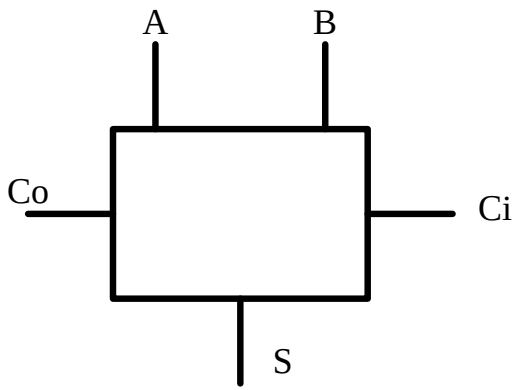
上海交通大学
Shanghai Jiao Tong University

Verilog HDL 硬件描述语言（1）

上海交通大学微电子学院
蒋剑飞

全加器

- 全加器表述



$$P = A \oplus B \ .$$

$$S = P \oplus Ci$$

$$Co = PCi + P'A$$

真值表

A	B	Ci	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加器的Verilog描述

$$P = A \oplus B \ .$$

$$S = P \oplus Ci$$

$$Co = PCi + P'A$$

模块名

```
module full_adder ( a, b, ci, s, co);

input a,b,ci;
output s;
output co;

wire a,b,ci,s,co;
wire p;

assign    p=a^b;
assign    s=p^ci;
assign    co=(p & ci) | (!p  &  a);

endmodule
```

模块

有分号

端口声明

类型定义

表达式

没有分号

module（模块）

- 模块是Verilog的基本设计描述单位，用于描述某个设计的功能或者结构，以及他们与其他模块通信的外部端口。
- module和endmodule总是成对出现，是系统的保留字。
- Verilog文件的后缀是“.v”,如full_adder.v。一个verilog文件中可以包括多个module，一般建议一个module对应一个文件。

标识符

- Verilog HDL中的标识符（identifier)可以是任意一组字母、数字、$符（$)、下划线（_)的组合。
  标识符区分大小写。
  标识符的第一个字母必须是字母或者是下划线。

- 系统标识符（关键字、保留字）
  Verilog HDL定义了一系列的保留字，保留字都是小写的。
  例：module，endmodule, input, ouput, wire, assign等

- 用户自定义标识符需符合上面规定

标识符举例

- 合法的

```
Count
COUNT              //与Count不同
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

注释

- 注释
  - 单行注释:

    //本行被注释

  - 多行注释:

    /* 本段被注释 …*/

  - 养成良好的注释习惯

```
//version:xxxx
//date:xxx
//author:xxx
/*=========================
          a         b
          |         |
         ----------------
  co__ |      FA       |__ ci
         |               |
         ----------------
                  | S
=========================*/
module full_adder ( a, b, ci, s, co);
input a,b,ci;                 //c is a carry_in。
ouput s;                       //
output co;                    //
wire a,b,c,s,co;
wire p;
assign     p=a^b;
assign     s=p^ci;
assign     co=(p & ci) | (!p  &  a);
endmodule
```

格式

- 格式
  - 除区分大小写外，Verilog HDL是自由格式的，可以跨越多行，也可以在一行内编写。

```
module full_adder ( a, b, ci, s, co);
input a,b,ci;
output s;
output co;
wire
a,b,ci,s,co;
wire p;
assign     p=a^b;
assign     s=p^ci;
assign     co=(p & ci) | (!p  &  a);
endmodule
```

  - Verilog中三种符号认为是空白，没有特殊意义
    空格、制表符、换行符

端口

- 模块的端口是模块与外部模块的通信接口，端口可以分为三种类型：
  - 输入端口　（input）
  - 输出端口　（output）
  - 输入输出端口（inout）

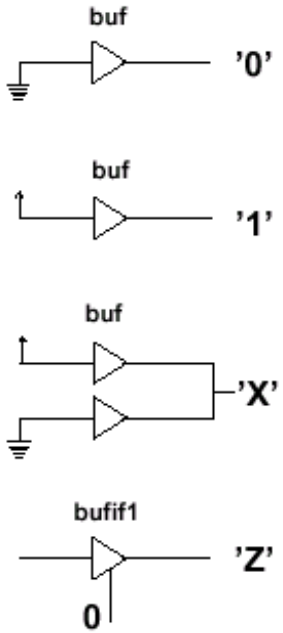
```
module test (address,ctrl,data);

input  [1:0] address;
output [2:0] ctrl;
inout  [8:0]data;
```

端口定义中需要指定类型、宽度

```
input  a,b,c;
wire  a,b,c;
```

值集合与数据类型

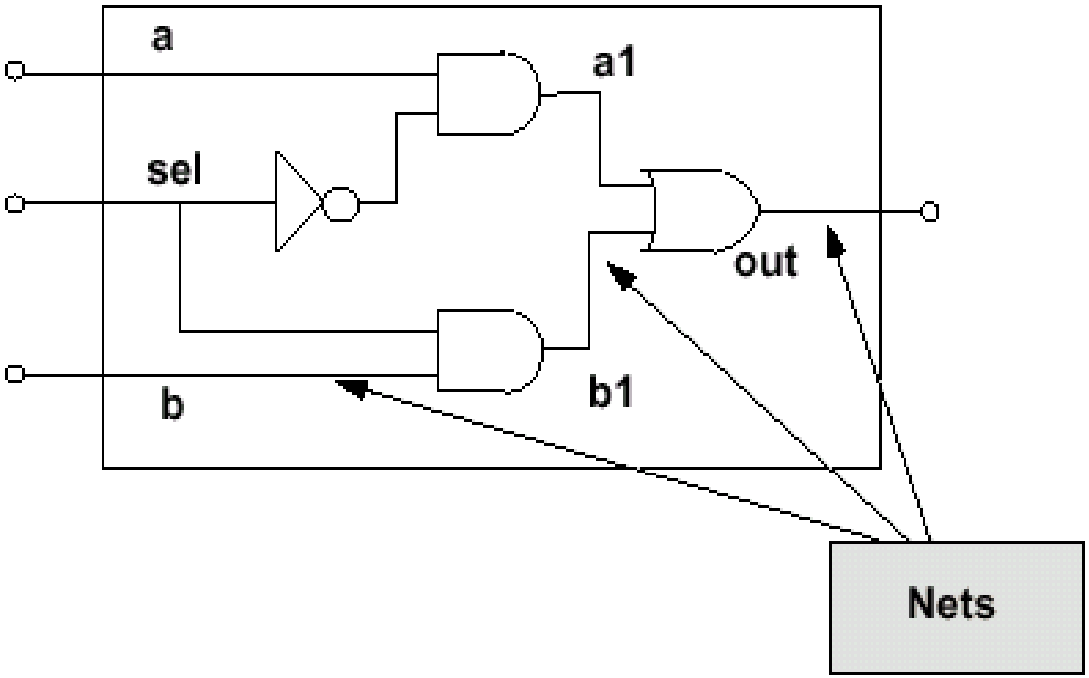
- 值集合
  - **逻辑0，false**
  - **逻辑1，true**
  - **未知值 " x"**
  - **高阻 " z"**



- 数据类型
  - 线网类型（nets）
  - 寄存器类型（registers）

线网(1)

- 线网类型表示单元之间的物理连线，线网不存储值，它的值由驱动单元的值决定，比如连续的赋值或者门的输出，如果没有驱动，线网的缺省值是Z（trireg类型除外）。



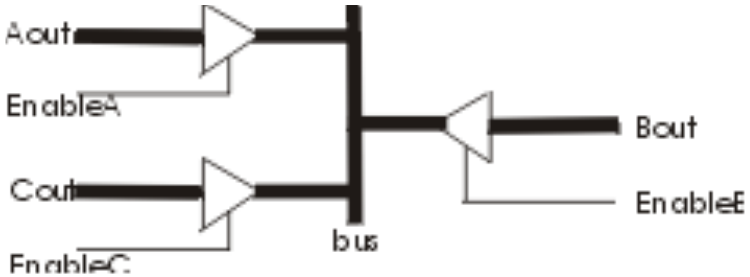
线网（2）

- 线网类型包括以下11种子类型

net类型	功 能
wire, tri	标准内部连接线(缺省)
supply1, supply0	电源和地
wor, trior	多驱动源线或
wand, triand	多驱动源线与
trireg	能保存电荷的net，用于电容节点建模
tri1, tri0	无驱动时上拉/下拉

wire/tri类型

- 它们是最常见的线网类型，两者的语法与功能一致，tri本意用于描述多个驱动源同时驱动一根线的线网类型，而wire本意用于描述一个驱动源的驱动。实际上应用中，没有区别，为了可读性在多个驱动源时建议使用tri类型。



- Wire/tri在多个驱动源时值的判断

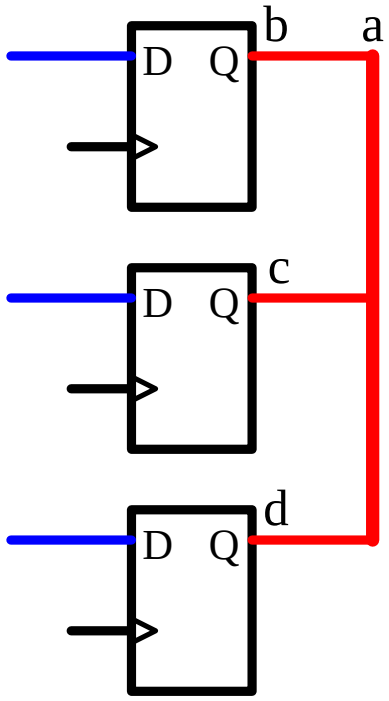
Table 3-2—Truth table for wire and tri nets

wire/ tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

系统缺省的数据类型是1比特的wire类型。

设计举例

```
module test_wires ();
    wire a0;
    wand a1;
    wor  a2;
    reg b,c,d;
    assign a0=b;
    assign a0=c;
    assign a0=d;
initial
begin
    b=1;
    c=0;
    d=1;
    #1 $display ("%b", a0 );
end
endmodule
```



a0

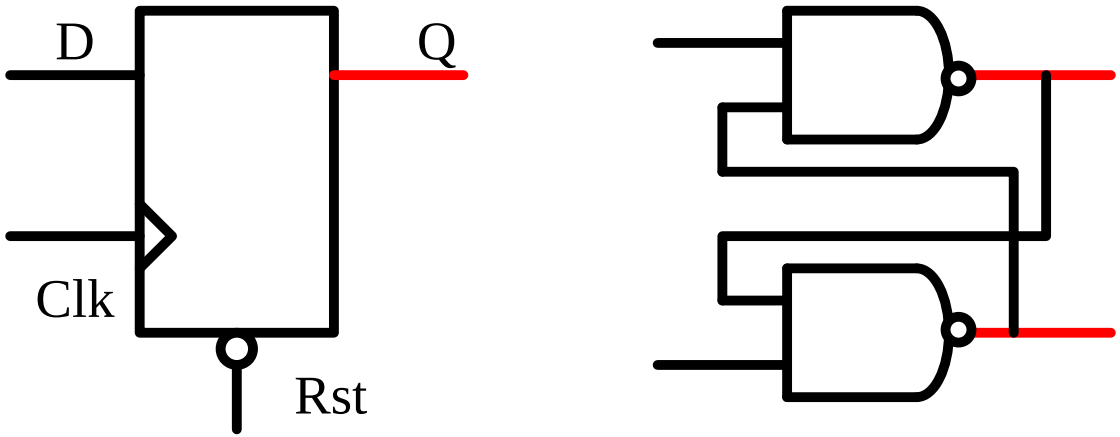
x

寄存器类型（register)

- 寄存器类型(register)分为5种子类型
  - reg类型
  - integer类型
  - real类型
  - time类型（testbench中介绍）
  - realtime类型（testbench中介绍）

reg类型

- reg类型是数据存储单元的抽象。

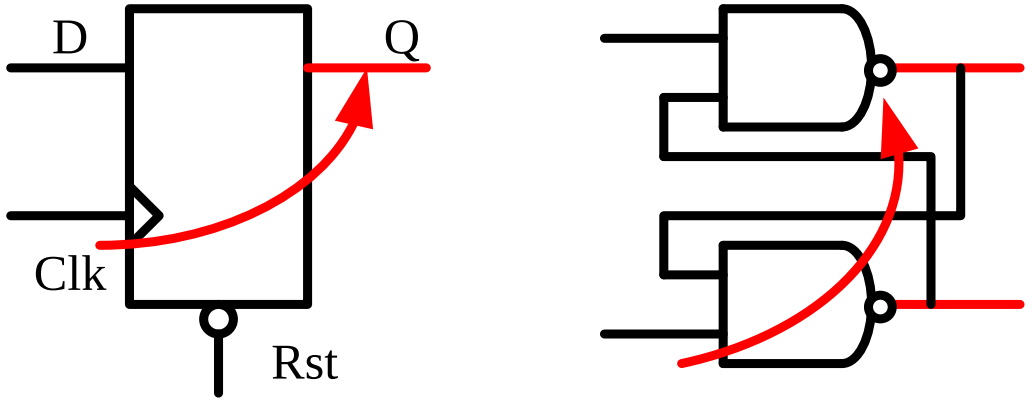


- reg类型与wire一起类型是RTL级描述的基本数据类型。

在硬件描述语言中，有时候数据类型并不一定与硬件电路相关，
在组合逻辑的行为描述中，也会定义reg类型。

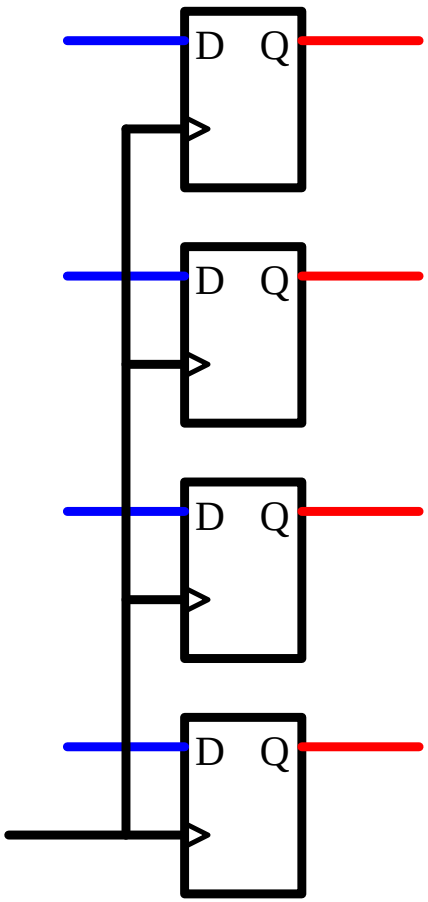
reg类型与端口

- reg类型不能作为输入，对于包括输入的inout端口也适用。



这是初学者经常且最容易犯的错误之一

向量




```
wire [3:0] data_in;
reg  [3:0] data_out;
```

```
wire    [msb:lsb]    data_in;
```

MSB和LSB须是常数。
MSB和LSB可以是正数，负数，或者0。
LSB可以比MSB大，或者小。

```
reg [-1:4] b;              // a 6-bit vector register
wire w1, w2;               // declares two wires
reg [4:0] x, y, z;     // declares three 5-bit registers
```

建议的定义方式：

```
wire [3:0] data_in;
reg  [3:0] data_out;
```

常量

```
256           //非定长的十进制数
4'b10_11      //定长的整数常量
8'h0a         //定长的整数常量
'b1,'hfba     //非定长的整数常量
90.006        //实数常量
"bond"        //字符串常量，每个字符作为8位的
              //ascii值存储,使用时要用双引号
wire  [3:0] data_out;
wire  [0:3] data_in;
wire [9:0]  ask;

assign  data_out=4'b1000;     //data_in[3]=1
assign  data_in =4'b1000;     //data_in[0]=1
assign  ask="x";              //8'b01111000  (120)
```

常量中的符号处理

```
wire  [3:0] data_out;
assign data_out=12;          //1100
assign data_out=-12;         //0100
assign data_out=5'b01100;    //1100
assign data_out=5'b10100;    //0100
```

```
wire  [4:0] data_out;
assign data_out=12;          //01100
assign data_out=-12;         //10100
assign data_out=5'b01100;    //01100
assign data_out=5'b10100;    //10100
```

```
wire  [6:0] data_out;
assign data_out=12;          //0001100
assign data_out=-12;         //1110100
assign data_out=5'b01100;    //0001100
assign data_out=5'b10100;    //0010100
```

- Verilog表达式中的常量首先进行符号位扩展，再将需要的位进行赋值操作，根据表达式中变量的长度对表达式的值自动地进行调整。

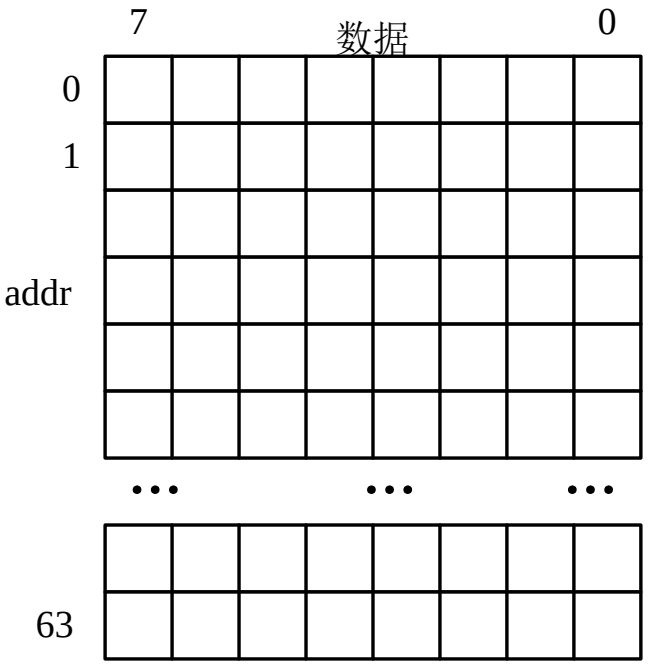
- Verilog自动截断或扩展赋值语句中右边的值以适应左边变量的长度。

Verilog中的符号处理

- Unsigned
  - reg类型
  - time类型
  - net 类型
- Signed
  - integer类型
  - real类型
  - realtime类型
  - 常量
- Signed 声明
  - wire signed
  - reg signed

net类型
wire, tri
supply1, supply0
wor, trior
wand, triand
trireg
tri1, tri0

存储器



寄存器组可以用来建模ROM，RAM以及register file。

reg [msb:lsb] memory [upper1:lower1];

reg [7:0] memory1 [0:63]; //64个8位的存储器

reg mema [1:5]; //5个一位的存储器

reg [1:5] rega; //1个5位的寄存器

区别：对于存储器的访问须指定地址，不能实现同时的所有地址访问。

rega = 0; // Legal syntax

mema = 0; // illegal syntax

mema[1] = 0; //Assigns 0 to the first element of mema。

位选择与部分选择

- 位选择从向量中抽取特定的位

- 部分选择

net_or_reg_vector [msb_const_expr : lsb_const_expr]

```
wire  a,a2;
wire [7:0]  b1,b2;
wire [15:0] d;
reg [7:0] memory [0:15];

assign   a=d[6];                    //位选择
assign   b1[7:4]=d[15:12];
assign   b2=memory[3]
```

操作符

- Verilog中的操作符

分类	操作符
连接复制符	{}，{{}}
算术运算符	*， /， +，-， %
关系操作符	<, <=, >, >=
相等操作符	==,!=, ===,!==
逻辑操作符	!,&&,||
比特操作符	~, &, |, ^, ^~(~^)
归约操作符	&, ~&, |, ~|, ^, ~^(^~)
移位操作符	<<, >>
条件操作符	? :
事件操作符	or

连接复制操作符

连接复制操作符{}，{{}}

```
wire [7:0]  a,b,c,d,e;
wire [15:0] f;
assign e={a[7:6], b[5:4], c[3:2], a[1:0]};  //连接
assign f={a,b};
…
data1={3'b101,2'b11,3'd7};                  //连接
…
data2={128{3'b101}};                        //复制
data2={100{1'b1},8{a},8{f[15:7]}};   //复制，连接
```

次数

注意：连接的数之间用逗号分开
连接的数必须指定位数

可以从不同的矢量中选择位并用它们组成一个新的矢量。
用于位的重组和矢量构造

在级联和复制时，必须指定位数，否则将产生错误。
下面是类似错误的例子：

```
a[7:0] = {4{ ´b10}};
b[7:0] = {2{ 5}};
c[3:0] = {3´b011, ´b0};
```

算术运算

- 算术运算有* ， /， +， - ， %

```
reg [3:0] a,b;
wire [3:0] c,d,f,g,h,m,n,x,y;
wire [8:0] e,i;
initial          begin
a=4'b1010;b=4'b1111; end
assign c=a+b;                  //c=4’b1001
assign d=a-b;                  //d=4'b1011
assign e=a*b;                  //e=9'b010010110
assign f=a/b;                  //d=4’b0000
assign g=a%b;                   //g=4'b1010
assign h=10-3;                 //h=4'b0111
assign i=10*-3 ;               //I=9’b111100010
assign m=10/-3;                 //m=4’b1101
assign n=10%-3;                 //n=4’b0001
```

- 整数除法截断小数部分
- 取模运算的结果符号与第一个操作数相同
- 操作数中有x或者z，结果是x

ble 4-8—Data type Interpretation by arithmetic operators

Data type	Interpretation
net	Unsigned
reg	Unsigned
integer	Signed, 2's complement
time	Unsigned
real, realtime	Signed, floating point

关系操作符

关系操作符 <, <=, >, >=

```
reg [2:0] a,b,c,d1,d2,e,f;
integer j,k,q;
a=(3'b101 < 3'b11x);     //a=x
b=4'b101;
c=2'b11;
e=(b>c);        //e=1
j=- 4'd10;
k=-10;q=5;
d1=j > b;        //d1=1，有符号数与无符号数的比较
d2=j>q;          //d2=0;
f=k>q;          //f=0
```

- 当操作数含有x或者z时，结果是x
- 当操作数位宽不同时，位宽小的数最高为填0，并扩展成相同位宽后进行比较。
- 符号法则遵循数据类型的规定
- 避免出现有符号数与无符号数的比较

逻辑操作符

- 逻辑操作符 &&, ||, !

```
...
a = 4'b0011;     //逻辑值为 "1"
b = 4'b10xz;     //逻辑值为 "1"
c = 4'b0z0x;     //逻辑值为 "x"
e = a && b;      // e = 1
f =  a && c;     // f = x
g = ！ a;        // g=0
```

- 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。
- 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。

相等操作符

- 逻辑相等==，！=

  操作数中只要含有x或者z，结果是x

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

- Case等===，！==

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

```
…
a = 4‘b101z;
b = 4‘b10xz;
c=  4‘b10xz;
d =（a==b);        //d=x
e = (a===b);       // e=0
f  = (b===c);      //  f=1
```

比特操作符

- 比特操作符:~， &， |， ^， ~^(^~)

```
…
a=4’b1011;
b=4’b1010;
h=4’b1x0x;
c=~a;          //c=4’b0100;
d=a&b;         //d=4’b1010;
e=a^b;         //e=4’b0001;
f=a|h;         //f=4’b1x11;
```

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

~	
0	1
1	0
x	x
z	x

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

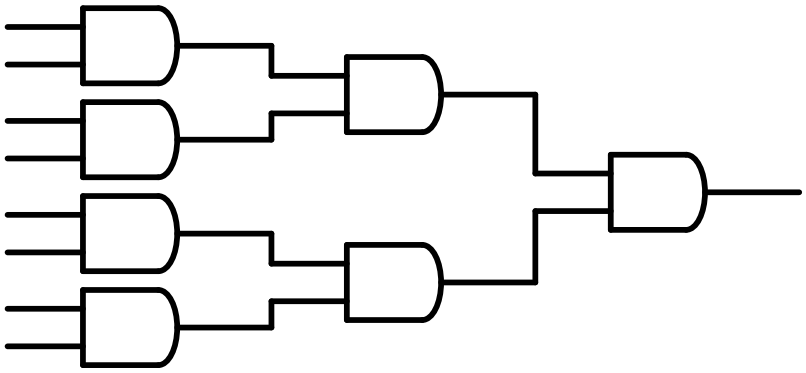
^~ ~^	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

归约操作符

- 归约操作：&, ~&, |, ~|, ^, ^~(~^)

```
 …
a=4’b1011;
b=4’b1010;
h=4’b1x0x;
c=&a;          //c=1’b0;
d=|b;          //d=1’b1;
e=&h;          //e=1’b0;
f=^a;          //f=1’b1;
```

- 归约操作符的操作数只有一个。
- 对操作数的所有位进行位操作。
- 结果只有一位，可以是0, 1, X。

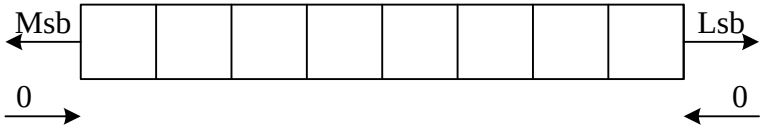


```
wire [7:0]  data;
full_p1=(data==8’hff);

full_p2=&data;
```

移位操作符

- 移位操作符: <<, >>

```
…
wire [7:0] data,result;

assign result = data <<3;

assign result ={data[4:0],3'b0};
```



- 在移位操作中，右边的操作数总是被当成无符号数，因此在需要位扩展时总是填0。
- 移入的位只能是0.

条件操作符

条件操作符：？：

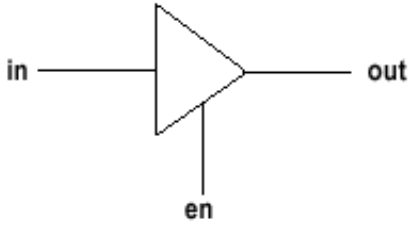
result=expression1 ? expression2:expression3

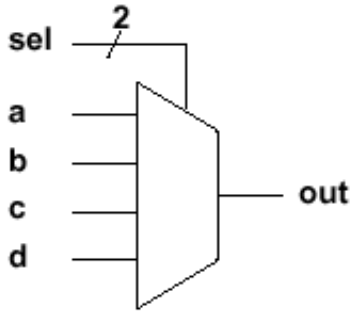
expression1 为真执行 **expression2**

否则执行**expression3**

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

```
…
out=en？ in: 1’bz;      //三态门

out=(sel==2’b00)?a:
      (sel==2’b01)?b:
      (sel==2’b10)?c:d;
```





事件操作符

- 事件操作符： or

在行为级描述中使用

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


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

操作符的优先级

优先级

操作符类型	符号
连接及复制操作符	{}，　{{}}
一元操作符	+，-，！，　~，　&，　|，　^
算术操作符	*，　/，　%
	+，　-
逻辑移位操作符	<<，　　>>
关系操作符	>，　<，　>=，　<=
相等操作符	= =，　= = =，　!= ，　!= =
按位操作符	&，
	^，　~^，
	|
逻辑操作符	&&，
	||
条件操作符	？：

最高

优先级

最低

除条件操作以外，操作符按照从左到右的方式组织。优先级高的操作符先组织。用”（）”可以改变操作符的优先级.

用Verilog描述一个全加器

```
module  testing ( a, b, ci, s, co);

input a,b,ci;
output s;
output co;

wire a,b,ci,s,co;

assign  {co,s}=a+b+ci;

endmodule
```

输入端口类型不能是reg

用Verilog描述一个反相器

```
module  inv_test ( a, y);
input a;
output y;
wire a, y;
assign y=!a;
endmodule
```

Thanks