# 第3章 Verilog HDL初步

## 3.1 硬件描述语言简介

### 3.1.1 概述

硬件描述语言是一种用形式化方法来描述数字电路和系统的语言。数字电路系统的设计者利用 HDL 可以从顶层到底层(从抽象到具体)逐层描述自己的设计思想,用一系列分层次的模块来表示复杂的数字系统。逻辑设计完成后,利用 EDA 工具逐层进行仿真验证,电路功能验证完成后,将设计工程中需要变为具体物理电路的模块组合经由自动综合工具转换到门级电路网表,接下来再用 ASIC 或 FPGA 自动布局布线工具把网表转换为具体电路布线结构。基于 HDL 的开发流程如图 3-1 所示。在写入物理器件之前,还可以用 Verilog 的门级模型(系统基本元件或用户自定义元件(User-Defined Primitives,UDP))来代替具体基本元件。因为其逻辑功能和时延特性与真实的物理元件完全一致,所以在仿真工具的支持下能验证复杂数字系统物理结构的正确性,使投片的成功率达到 100%。目前,这种称为高层次设计的方法已被广泛采用。据统计,目前在美国硅谷有 90%以上的 ASIC 和 FPGA 已采用硬件描述语言方法进行设计。

图 3-1 基于 HDL 的开发流程

硬件描述语言已成功应用于 FPGA 设计的各个阶段,包括建模、仿真、验证和综合等。到 20 世纪 80 年代,已出现了上百种硬件描述语言,并对 EDA 设计起到了极大的促进和推动作用。但是,这些语言一般各自面向特定的设计领域与层次,而且众多的语言使用户无所适从,因此急需一种面向设计的多领域、多层次并得到普遍认同的标准硬件描述语言。进入 20 世纪 80 年代后期,硬件描述语言朝着标准化的方向发展。最终,VHDL 和 Verilog HDL 适应了这种趋势的要求,先后成为 IEEE 标准。把硬件描述语言用于自动综合只有 10 多年的历史。用综合工具把可综合的 HDL 模块自动转换为具体电路的功能研究发展非常迅速,大大地提高了复杂数字系统的设计生产率。

VHDL 和 Verilog HDL 的功能都很强大,在一般的应用设计中设计者使用任何一种语言都可以完成自己的任务。接下来将分别对两种语言进行介绍。

### 3.1.2 Verilog HDL

Verilog HDL 是一种硬件描述语言,用于从算法级、门级到开关级的多种抽象设计层次的数字系统建模。建模的数字系统对象的复杂性介于简单的门和完整的电子数字系统之间。数字系统能够按层次描述,并可在相同描述中显式地进行时序建模。

Verilog HDL 的描述能力包括设计的行为特性、设计的数据流特性、设计的结构组成以及包含响应监视与设计验证方面的时延和波形产生机制，这些都使用同一种建模语言。此外，Verilog HDL 提供了编程语言接口，通过该接口可以在仿真、验证期间从设计外部访问设计，包括仿真的具体控制和运行。

Verilog HDL 不仅定义了语法，而且对每个语法结构都定义了清晰的仿真语义。因此，用这种语言编写的模型能够使用 Verilog 仿真器进行验证。Verilog HDL 从 C 语言中继承了多种操作符和结构。此外，Verilog HDL 还提供了扩展的建模能力，其中许多扩展最初很难理解。但是，Verilog HDL 的核心子集非常易于学习和使用，这对大多数建模应用来说完全能够满足需要。

1. Verilog HDL 的发展

Verilog HDL 最初是在 1983 年由 Gateway Design Automation 公司为其模拟器产品开发的硬件建模语言，当时它只是一种专用语言。由于相关模拟、仿真器产品的广泛使用，Verilog HDL 作为一种便于使用且实用的语言逐渐为众多设计者接受。Verilog HDL 于 1990 年被推向公众领域，OVI(Open Verilog International)是促进 Verilog 发展的国际性组织。1992 年，OVI 决定致力于推广 Verilog OVI 标准成为 IEEE 标准，这一努力最后获得成功，Verilog 语言于 1995 年成为 IEEE 标准，称为 IEEE Std 1364-1995。完整的标准在 Verilog 硬件描述语言参考手册中有详细描述。

2. Verilog HDL 的主要特点

Verilog HDL 的主要特点包括：

(1) 包含基本逻辑门，如 and、or 和 nand 等内置在语言元素中。

(2) 用户自定义元件创建的灵活性。用户自定义元件既可以是组合逻辑基本元件，也可以是时序逻辑基本元件。

(3) 开关级基本结构模型，如 PMOS 和 NMOS 等内置在语言中。

(4) 提供显式语言结构指定设计中端口到端口的延时、路径延时以及设计的时序检查。

(5) 可采用三种不同方式或混合方式对设计建模，这些方式包括行为描述方式(使用过程化结构建模)、数据流方式(使用连续赋值语句方式建模)、结构化方式(使用门和模块实例语句描述建模)。

(6) Verilog HDL 中有线网和寄存器两种类型。线网类型表示构件间的物理连线，而寄存器类型表示抽象的数据存储元件。

(7) 能够描述层次设计，可使用模块实例结构描述任何层次。

(8) 设计的规模可以是任意的，语言不对设计的规模(大小)施加任何限制。

(9) Verilog HDL 具有通用性，符合 IEEE 标准。

(10) 具有良好的可阅读性，因此它可作为 EDA 工具和设计者之间的交互语言。

(11) Verilog HDL 的描述能力能够通过使用编程语言接口(Programming

Language Interface，PLI)机制进一步扩展。PLI是允许外部函数访问Verilog模块内信息、允许设计者与模拟器交互的子程序集合。

(12) 设计能够在多个层次上加以描述，从开关级、门级、寄存器传送级(RTL)到算法级，包括进程和队列级。

(13) 能够使用内置开关级基本元件在开关级对设计完整建模。

(14) 同一语言可用于生成模拟激励和指定测试的验证约束条件，如指定输入值等。

(15) Verilog HDL能够监视仿真验证的执行，即仿真验证执行过程中设计的值能够被监控和显示。这些值也能够用于与期望值比较，在不匹配的情况下打印报告消息。

(16) 在行为级描述中，Verilog HDL不仅能够在RTL级上进行设计描述，而且能够在体系结构级描述及其算法级行为上进行设计描述。

(17) 能够使用门和模块实例化语句在结构级进行结构描述。

(18) Verilog HDL的混合方式建模能力，即在一个设计中每个模块均可以在不同设计层次上建模。

(19) Verilog HDL还具有内置逻辑函数，如&(按位与)和|(按位或)。

(20) 高级编程语言结构，如条件语句、分支选择语句和循环语句等，Verilog HDL都可以使用。

(21) 可以显式地对并发和定时进行建模。

(22) 能够提供强有力的文件读写能力。

(23) Verilog HDL在特定情况下是非确定性的，即在不同的模拟器上模型可以产生不同的结果，如事件队列上的事件顺序在标准中没有定义。

### 3.1.3 VHDL

VHDL是当前广泛使用的硬件描述语言之一，并被IEEE和美国国防部列为标准的HDL。

1. VHDL的发展

VHDL在1982年由美国国防部开始开发，在1987年被IEEE采用为标准硬件描述语言。在实际使用过程中，逐步发现了1987年版本的一些缺陷，并于1993年对1987版进行了修订。因此，现在有两个版本的VHDL，即1987年的IEEE 1076(VHDL87)和1993年进行的修订(VHDL93)。VHDL已成为开发设计可编程逻辑器件的重要工具。

2. VHDL的主要特点

VHDL能够成为标准化的硬件描述语言并获得广泛应用，必然具有很多其他硬件描述语言所不具备的优点。归纳起来，VHDL语言主要具有以下特点：

(1) 功能强大，设计方式灵活。VHDL具有功能强大的语言结构，可用简洁明确的

代码来描述复杂硬件电路。

VHDL 的设计方法灵活多样，既支持自顶向下的设计方式也支持自底向上的设计方法，既支持模块化设计方法也支持层次化设计方法。

在数字系统设计中，自上而下是一种常见的设计方法，其中“上”指的是整个数字系统的功能和定义，“下”指的是组成系统的功能部件（子模块）。自上而下的设计就是根据整个系统的功能按照一定的原则把系统划分为若干个子模块，然后分别设计实现每个子模块，最后把这些子模块组装成完整的数字系统。

（2）具有强大的硬件描述能力。VHDL 具有多层次描述系统硬件功能的能力，既可描述系统级电路，也可以描述门级电路。既可以采用行为描述、寄存器传输描述或者结构描述，也可以采用三者的混合描述方式。

VHDL 的强大描述能力还体现在它具有丰富的数据类型。VHDL 既支持标准定义的数据类型，也支持用户定义的数据类型，这样会给硬件描述带来较高的灵活性。

（3）具有很强的可移植能力。因为 VHDL 是一种标准语言，故 VHDL 的设计描述可以被不同的工具支持，可以从一种仿真工具移植到另一种仿真工具，从一种综合工具移植到另一种综合工具，从一种工作平台移植到另一种工作平台去执行。

（4）设计描述与器件无关（与工艺无关的描述性）。采用 VHDL 描述硬件电路时，设计人员并不需要首先考虑选择进行设计的器件。这样做的好处是可以使设计人员集中精力进行电路设计的优化，而不需要考虑其他的问题。当硬件电路的设计描述完成以后，VHDL 允许采用多种不同的器件结构来实现。若需对设计进行资源利用和性能方面的优化，也并不要求设计者非常熟悉器件的结构。

（5）易于共享和复用。VHDL 采用基于库的设计方法。在设计过程中，设计人员可以建立各种可再次利用的模块，一个大规模的硬件电路的设计不可能从门级电路开始一步步地进行设计，而是一些模块的组合。这些模块可以预先设计或者使用现有设计中的存档模块，将这些模块存放在设计库中，就可以在以后的设计中进行复用。也就是说，VHDL 可以使设计成果在设计人员之间方便地进行交流和共享，从而减少硬件电路设计的工作量，缩短开发周期。

（6）具有良好的可读性。VHDL 结构清晰，逻辑严谨，很容易被工程技术人员理解。

### 3.1.4 Verilog HDL 与 VHDL 的比较

VHDL 和 Verilog HDL 有类似的地方，掌握其中一种语言以后，可以通过短期的学习较快地学会另一种语言。当然，若从事集成电路设计，则必须首先掌握 Verilog HDL，因为在 IC 设计领域，90％以上的公司都是采用 Verilog HDL 进行 IC 设计。对于 PLD/FPGA 设计者而言，两种语言可以自由选择。

（1）VHDL 语法严格，书写规则比 Verilog HDL 烦琐一些；而 Verilog HDL 是在 C 语言的基础上发展起来的一种硬件描述语言，语法较自由，但其自由的语法也容易让少数初学者出错。

(2) 二者描述的层次不同。VHDL更适合行为级和RTL级的描述,Verilog HDL通常只适用于RTL级和门电路级的描述。

VHDL是一种高级描述语言,适用于描述电路的行为,即描述电路的功能,然后由综合器根据功能要求来生成符合要求的电路网表; Verilog HDL是一种较低级的描述语言,适用于描述门级电路,描述风格接近于原理图,从某种意义上来说,它是原理图的高级文本表示方式。VHDL虽然也可以直接描述门电路,但这方面能力不如Verilog HDL; 反之,在高级描述方面,Verilog HDL不如VHDL。

(3) 在VHDL设计中,大量的工作是由综合器完成的,设计者所做的工作相对较少; Verilog HDL设计中,设计者的工作量通常比较大,对设计人员的硬件水平要求比较高。

(4) 掌握VHDL设计技术比较困难。这是因为VHDL不太直观,需要有EDA编程基础,一般认为需要半年以上的专业培训才能掌握VHDL的基本设计技术。但在熟悉以后,设计效率明显高于Verilog HDL。

## 3.2 Verilog HDL的语法规则

### 3.2.1 词法规定

1. 间隔符

间隔符主要起分隔文本的作用,Verilog HDL的间隔符包括空格符(\b)、Tab键(\t)、换行符(\n)及换页符。若间隔符并非出现在字符串中,则该间隔符被忽略。所以编写程序时,可以跨越多行书写,也可以在一行内书写。

2. 标识符和关键字

给对象(如模块名、电路的输入与输出端口、变量等)取名所用的字符串称为标识符,标识符通常由英文字母、数字、$符和下画线组成,并且规定标识符必须以英文字母或下画线开始,不能以数字或$符开头。标识符是区分大小写的。例如:"clk""counter8""_net""bus_A"等都是合法的标识符; "2cp""$latch""a*b"则是非法的标识符; A和a是两个不同的标识符。

关键字是Verilog HDL本身规定的特殊字符串,用来定义语言的结构,通常为小写的英文字符串。例如,"module""endmodule""input""output""wire""reg"和"and"等都是关键字。关键字不能作为标识符使用。

3. 注释符

Verilog HDL支持两种形式的注释符: /*……*/和//……。其中,/*……*/为多行注释符,用于书写多行注释; //……为单行注释符,以双斜线//开始到行尾结束为注释文字。注释只是为了改善程序的可读性,在编译时不起作用。注意: 多行注释不能

嵌套。

### 3.2.2 逻辑值集合

为了表示数字逻辑状态，Verilog HDL 规定了 4 种基本的逻辑值，见表 3-1。

**表 3-1 4 种逻辑状态的表示**

| | |
|---|---|
| 0 | 逻辑 0、逻辑假 |
| 1 | 逻辑 1、逻辑真 |
| x 或 X | 不确定的值(未知状态) |
| z 或 Z | 高阻态 |

### 3.2.3 常量及其表示

在程序运行过程中，其值不能被改变的量称为常量。Verilog HDL 中有两种类型的常量，分别为整数型常量和实数型常量。

整数型常量有两种不同的表示方法：一是使用简单的十进制数的形式表示常量，如 30、－2 都是十进制数表示的常量，用这种方法表示的常量被认为是有符号的常量。二是使用带基数的形式表示常量，其格式为 <+/－>< size >'< base format >< number >。其中：<+/－>表示常量是正整数还是负整数，当常量为正整数时，前面的正号可以省略；< size >用十进制数定义了常量对应的二进制数的宽度；基数符号< base format >定义了后面数值< number >的表示形式，在数值表示中，最左边是最高有效位，最右边是最低有效位。整数型常量可以用二进制数(基数符号为 b 或 B)的形式表示，还可以用十进制数(基数符号为 d 或 D)、十六进制数(基数符号为 h 或 H)和八进制数(基数符号为 o 或 O)的形式表示。

下面是整数型常量实例：

```
3'b101                  //位宽为 3 位的二进制数 101
-4'd10                  //位宽为 4 位的十进制数 - 10
4'b1x0x                 //位宽为 4 位的二进制数 1x0x
12'h13x                 //位宽为 12 位的十六进制数,其中最低 4 位为未知数 x
23456                   //位宽为 32 位的十进制数 23456,十进制数的基数符号可以省略
'hc3                    //位宽为 32 位的十六进制数 c3
```

在整数的表示中要注意以下三个方面：

(1) 为了增加数值的可读性，可以在数字之间增加下画线，如 8'b1001_1100 是位宽为 8 位的二进制数 10011100。

(2) 在二进制数表示中，x、z 只代表相应位的逻辑状态；在八进制数表示中，一位 x 或 z 代表 3 个二进制位都处于 x 或 z 状态；在十六进制数表示中，一位 x 或 z 代表 4 个二进制位都处于 x 或 z 状态。

(3) 当位宽<size>没有被说明时,整数的位宽为机器的字长(至少为32位)。当位宽比数值的实际二进制位数小时,高位部分被舍去;当位宽比数值的实际二进制位数大,且最高位为0或1时,则高位由0填充;当位宽比数值的实际二进制位数多,但最高位为x或z时,高位相应由x或z填充。

实数型常量也有两种表示方法:一是使用简单的十进制记数法,如0.1、2.0、5.67等都是十进制记数法表示的实数型常量。二是使用科学记数法,如23.5e2、3.6E2、5E-4等都是使用科学记数法表示的实数型常量度,它们以十进制记数法表示分别为2350、360.0和0.0005。

为了将来修改程序的方便和改善可读性,Verilog HDL允许用参数定义一个标识符来代表一个常量,称为符号常量。定义的格式为

```
parameter param1 = const_expr1,param2 = const_expr2, … ;
```

下面是符号常量的定义实例:

```
parameter BIT = 1,BYTE = 8,PI = 3.14;
parameter DELAY = (BYTE + BIT)/2;
```

### 3.2.4 变量的数据类型

在程序运行过程中其值可以改变的量称为变量。在Verilog HDL中,变量有两大类数据类型:一类是线网类型;另一类是寄存器类型。

1. 线网类型

线网类型是硬件电路中元件之间实际连线的抽象。线网类型变量的值由驱动元件的值决定。例如:图3-2中线网L和与门G1的输出相连,线网L的值由与门的驱动信号a和b所决定,即L=a&b。a、b的值发生变化,线网L的值会立即跟着变化。当线网类型变量被定义后,没有被驱动元件驱动时,线网的默认值为高阻态z(线网trireg除外,它的默认值为x)。

常用的线网类型由关键词wire(连线)定义。若没有明确地说明线网类型变量的位宽长度的矢量,则线网类型变量的位宽为1位。在Verilog模块中若没有明确地定义输入、输出变的数据类型,则默认为位宽为1的wire型变量。wire型变量的定义格式如下:

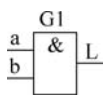

图3-2 线网示意图

```
wire [m - 1:0]变量名1,变量名2,… …,变量名n;
```

其中,方括号内以冒号分隔的两个数字定义了变量的位宽,位宽也可以用[m:1]的形式定义。

下面是wire型变量定义的一些例子:

```
wire a,b;                          //定义两个 wire(连线)类型的变量
wire L;                            //定义 1 个 wire(连线)类型的变量
wire [7:0] databus;                //定义 1 组 8 位总线
wire [32:1] busA, busB, busC;      //定义 3 组 32 位总线
```

除了常用的 wire 类型之外,还有一些其他的线网类型,如表 3-2 所示。这些类型变量的定义格式与 wire 型变量的定义相似。

**表 3-2　线网类型变量及其说明**

| 线网类型 | 功能说明 |
|---|---|
| wire,tri | 用于行为描述中对寄存器型变量的说明 |
| wor,trior | 多重驱动时,具有“线或”特性的线网说明 |
| wand,triand | 多重驱动时,具有“线与”特性的线网说明 |
| trireg | 具有电荷保持特性的线网类型,用于开关级建模 |
| tri1 | 上拉电阻,用于开关级建模 |
| tri2 | 下拉电阻,用于开关级建模 |
| supply1 | 电源线,逻辑 1,用于开关级建模 |
| supply2 | 电源线,逻辑 0,用于开关级建模 |

2. 寄存器类型

寄存器类型表示一个抽象的数据存储单元,它具有状态保持作用。寄存器型变量只能在 initial 或 always 内部赋值。寄存器型变量在没有赋值前,默认值是 x。

Verilog HDL 中,有 4 种寄存器类型的变量,如表 3-3 所示。

**表 3-3　寄存器类型变量及其说明**

| 寄存器类型 | 功能说明 |
|---|---|
| reg | 用于行为描述中对寄存器类型变量的说明 |
| integer | 32 位带符号的整数型变量 |
| real | 64 位带符号的实数型变量,默认值为 0 |
| time | 64 位无符号的时间型变量 |

常用的寄存器类型由关键词 reg 定义。若没有明确地说明寄存器类型变量是多少位宽的矢量,则寄存器变量的位宽为 1 位。reg 型变量的定义格式如下:

```
reg[m-1:0]变量名 1,变量名 2,……,变量名 n;
```

下面是 reg 型变量定义的一些例子:

```
reg clock;                         //定义 1 个 reg 类型的变量 clock
reg [3:0] counter;                 //定义 1 个 4 位 reg 类型的矢量
```

integer、real、time 这 3 种寄存器类型变量都是纯数学的抽象描述,不对应任何具体的硬件电路。integer 类型变量通常用于对整数型常量进行存储和运算,在算术运算中 integer 类型数据被视为有符号的数,用二进制补码的形式存储。而 reg 类型数据通常被

当作无符号数来处理。每个 integer 类型变量存储一个至少 32 位的整数值。注意 integer 类型变量不能使用位矢量，形如“integer [3:0] num;”的变量定义是错误的。integer 型变量的应用举例如下：

```
integer counter;                    //用作计数器的通用整数类型变量的定义
initial
    counter = -1;                   //-1 被存储在整数型变量 counter 中
```

real 类型变量由关键字 real 定义，它通常用于对实数型常量进行存储和运算，实数不能定义范围，其默认值为 0。当实数值被赋给一个 integer 类型变量时，只保留整数部分的值，小数点后面的值被截掉。

time 类型变量由关键字 time 定义，它主要用于存储仿真的时间，它只存储无符号数。每个 time 类型变量存储一个至少 64 位的时间值。为了得到当前的仿真时间，常调用系统函数 $time。仿真时间和实际时间之间的关系由用户使用编译指令 timescale 进行定义。

3. 存储器的表示

在数字电路的仿真中，人们经常需要对存储器（如 RAM、ROM 等）进行建模。在 Verilog HDL 中，将存储器看作是由一组寄存器阵列构成的，阵列中的每个元素称为一个字，每个字可以是 1 位或多位。存储器定义的格式如下：

```
reg [msb:lsb]  memory1[upper1:lower1], memory2[upper2:lower2], …… ;
```

其中，memory1、memory2 等为存储器的名称，[upper1:lower1]定义了存储器 memory1 的地址间的范围，高位地址写在方括号的左边，低位地址写在方括号的右边；[msb:lsb]定义了存储器中每个单元（字）的位宽。

下面是存储器定义的一个例子。图 3-3 给出了定义的示意图。

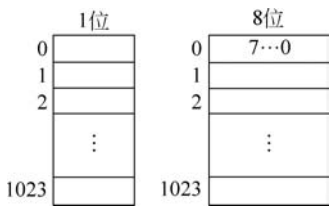


图 3-3　存储器定义示意图

```
reg mem1bit[1023:0];          //定义了 1024 个 1 位
                              //存储器变量 mem1bit
reg [7:0] membyte[1023:0];    //定义了 1024 个 8 位存储器变量 membyte
```

需要注意的是：Verilog 中定义的存储器只有字寻址的能力，即对存储器赋值时，只能对存储器中的每个单元（字）进行赋值，不能将存储器作为一个整体在一条语句中对它赋值。也不能对存储器一个单元中的某几位进行操作。若要判断存储器一个单元中某几位的状态，则可以先将该单元的内容赋给 reg 类型变量，再对 reg 类型变量中相应位进行判断。

```
reg datamem[5:1];                   //定义了 5 个 1 位存储器 datamem
initial
    datamem = 5'b11001;             //非法，不能将存储器作为一个整体对所有单元同时赋值
```

下面是对存储器中每个单元(字)进行赋值的正确实例：

```
reg [3:0] reg_A;                    //定义了 1 个 4 位的寄存器变量 reg_A
reg [3:0] romA[4:1];                //定义了 4 个 4 位存储器 romA
initial
    begin
        romA[4] = 4'hA;             //正确,对存储器中的 1 个单元赋值
        romA[3] = 4'h8;
        romA[2] = 4'hF;
        romA[1] = 4'h2;
        reg_A = romA[2];            //正确,允许将存储器中某单元的内容赋给寄存器型变量
end
```

另外需要说明的是,Verilog 中定义的存储器只是对存储器行为的抽象描述,并不涉及存储器的物理实现。若用 Verilog 中的定义去综合一个存储器,则它将全部由触发器实现。考虑到 RAM、ROM 的特殊性,在实际设计存储器时,总是通过直接调用厂家提供的存储器宏单元库的方式实现存储器,这里的定义仅用于行为描述与仿真。

### 3.2.5 Verilog HDL 运算符

Verilog HDL 定义了多种运算符,可对一个、两个或三个操作数进行运算。表 3-4 按类别列出了这些运算符。Verilog HDL 中的有些运算符与 C 语言中的相似,下面对部分运算符进行介绍。

**表 3-4 Verilog HDL 运算符的类型与符号**

| 运算符类型 | 运算符 | 功能说明 | 操作数个数 | 运算符类型 | 运算符 | 功能说明 | 操作数个数 |
|---|---|---|---|---|---|---|---|
| 算术运算符 | +,-,*,/ | 算术运算 | 2 | 缩位运算符 | & | “与”缩位 | 1 |
| | % | 求模 | 2 | | ~& | “与或”缩位 | 1 |
| 关系运算符 | <,>,<=,>= | 关系运算 | 2 | | \| | “或”缩位 | 1 |
| | | | | | ~\| | “异或”缩位 | 1 |
| 相等运算符 | == | 逻辑相等 | 2 | | ^~or~^ | “异或非(同或)”缩位 | 1 |
| | != | 逻辑不等 | 2 | | | | |
| | === | 全等 | 2 | 移位运算符 | << | 向左移位 | 2 |
| | !== | 不全等 | 2 | | >> | 向右移位 | 2 |
| 逻辑运算符 | ! | 逻辑非 | 1 | 条件运算符 | ?: | 条件运算 | 3 |
| | && | 逻辑与 | 2 | | | | |
| | \|\| | 逻辑或 | 2 | | | | |
| 按位运算符 | ~ | 按位“非” | 1 | 位拼接运算符 | {} | 拼接(合并) | ≥2 |
| | & | 按位“与” | 2 | | | | |
| | \| | 按位“或” | 2 | | | | |
| | ^ | 按位“异或” | 2 | | | | |
| | ^~或~^ | 接位“异或非(同或)” | 2 | | | | |

1. 算术运算符

算术运算符又称为二进制运算符。在进行算术运算时，若某个操作数的某一位为 x（不确定值）或 z（高阻值），则整个表达式运算结果也为 x。例如，4'b101x+4'b0111，结果为 4'bxxxx。

两个整数进行除法运算时，结果为整数，小数部分被截去。例如，6/4 结果为 1。

两个整数取模运算得到的结果为两数相除后的余数，余数的符号与第一个操作数的符号相同。例如，−7%2 结果为−1，7%−2 结果为 1。

2. 相等运算符

“==（相等）”和“!=（不等）”又称为逻辑等式运算符，其运算结果可能是逻辑值 0、1 或 x（不定态）。相等运算符（==）逐位比较两个操作数相应位的值是否相等；若相应位的值都相等，则相等关系成立，返回逻辑值 1；否则，返回逻辑值 0。若任何一个操作数中的某一位为未知数 x 或高阻值 z，则结果为 x。当两个参与比较的操作数不相等时，则不等关系成立。“==”和“!=”运算规则如表 3-5 所示。

**表 3-5 “==”和“!=”运算规则**

| == | | 操作数 1 | | | | != | | 操作数 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | x | z | | | 0 | 1 | x | z |
| 操作数 2 | 0 | 1 | 0 | x | x | 操作数 2 | 0 | 0 | 1 | x | x |
| | 1 | 0 | 1 | x | x | | 1 | 1 | 0 | x | x |
| | x | x | x | x | x | | x | x | x | x | x |
| | z | x | x | x | x | | z | x | x | x | x |

“===（全等）”和“!==（不全等）”常用于 case 表达式的判别，所以又称为“case 等式运算符”，其运算结果是逻辑值 0 和 1。全等运算符允许操作数的某些位为 x 或 z，只要参与比较的两个操作数对应位的值完全相同，则全等关系成立，返回逻辑值 1；否则，返回逻辑值 0。不全等就是两个操作数的对应位不完全一致，则不全等关系成立。“===”与“!==”运算规则如表 3-6 所示。

**表 3-6 “===”和“!==”运算规则**

| === | | 操作数 1 | | | | !== | | 操作数 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | x | z | | | 0 | 1 | x | z |
| 操作数 2 | 0 | 1 | 0 | 0 | 0 | 操作数 2 | 0 | 0 | 1 | 1 | 1 |
| | 1 | 0 | 1 | 0 | 0 | | 1 | 1 | 0 | 1 | 1 |
| | x | 0 | 0 | 1 | 0 | | x | 1 | 1 | 0 | 1 |
| | z | 0 | 0 | 0 | 1 | | z | 1 | 1 | 1 | 0 |

下面是相等与全等运算符的一些运算实例。

假设 A=4'b1010，B=4'b1101，M=4'b1xxz，N=4'b1xxz，P=4'b1xxx，则运算结果如表 3-7 所示。

表 3-7　相等与全等运算实例结果

| A==B | A!=B | A==M | A===M | M===N | M===P | M!==P |
|---|---|---|---|---|---|---|
| 0 | 1 | x | 0 | 1 | 0 | 1 |

3. 逻辑运算符

进行逻辑运算时，其操作数可以是寄存器变量，也可以是表达式。逻辑运算的结果为 1 位：1 代表逻辑真，0 代表逻辑假，x 代表不定态。

若操作数是 1 位，则 1 表示逻辑真，0 表示逻辑假。

若操作数由多位组成，则将操作数作为一个整体看待，对非零的数作为逻辑真处理，对每位均为 0 的数作为逻辑假处理。

若任一个操作数中含有 x(不定态)，则逻辑运算的结果也为 x。

4. 按位运算符

按位运算符完成的功能是将操作数的对应位按位进行指定的运算操作，原来的操作数有几位，则运算的结果仍为几位。若两个操作数的位宽不一样，则仿真软件会自动将短操作数的左端高位部分以 0 补足(注意：若短的操作数最高位为 x，则扩展得到的高位也是 x)。表 3-8 是按位运算符的运算规则。

表 3-8　按位运算符运算规则

| &(与) | | 操作数 1 | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| 操作数 2 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 0 | 1 | x | x |
| | x | 0 | x | x | x |
| | z | 0 | x | x | x |
| \|(或) | | 操作数 1 | | | |
| | | 0 | 1 | x | z |
| 操作数 2 | 0 | 0 | 1 | x | x |
| | 1 | 1 | 1 | 1 | 1 |
| | x | x | 1 | x | x |
| | z | x | 1 | x | x |
| ^(异或) | | 操作数 1 | | | |
| | | 0 | 1 | x | z |
| 操作数 2 | 0 | 0 | 1 | x | x |
| | 1 | 1 | 0 | x | x |
| | x | x | x | x | x |
| | z | x | x | x | x |

续表

| ^～(同或) | | 操作数 1 | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| 操作数 2 | 0 | 1 | 0 | x | x |
| | 1 | 0 | 1 | x | x |
| | x | x | x | x | x |
| | z | x | x | x | x |
| ～(非) | | 操作数 1 | | | |
| | | 0 | 1 | x | z |
| 结果 | | 1 | 0 | x | x |

假设 A=4b'1010,B=4'1101,C=4'b10x1,则运算结果见表 3-9。

**表 3-9 按位运算实例结果**

| A&B | A\|B | A^B | A^～B | A&C | ～A | ～C |
|---|---|---|---|---|---|---|
| 4'b1000 | 4'b1111 | 4'b0111 | 4'b1000 | 4'b10x0 | 4'b0101 | 4'b01x0 |

5. 缩位运算符

缩位运算符仅对一个操作数进行运算,并产生一位逻辑值。缩位运算规则与表 3-8 所示的位运算相似,不同的是缩位运算符的操作数只有一个,运算时,按照从右到左的顺序依次对所有位进行运算。假设 A 是 1 个 4 位的寄存器,它的 4 位从左到右分别是 A[3]、A[2]、A[1]、A[0],则对 A 进行缩位运算时,先对 A[1]、A[0]进行运算,得到 1 位的结果,再将这个结果与 A[2]进行运算,其结果再接着与 A[3]进行运算,最终得到的结果为 1 位,因此被形象地称为缩位运算。

若操作数的某位为 x,则缩位运算的结果为 1 位的不定态 x。

下面是缩位运算的一些实例,假设 A=4'b1010,则结果如表 3-10 所示。

**表 3-10 缩位运算实例结果**

| &A | \|A | ^A | ～&A | ～\|A | ～^A |
|---|---|---|---|---|---|
| 1'b0 | 1'b1 | 1'b0 | 4'b1 | 4'b0 | 4'b0 |

6. 位拼接运算符

位拼接运算符是 Verilog HDL 中一种比较特殊的运算符,其作用是把两个或多个信号中的某些位拼接在一起进行运算。其用法如下:

{信号 1 的某几位,信号 2 的某几位,…,信号 n 的某几位}

即把几个信号的某些位详细地列出来,中间用逗号隔开,最后用大括号括起来表示一个整体信号。

对于一些信号的重复连接,可以使用简化的表示方式{n{A}}。这里 A 是被连接的

对象，n 是重复的次数，它表示将信号 A 重复连接 n 次。下面是连接运算符的运算实例：

```
reg A;reg[1:0] B, C;
A = 1'b1; B = 2'b00; C = 2'b10;

Y = {B,C}                          //结果 Y = 4'b0010
Y = {A,B[0],C[1],1'b1}             //结果 Y = 4'b1011,常数的位宽必须有初始值
Y = {4{A}}                         //结果 Y = 4'b1111
Y = {2{A},2{B},C}                  //结果 Y = 8'b1100_0010
Y = {A,B,5}                        //非法,因为常数 5 的位宽不确定
```

### 3.2.6 赋值语句

在 Verilog HDL 中，信号有非阻塞和阻塞两种赋值方式。

1. 非阻塞赋值方式

其语句的表示式为

```
y<=x;
```

语法规则如下：

(1) 在语句块中，上一条语句所赋的变量值不能立即被后续语句所用；

(2) 块结束后才能完成本次赋值操作，而所赋的变量值是上一次赋值得到的；

(3) 在编写可综合的时序逻辑模块时，这是最常用的赋值方法。

**注意**：非阻塞赋值符“<=”与小于等于符“<=”看起来是一样的，但意义完全不同，小于等于符是关系运算符，用于比较大小，而非阻塞赋值符号用于赋值操作。

2. 阻塞赋值方式

其语句的表示式为

```
y = x;
```

语法规则如下：

(1) 赋值语句执行完后，块才结束；

(2) y 的值在赋值语句执行完成后立刻改变；

(3) 在时序逻辑中使用时，可能会产生意想不到的结果。

下面举例说明非阻塞赋值方式和阻塞赋值方式的区别。非阻塞赋值实例：

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

在这段实例中，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，这里用到了两个触发器。

**注意**：赋值是在 always 块结束后执行的，c 为原来 b 的值。这个 always 块实际描述的电路功能如图 3-4 所示。

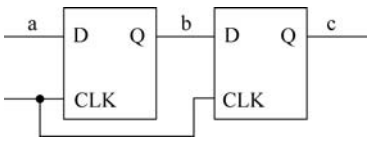

图 3-4　阻塞方式赋值

阻塞赋值实例：

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

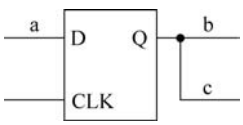

图 3-5　阻塞方式赋值

这段实例中的 always 块用了阻塞赋值方式。clk 信号的上升沿到来时，将发生如下变化：b 立即取 a 的值，c 立即取 b 的值（即等于 a），生成的电路如图 3-5 所示，图中只用了一个触发器来寄存 a 的值，同时输出给 b 和 c。这大概不是设计者的初衷，采用非阻塞赋值方式就可以避免这种错误。

## 3.3　Verilog HDL 基础与程序结构

### 3.3.1　模块

模块是 Verilog HDL 的基本描述单位，用于描述某个设计的功能或结构及其与其他模块通信的外部端口。一个设计的结构可使用开关级基本单元、门级基本单元和用户定义的基本单元方式描述；设计的数据流行为使用连续赋值语句进行描述；时序行为使用过程结构描述。一个模块可以在另一个模块中调用。

一个模块的基本语法如下：

```
module module_name (port_list);        //模块定义开始
    Declarations:                      //模块声明部分
    reg, wire, parameter,              //寄存器、线网型变量以及参数定义
    input, output, inout,              //端口定义
    function, task, . . .              //函数及过程定义
    Statements:                        //行为描述语句部分
    Initial statement                  //变量初始化
    Always statement                   //时序控制语句
    Module instantiation               //模块调用初始化
    Gate instantiation                 //门级语句初始化
    UDP instantiation                  //用户自定义基本模块初始化
    Continuous assignment              //连续赋值语句
endmodule                              //模块定义结束
```

模块的定义从关键字 module 开始，到关键字 endmodule 结束，每条 Verilog HDL 语句以“;”作为结束(块语句、编译向导、endmodule 等少数除外)。

一个完整的 Verilog 模块由以下 4 部分组成：

(1) 模块定义行：module module_name (port_list)。

(2) 说明部分用于定义不同的项，例如：模块描述中使用的寄存器和参数。语句定义设计的功能和结构。说明部分和语句可以分布在模块中的任何地方；但是变量、寄存器、线网和参数等的说明部分必须在使用前出现。为了使模块描述清晰和具有良好的可读性，最好将所有的说明部分放在语句前。

说明部分包括：寄存器、线网以及参数，如 reg、wire、parameter；端口类型说明行，如 input、output、inout；函数、任务等，如 function、task。

(3) 描述主体部分：这是一个模块最重要的部分，在这里描述模块的行为和功能、子模块的调用和连接、逻辑门的调用、用户自定义部件的调用、初始态赋值、always 块以及连续赋值语句等。

(4) 结束行，以 endmodule 结束，该关键字后无分号。

以下是建模一个半加器电路模块的简单实例：

```
module  HalfAdder (A, B, Sum, Carry);
input  A, B;
output  Sum, Carry;
assign #2 Sum = A^B;
assign #3 Carry = A & B;
endmodule
```

模块的名字是 HalfAdder。模块有 4 个端口：两个输入端口 A 和 B，两个输出端口 Sum 和 Carry。由于没有定义端口的位数，所有端口宽度都为 1 位；同时，由于没有各端口的数据类型说明，这四个端口都是线网数据类型。

模块包含两条描述半加器数据流行为的连续赋值语句。从这种意义上讲，这些语句在模块中出现的顺序无关紧要，这些语句是并发的。每条语句的执行顺序依赖于发生在变量 A 和 B 上的事件。

在模块中，可用数据流方式、行为方式、结构方式及三种描述方式的混合描述一个设计。

下面首先对 Verilog HDL 的时延作简要介绍。

### 3.3.2 时延

Verilog HDL 模型中的所有时延都根据时间单位定义。下面是带时延的连续赋值语句实例。时间单位是由 timescale 定义的，timescale 将在后面讲述。

```
assign #2 Sum = A ^ B;
#2 指 2 个时间单位.
```

使用编译指令将时间单位与物理时间相关联。这样的编译器指令需在模块描述前定义,如下所示:

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

此语句说明时延的时间单位为 1ns,并且时间精度为 100ps(即所有的时延必须被限定在 0.1ns 内)。如果此编译器指令所在的模块包含上面的连续赋值语句,#2 代表 2ns。

如果没有这样的编译器指令,Verilog HDL 仿真器会指定一个默认时间单位。IEEE Verilog HDL 标准中没有规定默认时间单位。

### 3.3.3 常用语句

1. always 结构型语句

always 本身是一个无限循环语句,即不停地循环执行其内部的过程赋值语句,直到仿真过程结束。always 语句主要用于对硬件电路的行为功能进行描述,也可以在测试模块中对时钟信号进行描述。但用它来描述硬件电路的逻辑功能时,通常在 always 后面紧跟循环的控制条件。always 语句一般用法如下:

```
always @(事件控制表达式)
    begin:块名
        块内局部变量的定义;
        一条或多条过程赋值语句;
    end
```

这里,“事件控制表达式”也称为敏感事件表,即等待确定的事件发生或某一特定的条件变为“真”,它是执行后面过程赋值语句的条件。“过程赋值语句”左边的变量必须定义成寄存器数据类型,右边变量可以是任意数据类型。若 always 语句后面没有“事件控制表达式”,则认为循环条件总为“真”。begin 和 end 将多条过程赋值语句包围起来,组成一个顺序语句块,块内的语句按照排列顺序依次执行,最后一条语句执行完成后,执行挂起,然后 always 语句处于等待状态,等待下一个事件的发生。“块名”是给顺序块取的名字,可以使用任何合法的标识符。注意,当 begin 和 end 之间只有一条语句,且没有定义局部变量时,关键词 begin 和 end 可以省略。

2. 顺序语句块

顺序语句块是由块标识符 begin、end 包围界定的一组行为描述语句,其作用是给块中这组行为描述语句进行打包处理,使之在形式上与一条语句相一致。

begin、end 是顺序语句块的标识符,位于这个块内部的各条语句按照书写的先后顺序依次执行,块中每条语句给出的时延都是相对于前一条语句执行结束时的相对时间。因而,由 begin、end 界定的语句块称为语句块(简称为顺序块或串行块)。

顺序块的起始执行时间就是块中第一条语句开始被执行的时间，执行结束的时间就是块中最后一条语句执行完成的时间，即最后一条语句执行完后，程序流程控制跳出该语句块。

在 Verilog 语言中，可以给每个语句块取一个名字，方法是在关键字 begin 后面加上一个冒号，之后给出名字即可。取了名字的块称为命名块。

3. 条件语句

条件语句就是根据判断条件来确定下一步的运算。Verilog 语言中有 3 种形式的 if 语句，一般用法如下：

```
    if (condition_expr) true_statement;
或   if (condition_expr) true_statement;
    else false_statement;
或   if (condition_expr1) true_statement1;
    else if (condition_expr2) true_statement2;
    else if (condition_expr3) true_statement3;
⋮
    else default_statement;
```

if 后面的条件表达式一般为逻辑表达式或关系表达式。执行 if 语句时，首先计算表达式的值，若结果为 0、x 或 z，按“假”处理，若结果为 1，按“真”处理，执行相应的语句。

**注意**：在第三种形式中，从第一个条件表达式 condition_expr1 开始依次进行判断，直到最后一个条件表达式被判断完毕，如果所有的表达式都不成立，才会执行 else 后面的语句。这种判断上的先后次序本身隐含着一种优先级关系，在使用时应予以注意。

4. 多路分支语句

case 语句是一种多分支条件选择语句，一般形式如下：

```
case (case_expr)
    item_expr1:statement1;
⋮
    item_expr2:statement2;
    default:default_statement;  //default 语句可以省略
```

执行时，首先计算 case_expr 的值，然后依次与各分支项中表达式的值进行比较：如果 case_expr 的值与 case_expr1 的值相等，就执行语句 statement1；如果 case_expr 的值与 case_expr2 的值相等，就执行语句 statement2；……；如果 case_expr 的值与所有列出来的分支项的值都不相等，就执行语句 default_statement。

使用时需要注意以下三点：

(1) 每个分支项中的语句可以是单条语句，也可以是多条语句。如果是多条语句，必须在多条语句的最前面写上关键词 begin，在这些语句的最后写上关键词 end，这样多条语句就成了一个整体，称为顺序语句块。

(2) 每个分支项表达式的值必须各不相同,一旦判断到与某分支的值相同并执行相应语句后,case 语句的执行便结束了。

(3) 若某几个连续排列的分支执行同一条语句,则这几个分支项表达式之间可以用逗号分隔,将语句写在这几个分支项表达式的最后一个中。

下面是一个对 4 选 1 数据选择器进行建模的实例:

```
module mux4to1_bh(out,in,s1,s2,en);
    input [3:0] in;
    input  s1,s2;
    output  out;
    reg out;
    always@(in or s1 or s2 or en)
    begin
        if (en == 1)  out = 0;        //也可以写成 if(en)  out = 0;
        else
          case ({s1,s0})
            2'd0:out = in[0];
            2'd1:out = in[1];
            2'd2:out = in[2];
            2'd3:out = in[3];
            default: $ display("invalid control signals");
          endcase
    end
endmodule
```

此建模的标识是 always 结构,always 后面跟着循环执行的条件@(in or s1 or s0 or en)(注意后面没有分号),它表示圆括号内的任一个变量发生变化时,后面的过程赋值语句就会被执行一次,执行完最后一条语句后,执行挂起,always 语句等待变量再次发生变化。因此,将圆括号内列出的变量称为敏感变量。对组合逻辑电路来说,所有的输入信号都是敏感变量,应该写在圆括号内。

由上面可总结以下几点注意事项:

(1) 敏感变量之间使用关键词 or 代替了逻辑或运算符(|)。

(2) 过程赋值语句只能给寄存器型变量赋值,因此输出变量 out 被定义成 reg 数据类型。

(3) 在对 4 选 1 数据选择器的模型仿真时,若输入{s1,s2}出现未知状态 x,则程序会执行 default 后面的语句。但在实际的硬件电路中输入信号 s1、s2 一般不会出现 x 的情况,所以在写可综合的代码时 default 语句可以省略不写。

case 语句还有两种变体,即 casez 和 casex。在 casez 语句中,将 z 视为无关值,如果比较的双方(case_expr 的值与 item_expr 的值)有一方的某一位的值是 z,该位的比较就不予考虑,即认为这一位的比较结果永远为“真”,因此只需关注其他位的比较结果。在 casex 语句中,将 z 和 x 都视为无关值,对比较双方(case_expr 的值与 item_expr 的值)出现 z 或 x 的相应位均不考虑。注意,对无关值可以用“?”表示。除了用关键词 casez 或

casex 来代替 case 以外，casez 和 casex 的用法与 case 语句的用法相同。

下面是一个 4-2 线优先编码器的建模实例：

```
module Priority_encoder(In,out_coding);
    input[3:0] In;
    output[1:0] out_coding;
    wire[3:0] In;
    reg[1:0] out_coding;
    always@(In)
    begin
      casez(In)                        //逻辑值 z 代表无关值
        4'b1???: out_coding = 2'b11;
        4'b01??: out_coding = 2'b10;
        4'b001?: out_coding = 2'b01;
        4'b0001: out_coding = 2'b00;
        default:  out_coding = 2'b00;
      endcase
    end
endmodule
```

5. 循环语句

Verilog 语句提供了 forever、repeat、while 和 for 四种类型的循环语句。所有循环语句都只能在 initial 或 always 内部使用，循环语句内部可以包含时延控制。

1）forever 循环语句

forever 是一种无限循环语句，其语法如下：

```
forever 语句块
```

该语句不停地循环执行后面的过程语句块。一般在语句块内容要使用某种形式的时序控制结构；否则，Verilog 仿真器将会无限循环，后面的语句将永远无法被执行。

2）repeat 循环语句

repeat 是一种预先指定循环次数的循环语句。其语法如下：

```
repeat(循环次数表达式)语句块
```

其中，“循环次数表达式”用于指定循环次数，它可以是一个整数、变量或一个数值表达式。若是变量或数值表达式，其取值只有在第一次进入循环时得到计算，即事先确定循环次数。若循环次数表达式的值不确定，即 x 或 z，则循环次数按 0 处理。

3）while 循环语句

while 是一种有条件的循环语句。其语法如下：

```
while(条件表达式)语句块
```

该语句只有在指定的条件表达式取值为“真”时，才会重复执行后面的过程语句；否则，就不执行循环体。若表达式在开始时为假，则过程语句永远不会执行。若条件表达

式的值为 x 或 z,则按 0(假)处理。

4) for 循环语句

for 语句是一种条件循环语句,只有在指定的条件表达式成立时才进行循环。其语法如下:

```
for(表达式 1; 条件表达式 2; 表达式 3)  语句块
```

其中:“表达式 1”用来对循环计数变量赋初值,只在第一次循环开始前计算一次。“条件表达式 2”是循环执行时必须满足的条件,在循环开始后,先判断这个条件表达式的值,若为“真”,则执行后面的语句块;接着计算“表达式 3”,修改循环数变量的值,即增加或减少循环次数。然后再次对“条件表达式 2”进行计算和判断,若“条件表达式 2”的值仍为“真”,则继续执行上述的循环过程;若“条件表达式 2”的值为“假”,则结束循环,退出 for 循环语句的执行。

下面是用 for 语句、移位操作及加法运算实现 8 位乘法器的实例。

```
module _8bit_mutiplier(Result, opA, opB);
    parameter SIZE = 8, LONGSIZE = 16;
    input[SIZE - 1:0] opA, opB;
    output[LONGSIZE - 1:0] Result;
    wire[SIZE - 1:0] opA, OPB;
    reg[LONGSIZE - 1:0] Result;
      always@(opA or opB)
      begin: mult
        integer index;
        Result = 0;
        for(index = 0;index < = SIZE - 1;index = index + 1)
              if (opB[index] == 1)
                    Result = Result + (opA << index);
      end
    endmodule
```

### 3.3.4 系统任务和函数

以 $ 字符开始的标识符表示系统任务或系统函数。Verilog HDL 提供了一系列的系统功能调用,任务型的功能调用称为系统任务,函数型的调用称为系统函数。系统任务提供了一种封装行为的机制,这种机制可在设计的不同部分被调用,任务可以返回 0 个或多个值。系统函数除只能返回一个值以外,其余与任务相同。此外,函数在 0 时刻执行,即不允许时延,而任务可以带有时延。一般可以统称为系统函数。

Verilog HDL 中的系统任务和系统函数是面向仿真的,嵌入 Verilog HDL 语句中的仿真系统功能调用。这一部分与相关的仿真器直接相关,不同的仿真器,支持的系统函数可能会有所不同,但是大多数系统函数都是支持的。下面介绍最基本的、最常用的系统任务和系统函数。

系统任务和系统函数可以分成以下五类：

(1) 输出控制类系统函数：仿真过程的状态信息以及仿真结果的输出都必须按一定的格式进行表示，Verilog 所提供的输出控制类系统函数的目的是完成对输出量的格式控制。属于这一类的有 $display、$write、$minitor 等。

(2) 仿真时标类系统函数：Verilog 中有一组与模拟时间定标相关的系统函数，如 $time、$realtime 等。

(3) 进程控制类系统任务：这一类系统任务用于对仿真进程控制，如 $finish、$stop 等。

(4) 文件读写类系统任务：用于控制对数据文件读写方式，如 $readmem 等。

(5) 其他类：比如随机数产生系统函数，如 $random 等。

1. $display 和 $write

调用格式为

```
$display("格式控制输出和字符串",输出变量名表);
$write("格式控制输出和字符串",输出变量名表);
```

输出变量名表就是指要输出的变量，各变量之间用逗号相隔。格式控制输出内容包括需要输出的普通字符和对输出变量显示方式控制的格式说明符，格式说明符和变量需要一一对应。$display 和 $write 的区别是前者输出结束后自动换行，后者不会。

格式控制符用于对变量的格式进行控制，指定变量按照一定的格式输出，如表 3-11 所示。

**表 3-11　格式说明符与输出格式的关系**

| 格式说明符 | 输 出 格 式 |
|---|---|
| %h 或%H | 以十六进制的格式输出 |
| %d 或%D | 以十进制的格式输出 |
| %o 或%O | 以八进制的格式输出 |
| %b 或%B | 以二进制的格式输出 |
| %c 或%C | 以 ASCII 字符形式输出 |
| %s 或%S | 以字符串方式输出 |
| %v 或%V | 输出线网型数据的驱动强度 |
| %t 或%T | 输出仿真系统所使用的时间单位 |
| %m 或%M | 输出所在模块的分级名 |
| %e 或%E | 将实型量以指数方式显示 |
| %f 或%F | 将实型量以浮点方式显示 |
| %g 或%G | 将实型量以上面两种较短的方式显示 |

1) 控制字符“h、d、o、b”

用于对整型量数据的输出控制。对于有位宽定义的变量，输出的宽度将由位宽和输出的数制格式两方面决定。若数据的前面有很多个前导 0，则可以在控制字符前加 0，如%0b，这样一个数据为 0010 的就显示为 10。

通常情况下，每一个变量都需要一个对应有控制字符，若采用系统默认格式，则$display函数将按十进制方式显示，$displayh代表默认态为十六进制，$displayo代表默认态为八进制，$displayb代表默认态为二进制。

在数据中可能会有某些位是不定态x或者高阻态z，若用二进制方式显示，则每一位都将显示出来；若对于八进制或者十六进制，则它们的一位相当于二进制的三位或者四位；若这几位都是x，则八进制、十六进制的相应位也为x；若这几位不全是x，只是个别位是x，则八进制、十六进制的相应位为x。对于高阻z规则相同。但是对于十进制的表示时，由于没有相互对应的位，所以把十进制数当作一个整体对待，规则相同。若全部为x或者z，则十进制为x或者z；若部分为x或者z，则十进制为x或者z；但是对于既有x又有z，没有规定统一的标准。

2）控制字符“c、s”

用于把变量转化成字符或者字符串进行输出。对于%c(或者%C)，若变量的位宽大于8位，则只取最低8位，输出它的ASCII字符；若变量低于8位，则高位补0。对于%s(或者%S)，若变量位宽小于8位，则高位补0并输出它的ASCII字符，若变量宽度大于8位，则从低位开始每8位输出对应的ASCII字符，一直到剩余高端部分全部为0时停止。

3）控制字符“v”

控制字符“v”只能用于一位宽的线网型变量，用于输出它的驱动强度。Verilog HDL中定义了8级驱动强度，定义及缩写表示如图3-12所示。

**表3-12　8级驱动强度的表示及对应关系**

| 缩写符号 | 强度名称 | 强度等级 |
|---|---|---|
| Su | Super | driver 7 |
| St | Strong | driver 6 |
| Pu | Pull | driver 5 |
| La | Large | capacitor 4 |
| We | Weak | driver 3 |
| Me | Medium | driver 2 |
| Sm | Small | capacitor 1 |
| Hi | High | impedance 0 |

4）控制字符“t、m”

这两个控制字符都不需要有相应的输出变量与之对应，因为它们反映的是仿真系统或者模块本身的信息。%t给出了系统运行仿真程序所用的仿真时间以什么为单位。%m给出当前所在模块的名称。需要说明的是，它显示的名称是分级名，也就是模块被调用时的调用名；另外，除了模块外，任务、函数、有名块，都构成一个新的层次并将在分级名中出现。

5）控制字符“e、f、g”

这三个控制字符是专门为实数型常量或变量的输出而设置的，对它们的定义C语言中的相应定义相同。

2. $monitor

与$display与$write一样，$monitor同属于输出控制类，它的调用形式可以有以下三种：

```
$monitor("格式控制输出和字符串",输出变量名表);
$monitoron;
$monitoroff;
```

以上第一种的格式与上面的$display完全一致，不同点是，$display每调用一次执行一次，而$monitor一旦被调用，就会随时对输出变量名表中的每一个变量进行检测，如果发现其中任何一个变量在仿真过程中发生了变化，就会按照$monitor中的格式产生一次输出。

为了明确输出的信息究竟在仿真过程中的什么时刻产生的，通常情况下$monitor的输出中会用到一个系统函数$time，例如：

```
$monitor( $time,"signal1 = %b signal2 = %b", signal1,signal2);
```

对$time的返回值也可以进行格式控制，例如：

```
$monitor("%d signal1 = %b signal2 = %b", $time, signal1,signal2);
```

由于$monitor一旦被调用后就会启动一个后台进程，因而不可能在有循环性质的表达式中出现，例如：always过程块或者其他高级程序循环语句，在实际应用中，$monitor通常位于initial过程块的最开始处，保证从一开始就实时地检测所关心的变量的变化状态。

3. $time和$realtime

$time和$realtime属于仿真时标类系统函数，对这两个函数调用，将返回从仿真程序开始执行到被调用时刻的时间，不同之处在于$time返回的是64位整数，而$realtime返回的是一个实型数。例如：

```
'timescale 10ns /1ns
module time_demo;
reg ar;
parameter delay = 1.6
initial
    begin
        $display ("time value");
        $monitor( $time, "var = %b",var);
        #delay var = 1;
        #delay var = 0;
        #1000 $finish;
    end
endmodule
```

显示结果如下：

```
time value
0 var = x
2 var = 1
3 var = 0
```

这个例子中，系统时间定标为10ns，为计时单位，所以delay=1.6实际代表的时间是16ns。按照上例中的时序描述，16ns之后变量赋值一次，再过16ns即32ns时再赋值一次，按理说，输出时间应该是1.6和3.2，可是实际输出是2和3，这是因为$time在返回时间变量时进行了四舍五入。

若把上例中的$time换成$realtime，则直接可以得到下面按照实数型方式显示的检测结果，没有四舍五入引起的误差问题。

```
time value
0 var = x
1.6 var = 1
3.2 var = 0
```

4. $finish和$stop

这两个系统任务用于控制仿真进程。

$finish的调用方式如下：

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

它的作用是中止仿真器的运行，结束仿真过程。可以带上一个参数，参数n只能取以下3个值：0，不输出任何信息；1，输出结束仿真的时间和仿真文件的位置；2，在1的基础上增加对CPU时间、机器内存占用情况等统计结果的输出。

$finish不指明参数时，默认为1。

$stop的调用方式相同和$finish相同，参数也相同。不同的是，$stop的作用只相当于一个pause的暂停语句，仿真程序在执行到$stop时，暂停下来，这时设计人员可以输入相应的命令，对仿真过程进行交互控制，比如用force/release语句，对某些信号实行强制性修改，在不退出仿真进程的前提下，进行仿真调试。

5. $readmem

Verilog中针对文件的读写控制有许多相应的系统任务和系统函数，这里只介绍$readmem，它的作用是把一个数据文件中的数据内容读入指定的存储器中。其有两种调用方式：

```
$readmemb("文件名",存储器名,起始地址,结束地址);
$readmemh("文件名",存储器名,起始地址,结束地址);
```

这里，文件名是指数据文件的名字，必要时需要包括相应的路径名；存储器名是需要读入数据的存储器的名字，起始地址和结束地址是表明读取的数据从什么地方开始存放。

若默认起始地址，则从存储器的第一个地址开始存放；若默认结束地址，则一直存放到存储器的最后一个地址为止。

$readmemb 和 $readmemh 区别是对数据文件存放格式的不同，前者要求以二进制方式存放，后者要求以十六进制方式存放。

6. $random

该函数能够产生一个随机数，其调用格式为

```
$random %b
```

其中，b>0，它将产生一个值为(－b＋1)～(b－1)的随机数。

这样，仿真过程在需要时可以为测试模块提供随机脉冲序列，例如：

```
reg[7:0] ran_num;
always
#(140 + ( $random %60)) ran_num = $random %60
```

这样 ran_num 的值在－59～＋59 随机产生，且随机数产生的时延间隔在 81～159 变化。

### 3.3.5 编译指令

以“`”(反单引号)开始的某些标识符是编译器指令。在编译 Verilog HDL 时，特定的编译器指令在整个编译过程中有效(编译过程可跨越多个文件)，直到遇到其他的不同编译程序指令。

1. `define 和`undef

`define 指令用于文本替换，它很像 C 语言中的#define 指令。例如：

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

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

`undef 指令取消前面定义的宏。例如：

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

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

2. `ifdef、`else 和`endif

这些编译指令用于条件编译。例如：

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

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

`else 程序指令对于`ifdef 指令是可选的。

3. `default_nettype

该指令指定隐式线网类型，也就是为那些没有被说明的连线定义线网类型。例如：

```
`default_nettype wand
```

该实例定义的默认的线网为线与类型。因此，若在此指令后面的任何模块中没有说明的连线，则假定该线网为线与类型。

4. `include

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

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

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

5. `resetall

该编译器指令将所有的编译指令重新设置为默认值。

```
`resetall
```

例如：该指令使得默认连线类型为线网类型。

6. `timescale

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

```
`timescale time_unit / time_precision
```

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

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

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

```
`timescale 1ns /100ps
module AndFunc (Z, A, B);
    output Z;
    input A, B;
        and # (5.22, 6.17 ) Al (Z, A, B);        //规定了上升及下降时延值
endmodule
```

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

```
"'timescale 10ns /1ns"
```

代替上例中的编译器指令，则 5.22 对应 52ns，6.17 对应 62ns。

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

```
`timescale 1ns /100ps
module AndFunc (Z, A, B);
    output Z;
    input A, B;
        and # (5.22,6.17 ) Al (Z, A, B);
endmodule
`timescale 10ns/ 1ns
module TB;
    reg PutA, PutB;
    wire GetO;
    initial
    begin
        PutA = 0;
        PutB = 0;
        #5.21 PutB = 1;
        #10.4 PutA = 1;
        #15 PutB = 0;
    end
    AndFunc AF1(GetO, PutA, PutB);
endmodule
```

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

7. `unconnected_drive 和`nounconnected_drive

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

```
`unconnected_drive pull1
. . .
/* 在这两个程序指令间的所有未连接的输入端口为正偏电路状态(连接到高电平)*/
`nounconnected_drive
`unconnected_drive pull0
. . .
/* 在这两个程序指令间的所有未连接的输入端口为反偏电路状态(连接到低电平)*/
`nounconnected_drive
```

8. `celldefine 和`endcelldefine

这两个程序指令用于将模块标记为单元模块。它们表示包含模块定义，如下例所示：

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

某些 PLI 例程使用单元模块。

## 3.4 Verilog HDL 的建模

### 3.4.1 门级元件

Verilog HDL 中提供下列内置基本门元件：

（1）多输入门元件，如 and、nand、or、nor、xor、xnor；

（2）多输出门元件，如 buf、not；

(3) 三态门,如 bufif0、bufif1、notif0、notif1;

(4) 上拉、下拉电阻,如 pullup、pulldown;

(5) MOS 开关,如 cmos、pmos、nmos、rcmos、rnmos、rpmos;

(6) 双向开关,如 tran、tranif0、tranif1、rtran、rtranif0、rtranif1。

门级逻辑设计描述中可使用具体的门实例语句。下面是简单的门实例语句的格式:

```
gate_type [instance_name] (term1, term2, . . . ,termN);
```

**注意**:instance_name 是可选的;gate_type 为前面列出的某种门类型;termN 用于表示与门的输入/输出端口相连的线网或寄存器。

同一类型门的多个实例能够在一个结构形式中定义。语法如下:

```
gate_type [instance_name1] (term11, term12, . . .,term1N),
          [instance_name2] (term21, term22, . . .,term2N),
          . . .
          [instance_nameM] (termM1, termM2, . . .,termMN);
```

1. 多输入门

内置的多输入门包括 and、nand、or、nor、xor 和 xnor。这些逻辑门只有单一输出,一个或多个输入。多输入门实例语句的语法如下:

```
multiple_input_gate_type  [instance_name] (OutputA, Input1, Input2, . . .,InputN);
```

第一个端口是输出,其他端口是输入。

下面是几个具体实例:

```
and A1(Out1, In1, In2);
and RBX (Sty, Rib, Bro, Qit, Fix);
xor (Bar,Bud[0],Bud[1],Bud[2]),(Car,Cut[0],Cut[1]),(Sar,Sut[2],Sut[1],Sut[0],Sut[3]);
```

第一个门实例语句是单元名为 A1,输出为 Out1,并带有两个输入 In1 和 In2 的双输入与门。第二个门实例语句是四输入与门,单元名为 RBX,输出为 Sty,4 个输入为 Rib、Bro、Qit 和 Fix。第三个门实例语句是异或门的具体实例,没有单元名。它的输出是 Bar,三个输入分别为 Bud[0]、Bud[1]和 Bud[2]。同时,这一个实例语句中还有两个相同类型的单元。

2. 多输出门

多输出门包括 buf 和 not。这些门都只有单个输入,一个或多个输出。这些门的实例语句的基本语法如下:

```
multiple_output_gate_type  [instance_name] (Out1, Out2, . . . OutN ,InputA);
```

最后的端口是输入端口,其余的所有端口为输出端口。

例如:

```
buf B1(Fan [0],Fan [1],Fan [2],Fan [3],Clk);
not N1(PhA,PhB,Ready);
```

在第一个门实例语句中，Clk 是缓冲门的输入。门 B1 有 4 个输出：Fan[0]～Fan[3]。在第二个门实例语句中，Ready 是非门的唯一输入端口。门 N1 有两个输出：PhA 和 PhB。

3. 三态门

三态门包括 bufif0、bufif1、notif0 以及 notif1。这些门用于对三态驱动器建模。它们含有一个输出、一个数据输入和一个控制输入。三态门实例语句的基本语法如下：

```
tristate_gate [instance_name] (OutputA,InputB,ControlC);
```

第一个端口 OutputA 是输出，第二个端口 InputB 是数据输入，ControlC 是控制输入。根据控制输入，输出可被驱动到高阻状态，即值 z。对于 bufif0，若控制输入为 1，则输出为 z；否则数据将传输至输出端。对于 bufif1，若控制输入为 0，则输出为 z。对于 notif0，若控制输出为 1，则输出为 z；否则，输入数据值的非将传输到输出端。对于 notif1，若控制输入为 0；则输出为 z。

例如：

```
bufif1 BF1(Dbus,MemData,Strobe);
notif0 NT2 (Addr, Abus, Probe);
```

第一个实例中，当 Strobe 为 0 时，bufif1 门 BF1 驱动输出 Dbus 为高阻；否则，MemData 被传输至 Dbus。在第二个实例语句中，当 Probe 为 1 时，Addr 为高阻；否则，Abus 的非被传输到 Addr。

4. 上拉、下拉电阻

上拉电阻和下拉电阻包括 pullup 和 pulldown。这类门设备没有输入只有输出。上拉电阻将输出置为 1。下拉电阻将输出置为 0。门实例语句形式如下：

```
pull_gate[instance_name] (OutputA);
```

门实例的端口表只包含 1 个输出。例如：

```
pullup PUP (Pwr);
```

此上拉电阻实例名为 PUP，输出 Pwr 置为高电平 1。

5. MOS 开关

MOS 开关包括 cmos、pmos、nmos、rcmos、rpmos 和 rnmos。这类门用于为单向开关建模。即数据从输入流向输出，并且可以通过设置合适的控制输入关闭数据流。

pmos(PMOS 管)、nmos(NMOS 管)，rnmos(r 代表电阻)和 rpmos 开关有一个输出、一个输入和一个控制输入。实例的基本语法如下：

```
gate_type[instance_name] (OutputA, InputB, ControlC);
```

第一个端口为输出，第二个端口是输入，第三个端口是控制输入端。当 nmos 和 rnmos 开关的控制输入为 0 时(pmos 和 rpmos 开关的控制为 1 时)，开关关闭，即输出为 z；当控制为 1 时，输入数据传输至输出。与 nmos 和 pmos 相比，rnmos 和 rpmos 在输入引线和输出引线之间存在高阻抗(电阻)。因此，当数据从输入传输至输出时，对于 rpmos 和 rmos 存在数据信号强度衰减。

例如：

```
pmos P1 (BigBus, SmallBus, GateControl);
rnmos RN1 (ControlBit, ReadyBit, Hold);
```

第一个实例为一个实例名为 P1 的 pmos 开关。开关的输入为 SmallBus，输出为 BigBus，控制信号为 GateControl。

这两个开关实例语句的语法形式如下：

```
(r)cmos [instance_name] (OutputA, InputB, Ncontrol, Pcontrol);
```

第一个端口为输出端口，第二个端口为输入端口，第三个端口为 N 通道控制输入，第四个端口为是 P 通道控制输入。cmos(rcmos)开关行为与带有公共输入、输出的 pmos(rpmos)和 nmos(rnmos)开关组合十分相似。

6. 双向开关

双向开关包括 tran、tranif0、tranif1、rtran、rtranif0 和 rtranif1。这些开关是双向的，即数据可以双向流动，并且当数据在开关中传播时没有时延。tranif0、rtranif0、tranif1、rtranif1 开关能够通过设置合适的控制信号来关闭，tran 和 rtran 开关不能被关闭。

tran 或 rtran(tran 的高阻态版本)开关实例语句的语法如下：

```
(r)tran [instance_name] (SignalA,SignalB );
```

端口表只有两个端口，并且无条件地双向流动，即从 SignalA 至 SignalB；反之亦然。

其他双向开关的实例语句的语法如下：

```
gate_type [instance_name] (SignalA, SignalB, ControlC);
```

前两个端口是双向端口，即数据从 SignalA 流向 SignalB；反之亦然。第三个端口是控制信号。如果对 tranif0 和 tranif0，ControlC 是 1；对 tranif1 和 rtranif1，ControlC 是 0；反之禁止双向数据流动。对于 rtran、rtranif0 和 rtranif1，当信号通过开关传输时，信号强度减弱。

7. 门时延

可以使用门时延定义门从任何输入到其输出的信号传输时延。门时延可以在门自身实例语句中定义。带有时延定义的门实例语句的语法如下：

```
gate_type [delay][instance_name](terminal_list);
```

时延规定了门时延，即从门的任意输入到输出的传输时延。当没有强调门时延时，默认的时延值为0。

门时延由三类时延值组成，即上升时延、下降时延和截止时延。

门时延定义可以包含0个、1个、2个或3个时延值。表3-13为不同个数时延值说明条件下，各种具体的时延取值。

**表3-13 各种具体的时延取值**

| 时延属性 | 无时延 | 1个时延(d) | 2个时延(d1,d2) | 3个时延(dA,dB,dC) |
|---|---|---|---|---|
| 上升 | 0 | d | d1 | dA |
| 下降 | 0 | d | d2 | dB |
| to_x | 0 | d | min(d1, d2) | min(dA, dB, dC) |
| 截止 | 0 | d | min(d1, d2) | dC |

**注意**：转换到x的时延(to_x)不但被明确地定义，还可以通过其他定义的值决定。

下面是一些具体实例。注意Verilog HDL模型中的所有时延都以单位时间表示。单位时间与实际时间的关联可以通过'timescale'编译器指令实现。

例如，非门

```
not N1 (Qbar, Q);
```

因为在非门N1中没有定义时延，门时延为0。

例如，与非门

```
nand #6 (Out, In1, In2);
```

在该与非门实例中，所有时延均为6，即上升时延和下降时延都是6。因为输出绝不会是高阻态，截止时延不适用于与非门，转换到x的时延也是6。

例如，与门

```
and#(3,5) (Out, In1, In2, In3);
```

在与门实例中，上升时延被定义为3，下降时延为5，转换到x的时延是3和5中间的最小值，即3。

例如，三态门

```
notif1#(2,8,6) (Dout, Din1, Din2);
```

在三态门实例中，上升时延为2，下降时延为8，截止时延为6，转换到x的时延是2、8和6中的最小值，即2。

对多输入门(如与门和非门)和多输出门(如缓冲门和非门)总共只能够定义2个时延(因为输出绝不会是z)。三态门共有3个时延，并且上拉电阻和下拉电阻实例门不能有任何时延。时延定义形式如下：

```
min:typ:max
```

门时延也可采用 min:typ:max 形式定义：

```
minimum: typical: maximum
```

最小值、典型值和最大值必须是常数表达式。下面是使用这种形式的实例：

```
nand#(2:3:4, 5:6:7) (Pout, Pin1, Pin2);
```

选择使用哪种时延通常作为模拟运行中的一个选项。例如，如果执行最大时延模拟，与非门单元使用上升时延 4 和下降时延 7。

此外，程序块也能够定义门时延。

### 3.4.2 数据流建模

用数据流建模最基本的机制就是使用连续赋值语句。在连续赋值语句中，某个值被赋给线网变量。连续赋值语句的语法为

```
wire[位宽说明] 变量名 1,变量名 2,…,变量名 n;
assign [delay]变量名 = 表达式;
```

右边表达式使用的操作数无论何时发生变化，右边表达式都重新计算，并且在指定的时延后变化值被赋予左边表达式的线网变量。时延定义了右边表达式操作数变化与赋值给左边表达式之间的持续时间。如果没有定义时延值，默认时延为 0。

**注意**：*在 assign 语句中，左边变量的数据类型必须是 wire 型。*

下面的例子是使用数据流建模对 2-4 解码器电路建模的实例模型：

```
`timescale 1ns /1ns
module Decoder2x4 (A, B, EN, Z);
    input A, B, EN;
    output [0:3] Z;
    wire Abar, Bbar;
        assign #1 Abar = ~ A;                          //语句 1
        assign #1 Bbar = ~ B;                          //语句 2
        assign #2 Z[0] = ~ (Abar & Bbar & EN) ;        //语句 3
        assign #2 Z[1] = ~ (Abar & B & EN) ;           //语句 4
        assign #2 Z[2] = ~ (A & Bbar & EN) ;           //语句 5
        assign #2 Z[3] = ~ (A & B & EN) ;              //语句 6
endmodule
```

以反引号“`”开始的第一条语句是编译器指令，编译器指令`timescale 将模块中所有时延的单位设置为 1ns，时间精度为 1ns。例如，在连续赋值语句中时延值 #1 和 #2 分别对应时延 1ns 和 2ns。

模块 Decoder2×4 有 3 个输入端口和 1 个 4 位输出端口。线网类型说明了两个连线

型变量 Abar 和 Bbar(连线类型是线网类型的一种)。此外,模块包含 6 个连续赋值语句。

当 EN 在 5ns 时变化,语句 3、4、5 和 6 执行。这是因为 EN 是这些连续赋值语句中右边表达式的操作数。Z[0]在第 7ns 时被赋予新值 0。当 A 在 15ns 时变化,语句 1、5 和 6 执行。执行语句 5 和 6 不影响 Z[0]和 Z[1]的取值。执行语句 5 导致 Z[2]值在 17ns 时变为 0。执行语句 1 导致 Abar 在 16ns 时被重新赋值。由于 Abar 的改变,反过来又导致 Z[0]值在 18ns 时变为 1。

注意连续赋值语句对电路的数据流行为建模的方式,该建模方式是隐式而非显式的。此外,连续赋值语句是并发执行的,也就是说各语句的执行顺序与其在描述中出现的顺序无关。

### 3.4.3 行为级建模

行为级建模用于描述数字逻辑电路的功能和算法。在 Verilog HDL 中,行为级建模主要使用由关键词 initial 或 always 定义的两种结构类型的描述语句。

(1) initial 语句:此语句只执行一次。initial 语句主要是一条面向仿真的过程语句,不能用来描述硬件逻辑电路的功能。

(2) always 语句:此语句总是循环执行,或者说此语句重复执行。

只有寄存器类型数据能够在这两种语句中被赋值。寄存器类型数据在被赋新值前保持原有值不变。所有的初始化语句和 always 语句在 0 时刻并发执行。

下例应用 always 语句对 1 位全加器电路进行建模:

```
module FA_Seq (A, B, Cin, Sum, Cout);
    input A, B, Cin;
    output Sum, Cout;
    reg Sum, Cout;
    reg T1, T2, T3;
    always@( A or B or Cin )
        begin
          Sum = (A ^ B) ^ Cin;
          T1 = A & Cin;
          T2 = B & Cin;
          T3 = A & B;
          Cout = (T1| T2) | T3;
        end
endmodule
```

模块 FA_Seq 有三个输入和两个输出。由于 Sum、Cout、T1、T2 和 T3 在 always 语句中被赋值,它们被声明为 reg 类型(reg 是寄存器数据类型的一种)。always 语句中有一个与事件控制(紧跟在字符@后面的表达式)相关联的顺序过程(begin-end 对)。这意味着只要 A、B 或 Cin 上发生事件,即 A、B 或 Cin 之一的值发生变化,顺序过程就执行。在顺序过程中的语句顺序执行,并且在顺序过程执行结束后被挂起。顺序过程执行完成后,always 语句再次等待 A、B 或 Cin 上发生的事件。

在顺序过程中出现的语句是过程赋值模块化的实例。模块化过程赋值在下一条语句执行前完成执行。过程赋值可以有一个可选的时延。

时延可以分两种类型：语句间时延，描述时延语句执行的时延；语句内时延，描述右边表达式数值计算与左边表达式赋值间的时延。

下面是语句间时延的示例：

```
Sum = (A ^ B) ^ Cin;
#4 T1 = A & Cin;
```

在第二条语句中的时延规定赋值时延 4 个时间单位执行。也就是说，在第一条语句执行后等待 4 个时间单位，然后执行第二条语句。下面是语句内时延的示例：

```
Sum = #3 (A^ B) ^ Cin;
```

这个赋值中的时延意味着：首先计算右边表达式的值，等待 3 个时间单位，然后赋值给 Sum。

如果在过程赋值中未定义时延，默认值为 0 时延，也就是说，赋值立即发生。

下面是 initial 语句的示例：

```
`timescale 1ns / 1ns
module Test (Pop, Pid);
    output Pop, Pid;
    reg Pop, Pid;
    initial
    begin
        Pop = 0;                    // 语句 1
        Pid = 0;                    // 语句 2
        Pop = #5 1;                 // 语句 3
        Pid = #3 1;                 // 语句 4
        Pop = #6 0;                 // 语句 5
        Pid = #2 0;                 // 语句 6
    end
endmodule
```

initial 语句包含一个顺序过程。这一顺序过程在 0ns 时开始执行，并且在顺序过程中所有语句全部执行完毕后，initial 语句永久挂起。这一顺序过程包含带有定义语句内时延的分组过程赋值的实例。语句 1 和 2 在 0ns 时执行。语句 3 也在 0 时刻执行，导致 Pop 在 5ns 时被赋值。语句 4 在 5ns 执行，并且 Pid 在 8ns 时被赋值。同样，Pop 在 14ns 时被赋值 0，Pid 在 16ns 时被赋值 0。语句 6 执行后，initial 语句永久被挂起。

### 3.4.4 结构化建模

在 Verilog HDL 中可使用如下方式描述结构：

(1) 内置门基本元件(门级)；

（2）开关级基本元件（晶体管级）；

（3）用户定义的基本元件（门级）；

（4）模块实例（创建层次结构）。

各种基本元件通过线网实现相互连接，构成一个逻辑功能单元。下面的实例使用内置门基本元件描述全加器电路的结构，如图 3-6 所示。

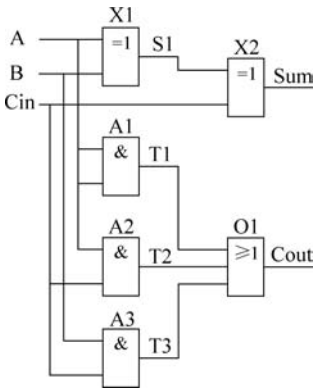


图 3-6　全加器电路

```
module FA_Str (A, B, Cin, Sum, Cout);
    input A, B, Cin;
    output Sum, Cout;
    wire S1, T1, T2, T3;
      xor X1 (S1, A, B), X2 (Sum, S1, Cin);
      and A1 (T3, A, B),A2 (T2, B, Cin),A3 (T1, A, Cin);
      or  O1 (Cout, T1, T2, T3);
endmodule
```

在这一实例中，模块包含门的实例语句，也就是说包含内置门 xor、and 和 or 的实例语句。门实例由线网类型变量 S1、T1、T2 和 T3 互连。由于没有指定顺序，门实例语句可以以任何顺序出现；对电路功能以纯结构的方式进行描述，xor、and 和 or 是内置门原语；X1、X2、A1 等是实例名称。紧跟在每个门后的信号列表是它的互连；列表中的第一个是门输出，余下的是输入。例如，S1 与 xor 门实例 X1 的输出连接，而 A 和 B 与实例 X1 的输入连接。

4 位全加器可以使用 4 个 1 位全加器模块描述。下面是 4 位全加器的结构描述形式：

```
module FourBitFA (FA, FB, FCin, FSum,FCout );
    parameter SIZE = 4;
    input [SIZE:1] FA, FB;
    output [SIZE:1] FSum
    input FCin;
    input FCout;
    wire [ 1: SIZE - 1] FTemp;
    FA_Str
    FA1( .A (FA[1]), .B(FB[1]), .Cin(FCin),.Sum(FSum[1]), .Cout(FTemp[1]));
    FA2( .A (FA[2]), .B(FB[2]), .Cin(FTemp[1]),.Sum(FSum[2]), .Cout(FTemp[2]));
    FA3(FA[3], FB[3], FTemp[2], FSum[3], FTemp[3]);
    FA4(FA[4], FB[4], FTemp[3], FSum[4], FCout);
endmodule
```

在这一实例中，模块 FourBitFA 引用由模块 FA_Str 定义的实例部件 FA，FourBitFA 是上层模块，模块 FA_Str 称为子模块。在实例部件 FA 中，带“.”表示被引用模块的端口，名称必须与被引用模块 FA_Str 的端口定义一致，小括号中表示在本模块中与之连接的线路。在模块实例语句中，端口可以与名称或位置关联。前两个实例 FA1 和 FA2 使用命名关联方式，也就是说，端口的名称和它连接的线网被显式描述。最后两

个实例语句，实例 FA3 和 FA4 使用位置关联方式将端口与线网关联。这里关联的顺序很重要，例如，在实例 FA4 中，第一个 FA[4]与 FA_Str 的端口 A 连接，第二个 FB[4]与 FA_Str 的端口 B 连接，余下的以此类推。

### 3.4.5 混合设计描述方式

在模块中，结构化描述和行为描述可以自由混合使用。也就是说，模块描述中可以包含实例化的门、模块实例化语句、连续赋值语句以及 always 语句和 initial 语句的混合。它们之间可以相互包含。来自 always 语句和 initial 语句（只有寄存器类型数据可以在这两种语句中赋值）的值能够驱动门或开关，而来自门或连续赋值语句（只能驱动线网）的值能够反过来用于触发 always 语句和 initial 语句。

下面是混合设计方式的 1 位全加器实例：

```
module FA_Mix (A, B, Cin, Sum, Cout);
    inputA,B, Cin;
    output Sum, Cout;
    reg Cout;
    reg T1, T2, T3;
    wire S1;
      xor X1(S1, A, B);              //门实例语句
    always@( A or B or Cin )
    begin
      T1 = A & Cin;
      T2 = B & Cin;
      T3 = A & B;
      Cout = (T1| T2) | T3;
    end
      assign Sum = S1^ Cin;          //连续赋值语句
endmodule
```

只要 A 或 B 上有事件发生，就执行门实例语句。只要 A、B 或 Cin 上有事件发生，就执行 always 语句，并且只要 S1 或 Cin 上有事件发生，就执行连续赋值语句。

## 3.5 基于 Verilog HDL 的数字电路基本设计

### 3.5.1 简单组合逻辑设计

下例是一个可综合的数据比较器，很容易看出它的功能是比较数据 a 与数据 b，若两个数据相同，则给出结果 1，否则给出结果 0。在 Verilog HDL 中，描述组合逻辑时常使用 assign 结构。注意 equal=(a==b)?1:0，这是一种在组合逻辑实现分支判断时常用的格式。

```
//-------------- compare.v ------------------
module compare(equal,a,b);
```

```
  input a,b;
  output equal;
  reg equal;
  initial
  begin
    assign equal = (a == b)?1:0;       //a 等于 b 时,equal 输出为 1; a 不等于 b 时,
                                       //equal 输出为 0
  end
endmodule
```

测试模块用于检测模块设计得正确与否,它给出模块的输入信号,观察模块的内部信号和输出信号,若发现结果与预期的有所偏差,则要对设计模块进行修改。

测试模块源代码:

```
`timescale 1ns /1ns          //定义时间单位
`include"./compare.v"        //包含模块文件,在有的仿真调试环境中并不需要此语句,
                             //而需要从调试环境的菜单中输入有关模块文件的路径和名称
module compare_test;
    reg a,b;
    wire equal;
    initial                  //initial 常用于在仿真时给出信号
        begin
              a = 0;
              b = 0;
            #100 a = 0; b = 1;
            #100 a = 1; b = 1;
            #100 a = 1; b = 0;
            #100 $stop;                        //系统任务,暂停仿真以便观察仿真波形
        end
    compare compare1(.equal(equal),.a(a),.b(b));   //调用模块
endmodule
```

比较器仿真波形(部分)如图 3-7 所示。

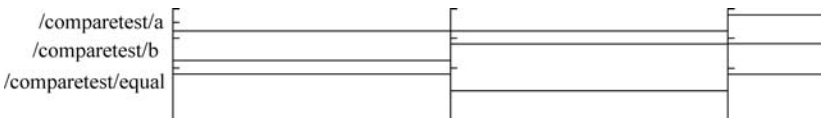


图 3-7　比较器仿真波形(部分)

### 3.5.2　简单时序逻辑电路的设计

在 Verilog HDL 中,相对于组合逻辑电路,时序逻辑电路也有规定的表述方式。在可综合的 Verilog HDL 模型中,通常使用 always 块和@(posedge clk)或@(negedge clk)的结构来表述时序逻辑。下面是 1/2 分频器的可综合模型:

```
//--------------- half_clk.v ------------------
module half_clk(reset,clk_in,clk_out);
```

```
input clk_in,reset;
output clk_out;
reg clk_out;
always@(posedge clk_in)
begin
    if(!reset) clk_out = 0;
    else clk_out = ~clk_out;
end
endmodule
```

在 always 块中，被赋值的信号都必须定义为 reg 型，这是由时序逻辑电路的特点决定的。对于 reg 型数据，如果未对它进行赋值，仿真工具会认为它是不定态。为了能正确地观察到仿真结果，在可综合的模块中通常定义一个复位信号 reset，当 reset 为低电平时，对电路中的寄存器进行复位。

测试模块的源代码：

```
//--------------- test half_clk clk_Top.v ------------------
`timescale 1ns/100ps
`define clk_cycle 50
module clk_test_Top
    reg clk, reset;
    wire clk_out;
    always #`clk_cycle clk = ~clk;
    initial
    begin
            clk = 0;
            reset = 1;
            #100 reset = 0;
            #100 reset = 1;
            #10000 $ stop;
    end
    half_clk half_clk(.reset(reset),.clk_in(clk),.clk_out(clk_out));
endmodule
```

1/2 分频器仿真波形如图 3-8 所示。

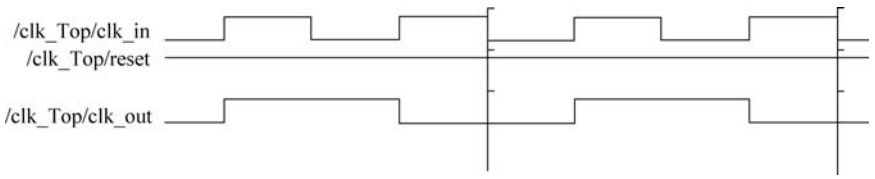


图 3-8　1/2 分频器仿真波形

### 3.5.3　利用条件语句实现较复杂的时序逻辑电路

与常用的高级程序语言一样，为了描述较为复杂的时序关系，Verilog HDL 提供了条件语句供分支判断时使用。在可综合的 Verilog HDL 模型中常用的条件语句有 if…else

和 case…endcase 两种结构，用法与 C 程序语言中类似。二者比较而言，if…else 用于不太复杂的分支关系，实际编写可综合的模块，特别是用状态机构成的模块时，更常用的是 case…endcase 风格的代码。下面给出有关 if…else 的范例。

范例设计的是一个可综合的分频器，能够将 10MHz 的时钟分频为 500kHz 的时钟。其基本原理与 1/2 分频器是一样的，但是需要定义一个计数器，以便准确获得 1/20 分频模块。具体如下：

```
//---------------- fdivision.v ------------------
module fdivision(RESET,F10M,F500K);
    input F10M,RESET;
    output F500K;
    reg F500K;
    reg [7:0]j;
always@(posedge F10M)
        if(!RESET)                    //低电平复位
            begin
                F500K <= 0;
                j <= 0;
            end
        else
begin
    if(j == 19)                       //对计数器进行判断,以确定 F500K 信号是否反转
        begin
            j <= 0;
                F500K <= ~F500K;
            end
    else
        j <= j + 1;
    end
endmodule
```

测试模块源代码：

```
//---------------- fdivision_Top.v ------------------
`timescale 1ns/100ps
`define clk_cycle 50
module division_Top;
    reg F10M_clk,RESET;
    wire F500K_clk;
    always #`clk_cycle F10M_clk = ~ F10M_clk;
    initial
    begin
        RESET = 1;
        F10M = 0;
        #100 RESET = 0;
        #100 RESET = 1;
        #10000 $stop;
    end
```

```
    fdivision fdivision(.RESET(RESET),.F10M(F10M_clk),.F500K(F500K_clk));
endmodule
```

10MHz 信号 2 分频器的仿真波形如图 3-9 所示。

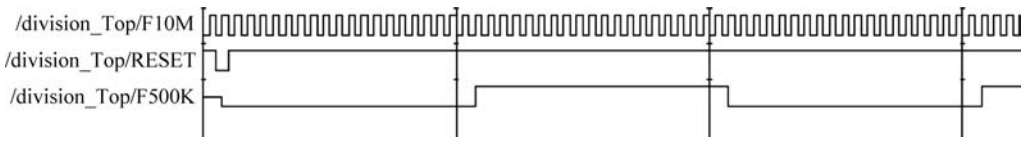


图 3-9　10MHz 信号 2 分频器的仿真波形

### 3.5.4　设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别

在 always 块中，阻塞赋值可以理解为赋值语句是顺序执行的，而非阻塞赋值可以理解为赋值语句是并发执行的。实际的时序逻辑设计中，一般情况下更多地使用非阻塞赋值语句，有时为了在同一周期实现相互关联的操作，也使用阻塞赋值语句。需要注意，在实现组合逻辑的 assign 结构中都必须采用阻塞赋值语句。下例通过分别采用阻塞赋值语句和非阻塞赋值语句的看上去非常相似的两个模块 blocking.v 和 non_blocking.v 来阐明两者之间的区别：

```
//--------------- blocking.v ------------------
module blocking(clk,a,b,c);
    output [3:0]b,c;
    input [3:0] a;
    input clk;
    reg [3:0]b,c;
    always@(posedge clk)
    begin
        b = a;
        c = b;
        $display("Blocking: a = %d, b = %d, c = %d.",a,b,c);
    end
endmodule
//--------------- non_blocking.v ------------------
module non_blocking(clk,a,b,c);
    output [3:0]b,c;
    input [3:0] a;
    input clk;
    reg [3:0]b,c;
    always@(posedge clk)
    begin
        b <= a;
        c <= b;
        $display("Non_Blocking: a = %d, b = %d, c = %d.",a,b,c);
    end
```

```
endmodule
```

测试模块源代码：

```
//--------------- compareTop.v ------------------
`timescale 1ns/100ps
`include"./blocking.v"
`include"./non_blocking.v"
module compareTop;
    wire [3:0] b1,c1,b2,c2;
    reg [3:0] a;
    reg clk;
    initial
    begin
        clk = 0;
        forever #50 clk = ~clk;
    end
    initial
    begin
        a = 4'h3;
        $display("_____");
        # 100 a = 4'h7;
        $display("_____");
        # 100 a = 4'hf;
        $display("_____");
        # 100 a = 4'ha;
        $display("_____");
        # 100 a = 4'h2;
        $display("_____");
        # 100
        $display("_____");
        $stop;
    end
    non_blocking non_blocking(clk,a,b2,c2);
    blocking blocking(clk,a,b1,c1);
endmodule
```

两种赋值方式的程序的仿真波形比较如图 3-10 所示。

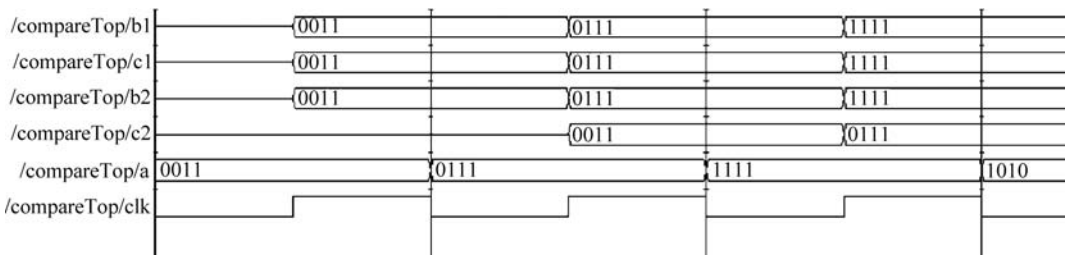


图 3-10　两种赋值方式的程序的仿真波形比较

### 3.5.5 用 always 块实现较复杂的组合逻辑电路

仅使用 assign 结构来实现组合逻辑电路，在设计中会发现很多地方会显得冗长且效率低下。而适当地采用 always 块来设计组合逻辑，往往会更具实效。

下面是一个简单的指令译码电路的设计示例。该电路通过对指令的判断，对输入数据执行相应的操作，包括加、减、与、或和求反，并且无论是指令作用的数据还是指令本身发生变化，结果都要做出及时的反应。显然，这是一个较为复杂的组合逻辑电路，采用 assign 语句表达非常复杂。示例中使用了电平敏感的 always 块，电平敏感的触发条件是指在@后的括号内电平列表中的任何一个电平发生变化（与时序逻辑不同，它在@后的括号内没有边沿敏感关键字，如 posedge 或 negedge）就能触发 always 块的动作，并且运用了 case 结构来进行分支判断，不但设计思想得到直观的体现，而且代码看起来非常整齐、便于理解。

```
//--------------- alu.v ------------------
`define plus 3'd0
`define minus 3'd1
`define band 3'd2
`define bor 3'd3
`define unegate 3'd4
module alu(out,opcode,a,b);
    output[7:0] out;
    reg[7:0] out;
    input[2:0] opcode;
    input[7:0] a,b;                    //操作数
    always@(opcode or a or b)          //电平敏感的 always 块
    begin
        case(opcode)
            `plus: out = a + b;        //加操作
            `minus: out = a - b;       //减操作
            `band: out = a&b;          //求与
            `bor: out = a|b;           //求或
            `unegate: out = ~a;        //求反
            default: out = 8'hx;       //未收到指令时，输出任意态
        endcase
    end
endmodule
```

同一组合逻辑电路分别用 always 块和连续赋值语句 assign 描述时，代码的形式差别很大，但是在 always 中适当运用 default（在 case 结构中）和 else（在 if…else 结构中），通常可以综合为纯组合逻辑，尽管被赋值的变量一定要定义为 reg 型。不过，若不使用 default 或 else 对默认项进行说明，则易生成意想不到的锁存器，这一点应注意。

### 3.5.6 在 Verilog HDL 中使用函数

与一般的程序设计语言一样，Verilog HDL 也可使用函数以适应对不同变量采取同一运算的操作。Verilog HDL 函数在综合时被理解成具有独立运算功能的电路，每调用一次函数相当于改变这部分电路的输入以得到相应的计算结果。

下例是函数调用的一个简单示范，采用同步时钟触发运算的执行，每个 clk 时钟周期都会执行一次运算。并且在测试模块中，通过调用系统任务 $ display 在时钟的下降沿显示每次计算的结果。

```
module tryfunct(clk,n,result,reset);
    output[31:0] result;
    input[3:0] n;
    inputreset,clk;
    reg[31:0] result;
    always @(posedge clk)              //clk 的上沿触发同步运算
    begin
        if(!reset)                     //reset 为低时复位
        result <= 0;
        else
            begin
            result <= n * factorial(n)/((n*2)+1);
            end
    end
    function [31:0] factorial;         //函数定义
    input [3:0] operand;
    reg [3:0] index;
    begin
        factorial = operand ? 1 : 0;
        for(index = 2; index <= operand; index = index + 1)
        factorial = index * factorial;
    end
    endfunction
endmodule
```

测试模块源代码如下：

```
`include"./step6.v"
`timescale 1ns/100ps
`define clk_cycle 50
module tryfuctTop;
reg[3:0] n,i;
reg reset,clk;
wire[31:0] result;
initial
    begin
          n = 0;
```

```
            reset = 1;
            clk = 0;
            #100 reset = 0;
            #100 reset = 1;
            for(i = 0;i <= 15;i = i + 1)
            begin
                #200 n = i;
            end
            #100 $stop;
        end
        always #`clk_cycle clk = ~clk;
        tryfunct tryfunct(.clk(clk),.n(n),.result(result),.reset(reset));
    endmodule
```

上例中函数 factorial(n)实际上就是阶乘运算。应注意的是,在实际的设计中,不希望设计中的运算过于复杂,以免在综合后带来不可预测的后果。经常的情况是,把复杂的运算分成几个步骤,分别在不同的时钟周期完成。

阶乘运算器仿真波形如图 3-11 所示。

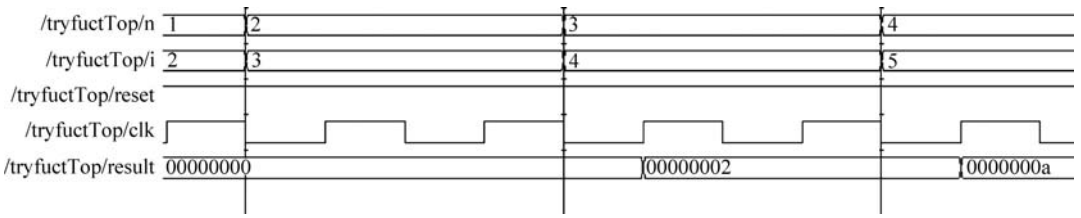


图 3-11　阶乘运算器仿真波形

### 3.5.7　在 Verilog HDL 中使用任务

仅有函数并不能完全满足 Verilog HDL 中的运算需求。当希望能够将一些信号进行运算并输出多个结果时,采用函数结构会非常不方便,而任务结构在这方面的优势十分突出。任务本身并不返回计算值,但是它通过类似 C 语言中形参与实参的数据交换,非常快捷地实现运算结果的调用。此外,还可利用任务来帮助实现结构化的模块设计,将批量的操作以任务的形式独立出来,使设计的目的非常清楚。

下面是一个利用 task 和电平敏感的 always 块设计比较后重组信号的组合逻辑的实例。可以看到,利用 task 非常方便地实现了数据之间的交换,如果要用函数实现相同的功能是非常复杂的;另外,task 也避免了直接用一般语句来描述所引起的不易理解和综合时产生冗余逻辑等问题。

模块源代码如下:

```
//--------------- sort4.v ------------------
module sort4(ra,rb,rc,rd,a,b,c,d);
    output[3:0] ra,rb,rc,rd;
```

```
    input[3:0] a,b,c,d;
    reg[3:0] ra,rb,rc,rd;
    reg[3:0] va,vb,vc,vd;
    always @ (a or b or c or d)
    begin
        {va,vb,vc,vd} = {a,b,c,d};
        sort2(va,vc);                  //va 与 vc 互换
        sort2(vb,vd);                  //vb 与 vd 互换
        sort2(va,vb);                  //va 与 vb 互换
        sort2(vc,vd);                  //vc 与 vd 互换
        sort2(vb,vc);                  //vb 与 vc 互换
        {ra,rb,rc,rd} = {va,vb,vc,vd};
end
task sort2;
      inout[3:0] x,y;
      reg[3:0] tmp;
      if(x > y)
      begin
        tmp = x;                  //x 与 y 变量的内容互换,要求顺序执行,所以采用阻塞赋值方式
            x = y;
            y = tmp;
      end
endtask
endmodule
```

应注意的是,task 中的变量定义与模块中的变量定义不尽相同,它们并不受输入、输出类型的限制。如此例,x 与 y 对于 task sort2 来说虽然是 inout 型,但实际上它们对应的是 always 块中变量,都是 reg 型变量。

测试模块源代码如下:

```
`timescale 1ns/100ps
`include "sort4.v"
module task_Top;
    reg[3:0] a,b,c,d;
    wire[3:0] ra,rb,rc,rd;
    initial
    begin
              a = 0;b = 0;c = 0;d = 0;
              repeat(5)
              begin
                  #100 a  = { $ random} % 15;
                  b  = { $ random} % 15;
                  c  = { $ random} % 15;
                  d  = { $ random} % 15;
              end
      end
      #100  $ stop;
      sort4 sort4(.a(a),.b(b),.c(c),.d(d), .ra(ra),.rb(rb),.rc(rc),.rd(rd));
endmodule
```

信号重组模块仿真波形如图 3-12 所示。

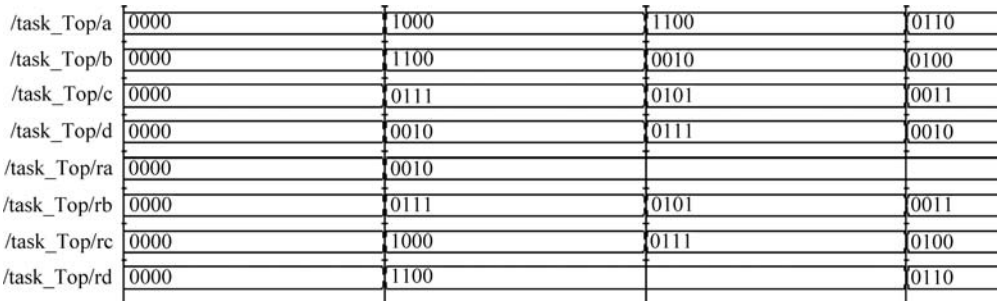


图 3-12　信号重组模块仿真波形

### 3.5.8　利用有限状态机进行复杂时序逻辑的设计

在数字电路中已经学习过通过建立有限状态机来进行数字逻辑的设计，而在 Verilog HDL 硬件描述语言中，这种设计方法得到进一步发展。通过 Verilog HDL 提供的语句，可以直观地设计出适合更为复杂的时序逻辑的电路。

下例是一个简单的状态机设计，功能是检测一个 5 位二进制序列“10010”。考虑到序列重叠的可能，有限状态机共提供 8 个状态(包括初始状态 IDLE)。

模块源代码如下：

```
//seqdet.v
module seqdet(x,z,clk,rst,state);
    inputx,clk,rst;
    output z;
    output[2:0] state;
    reg[2:0] state;
    wire z;
    parameter IDLE = 'd0, A = 'd1, B = 'd2,C = 'd3,
                  D = 'd4,E = 'd5, F = 'd6,G = 'd7;
    assign z  = ( state == E && x == 0 )? 1 : 0;   //输出为 1 的条件为(state = E 同时 x = 0 )
    always@(posedge clk)
    if(!rst)
      begin
          state <= IDLE;
      end
    else
    casex(state)
      IDLE : if(x == 1)
          begin
              state <= A;
          end
      A: if(x == 0)
          begin
```

```
            state <= B;
        end
    B: if(x == 0)
        begin
            state <= C;
        end
    else
        begin
            state <= F;
        end
    C: if(x == 1)
        begin
            state <= D;
        end
    else
        begin
            state <= G;
        end
    D: if(x == 0)
        begin
            state <= E;
        end
    else
        begin
            state <= A;
        end
    E: if(x == 0)
        begin
            state <= C;
        end
    else
        begin
            state <= A;
        end
    F: if(x == 1)
        begin
            state <= A;
        end
    else
        begin
            state <= B;
        end
    G: if(x == 1)
        begin
            state <= F;
        end
    default:state = IDLE;              //默认状态为初始状态
    endcase
```

```
endmodule
```

测试模块源代码如下：

```
//--------------- seqdet.v -----------------
`timescale 1ns/1ns
`include"./seqdet.v"
module seqdet_Top;
    regclk,rst;
    reg[23:0] data;
    wire[2:0] state;
    wirez,x;
    assign x = data[23];
    always #10 clk = ~clk;
    always@(posedge clk)
    data = {data[22:0],data[23]};
    initial
    begin
              clk = 0;
              rst = 1;
              #2 rst = 0;
              #30 rst = 1;
              data = 'b1100_1001_0000_1001_0100;
              #500 $stop;
    end
    seqdet m(x,z,clk,rst,state);
endmodule
```

序列检测器仿真波形如图 3-13 所示。

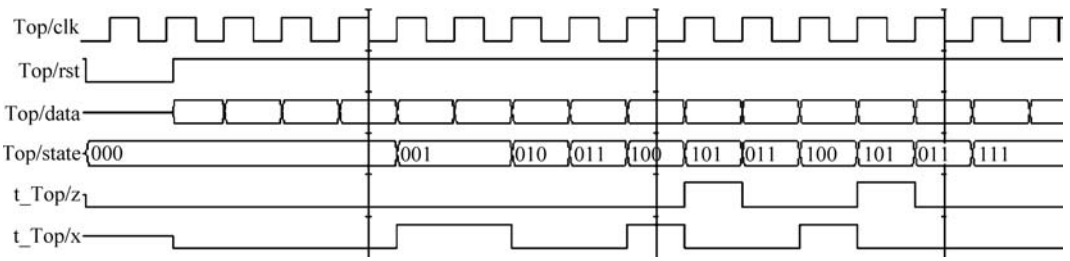


图 3-13　序列检测器仿真波形

### 3.5.9　利用状态机的嵌套实现层次结构化设计

上一个例子学习了如何使用状态机。实际上，单个有限状态机控制整个逻辑电路的运转在实际设计中是不多见的，往往是状态机套用状态机，从而形成树状的控制核心。这一点也与提倡的层次化、结构化的自顶而下的设计方法相符，下面将提供一个这样的示例以供大家学习。

该例是一个简化的紫外线可擦除只读存储器(EPROM)的串行写入器。事实上,它是由一个 EPROM 读写器设计中实现写功能的部分做删节得到的,去除了 EPROM 的启动、结束和 EPROM 控制字的写入等功能,只具备这样一个雏形。工作步骤:①地址的串行写入;②数据的串行写入;③给信号源应答,信号源给出下一个操作对象;④结束写操作。通过移位指令并行数据得以一位一位输出。

模块源代码如下:

```
module writing(reset,clk,address,data,sda,ack);
    inputreset,clk;
    input[7:0] data,address;
    output sda,ack;                    //sda 负责串行数据输出;
                                       //ack 是一个对象操作完毕后,模块给出的应答信号
    reg link_write;                    //link_write 决定何时输出
    reg[3:0] state;                    //主状态机的状态字
    reg[4:0] sh8out_state;             //从状态机的状态字
    reg[7:0] sh8out_buf;               //输入数据缓冲
    reg finish_F;                      //用以判断是否处理完一个操作对象
    reg ack;
    parameter idle = 0,addr_write = 1,data_write = 2,stop_ack = 3;
    parameter bit0 = 1,bit1 = 2,bit2 = 3,bit3 = 4,bit4 = 5,bit5 = 6,bit6 = 7,bit7 = 8;
    assign sda = link_write? sh8out_buf[7] : 1'bz;
    always@(posedge clk)
    begin
      if(!reset)                       //复位
          begin
              link_write <= 0;
              state <= idle;
              finish_F <= 0;
              sh8out_state <= idle;
              ack <= 0;
              sh8out_buf <= 0;
          end
      else
          case(state)
          idle:
              begin
                  link_write <= 0;
                  state <= idle;
                  finish_F <= 0;
                  sh8out_state <= idle;
                  ack <= 0;
                  sh8out_buf <= address;
                  state <= addr_write;
              end
      addr_write:                      //地址输入
          begin
              if(finish_F == 0)
```

```
                begin shift8_out; end
            else
                begin
                    sh8out_state <= idle;
                    sh8out_buf <= data;
                    state <= data_write;
                    finish_F <= 0;
                end
        end
    data_write:                    //数据写入
        begin
            if(finish_F == 0)
                begin shift8_out; end
            else
                begin
                    link_write <= 0;
                    state <= stop_ack;
                    finish_F <= 0;
                    ack <= 1;
                end
        end
    stop_ack:                      //完成应答
        begin
            ack <= 0;
            state <= idle;
        end
    endcase
    end
    task shift8_out;               //串行写入
        begin
        case(sh8out_state)
        idle:
            begin
                link_write <= 1;
                sh8out_state <= bit0;
            end
        bit0:
            begin
                link_write <= 1;
                sh8out_state <= bit1;
                sh8out_buf <= sh8out_buf << 1;
            end
        bit1:
            begin
                sh8out_state <= bit2;
                sh8out_buf <= sh8out_buf << 1;
            end
        bit2:
```

```
                begin
                    sh8out_state <= bit3;
                    sh8out_buf <= sh8out_buf << 1;
                end
            bit3:
                begin
                    sh8out_state <= bit4;
                    sh8out_buf <= sh8out_buf << 1;
                end
            bit4:
                begin
                    sh8out_state <= bit5;
                    sh8out_buf <= sh8out_buf << 1;
                end
            bit5:
                begin
                    sh8out_state <= bit6;
                    sh8out_buf <= sh8out_buf << 1;
                end
            bit6:
                begin
                    sh8out_state <= bit7;
                    sh8out_buf <= sh8out_buf << 1;
                end
            bit7:
                begin
                    link_write <= 0;
                    finish_F <= finish_F + 1;
                end
            endcase
        end
endtask
endmodule
```

测试模块源代码如下：

```
`timescale 1ns/100ps
`define clk_cycle 50
module writingTop;
    reg reset,clk;
    reg[7:0] data,address;
    wire ack,sda;
    always #`clk_cycle clk = ~clk;
    initial
    begin
        clk = 0;
        reset = 1;
        data = 0;
        address = 0;
```

```
            #(2*`clk_cycle) reset = 0;
            #(2*`clk_cycle) reset = 1;
            #(100*`clk_cycle) $stop;
        end
        always @(posedge ack)              //接收到应答信号后,给出下一个处理对象
            begin
                data = data + 1;
                address = address + 1;
            end
        writing writing(.reset(reset),.clk(clk),.data(data),.address(address),.ack(ack),.
sda(sda));
endmodule
```

简易 EPROM 串行写入器仿真波形如图 3-14 所示。

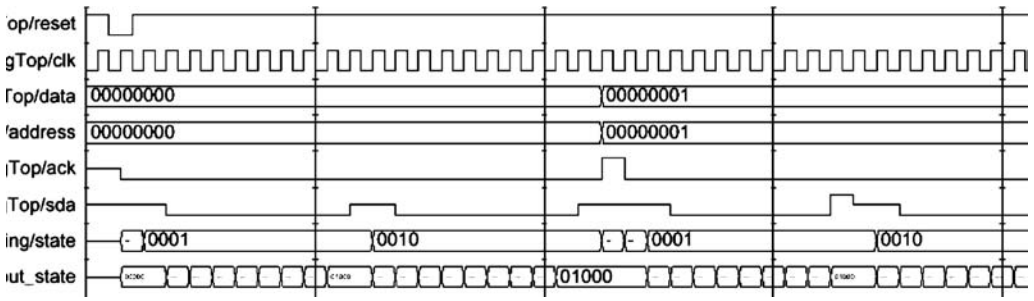


图 3-14　简易 EPROM 串行写入器仿真波形

### 3.5.10　通过模块之间的调用实现自顶向下的设计

现代硬件系统的设计过程与软件系统的开发相似,设计一个大规模的集成电路往往由模块多层次的引用和组合构成。层次化、结构化的设计过程,能使复杂的系统容易控制和调试。在 Verilog HDL 中,上层模块引用下层模块与 C 语言中程序调用有些类似,被引用的子模块在综合时作为其父模块的一部分被综合,形成相应的电路结构。在进行模块实例引用时,必须注意模块之间对应的端口,即子模块的端口与父模块的内部信号必须明确无误地一一对应,否则容易产生意想不到的后果。

下面给出的例子是设计中遇到的一个实例,其功能是将并行数据转化为串行数据送至外部电路编码,并将解码后得到的串行数据转化为并行数据交由 CPU 处理。显而易见,这实际上是两个独立的逻辑功能,分别设计为独立的模块,然后再合并为一个模块显得目的明确、层次清晰。

```
//---------------- p_to_s.v ------------------
module p_to_s(D_in,T0,data,SEND,ESC,ADD_100);
    output D_in,T0;                    // D_in 是串行输出,T0 是移位时钟并触发
                                       // CPU 中断,以确定何时给出下一个数据
    input [7:0] data;                  //并行输入数据
```

```
    input SEND,ESC,ADD_100;              //SEND、ESC共同决定是否进行并串数据
                                         //转化,ADD_100决定何时置数
    wire D_in,T0;
    reg [7:0] DATA_Q,DATA_Q_buf;
    assign T0 = ! (SEND & ESC);          //形成移位时钟
    assign D_in = DATA_Q[7];             //给出串行数据
    always @(posedge T0 or negedge ADD_100)   //ADD_100下沿置数,T0上沿移位
      begin
          if(!ADD_100)
                DATA_Q = data;
          else
            begin
                DATA_Q_buf = DATA_Q << 1;       //DATA_Q_buf作为中介,以使综合
                DATA_Q = DATA_Q_buf;  //器能识别
            end
      end
endmodule
```

在p_to_s.v中,由于移位运算虽然可综合,但并不是简单的RTL级描述,直接用DATA_Q<=DATA_Q<<1的写法在综合时会使综合器不易识别。另外,在该设计中,由于时钟T0的频率较低,所以没有像以往那样采用低电平置数,而是采用ADD_100的下降沿置数。

```
//--------------- s_to_p.v ------------------
module s_to_p(T1, data, D_out,DSC,TAKE,ADD_101);
    output T1;                           //触发CPU中断,以确定CPU何时取变换
                                         //得到的并行数据
    output [7:0] data;
    input D_out, DSC, TAKE, ADD_101;  //D_out提供输入串行数据.DSC、TAKE
                                         //共同决定何时取数
    wire [7:0] data;
    wire T1,clk2;
    reg [7:0] data_latch, data_latch_buf;
    assign clk2 = DSC & TAKE ;           //提供移位时钟
    assign T1 = !clk2;
    assign data = (!ADD_101) ? data_latch : 8'bz;
    always@(posedge clk2)
      begin
          data_latch_buf = data_latch << 1;//data_latch_buf作缓冲,使综合器
          data_latch = data_latch_buf;//能够识别
          data_latch[0] = D_out;
      end
endmodule
```

将上面的两个模块合并起来的sys.v的源代码如下:

```
//--------------- sys.v ------------------
`include"./p_to_s.v"
`include"./s_to_p.v"
modulesys(D_in,T0,T1, data, D_out,SEND,ESC,DSC,TAKE,ADD_100,ADD_101);
    input D_out,SEND,ESC,DSC,TAKE,ADD_100,ADD_101;
```

```
    inout [7:0] data;
    output D_in,T0,T1;
    p_to_s p_to_s(.D_in(D_in),.T0(T0),.data(data),
                  .SEND(SEND),.ESC(ESC),.ADD_100(ADD_100));
    s_to_p s_to_p(.T1(T1),.data(data),.D_out(D_out),
                  .DSC(DSC),.TAKE(TAKE),.ADD_101(ADD_101));
endmodule
```

测试模块源代码如下：

```
//--------------- Top test file for sys.v ------------------
`timescale 1ns/100ps
`include"./sys.v"
module Top;
    reg D_out,SEND,ESC,DSC,TAKE,ADD_100,ADD_101;
    reg[7:0] data_buf;
    wire [7:0] data;
    wire clk2;
    assign data = (ADD_101) ? data_buf : 8'bz;  //data 在 sys 中是 inout 型变量,ADD_101
                                                //控制 data 是作为输入还是进行输出
    assign clk2  = DSC && TAKE;
    initial
      begin
          SEND = 0;
          ESC = 0;
          DSC = 1;
          TAKE = 1;
          ADD_100 = 1;
          ADD_101 = 1;
      end
initial
      begin
          data_buf = 8'b10000001;
          #90 ADD_100 = 0;
          #100 ADD_100 = 1;
      end
always
      begin
          #50;
          SEND = ~SEND;
          ESC = ~ESC;
      end
initial
      begin
          #1500 ;
          SEND = 0;
          ESC = 0;
          DSC = 1;
          TAKE = 1;
```

```
            ADD_100 = 1;
            ADD_101 = 1;
            D_out = 0;
            #1150 ADD_101 = 0;
            #100 ADD_101  = 1;
            #100  $ stop;
        end
always
        begin
            #50 ;
            DSC = ~DSC;
            TAKE = ~TAKE;
        end
always@(negedge clk2) D_out = ~D_out;
sys sys(.D_in(D_in),.T0(T0),.T1(T1),.data(data),.D_out(D_out),
            .ADD_101(ADD_101), .SEND(SEND),.ESC(ESC),.DSC(DSC),
            .TAKE(TAKE),.ADD_100(ADD_100));
endmodule
```

数据串/并双向转换模块仿真波形如图 3-15 所示。

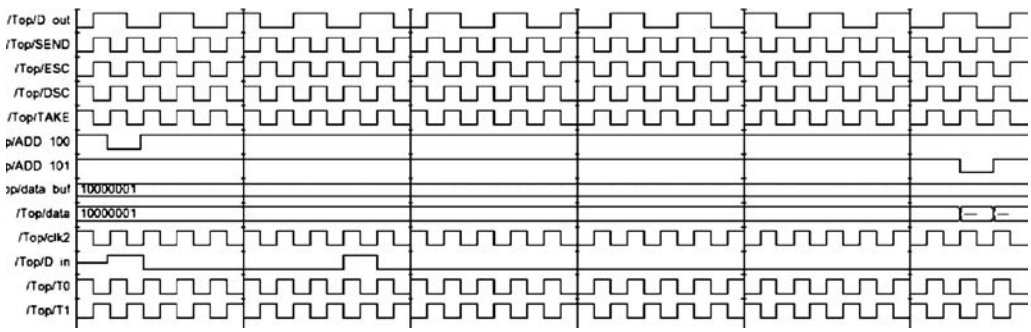


图 3-15　数据串/并双向转换模块仿真波形