第 5 章　Verilog HDL 语言基础

Verilog HDL 是目前使用最多的硬件描述语言之一，具有强大的硬件描述能力，可以从开关级、门级、RTL 级到系统级/算法级对数字系统进行建模，并支持多层次的混合描述。Verilog HDL 与其他硬件描述语言一样，独立于具体实现技术和制造工艺，使设计者可以选择不同的实现方式。本章我们将详细介绍 Verilog HDL 的语言要素和程序设计方法。

5.1 Verilog HDL 概述

Verilog HDL 是 Gateway Design Automation 公司在 1983 年开发出来的，专用于其模拟器产品的开发建模。由于该公司的仿真器 Verilog-XL 取得了巨大的成功，再加上这种语言比较实用，与 C 语言的编写风格相似，所以逐渐被众多设计者接受。Gateway Design Automation 公司在 1989 年被 Cadence 公司收购，因此 Verilog HDL 语言的所有权也归属于 Cadence 公司。

1990 年 Cadence 公司公开了 Verilog HDL，并且成立了 OVI（Open Verilog International）组织推广 Verilog HDL。经过几年的发展，Verilog HDL 于 1995 年成为 IEEE 标准，称为 IEEE STD 1364-1995。之后，标准化组织对 Verilog-1995 进行了修正和扩展，到了 2001 年，Verilog HDL 的第二个 IEEE 标准发布，称为 IEEE STD 1364-2001。

Verilog-2001 是目前的主流版本，它对 Verilog-1995 进行了重大改进，添加了诸如通用敏感列表、多维数组、生成语句块和命名端口连接等实用功能，大部分的 EDA 设计工具都支持 Verilog-2001 版本。

2005 年，IEEE 又推出了 IEEE STD 1364-2005 版本的 Verilog HDL，但这个版本只是对 Verilog-2001 进行了细微的修正。这个版本中新增加的 Verilog-AMS 可以对模拟和混合信号系统进行建模。同年，IEEE 还公布了 System Verilog 语言的 IEEE 1800-2005 标准。System Verilog 语言在 Verilog-2001 的基础上，提高了系统级设计的能力，将设计推向了系统级空间和验证级空间。

2009 年，IEEE STD 1364-2005 版本和 IEEE 1800-2005 版本合并为 IEEE 1800-2009，成为了统一的 System Verilog 硬件描述验证语言（Hardware Description and Verification Language， HDVL）。该版本在 2012 年又进行了修订。

Verilog HDL 可以在不同层次上对硬件电路进行描述，设计者可以根据任务目标、性能指标、应用场景等进行选择，也可以采用不同层次混合描述方法。设计层次从高到低可以分为系统级/算法级、RTL 级、门级和开关级，它们的主要内容如下：

（1）系统级/算法级（System/Algorithm）

该层次主要描述电路系统的行为，关注各类复杂算法的实现，是高层次、抽象的描述，通常用于建模和仿真。大多数情况下，系统级设计不能直接综合，但目前也有支持部分系统级设计的高级综合工具。目前很多综合工具可以支持算法的描述，并能直接生成 RTL 代码或门级电路网表。

（2）RTL 级（Register Transfer Level）

该层次主要描述数字逻辑电路中寄存器的行为以及寄存器之间的组合逻辑关系，是硬件描述语言主要的应用层次，可综合程度较高。

（3）门级（Gate Level）

该层次描述基本的逻辑门功能以及逻辑门之间的连接关系，以基本的逻辑门来实现整个电路的功能。

（4）开关级（Switch Level）

该层次描述器件中三极管和存储节点的功能以及它们之间的连接模型。芯片设计中，布局布线结果就是形成开关电路。在 FPGA 设计中，一般不采用开关级设计。

5.2 Verilog HDL 程序的基本结构

为了对 Verilog HDL（以下简称 Verilog）程序有一个总体认识，我们首先来看一个 Verilog 程序实例，了解组成一个 Verilog 程序的主要部分和基本语言要素。

5.2.1 Verilog HDL 程序结构说明

例 5-1 是一个模为 256 的计数器的 Verilog 程序，其结构图如图 5-1 所示，功能仿真如图 5-2 所示。该程序描述的电路模块的名字叫做 CounterHEX，CLK、RST 是输入端口，COUT 是输出端口。CLK 是时钟信号，RST 是异步复位信号。当 RST 为逻辑 0 时，计数器复位，输出端口 COUT 清零；当 RST 为逻辑 1 时，在每个 CLK 的上升沿，计数器进行计数，COUT 为计数值。

【例 5-1】 Verilog 程序的基本结构

```
module CounterHEX(CLK, RST, COUT);
  input CLK, RST;
  output [7:0] COUT;

  reg [7:0] COUTA;

  always @(posedge CLK or negedge RST)
  begin
      if (!RST) COUTA <= 0;
      else COUTA <= COUTA + 1;
  end

  assign COUT = COUTA;

endmodule
```

端口说明

数据类型说明

电路描述语句

完整电路模块

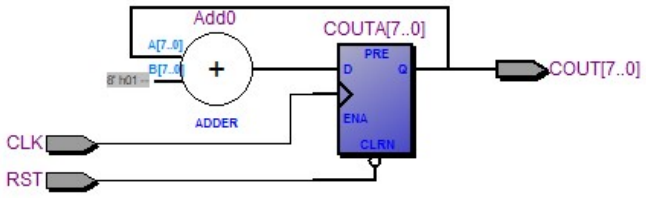


图 5-1 例 5-1 的结构图

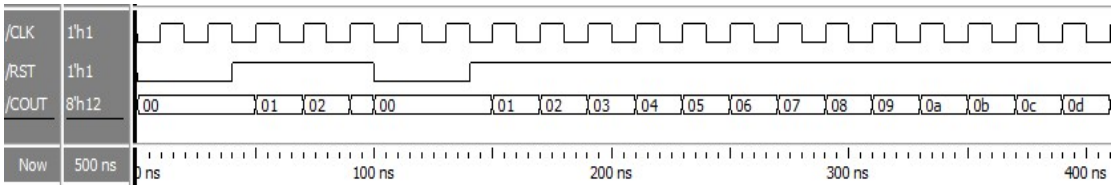


图 5-2 例 5-1 的功能仿真波形

描述一个电路模块的完整的 Verilog 程序包含在 module…endmodule 结构之中，如例 5-1 所示，其内部还包括端口说明、数据类型说明、参数说明和电路描述语句等部分。module 本身就有“模块”的意思，在这里也指电路功能模块。如果没有 module…endmodule 结构，只能表示电路片段。一个 module…endmodule 结构所描述的电路功能可以复杂，也可以简单，比如可以在一个 module…endmodule 结构中描述一个 CPU，也可以描述一个计数器。一个模块可以调用其他的模块，而多个模块也可以结合起来，构建出一个更大的模块。

模块 module…endmodule 结构的一般语法格式为：

```
module 模块名（模块端口列表）;
    端口说明;
        （input，output，inout）
    数据类型、参数、函数等说明;
        （wire，reg，parameter，function，task…）
    电路描述语句;
        （initial 语句，always 语句，例化语句等）
endmodule
```

模块定义以 module 开头，其后间隔一个或多个空格，是“模块名”，这是设计者给模块取的名字。虽然模块名可以取任意合法的标识符，但建议根据模块的具体功能来取模块名，起到提示的作用。比如全加器可以取名 full_add，模 16 计数器可以取名 counter16 等。例 5-1 中的模块名为 CounterHEX。

模块名右侧的括号中是模块端口列表，其中要列出该模块所有输入、输出端口的名称。如例 5-1 中，（CLK，RST，COUT）就是端口列表，这里有三个端口，它们之间用逗号或“or”隔开，即（CLK or RST or COUT）。端口名也是由设计者确定的，可以取任意合法的标识符。注意 Verilog 语言区分字母的大小写，即 CLK 与 clk 是不一样的标识符。模块端口列表之后还有一个分号，在书写时千万不要丢掉。

在 module 语句之后是模块的端口说明。对于一个电路模块来说，它的端口是信号进入或者输出的通道，因此要对端口的数据流动方向加以说明。端口说明有 input、output、inout 三种模式，如图 5-3 所示。

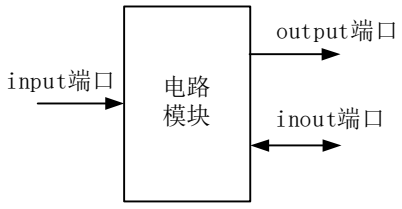


图 5-3 Verilog 端口模式

（1）input：输入端口。用 input 说明的端口为输入端口，表示数据流可以通过该类端口送入到所设计的电路模块中，但不能反向输出。从程序的角度来看，输入端口只能被读取，不能被赋值。输入端口的说明语句格式为：

input 端口名 1，端口名 2，……；

input [msb: lsb] 端口名 1，端口名 2，……；

input 是关键字，后面跟端口名。如果要说明多个相同的输入端口，可以把端口都列出来，端口名之间用逗号隔开，最后以分号结尾。上面第一个语句说明的输入端口只有一位，或称单比特端口，第二个语句说明的是多比特端口，或矢量端口，其位宽由[msb: lsb]来说明，msb 表示最高有效位，lsb 表示最低有效位。例如：

```
input CLK, RST;  //说明 CLK、RST 为一位输入端口。
input [3:0] ADDR; //说明 ADDR 为四位输入端口，可以等同于 ADDR[3]、ADDR[2]、
                    ADDR[1]、ADDR[0]四个一位端口。
input [1:8] DBUS; //说明 DBUS 为八位输入端口，注意小序号在前。
```

（2）output：输出端口。用 output 说明的端口为输出端口，表示电路模块中的数据流可以通过该类端口向外输出，但不能反向输入。从程序的角度来看，输出端口只能被赋值，不能被读取。输出端口的说明语句格式为：

output 端口名 1，端口名 2，……；

output [msb: lsb] 端口名 1，端口名 2，……；

output 是关键字，后面跟端口名。如果要说明多个相同的输出端口，可以把端口都列出来，端口名之间用逗号隔开，最后以分号结尾。上面第一个语句说明的输出端口只有一位，或称单比特端口，第二个语句说明的是多比特端口，或矢量端口，其位宽由[msb: lsb]来说明，msb 表示最高有效位，lsb 表示最低有效位。例如：

```
output SUM, Y;  //说明 SUM、Y 为一位输出端口；
output [7:0] COUT;  //说明 COUT 为八位输出端口，可以等同于 COUT[7]~COUT[0]八个一位
                      输出端口。
```

（3）inout：双向端口。用 inout 说明的端口为双向端口，表示数据流既可以通过该类端口向模块内输入，又可以向模块外输出。从程序的角度来看，双向端口既能被赋值，也能被读取。需要注意的是，双向端口的双向传输功能不是同时的，某一个时刻，该端口只能是输入或输出一种功能，比如存储器的数据线。输出端口的说明语句格式为：

inout 端口名 1，端口名 2，……；

inout [msb: lsb] 端口名 1，端口名 2，……；

inout 是关键字，后面跟端口名。如果要说明多个相同的双向端口，可以把端口都列出来，端口名之间用逗号隔开，最后以分号结尾。上面第一个语句说明的双向端口只有一位，或称单比特端口，第二个语句说明的是多比特端口，或矢量端口，其位宽由[msb: lsb]来说明，msb 表示最高有效位，lsb 表示最低有效位。例如：

```
inout A, B;  //说明 A、B 为一位双向端口；
inout [7:0] DATA;  //说明 DATA 为八位双向端口，可以等同于 DATA[7]~DATA[0]八个一位
                     双向端口。
```

根据 Verilog-2001 版本，端口说明也可以与端口列表结合起来书写，使整个程序更加简介。例如下面的端口列表和说明：

```
module DFF(CLK, D, Q);
  input CLK;
  input [3:0] D;
  output [3:0] Q;
  ……
endmodule
```

也可以写成：

```
module DFF( input CLK, input [3:0] D, output [3:0] Q );
  ……
endmodule
```

程序结构的端口说明之后是端口、变量的数据类型说明，这一部分将在下一节进行描述。在例 5-1 中，使用了一个 always 语句（过程赋值语句）和一个 assign 语句（连续赋值语句）描述电路功能，这些 Verilog 语句将在后续各小节中分类进行说明。程序最后以 endmodule 结束，末尾不加任何符号。

5.2.2 Verilog HDL 语言要素

与其他高级语言类似，Verilog 语言也是由标识符、表达式、数据类型、描述语句等语言要素组成，这些内容是构建程序的基础，而且有相应的规范。在编写程序时，要注意遵守这些语言规范。

5.2.2.1 标识符

在设计电路模块时，设计者要给电路模块、端口、变量等对象起名字，这些名字就是标识符（identifier）。在 Verilog 语言中，标识符的选取要遵守以下规则：

- 标识符可以包括的符号为 26 个英文字母（包括大小写）、数字 0~9、符号$和_（下划线），以及这些符号的组合；
- 标识符的第一个符号必须是字母或下划线；
- 标识符中不能有两个连续的下划线，单一下划线前或后必须有英文字母或数字；
- 标识符中的英文字母区分大小写；
- 标识符中的字符数不能大于 1024 个；
- 标识符不能使用 Verilog 的关键字。

以下标识符为合法标识符：

```
adder，counter8，code8_to_3，_DECODER_，REG，FOUR$
```

以下标识符为不合法标识符：

```
3to8，Mux_ _41，add8*8，reg，&read，Not-Ack
```

还有一类标识符叫做转义标识符（escaped identifier），可以包含任意可打印字符。转义标识符以反斜线符号“\“开头，以空白符结尾。空白符可以是一个空格、一个制表字符或换行符。例如：

```
\74LS04
\{*******}
\OutGate
```

因为反斜杠和空白符并不是转义标识符的一部分，所以“\OutGate”和“OutGate”是一样的。

5.2.2.2 空白符

空白符包含空格、制表字符和换行符。除了出现在字符串中之外，空白符仅用于分隔标

识符，在编译时被忽略。

Verilog 程序是自由书写格式，可以在一行内书写，也可以跨越多行书写，因此，空白符没有特殊的含义。例如下面在一行中书写的语句：

```
initial begin Data = 4'b0001;  #10 Data = 4'b0010; end
```

也可以写成：

```
initial
    begin
    Data = 4'b0001;
    #10 Data = 4'b0010;
    end
```

5.2.2.3 关键字

Verilog 语言中预先定义和使用的一些有固定含义的英文单词称为关键字（Key Word），也叫做保留字，例如 module、input、begin 等。Verilog 中的关键字必须用小写字母，如果用大写字母，如 MODULE、INPUT 等就不是关键字。关键字具有特殊含义，不能作为一般标识符来使用。Verilog-2001 版本中的关键字参见本章后面的附录。

5.2.2.4 注释

在 Verilog 程序中也可以添加注释，以方便程序阅读、理解和管理。Verilog 程序有两种注释方式：单行注释和多行注释。下面第一个例子是单行注释，采用双斜杠表示注释开始，并且注释在本行完成。第二个例子是多行注释，跨行的注释包括在 /*…*/ 之内。

```
assign Q = D; //将 D 的值赋给 Q。

assign Q = (RST)? 0 : D; /* 该语句首先判断 RST 的值。若 RST 为逻辑 1，则将 0 赋给 Q；
                            否则将 D 的值赋给 Q。*/
```

5.2.2.5 数字表达方式

Verilog 定义了四种基本的值：0、1、x 和 z，它们的含义为：

- 0：表示逻辑 0、事件判别为“伪”；
- 1：表示逻辑 1、事件判别为“真”；
- x/X：未知或不确定；
- z/Z：高阻态；

这四种基本的值是内置在 Verilog 语言中的，例如一个为 0 的值就是指逻辑 0。x 和 z 是不区分大小写的，即 x/X 都表示“不确定”，z/Z 都表示“高阻态”，0x1z 和 0X1Z 是相同的。在逻辑门或者表达式中的 z 通常解释为 x。其他的数字常量都是由以上四种基本值组成的，分为整数、实数和字符串。

1．整数

整数的书写格式有两种：简单的十进制格式和基数格式。简单的十进制格式是可以带“+”和“-”的数字序列，比如下面的语句：

```
A <= 16; //十进制数 16 转换为 32 位的二进制赋给 A。若 A 的位数少于 32 位，则高位被截掉；
B <= -25;//-25 转换成 32 位二进制补码赋给 B。若 B 的位数少于 32 位，则高位被截掉。
```

基数格式包括位宽（size）、撇号（反引号）、进制（base）和数值（value），具体形式为：

<位宽（size）> ′<进制（base）> <数值（value）>

<位宽>表示数值的二进制位数，与<进制>之间用撇号隔开。<进制>表示后面的数字采用的是哪一种进制格式。b 或 B 表示二进制，o 或 O 表示八进制，d 或 D 表示十进制，h 或 H 表示十六进制。<数值>是基于<进制>的数字序列，包括 x、z。值 x、z 以及十六进制中的 a~f 不区分大小写。当<数值>的实际位宽大于指定的<位宽>时，高位被截掉。以下是基数格式的一些实例：

```
4'b1101 //位宽为 4 的二进制数，数值为 1101
6'o025 //位宽为 6 的八进制数，转换成二进制是 010101
8'h9A //位宽为 8 的十六进制数，转换成二进制是 10011010
10'B11_0001_1011 //位宽为 10 的二进制数，数值中的下划线只起到方便阅读的作用
-6'd23 //位宽为 6 的十进制数，转换成二进制补码为 101001
7'Hz // 将 z 扩展成 7 位，即 zzzzzzz
```

当数值中的数位较多时，可以在其中插入下划线，对数位进行分隔，以方便阅读，如 32'b1000_0101_1100_0011。注意下划线不能出现在数值的最高位前，它本身没有任何意义，在编译被忽略。

x（或 z）在二进制中表示一个 x（或 z），在八进制中表示三个 x（或 z），在十六进制中表示四个 x（或 z）。例如，6'o1x 与二进制 001xxx 相同，8'h8z 与二进制 1000zzzz 相同。

如果数字中没有给出<位宽>，则位宽为相应数值定义的长度。例如，'o43F 是八进制数，每个八进制数占 3 位，因此其位宽为 9，即'o43F=9'o43F。'hA126 是十六进制数，每个十六进制数占 4 位，因此其位宽为 16。

如果数字中没有给出<位宽>和<进制>，只给出了<数值>，则默认为十进制数，即简单的十进制格式。十进制数默认为 32 位长度，正数可以用二进制原码表示，负数可以用二进制补码表示。例如 78=32'h0000004E，-5=32'hFFFFFFFB。注意，负号要放在<位宽>之前，不能放在<数值>之前，即 6'd-23 不合法。

如果<位宽>定义的长度比<数值>所需的长度大，一般在左边补 0，例如 4'b01=4'b0001。如果此数最高位为 x 或 z，则在高位补 x 或 z，例如 4'bx1=4'bxxx1。如果<位宽>定义的长度比<数值>所需的长度小，则<数值>的高位被截掉，例如 3'b110001=3'b001。

问号"？"有时也可以代替高阻态 z 出现在数字中，表示"不关心"，以提高程序的阅读性。

2．实数

在 Verilog 中，实数有两种书写格式。第一种是采用十进制数格式，但必须带有小数点，且小数点两边都要有数字，例如"7."就是不正确的；第二种是科学计数法格式，用 e 或 E 表示指数。以下都是正确的实数表示。

```
7.0，1.54，15_000_067.584，0.1314，237.2e2（=23720.0），8E-3（= -0.008）
```

Verilog 定义了实数隐含地转换成整数的方式，即采用四舍五入的方式转换成最接近的整数。例如：

```
16.478，16.34  //转换为整数 16
-9.57，-9.8    //转换成整数-10
-31.02         //转换成整数-31
```

3．字符串

字符串是用双引号括起来的字符序列，例如“INTERNAL ERROR”、“ALL IS ONE”等，不能分成多行书写。在 Verilog 中，字符串主要在仿真时显示一些提示信息。

字符串中每个字符是用其 8 位 ASCII 码来保存，因此保存一个字符串所需要的位宽为字符个数乘以 8。例如，要保存字符串“INTERNAL ERROR”，需要的位宽是 14*8 位。

5.3 Verilog HDL 的数据类型

Verilog 语言是描述数字逻辑电路的语言，因此它必须要具有与硬件电路中物理连线、寄存器、存储器、传输单元等相对应的模型，比如常量和变量。常量的数值一旦定义了就不能更改，变量的数值在程序中可以更改。不同的变量具有不同的特性，可以用数据类型来区分和表达。

Verilog 语言定义了两大类数据类型：线网类型（net type）和寄存器类型（register type），这两种类型各自又分为多种子类型。

5.3.1 线网类型

线网类型的变量通常表示硬件结构之间的物理连线，它的值是由驱动源的值决定的。如果线网类型的变量没有驱动源，则其默认值为 z。线网类型通常用在由 assign 引导的连续赋值语句中。

根据功能不同，线网类型又可以分为多种子类型，如表 5-1 所示，其中常用的 wire、tri、supply0 和 supply1 类型可以综合，其他类型主要在建模和仿真时使用。

表 5-1 线网子类型一览表

序号	名称	说明	序号	名称	说明
1	wire	连线	7	trireg	三态寄存器
2	tri	三态线网	8	tri1	上拉类型
3	wor	线或驱动	9	tri0	下拉类型
4	trior	三态线或驱动	10	supply0	对“地”建模类型
5	wand	线与驱动	11	supply1	对电源建模类型
6	triand	三态线与驱动			

1．wire 类型

wire 是最常用的线网类型，即连线类型，其定义的一般语法格式为：

wire 变量名 1，变量名 2，……；

wire [msb: lsb] 变量名 1，变量名 2，……；

wire 是关键字，后面跟变量名。如果要说明多个相同类型的变量，可以把变量名都列出来，变量名之间用逗号隔开，最后以分号结尾。上面第一个语句说明的 wire 类型只有一位，第二个语句说明的是多位 wire 类型，其位宽由[msb: lsb]来说明，msb 表示最高有效位，lsb 表示最低有效位。位宽[msb: lsb]可以从小数到大数，如[1:8]，也可以从大数到小数，如[3:0]。

例 5-2 是一个包含了 wire 类型定义的例子，描述的逻辑电路如图 5-4 所示。变量 D、E 被定义为 wire 类型，并在 assign 语句中被赋值。在 Verilog 中，输入、输出端口默认为 wire 类型，因此，输出端口 F 可以直接在 assign 语句中赋值。

【例 5-2】 wire 类型定义

```
module mylogic(A, B, C, F);
  input A, B, C;
  output F;
  wire D, E;
  assign D = A & B; //A 和 B 进行逻辑与操作，&是按位逻辑与操作符
  assign E = ~ C;   //对 C 进行逻辑取反操作，~是按位逻辑取反操作符
  assign F = D | E; //D 和 E 进行逻辑或操作，|是按位逻辑或操作符
endmodule
```

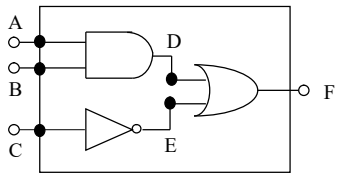


图 5-4 简单逻辑电路

wire 类型的变量可以作为任何类型的表达式或赋值语句（包括连续赋值语句和过程赋值语句）的输入信号，也可以作为连续赋值语句或模块例化语句中的输出信号。在例 5-2 中，D、E 在前两个 assign 语句中是被赋值的输出信号，最后一个 assign 语句中，D、E 出现在右边的表达式中，作为赋值语句的输入信号。

2．tri 类型

tri 是三态线网类型，与 wire 类型的特性、语法一致。 tri 类型可以用来描述多个驱动源驱动同一根线的连接情况。如果有多个驱动源连接到同一个 tri 类型的变量上，则该变量值由表 5-2 确定。

表 5-2 多驱动源时的 wire/tri 取值

wire/tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

例如：

```
tri A, B, F;
……
assign F = A;
assign F = B;
```

F 有 A 和 B 两个驱动源，若 A 为 0，B 为 1 时，则 F 为 x。

3．wor 和 trior 类型

wor 和 trior 是线或类型和三态线或类型，它们的功能和用法一样。如果驱动源为 1，则线或值也为 1。如果有多个驱动源连接到同一个线或类型的变量上，则该变量值由表 5-3 确定。

表 5-3 多驱动源时的 wor/trior 取值

wor/trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

例如：

```
wor [1:0] A;
trior S;
```

4．wand 和 triand 类型

wand 和 triand 是线与类型和三态线与类型，它们的功能和用法一样。如果驱动源为 0，则线与值也为 0。如果有多个驱动源连接到同一个线与类型的变量上，则该变量值由表 5-4 确定。

表 5-4 多驱动源时的 wand/triand 取值

wand/triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

例如：

```
wand [1:0] DBUS;
triand clk, reset;
```

5．trireg 类型

trireg 是三态寄存器类型，用于节点电容的建模，默认值为 x。当所有驱动源都为高阻态 z 时，三态寄存器类型的变量保存作用在其上的最后一个值。例如：

```
trireg [1:8] PBUS;
```

6．tri0 和 tri1 类型

tri0 和 tri1 类型可用于线逻辑的建模。在无驱动源的情况下，tri0 线网变量为 0，相当于下拉到逻辑 0，tri1 线网变量为 1，相当于上拉到逻辑 1。如果有多个驱动源连接到同一个 tri0 或 tri1 类型的变量上，则该变量值由表 5-5 确定。

表 5-5 多驱动源时的 tri0/tri1 取值

tri0(tri1)	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0(1)

7．supply0 和 supply1 类型

supply0 类型用来对“地”建模，即逻辑 0，supply1 类型用来对电源建模，即逻辑 1。

例如：

```
supply0 AGND, DGND;
supply1 [2:0] VCC;
```

5.3.2 寄存器类型

寄存器类型的变量不仅可以描述组合逻辑电路，还可以描述具有存储功能的单元。这类变量在下一次赋值之前保持前一个值不变，起到寄存器的作用。寄存器类型又可以分为五种子类型，分别为 reg、integer、time、real 和 realtime。其中 reg 和 integer 可以进行综合，time、real 和 realtime 主要用于仿真和建模。

5.3.2.1 reg 类型

reg 是最常用的寄存器类型，其定义的一般语法格式为：

reg 变量名 1，变量名 2，……；

reg [msb: lsb] 变量名 1，变量名 2，……；

reg 是关键字，后面跟变量名。如果要说明多个相同类型的变量，可以把变量名都列出来，变量名之间用逗号隔开，最后以分号结尾。上面第一个语句说明的 reg 类型只有一位，第二个语句说明的是多比特 reg 类型，其位宽由[msb: lsb]来说明，msb 表示最高有效位，lsb 表示最低有效位。例如：

```
reg AT,BT;
reg [3:0] ABUS, PBUS;
reg [1:8] Datain;
```

多位 reg 变量相当于多个一位 reg 变量的组合，如果不做特殊说明，变量中的每一位都可以单独使用，或取其中一段使用。例如上例中定义的 ABUS 就等效于 ABUS[3]、ABUS[2]、ABUS[1]和 ABUS[0]四个一位的 reg 变量，可以单独赋值或取一段赋值，如：

```
ABUS[3] = 1'b1;
ABUS[2:0] = 3'b000;
```

reg 类型的变量只能用于 always 和 initial 引导的过程赋值语句中，不能用在 assign 引导的连续赋值语句中，而 wire 类型的变量不能在过程赋值语句中被赋值。因为输出端口默认是 wire 类型，不能在过程赋值语句中被赋值，所以要在过程赋值语句中对输出端口赋值的话，必须要将其重新定义为 reg 类型。例 5-3 对例 5-1 进行了修改，去掉了内部 reg 变量 COUTA，将输出端口 COUT 重新定义为 reg 类型，可以直接用在过程赋值语句 always 中。注意，输入端口不能被定义为寄存器类型。

【例 5-3】 输出端口重新定义为 reg 类型

```
module CounterHEX(CLK, RST, COUT);
  input CLK, RST;
  output [7:0] COUT;

  reg [7:0] COUT;   //定义 COUT 为 reg 类型

  always @(posedge CLK or negedge RST)
  begin
      if (!RST) COUT <= 0;
      else COUT <= COUT + 1;
```

```
  end
endmodule
```

5.3.2.2 integer 类型

integer 类型包含整数值，多用于循环变量，或者进行高层次的建模，其定义的一般语法格式为：

integer 变量名 1，变量名 2，……，变量名 *N* [msb: lsb]；

integer 是关键字，后面跟变量名。如果要说明多个相同类型的变量，可以把变量名都列出来，变量名之间用逗号隔开，最后以分号结尾。[msb: lsb]表示的是变量的范围界限，不是位宽。integer 类型的定义没有位宽说明，位宽默认为 32 位。例如下面的语句定义了五个 integer 类型变量 J、K[3]、K[2]、K[1]、K[0]，每个变量的位宽都是 32 位。

```
integer J, K[3:0]; //定义了 J,K[3],K[2],K[1],K[0]五个 integer 类型的变量
```

给 integer 变量赋的值是有符号的整数值，负数是 32 位二进制补码，整数值的书写格式可以是简单的十进制格式或基数格式。例 5-4 是一个 integer 变量使用的实例，其中 i 是 integer 变量，作为循环语句的循环变量，使用方法与 C 语言相似。

【例 5-4】 integer 变量的使用

```
module MyAnd(A, B, C);
  input [7:0] A, B;
  output [7:0] C;

  reg [7:0] C;
  integer i;

  always @(A or B)
  begin
      for (i=0; i<8; i=i+1)    //i 作为循环变量
          C[i] = A[i] & B[i];
  end
endmodule
```

integer 类型变量是一个整体，不能对其中的某一位或某几位进行操作。例如，对于上面定义的 J，J[8]和 J[3:0]都是不合法的。一种截取整数位值的方法是先把整数赋给 reg 类型的变量，再截取位值。例如：

```
reg [31:0] JREG;
integer J;
……
JREG = J; //类型转换自动完成，此时 JREG[8]和 JREG[3:0]是合法的
……
```

也可以把 reg 变量的值赋给 integer 变量，类型转换自动完成。如果 reg 变量少于 32 位的话，高位补 0。例如：

```
reg [3:0] JREG;
integer J;
……
JREG = 4'b0111;
J = JREG; //类型转换自动完成，此时 J=32'b00000……0111
……
```

5.3.2.3 time 类型

time 类型的变量用于仿真时间的存储与管理，其定义的一般语法格式为：

time 变量名 1，变量名 2，……，变量名 *N* [msb: lsb]；

time 是关键字，后面跟变量名。如果要说明多个相同类型的变量，可以把变量名都列出来，变量名之间用逗号隔开，最后以分号结尾。[msb: lsb]表示的是变量的范围界限，不是位宽。time 类型的变量位宽一般为 64 位。下面的第一个语句定义了一个时间变量 current_time，第二个语句定义了一个时间变量数组，从 event_time[0]~event_time[31]。

```
time current_time,
time event_time[0:31]; //定义了 32 个 time 类型的变量
```

5.3.2.4 real 和 realtime 类型

real 和 realtime 类型分别是实数寄存器类型和实数时间寄存器类型，两者使用完全一样，其定义的一般语法格式为：

real 变量名 1，变量名 2，……，变量名 *N*；

realtime 变量名 1，变量名 2，……，变量名 *N*；

real 和 realtime 类型定义中没有范围界限，是双精度浮点数，默认值为 0，位宽为 64 位。它们主要用于辅助设计和仿真，不可综合，也不对应任何硬件电路。例如：

```
real as;
as = 1.34;
```

5.3.3 存储器类型

存储器类型不是一种独立的数据类型，而是由 reg 类型扩展得到的类型，也可以把存储器类型看成一个寄存器数组。存储器类型经常用于存储器的建模和描述，其定义的一般语法格式为：

reg [N-1: 0] 存储器变量名 [M-1: 0]；

定义存储器类型的关键字还是 reg，后面是位宽表达式[N-1: 0]，然后是存储器变量名和存储器深度（容量）表达式[M-1: 0]，即定义了一个包含 M 个寄存器的寄存器数组，每个寄存器的位宽是 N 位。例如：

```
reg [7:0] MEM [0:127]; //定义了一个位宽为 8、深度为 128 的存储器
reg  XROM [1:8]; //定义了一个位宽为 1、深度为 8 的存储器
```

上面第一句可以理解为定义了一个位宽为 8、深度为 128 的存储器 MEM，也可以理解为定义了一个寄存器数组，该数组中有 128 个寄存器，用 MEM[0]~MEM[127]来表示，每个寄存器的位宽是 8。第二句没有写出位宽，此时默认位宽为 1，即定义了一个寄存器数组 XROM，包含 8 个寄存器，每个寄存器只有 1 位。

在程序设计中，要注意寄存器和存储器定义的不同，下面两句定义语句中，第一句是寄存器定义，第二句是存储器定义。

```
reg [1:8] XREG;
reg  XROM [1:8];
```

对寄存器可以进行整体赋值，对存储器不能进行整体赋值。例如对以上定义的两个变量，“XREG = 8'b10010011；”是合法的，但“XROM = 8'b10010011；”是不合法的。

对存储器赋值可以采用对其中每个字赋值的方式，例如：

```
reg [7:0] mymem [0:3];
……
```

```
mymem[0] = 8'b00000001;
mymem[1] = 8'b00000010;
mymem[2] = 8'b00000100;
mymem[3] = 8'b00001000;
```

也可以采用系统任务给存储器进行赋值。系统任务和系统函数在后续给予介绍，这里先给出一个例子。

```
reg [7:0] mymem [0:63];
……
$readmemb ("ram.dat", mymem);
```

$readmemb 是 Verilog 中的系统任务，功能是加载二进制值，它可以从文本文件中读取二进制值，加载到存储器中。在此例中，mymem 是存储器，ram.dat 是包含二进制值的文件。

5.3.4 参数类型

为了方便程序的阅读和修改，Verilog 中定义了参数说明，即采用一个标识符来代表一个具体数值或字符串。参数类似于常量，采用关键词 parameter 进行说明，其定义的一般语法格式为：

parameter 标识符 1 = 表达式 1，标识符 2 = 表达式 2，……，标识符 n = 表达式 n ；

以上定义中，表达式可以是具体数值、字符串，也可以是以前定义过的标识符。参数可以用来表示存储器位宽、深度、总线宽度、延迟时间等。参数说明一般在程序的开始处，只能被赋值一次。定义了参数以后，后续程序就可以使用定义过的参数。例如：

```
parameter BASE = 2'b10, PI = 3.14;
parameter NOTE = "Compile is failure!";
……
parameter width = 8;
input [width-1:0] A, B;
```

另外，还有一个与 parameter 功能相似的参数定义关键词 localparam，它的用法与 parameter 相同，但定义的是局部参数，不能被外部程序使用。

5.4 Verilog HDL 的操作符

表达式是 Verilog 语句的重要组成部分，一般由操作数和操作符组成的。操作数可以是常量、参数、线网类型变量、寄存器类型变量，也可以是数组类型变量中的位选择、位段选择、存储器单元以及函数调用等。这些操作数在前面已经进行了说明，本小节着重对 Verilog 语言中的操作符进行介绍。

按照携带的操作数的个数划分，Verilog 中的操作符可以分为单元操作符、二元操作符和三元操作符。单元操作符带一个操作数，且操作数放在操作符的右边，如“~ CLK”；二元操作符带两个操作数，操作数放在操作符的两边，如“A && B”；三元操作符带三个操作数，三个操作数被操作符分割开，如“rst ? a : b”。

按照功能来划分，操作符又可以分为按位逻辑操作符、逻辑操作符、缩位操作符、算术操作符、关系操作符、移位操作符、条件操作符和拼接操作符等。

5.4.1 按位逻辑操作符

按位逻辑操作符是 Verilog 中针对二进制位的基本逻辑操作符，包含的操作符如表 5-6 所示。其中，除了逻辑取反操作符“~”为单元操作符以外，其他操作符都是二元操作符。

表 5-6 按位逻辑操作符

操作符	名称	操作数	实例
~	逻辑取反	1	若 A=4'b1001，则~A=4'b0110
&	逻辑与	2	若 A=4'b1001,B=4'b0111，则 A&B=4'b0001
|	逻辑或	2	若 A=4'b1001,B=4'b0111，则 A|B=4'b1111
^	逻辑异或	2	若 A=4'b1001,B=4'b0111，则 A^B=4'b1110
~^或^~	逻辑异或非（同或）	2	若 A=4'b1001,B=4'b0111，则 A~^B=4'b0001

按位逻辑操作符针对操作数的每一位进行操作，图 5-5 给出了按位逻辑操作符对四种逻辑取值的操作结果。

如果两个操作数的位宽不等，综合器将按照位宽较大的操作数进行运算，位宽较小的操作数在左边补 0。例如：

```
A = 5'b11001;
B = 4'b0111;
```

则 A^B 的结果是 5 位二进制数 5'b11110。

逻辑与 &	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

逻辑或 |	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

逻辑异或 ^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

逻辑异或非 ~^	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

逻辑非 ~	0	1	x	z
	1	0	x	x

图 5-5 按位逻辑操作结果

5.4.2 逻辑操作符

逻辑操作符有三种，如表 5-7 所示。其中逻辑与“&&”和逻辑或“||”是二元操作符，逻辑非是单元操作符。

表 5-7 逻辑操作符

操作符	名称	操作数	实例
&&	逻辑与	2	若 A=4’b1001,B=4’b0000, 则 A&&B=1’b0
||	逻辑或	2	若 A=4’b1001,B=4’b0000, 则 A||B=1’b1
!	逻辑非	1	若 A=4’b1001, 则!A=1’b0

无论操作数有多少位，逻辑操作符只在逻辑 1 和逻辑 0 上操作，结果只有一位逻辑 1 或逻辑 0。例如当 A=4’b1001，B=4’b0000 时，要计算 A&&B 的话，首先判断 A 和 B 是否为全 0。A 不是全 0，即非 0，因此 A 为逻辑 1（或判断为真），B 为全 0，即为逻辑 0（或判断为假），因此 A&&B=1’b1 & 1’b0，结果为逻辑 0。

以上的操作过程也可以理解为先将操作数的所有位进行或操作，得到的结果再进行逻辑操作符定义的操作，即 A&&B=（1|0|0|1）&&（0|0|0|0）=1’b1 & 1’b0=1’b0。

当操作数中任何一位为 x 或 z 时，则结果为 x。例如：x && a、!x 的结果都为 x。

5.4.3 缩位操作符

缩位操作符有六种，如表 5-8 所示。所有的缩位操作符都是单元操作符，针对一个操作数的所有位，从左到右进行操作，得到一位结果。例如，若 A=4’b1001，则 &A=1&0&0&1=1’b0。

表 5-8 缩位操作符

操作符	名称	操作数	实例
&	缩位与	1	若 A=4’b1001,B=4’b1111, 则&A=1’b0, &B=1’b1，若某位的值为 x 或 z，则结果为 x。
~&	缩位与非	1	若 A=4’b1001,B=4’b1111, 则~&A=1’b1, ~&B=1’b0
|	缩位或	1	若 A=4’b1001,B=4’b0000, 则|A=1’b1, |B=1’b0，若某位的值为 x 或 z，则结果为 x。
~|	缩位或非	1	若 A=4’b1001,B=4’b0000, 则~|A=1’b0, ~|B=1’b1
^	缩位异或	1	若 A=4’b1001,B=4’b0111, 则^A=1’b0, ^B=1’b1，若某位的值为 x 或 z，则结果为 x。
~^或^~	缩位异或非	1	若 A=4’b1001,B=4’b0111, 则~^A=1’b1, ~^B=1’b0

5.4.4 算术操作符

算术操作符有五种，如表 5-9 所示，它们都是二元操作符，其中加法、减法、乘法可以综合出具体电路，除法和取模操作在被除数是 2 的幂次方的情况下才能进行综合。

表 5-9 算术操作符

操作符	名称	操作数	实例
+	加法	2	若 A=4'b1101, B=4'b0101, F 位宽为 8，则： F=A+B=8'b00010010
-	减法	2	F=A-B=8'b00001000
*	乘法	2	F=A*B=8'b01000001
/	除法	2	F=A/B=8'b00000010，取商的整数 2，小数丢弃
%	取模	2	F=A%B=8'b00000011，除法取余数 3，取第一个操作数符号

在进行算术运算时，如果操作数的位宽不同，则表达式的结果取最大的位宽。在赋值语句中，结果的位宽以左侧目标变量的位宽为准。例如：

```
reg [3:0] AT, BT, FT;
reg [4:0] CT;
reg [7:0] S;
……
S = BT * (AT + CT);
FT = AT + BT;
```

在第一个赋值语句中，因为 AT 和 CT 的位宽不一样，因此表达式（AT+CT）的位宽取较大者，即 CT 的 5 位，AT 高位补 0。整个赋值语句左侧目标 S 的位宽为 8 位，因此最后的计算结果为 8 位。第二个赋值语句左侧目标 FT 的位宽是 4，最后结果的位宽也为 4。如果 AT+BT 的结果大于 4 位，则高位被丢弃。

算术运算中的操作数可以是二进制数、整数或者它们的混合类型。在具体计算时，要注意所处理的数据是作为无符号数对待的还是作为有符号数对待的。一般而言，存储在 wire、reg 类型中的数据以无符号数对待，存储在 integer 类型中的数据以有符号数对待。例如：

```
reg [5:0] DATA0;
integer  DATA1;
……
DATA0 = -6d12; //赋值后，DATA0 中存储的是 110100，作为无符号数 52 看待
DATA1 = -6d12; //赋值后，DATA1 中存储的是 110100，作为有符号数-12 的补码看待
```

在上面的例子中，DATA0 和 DATA1 分别定义为 reg 和 integer 类型。当它们被赋值-12 时，存入变量中的值为-12 的二进制补码，即 110100。虽然形式上两者一致，但 DATA0 作为无符号数，其表示的值是 52，DATA1 作为有符号数，其表示的值是-12。DATA0 除以 2 的结果是 26，即 011010，而 DATA1 除以 2 的结果是-6，即 111010。

可以用 signed 对 wire、reg 类型进行说明，显式设定为有符号数。例如：

```
input signed [3:0] A, B; //对输入端口 A、B 设定为有符号数
reg signed data; //对 reg 变量 data 设定为有符号数
```

5.4.5 关系操作符

关系操作符分为不等操作符和相等操作符两类，每一类又各自包含四种操作符，如表 5-10 和表 5-11 所示，它们都是二元操作符。关系操作符的结果为“真”（逻辑 1）或“假”（逻辑 0）。

在不等操作中，如果操作数中有 x 或 z，则不等操作的结果为 x；如果操作数的位宽不一致，则将长度较小的操作数的左边补 0，然后再进行比较。

表 5-10 不等操作符

操作符	名称	操作数	实例
>	大于	2	15 > 60 结果为 0；4'b0110 > 4'b0101 结果为 1
<	小于	2	15 < 60 结果为 1；28 < 6'b1x0101 结果为 x；
>=	大于或等于	2	4'b110 >= 4'b0101 结果为 1
<=	小于或等于	2	4'b0111 <= 4'b0011 结果为 0

表 5-11 相等操作符

操作符	名称	操作数	实例
==	等于	2	4'b0110==4'b110 结果为 1；2'b0x==2'b0x 结果为 x
!=	不等	2	7 != 20 结果为 1
===	全等	2	2'b0x===2'b0x 结果为 1； 4'b0110===4'b110 结果为 0；
!==	不全等	2	4'b0110!==4'b110 结果为 1

在使用等于操作符“= =”进行比较时，操作数按照二进制逐位进行比较。当两个操作数相等时，结果为“真”，否则为“假”。如果两个操作数的位宽不等，则将长度较小的操作数的左边补 0，然后再进行比较。如果两个操作数中包含 x 或 z，则结果为 x。

在使用全等操作符“= = =”进行比较时，将 x 或 z 当成确定值进行比较，当两个操作数的表述完全相同时，结果为“真”。如果两个操作数的位宽不等，则直接判定为“假”。

5.4.6 移位操作符

移位操作符有逻辑左移操作符和逻辑右移操作符，在 Verilog-2001 版本中还增加了有符号数的左移和右移操作，如表 5-12 所示。移位操作符左侧操作数为被移位的数据，右侧操作数为移动的位数。逻辑左移和右移操作在空出的位置补 0，有符号数右移操作在空出的位置补符号位，有符号数左移操作在空出的位置补 0。

表 5-12 移位操作符

操作符	名称	操作数	实例
<<	逻辑左移	2	8'b01101110 << 3 结果为 8'b01110000
>>	逻辑右移	2	8'b01101110 >> 4 结果为 8'b00000110
<<<	有符号数左移	2	8'b10001010 <<< 2 结果为 8'b00101000
>>>	有符号数右移	2	8'b10001010 >>> 2 结果为 8'b11100010

5.4.7 条件操作符

条件操作符的作用是根据条件表达式的值进行选择，其一般语法格式为：

条件表达式 ？ 表达式 1 ：表达式 2

如果条件表达式为“真”（逻辑 1），则选择表达式 1 的值；如果条件表达式为“假”（逻辑 0），则选择表达式 2 的值。例如：

```
Date = （Flag > 30） ? Cur : Nex;
```

先对表达式 Flag > 30 进行判断，如果为“真”，则将 Cur 的值赋给 Data，否则，将 Nex 的值赋给 Data。例 5-5 是采用条件操作符描述的 D 触发器。

【例 5-5】 条件操作符描述的 D 触发器

```
module MYDFF(CLK, RST, D,  Q);
  input CLK, RST, D;
  output  Q;
  reg Q;
  always @(posedge CLK or negedge RST)
      Q = (!RST) ? 1'b0 : D;
endmodule
```

5.4.8 拼接操作符

拼接操作符用花括号{}表示，可以将多个操作数的二进制表达式或者其中的某些位连接成一个新的操作数，其一般语法格式为：

{表达式 1，表达式 2，……，表达式 n}

这里的表达式可以是标量、矢量、矢量的某些位等，例如：

```
reg A, B;
reg [3:0] A1BUS, A2BUS;
reg [7:0] CBUS;
……
CBUS = {A1BUS, A2BUS[3], A2BUS[2], A, B};
```

进行连接的每个操作数的位宽必须是固定的，位宽未知的常数不能进行拼接操作，例如{A1BUS，5}是非法的。拼接操作可以重复和嵌套，以简化某些表达方式，例如：

```
CBUS = {2{A1BUS}}; // 等效于{A1BUS, A1BUS}
CBUS = {4{A, B}};  // 等效于{A, B, A, B, A, B, A, B};
CBUS = {4{2'b10}}; // 等效于 10101010;
```

5.5 Verilog HDL 的语句

Verilog 描述硬件电路的基本单元是模块，在程序中要描述模块的功能、结构以及输入、输出端口等。一个 Verilog 模块的基本结构为：

```
module module_name (port_name);
  Declarations:              //说明部分
  reg, wire, parameter,
  input, output, inout,
  function, task, ……
  Statements:                //语句部分
  initial statement          // initial 语句
  always statement           // always 语句
  module instantiation       // 模块例化语句
  gate instantiation         // 基本门例化语句
  UDP instantiation          // 用户自定义例化语句
  continuous assignment      // 连续赋值语句
endmodule
```

Verilog 模块的基本结构可以分为说明部分和语句部分。说明部分对模块做一些定义，比如端口、变量、参数等，大部分内容已经在前面各小节有所介绍。语句部分描述电路的具体功能，不同的语句可以分别从数据流、行为和结构等方面对电路进行描述。以上所列出的

语句是常用的一些并行语句，这些语句并不一定出现在同一个模块中。写程序时，要根据实际需要，选择其中的某些语句对电路进行描述。

说明部分的一些内容，如变量、参数的定义，可以放在模块内的任何地方，并不一定要放在所有语句之前，但一定要放在使用它的语句之前，这一点跟 C 语言相似。为了使程序的结构清晰，便于阅读，建议将说明部分放在所有语句之前。

以上所列出的语句都是并行语句。并行语句的含义就是这些语句所描述的电路模块是同时工作的，与这些语句在程序中书写的前后顺序无关。在例 5-1 中有一个 always 语句和一个 assign 语句，always 语句在前，assign 语句在后。因为它们是并行语句，因此可以调换它们的顺序，写成如下形式。注意 always 语句是一个复合结构，其中还包括 if 语句。

```
……
assign COUT = COUTA;
always @(posedge CLK or negedge RST)
begin
    if (!RST) COUTA <= 0;
    else COUTA <= COUTA + 1;
end
……
```

Verilog 语句都可以用于仿真，但只有一部分语句可以综合，得到具体的电路结构。因此要编写可综合程序时，必须注意选用的语句和设计方式。

5.5.1 连续赋值语句

Verilog 模块有数据流描述方式、行为描述方式和结构描述方式，其中采用连续赋值语句的描述方式被称为数据流描述方式，它的一般语法格式为：

assign [延时] 目标变量名 = 表达式；

assign 是连续赋值语句的关键字，方括号表示可以省略的部分，目标变量必须是线网型变量，赋值符号采用“=”。右边表达式可以是线网类型、寄存器类型或函数，最后以分号结束。

连续赋值语句的执行过程是：当等号右边的表达式中任何一个变量发生了变化，就会导致表达式重新计算一次，并将结果赋值给左边的目标变量。如果语句中没有延时部分，则表达式的值立即赋给目标变量；如果有延时部分，则经过给定的延时之后，表达式的值赋给目标变量。延时只在仿真时有意义，不能综合。这种赋值操作过程持续保持下去，所以叫做连续赋值语句。

在例 5-2 中，采用了三个 assign 语句，它们是并行执行的，所以书写顺序可以任意调换。由于 wire 和 assign 的特性很相似，可以直接用 wire 替代 assign。例如，例 5-2 中对 F 的赋值语句可以写成：

```
wire F = D | E;
```

例 5-6 给出了用 assign 语句设计的无符号数加法器程序。IN1、IN2 是位宽为 4 的无符号数，它们相加的结果存放在 COUT 中，进位存放在 C 中。

【例 5-6】用连续赋值语句设计的加法器

```
module ADD2(IN1, IN2, C, COUT);
  input [3:0] IN1, IN2;
  output  C;
  output [3:0] COUT;
  assign {C, COUT} = IN1 + IN2;
endmodule
```

5.5.2 过程语句

在 Verilog 程序中，过程语句用于对电路的行为进行描述。对复杂电路进行设计，行为描述方式更加清楚、有效。过程语句主要有 always 语句和 initial 语句两种。一般 initial 语句用于仿真，用作初始化设置，不能进行综合。always 语句可以进行综合，它是 Verilog 程序中使用最多的过程语句。

5.5.2.1 always 语句

always 语句其实是一个语句结构，其内部可以包含其他的语句或语句块，它的一般语法格式为：

always　@ (敏感信号列表)

　　begin　语句或语句块；end

always 语句结构以 always　@做引导。在@符号之后，是一个敏感信号列表。只要敏感信号列表中任意一个信号有变化，就会触发 always 语句执行一遍，即 always 语句对这些信号“敏感”。一般要求把 always 语句所有的输入信号都作为敏感信号。在例 5-4 中，将输入 A、B 作为敏感信号。

敏感信号列表的写法有以下几种形式，其中前三种是电平触发，即敏感信号的电平发生了变化，比如从 0 变成 1，或者从 1 变成 0，就触发 always 语句执行一遍。后两种是边沿触发，即当敏感信号的上升沿出现（用关键字 posedge）或者下降沿出现（用关键字 negedge），就触发 always 语句执行一遍。always 语句执行完一遍以后，又处于等待状态，等待下一次触发，所以 always 语句是可以循环执行的。

```
always @(A or B or C)  //假设有 A、B、C 三个敏感信号，可以用 or 把它们隔开；
always @(A，B，C)      //敏感信号之间可以用逗号隔开；
always @(*)       //在综合器默认所有敏感信号都列入的情况下，可以用*代替敏感信号列表；
always @(posedge CLK)   //如果 CLK 有上升沿出现，则触发 always 语句；
always @(posedge CLK or negedge RST)   //如果 CLK 有上升沿出现，或者 RST 为低，则触发 always 语句；
```

always 结构中包含的语句具有顺序执行的特点，也叫做顺序语句，如 if 语句、case 语句等，将在 5.5.2.3 小节进行说明。一个 always 语句中可以包含一条顺序语句，也可以包含多条顺序语句，这些顺序语句用 begin……end 包括起来，表示一个顺序语句块。例 5-7 描述的是一个 4 选 1 的多路选择器，always 语句中包括了一个 case 语句。

【例 5-7】4 选 1 选择器

```
module MUX4to1(A, B, C, D, S, Y);
  input A, B, C, D;
  input [1:0] S;
  output  Y;
  reg Y;
  always @(A or B or C or D or S)
      begin
        case (S)
          2'b00: Y <= A;
          2'b01: Y <= B;
          2'b10: Y <= C;
          2'b11: Y <= D;
```

```
            Default: Y <= A;
          endcase
        end
    endmodule
```

在以上例子中，always 语句中只包含一个 case 语句，所以 begin……end 可以省略。可以看出，case 语句也是一个复合结构，将在 5.5.2.3 小节介绍。

例 5-8 是一个计数器的描述程序，采用了两个 always 语句。第一个 always 语句描述的是一个计数器的操作，当时钟上升沿到来时，计数器加一。第二个 always 语句描述输出使能信号 OE 的作用。当 OE 为 1 时，使能有效，允许计数器的值输出，否则，计数器的值保持为高阻态。

【例 5-8】十六进制计数器 1

```
module COUNTER16_1(CLK, RST, OE, COUT);
  input CLK, RST, OE;
  output  [3:0] COUT;
  reg [3:0] COUT;
  reg [3:0] CT;

  always @(posedge CLK or negedge RST)
  begin
      if (!RST)
          CT <= 4'b0000;
      else
          CT <= CT + 1;
  end
  always @(OE)
      COUT = OE ? CT: 4'bzzzz;
endmodule
```

例 5-8 中的两个 always 语句是并行执行的，它们的顺序可以调换。另外，always 语句与 assign 语句在一些情况下可以相互转换。例如，例 5-8 也可以写成例 5-9 的形式，其中第二个 always 语句用 assign 语句替代了。

【例 5-9】十六进制计数器 2

```
module COUNTER16_2(CLK, RST, OE, COUT);
  input CLK, OE, RST;
  output  [3:0] COUT;
  reg [3:0] CT;

  always @(posedge CLK or negedge RST)
  begin
      if (!RST)
          CT <= 4'b0000;
      else
          CT <= CT + 1;
  end

  assign COUT = OE ? CT: 4'bzzzz;
endmodule
```

在仿真程序中，经常会用到不带敏感信号列表的 always 语句，例如：

```
always #10 CLK = ~CLK;
```

这个语句中有一个时延#10，表示每隔 10 个时间单位 CLK 就翻转一次，因此可以产生周期为 20 个时间单位的时钟信号。若时间单位为 1ns，则时钟周期为 20ns。

在 always 语句中，赋值符号有“<=”和“=”两种形式。“<=”叫做非阻塞式赋值，“=”叫做阻塞式赋值。这两种赋值方式的区别将在 5.5.2.3 小节介绍。

5.5.2.2 initial 语句

initial 语句也是一个过程语句，它本身也是并行语句，与其他 initial 语句、always 语句和 assign 语句并行执行。initial 语句没有敏感信号列表，一般用在仿真程序中，不可综合。initial 语句的一般语法格式为：

initial

begin　语句或语句块；end

initial 语句中的所有内部语句用 begin……end 组成一个块。如果只有一条语句，也可以不要 begin……end。从仿真的 0 时刻开始，initial 内部各条语句顺序执行，而且只执行一遍。例如：

```
reg X,Y;
……
initial  X = 4; //无时延的过程赋值语句，X 在 0 时刻被赋值为 4
initial  #3 Y = 8; //有时延的过程赋值语句，从 0 时刻开始经过 3 个时间单位之后，Y 被赋值 8
```

例 5-10 是一个 initial 语句的应用实例。该程序片段用在 Verilog 模块的仿真中，给输入信号添加激励，产生输入信号流。

【例 5-10】initial 语句的应用实例

```
`timescale 1ns/100ps //仿真时间标度语句
module test;
  reg D1, D2, D3;
  initial
      begin
          D1 = 0; D2 = 0; D3 = 0; //从 0 时刻执行，三个变量都赋值 0
          # 10  D3 = 1; //过了 10ns，D3 赋值为 1，其他两个不变
          # 10  D2 = 1; D3 = 0; //又过了 10ns，D2 赋值为 1，D3 赋值为 0，D1 不变
          # 10  D3 = 1; //又过了 10ns，D3 赋值为 1，其他两个不变
          # 10  D1 = 1; D2 = 0; D3 = 0;
          # 10  D3 = 1;
          # 10  D2 = 1; D3 = 0;
          # 10  D3 = 1;
          # 10  $finish //$finish 是系统函数，退出仿真
      end
endmodule
```

在 Verilog 中，以撇号（反引号）引导的语句是预编译指令，可以在综合之前对程序某些内容预先处理。如例 5-10 中的`timescale 就是预编译指令，说明仿真的时间单位和时间精度，其一般的语法格式为：

`timescale　仿真时间单位/仿真时间精度

仿真时间单位和仿真时间精度的数值只能取 1、10、100 三种，时间单位可以取秒（s）、毫秒（ms）、微秒（us）、纳秒（ns）和皮秒（ps）。仿真时间单位不能小于仿真时间精度，比如 1ns/100ps、10ns/1ns、1ns/1ns 都是正确的表述。如果程序中书写的延时数值的精度高

于仿真时间精度，则按照四舍五入进行取舍。例如针对例 5-10，如果有：

```
# 10.22  D1 = 1; D2 = 0; D3 = 0;
```

则近似成 10.2 个时间单位，即 100ps 以下的延时就无法表示了。

`timescale 预编译命令放在 module 之外，影响后面所有的仿真延迟时间。例 5-10 产生的激励信号如图 5-6 所示。

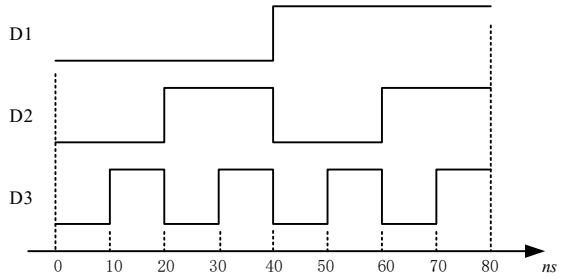


图 5-6 例 5-10 产生的激励信号

5.5.2.3 其他行为描述语句

以上介绍了两种过程语句：always 语句和 initial 语句，它们都是复合结构，内部还包含其他行为描述语句，如过程赋值语句、if 条件语句、case 条件语句、循环语句等，以下进行介绍。

1．过程赋值语句

过程赋值语句只能用在 always 语句或 initial 语句中，赋值符号有“<=”和“=”两种形式，“<=”叫做非阻塞式赋值，“=”叫做阻塞式赋值。赋值符号左边的变量必须是寄存器类型。例如：

```
reg A, B;
reg Temp1, Temp2, Temp3;
……
Temp1 = ~ A;
Temp2 = #2 A & B;
#5 Temp3 = A ^ B;
……
```

第一个过程赋值语句没有延时，右边的表达式~A 计算出来以后，立即赋值给 Temp1。第二个赋值语句的右边有延时#2，是语句内部的延时，即右边的 A&B 计算出来以后，并不是立即赋给 Temp2，而是等待 2 个时间单位，再赋值给 Temp2。第三个语句的延时是放在整个赋值语句的前面，表示语句之间的延时，即前一条语句执行完以后，要延时 5 个单位时间，然后再计算右边表达式的值并赋给 Temp3。

（1）阻塞式赋值语句

采用符号“=”的赋值语句叫做阻塞式赋值语句，其含义就是在这条语句执行的同时，在同一个过程语句中的其他的语句被“阻塞”，即其他语句不能执行，而且该语句的赋值结果是在该语句执行完就立即得到的。例 5-11 是阻塞式赋值语句的实例。

【例 5-11】阻塞式赋值语句实例

```
module EXAMPLE_BLOCKING(A, B, Cin, Cout, Dout);
  input A, B, Cin;
  output  Cout, Dout;
  reg Cout, Dout;
```

```
    reg T1, T2, T3;
    always @(A, B, Cin)
    begin
        T1 = A & B;
        T2 = B & Cin;
        T3 = A & Cin;
        Cout = (T1 | T2) | T3;
        Dout = (A ^ B) ^ Cin;
    end
  endmodule
```

在例 5-11 的 always 语句中有五个阻塞式赋值语句。当过程被触发时，它们按顺序执行。当第一句“T1 =A & B;”执行时，后面四句被阻塞，不能执行。当第一句执行完了，T1 获得了 A & B 的值以后，才能执行第二句。第二句执行时，同样也阻塞其他语句。第二句执行完了，T2 获得了 B & Cin 的值以后，才能执行第三句。后面的几句依此类推。

在 initial 语句中，阻塞式赋值语句的执行方式也是一样的。例如：

```
  initial
    begin
        Din = #5 0;
        Din = #3 1;
        Din = #10 0;
    end
    ……
```

第一条语句执行时，先延时 5 个时间单位，然后 Din 被赋值为 0，其他两条语句不能执行。第二句执行时，先延时 3 个时间单位，再将 Din 赋值 1。最后执行第三条语句，先延时 10 个时间单位，再将 Din 赋值 0。

（2）非阻塞式赋值语句

采用符号“<=”的赋值语句叫做非阻塞式赋值语句，其含义就是在该语句执行的同时，并不限制其他的语句执行，而且该语句的赋值结果要在整个过程结束时才能获得。例 5-12 是非阻塞式赋值语句的实例。

【例 5-12】非阻塞式赋值语句实例

```
  module EXAMPLE_NONBLOCKING(A, B, C, D);
    input A, B;
    output  C, D;
    reg C, D;
    always @(A, B)
    begin
        C <= A & B;
        D <= A ^ B;
    end
  endmodule
```

在例 5-12 的 always 语句中有两个非阻塞式赋值语句。当过程被触发时，可以把这两个语句的执行看成是顺序“启动”。首先是第一句“C<=A & B;”启动，但它没有阻塞其他语句，所以第二句“D<=A ^ B;”随后也“启动”了。这两个语句虽然同时执行，但直到 always 过程结束时，C 才能获得 A & B 的值，D 才能获得 A^B 的值。

（3）两种过程赋值语句的区别

阻塞式赋值语句和非阻塞式赋值语句在某些情况下是可以互换的，比如例 5-12 中的两个非阻塞式赋值语句就可以换成阻塞式赋值语句，结果不变。但一般情况下，两种过程赋值语句会产生不同的结果。以下用一个例子来说明。

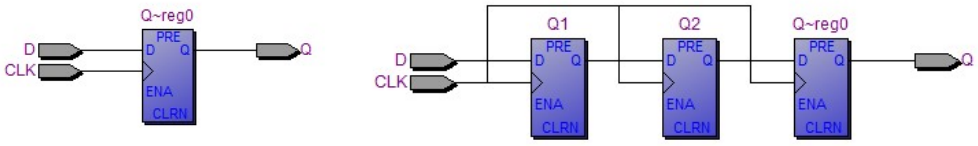
例 5-13 和例 5-14 除了 always 语句中分别采用阻塞式赋值语句和非阻塞式赋值语句之外，其他部分都相同，但是综合出来的硬件电路有很大的差别，如图 5-7 所示。

【例 5-13】阻塞式赋值语句描述 D 触发器

```
module EXAMPLE_B(CLK, D, Q);
  input CLK, D;
  output  Q;
  reg Q;
  reg Q1,Q2;
  always @(posedge CLK)
  begin
      Q1 = D;
      Q2 = Q1;
      Q = Q2;
  end
endmodule
```

【例 5-14】非阻塞式赋值语句描述 D 触发器

```
module EXAMPLE_NB(CLK, D, Q);
  input CLK, D;
  output  Q;
  reg Q;
  reg Q1,Q2;
  always @(posedge CLK)
  begin
      Q1 <= D;
      Q2 <= Q1;
      Q <= Q2;
  end
endmodule
```



（a）阻塞式赋值语句　　（b）非阻塞式赋值语句

图 5-7 两种过程赋值语句综合结果对比

在例 5-13 中，当时钟上升沿到来时，always 过程语句被触发，三条阻塞式赋值语句按顺序执行，并且互相阻塞。第一句执行完了，Q1 立即就获得了 D 的值；第二句执行完了，Q2 就得到了 Q1 的值。因为执行第二句时，Q1 已经得到 D 的值，所以第二句执行完了，Q2 的值也等于 D。同样，第三句执行完了，Q 得到 Q2 的值，即 D 的值。因此，例 5-13 中的三条阻塞式赋值语句可以等效为语句“Q = D;”，综合出来的结果就是一个标准的 D 触发器，如图 5-7（a）所示。

在例 5-14 中，当时钟上升沿到来时，always 过程语句被触发，三条非阻塞式赋值语句按顺序“启动”，不互相制约，同时执行。在第一句还未执行完时，第二句就启动了，此时 Q1 还没有获得 D 的新值，所以第二句在执行的时候，是把 Q1 当前的值传送给 Q2。同样，第三句执行时，Q2 还没有得到 Q1 的新值，所以第三句在执行时，是把 Q2 当前的值传递给 Q。在 always 过程语句结束时，Q1、Q2、Q 的值都得到更新。因此，三条非阻塞式赋值语句具有数据存储、按时钟传递的特征，综合出来的结果就是三个 D 触发器的传输链，如图 5-7（b）所示。

注意，在一个过程语句中，对一个变量的赋值可以采用阻塞式赋值语句或者非阻塞式赋值语句，但不能混合使用。

2. if 条件语句

if 条件语句是最常见的条件语句之一，可以根据条件满足与否，执行不同的操作。if 条件语句只能放在过程语句中，它的一般语法格式有三种：

```
（1）if （条件表达式）
        begin
          语句块；
        end
（2）if （条件表达式）
        begin
          语句块 1；
        end
     else
        begin
          语句块 2；
        end
（3）if （条件表达式 1）
        begin
          语句块 1；
        end
     else if （条件表达式 2）
        begin
          语句块 2；
        end
     …….
     else if （条件表达式 n）
        begin
          语句块 n；
        end
     else
        begin
          语句块 n+1；
        end
```

if 条件语句中的“条件表达式”必须放在括号内，即便是一个标识符，也要放在括号内，

如 if (A)。条件表达式可以是关系表达式，如 if (A= =B)，或者是逻辑表达式，如 if (A & B)。if (A & B)也可以看成是 if (A & B = =1)的简写形式。

如果条件表达式中的关系成立，则判断为“真”，关系不成立，则判断为“假”。对于逻辑表达式，如果运算结果为 1，则判断为“真”，如果运算结果为 0、x、z，则判断为“假”。

格式一的 if 语句的执行过程是：如果条件表达式被判断为“真”，就顺序执行后面的语句块，否则跳过此 if 语句。例 5-15 给出了用 if 语句设计锁存器程序，图 5-8 为其功能仿真波形。从图 5-8 可见，当 EN 为高电平时，Q 值随 D 值改变而改变。当 EN 为低电平时，Q 值保持不变。

【例 5-15】用 if 语句设计锁存器

```
module LATCH8(EN, D, Q);
  input EN;
  input [7:0] D;
  output  [7:0] Q;
  reg [7:0] Q;
    always @(EN or D)
    begin
      if (EN)
          Q <= D;
    end
endmodule
```

在例 5-15 中，当使能信号 EN 为 1 时，执行锁存操作，将输入 D 赋值给 Q。如果 EN 不为 1，则没有给出处理说明，所以格式一是不完整的 if 语句。当条件表达式不成立时，格式一的默认处理方式是保持输出原来的值不变。这个特性与锁存器非常吻合，适于设计带有锁存器的电路。如果在不需要锁存器的情况下使用，可能会导致综合后产生不必要的锁存器。

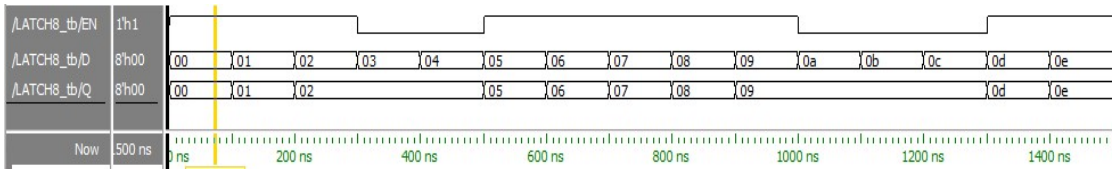


图 5-8　例 5-15 的功能仿真波形

格式二的 if 语句的执行过程是：如果条件表达式被判断为“真”，就顺序执行语句块 1，否则，就顺序执行语句块 2。格式二是完整的 if 语句，对条件表达式的真、假情况都给出了处理语句，可以用于设计具有二分支结构的组合逻辑电路。例 5-16 给出了用格式二的 if 语句设计选择器的程序，图 5-9 为其时序仿真波形。

【例 5-16】用 if 语句设计二选一选择器

```
module SEL2(A, B, S, C);
  input A, B, S;
  output  C;
  reg C;
    always @(A, B, S)
    begin
      if (S)
          C <= A;
      else
          C <= B;
    end
endmodule
```

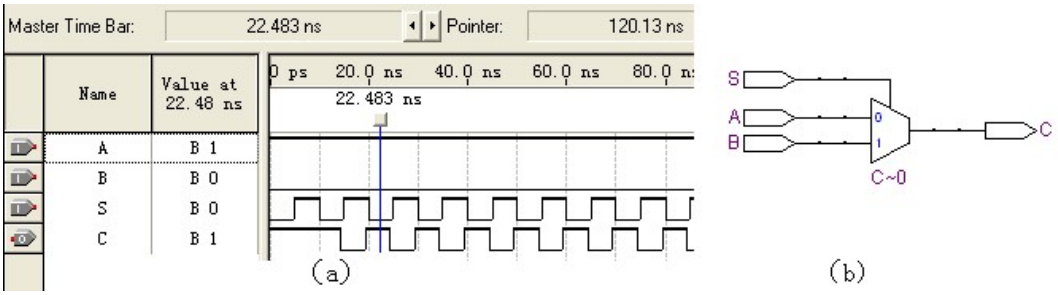


图 5-9　例 5-16 的时序仿真波形和 RTL 原理图

格式三的 if 语句是多重条件判断语句，它的执行过程是：如果给定的条件表达式 1 为“真”，就顺序执行语句块 1，否则就判断条件表达式 2；如果条件 2 为“真”，就顺序执行语句块 2，否则就判断条件表达式 3。依此类推，直到判断条件表达式 n。如果条件 n 为“真”，就顺序执行语句块 n，否则就执行 else 之后的语句块 n+1。在这种形式的 if 语句中，一次执行过程中只能有一个语句块被执行，所以这种 if 语句具有优先级，也就是说，即使条件表达式 1、条件表达式 2 到条件 n 都为“真”，但由于有优先级的存在，只会执行语句块 1。所以条件表达式次序的变化会造成逻辑功能的变化。例 5-17 给出了用 if 语句实现的四选一选择器。

【例 5-17】if 语句实现四选一选择器

```
module SEL4(A, B, C, D, S, Y);
  input A, B, C, D;
  input [1:0] S;
  output  Y;
  reg Y;
    always @(A, B, C, D, S) begin
      if (S==2'b00)
          Y <= A;
      else if (S==2'b01)
          Y <= B;
      else if (S==2'b10)
          Y <= C;
      else
          Y <= D;
    end
endmodule
```

if 语句可以进行多重嵌套，例如 if-if-else 语句，这时要注意 else 到底归属于哪一个 if。在 Verilog 语言里规定 else 归属于最靠近的没有 else 的 if。例如：

```
if (CLK)
  if (RST)
      DOUT <= 4'b0000;;
  else
      DOUT <= DOUT + 1;
```

其中，else 归属于 if(RST)。为了使程序清楚易读，可以在 if 嵌套语句中加上 begin……end。例如：

```
if (S1)
  begin
  if (S0)
```

```
        Y <= 2'b11;
    else
        Y <= 2'b10;
    end
  else
    begin
    if (S0)
        Y <= 2'b01;
    else
        Y <= 2'b00;
    end
```

3．case 语句

case 语句是另一种常用的条件语句，只能出现在过程语句中，也可以产生多种执行分支。case 语句的一般语法格式为：

case （条件表达式）

取值 1：begin 语句块 1；end

取值 2：begin 语句块 2；end

……

取值 n：begin 语句块 n；end

[default：begin 语句块 n+1；end]

endcase；

case 语句的执行过程是：首先对条件表达式进行计算，得到的值与后面的各项分支取值进行对比，有相等的值就执行后面相应的语句块。例如，条件表达式的值等于取值 1，则顺序执行语句块 1，执行完就跳出 case 语句。

如果 case 语句中列出的各项取值能够覆盖条件表达式的取值范围，则可以省略掉 default 分支；如果不能覆盖的话，则必须要加上最后一项 default 分支，以免综合出不必要的锁存器。各项取值并不要求互相排斥，可以列出相同的值，但只有第一个匹配的值对应的语句块会执行，执行完就跳出 case 语句。

case 语句中的各项取值可以是常量，也可以是参数、标识符等。例 5-7 是采用 case 语句设计的四选一选择器，其中各项取值使用的是常量表达式，而例 5-18 中的各项取值采用了参数表达式。

【例 5-18】取值为参数表达式的 case 语句

```
module EXAMPLE_ALU(X, Y, OPcode, R);
  parameter ADD=2'b00, SUB=2'b01, MUL=2'b10, DIV=2'b11;
  input [7:0] X, Y;
  input [1:0] OPcode;
  output  [15:0] R;
  reg [15:0] R;
  always @(X, Y, OPcode)
      case (OPcode)
          ADD: R = X + Y;
          SUB: R = X - Y;
          MUL: R = X * Y;
          DIV: R = X / Y;
      endcase
endmodule
```

case 语句在执行过程中，对条件表达式和各项取值进行逐位比较，每一位的取值可能是0、1、x、z，即把 x 和 z 也当作一个值来对比。如果条件表达式和各项取值的位宽不同，则采用左边补 0 的方式，统一扩展到最长的位宽。如果有多个取值对应同样的操作，可以把它们一起列出来，中间用逗号分开，例如：

```
case (num)
  1'b0: dout = in1;
  1'b1: dout = in2;
  1'bx, 1'bz: dout = 4'bxxxx;
endcase
```

与 case 语句相对应的还有两种语句：casex 语句和 casez 语句。这两种语句只在处理 x 和 z 时与 case 语句不一样，其他操作与 case 语句相同。在 casez 语句中，如果条件表达式或者各项取值表达式中出现 z 值，那么这个值被忽略掉（或者叫做不关注、无关位），不进行比较。casex 语句把 x 和 z 值都忽略掉，不进行比较。例如：

```
casez (mask)
  4'b1zzz: Ans = 4'b1000;
  4'b01zz: Ans = 4'b0100;
  4'b001z: Ans = 4'b0010;
  4'b0001: Ans = 4'b0001;
  default: Ans = 4'b0000;
endcase
```

4. 循环语句

循环语句的作用是重复地执行一个语句块。循环语句在计算机程序设计中很受欢迎，但在硬件描述语言中却不一样，过多的循环会大大增加系统资源的开销，同时增加综合的难度，所以循环语句一般用来做算法的实现，而不用于控制电路的设计。Verilog 中有四种循环语句，分别是 for 语句、while 语句、repeat 语句和 forever 语句，其中 forever 语句只能在仿真时使用，不能进行综合。

（1）for 语句

for 语句的语法格式为：

for （循环变量初始值表达式；循环控制条件表达式； 循环变量修改表达式）
    循环体语句块；

for 语句中的循环变量一般采用 integer 类型，它的执行步骤是：

a） 由“循环变量初始值表达式”确定循环变量的初始值；

b） 计算“循环控制条件表达式”的值，若为“真”，则执行循环体语句块，并转到步骤 c；若为“假”，则结束循环语句；

c） 按照“循环变量修改表达式”更改循环变量的值。可以采取递增或递减的修改方式，然后转到步骤 b，继续执行。

例 5-19 是采用移位相加实现 4×4 乘法器的程序实例，其中使用了 for 语句。循环变量 i 在使用之前进行了定义。

【例 5-19】for 语句实现 4×4 乘法器

```
module MUL4_4_1(X, Y, R);
  input [3:0] X, Y;
  output [7:0] R;
  reg [7:0] R;
  integer i;
```

```
  always @(X or Y)
    begin
      R = 0;
      for (i=1; i<=4; i=i+1)
          if (Y[i-1])
             R = R + (X<<(i-1));
    end
endmodule
```

（2）while 语句

while 语句的语法格式为：

while （循环控制条件表达式）

循环体语句块；

while 语句的执行过程是：计算“循环控制条件表达式”的值，若为“真”，就执行一遍循环体语句块，然后重复这个操作直到“循环控制条件表达式”的值为“假”，结束循环语句。在 while 语句的循环体语句中，一定要有类似“循环变量修改”的表达式。

例 5-20 将例 5-19 中的 for 语句改为 while 语句，同样可以实现 4×4 乘法器。

【例 5-20】while 语句实现 4×4 乘法器

```
module MUL4_4_2(X, Y, R);
  input [3:0] X, Y;
  output [7:0] R;
  reg [7:0] R;
  integer i;
  always @(X or Y)
    begin
      R = 0;
      i = 0;
      while (i<4) begin
        i = i + 1;
        if (Y[i-1])
          R = R + (X<<(i-1));
      end
    end
endmodule
```

（3）repeat 语句

repeat 语句是按照给定的循环次数执行的语句，其语法格式为：

repeat （循环次数表达式）

循环体语句块；

repeat 语句中的“循环次数表达式”的值可以是常数，也可以是变量，该值在 repeat 语句执行前就已确定。例 5-21 是利用 repeat 语句实现的 4×4 乘法器。

【例 5-21】repeat 语句实现 4×4 乘法器

```
module MUL4_4_3(X, Y, R);
  input [3:0] X, Y;
  output [7:0] R;
  reg [7:0] R, XT;
  reg [3:0] YT;
```

```
always @(X or Y)
  begin
    R = 0;
    XT = X;
    YT = Y;
    repeat (4) begin
      if (YT[0]) begin
        R = R + XT; end
      YT = YT >> 1;
      XT = XT << 1;
    end
  end
endmodule
```

（4）forever 语句

forever 语句的语法格式为：

forever　循环体语句块；

forever 语句表示连续不断地执行循环体语句块，只能用在仿真程序中，产生周期性的激励信号，不能进行综合，因此 forever 语句通常用在 initial 语句中。例如，可以利用 forever 语句产生连续不断的时钟信号。

```
reg CLK;
……
initial begin
  CLK = 0;
  forever #10 CLK = ~ CLK;
end
```

5.5.3 任务和函数

在计算机程序语言设计中，经常会把某些重复使用的功能写成一个独立的子程序。主程序需要使用这些功能时，就直接调用子程序，这样可以减少代码量，也使主程序的结构更加简洁、易读。在 Verilog 中，也有类似的思想，就是把一些重复性比较大的结构写成一个类似“子程序”的结构。这样的结构有两类，一类叫做任务（task），另一类叫做函数（function）。Verilog 程序可以在不同的位置调用同一个任务或函数。

5.5.3.1 任务

定义任务的一般语法格式为：

task　任务名；

　　[端口及数据类型说明；]

　　begin 语句块；end

endtask

任务的定义包含在“task……endtask”之内，task 是关键字，后面的“任务名”是该任务在后续使用中的识别标志，任务名后面有分号。“端口及数据类型说明”是此任务的端口定义语句和数据类型说明。端口相当于此任务的“参数”，通过端口可以进行值的传入和传出。task 也可以没有端口。任务中可以使用定义该任务的模块中的全局变量。

任务定义中的语句是顺序语句，多条顺序语句要用“begin…end”括起来。任务语句中

不能出现 always 和 initial 这样的并行语句。以下是一个 task 的定义。

```
task Reverse_Bit;
  input [7:0] X;
  output [7:0] Y;
  integer i;
  begin
      for (i=0; i<=7; i=i+1)
        Y[i] = X[7-i];
  end
endtask
```

在以上例子中，开始处是输入、输出端口的定义。注意端口定义的顺序决定了它们在调用任务时的顺序。

任务调用语句的一般语法格式为：

任务名 （端口 1，端口 2，…，端口 n）；

任务调用语句中的端口名的排列顺序要与任务定义中的端口顺序相匹配。任务调用语句是过程性语句，因此其中的端口必须是 reg 类型。例如对以上 Reverse_Bit 的调用语句可以写成：

```
reg [7:0] A, B;
……
Reverse_Bit(A, B); //A 对应 X，是输入数据；B 对应 Y，是位反转后的输出数据；
```

例 5-22 是一个包含任务定义和调用的实例。该任务的功能是计算一个 8 位数据中包含的 1 的个数，在模块中被调用了两次，最后输出数值较大者。

【例 5-22】任务定义和调用实例

```
module TASKDEMO(Din1, Din2, R);
  input [7:0] Din1, Din2;
  output [2:0] R;
  reg [2:0] R, Dout1, Dout2;
  task Bit_Num;
    input [7:0] X;
    output [2:0] Y;
    integer i;
    begin
      Y = 3'b000;
      for (i=0; i<=7; i=i+1)
        if (X[i])
          Y = Y + 1;
    end
  endtask

  always @(Din1, Din2)
    begin
      Bit_Num (Din1, Dout1);
      Bit_Num (Din2, Dout2);
    end

  assign R = (Dout1>=Dout2) ? Dout1 : Dout2;
endmodule
```

5.5.3.2 函数

定义函数的一般语法格式为：

function [返回值位宽] 函数名；

输入端口说明；

[其他说明；]

begin 语句块；end

endfunction

函数的定义包含在“function……endfunction”之内，function 是关键字，“返回值位宽”说明返回值的类型和二进制位数。如果没有“返回值位宽”，则默认为返回值是 1 位二进制数。“函数名”是该函数在后续使用中的识别标志，函数名后面有分号。

“输入端口说明”给出了函数的输入端口定义和数据类型。函数可以包含多个输入端口，但至少包含一个输入端口，但不能包含常规意义上的输出端口和双向端口。函数的输入端口相当于此函数的“参数”，通过输入端口可以将值传递给函数。

函数定义中的语句是顺序语句，多条顺序语句要用“begin…end”括起来。与任务语句一样，函数中也不能出现 always 和 initial 这样的并行语句。以下是一个 function 定义，将任务 Reverse_Bit 改写成了函数 Reverse_Bit。

```
function [7:0] Reverse_Bit;
  input [7:0] X;
  integer i;
  begin
      for (i=0; i<=7; i=i+1)
        Reverse_Bit[i] = X[7-i];
  end
endfunction
```

从上面的例子可以看出，虽然函数没有直接定义输出端口，但是其函数名隐含地就是一个寄存器类型变量，在函数内部可以直接对这个变量赋值。函数的调用是把函数名作为一个操作数来使用，没有独立的调用语句。其调用格式为：

函数名 （输入端口 1，输入端口 2，…，输入端口 n）

例如，对于以上定义的函数 Reverse_Bit，调用的语句为：

```
reg [7:0] A，F;
……
F = Reverse_Bit(A); //A 对应输入参数 X；Reverse_Bit(A)是位反转后的输出数据；
```

因为函数调用是被当作一个操作数来使用，所以在过程语句或者连续赋值语句中都可以调用函数，一个函数内部也可以调用其他的函数，但不能调用任务。例 5-23 是一个包含函数定义和调用的实例。该函数的功能是对两个无符号数进行比较，返回较大的值。

【例 5-23】函数定义和调用实例

```
module FUNCTIONDEMO(Din1, Din2, R);
  parameter  N = 8;
  input [N-1:0] Din1, Din2;
  output [N-1:0] R;
  function [N-1:0] MAX_Num;
    input [N-1:0] X, Y;
    begin
      if (X>=Y)
```

```
            MAX_Num = X;
        else
            MAX_Num = Y;
    end
  endfunction
  assign R = MAX_Num(Din1, Din2);
endmodule
```

5.5.4 模块例化

在 Verilog 程序设计中，可以重复使用一些已经设计好的基本的电路模块，以提高设计效率，使设计更有层次。模块例化（Module Instantiation）就是指在当前的设计中调用已经设计完成的其他模块的过程，它是 Verilog 设计中结构化描述方式的基础。

5.5.4.1 模块例化语句

在模块例化中，被调用的模块也叫做实例（Instance）、实体或元件，它应该在模块例化之前就已经设计完成了。下面我们先设计一个半加器，并将它作为元件来调用，进而设计一个全加器。

例 5-24 是一个半加器的 Verilog 程序。其中 A、B 是两个 1 位输入信号，R 是 A、B 相加的和，C 是 A、B 相加的进位。该程序中采用了两个 assign 语句，它们的顺序是可调换的。

【例 5-24】半加器实例

```
module H_Adder(A, B, R, C);
  input A, B;
  output R, C;
  assign R = A ^ B;
  assign C = A & B;
endmodule
```

半加器设计好了以后，它就可以当作一个元件，被其他设计调用。在调用的时候，模块名 H_Adder 就是元件名。

例 5-25 是采用模块例化方式，调用半加器 H_Adder 的 Verilog 程序。其中 a、b 是两个相加的 1 位输入信号，cin 是输入进位信号，r 是全加器的和，cout 是输出进位信号。

【例 5-25】全加器实例

```
module F_Adder(a, b, cin, r, cout);
  input a, b, cin;
  output r, cout;
  wire R1, C1, C2;
  H_Adder U1(.A(a), .R(R1), .B(b), .C(C1)); //模块例化语句 1，端口名关联法
  H_Adder U2(R1, cin, r, C2); //模块例化语句 2，位置关联法
  assign cout = C1 | C2;
endmodule
```

在例 5-25 中，有三个并行语句，其中一个是 assign 语句，另外两个就是模块例化语句。每个模块例化语句调用一次元件，可以用多个模块例化语句对同一个元件进行多次调用。例 5-25 就是用两个例化语句，对半加器 H_Adder 调用了两次。模块例化语句的语法格式有以下两种：

元件名　例化名　（.元件端口名 1（外接端口名 1），……）；//端口名关联法

元件名　例化名　（端口名 1，端口名 2，…，端口名 n）；// 位置关联法

书写模块例化语句时，首先要写出被调用的元件名，也就是被调用的模块名。比如全加器的程序要调用半加器，则元件名就是半加器的模块名 H_Adder。元件名之后是例化名，就是给此次调用事件取一个名称。这个例化名由用户自己来取，应满足 Verilog 标识符的要求，不能重复，比如 U1、U2 等。例化名之后是由括号括起来的端口关联表。

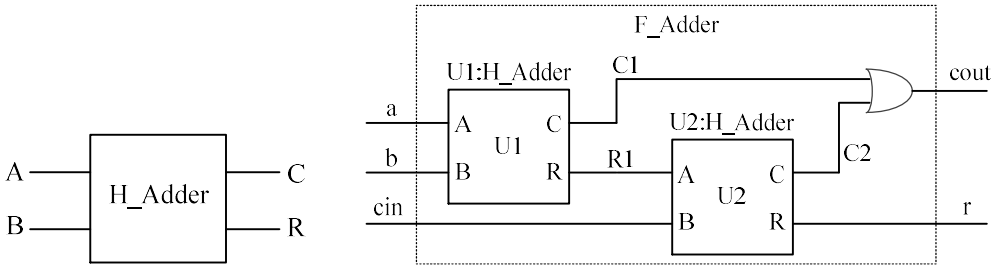
端口关联表有端口名关联法和位置关联法两种书写格式。例 5-25 中的第一个例化语句采用的是端口名关联法，即：

```
H_Adder U1(.A(a), .R(R1), .B(b), .C(C1));
```

端口关联表中不仅出现了当前设计的端口名或线网变量，如 a、b、R1、C1，还出现了元件 H_Adder 的端口名，如 A、B、R、C，并且它们之间有明确的对应关系。比如.A(a)表示元件端口 A 与当前设计的端口 a 有连接关系，.R(R1)表示元件端口 R 与当前设计的线网变量 R1 有连接关系。由于元件端口和当前设计端口的连接关系有明确的说明，所以端口关联表的端口排列顺序可以改变，即可以写成(.A(a), .B(b), .R(R1), .C(C1))，也可以写成其他顺序，如(.A(a), .R(R1), .B(b), .C(C1))。

例 5-25 中第二个例化语句采用的是位置关联法， 端口关联表中只出现当前设计的端口名或变量，例如，cin、r 是全加器的端口，R1、C2 是全加器内部的线网型变量，没有出现元件 H_Adder 的端口名 A、B、R、C，此时元件端口和当前设计端口的连接关系通过位置隐含地进行说明。比如，元件 H_Adder 的端口是按照 A、B、R、C 来排列，例化语句的端口关联表是按照 R1、cin、r、C2 来排列，则处于第一个位置的 R1 和 A 就通过相同的位置有了连接关系，依次类推，cin 连接 B，r 连接 R，C2 连接 C。使用位置关联法时，端口关联表的顺序不能随意变动。

元件例化语句也可以用图 5-10 来说明。在图 5-10 中，（a）是元件 H_Adder 的端口示意图，（b）是全加器的端口示意图及内部结构图。在图（b）中，U1 是第一个例化语句调用的元件，U2 是第二个例化语句调用的元件。全加器定义的三个线网变量 R1、C1、C2 分别与三条内部连线相对应。通过图（b），可以看出元件端口与全加器端口及内部线网变量的连接关系。



（a）半加器端口　　（b）全加器端口和内部结构

图 5-10 半加器和全加器结构

在调用元件时，元件的某些端口如果不需要连接，可以写成诸如.A( )的形式，括号内是空的，表示该端口悬空。悬空的端口如果是输入端口，其值为高阻态 z；如果是输出端口，表示该端口不使用。

在图 5-10 的结构中，全加器调用了半加器，形成了层次结构，半加器为低层，全加器为高层。这里的“低层”和“高层”是相对的，如果全加器作为元件被其他模块调用，那么全加器又变成了低层。在一个电路设计中，这种层次结构可以有若干层，处于最高层的层级叫做顶层（Top level）。

作为元件的模块与当前的设计模块可以存在同一个文件中，也可以分开存为两个文件。当它们存在同一个文件中时，要分清哪个是高层模块，哪个是低层模块，文件名要与高层的模块名相同。当它们分开存放时，文件名为各自的模块名。一般推荐分开存放以使设计的层

次化更清晰。

5.5.4.2 参数化模块例化语句

模块例化方式可以使用已有的设计成果，提高设计效率。但是一个模块设计完成以后，往往功能就固定了，有时不符合新设计的需求。比如已有一个 4×4 乘法器，但是新设计中需要一个 8×8 乘法器，就无法直接调用 4×4 乘法器。

为了解决以上问题，在设计元件时可以加上一些参数，形成参数化元件。模块例化的时候可以通过带参数的例化语句或参数定义语句（defparam）来实现不同功能的元件。以下我们就以参数化 4×4 乘法器为例进行说明。

例 5-26 是带参数的 4×4 乘法器程序，其中的参数就是被乘数、乘数的位宽，用 W 表示。与例 5-19 相比，带参数的元件设计是在程序说明部分先用 parameter 定义参数，程序中凡是跟位宽有关的地方都用 W 表示。

【例 5-26】带参数的 4×4 乘法器

```
module MULTIPLER4 (X, Y, R);
 parameter W=4;
 input [W:1] X, Y;
 output [2*W:1] R;
 reg [2*W:1] R;
 integer i;
 always @(X or Y)
  begin
     R = 0;
     for (i=1; i<=W; i=i+1)
         if (Y[i])
            R = R + (X<<(i-1));
  end
endmodule
```

在 Verilog-2001 标准中，允许在模块名之后直接添加参数表达式#(parameter W=4)，写成如下的形式：

```
module MULTIPLER4 #(parameter W=4)(X, Y, R);
```

1.带参数的例化语句

例 5-26 是一个完整的 4×4 乘法器程序。若将该乘法器作为元件进行例化，得到一个 8×8 乘法器，则采用带参数的例化语句方式，程序可以写成例 5-27 的形式。

【例 5-27】带参数的例化语句

```
module MULTIPLER8_1 (X1, Y1, R1);
 input [7:0] X1, Y1;
 output [15:0] R1;
    MULTIPLER4 #(.W(8)) L1(.X(X1), .Y(Y1), .R(R1));
endmodule
```

例化语句中的#(.W(8))就是参数传递表，将 8 传递给 W，这样就可以得到一个 8×8 乘法器。如果元件程序中定义了多个参数，如：

```
module PARADEMO #(parameter W1=4, parameter W2=2)(X, Y, R);
```

那么例化语句可以写成：

```
PARADEMO #(.W1(8), .W2(3)) L1(.X(X1), .Y(Y1), .R(R1));
```

2. 参数定义语句

用参数定义语句 defparam 也可以对元件中定义的参数进行赋值。defparam 的语法格式为：

defparam　例化名 . 参数名 = 参数值；

采用参数定义语句，例 5-27 可以改成为例 5-28 的形式。

【例 5-28】参数定义语句

```
module MULTIPLER8_2 (X1, Y1, R1);
  input [7:0] X1, Y1;
  output [15:0] R1;
  defparam L1.W = 8;
     MULTIPLER4  L1(.X(X1), .Y(Y1), .R(R1));
endmodule
```

如果元件程序中定义了多个参数，用 defparam 可以对多个参数赋值，例如：

```
defparam L1.W1 = 8, L1.W2 = 4;
```

5.5.4.3 基本库元件例化语句

以上两个小节介绍了对已经设计完成的模块进行重复利用的方法，即模块例化方法，其中作为元件的模块需要我们预先进行设计，然后才能调用。在 Verilog 语言库中，已经对一些基本的门电路进行了定义，这些模块不需要我们再进行设计了，可以直接在程序中进行调用。

基本库元件模块包括以下一些类型：

- 多输入门：and、nand、or、nor、xor、xnor
- 多输出门：buf、not
- 三态门：bufif0、bufif1、notif0、notif1
- 上拉、下拉电阻：pullup、pulldown
- MOS 开关：cmos、nmos、pmos、rcmos、rnmos、rpmos
- 双向开关：tran、tranif0、tranif1、rtran、rtranif0、rtranif1

1．多输入门

多输入门有 6 个，分别是与门（and）、与非门（nand）、或门（or）、或非门（nor）、异或门（xor）和同或门（xnor）。这类基本门有多个输入端口，但只有一个输出端口。多个输入信号进行与、或、异或、同或逻辑操作的处理方式见图 5-5，与非、或非逻辑操作的处理方式见图 5-11。多输入门的例化语句格式为：

基本门元件名　[例化名]　（输出端口名，输入端口名 1，输入端口名 2，…）；

其中，例化名可以省略。输出端口默认放在端口关联表的第一个，后面是若干个输入端口。例如：

```
and  U1 (dout, din1, din2); //两输入与门
or  U2 (dout, din1, din2, din3); //三输入或门
xor  (dout, din1, din2); //两输入异或门，省略了例化名
```

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

图 5-11 多输入门 nand 和 nor 的逻辑处理方式

2．多输出门

多输出门有 2 个：缓冲门（buf）和非门（not）。这两个基本门只有一个输入端口，但可以有一个或多个输出端口。多输出门的逻辑处理方式如图 5-12 所示，结构如图 5-13 所示，其例化语句格式为：

基本门元件名 [例化名] （输出端口名 1，…，输出端口名 n，输入端口名）；

其中，例化名可以省略。输入端口默认放在端口关联表的最后，前面是若干个输出端口。例如：

```
buf  B1 (dout1, dout2, dout3, din); //三输出缓冲门
not  N1 (dout1, dout2, din); //两输出非门
```

buf	0	1	x	z
输出	0	1	x	x

not	0	1	x	z
输出	1	0	x	x

图 5-12 buf 和 not 的逻辑处理方式

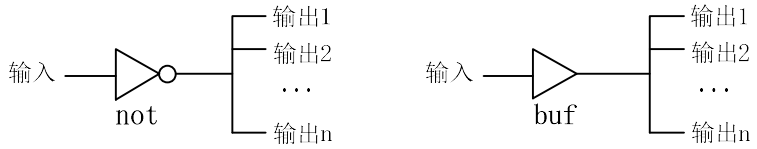


图 5-13 buf 和 not 的结构示意图

3．三态门

三态门有 4 个，分别是 bufif0、bufif1、notif0、notif1。这类基本门可以对驱动器进行建模，有一个输出端口、一个数据输入端口、一个控制输入端口，其例化语句格式为：

三态门元件名 [例化名] （输出端口名，输入数据端口名，输入控制端口名）；

其中，例化名可以省略。端口关联表的第一个是输出端口，第二个是数据输入端口，第三个是控制输入端口。三态门功能表如表 5-13 所示。

表 5-13 三态门功能

三态门	控制输入端口	输出端口
bufif0	0	输入数据值
	1	z
bufif1	0	z
	1	输入数据值
notif0	0	输入数据的非值
	1	z
notif1	0	z
	1	输入数据的非值

以下是调用三态门的两个例化语句。在第一个例化语句中，三态门的控制输入 inCTRL1 为逻辑 0 时，输出 outA1 为高阻状态 z，否则，输入 inB1 传递给 outA1；在第二个例化语句中，三态门的控制输入 inCTRL2 为逻辑 1 时，输出 outA2 为高阻状态 z，否则，输入 inB2 的值取反，传递给 outA2。这两个例化语句描述的电路结构如图 5-14 所示。

```
bufif1  BF1 (outA1, inB1, inCTRL1);
notif0  NT1 (outA2, inB2, inCTRL2);
```

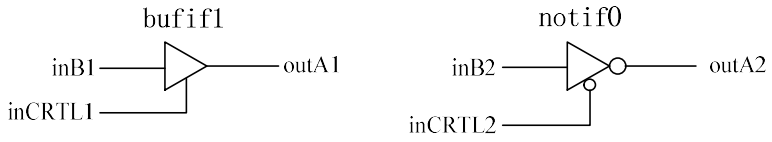


图 5-14　bufif1 和 notif0 的结构示意图

4．上拉、下拉电阻

上拉、下拉电阻类型的基本门分别用 pullup、pulldown 表示，只有输出端口，没有输入端口。上拉电阻将输出置为 1，下拉电阻将输出置为 0，其例化语句格式为：

上拉/下拉电阻元件名　[例化名]　（输出端口名）；

例如：

```
pulldown  PD (A);
```

此例化语句就是将 A 置为低电平。

5．MOS 开关

MOS 开关包括 cmos、nmos、pmos、rcmos、rnmos、rpmos 几种类型，可以为单向开关建模。nmos 是 n 类型 MOS 管，pmos 是 p 类型 MOS 管，rnmos、rpmos 中的“r”表示电阻，是在输入引线和输出引线之间引入高阻抗。这四种 MOS 开关元件有一个输出端口、一个输入端口和一个控制端口，其例化语句格式为：

MOS 开关元件名　[例化名]　（输出端口名，输入端口名，控制端口名）；

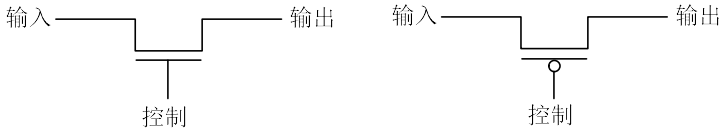
如果 nmos、rnmos 的控制输入为 0，pmos、rpmos 的控制输入为 1，那么开关关闭，输出为高阻 z。否则，输入数据传送至输出端。例如：

```
nmos  N1 (outA1, inB1, inCTRL1);
rpmos  RP1 (outA2, inB2, inCTRL2);
```

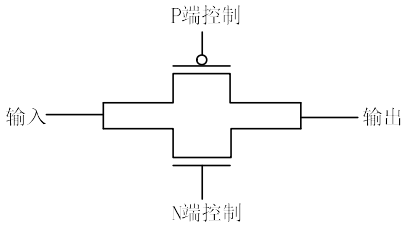
cmos、rcmos 是互补型 MOS 管，有一个输出端口、一个输入端口和两个控制端口，其例化语句格式为：

(r)cmos　[例化名]　（输出端口名，输入端口名，n 控制端口名，p 控制端口名）；

图 5-15 给出了几种 MOS 开关的结构示意图。



（a）nmos 开关　　（b）pmos 开关



（c）cmos 开关

图 5-15　几种 MOS 管结构示意图

6．双向开关

双向开关包括 tran、rtran、tranif0、tranif1、rtranif0、rtranif1。tran 和 rtran 是无条件双向开关，不能被关闭；后四种双向开关可以通过控制信号来关闭。双向开关的例化语句格式有以下两种：

开关元件名　[例化名]　（端口名 1，端口名 2）；//tran、rtran 例化语句格式

开关元件名　[例化名]　（端口名 1，端口名 2，控制端口名）；//其他开关例化语句

tran 和 rtran 的端口表只有两个端口，可以无条件地双向传输信号。其他双向开关的端口列表中，第一个参数和第二个参数是双向数据端口，最后一个是控制端口。如果 tranif0 和 rtranif0 的控制端是 1，tranif1 和 rtranif1 的控制端是 0 的话，则禁止数据双向流动。对于 rtran、rtranif0、rtranif1 类型的开关，信号通过开关传输时，信号强度有所减弱。

5.5.5 块语句

在前面的一些程序实例中，出现了 begin…end 结构，这就是一个块语句结构。块语句就是把多条语句组合成语法结构上相当于一条语句的机制。块语句分为顺序块语句和并行块语句两种类型。

5.5.5.1 顺序块语句

顺序块语句处于 initial 或 always 引导的过程语句中，其一般语法格式为：

```
begin　[：块名]
    [块内变量、参数定义];
    语句 1；
    语句 2；
    ……
    语句 n；
end
```

书写顺序块语句的时候，可以给这个块语句取一个名字，也可以把块名省略掉。在块语句内部可以根据需要定义块内变量或参数，但是这些变量或参数只能在该块内使用，是局部量。顺序块语句中的语句必须是顺序语句，如 if 语句、case 语句、赋值语句等，这些语句按顺序执行或启动。当块中最后一条语句执行完以后，该语句块才结束。例如：

```
begin : SBLK
  reg [3:0] A;
  A = B & C;
  E = ~A;
end
```

如果只有一条顺序语句，则可以不加顺序块语句。以下是仿真中常常采用的产生输入信

号波形的顺序块语句。假设仿真时间单位为 10ns，在 0 时刻，t 值为 0，经过 2 个时间单位，即 20ns，t 值变为 1，再经过 3 个时间单位，即 30ns，t 值变为 0。依此类推，得到的信号波形如图 5-16 所示。顺序块语句中的赋值语句是顺序执行的，因此其时间延迟量是累加的。

```
begin //顺序块语句开始，假设时间单位为 10ns;
  t = 0;
  #2 t = 1;
  #3 t = 0;
  #1 t = 1;
  #5 t = 0;
end
```

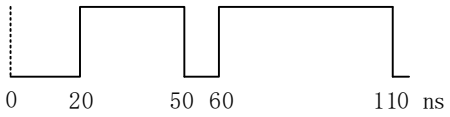


图 5-16 顺序语句块产生的波形

5.5.5.2 并行块语句

并行块语句采用的标识结构为 fork…join，里面包含的语句是并行执行的，其一般语法格式为：

fork [：块名]
[块内变量、参数定义];
语句 1；
语句 2；
……
语句 n；
join

并行语句块的块名和块内变量、参数定义与顺序语句块相同，但其内部各语句的执行是并行的。当并行语句块中执行时间最长的语句完成之后（并不一定是最后的语句），就跳出整个语句块。例如：

```
fork //并行块语句开始，假设时间单位为 10ns;
  t = 0;
  #2 t = 1;
  #5 t = 0;
  #6 t = 1;
  #11 t = 0;
join
```

在以上的例子中，赋值语句是并行执行的，时间延迟量都是针对 0 时刻的，得到的信号波形与图 5-16 相同。

5.6 Verilog HDL 程序设计

以上各小节详细介绍了 Verilog HDL 的程序结构、语言要素及主要语句。在此基础上，本节将通过具体实例来进一步阐述如何利用 Verilog HDL 进行数字电路的设计。

5.6.1 组合电路设计

组合电路设计是数字电路设计中最简单、最基本的设计。所谓组合电路是指数字电路在任何时刻的输出仅取决于该时刻的输入，而与电路先前时刻的状态无关。组合电路的设计主要有简单门电路、编码器/译码器、选择器、比较器及算术运算电路等。

5.6.1.1 简单门电路

简单的门电路主要有与门、或门、非门/反相器、与非门、或非门以及异或门等。这些基本门电路可以采用逻辑运算符、行为描述语句来实现，也可以调用库元件来实现。下面以二输入的异或门为例来说明，其它简单门电路的设计与此类似。

二输入异或门电路的逻辑表达式为：Y=A⊕B，其电路符号如图 5-17 所示，真值表如表 5-14 所示。

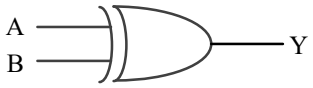


图 5-17 二输入异或电路符号

表 5-14 二输入异或真值表

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

例 5-29 为二输入异或门电路的三种实现方式。方法一是根据二输入异或门电路的逻辑表达式进行描述的，采用了逻辑运算符。方法二是根据二输入异或门电路的输入、输出行为来描述的。方法三采用模块例化语句调用库元件 xor。这三种方法分别是数据流描述、行为描述和结构化描述。

【例 5-29】异或门电路的 Verilog 实现

```
module XOR_1 (A, B, Y); //方法一，采用逻辑运算符
  input A, B;
  output Y;
  assign Y = A ~^ B;
endmodule

module XOR_2 (A, B, Y); //方法二，采用输入、输出行为描述
  input A, B;
  output Y;
  reg Y;
  always@(A, B)
      case ({A, B})
          2'b00: Y <= 0;
          2'b01: Y <= 1;
          2'b10: Y <= 1;
          2'b11: Y <= 0;
```

```
            Default: Y <= x;
        endcase
endmodule

module XOR_3 (A, B, Y); //方法三，调用库元件
  input A, B;
  output Y;
      xor U1 (Y, A, B);
endmodule
```

5.6.1.2 编码器和译码器

在组合逻辑电路设计过程中，常常会用基本门电路构建更复杂一些的电路，如编码器、译码器和选择器等。这些电路已成为像基本门电路一样重要的逻辑电路。下面以 8-3 编码器和七段数码管译码器为例来介绍编码器和译码器的设计。

编码器就是把 2^n 个输入信号转化为 n 位编码输出，如 8-3 编码器。8-3 编码器的逻辑电路如图 5-18 所示，其真值表如表 5-15 所示，例 5-30 为其 Verilog 程序。本例采用的是正电平编码方式，也可以采用负电平来进行编码。

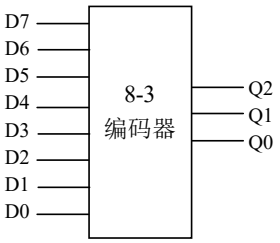


图 5-18　8-3 编码器框图

表 5-15　8-3 编码器真值表

D7	D6	D5	D4	D3	D2	D1	D0	Q2	Q1	Q0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

【例 5-30】8-3 编码器的 Verilog 程序

```
module Coder83 (D, Q);
  input [7:0] D;
  output [2:0] Q;
  reg [2:0] Q;
  always@(D)
      case (D)
          8'b00000001: Q <= 3'b000;
          8'b00000010: Q <= 3'b001;
```

```
            8'b00000100: Q <= 3'b010;
            8'b00001000: Q <= 3'b011;
            8'b00010000: Q <= 3'b100;
            8'b00100000: Q <= 3'b101;
            8'b01000000: Q <= 3'b110;
            8'b10000000: Q <= 3'b111;
            Default: Q <= 3'bxxx;
        endcase
    endmodule
```

译码器可以看作是编码器的逆过程，比如 3-8 译码器就是 8-3 编码器的逆过程。七段数码管是用来显示十六进制数 0~F 及小数点的显示设备，它的内部有八个发光二极管，其中七个发光二级管用来显示组成数字的段 a、b、c、d、e、f、g，一个发光二极管用来显示小数点 dp，另外还有一个公共输入端 COM，如图 5-19 所示。当所有发光二极管的正极连接在一起且负极分开时，称为共阳极数码管；当所有的负极连接在一起且正极分开时，称为共阴极数码管。图 5-19 所示为共阳极数码管，我们以此为例来进行介绍。七段数码管译码器的真值表如表 5-16 所示，其中 dp 不参与译码。当共阳极 COM 接高电平，dp 端接低电平时，小数点变亮，反之，小数点不亮。例 5-31 为此译码器的 Verilog 程序。

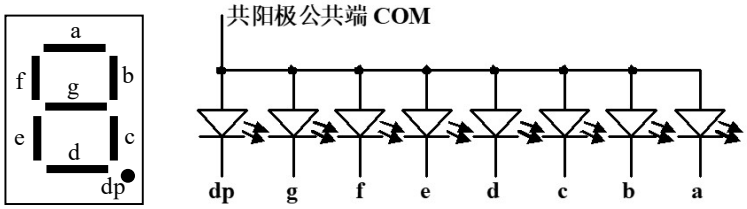


图 5-19 七段共阳数码管示意图

表 5-16 七段数码管译码真值表（共阳极）

q3	q2	q1	q0	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0
1	0	1	0	0	0	0	1	0	0	0
1	0	1	1	1	1	0	0	0	0	0
1	1	0	0	0	1	1	0	0	0	1
1	1	0	1	1	0	0	0	0	1	0
1	1	1	0	0	1	1	0	0	0	0
1	1	1	1	0	1	1	1	0	0	0

【例 5-31】七段数码管译码器的 Verilog 程序

```
module Seg7Led (q, A, B, C, D, E, F, G);
  input [3:0] q;
  output  A, B, C, D, E, F, G;
  reg [6:0] ATOG;
  always@(q)
  begin
      ATOG = 7'b0000000;
      case (q)
          4'b0000: ATOG = 7'b0000001;
          4'b0001: ATOG = 7'b1001111;
          4'b0010: ATOG = 7'b0010010;
          4'b0011: ATOG = 7'b0000110;
          4'b0100: ATOG = 7'b1001100;
          4'b0101: ATOG = 7'b0100100;
          4'b0110: ATOG = 7'b0100000;
          4'b0111: ATOG = 7'b0001111;
          4'b1000: ATOG = 7'b0000000;
          4'b1001: ATOG = 7'b0000100;
          4'b1010: ATOG = 7'b0001000;
          4'b1011: ATOG = 7'b1100000;
          4'b1100: ATOG = 7'b0110001;
          4'b1101: ATOG = 7'b1000010;
          4'b1110: ATOG = 7'b0110000;
          4'b1111: ATOG = 7'b0111000;
          Default: ATOG = 7'b0000000;
      endcase
  end

  assign A = ATOG[6];
  assign B = ATOG[5];
  assign C = ATOG[4];
  assign D = ATOG[3];
  assign E = ATOG[2];
  assign F = ATOG[1];
  assign G = ATOG[0];
endmodule
```

5.6.1.3 选择器

选择器又称数据开关，其功能是把多路并行输入信号的一路送到唯一的输出线上。通常选择器有 2^n 个并行输入、一个输出及 n 个数据选择控制。选择哪一路输入信号输出是由数据选择控制来决定的。下面以 4 选 1 选择器为例，来说明选择器的设计方法。

图 5-20 给出了 4 选 1 选择器的电路示意图，表 5-17 为其功能表。例 5-7 已经给出了用 case 语句设计的 4 选 1 选择器，这里我们采用条件操作符来描述 4 选 1 选择器，如例 5-32 所示。4 选 1 选择器还可以采用 if 语句或者库元件调用等方式来进行设计。

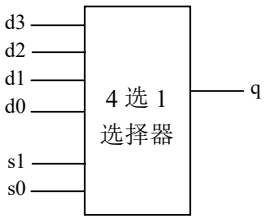


图 5-20　4 选 1 选择器示意图

表 5-17　4 选 1 选择器功能表

s1	s0	q
0	0	d0
0	1	d1
1	0	d2
1	1	d3

【例 5-32】4 选 1 选择器的 Verilog 程序

```
module Selector (d0, d1, d2, d3, s1, s0, q);
  input d0, d1, d2, d3, s1, s0;
  output q;
  wire T1, T2;
  assign  T1 = S0 ? d1 : d0;
  assign  T2 = S0 ? d3 : d2;
  assign  q = S1 ? T2 : T1;
endmodule
```

5.6.1.4 比较器

当对两路或以上的信号判断大小时，就要用到比较器。比较器是数字逻辑电路设计中常用的电路，一般有两种形式：一种只判断两路信号是否相等，另一种就是判断两路信号是否相等和比较大小。下面通过例 5-33 和 5-34 来说明两路 8 位信号的比较器设计。

【例 5-33】只判断是否相等的比较器

```
module Equal (a, b, q);
  input [7:0] a, b;
  output q;
  assign  q = (a == b) ? 1'b1 : 1'b0;
endmodule
```

例 5-33 比较 a 和 b 是否相等，相等时输出 q 为 ‘1’，否则为‘0’。

【例 5-34】判断相等和大小的比较器

```
module Comp (a, b, q);
  input [7:0] a, b;
  output [2:0] q;
  reg [2:0] q;
  always@ (a, b) begin
    if (a == b) q <= 3'b010;
    else if (a > b) q <= 3'b100;
```

```
    else  q <= 3'b001;
  end
endmodule
```

例 5-34 对 a 和 b 进行比较，如果相等，则 q 为 3'b010（q（1）为‘1’），如果 a 大于 b，则 q 为 3'b100（q（2）为‘1’），如果 a 小于 b，则 q 为 3'b001（q（0）为‘1’）。

5.6.1.5 算术运算电路

算术运算电路是数字电路设计的重要电路之一，常用的有加法器、乘法器、减法器和除法器等。其中加法器和乘法器两种电路是最基础的运算电路，其它的电路基本上都可以通过这两种电路来实现。下面通过例 5-35 和 5-36 来介绍两位全加器和两位乘法器的设计。

例 5-35 描述的是两位全加器的电路模块，其中 a、b 是输入的 2-bit 加数，cin 是进位输入，q 是输出的 2-bit 和值，cout 是进位输出。

【例 5-35】两位全加器

```
module FullAdd2 (a, b, cin, q, cout);
  input [1:0] a, b;
  input cin;
  output [1:0] q;
  output cout;
  wire [2:0] Sum;
  assign Sum = a + b;
  assign q = Sum[1:0];
  assign cout = Sum[2];
endmodule
```

【例 5-36】两位乘法器

```
module MUL2_2(M, N, R);
  input [1:0] M, N;
  output [3:0] R;
  wire [1:0] T3;
  reg [1:0] T1, T2;
  always @(M or N) begin
    begin
      if (N[0]) T1 = M;
      else T1 = 2'b00;
    end
    begin
      if (N[1]) T2 = M;
      else T2 = 2'b00;
    end
  end

  assign R[0] = T1[0];
  assign T3 = {0, T1(1)};

    FullAdd2  U1(.a(T2), .b(T3), .cin(1'b0), .q(R[2:1]), .cout(R[3]));
endmodule
```

例 5-36 描述的是两位乘法器的电路模块，其中 M、N 是输入的 2-bit 乘数，R 是 4-bit 乘法结果。因为二进制乘法可以转化为移位和加法操作，所以例 5-36 中的元件例化 U1 使用了例 5-35 中的两位全加器。

5.6.2 时序电路设计

数字电路设计中的时序电路是这样一类电路：电路在任一时刻的输出不仅取决于该时刻的输入信号，还取决于先前时刻的状态，即与当前时刻和先前时刻的输入都有关，也可以说时序电路是具有“记忆”能力的电路。时序电路的驱动信号是时钟信号，即时序电路只有在时钟信号的边沿到来时，它的状态才会更新。常用的时序电路模块有触发器、寄存器、计数器、分频器和倍频器等。由于时序电路是时钟驱动的，因此时钟信号的描述及使用是非常重要的，所以在介绍具体的时序电路设计前，先对时钟信号和复位信号进行说明。

5.6.2.1 时钟信号和复位信号

（1）时钟信号描述

时钟信号描述时序电路的执行条件，它一般出现在 always 语句的敏感信号列表里，如 always@ (posedge CLOCK)或者 always@ (negedge CLOCK)。这里 posedge 表示上升沿触发，negedge 表示下降沿触发，CLOCK 是时钟信号。例 5-37 是采用时钟边沿触发方式设计的两种 D 触发器，其中 EDFF1 没有复位信号，EDFF2 有异步复位信号 RST。

【例 5-37】 无复位 D 触发器和异步复位 D 触发器

```
module EDFF1(CLK, D, Q);
  input CLK, D;
  output  Q;
  reg Q;
  always @(posedge CLK)
      Q <= D;
endmodule

module EDFF2(CLK, RST, D, Q);
  input CLK, RST, D;
  output  Q;
  reg Q;
  always @(posedge CLK or negedge RST)
      if (!RST) Q <= 0;
      else Q <= D;
endmodule
```

如果 always 语句的敏感信号表里采用 posedge 或 negedge 边沿触发，那么其他输入端口信号就不能列入敏感信号表了。比如在 EDFF1 中，always 语句的敏感信号表是(posedge CLK)，表示该语句在 CLK 上升沿到来时被触发，那么输入 D 就不能再列入敏感信号表。一旦 CLK 被定义为边沿敏感信号，它就不能在 always 结构中再出现了。

always 语句的敏感信号表中不允许有两个边沿触发信号。在 EDFF2 中，always 语句的敏感信号表是(posedge CLK or negedge RST)，好像是有两个边沿触发信号。但是从程序中可以看到，RST 又出现在 if 语句中，并对其电平进行了判断。如果 RST 是低电平，则 D 触发器复位（negedge RST 就表示 RST 是低电位触发），因此 RST 还是电平敏感信号，与 negedge RST 的表述不一致，这一点要特别注意。

表述为边沿敏感、实际为电平敏感的信号，如 RST，只能作为异步复位或置位信号，不能作为一般的异步逻辑信号。EDFF2 中的 RST 就是异步复位信号，仅作为控制条件判断使用，不能作为一般异步逻辑信号再赋给其他信号。例如“A<=RST”是不允许的。

（2）同步复位和异步复位

触发器的初始状态是由复位信号决定的，复位信号对触发器复位的操作可分为同步复位和异步复位两种。所谓同步复位，是指在给定的时钟边沿到来且复位信号有效时，才会复位，或者说，复位信号也是由时钟触发条件控制来执行的，如例 5-38 所示。异步复位是一旦复位信号有效，就会复位触发器，不管时钟边沿是否到来，如例 5-37 的 EDFF2 所示。

【例 5-38】 同步复位 D 触发器

```
module EDFF3(CLK, RST, D, Q);
  input CLK, RST, D;
  output  Q;
  reg Q;
  always @(posedge CLK)
      if (!RST) Q <= 0;
      else Q <= D;
endmodule
```

5.6.2.2 触发器

在复杂的数字电路设计中，不仅要对输入信号进行逻辑和算术运算，有时还要对这些信号和运算结果进行保存，这时就要设计具有记忆功能的逻辑电路来保存这些数据。触发器就是具有记忆功能的逻辑电路，它可以存储一位数据。常见的触发器有 D 触发器、JK 触发器、T 触发器和 RS 触发器等。下面以具有异步置位和复位的 JK 触发器为例来说明触发器的 Verilog 设计。

具有异步置位和复位的 JK 触发器的电路示意图如图 5-21 所示，其功能表如表 5-18 所示，例 5-39 给出了它的 Verilog 描述。

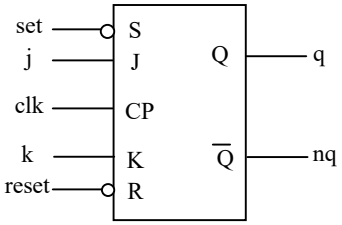


图 5-21 JK 触发器电路示意图

表 5-18 带异步置位/复位的 JK 触发器功能表

S	R	CP	J	K	Q	$\overline{Q}$
0	1	X	X	X	1	0
1	0	X	X	X	0	1
0	0	X	X	X	无效	
1	1	上升沿	0	0	保持	保持
1	1	上升沿	0	1	0	1
1	1	上升沿	1	0	1	0
1	1	上升沿	1	1	翻转	翻转
1	1	0	X	X	保持	保持

【例 5-39】JK 触发器

```
module myJKff (J, K, CLK, SET, RESET, Q, NQ);
  input J, K, CLK, SET, RESET;
  output  Q, NQ;
  reg Q, NQ;

  always @(posedge CLK or negedge SET or negedge RESET)
      if (!SET) begin Q <= 1; NQ <= 0; end
      else if (!RESET) begin Q <= 0; NQ <= 1; end
      else if (SET & RESET) begin
          if ((J) & (!K)) begin Q <= 1; NQ <= 0; end
          else if ((!J) & K) begin Q <= 0; NQ <= 1; end
          else if (J & K) begin Q <= ~Q; NQ <= ~NQ; end
          else begin Q <= Q; NQ <= NQ; end
      end
endmodule
```

5.6.2.3 寄存器

在时序逻辑电路设计中，能存储多位二进制数据的同步时序电路被称为寄存器。寄存器是时序逻辑电路设计中的基本模块，由于它是用来存储多位数据的，所以可以用多个具有一位存储能力的触发器来实现。根据功能和形式的不同，寄存器一般可分为通用寄存器、锁存器和移位寄存器等。从功能和存储数据的角度看，寄存器和锁存器是相同的，但寄存器是同步时钟控制的，而锁存器是电平信号控制的。两者适用的场合一般取决于控制方式、控制信号与数据间的时间关系。若数据滞后于控制信号有效，则使用锁存器；若数据提前于控制信号到达且要求同步操作，则使用寄存器。下面以移位寄存器为例来介绍其 Verilog 设计。

移位寄存器除了具有存储数据的功能外，还具有移位功能。移位寄存器通常用来实现串/并转换、数值运算及数据处理等。常用的移位寄存器主要有串入/串出、串入/并出及循环移位寄存器等。图 5-22 是用 D 触发器构成的串入/并出四位移位寄存电路示意图，例 5-40 是其 Verilog 实现。

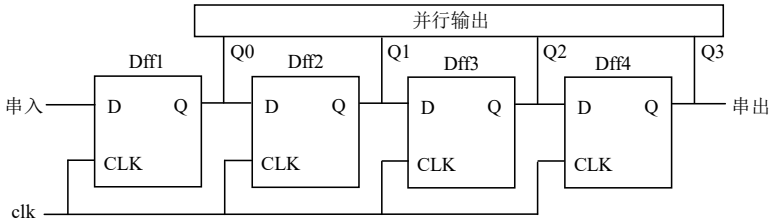


图 5-22　D 触发器构成的串入/并出四位移位寄存器

【例 5-40】串入/并出四位移位寄存器

```
module ShiftReg1 (D, CLK, Q); //调用四个 D 触发器的串入/并出移位寄存器
  input D, CLK;
  output [3:0] Q;
  EDFF1 Dff1 (.CLK(CLK), .D(D), .Q(Q[0]));
  EDFF1 Dff2 (.CLK(CLK), .D(Q[0]), .Q(Q[1]));
  EDFF1 Dff3 (.CLK(CLK), .D(Q[1]), .Q(Q[2]));
  EDFF1 Dff4 (.CLK(CLK), .D(Q[2]), .Q(Q[3]));
endmodule
```

```
module ShiftReg2 (D, CLK, Q); //采用非阻塞赋值语句的串入/并出移位寄存器
  input D，CLK;
  output [3:0] Q;
  reg [3:0] Q;
  always @(posedge CLK)
    begin
      Q[0] <= D;
      Q[1] <= Q[0];
      Q[2] <= Q[1];
      Q[3] <= Q[2];
    end
endmodule
```

本例中的第一种设计调用了例 5-37 中的 D 触发器模块 EDFF1。D 触发器作为一个元件，在移位寄存器的设计中被调用了四次。第二种设计采用了非阻塞赋值语句，也可以综合出四个触发器。

例 5-41 是带参数的循环移位寄存器的 Verilog 实现，它描述的是在每个时钟上升沿寄存器循环右移一位且具有异步并行装载的通用循环移位寄存器。

【例 5-41】通用循环移位寄存器

```
module LoopShiftReg #(parameter Width=8) (CLK, LOAD, D, Q);
  input CLK, LOAD;
  input [Width-1: 0] D;
  output [Width-1: 0] Q;
  reg [Width-1: 0] TQ;
  always @(posedge CLK or posedge LOAD)
    begin
    if (LOAD) TQ <= D;
    else
      TQ <= {TQ[0], TQ[Width-1:1]};
    end
  assign Q = TQ;
endmodule
```

5.6.2.4 计数器

计数器也是时序逻辑电路设计中最基本、最常用的时序电路。计数器的功能是记录时钟脉冲的个数，即利用一组触发器按照一定规律随时钟变化来记录时钟的个数。计数器所能记录的时钟脉冲的最大数目称为计数器的模。计数器主要用于计数、分频、定时、产生节拍脉冲和脉冲序列以及数字运算等。

计数器的种类繁多，可以按照不同的角度进行分类。例如，按照计数器中触发器的时钟工作方式，可分为同步计数器和异步计数器；按照编码方式，可分为二进制计数器、BCD 码计数器、格雷码计数器和循环计数器等。下面主要介绍同步计数器和异步计数器。

同步计数器是指在时钟脉冲的作用下，组成计数器的各个触发器的状态同时发生改变。例 5-42 是四位二进制同步计数器的 Verilog 实现程序，图 5-23 是它的 RTL 原理图，具有异步复位端、同步使能端和进位输出端。

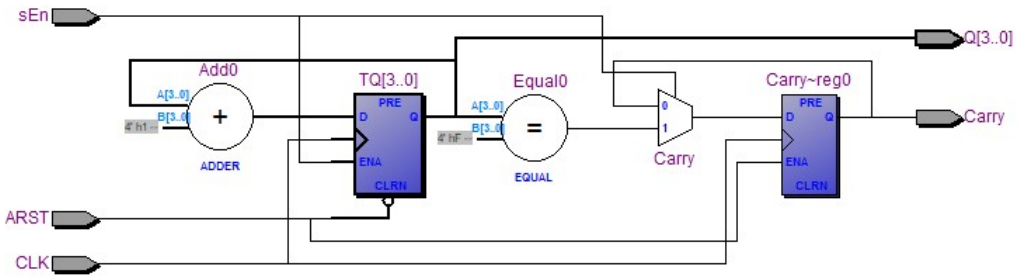


图 5-23 四位同步计数器的 RTL 原理图

【例 5-42】四位二进制同步计数器

```
module SyncCounter (CLK, ARST, sEn, Carry, Q);
  input CLK, ARST, sEn;
  output Carry;
  output [3:0] Q;
  reg Carry;
  reg [3:0] TQ;
      assign Q = TQ;
      always @(posedge CLK or negedge ARST)
      begin
          if (!ARST) TQ <= 4'b0000;
          else if (sEn) begin
              if (TQ==4'b1111)
                  begin
                  Carry <= 1'b1;
                  TQ <= 4'b0000;
                  end
              else
                  begin
                  Carry <= 1'b0;
                  TQ <= TQ + 1;
                  end
          end
      end
endmodule
```

异步计数器是指构成计数器的各个触发器的状态不是在同一时钟的作用下同时改变，而是低位计数触发器的输出作为相邻高位计数触发器的时钟级联起来的，通过低位触发器的输出来触发高位触发器。图 5-24 是利用四个 D 触发器构成的四位异步计数器，例 5-43 为其 Verilog 实现程序。在例 5-43 中，先设计了一个具有异步复位的 D 触发器，然后在四位异步计数器中将 D 触发器作为元件调用。在文件存放时，可以把 D 触发器单独存成一个文件，文件名为 AR_Dff.v，把四位异步计数器存为 AsyncCounter.v，其中.v 是 Verilog 源代码文件的后缀。

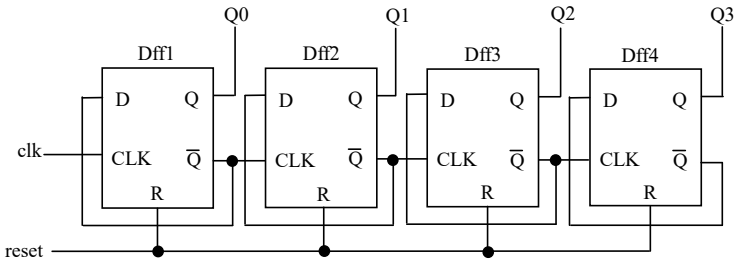


图 5-24　四位异步计数器电路示意图

【例 5-43】四位异步计数器

```
module AR_Dff (CLK, RESET, D, Q, NQ);// 异步复位 D 触发器
  input CLK, RST, D;
  output  Q, NQ;
  reg Q, NQ;
  always @(posedge CLK or posedge RESET)
    if (RST) begin Q <= 0; NQ <= 1; end
    else begin Q <= D; NQ <= ~D; end
endmodule

module AsyncCounter (CLK, RESET, Q);
  input CLK, RESET;
  output [3:0] Q;
  wire [1:4] TQ;
 AR_Dff Dff1 (.CLK(CLK),.RESET(RESET),.D(TQ[1]),.Q(Q[0]),.NQ(TQ[1]));
 AR_Dff Dff2 (.CLK(TQ[1]),.RESET(RESET),.D(TQ[2]),.Q(Q[1]),.NQ(TQ[2]));
 AR_Dff Dff3 (.CLK(TQ[2]),.RESET(RESET),.D(TQ[3]),.Q(Q[2]),.NQ(TQ[3]));
 AR_Dff Dff2 (.CLK(TQ[3]),.RESET(RESET),.D(TQ[4]),.Q(Q[3]),.NQ(TQ[4]));
endmodule
```

5.6.2.5 分频器

在数字逻辑电路设计中，时钟信号非常关键。有时硬件系统只提供一个或几个时钟信号，不能满足复杂的系统设计，这时就要通过倍频来获得更高频率的时钟信号，通过分频来获得较低频率的时钟信号。在基于 CPLD/FPGA 的数字逻辑设计中，倍频是无法通过硬件描述语言来实现的，它主要是采用 FPGA 自带的锁相电路（如 PLL 或 DLL 等）来实现。而分频除了用锁相电路实现外，还可以通过 Verilog 描述来实现，且可实现各种分频电路，如 2 的幂次分频、偶数分频、奇数分频及小数分频等。

在例 5-44 中，分频器的分频系数是 2 的幂次方，可实现 2、4、8、16 等分频电路，输出信号的占空比为 1:1。图 5-25 给出了例 5-44 的时序仿真结果，实现了 2、4、8、16 的分频信号。从例 5-44 可以看出，要实现更大的 2 的幂次方的分频电路，只要增加计数器 cnt 的位数即可。

【例 5-44】2 的幂次方分频器

```
module Power2FD (clk, clk2D, clk4D, clk8D, clk16D);
  input clk;
  output clk2D, clk4D, clk8D, clk16D;
  reg [3:0] cnt;
```

```
  always @(posedge CLK)
    begin cnt <= cnt + 1; end
  assign clk2D = cnt[0];
  assign clk4D = cnt[1];
  assign clk8D = cnt[2];
  assign clk16D = cnt[3];
endmodule
```

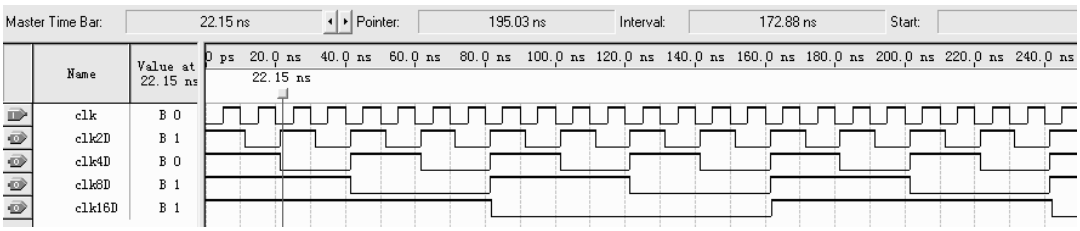


图 5-25　例 5-44 的时序仿真波形

例 5-45 是分频系数为偶数的分频器的 Verilog 描述，此处具体的分频系数是 6。程序中采用了计数的方式，当输入信号的周期计到 3 个时，输出信号就翻转，即输入信号六个周期的时长等于输出信号一个周期的时长，达到 6 分频的目的。以此类推，偶数 N 分频时，只要设置计数器从 0 计到$\frac{N}{2}-1$时对输出的分频时钟进行翻转即可实现任意的偶数分频。图 5-26 是例 5-45 的时序仿真结果，可以看出，输出信号是输入信号的六分频信号。

【例 5-45】偶数分频

```
module SixFD (clk, clk6D);
  input clk;
  output clk6D;
  reg clk6D;
  reg [1:0] cnt;
  always @(posedge clk)
    begin
    if (cnt==2'b10) begin cnt <= 2'b00; clk6D <= ~clk6D; end
    else cnt <= cnt + 1;
    end
endmodule
```

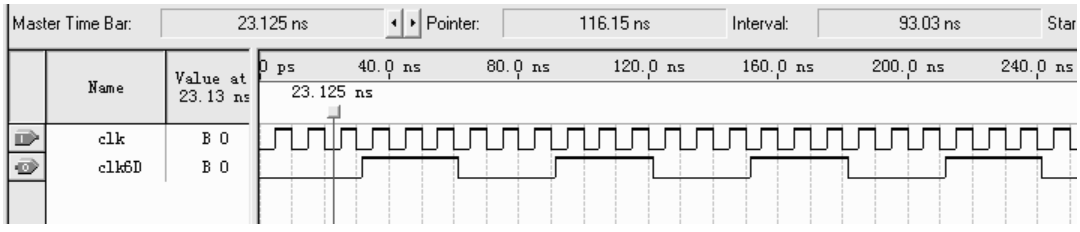


图 5-26　例 5-45 的时序仿真波形

例 5-46 是分频系数为奇数的分频器的 Verilog 描述，此处具体的分频系数是 5。clk 是输入的原始时钟信号，clk5D23R 是 clk 上升沿触发的五分频输出信号，占空比为 2:3，clk5D23F 是 clk 下降沿触发的五分频输出信号，占空比为 2:3，clk5D 是五分频输出信号，占空比为 1:1。图 5-27 是例 5-46 的时序仿真结果。

【例 5-46】奇数分频

```
module FiveFD (clk, clk5D23R, clk5D23F, clk5D);
  input clk;
  output clk5D23R, clk5D23F, clk5D;
  reg clk5D23R, clk5D23F;
  reg [2:0] cnt1, cnt2;
  always @(posedge clk)
    begin
    if (cnt1==3'b010) begin clk5D23R <= 1'b1; cnt1 <= 3'b011; end
    else if (cnt1==3'b100) begin clk5D23R <= 1'b0; cnt1 <= 3'b000; end
    else cnt1 <= cnt1 + 1;
    end
  always @(negedge clk)
    begin
    if (cnt2==3'b010) begin clk5D23F <= 1'b1; cnt2 <= 3'b011; end
    else if (cnt2==3'b100) begin clk5D23F <= 1'b0; cnt2 <= 3'b000; end
    else cnt2 <= cnt2 + 1;
    end

  assign clk5D = clk5D23R | clk5D23F;
endmodule
```

从例 5-46 可看出 N 倍奇数分频电路的描述原理为：首先设计两个模 N 的计数器（其中一个计数器为原始时钟信号的上升沿触发，另一个则为下降沿触发），然后在计数器分别计数到$\frac{N-1}{2}$和 N-1 时对输出时钟进行翻转（或取相反的电平值），这样就可得到两个 N 奇数分频的时钟信号，但其占空比不是 1：1，而是$\frac{N-1}{2}$：$\frac{N+1}{2}$。要得到占空比为 1：1 的 N 倍分频时钟信号，只要将两个计数器产生的两个时钟信号相“或”即可。

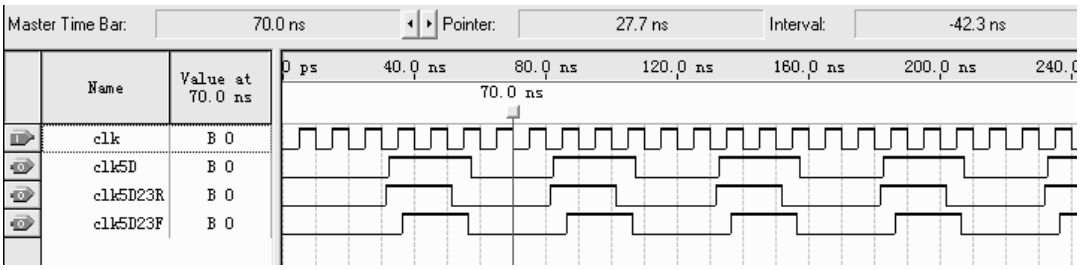


图 5-27　例 5-46 的时序仿真波形

例 5-47 是分频系数为小数的分频器的 Verilog 描述，此处具体的分频系数是 2.3。图 5-28 是例 5-47 的时序仿真结果。

【例 5-47】小数分频

```
module FractionFD (clk, clkout);
  input clk;
  output clkout;
  reg [4:0] cnt;
  always @(posedge clk)
    begin
    case (cnt)
```

```
        0,1,3,5,7,8,10,12,14,15,17,19,21: begin
             clkout <= 1'b0; cnt <= cnt+1; end
        2,4,6,9,11,13,16,18,20,22: begin
             clkout <= 1'b1;
             if (cnt==22) cnt<=5'b00000;
             else cnt <= cnt + 1; end
        default: cnt <= 5'b00000;
    endcase
    end
endmodule
```

小数分频的原理是：在若干个分频周期中采取某种方法使某几个周期少计或多计一个时钟，从而在整个计数周期总体平均上获得一个小数分频。简单地说，就是把小数分频系数 X 扩大 n（一般取 10 的幂次倍）倍后变为整数 N，然后以 N 为模设计计数器。在此计数器计数的同时，对输出时钟信号进行 n 次的翻转操作，即可实现 X 分频操作。翻转操作相当于 X（下取整）分频和 X+1（下取整）分频。另外，在实现翻转操作时要考虑尽可能的使整个输出时钟均匀，这时相当于两种分频信号均匀地组合在一起。

以上两种分频信号组合的一般做法是：用 n 减去 X 的小数部分乘以 n 做计数步长，当计数值没有达到一个步长时，采用 X（下取整）分频；当计数值达到一个步长时，进行一次 X+1（下取整）分频，并重新记录步长。这个过程可以事先计算，然后直接在 Verilog 描述时列出取值即可。在例 5-47 中，以 23 为模设计计数器，步长为 7，计数值小于 7 时，按 2 分频计数，当计数值达到步长 7 时，进行一次 3 分频。3 分频之后，重新统计步长，采用 2 分频计数到 14，再进行一次 3 分频，依此类推。由于分频时计数值不稳定，会造成分频输出抖动大。因此，在精度要求较高的场合，尽量不要采用此法来分频。

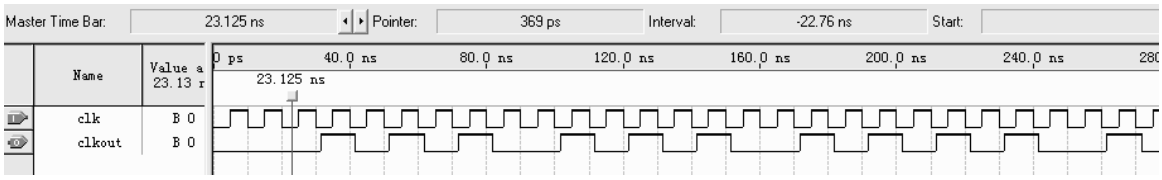


图 5-28　例 5-47 的时序仿真波形

5.6.3 有限状态机

数字系统设计中常常会同时包含组合逻辑电路和时序逻辑电路，其结构如图 5-29 所示。在这种结构中，组合逻辑电路进行运算，产生的数据由寄存器构成的时序逻辑电路来存储和转移。从图 5-29 可见，时序逻辑电路的输入来自组合逻辑电路的一部分输出，这一部分输出信号也被叫做“次态”或“下一状态”。而时序逻辑电路的输出又反馈到组合逻辑的输入端，这一部分反馈信号也被称为“现态”或“当前状态”。因此整个数字系统的工作状态就在有限个数目的稳定的状态中转换，形成了时序逻辑电路的有限状态机（Finite State Machine，FSM）模型。

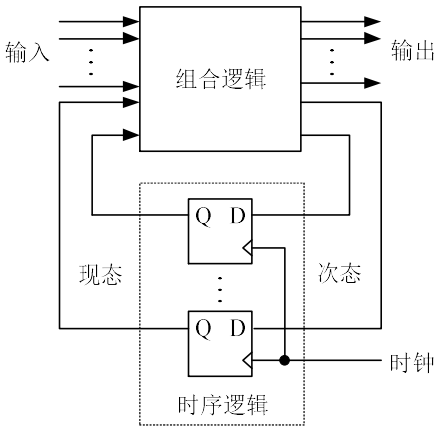
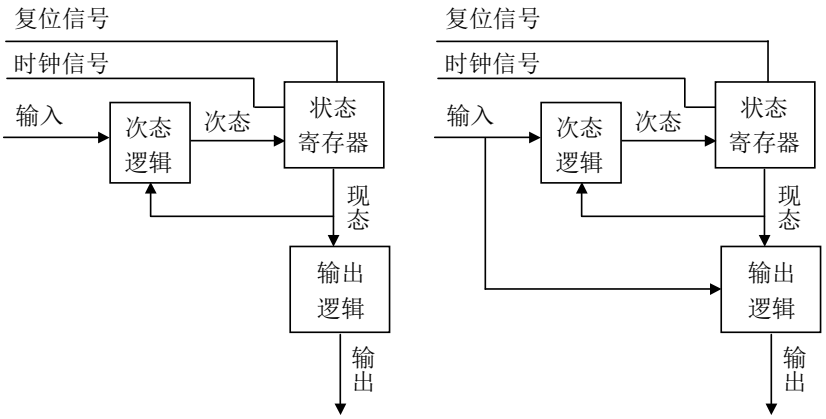


图 5-29 有限状态机模型

有限状态机是数字系统设计中常用的控制单元，其运算速度快，但通用性差，对每个具体的电路都要设计不同的状态机。常用的有限状态机有两种：Moore 状态机和 Mealy 状态机，它们的结构可以改画成图 5-30 的形式。对于 Moore 状态机，其输出只与状态机的当前状态有关，如图 5-30（a）所示；对于 Mealy 状态机，其输出取决于状态机的当前状态和状态机的输入，如图 5-30（b）所示。

用 Verilog 描述有限状态机没有固定的格式，但要遵循一些基本的编码原则：

- 至少包括一个用于指定有限状态机状态的变量；
- 要有一个时钟用于状态同步；
- 要有状态转换条件和输出指定；
- 要有同步或异步复位信号。



（a）Moore 状态机结构图　　（b）Mealy 状态机结构图

图 5-30 两种状态机的结构框图

在用 Verilog 描述有限状态机时，一般使用 always 过程语句和 case 语句，有三种描述方式：单 always 过程、双 always 过程和三 always 过程描述，如表 5-19 所示，表中的次态逻辑、状态寄存器和输出逻辑与图 5-30 中的各部分对应。

单 always 过程描述方式是把次态逻辑、状态寄存器和输出逻辑放在一个 always 过程语句中，代码简单，并有时钟控制，所以输出为寄存器输出，无毛刺，但代码维护和优化困难，占用资源较多。双 always 过程描述方式将组合逻辑和时序逻辑分开，便于资源优化，时间性能好，但组合逻辑输出可能会有毛刺。三 always 过程描述方式的程序结构清晰，在设计同步输出的状态机时，不会产生毛刺，但综合以后的面积比双 always 过程描述方式大。

表 5-19　有限状态机的三种 Verilog 描述方式

描述方式		过程描述的功能	过程个数
单 always 过程描述		单个 always 过程描述次态逻辑、状态寄存器和输出逻辑	1
双 always 过程描述	形式一	always 过程 1：描述次态逻辑和输出逻辑 always 过程 2：描述状态寄存器	2
	形式二	always 过程 1：描述次态逻辑 always 过程 2：描述状态寄存器和输出逻辑	2
	形式三	always 过程 1：描述次态逻辑和状态寄存器 always 过程 2：描述输出逻辑	2
三 always 过程描述		三个 always 过程分别描述次态逻辑、状态寄存器和输出逻辑。	3

下面通过一个状态为（S0，S1，S2，S3）的四状态控制来介绍 Moore 状态机和 Mealy 状态机的 Verilog 描述。此状态机的状态转换如图 5-31 所示，状态转换在时钟上升沿进行，有异步复位信号，复位时状态机处于 S0 状态。例 5-48 为其 Verilog 描述，程序中同时描述了 Moore 状态机和 Mealy 状态机，图 5-32 为时序仿真波形。

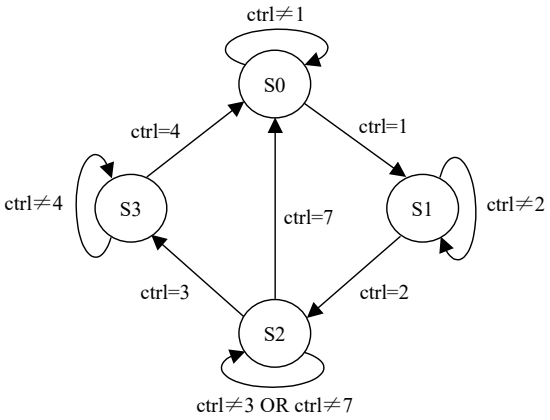


图 5-31　四状态（S0，S1，S2，S3）状态机的状态转换图

【例 5-48】有限状态机之一

```
module FSM1 (clk, reset, ctrl, Moore, Mealy);
  input clk, reset;
  input [2:0] ctrl;
  output [1:0] Moore;
  output Mealy;
  reg [1:0] Moore;
  reg Mealy;
  parameter S0=0, S1=1, S2=2, S3=3; //定义各种状态及其取值
  reg [1:0] st; //定义状态变量
  always @(posedge clk or negedge reset) begin
   if (!reset) st <= S0;
   else begin
     case (st)
          S0: begin
              If (ctrl==3'b001) st <= S1;
              else st <= S0; end
          S1: begin
              If (ctrl==3'b010) st <= S2;
```

```
            else st <= S1; end
        S2: begin
            If (ctrl==3'b011) st <= S3;
            else if (ctrl==3'b111) st <= S0;
            else st <= S2; end
        S3: begin
            If (ctrl==3'b100) st <= S0;
            else st <= S3; end
        default: st <= S0;
    endcase  end
  end

always @(st) begin  //Moore 状态机输出
    case (st)
        S0: Moore <= 2'b00;
        S1: Moore <= 2'b10;
        S2: Moore <= 2'b11;
        S3: Moore <= 2'b01;
        Default: Moore <= 2'b00;
    endcase
    end

always @(st) begin  //Mealy 状态机输出
    if ((st==S2) & (ctrl==6)
        Mealy <= 1;
    else
        Mealy <= 0;
    end
endmodule
```

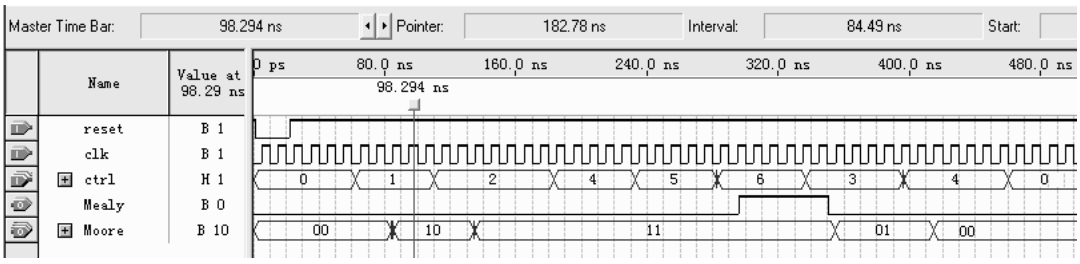


图 5-32　例 5-48 的时序仿真波形

Verilog 描述有限状态机的时候，除了有输入/输出端口的说明之外，还要有各种状态和状态变量的定义。状态机的各种状态取值用 parameter 来定义，要写出具体的值或编码，例如：

```
parameter [1:0] S0=0, S1=1, S2=2, S3=3;
reg [1:0] st;
```

在图 5-31 中，状态有四个，因此可以将其定义为 S0、S1、S2、S3，也可以用其他的标识符来表示这四个状态，如 S4、S5、S6、S7 或者 A00、A10、A20、A30 等，同时要定义各状态的取值。状态的取值也叫做状态编码，常用的有顺序码（二进制码）、格雷码（Gray）、一位热码（One-hot）等。例 5-48 中采用的是顺序码，即用二进制 00、01、10、11 对状态

进行编码。

parameter 后面的[1:0]可以省略掉，因为后面定义状态变量 st 的时候也说明了 st 的宽度。st 是状态变量，其取值只能是 parameter 定义的几种状态。在本例中，st 的取值只能是 S0、S1、S2、S3。

例 5-48 采用的是双 always 过程的描述方式，具体为形式三。第一个 always 过程语句描述的是次态逻辑和状态寄存器，第二个 always 过程语句描述的是输出逻辑。因为本例是把 Moore 状态机和 Mealy 状态机放在一个程序里，所以第一个 always 过程语句和第二个 always 过程语句描述的是 Moore 状态机，第一个 always 过程语句和第三个 always 过程语句描述的是 Mealy 状态机。

例 5-49 采用三 always 过程语句的描述方式来描述 Moore 状态机。三个 always 过程语句分别描述状态寄存器、次态逻辑和输出逻辑。程序中需要定义两个状态变量 current_state 和 next_state，分别表示“现态”和“次态”。

【例 5-49】有限状态机之二

```
module FSM2 (clk, reset, ctrl, Moore);
  input clk, reset;
  input [2:0] ctrl;
  output [1:0] Moore;
  reg [1:0] Moore;
  parameter S0=0, S1=1, S2=2, S3=3; //定义各种状态及其取值
  reg [1:0] current_state, next_state; //定义状态变量

  always @(posedge clk or negedge reset) begin
    if (!reset) current_state <= S0;
    else current_state <= next_state;
  end

  always @(current_state, ctrl) begin
     case (current_state)
          S0: begin
              If (ctrl==3'b001) next_state <= S1;
              else next_state <= S0; end
          S1: begin
              If (ctrl==3'b010) next_state <= S2;
              else next_state <= S1; end
          S2: begin
              If (ctrl==3'b011) next_state <= S3;
              else if (ctrl==3'b111) next_state <= S0;
              else next_state <= S2; end
          S3: begin
              If (ctrl==3'b100) next_state <= S0;
              else next_state <= S3; end
          default: next_state <= S0;
     endcase
     end

always @(current_state) begin
    case (current_state)
       S0: Moore <= 2'b00;
```

```
            S1: Moore <= 2'b10;
            S2: Moore <= 2'b11;
            S3: Moore <= 2'b01;
            Default: Moore <= 2'b00;
        endcase
        end

    endmodule
```

5.7 Verilog HDL 仿真

当一个数字系统设计完成以后，就要对其功能和性能进行评估和检测，仿真就是通常使用的一种检测方式，可以在设计早期发现系统设计中存在的问题。仿真一般分为功能仿真和时序仿真。功能仿真检测的是逻辑电路的功能，忽略电路固有的延时特征，可以对系统进行快速评估。时序仿真要结合电路固有的延时特性来检测电路的功能，更符合电路的实际情况，但仿真难度大，耗时长。

Verilog 语言中有一部分语句和定义是只针对仿真的，不能综合成硬件电路。在此我们先介绍用于仿真、测试的预编译指令、系统任务和系统函数，再介绍仿真测试平台。

5.7.1 预编译指令

预编译指令是 Verilog 语言中一类特殊的语句，其作用是在模块编译之前，预先通知编译器进行一些“预处理”，处理的结果再和源代码一起进行编译。

预编译指令由反引号（`）开头，其作用范围是从定义开始到文件结束，或遇到新的预编译指令为止。主要的预编译指令有以下几种，其中`timescale 是仿真时间标度指令，已经在 5.5.2.2 小节进行了介绍，此处不再赘述。

```
`define, `undef
`ifdef, `else, `endif
`include
`default_nettype
`resetall
`timescale
```

5.7.1.1 `define 指令

`define 指令与 C 语言中的#define 相似，用于文本的替代，即用一个宏名代替一个字符串，其一般书写格式为：

`define　宏名　字符串

例如：

```
`define BUS_SIZE 16  //用 BUS_SIZE 替代 16
`define ADD a+b      //用 ADD 替代 a+b
```

在文件中引用已经定义过的宏名的时候，必须要加上反引号，如`ADD。在编译时，先把宏名和字符串做置换，不做语法检查。如果定义有错误，则在宏定义展开以后做编译时报错。

`undef 的作用是取消宏名，例如：

```
`define BUS_SIZE 16 //用 BUS_SIZE 替代 16
```

```
……
reg [`BUS_SIZE:1] Bus;
……
`undef BUS_SIZE  //此指令之后，BUS_SIZE 不再有效
```

5.7.1.2 `ifdef 指令

`ifdef、`else 和`endif 指令用于有条件的编译。一般情况下，源代码的所有行都要进行编译，但也有特殊情况，某些行在条件满足的情况才进行编译，否则就不进行编译，此时就可以使用`ifdef、`else 和`endif 指令，其一般书写格式为：

```
`ifdef  宏名
    语句段 1；
`else
    语句段 2；
`endif
```

例如：

```
`ifdef ADD_BUS    //如果定义了宏名 ADD_BUS
   pameter BUS_SIZE = 16; //则 BUS_SIZE 为 16
`else
   pameter BUS_SIZE = 0; //则 BUS_SIZE 为 0
`endif
```

其中，else 部分是可选的。

5.7.1.3 `include 指令

`include 是文件包含指令，其作用是在一个文件中嵌入另一个文件，一起进行编译，其一般书写格式为：

```
`include  “路径/文件名”
```

一个`include 只能包含一个文件，如果要包含多个文件，就要写多个`include 指令。在编译的时候，`include 指令会被文件里的内容替代。例如：

```
`include “source_code.v”
`include “D:/decode/primitive.v”
```

对上面第二个语句而言，在编译时，该指令会被“D:/decode/primitive.v”里的内容替代。

5.7.1.4 `default_nettype 指令

`default_nettype 指令的作用是设定默认的线网类型，例如：

```
`default_nettype wand
```

表示将默认的线网类型设置为线与类型。在此指令之后，所有默认的线网类型都是线与类型。

5.7.1.5 `resetall 指令

`resetall 指令的作用是将所有编译指令重新设置为默认值。例如：

```
`resetall
```

如果前面由`default_nettype 指令将默认的线网类型设置成线与类型，则执行`resetall 以后，默认的连线重新为线网类型。

5.7.2 系统任务和系统函数

Verilog 语言中提供了丰富的内置式系统任务和系统函数，主要用于仿真和测试。常用的系统任务和系统函数包括显示任务、文件输入/输出任务、仿真结束/暂停任务、仿真时间函数等。系统任务和系统函数都以“$”符号开头。

5.7.2.1 显示任务

在仿真时，显示任务用于显示信息和输出结果。常用的显示任务有：$display、$write、$strobe 和$monitor 等。

（1）$display

系统任务$display 的作用是根据显示格式要求显示字符串、数值等内容，它的一般语法格式为：

$display （“带格式的字符串”，参数 1，参数 2，…）；

其中带格式的字符串用双引号括起来，例如：

```
$display ("TEST IS FAILURE."); //字符串显示
$display ("A is %d, B is %h", A, B); //数值显示，设 A=20，B=115
```

运行以后显示的结果分别是：

```
TEST IS FAILURE.
A is 20, B is 73
```

在字符串表达式中，有些字符前面的“\”和“%”是转义符，例如“\t”表示 Tab，“\n”表示换行，“%b”表示二进制格式显示，“%d”表示十进制格式显示等。例如：

```
$display ("1\t%d\n2\t%d\\", A, B); //数值显示，设 A=20，B=115
```

运行以后显示的结果是：

```
1 20
2 115\
```

表 5-20 给出了常用转义符的含义，其中“%”后的字符可以是大写字母或者小写字母，含义都一样。

表 5-20 Verilog 常用转义符

转义符	含义	转义符	含义
\n	换行	%b 或%B	以二进制格式显示
\t	Tab	%o 或%O	以八进制格式显示
\\	字符\	%d 或%D	以十进制格式显示
%%	字符%	%h 或%H	以十六进制格式显示
%t 或%T	显示当前时间	%c 或%C	以 ASCII 码格式显示
%f 或%F	以十进制实数显示	%s 或%S	显示字符串

（2）$write

系统任务$write 的作用与$display 一样，也是在仿真器的命令行窗口按要求显示结果。它的一般语法格式为：

$write （“带格式的字符串”，参数 1，参数 2，…）；

$write 与$display 的区别是$display 在文本之后有一个换行符，而$write 没有换行符。例如：

```
$write ("Simulation time is "); //字符串显示，最后不换行
$write ("%t\n", $time); //当前仿真时间显示，假设为 20，$time 是模拟时间任务
```

显示结果为：

```
Simulation time is 20
```

（3）$strobe

系统任务$strobe 的作用与$display 类似，不同之处在于，$display 可以显示当前变量值，而$strobe 要等到当前时刻的所有事件都完成之后，才按要求显示结果。它的一般语法格式为：

$strobe （“带格式的字符串”，参数 1，参数 2，…）；

以下是一个$strobe 的例子：

```
integer i;
initial begin
  i = 1;
  $display ("After first assignment, i = %d", i);
  $strobe ("When strobe is executed, i = %d", i);
  i = 2;
  $display ("After second assignment, i = %d", i);
end
```

显示的结果为：

```
After first assignment, i = 1
When strobe is executed, i = 2
After second assignment, i = 2
```

$display 的显示结果比较容易理解，当 i 值改变以后，$display 显示结果就改变。而$strobe 是在该时刻所有的事件都执行完了以后才显示结果，此时，i 的值已经为 2 了。

（4）$monitor

$monitor 是一个监控任务，其作用是对多个参数进行监控。只要参数表中有一个参数发生变化，$monitor 就启动，显示格式化字符串。它的一般语法格式

$monitor （“带格式的字符串”，参数 1，参数 2，…）；

在仿真过程中，只允许一个$monitor 任务执行。如果有多个$monitor 任务，则只有最后一个有效。例如：

```
initial begin
  a = 0; b = 0;
  $monitor ("\$monitor: a = %d", a);
  a = 1;
  $display ("\$display: a = %d", a);
  a = 2;
  $monitor ("\$monitor: b = %d", b);
  a = 3;
  $display ("\$display, a = %d", a);
end
```

显示的结果为：

```
$display: a = 1
$display: a = 3
$monitor: b = 0
```

另外，可以使用系统任务$monitoroff 和$monitoron 来关闭所有监控任务和打开所有监控任务，使得监控任务更加灵活方便。

5.7.2.2 文件输入/输出任务

在一些大型设计仿真时，常常要将文件中的数据读出或将数据写入文件，因此要用到对文件进行操作的系统任务和系统函数。在 Verilog 中，常用的文件输入/输出系统任务和系统函数有以下一些：

文件句柄 = $fopen （“文件路径/文件名”）；

$fdisplay （文件句柄，“带格式字符串”，参数 1，参数 2，…）；

$fstrobe （文件句柄，“带格式的字符串”，参数 1，参数 2，…）；

$fmonitor （文件句柄，“带格式的字符串”，参数 1，参数 2，…）；

$fwrite （文件句柄，“带格式的字符串”，参数 1，参数 2，…）；

$fclose（文件句柄）；//关闭文件

$feof（文件句柄）； //查询是否到文件末尾

系统函数$fopen 用于打开一个文件，其返回值是文件句柄，即关于文件的一个整数或指针。$fclose 用于关闭文件。例如：

```
integer file_pt;
initial  begin
  file_pt = $fopen (“div.tq”);
  ……
  $fdisplay (file_pt,”The simulation time is %t”, $time);
  ……
  $fclose (file_pt);
end
```

前面介绍了$display、$write、$strobe 和$monitor 等几种显示任务，这些显示任务都有相应的用于文件的变种，即$fdisplay、$fwrite、$fstrobe 和$fmonitor 等。这些用于文件的系统任务可以将信息写入到文件中，其第一个参数都是文件句柄。在以上的例子中，当$fdisplay 执行完以后，文件中出现如下信息：

```
The simulation time is       0
```

另外，$readmemb 和$readmemh 是对文件进行读取的两个系统任务，它们可以将数据从文件中读出来，并加载到存储器中去。其语句格式为：

$readmemb （“文件名”，存储器名，起始地址，终止地址）；

$readmemh （“文件名”，存储器名，起始地址，终止地址）；

其中“文件名”和“存储器名”必须有，“起始地址”和“终止地址”是可选项，用十进制表示。文件中只能包括空白（包括空格、换行）、注释和二进制数据（针对$readmemb）或十六进制数据（针对$readmemh）。例如：

```
reg [0:3] MEMA[0:31];//定义一个 32 个单元的存储器 MEMA，每个单元位宽为 4
……
initial
  $readmemb (”example.dat”, MEMA);//从文件中读出的数据装载到 MEMA 每个单元中，
                                  //起始单元是存储器定义范围左边的值；
```

若写成：

```
  $readmemb (”example.dat”, MEMA, 16, 31);
```

表示读出的数据装载到存储器的起始地址是 16，终止地址是 31。如果数据文件的数据多于存储器的单元数，则数据存入到该存储器的终止地址为止。反之，未赋值的存储器单元内的值默认为 x。

5.7.2.3 仿真结束/暂停任务

常用的仿真控制任务有$finish 和$stop，语法格式为：

$finish；

$stop；

$finish 的作用是结束仿真程序，返回操作系统。$stop 是暂停仿真程序，进入一种交互模式，将控制权交给用户。

5.7.2.4 仿真时间函数

仿真时间函数有如下几种类型：

$time　//返回 64 位整型仿真时间

$stime　//返回 32 位整型仿真时间

$realtime //返回实数型仿真时间

5.7.3 Verilog 测试平台

Verilog 测试平台（Test Bench）是指用来测试一个 Verilog 实体的程序，也是用 Verilog 的基本语法要素来编写。该测试平台可以产生激励信号，通过模块例化语句把激励信号传递给被测试的 Verilog 实体，并可以存储或观察被测试的 Verilog 实体的输出信号。将测试输出信号与期望值相比较，就可以判断被测试的 Verilog 实体的内部结构是否正确。

例 5-50 是例 5-24 的半加器的测试平台，我们以此来说明测试平台的编写方法。为方便阅读，将半加器程序重新写在测试平台之后。

【例 5-50】半加器测试平台

```
`timescale 10ns/1ns  //仿真时间标度语句必须存在
module H_Adder_tb;   //仿真测试平台没有端口
  reg a, b;          //测试输入信号，定义为 reg 型
  wire r, c;         //测试输出信号，定义为 wire 型
  //采用 initial 语句对测试输入信号赋值，覆盖输入信号的各种组合
  initial begin
      a = 0; b = 0;
      #10 begin a = 0; b = 1; end
      #10 begin a = 1; b = 0; end
      #10 begin a = 1; b = 1; end
      #10 begin a = 0; b = 0; end
      #10 $stop;
      end
  //采用模块例化语句调用半加器，把激励信号传递给半加器
  H_Adder U1 (.A(a), .B(b), .R(r), .C(c));
endmodule

module H_Adder(A, B, R, C); //半加器程序
  input A, B;
  output R, C;
  assign R = A ^ B;
  assign C = A & B;
endmodule
```

在写测试平台时，首先要给出仿真时间标度语句，该语句属于预编译指令，已经在 5.5.2.2 小节进行了介绍，此处不再赘述。本例中仿真的时间单位是 10ns，时间精度是 1ns。仿真时间标度语句在 module 之外，因此它对以下所有的 module 都有效。

测试平台本身也是一个模块，因此要有模块名，本例中的模块名是 H_Adder_tb，但是后面没有端口名列表，表明这个模块没有对外的输入、输出通道，只有内部信号。

H_Adder_tb 要对半加器 H_Adder 进行验证，因此其内部设置的信号要跟 H_Adder 的端口对应。H_Adder 有两个输入端口 A、B 和两个输出端口 R、C，那么 H_Adder_tb 内部设置的对应信号分别为 a、b、r、c（也可以用相同信号名 A、B、R、C）。要注意的是，对应于输入端口的 a、b 要定义为 reg 型，对应于输出端口的 r、c 要定义为 wire 型。

H_Adder_tb 中有两个并行语句，一个是 initial 语句，一个是模块例化语句，它们的书写顺序可以调换。模块例化语句是将被测试的半加器作为元件放入到测试平台中，initial 语句是对半加器的输入端加入激励信号。也可以用两个 initial 语句分别对 a、b 加入激励信号。initial 语句在 5.5.2.2 小节进行了介绍，此处不再赘述。图 5-33 是例 5-50 测试平台的结构框图。

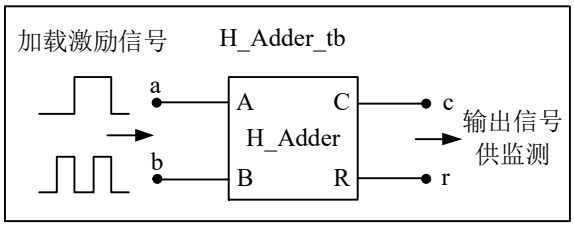


图 5-33 半加器测试平台结构框图

例 5-51 是例 5-8 的十六进制计数器的测试平台。为方便阅读，将十六进制计数器程序重新写在测试平台之后。

【例 5-51】十六进制计数器测试平台

```
`timescale 10ns/1ns
module COUNTER16_tb;
  reg clk, rst, oe;     //测试输入信号，定义为 reg 型
  wire [3:0] cout;      //测试输出信号，定义为 wire 型
  COUNTER16_1 U1 (clk, rst, oe, cout); //模块例化语句，位置关联
  //采用 initial 语句和 always 语句对时钟信号赋值
  initial clk = 0; //时钟信号初始设为 0
  always #1 clk = ~clk; //产生周期为 20ns 的时钟信号
  //采用 initial 语句对测试输入信号 rst 赋值
  initial begin
      #0 rst = 0; #5 rst = 1; #15 rst = 0; #4 rst = 1;
      #60 $stop;
      end
  //采用 initial 语句对测试输入信号 oe 赋值
  initial begin
      #0 oe = 1; #28 oe = 0; #6 oe = 1;
      #70 $stop;
      end
endmodule

module COUNTER16_1(CLK, RST, OE, COUT); //十六进制计数器
  input CLK, OE, RST;
```

```
    output  [3:0] COUT;
    reg [3:0] COUT;
    reg [3:0] CT;

    always @(posedge CLK or negedge RST)
    begin
        if (!RST)
            CT <= 4'b0000;
        else
            CT <= CT + 1;
    end
    always @(OE)
        COUT = OE ? CT: 4'bzzzz;
endmodule
```

在测试平台中编写激励信号时应尽量全面地覆盖输入信号的各种组合情况，保证对被测模块的有效验证。激励信号的编写方式有以下几种：

1．周期信号

周期信号，如时钟信号，可以采用 initial 语句和 always 语句结合的方式产生，如例 5-51 中所示：

```
initial clk = 0;
always #1 clk = ~clk;
```

另外，也可以利用 forever 语句产生周期性激励信号，例如：

```
initial begin
clk = 0;
forever #1 clk = ~clk;
end
```

对于占空比非 1:1 的周期性信号，可以采用 always 语句产生。下面的语句产生的信号为高电平持续 2 个时间单位，低电平持续 3 个时间单位，如图 5-34 所示，图中时间单位设为 10ns。

```
always begin
  #2 clk = 0;
  #3 clk = 1;
end
```

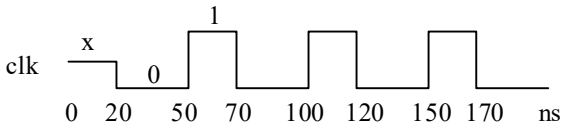


图 5-34 占空比非 1:1 的周期信号

2．非周期信号

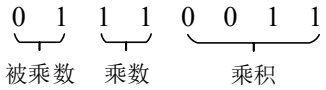
当非周期信号数据量较少时，可以采用 initial 语句进行穷举，产生激励信号，如例 5-50 所示：

```
initial begin
  a = 0; b = 0;
  #10 begin a = 0; b = 1; end
  #10 begin a = 1; b = 0; end
  #10 begin a = 1; b = 1; end
```

```
#10 begin a = 0; b = 0; end
#10 $stop;
end
```

当激励信号数据较多时，无法手工输入，则可以将激励信号存放在一个文件中，调用系统函数读入数据。也可以将输出数据存放到一个文件中，以备后续分析。

例 5-52 是对例 5-36 两位乘法器的仿真测试平台。这里把输入激励信号和预期运算结果放在一个数据文件 test.dat 中，每个数据是 8 位，格式为：



仿真时，将文件 test.dat 中的数据读入到一个存储器中，然后把存储器中的被乘数和乘数逐个传送给乘法器，由乘法器计算结果，得到计算值 r，并与期望值 r_EX 作对比，显示对比结果。

【例 5-52】两位乘法器测试平台

```
`timescale 1ns/100ps
module MUL2_tb;
  reg [1:0] m, n;     //测试输入信号，定义为 reg 型
  wire [3:0] r;      //测试输出信号，定义为 wire 型
  reg [1:8] mymem [1:16];  //定义一个 8 位存储器，16 个单元
  reg [3:0] r_EX;
  integer J;

  MUL2_2  U1(m, n, r); //模块例化语句，位置关联
  initial begin
      $readmemb ("test.dat", mymem);
      for (J=1; J<=16; J=J+1)
          begin
              {m, n, r_EX} = mymem[J];
              #5; //延迟 5 个时间单位，等待电路稳定

              if (r != = r_EX)
                  $display ("Mismatch on vector %b", mymem[J]);
              else
                  $display ("No mismatch on vector %b", mymem[J]);
          end
      end
endmodule
```

仿真输出数据也可以写入到一个文件中，便于后续分析研究。例如：

```
integer file_dt;
initial  begin
  file_dt = $fopen ("simdata.dat"); //打开一个数据文件
  ……
  $fwrite (file_dt, "%b/n", dataout); //将仿真数据写入文件
  ……
  $fclose (file_dt);
end
```

5.8 Verilog HDL 模板

以下我们列出 Verilog HDL 的主要语句格式，以便使用时查询。在实际使用时，可以根据需要给模块、端口、信号、变量等取名字，确定端口和各种参数的数量。注意模板中的各个部分并不是每个程序都需要的。

```
module  module_name  （port_name1, port_name2, …）;
    // 端口说明
    input  port_name1，port_name2，…;
    input  [msb: lsb]  port_name3，port_name4，…;
    output  port_name5，port_name6，…;
    output  [msb: lsb]  port_name7，port_name8，…;
    inout  port_name9，port_name10，…;
    inout  [msb: lsb]  port_name11，port_name12，…;

    // 数据类型、参数、任务、函数等说明
    wire  port_name5，…;
    wire  [msb: lsb]  port_name7，…;
    reg  port_name6，…;
    reg  [msb: lsb]  port_name8，…;
    parameter  identifier  =  expression；…
    integer  index;

    task  task_name;
        input  task_port_name1，…;
        input  [msb: lsb] task_port_name2，…;
        output  task_port_name3，…;
        output  [msb: lsb]  task_port_name4，…;
        inout  port_name5，…;
        inout  [msb: lsb]  port_name6，…;
        reg  local_reg_name;
        integer  identifier;
        begin
            procedural_statement
            (always, if, case, for, …)
        end
    endtask

    function  [msb: lsb]  function_name
        input  func_port_name1，…;
        input  [msb: lsb]  func_port_name2，…;
        reg  local_ reg_name;
        integer  identifier;
        begin
            procedural_statement
```

```
            (always, if, case, for, …)
        end
endfunction

// 以下为电路描述语句
// 连续赋值语句
assign   [delay]   target_name = expression;

// always 语句 1
always @(sensitibity list)
        begin
          case   (expression)
            item1:   begin procedural_statement end;
            item2:   begin procedural_statement end;
            ……
            default: begin procedural_statement end;
          endcase

          for (index=value1; index<=MAX ; index=index+1)
            begin
                procedural_statement;
            end
        end

// always 语句 2
always @(posedge CLOCK or negedge RESET)
        begin
          if   (expression_1)
            begin
                procedural_statement;
            end
          else if   (expression_2)
            begin
                procedural_statement;
            end
            …….
          else if   (expression_n)
            begin
                procedural_statement;
            end
          else
            begin
                procedural_statement;
            end
```

```
        while  (expression)
        begin
            procedural_statement;
        end
    end

// initial 语句 1
    initial
        begin
            identifier = value1;
            # delay1  identifier = value2;
            # delay2  identifier = value3;
            ……
        end

// initial 语句 2
    initial
        begin
            procedural_statement;
        end

// 模块例化语句
    module_name  instance_name  (port_associations);
    ( examples:
    H_Adder  U1 (.A(a),  .B(b),  .R(r),  .C(c));
    MULTIPLER4  #(.W(8))  L1 (.X(X1),  .Y(Y1),  .R(R1));
    )

// 基本库元件例化语句
    and  U1  (dout, din1, din2);          nand  U2  (dout, din1, din2);
    or  U3  (dout, din1, din2, din3);     xor  (dout, din1, din2);
    nor  U4  (dout, din1, din2, din3);    xnor  (dout, din1, din2);
    buf  B1  (dout1, dout2, dout3, din);  not  N1  (dout1, dout2, din);
    notif0  NT0  (outA2, inB2, inCTRL2);  notif1  NT1  (outA3, inB3, inCTRL3);
    bufif0  BF0  (outA0, inB0, inCTRL0);
    bufif1  BF1  (outA1, inB1, inCTRL1);
endmodule
```

附录

IEEE Std 1364-2001 标准中的 Verilog 关键字：

always	and	assign	automatic		
begin	buf	bufif0	bufif1		
case	casex	casez	cell	cmos	config
deassign	default	defparam	disable		
edge	else	end	endcase	endconfig	endfunction
endgenerate	endmodule	endprimitive	endspecify	endtable	endtask
event					
for	force	forever	fork	function	
highz0	highz1				
if	ifnone	ifcdir	include	initial	inout
input	instance	integer			
join					
large	liblist	library	localparam		
macromodule	medium	module			
nand	negedge	nmos	nor	not	notif0
notif1	noshowcancelled				
or	output				
parameter	pmos	posedge	primitive	pull0	pull1
pullup	pulldown	pulsestyle_onevent	pulsestyle_ondetect		
rcmos	real	realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0	rtranif1	
scalared	showcancelled	signed	small	specify	specparam
strong0	strong1	supply0	supply1		
table	task	time	tran	tranif0	tranif1
tri	tri0	tri1	triand	trior	trireg
unsigned	use	uwire			
vectored					
wait	wand	weak0	weak1	while	wire
wor					
xnor	xor				

习题

1. 描述 Verilog 标识符的自定义原则，并指出下列标识符是否合法：

   3ABC，$ABC，REG#，reg，\3ABC\，WIRE，\A3BC，ABC_3，_A#B，A-B

2. 下列表达式的二进制形式是什么？

   5'd23，8'o55，'B110x1，3'h12，'Hef，16'h95fc，4'hz，6'b101，4'o28，-6'd18

3. 下列实数书写格式是否正确？

   3.26，5E-2，6，_78_000.37，48.92e2

4. Verilog 模块的端口类型有哪几种？说明每种端口的数据流向。

5. 简述 wire 类型和 reg 类型的区别。

6. 完成以下定义。

   定义一个宽度为 8 的 wire 类型地址总线 ADD_BUS；

   定义两个单比特 reg 类型的变量 AT、BT；

   定义一个名称为 idex 的整数；

   定义参数 D0=6'b100110，D1=5'd17；

   定义一个位宽为 8、容量为 128 的存储器 MEM128；

7. 指出下列表达的错误之处。

   （1）integer　[3:0]　J;

   （2）reg mymem [1:16];

   ……

   mymem <= 16'h369C;

   （3）integer　STUDENT;

   reg　[8:1]　NAME;

   ……

   NAME <= STUDENT[17:10];

8. 设 A=4'b1010，B=4'b0011，C=2'b00，D=8'h71，则下面各表达式的值是多少？

（1）A&B　（2）C|D　（3）A&&D　（4）^D　（5）!C

（6）A<B　（7）B+D　（8）B>>2　（9）{A，C}　（10）{2{D,C}}

9. Verilog HDL 中有哪些主要的并行语句？它们在程序中的书写顺序是可以调换的吗？

10. always 语句的敏感信号列表的作用是什么？

11. 画出下列 initial 语句产生的信号波形，设时间单位为 10ns。

```
initial
   begin
   A = 0; B = 0;
   # 2  A = 1; B = 1;
   # 3  A = 0;
   # 4  A = 1; B = 0;
   # 5  $finish
   end
```

12. 分别利用 if 和 case 语句实现 16-4 编码器。

13. 定义一个位宽为 8，容量为 64 的存储器，并采用 for 语句将该存储器所有单元都初始化为 8'hFF。

14. 阻塞式赋值和非阻塞式赋值有什么区别？

15. 定义一个完成奇、偶校验的任务，该任务有一个 8 位输入数据端口 datain，一个选择奇校验还是偶校验的选择输入端 sel（1 表示奇校验，0 表示偶校验），还有一个结果输出端 result。

16. 定义一个函数，实现数字 0~9 到其 ASCII 码的转换。

17. 下表为格雷码的编码方式，利用状态机实现输出为 4 位格雷码的计数器。

计数值	0	1	2	3	4	5	6	7
格雷码	0000	0001	0011	0010	0110	0111	0101	0100
计数值	8	9	10	11	12	13	14	15
格雷码	1100	1101	1111	1110	1010	1011	1001	1000

18. 图 1 所示为一个分频器，clk_in 为 10MHz 的输入时钟信号，reset 为异步复位信号，clk_out 需输出 1Hz 的时钟信号。当 reset 为低电平时，分频器复位，输出保持低电平；当 reset 为高电平时，分频器正常工作。编写实现该分频器的 Verilog 程序。

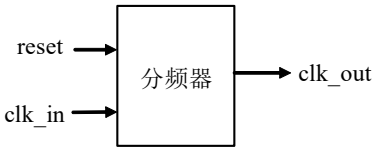


图 1

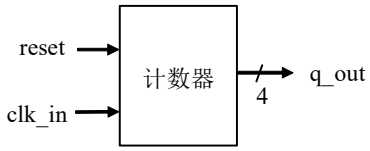


图 2

19. 设计一个十进制计数器，外围端口如图 2 所示。clk_in 为输入时钟信号，reset 为异步复位信号，q_out 为输出结果。当 reset 为低电平时，计数器复位，输出为 4'b0000；当 reset 为高电平时，计数器正常计数。编写实现该计数器的 Verilog 程序。

20. 图 3 是一个十进制计数器，其输出直接驱动七段数码管，由数码管显示计数值。以例 5-31 七段数码管译码器和第 19 题的计数器作为两个元件，采用模块例化方式设计该计数器模块。

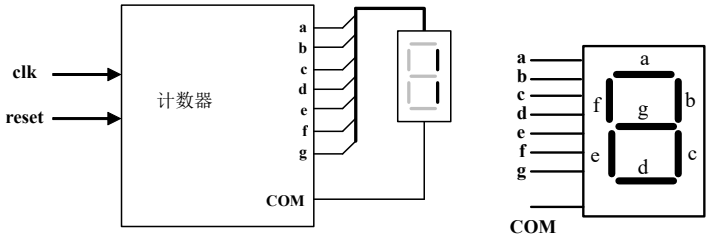


图 3