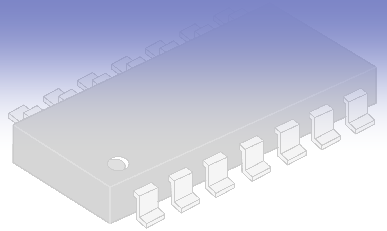
第 4 章

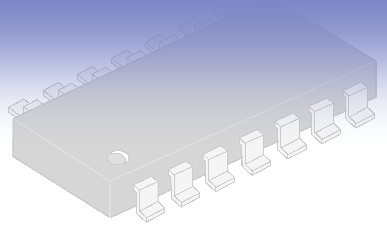
门 级 建 模

内容



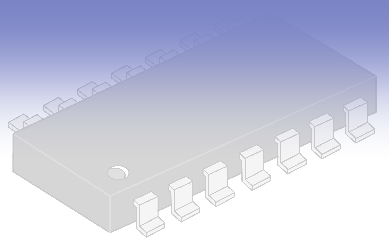
- 门的类型
- 门延时

Verilog 门级建模



- 使用门级原语的结构（Structural）描述
  - 使用基本单元
    - 如 and/or/not/xor 门
  - 描述电路中的各个基本逻辑单元，如 and/or/not/xor 门、各种类型的触发器等之间的相互连接关系
  - 与电路原理图（schematic）匹配
    - 相当于门级电路原理图的 HDL 描述

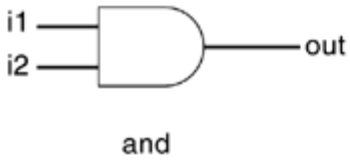
门的类型



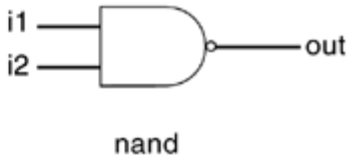
- 原语（primitive）
  - 预先定义的常用功能模型
- 完整的预定义门原语
  - 可以使用逻辑门设计数字电路
  - 门级原语是 Verilog 内建的、预定义的基本单元（模块）
  - 无需声明就可直接使用
- 基本逻辑门分为两类
  - 与/或门 —— **`and / or`**
  - 缓冲器/非门 —— **`buf / not`**
  - 可以使用这些门设计任何组合逻辑电路

与门（and）和或门（or）

- 用法
  - 门标识符(**输出端口**，输入端口1，输入端口2，输入端口3，...);
  - **1个输出端**
  - 多个（两个或两个以上）输入端
  - 端口列表中的**第一个**端口是输出端口
- 功能
  - 当任意一个输入端口的值发生变化时，立即重新计算输出端口的值
- 原语
  - **and、nand、or、nor、xor、xnor**
- 基本门的逻辑符号



and



nand

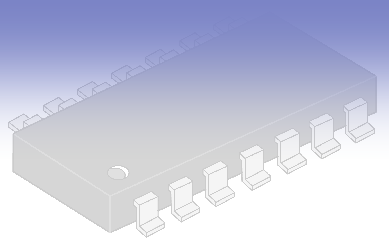
or

nor

xor

xnor

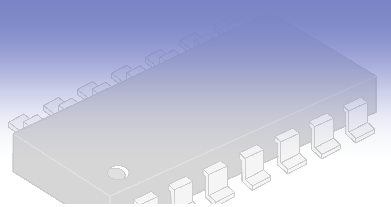
例、与门/或门实例引用



- 使用门原语实例构建逻辑电路
  - 输入端口的个数可以超过 2 个
  - 可以不指定具体实例的名称（不给定标识符）

```
wire OUT, IN1, IN2;

// 基本门实例
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);

// 3 输入端与非门
nand na1_3inp(OUT, IN1, IN2, IN3);

// 引用门原语实例时，可以没有实例名
and (OUT, IN1, IN2);
```

基本门的真值表（Truth table）



- 根据真值表计算基本门的输出
- 输入端口超过 2 个时
  - 计算 2 个输入的运算结果，得到的结果**再与第 3 个端口的输入值进行运算**

and	i1			
i2	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

nand	i1			
i2	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

or	i1			
i2	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

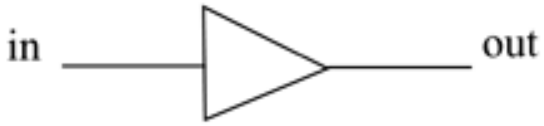
nor	i1			
i2	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xor	i1			
i2	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

xnor	i1			
i2	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

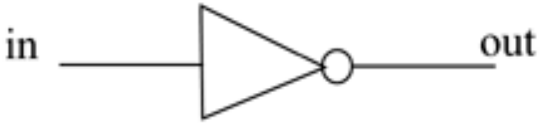
缓冲器/非门

- 原语：**buf（缓冲器），not （非门）**
- 用法
  - 多个输出，所有输出端的值都相同
  - 一个输入，端口列表中**最后一个是输入端**
- 逻辑符号
  - 单输入端和单输出端的 buf 和 not



buf

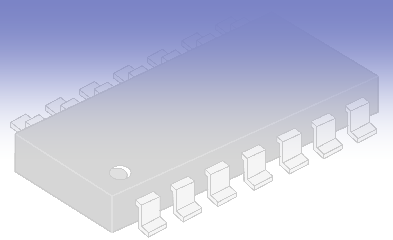
buf	in	out
	0	0
	1	1
	x	x
	z	x



not

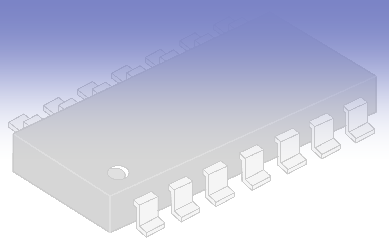
not	in	out
	0	1
	1	0
	x	x
	z	x

例、缓冲器/非门的实例

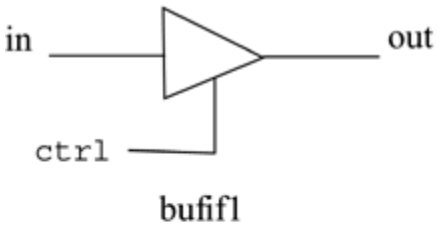


```
// 缓冲器和非门的实例引用
buf b1(OUT1, IN);
not n1(OUT1, IN);

// 多输出缓冲器实例引用
buf b1_2out(OUT1, OUT2, IN);

// 省略实例名
not (OUT1, IN);
```

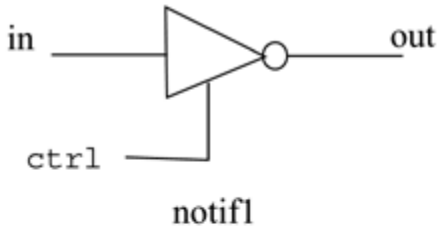
三态门



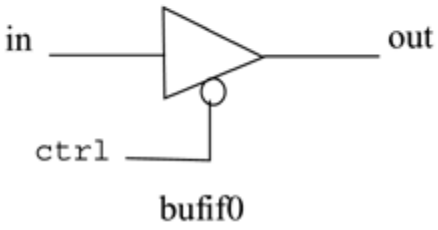
- 带控制端的缓冲器/非门
- 原语：
  - **bufif1，bufif0，notif1， notif0**
- 功能
  - 只有在控制信号有效时才能传递数据（信号）
  - **如果控制信号无效，则输出为高阻抗 Z**
- 逻辑符号
  - 单输入端和单输出端的 三态门



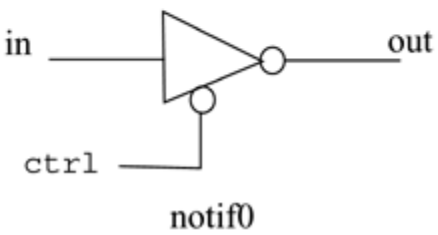
bufif1



notif1

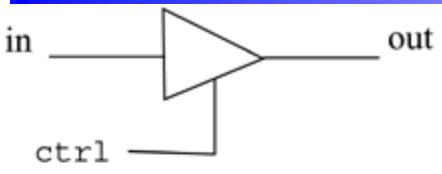


bufif0

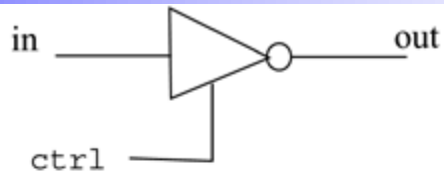


notif0

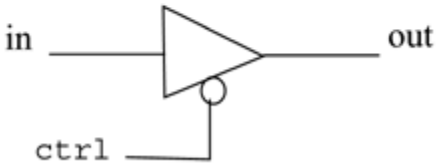
三态门的真值表



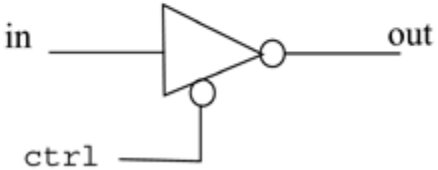
bufif1



notif1



bufif0



notif0

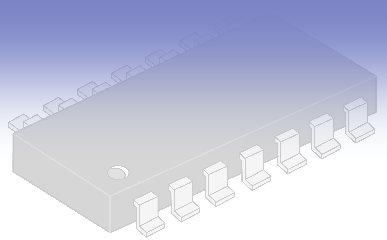
	bufif1	ctrl 0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

	bufif0	ctrl 0	1	x	z
in	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

	notif1	ctrl 0	1	x	z
in	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

	notif0	ctrl 0	1	x	z
in	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

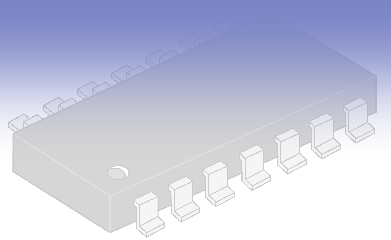
例、三态门的使用



- 用途
  - 多个驱动信号源共用一条信号线
  - 通过控制端时分复用，以免一条信号线同时被多个信号源驱动

```
//Instantiation of bufif gates.
bufif1 b1(out, in, ctrl);
bufif0 b0(out, in, ctrl);

//Instantiation of notif gates
notif1 n1(out, in, ctrl);
notif0 n0(out, in, ctrl);
```

门实例数组

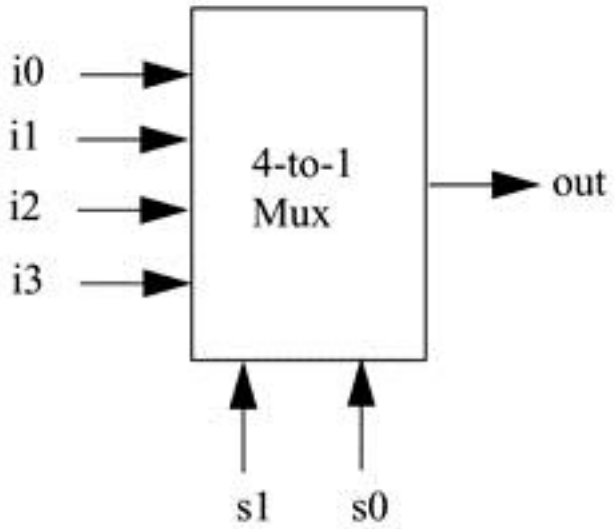


- 通过定义门实例数组，可以简化对某种类型的门多次调用
- 例

```
wire [7:0] OUT, IN1, IN2;

// 门实例数组引用
nand n_gate[7:0](OUT, IN1, IN2);

// 与下面 8 条实例引用语句相同
nand n_gate0(OUT[0], IN1[0], IN2[0]);
nand n_gate1(OUT[1], IN1[1], IN2[1]);
nand n_gate2(OUT[2], IN1[2], IN2[2]);
nand n_gate3(OUT[3], IN1[3], IN2[3]);
nand n_gate4(OUT[4], IN1[4], IN2[4]);
nand n_gate5(OUT[5], IN1[5], IN2[5]);
nand n_gate6(OUT[6], IN1[6], IN2[6]);
nand n_gate7(OUT[7], IN1[7], IN2[7]);
```

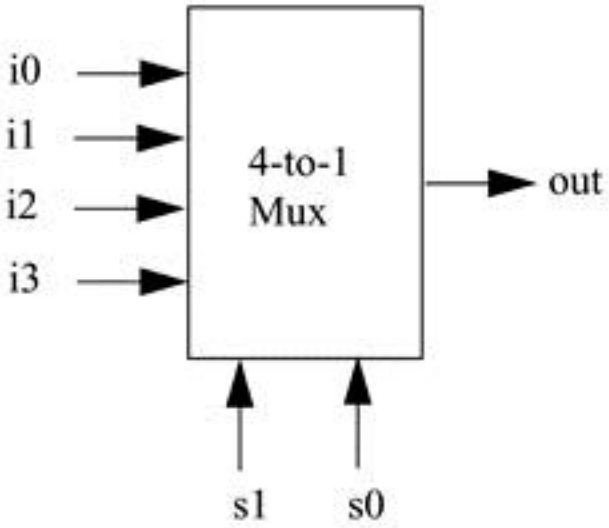
门级结构描述的数字电路设计

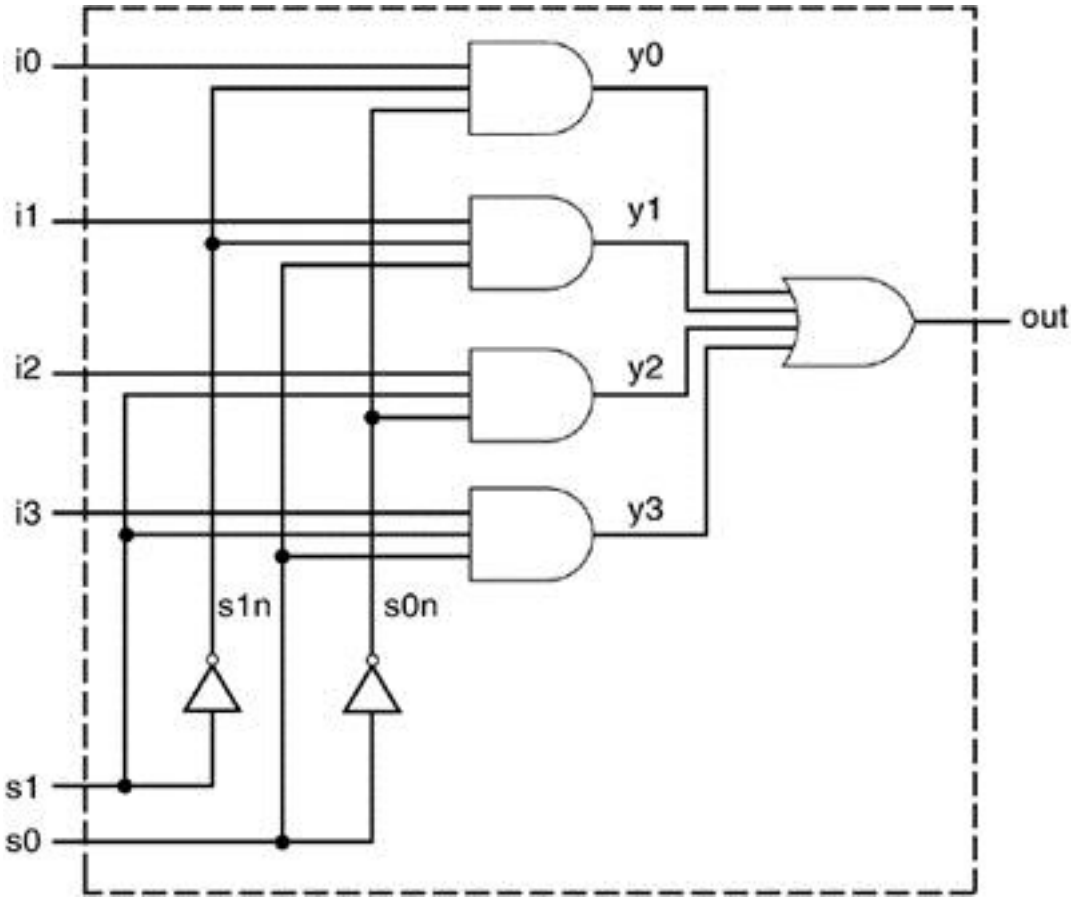
- 多路复用（选择）器
  - 根据控制信号从两个或多个输入源中选择一个输出
- 例
  - 设计一个带有两位选择信号的四选一多路选择器



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

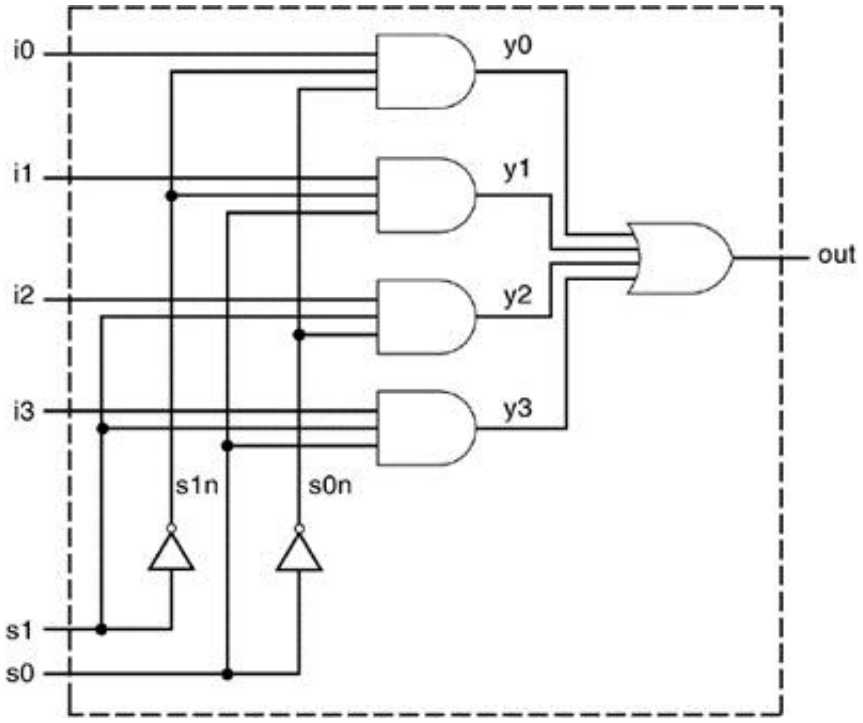
四选一多路选择器的逻辑图





多路选择器的Verilog门级描述

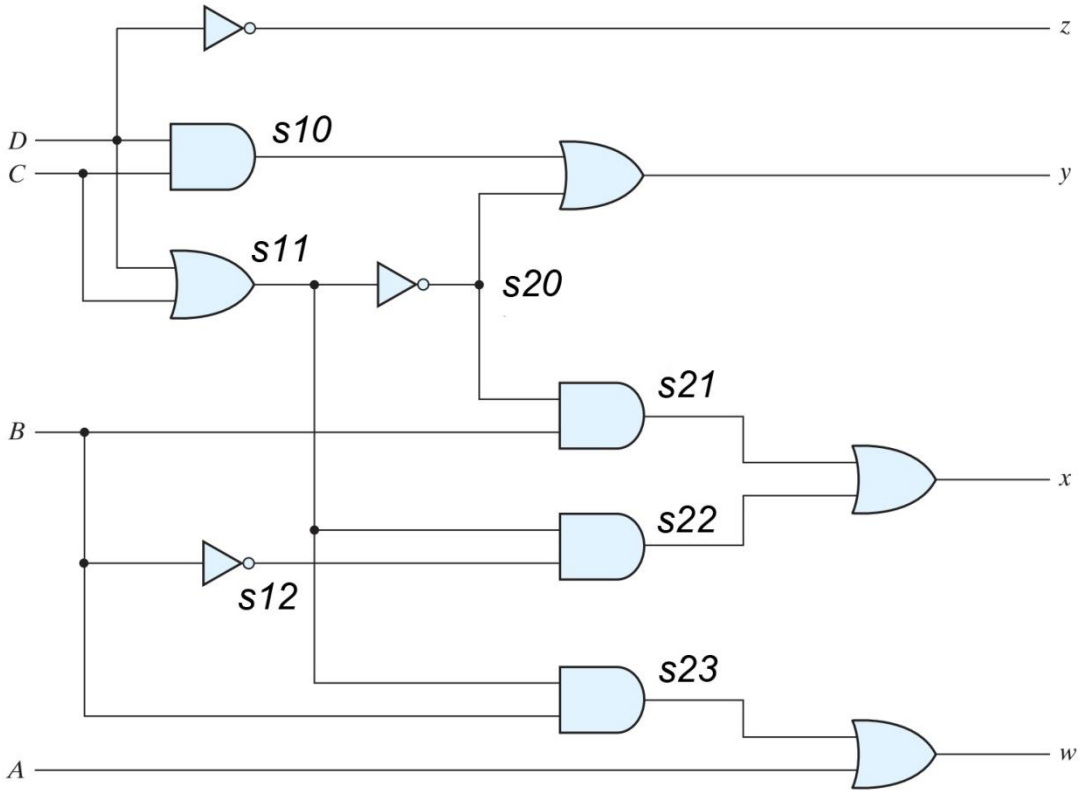
```
// Module 4-to-1 multiplexer.
module mux4_to_1 (output out, input i0, i1, i2, i3, s1, s0);
    // Internal wire declarations
    wire s1n, s0n;
    wire y0, y1, y2, y3;

    // Gate instantiations
    // Create s1n and s0n signals.
    not u10(s1n, s1),
        u11(s0n, s0);

    // 3-input and gates instantiated
    and  u20(y0, i0, s1n, s0n),
         u21(y1, i1, s1n, s0),
         u22(y2, i2, s1, s0n),
         u23(y3, i3, s1, s0);

    // 4-input or gate instantiated
    or u30(out, y0, y1, y2, y3);
endmodule
```



将BCD码转换余3码

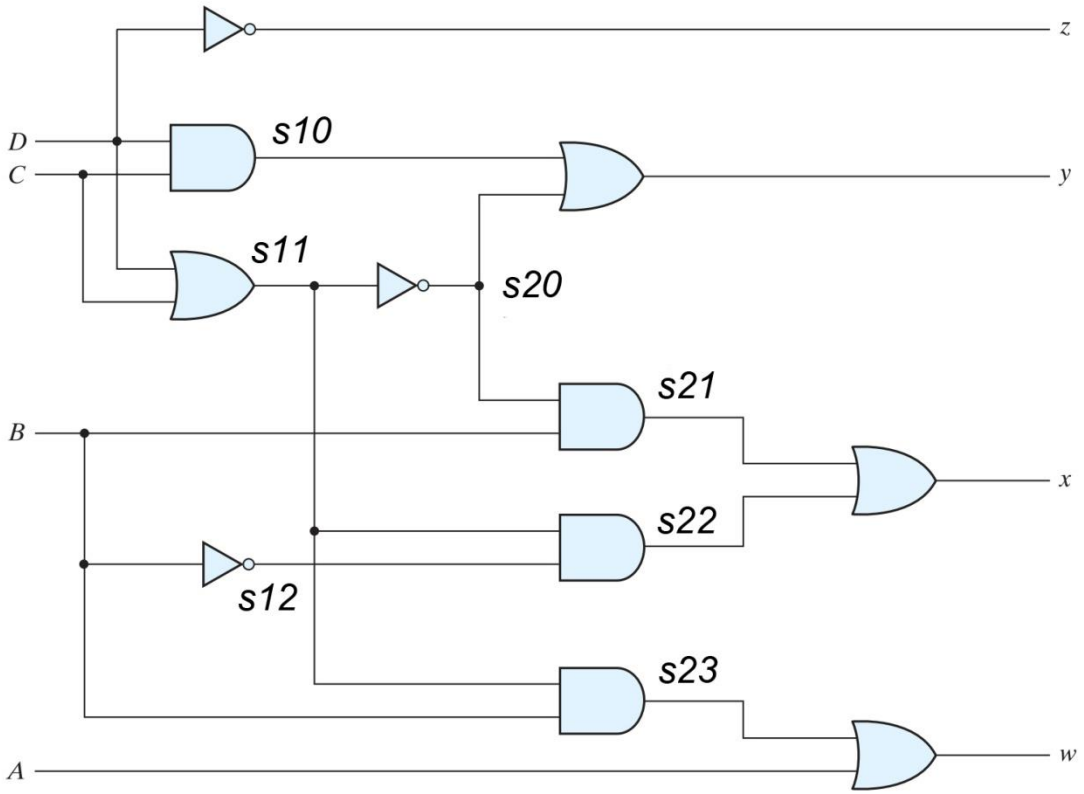
- BCD（ binary coded decimal，8421码）
- excess-3（余3码）
  - excess-3 = BCD + 0011

Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0



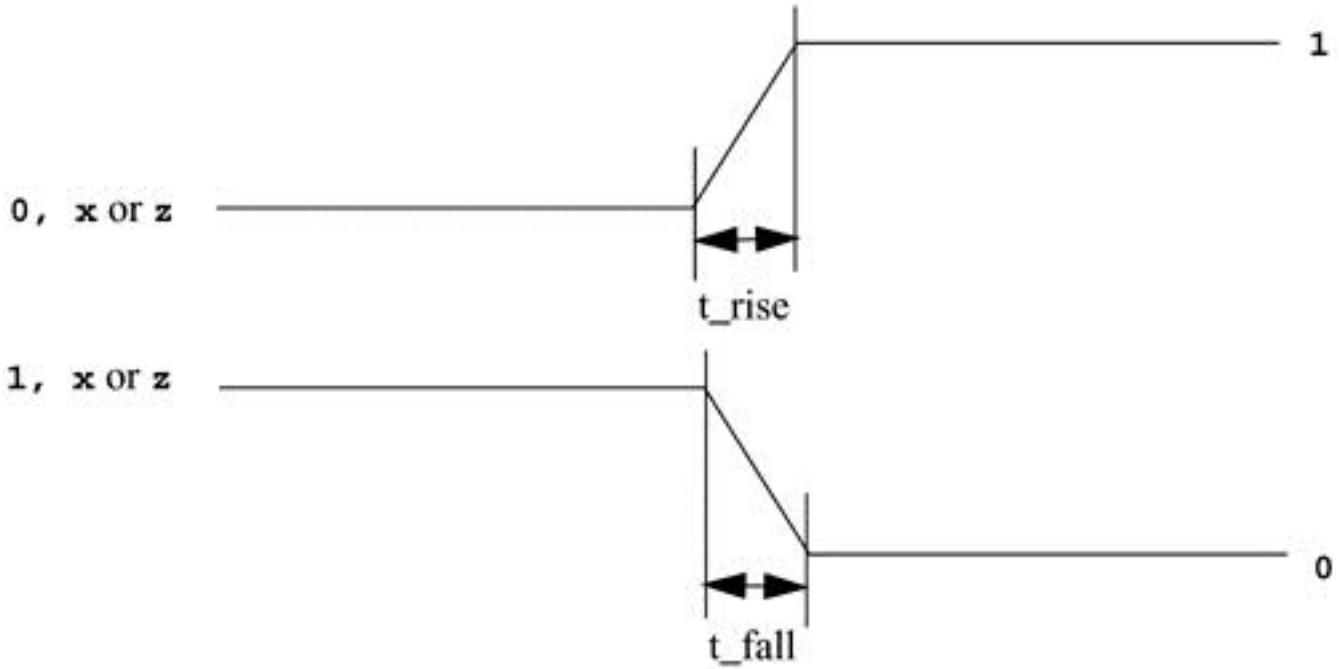
BCD 转换为余3码的Verilog门级描述

```
module BCD2excess3( output w, x, y, z,
                    input  A, B, C, D );

    wire s10, s11, s12;
    wire s20, s21, s22, s23;

    not u10(z, D);
    and u11(s10, D, C);
    or  u12(s11, D, C);
    not u13(s12, B);

    or  u20(y, s10, s20);
    not u21(s20, s11);
    and u22(s21, s20, B),
        u23(s22, s11, s12),
        u24(s23, s11, B);

    or  u30(x, s21, s22),
        u31(w, s23, A);
endmodule
```

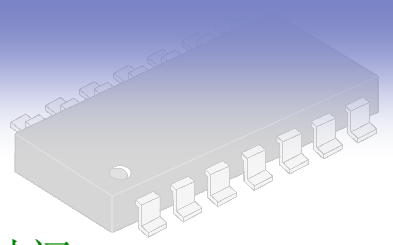


门延时

- 实际电路中，任何一个门都有延时
  - 在 Verilog 门级原语中，有 3 种从输入到输出的延时
    - 上升延时（Rise delay ）
      - 输入发生变化后，输出从 0，x，z 变化成高电平 1 所需的时间
    - 下降延时（Fall delay ）
      - 输入发生变化后，输出从 1，x，z 变化成低电平 0 所需的时间
    - 关断延时（Turn-off delay）
      - 输出从 0，1，x 变化成高阻抗 z 所需的时间



3 种延时说明

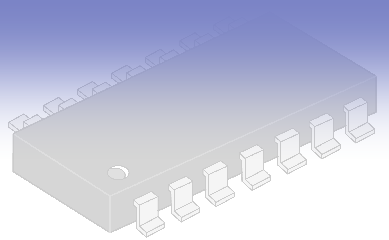


```
// 3种延时（上升、下降和关断）都等于 delay_time 表示的延时时间
and #(delay_time) a1(out, i1, i2);

// 说明了上升延时和下降延时时间，关断延时时间 = min(rise_val, fall_val)
and #(rise_val, fall_val) a2(out, i1, i2);

// 说明了上升延时、下降延时和关断延时时间
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

- 例

```
// Delay of 5 for all transitions
and #(5) a1(out, i1, i2);
// Rise = 4, Fall = 6
and #(4,6) a2(out, i1, i2);
// Rise = 3, Fall = 4, Turn-off = 5
bufif0 #(3,4,5) b1 (out, in, control);
```

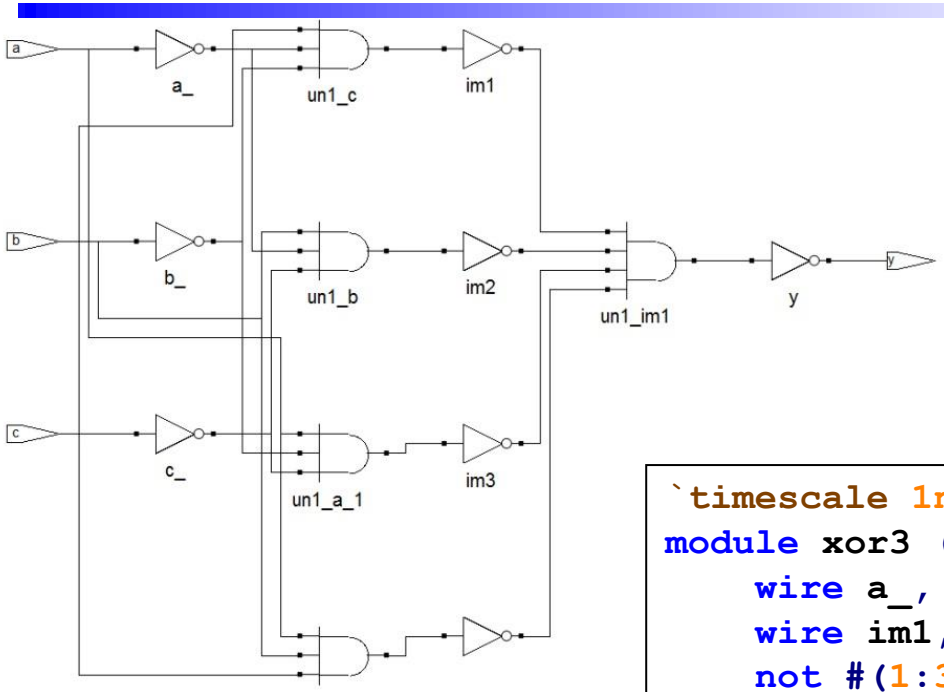
最大、最小和典型延时

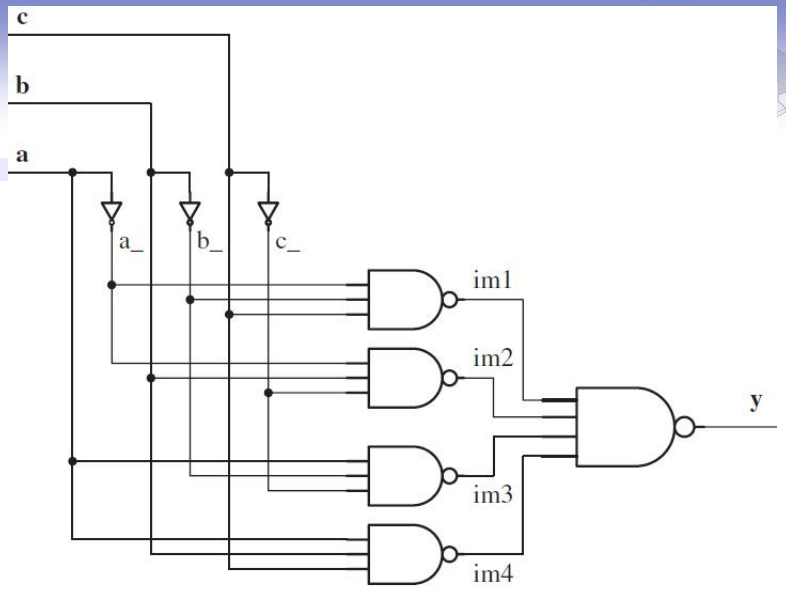


- 可以指定每种类型延时的最小值、典型值、最大值
  - 格式：minimum : typical : maximum
- 在仿真时决定选择使用哪一种延时——用户制定或使用缺省
- 例、

```
// One delay
// 如果选择使用 mindelays, delay= 4
// 如果选择使用 typdelays, delay= 5
// 如果选择使用 maxdelays, delay= 6
and #(4:5:6) a1(out, i1, i2);

// Two delays
// 如果选择使用 mindelays, rise= 3, fall= 5, turn-off = min(3,5)
// 如果选择使用 typdelays, rise= 4, fall= 6, turn-off = min(4,6)
// 如果选择使用 maxdelays, rise= 5, fall= 7, turn-off = min(5,7)
and #(3:4:5, 5:6:7) a2(out, i1, i2);

// Three delays
// 如果选择使用 mindelays, rise= 2 fall= 3 turn-off = 4
// 如果选择使用 typdelays, rise= 3 fall= 4 turn-off = 5
// 如果选择使用 maxdelays, rise= 4 fall= 5 turn-off = 6
and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
```

例、最大、最小和典型延时



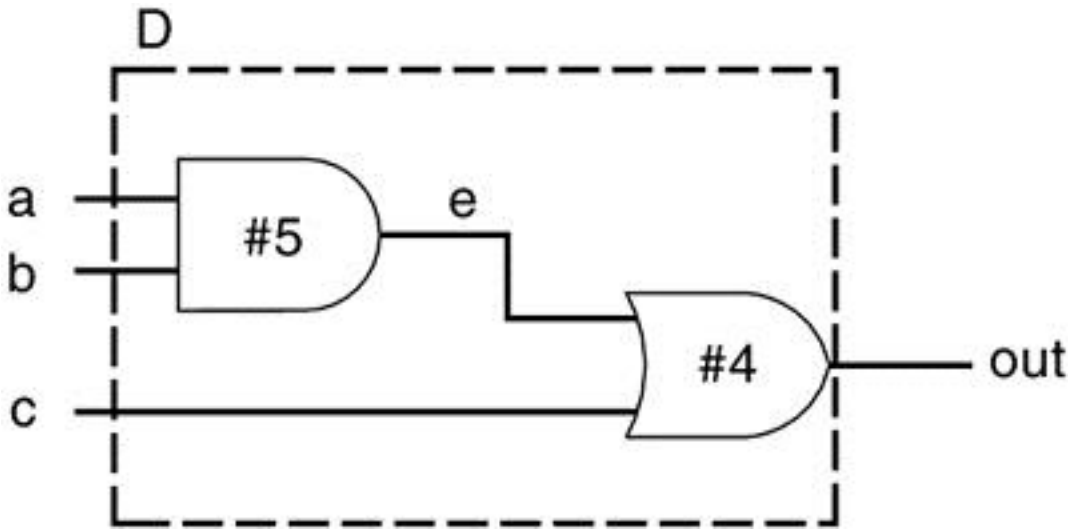


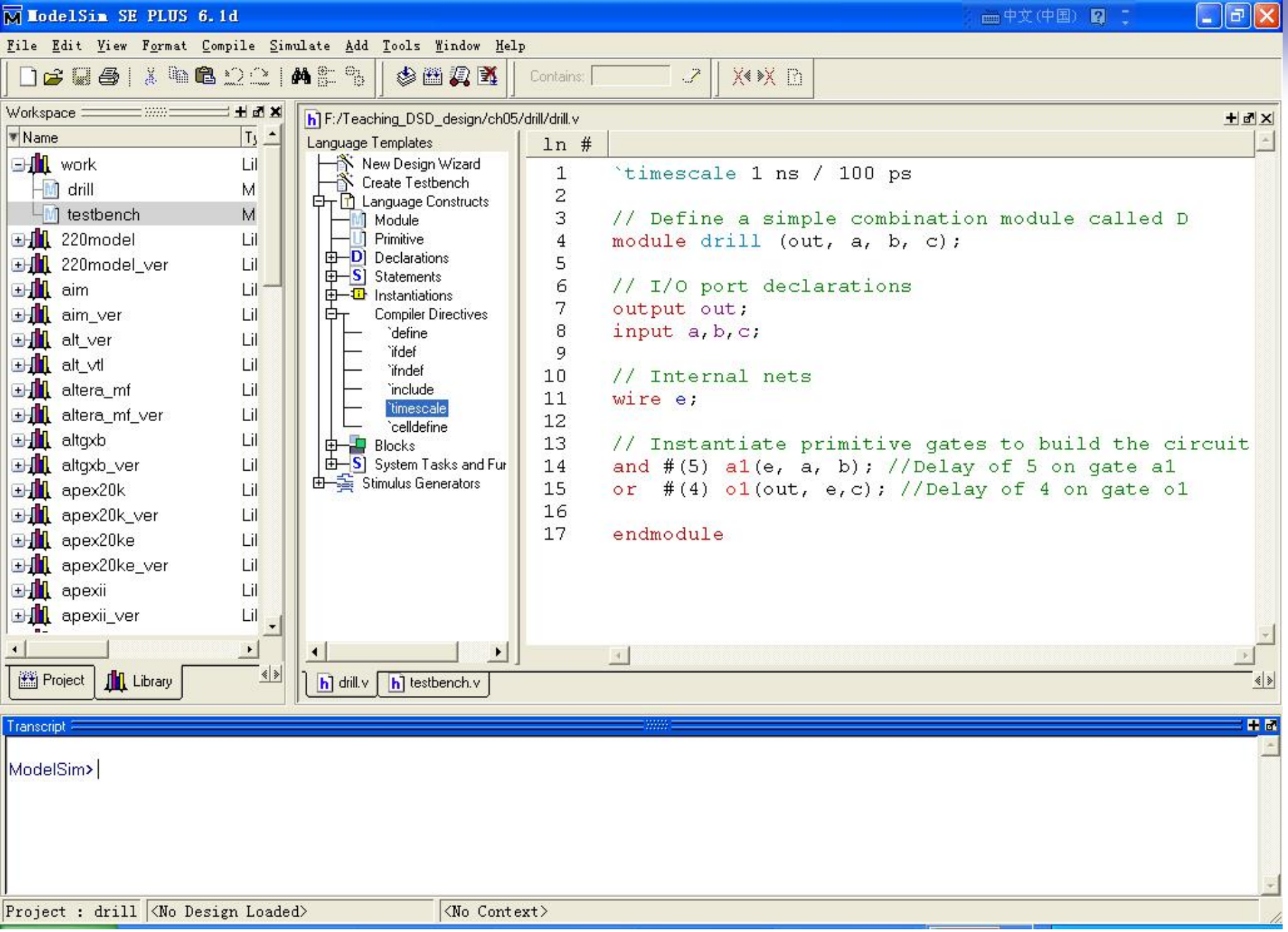

```
`timescale 1ns/100ps
module xor3 (output y, input a, b, c);
    wire a_, b_, c_;
    wire im1, im2, im3, im4;
    not #(1:3:5, 2:4:6)
        u10( a_, a ),
        u11( b_, b ),
        u12( c_, c );

    nand #(2:4:6, 3:5:7)
        u20( im1, a_, b_, c ),
        u21( im2, a_, b, c_ ),
        u22( im3, a, b_, c_ ),
        u23( im4, a, b, c );

    nand #(2:4:6, 3:5:7) u30(y, im1, im2, im3, im4);
endmodule
```

举例

- 建立逻辑电路的时序模型
  - 逻辑功能

$$out = (a \bullet b) + c$$




ModelSim SE PLUS 6.1d
File Edit View Format Compile Simulate Add Tools Window Help
Contains:
Workspace
Name
work
drill
testbench
220model
220model_ver
aim
aim_ver
alt_ver
alt_vtl
altera_mf
altera_mf_ver
altgxb
altgxb_ver
apex20k
apex20k_ver
apex20ke
apex20ke_ver
apexii
apexii_ver
Project
Library
F:/Teaching_DSD_design/ch05/drill/drill.v
Language Templates
New Design Wizard
Create Testbench
Language Constructs
Module
Primitive
Declarations
Statements
Instantiations
Compiler Directives
`define
`ifdef
`ifndef
`include
`timescale
`celldefine
Blocks
System Tasks and Fur
Stimulus Generators
ln #
1 `timescale 1 ns / 100 ps
2
3 // Define a simple combination module called D
4 module drill (out, a, b, c);
5
6 // I/O port declarations
7 output out;
8 input a,b,c;
9
10 // Internal nets
11 wire e;
12
13 // Instantiate primitive gates to build the circuit
14 and #(5) a1(e, a, b); //Delay of 5 on gate a1
15 or #(4) o1(out, e,c); //Delay of 4 on gate o1
16
17 endmodule
drill.v
testbench.v
Transcript
ModelSim>
Project : drill
<No Design Loaded>
<No Context>

F:/Teaching_DSD_design/ch05/drill/testbench.v

File Edit View Tools

Language Templates

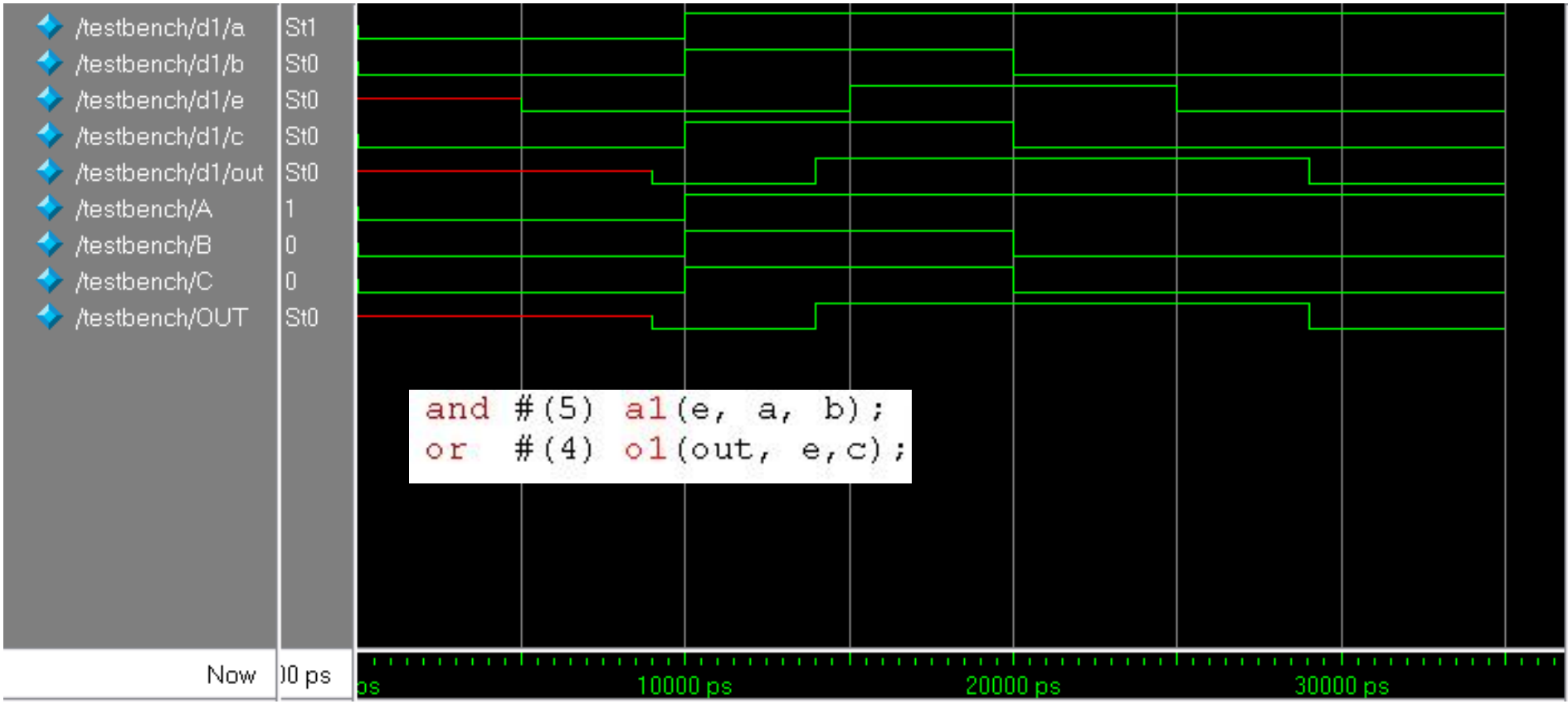
- New Design Wizard
- Create Testbench
- Language Constructs
  - Module
  - Primitive
  - Declarations
  - Statements
  - Instantiations
  - Compiler Directives
    - `define
    - `ifdef
    - `ifndef
    - `include
    - `timescale
    - `celldefine
  - Blocks
  - System Tasks and Fur
- Stimulus Generators

```
ln #
 1   `timescale 1 ns / 100 ps
 2
 3   // testbench (top-level module)
 4   module testbench;
 5
 6   // Declare variables
 7   reg A, B, C;
 8   wire OUT;
 9
10   // Instantiate the module D
11   drill d1( OUT, A, B, C);
12
13   // Stimulate the inputs. Finish the simulation at 40 time units.
14   initial
15   begin
16     A= 1'b0; B= 1'b0; C= 1'b0;
17
18     #10 A= 1'b1; B= 1'b1; C= 1'b1;
19
20     #10 A= 1'b1; B= 1'b0; C= 1'b0;
21
22     #20 $finish;
23   end
24
25   endmodule
26
27
```

Ln: 2 Col: 0

仿真结果

```
initial
begin
  A= 1'b0; B= 1'b0; C= 1'b0;

  #10 A= 1'b1; B= 1'b1; C= 1'b1;

  #10 A= 1'b1; B= 1'b0; C= 1'b0;

  #20 $finish;
end
```

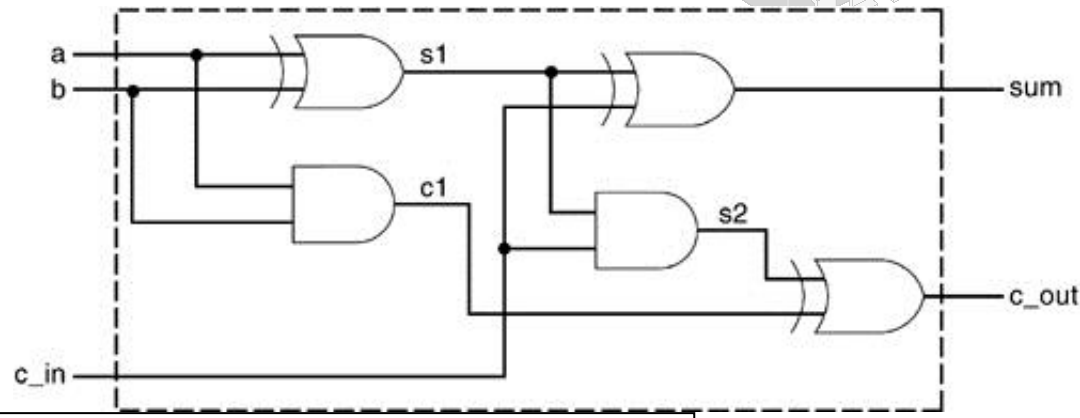


门级建模举例——四位脉动进位全加器

- 1位全加器的数学表达式

$$sum = (a \oplus b \oplus cin)$$

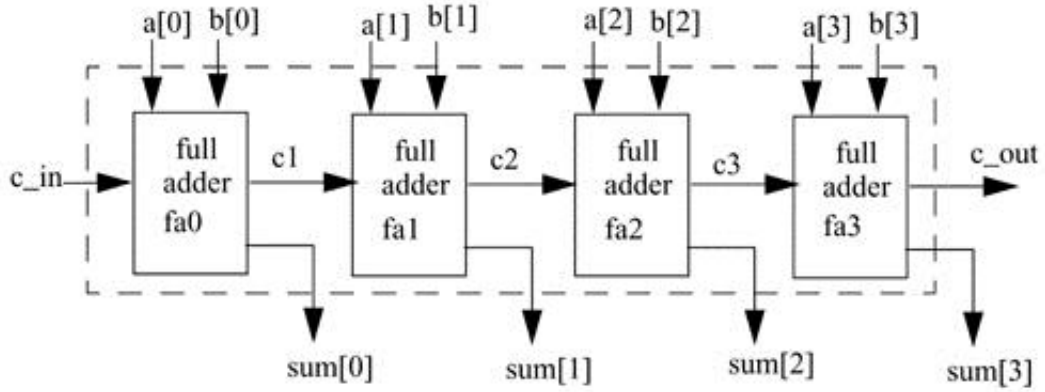
$$cout = (a \bullet b) \oplus cin \bullet (a \oplus b)$$




```
// Define a 1-bit full adder
module fulladd(output cout, sum, input a, b, cin);
    // Internal nets
    wire s1, c1, s2;

    // Instantiate logic gate primitives
    xor u10(s1, a, b);
    and u11(c1, a, b);

    xor u20(sum, s1, cin);
    and u21(s2,  s1, cin);

    xor u30(cout, s2, c1);
endmodule
```

四位脉动进位全加器

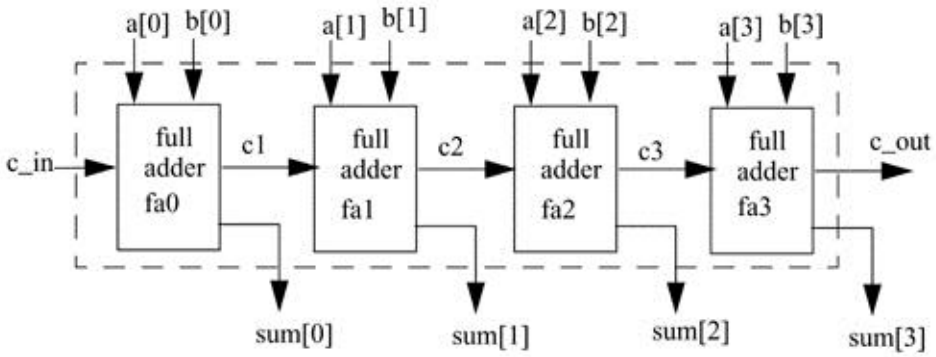
- 4位脉动进位全加器




```
`include "fulladd.v"
// Define a 4-bit full adder
module fulladd4(output cout,
                output [3:0] sum,
                input  [3:0] a, b,
                input  cin);
    // Internal nets
    wire c1, c2, c3;

    // Instantiate four 1-bit full adders.
    fulladd fa0(c1,   sum[0], a[0], b[0], cin);
    fulladd fa1(c2,   sum[1], a[1], b[1], c1 );
    fulladd fa2(c3,   sum[2], a[2], b[2], c2 );
    fulladd fa3(cout, sum[3], a[3], b[3], c3 );
endmodule
```

模块实例数组

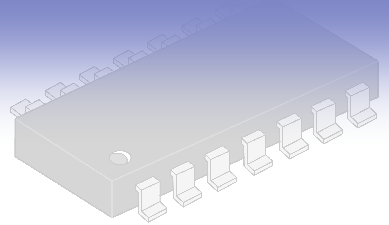
- 使用模块实例数组实现4位脉动进位全加器




```
`include "fulladd.v"
// Define a 4-bit full adder
module fulladd4(output cout,
                output [3:0] sum,
                input  [3:0] a, b,
                input  cin);
    // Internal nets
    wire c1, c2, c3;

    // Instantiate four 1-bit full adders.
    fulladd fa0(c1,   sum[0], a[0], b[0], cin);
    fulladd fa1(c2,   sum[1], a[1], b[1], c1 );
    fulladd fa2(c3,   sum[2], a[2], b[2], c2 );
    fulladd fa3(cout, sum[3], a[3], b[3], c3 );
endmodule
```

```
`include "fulladd.v"
// Define a 4-bit full adder
// 4-bit Adder Using Array of Instances
module fulladder4b(output cout,
                   output [3:0] sum,
                   input  [3:0] a, b,
                   input  cin);
    // Internal nets
    wire c1, c2, c3;

    // Instantiate four 1-bit full adders.
    fulladd fa0[0:3] ( {c1, c2, c3, cout}, sum, a, b, {cin, c1, c2, c3});
endmodule
```

激励模块和仿真结果



```
`include "fulladder4b.v"
// Define the stimulus (top level module)
module stimulus;
    // Set up variables
    reg [3:0] A, B;
    reg C_IN;
    wire [3:0] SUM;
    wire C_OUT;

    // Instantiate the 4-bit full adder. call it FA1_4
    fulladder4b FA1_4(SUM, C_OUT, A, B, C_IN);

    // Set up the monitoring for the signal values
    initial begin
        $monitor($time," A= %b, B=%b, C_IN= %b, --- C_OUT= %b, SUM= %b\n",
                 A, B, C_IN, C_OUT, SUM);
    end

    // Stimulate inputs
    initial begin
            A = 4'd0; B = 4'd0; C_IN = 1'b0;
        #5  A = 4'd3; B = 4'd4;
        #5  A = 4'd2; B = 4'd5;
        #5  A = 4'd9; B = 4'd9;
        #5  A = 4'd10; B = 4'd15;
        #5  A = 4'd10; B = 4'd5; C_IN = 1'b1;
    end
endmodule
```

```
 0 A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000
 5 A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111
10 A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111
15 A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010
20 A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001
25 A= 1010, B=0101, C_IN= 1, --- C_OUT= 1, SUM= 0000
```