Verilog HDL语言初步

主讲 杨全胜

东南大学计算机科学与工程学院

第一部分 Verilog HDL语言

§1 综述

一. 什么是硬件描述语言?

硬件描述语言是一种用文本形式来描述和设计电路的语言。是硬件设计人员和电子设计自动化（EDA）工具之间的界面。

功能：

1）编写设计文件；

2）建立电子系统行为级的仿真模型；

3）自动综合以生成符合要求且在电路结构上可以实现的数字逻辑网表（Netlist）；

4）写入到CPLD和FPGA器件中。

二. 为什么要用HDL?

1、电路设计的规模越来越大，复杂度越来越高。

2、电子领域的竞争越来越激烈，开发周期要短。

3、调试电路速度快。不必修改电路原理图原型，只需要对HDL进行修改。

4、易于理解，易于维护。

5、有许多易于掌握的仿真、综合和布局布线工具。

三、Bottom Up和 Top down的设计方法

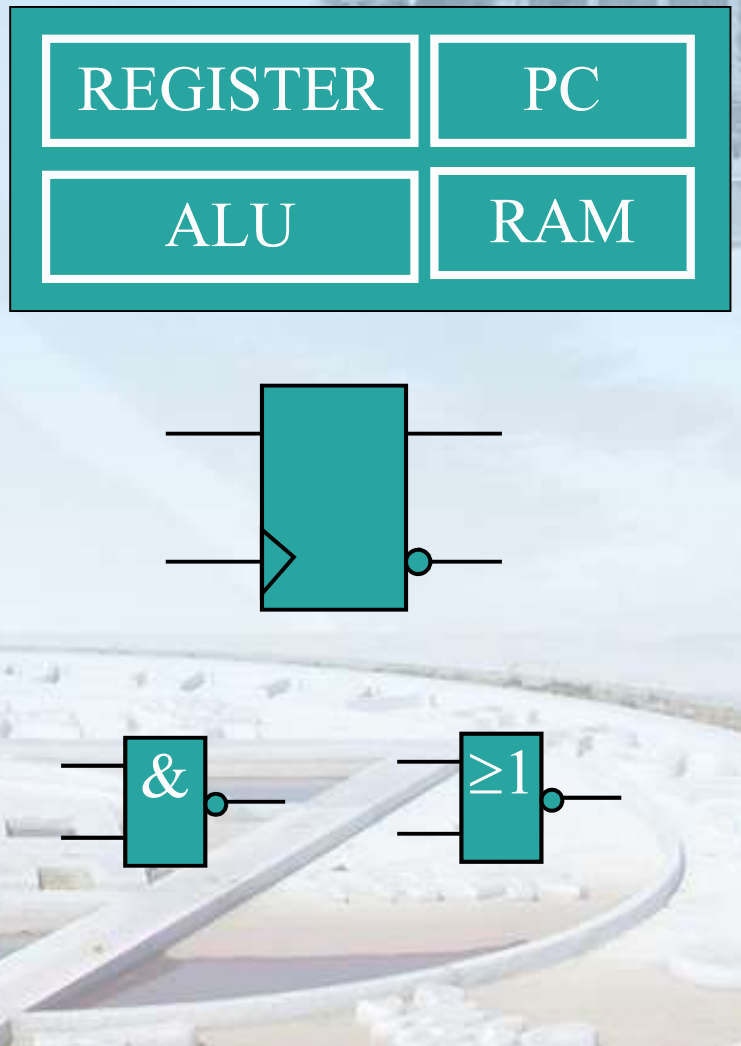
1. Bottom Up的设计方法

4.完成整个系统测试与性能分析

3.由各个功能模块连成一个完整系统

2.由逻辑单元组成各个独立的功能模块

1.由基本门构成各个组合与时序逻辑



★传统的电路系统设计方法的步骤:

1.采用自下而上的设计方法-从状态图的简化，写出最简逻辑表达式;

2. 采用通用逻辑元器件-通常采用74系列和CMOS4000系列的产品进行设计;

3.在系统硬件设计的后期进行调试和仿真；

只有在部分或全部硬件电路连接完毕，才可以进行电路调试，一旦考虑不周到，系统设计存在较大缺陷，则要重新设计，使设计周期延长。

4.设计结果是一张电路图；

当设计调试完毕后，形成电原理图，该图包括元器件型号和信号之间的互连关系等等

优点：

1.设计人员对于用这种方法进行设计比较熟悉；

2.实现各个子块电路所需的时间短。

缺点：

1.一般来讲，对系统的整体功能把握不足；

2.实现整个系统的功能所需的时间长，因为必须先将各个小模块完成，使用这种方法对设计人员之间相互进行协作有比较高的要求。

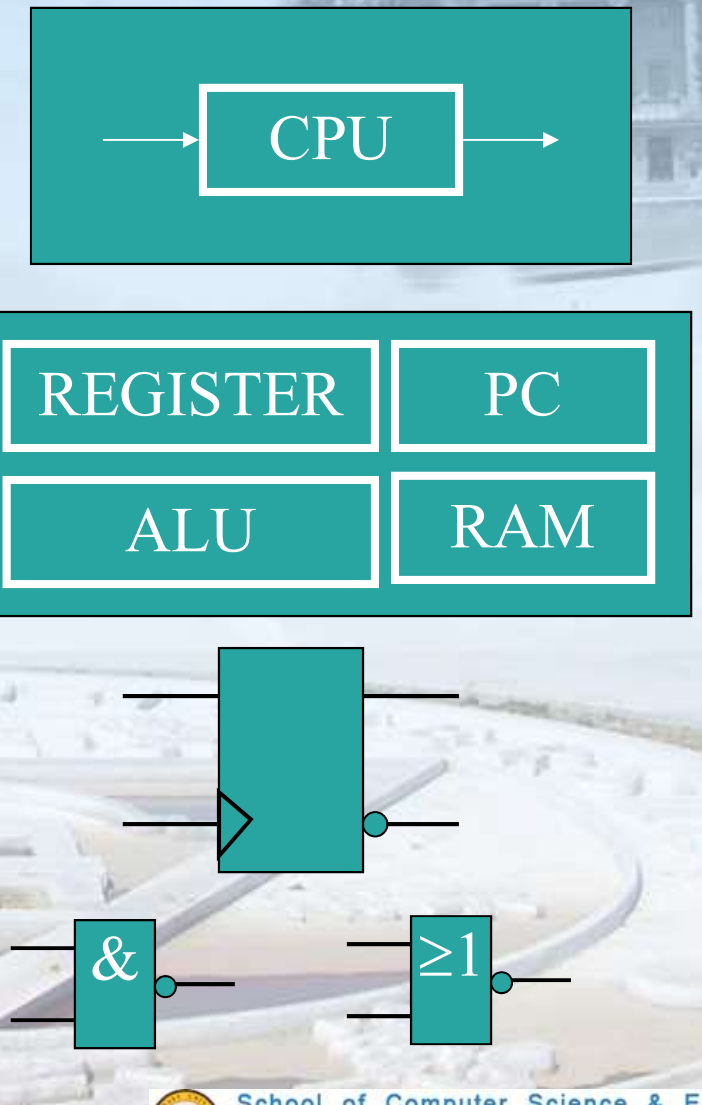
2. Top down 的设计方法

1. 系统层：顶层模块，行为级描述，功能模拟和性能评估

2.各个功能模块划分，设计和验证

3.各个功能模块系统级联合验证

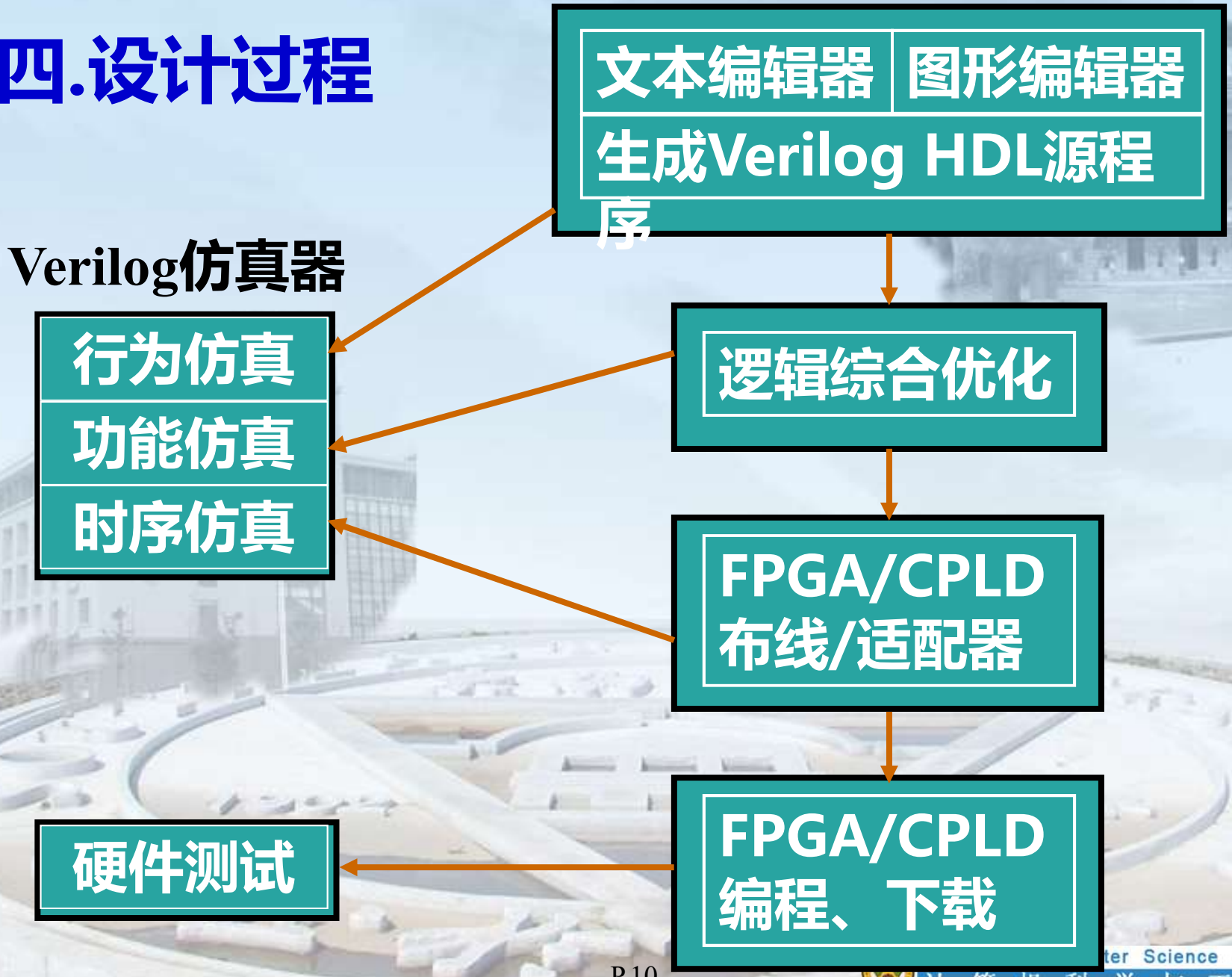
4.工艺库映射



Top down 的设计方法的特点：

- 从系统层开始设计和优化，保证了设计结果的正确性
- 适合复杂的、大规模电路的设计
- 缩短设计周期
- 依赖于先进的EDA设计工具和环境，费用昂贵
- 需要精确的工艺库支持

四.设计过程


文本编辑器
图形编辑器
生成Verilog HDL源程序
逻辑综合优化
FPGA/CPLD
布线/适配器
FPGA/CPLD
编程、下载
硬件测试
Verilog仿真器
行为仿真
功能仿真
时序仿真

§2 Verilog HDL设计初步

内容简介

- Verilog HDL与C语言的比较
- Verilog模块的基本结构
- 逻辑功能的定义

一. Verilog HDL与C语言的比较

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

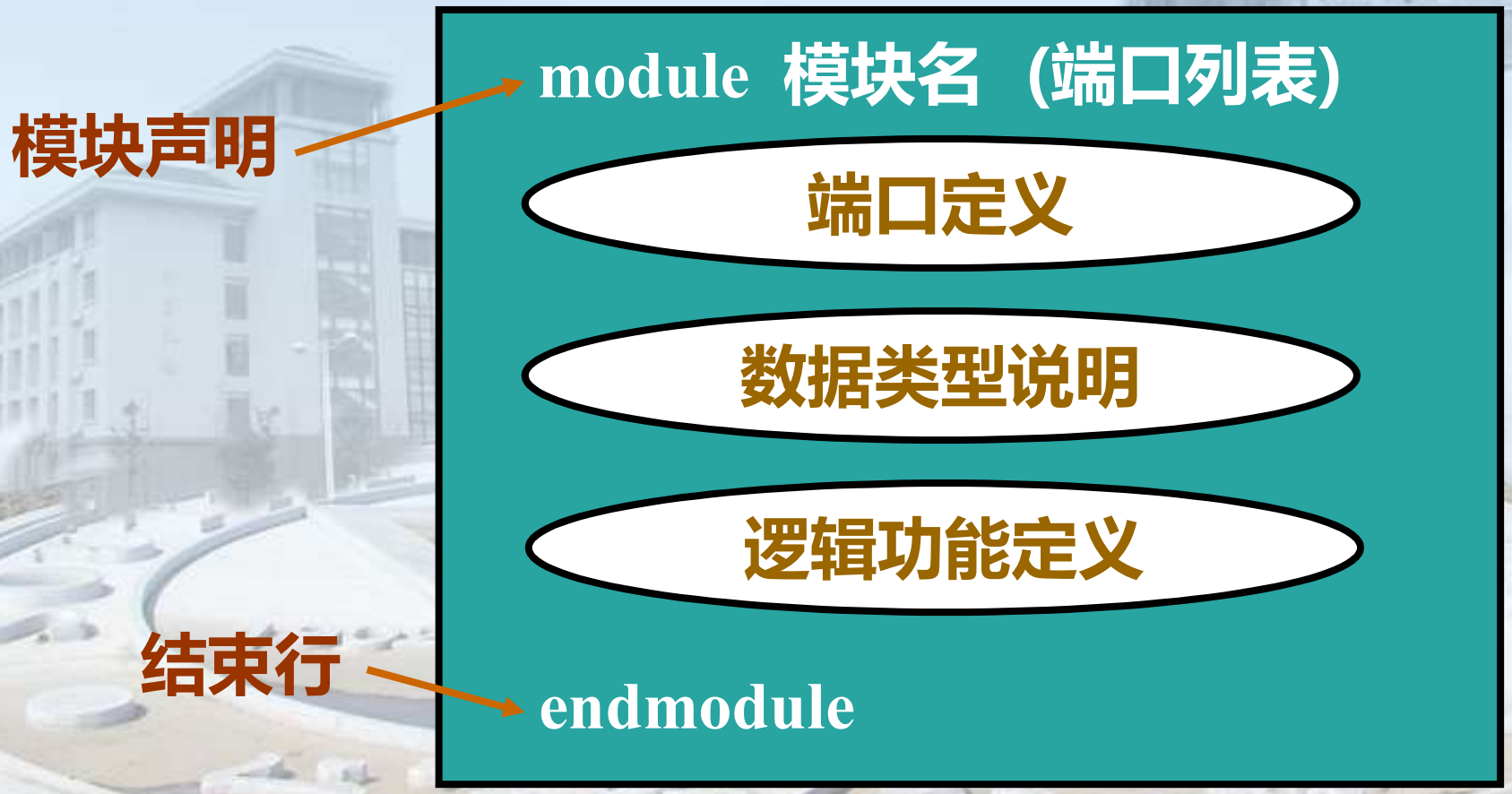
C语言	Verilog语言
function	module, function
if-then-else	if-then-else
for	for
while	while
case	case
break	break
define	define
printf	printf
int	int

Verilog HDL与C语言运算符的比较

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

二. Verilog模块的基本结构

由关键词module和endmodule定义



1.模块声明

module——关键词

模块名—— 模块唯一的标识符

端口列表——是由输入、输出和双向端口的端口表达式按一定的次序组成的一个列表，它用来指明模块所具有的端口，这些端口用来与其它模块进行连接。

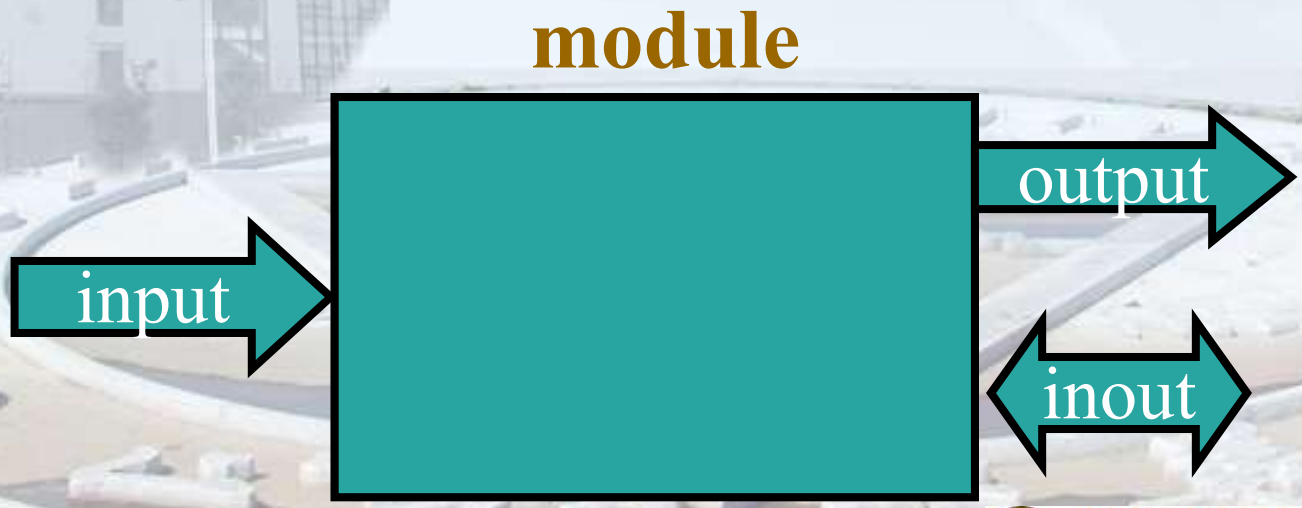
2. 端口定义

又称“端口声明语句”，用来进行端口方向的说明。 Verilog语言中有如下三种端口声明语句：

1）input——对应的端口是输入端口

2）output——对应的端口是输出端口

3）inout——对应的端口是双向端口



3. 数据类型说明

用来指定模块内用到的数据对象的类型。

wire——连线型

wire A，B，C，D；//定义信号A~D为wire型

reg——寄存器型

reg [3:0] out；//定义信号out的数据类型为4位reg型

缺省数据类型为wire型

4. 逻辑功能定义

模块中最核心部分，有三种方法可在模块中产生逻辑。

1）用“assign”持续赋值语句定义

例： assign a = b & c;

2）调用元件（元件例化）

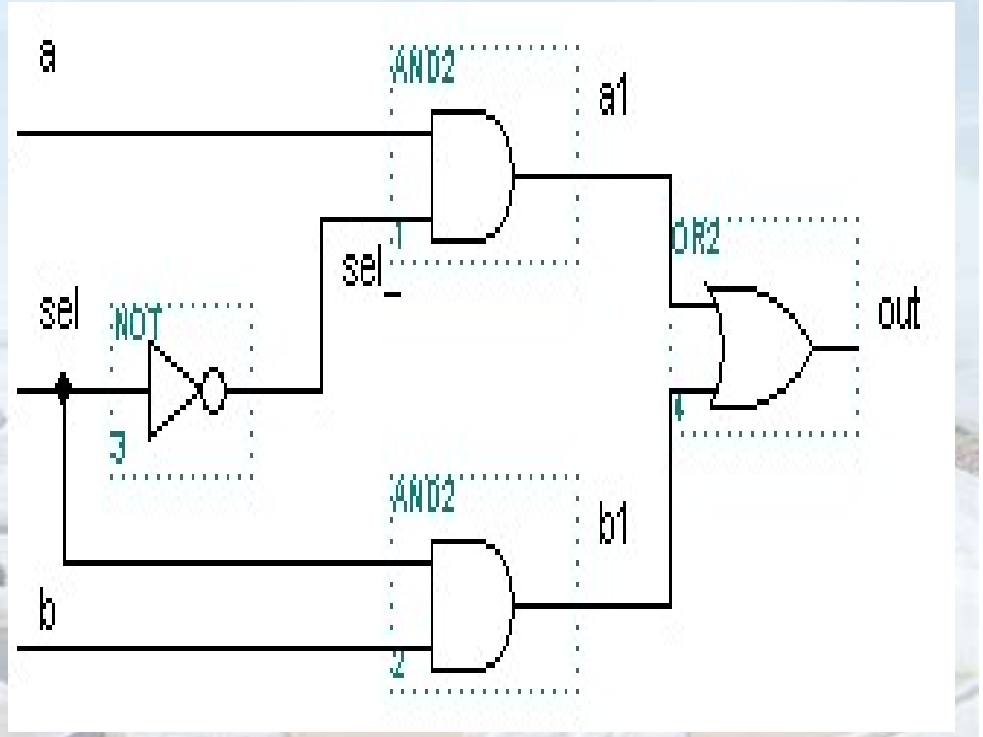
类似于在电路图输入方式下调入图形符号完成设计。

元件例化的格式为：

门元件名 <实例名> （<端口列表>);

例：调用模块的例子

```
module  MUX2-1 (out，  a，  b，  sel）；
        output  out;
        input  a，  b，  sel;
        not (sel_，  sel);
        and (a1，  a，  sel _);
        and ( b1，  b，  sel);
        or (out，  a1，  b1);
endmodule
```



设计师自己设计的各种模块也可以看作元件，被顶层文件或其他文件调用：

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。

举例：由1位全加器组成的4位全加器

```
module full_add (a,b,cin,sum,cout);
  input   a,b,cin;
  output  sum,cout;
  assign {cout,sum} = a+b+cin;
endmodule
module add4(sum,cout,a,b,cin);
  output [3:0] sum;
  output cout;
  input [3:0] a,b;
  input cin;
```

```
wire cin1,cin2,cin3;
full_add f0 (a[0],b[0],cin,sum[0],cin1);
full_add f1 (a[1],b[1],cin1,sum[1],cin2);
full_add f2 (.a(a[2]),.b(b[2]),.cin(cin2),
             .sum(sum[2]),.cout(cin3));
full_add f3 (.cin(cin3),.a(a[3]),.b(b[3]),
             .cout(cout),.sum(sum[3]));
endmodule
```

3）用“always”过程块赋值

例：

```
always @ (posedge  clk)
    begin
        if(reset)  out=0;
        else  out=out+1;
    end
```

过程块、持续赋值语句与实例应用要点总结：

1. 在Verilog模块中，所有的过程块（如initial、always）、持续赋值语句、实例引用之间都是并行的；
2. 它们表示的是一种通过变量名互相连接的关系；
3. 在同一模块中这三者出现的先后顺序没有关系；
4. 只有持续赋值语句assign和实例引用语句可以独立于过程块而存在于模块的功能定义部分。

§3 Verilog HDL语言要素

内容简介

- 词法
- 数据类型
- 寄存器和存储器
- 运算符

一. 词法

1. 空白符（间隔符）

包括：空格(\b)、tab(\t)(制表符)、换行符(\n)及换页符。

空白符使代码错落有致、阅读方便。综合时，空白符被忽略。但是在字符串中空白和制表符会被认为是有意义的字符。

Verilog程序可以不分行：

```
initial  begin  ina=3'b001;  inb=3'b011; end
```

也可以加入空白符采用多行编写：

```
initial
   begin
      ina=3'b001;
      inb=3'b011;
   end
```

2. 注释

有两种注释形式：

- 单行注释：以//开始到本行结束。
- 多行注释：以/*开始到*/结束。

```
/*举例说明*/
module  addbit(a,b,ci,sum,co);
//输入端口
input  a;
input  b;.....
```

3. 数字与字符串

Verilog HDL有下面4种基本逻辑状态：

0——低电平、逻辑0或“假”

1——高电平、逻辑1或“真”

X——未知状态

Z——高阻态

X、Z不分大小写

1）整数

有4种进制表示形式：

- 二进制整数（b或B）
- 十进制整数（d或D）
- 十六进制整数（h或H）
- 八进制整数（o或O）

常数按照其数值类型可以划分为整数和实数两种

数字表达方式有以下3种：

- <对应的二进制数的位宽'> <进制> <数字>
- <进制> <数字>
- <数字>

举例：

8'b11000101 //位宽为8位的二进制数

8'hd5 //位宽为8位的十六进制数d5H

5'o27 //位宽为5位的八进制数27O

```
4'B1X_01        //4位二进制数1X01
5'HX            //5位十六进制数XX
4'hz            //4位十六进制数z
8'h  2  A       //位宽与字符间允许有空格
-8‘D5           //8位二进制数，-5的补码
```

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。

数值常量中的下划线“_”是为了增加可读性，可以忽略。如8'b1100_0110表示8位二进制数。

数值常量中的“？”表示高阻状态。

例：2'B1？表示2位的二进制数其中的一位是高阻状态。

如果没有定义一个整数型的长度，数的长度为相应值中定义的位数。下面是两个例子：

'o721　　//9位2进制位宽的八进制数

'hAF　　//8位2进制位宽的十六进制数

如果定义的长度比为常量指定的长度长，通常在左边填0补位。但是如果数最左边一位为x或z，就相应地用x或z在左边补位。例如：

10'b10 左边添0占位, 0000000010

10'bx0x1 左边添x占位,xxxxxxx0x1

如果定义的位宽比实际的位数小，那么最左边的位相应地被截断：

```
3'b1001_0011    //与3'b011相等
5'h0FFF         //与5'h1F相等
```

2）实数

有两种表示方法：

- 十进制表示方法

  2.0

  5.67

  2.　　//非法：小数点两侧必须有数字

- 科学计数法

  43_5.1e2　　//43510.0（下划线忽略）

  9.6E2　　//960.0

  5E-4　　//0.0005

下面的几个例子是无效的格式：

.25

3.

7.E3

.8e-2

实数可以转化为整数，根据四舍五入的原则，而不是截断原则，当将实数赋给一个整数时，这种转化会自行发生，例如：在转化成整数时，实数25.5和25.8都变成26，而25.2则变成25。

3）字符串

字符串是双引号内的字符序列，不能分成多行书写。若字符串用做Verilog HDL表达式或赋值语句中的操作数，则字符串被看作8位似的ASCII值序列，每一个字符对应8位ASCII值。

例1：字符串变量声明

```
reg  [8*12:1]   stringvar;
initial
    begin
        stringvar="hello  world!";
    end
```

转意符：

特殊字符表示	意义
\n	换行符
\t	Tab键
\\	符号\
*	符号*
\ddd	3位八进制表示的ASCII值
%%	符号%

4. 标识符

Verilog HDL中的标识符可以是任意一组字母、数字以及符号“$”和“_”（下划线）的组合，但是标识符的第一个字符必须是字母或下划线。标识符是区分大小写的。

合法标识符：

count　　_A1_d2

COUNT　　R56_68

非法标识符：

30 count　　//标识符不允许以数字开头

out *　　//标识符中不允许包含*

5. 关键字

Verilog HDL内部已经使用的词称为**关键字或保留字**。这些关键字用户不能随便使用。在编写程序时，变量的定义不要与这些关键词冲突。

所有的关键字都是小写

二. 数据类型

Verilog HDL中共有19种数据类型。数据类型是用来表示数字电路硬件中的数据储存和传送元件的。这里主要介绍4种最基本的数据类型。

1. 连线型（Net Type）

net type 相当于硬件电路中的各种物理连线。

特点：输出的值紧跟输入值的变化而变化。

Net Type的变量不能存储值，而且必须受到驱动器的驱动。

两种驱动方式：

1）在结构描述中将它连接到一个逻辑门或模块的输出端。

2）用持续赋值语句assign对其进行赋值。

当没有驱动源对其驱动时，它将保持高阻态。

为了能够精确地反映硬件电路中各种可能的物理信号连接特性， Verilog HDL提供了多种连线型数据。常用的有wire型和tri型。这两种变量都用于连接器件单元，它们具有相同的语法格式和功能。

wire型变量：通常用来表示单个门驱动或连续赋值语句驱动的连线型数据。

tri型变量：通常用来表示多驱动器驱动的连线型数据。

wire型变量的格式：

wire [n-1:0] 数据名1，数据名2，……，数据名n；

wire——wire型数据确认符；

[n-1:0]——代表该数据的位宽。缺省状态，位宽默认值为1。这里的位是二进制的位。

数据名——若一次定义多个数据，数据名之间用逗号隔开。

声明语句的最后用分号表示语句的结束。

例1. 定义数据总线宽8位，地址总线宽20位。

```
wire[7:0]  databus;          // databus宽8位
wire[19:0]  addrbus;         // addrbus宽20位
```

或:

```
wire[8:1]  databus;
wire[20:1]  addrbus;
wire  a;              //定义了一个1位的wire型数据
```

例2. 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
```

例3. 可只使用多位数据中的几位，但要注意位宽。

```
wire[7:0]  out;
wire[3:0]  in;
assign  out[5:2]=in;
```

说明：

1） wire 型变量常用来表示以assign语句赋值的**组合逻辑**信号。

2）输入/输出信号缺省时自动定义为wire 型。

3）对综合器而言， wire 型信号的每一位可以取0，1，X或Z中的任意值。

2. 寄存器型（Register Type）

寄存器是数据存储单元的抽象。寄存器型数据对应的是具有状态保持作用的硬件电路，如触发器、锁存器等。

寄存器型数据和连线型数据的区别：

寄存器型数据保持最后一次的赋值。而连线型数据需有持续的驱动。

reg ——常用的寄存器型变量

reg型数据的格式：

reg [n-1:0] 数据名1，数据名2，…数据名n；

例1.

```
reg   a, b;  //定义了两个reg型变量
reg [7:0]   qout;        //定义qout为8位宽的reg
                              型变量
```

说明：

1） reg型数据常用来表示“always”模块内的指定信号，常代表触发器。在“always”模块内被赋值的每一个信号都必须定义成reg型。

2）对于reg型数据，其赋值语句的作用就如同改变一组触发器的存储单元的值。

3）若reg型数据未初始化（即缺省），则初始值为不定状态。

3. 参数型（parameter）

在Verilog HDL中，用parameter来定义常量，即用它来定义变量的位宽及延时等。

格式：

parameter 参数名1=表达式1，参数名2=表达式2，…；

parameter常用来定义延迟时间和变量宽度。

例：

```
parameter  e=2，f=9；//定义两个常数参数
parameter  r=5.7；    //定义r为一个实型参数
parameter  a_delay=（r+f）/2；
                          //用常数表达式赋值
```

三. 寄存器和存储器

用reg类型变量可构成寄存器和存储器

1. 寄存器

reg [7:0] mybyte;

A= mybyte[6]； //将mybyte的第6位赋值给A

B= mybyte[5:2]； //将mybyte的第5，4，3，2位赋值给B

例：

```
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
```

寄存器可以取任意长度。寄存器中的值通常被解释为无符号数。

2. 存储器

Verilog HDL通过对reg型变量建立数组来对存储器建模，可以描述RAM型存储器、ROM存储器和reg 文件。数组中的每一个单元通过一个数组索引进行寻址。在Verilog语言中没有多维数组存在，memory型数据是通过扩展reg型数据的地址范围来生成的。

格式：

reg [n-1:0] 存储器名[m-1:0];

或

reg [n-1:0] 存储器名[m:1];

reg [n-1:0]：定义了存储器中每一个存储单元的大小。

[m-1:0]：定义了该存储器中有多少个这样的单元。

例1. 定义一个存储器，1024个字节，每个字节8位。

```
reg [7:0]  mymem[1023:0];
```

例2. 存储器与寄存器的区别

```
reg [n-1:0]  rega;      //一个n位的寄存器
reg  mema[n-1:0];    //n个一位寄存器组成的存储器组
reg [3:0]   Amem[63:0];
```

说明：

1）数组的维数不能大于2。

2）存储器属于寄存器数组类型。连线数据类型没有相应的存储器类型。

3）单个寄存器说明既能够用于说明寄存器类型，也可以用于说明存储器类型。

例：

parameter ADDR_SIZE = 16 , WORD_SIZE = 8;

reg [WORD_SIZE:1] RamPar [ADDR_SIZE-1 : 0]，DataReg；

RamPar——存储器，是16个8位寄存器数组；

DataReg——8位寄存器。

4）在赋值语句中需要注意如下区别：**存储器赋值不能在一条赋值语句中完成，但是寄存器可以。**因此在存储器被赋值时，需要定义一个索引。下例说明它们之间的不同。

```
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
```

有一种存储器赋值的方法是分别对存储器中的每个字赋值。例如：

```
reg [3:0]   Xrom [4:1];
Xrom[1] = 4'hA;
Xrom[2] = 4'h8;
Xrom[3] = 4'hF;
Xrom[4] = 4'h2;
```

四. 运算符

Verilog语言参考了C语言中大多数运算符的语义和句法。但Verilog中没有增1(i++)和减1 (i– –)运算符。

1. 算术运算符

- + （一元加和二元加）
- - （一元减和二元减）
- * （乘）
- / （除）
- % （取模）

说明:

1）两个整数相除，结果值要略去小数部分，只取整数部分；

2）取模运算时，结果的符号位采用模运算式里第一个操作数的符号位；

模运算表达式	结果	说明
10 % 4	2	余数为2
12 % 3	0	整数
-11 % 5	-1	余数为-1

3）在进行算术运算操作时，如果某个操作数有不确定的值X或Z，那么整个结果为X。

例：

'b10x1 + 'b01111 结果为不确定数'bxxxxx

4）无符号数和有符号数

- 若操作数为寄存器型或连线型，或基数格式表示形式的整数则为无符号数；
- 若为整型或实型，则可以是有符号数。

例：

```
reg [5:0] Bar；
integer Tab;
. . .
Bar = -6'd12；  //寄存器变量Bar的十进制数为52，向量值为110100。
Tab = -6'd12；  //整数Tab的十进制数为-12，位形式为110100。
```

5）算术操作结果的长度

算术表达式结果的长度由最长的操作数决定。在赋值语句下，算术操作结果的长度由操作符左端目标长度决定。

```
reg [3:0] Arc, Bar, Crt;
reg [5:0] Frx;
. . .
Arc = Bar + Crt;
Frx = Bar + Crt;
```

例：算术运算符应用的一个例子。

```
module arithmetic (a，b，out1，out2，out3，
                                    out4，out5)

    input [2:0] a，b；

    output [3:0] out1；

    output [4:0] out3；

    output [2:0] out2，  out4，out5；

    reg [3:0] out1；

    reg [4:0] out3；

    reg [2:0] out2，  out4，out5；
```

```
always @ (a or b)
    begin
        out1=a+b ;
        out2=a-b ;
        out3=a*b ;
        out4=a/b ;
        out5=a%b ;
    end
endmodule
```

2. 逻辑运算符

逻辑运算符有3种:

- && (逻辑与)
- || (逻辑或)
- ！ (逻辑非)

说明:

1） && 和|| 为二目运算符，要求有两个操作数。

例 (a>b) && (b>c) ， a && b

(a<b) || (b<c) ， a || b

2）！是单目运算符，只要求一个操作数。

例：！(a>b) ， !a

3）在一个逻辑表达式中，如果包含多个逻辑运算符，

如：！a&&b||(x>y)&&c

按以下优先次序：

- ！→ && →||
- 逻辑运算符中，“&& ”和“||”的优先级别低于关系运算符，“！”高于算术运算符。

3. 位运算

在Verilog语言中有7种位逻辑运算符：

- ~ 按位取反；
- | 按位或；
- & 按位与；
- ^ 按位异或；
- ^~或~^ 按位异或非；
- ~& 按位与非；
- ~| 按位或非；

例：若A=5'b11001；B=5'b10101，则：

~A=5'b00110

A&B= 5'b10001

A|B= 5'b11101

A^B= 5'b01100

说明：

1）按位运算符中，除了“~”为单目运算符外，其余均为双目运算符。

2）对于双目运算符，如果操作数长度不相等，长度较小的操作数在最左侧添0补位。

3）无论单目按位运算符还是双目按位运算符，经过按位运算后，原来的操作数有几位，所得结果仍为几位。

4）不要将逻辑运算符和按位运算符相混淆。

4. 关系运算符

Verilog关系运算符有：

- >（大于）
- <（小于）
- >=（大于等于）
- <=（小于等于）

例：关系运算符应用的一个例子。

```
module relation (a，b，out1，out2，out3，out4)
   input [2:0] a , b;
   output out1 , out2，out3，out4 ;
   reg out1，out2，out3，out4 ;
   always @ (a or b)
       begin
          out1=a<b ;
          out2=a<=b ;
          out3=a>b ;
          if (a>=b)
            out4=1
          else
            out4=0
      end
endmodule
```

说明：

1）在进行关系运算时，若声明的关系为“假”，则返回值是“0”；若声明的关系为“真”，则返回值是“1”；

2）若某个操作数的值不定，则关系是模糊的，返回值是不定值。

3）所有关系运算符有着相同的优先级别。关系运算符的优先级别低于算术运算符。

5. 等式运算符

等式运算符有4种

- ==（等于）
- !=（不等于）
- ===（全等）
- !==（非全等）

双目运算符,要求有两个操作数，得到的结果是1位的逻辑值。

- 声明的关系为真，结果为1；
- 声明的关系为假，结果为0；

“==”与“===”的区别：

相等运算符真值表

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

全等运算符真值表

===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

6. 缩位运算符（归约运算符）

单目运算符，也有与、或、非运算。包括下面几种：

- &——与
- ~&——与非
- |——或
- ~|——或非
- ^——异或
- ^~，~^——同或

其与、或、非运算规则类似于位运算符的运算规则，但其运算过程不同。

位运算：

对操作数的相应位进行与、或、非运算，操作数是几位数，则运算结果是几位。

缩位运算：

对单个操作数进行与、或、非递推运算，最后的运算结果是1位的二进制数。

具体运算过程：

第一步：先将操作数的第1位与第2位进行与、或、非运算；

第二步：将运算结果与第3位进行与、或、非运算，依次类推，直至最后一位。

```
例：reg[3:0]  a;
    reg  b;
    b=&a;
```

```
若：A=5'b11001
则：&A=0;
    |A=1;
```

7. 移位运算符

- >> ——右移
- << ——左移

使用方法：

a>>n　　或　　a<<n

a——代表要进行移位的操作数；

n ——代表要移几位

这两种移位运算都用0来填补移出的空位

```
module  shift;
    reg[3:0]  start, result;
        initial
        bigin
            start=1;
            result= (start<<2);
        end
endmodule
```

8. 条件运算符

?：——条件运算符，有三个操作数，与C语言相同。

格式：

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。

```
module  add_or_sub(a,b,op,result);
    parameter  ADD=1'b0;
    input [7:0]  a,b;
    input  op;
    output [7:0]  result;
    assign  result=(op== ADD)?a+b:a-b;
endmodule
```

9. 位拼接运算——{ }

这是一个特殊的运算符，这一运算符可以将两个或更多个信号的某些位并接起来进行运算操作。其使用方法是把某些信号的某些位详细地列出来，中间用逗号分开，最后用大括号括起来表示一个整体信号。

格式：

{信号1的某几位，信号2的某几位，…，信号n的某几位}

例：wire [7:0] Dbus;

wire [11:0] Abus;

assign Dbus [7:4] = {Dbus [0], Dbus [1], Dbus[2], Dbus[3]};//以反转的顺序将低端4位赋给高端4位。

assign Dbus = {Dbus [3:0], Dbus [7:4]};

//高4位与低4位交换。

由于非定长常数的长度未知，不允许连接非定长常数。

例如，下列式子非法：

{Dbus，5} //不允许连接操作非定长常数。

运算符优先级排序：

高优先级别

```
!  ~
*   /   %
+   -
<<   >>
<   <=   >   >=
==   !=   ===   !==
&
^   ^~
|
&&
||
?:
```

低优先级别

§4 Verilog HDL行为语句

内容简介

- 过程语句
- 块语句
- 赋值语句
- 条件语句
- 循环语句
- 编译向导语句

Verilog HDL是由模块组成的

一. 过程语句

- 过程块
  - 过程语句
    - initial
    - always
  - 语句块
    - 过程性赋值语句
      - 过程赋值语句
      - 过程持续赋值语句
    - 高级程序语句
      - 条件分支语句
      - 循环控制语句

1. always过程语句

可选项

格式:

always @（敏感信号表达式）

begin

//过程赋值

//if-else，case，casex，casez选择语句

//while，repeat，for循环

//task，function调用

end

过程块

1）敏感信号——只要表达式中某个信号发生变化，就会引发块内语句的执行。

```
@ (a)                          //当信号a的值发生变化时
@ (a  or  b)                   //当信号a或b的值发生变化时
@ (posedge  clock)             //当clock上升沿到来时
@ (negedge  clock)             //当clock下降沿到来时
@ (posedge  clk  or  negedge  reset)
//当clk的上升沿或reset的下降沿到来时
```

```
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)
  case (sel)
    2'b00：out=in0；
    2'b01：out=in1；
```

```
        2'b10： out=in2 ；
        2'b11： out=in3 ；
        default ： out=2'bx ；
    endcase
endmodule
```

2） posedge 与 negedge 关键字

例1：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；  //同步清零，低有效
        else  if (load)     out=data；  //同步预置
        else                    out=out+1；  //计数
        end
endmodule
```

例2：时钟信号为clk，clear为异步清零信号

```
always @ (posedge  clk  or  posedge  clear)
always @ (posedge  clk  or  negedge  clear)
```

错误的描述:

```
always @ (posedge  clk  or  negedge  clear)
    begin
       if  (clear)      //应改为if  (!clear)
             out=0;
      else    out=in;
    end
```

3）用always过程块实现组合逻辑功能

- 敏感信号表达式内不能包含posedge 与 negedge 关键字
- 组合逻辑的所有输入信号都要作为“信号名”出现在敏感信号表达式中。

例:有什么问题?

```
module three_and(f, a, b, c);
output f;
input a, b, c ;
reg f;
always @ (a or b)
    begin                    //应改为@ (a or b or c)
        f=a&b&c;
    end
endmodule
```

4）用always过程块实现时序逻辑功能

- 敏感信号表达式内可以有posedge 与 negedge 关键字，也可以只有信号名；
- 不要求所有输入信号都出现在敏感信号列表的“信号名”中。

例：时钟下降沿触发的D触发器

```
module D_FF(Q，D，CLK);
output Q;
input D，CLK;
reg Q;
always @(negedge CLK)
   begin
      Q = D;
   end
endmodule
```

说明：

1）always过程语句后面可以是一个敏感事件列表,该敏感事件列表的作用是用来激活always过程语句的执行；

2）如果always过程块中的敏感事件列表缺省，则认为触发条件始终被满足， always过程块将无条件地循环执行下去，直到遇到$finish或$stop系统任务为止；

3）进行仿真时， always过程块是从模拟0开始执行的，且always语句在仿真过程中是不断重复执行的；

4）敏感事件列表由一个或多个“事件表达式”构成，事件表达式说明了启动块内语句执行时的触发条件，**当存在多个事件表达式时要用关键词or将多个触发条件组合起来**。

Verilog规定：只要这些事件表达式所代表的多个触发条件中有一个成立，就启动块内语句的执行。

5）**切勿将变量引入敏感信号列表**。

6）always过程块和initial过程块都不能嵌套使用。

例：不恰当使用always语句而产生仿真死锁的情况。

```
always
        begin
            clk=~clk;
        end
```

当敏感信号列表缺省时，语句块将一直执行下去，这就可能在仿真时产生仿真死锁情况

```
always
        begin
            #50 clk=~clk;
        end
```

加上时延控制“#50”产生一个周期为100的方波信号

2. initial过程块

格式：

```
initial
    begin
    语句1；
    语句2；
      ⋮
    语句n；
    end
```

过程块

说明：

1） initial语句后面没有“敏感信号列表”；

2） initial过程块中的语句是从模拟0开始执行，它在仿真过程中只执行一次，在执行完后，该initial过程块就被挂起，不再执行；

3） initial过程块的使用主要是面向功能模拟的，通常不具综合性。

例1：用initial过程语句对测试变量A、B、C赋值。

```
`timescale   1ns/100ps
module   test;
reg  A，B，C；
initial
   begin
          A=0；B=1；C=0；
    #50   A=1；B=0；
```

```
    #50   A= 0;  C=1;
    #50   B=1;
    #50   B=0;  C=0;
    #50   $finish;
    end
endmodule
```

例2：initial过程块用于对变量和存储器进行初始化。

```
module  register_initialize(memory);
inout  areg;
inout memory;
parameter  size=1024, bytesize=8;
reg  [bytesize-1:0]   memory [size-1:0];
```

```
initial
  begin:SEQ-BLK-A
    integer: index ;
    for(index=0; index<size; index=index+1)
      memory[index]=0;
      areg=0;
  end
endmodule
```

3. 两类语句在模块中的使用

```
module   tese
   reg  sa，  sb，  ze;
   initial
        begin
            sa=0;
            sb=0;
            #5  sb=1
            #5  sa=1;
            #5  sb=0;
        end
    always @ (sa  or  sb)    ze=sa^sb;
endmodule
```

二. 块语句

在Verilog HDL中有两类语句块：

1. 串行块（begin-end）

格式：

```
begin：<块名>
  块内局部变量说明;
时间控制1        行为语句1;
……
时间控制n        行为语句n;
end
```

说明：

1）串行块内的语句**按顺序方式**执行；

2）每条语句中的时延值与其前一条语句执行的仿真时间有关；

3）一旦顺序语句块执行结束，跟随顺序语句块过程的下一条语句继续执行。

例：用begin-end串行块产生信号波形

```
'timescale   10ns/1ns
   module   wave1;
   reg   wave ;
   parameter   cycle=10 ;
   initial
      begin
                     wave=0 ;
      #(cycle/2)   wave=1 ;
      #(cycle/2)   wave=0 ;
```

```
    #(cycle/2)   wave=1 ;
    #(cycle/2)   wave=0 ;
    #(cycle/2)   wave=1 ;
    #(cycle/2)   $finish ;
    end
initial  $monitor($time,,, “wave=%b”, wave);
endmodule
```

2. 并行块(fork-join)

格式：

fork：<块名>

块内局部变量说明；

时间控制1　　行为语句1；

……

时间控制n　　行为语句n；

join

说明：

1）**块内语句是同时执行**的，即程序流程控制一进入到该并行块，块内语句则开始同时并行执行。

2）块内每条语句的延迟时间是相对于程序流程控制进入到块内的仿真时间的。

3）延迟时间用来给赋值语句提供执行时序。

4）当按时间时序排序在最后的语句执行完后，程序流程控制跳出该程序块。

例：用fork-join并行块产生信号波形

```
'timescale   10ns/1ns
   module   wave2;
   reg   wave ;
   parameter   cycle=5 ;
   initial
      fork
                        wave=0 ;
      #(cycle)      wave=1 ;// 5*10ns延迟
      #(2*cycle)   wave=0 ;//2*5*10ns延迟
```

```
        #(3*cycle)    wave=1 ;  //3*5*10ns延迟
        #(4*cycle)    wave=0 ;
        #(5*cycle)    wave=1 ;
        #(6*cycle)    $finish ;
    join
initial  $monitor($time,,, “wave=%b”, wave);
endmodule
```

三.赋值语句

1. 持续赋值语句（不能出现在过程块中）

持续赋值语句只能对连线型变量wire进行赋值，不能对寄存器型变量进行赋值。

格式：

连线型变量类型 [*连线型变量位宽*] 连线型变量名

assign #（延时量） 连线型变量名=赋值表达式

可选项

“延时量”的基本格式：

#（delay1， delay2， delay3）

delay1——上升延时；

delay2——下降延时；

delay3——转移到高阻态延时。

如果“延时量”这项缺省，默认为0延时。

例：

```
module  and_cont_assignment(z,x,y);
input [3:0] x，  y;
output [3:0]z;
wire  [3:0]z，  x，  y;
assign  #(1.5，  1.0，  2.0) z=x&y;
endmodule
```

1）标量连线型

```
wire  a,b;
assign  a=b;
```

2）向量连线型

```
wire[ 7:0]  a,b;
assign  a=b;
```

3）向量连线型变量中的某一位

```
wire[ 7:0]  a,b;
assign  a[3]=b[3];
```

4）向量连线型变量中的某几位

```
wire [7:0]  a,b;
assign  a[3:2]=b[1:0];
```

5）上面几种类型的任意拼接运算

```
wire  a, c;
wire[1:0]  b;
assign  {a，  c}=b;
```

说明：

1）持续赋值用来描述组合逻辑。

2）持续赋值语句驱动连线型变量，输入操作数的值一发生变化，就重新计算并更新它所驱动的变量。

3）连线型变量没有数据保持能力。

4）若一个连线型变量没有得到任何连续驱动，则它的取值将为不定态“x”。

5）在仿真时，只要右端赋值表达式内的任一操作数发生变化，就会立即触发对被赋值连线型变量的更新操作。

6）如果持续赋值语句带有延时，则在仿真时只要右端赋值表达式中的任一信号发生变化，都将立即对赋值表达式进行重新计算，然后进入延时等待状态，待指定延时过去后再进行赋值。

例：用持续赋值语句实现4位全加器。

```
module  adder_4(a,b,ci,sum,co);
input  [3:0] a,b;
input ci;
output  [3:0] sum;
output co;
assign  {co,sum}=a+b+ci;
endmodule
```

2. 过程赋值语句

过程赋值是在always和initial语句内的赋值，
它只能对寄存器数据类型的变量赋值。

格式：

<被赋值变量> = <赋值表达式>

——阻塞型赋值

<被赋值变量> <= <赋值表达式>

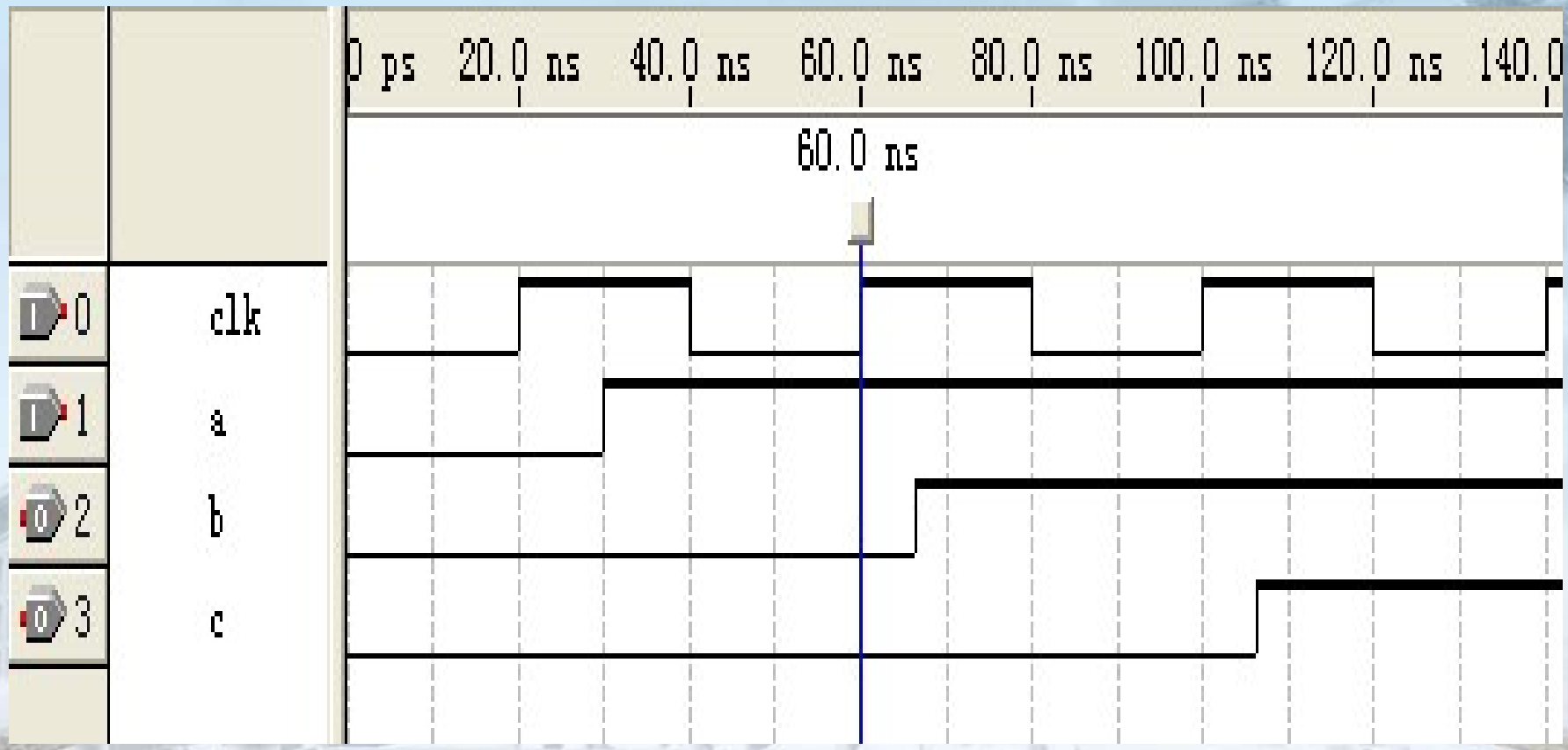
——非阻塞型赋值

1）非阻塞型赋值方式（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的值并不是立即就改变的。

例1：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
```

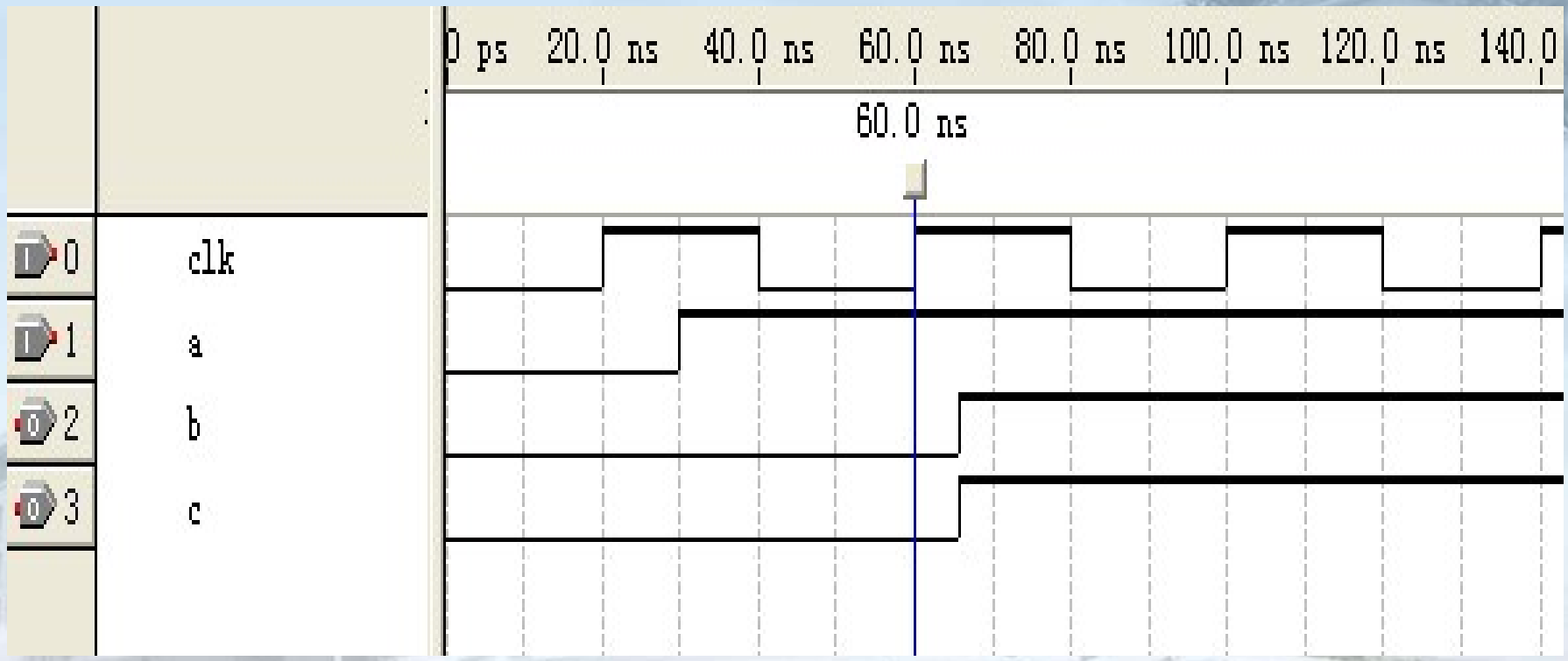

0 ps
20.0 ns
40.0 ns
60.0 ns
80.0 ns
100.0 ns
120.0 ns
140.0
60.0 ns
0
clk
1
a
2
b
3
c

2）阻塞赋值方式（如b=a;）

阻塞赋值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即改变，如果在一个语句块中有多条阻塞赋值语句，则前面赋值语句没有完成之前，后面赋值语句不能被执行，仿佛被阻塞一样。

例2：阻塞赋值

```
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
```


0 ps
20.0 ns
40.0 ns
60.0 ns
80.0 ns
100.0 ns
120.0 ns
140.0
60.0 ns
0
clk
1
a
2
b
3
c

四.条件语句

- 高级程序语句
  - 条件分支语句
    - if-else条件分支
    - case条件分支
  - 循环控制语句
    - forever
    - repeat
    - while
    - for

1. if-else

Verilog HDL语言提供了3种形式的if语句：

```
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；
```

说明：

1）3种形式的if语句在if后面都有“表达式”，一般为逻辑表达式或关系表达式。系统对表达式的值进行判断，**若为0，x，z，按“假”处理**；若为1，按“真”处理，执行指定语句。

2）在if和else后面可以包含单个或多个语句，多句时用“begin-end”块语句括起来。

3）在if语句嵌套使用时，要注意if与else的配对关系。

例1：
```
module   sel-from-three(q,sela,selb,a,b,c);
    input   sela,selb,a,b,c;
    output   q;
    reg   q;
    always   @(sela  or  selb  or  a  or  b  or  c);
      begin
          if(sela)  q=a;
          else  if(selb)  q=b;
          else  q=c;
      end
endmodule
```

例2：

```
module count60(qout，cout，data，load，cin，
                        reset，clk);
input [7:0] data;
input load，cin，reset，clk;
output [7:0] qout;
output cout;
reg [7:0] qout;
always @(posedge clk);
  begin
     if (reset)          qout<=0;
     else if (load)   qout<=data;
```

```
else   if (cin)
        begin
          if (qout[3:0]==9)
          begin
            qout[3:0]<=0;
            if (qout[7:4]==5)   qout[7:4]<=0;
            else  qout[7:4]<=qout[7:4]+1;
          end
          else    qout[3:0]<=qout[3:0]+1;
        end
   end
assign  cout=((qout==8'h59)&cin)?1:0;
endmodule
```

2. case

if语句有两个分支，而case语句是一种多路分支语句，故case语句可用于译码器、数据选择器、状态机、微处理器的指令译码等。

case语句有case、 casez、 casex三种表示方式：

1） case语句

格式：

```
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
```

例：BCD码-七段数码管显示译码

```
module  decode4_7(decodeout，indec);
output [6:0]  decodeout;
input [3:0]  indec;
reg [6:0]  decodeout;
always @ (indec)
  begin
     case (indec)
     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
```

2） casez与casex语句

casez与casex的格式与case完全相同，但在执行时有区别。

case语句比较规则

case	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

casez语句比较规则

casez	0	1	x	z
0	1	0	0	1
1	0	1	0	1
x	0	0	1	1
z	1	1	1	1

casex语句比较规则

casex	0	1	x	z
0	1	0	1	1
1	0	1	1	1
x	1	1	1	1
z	1	1	1	1

例1：用casez语句实现操作码译码

```
module   decode_ casez(a, b, opcode, out);
input[7:0]   a, b;
input[4:1]  opcode;
output[7:0]   out;
reg [7:0]  out;
always  @  (a  or  b  or  opcode)
   begin
       casez  (opcode)
       4'b1zzz:     out=a+b;
       4'b01xx:    out=a-b;
       4'b0001:   out=(~b)+1;
       endcase
   end
endmodule
```

例2：用casex语句实现操作码译码

```
module   decode_ casez(a, b, opcode, out);
input[7:0]   a, b;
input[4:1]  opcode;
output[7:0]   out;
reg [7:0]  out;
always  @  (a  or  b  or  opcode)
        begin
        casex  (opcode)
        4'b1zzx:     out=a+b;
        4'b01xx:    out=a-b;
        4'b0001:   out=(~b)+1;
        endcase
    end
endmodule
```

3. 条件语句使用要点

在使用条件语句时，应注意列出所有条件分支，否则编译器认为条件不满足时，会引进一个锁存器保持原值。在组合电路中应避免这种隐含锁存器的存在。

因为每个变量至少有4种取值，为包含所有分支，可在if语句后加上 else；在 case语句后加上 default。

例：隐含锁存器举例

```
module buried_ff(c,b,a);
output c;
input b,a;
reg c;
always @(a or b)
      begin
            if((b==1)&&(a==1))  c=a&b;
      end
endmodule
```

else c=0;

五. 循环语句

有4种类型的循环语句，可用来控制语句的执行次数：

（1）forever：连续地执行语句。

（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。

1. forever语句

功能：无限循环。一般用在initial中。

格式：

```
forever   语句;
```

或

```
forever   begin
          .....
          end
```

用途：产生周期性波形作为仿真测试信号。

例1：

```
module   clk_gen(clk);
output   clk;
initial
   begin
        clk=0;
        #1000;
        forever
        #25  clk=~clk;
   end
endmodule
```

例2:

```
module   clk_gen(clk);
output   clk;
integer  counter;
initial
  begin
    counter=0;
    clk=0;
    #1000;
```

```
begin: FOREVER_PART
    forever
    begin
      counter=counter +1;
      if (counter>200)   disable FOREVER_PART;
       #25  clk=~clk;
    end
  end
end
endmodule
```

2. repeat语句

功能：该循环语句内的循环体部分被重复执行指定的次数。

格式：

repeat （循环次数表达式） 语句；

或 repeat （循环次数表达式）

begin

……

end

例1：用repeat循环语句来实现循环移位

```
module   drift (data,  num,  ctrl);
inout [16: 1]  data;
input [4: 1]  num;
input    ctrl;
reg [16: 1]  data;
reg    tmp;
always  @ (ctrl)
      if (ctrl==1)
```

```
repeat (num)
begin
        tmp=data[16];
        data={data[15:0], tmp};
end
endmodule
```

例2：用repeat实现8位二进制数的乘法

```
module mult_repeat(outcome,a,b);
parameter size=8;
input[size:1] a,b;
output[2*size:1] outcome;
reg[2*size:1] temp_a,outcome;
reg[size:1] temp_b;
always @(a or b)
begin
outcome=0;
```

```
temp_a=a;
temp_b=b;
repeat(size)      // size为循环次数
  begin
  if(temp_b[1])
  outcome=outcome+temp_a;
  temp_a=temp_a<<1;      //操作数a左移一位
  temp_b=temp_b>>1;      //操作数b右移一位
  end
end
endmodule
```

3. while语句

功能：条件循环。

格式：

```
        while  (循环执行条件表达式)     语句;
或      while  (循环执行条件表达式)
                 begin
                 ……
                 end
```

4. for语句

功能：条件循环。只有在指定的条件表达式成立时才进行循环。

格式：

for（循环变量赋初值；循环条件结束；循环变量增值）

执行语句；

§5 进程、任务与函数

- 进程
- 任务
- 函数
- 任务与函数的区别

一. 进程

表示进程的方法：

- always过程块
- initial过程
- assign赋值语句
- 元件例化

1. 进程的特点

1) 进程只有两种状态：执行、等待。

2) 进程一般由敏感信号的变化来启动。

3) 进程内部的语句是顺序执行的。

4) 多个进程之间是并行执行的，与进程在程序中的位置无关。

5) 进程之间的通信是由信号来传递的。

2. 举例：加法计数器中的进程

```
module count(data,clk,reset,load,cout,qout);

output cout;

output[3:0] qout;

reg[3:0] qout;

input[3:0] data;

input clk,reset,load;
```

```
always @(posedge clk)
begin
    if (!reset)       qout= 4'h00;
    else if (load)  qout= data;
         else          qout=qout + 1;
end
assign cout=(qout==4'hf)?1:0;
endmodule
```

二. 任务

行为描述模块内可以包含任务和函数定义，这两部分在行为描述模块中都是可选的，类似于一种子程序结构。

引入任务和函数的目的：

将一个很大的程序模块分解成许多较小任务和函数，便于理解和调试。

1. 任务的定义

```
task  < 任务名 >;     //注意无端口列表
      端口及数据类型声明语句;
      局部变量说明;
      语句1;
      ……
      语句n
endtask
```

说明：

1. 端口及数据类型声明语句

   用于对任务各个端口的宽度和类型进行说明。

2. 局部变量说明

   用来对任务内用到的局部变量进行宽度和类型说明。

3. 语句1~语句n

   行为语句，指明了任务被调用时需要进行的操作。

4. 任务定义与“过程块”、“持续赋值语句”及“函数定义”以并列的方式存在于行为描述中，“任务定义”不能出现在任何过程块的内部。

任务定义举例：

```
task   read_mem;
 input  [15: 0]   address;
 output  [31: 0]   data;
 reg  [3: 0]     counter;
reg  [7: 0]     temp [4:1];
 begin
    for (counter=1;  counter<=4; counter=counter+1)
        temp[counter]=mem[address+counter-1];
    data={temp[1],temp[2],temp[3],temp[4]};
 end
endtask
```

2. 任务的调用

格式：

＜任务名＞ （端口1，端口2，…，端口n）；

说明：

1. 任务调用语句只能出现在过程块中。
2. 当被调用的任务具有输入、输出端口时，任务调用语句必须包含端口列表，其列表内各端口名出现的顺序和类型必须与任务定义结构中端口说明部分的端口顺序和类型一致。
3. 只有寄存器类的变量才能与任务的输出端口相对应。

对任务“read_mem”进行调用

```
module   demo_task_invo;
reg [7: 0]  mem[128: 0];
reg  [15: 0]   a;
reg  [31: 0]   b;
initial
   begin
        a=0;
        read_mem(a,  b);
        #10    a=64;
        read_mem(a, b);
   end
        <任务“ read_mem”定义部分 >
endmodule
```

三. 函数

1. 函数的定义

```
function  < 返回值位宽或类型说明 >  函数名;
        输入端口声明;
        局部变量定义;
        begin  行为语句1;
        ……
        行为语句n
        end
endfunction
```

定义函数举例

```
function  [7: 0]   get0;
input [7: 0]   x;
reg [7: 0]   count;
integer   i;
   begin
       count=0;
       for (i=0;  i<=7;  i=i+1)
       if (x[i]=1'b0)   count=count+1;
       get0=count;
   end
endfunction
```

2. 函数的调用

格式:

<函数名> （<输入表达式1>，…<输入表达式m>）；

说明:

1）函数的调用不能单独作为一条语句出现，只能作为操作数出现在调用语句内。

例1：用函数和case语句描述的编码器（不含优先顺序）

```
module code_83(din,dout);
input[7:0] din;
output[2:0] dout;
function[2:0] code;
    input[7:0] din;
    casex (din)
        8'b1xxx_xxxx : code = 3'h7;
        8'b01xx_xxxx : code = 3'h6;
        8'b001x_xxxx : code = 3'h5;
```

```
        8'b0001_xxxx : code = 3'h4;
        8'b0000_1xxx : code = 3'h3;
        8'b0000_01xx : code = 3'h2;
        8'b0000_001x : code = 3'h1;
        8'b0000_000x : code = 3'h0;
        default: code = 3'hx;
    endcase
endfunction
assign dout = code(din) ;
endmodule
```

2）函数调用既能出现在过程块中，也能出现在assign持续赋值语句中。

3）定义函数时，没有端口列表名，但调用函数时，需列出端口列表名，端口名的排序和类型必须与定义时的相一致。

4）函数不能调用任务，但任务可以调用别的任务和函数，且调用个数不限。

5）函数的调用与定义必须在一个module模块内。

3. 任务与函数的区别

1）输入与输出

任务：可有任意个各种类型的参数。

函数：至少有一个输入，不能将inout类型作为输出。

2）调用

任务：只可在过程语句中调用，不能在assign中调用。

3）定时和事件控制（@，#和wait）

任务：可包含定时和事件控制语句。

函数：不可包含定时和事件控制语句。

4）调用其它任务和函数

任务：可调用其它任务和函数。

函数：可调用其它函数，但不可调用其它任务。

5）返回值

任务：不向表达式返回值。

函数：向调用它的表达式返回一个值。

§6 Verilog HDL的描述风格

- 结构描述
- 行为描述
- 数据流描述

一. 结构描述（门级描述）

结构描述方式是将硬件电路描述成一个分级子模块系统的一种描述方式。在这种描述方式下，组成硬件电路的各个子模块之间的相互层次关系及相互连接关系都需要得到说明。

可通过如下方式来描述电路的结构：

1）调用Verilog内置门元件（门级结构描述）

2）调用开关级元件（开关级结构描述）

3）用户自定义元件UDP（门级）

1. Verilog内置门元件

共内置26个基本元件，其中14个是门级元件，12个为开关级元件。

1）多输入门

and——与门；　　xor——异或门；

nand——与非门；　　xnor——异或非门；

nor——或非门；　　or——或门；

2）多输出门

buf——缓冲器；　　not——非门。

3）三态门

bufif1—高电平使能缓冲器　bufif1—低电平使能缓冲器

notif1—高电平使能非门　　notif0—低电平使能非门

2. 门元件的调用

格式：

门元件名字　例化的门名字　（端口列表）

其中：

1）多输入门的端口列表按下面的顺序列出：

(输出, 输入1, 输入2, ……);

例：/*三输入与门，名字为a1*/

```
and  a1(out, in1, in2, in3);
```

2）对三态门，按如下顺序列出输入、输出端口：

(输出, 输入, 使能控制端);

例：/*高电平使能的三态门*/

bufif1 mytri1(out, in, enable);

3）对buf和not两种元件的调用，允许有多个输出，但只能有一个输入。

例：/*1个输入in，2个输出out1，out2 */

not n1(out1, out2, in);

例：调用门元件实现4选一数据选择器。

```
module mux4_1a(out,in1,in2,in3,in4,cntrl1,cntrl2);
output out;
input in1,in2,in3,in4,cntrl1,cntrl2;
wire notcntrl1,notcntrl2,w,x,y,z;
not   (notcntrl1,cntrl2),
      (notcntrl2,cntrl2);
```

```
and   (w,in1,notcntrl1,notcntrl2),
      (x,in2,notcntrl1,cntrl2),
      (y,in3,cntrl1,notcntrl2),
      (z,in4,cntrl1,cntrl2);
or    (out,w,x,y,z);
endmodule
```

二. 行为描述

行为描述的目标不是对电路的具体硬件结构进行说明，仅从电路的行为和功能的角度来描述某一电路模块。

对设计者来说，采用的描述级别越高，设计越容易。所以从容易设计的角度，在电路设计中，除非一些关键路径的设计采用结构描述外，一般更多地采用行为描述方式。

例：用case语句实现4选一数据选择器。

```
module
mux4_1b(out,in1,in2,in3,in4,cntrl1,cntrl2);

output out;

input in1,in2,in3,in4,cntrl1,cntrl2;

reg out;

always@(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)
```

```
case({cntrl1,cntrl2})
2'b00:out=in1;
2'b01:out=in2;
2'b10:out=in3;
2'b11:out=in4;
default:out=2'bx;
endcase
endmodule
```

三. 数据流描述（RTL级描述）

数据流描述方式与布尔表达式比较类似，在这种描述方式中，电路不再被描述为逻辑单元之间的连接，而是被描述为一系列赋值语句。通常，在数据流描述方式中使用的是持续赋值语句assign。

三. 数据流描述（RTL级描述）

例：数据流方式描述的4选一数据选择器

```
module
mux4_1c(out,in1,in2,in3,in4,cntrl1,cntrl2);
output out;
input in1,in2,in3,in4,cntrl1,cntrl2;
assign out=(in1 & ~cntrl1 & ~cntrl2)|
           (in2 & ~cntrl1 & cntrl2)|
           (in3 & cntrl1 &~cntrl2)|
           (in4 & cntrl1 & cntrl2);
endmodule
```

§7 Verilog HDL设计

组合逻辑电路举例

1. 三态门

```
module tri_1(in,en,out);
input in,en;
output out;
tri out;
bufif1 b1(out,in,en); //注意三态门端口的排列顺序
endmodule
```

用数据流方式描述的三态门：

```
module tri_2(out,in,en);

output out;

input in,en;

assign out = en ? in : 'bz;

endmodule
```

2. 译码器

```
module decoder_38(out,in);
output[7:0] out;
input[2:0] in;
reg[7:0] out;
always @(in)
begin
```

```
    case(in)
  3'd0: out=8'b11111110;
  3'd1: out=8'b11111101;
  3'd2: out=8'b11111011;
  3'd3: out=8'b11110111;
  3'd4: out=8'b11101111;
  3'd5: out=8'b11011111;
  3'd6: out=8'b10111111;
  3'd7: out=8'b01111111;
    endcase
  end
  endmodule
```

3. 8-3优先编码器

```
module encoder8_3(none_on,outcode,a,b,c,d,e,f,g,h);

output none_on;

output[2:0] outcode;

input a,b,c,d,e,f,g,h;

reg[3:0] outtemp;

assign {none_on,outcode}=outtemp;

always @(a or b or c or d or e or f or g or h)
begin
```

```
        if(h)       outtemp=4'b0111;
        else if(g)  outtemp=4'b0110;
        else if(f)  outtemp=4'b0101;
        else if(e)  outtemp=4'b0100;
        else if(d)  outtemp=4'b0011;
        else if(c)  outtemp=4'b0010;
        else if(b)  outtemp=4'b0001;
        else if(a)  outtemp=4'b0000;
        else        outtemp=4'b1000;
    end
endmodule
```

时序逻辑电路举例

1. 基本D触发器

```
module DFF(Q,D,CLK);
output Q;
input D,CLK;
reg Q;
always @(posedge CLK)
begin
Q <= D;
end
endmodule
```

2. 带异步清0、置1的D触发器

```
module DFF1(q,qn,d,clk,set,reset);
input d,clk,set,reset;
output q,qn;
reg q,qn;
always @(posedge clk or negedge set or negedge reset)
begin
if (!reset)
   begin
      q <= 0;
      qn <= 1;
   end
```

```
    else if (!set)
    begin
        q <= 1;
        qn <= 0;
    end
    else
    begin
        q <= d;
        qn <= ~d;
    end
end
endmodule
```

3. 带异步清0、置1的JK触发器

```
module JK_FF(CLK,J,K,Q,RS,SET);
input CLK,J,K,SET,RS;
output Q;
reg Q;
always @(posedge CLK or negedge RS or
negedge SET)
begin
if(!RS)       Q <= 1'b0;
else if(!SET)      Q <= 1'b1;
else
```

```
        case({J,K})
        2'b00 :     Q <= Q;
        2'b01 :     Q <= 1'b0;
        2'b10 :     Q <= 1'b1;
        2'b11 :     Q <= ~Q;
        default:    Q<= 1'bx;
        endcase
    end
    endmodule
```

4. 8位数据锁存器

```
module latch_8(qout,data,clk);
output[7:0] qout;
input[7:0] data;
input clk;
reg[7:0] qout;
always @(clk or data)
begin
if (clk) qout<=data;
end
endmodule
```

5. 8位移位寄存器

```
module shifter(din,clk,clr,dout);
input din,clk,clr;
output[7:0] dout;
reg[7:0] dout;
always @(posedge clk)
begin
  if (clr)  dout<= 8'b0;
```

```
else
  begin
  dout <= dout << 1;
  dout[0] <= din;
  end
end
endmodule
```

§8 Verilog HDL优化设计

一、层次化设计方法分层原则

- 将所有的算术运算安排在同一层中，状态机、随机逻辑、数据路径等逻辑类型作为独立的模块设计
- 模块的输入尽量不要悬空，输出应尽量寄存。
- 单个功能块应保持在2000～5000门之间，HDL语言的行数不超过400行
- 尽量采用专用的IP核进行设计，或工艺库中预定义的诸如RAM、ROM、乘法器等。

二、模型的优化

- 综合所生成的逻辑易受到模型描述方式的影响，把语句从一个位置移到另一个位置，或者拆分表达式都会对所生成的逻辑产生重大影响，这可能会造成综合出的逻辑门数有所增减，也可能改变其他定时特性。

三、资源分配

- 所谓资源分配指的是互斥条件下共享算术逻辑运算单元(ALU)的过程
- 可以通过多路选择器的引入减少ALU的个数，从而节省资源

```
if(!ShReg)
  DataOut = AddLoad + ChipSelectN;
else if(ReadWrite)
  DataOut = ReadN+WriteN;
else
  DataOut = AddLoad +ReadN;
```

使用了3个加法器

```
if(!ShReg)
  begin
      temp1 = AddLoad;
      temp2 = ChipSelectN;
   end
else if(ReadWrite)
  begin
      temp1 = ReadN;
      temp2 = WriteN;
   end
else
 begin
      temp1 = AddLoad;
      temp2 = ReadN;
   end
  DataOut = temp1 +temp2;
```

只用了1个加法器，但多了一个多路选择器，注意权衡

四、公共子表达式提取

- **交换律与结合率的使用**

```
lam = a – (b+c);
lom = a + (b-c);
应提取temp = a-c
```

- **for循环中的子式提到循环外**

```
for(…..) begin
……;  tip = car -6;
end
应改为
temp = car -6;
for(…..) begin
……;  tip = temp;
end
```

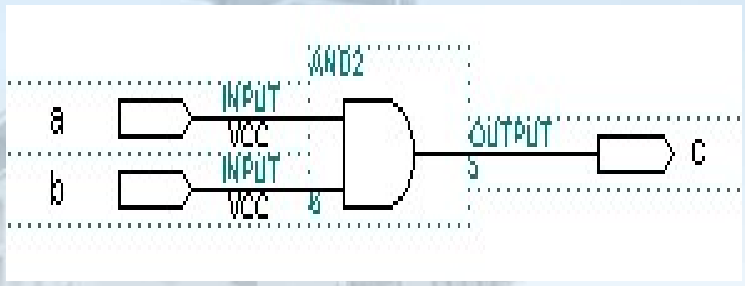
- **if/case语句的互斥分支中的子式提到条件语句外**

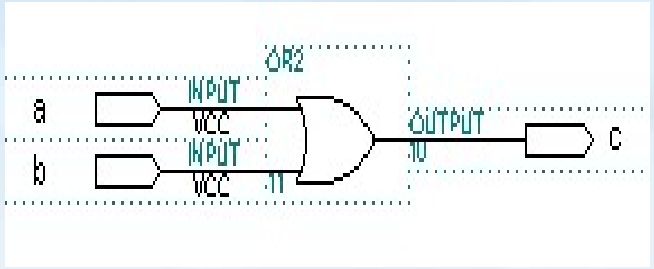
```
if(test)
  ax = a & (b + c);
else
  by = (b + c) | t;
应改为
temp = b + c;
if(test)
  ax = a & temp;
else
  by = temp  | t;
```

五、毛刺的解决

一个最简单的组合逻辑电路

- 一个两输入的与门或者两输入的或门是最简单的电路

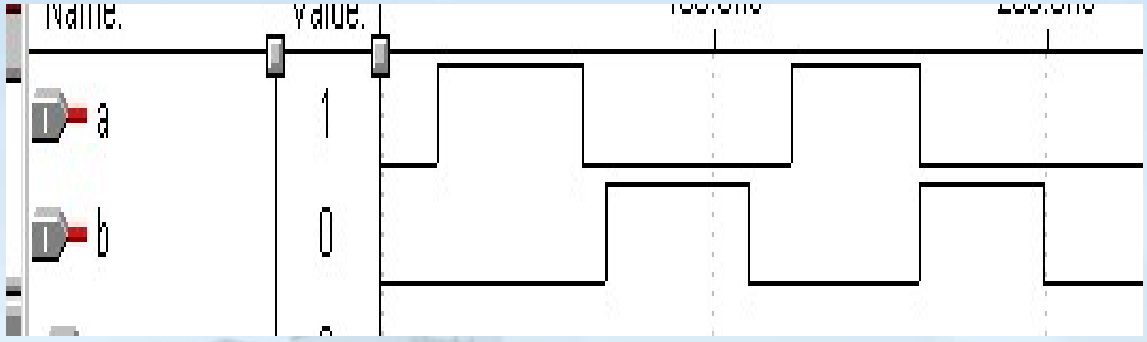




！！！但可编程器件对这么简单的电路的处理也会出错

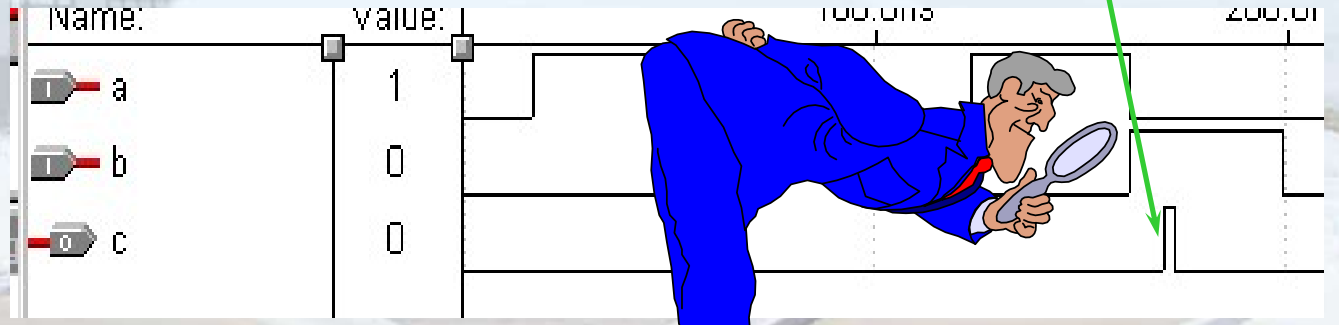
两输入与门的仿真结果

但问题真的是这么简单?



输入波形

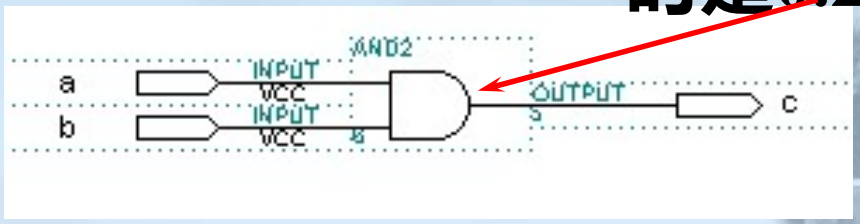
这部分有问题
这一定是器件的原因

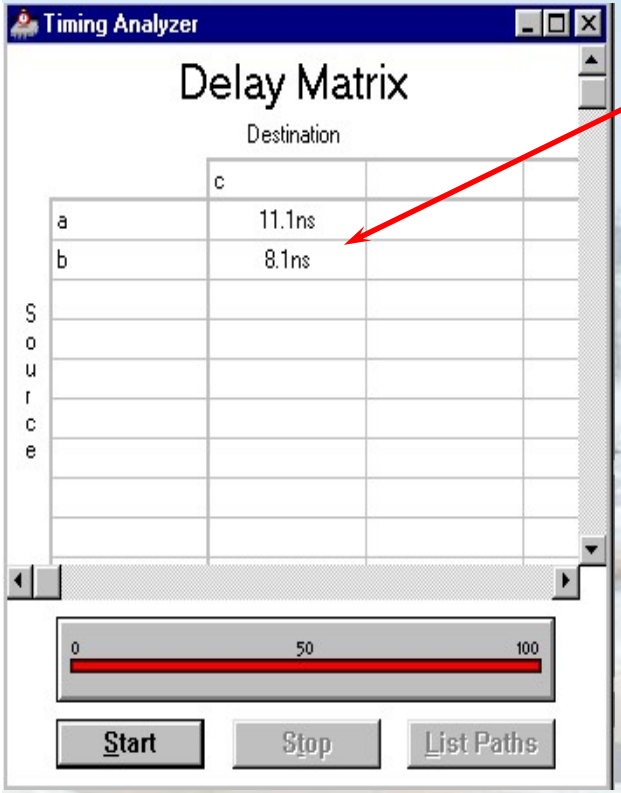


输出波形

再进一步分析

假设与门的内部延时是0.2nS





这代表什么意思呢?

对信号B进行简单计算:

(Trace delay of b) + AND gate internal delay = 8.1ns

(Trace delay of b) + 0.2ns = 8.1ns

(Trace delay of b) = 7.9ns

对信号A进行简单计算:

(Trace delay of a) + AND gate internal delay = 11.1ns

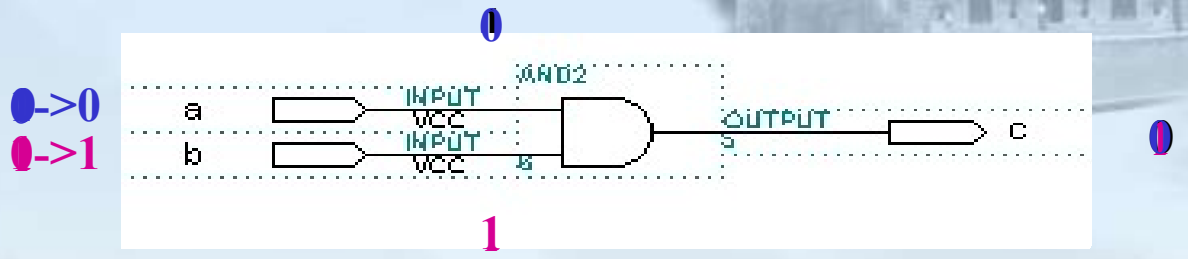
(Trace delay of a) + 0.2ns = 11.1ns

(Trace delay of a) = 10.9ns

结果再分析

(Trace delay of a) = 10.9ns　　(Trace delay of b) = 7.9ns

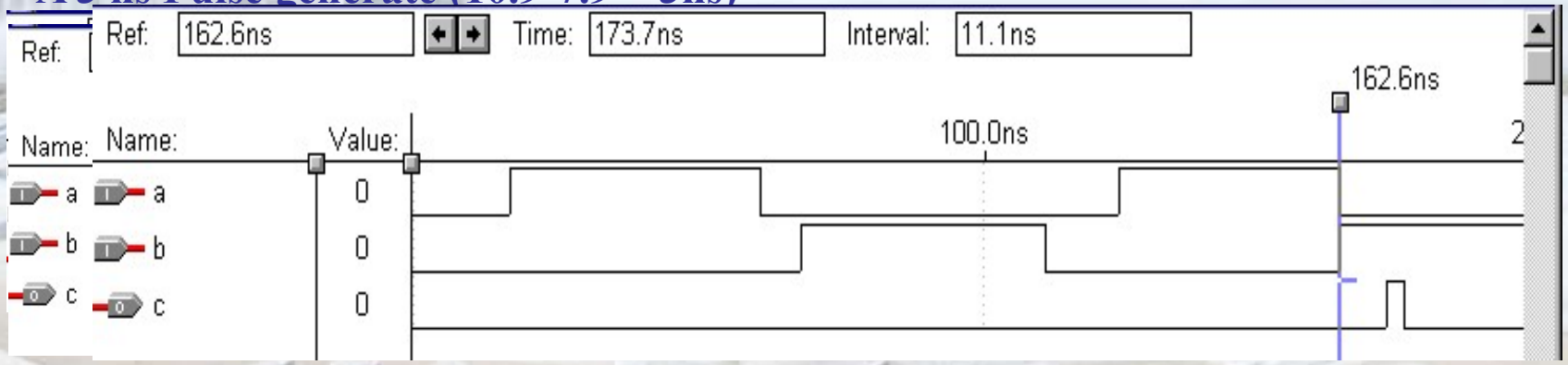
Time : 7.9ns



Output C change from "0" to "1" at 8.1ns

Output C change back from "1" to "0" as the final result

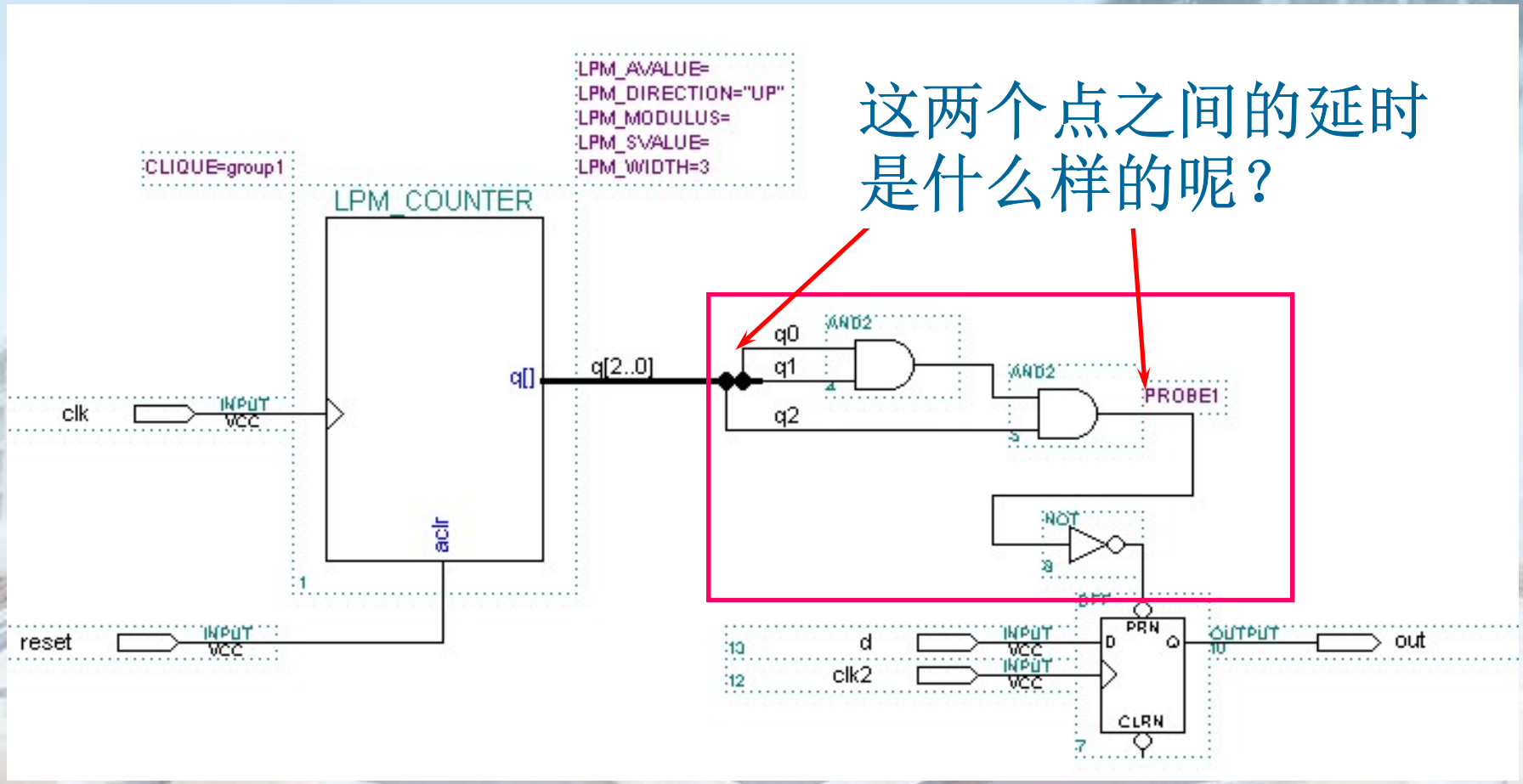
A 3 ns Pulse generate (10.9-7.9 = 3ns)



总　结

- 设计一个 2输入的与门也不是像1+1=2那么简单
- 在组合逻辑设计中我们需要考虑 Trace Delay and Gate Delay
- 函数：C的输出为 “0”
- 时序：C的输出有一个3ns宽的毛刺
- 这3ns的毛刺主要是由 Trace Delay造成的
- 组合电路工作时不仅和逻辑函数相关还和时序相关
- 当某一时刻同时有一个以上的信号发生变化时容易产生毛刺
- 组合逻辑电路是会产生毛刺的

再讨论一个例子


这两个点之间的延时
是什么样的呢？
LPM_AVALUE=
LPM_DIRECTION="UP"
LPM_MODULUS=
LPM_SVALUE=
LPM_WIDTH=3
CLIQUE=group1
LPM_COUNTER
clk
INPUT
VCC
q[]
q[2..0]
aclr
reset
q0
q1
q2
AND2
AND2
PROBE1
NOT
d
clk2
PRN
D
Q
CLRN
OUTPUT
out

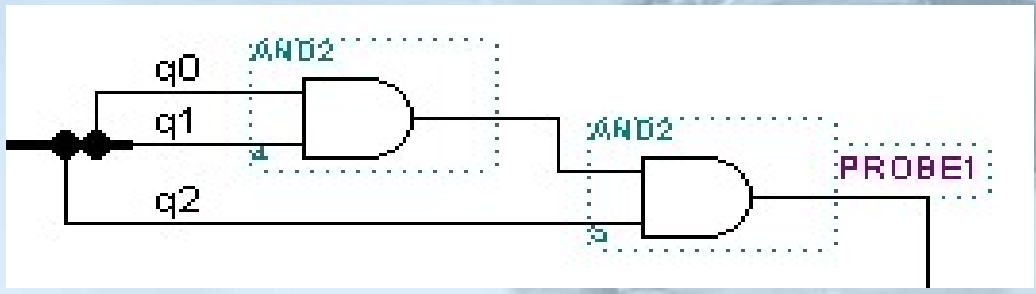
分　析

Timing Analyzer

Delay Matrix

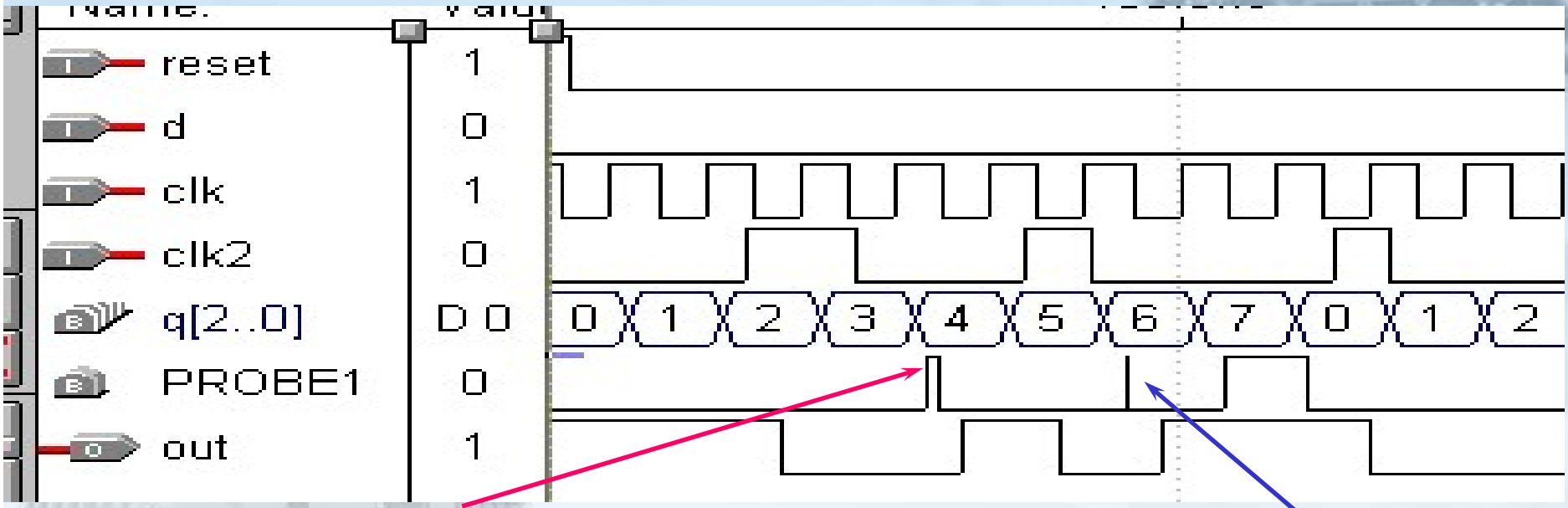
Destination

	PROBE1
lpm_counter:1|dffs[0]	3.9ns
lpm_counter:1|dffs[1]	3.7ns
lpm_counter:1|dffs[2]	2.0ns



q2	q1	q0		
0	0	0	← 毛刺？	No
0	0	1	← 毛刺?	No
0	1	0	← 毛刺?	No
0	1	1	← 毛刺?	Yes
1	0	0	← 毛刺?	No
1	0	1	← 毛刺?	Yes
1	1	0	← 毛刺?	No
1	1	1		

仿真结果分析



从 “3”变到 “4”的时候产生毛刺

从 “5”变到 “6”的时候产生毛刺

两个不希望看到的毛刺

仿真结果正确，可编程器件没有问题

毛刺的宽度

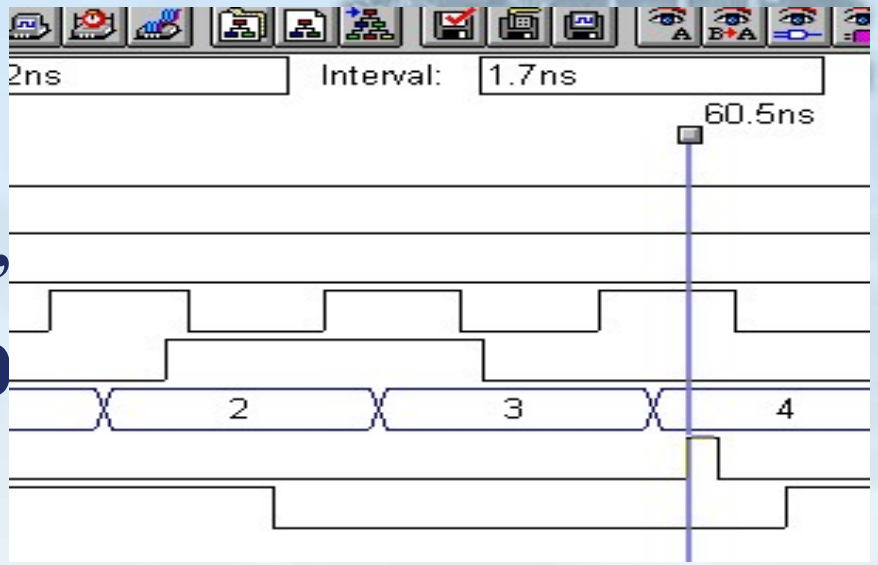
Timing Analyzer

Delay Matrix

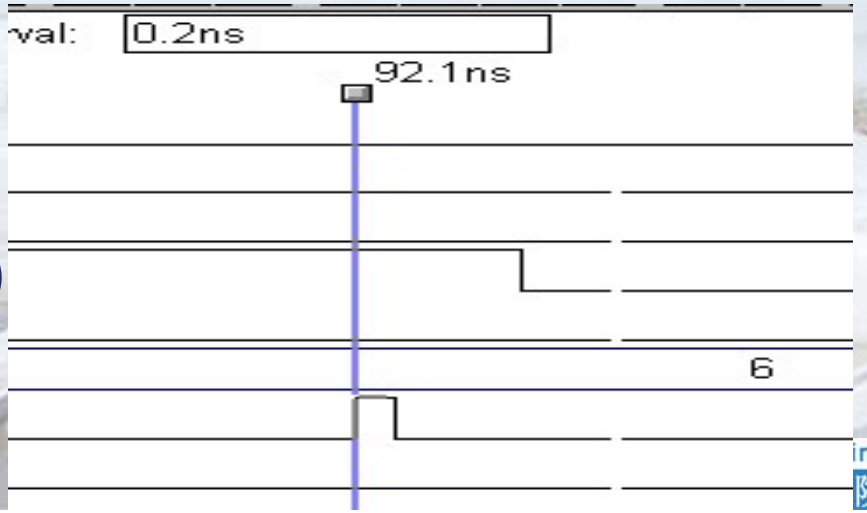
Destination

	PROBE1
lpm_counter:1|dffs0.Q	3.9ns
lpm_counter:1|dffs1.Q	3.7ns
lpm_counter:1|dffs2.Q	2.0ns

“3” to “4”
011 -> 100



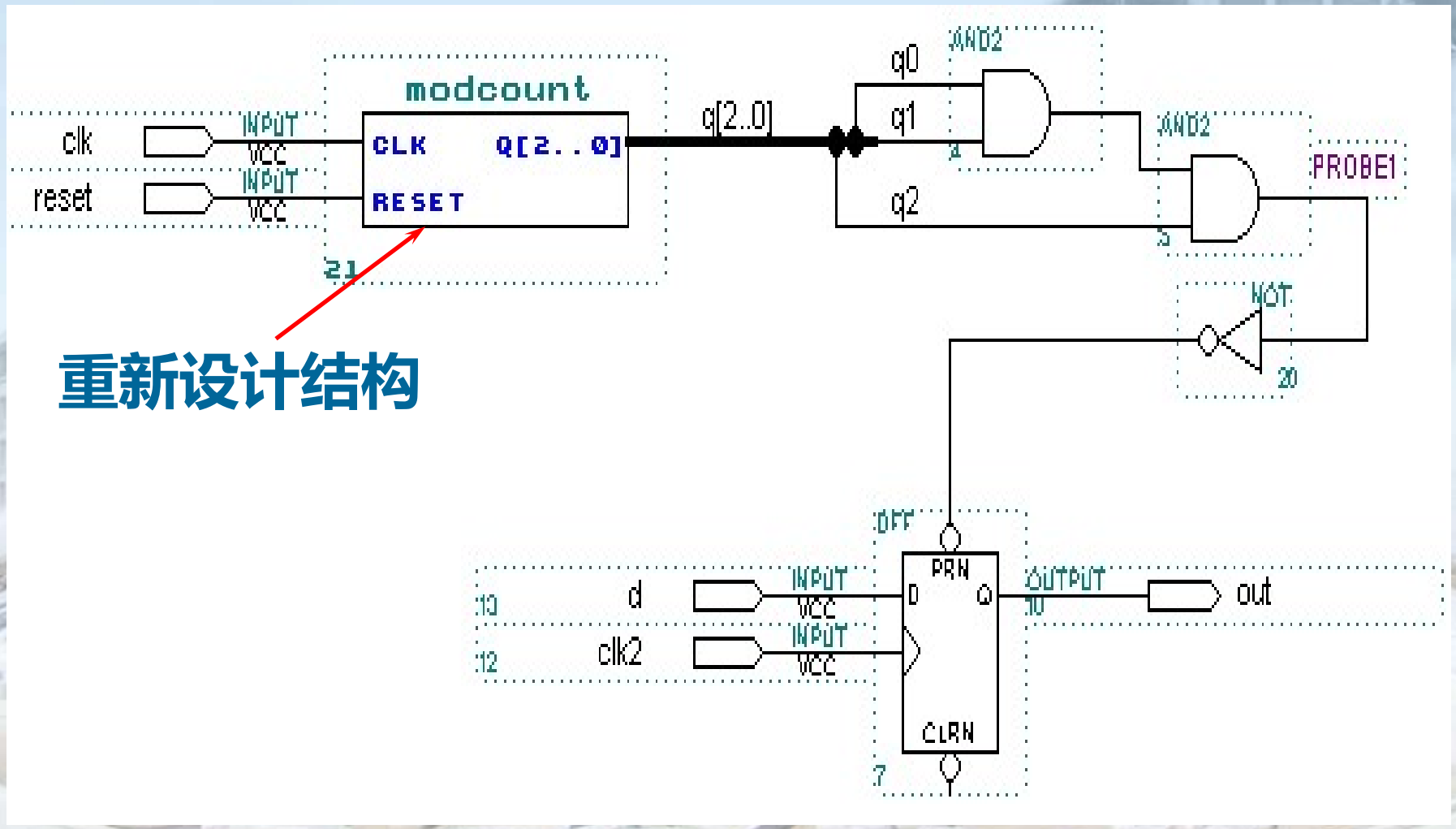
“5” to “6”
101 -> 110



结　论

- 如果我们知道毛刺是怎么产生的
  - 我们可以计算出毛刺出现的具体时间
  - 我们可以计算出毛刺的脉冲宽度
- 当组合逻辑输出用做以下功能时必须加以注意
  - 触发器的CLEAR端
  - 触发器的PRESET 端
  - 触发器的CLOCK 端
  - 锁存器的控制端
  - 其他 ……

消除毛刺的方法(一)


modcount
clk
reset
INPUT
VCC
CLK
RESET
Q[2..0]
q[2..0]
q0
q1
q2
AND2
AND2
PROBE1
NOT
DFF
PRN
D
Q
CLRN
d
clk2
OUTPUT
out
重新设计结构

格雷码（相邻码只有一位改变）

0	0000	8	1100
1	0001	9	1101
2	0011	10	1111
3	0010	11	1110
4	0110	12	1010
5	0111	13	1011
6	0101	14	1001
7	0100	15	1000

消除毛刺的方法(二)

有毛刺的程序：

```
module longframe1(clk,strb);
parameter delay=8;
input clk;
output strb;
reg strb;
reg[7:0] counter;
```

```
always@(posedge clk)
begin
        if(counter==255)      counter=0;
         else                 counter=counter+1;
 end
always@(counter)
 begin
    if(counter<=(delay-1))  strb=1;
    else                    strb=0;
end
endmodule
```

输出端加 D 触发器后消除毛刺的程序：

```
module longframe2(clk,strb);
parameter delay=8;
input clk;
output strb;
reg[7:0] counter;
reg temp;
reg strb;
```

```
always@(posedge clk)
begin
        if(counter==255)      counter=0;
       else                   counter=counter+1;
 end

always@(posedge clk)
     begin
     strb=temp;                    //引入一个触发器
     end
```

```
always@(counter)
begin
        if(counter<=(delay-1))  temp=1;
        else                    temp=0;
end
endmodule
```