

第5章 条件语句、循环语句、块语句与生成语句

5.1 条件语句（if_else语句）

if语句是用来判定所给定的条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。

Verilog HDL语言提供了三种形式的if语句。

(1) if(表达式) 语句

例如:

```
if ( a > b )  
    out1 <= int1;
```

(2) if(表达式)

语句1

else

语句2

例如:

```
if(a>b)  
    out1<=int1;  
else  
    out1<=int2;
```

(3)

```
if(表达式1) 语句1;  
else if(表达式2) 语句2;  
else if(表达式3) 语句3;  
.....  
else if(表达式m) 语句m;  
else 语句n;
```

注意（书上6点）：

在if和else后面可以包含一个内嵌的操作语句(如上例)，也可以有多个操作语句，此时用begin和end这两个关键词将几个语句包含起来成为一个复合块语句。

if语句的嵌套

在if语句中又包含一个或多个if语句称为if语句的嵌套。

应当注意if与else的配对关系，else总是与它上面的最近的if配对。如果if与else的数目不一样，为了实现程序设计者的企图，可以用begin_end块语句来确定配对关系。


```
if(index>0)
    for(scani=0;scani<index;scani=scani+1)
        if(memory[scani]>0)
            begin
                $display("...");
                memory[scani]=0;
            end
    else /*WRONG*/
        $display("error-indexiszero");
```

尽管程序设计者把else写在与第一个if(外层if)同一列上，希望与第一个if对应，但实际上else是与第二个if对应，因为它们相距最近。

正确的写法应当是这样的：

```
if(index>0)
begin
    for(scani=0;scani<index;scani=scani+1)
        if(memory[scani]>0)
            begin
                $display("...");
                memory[scani]=0;
            end
        end
    end
else /*WRONG*/
    $display("error-indexiszero");
```

一位16进制计数器

```
module counter16(q, clk);  
output[3:0] q;  
input clk;  
reg[3:0] q;
```

```
    always @(posedge clk)  
        q<=q+1'b1;
```

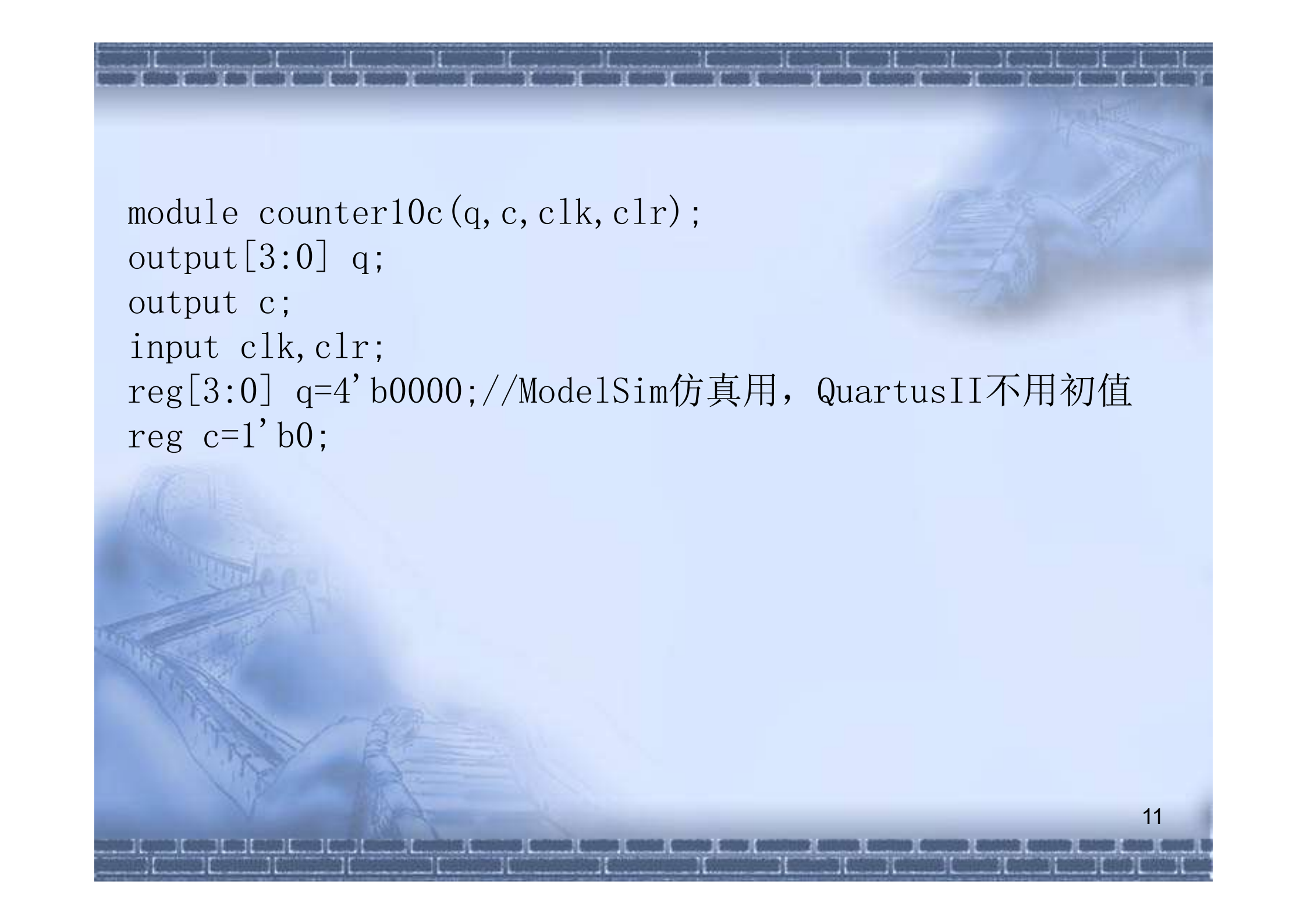
```
endmodule
```


一位16进制计数器，增加清零端

```
module counter16b(q, clk, clr);  
output[3:0] q;  
input clk, clr;  
reg[3:0] q=4'b0000; //ModelSim仿真用，QuartusII不用初  
值
```

```
    always @(posedge clk or posedge clr)  
    begin  
        if(clr)  
            q<=4'b0000;  
        else  
            q<=q+1'b1;  
        end  
    endmodule
```

```
module counter10b(q, clk, clr);  
output[3:0] q;  
input clk, clr;  
reg[3:0] q=4'b0000; //ModelSim仿真用, QuartusII不用初值  
always @(posedge clk or posedge clr)  
begin  
    if(clr)  
        q<=4'b0000;  
    else  
        begin  
            if (q==4'b1001)  
                q<=4'b0000;  
            else  
                q<=q+1'b1;  
            end  
        end  
    end  
endmodule
```



```
module counter10c(q, c, clk, clr);  
output[3:0] q;  
output c;  
input clk, clr;  
reg[3:0] q=4'b0000;//ModelSim仿真用, QuartusII不用初值  
reg c=1'b0;
```

```
always @(posedge clk or posedge clr)
begin
    if(clr)
    begin
        q<=4' b0000;
        c<=1' b0;
    end
    else
```

```
begin
    if (q==4' b1000)
    begin
        q<=q+1' b1;
        c<=1' b1;
    end
    else if (q==4' b1001)
    begin
        q<=4' b0000;
        c<=1' b0;
    end
    else
    begin
        q<=q+1' b1;
        c<=1' b0;
    end
end
end
endmodule
```


case语句

case语句是一种多分支选择语句，if语句只有两个分支可供选择，而实际问题中常常需要用到多分支选择，Verilog语言提供的case语句直接处理多分支选择。

它的一般形式如下：

- 1) case(表达式) <case分支项> endcase
- 2) casez(表达式) <case分支项> endcase
- 3) casex(表达式) <case分支项> endcase

case分支项的一般格式如下：

分支表达式：语句

缺省项(default项)：语句

- (1) case括弧内的表达式称为控制表达式，case分支项中的表达式称为分支表达式。控制表达式通常表示为控制信号的某些位，分支表达式则用这些控制信号的具体状态值来表示，因此分支表达式又可以称为常量表达式。
- (2) 当控制表达式的值与分支表达式的值相等时，就执行分支表达式后面的语句。如果所有的分支表达式的值都没有与控制表达式的值相匹配的，就执行default后面的语句。
- (3) default项可有可无，一个case语句里只准有一个default项。
- (4) case语句各分支表达式间未必是并列互斥关系，允许出现多个分支取值同时满足case表达式的情况。这种情况下将执行最先满足表达式的分支项，然后跳出case语句，不再检测其余分支项目。

(5) 执行完case分项后的语句，则跳出该case语句结构，终止case语句的执行。

(6) 在用case语句表达式进行比较的过程中，只有当信号的对应位的值能明确进行比较时，比较才能成功。因此要注意详细说明case分项的分支表达式的值。

(7) case语句的所有表达式的值的位宽必须相等，只有这样控制表达式和分支表达式才能进行对应位的比较。一个经常犯的错误是用'bx, 'bz 来替代 n'bx, n'bz，这样写是不对的，因为信号x, z的缺省宽度是机器的字节宽度，通常是32位(此处 n 是case控制表达式的位宽)。

```
module test_case;  
reg[1:0] select;  
  initial  
  begin  
    select=2'b01;  
    case (select)  
      2'b01:$display("second 2b01...");  
      2'b01:$display("third 2b01...");  
      2'b01,2'b00:$display("2b01");  
      2'b11:$display("2b11");  
      default:$display("default...");  
    endcase  
  end  
endmodule
```



```
module exb5_1;
reg[1:2] select;
reg[3:0] result;

always @(select)
begin
    case ( select[1:2] )
        2 'b00:  result = 0;
        2 'b01:  result = 4' b0001;
        2 'b0x,
        2 'b0z:  result = 4' b0x0z;
        2 'b10:  result = 4' b0010;
        2 'bx0,
        2 'bz0:  result = 4' bx0z0;
        default: result = 4' bxxxx;
    endcase
end
```



```
initial
begin
    select=2' b00;
    #100;
    select=2' b01;
    #100;
    select=2' b00;
    #100;
    select=2' b0x;
    #100;
    select=2' b00;
    #100;
    select=2' b0z;
    #100;
    select=2' bx0;
    #100;
    select=2' b11;
end
endmodule
```

case语句与if_else_if语句的区别主要有两点:

- 1) 与case语句中的控制表达式和多分支表达式这种比较结构相比, if_else_if结构中的条件表达式更为直观一些。
- 2) 对于那些分支表达式中存在不定值x和高阻值z位时, case语句提供了处理这种情况的手段

Verilog HDL针对电路的特性提供了case语句的其它两种形式用来处理case语句比较过程中的不必考虑的情况(don't care condition)。其中casez语句用来处理不考虑高阻值z的比较过程, casex语句则将高阻值z和不定值都视为不必关心的情况。所谓不必关心的情况, 即在表达式进行比较时, 不将该位的状态考虑在内。这样在case语句表达式进行比较时, 就可以灵活地设置以对信号的某些位进行比较。

case	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

casez	0	1	x	z
0	1	0	0	1
1	0	1	0	1
x	0	0	1	1
z	1	1	1	1

casex	0	1	x	z
0	1	0	1	1
1	0	1	1	1
x	1	1	1	1
z	1	1	1	1

例5.3

```
reg[7:0] ir;
```

```
casez(ir)
```

```
    8 'b1??????: instruction1(ir);
```

```
    8 'b01?????: instruction2(ir);
```

```
    8 'b00010???: instruction3(ir);
```

```
    8 'b000001??: instruction4(ir);
```

```
endcase
```

```
module test_casex;  
reg[1:0] select;  
  
always @(select)  
begin  
  
    casex (select)  
        2'b01:$display("%t 2b01...", $time);  
        2'b10:$display("%t 2b10...", $time);  
  
        2'b00:$display("%t 2b00", $time);  
        2'b11:$display("%t 2b11", $time);  
        default:$display("%t default...", $time);  
    endcase  
end
```



```
initial
begin
    select=2' b00;
    #100;
    select=2' b01;
    #100;
    select=2' bzx;
    #100
    select=2' b1x;
end
endmodule
```

```
module mux4_to_1b(out, i0, i1, i2, i3, s1, s0);  
output[1:0] out;  
input[1:0] i0, i1, i2, i3;  
input s1, s0;  
reg[1:0] out;  
  
always @(s1 or s0 or i0 or i1 or i2 or i3)  
begin  
    case ({s1, s0})  
        2'b00: out = i0;  
        2'b01: out = i1;  
        2'b10: out = i2;  
        2'b11: out = i3;  
        default: out = 2'bxx;  
    endcase  
end  
endmodule
```

```
module t_mux4_to_1b;
reg[1:0] i0,i1,i2,i3;
reg s1,s0;
wire[1:0] out;

mux4_to_1b u1(out,i0,i1,i2,i3,s1,s0);

initial
fork
    i0=2'b00;
    i1=2'b01;
    i2=2'b10;
    i3=2'b11;
    s0=1'b0;
    s1=1'b0;
```

```
#100 s0=1' b1;  
#100 s1=1' b0;  
#200 s0=1' b0;  
#200 s1=1' b1;  
#300 s0=1' b1;  
#300 s1=1' b1;  
#500 s0=1' bz;  
#500 s1=1' bx;  
join  
endmodule
```

Verilog HDL设计中容易犯的一个通病是由于不正确使用语言，生成了并不想要的锁存器。

```
always @ (a1 or d ) // 有锁存器
begin
    if ( a1 )  q = d;
end
```

```
always @ (a1 or d ) // 无锁存器
begin
    if ( a1 )  q = d;
    else  q = 0;
end
```


//有锁存器

```
always @(sel[1:0] or a or b)
  case(sel[1:0])
    2'b00:q<=a;
    2'b11:q<=b;
  endcase
```

//无锁存器

```
always @(sel[1:0] or a or b)
  case(sel[1:0])
    2'b00:q<=a;
    2'b11:q<=b;
    default:q<= 'b0;
  endcase
```

5.5 循环语句

循环语句

在Verilog HDL中存在着四种类型的循环语句，用来控制执行语句的执行次数。

forever

repeat

while

for

forever

```
reg clock;  
initial  
begin  
    clock=1' b0;  
    forever #10 clock=~clock;  
end
```

产生周期为**20**个单位时间的时钟信号

repeat语句

repeat语句的格式如下:

repeat (表达式) 语句;

或

repeat (表达式) begin 多条语句; end

实现乘法器

```
module test_repeat;  
parameter size=8, longsize=16;  
reg [size:1] opa, opb;  
reg [longsize:1] result;
```

```
initial
```

```
begin
```

```
    opa=8' ha2;
```

```
    opb=8' h3f;
```

```
    #100;
```

```
    opa=8' hf5;
```

```
    opb=8' hc6;
```

```
end
```



```
always @(opa or opb)
begin: mult
    reg [longsize:1] shift_opa, shift_opb;
    shift_opa = opa;
    shift_opb = opb;
    result = 0;
    repeat(size)
    begin
        if(shift_opb[1])
            result = result + shift_opa;
        shift_opa = shift_opa <<1;
        shift_opb = shift_opb >>1;
    end
end
endmodule
```

while 语句

while语句的格式如下:

while(表达式) 语句;

或用如下格式:

while(表达式) begin 多条语句; end

对rega中值为1的位进行计数

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

for语句

初始化memory型变量

```
begin:      init_mem
```

```
reg[7:0] tempi;
```

```
    for(tempi=0;tempi<memsize;tempi=tempi+1)
```

```
        memory[tempi]=0;
```

```
end
```

```
parameter size = 8, longsize = 16;  
reg[size:1] opa, opb;  
reg[longsize:1] result;
```

```
begin:mult  
    integer bindex;  
    result=0;  
    for( bindex=1; bindex<=size; bindex=bindex+1 )  
        if(opb[bindex])  
            result = result + (opa<<(bindex-1));  
end
```


对rega这个八位二进制数中值为1的位进行计数的另一种方法。

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

5.6 命名块的禁用

Verilog通过关键字**disable**提供了一种终止命名块执行的方法。可用来从循环中退出、处理错误条件以及根据控制信号来控制某些代码段是否被执行。

对块语句的禁用导致紧接在块后面的那条语句被执行。

//从矢量标志寄存器的低有效位开始查找第一个值为1的位

```
module ex5_13;  
reg [15:0] flag;  
integer i; //用于计数的整数  
  
initial  
begin  
    flag = 16'b 0010_0000_0000_0000;  
    i = 0;
```

```
begin: block1    //while循环声明中的主模块是命名块block1
    while(i < 16)
    begin
        if (flag[i])
        begin
            $display("Encountered a TRUE bit at element
                        number %d", i);
            disable block1; // 在标志寄存器中找到了值为真
                            // (1) 的位, 禁用block1
        end
        i = i + 1;
    end
end
end
endmodule
```

5.7 生成块

生成语句可以动态地生成Verilog代码。使用生成语句能够大大简化程序的编写过程。

Verilog中有3种创建生成语句的方法。

循环生成

条件生成

case生成

循环生成语句

例 对两个N位总线变量进行按位异或。

```
module bitwise_xor (out, i0, i1) ;  
//参数声明语句。参数可以重新定义  
parameter  N=32; //缺省的总线位宽为32位
```

```
//端口声明语句  
output[ N-1:0]  out;  
input[N-1:0]  i0, i1;
```

```
//声明一个临时循环变量。  
//该变量只用于生成块的循环计算。  
//Verilog仿真时该变量在设计中并不存在  
genvar  j  ;
```

//用一个单循环生成按位异或的异或门 (xor)

generate

for(j=0; j<N; j=j+1)

begin : xor_loop

xor g1(out[j], i0[j], i1[j]) ;

end // 在生成块内部结束循环

endgenerate //结束生成块

endmodule

```
module bitwise_xor2(out, i0, i1) ;  
parameter  N=32;  //缺省的总线位宽为32位  
  
output[N-1:0]  out ;  
input[N-1:0]   i0, i1 ;  
  
genvar  j ; //声明一个临时循环变量  
  
//异或门可以用always块来替代  
reg [N-1:0]  out ;  
generate  
    for (j =0; j<N;j =j+1)  
        begin : bit  
            always @(i0[j] or i1[j]) out[j]=i0[j]^i1[j] ;  
        end  
    endgenerate  
endmodule
```

循环生成语句若干特点

(1) 在仿真开始之前，仿真器会对生成块中的代码进行确立（展平），将生成块转换为展开代码，然后对展开的代码进行仿真。因此，生成块的本质是使用循环内的一条语句来代替多条重复的Verilog语句，简化用户的编程。

(2) 关键词genvar用于声明生成变量，生成变量只能用在生成块中，在确立后的仿真码中，生成变量是不存在的。

(3) 一个生成变量的值只能由循环生成语句来改变。

(4) 循环生成语句可以嵌套使用，不过使用同一个生成变量作为索引的循环生成语句不能相互嵌套。

(5) xor_loop是赋予循环生成语句的名字，目的在于通过它对循环生成语句之中的变量进行层次化引用。因此，循环生成语句中各个异或门的相对层次名为：xor_loop[0].g1, xor_loop[1].g1, ..., xor_loop[31].g1


```
//本模块生成一个门级脉动加法器
module ripple_adder(co, sum, a0, a1, ci) ;
//参数声明语句，参数可以重新定义。
parameter  N = 4 ; // 缺省的总线位宽为4

//端口声明语句
output [ N-1 : 0 ] sum ;
output  co ;
input [N-1 : 0 ]  a0 ,  a1 ;
input ci ;

wire [N: 0 ]  carry ;////////////////////////////////////此处修改

    assign carry [0] = ci ;
```



```
genvar i ;  
//用一个单循环生成按位异或门等逻辑  
generate  
  for ( i = 0 ; i < N ; i = i + 1 )  
  begin : r_loop  
    wire t1 , t2 , t3 ;  
    xor g1 ( t1 , a0[ i ] , a1 [ i ] ) ;  
    xor g2 ( sum [ i ] , t1 , carry [ i ] ) ;  
    and g3 ( t2 , a0[ i ] , a1 [ i ] ) ;  
    and g4 ( t3 , t1 , carry [ i ] ) ;  
    or g5 (carry [ i + 1 ] , t2 , t3 ) ;  
  end // 生成块内部循环的结束  
endgenerate //生成块的结束  
  
assign co = carry [ N ] ;  
endmodule
```

```
xor:r_loop[0]. g1, r_loop[1]. g1, r_loop[2]. g1, r_loop[3]. g1,  
    r_loop[0]. g2, r_loop[1]. g2, r_loop[2]. g2, r_loop[3]. g2;
```

```
and:r_loop[0]. g3, r_loop[1]. g3, r_loop[2]. g3, r_loop[3]. g3,  
    r_loop[0]. g4, r_loop[1]. g4, r_loop[2]. g4, r_loop[3]. g4;
```

```
or:r_loop[0]. g5, r_loop[1]. g5, r_loop[2]. g5, r_loop[3]. g5
```

上面生成的实例用下面这些生成的线网连接起来

```
Nets : r_loop[0]. t1 , r_loop[0]. t2 , r_loop[0]. t3  
r_loop[1]. t1 , r_loop[1]. t2 , r_loop[1]. t3  
r_loop[2]. t1 , r_loop[2]. t2 , r_loop[2]. t3  
r_loop[3]. t1 , r_loop[3]. t2 , r_loop[3]. t3
```

条件生成语句

```
// 本模块实现一个参数化乘法器
module multiplier (product, a0, a1) ;
//参数声明，该参数可以重新定义
parameter  a0_width = 8 ;
parameter  a1_width = 8 ;

//本地参数声明
//本地参数不能用参数重新定义（defparam）修改，
//也不能在实例引用时通过传递参数语句，即 #（参数1，参数2，...）的方法修改
localparam  product_width = a0_width + a1_width ;
```

```
//端口声明语句
output [ product_width - 1 : 0 ] product ;
input [ a0_width - 1 : 0 ] a0 ;
input [ a1_width - 1 : 0 ] a1 ;
//有条件地调用（实例引用）不同类型的乘法器
//根据参数a0_width 和 a1_width的值，在调用时
//引用相对应的乘法器实例。
generate
    if ((a0_width<8) || (a1_width<8))
        cal_multiplier #(a0_width,a1_width)
                        m0(product,a0,a1);
    else
        tree_multiplier #(a0_width,a1_width)
                        m0(product,a0,a1) ;
endgenerate //生成块的结束
endmodule
```



```
module cal_multiplier(product, a0, a1) ;  
parameter  a0_width = 7 ;  
parameter  a1_width = 7 ;  
localparam product_width = a0_width + a1_width ;  
output [ product_width - 1 : 0 ]  product ;  
input  [ a0_width - 1 : 0 ]  a0 ;  
input  [ a1_width - 1 : 0 ]  a1 ;  
  
reg[product_width-1:0]  product ;  
  
    always @(a0 or a1)  
    begin  
        product=a0*a1;  
        $display("This is cal_multiplier...");  
    end  
  
endmodule
```



```
module tree_multiplier ( product , a0 , a1 ) ;  
parameter  a0_width = 8 ;  
parameter  a1_width = 8 ;  
localparam product_width = a0_width + a1_width ;  
output [ product_width - 1 : 0 ]  product ;  
input  [ a0_width - 1 : 0 ]  a0 ;  
input  [ a1_width - 1 : 0 ]  a1 ;  
  
reg[product_width-1:0]  product ;  
  
    always @(a0 or a1)  
    begin  
        product=a0*a1;  
        $display("This is tree_multiplier...");  
    end  
  
endmodule
```

```
module test_multiplier;  
  wire[7:0] product;  
  reg[3:0] a0, a1;
```

```
  initial  
  begin  
    a0=4' ha;  
    a1=4' hb;  
    #100;  
    a0=4' h2;  
    a1=4' h3;  
  end
```

```
  multiplier #(4,4) u1( product , a0 , a1 );  
endmodule
```

```
module test_multiplier2;
parameter a0_w=8,a1_w=8;
localparam p_m=a0_w+a1_w;
  wire[p_m-1:0] product;
  reg[a0_w-1:0] a0;
  reg[a1_w-1:0] a1;
  initial
  begin
    a0=8'h0a;
    a1=8'h0b;
    #100;
    a0=8'h20;
    a1=8'h30;
  end
```

```
  multiplier #(a0_w,a1_w) u1( product , a0 , a1 );
endmodule
```

case生成语句

//本模块生成N位的加法器

```
module adder ( co , sum , a0 , a1 , ci );
```

//参数声明，本参数可以重新定义

```
parameter  N = 4 ;    // 缺省的总线位宽为4
```

//端口声明

```
output [ N-1 : 0 ]  sum ;
```

```
output  co ;
```

```
input [ N-1 : 0 ]  a0 , a1 ;
```

```
input  ci ;
```

```
// 根据总线的位宽，调用（实例引用）相应的加法器
//参数N在调用（实例引用）时可以重新定义，调用（实例引用）
// 不同位宽的加法器是根据不同的N来决定的。
generate
    case ( N )
        // 当N=1，或2 时分别选用位宽为1位或2位的加法器
        1:adder_1bit adder1(co, sum, a0, a1, ci); // 1位的加法器
        2:adder_2bit adder2(co, sum, a0, a1, ci); // 2位的加法器
        // 缺省的情况下选用位宽为N位的超前进位加法器
        default:adder_cla #(N) adder3(co, sum, a0, a1, ci);
    endcase
endgenerate //生成块的结束

endmodule
```