



Verilog

硬件描述语言与设计



第三章、语法与要素



第三章、语法与要素



3.1 数据类型

3.2 模块端口

3.3 表达式

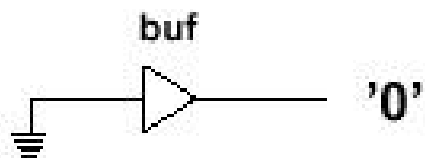
3.4 标准主要差别



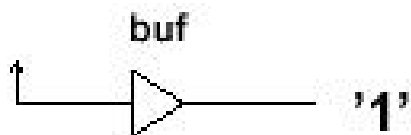
3.1 数据类型

3.1.1 数值

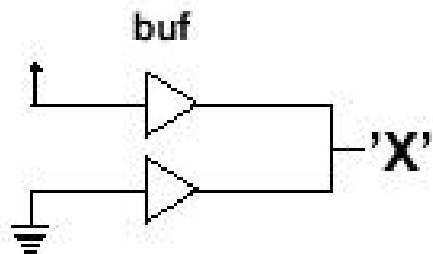
数字电路中的数值信号通常由高、低电平的电平值决定，因此Verilog硬件描述语言规定的数值包含**四种类型**。



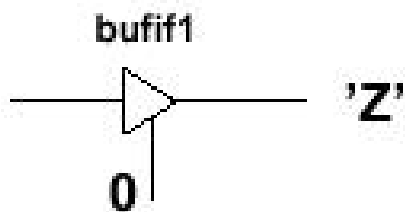
0、低、逻辑低、地、VSS、伪



1、高、逻辑高、电源、VDD、真



X、不确定，逻辑冲突无法确定其逻辑值



HiZ、高阻抗、三态、无驱动源



3.1 数据类型

■ 驱动强度

- Verilog HDL语言中规范了逻辑门级电路输出的驱动强度。驱动强度分为8个等级，从7（最强）到0（最弱）。
- 不同驱动强度的竞争，结果由强度优先决定；同驱动强度的结果为不定值。

驱动名称	驱动强度	类型
Supply	7（最强）	驱动
Strong	6	驱动
Pull	5	驱动
large	4	存储
Weak	3	驱动
Medium	2	存储
Small	1	存储
Highz	0（最弱）	高阻



3.1 数据类型

■ 线网（net）类型

为表述电路器件之间物理连线而定义其为**线网数据类型**，只有连接功能。

■ 线网类型的分类

类型（关键字）	功能
wire, tri	对应于标准的互连线（缺省）
supply1, supply2	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

红色的
为不可
综合的
net类型

- 如果没明确声明连接是何种类型，默认是指**wire**类型
- **tri**类型可以用于描述多个驱动源驱动同一根线的线网类型



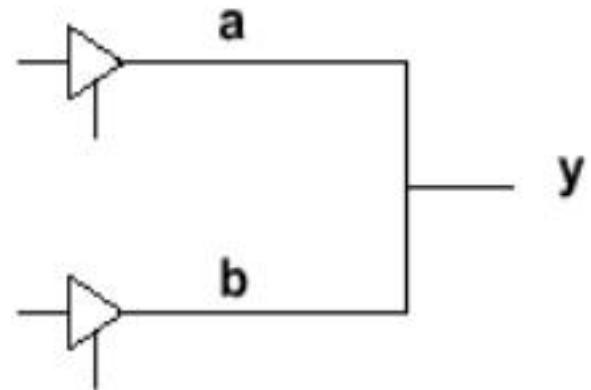
3.1.2 线网类型

- **wire**类型是最常用的类型，用于模块或逻辑门之间的物理连线，只有连接功能。
- 驱动端信号的改变会立刻传递到输出的连线上。
- **wire**和**tri**类型有相同的功能。用户可根据需要将线网定义为**wire**或**tri**以提高可读性。
- 例如，可以用**tri**类型表示一个**net**有多个驱动源。或者将一个**net**声明为**tri**以指示这个**net**可以是高阻态Z。可推广至**wand**和**triand**、**wor**和**trior**。
- **triereg**类型很象**wire**类型，但**triereg**类型在没有驱动时保持以前的值。这个值的强度随时间减弱。
- **wand**、**wor**逻辑功能与**wire**的区别见下页表。



3.1.2 线网类型

- net类逻辑冲突的决断
- Verilog有预定义的决断函数
- 支持与工艺无关的逻辑冲突决断



Wire/Tri

$\begin{smallmatrix} b \\ \backslash a \end{smallmatrix}$	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

y

Wand/Triand

$\begin{smallmatrix} b \\ \backslash a \end{smallmatrix}$	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

y

Wor/Trior

$\begin{smallmatrix} b \\ \backslash a \end{smallmatrix}$	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

y



3.1.2 线网类型（续）

■ wire型变量

- 最常用的net型变量，常以assign关键字指定组合逻辑信号
- 模块中的输入/输出信号类型声明缺省为wire型
- 可用做任何方式的输入，也可以用做“assign”语句或实例元件的输出。

■ 线网类型端口定义规则

`<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。



3.1.2 线网类型

举例

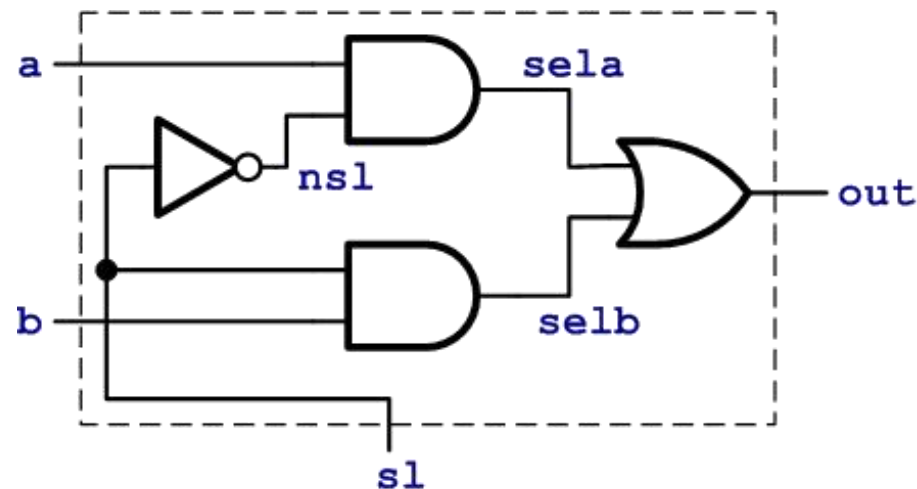
```
wire d; // 默认为1 bit连线
wire [7:0] x, y, z; // x,y,z都是8 bit的wire型连线
tri [7:0] data_bus; // 8-bit三态线网
wire signed [7: 0] result; // 带符号的线网
wire #(2.4,1.8) carry; // 带上升、下降延迟的线网
wire [15:0] (strong1,pull0) sum = a + b;
// 16-bit 带驱动强度的隐式连续赋值语句（后续讲解）
```

例3.2： MUX门级线网结构（我们尝试勾勒出逻辑电路图？）

```
module mux2_1(out,in1,in2,sel);
output out;
input in1,in2,sel;
wire sel_ba,and_out1,and_out2;
```

```
    not u1(sel_ba,sel);
    and u2(and_out1,in1,sel_ba);
    and u3(and_out2,in2,sel);
    or u4(out,and_out1,and_out2);
```

```
endmodule
```





3.1 数据类型

■ 3.1.3 变量声明

Verilog语言中另一重要数据类型是**变量**，使用前变量需要进行声明。变量是数据存储单元的抽象，可以存储数据并能赋值给其它**变量**或**线网**。

变量类型	初始值
reg, time, integer	x
real, realtime	0.0

❖ 变量声明：

```
reg a, b, c;
```

//三个标量（1bit）的变量

```
reg signed[7:0] d1, d2;
```

//两个8bit符号变量

```
integer i, j;
```

//两个符号的整数变量

```
real r1, r2;
```

//两个双精度变量



3.1.3 变量声明

■ 线网类型和变量类型的比较

线网类型	变量类型
模块实例或 原始实例的驱动	变量仅存逻辑值， 不存储强度
连接到被声明模块的输入 或输出端	过程赋值的左侧是变量类型
在连续赋值的左边	有符号的浮点数，双精度



3.1.4 寄存器类型

- 寄存器类型是一种变量类型，其赋新值以前保持原数值的。
- 寄存器是数据存储单元的抽象(不是真正的硬件)，通过赋值语句可以改变寄存器存储值，相当于高级语言中的变量。
- 与线网数据类型不同，寄存器类型不需要数据源持续驱动就可以保持原数值。
- 寄存器类型即可以声明时序逻辑器件，也可声明组合逻辑。

变量	功能
reg	无符号整数变量，可以选择不同的位宽（标量或矢量）
integer	有符号整数变量，32位宽，2的补码的算术运算。
real	双精度有符号的浮点数，用法与integer相同。
time	无符号整数变量，64位宽（Verilog-XL仿真工具用）
realtime	与real内容一致，可用作实数仿真时间的保存与处理



3.1.4 寄存器类型

■ reg型变量声明

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

`reg_type`: 寄存器类型

`range`: 矢量范围, `[MSB: LSB]`格式。仅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

■ 例如

`reg a; // 一个标量寄存器`

`reg [3: 0] v; // 从MSB到LSB的4位寄存器向量`

`reg [7: 0] m, n; // 两个8位寄存器`



3.1.4 寄存器类型

- reg型变量与 net型的根本区别：
 - reg型变量需要被明确地赋值，并且在被重新赋值前一直保持原值。
 - reg型变量必须通过过程赋值语句赋值！不能通过assign语句赋值！
 - 在过程块内被赋值的每个信号必须定义成reg型！
- 寄存器类型和线网类型的数据均可声明为向量（位宽大于1bit）。



3.1.5 阵列（数组类型）

Verilog HDL语法允许声明 reg, integer, time, realtime 及向量类型的阵列，可以声明多维数组。阵列中的元素可以是标量或矢量，阵列可以是线网、变量、整数或表达式。

```
wire [7:0] w_array [4:0]; // 5个8-bit 线网组成的阵列
```

用数组时一次只能对一个元素进行操作，而不能如向量那样同时对连续的几个位进行操作。表示数组某个元素时，允许使用变量来表示元素的索引，如 numb[i]=12，但表示向量时则不可以使用变量表示。

Verilog通常用寄存器型变量建立的一维数组来描述存储器，可以是RAM、ROM存储器和寄存器数组。定义格式为：

```
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



3.1.5 阵列（数组类型）（续）

```
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
```

需要注意，对存储器进行地址索引的表达式必须是常数表达式。**Verilog-1995不支持多维数组**，也就是说，只能对存储器字进行寻址，而不能对存储器中一个字的位寻址。**Verilog-2001以后规则支持多维数组**。

尽管数组类型和寄存器型数据的定义都是寄存器，但二者还是有很大区别的。一个n位的寄存器可以在一条赋值语句中直接进行赋值，而一个存储器数组则不行。

例：一个由n个1位寄存器构成的存储器是不同于一个n位寄存器的。

```
reg    [n-1 : 0]    rega;    // 一个n位的寄存器型数据类型  
reg    memb        [n-1 : 0]; // 一个由n个1位寄存器成的存储器数组
```



3.1.5 阵列（数组类型）（续）

- `integer number [0:99];`
 //声明一个有100个元素的整数数组
- `number [25] = 1234;`
 //将1234赋值给第26个元素
- `reg [7:0] my_input [65535:0];`
 //声明一个有65536个元素的8-bit位宽的寄存器
- `my_input [97] = 8'b10110101;`
 //将10110101分别赋值给第98个元素的7至0位
- `reg my_reg [0:3][0:4];`
 //声明一个具有20个元素的二维寄存器数组
- `my_reg [1][2] = 1'b1;`
 //将1赋值给二维数组的第2行、第3列元素



3.1 数据类型

■ 3.1.6 标量与矢量

位宽是1bit的线网和寄存器类型数据被定义为标量，而位宽大于1bit的线网和寄存器类型数据被定义为矢量。

```
wire n;           //默认为 1bit 标量的线网变量类型
reg regn;         //默认为 1bit 标量的寄存器变量类型
wire [7:0]        // 8bit 位宽的矢量线网变量
reg [n-1:0] rega; // nbit 矢量的寄存器型数据类型
```

矢量通过关键字与变量名之间的方括号的高位/低位表示其变量的位宽[高位: 低位]，最左边的位定义为MSB，最右边的位定义为LSB。在矢量变量声明中可以通过关键字vected和scalared进行扩展声明。



3.1.6 标量与矢量（自学）

矢量的常量和变量

对于Verilog-1995的IEEE标准，可以选择向量的任何一位作为输出，也可以选择向量的连续几位输出，不过此时连续几位的始末数值的index**需要是常量**。其表达式如下：

```
vect[msb_expr : lsb_expr];
```

//其中msb_expr和lsb_expr必须是常量表达式

对于verilog-2001的标准中，index**可以使用变量，并可以进行组选择**（part_select）的功能

[<start_bit> + : width_expr]从起始位开始递增，位宽为width_expr
[<start_bit> - : width_expr]从起始位开始递减，位宽为width_expr

其中，start_bit可以是变量，而width_expr**必须**是常量。其中，
+：表示由start_bit向上递增width_expr位，-：表示由start_bit向下递减width_expr位。



3.1 数据类型

■ 3.1.7 参数

在Verilog HDL 语言中使用关键字**parameter**来定义模块中保持不变的常量，即用参数来取代一个常数，提高程序的可读性和易维护性。

parameter 参数名1 = 表达式, 参数名2 = 表达式, ...;

```
module mod_param (out, ln1, ln2);  
parameter      width1 = 16,width2=8;  
output [width1-1:0] bus;  
output [width2-1:0] data;  
endmodule
```

- ✓ 参数型常量经常用于定义延迟时间和变量**位宽**。
- ✓ 在模块或实例引用时，可通过**参数传递**改变在被引用模块或实例中已定义的参数。
- ✓ 可用**字符串**表示的任何地方，都可以用定义的参数来代替。
- ✓ **参数是本地的，其定义只在本模块内有效**。（区别`define）



3.1.7 参数（自学）

- 关键字specparam声明指定参数，其目的仅用于提供定时和延迟值。
- Verilog语法中还包含局部参数，即关键字localparam。局部参数的值不可改变
- 参数重载（overriding）
 - defparam语句在编译时重载参数值；
 - 一个模块可以出现多个defparam语句；
 - Defparam语句可以覆盖任何参数；
 - defparam语句引用参数的层次化名称；
 - 使用defparam语句可单独重载任何参数值；
 - 局部参数可以用localparam关键字声明，它不能够进行参数重载。



3.1.7 参数

■ defparam语句（现在综合工具还不支持）

```
module test;  
parameter Id_num = 0;  
  
endmodule  
  
module top;  
//改变引用的实例模块中的参数值  
defparam w1.Id_num = 1, w2.Id_num = 2;  
//调用两个实例模块  
hello_world w1();  
hello_world w2();  
  
endmodule
```



3.2 Verilog语言中的端口



3.2.1 端口命名

3.2.2 端口声明

3.2.3 连接方式



3.2 端口

- 端口是所设计的模块与外界环境实现信号交互的接口，它属于标识符。所设计模块的端口语法表述如下：

module 模块名（端口1，端口2，端口3，……）；

```
module test (a,c,e,g,h);  
input [7:0] a;  
input signed [7:0] c;  
output [7:0] e;  
output signed [7:0] g;  
  
wire [7:0] c; // 线网型带符号输入端口  
reg [7:0] g; // 寄存器型带符号输出端口  
  
endmodule
```



3.2.1 端口命名

端口命名规则需要统一，有序信号命名的整体要求为：被命名端口标识符具有一定的意义，简介易懂，且项目命名规则唯一。**简化的端口名不能与Verilog HDL中的关键字相同！**

例：

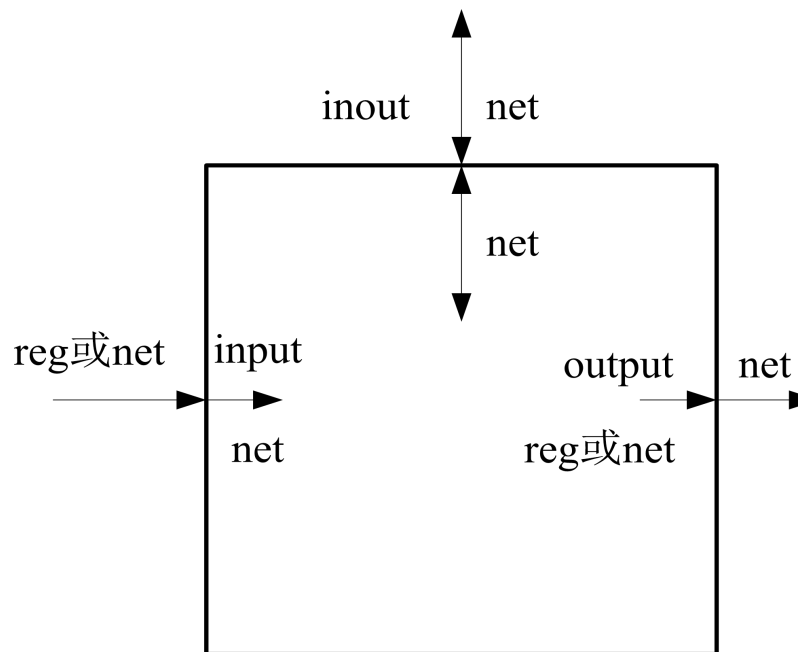
```
input [3:0]          sys_out, sys_in;
input               clk, clk_freq_50m,
clk_freq_200m;
input               reset, enable;
output[15:0]         aes_out;
output              driver_out;
```



3.2.2 端口声明

端口列表中的所有端口必须在模块中进行声明，Verilog中的端口具有三种类型。端口声明中也涉及到四个方面。

数据方向	数据类型	端口位宽	符号
input 输入端口	线网型	[n:0]	signed/ unsigned
output 输出端口	线网型或 寄存器型	[n:0]	signed/ unsigned
inout 双向端口	线网型	[n-1:0]	signed/ unsigned



在Verilog HDL中，所有端口的数据类型隐含时声明为线网型，如果端口信号仅实现信号连接功能，则无需再声明为线网数据类型。但当输出端口需要保存数值时，就必须将其声明为寄存器数据类型。



3.2.3 端口连接方式

■ 方式1：端口位置连接

端口位置连接规则：调用模块的端口名必须与被调用模块端口列表中的位置保持一致。格式如下：

模块名 模块实例名 (连接端口1 的端口名, 连接端口2的
端口名, 连接端口3的端口名, ...);



```
Block1 dut1(in_a, in_b, in_c, ou_d, ou_e);
```



3.2 端口连接方式

■ 连接方式2： 端口命名连接

端口命名连接规则：各模块端口与外界信号按模块名进行连接。格式如下：

模块名 模块实例名(.被调模块端口名1(调用模块内端口名1), .被调模块端口名2(调用模块内端口名2),被调模块端口名n(调用模块内端口名n));



点不可少

- 端口命名连接不必严格按照端口的顺序对应书写，可以提高程序的**可读性**和**可移植性**；
- 并且此方法可以**悬空**某个端口而不用特别说明。



3.2.3 端口命名连接（续）

置空端口：

- 在实例语句中，置空端口是表达式所对应的模块端口为空置状态，即未与其他模块端口相连的空置端。
- 未与其它模块端口相连的置空端，输入端置空时其值为**高阻态 z** 。
- 输出端置空时，表示该输出端口未与其它模块连接使用。

端口位宽：

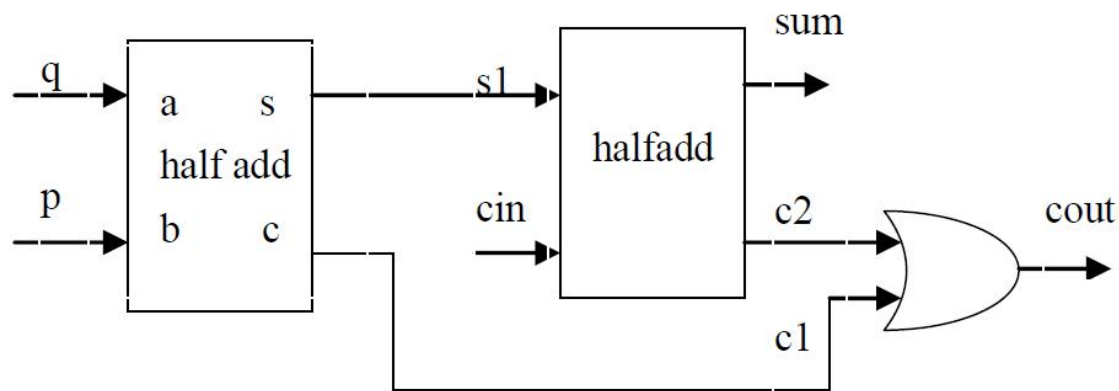
- 当主调模块端口和被调用模块端口的长度不同时，端口通过无符号数的**右对齐或截断方式**进行匹配。



3.2 端口

■ 例3.21: 全加器

```
module full(p,q,cin,sum,cout); //:
input p,q,cin;
output sum,cout;
    wire s1,c1,c2;
```



//两个实例调用语句

```
half      h1 (p,q,s1,c1); //端口位置连接。
half      h2 (.c(cin),.s(sum),.a(s1),.b(c2)); //端口命名连接。
or        h3 (cout, c1,c2) ;
```

endmodule

```
module half(b,a,s,c); //被调用的半加器模块
input a,b;
output s,c;
    //求和描述语句;
    //进位描述语句;
endmodule
```



第三章、Verilog 高级语法与要素

3.3 Verilog语言中的表达式

3.3.1 运算表达

3.3.2 操作符

3.3.3 操作数

3.3.4 主要标准差别



3.3 表达式

■ 3.3.1 运算表达

包括逻辑运算和算术运算。Verilog HDL语法中规定运算表达式由语法定义的操作符和操作数组成，目的是根据操作符的意义对操作数进行操作得到计算结果。

表达式的所有中间结果应取**最大**操作数的长度（赋值时，此规则也包括左端目标）。

`a && c ;` //表示对操作数a和c进行的**逻辑与**操作

`in1 + in2 ;` //表示对操作数in1和in2进行相加的算术操作

`12 / 3 ;` // 除法操作



3.3.2 操作符

■ 单目操作符

包括位与、位或、位异或、位同或、位非、一元逐位加减。

若其位宽不相等，则对较短操作数左边补零，使其位宽相等。取反操作符只有一个操作数，是对操作数逐位取反。

在Verilog HDL 语法中，负数采用二进制补码表示，建议采用整数或实数形式表示负数。单目操作符还包括一元缩减操作符。以实现单比特结果而执行的逐位操作运算。

一元缩减操作符	&	~&		~	^	~^
4'b0000	0	1	0	1	0	1
4'b0110	0	1	1	0	0	1
4'b01xx	0	1	1	0	x	x
4'b01z0	0	1	1	0	x	x

避免使用<nn>'<base><number>格式带来的错误，计算中将会转换为无符号的2的补码形式。



3.3.2 操作符

■ 算术操作符

部分单目操作符也是算术操作符，算术操作符按如下执行

- ①将负数赋值给 reg 或其他无符号变量使用二进制补码；
- ②如果操作数的某一位是x或z，则结果为x；
- ③在整数除法中，余数舍弃；
- ④模运算中使用第一个操作数的符号。

```
rega = 3; regb = 4'b1010;
int = -3;    //int = 1111.....1111_1101
ans = 5 * int;    // ans = -15
ans = (int + 5)/ 2;    // ans = 1
ans = 5/ int;    // ans = -1
num = rega + regb;    // num = 1101
num = rega + 1;    // num = 0100
num = int;    // num = 1101
num = regb % rega;    // num = 1
```

```
10%3 = 1;
-10%3=-1;
11%-3=2;    //符号以第一个数为准
-4' d12 %3 = 1
//补码计算过程1100 → 1011 → 0100
```

```
result = base ** exponent
//指数操作运算
```



3.3.2 操作符

■ 逻辑操作符

包括逻辑与&&，逻辑或||，逻辑非！

其得到的结果为逻辑数值0，1，x。分别表示假，真，未定。对于非零操作数，等同于逻辑真，为零的操作数，等同于逻辑假。操作数中任意一位为x，z，**其等同于**逻辑不确定x。

```
rega = 4'b0011;
```

//逻辑值为 “1”

```
regb = 4'b10xz;
```

//逻辑值为 “1”

```
regc = 4'b0z0x;
```

//逻辑值为 “x”



```
ans = rega && 0;
```

// ans = 0

```
ans = rega || 0;
```

// ans = 1

```
ans = regb && rega;
```

// ans = 1

```
ans = regc || 0;
```

// ans = x

```
ans = !rega
```

//ans = 0 逻辑反

```
ans = ~rega
```

//ans = 1100 按位取反



3.3.2操作符

■ 关系操作符

大于	“ $>$ ”	小于	“ $<$ ”
大于等于	“ $\geq$ ”	小于等于	“ $\leq$ ”

关系为假，返回值为 0，关系为真，返回值为 1；操作数的值不确定，则返回值不确定。关系操作符的优先级低于算术操作符的优先级

例：

$a > b + 1$ 等同于 $a > (b + 1)$



3.3.2 操作符

■ 等价操作符

对符号两侧的操作数进行逐位对比的操作。它的优先级低于关系操作符。

$a == b$	a逻辑等于 b
$a != b$	a逻辑不等于b
$a === b$	a全等 b, 包括 x和 z
$a !== b$	a不全等 b, 包括 x和 z

当用全等和非全等操作符比较时，多操作数中出现的确定值x和高阻值z也要进行比较，当两个操作数**完全一致**时，结果为1，否则为0。

$a == b$, $a != b$ 在遇到操作数中某些不确定值x和高阻值z时，其结果会出现不确定值x。

**全等操作符和不全等操作符
一般情况下都不可综合。**



3.3.2 操作符

■ 移位操作符

包括逻辑移位和算术移位。**逻辑移位**时，向量产生的空余位用0填充；**算术移位**时根据内容确定空余位补充值，表达式为正数时，补充0，表达式为负数时，补充1。

算术左移	<<<	逻辑左移	<<
算术右移	>>>	逻辑右移	>>

A = 4'b1010

Z = a >> 2;

//z结果是0010，右移2位，最高位用0补充

Z = a << 1;

//z 结果是0100，左移1位，最低位用0补充

start = -10;

result = (start >>> 3); //最高位用1补充，得到4'b1110，结果为-2，



3.3.2 操作符

条件操作符
优先级最低

■ 条件操作符

条件操作符根据条件表达式的值选择表达式：

$\langle \text{条件表达式} \rangle ? \langle \text{真表达式} \rangle : \langle \text{假表达式} \rangle ;$

首先计算条件表达式，如果结果为真，则选择问号后的表达式，若为假，则选择冒号后面的表达式。如果为不确定态x，且真、假两个表达式不相等，则输出**不确定值**。

条件表达式可以嵌套使用，即真、假表达式本身可以是一个完整的条件操作表达式。



3.3.2 操作符

■ 其它操作符

包含赋值操作符、拼接操作符、重复操作符等。

赋值操作符包含：

非阻塞（non-blocking）赋值方式：

赋值符号为 $\leq$ ，如 $b \leq a$ ；

阻塞（blocking）赋值方式：

赋值符号为 $=$ ，如 $b = a$ ；

它的优先级较低，往往最后读取，它影响逻辑赋值关系表达，也影响电路结构和时序关系。



3.3.2 操作符

拼接操作符 { }

将多个操作数拼接在一起组成新的操作数。被拼接的操作数位宽必须确定，变量类型可以是变量线网或寄存器、向量线网或寄存器、位选、组选和确定位宽的常数。

```
rega = 8'b0000_0011;  
regb = 8'b0000_0100;  
regc = 8'b0001_1000;  
regd = 8'b1110_0000;
```

```
out = {regc[4: 3], regd[7: 5], regb[2], rega[1: 0]};
```

```
// out = 8'b11111111
```

```
{b, {3{a, b}}}
```

```
// 相当于 {b, a, b, a, b, a, b}，即(a,b)重复拼接3次
```



3.3.2 操作符（续）

触发操作符 “ $\rightarrow$ ” 和识别操作符 “@”

Verilog中，命名事件控制通过关键字event声明变量类型，实现触发事件和识别事件两种功能：

- 1.它可以发生在任何特定的时间。
- 2.没有持续的时间消耗。

```
parameter d = 50; // d 声明为变量
reg [7:0] r;      // r 被声明 8-bit reg

begin
    #d r = 'h35;
    #d r = 'hF7;
    #d  $\rightarrow$  end_wave;
    //触发事件end_wave
end
```

```
fork
    @enable_a                //识别命名事件
begin
    #ta wa = 0;
    #ta wa = 1;
    #ta wa = 0;
end
    @enable_b                //识别命名事件
begin
    #tb wb = 1;
end
join
```



3.3.2.9 操作符的优先级

优先级别	操作符	说明
最高	! ~ & ~& ~ ^ ~^ + -	逻辑非、按位取反、 一元缩减、 一元逐位加减
	* / ** %	乘、除、指数、取模
依次递减	+ -	算术加、减
	<< >>	移位
	< > <= >=	关系
	= = != = = = != =	等价
	& ~&	按位、与/与非
	^ ^~ ~^	按位、异或/同或
	~	按位、或/或非
	&&	逻辑与
		逻辑或
	{,} ({n{m}})	拼接、重复
最低	?:	条件操作



3.3 表达式

■ 3.3.3 操作数

Verilog语言中规定的数据类型都可以是操作数，可以是常数、线网、寄存器、位选、组选、存储器，函数调用等。

整数在操作数中的表达式分为三种形式：

无符号、无进制基底的整数（如：8、200）；

无符号、有进制基底的整数（如：`d8、`b1100）；

有符号、有进制基底的整数（如：4`d8、-`d12）。

一个无基底的整数解释为有符号的2补码的格式，无符号基底整数解释为无符号值。

位选是向量线网或向量寄存器的某一位，**组选**是向量线网或向量寄存器的一组选定的位。



3.3.3 操作数

■ integer inta;

- `inta = -12 / 3;` // The result is -4.
- `inta = -'d 12 / 3;` // The result is 1431655761.
- `inta = -'sd 12 / 3;` // The result is -4.
- `inta = -4'sd 12 / 3;` (这个结果多少? 思考一下)

// -4'sd12 is the negative of the 4-bit

// quantity 1100, which is -4. $-(-4) = 4$.

// The result is 1.