



# 第11章 Verilog HDL

**11.1 Verilog HDL程序的基本结构**

**11.2 Verilog HDL的基本元素**

**11.3 Verilog HDL的基本语句**

**11.4 Verilog HDL程序设计示例**

# 第 11 章 Verilog HDL

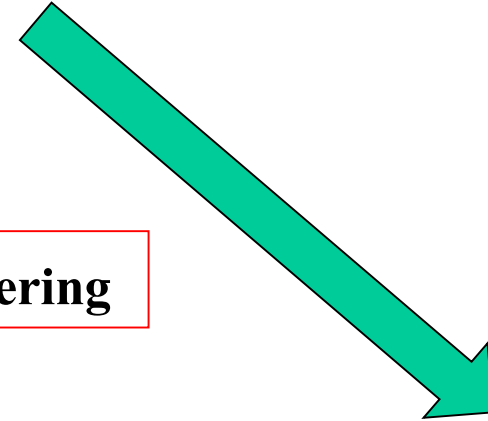
## Verilog Hardware Description Language

电子设计自动化已经成为电路设计的基本工具，代表着电路系统设计的发展水平。

**CAD** Computer Aided Design

**CAE** Computer Aided Engineering

**EDA** Electronic Design Automation



硬件描述语言（HDL）是EDA使用的基本语言。有VHDL和Verilog HDL两种。Verilog HDL是工业界使用最广泛的语言。

## Verilog HDL 语言特点

1. 既能进行电路设计与综合，又可以用于模拟仿真。
2. 能够在多个层次上对所设计的系统加以描述。

开关级，门级，寄存器传输级（RTL），算法级，系统级

对设计规模没有限制

### 3. 电路描述风格灵活多样

行为表述； 结构描述； 数据流表述

支持混合建模：各个模块可以在不同的设计层次建模与描述

行为描述语句： 类似于高级语言

如条件语句、赋值语句、循环语句

结构描述语句

门级结构描述： `and, or, nand`

开关及建模： `pmos, nmos, cmos`

用户定义原语句（UDP）

既可以是组合逻辑也可以是时序逻辑，还可以通过编程语言接口PLI机制进行扩展。

## § 11.1 Verilog HDL程序的基本结构

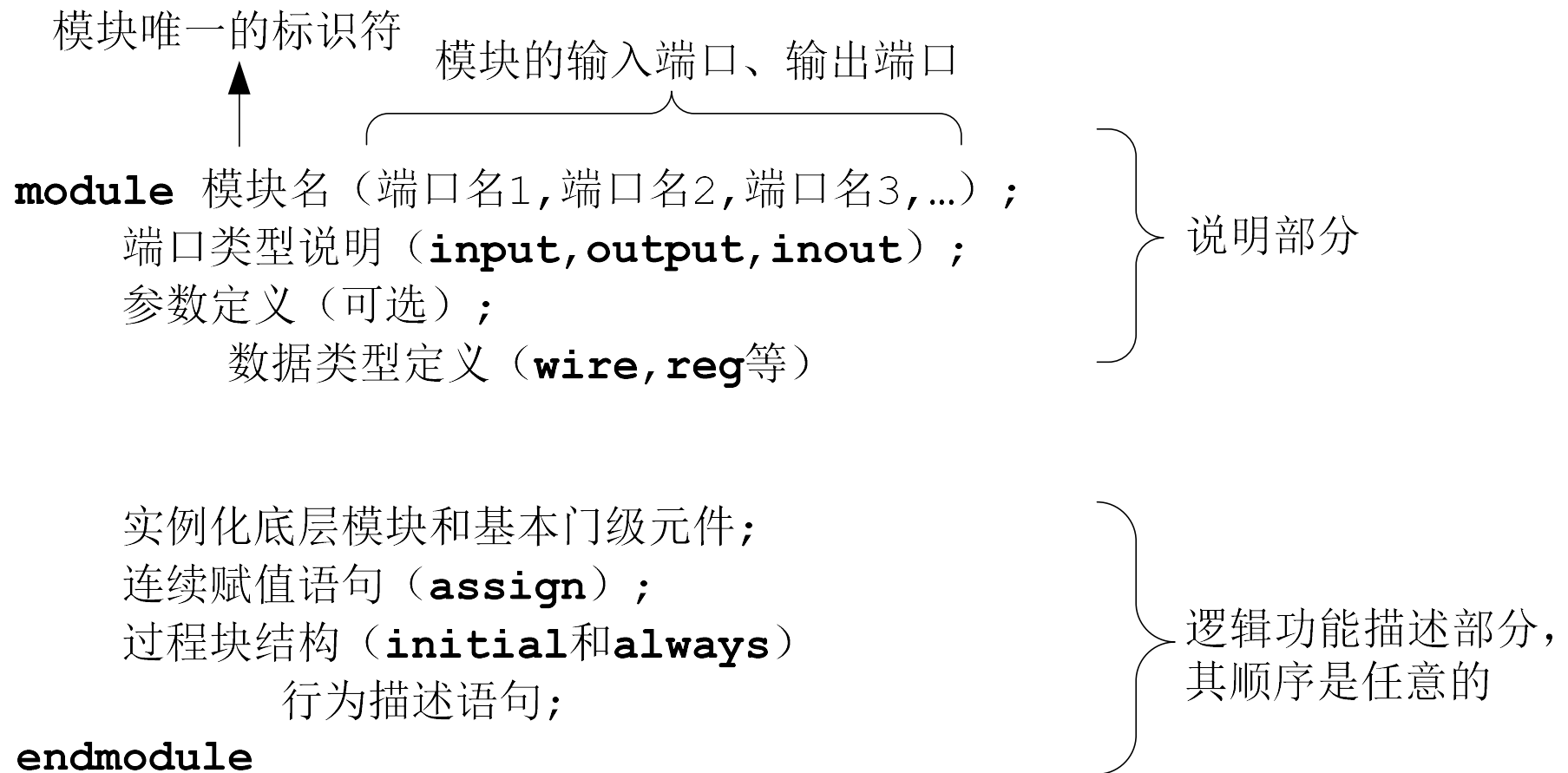
Verilog HDL程序是由若干个模块构成，每一个模块又可以由若干个子模块构成。

系统级、算法级、**RTL级**、门级、开关级

模块包括4个主要部分：

{	端口定义	声明输入输出端口与类型
	端口类型说明	
	数据类型定义	定义是模块中所用到的信号
	逻辑功能描述	使用例化、赋值语句或过程语句结构进行功能描述

# 模块的语法结构



## 1. 模块的端口定义 声明了模块的输入输出口

端口表示模块的输入和输出口名，是与别的模块联系端口的标识

引用时的连接方式：顺序连接；端口名称引用

例：模块名(连接端口1信号名,连接端口2信号名,连接端口3信号名,...); 顺序

模块名(.端口1名(连接信号1名),.端口2名(连接信号2名),...); 端口名称引用

## 2. 端口类型说明 凡是在端口定义中出现的端口都必须有类型说明

例：

`input`[信号位宽-1:0]端口名;

`output`[信号位宽-1:0]端口名;

`inout`[信号位宽-1:0]端口名;

可以与端口定义语句合并

```
module 模块名(input port1, input port2, ... output port1, output port2...);
```

### 3. 数据类型定义      对信号（端口、节点信号等）进行定义

指定数据对象为寄存器型（reg）、线型（wire）

## 4. 逻辑功能描述

### （1）结构描述方法      - 例化

实例化底层模块，即调用其他已定义好的底层模块对电路的功能进行描述，或调用Verilog内部基本门级元件描述电路的结构。



```
and U1(c,a,b);    //一个名为U1的与门，a和b为输入，c为输出
```

## (2) 数据流描述 - 连续赋值

```
assign a=b&c;    //描述了一个有两输入的与门
```

## (3) 行为描述方法 - 过程块语句机构

```
always @(posedge areg) //当areg信号的上升沿出现时把tick信号反相  
begin  
tick=~tick;  
end
```

## § 11.2 Verilog HDL的基本元素

### 1. 注释符

改善程序可读性，在编译时不起作用

<code>/*.....*/</code>	为多行注释符
<code>//</code>	为单行注释符

### 2. 标识符

用于定义模块名、端口名、信号名

任意一组字母、数字、\$符号和\_（下划线）符号的组合。

- 第一个字符必须是字母或下划线
- 总数不超1024
- 区分大小写

### 3. 关键字

保留字，具有特定和专有的语法作用

- 不能再对这些关键字做新的定义
- 只有小写的关键字才是保留字

## 4. 间隔符 起分隔文本的作用，使程序排列得更整齐更利于阅读。

保留字包括空格符（\b）、制表符（\t）、换行符（\n）及换页符

## 5. 操作符

**按操作数个数：**单目操作符、双目操作符和三目操作符；

按功能	{	连接与复制操作符	{ } { { } }	↑ 最高         最低
		一元操作符	! ~ &   ^	
		算数操作符	* / %	
			+ -	
		移位操作符	<< >>	
		关系操作符	< > <= >=	
		相等与全等操作符	== === != !==	
		位操作符	& ^	
		逻辑操作符	&&	
		条件操作符	?:	

## (1) 算数操作符

+（加法操作符），-（减法操作符），\*（乘法操作符），/（除法操作符），%（取模操作符）

8/3 //结果为2，整数除法截断任何小数部分

-10%3 //结果为-1，取模操作符求出与第一个操作数符号相同的余数

区分：有符号数 - 存储在线网、一般寄存器和基数形式表示的整数中  
无符号数 - 整数寄存器和十进制形式表示的整数中

```
wire [4:1] B; //定义了4位的wire型数据B
```

```
B=-4'd12; /*寄存器变量B只能存储无符号数，右端表达式的值为 110100（12的二进制补码），所以赋值后，B存储的十进制数为52*/
```

## (2) 位操作符      单目操作符、双目操作符

A=5'b11001, B=5'b10101;

A=~A; //A的值为5'b00110

A=A&B;                    //A的值为5'b10001

A=A | B; //A的值为5'b11101

(3) 规约操作符 对操作数逐位进行运算，运算结果是一位逻辑值

## &(归于与):

如果存在位值为0，为0；如果存在位值为x或z，为x；否则为1。

$\sim \&$  (归约与非), 与归于与 (&) 相反

| (归于或):

如果存在位值为1，为1；如果存在位值为x或z，则x；否则为0。

$\sim|$  (归约或非), 与归于或( $|$ )相反。

$\wedge$  (归于异或):

如果存在位值为x或z, x; 如果操作数中有偶数个1, 则为0;  
否则为1。

$\sim\wedge$  (归于异或非), 与归约异或( $\wedge$ )相反。

```
A=4'b1010;
```

```
B=&A; //B=0
```

```
B=~&A; //B=1
```

```
B=|A; //B=1
```

```
B=~|A; //B=0
```

```
B=^A; //B=0
```

```
B=~^A; //B=1
```

#### (4) 逻辑操作符

&& (逻辑与)

|| (逻辑或)

! (逻辑非)

#### (5) 关系操作符

比较结果为真，1；否则为0

> (大于)、< (小于)、>= (不小于) 和 <= (不大于)

#### (6) 相等与全等操作符

比较结果为真，1；否则为0

== (逻辑相等)  
!= (逻辑不等)

逻辑值做比较，值x和z具有通常的意义，如果两个操作数之一包含x或z，结果为未知的值 (x)

=== (全等)  
!== (非全等)

严格按位进行比较，把不定态 (x) 和高阻态 (z) 看做逻辑状态进行比较，比较结果不存在不定态，一定是1或0。

## (7) 位移操作符      逻辑移位，空闲位添0补位

<< (左移)、>> (右移)

两个操作数，右侧操作数表示的是左侧操作数所移动的位数

```
reg [1:8] A;           //定义了1个8位的reg型变量A
A=4'b0110;            //A的值为0000_0110
A=A>>2;               //A的值为0000_0001
```

可以支持部分指数操作，二进制 $A \times 2^3$       移位操作 $A \ll 3$

## (8) 连接和复制操作符

连接操作符是将多组信号用大括号括起来，拼接成一组新信号

`{expr1,expr2,...,exprN}`

复制操作通过指定的重复次数来执行操作

`{repetition_number{ expr1,expr2,...,exprN }}`



重复次数



## (9) 条件操作符                      唯一的三目操作符

根据条件表达式的值选择表达式

`cond_expr ? expr1 : expr2`

cond\_expr为真，则选择expr1；如果为假，则选择expr2

如果为x或z，则结果是将两个待选择的表达式进行计算，然后把两个计算结果按位进行运算得到最终结果

如果两个表达式的某一位都为1，则这一位的最终结果是1；如果都是0，则这一位的结果是0；否则结果为x。

**assign** `out=(sel==0) ? a : b;`

sel为0，则out=a；若sel为1，则out=b。

sel为x或z                      当a=b=0时，out=0  
                                    当a≠b时，out值不确定

## 6. 数据类型 表示数字电路硬件中的数据储存和传送元素

### (1) 常量

在程序运行的过程中，其值不能被改变

整型、实数型和字符串型

基本值的类型：0， 1， X， Z

整数型常量-整数型

表示方式

{ 简单十进制格式，有符号数；30， -26  
基数格式，〈位宽〉 ‘〈进制〉〈数字〉，无符号数

位宽—定义常量的二进制位数(长度)，为可选项

进制—二(b、B)，八(o、O)，十(d、D)或十六(h、H)进制

数字—所选进制的任何合法的值，可以包括 (x、X)和(z、Z)

8'b1011\_0011 //8位二进制数

64'hfe11 //64位二进制数

定义的长度 > 指定的长度 左边添0补位, 如果左边为X,Z, 用其补位。  
< 左边截掉

19

## 实数型常量-实数

表示方式 { 十进制格式, 数字和小数点组成(必须有小数点)  
指数格式, 由数字和字符e(E)组成

**26e-3** //表示0.026

## 字符串型常量-字符序列

### 双引号括起来的字符序列

字符串不能分成多行书写, 字符串中的特殊字符必须用 “\” 来说明

\n换行符;  
\t制表符;  
\\字符 “\” 本身;

转义符

19

## (2) 变量

在程序运行的过程中，其值可以改变

线网型、寄存器型和参数型

线网型变量-导线

具有多种类型

驱动方式 { 结构描述：连接到逻辑门或模块输出  
连续赋值：assign 赋值

类型

功能

**wire, tri**

对应于标准的互连线，可缺省。最常见

**supply1, supply0**

对应于电源线或接地线

**wor, trior**

对应于有多个驱动源的线或逻辑连接

**wand, triand**

对应于有多个驱动源的线与逻辑连接

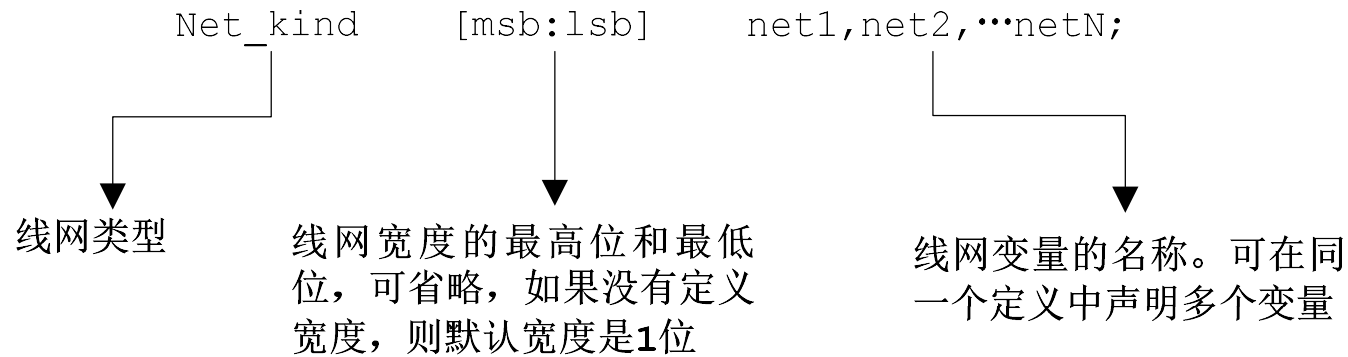
**triereg**

对应于有电容存在且能暂时存储电平的连接

**tri1, tri0**

对应于需要上拉或下拉的连接

## 语 法 格 式



注意：端口声明中input或inout型的端口，只能被定义为线网型

寄存器型变量-寄存器

保持信号不变

设计中必须将寄存器变量放在过程语句(如initial、always)

## 类型

## 功能

**reg**

可以选择不同的位宽，**最常用**

**integer**

有符号整数，32位，算术操作，可产生2的补码

**real**

有符号的浮点数，双精度，64位

**time**

无符号整数变量，64位

**realtime**

实数型时间寄存器

## 语法格式

```
reg [msb:lsb] mem1 [upper1:lower1], mem2 [upper2:lower2], ...;
```

寄存器最高位，最低位

寄存器数组最高位，最低位

```
reg [4:1] a[64:1] //a为64个4位寄存器的数组
```

```
integer integer1, integer2, ...integerN [msb:lsb];
```

参数型变量-无物理模型

其值为常量，增加可读可移植性

定义RAM的地址线位宽、数据线位宽，使用参数型变量

```
parameter ADDWIDTH=10, DATAWIDTH=8;
```

/\*定义了地址线的位宽为10，可对 $2^{10}$ 个存储单元进行寻址；定义了数据线的位宽为8，即每个存储单元有8位\*/

## 引用模块时可更改parameter型变量的值

【例1】

```
module ram(input1,input2,...,output1,output2,...);
parameter ADDWIDTH=10,DATAWIDTH=8; //地址线位宽为10，数据线位宽为8
...
endmodule

module top;
...
//模块top用不同的地址线和数据线位宽构建了两个不同的RAM模块
ram #(12) ram1(input1,input2,...,output1,output2,...);
    //地址线位宽为12，数据线位宽仍为8
ram #(20,4) ram2(input1,input2,...,output1,output2,...);
    //地址线位宽为20，数据线位宽为4
endmodule
```

## 使用defparam语句更改parameter型变量的值

【例2】

```
module modify_parameter;

defparam top.ram2.ADDWIDTH=16; //修改RAM2的地址线位宽为16

endmodule
```

## 数组变量-存储器模块RAM, ROM

一组寄存器型的变量可以定义为数组

```
reg [7:0] mem[15:0]; //定义了一个8*16位的存储器mem
```

```
integer mem[0:7] //定义了一个整数型数组，相当于一个32*8位的存储器
```

```
real mem[7:0] //定义了一个时间型数组，相当于一个64*8位的存储器
```

```
time mem[0:7] //定义了一个实型数组，相当于一个64*8位的存储器
```



## § 11.3 Verilog HDL的基本语句

### 1. 过程结构语句

initial; always

#### (1) initial 语句

只执行一次。无触发条件。

#### 语法格式

initial  
语句块

其中，语句块的格式为：

<块定义语句1>

时间控制1 行为语句1;

...

时间控制n 行为语句n;

<块定义语句2>

#### 【例3】

```
initial
begin
a='b000000; //初始时刻是0
#10 a='b011000;
#10 a='b011010;
#10 a='b011011;
#10 a='b010011;
#10 a='b001100;
end
```

延时10个时间单位后a赋予新值011000

常用于仿真中对激励信号或赋初值初始化

逻辑综合工具不综合

## (2) always 语句

具有触发条件

语法格式

时间控制为边沿触发或者电平触发

always @ <敏感信号表达式>

语句块

其中，语句块的格式为：

<块定义语句1>

时间控制1 行为语句1;

...

时间控制n 行为语句n;

<块定义语句2>

### 【例4】

```
reg[7:0] counter;  
reg tick;  
always @(posedge areg) /* 当  
areg信号的上升沿出现时把tick信号反相，  
并且把counter增加1*/  
begin  
tick=~tick;  
counter=counter+1;  
end
```

常用于对硬件功能模块的描述

可以综合成为寄存器组合逻辑

## 2. 语句块

由begin-end或fork-join界定的一组语句

### (1) 顺序语句块

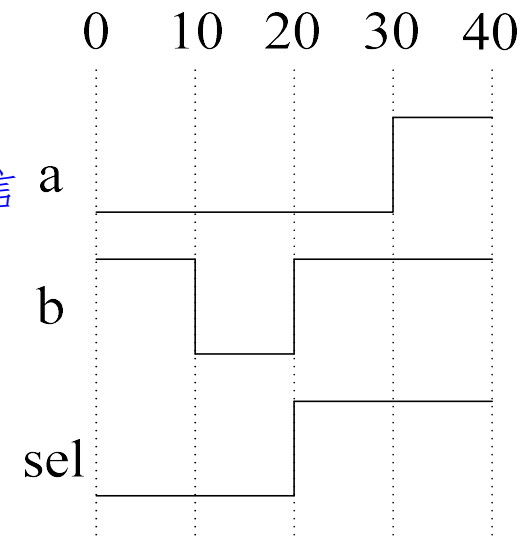
按给定顺序执行

语法格式

```
begin
  时间控制1 行为语句1;
  ...
  时间控制n 行为语句n;
end
```

#### 【例5】

```
begin
  a=0;b=1;sel=0;      /*加入激励信号,即产生输入a、b、sel*/
  #10 b=0;             // #10语句间延时
  #10 b=1;sel=1;
  #10 a=1;
end
```



## (2) 并行语句块

块内语句同时执行

### 语法格式

Fork

时间控制1 行为语句1;

...

时间控制n 行为语句n;

join

### 【例6】

**fork**

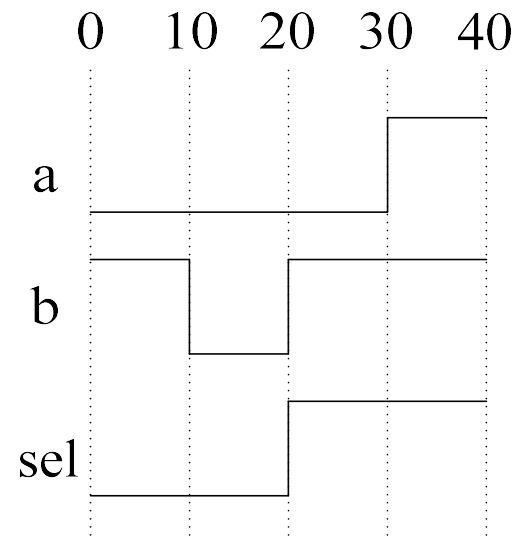
a=0;b=1;sel=0;

#10 b=0;

#20 b=1;sel=1;

#30 a=1;

**join**



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刻而言

### 3. 时序控制

用来对过程块中各条语句的执行时间（时序）进行控制

- (1) 延时控制      为行为语句的执行指定一个延迟时间

语法格式

#<延迟时间> 行为语句; 或      #<延迟时间>;

- (2) 事件控制      为行为语句的执行由触发事件触发

电平敏感事件触发

wait

语法格式

wait(条件表达式) 语句块;  
wait(条件表达式) 行为语句;  
wait(条件表达式);

wait (enable)  
    //当enable为高电平时执行加法  
sum=a+b;

## 边沿敏感事件触发

**语法格式**      指定信号的跳变沿才触发语句的执行

@(信号名)              //有变化时就触发事件

@(posedge信号名)    //有上升沿就触发事件

@(negedge信号名)    //有下降沿就触发事件

@(敏感事件1or敏感事件2or...) //敏感事件之一触发事件

通常与always语句联合使用，构建电路功能模块

## 4. 赋值语句

### (1) 连续赋值

**assign**

只能为线网型变量赋值

语法格式

**assign** net\_value=expression(表达式);

常量或由运算符（如逻辑运算符、算术运算符）参与的表达式

<b>wire</b> a,b;	//定义了两个1位的输入信号
<b>wire</b> out1,out2;	//定义两个1位的输出信号
<b>assign</b> out1=a&b;	//out1输出了a和b的与值
<b>assign</b> out2=a b;	//out2输出了a和b的或值

## (2) 过程赋值

最常见

只出现在initial(仿真)和  
always (主设计) 语句块内

只能给寄存  
器变量赋值

右端可以是  
任意表达式

阻塞赋值语句

“=”

不能被打断

先计算右侧表达式的值，然后赋值给等号左端目标

非阻塞赋值语句

“<=”

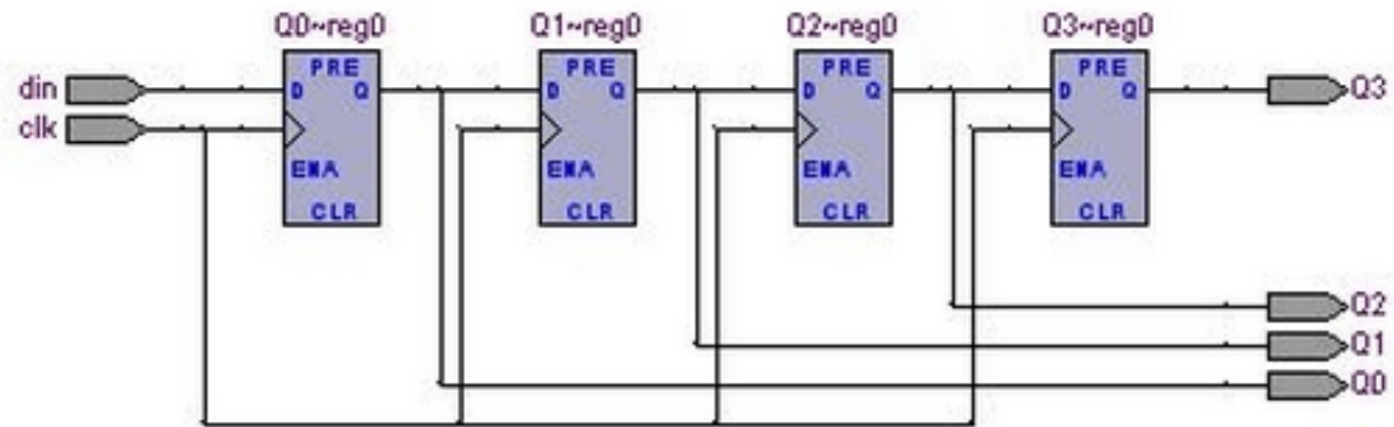
只能用于寄存器赋值

块结束之后才能真正完成赋值



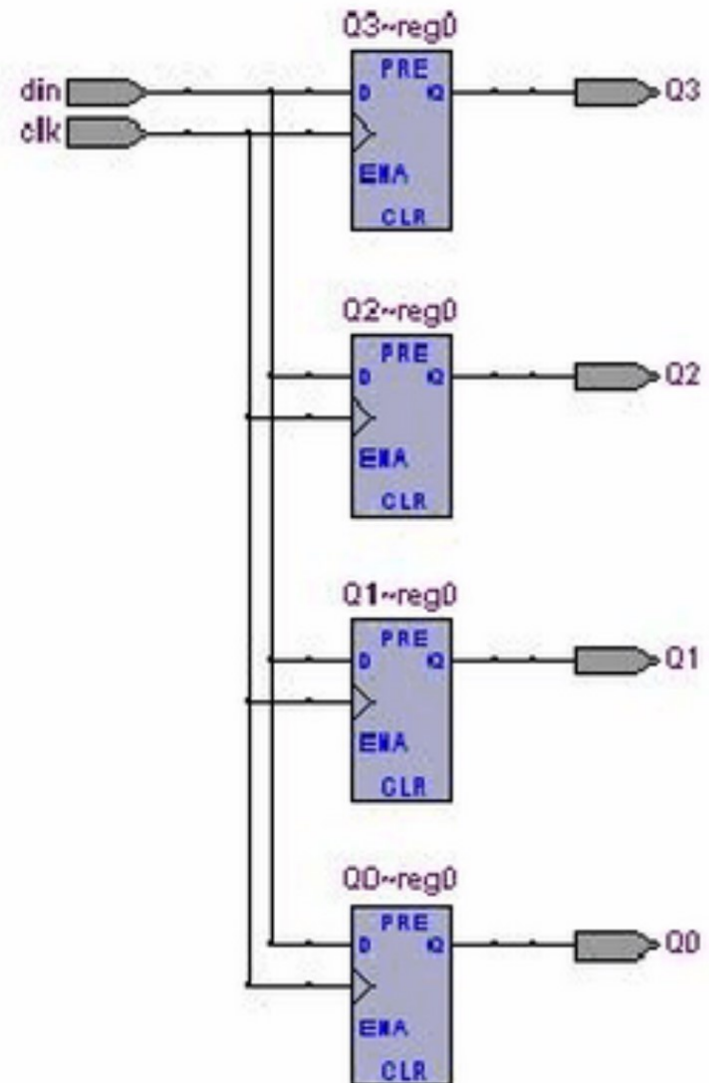
用阻塞赋值方式描述一个移位寄存器

```
module weisel(Q0,Q1,Q2,Q3,din,clk);  
    output Q0,Q1,Q2,Q3;  
    input clk,din;  
    reg Q0,Q1,Q2,Q3;  
    always @(posedge clk)  
    begin  
        Q3=Q2;  
        Q2=Q1;  
        Q1=Q0;  
        Q0=din;  
    end  
endmodule
```



将四条阻塞赋值语句的顺序完全颠倒的话，则综合器实际上等效成四个触发器

```
module wise1(Q0,Q1,Q2,Q3,din,clk);
    output Q0,Q1,Q2,Q3;
    input clk,din;
    reg Q0,Q1,Q2,Q3;
    always @(posedge clk)
        begin
            Q0=din;
            Q1=Q0;
            Q2=Q1;
            Q3=Q2;
        end
endmodule
```



## 5. 分支语句

(1) if-else语句      条件是否满足进行选择  
三种形式

**if(条件表达式) 块语句**

映射成锁存器

条件表达式为真时执行块语句，其他情况下（如为0、x、z）均为条件不成立

**if(条件表达式) 语句1  
else 语句2**

映射成二选一多路选择器

```
always@ (enable or a or b)
if (enable)    //enable=1时执行
out=a;
else           //enable≠1时执行
out=b;
```

等效语句    **assign out=(enable) ? a : b;**

```
if (条件表达式1) 块语句1
    else if (条件表达式2) 块语句2
    ...
    else if (条件表达式n) 块语句n
    else 块语句n+1
```

映射成多路选择器

用于多路选择控制，条件判断的先后顺序为优先级

(2) case语句 实现多路分支选择控制

用于微处理器指令译码功能的描述和有限状态机的描述

语法格式

```
case (敏感表达式)
    值1: 块语句1
    值2: 块语句2
    ...
    值n: 块语句n
    default: 块语句n+1 //缺省分支
endcase
```

第一个与条件表达式  
值相匹配的分支中的  
语句被执行，执行完  
这个分支后将跳出

case

## 6. 循环语句      只能在initial或always语句模块中使用。

### (1) forever语句

只能在initial块中

产生周期波形，用于仿真测试信号

语法格式      **forever** 语句;

或者      **forever**  
         **begin**  
         多条语句  
         **end**

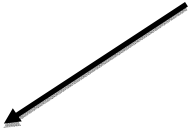
例:      **reg** clk;              //定义一个寄存器变量clk  
         **initial**  
         **begin**  
              clk=0;              //clk初始值为0  
              **forever**              /\*这种行为描述方式可以非常灵活地描述时钟，  
              可以控制时钟的开始时间及周期占空比，仿真效率也高\*/  
              **begin**  
                   #10 clk=1              //延时10个时间单位后clk值为1  
                   #5 clk=0              //延时5个时间单位后clk值为0  
              **end**  
         **end**

## (2) repeat 循环语句

将一条语句循环执行确定的次数

语法格式

通常为常量表达式



**repeat**(循环次数表达式) 语句;

或者

**repeat** (循环次数表达式)

**begin**

多条语句

**end**

### (3) while循环语句

条件满足时重复执行过程语

语法格式

**while** (条件表达式) 语句;

或者

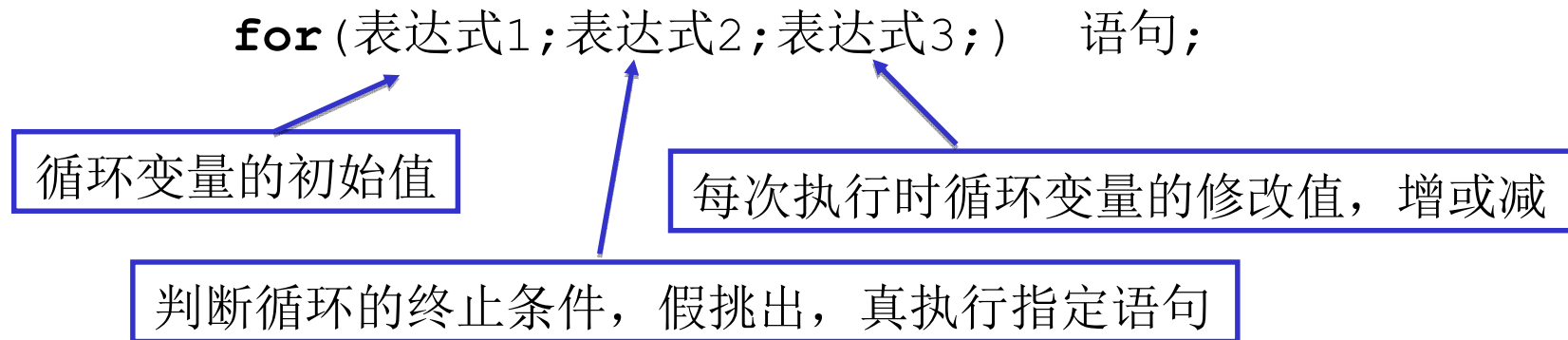
```
while (条件表达式)
begin
    多条语句
end
```

例：用while循环语句对count进行计数和输出。

```
initial
begin
    count=0;
    while (count<10) //当count值小于10时执行下面语句
    begin
        $display("count%d",count); //向屏幕输出count的值
        #10 count=count+1; //延时10个时间单位后，count值加1
    end
end
```

#### (4) for循环语句

##### 语法格式



##### 例：for循环语句示例

```
initial  
for (a=0;a<10;a=a+1)  
begin  
    $display("a%d",a); //向屏幕输出a的值  
End
```

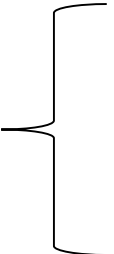
/\*a初值为0，当a<10时执行下面语句并自增1，循环判断，直至跳出循环\*/



## § 11.4 Verilog HDL程序设计示例

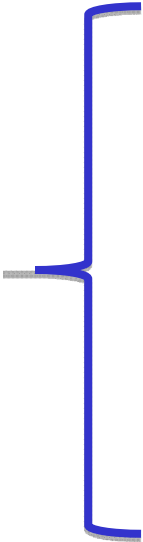
### 1. 基本逻辑门电路设计

电路描述的基本方式



- 数据流描述方式;
- 行为描述方式;
- 结构化描述;
- 混合方式;

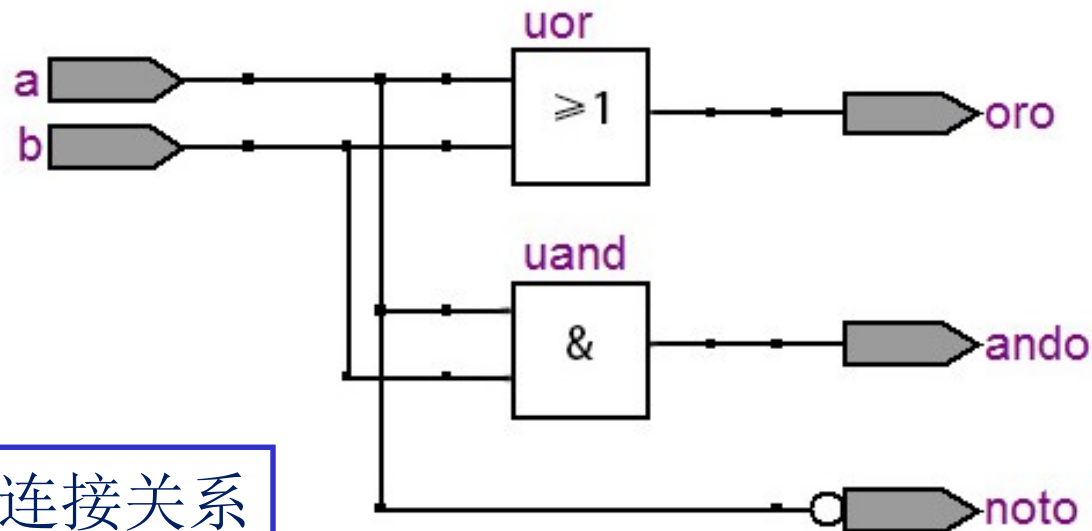
Verilog HDL中提供下列内置基本门与器件

- 
- a) 多输入门: and, nand, or, nor, xor, xnor;
  - b) 多输出门: buf, not;
  - c) 三态门: bufif0, bufif1, notif0, notif1;
  - d) 上拉、下拉电阻: pullup, pulldown;
  - e) MOS开关: cmos, nmos, pmos, rcmos, rnmos, rpmos;
  - f) 双向开关: tran, tranif0, tranif1, rtran, rtranif0, rtranif1

## 基本门电路

## 结构化描述方式

强调描述连接关系



```
module basegate (a, b, noto, ando, oro); //模块名，端口列表
    input a; //输入端口声明
    input b;
    output ando; //输出端口声明
    output noto;
    output oro;
    and uand (ando, a, b); //调用Verilog HDL语言原语and、not、or
    not unot (noto, a);
    or uor (oro, a, b);
endmodule //模块结束语句
```

## 数据流描述方式

### 强调数据赋值顺序

```
module basegate (a, b, noto, ando, oro);  
    input a; input b;  
    output ando; output noto;  
    output oro;  
    //采用 assign 语句数据流描述方式  
    assign ando = a & b; //连续赋值语句  
    assign noto = ~a;  
    assign oro = a | b;  
endmodule
```

## 行为描述方式

强调always过程中的连续赋值，实际也就是行为

```
module basegate (a, b, noto, ando, oro);  
    input a;  
    input b;  
    output ando;  
    output noto;  
    output oro;  
    reg ando, noto, oro; //在always语句中被赋值对象应声明为reg型  
    always @(a or b) //always过程连续赋值  
    begin  
        ando <= a&b;  
        noto <=~a;  
        oro <=a | b;  
    end  
endmodule
```

## 2. 组合电路设计

### (1) 1位半加器

$$sum = a \oplus b = (a + b) \overline{a} \overline{b}$$

$$cout = ab = \overline{\overline{a} \overline{b}}$$

```
module adder (cout, sum, a, b);
```

```
output      cout;
```

```
output      sum;
```

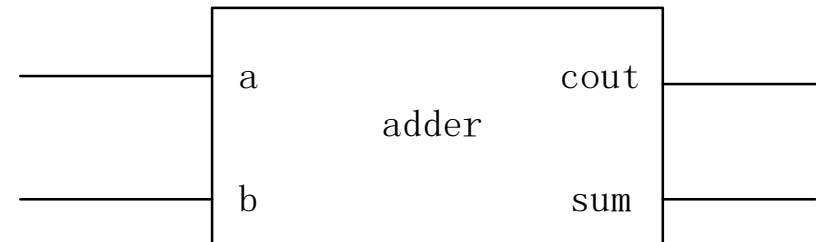
```
input  a, b;
```

```
wire      cout, sum;           //wire变量声明
```

```
assign { cout, sum } =a+b;     //数据流语句 a+b
```

相加

```
endmodule
```



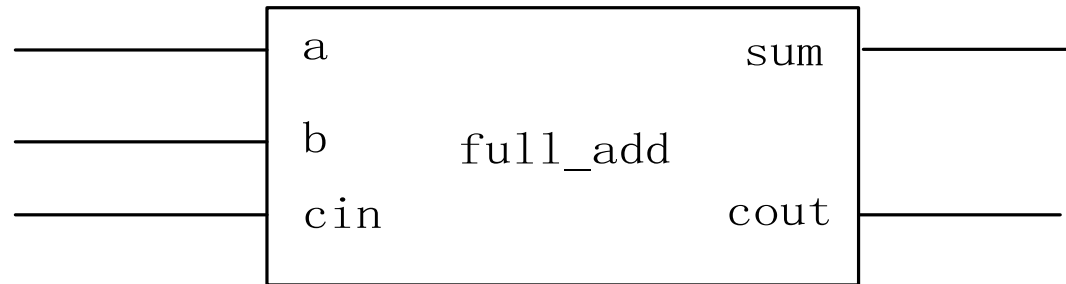
真值表

a	b	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(2) 1位全加器

$$\text{Sum} = a \oplus b \oplus \text{cin}$$

$$\text{count} = ab + b\text{cin} + a\text{cin} = ab + (a \oplus b)\text{cin}$$



```
module full_add (a, b, cin, sum,  
cout);
```

```
    input a, b, cin;
```

```
    output sum, cout;
```

```
    reg sum, cout;
```

```
    reg m1, m2, m3;
```

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

```
        begin
```

```
            sum = ( a^b ) ^ cin;
```

```
            m1=a&b;
```

```
            m2=b&cin;
```

```
            m3=a&cin;
```

```
            cout= ( m1|m2 ) | m3;
```

```
        end
```

```
endmodule
```

//always过程连续赋值

always语句中有一个与事件控制  
(begin-end对)。这意味着只要  
a、b或cin上发生事件，即a、b或  
cin任一值发生变化，顺序过程就  
执行。

#### (4) 3-8译码器74LS138

输入端a, b, c和8个译码输出端Y0~Y7, 附加控制输入端e1、e2和e3

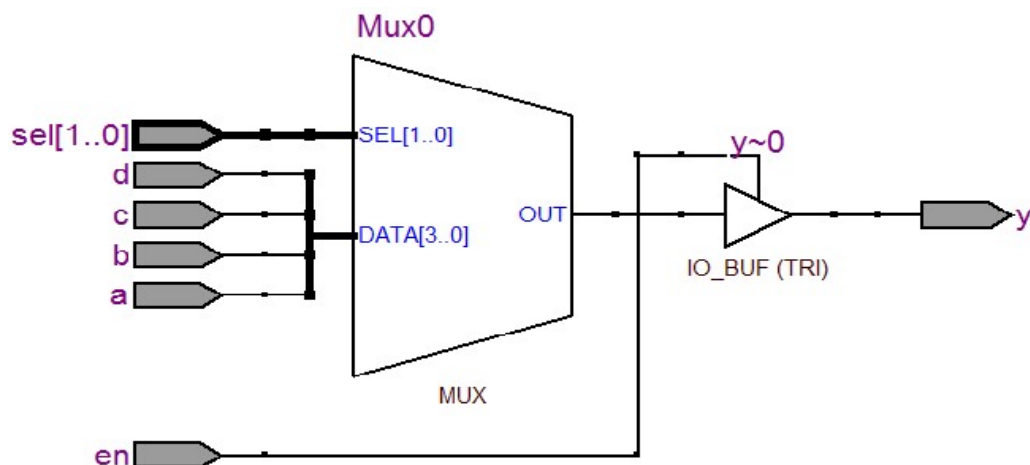
当[e1 e2 e3]=100时, 工作状态

```
module decoder3_8 (Y, a, b, c, e1, e2, e3);
output [7:0] Y;
input a, b, c; input e1, e2, e3;
reg [7:0] Y;
always @ (a or b or c or e1 or e2 or e3) //always过程连续赋值
begin
    if ((e1==1) & (e2==0) & (e3==0)) //判断译码器是否处于工作状态
    begin
        case ({c, b, a}) //将c、b、a用位拼接运算符拼接成3位信号
            3'd0:Y=8'b1111_1110;
            3'd1:Y=8'b1111_1101;
            3'd2:Y=8'b1111_1011;
            3'd3:Y=8'b1111_0111;
            .....
            3'd7:Y=8'b0111_1111;
            default:Y =8'bX;
        endcase
    end
    else
        Y=8'b1111_1111; //译码器是否处于非工作状态
    end
endmodule
```

## (5) 数据选择器

根据选择信号，决定哪路输入信号送到输出信号

```
module dataselector (a, b, c, d, sel, en, y);  
    input a;           //输入端口声明  
    input b;  
    input c;  
    input d;  
    input [1:0] sel;  
    input en;  
    output y;          //输出端口声明  
    reg y;             //在always语句中被赋值的信号要声明为reg类型  
    always @ (a or b or c or d or sel or en) //always过程连续赋值  
    begin  
        if (1'b1 == en) //判断使能信号en是否有效  
        begin  
            case (sel) //在case语句中实现数据选择过程  
                2'b00: y <= a;  
                2'b01: y <= b;  
                2'b10: y <= c;  
                2'b11: y <= d;  
                default: y <= 1'bz;  
            endcase  
        end  
        else  
            y <= 1'bz; //使能信号为低无效时，输出y为高阻  
        end  
    end  
endmodule
```

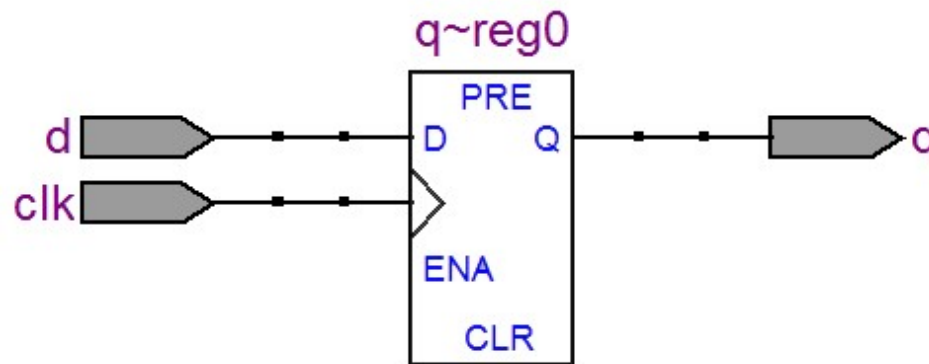




### 3. 时序逻辑电路设计

#### (1) D触发器的设计

```
module dtrigger (clk, d, q);  
    input clk;    //时钟信号  
    input d;      //D触发器输入信号  
    output q;     //D触发器输出信号  
    reg q;        //在always语句中被赋值的信号要声明为reg类型  
    always @(posedge clk) //clk上升沿语句，下降沿为negedge clk  
    begin  
        q <= d;  
    end  
endmodule
```

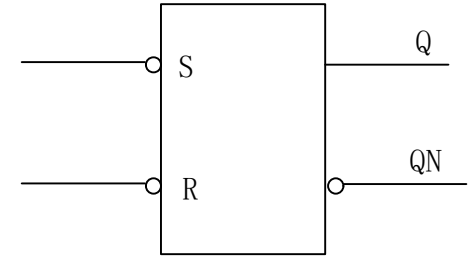
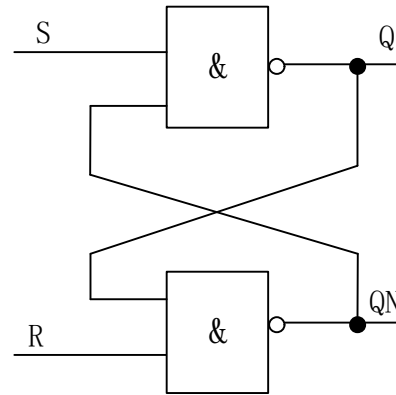


综合后生成的寄存器传输级结构

## (2) RS触发器

### A. 采用结构描述的基本RS触发器

```
module RSff1 (R, S, Q, QN);  
    input R, S;  
    output Q, QN;  
    nand U1 (Q, S, QN),  
           U2 (QN, R, Q);  
endmodule
```



### B. 采用行为描述的基本RS触发器

```
module RSff2 (R, S, Q, QN);  
    input R, S;  
    output Q, QN;  
    reg Q, QN;  
    always@ (R or S)           //R、S的各种组合  
        if( {R,S}==2'b01)      begin Q<=0;QN<=1;end  
        else if ( {R,S}==2'b10) begin Q<=1;QN<=0;end  
        else if ( {R,S}==2'b11) begin Q<=Q;QN<=QN;end  
        else                    begin Q<=1'bX;QN <=1'bX;end  
endmodule
```

### (3) 4位计数器

```
module counter (qout, reset, clk);
```

```
    output [3:0] qout;
```

```
    input clk, reset;
```

```
    reg [3:0] qout;
```

```
    always @ (posedge clk)
```

```
        begin
```

```
            if (reset) qout<=0;
```

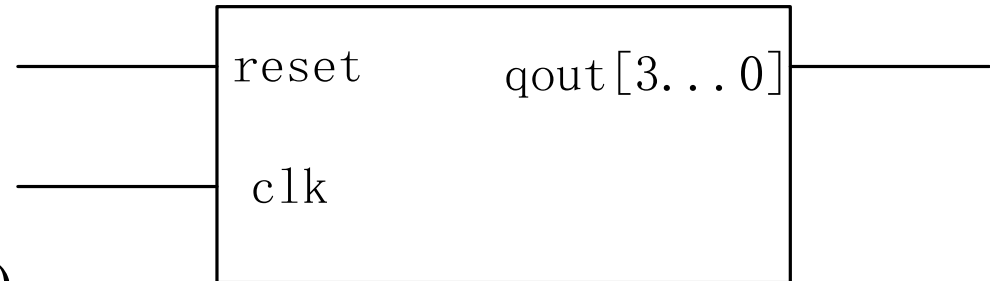
//reset有效 qout=0

```
            else      qout<=qout+1;
```

//自动加1

```
        end
```

```
endmodule
```



#### (4) 60进制同步计数器

```
module count60 (clk, rst, qh, ql, cout);
    input clk, rst;
    output [3:0] ql;
    output [2:0] qh;
    output cout;
    reg [3:0] ql;
    reg [2:0] qh;
    always @(posedge clk or posedge rst)
        begin
            if (!rst) ql<=0;
            else if (ql==9)
                begin
                    if (qh==5) {qh,ql}<=7'd0;
                    else
                        begin
                            ql<=0;
                            qh<=qh+1;
                        end
                    end
                end
            else ql<=ql+1;
        end
    assign cout=(qh==5 && ql==9)?1:0;
endmodule
```

## (5) 移位寄存器设计

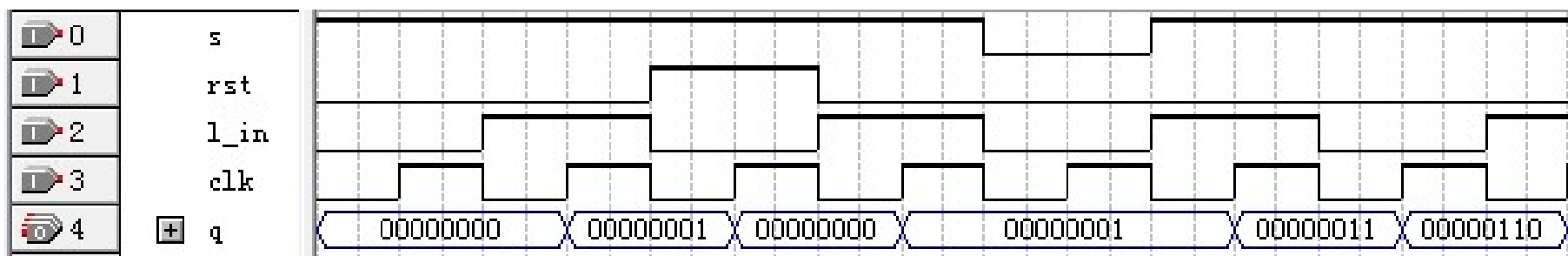
存储、移位功能

左移、右移和双向移位寄存器

### 8位左移移位寄存器

```
module shiftleft_reg (clk, rst, l_in, s, q);  
    input clk, rst, l_in, s;  
    output [7:0] q;  
    reg [7:0] q;    //在always语句中被赋值的信号要声明为reg类型  
    always @(posedge clk)  
    begin  
        if (rst)  
            q<=8'b0;  
        else if (s)  
            q<={q[6:0],l_in};  
        else  
            q<=q;  
        end  
    endmodule
```

8位左移移位寄存器是通过将l\_in放在q[6:0]的右边并置成一个8位向量，将其锁存在q[7:0]来实现的。



8位左移移位寄存器的功能仿真波形图

输入l\_in的变化，每当时钟上升沿到来后，并且

当控制端s=1时：

输出将输入的数据放在输出的最低位上，实现左移的功能；

当控制端s=0时：

寄存器保持

# 小 结

## ■ Verilog HDL程序的基本结构

- 端口定义、端口类型说明、数据类型说明、逻辑功能描述

## ■ Verilog HDL的数据类型

- 标识符、关键字、常用操作符
- 数据类型（**wire**、**reg**）

## ■ Verilog HDL语言的基本语句

- 过程结构语句（**initial**, **always**）
- 语句块（**begin...end**, **fork...join**）
- 赋值语句

作业:

**11.2,      11.3,      11.4,**

**11.5,      11.8,      11.9,**

**11.12      11.13      11.15.**