《FPGA 应用开发入门与典型实例》（修订版）

作者：华清远见

第 3 章　硬件描述语言 Verilog HDL 基础

本章目标

- 初步掌握 Verilog HDL 语言的基本语句
- 学会用 Verilog HDL 语言设计简单的组合逻辑和时序逻辑电路
- 通过典型实例，熟悉 FPGA 设计的完整流程

专业始于专注 卓识源于远见

3.1 Verilog HDL 语言简介

3.1.1 Verilog HDL 的历史和进展

1．什么是 Verilog HDL

Verilog HDL 是硬件描述语言的一种，用于数字电子系统设计。它允许设计者用它来进行各种级别的逻辑设计，可以用它进行数字逻辑系统的仿真验证、时序分析、逻辑综合。它是目前应用最广泛的一种硬件描述语言之一。

2．Verilog HDL 的历史

Verilog HDL 是在 1983 年由 GDA（GateWay Design Automation）公司的 Phil Moorby 首创的。Phil Moorby 后来成为 Verilog-XL 的主要设计者和 Cadence 公司（Cadence Design System）的第一个合伙人。
在 1984 年～1985 年，Moorby 设计出了第一个关于 Verilog-XL 的仿真器，1986 年，他对 Verilog HDL 的发展又做出了另一个巨大贡献：即提出了用于快速门级仿真的 XL 算法。
随着 Verilog-XL 算法的成功，Verilog HDL 语言得到迅速发展。1989 年，Cadence 公司收购了 GDA 公司，Verilog HDL 语言成为 Cadence 公司的私有财产。1990 年，Cadence 公司决定公开 Verilog HDL 语言，于是成立了 OVI（Open Verilog International）组织来负责 Verilog HDL 语言的发展。

3．Verilog HDL 的进展

基于 Verilog HDL 的优越性，IEEE 于 1995 年制定了 Verilog HDL 的 IEEE 标准，即 Verilog HDL1364-1995。其后，又在 2001 年发布了 Verilog HDL1364-2001 标准。
据有关文献报道，目前在美国使用 Verilog HDL 进行设计的工程师大约有 60000 人，全美国有 200 多所大学教授用 Verilog 硬件描述语言的设计方法。在我国台湾地区几乎所有著名大学的电子和计算机工程系都讲授 Verilog 有关的课程。

3.1.2 VHDL 和 Verilog HDL 语言对比

Verilog HDL 和 VHDL 都是用于逻辑设计的硬件描述语言，并且都已成为 IEEE 标准。VHDL 是在 1987 年成为 IEEE 标准，Verilog HDL 则在 1995 年才正式成为 IEEE 标准。
之所以 VHDL 比 Verilog HDL 早成为 IEEE 标准，这是因为 VHDL 是美国军方组织开发的，而 Verilog HDL 则是从一个普通的民间公司的私有财产转化而来。
VHDL 其英文全名为 VHSIC Hardware Description Language，而 VHSIC 则是 Very High Speed Integrated Circuit 的缩写，意为甚高速集成电路，故 VHDL 其准确的中文译名为甚高速集成电路的硬件描述语言。

1．共同点

Verilog HDL 和 VHDL 作为描述硬件电路设计的语言，其共同的特点在于。

- 能形式化地抽象表示电路的结构和行为。
- 支持逻辑设计中层次与领域的描述。
- 可借用高级语言的精巧结构来简化电路的描述。

- 具有电路仿真与验证机制以保证设计的正确性。
- 支持电路描述由高层到低层的综合转换。
- 硬件描述与实现工艺无关（有关工艺参数可通过语言提供的属性包括进去）。
- 便于文档管理，易于理解和设计重用。

2．不同点

但是 Verilog HDL 和 VHDL 又各有其自己的特点。

由于 Verilog HDL 早在 1983 年就已推出，因而 Verilog HDL 拥有更广泛的设计群体，成熟的资源也远比 VHDL 丰富。

与 VHDL 相比，Verilog HDL 的最大优点是：它是一种非常容易掌握的硬件描述语言，只要有 C 语言的编程基础，通过二十学时的学习，再加上一段时间的实际操作，可在二～三个月内掌握这种设计技术。

而掌握 VHDL 设计技术就比较困难。这是因为 VHDL 不很直观，需要有 Ada 编程基础。

目前版本的 Verilog HDL 和 VHDL 在行为级抽象建模的覆盖范围方面也有所不同。一般认为 Verilog HDL 在系统级抽象方面比 VHDL 略差一些，而在门级开关电路描述方面比 VHDL 强得多。

3.2 Verilog HDL 程序基本结构

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。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 5 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他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 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，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。下面将介绍 Verilog HDL 中的基本结构和语法。

3.2.1 Verilog HDL 程序入门

首先来看几个 Verilog HDL 程序，然后从中分析 Verilog HDL 程序的特性。

例 3.1：加法器。

```
module  adder  ( count,sum,a,b,cin ) ;        //加法器模块端口声明
     input [2:0] a,b;                         //端口说明
     input  cin;
     output  count;
     output [2:0] sum;
     assign {count,sum} = a + b + cin;        //加法器算法实现
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 3.2：比较器。

```
module  compare  ( equal,a,b ) ;      //比较器模块端口声明
       output  equal;                 //输出信号 equal
       input [1:0] a,b;               //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 3.3：使用原语的三态驱动器。

```
module  trist2(out,in,enable);     //三态启动器模块端口声明
    output  out;                   //端口说明
    input   in, enable;
    bufif1  mybuf(out,in,enable); //实例化宏模块 bufif1
endmodule
```

这个例子描述了一个名为 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 bufifl 来实现其功能。

例 3.4：自行设计的三态驱动器。

```
module  trist1(out,in,enable);     //三态启动器模块端口声明
    output  out;                   //端口说明
    input  in, enable;
    mytri  tri_inst(out,in,enable);//实例化由 mytri 模块定义的实例元件 tri_inst
endmodule
```

```
    module  mytri（out,in,enable）;     //三态启动器模块端口声明
    output  out;                        //端口说明
    input  in, enable;
    assign  out = enable? in : 'bz;     //三态启动器算法描述
endmodule
```

这个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。

3.2.2 模块的框架

模块的内容包括 I/O 声明、I/O 说明、内部信号声明和功能定义。

1．I/O 声明

模块的端口声明了模块的输入输出端口，其格式如下：

```
Module     模块名（端口 1，端口 2，端口 3，端口 4, …）;
```

2．I/O 说明

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… ）
```

3．内部信号声明

在模块内用到的和与端口有关的 wire 和 reg 变量的声明，如下所示：

```
reg ［width-1 : 0］ R 变量 1，R 变量 2 …;
wire ［width-1 : 0］ W 变量 1，W 变量 2 …;
```

4．功能定义

模块中最重要的部分是逻辑功能定义部分，有 3 种方法可在模块中产生逻辑。
（1）用“assign”声明语句。

```
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。

```
and and_inst（ q, a, b ）;
```

采用实例元件的方法在电路图输入方式下，调入库元件。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惟一的，以避免与其他调用与门（and）的实例混淆。
（3）用“always”块。

```
always @（posedge clk or posedge clr） begin    //时钟上升沿触发，异步清零
   if（clr）  q <= 0;                            //清零
   else  if（en） q <= d;                        //使能赋值
   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。
需要注意的是，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 3 个例子分别采用了“assign”语句、实例元件和“always”块。这 3 个例子描述的逻辑功能是同时执行的。也就是说，如果把这 3 项写到一个 Verilog 模块文件中去，它们的次序不会影响逻辑实现的功能。这 3 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，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。

3.3 Verilog HDL 语言的数据类型和运算符

3.3.1 常用数据类型

Verilog HDL 中总共有 19 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。在本书中，我们先只介绍 4 个最基本的数据类型，它们分别是：reg 型，wire 型，integer 型和 parameter 型。其他数据类型在后面的章节里逐步介绍，读者也可以查阅附录中 Verilog HDL 语法参考书的有关章节逐步掌握。其他的类型如下：
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 型和 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。下面对最常用的几种进行介绍。

3.3.1.1 常量

常量是在程序运行过程中其值不能被改变的量。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。

1．数字

（1）整数。
在 Verilog HDL 中，整型常量有以下 4 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 3 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 位二进制数数字的位宽为 4，一个 4 位十六进制数数字的位宽为 16（因为每单个十六进制数要用 4 位二进制数来表示），如下例所示：

```
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。
```

（2）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 可以用来定义十六/八/二进制数的四/三/一位二进制数的状态。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，如下例所示：

```
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
```

（3）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，并且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，如下例所示：

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

（4）下划线（underscore_）。
下划线可以用来分隔数字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，例如：

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

当常量不声明位数时，默认值是 32 位，每个字母用 8 位的 ASCII 值表示，例如：

```
10 = 32'd10 = 32'b1010           //十进制和二进制
1=32'd1=32'b1                    //十进制和二进制
-1=-32'd1=32'hFFFFFFFF           //十进制和十六进制
'BX=32'BX=32'BXXXXXXX…X          //默认声明为 32 位
"AB"=16'B01000001_01000010       //每个字母用 8 位表示
```

2．参数（Parameter）

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，例如：

```
parameter  msb=7;                                  //定义参数 msb 为常量 7
parameter  e=25, f=29;                             //定义两个常数参数
parameter  r=5.7;                                  //声明 r 为一个实型参数
parameter  byte_size=8, byte_msb=byte_size-1;      //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。下面将通过一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。

```
module Decode(A,F);                                //模块声明
     parameter Width=1, Polarity=1;                //参数声明
     ……
endmodule

module Top;
     wire[3:0] A4;                                 //连线资源声明
     wire[4:0] A5;
     wire[15:0] F16;
     wire[31:0] F32;
     Decode  #(4,0) D1(A4,F16);                    //模块引用，并传递参数（4,0）
     Decode  #(5)  D2(A5,F32);                     //模块引用，并传递参数（5）
endmodule
```

在引用 Decode 实例时，D1 和 D2 的 Width 将采用不同的值，分别为 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，Polarit=1”。

3.3.1.2 变量

变量是在程序运行过程中，其值可以改变的量。在 Verilog HDL 中变量类型有很多种，这里只对常用的几种变量进行介绍。

1．网络类型变量

网络类型表示结构实体（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。

常用的网络类型变量包括 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。

表 3.1 所示为在同等驱动强度下，两个驱动源驱动的 wire 型和 tri 型变量的真值表。

表 3.1　　wire/tri 型变量真值表

wire/tri 型变量双驱动源运算结果				
驱动源 1 / 驱动源 2	0	1	x	z
0	0	x	x	0
1		1	x	1
x	x	x	x	x
z	0	1	x	z

wire 型变量常用来表示用于以 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 型变量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 型变量的声明格式如下：

```
wire [n-1:0] 变量名 1,变量名 2,…,变量名 i;   //共有 i 条总线，每条总线内有 n 条线路
```

也可以如下表示：

```
wire [n:1] 变量名 1,变量名 2,…,变量名 i;    //共有 i 条总线，每条总线内有 n 条线路
```

其中，wire 是 wire 型变量的确认符，[n-1:0] 和 [n:1] 代表该变量的位宽，即该变量有几位，最后跟着的是变量的名字。如果一次定义多个变量，变量名之间用逗号隔开。声明语句的最后要用分号表示语句结束。如下所示：

```
wire  a;                          //定义了一个一位的 wire 型变量
wire [7:0] b;                     //定义了一个八位的 wire 型变量
wire [4:1] c, d;                  //定义了两个四位的 wire 型变量
```

2．寄存器型变量

寄存器是数据储存单元的抽象。寄存器型变量的关键字是 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。reg 类型变量的缺省初始值为不定值，即 x。

reg 型变量常用来表示用于“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。和 wire 型变量类似，reg 型变量的声明格式如下：

```
reg [n-1:0] 变量名 1,变量名 2,…,变量名 i;      //共有 i 条总线，每条总线内有 n 条线路
```

也可以如下表示：

```
reg [n:1]   变量名 1,变量名 2,…,变量名 i;      //共有 i 条总线，每条总线内有 n 条线路
```

其中，reg 是 reg 型变量的确认标识符，[n-1:0] 和 [n:1] 代表该变量的位宽，即该变量有几位（bit），最后跟着的是变量的名字。如果一次定义多个变量，变量名之间用逗号隔开。声明语句的最后要用分号表示语句结束。如下所示：

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

reg 型变量可以赋正值，也可以赋负值。但当一个 reg 型变量是一个表达式中的操作数时，它的值将被当作是无符号值，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。

3．存储器型变量

Verilog HDL 通过对 reg 型变量建立数组来对存储器建模，用于描述 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。由于在 Verilog 语言中没有多维数组存在，因此 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];          //定义一个名为 mema 的 256×8 的存储器
```

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 0～255。需要注意的是，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。 例如：

```
parameter  wordsize=16, memsize=256;                    //定义两个参数
reg [wordsize-1:0] mem[memsize-1:0],writereg, readreg;  //使用可变参数来定义存储器
```

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，如下所示：

```
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，例如：

```
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他的寄存器的值。例如可以用一个加法计数器来做 RAM 的地址索引。

3.3.2 常用运算符

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。

- 算术运算符：（+, - , × ， /,% ）。
- 赋值运算符：（=,<= ）。
- 关系运算符：（>,<,>=,<= ）。
- 逻辑运算符：（ &&,||,! ）。
- 条件运算符：（ ?: ）。
- 位运算符：（ ~ ,|,^,&,^~ ）。
- 移位运算符：（ <<,>> ）。
- 拼接运算符：（ { } ）。
- 其他

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以下 3 种。
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。
二目运算符（binary operator）：可以带两个操作数，操作数放在运算符的两边。
三目运算符（ternary operator）：可以带三个操作数，这三个操作数用三目运算符分隔开。
例如：

```
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。

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。

- +:（加法运算符或正值运算符,如 ega+regb、+3 ）。
- −:（减法运算符或负值运算符，如 rega−3、−3 ）。
- ×:（乘法运算符，如 rega×3 ）。
- /:（除法运算符，如 5/3 ）。
- % :（模运算符或求余运算符，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，例如：

```
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。

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 4 种状态值 1、0、x 和 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 5 种位运算符。

- ~ ：（取反）
- & ：（按位与）
- | ：（按位或）
- ^ ：（按位异或）
- ^~：（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。

- “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。如表 3.2 所示为单目运算符～的运算规则表。

表 3.2 ～ 运算规则表

～运算	
操 作 数	结 果
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
```

- “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，其运算规则如表 3.3 所示。

表 3.3 & 运算规则表

& 运算			
操作数 1 / 操作数 2	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

- “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，其运算规则如表 3.4 所示。

表 3.4 | 运算规则表

| 运算			
操作数 1 / 操作数 2	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

- “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，其运算规则如表 3.5 所示。

表 3.5 ^ 运算规则表

^ 运算			
操作数 1 / 操作数 2	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

- “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，其运算规则如表 3.6 所示。

表 3.6 ^~ 运算规则表

^~ 运算			
操作数 1 / 操作数 2	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

- 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．逻辑运算符

在 Verilog HDL 语言中存在 3 种逻辑运算符。

- &&:（逻辑与）
- ||:（逻辑或）
- ！:（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）。如表 3.7 所示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。

表 3.7 逻辑运算真值表

操 作 数		逻辑运算及结果			
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，“!”的优先级别高于算术运算符，例如。

```
(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。

4．关系运算符

关系运算符共有以下 4 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，例如。

```
a < size-1          //这种表达方式等同于下面一行的表达方式
a < (size-1)
size - (1 < a)      //这种表达方式不等同于下面一行的表达方式
size - 1 < a
```

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。

5．等式运算符

在 Verilog HDL 语言中存在 4 种等式运算符。

- ==:（等于）
- != :（不等于）
- ===:（等于）
- !==:（不等于）

这 4 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。
“===”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 4 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。

表 3.8　　等式运算符真值表

	=== 运算			
操作数 1	0	1	x	z

操作数 2				
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

== 运算				
操作数 1 / 操作数 2	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“==”与“===”的区别。

```
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
```

6．移位运算符

在 Verilog HDL 中有两种移位运算符。

- <<:（左移位运算符）
- >>:（右移位运算符）

其使用方法如下：

```
a >> n;
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，下面举例说明。

```
4'b1001<<1 = 5'b10010;              //左移 1 位后用 0 填补低位
4'b1001<<2 = 6'b100100;             //左移 2 位后用 00 填补低位
1<<6 = 32'b1000000;                 //左移 6 位后用 000000 填补低位
4'b1001>>1 = 4'b0100;               //右移 1 位后，低 1 位丢失，高 1 位用 0 填补
4'b1001>>4 = 4'b0000;               //右移 4 位后，低 4 位丢失，高 4 位用 0 填补
```

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，例如：

```
{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。

位拼接也可以用重复法来简化表达式，如下所示：

```
{4{w}}          //等同于{w,w,w,w}
```

位拼接还可以用嵌套的方式来表达，如下所示：

```
{b,{3{a,b}}}    //等同于{b,a,b,a,b,a,b}
```

用于表示重复的表达式必须是常数表达式，如上例中的 4 和 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与、或、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如下。

（1）先将操作数的第一位与第二位进行与、或、非运算。

（2）将运算结果与第三位进行与、或、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，可参照位运算符的运算规则介绍。

9．优先级别

各种运算符的优先级别关系如表 3.9 所示。

表 3.9　　运算符优先级别表

运　算　符	优 先 级 别
！～	高
* / %	

+ − << >> < <= > >= == != === !== & ^ ^~ | && || ? :	 低

3.4 Verilog HDL 语言的赋值语句和块语句

3.4.1 非阻塞赋值和阻塞赋值

在 Verilog HDL 语言中，信号有两种赋值方式：非阻塞（Non_Blocking）赋值方式和阻塞（Blocking）赋值方式。

（1）非阻塞赋值方式。

```
典型语句: 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）阻塞赋值方式。

```
典型语句: 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。
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 3.5：非阻塞赋值。

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

例 3.5 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。需要注意的是赋值是在“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图 3.1 所示。

例 3.6：阻塞型赋值。

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

例 3.6 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）。综合的电路如图 3.2 所示。

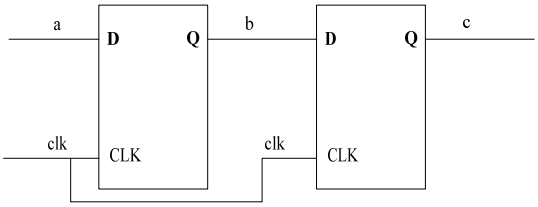


图 3.1　非阻塞赋值综合电路

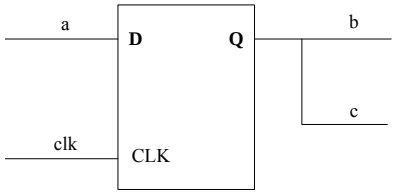


图 3.2　阻塞赋值综合电路

它只用了一个触发器来寄存 a 的值，并同时输出给 b 和 c。这不是设计者的初衷，如果采用例 3.5 所示的非阻塞赋值方式就可以避免这种错误。

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，使其在格式上看更像一条语句。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；另一种是 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。

1．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
```

或者：

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

```
    end
```

其中：

（1）块名即该块的名字，是一个标识符，其作用后面再详细介绍。

（2）块内声明语句可以是参数声明语句，reg 型变量声明语句，integer 型变量声明语句或者 real 型变量声明语句。

下面举例说明。

例 3.7：顺序块。

```
    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，如例 3.8 所示。

例 3.8：加延时顺序块。

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

2．并行块

并行块有以下 4 个特点。

（1）块内语句是同时执行的，即程序流程控制一进入该并行块，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
```

或者：

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

其中：

（1）块名即标识该块的一个名字，相当于一个标识符。

（2）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、ime 型变量声明语句或者事件（event）说明语句。

下面举例说明。

例 3.9：并行块 1。

```
fork
     #50   r = 'h35;      //在绝对时间 50 单位后，r 被赋值
     #100  r = 'hE2;      //在绝对时间 100 单位后（非绝对时间 150），r 再次被赋值
     #150  r = 'h00;
     #200  r = 'hF7;
     #250  -> end_wave;  //在绝对时间 250 单位后，触发事件 end_wave
join
```

在这个例子中用并行块来替代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。

3．块名

在 Verilog HDL 语言中，可以给每一个块取名字，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他语句调用，如被 disable 语句调用。

（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惟一的存储地址，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。需要注意的是，块名和变量名一样，都不能是关键词。Verilog 的关键词参见 3.4.3 小节。

4．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。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，如下所示。

例 3.10：并行块 2。

```
fork
     #250  -> end_wave;      //按下面几条语句顺序执行结果和例［3.10］的执行结果一样
     #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。

3.4.3 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。

在编写 Verilog HDL 程序时，变量名、端口名、块名等的定义不要与这些关键词冲突。

3.5 Verilog HDL 语言的条件语句

3.5.1 if 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 3 种形式的 if 语句。

（1）无分支。

语法形式：

```
if （表达式） 语句;
```

例如：

```
if （a > b） out1 <= int1;    //若 a 大于 b，将 int1 赋予 out1
```

（2）单级分支。

语法形式：

```
if（表达式）语句 1;
else 语句 2;
```

例如：

```
if（a>b） out1<=int1;      //若 a 大于 b，将 int1 赋予 out1；否则，将 int2 赋予 out1
else out1<=int2;
```

（3）多级分支

语法形式：

```
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;       //若 a 大于 b，将 int1 赋予 out1
else  if（a==b）   out1<=int2;       //否则，如果 a 等于 b，将 int2 赋予 out1
```

```
else              out1<=int3;        //否则，将 int3 赋予 out1
```

关于 if 语句有如下 6 点说明。

（1）3 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 或 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。

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，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下所示。

```
if（a>b）begin                    //使用 begin_end 语句实现多个赋值操作
    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，例如：

```
if（expression）  等同于  if（ expression == 1 ）
if（! expression）等同于  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，如下所示：

```
if（index>0）                                    //内嵌 for 语句，无 else 分支
   for（scani=0;scani<index;scani=scani+1）     //内嵌 if 语句
     if（memory［scani］>0） begin              //使用 begin_end 语句，有 else 分支
         $display（"..."）;
```

```
            memory[scani]=0;
        end
        else                                    //此处为内嵌 if 语句的分支
            $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。正确的写法如下：

```
if(index>0) begin                           //内嵌 for 语句，有 else 分支
    for(scani=0;scani<index;scani=scani+1)  //内嵌 if 语句
        if(memory[scani]>0) begin          //使用 begin_end 语句，无 else 分支
            $display("...");
            memory[scani]=0;
      end
end
else                                        //此处为外部 if 语句的分支
    $display("error-indexiszero");
```

（6）if_else 例子。

下面这段程序用 if_else 语句来检测变量 index 以决定 modify-seg1、modify-seg2、modify-seg3 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 10 行定义寄存器和参数。

```
reg [31:0]  instruction, segment_area[255:0];   //定义寄存器
reg [7:0]   index;
reg [5:0]   modify_seg1, modify_seg2, modify_seg3;
parameter                                       //定义参数
     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                        //index<20 时，执行下列操作
     instruction = segment_area[index + modify_seg1];
     index = index + inc_seg1;
end
else  if(index<segment3) begin                  //20<index<64 时，执行下列操作
     instruction = segment_area[index + modify_seg2];
     index = index + inc_seg2;
end
else  if (index<data) begin                     //64<index<128 时，执行下列操作
     instruction = segment_area[index + modify_seg3];
     index = index + inc_seg3;
end
else                                            //index>128 时，执行下列操作
     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>的一般语法格式如下：

```
分支表达式:      语句
缺省项（default 项）:      语句
```

关于 case 语句的几点说明如下。

（1）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。

（2）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。

（3）default 项可有可无，一个 case 语句里只能有一个 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;      //rega 等于 0 时
    16 'd1:  result = 10 'b1011111111;      //rega 等于 1 时
    16 'd2:  result = 10 'b1101111111;      //rega 等于 2 时
    16 'd3:  result = 10 'b1110111111;      //rega 等于 3 时
    16 'd4:  result = 10 'b1111011111;      //rega 等于 4 时
    16 'd5:  result = 10 'b1111101111;      //rega 等于 5 时
    16 'd6:  result = 10 'b1111110111;      //rega 等于 6 时
    16 'd7:  result = 10 'b1111111011;      //rega 等于 7 时
    16 'd8:  result = 10 'b1111111101;      //rega 等于 8 时
    16 'd9:  result = 10 'b1111111110;      //rega 等于 9 时
    default: result = 'bx;                  //rega 不等于上面的值时
endcase
```

（4）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）。

（5）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。

（6）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，因此要详细说明 case 分项的分支表达式的值。

（7）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）。

case 语句与 if 语句的区别主要有以下两点。

（1）与 case 语句中的控制表达式和多分支表达式相比，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 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 case 语句。

例 3.11：case 语句 1。

```
case （select[1:2])
     2 'b00:  result = 0;            //select[1:2]等于00时
     2 'b01:  result = flaga;        //select[1:2]等于01时
     2 'b0x,
     2 'b0z:  result = flaga? 'bx: 0; //select[1:2]等于0x和0z时，执行表达式
     2 'b10:  result = flagb;        //select[1:2]等于10时
     2 'bx0,
     2 'bz0:  result = flagb? 'bx :0; //select[1:2]等于x0和z0时，执行表达式
     default: result = 'bx;          //select[1:2]不等于上面的值时
endcase
```

例 3.12：case 语句 2

```
case（sig）
    1 'bz:    $display（"signal is floating"）; //sig为高阻时，打印输出
    1 'bx:    $display（"signal is unknown"）;  //sig为不定状态时，打印输出
    default:  $display（"signal is %b", sig）;  //为其他时，即0或1时，打印输出
endcase
```

针对电路的特性，Verilog HDL 提供了 case 语句的其他两种形式用来处理不必考虑的情况（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。如表 3.10 所示为 case、casez、casex 的真值表：

表 3.10　　case 语句真值表

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

下面给出两个例子来分别说明 casez 语句和 casex 语句。

例 3.13：casez 语句。

```
reg[7:0] ir;
casez（ir）
    8 'b1???????: instruction1（ir）;          //只判断ir的最高位
    8 'b01??????: instruction2（ir）;          //只判断ir的高2位
    8 'b00010???: instruction3（ir）;          //只判断ir的高5位
```

```
        8 'b000001??: instruction4（ir）;          //只判断 ir 的高 6 位
    endcase
```

例 3.14：casex 语句。

```
    reg［7:0］ r, mask;
    mask = 8'bx0x0x0x0;
    casex（r^mask）                                //判断 r^mask 的结果
        8 'b001100xx: stat1;                       //不考虑低 2 位
        8 'b1100xx00: stat2;                       //不考虑第 3、4 位
        8 'b00xx0011: stat3;                       //不考虑第 5、6 位
        8 'bxx001100: stat4;                       //不考虑高 2 位
    Endcase
```

3.5.3 其他条件语句

上面提到的 if 语句和 case 语句都只能应用于 always 语句内部。如果需要在 always 语句之外应用条件语句，可以采样这样的语法结构：

```
    assign data = (sel)?  a : b;
```

上面的语句的含义相当于：

```
    if （sel = 1）
        data = a;
    else
        data =  b;
```

3.6 Verilog HDL 语言的其他常用语句

3.6.1 循环语句

在 Verilog HDL 中存在着 4 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 3 个步骤来决定语句的循环执行。

① 先给控制循环次数的变量赋初值。

② 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，转到步骤③。

③ 执行一条赋值语句来修正控制循环变量次数的变量的值，然后返回步骤②。

下面将详细地对各种循环语句进行介绍。

1．forever 语句

forever 语句的格式如下：

```
    forever 语句;
```

或者：

```
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 语句不同之处在于它不能独立写在程序中，而必须写在 initial 块中。

2．repeat 语句

repeat 语句的格式如下：

```
repeat（表达式） 语句;
```

或者：

```
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;                    //参数声明
reg [size:1] opa, opb;                           //寄存器声明
reg [longsize:1] result;
begin: mult                                      //为 begin_end 模块定名模块名
    reg [longsize:1] shift_opa, shift_opb;       //寄存器声明
    shift_opa = opa;                             //将 opa、opb 的值赋为 shift_opa、shift_opb
    shift_opb = opb;
    result = 0;
    repeat（size） begin                          //循环次数
        if（shift_opb[1])
        result = result + shift_opa;             //加法操作
        shift_opa = shift_opa <<1;               //左移 1 位
        shift_opb = shift_opb >>1;               //右移 1 位
    end
end
```

3．while 语句

while 语句的格式如下：

```
while（表达式） 语句
```

或者：

```
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 8 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              //当 tempreg 中有不为 0 的位时，循环执行
        if(tempreg[0])  count = count + 1;    //低位为 1 时，计数
        tempreg = tempreg>>1;                 //否则右移 1 位,此时高位用 0 填补
    end
end
```

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，然后执行步骤③；若为假（0），则结束循环，转到步骤⑤。

③ 若表达式为真，在执行指定的语句后，求解表达式 3。

④ 转到步骤②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 3.15 用 for 语句来初始化 memory。例 3.16 则用 for 循环语句来实现前面用 repeat 语句实现的乘法器。

例 3.15：for 语句 1。

```
begin: init_mem
    reg[7:0] tempi;
    for(tempi=0;tempi<memsize;tempi=tempi+1)  //使用 for 循环语句初始化存储器
        memory[tempi]=0;
    end
```

例 3.16：for 语句 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 ) //使用 for 循环语句实现前面的乘法器
        if(opb[bindex])
            result = result + (opa<<(bindex-1)); //加法并移位
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 8 位二进制数中值为 1 的位进行计数的另一种方法，如下所示：

```
begin: count1s
    reg[7:0] tempreg;
    count=0;
    for( tempreg=rega; tempreg; tempreg=tempreg>>1 ) //循环变量增值表达式使用右移操作
            if(tempreg[0])
                count=count+1;
end
```

3.6.2 结构说明语句

Verilog 语言中的任何过程模块都从属于以下 4 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，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，其具体使用方法在第 4 章中详细介绍。这里只对 initial 和 always 语句加以介绍。

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 3.17：initial 语句 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 3.18：initial 语句 2。

```
    initial begin
        inputs = 'b000000;                  //初始时刻为 0
        #10 inputs = 'b011001;              //赋值时刻为 10
        #10 inputs = 'b011011;              //赋值时刻为 20
        #10 inputs = 'b011000;              //赋值时刻为 30
        #10 inputs = 'b001000;              //赋值时刻为 40
    end
```

从这个例子中，我们可以看到 initial 语句的另一个用途，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。

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，例如：

```
    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，例如：

```
    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。

```
    reg［7:0］ counter;
    reg tick;
    always @（posedge areg） begin
        tick = ~tick;                  //tick 反相
        counter = counter + 1;         //计数器递增
    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） begin   //由两个沿触发的 always 块
        …
    end
```

```
always @( a or b or c ) begin                    //由多个电平触发的 always 块
   …
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，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。
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，转换为表示组合逻辑的门级逻辑或带锁存器的组合逻辑。

3.7 Verilog HDL 语言实现组合逻辑电路

数字逻辑电路分为两种，分别是组合逻辑与时序逻辑。
（1）组合逻辑：输出只是当前输入逻辑电平的函数（有延时），与电路的原始状态无关的逻辑电路。也就是说，当输入信号中的任何一个发生变化时，输出都有可能会根据其变化而变化，但与电路目前所处的状态没有任何关系。
其中组合逻辑是由与、或、非门组成的网络。常用的组合电路有：多路器、数据通路开关、加法器、乘法器等。
（2）时序逻辑：输出不只是当前输入的逻辑电平的函数，还与电路目前所处的状态有关。
时序逻辑由多个触发器和多个组合逻辑块组成的网络，常用的有：计数器、复杂的数据流动控制逻辑、运算控制逻辑、指令分析和操作控制逻辑等。同步时序逻辑是设计复杂的数字逻辑系统的核心。时序逻辑借助于状态寄存器记住它目前所处的状态。在不同的状态下，即使所有的输入都相同，其输出也不一定相同。

3.7.1 assign 语句实现组合逻辑

组合逻辑电路可以用 assign 语句实现，例如：
例 3.19：assign 加法器。

```
wire a,b,c;
assign c = a + b;              //加法器
```

例 3.19 实现的是一个简单的加法器，assign 语句也可以实现复杂一些的组合逻辑电路，例如：
例 3.20：assign 选择器。

```
wire a,b,c;
wire ena;
assign c = ena ? a : b;     //数据选择器
```

例 3.20 实现的是一个数据选择器。如果组合逻辑比较复杂，用 assign 语句书写就会比较繁琐，可读性较差。例如用 assign 语句实现一个 8 选 1 数据选择器，如下所示：
例 3.21：assign 8 选 1 选择器。

```
wire a0,a1,a2,a3,a4,a5,a6,a7,b;
wire [2:0] addr;
assign b =                    //8 选 1 数据选择器
      (addr == 3'd0) ? a0 :(addr == 3'd1) ? a1 :
      (addr == 3'd2) ? a2 :(addr == 3'd3) ? a3 :
      (addr == 3'd4) ? a4 :(addr == 3'd5) ? a5 :
      (addr == 3'd6) ? a6 : a7;
               //在该表达式中，当 addr 不等于 d0～d6 时，b 等于 a7
```

```
                    //当 addr 等于 d6 时，b 等于 a6；当 addr 等于 d5 时，b 等于 a5，且优先级
                    //高于 addr 等于 d6 时的情况，依次类推
```

所以复杂的组合逻辑电路最好用 always 块实现。
从上面的几个例子可以看出，使用 assign 语句描述组合逻辑电路时，格式为：

```
    assign 输出变量 = 输入变量之间的运算结果;
```

3.7.2 always 块实现组合逻辑

组合逻辑电路也可以用 assign 语句实现，例如：
例 3.22：always 加法器。

```
    wire a,b,c;
    always @ (a or b)       //当 a 和 b 有变化时，触发加法器操作
         c = a + b;
```

上面这个例子实现了一个加法器，如果需要实现一个数据选择器，可以书写如下：
例 3.23：always 选择器。

```
    wire a,b,c;
    wire ena;
    always @ (a or b or ena)       //当 a、b 和 ena 有变化时，进行下列操作
        if(ena == 1'b0)      c = b;
        else   c = a;
```

如果想实现一个比较复杂的组合逻辑电路，例如：
例 3.24：always8 选 1 选择器。

```
    wire a0,a1,a2,a3,a4,a5,a6,a7,b;
    wire [2:0] addr;
    always @ (a0 or a1 or a2 or a3 or a4 or a5 or a6 or a7 or addr) begin
         case(addr)            //使用 case 语句实现 8 选 1 数据选择器
              3'd0: b = a0;    //只有当 a0~a7 以及 addr 有变化时，才触发 case 的操作
              3'd1: b = a1;
              3'd2: b = a2;
              3'd3: b = a3;
              3'd4: b = a4;
              3'd5: b = a5;
              3'd6: b = a6;
              3'd7: b = a7;
         endcase
    end
```

由于在 always 块中可以使用 if、case 等语句，所以对于复杂的组合逻辑，使用 always 语句进行描述显得层次更加清楚，可读性更强。
从上面几个例子可以看出，使用 always 语句描述组合逻辑电路时，格式为：

```
    always @ (敏感变量 1 or 敏感变量 2 or 敏感变量 3 or …) begin
         各种语句的组合
    end
```

其中的敏感变量包括所有的会引起输出变化的输入变量以及相应的控制变量。另外，使用 always 语句描述组合逻辑电路时，应该使用阻塞赋值方式，即“=”，而不是“<=”。

3.8 Verilog HDL 语言实现时序逻辑电路

在 Verilog HDL 语言中，时序逻辑电路使用 always 语句块来实现。例如，实现一个带有异步复位信号的 D 触发器如下。

例 3.25：带异步复位的 D 触发器 1。

```
wire Din;
wire clock,rst;
reg Dout;
always @ (posedge clock or negedge rst)        //带有异步复位
    if(rst == 1'b0)    Dout <= 1'b0;
    else Dout   <= Din;                        //D 触发器数据输出
```

在例 3.25 中，每当时钟 clock 上升沿到来后，输出信号 Dout 的值便更新为输入信号 Din 的值。当复位信号下降沿到来时，Dout 的值就会变成 0。

必须注意的是，在时序逻辑电路中，通常使用非阻塞赋值，即使用“<=”。当 always 块整个完成之后，值才会更新，例如：

例 3.26：带异步复位的 D 触发器 2。

```
wire Din;
wire clock,rst;
reg Dout;
always @ (posedge clock or negedge rst)    //带有异步复位
     if(rst == 1'b0)    out <= 1'b0;
     else begin
          Dout <= Din;                     //D 触发器输出值还处于锁定状态
          Dout <= 1'b1;                    //D 触发器输出值依然处于锁定状态
     End                                   //D 触发器的输出为 1
```

在例 3.26 中，Dout 首先被赋值为 Din，此时 Dout 的值并没有发生改变；接着 Dout 又被赋值为 1，此时 Dout 的值依然没发生改变；直到这个 always 模块完成，Dout 的值才变成最后被赋的值，此例中 Dout 的值为 1。

在时序逻辑电路中，always 的时间控制是沿触发的，可以单个信号也可以多个信号，中间需要用关键字“or”连接，例如：

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

其中有一个时钟信号和一个异步复位信号。

```
always @(posedge clock1 or posedge clock2 or posedge reset) begin
//由 3 个沿触发的 always 块
   …
end
```

其中有两个时钟信号和一个异步复位信号。

一般而言，同步时序逻辑电路更稳定，所以建议尽量使用一个时钟触发。

3.9 Verilog HDL 语言与 C 语言的区别与联系

数字电路设计工程师一般都学习过编程语言、数字逻辑基础、各种 EDA 软件工具的使用。就编程语言而言，国内外大多数学校都以 C 语言为标准，只有少部分学校使用 Pascal 和 Fortran。
算法的描述和验证常用 C 语言来做。例如要设计 Reed-Solomen 编码/解码器，可以分为下面几个步骤。

- 先深入了解 Reed-Solomen 编码/解码的算法。
- 编写 C 语言的程序来验证算法的正确性。
- 运行描述编码器的 C 语言程序，把在数据文件中的多组待编码的数据转换为相应的编码后数据，并存入文件。
- 编写一个加干扰用的 C 语言程序，用于模拟信道。它能产生随机误码位（并把误码位个数控制在纠错能力范围内），将其加入编码后的数据文件中。运行该加干扰程序，产生带误码位的编码后的数据文件。
- 编写一个解码器的 C 语言程序，运行该程序把带误码位的编码文件解码为另一个数据文件。

比较原始数据文件和生成的文件，便可知道编码和解码的程序是否正确（能否自动纠正纠错能力范围内的错码位），用这种方法我们就可以来验证算法的正确性。但这样的数据处理其运行速度只与程序的大小和计算机的运行速度有关，也不能独立于计算机而存在。
如果要设计一个专门的电路来进行这种对速度有要求的实时数据处理，除了以上介绍的 C 程序外，还需编写硬件描述语言（如 Verilog HDL 或 VHDL）的程序。然后进行仿真以便从电路结构上保证算法能在规定的时间内完成，并能与前端和后端的设备或器件正确无误地交换数据。
用硬件描述语言（HDL）的程序设计硬件的好处在于易于理解、易于维护，调试电路速度快，有许多的易于掌握的仿真、综合和布局布线工具，还可以用 C 语言配合 HDL 来做逻辑设计的前后仿真，验证功能是否正确。
在算法硬件电路的研制过程中，计算电路的结构和芯片的工艺对运行速度有很大的影响。所以在电路结构确定之前，必须经过多次仿真。

- C 语言的功能仿真。
- C 语言的并行结构仿真。
- Verilog HDL 的行为仿真。
- Verilog HDL RTL 级仿真。
- 综合后门级结构仿真。
- 布局布线后仿真。
- 电路实现验证。

下面介绍用 C 语言配合 Verilog HDL 来设计算法的硬件电路块时考虑的三个主要问题：

1．为什么选择 C 语言与 Verilog 配合使用

首先，C 语言很灵活，查错功能强，还可以通过 PLI（编程语言接口）编写自己的系统任务直接与硬件仿真器（如 Verilog-XL）结合使用。C 语言是目前世界上应用最为广泛的一种编程语言，因而 C 程序的设计环境比 Verilog HDL 更完整。此外，C 语言可应用于许多领域，有可靠的编译环境，语法完备，缺陷较少。比较起来，Verilog 语言只是针对硬件描述的，在别处使用（如用于算法表达等）并不方便。而且 Verilog 的仿真、综合、查错工具等大部分软件都是商业软件，与 C 语言相比缺乏长期大量的使用，可靠性较差，亦有很多缺陷。所以，只有在 C 语言的配合使用下，Verilog 才能更好地发挥作用。
面对上述问题，最好的方法是 C 语言与 Verilog 语言相辅相成，互相配合使用。这就是既要利用 C 语言的完整性，又要结合 Verilog 对硬件描述的精确性，来更快、更好地设计出符合性能要求的硬件电路系统。
利用 C 语言完善的查错和编译环境，设计者可以先设计出一个功能正确的设计单元，以此作为设计比较的标准。然后，把 C 程序一段一段地改写成用并型结构（类似于 Verilog）描述的 C 程序，此时还是在 C 的环境里，使用的依然是 C 语言。

如果运行结果都正确，就将 C 语言关键字用 Verilog 相应的关键字替换，进入 Verilog 的环境。将测试输入同时加到 C 与 Verilog 两个单元，将其输出做比较。这样很容易发现问题的所在，更正后再做测试，直至正确无误。

2．C 语言与 Verilog 语言互相转换中存在的问题

混合语言设计流程往往会在两种语言的转换中会遇到许多难题，如下所示。

- 怎样把 C 程序转换成类似 Verilog 结构的 C 程序。
- 如何增加并行度，以保证用硬件实现时运行速度达到设计要求。
- 怎样不使用 C 中较抽象的语法（例如迭代、指针、不确定次数的循环等）。也能来表示算法（因为转换的目的是用可综合的 Verilog 语句来代替 C 程序中的语句，而可用于综合的 Verilog 语法是相当有限的，往往找不到相应的关键字来替换）。

C 程序是按行依次执行的，属于顺序结构。而 Verilog 描述的硬件是可以在同一时间同时运行的，属于并行结构。这两者之间有很大的冲突。另外，Verilog 的仿真软件也是顺序执行的，在时间关系上同实际的硬件是有差异的，可能会出现一些无法发现的问题。

C 语言的函数调用与 Verilog 中模块的调用也有区别。C 程序调用函数是没有延时特性的，一个函数是惟一确定的，对同一个函数的不同调用是一样的。而 Verilog 中对模块的不同调用是不同的，即使调用的是同一个模块，必须用不同的名字来指定。

Verilog 的语法规则很死，限制很多，能用的判断语句有限，仿真速度较慢，查错功能差，错误信息不完整。仿真软件通常也很昂贵，而且不一定可靠。C 语言的花样则很多，转换过程中会遇到一些困难。

C 语言没有时间关系，转换后的 Verilog 程序必须要能做到没有任何外加的人工延时信号，否则将无法使用综合工具把 Verilog 源代码转化为门级逻辑。

3．如何利用 C 语言来加快硬件的设计和查错

如表 3.11 所示为常用的 C 语言与 Verilog 相对应的关键字与控制结构。

表 3.11　　C 语言与 Verilog 相对应的关键字与控制结构表

C	Verilog
sub-function	module、function、task
if-then-else	if-then-else
case	case
{,}	begin、end
for	for
while	while
break	disable
define	define
int	int
printf	monitor、display、strobe

如表 3.12 所示为 C 语言与 Verilog 相对应的运算符。

表 3.12　　C 语言与 Verilog 对应运算符表

C	Verilog	功　能
*	*	乘
/	/	除

+	+	加
-	-	减
%	%	取模
!	!	反逻辑
&&	&&	逻辑与
||	||	逻辑或
>	>	大于
<	<	小于

续表

C	Verilog	功　能
>=	>=	大于等于
<=	<=	小于等于
==	==	等于
!=	!=	不等于
~	~	位反相
&	&	按位逻辑与
|	|	按位逻辑或
^	^	按位逻辑异或
~^	~^	按位逻辑同或
>>	>>	右移
<<	<<	左移
?:	?:	相当于 if-else

从上面的讲解我们可以总结如下。

- C 语言与 Verilog 硬件描述语言可以配合使用，辅助设计硬件。
- C 语言与 Verilog 硬件描述语言很像，但要稍加限制。
- C 语言的程序很容易转成 Verilog 的程序。

3.10　Verilog HDL 程序设计经验

对于 Verilog HDL 的初学者，经常会对语法中的几个容易混淆的地方产生困惑。下面列出几个常见问题和解决它们的小窍门。

1．“=”和“<=”的区分方法

前面的内容已经从原理上解释了阻塞（=）和非阻塞（<=）赋值的区别，但对于初学者来说，在实际应用过程中还会产生一些困惑。下面的方法可以帮助初学者来弄清楚两种赋值符号的应用场合。

在 always 语句中，所有的赋值符号用非阻塞的，即“<=”；　在 always 语句外，所有的赋值符号用阻塞的，即“=”。

2．“reg”和“wire”的区分方法

reg 类型和 wire 类型是 Verilog HDL 语法中两种最常用的变量。在对 module 定义的端口信号进行类型描述的时候，初学者会对何时需要指定为 reg 型感到困惑。可以参考下面的方法。

（1）如果这个信号需要在 always 块里面被赋值，那么必须指定为 reg 类型的。

（2）如果这个信号需要在 always 块外面被赋值，那么必须指定为 wire 类型的。如果这个信号是端口信号，那么没默认的类型就是 wire 类型的，不需要另外指定。

3.11 典型实例 3：数字跑表

3.11.1 实例的内容及目标

1．实例的主要内容

本节通过 Verilog HDL 语言编写一个具有“百分秒、秒、分”计时功能的数字跑表，可以实现一个小时以内精确至百分之一秒的计时。

数字跑表的显示可以通过编写数码管显示程序来实现，本实例只给出数字跑表的实现过程。读者还可以通过增加小时的计时功能，实现完整的跑表功能。

2．实例目标

本实例主要实现了计数及进位的设计，通过几个 always 模块的设计实现一个特定用途的模块——数字跑表。通过本实例，读者应达到下面的一些实例目标。

- 初步掌握 Verilog 语言的设计方法。
- 完成一个数字跑表的设计。

3.11.2 原理简介

本数字跑表首先要从最低位的百分秒计数器开始，按照系统时钟进行计数。计数至 100 后向秒计数器进位，秒计数器以百分秒计数器的进位位为时钟进行计数。计数至 60 后向分计数器进位，分计数器以秒计数器的进位位为时钟进行计数，读者可以自行增加小时计数器。

数字跑表巧妙地运用进位位作为计数时钟来减少计数的位数。如果统一使用系统时钟作为计数时钟，那秒计数器将是一个 6000 进制的计数器，而分计数器将是一个 3600000 进制的计数器。这样将极大的浪费 FPGA 的逻辑资源。而使用进位位作为计数时钟，只需要一个 100 进制的计数器和两个 60 进制的计数器。

如图 3.3 是本实例的数字跑表模块图。

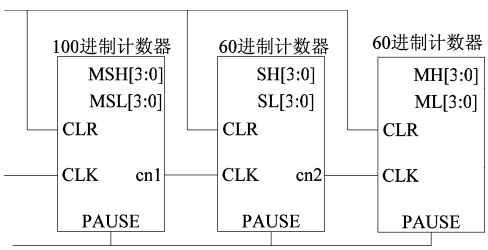


图 3.3 数字跑表模块图

在实际的设计中，为了使计数器更加简单，计数器使用高低位两个计数器实现。100 进制计数器分别是高位 10 进制计数器，低位 10 进制计数器；60 进制计数分别是高位 6 进制计数器，低位 10 进制计数器。这样整个数字跑表使用 6 个计数器实现。

同时由于 10 进制计数器重复使用了 5 次，可以使用独立的模块实现 10 进制计数器，这样就可以通过模块复用来节省整个模块使用的资源。

数字跑表提供了清零位 CLR 和暂停位 PAUSE，百分秒的时钟信号可以通过系统时钟分频提供。分频至 1/100s，即可实现真实的时间计数。详细的时钟分频设计读者可参考相关的资料实现，在本实例中不再提供。

3.11.3 代码分析

下面给出这个数字跑表的源代码，读者可以将这些源代码嵌入自己的工程设计中，来实现数字跑表的功能。首先给出代码中端口信号的定义，读者可根据这些端口与自己的工程设计进行连接。

- CLK：时钟信号。
- CLR：异步复位信号。
- PAUSE：暂停信号。
- MSH、MSL：百分秒的高位和低位。
- SH、SL：秒信号的高位和低位。
- MH、ML：分钟信号的高位和低位。

下面是数字跑表的 Verilog HDL 源代码及说明。

```
module paobiao(CLK,CLR,PAUSE,MSH,MSL,SH,SL,MH,ML);
     //端口说明
     input CLK,CLR;
     input PAUSE;
     output[3:0] MSH,MSL,SH,SL,MH,ML;
     //内部信号说明
     reg[3:0] MSH,MSL,SH,SL,MH,ML;
     reg cn1,cn2;                    //cn1为百分秒向秒的进位，cn2为秒向分的进位

     //百分秒计数模块，每计满100，cn1产生一个进位
     always @(posedge CLK or posedge CLR) begin
         if(CLR) begin               //异步复位
           {MSH,MSL}<=8'h00;
            cn1<=0;
          end
          else if(!PAUSE) begin      //PAUSE为0时正常计数，为1时暂停计数
               if(MSL==9) begin
                    MSL<=0;          //低位计数至10时，低位归零
                    if(MSH==9) begin
                         MSH<=0;     //低、高位计数至10时，高位归零
                         cn1<=1;     //低、高位计数至10时，触发进位位
                    end
                    else             //低位计数至10，高位计数未至10时，高位计数
                         MSH<=MSH+1;
               end
               else begin
                    MSL<=MSL+1;      //低位计数未至10时，低位计数
                    cn1<=0;          //低位计数未至10时，不触发进位位
               end
          end
     end

     //秒计数模块，每计满60，cn2产生一个进位
     always @(posedge cn1 or posedge CLR) begin
          if(CLR) begin              //异步复位
```

```
                {SH,SL}<=8'h00;
                cn2<=0;
            end
            else if(SL==9) begin
                SL<=0;              //低位计数至 10 时，低位归零
                if(SH==5) begin
                    SH<=0;          //低位计数至 10，高位计数至 6 时，高位归零
                    cn2<=1;         //低位计数至 10，高位计数至 6 时，触发进位位
                end
                else
                    SH<=SH+1;       //低位计数至 10，高位计未数至 6 时，高位计数
            end
            else begin
                SL<=SL+1;           //低位计数未至 10 时，低位计数
                cn2<=0;             //低位计数未至 10 时，不触发进位位
            end
        end

        //分钟计数模块，每计满 60，系统自动清零
        always @(posedge cn2 or posedge CLR) begin
            if(CLR) begin           //异步复位
                {MH,ML}<=8'h00;
            end
            else if(ML==9) begin
                ML<=0;              //低位计数至 10 时，低位归零
                if(MH==5)
                    MH<=0;          //低位计数至 10，高位计数至 6 时，高位归零
                else
                    MH<=MH+1;       //低位计数至 10，高位计未数至 6 时，高位计数
            end
            else
                ML<=ML+1;           //低位计数未至 10 时，低位计数
            end

    endmodule
```

通过上面的这 3 个模块，即可实现数字跑表的功能。

3.11.4 参考设计

本实例相关参考设计文件在本书实例代码的“典型实例 3”文件夹。

3.12 典型实例 4：PS/2 接口控制

3.12.1 实例的内容及目标

1．实例的主要内容

本实例通过 Verilog 编程实现在红色飓风 II 代 Xilinx 开发板上面实现对键盘、LCD、RS-232 等接口或者器件进行控制，将有键盘输入的数据在 LCD 上面显示出来，或者通过 RS-232 在 PC 机上的超级终端上显示出来。

2．实例目标

通过本实例，读者应达到下面的目标。

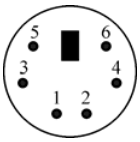
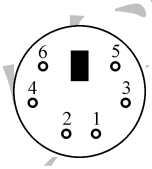
- 了解 PS/2 接口协议。
- 掌握键盘的工作原理。
- 编写 Verilog 程序实现通过开发板上 PS/2 接口读取键盘的输入信息。

3.12.2 原理简介

PS/2 键盘履行一种双向同步串行协议。换句话说每次数据线上发送一位数据，并且每在时钟线上发一个脉冲就被读入，键盘可以发送数据到主机，而主机也可以发送数据到设备。但主机总是在总线上有优先权，它可以在任何时候抑制来自于键盘的通信，只要把时钟拉低即可。

本实例要编写一个能实现 PS/2 端口功能的 Verilog 程序。首先我们要了解 PS/2 端口的结构与管脚功能定义，如表 3.13 所示。

表 3.13　　PS/2 端口结构及管脚定义

端口结构		管脚定义	
		1	数据
		2	未实现，保留
		3	电源地
		4	电源，+5V
		5	时钟
插头	插座	6	未实现，保留

可以看到，PS/2 里面只有一个数据口，若要分辨很多按键就需要一个高效率的分辨方法。键盘的处理器花费很多的时间来扫描或监视按键矩阵。如果它发现有键被按下，释放或按住键盘，将发送扫描码的信息包到计算机。

扫描码有两种不同的类型：“通码”和“断码”。当一个键被按下或按住就发送“通码”，当一个键被释放就发送“断码”。每个按键被分配了惟一的“通码”和“断码”。这样，主机通过查找惟一的扫描码就可以测定是哪个按键。

每个键一整套的通断码组成了扫描码集，图 3.4 中包含了键盘上面大部分按键的扫描码。

当按键被释放以后，键盘回在扫描码前面加上一个“F0”作为按键松开信号。同时有的按键是 Extended（扩展）键，此时要在它们的扫描码前面加上一个“E0”作为开头，这种按键松开以后将在扫描码前面附加上“E0F0”。

下面我们来了解信号是如何从键盘输入通过 PS/2 端口的数据线输入的。首先键盘要检测数据线和时钟线是否都为高，只有它们都处在高的状态才可以写数据。从键盘发送到主机的数据在时钟信号的下降沿（当时钟从高变到低）的时候被读取。

键盘主要使用一种每帧包含 11 位的串行协议：第一位是起始位，始终为“0”；接下来是 8 位数据位，排列顺序是由低到高；再后面是奇偶校验位；最后是结束位，始终为“1”。如图 3.5 所示为该协议的时序图。

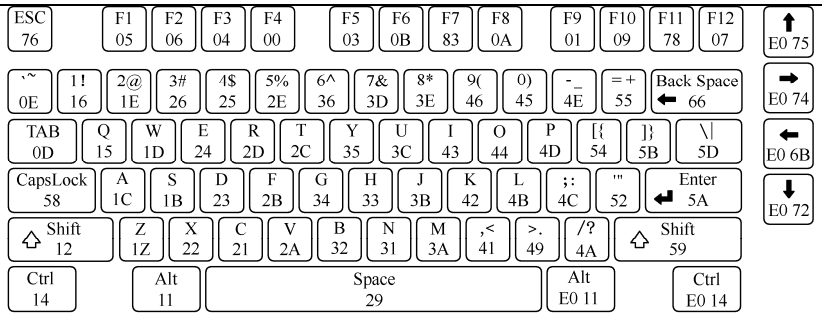


图 3.4　键盘扫描码

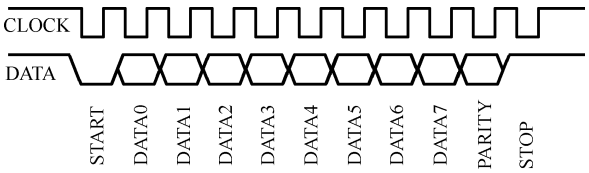


图 3.5　键盘串行协议

3.12.3　实例详解

本实例的具体的实验步骤可参见 2.6 节，在此不再详述，仅给出主要的操作流程。

（1）启动 ISE 软件。

（2）创建新工程。

（3）编写键盘串行协议。

按照图 3.5 的时序编写相应的 Verilog 代码，具体源代码参见实例代码。

（4）添加设计输入。

将编写好的接口协议加载至工程中，完成接口控制与系统的对接。

（5）设置器件及管脚约束。

按照开发板的说明进行相关的设置。

（6）下载验证。

下载程序后，将键盘接至开发板上，通过开发板的 LCD 可以看到键盘输入的字符。

3.12.4　参考设计

本实例相关参考设计文件在本书实例代码的典型实例 4 文件夹。

3.13　典型实例 5：交通灯控制器

3.13.1　实例的内容及目标

1．实例的主要训练内容

本实例通过 Verilog HDL 语言设计一个简易的交通等控制器，实现一个具有两个方向、共 8 个灯并具有时间倒计时功能的交通灯功能。

2．实例目标

通过本实例，读者应达到下面的目标。

- 掌握 Verilog 设计一个交通等控制器的方法。
- 初步掌握 Verilog 语言的设计方法。

3.13.2 原理简介

交通灯是城市交通中不可缺少的重要工具，是城市交通秩序的重要保障。本实例就是实现一个常见的十字路口交通灯功能。读者通过学习这个交通灯控制器，可以实现一个更加完整的交通灯。例如实现实时配置各种灯的时间，手动控制各个灯的状态等。

一个十字路口的交通一般分为两个方向，每个方向具有红灯、绿灯和黄灯 3 种，另外每个方向还具有左转灯，因此每个方向具有 4 个灯。

这个交通灯还为每一个灯的状态设计了倒计时数码管显示功能。可以为每一个灯的状态设置一个初始值，灯状态改变后，开始按照这个初始值倒计时。倒计时归零后，灯的状态将会改变至下一个状态。

值得注意的是，交通灯两个方向的灯的状态是相关的。也就是说，每个方向的灯的状态影响着另外一个方向的灯的状态，这样才能够协调两个方向的车流。如果每个方向的灯是独立变化的，那么交通灯就没有了意义。

如表 3.14 所示是两个方向（假设为 A，B 方向）灯的状态的对应情况。

表 3.14 交通灯两个方向灯状态对应表

方向 A	方向 B
红灯亮	黄灯亮或绿灯亮
直行绿灯亮	红灯亮
黄灯亮	红灯亮
左转灯	红灯亮

在实际的交通系统中，直行绿灯、左转绿灯和红灯的变化之间都应该有黄灯作为缓冲，以保证交通的安全。因此假如我们假设方向 A 的黄灯亮的时间持续 5s，直行绿灯灯亮的时间持续 40s，左转灯灯亮的时间持续 15s，则方向 B 红灯灯亮的时间持续为（直行绿灯+黄灯+左转绿灯+黄灯）所消耗的时间，一共为 65s。

同样假设方向 B 黄灯亮的时间持续 5s，直行绿灯灯亮的时间持续 30s，左转灯灯亮的时间持续 15s，则方向 B 红灯灯亮的时间持续为（直行绿灯+黄灯+左转绿灯+黄灯）所消耗的时间，一共为 55s。

具体时间参数的设定读者可以根据需要进行修改，但是一定要保证两个方向的灯的状态符合表 3.14 的要求。

3.13.3 代码分析

下面给出交通灯控制器的 Verilog HDL 源代码，首先介绍交通灯端口信号的定义及说明，读者可以通过这些端口将此交通灯模块实例化至自己的工程设计中。

- CLK：同步时钟。
- EN：使能信号，为高电平时，控制器开始工作。
- LAMPA：控制 A 方向 4 盏灯的状态；其中，LAMPA0～LAMPA3 分别控制 A 方向的左拐灯、绿灯、黄灯和红灯。
- LAMPB：控制 B 方向 4 盏灯的状态；其中，LAMPB0～LAMPB3 分别控制 B 方向的左拐灯、绿灯、黄灯和红灯。
- ACOUNT：用于 A 方向灯的时间显示，8 位，可驱动两个数码管。
- BCOUNT：用于 B 方向灯的时间显示，8 位，可驱动两个数码管。

下面是交通灯的 Verilog HDL 源代码及说明。

```
module traffic(CLK,EN,LAMPA,LAMPB,ACOUNT,BCOUNT);
    //端口说明
    output［7:0］ ACOUNT,BCOUNT;
    output［3:0］ LAMPA,LAMPB;
    input CLK,EN;
    //内部信号说明
    reg［7:0］ numa,numb;          //ACOUNT 和 BCOUNT 的内部信号
    reg tempa,tempb;
    reg［2:0］ counta,countb;      //方向 A 和方向 B 的灯的状态
    reg［7:0］ ared,ayellow,agreen,aleft,bred,byellow,bgreen,bleft;
    reg［3:0］ LAMPA,LAMPB;

    //设置各交通灯的持续时间初始化值，红灯的值由另一个方向的黄灯和绿灯计算得出。
    always @(EN)
        if(!EN) begin              //使能信号 EN 无效时，对交通灯的计数值进行初始化
            ared       <=8'd55;    //55 s , 30 + 5 + 15 + 5
            ayellow    <=8'd5;     //5 s
            agreen     <=8'd40;    //40 s
            aleft      <=8'd15;    //15 s
            bred       <=8'd65;    //65 s , 40 + 5 + 15 + 5
            byellow    <=8'd5;     //5 s
            bleft      <=8'd15;    //15 s
            bgreen     <=8'd30;    //30 s
        end
        assign ACOUNT=numa;                  //8 位数码管输出
    assign BCOUNT=numb;               //8 位数码管输出
        //控制 A 方向 4 种灯的模块
    always @(posedge CLK) begin
        if(EN) begin                  //使能有效时，交通灯开始工作
            if(!tempa) begin
                tempa<=1;
                case(counta)          //控制灯状态的顺序
                    0: begin          //状态 0
                        numa<=agreen; //直行绿灯亮
                        LAMPA<=2;         //输出 0010
                        counta<=1;        //进入下一个状态
                    end
                    1: begin              //状态 1
                        numa<=ayellow;    //黄灯亮
                        LAMPA<=4;         //输出 0100
                        counta<=2;        //进入下一个状态
                    end
                    2: begin              //状态 2
                        numa<=aleft;      //左转绿灯亮
                        LAMPA<=1;         //输出 0001
                        counta<=3;        //进入下一个状态
```

```
                end
                3: begin                //状态 3
                    numa<=ayellow;      //黄灯亮
                    LAMPA<=4;           //输出 0100
                    counta<=4;          //进入下一个状态
                end
                4: begin                //状态 4
                    numa<=ared;         //红灯亮
                    LAMPA<=8;           //输出 1000
                    counta<=0;          //进入下一个状态（状态 0）
                end
                default:                //默认状态
                    LAMPA<=8;           //红灯亮，输出 1000
            endcase
        end
        else begin                  //每一个状态的倒计时
           if(numa>1)               //判断倒计时未归零时分别对高地位进行递减
                if(numa[3:0]==0) begin
                    numa[3:0]<=4'b1001;
                    numa[7:4]<=numa[7:4]-1;
                end
                else
                    numa[3:0]<=numa[3:0]-1;
            if (numa==2)
                tempa<=0;   //倒计时结束，返回灯状态变化判断，将进入下一个状态
        end
    end
    else begin
        LAMPA<=4'b1000;         //使能无效时，红灯亮
        counta<=0;              //返回方向 A 的状态 0(绿灯状态)
        tempa<=0;               //进入状态变化判断
    end
end
   //控制 B 方向 4 种灯的模块，模块的语言描述与方向 A 的描述基本一致，这里不再重复注释
always @(posedge CLK) begin
    if (EN) begin
        if(!tempb) begin
            tempb<=1;
            case (countb)
                0: begin
                    numb<=bred;
                    LAMPB<=8;
                    countb<=1;
                end
                1: begin
                    numb<=bgreen;
                    LAMPB<=2;
```

```
                countb<=2;
            end
            2: begin
                numb<=byellow;
                LAMPB<=4;
                countb<=3;
            end
            3: begin
                numb<=bleft;
                LAMPB<=1;
                countb<=4;
            end
            4: begin
                numb<=byellow;
                LAMPB<=4;
                countb<=0;
            end
            default:
                LAMPB<=8;
        endcase
    end
    else begin //倒计时
        if(numb>1)
            if(!numb[3:0]) begin
                numb[3:0]<=9;
                numb[7:4]<=numb[7:4]-1;
            end
        else
            numb[3:0]<=numb[3:0]-1;
        if(numb==2)
            tempb<=0;
    end
  end
  else begin
    LAMPB<=4'b1000;
    tempb<=0;
    countb<=0;
  end
 end
endmodule
```

通过上面这个 Verilog HDL 模块，基本实现了交通灯控制器的基本功能。读者可将此设计应用于实际的硬件系统中，通过晶振、FPGA、开关、LED 灯及数码管等资源即可完成硬件实现。

3.13.4 参考设计

本实例相关参考设计文件参见实例代码的“典型实例 5”文件夹。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路银海大厦 A 座 8 层，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218