第三章 Verilog HDL的基本语法

前言

Verilog HDL是一种用于数字逻辑电路设计的语言。用Verilog HDL描述的电路设计就是该电路的Verilog HDL模型。Verilog HDL既是一种行为描述的语言也是一种结构描述的语言。这也就是说，既可以用电路的功能描述也可以用元器件和它们之间的连接来建立所设计电路的Verilog HDL模型。Verilog模型可以是实际电路的不同级别的抽象。这些抽象的级别和它们对应的模型类型共有以下五种：

- 系统级(system):用高级语言结构实现设计模块的外部性能的模型。
- 算法级(algorithm):用高级语言结构实现设计算法的模型。
- RTL级(Register Transfer Level):描述数据在寄存器之间流动和如何处理这些数据的模型。
- 门级(gate-level):描述逻辑门以及逻辑门之间的连接的模型。
- 开关级(switch-level):描述器件中三极管和储存节点以及它们之间连接的模型。

一个复杂电路系统的完整Verilog HDL模型是由若干个Verilog HDL模块构成的，每一个模块又可以由若干个子模块构成。其中有些模块需要综合成具体电路，而有些模块只是与用户所设计的模块交互的现存电路或激励信号源。利用Verilog HDL语言结构所提供的这种功能就可以构造一个模块间的清晰层次结构来描述极其复杂的大型设计，并对所作设计的逻辑电路进行严格的验证。

Verilog HDL行为描述语言作为一种结构化和过程性的语言，其语法结构非常适合于算法级和RTL级的模型设计。这种行为描述语言具有以下功能：

- 可描述顺序执行或并行执行的程序结构。
- 用延迟表达式或事件表达式来明确地控制过程的启动时间。
- 通过命名的事件来触发其它过程里的激活行为或停止行为。
- 提供了条件、if-else、case、循环程序结构。
- 提供了可带参数且非零延续时间的任务(task)程序结构。
- 提供了可定义新的操作符的函数结构(function)。
- 提供了用于建立表达式的算术运算符、逻辑运算符、位运算符。
- Verilog HDL语言作为一种结构化的语言也非常适合于门级和开关级的模型设计。因其结构化的特点又使它具有以下功能：
  - 提供了完整的一套组合型原语(primitive);
  - 提供了双向通路和电阻器件的原语;
  - 可建立MOS器件的电荷分享和电荷衰减动态模型。

  Verilog HDL的构造性语句可以精确地建立信号的模型。这是因为在Verilog HDL中，提供了延迟和输出强度的原语来建立精确程度很高的信号模型。信号值可以有不同的的强度，可以通过设定宽范围的模糊值来降低不确定条件的影响。

Verilog HDL作为一种高级的硬件描述编程语言，有着类似C语言的风格。其中有许多语句如：if语句、case语句等和C语言中的对应语句十分相似。如果读者已经掌握C语言编程的基础，那么学习Verilog HDL并不困难，我们只要对Verilog HDL某些语句的特殊方面着重理解，并加强上机练习就能很好地掌握它，利用它的强大功能来设计复杂的数字逻辑电路。下面我们将对Verilog HDL中的基本语法逐一加以介绍。

3.1.简单的Verilog HDL模块

3.1.1.简单的Verilog HDL程序介绍

下面先介绍几个简单的Verilog HDL程序,然后从中分析Verilog HDL程序的特性。

```
例[3.1.1]:module  adder ( count,sum,a,b,cin );
                  input [2:0] a,b;
                  input   cin;
                  output  count;
                  output [2:0] sum;
                    assign {count,sum} = a + b + cin;
                  endmodule
```

这个例子通过连续赋值语句描述了一个名为adder的三位加法器可以根据两个三比特数a、b和进位（cin）计算出和（sum）和进位（count）。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。

```
例[3.1.2]：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
```

这个程序通过连续赋值语句描述了一个名为compare的比较器。对两比特数 a、b 进行比较，如a与b相等，则输出equal为高电平，否则为低电平。在这个程序中,/*.....*/和//.....表示注释部分,注释只是为了方便程序员理解程序,对编译是不起作用的。

```
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。

通过上面的例子可以看到:

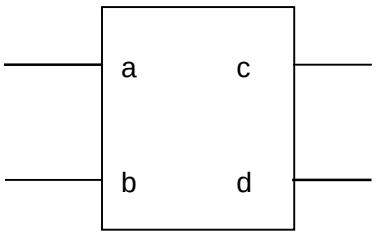
- Verilog HDL程序是由模块构成的。每个模块的内容都是嵌在module和endmodule两个语句之间。每个模块实现特定的功能。模块是可以进行层次嵌套的。正因为如此,才可以将大型的数字电路设计分割成不同的小模块来实现特定的功能,最后通过顶层模块调用子模块来实现整体功能。
- 每个模块要进行端口定义,并说明输入输出口,然后对模块的功能进行行为逻辑描述。
- Verilog HDL程序的书写格式自由,一行可以写几个语句,一个语句也可以分写多行。
- 除了endmodule语句外,每个语句和数据定义的最后必须有分号。
- 可以用/*.....*/和//.....对Verilog HDL程序的任何部分作注释。一个好的,有使用价值的源程序都应当加上必要的注释,以增强程序的可读性和可维护性。

3.1.2.模块的结构

Verilog的基本设计单元是“模块”(block)。一个模块是由两部分组成的，一部分描述接口，另一部分描述逻辑功能，即定义输入是如何影响输出的。下面举例说明：

```
module  block (a,b,c,d);
input a,b;
output c,d;

assign c= a | b ;
assign d= a & b;
endmodule
```



请看上面的例子，程序模块旁边有一个电路图的符号。在许多方面，程序模块和电路图符号是一致的，这是因为电路图符号的引脚也就是程序模块的接口。而程序模块描述了电路图符号所实现的逻辑功能。上面的Verilog设计中，模块中的第二、第三行说明接口的信号流向，第四、第五行说明了模块的逻辑功能。以上就是设计一个简单的Verilog程序模块所需的全部内容。

从上面的例子可以看出，Verilog结构完全嵌在module和endmodule声明语句之间，每个Verilog程序包括四个主要部分：端口定义、I/O说明、内部信号声明、功能定义。

3.1.3.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, ...);

3.1.4.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**：模块中最重要的部分是逻辑功能定义部分。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;

这种方法的句法很简单，只需写一个“assign”，后面再加一个方程式即可。例子中的方程式描述了一个有两个输入的与门。

2）.用实例元件

如： and and_inst(q, a, b);

采用实例元件的方法象在电路图输入方式下，调入库元件一样。键入元件的名字和相连的引脚即可，表示在设计中用到一个跟与门（and）一样的名为and_inst的与门，其输入端为a, b，输出为q。要求每个实例元件的名字必须是唯一的，以避免与其他调用与门（and）的实例混淆。

3）.用“always”块

如：always @(posedge clk or posedge clr)
begin
if(clr) q <= 0;
else if(en) q <= d;
end

采用“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。上面的例子用“always”块生成了一个带有异步清除端的D触发器。“always”块可用很多种描述手段来表达逻辑，例如上例中就用了if...else语句来表达逻辑关系。如按一定的风格来编写“always”块，可以通过综合工具把源代码自动综合成用门级结构表示的组合或时序逻辑电路。

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。上面三个例子分别采用了“assign”语句、实例元件和“always”块。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 VeriIog 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，但是模块内部的语句是顺序执行的。 看一下“always”内的语句，你就会明白它是如何实现功能的。 if..else... if 必须顺序执行，否则其功能就没有任何意义。如果else语句在if语句之前执行，功能就会不符合要求！为了能实现上述描述的功能，“always”模块内部的语句将按照书写的顺序执行。

3.2.数据类型及其常量、变量

Verilog HDL中总共有十九种数据类型,数据类型是用来表示数字电路硬件中的数据储存和传送元素的。在本教材中我们先只介绍四个最基本的数据类型,它们是：

reg型、wire型、integer型、parameter型

其它数据类型在后面的章节里逐步介绍，同学们也可以查阅附录中Verilog HDL语法参考书的有关章节逐步掌握。其它的类型如下：

large型、medium型、scalared型、time型、small型、tri型、trio型、tri1型、triand型、trior型、trireg型、vectored型、wand型、wor型。这些数据类型除time型外都与基本逻辑单元建库有关，与系统设计没有很大的关系。在一般电路设计自动化的环境下，仿真用的基本部件库是由半导体厂家和EDA工具厂家共同提供的。系统设计工程师不必过多地关心门级和开关级的Verilog HDL语法现象。

Verilog HDL语言中也有常量和变量之分。它们分别属于以上这些类型。下面就最常用的几种进行介绍。

3.2.1.常量

在程序运行过程中,其值不能被改变的量称为常量。下面首先对在Verilog HDL语言中使用的数字及其表示方式进行介绍。

一. 数字

- **整数**:

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。

在表达式中,位宽指明了数字的精确位数。例如:一个4位二进制数的数字的位宽为4,一个4位十六进制数的数字的位宽为16(因为每单个十六进制数就要用4位二进制数来表示)。见下例:

```
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和z值:**

在数字电路中,x代表不定值,z代表高阻值。一个x可以用来定义十六进制数的四位二进制数的状态,八进制数的三位,二进制数的一位。z的表示方式同x类似。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。见下例:

```
4'b10x0   //位宽为4的二进制数从低位数起第二位为不定值
4'b101z   //位宽为4的二进制数从低位数起第一位为高阻值
12'dz     //位宽为12的十进制数其值为高阻值(第一种表达方式)
12'd?     //位宽为12的十进制数其值为高阻值(第二种表达方式)
8'h4x     //位宽为8的十六进制数其低四位值为不定值
```

- **负数:**

一个数字可以被定义为负数,只需在位宽表达式前加一个减号,减号必须写在数字定义表达式的最前面。注意减号不可以放在位宽和进制之间也不可以放在进制和具体的数之间。见下例:

```
-8'd5  //这个表达式代表5的补数（用八位二进制数表示）
8'd-5  //非法格式
```

- **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，每个字母用8位的ASCII值表示。

例：

10＝32'd10＝32'b1010

1=32’d1=32’b1
-1=-32’d1=32’hFFFFFFFF
‘BX=32’BX=32’BXXXXXXX...X
“AB”=16’B01000001_01000010

二．参数(Parameter)型

在Verilog HDL中用parameter来定义常量,即用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型数据是一种常数型的数据，其说明格式如下：

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

parameter是参数型数据的确认符，确认符后跟着一个用逗号分隔开的赋值语句表。在每一个赋值语句的右边必须是一个常数表达式。也就是说，该表达式只能包含数字或先前已定义过的参数。见下列：

```
parameter  msb=7;        //定义参数msb为常量7
parameter  e=25, f=29;   //定义二个常数参数
parameter  r=5.7;        //声明r为一个实型参数
parameter  byte_size=8, byte_msb=byte_size-1; //用常数表达式赋值
parameter  average_delay = (r+f)/2;           //用常数表达式赋值
```

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。下面将通过两个例子进一步说明在层次调用的电路中改变参数常用的一些用法。

[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
```

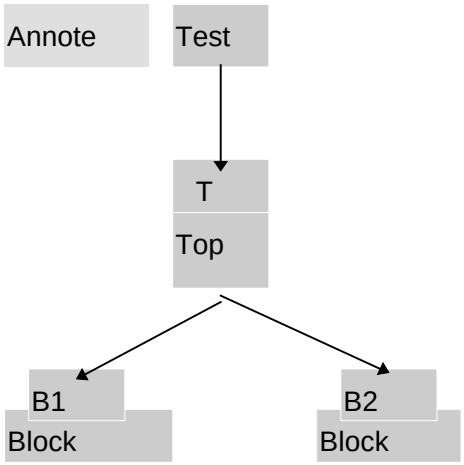
[例2]：下面是一个多层次模块构成的电路，在一个模块中改变另一个模块的参数时，需要使用defparam命令

```
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
```



3.2.2 变量

变量即在程序运行过程中其值可以改变的量,在Verilog HDL中变量的数据类型有很多种,这里只对常用的几种进行介绍。

网络数据类型表示结构实体(例如门)之间的物理连接。网络类型的变量不能储存值，而且它必需受到驱动器(例如门或连续赋值语句，assign)的驱动。如果没有驱动器连接到网络类型的变量上，则该变量就是高阻的，即其值为z。常用的网络数据类型包括wire型和tri型。这两种变量都是用于连接器件单元，它们具有相同的语法格式和功能。之所以提供这两种名字来表达相同的概念是为了与模型中所使用的变量的实际情况相一致。wire型变量通常是用来表示单个门驱动或连续赋值语句驱动的网络型数据，tri型变量则用来表示多驱动器驱动的网络型数据。如果wire型或tri型变量没有定义逻辑强度(logic strength)，在多驱动源的情况下，逻辑值会发生冲突从而产生不确定值。下表为wire型和tri型变量的真值表(注意:这里假设两个驱动源的强度是一致的，关于逻辑强度建模请参阅附录:Verilog语言参考书)。

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

一. 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。wire型信号可以用作任何方程式的输入，也可以用作“assign”语句或实例元件的输出。

wire型信号的格式同reg型信号的很类似。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;

wire是wire型数据的确认符，[n-1:0]和[n:1]代表该数据的位宽，即该数据有几位。最后跟着的是数据的名字。如果一次定义多个数据，数据名之间用逗号隔开。声明语句的最后要用分号表示语句结束。看下面的几个例子。

```
wire  a;          //定义了一个一位的wire型数据
wire [7:0] b;     //定义了一个八位的wire型数据
wire [4:1] c, d;  //定义了二个四位的wire型数据
```

二． 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。Verilog HDL语言提供了功能强大的结构语句使设计者能有效地控制是否执行这些赋值语句。这些控制结构用来描述硬件触发条件，例如时钟的上升沿和多路器的选通信号。在行为模块介绍这一节中我们还要详细地介绍这些控制结构。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是reg型数据的确认标识符，[n-1:0]和[n:1]代表该数据的位宽，即该数据有几位（bit)。最后跟着的是数据的名字。如果一次定义多个数据，数据名之间用逗号隔开。声明语句的最后要用分号表示语句结束。看下面的几个例子：

```
reg  rega;          //定义了一个一位的名为rega的reg型数据
reg [3:0]  regb;    //定义了一个四位的名为regb的reg型数据
reg [4:1]  regc, regd; //定义了两个四位的名为regc和regd的reg型数据
```

对于reg型数据，其赋值语句的作用就象改变一组触发器的存储单元的值。在Verilog中有许多构造(construct)用来控制何时或是否执行这些赋值语句。这些控制构造可用来描述硬件触发器的各种具体情况，如触发条件用时钟的上升沿等，或用来描述具体判断逻辑的细节，如各种多路选择器。**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。例如：当一个四位的寄存器用作表达式中的操作数时，如果开始寄存器被赋以值-1,则在表达式中进行运算时，其值被认为是+15。

注意：

---

reg型只表示被定义的信号将用在“always”块内，理解这一点很重要。并不是说reg型信号一定是寄存器或触发器的输出。虽然reg型信号常常是寄存器或触发器的输出，但并不一定总是这样。在本书中我们还会对这一点作更详细的解释。

三． 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。数组中的每一个单元通过一个数组索引进行寻址。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];
```

在这里，reg[n-1:0]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。存储器名后的[m-1:0]或[m:1]则定义了该存储器中有多少个这样的寄存器。最后用分号结束定义语句。下面举例说明：

```
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。**

另外，在同一个数据类型声明语句里，可以同时定义**存储器**型数据和reg型数据。见下例：

```
parameter  wordsize=16,          //定义二个参数。
           memsize=256;
reg [wordsize-1:0] mem[memsize-1:0],writereg, readreg;
```

尽管memory型数据和reg型数据的定义格式很相似，但要注意其不同之处。如一个由n个1位寄存器构成的存储器组是不同于一个n位的寄存器的。见下例：

```
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
```

如果想对memory中的存储单元进行读写操作，必须指定该单元在存储器中的地址。下面的写法是正确的。

```
mema[3]=0;  //给memory中的第3个存储单元赋值为0。
```

进行寻址的地址索引可以是表达式，这样就可以对存储器中的不同单元进行操作。表达式的值可以取决于电路中其它的寄存器的值。例如可以用一个加法计数器来做RAM的地址索引。本小节里只对以上几种常用的数据类型和常数进行了介绍，其余的在以后的章节的示例中用到之处再逐一介绍。有兴趣的同学可以参阅附录：Verilog语言参考书

3.3. 运算符及表达式

Verilog HDL语言的运算符范围很广，其运算符按其功能可分为以下几类:

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

在Verilog HDL语言中运算符所带的操作数是不同的，按其所带操作数的个数运算符可分为三种:

1) 单目运算符(unary operator):可以带一个操作数,操作数放在运算符的右边。
2) 二目运算符(binary operator):可以带二个操作数,操作数放在运算符的两边。
3) 三目运算符(ternary operator):可以带三个操作,这三个操作数用三目运算符分隔开。

见下例:

```
clock = ~clock;        // ~是一个单目取反运算符, clock是操作数。
c = a | b;             // 是一个二目按位或运算符, a 和 b是操作数。
r = s ? t : u;         // ?: 是一个三目条件运算符, s,t,u是操作数。
```

下面对常用的几种运算符进行介绍。

3.3.1.基本的算术运算符

在Verilog HDL语言中，算术运算符又称为二进制运算符，共有下面几种：

1) + (加法运算符,或正值运算符,如 rega＋regb，＋3)
2) － (减法运算符，或负值运算符，如 rega－3,－3)
3) × (乘法运算符，如rega*3)
4) / (除法运算符，如5/3)
5) % (模运算符，或称为求余运算符，要求%两侧均为整型数据。如7%3的值为1)

在进行整数除法运算时，结果值要略去小数部分，只取整数部分。而进行取模运算时，结果值的符号位采用模运算式里第一个操作数的符号位。见下例。

模运算表达式	结果	说明
10%3	1	余数为1
11%3	2	余数为2
12%3	0	余数为0即无余数
-10%3	-1	结果取第一个操作数的符号位,所以余数为-1
11%3	2	结果取第一个操作数的符号位,所以余数为2.

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值x。

3.3.2.位运算符

Verilog HDL作为一种硬件描述语言,是针对硬件电路而言的。在硬件电路中信号有四种状态值1,0,x,z.在电路中信号进行与或非时，反映在Verilog HDL中则是相应的操作数的位运算。Verilog HDL提供了以下五种位运算符：

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

说明:

- 位运算符中除了~是单目运算符以外,均为二目运算符,即要求运算符两侧各有一个操作数.
- 位运算符中的二目运算符要求对两个操作数的相应位进行运算操作。

下面对各运算符分别进行介绍:

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

举例说明:

```
rega='b1010;//rega的初值为'b1010
rega=~rega;//rega的值进行取反运算后变为'b0101
```

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.

3.3.3 逻辑运算符

在Verilog HDL语言中存在三种逻辑运算符:

1) && 逻辑与
2) || 逻辑或
3) ! 逻辑非

"&&"和"||"是二目运算符,它要求有两个操作数,如(a>b)&&(b>c),(a<b)||(b<c)。"!"是单目运算符,只要求一个操作数,如!(a>b)。下表为逻辑运算的真值表。它表示当a和b的值为不同的组合时,各种逻辑运算所得到的值。

a	b	!a	!b	a&&b	a||b
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

逻辑运算符中"&&"和"||"的优先级别低于关系运算符,"!" 高于算术运算符。见下例:

(a>b)&&(x>y) 可写成: a>b && x>y
(a==b)||(x==y) 可写成:a==b || x==y
(!a)||(a>b) 可写成: !a || a>b

为了提高程序的可读性,明确表达各运算符间的优先关系,建议使用括号.

3.3.4.关系运算符

关系运算符共有以下四种:

```
a < b        a小于b
a > b        a大于b
a <= b       a小于或等于b
a >= b       a大于或等于b
```

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。

所有的关系运算符有着相同的优先级别。关系运算符的优先级别低于算术运算符的优先级别。见下例:

```
a < size-1          //这种表达方式等同于下面
a < (size-1)        //这种表达方式。
size - ( 1 < a )    //这种表达方式不等同于下面
size - 1 < a        //这种表达方式。
```

从上面的例子可以看出这两种不同运算符的优先级别。当表达式size－(1<a)进行运算时，关系表达式先被运算，然后返回结果值0或1被size减去。而当表达式 size－1<a 进行运算时，size先被减去1，然后再同a相比。

3.3.5.等式运算符

在Verilog HDL语言中存在四种等式运算符:

1) == (等于)
2) != (不等于)
3) === (等于)
4) !== (不等于)

这四个运算符都是二目运算符,它要求有两个操作数。"=="和"!="又称为逻辑等式运算符。其结果由两个操作数的值决定。由于操作数中某些位可能是不定值x和高阻值z,结果可能为不定值x。**而"==="和"!=="运算符则不同,它在对操作数进行比较时对某些位的不定值x和高阻值z也进行比较,两个操作数必需完全一致，其结果才是1，否则为0。**"==="和"!=="运算符常用于case表达式的判别,所以又称为"case等式运算符"。这四个等式运算符的优先级别是相同的。下面画出＝＝与＝＝＝的真值表，帮助理解两者间的区别。

= = =	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1

= =	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

下面举一个例子说明“＝＝”和“＝＝＝”的区别。
例：

```
if(A==1'bx)  $display("AisX"); (当A等于X时，这个语句不执行)
if(A===1'bx) $display("AisX"); (当A等于X时，这个语句执行)
```

3.3.6.移位运算符

在Verilog HDL中有两种移位运算符：
<< (左移位运算符) 和 >>(右移位运算符)。
其使用方法如下：
a >> n 或 a << n
a代表要进行移位的操作数，n代表要移几位。这两种移位运算都用0来填补移出的空位。下面举例说明:

```
module  shift;
  reg [3:0]  start, result;
    initial
    begin
        start  = 1;     //start在初始时刻设为值0001
        result = (start<<2);
```

```
                //移位后，start的值0100，然后赋给result。
            end
        endmodule
```

从上面的例子可以看出，start在移过两位以后，用0来填补空出的位。

进行移位运算时应注意移位前后变量的位数，下面将给出一例。

```
例：4'b1001<<1 = 5'b10010;  4'b1001<<2 = 6'b100100;
    1<<6 = 32'b1000000;     4'b1001>>1 = 4'b0100;  4'b1001>>4 = 4'b0000;
```

3.3.7.位拼接运算符(Concatation)

在Verilog HDL语言有一个特殊的运算符：**位拼接**运算符{}。用这个运算符可以把两个或多个信号的某些位拼接起来进行运算操作。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}
```

即把某些信号的某些位详细地列出来，中间用逗号分开，最后用大括号括起来表示一个整体信号。见下例：

```
{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。

3.3.8.缩减运算符(reduction operator)

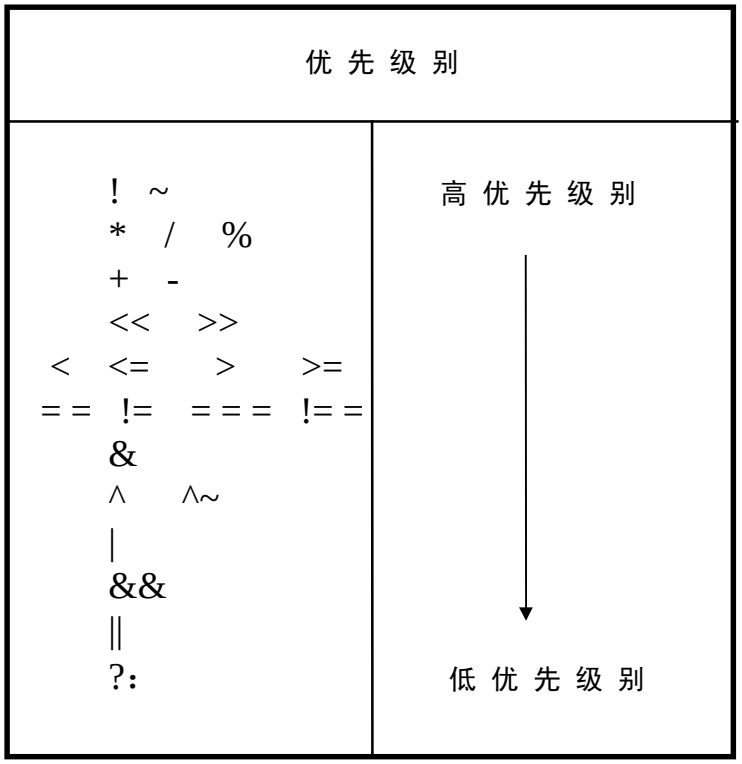
缩减运算符是单目运算符,也有与或非运算。其与或非运算规则类似于位运算符的与或非运算规则,但其运算过程不同。位运算是对操作数的相应位进行与或非运算,操作数是几位数则运算结果也是几位数。而缩减运算则不同,缩减运算是对单个操作数进行或与非递推运算,最后的运算结果是一位的二进制数。缩减运算的具体运算过程是这样的:第一步先将操作数的第一位与第二位进行或与非运算,第二步将运算结果与第三位进行或与非运算,依次类推,直至最后一位。

```
例如：reg [3:0] B;
      reg C;
          C = &B;
          相当于：
          C =( (B[0]&B[1]) & B[2] ) & B[3];
```

由于缩减运算的与、或、非运算规则类似于位运算符与、或、非运算规则,这里不再详细讲述,请参照位运算符的运算规则介绍。

3.3.9.优先级别

下面对各种运算符的优先级别关系作一总结。见下表:



3.3.10.关键词

在Verilog HDL中，所有的关键词是事先定义好的确认符,用来组织语言结构。关键词是用小写字母定义的,因此在编写原程序时要注意关键词的书写,以避免出错。下面是Verilog HDL中使用的关键词(请参阅附录：Verilog语言参考手册)：

always, and, assign, begin, buf, bufif0, bufif1, case, casex, casez, cmos, deassign, default, defparam, disable, edge, else, end, endcase, endmodule, endfunction, endprimitive, endspecify, endtable, endtask, event, for, force, forever, fork, function, highz0, highz1, if, initial, inout, input, integer, join, large, macromodule, medium, module, nand, negedge, nmos, nor, not, notif0, notif1, or, output, parameter, pmos, posedge, primitive, pull0, pull1, pullup, pulldown, rcmos, reg, releses, repeat, mmos, rpmos, rtran, rtranif0, rtranif1, scalared, small, specify, specparam, strength, strong0, strong1, supply0, supply1, table, task, time, tran, tranif0, tranif1, tri, tri0, tri1, triand, trior, trireg, vectored, wait, wand, weak0, weak1, while, wire, wor, xnor, xor

注意在编写Verilog HDL程序时,变量的定义不要与这些关键词冲突.

3.4 赋值语句和块语句

3.4.1 赋值语句

在Verilog HDL语言中，信号有两种赋值方式：

(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。**
3) 这是一种比较常用的赋值方法。（特别在编写可综合模块时）

(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。**
3) 可能会产生意想不到的结果。

非阻塞赋值方式和阻塞赋值方式的区别常给设计人员带来问题。问题主要是给"always"块内的reg型信号的赋值方式不易把握。到目前为止，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;
```

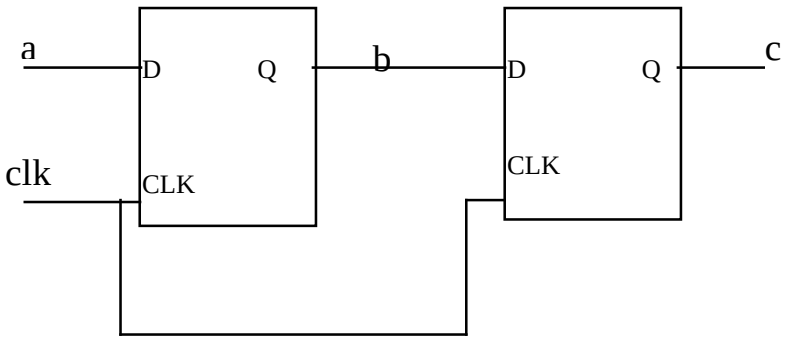
这种方式的赋值并不是马上执行的，也就是说"always"块内的下一条语句执行后，b并不等于a，而是保持原来的值。"always"块结束后，才进行赋值。而另一种赋值方式阻塞赋值方式，如下所示:

```
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。尽管这种方式看起来很直观，但是可能引起麻烦。下面举例说明:

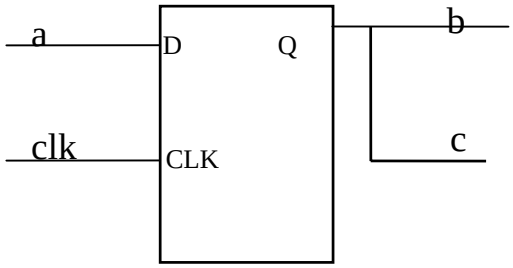
```
[例1]:always @( posedge clk )
              begin
                 b<=a;
                 c<=b;
              end
```

[例1] 中的"always"块中用了非阻塞赋值方式，定义了两个reg型信号b和c，clk信号的上升沿到来时，b就等于a，c就等于b，这里应该用到了两个触发器。请注意：赋值是在"always"块结束后执行的，c应为原来b的值。这个"always"块实际描述的电路功能如下图所示:




```
[例2]: always @(posedge  clk)
            begin
                b=a;
                c=b;
            end
```

[例2]中的 "always"块用了阻塞赋值方式。clk信号的上升沿到来时，将发生如下的变化：b马上取a的值，c马上取b的值(即等于a)，生成的电路图如下所示只用了一个触发器来寄存器a的值，又输出给b和c。这大概不是设计者的初衷，如果采用[例1]所示的非阻塞赋值方式就可以避免这种错误。



关于赋值语句更详细的说明请参阅第七章中深入理解阻塞和非阻塞赋值小节。

3.4.2 块语句

块语句通常用来将两条或多条语句组合在一起，使其在格式上看更象一条语句。块语句有两种，一种是begin_end语句，通常用来标识顺序执行的语句，用它来标识的块称为顺序块。一种是fork_join语句，通常用来标识并行执行的语句，用它来标识的块称为并行块。下面进行详细的介绍。

一．顺序块

顺序块有以下特点：

1） 块内的语句是按顺序执行的，即只有上面一条语句执行完后下面的语句才能执行。
2） 每条语句的延迟时间是相对于前一条语句的仿真时间而言的。
3） 直到最后一条语句执行完，程序流程控制才跳出该语句块。

顺序块的格式如下：

```
begin
    语句1;
    语句2;
    ......
    语句n;
end
```

或

```
begin:块名
     块内声明语句
     语句1;
     语句2;
     ......
     语句n;
end
```

其中：

- 块名即该块的名字，一个标识名。其作用后面再详细介绍。
- 块内声明语句可以是参数声明语句、reg型变量声明语句、integer型变量声明语句、real型变量声明语句。

下面举例说明：

```
[例1]：begin
          areg = breg;
          creg = areg;   //creg的值为breg的值。
       end
```

从该例可以看出，第一条赋值语句先执行，areg的值更新为breg的值，然后程序流程控制转到第二条赋值语句,creg的值更新为areg的值。因为这两条赋值语句之间没有任何延迟时间,creg的值实为breg的值。当然可以在顺序块里延迟控制时间来分开两个赋值语句的执行时间，见[例2]：

```
[例2]： begin
            areg = breg;
            #10 creg = areg;
            //在两条赋值语句间延迟10个时间单位。
        end

[例3]：parameter  d=50;  //声明d是一个参数
        reg [7:0]  r;     //声明r是一个8位的寄存器变量
        begin             //由一系列延迟产生的波形
            #d  r = 'h35;
            #d  r = 'hE2;
            #d  r = 'h00;
            #d  r = 'hF7;
            #d  -> end_wave;  //触发事件end_wave
        end
```

这个例子中用顺序块和延迟控制组合来产生一个时序波形。

二． 并行块

并行块有以下四个特点：

1) 块内语句是同时执行的，即程序流程控制一进入到该并行块，块内语句则开始同时并行地执行。
2) 块内每条语句的延迟时间是相对于程序流程控制进入到块内时的仿真时间的。
3) 延迟时间是用来给赋值语句提供执行时序的。
4) 当按时间时序排序在最后的语句执行完后或一个disable语句执行时，程序流程控制跳出该程序块。

并行块的格式如下：

```
fork
    语句1;
    语句2;
    .......
    语句n;
join

或
fork:块名
块内声明语句
     语句1;
     语句2;
     ......
     语句n;
join
```

其中：

- 块名即标识该块的一个名字，相当于一个标识符。
- 块内说明语句可以是参数说明语句、reg型变量声明语句、integer型变量声明语句、real型变量声明语句、time型变量声明语句、事件(event)说明语句。

下面举例说明：

```
[例4]：fork
            #50   r = 'h35;
            #100  r = 'hE2;
            #150  r = 'h00;
            #200  r = 'hF7;
            #250    ->  end_wave;         //触发事件end_wave.
       join
```

在这个例子中用并行块来替代了前面例子中的顺序块来产生波形，用这两种方法生成的波形是一样的。

三． 块名

在VerilgHDL语言中，可以给每个块取一个名字，只需将名字加在关键词begin或fork后面即可。这样做的原因有以下几点。

1) 这样可以在块内定义局部变量，即只在块内使用的变量。
2) 这样可以允许块被其它语句调用，如被disable语句。
3) 在Verilog语言里，所有的变量都是静态的，即所有的变量都只有一个唯一的存储地址，因此进入或跳出块并不影响存储在变量内的值。

基于以上原因，块名就提供了一个在任何仿真时刻确认变量值的方法。

四. 起始时间和结束时间

在并行块和顺序块中都有一个起始时间和结束时间的概念。对于顺序块，起始时间就是第一条语句开始被执行的时间，结束时间就是最后一条语句执行完的时间。而对于并行块来说，起始时间对于块内所有的语句是相同的，即程序流程控制进入该块的时间，其结束时间是按时间排序在最后的语句执行完的时间。

当一个块嵌入另一个块时，块的起始时间和结束时间是很重要的。至于跟在块后面的语句只有在该块的结束时间到了才能开始执行，也就是说，只有该块完全执行完后，后面的语句才可以执行。

在fork_join块内，各条语句不必按顺序给出，因此在并行块里，各条语句在前还是在后是无关紧要的。见下例：

```
[例5]：fork
            #250  -> end_wave;
            #200  r = 'hF7;
            #150  r = 'h00;
            #100  r = 'hE2;
            #50   r = 'h35;
       join
```

在这个例子中，各条语句并不是按被执行的先后顺序给出的，但同样可以生成前面例子中的波形。

3.5.条件语句

3.5.1. if_else语句

if语句是用来判定所给定的条件是否满足，根据判定的结果（真或假)决定执行给出的两种操作之一。Verilog HDL语言提供了三种形式的if语句。

(1).if(表达式)语句

例如： if (a > b) out1 <= int1;

(2).if(表达式) 语句1
else 语句2

例如：
```
if(a>b)      out1<=int1;
else         out1<=int2;
```

(3).if(表达式1) 语句1;
else if(表达式2) 语句2;
else if(表达式3) 语句3;
.....
else if(表达式m) 语句m;
else 语句n;

例如：
```
if(a>b)  out1<=int1;
     else  if(a==b)  out1<=int2;
     else            out1<=int3;
```

六点说明：

(1).三种形式的if语句中在if后面都有“表达式”，一般为逻辑表达式或关系表达式。系统对表达式的值进行判断，若为0,x,z，按“假”处理，若为1，按“真”处理，执行指定的语句。

(2).第二、第三种形式的if语句中，在每个else前面有一分号，整个语句结束处有一分号。
例如：

```
If (a>b)
    out1 <=int1;
else
    out1 <=int2;
```

各 有 一 个 分 号

这是由于分号是Verilog HDL语句中不可缺少的部分，这个分号是if语句中的内嵌套语句所要求的。如果无此分号，则出现语法错误。但应注意，不要误认为上面是两个语句（if语句和else语句）。它们都属于同一个if语句。else子句不能作为语句单独使用，它必须是if语句的一部分，与if配对使用。

(3).在if和else后面可以包含一个内嵌的操作语句(如上例)，也可以有多个操作语句，此时用begin和end这两个关键词将几个语句包含起来成为一个复合块语句。如：

```
if(a>b)
     begin
       out1<=int1;
       out2<=int2;
     end
else
      begin
        out1<=int2;
        out2<=int1;
      end
```

注意在end后不需要再加分号。因为begin_end内是一个完整的复合语句，不需再附加分号。

(4).允许一定形式的表达式简写方式。如下面的例子：

```
        if(expression)  等同与  if( expression == 1 )
        if(! expression) 等同与  if( expression != 1 )
```

(5).if语句的嵌套

在if语句中又包含一个或多个if语句称为if语句的嵌套。一般形式如下:

```
        if(expression1)
             if(expression2) 语句1     (内嵌if)
             else   语句2
        else
             if(expression3)  语句3    (内嵌if)
             else   语句4
```

应当注意if与else的配对关系，else总是与它上面的最近的if配对。如果if与else的数目不一样,为了实现程序设计者的企图,可以用begin_end块语句来确定配对关系。例如:

```
        if(  )
             begin
             if(    )  语句1     (内嵌if)
             end
        else
             语句2
```

这时begin_end块语句限定了内嵌if语句的范围，因此else与第一个if配对。注意begin_end块语句在if_else语句中的使用。因为有时begin_end块语句的不慎使用会改变逻辑行为。见下例：

```
        if(index>0)
             for(scani=0;scani<index;scani=scani+1)
                  if(memory[scani]>0)
                    begin
                        $display("...");
                        memory[scani]=0;
                    end
        else /*WRONG*/
        $display("error-indexiszero");
```

尽管程序设计者把else写在与第一个if(外层if)同一列上，希望与第一个if对应，但实际上else是与第二个if对应，因为它们相距最近。正确的写法应当是这样的：

```
        if(index>0)
             begin
             for(scani=0;scani<index;scani=scani+1)
                 if(memory[scani]>0)
                    begin
                       $display("...");
                       memory[scani]=0;
                    end
             end

        else   /*WRONG*/
       $display("error-indexiszero");
```

(6).if_else例子。

下面的例子是取自某程序中的一部分。这部分程序用if_else语句来检测变量index以决定三个寄存器modify_segn中哪一个的值应当与index相加作为memory的寻址地址。并且将相加值存入寄存器index以备下次检测使用。程序的前十行定义寄存器和参数。

```
//定义寄存器和参数。
```

```
-----------------------------------------------------------------------------------------------------------------------
------
reg [31:0]  instruction, segment_area[255:0];
reg [7:0]   index;
reg [5:0]   modify_seg1, modify_seg2, modify_seg3;
parameter
       segment1=0,  inc_seg1=1,
       segment2=20, inc_seg2=2,
       segment3=64, inc_seg3=4,
       data=128;
//检测寄存器index的值
if(index<segment2)
       begin
           instruction = segment_area[index + modify_seg1];
           index = index + inc_seg1;
       end
else  if(index<segment3)
       begin
           instruction = segment_area[index + modify_seg2];
           index = index + inc_seg2;
       end
else  if (index<data)
       begin
           instruction = segment_area[index + modify_seg3];
           index = index + inc_seg3;
       end
else
       instruction = segment_area[index];
```

3.5.2. case语句

case语句是一种多分支选择语句，if语句只有两个分支可供选择，而实际问题中常常需要用到多分支选择，Verilog语言提供的case语句直接处理多分支选择。case语句通常用于微处理器的指令译码，它的一般形式如下:

1) case(表达式) <case分支项> endcase
2) casez(表达式)<case分支项> endcase
3) casex(表达式)<case分支项> endcase

case分支项的一般格式如下:

分支表达式: 语句
缺省项(default项): 语句

说明:

a) case括弧内的表达式称为控制表达式，case分支项中的表达式称为分支表达式。控制表达式通常表示为控制信号的某些位，分支表达式则用这些控制信号的具体状态值来表示，因此分支表达式又可以称为常量表达式。
b) 当控制表达式的值与分支表达式的值相等时，就执行分支表达式后面的语句。如果所有的分支表达式的值都没有与控制表达式的值相匹配的，就执行default后面的语句。
c) default项可有可无,一个case语句里只准有一个default项。下面是一个简单的使用case语句的例子。该例子中对寄存器rega译码以确定result的值。

```
reg [15:0]  rega;
reg [9:0]   result;
```

```
case(rega)
16 'd0:  result = 10 'b0111111111;
16 'd1:  result = 10 'b1011111111;
16 'd2:  result = 10 'b1101111111;
16 'd3:  result = 10 'b1110111111;
16 'd4:  result = 10 'b1111011111;
16 'd5:  result = 10 'b1111101111;
16 'd6:  result = 10 'b1111110111;
16 'd7:  result = 10 'b1111111011;
16 'd8:  result = 10 'b1111111101;
16 'd9:  result = 10 'b1111111110;
default:   result = 'bx;
endcase
```

d) 每一个case分项的分支表达式的值必须互不相同，否则就会出现矛盾现象(对表达式的同一个值，有多种执行方案)。
e) 执行完case分项后的语句，则跳出该case语句结构，终止case语句的执行。
f) 在用case语句表达式进行比较的过程中，只有当信号的对应位的值能明确进行比较时，比较才能成功。因此要注意详细说明case分项的分支表达式的值。
g) case语句的所有表达式的值的位宽必须相等，只有这样控制表达式和分支表达式才能进行对应位的比较。一个经常犯的错误是用'bx, 'bz 来替代 n'bx, n'bz，这样写是不对的，因为信号x, z的缺省宽度是机器的字节宽度，通常是32位(此处 n 是case控制表达式的位宽)。

下面将给出 case, casez, casex 的真值表：

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

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

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

case语句与if_else_if语句的区别主要有两点:

1) 与case语句中的控制表达式和多分支表达式这种比较结构相比，if_else_if结构中的条件表达式更为直观一些。
2) 对于那些分支表达式中存在不定值x和高阻值z位时,case语句提供了处理这种情况的手段。下面的两个例子介绍了处理x,z值位的case语句。

[例1]:

```
case ( select[1:2] )
  2 'b00:  result = 0;
  2 'b01:  result = flaga;
  2 'b0x,
  2 'b0z:  result = flaga? 'bx : 0;
  2 'b10:  result = flagb;
  2 'bx0,
  2 'bz0:  result = flagb? 'bx : 0;
  default: result = 'bx;
endcase
```

[例2]:

```
        case(sig)
          1 'bz:    $display("signal is floating");
          1 'bx:    $display("signal is unknown");
          default:  $display("signal is %b", sig);
        endcase
```

Verilog HDL针对电路的特性提供了case语句的其它两种形式用来处理case语句比较过程中的不必考虑的情况(don't care condition)。其中casez语句用来处理不考虑高阻值z的比较过程，casex语句则将高阻值z和不定值都视为不必关心的情况。所谓不必关心的情况，即在表达式进行比较时，不将该位的状态考虑在内。这样在case语句表达式进行比较时，就可以灵活地设置以对信号的某些位进行比较。见下面的两个例子:

```
[例3]: reg[7:0] ir;
        casez(ir)
           8 'b1???????: instruction1(ir);
           8 'b01??????: instruction2(ir);
           8 'b00010???: instruction3(ir);
           8 'b000001??: instruction4(ir);
        endcase

[例4]: reg[7:0] r, mask;
        mask = 8'bx0x0x0x0;
        casex(r^mask)
           8 'b001100xx: stat1;
           8 'b1100xx00: stat2;
           8 'b00xx0011: stat3;
           8 'bxx001100: stat4;
        endcase
```

3.5.3.由于使用条件语句不当在设计中生成了原本没想到有的锁存器

Verilog HDL设计中容易犯的一个通病是由于不正确使用语言，生成了并不想要的锁存器。下面我们给出了一个在“always"块中不正确使用if语句，造成这种错误的例子。

```
always @(al or d)
 begin
    if(al)q<=d;
 end
```

有锁存器

```
always @(al or d)
 begin
    if(al) q<=d;
    else   q<=0
 end
```

无锁存器

检查一下左边的"always"块，if语句保证了只有当al=1时，q才取d的值。这段程序没有写出 al = 0 时的结果，那么当al=0时会怎么样呢？

在"always"块内，如果在给定的条件下变量没有赋值，这个变量将保持原值，也就是说会生成一个锁存器！

如果设计人员希望当 al = 0 时q的值为0，else项就必不可少了，请注意看右边的"always"块，整个Verilog程序模块综合出来后，"always"块对应的部分不会生成锁存器。

Verilog HDL程序另一种偶然生成锁存器是在使用case语句时缺少default项的情况下发生的。

case语句的功能是：在某个信号（本例中的sel)取不同的值时，给另一个信号（本例中的q)赋不同的值。注意看下图左边的例子，如果sel=0,q取a值，而sel=11,q取b的值。这个例子中不清楚的是：如果sel取00和11以外的值时q将被赋予什么值？在下面左边的这个例子中，程序是用Verilog HDL写的，即默认为q保持原值，这就会自动生成锁存器。

```
always @(sel[1:0] or a or b)
   case(sel[1:0])
   2'b00:    q<=a;
   2'b11:    q<=b;
   endcase
```

有 锁 存 器

```
always @(sel[1:0] or a or b)
   case(sel[1:0])
     2'b00:    q<=a;
     2'b11:    q<=b;
     default:    q<='b0;
   endcase
```

无 锁 存 器

右边的例子很明确，程序中的case语句有default项，指明了如果sel不取00或11时，编译器或仿真器应赋给q的值。程序所示情况下，q赋为0,因此不需要锁存器。

以上就是怎样来避免**偶然生成锁存器的错误。如果用到if语句，最好写上else项。如果用case语句，最好写上default项。遵循上面两条原则，就可以避免发生这种错误，使设计者更加明确设计目标，同时也增强了Verilog程序的可读性。**

3.6.循环语句

在Verilog HDL中存在着四种类型的循环语句，用来控制执行语句的执行次数。

1) forever　　连续的执行语句。
2) repeat　　连续执行一条语句 n 次。
3) while　　执行一条语句直到某个条件不满足。如果一开始条件即不满足(为假)，则语句一次也不能被执行。
4) for通过以下三个步骤来决定语句的循环执行。
   a) 先给控制循环次数的变量赋初值。
   b) 判定控制循环的表达式的值，如为假则跳出循环语句，如为真则执行指定的语句后，转到第三步。
   c) 执行一条赋值语句来修正控制循环变量次数的变量的值，然后返回第二步。

下面对各种循环语句详细的进行介绍。

3.6.1.forever语句

forever语句的格式如下：

```
forever  语句;  或
```

```
          forever  begin  多条语句 end
```

forever循环语句常用于产生周期性的波形，用来作为仿真测试信号。它与always语句不同处在于不能独立写在程序中，而必须写在initial块中。其具体使用方法将在"事件控制"这一小节里详细地加以说明。

3.6.2.repeat语句

repeat语句的格式如下:

```
          repeat(表达式) 语句;  或
          repeat(表达式)   begin 多条语句 end
```

在repeat语句中，其表达式通常为常量表达式。下面的例子中使用repeat循环语句及加法和移位操作来实现一个乘法器。

```
        parameter size=8,longsize=16;
        reg [size:1] opa, opb;
        reg [longsize:1] result;

          begin: mult
            reg [longsize:1] shift_opa, shift_opb;
            shift_opa = opa;
            shift_opb = opb;
            result = 0;
            repeat(size)
              begin
               if(shift_opb[1])
                    result = result + shift_opa;

               shift_opa = shift_opa <<1;
               shift_opb = shift_opb >>1;
              end
          end
```

3.6.3.while语句

while语句的格式如下：

```
      while(表达式)  语句
```

或用如下格式：

```
      while(表达式)  begin  多条语句   end
```

下面举一个while语句的例子，该例子用while循环语句对rega这个八位二进制数中值为1的位进行计数。

```
        begin:      count1s
          reg[7:0] tempreg;
          count=0;
          tempreg = rega;
          while(tempreg)
            begin
              if(tempreg[0])  count = count + 1;
```

```
                tempreg = tempreg>>1;
             end
        end
```

3.6.4.for语句

for语句的一般形式为：

```
for（表达式1；表达式2；表达式3）  语句
```

它的执行过程如下：

1) 先求解表达式1；
2) 求解表达式2，若其值为真（非0），则执行for语句中指定的内嵌语句，然后执行下面的第3步。若为假(0)，则结束循环，转到第5步。
3) 若表达式为真，在执行指定的语句后，求解表达式3。
4) 转回上面的第2步骤继续执行。
5) 执行for语句下面的语句。

for语句最简单的应用形式是很易理解的，其形式如下：

```
for(循环变量赋初值；循环结束条件；循环变量增值)
    执行语句
```

for循环语句实际上相当于采用while循环语句建立以下的循环结构：

```
begin
   循环变量赋初值；
   while(循环结束条件)
       begin
           执行语句
           循环变量增值;
       end
end
```

这样对于需要8条语句才能完成的一个循环控制，for循环语句只需两条即可。

下面分别举两个使用for循环语句的例子。例1用for语句来初始化memory。例2则用for循环语句来实现前面用repeat语句实现的乘法器。

```
[例1]：begin:init_mem
         reg[7:0] tempi;
           for(tempi=0;tempi<memsize;tempi=tempi+1)
               memory[tempi]=0;
       end

[例2]：parameter  size = 8, longsize = 16;
        reg[size:1] opa, opb;
        reg[longsize:1] result;

        begin:mult
            integer bindex;
            result=0;
            for( bindex=1; bindex<=size; bindex=bindex+1 )
                  if(opb[bindex])
                     result = result + (opa<<(bindex-1));
        end
```

在for语句中，循环变量增值表达式可以不必是一般的常规加法或减法表达式。下面是对rega这个八位二进制数中值为1的位进行计数的另一种方法。见下例：

```
begin: count1s
    reg[7:0] tempreg;
    count=0;
    for( tempreg=rega; tempreg; tempreg=tempreg>>1 )
         if(tempreg[0])
             count=count+1;
end
```

3.7.结构说明语句

Verilog语言中的任何过程模块都从属于以下四种结构的说明语句。

1) initial说明语句
2) always说明语句
3) task说明语句
4) function说明语句

initial和always说明语句在仿真的一开始即开始执行。initial语句只执行一次。相反，always语句则是不断地重复执行，直到仿真过程结束。在一个模块中，使用initial和always语句的次数是不受限制的。task和function语句可以在程序模块中的一处或多处调用。其具体使用方法以后再详细地加以介绍。这里只对initial和always语句加以介绍。

3.7.1.initial语句

initial语句的格式如下：

```
initial
     begin
          语句1;
          语句2;
          ......
          语句n;
     end
```

举例说明：

[例1]：

```
initial
  begin
     areg=0;       //初始化寄存器areg
     for(index=0;index<size;index=index+1)
         memory[index]=0;       //初始化一个memory
  end
```

在这个例子中用initial语句在仿真开始时对各变量进行初始化。

[例2]：

```
initial
  begin
       inputs = 'b000000;      //初始时刻为0
       #10 inputs = 'b011001;
```

```
        #10 inputs = 'b011011;
        #10 inputs = 'b011000;
        #10 inputs = 'b001000;
    end
```

从这个例子中，我们可以看到initial语句的另一用途，即用initial语句来生成激励波形作为电路的测试仿真信号。一个模块中可以有多个initial块，它们都是并行运行的。initial块常用于测试文件和虚拟模块的编写，用来产生仿真测试信号和设置信号记录等仿真环境。

3.7.2.always语句

always语句在仿真过程中是不断重复执行的。

其声明格式如下：

always〈时序控制〉〈语句〉

always语句由于其不断重复执行的特性，只有和一定的时序控制结合在一起才有用。如果一个always语句没有时序控制，则这个always语句将会发成一个仿真死锁。见下例：

[例1]：always areg = ~areg;

这个always语句将会生成一个0延迟的无限循环跳变过程，这时会发生仿真死锁。如果加上时序控制，则这个always语句将变为一条非常有用的描述语句。见下例：

[例2]：always #half_period areg = ~areg;

这个例子生成了一个周期为:period(=2*half_period) 的无限延续的信号波形，常用这种方法来描述时钟信号，作为激励信号来测试所设计的电路。

```
[例3]：reg[7:0] counter;
       reg tick;
       always @(posedge areg)
          begin
             tick = ~tick;
             counter = counter + 1;
          end
```

这个例子中,每当areg信号的上升沿出现时把tick信号反相，并且把counter增加1。这种时间控制是always语句最常用的。

always 的时间控制可以是沿触发也可以是电平触发的，可以单个信号也可以多个信号，中间需要用关键字 or 连接，如：

```
always @(posedge clock or posedge reset)  //由两个沿触发的always块
   begin
   ......
   end

always @( a or b or c )                   //由多个电平触发的always块
  begin
   ......
```

```
end
```

沿触发的always块常常描述时序逻辑，如果符合可综合风格要求可用综合工具自动转换为表示时序逻辑的寄存器组和门级逻辑，而电平触发的always块常常用来描述组合逻辑和带锁存器的组合逻辑，如果符合可综合风格要求可转换为表示组合逻辑的门级逻辑或带锁存器的组合逻辑。一个模块中可以有多个always块，它们都是并行运行的。

3.7.3.task和function说明语句

task和function说明语句分别用来定义任务和函数。利用任务和函数可以把一个很大的程序模块分解成许多较小的任务和函数便于理解和调试。输入、输出和总线信号的值可以传入、传出任务和函数。任务和函数往往还是大的程序模块中在不同地点多次用到的相同的程序段。学会使用task和function语句可以简化程序的结构，使程序明白易懂，是编写较大型模块的基本功。

一．task和function说明语句的不同点

任务和函数有些不同，主要的不同有以下四点：

1） 函数只能与主模块共用同一个仿真时间单位，而任务可以定义自己的仿真时间单位。
2） 函数不能启动任务，而任务能启动其它任务和函数。
3） 函数至少要有一个输入变量，而任务可以没有或有多个任何类型的变量。
4） 函数返回一个值，而任务则不返回值。

函数的目的是通过返回一个值来响应输入信号的值。任务却能支持多种目的，能计算多个结果值，这些结果值只能通过被调用的任务的输出或总线端口送出。Verilog HDL模块使用函数时是把它当作表达式中的操作符，这个操作的结果值就是这个函数的返回值。下面让我们用例子来说明：

例如，定义一任务或函数对一个16位的字进行操作让高字节与低字节互换，把它变为另一个字(假定这个任务或函数名为: switch_bytes)。

任务返回的新字是通过输出端口的变量，因此16位字字节互换任务的调用源码是这样的：

```
switch_bytes(old_word,new_word);
```

任务switch_bytes把输入old_word的字的高、低字节互换放入new_word端口输出。

而函数返回的新字是通过函数本身的返回值，因此16位字字节互换函数的调用源码是这样的：

```
new_word = switch_bytes(old_word);
```

下面分两节分别介绍任务和函数语句的要点。

二． task说明语句

如果传给任务的变量值和任务完成后接收结果的变量已定义，就可以用一条语句启动任务。任务完成以后控制就传回启动过程。如任务内部有定时控制，则启动的时间可以与控制返回的时间不同。任务可以启动其它的任务，其它任务又可以启动别的任务，可以启动的任务数是没有限制的。不管有多少任务启动，只有当所有的启动任务完成以后，控制才能返回。

1） 任务的定义

定义任务的语法如下：

任务:

```
task <任务名>;
    <端口及数据类型声明语句>
    <语句1>
    <语句2>
    .....
    <语句n>
endtask
```

这些声明语句的语法与模块定义中的对应声明语句的语法是一致的。

2) 任务的调用及变量的传递

启动任务并传递输入输出变量的声明语句的语法如下：

任务的调用：

```
<任务名>(端口1，端口2，...，端口n);
```

下面的例子说明怎样定义任务和调用任务：

任务定义：

```
task  my_task;
  input a, b;
  inout  c;
  output d, e;
  ...
  <语句>    //执行任务工作相应的语句
  ...
  c = foo1;//赋初始值
  d = foo2;//对任务的输出变量赋值t
  e = foo3;
endtask
```

任务调用：

```
my_task(v,w,x,y,z);
```

任务调用变量(v,w,x,y,z)和任务定义的I/O变量(a,b,c,d,e)之间是一一对应的。当任务启动时，由v,w,和x.传入的变量赋给了a,b,和c，而当任务完成后的输出又通过c,d和e赋给了x,y和z。下面是一个具体的例子用来说明怎样在模块的设计中使用任务，使程序容易读懂：

```
module traffic_lights;
  reg  clock, red, amber, green;
  parameter  on=1, off=0, red_tics=350,
  amber_tics=30,green_tics=200;
       //交通灯初始化
       initial   red=off;
       initial   amber=off;
       initial   green=off;
   //交通灯控制时序
       always
        begin
              red=on;          //开红灯
              light(red,red_tics);  //调用等待任务
              green=on;        //开绿灯
              light(green,green_tics);     //等待
              amber=on;        //开黄灯
              light(amber,amber_tics);     //等待
```

```
            end
        //定义交通灯开启时间的任务
        task  light (color,tics);
          output  color;
          input[31:0] tics;
        begin
          repeat(tics) @(posedge clock);//等待tics个时钟的上升沿
          color=off;//关灯
          end
        endtask
        //产生时钟脉冲的always块
        always
          begin
            #100 clock=0;
            #100 clock=1;
          end
      endmodule
```

这个例子描述了一个简单的交通灯的时序控制，并且该交通灯有它自己的时钟产生器。

二．function说明语句

函数的目的是返回一个用于表达式的值。

- 定义函数的语法:

```
function <返回值的类型或范围> (函数名);
    <端口说明语句>
    <变量类型说明语句>
        begin
        <语句>
        ........
        end
endfunction
```

请注意<返回值的类型或范围>这一项是可选项，如缺省则返回值为一位寄存器类型数据。下面用例子说明：

```
function [7:0] getbyte;
input [15:0] address;
begin
  <说明语句>              //从地址字中提取低字节的程序
  getbyte = result_expression; //把结果赋予函数的返回字节
end
endfunction
```

- 从函数返回的值

函数的定义蕴含声明了与函数同名的、函数内部的寄存器。如在函数的声明语句中<返回值的类型或范围>为缺省,则这个寄存器是一位的，否则是与函数定义中<返回值的类型或范围>一致的寄存器。函数的定义把函数返回值所赋值寄存器的名称初始化为与函数同名的内部变量。下面的例子说明了这个概念：getbyte被赋予的值就是函数的返回值。

- 函数的调用

函数的调用是通过将函数作为表达式中的操作数来实现的。

其调用格式如下：

<函数名> (<表达式><,<表达式>>*)

其中函数名作为确认符。下面的例子中通过对两次调用函数getbyte的结果值进行位拼接运算来生成一个字。

```
word = control? {getbyte(msbyte),getbyte(lsbyte)} : 0;
```

- 函数的使用规则

与任务相比较函数的使用有较多的约束，下面给出的是函数的使用规则：

1) 函数的定义不能包含有任何的时间控制语句，即任何用＃、@、或wait来标识的语句。
2) 函数不能启动任务。
3) 定义函数时至少要有一个输入参量。
4) 在函数的定义中必须有一条赋值语句给函数中的一个内部变量赋以函数的结果值，该内部变量具有和函数名相同的名字。

- 举例说明

下面的例子中定义了一个可进行阶乘运算的名为factorial的函数，该函数返回一个32位的寄存器类型的值，该函数可后向调用自身，并且打印出部分结果值。

```
module  tryfact;
      //函数的定义------------------------------
     function[31:0]factorial;
       input[3:0]operand;
       reg[3:0]index;
       begin
          factorial = operand? 1 : 0;
          for(index=2;index<=operand;index=index+1)
          factorial = index * factorial;
       end
     endfunction
//函数的测试-------------------------------------
reg[31:0]result;
reg[3:0]n;
initial
begin
   result=1;
   for(n=2;n<=9;n=n+1)
   begin
      $display("Partial result n= %d result= %d", n, result);
      result = n * factorial(n)/((n*2)+1);
   end
 $display("Finalresult=%d",result);
end
endmodule//模块结束
```

前面我们已经介绍了足够的语句类型可以编写一些完整的模块。在下一章里，我们将举许多实际的例子进行介绍。这些例子都给出了完整的模块描述，因此可以对它们进行仿真测试和结果检验。通过学习和练习我们就能逐步掌握利用Verilog HDL设计数字系统的方法和技术。

3.8.系统函数和任务

Verilog HDL语言中共有以下一些系统函数和任务：

$bitstoreal, $rtoi, $display, $setup, $finish, $skew, $hold,

$setuphold, $itor, $strobe, $period, $time, $printtimescale,
$timefoemat, $realtime, $width, $real tobits, $write, $recovery,
在Verilog HDL语言中每个系统函数和任务前面都用一个标识符$来加以确认。这些系统函数和任务提供了非常强大的功能。有兴趣的同学可以参阅附录：Verilog语言参考手册。下面对一些常用的系统函数和任务逐一加以介绍。

3.8.1.$display和$write任务

格式：

```
$display(p1,p2,....pn);
$write(p1,p2,....pn);
```

这两个函数和系统任务的作用是用来输出信息，即将参数p2到pn按参数p1给定的格式输出。参数p1通常称为“格式控制”，参数p2至pn通常称为“输出表列”。这两个任务的作用基本相同。$display自动地在输出后进行换行,$write则不是这样。如果想在一行里输出多个信息，可以使用$write。在$display和$write中，其输出格式控制是用双引号括起来的字符串，它包括两种信息：

- 格式说明，由"%"和格式字符组成。它的作用是将输出的数据转换成指定的格式输出。格式说明总是由“%”字符开始的。对于不同类型的数据用不同的格式输出。表一中给出了常用的几种输出格式。

表一

输出格式	说明
%h或%H	以十六进制数的形式输出
%d或%D	以十进制数的形式输出
%o或%O	以八进制数的形式输出
%b或%B	以二进制数的形式输出
%c或%C	以ASCII码字符的形式输出
%v或%V	输出网络型数据信号强度
%m或%M	输出等级层次的名字
%s或%S	以字符串的形式输出
%t或%T	以当前的时间格式输出
%e或%E	以指数的形式输出实型数
%f或%F	以十进制数的形式输出实型数
%g或%G	以指数或十进制数的形式输出实型数 无论何种格式都以较短的结果输出

- 普通字符，即需要原样输出的字符。其中一些特殊的字符可以通过表二中的转换序列来输出。下面表中的字符形式用于格式字符串参数中，用来显示特殊的字符。

表二：

换码序列	功能
\n	换行
\t	横向跳格(即跳到下一个输出区)
\\	反斜杠字符\

\"	双引号字符"
\o	1到3位八进制数代表的字符
%%	百分符号%

在$display和$write的参数列表中，其“输出表列”是需要输出的一些数据，可以是表达式。下面举几个例子说明一下。

```
[例1]：module  disp;
         initial
           begin
            $display("\\\t%%\n\"\123");
           end
       endmodule
```

输出结果为

```
       \%
       "S
```

从上面的这个例子中可以看到一些特殊字符的输出形式(八进制数123就是字符S)。

```
[例2]：module disp;
        reg[31:0] rval;
        pulldown(pd);
        initial
         begin
         rval=101;
          $display("rval=%h hex %d decimal", rval, rval);
          $display("rval=%o otal %b binary", rval, rval);
          $display("rval has %c ascii character value",rval);
          $display("pd strength value is %v",pd);
          $display("current scope is %m");
          $display("%s is ascii value for 101",101);
          $display("simulation time is %t",$time);
         end
endmodule
```

其输出结果为：

```
rval=00000065 hex 101 decimal
rval=00000000145 octal 00000000000000000000000001100101 binary
rval has e ascii character value
pd strength value is StX
current scope is disp
e is ascii value for 101
simulation time is 0
```

输出数据的显示宽度

在$display中,输出列表中数据的显示宽度是自动按照输出格式进行调整的。这样在显示输出数据时，在经过格式转换以后，总是用表达式的最大可能值所占的位数来显示表达式的当前值。在用十进制数格式输出时，输出结果前面的0值用空格来代替。对于其它进制，输出结果前面的0仍然显示出来。例如对于一个值的位宽为12位的表达式，如按照十六进制数输出，则输出结果占3个字符的位置，如按照十进制数输出,则输出结果占4个字符的位置。这是因为这个表达式的最大可能值为FFF(十六进制)、

4095(十进制)。可以通过在%和表示进制的字符中间插入一个0自动调整显示输出数据宽度的方式。见下例：

```
$display("d=%0h a=%0h",data,addr);
```

这样在显示输出数据时，在经过格式转换以后，总是用最少的位数来显示表达式的当前值。下面举例说明：

```
[例3]：module printval;
          reg[11:0]r1;
          initial
           begin
               r1=10;
               $display("Printing with maximum size=%d=%h",r1,r1);
               $display("Printing with minimum size=%0d=%0h",r1,r1);
           end
       enmodule
```

输出结果为：

```
Printing with maximum size=10=00a:
printing with minimum size=10=a;
```

如果输出列表中表达式的值包含有不确定的值或高阻值，其结果输出遵循以下规则：

(1).在输出格式为十进制的情况下：

- 如果表达式值的所有位均为不定值，则输出结果为小写的x。
- 如果表达式值的所有位均为高阻值，则输出结果为小写的z。
- 如果表达式值的部分位为不定值，则输出结果为大写的X。
- 如果表达式值的部分位为高阻值，则输出结果为大写的Z。

(2).在输出格式为十六进制和八进制的情况下：

- 每4位二进制数为一组代表一位十六进制数，每3位二进制数为一组代表一位八进制数。
- 如果表达式值相对应的某进制数的所有位均为不定值，则该位进制数的输出的结果为小写的x。
- 如果表达式值相对应的某进制数的所有位均为高阻值，则该位进制数的输出结果为小写的z。
- 如果表达式值相对应的某进制数的部分位为不定值，则该位进制数输出结果为大写的X。
- 如果表达式值相对应的某进制数的部分位为高阻值，则该位进制数输出结果为大写的Z。

对于二进制输出格式，表达式值的每一位的输出结果为0、1、x、z。下面举例说明：

语句输出结果：

```
$display("%d", 1'bx);                                          输出结果为：x
$display("%h", 14'bx0_1010);                                   输出结果为：xxXa
$display("%h %o",12'b001x_xx10_1x01,12'b001_xxx_101_x01);      输出结果为：XXX 1x5X
```

注意：因为$write在输出时不换行，要注意它的使用。可以在$write中加入换行符\n，以确保明确的输出显示格式。

3.8.2.系统任务$monitor

格式：

```
$monitor(p1,p2,.....，pn);
$monitor;
$monitoron;
$monitoroff;
```

任务$monitor提供了监控和输出参数列表中的表达式或变量值的功能。其参数列表中输出控制格式字符串和输出表列的规则和$display中的一样。当启动一个带有一个或多个参数的$monitor任务时，仿真器则建立一个处理机制，使得每当参数列表中变量或表达式的值发生变化时，整个参数列表中变量或表达式的值都将输出显示。如果同一时刻，两个或多个参数的值发生变化，则在该时刻只输出显示一次。但在$monitor中，参数可以是$time系统函数。这样参数列表中变量或表达式的值同时发生变化的时刻可以通过标明同一时刻的多行输出来显示。如：

```
$monitor($time,,"rxd=%b txd=%b",rxd,txd);
```

在$display中也可以这样使用。注意在上面的语句中，“,,"代表一个空参数。空参数在输出时显示为空格。

$monitoron和$monitoroff任务的作用是通过打开和关闭监控标志来控制监控任务$monitor的启动和停止,这样使得程序员可以很容易的控制$monitor何时发生。其中$monitoroff任务用于关闭监控标志，停止监控任务$monitor，$monitoron则用于打开监控标志，启动监控任务$monitor。通常在通过调用$monitoron启动$monitor时，不管$monitor参数列表中的值是否发生变化，总是立刻输出显示当前时刻参数列表中的值，这用于在监控的初始时刻设定初始比较值。在缺省情况下，控制标志在仿真的起始时刻就已经打开了。**在多模块调试的情况下，许多模块中都调用了$monitor,因为任何时刻只能有一个$monitor起作用，因此需配合$monitoron与$monitoroff使用，把需要监视的模块用$monitoron打开，在监视完毕后及时用$monitoroff关闭，以便把$monitor 让给其他模块使用。$monitor与$display的不同处还在于$monitor往往在initial块中调用，只要不调用$monitoroff,$monitor便不间断地对所设定的信号进行监视。**

3.8.3.时间度量系统函数$time

在Verilog HDL中有两种类型的时间系统函数：$time和$realtime。用这两个时间系统函数可以得到当前的仿真时刻。

- 系统函数$time

$time可以返回一个64比特的整数来表示的当前仿真时刻值。该时刻是以模块的仿真时间尺度为基准的。下面举例说明。

```
[例1]：`timescale  10ns/1ns
       module  test;
         reg  set;
         parameter  p=1.6;
         initial
           begin
             $monitor($time,,"set=",set);
             #p set=0;
             #p set=1;
           end
      endmodule
```

输出结果为：

```
0 set=x
2 set=0
3 set=1
```

在这个例子中，模块test想在时刻为16ns时设置寄存器set为0，在时刻为32ns时设置寄存器set为1。但是由$time记录的set变化时刻却和预想的不一样。这是由下面两个原因引起的：

1) $time显示时刻受时间尺度比例的影响。在上面的例子中，时间尺度是10ns，因为$time输出的时刻总是时间尺度的倍数，这样将16ns和32ns输出为1.6和3.2。
2) 因为$time总是输出整数，所以在将经过尺度比例变换的数字输出时，要先进行取整。在上面的例子中，1.6和3.2经取整后为2和3输出。注意：时间的精确度并不影响数字的取整。

- $realtime系统函数

$realtime和$time的作用是一样的，只是$realtime返回的时间数字是一个实型数，该数字也是以时间尺度为基准的。下面举例说明：

```
[例2]: `timescale10ns/1ns
       module test;
         reg set;
         parameter  p=1.55;
         initial
           begin
             $monitor($realtime,,"set=",set);
             #p set=0;
             #p set=1;
           end
       endmodule
```

输出结果为：

```
0 set=x
1.6 set=0
3.2 set=1
```

从上面的例子可以看出，$realtime将仿真时刻经过尺度变换以后即输出，不需进行取整操作。所以$realtime返回的时刻是实型数。

3.8.4.系统任务$finish

格式：

```
$finish;
$finish(n);
```

系统任务$finish的作用是退出仿真器，返回主操作系统，也就是结束仿真过程。任务$finish可以带参数，根据参数的值输出不同的特征信息。如果不带参数，默认$finish的参数值为1。下面给出了对于不同的参数值，系统输出的特征信息：

0 不输出任何信息
1 输出当前仿真时刻和位置
2 输出当前仿真时刻，位置和在仿真过程中
所用memory及CPU时间的统计

3.8.5.系统任务$stop

格式：

```
$stop;
$stop(n);
```

$stop任务的作用是把EDA工具(例如仿真器)置成暂停模式，在仿真环境下给出一个交互式的命令提示符，将控制权交给用户。这个任务可以带有参数表达式。根据参数值(0，1或2)的不同，输出不同的信息。参数值越大，输出的信息越多。

3.8.6.系统任务$readmemb和$readmemh

在Verilog HDL程序中有两个系统任务$readmemb和$readmemh用来从文件中读取数据到存贮器中。这两个系统任务可以在仿真的任何时刻被执行使用，其使用格式共有以下六种：

1) $readmemb("<数据文件名>",<存贮器名>);
2) $readmemb("<数据文件名>",<存贮器名>,<起始地址>);
3) $readmemb("<数据文件名>",<存贮器名>,<起始地址>,<结束地址>);
4) $readmemh("<数据文件名>",<存贮器名>);
5) $readmemh("<数据文件名>",<存贮器名>,<起始地址>);
6) $readmemh("<数据文件名>",<存贮器名>,<起始地址>,<结束地址>);

在这两个系统任务中，被读取的数据文件的内容只能包含：空白位置(空格，换行，制表格(tab)和form-feeds)，注释行(//形式的和/*...*/形式的都允许)，二进制或十六进制的数字。数字中不能包含位宽说明和格式说明，对于$readmemb系统任务，每个数字必须是二进制数字，对于$readmemh系统任务，每个数字必须是十六进制数字。数字中不定值x或X，高阻值z或Z，和下划线(_)的使用方法及代表的意义与一般Verilog HDL程序中的用法及意义是一样的。另外数字必须用空白位置或注释行来分隔开。

在下面的讨论中，地址一词指对存贮器(memory)建模的数组的寻址指针。当数据文件被读取时，每一个被读取的数字都被存放到地址连续的存贮器单元中去。存贮器单元的存放地址范围由系统任务声明语句中的起始地址和结束地址来说明，每个数据的存放地址在数据文件中进行说明。当地址出现在数据文件中，其格式为字符“@”后跟上十六进制数。如：

```
@hh...h
```

对于这个十六进制的地址数中，允许大写和小写的数字。在字符“@”和数字之间不允许存在空白位置。可以在数据文件里出现多个地址。当系统任务遇到一个地址说明时，系统任务将该地址后的数据存放到存贮器中相应的地址单元中去。

对于上面六种系统任务格式,需补充说明以下五点：

1) 如果系统任务声明语句中和数据文件里都没有进行地址说明，则缺省的存放起始地址为该存贮器定义语句中的起始地址。数据文件里的数据被连续存放到该存贮器中，直到该存贮器单元存满为止或数据文件里的数据存完。
2) 如果系统任务中说明了存放的起始地址，没有说明存放的结束地址，则数据从起始地址开始存放，存放到该存贮器定义语句中的结束地址为止。
3) 如果在系统任务声明语句中，起始地址和结束地址都进行了说明，则数据文件里的数据按该起始地址开始存放到存贮器单元中，直到该结束地址，而不考虑该存贮器的定义语句中的起始地址和结束地址。
4) 如果地址信息在系统任务和数据文件里都进行了说明，那么数据文件里的地址必须在系统任务中地址参数声明的范围之内。否则将提示错误信息，并且装载数据到存贮器中的操作被中断。
5) 如果数据文件里的数据个数和系统任务中起始地址及结束地址暗示的数据个数不同的话，也要提示错误信息。

下面举例说明：

先定义一个有256个地址的字节存贮器 mem:

```
reg[7:0] mem[1:256];
```

下面给出的系统任务以各自不同的方式装载数据到存贮器mem中。

```
initial  $readmemh("mem.data",mem);
initial  $readmemh("mem.data",mem,16);
initial  $readmemh("mem.data",mem,128,1);
```

第一条语句在仿真时刻为0时，将装载数据到以地址是1的存贮器单元为起始存放单元的存贮器中去。第二条语句将装载数据到以单元地址是16的存贮器单元为起始存放单元的存贮器中去，一直到地址是256的单元为止。第三条语句将从地址是128的单元开始装载数据，一直到地址为1的单元。在第三种情况中，当装载完毕，系统要检查在数据文件里是否有128个数据，如果没有，系统将提示错误信息。

3.8.7.系统任务 $random

这个系统函数提供了一个产生随机数的手段。当函数被调用时返回一个32bit的随机数。它是一个带符号的整形数。

$random一般的用法是：$ramdom % b ,其中 b>0.它给出了一个范围在（-b+1):(b-1)中的随机数。下面给出一个产生随机数的例子：

```
reg[23:0] rand;
rand = $random % 60;
```

上面的例子给出了一个范围在－59到59之间的随机数，下面的例子通过位并接操作产生一个值在0到59之间的数。

```
reg[23:0] rand;
rand = {$random} % 60;
```

利用这个系统函数可以产生随机脉冲序列或宽度随机的脉冲序列，以用于电路的测试。下面例子中的Verilog HDL模块可以产生宽度随机的随机脉冲序列的测试信号源,在电路模块的设计仿真时非常有用。同学们可以根据测试的需要，模仿下例，灵活使用$random系统函数编制出与实际情况类似的随机脉冲序列。

[例]
```
`timescale 1ns/1ns
module random_pulse( dout );
output [9:0] dout;
reg dout;
integer delay1,delay2,k;
  initial
    begin
     #10 dout=0;
     for (k=0; k< 100; k=k+1)
       begin
       delay1 = 20 * ( {$random} % 6);
       // delay1 在0到100ns间变化
       delay2 = 20 * ( 1 + {$random} % 3);
       // delay2 在20到60ns间变化
```

```
                #delay1  dout = 1 << ({$random} %10);
                //dout的0--9位中随机出现1，并出现的时间在0-100ns间变化
                #delay2  dout = 0;
                //脉冲的宽度在在20到60ns间变化
                end
            end
        endmodule
```

3.9.编译预处理

Verilog HDL语言和C语言一样也提供了编译预处理的功能。“编译预处理”是Verilog HDL编译系统的一个组成部分。Verilog HDL语言允许在程序中使用几种特殊的命令(它们不是一般的语句)。Verilog HDL编译系统通常先对这些特殊的命令进行“预处理”，然后将预处理的结果和源程序一起在进行通常的编译处理。

在Verilog HDL语言中，为了和一般的语句相区别，这些预处理命令以符号“`”开头(注意这个符号是不同于单引号“’”的)。这些预处理命令的有效作用范围为定义命令之后到本文件结束或到其它命令定义替代该命令之处。Verilog HDL提供了以下预编译命令：

`accelerate，`autoexpand_vectornets，`celldefine，`default_nettype，`define，`else，`endcelldefine,`endif,`endprotect,`endprotected,`expand_vectornets,`ifdef,`include,`noaccelerate,`noexpand_vectornets ，`noremove_gatenames ，`noremove_netnames ，`nounconnected_drive，`protect，`protecte，`remove_gatenames，`remove_netnames，`reset,`timescale，`unconnected_drive

在这一小节里只对常用的`define、`include、`timescale进行介绍，其余的请查阅参考书。

3.9.1.宏定义 `define

用一个指定的标识符(即名字)来代表一个字符串，它的一般形式为:

```
`define 标识符(宏名) 字符串(宏内容)

如：`define signal string
```

它的作用是指定用标识符signal来代替string这个字符串，在编译预处理时，把程序中在该命令以后所有的signal都替换成string。这种方法使用户能以一个简单的名字代替一个长的字符串，也可以用一个有含义的名字来代替没有含义的数字和符号，因此把这个标识符(名字)称为“宏名”，在编译预处理时将宏名替换成字符串的过程称为“宏展开”。`define是宏定义命令。

```
[例1]：`define  WORDSIZE 8
       module
       reg[1:`WORDSIZE]  data;  //这相当于定义 reg[1:8] data;
```

关于宏定义的八点说明：

1) 宏名可以用大写字母表示，也可以用小写字母表示。建议使用大写字母，以与变量名相区别。

2) `define命令可以出现在模块定义里面，也可以出现在模块定义外面。宏名的有效范围为定义命令之后到原文件结束。通常，`define命令写在模块定义的外面，作为程序的一部分，在此程序内有效。

3) 在引用已定义的宏名时，必须在宏名的前面加上符号“`”，表示该名字是一个经过宏定义的名字。

4) 使用宏名代替一个字符串，可以减少程序中重复书写某些字符串的工作量。而且记住一个宏名要比记住一个无规律的字符串容易，这样在读程序时能立即知道它的含义，当需要改变某一个变量时，可以只改变`define命令行，一改全改。如例1中，先定义WORDSIZE代表常量8，这时寄存器data是一个8位的寄存器。如果需要改变寄存器的大小，只需把该命令行改为：`define WORDSIZE 16。这样寄存器data则变为一个16位的寄存器。由此可见使用宏定义，可以提高程序的可移植性和可读性。

5) 宏定义是用宏名代替一个字符串，也就是作简单的置换，不作语法检查。预处理时照样代入，不管含义是否正确。只有在编译已被宏展开后的源程序时才报错。

6) **宏定义不是Verilog HDL语句，不必在行末加分号。如果加了分号会连分号一起进行置换。**如：

```
[例2]：module  test;
        reg  a, b, c, d, e, out;
       `define  expression  a+b+c+d;
          assign out = `expression + e;
            ...
       endmodule
```

经过宏展开以后，该语句为：

```
assign  out = a+b+c+d;+e;
```

显然出现语法错误。

7) 在进行宏定义时，可以引用已定义的宏名，可以层层置换。如：

```
[例3]：module test;
         reg  a, b, c;
         wire out;
         `define aa a + b
         `define cc c + `aa
               assign out = `cc;
      endmodule
```

这样经过宏展开以后，assign语句为

```
assign  out = c + a + b;
```

8) 宏名和宏内容必须在同一行中进行声明。如果在宏内容中包含有注释行，注释行不会作为被置换的内容。如：

```
[例4]: module
         `define typ_nand nand #5  //define a nand with typical delay
          `typ_nand g121(q21,n10,n11);
            .........
       endmodule
```

经过宏展开以后，该语句为：

```
nand #5 g121(q21,n10,n11);
```

宏内容可以是空格，在这种情况下，宏内容被定义为空的。当引用这个宏名时，不会有内容被置换。

注意：组成宏内容的字符串不能够被以下的语句记号分隔开的。

- 注释行

- 数字
- 字符串
- 确认符
- 关键词
- 双目和三目字符运算符

如下面的宏定义声明和引用是非法的。

```
`define  first_half  "start of string
$display(`first_half end of string");
```

注意在使用宏定义时要注意以下情况：

1) 对于某些 EDA软件，在编写源程序时，如使用和预处理命令名相同的宏名会发生冲突，因此建议不要使用和预处理命令名相同的宏名。
2) 宏名可以是普通的标识符(变量名)。例如signal_name 和 'signal_name的意义是不同的。但是这样容易引起混淆，建议不要这样使用。

3.9.2.“文件包含”处理`include

所谓“文件包含”处理是一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。Verilog HDL语言提供了`include命令用来实现“文件包含”的操作。其一般形式为:

```
`include “文件名”
```

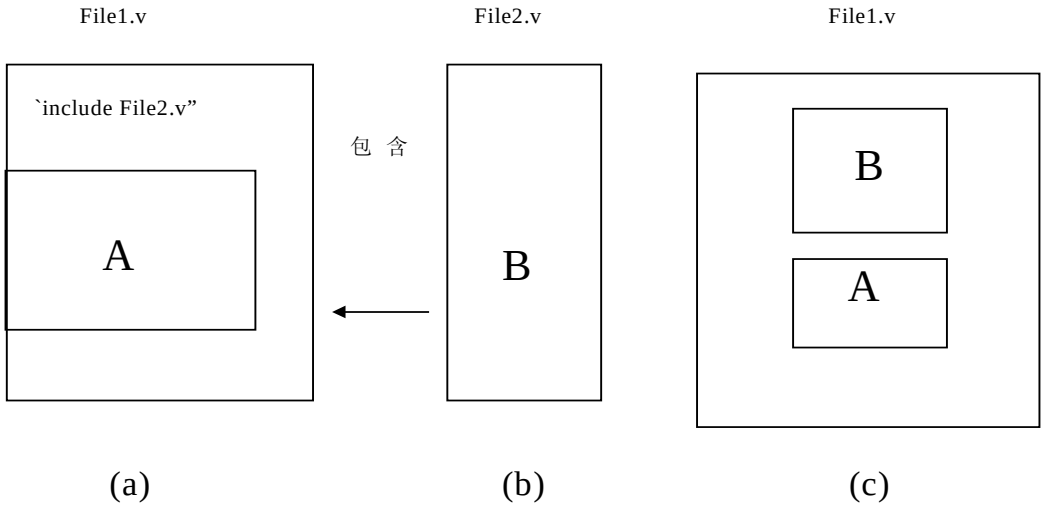


图 3－9－2

图3-9-2表示“文件包含”的含意。图3-9-2(a)为文件File1.v,它有一个`include "File2.v"命令，然后还有其它的内容(以A表示)。图3-9-2(b)为另一个文件File2.v,文件的内容以B表示。在编译预处理时，要对`include命令进行“文件包含”预处理:将File2.v的全部内容复制插入到 `include "File2.v"命令出现的地方，即File2.v 被包含到File1.v中，得到图3-9-2(c)所示的结果。在接着往下进行的编译中，将“包含”以后的File1.v作为一个源文件单位进行编译。

“文件包含”命令是很有用的，它可以节省程序设计人员的重复劳动。可以将一些常用的宏定义命令或任务(task)组成一个文件，然后用`include命令将这些宏定义包含到自己所写的源文件中，相当于

工业上的标准元件拿来使用。另外在编写Verilog HDL源文件时，一个源文件可能经常要用到另外几个源文件中的模块，遇到这种情况即可用`include命令将所需模块的源文件包含进来。

[例1]：
(1)文件aaa.v
```
module aaa(a,b,out);
  input a, b;
  output out;
  wire out;
    assign  out = a^b;
endmodule
```

(2)文件 bbb.v
```
`include  "aaa.v"
module  bbb(c,d,e,out);
  input  c,d,e;
  output  out;
  wire  out_a;
  wire  out;
   aaa  aaa(.a(c),.b(d),.out(out_a));
    assign  out=e&out_a;
endmodule
```

在上面的例子中，文件bbb.v用到了文件aaa.v中的模块aaa的实例器件，通过“文件包含”处理来调用。模块aaa实际上是作为模块bbb的子模块来被调用的。在经过编译预处理后，文件bbb.v实际相当于下面的程序文件bbb.v：

```
module aaa(a,b,out);
  input a, b;
  output  out;
  wire out;
    assign  out = a ^ b;
endmodule

module bbb( c, d, e, out);
  input c, d, e;
  output out;
  wire out_a;
  wire out;
   aaa  aaa(.a(c),.b(d),.out(out_a));
    assign out= e & out_a;
endmodule
```

关于“文件包含”处理的四点说明：

1) 一个`include命令只能指定一个被包含的文件，如果要包含n个文件，要用n个`include命令。注意下面的写法是非法的`include"aaa.v""bbb.v"
2) `include命令可以出现在Verilog HDL源程序的任何地方，被包含文件名可以是相对路径名，也可以是绝对路径名。例如：'include"parts/count.v"
3) 可以将多个`include命令写在一行，在`include命令行，只可以出空格和注释行。例如下面的写法是合法的。

```
'include "fileB" 'include "fileC" //including fileB and fileC
```

4) 如果文件1包含文件2，而文件2要用到文件3的内容，则可以在文件1用两个`include命令分别包含文件2和文件3，而且文件3应出现在文件2之前。例如在下面的例子中，即在file1.v中定义：

```
`include"file3.v"
`include"file2.v"

module test(a,b,out);
input[1:`size2] a, b;
output[1:`size2] out;
wire[1:`size2] out;
assign  out= a+b;
endmodule

file2.v的内容为：
`define size2  `size1+1
.
.
.

file3.v的内容为：
`define size1  4
.
.
.
```

这样，file1.v和file2.v都可以用到file3.v的内容。在file2.v中不必再用 `include "file3.v" 了。

5) 在一个被包含文件中又可以包含另一个被包含文件，即文件包含是可以嵌套的。例如上面的问题也可以这样处理，见图3-9-3.

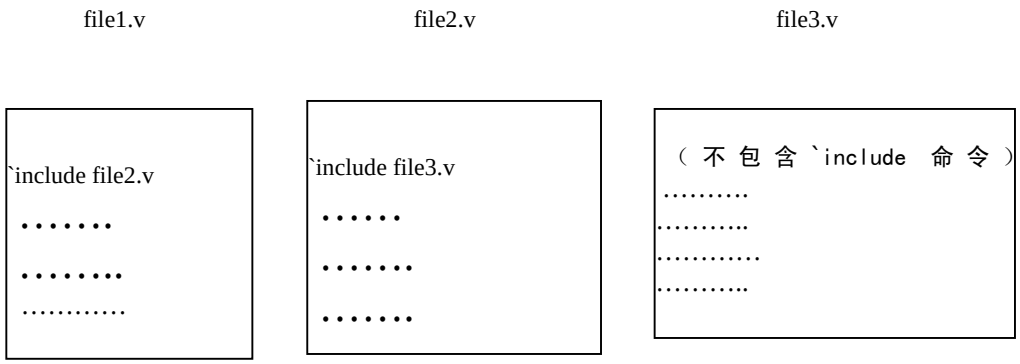


图 3-9-3

它的作用和图3-9-4的作用是相同的。

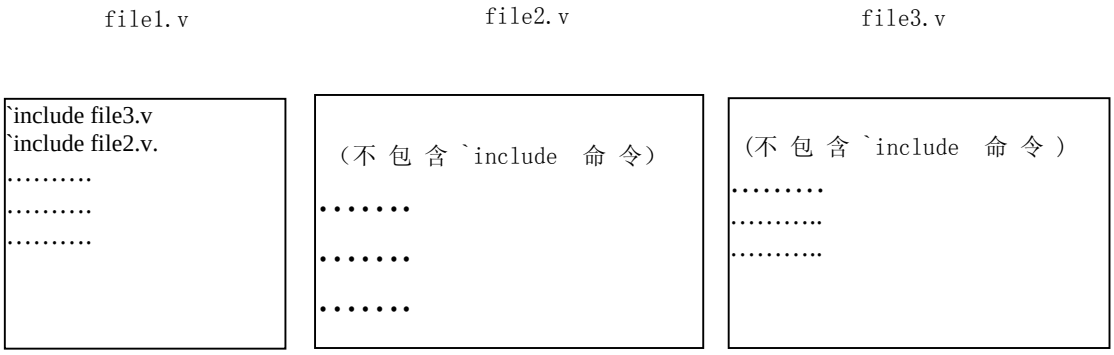


图3-9-4

3.9.3.时间尺度 `timescale

`timescale命令用来说明跟在该命令后的模块的时间单位和时间精度。使用`timescale命令可以在同一个设计里包含采用了不同的时间单位的模块。例如，一个设计中包含了两个模块，其中一个模块的时间延迟单位为ns，另一个模块的时间延迟单位为ps。EDA工具仍然可以对这个设计进行仿真测试。

`timescale 命令的格式如下：

`timescale<时间单位>/<时间精度>

在这条命令中，时间单位参量是用来定义模块中仿真时间和延迟时间的基准单位的。时间精度参量是用来声明该模块的仿真时间的精确程度的，该参量被用来对延迟时间值进行取整操作(仿真前)，因此该参量又可以被称为取整精度。如果在同一个程序设计里，存在多个`timescale命令，则用最小的时间精度值来决定仿真的时间单位。另外时间精度至少要和时间单位一样精确，时间精度值不能大于时间单位值。

在`timescale命令中，用于说明时间单位和时间精度参量值的数字必须是整数，其有效数字为1、10、100，单位为秒(s)、毫秒(ms)、微秒(us)、纳秒(ns)、皮秒(ps)、毫皮秒(fs)。这几种单位的意义说明见下表。

时间单位	定义
s	秒(1S)
ms	千分之一秒(10^{-3}S)
us	百万分之一秒(10^{-6}S)
ns	十亿分之一秒(10^{-9}S)
ps	万亿分之一秒(10^{-12}S)
fs	千万亿分之一秒(10^{-15}S)

下面举例说明`timescale命令的用法。

[例1]： `timescale 1ns/1ps
在这个命令之后，模块中所有的时间值都表示是1ns的整数倍。这是因为在`timescale命令中，定义了时间单位是1ns。模块中的延迟时间可表达为带三位小数的实型数，因为`timescale命令定义时间精度为1ps.

[例2]： `timescale 10us/100ns
在这个例子中，`timescale命令定义后，模块中时间值均为10us的整数倍。因为`timesacle 命令定义的时间单位是10us。延迟时间的最小分辨度为十分之一微秒(100ns)，即延迟时间可表达为带一位小数的实型数。

例3： `timescale 10ns/1ns

```
module  test;
reg  set;
parameter  d=1.55;
   initial
     begin
       #d set=0;
       #d set=1;
     end
endmodule
```

在这个例子中，`timescale命令定义了模块test的时间单位为10ns、时间精度为1ns。因此在模块test中，所有的时间值应为10ns的整数倍，且以1ns为时间精度。这样经过取整操作，存在参数d中的延迟时间实际是16ns(即1.6×10ns)，这意味着在仿真时刻为16ns时寄存器set被赋值0，在仿真时刻为32ns时寄存器set被赋值1。仿真时刻值是按照以下的步骤来计算的。

1) 根据时间精度，参数d值被从1.55取整为1.6。
2) 因为时间单位是10ns，时间精度是1ns，所以延迟时间#d作为时间单位的整数倍为16ns。
3) EDA工具预定在仿真时刻为16ns的时候给寄存器set赋值0(即语句 #d set=0;执行时刻)，在仿真时刻为32ns的时候给寄存器set赋值1(即语句 #d set=1;执行时刻)，

注意：如果在同一个设计里，多个模块中用到的时间单位不同，需要用到以下的时间结构。

1) 用`timescale命令来声明本模块中所用到的时间单位和时间精度。
2) 用系统任务$printtimescale来输出显示一个模块的时间单位和时间精度。
3) 用系统函数$time和$realtime及%t格式声明来输出显示EDA工具记录的时间信息。

3.9.4.条件编译命令`ifdef、`else、`endif

一般情况下，Verilog HDL源程序中所有的行都将参加编译。但是有时希望对其中的一部分内容只有在满足条件才进行编译，也就是对一部分内容指定编译的条件，这就是“条件编译”。有时，希望当满足条件时对一组语句进行编译，而当条件不满足是则编译另一部分。

条件编译命令有以下几种形式：

1) `ifdef 宏名 (标识符)
   程序段1
   `else
   程序段2
   `endif

它的作用是当宏名已经被定义过(用`define命令定义)，则对程序段1进行编译，程序段2将被忽略；否则编译程序段2，程序段1被忽略。其中`else部分可以没有，即：

2) `ifdef 宏名 (标识符)

```
            程序段1
            `endif
```

这里的 “宏名” 是一个Verilog HDL的标识符，“程序段”可以是Verilog HDL语句组，也可以是命令行。这些命令可以出现在源程序的任何地方。**注意：被忽略掉不进行编译的程序段部分也要符合Verilog HDL程序的语法规则。**

通常在Verilog HDL程序中用到`ifdef、`else、`endif编译命令的情况有以下几种：

- 选择一个模块的不同代表部分。
- 选择不同的时序或结构信息。
- 对不同的EDA工具，选择不同的激励。

3.10.小结

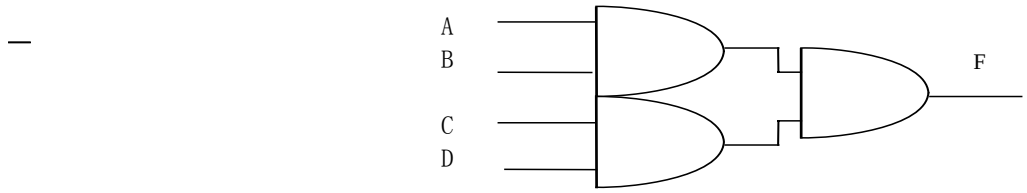
Verilog HDL的语法与C语言的语法有许多类似的地方，但也有许多不同的地方。我们学习Verilog HDL语法要善于找到不同点，着重理解如：阻塞〔Blocking〕和非阻塞〔Non-Blocking〕赋值的不同；顺序块和并行块的不同；块与块之间的并行执行的概念；task和function的概念。Verilog HDL还有许多系统函数和任务也是C语言中没有的如:$monitor、$readmemb、$stop等等，而这些系统任务在调试模块的设计中是非常有用的，我们只有通过阅读大量的Verilog调试模块实例，经过长期的实践，经常查阅附录中的Verilog语言参考手册才能逐步掌握，。

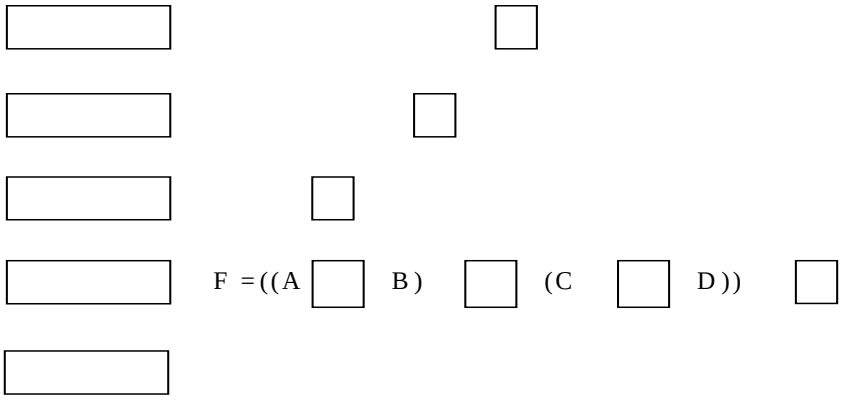
3.11.思考题

在这一小结中我们将针对以上介绍的基本语法做一些练习。希望读者能在仔细阅读以上的内容后，认真思考以下的习题，这将有效地帮助你正确地理解基本语法的要点。

1）以下给出了一个填空练习，请将所给各个选项根据电路图，填入程序中的适当位置。

```
assign    module ; ~ | & input output
 inputs    outputs    endmodule
 A , B , C , D
 AOI  ( A, B, C, D, F )
```

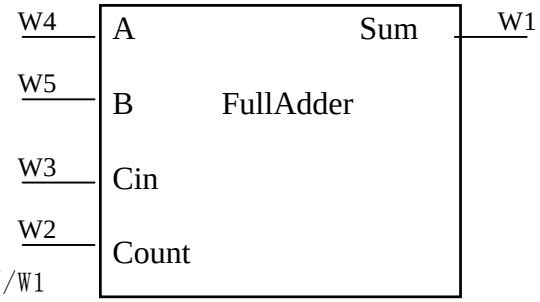




标准答案：

```
module  AOI(A,B,C,D,F);
input   A,B,C,D;
output  F;
assign  F = ((A&B)&(C&D));
endmodule
```

2〕 在这一题中，我们将作有关层次电路的练习，通过这个练习，你将加深对模块间调用时，管脚间连接的理解。假设已有全加器模块FullAdder,若有一个顶层模块调用此全加器，连接线分别为W4，W5，W3，W1和W2。请在调用时正确地填入I/O的对应信号。




```
module FullAdder(A,B,Cin,Sum,Cout);
input A, B, Cin;
output Sum, Cout;

endmodule
module Top.....
  FullAdder FA(
                        ,//W1
                        ,//W2
                        ,//W3
                        ,//W4
                        );//W5
endmodule
```

标准答案：

```
moduleTop...
FullAdderFA(    .Sum(W1), //W1
                .Cout(W2),//W2
                .Cin(W3), //W3
                .A(W4),   //W4
                .B(W5));  //W5
endmodule
```

3)下面这道题是一个测试模块，因此没有输入输出端口，请将相应项填入合适的位置。

```
module  TestFixture;
        [                          ]   MUX2  M (SEL , A , B , F)
        [                          ]   reg  A, B , SEL;
                                       wire  F;
initial
begin
        [                          ]   $monitor (SEL , A ,B , ,F) ;

end
initial
        [                          ]   SEL=0; A=0;  B=0;
                                       #10   A=1;
                                       #10   SEL=1;   #10  B=1;
endmodule
```

标准答案：

```
module  TestFixture
reg  A,B,SEL;
wire F;
MUX2M(SEL,A,B,F);
initial
begin
  SEL=0; A=0; B=0;
  #10  A=1;
```

```
------------------------------------------------------------------------------------------------------------------------
------
  #10  SEL=1; #10 B=1;
end
initial
  $monitor(SEL,A,B,,F);
endmodule
```

4)指出下面几个信号的最高位和最低位。

```
reg [1:0] SEL; input [0:2] IP; wire [16:23] A;
```

标准答案:

```
MSB:SEL[1] MSB:IP[0] MSB:A[16]
LSB:SEL[0] LSB:IP[2] LSB:A[23]
```

5)P,Q,R都是4bit的输入矢量，下面哪一种表达形式是正确的。

```
1)input P[3:0],Q,R;
2)input P,Q,R[3:0];
3)input P[3:0],Q[3:0],R[3:0];
4)input [3:0] P,[3:0]Q,[0:3]R;
5)input [3:0] P,Q,R;
```

标准答案:5)

6)请将下面选项中的正确答案填人空的方括号中。

1. (0:2) 2. (P:0) 3. (Op1:Op2) 4.(7:7) 5. (2:0) 6. (7:0)

```
reg [7:0] A;
reg [2:0] Sum, Op1, Op2;
reg P, OneBit;

initial
begin
Sum=Op1+Op2;
P=1;
A[ ]=Sum;
.....
end
```

标准答案:5

7)请根据以下两条语句，从选项中找出正确答案。

```
7.1) reg [7:0] A;
       A=2'hFF;
```

1) 8'b0000_0011 2) 8'h03 3) 8'b1111_1111 4) 8'b11111111

标准答案:1)

```
7.2) reg [7:0] B;
       B=8'bZ0;
```

1) 8'0000_00Z0 2) 8'bZZZZ_0000
3) 8'b0000_ZZZ0 4) 8'bZZZZ_ZZZ0

标准答案:4)

8)请指出下面几条语句中变量的类型。

```
8.1) assign A=B;
8.2) always #1
       Count=C+1;
```

标准答案:
A(wire) B(wire/reg) Count(reg) C(wire/reg)

9)指出下面模块中Cin，Cout，C3，C5，的类型。

```
module  FADD(A,B,Cin,Sum,Cout);
input  A, B, Cin;
output Sum, Cout;
....
endmodule
module Test;
...
FADDM(C1,C2,C3,C4,C5);
...
endmodule
```

标准答案:
Cin(wire) Cout(wire/reg) C3(wire/reg) C5(wire)

10〕在下一个程序段中，当ADDRESS的值等于5'b0X000时，问casex执行完后A和B的值是多少。

```
A=0;
B=0;
casex(ADDRESS)
5'b00???:  A=1;
5'b01???:  B=1;
5'b10?00,5'b11?00:
            begin
               A=1;
               B=1;
            end
endcase
```

标准答案: A=1 and B=0;

11)在下题中，事件A分别在10，20，30发生，而B一直保持X状态，问在50时Count的值是多少。

```
reg [7:0] Count;
initial
   Count=0;
always
   begin
   @(A)  Count=Count+1;
   @(B)  Count=Count+1;
   end
```

标准答案：Count=1;
(这是因为当A第一次发生时，Count的值由0变为1，然后事件控制 @(B) 阻挡了进程。)

12)在下题中initial块执行完后I，J，A，B的值会是多少。

```
reg [2:0] A;
reg [3:0] B;
integer I, J;
 initial
  begin
     I=0;
     A=0;
     I=I-1;
```

```
                    J=I;
                    A=A-1;
                    B=A;
                    J=J+1;
                    B=B+1;
                  end
```

标准答案:

```
I=-1          (整数可为负数)
J=0
A=7           (A为reg型为非负数，又因为A为3位即为111)
B=8           (在B＝A时，B＝0111,然后B＝B＋1，所以B=4'b1000)
```

13)在下题中，当V的值发生变化且为-1时，执行完always块后
Count的值应是多少？

```
          reg[7:0]V;
          reg[2:0]Count;

          always @(V)
          begin
          Count=0;
          while(～V[Count])
          Count=Count+1;
          end
```

标准答案:Count=0;

14)在下题中循环执行完后，V的值是多少？

```
            reg [3:0] A;
            reg V ,W;
            integer K;
            ....
            A=4'b1010;
            for(K=2;K>=0;K=K-1)
               begin
                  V=V^A[k];
                  W=A[K]^A[K+1];
               end
```

标准答案:V的值是它进人循环体前值的取反。
(因为V的值与0，1，0 进行了异或，与1的异或改变了V的值。)

15)在下题中，给出了几种硬件实现，问以下的模块被综合后可能是哪一种?

```
       always @(posedge Clock)
           if(A)
               C=B;
```

1.不能综合。
2.一个上升沿触发器和一个多路器。
3.一个输入是A，B，Clock的三输入与门。
4.一个透明锁存器。
5.一个带clock有始能引脚的上升沿触发器。

标准答案:2,5

16)在下题中，always状态将描述一个带异步Nreset和Nset输入端的上升沿触发器，则空括号内应填入什么，可从以下五种答案中选择。

```
        always @(          )
        if(!Nreset)
        Q<=0;
        else if(!Nset)
        Q<=1;
        else
        Q<=D;
        1.negedge Nset or posedge Clock
        2.posedge Clock
        3.negedge Nreset or posedge Clock
        4.negedge Nreset or negedge Nset or posedge Clock
        5.negedge Nreset or negedge Nset
```

标准答案:4

17)在下题中，给出了几种硬件实现，问以下的模块被综合后可能是哪一种?

1.带异步复位端的触发器。
2.不能综合或与预先设想的不一致。
3.组合逻辑。
4.带逻辑的透明锁存器。
5.带同步复位端的触发器。

```
1.always @(posedge Clock)
        begin
           A<=B;
           if(C)
              A<=1'b0;
        end
```

标准答案：5

```
2.always @( A or B)
      case(A)
         1'b0: F=B;
         1'b1: G=B;
      endcase
```

标准答案:2

```
3.always @( posedge  A or posedge B )
      if(A)
         C<=1'b0;
       else
         C<=D;
```

标准答案:1

```
4.always @(posedge Clk or negedge Rst)
     if(Rst)
       A<=1'b0;
     else
       A<=B;
```

标准答案:2 (产生了异步逻辑)

18)在下题中,模块被综合后将产生几个触发器?

```
      always @(posedge Clk)
          begin: Blk
```

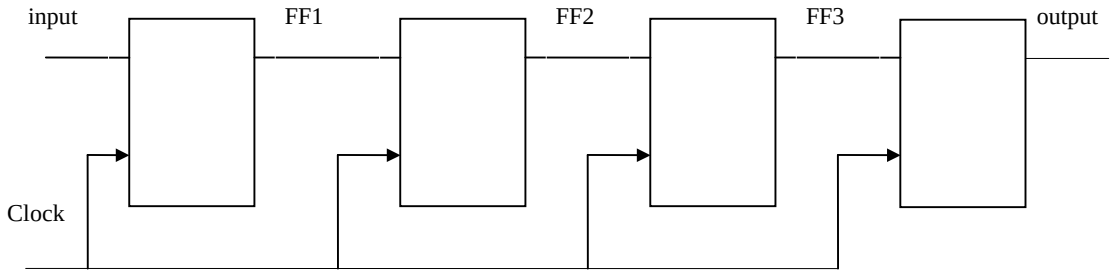
```
            reg B, C;
               C = B;
               D <= C;
               B = A;
         end
```

1. 2个寄存器 B 和 D
2. 2个寄存器 B和 C
3. 3个寄存器 B, C 和 D
4. 1个寄存器 D
5. 2个寄存器 C 和D

标准答案:2

19)在下题中，各条语句的顺序是错误的。请根据电路图调整好它们的次序。

```
  Output =FF3 ;
reg  FF1 , FF2 ,  FF3 ;
      FF2 =FF1 ;
  always @ (posedge Clock)
  end
      FF3 = FF2 ;
   begin
     FF1 = Input;
```



标准答案:

```
reg FF1,FF2,FF3;
always @(posedge Clock)
begin
   Output= FF3;
   FF3 = FF2;
   FF2 = FF1;
   FF1 = Input;
end
```

---

20)根据左表中SEL与OP的对应关系，在右边模块的空括号中填入相应的值。

```
SEL：OP
000：1
001：3
010：1
011：3
100：0
101：3
110：0
111：3
```

```
casex(SEL)
3'b(  ):  OP=3;
3'b(  ):  OP=1;
3'b(  ):  OP=0;
endcase
```

标准答案:

```
casex(SEL)
3'bXX1: OP=3;
3'b0X0: OP=1;
3'b1X0: OP=0;
endcase
```

21)在以下表达式中选出正确的.

1) 4'b1010 & 4'b1101 = 1'b1
2) ～4'b1100 = 1'b1
3) !4'b1011 || !4'b0000 = 1'b1
4) & 4'b1101 = 1'b1
5) 1b'0 || 1b'1 = 1'b1
6) 4'b1011 && 4'b0100 = 4'b1111
7) 4'b0101<<1 =5'b01011
8) !4'b0010 is 1'b0
9) 4'b0001 || 4'b0000 = 1'b1

标准答案:3), 5), 8), 9)

22)在下一个模块旁的括号中填入display的正确值。

```
integerI;
reg[3:0]A;
reg[7:0]B;
initial
begin
I=-1;A=I;B=A;
$display("%b",B);(    )
A=A/2;
$display("%b",A);(    )
B=A+14
$diaplay("%d",B);(    )
A=A+14;
$display("%d",A);(    )
A=-2;I=A/2;
$display("%d",I);(    )
end
```

标准答案：

```
I=-1;A=I;B=A;
```

```
$display("%b",B);(00001111)
A=A/2;
$display("%b",A);(0111)
B=A+14
$diaplay("%d",B);(21)
A=A+14;
$display("%d",A);(5)（A为4位,所以21被截为5）
A=-2;I=A/2;
$display("%d",I);(7)(A=-2,则是1110)
```

23)请问{1,0}与下面哪一个值相等。

1). 2'b01 2). 2'b10 3). 2'b00

4). 64'H0000000000002 5). 64'H0000000100000000

标准答案:5

（位拼接运算符必须指明位数，若不指明则隐含着为32位的 二进制数[即整数]。）

24)根据下题给出的程序，确定应将哪一个选项填入尖括号内。

1. defs.Reset 2."defs.v".Reset

3. M.Reset 4.Reset

```
module defs;
   parameter Reset = 8'b10100101;
endmodule                                (file defs.v)
```

```
module   M  ;
.....
   if   (OP==<    >)
   Bus = 0 ;                             (file M.v)
endmodule
```

1 标准答案：1

(模块间调用时，若引用其他模块定义的参数，要加上其他模块名，做为这个参数的前缀。)

```
module M
'include "defs.v"
....
if(OP==<defs.Reset>)
Bus=0;
endmodule
```

2. 标准答案:4

```
parameter Reset=8'b10100101; (File defs.v)
module M
'include "defs.v"
....
if(OP==<Reset>)
```

```
            Bus=0;
            endmodule
```

25)如果调用Pipe时,想把Depth的值变为8，问程序中的空括号内应填入何值?

```
            Module Pipe(IP,OP)
            parameter Option=1;
            parameter Depth=1;
            ...
            endmodule
            Pipe( ) P1(IP1,OP1);
```

标准答案:#(1,8)
（其中1对应参数Option,8对应参数Depth.）

26)若想使P1中的Depth的值变为16，则应向空括号中填入哪个选项。

```
            module Pipe (IP ,OP);
                parameter Option =1;
                parameter Depth = 1;
                ...........
            endmodule

            module
                Pipe P1(IP1 ,OP1);
                (            );
            endmodule
```

1.defparam P1.Depth=16;
2.parameter P1.Depth=16;
3.parameter Pipe.Depth=16;
4.defparam Pipe.Depth=16;
标准答案:1
(用后缀改变引用模块的参数要用defparam及用本模块名作为引用参数的前缀，如p1.Depth。)

27)如果我们想在Test的monitor语句中观察Count的值，则在空括号中应填入什么?

```
            Module Test
            Top T();
            initial
            $monitor(    )
            endmodule

            module Top;
            Block B1();
            Block B2();
            endmodule

            module Block;
            Counter C();
            endmodule
```

```
        module Counter;
        reg [3:0] Count;
        ....
        endmodule
```

标准答案:T.B1.C.Countor Test.T.B1.C.Count

28) 下题中用initial块给reg[7:0]V符值，请指明每种情况下V的8位都是什值。
这道题说明在数的表示时，已标明字宽的数若用XZ表示某些位，只有在最左边的X或Z具有扩展性。

```
 Reg  [ 7 :0 ] V

initial
begin
     V = 8'b0;
     V = 8'b1;
     V = 8'bX;
     V = 8'BZX;
     V = 8'BXXZZ;
     V = 8'b1X;
end
```

标 准 答 案 ：

```
        8'b00000000
        8'b00000001
        8'bXXXXXXXX
        8'bZZZZZZZX
        8'BXXXXXXZZ
        8'b0000001X
```