第 5 章

Verilog行为描述

Verilog 提供了行为描述能力，具有丰富的行为语句，适合描述复杂的时序关系，实现复杂逻辑功能的设计。

5.1 行为级建模

所谓行为级建模，或称行为描述，是对设计实体的数学模型的描述，其抽象程度高于结构描述。行为描述类似于高级编程语言，当描述一个设计实体的行为时，无须知道其内部电路构成，只需描述清楚输入与输出信号的行为。

Verilog HDL 行为级建模是基于过程实现的，过程包含以下 4 种。

(1) initial

(2) always

(3) task

(4) function

用 initial 过程和 always 过程实现行为级建模，每个 initial 过程块和 always 过程块都执行一个行为流，initial 过程块中的语句只执行一次，always 过程则不断重复执行。例如：

```
module behave;
reg [1:0] a, b;
initial  begin  a = 'b1;  b = 'b0; end
always   begin  #50  a = ~a;  end
always   begin  #100 b = ~b;  end
endmodule
```

上例中，initial 语句为变量 a 和变量 b 分别赋初始值 1 和 0，之后 initial 语句不再执行；always 语句则重复执行 begin-end 块(也称顺序块)，因此，每隔 50 个时间单位，变量 a 取反，每隔 100 个时间单位，变量 b 取反。

一个模块中可以包含多个 initial 语句和 always 语句，但两种语句不能嵌套使用。

5.1.1 always 过程语句

always 过程使用模板如下。

```
always @(<敏感信号列表 sensitivity list>)
begin
   //过程赋值
   //if-else,case 选择语句
   //while,repeat,for 循环语句
   //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，仅当触发条件满足时，其后的begin-end块语句才被执行。

在例5.1中，posedge clk表示将时钟信号clk的上升沿作为触发条件，而negedge clr表示将clr信号的下降沿作为触发条件。

例5.1 同步置数、异步清零的计数器。

```
module count(
   input load,clk,clr,
   input[7:0] data,
   output reg[7:0] out);
always @(posedge clk or negedge clr)          //clk上升沿或clr下降沿触发
begin
   if(!clr)out <= 8'h00;                      //异步清零,低电平有效
   else if(load)   out <= data;               //同步预置
   else out <= out + 1;                       //计数
end
endmodule
```

注意：在例5.1中，clr信号下降沿到来时清零，故低电平清零有效。如果需要高电平清零有效，则应将clr信号上升沿作为敏感信号：

```
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
```

过程体内的描述应与敏感信号列表在逻辑上一致，比如下面的描述是错误的：

```
always @(posedge clk or negedge clr)  begin
   if(clr) out <= 0;    //与敏感信号列表中clr下降沿触发矛盾,应改为if(!clr)
   else out <= out + 1;  end
```

例5.2给出的是一个指令译码电路的示例，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，这是一个组合逻辑电路，如果采用assign语句描述，则表达比较烦琐。本例采用always过程和case语句表达，可使设计思路得到直观体现。

例5.2 指令译码电路示例。

```
`define add       3'd0
`define minus     3'd1
`define band      3'd2
`define bor       3'd3
`define bnot      3'd4
module alu(
   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,输出不定态
    endcase   end
endmodule
```

5.1.2 initial 过程

initial 过程的使用格式如下。

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

initial 语句不带触发条件,过程中的块语句沿时间轴只执行一次。

注意:initial 语句是可以综合的,只不过不能添加时序控制语句,因此作用有限,一般只用于变量的初始化。

下面的测试模块用 initial 语句完成对测试变量 a、b、c 的赋值。

```
`timescale 1ns/1ns
module test1;
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
```

下面的代码用 initial 语句对 memory 存储器进行初始化,将所有存储单元的初始值都置为 0,存储器不能整体赋值,只能每个单元(元素)分别赋值。

```
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 过程时序控制

Verilog HDL 提供了两种时序控制方法,用于激活过程语句的执行:延时控制(用#表示)和事件控制(用@表示)。

5.2.1 延时控制

延时的表示方式如下。

```
# 10   rega = regb;                    //一般延时
rega = # 10 regb;                     //内嵌延时
#d rega = regb;                       //用参数表示延时
#((d+e)/2) rega = regb;               //用参数表示延时
```

例 5.3 中,使用了多种方式表示延时。

例 5.3 延时控制示例。

```
`timescale 1ns/1ns
module test;
reg a;
parameter DELY  = 10;                 //定义参数
initial  begin a=0;                   //0ns,a=0
   #5          a=1;                   //5ns,a=1,一般延时表示
   #DELY       a=0;                   //15ns,a=0,用参数表示延时
   #(DELY/2)   a=1;                   //20ns,a=1
   a= #10      0;                     //30ns,a=0,内嵌延时表示
   a= #5       1;                     //35ns,a=1
   #10   $finish;   end
endmodule
```

用 ModelSim 运行例 5.3,输出波形图如图 5.1 所示。

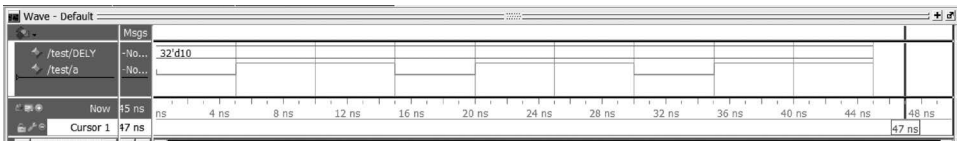


图 5.1 延时控制示例输出波形图

5.2.2 事件控制

在 Verilog HDL 中,事件是指某个 reg 型或 wire 型变量的值发生了变化。

事件控制可用如下格式表示。

```
@(event_expression)          //event_expression 可以是边沿、电平和命名事件
```

1. 一般事件控制

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。

关键字 posedge 是指从 0 到 X、Z、1,以及从 X、Z 到 1 的正跳变(上升沿);negedge 是指从 1 到 X、Z、0,以及从 X、Z 到 0 的负跳变(下降沿),如表 5.1 所示。

表 5.1 关键字 posedge 和 negedge 说明

posedge(正跳变)	negedge(负跳变)	posedge(正跳变)	negedge(负跳变)
0→X	1→X	X→1	X→0
0→Z	1→Z	Z→1	Z→0
0→1	1→0		

以下是边沿触发的示例。

```
@(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
```

对于组合电路,事件通常是输入变量的值发生了变化,可表示如下。

```
@(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
```

2. 命名事件

用户可以声明 event 类型的变量,并触发该变量,识别该事件是否发生。命名事件用关键字 event 声明,触发信号用"->"表示,见例 5.4。

例 5.4 命名事件触发。

```
`timescale 1ns/1ns
module tb_evt;
   event a_event;                   //声明 event 类型的变量
   event b_event;                   //声明 event 类型的变量
initial begin
    #20 -> a_event;
    #30 -> a_event;
    #50 -> a_event;
    #10 -> b_event;
end
always @(a_event) $display("T= %0t [always] a_event is triggered", $time);
initial begin
  #25 @(a_event) $display("T= %0t [initial] a_event is triggered", $time);
  #10 @(b_event) $display("T= %0t [initial] b_event is triggered", $time);
end
endmodule
```

用 ModelSim 运行例 5.4,输出如下。

```
# T= 20 [always] a_event is triggered
# T= 50 [always] a_event is triggered
# T= 50 [initial] a_event is triggered
# T= 100 [always] a_event is triggered
# T= 110 [initial] b_event is triggered
```

3. 敏感信号列表

当多个信号或事件中任一个发生变化都能触发语句的执行时,Verilog HDL 用关键字 or 连接多个事件或信号,这些事件或信号组成的列表称为"敏感列表",也可以用逗号","代替 or。示例如下。

```
always @(a, b, c, d, e)             //用逗号分隔敏感信号
always @(posedge clk, negedge rstn) //用逗号分隔敏感信号
always @(a or b, c, d or e)         //or 和逗号混用,分隔敏感信号
```

在RTL的设计中,经常需要在敏感信号列表中列出所有的输入信号,Verilog-2001中,采用隐式事件表达式解决此问题,采用隐式事件表达式后,综合器会自动从过程块中读取所有的net和variable型输入变量并添加到事件表达式中,这也解决了容易漏写输入变量的问题。

隐式事件表达式可采用下面两种形式之一。

```
always @ *                                  //形式1
always @( * )                               //形式2
```

比如:

```
always @( * )                               //等同于 @(a or b or c or d or f)
   y = (a & b) | (c & d) | myfunction(f);
always @ *                                  //等同于 @(a or b or c or d or tmp1 or tmp2)
begin  tmp1 = a & b;
       tmp2 = c & d;
       y = tmp1 | tmp2;
end
```

4. 电平敏感事件控制

Verilog还支持将电平作为敏感信号来控制时序,即后面语句的执行需要等待某个条件为真,并使用关键字wait表示这种电平敏感情况。示例如下。

```
begin
wait (!enable) #10 a = b;
               #10 c = d;
end
```

如果enable的值为1,则wait语句将延迟语句(#10 a = b;)的计算,直到enable的值变为0。如果进入begin-end块时enable已经为0,会立刻执行(#10 a = b;)语句。

5.3 过程赋值

过程赋值必须置于always、initial、task和function过程内,属于“激活”类型的赋值,用于为reg、integer、time、real、realtime和存储器等数据类型的对象赋值。

5.3.1 variable型变量声明时赋值

在声明variable型变量时可以为其赋初值,可将其看作过程赋值的一种特殊情况,variable型变量会保持该值,直到遇到对该变量的下一条赋值语句。数组不支持在声明时赋值。示例如下。

```
reg[3:0] a = 4'h4;
```

上面的语句等同于:

```
reg[3:0] a;
```

```
initial a = 4'h4;
```

以下是声明变量时赋值的一些示例。

```
integer i = 0, j;
real r1 = 2.5, n300k = 3E6;
time t1 = 25;
realtime rt1 = 2.5;
```

5.3.2 阻塞过程赋值

Verilog HDL 包含以下两种类型的过程赋值语句。

(1) 阻塞过程赋值语句。

(2) 非阻塞过程赋值语句。

阻塞过程赋值语句和非阻塞过程赋值语句在顺序块中指定不同的过程流。

阻塞过程赋值符号为“=”(与连续赋值符号相同),示例如下。

```
b = a;
```

阻塞过程赋值在该语句结束时立即完成赋值操作,即 b 的值在该条语句结束后立刻改变。如果一个 begin-end 块中有多条阻塞过程赋值语句,那么在前面的赋值语句完成之前,后面的语句不能执行,仿佛被阻塞了一样,因此称为阻塞过程赋值。

阻塞过程赋值的示例如下。

```
rega = 0;
rega[3] = 1;                          //位选
rega[3:5] = 7;                        //段选
mema[address] = 8'hff;                //为存储器单元赋值
{carry, acc} = rega + regb;           //位拼接赋值
```

5.3.3 非阻塞过程赋值

非阻塞过程赋值的符号为“<=”(与关系操作符中的小于或等于号相同)。

```
b <= a;
```

非阻塞过程赋值可以在同一时间为多个变量赋值,而无须考虑语句顺序或相互依赖性,非阻塞过程赋值语句是并发执行的(相互间无依赖关系),故其书写顺序对执行结果无影响。

例 5.5 是非阻塞过程赋值的示例。

例 5.5 非阻塞过程赋值的示例。

```
`timescale 1ns/1ns
module evaluate;
reg a, b, c;
initial begin  a = 0;b = 1;c = 0;  end
```

```
always c = #5  ~c;
always @(posedge c)  begin
   a <= b;
   b <= a;
#100 $finish;  end
endmodule
```

上例的执行结果如图 5.2 所示。

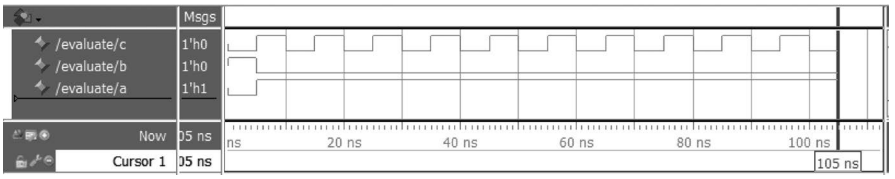


图 5.2　非阻塞过程赋值的执行结果

5.3.4　阻塞过程赋值与非阻塞过程赋值的区别

由例 5.6 可以看出阻塞过程赋值和非阻塞过程赋值的区别。

例 5.6　阻塞过程赋值和非阻塞过程赋值的区别。

```
`timescale 1ns/1ns
module non_block;
reg a, b, c, d, e, f;
initial begin                          //阻塞过程赋值
   a = #10 1;                          //在时刻 10,a 赋值为 1
   b = #6 0;                           //在时刻 16,b 赋值为 0
   c = #8 1;                           //在时刻 24,c 赋值为 1
end
initial begin                          //非阻塞过程赋值
   d <= #10 1;                         //在时刻 10,d 赋值为 1
   e <= #6 0;                          //在时刻 6, e 赋值为 0
   f <= #8 1;                          //在时刻 8, f 赋值为 1
#30 $finish;
end
endmodule
```

例 5.6 的执行结果如图 5.3 所示，可看出非阻塞过程赋值语句的执行均是从时刻 0 开始的，各语句的延时也是从时刻 0 开始计算的；而阻塞过程赋值各语句是按顺序执行的，各条语句的延时是从上条语句执行完开始计算的。

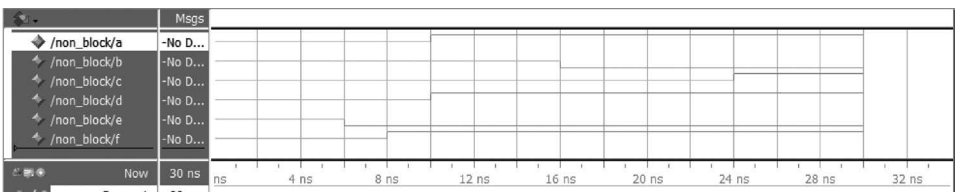


图 5.3　例 5.6 的执行结果

在如下代码中，一个 begin-end 块中同时存在阻塞过程赋值和非阻塞过程赋值。

```
module non_block1;
```

```
reg a, b;
initial begin
   a = 0;
   b = 1;
   a <= b;
   b <= a; end
initial begin   $monitor ( $time, ,"a =  %b b =  %b", a, b);
   #100 $finish; end
endmodule
```

上面代码的执行结果如下。

```
a = 1
b = 0
```

根据阻塞过程赋值和非阻塞过程赋值的特点,得到这样的结果也不难理解。

例 5.7 显示了如何将 i[0]的值赋给 r1,以及如何在每次延时后进行赋值操作。

例 5.7 非阻塞过程赋值。

```
module multiple;
reg r1;
reg [2:0] i;
initial  begin
   for (i = 0; i <= 6; i = i+1)
   r1 <= # (i*10) i[0];                    //赋值给 r1,而不取消以前的赋值
end
endmodule
```

运行例 5.7 后,r1 的波形图如图 5.4 所示。

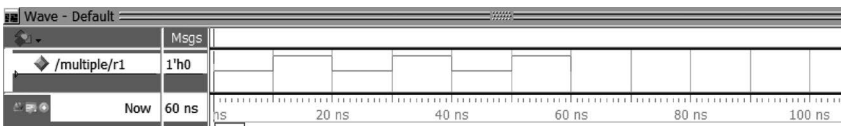


图 5.4 r1 的波形图

例 5.8 也说明了非阻塞过程赋值与阻塞过程赋值的区别。

例 5.8 非阻塞过程赋值与阻塞过程赋值的区别。

```
//非阻塞过程赋值模块
module non_block2(
    input clk,a,
    output reg c,b);
always @(posedge clk)
    begin
    b <= a;
    c <= b;
    end
endmodule
```

```
//阻塞过程赋值模块
module block2(
    input clk,a,
    output reg c,b);
always @(posedge clk)
    begin
    b = a;
    c = b;
    end
endmodule
```

将上面两段代码综合,结果分别如图 5.5 和图 5.6 所示。

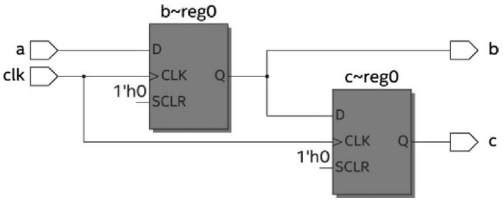


图 5.5　非阻塞过程赋值综合结果

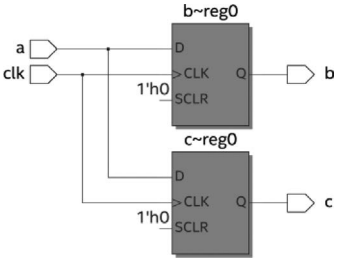


图 5.6　阻塞过程赋值综合结果

5.4　过程连续赋值

过程连续赋值是在过程中对 net 和 variable 数据类型进行连续赋值。过程连续赋值语句比普通的过程赋值语句优先级更高,可以改写所有其他语句的赋值。过程连续赋值能连续驱动赋值对象,即过程连续赋值发生作用时,其右端表达式中任意操作数的变化都会引起过程连续赋值语句的重新执行和响应。

过程连续赋值主要有两种:assign 和 deassign;force 和 release。

5.4.1　assign 和 deassign

assign(过程连续赋值操作)与 deassign(取消过程连续赋值操作)的赋值对象只能是 variable 型变量,不能是 net 型变量。

赋值过程中对 variable 型变量进行连续赋值,该值将保持到被重新赋值。

带异步复位和置位端的 D 触发器可以用 assign 与 deassign 描述,见例 5.9。

例 5.9　用 assign 与 deassign 描述带异步复位和置位端的 D 触发器。

```
module dff_assign(
   input d, clock,
   input clear, preset,
   output reg q);
always @(clear or preset)
   if(!clear)   assign q = 0;                    //assign 语句赋值 0
   else if(!preset)   assign q = 1;              //assign 语句赋值 1
   else   deassign q;                            //q 被 deassign 语句取消赋值
always @(posedge clock)
   q = d;
endmodule
```

上例中,当 clear 端或 preset 端为 0 时,通过 assign 语句分别对 q 端置 0、置 1,此时时钟边沿对 q 端输出不再产生影响;这一状态一直持续到 clear 端和 preset 端均不为 0 时,此时执行一条 deassign 释放语句,结束对 q 端的强行控制,正常的过程赋值语句又重新起作用。

注意:多数综合器不支持 assign 与 deassign,它们多用于仿真。

5.4.2　force 和 release

force(强制赋值)与 release(取消强制赋值)也是过程连续赋值语句,其使用方法和效

果与 assign、deassign 类似，但赋值对象可以是 variable 型变量，也可以是 net 型变量。

因为是无条件强制赋值，多用于交互式调试过程，避免在设计模块中使用。

当 force 作用于 variable 型变量时，该变量当前值被覆盖；release 作用时该变量继续保持强制赋值的值，之后其值可被原有的过程赋值语句改变。

当 force 作用于 net 型变量时，该变量也会被强制赋值；一旦 release 作用于该变量，其值马上变为原值，具体见例 5.10。

例 5.10 用 force 与 release 赋值。

```
`timescale 1ns/1ns
module test_force;
reg a, b, c, d;
wire e;
and g1(e, a, b, c);
initial begin
 $monitor("%d d=%b,e=%b", $stime, d, e);
assign d = a & b & c;
       a = 1; b = 0; c = 1;
   #10; force d = (a | b | c);                    //用 force 强制赋值
        force e = (a | b | c);
   #10; release d;                                //用 release 取消强制赋值
        release e;
   #10  $finish;
end
endmodule
```

例 5.10 的运行结果如下。

```
0   d=0,e=0
10  d=1,e=1
20  d=0,e=0
```

5.5 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，块标识符可省略。

5.5.1 begin-end 串行块

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
```

上面的语句最后将 regb、regc 的值都更新为 rega 的值。

仿真时，begin-end 串行块中每条语句前面的延时都是从前一条语句执行结束时计算的。例如，例 5.11 用 begin-end 串行块产生一段周期为 10 个时间单位的信号波形。

例 5.11 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
```

上例用 ModelSim 仿真后，波形如图 5.7 所示，信号周期为 10 个时间单位(100ns)。

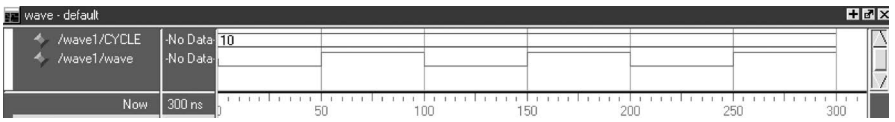


图 5.7　例 5.11 输出的波形

5.5.2　fork-join 并行块

fork-join 并行块中的所有语句都是并发执行的。示例如下。

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

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，故执行后，regb 与 regc 的值是不同的。

仿真时，fork-join 并行块中每条语句前面的延时都是相对于该并行块的起始执行时间的，即起始时间对于块内所有的语句是相同的。要用 fork-join 并行块产生一段与例 5.11 相同的信号波形，应该像例 5.12 中一样标注延时。

例 5.12　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
```

上面的代码用 ModelSim 仿真后，可得到与图 5.7 相同的波形。

5.5.3 块命名

可以给块语句命名，只需将名字加在 begin、fork 关键字后面即可。

块命名的作用有如下几点。

(1) 可在块内定义局部变量，该变量只在块内有效。

(2) 可用 disable 语句终止该命名块的执行，并执行其后的语句。

(3) 可通过层次路径名对命名块内的任一变量进行访问。

例如：

```
begin : break
for (i = 0; i < n; i = i + 1) begin : continue
@ clk
   if(a == 0) disable continue;          //终止 continue 循环
statements
@clk
   if(a == b) disable break;             //终止 break 循环
statements
end  end
```

再如：

```
module  tb;
initial
begin : block1              //名字为 block1 的顺序命名块
  integer n;                //n 是本地变量，可通过层次路径名 tb.block1.n 被其他模块访问
  ...  end
initial
fork : block2               //名字为 block2 的并行命名块
  reg n;                    //n 是本地变量，可通过层次路径名 tb.block2.n 被其他模块访问
  ...
join
```

disable 语句提供了一种终止命名块执行的方法。例 5.13 是用 disable 语句终止命名块的示例，从寄存器的最低有效位开始寻找第一个值为 1 的位，找到该位后，用 disable 语句终止命名块的执行，并输出该比特位的位置。

例 5.13 用 disable 语句终止命名块的示例。

```
`timescale 1ns/1ns
module nameblock_tb;
reg [15:0] flag;
integer i;                  //用于计数的整数
initial  begin
  flag = 16'b 0001_0100_0000_0000;
  i = 0;
  begin: detect_1           //块命名为 detect_1
```

```
  while(i < 16)
   begin
   if(flag[i])              //从 flag 寄存器的最低有效位开始寻找第一个值为 1 的位
   begin
      $ display("Detect a bit 1 at element number % d", i);
   disable detect_1;        //在寄存器中找到值为 1 的位,则终止 detect_1 命名块的执行
   end
   i = i + 1;
   end  end  end
endmodule
```

用 ModelSim 运行上例后,输出如下,表示在第 10 位处发现第一个值为 1 的位。

```
Detect a bit 1 at element number         10
```

5.6 条件语句

Verilog HDL 行为级建模有赖于行为语句,这些行为语句如表 5.2 所示。表中的过程语句、块语句、赋值语句前面已介绍,本节着重介绍条件语句。

表 5.2 Verilog HDL 的行为语句

类 别	语 句	可综合性
过程语句	initial	√
	always	√
	task, function	—
块语句	begin-end 串行块	√
	fork-join 并行块	—
赋值语句	连续赋值 assign	√
	过程赋值=、<=	√
	过程连续赋值: assign, deassign; force, release	—
条件语句	if-else	√
	case	√
循环语句	for	√
	repeat	—
	while	—
	forever	—

条件语句有 if-else 和 case 语句两种,都属于顺序语句,应放在过程语句内使用。

5.6.1 if-else 语句

if 语句的格式与 C 语言中 if-else 语句的格式类似,其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 串行块语句括起来。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 串行块语句括起来。

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,否则执行语句 2。例如,例 5.14 展示了用二重选择 if 语句描述的三态非门。

例 5.14 用二重选择 if 语句描述的三态非门。

```
module tri_not(
    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 5.15 是用多重选择 if 语句描述的 1 位二进制数比较器。

例 5.15 用多重选择 if 语句描述的 1 位二进制数比较器。

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

例 5.16 是用多重选择 if 语句实现的模 60 的 8421BCD 码加法计数器。

例 5.16 模 60 的 8421BCD 码加法计数器。

```
module count60bcd(
    input load,clk,reset,
    input[7:0] data,
    output reg[7:0] qout,
    output cout);
always @(posedge clk)   begin
    if(!reset)   qout<=0;                          //同步复位
    else if(load==1'b0) qout<=data;                //同步置数
    else if((qout[7:4] == 5)&&(qout[3:0] == 9))   qout<= 0;
                                                   //计数达到 59 时,输出清零
```

```
    else if(qout[3:0] == 4'b1001)            //低位达到 9 时,低位清零,高位加 1
      begin
      qout[3:0] <= 0;
      qout[7:4] <= qout[7:4] + 1; end
    else  begin                              //否则高位不变,低位加 1
      qout[7:4] <= qout[7:4];
      qout[3:0] <= qout[3:0] + 1'b1; end
end
assign cout = (qout == 8'h59)?1:0;           //产生进位输出信号
endmodule
```

例 5.17 是模 60 的加法计数器的 Test Bench 测试代码。

例 5.17 模 60 的 8421BCD 码加法计数器的 Test Bench 测试代码。

```
`timescale 1ns/1ns
module count60bcd_tb;
parameter PERIOD = 20;                   //定义时钟周期为 20ns
reg clk,rst,load;
reg[7:0] data = 8'b01010100;             //置数端为 54
wire[7:0] qout;
wire cout;
initial begin clk = 0;
   forever begin #(PERIOD/2) clk = ~clk; end
end
initial begin
   rst <= 0; load <= 1;                  //复位信号
   repeat(2) @(posedge clk);
   rst <= 1;
   repeat(5) @(negedge clk);
   load <= 0;                            //置数信号
   @(negedge clk);
   load <= 1;
   #(PERIOD * 100) $stop;  end
count60bcd i1(.reset(rst), .clk(clk), .load(load),
              .data(data), .qout(qout), .cout(cout));
endmodule
```

在 ModelSim 中运行例 5.17,得到图 5.8 所示的仿真波形,表明功能正确。

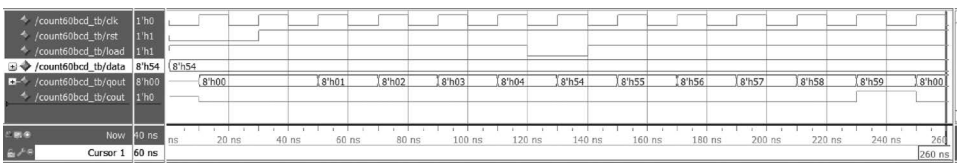


图 5.8 模 60 的 8421BCD 码加法计数器的仿真波形

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;
   …
```

5.6.2 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
```

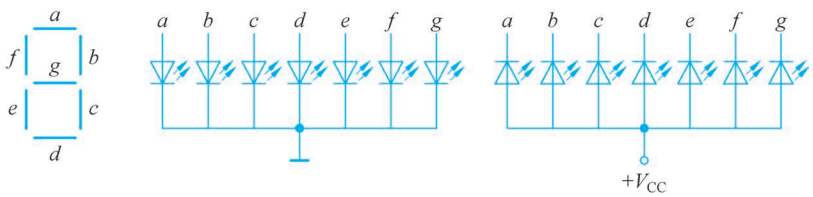
当敏感表达式的值为 1 时，执行语句 1；值为 2 时，执行语句 2；以此类推；若敏感表达式的值与上面列出的值都不相符，则执行 default 后面的语句 $n+1$。若前面已列出了敏感表达式所有可能的取值，则 default 语句可省略。

下例是用 case 语句描述的三人表决电路。

例 5.18 用 case 语句描述的三人表决电路。

```
module vote3(
   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;        //表决未通过
   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
```

例 5.19 是用 case 语句编写的 BCD 码-7 段数码管译码电路，实现 4 位 8421 码到 7 段数码管显示译码的功能。7 段数码管由 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)，用于显示字母、数字。图 5.9 是 7 段数码管的结构与共阴极、共阳极两种连接方式的示意图。假定采用共阴极连接方式，用 7 段数码管显示 0～9 十个数字，则相应译码电路的 Verilog 描述如例 5.19 所示。



(a) 7段数据管结构　(b) 共阴极连接　(c) 共阳极连接

图 5.9 7 段数码管

例 5.19 BCD 码-7 段数码管译码电路。

```
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 @ *   begin                                 //使用通配符
   case({D3,D2,D1,D0})                             //译码,共阴极连接
   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
```

下例是用 case 语句描述的下降沿触发的 JK 触发器。

例 5.20 带异步清零/异步置 1(低电平有效)下降沿触发的 JK 触发器(74112)。

```
module jk_ff
   (input clk,j,k,pr,clr,
   output reg q,qn);
always @(negedge clk, negedge clr, negedge pr)  begin
   if(!pr) begin q <= 1'b1;qn <= 1'b0; end          //异步置 1
   else if(!clr)  begin q <= 1'b0;qn <= 1'b1; end   //异步清零
   else case({j,k})
   2'b00: begin q <=  q; qn <= qn; end              //保持
   2'b01: begin q <= 1'b0;qn <= 1'b1; end           //置 0
   2'b10: begin q <= 1'b1;qn <= 1'b0; end           //置 1
   2'b11: begin q <= ~q; qn <= ~qn; end             //翻转
   default:begin q <= 1'bx;qn <= 1'bx; end
   endcase
end
endmodule
```

由例 5.20 可以看出,用 case 语句描述实际上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。

5.6.3 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态 z,就不必考虑这些位的比较,因此只需关注其他位的比较结果。而在 casex 语句中,则将这种处理方式进一步扩展到对 x 的处理。即如果比较的双方中有一方某些位的值是 x 或 z,就不必考虑这些位的比较。

表 5.3 给出了 case、casez 和 casex 语句的规则比较。

表 5.3 case、casez 和 casex 语句的规则比较

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“?”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,则有 out = 1
casez(a)
3'b1??:out = 1;             //如果 a = 100、101、110、111 或 1xx、1zz 等,则有 out = 1
3'b01?:out = 1;             //如果 a = 010、011、01x、01z,则有 out = 1
```

例 5.21 是用 casez 语句及符号“?”描述的数据选择器的示例。

例 5.21 用 casez 语句及符号“?”描述的数据选择器。

```
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;         //无须再加 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，会引入一个触发器保持原值，在设计组合电路时，应避免这种隐含触发器的存在。当然，在很多情况下，不可能列出所有分支，因为每个变量至少有 4 种取值：0、1、z、x。为了包含所有分支，可在 if 语句后加上 else 语句；在 case 语句后加上 default 语句。

例 5.22 是一个隐含锁存器的示例。

例 5.22 隐含锁存器示例。

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

此例原意是描述 2 输入与门，由于省略了 else 语句，综合时会默认 else 语句为“c=

c；”，因此会形成一个隐含锁存器，其综合结果如图 5.10 所示。如需改为 2 输入与门功能，只需加上“else c=0;”语句即可。

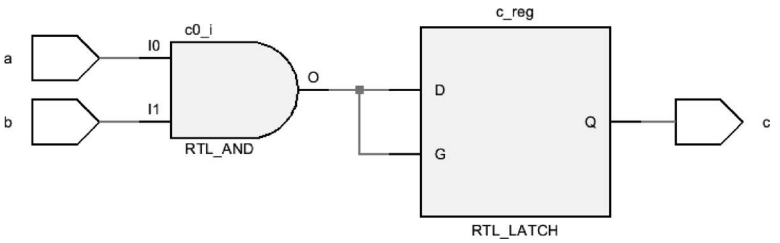


图 5.10　隐含锁存器综合结果

5.7　循环语句

Verilog HDL 有 4 种类型的循环语句，用于控制语句的执行次数。

(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；多在 initial 块中使用，用于生成时钟等周期性波形。

5.7.1　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;
```

例 5.23 通过 7 人投票表决器示例说明 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，则表决通过。用 vote[7:1]表示 7 人的投票情况，1 代表赞成，即 vote[i]为 1 表示第 i 个人赞成，pass=1 表示表决通过。

例 5.23　7 人投票表决器。

```
module vote7(
   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 5.24　用 for 语句实现两个 8 位二进制数相乘。

```
module mult_for   #(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 5.25　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 5.25 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 5.11 所示。如果将变量 y 的初值改为 0，则上例变为偶校验电路。

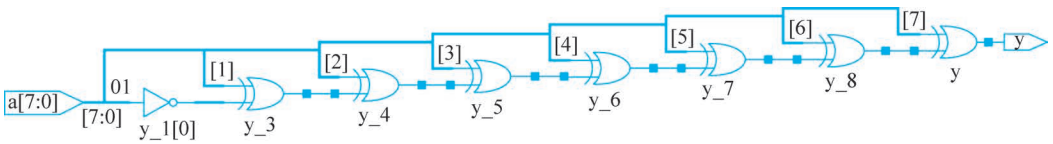


图 5.11　奇校验电路 RTL 综合结果

注意：大多数综合器支持 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。

5.7.2　repeat、while 和 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 5.26 用 repeat 语句和移位操作符实现两个 8 位二进制数的乘法。

例 5.26　用 repeat 语句和移位操作符实现两个 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
```

执行 while 语句时,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,然后判断条件表达式是否为真,若为真,再执行一遍后面的语句,如此重复,直至循环执行条件表达式不为真。因此,在执行语句中必须有一条改变条件表达式值的语句。

在下面的代码中,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
```

下面的示例分别用 repeat 和 while 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 软件运行上面的代码，其输出结果如下。

```
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，常用于产生周期性波形。forever 语句多用于仿真模块的 initial 过程，可以用 disable 语句中断循环，也可以用系统函数 $ finish 退出 forever 循环。

下面的代码是用 forever 语句产生时钟信号，其产生的时钟波形如图 5.12 所示。

```
`timescale 1ns/1ns
module loopf;
reg clk;
initial begin
   clk = 0;
   forever begin
   clk = ~clk;   #5; end
end
endmodule
```

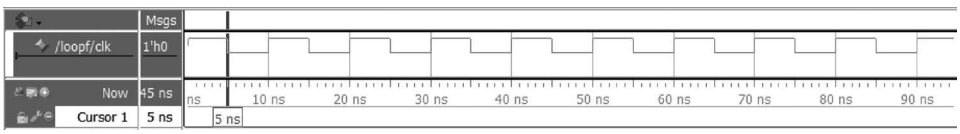


图 5.12　时钟信号波形图

5.8 任务

5.8.1 任务概述

任务的定义方式如下。

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

任务调用的格式如下。

```
<任务名> (端口 1,端口 2,...);      //任务调用时的端口顺序应与定义时的端口顺序保持一致
```

定义任务可以写为如下两种形式。

```
task sum;          //任务定义形式 1
input[7:0] a, b;
output[7:0] s;
begin s = a + b; end
endtask
```

```
task sum(      //任务定义形式 2
   input[7:0] a, b,
   output[7:0] s);
begin s = a + b; end
endtask
```

调用任务时,可以这样使用。

```
module task_inst(
   input[7:0] x, y,
   output reg[7:0] z);
always@ *   begin
   sum(x, y, z);       //任务调用,变量 x 和 y 的值赋给 a 和 b;任务完成后,s 的值赋给 z
end
endmodule
```

当用综合器综合上面的代码时,应将 task 任务源码置于模块内,不可在模块外定义。任务调用时的端口变量和定义时的端口变量应是一一对应的。

5.8.2 任务示例

例 5.27 用任务实现了交通灯时序控制电路。

例 5.27 用任务实现交通灯时序控制电路。

```
module traffic_lights;
reg clock, red, amber, green;
parameter  on = 1, off = 0, red_tics = 350,
           amber_tics = 30, green_tics = 200;
initial red = off;
initial amber = off;
initial green = off;
always begin                              //控制灯顺序
   red = on;                              //红灯亮
   light(red, red_tics);                  //任务例化
   green = on;                            //绿灯亮
   light(green, green_tics);              //任务例化
   amber = on;                            //黄灯亮
   light(amber, amber_tics);              //任务例化
end
task light;                               //任务定义
output color;
input[31:0] tics;                         //延时
begin
   repeat (tics) @(posedge clock);
   color = off;                           //灯灭
end
endtask
always begin                              //产生时钟波形
   #100 clock = 0;
```

```
    #100 clock = 1;
end
endmodule
```

用 ModelSim 运行上面的代码，交通灯时序控制电路的仿真波形图如图 5.13 所示。

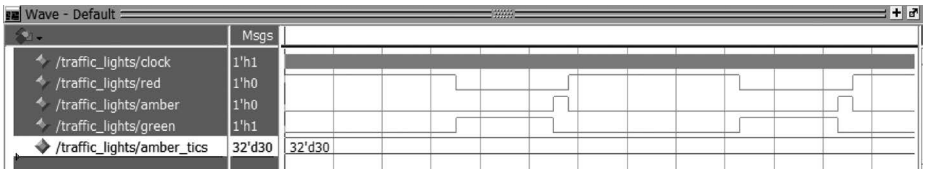


图 5.13　交通灯时序控制电路的仿真波形图

例 5.28 定义了一个实现两个操作数按位与的任务，并在算术逻辑单元中调用该任务。

例 5.28　用任务实现两个操作数按位与。

```
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 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，此处的端口 a,b,c 分别对应任务定义时的 a,b,out */
   2'b01:c = a|b;                     //相或
   2'b10:c = a - b;                   //相减
   2'b11:c = a + b;                   //相加
   endcase
   end
endmodule
```

编写如下激励代码对上例进行验证。

```
`timescale 100ps/1ps
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 运行上面的代码,得到图 5.14 所示的输出波形图。

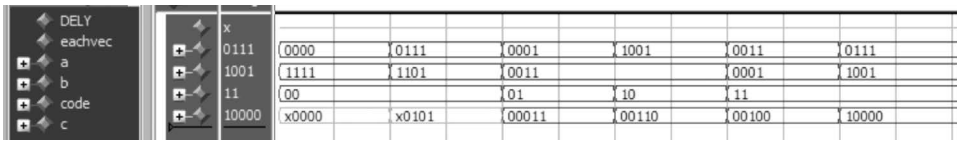


图 5.14　用任务实现按位与的输出波形图

在例 5.29 中用任务实现异或功能。

例 5.29　用任务实现异或功能。

```
module xor_oper
  #(parameter  N = 4)
   (input  clk, rstn ,
   input[N-1:0]  a, b,
   output [N-1:0]  co);
reg[N-1:0]  co_t;
always @( * )  begin
   xor_tsk(a, b, co_t);                    //任务例化
   end
reg[N-1:0]  co_r;
always @(posedge clk or negedge rstn) begin
   if(!rstn) begin  co_r  <= 'b0;  end
   else begin  co_r <= co_t;  end
   end
assign  co = co_r;
/* ------------ task ---------- */
task xor_tsk;
input [N-1:0]  numa;
input [N-1:0]  numb;
output [N-1:0]  numco;
   #3  numco  = numa ^ numb;          //实现异或功能
endtask
endmodule
```

例 5.29 的 RTL 综合视图如图 5.15 所示。

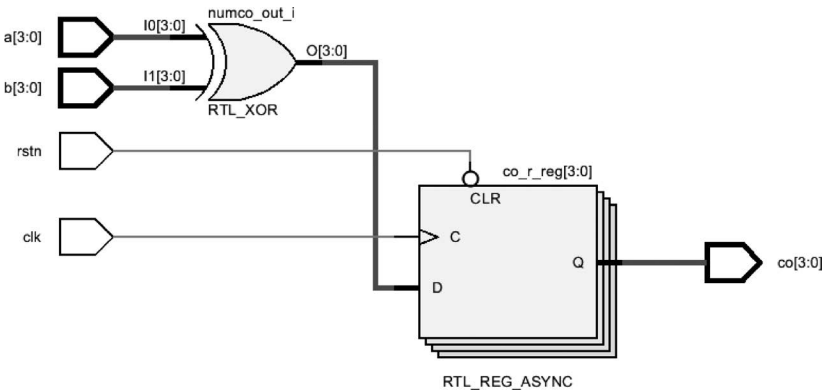


图 5.15　例 5.29 的 RTL 综合视图

注意：在使用任务时，应注意如下几点。

（1）任务的定义与调用必须在一个模块内。

（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，其被激活。任务的调用与模块的调用一样，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一致。

（4）一个任务可以调用别的任务和函数，可调用的任务和函数个数不受限制。

5.9 函数

5.9.1 函数概述

在 Verilog HDL 模块中，如果多次用到重复的代码，可以将这部分代码摘取出来，定义为函数。综合时每调用一次函数，则复制或平铺该电路一次，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。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。

函数调用通常是在表达式中调用函数的返回值，并将函数作为表达式中的一个操作数来实现。函数调用的格式如下。

```
<函数名> ( <表达式> <表达式> );
```

例 5.30 用函数和 case 语句定义了一个 8-3 编码器，并用 assign 语句调用了该函数。

例 5.30 用函数和 case 语句定义的编码器(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
```

例 5.31 用函数实现输入向量从原码到补码的转换,使用 comp2(vect)形式进行调用,其中 vect 是输入的 8 位原码,位宽用参数 N 表示。

例 5.31 用函数实现输入向量从原码到补码的转换。

```
`timescale 1ns/1ns
module comp2_fuct
  #(parameter N = 8)                        //位宽用参数定义
   (input[N-1:0]  vect,
   output[N-1:0]  result);
assign  result = comp2(vect);
//-----------------------------------------------
function [N-1:0] comp2;                     //函数定义
input[N-1:0]  ain;
begin  comp2 = ain[N-1]?{ain[N-1],~ain[N-2:0]+1'b1} : ain; end
endfunction
endmodule
```

例 5.32 是例 5.31 的 Test Bench 测试代码,图 5.16 是其测试输出波形图,说明运算功能正确。

图 5.16 将 8 位原码转换为 8 位补码的测试输出波形图

例 5.32 将 8 位原码转换为 8 位补码的测试代码。

```
`timescale 1ns/1ns
module comp2_fuct_tb;
parameter MSB = 8;
reg[MSB-1:0]  ain;
wire[MSB-1:0]  y_out;
comp2_fuct #(.N(MSB)) u1(.vect(ain),.result(y_out));
initial begin  ain <= 0;
   # 3000   $stop;  end
always #10  ain <= ain + 1;                 //让 ain 变化,遍历逻辑值
endmodule
```

与 C 语言相似,Verilog 使用函数以适应对不同操作数进行同一运算的操作。函数在综合时被转换为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,以得到相应的结果。

例 5.33 用函数实现一个带控制端的实现整数运算的电路,分别实现正整数的平方、立方和阶乘运算。

例 5.33 用函数实现正整数的平方、立方和阶乘运算。

```
module calculate(
   input  clk,clr,
```

```
    input[1:0]   sel,
    input[3:0]   n,
    output reg[31:0]   result);
always @(posedge clk) begin
    if(!clr)   result <= 0;
    else   begin
    case(sel)
    2'd0: result <= square(n);
    2'd1: result <= cubic(n);
    2'd2: result <= factorial(n);                    //调用 factorial 函数
    endcase
end   end
//-------------------------------------------------
function [31:0] square;                              //平方运算函数定义
input[3:0] operand;
begin   square = operand * operand;   end
endfunction
//-------------------------------------------------
function [31:0] cubic;
input[3:0] operand;
begin   cubic = operand * operand * operand;   end
endfunction
//-------------------------------------------------
function [31:0] factorial;                           //阶乘运算函数定义
input[3:0] operand;
integer i;
begin
    factorial = 1;
    for(i = 2; i <= operand; i = i + 1)
    factorial = i * factorial;
end
endfunction
endmodule
```

例 5.34 是例 5.33 的 Test Bench 测试代码，图 5.17 是其测试输出波形图。

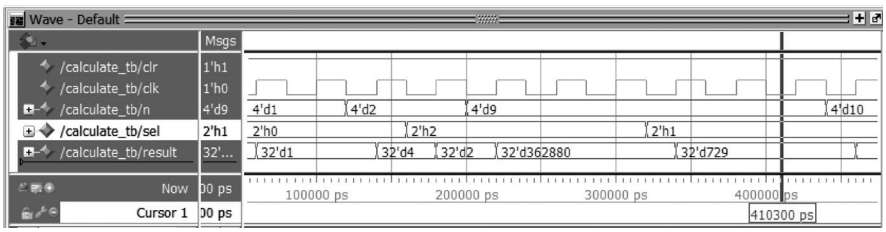


图 5.17　平方、立方和阶乘运算电路的测试输出波形图

例 5.34　平方、立方和阶乘运算电路的 Test Bench 测试代码。

```
`timescale  1ns/100ps
module calculate_tb;
reg[3:0] n;
reg  clr,clk;
reg[1:0] sel;
wire[31:0] result;
parameter CYCLE = 20;
calculate u1(.clk(clk),.n(n),.result(result),.clr(clr),.sel(sel));
initial begin clk = 0;
    forever   # CYCLE clk = ~clk; end                //产生时钟信号
```

```
initial  begin
   {n, clr, sel} <= 0;
   #40 clr = 1;
   repeat(10)
   begin
   @(negedge clk) begin
   n = { $random} % 11;
   @(negedge clk)
   sel = { $random} % 3;
   end end
   #1000 $stop;
end
endmodule
```

注意：函数的定义中蕴含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。

下面的示例定义了 clogb2 函数，该函数完成以 2 为底的对数运算，调用 clogb2 函数，由 RAM 模块的深度换算出所需的地址宽度。

```
module ram_model(address, write, cs, data);
parameter data_width = 8;
parameter ram_depth = 256;
localparam adder_width = clogb2(ram_depth);       //调用 clogb2 函数
input[adder_width - 1:0] address;
input write, cs;
inout [data_width - 1:0] data;
reg[data_width - 1:0] data_store[0:ram_depth - 1];

function integer clogb2(input integer value);      //定义 clogb2 函数
begin
   for(clogb2 = 0; value > 0; clogb2 = clogb2 + 1;)
   value = value >> 1;
end
endfunction
```

注意：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 模块内。

（2）函数只允许有输入变量且必须至少有一个输入变量，输出变量名为函数名本身，函数名须定义数据类型和位宽。调用函数时需列出端口，其排序和类型应与定义时一致，这一点与任务相同。

（3）函数可以出现在持续赋值 assign 的右端表达式中。

（4）函数定义不能包含任何时间控制语句，不能使用#、@或 wait 等符号；函数定义时不能使用非阻塞赋值。

5.9.2 任务和函数的区别

表 5.4 对任务与函数进行了比较。

表 5.4 任务与函数的比较

比 较 项 目	任 务	函 数
输入与输出	可有任意多个多种类型的参数	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	函数不能包含定时和事件控制语句
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

合理使用任务和函数会使程序显得结构清晰，一般的综合器对任务和函数都是支持的，部分综合器不支持任务。

注意：对于任务、函数的差异应注意如下几点。

（1）函数应在一个模拟时间单元中执行；任务可以包含时间控制语句。

（2）函数不能启用任务；任务可以启用其他任务和函数。

（3）函数应至少有一个输入类型参数，且不应有输出或输出类型参数；任务可以有任意类型的零个或多个参数。

（4）函数应返回单个值；任务无返回值。

5.10 automatic 任务和函数

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

5.10.1 automatic 任务

任务本质上是静态的，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，结果可能出错。Verilog-2001 标准中增加了关键字 automatic，使空间可动态分配，任务可重入，若在模块中的多个地方同时调用该任务，则任务可以并发执行。

例 5.35 给出一个静态任务的示例。

例 5.35 静态任务示例。

```
module task_tb;
  integer i = 0;                          //变量 i 在模块中声明
   initial  disp_ask( );                  //任务的调用
   initial  disp_ask( );
   initial  disp_ask( );
task disp_ask( );
begin
   i = i + 1;
   $ display("i =  % 0d",i);
```

```
end
endtask
endmodule
```

用 ModelSim 运行例 5.35,TCL 窗口输出信息如下。

```
i = 1
i = 2
i = 3
```

若在上面的任务定义中增加关键字 automatic,则定义了可重入任务,如例 5.36 所示,则任务的多次调用是并发执行的。

例 5.36 automatic 任务示例。

```
module auto_tb;
   initial  disp_ask( );                    //任务的调用
   initial  disp_ask( );
   initial  disp_ask( );
task automatic disp_ask( );
integer i = 0;                              //变量 i 在任务中声明
begin
   i = i + 1;
   $ display("i =  % 0d",i);
end
endtask
endmodule
```

用 ModelSim 运行例 5.36,TCL 窗口输出信息如下。

```
i = 1
i = 1
i = 1
```

5.10.2 automatic 函数

关键字 automatic 用于函数,表示函数的迭代调用,也称递归函数。

如例 5.33 中的阶乘运算,可采用递归函数实现,如例 5.37 所示,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!)。

比较例 5.33 与例 5.37,可体会函数与递归函数的区别。

例 5.37 实现 32 位无符号整数的阶乘运算。

```
module tryfact;
function automatic integer factorial;              //函数定义
input[31:0] opa;
   if (opa > = 2)
   factorial = factorial(opa - 1) * opa;           //迭代调用
   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
```

上面的 factorial 函数是用 if 语句实现的，也可以用条件操作符写为下面的形式：

```
function automatic  integer factorial;
input integer opa;
integer  i;
   begin
   factorial = (opa >= 2) ? opa * factorial(opa - 1) : 1;
   end
endfunction
```

例 5.37 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
```

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。示例如下。

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

例 5.38 用函数实现一个 16 位数据的高低位转换，将最高有效位转换为最低有效位，次高位转换为次低位，以此类推。

例 5.38　用函数实现数据的高低位转换。

```
`timescale 1ns/1ns
module bit_invert;
reg[7:0] din;
wire[7:0] result;
assign result = invert(din);                       //函数调用
initial begin  din = 8'b00101101;
   # 30 din = 8'b00101111;
   # 20 din = 8'b10001111;
   # 30 $ stop;
end
initial $ monitor( $ time,,,"ain = % b result = % b",din,result);
function automatic unsigned[7:0] invert(
   input[7:0] data);
integer i;
begin
```

```
for (i = 0; i < 8; i = i + 1)
   invert[i] = data[7 - i];
end
endfunction
endmodule
```

用 ModelSim 运行例 5.38,TCL 窗口输出信息如下,可见转换结果正确。

```
0    ain = 00101101   result = 10110100
30   ain = 00101111   result = 11110100
50   ain = 10001111   result = 11110001
```

5.11 编译指令

编译指令语句以符号“`”(该符号 ASCII 码为 0x60)开头,编译时,编译器通常先对这些指令语句进行预处理,然后将预处理的结果和源程序一起编译。Verilog HDL 提供了十余条编译指令,部分如下。

(1) `timescale;

(2) `define,`undef;

(3) `ifdef,`else,`elsif,`endif,`ifndef;

(4) `include;

(5) `default_nettype; `resetall; `celldefine,`endcelldefine;

(6) `unconnected_drive,`nounconnected_drive;

(7) `begin_keywords,`end_keywords; `line,`pragma。

5.11.1 `timescale

`timescale 用于定义时延、仿真的时间单位和时间精度,其使用形式如下。

```
`timescale <time_unit>/<time_precision>
`timescale <时间单位>/<时间精度>
```

用于表示时间单位的符号有 s、ms、μs、ns、ps 和 fs,分别表示秒、10^{-3}s、10^{-6}s、10^{-9}s、10^{-12}s 和 10^{-15}s。时间精度可以和时间单位一样,但是时间精度大小不能超过时间单位大小。例 5.39 给出了它的定义。

例 5.39 `timescale 的定义示例。

```
`timescale 1ns/100ps
module andgate(
   output out,
   input a,b);
and  #(4.34,5.86)  a1(out,a,b);          //门延时定义
endmodule
```

在例 5.39 中,`timescale 指令定义延时以 1ns 为单位,精度为 100ps(精确到 0.1ns),因此,门延时值 4.34 对应 4.3ns,延时值 5.86 对应 5.9ns。如果将`timescale 指令定义为

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

那么延时值 4.34 对应 43ns，5.86 对应 59ns。再如：

```
`timescale 10ns/1ns
module test;
reg set;
parameter d = 1.55;
initial begin
    #d set = 0;                          //16ns(1.6×10)时，set 赋值为 0
    #d set = 1;                          //32ns(1.6×10+1.6×10)时，set 赋值为 1
end
endmodule
```

注意：(1) `timescale 指令在模块说明外部出现，并且影响后面所有的延时值；在编译过程中，`timescale 指令会影响后面所有模块中的时间值，直至遇到另一个`timescale 指令或`resetall 指令。

(2) Verilog HDL 中没有默认的`timescale，如果没有指定`timescale，Verilog HDL 模块就会继承前面编译模块的`timescale 参数。

(3) 如果一个设计中的多个模块都带有`timescale，模拟器总是定位在所有模块的最小延时精度上，并且将所有延时都相应地换算为最小延时精度，延时单位不受影响。

示例如下。

```
`timescale 1ns/1ns
module top;                              //顶层模块
reg  a, b;
wire cout;
initial begin
   a = 1; b = 0;
   # 2.25  a = 0;
   # 5.5   b = 1;  end
andgate g1(cout,a,b);                      //andgate 模块见例 5.39
endmodule
```

在上面的代码中，延时值 2.25 对应 2ns，延时值 5.5 对应 6ns。

但由于子模块 andgate 中定义时间精度为 100ps，故该例中的延时精度变为 100ps。

`timescale 的时间精度设置会影响仿真时间，时间精度越小，仿真时占用内存越多，实际耗用的仿真时间就越长。

$printtimescale 系统任务可用于显示当前的时间单位和时间精度。

5.11.2 `define 和`undef

`define 用于定义宏名，类似于 C 语言中的#define。使用形式如下。

```
`define  宏名  字符串
```

例如：

```
`define WORDSIZE 8
reg[`WORDSIZE:1] data;                    //相当于定义 reg[8:1] data;
//---------------------
`define var_nand(dly) nand #dly           //定义带延时的与非门
`var_nand(2) g1 (q21, n10, n11);
`var_nand(5) g2 (q22, n10, n11);
```

又如：

```
`define max(a,b) ((a) > (b) ? (a) : (b))
n = `max(p + q, r + s);
n = ((p + q) > (r + s)) ? (p + q) : (r + s);     //该语句等同于上面两条语句
```

`undef 用于取消之前的宏定义，示例如下。

```
`define WORDSIZE 16
reg[`WORDSIZE:1] data;
  …
`undef  WORDSIZE
```

注意：使用`define 语句时应注意如下几点。

(1) `define 宏定义语句行末没有分号。

(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的，可以是整个工程。也就是说，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，一般将其放在模块外。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，可通过参数传递改变下层模块中参数的值。

5.11.3 `ifdef、`else、`elsif、`endif 和`ifndef

`ifdef、`else、`elsif、`endif、`ifndef 均属于条件编译指令。

下例中定义了 3 种显示模式，如果定义了 VIDEO_480_272(用`define 语句定义，如`define VIDEO_480_272)，则使用第 1 套参数；如果定义了 VIDEO_640_480，则使用第 2 套参数；否则使用第 3 套参数。

```
`ifdef  VIDEO_480_272                     //480×272 显示模式
   parameter H_ACTIVE = 16'd480;
   parameter V_ACTIVE = 16'd272;
`endif
//-------------------------------------------------
`ifdef  VIDEO_640_480                     //640×480 显示模式
   parameter H_ACTIVE = 16'd640;
   parameter V_ACTIVE = 16'd480;
`endif
//-------------------------------------------------
```

```
`ifdef  VIDEO_800_480                    //800×480 显示模式
   parameter H_ACTIVE = 16'd800;
   parameter V_ACTIVE = 16'd480;
`endif
```

上例中只使用`ifdef 和`endif 组成条件编译指令块，也可以增加`elsif、`else 编译指令，则上面的示例可改为如下形式。

```
`ifdef  VIDEO_480_272                    //480×272 显示模式
   parameter H_ACTIVE = 16'd480;
   parameter V_ACTIVE = 16'd272;
//-----------------------------------------------
`elsif  VIDEO_640_480                    //640×480 显示模式
   parameter H_ACTIVE = 16'd640;
   parameter V_ACTIVE = 16'd480;
//-----------------------------------------------
`else  VIDEO_800_480                     //800×480 显示模式
   parameter H_ACTIVE = 16'd800;
   parameter V_ACTIVE = 16'd480;
`endif
```

可以指定条件编译指令`ifdef、`else 和`endif 仅对程序中的部分内容进行编译，该指令有 3 种使用形式。

第 1 种使用形式如下。

```
`ifdef  宏名
   语句块
`endif
```

这种形式表示：若宏名在程序中被定义(用`define 语句定义)过，则下面的语句块参与源文件的编译；否则，该语句块不参与源文件的编译。

第 2 种使用形式如下。

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

这种形式表示：若宏名在程序中被定义(用`define 语句定义)过，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。

第 3 种使用形式如下。

```
`ifdef  宏名
   语句块 1
`ifdef  语句块 2
`else   语句块 3
`endif
```

例 5.40 给出了`ifdef 的用法。

例 5.40　`ifdef 的用法示例。

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

若在后面的程序中有“`define add”,则执行“assign out=a+b;”操作;若没有该定义语句,则执行“assign out=a-b;”操作。

也可用`ifndef 指令语句设置条件编译,表示如果没有相关的宏定义,则执行相关语句。比如,上面的示例如果使用`ifndef 指令改写,则如例 5.41 所示,这两例的操作是相同的,只是表达方式不同。

例 5.41 `ifndef 用法示例。

```
module compile_ndef(
   input a,b,
   output out);
`ifndef add                                   //`ifndef 指令
   assign out = a - b;
`else  assign out = a + b;
`endif
endmodule
```

5.11.4 `include

使用`include 可以在编译时将一个 Verilog 文件包含到另一个文件中,其格式如下。

```
`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到设计文件中。

`include 用于指定包含其他文件的内容,被包含的文件名必须放在双引号中,被包含的文件既可以使用相对路径,也可以使用绝对路径;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。示例如下。

```
`include  "parts/count.v"
`include  "../../fileA.v"
`include  "fileB"
```

注意: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 命令进行包含;多个`include 命令可以写在一行,但命令行中只可以出现空格和注释,示例如下。

```
`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，须指明其路径。

(3) 文件允许嵌套包含，但限制其数量，最多为 15 个。

习题 5

5-1 用行为描述方式设计带异步复位端和异步置位端的 JK 触发器，并进行综合。

5-2 initial 语句与 always 语句的区别是什么？

5-3 用行为语句设计模 100 加法计数器，且计数器带有同步复位端。

5-4 分别编写 4 位串并转换程序和 4 位并串转换程序。

5-5 用 case 语句描述 4 位双向移位寄存器。74LS194 是 4 位双向移位寄存器，采用 16 引脚双列直插式封装，其引脚排列如图 5.18 所示。74LS194 具有异步清零、数据保持、同步左移、同步右移、同步置数 5 种工作模式。CLR 为异步清零输入，低电平有效，S_1、S_0 为方式控制输入：$S_1S_0=00$ 时，74LS194 工作于保持方式；$S_1S_0=01$ 时，74LS194 工作于右移方式，其中 D_R 为右移数据输入端，Q_3 为右移数据输出端；$S_1S_0=10$ 时，74LS194 工作于左移方式，其中 D_L 为左移数据输入端，Q_0 为左移数据输出端；$S_1S_0=11$ 时，74LS194 工作于同步置数方式，其中 $D_3\sim D_0$ 为并行数据输入端。请用 case 语句描述实现 74LS194 的上述逻辑功能。

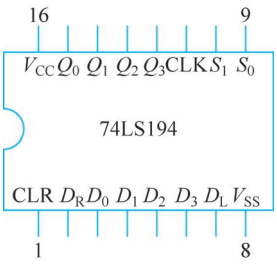


图 5.18 4 位双向移位寄存器 74LS194 引脚排列

5-6 用 if 语句描述四舍五入电路的功能，假定输入的是一位 BCD 码。

5-7 试编写两个 8 位二进制带符号数相减的 Verilog 程序。

5-8 使用 for 循环语句对一个深度为 16(地址从 0～15)、位宽为 8 位的存储器(寄存器类型数组)进行初始化，为所有存储单元赋初始值 0，将存储器命名为 cache。

5-9 任务和函数的不同点有哪些？

5-10 分别用任务和函数描述一个 4 选 1 数据选择器。

5-11 用函数实现一个用 7 段数码管交替显示 26 个英文字母的程序，自定义字符的形状。

5-12 用函数实现一个 16 位数据的高低位转换，将最高有效位转换为最低有效位，次高位转换为次低位，以此类推。

5-13 用任务完成无符号数的大小排序，设 a、b、c、d 是 4 个 8 位无符号数，按从小到大的顺序重新排列并输出到 4 个寄存器中存储。

5-14 编写一个 Verilog 程序，生成偶校验位。输入一个 8 位数据，输出一个包含数据和偶校验位的码字。