

# System Verilog入门（二）

# 实验目的

- 组合逻辑电路的设计
- 描述风格
- 仿真

# module结构

```
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
  - wire, tri等;
- variable型
  - 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

# 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
  - 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

# 运算符（1）

## ➤ 算术运算符（Arithmetic operators）

- 常用的算术运算符包括：

- + 加

- - 减

- \* 乘

- /

## ➤ 逻辑运算符（Logical operators）

- && 逻辑与

- || 逻辑或

- ! 逻辑非

# 运算符（2）

## ➤ 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。
- $\sim$  按位取反
- $\&$  按位与
- $|$  按位或
- $\wedge$  按位异或
- $\wedge\sim, \sim\wedge$  按位同或（符号  $\wedge\sim$  与  $\sim\wedge$  是等价的）

## ➤ 关系运算符（Relational operators）

- $<$  小于
- $\leq$  小于或等于
- $>$  大于
- $\geq$  大于或等于



# 运算符（3）

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

# 运算符（4）

- 条件运算符（conditional operators）？
  - 三目运算符： `signal = condition ? true_expression : false_expression;`
  - `A = B ? 1:0`
- 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}
  - {A[3, A[1]}

# 运算符的优先级

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

# 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。

# 块语句

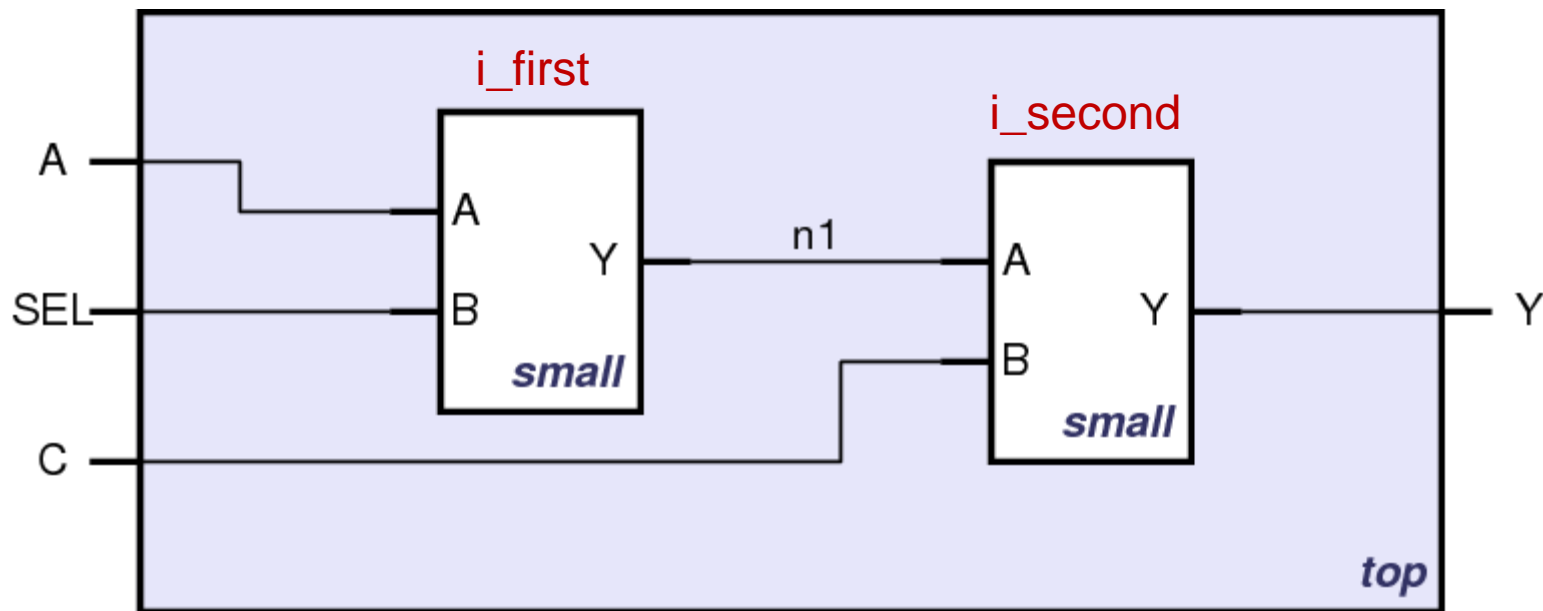
- 块语句是由块标识符**begin-end**或**fork-join**界定的一组语句，当块语句只包含一条语句时，块标识符可以缺省
- **begin-end**串行块中的语句按串行方式顺序执行，比如

```
begin  
    regb = rega;  
    regc = regb;  
end
```

# 描述风格

- 结构描述（门级描述）
  - 更像进行硬件的设计
  - 模块主体包含电路的门级描述
  - 描述模块是如何相互连接的
  - 每个模块包含其它模块（实例） 以及这些模块之间的互连关系
- 行为描述
  - 描述模块的行为，而不设计具体的模块
  - 包含逻辑和数学运算符
  - 抽象水平高于门级描述
    - 相同的行为描述会有多种可能的门级实现方式
- 数据流描述
- 实际的系统会同时使用上述的实现方式

# 结构描述



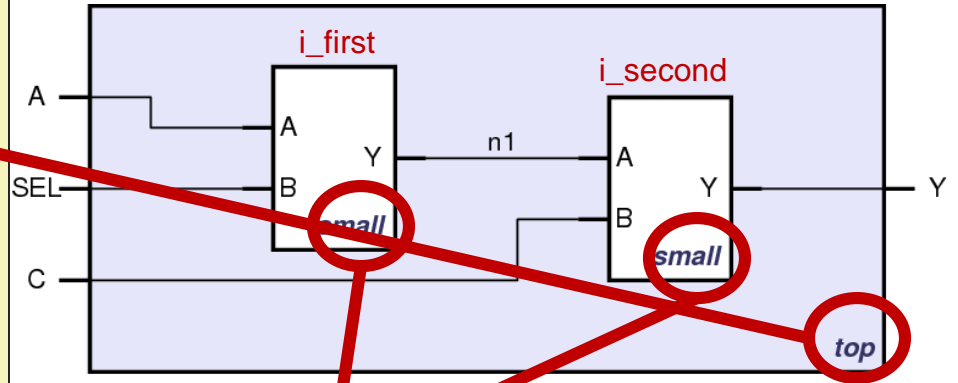
**top**的模块由两个**small**模块组成，上图是它们直接的连线关系

# 结构描述

- 模块定义

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

```
// description of small
```

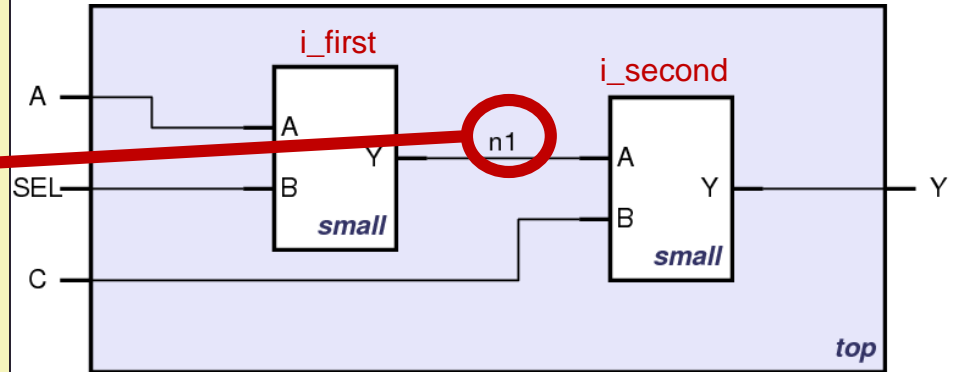
```
endmodule
```



# 结构描述

- 定义连线关系 (模块互联)

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;  
  
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;  
  
  // description of small  
  
endmodule
```

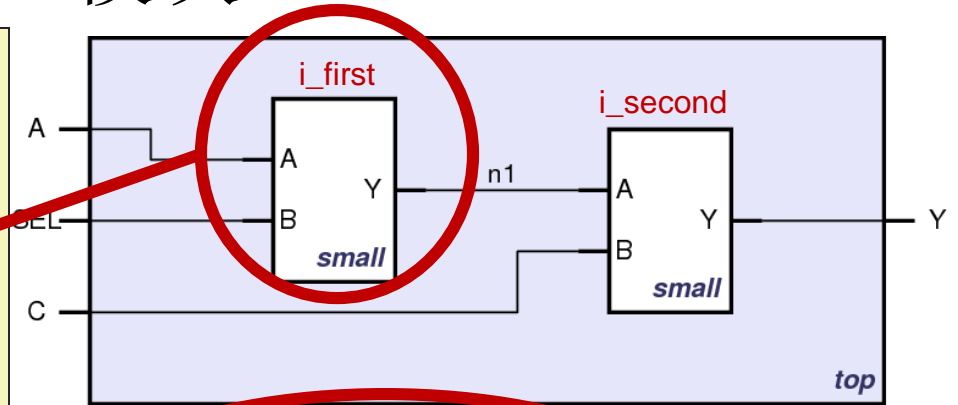
# 结构描述

- 第一个例化的 “small” 模块

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

```
// instantiate small once  
small i_first ( .A(A),  
                .B(SEL),  
                .Y(n1) );
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

```
// description of small
```

```
endmodule
```

# 结构描述

- 第二个例化的 “small” 模块

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

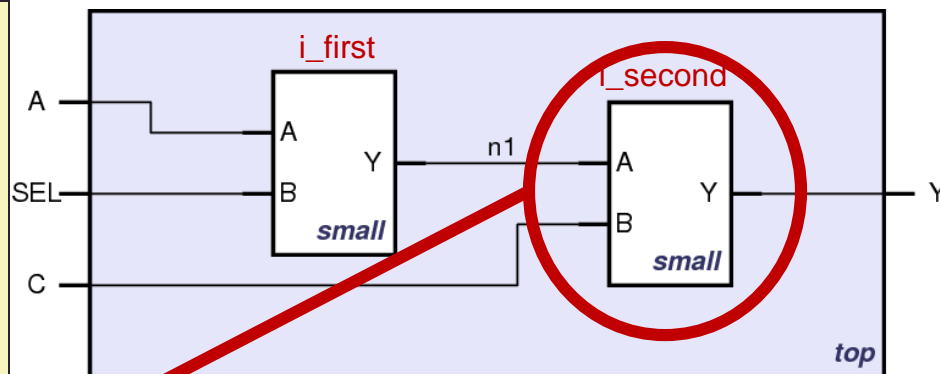
```
// instantiate small once
```

```
small i_first ( .A(A),  
                .B(SEL),  
                .Y(n1) );
```

```
// instantiate small second time
```

```
small i_second ( .A(n1),  
                 .B(C),  
                 .Y(Y) );
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

```
// description of small
```

```
endmodule
```

# 结构描述

- 模块实例化的简短形式

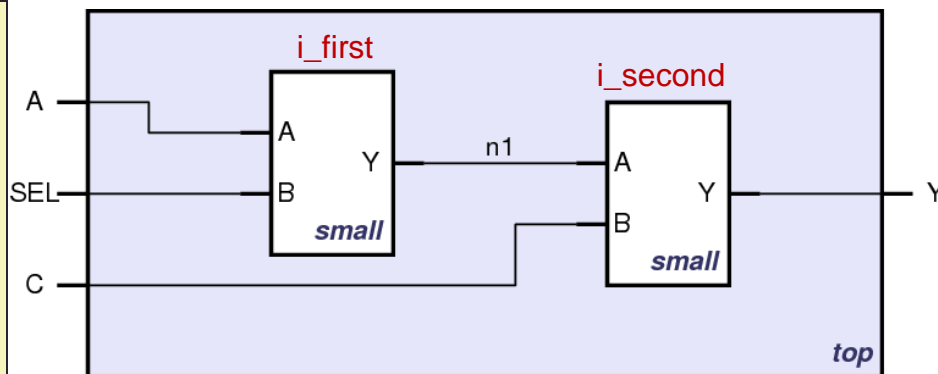
```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

```
// alternative short form  
small i_first ( A, SEL, n1 );
```

```
/* In short form above,  
   pin order very important */
```

```
// safer choice; any pin order  
small i_second ( .B(C),  
                 .Y(Y),  
                 .A(n1) );
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

```
// description of small
```

```
endmodule
```

简短的形式不是好的做法  
降低了代码的可维护性

# 结构描述

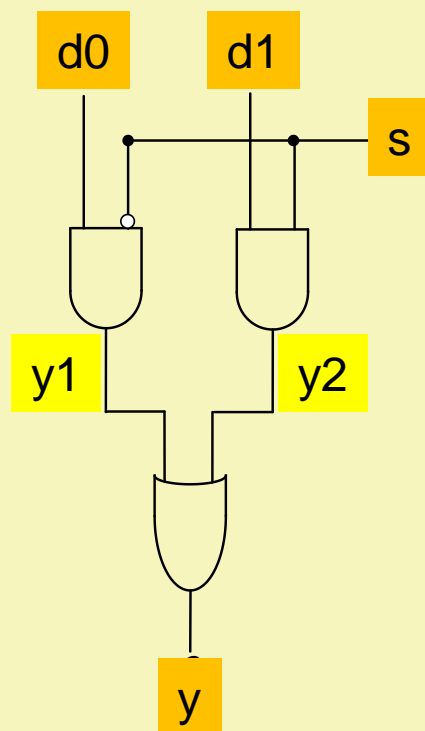
- 支持基本的逻辑门作为预定义的基本模块
  - 这些基本模块像其它模块一样被实例化，已经预定义，不需要模块定义

```
module mux2(input wire d0, d1,  
            input wire s,  
            output wire y);
```

```
    wire ns, y1, y2;
```

```
    not  g1 (ns, s);  
    and  g2 (y1, d0, ns);  
    and  g3 (y2, d1, s);  
    or   g4 (y, y1, y2);
```

```
endmodule
```



## RTL ANALYSIS

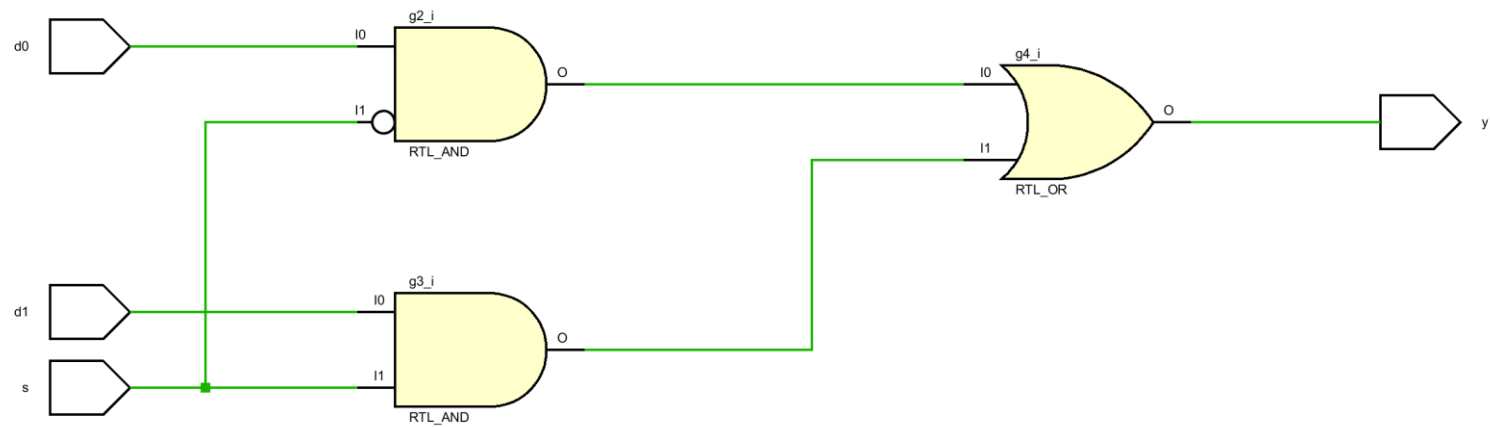
### Open Elaborated Design

Report Methodology

Report DRC

Report Noise

Schematic



## SYNTHESIS

Run Synthesis

### Open Synthesized Design

Constraints Wizard

Edit Timing Constraints

Set Up Debug

Report Timing Summary

Report Clock Networks

Report Clock Interaction

Report Methodology

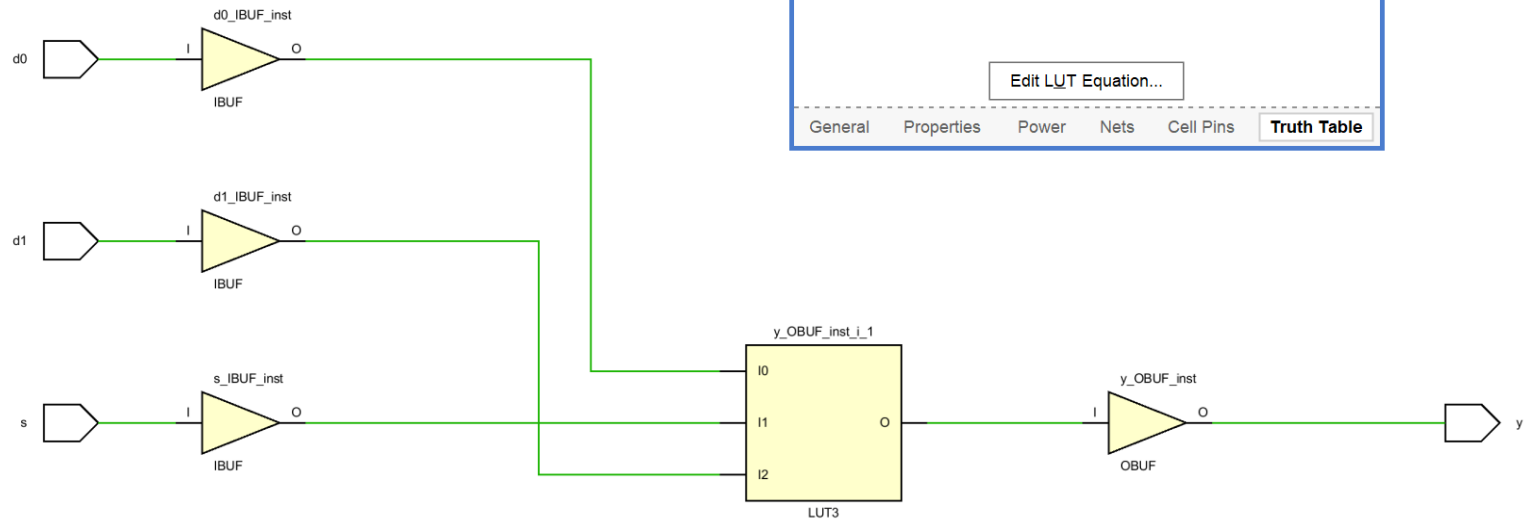
Report DRC

Report Noise

Report Utilization

Report Power

Schematic














| Cell Properties |    |    |                           |  |
|-----------------|----|----|---------------------------|--|
| y_OBUF_inst_i_1 |    |    |                           |  |
| I2              | I1 | I0 | O= $I0 \& I11 + I1 \& I2$ |  |
| 0               | 0  | 0  | 0                         |  |
| 0               | 0  | 1  | 1                         |  |
| 0               | 1  | 0  | 0                         |  |
| 0               | 1  | 1  | 0                         |  |
| 1               | 0  | 0  | 0                         |  |
| 1               | 0  | 1  | 1                         |  |
| 1               | 1  | 0  | 1                         |  |
| 1               | 1  | 1  | 1                         |  |

Edit LUT Equation...

General Properties Power Nets Cell Pins Truth Table

# 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 |
|      | buiif0 |    | 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

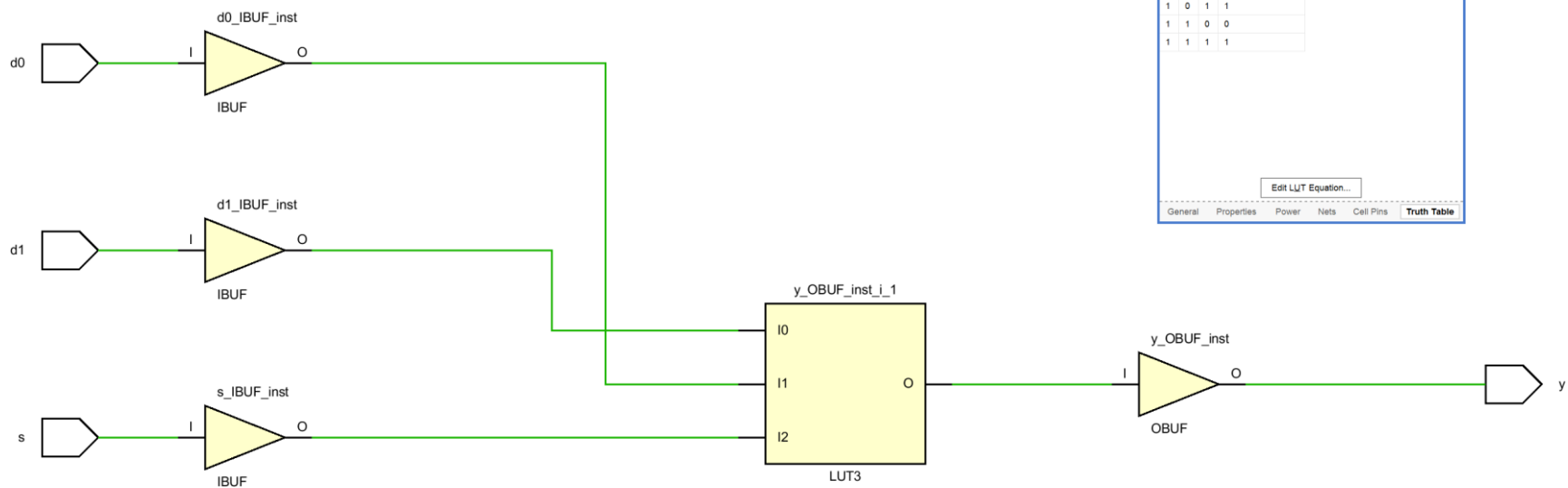
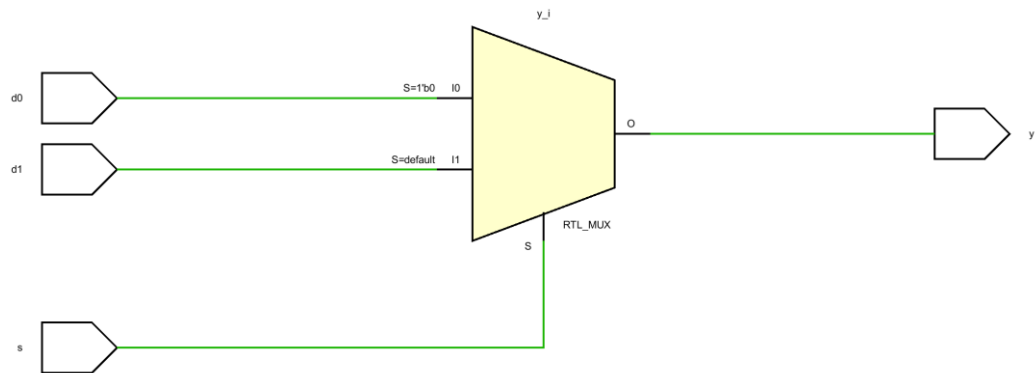
- 对于buf和not两种元件的调用，需注意的是：它们允许有多个输出，但只能有一个输入。比如：
- not N1 (out1,out2,in)
- buf B1(out1,out2,out3,in);

# 行为描述

- 针对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

# 行为描述

```
module mux2(input wire d0, d1,  
            input wire s,  
            output reg y);  
  
    // here comes the circuit description  
    always_comb begin  
        if (!s )  
            y = d0;  
        else  
            y = d1;  
        end  
    endmodule
```



Cell Properties

y\_OBUF\_inst\_1

| I2 | I1 | I0 | O=I1 & I2 + I0 & I2 |
|----|----|----|---------------------|
| 0  | 0  | 0  | 0                   |
| 0  | 0  | 1  | 0                   |
| 0  | 1  | 0  | 1                   |
| 0  | 1  | 1  | 1                   |
| 1  | 0  | 0  | 0                   |
| 1  | 0  | 1  | 1                   |
| 1  | 1  | 0  | 0                   |
| 1  | 1  | 1  | 1                   |

Edit LUT Equation...

General Properties Power Nets Cell Pins Truth Table

**行为描述同样定义了硬件电路的模型**

# 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。

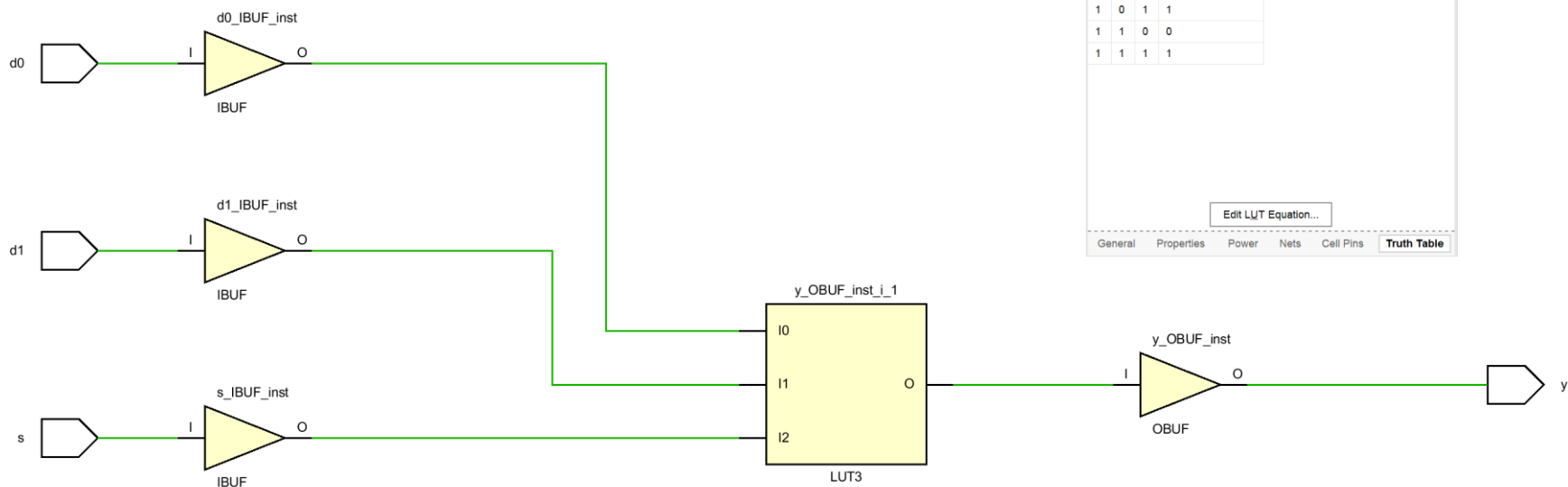
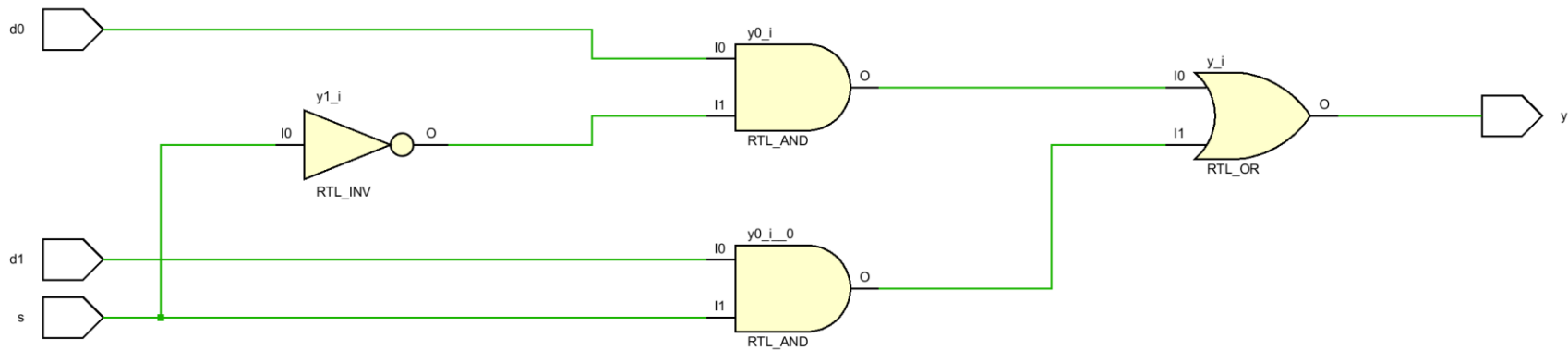
# 数据流描述

- 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
- 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

# 数据流描述

```
module mux2(input wire d0, d1,  
            input wire s,  
            output wire y);  
  
    assign y = d0 & !s | d1 & s;  
endmodule
```





**Cell Properties**

$y\_OBUF\_inst\_i\_1$

| I2 | I1 | I0 | O= $I1 \& I12 + I0 \& I2$ |
|----|----|----|---------------------------|
| 0  | 0  | 0  | 0                         |
| 0  | 0  | 1  | 0                         |
| 0  | 1  | 0  | 1                         |
| 0  | 1  | 1  | 1                         |
| 1  | 0  | 0  | 0                         |
| 1  | 0  | 1  | 1                         |
| 1  | 1  | 0  | 0                         |
| 1  | 1  | 1  | 1                         |

Edit LUT Equation...

General Properties Power Nets Cell Pins **Truth Table**

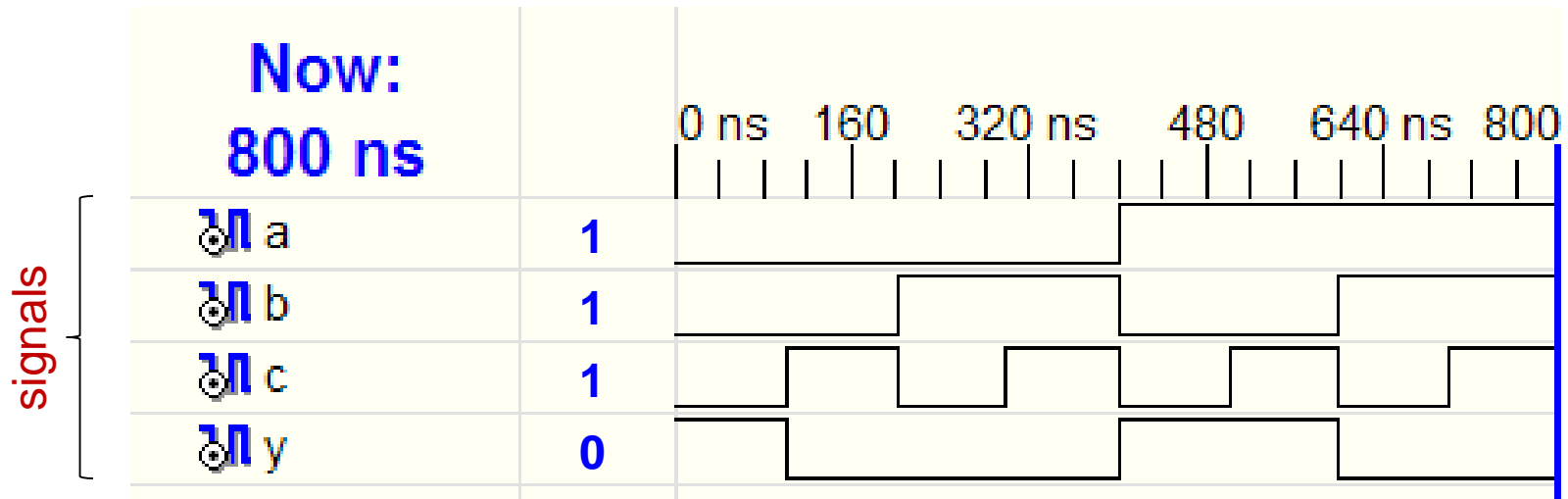
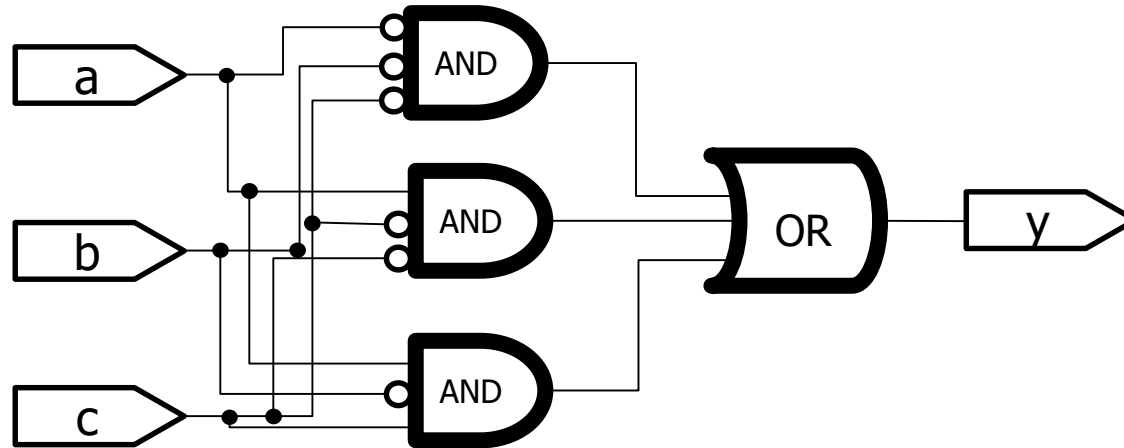
# 不同描述风格的设计

- 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。

# 波形

```
module example (a, b, c, y);  
    input a;  
    input b;  
    input c;  
    output y;  
  
    // here comes the circuit description  
    assign y = ~a & ~b & ~c |  
               a & ~b & ~c |  
               a & ~b & c;  
  
endmodule
```

# 波形



Waveform Diagram

time

# 延时

- 延时是不可综合的，但是在仿真中非常有用
- 用来仿真硬件的模型

```
'timescale 1ns/1ps
module simple (input a, output z1, z2);

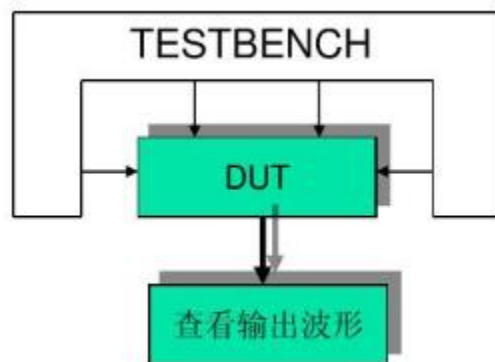
assign #5 z1 = ~a; // inverted output after 5ns
assign #9 z2 = a;  // output after 9ns

endmodule
```

1ns代码中时间的单位，1ps仿真的粒度

# 仿真

- RTL级行为仿真（又称作为功能仿真、前仿真）
- 综合后门级仿真
- 时序仿真（又称为后仿真）

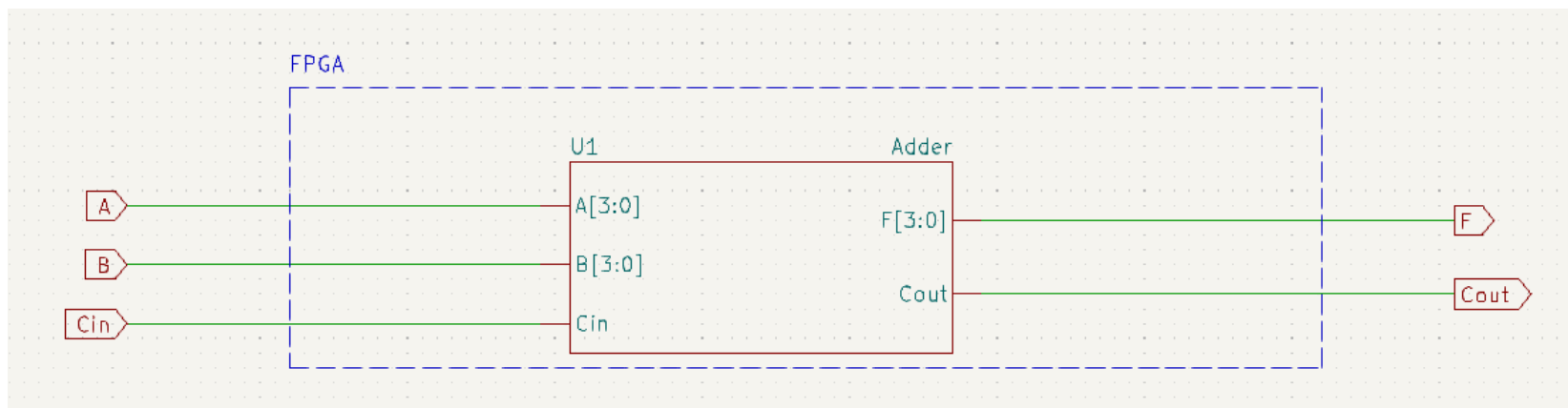


- **Test Bench**

- 用于产生电路的测试环境
- 不需要综合
- 可用Verilog HDL的任何语句可嵌入一些内部命令如\$display,\$monitor和\$readmemh等用于调试
- 主要是指功能测试

# 实验内容

- 用硬件描述语言实现1位全加器的设计，并使用自己设计的1位全加器构建4位全加器，
- 进行功能仿真
- 下载到FPGA中进行验证。





# 提高要求

- 设计实现超前进位的四位加法器；
- 使用SystemVerilog自带的加法运算实现四位全加器；
- 将结果采用十进制表示，并使用自带译码的七段数码管显示结果。