



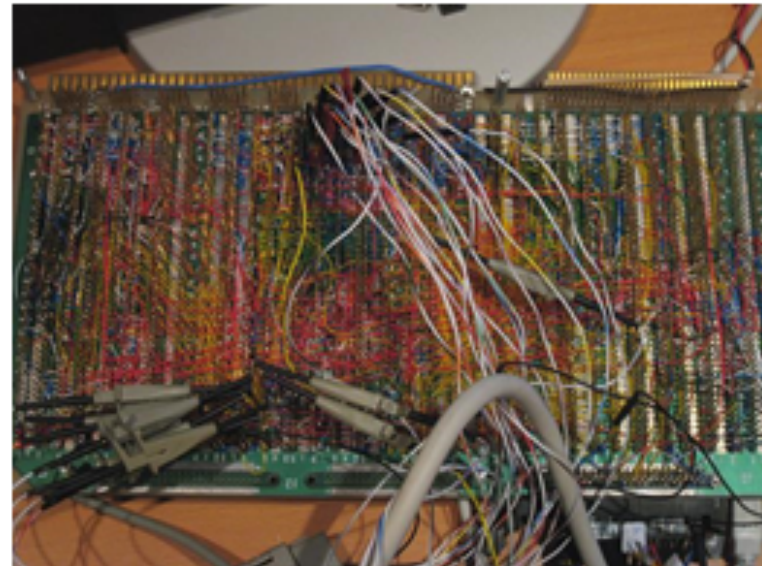
硬件编程语言Verilog

何杰

hejie@ustb.edu.cn

右图是采用了哪种数字系统实现方法

- A ASSP
- B PLD
- C ASIC





PLD实现方法中最常用的器件是哪几个?

- ☐ A FPGA
- ☐ B CPLD
- ☐ C MCU
- ☐ D CPU



基于EDA的数字系统设计方法关键要素是什么？

- ☐ A EDA工具
- ☐ B TOP-DOWN设计思想
- ☐ C 层次化设计方法
- ☐ D 硬件描述语言（HDL）



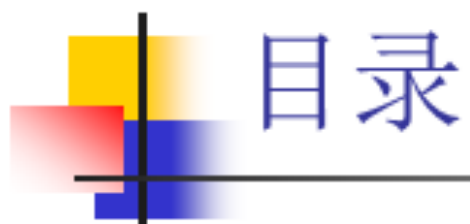
用HDL描述数字系统时，最常用的描述层次

- A 系统级
- B RTL级
- C 逻辑级
- D 电路级



那个家芯片公司有自己的芯片生产工厂

- A 高通
- B intel
- C 展讯
- D ARM
- E 华为



目录

- Verilog 简介
- Verilog设计及仿真概述
- Verilog基本语法
- Verilog电路描述



什么是硬件描述语言HDL

- 具有特殊结构能够对硬件逻辑电路的功能进行描述的一种高级编程语言
- 这种特殊结构能够：
 - 描述电路的连接
 - 描述电路的功能
 - 在不同抽象级上描述电路
 - 描述电路的时序
 - 表达具有并行性
- HDL主要有两种：**Verilog**和**VHDL**
 - **Verilog**起源于C语言，因此非常类似于C语言，容易掌握



Verilog的特点

- 描述电路的行为
- 依靠EDA工具综合出具体的电路
- 优点
 - 工艺无关性
 - 可移植性
 - 易于维护



Verilog的应用

- **VerilogHDL**是一种用于数字逻辑电路设计的语言，既可以用电路的功能描述也可以用元器件和它们之间的连接来建立电路的模型
- **Verilog**模型可以是实际电路的不同级别的抽象。这些抽象的级别和它们对应的模型类型共有以下五种
 - **系统级(system)**: 用高级语言结构实现设计模块的外部性能的模型
 - **算法级(algorithmic)**: 用高级语言结构实现设计算法的模型。
 - **RTL级(Register Transfer Level)**: 描述数据在寄存器之间流动和如何处理这些数据的模型。
 - **门级(gate-level)**: 描述逻辑门以及逻辑门之间的连接的模型。
 - **开关级(switch-level)**: 描述器件中三极管和储存节点及其之间连接的模型



Verilog for RTL Design

- 面向RTL的Verilog语法
 - 面向综合的Verilog语法子集
 - 面向测试的Verilog语法子集
- 目的
 - 学会如何用语言进行电路设计



Verilog HDL与C语言比较

- **Verilog HDL**是在C语言基础上发展起来的，保留了C语言的结构特点。
- 但C语言的各函数之间是串行的，而Verilog的各个模块间是并行的

C语言	Verilog	功能	C语言	Verilog	功能
+	+	加	>=	>=	大于等于
-	-	减	<=	<=	小于等于
*	*	乘	==	==	等于
/	/	除	!=	!=	不等于
%	%	取模	~	~	取反
!	!	逻辑非	&	&	按位与
&&	&&	逻辑与			按位或
		逻辑或	^	^	按位异或
>	>	大于	<<	<<	左移
<	<	小于	>>	>>	右移

设计实例—初探Verilog

```
module Add_half ( sum, c_out, a, b );
```

```
  input  a, b;
```

```
  output sum, c_out;
```

```
  wire   c_out_bar;
```

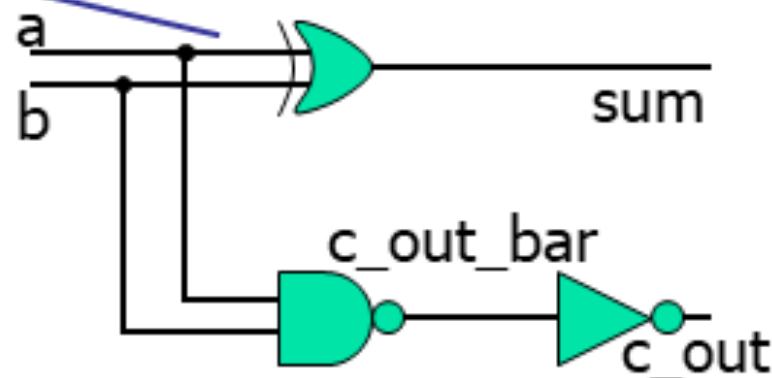
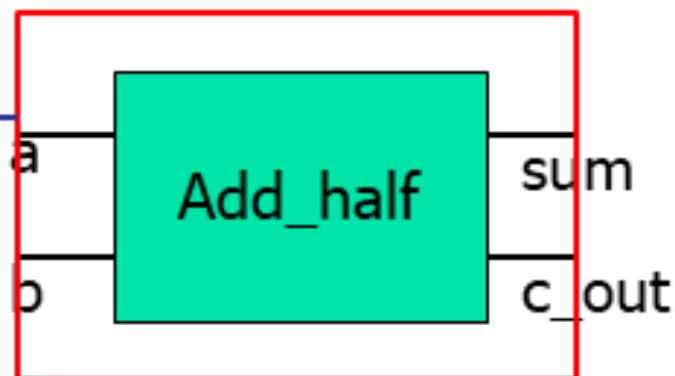
```
  xor (sum, a, b);
```

```
  nand (c_out_bar, a, b);
```

```
  not (c_out, c_out_bar);
```

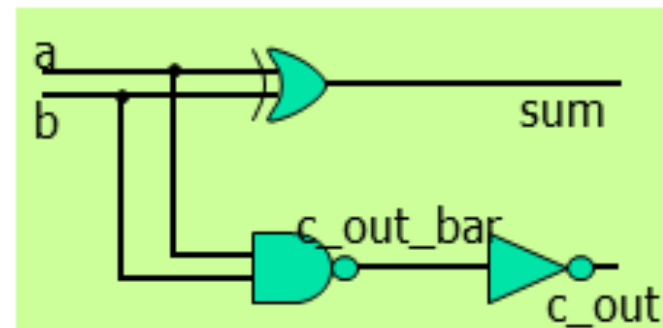
```
endmodule
```

- 一个复杂的电路的完整Verilog HDL模型是由若干个Verilog HDL模块构成的，每一个模块又可以由若干个子模块构成



设计实例—初探Verilog

```
module Add_half (Module name sum, c_out, a, b); Module ports  
  input a, b; Declaration of port modes  
  output sum, c_out; Declaration of port modes  
  wire c_out_bar; Declaration of internal signal  
  
  xor (sum, a, b); Instantiation of primitive gates  
  nand (c_out_bar, a, b); Instantiation of primitive gates  
  not (c_out, c_out_bar); Instantiation of primitive gates  
  
endmodule Verilog keywords
```



行为描述

```
module Add_half ( sum, c_out, a, b );
```

```
  input      a, b;
```

```
  output     sum, c_out;
```

```
  reg sum, c_out;
```

```
  always @ ( a or b )
```

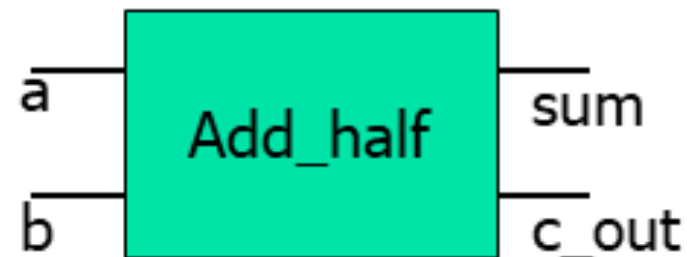
```
    begin
```

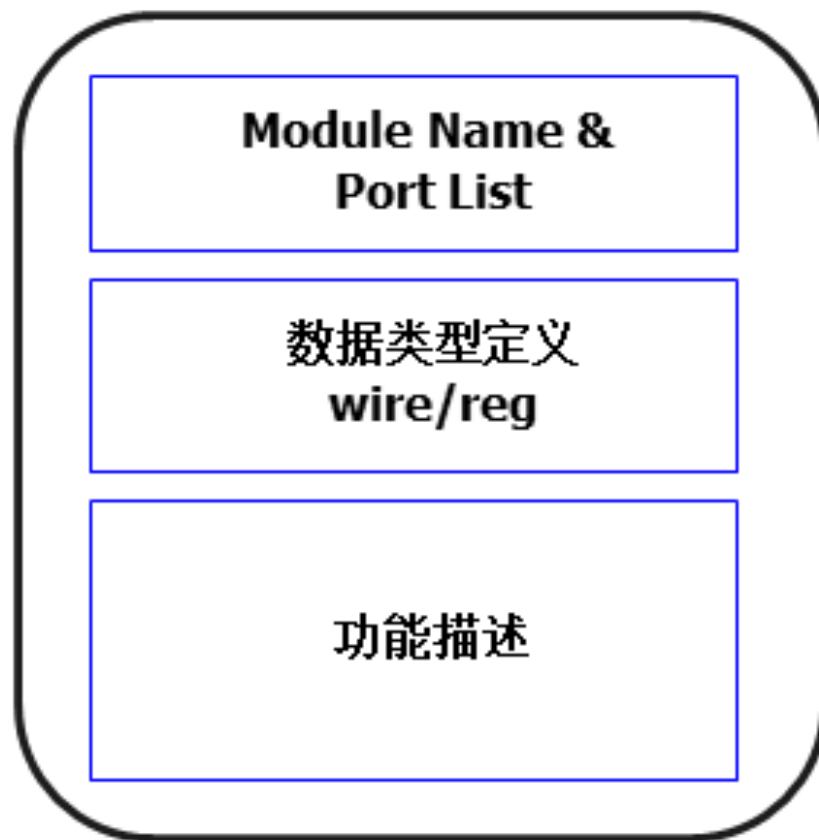
```
      sum = a ^ b;           // Exclusive or
```

```
      c_out = a & b; // And
```

```
    end
```

```
endmodule
```





Verilog代码区分大小写

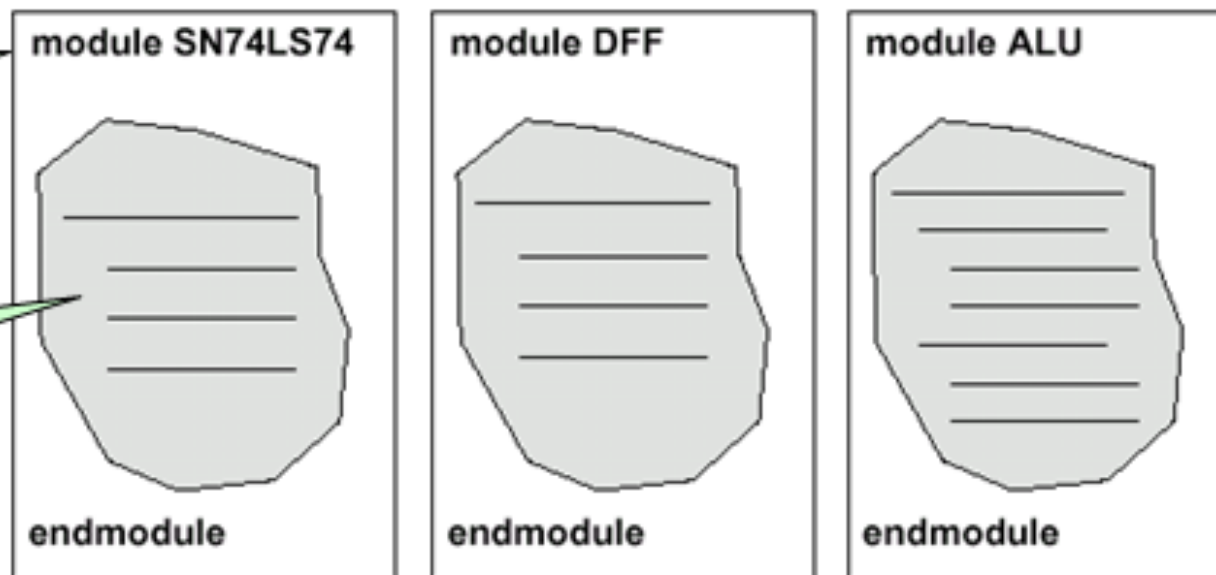


VERILOG设计及仿真概述

语言的主要特点—module模块

module是层次化设计的基本构件

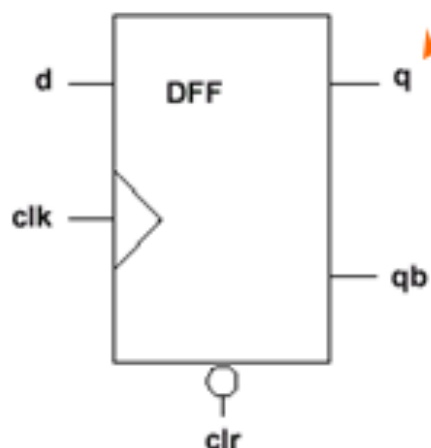
逻辑描述放在**module**内部



- **module**能够表示：
 - 物理块，如IC或ASIC单元
 - 逻辑块，如一个CPU设计的ALU部分
 - 整个系统
- 每一个模块的描述从关键词 **module** 开始，有一个名称（如 SN74LS74, DFF, ALU 等等），由关键词 **endmodule** 结束。

语言的主要特点—模块端口

端口等价于硬件的引脚(pin)



```
module DFF (d, clk, clr, q, qb);  
  input d, clk, clr;  
  output q, qb;  
  
  // Internal logic represented by a grey octagon  
  
endmodule
```

端口在模块名字后的括号中列出

端口声明语句，用来进行端口方向的说明。Verilog语言中有三种端口声明语句：

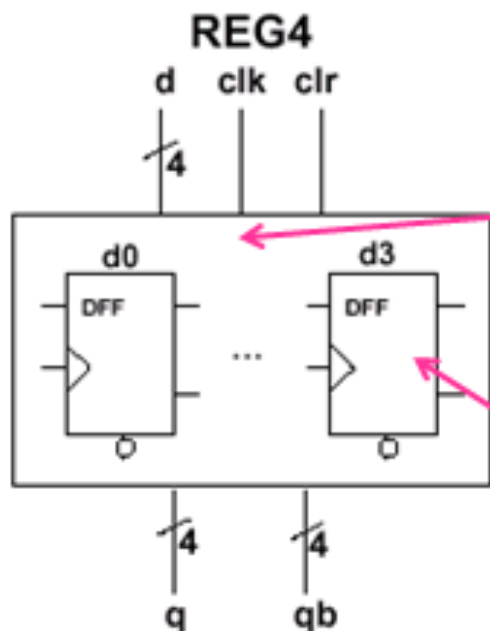
input - 对应的端口是输入端口

output - 对应的端口是输出端口

inout - 对应的端口是双向端口

- 注意模块的名称**DFF**，端口列表及说明
- 模块通过端口与外部通信

语言的主要特点—模块实例化



```
module DFF (d, clk, clr, q, qb);
```

```
....  
endmodule
```

```
module REG4( d, clk, clr, q, qb);
```

```
output [3: 0] q, qb;
```

```
input [3: 0] d;
```

```
input clk, clr;
```

```
DFF d0 (d[0], clk, clr, q[0], qb[ 0]);
```

```
DFF d1 (d[1], clk, clr, q[1], qb[ 1]);
```

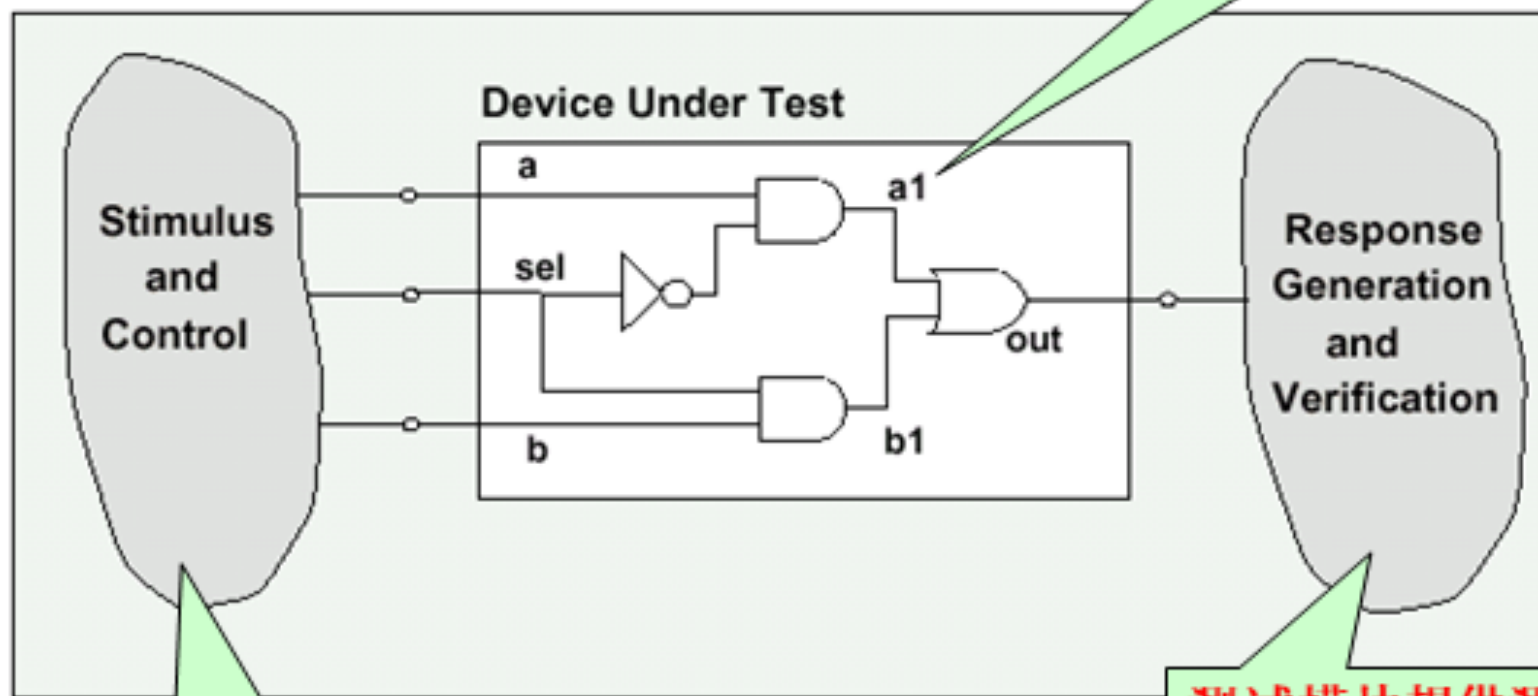
```
DFF d2 (d[2], clk, clr, q[2], qb[ 2]);
```

```
DFF d3 (d[3], clk, clr, q[3], qb[ 3]);
```

```
endmodule
```

一个完整例子

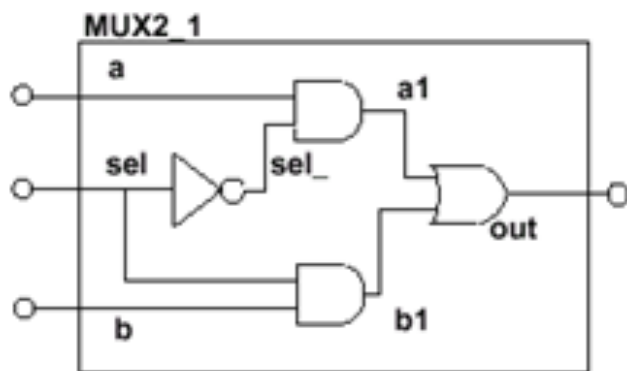
待测试模块DUT
是一个二选一选
通器



测试模块采用行
为级描述

测试模块提供测
试激励及验证机
制

被测器件DUT(device under test)



- 测试模块可以通过模块名及端口说明使用多路器。实例化多路器时不需要知道其实现细节。这正是自上而下设计方法的一个重要特点。模块的实现可以是行为级也可以是门级，但并不影响高层次模块对它的使用。

```
module MUX2_1 (out, a, b, sel);
    // Port declarations
    output out;
    input a, b, sel;
    wire out, a, b, sel;
    wire sel_, a1, b1;
    // The netlist
    not (sel_, sel);
    and (a1, a, sel_);
    and (b1, b, sel);
    or (out, a1, b1);
endmodule
```

多路器由关键词 **module** 和 **endmodule** 开始及结束。

已定义的 Verilog 基本单元的实例



测试模块



为什么没有端口?

```
module mux_tb;  
    // Data type declaration  
  
    // Instantiate modules  
  
    // Apply stimulus  
  
    // Display results  
  
endmodule
```

由于mux_tb已经是最顶层模块，不会被其它模块实例化。因此不需要有端口。

测试模块

```
module mux_tb;  
    // Data type declaration  
  
    // Instantiate modules  
    MUX2_1 mux (out, a, b, sel);  
    // Apply stimulus  
  
    // Display results  
  
endmodule
```

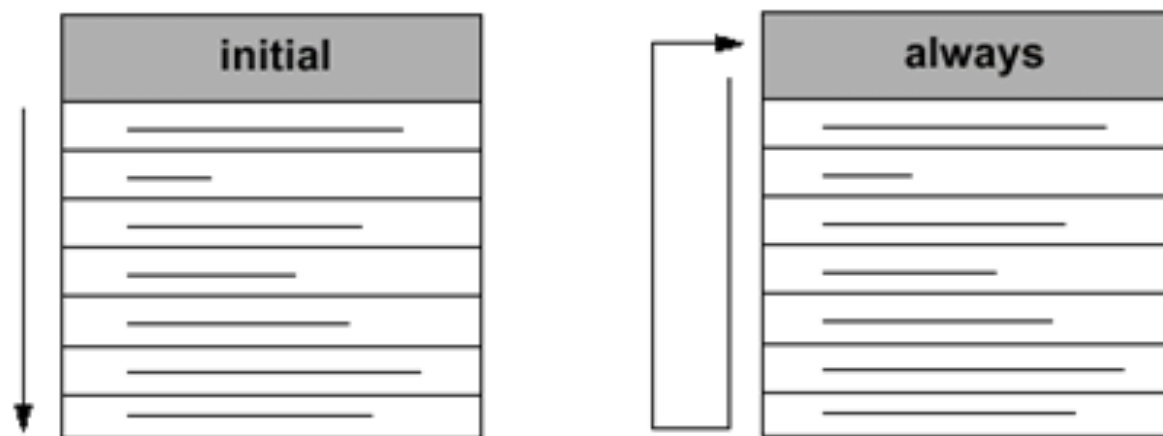
多路器实例化语句

MUX的实例化语句包括：

- 模块名字：与引用模块相同
- 实例名字：任意，但要符合标记命名规则
- 端口列表：与引用模块的次序相同

测试模块 - 过程

- 过程语句有两种：
 - *initial* : 只执行一次
 - *always* : 循环执行



通常采用过程语句进行行为级描述

激励信号在过程语句中描述

所有过程在时间0执行一次；过程之间是并行执行的

测试模块 - 激励描述

Time	Values		
	a	b	sel
0	0	1	0
5	0	0	0
10	0	1	1
15	1	1	1

```
module mux_tb;
// Data type declaration
  reg a, b, sel;
  wire out;
// MUX instance
  MUX2_1 mux (out, a, b, sel);
// Apply stimulus
  initial
  begin
    a = 0; b = 1; sel = 0;
    #5 b = 0;
    #5 b = 1; sel = 1;
    #5 a = 1;
    #5 $finish;
  end
// Display results
endmodule
```

只有always/initial过程的输出才需要定义为reg

端口信号的缺省类型是wire, 可以省略

- a, b, sel说明为reg类数据。reg是寄存器类数据, 在重新赋值前一直保持当前值
- #5 表示等待5个时间单位
- \$finish是结束仿真的系统任务



测试模块 – 响应输出

- **Verilog**提供了一些系统任务和系统函数，包括
 - *\$time* 系统函数，给出当前仿真时间
 - *\$monitor* 系统任务，若参数列表中的参数值发生变化，则在时间单位末显示参数值

`$monitor ([“format_specifiers”,] <arguments>);`

例如：

`$monitor($time, o, in1, in2);`

`$monitor($time, , out, , a, , b, , sel);`

`$monitor($time, “%b %h %d %o”, sig1, sig2, sig3, sig4);`

测试激励

```
module mux_tb;
  // Data type declaration
  reg a, b, sel;
  wire out;
  // MUX instance
  MUX2_1 mux (out, a, b, sel);
  // Apply stimulus
  initial begin
    a = 0; b = 1; sel = 0;
    #5 b = 0; #5 b = 1; sel = 1;
    #5 a = 1;
    #5 $finish;
  end
  // Display results
  initial
    $monitor($time," out=%b a=%b b=%b sel=%b", out, a, b, sel);
endmodule
```

结果输出

```
0      out= 0   a= 0   b= 1
sel= 0
5      out= 0   a= 0   b= 0
sel= 0
10     out= 1   a= 0   b= 1
sel= 1
15     out= 1   a= 1   b= 1
sel= 1
```



VERILOG基本语法



Verilog语言要素

- 空格和注释
- 操作符
- 数
- 标识符
- 特殊符号
- 关键词

2018/11/15

33/71

空格和注释

- **Verilog** 是一种格式很自由的语言，代码跨行和在一行内编写是一样的。
- 空格只起到分隔符的作用，没有其他用处。

```
initial begin  clk = 0;  forever  #10 clk = ~clk ;  
end
```

```
/******  
   code to generate clock signal  
******/
```

← 多行注释

```
initial begin  
  clk = 0;  
  forever  
    #10 clk = ~clk;    // invert clock  
end
```

← 单行注释

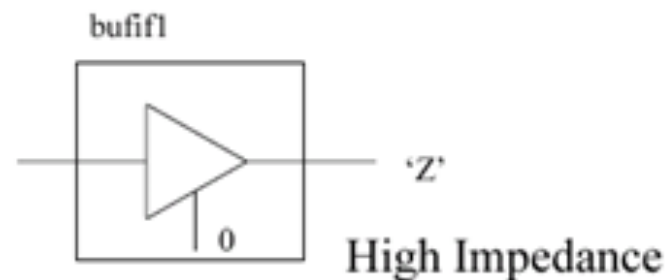
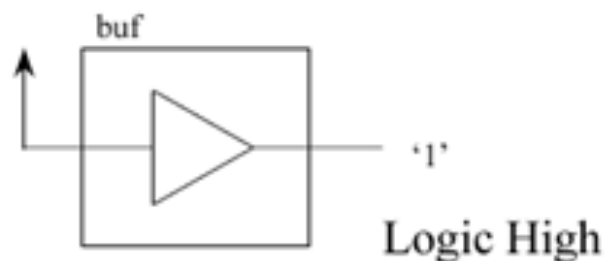
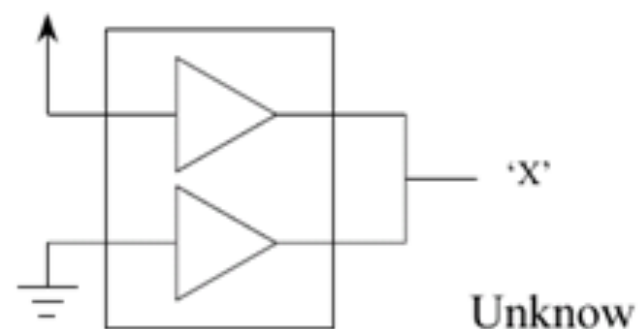
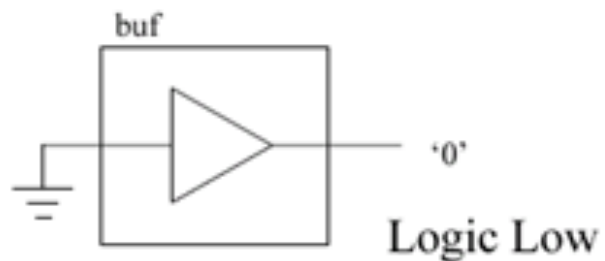


常用的操作符

- 算术操作: $+$, $-$, $*$, $/$, $\%$ (取模)
- 关系操作: $>$, $<$, $>=$, $<=$, $==$ (相等), $!=$ (不等)
- 逻辑操作: $\&\&$, $||$, $!$,
- 位操作: \sim (非), $\&$, $|$, \wedge (异或), $\sim\wedge$ (同或)
- 归约操作: $\&$, $\sim\&$, $|$, $\sim|$, \wedge , $\sim\wedge$
- 移位操作: $<<$, $>>$
- 条件操作: $?:$
- 连接和复制: $\{ , \}$

Verilog的四值逻辑

■ 0, 1, X, Z



数的表示方

X可以用来定义十六进制数的**4**位二进制状态，八进制数的**3**位，二进制数的**1**位。**Z**的表示方法同**X**类似。

- **<size>'<base><value>**
 - **Size:** 以bit为单位
 - **Base:** b(二进制),o(八进制),d(十进制),h(16进制)
 - **Value:**和进制相应的数值, **x,z,?** (**x,z**不区分大小写)
- 例
 - **16** //32位十进制数
 - **8'd16**
 - **8'h10**
 - **8'b0001_0010**
 - **32'bx** //32位x
 - **2'b?** **?、z、Z**都表示高阻

标识符

- 标识符就是用户为程序中的**Verilog** 对象所起的名字
- **a-z, A-Z, 0-9, _, \$**
- 标识符必须以英语字母 (**a-z, A-Z**) 起头, 或者用下横线符 (**_**) 起头
- 标识符最长可以达到**1023**个字符
- **Verilog**语言是大小写敏感的

```
module adder(a,b,cin,s,cout);  
  input a,b,c;  
  output s,cout;  
  ...  
endmodule
```

标识符

关键词

- **Verilog** 是大小写敏感的
- 所有的**Verilog** 关键词都是小写的

always	and	assign	begin	buf
bufif0	bufif1	case	casez	disable
cmos	deassign	default	defparam	endfunction
edge	else	end	endcase	endtask
endmodule	endprimitive	endspecify	endtable	fork
event	for	force	forever	initial
function	highz0	highz1	if	large
inout	input	integer	join	negedge
macromodule	medium	module	nand	notif1
nmos	nor	not	notif0	rcmos
pull0	pull1	pulldown	pullup	rpmos
reg	release	repeat	rmos	small
rtran	rtranif0	rtranif1	scalared	supply0
specify	specparam	strong0	strong1	tran
supply1	table	task	time	tri0
tranif0	tranif1	tri	tri0	tri1
triand	trior	vectored	wait	wand
weak0	weak1	while	wire	wor
xnor	xor			



特殊符号 “#”

- 特殊符号 “#” 表示延迟

- 过程赋值语句里的延迟

```
initial begin    #10 rst=1;    end
```

- 门级实例引用的延迟

```
not #1 not1(nsel, sel);
```

常用数据类型

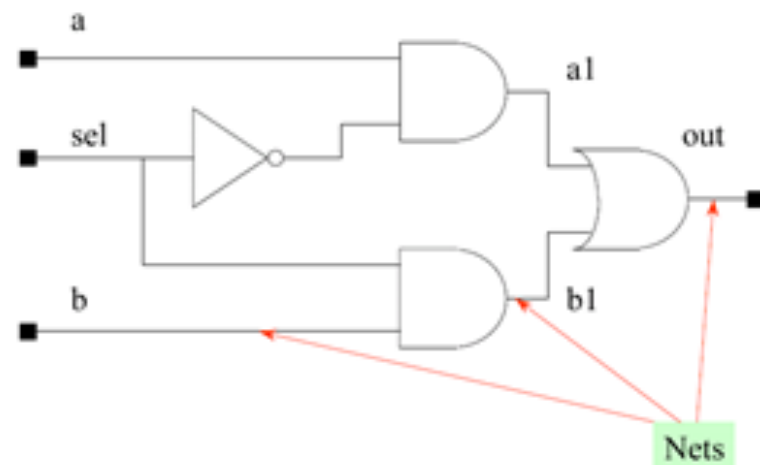
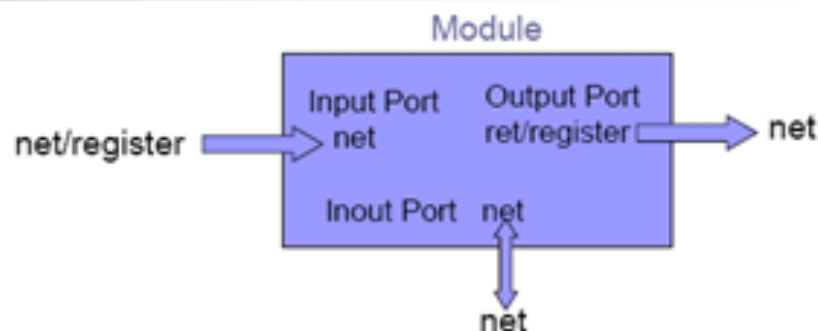
■ wire (线网)

- 表示元件之间的物理连接
- `wire a,b;`
- `wire [3:0] address;`
- Wire的缺省值是z

■ reg (寄存器)

- `always/initial`过程的输出
- 不要和flip-flop(D触发器)混淆
- `reg [1:5] state, newstate;`
- reg的缺省值是x

■ 除了应该定义为reg的都定义为wire





线网声明

wire型变量的声明格式:

wire [n-1:0] 数据名1, 数据名2,, 数据名m;

reg型变量的声明格式:

reg [n-1:0] 数据名1, 数据名2,, 数据名m;



net和register声明举例

```
reg a;                // 一个一位寄存器
wire w;              // 一个一位wire类型net
reg [3: 0] v;         // 从MSB到LSB的4位寄存器
reg [7: 0] m, n;      // 两个8位寄存器
tri [15: 0] busa;     // 16位三态总线
wire [31: 0] databus, addrbus; // 两个32位总线
wire [32: 1] abus, bbus;
```



例

- 多位**wire** 型数据可按下面方法使用

wire[7:0] in, out; //定义两个8位wire型向量

assign out=in; // assign 就是持续赋值语句

- 使用多位数据中的几位

wire[7:0] out;

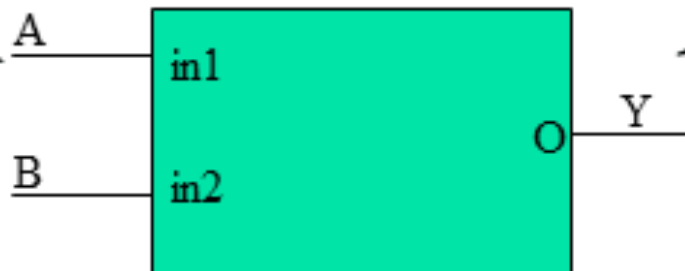
wire[3:0] in;

assign out[5:2]=in;

选择正确的数据类型

输入端口可以由
net/register驱动，但
输入端口只能是net

双向端口输入/输出
只能是net类型



输出端口可以是
net/register类型，输
出端口只能驱动net

```
module top;
wire y;
reg a, b;
  DUT u1 (y, a, b);
  initial begin
    a = 0; b = 0;
    #5 a = 1;
  end
endmodule
```

在过程块中只能给
register类型赋值

```
module DUT (Y, A, B);
output Y;
input A, B;
wire Y, A, B;
  and (Y, A, B);
endmodule
```

若Y, A, B说明为
reg则会产生错误。

存储器(memory)

- 在Verilog中可以说明一个寄存器数组。

`integer NUMS [7: 0]; // 包含8个整数数组变量`

- `reg`类型的数组通常用于描述存储器

其语法为: `reg [MSB:LSB] <memory_name> [first_addr:last_addr];`

`[MSB:LSB]`定义存储器字的位数

`[first_addr:last_addr]`定义存储器的深度

例如:

`reg [15: 0] MEM [0:1023]; // 1K x 16存储器`

`reg [7: 0] PREP ['hFFFE: 'hFFFF]; // 2 x 8存储器`

- 描述存储器

`reg [15: 0] MEM3 [1023: 0];`

存储器寻址

- 存储器元素可以通过存储器索引 (**index**) 寻址, 也就是给出元素在存储器的位置来寻址。

`mem_name [addr_expr]`

- Verilog** 不支持多维数组。也就是说只能对存储器字进行寻址, 而不能对存储器中一个字的位寻址。

```
module mems;  
  reg [8: 1] mema [0: 255]; // declare memory called mema  
  reg [8: 1] mem_word;      // temp register called mem_word  
  ...  
  initial  
  begin  
    $displayb( mema[5]);    //显示存储器中第6个字的内容  
    mem_word = mema[5];  
    $displayb( mem_word[8]); // 显示第6个字的最高有效位  
  end  
endmodule
```

若要对存储器字的某些位存取, 只能通过暂存器传递



说明

- 寄存器赋值可以在一条赋值语句中完成，但是存储器不可以。因此在存储器被赋值时，需要定义一个索引。

```
reg [5:1] Dig;    // Dig为5位寄存器。
```

```
...
```

```
Dig = 5'b11011;  // 赋值正确
```

```
reg Bog[5:1];    // Bog为5个1位寄存器组成的的存储器组
```

```
...
```

```
Bog = 5'b11011;  // 赋值不正确
```



系统任务和编译预处理



1、系统任务

- **\$<identifier>**
- **\$**符号指示这是系统任务和函数
- 系统函数有很多，如
 - 返回当前仿真时间**\$time**
 - 显示/监视信号值(**\$display, \$monitor**)
 - 停止仿真**\$stop**
 - 结束仿真**\$finish**

\$monitor(\$time, "a = %b, b = %h", a, b);

当信号**a**或**b**的值发生变化时，系统任务**\$monitor**显示当前仿真时间，信号**a**值(二进制格式), 信号**b**值（16进制格式）。



2、编译预处理

- 符号说明一个编译预处理
- 这些编译预处理使仿真编译器进行一些特殊的操作
- 编译预处理一直保持有效直到被覆盖或解除

3、宏定义- `define

- 提供了一种简单的文本替换的功能

`define <macro_name> <macro_text>

在编译时<macro_text>替换<macro_name>。可提高描述的可读性。

```
`define not_delay #1
`define and_delay #2
`define or_delay #1
module MUX2_1 (out, a, b, sel);
output out;
input a, b, sel;
    not `not_delay not1( sel_, sel);
    and `and_delay and1( a1, a, sel_);
    and `and_delay and2( b1, b, sel);
    or `or_delay or1( out, a1, b1);
endmodule
```

定义not_delay

使用not_delay

4、文本包含-`include

- 在当前内容中插入一个文件

格式: ``include "<file_name>"`

如 ``include "global.v"`

``include "parts/count.v"`

``include "../..../library/mux.V"`

可以是相对路径或绝对路径

- 说明

- **include**保存文件中的全局的或经常用到的一些定义, 如文本宏
- 一个**`include**命令只能指定一个被包含的文件
- 文件包含是可以嵌套的

5、时间尺度-timescale

- **`timescale** 说明时间单位及精度

格式: **`timescale** <time_unit> / <time_precision>

如: **`timescale** 1 ns / 100 ps

time_unit: 延时或时间的测量单位

time_precision: 延时值超出精度要先舍入后使用

- **`timescale** 必须在模块之前出现

```
`timescale 1 ns / 10 ps
```

```
// All time units are in multiples of 1 nanosecond
```

```
module MUX2_1 (out, a, b, sel);
```

```
output out;
```

```
input a, b, sel;
```

```
    not #1 not1( sel_, sel);
```

```
    and #2 and1( a1, a, sel_);
```

```
    and #2 and2( b1, b, sel);
```

```
    or #1 or1( out, a1, b1);
```

```
endmodule
```



时间尺度

- **time_precision**不能大于**time_unit**
- **time_precision**和**time_unit**的表示方法: **integer unit_string**
 - integer: 可以是1, 10, 100
 - unit_string: 可以是s(second), ms(millisecond), us(microsecond), ns(nanosecond), ps(picosecond), fs(femtosecond)
 - 以上integer和unit_string可任意组合
- **precision**的时间单位应尽量与设计的实际精度相同。
 - precision是仿真器的仿真时间步。
 - 若time_unit与precision_unit差别很大将严重影响仿真速度。
 - 如说明一个`timescale 1s / 1ps, 则仿真器在1秒内要扫描其事件序列 10^{12} 次; 而`timescale 1s/1ms则只需扫描 10^3 次。



时间尺度

- 所有**timescale**中的最小值决定仿真时的最小时间单位。因为仿真器必须对整个设计进行精确仿真

```
`timescale 1ns/ 10ps
module1 (...);
    not #1.23 (...) // 1.23ns
    ...
endmodule

`timescale 100ns/ 1ns
module2 (...);
    not #1.23 (...) // 123ns
    ...
endmodule

`timescale 1ps/ 100fs
module3 (...);
    not #1.23 (...) // 1.23ps
    ...
endmodule
```



VERILOG电路描述



电路描述手段

■ Verilog电路描述

- 连续赋值语句(assign)
- 过程(always)
 - 阻塞赋值(=)
 - 非阻塞赋值(<=)
 - if语句
 - case语句
- 元件例化

连续赋值语句 (assign)

- 实现组合逻辑电路

```
module test(out2,out1,in);  
  output out2,out1;  
  input in;  
  ...  
  wire a=b&c; //declare and assign  
  
  assign out2 = ~ in ;  
  assign out1 = sel? i1:i0;  
endmodule
```

可以是复杂的布尔函数

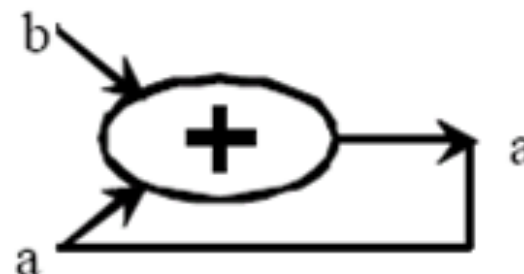
二选一MUX

sel为' 1'时选择i1; 否则i0

连续赋值语句 (assign)

- 避免出现反馈环路

```
assign a = b + a ;
```



元件例化

```
AND u1(a,b,and_out);
```

```
AND u1(.a(a), .b(b), .o(and_out));
```

- 格式:

- 模块名 <实例名> (<端口列表>);

- 端口列表有两种表示方式,

- 第一种方式显式给出端口与信号之间的对应关系:

- (.端口名(信号值表达式), .端口名(信号值表达式),)

- 第二种方法是隐式给出端口与信号之间的关系:

- (信号值表达式, 信号值表达式,)

这种方式下, 例化的端口列表中信号的顺序要与该模块定义中的端口列表中端口顺序严格一致。而第一种方法则无此要求。



过程

■ 过程 (**always**)

- 阻塞赋值(=)
- 非阻塞赋值(<=)
- **if**语句
- **case**语句

```
always @(posedge/negedge sig or...)  
begin  
    语句块  
end
```

```
always @(a or b or c or ...)  
begin  
    语句块  
end
```



if语句

```
if ( 表达式 )  
    语句  
else  
    语句
```

```
if ( 表达式 )  
    语句  
else if ( 表达式 )  
    语句  
else  
    语句
```

```
if ( a > b )  
    res = 1;  
else if ( a < c )  
    begin  
        res = 2;  
        out = 4;  
    end  
else  
    res = 3;
```



case语句

```
`define OP_LOAD 2'b00  
`define OP_ADD 2'b01  
`define OP_SUB 2'b10  
  
always @(op) begin  
  case ( op )  
    `OP_LOAD : out = din;  
    `OP_SUB   : out = a - b;  
    default   : out = a + b;  
  endcase  
end
```



阻塞赋值和非阻塞赋值

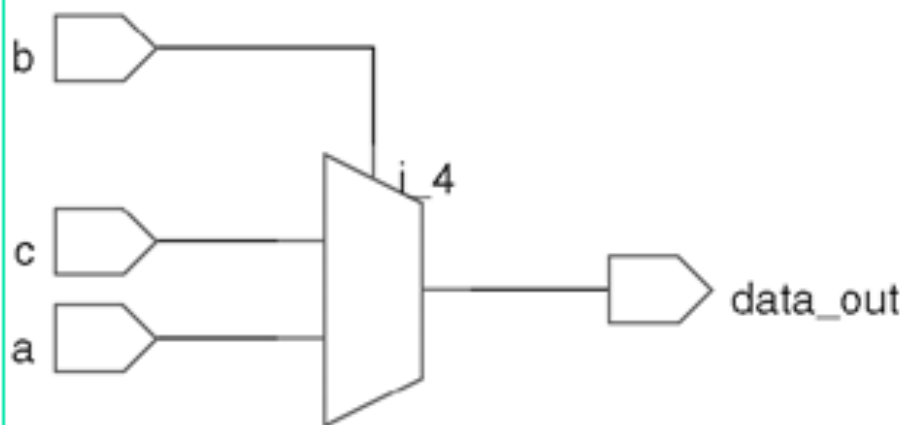
- 阻塞赋值(=)
 - 阻塞性过程赋值在其后所有语句执行前执行，即在下一语句执行前该赋值语句完成执行。
- 非阻塞性(<=)
 - 过程赋值中，对目标的赋值是非阻塞的(因为时延)，但可预定在将来某个时间步发生(根据时延；如果是0时延，那么在当前时间步结束)。
 - 当非阻塞性过程赋值被执行时，计算右端表达式，右端值被赋予左端目标，并继续执行下一条语句。
 - 预定的最早输出将在当前时间步结束时，这种情况发生在赋值语句中没有时延时。在当前时间步结束或任意输出被调度时，即对左端目标赋值。
- 在同一个**always/initial**块里不要混用两种赋值语句

阻塞赋值实现组合逻辑电路

- 在实现组合逻辑电路的**always**块里用阻塞赋值

```
module test(...);  
...  
reg dout;  
  
always @(a or b or c)  
  if (b)  
    dout = a ;  
  else  
    dout = c ;  
  
endmodule
```

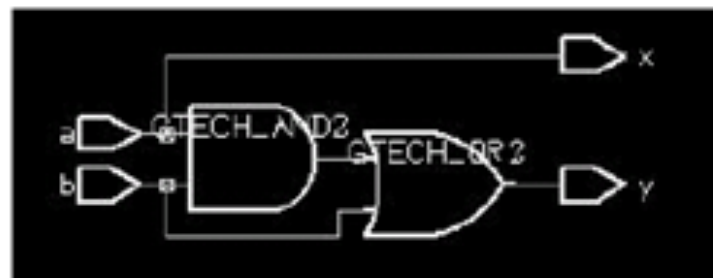
always块的输出



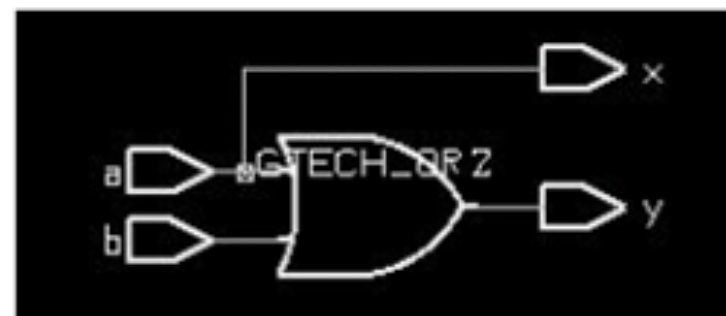
阻塞赋值实现组合逻辑电路

- 在实现组合逻辑电路的**always**块里用阻塞赋值

```
always @(a or b or x) begin
    x = a & b ;
    y = x | b ;
    x = a ;
end
```



```
always @(a or b or x) begin
    x <= a & b ;
    y <= x | b ;
    x <= a ;
end
```

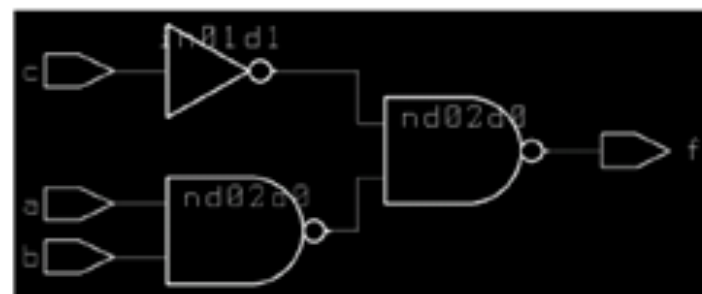


always块的敏感表

- 实现组合逻辑电路的**always**块敏感表必须写全，否则仿真结果和综合结果会不一致

```
always @(a or b or c)  
f = a&b | c;
```

```
always @(a or b)  
f = a&b | c;
```

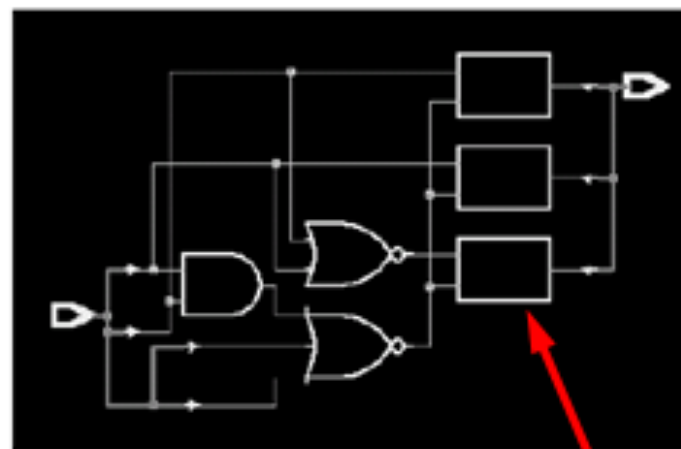


避免Latch的产生

- 实现组合逻辑电路的**always**块中**if**和**case**语句的分支必须写全。

```
always @(bcd) begin
  case ( bcd )
    4'd0 : out = 3'b001;
    4'd1 : out = 3'b010;
    4'd2 : out = 3'b011;
  endcase
end
```

```
always @(a or b or gate )
begin
  if (gate)
    out = a | b;
end
```

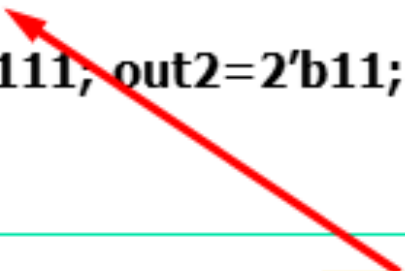


Latch



避免Latch的产生

```
always @(bcd) begin
  case ( bcd )
    2'b00 : begin out1 = 3'b001; out2=2'b00; end
    2'b01 : begin out1 = 3'b011; out2=2'b10; end
    2'b10 : out1 = 3'b101;
    2'b11 : begin out1 = 3'b111; out2=2'b11; end
  endcase
end
```

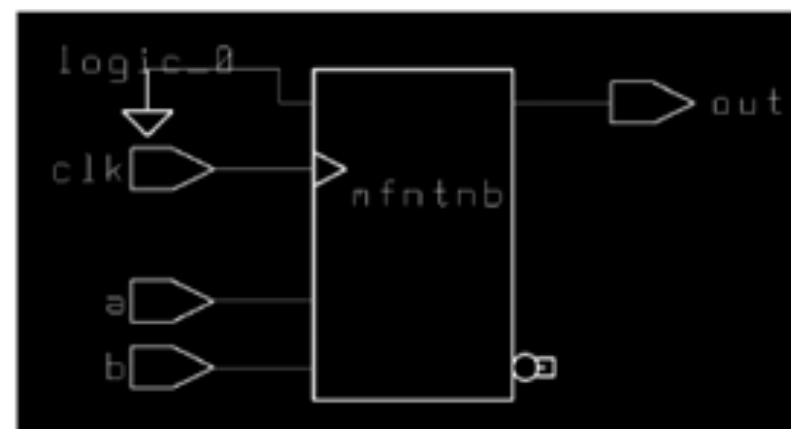


没有改变out2
导致产生Latch

触发器

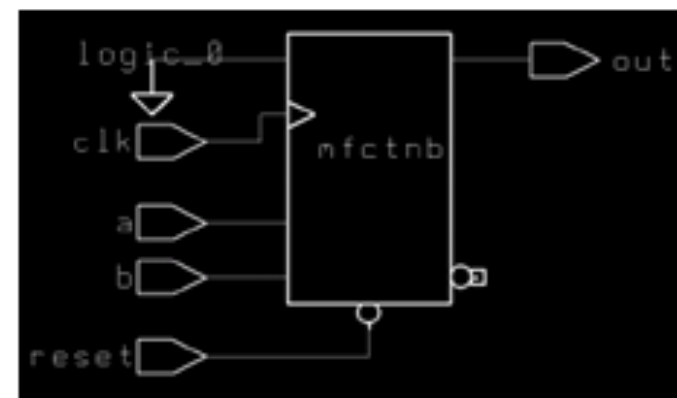
- **always**块的`@(posedge clk)`或`@(negedge clk)`
- 赋值语句的输出信号

```
always @(posedge clk) begin  
    out <= a&b;  
end
```



带低电平有效异步复位端的触发器

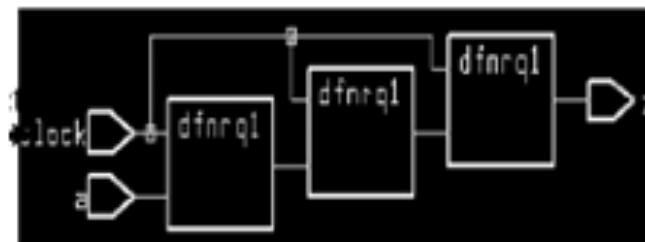
```
always @(posedge clk or negedge  
reset) begin  
    if ( !reset )  
        out <= 0;  
    else  
        out <= a&b ;  
end
```



都必须是边沿

非阻塞赋值（<=）实现时序电路

```
always @(posedge clk)
begin
  x <= a;
  y <= x;
  z <= y;
end
```



```
always @(posedge clk)
begin
  x = a;
  y = x;
  z = y;
end
```

