Verilog 中 Generate 语句的使用

WARNING: for and while loop statements are valid in Verilog (and can be simulated), but cannot always be mapped to hardware. Generate statements are the appropriate use for for loops.

设计中有时需要编写很多结构相同但是参数不同的赋值语句或者逻辑语句。C 语言中常用 for 语句来解决此类问题，Verilog-2001 则为我们提供了 generate 语句。generate 语句有 generate_for、generate_if、generate_case 三种语句。

generate 语句格式如下：

```
genvar 循环变量名;
generate
    // generate 循环语句
    // generate 条件语句
    // generate 分支语句
    // 嵌套的 generate 语句
endgenerate
```

1、generate_for 语句

（1）必须使用 genvar 申明一个正整数变量，用作 for 循环的判断。

（2）需要复制的语句必须写到 begin_end 语句里面，即使只有一句！

（3）begin_end 需要有一个类似于模块名的名字。

下面举例进行说明：

例 1:

```
module buffer_1 (
    input    wire    in,
    output   wire    out
);
assign out = ~in;
endmodule

//8bit width buffer
module buffer_8 (
    input    wire [7:0] din,
    output   wire [7:0] dout
);

//Generate block
genvar i;
generate
    for(i = 0; i < 8; i = i + 1) begin: BLOCK
        buffer_1 buffer_1_inst (.in(din[i]), .out(dout[i]));
    end
```

```
endgenerate

endmodule
```

在 buffer_8 中例化 buffer_1 8 次，这里有几点需要注意：

① 循环变量 i 必须是 genvar 类型的，不可以是 reg 型，integer 型；

② for 循环之后的的 begin 最好加上一个标号（如 BLOCK1，例化模块的名字）；

例 2:

```
module nbit_xor #(
    parameter SIZE=16
)(
    input(SIZE-1:0) a,b,output[SIZE-1:0] y
);
   genvar gv_i;
   generate
      for(gv_i=0;gv_i<SIZE;gv_i++)
      begin:sblka
        xor uxor(y[gv_i],a[gv_i],b[gv_i]);
      end
 endgenerate
endmodule
```

for 循环中使用的循环变量 gv_i 被称为 genvar 变量，这种变量必须用 genvar 来声明，并且只能在 generate 循环语句中使用；此外，generate 块需要标签，用来表示循环的实例化名称，在上例中是 sblka.

```
1   `timescale 1 ns /1 ps
2
3   module nbit_xor
4   #(parameter SIZE = 16)
5   (
6   input  [SIZE-1 : 0] a,
7   input  [SIZE-1 : 0] b,
8   output [SIZE-1 : 0] y
9   );
10
11  genvar gv_i; // 这这类型的变量只能在generate循环语句中使用
12
13  generate
14      for (gv_i = 0 ; gv_i < SIZE ; gv_i = gv_i +1)
15      begin : label
16          // label用来表示generate循环的实例名称
17          xor u_xor(y[gv_i] , a[gv_i] , b[gv_i]);
18          // 实例化后的结果如下：
19          // label[0].u_xor (y[0] , a[0] , b[0]);
20          // label[1].u_xor (y[1] , a[1] , b[1]);
21          // label[2].u_xor (y[2] , a[2] , b[2]);
22          // ... ...
23          // label[SIZE-1].u_xor (y[SIZE-1] , a[SIZE-1] , b[SIZE-1]);
24          // 实例化后的层次路径如下：
25          // nbit_xor.label[0].u_xor;
26          // ... ...
27          // 同理，还可以引用别的已经定义的module在generate语句中实例化
28      end
29  endgenerate
30
31  endmodule
```

例 3:

```
module test (bin, gray);
    parameter SIZE=8;
    output [SIZE-1:0] bin;
    input [SIZE-1:0] gray;
    genvar i; //genvar i;也可以定义到 generate 语句里面
    generate
        for(i=0;i<SIZE;i=i+1)
        begin: bit
            assign bin[i]=^gray[SIZE-1:i];
        end
    endgenerate
endmodule
```

等同于下面语句：

```
assign bin[0]=^gray[SIZE-1:0];
assign bin[1]=^gray[SIZE-1:1];
```

```
assign bin[2]=^gray[SIZE-1:2];
assign bin[3]=^gray[SIZE-1:3];
assign bin[4]=^gray[SIZE-1:4];
assign bin[5]=^gray[SIZE-1:5];
assign bin[6]=^gray[SIZE-1:6];
assign bin[7]=^gray[SIZE-1:7];
```

例 4:

```
generate
    genvar i;
    for(i=0;i<SIZE;i=i+1)
    begin:shifter
        always@(posedge clk)
            shifter[i]<=(i==0)?din:shifter[i-1];
    end
endgenerate
```

相当于:

```
always@(posedge clk)
    shifter[0]<=din;
always@(posedge clk)
    shifter[1]<=shifter[0];
always@(posedge clk)
    shifter[2]<=shifter[1];
.....
    .....
always@(posedge clk)
    shifter[SIZE]<=shifter[SIZE-1];
```

2. generate-conditional 条件语句

generate 条件语句最常见的格式如下：

```
if(condition)
  statements
else
  statements
```

condition 必须是一个静态的条件，即在细化（Elaborated）期间计算得出。statements 可以是任何能够在模块中出现的语句，例如 always 语句。注意，由于条件的值可能取决于从上层模块中传递过来的参数，因此条件的值可能不能再细化期间被完全算

出。

例 1：

```
module adder
 #(parameter SIZE=4)
 (input[SIZE-1:0] a,b,
  output[SIZE-1:0] sum,
  output carry_out);
  wire [SIZE-1:0] carry;

  genvar gv_k;
  generate
    for(gv_k = 0; gv_k < SIZE; gv_k ++)
      begin: gen_blk_adder
        if(gv_k == 0)
          half_adder u_ha (
                              .a(a[gv_k]),
              .b(b[gv_k]),
                .sum(sum[gv_k]),
                .carry_out(carry[gv_k]),
           );
        else
          full_adder u_ha (
                              .a(a[gv_k]),
              .b(b[gv_k]),
                .sum(sum[gv_k]),
                .carry_in(carry[gv_k-1]),
                .carry_out(carry[gv_k]),
           );
      end
  endgenerate
endmodule
```

例 2：

```
`timescale 1 ns /1 ps

module shift_register
#(parameter BITS = 8)
(
input  shift_in,
input  clk,
input  resetn,
output shift_out
);

wire [BITS-1 : 0] tq;

genvar gv_i; // 这种类型的变量只能在generate循环语句中使用
// generate-conditional语句实质上是将条件语句放入到generate-for结构中
/* 在该条件语句结构中，if语句的条件必须是静态的，这样的条件表达式
   必须是由常数或者参数组成，也意味着在程序运行期间保持不变*/
generate
    for (gv_i = 0 ; gv_i < BITS ; gv_i = gv_i +1)
    begin : label
        if (gv_i == BITS-1)      // 此条件成立时，u0dff
        begin
            dflip_flop u0dff (.d(serial_in) , .resetn(resetn) , .ck(clk) , .q(tq[gv_i]));
        end
        else
        begin
            if (gv_i == 0)       // 此条件成立时，u1dff
            begin
                dflip_flop u1dff (.d(tq[gv_i + 1]) , .resetn(resetn) , .ck(clk) , .q(serial_out));
            end
            else                 // 其他情况下，u2dff
            begin
                dflip_flop u2dff (.d(tq[gv_i + 1]) , .resetn(resetn) , .ck(clk) , .q(tq[gv_i]));
            end
        end
    end
endgenerate

endmodule
```

3．generate-case 分支语句

generate 分支语句与条件语句类似，只不过分支语句是用分支来进行条件选择。

例 1：

```
module adder #(
    parameter SIZE=4
  parameter IMPLEMENTATION_LEVEL=0
)(
    input[SIZE-1:0] arg1,arg2,
  output[SIZE-1:0] result,
);
  generate
    case(IMPLEMENTATION_LEVEL)
       0: assign result=arg1+arg2;
       1:.....;
       2:.....;
       3:.....;
       default:......;
  endgenerate
endmodule
```

例 2：

```
1   `timescale 1 ns /1 ps
2
3   module adder
4   #(parameter N =4)
5   (
6   input            ci;
7   input  [N-1 : 0] a0 , a1;
8   output           co;
9   output [N-1 : 0] sum
10  );
11
12  // 注意：未定义genvar变量
13  // 根据总线的位宽，调用相应的加法器
14  // 参数N在调用时可以重新定义，但是该值也是静态的，因为每次调用时N都是必须确定的
15  // 调用不同位宽的加法器是根据不同的N来决定的
16  generate
17  case(N)
18      // 当N=1或者2时分别选用位宽为1位或2位的加法器
19      1 : adder_1bit adder1(co , sum , a0 , a1 , ci);  // 1位的加法器
20      2 : adder_2bit adder2(co , sum , a0 , a1 , ci);  // 2位的加法器
21      // 默认的情况下选用位宽为N位的加法器
22      default : adder_cla #(N) adder3(co , sum , a0 , a1 , ci);
23  endgenerate
24
25  endmodule
```

转载自： http://xilinx.eetrend.com/blog/2019/100045571.html
这篇帖子也介绍的很好：
https://www.cnblogs.com/nanoty/archive/2012/11/13/2768933.html