

6 Verilog 硬件描述语言

- 6.1 硬件描述语言简介
- 6.2 Verilog HDL与C语言
- 6.3 Verilog 的数据类型
- 6.4 Verilog运算符及优先级
- 6.5 Verilog 模块结构
- 6.6 Verilog 设计层次与风格
- 6.7 Verilog 行为语句
- 6.8 Verilog 有限状态机设计



6.1 硬件描述语言简介

硬件描述语言HDL (Hardware Description Language)：一种编程语言，用软件方法对硬件的结构和运行进行建模。它是对硬件电路及其执行过程的描述，所以程序设计过程也叫电路建模过程。

HDL 的优势

与原理图设计方法相比：

- (1) **HDL**设计电路能获得非常抽象级的描述。设计者在电路设计时不必考虑工艺实现的具体细节。
- (2) **HDL**描述电路设计，在设计的前期就可以完成电路功能级的验证。
- (3) 用**HDL**设计电路类似于计算机编程。带有注解的文字性描述更有利于电路的开发与调试。

6.2 Verilog语言与C语言

常用的关键字

C 语言	Verilog HDL 语言
function	module,function,task
if-then-else	if-then-else
for	for
while	while
case	case
break	break
define	define
printf	printf
int	int
{...}	begin...end

运算符

C 语言	Verilog HDL 语言	功能
+	+	加
-	-	减
*	*	乘
/	/	除
%	%	取模
!	!	逻辑取反
&&	&&	逻辑与
		逻辑或
>	>	大于
<	<	小于
>=	>=	大于等于
<=	<=	小于等于
==	==	等于
!=	!=	不等于
~	~	位反相
&	&	按位逻辑与
		按位逻辑或
^	^	按位逻辑异或
~^	~^	按位逻辑同或
>>	>>	右移
<<	<<	左移
?:	?:	同等于 if-else 叙述

本质区别在于“运行”方式：

C 程序是一行接一行依次执行的，属于顺序结构，串行执行。而Verilog 描述的硬件是可以在同一时间同时运行的，属于**并行结构，并发执行**。

Verilog HDL所设计的硬件电路的所有单元都是并行工作的。一旦设备电源开启，硬件的每个单元就会一直处于运行状态。



6.3 Verilog的数据类型

常量

在程序运行值不能被改变的量称为常量。常量分为整型、实数型和字符型三类。

整型常量即整数，可以是二进制（b或B）、十进制（d或D）、十六进制（h或H）与八进制（o或O）

格式形式： <位宽>'<进制><数字>

例：8'ha2, 8HA2 字母不区分大小写

4'd2----4位十进制数

8'b1x10110z

x表示不定值，z表示高阻值。



12		unsized decimal (zero-extended to 32 bits)
'H83a		unsized hexadecimal (zero- extended to 32 bits)
8'b1100_ 0001	8-bit binary	
64'hff01	64-bit hexadecimal (zero- extended to 64 bits)	
9'o17	9-bit octal	
32'bz01x	Z-extended to 32 bits	
3'b1010_ 1101	3-bit number, truncated to 3'b101	
6.3	decimal notation	
32e- 4	scientific notation for 0.0032	
4.1E3	scientific notation for 4100	
-8' d72	负数要在位宽前加一个减号	

- 数字中 () 忽略, 便于查看
- 没有定义大小 (**size**) 整数缺省为32位
- 缺省数基为十进制
- 当数值 **value** 大于指定的大小时, 截去高位。如 2'b1101表示的是2'b01
- 实数可用科学表示法或十进制表示

<尾数><e或E><指数>, 表示: 尾数 $\times 10^{\text{指数}}$



- **parameter**定义常量，即用parameter定义一个标志符，代表一个常量，称为符号常量。表示**运行时的常数(run-time constants)**

例：

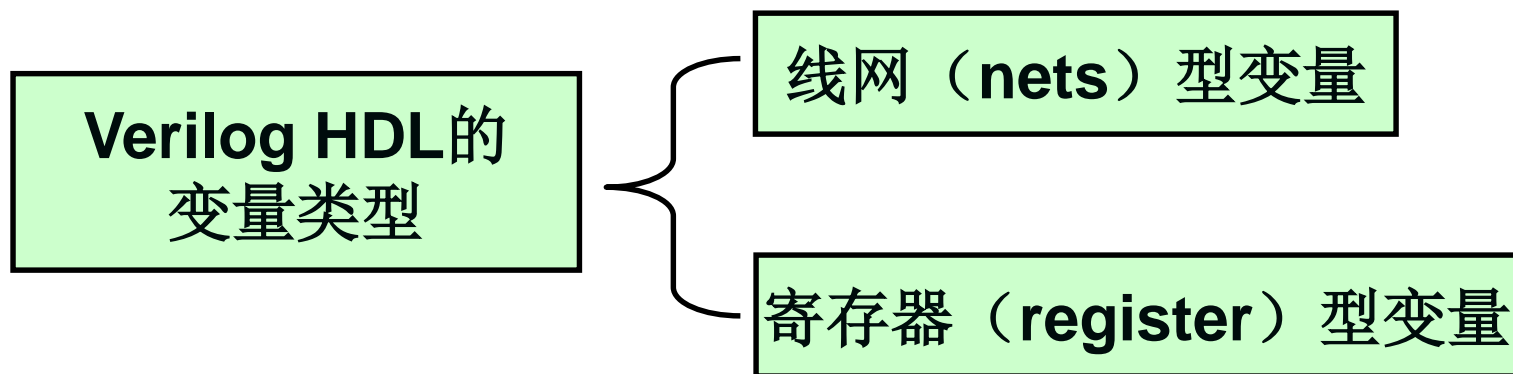
```
parameter r=5.7;           //声明r为一个实数型参数
parameter byte_size=8, byte_msb=byte_size-1;
parameter average_delay=(r+f) / 2; //用常数表达式赋值
```

```
module mod1( out, in1, in2);
    . . .
    parameter p1 = 8,
                x_ word = 16'bx,
                . . .
    wire [p1: 0] w1; // A wire
    declaration using
    . . .
endmodule
```



变量

值可以改变的量。用来表示数字电路中的物理连线、数据存储和传输单元等物理量，并占据一定存储空间，在该存储空间内存放变量的值。



线网型变量: 指输出根据输入的变化而更新的变量，指硬件电路中的物理连线。常用的连线型变量有**wire**型（连线型）和**tri**型（三态型）。

```
<net_type> [range] [delay] <net_name>[,  
net_name];
```

net_type: net类型

range: 矢量范围，以[MSB: LSB]格式

delay: 定义与net相关的延时

net_name: net名称，一次可定义多个net，用逗号分开。

wire a; //定义了一个1位的wire型数据，

wire [3:0] c, d; //定义了二个4位的wire型数据c和d

tri [15: 0] busa; // 16位三态总线

wire [0: 31] w1, w2; // 两个32位wire，MSB为bit0



寄存器型变量: 表示一个抽象的数据存储单元，可以通过赋值语句改变寄存器内存储的值。寄存器型变量对应的是具有状态保持作用的电路元件，如触发器或寄存器。

寄存器声明

```
<reg_type> [range] <reg_name>[, reg_name];
```

reg_type: 寄存器类型

range: 矢量范围，以[MSB: LSB]格式。只对reg类型有效

reg_name : 寄存器名称，一次可定义多个寄存器，用逗号分开

```
reg [3: 0] regb; //定义了一个4位的名为regb的reg型数据
```

```
reg [4: 1] regc, regd; //定义了二个4位的名为regc和regd的reg型数据
```

对寄存器的赋值:

```
reg [ 1:5 ] dig; //dig为一个5位寄存器
```

```
dig=5'b11011; //可以在一条赋值语句中完成对寄存器的赋值
```



reg和wire类型的区别和用法:

wire型为直通，即只要输入有变化，马上无条件地反映。

reg型一定要有触发，输出才会反映输入。

寄存器型数据保持最后一次的赋值，而线型数据需要持续的驱动。

wire只能被assign连续赋值，reg只能在过程块always中赋值。



信号类型确定方法:

- 信号可以分为端口信号和内部信号。出现在端口列表中的信号是端口信号，其它的信号为内部信号。
- 对于端口信号，输入端口只能是net类型。输出端口可以是net类型，也可以是register类型。若输出端口在过程块中赋值则为register类型；若在过程块外赋值(包括实例化语句)，则为net类型。
- 内部信号类型与输出端口相同，可以是net或register类型。判断方法也与输出端口相同。若在过程块中赋值，则为register类型；若在过程块外赋值，则为net类型。



选择数据类型时常犯的错误举例

example.v

修改前:

```
module example(o1, o2, a, b, c, d);  
    input a, b, c, d;  
    output o1, o2;  
    reg c, d;  
    reg o2  
    and u1(o2, c, d);  
    always @(a or b)  
        if (a) o1 = b; else o1 = 0;  
endmodule
```

修改后:

```
module example(o1, o2, a, b, c, d);  
    input a, b, c, d;  
    output o1, o2;  
    // reg c, d;  
    // reg o2  
    reg o1;  
    and u1(o2, c, d);  
    always @(a or b)  
        if (a) o1 = b; else o1 = 0;  
endmodule
```

寄存器数组(REGISTER ARRAYS)

- 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
```

- 描述存储器时可以使用参数或任何合法表达式

```
parameter wordsize = 16;
```

```
parameter memsize = 1024;
```

```
reg [wordsize-1: 0] MEM3 [memsize-1: 0];
```



文字规则—标识符(IDENTIFIERS)

- 标识符是用户在描述时给Verilog对象起的名字
- 标识符必须以字母(a-z, A-Z)或(_)开头, 后面可以是字母、数字、(\$)或(_)。
- 最长可以是1023个字符
- 标识符区分大小写, sel和SEL是不同的标识符
- 模块、端口和实例的名字都是标识符

```
module MUX2_1(out, a, b, sel);  
output out;  
input a, b, sel;  
    not not1 (sel_, sel);  
    and and1 (a1, a, sel_);  
    and and2 (b1, b, sel);  
    or  or1  (out, a1, b1);  
endmodule
```

Verilog标识符



6.4 Verilog的运算符及优先级

按其功能可分为以下九类：

- (1)算术运算符(+, -, X, / , %);
- (2)赋值运算符(=, <=);
- (3)关系运算符(>, <, >=, <=);
- (4)逻辑运算符(&&, ||, !逻辑非);
- (5)条件运算符(?:);
- (6)位运算符(~ , |, &, ^异或, ^~同或);
- (7)移位运算符(<<, >>);
- (8)拼接运算符({ }将两个或多个信号的某些位拼接起来);
- (9)其他（等号运算符、缩减运算符）。



运算符所带的操作数可不同，按**所带操作数的个数**运算符可分为3种：

(1)单目运算符：可以带一个操作数，操作数放在运算符的右边。

如 `clock = ~clock;` //~是一个单目取反运算符，`clock`是操作数

(2)双目运算符：可带两个操作数，操作数放在运算符的两边。

如 `c = a|b;` //是一个二目按位或运算符，`a`和`b`是操作数

(3)三目运算符（条件运算符？：）

格式： 信号 = 条件 ? 表达式1 : 表达式2;

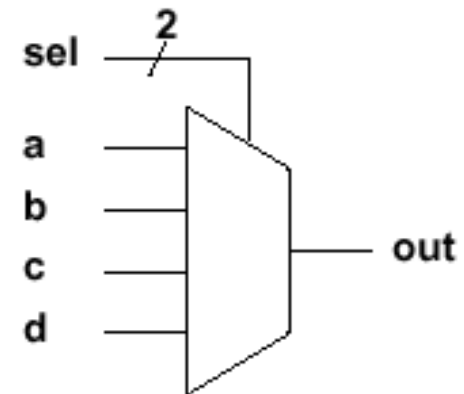
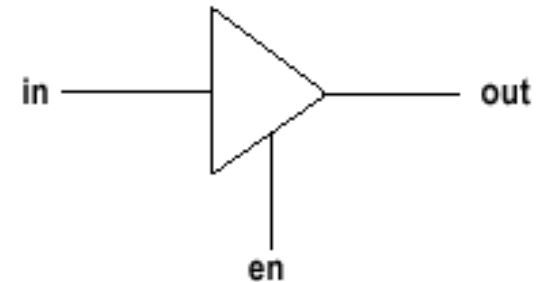
当条件成立时，信号取表达式1的值,反之取表达式2的值.

例如： `assign out=(sel==0)?a:b;` //持续赋值，如果`sel`为0，则`out=a`;
 否则`out=b`



问题： 以下Verilog模块实现什么逻辑功能？

```
module likebufif( in, en, out);  
    input in;  
    input en;  
    output out;  
    assign out = (en == 1) ? in : 'bz;  
endmodule  
  
module like4to1( a, b, c, d, sel, out);  
    input a, b, c, d;  
    input [1: 0] sel;  
    output out;  
    assign out = sel == 2'b00 ? a :  
                  sel == 2'b01 ? b :  
                  sel == 2'b10 ? c : d;  
endmodule
```



按位操作符

~	not
&	and
	or
^	xor
~ ^	xnor
^ ~	xnor

- 按位操作符对矢量中相对应位运算。

```
regb = 4'b1 0 1 0
```

```
regc = 4'b1 x 1 0
```

```
num = regb & regc = 1 0 1 0 ;
```

- 位值为x时不一定产生x结果。

当两个操作数位数不同时，位数少的操作数零扩展到相同位数。

```
a = 4'b1011;
```

```
b = 8'b01010011;
```

```
c = a | b; // a零扩展为 8'b00001011
```

```
module bitwise ();  
    reg [3: 0] rega, regb, regc;  
    reg [3: 0] num;  
    initial begin                //以下3句顺序  
        执行  
        rega = 4'b1001;  
        regb = 4'b1010;  
        regc = 4'b11x0;  
    end  
    initial fork                  // num = 0000  
        #10 num = rega &         // num = 1000  
        #20 num = rega &         // num = 1011  
        #30 num = rega |         // num = 10x0  
        #40 num = regb &         // num = 1110  
        #50 num = regb | regc;  
        #60 $finish;  
    join  
endmodule
```


逻辑操作符

!	not
&&	and
	or

- 逻辑操作符的结果为一位1, 0或x。
- 逻辑操作符只对逻辑值运算。
- 如操作数为全0, 则其逻辑值为false
- 如操作数有一位为1, 则其逻辑值为true
- 若操作数只包含0、x、z, 则逻辑值为x

逻辑反操作符将操作数的逻辑值取反。例如, 若操作数为全0, 则其逻辑值为0, 逻辑反操作值为1。

```
module logical ();
    parameter five = 5;
    reg ans;
    reg [3: 0] rega, regb, regc;
    initial
        begin
            rega = 4'b0011; //逻辑值为“1”
            regb = 4'b10xz; //逻辑值为“1”
            regc = 4'b0z0x; //逻辑值为“x”
        end
    initial fork
        #10 ans = rega && 0;      // ans = 0
        #20 ans = rega || 0;      // ans = 1
        #30 ans = rega && five;    // ans = 1
        #40 ans = regb && rega;    // ans = 1
        #50 ans = regc || 0;      // ans = x
        #60 $finish;
    join
endmodule
```



逻辑反与位反的对比

! logical not 逻辑反

~ bit-wise not 位反


- 逻辑反的结果为一位1, 0或x。
- 位反的结果与操作数的位数相同

逻辑反操作符将操作数的逻辑值取反。例如，若操作数为全0，则其逻辑值为0，逻辑反操作值为1。

```
module negation();  
    reg [3: 0] rega, regb;  
    reg [3: 0] bit;  
    reg log;  
    initial begin  
        rega = 4'b1011;  
        regb = 4'b0000;  
    end  
    initial fork  
        #10 bit = ~rega;    // num = 0100  
        #20 bit = ~regb;    // num = 1111  
        #30 log = !rega;    // num = 0  
        #40 log = !regb;    // num = 1  
        #50 $finish;  
    join  
endmodule
```

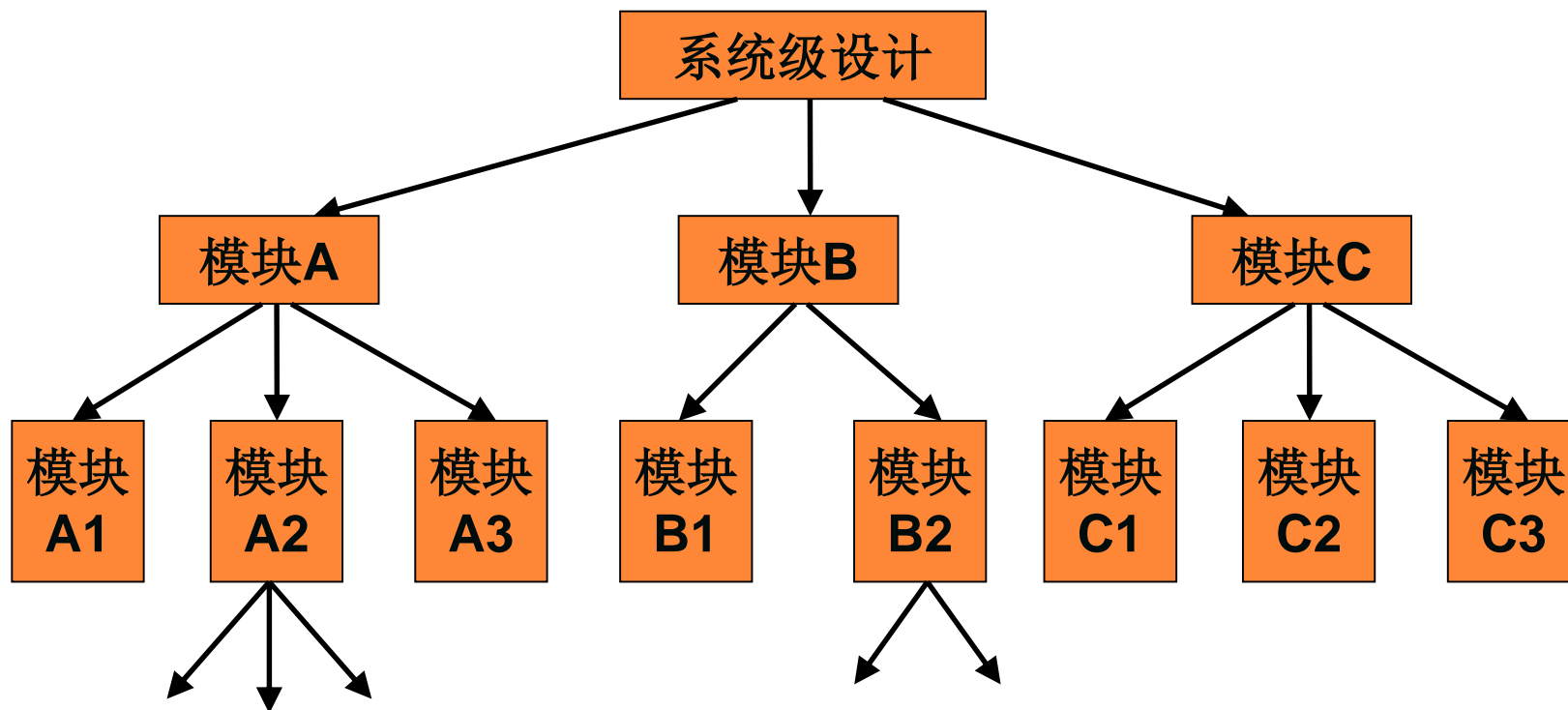


运算符的优先级

运算类型	运算符	优先级
单目运算	$+$, $-$, $!$, \sim	高优先级  低优先级
乘、除、取模	$*$, $/$, $\%$	
双目运算（加、减）	$+$, $-$	
移位	\ll , \gg	
关系	$<$, $<=$, $>$, $>=$	
等价	$==$, $!=$, $===$, $!==$	
按位与、单目运算（与、与非）	$\&$, $\sim\&$	
单目或双目运算（异或、同或）	\wedge , $\wedge\sim$	
按位或、单目运算（或、或非）	$ $, $\sim $	
逻辑与	$\&\&$	
逻辑或	$ $	
条件	$?:$	

6.5 Verilog模块的结构

Verilog HDL的设计方法：自上向下（Top-down）



搭积木，每个模块都是一个积木块，不同的积木块有不同的功能要求,制作每个积木块的过程就是模块编程，最后把积木块（模块）搭在一起完成整个系统设计。

“模块”（module）：Verilog程序的基本设计单元

1、模块声明：包括模块名字，模块输入、输出端口列表。格式：

module 模块名(端口1, 端口2,); //模块开始
endmodule //模块结束

2、端口定义：是模块与外界连接和通信的信号线。定义类型有3种。

3、信号类型声明：对模块中用到的所有信号（包括端口信号、节点信号等）进行数据类型的定义。

可将端口声明和信号类型声明放在一条语句中，如：

```
output reg cout; //cout为输出端口，其数据类型为reg型  
output wire[7:0] A; //A为输出端口，其数据类型为8位wire型
```

4、逻辑功能描述：最重要部分。用assign, always, 调用元件（元件例化，如同在C语言下调用库函数一样）等方法来描述逻辑功能。

```
module 模块名（端口列表）；
```

端口定义

```
input 输入端口
```

```
output 输出端口
```

```
inout 输入/输出端口
```

数据类型说明：

```
wire 连线型
```

```
reg 寄存器型
```

```
parameter 参数型
```

逻辑功能描述：

```
assign
```

```
always
```

```
function
```

```
task
```

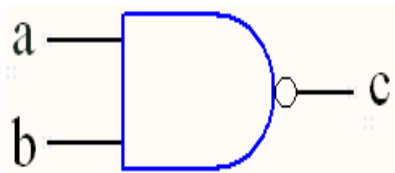
```
.....
```

```
endmodule
```

Verilog模块的基本结构

逻辑功能描述

1) 用**assign持续赋值语句**描述组合逻辑。assign语句一般给wire型数据信号赋值。这种描述方法的句法很简单，只需在assign后加上一个逻辑方程式即可。

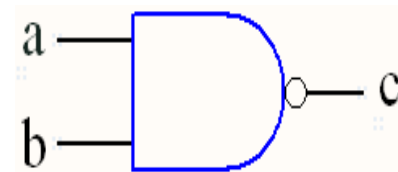


<pre>module not2_inst(a,b,c);</pre>	//模块名为not2_inst，端口列表a,b,c
<pre>input a,b;</pre>	//模块的输入端口为a,b
<pre>output c;</pre>	//模块的输出端口为c
<pre>wire a,b,c;</pre>	//定义信号的数据类型
<pre>assign c=~(a&b);</pre>	//逻辑功能描述
<pre>endmodule</pre>	//模块结束



2) 用**always**过程块描述。

```
module not2_inst(      //模块名为not2_inst,
    input a,b,          //模块的输入端口为a,b, 信号类型默认为wire型
    output reg c         //always过程块中的被赋值变量必须是reg型
);
    always @(a or b)     //always过程块及敏感信号列表
    begin                //always过程块开始
        c=~(a&b);        //逻辑功能描述
    end                  //always过程块结束
endmodule               //模块结束
```



两种方法综合后得到的电路完全相同。因此，相同的电路可采用不同的描述方法。

但是，**always**过程语句虽可描述组合逻辑电路，但更多的是用于描述时序逻辑电路。

always过程赋值语句用于对寄存器类型(reg)的变量进行赋值。

(1) 非阻塞(non_blocking)赋值方式

非阻塞赋值使用“<=”语句，如 $b \leq a$;

非阻塞赋值在块结束时才完成赋值操作，即b的值并不是立刻就改变的。

(2) 阻塞(blocking)赋值方式

阻塞赋值使用“=”语句，如 $b = a$;

阻塞赋值在该语句结束时就完成赋值操作，即b的值在该赋值语句结束后立刻改变。

注千万不要将阻塞赋值和非阻塞赋值与**assign**赋值语句混淆起来，**assign**赋值语句根本不允许出现在**always**语句块中。

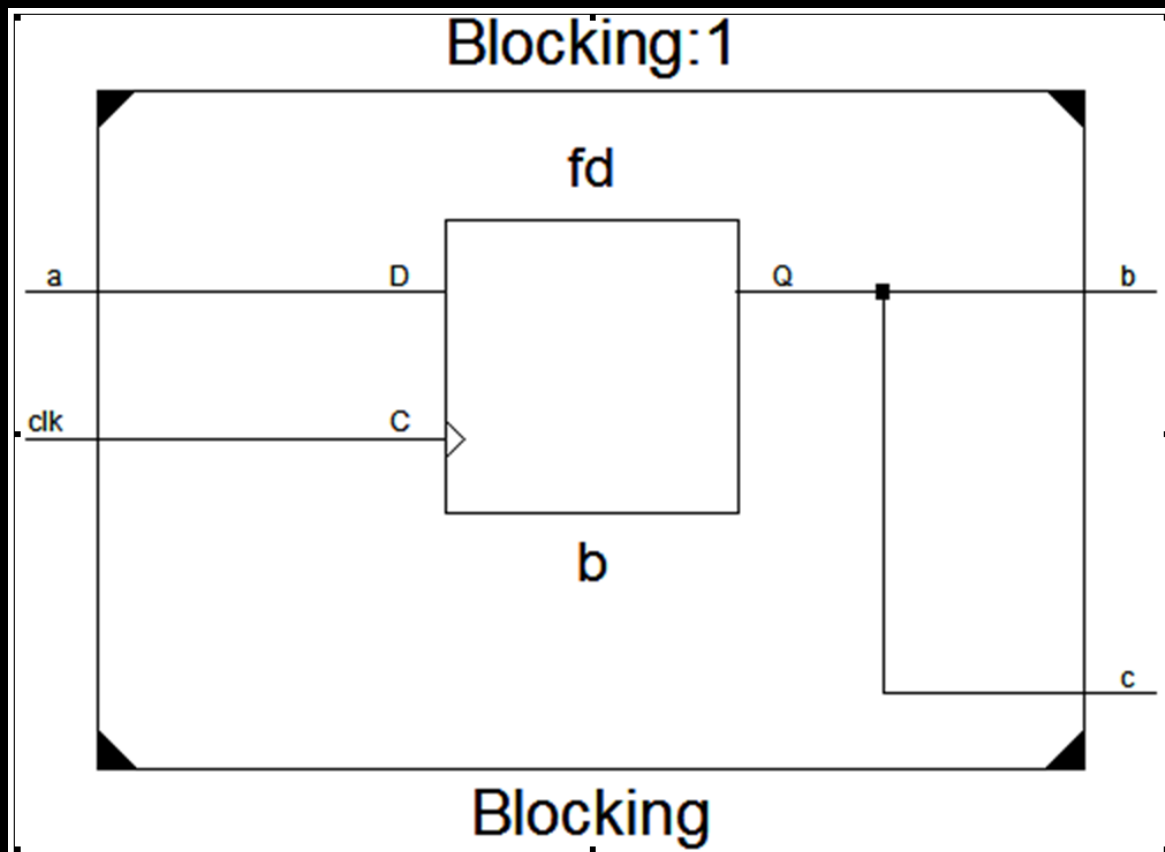
可以简单地把阻塞赋值看成是串行语句，把非阻塞赋值看成并行语句。



阻塞赋值与非阻塞赋值的区别

阻塞赋值:

```
module Blocking(  
    input a,  
    input clk,  
    output reg b,  
    output reg c  
);  
    always @(posedge clk)  
    begin  
        b=a;  
        c=b;  
    end  
endmodule
```



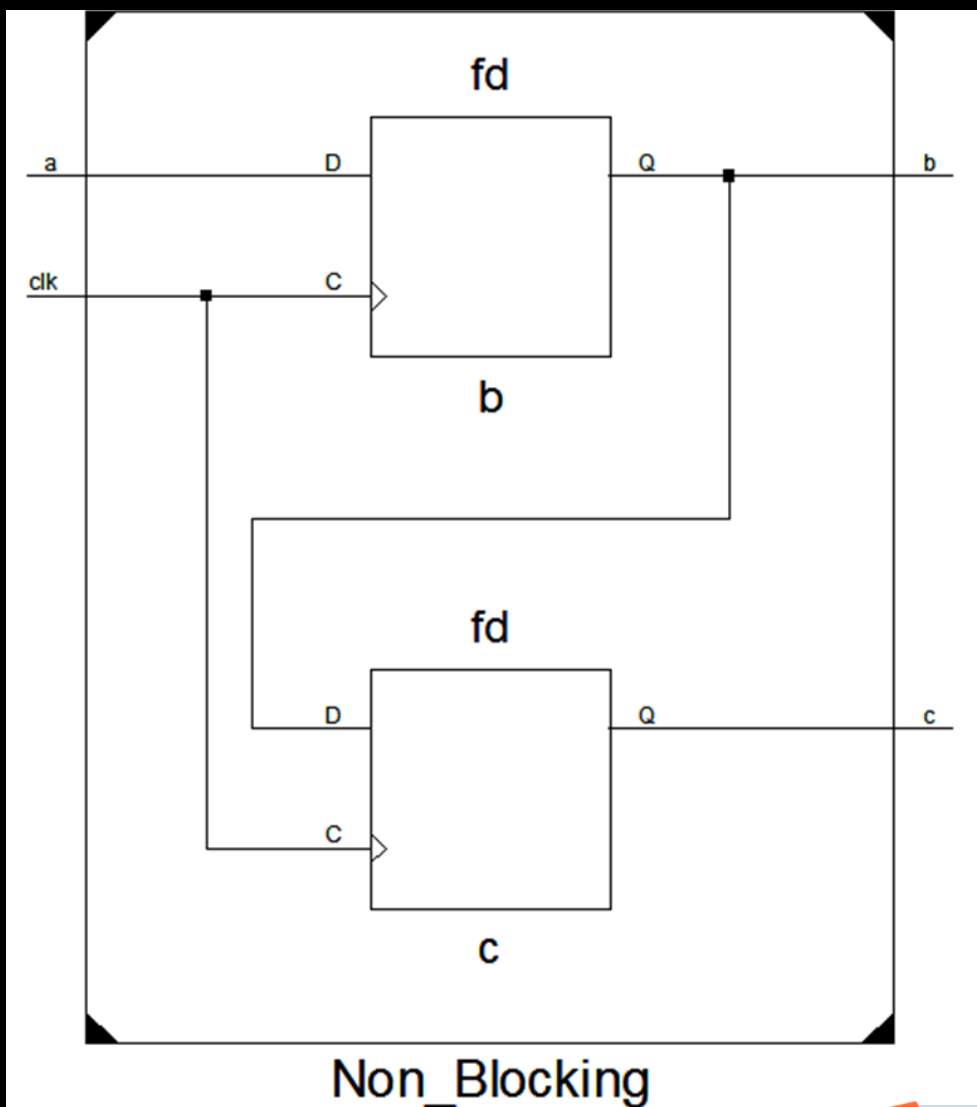
阻塞赋值在该语句结束时就完成赋值操作，即b的值在该赋值语句结束后立刻改变。



阻塞赋值与非阻塞赋值的区别

非阻塞赋值:

```
module Non_Blocking(  
    input a,  
    input clk,  
    output reg b,  
    output reg c  
);  
    always @(posedge clk)  
        begin  
            b<=a;  
            c<=b;  
        end  
endmodule
```



阻塞赋值与非阻塞赋值的区别

采用非阻塞赋值能够实现移位寄存器：

```
module Shiftreg Non Blocking(
```

```
    input clk,
```

```
    input serin,
```

```
    output reg [3:0] q
```

```
);
```

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

```
        begin
```

```
            q[0] <= serin;
```

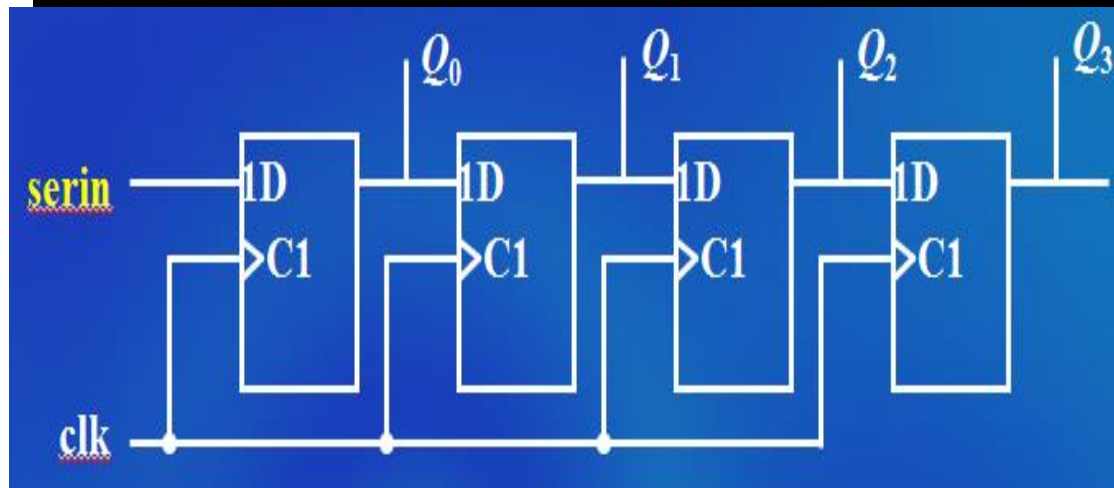
```
            q[1] <= q[0];
```

```
            q[2] <= q[1];
```

```
            q[3] <= q[2];
```

```
        end
```

```
    endmodule
```



非阻塞赋值并行执行



逻辑功能描述

3)调用元件（元件例化）： Verilog内部定义了一些基本门级元件模块。使用实例元件的调用语句，不必重新编写这些基本门级元件模块，直接调用这些模块。







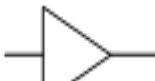
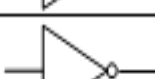
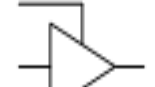
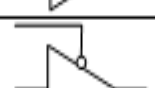
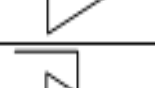

nand nand3_inst(f,a,b,c);

//调用一个三输入与非门

上述语句表示在设计中用到一个名为nand3_inst的三输入端与非门，其输入端为a、b、c，输出端为f。

```
module driver (in, out, en);  
    input [2: 0] in;  
    output [2: 0] out;  
    input en;  
    bufif0 u[2:0] (out, in, en);  
endmodule
```

// array of buffers

类别	关键字	符号示意图	门名称
多输入门	and		与门
	nand		与非门
	or		或门
	nor		或非门
	xor		异或门
	xnor		异或非门
多输出门	buf		缓冲器
	not		非门
三态门	bufif1		高电平使能三态缓冲器
	bufif0		低电平使能三态缓冲器
	notif1		高电平使能三态非门
	notif0		低电平使能三态非门

模块间的调用

Verilog HDL通过“模块调用”或称为“模块实例化”的方法实现子模块与高层模块的连接。

调用方法:

① 采用位置对应方式：严格按照模块定义的端口顺序来连接，不用标明原模块定义时规定的端口名。

例如，已经定义了一个模块Design:

```
module Design(端口1, 端口2, 端口3.....);
```

现在实例化一个与Design相同功能的模块u1:

```
Design u1(u1的端口1, u1的端口2, u1的端口3.....);
```



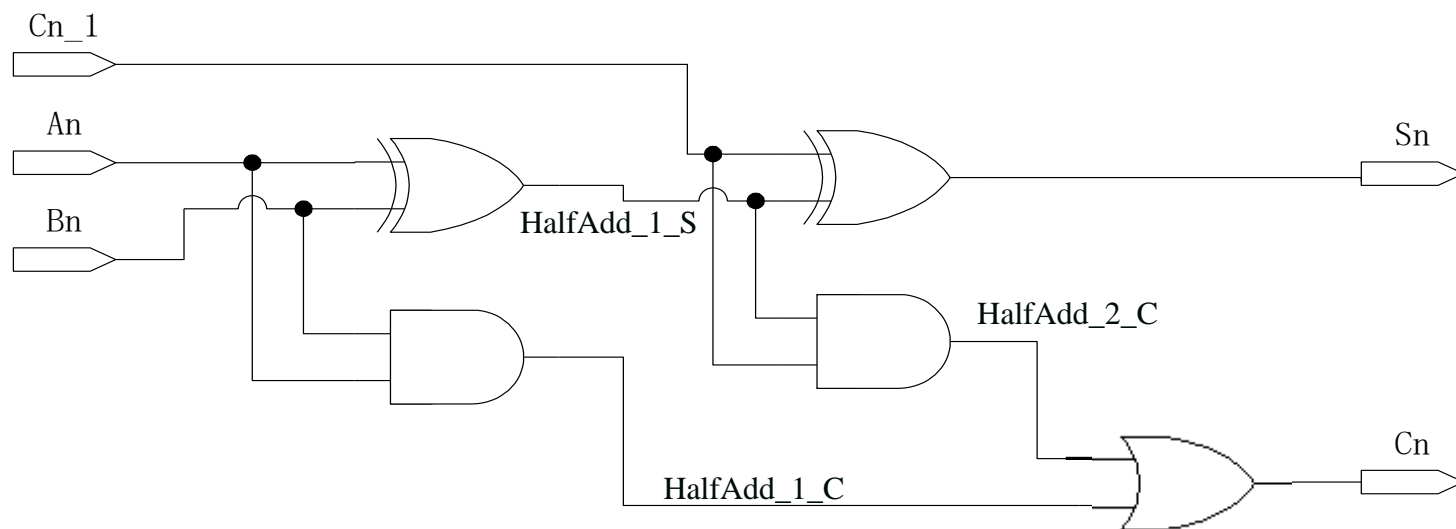
②采用信号名对应方式：引用时用“.”符号标明原模块定义时规定的端口名。调用时端口次序与位置无关，不必按顺序。

```
Design u2( .(端口1(u2的端口1),  
            .(端口2(u2的端口2),  
            .(端口3(u2的端口3),  
            .....  
            );
```

建议：采用这种方式，当被调用的模块管脚改变时不易出错。



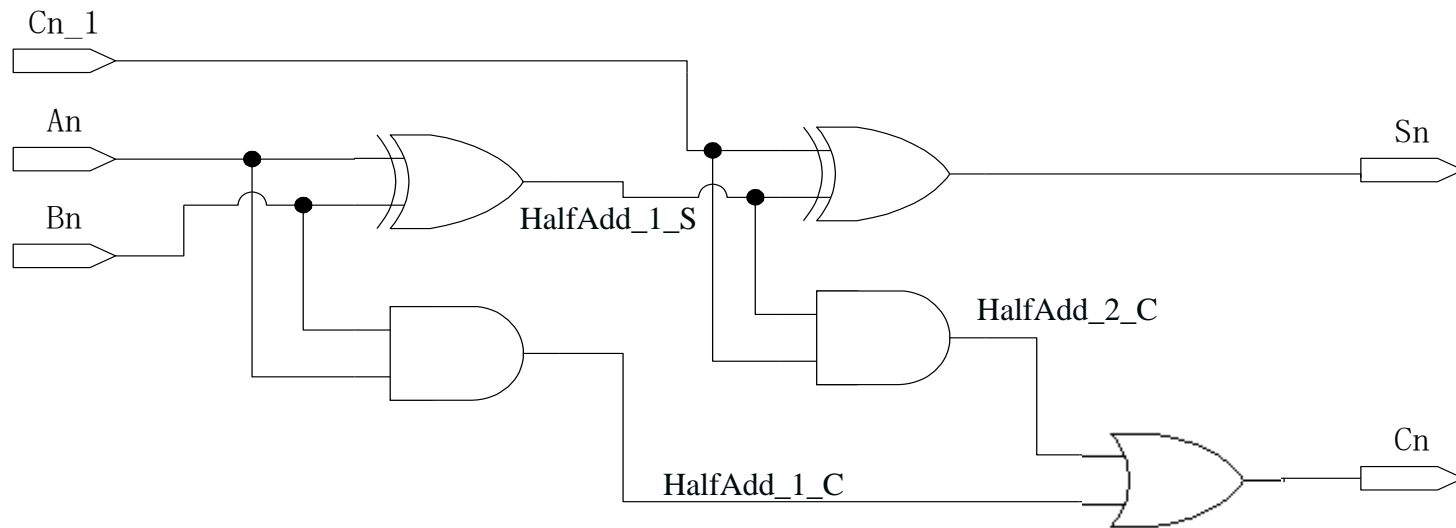
例：采用位置对应方式，调用两个半加器设计一个全加器。



[解] 首先设计半加器（文件名HalfAdd.v）：

```
module HalfAdd(  
    input A,B,  
    output S,C  
);  
    assign S=A^B;  
    assign C=A&B;  
endmodule
```





调用半加器设计全加器（top.v）：

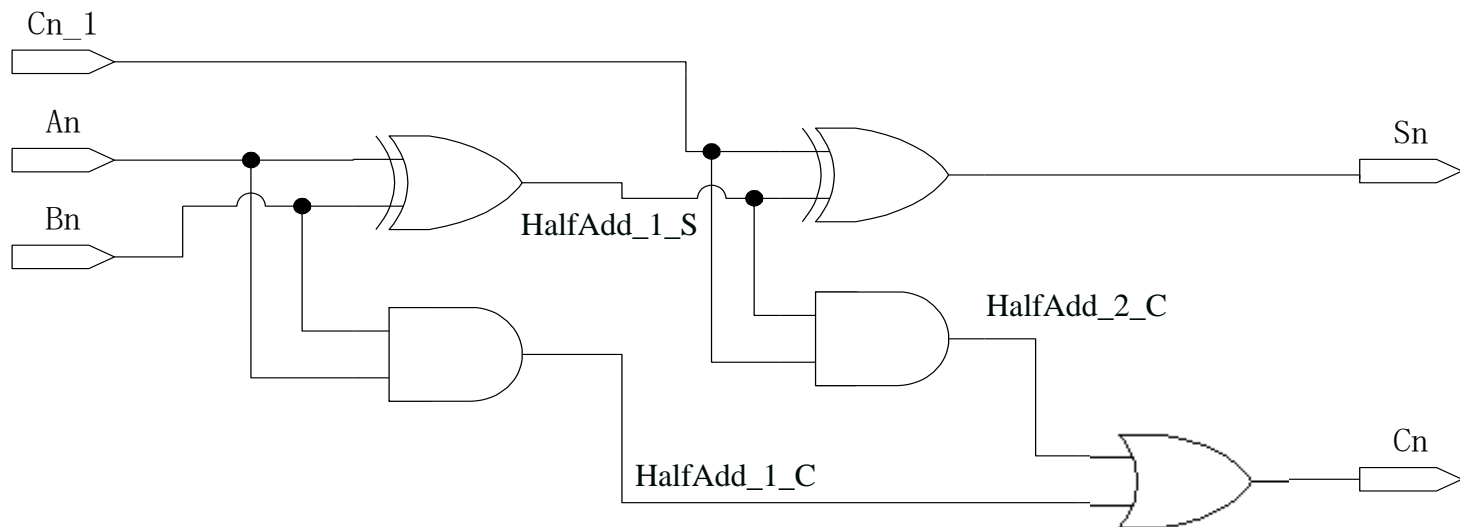
```

module FullAdd(
    input An,Bn,Cn_1,
    output Sn,Cn
);
    wire HalfAdd_1_S; //定义连线型变量,必须是wire型
    wire HalfAdd_1_C;
    wire HalfAdd_2_C;
    assign Cn= HalfAdd_1_C|HalfAdd_2_C;
    HalfAdd HalfAdd_1(An, Bn, HalfAdd_1_S, HalfAdd_1_C);
    HalfAdd HalfAdd_2(Cn_1, HalfAdd_1_S,Sn, HalfAdd_2_C);
endmodule
  
```

C调用函数时对同一个函数的不同调用是一样的。
Verilog中对模块的不同调用是不同的，**即使调用的是同一个模块，也必须使用不同的名字来指定。**



采用信号名对应方式调用半加器设计全加器（top.v）：



```
module FullAdd(  
    input An,Bn,Cn_1,  
    output Sn,Cn  
);  
    wire HalfAdd_1_S;           //定义连线型变量，必须是wire型  
    wire HalfAdd_1_C;  
    wire HalfAdd_2_C;  
    assign Cn= HalfAdd_1_C|HalfAdd_2_C;  
    HalfAdd HalfAdd_1(.A(An), .B(Bn), .S( HalfAdd_1_S), .C(HalfAdd_1_C));  
    HalfAdd HalfAdd_2(.A(Cn_1), .B(HalfAdd_1_S), .S(Sn), .C(HalfAdd_2_C));  
endmodule
```

★ 模块的总结

模块（**module**）是Verilog设计中的基本单元，每个Verilog设计的系统中都由若干**module**组成，每个.v程序都有且仅有一个模块。

- ①模块在语言形式上是以关键词**module**开始，**endmodule**结束的一段程序。
- ②模块的实际意义是代表硬件电路上的逻辑实体。
- ③每个模块都实现特定的功能。
- ④模块之间是并行运行的。
- ⑤模块是分层的，高层模块通过调用、连接低层模块的实例来实现复杂的功能。
- ⑥各模块连接完成整个系统，因此，需要一个顶层模块（**top-module**）。

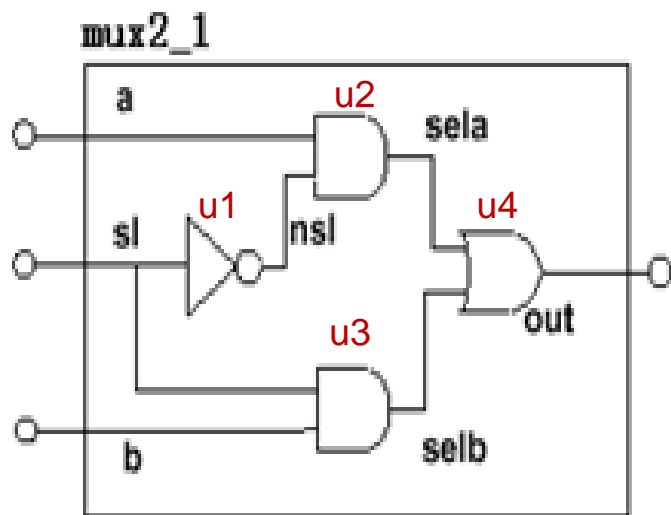


6.6 Verilog设计的层次与风格

如同对一个逻辑问题可用多种方法描述一样，Verilog也可以从电路结构、表达式和行为（即要实现的功能）几方面来描述一个逻辑问题。

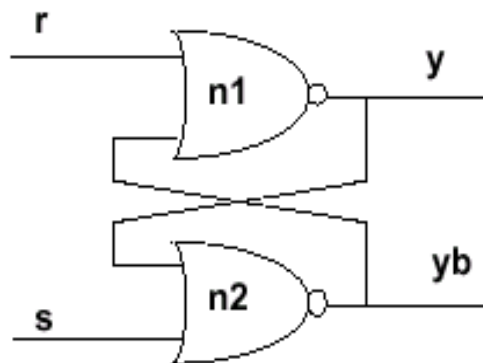
1、结构（Structural）描述： 用程序把电路的具体结构描述出来。通过调用内置元件或用户自定义的基本单元描述电路的结构。侧重于表示一个电路由哪些基本元件组成，以及这些基本元件的相互连接关系。

例：调用门元件实现2选1的MUX。



```
module mux2_1 (  
    input a, b, sl,  
    output out  
);  
    wire nsl, sela, selb;  
    not u1 (nsl, sl);  
    and u2 (sela, a, nsl);  
    and u3 (selb, b, sl);  
    or u4 (out, sela, selb);  
endmodule
```

问题： 以下Verilog模块实现什么逻辑功能？

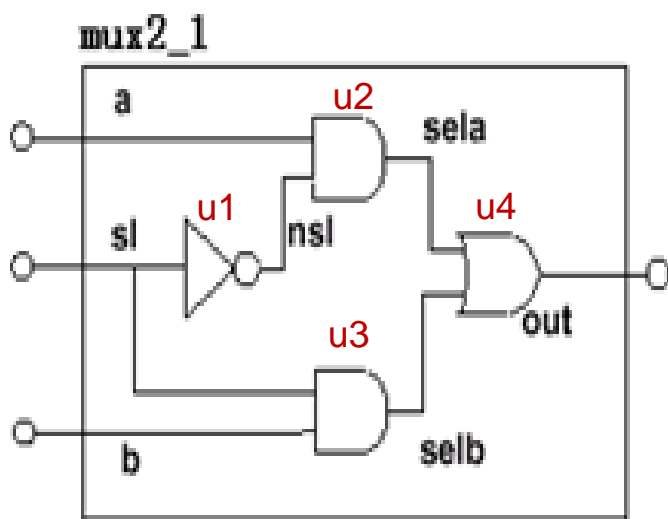


```
module U1 (y, yb, r, s);
    output y, yb;
    input r, s;
    nor n1( y, r, yb);
    nor n2( yb, s, y);
endmodule
```

答：RS锁存器



2、数据流（Data Flow）描述： 用程序把逻辑表达式写出来。
主要使用持续赋值语句assign，多用于描述组合逻辑电路。



例：用数据流描述2选1 MUX。

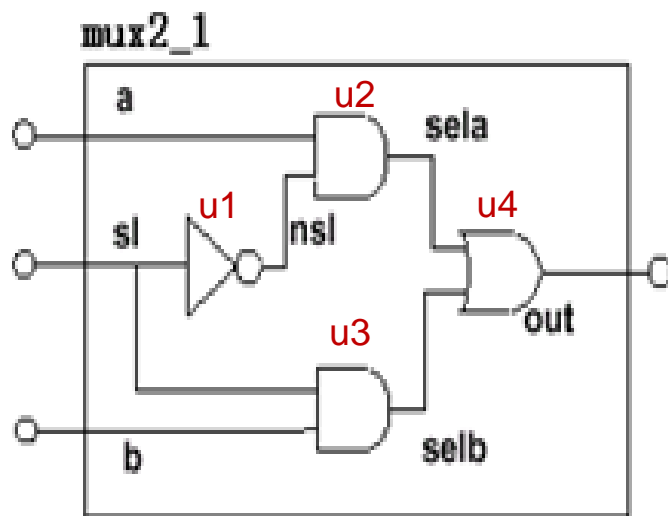
[解] :

```
module mux2_1 (  
    input a, b, s1,  
    output out  
);  
    assign out=(a & ~s1)|(b & s1);  
endmodule
```

与用传统的逻辑方程设计电路很相似。设计中只要有了布尔代数表达式就很容易将它用数据流方式表达出来。表达方法是**用Verilog中的逻辑运算符置换布尔逻辑运算符即可。**

3、行为（Behavioural）描述：只关注逻辑电路输入、输出的因果关系（行为特性），即在何种输入条件下，产生何种输出（操作），并不关心电路的内部结构。EDA的综合工具能自动将行为描述转换成电路结构，形成网表文件。

[例】用行为描述2选1 MUX。

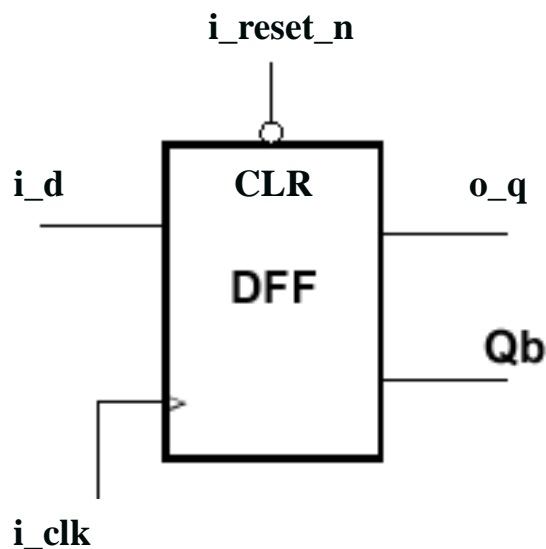


[解]：

```
module muxtwo (  
    input a, b, sl,  
    output reg out  
);  
always @( sl or a or b)  
    if (!sl) out=a;  
    else out=b;  
endmodule
```

行为描述设计者只需写出源程序，而挑选电路方案的工作由EDA软件自动完成。设计者而言，采用的描述级别越高，设计越容易；对综合器而言，行为级的描述为综合器的优化提供了更大的空间，较之门级结构描述更能发挥综合器的性能，所以在电路设计中，除非一些关键路径的设计采用门级结构描述外，**一般更多地采用行为级建模方式。**

- Verilog有高级编程语言结构用于行为描述，包括：
wait, while, if then, case和forever
- Verilog的行为建模是用一系列以高级编程语言编写的并行的、动态的过程块来描述系统的工作。



DFF

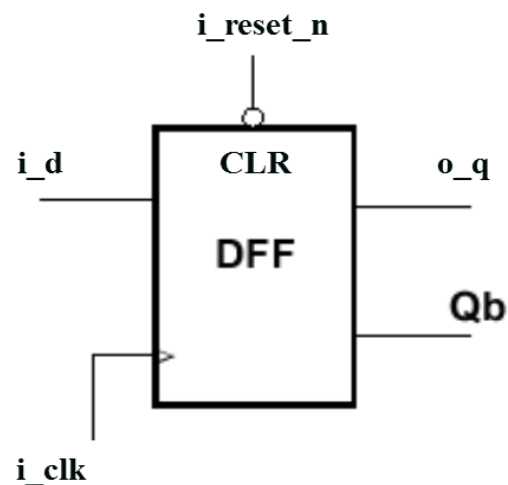
在每一个时钟上升沿，
若C1r不是低电平，
置Q为D值，
置Qb为D值的反

无论何时C1r变低
置Q为0，
置Qb为1

```

module dff_reg
(
    input    i_reset_n,
    input    i_clk,
    input    i_d,
    output reg o_q
);
always@( negedge i_reset_n or posedge i_clk )
begin
    if (!i_reset_n)
        o_q <= 1'b0;
    else
        o_q <= i_d;
end
endmodule

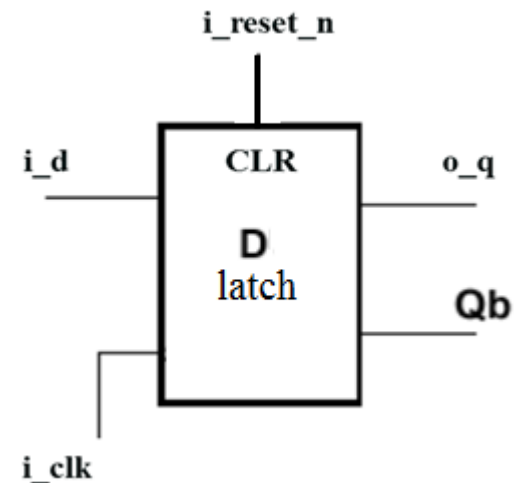
```



```

module latch_rst
(
    input    i_reset,
    input    i_clk,
    input    i_d,
    output reg o_q
);
always@( i_reset or i_clk or i_d )
begin
    if (i_reset)
        o_q = 1'b0;
    else if (clk)
        o_q = d;
end
endmodule

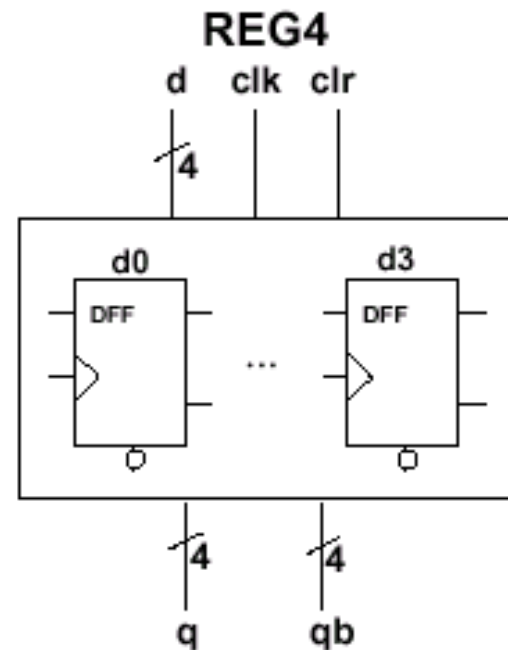
```



问题： 以下Verilog模块实现什么逻辑功能？

```
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
```



问题： 以下Verilog模块实现什么逻辑功能？

```
module test2(clk,clr,out);
input clk,clr;
output[3:0] out;
reg[3:0] out;
always @(posedge clk or negedge clr)
begin
    if (!clr) out<= 4'h0;
    else
        begin
            out<=(out>> 1);
            out[3]<= ~out[0];
        end
    end
end
endmodule
```

4位扭环形计数器



4、混合描述：同时具有上述三种描述方法。在一个.v程序中，用**assign**语句描述简单的组合逻辑电路，用**always**语句描述较为复杂的逻辑过程。一个.v程序所表达的逻辑电路可由多个**assign**语句和多个**always**过程块来描述。多个**assign**语句和多个**always**过程块是同时并发执行的。

并行执行

```
module ex (.....);  
    input .....;  
    output .....;  
    reg .....;  
    assign a=b&c; //数据流描述  
    always @(.....) //行为描述  
        begin  
            ...  
        end  
    and u1(a,b,c); // 结构描述  
endmodule
```



6.8 Verilog有限状态机设计

状态机：一个系统或机器，由若干个状态构成，触发后会发生状态的转移，此系统或机器称之为状态机。

状态往往是有限的，所以状态机又称为有限状态机（finite-state machine, FSM）。三个特征：

- 状态总数（**state**）是有限的。
- 它有记忆的能力，能够记住当前的状态。任一时刻，只处在一种状态之中。
- 某种条件下，会从一种状态转变到另一种状态。



状态机类型

Moore有限状态机：输出仅依赖于内部状态，跟输入无关。

Mealy有限状态机：输出不仅决定于内部状态，还跟外部输入有关。

一个对象的状态越多、发生的事件越多，就越**适合**采用有限状态机来描述。

例如：用六个数码管和两个按键设计一个电子表。要有年、月、日、时、分、秒显示，还要有秒表和校时功能，分别用按键**SelectKey**和**AdjustKey**完成操作。



功能描述

当前显示状态	按键操作	操作结果
显示时间	按 SelectKey 键	屏幕显示变成日期
显示日期	按 SelectKey 键	屏幕显示变成秒表
显示秒表	按 SelectKey 键	屏幕显示变成时间
显示秒表	按 AdjustKey 键	秒表归 0
显示时间	按 AdjustKey 键	屏幕时间、日期交替显示
时间、日期交替显示	按 AdjustKey 键	屏幕“时”闪烁显示
“时”闪烁显示	按 SelectKey 键	屏幕“分”闪烁显示
“分”闪烁显示	按 SelectKey 键	屏幕“年”闪烁显示
“年”闪烁显示	按 SelectKey 键	屏幕“月”闪烁显示
“月”闪烁显示	按 SelectKey 键	屏幕“日”闪烁显示
“日”闪烁显示	按 SelectKey 键	屏幕回到时间显示
“时”闪烁显示	按 AdjustKey 键	屏幕“时”加 1，超过 23 回 0
“分”闪烁显示	按 AdjustKey 键	屏幕“分”加 1，超过 59 回 0
“年”闪烁显示	按 AdjustKey 键	屏幕“年”加 1，超过 99 回 0
“月”闪烁显示	按 AdjustKey 键	屏幕“月”加 1，超过 12 回 0
“日”闪烁显示	按 AdjustKey 键	屏幕“日”加 1，超过 31 回 0

程序的运行完全取决于两个键的次序，有无数种次序组合，根本不可能画出流程图来。

功能的“语言描述”在语法上似乎有某种规律，就是：当系统处于某状态（S1）时，如果发生了什么事情(E)，就执行某功能(F)，然后系统变成新状态（S2），只要能用上面这句话描述的系统，都可以用一种状态跳转机制很方便的实现，并且一句话其实就是一个if(...)，无论有多么复杂的功能，只要能用上面这句话描述，都可以通过状态机编程实现。

整个系统中有2个事件分别是：SelectKey键按下和AdjustKey键按下。

SelectKey按下时执行：

```
{
    if(Status==TIME)                //当显示时间时按下SelectKey键
        { Status=DATE; }           //变成显示日期
    if(Status==DATE)                //当显示日期时按下SelectKey键
        { Status=SEC; }             //变成显示秒钟
    if(Status==SEC)                 //当显示秒钟时按下SelectKey键
        { Status=TIME; }           //变成显示时间
    .....
    .....
}
```

AdjustKey按下时执行：

```
{
    if(Status==SEC)                 //当显示秒钟时按下AdjustKey键
        { Secound=0; }             //秒归0
    if(Status==TIME)               //当显示时间时按下AdjustKey键
        { Status=TIMEDATE; }       //变成时间日期交替显示
    .....
    .....
}
```

和自然语言描述完全一致，很快就能写完程序。上述一大堆if用case来实现更好。



本章小结

采用Verilog HDL设计比采用电路图输入的方法更有优越性；Verilog HDL与VHDL相比更加基础、更易掌握，因其与C语言类似；Verilog HDL可用于复杂数字逻辑电路和系统的总体仿真、子系统仿真和具体电路综合等各个设计阶段。

Verilog HDL程序由模块构成，每个模块位于`module`和`endmodule`之间；每个模块要进行端口定义，并说明是输入口还是输出口，然后对模块的功能进行描述；模块可以层次嵌套，将电路设计分割成不同的小模块来实现特定的功能，用一个上层模块通过实例引用把这些模块连接起来。

硬件描述语言实现一个复杂的数字逻辑系统时，只关心系统的行为表现，而不用关心系统的内部硬件结构，以及硬件上如何实现，编译软件会自动将硬件描述语言翻译成相应的硬件电路，因此应重点掌握如何用`always`行为块描述语句描述数字逻辑过程，对于简单的组合逻辑部分用`assign`赋值语句实现。