

Verilog的数据类型

Verilog中值的种类

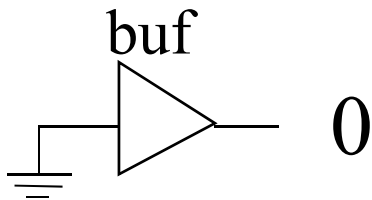
Verilog 使用了4值逻辑和八种信号强度来对实际的硬件电路建模，4值电平逻辑如下表：

值的级别	硬件电路中的条件
<i>0</i>	逻辑 0，条件为假
<i>1</i>	逻辑 1，条件为真
<i>x</i>	逻辑值不确定
<i>z</i>	高阻，浮动状态

Verilog 还是用强度值来解决数字电路中不同强度的驱动源之间的赋值冲突，逻辑 0 和 1 可以拥有如下表所示的强度值：

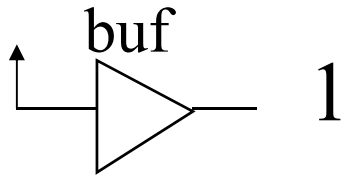
强度等级	类型	程度
<i>supply</i>	驱动	最强
<i>strong</i>	驱动	↑
<i>pull</i>	驱动	
<i>larger</i>	存储	
<i>weak</i>	驱动	
<i>medium</i>	存储	
<i>small</i>	驱动	↓
<i>highz</i>	驱动	最弱

Verilog 的四种逻辑值



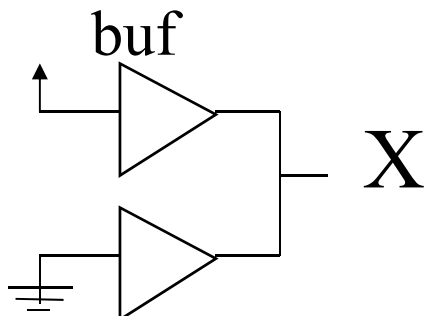
0

0、低、伪、逻辑低、地、**vss**、负插入



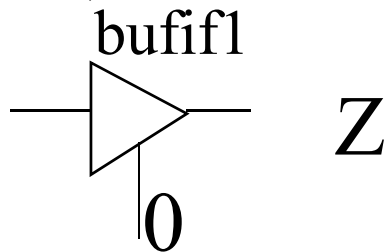
1

1、高、真、逻辑高、电源、**VDD**、正插入



X

X、不确定：逻辑冲突无法确定其逻辑值



Z

HiZ、高阻抗、三态、无驱动源

对于驱动源的强度信号的说明：

如果两个具有不同强度的信号驱动同一个线网，则竞争结果值为高强度的信号值。例如，两个强度分别为 **strong1** 和 **weak0** 的信号之间发生竞争，则结果服从 **strong1**；如果两个强度相同的信号之间发生竞争，则记为不确定；例如两个强度为 **strong1** 和 **strong0** 的信号发生竞争，则结果为 **x**。只有 **triereg** 类型的线网可以具有存储强度分为 **large, medium, small** 三个等级。一般来说对信号竞争、**mos** 器件、动态 **mos** 和其它底层器件的精度建模，强度具有很大作用。

变量的主要数据类型

主要有3类数据类型：

- **线网**(*net*): *net type* 表示 *Verilog* 结构化元件间的物理连线。它的值由驱动元件的值决定。
- **寄存器**(*register*): *register type*表示一个抽象的数据存储单元。并且它的值从一个赋值到另一个赋值被保存下来。
- **参数** (*parameters*): 运行时的常数(*run-time constants*)。

线网 (net)

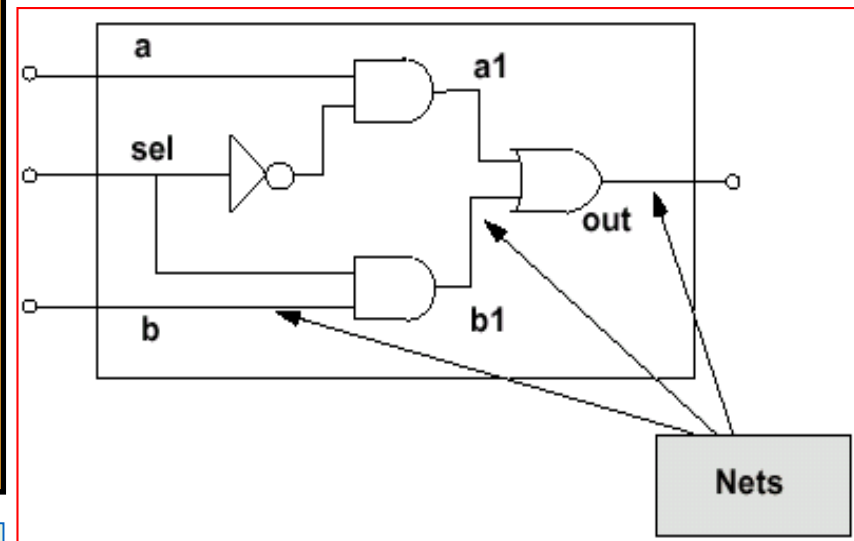
线网：表示器件之间的物理连接，线网类型的变量不能存储值，而且必须受到驱动器（门元件、模块输出端、连续赋值 *assign* 语句）的驱动。

注意：

(1) 当没有驱动器连接到网络类型的变量，则该变量就为高阻状态 (**z**)。

(2) 没有声明的 *net* 的缺省类型为 1 位（标量）*wire* 类型。

缺省时可用下面的编译向导修改：
``default_nettype <nettype>`



线网 (net)

*net*并非一个关键字，它是一组类型的，包括以下几种：

*wire*类型是最常用的类型，只有连接功能

综合编译器不支持的*net*类型

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。

Net 类型	功 能
<i>wire, tri</i>	线(缺省)
<i>supply1, supply0</i>	电源
<i>wor, trior</i>	三态线或
<i>wand, triand</i>	多驱动源线与
<i>triereg</i>	能保存电荷的 <i>net</i>
<i>tri1, tri0</i>	无驱动时上拉/下拉

wand, triand, wor 和 *trior* 与 *tri* 相似

Verilog中net和register声明语法

net 声明的语法格式如下:

<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。

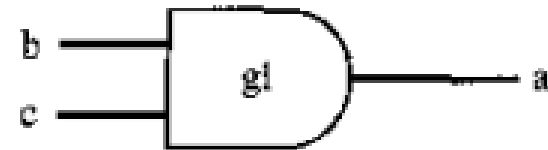
```
wire a;
```

```
//声明电路中a是 wire 类型
```

```
wire b, c;
```

```
//声明电路中b, c 是wire类型
```

```
wand [2:0] Addr;
```



寄存器类 (register)

寄存器类型用来表示存储元件，在赋新值以前保持原值。与线网不同寄存器不需要驱动源。寄存器类的四种数据类型如下：

寄存器类型	功能
-------	----

<i>reg</i>	可定义的非符号整数变量，可以是标量(1位)或矢量，是最常用的寄存器类型。
------------	--------------------------------------

<i>integer</i>	32 位有符号整数变量，算术操作产生二进制补码形式的结果。通常用作不会由硬件实现的的数据处理。
----------------	---

<i>real</i>	双精度的带符号浮点变量，用法与 <i>integer</i> 相同。
-------------	------------------------------------

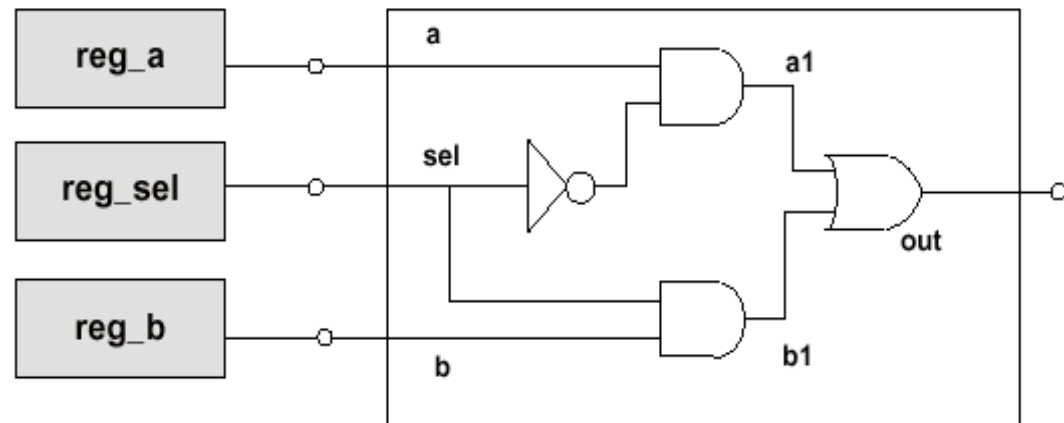
<i>time</i>	64 位非符号整数变量，用于仿真时间的保存与处理。
-------------	---------------------------

<i>realtime</i>	与 <i>real</i> 内容一致，但可以用作实数仿真时间的保存与处理。
-----------------	---------------------------------------

寄存器类 (register)

注意：

- (1) 不要混淆寄存器数据类型与结构级存储元件。
- (2) 寄存器类型的变量具有 **x** 的缺省值。
- (3) 寄存器类型多应用于行为模型描述及激励描述。
- (4) 它只能在 *always* 语句和 *initial* 语句中被赋值。
- (5) 实数和实数时间类型寄存器中的值被解释为有符号浮点数。



Verilog中net和register声明语法

register 声明的语法格式如下:

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

reg_type: *reg* 类型。

range: 矢量范围, 以 *[MSB: LSB]* 格式。

delay: 定义与 *net* 相关的延时。

reg_name: *reg* 名称, 一次可定义多个 *reg*, 用逗号分开。

举例: *reg a;* //一个标量寄存

reg [7: 0] m, n; // 两个8位寄存器

reg sign [63:0] m; //64位带符号的值

real delta; //可以在 *initial* 语句中赋值 *delta = 4e10*或 2.13

integer i; //32位的整型变量, $i = delta$; (2.13) $delta = 2$

time CurrTime; //*CurrTime*存储一个至少64位的时间

net和register举例

整型寄存器中
码数，而reg寄存
符号数。

```
reg [1:4] Comb;  
Comb = -2; //Comb  
Comb=5; // Comb
```

```
integer J;  
reg [3:0] Bcq;  
J = 6;          // J 的值为 32'b0000...00110。  
Bcq = J;        // Bcq的值为 4'b0110。  
Bcq = 4'b0101;  
J = Bcq;        // J 的值为 32'b0000...00101。  
J = -6;         // J 的值为 32'b1111...11010。
```

real 说明的变量的缺省值为0。不允许对real
声明值域、位界限或字节界限。当将值x和z赋
予real类型寄存器时，这些值作0处理。

```
realRamCnt;  
...  
RamCnt = 'b01x1Z;  
RamCnt在赋值后的值为'b01010
```

左边的位进

选择正确的数据类型

模块的端口。缺省的端口是，端口可被显新声明为 *reg* 型：器说明中，线网度相同。

```
Module Micro (PC, Instr, NextAddr);  
//端口说明  
input [3:1] PC;  
output [1:8] Instr;  
inout [16:1] NextAddr;  
//重新说明端口类型：  
wire [16:1] NextAddr;  
/* 该说明是可选的，但如果指定了，就必须与它的端口说明保持相同长度。*/  
reg [1:8] Instr;  
/* Instr 已被重新说明为 reg 类型，因此它能在 always 语句或在 initial 语句中赋值。*/  
...  
endmodule
```

选择正确的数据类型

输入端口可以由net/register驱动, 但输入端口只能是net

双向端口输入/输出只能是net类型

在过程块中只能给register类型赋值

输出端口可以是net/register类型, 输出端口只能驱动net

```
module top;  
  wire y;  
  reg a, b;  
  DUT u1 (y, a, b) ;  
  initial begin  
    a = 0; b = 0;  
    #5 a = 1;  
  end  
endmodule
```

```
module DUT (Y, A, B);  
  output Y;  
  input A, B;  
  wire Y, A, B;  
  and (Y, A, B) ;  
endmodule
```

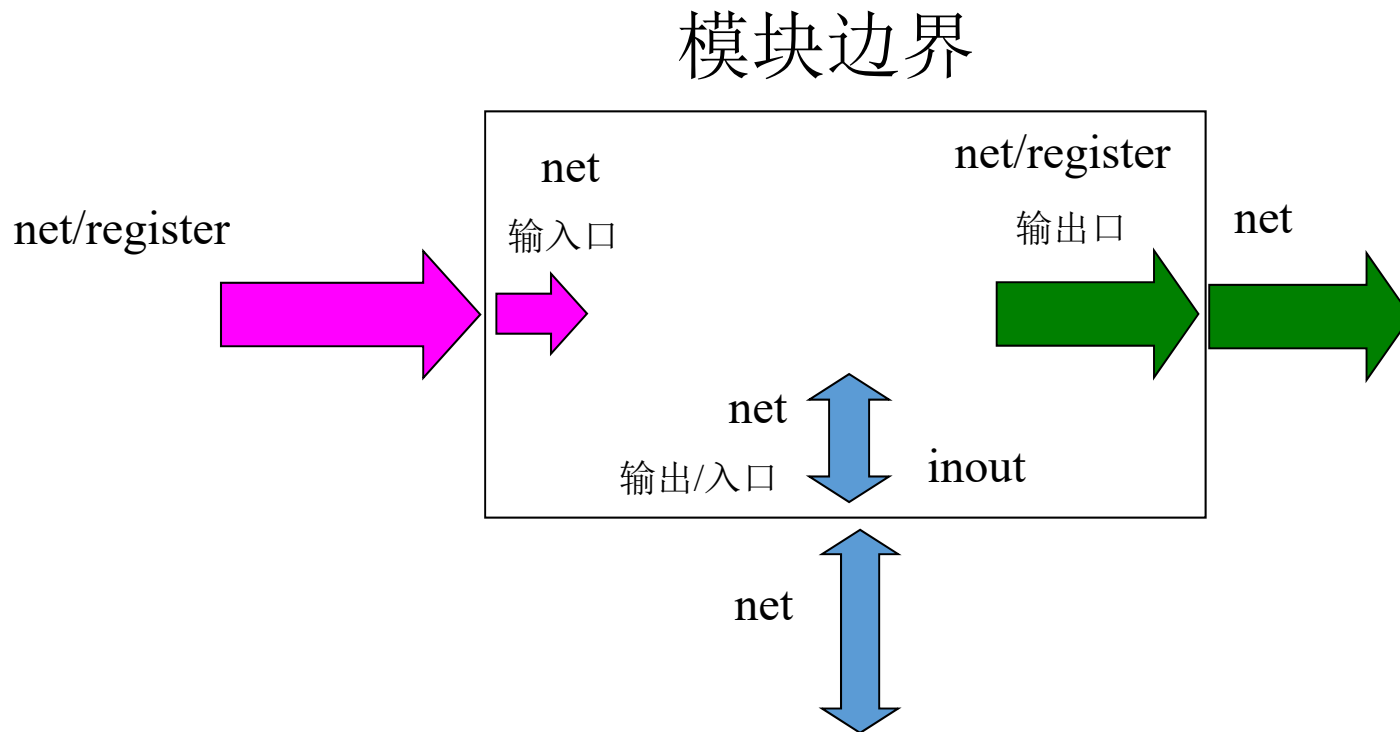
若Y, A, B说明为reg则会产生错误。

数据类型定义

输入输出端口定义规则:

- (1) 输入端口可以由 **net/register** 驱动，但输入端口只能是 **net**。
- (2) 输出端口是 **net/register** 类型，输出端口只能驱动 **net**。若在过程块中赋值，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。
- (3) 双向端口输入/输出只能是 **net** 类型。
- (4) 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。

上述规则可由右图来表示：



！ 注意：

若信号既需要在过程块中赋值，又需要在过程块外赋值（这种情况是有可能出现的，如决断信号），这时需要一个中间信号转换。

选择数据类型举例

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
```

选择数据类型时常犯的错误

- 在过程块中对变量赋值时，忘了把它定义为寄存器类型（reg）或已把它定义为连接类型了（wire）
- 把实例的输出连接出去时，把它定义为寄存器类型了
- 把模块的输入信号定义为寄存器类型了。

这是经常犯的三个错误！！！！

端口与外部信

```
module top;  
  //声明连接变量  
  reg [3:0] A, B;  
  reg C_IN;  
  wire [3:0] SUM;  
  wire C_OUT;  
  fulladd4 fa_ordered (SUM, C_OUT, A, B, C_IN);  
  fulladd4 fa_ordered(SUM, , A, B, C_IN);  
  fulladd4 fa_byname(.sum(SUM), .c_cout(C_OUT), .a(A),  
                    .b(B), .c_in(C_IN));  
  fulladd4 fa_byname(.sum(SUM), .a(A), .b(B), .c_in(C_IN));  
  ...  
endmodule
```

```
module fulladd4(sum, c_cout, a, b, c_in)  
  output [3:0] sum;  
  output c_out;  
  input [3:0] a, b;  
  input c_in;  
  ...  
endmodule
```

寄存器数组(Register Arrays)

- 在Verilog中允许声明reg, integer, time, real, realtime及其向量类型的数组，而对数组的位宽没有限制。其语法为：
register <数组名_name>

注意：

不要将数组和线网或寄存器向量混淆起来。向量是一个单独的元件，它的位宽n；数组由多个元件组成，其中的每个元件的位宽为n或1；

数组举例：

```
integer NUMS [7: 0]; // 包含8个integer型元素
time t_vals [3: 0]; // 4个time型元素
reg bool [31:0]; //32个1位布尔寄存器变量组成的数组
tri [15: 0] busa; // 16位三态总线
wire [0: 31] w1, w2; // 两个32位wire，MSB为bit0
integer matrix [4:0] [0:255]; //2维的整数型数组
reg [63:0] array_4d [15:0] [7:0] [7:0] [255:0];
//4维64位寄存器型数组
wire [7:0] w_array2 [5:0];
```

寄存器数组赋值

在赋值语句中需要注意如下区别：数组赋值不能在一条赋值语句中完成，但是寄存器可以。因此在存储器被赋值时，需要定义一个索引。

数组赋值举例：

```
count [5] = 0;      //把 count 数组中的第5个整数型单元复位
port_id[3] = 0;    //把port_id数组中的第3个寄存器型单元复位
matrix [1] [0] = 33559; // 把数组中第1行第0列的整数单元置
为33559
array_4d [0] [0] [0] [0] [15:0] = 0; /*把四维数组中索引号为
[0][0][0][0]的0-15位都置0 */
port_id = 0;       //非法，企图写整个数组
matrix [1] = 0;    //非法，企图写数组整个1行
```

存储器寻址(memory addressing)

Verilog 中，使用寄存器的一维数组来表示存储器，而存储器元素可以通过存储器索引（**index**）寻址，也就是给出元素在存储器的位置来寻址。其格式如下：

```
reg [n-1:0] <mem_name> [m-1:0]
```

其中，**n** 定义了存储器中每一个存储单元的大小，即该存储单元是一个 **n** 位的寄存器；**m** 定义了该存储器中有多少个这样的寄存器

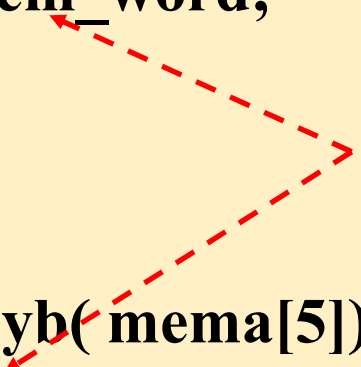
注意：

Verilog 只能对存储器字进行寻址，而不能对存储器中一个字的位进行寻址。

存储器寻址举例

存储器寻址只是寄存器数组赋值的特例：

```
module mems;  
  reg [8: 1] mema [0: 255]; // declare memory called mema  
  reg [8: 1] mem_word;      // temp register called mem_word  
  ...  
  initial  
    begin  
      $displayb(mema[5]);    //显示存储器中第6个字的内容  
      mem_word = mema[5];  
      $displayb(mem_word[8]); //显示第6个字的最高有效位  
    end  
endmodule
```



若要对存储器字的某些位存取，只能通过暂存器传递

参数型

参数型：在**Verilog** 中使用 **parameter** 来定义一个标识符代表一个常量，称谓符号常量，也即标识符的常量形式，其格式如下：

parameter 参数名1=表达式,.....,参数名n =表达式

说明：

- (1) **parameter**是参数型数据的确认符。
- (2) 可一次定义多个参数，用逗号隔开。
- (3) 在每一个赋值语句的右边必须是一个常数表达式。该表达式只能包含数字或先前已定义过的参数。

用法:

- (1) 参数经常用于定义延迟时间和变量宽度, 在模块或实例引用时可以通过参数传递改变在被引用模块或实例中已定义的参数。
- (2) 在使用文字的地方都可以使用参数。
- (3) 参数的定义是局部参数时 (*localparam*), 只在当前模块中有效。

举例:

```
parameter msb = 7;      //定义参数 msb 为常量7
parameter e = 25, f = 29; //定义两个常数参数
parameter r = 5.7;      //声明 r 为一个实型参数
parameter byte_size = 8, byte_msb = byte_size-1;
//常数表达式赋值
parameter average_delay = (r_f)/2; //常数表达式赋值
```

注意：参数 **file** 不是 **string**，而是一个整数，其值是所有字母的扩展 **ASCII** 值。若 **file**="AB"，则 **file** 值为 8'h4142。

```
module mod1( out, in
    ...
parameter cycle = 20,
    setup = cycle/2 - prop_del,
    p1 = 8,
    x_word = 16'bx,
    file = "/usr1/design/mem_file.dat";
    ...
    wire [p1: 0] w1;
// A wire declaration using parameter
    ...
endmodule
```

参数重载(overriding)

模块实例化时参数重载

```
module mod1( out, in1, in2);  
...  
parameter p1 = 8,  
           real_constant = 2.039,  
           x_word = 16'bx,  
           file =  
           "/usr1/jdough/design/mem_file.dat";  
...  
endmodule  
module top;  
...  
  mod1 #(5, 3.0, 16'bx, mem_file.dat)  
    in1, in2);  
...  
endmodule
```

次序与原
说明相同

使用 #

不需要给所有参数赋新值，
但必须按顺序赋值。

因为 # 说明延时的
时候只能用于 **gate**
或过程语句，不能用于
模块实例。 **gate** 在
实例化时只能有延时，
不能有模块参数。

为什么编译器认为这是参数而不是延时呢？

复习(review)

问题：

1. 在**Verilog**中，什么情况下输出端会输出**X**值？
2. **net**和**register**类型的主要区别是什么？
3. 在**Verilog**中如何定义一个常数？

解答：

1. 若输出端输出**X**值，一种可能是输出**net**上发生驱动冲突，二是由一个未知值传递到**net**上引起。
2. **register**有存储功能，而**net**必须持续驱动。
3. 在**Verilog**中使用**parameter**定义一个常数。文本宏也是常数的一种形式。