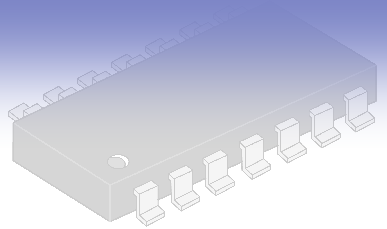
第 3 章

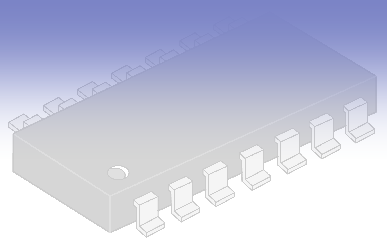
Verilog 的语言基础

内容



- 模块的结构
- 基本值
- 数据类型
- 操作数
- 操作符
- 表达式

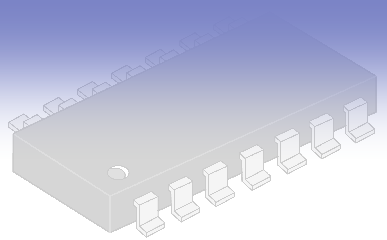
Verilog 模块的结构



- Verilog 模块的结构

```
module module_name( port_list );
    Declaration:
        input,output, inout,
        wire, reg, parameter,
        task, function,...
    Statement:
        Initial statement;
        Continuous assignmet;
        Always statement;
        Generate statement;
        Module instantiation;
        Gate instantiation;
        UDP instantiation;
endmodule
```

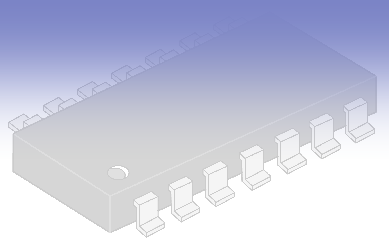
模块名、端口和内部信号（变量）



- 使用标识符表示模块名、端口和内部信号（变量）
- 标识符（identifier）
  - 任意一组字母、数字、$符号和下划线（ _ ）符号的组合
  - 第一个字母必须是字母或下划线
  - **标识符区分字母的大小写**
  - **标识符不能使用 Verilog 的关键字（参见教材：附录C，P263）**
- 例
  - Count
  - COUNT　　//与 Count 不同
  - _R1_D2
  - READY$
- 注释 —— 与 C++ 相同
  - 两种形式
    - /* 第一种，可以扩展至
    - 　多行*/

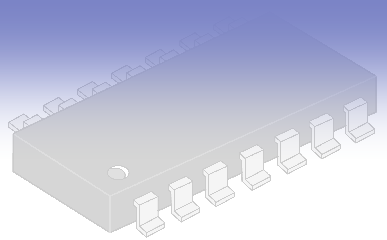
    - // 第二种，在本行结束

内部信号声明



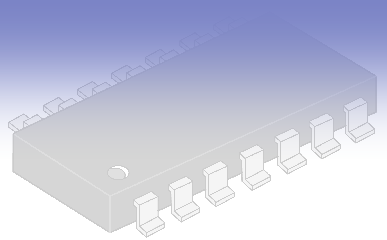
- 内部信号（变量）
  - 线网型（wire）
    - 相当于连线
    - 组合逻辑电路的内部逻辑值
  - 寄存器型（reg）
    - 相当于电路内部存储单元的值
    - 时序逻辑电路中各种触发器的状态
- 信号（变量）的声明
  - reg [BITS - 1 : 0] 变量1，变量2，变量3，，…；
  - wire [BITS - 1 : 0] 变量1，变量2，变量3，，…；

基本值



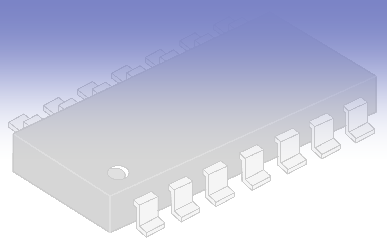
- Verilog HDL 有 4 种基本值
  - ①**0**：逻辑 0 或 false
  - ②**1**：逻辑 1 或 true
  - ③**x**：未知（或不确定状态）
  - ④**z**：高阻，浮动状态
- 4 种基本值内置于 Verilog 语言中
  - 即：
    - 0 值：指的是逻辑 0
    - x 值：不确定状态
    - z 值：总是意味着高阻抗
  - **x 和 z 不分大小写**
  - 即：
    - 0x1z 与 0X1Z 相同
  - 线网（**wire**）类型的变量的默认值为 z
  - 寄存器（**reg**）类型的变量的默认值为 x
  - **wire** 和 **reg** 型信号（变量）的默认位宽为 1

常量



- 三类常量
  - ①整型 ——常量中的数字可以包含4种基本值
    - 即： `a=8'b1x0z_1101`
  - ②实数型
  - ③字符串型
- 整数有两种书写格式
  - ①简单的十进制数格式
  - **②基数格式**
  - 十进制数格式
    - 带有一个可选的“+”（一元）或“—”（一元）操作符的数字序列
    - 例、
    - 32：十进制正 32
    - +32：十进制正 32
    - -16：十进制负 16

整数表示法

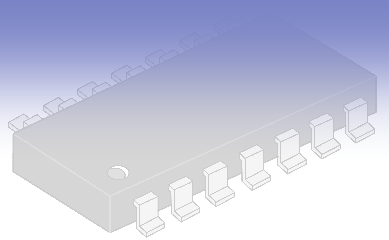


- 整数基数格式为：

  *[size]’[**signed**]* <base><value>

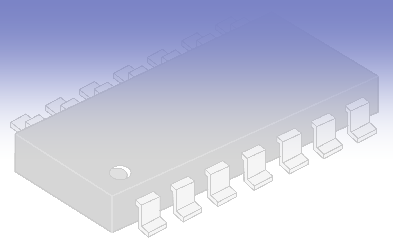
  ［位宽］’［有符号］〈进制〉 〈数值〉
- size（位宽）
  - 定义常量用二进制表示的数字位数长度（位宽）
- signed（有符号数标志）
  - s/S：表示有/无符号数，如果有 s/S，则为有符号数，否则，无符号数
- base（进制）
  - d/D：十进制
  - b/B：二进制
  - o/O(ou)：八进制
  - h/H：十六进制
  - 缺省为十进制
- value（数值）
  - 基于 base 的值的数字序列
- 值 x 和 z 以及十六进制中 a 到 f 不分大小写

整数的表示



- 例、
  - 5’O37　　// 5 位 8 进制数
  - 4’D2　　// 4 位 10 进制数
  - 4’B1x_01　　// 4 位 2 进制数
  - 8’**s**h51　　// 8 位有符号 16 进制数，0101_0001 = 81(dec)
  - 6’**s**o72　　// 6 位有符号 8 进制数，111_010 = -6 (dec)
  - 4’d-2　　// **非法**，数值部分不能为负
  - 8□’h□2A　　// 合法，在位宽和字符之间，以及基数和数值之间
    // 允许出现空格
  - 3’□b001　　// **非法**，’ 和基数 b 之间，不允许出现空格
  - (2+3)’b10　　// **非法**，位宽不能使用表达式

整数的表示

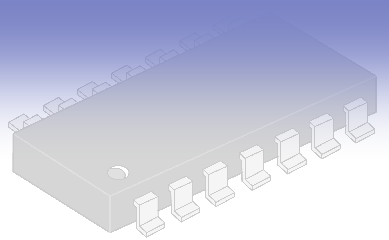


- x 值和 z 值
  - 4'b1x_01 // 4 位二进制
  - 4'bX0 // 4 位二进制数，低位起第 2 位为不定值，相当于 4'bxxx0
  - 4'hZ // 4 位十六进制高阻，相当于 4'bzzzz

```
`timescale 1ns/1ns
module ex00;
    reg [3:0] a, b, c;
    initial begin
        a = 4'b1x_01;     //    4 位二进制
        #5 b = 4'bX0;     //    4 位二进制数，低位起第 2 位为不定值
        #5 c = 4'hZ;      //    4 位十六进制高阻，相当于 4'bzzzz
    end

    initial
        $monitor("At time t=%4t, a=%4b, b=%4b, c=%4b", $time, a, b, c);
endmodule
```

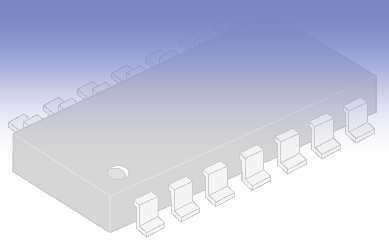
```
# At time t=    0, a=1x01, b=xxxx, c=xxxx
# At time t=    5, a=1x01, b=xxx0, c=xxxx
# At time t=   10, a=1x01, b=xxx0, c=zzzz
```

整数的表示



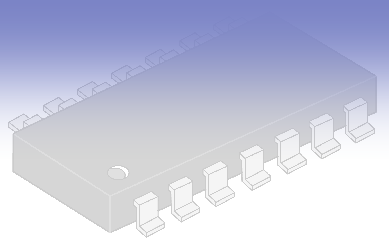
- ? —— 可替代 z 值
  - 12’d?　　// 12 位十进制数，其值为高阻
  - 4’h?　　// 4 位十六进制高阻，相当于 4’bzzzz
- 下划线（underscore_）
  - 可以用在整数或实数中
  - 不能作为首字符，不能出现在位宽和进制处
  - 本身没有意义，仅用来提高易读性
- 例、
  - 16’d1010_1011_1100_0001　　// 合法，16 位二进制数
  - 8’b_0101_1100　　// 非法

默认位宽



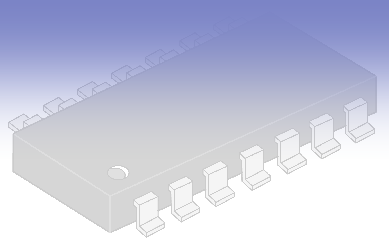
- 常量
  - 不说明位数时，默认的位宽与使用的计算机和仿真器有关
    - 在 32 位操作系统平台上，一般是 32 位
- 例、 32 位系统平台上，默认 32 位
  - 10 = 32’d10 = 32’b1010
  - 1 = 32’d1 = 32’b1
  - -1 = -32d1 = 32’hFFFF_FFFF
  - ’BX = 32’bxxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx

实数



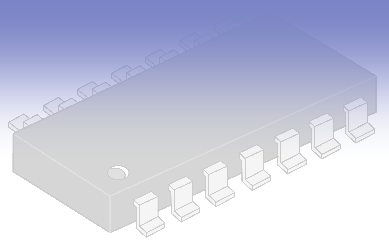
- 两种形式定义
  - 十进制计数
  - 例、
    - 2.0　　// 合法
    - 3.1415926　　// 合法
    - 0.1　　// 合法
    - 6.　　// 非法，小数点两侧必须有一位数字
  - 科学计数法
    - 12_5.1e2　　// 忽略下划线，值为 12510.0
    - 3.6E2　　// e 与 E 同
    - 5E-4　　// 0.0005

字符串（1）



- 双引号（“字符串”）内的字符序列
- 字符串不能分成多行书写
- 每个字母用 8 位 ASCII 值表示
  - 保存在 reg 类型的变量中
  - 可以看作是无符号整数
- 例1、
  - “AB” = 16’B0100_0001_0100_0010
- 例2、存储字符串：Verilog HDL
  - reg [1: 8*11] message;
  - message = “Verilog HDL”;

字符串（2）

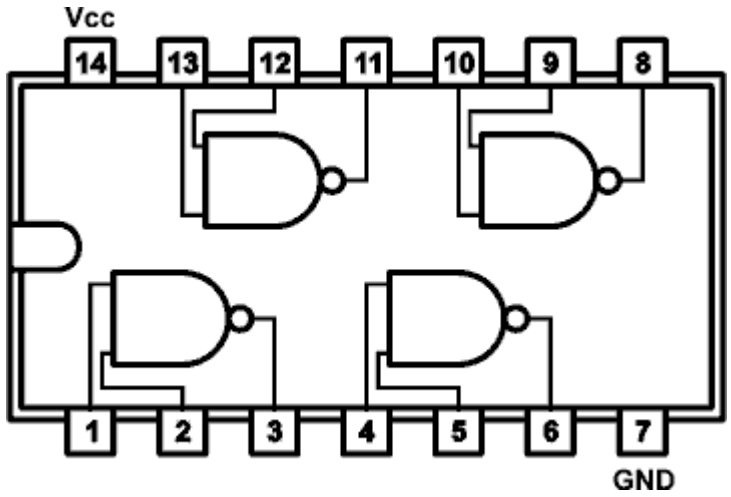


- 反斜线（\）用于特殊字符
  - \n　　// 换行
  - \t　　// 制表
  - \\　　// 字符 \
  - \”　　// 字符 ”
- \ooo　　// 1 到 3 个 8 进制数字字符
  - 例
    - \206　　// 8 进制数 206 对应的字符

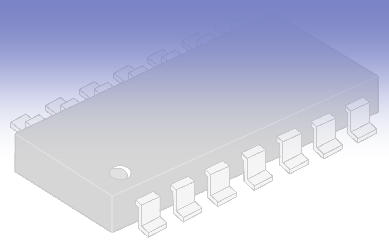
参数（1）

- 参数代表常量
  - 使用关键字 parameter 在模块内定义一个用标识符代表的常量
  - 提高程序的可读性和可维护性
- 模块实例的参数值可以重载
  - 通过参数重载可以对模块实例进行定制
  - 比较——Verilog 的 parameter 与 C++ 中的 const

```
 2   module comb( y, a, b );
 3      parameter N = 4;
 4
 5      output [N-1:0] y;
 6      input  [N-1:0] a, b;
 7
 8      assign y = ~(a & b);
 9
10   endmodule
11
12   module Top;
13
14       wire [7:0] p_y;
15
16       reg [7:0] p_a, p_b;
17
18       comb #(.N(8)) m_comb( .y(p_y), .a(p_a), .b(p_b) );
19   endmodule
```

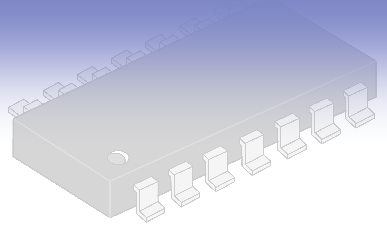


参数（2）



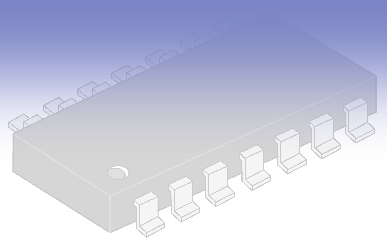
- 参数型常量说明格式：
  - parameter 参数名1 = 表达式，参数名2 = 表达式，…，参数名n = 表达式;
- 注
  - 参数说明由关键字和赋值语句表组成
  - 每个赋值语句用逗号（，）分开
  - 赋值语句右边的表达式 —— 必须是常数表达式
    - 只能是数字和已经定义过的参数

例、参数型常量的应用



- 参数型常量常用于定义延迟时间和变量宽度
- 例、
  - parameter msb = 7;
  - parameter a = 28, b = 29, c = 30, d = 31;
  - parameter pi = 3.1415926;
  - parameter BYTE_SIZE = 8, WORD_SIZE = 64;
  - parameter word_msb = WORD_SIZE - 1;
  - parameter average_delay = (a + b + c +d) / 4;

局部参数

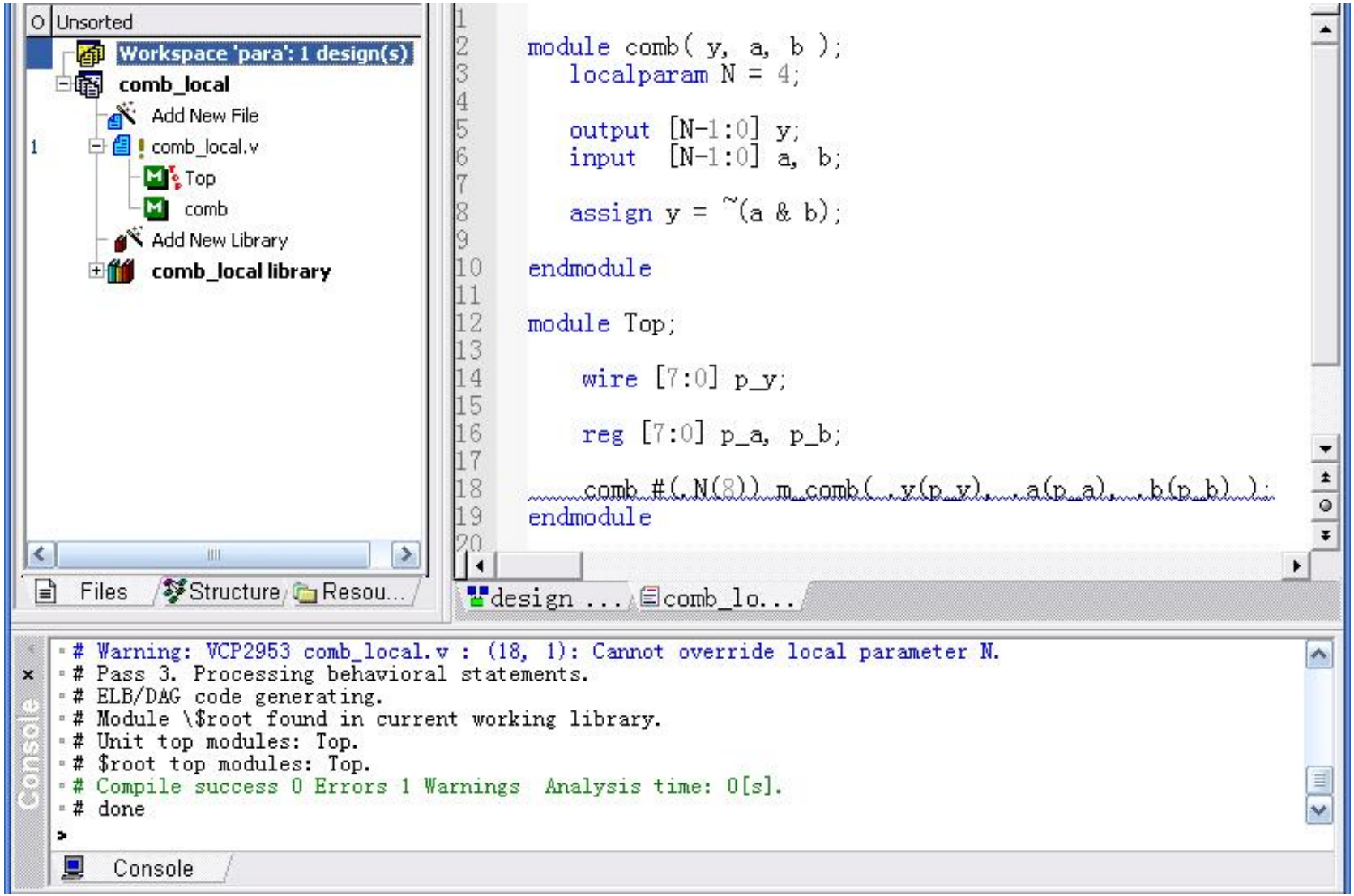


- 使用关键字 localparam 定义
  - 作用等同于 parameter
  - 其值不能重载修改
- 例、使用局部参数定义状态机的状态编码
  - 状态机的状态编码是不允许修改的

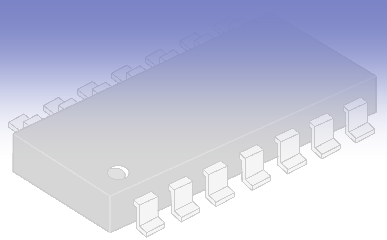
```
localparam state1 = 4'b0001,
           state2 = 4'b0010,
           state3 = 4'b0100,
           state4 = 4'b1000;
```

局部参数

- 其值不能重载修改

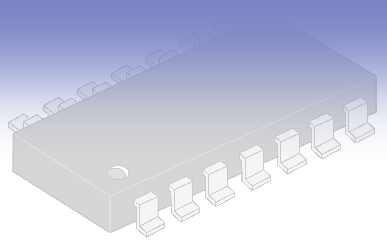


数据类型



- Verilog HDL 中的变量有两大数据类型
  - 线网类型（net type）
    - 表示 Verilog 结构化元件间的物理连线
    - 必须接受驱动源（门、assign）的驱动
    - 如果没有驱动源连接到线网类型的变量上，该变量为高阻，其值为 z
  - 寄存器类型（register type）
    - 表示一个抽象的数据存储单元
    - 只能在 initial、always 语句中被赋值，并保存下来
    - 缺省值是 **x**

线网类型的说明语法



- 格式：
  - **net_kind [MSB : LSB] *net1, net2, net3, net$_n$***
  - **net_kind**：线网类型的一种
  - **MSB : LSB** ：定义 线网宽度的范围的常量表达式，范围是可选的，缺省位宽是 1
- 例、
  - wire ready, start; // 2 个 1 位连线
- wire 类型
  - assign 语句的输出
  - 实例引用语句的输出
  - Verilog 模块的中的输入/输出端口的默认类型
  - 任何逻辑表达式中的输入
    - 例：
      - `assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;`

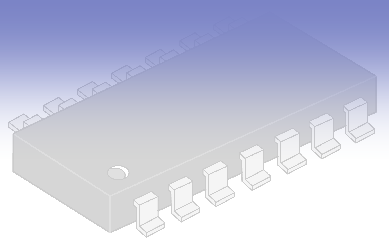
wire 和 tri 线网

- wire 和 tri 语法和功能相同
  - wire 用于描述单个门或连续赋值驱动的连线（net）线网类型
  - tri 用于描述多个驱动源驱动同连线（net）的线网类型
    - Two different names are used for more readability. Preferably ***wire*** nets may be used **if a net has only on driver**. If a net has **more than one driver**, then ***tri*** net may be used instead.
  - 多个驱动源驱动一个连线（或三态线），线网的有效值见下表决定

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

- 例、
  - `tri y;`
  - ……
  - `assign y = a & b;`
  - ……
  - `assign y = a ^ b;`
- 这里
  - y 有两个驱动源，应当声明为 tri 类型
  - 如果 a & b = 01x，且 a ^ b =11z，
  - 则 y = **x1x**

reg 寄存器类型



- 数据存储单元的抽象——最常见的寄存器类型
- 格式
  - **reg** [**MSB : LSB**] *reg1, reg2, reg3, … , reg$_N$*
  - **[MSB : LSB]**
    - 定义了位宽的范围，MSB 和 LSB 均为常数表达式
    - 范围是可选的，缺省值为 1
  - 寄存器类型的数据可以取任意长度
  - 寄存器类型的变量中的值被解释为无符号数
  - 负数以其二进制补码保存
  - reg 类型变量的默认初始值为 x
- 例
  - reg cnt;
  - reg [1 : 8] comb
  - reg [3 : 0] tag;

```
tag = -2;    // tag 中保存的数值是：4'd14（1110），1110 是 -2 的补码
             // 1_010→1_101（取反）→1110（加1）
tag = 5 ;    // tag中保存的数值是： 5（0101）
```

带符号的 reg 寄存器类型

- 当寄存器类型变量声明为有符号类型变量时，可用于带符号算术运算
- 例、
  - **reg signed** [63:0] m; // 64位带符号的变量

```
module negativenumbers;
    reg [4:0] x;
    reg [4:0] y;
    initial begin
        x = 5;
        $display("x = %5b", x);
        y = -x;
        $display("y = %5b", y);
        $display("y = %d",  y);
        $finish;
    end
endmodule
```

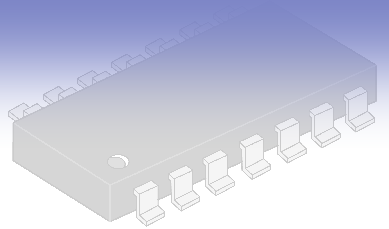
输出：x = 00101
y = 11011 // 11011 是 -5（101） 的补码
y = 27

```
module negativenumbers;
  reg [4:0] x;
  reg signed [4:0] y;

  initial begin

    x = 5;
    $display("x = %5b", x );

    y = -x;
    $display("y = %d", y );

    $finish;
  end
endmodule
```

输出：x = 00101
y = -5 // y 的内部表示仍是 11011，
// -5 的补码

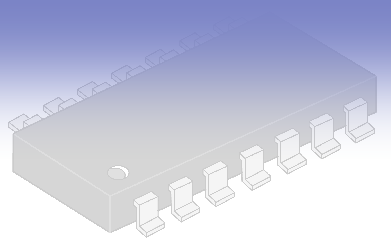
如果使用： $display("y = %4b", y);
输出： y= 11011

带符号的 wire 类型



```
reg         [15:0] a; // Unsigned

reg signed  [15:0] b;

wire signed [16:0] signed_a;

wire signed [31:0] a_mult_b;


assign signed_a = a; // Convert to signed

assign a_mult_b = signed_a * b
```

部分（位）选择（Vector Part Select ）



- 部分选择
  - 可以指定选择多位线宽的变量的 1 位和若干个相邻位
- 例

声明：

```
wire [7:0]  bus;                    // 8-bit 总线
wire [31:0] busA, busB, busC;       // 3 条 32-bit 总线

reg [0:40]  addr;                   // 41-bit 宽地址
```

部分选择：

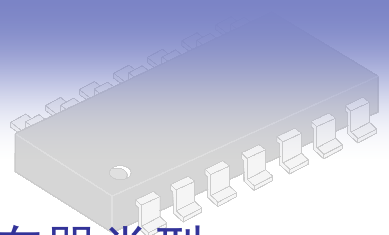
```
busA[7]     // 指定选择 busA 的第 7 位
bus[2:0]    // bus 的最低 3 位
            // 不能使用 bus[0:2]，因为 bus 的声明是高位在左
addr[0:1] // addr 的两个最高位
```

可变部分（位）选择（Variable Vector Part Select ）

- **[<starting_bit>+ : width]** —— 从起始位开始递增，位宽为 **width**
- **[<starting_bit>- : width]** —— 从起始位开始递减，位宽为 **width**
- 起始位可以是一个变量，位宽必须是一个常量

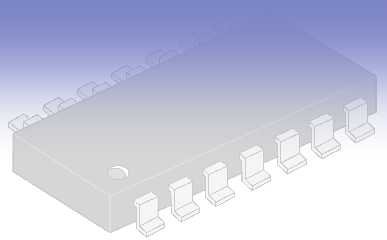
```
reg [255:0] data1;  // data1[255]是最高有效位
reg [0:255] data2;  // data2[0]是最高有效位
reg [7:0]   byte;

// 部分选择
byte = data1[31-:8];  // starting bit = 31, width =8 => data1[31:24]
byte = data1[24+:8];  // starting bit = 24, width =8 => data1[31:24]
byte = data2[31-:8];  // starting bit = 31, width =8 => data2[24:31]
byte = data2[24+:8];  // starting bit = 24, width =8 => data2[24:31]

// 起始位是变量，位宽必须是常数，如，使用循环语句选择一个向量的所有位
for ( j=0; j<=31; j=j+1 )
    byte = data1[(j*8)+ : 8];  // 依次选择 [7:0], [15:8]...[255:248]

// 用于初始化一个向量的一部分，如果 byteNum = 1, 部分向量 [15:8] 被清零
data1[(byteNum*8)+:8] = 8'b0;
```

整数、实数和时间寄存器类型



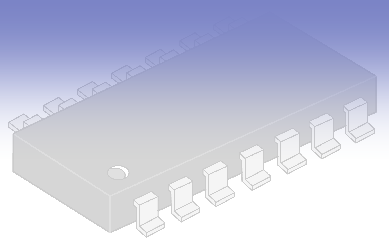
- 除 reg 之外，还有 integer、real、 time 和 realtime 4 种寄存器类型
- integer
  - 32-bit 有符号补码（2's complement）
  - 例、整数声明
  - **integer** a, b;
- real —— 64-bit real number
  - 例、 声明实数
  - **real** c, d;
  - 实数缺省初始值为 0
- time
  - 一般为 64-bit 无符号，相当于 64-bit reg 型数据
  - 例、
  - **time** last_chng; // time value
  - $display("At %t, value=%d",$time,val_now)
- realtime
  - realtime 与 real 相同，完全可以互换使用
  - 例、 声明实数和实数时间变量
  - **real** float ; // a variable to store real value
  - **realtime** rtime ; // a variable to store time as a real value

存储器（memory）类型



- 存储器是一个寄存器数组
- 说明格式：
  - `reg [MSB : LSB] memory1[upper1:lower1],`
    `memory2[upper2:lower2], ... ;`
- 这里
  - `[MSB : LSB]`
    - 定义了存储器中每个存储单元的大小
  - `[upper1:lower1]`
    - 定义了存储器中每个存储单元的个数
- 例、
  - parameter WSIZE = 64, MSIZE = 256;
  - reg [WSIZE – 1 : 0] mem [MSIZE – 1 : 0], wblock, rblock;

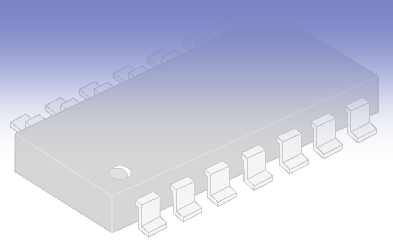
例、存储器类型



- reg [7 : 0] mem [0 : 63]; // mem 为 64 x 8 位存储器
- reg [7 : 0] count; // count 是一个 8 位寄存器类型变量
- count = 0 ; // 合法
- mem = 0; // 非法赋值
- mem[1] = 0; // 合法，对 mem 的第 2 个单元赋值

- 使用地址索引选择存储器的的单元
  - 索引越界或计算时遇到 x 或 z，则部分选择的值为 x
- 可以对数组元素部分位选择
- 例：
  - reg [15 : 0] d_ram [127:0]; // d_ram 为 128x16 位存储器
  - reg [7 : 0] group; tag
  - group = d_ram[51][11:4];
  - tag = d_ram[100][10]; // 选择的是什么？

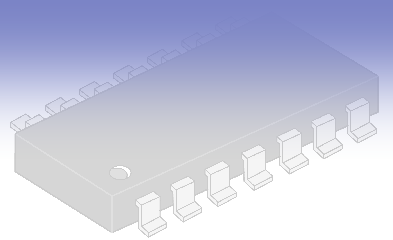
各种类型的数组



- 可以声明 reg, integer, time, real, realtime 及wire类型的数组

```
integer count[0:7];             // 8 个计数变量组成的数组
reg bool[31:0];                 // 32 个 1-bit 寄存器类型变量
time chk_point[1:100];          // 100 个 time 类型寄存器变量
reg [4:0] port_id[0:7];         // 8 个位宽为 5 的端口标识符
integer matrix[4:0][0:255];     // 2 维整数变量

// 4 维 64-bit 寄存器类型数组
reg [63:0] array_4d [15:0][7:0][7:0][255:0];

wire [7:0] w_array2 [5:0];      // 8 位线宽线网型变量数组
wire w_array1[7:0][5:0];        // 2 维 1 位线宽的线网型变量数组
```

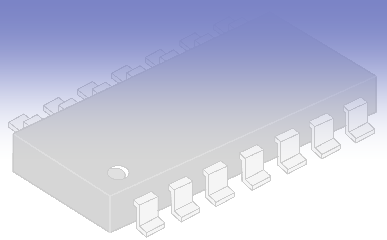
数组元素的引用



- <array_name>[<subscript>]

```
count[5] = 0;          // 把整型数组 count 的第 5 个元素置 0
chk_point[100] = 0; // 把时间类型数组的第 100 个元素（64位）复位
port_id[3] = 0;       // port_id 的第 3 个元素（5位）置零

matrix[1][0] = 33559; // 整型数组的元素（32位）赋值 33559

// 4 维寄存器类型数组的元素 array_4d[0][0][0][0]（64位）的
// 低 16 位 15: 0 置零
array_4d[0][0][0][0][15:0] = 0;

port_id = 0;      // 非法，对整个数组赋值
matrix [1] = 0; // 非法，对数组的行赋值（第 2 行）
```

表达式

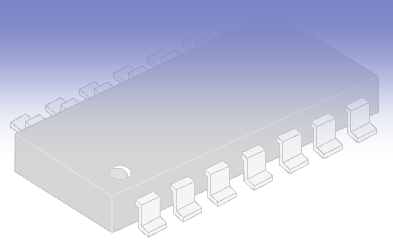


- 由操作数（operand）和操作符（operator）组成
- 可以在出现数值的任何地方使用

- 操作数
  - ① 常数
  - ② 参数
  - ③ 线网类型变量
  - ④ 寄存器类型变量
  - ⑤ 位选择
  - ⑥ 部分（位）选择
  - ⑦ 存储器单元
  - **⑧ 函数调用**

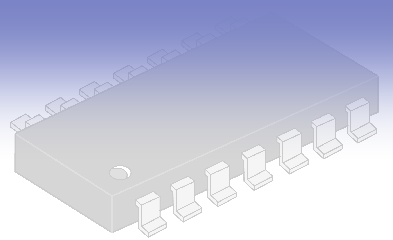
- 操作（运算）符
  - ① 算术
  - ② 关系
  - ③ 相等
  - ④ 逻辑
  - ⑤ 按位
  - ⑥ 移位
  - ⑦ 条件
  - **⑧ 拼接和复制**
  - **⑨ 缩减**

3 种运算符



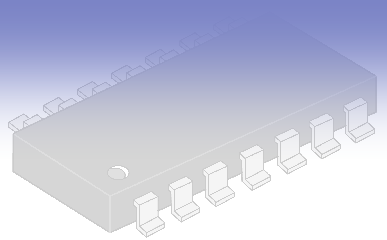
- 单目
  - 带一个操作数，操作数在运算符的右边
  - 例、
    - clock = ~clock　　// ~ 取反，单目运算符，clock 是操作数
- 双目
  - 带两个操作数，操作数在运算符的两边
  - 例、
    - y = a & b　　// & 按位与（and），双目运算符，a、b 是操作数
- 三目
  - 带三个操作数，由三目运算符分隔开
  - 例、
    - y = condition ? a : b　　// ? : 三目运算符，condition、a、b 是操作数

算术运算符



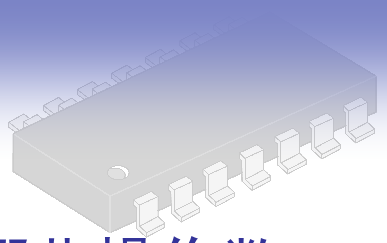
- +
  - 双目加法，单目正值
- -
  - 双目减法，单目负值
- *
  - 乘法
- /
  - 除法，**整数除法**，截断任何小数部分
    - 7/4　　// 结果为1
- %
  - 取模（或：求余），要求两个操作数均为整数，求出与第一个操作数符号相同的余数
    - 7 % 4　　// 结果为 3
    - -7 % 4　　// 结果为 -3
    - 11 % -3　　// 结果为 2
- 如果算术运算符中的任意操作数是 x 或 z，则整个结果为 x
- 例、
  - 5'b0_10**X**1 +5 'b0_1111 = 5'b**x_xxxx**
  - 5'b1_**Z**010 + 5'b0_0010 = 5'b**x_xxxx;**

算术运算结果的长度



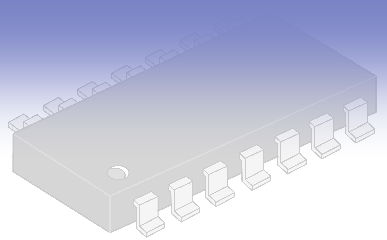
- （独立）表达式的运算结果的长度由最长的操作数决定
  - 如： if (a + b)
- 在赋值语句中，由左边目标的长度决定
- 例、
  - reg [3 : 0] a, b, c;
  - reg [5 : 0] F;
  - c = a + b; // 结果长度为 4 位，相加溢出部分被丢弃
  - F = a + b; // 结果长度位 6 位
- 如何确定中间结果长度
- 例、
  - wire [4 : 1] a, b;
  - wire [5 : 1] c;
  - wire [6 : 1] d;
  - wire [8 : 1] F;
  - F = (a + c) + (b + d);
  - 右边操作数最长的是 6 位，将左端包含在内，最大为 8，则使用 8 位进行运算
  - 即：
    - 如果 (a + c) + (b + d) 是一个独立的表达式， (a + c) 的结果为 6 位
    - 在这个赋值语句中，(a + c) 的结果为 8 位

无符号和有符号数



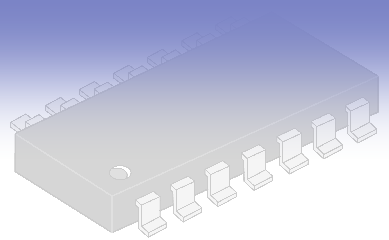
- 进行算术运算应当注意哪些操作数是无符号数，哪些操作数是有符号数
- 无符号数
  - 线网（wire）
  - 普通寄存器（reg）
- 有符号数
  - 整数类型 ： integer
  - 有符号寄存器类型：reg signed
  - 有符号线网类型： wire signed
- 表达式类型规则（IEEE Std）
  - 表达式类型只依赖于操作数，不依赖于其左边被赋值变量（LHS）
  - **十进制数是有符号数**
  - 带符号基数格式表示的整数是有符号数，反之，是无符号数
    - 例、4'sd12
  - 位选择和部分位选择是无符号数

无符号和有符号数举例



- 例、
  - reg [5 : 0] bar;
  - integer tab;
  - bar = -4’d12; // 寄存器变量 bar 的十进制数为 -12， 二进制表示为 110100
  - tab = -4’d12; // 整数变量 tab 的十进制数为 -12，
  - 这里假设，整数默认为32位，二进制补码表示为：
  - 32’b1111_1111_1111_1111_1111_1111_1111_0100
- 注：
  - bar 是普通寄存器型变量，只存无符号数
    - 右端的表达式的值为 ’ b11_0100（12 的二进制补码），读出数值为 52
      - 12 dec = 00_1100 bin ➔ 11_0011 + 1 = 11_0100
  - tab 是整型变量，可以存储十进制数 -12，
    - 在 tab 中存储的位向量为 ：
    - 32’b1111_1111_1111_1111_1111_1111_1111_0100
    - 读出的数值为 -12

位运算符



- 电路信号有4种状态值
  - 1，0，x，z
- Verilog 有 5 种位运算——对信号值的二进制位操作

① ~
- 一元非，按位取反（not）

② &
- 二元与，按位与（and）

③ |
- 二元或，按位或（or）

④ ^
- 二元异或，按位异或（xor）

⑤ ^~
- 二元异或非，按位同或

位运算规则

~ 非	0	1	x	z
	1	0	x	x

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

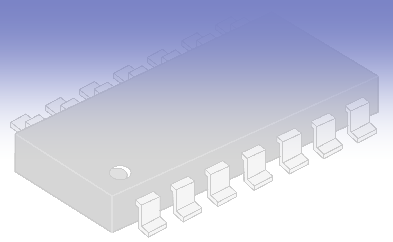
^ 异或	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

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

^~ 异或非	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

- 对应的位进行运算
- 例、位运算
  - 设
    - A = 'b0110
    - B = 'b0100
  - 则
    - A | B　结果为：0110
    - A & B　结果为：0100
- 如果操作数长度不同，长度较小的操作数在最左边添 0 补位
  - 例、
    - 'b0110 ^ 'b100010
    - 即：
    - 'b000110 ^ 'b100010
    - 结果为：100100

逻辑运算符



- 逻辑运算用于条件判断
  - **相当于总是 1 位二进制位运算**
- 3 种逻辑运算符
  - && 逻辑与
  - || 逻辑或
  - ! 逻辑非
- 这些运算符在逻辑值0（假，false）或1（真，true）操作
- 例、
  - 设 blue = 'b0; // false
  - green = 'b1; // true
  - 则
    - blue && green // 结果为 0，（假)
    - blue || green // 结果为 1，（真)
    - ! blue // 结果为 0，（假)

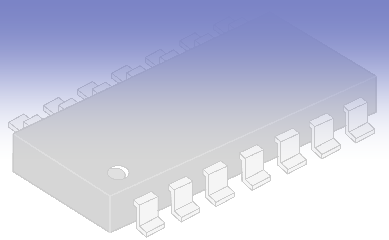
逻辑非的真值

a	!a
0	1
1	0

逻辑与/或的真值

b	a	a && b	a || b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

向量的逻辑运算



- 对于向量的逻辑运算，非0（零）向量作为 1 处理
- 例
  - bus_a = 'b0110;
  - bus_b = 'b0001;
- 则
  - bus_a || bus_b　　// 结果为 1
  - bus_a && bus_b　　// 结果为 1
- 且
  - ! bus_a　　// 结果为 0
  - ! bus_b　　// 结果为 0
- 即，! bus_a 与 ! bus_b 结果相同

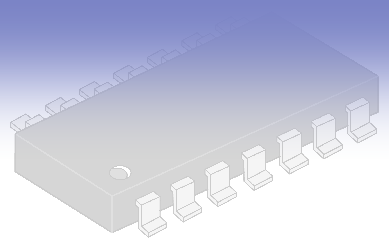
位运算和向量的逻辑运算

- 位运算 —— 表达式的运算结果的长度由最长的操作数决定，在赋值语句中，由左边目标的长度决定
- 向量的逻辑运算 —— 对于向量的逻辑运算，非0（零）向量作为 1 处理

```
# At time t=    0, val_a||val_b = 1, val_a&&val_b = 1
# At time t=    2, val_a=0110, val_b=0001, a=0001, b=0001
# At time t=    7, val_a=0110, val_b=0001, c=0000, d=0000
# At time t=   12, flag=x, e=000x, f=xxxx
```

```
// File: ex02.v
`timescale 1ns/1ns
module ex02;
  reg flag;
  reg [3:0] val_a, val_b;
  reg [3:0] a, b, c, d, e, f;
  initial begin
    val_a = 'b0110;
    val_b = 'b0001;
    $display("At time t=%4t, val_a||val_b = %b, val_a&&val_b = %b",
             $time, ( val_a||val_b), (val_a&&val_b));
    #2 a = val_a || val_b;    b = val_a && val_b;
    $display("At time t=%4t, val_a=%4b, val_b=%4b, a=%4b, b=%b",$time, val_a, val_b, a,b );
    #5 c = !val_a;    d = !val_b;
    $display("At time t=%4t, val_a=%4b, val_b=%4b, c=%4b, d=%b",$time, val_a, val_b, c,d );

    #5 flag = 1'bx;    e = !flag;    f  = a + e;
    $display("At time t=%4t, flag=%1b, e=%4b, f=%4b", $time, flag, e, f);
  end
endmodule
```

关系运算符



- 关系运算符有：
  - > 大于
  - < 小于
  - >= 不小于
  - <= 不大于
- 关系运算符的结果为真（1）或假（0）
- 如果操作数中有一位为 x 或 z，那么结果为 x
- 例
  - 23 > 45 // 结果为假（0）
  - ‘b01000 > ’b01100 // 结果为假（0）
- 关系运算符常和逻辑运算符配合使用
- 例
  - (a > b) && (u > y)
  - (! a) || (a > b)

相等关系（等式）运算符

- 4 种相等关系运算符
  - == 逻辑相等
  - != 逻辑不等
  - === 全等
  - !== 非全等
- 如果比较结果为假，则为0，否则为1
- 在逻辑比较中，如果两个操作数之一包含 x 或 z，结果为不定值 x
- 在全等（===）比较中，值 x 和 z 严格按位比较，如果两个操作数完全一致，其结果为 1，否则为 0
- 例，如果
  - data = 'b11x0;
  - addr = 'b11x0;
- 则
  - data == addr // 不确定，值为 x
  - data === addr // 为真，值为 1
- 如果操作数的长度不等，长度较小的操作数左边添 0 补位
- 相等关系运算符常和逻辑运算符配合使用
  - 如：
  - (a == b) || (q === r)

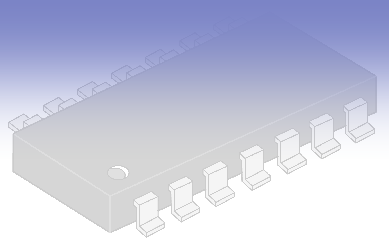
全等运算的真值

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

逻辑相等运算的真值

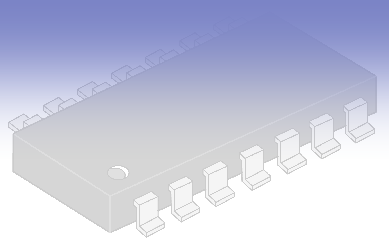
==	0	1	x	x
0	**1**	0	x	x
1	0	**1**	x	x
x	x	x	x	x
z	x	x	x	x

移位运算符



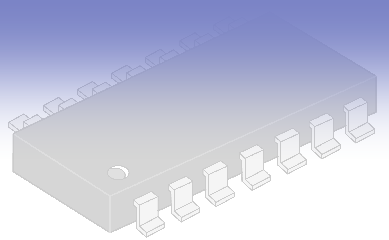
- 移位运算符有两种：
  - << 左移
  - >> 右移
- 使用方法
  - operand << n
  - operand >> n
  - 这里
    - operand —— 要进行移位的操作数
    - n —— 移位的次数
  - 用 0 填补移出的空位
  - 如果右侧的操作数的值为 x 或 z，移位操作的结果为 x
- 例
  - reg [7: 0] qr;
  - qr = 4’b0111; // qr = 8’b0000_0111
  - qr >> 2 // 结果是 8’b0000_0001
- 例
  - reg [4:0] a;
  - a = 4’b1001 << 1 // 结果是 5’b10010
  - reg [5:0] b;
  - b = 4’b1001 << 2 // 结果是 6’b100100

拼接（连接）运算符



- 拼接（concatation）—— 可以用作左值（LHS）
  - 将多个表达式的某些位合并成一个大的多位表达式
  - 使用方法：
    - { *expr1, expr2, … , exprN* }
  - 例
    - wire [7: 0] bus0;
    - assign bus0[7 : 4] = { bus0[0], bus0[1], bus0[2], bus0[3] };
      - // 以反转的顺序将低 4 位赋给高 4 位
    - assign bus0 = { bus0[3:0], bus0[7:4]};
      - // 高 4 位与低 4位交换
  - **拼接操作不允许使用非定长数**
    - 如
    - { bus0, 5}　// 非法，常数 5 的位数没有指定

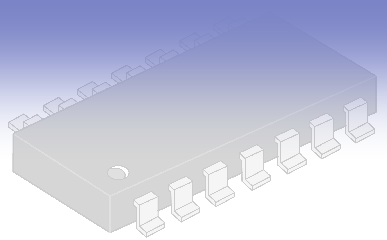
重复复制



重复复制

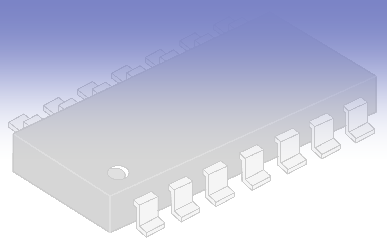
- 使用方法：
  - { *repetition_number* {*expr1, expr2, … , exprN* } }
- 例
  - wire abus[11: 0];
  - abus = { 3 { 4’b1011 }}; // 结果得到位向量 12’b1011_1011_1011
  - abus = { { 4{bus0[7] } }, bus0 }
  - { 4 { 1’b1 } } // 结果为 4’b1111
  - { 3 { ack } } // 等同于 { ack, ack, ack }

缩减（规约）运算符（1）



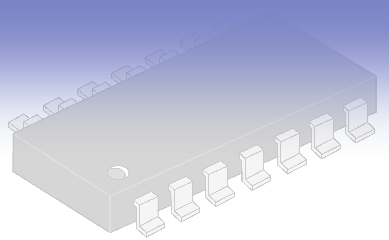
- 在单一操作数上的所有位上操作，并产生只有 1 位的结果
  - 计算过程
    1. 第 1 位与第 2 位进行运算
    2. 第 1 步的结果与第 3 位进行运算
    3. …
    4. 第 n-1 步的结果与第 n 位进行运算
- 缩减运算符有：
  - &
    - 缩减与
    - 如果操作数中存在值为 0 的位，则结果为 0
    - 如果存在值为 x 和 z的位，则结果为 x
    - 否则，结果为 1
  - ~&
    - 缩减与非
    - 与缩减运算符 & 的结果相反

缩减运算符（2）



- |
  - 缩减或
  - 如果操作数中存在值为 1 的位，则结果为 1
  - 如果存在值为 x 和 z的位，则结果为 x
  - 否则，结果为 0
- ~|
  - 缩减或非
  - 与缩减运算符 | 的结果相反
- ^
  - 缩减异或
  - 如果存在值为 x 和 z的位，则结果为 x
  - 如果操作数中有偶数个 1，则结果为 0
  - 否则，结果为 1
- ~^
  - 缩减异或非
  - 与缩减运算符 ^ 的结果相反

例、缩减运算



- 设
  - A = ’b0110;
  - B = ’b0100;
- 则
  - |B　　// 结果为 1
  - &B　　// 结果为 0
  - ~^A　　// 结果为 1
- 例、使用缩减异或运算（ ^ ）确定向量中是否有位为 x
  - MyReg = 4’b01x0;
  - ^ MyReg 的结果为 x
  - 检测方法
    - if (^MyReg === 1’bx)
    - 　　$display(“在向量中 MyReg 中有一个不确定位！ ”);

向量的逻辑运算举例

```
`timescale 1ns/1ns
module logic_eval();
    reg [3:0] a;
    reg [3:0] data;

    initial begin
        a = 4'b0;
        #5 a = 4'b1;
        #5 a = 4'bz0x1;
        #5 a = 4'b1z00;
        #5 a = 4'b0zx0;
        #5 a = 4'b0xx0;
        #5 a = 4'bx;
        #5 a = 4'bz;
    end

    always @(*) data = !(!a);

    initial
        $monitor("At %2t time, a=%4b, data=%4b", $time, a, data);
endmodule
```

```
# At  0 time, a=0000, data=0000
# At  5 time, a=0001, data=0001
# At 10 time, a=z0x1, data=0001
# At 15 time, a=1z00, data=0001
# At 20 time, a=0zx0, data=000x
# At 25 time, a=0xx0, data=000x
# At 30 time, a=xxxx, data=000x
# At 35 time, a=zzzz, data=000x
```

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

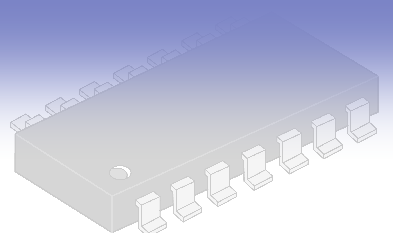
条件运算符

- 根据条件表达式的值选择结果表达式
- 形式如下：
  - *cond_expr* ? *expr1* : *expr2*
  - 如果 *cond_expr* 为真（值为 1 ），选择 *expr1*
  - 如果 *cond_expr* 为假（值为 0 ），选择 *expr2*
  - 如果 *cond_expr* 为 x 或 z，分别计算表达式 *expr1* 和 *expr2*
    - 逐位比较 *expr1* 和 *expr2*
    - 如果相等，则为该位结果值，否则为X，计算规则见右边表格
- 例
  - data_out = (a) ? 4'b110x : 4'b1000;
    - 如果 a 为真，则 data_out = 4’b110x
    - 如果 a 为假，则 data_out = 4’b1000
    - 如果 a 为 x，则 data_out = 4’b1x0x
    - 如果 a 为 z，则 data_out = 4’b1x0x

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

条件表达式不定态计算规则

条件运算符举例



```
`timescale 1ns/1ns
module cond_expr();
    reg [3:0] a;
    reg [3:0] data;

    initial begin
        a = 4'b0;
        #5 a = 4'b1;
        #5 a = 4'bz0x1;
        #5 a = 4'b1z00;
        #5 a = 4'b0zx0;
        #5 a = 4'b0xx0;
        #5 a = 4'bx;
        #5 a = 4'bz;
    end
    always @(*) data = (a) ? 4'b110x : 4'b1000;
    initial
        $monitor("At %2t time, a=%4b, data=%4b", $time, a, data);
endmodule
```

```
# At  0 time, a=0000, data=1000
# At  5 time, a=0001, data=110x
# At 10 time, a=z0x1, data=110x
# At 15 time, a=1z00, data=110x
# At 20 time, a=0zx0, data=1x0x
# At 25 time, a=0xx0, data=1x0x
# At 30 time, a=xxxx, data=1x0x
# At 35 time, a=zzzz, data=1x0x
```

各种运算符的优先级