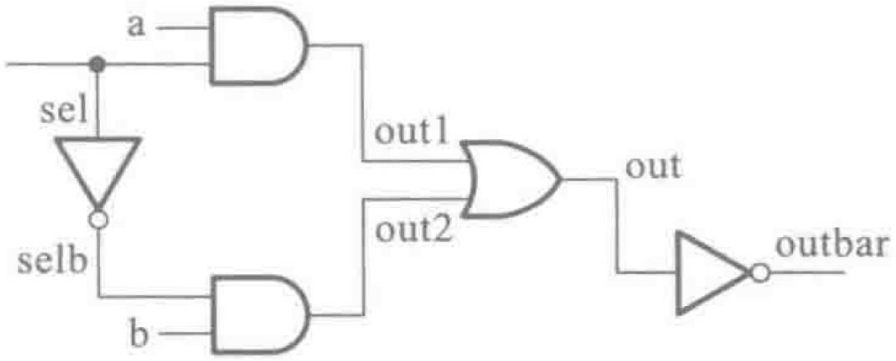
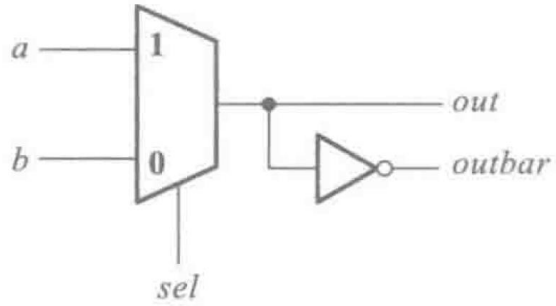
Structural Specification




```
module muxgate (a, b, out, outbar, sel);
                                        //这是一个 2 选 1 数据选择器,名为 muxgate
    input a, b, sel;                    //定义输入端口为 a, b 和 sel
    output out, outbar;                 //定义输出端口为 out 和 outbar
    wire out1, out2, selb;              //定义内部的三个连接点 out1, out2, selb
        and a1 (out1, a, sel);          //调用一个与门 a1
        not i1 (selb, sel);             //调用一个反相器 i1
        and a2 (out2, b,selb);          //调用一个与门 a2
        or o1 (out, out1, out2);        //调用一个或门 o1
        assign outbar = ~out;
endmodule
```

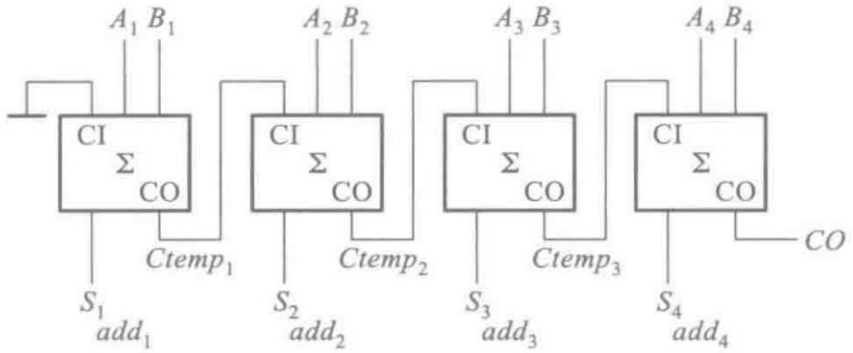
Behavioral Specification



$Out = sel \cdot a + sel' \cdot b$

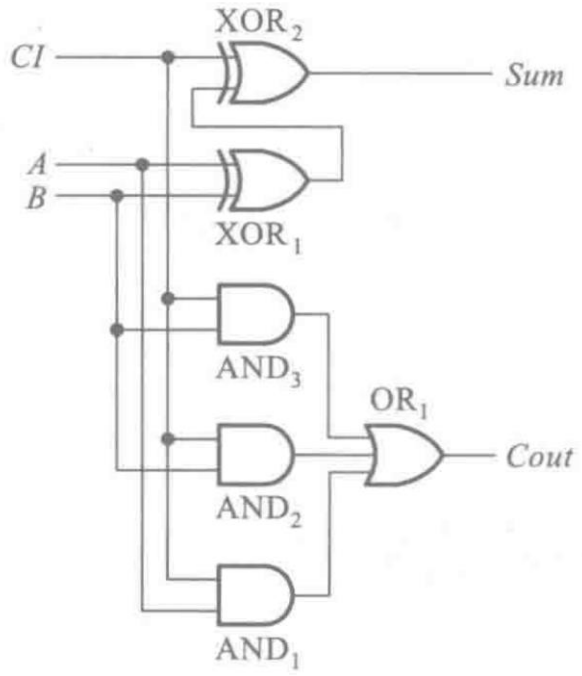
```
module mux_2_to_1(a, b, out, outbar, sel);
            //这是一个 2 选 1 数据选择器,名为 mux_2_to_1
    input a, b, sel;          //定义该模块的输入端口为 a, b 和 sel
    output out, outbar;       //定义该模块的输出端口为 out 和 outbar
    assign out = sel ? a : b; //如果 sel = 1,将 a 赋值给 out
                              //如果 sel = 0,将 b 赋值给 out
    assign outbar = ~out;     //将 out 取反后赋值给 outbar
    endmodule                 //模块描述结束
```

Combinational logic – FA




```
//对 4 位串行进位加法器的顶层结构的描述
   module Four_bit_fulladd (A, B, CI, S, CO);    //4 位全加器模块名称和端口名
      parameter size=4;                          //定义参数
      input  [size:1] A, B;
      output  [size:1] S;
      input  CI;
      output CO;
      wire  [1:size-1] Ctemp                     //定义模块内部的连接线
   onebit_fulladd                                //调用 1 位全加器
      add1(A[1], B[1], CI,S[1], Ctemp[1]),//实例化, 调用 1 位全加器
      add2(A[2], B[2], Ctemp[1], S[2], Ctemp[2]),//实例化,调用 1 位全加器
      add3(A[3], B[3], Ctemp[2], S[3], Ctemp[3]),//实例化,调用 1 位全加器
      add4(A[4], B[4], Ctemp[3], S[4], CO);//实例化 4
   endmodule                                     //结束
```

Combinational logic – FA




```
//对 1 位全加器的内部结构的描述
module onebit_fulladd (A, B, CI, Sum, Cout); //1 位全加器模块名称和端口名
    input A, B, CI;
    output Sum, Cout;
    wire Sum_temp, C_1, C_2, C_3; //定义模块内部的连接线
    xor
    XOR1( Sum_temp, A, B),
    XOR2( Sum, Sum_temp, CI); //两次调用异或门实现 Sum=A⊕B⊕CI
    and          //调用 3 个与门 AND1, AND2, AND3
    AND3( C_3, A, B),
    AND2( C_2, B, CI);
    AND1( C_1, A, CI),
    or
    OR1( Cout, C_1, C_2, C_3);
                 //调用或门实现 Cout=AB+A(CI)+B(CI)
    endmodule    //结束
```

Combinational logic – FA

```
module Four_bit_fulladd (A, B, CI, S, CO); //4 位全加器模块名称和端口名
       parameter size=4;                   //定义参数
       input  [size:1] A, B;               //定义加数和被加数的位数为 4
       output  [size:1] S;                 //定义和的位数为 4
       input  CI;
       output CO;
       assign {CO, S} = A + B + CI         //加运算后的结果为 5 位
   endmodule                                       //结束
```

Sequential logic – FF

（1）具有同步清零端 reset 的正边沿触发的 *D* 触发器 dff_sync_reset

```
module dff_sync_reset(data,clk,reset,q);          //触发器的外部封装
input data,clk,reset;                             //触发器输入信号——数据 data,
                                                  //触发时钟信号 clk,同步清零信号 reset
output q;                                         //触发器的输出信号 q
reg q;                                            //定义 q 的数据类型
always@(posedge clk)                              //开始描述功能,触发信号 clk 正边沿到达时
                                                  //完成下面的功能
    if(~reset)begin                               //首先判断同步清零信号 reset 是否为 0
    q<=1'b0;                                      //如果为 0,q 置 0
    end else begin                                //若 reset 不为 0,执行下面
    q<=data;                                      //在触发信号触发后
                                                  //且不清零时,将数据输入 data 写入 q
    end
    endmodule
```

Sequential logic – FF

（2）具有异步清零端的上升沿触发的 T 触发器

```
module tff_aync_reset(t,clk,reset,q);                    //触发器的外部封装

input t,clk,reset;                                       //触发器输入信号——数据 t
                                                         //触发时钟信号 clk,异步清零信号 reset

output q;                                                //触发器的输出信号 q
reg q;                                                   //定义 q 的数据类型
always@ (posedge clk or negedge reset)                   //开始描述功能,触发信号 clk 正边沿到达时
                                                         //或 reset 信号负边沿到达后
                                                         //完成下面的功能
if( ~reset) begin                                        //首先判断 reset 是否为 0
q<=1'b0;                                                 //如果 reset=0,则将 q 置 0
end else if(t) begin                                     //否则实现 T 触发器的功能
q<=!q;                                                   //T=1,则 q^n=q'
end
endmodule
```

Sequential logic – Counter

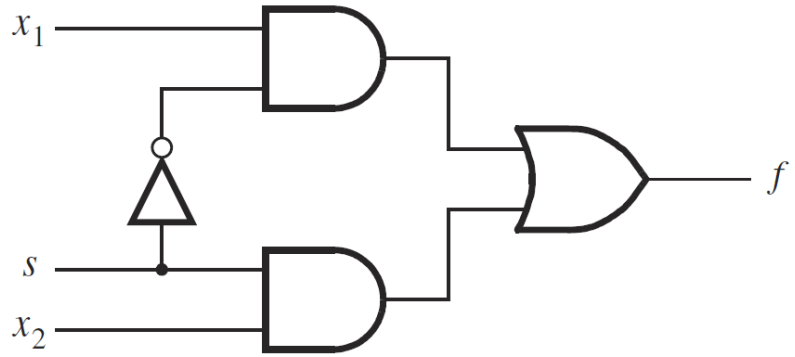
```
module counter13(clk,q,c);                    //计数器的外部封装
input clk;                                    //将计数脉冲定义为输入信号
output reg[3:0]q;                             //定义输出的数据类型,表示状态需要的状态编
                                                码的位数

output reg c;                                 //定义进位输出的数据类型
//下面是时序逻辑部分
always@(posedge clk)                          //正边沿触发
        if(q==12)                             //判断计数器是否计满
        q<=0;                                 //若计满,回到初态
      else
        q<=q+1;                               //否则,每次触发脉冲到,状态数加1
   end                                        //状态转换部分的描述结束
//下面是组合逻辑部分
always@(q)begin                               //当计数器的状态q发生变化时
if(q==12)                                     //判断计数器是否计满到状态1100
      c<=1;                                   //如果已经计满,将进位信号c置1
   else                                       //否则c为0
      c<=0;
   end                                        //组合部分的描述结束
endmodule
```

Verilog Syntax

- Verilog is case sensitive.
- Multiple statements can appear on a single line. White space
- Characters, such as SPACE and TAB, and blank lines are ignored
- A comment begins with the double slash “//” and continues to the end of the line.

Structural Specification

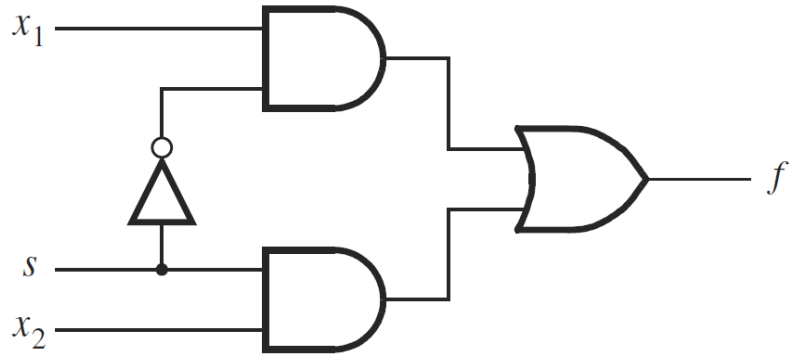



```
module  example1 (x1, x2, s, f);
    input  x1, x2, s;
    output  f;

    not (k, s);
    and (g, k, x1);
    and (h, s, x2);
    or (f, g, h);

endmodule
```

Behavioral Specification




```
module  example3 (x1, x2, s, f);
    input x1, x2, s;
    output f;

    assign  f = (~s & x1) | (s & x2);

endmodule
```

Continuous assignment – assign

```
module  example3 (x1, x2, s, f);
    input x1, x2, s;
    output f;

    assign  f = (~s & x1) | (s & x2);

endmodule
```

- Whenever any signal on the right-hand side changes its state, the value of f will be re-evaluated.

procedural statement – if-else

```
module  example5 (x1, x2, s, f);
   input  x1, x2, s;
   output f;
   reg f;

   always @(x1 or x2 or s)
      if (s == 0)
         f = x1;
      else
         f = x2;

endmodule
```

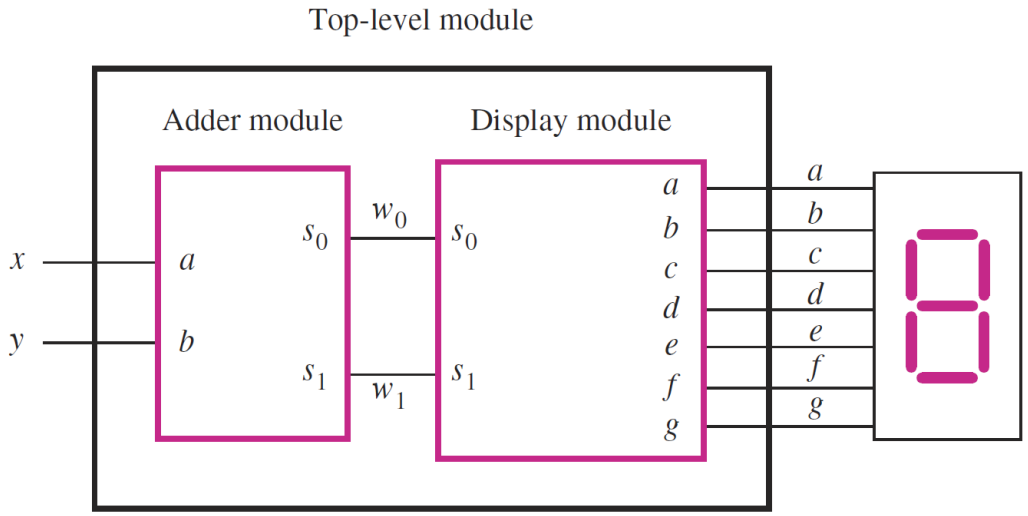
```
module  example5 (input x1, x2, s, output reg f);

  always @(x1, x2, s)
     if (s == 0)
        f = x1;
     else
        f = x2;

endmodule
```

- Procedural statements be contained inside a construct called an always block
- after the @ symbol, in parentheses, is called the sensitivity list.
- The statements inside an always block are executed by the simulator only when one or more of the signals in the sensitivity list changes value
- If a signal is assigned a value using procedural statements, then Verilog syntax requires that it be declared as a variable reg.
- An important property of the always block is that the statements it contains are evaluated in the order given in the code. This is in contrast to the continuous assignment statements, which are evaluated concurrently and hence have no meaningful order.

Hierarchical Verilog Code




```
// An adder module
module  adder (a, b, s1, s0);
    input a, b;
    output s1, s0;

    assign s1 = a & b;
    assign s0 = a ^ b;

endmodule
```

```
// A module for driving a 7-segment display
module display (s1, s0, a, b, c, d, e, f, g);
    input s1, s0;
    output a, b, c, d, e, f, g;

    assign a = ~s0;
    assign b = 1;
    assign c = ~s1;
    assign d = ~s0;
    assign e = ~s0;
    assign f = ~s1 & ~s0;
    assign g = s1 & ~s0;

endmodule
```

```
module adder_display (x, y, a, b, c, d, e, f, g);
    input x, y;
    output a, b, c, d, e, f, g;
    wire w1, w0;

    adder U1 (x, y, w1, w0);
    display U2 (w1, w0, a, b, c, d, e, f, g);

endmodule
```

Reading materials

- Fundamentals of Digital Logic with Verilog Design-Stephen Brown, Zvonko Vranesic
- Section 4.7 & 6.5.2 of 阎石 book