System Verilog入门

System Verilog

- 硬件描述语言
- Verilog + c++
- 具有很强的电路描述与建模能力，能从多个层次对数字系统进行描述和建模。

System Verilog

- 区分大小写
- 用 **//** 进行单行注释,用 **/*** 与 ***/** 进行跨行注释
- 标识符（identifier）可以是任意一组字母、数字、**$** 符号和 **_**(下划线)符号的合，但标识符的第一个字符必须是字母或者下划线，不能以数字或者**$** 符开始

System Verilog

- 四种基本的值来表示硬件电路中的电平逻辑：
  - 0：逻辑 0 或 "假"
  - 1：逻辑 1 或 "真"
  - x 或 X：未知
  - z 或 Z：高阻
- 整数数值表示方法
  - 十进制('d 或 'D)，十六进制('h 或 'H)，二进制（'b 或 'B），八进制（'o 或 'O）
  - 4'b1011　　*// 4bit 数值*
  - 32'h3022_c0de　*// 32bit 的数值*

数据类型

- wire
  - 表示硬件单元之间的物理连线，由其连接的器件输出端连续驱动
- reg
  - 用来表示存储单元，它会保持数据原有的值，直到被改写
- logic

连续赋值

- 用于对 wire 型变量进行赋值
- assign　LHS_target = RHS_expression ；
  - LHS_target 必须是一个标量或者线型向量，而不能是寄存器类型。
  - RHS_expression 的类型没有要求，可以是标量或线型或存器向量，也可以是函数调用。
  - 只要 RHS_expression 表达式的操作数有事件发生（值的变化）时，RHS_expression 就会立刻重新计算，同时赋值给 LHS_target。

```
module converter(
    input  wire key,
    output wire led
);
    assign led = ~key;
endmodule
```

always 语句

- always 语句是重复执行的。always 语句块从 0 时刻开始执行其中的行为语句；当执行完最后一条语句后，便再次执行语句块中的第一条语句，如此循环反复。
- always_comb适用于组合逻辑电路

```
module sevenseg(
    input  logic[3:0] address,
    output logic[6:0] data
);

always_comb
     case(address)
          4'b0000  : data = 7'b1111111;
          default  : data = 7'b0000000;
     endcase
endmodule
```

Verilog模块的模板

- **module <**顶层模块名>（<输入输出端口列表>）;
- **output** 输出端口列表；//输出端口声明
- **input** 输入端口列表；//输入端口声明
- /*定义数据，信号的类型，函数声明*/
- **reg** 信号名;
- //逻辑功能定义
- **assign <**结果信号名>=<表达式>； //使用**assign**语句定义逻辑功能
- //用**always**块描述逻辑功能
- **always @ (<**敏感信号表达式>）
- **begin**
- //过程赋值
- **//if-else，case**语句
- **//while，repeat，for**循环语句
- **//task，function**调用
- **end**
- //调用其他模块
- <调用模块名**module_name > <**例化模块名>（<端口列表**port_list >);**
- //门元件例化
- 门元件关键字 <例化门元件名>（<端口列表**port_list>);**
- **endmodule**

数据类型

- Verilog中的变量分为如下两种数据类型：
  - net型
  - variable型
- net型中常用的有wire, tri;
- variable型包括reg, integer等;

net型

- net型数据相当于硬件电路中的各种物理连线，其特点是输出的值紧跟输入值的变化而变化。对连线型有两种驱动方式，一种方式在结构描述中将其连到一个门原件或模块的输出端；另一种方式是用持续赋值语句assign对其进行赋值
- wire是最常用的net型变量
- wire型变量的定义格式如下：
- wire数据名1，数据名2，……数据名n;
- 例如：wire a,b;　　//定义了两个wire型变量a和b

Variable型

- variable型变量必须放在过程语句（如initial, always)中，通过过程赋值语句赋值；在always, inital等过程块内被赋值的信号也必须定义成variable型
  - 注意：variable型变量并不意味着一定对应硬件上的一个触发器或寄存器等存储元件，在综合器进行综合的时候，variable型变量会根据具体情况来确定是映射成连线还是映射为触发器或者寄存器
- reg 型变量是最常用的一种variable型变量
- reg 数据名1，数据名2，……数据名n;
- 例如：reg a,b;　//定义了两个reg型变量a和b

向量

- 标量与向量
- 宽度为1位的变量称为标量，如果在变量声明中没有指定位宽，则默认为标量（1位）。举例如下：
  - wire a; //a为标量
  - reg clk; //clk为标量reg型变量
- n线宽大于1位的变量（包括net型和variable型）称为向量（vector）。向量的宽度用下面的形式定义：
  - [msb : lsb]
- 比如：
  - wire[3:0] bus; //4位的总线

位选择和域选择

- 在表达式中可任意选中向量中的一位或相邻几位，分别称为位选择和域选择，例如：
- A=mybyte[6]; //位选择
- B=mybyte[5:2]; //域选择
- 再比如：
- reg[7:0] a,b; reg[3:0] c; reg d;
- d=a[7]&b[7]; //位选择
- c=a[7:4]+b[3:0]; //域选择

运算符（1）

- 1 算术运算符（Arithmetic operators）
  - 常用的算术运算符包括：
  - + 加
  - - 减
  - * 乘
  - /
- 2．逻辑运算符（Logical operators）
  - && 逻辑与
  - || 逻辑或
  - ! 逻辑非

运算符（2）

- 3. 位运算符（Bitwise operators）
  - 位运算，即将两个操作数按对应位分别进行逻辑
  - 运算。
  - ~ 按位取反
  - & 按位与
  - | 按位或
  - ^ 按位异或
  - ^~,~^按位同或（符号^~与~^是等价的）
- 4. 关系运算符（Relational operators）
  - < 小于
  - <= 小于或等于
  - > 大于
  - >= 大于或等于

运算符（3）

- 5．等式运算符（Equality Operators）
  - == 等于
  - != 不等于
  - === 全等
  - !== 不全等
- 6．缩位运算符（Reduction operators）
  - & 与
  - ~& 与非
  - | 或
  - ~| 或非
  - ^ 异或
  - ^~,~^ 同或
- 7．移位运算符（shift operators）
  - >> 右移
  - << 左移

运算符（4）

- 8．条件运算符（conditional operators）

  ？：

  – 三目运算符，其定义方式如下：signal=condition?true_expression:false_expression;
  – 即：信号=条件?表达式1:表达式2;当条件成立时，信号取表达式1的值，反之取表达式2的值。
- 9．位拼接运算符（concatenation operators）

  {}

  – 该运算符将两个或多个信号的某些位拼接起来。{信号1的某几位，信号2的某几位，……，信号n的某几位}

运算符的优先级

运 算 符	优先级
! ~	高优先级
* / %	
+ -	
<< >>	
< <= > >=	
== != === !==	
& ~&	
^ ^~	
| ~|	
&&	
||	
?:	低优先级

过程语句

- initial
- always
- 在一个模块（module）中，使用initial和always语句的次数是不受限制的。initial语句场用于仿真中的初始化，initial过程块中的语句仅执行一次；always块内的语句则是不断重复执行的。

always过程语句使用模板

- always@(<敏感信号表达式event-expression>)
- begin
- //过程赋值
- //if-else, case, casex, casez选择语句
- //while, repeat, for循环
- //task, function调用
- end
- “always”过程语句通常是带有触发条件的，触发条件写在敏感信号表达式中，只有当触发条件满足时，其后的”begin-end”块语句才能被执行。

敏感信号表达式

- 若有两个或两个以上信号时，使用or连接
- 例如：
- @(a)
- @(a or b)
- @(posedge clock)
- @(negedge clock)
- @(posedge clk or negedge reset)

敏感信号列表举例

- 4选1数据选择器

```
module mux4_1(out,in0,in1,in2,in3,sel);
output out;
input in0,in1,in2,in3;
input[1:0] sel; reg out;
always @(in0 or in1 or in2 or in3 or sel)        //敏感信号列表
   case(sel)
      2'b00: out=in0;
      2'b01: out=in1;
      2'b10: out=in2;
      2'b11: out=in3;
      default: out=2'bx;
   endcase
endmodule
```

posedge和negedge关键字

- 对于时序电路，事件通常是由时钟边沿触发的，为表达边沿这个概念，Verilog提供了posedge和negedge关键字来描述
- 同步置数、同步清零的计数器

```
module count(out,data,load,reset,clk);
output[7:0] out;
input[7:0] data;
input load,clk,reset;
reg[7:0] out;
always @(posedge clk)              //clk上升沿触发
begin
   if(!reset) out=8'h00;           //同步清0，低电平有效
   else if(load) out=data;         //同步预置
   else out=out+1;                 //计数
end
endmodule
```

块语句

- 块语句是由块标识符begin-end或fork-join界定的一组语句，当块语句只包含一条语句时，块标识符可以缺省
- begin-end串行块中的语句按串行方式顺序执行，比如
  - begin
  - regb=rega;
  - regc=regb;
  - end
- 由于begin-end块内的语句顺序执行，在最后，将regb,regc的值都更新为rega的值，该begin-end块执行完后，regb,regc的值是相同的。

赋值语句

- 持续赋值语句（continuous assignments）

- assign为持续赋值语句，主要用于对wire型变量的赋值。
- 比如： assign c=a&b;
- 在上面的赋值中，a,b,c三个变量皆为wire型变量，a和b信号的任何变化，都将随时反映到c上来

过程赋值语句

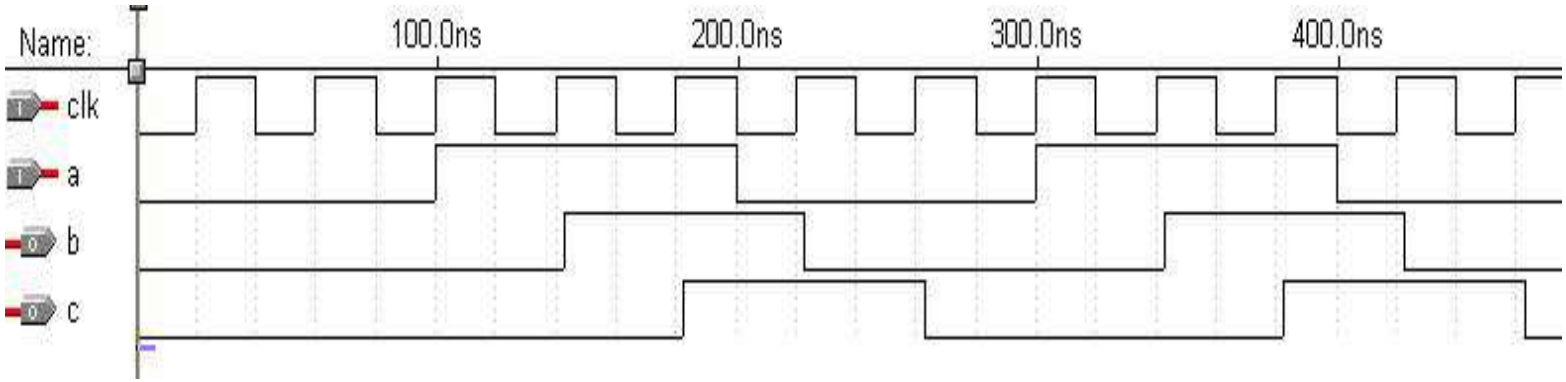
- 过程赋值语句多用于对reg型变量进行赋值。
- （1）非阻塞（non_blocking）赋值方式
- 赋值符号为”<=“，比如：b<=a; 非阻塞赋值在整个过程块结束时才完成赋值操作，即b的值并不是立刻就改变的。
- （2）阻塞（blocking）赋值方式
- 赋值符号为”=”，如：b=a; 阻塞赋值语句在该语句结束时就立即完成赋值操作，即b的值在该条语句结束后立刻改变。如果在一个块语句中，有多条阻塞赋值语句，那么在前面的赋值语句没有完成之前，后面的语句不能被执行，仿佛被阻塞了（blocking）一样，因此被称为是阻塞赋值方式。

阻塞赋值与非阻塞赋值

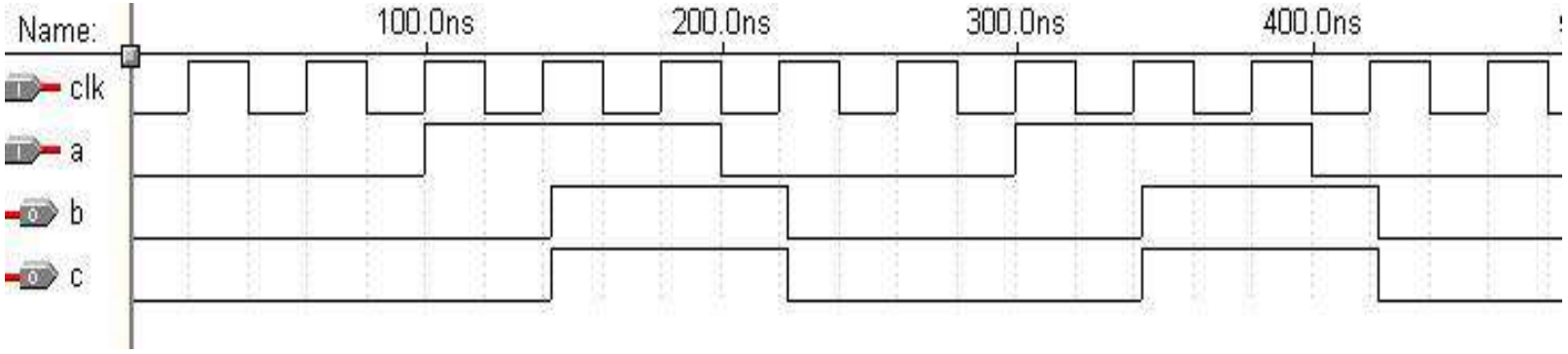
```
module
non_block(c,b,a,clk);
   output c,b;
   input clk,a;
   reg c,b;
   always @(posedge clk)
   begin
      b<=a;
      c<=b;
   end
endmodule
```

```
module block(c,b,a,clk);
   output c,b;
   input clk,a;
   reg c,b;
   always @(posedge clk)
   begin
      b=a;
      c=b;
   end
endmodule
```

阻塞赋值与非阻塞赋值的仿真波形

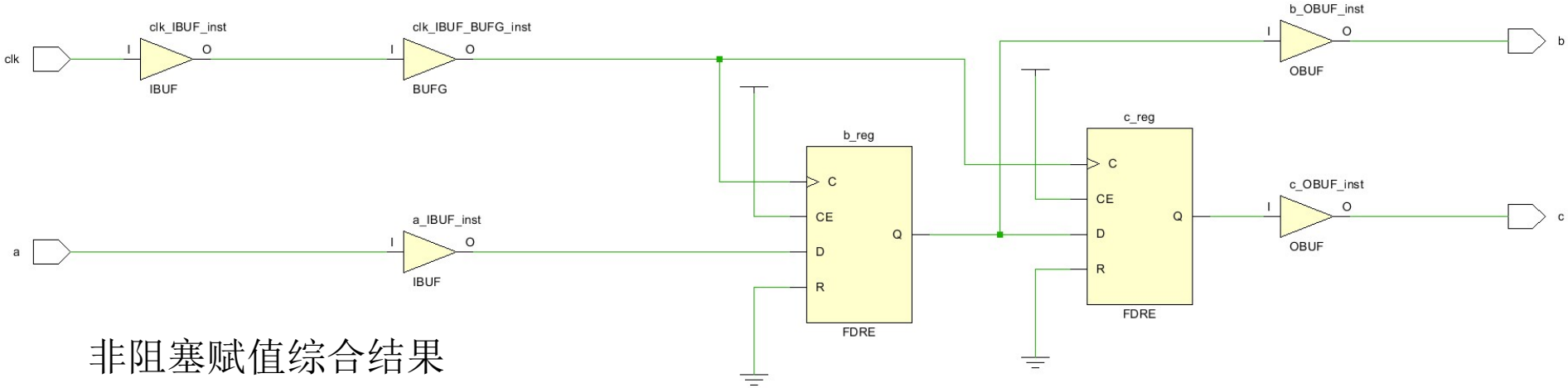


非阻塞赋值仿真波形图

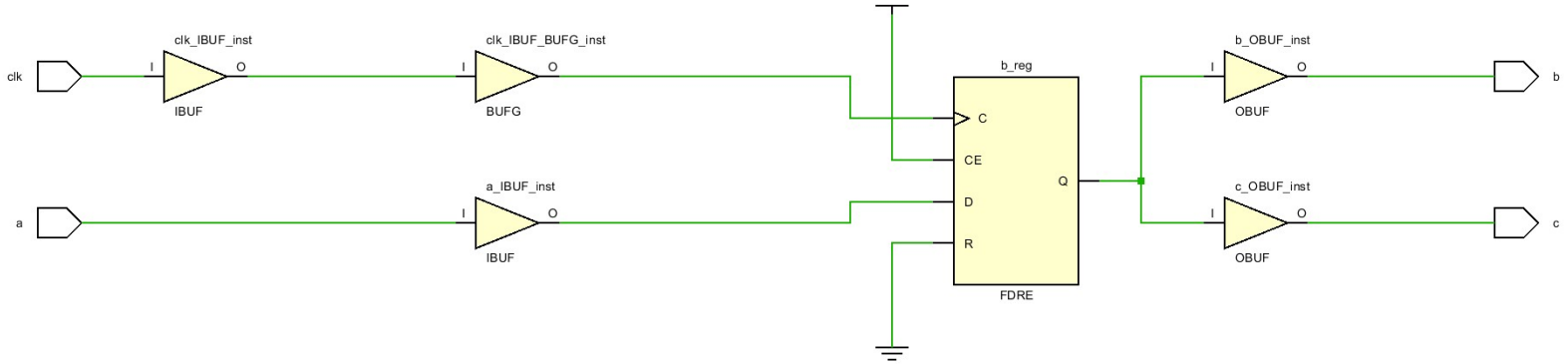


阻塞赋值仿真波形图

阻塞赋值和非阻塞赋值的综合结果



非阻塞赋值综合结果



阻塞赋值综合结果

条件语句

- if-else语句使用方法有以下3种
- （1）if（表达式） 语句1;
- （2）if（表达式） 语句1;
- else 语句2;
- （3）if （表达式1）语句1;
- else if （表达式2） 语句2;
- else if （表达式3） 语句3;
- ……..
- else if （表达式n） 语句n;
- else 语句n+1;

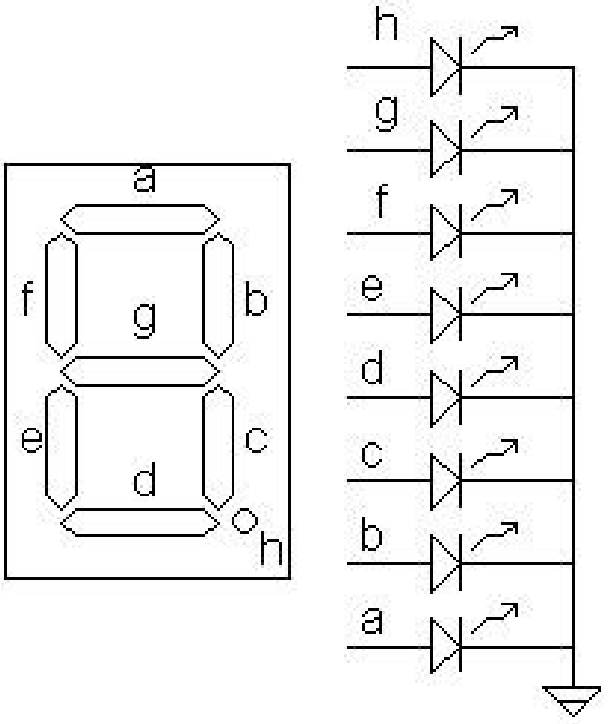
case语句

- case语句的使用格式如下
- case (敏感表达式)
- 值1：语句1；
- 值2：语句2；
- ……
- 值n：语句n;
- default：语句n+1
- endcase

BCD码—七段数码管显示译码器

```
module decode4_7(decodeout,indec);
output[6:0] decodeout;
input[3:0] indec; reg[6:0] decodeout;
always @(indec)
begin case(indec) //用case语句进行译码
   4'd0:decodeout=7'b1111110;
   4'd1:decodeout=7'b0110000;
   4'd2:decodeout=7'b1101101;
   4'd3:decodeout=7'b1111001;
   4'd4:decodeout=7'b0110011;
   4'd5:decodeout=7'b1011011;
   4'd6:decodeout=7'b1011111;
   4'd7:decodeout=7'b1110000;
   4'd8:decodeout=7'b1111111;
   4'd9:decodeout=7'b1111011;
   default: decodeout=7'bx;
endcase end
endmodule
```

p018.v



循环语句

- 在verilog中存在四种类型的循环语句，用来控制语句的执行次数。这四种语句分别为：
- （1）forever：连续执行语句；多用在initial块中，以生成时钟等周期性波形。
- （2）repeat：连续执行一条语句n次。
- （3）while：执行一条语句直到某个条件不满足
- （4）for：有条件的循环语句

```
initial
  begin
  for(i=0;i<4;i
  =i+1)
  out =out +1;
end
```

```
initial
begin
repeat(5)
    out = out +1;
end
```

```
initial
begin
  i=0;
  while(i<0)
    i = i + 1 ;
end
```

for语句

- for语句的使用格式如下（同C语言）：
- for (表达式1；表达式2；表达式3）语句；
- 即: for（循环变量赋初值；循环结束条件；循环变量增值）执行语句；

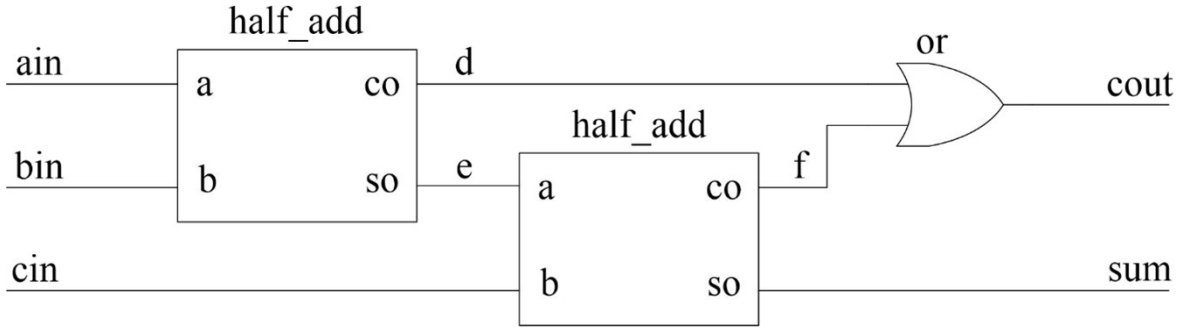
用for语句描述七人投票表决器

```
module voter7(pass,vote);
output pass;
input[6:0] vote;
reg[2:0] sum;
integer i;
reg pass;

always @(vote)
begin
   sum=0;
   for(i=0;i<=6;i=i+1)                    //for语句
      if(vote[i]) sum=sum+1;
   if(sum[2]) pass=1;                     //超过4人赞成，则通过
   else pass=0;
end
endmodule
```

模块例化方式设计的1位全加器

```
module full_add(ain,bin,cin,sum,cout);
   input ain,bin,cin;
   output sum,cout;
   wire d,e,f; //用于内部连接的节点信号

   half_add u1(ain,bin,e,d);
   //半加器模块调用，采用位置关联方式
   half_add u2(e,cin,sum,f);
   or u3(cout,d,f); //或门调用
endmodule
```

```
module half_add(a,b,so,co);
   input a,b;
   output so,co;

   assign co=a&b;
   assign so=a^b;
endmodule
```

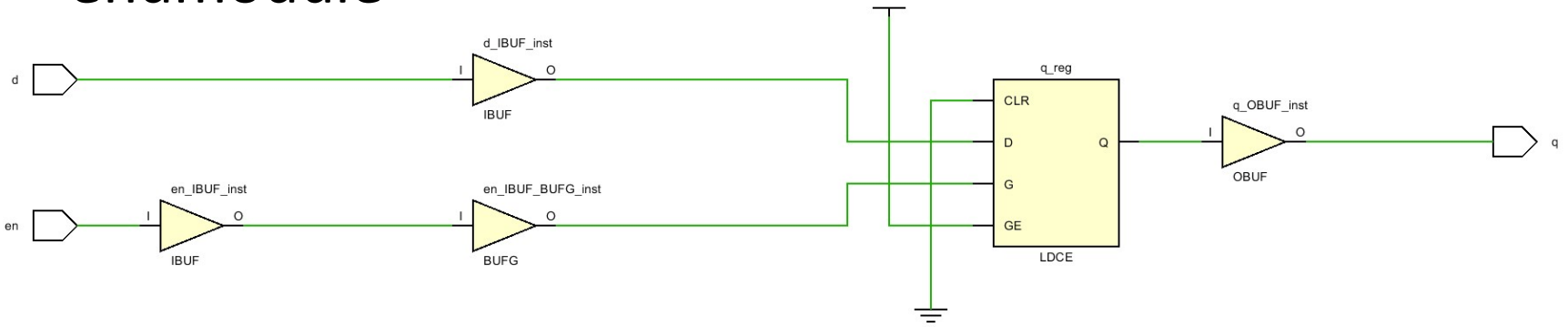


时序逻辑

- D锁存器

```
module dlatch(q,d,en);
output logic q;
input logic d,en;

always_latch
    if(en)
        q<=d;
endmodule
```

•LDCE：门控信号+异步复位（复位值为0）
•LDPE：门控信号+异步置位（置位值为1）

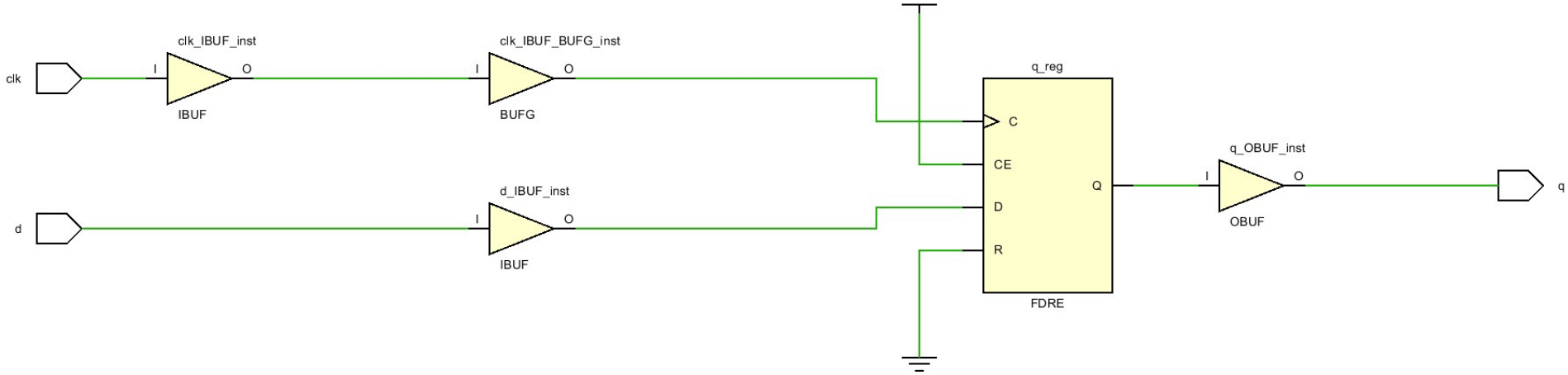


时序逻辑

- 触发器

```
module dff(q,d,clk);
 output logic q;
 input logic d,clk;

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

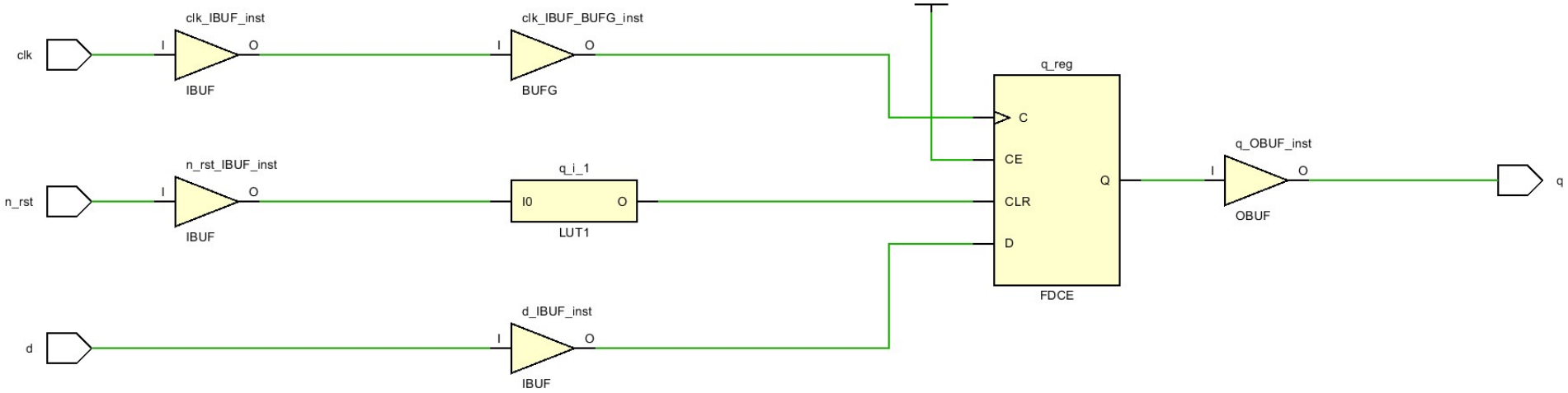
- FDCE：时钟使能+异步复位（复位值为0）
- FDPE：时钟使能+异步置位（置位值为1）
- FDRE：时钟使能+同步复位（复位值为0）
- FDSE：时钟使能+同步置位（置位值为1）



时序逻辑

- 异步复位触发器

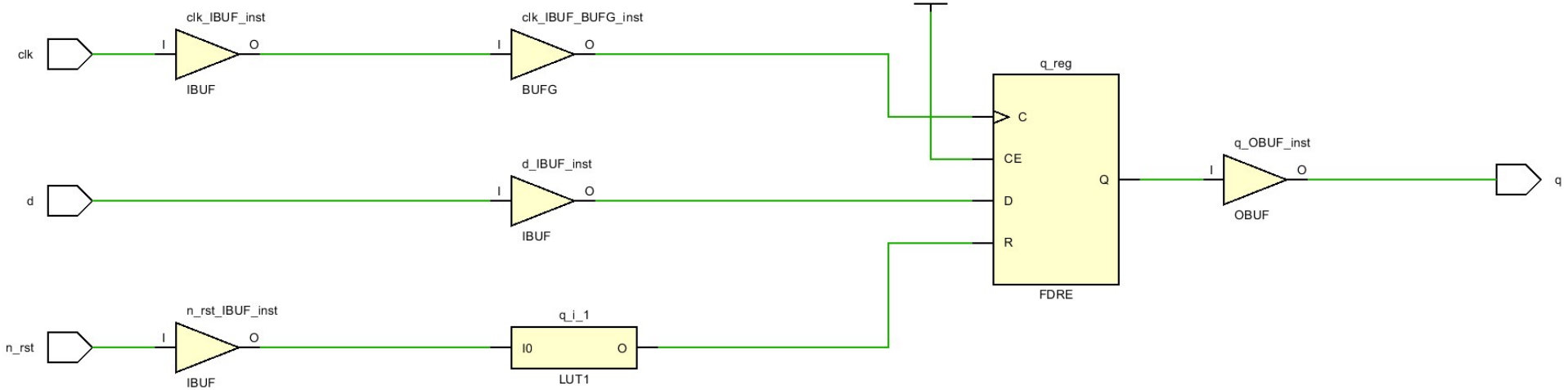
```
module dffr(q,d,clk,n_rst);
  output logic q;
  input logic d,clk,n_rst;

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



时序逻辑

- 同步复位触发器

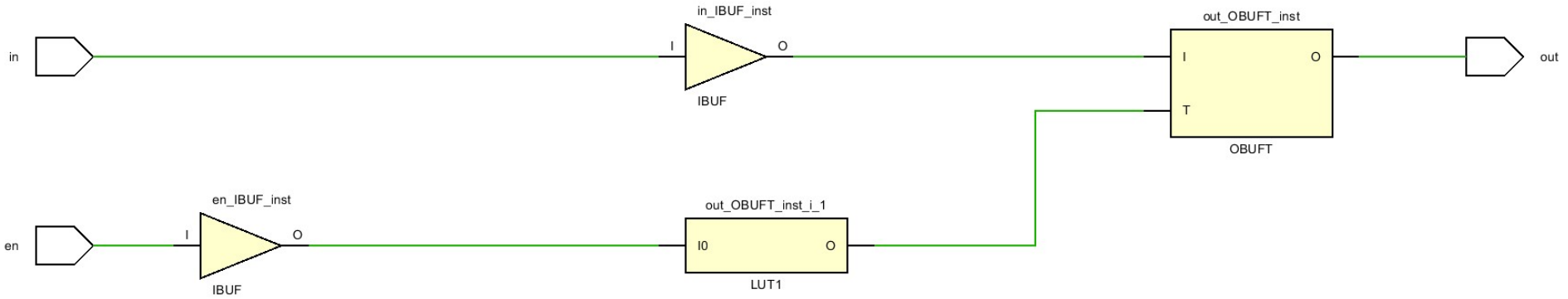
```
module dffsr(q,d,clk,n_rst);
  output logic q;
  input logic d,clk,n_rst;

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



三态逻辑设计

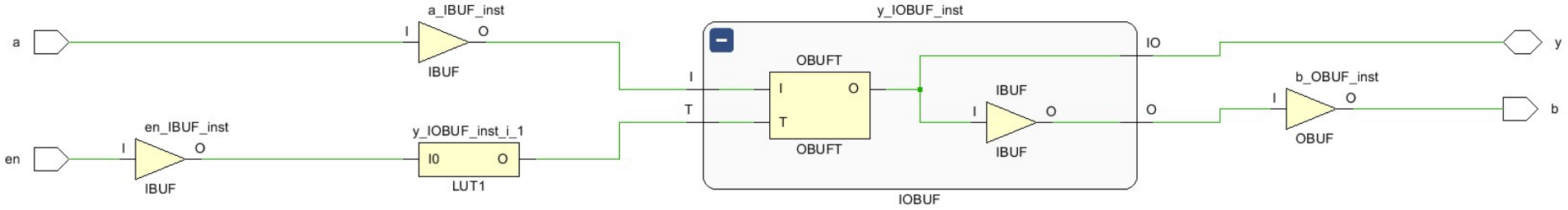
- 行为描述的三态门

```
module tristate1(in,en,out);
input in,en;
output reg out;
always @(in or en)
    begin if(en) out<=in;
    else out<=1'bz; end
endmodule
```



三态双向驱动器

```
module bidir(y,a,en,b);
input a,en; output b;
inout y;
    assign y=en?a:'bz;
    assign b=y;
endmodule
```

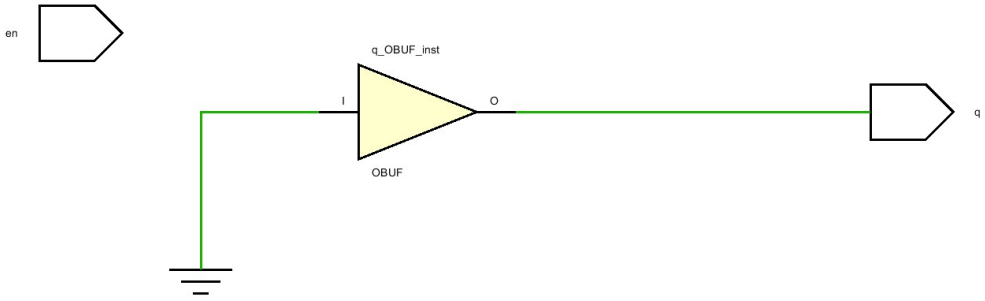


always语句

- Always_comb - 允许仿真工具检查正确的组合逻辑代码风格

```
always_comb
begin
    tmp1 = a & b;
    tmp2 = c & d;
end
```

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



10166 SystemVerilog RTL Coding error at non_block.sv(6): always_comb construct does not infer purely combinationa

[Synth 8-87] always_comb on 'q_reg' did not result in combinational logic [test.sv:6]

always语句

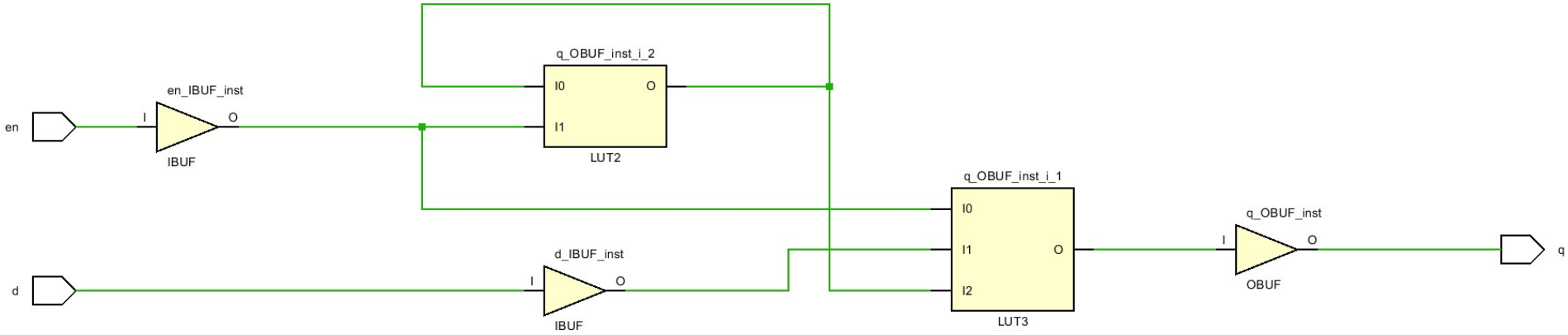
- always_latch - 允许仿真工具检查正确的锁存器逻辑代码风格

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

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

```
10165 SystemVerilog RTL Coding error at non_block.sv(6): always_latch construct does not infer clocked logic
```

反馈环不能生成锁存器



always语句

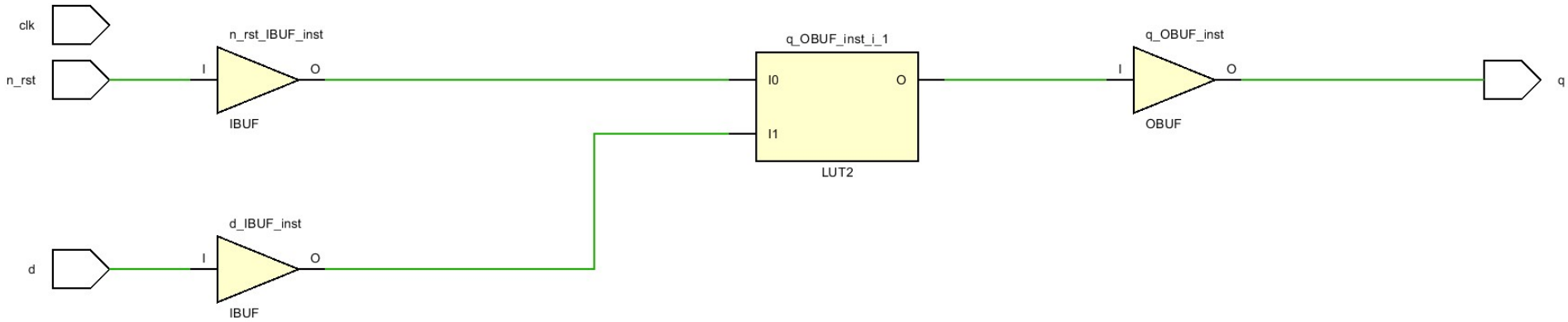
- always_ff - 允许仿真工具检查正确的寄存器逻辑代码风格

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

[Synth 8-567] referenced signal 'd' should be on the sensitivity list [test.sv:5]

[Synth 8-3331] design dff has unconnected port clk (1 more like this)

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



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

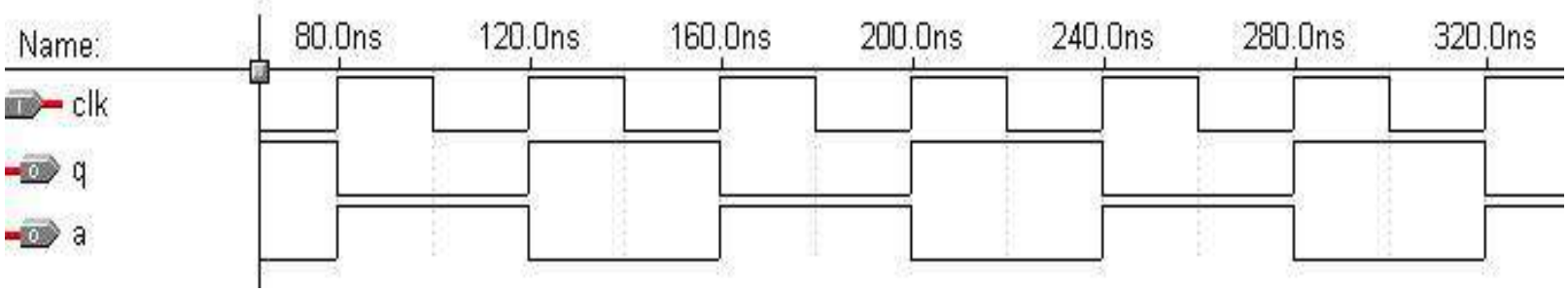
- 顺序执行模块1

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

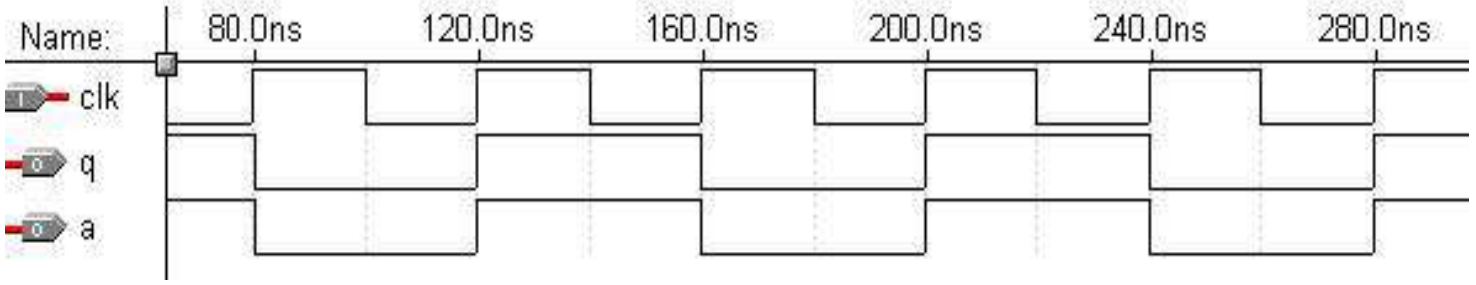
- 顺序执行模块2

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

顺序执行的时序效果



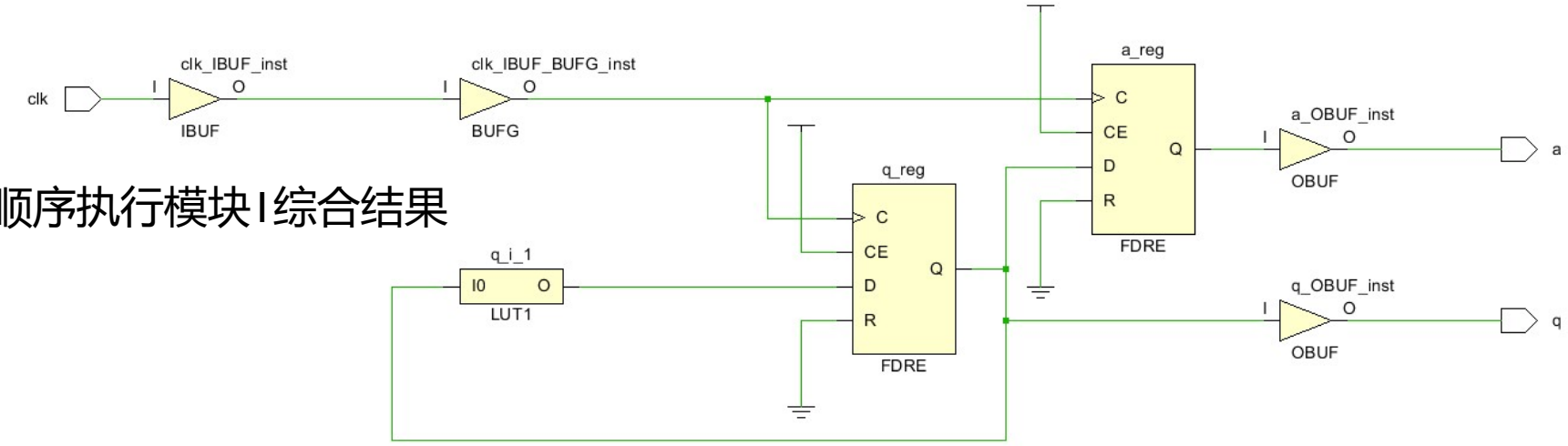
顺序执行模块1仿真波形图



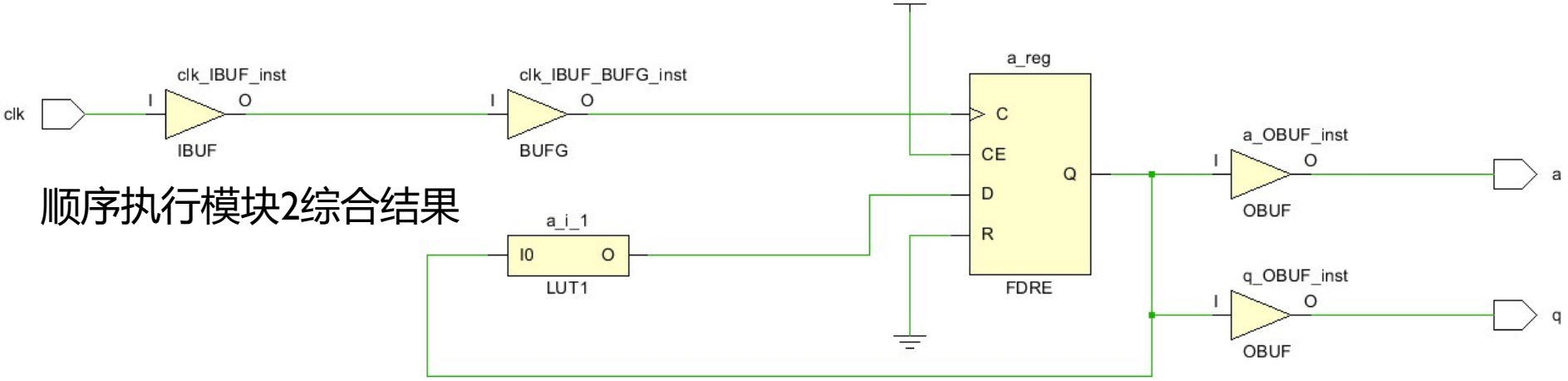
顺序执行模块2仿真波形图

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



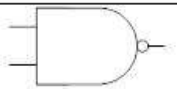
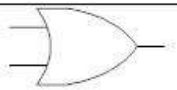
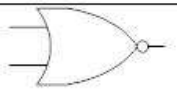
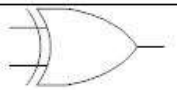
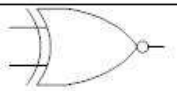
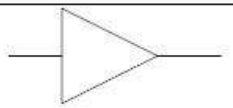
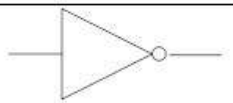
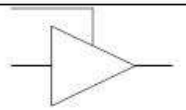
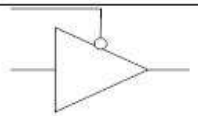
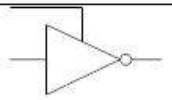
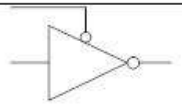
描述风格

- 结构（Structural）描述
- 行为（Behavioural）描述
- 数据流（Data Flow）描述

结构描述

- 在Verilog程序中可通过如下方式描述电路的结构
  - 调用Verilog内置门元件
  - 用户自定义元件

内置门元件

类　别	关　键　字	符号示意图	门　名　称
多输入门	and		与门
	nand		与非门
	or		或门
	nor		或非门
	xor		异或门
	xnor		异或非门
多输出门	buf		缓冲器
	not		非门
三态门	bufif1		高电平使能三态缓冲器
	buif0		低电平使能三态缓冲器
	notif1		高电平使能三态非门
	notif0		低电平使能三态非门

门元件的调用

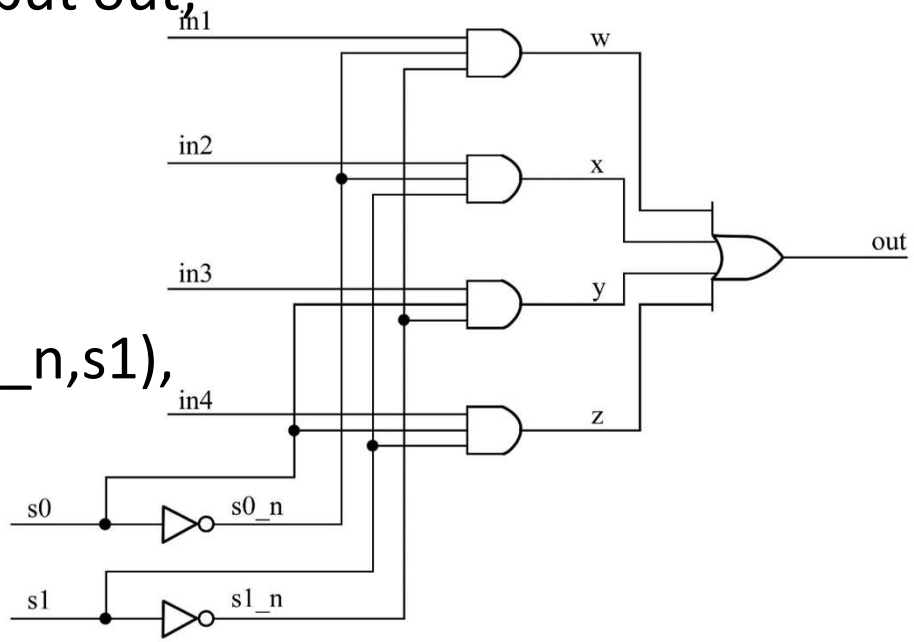
- 调用门元件的格式为：
  - 门元件名字<例化的门名字>(<端口列表>)
  - 其中普通门的端口列表按下面的顺序列出：
  - （输出，输入1，输入2，输出3，……）；
  - 比如
  - and a1(out,in1,in2,in3);
- 对于三态门，则按如下顺序列出输入输出端口：
  - （输出，输入，使能控制端）；
  - 比如：
  - bufif1 mytri1(out,in,enable); //高电平使能的三态门

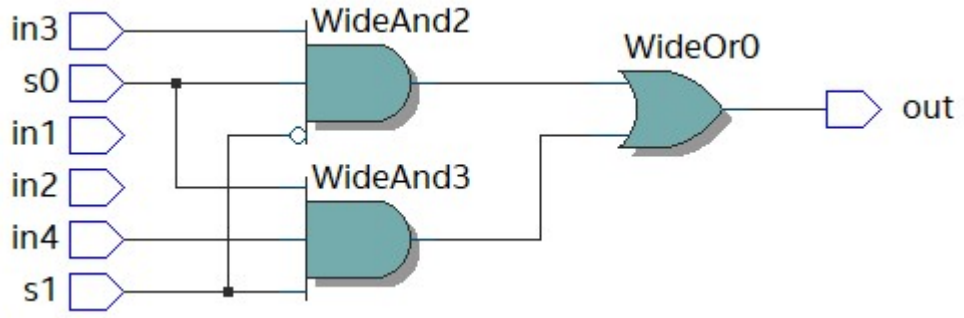
门元件的调用

- 对于buf和not两种元件的调用，需注意的是：它们允许有多个输出，但只能有一个输入。比如：
- not N1（out1,out2,in)
- buf B1(out1,out2,out3,in);

调用门元件实现的4选1 MUX

```
module mux4_1a(out,in1,in2,in3,in4,s0,s1);
        input in1,in2,in3,in4,s0,s1; output out;
        wire s0_n,s1_n,w,x,y,z;

        not (sel0_n,s0),(s1_n,s1);
        and (w,in1,s0_n,s1_n),(x,in2,s0_n,s1),
(y,in3,s0,s1_n),(z,in4,s0,s1);
        or (out,w,x,y,z);
endmodule
```



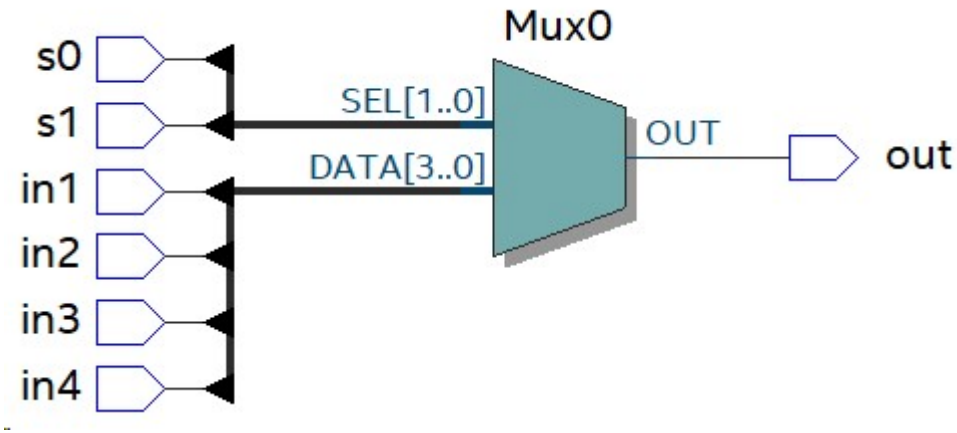


行为描述

- 针对设计实体的数学模型的描述，其抽象程度远高于结构描述方式。行为描述类似于高级编程语言，单描述一个设计实体的行为时，无需知道具体电路的结构，只需要描述清楚输入与输出信号的行为，而不需要花费更多的精力关注设计功能的门级实现

用case语句描述的4选1 MUX

```
module mux4_1b(out,in1,in2,in3,in4,s0,s1);
input in1,in2,in3,in4,s0,s1;
output reg out;
Always_comb
   case({s0,s1})
      2'b00:out=in1;
      2'b01:out=in2;
      2'b10:out=in3;
      2'b11:out=in4;
      default:out=2'bx;
   endcase
endmodule
```



采用行为描述方式时需注意

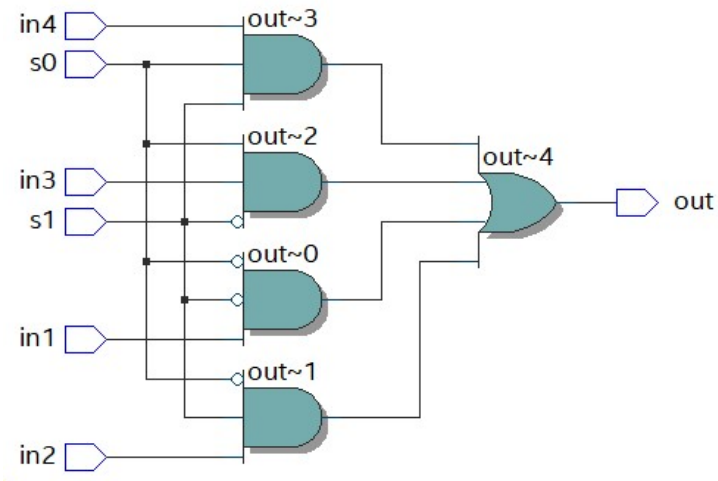
- 用行为描述模式设计电路，可以降低设计难度。行为描述只需表示输入和输出之间的关系，不需要包含任何结构方面的信息。设计者只需写出源程序，而挑选电路方案的工作由EDA软件自动完成。
- 在电路的规模较大或者需要描述复杂的逻辑关系时，应首先考虑用行为描述方式设计电路，如果设计的结果不能满足资源占有率的要求，则应该变描述方式。

数据流描述

- 数据流描述方式主要使用持续赋值语句，多用于描述组合逻辑电路

数据流描述的4选1 MUX

```
module mux4_1c(out,in1,in2,in3,in4,s0,s1);
    input in1,in2,in3,in4,s0,s1;
    output out;
    assign out=(in1 & ~s0 & ~s1)|(in2 & ~s0 & s1)|
  (in3& s0 & ~s1)|(in4 & s0 & s1);
endmodule
```



数据流描述

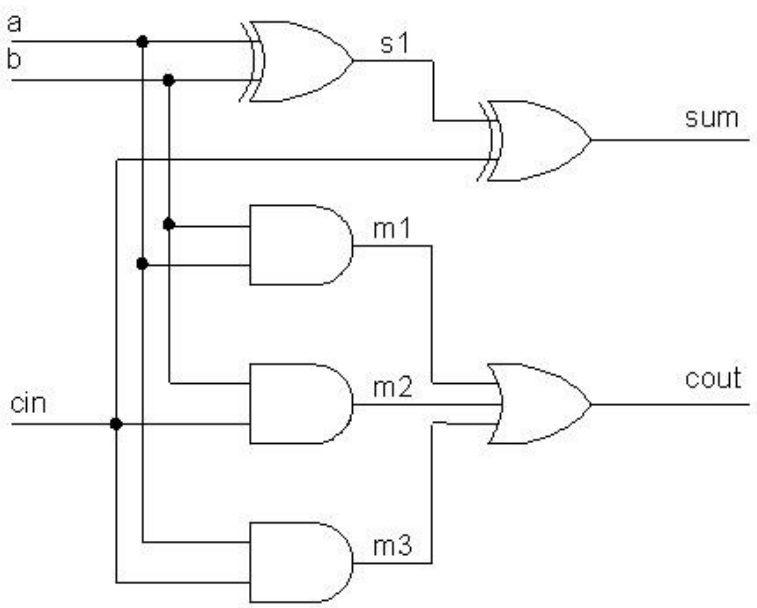
- 用数据流描述模式设计电路与用传统的逻辑方程设计电路很相似。设计中只要有了布尔代数表达式就很容易将它用数据流方式表达出来。

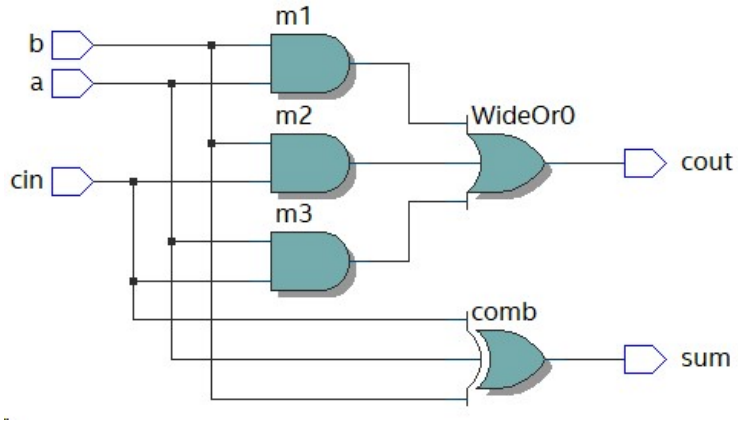
不同描述风格的设计

- 对于设计者而言，采用的描述级别越高，设计越容易；对于综合器而言，行为级的描述为综合器的优化提供了更大的空间，较之门级结构描述更能发挥综合器的性能，所以在电路设计中，除非一些关键路径的设计采用门级结构描述外，一般更多采用行为建模方式。

调用门元件实现的1位全加器

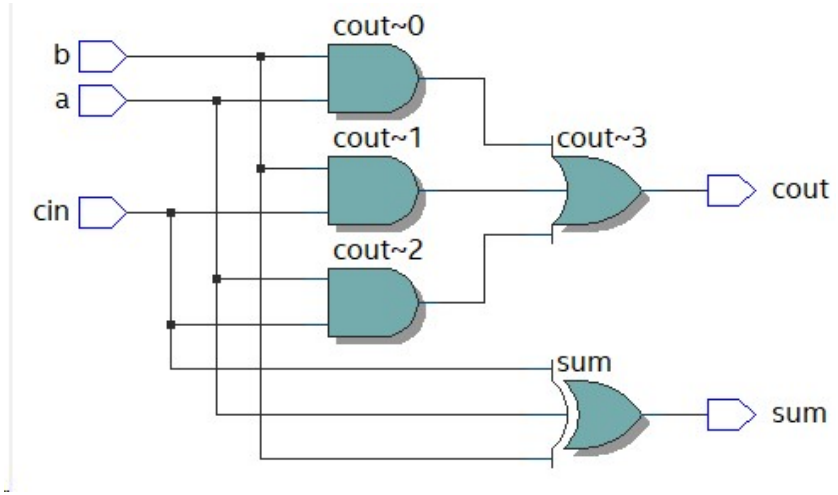
```
module full_add1(a, b, cin, sum, cout);
        input a, b, cin;
        output sum, cout;
        wire s1,m1, m2, m3;

        and (m1, a, b),
                    (m2, b, cin),
                    (m3, a, cin);
        xor (s1, a, b),
                    (sum, s1, cin);
        or (cout, m1, m2, m3);
endmodule
```





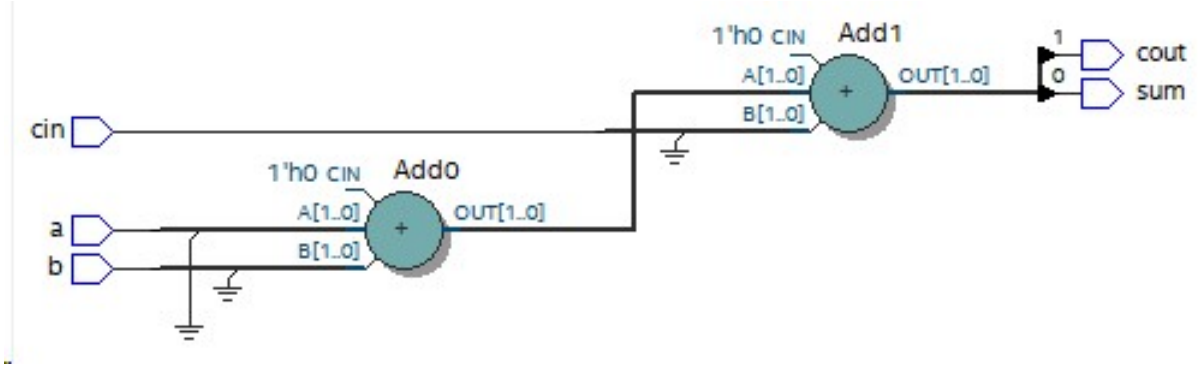
数据流描述的1位全加器

```
module full_add2(a,b,cin,sum,cout);
    input a, b, cin;
    output sum, cout;
    assign sum = a ^ b ^ cin;
    assign cout = (a & b ) | (b & cin ) | (cin & a );
endmodule
```



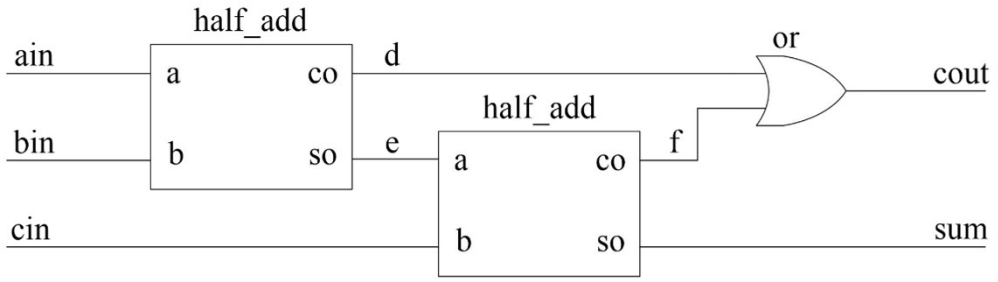
行为描述的1位全加器

```
module full_add3(a,b,cin,sum,cout);
        input a,b,cin;
        output reg sum,cout;
        always_comb
        begin
                {cout,sum}=a+b+cin;
        end
endmodule
```

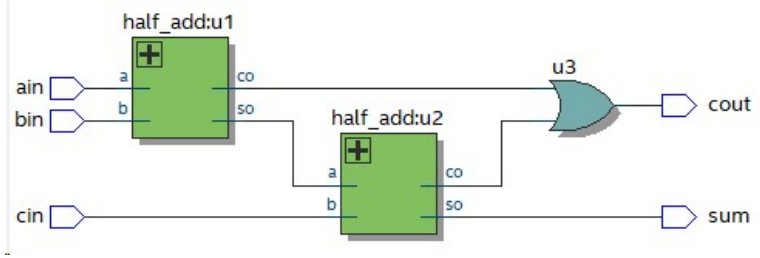


采用层次化方式设计的1位全加器

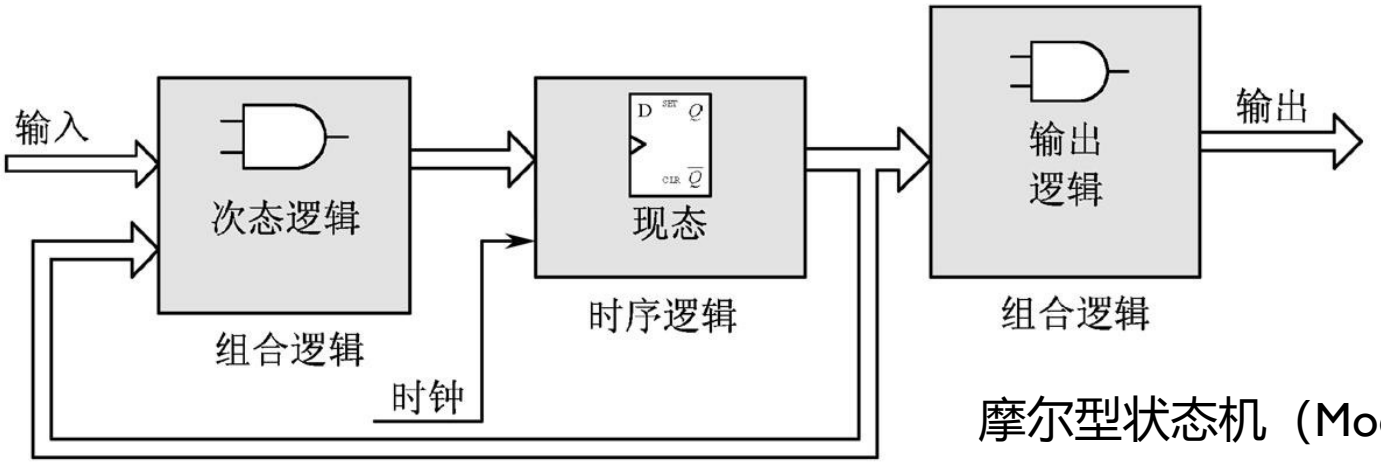
```
module half_add(a,b,so,co);
input a,b;
output so,co;
    assign co=a&b;
    assign so=a^b;
endmodule
```



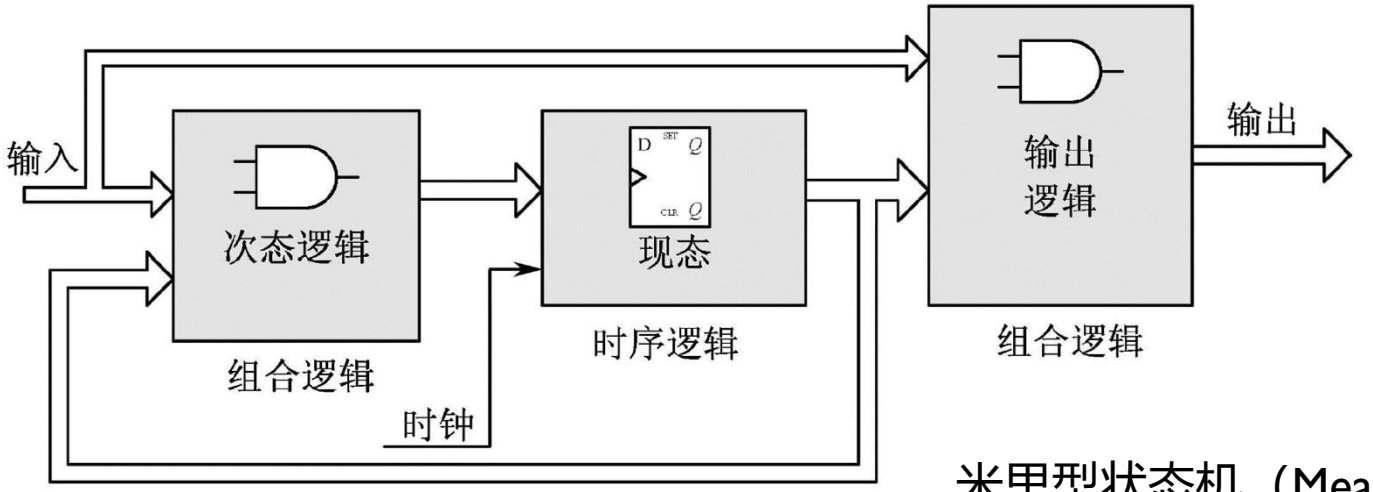

```
module full_add(ain,bin,cin,sum,cout);
        input ain,bin,cin;
        output sum,cout;
        wire d,e,f; //用于内部连接的节点信号

        half_add u1(ain,bin,e,d);
        //半加器模块调用，采用位置关联方式
        half_add u2(e,cin,sum,f);
        or u3(cout,d,f); //或门调用
endmodule
```



有限状态机



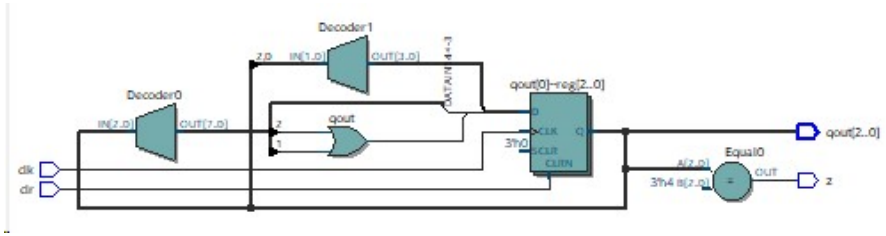
摩尔型状态机（Moore）

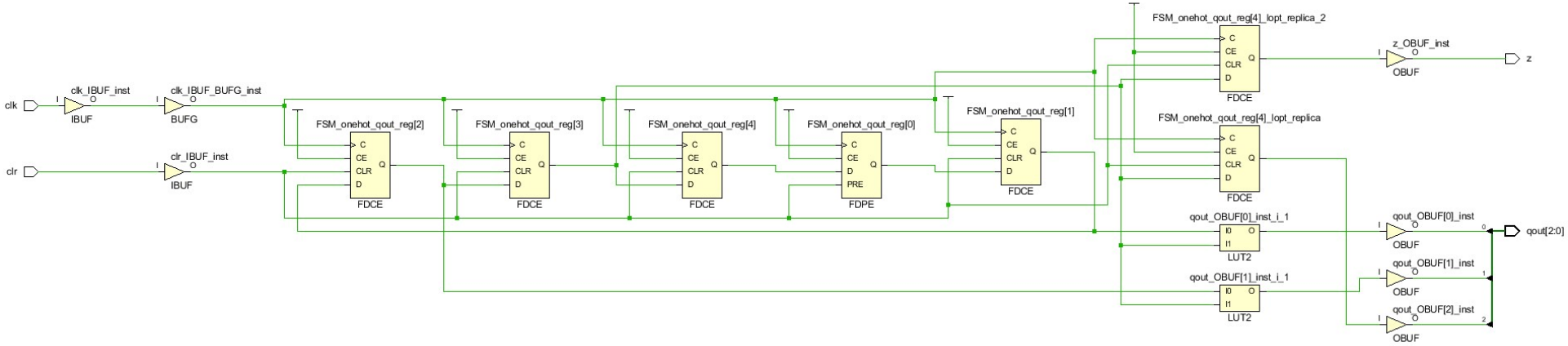


米里型状态机（Mealy）

用状态机设计模5计数器

```
module fsm(clk,clr,z,qout);
        input clk,clr; output reg z; output reg[2:0] qout;

        always_ff @(posedge clk, posedge clr) //此过程定义状态转换
        begin
                if(clr) qout<=0; //异步复位
                else case(qout)
                        3'b000: qout<=3'b001;
                        3'b001: qout<=3'b010;
                        3'b010: qout<=3'b011;
                        3'b011: qout<=3'b100;
                        3'b100: qout<=3'b000;
                        default: qout<=3'b000; /*default语句*/
                endcase
        end

        always_comb/*此过程产生输出逻辑*/
        begin
                case(qout)
                        3'b100: z=1'b1;
                        default:z=1'b0;
                endcase
        end
endmodule
```



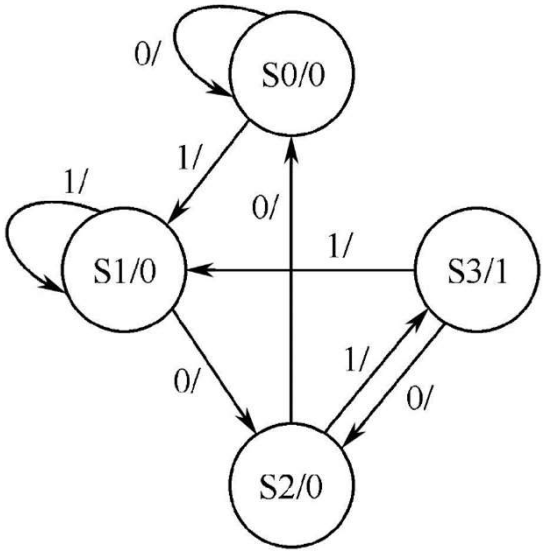

clk
clk_IBUF_inst
IBUF
clk_IBUF_BUFG_inst
BUFG
clr
clr_IBUF_inst
IBUF
FSM_onehot_qout_reg[2]
FSM_onehot_qout_reg[3]
FSM_onehot_qout_reg[4]
FSM_onehot_qout_reg[0]
FSM_onehot_qout_reg[1]
C
CE
CLR
D
Q
PRE
FDCE
FDPE
FSM_onehot_qout_reg[4]_lopt_replica_2
FSM_onehot_qout_reg[4]_lopt_replica
z_OBUF_inst
OBUF
z
qout_OBUF[0]_inst_i_1
qout_OBUF[1]_inst_i_1
I0
I1
LUT2
qout_OBUF[0]_inst
qout_OBUF[1]_inst
qout_OBUF[2]_inst
qout[2:0]

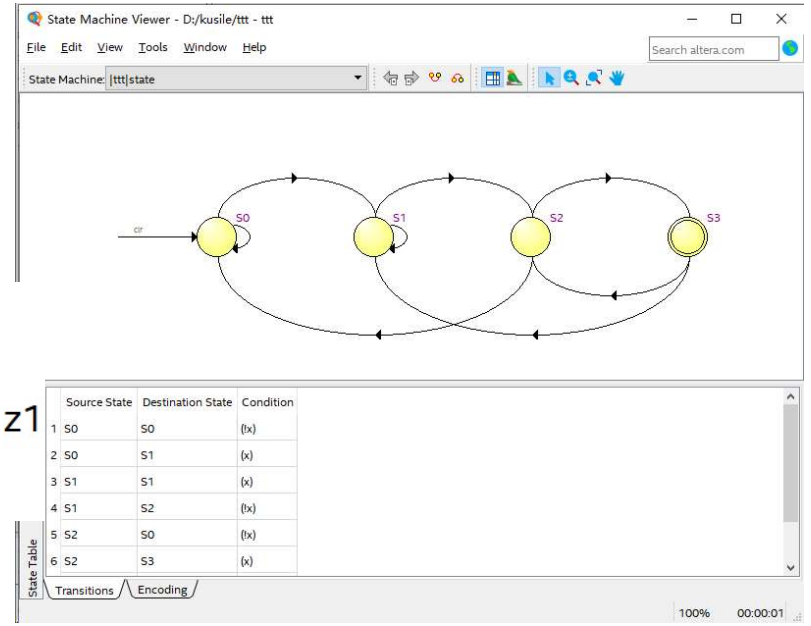
有限状态机的几种描述方式

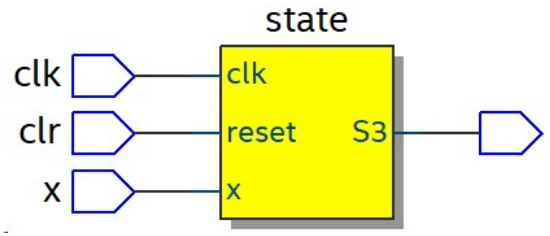
- 用三个过程描述：即现态（CS），次态（NS），输出逻辑（OL）各用一个always过程描述。
- 双过程描述（CS+NS，OL双过程描述）：使用两个always过程来描述有限状态机，一个过程描述现态和次态时序逻辑（CS+NS）；另外一个过程描述输出逻辑（OL）。
- 双过程描述（CS，NS+OL双过程描述）：一个过程用来描述现态（CS）；另一个过程描述次态和输出逻辑（NS+OL）。
- 单过程描述：在单过程描述方式中，将状态机的现态、次态和输出逻辑（CS+NS+OL）放在一个always过程中进行描述。

101序列检测器的Verilog描述（三个过程）

```
module  ttt(clk,clr,x,z1);
    input  clk,clr,x;
    output reg z1;
    reg[1:0]  state,next_state;
    parameter  S0=2'b00,S1=2'b01,S2=2'b11,S3=2'b10;

    always_ff  @(posedge  clk,  posedge  clr)
    begin
        if(clr)
            state<=S0;
        else
            state<=next_state;
    end

    always_comb
    begin
        case(state)
            S0:
                begin
                    if(x)  next_state<=S1;
                    else  next_state<=S0;
                end
            S1:
                begin
                    if(x)  next_state<=S1;
                    else  next_state<=S2;
                end
            S2:
                begin
                    if(x)  next_state<=S3;
                    else  next_state<=S0;
                end
            S3:
                begin
                    if(x)  next_state<=S1;
                    else  next_state<=S2;
                end
            default:
                next_state<=S0:
        endcase
    end

    always_comb
    begin
        case(state)
            S3:z1=1'b1;
            default:z1=1'b0;
        endcase
    end
endmodule
```

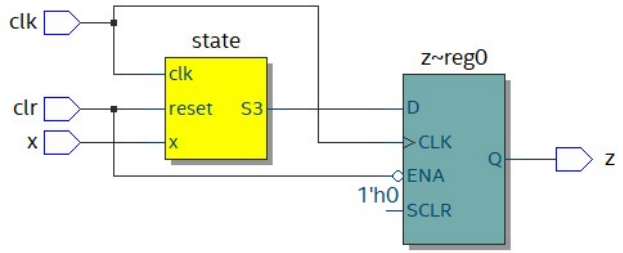


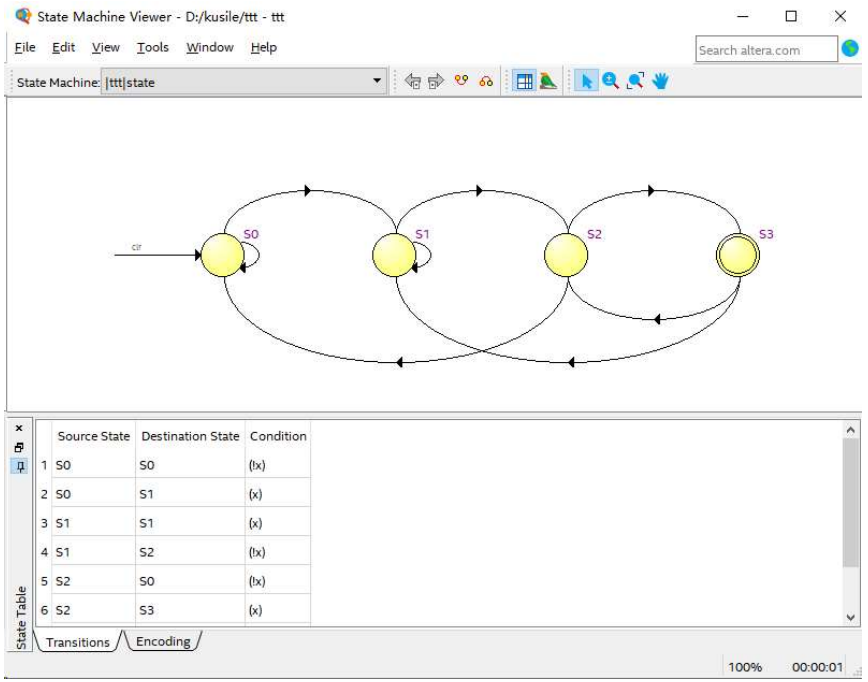




单过程描述

```
module fsm4_seq101(clk,clr,x,z);
    input clk,clr,x;
    output reg z;
    reg[1:0] state;
    parameter S0=2'b00,S1=2'b01,S2=2'b11,S3=2'b10;

    always_ff @(posedge clk, posedge clr)
    begin
        if(clr) state<=S0;
        else
            case(state)
                S0:
                begin
                    if(x) begin state<=S1;z=1'b0;end
                    else begin state<=S0;z=1'b0;end
                end
                S1:
                begin
                    if(x) begin state<=S1;z=1'b0;end
                    else begin state<=S2;z=1'b0;end
                end
                S2:
                begin
                    if(x) begin state<=S3;z=1'b0;end
                    else begin state<=S0;z=1'b0;end
                end
                S3:
                begin
                    if(x) begin state<=S1;z=1'b1;end
                    else begin state=S2;z=1'b1;end
                end
                default:begin state<=S0;z=1'b0;end
            endcase
    end
endmodule
```





状态编码

- 常用的编码方式

- 顺序编码
- 格雷编码
- 约翰逊编码
- 一位热码

一般使用case，casez和casex语句来描述状态之间的转换，用case语句表述比用if-else语句更加清晰明了

State Variables

State	One-Hot Code	Binary Code	Gray Code
S0	00001	000	000
S1	00010	001	001
S2	00100	010	011
S3	01000	011	010
S4	10000	100	110

Table 1:An example of state Encoding for a 4 state Machine

有限状态机设计要点

- 起始状态的选择：起始状态是指电路复位后所处的状态，选择一个合理的起始状态将使整个系统简洁，高效。多数EDA软件会自动为基于状态机的设计选择一个最佳的起始状态。
- 有限状态机的同步复位
- 有限状态机的异步复位

多余状态的处理

- 一般有如下两种处理多余状态的方法：
  - 在case语句中使用default分支决定如果进入无效状态所采取的措施；
  - 编写必要的Verilog源代码明确定义进入无效状态所采取的行为。