第3章 Verilog HDL 基础

Verilog HDL 是一种硬件描述语言，以文本形式来描述数字系统硬件的结构和行为。用它可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。本章主要对 Verilog HDL 相关的语法知识和代码编写规范等内容进行介绍。

3.1 Verilog HDL 门级描述

门级建模比较接近电路底层，设计时主要考虑使用到了哪些门，然后按照一定的连接线组成一个大的电路，所以注重的是门的使用，关键的语法在于门的实例化引用。一个完整的门级描述实例一般包含模块定义、端口声明、内部连线声明、门级调用几个部分。程序 3.1 中给出了如图 3.1 所示电路图的门级描述实例。

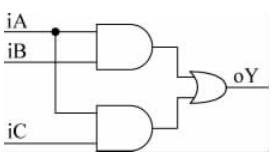

图 3.1 门级描述实例电路图

程序 3.1 门级描述实例

```
module logic_gates(oY,iA,iB,iC);
      output oY;
      input  iA,iB,iC;

      and  (and1,iA,iB);
      and  (and2,iA,iC);
      or   (oY,and1,and2);
endmodule
```

下面将结合程序 3.1 中的例子对门级描述基本语法进行介绍。

3.1.1 模块定义

模块定义以关键字 module 开始，以关键字 endmodule 结束，在这两个关键字之间的代码被识别为一个模块，描述一个具有某种基本功能的电路模型，其基本语法结构如下：

```
module 模块名(端口名 1,端口名 2,…);
…
endmodule
```

这种定义方式与 C 语言中的主函数定义类似，其目的就是为了标识一个代码的界限。

module 和 endmodule 是 Verilog HDL 的基本语法，除部分编译指令语法之外的任何 Verilog HDL 代码都要写在这两个关键字之中。按语法来说，在一个 Verilog HDL 源文件中可以编写多个模块，即一个“.v”文件中可以包含多个 module，但是为了便于管理，一般在一个“.v”文件中仅编写一个 module。在关键字 module 之后还要跟上一个字符串作为该模块的名称，与 module 以空格隔开。这个名称是设计者自己来定义的，只要满足语法要求都可以作为模块名称来使用。此类由设计者自己定义的字符串称为标识符。程序 3.1 中的 logic_gates 就是这个模块的名称。

在模块的名称之后还可以有端口列表，如上述代码中的(oY,iA,iB,iC)就是端口列表。端口是模块和外界环境交换数据的接口，端口列表中必须出现本模块所具有的全部输入和输出端口，端口列表一般都是分为输入和输出两部分来书写，可以先写输入端口后写输出端口，也可以反过来。端口列表用括号区分，括号内部写出所有的端口，每个端口的名称可以自己命名，属于标识符。不同的端口之间以逗号隔开，仅列出名称，而不用体现该端口所具有的位宽。当把 module 关键字、模块名称、端口列表都写完后，需要在此行的末尾添加一个分号，作为本行结束的标志，模块的定义也就完成了。

在模块定义中，自定义的标识符需要满足一定的语法规范。Verilog HDL 中的标识符由字母、数字、下画线(_)和美元符($)组成。标识符是区分大小写的。Verilog HDL 标识符的第一个字符必须是字母或下画线，不能以数字或美元符开始。以美元符开始的标识符是为系统函数保留的。另外还有一些 Verilog HDL 基本语法中使用到的关键字作为保留字，是不能用作标识符的，如 always、and、assign、include、automatic、initial、begin、inout、buf、input、bufif0、instance、bufif1、integer、signed、case、join、small、casex、large、specify、casez、specparam、cell、library、cmos、localparam、strong、config、supply0、deassign、supply1、default、module、defparam、nand、task、design、negedge、time、disable、nmos、else、edge、posedge、use、primitive、endtask、wait、event、for、pulldown、weak0、force、forever、while、fork、wire、function、generate、real、xnor、xor、reg、release、if、repeat 等。

3.1.2　端口声明

模块定义中的端口列表仅列出了本模块具有哪些端口，但这些端口是输入还是输出并没有定义，这就需要在模块中声明。端口声明的作用就是声明端口的类型、宽度等信息。端口的类型有输入端口、输出端口和双向端口三种，关键字分别是 input、output 和 inout。端口定义时默认 1 位宽度，即只能传播 1 位的有效信息，如果定义的端口中包含多位信息，需要指定端口的宽度，其语法结构如下：

```
端口类型　[端口宽度]　端口名;
```

端口类型即上述 input、output 和 inout。中间的“[]”区域就是端口宽度的定义，然后接端口名称，代码行的末尾添加分号“;”表示结束。例如，可以做如下声明：

```
input  [2:0]  cin;
output [0:4]  cout;
```

第一行代码声明了一个名为 cin 的输入端口，端口宽度为 3 位，按从左至右的顺序依次是 cin[2]、cin[1]和 cin[0]。第二行声明了一个名为 cout 的输出端口，端口宽度为 5 位，从

左至右依次是 cout[0]、cout[1]、cout[2]、cout[3]和 cout[4]。

端口声明中默认会把定义的端口声明为 wire 类型，即线网类型。对于端口的三种类型，除了 output 可能是寄存器类型(reg 类型)外，input 和 inout 都必须是 wire 类型。线网类型和寄存器类型的区别在于：线网类型描述的电路形式是连线，线的一端有了数据立刻会传送到另外一端，一端的数据消失则另一端数据也消失，不能够保存数值；寄存器类型描述的电路形式是寄存器，可以保存某个数值直到下次更新。

3.1.3 门级调用

端口声明之后的部分就是门级调用，门级调用的语法格式如下：

```
逻辑门类型 <实例名称(可选)> (端口连接);
```

逻辑门类型指的是常用的基本逻辑门，如程序 3.1 中的 and 和 or 就是基本逻辑门。基本逻辑门一般可以分为两大类：单输入逻辑门和多输入逻辑门。单输入逻辑门包括两种：缓冲器 buf 和非门 not，其电路符号图如图 3.2 所示。

图 3.2 单输入逻辑门

这两个逻辑门的功能如表 3.1 所示。

表 3.1 单输入逻辑门功能表

关 键 字	buf				not			
输入信号 A	0	1	x	z	0	1	x	z
输出信号 Y	0	1	x	z	1	0	x	z

下面是一个非门调用的例子：

```
not  n1(o, i);
```

逻辑门类型 not 就表示在本模块内调用了一个非门，实例名称就表示在本模块内这个非门就叫作 n1。原有的非门定义的信号在调用模块中都不再继续使用，而是使用调用语句中给出的输入信号 i 和输出信号 o。这个调用过程在 Verilog HDL 中称为模块的实例化过程。

在同一个模块中实例名称不要重复。逻辑门实例名称也可以不定义，如：

```
not  (nS1,S1);
```

此代码就是调用了一个非门，该非门的输入为 S1，输出为 nS1。这里要强调一点，没有定义实例名称并不意味着该逻辑门没有名称，而是 Verilog HDL 的内建语法会在编译的过程中自动给这个逻辑门进行命名，使得每个逻辑门都会有自己独有的实例名称。

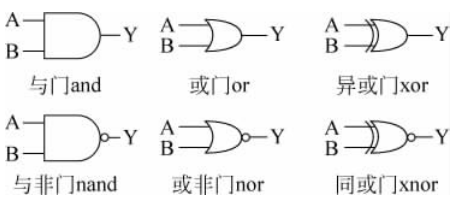

图 3.3 多输入逻辑门

多输入逻辑门中比较常见的有 6 种，分别是与门 and、与非门 nand、或门 or、或非门 nor、异或门 xor 和同或门 xnor，电路符号如图 3.3 所示，电路功能见表 3.2，表格中为输出信号 Y 的值。

多输入逻辑门可以有多个输入，但仅有一个输出，下面是多输入逻辑门调用的例子。

```
and  a1(out1,in1,in2,in3);
or   (out1,in1,in2);
xor  x1(out1,in1,in2);
```

表 3.2　多输入逻辑门功能表

与门 and	B 0	B 1	B x	B z
A 0	0	0	0	0
A 1	0	1	x	x
A x	0	x	x	x
A z	0	x	x	x

与非门 nand	B 0	B 1	B x	B z
A 0	1	1	1	1
A 1	1	0	x	x
A x	1	x	x	x
A z	1	x	x	x

异或门 xor	B 0	B 1	B x	B z
A 0	0	1	x	x
A 1	1	0	x	x
A x	x	x	x	x
A z	x	x	x	x

或门 or	B 0	B 1	B x	B z
A 0	0	1	x	x
A 1	1	1	1	1
A x	x	1	x	x
A z	x	1	x	x

或非门 nor	B 0	B 1	B x	B z
A 0	1	0	x	x
A 1	0	0	0	0
A x	x	0	x	x
A z	x	0	x	x

同或门 xnor	B 0	B 1	B x	B z
A 0	1	0	x	x
A 1	0	1	x	x
A x	x	x	x	x
A z	x	x	x	x

还有一类比较特殊的门，是带有控制信号的，包括 bufif1、bufif0、notif1 和 notif0，在原有的 buf 和 not 门上增加了一个控制信号，当控制信号生效时，输出有效数据，当控制信号不生效时，输出数据变为高阻态，所以也称为三态门，电路符号如图 3.4 所示。

这 4 个带控制信号的逻辑门功能如表 3.3 所示，表中的“0/z”和“1/z”表示根据输入信号和控制信号的强度值不同可以得到不同的输出值。

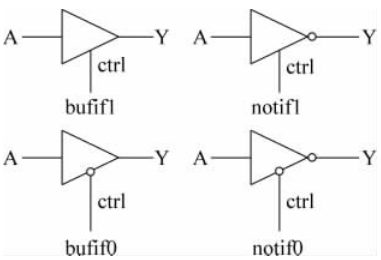

图 3.4　三态门

表 3.3　三态门功能表

三态门 bufif1	ctrl 0	ctrl 1	ctrl x	ctrl z
A 0	z	0	0/z	0/z
A 1	z	1	1/z	1/z
A x	z	x	x	x
A z	z	x	x	x

三态门 bufif0	ctrl 0	ctrl 1	ctrl x	ctrl z
A 0	0	z	0/z	0/z
A 1	1	z	1/z	1/z
A x	x	z	x	x
A z	x	z	x	x

三态门 notif1	ctrl 0	ctrl 1	ctrl x	ctrl z
A 0	z	1	1/z	1/z
A 1	z	0	0/z	0/z
A x	z	x	x	x
A z	z	x	x	x

三态门 notif0	ctrl 0	ctrl 1	ctrl x	ctrl z
A 0	1	z	1/z	1/z
A 1	0	z	0/z	0/z
A x	x	z	x	x
A z	x	z	x	x

调用三态门的语法如下：

```
三态门类型  实例名称 (输出信号,输入信号,控制信号);
```

图 3.4 中的 4 个三态门可以写成如下代码：

```
bufif1  b1(Y,A,ctrl);
bufif0  b2(Y,A,ctrl);
bufif1  n1(Y,A,ctrl);
bufif0  n2(Y,A,ctrl);
```

如果在一个模块中需要调用多个同种逻辑门，也可以使用如下的语法：

```
nand  n[2:0]  (Y,A,B);
```

该行语句等同于下面 3 条语句：

```
nand  n2(Y2,A2,B2);
nand  n1(Y1,A1,B1);
nand  DWn0(Y0,A0,B0);
```

这种语法称为实例数组，在使用实例数组的时候，实例名称必须定义。

3.1.4 模块的实例化

前面介绍了一些基本逻辑门的调用，这些被调用的逻辑门也属于模块，只不过在 Verilog HDL 的内建语法中已经定义好了，设计过程中直接拿来使用即可。Verilog HDL 的语法将模块内调用其他模块来完成设计的过程统称为模块的实例化。模块实例化语法结构如下：

```
模块名称  实例名称(端口连接);
```

模块名称即设计者已经定义好的其他模块的模块名，实例名称是在本模块内定义的新名称。端口连接是在当前模块中把实例化的模块所包含的端口进行连接，有两种连接方式：按顺序连接和按名称连接。

程序 3.2　按顺序连接实例化端口

```
module Test;
      reg a,b,c;        //定义为 reg 是因为它们要连接实例化模块的输入端
      wire y;           //定义为 wire 是因为它们要连接实例化模块的输出端
      …
      logic_gates mylogic_gates(y,a,b,c);
endmodule

module logic_gates(oY,iA,iB,iC);
       output oY;
       input iA,iB,iC;
       …
endmodule
```

按顺序连接要求连接到实例的信号必须与模块声明时目标端口在端口列表中的位置保持一致。另外，把实例化模块的输入端口所连接的信号定义为 reg，把实例化模块的输出端口所连接的信号定义为 wire，若实例化模块中有双向端口，所连接的信号也要定义为 wire，这是必须遵守的语法要求。观察程序 3.2 中的代码。此代码共有两个模块，第二个模块就是程序 3.1 中定义的门电路 logic_gates，这里只给出了模块定义和端口声明部分。第一个模块是一个测试模块 Test，在这个模块中调用了模块 logic_gates，实例化时重新命名为 mylogic_gates，连接顺序按照下面 logic_gates 模块中定义的接口顺序依次连接。可以看出，连接的顺序与 logic_gates 中的端口声明部分无关，仅考虑模块定义中的端口列表顺序，即：

```
module logic_gates(oY,iA,iB,iC);
```

按此顺序完成模块的实例化后，在顶层的 Test 模块中，仅能看到 logic_gates 这个模块和其外部的 abc 等连接信号，而内部结构就看不到了，需要查看 logic_gates 实例的原定义模块才能看到它的内部结构。当模块的端口比较多的时候，端口的先后次序就容易混淆，按顺序连接方式就容易发生错误，此时就可以使用按名称连接的方式。按名称连接方式的端口连接语法如下：

```
.原模块中端口名称(新模块中连接信号名称)
```

在程序 3.3 中特地把端口顺序调整得与原 logic_gates 中端口列表的顺序不同。在实际设计过程中，端口连接线最好和原有模块的端口名称一致，或者名称具有实际意义，以增加代码的可读性和可维护性。

程序 3.3　按名称连接实例化端口

```
module Test;
   reg a,b,c;
   wire y;
   …
   logic_gates my logic_gates (.oY(y),iB(b),iC(c),.iA(a));    //按名称连接
endmodule
```

如果在模块实例化的过程中，有些端口没有使用到，不需要进行连接，可以直接悬空。对于按顺序连接方式，可以在不需要连接的端口位置直接留一个空格，以逗号来表示这个端口在原模块中的存在。对于按名称连接方式，没有出现的端口名称就直接被认为是没有连接的端口。

两种端口连接方式在语法上都是可行的，通常按顺序连接只使用在门级建模和规模比较小的代码中，如简单的实验、课程的作业、自己编写研究的小段代码等。按名称连接可以使用在所有的代码中，而且在实际设计中使用的端口名称都是具有实际意义的，使用按名称连接方式可以方便代码的调试，增加代码的可读性，所以在正式的设计代码中大都是使用按名称连接方式的。

3.1.5 内部连线声明

内部连线是在模块实例化过程中，在被实例化的各个模块之间连接输入和输出信号的数据连线，即前面提到的 wire 类型，其定义语法如下：

```
wire [线宽] 线名称;
```

在 Verilog HDL 代码的编译过程中，凡是在模块实例化中没有定义过的端口连接信号均被默认为 1 位的 wire 类型。设计中都要把这些信号显式地声明出来，避免出现位宽不匹配的现象。在程序 3.4 中已加上声明，使程序 3.1 的实例变得完整。

程序 3.4 完整的门级建模实例代码

```
module logic_gates(oY,iA,iB,iC);
   output oY;
   input iA,iB,iC;
   wire and1,and2;        //连接线

   and (and1,iA,iB);
   and (and2,iA,iC);
   or (oY,and1,and2);
endmodule
```

3.1.6 层次化设计

掌握了基本的 Verilog HDL 门级语法后，就可以从整体上了解数字电路设计的自顶向下的流程了。如图 3.5 所示，在电路设计过程中，设计者先完成一个整体设计规划，然后把这个设计拆分为几个子模块，这些子模块都具有某种功能，完成了这些功能子模块就可以组建起整个设计。这些子模块内部还可以继续划分，也包含一些功能子模块……以此类推，直到最后得到了底层的子模块为止，然后依次完成这些子模块的建模，最后把这些模块使用 Verilog HDL 的实例化语句依次组建成整体设计。

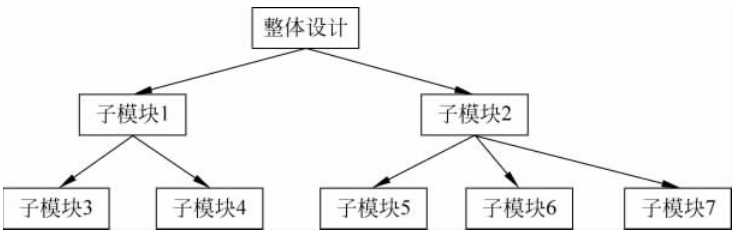

图 3.5 自顶向下设计流程

以如图 3.6 所示的在 7 段数码管上显示 4 位十六进制数的电路为例，模块 x7seg 的设计为顶层模块设计，而 4 位 4 选 1 多路选择器模块 MUX44、7 段数码管显示驱动模块 hex7seg、7 段数码管数位选择器模块 ancode、时钟分频计数器模块 ctr2bit 的设计则分别是底层模块的设计，顶层模块通过实例化调用底层模块，实现在 4 个 7 段显示管上显示 4 位十六进制数的功能。

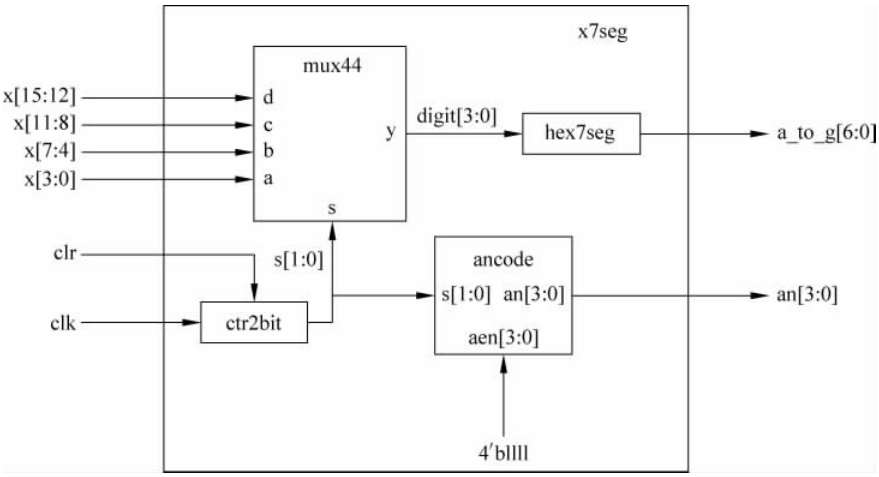

图 3.6　在 7 段数码管上显示 4 位十六进制数的电路

3.2　Verilog HDL 数据流级描述

对于规模比较小的电路采用 3.1 节介绍的门级描述方法，设计者能够直观地进行电路逻辑功能建模。当电路规模变大时，如果仅使用门级描述依次完成所有逻辑门的实例化，建模工作就变得非常烦琐而且容易出错。这就要求设计者能够从更高的抽象层次对硬件电路进行描述建模。数据流级描述便是抽象层次描述的一种。它从数据流动的角度来描述整个电路，所以大多数情况下它依然离不开基本的电路结构图或逻辑表达式。但是数据流语句的描述重点是数据如何在电路中“流动”，即数据的传输和变化情况，所以体现在描述语句中，重点是在整个电路从输入到输出的过程中，输入信号经过哪些处理或者运算，最终才能得到最后的输出信号。而这些数据的处理过程，就是通过等式右侧的由操作符和操作数组成的运算表达式来获得的。如图 3.1 所示的电路图还可以采用数据流级建模，代码如程序 3.5 所示。

程序 3.5　数据流级建模实例

```
module logic_gates(oY,iA,iB,iC);
   output oY;
   input iA,iB,iC;

   assign oY = (iA&iB)|(iA&iC)
endmodule
```

3.2.1　assign 语句

连续赋值语句(assign 语句)是 Verilog HDL 数据流建模的基本语句，用于对线网进行赋值。它等价于门级描述，然而是从更高的抽象角度来对电路进行描述。连续赋值语句必须以关键词 assign 开始，其语法如下：

```
assign [drive_strength] [delay] net_value = expression
```

注意，上面语法中的[drive_strength]是可选项，其默认值为 strong1 和 strong0。[delay]也是可选项，用于指定赋值的延迟。expression 由操作符和操作数组成。程序 3.6 给出了连续赋值语句的样例。

程序 3.6　连续赋值样例

```
//连续赋值语句。out 是线网，i1 和 i2 也是线网
assign out = i1&i2;
//向量线网的连续赋值语句。addr 是 16 位的向量线网，addr1 和 addr2 是 16 位的向量寄存器
assign addr[15:0] = addr1[15:0] ^ addr2[15:0];
//拼接操作。赋值操作符左侧是标量线网和向量线网的拼接
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

连续赋值语句具有以下特点。

(1) 连续赋值语句等号左边的 net_value 必须是一个标量或向量线网，或者是标量或向量线网的拼接，而不能是向量或向量寄存器。

(2) 连续赋值语句总是处于激活状态。只要任意一个操作数发生变化，表达式就会被立即重新计算，并且将结果赋给等号左边的线网。

(3) 等号右边表达式 expression 的操作数可以是标量或向量的线网或寄存器，也可以是函数调用。

(4) 赋值延迟用于控制对线网赋予新值的时间，根据仿真时间单位进行说明。赋值延迟类似于门延迟，对于描述实际电路的时序是非常有用的。

隐式连续赋值：除了首先声明然后对其进行连续赋值以外，Verilog 还提供了另一种对线网赋值的简便方法，即在线网声明的同时对其进行赋值。由于线网只能被声明一次，因此对线网的隐式声明赋值只能有一次。下面的例子中对隐式声明赋值和普通的连续赋值进行了比较。

```
//普通的连续赋值
wire out;
assign out = in1 & in2;
//使用隐式连续赋值实现与上面两条语句同样的功能
wire out = in1 & in2;
```

隐式线网声明：如果一个信号名被用在连续赋值语句的左侧，那么 Verilog 编译器认为该信号是一个隐式声明的线网。如果线网被连接到模块的端口上，则 Verilog 编译器认为隐式声明线网的宽度等于模块端口的宽度。举例如下。

```
//连续赋值，out 为线网类型
wire i1, i2;
assign out = i1 & i2;
//注意 out 并未声明为线网，但 Verilog 仿真器会推断出 out 是一个隐式声明的线网
```

连续赋值语句中的延迟用于控制任一操作数发生变化到语句左值被赋予新值之间的时间间隔。指定赋值延迟的方法有三种：普通赋值延迟、隐式赋值延迟和线网声明延迟。

1. 普通赋值延迟

即在连续赋值语句中说明延迟值，延迟值位于关键字 assign 的后面。例如：

```
assign #10 out = in1 & in2;      //连续赋值语句中的延迟
```

在上面的例子中，# 表示时间延迟，如果 in1 和 in2 中的任意一个发生变化，那么在计算表达式 in1&in2 的新值并将新值赋给语句左值之前，会产生 10 个时间单位的延迟。如果在此 10 个时间单位期间，即左值获得新值之前，in1 或 in2 的值再次发生变化，那么在计算表达式的新值时会取 in1 或 in2 的当前值。这种性质被称为惯性延迟。也就是说，脉冲宽度小于赋值延迟的输入变化不会对输出产生影响。

2. 隐式连续赋值延迟

使用隐式连续赋值语句来说明对线网的赋值以及赋值延迟。隐式连续赋值等效于声明一个线网并且对其进行连续赋值。举例如下。

```
wire #10 out = in1 & in2;        //隐式连续赋值延迟
```

3. 线网声明延迟

Verilog 允许在声明线网的时候指定一个延迟，这样对该线网的任何赋值都会被推迟指定的时间。举例如下。

```
wire #10 out;                    //线网声明延迟
assign out = in1 & in2;
```

上面的线网延迟声明和赋值语句和下面两条语句是等效的。

```
wire out;
assign #10 out = in1 & in2;
```

3.2.2　操作符

Verilog HDL 的操作符有很多种，按其功能大致可以分为逻辑操作符、位操作符、算术操作符、关系操作符、移位操作符、拼接操作符、缩减操作符、条件操作符。如果按其处理操作数的个数可以分为单目操作符、双目操作符和三目操作符。相关操作符的介绍见附录 A。操作符之间有优先级的概念，如果不注意使用会产生错误，操作符的优先级高低可见表 3.4。

表 3.4　操作符优先级

操　作	操作符号	优先级别
按位取反，逻辑非	！　～	最高 ↓ 最低
乘、除、取模	＊　／　％	
加、减	＋　－	
移位	＜＜　＞＞	
关系	＜　＜＝　＞　＞＝	
等价	＝＝　！＝　＝＝＝　！＝＝	
缩减、按位	&　～&	
	^　^～	
	|　～|	
逻辑	&&	
	||	
条件	?：	

3.2.3 操作数

在操作数中首先要介绍的是数字，数字并不是数据类型中的某一种，但可以使用数字对数据类型进行赋值。数字的基本格式如下：

```
<位宽>'<进制><数值>
```

Verilog HDL 中支持 4 种进制形式：二进制、八进制、十进制和十六进制，分别用 b、o、d、h 来表示(不区分大小写)。数值部分指在相应进制下的数值。位宽表示了一个数字包含几位信息，指明了数字的精确位数，这个位数是以该数字转化为二进制后所具有的宽度来表示的，举例如下。

```
2'b01
4'd11
```

当位宽大于数值宽度时，如果数值部分是确切的数值，缺少的部分采用补零原则；当位宽小于数值宽度时，采用低位对其直接截取的方式，保留位宽中定义的宽度。在数值部分出现不定态 x 和高阻态 z 时，x 和 z 也会根据进制的不同被扩展为不同的宽度。例如，在八进制中一个 x 相当于三位的二进制数 xxx，在十六进制中就变为四位的二进制数 xxxx。特别的，在数值的首位为 x 和 z 时，如果出现了位宽大于数值宽度的情况，则缺少的位分别按 x 或 z 补齐。

如果格式中缺少了位宽或进制，则会有其他等效方法。如果数字中只包含进制和数值部分，则位宽采用默认宽度，主要取决于所使用机器的系统宽度和仿真器所支持的宽度，一般为 32 位。如果仅有数值部分，则在默认宽度的基础上默认进制为十进制。数字也可以表示负数，在数字前直接添加负号即可，此时表示的是当前负数的二进制补码，负号不可以放在数值部分，举例如下。

```
-4'd6
```

操作数有很多数据类型，本节对几种在设计和仿真中常用的数据类型进行介绍。

1. 线网

线网(net)表示硬件单元之间的连接。就像在真实的电路中一样，线网由其连接器件的输出端连续驱动，包括 wire, wand, wor, tri, triand, trior 以及 trireg 等，其中，wire 类型的线网声明最为常用。wire 这个术语和 net 经常互换使用。如果没有显式地说明为向量，则默认线网的位宽为 1。线网的默认值为 z(trireg 类型的线网例外，其默认值为 x)。线网的值由其驱动源确定，如果没有驱动源，则线网的值为 z。举例如下。

```
wire a;                 //声明 a 是 wire(连线)类型
wire d = 1'b0;          //连线 d 在声明时,d 被赋值为逻辑值 0
```

2. 寄存器

用来表示存储元件，它保持原有的数值，直到被改写。注意，不要将这里的寄存器与实际电路中由边沿触发的触发器构成的硬件寄存器混淆。在 Verilog 中，术语 register 仅仅意味着一个保存数值的变量。与线网不同，寄存器不需要驱动源，而且也不像硬件寄存器那样需要时钟信号。在仿真过程中的任意时刻，寄存器的值都可以通过赋值来改变。寄存器数

据类型一般通过使用关键字 reg 来声明，默认值为 x。举例如下。

```
reg reset;                          //声明能保持数值的变量 reset
reset = 1'b1;                       //把 reset 初始化为 1,使数字电路复位
#100 reset = 1'b0;                  //经过 100 个时间单位后,reset 置逻辑 0
```

寄存器也可以声明为带符号(Signed)类型的变量，这样的寄存器就可以用于带符号的算术运算。举例如下。

```
reg signed [63:0] m;                //64 位带符号的值
```

3. 向量

线网和寄存器类型的数据均可以声明为向量(位宽大于 1)。如果在声明中没有指定位宽，则默认为标量(1 位)。举例如下。

```
wire a;                             //标量线网变量,默认宽度
wire [7:0] bus;                     //8 位的总线
reg c;                              //标量寄存器,默认宽度
reg [0:40] virtual_addr;            //向量寄存器,41 位宽的虚拟地址
```

向量通过[high：low]或[low：high]进行说明，方括号中左边的数总是代表向量的最高有效位。在上面的例子中，向量 virtual_addr 的最高有效位是它的第 0 位。

向量域选择：对于上面例子中声明的向量，可以指定它的某一位或若干个相邻位。举例如下。

```
bus [7]                             //向量 bus 的第 7 位
bus [2:0]                           //向量 bus 的最低 3 位
```

上面例子如果写成 bus[0:2]是非法的，因为高位应该写在范围说明的左侧。

可变的向量域选择：除了用常量指定向量域以外，Verilog HDL 还允许指定可变的向量域选择，设计者可以通过 for 循环来动态地选取向量的各个域。下面是动态域选择的两个专用操作符。

```
[<starting_bit>+:width]             //从起始位开始递增,位宽为 width
[<starting_bit>-:width]             //从起始位开始递减,位宽为 width
```

起始位可以是一个变量，但是位宽必须是一个常量。程序 3.7 说明了可变的向量域选择的使用方法。

程序 3.7　可变向量域选择方法

```
reg [255:0] data1;                  //data1[255]是最高有效位
reg [0:255] data2;                  //data2[0]是最高有效位
reg [7:0] byte;                     //用变量选择向量的一部分
byte = data1[31-:8];                //从第 31 位算起,宽度为 8 位,相当于 data1[31:24]
byte = data1[24+:8];                //从第 24 位算起,宽度为 8 位,相当于 data1[31:24]
byte = data2[31-:8];                //从第 31 位算起,宽度为 8 位,相当于 data2[24:31]
byte = data2[24+:8];                //从第 24 位算起,宽度为 8 位,相当于 data2[24:31]
```

```
//起始位可以是变量:但宽度必须是常数.因此可以通过可变域选择
//用循环语句选取一个很长的向量的所有位
for (j = 0; j <= 31; j = j + 1)
byte = data1[(j * 8) + : 8];          //次序是[7:0], [15:8]… [255:248]
```

4. 整数

整数是一种通用的寄存器数据类型,用于对数量进行操作,使用关键字 integer 进行声明。虽然可以使用 reg 类型的寄存器变量作为通用的变量,但声明一个整数类型的变量来完成计数等功能显然更为方便。整数的默认位宽为主机的字的位数。与具体实现有关,但最小应为 32 位。声明为 reg 类型的寄存器变量为无符号数,而整数类型的变量则为有符号数。举例如下。

```
integer counter;                      //一般用途的变量,作为计数器
counter = -1;                         //把 -1 存储到计数器中
```

5. 实数

实常量和实数寄存器数据类型使用关键字 real 来声明,可以用十进制或科学记数法(例如 3e6 代表 3 000 000)来表示。实数声明不能带有范围,其默认值为 0。如果将一个实数赋给一个整数,那么实数将会被取整为最接近的整数。举例如程序 3.8 所示。

程序 3.8 实数声明样例

```
real delta;                           //定义一个名为 delta 的实型变量
delta = 4e10;                         //delta 被赋值, 用科学计数法表示
delta = 2.13;                         //delta 被赋值为 2.13
integer i;                            //定义一个名为 i 的整型变量
i = delta;                            //i 得到值 2(2.13 取整数部分)
```

6. 时间寄存器

仿真是按照仿真时间进行的,Verilog 使用一个特殊的时间寄存器数据类型来保存仿真时间。时间变量通过使用关键字 time 来声明,其宽度与具体实现有关,最小为 64 位。通过调用系统函数 $time 可以得到当前的仿真时间。举例如下。

```
time save_sim_time;                   //定义时间类型的变量 save_sim_time
save_sim_time = $time;                //把当前的仿真时间记录下来
```

仿真时间的单位为秒(s),和真实时间的表示方法相同。但是,真实时间和仿真时间的对应关系需要由用户来定义。

7. 数组

在 Verilog 中允许声明 reg, integer, time, real, realtime 及其向量类型的数组,对数组的维数没有限制,即可以声明任意维数的数组。线网数组也可用于连接实例的端口,数组中的每个元素都可以作为一个标量或向量,以同样的方式来使用,形如<数组名>[<下标>]。对于多维数组来讲,用户需要说明其每一维的索引。举例如程序 3.9 所示。

程序 3.9 数组声明样例

```
integer count[0:7];                                  //由 8 个计数变量组成的数组
reg bool[31:0];                                      //由 32 个 1 位的布尔(boolean)寄存器变量组
                                                     //成的数组
time chk_point[1:100];                               //由 100 个时间检查变量组成的数组
reg [4:0] port_id[0:7];                              //由 8 个端口标识变量组成的数组,端口变量
                                                     //的位宽为 5
integer matrix[4:0][0:255];                          //二维的整数型数组
reg [63:0] array_4d[15:0][7:0][7:0][255:0];          //四维 64 位寄存器型数组
wire [7:0] w_array2[5:0];                            //声明 8 位向量的数组
wire w_array1 [7:0][5:0];                            //声明 1 位线型变量的二维数组
```

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,它的位宽为 n; 数组由多个元件组成,其中的每个元件的位宽为 n 或 1。程序 3.10 给出了对数组元素赋值的例子。

程序 3.10 数组元素赋值样例

```
count[5] = 0;                 //把 count 数组中的第 5 个整数型单元(32 位)复位
port_id[3] = 0;               //把 port_id 数组中的第 3 个寄存器型单元(5 位)复位
matrix[1][0] = 33559;         //把数组中第 1 行第 0 列的整数型单元(32 位)置为 33559
port_id = 0;                  //非法,企图写整个数组
matrix [1] = 0;               //非法,企图写数组的整个第 2 行
```

8. 存储器

在数字电路仿真中,人们常常需要对寄存器文件、RAM 和 ROM 建模。在 Verilog 中,使用寄存器的一维数组来表示存储器。数组的每个元素称为一个元素或一个字,由一个数组索引来指定,每个字的位宽为 1 位或多位。注意,n 个 1 位寄存器和一个 n 位寄存器是不同的。如果需要访问存储器中的一个特定的字,则可以通过将字的地址作为数组的下标来完成。

```
reg mem1bit[0:1023];                  //1KB(1 位)存储器 mem1bit
reg [7:0] membyte[0:1023];            //1KB(8 位)存储器 membyte
membyte[511] = 0;                     //将存储器 membyte 地址 511 处所存的内容清零
```

9. 参数

Verilog 允许使用关键字 Parameter 在模块内定义常数。参数代表常数,不能像变量那样赋值,但是每个模块实例的参数值可以在编译阶段被重载。通过参数重载使得用户可以对模块实例进行定制。除此之外,还可以对参数的类型和范围进行定义。举例如下。

```
parameter port_id = 5;                //定义常数 port_id 为 5
parameter cache_line_width = 256;     //定义高速缓冲器总线宽度为常数 256
parameter signed [15:0] WIDTH;        //把参数 WIDTH 规定为有正负号,宽度为 16 位
```

通过使用参数,用户可以更加灵活地对模块进行说明。用户不但可以根据参数来定义

模块，还可以方便地通过参数值重定义来改变模块的行为：通过模块实例化或使用defparam语句改变参数值。在参数定义时需要注意避免使用硬编码。Verilog中的局部参数使用关键字localparam来定义，其作用等同于参数，区别在于它的值不能改变，不能通过参数重载语句(defparam)或通过有序参数列表或命名参数赋值来直接修改。例如，状态机的状态编码是不能被修改的，为了避免被意外地更改，应当将其定义为局部参数。举例如下。

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

10. 字符串

保存在reg类型的变量中，每个字符占用8位(一个字节)，因此寄存器变量的宽度应足够大，以保证容纳全部字符。如果寄存器变量的宽度大于字符串的大小(位)，则Verilog使用0来填充左边的空余位；如果寄存器变量的宽度小于字符串的大小(位)，则Verilog截去字符串最左边的位。因此，在声明保存字符串的reg变量时，其位宽应当比字符串的位长稍大。举例如下。

```
reg [8*18:1] string_1;                    //声明变量 string_1,其宽度为 18 个字节
initial
string_1 = "Hello Verilog World";         //字符串可以存储在变量中
```

有一些特殊字符在显示字符串时具有特定的意义，例如换行符、制表符和显示参数的值。如果需要在字符串中显示这些特殊的字符，则必须加前缀转义字符。

3.3 Verilog HDL 行为级描述

在3.2节的数据流级描述中已经将硬件建模从比较底层的门级结构提升到了数据流级。但数据流级描述除了个别语句外，主要的部分还是使用操作符来描述电路的逻辑操作或者计算公式，没有实现真正意义上的功能描述。行为级描述则可以实现从抽象层次更高的级别来描述功能电路。本节内容将对行为描述的语法进行详细介绍。

3.3.1 initial 结构和 always 结构

在Verilog中有两种结构化过程语句：initial语句和always语句，它们是行为级建模的两种语句。其他所有的行为语句只能出现在这两种结构化过程语句里。Verilog中各个执行流程并发执行，而不是顺序执行的。每个initial语句和always语句代表一个独立的执行过程，每个执行过程从仿真时间0开始执行，并且这两种语句不能嵌套使用。

1. initial 结构

所有在initial语句内的语句构成了一个initial块。initial块从仿真0时刻开始执行，在整个仿真过程中只执行一次。如果一个模块中包括若干个initial块，则这些initial块从仿真0时刻开始并发执行，且每个块的执行是各自独立的。如果在块内包含多条行为语句，那么需要将这些语句组成一组，一般是使用关键字begin和end将它们组合为一个块

语句；如果块内只有一条语句，则不必使用 begin 和 end。程序 3.11 给出了使用 initial 语句的例子。三条 initial 语句在仿真 0 时刻开始并行执行。如果在某一条语句前面存在延迟 #< delay >，那么对这条 initial 语句的仿真将会停顿下来，在经过指定的延迟时间之后再继续执行。

程序 3.11　initial 语句

```
module stimulus;
reg x,y,a,b,m;
initial
    m = 1'b0;
initial
begin
    #5 a = 1'b1;
    #25 y = 1'b1;
end
initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end
initial
    #50 $finish;
endmodule
```

2. always 结构

always 语句包括的所有行为语句构成了一个 always 语句块。该 always 语句块从仿真 0 时刻开始顺序执行其中的行为语句；在最后一条执行完成之后，再次开始执行其中的第一条语句，如此循环往复，直至整个仿真结束。因此 always 语句通常用于对数字电路中一组反复执行的活动进行建模。程序 3.12 给出了用 always 语句为时钟发生器建立模型的一种方法。

程序 3.12　always 语句

```
module clock_gen (output reg clock);
initial
   clock = 1'b0;
always
   #10 clock = ~clock;     //每半个周期把 clock 信号的值翻转一次(周期 = 20)
initial
   #1000 $finish;
endmodule
```

在这个例子中，clock 信号是在 initial 语句中进行初始化的，如果将初始化放在 always 块内，那么 always 语句的每次执行都会导致 clock 被初始化。如果没有 $stop 或 $finish 语句停止仿真，那么这个时钟发生器将一直工作下去。

基于事件的控制在实际建模中使用最多，也是行为级建模的一个重要控制方式，其控制方式为“@”引导的事件列表，也称为敏感列表，如下所示。

```
always @ (敏感事件列表)
```

敏感事件列表是由设计者来指定的。在模块中，任何信号的变化都可以称为事件。一旦这些事件发生了，always 结构中的语句就会被执行。换言之，always 结构时刻观察敏感事件列表中的信号，等待敏感事件出现，然后执行本结构中的语句。如果敏感事件有多个，可以使用 or 或“，”来隔开，这些事件是或的关系，只需要满足一个就会触发并执行 always 结构，如程序 3.13 所示。

程序 3.13　基于事件控制 always 语句

```
always @ (a,b)
begin
    e = c&d;
    f = c|d;
end
```

这段代码中都是以信号的名称作为敏感事件，表示的是对信号的电平敏感，即信号只要发生了变化，就要执行 always 结构，这个变化指的是仿真器可以识别的任意变化，例如，从 0 变到 1 或从 1 变到 0。使用这种控制方式可以设计对电平信号敏感的电路，所有的组合逻辑电路采用的都是这种控制方式。敏感事件列表中的事件可以用“*”代替，这个“*”表示的是该 always 结构中所有的输入信号。时序电路采用的敏感列表一般是边沿敏感的，信号的边沿用 posedge(上升沿)和 negedge(下降沿)来表示，但边沿敏感不能使用“*”来省略。

3.3.2　顺序块和并行块

块语句包括两种类型：顺序块和并行块。

1. 顺序块

关键字 begin 和 end 用于将多条语句组成顺序块。顺序块具有以下特点。

(1) 顺序块中的语句是一条接一条按顺序执行的；只有前面的语句执行完成之后才能执行后面的语句(带有内嵌延迟控制的非阻塞赋值语句除外)。

(2) 如果语句包括延迟或事件控制，那么延迟总是相对于前面那条语句执行完成的仿真时间的。如程序 3.14 中的例子 1，在仿真 0 时刻，x，y，z 和 w 的最终值分别为 0，1，1 和 2。在例子 2 中，这 4 个变量的最终值也是 0，1，1 和 2。但是块语句完成时的仿真时刻为 35。

程序 3.14　顺序块

```
//例子 1
reg x,y;
```

```
reg [1:0] z,w;
initial
begin
   x = 1'b0;
   y = 1'b1;
   z = {x,y};
   w = {y,x};
end

//例子 2:带延迟的顺序块
reg x,y;
reg [1:0]z,w;
initial
begin
   x = 1'b0;                    // 在仿真时刻 0 完成
   #5 y = 1'b1;                 // 在仿真时刻 5 完成
   #10 z = {x,y};               // 在仿真时刻 15 完成
   #20 w = {y,z};               // 在仿真时刻 35 完成
end
```

2. 并行块

并行块由关键字 fork 和 join 声明。并行块具有以下特性。

(1) 并行块内的语句并发执行。

(2) 语句执行的顺序是由各自语句中的延迟或事件控制决定的。

(3) 语句中的延迟或事件控制是相对于块语句开始执行的时刻而言的。

注意,顺序块和并行块之间的根本区别在于:并行块中所有的语句同时开始执行,语句之间的先后顺序是无关紧要的。将程序 3.14 中带有延迟的顺序块转换为一个并行块,转换后代码见程序 3.15。除了所有语句在仿真 0 时刻开始执行以外,仿真结果是完全相同的。这个并行块执行结束的时间是仿真时刻 20,而不是仿真时刻 35。

程序 3.15　并行块

```
reg x, y;
reg [1:0] z, w;
initial
fork
   x = 1'b0;                    // 在仿真时刻 0 完成
   #5 y = 1'b1;                 // 在仿真时刻 5 完成
   #10 z = {x,y};               // 在仿真时刻 10 完成
   #20 w = {y,z};               // 在仿真时刻 20 完成
 join
```

并行块提供了并行执行语句的机制。可以将并行块的关键字 fork 看成是将一个执行流分成多个独立的执行流,而关键字 join 则是将多个独立的执行流合并为一个执行流。每个独立的执行流之间是并发执行的。在使用并行块时需要注意,如果两条语句在同一时刻

对同一个变量产生影响，那么将会引起隐含的竞争，这种情况是需要避免的。

3. 块语句的特点

下面讨论块语句具有的三个特点：嵌套块、命名块和命名块的禁用。

(1) 嵌套块：顺序块和并行块可以嵌套使用，使用时要注意每一个嵌套块开始的时间，如程序 3.16 所示。

程序 3.16　嵌套块

```
initial
begin
   x = 1'b0;
fork
  #5 y = 1'b1;
  #10 z = {x, y};
join
  #20 w = {y, x};
end
```

(2) 命名块：块可以具有自己的名字，称为命名块。程序 3.17 给出了命名块和命名块的层次名引用。命名块具有如下特点：

① 命名块中可以声明局部变量。

② 命名块是设计层次的一部分，命名块中声明的变量可以通过层次名引用进行访问。

③ 命名块可以被禁用，例如停止其执行。

程序 3.17　命名块

```
module top;
initial
begin: block1              //名字为 block1 的顺序命名块
integer i;                 //整型变量 i 是 block1 命名块的静态局部变量
                           //可以通过层次名 top.block1.i 被其他模块访问
…
end

initial
fork: block2               //名字为 block2 的并行命名块
reg i;                     //寄存器变量 i 是 block2 命名块的静态局部变量
                           //可以通过层次名 top.block2.i 被其他模块访问
…
join
```

(3) 命名块的禁用：Verilog 通过关键字 disable 提供了一种终止命名块执行的方法。disable 可以用来从循环中退出、处理错误条件以及根据控制信号来控制某些代码段是否被执行。对块语句的禁用导致紧接在块后面的那条语句被执行。对于 C 程序员来说，这一点非常类似于使用 break 退出循环。两者的区别在于 break 只能退出当前所在的循环，而使用 disable 则可以禁用设计中的任意一个命名块，如程序 3.18 所示。

程序 3.18 命名块的禁用

```
//从标志寄存器的最低有效位开始查找第一个值为 1 的位
reg [15:0] flag;
integer i;                    //用于计数的整数

initial
begin
flag = 16'b 0010_0000_0000_0000;
i = 0;
  begin: block1               //while 循环声明中的主模块是命名块 block1
  while (i<16)
     begin
     if(flag[i])
        begin
        $display("Encountered a TRUE bit at element number %d", i);
        disable block1;       //在标志寄存器中找到了值为真(1)的位,禁用 block1
        end
        i = i + 1;
        end
   end
end
```

3.3.3 if 语句

条件语句用于根据某个条件来确定是否执行其后的语句,关键字 if 和 else 用于表示条件语句。Verilog 语言共有三种类型的条件语句,条件语句的用法见程序 3.19。if 条件成立或不成立时,执行的语句可以是一条语句,也可以是一组语句。如果是一组语句,则通常使用 begin 和 end 关键字将它们组成一个块语句。

程序 3.19 条件语句举例

```
//第一类条件语句:没有 else 语句
if(!lock) buffer = data;
if(enable) out = in;
//第二类条件语句:有一个 else 语句
if(number_queued < MAX_Q_DEPTH)
  begin
    data_queue = data;
    number_queued = number_queued + 1;
  end
else
  $display("Queue Full. Try again");
//第三类条件语句:嵌套的 if - else - if 语句
if(alu_control == 0)
  y = x + z;
```

```
else if (alu_control == 1)
  y = x - z;
else if (alu_control == 2)
  y = x * z;
else
  $display("Invalid ALU control signal ");
```

3.3.4 case 语句

case 语句使用关键字 case、default 和 endcase 来表示。

```
case (expression)
 alternative1: statement1;
 alternative2: statement2;
 ⋮
 default: default_statement;
endcase
```

case 语句中的每一条分支语句都可以是一条语句或一组语句。多条语句需要使用关键字 begin 和 end 组合为一个块语句。在执行时，首先计算条件表达式的值，然后按顺序将它和各个候选项进行比较。如果等于第一个候选项，则执行对应的语句 statement1；如果和全部候选项都不相等，则执行 default_statement 语句。注意，default_statement 语句是可选的，而且在一条 case 语句中不允许有多条 default_statement。另外，case 语句可以嵌套使用。程序 3.20 给出了 case 语句的一个例子。

程序 3.20　case 语句举例

```
reg [1:0] alu_control;
…
Case (alu_control)            //根据不同的 alu_control 信号,执行不同的语句
   2'd0 : y = x + z;
   2'd1 : y = x - z;
   2'd2 : y = x * z;
default : $display("Invalid ALU control signal");
endcase
```

case 语句逐位比较表达式的值和候选项的值，每一位的值可能是 0，1，x 或 z。如果两者的位宽不相等，则使用 0 填补空缺位来使两者的位宽相等。若选择信号中有不确定值 x，则输出为 x；若选择信号中没有不确定值 x，但有高阻值 z，则输出为 z。程序 3.21 给出了带 x 和 z 的 case 语句举例。在程序中，把引起同一输出块执行的多个选择信号组合，如2′bz0，2′bz1，2′bzz，2′b0z 和 2′b1z 放在同一个语句块的候选项中，使用逗号将其分开。

程序 3.21 带 x 和 z 的 case 语句举例

```
module demultiplexer1_to_4 (out0, out1, out2, out3, in, s1, s0);
output out0, out1, out2, 0ut3;
reg out0, out1, out2, out3;
input in;
input s1,s0;

always @(s1 or s0 or in)
begin
case ({s1,s0})
   2'b00 :
   begin out0 = in; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz; end
   2'b01 :
   begin out0 = 1'bz; out1 = in; out2 = 1'bz; out3 = 1'bz; end
   2'b10 :
   begin out0 = 1'bz; out1 = 1'bz; out2 = in; out3 = 1'bz; end
   2'b11 :
   begin out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = in; end
   2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx:
   begin out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx; end
   2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z:
   begin out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz; end
default : $display("Unspecified control signals");
endcase
end
endmodule
```

除了上面讲述的 case 语句之外，case 语句还有两个变形，分别使用关键字 casex 和 casez 来表示。casex 语句将条件表达式或候选项表达式中的 x 作为无关值。casez 语句将条件表达式或候选项表达式中的 z 作为无关值，所有值为 z 的位也可以用“?”来代表；casex 和 casez 的使用可以让我们在 case 表达式中只对非 x 或非 z 的位置进行比较。程序 3.22 给出了 casex 的用法举例。casez 的使用与 casex 的使用类似。

程序 3.22 casex 的用法举例

```
reg [3:0] encoding;
integer state;

casex (encoding)       //逻辑值 x 表示无关位
   4'b1xxx : next_state = 3;
   4'bx1xx : next_state = 2;
   4'bxx1x : next_state = 1;
   4'bxxx1 : next_state = 0;
   default : next_state = 0;
endcase
```

3.3.5 循环语句

Verilog 语言中有 4 种类型的循环语句：while，for，repeat 和 forever。这些循环语句的语法与 C 语言中的循环语句类似。循环语句只能在 always 或 initial 块中使用，循环语句

可以包含延迟表达式。

1. while 循环

while 循环使用关键字 while 来表示。while 循环执行的中止条件是 while 表达式的值为假。如果遇到 while 语句时 while 表达式的值已经为假,那么循环语句一次也不执行。如果循环中有多条语句,则必须将它们组合成为 begin 和 end 块。程序 3.23 给出了 while 循环的例子。

程序 3.23 while 循环

```
// 说明 1:计数器变量 count 从 0 到 127,并显示 count 值,到 128 时停止计数
   integer count;

   initial
   begin
      count = 0;
      while(count < 128)          // 执行循环直到计数器为 127,当计数器为 128 时退出
      begin
         $display("Count = %d", count);
         count = count + 1;
      end
  end
```

2. for 循环

for 循环使用关键字 for 来表示,它由以下三个部分组成。

(1) 初始条件;

(2) 检查终止条件是否为真;

(3) 改变控制变量的过程赋值语句。

程序 3.23 中用 while 循环语句描述的计数器也可以用 for 循环语句来描述,见程序 3.24。由于初始条件和完成自加操作的过程赋值语句都包括在 for 循环中,无须另外说明,因此 for 循环的写法较 while 循环更为紧凑。但是要注意 while 循环比 for 循环更为通用,并不是在所有情况下都能使用 for 循环来代替 while 循环。for 循环一般用于具有固定开始和结束条件的循环。如果只有一个执行循环的条件,最好还是使用 while 循环。

程序 3.24 for 循环

```
// 数组元素的初始化
integer state [0:31];
integer i;
initial
begin
for (i = 0; i < 32; i = i + 2)      // 把所有偶元素初始化为 0
    state[i] = 0;
for (i = 1; i < 32; i = i + 2)      // 把所有奇元素初始化为 1
    state[i] = 1;
end
```

3. repeat 循环

用关键字 repeat 来表示。repeat 循环的功能是执行固定次数的循环，它不能像 while 循环那样根据一个逻辑表达式来确定循环是否继续进行。repeat 循环的次数必须是一个常量、一个变量或者一个信号。如果循环重复次数是变量或者信号，循环次数是循环开始执行时变量或者信号的值，而不是循环执行期间的值。

程序 3.25 给出了如何使用 repeat 循环对数据缓冲区建模，这个数据缓冲区的功能是在收到开始信号之后第 8 个时钟上升沿处锁存输入数据。

程序 3.25　repeat 循环

```
module data_buffer (data_en, data, clock);
parameter cycles = 8;
input data_en;
input [15:0] data;
input clock;
reg [15:0] buffer [0:7];
integer i;

always @(posedge clock)
begin
if (data_en)                    //data_en 信号为真
begin
   i = 0;
   repeat (cycles)              //在接下来的 8 个时钟周期的正跳变沿存储数据
      begin
        @(posedge clock) buffer[i] = data;
        i = i + 1;
      end
  end
end
endmodule
```

4. forever 循环

关键字 forever 用来表示永久循环。在永久循环中不包含任何条件表达式，只执行无限的循环，直到遇到系统任务 $finish 为止。forever 循环等价于条件表达式永远为真的 while 循环，例如 while(1)。如果需要从 forever 循环中退出，可以使用 disable 语句。

通常情况下，forever 循环是和时序控制结构结合使用的：如果没有时序控制结构，那么仿真器将无限次地执行这条语句，并且仿真时间不再向前推进，使得其余部分的代码无法执行。程序 3.26 给出了 forever 循环的使用举例。

程序 3.26　forever 循环

```
//例 1:时钟发生器
reg clock;
initial
begin
```

```
    clock = 1'b0;
    forever #10 clock = ~clock;          //时钟周期为 20 个单位时间
end

//例 2:在每个时钟正跳变沿处使两个寄存器的值一致
reg clock;
reg x, y;

initial
    forever @(posedge clock) x = y;
```

3.3.6 过程赋值语句

过程赋值语句的更新对象是寄存器、整数、实数或时间变量。这些类型的变量在被赋值后,其值将保持不变,直到被其他过程赋值语句赋予新值。这与连续赋值语句是不同的。连续赋值语句总是处于活动状态,任意一个操作数的变化都会导致表达式的重新计算以及重新赋值,但过程赋值语句只有在执行到的时候才会起作用。Verilog 包括两种类型的过程赋值语句:阻塞赋值语句和非阻塞赋值语句。

1. 阻塞赋值语句

顺序块语句中的阻塞赋值语句按顺序执行,它不会阻塞其后并行块中语句的执行。阻塞赋值语句使用"="作为赋值符。由于阻塞赋值语句是按顺序执行的,因此如果在一个 begin-end 块中使用了阻塞赋值语句,那么这个块语句表现的是串行行为。在程序 3.27 中,只有在语句 x = 0 执行完成之后,才会执行 y = 1,而语句 count = count + 1 按顺序在最后执行。begin-end 块中各条语句执行的仿真时间如下。

(1) x = 0 到 reg_b = reg_a 之间的语句在仿真 0 时刻执行;

(2) 语句 reg_a[2] = 0 在仿真时刻 15 执行;

(3) 语句 reg_b[15 : 13]={x, y, z}在仿真时刻 25 执行;

(4) 语句 count = count + 1 在仿真时刻 25 执行。

程序 3.27 阻塞赋值语句

```
reg x,y,z;
reg [15:0] reg_a,reg_b;
integer count;

//所有行为语句必须放在 initial 或 always 块内部
initial
begin
    x = 0; y = 1; z = 1;                 //标量赋值
    count = 0;                           //整型变量赋值
    reg_a = 16'b0; reg_b = reg_a;        //向量的初始化
    #15 reg_a[2] = 1'b1;                 //带延迟的位选赋值
    #10 reg_b[15:13] = {x,y,z}           //把拼接操作的结果赋值给向量的部分位(域)
    count = count + 1;                   //给整型变量赋值(递增)
end
```

注意，在对寄存器类型变量进行过程赋值时，如果赋值符两侧的位宽不相等，则采用以下原则。

（1）如果右侧表达式的位宽较宽，则将保留从最低位开始的右侧值，把超过左侧位宽的高位丢弃；

（2）如果左侧位宽大于右侧位宽，则不足的高位补 0。

2. 非阻塞赋值语句

非阻塞赋值语句允许赋值调度，但它不会阻塞位于同一个顺序块中其后语句的执行。非阻塞赋值使用“<=”作为赋值符。读者会注意到，它与“小于等于”关系操作符是同一个符号，但在表达式中它被解释为关系操作符，而在非阻塞赋值的环境下被解释成非阻塞赋值。为了说明非阻塞赋值的意义以及与阻塞赋值的区别，让我们来考虑将程序 3.27 中的部分阻塞赋值改为非阻塞赋值后的结果，程序 3.28 给出了修改后的语句。

程序 3.28　非阻塞赋值语句

```
reg x,y,z;
reg [15:0] reg_a,reg_b;
integer count;

initial
begin
    x = 0; y = 1; z = 1;                    //标量赋值
    count = 0;                              //整型变量赋值
    reg_a = 16'b0; reg_b = reg_a;           //向量的初始化
    reg_a[2] <= #15 1'b1;                   //带延迟的位选赋值
    reg_b[15:13] <= #10 {x,y,z};            //把拼接操作的结果赋值给向量的部分位(域)
    count <= count + 1;                     //给整型变量赋值(递增)
end
```

在这个例子中，从 x = 0 到 reg_b = reg_a 之间的语句是在仿真 0 时刻顺序执行的，之后的三条非阻塞赋值语句在 reg_b = reg_a 执行完成后并发执行。

（1）reg_a[2] = 0 被调度到 15 个时间单位之后执行，即仿真时刻为 15；

（2）reg_b[15:13] = {x, y, z}被调度到 10 个时间单位之后执行，即仿真时刻为 10；

（3）count = count + 1 被调度到无任何延迟执行，即仿真时刻为 0。

从上面的分析中可以看到，仿真器将非阻塞赋值调度到相应的仿真时刻，然后继续执行后面的语句，而不是停下来等待赋值的完成。一般情况下，非阻塞赋值是在当前仿真时刻的最后一个时间步，即阻塞赋值完成之后才执行的。在上面的例子中，我们把阻塞和非阻塞赋值语句混合在一起使用，目的是想更清楚地比较和说明它们的行为。需要提醒读者注意的是，不要在同一个 always 块中混合使用阻塞和非阻塞赋值语句。非阻塞赋值可以被用来为常见的硬件电路行为建立模型，例如当某一事件发生后，多个数据并发传输的行为。

3.3.7　任务与函数

在程序设计过程中，设计者经常需要在程序的许多不同地方实现相同的功能，此时可以把这些公共的部分提取出来，写成子程序供重复使用，在需要的位置直接调用子程序。

Verilog HDL 语法中也提供了类似的语法，就是任务和函数，设计者可以把所需的代码编写成任务和函数的形式，使代码更简洁。

1. 任务

任务的弹性程度比函数大，在任务中可以调用其他任务或函数，还可以包含延迟、时间控制等语法。从一个模块的代码结构上来讲，任务应该和 initial、always 结构同处于一个层次，严格来说它属于行为级建模，所以只要行为级可以使用的语法在任务中都是支持的，这一点要和后面的函数区分。任务的声明格式如程序 3.29 所示。

程序 3.29　任务的声明格式

```
task    任务名称;
input   [宽度声明] 输入信号名;
output  [宽度声明] 输出信号名;
inout   [宽度声明] 双向信号名;
reg     任务所用变量声明;
begin
⋮       任务包含语句
end
endtask
```

针对任务格式的语法要求，依次解释如下。

(1) 任务声明以 task 开始，以 endtask 结束，中间部分是任务包含的语句。

(2) 任务名称就是一个标识符，满足标识符语法要求即可。

(3) 任务可以有输入信号 input、输出信号 output、双向信号 inout 和供本任务使用的变量，变量不仅包括上面写出的 reg 型，行为级中支持的类型如 integer、time 等都可以使用。

(4) 任务从整体形式上看和模块十分相似，task 和 endtask 类似于 module 和 endmodule，但任务虽然有输入输出信号，却没有端口列表。

(5) 完成信号和变量的声明后，可以用 begin 和 end 封装 task 功能描述语句，也可以使用 fork 和 join 并行块来封装，但要注意此语句块结构前没有 initial 和 always 结构。

(6) 对于 begin…end 所包含的任务语句部分，遵循行为级建模语法即可。

举例如程序 3.30 所示。

程序 3.30　4 位全加器的任务

```
task   add4;
input  [3:0] x,y;
input  cin;
output [3:0] s;
output cout;

begin
   {cout,s} = x + y + cin;
end
endtask
```

任务调用时应采用如下格式：

```
任务名(信号对照列表);
```

例如，对程序 3.30 中的 add4 任务进行调用，就可以使用如下语句：

```
add4(a,b,c,d,e);
```

任务的调用要注意以下几点。

(1) 任务调用时要写出任务调用的名称来进行调用，这一点与模块实例化过程相似，但是任务调用不需要使用实例化名称，像 add4 这个任务名可直接写出调用对应任务。

(2) 任务的功能描述虽然和 always、initial 处于同一层次，但是任务调用必须发生在 initial、always、task 中。

(3) 任务中如果有输入、输出或双向信号，按照类似实例化语句中按名称连接的方式连接信号。

(4) 任务的信号连接也要遵循基本的连接要求。

(5) 任务调用后需要添加分号，作为行为级语句的一个语句处理。

(6) 任务不能实时输出内部值，而是只能在整个任务结束时得到一个最终的结果，输出的值也是这个最终结果的值。

2. 函数

函数与任务不同，任务其实没有太多的语法限制，可以把组合逻辑编写成任务，也可以使用时序控制等语法来完成任务。但对函数来说，仅仅可以把组合逻辑编写成函数，因为函数中不能有任何的时序语句，而且函数不能调用任务，这是受函数自身语法要求限制的。函数声明格式如程序 3.31 所示。

程序 3.31　函数的声明格式

```
function 返回值的类型和范围 函数名;
input [端口范围] 端口声明;
reg、integer 等变量声明;
begin
   阻塞赋值语句块
end
endfunction
```

函数的基本要求和注意事项如下。

(1) 函数以关键字 function 开头，以关键字 endfunction 结尾。

(2) 在关键字 function 后和函数名称之间，要添加返回值的类型和范围。定义返回值类型时如果不指定类型，则会默认定义为 reg 类型，如果没有指定范围，默认为 1 位。

(3) 函数至少需要一个输入信号，没有输出信号，所以 output 之类的声明是无效的。函数的运算结果就是通过上一步定义的返回值进行传递的，也就是说函数只能得到一个运算结果，相当于只有一个输出。

(4) 函数内部可以定义自身所需的变量。

(5) 函数的功能语句也可以用 begin…end 进行封装。虽然使用 fork…join 在语法上是

允许的，但出于可综合的角度考虑，一般还是使用顺序块。

（6）函数的 begin…end 块内部有一些要求。首先不能有任何时间相关的语法，如@引导的事件、#引导的延迟等，而且用于时序电路描述的非阻塞语句也不能使用，但 if 语句、case 语句或循环语句等与时序电路没有直接关系的语句仍然可以使用；其次必须要有语句明确规定函数中的返回值是如何得到赋值的。

举例如程序 3.32 所示。

程序 3.32　阶乘计算函数

```
function integer factorial;
input [3:0] a;
integer i;
begin
   factorial = 1;
   for(i = 2;i <= a;i = i + 1)
       factorial = i * factorial;
end
endfunction
```

函数调用格式如下：

```
待赋值变量 = 函数名称(信号对照表);
```

函数的调用需要注意以下事项。

（1）函数的调用不像任务调用一样可以只出现任务名，函数调用之后必须把返回值赋给某个变量。任务有输出信号，直接通过输出信号的连接就可以把任务所得的结果进行输出。而函数没有直接定义的输出信号，是通过返回值，采用把函数的返回值赋值给某个变量的形式完成输出。

（2）信号对照列表部分需要按照函数内部声明的顺序出现。

（3）函数调用也作为行为级建模的一条语句，出现在 initial、always、task、function 结构中，即函数可以被任务调用，但任务不能被函数调用。

Verilog HDL 除了可以允许设计者自己编写任务和函数外，还提供了可以直接使用的系统任务和系统函数。系统任务和函数都以 $ 作为开头，如 $monitor、$finish、$time 等，其调用方法和设计者自己编写的任务和函数完全相同。

3.3.8　设计的可综合性

用 Verilog HDL 编写的设计模块最终要生成实际工作的电路，因此，设计模块的语法和编写代码风格会对后期电路产生影响，所以，若要编写可实现的设计模块，就需要注意一些问题，本节将对此进行着重介绍。

1. 可综合语法

可综合的设计是最终实现电路所必需的，所以弄清哪些语法是可综合的、哪些语法是不可综合的非常有必要。而且设计者也必须知道一个代码能否被综合成最终电路，像写一个简单的除法 a/b，想妄图直接通过综合工具生成一个除法器是不现实的。类似的情况还可

能会出现在设计有符号数、浮点数等输入情况时，设计者的思路一定要从软件角度转变到硬件角度，很多在软件中可以直接使用的情况到了硬件电路就需要从很底层的角度来编写。

可综合设计先要弄清哪些语法可以被综合，按在模块中出现的顺序总结如下。

(1) module 和 endmodule 作为模块声明的关键字，必然是可以被综合的。

(2) 输入 input、输出 output 和双向端口 inout 的声明是可以被综合的。

(3) 变量类型 reg、wire、integer 都是可以被综合的。

(4) 参数 parameter 和宏定义 define 是可以被综合的。

(5) 所有的 Verilog HDL 内建门都是可以使用的，如 and、or 之类都是可以在可综合设计中使用的。

(6) 数据流级的 assign 语句是可以被综合的。

(7) 行为级中敏感列表支持电平和边沿变化，类似 posedge、negedge 都是可以被综合的。

(8) always、function 是可以被综合的，task 中若不含延迟也可以被综合。

(9) 顺序块 begin…end 可以被综合。

(10) if 和 case 语句可以被综合。

在 Verilog HDL 中不可被综合的语法这里也简单列出来，读者在设计可综合模型时应注意避免出现。

(1) 初始化 initial 结构不能被综合，电路中不会存在这样的单元。电路中一旦通电就会自动获得初始值，除此之外时序电路可以用复位端完成初始化组合，电路不需要初始化。

(2) ＃带来的延迟不可被综合。电路中同样也不会存在这样简单的延迟电路，所有的延迟都要通过计时电路或交互信号来完成。

(3) 并行块 fork…join 不可被综合，并行块的语义在电路中不能被转化。

(4) 用户自定义原语 UP 不可被综合。

(5) 时间变量 time 和实数变量 real 不能被综合。

(6) wait、event、repeat、forever 等行为级语法不可被综合。

(7) 一部分操作符可能不会被综合，例如，除法/操作和求余数％操作。

由于综合工具也在不断更新和加强，有些现在不能被综合的语法慢慢地会变得可以综合。像比较简单的 initial 结构在一些 FPGA 工具中也可以被识别，同时能被转化为电路形式。而有些语句是由于语法特点被综合工具限制了，比较典型的就是 for 语句。for 循环语句简洁明了，编写代码非常方便，但在综合过程中会被完全展开，如 for(i＝0；1＜9；i＝i＋l)这条语句在综合工具中就会被展开成 10 个语句并形成 10 个相似的电路，这些电路都会出现在最终的电路图里，造成电路规模展开过大。而且 for 循环中的 i 一般都比较大，这样展开的效果就更加明显。但使用 for 的时候设计者的思路其实是想要通过一个简单的电路完成判断，然后执行 for 所包含的语句，这样设计者和综合工具之间的处理过程不一样，只能以综合工具为准。在一些生成语句中可以由 for 循环生成一些基本单元门，此时设计思路和综合工具的处理过程一致，这时就是可以综合的。

不可综合的语句在仿真工具中是编译不出来的，因为仿真工具只能检查仿真相关的语法，不能考虑后期综合电路的情况，而仿真所用的测试模块没有语法限制，所以无法提供可综合语法的帮助。在实际的设计过程中读者可以直接使用一些 FPGA 的工具来尝试编译所写代码，理解哪些语法是可综合的、哪些是不可综合的。

2. 代码风格

Verilog HDL 的代码风格会影响到最后的电路实现，本节中仅对一些共通的规范做介绍，说明可能出现的问题。读者在学习过程中要注意养成一个基本的习惯，形成比较良好的代码风格。

1）阻塞与非阻塞

使用的赋值类型依赖于所描述的逻辑类型：在时序块 RTL 代码中使用非阻塞赋值，非阻塞赋值保存值直到时间片段的结束，从而避免仿真时的竞争情况或结果的不确定性；在组合的 RTL 代码中使用阻塞赋值，阻塞赋值立即执行。

当用 always 块为组合逻辑建模，使用“阻塞赋值”；当在同一个 always 块里面既为组合逻辑又为时序逻辑建模，使用“非阻塞赋值”；不要在同一个 always 块里面混合使用“阻塞赋值”和“非阻塞赋值”。

2）多重驱动问题

多重驱动问题是初学者最容易犯的错误之一，主要原因就是逻辑划分不清。在可综合的模块中，一个信号的赋值只发生在一个 always 结构中，如果出现在两个 always 结构中就构成了多重驱动，综合工具会认为这两个电路会尝试对同一个变量赋值，实际效果就会造成电路信号的碰撞，然后生成无法预料的结果。所以设计者在设计模块的时候一般都会在一个 always 结构中把某个输出的所有情况都写清楚，确保没有考虑不全的情况，然后再去编写其他输出的情况。多重驱动问题一般发生在有多个判断条件的情况时，此时的设计思路不要考虑“在这些情况下设计模块的输出都应该是什么”，而是要考虑“每个输出在这些情况下都应该输出什么”，也就是不要从情况入手，而要从输出的角度来看待电路。而在 Verilog HDL 编写设计模块的语法指导中也建议设计者每个 always 结构完成一个信号的赋值，除非几个信号产生变化的情况都相同或者信号之间有强烈的依赖关系时才放在一起。

3）敏感列表不完整

在@引导的敏感列表中必须包含完整的敏感列表，这是针对组合逻辑电路而言的。时序电路中@的敏感事件只是 clock 的边沿或 reset 一类信号的边沿情况，若出现其他变量就会变成异步电路，而异步电路的设计很多综合工具并不支持或支持得很差，需要人工帮助。组合逻辑电路敏感列表不完备就会造成仿真结果不正确，以及最终实现的电路结构不正确或出现锁存器结构。例如如下代码：

```
always @(a)
c = a ^ ~b;
```

这个代码中希望生成的是同一个电路，但是敏感列表缺少了 b，这样 b 的变化不会促使 always 结构发生变化。此代码综合后可能会生成一个带控制端的锁存器的电路形式，当然也可能是正确的，但设计者不能过分依赖综合工具。把敏感列表补充完整如下：

```
always@(a or b)
c = a ^ ~b;
```

4）if 与 else 不成对出现

观察如下程序 3.33。

程序 3.33　if 与 else 不成对出现举例

```
reg [1:0] out;
always @ (posedge clock)
begin
   if (s == 2'b00) out <= 2'b00;
   if (s == 2'b11) out <= 2'b11;
end
```

存在两个问题：代码优先级问题和 if…else 问题。

代码优先级问题是指两个 if 先判断哪一个？如果电路整体资源情况包括时序、面积等比较充裕，先执行哪个都可以，但如果电路资源情况比较紧张，这两个 if 语句的先后就可能决定了时序上的成功和失败。因为这两个 if 语句最后实现电路的情况完全由不同的综合工具自己定义，最终电路不可确定。

所谓 if…else 问题，就是出现了一个 if，必然要出现与之对应的 else，否则电路中就容易出现锁存器。锁存器这种电路结构在非故意使用的情况下出现就是错误的，而 else 的不使用是造成锁存器被综合出来的原因之一。将上述代码修改如下可以避免这两个问题如程序 3.34 所示。

程序 3.34　if 与 else 不成对出现修正举例

```
reg [1:0] out;
always @ (posedge clock)
begin
   if (s == 2'b00) out <= 2'b00;
   else if (s == 2'b11) out <= 2'b11;
   else out <= 2'b11;
end
```

5）case 语句缺少 default

在 case 语句中也容易出现锁存器，如程序 3.35 所示。

程序 3.35　case 语句缺少 default 举例

```
reg [1:0] sel;
 always @(sel,a,b)
 begin
 case (sel)
     2'b00:out = a + b;
     2'b01:out = a - b;
     2'b00:out = a + b;
     2'b00:out = a + b;
 end
```

该 case 语句中缺少了 default，效果和 if 语句中缺少 case 一样，容易被综合工具综合成锁存器，无论 default 情况是否存在都要添加这一项，而且不要对其赋值为 x，类似于：

```
default:out = 2'bxx;
```

这样在仿真过程中是可以的，运行时比较类似电路刚启动所处的未知状态，但实际综合过程中 x 值是被忽略的，所以要给出一个明确的赋值。如果设计者不知道应该在 default 中产生什么输出值，或者在 else 中产生什么输出值，也可以仅添加一个 default 而不添加任何语句，如下：

```
default:;
```

6）组合和时序混合设计

组合和时序混合设计是因为设计划分不清造成的，观察程序 3.36。

程序 3.36　组合和时序混合设计举例

```
reg x,y,z;
always @(x,y,z,posedge reset)
begin
if (reset)
    out = 0;
else
    out = x ^ y ^ z;
end
```

这个例子中，设计者一方面希望完成异或，另一方面又希望能在一个 always 结果中完成清零过程，得到一个混合设计模块。从根本上将二者划分开是最好的解决途径，拆分代码如程序 3.37 所示。

程序 3.37　组合和时序混合设计拆分举例

```
reg x,y,z;
always @(posedge reset)
begin
if (reset)
   begin
      x <= 0;
      y <= 0;
      z <= 0;
   end
else…
end
assign out = x ^ y ^ z;
```

在建立可综合模型时，能使用数据流语句实现组合逻辑电路时应尽量使用数据流描述建模，而不要使用行为级的阻塞赋值，因为 assign 语句层次较低，综合转化不容易发生歧义，所写语句与最后实现电路一致性较高。

3.4　Verilog HDL 测试平台描述

编写 TestBench 的目的主要是对使用硬件描述语言（HDL）设计的电路进行仿真验证，测试设计电路的功能、部分性能是否与预期的目标相符。编写 TestBench 进行测试的过程

大致如下。

(1) 实例化需要测试的设计；

(2) 产生模拟激励(波形)；

(3) 将产生的激励加入到被测试模块并观察其输出响应；

(4) 将输出响应与期望进行比较，从而判断设计的正确性。

3.4.1　基本的 TestBench 结构

TestBench 模块没有输入输出，在 TestBench 模块内实例化待测设计的顶层模块，并把测试行为的代码封装在内，直接对测试系统提供测试激励。下面是一个基本的 TestBench 结构模块。

```
module testbench;
    //数据类型声明
    //对被测试模块实例化
    //产生测试激励
    //对输出响应进行收集
endmodule
```

程序 3.38 是程序 3.39 带复位端的 D 触发器的测试模块，代码如下。

程序 3.38　对带复位端的 D 触发器进行验证的测试模块

```
`timescale 1ns/1ns
module tb92;
reg clock,reset,d;
wire q2,q3;                          //变量声明

initial clock = 0;
always #5 clock = ~clock;            //生成时钟信号
initial d = 1;
begin
   reset = 1;
   #12 reset = 0;
   #11 reset = 1;                    //仿真信号产生
   #17 $stop;                        //仿真控制
end
dff2 dff2(clock,reset,d,q2);         //待测模块的模块实例化
endmodule
```

程序 3.39　带复位端的 D 触发器

```
module dff2(clock,reset,d,q);
input clock,reset,d;
output q;
reg q;

always @(posedge clock or negedge reset)
begin
```

```
    if(!reset)
       q<=0;
    else
       q<=d;
end
endmodule
```

3.4.2 激励信号描述

1. 时钟信号

时钟信号是时序电路所必需的信号之一，该信号可以由多种方式产生。可以使用 initial 和 always 结构共同生成时钟信号，被动地检测响应时使用 always 语句，主动响应时使用 initial 语句。它们的区别是：initial 语句只执行一次，always 语句不断地重复执行，如程序 3.40 所示。

程序 3.40 initial 和 always 结构生成时钟信号

```
reg clock1;
 initial
    clock1 = 0;
 always
  #5 clock1 = ~clock1;
```

采用此代码生成的是一个占空比为 50%的时钟。还可以只用 always 结构生成时钟，如程序 3.41 所示。

程序 3.41 always 结构占空比为 50%的时钟举例

```
reg clock2;
always
begin
   #5 clock2 = 0;
   #5 clock2 = 1;
end
```

采用这种方式还可以生成任意占空比的时钟信号，如下代码生成了一个占空比为 75%的时钟，如程序 3.42 所示。

程序 3.42 always 结构任意占空比时钟举例

```
reg clock3;
always
begin
   #15 clock3 = 0;
   #5 clock3 = 1;
end
```

还可以仅使用 initial 结构来生成时钟，如程序 3.43 所示。

程序 3.43　initial 结构生成时钟举例

```
initial
begin
 clock4 = 0;
 forever
   #10 clock4 = ~clock4;
end
```

或者在 forever 基础上添加 begin…end 块，生成任意占空比的时钟信号，如程序 3.44 所示。

程序 3.44　forever 结构任意占空比时钟举例

```
initial
begin
    clock5 = 0;
    forever
    begin
      #10 clock5 = 1;
      #10 clock5 = 0;
    end
end
```

上述代码中的时钟周期值可以借助参数完成，将参数设置为全周期时间长度，然后采用除法完成所需周期，如程序 3.45 所示。

程序 3.45　时钟周期设置举例

```
reg clock6
parameter cycle = 15;
always
begin
   #(cycle/3) clock6 = 0;
   #((cycle/3) * 2) clock6 = 1;
end
```

2. 复位信号

由于时序电路一般会有一个复位端把电路回归到初始状态，所以为了保证时序电路的工作正确，仿真开始时会给电路一个复位信号使其完成初始化。出于电路稳定性和节约功耗两方面考虑，选择高电平作为复位信号，电路正常工作时只需要维持低电平即可。

复位信号可分为异步复位信号和同步复位信号。同步复位信号是指时钟有效沿到来时对触发器进行复位所产生的信号；异步复位信号不依赖于时钟信号，只在系统复位有效时产生复位信号。代码如程序 3.46 所示。

程序 3.46　复位信号举例

```
//异步复位信号
parameter PERIOD = 10;
reg Rst_n;
initial
begin
   Rst_n = 1;
   #PERIOD Rst_n = 0;          //10ns 时开始复位,持续时间 50ns
   #(5 * PERIOD)Rst_n = a;
end

//同步复位信号
initial
begin                          //将 Rst_n 初始化为 1,在第一个时钟下降沿复位,延时 30ns,
   Rst_n = 1;                  //在下一个时钟下降沿撤销复位
   @(negedge clock);           //等待时钟 clock 下降沿
   Rst_n = 0;
   #30
   @(negedge clock);
   Rst_n = 1;
end
```

3.4.3　编译指令

Verilog HDL 和 C 语言一样也提供了编译预处理的功能。Verilog HDL 允许在程序中使用几种特殊的命令,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再进行通常的编译处理。在 Verilog HDL 中,为了和一般的语句相区别,这些预处理命令以符号“`”开头(位于主键盘左上角,其对应的上键盘字符为“~”。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他命令定义替代该命令之处。本节对部分常用编译指令进行介绍。

1. 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时间单位的模块。例如,一个设计中包含两个模块,其中一个模块的时间延迟单位为纳秒(ns),另一个模块的时间延迟单位为皮秒(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(μs)、纳秒(ns)、皮秒(ps)、飞秒(fs)。在程序 3.47 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。

程序 3.47　`timescale 命令

```
`timescale 10 ns/1 ns
module test;
reg set;
parameter d = 1.55;
initial
begin
   #d set = 0;
   #d set = 1;
end
endmodule
```

2. 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)
```

例如:

```
`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 string 都替换成 signal。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。

关于宏定义需要注意以下几个问题。

(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。当需要改变某一个变量时,可以只改变 `define 命令行,一改全改。

(5) 宏定义是用宏名代替一个字符串,也就是做简单的置换,不做语法检查。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。

(7) 宏定义可以嵌套使用,如程序 3.48 所示。

程序 3.48 宏定义嵌套使用举例

```
module test;
reg a,b,c;
wire out;
`define aa a + b
`define cc c + `aa
assign out = `cc;
endmodule
```

(8) 如果不想让宏定义生效,可以使用 `undef 指令取消前面定义的宏,如程序 3.49 所示。

程序 3.49 宏定义取消举例

```
`define WORD 16
…
wire [1:`WORD] bus;
…
`undef WORD        //此语句之后,WORD 无效
reg [0:`WORD - 1] cev;
```

3. 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时才进行编译,也就是对一部分内容指定编译的条件,这就是条件编译。条件编译命令的一般形式如下:

```
`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。这里的“程序段”可以是 Verilog HDL 语句组,也可以是命令行。需要注意的是: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。

(1) 选择一个模块的不同代表部分。

(2) 选择不同的时序或结构信息。

(3) 对不同的 EDA 工具,选择不同的激励。

最常用的情况是：Verilog 代码中的一部分可能适用于某个编译环境，但不适用于另一个环境。如设计者不想为两个环境创建两个不同版本的 Verilog 设计，还有一种方法就是所谓的条件编译，即设计者在代码中指定其中某一部分只有在设置了特定的标志后，这一段代码才能被编译。

4. 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 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"
```

文件包含命令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。

关于文件包含处理命令要注意以下几个问题。

(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。

(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。

(3) 可以将多个 `include 命令写在一行，在 `include 命令行，可以出现空格和注释行。例如，下面的写法是合法的。

```
`include "fileB"  `include "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 中用两个 `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。这样在文件 2 中不用包含文件 3。

(5) 文件的包含可以嵌套。

许多 Verilog 编译器支持多模块编译，也就是说只要把需要用 `include 包含的所有文件都放置在一个项目中。建立存放编译结果的库，用模块名就可以把所有有关的模块联系在一起，此时在程序模块中就不必使用 `include 编译预处理指令。

3.4.4　测试相关的系统任务和系统函数

1. 显示任务 $display 和 $write

这两个任务能够把指定的信息输出到输出设备中，如仿真器的显示窗口。任务格式如下：

```
$display/ $write (p1,p2,…,pn);
```

这两个任务的作用基本相同，即将参数 p2 到 pn 按参数 pl 给定的格式输出。$display 自动地在输出后进行换行，$write 则不换行。如果想在一行里输出多个信息，可以使用 $write。参数 p1 通常称为“格式控制”，参数 p2 至 pn 通常称为“输出表列”。在 $display 和 $write 中，其输出格式控制是用双引号括起来的字符串，它包括以下两种信息。

(1) 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。表 3.5 中给出了常用的几种输出格式。

表 3.5 输出格式及说明

输出格式	说 明	输出格式	说 明
%h 或%H	以十六进制数的形式输出	%s 或%S	以字符串形式输出
%d 或%D	以十进制数的形式输出	%e 或%E	以指数的形式输出实型数
%o 或%O	以八进制数的形式输出	%t 或%T	以当前的时间格式输出
%b 或%B	以二进制数的形式输出	%f 或%F	以十进制数的形式输出实型数
%c 或%C	以 ASCII 码字符的形式输出	%g 或%G	以指数或十进制数的形式输出实型数,以较短的结果输出
%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

(2) 特殊字符,用于格式字符串参数中显示特殊的字符。其输出方式见表 3.6。

表 3.6 特殊字符输出方式

输出格式	功 能	输出格式	功 能
\n	换行	\"	双引号字符"
\t	横向跳格	%%	百分符号%
\\	反斜杠字符\	\o	将 o 表示的 1～3 位八进制数代表的字符输出

在 $display 和 $write 的参数列表中,其“输出列表”需要输出一些数据,可以是表达式,如程序 3.50 所示。

程序 3.50 显示任务输出举例

```
module disp;
initial
begin
    $display("\\\t% %\n\"\123");
end
endmodule
```

输出结果为:

```
\%
"S
```

如果输出列表中表达式的值包含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:

① 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。

② 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。

③ 如果表达式值的部分位为不定值,则输出结果为大写的 X。

④ 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:

① 每 4 位二进制数为一组代表一位十六进制数,每三位二进制数为一组代表一位八进制数。

② 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。

③ 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。

④ 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。

⑤ 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。

(3) 对于二进制输出格式，表达式值的每一位的输出结果为 O、l、x、z。

2. 监视任务 $monitor

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，格式如下：

```
$monitor(p1,2,…,pn);
$monitor;
$monitoron;
$monitoroff;
```

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。例如：

```
$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地控制 $monitor 何时发生。其中，$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。在默认情况下，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. 探测任务 $strobe

探测任务的语法和显示任务完全相同，也是把信息显示出来，格式如下：

```
$strobe(p1,p2,…,pn);
```

$strobe 命令和 $display 命令的区别是：$strobe 命令会在当前时间步骤结束时完成，即发生在向下一个时间步骤运行之前；$display 是只要被仿真器看到，就会立即执行。

4. 仿真控制任务 $finish

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，格式如下：

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

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。

5. 仿真控制任务 $stop

$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，格式如下：

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

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。

6. 文件控制任务 $readmemb 和 $readmemh

在 Verilog HDL 程序中有两个系统任务 $readmemb 和 $readmemh 用来从文件中读取数据到存储器中。$readmemb 要求文件中必须是二进制数值，$readmemh 要求文件中必须是十六进制数值。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几种。

```
$readmemb/ $readmemh ("<数据文件名>",<存储器名>);
$readmemb/ $readmemh ("<数据文件名>",<存储器名>,<起始地址>);
$readmemb/ $readmemh ("<数据文件名>",<存储器名>,<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、制表符和换页符)，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，数字中不定值 x 或 X，高阻值 z 或 Z 和下画线(_)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。

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储器单元中去。存储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。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储器中相应的地址单元中去。代码如程序3.51 所示。

程序 3.51　系统函数 $readmemb 仿真样例

```
module test;
   reg [7:0] memory [0:7];                    //声明有 8 个 8 位的存储单元
   integer i;
```

```
    initial
    begin
      $readmemb("init.dat",memory);                    //读取存储器文件 init.dat 到存储器中的
                                                       //给定地址
    for (i = 0;1 < 8;i = i + 1)
     $display("Memory[ %d] = %b",i,memory[i] );        //显示初始化后的存储器内容
    end
endmodule
```

文件 init.dat 包含初始化数据。用@＜地址＞在数据文件中指定地址，地址以十六进制数说明。数据用空格符分隔。数据可以包含 x 或者 z。未初始化的位置默认值为 x。名为 init.dat 的样本文件内容如下所示：

```
@002
11111111  01010101
00000000  10101010

@006
1111zzzz  00001111
```

当仿真测试模块时，将得到下面的输出：

```
Memory[0] = xxxxxxxx
Memory[1] = xxxxxxxx
Memory[2] = 11111111
Memory[3] = 01010101
Memory[4] = 00000000
Memory[5] = 10101010
Memory[6] = 1111zzzz
Memory[7] = 00001111
```

对于 $readmemb 和 $readmemh 系统任务格式，需要补充说明以下 5 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，则默认的存放起始地址为该存储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储器中，直到该存储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，存放到该存储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储器单元中，直到该结束地址，而不考虑该存储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，并且装载数据到存储器中的操作被中断。

(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。

7. 仿真时间函数 $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 可以返回一个 64 位的整数来表示当前仿真时刻值。该时刻是以模块的仿真时间尺度为基准的，如程序 3.52 所示。

程序 3.52　系统函数 $time 仿真样例

```
`timescale 10 ns/1 ns
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 想在时间为 16ns 时设置寄存器 set 为 0，在时间为 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。在程序 3.52 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。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。程序 3.52 中的 1.6 和 3.2 经取整后为 2 和 3 输出。

（2）$realtime 系统函数和 $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，如程序 3.53 所示。

程序 3.53　系统函数 $realtime 仿真样例

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

从上述结果可以看出，$realtime 将仿真时刻经过尺度变换以后输出，无须进行取整操作。注意，时间的精确度不影响取整过程。

8. 随机函数 $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 32 位的随机数。它是一个带符号的整型数。$random 一般的用法是：

```
$random % 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－59～59 的随机数，下面的例子通过位并接操作产生一个值在 0～59 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。程序 3.54 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。

程序 3.54　生成宽度随机的随机脉冲序列的测试信号源

```
`timescale 1ns/1ns
module random_pulse(dout);
output [9:0] dout;
reg [9:0] 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 0～100ns 变化
           delay2 = 20 * (1 + { $random} % 3);          // delay2 在 20～60ns 变化
           #delay1 dout  =  1 <<({ $random} % 10);
           //dout 的 0～9 位中随机出现 1,并且出现的时间在 0～100ns 变化
           #delay2 dout = 0;                            //脉冲的宽度在 20～60ns 变化
         end
    end
endmodule
```

3.5 状态机描述

状态机通常用于构建转移状态有限的系统。其中,转移的过程取决于当前状态和外部输入。在实际操作中,状态机的主要应用是作为大型数字系统的控制器,该系统将检查外部命令和状态,并激活适当的控制信号来控制一个数据通路,该数据通路通常是由常规的时序电路组成。这也被称为带有数据通路的状态机。本章将首先对状态机的类型和表示方法进行概述,其次给出状态机的 Verilog HDL 描述方法。

3.5.1 状态机类型

如图 3.7 所示,状态机和常规时序电路的基本框图是相同的。它由一个状态寄存器,下一状态逻辑以及输出逻辑共同组成。状态机通常分为两类:若输出只和状态有关而与输入无关,则称为摩尔状态机;若输出不仅和状态有关,而且和输入也有关系,则称为米里状态机。这两种类型的输出都可能存在于一个复杂的状态机中,我们简单地称其包含摩尔输出和米里输出。摩尔输出和米里输出相似但不尽相同,了解它们的细微差异是设计控制器的关键。

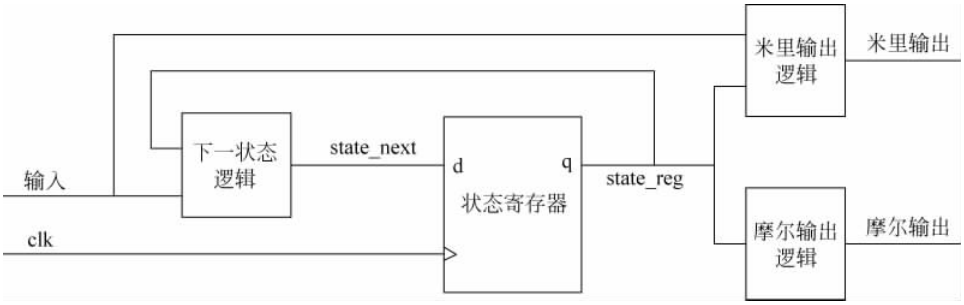

图 3.7 同步状态机原理框图

3.5.2 状态机表示方法

状态机通常由抽象的状态图或算法状态机图(ASM 流程图)来表示,这两种表示方法提供了相同的信息,都会在图形表示中给出状态机的输入、输出、状态和转移。状态图表示方法对于简单的应用能够很好地描述,而 ASM 流程图表示方法有点儿像一个流程图,对于转移条件复杂的应用能够更好地描述。

1. 状态图

状态图是由结点和带注释的转移弧组成的,每一个结点被画成一个圆用来表示状态。一个单一的结点和它的转移弧如图 3.8(a)所示。以输入信号表示的逻辑表达式与每一个转移弧相关联,并用来表示一个特定的条件。当相应的表达式被确定为真时,此转移弧就被选中。

一个具有代表性的状态图如图 3.9(a)所示。状态机有三种状态,两个外部输入信号(a 和 b),一个摩尔输出信号(y1),和一个米里输出信号(y0)。当状态机处于 s0 或 s1 的状

态中，y1 信号就会置为有效。当状态机处于 s0 的状态中，并且 a 和 b 的信号都为 1，y0 信号就会置为有效。由于摩尔的输出值只依赖于当前状态，所以它被放置在了结点中。而米里的输出值则关联了转移弧的条件，因为它们依赖于当前状态和外部输入。为了减少图表中的分支，只有有效的输出值被列出，其他情况下输出信号采用默认值。

2. ASM 流程图

ASM 流程图的基本单元是 ASM 块，整个 ASM 流程图由 ASM 块的连接组成。一个 ASM 块包含一个状态框，一个可选的决策框，以及条件输出框。图 3.8(b)给出了 ASM 块示例。

每一个状态框都表示了状态机的一种状态，同时也列出了有效的摩尔输出值。需要注意的是，它只存在一个出口路径。决策框会对输入条件进行判断，并确定需要选择的出口路径。决策框有两个出口路径，标记为 T 和 F，分别与条件值的真和假相对应。条件输出框则列出了有效的米里输出值，并且通常是放在决策框之后。它表明了只有决策框中相应的条件满足时，其所列的输出信号才可以被激活。

一个状态图可以很容易地转换为一个 ASM 流程图，反之亦然。图 3.9(b)给出了状态图 3.9(a)对应的 ASM 流程图。

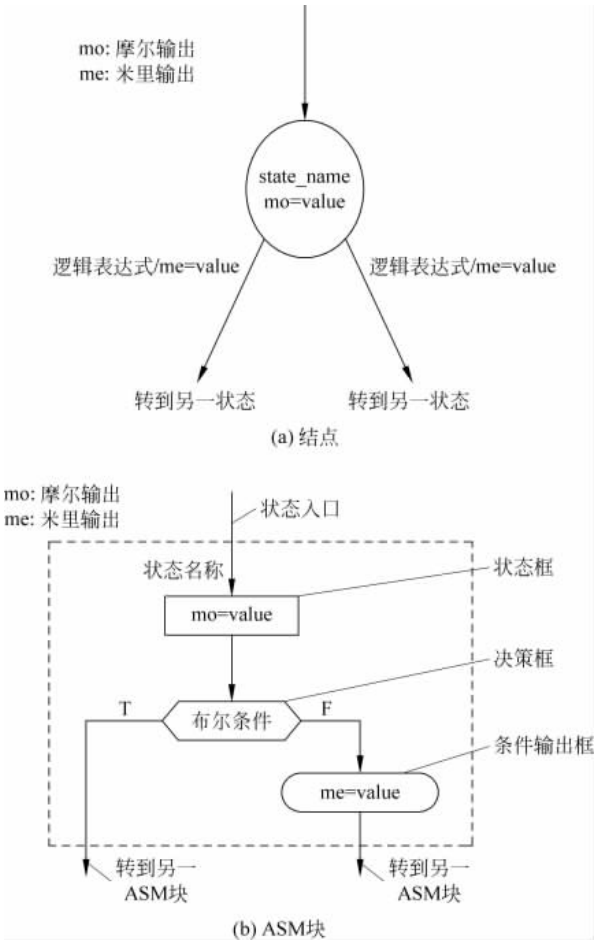

图 3.8　状态标志

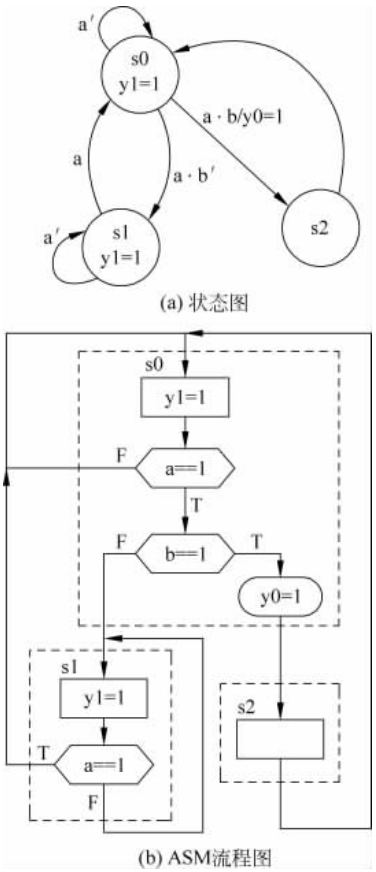

图 3.9　状态机示例

3.5.3 状态机的 Verilog HDL 描述方法

状态机描述的整体流程和常规时序电路类似。首先需要分配状态寄存器，然后给出下一状态逻辑和输出逻辑的组合代码。其中主要的区别就是下一状态的逻辑。对于状态机而言，下一状态逻辑的代码通常遵循状态流程或 ASM 流程图。

为了清晰和灵活，常使用符号常量来表示状态机的状态。举例来说，在图 3.9 中的三种状态可以被定义为

```
localparam [1:0] s0 = 2'b00,
                 s1 = 2'b01,
                 s2 = 2'b10;
```

在综合过程中，软件通常可以识别状态机的结构，并且可以把这些符号常数映射为不同的二进制表示(例如独热码)，这一过程被称为状态分配。

状态机的完整源代码在程序 3.55 中给出。其中包含状态寄存器、下一状态逻辑、摩尔输出逻辑，以及米里输出逻辑几个部分。其中的关键部分是下一状态逻辑。它使用了包含 state_reg 信号的 case 语句来作为选择表达式。下一状态(state_next 信号)是由当前状态(state_reg)和外部输入来确定的。每个状态的代码基本上是对应图 3.9(b)中每个 ASM 块内的活动。

程序 3.55 状态机样例

```
module fsm_eg_mult_seg
    (
     input wire clk, reset,
     input wire a, b,
     output wire y0, y1
    );
    // 符号状态声明
    localparam [1:0] s0 = 2'b00,
                     s1 = 2'b01,
                     s2 = 2'b10;
    //信号声明
     reg [1:0] state_reg, state_next;
    //状态寄存器
     always @(posedge clk, posedge reset)
     begin
        if (reset)
           state_reg <= s0;
        else
           state_reg <= state_next;
     end
    //下一状态逻辑
    always @ *
        case (state_reg)
            s0: begin
            if (a)
```

```
            begin
                    if(b)
                        state_next = s2;
                    else
                        state_next = s1;
                end
                else
                    state_next = s0;
                end
            s1:begin
                if(a)
                    state_next = s0;
                else
                    state_next = s1;
                end
            s2: state_next = s0;
            default: state_next = s0;
        endcase
        end
    //摩尔输出逻辑
    assign y1 = (state_reg == s0) || (state_reg == s1);
    //米里输出逻辑
    assign y0 = (state_reg == s0) & a & b;

endmodule
```

状态机的另一种 Verilog HDL 代码描述方式是将下一状态逻辑和输出逻辑合并为一个单独的组合模块，如程序 3.56 所示。下一状态逻辑和输出逻辑的代码都严格地遵循 ASM 流程图。一旦画出了详细的状态图或 ASM 流程图，将状态机转换成 Verilog HDL 代码几乎只是一个机械过程。程序 3.55 和程序 3.56 可以作为模板来完成这个过程。

程序 3.56　合并组合逻辑的状态机

```
module fsm_eg_2_seg
    (
     input wire clk, reset,
     input wire a, b,
     output reg y0, y1
    );
     //符号状态声明
    localparam [1:0] s0 = 2'b00,
                     s1 = 2'b01,
                     s2 = 2'b10;
    //信号声明
     reg [1:0] state_reg, state_next;
    //状态寄存器
     always @(posedge clk, posedge reset)
     begin
```

```
        if (reset)
           state_reg <= s0;
        else
           state_reg <= state_next;
    end
    //下一状态逻辑和输出逻辑
    always @ *
    begin
        state_next = state_reg;          // 默认 next_state: state_reg
        y1 = 1'b0;                       // 默认输出: 0
        y0 = 1'b0;                       // 默认输出: 0
        case (state_reg)
            s0: begin
                    y1 = 1'b1;
                    if (a)
                    begin
                       if (b)
                           begin
                               state_next = s2;
                               y0 = 1'b1;
                           end
                        else
                            state_next = s1;
                    end
                end
            s1: begin
                y1 = 1'b1;
                if (a)
                   state_next = s0;
                end
            s2: state_next = s0;
            default: state_next = s0;
        endcase
    end
endmodule
```

3.5.4 状态机设计实例——上升沿检测器

当输入信号从 0 转为 1 时,电路会生成一个很短的时间周期标记,也就是上升沿检测器。它通常用来表示一个随时间缓慢变化的输入信号的开始。本节将分别基于摩尔状态机和米里状态机进行上升沿检测器的设计,并比较它们的不同之处。

1. 基于摩尔状态机的设计

基于摩尔状态机的边缘检测器的状态图和 ASM 流程图见图 3.10。状态 zero 和 one 表示在一段时间里输入信号为 0 和 1。当输入在 zero 状态变为 1 时则出现上升沿。状态机将会进入到 edg 状态并使得输出 tick 置为有效。比较典型的时序图可以参考图 3.11 的中间部分,代码见程序 3.57。

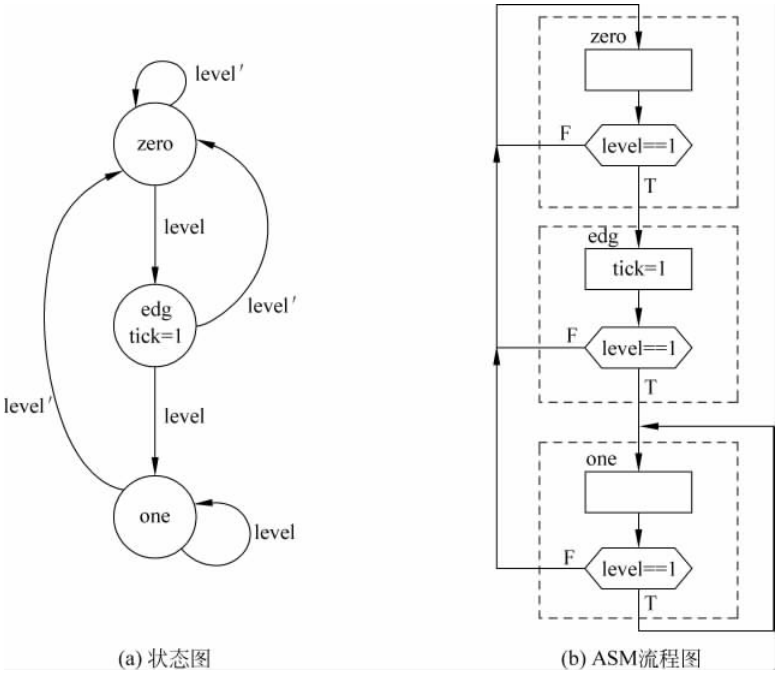

图 3.10　基于摩尔状态机的上升沿检测器

程序 3.57　基于摩尔状态机的上升沿检测器代码实现

```
module edge_detect_moore
    (
     input wire clk, reset,
     input wire level,
     output reg tick
    );
    //符号状态声明
    localparam [1:0]
        zero = 2'b00,
        edg = 2'b01,
        one = 2'b10;
    //信号声明
    reg [1:0] state_reg, state_next;
    //状态寄存器
    always @(posedge clk, posedge reset)
    begin
        if (reset)
            state_reg <= zero;
        else
            state_reg <= state_next;
    end
    //下一状态逻辑和输出逻辑
    always @ *
    begin
```

```
        state_next = state_reg;          //默认状态: state_reg
        tick = 1'b0;                     //默认输出: 0
        case (state_reg)
            zero:
                if (level)
                    state_next = edg;
            edg:
                begin
                    tick = 1'b1;
                    if (level)
                        state_next = one;
                    else
                        state_next = zero;
                end
            one:
                if (~level)
                    state_next = zero;
            default: state_next = zero;
        endcase
    end
endmodule
```

2. 基于米里状态机的设计

基于米里状态机的上升沿检测器的状态图和 ASM 流程图见图 3.12。在 zero 和 one 状态具有类似的含义。当状态机处于零状态且输入变为 1 时，输出立即置为有效。当状态机在下一时钟周期的上升沿进入 one 状态时，输出被置为无效。典型的时序图可以参考图 3.11 的下面部分。

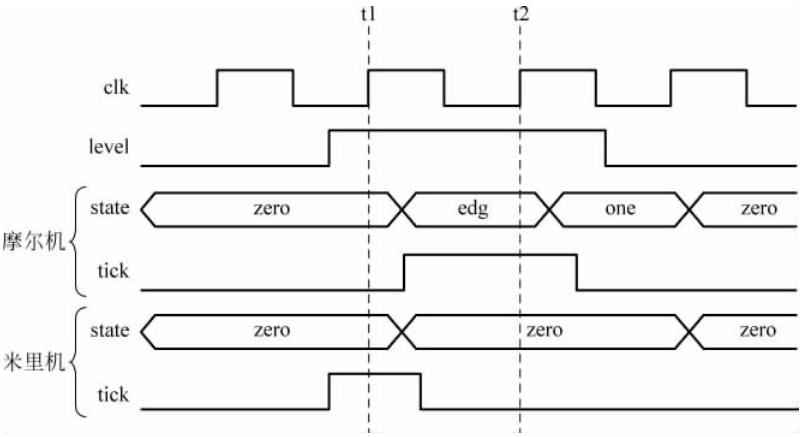

图 3.11　两种上升沿检测器的时序图

需要注意的是，由于传输延迟，输出信号在下一时钟周期的上升沿依然有效(t1 区间中)。具体代码见程序 3.58。

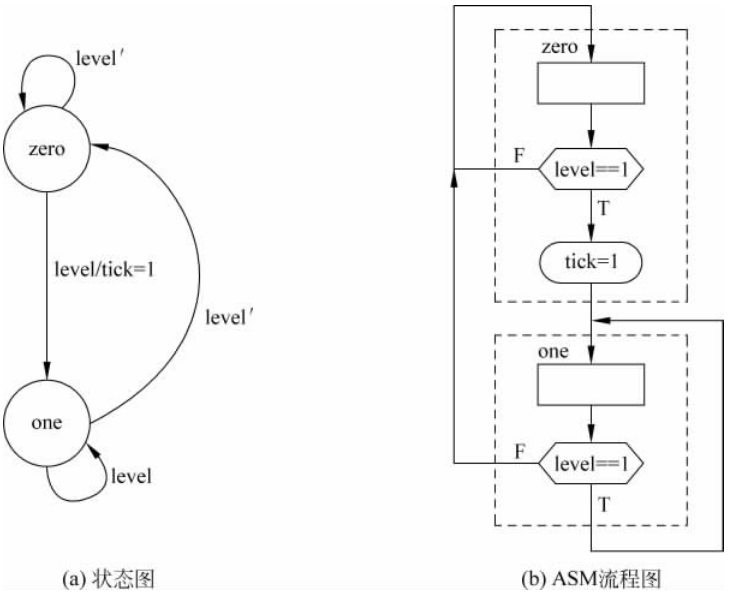

图 3.12　基于米里状态机的上升沿检测器

程序 3.58　基于米里状态机的上升沿检测器代码实现

```
module edge_detect_mealy
    (
     input wire clk, reset,
     input wire level,
     output reg tick
    );
    //符号状态声明
    local param zero = 1'b0,
                one = 1'b1;
    //信号声明
    reg state_reg, state_next;
    //状态寄存器
    always @(posedge clk, posedge reset)
    begin
        if (reset)
           state_reg <= zero;
        else
           state_reg <= state_next;
    end
    //下一状态逻辑和输出逻辑
    always @ *
    begin
        state_next = state_reg;          //默认状态: state_reg
        tick = 1'b0;                     //默认输出: 0
        case (state_reg)
            zero:begin
                if (level)
```

```
                    begin
                        tick = 1'b1;
                        state_next = one;
                    end
                end
            one:
                if (~level)
                    state_next = zero;
            default: state_next = zero;
        endcase
    end
endmodule
```

3. 直接实现

由于上升沿检测器电路的转换状态是很简单的,因此可以不借助状态机就直接实现。为了进行更直观的比较,图 3.13 给出了上升沿检测器的门级实现。该电路图可以这样理解,只有在当前输入是 1 且保存在寄存器中的前一输入为 0 时,输出才认为是有效的。相应的代码可以参考程序 3.59。

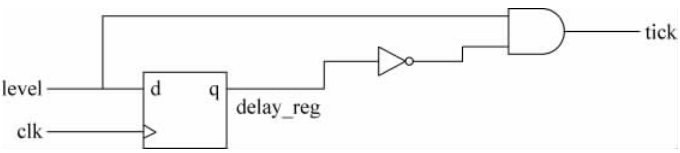

图 3.13　上升沿检测器的门级实现

程序 3.59　上升沿检测器的门级实现

```
module edge_detect_gate
    (
     input wire clk, reset,
     input wire level,
     output wire tick
    );
    //信号声明
    reg delay_reg;
    //延时寄存器
     always @(posedge clk, posedge reset)
     begin
        if (reset)
            delay_reg <= 1'b0;
        else
            delay_reg <= level;
     end
    //解码逻辑
     assign tick = ~delay_reg & level;
endmodule
```

尽管在程序 3.58 和程序 3.59 中的描述似乎有很大的差异，但实际上它们描述的是同样的电路。如果将 0 和 1 作为 zero 和 one 状态，那么电路图可以由状态机得到。

4. 对比

鉴于基于摩尔状态机和基于米里状态机的设计都可以在输入信号的上升沿产生一个短的时间基准，故只有几个细微的差别。基于米里状态机的设计需要较少的状态且响应速度快，但它的输出宽度会发生变化，并且输入错误可能影响输出。

对于两种设计之间的选择，主要取决于使用输出信号的子系统。多数情况下，子系统是共享时钟信号的同步系统。由于状态机的输出仅在时钟周期的上升沿采样，只要输出信号在边沿保持稳定，输出宽度和输入错误并不会造成影响。需要注意的是，米里输出信号在 t1 阶段就可用于采样，而摩尔输出要在 t2 阶段才可以采样，故米里输出相比于摩尔输出快了一个时钟周期。因此，基于米里状态机的电路更适合这种类型的应用。