第3章　Verilog语言要素

本章介绍Verilog HDL的基本要素，包括标识符、注释、数值、编译程序指令、系统任务和系统函数。另外，本章还介绍了Verilog硬件描述语言中的两种数据类型。

3.1　标识符

Verilog HDL中的标识符(identifier)可以是任意一组字母、数字、 $符号和_(下划线)符号的组合，但标识符的第一个字符必须是字母或者下划线。另外，标识符是区分大小写的。以下是标识符的几个例子：

```
Count
COUNT        //与Count不同。
_R1_D2
R56_68
FIVE$
```

转义标识符(escaped identifier)可以在一条标识符中包含任何可打印字符。转义标识符以 \(反斜线)符号开头，以空白结尾（空白可以是一个空格、一个制表字符或换行符）。下面例举了几个转义标识符：

```
\7400
\.*.$
\{******}
\~Q
\OutGate     与OutGate相同。
```

最后这个例子解释了在一条转义标识符中，反斜线和结束空格并不是转义标识符的一部分。也就是说，标识符 *OutGate* 和标识符 *OutGate*恒等。

Verilog HDL定义了一系列保留字，叫做关键词，它仅用于某些上下文中。 附录A列出了语言中的所有保留字。注意只有小写的关键词才是保留字。例如，标识符 always(这是个关键词)与标识符*ALWAYS*(非关键词)是不同的。

另外，转义标识符与关键词并不完全相同。标识符 \initial 与标识符initial（这是个关键词）不同。注意这一约定与那些转义标识符不同。

3.2　注释

在Verilog HDL中有两种形式的注释。

```
/*第一种形式:可以扩展至
  多行 */

//第二种形式:在本行结束。
```

3.3　格式

Verilog HDL区分大小写。也就是说大小写不同的标识符是不同的。此外， Verilog HDL是

自由格式的，即结构可以跨越多行编写，也可以在一行内编写。白空（新行、制表符和空格）没有特殊意义。下面通过实例解释说明。

```
initial begin Top = 3'b001; #2 Top = 3'b011; end
```

和下面的指令一样：

```
initial
  begin
Top = 3'b001;
#2 Top = 3'b011;
end
```

3.4 系统任务和函数

以$字符开始的标识符表示系统任务或系统函数。任务提供了一种封装行为的机制。这种机制可在设计的不同部分被调用。任务可以返回 0个或多个值。函数除只能返回一个值以外与任务相同。此外，函数在0时刻执行，即不允许延迟，而任务可以带有延迟。

```
$display ("Hi, you have reached LT today");
/* $display 系统任务在新的一行中显示。*/
$time
//该系统任务返回当前的模拟时间。
```

系统任务和系统函数在第 10章中详细讲解。

3.5 编译指令

以`（反引号）开始的某些标识符是编译器指令。在 Verilog 语言编译时，特定的编译器指令在整个编译过程中有效（编译过程可跨越多个文件），直到遇到其它的不同编译程序指令。完整的标准编译器指令如下：

- `define, `undef
- `ifdef, `else, `endif
- `default_nettype
- `include
- `resetall
- `timescale
- `unconnected_drive, `nounconnected_drive
- `celldefine, `endcelldefine

3.5.1 `define 和`undef

`define指令用于文本替换，它很像C语言中的#define 指令，如：

```
`define MAX_BUS_SIZE 32
. . .
reg [ `MAX_BUS_SIZE - 1:0 ] AddReg;
```

一旦`define 指令被编译，其在整个编译过程中都有效。例如，通过另一个文件中的`define指令，*MAX_BUS_SIZE* 能被多个文件使用。

`undef 指令取消前面定义的宏。例如：

```
`define WORD 16  //建立一个文本宏替代。
. . .
wire [ `WORD : 1] Bus;
```

```
. . .
`undef WORD
// 在`undef编译指令后, WORD的宏定义不再有效.
```

3.5.2 `ifdef、`else 和`endif

这些编译指令用于条件编译，如下所示：

```
`ifdef WINDOWS
 parameter WORD_SIZE = 16
`else
 parameter WORD_SIZE = 32
`endif
```

在编译过程中，如果已定义了名字为 *WINDOWS*的文本宏，就选择第一种参数声明，否则选择第二种参数说明。

`else 程序指令对于`ifdef 指令是可选的。

3.5.3 `default_nettype

该指令用于为隐式线网指定线网类型。也就是将那些没有被说明的连线定义线网类型。

```
`default_nettype wand
```

该实例定义的缺省的线网为线与类型。因此，如果在此指令后面的任何模块中没有说明的连线，那么该线网被假定为线与类型。

3.5.4 `include

`include 编译器指令用于嵌入内嵌文件的内容。文件既可以用相对路径名定义，也可以用全路径名定义, 例如:

```
`include " . . / . . /primitives.v
```

编译时，这一行由文件“ ../../primitives.v ”的内容替代。

3.5.5 `resetall

该编译器指令将所有的编译指令重新设置为缺省值。

```
`resetall
```

例如，该指令使得缺省连线类型为线网类型。

3.5.6 `timescale

在Verilog HDL 模型中，所有时延都用单位时间表述。使用 `timescale编译器指令将时间单位与实际时间相关联。该指令用于定义时延的单位和时延精度。 `timescale编译器指令格式为：

```
`timescale time_unit / time_precision
```

time_unit 和*time_precision* 由值1、10、和100以及单位s、ms、us、ns、ps和fs组成。例如：

```
`timescale 1ns/100ps
```

表示时延单位为1ns, 时延精度为100ps。`timescale 编译器指令在模块说明外部出现, 并且影响后面所有的时延值。例如:

```
`timescale 1ns/ 100ps
module AndFunc (Z, A, B);
  output Z;
  input A, B;

  and # (5.22, 6.17 )Al (Z, A, B);
    //规定了上升及下降时延值。
endmodule
```

编译器指令定义时延以ns为单位，并且时延精度为1/10 ns（100 ps）。因此，时延值5.22对应5.2 ns, 时延6.17对应6.2 ns。如果用如下的`timescale程序指令代替上例中的编译器指令，

```
`timescale 10ns/1ns
```

那么5.22对应52ns, 6.17对应62ns。

在编译过程中，`timescale指令影响这一编译器指令后面所有模块中的时延值，直至遇到另一个`timescale指令或`resetall指令。当一个设计中的多个模块带有自身的`timescale编译指令时将发生什么？在这种情况下，模拟器总是定位在所有模块的最小时延精度上，并且所有时延都相应地换算为最小时延精度。例如，

```
`timescale 1ns/ 100ps
module AndFunc (Z, A, B);
  output Z;
  input A, B;

  and # (5.22, 6.17 )Al (Z, A, B);
endmodule

`timescale 10ns/ 1ns
module TB;
  reg PutA, PutB;
  wire GetO;

  initial
    begin
      PutA = 0;
      PutB = 0;
      #5.21 PutB = 1;
      #10.4 PutA = 1;
      #15 PutB = 0;
    end
  AndFunc AF1(GetO, PutA, PutB);
endmodule
```

在这个例子中，每个模块都有自身的`timescale编译器指令。`timescale编译器指令第一次应用于时延。因此，在第一个模块中，5.22对应5.2 ns, 6.17对应6.2 ns; 在第二个模块中5.21对应52 ns, 10.4对应104 ns, 15对应150 ns。如果仿真模块*TB*，设计中的所有模块最小时间精度为100 ps。因此，所有延迟（特别是模块*TB*中的延迟）将换算成精度为100 ps。延迟52 ns现在对应520∗100 ps，104对应1040∗100 ps，150对应1500∗100 ps。更重要的是，仿真使用100 ps为时间精度。如果仿真模块*AndFunc*，由于模块*TB*不是模块*AddFunc*的子模块，模块*TB*中的`timescale程序指令将不再有效。

3.5.7 `unconnected_drive和`nounconnected_drive

在模块实例化中，出现在这两个编译器指令间的任何未连接的输入端口或者为正偏电路状态或者为反偏电路状态。

```
`unconnected_drive pull1
. . .
/*在这两个程序指令间的所有未连接的输入端口为正偏电路状态（连接到高电平）*/
`nounconnected_drive

`unconnected_drive pull0
. . .
/*在这两个程序指令间的所有未连接的输入端口为反偏电路状态（连接到低电平）*/
`nounconnected_drive
```

3.5.8 `celldefine 和 `endcelldefine

这两个程序指令用于将模块标记为单元模块。它们表示包含模块定义，如下例所示。

```
`celldefine
module FD1S3AX (D, CK, Q) ;
  . . .
endmodule
`endcelldefine
```

某些PLI例程使用单元模块。

3.6 值集合

Verilog HDL有下列四种基本的值：

1) **0**：逻辑0或“假”
2) **1**：逻辑1或“真”
3) **x**：未知
4) **z**：高阻

注意这四种值的解释都内置于语言中。如一个为 z的值总是意味着高阻抗，一个为 0的值通常是指逻辑0。

在门的输入或一个表达式中的为“ z ”的值通常解释成“ x ”。此外，x值和z值都是不分大小写的，也就是说，值0x1z与值0X1Z相同。Verilog HDL中的常量是由以上这四类基本值组成的。

Verilog HDL中有三类常量：

1) 整型
2) 实数型
3) 字符串型

下划线符号（_）可以随意用在整数或实数中，它们就数量本身没有意义。它们能用来提高易读性；唯一的限制是下划线符号不能用作为首字符。

3.6.1 整型数

整型数可以按如下两种方式书写：

1) 简单的十进制数格式

2) 基数格式

1. 简单的十进制格式

这种形式的整数定义为带有一个可选的“+”(一元)或“－”(一元)操作符的数字序列。下面是这种简易十进制形式整数的例子。

```
32          十进制数32
－15         十进制数－15
```

这种形式的整数值代表一个有符号的数。负数可使用两种补码形式表示。因此 32在5位的二进制形式中为10000，在6位二进制形式中为110001；－15在5位二进制形式中为10001，在6位二进制形式中为110001。

2. 基数表示法

这种形式的整数格式为：

```
[size ]'base value
```

size 定义以位计的常量的位长；*base*为o或O（表示八进制），b或B（表示二进制），d或D（表示十进制），h或H（表示十六进制）之一；*value*是基于*base*的值的数字序列。值 x和z以及十六进制中的a到f不区分大小写。

下面是一些具体实例：

```
5'O37          位八进制数
4'D2           位十进制数
4'B1x_01       位二进制数
7'Hx           位x(扩展的x)，即xxxxxxx
4'hZ           位z(扩展的z)，即zzzz
4'd-4          非法：数值不能为负
8'h 2 A        在位长和字符之间,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:位长不能够为表达式
```

注意，x（或z）在十六进制值中代表4位x（或z），在八进制中代表3位x（或z），在二进制中代表1位x（或z）。

基数格式计数形式的数通常为无符号数。这种形式的整型数的长度定义是可选的。如果没有定义一个整数型的长度，数的长度为相应值中定义的位数。下面是两个例子：

```
'o721          位八进制数
'hAF           位十六进制数
```

如果定义的长度比为常量指定的长度长，通常在左边填 0补位。但是如果数最左边一位为x或z，就相应地用x或z在左边补位。例如：

```
10'b10         左边添0占位，0000000010
10'bx0x1       左边添x占位,xxxxxxx0x1
```

如果长度定义得更小，那么最左边的位相应地被截断。例如：

```
3'b1001_0011与3'b011 相等
5'H0FFF 与5'H1F 相等
```

？字符在数中可以代替值z在值z被解释为不分大小写的情况下提高可读性（参见第8章）。

3.6.2 实数

实数可以用下列两种形式定义：

1) 十进制计数法；例如

```
2.0
5.678
11572.12
0.1
2.                    //非法：小数点两侧必须有1位数字
```

2) 科学计数法； 这种形式的实数举例如下：

```
23_5.1e2          其值为23510.0; 忽略下划线
3.6E2               360.0 e(与E相同)
5E-4              0.0005
```

Verilog语言定义了实数如何隐式地转换为整数。实数通过四舍五入被转换为最相近的整数。

```
42.446, 42.45     转换为整数42
92.5, 92.699      转换为整数93
-15.62            转换为整数-16
-26.22            转换为整数-26
```

3.6.3 字符串

字符串是双引号内的字符序列。字符串不能分成多行书写。例如：

```
"INTERNAL ERROR"
"REACHED->HERE"
```

用8位ASCII值表示的字符可看作是无符号整数。因此字符串是 8位ASCII值的序列。为存储字符串“ INTERNAL ERROR ”，变量需要8*14位。

```
reg [1 : 8*14]Message;
. . .
Message = "INTERNAL ERROR"
```

反斜线 (\ ) 用于对确定的特殊字符转义。

```
\n        换行符
\t        制表符
\\        字符\本身
\"        字符"
\206      八进制数206对应的字符
```

3.7 数据类型

Verilog HDL 有两大类数据类型。

1) 线网类型。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，例如连续赋值或门的输出。如果没有驱动元件连接到线网，线网的缺省值为 z。

2) 寄存器类型。register type表示一个抽象的数据存储单元，它只能在 always语句和initial语句中被赋值，并且它的值从一个赋值到另一个赋值被保存下来。寄存器类型的变量具有 x 的缺省值。

3.7.1 线网类型

线网数据类型包含下述不同种类的线网子类型。

- wire
- tri
- wor
- trior
- wand
- triand
- trireg
- tri1
- tri0
- supply0
- supply1

简单的线网类型说明语法为：

```
net_kind [msb:lsb] net1, net2, . . . , netN;
```

net_kind **是上述线网类型的一种。** *msb***和***lsb* **是用于定义线网范围的常量表达式；范围定义是可选的；如果没有定义范围，缺省的线网类型为** 1**位。下面是线网类型说明实例。**

```
wire Rdy, Start;    //2个1位的连线。
wand [2:0] Addr;  //Addr是3位线与。
```

当一个线网有多个驱动器时，即对一个线网有多个赋值时，不同的线网产生不同的行为。例如，

```
wor Rde;
…
assign Rde = Blt & Wyl;
…
assign Rde = Kbl | Kip;
```

本例中，*Rde***有两个驱动源，分别来自于两个连续赋值语句。由于它是线或线网，** *Rde***的有效值由使用驱动源的值（右边表达式的值）的线或(wor)表（参见后面线或网的有关章节）决定。**

1. wire**和**tri**线网**

用于连接单元的连线是最常见的线网类型。连线与三态线 (tri)**网语法和语义一致；三态线可以用于描述多个驱动源驱动同一根线的线网类型；并且没有其他特殊的意义。**

```
wire Reset;
wire [3:2] Cla, Pla, Sla;
tri [ MSB-1 : LSB +1] Art;
```

如果多个驱动源驱动一个连线（或三态线网），线网的有效值由下表决定。

wire (**或** 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

下面是一个具体实例：

```
assign Cla = Pla & Sla;
…
assign Cla = Pla ^ Sla;
```

在这个实例中，*Cla***有两个驱动源。两个驱动源的值（右侧表达式的值）用于在上表中索引，**

以便决定*Cla*的有效值。由于*Cla*是一个向量，每位的计算是相关的。例如，如果第一个右侧表达式的值为**01x**，并且第二个右测表达式的值为**11z**，那么*Cla* 的有效值是**x1x** (第一位0和1在表中索引到**x**，第二位1和1在表中索引到1，第三位**x** 和**z**在表中索引到**x**)。

2. wor和trior线网

线或指如果某个驱动源为 1，那么线网的值也为 1。线或和三态线或(trior)在语法和功能上是一致的。

```
wor [MSB:LSB] Art;
trior [MAX－1: MIN－1] Rdx, Sdx, Bdx;
```

如果多个驱动源驱动这类网，网的有效值由下表决定。

wor (或 trior)	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

3. wand和triand线网

线与(wand)网指如果某个驱动源为 0，那么线网的值为 0。线与和三态线与 (triand)网在语法和功能上是一致的。

```
wand [-7 : 0] Dbus;
triand Reset, Clk;
```

如果这类线网存在多个驱动源，线网的有效值由下表决定。

wand (或 triand)	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

4. trireg线网

此线网存储数值（类似于寄存器），并且用于电容节点的建模。当三态寄存器 (trireg)的所有驱动源都处于高阻态，也就是说，值为 z时，三态寄存器线网保存作用在线网上的最后一个值。此外，三态寄存器线网的缺省初始值为 x。

```
trireg [1:8] Dbus, Abus;
```

5. tri0和tri1线网

这类线网可用于线逻辑的建模，即线网有多于一个驱动源。 tri0 （tri1）线网的特征是，若无驱动源驱动，它的值为 0 （tri1的值为1）。

```
tri0 [－3:3] GndBus;
tri1 [0:－5] OtBus, ItBus;
```

下表显示在多个驱动源情况下 tri0或tri1网的有效值。

tri0 (tri1)	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0(1)

6. supply0和supply1线网

supply0用于对“地”建模，即低电平0；supply1网用于对电源建模，即高电平1；例如：

```
supply0 Gnd, ClkGnd;
supply1 [2:0] Vcc;
```

3.7.2 未说明的线网

在Verilog HDL中，有可能不必声明某种线网类型。在这样的情况下，缺省线网类型为1位线网。

可以使用`default_nettype编译器指令改变这一隐式线网说明方式。使用方法如下：

```
`default_nettype net_kind
```

例如，带有下列编译器指令：

```
`default_nettype wand
```

任何未被说明的网缺省为1位线与网。

3.7.3 向量和标量线网

在定义向量线网时可选用关键词scalared或vectored。如果一个线网定义时使用了关键词vectored，那么就不允许位选择和部分选择该线网。换句话说，必须对线网整体赋值（位选择和部分选择在下一章中讲解）。例如：

```
wire vectored[3:1] Grb;
  //不允许位选择Grb[2]和部分选择Grb [3:2]
wor scalared[4:0] Best;
  //与wor [4:0] Best相同，允许位选择Best [2]和部分选择Best [3:1]。
```

如果没有定义关键词，缺省值为标量。

3.7.4 寄存器类型

有5种不同的寄存器类型。

- reg
- integer
- time
- real
- realtime

1. reg寄存器类型

寄存器数据类型reg是最常见的数据类型。reg类型使用保留字reg加以说明，形式如下：

```
reg [ msb: lsb] reg1, reg2 . . .  regN;
```

*msb*和*lsb* 定义了范围，并且均为常数值表达式。范围定义是可选的；如果没有定义范围，缺省值为1位寄存器。例如：

```
reg [3:0] Sat;              /Sat为4 位寄存器。
reg Cnt;                    /1位寄存器。
reg [1:32] Kisp, Pisp, Lisp
```

寄存器可以取任意长度。寄存器中的值通常被解释为无符号数，例如：

```
reg [1:4] Comb;
```

```
...
Comb = -2;  //Comb 的值为14（1110），1110是2的补码。
Comb = 5;   //Comb的值为15（0101）。
```

2. 存储器

存储器是一个寄存器数组。存储器使用如下方式说明：

```
reg [ msb: 1sb] memory1 [ upper1: lower1],
                memory2 [upper2: lower2],...   ;
```

例如：

```
reg [0:3 ] MyMem [0:63]
   //MyMem为64个4位寄存器的数组。
reg Bog [1:5]
  //Bog为5个1位寄存器的数组。
```

*MyMem*和*Bog*都是存储器。数组的维数不能大于 2。注意存储器属于寄存器数组类型。线网数据类型没有相应的存储器类型。

单个寄存器说明既能够用于说明寄存器类型，也可以用于说明存储器类型。

```
parameter ADDR_SIZE = 16 , WORD_SIZE = 8;
reg [1: WORD_SIZE] RamPar [ ADDR_SIZE-1 : 0], DataReg;
```

*RamPar*是存储器，是16个8位寄存器数组，而DataReg是8位寄存器。

在赋值语句中需要注意如下区别：存储器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。下例说明它们之间的不同。

```
reg [1:5] Dig;  //Dig为5位寄存器。
...
Dig = 5'b11011;
```

上述赋值都是正确的，但下述赋值不正确：

```
reg BOg[1:5];  //Bog为5个1位寄存器的存储器。
...
Bog = 5'b11011;
```

有一种存储器赋值的方法是分别对存储器中的每个字赋值。例如：

```
reg [0:3] Xrom [1:4]
...
Xrom[1] = 4'hA;
Xrom[2] = 4'h8;
Xrom[3] = 4'hF;
Xrom[4] = 4'h2;
```

为存储器赋值的另一种方法是使用系统任务：

1) **$readmemb （加载二进制值）**

2) **$readmemh （加载十六进制值）**

这些系统任务从指定的文本文件中读取数据并加载到存储器。文本文件必须包含相应的二进制或者十六进制数。例如：

```
reg [1:4] RomB [7:1] ;
$ readmemb ("ram.patt", RomB);
```

*Romb*是存储器。文件“ ram.patt ”必须包含二进制值。文件也可以包含空白空间和注释。下面是文件中可能内容的实例。

```
1101
```

```
1110
1000
0111
0000
1001
0011
```

系统任务$readmemb促使从索引7即*Romb*最左边的字索引，开始读取值。如果只加载存储器的一部分，值域可以在$readmemb方法中显式定义。例如：

```
$readmemb ("ram.patt",RomB, 5, 3);
```

在这种情况下只有 *Romb*[5],*Romb*[4]和*Romb*[3]这些字从文件头开始被读取。被读取的值为1101、1100和1000。

文件可以包含显式的地址形式。

```
@hex_address value
```

如下实例：

```
@5   11001
@2   11010
```

在这种情况下，值被读入存储器指定的地址。

当只定义开始值时，连续读取直至到达存储器右端索引边界。例如：

```
$readmemb ("rom.patt",RomB, 6);
   //从地址6开始，并且持续到1。
$readmemb ( "rom.patt",RomB, 6, 4);
   //从地址6读到地址4。
```

3. Integer寄存器类型

整数寄存器包含整数值。整数寄存器可以作为普通寄存器使用，典型应用为高层次行为建模。使用整数型说明形式如下：

```
integer  integer1, integer2..   intergerN[msb:1sb] ;
```

*msb*和*lsb*是定义整数数组界限的常量表达式，数组界限的定义是可选的。注意容许无位界限的情况。一个整数最少容纳 32位。但是具体实现可提供更多的位。下面是整数说明的实例。

```
integer A, B, C;  //三个整数型寄存器。
integer Hist [3:6];  //一组四个寄存器。
```

一个整数型寄存器可存储有符号数，并且算术操作符提供 2的补码运算结果。

整数不能作为位向量访问。例如，对于上面的整数 *B*的说明，*B*[6]和*B*[20:10]是非法的。一种截取位值的方法是将整数赋值给一般的 reg类型变量，然后从中选取相应的位，如下所示：

```
reg [31:0] Breg;
integer Bint;
...
//Bint[6]和Bint[20:10]是不允许的。
...
Breg = Bint;
/*现在，Breg[6]和Breg[20:10]是允许的，并且从整数Bint获取相应的位值。*/
```

上例说明了如何通过简单的赋值将整数转换为位向量。类型转换自动完成，不必使用特定的函数。从位向量到整数的转换也可以通过赋值完成。例如：

```
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。
Bcq = J;          //Bcq的值为4'b1010。
```

注意赋值总是从最右端的位向最左边的位进行；任何多余的位被截断。如果你能够回忆起整数是作为2的补码位向量表示的，就很容易理解类型转换。

4. time类型

time类型的寄存器用于存储和处理时间。 time类型的寄存器使用下述方式加以说明。

```
time time_id1, time_id2. . ., time_idN [ msb:1sb];
```

*msb*和*lsb*是表明范围界限的常量表达式。如果未定义界限，每个标识符存储一个至少 64位的时间值。时间类型的寄存器只存储无符号数。例如：

```
time Events [0:31];   //时间值数组。
time CurrTime;        //CurrTime 存储一个时间值。
```

5. real和realtime类型

实数寄存器（或实数时间寄存器）使用如下方式说明：

```
//实数说明：
real real_reg1, real_reg2,...,   real_regN;
//实数时间说明：
realtime realtime_reg1, realtime_reg2,...,   realtime_regN;
```

realtime与real类型完全相同。例如：

```
real  Swing, Top;
realtime CurrTime;
```

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。

```
real RamCnt;
...
RamCnt = 'b01x1Z;
```

*RamCnt*在赋值后的值为'b01010。

3.8 参数

参数是一个常量。参数经常用于定义时延和变量的宽度。使用参数说明的参数只被赋值一次。参数说明形式如下：

```
parameter param1 = const_expr1, param2 = const_expr2, ...,
          paramN = const_exprN;
```

下面为具体实例：

```
parameter LINELENGTH = 132, ALL_X_S = 16'bx;
parameter BIT = 1, BYTE = 8, PI = 3.14;
parameter STROBE_DELAY = ( BYTE + BIT) / 2;
```

```
parameter TQ_FILE = " /home/bhasker/TEST/add.tq";
```

参数值也可以在编译时被改变。改变参数值可以使用参数定义语句或通过在模块初始化语句中定义参数值（这两种机制将在第9章中详细讲解）。

习题

1. 下列标识符哪些合法，哪些非法？

```
COunT, 1_2 Many, \**1, Real?, \wait, Initial
```

2. 系统任务和系统函数的第一个字符标识符是什么？
3. 举例说明文本替换编译指令？
4. 在Verilog HDL中是否有布尔类型？
5. 下列表达式的位模式是什么？

```
7'o44, 'Bx0, 5'bx110, 'hA0, 10'd2, 'hzF
```

6. 赋值后存储在*Qpr*中的位模式是什么？

```
reg [1:8*2] Qpr;
. . .
Qpr = "ME" ;
```

7. 如果线网类型变量说明后未赋值，其缺省值为多少？
8. Verilog HDL 允许没有显式说明的线网类型。如果是这样，怎样决定线网类型？
9. 下面的说明错在哪里？

```
integer [0:3] Ripple;
```

10. 编写一个系统任务从数据文件“ memA.data ”中加载32×64字存储器。
11. 写出在编译时覆盖参数值的两种方法。