第5章 Verilog语句语法

Verilog HDL支持许多行为语句，使其成为结构化和行为性的语言，这些行为语句包括过程语句、块语句、赋值语句、条件语句、循环语句、编译指示语句等，如表5.1所示。

表5.1　Verilog HDL的行为语句

类　别	语　句	可　综　合
过程语句	initial	
	always	√
块语句	串行块 begin-end	√
	并行块 fork-join	
赋值语句	持续赋值 assign	√
	过程赋值=、<=	√
条件语句	if-else	√
	case	√
循环语句	for	√
	repeat	
	while	
	forever	
编译指示语句	`define	√
	`include	
	`ifdef，`else，`endif	√

几乎所有的HDL语句都可用于仿真，但可综合的语句通常只是HDL语句的一个核心子集，不同综合器支持的HDL语句集通常有所不同。学习行为语句时，应该对语句的可综合性有所了解。目前，可综合的Verilog子集也在向标准化发展，已经推出的IEEE Std 1364[2].1—2002标准为Verilog语言的RTL级综合定义了一系列的建模准则。

编写HDL程序，就是在描述一个电路，每一段程序都对应着相应的硬件电路结构，应深入理解两者的关系。综合器可将HDL文本对应的硬件电路以图形的方式呈现出来，便于学习者建立HDL程序与硬件电路之间的对应关系。

5.1　过程语句

Verilog语言中的多数过程模块都从属于always和initial两种过程语句。

在一个模块(module)中，使用always和initial语句的次数是不受限制的。always块内的语句是不断重复执行的；always过程语句是可综合的，在可综合的电路设计中广泛采用。initial语句常用于仿真中的初始化；initial过程块中的语句只执行一次。

5.1.1　always过程语句

always过程语句使用模板如下：

```
always @(<敏感信号列表 sensitivity list>)
begin
    //过程赋值
    //if-else,case,casex,casez 选择语句
    //while,repeat,for 循环
    //task,function 调用
end
```

always 过程语句通常带有触发条件。触发条件写在敏感信号表达式中，只有当触发条件满足时，其后的 begin-end 块语句才能被执行。因此，此处首先讲解敏感信号列表(Sensitivity List)的含义，以及如何写敏感信号表达式。

1. 敏感信号列表

敏感信号列表，又称事件表达式或敏感信号表达式，即当该列表中变量的值改变时，就会引发块内语句的执行。因此，敏感信号列表中应列出影响块内取值的所有信号。有两个或两个以上信号时，它们之间用 or 连接。例如：

```
@(a)                                    //当信号 a 的值发生改变
@(a or b)                               //当信号 a 或信号 b 的值发生改变
@(posedge clock)                        //当 clock 的上升沿到来时
@(negedge clock)                        //当 clock 的下降沿到来时
@(posedge clk or negedge reset)         //当 clk 的上升沿或 reset 信号的下降沿到来时
```

如例 5.1 中用 case 语句描述的 4 选 1 数据选择器，只要输入信号 in0、in1、in2、in3，或选择信号 sel 中的任一个发生改变，输出就会改变，所以敏感信号列表写为

```
@ (in0 or in1 or in2 or in3 or sel)
```

【例 5.1】 用 case 语句描述的 4 选 1 数据选择器。

```
module mux4_1
            (input in0,in1,in2,in3,
            input[1:0] sel, output reg out);
always @(in0 or in1 or in2 or in3 or sel)              //敏感信号列表
   case(sel)
   2'b00:   out = in0;
   2'b01:   out = in1;
   2'b10:   out = in2;
   2'b11:   out = in3;
   default:out = 2'bx;
   endcase
endmodule
```

敏感信号分为边沿敏感型和电平敏感型两种。每个 always 过程最好只由一种类型的敏感信号来触发，避免将边沿敏感型和电平敏感型信号列在一起。例如下面的例子：

```
always @(posedge clk or posedge clr)
      //两个敏感信号都是边沿敏感型
always @(A or B)
      //两个敏感信号都是电平敏感型
always @(posedge clk or clr)
      //不建议这样用,不宜将边沿敏感型和电平敏感型信号列在一起
```

2. posedge 与 negedge 关键字

对于时序电路,事件通常是由时钟边沿触发的。为表达边沿这个概念,Verilog HDL提供了 posedge 和 negedge 两个关键字来描述。

【例 5.2】 同步置数、同步清零的计数器。

```
module count                                  //模块声明采用 Verilog-2001 格式
          (input load,clk,reset,
           input[7:0] data,
           output reg[7:0] out);
always @ (posedge clk)                        //clk 上升沿触发
   begin
     if(!reset)       out<=8'h00;             //同步清零,低电平有效
     else if(load)    out<=data;              //同步预置
     else             out<=out+1;             //计数
   end
endmodule
```

在例 5.2 中,posedge clk 表示时钟信号 clk 的上升沿作为触发条件,而 negedge clk 表示时钟信号 clk 的下降沿作为触发条件。

在例 5.2 中,没有将 load、reset 信号列入敏感信号列表,因此属于同步置数、同步清零,这两个信号要起作用,必须有时钟的上升沿到来。对于异步的清零/置数,如时钟信号为 clk,clr 为异步清零信号,则敏感信号列表应写为

```
always @(posedge clk or posedge clr)
                  //clr 信号上升沿到来时清零,故高电平清零有效
always @(posedge clk or negedge clr)
                  //clr 信号下降沿到来时清零,故低电平清零有效
```

若有其他异步控制信号,可按此方式加入。

注意:块内的逻辑描述要与敏感信号列表中信号的有效电平一致。

例如,下面的描述是错误的。

```
always @(posedge clk or negedge clr) //低电平清零有效
begin
   if(clr) out<=0;                  //与敏感信号列表中低电平清零有效矛盾,应改为 if(!clr)
   else out<=in;
end
```

3. Verilog-2001 标准对敏感信号列表的新规定

Verilog-2001 标准对敏感信号列表做了新的规定。

(1) 敏感信号列表中可用逗号分隔敏感信号。

在 Verilog-2001 中,可用逗号分隔敏感信号。例如:

```
always @(a or b or cin)
always @(posedge clk or negedge clr)
```

上面的语句按照 Verilog-2001 标准可写为下面的形式。

```
always @(a,b,cin)                          //用逗号分隔信号
always @(posedge clk,negedge clr)
```

(2) 在敏感信号列表中使用通配符 * 。

用 always 过程块描述组合逻辑时,应在敏感信号列表中列出所有的输入信号,在 Verilog-2001 中,可用通配符 * 来表示包括该过程块中的所有信号变量。

例如,在 Verilog-1995 中,一般这样写敏感信号列表:

```
always @(a or b or cin)
     {cout,sum} = a + b + cin;
```

上面的敏感信号列表在 Verilog-2001 中可表示为如下两种形式,这两种形式是等价的。

```
always @ *                                 //形式 1
     {cout,sum} = a + b + cin;
always @( * )                              //形式 2
     {cout,sum} = a + b + cin;
```

4. 用 always 过程块实现较复杂的组合逻辑电路

always 过程语句通常用来对寄存器类型的数据进行赋值,但 always 过程语句也可以用来设计组合逻辑。在有些情况下,使用 assign 实现组合逻辑电路会显得冗长且效率低下,而适当采用 always 过程语句来实现,能收到更好的效果。

例 5.3 是一个指令译码电路的例子。该例通过指令判断对输入数据执行相应的操作,包括加、减、求与、求或、求反。这是一个较为复杂的组合逻辑电路,如果采用 assign 语句描述,表达起来非常复杂。在本例中使用了电平敏感的 always 块,并采用 case 结构来进行分支判断,不但设计思想得到直观的体现,而且代码看起来整齐有序。

【例 5.3】 用 always 过程语句描述的简单算术逻辑单元。

```
`define add     3'd0
`define minus   3'd1
`define band    3'd2
`define bor     3'd3
`define bnot    3'd4
module alu(out,opcode,a,b);
input[2:0] opcode;                    //操作码
input[7:0] a,b;                       //操作数
output reg[7:0] out;
always@ *                             //或写为 always@( * )
begin   case(opcode)
     `add:    out = a + b;            //加操作
     `minus: out = a - b;             //减操作
     `band:    out = a&b;             //按位与
     `bor:    out = a|b;              //按位或
     `bnot:    out = ~a;              //按位取反
     default:out = 8'hx;              //未收到指令时,输出任意态
     endcase
end
endmodule
```

图 5.1 是例 5.3 的 RTL 综合结果，是由加法器、门电路、数据选择器等模块构成的。

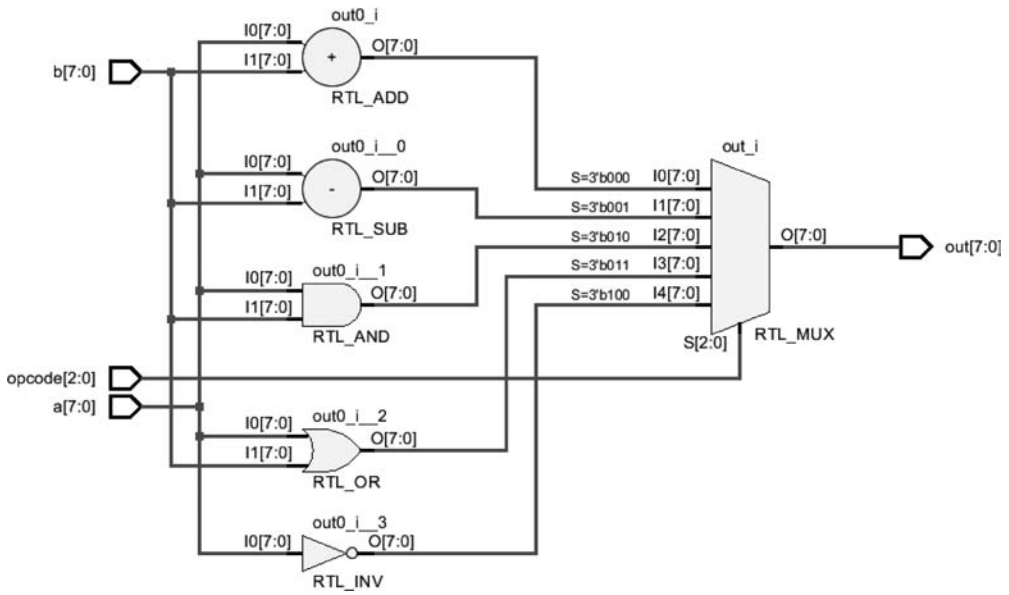


图 5.1 例 5.3 的 RTL 综合结果

5.1.2 initial 过程语句

initial 语句的使用格式如下：

```
initial
  begin
     语句 1;
     语句 2;
     …
  end
```

initial 语句不带触发条件，initial 过程中的块语句沿时间轴只执行一次。initial 语句通常用于仿真模块中对激励向量的描述，或用于给寄存器变量赋初值，它是面向模拟仿真的过程语句，通常不能被逻辑综合工具支持。

下面举例说明 initial 语句的使用方法。如例 5.4 的测试模块中利用 initial 语句完成对测试变量 a、b、c 的赋值。

【例 5.4】 用 initial 过程语句对测试变量赋值。

```
`timescale 1ns/1ns
module test;
reg a,b,c;
initial  begin     a = 0;b = 1;c = 0;
             #50   a = 1;b = 0;
             #50   a = 0;c = 1;
             #50   b = 1;
             #50   b = 0;c = 0;
             #50 $finish;   end
endmodule
```

例 5.4 对 a、b、c 的赋值相当于描述了如图 5.2 所示的波形。

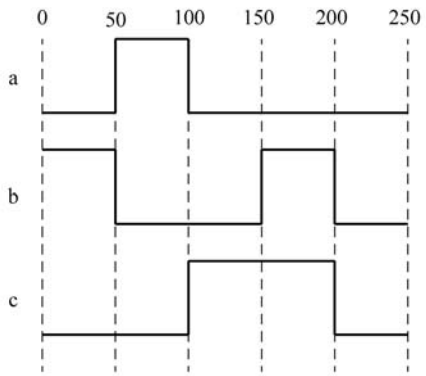


图 5.2 例 5.4 所定义的波形

下面的代码用 initial 语句对 memory 存储器进行初始化，将其所有存储单元的初始值都置为 0。

```
initial
  begin
    for(addr = 0;addr < size;addr = addr + 1)
    memory[addr] = 0;                          //对 memory 存储器进行初始化
  end
```

5.2 块语句

块语句是由块标识符 begin-end 或 fork-join 界定的一组语句，当块语句只包含一条语句时，块标识符可以缺省。下面分别介绍串行块 begin-end 和并行块 fork-join。

5.2.1 串行块 begin-end

begin-end 串行块中的语句按串行方式顺序执行。例如：

```
begin
  regb = rega;
  regc = regb;
end
```

由于 begin-end 块内的语句顺序执行，最后将 regb、regc 的值都更新为 rega 的值，该 begin-end 块执行完后，regb、regc 的值是相同的。

在仿真时，begin-end 块中的每条语句前面的延时都是相对于前一条语句执行结束的相对时间。如例 5.5，模块产生了一段周期为 10 个时间单位的信号波形。

【例 5.5】 用 begin-end 串行块产生信号波形。

```
`timescale 10ns/1ns
module wave1;
parameter CYCLE = 10;
reg wave;
initial
  begin                wave = 0;
    #(CYCLE/2)         wave = 1;
    #(CYCLE/2)         wave = 0;
    #(CYCLE/2)         wave = 1;
    #(CYCLE/2)         wave = 0;
    #(CYCLE/2)         wave = 1;
    #(CYCLE/2)          $stop;
end
initial  $monitor( $time,,,"wave = %b",wave);
endmodule
```

上面的程序用 ModelSim 编译仿真后，可得到一段周期为 10 个时间单位(100ns)的信号波形，如图 5.3 所示。

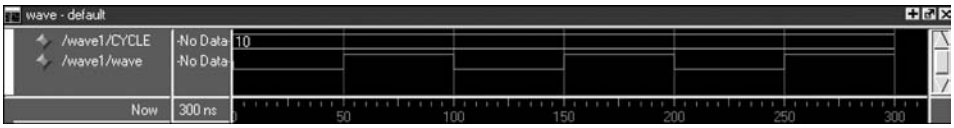


图 5.3　例 5.5 所描述的波形

5.2.2　并行块 fork-join

并行块 fork-join 中的所有语句是并发执行的。例如：

```
fork
regb = rega;
regc = regb;
join
```

由于 fork-join 并行块中的语句是同时执行的，在上面的块语句执行完后，regb 更新为 rega 的值，而 regc 的值更新为改变之前的 regb 的值，故执行完后，regb 与 regc 的值是不同的。

在进行仿真时，fork-join 并行块中的每条语句前面的时延都是相对于该并行块的起始执行时间的。如要用 fork-join 并行块产生一段与例 5.5 相同的信号波形，应该像例 5.6 这样标注时延。

【例 5.6】　用 fork-join 并行块产生信号波形。

```
`timescale 10ns/1ns
module wave2;
parameter CYCLE = 5;
reg wave;
initial
  fork              wave = 0;
    #(CYCLE)        wave = 1;
    #(2 * CYCLE)    wave = 0;
    #(3 * CYCLE)    wave = 1;
    #(4 * CYCLE)    wave = 0;
    #(5 * CYCLE)    wave = 1;
    #(6 * CYCLE)    $stop;
join
initial $monitor($time,,,"wave = %b",wave);
endmodule
```

上面的程序用 ModelSim 编译仿真后，可得到与图 5.3 相同的信号波形。将例 5.5

和例 5.6 进行对比，可体会 begin-end 串行块和 fork-join 并行块的区别。

5.3 赋值语句

5.3.1 持续赋值与过程赋值

Verilog 语言有持续赋值与过程赋值两种赋值方式。

1. 持续赋值语句

assign 为持续赋值语句(Continuous Assignment)，主要用于对 wire 型变量的赋值。例如：

```
assign c = a&b;
```

在上面的赋值中，a、b、c 皆为 wire 型变量，a 和 b 信号的任何变化，都将随时反映到 c 上来。例 5.7 是用持续赋值方式定义的 2 选 1 多路选择器。

【例 5.7】 用持续赋值方式定义 2 选 1 多路选择器。

```
module mux2_1                          //模块声明采用 Verilog - 2001 格式
            (input a,b,sel,
             output out);
assign out = (sel == 0)?a:b;           //持续赋值，如果 sel 为 0，则 out = a; 否则 out = b
endmodule
```

例 5.8 采用 assign 语句描述了一个基本 RS 触发器，图 5.4 是其综合结果。

【例 5.8】 基本 RS 触发器。

```
module rs_ff
          (input r,s,
           output q,qn);
assign qn = ~(r & q);
assign  q = ~(s & qn);
endmodule
```

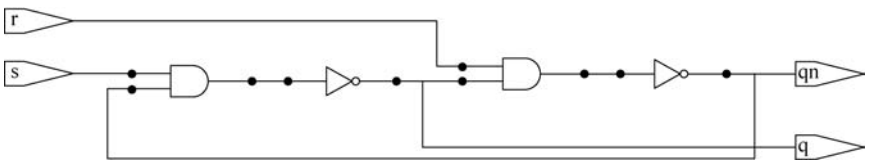


图 5.4 基本 RS 触发器综合结果

例 5.9 采用持续赋值语句实现了对 8 位带符号二进制数的求补码运算；图 5.5 是该例的综合结果，采用的是按位取反再加 1 的实现方法。

【例 5.9】 用持续赋值语句实现对 8 位带符号二进制数的求补码运算。

```
module buma
          ( input[7:0] ain,                    //8 位二进制数
            output[7:0] yout);                 //补码输出信号
assign yout = ~ain + 1;                        //求补
endmodule
```

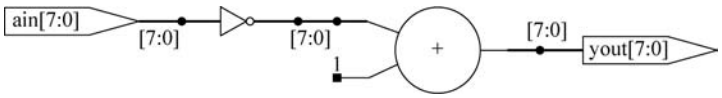


图 5.5 求补电路综合结果

2. 过程赋值语句

过程赋值语句(Procedural Assignment)多用于对 reg 型变量进行赋值。过程赋值有非阻塞赋值和阻塞赋值两种方式。

(1) 非阻塞(non_blocking)赋值方式：赋值符号为“<=”。例如：

```
b <= a;
```

非阻塞赋值在整个过程块结束时才完成赋值操作，即 b 的值并不是立刻改变的。

(2) 阻塞(blocking)赋值方式：赋值符号为“=”。例如：

```
b = a;
```

阻塞赋值在该语句结束时就立即完成赋值操作，即 b 的值在该条语句结束后立刻改变。如果一个块语句中有多条阻塞赋值语句，那么在前面的赋值语句完成之前，后面的语句不能被执行，仿佛被阻塞了(blocking)一样，因此称为阻塞赋值方式。例 5.10 是用阻塞赋值方式定义的 2 选 1 多路选择器。

【例 5.10】 用阻塞赋值方式定义 2 选 1 多路选择器。

```
module mux2_1_block
                    (input a,b,sel,
                     output reg out);
always @ *
   begin  if(sel == 0) out = a;
          else out = b;   end
endmodule
```

5.3.2 阻塞赋值与非阻塞赋值

阻塞赋值方式和非阻塞赋值方式的区别常给设计人员带来问题。为弄清非阻塞赋

值与阻塞赋值的区别，可见例 5.11。

【例 5.11】 非阻塞赋值与阻塞赋值。

```
//非阻塞赋值模块
module non_block
     (input clk,a,
      output reg c,b);
always @(posedge clk)
  begin
  b <= a;
  c <= b;
  end
endmodule
```

```
//阻塞赋值模块
module block
     (input clk,a,
      output reg c,b);
always @(posedge clk)
  begin
  b = a;
  c = b;
  end
endmodule
```

将上面两段代码进行综合和仿真，可分别得到如图 5.6 和图 5.7 所示的波形。

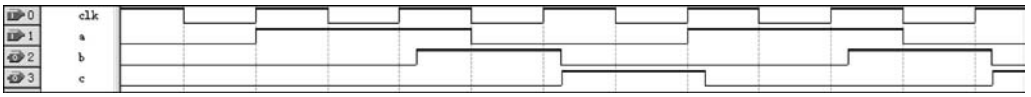


图 5.6 非阻塞赋值的时序仿真波形

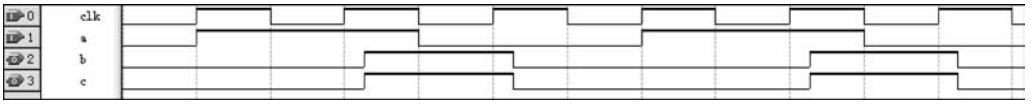


图 5.7 阻塞赋值的时序仿真波形

从图中可看出二者的区别：对于非阻塞赋值，c 的值落后 b 的值一个时钟周期，这是因为该 always 块中两条语句是同时执行的，因此每次执行完后，b 的值得到更新，而 c 的值仍是上一时钟周期的 b 值。对于阻塞赋值，c 的值和 b 的值相同，因为 b 的值是立即更新的，更新后又赋给了 c，因此 c 与 b 的值相同。

综合后的电路分别如图 5.8 和图 5.9 所示。

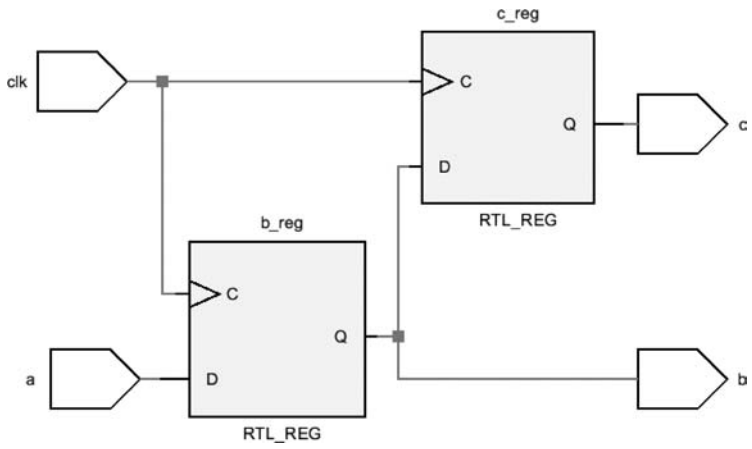


图 5.8 非阻塞赋值综合结果

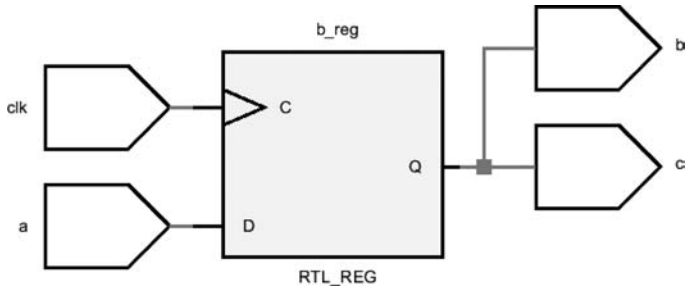


图 5.9　阻塞赋值综合结果

通过上面的讨论可以认为，在 always 过程块中，阻塞赋值可以理解为赋值语句是顺序执行的，而非阻塞赋值可以理解为赋值语句是并发执行的。为避免出错，在同一块内，最好不要将输出再作为输入使用。为使阻塞赋值方式完成与上述非阻塞赋值同样的功能，可采用两个 always 块来实现，如下面的代码所示。其中的两个 always 过程块是并发执行的。

```
module non_block(input clk,a,
                 output reg c,b);
always @(posedge clk)
   begin  b = a;  end
always @(posedge clk)
   begin  c = b;  end
endmodule
```

阻塞赋值与非阻塞赋值是学习 Verilog 语言的难点之一，这两种赋值方式将在后面进一步讲解。

5.4　条件语句

条件语句有 if-else 语句和 case 语句两种，都属于顺序语句，应放在 always 块内。下面分别介绍这两种语句。

5.4.1　if-else 语句

if 语句的格式与 C 语言中的 if-else 语句格式类似，使用方法有以下几种。

```
(1)if(表达式)          语句 1;                  //非完整性 if 语句
(2)if(表达式)          语句 1;                  //二重选择的 if 语句
   else                语句 2;
(3)if(表达式 1)        语句 1;                  //多重选择的 if 语句
   else if(表达式 2)   语句 2;
```

```
    else if(表达式 3)   语句 3;
    …
    else if(表达式 n)   语句 n;
    else                语句 n+1;
```

在上述方式中,“表达式”一般为逻辑表达式或关系表达式,也可能是 1 位的变量。系统对表达式的值进行判断,若为 0、x、z,则按“假”处理;若为 1,则按“真”处理,执行指定语句。语句可以是单句,也可以是多句,多句时用 begin-end 块语句括起来。if 语句也可以多重嵌套,对于 if 语句的嵌套,若不清楚 if 和 else 的匹配,最好用 begin-end 语句括起来。

下面举例说明 if 语句常用的几种使用方法。

1. 二重选择的 if 语句

首先判断条件是否成立,如果 if 语句中的条件成立,那么程序会执行语句 1;否则,程序执行语句 2。如例 5.12 是用二重选择的 if 语句描述的三态非门。

【例 5.12】 二重选择 if 语句描述的三态非门。

```
module tri_not(x,oe,y);
input x,oe; output reg y;
always @(x,oe)
  begin  if(!oe)    y<=~x;
           else     y<=1'bZ;
  end
endmodule
```

2. 多重选择的 if 语句

例 5.13 用多重选择 if 语句描述了一个 1 位二进制数比较器。

【例 5.13】 比较两个 1 位二进制数大小。

```
module compare1(a,b,less,equ,larg);
input a,b; output reg less,equ,larg;
always @(a,b)
  begin if(a>b) begin larg<=1'b1;equ<=1'b0;less<=1'b0;end
    else if(a==b) begin equ<=1'b1;larg<=1'b0;less<=1'b0;end
    else begin less<=1'b1;larg<=1'b0;equ<=1'b0;end
end
endmodule
```

3. 多重嵌套的 if 语句

if 语句可以嵌套，多用于描述具有复杂控制功能的逻辑电路。

多重嵌套的 if 语句的格式如下。

```
if(条件 1)  语句 1;
if(条件 2)  语句 2;
    …
```

例 5.14 是用多重嵌套的 if 语句实现的模为 60 的 8421 BCD 码加法计数器。

【例 5.14】 模为 60 的 8421 BCD 码加法计数器。

```
module count60                                    //模块声明采用 Verilog - 2001 格式
             ( input load,clk,reset,
               input[7:0] data,
               output reg[7:0] qout,
               output cout);
always @(posedge clk)                             //时钟上升沿时计数
  begin
    if(reset)       qout <= 0;                    //同步复位
    else if(load)   qout <= data;                 //同步置数
    else  begin
        if(qout[3:0] == 9)                        //低位是否为 9
          begin qout[3:0]<= 0;                    //回 0
          if(qout[7:4] == 5)  qout[7:4]<= 0;      //判断高位是否为 5,若是,则回 0
          else qout[7:4]<= qout[7:4] + 1;         //高位不为 5,则加 1
          end
        else qout[3:0]<= qout[3:0] + 1;           //低位不为 9,则加 1
          end
  end
assign cout = (qout == 8'h59)?1:0;                //产生进位输出信号
endmodule
```

5.4.2 case 语句

相对 if 语句只有两个分支而言，case 语句是一种多分支语句，故 case 语句多用于多条件译码电路，如描述译码器、数据选择器、状态机及微处理器的指令译码等。case 语句有 case、casez、casex 三种表示方式，这里分别予以说明。

1. case 语句

case 语句的使用格式如下：

```
case (敏感表达式)
      值 1: 语句 1;                          //case 分支项
      值 2: 语句 2;
          …
      值 n: 语句 n;
      default: 语句 n + 1;
endcase
```

当敏感表达式的值为 1 时,执行语句 1;值为 2 时,执行语句 2;以此类推;若敏感表达式的值与上面列出的值都不符,则执行 default 后面的语句。若前面已列出了敏感表达式所有可能的取值,则 default 语句可以省略。

例 5.15 是一个用 case 语句描述的 3 人表决电路,其综合结果见图 5.10,该电路由与门、或门实现。

【例 5.15】 用 case 语句描述 3 人表决电路。

```
module vote3                                          //模块声明采用 Verilog - 2001 格式
            ( input a,b,c,
              output reg pass);
always @(a,b,c)
  begin
    case({a,b,c})                                     //用 case 语句进行译码
    3'b000,3'b001,3'b010,3'b100: pass = 1'b0;         //表决不通过
    3'b011,3'b101,3'b110,3'b111: pass = 1'b1;         //表决通过
                  //注意多个选项间用逗号,连接
    default: pass = 1'b0;
    endcase
  end
endmodule
```

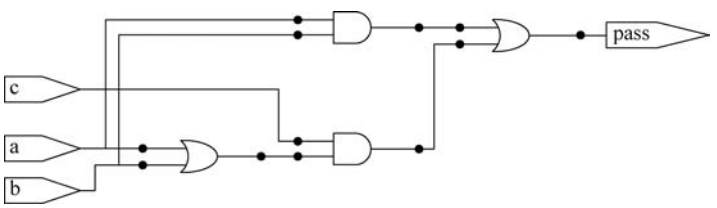
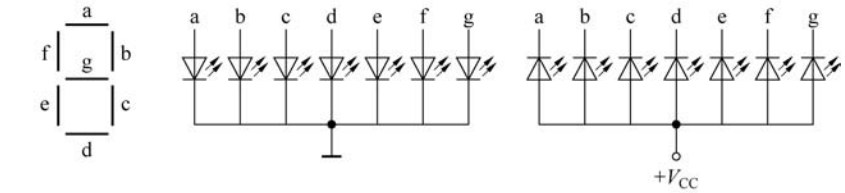


图 5.10　3 人表决电路综合结果

下面的例子是用 case 语句编写的 BCD 码 7 段数码管译码电路,实现 4 位 8421 BCD 码到 7 段数码管显示译码的功能。7 段数码管实际上是由 7 个长条形的发光二极管组成的(一般用 a、b、c、d、e、f、g 分别表示 7 个发光二极管),多用于显示字母、数字。图 5.11 是 7 段数码管的结构与共阴极、共阳极两种连接方式示意图。假定采用共阴极连接方式,用 7 段数码管显示 0～9 这 10 个数字,则相应的译码电路的 Verilog 描述如例 5.16 所示。



(a) 7段数据管结构　　(b) 共阴极连接　　(c) 共阳极连接

图 5.11　7 段数码管

【例 5.16】 BCD 码 7 段数码管译码器。

```
module decode4_7
               ( input D3,D2,D1,D0,                    //输入的 4 位 BCD 码
                 output reg a,b,c,d,e,f,g);
always @ *                                            //使用通配符
  begin
    case({D3,D2,D1,D0})                               //用 case 语句进行译码
    4'd0:{a,b,c,d,e,f,g} = 7'b1111110;                //显示 0
    4'd1:{a,b,c,d,e,f,g} = 7'b0110000;                //显示 1
    4'd2:{a,b,c,d,e,f,g} = 7'b1101101;                //显示 2
    4'd3:{a,b,c,d,e,f,g} = 7'b1111001;                //显示 3
    4'd4:{a,b,c,d,e,f,g} = 7'b0110011;                //显示 4
    4'd5:{a,b,c,d,e,f,g} = 7'b1011011;                //显示 5
    4'd6:{a,b,c,d,e,f,g} = 7'b1011111;                //显示 6
    4'd7:{a,b,c,d,e,f,g} = 7'b1110000;                //显示 7
    4'd8:{a,b,c,d,e,f,g} = 7'b1111111;                //显示 8
    4'd9:{a,b,c,d,e,f,g} = 7'b1111011;                //显示 9
    default:{a,b,c,d,e,f,g} = 7'b1111110;             //其他均显示 0
    endcase
  end
endmodule
```

例 5.17 是用 case 语句描述的 JK 触发器。

【例 5.17】 用 case 语句描述下降沿触发的 JK 触发器。

```
module jk_ff
             (input clk,j,k,
              output reg q);
always @(negedge clk)
  begin
    case({j,k})
    2'b00: q <= q;                  //保持
    2'b01: q <= 1'b0;               //置 0
    2'b10: q <= 1'b1;               //置 1
    2'b11: q <= ~q;                 //翻转
    endcase
  end
endmodule
```

从例 5.17 可以看出，用 case 语句描述实际上就是将模块的真值表描述出来，如果已知模块的真值表，不妨用 case 语句对其进行描述。

2. casez 与 casex 语句

在 case 语句中，敏感表达式与值 1～n 的比较是一种全等比较，必须保证两者的对应位全等。casez 与 casex 语句是 case 语句的两种变体，在 casez 语句中，如果分支表达式某些位的值为高阻 z，那么对这些位的比较就不予考虑，因此只需关注其他位的比较结果。而在 casex 语句中，则把这种处理方式进一步扩展到对 x 的处理。也就是说，如果比较的双方有一方的某些位的值是 x 或 z，那么这些位的比较就都不予考虑。

表 5.2 是 case、casez 和 casex 进行比较时的规则。

表 5.2　case、casez 和 casex 语句的比较规则

case	0	1	x	z	casez	0	1	x	z	casex	0	1	x	z
0	1	0	0	0	0	1	0	0	1	0	1	0	1	1
1	0	1	0	0	1	0	1	0	1	1	0	1	1	1
x	0	0	1	0	x	0	0	1	1	x	1	1	1	1
z	0	0	0	1	z	1	1	1	1	z	1	1	1	1

此外，还有另一种标识 x 或 z 的方式，即用表示无关值的符号“?”来表示。例如：

```
case(a)
2'b1x:out = 1;                    //只有 a = 1x,才有 out = 1
casez(a)
2'b1x:out = 1;                    //如果 a = 1x、1z,则有 out = 1
casex(a)
2'b1x:out = 1;                    //如果 a = 10、11、1x、1z 等,都有 out = 1
casez(a)
3'b1??:out = 1;                   //如果 a = 100、101、110、111 或 1xx、1zz 等,都有 out = 1
3'b01?:out = 1;                   //如果 a = 010、011、01x、01z,都有 out = 1
```

例 5.18 是一个采用 casez 语句描述并使用了符号“?”的数据选择器的例子。

【例 5.18】 用 casez 语句描述数据选择器。

```
module mux_casez
               (input a,b,c,d, input[3:0] select,
                output reg out);
always @ *
begin
    casez(select)
    4'b???1:out = a;
    4'b??1?:out = b;
    4'b?1??:out = c;
    4'b1???:out = d;                  //不需再加 default 语句
```

```
    endcase
  end
endmodule
```

在使用条件语句时，应注意列出所有条件分支，否则，编译器认为条件不满足时，会引进一个触发器保持原值。这一点可用于设计时序电路，而在组合电路设计中，应避免这种隐含触发器的存在。当然，一般不可能列出所有分支，因为每一变量至少有 4 种取值：0、1、z、x。为包含所有分支，可在 if 语句最后加上 else 语句；在 case 语句的最后加上 default 语句。

例 5.19 是一个隐含锁存器的例子。

【例 5.19】 隐含锁存器。

```
module buried_ff
              (input b,a,
               output reg c);
always @(a or b)
    begin
     if((b == 1)&&(a == 1))  c = a&b;
    end
endmodule
```

设计者原意是设计一个 2 输入与门，但因 if 语句中没有 else 语句，在对此语句逻辑综合时会默认 else 语句为“c＝c；”，即保持不变，所以形成了一个隐含锁存器。该例的综合结果如图 5.12 所示。

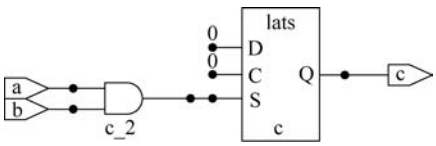


图 5.12 隐含锁存器

在对例 5.16 仿真，出现 a＝1 且 b＝1，c＝1 之后 c 的值会一直维持为 1。为改正此错误，只需加上“else c＝0；”语句即可，即

```
always @(a or b)
begin  if((b == 1)&&(a == 1)) c = a&b;
 else c = 0;
end
```

5.5 循环语句

Verilog 语言中存在 4 种类型的循环语句，用来控制语句的执行次数，分别如下。

(1) for：有条件的循环语句。

(2) repeat：连续执行一条语句 n 次。

(3) while：执行一条语句直到某个条件不满足。

(4) forever：连续地执行语句；多用在 initial 块中，以生成时钟等周期性波形。

5.5.1 for 语句

for 语句的使用格式如下(同 C 语言)。

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

下面通过 7 人表决器的例子说明 for 语句的使用：通过一个循环语句统计赞成的人数，若超过 4 人赞成，则通过。用 vote[7:1]表示 7 人的投票情况，1 代表赞成，即 vote[i]为 1 代表第 i 个人赞成，pass=1 表示表决通过。

【例 5.20】 用 for 语句描述 7 人投票表决器。

```
module voter7
               (input[7:1] vote,
                output reg pass);
reg[2:0] sum; integer i;
always @(vote)
  begin  sum = 0;
    for(i = 1;i <= 7;i = i + 1)                    //for 语句
        if(vote[i])  sum = sum + 1;
        if(sum[2])   pass = 1;                     //若超过 4 人赞成,则 pass = 1
        else         pass = 0;
  end
endmodule
```

例 5.21 中用 for 循环语句实现了两个 8 位二进制数的乘法操作。

【例 5.21】 用 for 语句实现两个 8 位二进制数相乘。

```
module mult_for                                    //模块声明采用 Verilog - 2001 格式
                #(parameter SIZE = 8)
                  (input[SIZE:1] a,b,              //操作数
                  output reg[2 * SIZE:1] outcome); //结果
integer i;
always @(a or b)
      begin  outcome <= 0;
        for(i = 1;i <= SIZE;i = i + 1)             //for 语句
        if(b[i]) outcome <= outcome + (a <<(i - 1));
       end
endmodule
```

例 5.22 是一个用 for 循环语句生成奇校验位的例子。

【例 5.22】 用 for 循环语句生成奇校验位。

```
module parity_check
                (input[7:0] a,
                output reg y);
integer i;
always @(a)
     begin   y = 1'b1;                                  //注意此处不能采用非阻塞赋值<=
     for(i = 0;i <= 7;i = i + 1)                        //for 语句
       y = y ^ a[i];   end                              //此处不能采用非阻塞赋值<=
endmodule
```

在例 5.22 中，for 循环语句执行 1⊕a[0]⊕a[1]⊕a[2]⊕a[3]⊕a[4]⊕a[5]⊕a[6]⊕a[7]运算，综合后生成的 RTL 综合结果如图 5.13 所示。如果将变量 y 的初值改为 0，则例 5.22 变为偶校验电路。

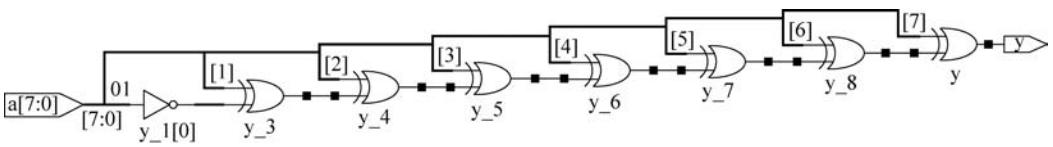


图 5.13 奇校验电路 RTL 综合结果

大多数综合器都支持 for 循环语句，在可综合的设计中，若需使用循环语句，应首先考虑用 for 语句实现。

5.5.2 repeat、while、forever 语句

1. repeat 语句

repeat 语句的使用格式如下。

```
repeat(循环次数表达式) begin
                        语句或语句块
                        end
```

例 5.23 利用 repeat 循环语句和移位运算符实现了两个 8 位二进制数的乘法。

【例 5.23】 用 repeat 实现两个 8 位二进制数的乘法。

```
module mult_repeat
                    #(parameter SIZE = 8)
                      (input[SIZE:1] a,b,
                       output reg[2 * SIZE:1] result);
reg[2 * SIZE:1] temp_a; reg[SIZE:1] temp_b;
always @(a or b)
  begin
    result = 0; temp_a = a; temp_b = b;
```

```
    repeat(SIZE)                          //repeat 语句,SIZE 为循环次数
       begin
       if(temp_b[1])                      //如果 temp_b 的最低位为 1,就执行下面的加法
       result = result + temp_a;
       temp_a = temp_a << 1;              //操作数 a 左移 1 位
       temp_b = temp_b >> 1;              //操作数 b 右移 1 位
       end
  end
endmodule
```

2. while 语句

while 语句的使用格式如下。

```
while(循环执行条件表达式) begin
                              语句或语句块
                            end
```

while 语句在执行时,首先判断循环执行条件表达式是否为真,若为真,则执行后面的语句或语句块;然后回头判断循环执行条件表达式是否为真,若为真,再执行一遍后面的语句,如此不断,直到条件表达式不为真。因此,在执行语句中必须有一条改变循环执行条件表达式值的语句。

例如在下面的代码中,利用 while 语句统计 rega 变量中 1 的个数。

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

下面的例子分别用 while 和 repeat 语句显示 4 个 32 位整数。

```
module loop1;
integer i;
initial           //repeat 循环
 begin i = 0; repeat(4)
   begin
 $ display("i = % h",i);i = i + 1;
 end  end
endmodule
```

```
module loop2;
integer i;
initial           //while 循环
 begin   i = 0; while(i < 4)
  begin
 $ display("i = % h",i);i = i + 1;
  end  end
endmodule
```

用 ModelSim 软件运行，其输出结果均为

```
i = 00000001                    //i 是 32 位整数
i = 00000002
i = 00000003
i = 00000004
```

3. forever 语句

forever 语句的使用格式如下。

```
forever  begin
     语句或语句块
end
```

forever 循环语句连续不断地执行后面的语句或语句块，常用于产生周期性的波形。forever 语句多用在 initial 语句中，若要用它进行模块描述，可用 disable 语句进行中断。

5.6 编译指示语句

Verilog 语言和 C 语言一样提供了编译指示功能。Verilog 语言允许在程序中使用特殊的编译指示(Compiler Directive)语句，在编译时，通常先对这些指示语句进行预处理，然后再将预处理的结果和源程序一起进行编译。

指示语句以符号“`”开头，以区别于其他语句。Verilog HDL 提供了十几条编译指示语句，如`define、`ifdef、`else、`endif、`restall 等。比较常用的有`define、`include 和`ifdef、`else、`endif。下面分别介绍这些常用语句。

1. 宏替换语句`define

`define 语句用于将一个简单的名字或标识符(或称为宏名)代替一个复杂的名字、字符串或表达式，其使用格式为

```
`define 宏名(标识符) 字符串
```

例如：

```
`define sum ina + inb + inc
```

在上面的语句中，用宏名 sum 代替一个复杂的表达式 ina＋inb＋inc，采用这样的定义后，程序中出现

```
assign out = `sum + ind;            //等价于 out = ina + inb + inc + ind;
```

再如：

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

从上面的例子可以看出：

(1) `define 宏定义语句行末是没有分号的。

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

(3) `define 的作用范围是跨模块(module)的，可以是整个工程，就是说，在一个模块中定义的`define 指令可以被其他模块调用，直到遇到`undef 失效。所以用`define 定义常量和参数时，一般习惯将定义语句放在模块外。与`define 相比，用 parameter 定义的参数作用范围只限于本模块内，但上层模块例化下层模块时，可通过参数传递，重新定义下层模块中参数的值。

2. 文件包含语句`include

`include 是文件包含语句，它可将一个文件全部包含到另一个文件中。其格式为

```
`include "文件名"
```

`include 类似于 C 语言中的#include <filename.h>结构，后者用于将内含全局或公用定义的头文件包含在设计文件中；`include 则用于指定包含任何其他文件的内容。被包含的文件既可以使用相对路径定义，也可以使用绝对路径定义；如果没有路径信息，则默认在当前目录下搜寻要包含的文件。`include 语句后加入的文件名称必须放在双引号中。

使用`include 语句时应注意以下几点。

(1) 一个`include 语句只能指定一个被包含的文件。如果需要包含多个文件，则需要使用多个`include 语句进行包含，多个`include 语句可以写在一行，但命令行中只可以出现空格和注释。例如：

```
`include "file1.v"   `include "file2.v"
```

(2) `include 语句可以出现在源程序的任何地方。被包含的文件若与包含文件不在同一个子目录下，必须指明其路径名。

(3) 文件包含允许多重包含，如文件 1 包含文件 2，文件 2 又包含文件 3，等等。

3. 条件编译语句`ifdef、`else、`endif

条件编译语句`ifdef、`else、`endif 可以指定仅对程序中的部分内容进行编译，这三个语句有以下两种使用形式。

(1) 第一种使用形式：

```
`ifdef  宏名
语句块
`endif
```

这种形式的意思是：若宏名在程序中被定义过(用`define 语句定义)，则下面的语句块参与源文件的编译；否则，该语句块将不参与源文件的编译。

(2) 第二种使用形式：

```
`ifdef  宏名
语句块 1
`else    语句块 2
`endif
```

这种形式的意思是：若宏名在程序中被定义过(用`define 语句定义)，则语句块 1 将被编译到源文件中；否则，语句块 2 将被编译到源文件中。如例 5.24 所示。

【例 5.24】 条件编译举例。

```
module compile
            ( input a,b,
              output out);
`ifdef add                              //宏名为 add
       assign out = a + b;
`else  assign out = a - b;
`endif
endmodule
```

在例 5.24 中，若在程序中定义了“`define add”，则执行“assign out＝a＋b;”操作，若没有该语句，则执行“assign out＝a－b;”操作。

5.7 任务与函数

任务和函数的关键字分别是 task 和 function。利用任务和函数可以把一个大的程序模块分解成许多小的子模块，方便调试，并能使程序结构清晰。

5.7.1 任务

任务(task)定义与调用的格式分别为

```
task <任务名>;                    //注意无端口列表
      端口及数据类型声明语句;
      其他语句;
endtask
```

任务调用的格式为

```
<任务名>(端口 1,端口 2,…);
```

需要注意的是,任务调用时和定义时的端口变量应是一一对应的。例如,下面是一个定义任务的例子。

```
task test;
input in1,in2; output out1,out2;
#1 out1 = in1&in2;
#1 out2 = in1|in2;
endtask
```

当调用该任务时,可使用如下语句:

```
test(data1,data2,code1,code2);
```

调用任务 test 时,变量 data1 和 data2 的值赋给 in1 和 in2;任务执行完成后,out1 和 out2 的值则赋给了 code1 和 code2。

在例 5.25 中,定义了一个完成两个操作数按位与操作的任务,然后在后面的算术逻辑单元的描述中,调用该任务完成与操作。

【例 5.25】 任务举例。

```
module alutask(code,a,b,c);
input[1:0] code; input[3:0] a,b;
output reg[4:0] c;
task my_and;                          //任务定义,注意无端口列表
input[3:0] a,b;                       //a,b,out 名称的作用域范围为 task 任务内部
output[4:0] out;
integer i;  begin for(i = 3;i > = 0;i = i - 1)
     out[i] = a[i]&b[i];              //按位与
end
endtask
always@(code or a or b)
     begin  case(code)
     2'b00:my_and(a,b,c);             /* 调用任务 my_and,需注意端口列表的顺序应与任务
定义时一致,这里的 a,b,c 分别对应任务定义中的 a,b,out */
     2'b01:c = a|b;                   //或
     2'b10:c = a - b;                 //相减
     2'b11:c = a + b;                 //相加
     endcase
     end
endmodule
```

为检验其功能,编写例 5.26 的激励脚本并对其仿真。

【例 5.26】 激励脚本。

```
`timescale 100 ps/ 1 ps
module alutask_vlg_tst();
parameter DELY = 100;
reg eachvec;
reg [3:0] a;reg [3:0] b;reg [1:0] code;
wire [4:0]  c;
 alutask i1(  .a(a),.b(b),.c(c),.code(code));
initial    begin
code = 4'd0;a = 4'b0000;b = 4'b1111;
#DELY   code = 4'd0;a = 4'b0111;b = 4'b1101;
#DELY   code = 4'd1;a = 4'b0001;b = 4'b0011;
#DELY   code = 4'd2;a = 4'b1001;b = 4'b0011;
#DELY   code = 4'd3;a = 4'b0011;b = 4'b0001;
#DELY   code = 4'd3;a = 4'b0111;b = 4'b1001;
$display("Running testbench");
end
always  begin
@eachvec;
end
endmodule
```

用 ModelSim 运行例 5.26 的脚本，得到如图 5.14 所示的仿真波形。

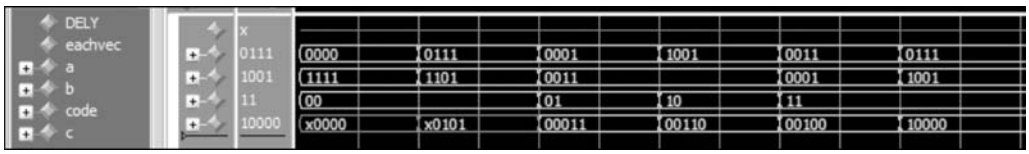


图 5.14　　例 5.26 的仿真波形

注意：在使用任务时，应注意以下几点。

(1) 任务的定义与调用必须在一个 module 内。

(2) 定义任务时，没有端口名列表，但需紧接着进行输入/输出端口和数据类型的说明。

(3) 当任务被调用时，任务被激活。任务的调用与模块调用一样，通过任务名调用实现。调用时，需列出端口名列表，端口名的排序和类型必须与任务定义时相一致。

(4) 一个任务可以调用别的任务和函数，可以调用的任务和函数个数不受限制。

5.7.2 函数

在 Verilog 模块中，如果多次用到重复的代码，则可以把这部分重复代码摘取出来，

定义成函数(function)。在综合时,每调用一次函数,则复制或平铺(flatten)该电路一次,所以函数不宜过于复杂。

函数可以有一个或者多个输入,但只能返回一个值,通常在表达式中调用函数的返回值。函数的定义格式如下。

```
function  <返回值位宽或类型说明> 函数名;
              端口声明;
              局部变量定义;
              其他语句;
endfunction
```

<返回值位宽或类型说明>是一个可选项,如果默认,则返回值为 1 位寄存器类型的数据。

函数的调用是通过将函数作为表达式中的操作数来实现的。调用格式如下。

```
<函数名>(<表达式><表达式>);
```

例 5.27 用函数定义了一个 8-3 编码器,并使用 assign 语句调用了该函数。

【例 5.27】 用函数和 case 语句描述编码器(不含优先顺序)。

```
module code_83(din,dout);
input[7:0] din; output[2:0] dout;
function[2:0] code;                    //函数定义
input[7:0] din;                        //函数只有输入,输出为函数名本身
     casex(din)
     8'b1xxx_xxxx:code = 3'h7;
     8'b01xx_xxxx:code = 3'h6;
     8'b001x_xxxx:code = 3'h5;
     8'b0001_xxxx:code = 3'h4;
     8'b0000_1xxx:code = 3'h3;
     8'b0000_01xx:code = 3'h2;
     8'b0000_001x:code = 3'h1;
     8'b0000_000x:code = 3'h0;
     default:code = 3'hx;
     endcase
endfunction
assign dout = code(din);               //函数调用
endmodule
```

与 C 语言相类似,Verilog 语言使用函数以适应对不同操作数采取同一运算的操作。函数在综合时被转换成具有独立运算功能的电路,每调用一次函数相当于改变这部分电路的输入以得到相应的计算结果。

例 5.28 中定义了一个实现阶乘运算的函数 factorial,该函数返回一个 32 位的寄存器类型的值。采用同步时钟触发运算的执行,每个 clk 时钟周期都会执行一次运算。

【例 5.28】 阶乘运算函数。

```
module funct(clk,n,result,reset);
input reset,clk; input[3:0] n; output reg[31:0] result;
always @(posedge clk)                          //在 clk 的上升沿时执行运算
     begin if(!reset) result <= 0;
     else
     begin result <= 2 * factorial(n); end   //调用 factorial 函数
     end
function[31:0] factorial;                      //阶乘运算函数定义(注意无端口列表)
input[3:0] opa;                                //函数只能定义输入端,输出端口为函数名本身
reg[3:0] i;
     begin  factorial = (opa >= 4'b1)?1:0;
     for(i = 2;i <= opa;i = i + 1)             //for 语句若要综合,opa 应赋具体的数值,如 9
     factorial = i * factorial;                //阶乘运算
     end
endfunction
endmodule
```

注意：函数的定义中蕴涵了一个与函数同名的、函数内部的寄存器。在函数定义时，将函数返回值所使用的寄存器名称设为与函数同名的内部变量，因此函数名被赋予的值就是函数的返回值。

Verilog-2001 标准中定义了一种递归函数(Function Automatic)，增加了一个关键字 automatic，表示函数的迭代调用。如上面的阶乘运算，可采用递归函数来描述，如例 5.29 所示，通过函数自身的迭代调用，实现了 32 位无符号整数的阶乘运算($n!$)。

比较例 5.28 与例 5.29 的异同，可体会函数与递归函数的区别。

【例 5.29】 阶乘递归函数。

```
module tryfact;
function automatic integer factorial;              //函数定义
input [31:0] operand;
if(operand >= 2)
factorial = factorial(operand - 1) * operand;
else
factorial = 1;
endfunction
integer result;
integer n;
initial begin
for(n = 0; n <= 7; n = n + 1) begin
result = factorial(n);                             //函数调用
$display(" %0d factorial = %0d", n, result);
end  end
endmodule // tryfact
```

例 5.29 的仿真结果如下：

```
0 factorial = 1
1 factorial = 1
2 factorial = 2
3 factorial = 6
4 factorial = 24
5 factorial = 120
6 factorial = 720
7 factorial = 5040
```

注意：在使用函数时，应注意以下几点。

(1) 函数的定义与调用必须在一个 module 内。

(2) 函数只允许有输入变量且必须至少有一个输入变量，输出变量由函数名本身担任，如在例 5.28 中，函数名 factorial 就是输出变量，在调用该函数时，“result＜＝2 * factorial(n);”自动将 n 的值赋给函数的输入变量 opa，完成函数计算后，将结果通过 factorial 名字本身返回，作为一个操作数参与 result 表达式的计算。因此，在定义函数时，需声明函数名的数据类型和位宽。

(3) 定义函数时没有端口名列表，但调用函数时需列出端口名列表，端口名的排序和类型必须与定义时一致。这一点与任务相同。

(4) 函数可以出现在持续赋值 assign 的右端表达式中。

(5) 函数的使用与任务相比有更多的限制和约束。函数不能启动任务，而任务可以调用别的任务和函数，且调用任务和函数的个数不受限制。在函数中不能包含有任何时间控制语句。

表 5.3 对 task 与 function 进行了比较。

表 5.3　task 与 function 的比较

比 较 项 目	task	function
输入与输出	可有任意个各种类型的参数	至少有一个输入，不能将 inout 类型作为输出
调用	任务只可在过程语句中调用，不能在连续赋值语句 assign 中调用	函数可作为表达式中的一个操作数来调用，在过程赋值和连续赋值语句中均可以调用
定时事件控制(＃，@和 wait)	任务可以包含定时和事件控制语句	函数不能包含这些语句
调用其他任务和函数	任务可调用其他任务和函数	函数可调用其他函数，但不可以调用其他任务
返回值	任务不向表达式返回值	函数向调用它的表达式返回一个值

合理使用 task 和 function 会使程序显得结构清晰而简单，一般的综合器对 task 和 function 都是支持的，但也有的综合器不支持 task。

5.8 Verilog-2001 语言标准

Verilog 语言处于不断的发展过程中，1995 年 IEEE 将 Verilog 采纳为标准，推出 Verilog-1995 标准(即 IEEE 1364-1995)；2001 年 3 月，IEEE 又批准了 Verilog-2001 标准(即 IEEE 1364-2001)。

目前，几乎所有的综合器、仿真器都能很好地支持 Verilog-2001 标准，Vivado 软件支持的 Verilog 标准包括 Verilog-1995、Verilog-2001 和 SystemVerilog。有必要较为深入地了解和学习 Verilog-2001 标准。

5.8.1 Verilog-2001 改进和增强的语法结构

Verilog-2001 标准对 Verilog 语言的改进和增强主要表现在以下三方面。

- 提高 Verilog 语言的行为级和 RTL 级建模的能力。
- 改进 Verilog 语言在深亚微米设计和 IP 建模方面的能力。
- 纠正和改进了 Verilog-1995 标准中的错误和易产生歧义之处。

下面举例说明具体的改进。

1. ANSI C 风格的模块声明

Verilog-2001 标准改进了端口的声明语句，使其更接近 ANSI C 语言的风格，可用于 module、task 和 function。同时允许将端口声明和数据类型声明放在同一条语句中。例如，在 Verilog-1995 标准中，可采用如下方式声明一个 FIFO 模块。

```
module fifo(in,clk,read,write,reset,out,full,empty);
parameter MSB = 3,DEPTH = 4;
input[MSB:0] in;
input clk,read,write,reset;
output[MSB:0] out; output full,empty;
reg[MSB:0] out; reg full,empty;
```

上面的模块声明在 Verilog-2001 标准中可以写成下面的形式。

```
module fifo_2001
        #(parameter MSB = 3,DEPTH = 4)        //参数定义，注意前面有"#"
          ( input[MSB:0] in,                  //端口声明和数据类型声明放在同一条语句中
            input clk,read,write,reset,
            output reg[MSB:0] out,
            output reg full,empty);
```

例 5.30 的 4 位格雷码计数器的模块声明部分采用了 Verilog-2001 格式。

【例 5.30】 4 位格雷码计数器。

```
module graycount  #(parameter WIDTH = 4)
          (output reg[WIDTH-1:0]  graycount,          //格雷码输出信号
           input wire  enable,clear,clk);             //使能、清零、时钟信号
reg [WIDTH-1:0]  bincount;
always @ (posedge clk)
  if(clear) begin
   bincount <= {WIDTH{1'b 0}} + 1;
   graycount <= {WIDTH{1'b 0}};
   end
   else if(enable) begin
     bincount <= bincount + 1;
     graycount <= {bincount[WIDTH-1],
     bincount[WIDTH-2:0] ^ bincount[WIDTH-1:1]};
    end
endmodule
```

4 位格雷码计数器的行为级仿真波形如图 5.15 所示，其输出按照格雷码编码，相邻码字只有一个比特位不同。

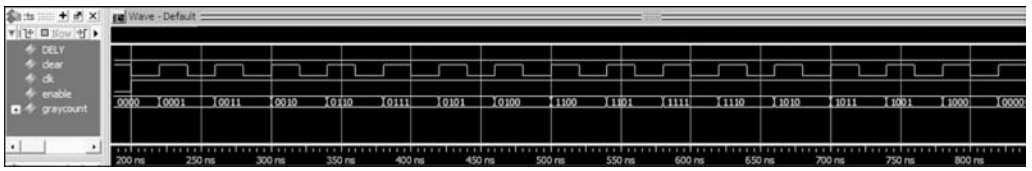


图 5.15 4 位格雷码计数器的行为仿真波形

2. 逗号分隔的敏感信号列表

在 Verilog-1995 标准中书写敏感信号列表时，通常用 or 来连接敏感信号。例如：

```
always @(a or b or cin)
    {cout,sum} = a + b + cin;
always @(posedge clk or negedge clr)
     if(!clr) q <= 0;  else q <= d;
```

在 Verilog-2001 标准中可用逗号分隔敏感信号，上面的语句可写为如下形式。

```
always @(a, b, cin)                    //用逗号分隔信号
       {cout,sum} = a + b + cin;
always @(posedge clock,negedge clr)
       if(!clr) q <= 0; else  q <= d;
```

3. 在组合逻辑敏感信号列表中使用通配符 *

用 always 过程块描述组合逻辑时，应在敏感信号列表中列出所有的输入信号，在

Verilog-2001 标准中可用通配符“*”来表示包含该过程块中的所有输入信号变量。

下面是在 Verilog-1995 标准和 Verilog-2001 标准中，4 选 1 MUX 的敏感信号列表书写格式的对比：

```
//Verilog - 1995
always @(sel or a or b or c or d)
      case(sel)
      2'b00:y = a;
      2'b01:y = b;
      2'b10:y = c;
      2'b11:y = d;
      endcase
```

```
//Verilog - 2001
always @ *              //通配符
      case(sel)
      2'b00:y = a;
      2'b01:y = b;
      2'b10:y = c;
      2'b11:y = d;
      endcase
```

4. generate 语句

Verilog-2001 标准新增了语句 generate。generate 循环可以产生一个对象（如 module、primitive，或者 variable、net、task、function、assign、initial 和 always）的多个例化，为可变尺度的设计提供便利。

generate 语句一般和循环语句、条件语句（for、if、case）一起使用。为此，Verilog-2001 标准增加了 4 个关键字 generate、endgenerate、genvar 和 localparam。genvar 是一个新的数据类型，用在 generate 循环中的标尺变量必须定义为 genvar 型数据。还要注意的是，for 循环的内容必须加 begin 和 end（即使只有一条语句），且必须给 begin 和 end 块语句起个名字。

下面是一个用 generate 语句描述的行波进位加法器的例子，它采用了 generate 语句和 for 循环产生元件的例化和元件间的连接关系。

【例 5.31】 采用 generate 语句和 for 语句循环描述的 4 位行波进位加法器。

```
module add_ripple #(parameter SIZE = 4)
(input[SIZE - 1:0] a,b,
 input cin,
 output[SIZE - 1:0] sum,
output cout);

wire[SIZE:0] c;
assign c[0] = cin;
generate
genvar i;
for(i = 0;i < SIZE;i = i + 1)
   begin : add
    wire n1,n2,n3;
xor g1(n1,a[i],b[i]);
xor g2(sum[i],n1,c[i]);
and g3(n2,a[i],b[i]);
and g4(n3,n1,c[i]);
```

```
or g5(c[i+1],n2,n3);  end
endgenerate
assign cout = c[SIZE];
endmodule
```

对例 5.31 用 Vivado 软件综合，其 RTL 综合原理图如图 5.16 所示。

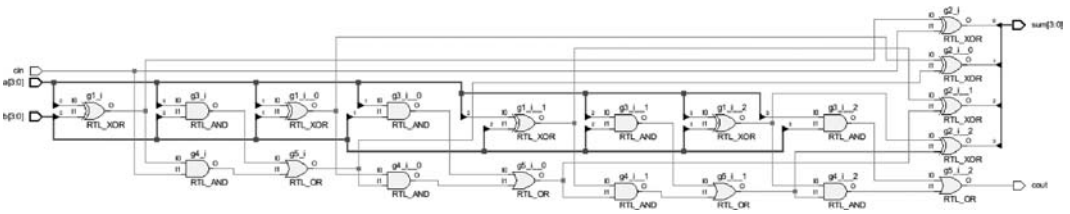


图 5.16　4 位行波进位加法器 RTL 综合原理图

下面的例子用 generate 语句描述一个可扩展的乘法器，当乘法器的 a 和 b 的位宽小于 8 时，生成 CLA 超前进位乘法器；否则生成 WALLACE 树状乘法器。

```
module multiplier(a, b, product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width + b_width;
input [a_width - 1:0] a;
input [b_width - 1:0] b;
output[product_width - 1:0] product;
generate
if((a_width < 8) || (b_width < 8))
CLA_multiplier #(a_width, b_width)
u1 (a, b, product);
else
WALLACE_multiplier #(a_width, b_width)
u1 (a, b, product);
endgenerate
endmodule
```

5. 带符号的算术扩展

signed 是 Verilog-1995 标准中的保留字，未使用。在 Verilog-2001 标准中，用 signed 来定义带符号的数据类型、端口、整数、函数等。

在 Verilog-2001 标准中，对带符号的算术运算做了如下几点扩充。

(1) wire 型和 reg 型变量可以声明为带符号(signed)变量。例如：

```
wire signed[7:0] a,b;
reg signed[15:0] data;
output signed[15:0] sum;
```

(2) 任何进制的整数都可以带符号；参数也可以带符号。例如：

```
12'sh54f                    //一个 12 位的十六进制带符号整数 54f
parameter p0 = 2`sb00,p1 = 2`sb01;
```

(3) 函数的返回值可以有符号。例如：

```
function signed[31:0] alu;
```

(4) 增加了算术移位操作符。

Verilog-2001 标准增加了算术移位操作符>>>和<<<，对于有符号数，执行算术移位操作时，用符号位填补移出的位，以保持数值的符号。例如，若定义有符号二进制数 A = 8'sb10100011，则执行逻辑右移和算术右移后的结果如下。

```
A >> 3;                          //逻辑右移后其值为 8'b00010100
A >>> 3;                         //算术右移后其值为 8'b11110100
```

(5) 新增了系统函数 $ signed()和 $ unsigned()，可以将数值强制转换为带符号的值或不带符号的值。例如：

```
reg[63:0]  a;                    //定义 a 为无符号数据类型
always@(a)  begin
  result1 = a/2;                 //无符号运算
  result2 = $ signed(a)/2;       //a 变为带符号数
end
```

6. 指数运算符 **

Verilog-2001 标准增加了指数运算符 **，执行指数运算，一般使用的是底数为 2 的指数运算(2^n)。例如：

```
parameter WIDTH = 16;
parameter DEPTH = 8;
reg[WIDTH - 1:0] data [0:(2 ** DEPTH) - 1];
                                 //定义了一个位宽为 16 位,2^8(256)个单元的存储器
```

7. 变量声明时进行赋值

Verilog-2001 标准规定可以在变量声明时对其赋初始值，所赋的值必须是常量，并且在下次赋值之前，变量都会保持该初始值不变。变量在声明时的赋值不适用于矩阵。

如下面的例子，在 Verilog-1995 标准中需要先声明一个 reg 变量 a，然后在 initial 块中为其赋值为 4'h4；而在 Verilog-2001 标准中可直接在声明时赋值，两者是等效的。

```
//Verilog-1995
reg[3:0] a;
 initial
  a = 4'h4;
```

```
 //Verilog-2001
reg[3:0] a = 4'h4;
```

也可同时声明多个变量，为其中的一个或者几个变量赋值，例如：

```
integer i = 0, j, k = 1;
real r1 = 2.5, n300k = 3E6;
```

在声明矩阵时，为其赋值是非法的，如下面的代码是非法的：

```
reg [3:0] array [3:0] = 0;                    //非法
```

8. 常数函数

Verilog-2001 标准增加了一类特殊的函数——常数函数，其定义和其他 Verilog 函数的定义相同，不同之处在于其赋值是在编译或详细描述(elaboration)时被确定的。

常数函数有助于创建可改变维数和规模的可重用模型。如下例定义了一个常数函数 clogb2，该函数返回一个整数，可根据 ram 的深度(ram 的单元数)来确定 ram 地址线的宽度。

```
module ram(address_bus, write, select, data);
parameter SIZE = 1024;
input [clogb2(SIZE) - 1:0] address_bus;
...
function integer clogb2 (input integer depth);
begin
   for(clogb2 = 0; depth > 0; clogb2 = clogb2 + 1)
   depth = depth >> 1;
end
endfunction
...
endmodule
```

注意：常数函数只能调用常数函数，不能调用系统函数。常数函数内部用到的参数(parameter)必须在该常数函数被调用之前定义。

9. 向量的位选和域选

在 Verilog-1995 标准中，可以从向量中取出一个或者若干相连比特，称为位选和域选，但被选择的部分必须是固定的。

Verilog-2001 标准对向量的部分选择进行了扩展，增加了索引的部分选择(indexed part selects)方式，其形式如下。

```
[base_expr      +:    width_expr]
// 起始表达式   正偏移     位宽
[base_expr      -:    width_expr]
// 起始表达式   负偏移     位宽
```

包括起始表达式(base_expr)和位宽(width_expr)。其中,位宽必须为常数,而起始表达式可以是变量;偏移方向表示选择区间是表达式加上位宽(正偏移),还是表达式减去位宽(负偏移)。例如:

```
reg [63:0] word;
reg [3:0] byte_num;                    //取值范围: 0～7
wire [7:0] byteN = word[byte_num * 8 +: 8];
```

上例中,如果变量 byte_num 当前的值是 4,则 byteN = word[39:32],起始位为 32(byte_num * 8),终止位 39 由宽度和正偏移 8 确定。再如:

```
reg[63:0] vector1;                     //小端(little-endian)次序
reg[0:63] ventor2;                     //大端(big-endian)次序
Byte = vector1[31-:8];                 //Byte = vector1[31:24]
Byte = vector1[24+:8];                 //Byte = vector1[31:24]
Byte = vector2[31-:8];                 //Byte = vector2[24:31]
Byte = vector2[24+:8];                 //Byte = vector2[24:31]
```

10. 多维矩阵

Verilog-1995 标准中只允许一维的矩阵变量(即 memory),Verilog-2001 标准对其进行了扩展,允许使用多维矩阵;矩阵单元的数据类型也扩展至 Variable 型(如 reg)和 Net 型(如 wire)均可。例如:

```
reg [7:0] array1 [0:255];
  //一维矩阵,存储单元为 reg 型
wire [7:0] out1 = array1[address];
  //一维矩阵,存储单元为 wire 型
wire [7:0] array3 [0:255][0:255][0:15];
  //三维矩阵,存储单元为 wire 型
wire [7:0] out3 = array3[addr1][addr2][addr3];
  //三维矩阵,存储单元为 wire 型
```

11. 矩阵的位选择和部分选择

在 Verilog-1995 标准中,不允许直接访问矩阵的某一位或某几位,必须首先将整个矩阵单元转移到一个暂存变量中,再从暂存变量中访问。例如:

```
reg[7:0]  mem[0:1023];                    //存储器(一维矩阵)
reg[7:0]  temp;
reg[3:0]  vect;
initial
  begin   temp = mem[55];
  vect = temp[3:0];                       //合法
  vect = mem[55][3:0];                    //非法
end
```

而在 Verilog-2001 标准中,可以直接访问矩阵的某个单元的一位或几位。例如:

```
reg [31:0] array2 [0:255][0:15];
wire [7:0] out2 = array2[100][7][31:24];
    //选择宽度为 32 位的二维矩阵中[100][7]单元的[31:24]字节
```

12. 模块实例化时的参数重载

当模块实例化时,其内部定义的参数(parameter)值是可以改变的(或称为参数重载)。在 Verilog-1995 标准中有两种方法改变参数值:一种是使用 defparam 语句显式地重载;另一种就是模块实例化时使用"#"符号隐式地重载,重载的顺序必须与参数在原定义模块中声明的顺序相同,并且不能跳过任何参数。由于这种方法容易出错,而且代码的含义不易理解,所以 Verilog-2001 标准增加了一种在线显式重载(in-line explicit redefinition)参数的方式,这种方式允许在线参数值按照任意顺序排列。例如:

```
module ram(...);                          //ram 模块定义
parameter WIDTH = 8;
parameter SIZE = 256;
...
endmodule

module my_chip (...);
...
RAM ram1 (...);                           //ram 模块例化 1
defparam ram1.SIZE = 1023;
//使用 defparam 语句显式地重新定义 SIZE = 1023
RAM #(8,1023) ram2 (...);                 //ram 模块例化 2
//使用#符号隐式地重载参数,注意参数的排列顺序
RAM #(.SIZE(1023)) ram3 (...);            //ram 模块例化 3
//在线显式重载参数 SIZE 为 1023
endmodule
```

13. register 改为 variable

在 Verilog 语言诞生后,一直用 register 这个词表示一种数据类型,但初学者很容易混淆 register 和硬件中的寄存器概念。而实际中,register 数据类型的变量常被综合器映

射为组合逻辑电路。

在 Verilog-2001 标准中，将 register 一词改为了 variable，以避免混淆。

14. 新增条件编译语句

Verilog-1995 标准支持条件编译语句`ifdef、`else、`endif，可以指定仅对程序中的部分内容进行编译。Verilog-2001 标准增加了条件编译语句`elsif 和`ifndef。

15. 超过 32 位的自动宽度扩展

在 Verilog-1995 标准中对超过 32 位的总线赋高阻时，如果不指定位宽，则只将低 32 位赋成高阻，高位补 0。如果想将所有位都置为高阻，必须明确指定位宽。例如：

```
//Verilog-1995 标准中的超过 32 位的总线赋高阻:
parameter WIDTH = 64;
reg [WIDTH-1:0] data;
data = 'bz;                  //赋值后,data = 'h00000000zzzzzzzz
data = 64'bz;                //赋值后,data = 'hzzzzzzzzzzzzzzzz
```

Verilog-2001 标准改变了赋值扩展规则，将高阻 z 或不定态 x 赋给未指定位宽的信号时，可以自动扩展到信号的整个位宽范围。例如：

```
//Verilog-2001 标准中将高阻或不定态赋给未指定位宽的信号:
parameter WIDTH = 64;
reg [WIDTH-1:0] data;
data = 'bz;                    //赋值后,data = 'hzzzzzzzzzzzzzzzz
```

16. 可重入任务(Reentrant Task)和递归函数(Recursive Function)

Verilog-2001 标准增加了一个关键字 automatic，可用于任务和函数的定义中。

(1) 可重入任务：任务本质上是静态的(Static Task)，同时并发执行的多个任务共享存储区。若某个任务在模块中的多个地方被同时调用，则这两个任务对同一块地址空间进行操作，结果可能是错误的。Verilog-2001 标准中增加了关键字 automatic，空间是动态分配的，使任务成为可重入的。若定义任务时使用了 automatic，则定义一个可重入任务。这两种类型的任务所消耗的资源是不同的。

(2) 递归函数：关键字 automatic 用于函数，表示函数的迭代调用。如在下面的例子中，通过函数自身的迭代调用，实现 32 位无符号整数的阶乘运算($n!$)。

```
function automatic [63:0] factorial;
input [31:0] n;
  if(n == 1)  factorial = 1;
  else
  factorial = n * factorial(n-1);                          //迭代调用
endfunction
```

由于 Verilog-2001 标准增加了关键字 signed，所以函数的定义还可在 automatic 后面加上 signed，返回有符号数。例如：

```
function automatic signed [63:0] factorial;
```

17. 文件和行编译指示

Verilog 编译和仿真工具需要不断地跟踪源代码的行号和文件名，Verilog 可编程语言接口(PLI)可以取得并利用行号和源文件的信息，以标记运行中的错误。但是，如果 Verilog 代码经过其他工具的处理，源代码的行号和文件名可能丢失。故在 Verilog-2001 标准中增加了`line，用来标定源代码的行号和文件名。

5.8.2 属性及 PLI 接口

Verilog-2001 标准还在以下方面做了改进和增强。

1. 设计管理

Verilog-1995 标准将设计管理工作交给软件来承担，但各仿真工具的设计管理方法各不相同，不利于设计的共享。为了更好地在设计人员之间共享 Verilog 设计，并提高某个特定仿真的可重用性，Verilog-2001 标准加强了对设计内容的管理和配置。

Verilog-2001 标准中增加了配置块(Configuration Block)，用它来指定每一个 Verilog 模块的版本及其源代码的位置。配置块位于模块定义之外，可以指定 Verilog 语言程序设计从顶层模块开始执行，找到在顶层模块中实例化的模块，进而确定其源代码的位置，照此顺序，直到确定整个设计的源程序。

Verilog-2001 标准中新增了关键字 config 和 endconfig，还增加了关键字 design、instance、cell、use 和 liblist，以供在配置块中使用。

下面的例子是一个简单的设计配置，test 是一个测试模块(Test Bench)，其中包含了设计模块 myChip，myChip 中又包含了其他实例化模块。

```
module test;
...
myChip dut (...);                              /* 设计模块实例化 */
...
endmodule
module myChip(...);
...
adder a1 (...);
adder a2 (...);
...
endmodule
```

配置块可以指定所有或个别实例化模块的源代码的位置。配置块位于模块定义之外，所以需要重新配置时，Verilog 源代码可以不做任何修改。

在下面的配置块中，design 语句指定了顶层模块及其源代码来源，rtlLib. top 表示顶层模块的源代码来自 rtlLib；default 和 liblist 语句相配合指定了所有在顶层模块中实例化的模块均来自 rtlLib 库和 gateLib 库；又使用 instance 语句具体指定了加法器实例 a2 的源程序来自门级库 gateLib。

```
config cfg4                          //给配置块命名
design rtlLib.top                    //指定从哪里找到顶层模块
default liblist rtlLib gateLib;
     //设置查找实例化模块的默认顺序
instance test.dut.a2 liblist gateLib;
     //明确指定模块例化使用哪一个库: a2 来自门级库 gateLib
endconfig
```

下面的语句指定了 RTL 库和 gateLib 库模块的位置。

```
library rtlLib ./*.v;
//RTL 库模块的位置(位于当前目录下)
library gateLib ./synth_out/*.v;
//gateLib 库模块的位置
```

2. 属性

属性用来向综合工具传递信息，以控制综合工具的行为和操作。属性包含在两个“*”之间，可用于对象的所有实例调用，也可只应用于某一个实例调用。与综合有关的属性语句如下。

```
(* synthesis, async_set_reset[ = "signal_name1,signal_name2,..."] *)
(* synthesis, black_box[ = <optional_value>] *)
(* synthesis, combinational[ = <optional_value>] *)
(* synthesis, fsm_state[ = <encoding_scheme>] *)
(* synthesis, full_case[ = <optional_value>] *)
(* synthesis, implementation = "<value>" *)
(* synthesis, keep[ = <optional_value>] *)
(* synthesis, label = "name" *)
(* synthesis, logic_block[ = <optional_value>] *)
(* synthesis, op_sharing[ = <optional_value>] *)
(* synthesis, parallel_case[ = <optional_value>] *)
(* synthesis, ram_block[ = <optional_value>] *)
(* synthesis, rom_block[ = <optional_value>] *)
(* synthesis, sync_set_reset[ = "signal_name1,signal_name2,..."] *)
(* synthesis, probe_port[ = <optional_value>] *)
```

Verilog 没有定义标准的属性，属性的名字和数值由工具厂商或其他标准来定义，目

前尚无统一的标准。

3. 增强的文件输入、输出操作

Verilog-1995 标准在文件的输入、输出操作方面功能非常有限。文件操作通常借助于 Verilog PLI(编程语言接口),通过与 C 语言的文件输入、输出库的访问来处理,并且规定同时打开的 I/O 文件数目不能超过 31 个。

Verilog-2001 标准增加了新的系统任务和函数,为 Verilog 语言提供了强大的文件输入、输出操作,而不再需要使用 PLI,并扩展了可以同时打开的文件数目至 230 个。这些新增的文件输入、输出系统任务和函数包括 $ferror、$fgetc、$fgets、$fflush、$fread、$fscanf、$fseek、$fscanf、$ftel、$rewind 和 $ungetc;还有读写字符串的系统任务,包括 $sformat、$swrite、$swriteb、$swriteh、$swriteo 和 $sscanf,用于生成格式化的字符串或者从字符串中读取信息。

增加了命令行输入任务 $test$plusargs 和 $value$plusargs。

4. VCD 文件的扩展

VCD 文件用于记录仿真过程中信号的变化,只记录在函数中指定的层次中相关的信号。信息的记录由 VCD 系统任务来完成。在 Verilog-1995 标准中,只有一种类型的 VCD 文件,即四状态类型,这种类型的 VCD 文件只记录变量在 0、1、x 和 z 状态之间的变化,不记录信号强度信息。而在 Verilog-2001 标准中增加了一种扩展类型的 VCD 文件,能够记录变量在所有状态之间的转换,同时记录信号强度信息。

扩展型 VCD 系统任务包括 $dumpports、$dumpportsoff、$dumpportson、$dumpportsall、$dumpportslimit、$dumpportsfulsh 和 $vcdclose。

5. 提高了对 SDF(标准延时文件)的支持

在 Verilog-1995 标准中,specparam 常数只能在 specify 块(指定块)中定义;Verilog-2001 标准允许在模块层级声明和使用 specparam 常数。Verilog-2001 标准基于最新的 SDF 标准(IEEE Std 1497—1999),提高了对 SDF(Standard Delay File)的支持度。

6. 编程语言接口的改进

编程语言接口(Programming Language Interface,PLI)包括三个 C 语言功能库,分别是 ACC、TF 和 VPI。Verilog-2001 标准清理和更正了旧的 ACC 和 TF 库中的许多定义,但并没有增加任何新的功能。Verilog-2001 标准对 PLI 的所有改进都体现在 VPI 库中,包括增加了 6 个 VPI 子程序:vpi_control()、vpi_get_data()、vpi_put_data()、vpi_get_userdata()、vpi_put_userdata()和 vpi_flush(),为用户提供了更大的便利。现将这 6 个函数的功能简单介绍如下。

(1) vpi_control()的作用是传递用户给仿真器的指令。

(2) vpi_get_data()和 vpi_put_data()相对应使用,从一次执行的 save/restart 位置获取数据。语法格式为 vpi_get_data(id, dataLoc, numOfBytes)。

(3) vpi_put_data()的作用是将数据放到一次仿真的 save/restart 位置。其语法格式为 vpi_put_data(id, dataLoc, numOfBytes)。其中,numOfBytes 是个正整数,以字节为单位指定了要放置的数据的数目,dataLoc 代表数据所在的位置,id 代表 vpi_get(vpiSaveRestartID, NULL)返回的 save/restart ID。函数的返回值是数据的字节数,若出错,则返回 0。

(4) vpi_get_userdata()和 vpi_put_userdata()相对应使用。从系统任务或系统函数实例的存储位置读取用户数据。语法格式为 vpi_get_userdata(obj)。

(5) vpi_put_userdata()将用户数据放置到系统任务/函数实例的存储位置。语法格式为 vpi_put_userdata(obj, userdata)。其中,obj 是指向系统任务或系统函数的句柄,userdata 代表要和系统任务或系统函数相关联的用户数据。函数的返回值为 1,出错时返回值为 0。

(6) vpi_flush()的作用是将仿真器输出缓冲区和 log 文件输出缓冲区清空。

习题 5

5.1 阻塞赋值和非阻塞赋值有什么区别?

5.2 用持续赋值语句描述一个 4 选 1 数据选择器。

5.3 用行为语句设计一个 8 位计数器,每次在时钟的上升沿,计数器加 1,当计数器溢出时,自动从零开始重新计数,计数器有同步复位端。

5.4 设计一个 4 位移位寄存器。

5.5 initial 语句与 always 语句的区别是什么?

5.6 分别用任务和函数描述一个 4 选 1 多路选择器。

5.7 总结任务和函数的区别。

5.8 在 Verilog 中,哪些操作是并发执行的?哪些操作是顺序执行的?

5.9 试编写求补码的 Verilog 程序,输入是带符号的 8 位二进制数。

5.10 试编写两个 4 位二进制数相减的 Verilog 程序。

5.11 有一个比较电路,当输入的一位 8421BCD 码大于 4 时,输出为 1,否则为 0,试编写出 Verilog 程序。

5.12 用 Verilog 语言设计一个类似 74138 的译码器电路,用 Vivado 软件对设计文件进行综合,观察综合视图。

5.13 用 Verilog 语言设计一个 8 位加法器,用 Vivado 软件进行综合和仿真。