数据源根的数据根据需要送到不同的通道上，实现数据分配功能的逻辑电路成为数据分配器。

```
module dmux (y0, y1, y2, y3, din, sel);  
    output y0, y1, y2, y3;  
    input[1:0] sel;  
    input din;  
    reg y0, y1, y2, y3;  
    always @ (din, sel)  
    begin  
        y0 = 0; y1 = 0; y2 = 0; y3 = 0;  
        case(sel[1:0])  
            2'b00: y0 = din;  
            2'b01: y1 = din;  
            2'b10: y2 = din;  
            2'b11: y3 = din;  
            default::;  
        endcase  
    end  
endmodule
```

数值比较器

在数字系统中，数值比较器就是两个数 **A**, **B** 进行比较，以判断其大小的逻辑电路，比较的结果有 **$A > B$** ， **$A = B$** ， **$A < B$** 三种情况，这三种情况仅有一种其值为真。

```
module comparator (y1, y2, y3, a, b);  
  output y1, y2, y3;  
  input[3:0] a, b;  
  reg y0, y1, y2, y3;  
  always @ (a, b)  
  begin  
    if(a > b) begin  
      y1 = 1; y2 = 0; y3 = 0;  
    end  
    else if(a == b) begin  
      y1 = 0; y2 = 1; y3 = 0;  
    end  
    else if(a < b) begin  
      y1 = 0; y2 = 0; y3 = 1;  
    end  
  end  
endmodule
```


加法器

//调用门元件

//数据流方式

//行为描述

//行为描述 **case** ,真值表

```
module half_add (sum, cout, a, b);
```

```
    output sum, cout;
```

```
    input a, b;
```

```
    reg[2:0] sum,cout;
```

```
    always @ (a, b)
```

```
    begin
```

```
        case[{a, b}]
```

```
            2'b00: begin cout = 0, sum = 0;end
```

```
            2'b01: begin cout = 0, sum = 1;end
```

```
            2'b10: begin cout = 0, sum = 1;end
```

```
            2'b11: begin cout = 1, sum = 0;end
```

```
        endcase
```

```
    end
```

```
endmodule
```

a, b);

半

全加器

// 数据流方式

module add1 (cin, sum, cout, a, b);

// 一位全加器的行为描述

cout, a, b);

// 混合方式

module add1 (cin, sum, cout, a, b);

output sum, cout;

input a, b;

reg cout, m1, m2, m3;

wire s1;

xor (s1, a, b);

always @ (a, b, cin)

begin

m1 = a & b;

m2 = a & cin;

m3 = cin & b;

cout = (m1 | m2) | m3;

end

assign sum = s1 ^ cin;

endmodule

n)|(cin & b);

· cin;

n3);

// 数据流方式

4位全加器

// 结构描述的4为级联全加器

// 4位全加器的行为描述

```
module add4 (cin, sum, cout, a, b);  
  output[3:0] sum;  
  output cout;  
  input[3:0] a, b;  
  input cin;  
  reg cout;  
  reg[3:0] sum;  
  always @ (*)  
    begin  
      {cout, sum} = a + b + cin;  
    end  
endmodule
```

```
b);  
cout, a, b);
```

```
a[0], b[0]);  
a[1], b[1]);  
a[2], b[2]);  
t, a[3],
```

超前进位加法器

```
module fulladd4(sum, c_out, a, b, cin);  
    output [3:0] sum;  
    output c_out;  
    input [3:0] a, b;  
    input cin;  
    wire p0, g0, p1, g1, p2, g2, p3, g3;  
    wire c4, c3, c2, c1;  
    assign p0 = a[0] ^ b[0];  
        p1 = a[1] ^ b[1];  
        p2 = a[2] ^ b[2];  
        p3 = a[3] ^ b[3];  
    assign g0 = a[0] & b[0];  
        g1 = a[0] & b[1];  
        g2 = a[0] & b[2];  
        g3 = a[0] & b[3];
```

```
    assign c1 = g0 | (p0 & cin),  
        c2 = g1 | (p1 & c1),  
        c3 = g2 | (p2 & c2),  
        c4 = g3 | (p3 & c2);  
  
    assign sum[0] = p0 ^ cin;  
        sum[1] = p1 ^ c1;  
        sum[2] = p2 ^ c2;  
        sum[3] = p3 ^ c3;  
  
    assign cout = c4;  
endmodule
```

减法器

//行为描述

```
module half_sub (dout, cout, a, b);  
    output dout, cout;
```

//行为描述, 1位全减器

```
module sub1 (cin, dout, cout, a, b);
```

//行为描述, 4位全减器

```
module sub4 (cin, dout, cout, a, b);  
    output[3:0] dout;  
    output cout;  
    input[3:0] a, b;  
    input cin;  
    reg[3:0] dout,  
    reg cout;  
    always @ (a, b)  
        begin  
            {cout, dout} = a - b - cin;  
        end  
endmodule
```

cin;



赋值语句

内容概要

- 一、赋值语句
- 二、非阻塞赋值与阻塞赋值的区别



一、赋值语句

- 分为两类：

① 连续赋值语句——assign语句，用于对wire型变量赋值，是描述组合逻辑最常用的方法之一。

[例] assign c=a&b; //a、b、c均为wire型变量

② 过程赋值语句——用于对reg型变量赋值，有两种方式：

- 非阻塞 (non-blocking)赋值方式：

赋值符号为<=，如 b <= a ；

- 阻塞 (blocking)赋值方式：

赋值符号为=，如 b = a ；

非阻塞赋值与阻塞赋值方式的主要区别

- **非阻塞** (non-blocking)赋值方式 ($b \leftarrow a$):
 - b的值被赋成新值a的操作,并不是立刻完成的,而是在块结束时才完成;
 - 块内的多条赋值语句在块结束时同时赋值;
 - 硬件有对应的电路。
- **阻塞** (blocking)赋值方式 ($b = a$):
 - b的值立刻被赋成新值a;
 - 完成该赋值语句后才能执行下一句的操作;
 - 硬件没有对应的电路,因而综合结果未知。

❖ 建议在初学时只使用一种方式,不要混用!

❖ 建议在可综合风格的模块中使用**非阻塞**赋值!

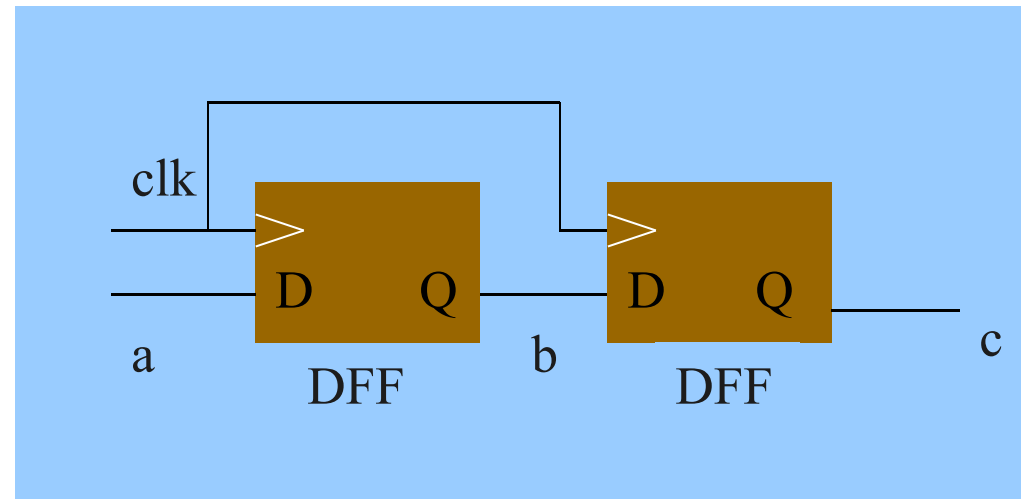
二、非阻塞赋值与阻塞赋值的区别

1. 非阻塞赋值方式

注：c的值比b的值落后一个时钟周期！

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

非阻塞赋值在
块结束时才完
成赋值操作！



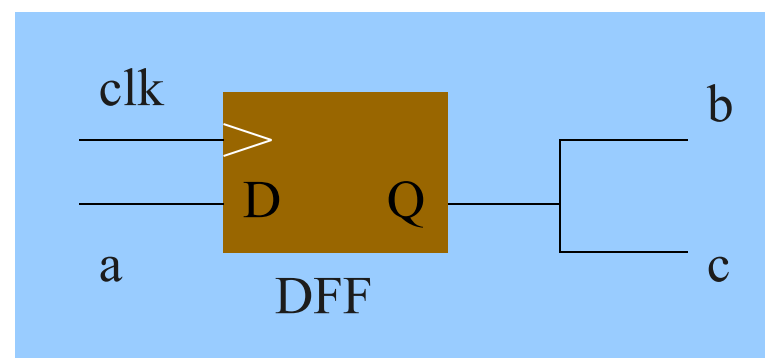
非阻塞的意思是每条赋值语句的结果直到 **always** 块的结尾才能看到。

always 块中所有非阻塞赋值语句在求值时所用的值全部都是进入 **always** 时，各个变量已具有的值。

2. 阻塞赋值方式

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

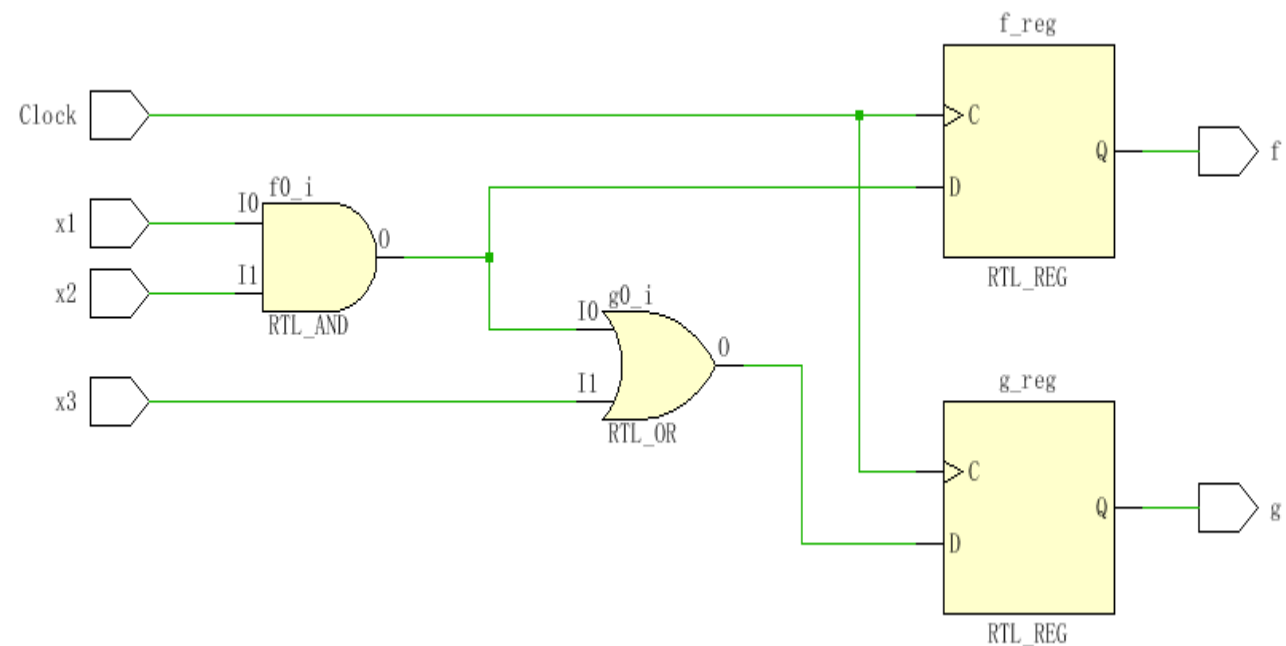
阻塞赋值在**该语句**结束时就完成了赋值操作！



注：在一个块语句中，如果有多条阻塞赋值语句，在前面的赋值语句没有完成之前，后面的语句就不能被执行，就像被阻塞了一样，因此称为**阻塞赋值方式**。
这里c的值与b的值一样 ！

阻塞赋值

```
87 module example7_5 (x1, x2, x3, Clock, f, g);  
88     input x1, x2, x3, Clock;  
89     output reg f, g;  
90     always @(posedge Clock)  
91     begin  
92         f = x1 & x2;  
93         g = f | x3;  
94     end  
95 endmodule
```

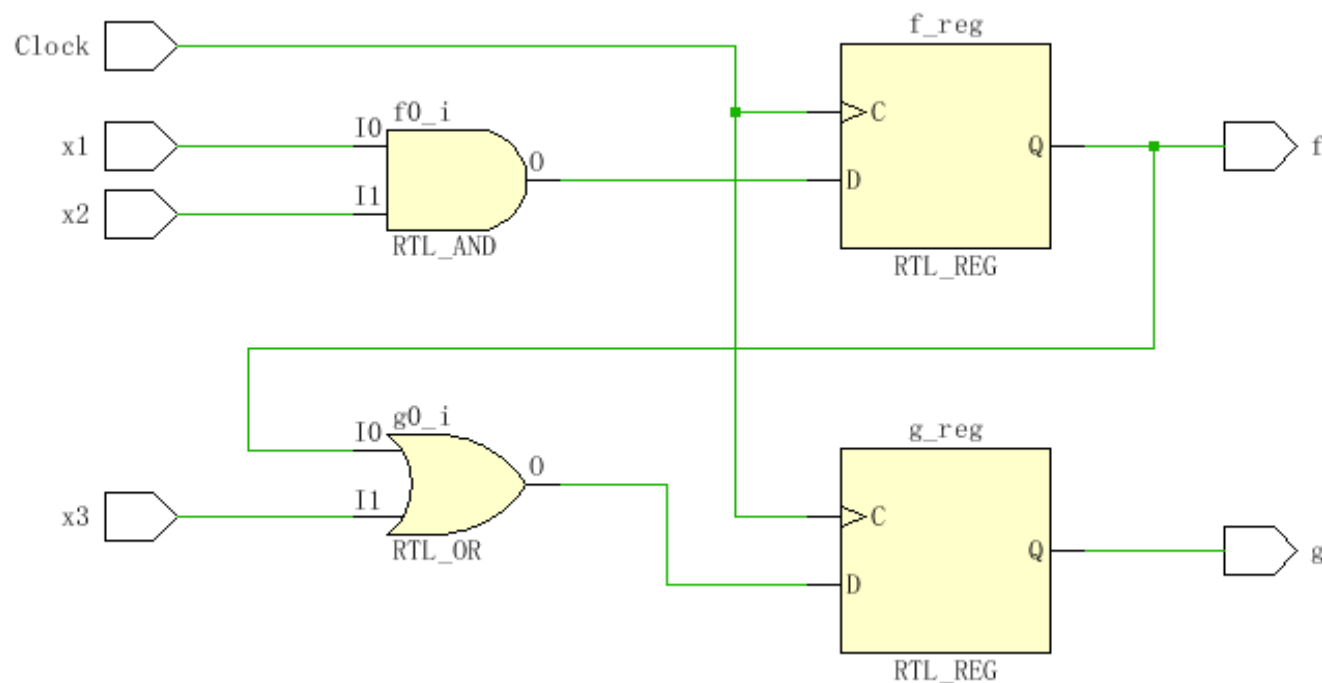


```

87 module example7_5 (x1, x2, x3, Clock, f, g);
88     input x1, x2, x3, Clock;
89     output reg f, g;
90     always @(posedge Clock)
91     begin
92         f <= x1 & x2;
93         g <= f | x3;
94     end
95 endmodule

```

非阻塞赋值

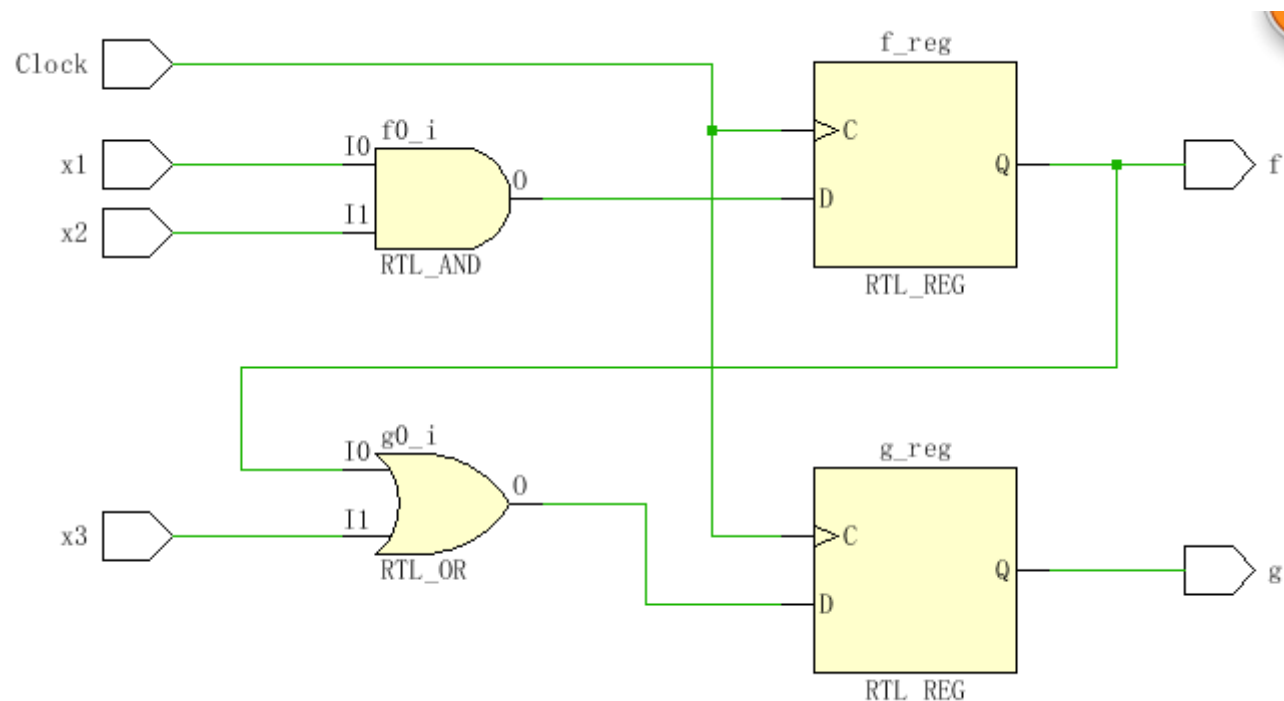


若把给f和g赋值的两条语句次序颠倒

```
87 module example7_5 (x1, x2, x3, Clock, f, g);  
88     input x1, x2, x3, Clock;  
89     output reg f, g;  
90     always @(posedge Clock)  
91     begin  
92         g = f | x3;  
93         f = x1 & x2;  
94     end  
95 endmodule
```

阻塞赋值

用阻塞赋值描述时序电路很容易生成错误的电路。阻塞赋值语句对语句顺序的依赖可能综合成错误的电路，因而是有风险的



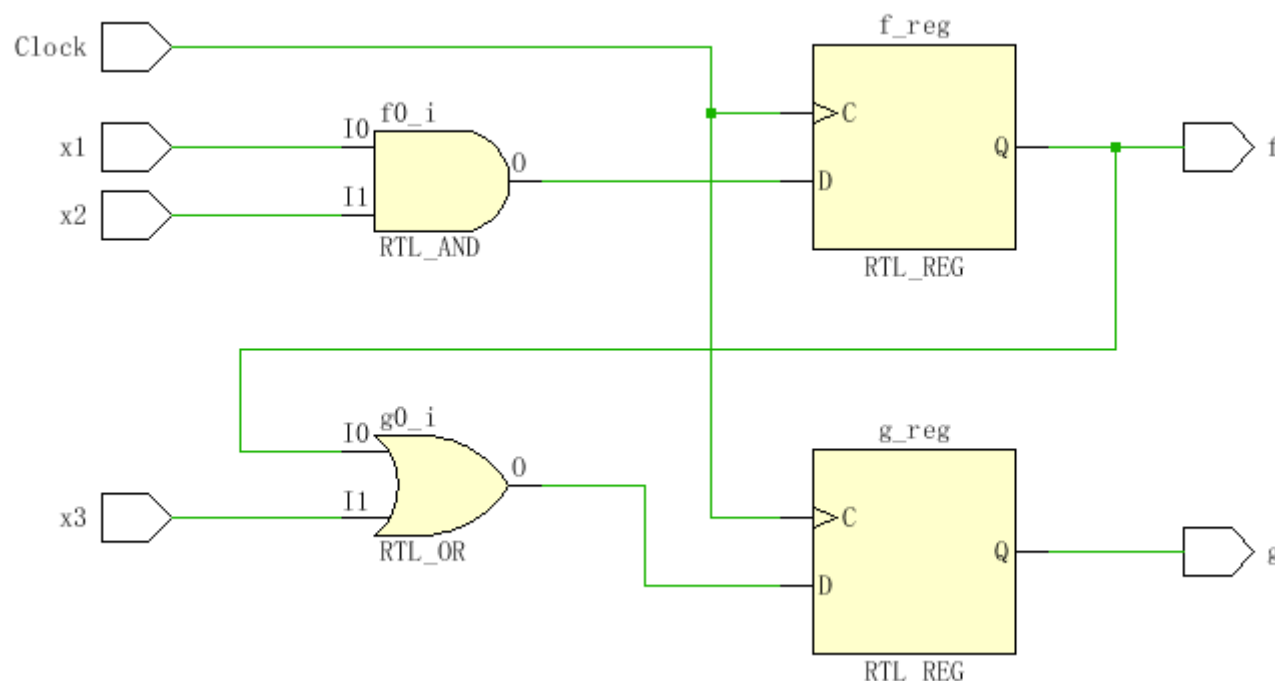
若把给f和g赋值的两条语句次序颠倒

```
87 module example7_5 (x1, x2, x3, Clock, f, g);  
88     input x1, x2, x3, Clock;  
89     output reg f, g;  
90     always @(posedge Clock)  
91     begin  
92         g <= f | x3;  
93         f <= x1 & x2;  
94     end  
95 endmodule
```

非阻塞赋值

最好用非阻塞赋值<=
来描述时序电路

语句顺序颠倒对非阻塞
赋值的代码没有任何影
响。





组合逻辑电路的非阻塞赋值

非阻塞赋值是否可以用于描述组合逻辑电路。答案是在大多数情况下可以用，但是当`always`块中后面的赋值语句依赖于前面赋值语句的结果时，非阻塞赋值会产生无意义的电路。

我们希望产生一个组合逻辑函数 f ，当 A 中相邻两位为 1 时， f 就等于1。用阻塞赋值描述这个函数的一种方法如下

```
always @ (A)
begin
  f = A[1]&A[0];
  f = f | (A[2]&A[1]);
end
```

这些声明语句实现了想要的逻辑函数，就是

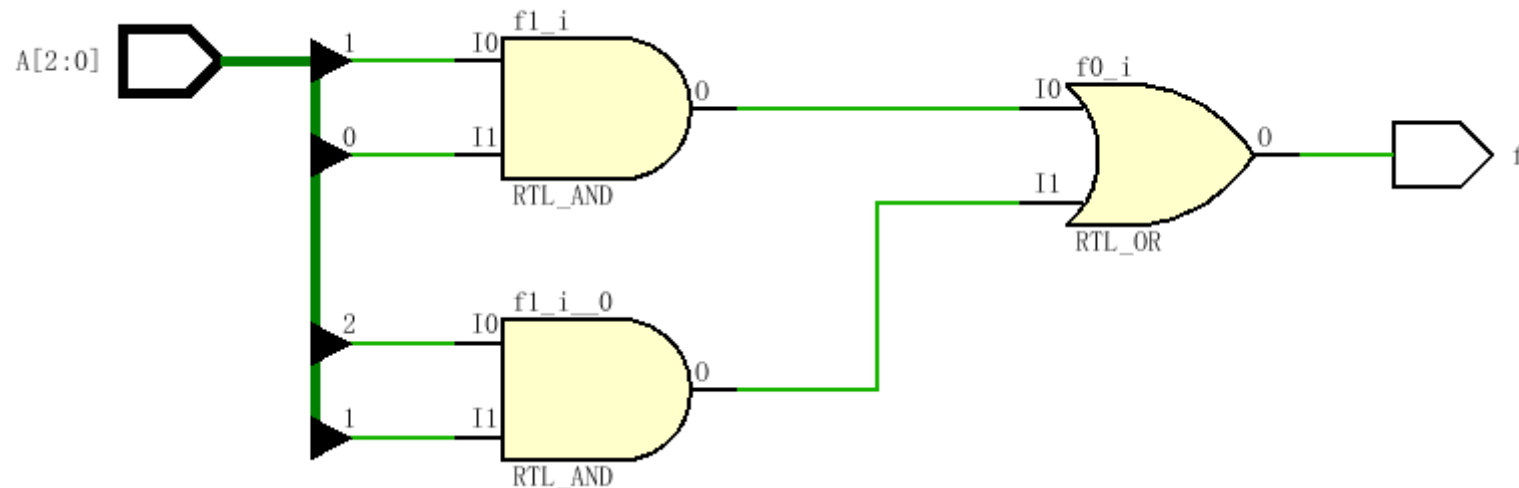
$$f = a_1a_0 + a_2a_1$$

```

87 module example ( f, A);
88     input [2:0]A;
89     output reg f;
90     always @ (A)
91     begin
92         f =A[1]&A[0];
93         f = f|(A[2]&A[1]);
94     end
95 endmodule

```

$$f = a_1a_0 + a_2a_1$$





现在考虑用非阻塞赋值将代码改为：

```
f <= A[1]&A[0];
```

```
f <= f | (A[2]&A[1]);
```

对应于这段代码，Verilog 语义方面有两点是很关键的：

1. 非阻塞赋值语句的结果仅在 always 块中所有语句求值结束后才可以看到。
2. 当always块中同一变量多次赋值后，只保留最后一次赋值的结果。

在这个例子中，在我们进入always块时f有一个未说明的初始值。第一条语句赋值f=a1a0，但是这个结果对于第二条语句是不可见的。它只能看到原始的还未赋值的f值。所以第二条语句越过（删除）了第一条语句，产生逻辑函数

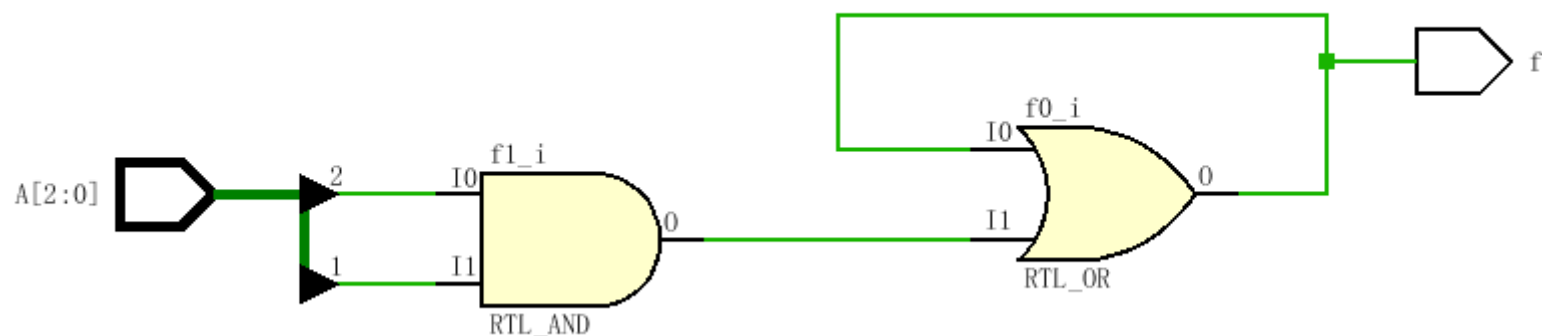
$$f = f + a_2a_1$$

```

87 module example ( f, A );
88     input [2:0]A;
89     output reg f;
90     always @ (A)
91     begin
92         f <= A[1]&A[0];
93         f <= f | (A[2]&A[1]);
94     end
95 endmodule

```

为避免产生不想要的时序电路，描写组合逻辑电路时最好使用阻塞赋值=。



时序电路设计

时序电路设计，时序电路是指在任一时刻的输出信号不仅与当时的输入信号有关，而且还与电路的原来状态有关。常用的时序逻辑电路有计数器、寄存器、锁存器和存储器等。

同步计数器

同步计数器，计数器的逻辑功能是用与记忆时钟脉冲的具体个数，通常计数器最多能记忆时钟的最大数目 m 称为计数器的模。基本原理是将几个触发器按照一定的顺序连接起来，然后根据触发器的组合状态，按照一定的技术规律随着时钟脉冲变化来记忆时钟脉冲的个数。

按照计数方向分为加法，减法和可逆计数器。

按照其中触发器是否与时钟同步又分为同步计数器和异步计数器。

```

module cnt16 (cout, q, clk, clr, load, en, d);
    output[3:0] q; //输出
    output cout; //进位信号
    input clk, clr, load, en;
    input[3:0] d;
    reg[3:0] q;
    reg cout;
    always @ (posedge clk) begin
        if (clr) begin q <= 0; end
        else if (load) begin q <= d; end
        else if (en) begin
            q <= q + 1;
            if(q == 4'b1111) begin cout <= 1; end
            else begin cout <= 0; end
        end
        else begin q <= q; end
    end
end
endmodule

```

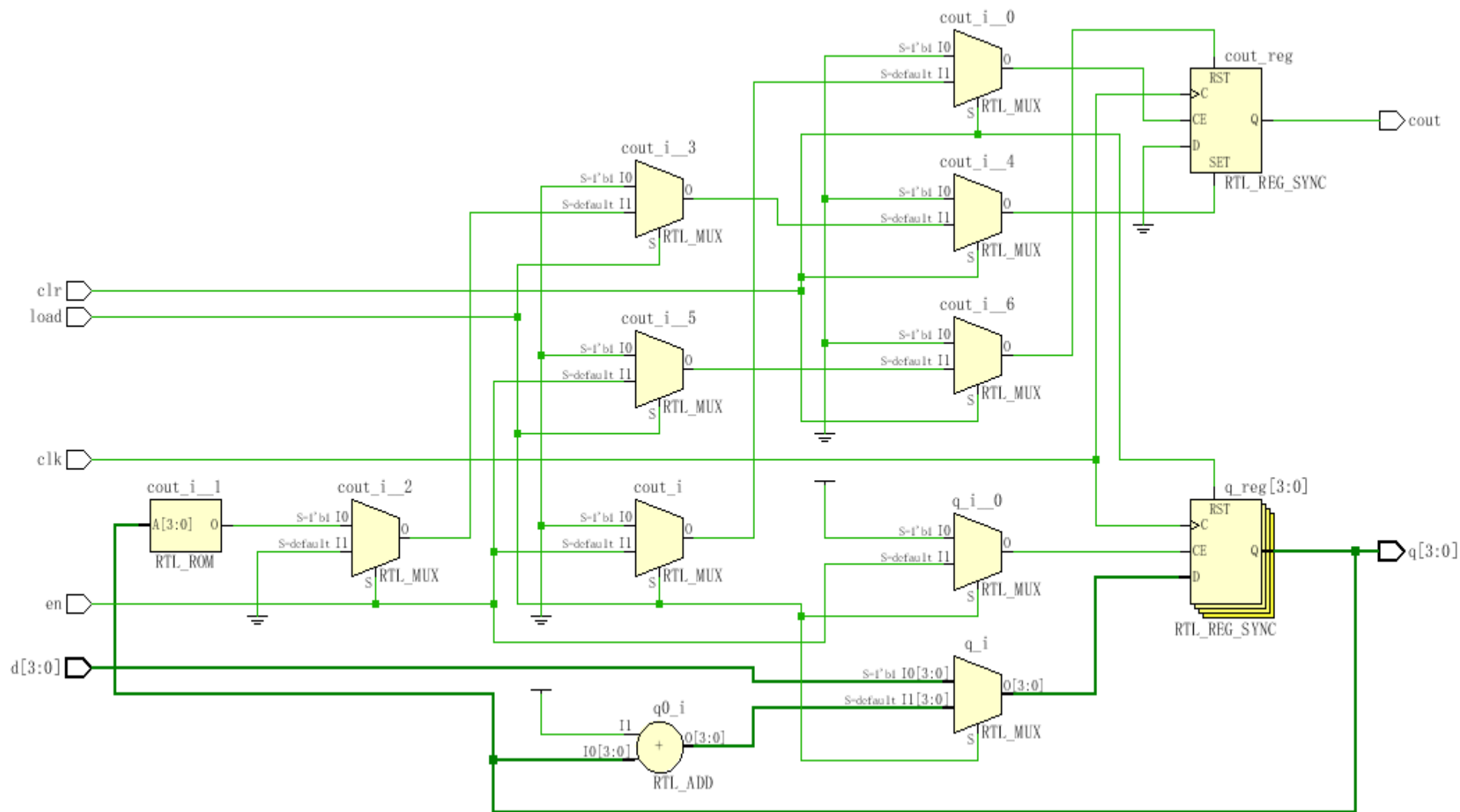
同步4位计数器，
同步清零，同
步置数。

```

113 module cnt16 (cout, q, clk, clr, load, en, d);
114     output[3:0] q; //输出
115     output cout; //进位信号
116     input clk, clr, load, en;
117     input[3:0] d;
118     reg[3:0] q;
119     reg cout;
120 always @ (posedge clk) begin
121     if (clr) begin q <= 0; end
122     else if (load) begin q <= d; end
123         else if (en) begin
124             q <= q + 1;
125             if(q == 4'b1111)
126                 cout <= 1;
127             else cout <= 0;
128         end
129     else begin q <= q; end
130 end
131 endmodule

```

同步4位计数器，
同步清零，同
步置数。



```

module cnt24 (ten, one, cout, clk, clr);
  output[3:0] ten, one; //输出
  output cout; //进位信号
  input clk, clr;
  reg[3:0] ten, one;
  reg cout;
  always @ (posedge clk) begin
    if (clr) begin ten <= 0; one <= 0; end
    else begin
      if({ten, one} == 8'b0010_0011) //24十进制
        begin ten <= 0; one <= 0; cout <= 1; end
      else if(one==4'b1001)
        begin one <= 0; ten<=ten+1;
          cout <= 0; end
        else begin
          one <= one + 1; cout <=0; end
        end
      end
    end
  endmodule

```

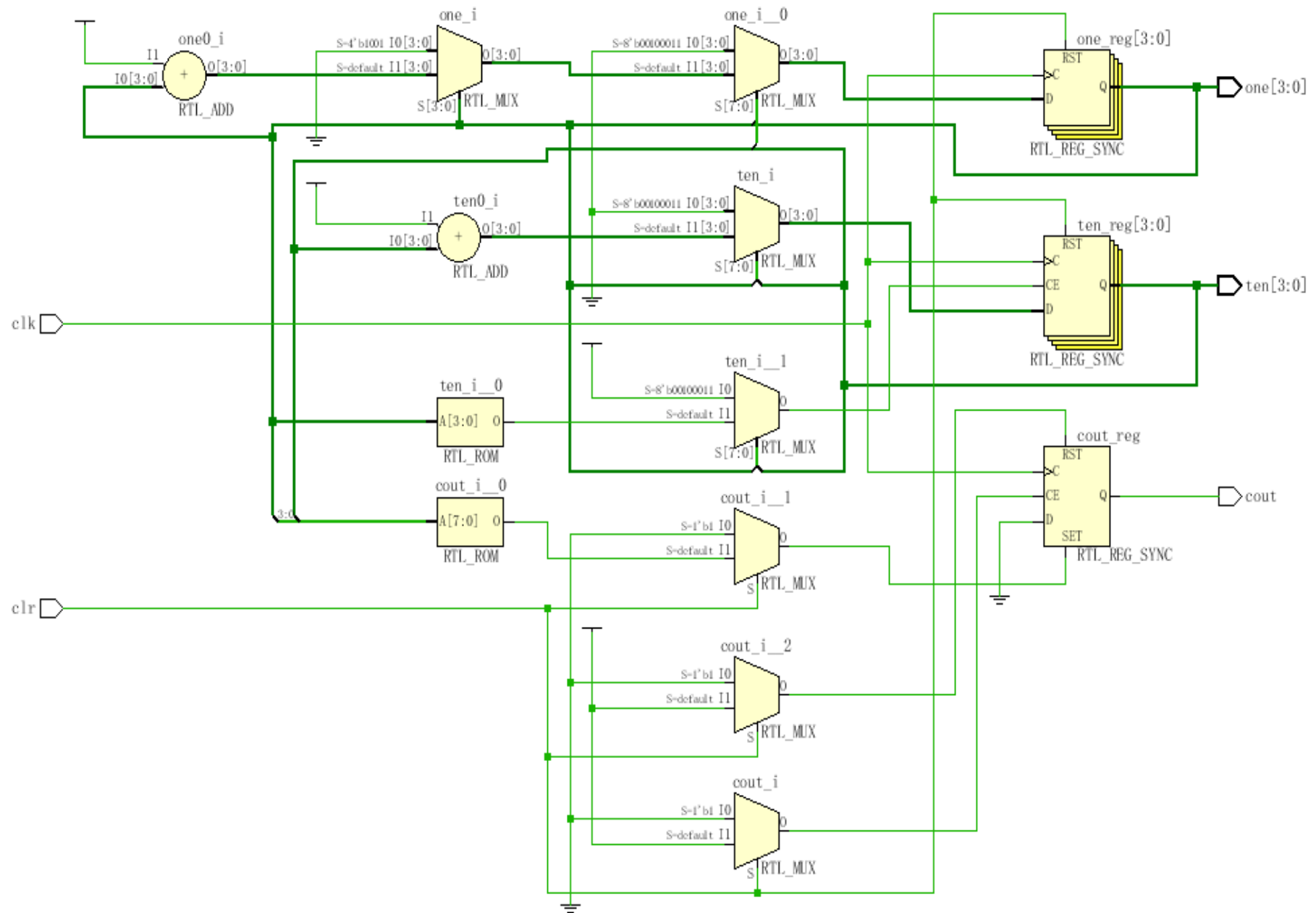
同步**24**进制计数器，同步清零。


```

96 module cnt24 (ten, one, cout, clk, clr);
97     output[3:0] ten, one; //输出
98     output cout; //进位信号
99     input clk, clr;
100     reg[3:0] ten, one;
101     reg cout;
102     always @ (posedge clk) begin
103         if (clr) begin ten <= 0; one <= 0; end
104         else begin
105             if ({ten, one} == 8'b0010_0011) //24十进制
106                 begin ten <= 0; one <= 0; cout <= 1; end
107             else if (one == 4'b1001)
108                 begin one <= 0; ten <= ten + 1;
109                     cout <= 0; end
110             else begin one <= one + 1; cout <= 0; end
111         end
112     end
113 endmodule

```

同步**24**进制计数器，同步清零。



```
always @ (posedge clk)
begin
    if (reset) qout <= 0;
    else if (load) qout <= data;
    else if (cin) begin
        if(qout[3:0] == 9) begin
            qout[3:0] <= 0;
            if(qout[7:4] == 5) qout[7:4] <= 0;
            else qout[7:4] <= qout[7:4]+1;
        end
        else qout[3:0] <= qout[3:0]+1;
    end
end
end
assign cout = ((qout == 8'h59)&cin)?1:0;
endmodule
```

```
module count60(qout, cout,
    data, load, cin, reset, clk);
    output [7:0] qout;
    output cout;
    input [7:0] data;
    input load, cin, clk, reset;
    reg [7:0] qout;
```

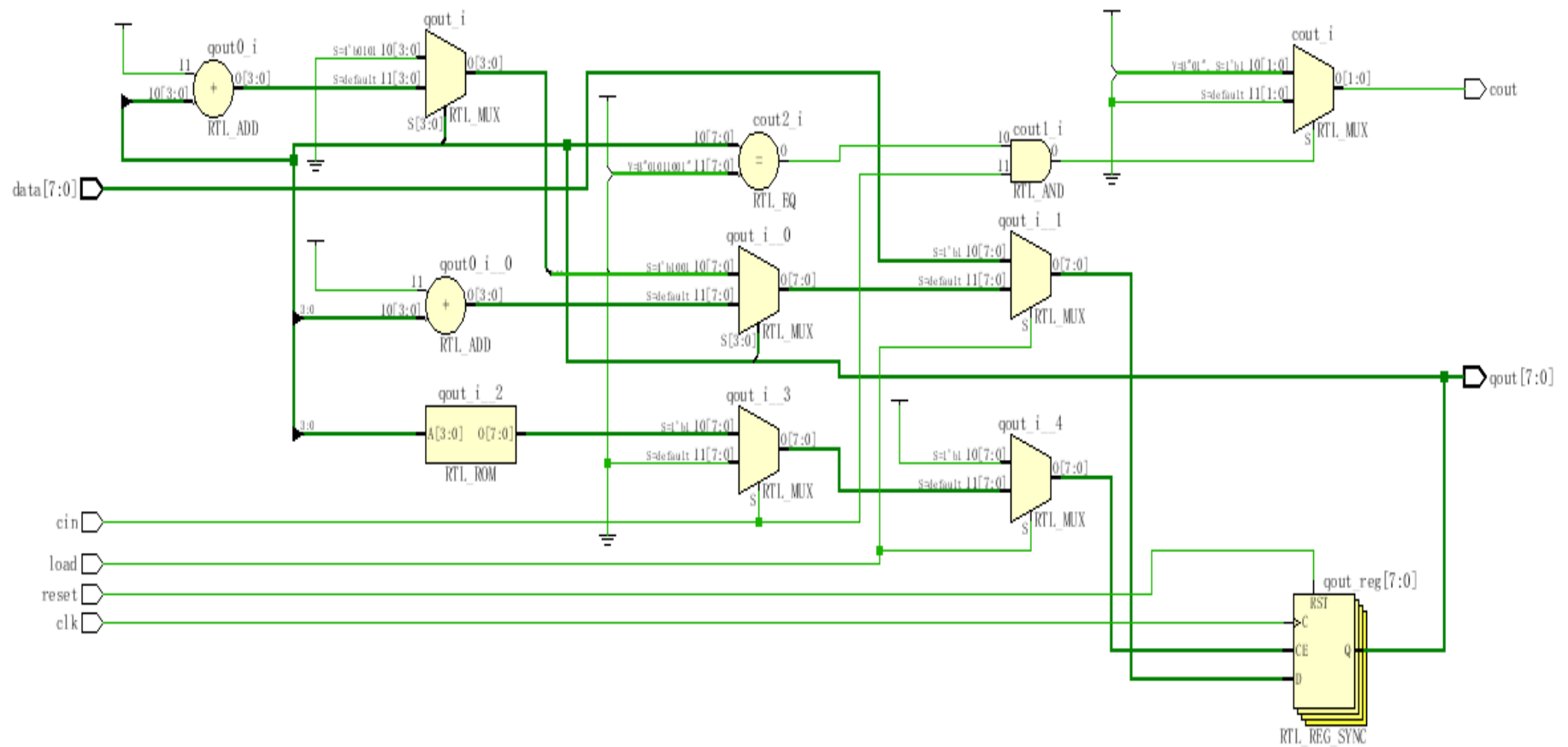
■ [例] 模为60的BCD码加法计数器counter60.v

```
count60.v
1 /* 模为60的BCD码加法计数器 */
2 module count60 (qout, cout, data, load, cin, reset, clk);
3     output[7:0]  qout;
4     output  cout;
5     input[7:0]   data;
6     input load, cin, reset, clk;
7     reg[7:0]     qout;
8     always @ (posedge clk)
9     begin
10         if (reset)          qout = 0;          // 同步复位
11         else if (load)       qout = data;       // 同步置数
12         else if (cin)        // 若cin = 1, 执行加1计数; 否则qout保持不变
13         begin
14             if (qout[3:0] == 9) // 低位是否为9?
15             begin
16                 qout[3:0] = 0; // 是则回0
17                 if (qout[7:4] == 5) // 高位是否为5?
18                     qout[7:4] = 0; // 是则回0
19                 else
20                     qout[7:4] = qout[7:4] + 1; // 高位不为5, 则高位加1
21             end
22         else
23             qout[3:0] = qout[3:0] + 1; // 低位不为9, 则低位加1
24         end
25     end
26     assign cout = ((qout == 8'h59) & cin) ? 1:0; //产生进位输出
27 endmodule
```

在always块内的语句是顺序执行的!

always块语句和assign语句是并行执行的!

<pre> 141 always @ (posedge clk) 142 begin 143 if (reset) qout <= 0; 144 else if (load) qout <= data; 145 else if (cin) begin 146 if(qout[3:0] == 9) 147 begin 148 qout[3:0] <= 0; 149 if(qout[7:4] == 5) qout[7:4] <= 0 150 else qout[7:4] <= qout[7:4]+1; 151 end 152 else qout[3:0] <= qout[3:0]+1; 153 end 154 end 155 assign cout = ((qout == 8'h59)&cin) ? 1:0; 156 endmodule </pre>	<pre> 135 module count60(qout, cout, data, load, cin, reset, clk) 136 output [7:0] qout; 137 output cout; 138 input [7:0] data; 139 input load, cin, clk, reset; 140 reg [7:0] qout; </pre>
--	---



异步计数器：异步计数器是指构成计数器的低位计数器触发的输出作为相邻计数器的时钟，这样逐级串联起来的一类计数器，时钟信号的这种接法又叫行波计数，异步计数器的技术延迟增加，从而影响了它的范围。

异步4位2进制计数器

```
always @ (posedge clk)
begin if(!rst) begin q[0] = 0; qn[0] = 1; end
      else begin q[0] = ~q[0]; qn[0] = ~qn[0]; end
end
always @ (posedge qn[0])
begin if(!rst) begin q[1] = 0; qn[1] = 1; end
      else begin q[1] = ~q[1]; qn[1] = ~qn[1]; end
end
always @ (posedge qn[1])
begin if(!rst) begin q[2] = 0; qn[2] = 1; end
      else begin q[2] = ~q[2]; qn[2] = ~qn[2]; end
end
always @ (posedge qn[2])
begin if(!rst) begin q[3] = 0; qn[3] = 1; end
      else begin q[3] = ~q[3]; qn[3] = ~qn[3]; end
end
endmodule
```

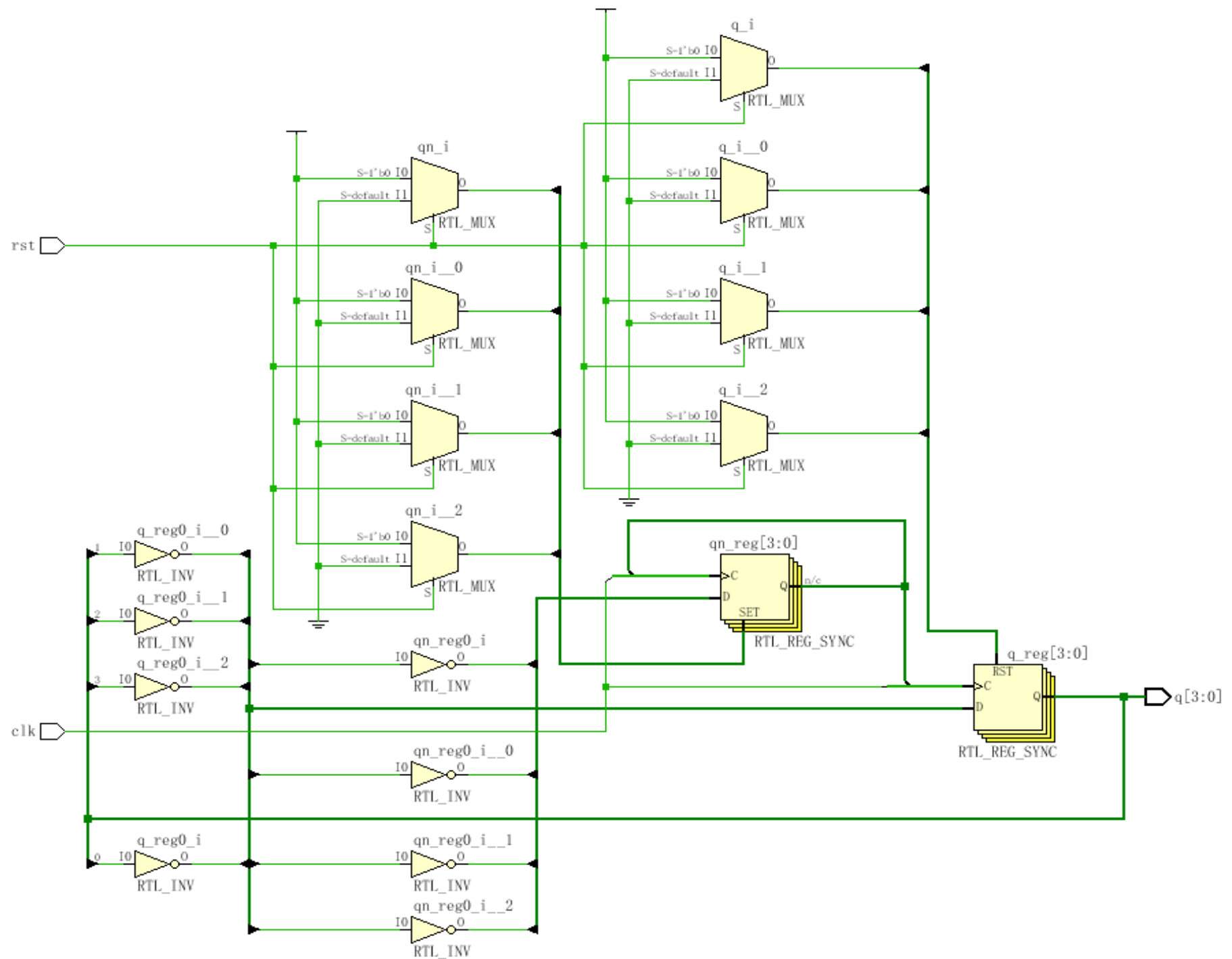
```
module yb_cnt4
    (q, clk, rst);
    output[3:0] q;
    input clk, rst;
    reg[3:0] q;
    reg[3:0] qn;
```



```

159 module yb_cnt4 (q, clk, rst);
160     output[3:0] q;
161     input clk, rst;
162     reg[3:0] q;
163     reg[3:0] qn;
164 always @ (posedge clk)
165     if(!rst) begin q[0] = 0; qn[0] = 1; end
166     else begin q[0] = ~q[0]; qn[0] = ~qn[0]; end
167 always @ (posedge qn[0])
168     if(!rst) begin q[1] = 0; qn[1] = 1; end
169     else begin q[1] = ~q[1]; qn[1] = ~qn[1]; end
170 always @ (posedge qn[1])
171     if(!rst) begin q[2] = 0; qn[2] = 1; end
172     else begin q[2] = ~q[2]; qn[2] = ~qn[2]; end
173 always @ (posedge qn[2])
174     if(!rst) begin q[3] = 0; qn[3] = 1; end
175     else begin q[3] = ~q[3]; qn[3] = ~qn[3]; end
176 endmodule

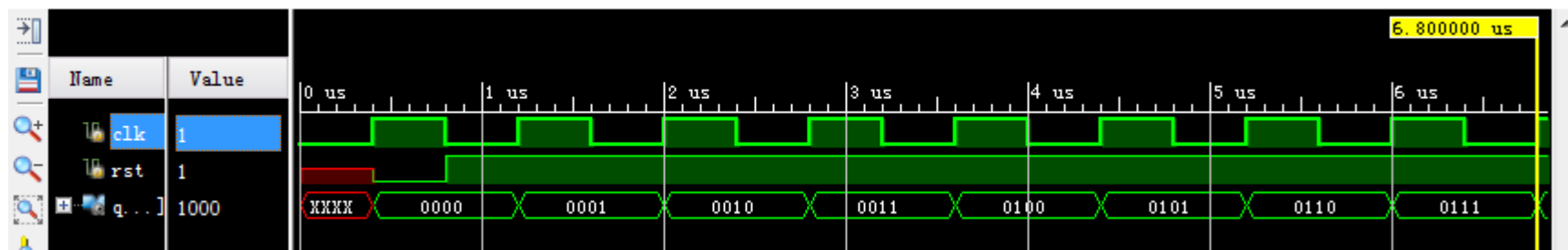
```



```

21 module sim( );
22     reg clk, rst;
23     wire [3:0] q;
24     initial begin
25         clk=0;
26         #400 rst=0;
27         #400 rst=1;
28     end
29     always #400 clk=~clk;
30     yb_cnt4 test(q, clk, rst);
31 endmodule

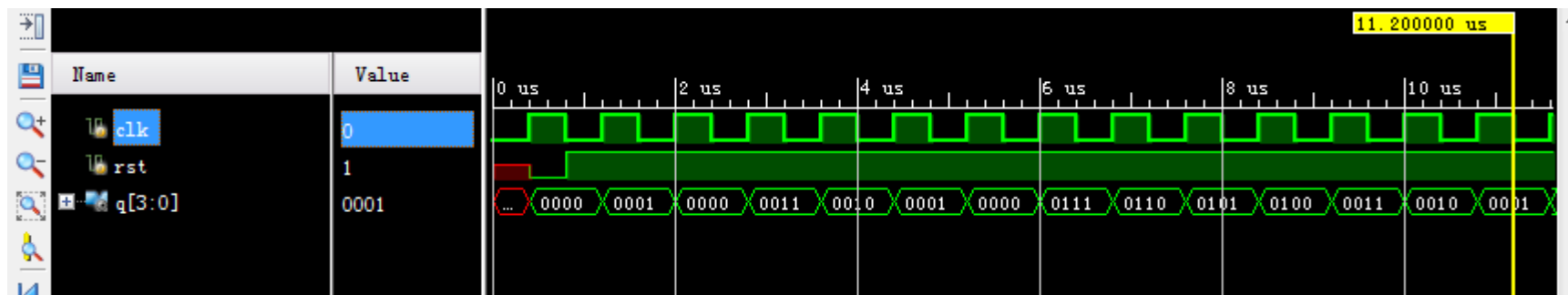
```

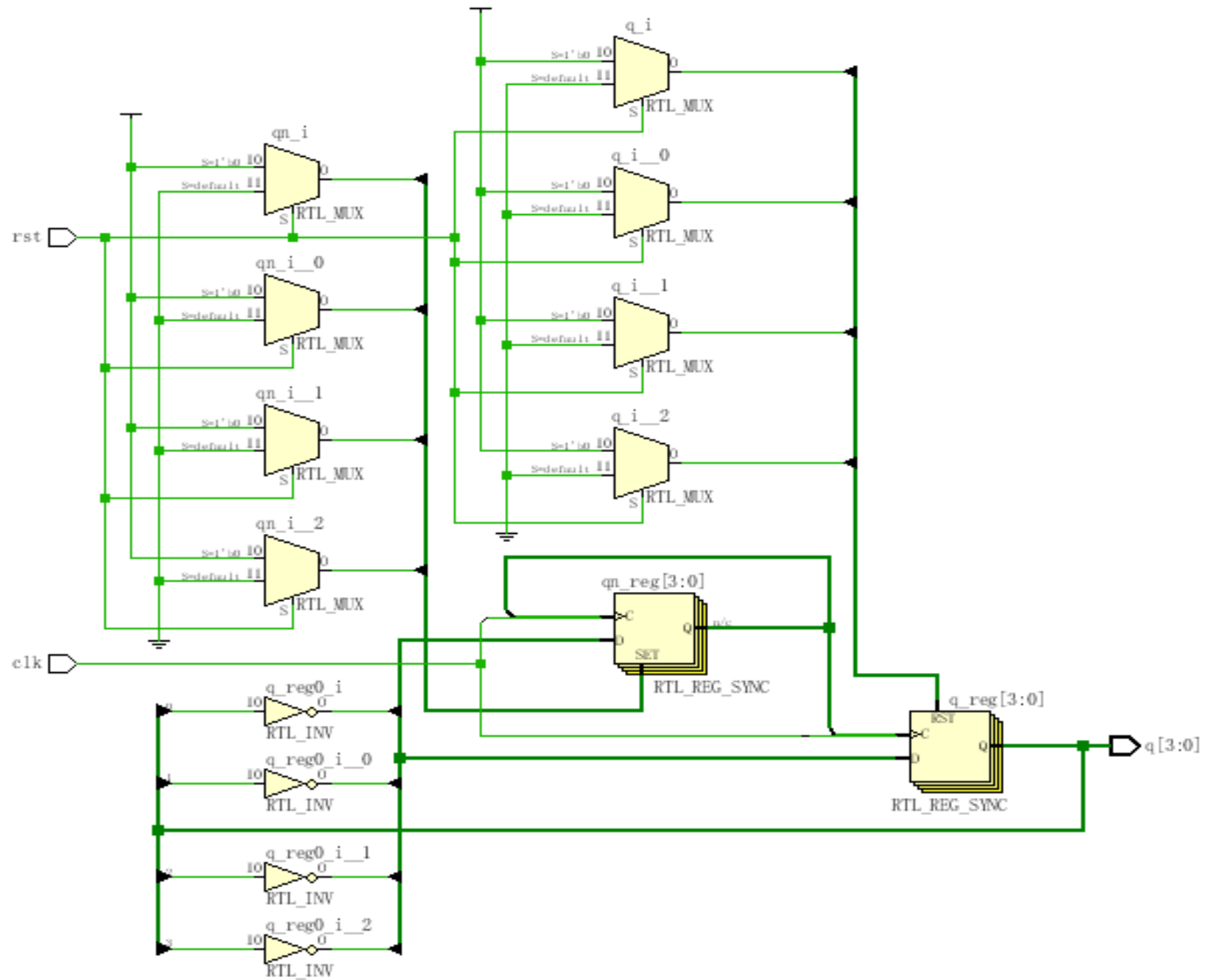


```

164 always @ (posedge clk)
165     if(!rst) begin q[0] <= 0; qn[0] <= 1; end
166     else begin q[0] <= ~q[0]; qn[0] <=~q[0]; end
167 always @ (posedge qn[0])
168     if(!rst) begin q[1] <= 0; qn[1] <= 1;end
169     else begin q[1] <= ~q[1]; qn[1] <=~q[1];end
170 always @ (posedge qn[1])
171     if(!rst) begin q[2] <= 0; qn[2] <= 1;end
172     else begin q[2] <= ~q[2]; qn[2] <=~q[2];end
173 always @ (posedge qn[2])
174     if(!rst) begin q[3] <= 0; qn[3] <= 1; end
175     else begin q[3] <= ~q[3]; qn[3] <=~q[3];end
176 endmodule

```





```

module yb_cnt16(q, clk, clr, load, m);
    output[6:0] q;
    input clk, clr, load;
    input[6:0] m;
    reg[6:0] q;
    reg[6:0] md;
    always @ (posedge clk)
    begin
        md <= m-1;
        begin if(!clr) begin q <= 0; end
            else begin if(load) begin q <= md; end
                else begin if(q == md)
                    begin q <= 0; end
                    else begin q <= q + 1; end
                end
            end
        end
    end
end
endmodule

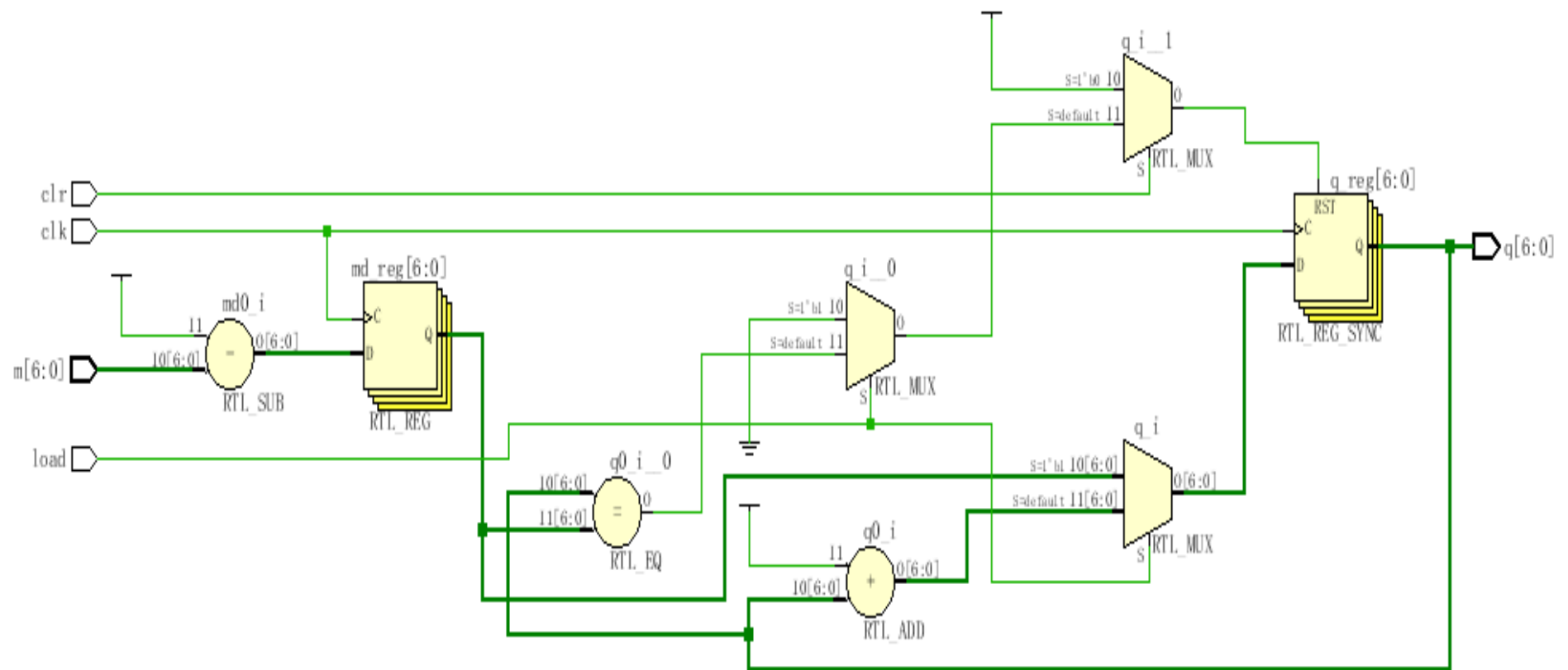
```

可变模计数器可以通过模值
控制端来改变计数器的模值

```

182 module yb_cnt16(q, clk, clr, load, m) ;
183     output[6:0]q;
184     input  clk, clr, load;
185     input[6:0] m;
186     reg[6:0]q;
187     reg[6:0]md;
188     always @ (posedge clk)
189     begin md <= m-1;
190         begin if(!clr) q<=0;
191             else begin if(load) q<=md;
192                 else begin if(q == md) q <= 0;
193                     else q<=q + 1;
194                 end
195             end
196         end
197     end
198 endmodule

```




```

module reg8_1 (q, d, clk, oe);
  output[7:0] q;//数据输出
  input[7:0] d; //数据输入
  input oe, clk; //三态控制端，时钟信号
  reg[7:0] q;
  always @ (posedge clk)
  begin
    if(oe) begin
      q <= 8'bz;
    end
    else begin
      q <= d;
    end
  end
endmodule

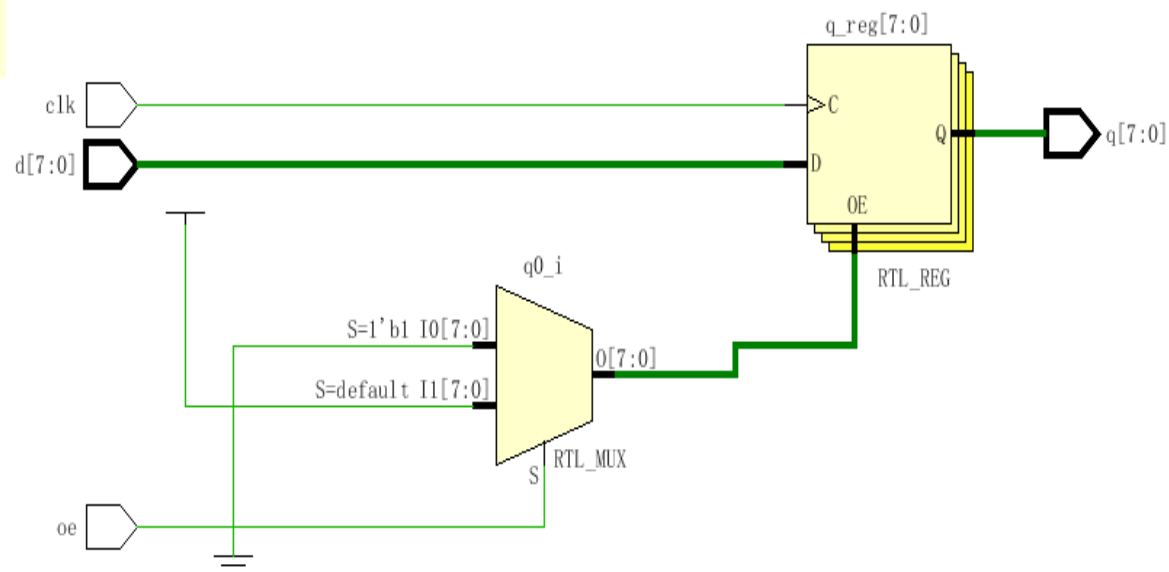
```

寄存器是数字电路中的基本模块，许多复杂的时序逻辑电路都是由它构成的。在数字系统中，寄存器是一种在某一特定信号的控制下用于存储一组二进制数据的时序电路。通常由触发器购得寄存器，把多个**D**触发器的时钟端连接起来就可以构成一个存储多位二进制代码的寄存器。

```

200 module reg8_1 (q, d, clk, oe);
201     output[7:0] q; //数据输出
202     input[7:0] d;  //数据输入
203     input oe, clk; //三态控制端, 时钟信号
204     reg[7:0] q;
205     always @ (posedge clk)
206     begin
207         if(oe)
208             q <= 8'bz;
209         else
210             q <= d;
211     end
212 endmodule

```



```

module reg8_1 (q, d, g, oe);
    output[7:0] q;//
    input[7:0] d; //
    input oe, g; //三态控制端，控制信号
    reg[7:0] q;
    always @ (*)
    begin
        if(oe) begin
            q <= 8'bz;
            end
        else begin
            if(g) q <= d;
            end
        end
    end
endmodule

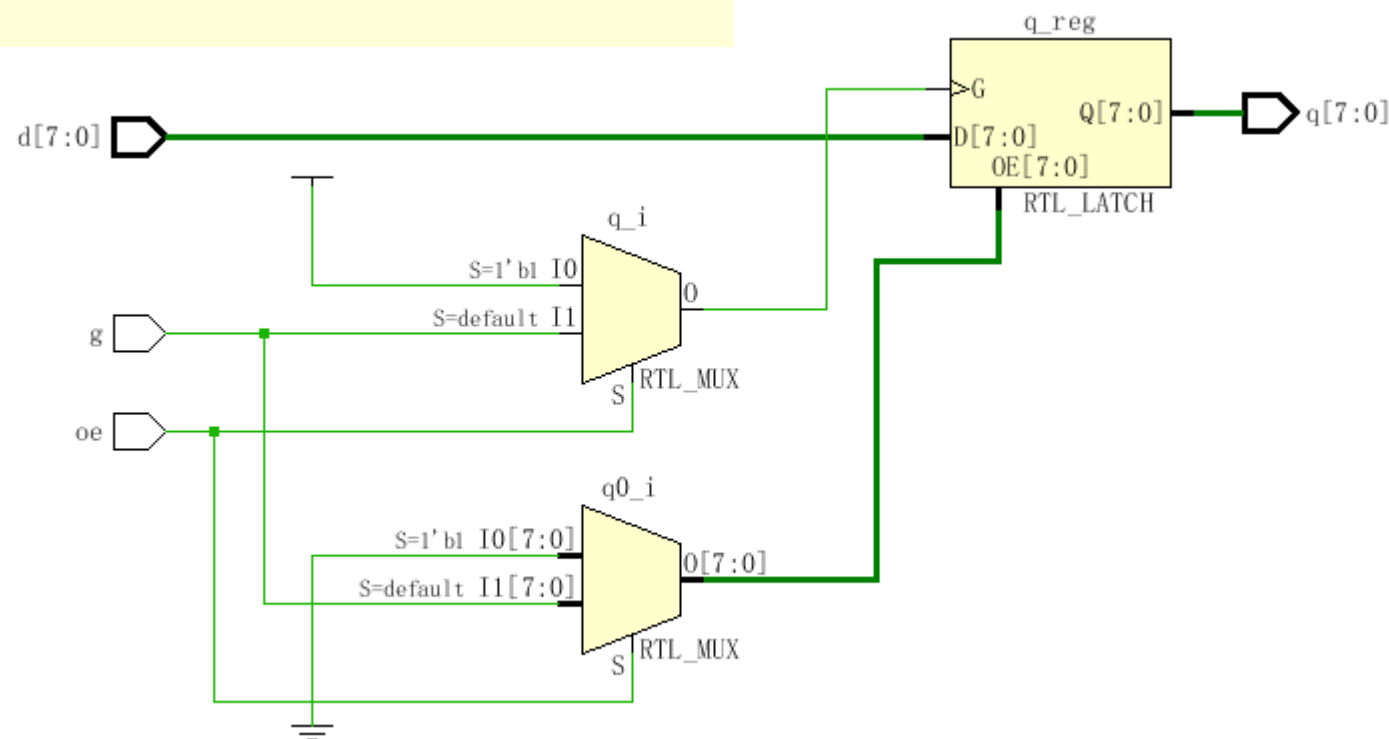
```

锁存器是一种与寄存器类似的器件。与寄存器采用同步时钟信号控制不同，锁存器是采用电位信号来进行控制的。

```

214 module reg8_1 (q, d, g, oe);
215     output[7:0] q; //
216     input[7:0] d;  //
217     input oe, g;   //三态控制端, 控制信号
218     reg[7:0] q;
219     always @ (*)
220         if(oe) q <= 8'bz;
221         else if(g) q <= d;
222 endmodule

```



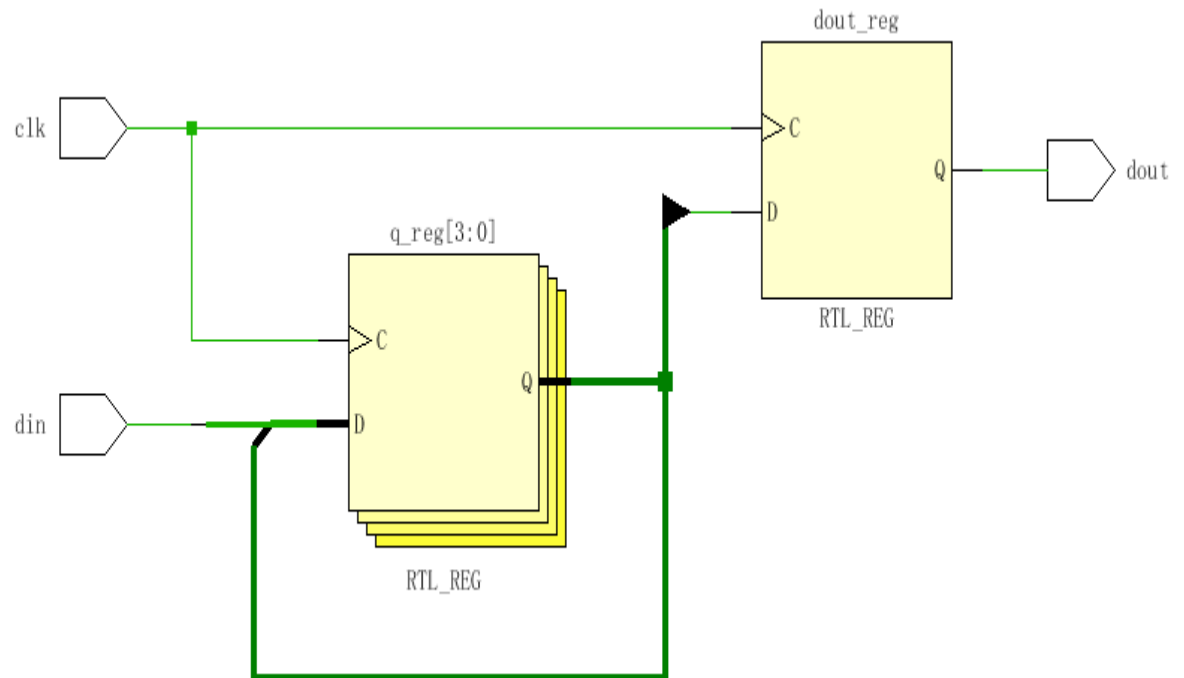
移位寄存器

移位寄存器是指寄存器里面存储的二进制数据能够在时钟信号的控制下一次左移或者右移，在数字电路中通常用于数据的串并转换、并串转换、数值运算等。包括双向移位寄存器、串入（并入）/串出（并出）移位寄存器。

```

module siso4 (dout, clk, din);//串入串出
  output dout;
  input clk;
  input din;
  reg dout;
  reg[3:0] q;
  always @ (posedge clk)
  begin
    q[0] <= din;
    q[3:1] <= q[2:0];
    dout <= q[3];
  end
endmodule

```



```

358 module shift4 (R, L, w, Clock, Q);
359     input [3:0] R;
360     input L, w, Clock;
361     output reg [3:0] Q;

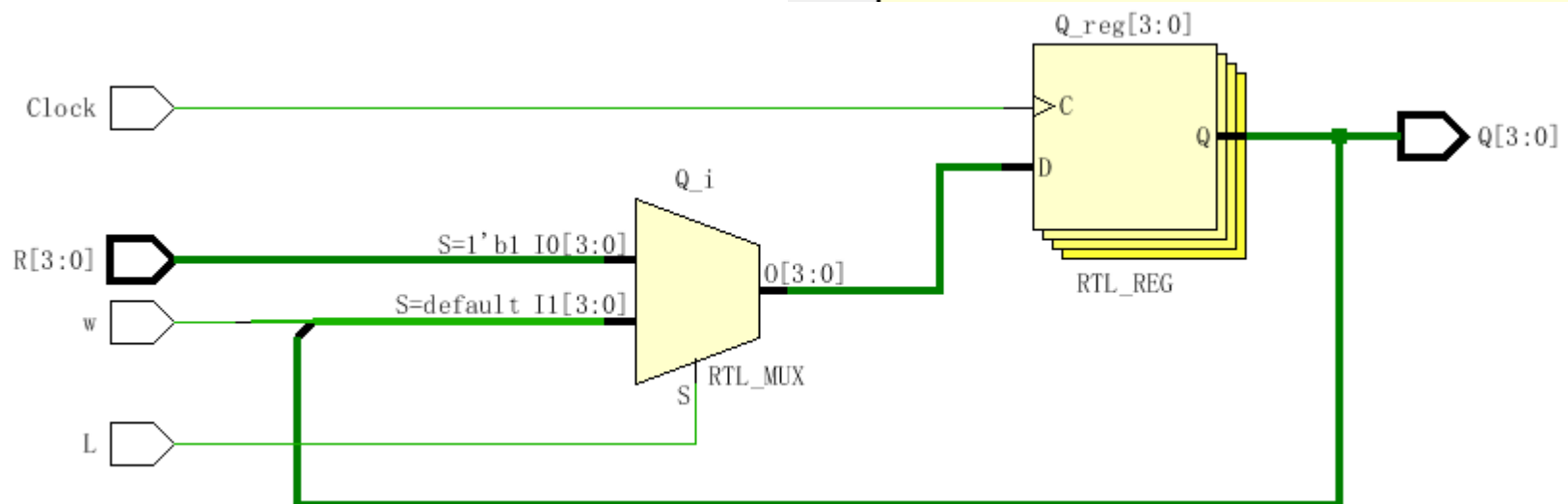
```

```

362     always @(posedge Clock)
363         if (L)
364             Q <= R;
365         else
366             begin
367                 Q[0] <= Q[1];
368                 Q[1] <= Q[2];
369                 Q[2] <= Q[3];
370                 Q[3] <= w;
371             end
372 endmodule

```

四位移位寄存器的另一种代码



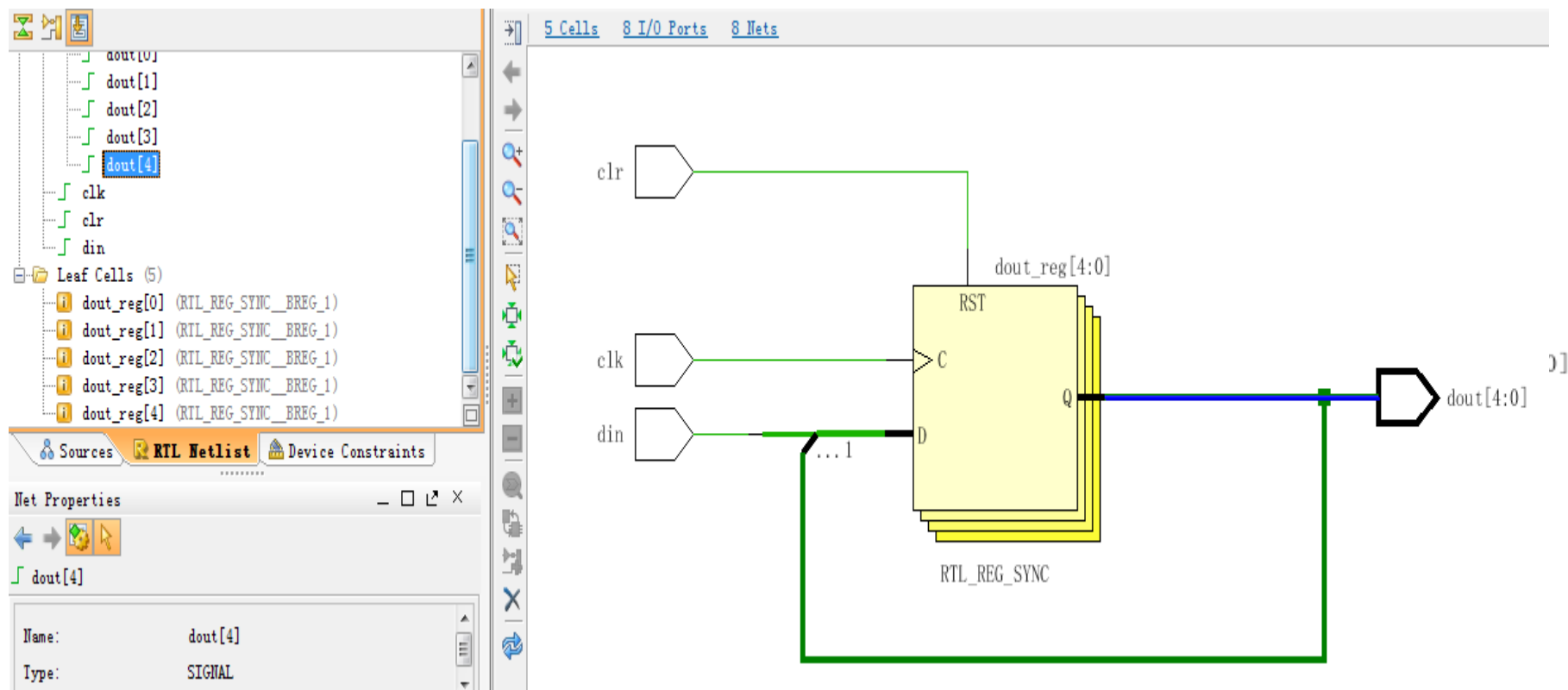
```
module sipo (dout, din,clr,clk);//串入并出
    output[4:0] dout;
    input clk, din,clr;
    reg[4:0] dout; //五位
    always @ (posedge clk )
    begin
        if(clr) begin
            dout <= 0;
        end
        else begin
            dout <= {dout, din};
        end
    end
endmodule
```



```

238 module sipo (dout, din, clr, clk); //串入并出
239     output[4:0] dout;
240     input clk, din, clr;
241     reg[4:0] dout; //五位
242     always @ (posedge clk )
243         if(clr)      dout <= 0;
244         else      dout <= {dout, din};
245 endmodule

```

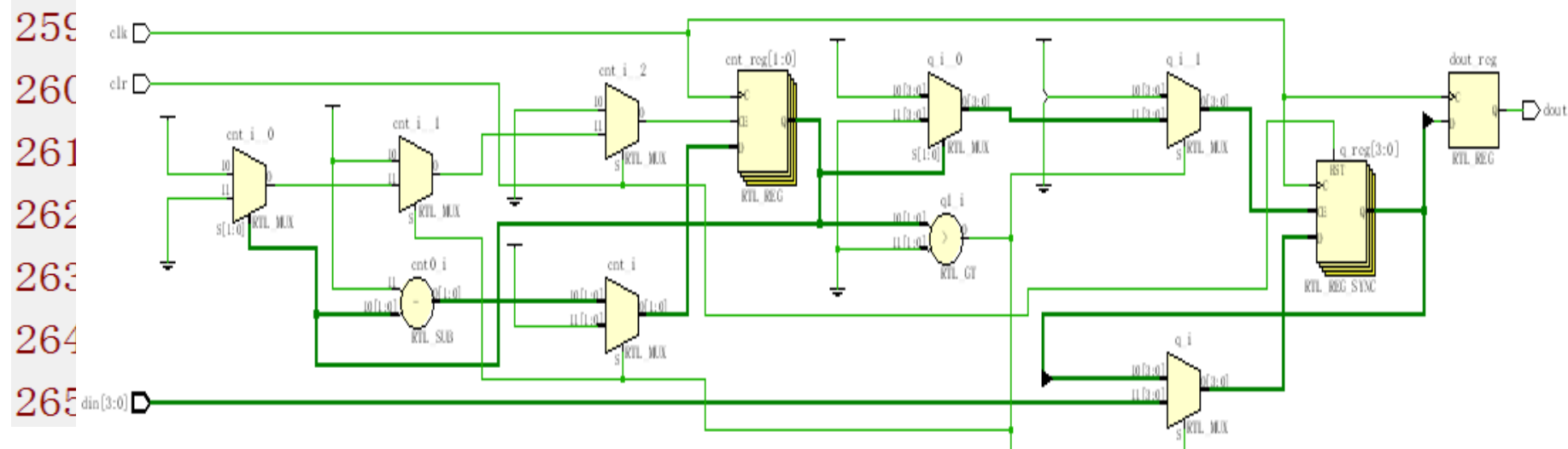


```
module piso4 ( dout, clk,clr,din);//并入串出
    output dout;//
    input clk, clr;
    input[3:0] din;
    reg dout;
    reg[1:0] cnt; //初始为0
    reg[3:0] q;
    always @(posedge clk)
    begin
        if(clr) begin q <= 4'b0000; end
        else begin if(cnt > 0) begin q[3:1] <= q[2:0];
                                cnt <= cnt -1; end
                    else if(cnt == 2'b00)
                        begin q <= din; cnt<=2'b11; end
                    end
        dout <= q[3];
    end
endmodule
```

```

247 module piso4 ( dout, clk, clr, din); //并入串出
248     output dout; //
249     input clk, clr;
250     input[3:0] din;
251     reg dout;
252     reg[1:0] cnt; //初始为0
253     reg[3:0] q;
254 always @ (posedge clk)
255     begin
256         if(clr)  q <= 4'b0000;
257         else begin if(cnt > 0)
258                     begin q[3:1] <= q[2:0];

```



```
always @ (posedge clk) //双向移位寄存器
```

```
begin
```

```
    if(left_right) begin
```

```
        q_tempt[7] <= din;
```

```
        for (i = 7; i >= 1; i = i - 1)
```

```
            begin q_tempt[i-1] <= q_tempt[i]; end
```

```
    end
```

```
    else begin
```

```
        q_tempt[0] <= din;
```

```
        for (i = 1; i <= 7; i = i + 1)
```

```
            begin q_tempt[i] <= q_tempt[i-1]; end
```

```
    end
```

```
    dout_r <= q_tempt[0];
```

```
    dout_l <= q_tempt[7];
```

```
end
```

```
endmodule
```

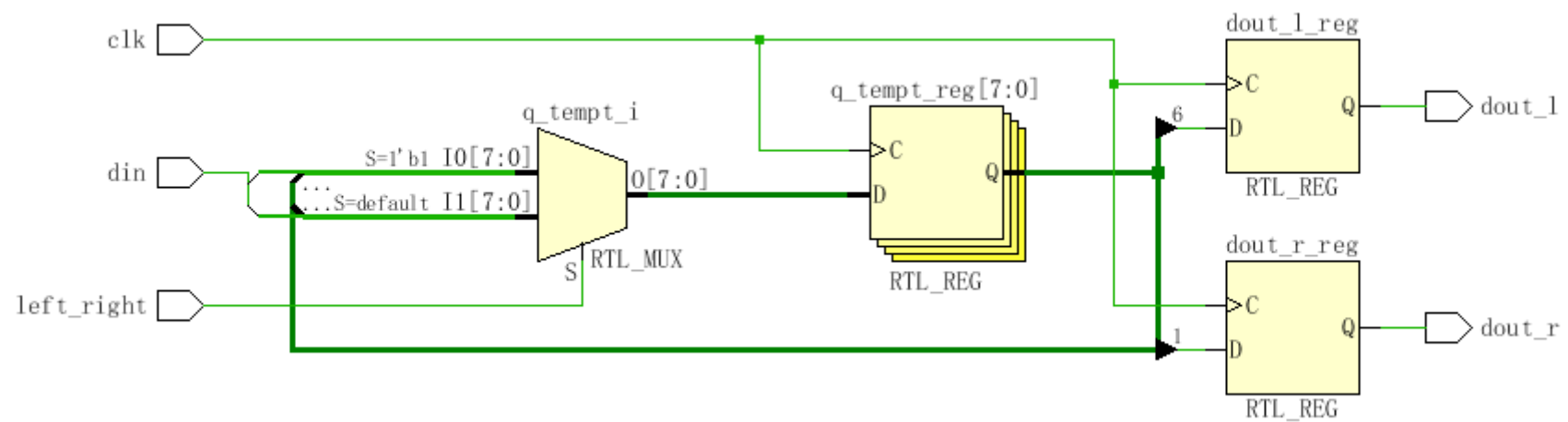
```
module d_reg (dout_l, dout_r, clk,  
              din, left_right);
```

```
    output dout_l, dout_r;
```

```
    input clk, din, left_right;
```

```
    reg dout_l, dout_r;
```

```
    reg[7:0] q_tempt; //内部有八位  
    integer i;
```



含异步复位端的D触发器

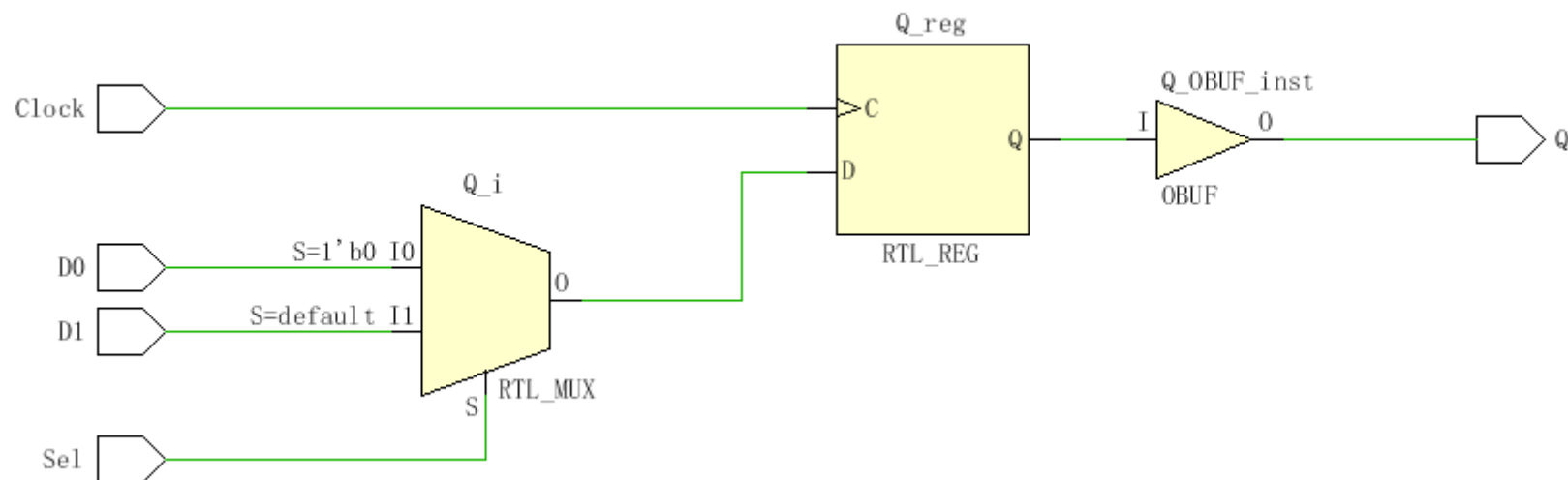
```
module flipflop (D, Clock, Resetn, Q);  
  input D, Clock, Resetn;  
  output Q;  
  reg Q;  
  always @(negedge Resetn or posedge Clock)  
    if (!Resetn)  
      Q <= 0;  
    else  
      Q <= D;  
endmodule
```

含同步复位端的D触发器

```
module flipflop (D, Clock, Resetn, Q);  
  input D, Clock, Resetn;  
  output Q;  
  reg Q;  
  always @(posedge Clock)  
    if (!Resetn)  
      Q <= 0;  
    else  
      Q <= D;  
endmodule
```

输入端有一 2选1多路器的D触发器

```
349 module muxdff (D0, D1, Sel, Clock, Q);  
350     input D0, D1, Sel, Clock;  
351     output reg Q;  
352     always @(posedge Clock)  
353         if (!Sel)  
354             Q <= D0;  
355         else  
356             Q <= D1;  
357 endmodule
```



分频器：在数字电路的设计中，分频器是一种应用十分广泛的电路，其功能就是对较高频率的信号进行分频。本质上，分频器是加法计数器的变种，其计数值由分频系数 $N=f_{in}/f_{out}$ 决定，其输出不是一般计数器的结果，而是根据分频常数对输出信号的高、低电平进行控制。通常来说，分频器常用于数字电路中的时钟信号进行分频，从而得到较低频率的时钟信号、选通信号、中断信号等。

偶数分频器

- 偶数分频器是指分频系数是偶数的分频器，分频系数 $N = 2^n$ ($n=1,2,\dots,n$)，如果输入信号的频率为 f ，则分频器的输出信号为： $f/2^n$

分频系数是2的整数次幂

```
module div248(div2,div4,div8,clk);  
    output div2, div4, div8;  
    input clk;  
    reg div2, div4, div8;  
    reg[2:0] cnt; //初始化  
    always @ (posedge clk)  
        begin  
            cnt <= cnt+1;  
            div2 <= cnt[0];  
            div4 <= cnt[1];  
            div8 <= cnt[2];  
        end  
endmodule
```

```
module clkdiv (  
    input mclk, //50MHz  
    input clr,  
    output clk190,  
    output clk48  
);  
reg [24:0] q; // 25 位计数器  
always @ (posedge mclk or posedge clr)  
    begin  
        if (clr == 1)  
            q <= 0;  
        else  
            q <= q + 1;  
        end  
        assign clk190 = q[17]; // 190 Hz  
        assign clk48 = q[19]; // 47.7 Hz  
endmodule
```

分频系数不是2 的整数次幂

```
module div6(div6, clk);  
    output div6;  
    input clk;  
    reg div6;  
    reg[2:0] cnt;  
    always @ (posedge clk)  
        begin  
            if (cnt == 3'b010) begin  
                div6 <= ~ div6;  
                cnt <= 0;  
            end  
            else begin  
                cnt <= cnt + 1;  
            end  
        end  
endmodule
```

占空比不是1: 1

```
module div6(div6, clk);  
    output div6;  
    input clk;  
    reg div6;  
    reg[2:0] cnt;  
    always @ (posedge clk)  
        begin  
            if (cnt == 3'b101) cnt <= 0;  
            else cnt <= cnt + 1;  
        end  
    always @ (posedge clk)  
        begin  
            if (cnt == 3'b000) div6 <= 0;  
            if (cnt == 2'b010) div6<=1;  
        end  
endmodule
```



always块语句

- 包含一个或一个以上的声明语句(如:过程赋值语句、任务调用、条件语句和循环语句等), 在仿真运行的全过程中, 在定时控制下被反复执行。



规则

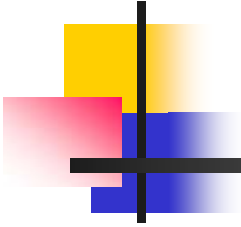
- 在always块中被赋值的只能是register型变量（如reg, integer, real, time）。
- 每个always块在仿真一开始便开始执行，当执行完块中最后一个语句，继续从always块的开头执行。

格式

always <时序控制> <语句>

注1： 如果always块中包含一个以上的语句，则这些语句必须放在begin_end或fork_join块中！

```
always @ (posedge clk or negedge clear)
begin
    if(!clear) qout = 0; //异步清零
    else      qout = 1;
end
```

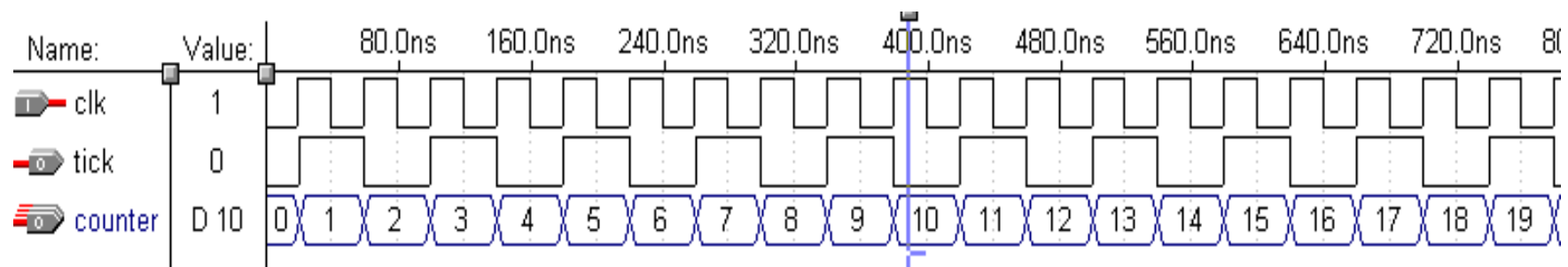
注2: always语句必须与一定的**时序控制**结合在一起才有用!
如果没有时序控制, 则易形成**仿真死锁**!

- **[例]**生成一个0延迟的无限循环跳变过程——形成仿真死锁!
always areg = ~areg;
- **[例]**在测试文件中, 用于生成一个无限延续的信号波形——时钟信号

```
'define half_period 50  
module half_clk_top;  
    reg reset, clk; // 输入信号  
    wire clk_out; // 输出信号  
    always #half_period clk = ~clk;  
    .....  
endmodule
```

- [例] 用always块语句产生T'FF和8位二进制计数器。

```
// Use always statement to generate T'FF and binary counter.
module always_demo (counter, tick, clk);
    output [7:0]    counter;
    output          tick;
    input          clk;
    reg [7:0]       counter;
    reg             tick;
    always @ (posedge clk)
    begin
        tick = ~tick;           // T'FF
        counter = counter + 1;   // binary counter
    end
endmodule
```



always块语句模板

一个变量不
能在多个
always块中
被赋值!

```
always @ (<敏感信号表达式>)
```

```
begin
```

```
// 过程赋值语句
```

```
// if语句
```

```
// case语句
```

```
// while, repeat, for循环
```

```
// task, function调用
```

```
end
```

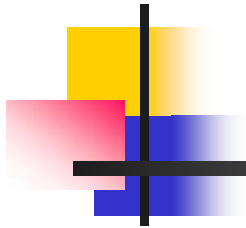
➤ 敏感信号表达式又称**事件**表达式或敏感表，当其值改变时，则执行一遍块内语句；

一般为输入

➤ 在敏感信号表达式中应列出影响块内取值的所有信号!

➤ 敏感信号可以为**单**个信号，也可为**多**个信号，中间需用关键字**or**连接!

➤ 敏感信号不要为x或z，否则会阻挡进程!



常用于描述
时序逻辑

常用于描述
组合逻辑

- always的时间控制可以为沿触发，也可为电平触发。
- 关键字posedge表示上升沿；negedge表示下降沿。

由两个沿触发的always 块

```
always@ (posedge clock or posedge reset)
begin
    .....
end
```

由多个电平触发的always 块

```
always@ (a or b or c)
begin
    .....
end
```



可综合性问题

- **always**块语句是用于综合过程的最有用的语句之一，但又常常是不可综合的。为得到最好的综合结果， **always**块程序应严格按以下模板来编写：

模板1

```
always @ (Inputs) //所有输入信号必须列出，用or隔开  
begin  
..... //组合逻辑关系  
end
```

模板2

```
always @ (Inputs) //所有输入信号必须列出，用or隔开  
if (Enable)  
begin  
..... //锁存动作  
end
```



模板5

```
always @ (posedge Clock) // Clock only
begin
    .....                // 同步动作
end
```

模板4

```
always @ (posedge Clock or negedge Reset)
// Clock and Reset only
begin
    if (! Reset)           // 测试异步复位电平是否有效
        .....            // 异步动作
    else
        .....            // 同步动作
end                        // 可产生触发器和组合逻辑
```



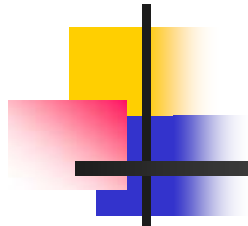
注意

❷ 当**always**块有多个敏感信号时，一定要采用**if - else if**语句，而不能采用并列的**if**语句！否则易造成一个寄存器有多个时钟驱动，将出现编译错误。

千万别写成**if**哦！

```
always @ posedge min_clk or negedge reset)
begin
    if (reset)
        min<=0;
    else if (min=8'h59) //当reset无效且min=8'h59时
        begin
            min<=0;h_clk<=1;
        end
    end
end
```

❷ 通常采用**异步**清零！只有在时钟周期很小或清零信号为电平信号时（容易捕捉到清零信号）采用同步清零。



语句的顺序执行与并行执行

内容概要

- 一、语句的顺序执行
- 二、语句的并行执行



语句的顺序执行与并行执行

一、语句的顺序执行

- 在 “always” 模块内，逻辑按书写的顺序执行。
- 顺序语句——“always” 模块内的语句。
- 在 “always” 模块内，若随意颠倒赋值语句的书写顺序，可能导致不同的结果！
- 注意阻塞赋值语句当本语句结束时即完成赋值操作！

语句的顺序执行与并行执行

[例]顺序执行模块1。

```
module serial1(q,a,clk);
```

```
    output q,a;
```

```
    input clk;
```

```
    reg q,a;
```

```
    always @(posedge clk)
```

```
        begin 对前一时刻的q值取反
```

```
            q=~q; //阻塞赋值语句
```

```
            a=~q;
```

```
        end 对当前时刻的q值取反
```

```
    endmodule
```

a和q的波形反相！

[]顺序执行模块2。

```
module serial2(q,a,clk);
```

```
    output q,a;
```

```
    input clk;
```

```
    reg q,a;
```

```
    always @(posedge clk)
```

```
        begin 对前一时刻的q值取反
```

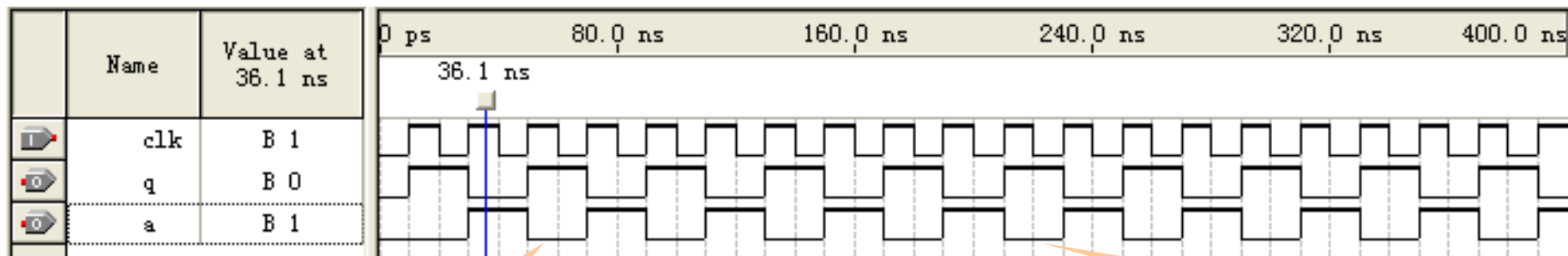
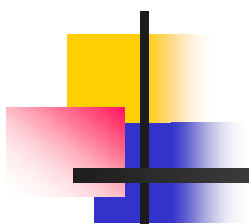
```
            a=~q;
```

```
            q=~q;
```

```
        end 对前一时刻的q值取反
```

```
    endmodule
```

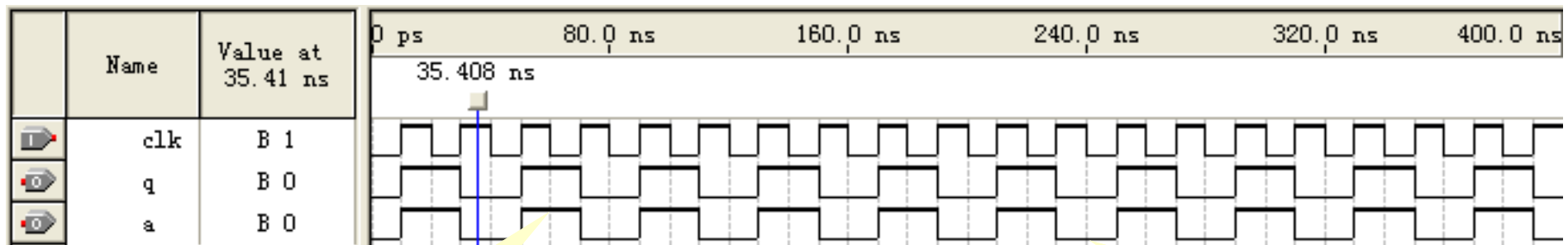
a和q的波形完全相同！



$q = \sim q;$
 $a = \sim q;$

serial1.vwf

a和q的波形反相!

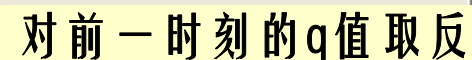


$a = \sim q;$
 $q = \sim q;$

serial2.vwf

a和q的波形完全一样!

- “always”模块、“assign”语句、实例元件都是同时（即并行）执行的！
- 它们在程序中的先后顺序对结果并没有影响。
- 下面将两条赋值语句分别放在两个“always”模块中，尽管两个“always”模块顺序相反，但仿真波形完全相同，q和a的波形完全一样。



对前一时刻的q值取反



[例]并行执行模块1。

```
module parall1(q,a,clk);  
    output q,a;  
    input clk;  
    reg q,a;  
    always @(posedge clk)  
        begin  
            q=~q;  
        end  
    always @(posedge clk)  
        begin  
            a=~q;  
        end  
endmodule
```

[例]并行执行模块2。

```
module parall2(q,a,clk);  
    output q,a;  
    input clk;  
    reg q,a;  
    always @(posedge clk)  
        begin  
            a=~q;  
        end  
    always @(posedge clk)  
        begin  
            q=~q;  
        end  
endmodule
```

- 建议:

- (1) 在进行设计前，一定要仔细分析并熟悉所需设计电路或系统的整个工作过程；合理划分功能模块；并弄清每个模块输入和输出间的逻辑关系！
- (2) 在调试过程中，仔细阅读并理解错误信息，随时查阅教材和课件上有关语法，纠正语法错误。

1. 一个变量不能在多个always块中被赋值！

- 这个问题一定要注意！否则编译不能通过。

[例1] 带异步清零、异步置位的D触发器

```
module DFF1(q,qn,d,clk,set,reset);  
    output q,qn;  
    input  d,clk,set,reset;  
    reg    q,qn;  
    always @(posedge clk or negedge set or negedge reset)  
        begin  
            if(!reset)      begin  
                q=0;qn=1;  
            end  
            else if(!set) begin  
                q=1;qn=0;  
            end  
            else            begin  
                q=d;qn=~d;  
            end  
        end  
endmodule
```

正确的写法

注：当某个变量有多个触发条件时，最好将它们放在一个always块中，并用if-else语句描述在不同触发条件下应执行的操作！

```

1 module DFF1_error(q,qn,d,clk,set,reset);
2     output q,qn;
3     input  d,clk,set,reset;
4     reg    q,qn;
5     always @(posedge clk or negedge reset )
6         begin
7             if(!reset)      begin
8                 q=0;qn=1;
9                 end
10            else            begin
11                q=d;qn=~d;
12                end
13            end
14    always @(posedge clk or negedge set)
15        begin
16            if(!set)        begin
17                q=1;qn=0;
18                end
19            else            begin
20                q=d;qn=~d;
21                end
22            end
23 endmodule

```

错误的写法

注： 这里q和qn在两个always块中都被赋值！
因为always块之间是并行操作，造成某些语句可能是互相矛盾的，所以编译器无所适从，只能报错！

- ✘ Error: Can't resolve multiple constant drivers for net "q" at DFF1_error.v(5)
- ✘ Error: Constant driver at DFF1_error.v(14)
- ✘ Error: Can't elaborate top-level user hierarchy



2. 在always块语句中，当敏感信号为两个以上的时钟边沿触发信号时，应注意不要使用多个if语句！以免因逻辑关系描述不清晰而导致编译错误。

- [例2] 在数码管扫描显示电路中，设计一个中间变量，将脉冲信号start转变为电平信号enable。

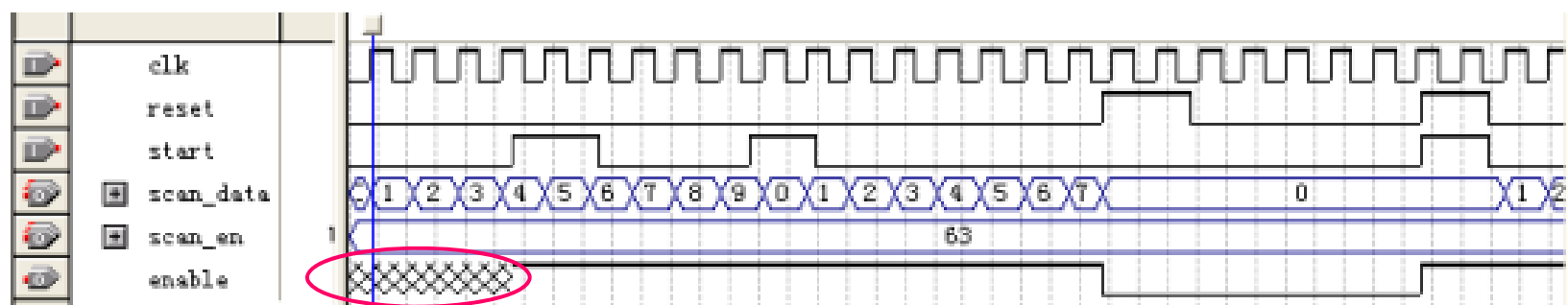
```
always@(posedge start or posedge reset)
  if (reset) enable <=0;
  if (start) enable<=1;
```

错误的写法

- 编译后出现了多条警告信息，指明在语句always
@(posedge start or posedge reset)中，变量enable不能被分配新的值！

```
Info: Running Quartus II Analysis & Synthesis
Info: Command: quartus_map --import_settings_files=on --export_settings_files=off clkscan1 -c clkscan1
Info: Found 1 design units, including 1 entities, in source file always_example2.v
Warning: Verilog HDL Always Construct warning at always_example2.v(12): variable enable may not be assigned a new vs
Warning: Verilog HDL warning at always_example2.v(22): can't infer register for Procedural Assignment in Always Cons
Warning: Verilog HDL assignment warning at always_example2.v(27): truncated value with size 5 to match size of target
```

- 其仿真波形如下：

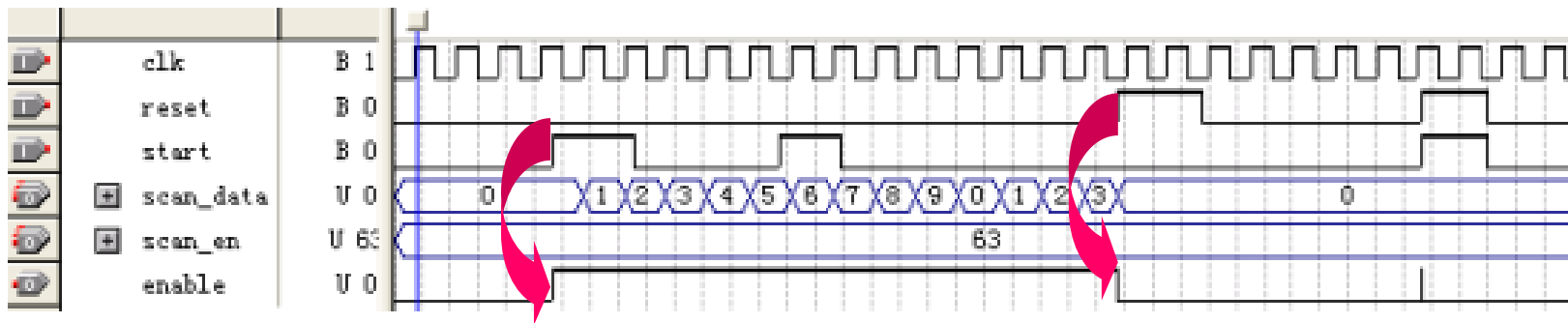


注：由于在最初一段，start和reset均为0，导致enable为不定态，则scan_data开始加1计数（正确情况应是在按下start时scan_data才开始加1计数）。当start和reset同时为1时，enable=1，则scan_data开始加1计数。

正确的写法

```
always@(posedge start or posedge reset)
  if (reset) enable <=0;
  else enable<=1;
```

- 语句“**else enable<=1;**”隐含了reset无效、且start有效的意思，因此与else if(start) enable<=1;效果一样！
- 正确的仿真波形如下：



注：可见在最初一段，当start和reset均为0时，enable被认为初值为0，则scan_data不计数，保持初值为0；一旦start有效时，则scan_data才开始加1计数。当start和reset同时为1时，先执行的是“if (reset) enable <=0;”，故enable仍为0，则scan_data保持原值0。



3. 当输出信号为总线信号时，一定要在I/O说明中指明其位宽！否则在生成逻辑符号时，输出信号被误认为是单个信号，而没有标明位宽，就不会当成总线信号。

- [例5] 声明一个位宽为5的输出信号run_cnt，其类型为reg型变量。

错误的写法

```
output run_cnt;  
reg[4:0]run_cnt;
```

正确的写法

```
output[4:0] run_cnt; //这里一定要指明位宽！  
reg[4:0]run_cnt;
```



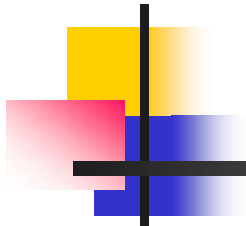
4. 当要用到计数器时，一定要根据计数最大值事先计算好所需的位宽！若位宽不够，则计数器不能计到你设定的最大值，当该计数器用作分频时，则输出时钟始终为0，所设计电路将不能按预定功能正常工作！

- [例4] 如某同学在做乐曲演奏电路实验时，对乐曲演奏子模块的仿真完全正确，high[5:0]、mid[5:0]、low[5:0]都有输出，但下载时音名显示数码管始终为000。
- 这主要是因为他分频子模块中clk_4Hz的分频用计数器count_4位宽设置不够，则clk_4Hz输出为0，故音名显示计数器high[5:0]、mid[5:0]、low[5:0]输出始终为0，电路不能正常工作。

错误的写法

```
module f20MHz_to_6MHz_4Hz(clkin,clr,clk_6M,clk_4);
    input clkin,clr;
    output      clk_6M,clk_4;
    reg         clk_6M,clk_4;
    reg[2:0]    count_6M;
    reg[15:0]   count_4;
    parameter   count_6M_width=5;
    parameter   count_4_width=5000000;
    always@(posedge clkin or posedge clr)
        begin
            if(clr) begin
                count_4=0;    clk_4=0;
            end
            else
                begin
                    if(count_4==count_4_width-1) //此条件不可能满足!
                        begin
                            count_4=0; clk_4=1;
                        end
                    else
                        begin
                            count_4=count_4+1; clk_4=0;
                        end
                end
        end
    end
endmodule
```

$2^{25}=8588608$ ，故计数器位宽应为25，应写为[22:0]。若写成[15:0]，则clk_4一直为0，则下载后数码管显示一直为0，扬声器一直是一个音调



5. 注意程序书写规范：语句应注意缩进，if-else语句注意对齐，应添加必要的注释！

6. 注意区分阻塞赋值和非阻塞赋值的区别。

- 在一个源程序中，要么都采用阻塞赋值语句，要么都采用非阻塞赋值语句，最好不要混合使用，否则可能逻辑关系出错！
- 为易于综合，建议均采用非阻塞赋值语句！