例子：

```
module compare ( equal,a,b );
output    equal;       //声明输出信号 equal
input [1:0] a,b;   //声明输入信号 a,b
   assign    equal=（a==b）？1：0;
/*如果 a、b 两个输入信号相等,输出为 1。否则为 0*/
endmodule
```

从例子中可以看出整个 Verilog HDL 程序是嵌套在 module
和 endmodule 声明语句里的。
除了 endmodule 语句外,每个语句和数据定义的最后必须有分号。
/*.....*/和//.....表示注释部分。

例[3.1.3]：

```
module    trist2(out,in,enable);
output    out;
input     in, enable;
   bufif1    mybuf(out,in,enable);
endmodule
```

这个程序描述了一个名为 trist2 的三态驱动器。程序通过调用一个在 Verilog 语言库中现存的三态驱
动器实例元件 bufif1 来实现其功能。

例[3.1.4]：

```
module trist1(out,in,enable);
output    out;
input    in, enable;
    mytri    tri_inst(out,in,enable);
     //调用由 mytri 模块定义的实例元件 tri_inst
endmodule

module    mytri(out,in,enable);
output    out;
input    in, enable;
    assign    out = enable? in : 'bz;
endmodule
```

这个程序例子通过另一种方法描述了一个三态门。在这个例子中存在着两个模块。模块 trist1 调用由
模块 mytri 定义的实例元件 tri_inst。模块 trist1 是顶层模块。模块 mytri 则被称为子模块。

一，模块的端口定义

模块的端口声明了模块的输入输出口。其格式如下：

module　模块名(口 1，口 2，口 3，口 4, ……);

二，模块内容

模块的内容包括 I/O 说明、内部信号声明、功能定义

I/O 说明的格式如下：

输入口：　input　端口名 1，端口名 2，……,端口名 i;　//(共有 i 个输入口)

输出口：　output 端口名 1，端口名 2，……,端口名 j;　//(共有 j 个输出口)

I/O 说明也可以写在端口声明语句里。其格式如下：

module　module_name(input port1,input port2, output port1,output port2…);

内部信号说明：在模块内用到的和与端口有关的 wire 和 reg 变量的声明。

如：　reg [width-1 : 0] R 变量 1，R 变量 2 。。。。;

　　wire [width-1 : 0] W 变量 1，W 变量 2 。。。。;

功能定义: 有三种方法可在模块中产生逻辑。

1）.用“assign”声明语句

如：　assign　a = b & c;

2）.用实例元件

如：　and　and_inst(q, a, b);

3）.用“ always” 块

如：always @(posedge clk or posedge clr)

begin

if(clr)　q <= 0;

else　if(en) q <= d;

end

“assign”语句是描述组合逻辑最常用的方法之一。而“always”块既可用于描述组合逻辑也可

描述时序逻辑。

事件控制：@([event], [event], …]

注意：

如果用 Verilog 模块实现一定的功能，首先应该清楚哪些是同时发生的,哪些是顺序发生的。这三个例子描述的逻辑功能是同时执行的。也就是说，如果把这三项写到一个 Verilog 模块文件中去，它们的次序不会影响逻辑实现的功能。这三项是同时执行的，也就是并发的。然而，在“always”模块内，逻辑是按照指定的顺序执行的。“always”块中的语句称为“顺序语句”，因为它们是顺序执行的。请注意，两个或更多的“always”模块也是同时执行的。

三，数据类型及其常量、变量

1， 常量

数字

在 Verilog HDL 中,整型常量即整常数有以下四种进制表示形式:

1)　二进制整数(b 或 B)

2) 十进制整数(d 或 D)
3) 十六进制整数(h 或 H)
4) 八进制整数(o 或 O)
数字表达方式有以下三种:
1) <位宽><进制><数字>这是一种全面的描述方式。
2) <进制><数字>在这种描述方式中,数字的位宽采用缺省位宽(这由具体的机器系统决定,但至少 32 位)。
3) <数字>在这种描述方式中,采用缺省进制十进制。

```
8'b10101100   //位宽为 8 的数的二进制表示, 'b 表示二进制
8'ha2         //位宽为 8 的数的十六进制， 'h 表示十六进制。
```

x 代表不定值,z 代表高阻值。z 还有一种表达方式是可以写作?。在使用 case 表达式时建议使用这种写法,以提高程序的可读性。

下划线(underscore_):
下划线可以用来分隔开数的表达以提高程序可读性。但不可以用在位宽和进制处,只能用在具体的数
字之间。见下例:

```
16'b1010_1011_1111_1010        //合法格式
8'b_0011_1010          //非法格式
```

当常量不说明位数时，默认值是 32 位。

2， 参数(Parameter)型
用 parameter 来定义一个标识符代表一个常量,称为符号
常量,即标识符形式的常量,采用标识符代表一个常量可提高程序的可读性和可维护性。

```
parameter   参数名 1＝表达式，参数名 2＝表达式,  …，  参数名 n＝表达式;
eg:parameter   e=25, f=29;   //定义二个常数参数
```

参数型常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通过参数传递改变在被引用模块或实例中已定义的参数。

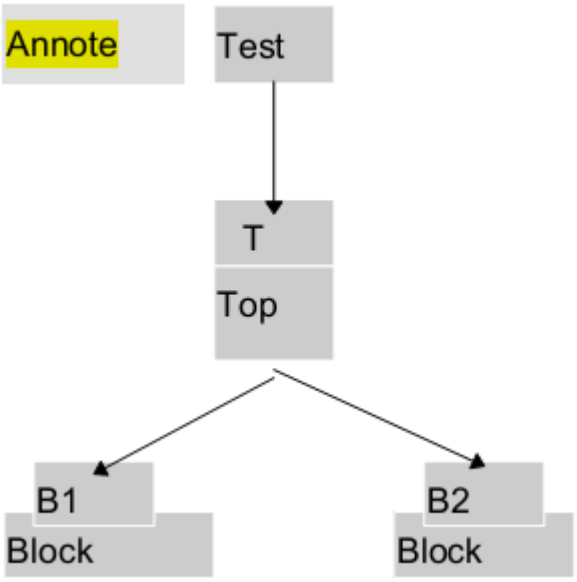
[例 1]: 在引用 Decode 实例时，D1，D2 的 Width 将采用不同的值 4 和 5，且 D1 的 Polarity 将为 0。可用例子中所用的方法来改变参数，即用 #(4,0)向 D1 中传递 Width=4,Polarity=0; 用#(5)向 D2 中传递 Width=5,Polarity 仍为 1。

```
module Decode(A,F);
parameter   Width=1, Polarity=1;
……..
endmodule
module   Top;
wire[3:0] A4;
wire[4:0] A5;
wire[15:0] F16;
```

```
wire[31:0] F32;
Decode    #(4,0)    D1(A4,F16);
Decode    #(5)       D2(A5,F32);
Endmodule
```

[例 2] ：下面是一个多层次模块构成的电路，在一个模块中改变另一个模块的参数时，需要使用 defparam 命令

```
Module Test;
  wire W;
    Top T ( );
emdmodule
module Top;
    wire W
        Block B1 ( );
        Block B2 ( );
endmodule
module Block;
        Parameter P = 0;
endmodule
module Annotate;
    defparam
      Test.T.B1.P = 2,
      Test.T.B2.P = 3;
endmodule
```



3 变量

wire 型

wire 型数据常用来表示用于以 assign 关键字指定的组合逻辑信号。Verilog 程序模块中输

入输出信号类型缺省时自动定义为 wire 型。
其格式如下：
wire [n-1:0] 数据名 1,数据名 2,…数据名 i; //共有 i 条总线，每条总线内有 n 条线路
或
wire [n:1] 数据名 1,数据名 2,…数据名 i;
eg: wire [7:0] b; //定义了一个八位的 wire 型数据

reg 型
寄存器是数据储存单元的抽象。寄存器数据类型的关键字是 reg.通过赋值语句可以改变寄存器储存的值，其作用与改变触发器储存的值相当。
reg 类型数据的缺省初始值为不定值，x。

reg 型数据常用来表示用于 “always” 模块内的指定信号，常代表触发器。通常，在设计中要由“ always”块通过使用行为描述语句来表达逻辑关系。在“ always” 块内被赋值的每一个信号都必须定义成 reg 型。
reg 型数据的格式如下：
reg [n-1:0] 数据名 1,数据名 2, … 数据名 i;
或
reg [n:1] 数据名 1,数据名 2, … 数据名 i;

reg 型数据的缺省初始值是不定值。reg 型数据可以赋正值，也可以赋负值。但当一个 reg 型数据是一个表达式中的操作数时，它的值被当作是无符号值，即正值。

memory 型
Verilog HDL 通过对 reg 型变量建立数组来对存储器建模，可以描述 RAM 型存储器，ROM 存储器和 reg 文件。
在 Verilog 语言中没有多维数组存在。memory 型数据是通过扩展 reg 型数据的地址范围来生成的。其格式如下：
reg [n-1:0] 存储器名[m-1:0]；
或 reg [n-1:0] 存储器名[m:1]；

eg:reg [7:0] mema[255：0];

这个例子定义了一个名为 mema 的存储器，该存储器有 256 个 8 位的存储器。该存储器的地址范围是 0 到 255。 **注意：对存储器进行地址索引的表达式必须是常数表达式。**

reg [n-1:0] rega; //一个 n 位的寄存器
reg mema [n-1:0]; //一个由 n 个 1 位寄存器构成的存储器组
一个 n 位的寄存器可以在一条赋值语句里进行赋值，而一个完整的存储器则不行。见下例：
rega =0; //合法赋值语句
mema =0; //非法赋值语句

四，运算符及表达式

1) 算术运算符(+,－,×，/,%)
2) **赋值运算符(=,<=)**
3) 关系运算符(>,<,>=,<=)
4) 逻辑运算符(&&,||,!)
5) 条件运算符(?:)
6) 位运算符(~,|,^,&,^~)
7) 移位运算符(<<,>>)
8) **拼接运算符({ })**
9) 其它

基本的算术运算符

进行取模运算时，结果值的符号位采用模运算式里第一个操作数的符号位。

eg:-10%3 = -1

注意： 在进行算术运算操作时，如果某一个操作数有不确定的值 x，则整个结果也为不定 x。

位运算符

在硬件电路中信号有四种状态值 1,0,x,z.

1) ~ //取反
2) & //按位与
3) | //按位或
4) ^ //按位异或
5) **^~ //按位同或(异或非)**

1) "取反"运算符~

~是一个单目运算符,用来对一个操作数进行按位取反运算。
其运算规则见下表:

~	
1	0
0	1
x	x

2) "按位与"运算符&

按位与运算就是将两个操作数的相应位进行与运算,
其运算规则见下表:

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

3) "按位或"运算符|

按位或运算就是将两个操作数的相应位进行或运算。
其运算规则见下表:

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

4) "按位异或"运算符^(也称之为XOR运算符)

按位异或运算就是将两个操作数的相应位进行异或运算。
其运算规则见下表:

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

5) "按位同或"运算符^~

按位同或运算就是将两个操作数的相应位先进行异或运算再进行非运算.
其运算规则见下表:

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

6) 不同长度的数据进行位运算

两个长度不同的数据进行位运算时,系统会自动的将两者按右端对齐.位数少的操作数会在相应的高位用 0 填满,以使两个操作数按位进行操作.

关系运算符

在进行关系运算时，如果声明的关系是假的(flase)，则返回值是 0，如果声明的关系是真的(true)，则返回值是 1，**如果某个操作数的值不定，则关系是模糊的，返回值是不定值。**

位拼接运算符(Concatation)

位拼接运算符{},其使用方法如下：

{信号 1 的某几位，信号 2 的某几位，..,..,信号 n 的某几位}

eg:{a,b[3:0],w,3 ’ b101}

也可以写成为

{a,b[3],b[2],b[1],b[0],w,1 ’ b1,1’ b0,1’ b1}

在位拼接表达式中不允许存在没有指明位数的信号。这是因为在计算拼接信号的位宽的大小时必需知道其中每个信号的位宽。

位拼接还可以用重复法来简化表达式。见下例：

{4{w}} //这等同于{w,w,w,w}

位拼接还可以用嵌套的方式来表达。见下例：

{b,{3{a,b}}} //这等同于{b,a,b,a,b,a,b}

用于表示重复的表达式如上例中的 4 和 3，必须是常数表达式。

缩减运算符(reduction operator)

缩减运算是对单个操作数进行或与非递推运算,最后的运算结果是一位的二进制数。缩减运算的具体运算过程是这样的:第一步先将操作数的第一位与第二位进行或与非运算,第二步将运算结果与第三位进行或与非运算,依次类推,直至最后一位。

例如：reg [3:0] B;

reg C;

C = &B;

相当于：

C =((B[0]&B[1]) & B[2]) & B[3];

缩减运算的与、或、非运算规则类似于位运算符与、或、非运算规则,

五，赋值语句和块语句

赋值语句

(1).非阻塞(Non_Blocking)赋值方式(如 b <= a;)

1) 块结束后才完成赋值操作。
2) b 的值并不是立刻就改变的。

(2).阻塞(Blocking)赋值方式(如 b = a;)

1) 赋值语句执行完后,块才结束。
2) b 的值在赋值语句执行完后立刻就改变的。

前面所举的例子中的"always"模块内的 reg 型信号都是采用下面的这种赋值方式:

b <= a;

这种方式的赋值并不是马上执行的，也就是说"always"块内的下一条语句执行后，b 并不等于 a，而是保持原来的值。"always"块结束后，才进行赋值。而另一种赋值方式阻塞赋值方式，如下所示:

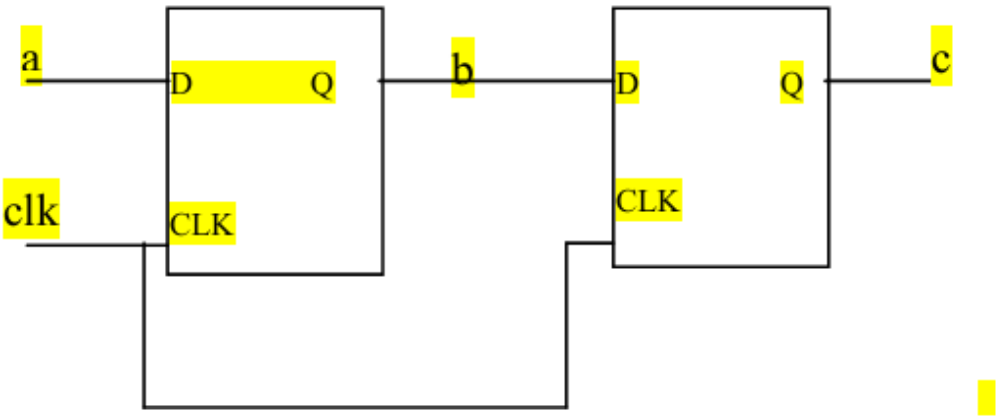
b = a;

这种赋值方式是马上执行的。也就是说执行下一条语句时，b 已等于 a。

eg:

```
always @( posedge clk )
begin
b<=a;
c<=b;
end
```

请注意：赋值是在"always"块结束后执行的，c 应为原来 b 的值。



eg:

```
always @(posedge    clk)
begin
b=a;
c=b;
end
```