Verilog组合逻辑电路

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
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。

case语句实现

```
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
endmodule
```

```
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
```

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);
n)|(cin & b);
cin;
n3);
```

```
//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
```

4位全加

```
//数据流方式
//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
```

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<= 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！

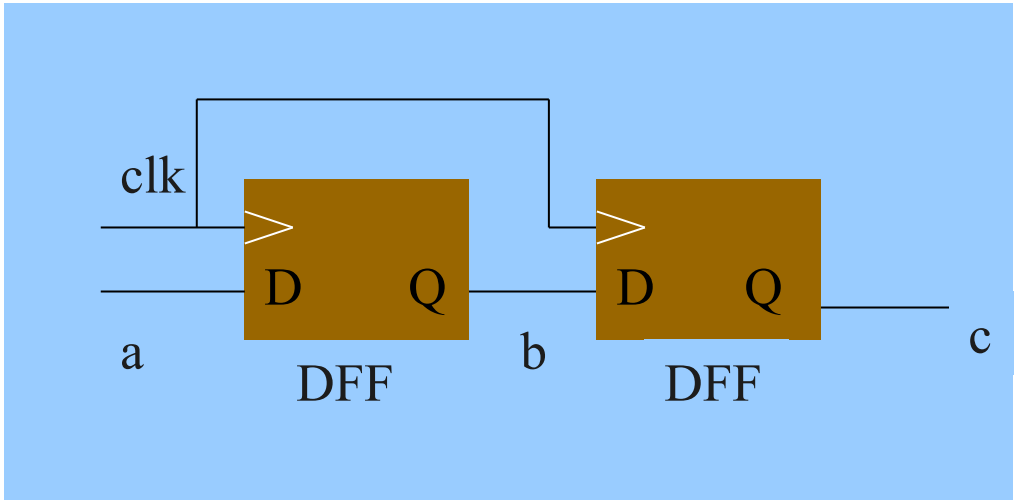
二、非阻塞赋值与阻塞赋值的区别

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！



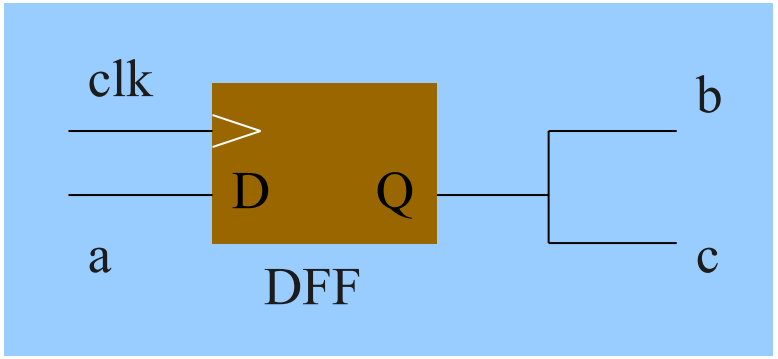
非阻塞的意思是每条赋值语句的结果直到 always块的结尾才能看到。

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！

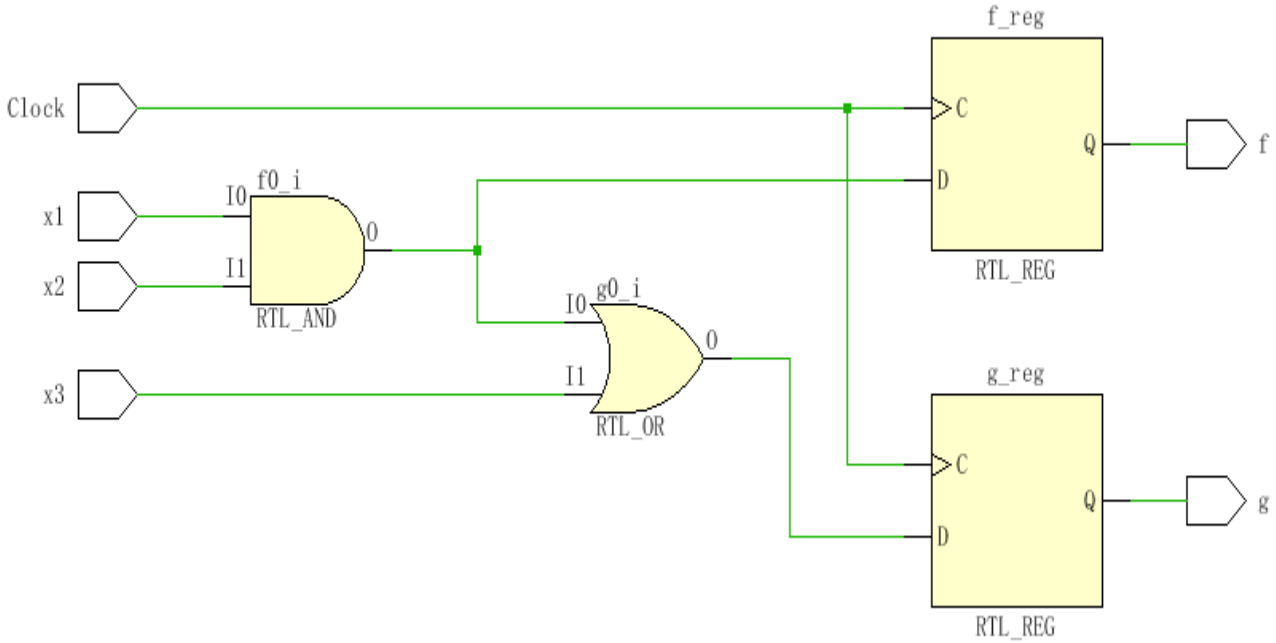


注：在一个块语句中，如果有多条阻塞赋值语句，在前面的赋值语句没有完成之前，后面的语句就不能被执行，就像被阻塞了一样，因此称为阻塞赋值方式。

这里c的值与b的值一样！

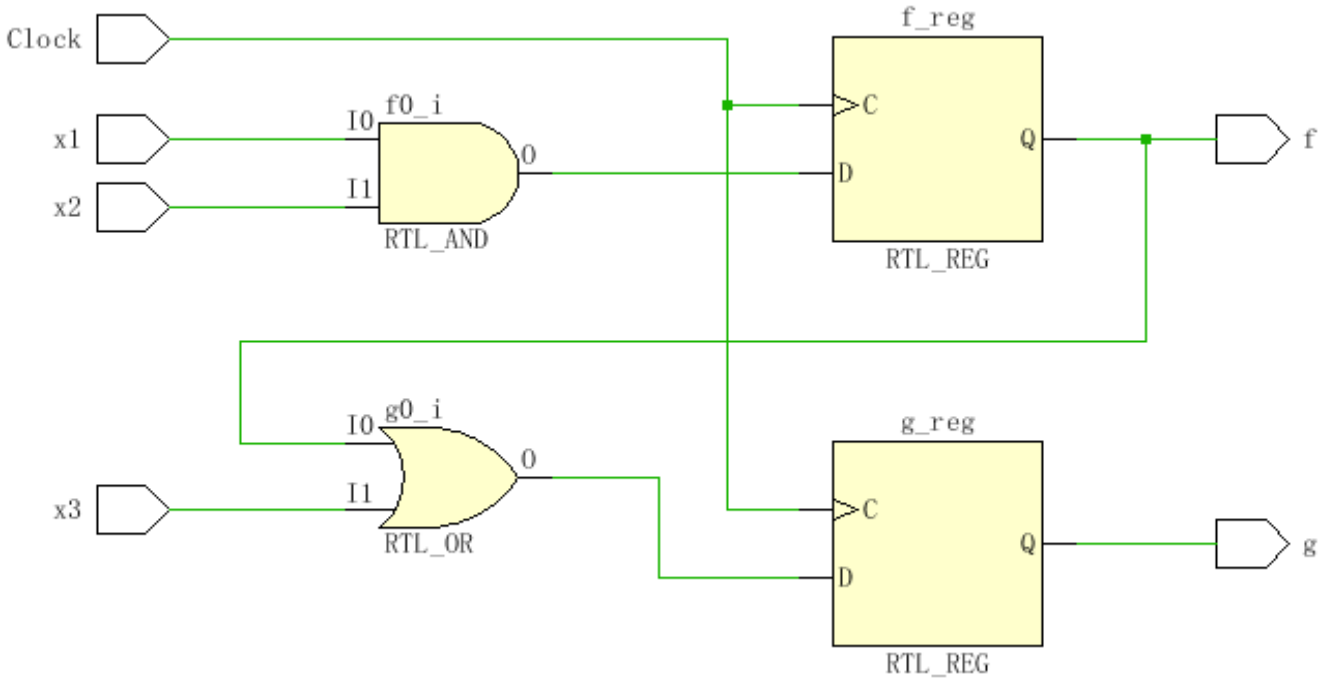
```
87 module example7_5 (x1, x2, x3, Clock, f, g);
88     input x1, x2, x3, Clock;
89     output reg f, g;
90     always @(posedge Clock)
91     begin
92         f = x1 & x2;
93         g = f | x3;
94     end
95 endmodule
```

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
```

非阻塞赋值

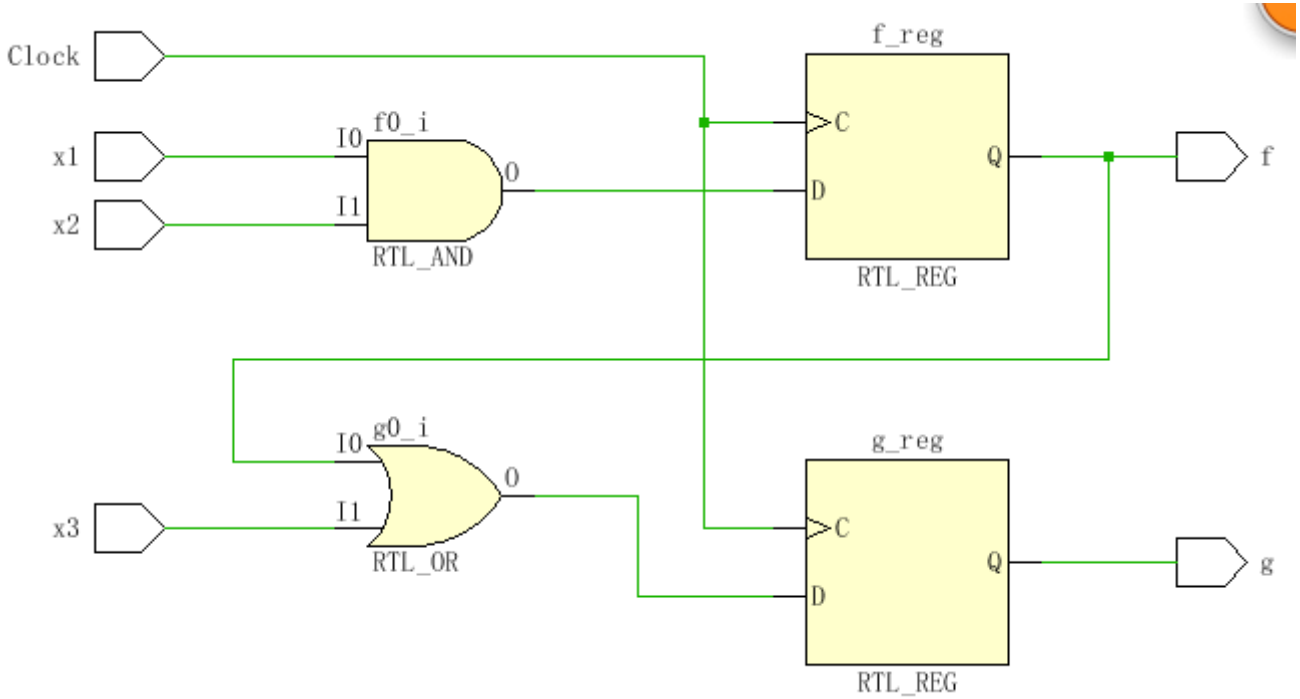


若把给f和g赋值的两条语句次序颠倒

```
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



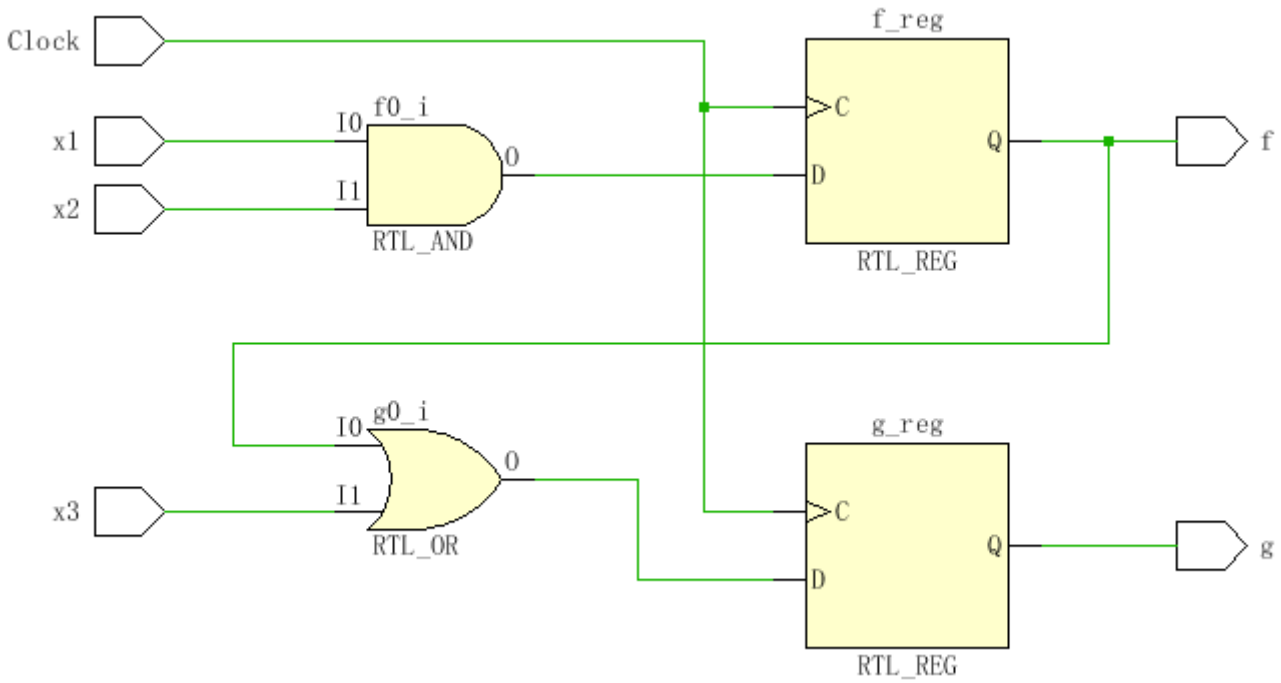
若把给f和g赋值的两条语句次序颠倒

```
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

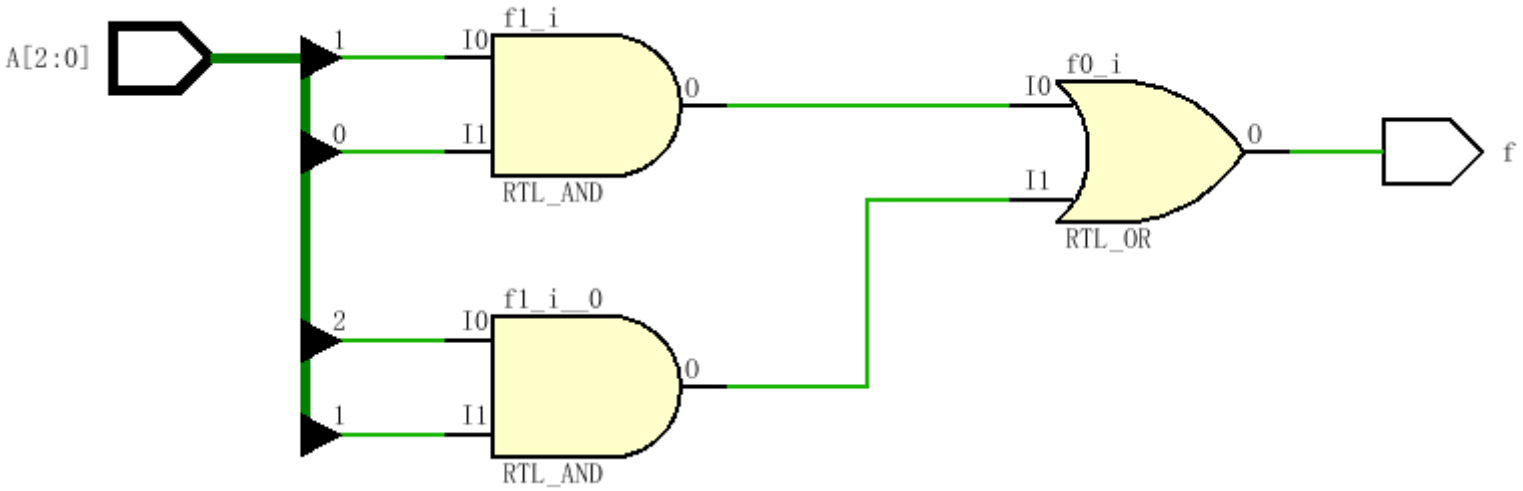
```
always @ (A)
begin
f =A[1]&A[0];
f = f|(A[2]&A[1]);
end
```

这些声明语句实现了想要的逻辑函数，就是

f=a1a0 + a2a1

```
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=a1a0 + a2a1



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 + a2a1
```

```
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=。